Objektově orientovaný návrh

Cíle objektově orientovaného návrhu

Cílem objektově orientovaného návrhu je vytvoření kvalitního software, který má následující vlastnosti:

- **robustnost** – cílem je vytvořit software, který je schopen správně reagovat i na neočekávané vstupy, které nejsou explicitně definovány pro danou aplikaci

- **Přizpůsobivost** (adaptability) – schopnost software běžet s minimálními úpravami i na jiné platformě (Windows, Unix, MacOs)

- **Znovupoužitelnost** (reusability) – cílem je vytvořit sw nebo jeho část tak, aby byla opětovně použitelná v jiných systémech. Vytváření programů se pak podobá stavebnici, kdy z hotových dílů vytváříme funkční celek

Principy objektově orientovaného návrhu

Mezi nejdůležitější principy objektově orientovaného návrhu, které vedou ke splnění uvedených cílů patří:

- **Abstrakce**
- **Zapouzdření (encapsulation)**
- **Modularita**

Abstrakce – cílem je rozdělit složitý problém na základní části, tyto části popsat a implementovat. Příkladem abstrakce mohou být **abstraktní datové typy** (ADT) – matematický model datových struktur, který specifikuje v jakém datovém typu budou uložena data a jaké operace lze nad těmito daty provádět. ADT v Javě = třída

Zapouzdření – cílem je ukrýt implementační detaily které nejsou pro uživatele podstatné a poskytnout uživateli určité rozhraní pomocí kterého může přistupovat k datovým strukturám. Chrání datové struktury před neoprávněnou manipulací

- **Modularita** znamená rozdělení softwarového systému do oddělených funkčních jednotek (modulů). Moduly mohou být navrženy tak aby byly využitelné v jiných systémech (princip znovupoužitelnosti).

Při návrhu složitého software obvykle skládáme celek z jednodušších komponent. Pro „lepší“ využití jednotlivých komponent využívá objektově orientovaný návrh principů **dědičnosti** a **polymorfizmu**

- **Dědičnost** - umožňuje vytvořit tzv. generické třídy, ze kterých je možné odvozovat třídy další přidávat do nich nové metody, popř. modifikovat metody existující. Dědičnost se využívá zejména proto abychom se vyhnuli nadbytečnému programovému kódu (popř. nadbytečným datovým strukturám)
- **Polymorfismus** (mnohotvarost) – umožňuje, aby stejné metody vykonávaly různou činnost v závislosti na objektu nad kterým pracují

Psaní programů v Javě (popř. jiných programovacích jazycích)

3 základní kroky :

1. **Návrh**
2. **Implementace a kódování**
3. **Testování, ladění, optimalizace**

Návrh:

nejdůležitější krok celého procesu vytváření software.

Cílem je vytvořit třídy a objekty. **ale jak ????**

- Jak identifikovat **třídy**?
- jak určit, jaké **atributy** mají mít?
- jak určit, jaké **metody** mají mít?
- jak tyto skutečnosti zachytit? - jedním ze způsobů UML diagramy

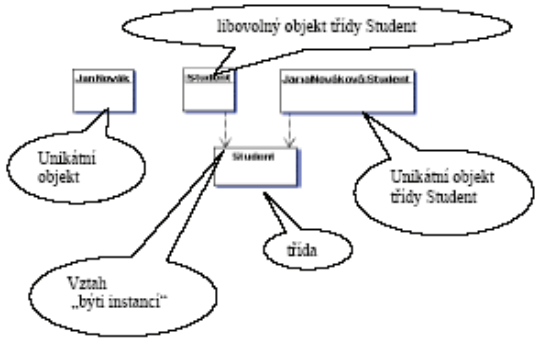
Jazyk UML, UML diagramy

- UML – unifikovaný modelovací jazyk
- Univerzální standard pro záznam, konstrukci, vizualizaci a dokumentaci artefaktů systému s převážně softwarovou charakteristikou
- Vznikl jako prostředek pro objektově-orientovanou analýzu a návrh (G. Booche, J. Rumbaugh, I. Jacobson)
- Skládá se z grafických prvků, které se dají vzájemně kombinovat do podoby diagramů. Účelem diagramů je vytvořit určité pohledy na navrhovaný systém.
  - Skupina pohledů se nazývá model
  - Model jazyka UML popisuje co má systém dělat, ale neříká jak to implementovat
- Definice UML obsahuje 4 základní části
  - Definice notace UML (syntaxe)
  - Metamodel UML (sémantika)
  - Jazyk OCL (Object Constrain Language) pro popis dalších vlastností modelu
  - Specifikace převodu do výměnných formátů (CORBA, IDL, XML, DTD)

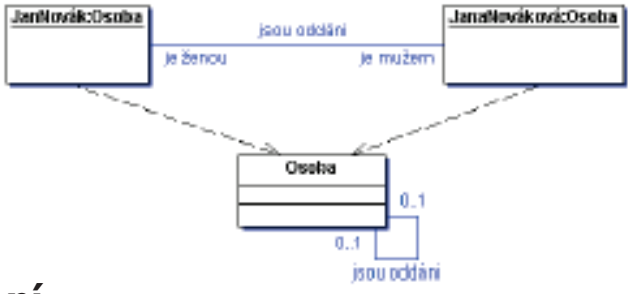
UML zahrnuje definici 8-mi typů diagramů, tj. 8 různých pohledů na model systému:

- **diagramy tříd a objektů** –popisují statickou strukturu systému, znázorňují datový model datový model systému od konceptuální úrovně až po implementaci
- **modely jednání** (diagramy případů užití) dokumentují možné případy použití systému události, na které musí systém reagovat,
- **scénáře činností** (diagramy posloupností) – popisují scénář průběhu určité činnosti systému,
- **diagramy spolupráce** – zachycují komunikaci spolupracujících objektů,
- **stavové diagramy** – popisují dynamické chování objektu nebo systému,
- **diagramy aktivit** – popisují průběh aktivit procesu nebo činností
- **diagramy komponent** – popisují rozdělení výsledného systému na funkční celky a definují náplň jednotlivých komponent,
- **diagramy nasazení** – popisují umístění funkčních celků (komponent) na výpočetní uzly informačního systému

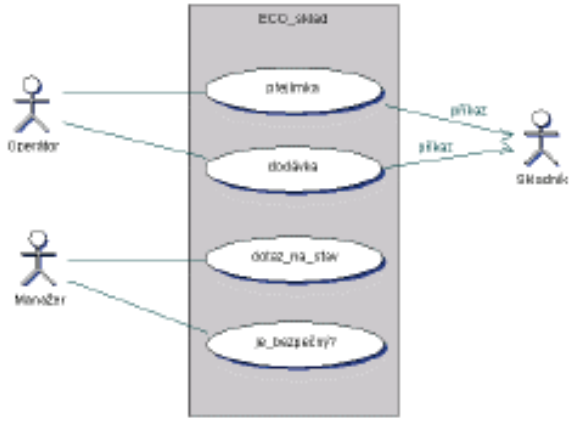
- **Diagram objektů**



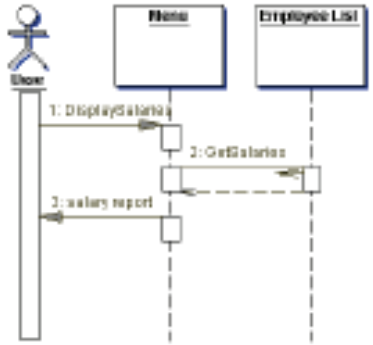
- **Diagram tříd**



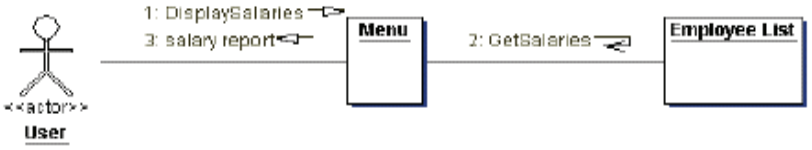
- **Model jednání**



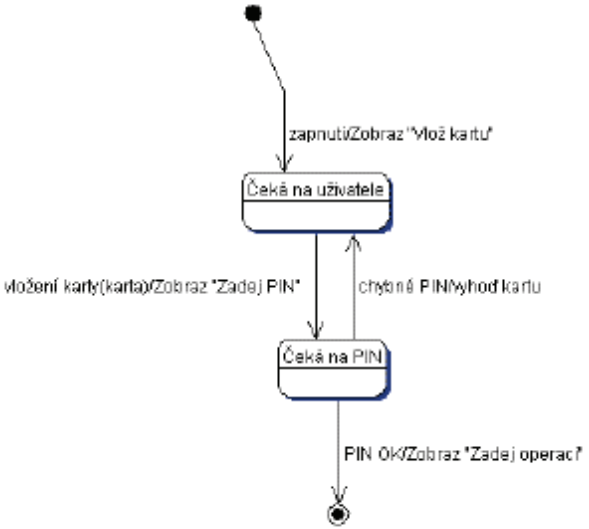
- **Scénáře činností**



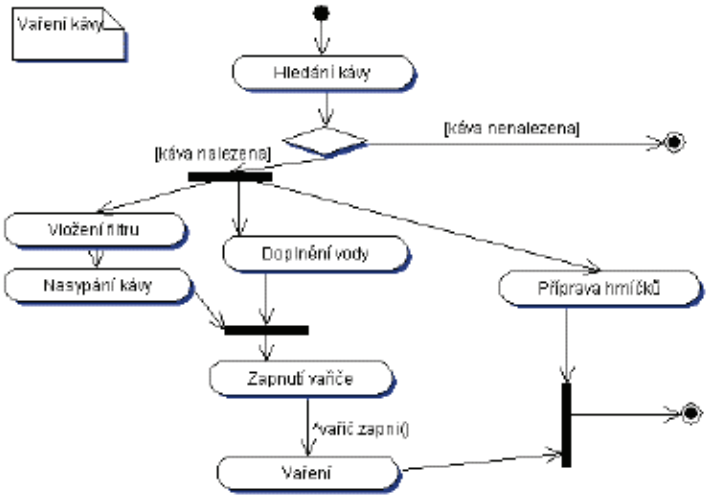
- **Diaramy spolupráce**



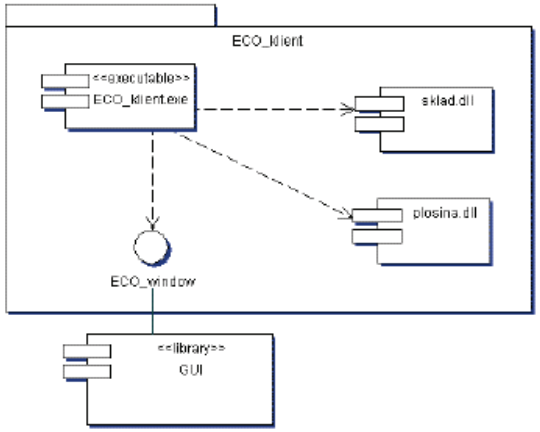
Stavový diagram



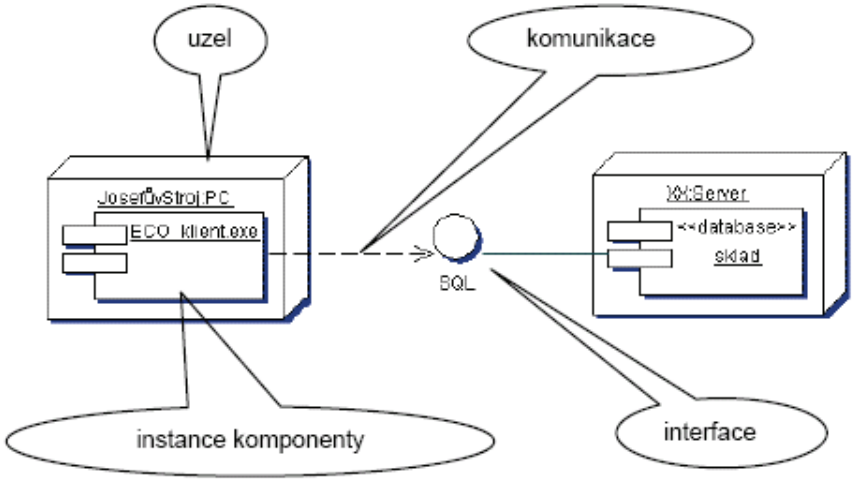
- **Diagram aktivit**



- **Diagram komponent**



- **Diagram nasazení**



Kódování:

Jakmile máme vytvořen návrh tříd je možné zapsat třídy v programovacím jazyce Java – buďto pomocí textového editoru nebo použitím integrovaného vývojového prostředí (JBuilder, JCreator, eclipse atd.)

Při zápisu programu dodržovat následující doporučení :

- používat smysluplné názvy identifikátorů
- jména tříd začínat s velkým písmenem (**Datum**,**Vektor**, atd.)
- jména metod a proměnných začínat s malým písmenem (**vlozPolozku(), jmenoStudenta,** atd.)
- pokud má proměnná charakter konstanty definovat jí jako konstantu (**final**) a název identifikátoru konstanty psát s velkými písmeny (**MAX_HODNOTA, MAX_POCET_PRVKU**)
- odsazovat příkazové bloky
- v každé třídě dodržovat následující pořadí deklarací
  - **Konstantní atributy třídy, tj. statické konstanty**
  - **Ostatní atributy třídy**
  - **Konstantní atributy instancí**
  - **Ostatní atributy instancí**
  - **Přístupové metody atributů třídy**
  - **Ostatní metody třídy**
  - **Konstruktory a metody vracející odkaz na instance třídy**
  - **Přístupové metody atributů instancí**
  - **Ostatní metody instancí**
  - **Testovací metody**
- používat komentáře zvlášť případech kde se vyskytují nejednoznačnosti, „podivné“ konstrukce atd. Komentáře psát nejlépe ve stylu javadoc

Testování a ladění:

- **Ladění**
  - nejjednodušší způsob – kontrolní výpisy použitím System.out.print(<String>)
  - použití speciálního programu - debuggeru - standardní jdk obsahuje základní řádkově orientovaný debugger **jdb**, vývojová prostředí (eclipse, JBuilder) obsahují debuggery s GUI (grafické uživatelské rozhraní)

- **Testování**

Testování a ladění je obvykle časově nejnáročnější činnost. Cílem je ověřit správnost programu a použitých metod popř. odstranit chyby které se během zápisu algoritmu vyskytly .

Testování provádíme na reprezentativní množině vstupů – tu je nutné zvolit tak, aby byla každá metoda třídy alespoň jednou zavolána. Totéž by mělo platit o každém příkazu v programu.

Program často „padá“ v důsledku neočekávaných vstupních hodnot -> obvykle se doporučuje nechat program proběhnou na **rozsáhlé množině náhodně vygenerovaných dat**

Příklad: pro otestování metody řazení pole celočíselných prvků volíme obvykle následující vstupy:

- pole nulové délky (neobsahující žádný vstup)
- pole s jedním prvkem
- pole s prvky stejné hodnoty
- seřazené pole
- pole seřazené v opačném pořadí

Komentáře a dokumentace

Chceme-li aby se program dožil vysokého produktivního věku musíme jej psát tak, aby bylo možno snadno upravit a doplnit o další funkce tj. psát program čitelně.

Čitelnost ovlivňují:

- **Správně volené identifikátory**
- **Komentáře vysvětlující náročnější obraty**

Druhy komentářů v Javě:

- Řádkový komentář začíná dvojicí znaků // a končí koncem řádku - poznámky ke kódu
- Obecný komentář /*…*/
- Dokumentační komentář – speciální typ obecného komentáře – /** … */ - je zpracováván programem javadoc.exe , který vytváří html dokumentaci

Dokumentační komentáře se zapisují těsně před dokumentovaný prvek (javadoc reaguje i na umístění komentářů)

- Komentáře popisující účel a použití třídy – patří před hlavičku třídy
- Komentář popisující účel atributu – patří před deklaraci tohoto atributu
- Komentáře popisující funkci nějaké metody a význam parametru – patří před hlavičku této metody

Pomocné značky pro javadoc:

@author – vkládá se do místa dokumentace k celé třídě. Zapisuje se za mí jméno autora třídy, více autorů je možné psát odděleně nebo za společnou značku

@version – vkládá se do místa dokumentace pro celou třídu a zapisuje se za ní číslo verze. Pro formát verze není žádný předpis

@param – používá se v místě dokumentace konstruktorů, za ní se uvádí přesný název parametru z hlavičky a za něj popis parametru

@returns – používá se v dokumentaci metod, které vracejí nějakou hodnotu, za ní se uvádí podrobný popis návratové hodnoty

@throw – používá se k popisu výjimky a důvodu proč jí metoda „vyhazuje“

Dědičnost, polymorfismus, interface, práce se soubory

Dědičnost

- dovoluje vybudování hierarchie tříd, které se postupně z generace na generaci rozšiřují
- používá se v případech, kdy se chceme vyhnout opakování kódu

Dědění je význačný nástroj pro vytváření opakovaně využitelných programových modulů.

Programový modul by měl být zároveň uzavřený a otevřený.

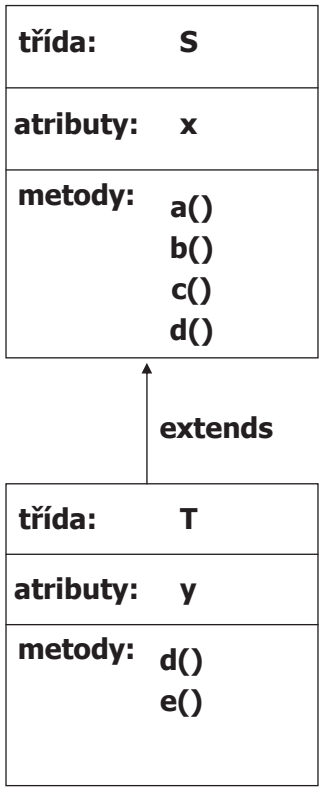
- **uzavřený** – pro jeho použití není potřeba nic přidávat, uživatel není oprávněn modul modifikovat
- **otevřený** – uživatel by měl mít možnost nevhodné věci modifikovat a nové přidávat

Dědění umožňuje v odvozené třídě:

- vše co bylo dobré v základní třídě, ponechat i v odvozené třídě
- vše co nám chybělo, jednoduše dodat
- vše co se nám nelíbilo, změnit

Realizace dědičnosti v Javě:

- musí existovat třída, která se stane rodičem (bázová třída, superclass), jméno této třídy se uvede v hlavičce třídy za klíčovým slovem **extends**
- je povolena pouze jednouchá dědičnost
- vícenásobná dědičnost pouze přes interface



- Objekt třídy **T** obsahuje:
  - atributy **x** a **y**
  - metody a(), b(), c(), d(), e()
  - metoda d(), překrývá metodu d() rodičovské třídy

Změnit vlastnosti metod v odvozené třídě je možné dvojím způsobem:

- **přetížením** (overloading) – použijeme stejné jméno metody, ale jiné parametry, popř. návratový typ
- **překrytím** (overriding, zasínění – hiding) – hlavička metody je identická s hlavičkou metody rodiče, ale metoda může dělat něco jiného

- Při využití dědičnosti pozor na konstruktor(y) potomka !!!! Během konstrukce musí být vždy umožněno volat konstruktor rodiče. Mohou nastat 2 případy:
  - Rodič má konstruktor bez parametrů nebo implicitní - potomek může mít konstruktor implicitní a nemusí být volán konstruktor rodiče
  - Rodič má konstruktor alespoň s jedním parametrem – konstruktor potomka **musí !!!** existovat a prvním příkazem musí být volání konstruktoru rodiče (příkaz super())

```
class Rodic {
  public int i;
  public Rodic(int parI) { i = parI; }
//  public Rodic() { i = 5; }
}

public class Potomek extends Rodic {
  public Potomek() {
    super(8);
  }
  public static void main(String[] args) {
    Potomek pot = new Potomek();
  }
}
```

- Finální metody třídy - používají se tehdy, pokud považujeme metodu za dokonalou a nechceme aby byla ve zděděných třídách překryta.

  Pozor !!! Tato metoda může být přetížena

```
class Rodic {
  public int i;
  public Rodic() { i = 1; }
  final int getI() { return i; }     // koncová metoda třídy  nelze překrýt
}

public class Potomek extends Rodic {
  //  int getI() { return i * 2; }  // chyba

  public static void main(String[] args) {
    Potomek pot = new Potomek();
    System.out.println("Hodnota je: " + pot.getI());
  }
}
```

- Abstraktní třídy – chceme-li aby byla metoda určitě překryta doplníme ji v rodičovské třídě klíčovým slovem *abstract*
  - Jakmile je jako abstract označena jedna z metod třídy musí být použito abstract i u třídy = abstraktní třída.
  - Ve zděděné třídě se musí přeprogramovat všechny metody označené jako abstraktní !!!

```
abstract class Rodic {
  public int i;
  public Rodic() { i = 1; }
  abstract int getI();
  final void setI(int novel) { i = novel; }
}

public class Potomek extends Rodic {
  int getI() { return i * 2; }
  void setI() { i = 5; }   // přetížená

  public static void main(String[] args) {
//    Rodic rod = new Rodic();    // chyba
    Potomek pot = new Potomek();
    pot.setI(3);
    System.out.println("Hodnota je: " + pot.getI());
    pot.setI();   // přetížená
    System.out.println("Hodnota je: " + pot.getI());
  }
}
```

- **Finální třídy – používají se tehdy, nechceme-li, aby byla třída zděděná (např. z důvodů optimalizace třídy během překladu)**
  - **Finální třída nesmí obsahovat abstraktní metody !!!!!**

- **Překrytí proměnné – u jednoduchých datových typů diskutabilní, používá se často u referenčních proměnných (viz. Herout)**

```
public class Progression {

    protected long first;
    protected long cur;

    /** Default constructor.  */
    Progression() {
      cur = first = 0;
    }

    /** Resets the progression to the first value.

    protected long firstValue() {
      cur = first;
      return cur;
    }

    /** Advances the progression to the next value.
     */
    protected long nextValue() {
      return ++cur; // default next value
    }

    /** Prints the first n values of the progression.
    public void printProgression(int n) {
      System.out.print(firstValue());
      for (int i = 2; i <= n; i++)
        System.out.print(" " + nextValue());
      System.out.println(); // ends the line
    }
  }
```

```
class ArithProgression extends Progression {

  /** Increment. */
  protected long inc;

  // Inherits variables first and cur.

  /** Default constructor setting a unit increment. */
  ArithProgression() {
    this(1);
  }

  /** Parametric constructor providing the increment. */
  ArithProgression(long increment) {
    inc = increment;
  }

  /** Advances the progression by adding the increment to the
      current value.
   *
   * @return next value of the progression
   */
  protected long nextValue() {
    cur += inc;
    return cur;
  }

  //  Inherits methods firstValue() and printProgression(int).
}
```

```
/**
*    Geometric Progression
*/


class GeomProgression extends Progression {

      // Inherits variables first and cur.

      /** Default constructor setting base 2. */
      GeomProgression() {
        this(2);
      }

      /** Parametric constructor providing the base.
       *
       * @param base base of the progression.
       */
      GeomProgression(long base) {
        first = base;
        cur = first;
      }

      /** Advances the progression by multiplying the base with
     the current value.
       *
       * @return next value of the progression
       */
      protected long nextValue() {
        cur *= first;
        return cur;
      }

      //  Inherits methods firstValue() and printProgression(int).
    }
}
```

```
    /**
     * Fibonacci progression.
     */
    class FibonacciProgression extends Progression {
      /** Previous value. */
      long prev;
      // Inherits variables first and cur.

      /** Default constructor setting 0 and 1 as the first two values.
    */
      FibonacciProgression() {
        this(0, 1);
      }
      /** Parametric constructor providing the first and second
    values.
       *
       * @param value1 first value.
       * @param value2 second value.
       */
      FibonacciProgression(long value1, long value2) {
         first = value1;
         prev = value2 - value1; // fictitious value preceding the first
      }

      /** Advances the progression by adding the previous value to
    the current value.
       *
       * @return next value of the progression
       */
      protected long nextValue() {
        long temp = prev;
        prev = cur;
        cur += temp;
        return cur;
      }
      // Inherits methods firstValue() and printProgression(int).
    }
```

Polymorfizmus

- polymorfizmus=vícetvarost, mnohotvarost. Jedná se o možnost využívat v programovém textu stejnou syntaktickou podobu metody s různou vnitřní reprezentací (voláme stejnou metodu a ta pokaždé dělá něco jiného)
- polymorfizmus má smysl tehdy. když má nějaká třída více potomků (více typů) a my k nim přistupujeme jednotným (typově nezávislým) způsobem
- k využití polymorfizmu je výhodné použití abstraktní třídy nebo interface, kdy nadefinujeme abstraktní metody s jasně definovanými parametry a návratovým typem a donutíme programátory, aby je překryli (implementovali). Polymorfismus je ale možné využívat i u neabstraktních tříd

Příklady: využití abstraktní třídy

```
abstract class Zivocich {
  String typ;
  Zivocich(String typ) { this.typ = new String(typ); }

  public void vypisInfo() {
    System.out.print(typ + ", ");
    vypisDelku();
  }

  public abstract void vypisDelku();
}

class Ptak extends Zivocich {
  int delkaKridel;

  Ptak(String typ, int delka) {
    super(typ);
    delkaKridel = delka;
  }

  public void vypisDelku() {
    System.out.println("delka kridel: " + delkaKridel);
  }
}

class Slon extends Zivocich {
  int delkaChobotu;

  Slon(String typ, int delka) {
    super(typ);
    delkaChobotu = delka;
  }

  public void vypisDelku() {
    System.out.println("delka chobotu: " + delkaChobotu);
  }
}
```

```
class Had extends Zivocich {
  int delkaTela;

  Had(String typ, int delka) {
    super(typ);
    delkaTela = delka;
  }

  public void vypisDelku() {
    System.out.println("delka tela: " + delkaTela);
  }
}

public class PolymAbstr {
  public static void main(String[] args) {
    Zivocich[] z = new Zivocich[6];
    for (int i = 0;  i < z.length;  i++) {
      switch ((int) (1.0 + Math.random() * 3.0)) {
        case 1: z[i] = new Ptak("ptak", i); break;
        case 2: z[i] = new Slon("slon", i); break;
        case 3: z[i] = new Had("had", i); break;
      }
    }

    Zivocich t;
    for (int i = 0;  i < z.length;  i++) {
      t = z[i];      // zbytecne, staci z[i].vypisInfo();
      t.vypisInfo();
    }
  }
}
```

- **použití neabstraktních tříd**

```
class Zivocich {
  public void vypisInfo() {
    System.out.print(getClass().getName() + ", ");
  }
}

class Ptak extends Zivocich {
  int delkaKridel;

  Ptak(int delka) { delkaKridel = delka; }

  public void vypisInfo() {
    super.vypisInfo();
    System.out.println("delka kridel: " + delkaKridel);
  }
}

class Slon extends Zivocich {
  int delkaChobotu;

  Slon(int delka) { delkaChobotu = delka; }

  public void vypisInfo() {
    super.vypisInfo();
    System.out.println("delka chobotu: " + delkaChobotu);
  }
}
```

```
class Had extends Zivocich {
  int delkaTela;

  Had(int delka) { delkaTela = delka; }

  public void vypisInfo() {
    super.vypisInfo();
    System.out.println("delka tela: " + delkaTela);
  }
}

public class PolymDeden {
  public static void main(String[] args) {
    Zivocich[] z = new Zivocich[6];
    for (int i = 0;  i < z.length;  i++) {
      switch ((int) (1.0 + Math.random() * 3.0)) {
        case 1: z[i] = new Ptak(i); break;
        case 2: z[i] = new Slon(i); break;
        case 3: z[i] = new Had(i); break;
      }
    }

    for (int i = 0;  i < z.length;  i++)
      z[i].vypisInfo();
  }
}
```

- **použití rozhraní**

```
interface Vazitelny {
  public void vypisHmotnost();
}

class Clovek implements Vazitelny {
  int vaha;
  String profese;

  Clovek(String povolani, int tiha) {
    profese = new String(povolani);
    vaha = tiha;
  }

  public void vypisHmotnost() {
    System.out.println(profese + ": " + vaha);
  }

  public int getHmotnost() { return vaha; }
}

class Kufr implements Vazitelny {
  int vaha;

  Kufr(int tiha) { vaha = tiha; }

  public void vypisHmotnost() {
    System.out.println("kufr: " + vaha);
  }
}
```

```
public class PolymRozhra {
  public static void main(String[] args) {
    int vahaLidi = 0;
    Vazitelny[] kusJakoKus = new Vazitelny[3];

    kusJakoKus[0] = new Clovek("programator", 100);
    kusJakoKus[1] = new Kufr(20);
    kusJakoKus[2] = new Clovek("modelka", 51);

    System.out.println("CD - individualni pristup");
    for (int i = 0;  i < kusJakoKus.length;  i++) {
      kusJakoKus[i].vypisHmotnost();
      if (kusJakoKus[i] instanceof Clovek == true)
//        vahaLidi += kusJakoKus[i].getHmotnost();
        vahaLidi += ((Clovek) kusJakoKus[i]).getHmotnost();
    }
    System.out.println("Ziva vaha: " + vahaLidi);
  }
}
```

Soubory v Javě

- Soubor je častým nástrojem pro komunikaci programu s okolním světem. Slouží k uchování informace na energeticky nezávislém médiu (disk, páska, CD/DVD, flash disk).

- Různé operační systémy využívají různé systémy správy souborů tzv. souborové systémy (file systems = způsob organizace dat na médiích), to se mimo jiné projeví i v různém způsobu v oddělování adresářů a ve specifikaci cesty k souboru např.:
  - Windows: FAT, NTFS – používá více disků, oddělovač adresářů je znak **/**
  - Unix: UFS, Ext2 – používá jediný disk (jeden adresářový strom), používá více disků, oddělovač adresářů je znak ****

- Java je nezávislá na platformě – obsahuje nástroje pro práci se soubory v jednotlivých operačních systémech. Kromě toho java podporuje distribuovaný a vícevláknový výpočet takže informace, kterou program čte nebo zapisuje může ležet principiálně kdekoliv – v souboru na disku, na síti, v paměti. Tato informace může mít podobu znaků, skupiny bytů, objektů apod.

Základní kroky při práci se soubory:

1. **Vytvořit instanci třídy File.** Třída File slouží jako manažer souborů popř. adresářů

konstruktory třídy File :

public File(String filename)
public File(String directory,String filename)

důležité metody:

boolean createNewFile() –vytvoří soubor
boolean delete() – zruší soubor
boolean exist() – true, pokud soubor existuje
boolean isDirectory() – true, pokud je to adresář
boolean isFile() – true, pokud je to soubor
boolean mkdir() – vytvoření adresáře
ong length() –
String getName() – vrací jméno souboru/adresáře
String getParent() – vrací jméno adresáře, ve kterém je soubor umístěn, popř. jméno nadřazeného adresáře

statické proměnné: - zajišťují nezávislost na platformě OS

char File.separatorChar
String FileSeparator
char File.pathSeparatorChar
String File.pathSeparator

```
String aktDir = System.getProperty("user.dir");
File soubAbs = new File(aktDir, "a.txt");
File soubRel = new File("TMP" + File.separator + "a.txt");
File soub = new File("a.txt");
```

2. **Otevřít proud** (stream, kanál), kterým „proudí" informace do/z programu, a nastavit vlastnosti proudu (bufferování, čtení po řádcích, formátováné čtení apod.)

Proudy mohou být:

- znakově orientované – základní jednotkou dat je 16 bitový Unicode znak (abstraktní třídy Reader a Writer)
- bajtově orientované – základní jednotka dat je osmibitová (abstraktní třídy InputStream a OutputStream)

Hlavičky metod abstraktních tříd:

– **Třída Reader**

```
int read()
int read(char[] pole)
int read(char[] pole, int index, int pocet)
```

– **Třída Writer**

```
void write(int i)
void write(char[] pole)
void write(char[] pole, int index, int pocet)
void write(String retez)
void write(String retez, int index, int pocet)
```

– **Třída InputStream**

```
int read()
int read(char[] pole)
int read(char[] pole, int index, int pocet)
```

– **Třída *OutputStream***

```
void write(int i)
void write(char[] pole)
void write(char[] pole, int index, int pocet)
```

Od těchto tříd jsou odvozeny:

- **třídy pro fyzický přesun dat –** pro práci se soubory jsou to třídy:
  - FileReader, FileWriter **-** přesun znaků
  - FileInputStream, FileOutputStream **-** přesun bytů
- **třídy vlastností (filtry)**
  - BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream – využití vyrovnávací paměti
  - PrintWriter, PrintStream – formátovaný výstup
  - DataInputStream, DataOutputStream – binární čtení/zápis základních datových typů
  - ObjectInputStream, ObjectOutputStream – binární čtení/zápis libovolných objektů

Příklady práce se soubory:

- **Vstup a výstup znaků**

```
import java.io.*;

public class IoZnaky {
  public static void main(String[] args) throws
    IOException {
   File frJm = new File("a.txt");
   File fwJm = new File("b.txt");

   if (frJm.exists() == true) {
    FileReader fr = new FileReader(frJm);
    FileWriter fw = new FileWriter(fwJm);
    int c;

    while ((c = fr.read()) != -1)
     fw.write(c);

    fr.close();
    fw.close();
   }
  }
}
```

- Vstup a výstup bajtů

```
import java.io.*;

public class IoBajty {
  public static void main(String[] args) throws
     IOException {
    File frJm = new File("a.txt");
    File fwJm = new File("c.txt");

    if (frJm.exists() == true) {
      FileInputStream fr = new
     FileInputStream(frJm);
      FileOutputStream fw = new
     FileOutputStream(fwJm);

      int c;

      while ((c = fr.read()) != -1)
        fw.write(c);

      fr.close();
      fw.close();
    }
  }
}
```

- čtení po řádcích - využívá vyrovnávací paměť

```
import java.io.*;

public class PoRadcich {
  public static void main(String[] argv) throws
     IOException {
    FileReader fr = new FileReader("a.txt");
    BufferedReader in = new BufferedReader(fr);
    FileWriter fw = new FileWriter("b.txt");
    BufferedWriter out = new BufferedWriter(fw);
    String radka;

    while((radka = in.readLine()) != null) {
      System.out.println(radka);
      out.write(radka);
      out.newLine();
    }

    fr.close();
    out.close();
  }
}
```

- formátovaný vstup - čte čísla, předpokládá každé číslo na nové řádce

```
import java.io.*;

public class FormatovanyVstup {
  public static void main(String[] args) throws
    IOException {
    FileReader fr = new FileReader("buf.txt");
    BufferedReader in = new BufferedReader(fr);
    String radka;
    int k, suma = 0;

    while((radka = in.readLine()) != null) {
      k = Integer.valueOf(radka).intValue();
      suma += k;
    }

    System.out.println("Soucet je: " + suma);
    fr.close();
  }
}
```

- neformátovaný binární vstup/výstup základních datových typů

```
import java.io.*;

public class BinarniZapis {
  public static void main(String[] args) throws
    IOException {
    FileOutputStream fwJm = new
     FileOutputStream("data.bin");
    DataOutputStream fw = new
     DataOutputStream(fwJm);
    int k, pocet;

    pocet = 2 + (int) (Math.random() * (10 - 2));
    fw.writeInt(pocet);

    for (int i = 0;  i < pocet;  i++) {
      k = (int) (1000.0 * Math.random());
      System.out.print(k + " ");
      fw.writeInt(k);
    }

    fw.writeDouble(Math.PI);
    fw.writeDouble(Math.E);
    System.out.println("\n" + Math.PI + " " +
     Math.E);
    fwJm.close();
```

```
FileInputStream frJm = new
    FileInputStream("data.bin");
   DataInputStream fr = new
    DataInputStream(frJm);

   pocet = fr.readInt();
   for (int i = 0;  i < pocet;  i++) {
     k = fr.readInt();
     System.out.print(k + " ");
   }

   double pi = fr.readDouble();
   double e = fr.readDouble();

   System.out.println("\n" + pi + " " + e);
   frJm.close();
  }
}
```

- formátovaný vstup – textový vstup základních datových typů
  vstup ve tvaru:

```
10   AAAAAA  BBBBBBB   12.5
12  CCCCCCC DDDDDDD  23.2
```

```
import java.io.*;
import java.util.*;
public class soubory {

    public static void main(String[] args) throws IOException {
            FileReader fr = new FileReader("inp.txt");
            BufferedReader in = new BufferedReader(fr);
            String radka,jm="",prjm="",cislo ;
            float x,suma;
            int k=0;
            x=0; suma=0;
            while ((radka=in.readLine()) != null){
                    StringTokenizer st=new StringTokenizer(radka);
                    while (st.hasMoreTokens()){
                            cislo=st.nextToken();
                            k=Integer.valueOf(cislo).intValue();
                            jm=st.nextToken();
                            prjm=st.nextToken();
                            cislo=st.nextToken();
                            x=Float.valueOf(cislo).floatValue();
                            }
                System.out.println(k+" "+jm+" "+prjm+" "+x);
                    suma+=x;

            }
            System.out.println("suma="+suma);
    fr.close();
            }
    }
```

Rozhraní (interface)

- Definuje soubor metod, které v něm nejsou implementovány. V deklaraci rozhraní jsou pouze hlavičky metod (jako u abstraktní třídy), třída, která rozhraní implementuje musí překrýt všechny jeho metody.

- Rozhraní se používá v případech kdy:
  - chceme třídě pomocí implementace rozhraní vnutit konkrétní metody
  - vidíme podobnost v různých třídách, ale pomocí dědění jde tato podobnost jen těžko realizovat (není např.. Možné vytvořit společného předka)
  - požadujeme vícenásobnou dědičnost

konstrukce rozhraní: - podobá se zápisu třídy

```
public interface Info {
  public void kdoJsem();
}
```

Použití jednoho rozhraní
- každá třída, která implementuje rozhraní ho musí uvést za klíčové slovo *implements*

```
public class Usecka implements Info {
  int delka;
  Usecka(int delka) { this.delka = delka; }
  public void kdoJsem() {
    System.out.println("Usecka");
  }
}

public class Koule implements Info {
  int polomer;
  Koule(int polomer) { this.polomer = polomer; }
  public void kdoJsem() {
    System.out.println("Koule");
  }
}

public class TestKoule {
  public static void main(String[] args) {
    Usecka u = new Usecka(5);
    Koule  k = new Koule(3);
    u.kdoJsem();
    k.kdoJsem();
  }
}
```

- Použití rozhraní jako typu referenční proměnné

  – pokud je rozhraní implementováno nějakou třídou lze deklarovat referenční proměnnou typu rozhraní, pomocí které lze přistupovat k instancím všech tříd, které toto rozhrani implementují

```
public class TestKoule {
  public static void main(String[] args) {
    Koule k = new Koule(3);
    Info i = new Usecka(5);
    i.kdoJsem();
    i = k;
    i.kdoJsem();
  }
}
```

- **Implementace více rozhraní jedinou třídou**
  - **třída může implementovat více jak jedno rozhraní = vicenásobná dědičnost**

```
public interface InfoDalsi {
  public void vlastnosti();
}
```

```
class Usecka implements Info, InfoDalsi {
  int delka;
  Usecka(int delka) { this.delka = delka; }
  public void kdoJsem() {
    System.out.print("Usecka");
  }
  public void vlastnosti() {
    System.out.println(" = " + delka);
  }
}

public class TestDvou {
  public static void main(String[] args) {
    Usecka u = new Usecka(5);
    u.kdoJsem();
    u.vlastnosti();
  }
}
```

- **Instance rozhraní může využívat jen metody rozhraní**
  - ➢**referenční proměnná typu rozhraní umožňuje přístup jen k metodám rozhraní, ostatní metody třídy nejsou z pohledu rozhraní známy**

```
class Usecka implements Info, InfoDalsi {
  int delka;

  Usecka(int delka) { this.delka = delka; }

  public void kdoJsem() {
    System.out.print("Usecka");
  }

  public void vlastnosti() {
    System.out.println(" = " + delka);
  }

  public int getDelka() { return delka; }
}

public class Test {
  public static void main(String[] args) {
    Info info = new Usecka(2);
    info.kdoJsem();
    // info.vlastnosti();                  // chyba
    // System.out.println(info.getDelka());   // chyba
  }
}
```

- **Implementované rozhraní se dědí beze změny**

Zdědíme=li třídu, která implementovala rozhraní, bude metoda z rozhraní přístupná v obou třídách, ale bude se jednat o tutéž metodu – metodu z rodičovské třídy

```
class Usecka implements Info {
  int delka;

  Usecka(int delka) { this.delka = delka; }

  public void kdoJsem() {
    System.out.println("Usecka");
  }
}

class Obdelnik extends Usecka {
  int sirka;

  Obdelnik(int delka, int sirka) {
    super(delka);
    this.sirka = sirka;
  }
}

public class Test {
  public static void main(String[] args) {
    Usecka u = new Usecka(5);
    Obdelnik o = new Obdelnik(2, 4);
    Info iu = new Usecka(6);
    Info io = new Obdelnik(3, 6);
    u.kdoJsem();     // vypíše Usecka
    o.kdoJsem();     //       -""-
    iu.kdoJsem();    //
    io.kdoJsem();    //
  }
}
```

UML

the Unified Modeling Language

Historie

- autory jsou G. Booch , J. Rumbaugh a I. Jacobson
- vývoj UML zahájen v roce 1994 Boochem a Rumbaughem ve společnosti Rational Software
- verze 1.0 vyšla v roce 1997
- nejnovější oficiální verze je 2.0

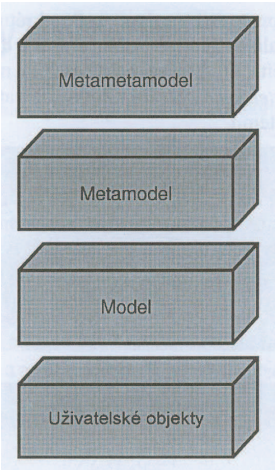
Co je to UML?

- UML (unified modeling language) je jednotný modelovací jazyk
- slouží k zakreslení, specifikaci, tvorbě a popisu součástí softwarových systémů
- popisuje, co má systém dělat
- neříká však, jak to má dělat
- úkolem UML je zpřehlednit návrh systému tak, aby mu porozuměli všichni zainteresovaní

Architektura UML

- UML je složeno ze čtyř vrstev
- Jednotlivé vrstvy se liší mírou obecnosti prvků

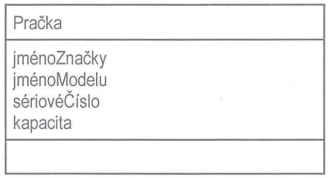
1. Vrstva uživatelských objektů
2. Modelová vrstva
3. Metamodelová vrstva
4. Metametamodelová vrstva



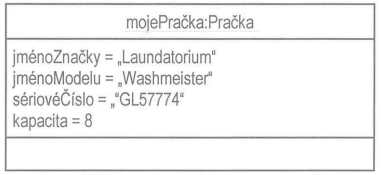
Základy jazyka UML

- UML se skládá z mnoha grafických prvků, které se dají kombinovat do podoby diagramů
- tyto diagramy umožňují dívat se na systém z různých pohledů
- nejběžnější diagramy:
  - diagram tříd
  - diagram objektů
  - diagram případu užití
  - diagram stavů (stavový diagram)
  - diagram sekvencí
  - diagram činností
  - diagram spolupráce
  - diagram komponent
  - diagram nasazení

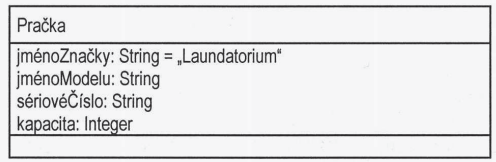
Diagram třídy



třída a její atributy



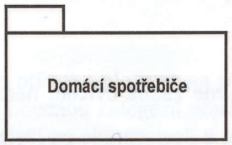
pojmenovaná instance



atribut s implicitní hodnotou

Diagram třídy

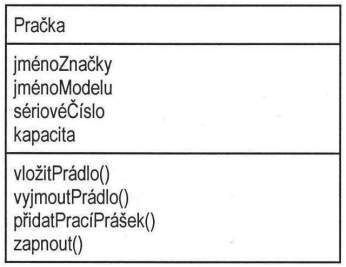
třída



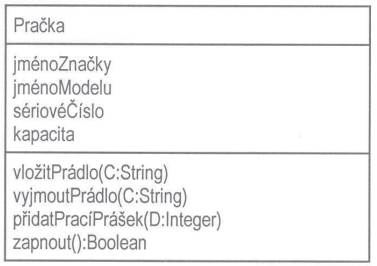
balíček

třída s názvem cesty

Diagram třídy



operace



příznaky

Diagram třídy

ProdejnéVýrobky
cena:Peníze /celkemProdanýchJednotek:Integer …
snížitCenu(diskontníSazba:Procenta) snížitCenu() celkemProdanýchJednotek(datum:Date) celkemProdanýchJednotek()

Přetěžování operací

Diagram třídy

Polygon {abstract}
/plocha:Plocha
vzítPlochu(out plocha:Plocha {abstract}

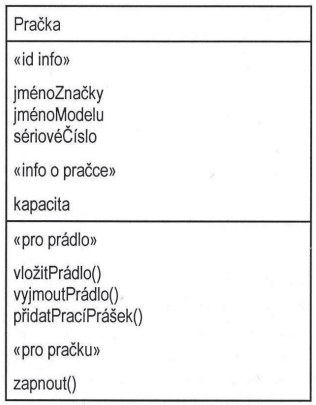
abstraktní metody a třídy

Diagram třídy

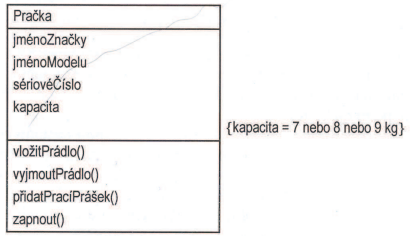
NějakáTřída
+ veřejnýAtribut:Třída1 # chráněnýAtribut:Třída2 - soukromýAtribut:Třída3
+ veřejnáOperace() # chráněnáOperace() - soukromáOperace()

viditelnost atributů a operací

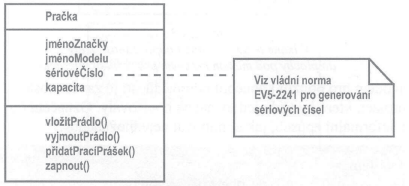
Diagram třídy



stereotyp

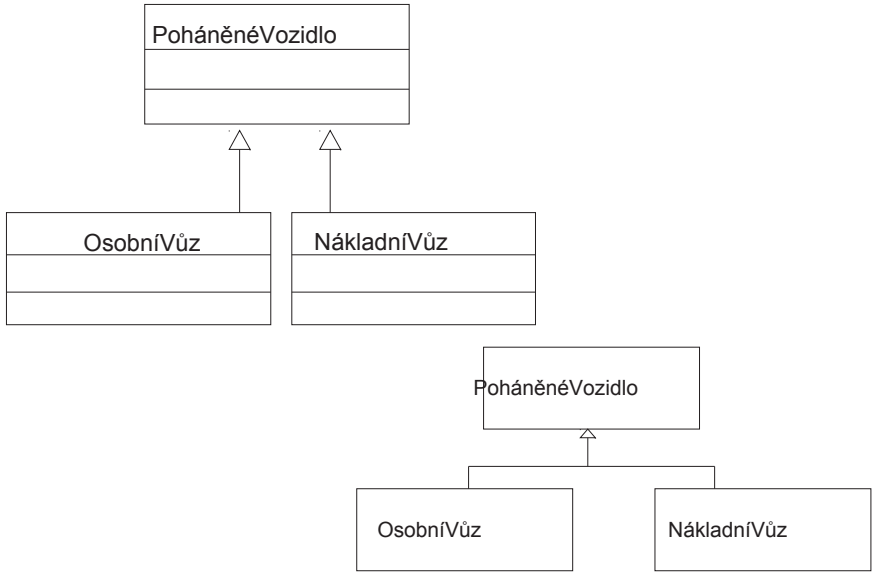


omezení

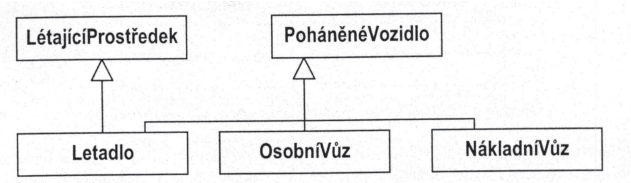


poznámky

Dědičnost - jednoduchá



Dědičnost - vícenásobná



Vztahy mezi třídami a objekty

Asociace

asociace říká, že objekty které jsou instancemi jedné třídy, mohou mít vztah s objekty jiné nebo stejné třídy např. objekty třídy Zaměstnanec budou mít asociaci k objektům třídy Oddělení.
v UML se znázorňují plnou čarou mezi třídami:

- u čáry volitelně název vztahu
- u názvu volitelně malý černý trojúhelníček ukazující, kterým směrem se má název vztahu číst

asociace může být i rekurzivní

- konce asociace mohou být volitelně popsány rolemi ve vztahu
- role popisující vzdálené konce asociací stejné třídy mají být jedinečné
- u rekurzivních vztahů (nadřízený řídí podřízeného) by role měla být uvedena vždy
- pojmenování rolí užitečné, pokud je více než jedna asociace mezi stejným párem tříd

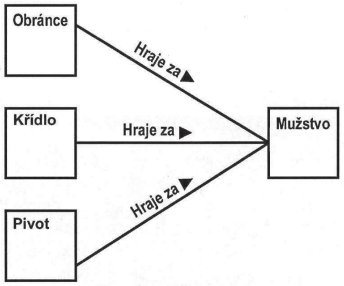
Práce se vztahy

asociace mezi třídami

každá třída hraje určitou roli

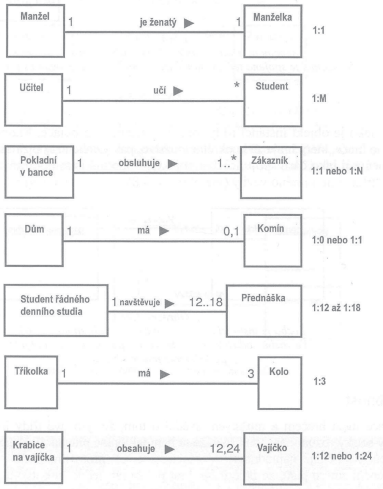
více asociací mezi třídami

Práce se vztahy



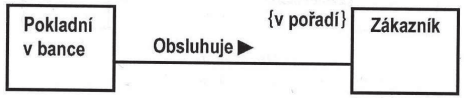
více tříd může být asociováno s jednou třídou

Práce se vztahy

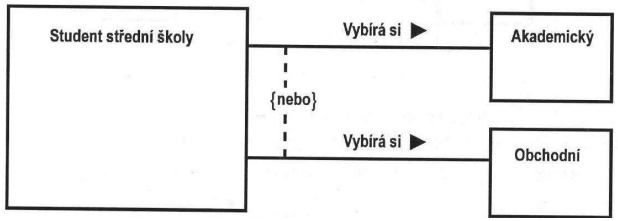


násobnost

Práce se vztahy



omezení asociace

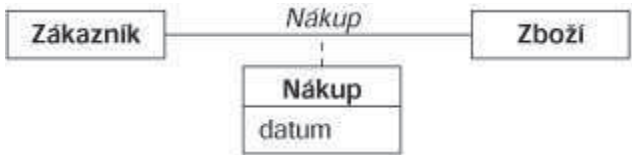


vztah „Nebo“

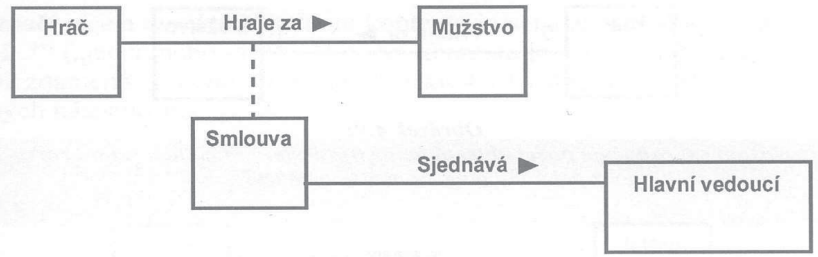
Asociace zobrazená jako třída

• pokud má asociace vlastnosti jako atributy, opeace a další asociace, můžeme pro ní vytvořit tzv. asociační třídu (analogie "asociativního indikátoru typu" v ERA diagramech)

Příklad:
Zákazník nakoupil zboží.
Asociace "nakoupil" sdružuje zákazníky a položky zboží - pokud bychom chtěli uchovat informaci o datu nákupu atd., tak to nepatří ani k zákazníkovi, ani ke zboží

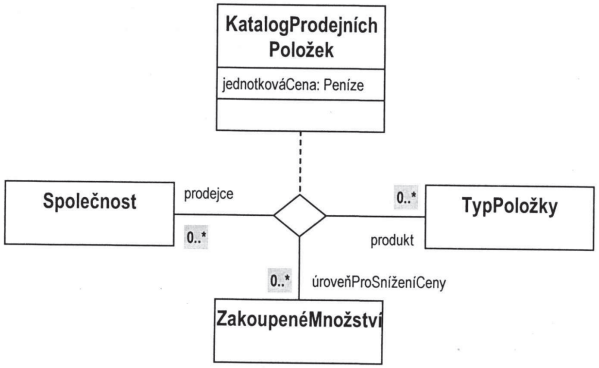


asociační třída



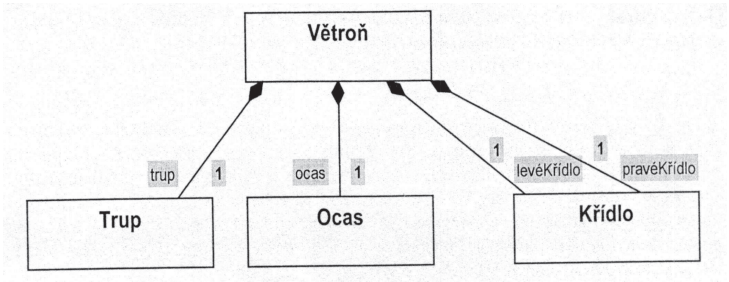
Asociace vyšších řádů

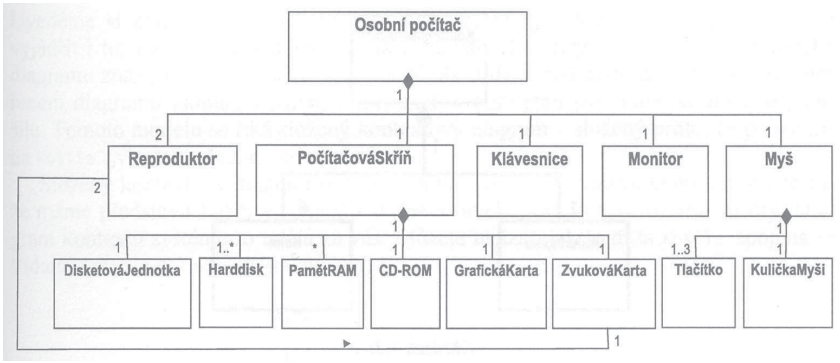
všechny dosavadní asociace byly binární, tj. do vztahu vstupovaly dvě strany
- binární asociace jsou zdaleka nejčastější, občas se může vyskytnout ternární atd.
- n-ární asociaci můžeme znázornit pomocí prázdného kosočtverce - následující obrázek ukazuje ternární asociaci, která je zároveň asociační třídou

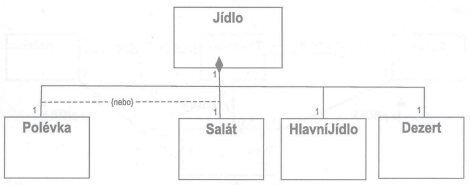


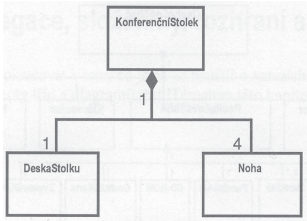
Asociace celku a částí

kompozice(složení)
kompozice je silná asociace - součást náleží právě jednomu složenému objektu
součást nemůže existovat samostatně (políčko nemůže existovat bez šachovnice, větroň nemůže existovat bez trupu, ocasu, křídel)
při zániku celku tedy zaniknou i jeho části
v UML se kompozice znázorňuje plným kosočtvercem nebo grafickým vnořením:









Agregace (seskupení)

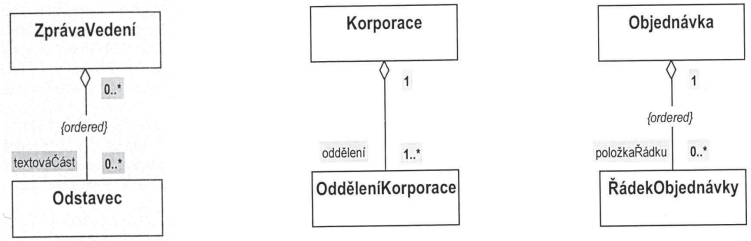
- jednou z nejčastějších binárních asociací
- objekt je vytvořen z dalších objektů = je agregátem množiny objektů

  Příklady:
  - objekt Stádo je agregátem Ovcí,
  - Les je agregátem Stromů,
  - Rodina bude agregátem objektu typu Muž, objektu typu Žena a množiny objektů Dítě
  - Předmět se může skládat z Přednášek, Cvičení, Zápočtové_úlohy, Zkoušky atd.

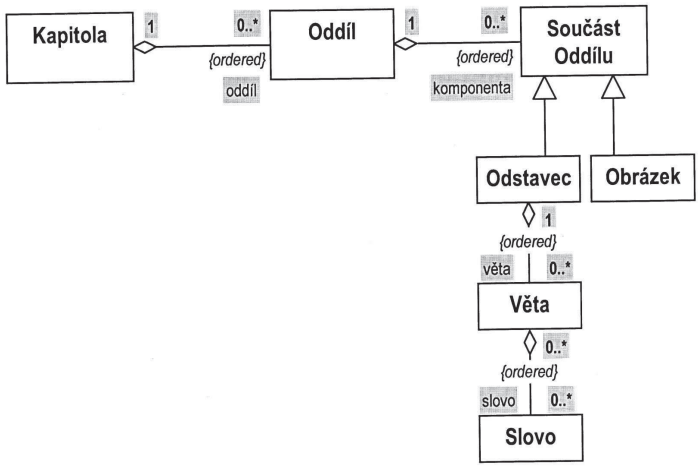
agregace je více než pouze součet svých částí, vzniká něco nového (agregát)

- agregát může vystupovat v některých operacích jako samostatná jednotka
- části mohou existovat samostatně, mohou být součástí dalších agregací

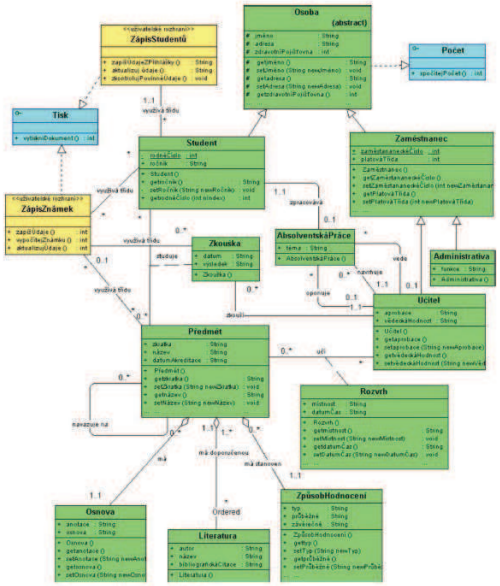
v UML se agregace znázorňuje prázdným kosočtvercem na straně agregátu:



Agregační struktura kapitoly v knize



Příklad diagramu tříd (informační systém školy)



Unifikovaný modelovací jazyk UML [1]

Karel Richta

katedra počítačů, FEL ČVUT v Praze
Karlovo nám. 13, 121 35 Praha 2
e-mail:richta@fel.cvut.cz

Klíčová slova: UML, OCL.

Abstrakt. *Komunikačním prostředkem informační komunity se postupem času stala angličtina. Chcete-li vystavit nějakou informaci tak, aby byla srozumitelná i mimo hranice ČR, použijete angličtinu. Podobně probíhá vývoj i v komunitě softwarových inženýrů. Během vývoje metodik softwarového inženýrství bylo navrženo mnoho různých víceméně formálních jazyků a prezentačních technik, které slouží pro popis softwarových produktů od konceptuálního modelu až po jeho implementaci. Různorodost zápisů přináší radost těm, kteří se neradi váží na určitý jazyk či grafickou techniku, méně radosti však činí těm, kteří musí takové zápisy po někom studovat a zpracovávat. Nové generace metodik softwarového inženýrství se snaží sjednotit užitečné vlastnosti různých metodik a integrovat je do nějaké společné sady. Jedním z nejdále propracovaných přístupů je tzv. unifikovaný modelovací jazyk UML (Unified Modeling Language) – kandidát na jakési "esperanto" moderního softwarového inženýrství.*

1. Úvod

Metodiky softwarového inženýrství představují soubor prezentačních technik (výrazových prostředků) a metodických postupů, které podporují systematický vývoj a údržbu programového díla, včetně plánování a řízení prací na programovém projektu. Metodiky se tedy mohou lišit prezentačními technikami (notací), metodickými postupy (proces) a dostupnými nástroji. Každá metodika přinesla něco nového, obvykle však též novou notaci. Rozmanité novinky jen výjimečně přinesly i novou kvalitu.

To bylo důvodem snahy o unifikaci vyjádření. Vznikají metodiky které se snaží sjednotit výhodné vlastnosti různých notací a metodických postupů. Patří sem např. metodika Fusion (Coleman, 1994), či Unified Method (Rumbaugh/Booch, 1995), která vznikla integrací metodiky OMT (Object Modeling Technique - Rumbaugh) a OOD (Object-Oriented Design - Booch).

Vývoj unifikované metodiky si dala za jeden ze svých hlavních cílů firma Rational [2], kterou založili Rumbaugh a Booch. Později se k nim připojil i Jacobson se svou metodikou OOSE (Object-Oriented Software Engineering) a společně vypracovali prvý návrh UML (1996). v roce 1997 převzalo tento návrh sdružení OMG (Object Management Group [1]) a pod označením UML 1.1 jej začalo doporučovat jako standard (1997). Současná verze z března 2003 má označení UML 1.5, pracuje se na verzi standardu UML 2.0.

UML je zamýšlen jako univerzální standard pro záznam, konstrukci, vizualizaci a dokumentaci artefaktů systémů s převážně softwarovou charakteristikou. Definice UML obsahuje 4 základní části:

- Definici notace UML (syntaxe).
- Metamodel UML (sémantika).

[1] *Vznik tohoto textu byl částečně motivován prací na výzkumném záměru MŠMT číslo MSM 212300014 "Výzkum v oblasti informačních technologií a komunikací" a spoluprací na grantovém projektu GAČR 201/03/0912 "Vyhledávání a indexování XML dokumentů".*

- Jazyk OCL (Object Constraint Language) pro popis dalších vlastností modelu.
- Specifikace převodu do výměnných formátů (CORBA IDL, XML DTD).

1.1 Notace UML

Při analýze požadavků se často jako nástroj pro komunikaci mezi zadavatelem a řešitelem využívá modelování. Vznikají různé modely navrhovaného systému, které mohou být na této úrovni testovány, ověřovány a upravovány. Teprve po dosažení shody mezi analytikem a zadavatelem může být model dekomponován a převeden do cílového implementačního jazyka.

Používané modely mohou být různých typů. Obecně lze model softwarového systému definovat jako souhrn informací, které jsou určitým způsobem strukturovány. Model by měl obsahovat veškeré shromážděné informace o systému.

Diagram je graficky znázorněný pohled na model. Diagram popisuje jistou část modelu pomocí grafických symbolů. Na rozdíl od modelu, jeden diagram málokdy popisuje celý systém – většinou je k popisu systému použito více pohledů, z nichž každý se zaměřuje na jeden aspekt modelu.

Analytické modely se zabývají zejména otázkou "**CO**" by měl systém dělat. Návrhové modely popisují dekompozici systému na programátorsky zvládnutelné části - zabývají se otázkou "**JAK**" by to mělo být uděláno. Implementační modely dokumentují implementaci.

UML se snaží shrnout nejlepší známé pohledy a definuje osm základních druhů diagramů, které budou popsány dále. UML sice nebrání vytváření dalších možných pohledů a diagramů, takový postup je však trochu proti duchu unifikace. Smyslem unifikace naopak je, aby stejné pohledy na model systému bylo možno vyjádřit tak, aby znázornění bylo obecně srozumitelné. Notace UML definuje, jak se zapisují základní pohledy na modelovaný systém.

1.2 Sémantika UML

Zápisy v UML (pohledy, diagramy) sestavené podle pravidel syntaxe musí mít pevně definovaný význam. Tím se zabývá popis sémantiky UML. Je rozčleněn do 4 vrstev, které na různé úrovni abstrakce popisují vlastnosti diagramů UML, včetně možných rozšíření. Na nejnižší úrovni jsou specifikovány vlastnosti primitivních údajů (dat) - typy dat, obory hodnot atributů. o úroveň výše je vyjádřena sémantika uživatelských objektů (tzv. model, nebo též meta-data) - např. co to je záznam o objektu typu "zaměstnanec". Ještě výše stojí popis prvků modelu (metamodel) - např. co to je třída, atribut, vztah, atd. Nejvýše je pak definice vlastností metamodelu (meta-metamodel), např. jak lze korektně vytvářet nové prvky modelu.

1.3 Jazyk OCL

Ne všechny vlastnosti modelu lze vyjádřit pomocí diagramů. Uvažme např. datový model systému, který se zabývá evidencí zaměstnanců. O každém zaměstnanci si potřebujeme pamatovat různé atributy, mimo jiné např. plat zaměstnance. Rovněž si potřebujeme pamatovat hierarchickou strukturu mezi zaměstnanci. Pomocí diagramů vyjádříme snadno požadavky na strukturu dat, těžko zde však zachytíme požadavky jako "nadřízený musí mít vyšší plat než jeho podřízení". UML nabízí (ale nevnucuje) možnost použít pro vyjádření takových omezení specifikační jazyk OCL (Object Constraint Language). OCL poslouží právě pro vyjádření komplikovanějších integritních omezení, popis vlastností operací a může být koneckonců použit i pro vyjádření sémantiky UML.

1.4 Specifikace výměnných formátů

Součástí definice UML je i specifikace výměnných formátů, sloužících např. pro přenos zápisů v UML mezi různými nástroji. Jako příklad lze uvést formát CORBA IDL (Interface Definition Language), či XMI (XML Metadata Interchange).

2. Lexikální elementy UML

Základem definice syntaxe UML je specifikace lexikálních elementů – nejmenších lexikálních jednotek, ze kterých se může dokumentace v UML skládat. Existují 4 základní druhy lexikálních elementů UML:

- **řetězec** (posloupnost znaků, znaková sada není omezena, v některých případech je doporučeno používat ASCII kódu – zajistí to snadný přenos do programového prostředí),
- **ikona** (grafický symbol zastupující element, neobsahující žádné složky),
- **2D-symbol** (grafický symbol zastupující element, který má obsah, případně rozdělený na složky),
- **spojka** (path - posloupnost úseček, které navazují, mohou mít zvýrazněny koncové body a mohou incidovat s hranicí 2D-symbolů - slouží k vyznačení propojení).

Elementy modelu dokumentovaného v UML potřebujeme označovat - potřebujeme jména objektů, tříd, atributů, metod, a pod. Jako jména se v UML používají **identifikátory** – speciální lexikální elementy typu řetězec, které obsahují pouze ASCII písmena, cifry a znaky '_' nebo '-'. Pro zápis identifikátorů, které zastupují víceslovní termín je vhodné zvolit nějakou konvenci, doporučené je používání některého z následujících zápisů:

```
josefNovak, josef_novak, josef-novak
```

Jméno se obvykle vztahuje k nějakému kontextu (scope), pak mluvíme o tzv. **kvalifikovaném jménu**. Např. úplné kvalifikované jméno pro objekt "objednávka", o kterém mluvíme v rámci "účetnictví" bude

```
Ucetnictvi::Objednavka
```

Řetězec přidružený ke grafickému symbolu nazýváme **návěští**. Pokud má identifikátor význam **klíčového slova**, uvádíme jej ve "francouzských" závorkách (**guillemets**):

```
<<keyword>>
```

Výrazy v UML jsou řetězce sestavené z lexikálních elementů podle jistých pravidel. Syntaxe výrazů není striktní, musí být ale zvolen nějaký dobře definovaný jazyk. Tj. výraz představuje formuli, kterou musí být možno v daném jazyce vyhodnotit (má svou **sémantiku**). Definice UML zahrnuje definici jazyka OCL (Object Constraint Language), který je použit pro definici sémantiky UML (metamodelu UML). Je doporučeno používat OCL všude tam, kde je to možné..

3. Diagramy UML

Dokumentace v UML se nemusí skládat pouze z (více, či méně formálních) textů. Ze základních lexikálních elementů lze vytvářet dvourozměrné diagramy – rozmístíme na ploše určitou sadu elementů a případně je propojíme pomocí spojek. Teoreticky lze použít libovolné elementy a spojky, z hlediska unifikace je však výhodné, pokud zvolíme určitou sadu elementů, která je vhodná pro vyjádření některého smysluplného pohledu na modelovaný systém. Touto volbou jsme de facto definovali **typ diagramu**. UML proto zahrnuje definici 8-mi typů diagramů, tj. 8-mi různých pohledů na model systému:

- **diagramy tříd** a **objektů** (class diagrams, object diagrams) popisují statickou strukturu systému, znázorňují datový model systému od konceptuální úrovně až po implementaci,
- **modely jednání** (diagramy případů užití - use case diagrams) dokumentují možné případy použití systému - události, na které musí systém reagovat,
- **scénáře činností** (diagramy posloupností - sequence diagrams) popisují scénář průběhu určité činnosti v systému,
- **diagramy spolupráce** (collaboration diagrams) zachycují komunikaci spolupracujících objektů,

- **stavové diagramy** (statechart diagrams) popisují dynamické chování objektu nebo systému, možné stavy a přechody mezi nimi,
- **diagramy aktivit** (activity diagrams) popisují průběh aktivit procesu či činnosti,
- **diagramy komponent** (component diagrams) popisují rozdělení výsledného systému na funkční celky (komponenty) a definují náplň jednotlivých komponent,
- **diagramy nasazení** (deployment diagrams) popisují umístění funkčních celků (komponent) na výpočetní uzly informačního systému.

Statickou strukturu systému vyjadřují diagramy tříd, diagramy spolupráce, diagramy komponent a diagram nasazení. **Funkční stránku** popisují model jednání, diagramy aktivit, scénáře událostí a diagramy spolupráce. **Dynamickou stránku** dokumentují stavové diagramy, scénáře událostí, diagramy spolupráce a diagramy aktivit.

Použití diagramů v jednotlivých fázích vývoje je dáno konkrétní metodikou, ale orientačně můžeme říci, že v rámci analýzy se využívají diagramy tříd, model jednání, diagramy aktivit, scénáře činností a stavové diagramy. Pro fázi návrhu jsou typické diagramy tříd, diagramy spolupráce, diagramy aktivit, diagramy komponent a diagramy nasazení. Ve fázi implementace se používají diagramy tříd, diagramy komponent a diagramy nasazení.

UML se snaží být univerzální notací pro všestranné použití od obchodního modelování po detailní návrh systémů pro práci v reálném čase. Notace, která by vyčerpávajícím způsobem pokryla tak široké spektrum potřeb by ale byla velmi komplikovaná a složitá. UML proto definuje pouze základní část (UML core) a umožňuje rozšíření notace dle konkrétních potřeb. Rozšíření UML lze dosáhnout použitím jiného jazyka (než OCL) pro výrazy. Standardní sémantiku UML je možno doplnit standardními omezeními, která umožňují změnit interpretaci elementů:

- **příznaky** (tags) – umožňují přidat k prvku diagramu další informace (atributy),
- **omezení** (constraints) – umožňují specifikovat omezení pro elementy (hodnoty atributů),
- **stereotypy** (stereotypes) – umožňují klasifikovat elementy.

Stereotypy umožňují klasifikovat elementy diagramů a tím vyjádřit další sémantiku. Zapisují se jako klíčová slova, ale(mohou se případně zobrazit jako ikony, což se příliš nedoporučuje - porušuje to princip jednotnosti, neboť tyto ikony nejsou standardní. Jako příklad mohou posloužit stereotypy:

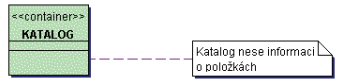
`<<database>>` (označení komponenty zabývající se správou dat)
`<<entity>>` (označení třídy reprezentující data)

Existují standardní stereotypy, např. `<<include>>`, `<<extend>>`, jejichž význam je definován sémantikou UML. Příznaky umožňují přidat k prvku diagramu další informace ve formě dvojice atribut=hodnota:

```
{ autor="Josef Novák", verze=1.2 }
```

Název atributu může být libovolný. Příznaky se zapisují do složených závorek, připojují se k názvu prvku diagramu a typicky je lze použít například ke specifikaci verze, autora, či implementačních poznámek. Omezení umožňují specifikovat požadavky na sémantiku prvků - lze pomocí nich formulovat různá omezení v modelu. Zapisují se jako výraz uzavřený do složených závorek. Příklad:

```
{ počet_zaznamu < 10000 }
```

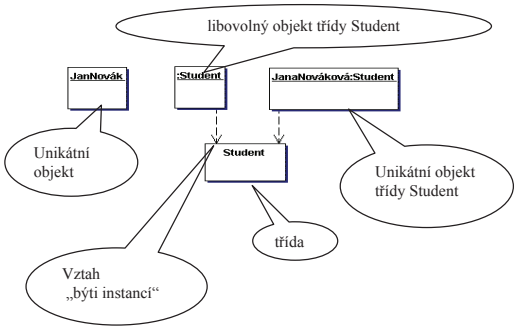


Omezení se může týkat více prvků - pak je spojeno s těmito prvky čárkovanou čárou. Často se kombinuje s poznámkami. Poznámka obsahuje libovolný text. Zobrazuje se jako obdélník s „přehnutým rohem". Může být přidružena k elementu, může obsahovat stereotyp (např. `<<constraint>>` - pak se jedná o specifikaci omezení).

4. Diagramy objektů (Object Diagrams)

Objekt je pojem, abstrakce, nebo věc s dobře definovanými hranicemi a významem. Každý objekt má tři charakteristiky: identitu, stav a chování.

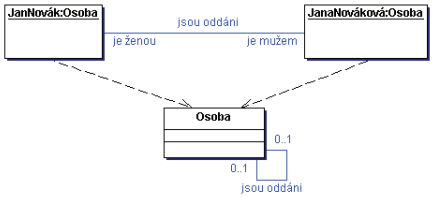
- **Stav** objektu je jedna z možných situací, ve kterých se objekt může nacházet. Stav objektu se může měnit a je definován sadou vlastností - atributy a vztahy.
- **Chování** určuje, jak objekt reaguje na žádosti jiných objektů a vyjadřuje vše, co může objekt dělat. Chování je implementováno sadou operací (metod).
- **Identita** znamená, že každý objekt je jedinečný.



5. Diagramy tříd (Class Diagrams)

Třídy zachycují společné vlastnosti sady objektů - **atributy** a **operace** (metody). Stereotypem lze zavést nové druhy (skupiny) tříd. Nejčastějšími skupinami (stereotypy) tříd jsou:

- **entitní třídy** (stereotyp je <<entity>>),
- **třídy rozhraní** (stereotyp je <<boundary>>) a
- **třídy řídicí** (stereotyp je <<control>>).



Vztahy mezi třídami (asociace) vyznačují možné vazby mezi objekty. Konce vztahů mohou být ohodnoceny rolí - role označují jakou roli objekt ve vztahu hraje. Pro zachycení kardinality a volitelnosti vztahů se používá notace N..M, kde N a M může být číslo nebo *, samotná * znamená totéž jako 0..*.

Pokud je zapotřebí si o vztahu něco pamatovat, požívají se tzv. přidružené třídy (atributy vztahů).

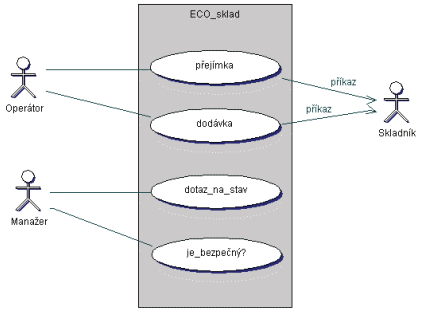
Speciální druhy vztahů představují agregace a generalizace:

- **agregace** (aggregation) – je druh vztahu, kdy jedna třída je součástí jiné třídy - vztah typu celek/část,
- **kompozice** (composition) – je silnější druh agregace - u kompozice je část přímo závislá na svém celku, zaniká se smazáním celku a nemůže být součástí více než jednoho celku,
- **generalizace** (generalization) – druh vztahu, kdy jedna třída je zobecněním vlastností jiné třídy (jiných tříd) - vztah typu nadtyp/podtyp, generalizace/specializace.

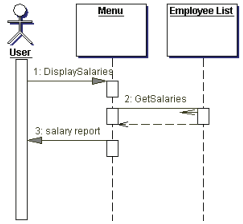
6. Model jednání (Use Case Model)

Model jednání slouží pro základní vymezení hranice mezi systémem a jeho okolím. Komponentami tohoto modelu jsou:

- **aktér** (actor) - uživatelská role nebo spolupracující systém
- **hranice systému** (systém boundary) - vymezení hranice systému
- **případ použití** (use case) - dokumentace události, na kterou musí systém reagovat
- **komunikace** - vazba mezi aktérem a případem použití (aktér komunikuje se systémem na daném případu).



Při vytváření modelu jednání je třeba brát v úvahu, že existují tzv. sekundární aktéři, tj. uživatelské role nebo spolupracující systémy, pro něž není systém přímo určen, ale které jsou pro jeho činnost nutné. Případy, kdy chceme vyznačit směr komunikace, značíme orientovanými šipkami. Mezi případy použití mohou existovat vztahy. Pokud chceme explicitně vyjádřit fakt, že takový vztah existuje, používáme stereotypy, standardně <<include>> - pokud jeden případ zahrnuje případ jiný (např. autentizaci), nebo <<extend>> - pokud nějaký případ rozšiřuje chování jiného (je zde možnost volby).

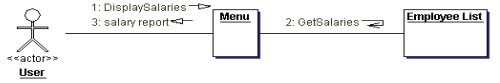


7. Scénáře činností (Sequence Diagrams)

Dokumentují spolupráci participantů na **scénáři** činnosti. Kladou přitom důraz na **časový** aspekt komunikace. Dokumentují **objekty** a **zprávy**, které si objekty posílají při řešení scénáře. Jsou vhodné pro popis scénáře při komunikaci s uživateli.

8. Diagramy spolupráce (Collaboration Diagrams)

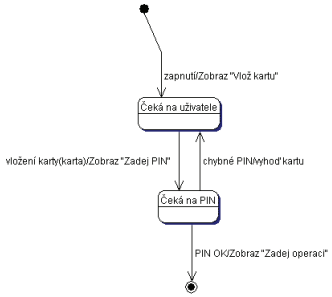
Podobně jako scénáře činností dokumentují diagramy kolaborace **spolupráci** objektů při řešení úlohy. Kladou větší důraz na **komunikační** aspekt (čas je vyjádřen číslováním).



Dokumentují **objekty** a **zprávy**, které si posílají při řešení problému. Jsou vhodné pro popis spolupráce objektů při návrhu komunikace.

9. Stavové diagramy (State Charts Diagrams)

Stavové diagramy slouží k popisu dynamiky systému. Stavový diagram definuje možné **stavy**, možné **přechody** mezi stavy, **události**, které přechody iniciují, **podmínky** přechodů a **akce**, které s přechody souvisí. Stavový diagram lze použít pro popis dynamiky objektu (pokud má rozpoznatelné stavy), pro popis metody (pokud známe algoritmus), či pro popis protokolu (včetně protokolu o styku uživatele se systémem). Přechod může být ohodnocen:




```
událost(parametry)[podmínka]/akce^zpráva
```

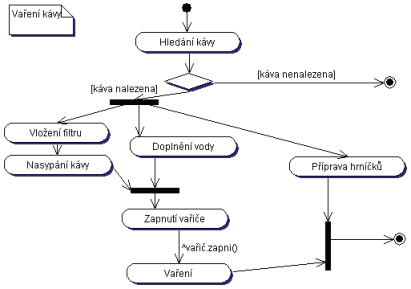
Každý stav může dále obsahovat popis akcí pro události vstup, výstup a opakované provádění:

```
entry/akce
exit/akce
do/akce
```

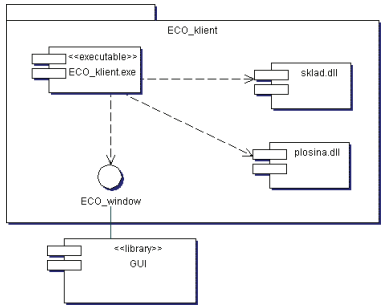
Stavové diagramy mohou být hierarchické. V nejnovějších verzích UML mohou obsahovat tzv. synchronizační značky (viz dále).

10. Diagramy aktivit (Activity Diagrams)

Varianta stavových diagramů, kde kromě stavů používáme **aktivity**. Nachází-li se system v nějakém stavu, je přechod do jiného stavu iniciován nějakou vnější událostí. U aktivity je, na rozdíl od stavu, přechod iniciován ukončením aktivity, přechody jsou vyvolány dokončením akce (jsou synchronní). Používají se pro dokumentaci tříd, metod, nebo případů použití (jako „workflow“). Nahrazují do určité míry v UML neexistující diagramy datových toků. Mohou obsahovat symbol „rozhodnutí“.



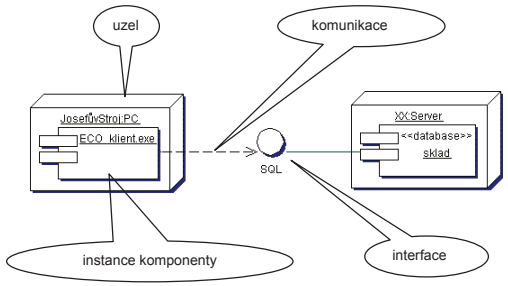
11. Diagramy komponent (Component Diagrams)



Vyjadřují (fyzickou) strukturu **komponent** systému. Popisují typy komponent - instance komponent jsou vyjádřeny v diagramu nasazení. Komponenty mohou být vnořeny do jiných komponent. Při vyjadřování **vztahu** mezi komponentami lze používat „interface“.

12. Diagramy nasazení (Deployment Diagrams)

Diagramy nasazení popisují fyzické rozmístění elementů systému na **uzly** výpočetního systému. Uzly a elementy jsou značeny obdobně jako objekty a třídy (může být uveden pouze typ, nebo konkrétní instance a typ - podtržena). Popisují nutné **vazby** mezi uzly (případně též použitý protokol - „interface“). Obsahují pouze **komponenty** potřebné pro běh aplikace - komponenty potřebné pro překlad a sestavení jsou uvedeny v diagramu komponent.



Shrnutí

Notace UML pomáhá sjednocení a čitelnosti dokumentace systémů. UML se ještě bude vyvíjet.

Poděkování

S potěšením zde mohu poděkovat firmě TogetherSoft Corporation (nyní Borland [4]), jejíž laskavá politika vůči vysokým školám mi umožnila pomocí nástroje Together Control Center připravit obrázky pro tuto publikaci.

Reference

1. OMG, Object Management Group; http://www.omg.org.
2. Rational software: http://www.rational.com/uml
3. Richta,K., Svačina,J.: UML: Teorie a praxe. In: Proc. of DATAKON 2003, pp. 1-26. ISBN 80-210-2958-7, Masarykova Universita, Brno, 2002.
4. Togethersoft: http://www.togethersoft.com
5. World Wide Web Consortium; http://www.w3.org.

Sumary

The paper briefly introduces the reader into the Unified Modeling Language UML. The paper only addresses the idea; it suggests to utilize common notation instead of specific languages. After considerable experience with the diagrams, the reader will optionally accept their usefulness.

Implementace abstraktních datových typů

- **Zásobník** (Stack)
- **Fronta** (Queue)
- **Obousměrná fronta** (Dequeue)
- **Vektor** (Vector)
- **Seznam** (List)

Metody pro práci se zásobníkem:

`push(o)` : Vkládá objekt **o** na vrchol zásobníku
Vstup: Objekt **Výstup:** Není

`pop()` : Odstraňuje objekt z vrcholu zásobníku. **Pokud je zásobník prázdný – chyba**
Vstup: Není **Výstup:** Object

`size()` : Vrací počet objektů v zásobníku
Vstup: Není **Výstup:** Integer

`isEmpty()`: Vrací *true* pokud je zásobník prázdný
Vstup: Není **Výstup:** Boolean

`top()`: Vrací objekt z vrcholu zásobníku bez jeho odstranění. Pokud je zásobník prázdný dojde k chybě.
Vstup: Není **Výstup:** Object

ADT Zásobník (Stack)

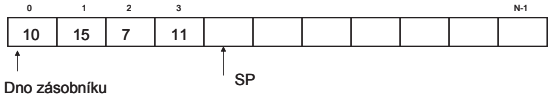
- Zásobník je datová struktura, do které se objekty vkládají a vybírají podle strategie **LIFO** (**L**ast-**I**n-**F**irst-**O**ut)

- Použití:
  - ukládání návratových adres podprogramů
  - ukládání změn v textu při operaci *Undo* v textových editorech
  - uchovávání historie adres ve Web browseru
  - zpracování výrazů v postfixových notacích
  - atd.

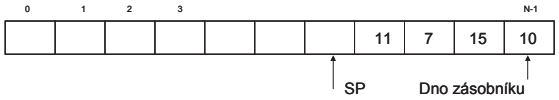
- Př. Posloupnost operací se zásobníkem a jejich výsledek

Operace	Výstup	S
push(5)	-	(5)
push(3)	-	(5,3)
pop()	3	(5)
push(7)	-	(5,7)
pop()	7	(5)
top()	5	(5)
pop()	5	()
pop()	chyba	()
isEmpty()	true	()
push(9)	-	(9)
push(7)	-	(9,7)

- **Implementace polem**
  - přímý – ve směru rostoucích adres (vrchol zásobníku roste, dno zásobníku je prvek pole s indexem 0)



  - Zpětný – ve směru klesajících adres (vrchol zásobníku klesá, dno zásobníku – prvek pole s indexem N-1)



SP – stack pointer, ukazuje na první volnou položku, (používá se i způsob kdy SP ukazuje na poslední obsazenou položku)

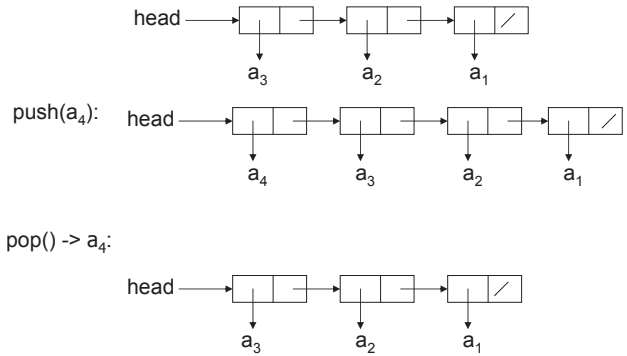
Příklad implementace zásobníku polem v Javě

```
public class StackA {
   private final int  CAPACITY=100;
   private Object[] s; // pole pro ulozeni prvku zasobniku
   private int sz;
   private int sp;    //ukazatel zasobniku

   public StackA(){
        s=new Object[CAPACITY];
        sp=0;
        sz=0;
   }

   public boolean isEmpty(){
        if (sp==0)
               return(true);
        else return(false);
   }

   public int size(){
        return(sz);
   }
```

```
   public Object top()throws StackEmptyException {
        if (sp==0)
               throw new StackEmptyException("Stack is empty");
        else return(s[sp-1]);
   }

   public Object pop() throws StackEmptyException {
        if (sp==0)
               throw new StackEmptyException("Stack is empty");
        else return(s[--sp]);
   }

   public void push(Object e) throws StackFullException{
        if (sp>=CAPACITY)
               throw new StackFullException("Stack is full");
        else s[sp++]=e;
   }

   public class StackEmptyException extends RuntimeException {
        public StackEmptyException(String err) {
               super(err);
        }
   }
```

```
public class StackFullException extends RuntimeException {
               public StackFullException(String err) {
                      super(err);
               }
       }
public static void main(String[] args) {
 StackA st=new StackA();
 int [] a={10, 12, 15, 21, 33};
 for (int i=0; i<a.length; i++)
 st.push(new Integer(a[i]));
 int n=st.size();
 System.out.println("Pocet prvku na zasobniku: "+n);
 while (!st.isEmpty())
 System.out.print(st.pop().toString()+"  ");
 System.out.println("\n"+"Pocet prvku na zasobniku: "+st.size());
       }
```

- **Implementace seznamem zřetězených prvků**
  - prvky se vkládají na čelo seznamu a vybírají se z čela



Příklad implementace zásobníku seznamem v Javě

```
public class StackL {
   private Node sp;  // ukazatel na vrchol zasobniku
   private int sz;    // pocet objektu v zasobniku

   public StackL() {
        sp=null;
        sz=0;
   }

   public boolean isEmpty() {
        if (sp==null)
                return true;
        else return false;
   }

   public int size(){ return(sz);} // vraci pocet objektu v
   zasobniku
   public void push(Object o){
        sp=new Node(o,sp);
        sz++;
   }
```

```
public class Node {
   private Object element;
   private Node   next;

   public Node() { element=null; next=null; }

   public Node(Object element, Node next){
       this.element=element;
       this.next=next;
   }

   public Object getElement(){ return (element); }

   public Node getNext() { return(next); }

   public void setElement(Object e){ element=e; }

   public void setNext(Node n){ next=n; }

}
```

```
public Object pop()throws StackEmptyException {
   if (sp==null)
        throw new StackEmptyException("Stack is empty");
   else {Object tmp=sp.getElement();
         sp=sp.getNext();
         sz--;
         return(tmp);
         }
   }

public Object top() throws StackEmptyException {
   if (sp==null)
        throw new StackEmptyException("Stack is empty");
   else return(sp.getElement());
   }

 public class StackEmptyException extends RuntimeException {
   public StackEmptyException(String err) {
        super(err);
        }
   }
```

```
public static void main(String[] args)  {
   StackL st=new StackL();
   int [] a={10, 12, 15, 21, 33};
   for (int i=0; i<a.length; i++)
   st.push(new Integer(a[i]));
   int n=st.size();
   System.out.println("Pocet prvku na zasobniku: "+n);
   while (!st.isEmpty())
    System.out.print(st.pop().toString()+"  ");
   System.out.println("\n"+"Pocet prvku na zasobniku:"+st.size());
   }
}
```

```
import java.util.*;

public class PokusStack {

   public static void main(String[] args) {
    Stack st=new Stack();
    int [] a={10, 12, 15, 21, 33};
     for (int i=0; i<a.length; i++)
        st.push(new Integer(a[i]));
     int n=st.size();
      System.out.println("Pocet prvku na zasobniku: "+n);
      while (!st.empty())
       System.out.print(st.pop().toString()+"  ");
      System.out.println("\n"+"Pocet prvku na zasobniku:"+
                          st.size());
   }
   )
```

```
Pocet prvku na zasobniku: 5
33  21  15  12  10
Pocet prvku na zasobniku: 0
```

Zásobník v Java Core API

- V Java Core Api je k dispozici třída java.util.Stack implementující zásobník (je oddděná od třídy Vector)
- Používá metody :

`boolean empty()`	–	vrací *true* pokud je zásobník prázdný
`Object peek()`	–	vrací objekt z vrcholu bez jeho odebrání ze zásobníku
`Object push(O)`	–	vkládá objekt **O** na vrchol zásobníku
`Object pop()`	–	vybírá objekt ze zásobníku
`int search(O)`	–	vrací pozici objektu **O** na zásobníku

ADT Fronta (Queue)

- Fronta je datová struktura, do které se objekty vkládají a ze které se vybírají podle strategie **FIFO** (**F**irst-**I**n-**F**irst-**O**ut)

Použití:

- systémy hromadné obsluhy (rezervace vstupenek apod.)
- simulace SHO
- na systémové úrovni - fronta požadavků na procesor popř. periferní zařízení

Metody pro práci s frontou

`enqueue(o)` : Vkládá objekt **o** na konec fronty
Vstup: Objekt **Výstup:** Není

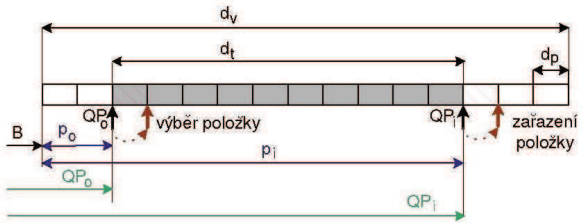
`dequeue()` : Odstraňuje objekt z čela fronty Pokud je fronta prázdná – chyba
Vstup: Není **Výstup:** Object

`size()` : Vrací počet objektů ve frontě
Vstup: Není **Výstup:** Integer

`isEmpty()` : Vrací *true* pokud je fronta prázdná
Vstup: Není **Výstup:** Boolean

`front()` : Vrací objekt z čela fronty bez jeho odstranění. Pokud je fronta prázdná dojde k chybě.
Vstup: Není **Výstup:** Object

- Implementace fronty polem prvků (tzv. cyklický buffer)



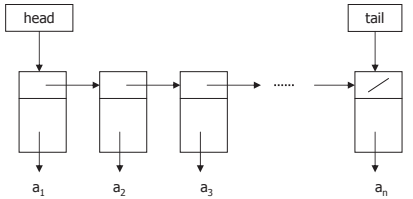
$$QP_i = B + (p_i + d_p) \bmod d_v$$
$$QP_0 = B + (p_0 + d_p) \bmod d_v$$

d_v – délka implementujícího vektoru
d_t – délka fronty
B – bázová adresa začátku fronty (index 0)

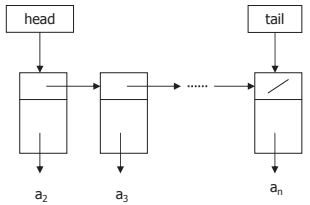
- Př. Posloupnost operací s frontou a jejich výsledek

Operace	Výstup	čelo← Q ←týl
enqueue(5)	-	(5)
enqueue(3)	-	(5,3)
dequeue()	5	(3)
enqueue(7)	-	(3,7)
dequeue()	3	(7)
front()	7	(7)
dequeue()	7	()
dequeue()	chyba	()
isEmpty()	true	()
enqueue(9)	-	(9)
enqueue(7)	-	(9,7)

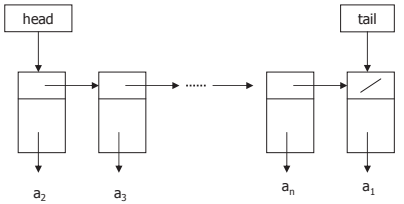
- **Implementace seznamem zřetězených prvků**
  - prvky se vkládají na konec seznamu a vybírají se z čela



dequeue -> a_1:



enqueue(a_1)



Fronta v Java Core API

- **Datový typ fronta není v Java Core API implementován, lze jej snadno implementovat pomocí třídy java.util.LinkedList**

Metody fronty	Odpovídající metody LinkedList
`size()`	`size()`
`isEmpty()`	`isEmpty()`
`front()`	`getFirst()`
`enqueue(o)`	`addLast(o)`
`dequeue()`	`removeFirst()`

Metody pro práci s obousměrnou frontou

`insertFirst(o)` : Vkládá objekt o na začátek fronty
Vstup: Objekt **Výstup:** Není

`insertLast(o)` : Vkládá objekt **o** na konec fronty
Vstup: Objekt **Výstup:** Není

`removeFirst()` : Odstraňuje objekt z čela fronty.
Pokud je fronta prázdná – chyba
Vstup: Není **Výstup:** Object

`removeLast()` : Odstraňuje objekt z konce fronty.
Pokud je fronta prázdná – chyba
Vstup: Není **Výstup:** Object

`size()` : Vrací počet objektů ve frontě
Vstup: Není **Výstup:** Integer

`isEmpty()` : Vrací *true* pokud je fronta prázdná
Vstup: Není **Výstup:** Boolean

`first()` : Vrací objekt z čela fronty.
Pokud je fronta prázdná – chyba
Vstup: Není **Výstup:** Object

`last()` : Vrací objekt z konce fronty.
Pokud je fronta prázdná – chyba
Vstup: Není **Výstup:** Object

ADT Obousměrná fronta (Double-Ended Queue)

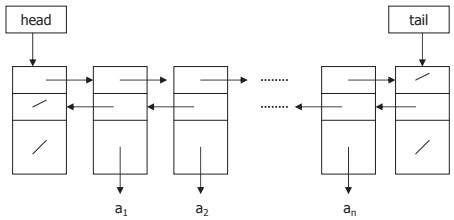
- Obousměrná fronta je datová struktura podobná frontě, ale umožňuje vkládání a výběr prvků na obou koncích

- Použití:
  - lze ji požít v aplikacích kde používáme zároveň zásobník i frontu
  - pomocí obousměrné fronty lze implementovat některé další ADT

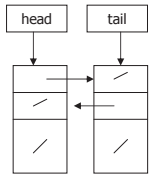
- Př. Posloupnost operací s obousměrnou frontou a jejich výsledek

Operace	Výstup	DQ
insertFirst(3)	-	(3)
insertFirst(5)	-	(5,3)
removeFirst()	5	(3)
insertLast(7)	-	(3,7)
removeFirst()	3	(7)
removeLast()	7	()
removeFirst()	chyba	()
isEmpty()	true	()

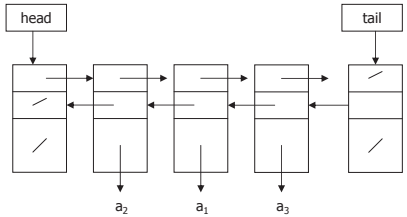
- **Implementace seznamem obousměrně zřetězených prvků**



Prázdná fronta :



insertFirst(a_1), insertFirst(a_2),insertLast(a_3) :



ADT Vektor (Vector)

- Vektor je lineární posloupnost prvků V, která obsahuje n prvků. Každý prvek vektoru V je přístupný prostřednictvím indexu r (rank) v rozsahu [0, n-1]. Vektor připomíná datový typ pole, ale není to pole!!!

Metody pro práci s vektorem :

`elemAtRank(r):` Vrací prvek Vektoru V v pozici **r**, r∈<0,n-1> jinak chyba, **n** je počet prvků ve vektoru.
Vstup: Integer **Výstup**: Objekt

`replaceAtRank(r,o):` Zamění prvek v pozici **r** prvkem **o** a vrátí původní prvek, r∈<0,n-1> jinak chyba.
Vstup: Integer **r** a Objekt **o** **Výstup**: Objekt

`insertAtRank(r,o):` Vloží nový prvek **o** v pozici **r**, r∈<0,n> jinak chyba.
Vstup: Integer **r** a Objekt **o** **Výstup**: Není

`removeAtRank(r):` Odstraní a vrátí prvek v pozici **r**, r∈<0,n-1> jinak chyba.
Vstup: Integer **r** **Výstup**: Není

Implementace zásobníku a fronty obousměrnou frontou

- Pokud máme implementovanou obousměrnou frontu, lze ji využít jako zásobník popř. jako obyčejnou frontu

Metody zásobníku	Odpovídající metody DQ
`size()`	`size()`
`isEmpty()`	`isEmpty()`
`top()`	`last()`
`push(o)`	`insertLast(o)`
`pop()`	`removeLast()`

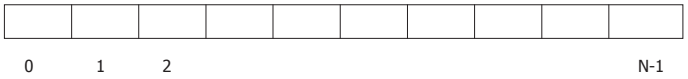
Metody fronty	Odpovídající metody DQ
`size()`	`size()`
`isEmpty()`	`isEmpty()`
`front()`	`first()`
`enqueue(o)`	`insertLast(o)`
`dequeue()`	`removeFirst()`

Př. Posloupnost operací s Vektorem a jejich výsledek

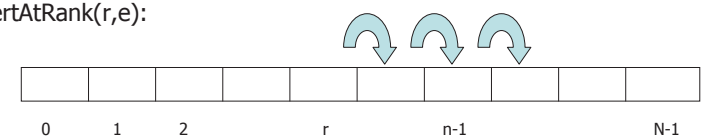
Operace	Výstup	V
insertAtRank(0,7)	-	(7)
insertAtRank(0,4)	-	(4,7)
elemAtRank(1)	7	(4,7)
insertAtRank(2,2)	-	(4,7,2)
elemAtRank(3)	chyba	(4,7,2)
removeAtRank(1)	7	(4,2)
insertAtRank(1,5)	-	(4,5,2)
insertAtRank(1,3)	-	(4,3,5,2)
insertAtRank(4,9)	-	(4,3,5,2,9)
elemAtRank(2)	5	(4,3,5,2,9)

- Implementace Vektoru polem pevné délky

V:



insertAtRank(r,e):



1. Vytvoření místa pro nový prvek přesunem prvků
2. Vložení prvku **e** na pozici **r**

```
public void insertAtRank(int r, Object e) {                    // O(n) time
        if (size == capacity) {                                 // An overflow
          capacity *= 2;
          Object[] b = new Object[capacity];
          for (int i=0; i&ltsize; i++)
                  b[i] = a[i];
          a = b;
        }
        for (int i=size-1; i>=r; i--)       // Shift elements up
          a[i+1] = a[i];
        a[r] = e;
        size++;
      }
    }
```

Implementace vektoru polem proměnné délky

```
public class ArrayVector {
   private Object[] a;                  // Array storing the elements of the vector
   private int capacity = 16;           // Length of array a
   private int size = 0;                // Number of elements stored in the vector
   // Constructor
   public ArrayVector() { a = new Object[capacity]; }       // O(1) time
   // Accessor methods
   public Object elemAtRank(int r) { return a[r]; }         // O(1) time
   public int size() { return size; }
   public boolean isEmpty() { return size() == 0; }         // O(1) time
   // Modifier methods
   public Object replaceAtRank(int r, Object e) {           // O(1) time
     Object temp = a[r];
     a[r] = e;
     return temp;
   }
   public Object removeAtRank(int r) {                                  // O(n) time
     Object temp = a[r];
     for (int i=r; i&ltsize-1; i++)      // Shift elements down
       a[i] = a[i+1];
     size--;
     return temp;
   }
```

ADT Seznam

- Seznam je lineární posloupnost prvků, které jsou propojeny ukazateli (pointery). Prvek se do seznamu vkládá na určitou pozici (obdoba indexu u vektoru).

Metody pro práci se seznamem :

`first()`: Vrací odkaz na první prvek seznamu S, je-li S prázdný nastává chyba.
Vstup: Není **Výstup**: Pozice (ukazatel na objekt)

`last()`: Vrací odkaz na poslední prvek seznamu S, je-li S prázdný nastává chyba.
Vstup: Není **Výstup**: Pozice (ukazatel na objekt)

`before(p)`: Vrací odkaz na prvek před prvkem na pozici **p**, chyba je-li p ukazatel na první prvek.
Vstup: Pozice **Výstup**: Pozice

`after(p)`: Vrací odkaz na prvek před prvkem na pozici **p**, chyba je-li p ukazatel na první prvek.
Vstup: Pozice **Výstup**: Pozice

`isFirst(p)`: Vrací true pokud **p** ukazuje na první prvek
Vstup: Pozice **Výstup**: Boolean

`isLast(p)`: Vrací true pokud **p** ukazuje na poslední prvek
Vstup: Pozice **Výstup**: Boolean

`replaceElement(p,o)`: Zamění prvek v pozici **p** prvkem **o** a vrátí původní prvek
Vstup: Pozice **p** Objekt **o** **Výstup**: Objekt

`swapElements(p,q)`: Zamění prvek v pozici **p** s prvkem v pozici q
Vstup: Pozice **p**,**q** **Výstup**: Není

`insertFirst(o)`: Vloží prvek **o** na začátek seznamu
Vstup: Objekt **o** **Výstup**: Pozice – ukazatel na nově vložený prvek

`insertLast(o)`: Vloží prvek **o** na konec seznamu
Vstup: Objekt **o** **Výstup**: Pozice – ukazatel na nově vložený prvek

`insertBefore(p,o)`: Vloží prvek **o** před prvek na pozici **p**, chyba je-li **p** ukazatel na první prvek
Vstup:Pozice **p,** Objekt **o** **Výstup**: Pozice – ukazatel na nově vložený prvek

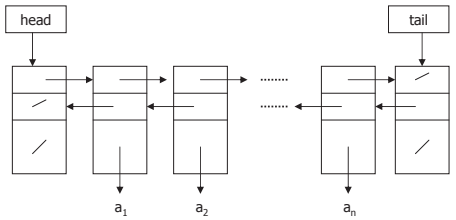
`insertAfter(p,o)`: Vloží prvek **o** za prvek na pozici **p**, chyba je-li **p** ukazatel na první prvek
Vstup:Pozice **p,** Objekt **o** **Výstup**: Pozice ukazatel na nově vložený prvek

`remove(p,o)`: Odstraní prvek v pozici **p** ze seznamu první prvek
Vstup:Pozice **p,** Objekt **o** **Výstup**: Pozice ukazatel na nově vložený prvek

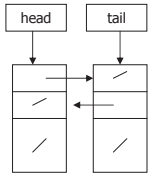
Př. Posloupnost operací se seznamem a jejich výsledek

Operace	Výstup	V
insertFirst(8)	$p_1(8)$	(8)
insertAfter(p_1,5)	$p_2(5)$	(8,5)
insertBefore(p_2,3)	$p_3(3)$	(8,3,5)
insertFirst(9)	$p_4(9)$	(9,8,3,5)
before(p_3)	$p_1(8)$	(9,8,3,5)
last()	$p_2(5)$	(9,8,3,5)
remove(p_4)	9	(8,3,5)
swapElements(p_1, p_2)	-	(5,3,8)
replaceElement(p_3,7)	3	(5,7,8)
insertAfter(first(),2)	$p_5(2)$	(5,2,7,8)

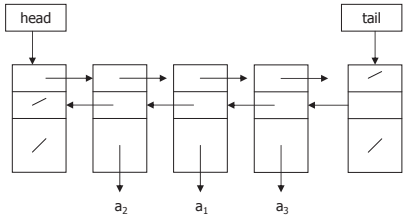
- **Implementace seznamem obousměrně zřetězených prvků**



Prázdný seznam :



insertFirst(a_1), insertFirst(a_2),insertLast(a_3) :



Metody třídy java.util.LinkedList

Method Summary	
void	**add**(int index, Object element) Inserts the specified element at the specified position in this list.
boolean	**add**(Object o) Appends the specified element to the end of this list.
boolean	**addAll**(Collection c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean	**addAll**(int index, Collection c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**addFirst**(Object o) Inserts the given element at the beginning of this list.
void	**addLast**(Object o) Appends the given element to the end of this list.
void	**clear**() Removes all of the elements from this list.

Object	**clone**() Returns a shallow copy of this LinkedList.
boolean	**contains**(Object o) Returns true if this list contains the specified element.
Object	**get**(int index) Returns the element at the specified position in this list.
Object	**getFirst**() Returns the first element in this list.
Object	**getLast**() Returns the last element in this list.
int	**indexOf**(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
int	**lastIndexOf**(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
ListIterator	**listIterator**(int index) Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
Object	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(Object o) Removes the first occurrence of the specified element in this list.
Object	**removeFirst**() Removes and returns the first element from this list.
Object	**removeLast**() Removes and returns the last element from this list.
Object	**set**(int index, Object element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
Object[]	**toArray**() Returns an array containing all of the elements in this list in the correct order.
Object[]	**toArray**(Object[] a) Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.

Kolekce v javě

- jsou to objekty tříd z balíku java.util
- slouží k uchování většího množství (předem neznámého) objektů

Výhody používání kolekcí:

- snižují množství programového kódu (datové struktury a algoritmy jsou již hotovy)
- zrychlují program a umožňují jeho vyladění (třídy kolekcí jsou pečlivě naprogramovány a optimalizovány aby výsledný kód byl co nejrychlejší)
- zvyšují čitelnost a přehlednost programu
- uchovávají „neomezené" předem neznámé množství objetů libovolného typu

Nevýhody :

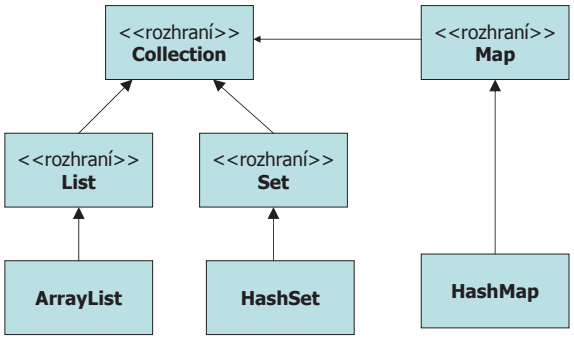
- do kolekcí nelze vkládat primitivní datové typy (int, float apod.), primitivní dat. Typ je nutné vkládat pomocí obalovací třídy (Integer, Float, atd.)
- kolekce jsou obvykle pomalejší než obyčejná pole (když víme kolik budeme vkládat objektů, raději používat pole)

Základní typy kolekcí a rozhraní:

- Collection – rozhraní pro práci se skupinou objektů (elementy, prvky)
- List – rozhraní k seznamům. Představuje uspořádanou kolekci
  - elementy přístupny pomocí indexů
  - kolekce může obsahovat stejné (duplicitní) elementy
  - uživatel určuje pořadí elementů

- ArrayList – pole proměnné délky, někdy nazývané seznam. Je to třída implementující metdy rozhraní Collection. Funkcí odpovídá zhruba ADT Vektor (nahrazuje třídu Vector z JDK1.1)
  - připomíná klasické pole – přístup k prvkům přes indexy
  - prvky jsou udržovány v určitém pořadí
  - není nutné na počátku definovat počet prvků - pole je „nafukovací"
  - v poli mohou být duplicitní prvky
- Set – rozhraní k množinám,
  - neuspořádaná kolekce,
  - nedá se využít pořadí elementů
  - nemohou obsahovat duplicitní elementy
- HashSet – množina. Třída implementující metody Set
  - obsahuje pouze unikátní prvky
  - pro přístup k prvkům nelze použít index, přístup pouze přes iterátor (implementace hashovací tabulkou – z hlediska uživatele třídy není důležité)

- HashMap – mapa. Třída implementující metody rozhraní Map
  - ukládá dvojice prvků klíč-hodnota
  - vyhledává se podle klíče – vybavuje se hodnota odpovídající hledanému klíči



Společné metody seznamů a množin

Rozhraní Collection

- metody pro plnění kolekce
  - ***boolean add(Object o)*** – vložení jednoho prvku
  - ***boolean addAll(Collection c)*** – vložení všech prvků z jiné kolekce
- metody pro ubírání z kolekce
  - ***void clear()*** – odstranění všech prvků z kolekce
  - ***booelan remove(Object o)*** – odstranění jednoho prvku z kolekce, je-li jich více odstraní se libovolný z nich
  - ***boolean removeAll(Collection c)*** – odstranění všech prvků nacházejících se současně v jiné kolekci
  - ***boolean retainAll(Collection c)*** – ponechání pouze prvků nacházejících se současně v jiné kolekci
- ***dynamické vlastnosti kolekcí***
  - ***int size()*** – vrací aktuální počet prvků kolekce
  - ***boolean isEmpty()*** – test na prázdnou kolekci
  - ***boolean contains(Object o)*** – test zda je je daný prvek obsažen v kolekci
  - ***boolean cointainsAll(Collection c)*** – test, zda jsou všechny prvky kolekce c obsaženy v dané kolekci
- získání přístupového objektu
  - ***Iterator iterator()*** – vrací objekt typu iterátor – vhodný pro jeden průchod kolekcí
- převod kolekce na běžné pole
  - ***Object[] toArray()*** – převod na pole typu Object
  - ***Object[] toArray(Object[] a)*** – převod na pole konkrétního typu

Rozhraní List

- změny v kolekci
  - ***void add(int index, Object o)*** – přidání prvku; prvky se stejným a vyšším indexem posunuty o jeden výše
  - ***Object set(int index, Object o)*** – změna prvku na daném indexu; prvky s vyšším indexem se indexy nemění
  - ***Object remove(int index)*** – odstranění prvku; prvky s vyšším indexem budou posunuty o jeden níže
- získání obsahu kolekce
  - ***Object get(int index)*** – vrátí prvek s daným indexem (ponechá ho v kolekci)

int indexOf(Object o) – vrátí index prvního nalezeného prvku shodného s parametrem metody o (hledá se od počátku), nebo -1 není-li prvek v kolekci

int lastIndexOf(Object o) – vrátí index posledního nalezeného prvku (hledá se od konce)

List subList(int startIndex, int endIndex) – vrátí podseznam ve kterém budou prvky od startIndex do endIndex-1 včetně

Třída ArrayList

implementuje metody rozhraní Collection a List, nahrazuje klasické pole a zhruba odpovídá ADT Vector

Metody Vector	Odpovídající metody ArrayList
`elementAtRank(r)`	`get(r)`
`replaceAtRank(r,o)`	`set(r,o)`
`insertAtRank(r,o)`	`add(r,o)`
`removeAtRank(r)`	`remove(r)`

Pozor : ArrayList má kromě velikosti také kapacitu, která je větší než aktuální velikost. Po překročení kapacity se alokuje nová kolekce, která má kapacitu 1.5 násobek původní kapacity. Počáteční kapacita se dá nastavit pomocí konstruktoru ArrayList(int pocatecniKapacita).

Nastavení počáteční kapacity je důležité kvůli rychlosti (viz Herout).

Implementace zásobníku a fronty pomocí třídy LinkedList

```
import java.util.*;

public class Zasobnik {
  private LinkedList zasob = new LinkedList();

  public void push(Object o) {
    zasob.addFirst(o);
  }

  public Object pop() {
    return zasob.removeFirst();
  }

  public Object top() {
    return zasob.getFirst();
  }
```

```
  public static void main(String[] args) {
     Zasobnik z = new Zasobnik();
     z.push("prvni");
     z.push("druhy");
     z.push("treti");
     System.out.println(z.top());
     System.out.println(z.pop());
     System.out.println(z.pop());
     System.out.println(z.pop());
   }
}
```

```
import java.util.*;

public class Fronta {
  private LinkedList fronta = new LinkedList();

  public void enqueue(Object o) {
    fronta.addLast(o);
  }

  public Object dequeue() {
    return fronta.removeFirst();
  }

public static void main(String[] args) {
    Fronta f = new Fronta();
    f.enqueue("prvni");
    f. enqueue("druhy");
    f. enqueue("treti");
    System.out.println(f. dequeue());
    System.out.println(f. dequeue());
    System.out.println(f. dequeue());
  }
}
```

Iterátory

- slouží pro postupný průchod kolekcí
- průchod pomocí iterátorů dovolují všechny kolekce (narozdíl od indexace)
- iterátor je něco jako zobecnění indexu
- každá třída kolekcí má metodu ***Iterator*** *iterator()* pomocí které lze vytvořit iterátor pro danou kolekci.

Metody iterátoru

iterátor umožňuje tři aktivity:

- zjistit zda v kolekci existuje další prvek
  boolean hasNext()
- přesunout se přes tento prvek a vrátit jej
  Object next()

  metoda nekontroluje zda existuje další prvek, pokud ne vyhodí výjimku NoSuchElementException

- zrušit aktuální prvek, odkazovaný předchozím next()
  void remove()

  tato metoda nevrací rušený prvek, a lze ji použít až po použití next(), jinak je vyhozena výjimka IllegalStateException.

- pokud vygenerujeme nový iterátor, tak ukazuje před první prvek kolekce. První použití metody next() způsobí, že iterátor přejde na první prvek a vrátí na něj odkaz. Při dalším volání *next()* iterátor přechází na následující prvek a vrací referenci na něj.
- objekt iterátoru se dá použít jen pro jeden průchod kolekcí, pro další průchod se musí vytvořit nový objekt iterátoru

```
import java.util.*;

class Hruska {
  private int cena;
  Hruska(int cena) { this.cena = cena; }
  public String toString() { return "" + cena; }
  public void tisk() { System.out.print(cena + ", "); }
}
```

```
public class IteratorZakladniPouziti {
  public static void main(String[] args) {
    ArrayList kosHrusek = new ArrayList();
    for (int i = 0;  i < 10;  i++) {
      kosHrusek.add(new Hruska(i + 20));
    }

    for (Iterator it = kosHrusek.iterator();  it.hasNext(); ) {
      System.out.print(it.next() + ", ");
    }
    System.out.println();

    Iterator it = kosHrusek.iterator();
    while (it.hasNext()) {
      ((Hruska) it.next()).tisk();
    }
    System.out.println();
  }
}
```

Vypíše:

20, 21, 22, 23, 24, 25, 26, 27, 28, 29
20, 21, 22, 23, 24, 25, 26, 27, 28, 29

Př. Použití metody remove()

```
import java.util.*;

class Hruska {
  private int cena;
  Hruska(int cena) { this.cena = cena; }
  public String toString() { return "" + cena; }
  public int getCena() { return cena; }
}

public class IteratorRemove {
  public static void main(String[] args) {
    ArrayList kosHrusek = new ArrayList();
    for (int i = 0;  i < 10;  i++) {
      kosHrusek.add(new Hruska(i + 20));
    }
```

```
for (Iterator it = kosHrusek.iterator();  it.hasNext(); ) {
      Hruska h = (Hruska) it.next();
      System.out.print(h + ", ");
      if (h.getCena() % 2 == 0)
        it.remove();
    }
    System.out.println();

    for (Iterator it = kosHrusek.iterator();  it.hasNext(); ) {
      System.out.print(it.next() + ", ");
    }
    System.out.println();
  }
}
```

Vypíše:

20, 21, 22, 23, 24, 25, 26, 27, 28, 29
21, 23, 25, 27, 29

Třída ListIterator

obsahuje kromě metod třídy Iterator ještě metody, které umožňují průchod seznamem od konce k jeho počátku

boolean hasPrevious(), Object previous()

třídu ListIterator mohou využít pouze objekty odvozené od List

Př. Průchod seznamem od posledního prvku k prvnímu

```
import java.util.*;

public class TestListIterator {
  public static void main(String[] argv) {
    String[] tmp = {"1", "2", "3", "4", "5"};
    List l = new ArrayList(Arrays.asList(tmp));
    System.out.println("Seznam:          " + l);
```

```
System.out.print("Seznam pozpatku: [");
    for (ListIterator i = l.listIterator(l.size());  i.hasPrevious(); ) {
      String s = (String) i.previous();
      System.out.print(s + ", ");
    }
    System.out.println("]");
  }
}
```

Vypíše:

```
Seznam:          [1, 2, 3, 4, 5]
Seznam pozpatku: [5, 4, 3, 2, 1]
```

Před použitím takto specializovaného iterátoru je třeba zvážit, zda využijeme jeho výhody, protože použijeme-li ho, nebude v budoucnu možné zaměňovat jednotlivé typy kolekcí.

ADT Strom

- Strom je datová struktura, která uchovává objekty (prvky) hierarchicky ve vztahu předchůdce-následovník (otec-syn)
- Stromy patří mezi často používané datové struktury v mnoha oblastech computer science
- Příklady použití :
  - reprezentace znalostí, stavového prostoru v umělé inteligenci
  - popis scény v oblasti zpracování a analýza obrazu, počítačová grafika
  - vyhledávací stromy v databázových systémech
  - rozhodovací stromy – expertní systémy
  - organizace adresářů a souborů v souborovém systému OS,
  - komprese dat (Hufmannovy kódovací stromy, fraktálová komprese)
  - atd.

Definice stromu

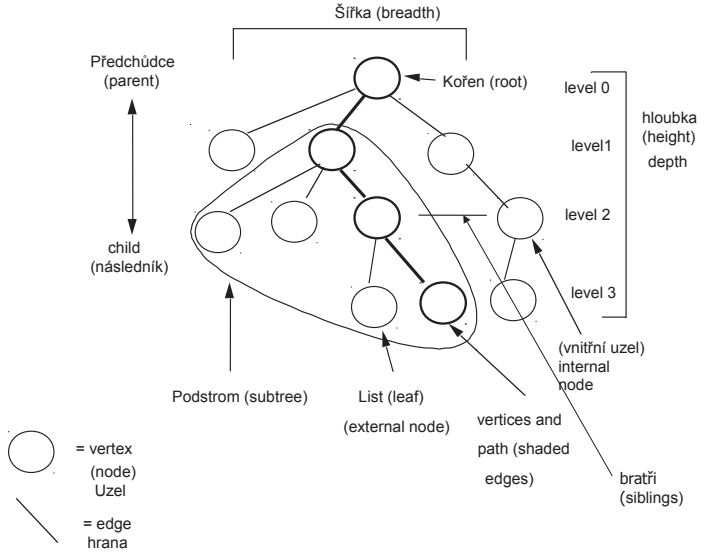
Def. 1. : Strom T je konečná množina jednoho nebo více prvků (uzlů), z nichž jeden je označen jako ***r*** kořen (root) a zbývající uzly jsou rozděleny do $n \geq 0$ disjunktních podmnožin $T_1, T_2, \ldots T_k$, které jsou také stromy a jejichž kořeny $r_1, r_2, \ldots, r_k$ jsou následníky kořene ***r*** .

Def. 2 : Strom je graf, ve kterém existuje pouze jedna cesta z uzlu ***r*** (kořene), do kteréhokoliv dalšího uzlu grafu (neobsahuje cykly)

Def. 3:

- Strom je prázdný,
- (nebo) strom obsahuje jediný uzel (kořen)
- (nebo) strom obsahuje kořen spojený s jedním nebo více podstromy

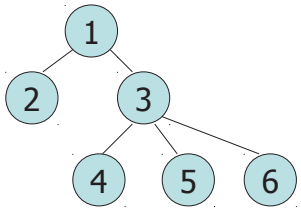
Základní terminologie

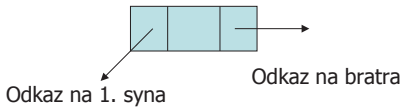


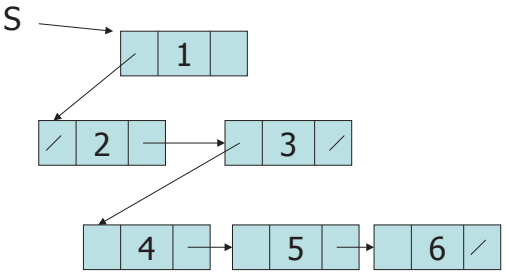
- **Kořen** - uzel který nemá předchůdce, ve stromu může být pouze jediný kořen
- **Listy** (vnější uzly) – uzly které nemají žádného následníka
- **Vnitřní uzly** – uzly které mají alespoň jednoho následníka
- **Cesta** – je-li $n_1, n_2, \ldots n_k$ množina uzlů ve stromu takových, že n_i je předchůdce n_{i+1}, pro $1 \leq i \leq k$, pak se tato množina nazývá cesta z uzlu n_1 do n_k
- **Délka cesty** – počet hran, které spojují uzly cesty
- **Hloubka uzlu** – délka cesty od kořene do uzlu
- **Výška stromu** – délka cesty od kořene k nejhlubšímu uzlu (největší hloubka)

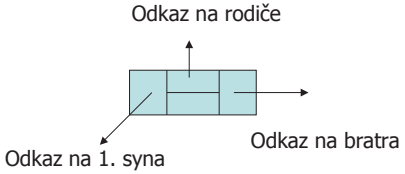
Typy stromů a implementace

- Obecný strom - vnitřní uzly stromu mohou mít libovolný počet následníků. Implementuje se výhradně jako dynamická struktura – seznam zřetězených prvků.

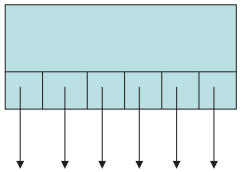






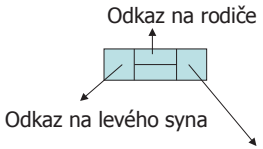


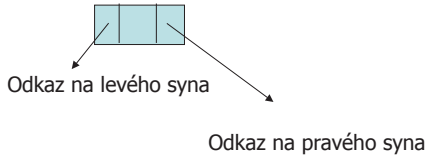
- n-ární strom – vnitřní uzly stromu obsahují maximálně n následníků. Implementuje se obvykle jako seznam zřetězených prvků, kde každý uzel stromu obsahuje pole ukazatelů na následníky (syny)



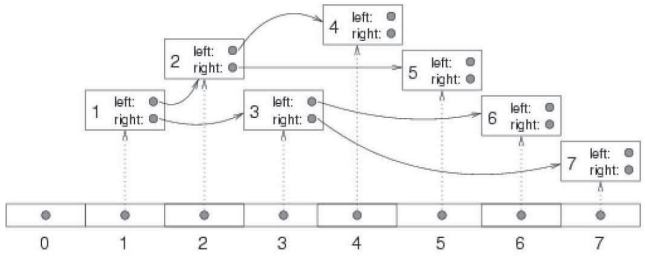
Odkazy na syny

- binární strom – speciální případ n-árního stromu, každý vnitřní uzel může mít nejvýše dva následníky (syny). Implementuje se jako seznam zřetězených prvků, někdy jako pole.





- Implementace binárního stromu jako vektor

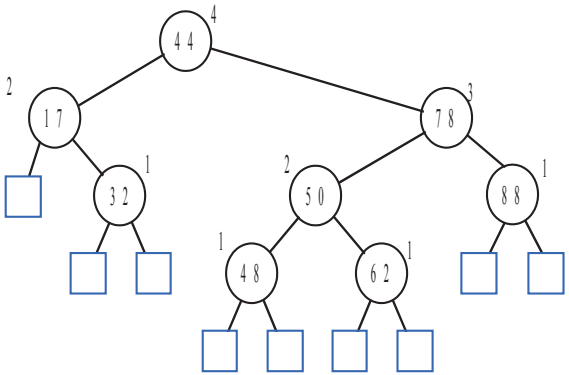


AVL strom

Adelson-Velskii, Landis

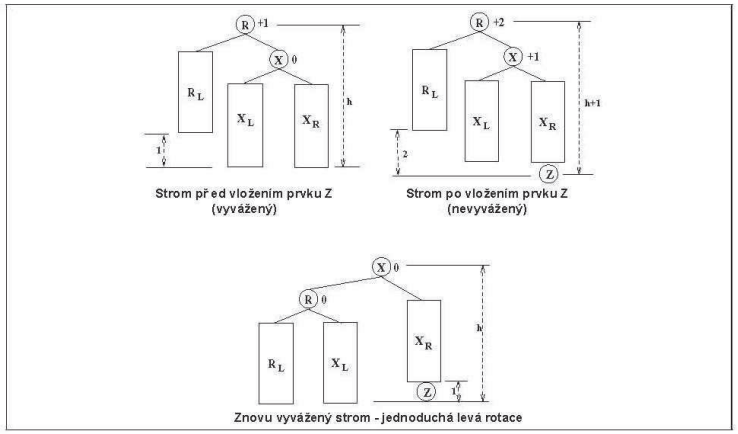
(vyvážený strom)

AVL strom je výškově vyvážený binární vyhledávací strom, pro který platí, že pro libovolný vnitřní uzel stromu se výška levého a pravého syna liší nejvýše o 1

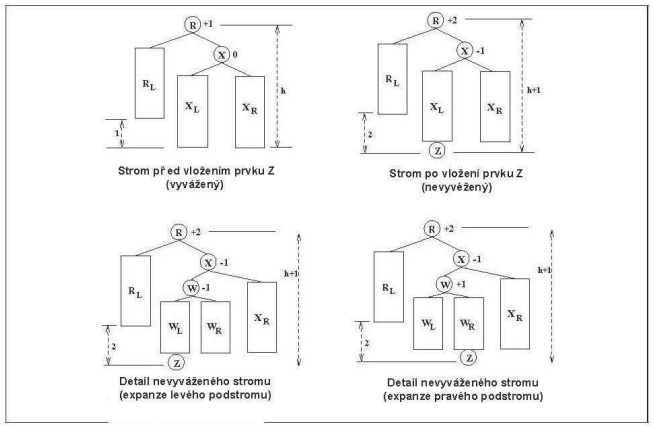


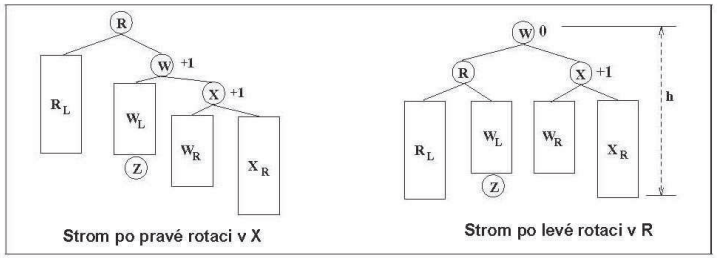
- Vyváženost AVL stromu se kontroluje po každé operaci vložení a zrušení prvku, v případě že je vyváženost porušena, provádí se opětovné vyvážení pomocí jedné popř. několika rotací v jednotlivých částech stromu.
- Implementace je obdobná jako u BVS, datová struktura pro uzel stromu je doplněna o celočíselnou proměnnou reprezentující stupeň vyváženosti uzlu, který může nabývat následujících hodnot:
  - 0 – oba podstromy jsou stejně vysoké
  - 1 – pravý podstrom je o 1 vyšší
  - 2 – pravý podstrom je o 2 vyšší
  - -1 – levý podstrom je o 1 vyšší
  - -2 – levý podstrom je o 2 vyšší
- Rotace :
  - Jednoduchá pravá (levá) – používáme pokud vyvažujeme přímou větev, tj. jsou-li znaménka stupně vyváženosti stejná
  - Dvojitá pravá (levá) – používá se tehdy pokud nejde použít jednoduchá rotace – vyvažujeme-li „zalomenou" větev.

- **Případ 1**: Pravý syn bude mít po operaci vložení hodnotu +1
  - Pokud měl uzel R před vložením prvku do pravého podstromu stupeň vyvážení +1, a pravý syn X uzlu R hodnotu 0 bude mít X po vložení prvku Z do pravého podstromu X_R hodnotu +1 a uzel R hodnotu +2. K opětovnému vyvážení použijeme levou rotaci v uzlu R



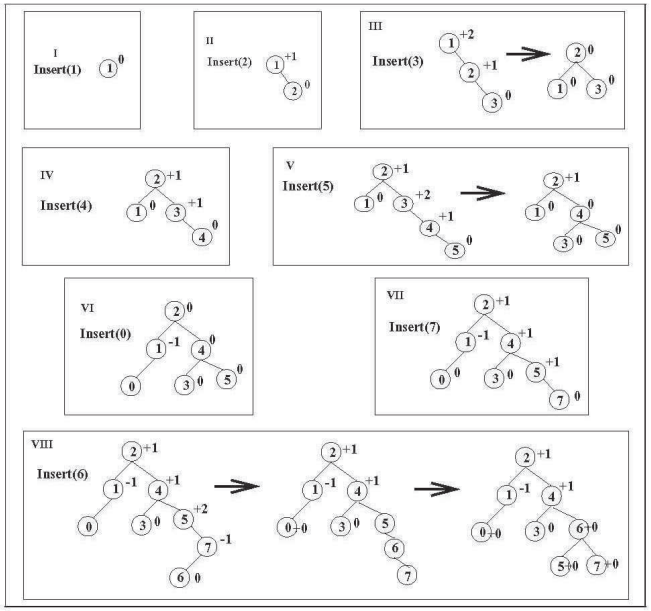
- Případ 2: Pravý syn bude mít po operaci vložení hodnotu -1
  - Stupeň vyvážení pravého podstromu uzlu R je v tomto případě -1, protože Z je vložen do X_L. Stupeň vyváženosti uzlu R se změní z +1 na +2. Ke opětovnému vyvážení je nutné provést 2 rotace, první je pravá rotace kolem X, druhá levá kolem R. Výška stromu zůstává po vyvážení stejná jako před vyvážením.





Algoritmus vložení prvku X do AVL stromu S

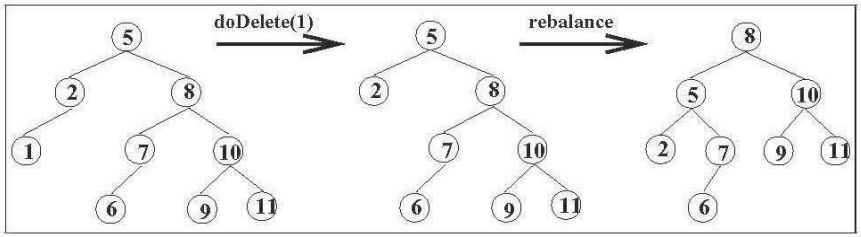
1. Vyhledávání X v S, dokud není zřejmé že prvek ve stromu S neexistuje
2. Zařazení X v místě, kde se ukončilo vyhledávání
3. Zpětný přepočet stupňů vyvážení S od přidaného uzlu vzhůru (od listu ke kořenu). V případě že došlo k rozvážení S (v místě, kde se poprvé vyskytne stupeň vyváženosti s hodnotou +2) se vykoná následující bod
4. Vyvážení S jednou ze dvou výše uvedených rotací



Vložení prvků {1,2,3,4,5,0,7,6} do AVL stromu

Zrušení prvků v AVL stromu

- Rušení prvků je podobné jako u BVS stromu, po zrušení prvku je nutné provést kontrolu stupně vyváženosti uzlů směrem ke kořeni a v případě že je to nutné znovu vyvážit strom pomocí jednoduchých popř. dvojnásobných rotací



B-Strom

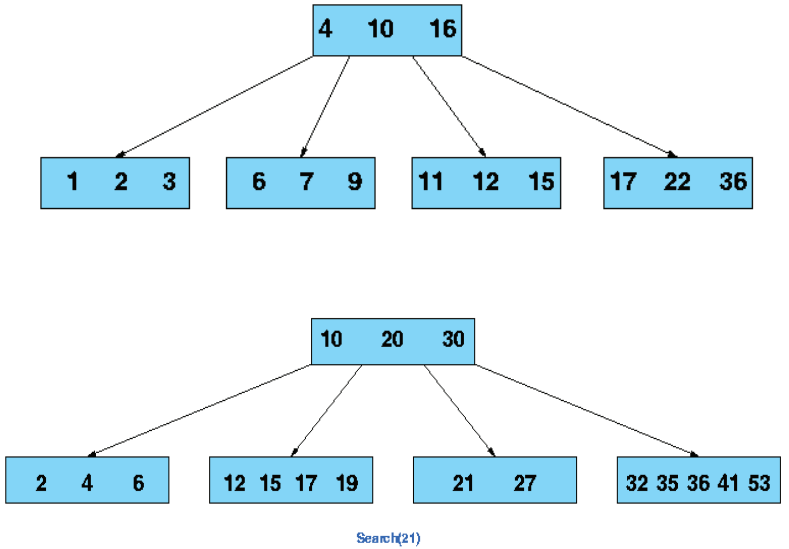
B-stromy

B-stromy jsou vyvážené stromy, které jsou optimalizovány pro případ, kdy je část stromu, popř. celý strom uložen na vnější paměti (např. magnetický disk). Vzhledem k tomu, že přístup na diskovou paměť je časově náročný, B-strom je navržen tak, aby optimalizoval (a minimalizoval) počet přístupů do vnější paměti.

Definice B-stromu

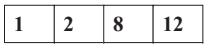
- B-strom řádu *m* je strom, kde každý uzel má maximálně m následníků a ve kterém platí:
  1. Počet klíčů v každém vnitřním uzlu, je o jednu menší než je počet následníků (synů)
  2. Všechny listy jsou na stejné úrovni (mají stejnou hloubku)
  3. Všechny uzly kromě kořene mají nejméně $\lceil \frac{m}{2} \rceil$ následníků($\lceil \frac{m}{2} \rceil$ -1 klíčů)
  4. Kořen je buď to list, nebo má od 2 do m následníků
  5. Žádný uzel neobsahuje více než m následníků (m-1 klíčů)

Příklad B-Stromu



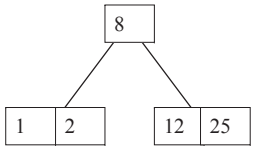
Vytvoření B-stromu

- Předpokládejme, že začínáme s prázdným B-stromem a ukládáme klíče v následujícím pořadí: **1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45**
- Chceme vytvořit B-strom řádu m=5, tj.každý uzel (kromě kořene) obsahuje nejméně 2 klíče a nejvýše 4 klíče.
- První 4 klíče :

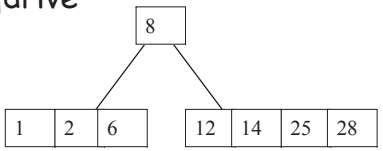


- Vložení páté položky (klíč 25) poruší podmínku 5 (dojde k tzv. přeplnění stránky)
- Přeplněná stránka se rozdělí na dvě, prostřední prvek se přesune do nadřazené stránky

Vytvoření B-stromu (pokračování)

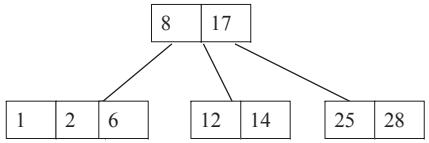


Další položky 6, 14, 28 budou vloženy do listů (listy se obsazují nejdříve

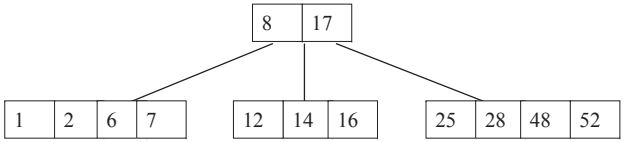


Vytváření B-stromu

Vložení 17 do pravé stránky způsobí přeplnění, stránka se rozdělí podle prostředního klíče a ten se přesune do kořene

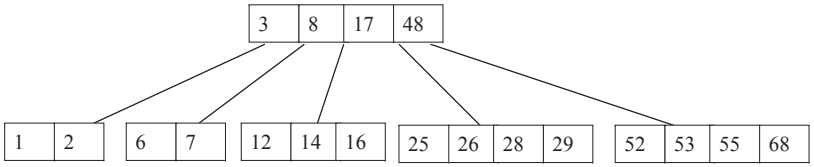


7, 52, 16, 48 se opět přidají do listů

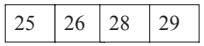


Vytváření B-stromu

Vložení 68 opět způsobí přeplnění stránky vpravo, klíč 48 se přesune do kořene, 3 přeplní levou stránku a po rozdělení přechází do kořene; 26, 29, 53, 55 jsou vloženy do listů

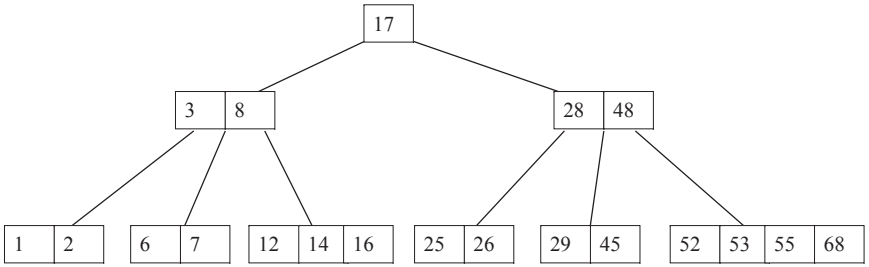


45 přeplní stránku



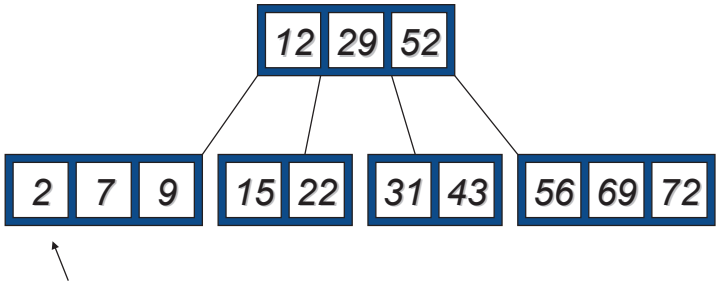
a klíč 28 se přesune do kořene, kde způsobí přeplnění a rozdělení kořenové stránky

Vytváření B-stromu



Rušení prvků B-stromu - rušení v listech bez podtečení velikosti stránky

Předpokládejme B-strom 5 řádu...



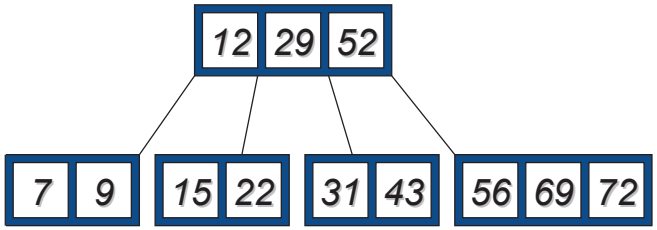
Zrušení klíče 2: Jelikož ve stránce je dostatečné množství klíčů, Dojde pouze ke zrušení hodnoty 2 v listové stránce

Vložení prvku do B-stromu (shrnutí)

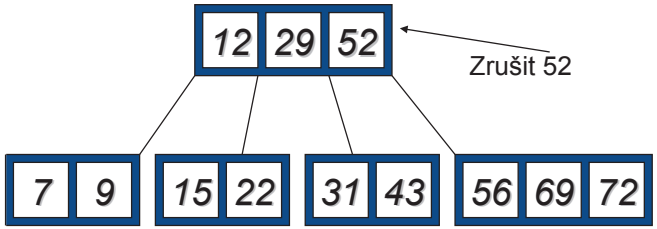
- Nový prvek se vždy vkládá do listové stránky, ve stránce se klíče řadí podle velikosti.
- Pokud dojde k přeplnění listové stránky, stránka se rozdělí na dvě a prostřední klíč se přesune do nadřazené stránky (pokud nadřazená stránka neexistuje, tak se vytvoří)
- Pokud dojde k přeplnění nadřazené stránky předchozí postup se opakuje dokud nedojde k zařazení nebo k vytvoření nového kořene.

Rušení prvků B-stromu - rušení v listech bez podtečení velikosti stránky

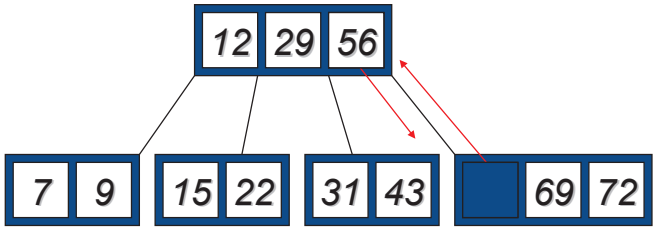
Předpokládejme B-strom 5 řádu...



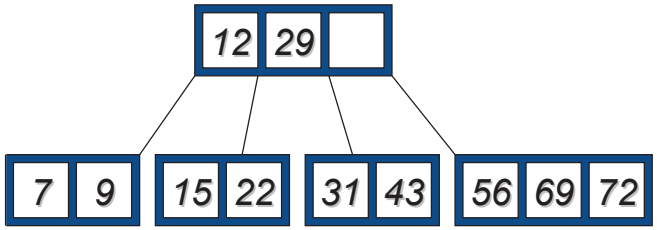
Rušení prvků B-stromu - rušení pvku v ostatních stránkách (kromě listů) bez podtečení velikosti stránky



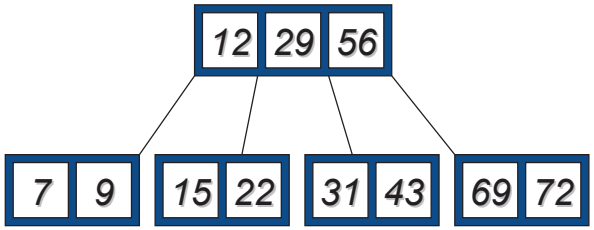
Rušení prvků B-stromu - rušení pvku v ostatních stránkách (kromě listů) bez podtečení velikosti stránky



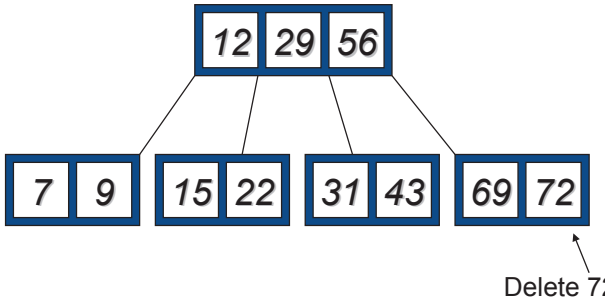
Rušení prvků B-stromu - rušení pvku v ostatních stránkách (kromě listů) bez podtečení velikosti stránky



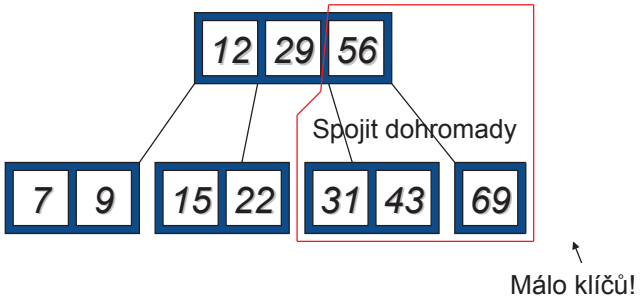
Rušení prvků B-stromu - rušení pvku v ostatních stránkách (kromě listů) bez podtečení velikosti stránky



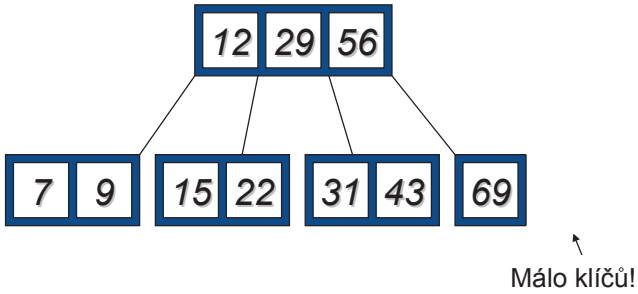
Rušení prvků ve stránce s minimálním počtem klíčů – sousední stránky obsahují také minimální počet klíčů



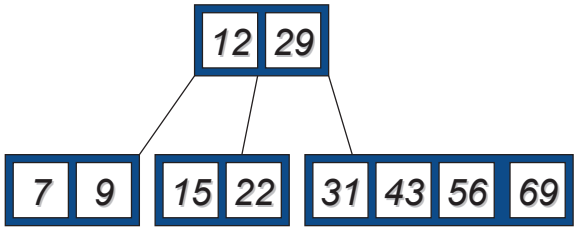
Rušení prvků ve stránce s minimálním počtem klíčů – sousední stránky obsahují také minimální počet klíčů



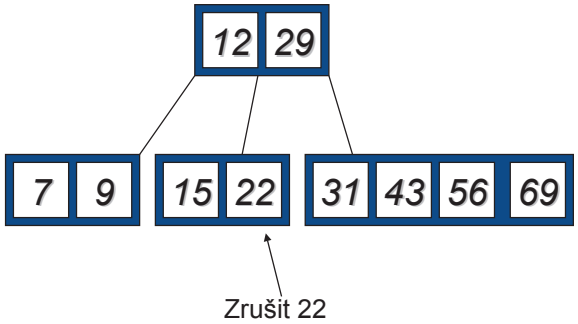
Rušení prvků ve stránce s minimálním počtem klíčů – sousední stránky obsahují také minimální počet klíčů



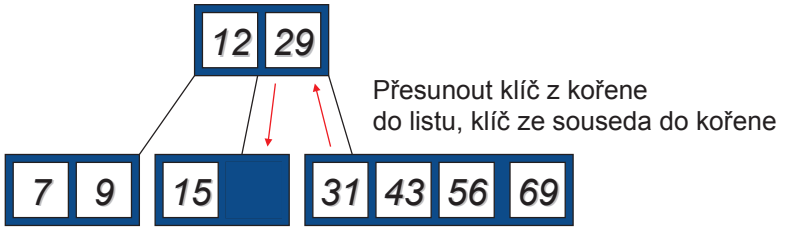
Rušení prvků ve stránce s minimálním počtem klíčů – sousední stránky obsahují také minimální počet klíčů



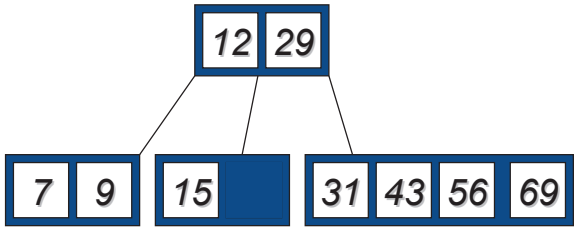
Rušení prvků ve stránce s minimálním počtem klíčů – sousední stránky obsahují dostatek klíčů



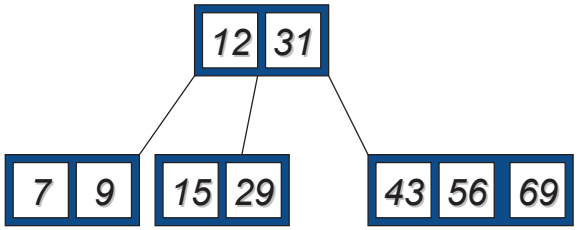
Rušení prvků ve stránce s minimálním počtem klíčů – sousední stránky obsahují dostatek klíčů



Rušení prvků ve stránce s minimálním počtem klíčů – sousední stránky obsahují dostatek klíčů



Rušení prvků ve stránce s minimálním počtem klíčů – sousední stránky obsahují dostatek klíčů



Analýza B-Stromů

- Maximální počet položek v B-stromu řádu m a výšky h:

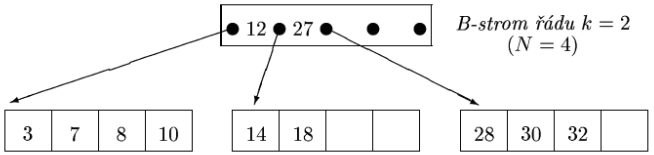
  kořen $m - 1$
  úroveň 1 $m(m - 1)$
  úroveň 2 $m^2(m - 1)$
  . . .
  úroveň h $m^h(m - 1)$

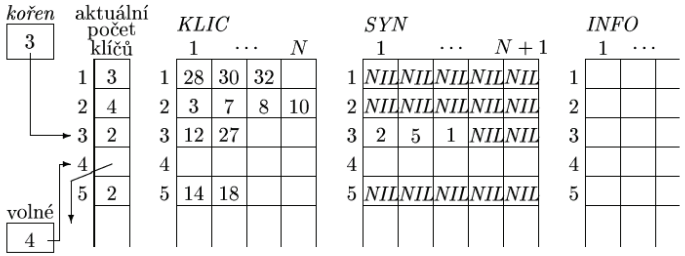
- Celkový počet položek je

  $(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$
  $[(m^{h+1} - 1)/ (m - 1)] (m - 1) = m^{h+1} - 1$

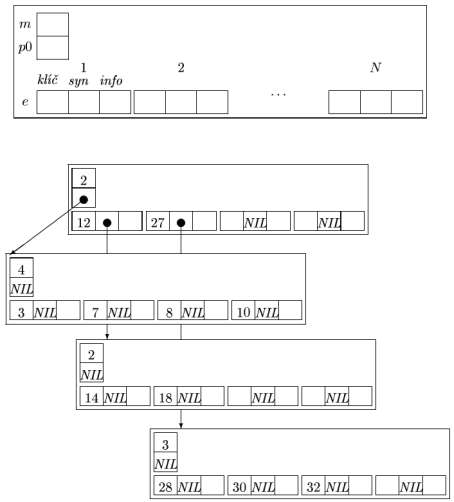
- Pokud $m = 5$ a $h = 2$ tak je celkem $5^3 - 1 = 124$ položek

Implementace B-stromu -statická





Implementace B-stromu - dynamická



Vytvoření B-stromu

B-Tree-Create(T)

```
x <- Allocate-Node()
leaf[x] <- TRUE
n[x] <- 0
Disk-Write(x)
root[T] <- x
```

Vložení prvku do B-Stromu

B-Tree-Insert(T, k)

```
r <- root[T]
if n[r] = 2t - 1
     then s <- Allocate-Node()
          root[T] <- s
          leaf[s] <- FALSE
          n[s] <- 0
          c1 <- r
          B-Tree-Split-Child(s, 1, r)
          B-Tree-Insert-Nonfull(s, k)
     else B-Tree-Insert-Nonfull(r, k)
```

Rozdělení stránky

B-Tree-Split-Child(x, i, y)

```
z <- Allocate-Node()
leaf[z] <- leaf[y]
n[z] <- t - 1
for j <- 1 to t - 1
     do keyj[z] <- keyj+t[y]
if not leaf[y]
     then for j <- 1 to t
          do cj[z] <- cj+t[y]
n[y] <- t - 1
for j <- n[x] + 1 downto i + 1
     do cj+1[x] <- cj[x]
ci+1 <- z
for j <- n[x] downto i
     do keyj+1[x] <- keyj[x]
keyi[x] <- keyt[y]
n[x] <- n[x] + 1
Disk-Write(y)
Disk-Write(z)
Disk-Write(x)
```

Vložení prvku do B-stromu

B-Tree-Insert-Nonfull(x, k)

```
i <- n[x]
if leaf[x]
     then while i >= 1 and k < keyi[x]
            do keyi+1[x] <- keyi[x]
               i <- i - 1
          keyi+1[x] <- k
          n[x] <- n[x] + 1
          Disk-Write(x)
     else while i >= and k < keyi[x]
          do i <- i - 1
          i <- i + 1
          Disk-Read(ci[x])
          if n[ci[x]] = 2t - 1
               then B-Tree-Split-Child(x, i, ci[x])
                    if k > keyi[x]
                         then i <- i + 1
          B-Tree-Insert-Nonfull(ci[x], k)
```

Vyhledávání v B-Stromu

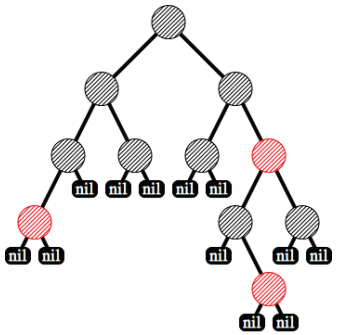
B-Tree-Search(x, k)

```
i <- 1
while i <= n[x] and k > keyi[x]
     do i <- i + 1
if i <= n[x] and k = keyi[x]
     then return (x, i)
if leaf[x]
     then return NIL
     else Disk-Read(ci[x])
          return B-Tree-Search(ci[x], k)
```

Red-Black Stromy

Binární Vyhledávací Stromy, u kterých je časová složitost operací v nejhorším případě rovná O(log n)

Vlastnosti Red-Black Stromů

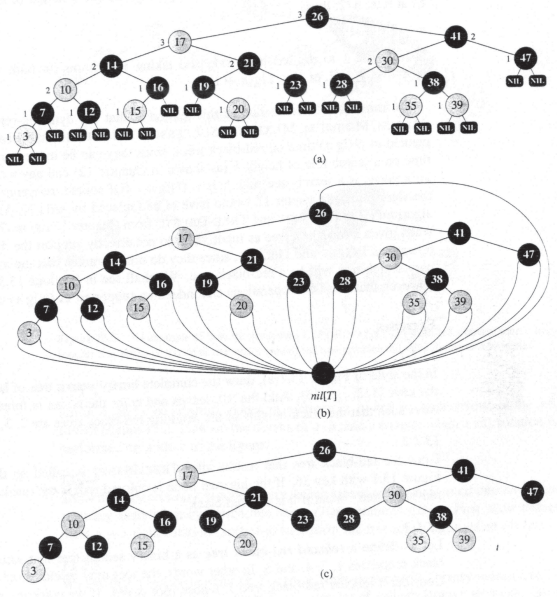


Vlastnosti Red-Black stromů

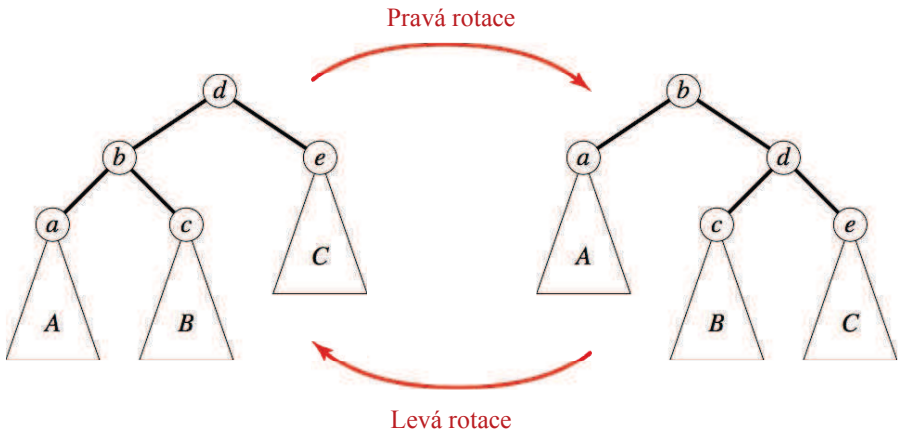
- **Každý uzel stromu je obarven červenou nebo černou barvou.**
- **Kořen stromu je obarven černě.**
- **Listy (nil) jsou černé.**
- **Červený uzel má pouze černé syny.**
- **Na kterékoliv cestě z kořene do listu leží stejný počet černých uzlů**

Černá výška (***black-height bh(x))*** uzlu ***x*** je počet černých uzlů na cestě z uzlu ***x*** (ale bez uzlu x) k listu. **Bh(T)** stromu T je rovna **bh(r),** kde r je kořen stromu.

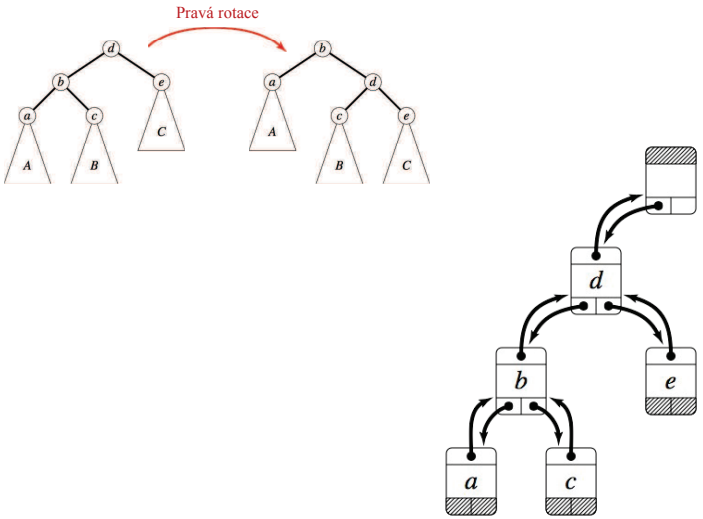
RB strom a implementace



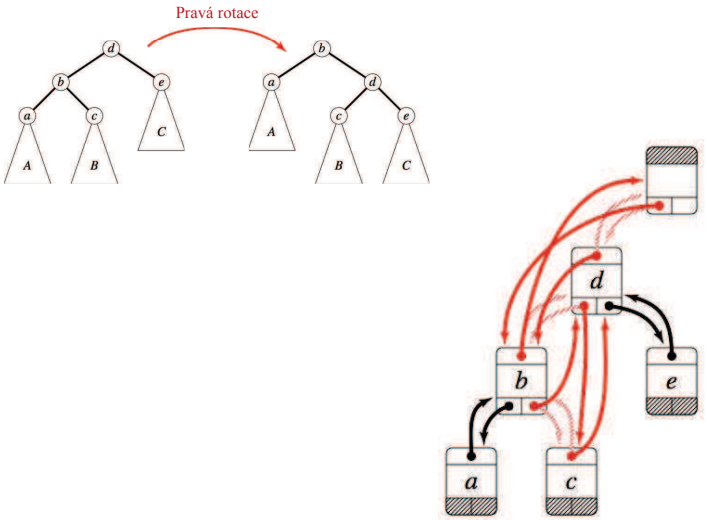
Rotace



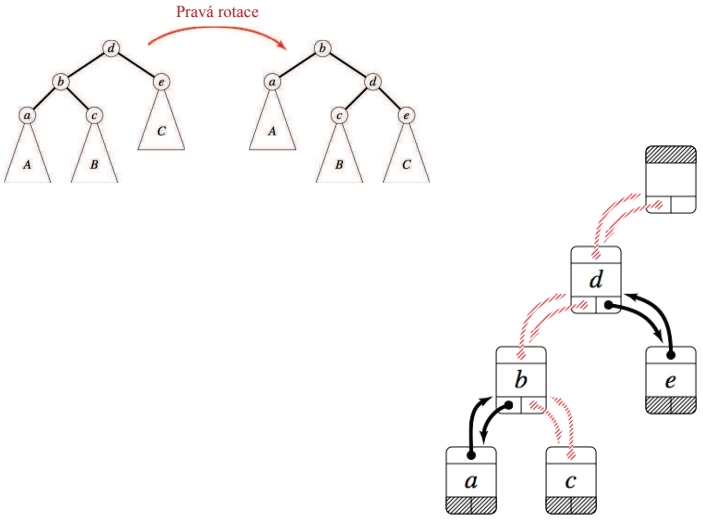
Operace rotace se provádí v konstantním čase



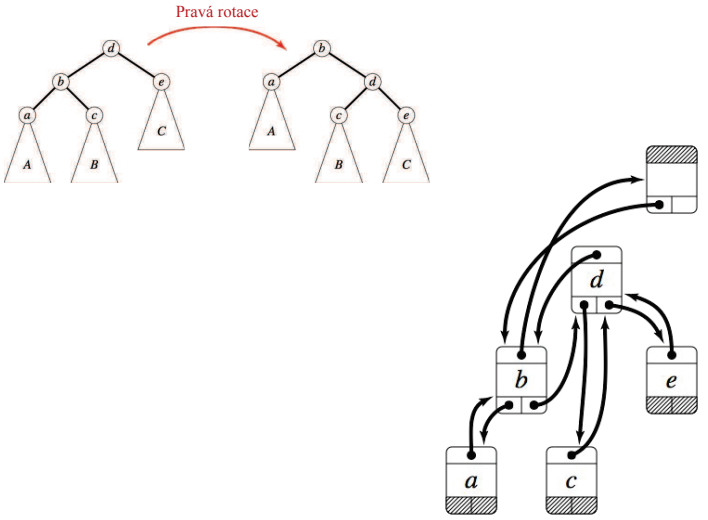
Operace rotace se provádí v konstantním čase



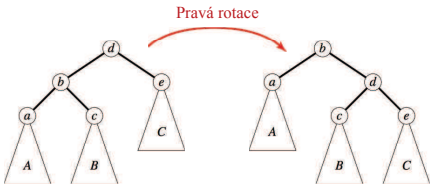
Operace rotace se provádí v konstantním čase



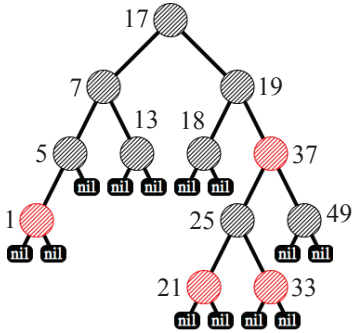
Operace rotace se provádí v konstantním čase



Operace rotace se provádí v konstantním čase



Vložení prvku do Red-Black stromu

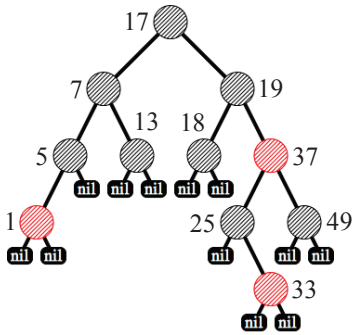


Vlastnosti Red-Black stromů

- **Každý uzel stromu je obarven červenou nebo černou barvou.** ✔
- **Kořen stromu je obarven černě.** ✔
- **Listy (nil) jsou černé.** ✔
- **Červený uzel má pouze černé syny.** ✔
- **Na kterékoliv cestě z kořene do listu leží stejný počet černých uzlů** ✔

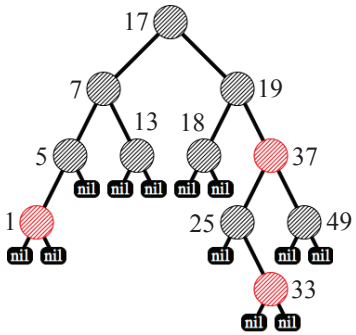
Vkládaný prvek: 21

Vložení prvku do Red-Black stromu



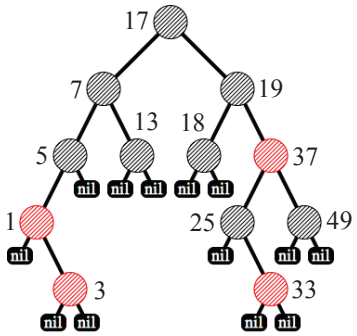
Vkládaný prvek: 21

Vložení prvku do Red-Black stromu



Vkládaný prvek: 3

Vložení prvku do Red-Black Stromu



Vkládaný prvek: 3

Vlastnosti Red-Black stromu

- **Každý uzel stromu je obarven červenou nebo černou barvou.** ✔
- **Kořen stromu je obarven černě.** ✔
- **Listy (nil) jsou černé.** ✔

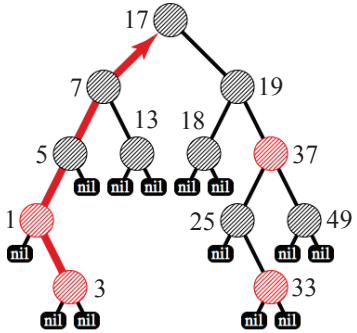
Problém dvou červených uzlů

- **Červený uzel má pouze černé syny.** ✘

- **Na kterékoliv cestě z kořene do listu leží stejný počet černých uzlů** ✔

Cíl: Obnovit vlastnosti **Red-black** stromu přebarvením uzlů popř. provedením rotací

Vložení prvku do Red-Black Stromu



Vkládaný prvek: 3

Vlastnosti Red-Black stromu

- **Každý uzel stromu je obarven červenou nebo černou barvou.** ✔
- **Kořen stromu je obarven černě.** ✔
- **Listy (nil) jsou černé.** ✔

Problém dvou červených uzlů

- **Červený uzel má pouze černé syny.** ✘

- **Na kterékoliv cestě z kořene do listu leží stejný počet černých uzlů** ✔

Cíl: Obnovit vlastnosti **Red-black** stromu přebarvením uzlů popř. provedením rotací

Obnovení vlastností Red-black stromu

Ve stromu existuje pouze jeden červený uzel x jehož předchůdce je červený

Postup:

Opravit problém dvou červených uzlů x

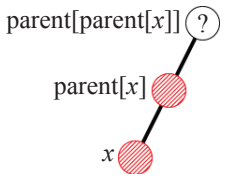
Oprava může způsobit stejný problém u předka -> je nutné postupovat směrem ke kořeni a upravit totéž i u předků

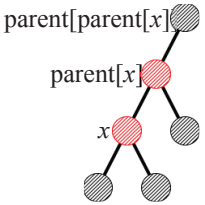
Platí následující:

Jelikož x má červeného předka pak x není kořen stromu.

Je-li parent[x] červený, pak ani on není kořenem tj. existuje parent[parent[x]].

Je-li parent[x] černý, pak úpravy končí.





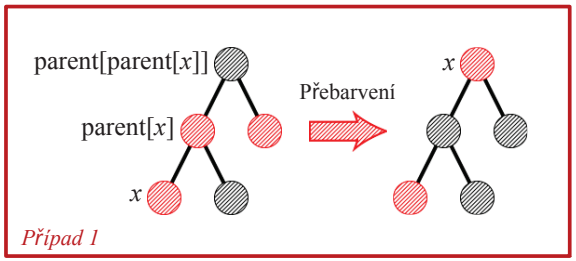
Algoritmus vložená prvků do RB stromu

```
RB-INSERT(T, z)
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB-INSERT-FIXUP(T, z)
```

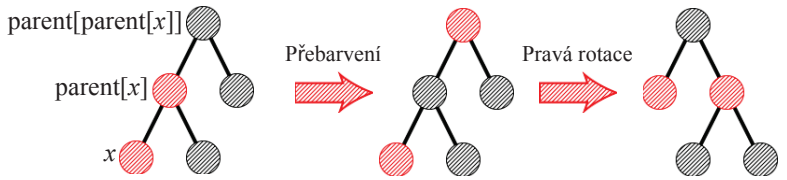
```
RB-INSERT-FIXUP(T, z)
 1  while color[p[z]] = RED
 2      do if p[z] = left[p[p[z]]]
 3            then y ← right[p[p[z]]]
 4                 if color[y] = RED
 5                    then color[p[z]] ← BLACK                ▷ Case 1
 6                         color[y] ← BLACK                   ▷ Case 1
 7                         color[p[p[z]]] ← RED               ▷ Case 1
 8                         z ← p[p[z]]                        ▷ Case 1
 9                    else if z = right[p[z]]
10                            then z ← p[z]                   ▷ Case 3
11                                 LEFT-ROTATE(T, z)          ▷ Case 3
12                         color[p[z]] ← BLACK                ▷ Case 2
13                         color[p[p[z]]] ← RED               ▷ Case 2
14                         RIGHT-ROTATE(T, p[p[z]])           ▷ Case 2
15            else (same as then clause
                    with "right" and "left" exchanged)
16  color[root[T]] ← BLACK
```

Obnovení vlastností Red-black stromu

- **Existují 3 případy:**
- parent[x] a jeho bratr jsou červení
- parent[x] je červený, jeho bratr je černý, and x je levý syn svého rodiče
- jako v případě 2; ale x je pravý syn svého rodiče



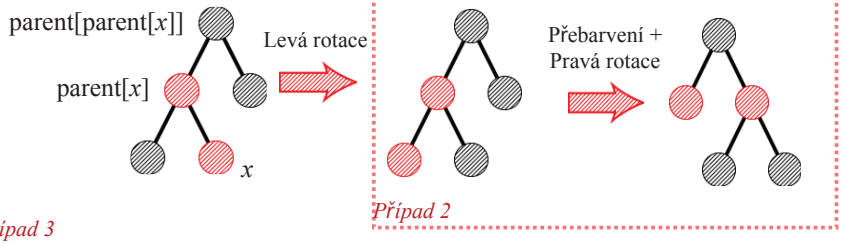
Případ 1



Případ 2

Obnovení vlastností Red-black stromu

- **Existují 3 případy:**
- parent[x] jeho bratr jsou červení
- parent[x] je červrný, and x je levý syn svého rodiče
- jako v případě 2; ale x je pravý syn svého rodiče



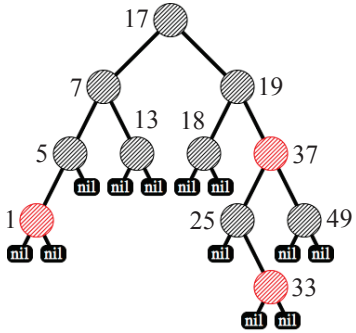
Případ 3 *Případ 2*

Vložení prvku do Red-black stromu - shrnutí

- **Pozorování:**
- **Každý z uvedených případů je proveden v konstantním čase**
- ***Případ 1* přesouvá x o dva kroky blíže ke kořeni a neprovádí se v něm žádné rotace – pouze se přebarvují uzly**
- **V *Případě 2* a 3, se provádí 1 nebo 2 rotace; pak úpravy končí**

Lemma: Vložení prvku do red-black stromu s n uzly má časovou složitost $O(\log n)$ a provádí se v něm pouze 2 rotace.

Zrušení uzlu v Red-Black stromu

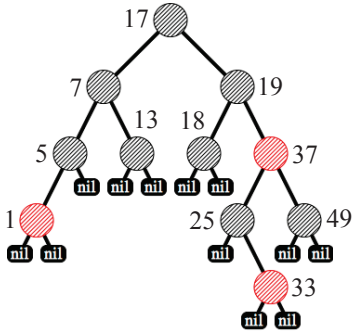


Vlastnosti Red-Black stromů

- Každý uzel stromu je obarven červenou nebo černou barvou. ✔
- Kořen stromu je obarven černě. ✔
- Listy (nil) jsou černé. ✔
- Červený uzel má pouze černé syny. ✔
- Na kterékoliv cestě z kořene do listu leží stejný počet černých uzlů ✔

Rušený prvek: 25

Zrušení uzlu v Red-Black stromu

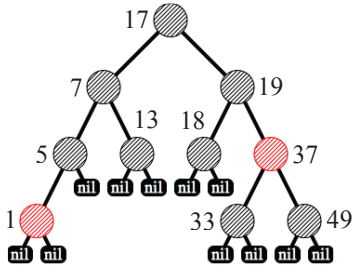


Vlastnosti Red-Black stromů

- Každý uzel stromu je obarven červenou nebo černou barvou. ✔
- Kořen stromu je obarven černě. ✔
- Listy (nil) jsou černé. ✔
- Červený uzel má pouze černé syny. ✔
- Na kterékoliv cestě z kořene do listu leží stejný počet černých uzlů ✔

Rušený prvek: 18

Zrušení uzlu v Red-Black stromu

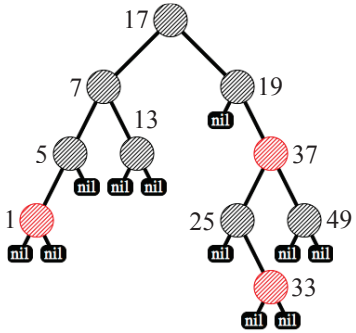


Vlastnosti Red-Black stromů

- Každý uzel stromu je obarven červenou nebo černou barvou. ✔
- Kořen stromu je obarven černě. ✔
- Listy (nil) jsou černé. ✔
- Červený uzel má pouze černé syny. ✔
- Na kterékoliv cestě z kořene do listu leží stejný počet černých uzlů ✔

Rušený prvek: 25

Zrušení uzlu v Red-Black stromu

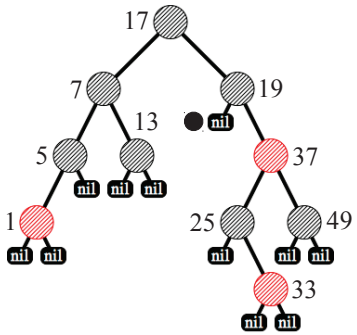


Vlastnosti Red-Black stromů

- Každý uzel stromu je obarven červenou nebo černou barvou. ✔
- Kořen stromu je obarven černě. ✔
- Listy (nil) jsou černé. ✔
- Červený uzel má pouze černé syny. ✔
- Na kterékoliv cestě z kořene do listu leží stejný počet černých uzlů ✔

Rušený prvek: 18

Zrušení uzlu v Red-Black stromu



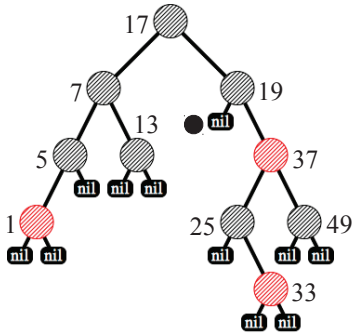
Vlastnosti Red-Black stromů

- **Každý uzel stromu je obarven červenou nebo černou barvou.** ✔
- **Kořen stromu je obarven černě.** ✔
- **Listy (nil) jsou černé.** ✔
- **Červený uzel má pouze černé syny.** ✔
- **Na kterékoliv cestě z kořene do listu leží stejný počet černých uzlů** ✔

Rušený prvek: 18

První krok: Označíme syna zrušeného uzlu jako extra černý (“doubly black”)

Zrušení uzlu v Red-Black stromu



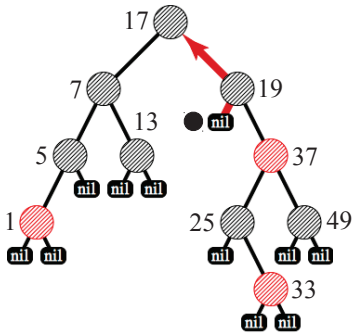
Vlastnosti Red-Black stromů

Barva neodpovídá

- **Každý uzel stromu je obarven červenou nebo černou barvou.** ✘
- **Kořen stromu je obarven černě.** ✔
- **Listy (nil) jsou černé.** ✔
- **Červený uzel má pouze černé syny.** ✔
- **Na kterékoliv cestě z kořene do listu leží stejný počet černých uzlů** ✔

Rušený prvek: 18

Zrušení uzlu v Red-Black stromu



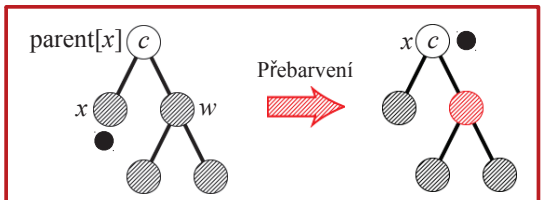
Rušený prvek: 18

Korekce barvy uzlů v RB stromu

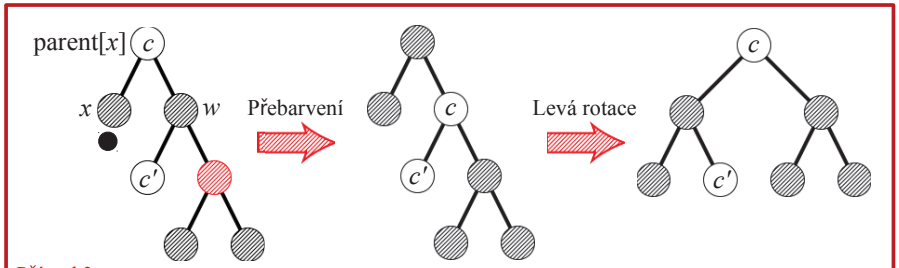
Předpokládejme že x je levý syn svého rodiče a jeho bratr w je černý.

Existují 3 případy, závisející na barvě synů w:

- Oba synové w jsou černí
- Pravý syn w je červený
- Levý syn w je červený a pravý je černý



Případ 1

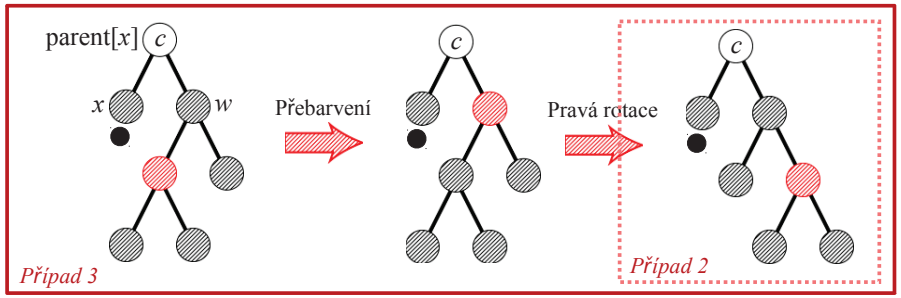


Případ 2

Korekce barvy uzlů v RB stromu

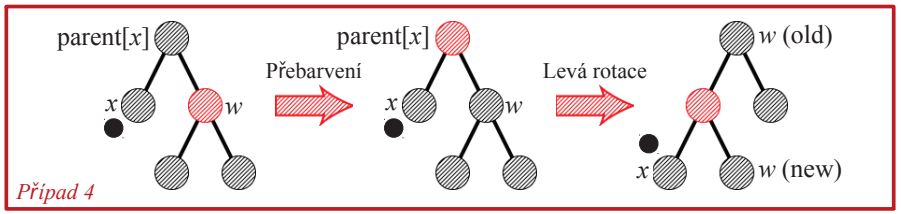
Předpokládejme že x je levý syn svého rodiče a jeho bratr w je černý.

- **Existují 3 případy, závisející na barvě synů w:**
- Oba synové w jsou černí
- Pravý syn w je červený
- Levý syn w je červený a pravý je černý



Korekce barvy uzlů v RB stromu

Případ 4: Pravý bratr w uzlu x je červený.



- **Pozorování:**
- Případy 2 a 3 provádí maximálně 2 rotace; pak je vše hotovo
- Případ 1 pouze přebarvuje a přesouvá korekci o jeden krok blíže ke kořeni
- Případ 4 provádí pouze jedinou rotaci a přesouvá korekci o **jeden krok dále od kořene!**

Lemma: Zrušení uzlu v red-black stromu s n uzly má časovou složitost $O(\log n)$ a provádí maximálně tři rotace.

Algoritmus zrušení prvku RB stromu

```
RB-DELETE(T, z)
 1  if left[z] = nil[T] or right[z] = nil[T]
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ nil[T]
 5     then x ← left[y]
 6     else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = nil[T]
 9     then root[T] ← x
10     else if y = left[p[y]]
11             then left[p[y]] ← x
12             else right[p[y]] ← x
13  if y ≠ z
14     then key[z] ← key[y]
15          copy y's satellite data into z
16  if color[y] = BLACK
17     then RB-DELETE-FIXUP(T, x)
18  return y
```

Algoritmus zrušení prvku RB stromu

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ root[T] and color[x] = BLACK
 2     do if x = left[p[x]]
 3           then w ← right[p[x]]
 4                if color[w] = RED
 5                   then color[w] ← BLACK                          ▷ Case 4
 6                        color[p[x]] ← RED                         ▷ Case 4
 7                        LEFT-ROTATE(T, p[x])                      ▷ Case 4
 8                        w ← right[p[x]]                           ▷ Case 4
 9                if color[left[w]] = BLACK and color[right[w]] = BLACK
10                   then color[w] ← RED                            ▷ Case 1
11                        x ← p[x]                                  ▷ Case 1
12                   else if color[right[w]] = BLACK
13                           then color[left[w]] ← BLACK            ▷ Case 3
14                                color[w] ← RED                    ▷ Case 3
15                                RIGHT-ROTATE(T, w)                ▷ Case 3
16                                w ← right[p[x]]                   ▷ Case 3
17                        color[w] ← color[p[x]]                    ▷ Case 2
18                        color[p[x]] ← BLACK                       ▷ Case 2
19                        color[right[w]] ← BLACK                   ▷ Case 2
20                        LEFT-ROTATE(T, p[x])                      ▷ Case 2
21                        x ← root[T]                               ▷ Case 2
22           else (same as then clause with "right" and "left" exchanged)
23  color[x] ← BLACK
```

Grafové algoritmy

Programovací techniky

Grafy – Úvod - Terminologie

FIGURE 12.4
Example of a Weighted Graph

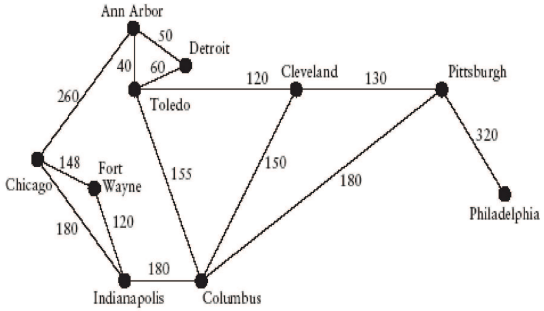
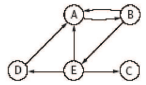


FIGURE 12.3
Example of a Directed Graph



Grafy – Úvod - Terminologie

- Graf je datová struktura, skládá se z množiny vrcholů “V” a množiny hran mezi vrcholy “E”
- Počet vrcholů a hran musí být konečný a nesmí být nulový u vrcholů ani u hran
- Grafy
  - Orientované – hrana (u,v) označena šipkou u->v
  - Neorientované – pokud ex. (u,v), existuje také (v,u)
- Souvislost – graf je souvislý, jestliže pro všechny v(i) z V existuje cesta do libovolného v(j) z V, nesouvislý graf je rozdělen na komponenty
- Strom – graf, ve kterém pro každé 2 vrcholy existuje právě jedna cesta

Grafy – Úvod – Reprezentace grafu

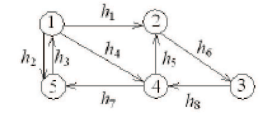
Incidenční matice:

Incidenční matice (Incidence matrix)
– orientovaný graf $G = (V,H,\varepsilon)$

$$a_{ik} = \begin{cases} 1, & \text{jestliže } \exists v \in V : \varepsilon(h_k) = (u_i, v) \\ -1, & \text{jestliže } \exists v \in V : \varepsilon(h_k) = (v, u_i) \\ 0, & \text{jinak} \end{cases}$$

- velikost incidenční matice $|V|.|H|$
- časová složitost ověření $(i,j) \in H$ je $O(|H|)$

Příklad 3.1



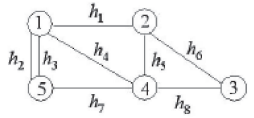
$\mathbf{A}^{\mathrm{I}} =$

	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8
1	1	1	−1	1	0	0	0	0
2	−1	0	0	0	−1	1	0	0
3	0	0	0	0	0	−1	0	1
4	0	0	0	−1	1	0	1	−1
5	0	−1	1	0	0	0	−1	0

Incidenční matice
– neorientovaný graf $G = (V,H,\varepsilon)$

$$a_{ik} = \begin{cases} 1, & \text{jestliže } h_k \text{ je incidentní s } u_i \\ 0, & \text{jinak} \end{cases}$$

Příklad 3.2

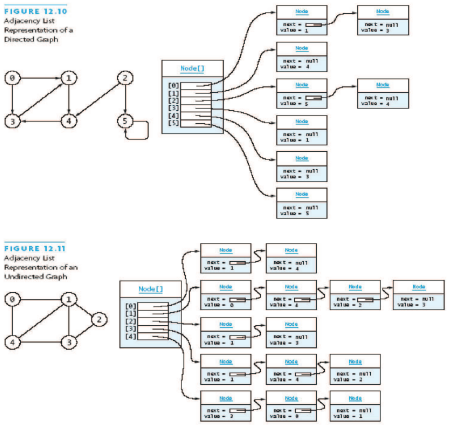


$$\mathbf{A}^I = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccccc} h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Grafy – Úvod – Reprezentace grafu

Seznamem sousednosti

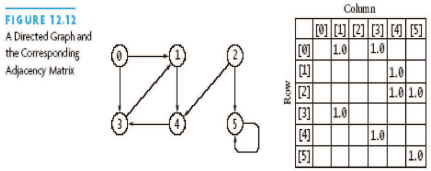
- Využívá pole seznamů
- Jeden seznam (sousedů) pro každý vrchol
- Vhodné, když chceme měnitelný počet hran

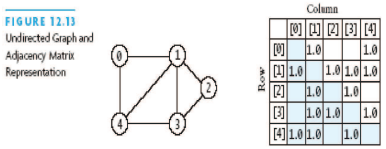


Grafy – Úvod – Reprezentace grafu

Matice sousednosti

- Dvou dimenzionální pole, hodnoty A(i,j) jsou ohodnocením hrany mezi vrcholy i,j
- Pro neohodnocený graf jsou prvky matice typu Boolean
- Neorientované grafy mají A symetrickou podle diagonály





Grafy – Úvod – Reprezentace grafu

- Plexová struktura
  - Vypadá jako reprezentace seznamem sousednosti, akorát vrcholy nejsou uloženy ve statickém poli, ale v dyn. struktuře (spojový seznam)
    - => Můžeme měnit počet hran i vrcholů

Hledání nejkratší cesty - v neohodnoceném grafu

Prohledávání do šířky

```
1. [Inicializace]
for ∀V−{r} do
    begin dosažen[v] := false; počet-hran-od-r[v] := ∞; předchůdce[v] := nil
    end;
počet-hran [r] := 0; dosažen[r] :=true; FRONTA := (r);
while 2. [Test ukončení] FRONTA ≠ ( ) do
    begin 3. [Volba vrcholu se začátku fronty]
          v := zač-fronty;
        4. [Postup do šířky]
          for ∀V_G^+(v) do    { V_G(v) – ve verzi pro neorientovaný graf }
             if not (dosažen[w])
                then begin dosažen[w] :=true;  předchůdce[w] := v;
                           počet-hran-od-r[w] :=počet-hran-od-r[v]+1;
                           FRONTA := FRONTA + (w)
                     end;
          FRONTA := FRONTA – (v)
    end;
```

Prohledávání do hloubky

```
for ∀v∈V do
    begin stav[v] := nedosažen; předchůdce[v] := nil
    end;
i := 0;
POSTUP-DO-HLOUBKY(r)
```

```
procedure POSTUP-DO-HLOUBKY(v)
    stav[v] := otevřen;  i := i+1;  poprvé[v] := i;
    for ∀w∈V_G^+(v) do          { V_G^+(v) – ve verzi pro neorientovaný graf }
        if stav[w]=nedosažen
           then begin předchůdce[w] := v;
                      POSTUP-DO-HLOUBKY(w)
                end;
    stav[v] := zavřen;  i := i+1;  naposled[v] := i;
```

Poznámka:

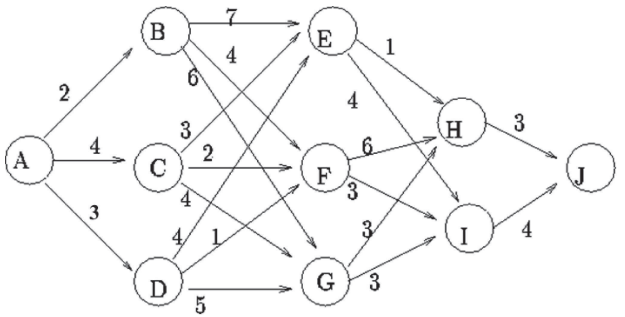
- vrchol *v* ve stavu zavřen
  … všichni následníci (sousedé) vrcholu *v* už jsou prozkoumání
- vrcholy ve stavu otevřen
  … zásobník rekurzivních volání
- $i \in \langle 1, 2|V| \rangle$
- $poprvé[v] < naposled[v]$

Poznámka - Hledání do hloubky (backtracking) - upgrade

- Ořezávání
  - **pokud vidím, že v n jakém uzlu (stavu) prohledáváním jeho následník UR IT nenajdu ešení, neprohledávám dále, ale vrátím se o úrove výš**
    - Příklad – v grafu projít všechny uzly a a neurazit přitom délku větší než D
- Heuristika
  - **Up ednostním n jakého následníka mezi jinými, podle n jakého kritéria, které mi SNAD urychlí hledání ešení**
    - Příklad(negrafový) – proskákat šachovnici koněm – půjdeme na ta pole, z kterých budeme mít nejméně možností dalšího skoku
- A* algoritmus – zahrnuje prohledávání do šířky i do hloubky s heuristikou i ořezáváním, viz dále.

Hledání nekratší cesty v acyklickém orientovaném grafu

- DAG (directed acyclic graph) shortest path algorithms
- Jaká je nejkratší cesta z A do J v tomto obrázku?



Hledání nekratší cesty v acyklickém orientovaném grafu

- Graf je kvůli své struktuře rozložitelný na etapy

Označíme:

1. **etapa: A**
2. **etapa: B, C, D**
3. **etapa: E, F, G**
4. **etapa: H, I**
5. **etapa: J**

Nechť S udává uzel v etapě j a $f_j(S)$ je nejkratší cesta mezi uzly S a J, platí

$$f_j(S) = \min_{\text{uzlu } z \text{ v etapě } j+1} \{c_{SZ} + f_{j+1}(Z)\}$$

Kde c_{sz} udává spočtený "oblak hran" *SZ*. Tímto dostáváme rekurentní vztah, který řeší náš problém.

DAG shortest path

- Složitost hledání – O(n+m)
- Nejdelší cesta v grafu

$$EC_1 = 0$$

$$EC_w = max_{(v,w)\in E}(EC_v + c_{v,w})$$

Použití:

- **např. diagram činností – ukazuje nejkratší čas ukončení projektu**
- **vzdálenost mezi městy atd.**

DAG – řešení úlohy

- **Za neme s $f_5(J)$**
- **Etapa 4**
  - Zde se nic nerozhoduje, jen přejdeme do J. Tím, že jdeme do J tedy dostáváme $f_4(H)=3$ a $f_4(I)=4$.
- **Etapa 3 (viz tabulka S_3)**
  - Jak spočítat $f_3(F)$. Z F můžeme jít do H nebo do I.
    - Hrana do H má hodnotu 6, následující je $f_4(H)=3$. Celkem tedy 9.
    - Hrana do I má hodnotu 3, následující je $f_4(I)=4$. Celkem tedy 7.
  - Jakmile se tedy dostaneme do F, je nelepší jít přes I s vydáním = 7.
  - Stejně pro E, G.
- **Pokra ujeme takto až do etapy 1**

S_3	$c_{S_3 Z_3} + f_4(Z_3)$		$f_3(S_3)$	Decision
	H	I		Go to
E	4	8	4	H
F	9	7	7	I
G	6	7	6	H

S_2	$c_{S_2 Z_2} + f_3(Z_2)$			$f_2(S_2)$	Decision
	E	F	G		Go to
B	11	11	12	11	E or F
C	7	9	10	7	E
D	8	8	11	8	E or F

S_1	$c_{S_1 Z_1} + f_2(Z_1)$			$f_1(S_1)$	Decision
	B	C	D		Go to
A	13	11	11	11	C or D

Dijkstrův algoritmus

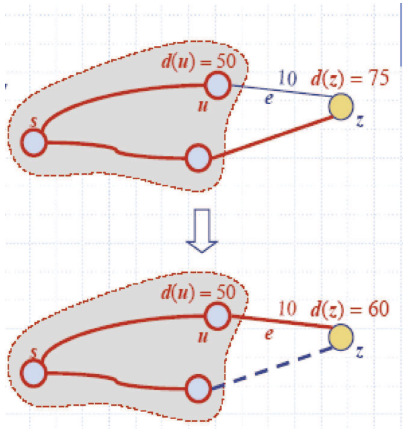
- **Hledání nejkratší cesty v ohodnoceném grafu z vrcholu *s* do vrcholu *t***
- **P edpokládáme:**
  - **Graf je souvislý**
  - **Neorientované hrany**
  - **Má nezáporn ohodnocené hrany**
- **Z navštívených vrchol vytvá íme mrak. Zpo átku mrak obsahuje po áte ní vrchol S, postupn p idáváme uzly až do nalezení cílového uzlu**
- **V každém kroku – p idáme do mraku vrchol *v* vn mraku s nejmenší vzdáleností d(v) (prioritní fronta)**
- **Opravíme vzdálenosti vrchol sousedících s v (vede asto k relaxaci hran)**

```
for ∀i ∈V do
    begin d[i] := w(s, i);
          označeno[i] := false;
          if w(s,i) < ∞ then před[i] := s
    end;
označeno[s] := true;
konec := false;
celková_délka_cesty := ∞;
repeat imin := 0;
       imin := i: (1) not označeno[i] ∧ (2) d[i] = min {d[u]} < ∞ { hladový postup }
       if imin = 0
         then konec := true { cesta nenalezena, pro všechny neoznačené vrcholy d[i]=∞ }
         else if imin = t
                then konec := true { cesta nalezena }
                else begin označeno[imin] := true;
                           for ∀i ∈{not označeno[i]} do

                                 if d[imin] + w[imin, i]) < d[i];
                                   then begin d[i] := d[imin] + w[imin, i];
                                              před[i] := imin
                                         end        { relaxace }
                     end
until konec;
{ if imin = t then celková_délka_cesty := d[imin] } { Dijkstra s → t }
```

Dijkstrův algoritmus

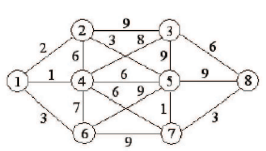
- Relaxace hran
  - Relaxace hrany e upravuje vzdálenost d(z) takto:

d(z)←min{d(z),d(u)+weight(e)}



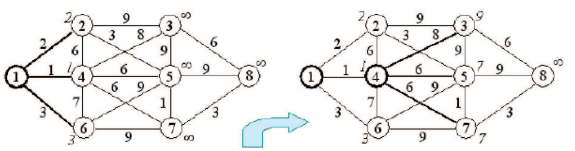
Příklad 7.1

V následujícím grafu najít nejkratší cestu z vrcholu 1 do vrcholu 8.



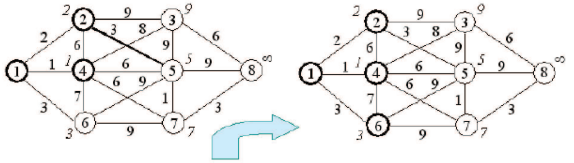
$$A = \begin{array}{c} \\ 1\\2\\3\\4\\5\\6\\7\\8 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \infty & 2 & \infty & 1 & \infty & 3 & \infty & \infty \\ 2 & \infty & 9 & 6 & 3 & \infty & \infty & \infty \\ \infty & 9 & \infty & 8 & 9 & \infty & \infty & 6 \\ 1 & 6 & 8 & \infty & 6 & 7 & 6 & \infty \\ \infty & 3 & 9 & 6 & \infty & 9 & 1 & 9 \\ 3 & \infty & \infty & 7 & 9 & \infty & 9 & \infty \\ \infty & \infty & \infty & 6 & 1 & 9 & \infty & 3 \\ \infty & \infty & 6 & \infty & 9 & \infty & 3 & \infty \end{bmatrix} \begin{array}{c} 1\\2\\3\\4\\5\\6\\7\\8 \end{array}$$

kopírovat vzdálenosti vstupů do vrcholů to 2,4, 6



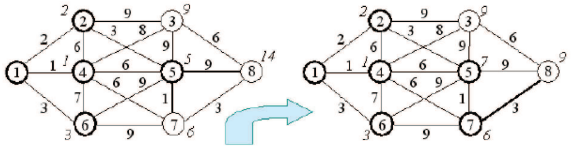
vzít 4 *a označit jej,*
aktualizovat vzdálenost do 3,5,7

vzít 2 *a označit jej,*
aktualizovat vzdálenost do 5



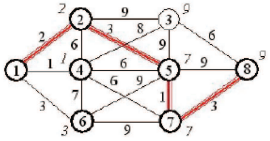
vzít 6 *a označit jej,*
žádné vzdálenosti se neaktualizují

nejmenší vzdálenost z neoznačených
vrcholů má 5, *označit jej,*
aktualizovat vzdálenost do 7,8



vzít 7 *a označit jej,*
aktualizovat vzdálenost do 8

8 *je cíl, vzít jej a označit*



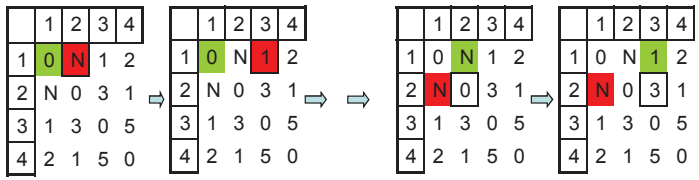
Nejkratší cesta z vrcholu 1 do 8 je: 1 – 2 – 5 – 7 – 8

Floyd-Warshallův algoritmus

- Algoritmus hledající minimální cestu mezi všemi páry vrcholů (all-pair shortest path algorithm)
- Vhodný pro husté grafy – v tom případě rychlejší než Dijkstra opakovaný pro všechny vrcholy
- Pracuje s maticí sousednosti
- Složitost $O(n^3)$
- Můžeme stanovit max. počet vrcholů, přes které se jde
- Je technikou dynamického programování – viz dále

Floyd-Warshallův algoritmus

- **P íklad –** je dána matice sousednosti grafu. FW algoritmem nalezněte nejkratší cestu mezi všemi vrcholy

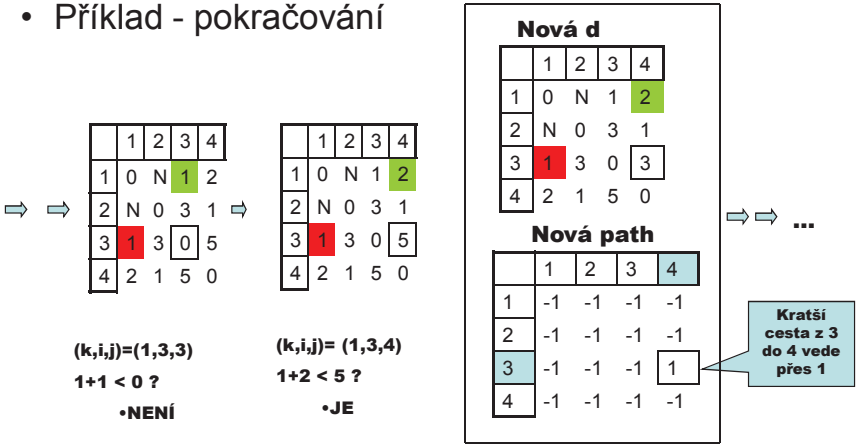



```
FLOYD-WARSHALL(G,d,mezi)
   vstup  :  souvislý graf G=(V, H) s nezáporným ohodnocením hran  w: H(G) →ℜ+;
   výstup :  d[i,j], i,j ∈V;
             mezi[i,j], i,j ∈V;
   for i := 1 to |V| do
      for j := 1 to |V| do
         begin d[i,j] := w[i,j];
               mezi[i,j] := null
         end;
   for k := 1 to |V| do
      for i := 1 to |V| do
         for j := 1 to |V| do
            if d[i,k]+d[k,j] < d[i,j]
               then begin d[i,j] := d[i,k]+d[k,j];
                          mezi[i,j] := k
                    end;
```

- Matice sousednosti je uložena ve **w**
- **d** je matice aktuálně spočtených nejkratších vzdáleností
- **mezi** je matice nejkratších mezicest, je nainicializována na -1 (null)
- V každém kroku algoritmu zjišťuji, jestli existuje mezi vrcholy i,j kratší cesta přes vrchol k, pokud ano, nastavím vzdálenost v **d[i][j]** na novou velikost a do **path[i][j]** zaznamenám vrchol k

Floyd-Warshallův algoritmus

- Příklad - pokračování



Floyd-Warshallův algoritmus

- Rekonstrukce nejkratší cesty mezi i,j
  - V **mezi[i][j]** je index vrcholu k, přes který se má jít. Pokud k není -1(null) podívám se do **mezi** na nejkratší cestu mezi i,k a k,j
  - Toto opakuji rekurzivně dokud nenajdu **mezi[i][j] == -1**.

Aplikace nejlevnější kostry

Elektrifikace území

O. Borůvka algoritmus hledání nejlevnější kostry objevil jako řešení konkrétní praktické úlohy *elektrifikace území*. V této úloze obce tvoří vrcholy grafu, elektrická spojení hrany a ohodnocení hran reprezentuje cenu za natažení vedení mezi obcemi. Úkolem bylo najít takové spojení, aby všechny obce byly připojeny do elektrické sítě a přitom cena spojení byla minimální.

Sjízdnost silnic

Máme silniční síť, z níž chceme udržovat sjízdnou co nejkratší část, která vzájemně propojí všechny obce dané oblasti.

Vybudování železniční sítě

Mezi městy daného regionu máme navrhnout železniční síť tak, aby každá 2 města byla po železnici dosažitelná a přitom náklady na vybudování železniční sítě byly minimální.

Minimální kostra grafu

- Algorimus na hledání minimální kostry grafu (Minimum Spannig Tree)

Kostra grafu:
- minimální souvislý podgraf, který obsahuje všechny vrcholy
- neobsahuje cykly – je to strom

Definice: Je dán souvislý graf G, jehož hrany jsou ohodnoceny reálnými čísly, které budeme nazývat cenami. Kostrou grafu, která má nejmenší ohodnocení hran mezi všemi kostrami, nazýváme „nejlevnější (minimální) kostrou grafu“

Kruskalův algoritmus:

[*Určit nejlevnější kostru v souvislém ohodnoceném grafu* $G=(V,H)$]

1. [*Inicializace kostry* K.]
   $V(K) := V(G)$; $H(K) := \emptyset$; $\{ K := (V, \emptyset) \}$
2. [*Přidání hrany do* K.]
   Nechť h je hrana s minimálním ohodnocením taková, že $h \notin H(K)$
   a $(V, H(K) \cup \{h\})$ je acyklický graf,
   $H(K) := H(K) \cup \{h\}$.
3. [*Test ukončení*.]
   Jestliže $|H(K)| = |V(G)| - 1$, pak $K = (V, H(K))$, jinak návrat na krok 2.

Alternativně:

1. $V(K) := V(G)$; $H(K) := H(G)$; $\{ K := G \}$
2. [*Odebrání hrany z* K.]
   Nechť h je hrana s maximálním ohodnocením taková, že $h \in H(K)$
   a $(V, H(K) - \{h\})$ je souvislý graf,
   $H(K) := H(K) - \{h\}$.
3. [*Test ukončení*.]
   Jestliže $|H(K)| = |V(G)| - 1$, pak $K = (V, H(K))$, jinak návrat na krok 2.

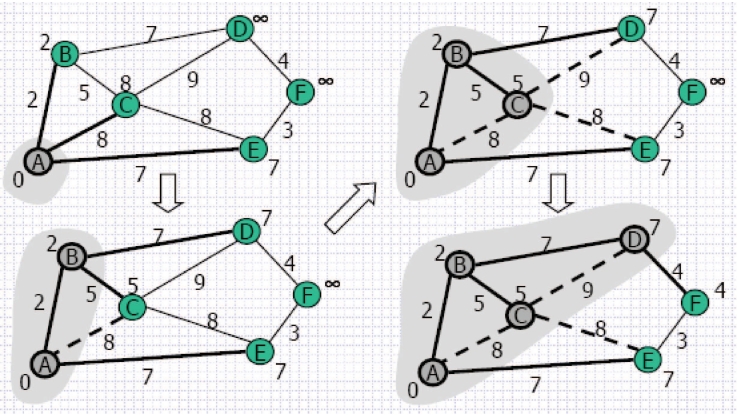
Borůvkův algoritmus

[*Určit nejlevnější kostru v souvislém ohodnoceném grafu G=(V,H)*]

1. [*Inicializace lesa L.*]
   $V(L) := V(G)$; $H(L) := \emptyset$; $\{ L := (V, \emptyset) \}$
2. [*Aktualizace lesa L.*]
   Pro každou komponentu L' lesa L určit nejlevnější hranu z hran vycházejících z L' k jiným komponentám. Nechť S je množina těchto hran. Pak položme
   $H(L) := H(L) \cup S$
   a určeme komponenty souvislosti v lese $L = (V, H(L))$
3. [*Test ukončení.*]
   Jestliže $|H(L)|=|V(L)|-1$, pak $K=(V,H(L))$, jinak návrat na krok 2.
   (Alternativně jestliže počet komponent je roven 1, pak $K=(V,H(L))$, jinak návrat na krok 2.)

Primův-Jarníkův algoritmus

- Příklad(1)



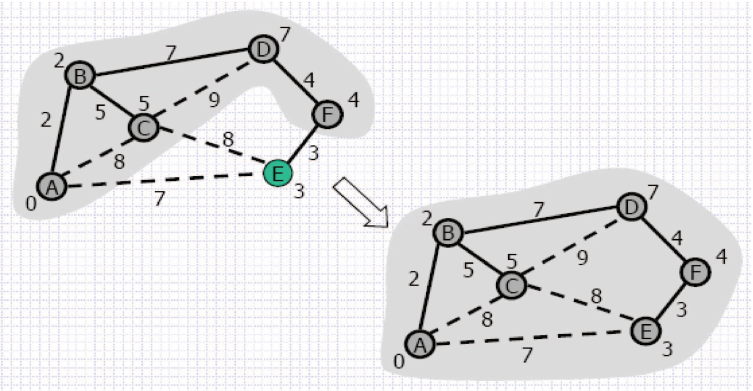
Primův – Jarníkův algoritmus

[*Určit nejlevnější kostru v souvislém ohodnoceném grafu G=(V,H)*]

1. [*Inicializace kostry K.*]
   Nechť v je libovolný vrchol grafu G,
   $V(K) := \{v\}$; $H(K) := \emptyset$; $\{ K := (\{v\},\emptyset) \}$
2. [Aktualizace *kostry.*]
   Nechť h je hrana s minimálním ohodnocením z hran spojujících vrchol $u \in V(K)$ s vrcholem $v \notin V(K)$, pak
   $V(K) := V(K) \cup \{v\}$; $H(K) := H(K) \cup \{h\}$
3. [Test *ukončení.*]
   Jestliže $|H(K)|=|V(G)|-1$, pak $K=(V,H(K))$, jinak návrat na krok 2.

Primův-Jarníkův algoritmus

- Příklad - pokračování



ADT Tabulka

Tabulka

- Datová struktura, která umožňuje vkládat a později vybírat informace podle identifikačního klíče. Mohou být:
  - pevně definované (LUT Look Up Table)
  - s proměnným počtem položek

Konvence:

k – klíč, kterým identifikujeme položku

$\mathbf{A_k}$ – adresní klíč tj. „adresa položky" (ve většině případů je to index

Hledání v tabulkách – parametry:

S – délka hledání položky (počet položek, které je nutno prozkoumat)

T – průměrná délka hledání (m je počet přístupů do tabulky)

$$T = \frac{1}{m}\sum_{i=1}^{m} S_i$$

A – průměrná délka prohledávání za předpokladu rovnoměrného přístupu (n je počet obsazených položek tabulky)

$$A = \frac{1}{n}\sum_{i=1}^{n} S_i$$

P – tzv. plnění tabulky(podíl obsazených položek)

$$P = \frac{n}{p}$$

kde p je velikost tabulky

Rozdělení tabulek podle způsobu organizace

- tabulky s přímým přístupem
- obyčejné vyhledávací tabulky
- tabulky se sekvenčním přístupem
- tabulky s rozptýlenými položkami

Tabulky s přímým přístupem

- **k** → **Ak** je prosté zobrazení, každá položka tabulky má své místo jednoznačně určené hodnotou Ak přímo odvozenou z k
- Optimální implementace tabulky je pomocí pole – indexy jsou přímo klíče v tabulce

  S=T=A=1
- Výhody:
  - rychlý přístup
  - jednoduchá implementace
- Nevýhody:
  - Velikost tabulky je daná rozsahem klíče, pro praktické účely bývá většinou neúnosná
  - Řídké pole – nerovnoměrný počet klíčů vzhledem k rozsahu tabulky.
- Příklady:
  - Telefonní síť – klíčem je telefonní číslo uživatele
  - Telefonní seznam – klíčem je jméno

Vyhledávací tabulky

- Vyhledává se podle hodnoty klíče
- Pořadí položek může být:
  - definované (uspořádané)
  - náhodné
- Strategie vyhledávání:
  1. sekvenční
  2. binární
  3. Fibonacciho
  4. kombinované

- Výhoda:
  - plnění vyhledávacích tabulek může být až 100%
- Nevýhoda
  - časově náročné vyhledávání (může být až lineární)

Sekvenční vyhledávání

- Položky v tabulce mohou být neuspořádané
- Princip spočívá v postupném porovnávání klíčů položek s hledaným klíčem až do nalezení shody, popř. nalezení konce tabulky

$$A = \frac{1}{2}(n+1)$$

- **Výhoda**: snadná implementace a časově nenáročná modifikace tabulky (implementace jako pole (popř. jako seznam), přidává se na konec pole, díry po odebrání prvků se vyplňují posledním prvkem – v případě implementace polem)

Binární vyhledávání

- lze použít v případě, že jsou položky tabulky seřazeny podle hodnoty klíče

Princip: porovnat hledaný klíč s klíčem uprostřed tabulky, pokud není shoda, hledat v levé popř. v pravé polovině tabulky (v závislosti na hodnotě klíče)

$$A \approx log(p)$$

Fibonacciho vyhledávání

Princip: stejný jako u binárního vyhledávání, testované prvky však nevolíme uprostřed, ale v poměru Fibonacciho čísel

Složitost: stejná jako u binárního vyhledávání, ale prvky na začátku jsou nalezeny rychleji

- Pro efektivní hledání se snažíme, aby tabulka měla $\mathbf{F_n}$**-1** prvků, kde $\mathbf{F_n}$ je určité Fibonacciho číslo

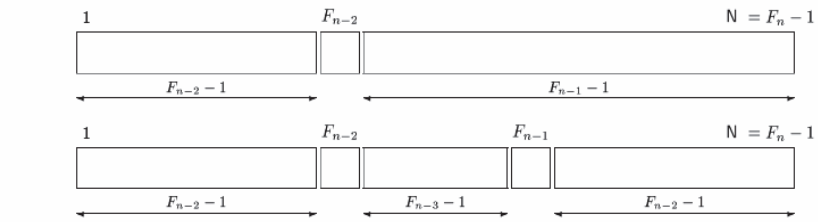
Fibonacciho posloupnost:

$$F_1 = 0, \qquad F_2 = 1,$$

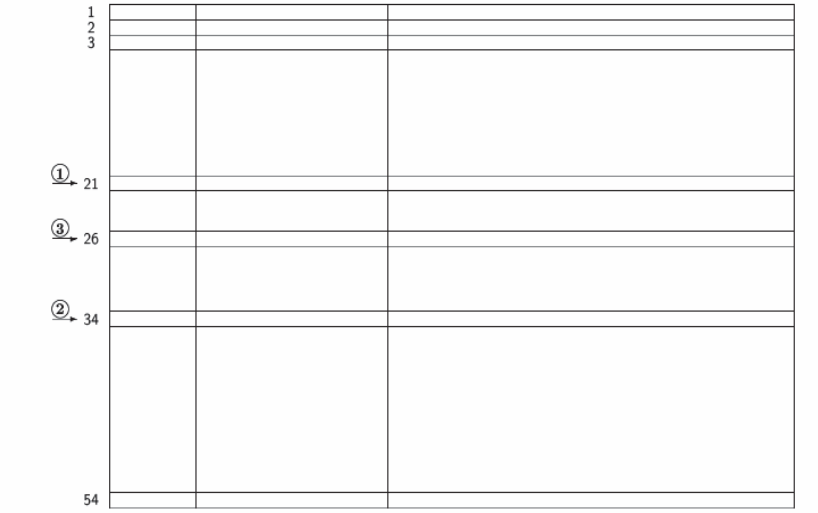
$$F_n = F_{n-1} + F_{n-2} \qquad \forall n \in \mathbf{N}, n > 2$$

Prvky Fibonacciho posloupnosti:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Př. Nalezení prvku na pozici 26 v tabulce o 54 prvcích.



Vyhledání:

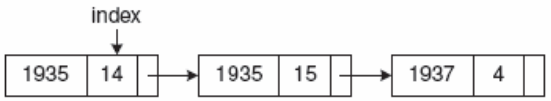


Hledání podle sekundárního klíče

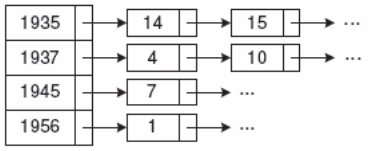
- Invertovaný soubor (indexovaný soubor)
  - jedná se o tabulku seřazenou podle sekundárního klíče, data tvoří primární klíč (nebo přímo index v původní tabulce)

1935	Fiala Jan
1935	Fiala Martin
1937	Bláha Josef
1945	Bláha Jan
1956	Fiala Petr

- invertovaný seznam
  - invertovaný soubor implementovaný jako zřetězený seznam

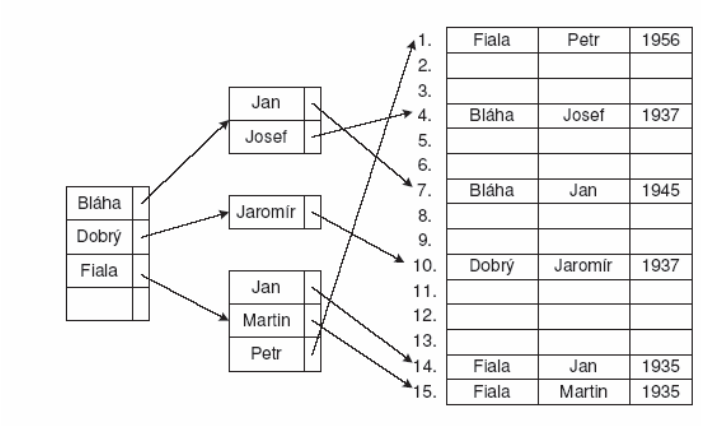


nebo



Vícerozměrné vyhledávání

- vyhledávání podle více klíčů realizujeme vícenásobným přístupem



tabulky s rozptýlenými položkami

- používají se v případě, že rozsah klíče N >> rozsah tabulky
- pro určení pozice v tabulce, na kterou máme uložit položkus klíčem k, používáme rozptylovací funkce (hash-funkci)

$$\mathbf{A_k = h(k),}$$

  která klíči **k** jednoznačně přiřazuje klíč $\mathbf{A_k}$
- může se stát, že pro různé položky $k_1 \neq k_2$ platí, že $h(k_1) = h(k_2)$ – tzv. synonymické položky (dochází ke kolizi)

Při návrhu a implementaci hash-tabulky je nutné vzít v úvahu:

- jak definovat rozptylovací funkci
- jak řešit ukládání synonymických položek

Požadavky na rozptylovací funkci:

1. Pro každé k je jednoznačně definovaná (a v přijatelném čase vyčíslitelná)
2. Vytváří minimální počet kolizí (minimum synonym)
3. Pravděpodobnostní rozdělění Ak=h(k) na intervalu <0, p-1> je rovnoměrné - lze využít pseudonáhodné funkce (*randomizační funkce*)

Realizace h(k):

hash funkci **i=h(k)** je možné realizovat následujícími způsoby:

- **i** je částí **k**
- **i** je částí operace nad **k**
- **i** je zbytkem po dělení rozsahem tabulky **p**
- **i** je zbytkem po děleni **N**, **N** je nejbližší menší prvočíslo než hodnota **p**
- i je dán váhovým součtem částí

$$i = \sum_{i=1}^{r} a_i x^i$$

kde $\mathbf{a}_i$ jsou váhy jednotlivých částí $\mathbf{k}^i$ klíče **k**

Tabulky s otevřeným rozptýlením

Každá pozice tabulky je potenciálně přístupná položce s libovolným klíčem, snadno vznikají shluky položek. Při kolizi se hodnota A_k přepočítá.

- rozptylovací funkci volíme například

$$h = h \mod p$$

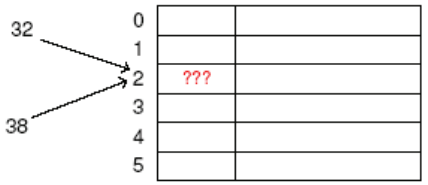
kde p je prvočíslo

- vhodné pro tabulky s velkým rozsahem klíčů, ale malým počtem položek

Podle způsobu řešení kolizí rozeznáváme čtyři podtypy:

1. Tabulky s otevřeným rozptýlením a *nedefinovaným způsobem* ukládání synonymických položek
   - nutno dodefinovat způsob přepočítání A_k při kolizi

   Příklad:



Kolize: prvky 32 a 38 mají stejnou hodnotu rozptylové funkce

2. Tabulky s otevřeným rozptýlením a ukládáním synonymických položek s *konstantním krokem* s (s vícenásobnou hashovací funkcí)
   - na klíč k aplikujeme funkci $h_0(k)$, pokud dojde ke kolizi, vypočteme znovu A_k podle $h_1(k)$ atd. až do té doby, než najdeme v tabulce volné místo

$$\begin{aligned} h_0(k) &= h(k) \\ h_1(k) &= (h(k)+s) \mod p \\ &\vdots \\ h_i(k) &= (h(k)+is) \mod p \end{aligned}$$

kde p je rozsah tabulky a s je přirozené číslo nesoudělné s p (nesoudělnost zaručí možnost dostat se na každou pozici tabulky)

Příklad: Tabulka s

- 8 pozicemi
- položkami s klíči DAVID, HANA, DANA, HELENA, EMIL, EVA, BOŽENA
- hashovací funkcí

$$h(k) = \text{pořadí 1. písmene v abecedě}$$

S = 1

0	HELENA	2
1	BOŽENA	1
2		
3	DAVID	1
4	DANA	2
5	EMIL	2
6	EVA	3
7	HANA	1

st. slozitost = 12/7

S = 3

0		
1	BOŽENA	1
2	HELENA	2
3	DAVID	1
4	EMIL	1
5	EVA	4
6	DANA	2
7	HANA	1

st. slozitost = 12/7

3. Tabulky s otevřeným rozptýlením a ukládáním synonym s *lineární vícenásobnou ukládací funkcí*

$$\begin{aligned} h_0(k) &= h(k) \\ h_i(k) &= (h(k) + a \cdot i + b) \mod p \end{aligned}$$

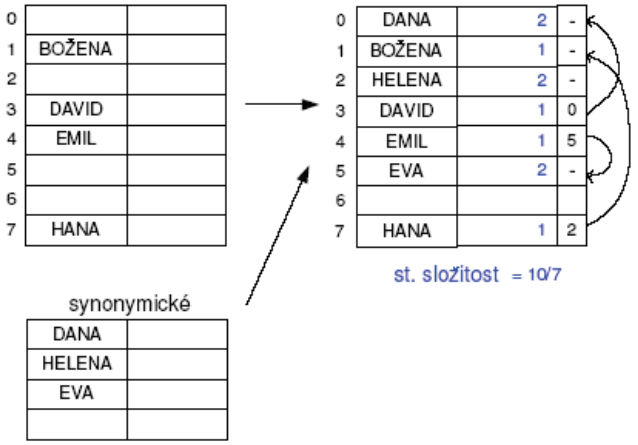
4. Tabulky s otevřeným rozptýlením a ukládáním synonym s *kvadratickou vícenásobnou ukládací funkcí*

$$\begin{aligned} h_0(k) &= h(k) \\ h_i(k) &= \left(h(k) + (-1)^i \cdot \left\lceil \frac{i}{2} \right\rceil^2\right) \mod p \end{aligned}$$

kde $\lceil x \rceil$ značí zaokrouhlení čísla x „nahoru“. U zjišťování zbytku po dělení (*mod*) je třeba dávat pozor na záporné argumenty — držet se definice!

Tabulky s otevřeným rozptýlením a vnitřním zřetězením

Ve fázi zařazování nových položek do tabulky si při výskytu kolize ponecháme synonymické položky vedle. Po zařazení všech položek vřadíme tato dočasně odložená synonyma na zbylá volná místa v tabulce (podle předem zvoleného systému) a příslušná synonyma zřetězíme do tzv. *řetězu synonym*



Přesun synonym do tabulky:

1. od začátku
2. od konce
3. s vícenásobnou hash-funkcí (to je výhodné, protože nám to „nerozboura“ rovnoměrné rozložení prvků)

- tento typ tabulky je vhodný, pokud tabulku na začátku jednou vytvoříme a potom opakovaně používáme
- problém je přidání položky na první místo v řetězci synonym, pokud toto místo již jiná položka obsadila.

Tabulky s uzavřeným rozptýlením a vnějším zřetězením

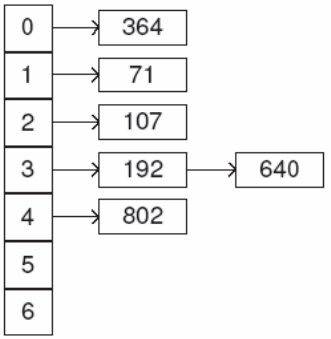
– přinášejí zlepšení poslední metody: rozdělíme tabulku na dvě části

1. *primární část* — obsahuje pouze položky, které nejsou synonyma
2. *sekundární část* (*zóna zřetězení*, *přeplnění*, atd.) — položky, které jsou synony: k položkám v primární části

– problémem je vypuštění prvního prvku v řetězci — řeší se zavedením *děr*

Metoda rozptýlených indexů

– tabulku chápeme jako vektor seznamů synonymických položek (řetězců)



– zařazování nových prvků:

1. na konec řetězu
2. uspořádaně — složitější implementace, ale přináší jisté urychlení

– velice oblíbená metoda (100% plnění, jednoduchá mplementace, . . .)

Skip-List

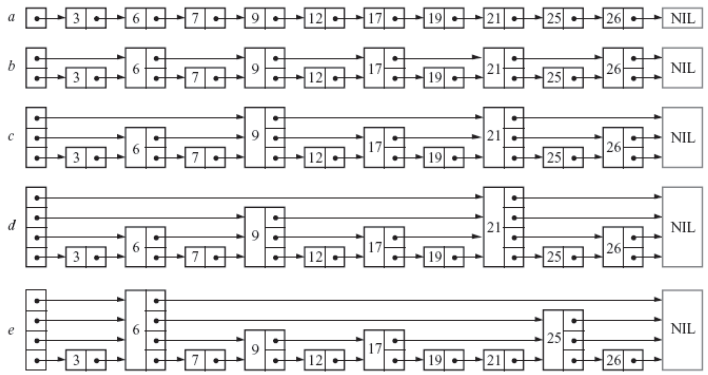
- je datová struktura, která může být použita jako náhrada za vyvážené stromy.
- představují pravděpodobnostní alternativu k vyváženým stromům (struktura jednotlivých uzlů se volí náhodně)
- Na rozdíl od stromů má **skip list** následující výhody:
  - jednoduchá implementace
  - jednoduché algoritmy vložení/zrušení
  - časová složitost vyhledávání je obdobná jako u stromů

Skip-List

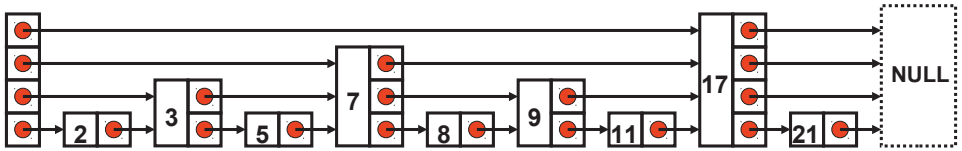
- prky v seznamu jsou uspořádány
- seznam obsahuje prvky, které mají k ukazatelů

  $1 \le k \le max\_level$
- uzel s *k*-ukazateli se nazývá uzel úrovně k
- seznam úrovně k – obsahuje prvky s maximálně k ukazateli
- ideální skip-list – každý 2^i-tý prvek má ukazatel, který ukazuje o 2^i prvků dopředu

Základní myšlenka zavedení skip-listů



seznam	složitost vyhledávání – nejhorší případ
a) obyčejný spoj.seznam	n
b) extra ukazatele mezi každým 2. uzlem	$\lceil n/2 \rceil + 1$
c) extra ukazatele mezi každým 4. uzlem	$\lceil n/4 \rceil + 1$
d) extra ukazatele mezi každým 2^i. uzlem	$\lceil \log n \rceil$
e) náhodná volba extra uzlů s ukazateli (skip list)	???



Pokud má každý 2^itý uzel 2^i ukazatelů na následující uzly, pak jsou uzly jednotlivých úrovní rozloženy následovně:

50% uzlů úrovně 1
25% uzlů úrovně 2
12.5% uzlů úrovně 3
atd.

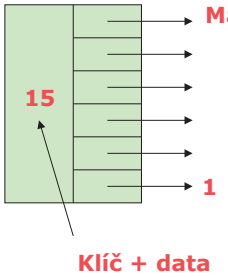
Výhoda: složitost vyhledávání O(log n)

Nevýhoda: po provedení operací insert/delete je nutné provádět restrukturalizaci seznamu

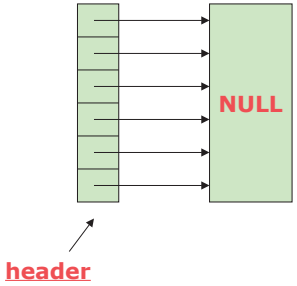
Řešení: ponechat rozložení uzlů ale vyhnout se restrukturalizaci – tj.
uzly úrovně k jsou vkládány náhodně s uvedeným pravděpodobnostním rozložením

Prvek Skip-listu

- každý prvek seznamu úrovně k má k ukazatelů (k se volí náhodně při vytvoření prvku)



Prázdný seznam



Inicializace seznamu_

- je vytvořena hlavička seznamu (obsahuje MaxLevel ukazatelů)
- všechny ukazatele se inicializují na NIL
- celkový počet úrovní MaxLevel se volí na základě maximálního počtu prvků N MaxLevel=$\log_2(N)$

Algoritmus vyhledávání

```
Search(list, searchKey)
    x := list→header
    -- loop invariant: x→key < searchKey
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
    -- x→key < searchKey ≤ x→forward[1]→key
    x := x→forward[1]
    if x→key = searchKey then return x→value
        else return failure
```

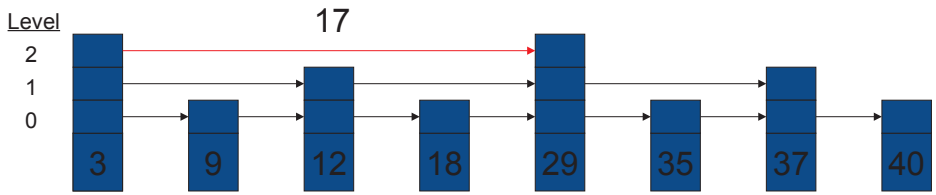
Vyhledávání

- Začínáme v nejvyšší úrovni
- Dokud je hledaný prvek větší než prvek na který ukazuje ukazatel,
  - posouváme se vpřed v dané úrovni .
- Pokud je hledaný prvek menší než následující klíč,
  - přesuneme se o jednu úroveň níž.
- Opakujeme postup pokud není prvek nalezen, nebo pokud není jisté (v úrovni 1), že prvek neexistuje.

- Časová složitost
  - nejlepší/průměrný případ : logaritmický
  - nejhorší případ : lineární (skip list přechází v normální spojový seznam)

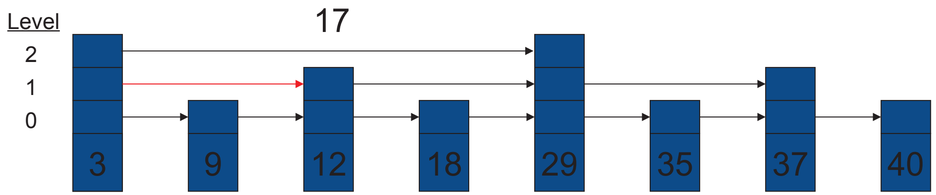
Vložení prvku

- Vyhledávacím algoritmem nalezněte pozici pro vložení prvku
  - zapamatujte pozici předchůdce
- Zvolte úroveň nově vkládaného uzlu
- Vložte nový uzel a pokud je to nutné zvětšete hodnotu MaxLevel



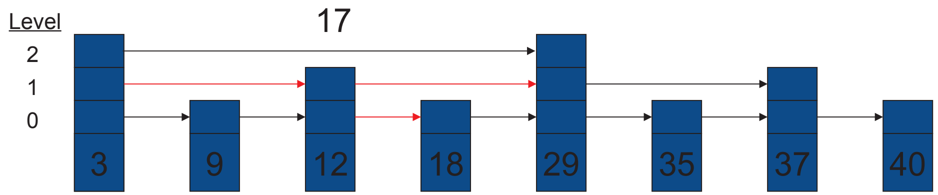
Vložení prvku

- Vyhledávacím algoritmem nalezněte pozici pro vložení prvku
  - zapamatujte pozici předchůdce
- Zvolte úroveň nově vkládaného uzlu
- Vložte nový uzel a pokud je to nutné zvětšete hodnotu MaxLevel



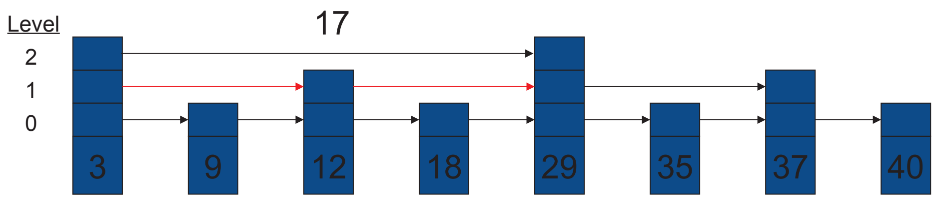
Vložení prvku

- Vyhledávacím algoritmem nalezněte pozici pro vložení prvku
  - zapamatujte pozici předchůdce
- Zvolte úroveň nově vkládaného uzlu
- Vložte nový uzel a pokud je to nutné zvětšete hodnotu MaxLevel



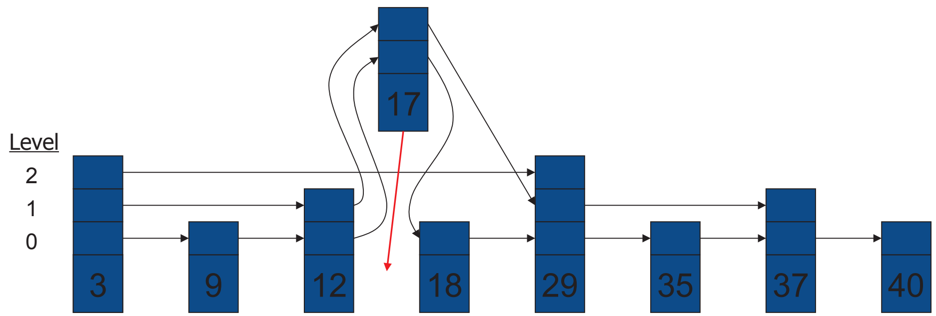
Vložení prvku

- Vyhledávacím algoritmem nalezněte pozici pro vložení prvku
  - zapamatujte pozici předchůdce
- Zvolte úroveň nově vkládaného uzlu
- Vložte nový uzel a pokud je to nutné zvětšete hodnotu MaxLevel



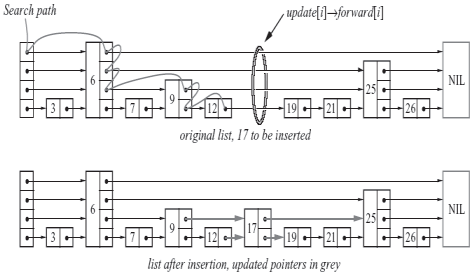
Vložení prvku

- Vyhledávacím algoritmem nalezněte pozici pro vložení prvku
  - zapamatujte pozici předchůdce
- Zvolte úroveň nově vkládaného uzlu
- Vložte nový uzel a pokud je to nutné zvětšete hodnotu MaxLevel



Volba náhodné úrovně

```
randomLevel()
    lvl := 1
    -- random() that returns a random value in [0...1)
    while random() < p and lvl < MaxLevel do
        lvl := lvl + 1
    return lvl
```



Algoritmus vložení prvku

```
Insert(list, searchKey, newValue)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
        -- x→key < searchKey ≤ x→forward[i]→key
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then x→value := newValue
    else
        lvl := randomLevel()
        if lvl > list→level then
            for i := list→level + 1 to lvl do
                update[i] := list→header
            list→level := lvl
        x := makeNode(lvl, searchKey, value)
        for i := 1 to level do
            x→forward[i] := update[i]→forward[i]
            update[i]→forward[i] := x
```

Porovnání s ostatními datovými strukturami

Implementation	Search Time	Insertion Time	Deletion Time
Skip lists	0.051 msec (1.0)	0.065 msec (1.0)	0.059 msec (1.0)
non-recursive AVL trees	0.046 msec (0.91)	0.10 msec (1.55)	0.085 msec (1.46)
recursive 2–3 trees	0.054 msec (1.05)	0.21 msec (3.2)	0.21 msec (3.65)
Self–adjusting trees:			
top-down splaying	0.15 msec (3.0)	0.16 msec (2.5)	0.18 msec (3.1)
bottom-up splaying	0.49 msec (9.6)	0.51 msec (7.8)	0.53 msec (9.0)

Table 2 - Timings of implementations of different algorithms

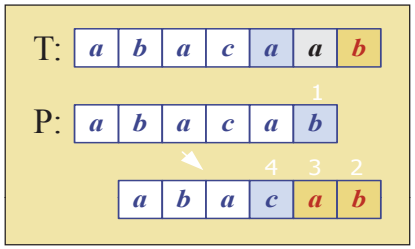
p	Normalized search times (*i.e.*, normalized $L(n)/p$)	Avg. # of pointers per node (*i.e.*, $1/(1-p)$)
$1/2$	1	2
$1/e$	0.94...	1.58...
$1/4$	1	1.33...
$1/8$	1.33...	1.14...
$1/16$	2	1.07...

Zrušení prvku

- Vyhledávacím algoritmem nalezněte pozici pro zrušení prvku
  - zapamatujte pozici předchůdce
- Zrušte uzel, je-li to nutné zmenšete MaxLevel.

```
Delete(list, searchKey)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then
        for i := 1 to list→level do
            if update[i]→forward[i] ≠ x then break
            update[i]→forward[i] := x→forward[i]
        free(x)
        while list→level > 1 and
                list→header→forward[list→level] = NIL do
            list→level := list→level – 1
```

Vyhledávání řetězců (Pattern Matching)



1. Co je vyhledávání řetězců ?

- Definice:
  - Pro daný textový řetězec T and a vzorový řetězec P, hledáme vzor P uvnitř textu
    - T: "the rain in spain stays mainly on the plain"
    - P: "n th"
- Aplikace:
  - textové editory, webové vyhledávače (např. Google), analýza obrazů, strukturní rozpoznávání

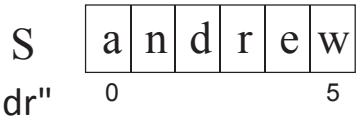
Přehled

1. Co je vyhledávání řetězců
2. Algoritmus „hrubé síly" (Brute-force)
3. Algoritmus Boyer-Moore
4. Knuth-Morris-Pratt algoritmus
5. Rabin-Karp algoritmus

Základní terminologie

- Předpokládejme, že S řetězec velikosti m.
- *podřetězec* S[i .. j] S je část řetězce mezi indexy i a j.
- *prefix (předpona)* S je podřetězec S[0 .. i]
- *suffix (přípona)* S je podřetězec S[i .. m-1]
  - i libovolný index mezi 0 a m-1

Příklad



- Podřetězec S[1..3] == "ndr"
- Všechny možné prefixy S:
  - "andrew", "andre", "andr", "and", "an", "a"
- Všechny možné suffixy S:
  - "andrew", "ndrew", "drew", "rew", "ew", "w"

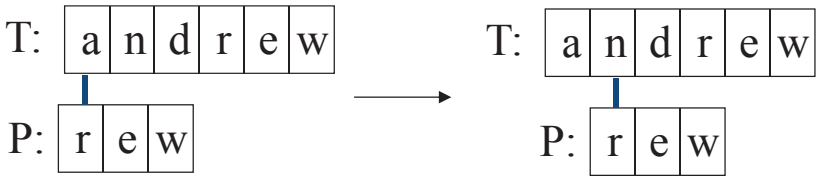
Brute Force v Javě

Vrací pozici, ve které začíná vzor, nebo -1

```
public static int bfMatching(String text,String template,int i){
    int j,
    int ret_val=-1;
    int n=text.length();
    boolean find=false;
   m=template.length();
   while (i<=n-m && !find) {
 j=0;
   while ((j<m) && (text.charAt(i+j)==template.charAt(j))) {
          j=j+1;
   }
   if (j==m) { ret_val=i;
               find=true;
             }
   i=i+1;
   }
   return(ret_val);
 }
```

2. Algoritmus „hrubé síly“ (Brute Force Algorithm)

- Pro každou pozici v textu T kontrolujeme zda v ní nezačíná vzor P.



P se posouvá po 1 znaku přes T

. . . .

Použití

```
public static void main(String[] args) {
     String text="pokus pohled pohoda podpora";
     String tmpl="po";
     int i;
     boolean nalezen=true;
     i=0;
     do { i=bfMatching(text,tmpl,i);
          if (i>=0) System.out.println("Nalezen v pozici
                                        i="+i);
             else nalezen=false;
          i=i+1;
     } while (nalezen);
}
```

Analýza

- Časová složitost Brute force algoritmu je O(mn) – nejhorší případ
- Většina vyhledáávání v běžném textu má složitost O(m+n).

continued

- Příklad – nejhorší případ:
  - T: "aaaaaaaaaaaaaaaaaaaaaaaaah"
  - P: "aaah"

- Brute force algoritmus je rychlý, pokud je abeceda textu „velká“
  - např. A..Z, a..z, 1..9, atd.

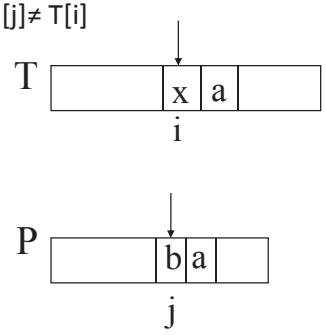
- Algoritmus je pomalý pro „malou“ abecedu
  - tj. 0, 1 (binární soubory, obrázkové soubory, atd.)

3. Boyer-Moore Algoritmus

- Boyer-Moore algoritmus vyhledávání je založen na

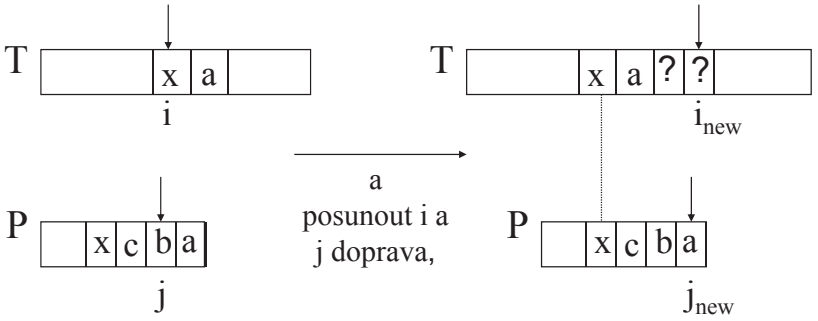
- 1. Zrcadlovém přístupu k vyhledávání
  - hledáme P v T tak, že začínáme na konci P a postupujeme zpět k začátku

- 2. Přeskočením skupiny znaků, které se neshodují (pokud takové znaky existují)
  - Tento případ se řeší v okamžiku kdy P[j]≠ T[i]
  - mohou nastat celkem 3 případy.



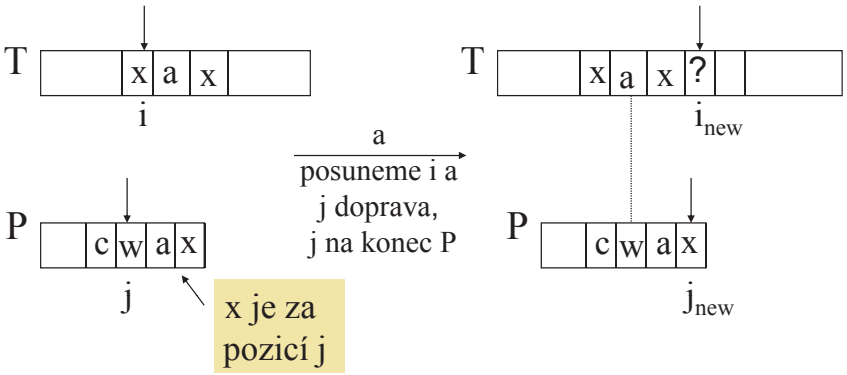
Případ 1

- Pokud P obsahuje x, pak zkusíme posunout *P* doprava tak, aby se poslední výskyt x dostal proti x obsaženému v T[i].



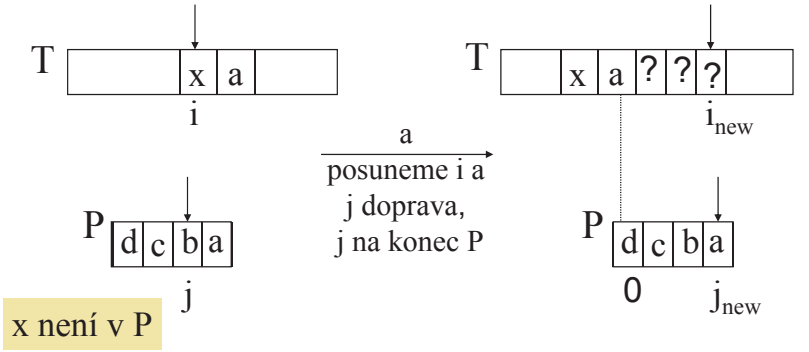
Případ 2

- P obsahuje x, ale posun doprava na poslední výskyt x není možný, pak posuneme P doprava o jeden znak k T[i+1].

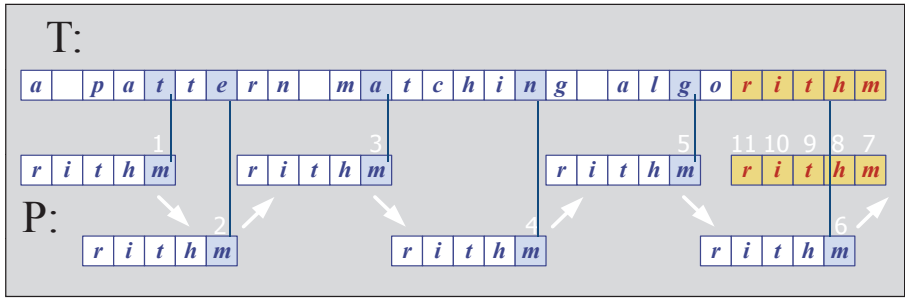


Případ 3

- Pokud není možné požít případ 1 a 2, pak posuneme P tak aby bylo P[0] zarovnáno s T[i+1].



Boyer-Moore příklad (1)



Příklad funkce Last()

- A = {a, b, c, d}
- P: "abacab"

P	a	b	a	c	a	b
	0	1	2	3	4	5

x	*a*	*b*	*c*	*d*
***Last*(*x*)**	4	5	3	-1

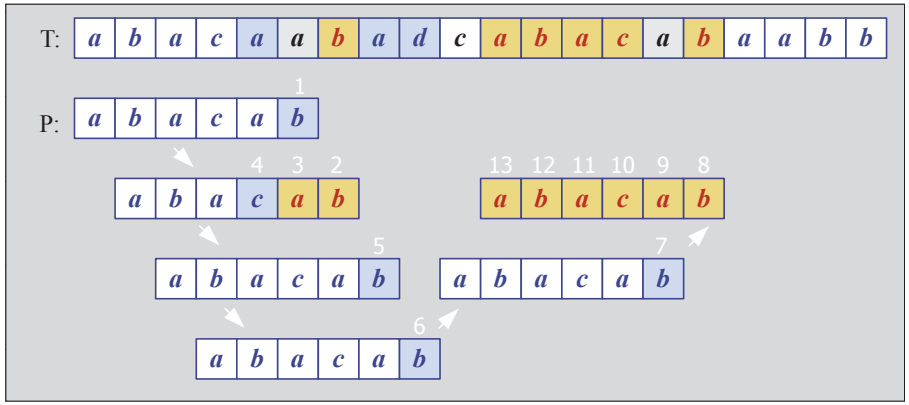
Funkce Last()

- Boyer-Moore algoritmus předzpracovává vzor P a pro danou abecedu A definuje funkci Last().
  - Last() zobrazuje všechny znaky abecedy A do množiny celých číslel
- Last(x) je definována jako : // x je znak v A
  - Největší index i pro který platí, že P[i] == x, nebo
  - -1 pokud žádný takový index v P neexistuje

Poznámka

- Last() se počítá pro každý vzor P před začátkem vyhledávání.
- Last() s obvykle uchovává jako pole (tabulka)

Boyer-Moore příklad (2)



x	*a*	*b*	*c*	*d*
L(*x*)	4	5	3	–1

```
      int j = m-1;
      do {
        if (pattern.charAt(j) == text.charAt(i))
          if (j == 0)
            return i; // match
          else { // zpětný průchod
            i--;
            j--;
          }
        else { // přeskočení znaků
           int lo = last[text.charAt(i)];  //last occ
           i = i + m - Math.min(j, 1+lo);
           j = m - 1;
        }
      } while (i <= n-1);

      return -1; // není shoda
} // konec algoritmu
```

Boyer-Moore in Javě

Vrací index ve kterém začíná vzor nebo -1

```
public static int bmMatch(String text,
                          String pattern)
{
  int last[] = buildLast(pattern);
  int n = text.length();
  int m = pattern.length();
  int i = m-1;

  if (i > n-1)
    return -1; // není shoda - vzor je
       // delší než text
```

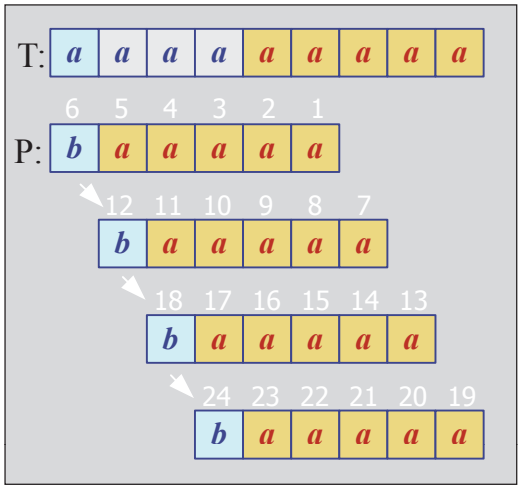
```
 public static int[] buildLast(String pattern)
/* vrací pole indexů posledního výskytu každého
    znaku ve vzoru */
 {
   int last[] = new int[128]; // ASCII znaky

   for(int i=0; i < 128; i++)
     last[i] = -1; // initializace

   for (int i = 0; i < pattern.length(); i++)
     last[pattern.charAt(i)] = i;

   return last;
 } // end of buildLast()
```

Použití

```
public static void main(String args[])
 { if (args.length != 2) {
     System.out.println("Usage: java BmSearch
                         <text> <pattern>");
     System.exit(0);
   }
   System.out.println("Text: " + args[0]);
   System.out.println("Pattern: " + args[1]);

   int posn = bmMatch(args[0], args[1]);
   if (posn == -1)
     System.out.println("Pattern not found");
   else
     System.out.println("Pattern starts at posn "
                                        + posn);
}
```

Příklad nejhoršího případu

- T: "aaaaa…a"
- P: "baaaaa"



Analýza

- Časová složitost Boyer-Moore algoritmu je v nejhorším případě O(nm + A)
- Boyer-Moore je rychlejší pokud je abeceda (A) velká, pomalý pro malou abecedu

  tj. algoritmus je vhodný pro text, špatný pro binární vstupy
- Boyer-Moore rychlejší než *brute force* v případě vyhledávání v textu.

4. KMP Algoritmus

- Knuth-Morris-Pratt (KMP) algoritmus vyhledává vzor v textu *zleva do prava* (jako brute force algoritmus).
- Posun vzoru je řešen mnohem inteligentněji než v brute force algoritmu.

- Pokud se vyskytne neshoda mezi textem and vzorem P v P[j], jaký je *největší možný posun* vzoru abychom se vyhnuly zbytečnému porovnávání?

- *Odpověď*: největší prefix P[0 .. j-1], který je suffixem P[1 .. j-1]

Příklad

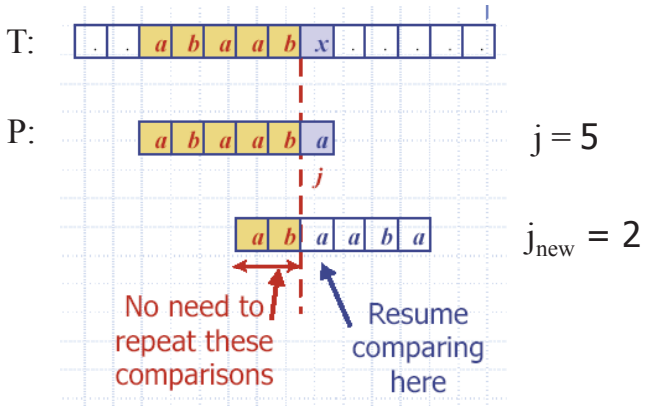
j == 5

- Nalezneme největší prefix (start) :
  "a b a a b" (P[0..j-1])

  jehož suffix (end) :
  "b a a b" (p[1 .. j-1])

- Odpověď: "a b"
- Nastavíme j = 2 // nová hodnota j

Příklad



KMP chybová funkce

- KMP předzpracovává vzor, abychom nalezli shodu prefixů vzoru se sebou samým.
- k = pozice před neshodou (j-1).
- *Chybová funkce (failure function)* F(k) definována jako nejdelší prefix P[0..k] který je také suffixem P[1..k].

Příklad chybové funkce

(k == j-1)

- P: "abaaba"

k	0	1	2	3	4	5
F(k)	0	0	1	1	2	3

F(k) velikost největšího prefixu, který je zároveň sufixem

- V programu je F() implementována,jako pole (popř. tabulka.)

KMP v Javě

```
public static int kmpMatch(String text,
                           String pattern)
 {
   int n = text.length();
   int m = pattern.length();

   int fail[] = computeFail(pattern);

   int i=0;
   int j=0;
     :
```

Použití chybové funkce

- Knuth-Morris-Pratt algoritmus modifikuje brute-force algoritmus.
  - Pokud se vyskytne neshoda v P[j] (i.e. P[j] != T[i]), pak
    k = j-1;
    j = F(k); // získání nové hodnoty j

```
  while (i < n) {
    if (pattern.charAt(j) == text.charAt(i)) {
      if (j == m - 1)
        return i - m + 1; // match
      i++;
      j++;
    }
    else if (j > 0)
  j = fail[j-1];
    else
      i++;
  }
  return -1; // no match
} // end of kmpMatch()
```

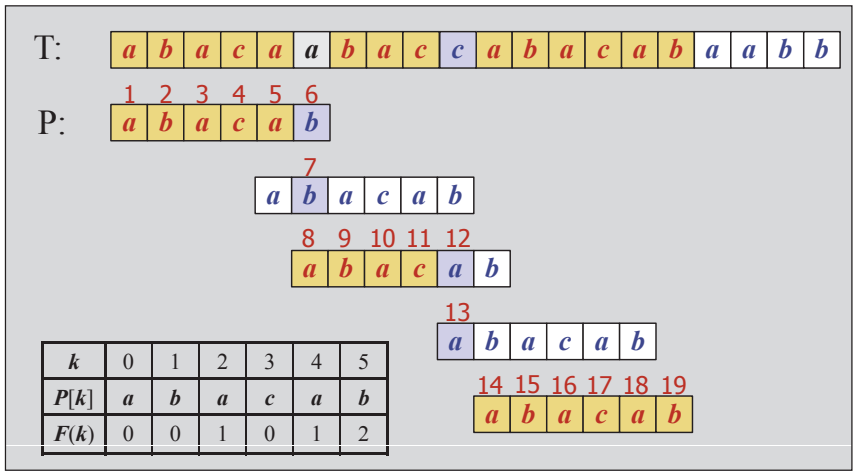
```
public static int[] computeFail(
                            String pattern)
{
  int fail[] = new int[pattern.length()];
  fail[0] = 0;

  int m = pattern.length();
  int j = 0;
  int i = 1;
 :
```

Použití

```
public static void main(String args[])
{ if (args.length != 2) {
    System.out.println("Usage: java KmpSearch
                              <text> <pattern>");
    System.exit(0);
  }
  System.out.println("Text: " + args[0]);
  System.out.println("Pattern: " + args[1]);

  int posn = kmpMatch(args[0], args[1]);
  if (posn == -1)
    System.out.println("Pattern not found");
  else
    System.out.println("Pattern starts at posn "
                                        + posn);
}
```

```
while (i < m) {
    if  (pattern.charAt(j) ==
         pattern.charAt(i)) {    //j+1 chars match
      fail[i] = j + 1;
      i++;
      j++;
    }
    else if (j > 0) // j follows matching prefix
      j = fail[j-1];
    else {      // no match
      fail[i] = 0;
      i++;
    }
  }
  return fail;
} // end of computeFail()
```

Příklad



k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0	0	1	0	1	2

KMP výhody

- KMP běží v optimálním čase: O(m+n)
- Algoritmus se nikdy neposouvá zpět ve vstupním textu T
  - To činí algoritmus obzvlášť výhodný zpracování velkých souborů

Rabin-Karp Algoritmus

- Výpočet kontrolního součtu:
  - Zvolíme prvočíslo q
  - Zvolíme d = $|\Sigma|$ - tj. počet všech možných znaků v použité abecedě

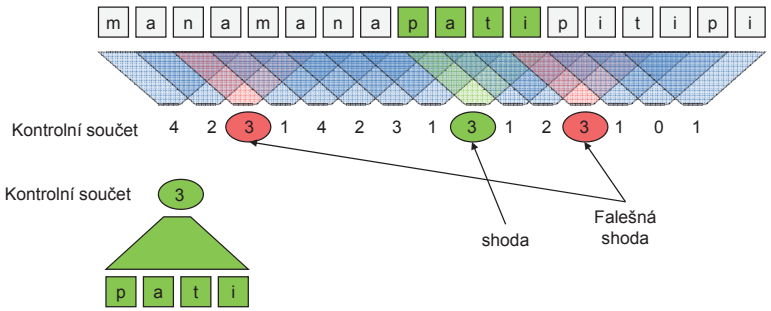
$$S_m(P) = \sum_{i=1}^{m} d^{m-i} P[i] \bmod q = P[m] + dP[m-1] + \ldots + d^{m-2} P[2] + d^{m-1} P[1] \bmod q$$

- Příklad:
  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - Pak d = 10, q = 13
  - Nechť P = 0815

  $S_4(0815) = (0 \cdot 1000 + 8 \cdot 100 + 1 \cdot 10 + 5 \cdot 1) \bmod 13 = 815 \bmod 13 = 9$

5. Rabin-Karp Algoritmus

- Základní myšlenka: Vypočítat
  - kontrolní součet pro vzor P (délky m) a
  - kontrolní součet pro každý podřetězec řetězce T délky m
  - procházet řetězcem T a porovnat kontrolní součet každého podřetězce s kontrolním součtem vzoru. Pokud dojde ke shodě vzoru provést test znak po znaku.



Jak vypočítat kontrolní součet : Hornerovo schéma

- Máme vypočítat $S_m(P) = \sum_{i=1}^{m} d^{m-i} P[i] \bmod q$
- Použitím

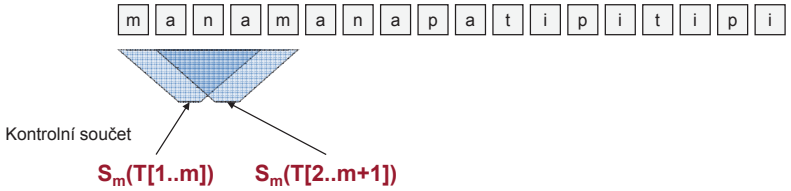
$$S_m(P) = \sum_{i=1}^{m} d^{m-i} P[i] = d\left(\sum_{i=1}^{m-1} d^{m-i-1} P[i]\right) + P[m] = dS_{m-1}(P[1..m-1]) + P[m] \pmod q$$

- Příklad:
  - $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - Pak d = 10, q = 13
  - Nechť P = 0815

  $S_4(0815) = ((((\mathbf{0} \cdot 10 + \mathbf{8}) \cdot 10) + \mathbf{1}) \cdot 10) + \mathbf{5} \bmod 13 =$
  $((((\mathbf{8} \cdot 10) + \mathbf{1}) \cdot 10) + \mathbf{5} \bmod 13 =$
  $(3 \cdot 10) + \mathbf{5} \bmod 13 = 9$

Jak vypočítat kontrolní součet pro text

•Začneme s $S_m(T[1..m])$



$$S_m\left((T[2..m+1]\right) \equiv d\left(S_m\left(T[1..m]\right) - d^{m-1}T[1]\right) + T[m+1]\right) \bmod q$$

Rabin-Karp Algoritmus

```
Rabin-Karp-Matcher(T,P,d,q)
1.  n ← length(T)
2.  m ← length(P)
3.  h ← d^(m-1) mod q
4.  p ← 0
5.  t0 ← 0
6.  for i ← 1 to m do
7.          p ← (d p + P[i]) mod q
8.          t0 ← (d t0 + T[i]) mod q
    od
9.  for s ←  0 to n-m do
10.         if p = ts then
11.                 if P[1..m] = T[s+1..s+m] then return "Pattern occurs with shift" s
            fi
12.         if s < n-m then
13.                 ts+1 ← (d(ts-T[s+1]h) + T[s+m+1]) mod q
            fi
    od
```

Kontrolní součet vzoru P

Kontrolní součet textu T[1..m]

Kontrolní součty se shodují provádí se test shody řetězců

výpočet kontrolního součtu pro T[s+1..s+m] s využitím kontrolního součtu T[s..s+m-1]

Vlastnosti Rabin-Karp algoritmu

- čas běhu Rabin-Karp algoritmu je v nejhorším případě O(m (n-m+1))
- Pravděpodobnostní analýza
  - Pravděpodobnost falešné shody je pro náhodný vstup 1/q
  - Předpokládaný počet falešných shod O(n/q)
  - Předpokládaný čas běhu Rabin-Karp algoritmu je O(n + m (v+n/q)) kde v je počet správných posuvů
- Pokud zvolíme q ≥ m a očekávaný počet posuvů je malý je předpokládaná doba běhu Rabin-Karp algoritmu O(n +m).

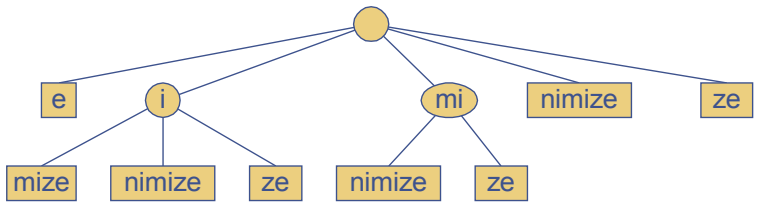
Algoritmy zpracování textů II

- datová struktura Trie
- nejdelší společná sekvence (LCS)
- nejkratší společná nad-sequence (SCS)
- vzdálenost mezi řetězci

Předzpracování řetězců

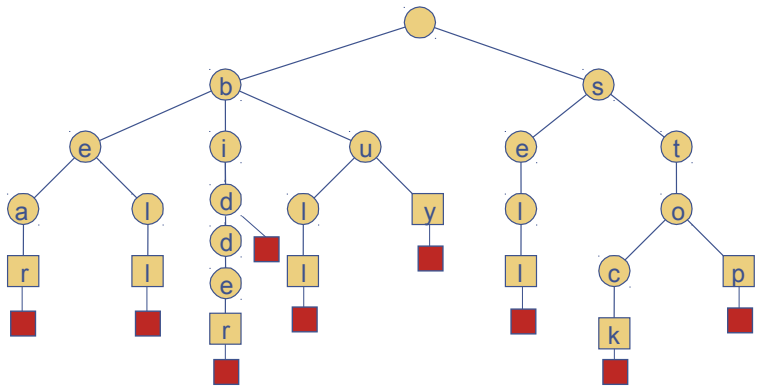
- U algoritmů vyhledávání řetězců se předzpracovává hledaný vzor, aby se urychlilo jeho vyhledávání
- Pro rozsáhlé neměnné texty ve kterých se často vyhledává je výhodnější předzpracovat celý text, než se zabývat předzpracováním vzoru (BM, KMP algoritmus)
- Trie je kompaktní datová struktura vhodná pro reprezentaci množiny retězců, kterými mohou být např. slova v textu
  - Trie umožňuje vyhledávat řetězce v čase úměrném velikosti hledaného vzoru

Datová struktura Trie



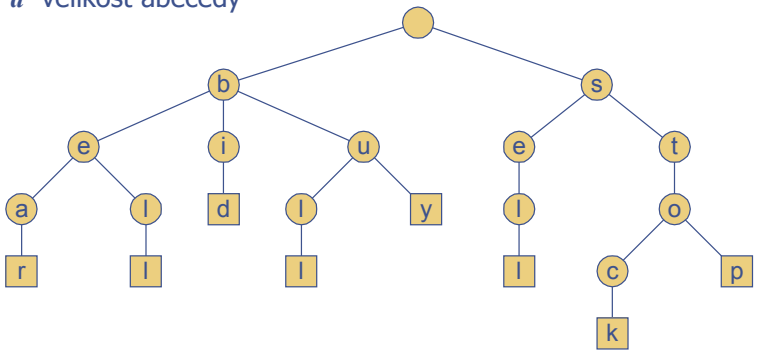
Standardní Trie

- Standardní trie pro množinu řetězců S je k-ární (k je velikost použité abecedy) uspořádaný strom, pro který platí:
  - Každý uzel, kromě kořene, je ohodnocen znakem
  - Následníci uzlu jsou abecedně uspořádány
  - Symboly v uzlech na cestě z kořene do externího uzlu tvoří řetězec množiny S
- Příklad: standardní trie pro množinu řetězců
  S = { bear, bell, bid, bull, buy, sell, stock, stop }



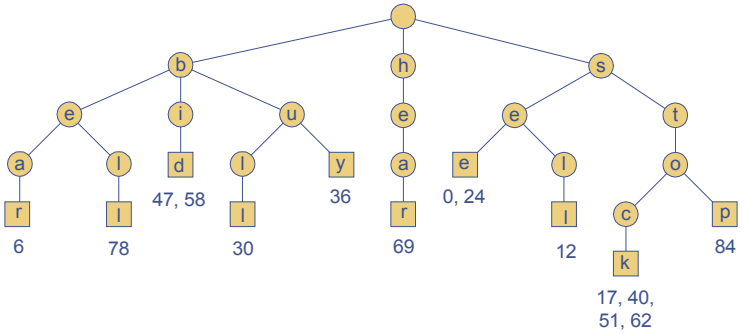
Analýza Standardní Trie

- Standardní trie vyžaduje $O(n)$ paměťového prostoru a umožňuje vyhledávání, vkládání a rušení v čase $O(dm)$, kde:
  - n celková velikost řetězců v S
  - m velikost zpracovávaného řetězce
  - d velikost abecedy



Vyhledávání slov pomocí Trie

- Slova z textu jsou uložena do trie
- V každém listu je zároveň uložena informace o pozici výskytu slova v textu

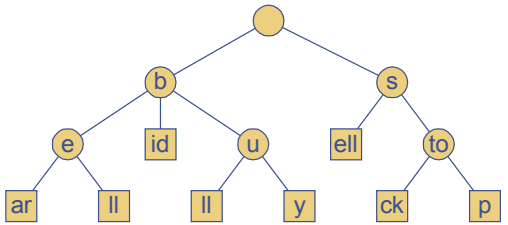


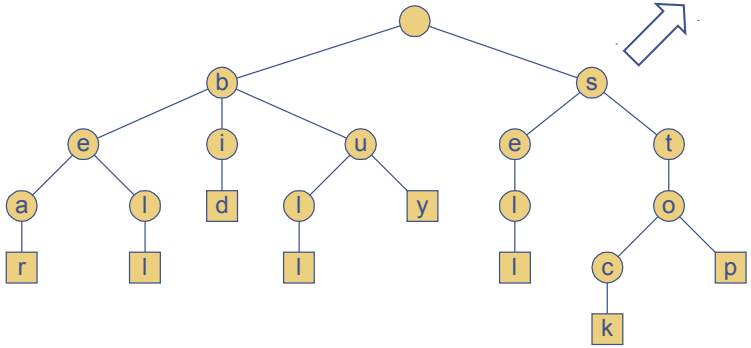
Typické použití datové struktury Trie

- Standardní trie umožňuje provádět následující operace nad předzpracovaným textem v čase O(m), kde m velikost slova X:
  - **Vyhledávání slov (Word Matching):** nalezení prvního výskytu slova X v textu.
  - **Vyhledávání prefixu (Prefix Matching):** Nalezení prvního výskytu nejdelšího prefixu slova X v textu.

Komprimovaná Trie

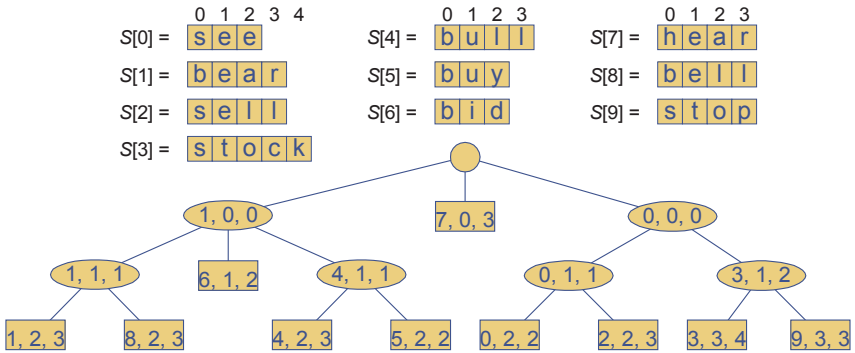
- Komprimovaná trie má vnitřní uzly stupně nejméně 2
- Získáva se ze standardní trie, kompresí řetězců tzv. „redundantních" uzlů tj. uzlů, které mají pouze jednoho následníka





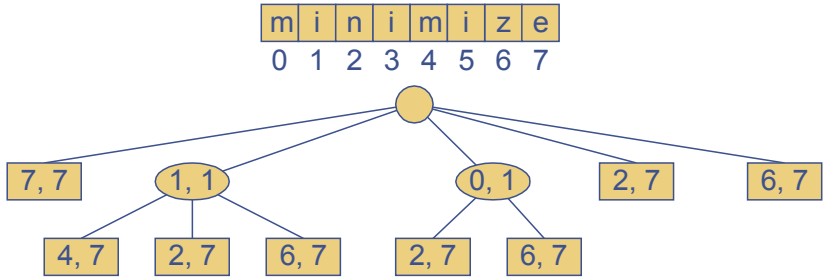
Kompaktní reprezentace komprimované Trie

- Kompaktní reprezentace komprimované trie pro pole řetězců:
  - Uchovává v uzlech trojici indexů (i,j,k) místo celých řetězců.
    i – index v poli (tabulce), kde je řetězec uložen
    j – počáteční index podřetězce uloženého v uzlu
    k – koncový index podřetězce uloženého v uzlu
  - Využívá $O(s)$ paměťového prostoru, kde s je počet řetězců v poli
  - Slouží jako pomocná indexová struktura



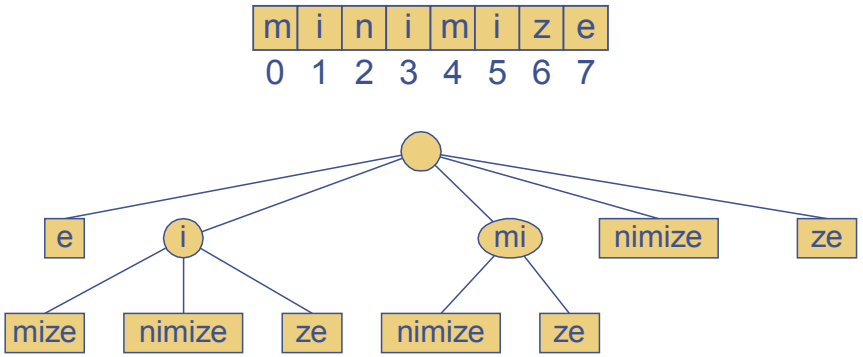
Analýza Suffixové Trie

- Kompaktní reprezentace suffixové trie řetězce X velikosti n vzniklého z abecedy mohutnosti d
  - Využívá $O(n)$ paměťového prostoru.
  - Umožňuje libovolné pokládání dotazů na přítomnost řetězce v textu X v čase $O(dm)$, kde m je velikost vzorového řetězce
  - Lze ji vytvořit v čase $O(n)$.



Suffixová Trie

- Suffixová trie řetězce X je komprimovaná trie všech suffixů X



Algoritmus vyhledávání řetězců suffixovou Trie

Algorithm suffixTrieMatch(T,P):
Input: Compact suffix trie T for a text X and pattern P
Output: Starting index of a substring of X matching P or an indication that P is not a substring of X

```
p ← P.length() { length of suffix of the pattern to be matched }
j ← 0 { start of suffix of the pattern to be matched }
v ← T.root()
repeat
  f ← true { flag indicating that no child was successfully processed }
  for each child w of v do
    i ← start(v)
    if P[j] = T[i] then
      { process child w }
      x ← end(w) − i + 1
      if p ≤ x then
        { suffix is shorter than or of the same length of the node label }
        if P[j..j + p − 1] = X[i..i + p − 1] then
          return i − j { match }
        else
          return "P is not a substring of X"
      else
        { suffix is longer than the node label }
        if P[j..j + x − 1] = X[i..i + x − 1] then
          p ← p − x { update suffix length }
          j ← j + x { update suffix start index }
          v ← w
          f ← false
          break out of the for loop
until f or T.isExternal(v)
return "P is not a substring of X"
```

Trie a Webové vyhledávání

- kolekce všech vyhledávaných slov (tzv. search engine index) je uchováván v komprimované trie.
- Každý uzel trie odpovídá hledanému slovu a je zároveň spojen se seznamem stránek (URLs) obsahující toto slovo - tzv. seznam výskytů (**occurrence list**).
- Trie se uchovává v interní paměti.
- Seznam výskytů se uchovává v externí paměti a jsou uspořádány podle důležitosti

LCS – Longest common subsequence

Algoritmus nalezení nejdelšího společného podřetězce

- LCS algoritmus je jedním ze způsobů jak posuzovat podobnost mezi dvěma řetězci
- algoritmus se často využívá v biologii k posuzování podobnosti DNA sekvencí (řetězců obsahujících symboly A,C,G,T)
- Příklad X = AGTCAACGTT, Y=GTTCGACTGTG
- Podřetězce jsou např. S = AGTG and S'=GTCACGT
- Jak lze tyto podřetězce nalézt ?
  - Použitím hrubé síly : pokud |X| = m, |Y| = n, pak existuje 2^m podřetězců x, které musíme porovnat s Y (n porovnání) tj. časová složitost vyhledání je $O(n\,2^m)$
  - Použití dynamického programování – složitost se sníží na O(nm)

Platí :

- Nechť $X=<x_1,x_2,\ldots,x_m>$ a $Y=<y_1,y_2,\ldots y_n>$ jsou řetězce a $Z=<z_1,z_2,\ldots,z_k>$ je libovolná LCS X a Y
- Jestliže $x_m = y_n$ pak $z_k = x_m = y_n$ a Z_{k-1} je LCS X_{m-1} a Y_{n-1}
- Jestliže $x_m \neq y_n$ a $z_k \neq x_m$, pak z toho vyplývá, že Z je LCS X_{m-1} a Y
- Jestliže $x_m \neq y_n$ a $z_k \neq y_n$, pak Z je LCS X a Y_{n-1}

Postup:

- Nejprve nalezneme délku LCS a podél „cesty", kterou budeme procházet, si budeme nechávat značky, které nám pomohou nalézt výslednou nejdelší společnou sekvenci
- Nechť X_i, Y_j jsou prefixy X a Y délky i a j .
- Nechť $c[i,j]$ je délka LCS X_i and Y_j
- Pak délka kompletní LCS X a Y bude $c[m,n]$

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{ve zbývajících situacích} \end{cases}$$

Rekurentní řešení

- Začneme s $i = j = 0$ (prázdné podřetězce x a y)
- Protože X_0 and Y_0 jsou prázdné řetězce je jejich LCS vždy prázdná (tj. $c[0,0] = 0$)
- LCS prázdného řetězce a libovolného jiného řetězce je také prázdná a tak pro každé i a j :

$$c[0, j] = c[i,0] = 0$$

- když určujeme hodnotu c[i,j], tak uvažujeme dva případy:
  - **První případ:** $x[i]=y[j]$: další symbol v řetězci X and Y se shoduje a délka LCS X_i a Y_j je rovna délce LCS kratších řetězců X_{i-1} a Y_{i-1} , zvětšená o 1.
  - **Druhý případ:** $x[i] != y[j]$ tj. symboly se neshodují a tudíž se délka LCS(X_i,Y_j) nezvětší a zůstává shodná jako předtím (tj. maximum z LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j))

LCS Algoritmus

```
LCS-Length(X, Y)
m = length(X),  n  = length(Y)
for i = 1 to m
     do c[i, 0] = 0
for j = 0 to n
      do c[0, j] = 0
for i = 1 to m
        do for j = 1 to n
              do if ( xi = = yj )
                      then c[i, j] = c[i - 1, j - 1] + 1
                           b[i, j] =" ←↑ "
              else if c[i - 1, j]>=c[i,  j - 1]
                              then  c[i, j] = c[i - 1, j]
                                    b[i, j] =" ↑ "
                              else c[i, j] = c[i, j - 1]
                                    b[i, j] =" ←"

return c and b
```

Příklad:

- Hledáme nejdelší společný podřetězec (LCS) řetězců
  - X = ABCB
  - Y = BDCAB

LCS(X, Y) = BCB

- X = A **B** **C** **B**
- Y = **B** D **C** A **B**

LCS příklad

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[6,5]

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi						
1	A	0					
2	B	0					
3	C	0					
4	B	0					

for i = 1 to m c[i,0] = 0

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0					
2	B	0					
3	C	0					
4	B	0					

for j = 0 to n c[0,j] = 0

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0 ↑				
2	B	0					
3	C	0					
4	B	0					

case i=1 and j=1
A != B
but, c[0,1]>=c[1,0]
so c[1,1] = c[0,1], and b[1,1] = ↑

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑			
2	B	0					
3	C	0					
4	B	0					

case i=1 and j=2
A != D
but, c[0,2]>=c[1,1]
so c[1,2] = c[0,2], and b[1,2] = ↑

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑	0 ↑		
2	B	0					
3	C	0					
4	B	0					

case i=1 and j=3
A != C
but, c[0,3]>=c[1,2]
so c[1,3] = c[0,3], and b[1,3] = ↑

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←
2	B	0					
3	C	0					
4	B	0					

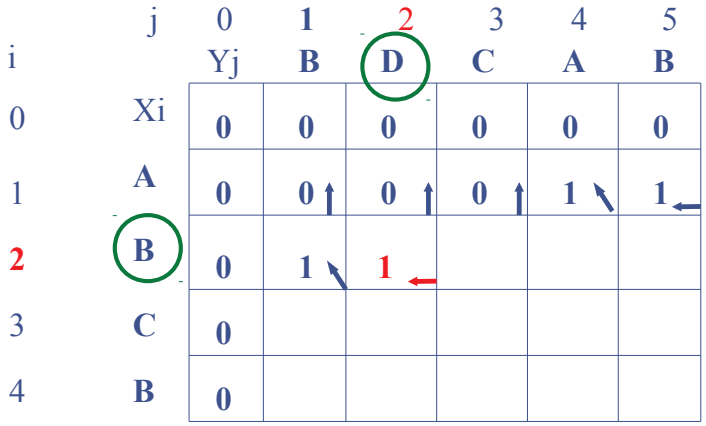
case i=1 and j=5
A != B
this time c[0,5]<c[1,4]
so c[1,5] = c[1, 4], and b[1,5] = ←

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑	0 ↑	1 ↖	1
2	B	0					
3	C	0					
4	B	0					

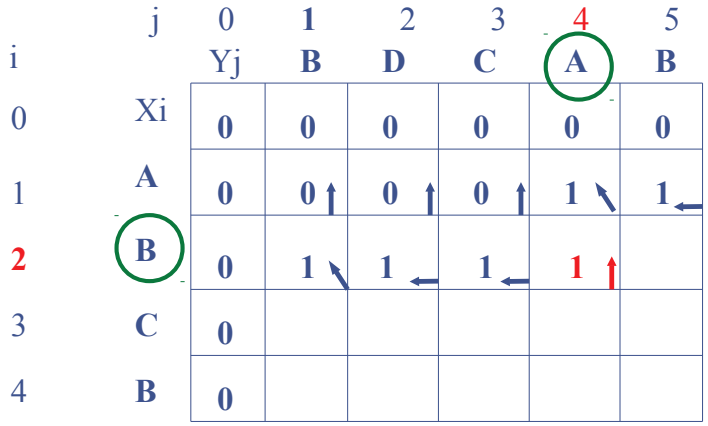
case i=1 and j=4
A = A
so c[1,4] = c[0,2]+1, and b[1,4] = ↖

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑	0 ↑	1 ↖	1
2	B	0	1 ↖				
3	C	0					
4	B	0					

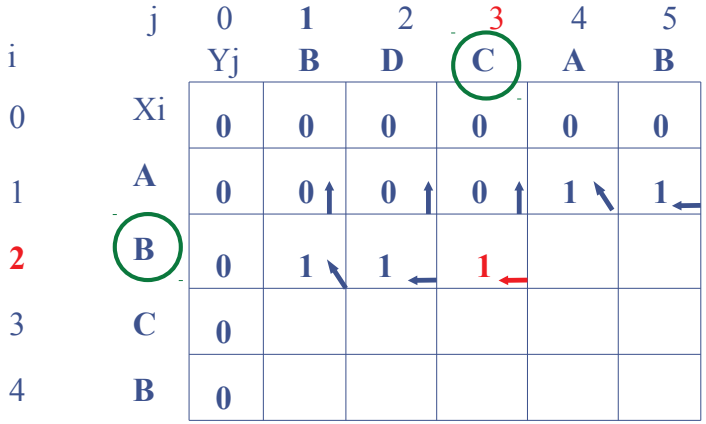
case i=2 and j=1
B = B
so c[2, 1] = c[1, 0]+1, and b[2, 1] = ↖



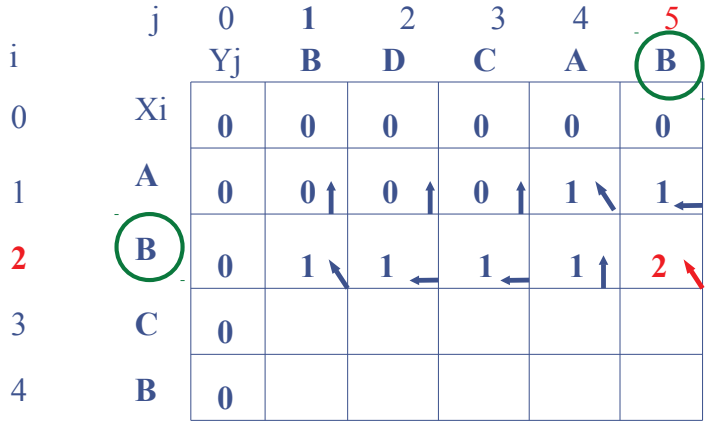
case i=2 and j=2
B != D
and c[1, 2] < c[2, 1]
so c[2, 2] = c[2, 1] and b[2, 2] = ←



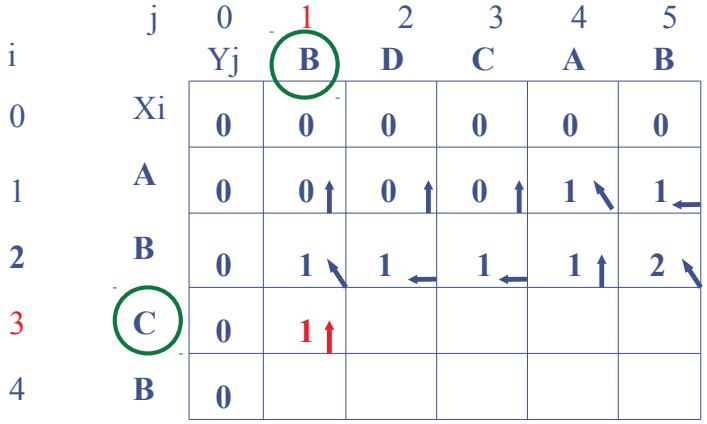
case i=2 and j=4
B != A
and c[1, 4] = c[2, 3]
so c[2, 4] = c[1, 4] and b[2, 2] = ↑



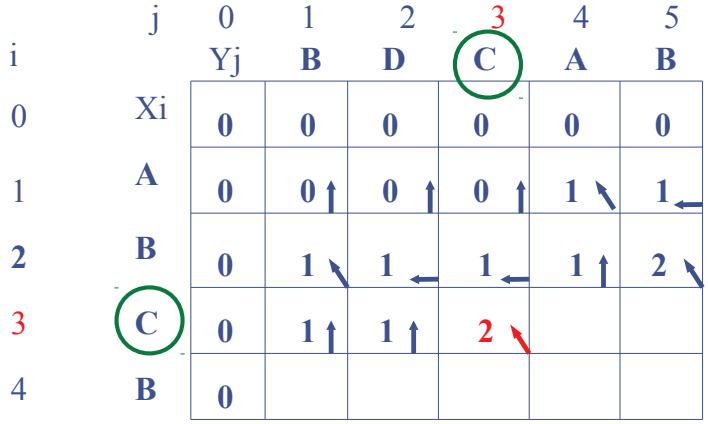
case i=2 and j=3
B != D
and c[1, 3] < c[2, 2]
so c[2, 3] = c[2, 2] and b[2, 3] = ←



case i=2 and j=5
B = B
so c[2, 5] = c[1, 4]+1 and b[2, 5] = ↖

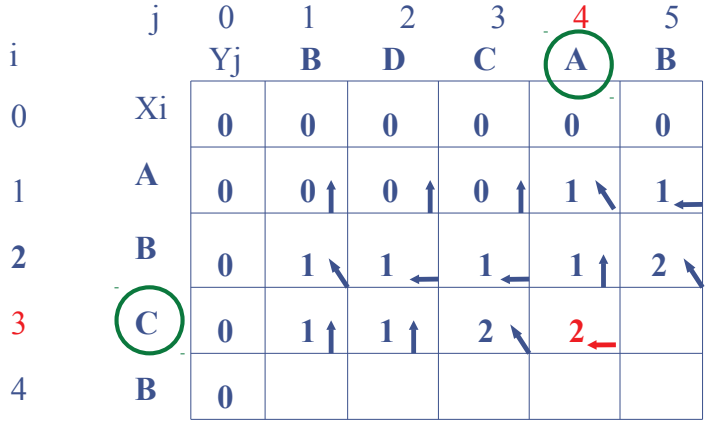


case i=3 and j=1
C != B
and c[2, 1] > c[3,0]
so c[3, 1] = c[2, 1] and b[3, 1] =



case i=3 and j= 3
C = C
so c[3, 3] = c[2, 2]+1 and b[3, 3] =

case i=3 and j= 2
C != D
and c[2, 2] = c[3, 1]
so c[3, 2] = c[2, 2] and b[3, 2] =



case i=3 and j= 4
C != A
c[2, 4] < c[3, 3]
so c[3, 4] = c[3, 3] and b[3, 3] =

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←
2	B	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖
3	C	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑
4	B	0					

case i=3 and j= 5
C != B
c[2, 5] = c[3, 4]
so c[3, 5] = c[2, 5] and b[3, 5] = ↑

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←
2	B	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖
3	C	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑
4	B	0	1 ↖	1 ↑			

case i=4 and j=2
B != D
c[3, 2] = c[4, 1]
so c[4, 2] = c[3, 2] and b[4, 2] = ↑

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←
2	B	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖
3	C	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑
4	B	0	1 ↖				

case i=4 and j=1
B = B
so c[4, 1] = c[3, 0]+1 and b[4, 1] = ↖

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←
2	B	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖
3	C	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑
4	B	0	1 ↖	1 ↑	2 ↑		

case i=4 and j= 3
B != C
c[3, 3] > c[4, 2]
so c[4, 3] = c[3, 3] and b[4, 3] = ↑

Nalezení LCS

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

case i=4 and j=4
B != A
c[3, 4] = c[4, 3]
so c[4, 4] = c[3, 4] and b[3, 5] = ↑

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

case i=4 and j=5
B= B
so c[4, 5] = c[3, 4]+1 and b[4, 5] = ↖

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

LCS (obrácené pořadí): **B C B**

LCS (správné pořadí): **B C B**

SCS – Shortest common super-sequence

Algoritmus nalezení nejkratšího společného „nadřetězce"

- Podobný algoritmu LCS
- **Definice:** Nechť X a Y jsou dva řetězce znaků. Řetězec Z je „nadřetězec" (**super-sequence)** řetězců X a Y pokud jsou oba řetězce X a Y podřetězcem (subsequence) Z.
- **Shortest common super-sequence algoritmus:**

  Vstup: dva řetězce X aY.

  Výstup: nejkratší společný „nadřetězec" X a Y.
- Příklad: X=**abc** a Y=**abb**. Oba řetězce **abbc abcb** jsou nejkratším společným „nadřetězcem" řetězců X a Y.

Rekurentní řešení

- Začneme s $i = j = 0$ (prázdné podřetězce x a y)
- Protože X_0 and Y_0 jsou prázdné řetězce je jejich SCS vždy prázdná (tj. $c[0,0] = 0$)
- SCS prázdného řetězce a libovolného jiného řetězce je rovná danému řetězci a tak pro každé i a j je délka :

  $c[0, j] = j$

  $c[i,0] = i$
- když určujeme hodnotu c[i,j], tak uvažujeme dva případy:
  - **První případ:** $x[i]=y[j]$: další symbol v řetězci X and Y se shoduje a délka SCS X_i a Y_j je rovna délce SCS kratších řetězců X_{i-1} a Y_{i-1} , zvětšená o 1.
  - **Druhý případ:** $x[i] != y[j]$ tj. symboly se neshodují a délka SCS(X_i,Y_j) je daná minimální hodnotou dvojice SCS(X_i, Y_{j-1}) and SCS(X_{i-1},Y_j)

Postup:

- Nejprve nalezneme délku SCS a podél „cesty", kterou budeme procházet, si budeme nechávat značky, které nám pomohou nalézt výslednou nejkratší společnou super-sekvenci
- Nechť X_i, Y_j jsou prefixy X a Y délky i a j .
- Nechť $c[i,j]$ je délka SCS X_i and Y_j
- Pak délka kompletní SCS X a Y bude v ***c[m,n]***

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \min(c[i,j-1]+1, c[i-1,j]+1) & \text{ve zbývajících situacích} \end{cases}$$

SCS Algoritmus

```
SCS-Length(X, Y)
m = length(X),  n  = length(Y)
for i = 1 to m
     do c[i, 0] = i
for j = 0 to n
      do c[0, j] = j
for i = 1 to m
        do for j = 1 to n
                do if ( xi = = yj )
                        then c[i, j] = c[i - 1, j - 1] + 1
                             b[i, j] =" ←↑ "
                else if c[i - 1, j]<=c[i,  j - 1]
                                then  c[i, j] = c[i - 1, j]+1
                                      b[i, j] =" ↑ "
                               else c[i, j] = c[i, j - 1]+1
                                      b[i, j] =" ←"

return c and b
```

Příklad:

- Hledáme nejkratší společný nadřetězec (SCS) řetězců
  - X = ABCB
  - Y = BDCAB

SCS(X, Y) = ABDCAB
- X = **A B C B**
- Y = **B D C A B**

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**	**1**					
2	**B**	**2**					
3	**C**	**3**					
4	**B**	**4**					

for i = 1 to m c[i,0] = i

LCS příklad

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[6,5]

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**1**	**2**	**3**	**4**	**5**
1	**A**	**1**					
2	**B**	**2**					
3	**C**	**3**					
4	**B**	**4**					

for j = 0 to n c[0,j] = j

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	1	2	3	4	5
1	A	1	2				
2	B	2					
3	C	3					
4	B	4					

case i=1 and j=1
A != B
but, c[0,1]+1<=c[1,0]+1
so c[1,1] = c[0,1]+1, and b[1,1] =

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	1	2	3	4	5
1	A	1	2	3	4		
2	B	2					
3	C	3					
4	B	4					

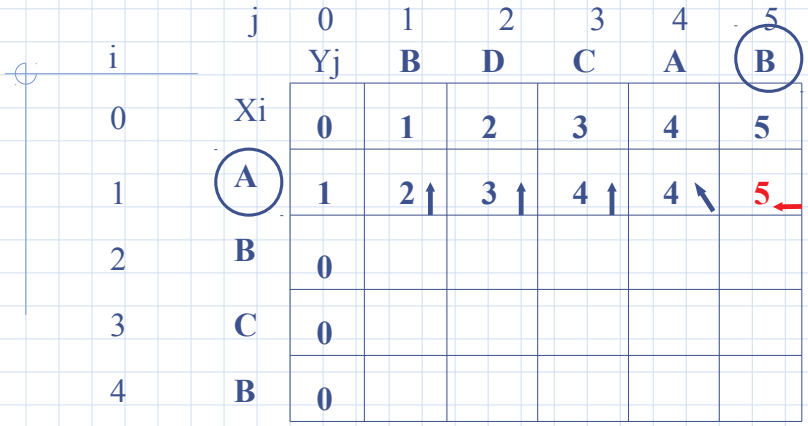
case i=1 and j=3
A != C
but, c[0,3]+1<=c[1,2]+1
so c[1,3] = c[0,3]+1, and b[1,3] =

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	1	2	3	4	5
1	A	1	2	3			
2	B	2					
3	C	3					
4	B	4					

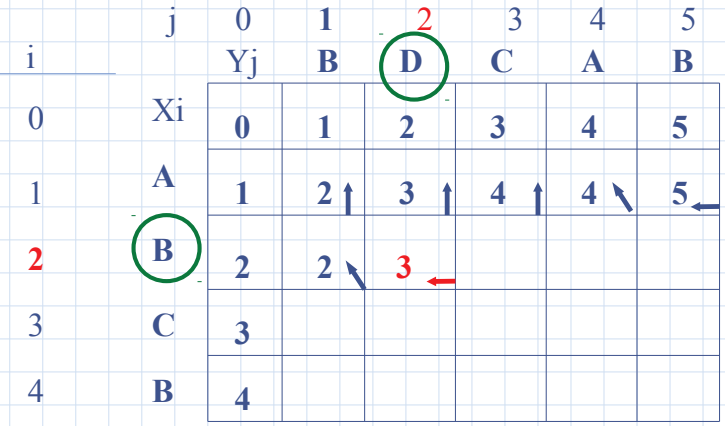
case i=1 and j=2
A != D
but, c[0,2]+1<=c[1,1]+1
so c[1,2] = c[0,2]+1, and b[1,2] =

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	1	2	3	4	5
1	A	1	2	3	4	4	
2	B	0					
3	C	0					
4	B	0					

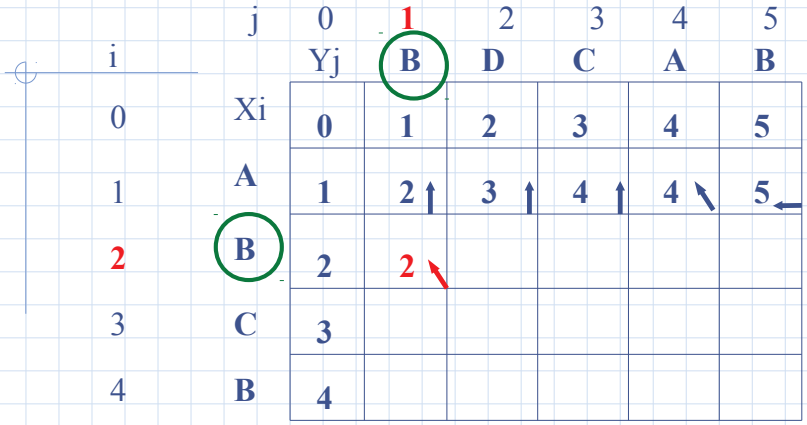
case i=1 and j=4
A = A
so c[1,4] = c[0,2]+1, and b[1,4] =



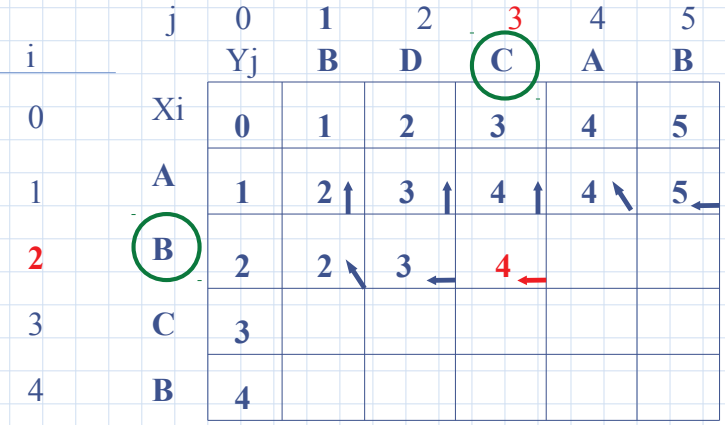
case i=1 and j=5
A != B
this time c[0,5]+1<c[1,4]+1
so c[1,5] = c[1, 4]+1, and b[1,5] = ←



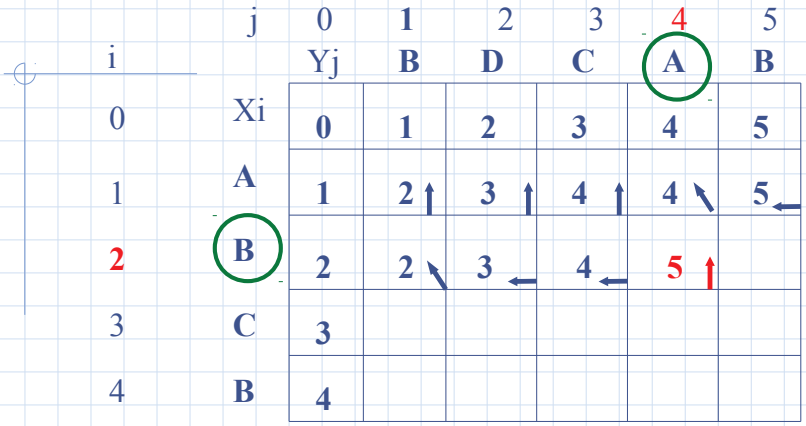
case i=2 and j=2
B != D
and c[1, 2]+1 > c[2, 1]+1
so c[2, 2] = c[2, 1]+1 and b[2, 2] = ←



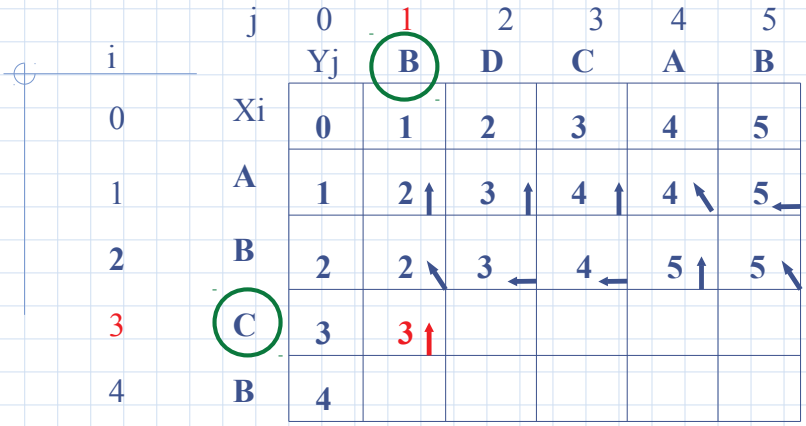
case i=2 and j=1
B = B
so c[2, 1] = c[1, 0]+1, and b[2, 1] = ↖



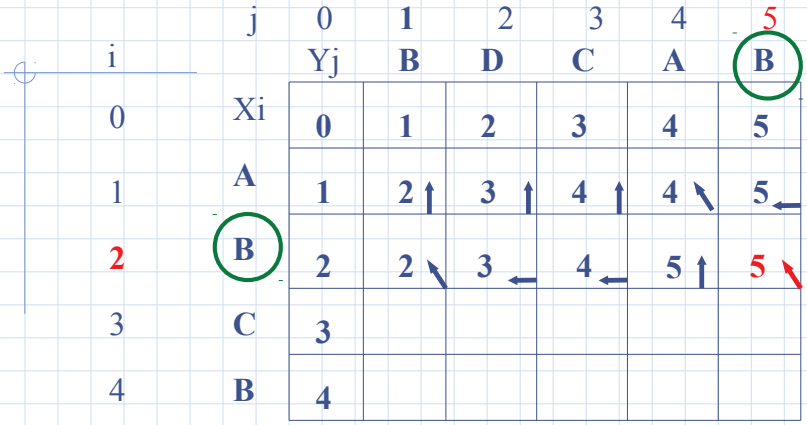
case i=2 and j=3
B != D
and c[1, 3]+1 > c[2, 2]+1
so c[2, 3] = c[2, 2]+1 and b[2, 3] = ←



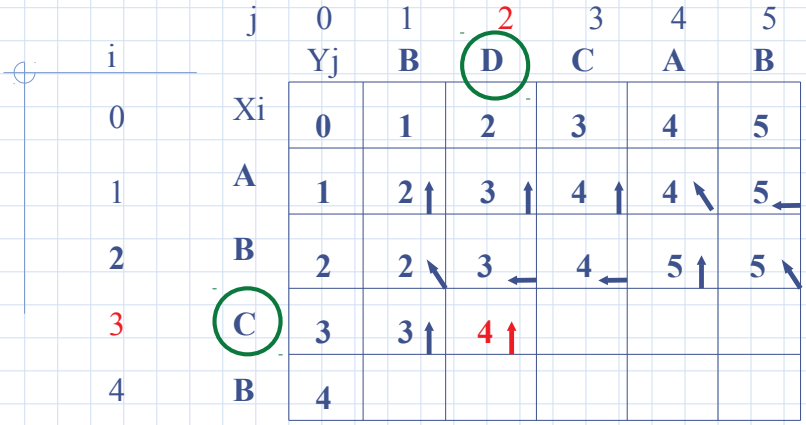
case i=2 and j=4
B != A
and c[1, 4]+1 = c[2, 3]+1
so c[2, 4] = c[1, 4]+1 and b[2, 2] =



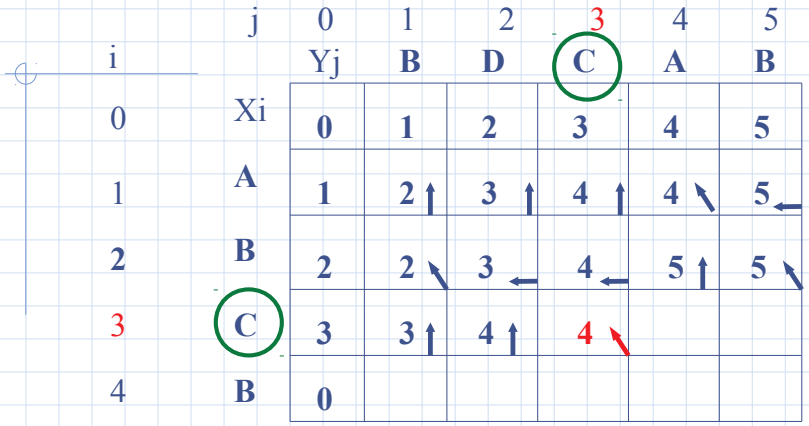
case i=3 and j=1
C != B
and c[2, 1] +1 < c[3,0]+1
so c[3, 1] = c[2, 1]+1 and b[3, 1] =



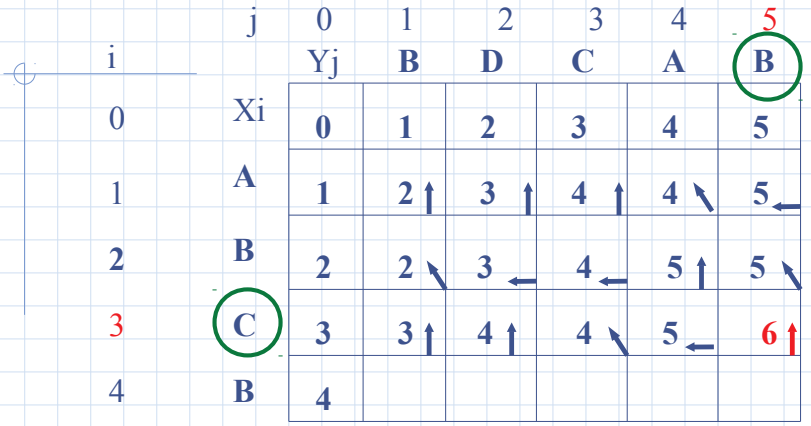
case i=2 and j=5
B = B
so c[2, 5] = c[1, 4]+1 and b[2, 5] =



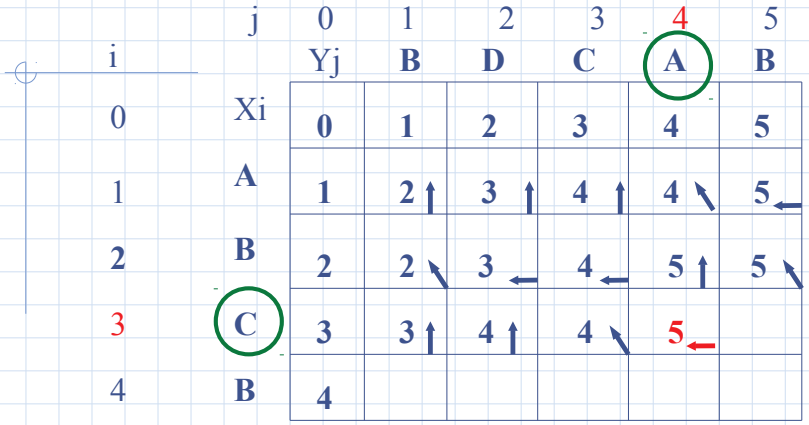
case i=3 and j= 2
C != D
and c[2, 2]+1 = c[3, 1]+1
so c[3, 2] = c[2, 2]+1 a b[3, 2] =



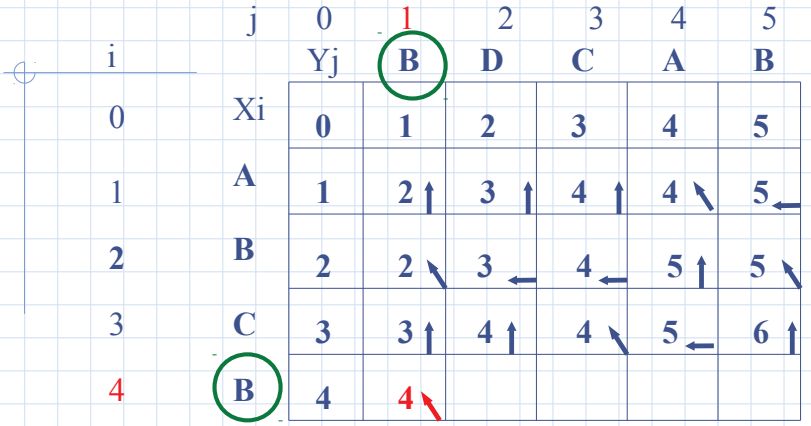
case i=3 and j= 3
C = C
so c[3, 3] = c[2, 2]+1 and b[3, 3] =



case i=3 and j= 5
C != B
c[2, 5]+1= c[3, 4]+1
tak c[3, 5] = c[2, 5]+1 a b[3, 5] =



case i=3 and j= 4
C != A
c[2, 4] +1 > c[3, 3]+1
so c[3, 4] = c[3, 3]+1 and b[3, 3] =



case i=4 and j=1
B = B
so c[4, 1] = c[3, 0]+1 and b[4, 1] =

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	1	2	3	4	5
1	A	1	2 ↑	3 ↑	4 ↑	4 ↖	5 ←
2	B	2	2 ↖	3 ←	4 ←	5 ↑	5 ↖
3	C	3	3 ↑	4 ↑	4 ↖	5 ←	6 ↑
4	B	4	4 ↖	5 ↑			

case i=4 and j=2
B != D
c[3, 2]+1 = c[4, 1]+1
so c[4, 2] = c[3, 2]+1 and b[4, 2] = ↑

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	1	2	3	4	5
1	A	1	2 ↑	3 ↑	4 ↑	4 ↖	5 ←
2	B	2	2 ↖	3 ←	4 ←	5 ↑	5 ↖
3	C	3	3 ↑	4 ↑	4 ↖	5 ←	6 ↑
4	B	4	4 ↖	5 ↑	5 ↑	6 ↑	

case i=4 and j=4
B != A
c[3, 4] +1= c[4, 3]+1
so c[4, 4] = c[3, 4] +1 and b[3, 5] = ↑

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	1	2	3	4	5
1	A	1	2 ↑	3 ↑	4 ↑	4 ↖	5 ←
2	B	2	2 ↖	3 ←	4 ←	5 ↑	5 ↖
3	C	3	3 ↑	4 ↑	4 ↖	5 ←	6 ↑
4	B	4	4 ↖	5 ↑	5 ↑		

case i=4 and j= 3
B != C
c[3, 3]+1 > c[4, 2]+1
so c[4, 3] = c[3, 3]+1 and b[4, 3] = ↑

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	1	2	3	4	5
1	A	1	2 ↑	3 ↑	4 ↑	4 ↖	5 ←
2	B	2	2 ↖	3 ←	4 ←	5 ↑	5 ↖
3	C	3	3 ↑	4 ↑	4 ↖	5 ←	6 ↑
4	B	4	4 ↖	5 ↑	5 ↑	6 ↑	6 ↖

case i=4 and j=5
B= B
so c[4, 5] = c[3, 4]+1 and b[4, 5] = ↖

Nalezení SCS

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	1	2	3	4	5
1	A		1	2	3	4	4	5
2	B		2	2	3	4	5	5
3	C		3	3	4	4	5	6
4	B		4	4	5	5	6	6

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3

SCS (obrácené pořadí): **B A C D B A**

SCS (správné pořadí): **A B D C A B**

Porovnávání řetězců (edit distance)

- přesné porovnávání dvou řetězců (vzájemná shoda) není použitelné v některých oblastech, které využívají symbolický popis (strukturní metody rozpoznávání)
- k testování podobnosti dvou řetězců

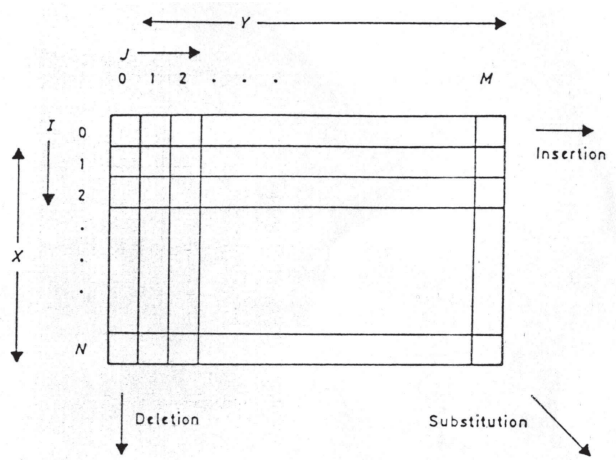
  $x=x_1x_2...x_n \in T^*$ a $y=y_1y_2...y_n \in T^*$

  (T je abeceda symbolů) je nutné definovat vhodnou metriku
- Hammingova metrika $d_H(x,y)$ – pouze pro řetězce stejné délky. Je definovaná jako počet odlišných symbolů ***x*** a ***y*** v odpovídajících si pozicích (např. řetězce abcab , bbdab mají $d_H=2$)
- Levensteinova metrika $d(x,y)$ (někdy označovaná jako edit distance), která je definovaná jako nejmenší počet transformací, které převedou řetězec **x** na řetězec **y**
- Transformace:
  - náhrada (substituce) symbolu **a** ∈ **T** v **x** symbolem **b** ∈ **T** v **y** **a≠b** (a→b)
  - vložení symbolu **a** ∈ **T** (ε→a) ε označuje prázdný symbol
  - zrušení symbolu **a** ∈ **T** (a→ ε)

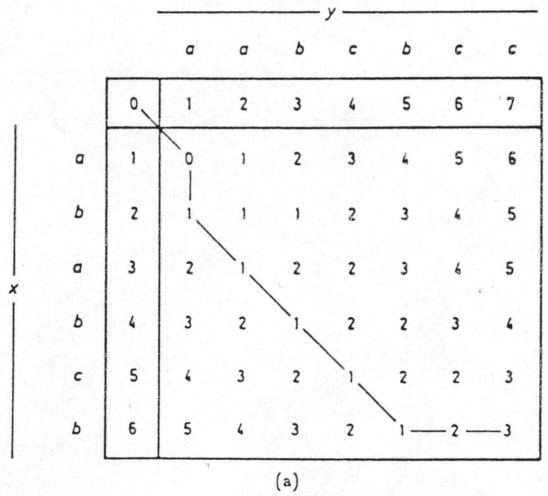
Algoritmus výpočtu vzdálenosti

```
input:
x = x_1 ... x_n ∈ T*, y = y_1 ... y_m ∈ T*, costs(a → b); a, b ∈ T ∪ {ε}
output:
d(x, y)
method:
0  begin
1  D(0,0) := 0;
2  for i = 1 to n do D(i,0) := D(i-1,0) + c(x(i) → ε);
3  for j = 1 to m do D(0,j) := D(0,j-1) + c(ε → y(j));
4  for i = 1 to n do
5      for j = 1 to m do
6          begin
7          m_1 := D(i-1,j-1) + c(x(i) → y(j));
8          m_2 := D(i-1,j) + c(x(i) → ε);
9          m_3 := D(i,j-1) + c(ε → y(j));
10         D(i,j) := min(m_1, m_2, m_3);
11         if m_1 = D(i,j) then set pointer from (i,j) to (i-1,j-1);
12         if m_2 = D(i,j) then set pointer from (i,j) to (i-1,j);
13         if m_3 = D(i,j) then set pointer from (i,j) to (i,j-1);
14         end;
15 d(x,y) := D(n,m);
16 end
```

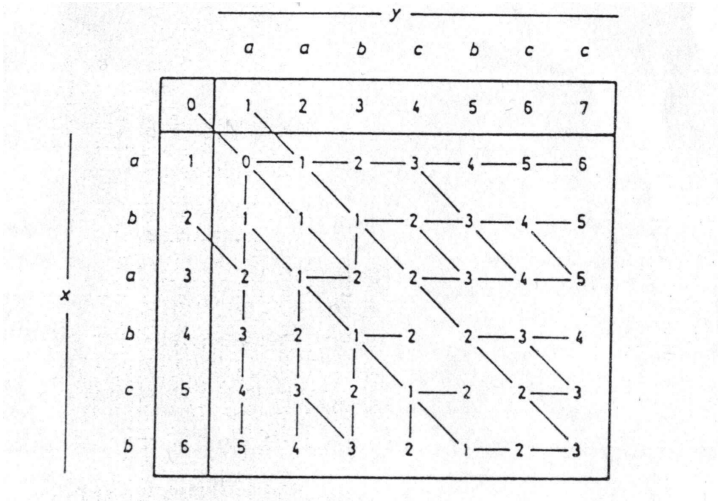
Matice pro výpočet vzdáleností

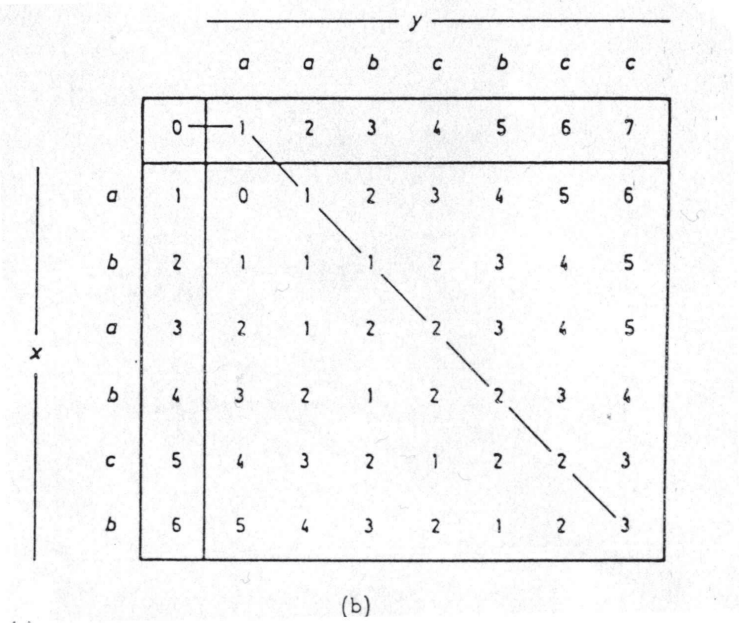


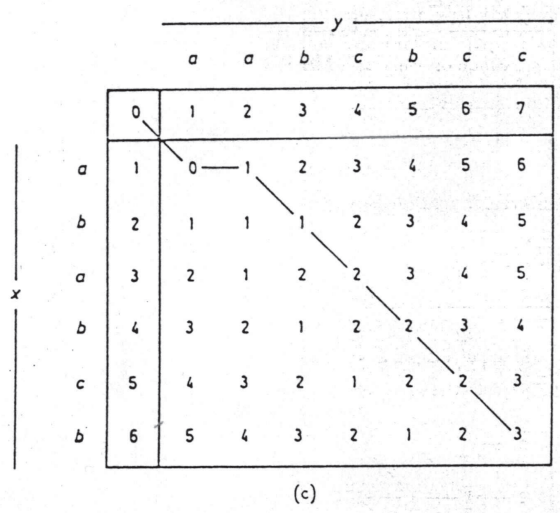
Rozdílné cesty, které vedou k úpravě řetězců



Příklad výpočtu vzdálenosti






y
a a b c b c c
0 1 2 3 4 5 6 7
x
a 1 0 1 2 3 4 5 6
b 2 1 1 1 2 3 4 5
a 3 2 1 2 2 3 4 5
b 4 3 2 1 2 2 3 4
c 5 4 3 2 1 2 2 3
b 6 5 4 3 2 1 2 3

(c)

Kompresní algoritmy

Komprese

Cíl komprese: redukovat objem dat za účelem

- přenosu dat
- archivace dat
- vytvoření distribuce sw
- ochrany před viry

Kvalita komprese:

- rychlost komprese
- symetrie/asymetrie kompresního algoritmu
  - Symetrické algoritmy – stejný čas potřebný pro kompresy i dekompresi
  - Asymetrické algoritmy –čas potřebný pro kompresi a dekompresi se liší
- kompresní poměr = poměr objemu komprimovaných dat k objemu dat nekomprimovaných

Komprese:

- bezztrátová - po kódování a dekódování je výsledek 100% shodný,
  - nižší kompresní poměr
  - požívají s výhradně pro kompresi textovů a v případech, kdy nelze připustit ztrátu informace
- ztrátová - po kódování a dekódování dochází ke ztrátě
  - obvykle vyšší kompresní poměr než bezztrátové
  - lze použít pouze v případech kdy ztráta je akceptovatelná (komprese obrazů, zvuku

Metody komprese:

- jednoduché – založené na kódování opakujících se posloupností znaků (RLE)
- statistické – založené na četnosti výskytu znaků v komprimovaném souboru (Huffmanovo kódování, Aritmetické kódování)
- slovníkové – založené na kodování všech vyskytujících se posloupností (LZW)
- transformační – založené na ortogonálních popř. jiných transformacích (JPEG, waveletová komprese, fraktálová komprese)

Jednoduché metody komprese

RLE (Run Length Encoding) – kódování délkou běhu

- **Charakteristika**: bezztrátová metoda
- **Použití:** zřídka pro kompresi textů, častěji pro obrazovou informaci
- **Princip:** opakující se symboly se kódují dvojicí
  (počet_opakování , symbol)

Kódování délky se provádí:

- přímo – u každého znaku je udán počet opakování

  Př. Vstup: AAAABBCDDDDABD
  Výstup: 4A2B1C4D1A1B1D

  Nevýhoda: pokud se znaky neopakují často nedochází ke kompresi, ale naopak k prodloužení kódovaného souboru

- pomocí *escape* sekvencí – kódují se pouze opakující se sekvence delší než 3 znaky, kratší sekvence se zapisují přímo do výstupního souboru

  Př. Vstup: AAAABBCDDDDABD
  Výstup: #4ABBC#4DABD

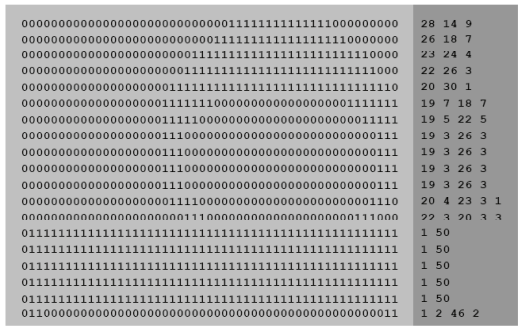
  Výhoda: neprodlužuje soubor, kde není co komprimovat to zůstane v původní podobě

Pozor !!! z množiny znaků je nutné vyčlenit symbol, který se nevyskytuje v komprimovaném souboru. Dále může nastat problém pokud je opakující se sekvence delší než 255 znaků (pokud kódujeme délku běhu na 8 bitech). Řešení závisí na konkrétní aplikaci

Použití RLE : např. obrazový formát BMP

Příklad použití RLE pro bitové obrazy

Natural encoding: 51 × 19 + 6 = 975 bits.
Run-length encoding: 63 × 6 + 6 = 384 bits.



raster of letter 'q' lying on its side RLE

Slovníková metoda komprese

LZW (Lempel-Ziv-Welch) metoda

Princip:

- vyhledávání opakujících se posloupností znaků, ukládání těchto posloupností do slovníku pro další použití a přiřazení jednoznakového kódu těmto posloupnostem.
- jednoprůchodová metoda (nevyžaduje předběžnou analýzu souboru)
- Kódované znaky musí mít délku (počet bitů) větší než délka původních znaků (např. pro ASCII znaky (8 bitů) se obvykle používá nová délka znaků 12 bitů popř. větší.)
- Při průchodu komprimovaným souborem se vytváří slovník (počet položek slovníku odpovídá hodnotě $2^{(\text{počet bitů nového kódu})}$, kde prvních $2^{(\text{počet bitů původního kódu})}$ položek jsou znaky původní abecedy a zbývající položky tvoří posloupnosti znaků obsažené v komprimovaném souboru.

Algoritmus komprese a vytvoření slovníku

```
S := přečti znak ze vstupu;

while (jsou další znaky na vstupu) do
begin
 C := přečti znak ze vstupu;
 if S+C je v kódovací tabulce then
   S := S+C
 else begin
     zapiš na výstup kód pro S
        přidej do kódovací tabulky (S+C)
        S := C
     end;
end;
zapiš na výstup kód pro S;
```

Výsledný výstupní řetězec:

65 66 67 256 258 257 68 259

Příklad: Komprese řetězce ABCABCABCDABC

Postup kódování

S (prefix)	C (suffix)	výstup (kód)
A	B	A(65)
B	C	B(66)
C	A	C(67)
A	B	—
AB	C	AB(256)
C	A	—
CA	B	CA(258)
B	C	—
BC	D	BC(257)
D	A	D(68)
A	B	—
AB	C	—
ABC	—	ABC(259)

kód	posloupnost
0..255	jednotlivé znaky
256	AB
257	BC
258	CA
259	ABC
260	CAB
261	BCD
262	DA

Algoritmus dekomprese a vytvoření slovníku

```
přečti OLD_CODE;
zapiš OLD_CODE na výstup;
while (na vstupu jsou další kódy) do
begin
  přečti NEW_CODE;
  if NEW_CODE není v kódovací tabulce then
    begin
      S := posloupnost zakódovaná kódem OLD_CODE;
      S := S+C;
    end
  else S := posloupnost zakódovaná kódem NEW_CODE;

  zapiš S na výstup;
  C := první znak S;
  přidej do kódovací tabulky (OLD_CODE+C);
  OLD_CODE := NEW_CODE;
end;
```

Vstupní řetězec:

65 66 67 256 258 257 68 259

OLD_CODE	NEW_CODE	S	C	Výstup
A(65)				A
A(65)	B(66)	B	B	B
B(66)	C(67)	C	C	C
C(67)	AB(256)	AB	A	AB
AB(256)	CA(258)	CA	C	CA
CA(258)	BC(257)	BC	B	BC
BC(257)	D(68)	D	D	D
D(68)	ABC(259)	ABC	A	ABC

Výstupní řetězec:

ABCABCABCDABC

Test existence NEW_CODE možná vypadá zbytečně, ale existují případy ve kterých to bez tohoto testu nefunguje, např. u řetězce ABABABAB !!!!

Použití : často používaná metoda u textových i grafických souborů (např. PKZIP, ARJ, ZIP,TIFF, GIF)

Statistické metody komprese

- Huffmanovo kódování
- Aritmetické kódování

Huffmanovo kódování

- algoritmus navržen v Davidem Huffmanem v roce 1952
- využívá optimálního (nejkratšího) prefixového kódu (kód žádného znaku není prefixem jiného znaku).
- kódové symboly mají proměnnou délku

Princip: Metoda je založená na stanovení četnosti výskytů jednotlivých znaků v kódovaném souboru a kódování znaků s největší četností slovem s nejkratší délkou.

Algoritmus kódování:

1. Zjištění četnosti jednotlivých znaků v kódovaném souboru
2. Vytvoření binárního stromu (Huffmanova kódu jednotlivých znaků)
   seřadíme posloupnost postupně zleva doprava doprava podle:
   - četnosti
   - podstrom bude vlevo před listem, větší podstrom před menším
   - pořadí v abecedě
3. Uložení stromu
4. Nahrazení symbolů jednotlivými kódy (posloupností bitů)

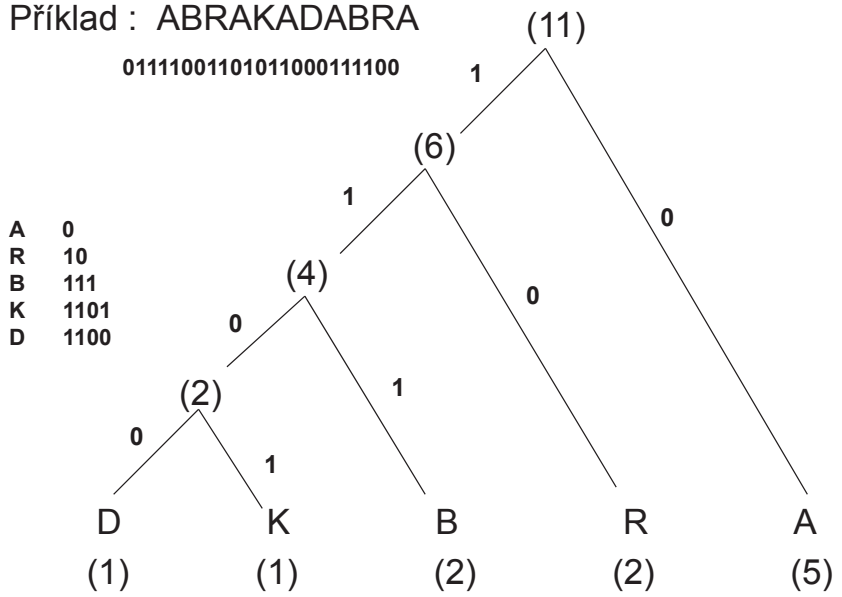
Algoritmus vytvoření stromu

jednotlivé znaky označíme za vrcholy grafu (listy stromu) a dáme je do seznamu S

```
while (S.length>1) {
 v S nalezneme dva vrcholy m, n s nejmenšími počty
   výskytů
 p = new Vrchol;
 p.left = m; p.right = n;
 p.count = m.count + n.count; // count je # výskytů
 S.remove(m, n); S.add(p);
 }
```

S obsahuje kořen stromu
najdeme kódy jednotlivých znaků (při průchodu z kořene do listu kódujeme 0 při kroku vlevo a 1 vpravo)

Příklad : ABRAKADABRA



- strom se ukládá na začátek kódované sekvence a přenáší se s komprimovaným souborem. Dekodér si ho nejprve vytvoří dekódovací strom a pak zpracovává vlastní kód.

Struktura stromu (ukládaná a spolu s kódem)

- Stromem se prochází do hloubky. Pro každý uzel se uloží bit 0, pro každý list se uloží bit 1 následovaný osmi bity s kódem listu. Uloženým stromem se při načítání postupuje takto: Jestliže narazíš na bit 0, vytvoř z aktuálního prvku uzel a postup do levého následovníka. Jestliže narazíš na bit 1, načti dalších osm (devět) bitů, ulož je do listu a postup na nejbližší volný prvek napravo. Načítání stromu skončí v momentě, kdy už není žádný volný prvek. Tímto způsobem se vygeneruje strom, kterým se při zpracování dat prochází.

- pro předchozí případ (řetězec ABRAKADABRA):

```
01A001R0001D01K01B
```

Dekomprese

1. Načtení a obnovení stromu, algoritmus je popsán při kompresi X
2. Vlastní dekomprese: Nahrazení kódů původními znaky.

```
v = vrchol stromu
while (!eof(input)) do {
  b = read bit
  if (b==0)
      v = v.left
  else v = v.right
  if (v je list) {
    write v.value // znak reprezentovany tímto listem
      v = vrchol stromu
    }
}
```

Aritmetické kódování

- Statistická metoda

- Kóduje celou zprávu jako jedno kódové slovo (v původní verzi číslo z intervalu [0,1).

Princip : Aritmetické kódování reprezentuje zprávu jako podinterval intervalu <0,1). Na začátku uvažujeme celý tento interval. Jak se zpráva prodlužuje, zpřesňuje se i výsledný interval a jeho horní a dolní mez se k sobě přibližují. Čím je kódovaný znak pravděpodobnější, tím se interval zúží méně a k zápisu delšího (to znamená hrubšího) intervalu stačí méně bitů. Na konec stačí zapsat libovolné číslo z výsledného intervalu.

Algoritmus komprese

1. Zjištění pravděpodobnosti P(i) výskytu jednotlivých znaků ve vstupním souboru
2. Stanovení příslušných kumulativních pravděpodobností K(0)=0, K(i)=K(i-1)+P(i-1) a rozdělení intervalu <0,1) na podintervaly I(i) odpovídající jednotlivým znakům (seřazeným podle abecedy)tak, aby délky těchto intervalů vyjadřovaly pravděpodobnosti příslušných znaků: I(i) = <K(i), K(i+1))
3. Uložení použitých pravděpodobností
4. Vlastní komprese

začínáme s intervalem I=<0,1): označme jeho dolní mez D(I), horni H(I) a délku intervalu L(I)=H(I)-D(I)

```
while (!eof) {
 read(i)
 I = <D(I)+K(i)*L(I), D(I)+K(i+1)*L(I))
}
write(D(I))
```

Příklad kódování

Vstup: **a** $L = 0$
$H = 0 + 0.5 \cdot 1 = 0.5$

b $L = 0 + 0.5(0.5 - 0) = 0.25$
$H = 0 + 0.5(0.5 - 0) + 0.25(0.5 - 0) = 0.375$

a $L = 0.25$
$H = 0.25+0.5 \cdot (0.375-0.25) = 0.3125$

a $L = 0.25$
$H = 0.25+0.5 \cdot (0.3125-0.25) = 0.28125$

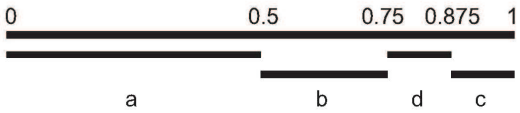
b $L = 0.25+0.5 \cdot (0.28125-0.25) = 0.265625$
$H = 0.25+0.5 \cdot (0.28125-0.25)+0.25 \cdot (0.28125-0.25) = 0.2734375$

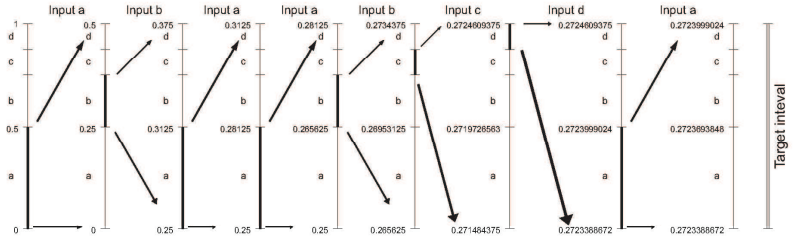
c $L = 0.265625+0.5 \cdot (0.273437c-0.265625)+0.25 \cdot (0.2734375-0.265625) = 0.271484375$
$H = 0.265625+0.5 \cdot (0.2734375-0.265625)+0.25 \cdot (0.2734375-0.265625)+0.125 \cdot 0.25 (0.2734375 0.265625)=0.2724609375$

Atd.

Příklad kódování

P(a)=0.5, P(b)=0.25, P(c)=0.125, P(d)=0.125





Dekomprese

1. Rekonstrukce použitých pravděpodobností
2. Vlastní dekomprese

```
read(X) přečteme uložené reálné číslo
while (není obnovena celá zpráva) {
  najdeme i, aby X bylo v [K(i), K(i+1))
  write(i)
  X=(X-K(i))/P(i)
 }
```

Ztrátové komprese

- JPEG
- Waveletová komprese
- Fraktálová komprese

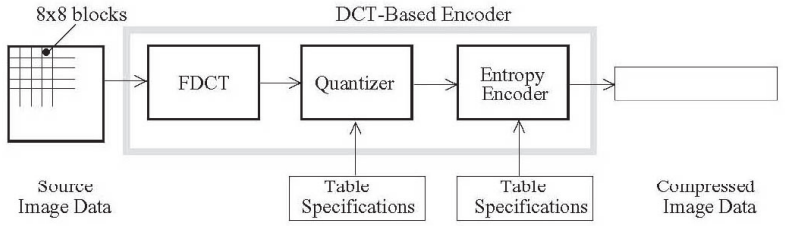
JPEG (Joint Photographic Experts Group)

- v současné době patří mezi nejvíce používané komprese u obrázků
- je vhodná pro komprimaci fotek, nevhodná pro např. technické výkresy (čarové výkresy) – dochází k viditelnému rozmazání

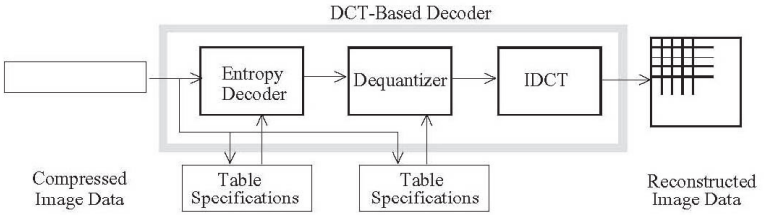
Princip:

- části obrazu se transformují do frekvenční oblasti (výsledkem je matice „frekvenčních“ koeficientů
- z matice koeficientů se odstraní koeficienty odpovídající vyšším frekvencím (rychlejší změny jasu – např. hrany v obraze)
- zbývající koeficienty se vhodným způsobem zkomprimují

Blokové schéma Jpeg komprese (kodér)



Blokové schéma Jpeg komprese (dekodér)



DCT – Diskrétní kosinová transformace

- transformuje kódovanou oblast do frekvenční oblasti
- je bezztrátová a existuje k ní inverzní transformace

Postup:

1. Zdrojový obraz se nejprve rozdělí na bloky 8x8 pixelů
2. Hodnoty jasu v každém bloku se nejprve transformují z intervalu $[0,2^{p}-1]$ na interval $[2^{p-1},2^{p-1}-1]$
3. Provede se diskrétní kosinová transformace podle vztahu:

$$F(u,v)=\frac{1}{4}C(u)\cdot C(v)\cdot\left[\sum_{x=0}^{7}\sum_{y=0}^{7}f(x,y)\cdot\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \qquad (DCT)$$

$$f(x,y)=\frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7}C(u)\cdot C(v)\cdot F(u,v)\cdot\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \qquad (IDCT)$$

$$C(u),C(v)=1/\sqrt{2},\ pro\ u,v=0$$

$$C(u),C(v)=1,\qquad pro\ u,v$$

Kvantizace / dekvantizace

- V tomto kroku se každý z 64 koeficientů DCT (IDCT] vydělí (vynásobí) odpovídajícím prvkem kvantizační matice a zaokrouhlí na nejbližší celé číslo. V tomto kroku dochází ke ztrátě informace !!!!!

kvantizace

$$F^{Q}(u,v)=Integer\left(\frac{F(u,v)}{Q(u,v)}\right)$$

dekvantizace

$$F^{Q'}(u,v)=F^{Q}(u,v)\cdot Q(u,v)$$

$Q_{50}=$

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

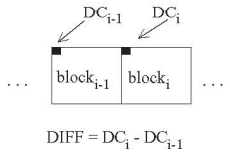
$$Q_q=\frac{Q_{50}(100-q)}{50}\qquad pro\ \ q\in(50,100)$$

$$Q_q=\frac{50\cdot Q_{50}}{q}\qquad pro\ \ q\in(0,50)$$

Kódování DCT koeficientů

- Koeficienty DCT se obvykle kódují pomocí statistických metod (Huffmann, aritmetické kódování)
- Koeficient v pozici (0,0) je označen jako DC koeficient (stejnosměrná složka), ostatní se označují jako AC koeficienty
- Vzhledem k tomu že DC koeficienty sousedních bloků jsou obvykle silně korelované (tj. střední hodnota jasů sousedních bloků je podobná) kódují se DC koeficienty odděleně od AC koeficientů
- kódování DC koeficientů – diference hodnot sousedních bloků (DC prvního bloku se kóduje jako přímá hodnota) - výsledná hodnota se kóduje jako dvojice

(velikost) (amplituda)

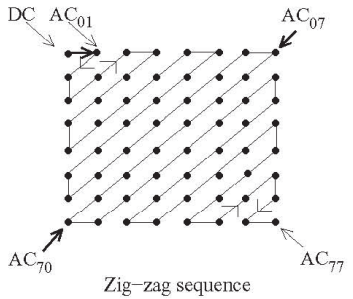


SIZE	AMPLITUDE
1	-1,1
2	-3,-2,2,3
3	-7..-4,4..7
4	-15..-8,8..15
5	-31..-16,16..31
6	-63..-32,32..63
7	-127..-64,64..127
8	-255..-128,128..255
9	-511..-256,256..511
10	-1023..-512,512..1023

- Kódování AC koeficientů – délkou běhu - nejprve se koeficienty uspořádají podle následujícího obrázku pak se kódují jako trojice

(RL,velikost) (amplituda)

kde RL je počet nul které jsou před kódovanou hodnotou, velikost a amplituda jsou shodné jako při kódování DC koeficientů



Příklad kódování bloku obrazu

139	144	149	153	155	155	155	155
144	151	153	156	159	156	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

(a) source image samples

235.6	-1.0	-12.1	-5.2	2.1	-1.7	-2.7	1.3
-22.6	-17.5	-6.2	-3.2	-2.9	-0.1	0.4	-1.2
-10.9	-9.3	-1.6	1.5	0.2	-0.9	-0.6	-0.1
-7.1	-1.9	0.2	1.5	0.9	-0.1	0.0	0.3
-0.6	-0.8	1.5	1.6	-0.1	-0.7	0.6	1.3
1.8	-0.2	1.6	-0.3	-0.8	1.5	1.0	-1.0
-1.3	-0.4	-0.3	-1.5	-0.5	1.7	1.1	-0.8
-2.6	1.6	-3.8	-1.8	1.9	1.2	-0.6	-0.4

(b) forward DCT coefficients

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

(c) quantization table

15	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(d) normalized quantized coefficients

240	0	-10	0	0	0	0	0
-24	-12	0	0	0	0	0	0
-14	-13	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(e) denormalized quantized coefficients

144	146	149	152	154	156	156	156
148	150	152	154	156	156	156	156
155	156	157	158	158	157	156	155
160	161	161	162	161	159	157	155
163	163	164	163	162	160	158	156
163	164	164	164	162	160	158	157
160	161	162	162	162	161	159	158
158	159	161	161	162	161	159	158

(f) reconstructed image samples

Příklad kódování bloku obrazu

15	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Předpoklad: předchozí blok měl kvantizovaného DC koeficientu +12

Poslopnost symbolů procelý obraz je pak možné kódovat Huffmanovým kódem

Posloupnost symbolů, které kódují blok je:

…….. **(2)(3), (1,2)(-2), (0,1)(-1), (0,1)(-1), (0,1)(-1), (0,0)** ……

Konec bloku

Waveletová komprese

Charakteristika:

- ztrátová komprese
- podobný princip jako u JPEG komprese
- využívá lineární transformaci (waveletovou transformaci)
- obvykle dosahuje vyšších kompresních poměrů

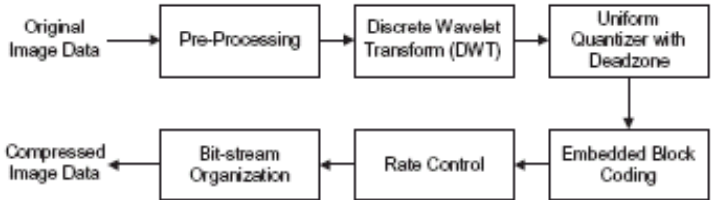
Použití:

- FBI – komprese otisků prstů
- JPEG 2000

JPEG 2000

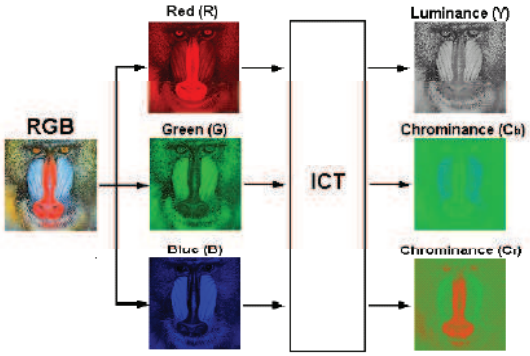
- cílem bylo navrhnout nový obrazový standard, který překonává některé nedostatky které byly u JPEG komprese
- vhodný pro rozdílné typy statických obrazů (binární, šedotónový, barevný) s rozdílnými charakteristikami (scenérie, technické výkresy, družicové snímky)
- vhodný pro rozdílné účely – přenos obrazů, archivace
- kódování JPEG 2000 může být ztrátové nebo bezztrátové

Blokové schéma kodovacího procesu



- barevná transformace z RGB do Y,C_r,C_b

$$\begin{bmatrix} Y \\ C_r \\ C_b \end{bmatrix} = \begin{bmatrix} 0.299 & 0.586 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$



Předzpracování

tři kroky předzpracování:

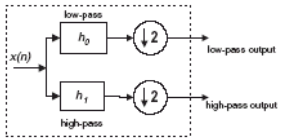
- **rozdělení obrazu na bloky** – bloky se nepřekrývají, jsou stejné velikosti, každý blok je komprimován samostatně s vlastními parametry komprese

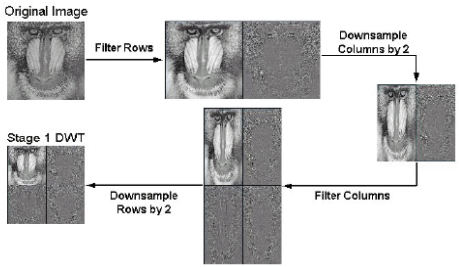
- **normalizace úrovní** – jelikož JPEG2000 používá filtr typu horní propust očekává se že rozsah vstupních hodnot je rozložen okolo nuly (je-li rozsah vstupu na B bitech bude vstup po normalizaci v rozsahu $-2^{B-1} \le x \le 2^{B-1}$

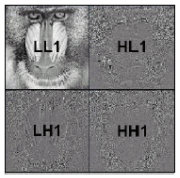
- **barevná transformace** – většinou jsou komprimovány barevné obrazy v RGB reprezentaci, ta je ale nevhodná pro ztrátovou kompresy (dochází k posuvu barev) používá se jiný barevný model (Y,C_r,C_b)

Diskrétní waveletová transformace

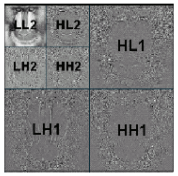
JPEG 2000 je založen na diskrétní waveletové dekompozici DWT



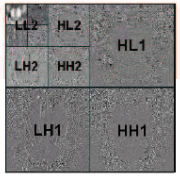




1. úroveň



2. úroveň

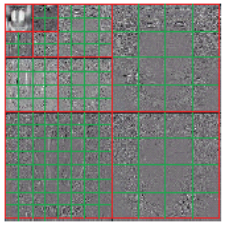


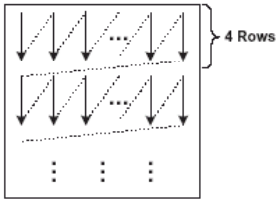
3. úroveň

- **počet úrovní dekompozice závisí na implementaci**

Systém kódování bloků

- každý dekompoziční blok je rozdělen do nepřekrývajících se menších bloků (64x64 nebo 32x32 koeficientů)

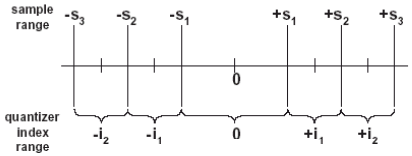




Kvantizace

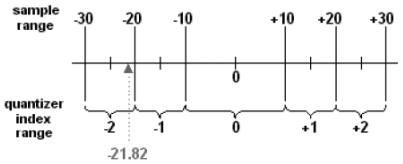
waveletové koeficienty z každé úrovně dekompozice jsou kvantizovány podle vztahu

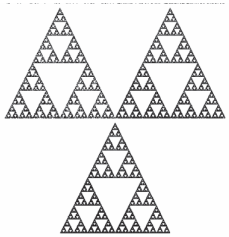
$$q = \mathrm{sign}(y)\left\lfloor \frac{|y|}{\Delta_b} \right\rfloor$$



výsledkem kvantizace je náhrada hodnoty každého koeficientu kvantizačním indexem

$$\text{Quantizer Index} = -\left\lfloor \frac{21.82}{10} \right\rfloor = -2$$





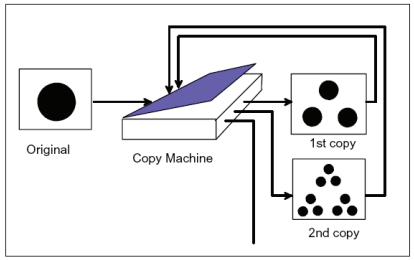
Fraktálová komprese obrazu

Úvod

- Termín fraktál poprvé použil Benoit Maldebrot (1975)
- Některé definice pojmu fraktál:
  - Fraktál je nerovný nebo fragmentovaný geometrický tvar, který může být rozdělen na části, které jsou (alespoň přibližně) menší kopií celku. Fraktály jsou obecně „sobě-podobné" (self-similar) to je malá část vypadá jako celý obraz
  - Fraktál je obraz nebo snímek, který může být kompletně popsán matematickým algoritmem (v libovolném rozlišení)
  - Fraktál je pevný bod (attractor) systému iterovaných funkcí (Iterated Function System)

Systém iterovaných funkcí

- soubor kontraktivních zobrazení
- nejlépe lze IFS vysvětlit pomocí kopírovacího stroje (Multiple Reduction Copying machine) s následujícími vlastnostmi:
  1. Kopírka obsahuje skupin čoček, nastavených tak, že mohou vytvářet překrývající se kopie originálu
  2. Každá čočka zmenšuje velikost originálu
  3. Kopírka pracuje iteračně ve zpětnovazebním režimu tj.,výstup je znovu přiveden na vstup



- Matematicky lze každou čočku popsat jako tzv. kontraktivní afinní zobrazení, které mění měřítko vstupu (zmenšuje), natáčí ho a kopíruje na výstup.

$$w_i = \begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

tj. každý bod vstupního obrazu (x,y) bude transformován do výstupního obrazuumístěn v nové pozici x', y', pro které platí

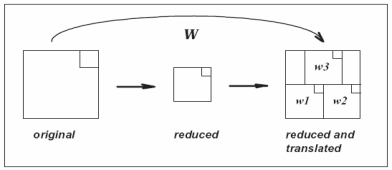
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

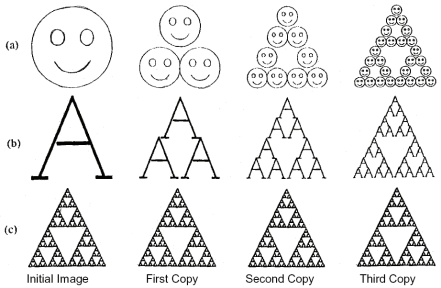
- Krontraktivní zobrazení : $x_1'=w(x_1)$, $x_2'=w(x_2)$

$$d(x_1,x_2)=s\cdot d(x_1',x_2') \qquad 0 < s < 1$$

Příklad IFS systému: (Sierpinského trojúhelník)

$$w_1 = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \end{bmatrix} \quad w_2 = \begin{bmatrix} 0.5 & 0 & 0.5 \\ 0 & 0.5 & 0 \end{bmatrix} \quad w_3 = \begin{bmatrix} 0.5 & 0 & 0.25 \\ 0 & 0.5 & 0.5 \end{bmatrix}$$





Zobecnění IFS systému pro šedotónové obrazy

- Pro šedotónové obrazy je IFS systém trojrozměrný a zobrazení má tvar

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = w_i \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \\ o_i \end{bmatrix}$$

kde s_i a o_i slouží k modifikaci jasu

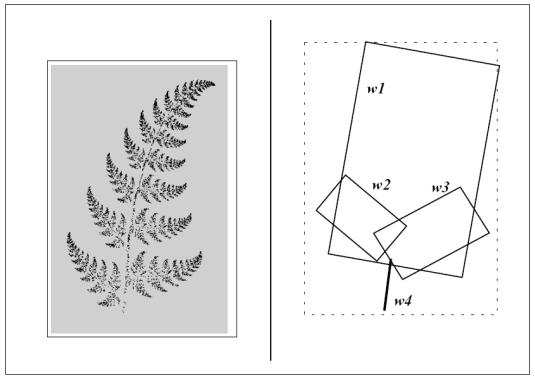
Další příklad IFS fraktálu: (Barnsleyova kapradina)

$$w_1 = \begin{bmatrix} 0.85 & 0.04 & 0.00 \\ -.04 & 0.85 & 1.6 \end{bmatrix}$$

$$w_2 = \begin{bmatrix} 0.20 & -.26 & 0.00 \\ 0.23 & 0.22 & 1.6 \end{bmatrix}$$

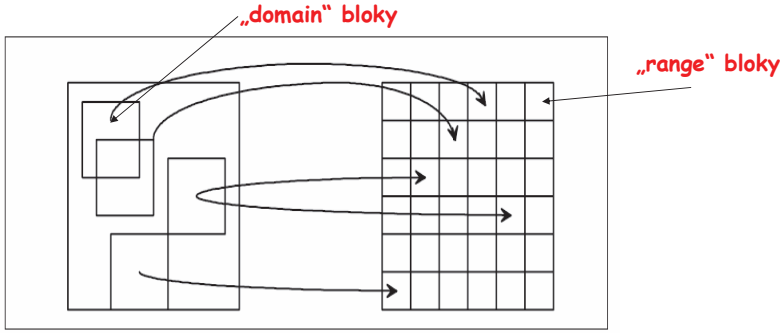
$$w_3 = \begin{bmatrix} -.15 & 0.28 & 0.00 \\ 0.26 & 0.52 & 0.44 \end{bmatrix}$$

$$w_4 = \begin{bmatrix} 0.00 & 0.00 & 0.00 \\ 0.00 & 0.16 & 0.00 \end{bmatrix}$$



Základní princip algoritmu fraktálové komprese

Základní princip spočívá v rozdělení komprimovaného obrazu na „range" bloky (nepřekrývají se) a vyhledávání „domain" bloků (mohou se překrývat) které jsou „range" blokům podobné



„domain" bloky se mohou vyskytovat buď v základním tvaru nebo v transformované podobě. Používají se následující transformace

1. Rotace o 0°
2. Rotace o 90°
3. Rotace o 180°
4. Rotace o 270°
5. Překlopení přes horizontální osu
6. Překlopení přes vertikální osu
7. Překlopení přes hlavní diagonálu
8. Překlopení přes vedlejší diagonálu

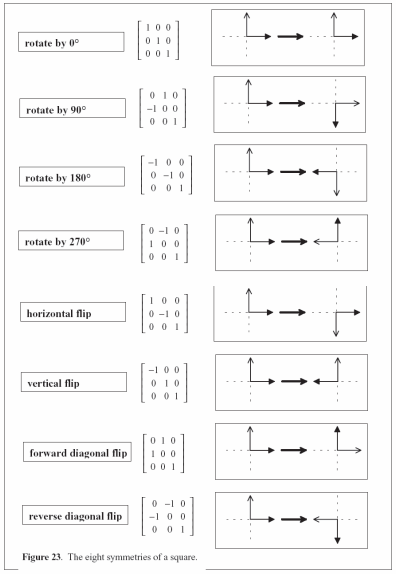


Figure 23. The eight symmetries of a square.

a) Pro každý doménový blok D_j a transformaci m_t (t=1,2,…,8) se vypočte $D_j^t=m_t(D_j)$ a na základě následujících rovnic se stanoví koeficienty ***s*** a ***o***

$$s=\frac{n\cdot(\sum_{i=1}^{n}d_i\cdot r_i)-(\sum_{i=1}^{n}r_i)\cdot(\sum_{i=1}^{n}d_i)}{n\cdot\sum_{i=1}^{n}d_i^2-(\sum_{i=1}^{n}d_i)^2}$$

$$o=\frac{1}{n}\left(\sum_{i=1}^{n}r_i-s\sum_{i=1}^{n}d_i\right)$$

b) Koeficienty ***s*** a ***o*** se kvantizují

c) Pro kvantizované koeficienty se podle následující rovnice vypočte chyba podobnosti bloků $E(D_j^t,R_i)$

$$E(D,R)^2=\frac{1}{n}\cdot\left[\sum_{i=1}^{n}r_i^2+s\cdot\left(s\cdot\sum_{i=1}^{n}d_i^2-2\cdot\sum_{i=1}^{n}d_i\cdot r_i+2\cdot o\cdot\sum_{i=1}^{n}d_i\right)+o\cdot\left(o\cdot n-2\cdot\sum_{i=1}^{n}d_i\right)\right]$$

Detailní algoritmus fraktálové komprese

1. **Segmentace obrazu** – komprimovaný obraz je rozdělen do bloků velikosti 8x8 (4x4) pixelů. Tyto bloky pokrývají celý obraz a nepřekrývají se. Tyto bloky se nazývají „range" bloky R_i

2. **Vytvoření souboru doménových bloků (domain pool)** – procházíme obraz zleva do prava shora dolů s krokem *k* pixelů a vytvoříme seznam tzv. doménových bloků, které mají dvojnásobnou velikost „range" bloků. V každém doménovém bloku jsou průměrovány sousední pixely a jsou uloženy do nového doménového bloku stejné velikosti jako „range" blok. Novým doménovým blokem přepíšeme blok původní.

For i=1 to N_R opakuj kroky 3 a 4 (N_R je počet „range" bloků)

3. **Vyhledávání** - pro každý „range" blok R_i nalezneme v souboru doménových bloků blok D^B, který se mu nejvíce podobá.

d) Nalezneme blok D_j^t s minimální chybou $E(D_j^t,R_i)$ t=1,2,…,8

e) V souboru doménových bloků nalezneme nejpodobnější blok tj.

$$D^B=\min_{t=1,2,\ldots,N_D}(D'_j)$$

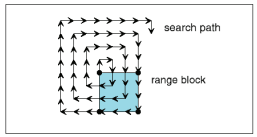
N_D je počet doménových bloků

4. **Výstupem je kód $w_i=(e_i,f_i,m_i,o_i,s_i)$**

5. **Výstupní posloupnost transformací je možné kódovat metodou bezztrátové komprese**

Strategie vyhledávání (vytvoření souboru doménových bloků)

1. **Metoda „hrubé" síly (heavy brute force)** – velikost kroku k=1. Časová složitost je $O(n^2)$, pro šedotónový obrazek vyžaduje algoritmus 2^{37}=128 Gflops
2. **„Light" Brute Force** – velikost kroku k>1. Kvalita výsledného obrazu může být v tomto případě horší protože přeskakujeme některé části obrazů, které by mohli být podobné s range blokem
3. **Omezená oblast vyhledávání** - oblast vyhledávání doménových bloků se redukuje pouze na okolí (např. kvadrant) testovaného „range" bloku.
4. **Lokální spirálové vyhledávání** – doménové bloky jsou vyhledávány na spirále začínající v pozici „range" bloku . Vyhledávání končí jakmile se nalezne vhodný doménový blok



5. **Hledání ve stejném místě jako je odpovídající range blok.** Jedná se o rychlé vyhledávání (složitost O(n)) s nízkou kvalitou
6. **Kategorizované vyhledávání** – každý doménový blok je zařazen do jedné ze 72 kategorií (3 třídy, 24 kategorií v každé třídě. Postup klasifikace bloků je následující: nejprve je blok rozdělen do 4 kvadrantů a pro každý kvadrant se vypočítá průměrná hodnota pixelů podle následujících rovnic:

$$A_i = \frac{1}{n}\sum_{j=1}^{n} r_j^i$$

$$V_i = \sum_{j=1}^{n}((r_j^i)^2 - A_i^2)$$

Poté je blok natočen do kanonické pozice – tj. pozice ve které je „světlost" kvadrantů shodná z některým z následujících vzorů (světlost=průměrná hodnota jasu pixelů)

class 1 class 2 class 3

Medians and Order Statistics

Nechť A je množina obsahující n různých prvků:

Definice: Statistika i-tého řádu je i-tý nejmenší prvek, tj.,
- minimum = statistika 1. řádu
- maximum = statistika n-tého řádu
- median(s) = $\lfloor (n+1)/2 \rfloor$ a $\lceil (n+1)/2 \rceil$ - pro sudý počet prvků existují dva mediány nejčastěji se uvažuje první z uvedených případů

Algoritmus výběru : Pro dané i, určit statistiku i-tého řádu
vstup: Množina A n (***různých***) čísel a hodnota i, $1 \le i \le n$
výstup: Prvek $x \in A$ který je větší než $(i - 1)$ prvků množiny A

O(nlgn) řešení algoritmu výběru

Algoritmus výběru : Pro dané i, určit statistiku i-tého řádu
vstup: Množina A n (***různých***) čísel a hodnota i, $1 \le i \le n$
výstup: Prvek $x \in A$ který je větší než $(i - 1)$ prvků množiny A

```
NaiveSelection(A, i)
1. A' ← FavoriteSort(A)

2. return A'[i]
```

Čas výpočtu:
*O(n*lg*n)* pro algoritmy řazení, které jsou založené na porovnávání
Může být lepší???

Základní myšlenka: Použít O(nlg(n)) algoritmus pro řazení čísle, (heapsort, mergesort), pak vybrat i-tý prvek pole.

Nalezení Minima (Maxima)

Čas výpočtu:
- jediný průchod polem
- pouze n-1 porovnání

```
Minimum(A)
1. lowest ← A[1]
2. for i ← 2 to n do
3.     lowest ← min(lowest, A[i])
```

Je tohle nejlepší možný čas potřebný pro nalezení minima (maxima)? Ano!
Proč je nutné n – 1 porovnání ?
- Algoritmus vyhledávající minimum musí porovnat každý prvek s „vítězem“ (nalezení minima/maxima lze chápat jako turnaj, ve kterém musí být n-1 pořážek ke stanovení vítěze)

Nalezení Minima & Maxima

Co když chceme současně vyhledat minimum a maximum ?

Kolik porovnání je nutné provést ?

- Postup A: nalezneme minimum a maximum odděleně tj. n - 1 porovnání pro min a pro max, tj. celkem 2n -2 porovnání
  Lze to udělat lépe ?

- Postup B: Zpracováváme prvky po dvojicích. Nejprve se porovná vzájemně porovná dvojice prvků vstupního pole a menší hodnota se porovná s dosavadním minimem, větší hodnota pak s maximema podle výsledků porovnání se uraví aktuální hodnota minima a maxima.
  Cena = 3 porovnání pro každé 2 prvky.
  Celková cena = $3\lfloor n/2 \rfloor$.

Finding Minimum & Maximum

Postup B: Zpracováváme prvky po dvojicích. Nejprve se porovná vzájemně porovná dvojice prvků vstupního pole a menší hodnota se porovná s dosavadním minimem, větší hodnota pak s maximem a podle výsledků porovnání se upraví aktuální hodnota minima a maxima.

Cena = 3 porovnání pro každé 2 prvky.

Celková cena = $3\lfloor n/2 \rfloor$.

```
FindMin&Max(A)
1.if length[A] % 2 == 0
2.    then if A[1] > A[2]
3.         then min = A[2]
4.              max = A[1]
5.         else min = A[1]
6.              max = A[2]
```

```
7.        else // n % 2 == 1
8.             then min=max=A[1]
9.    Compare the rest of the elements in
      pairs, comparing only the
      maximum element of each pair
      with max and the minimum
      element of each pair with min
```

Analýza FindMin&Max

- Je-li n sudé, potřebujeme 1 počáteční porovnání a pak 3(n-2)/2 + 1 porovnání = 3n/2 – 2
- Je-li n liché, potřebujeme 3(n-1)/2 porovnání
- V obou případech, je maximální počet porovnání ≤ $3\lfloor n/2 \rfloor$

```
FindMin&Max(A)
1.if length[A] % 2 == 0
2.    then if A[1] > A[2]
3.         then min = A[2]
4.              max = A[1]
5.         else min = A[1]
6.              max = A[2]
```

```
7.        else // n % 2 == 1
8.             then min=max=A[1]
9.    Compare the rest of the elements in
      pairs, comparing only the
      maximum element of each pair
      with max and the minimum
      element of each pair with min
```

Výběr statistiky i-tého řádu v lineárním čase

- Randomized-Select vrací i-tý nejmenší prvek A[p..r].

```
Randomized-Select(A, p, r, i)
1. if p = r then return A[p]
2. q ← Randomized-Partition(A, p, r)
3. k ← q – p + 1
4. if i = k then return A[q]
5. else if  i < k then return Randomized-Select(A, p, q-1, i)
6. else return Randomized-Select(A, q+1, r, i - k)
```

```
Randomized-Partition(A, p, r)
1. j ← Random(p, r)
2. swap A[r] ↔ A[j]
3. return Partition(A, p, r)
```

Výběr statistiky i-tého řádu v lineárním čase

- Algoritmus Randomized-Partition nejprve zamění A[r] s náhodně zvoleným prvkem A a pak zavolá proceduru Partition použitou v algoritmu QuickSort.

```
Randomized-Partition(A, p, r)
1. j ← Random(p, r)
2. swap A[r] ↔ A[j]
3. return Partition(A, p, r)
```

```
Partition(A, p, r)
1.x ← A[r]
2.i ← p – 1
3. for j ← p to r - 1 do
4.if A[j] ≤ x then
5.                i ← i + 1
6.                  swap A[i] ↔ A[j]
7. swap A[i+1] ↔  A[r]
8. return i + 1
```

Doba výpočtu algoritmu Randomized-Select

- Nejhorší případ : při nešťastném výběru vznikne n - 1 částí.
  $T(n) = T(n - 1) + \theta(n) = \theta(n^2)$
  (stejné jako nejhorší případ u algoritmu QuickSort)
- Nejlepší případ : při vhodné volbě se rychle redukuje velikost částí $T(n) = T(n/2) + \theta(n)$
- Průměrný případ : jako Quick-Sort, asymptoticky se bude blížit nejlepšímu případu

Určení statistiky i-tého řádu v lineárním čase (v nejhorším případě)

Modifikovaná verze algoritmu Partition x je vstupní parametr , který obsahuje hodnotu prvku, okolo kterého se vytvářejí části.

```
Partition(A, p, r, x)
1. i ← p - 1
2. for j ← p to r - 1 do
3.     if A[j] ≤ x then
4.         i ← i + 1
5.         swap A[i] ↔ A[j]
6. swap A[i+1] ↔ A[r]
7. return i + 1
```

Určení statistiky i-tého řádu v lineárním čase (v nejhorším případě)

Key: Zaručíme-li a „dobré" rozdělení když separujeme pole na dílčí části pak bude algoritmus běžet v lineárním čase.

```
Select(A, p, r, i)      /* i je i-tý hledaný prvek v pořadí. */
1. Rozdělit vstupní pole A ⌊n/5⌋ skupin velikosti 5 prvků ve
   skupině (a jedna zbývající skupina, pokud n % 5 != 0)
2. Určit medián každé skupiny o velikosti 5 metodou insert-
   sort pro 5 zvolit prostřední prvek.
3. volat Select rekurzívně, abychom určili hodnotu x, což je
   medián z ⌈n/5⌉ mediánů.
4. Rozdělit celé pole okolo prvku x, do dvou polí
   A[p, q-1] and A[q+1, r] a vrátit q, index bodu rozdělení
   (použít modifikovaný Partition Algoritmus viz dále).
5. Nechť k = q – p + 1
6. if (i = k) return x  (předpokládá, že index x je k)
   else if (i < k) then
        call Select(A, p, q-1, i)
   else call Select(A, q+1, r, i – k)
```

Doba výpočtu procedury of Select

Doba běhu (jednotlivé kroky):

1. O(n) (rozdělení do skupin po 5 prvcích)
2. O(n) (seřazení 5 prvků a nalezení mediánu je O(1) časová složitost)
3. $T(\lceil n/5 \rceil)$ (recurzívní volání k nalezení mediánu mediánů)
4. O(n) (rozdělení pole)
5. $T(7n/10 + 6)$ (maximální velikost podproblému)

Celková doba zpracování je dána rekurentně

$$T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$$

Doba výpočtu procedury of Select

Řešení rekurentní rovnice metodou odhadu. Odhadujeme

$T(n) \leq cn$

$$
\begin{aligned}
T(n) &= T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) \\
&\leq c\lceil n/5 \rceil + c(7n/10 + 6) + O(n) \\
&\leq c((n/5) + 1) + 7cn/10 + 6c + O(n) \\
&= cn - (cn/10 - 7c) + O(n) \\
&\leq cn
\end{aligned}
$$

Když $n >= 80$ $(cn/10 - 7c)$ nabývá kladných hodnot

Zvolíme-li dostatečně velké c tak $O(n) + (cn/10 - 7c)$ je kladné a platí předchozí řádek. (Např. $c = 200$)

Metody řazení s lineární časovou složitostí

Counting sort
Radix sort
Bucket sort

Úvod

- Porovnávací řazení: pořadí prvků v řazené posloupnosti je určováno pouze na základě porovnávání vstupních prvků
  - Libovolné porovnání must musí mít v nejhorším případě složitost $\Omega(n \lg n)$
  - Merge sort a heapsort jsou asymptoticky optimální
- Řazení s lineární časovou složitostí
  - K uspořádání prvků používá jiných operací než je porovnávání
  - Counting sort, radix sort, a bucket sort

Dolní hranice pro Comparison Sort

Decision-Tree Model

- Decision tree (Rozhodovací strom)
  - Úplný binární rozhodovací strom, který reprezentuje porovnání mezi prvky vstupního pole dané velikosti.
    - i:j → porovnáme a[i] a a[j]
    - Každý list je označen permutací $<\pi(1), \pi(2), \ldots, \pi(n)>$
      - n! je celkový počet permutací
  - Vykonávání kroků řadícího algoritmu odpovídá průchodu stromem od kořene k listu
  - Nutnou podmínkou toho, aby porovnávací řazení správně fungovalo je, že každá z n! permutací provedenou na vstupních prvcích se musí vyskytnout v jednom z listů rozhodovacího stromu a každý z těchto listů musí být dosažitelný z kořene.

The Decision-Tree Model

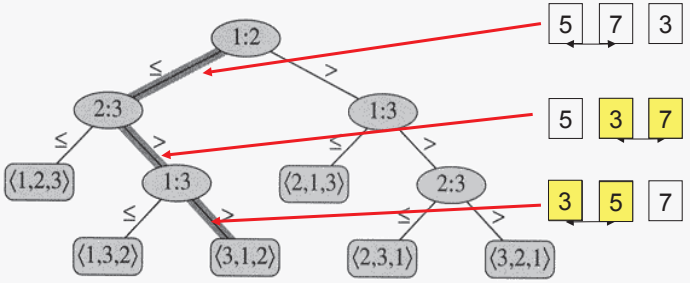


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i{:}j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, so the decision tree must have at least 6 leaves.

Dolní hranice pro nejhorší případ

- Délka nejdelší cesty z kořene do jeho dosažitelný listů reprezentuje nejhorší případ počtu porovnávání, které odpovídá jednotlivým krokům algoritmu
  - Výška rozhodovacího stromu
- ***Věta 8.1.*** Libovolný algoritmus porovnávacího řazení vyžaduje v nejhorším případě $\Omega(n \lg n)$.
- Uvažujme rozhodovací strom výšky h s l dosažitelnými listy
  - $n! \le l \le 2^h$
    - $h \ge \lg(n!)$ (protože lg funkce je monotónně rostoucí) $= \Omega(n \lg n)$

Counting Sort

Counting Sort Overview

- Předpoklad: n vstupních celočíselných prvků z rozsahu 0 to k (integer).
  - $\Theta(n+k)$ ➔ Když k=O(n), counting sort: $\Theta(n)$
- Základní myšlenka
  - Pro každý vstupní prvek x, určíme počet prvků menších nebo rovných x
  - Pro každé integer číslo i ($0 \le i \le k$), určíme kolik prvků má hodnotu i
    - Pak také víme, kolik prvků je menších nebo rovných i
- Algoritmus využívá následující proměnné
  - A[1..n]: vstupní prvky
  - B[1..n]: seřazené pole (výstup)
  - C[0..k]: pole ve kterém se uchovává počet prvků menších nebo rovných i

Counting Sort příklad

(Rozsah prvků od 0 do 5)

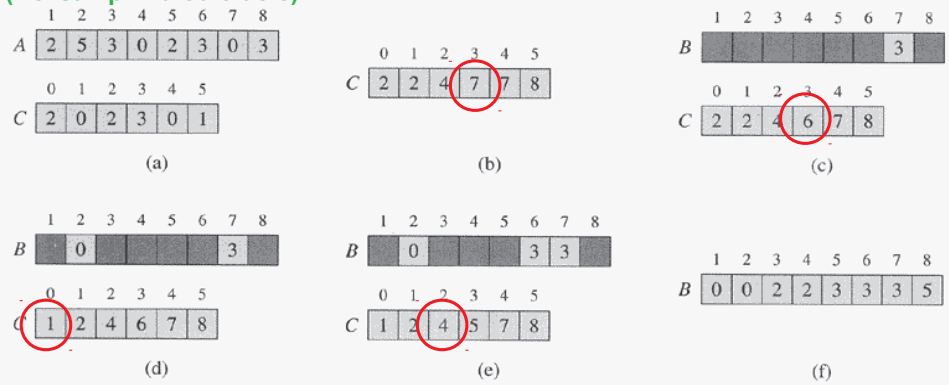
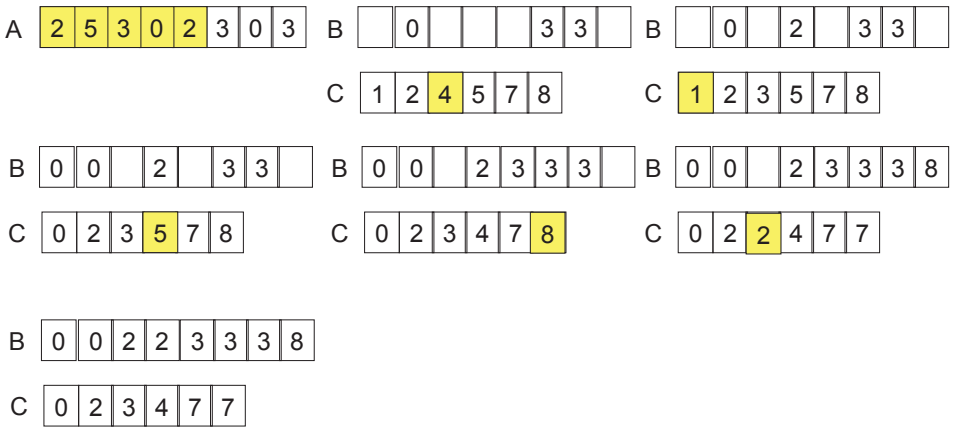


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Counting Sort Algoritmus Θ(n+k)

```
COUNTING-SORT(A, B, k)
1  for i ← 0 to k                                   Θ(k)
2      do C[i] ← 0
3  for j ← 1 to length[A]                           Θ(n)
4      do C[A[j]] ← C[A[j]] + 1
5  ▷ C[i] now contains the number of elements equal to i.
6  for i ← 1 to k                                   Θ(k)
7      do C[i] ← C[i] + C[i − 1]
8  ▷ C[i] now contains the number of elements less than or equal to i.
9  for j ← length[A] downto 1                       Θ(n)
10     do B[C[A[j]]] ← A[j]
11        C[A[j]] ← C[A[j]] − 1
```


Counting Sort příklad (pokračování)



Counting Sort je stabilní

- Algoritmus řazení je stabilní pokud
  - Čísla se stejnou hodnotou se ve výstupním (seřazeném) poli vyskytne ve stejné pozici, jako ve vstupním poli
- Řádek 9 algoritmu counting sort: for j←length[A] down to 1 je základem toho, aby algoritmus byl stabilní

Radix Sort

Radix Sort Příklad

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

1. Nejprve se začíná řadit podle nejméně významné číslice
2. To je základem toho, že algoritmus řazení čísel je stabilní

Radix Sort Algorithm

```
RADIX-SORT(A, d)
1  for i ← 1 to d
2      do use a stable sort to sort array A on digit i
```

d je celkový počet čísel

Lemma 8.3 Pro n d-místných čísel, u kterých každá číslice nabývá jedné z k možných hodnot, provádí RADIX-SORT řazení těchto čísel v čase Θ(d(n+k))

Bucket Sort

Bucket Sort

- Bucket sort běží v lineárním čase, pokud mají vstupní prvky rovnoměrné rozložení v interval [0, 1)
- Základní myšlenka
  - Rozdělíme interval [0,1) na n rovnoměrně rozložených subintervals (buckets)
  - Rozdělíme n čísle do těchto subintervalů
  - Seřadíme čísla v každém bucketu
  - Procházíme buckety v daném pořadí, vypisujeme prvky vkaždém bucketu

Bucket Sort příklad (pokračování)

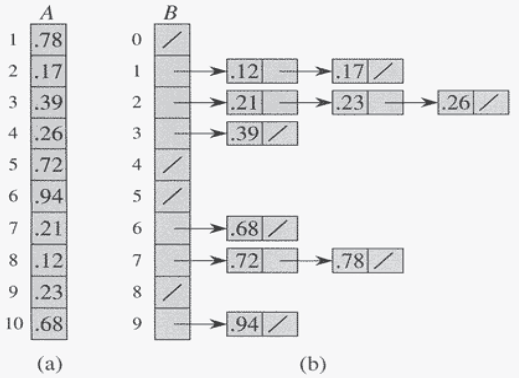
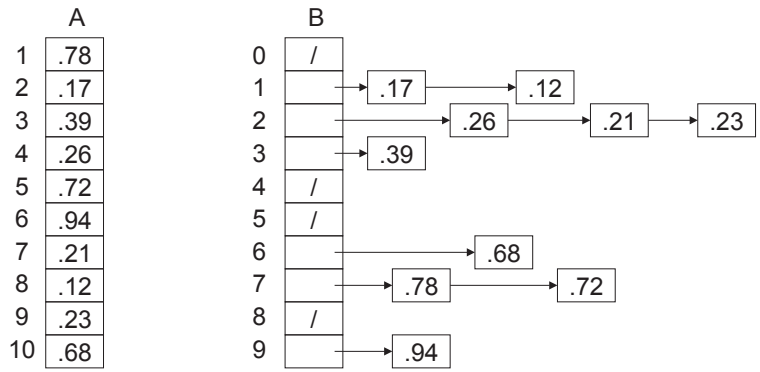


Figure 8.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket Sort příklad



Bucket Sort Algoritmus

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3       do insert A[i] into list B[⌊nA[i]⌋]
4  for i ← 0 to n − 1
5       do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Kombinatorické algoritmy

- Generování n-tic
- Generování permutací
- Generování kombinací

Použití: řešení některých úloh vyžaduje prozkoumat „prostor“ všech možných řešení a ten bývá často vyjádřen kombinatoricky (nalezení všech podmnožin, permutací, n-prvkových kombinací apod).

Generování n-tic

Použití: generování podmnožin dané množiny, generování, n-řádových čísel apod.

Postup:

Pro množiny $S_b=\{0,1\}$ nebo $S_d=\{0,1,2,3,4,5,6,7,8,9\}$ je problém jednoduchý.

1. Začneme s n-ticí $a= (000..000)_2$ nebo $a=(000..000)_{10}$
2. Postupně přičítáme 1 k binárnímu číslu a (pro S_b) nebo k dekadickému číslu (pro S_d)
3. Ukončíme pokud $a=(111…111)=2^{n-1}$ (pro S_b) nebo $a=(999…999)= 2^{10-1}$ (pro S_d)

Algoritmus pro generování obecných n-tic v lexikografickém pořadí, které pro které platí následující podmínky:

$$0 \le a_j < m_j \qquad pro \quad 1 \le j \le n$$

$$\begin{bmatrix} a_1 & a_2 & \cdots & a_n \\ m_1 & m_2 & \cdots & m_n \end{bmatrix}$$

Algorithm M (*Mixed-radix generation*). This algorithm visits all n-tuples that satisfy (1), by repeatedly adding 1 to the mixed-radix number in (2) until overflow occurs. Auxiliary variables a_0 and m_0 are introduced for convenience.

M1. [Initialize.] Set $a_j \leftarrow 0$ for $0 \le j \le n$, and set $m_0 \leftarrow 2$.

M2. [Visit.] Visit the n-tuple $(a_1, \ldots, a_n)$. (The program that wants to examine all n-tuples now does its thing.)

M3. [Prepare to add one.] Set $j \leftarrow n$.

M4. [Carry if necessary.] If $a_j = m_j - 1$, set $a_j \leftarrow 0$, $j \leftarrow j - 1$, and repeat this step.

M5. [Increase, unless done.] If $j = 0$, terminate the algorithm. Otherwise set $a_j \leftarrow a_j + 1$ and go back to step M2. ∎

Pokud je hodnota *n* dostatečně malá, lze algoritmus přepsat následovně: (n=4)

For $a_1 = 0, 1, \ldots, m_1 - 1$ (in this order) do the following:
  For $a_2 = 0, 1, \ldots, m_2 - 1$ (in this order) do the following:
    For $a_3 = 0, 1, \ldots, m_3 - 1$ (in this order) do the following:
      For $a_4 = 0, 1, \ldots, m_4 - 1$ (in this order) do the following:
        Visit (a_1, a_2, a_3, a_4).

Grayův binární kód

- V některých případech nevyhovuje lexikografické pořadí n-tic je potřeba volit jiné.
- Grayův kód vypisuje všech 2^n binárních n-tic v takovém pořadí, že mezi jednotlivými n-ticemi dochází k záměně pouze jediného bitu.

  Př. Grayův kód pro n=4

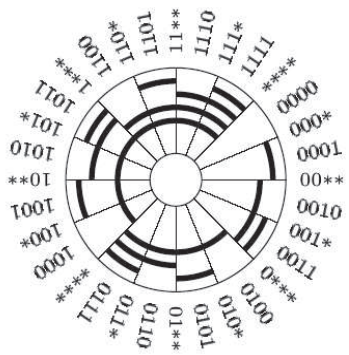
$$0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100,$$
$$1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000.$$

Použití Grayova kódu :

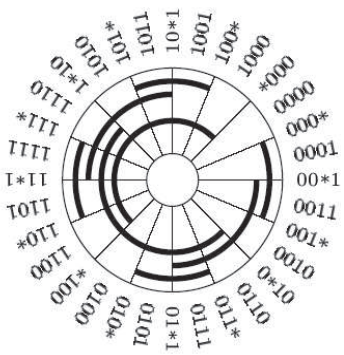
- přenos dat (umožňuje snadné zabezpečení paritou)
- Walshovy funkce, Walshova transformace (používá se při zpracování obrazů)

Snímání pozice natočení kotouče

a. Popis pozice lexikografickým binárním kódem
b. Grayovým kódem



(a) Lexicographic binary code.



(b) Gray binary code.

Fig. 10.

Algoritmus generování n-bitového Grayova kódu s paritou

Algorithm G (*Gray binary generation*). This algorithm visits all binary n-tuples $(a_{n-1}, \ldots, a_1, a_0)$ by starting with $(0, \ldots, 0, 0)$ and changing only one bit at a time, also maintaining a parity bit a_∞ such that

$$a_\infty = a_{n-1} \oplus \cdots \oplus a_1 \oplus a_0. \tag{14}$$

It successively complements bits $\rho(1)$, $\rho(2)$, $\rho(3)$, $\ldots$, $\rho(2^n - 1)$ and then stops.

G1. [Initialize.] Set $a_j \leftarrow 0$ for $0 \le j < n$; also set $a_\infty \leftarrow 0$.

G2. [Visit.] Visit the n-tuple $(a_{n-1}, \ldots, a_1, a_0)$.

G3. [Change parity.] Set $a_\infty \leftarrow 1 - a_\infty$.

G4. [Choose j.] If $a_\infty = 1$, set $j \leftarrow 0$. Otherwise let $j \ge 1$ be minimum such that $a_{j-1} = 1$. (After the kth time we have performed this step, $j = \rho(k)$.)

G5. [Complement coordinate j.] Terminate if $j = n$; otherwise set $a_j \leftarrow 1 - a_j$ and return to G2. ∎

Zrychlená varianta předchozího algoritmu – odstraňuje cyklus z kroku G4 a nahrazuje ho skupinou pointerů (f_n,…,f_0)

Algorithm L (*Loopless Gray binary generation*). This algorithm, like Algorithm G, visits all binary n-tuples $(a_{n-1}, \ldots, a_0)$ in the order of the Gray binary code. But instead of maintaining a parity bit, it uses an array of "focus pointers" $(f_n, \ldots, f_0)$, whose significance is discussed below.

L1. [Initialize.] Set $a_j \leftarrow 0$ and $f_j \leftarrow j$ for $0 \le j < n$; also set $f_n \leftarrow n$. (A loopless algorithm is allowed to have loops in its initialization step, as long as the initial setup is reasonably efficient; after all, every program needs to be loaded and launched.)

L2. [Visit.] Visit the n-tuple $(a_{n-1}, \ldots, a_1, a_0)$.

L3. [Choose j.] Set $j \leftarrow f_0$, $f_0 \leftarrow 0$. (If this is the kth time we are performing the present step, j is now equal to $\rho(k)$.) Terminate if $j = n$; otherwise set $f_j \leftarrow f_{j+1}$ and $f_{j+1} \leftarrow j + 1$.

L4. [Complement coordinate j.] Set $a_j \leftarrow 1 - a_j$ and return to L2. ∎

Grayův nebinární kód

V některých případech vyžadujeme obecný případ Grayova kódu tj. generování n-tic $(a_1, a_2, \ldots, a_n)$, kde $0 \le a_j \le m_j$, u kterých platí, že dvě následující n-tice liší pouze v jediné číslici

Př.: posloupnost trojic dekadických číslic v Grayově kódu (reflected Gray decimal)

$$000, 001, \ldots, 009, 019, 018, \ldots, 011, 010, 020, 021, \ldots, 091, 090, 190, 191, \ldots, 900,$$

Modulární Grayův kód

$$000, 001, \ldots, 009, 019, 010, \ldots, 017, 018, 028, 029, \ldots, 099, 090, 190, 191, \ldots, 900.$$

Generování Grayova nebinárního kódu

Algorithm H (*Loopless reflected mixed-radix Gray generation*). This algorithm visits all n-tuples $(a_{n-1}, \ldots, a_0)$ such that $0 \le a_j < m_j$ for $0 \le j < n$, changing only one coordinate by ± 1 at each step. It maintains an array of focus pointers $(f_n, \ldots, f_0)$ to control the actions as in Algorithm L, together with an array of directions $(d_{n-1}, \ldots, d_0)$. We assume that each radix m_j is ≥ 2.

H1. [Initialize.] Set $a_j \leftarrow 0$, $f_j \leftarrow j$, and $d_j \leftarrow 1$, for $0 \le j < n$; also set $f_n \leftarrow n$.

H2. [Visit.] Visit the n-tuple $(a_{n-1}, \ldots, a_1, a_0)$.

H3. [Choose j.] Set $j \leftarrow f_0$ and $f_0 \leftarrow 0$. (As in Algorithm L, j was the rightmost active coordinate; all elements to its right have now been activated.)

H4. [Change coordinate j.] Terminate if $j = n$; otherwise set $a_j \leftarrow a_j + d_j$.

H5. [Reflect?] If $a_j = 0$ or $a_j = m_j - 1$, set $d_j \leftarrow -d_j$, $f_j \leftarrow f_{j+1}$, and $f_{j+1} \leftarrow j + 1$. (Coordinate j has thus become passive.) Return to H2. ∎

Generování permutací

- Permutace *n* prvků je skupina všech prvků, které jsou uspořádány v jakémkoliv možném pořadí, tzn. výběr prvků závisí na pořadí.
- Pokud se prvky ve výběru nemohou opakovat, pak počet všech možných výběrů je určen vztahem

  P(n)= n!
- Pokud se hovoří o permutacích prvků, jsou tím obvykle myšleny permutace bez opakování.

  Př. Mějme skupinu tří různých prvků *a*,*b*,*c*.

  Permutace těchto prvků představují skupiny

  abc, *acb*, *bac*, *bca*, *cab*, *cba*

Generování lexikograficky uspořádaných permutací

Algorithm L (*Lexicographic permutation generation*). Given a sequence of n elements $a_1 a_2 \ldots a_n$, initially sorted so that

$$a_1 \le a_2 \le \cdots \le a_n, \tag{1}$$

this algorithm generates all permutations of $\{a_1, a_2, \ldots, a_n\}$, visiting them in lexicographic order. (For example, the permutations of $\{1, 2, 2, 3\}$ are

1223, 1232, 1322, 2123, 2132, 2213, 2231, 2312, 2321, 3122, 3212, 3221,

ordered lexicographically.) An auxiliary element a_0 is assumed to be present for convenience; a_0 must be strictly less than the largest element a_n.

L1. [Visit.] Visit the permutation $a_1 a_2 \dots a_n$.

L2. [Find j.] Set $j \leftarrow n-1$. If $a_j \geq a_{j+1}$, decrease j by 1 repeatedly until $a_j < a_{j+1}$. Terminate the algorithm if $j = 0$. (At this point j is the largest subscript such that we have already visited all permutations beginning with $a_1 \dots a_j$. Therefore the lexicographically next permutation will increase the value of a_j.)

L3. [Increase a_j.] Set $l \leftarrow n$. If $a_j \geq a_l$, decrease l by 1 repeatedly until $a_j < a_l$. Then interchange $a_j \leftrightarrow a_l$. (Since $a_{j+1} \geq \cdots \geq a_n$, element a_l is the smallest element greater than a_j that can legitimately follow $a_1 \dots a_{j-1}$ in a permutation. Before the interchange we had $a_{j+1} \geq \cdots \geq a_{l-1} \geq a_l > a_j \geq a_{l+1} \geq \cdots \geq a_n$; after the interchange, we have $a_{j+1} \geq \cdots \geq a_{l-1} \geq a_j > a_l \geq a_{l+1} \geq \cdots \geq a_n$.)

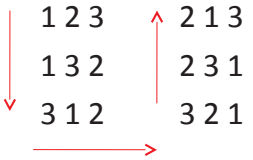
L4. [Reverse $a_{j+1} \dots a_n$.] Set $k \leftarrow j+1$ and $l \leftarrow n$. Then, if $k < l$, interchange $a_k \leftrightarrow a_l$, set $k \leftarrow k+1$, $l \leftarrow l-1$, and repeat until $k \geq l$. Return to L1. ∎

Základní myšlenka algoritmu:

1. Vezmeme posloupnost prvků {1,2, …, n-1}
2. Vložíme prvek n do každé permutace všemi možnými způsoby a uspořádáme do sloupců

Př. Vytváříme permutace nad množinou {1,2,3,4}

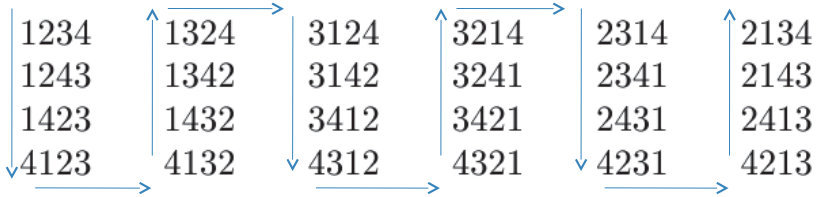
začneme pro n=2 → permutace {1 2, 2 1 } - permutace dáme do sloupců a přidáme 3 do všech možných pozic



A uspořádáme ve směru šipek

123, 132, 312, 321, 231, 213

Výsledky uložíme do sloupců a přidáme prvek 4



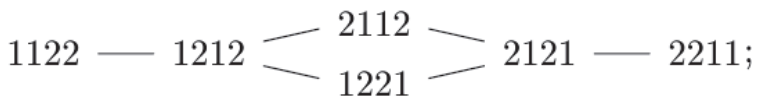
A podobně jako v předchozím případě uspořádáme a dostaneme výsledné permutace :

1234, 1243, 1423,4123,4132, 1432,1342, …, 2134, 2143,2413, 4213

Generování permutací ve kterých se mění pouze sousední elementy

Cíl: Podobně jako u Grayova kódu - vytvářet takové permutace, kde dochází ke změně pouze mezi sousedními prvky.

Tento postup není možné aplikovat pokud se v množině, nad kterou vytváříme permutace, opakují prvky.



Algoritmus generování permutací - dochází ke změnám pouze u sousedních prvků

Algorithm P (*Plain changes*). Given a sequence $a_1 a_2 \dots a_n$ of n distinct elements, this algorithm generates all of their permutations by repeatedly interchanging adjacent pairs. It uses an auxiliary array $c_1 c_2 \dots c_n$, which represents inversions as described above, running through all sequences of integers such that

$$0 \le c_j < j \qquad \text{for } 1 \le j \le n. \tag{5}$$

Another array $d_1 d_2 \dots d_n$ governs the directions by which the entries c_j change.

P1. [Initialize.] Set $c_j \leftarrow 0$ and $d_j \leftarrow 1$ for $1 \le j \le n$.

P2. [Visit.] Visit the permutation $a_1 a_2 \dots a_n$.

P3. [Prepare for change.] Set $j \leftarrow n$ and $s \leftarrow 0$. (The following steps determine the coordinate j for which c_j is about to change, preserving (5); variable s is the number of indices $k > j$ such that $c_k = k - 1$.)

P4. [Ready to change?] Set $q \leftarrow c_j + d_j$. If $q < 0$, go to P7; if $q = j$, go to P6.

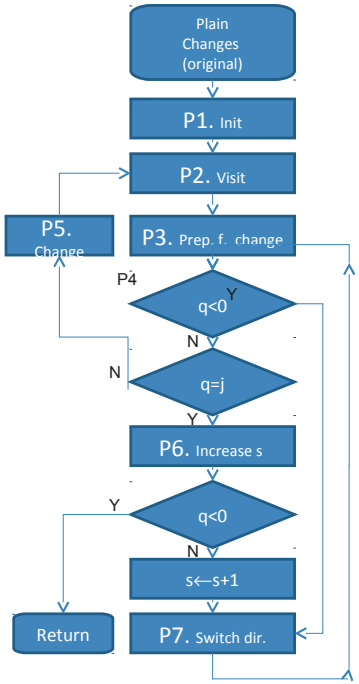
P5. [Change.] Interchange $a_{j-c_j+s} \leftrightarrow a_{j-q+s}$. Then set $c_j \leftarrow q$ and return to P2.

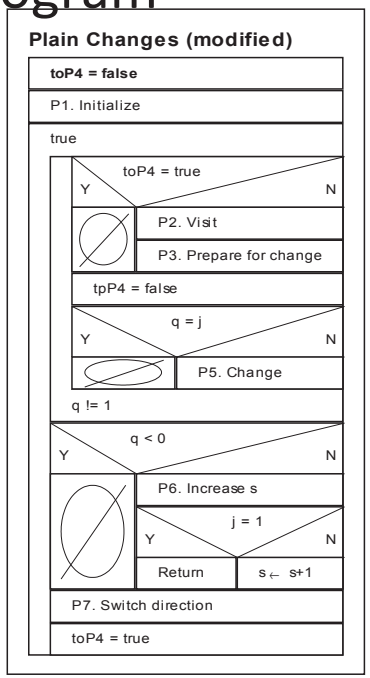
P6. [Increase s.] Terminate if $j = 1$; otherwise set $s \leftarrow s + 1$.

P7. [Switch direction.] Set $d_j \leftarrow -d_j$, $j \leftarrow j - 1$, and go back to P4. ∎

Generování kombinací

Struktogram





Kombinace *t*-té třídy z *n* prvků je skupina *t* prvků vybraných z celkového počtu *n* prvků, přičemž při výběru nezáleží na pořadí jednotlivých prvků.

Počet kombinací *t*-té třídy z *n* prvků bez opakování, tzn. žádný prvek výběru se nemůže opakovat, je

$$C_t(n) = \binom{n}{t} = \frac{n!}{t!(n-t)!}$$

- kde $\binom{n}{t}$ představuje kombinační číslo.

Uvažujme, že n=s+t .

V některé literatuře se kombinace uvádí ve tvaru (s,t)-kombinace.

Způsoby reprezentace (s,t)-kombinací:

1) Binárním řetězcem $a_{n-1}, \ldots, a_1, a_0$, pro který platí
   $a_{n-1} + \ldots a_1 + a_0 = t$. Prvky a_i nabývají hodnot
   0 … pokud prvek nebyl vybrán, nebo
   1 … pokud vybrán byl
2) Nebo vektorem $c_t, \ldots c_2\ c_1$ ve kterém jsou uloženy pozice vybraných prvků

Generování lexikografických kombinací

- Pro malé velikosti t , lze kombinace generovat následující sekvencí příkazů:

For $c_3 = 2, 3, \ldots, n-1$ (in this order) do the following:
  For $c_2 = 1, 2, \ldots, c_3 - 1$ (in this order) do the following:
    For $c_1 = 0, 1, \ldots, c_2 - 1$ (in this order) do the following:
      Visit the combination $c_3 c_2 c_1$. (15)

Table 1
THE (3, 3)-COMBINATIONS AND THEIR EQUIVALENTS

$a_5a_4a_3a_2a_1a_0$	$b_3b_2b_1$	$c_3c_2c_1$	$d_3d_2d_1$	$e_3e_2e_1$	$p_3p_2p_1p_0$	$q_3q_2q_1q_0$	path
000111	543	210	000	210	4111	3000	
001011	542	310	100	310	3211	2100	
001101	541	320	110	320	3121	2010	
001110	540	321	111	321	3112	2001	
010011	532	410	200	010	2311	1200	
010101	531	420	210	020	2221	1110	
010110	530	421	211	121	2212	1101	
011001	521	430	220	030	2131	1020	
011010	520	431	221	131	2122	1011	
011100	510	432	222	232	2113	1002	
100011	432	510	300	110	1411	0300	
100101	431	520	310	220	1321	0210	
100110	430	521	311	221	1312	0201	
101001	421	530	320	330	1231	0120	
101010	420	531	321	331	1222	0111	
101100	410	532	322	332	1213	0102	
110001	321	540	330	000	1141	0030	
110010	320	541	331	111	1132	0021	
110100	310	542	332	222	1123	0012	
111000	210	543	333	333	1114	0003	

- pro velká t lze použít následující algoritmus.

Algorithm L (*Lexicographic combinations*). This algorithm generates all t-combinations $c_t \ldots c_2 c_1$ of the n numbers $\{0, 1, \ldots, n-1\}$, given $n \geq t \geq 0$. Additional variables c_{t+1} and c_{t+2} are used as sentinels.

L1. [Initialize.] Set $c_j \leftarrow j - 1$ for $1 \leq j \leq t$; also set $c_{t+1} \leftarrow n$ and $c_{t+2} \leftarrow 0$.

L2. [Visit.] Visit the combination $c_t \ldots c_2 c_1$.

L3. [Find j.] Set $j \leftarrow 1$. Then, while $c_j + 1 = c_{j+1}$, set $c_j \leftarrow j - 1$ and $j \leftarrow j + 1$; repeat until $c_j + 1 \neq c_{j+1}$.

L4. [Done?] Terminate the algorithm if $j > t$.

L5. [Increase c_j.] Set $c_j \leftarrow c_j + 1$ and return to L2. ∎

The running time of this algorithm is not difficult to analyze. Step L3 sets $c_j \leftarrow j - 1$ just after visiting a combination for which $c_{j+1} = c_1 + j$, and the number of such combinations is the number of solutions to the inequalities

$$n > c_t > \cdots > c_{j+1} \geq j; \qquad (16)$$

Nebo rychlejší verze tohoto algoritmu – viz následující slide.

Algorithm T (*Lexicographic combinations*). This algorithm is like Algorithm L, but faster. It also assumes, for convenience, that $t < n$.

T1. [Initialize.] Set $c_j \leftarrow j - 1$ for $1 \le j \le t$; then set $c_{t+1} \leftarrow n$, $c_{t+2} \leftarrow 0$, and $j \leftarrow t$.

T2. [Visit.] (At this point j is the smallest index such that $c_{j+1} > j$.) Visit the combination $c_t \ldots c_2 c_1$. Then, if $j > 0$, set $x \leftarrow j$ and go to step T6.

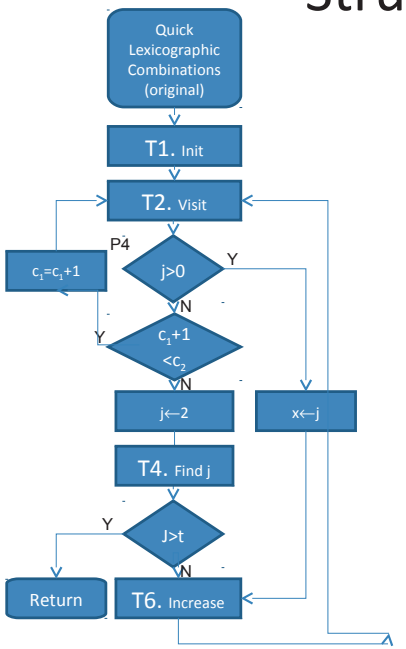
T3. [Easy case?] If $c_1 + 1 < c_2$, set $c_1 \leftarrow c_1 + 1$ and return to T2. Otherwise set $j \leftarrow 2$.

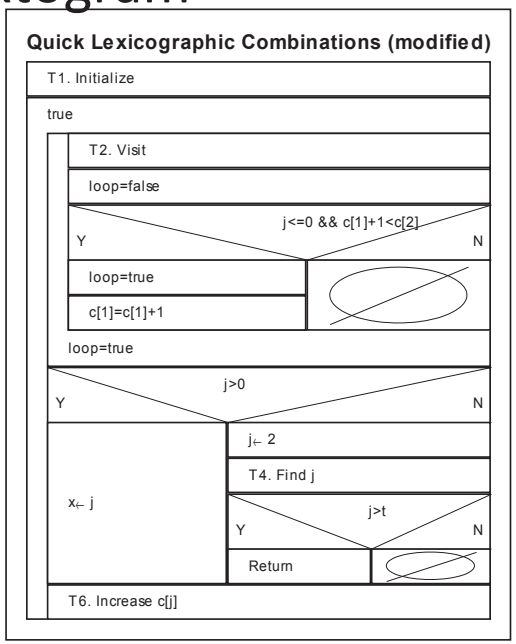
T4. [Find j.] Set $c_{j-1} \leftarrow j - 2$ and $x \leftarrow c_j + 1$. If $x = c_{j+1}$, set $j \leftarrow j + 1$ and repeat this step until $x < c_j$.

T5. [Done?] Terminate the algorithm if $j > t$.

T6. [Increase c_j.] Set $c_j \leftarrow x$, $j \leftarrow j - 1$, and return to T2. ∎

Struktogram



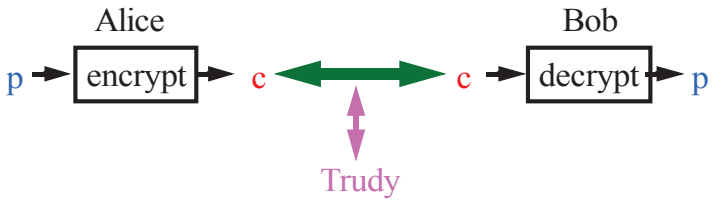


Úvod do kryptografie

- Secrecy
- Ciphers
- Secret Key Cryptography
- Key Exchange
- Public Key Cryptography
- Digital Signatures

Utajení

- Scénář: Alice chce poslat zprávu (přímý text, plaintext p) Bobovi. Komunikační kanál není chráněný a může být odposloucháván Trudy. Pokud mají Alice a Bob předem domluvený způsob šifrování zprávy (šifru, cipher), pak může být zpráva odeslána utajeně (šifrovaný text, ciphertext c)



Jakou použít šifru?
Jaká je složitost šifrování/dešifrování?
Jaká je velikost šifry vzhledem k otevřenému textu?
Pokud spolu Alice a Bob předtím nekomunikovali, jakým způsobem se dozvědí o šifře ?

Proč vlastně šifrujeme data?

- Potřeba utajovat určité informace je stará jak lidstvo
- o kryptologii hovoříme až v případě, kdy všichni používají stejný vyjadřovací prostředek (např. písmo)
- Od vzniku Internetu se mnohonásobně zvyšuje počet připojených počítačů
- V otevřených sítích jako je INTERNET je jednoduché jakoukoliv informaci ODPOSLECHNOUT a následně i ZNEUŽÍT!!
- v poslední době dochází k masovému využívání šifrování z různých důvodů
- Proto vznikla potřeba skrýt citlivé informace před nepovolanými osobami

Co to vlastně je kryptografie?

- Kryptosystém je systém umožňující šifrování a dešifrování zpráv.
- Šifrování, neboli kryptografie je transformace dat do nečitelné formy.

  Důvod - ochránit důvěrné a osobní informace znemožněním jejich čitelnosti těmi, komu nejsou určeny
- Dešifrování je opačný postup, tedy transformace šifrovaných dat do jejich původní (srozumitelné) podoby

Šifrování a dešifrování vyžaduje užití nějaké tajné informace, obvykle označované jako klíč

Šifrování – základní pojmy

- Kryptografie -věda o tvorbě šifer
- Kryptoanalýza - věda o prolamování šifer
- Kryptologie – věda o šifrování, zahrnuje kryptografii a kryptoanalýzu
- Otevřený text (plaintext) - originální tvar dat (to co má být zašifrováno)
- Šifrovaný text (ciphertext) – zašifrovaný tvar zprávy
- Šifrování (kryptování, enkryptování enciphering) – proces přeměny otevřeného textu na šifrovaný text
- Dešifrování (dekryptování, deciphering) – přeměna šifrovaného textu na otevřený text

Hodnocení kryptosystémů

- Různé úhly pohledu – rychlost výpočtu, bezpečnost, snadnost implementace

Základní zásady:

- šifrováním by neměl narůstat objem dat, pokud naroste, tak jen o konstantní velikost
- implementace by měla být únosně složitá
- rozumná implementace by měla být přiměřeně rychlá
- šifra by neměla obsahovat žádná omezení na data na která bude použita
- chyby při šifrování by se neměli nepřiměřeně šířit

Dělení šifer

Z hlediska zpracování zprávy:

- Blokové šifry – pracují s celými bloky dat (obvykle 8-128 bytů)
- Proudové šifry (streamové) - pracují s jednotlivými bitu zprávy zvlášť,
  - jsou považovány za méně bezpečné
  - jsou pomalejší než šifry blokové

Z hlediska šifrování :

- Symetrické šifry – odesilatel i příjemce sdílí jedno tajemství (klíč) nutné k šifrování a zašifrování zprávy
- Asymetrické šifry – odesilatel a příjemci šifrují a dešifrují zprávu různými klíči, nemusí spolu sdílet žádné tajemství.
  - Nevýhoda: je o několik řádů pomalejší než symetrická kryptografie

Historie šifrování

Steganografie – ukrývání zprávy jako takové (tajné inkousty, vyrývání zprávy do dřevěné tabulky zalité voskem, apod.)

Použití kódů – pro každou činnost se vytvoří kódové slovo

Substituční šifry

Obecně spočívá v nahrazení každého znaku zprávy jiným znakem podle nějakého pravidla. Nejstarší popis šifry Kámasútra (4.stol.). Nevýhoda - snadné prolomení šifry.

- **Posun písmen** (Caesarova šifra) – každé písmeno zprávy je posunuté o pevný počet pozic

  zaměň znak *a* za *d*
  zaměň znak *b* za *e*
  ...
  zaměň znak *z* za *c*

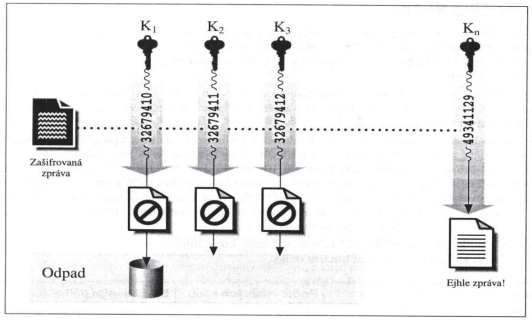
- **Tabulka záměn** – záměna znaku za jiný, bez jakékoliv souvislosti popř. na základě znalosti hesla

Šifra svobodných zednářů:

Narušení kódu

- Metoda hrubé síly - zkouší se všechny možné klíča a hledá se smysluplná zpráva.

  Nevýhoda: časově náročná metoda, substituční šifry jí dokáží čelit (volbou vhodně velkého klíče)



Narušení kódu

- Pokud Trudy použije statistiku jazyka ve kterém je šifrovaná zráva, může jednoduše prolomit monoalfabetickou substituční šifru.

  Např. v angličtině se nejčastěji vyskytují:
  - *znaky : e, t, o, a, n, i, ...*
  - bigrams: th, in, er, re, an, ...
  - trigrams: the, ing, and, ion, ...

- Využití frekvenční analýzy jazyka v kryptoanalýze poprvé zmiňuje v 9. století arabský filosof al-Kindi

■ *Příklad tabulky záměny, s použitím slova **VESLO** jako klíče:*

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
V	**E**	**S**	**L**	**O**	A	B	C	D	F	G	H	I	J	K	M	N	P	Q	R	T	U	W	X	Y	Z

nebo

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
V	**E**	**S**	**L**	**O**	P	Q	R	T	U	W	X	Y	Z	A	B	C	D	F	G	H	I	J	K	M	N

Příklad (S. Singh, The Code Book, 1999, v češtině Kniha kódů a šifer, 2003)

- Šifra
- PCQ VMJYPD LBYK LYSO KBXBJXWXV BXV ZCJPO EYPD KBXBJYUXJ LBJOO KCPK. CP LBO LBCMKXPV XPV IYJKL PYDBL, QBOP KBO BXV OPVOV LBO LXRO CI SX'XJMI, KBO JCKO XPV EYKKOV LBO DJCMPV ZOICJO BYS, KXUYPD: 'DJOXL EYPD, ICJ X LBCMKXPV XPV CPO PYDBLK Y BXNO ZOOP JOACMPLYPD LC UCM LBO IXZROK CI FXKL XDOK XPV LBO RODOPVK CI XPAYOPL EYPDK. SXU Y SXEO KC ZCRV XK LC AJXNO X IXNCMJ CI UCMJ SXGOKLU?'

  OFYRCDMO, LXROK IJCS LBO LBCMKXPV XPV CPO PYDBLK

Frekvenční analýza

- Identifikace často se vyskytujících písmen, bigramů, trigramů
- PCQ VMJYPD LBYK LYSO KBXBJXWXV BXV ZCJPO EYPD KBXBJYUXJ LBJOO KCPK. CP LBO LBCMKXPV XPV IYJKL PYDBL, QBOP KBO BXV OPVOV LBO LXRO CI SX'XJMI, KBO JCKO XPV EYKKOV LBO DJCMPV ZOICJO BYS, KXUYPD: 'DJOXL EYPD, X LBCMKXPV XPV CPO PYDBLK Y BXNO ZOOP JOACMPLYPD LC UCM LBO IXZROK CI FXKL XDOK XPV LBO RODOPVK CI XPAYOPL EYPDK. SXU Y SXEO KC ZCRV XK LC AJXNO X IXNCMJ CI UCMJ SXGOKLU?'

  OFYRCDMO, LXROK IJCS LBO LBCMKXPV XPV CPO PYDBLK
- První odhad: LBO je THE

Frekvenční analýza

- Předpokládejme, že pokud LBO reprezentuje THE můžeme nahradit L za T, B za H, and O za E a dostaneme
- PCQ VMJYPD THYK TYSE KHXHJXWXV HXV ZCJPE EYPD KHXHJYUXJ THJEE KCPK. CP THE THCMKXPV XPV IYJKT PYDHT, QHEP KHO HXV EPVEV THE LXRE CI SX'XJMI, KHE JCKE XPV EYKKOV THE DJCMPV ZEICJE HYS, KXUYPD: 'DJEXT EYPD, ICJ X LHCMKXPV XPV CPE PYDHLK Y HXNE ZEEP JEACMPTYPD TC UCM THE

  IXZREK CI FXKL XDEK XPV THE REDEPVK CI XPAYEPT EYPDK. SXU Y SXEE KC ZCRV XK TC AJXNE X IXNCMJ CI UCMJ SXGEKTU?'

  EFYRCDME, TXREK IJCS THE LHCMKXPV XPV CPE PYDBTK

Řešení

- **Kód**

X Z A V O I D B Y G E R S P C F H J K L M N Q T U W
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- **Původní text:** Now during this time Shahrazad had borne King Shahriyar three sons. On the thousand and first night, when she had ended the tale of Ma'aruf, she rose and kissed the ground before him, saying: 'Great King, for a thousand and one nights I have been recounting to you the fables of past ages and the legends of ancient kings. May I make so bold as to crave a favour of your majesty?' Epilogue, Tales from the Thousand and One Nights

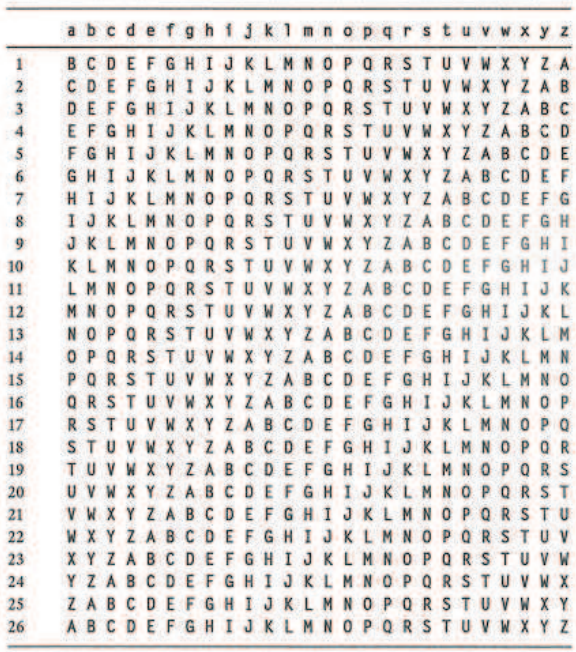
Aditivní šifry

- Vigenerova šifra - speciální případ polyalfabetické šifry. Základem šifrování je Vigenerův čtverec (otevřený text, následovaný 26 šifrovými abecedami, z nichž každá je oproti předchozí posunutá o jeden znak.

Šifrování se provádí tak, že každý znak šifrujeme podle jiné abecedy (jiného řádku). Jaký řádek čtverce použijeme je určeno klíčem (heslem)

Příklad šifrování textu „Zlato je uloženo v jeskyni“ s klíčem (heslem) „POKLAD“

P	O	K	L	A	D	P	O	K	L	A	D	P	O	K	L	A	D	P	O	K	L
z	l	a	t	o	j	e	u	l	o	z	e	n	o	v	j	e	s	k	y	n	i
O	Z	K	E	O	M	T	I	V	Z	Z	H	C	C	F	U	E	V	Z	M	X	T

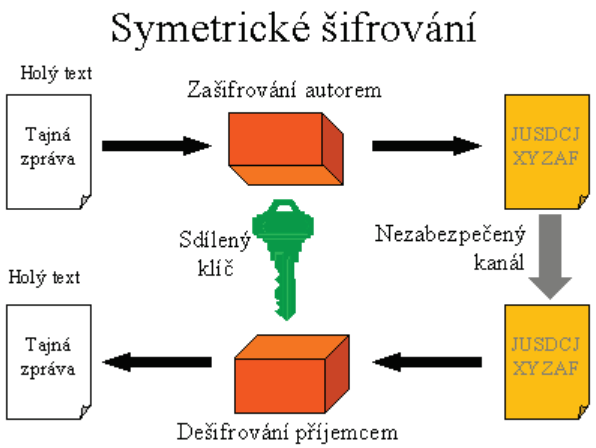


	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
2	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
3	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
4	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
5	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
6	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
7	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
8	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
9	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
10	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
11	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
12	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
13	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
14	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
15	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
16	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
17	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
18	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
19	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
20	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
21	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
22	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
23	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
24	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
25	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
26	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

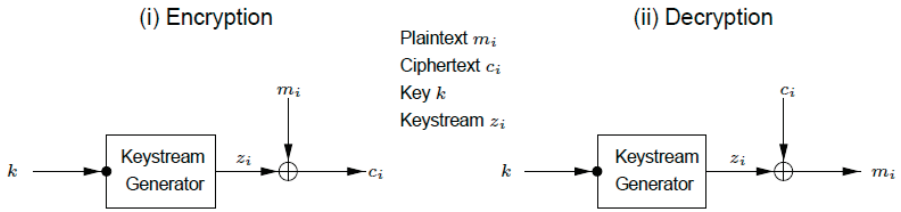
Symetrické šifrování

- Je nejpoužívanějším typem šifrovacího algoritmu
- Používá stejný šifrovací klíč k šifrování i dešifrování - což je jeho největší slabina
- Je velmi rychlý a používá se při velkém množství dat
- Klíč se musí dostat od odesilatele k adresátovi bezpečným kanálem (cestou), aby adresát mohl zprávu dešifrovat
- Pokud takový bezpečný kanál existuje, je často jednodušší zprávu nešifrovat a poslat ji rovnou tímto kanálem.

Symetrické šifrování



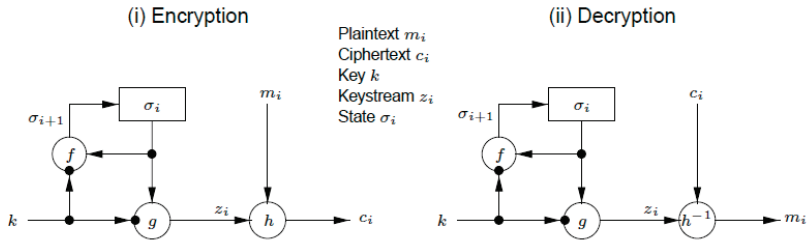
Binárně-aditivní proudová šifra – synchronní šifra, ve které jsou vstupní text ,šifrovaný text a keystream binární čísla a jako výstupní funkce h je použita operace XOR



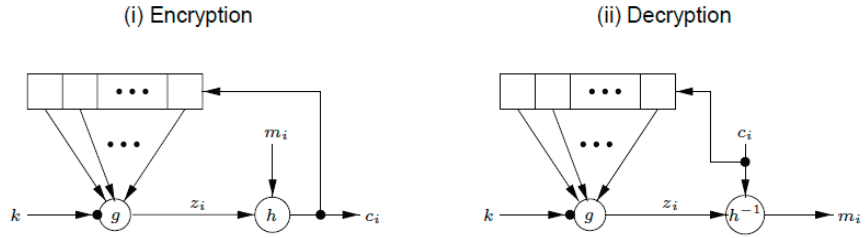
Proudové symetrické šifry

• n-bitový klíč K je použit pro generování proudu bitů delšího klíče (keystream), ten je použit k šifrování informace (provádí se operace XOR mezi bity keystreamu a vstupního textu.

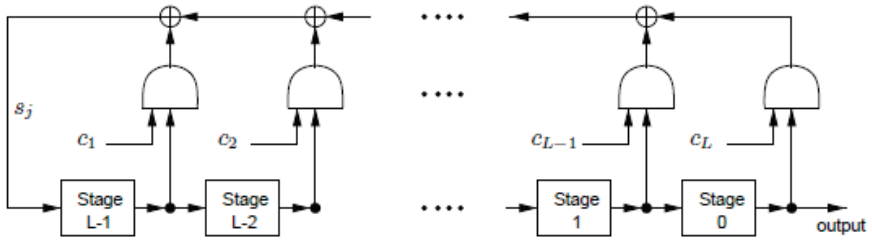
Proudové šifry se dělí na:
• synchronní - proudový klíč (keystream) je generován nezávisle na vstupním a šifrovaném textu.



• šifry s vlastní synchronizací - proudový klíč (keystream) je funkcí klíče a pevného počtu bitů šifrovaného textu



Proudové šifry používají ke generování proudového klíče posuvné registry (LSFR – Linear Feedback Shift Register)



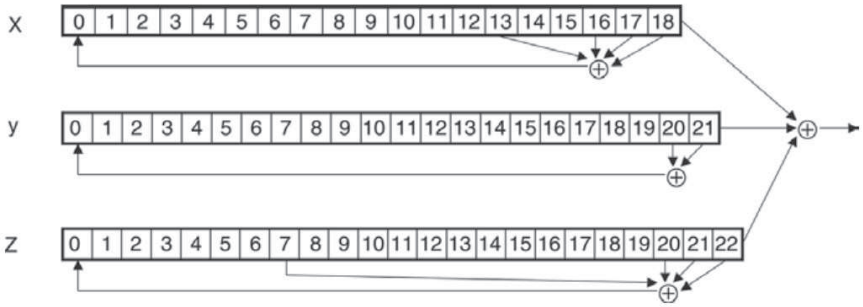
Šifra A5

A5 je proudová šifra vyvinutá k šifrování GSM hovoru mezi mobilní stanicí a základnovou stanicí BTS (Base Transceiver Station). Hovor v síti operátora, tj. od BTS přes BSC (Base Station Controller) až do ústředny MSC (Mobile Switching Center), není dále šifrován, takže ho lze odposlouchávat.
Existuje ve dvou variantách, které jsou na bázi proudových šifer: A5/1 a A5/2.

Šifra produkuje vždy 228 bitů proudu klíče, 114 bitů se používá pro šifrování komunikace od telefonu k základové stanici a 114 bitů pro šifrování komunikace v opačném směru.
Tajný klíč je uložen na SIM kartě telefonu.
Při každém spojení se sítí je z tajného klíče na SIM kartě a z náhodné výzvy o 128 bitech během autentizace vygenerován klíč K_c pro šifru A5.
Z tohoto klíče je pak vygenerováno 228 bitů proudu klíče.

Generátor proudového klíče tvoří 3 LSFR registry:

- X (19 bitů),
- Y (22 bitů),
- Z (23 bitů)



Operace v registru X

$$t = x_{13} \oplus x_{16} \oplus x_{17} \oplus x_{18}$$
$$x_i = x_{i-1} \quad \text{for } i = 18, 17, 16, \ldots, 1$$
$$x_0 = t$$

Operace v registru Y

$$t = y_{20} \oplus y_{21}$$
$$y_i = y_{i-1} \quad \text{for } i = 21, 20, 19, \ldots, 1$$
$$y_0 = t$$

Operace v registru Z

$$t = z_7 \oplus z_{20} \oplus z_{21} \oplus z_{22}$$
$$z_i = z_{i-1} \quad \text{for } i = 22, 21, 20, \ldots, 1$$
$$z_0 = t$$

Šifra A5/1 je implementována hardwarově - v každém hodinovém cyklu se určuje hodnota m jako

$$m = \text{maj}(x_8, y_{10}, z_{10})$$

kde funkce **maj(x,y,z)** vrací 0 pokud je většina bitů x,y,z nulová, jinak vrací 1. Registry X,Y, Z provádí posun, pokud jsou splněna následující pravidla :

$$\text{If } x_8 = m \text{ then } X \text{ steps}$$
$$\text{If } y_{10} = m \text{ then } Y \text{ steps}$$
$$\text{If } z_{10} = m \text{ then } Z \text{ steps}$$

Výsledný bit proudového klíče s je pak generován jako:

$$s = x_{18} \oplus y_{21} \oplus z_{22}$$

Mezi bitem proudového klíče a bitem otevřeného textu (při šifrování) popř. bitem šifrovaného textu se provádí operace XOR.

Šifra RC4

Používá se při zabezpečení bezdrátových WiFi sítí pracujících v bezlicenčních pásmech nazývané WEP (Wired Equivalent Privacy), které je součástí původního standardu IEEE 802.11 z roku 1999. Tato aplikace algoritmu RC4 však není ideálním příkladem použití.
WEP používá nevhodně aplikovanou proudovou šifru RC4 (Rivest Cipher verze 4). Podrobný popis algoritmu RC4 (vyvinutý Ronem Rivestem v roce 1987) byl známý pouze osobám, které podepsali důvěrný dodatek, neboť byl ve vlastnictví RSA Data Security, Inc. V září roku 1994 však anonymní odesílatel uveřejnil zdrojový kód algoritmu, a tak se rychle rozšířil po celém světě.
Algoritmus RC4 bylů v roce 2004 označen jako zastaralý a nedoporučovaný, přesto se stále používá.
Oproti A5, která je orientovaná na hw implementaci , RC4 je orientován na sw implementaci .
RC 4 generyje v každém kroku proudový klíč o délce 8 bitů (A5 generoval 1 bit).

Princip RC4:

základ algoritmu generování proudového klíče tvoří vyhledávací tabulka, která obsahuje permutace 256-bytových hodnot. Pokaždé, když je generován byte proudového klíče, je vyhledávací tabulka modifikována tak, aby obsahovala permutace množiny {0, 1, 2, …, 255}.

1.Inicializace tabulky klíčem key:

```
for i = 0 to 255
    S[i] = i
    K[i] = key[i mod N]
next i
j = 0
for i = 0 to 255
    j = (j + S[i] + K[i]) mod 256
    swap(S[i],S[j])
next i
i = j = 0
```

Klíč key může mít délku od 0 do 256 bytů

2. Generování bytu proudového klíče

```
i = (i + 1) mod 256
j = (j + S[i]) mod 256
swap(S[i], S[j])
t = (S[i] + S[j]) mod 256
keystreamByte = S[t]
```

3. XOR mezi bytem proudového klíče a otevřeným textem (při šifrování), popř. šifrovaným textem (při dešifrování).

DES – Data encryption standard

- Jde o nejpoužívanější šifru na světě.
- Je výsledkem veřejné soutěže v roce 1977.
- Délka klíče je 56 bitů, což už v době vzniku bylo považováno za nepříliš bezpečné.
- Tuto délku klíče do původního návrhu IBM vnesla National Security Agency.
- Jde o iterovanou šifru, kdy je původní blok otevřené zprávy postupně šifrován pomocí šifrovacích zobrazení $E_{k(1)}, E_{k(2)}, \ldots, E_{k(16)}$.
- Délka bloku je 64 bitů.
- Jednotlivá šifrování se nazývají *runda*.
- Původní klíč délky 56 bitů je *expandován* na 16 rundovních klíčů *k(1), k(2),…,k(16),* každý délky 48 bitů.

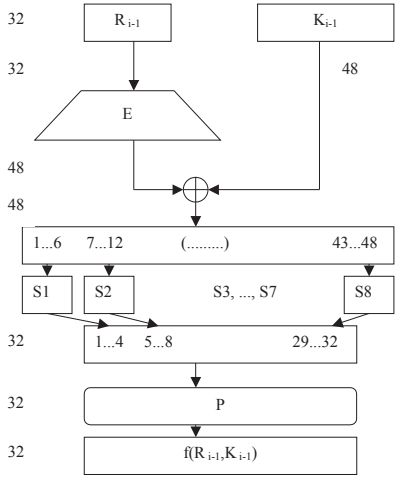
Rundovní funkce

Toto je schématické znázornění rundovní funkce.

E je expanzní funkce, která z posloupnosti 32 bitů udělá 48 bitů.

S – boxy nelineárně transformuji šestice bitů ve čtveřice bitů.

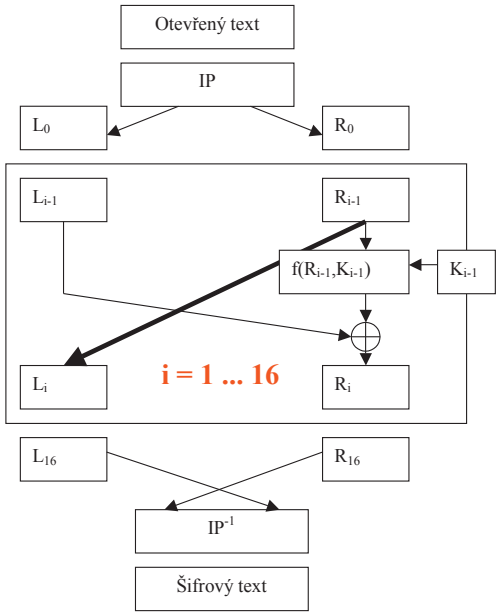
P je permutace na 32 bitech.



Základní schéma DES

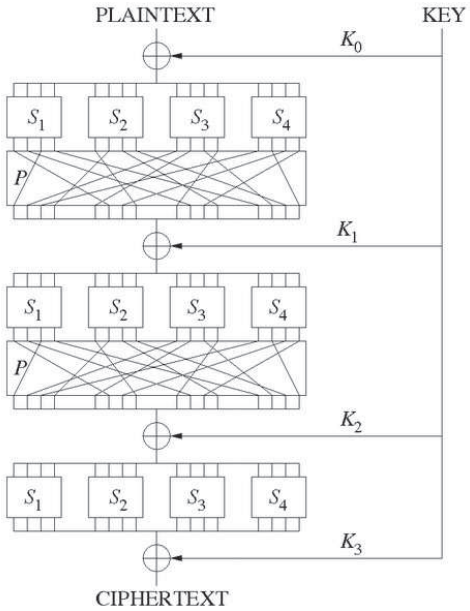
IP je nějaká permutace na 64 bitech.

Blok o 64 bitech se rozdělí na levou a pravou polovinu délky 32 bitů.

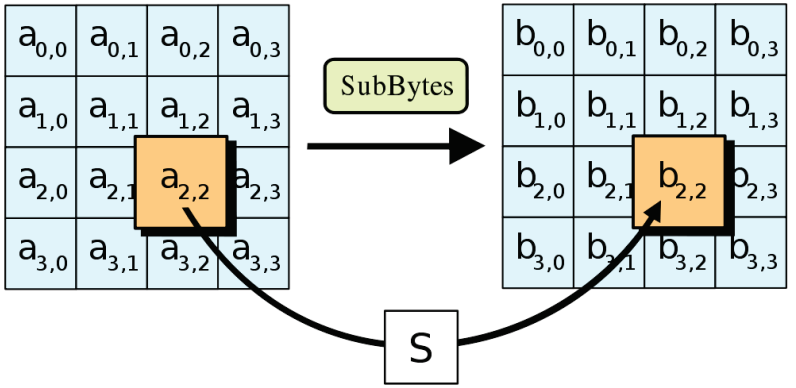


AES – Advanced encryption standard

- V roce 1997 byla vyhlášena celosvětová soutěž na návrh blokové šifry nové generace.
- Přihlásilo se 15 účastníků.
- Jako vítěz byla šifra navržená belgickými kryptology V. Rijmenem a J. Daemenem.
- Je založena na šifrovacím algoritmu Rijndael.
- Délka bloku je 128 bitů.
- AES podporuje tři délky klíčů – 128, 192 a 256 bitů.
- Počet rund se může měnit od 10 do 14 v závislosti na velikosti klíče.
- Algoritmus šifrování pracuje na principu tzv. substitučně permutační sítě SPN



Operace 1 rundy pro klíč délky 128 bitů



Popis algoritmu:

1.KeyExpansion – z šifrovacího klíče je odvozen rundovní klíč K_i

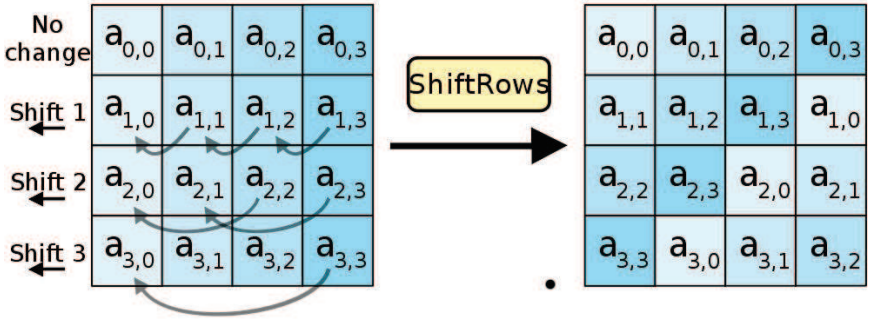
2.Inicializace - mezi zpracovávaným blokem a klíčem K_i je provedena operace XOR

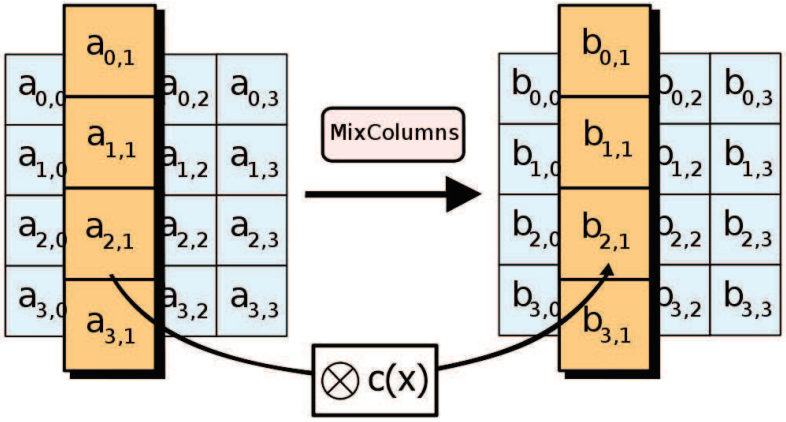
3.Provedení rundy - každá runda se skládá ze čtyř operací

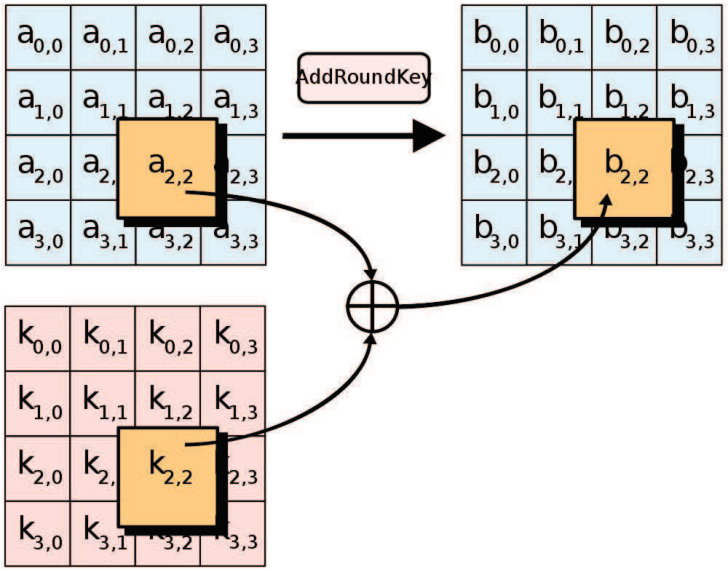
1. SubBytes - nelineární substituce, při které je každý byte nahrazen jiným z vyhledávací tabulky
2. ShiftRows - transpoziční krok – každý řádek stavu je cyklicky posunut
3. MixColumns – operace, která vezme bajty sloupce a lineární transformací je změní
4. AddRoundKey – mezi každým bytem stavu a subklíčem K_i se provádí bitový XOR

TABLE 3.4. AES ByteSub.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16







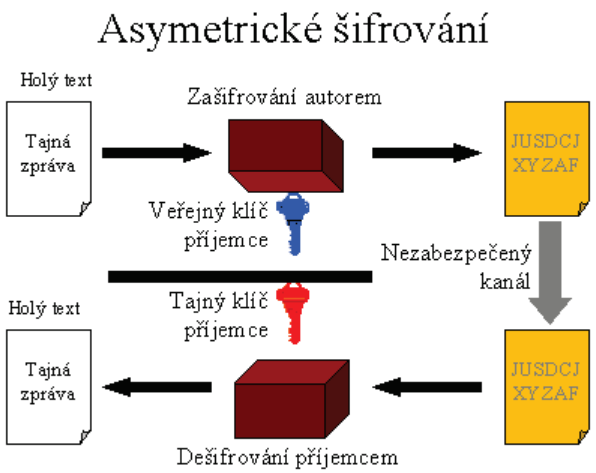
Další symetrické šifry:

- **Triple-DES** – šifrovací algoritmus DES se používá 3x se dvěma různými klíči
- **RC2, RC4** – Rivestovy kódy - klíč o délce 1 – 1024 bitů, RC2 je bloková šifra podobná DES, RC4 - proudová šifra
- **IDEA** (International Data Encryption Algorithm) – 128 bitový klíč

Asymetrické šifrování

- Používá jiný klíč k zašifrování a jiný klíč zpátky k dešifrování
- První z nich se nazývá veřejný, ostatní ho musejí znát. Druhý klíč se nazývá privátní
- Asymetrický šifrovací systém (systém s veřejným klíčem) je založen na principu jednocestné funkce, což jsou operace, které lze snadno provést pouze v jednom směru: ze vstupu lze snadno spočítat výstup, z výstupu však je velmi obtížné nalézt vstup.
- Nejběžnějším příkladem je například násobení: je velmi snadné vynásobit dvě i velmi velká čísla, avšak rozklad součinu na činitele (tzv. faktorizace) je velmi obtížný. (Na tomto problému je založen např. algoritmus RSA.)

Asymetrické šifrování



Asymetrické šifrování – algoritmy

- RSA (Rivest, Shamir, Aldeman) – algoritmus vhodný jak pro podepisování, tak pro šifrování

Princip:

Bezpečnost RSA je postavena na předpokladu, že rozložit velké číslo na součin prvočísel (faktorizace) je velmi obtížná úloha. Z čísla $n = pq$ je tedy v rozumném čase prakticky nemožné zjistit činitele p a q, neboť není znám žádný algoritmus faktorizace, který by pracoval v polynomiálním čase vůči velikosti binárního zápisu čísla n. Naproti tomu násobení dvou velkých čísel je elementární úloha.

Popis algoritmu:

Alice a Bob chtějí komunikovat prostřednictvím otevřeného (nezabezpečeného) kanálu a Bob by chtěl Alici poslat soukromou zprávu.

Tvorba klíčového páru:

Nejprve si bude Alice muset vyrobit pár veřejného a soukromého klíče:

1. Zvolí dvě různá velká náhodná prvočísla p a q.
2. Spočítá jejich součin $n = pq$.
3. Spočítá hodnotu Eulerovy funkce $\varphi(n) = (p - 1)(q - 1)$.
4. Zvolí celé číslo e menší než $\varphi(n)$, které je s $\varphi(n)$ nesoudělné.
5. Nalezne číslo d tak, aby platilo $de \equiv 1 \pmod{\varphi(n)}$.
6. Pokud e je prvočíslo tak $d = (1+r*\varphi(n))/e$, kde r = [(e-1)$\varphi(n)$^(e-2)]

Veřejným klíčem je dvojice (n, e), přičemž n se označuje jako *modul*, e jako *šifrovací* či *veřejný exponent*. **Soukromým klíčem** je dvojice (n, d), kde d se označuje jako ***dešifrovací* či *soukromý exponent***. (V praxi se klíče uchovávají v mírně upravené formě, která umožňuje rychlejší zpracování.)

- Veřejný klíč poté Alice uveřejní, respektive zcela pošle nešifrovaně Bobovi. Soukromý klíč naopak uchová v tajnosti.

Šifrování zprávy:

Bob nyní chce Alici zaslat zprávu *M*.

1. Tuto zprávu převede nějakým dohodnutým postupem na číslo *m* ($m < n$).
2. Šifrovým textem odpovídajícím této zprávě pak je číslo
   $c = m^l \bmod n$.
3. Tento šifrový text poté zašle nezabezpečeným kanálem Alici.

Dešifrování zprávy:

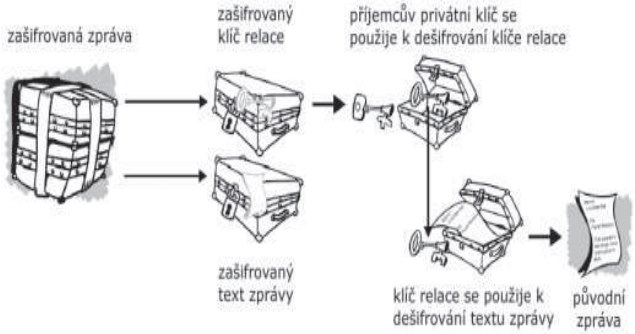
- Alice od Boba získá šifrový text *c*. Původní zprávu *m* získá následujícím výpočtem: $m = c^l \bmod n$.

Hybridní šifrování I.

- Odesilatel zvolí klíč, kterým symetricky zašifruje zprávu. Tento klíč zašifruje veřejným klíčem adresáta a pošle ho spolu se zprávou adresátovi
- Adresát tedy dostane asymetricky zašifrovaný klíč a symetricky zašifrovanou zprávu.
- Klíč dešifruje svým privátním klíčem a použije ho k dešifrování textu
- Tím zaniká problém distribuce klíče při symetrickém šifrování a zároveň se celý proces zrychlí.

Hybridní šifrování II.

- Pokud chtějí dva počítače komunikovat přes otevřenou síť, kde je každý může „odposlechnout“, vytvoří *relaci*
- Na začátku vygeneruje jeden z nich klíč, zašifruje ho veřejným klíčem 2. počítače a pošle
- Druhý počítač si klíč dešifruje, oba dva mají stejný klíč, který kromě nich nikdo jiný nezná. Mohou tedy používat symetrické šifrování
- Každá další komunikace je symetricky zašifrovaná



Kryptografické hashovací funkce

- Matematická funkce
- Vstup: posloupnost libovolné konečné délky (text, hudba, obrázky, video,…)
- Výstup: posloupnost konstantní délky (typicky stovky bitů) nazývá se - haš, hašé, hash, hashový kód, otisk zprávy

$$h=H(M)$$

Další obvyklé požadavky na HF zahrnují:

- **Nekorelovatelnost vstupních a výstupních bitů**
  - znemožnní statistickou kryptoanalýzu.
- **Odolnost vůči skoro-kolizím (*Near-collision resistance*)**
  - je obtížné nalézt *x* a *y* taková, že x≠y a zároveň *H*(*x*) a *H*(*y*) se liší jen v malém počtu bitů.
- **Lokální odolnost vůči získání předlohy.**
  - je obtížné najít i jen část vstupu *x* ze znalosti *H*(*x*)

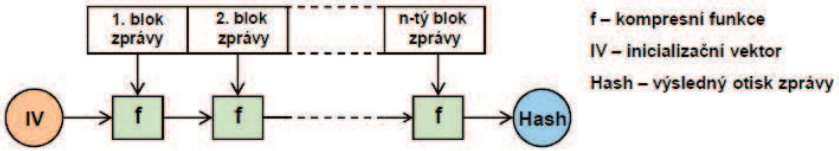
Použití hashovacích funkcí

- **Digitální / elektronický podpis –** zajištění integrity
- **PKI** - integrita X.509 certifikátů a seznamů CRL
- **Časové značky –** integrita časové značky
- **Kerberos** - generování klíčů
- **IEEE 802.1X EAP :** EAP-FAST, EAP-TLS, EAP-TTLS… používají TLS protokol, který používá hashovací funkce
- **APOP** - autentizace pomocí MD5
- **RADIUS** - integrita dat
- **IPsec (IKE, AH, ESP) –** integrita zpráv, generování pseudonáhodných posloupností.
- **SSL/TLS** - handshake protokol používá hashovací funkce kvůli tvorbě HMAC a při generování klíčů a IV
- **SSH –** HMAC a integrita přenášených dat
- **S/MIME** - použití hashovacích funkcí v digitálním podpise

Obecně se HF používají ke

- kontrola zachování integrity dat
- hashe pro digitální podpis
- kontrola uložených hesel
- porovnání obsahu dvou kopií dat
- generování pseudonáhodných posloupností (PRNG)

Merkle-Damgårdova struktura hashovací funkce



- bloky *f* provádějí kompresní funkci
- toto schéma využívá většina moderních hashovacích funkcí – MD-5, SHA-1, RIPEMD-160
- vstup musí být doplněn na celistvý násobek délky bloků
- musí být jednodznačně určitelné kolik se doplnilo (jinak by jednoduše vznikala řada kolizí)
- Merkle-Damgårdovo zesílení (doplnění výplně posledního bloku o délku zprávy)

Merkleova meta-metoda pro tvorbu HF

Vstup: funkce *f* odolná vůči kolizím.

Výstup: iterovaná CRHF h odolná vůči kolizím

Vstup délky *x* se rozdělí na *n* bloků $x_1, \ldots, x_n$ o délce *m* bitů. Poslední blok x_n se zleva doplní nulami na délku *m* bitů. Volitelně lze provést Merklovo-Dammgårdovao posílení.

Výpočet s-bitového výstupu zprávy *x:*

$h(x) = H_{t+1} = f(H_t \,||\, x_{t+1});$

$H_0 = 0$

$H_i = f(H_{i-1} \,||\, x_i).$

Používané hashovací funkce

MD2 – kompromitovaná, nepoužívá se
MD4 – kompromitovaná, nepoužívá se
MD5 –oblíbená, ale kompromitovaná funkce. Od srpna 2004 je veřejně znám postup nalezení kolizí a to i pro málo odlišná vstupní data.
RIPEMD-128 - kompromitovaná, nepoužívá se
RIPEMD-160 – může být kompromitována
WHIRLPOOL – považuje se za bezpečnou
TIGER – nebyla nalezena kolize
GRINDAHL – mladá (03/2007), jádro z AES

Používané hashovací funkce

SHA-0 – kompromitována, nepoužívá se
SHA-1 – oblíbená, ale již kompromitovaná funkce. V únoru 2005 byl zveřejněn objev algoritmu, který umožňuje nalézt kolizi podstatně rychleji než hrubou silou. Prakticky zatím neprovedeno.

SHA-2 – dosud považována za spolehlivou
- není to jedna hashovací funkce, ale více variant souhrnně označovaných jako SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512)

MD-4

- Většina dnes používaných kryptografických hashovacích funkcí vychází z algoritmu MD-4 (Message Digest)
- MD4 byla navržena s ohledem na efektivní zpracování na existujících 32bitových procesorech.
- teoretická odolnost algoritmu MD-4 proti kolizím je 2^{64} pro 128-bitový výstup
- v praxi byly nalezeny kolize v prostoru hash kódů 2^{20} ke kompresní funkci
- není považována za bezpečnou
- 3 kola po 16 krocích ..celkem 48 rund

MD-5

Vstup: řetězec proměnné délky (neomezeně dlouhé)
Výstup: pevná délka128 bitů
Počet rund: 4
Počet kroků v rundě: 16

Vstup je zpravován po úsecích délky 512 bitů
Zpráva je zarovnána tak, aby byla dělitelná 512.

Zarovnání:
- zpráva se doplní zprava o jeden bit s hodnotou „1“
- zpráva se doplní zprava bity s hodnotou „0“ až do délky o 64 bitů menší než je číslo dělitelné 512
- posledních 64 bitů obsahuje číslo reprezentující původní délku zprávy mod 2^{64}

Změny vůči MD-4

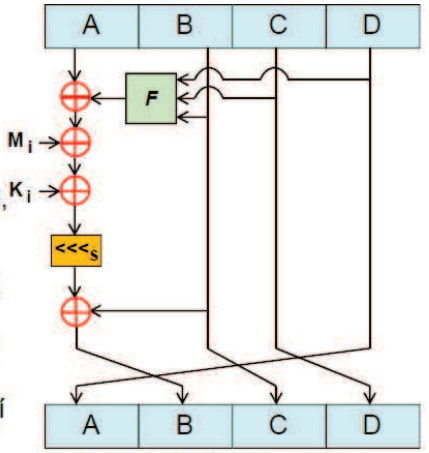
- konzervativní varianta MD4 (pomalejší)
- přidáno 4. kolo o 16 krocích
- celkem 64 rund
- změna logické funkce v druhé rundě
- jiné bitové posuny v jednotlivých krocích
- jiné aditivní konstanty (jedinečné v každé rundě)
- v každém kroku se připočítá výsledek z minulé rundy

(to urychluje tzv. lavinový efekt)

MD-5

M_i – vstupní zpráva 32bitů
K_i – konstanta 32bitů
– mění se pro každou operaci
F – nelineární funkce
$<<<_s$ – rotace o *s* bitů doleva, hodnota *s* se mění v každé operaci
⊕ – operace sčítání mod 2^{32}

A,B,C,D – pomocné registry
– na začátku konstanty
– po poslední rundě obsahují hash
- délka 128 bitů (4x32)



Jeden krok algoritmu MD-5

- vstupní blok M_i má délku 512 bitů
- před vstupem do vlastního algoritmu je rozdělen na 32 bitové části (16 různých bloků)
- každý blok vstupuje do kompresní funkce čtyřikrát, pokaždé s jinou nelineární funkcí *f*
- celkem tedy 4 kola *16 kroků = 64 rund
- konstanta K_i je spočítána jako celá část z 4294967296 násobku absolutní hodnoty funkce sin(i), kde i je úhel v radiánech

Operace v bloku „F“

$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$

$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$$

$$H(X,Y,Z) = X \oplus Y \oplus Z$$

$$I(X,Y,Z) = Y \oplus (X \vee \neg Z)$$

$\oplus$ *XOR*

$\vee$ *OR*

$\wedge$ *AND*

$\neg$ *NOT*

- v každém kole probíhá jiná nelineární operace v bloku F

Počáteční hodnoty IV

- registr A: 0x01234567
- registr B: 0x89abcdef
- registr C: 0xfedcba98
- registr D: 0x76543210

SHA-0 SHA-1

- SHA-0 představena NISTem v roce 1993 jako SHS (Secure Hash Standard)
- standard NIST PUB-180
- těsně před schválením v roce 1995 stažena (na pokyn NSA)
- mírně modifikována a schválena jako SHA-1 (PUB 180-1)
- byla přidána dodatečná rotace vlevo do každého z prováděných kol kompresní funkce
- v srpnu 1998 byl odhalen pravděpodobný důvod této změny – (Differential Collisions in SHA-0)
  www.springerlink.com/index/P795V6NJ1VJ525KP.pdf
- změna v SHA-1 ničí zarovnání bitů vstupu x po průchodu kompresní funkcí

- SHA – Secure Hash Algorithm
- odvozeno od MD4
- počet výstupních bitů rozšířen na 160
- kompresní funkce má o 1 kolo navíc
- každé kolo má 20 kroků místo původních 16
- celkem 80 rund (4x20)
- jiné hodnoty IV
- pět počátečních nenulových aditivních konstant
- konvence je big-endian (na rozdíl od algoritmů MD)

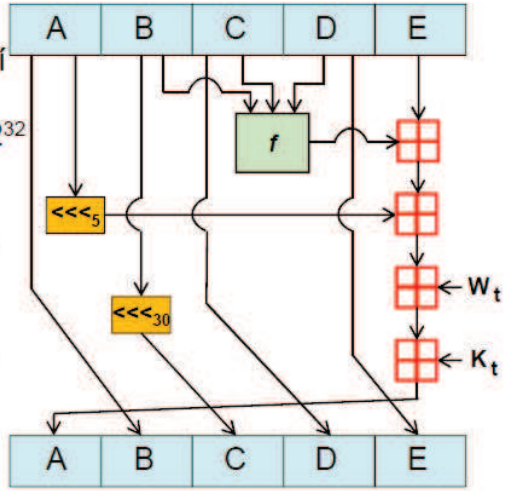
SHA-1

Vstup: řetězec proměnné délky (max. 2^{64} -1 bitů)
Výstup: pevná délka160 bitů
Počet rund: 4
Počet kroků v rundě: 20

Vstup je zpravován po úsecích délky 512 bitů
Zpráva je zarovnána tak, aby byla dělitelná 512.

Zarovnání: - zpráva se doplní zprava o jeden bit s hodnotou „1“
- zpráva se doplní zprava bity s hodnotou „0“ až do délky o 64 bitů menší než je číslo dělitelné 512
- posledních 64 bitů obsahuje číslo reprezentující původní délku zprávy

$<<<_5$ – rotace o 5
$<<<_{30}$ – rotace o 30
F – nelineární funkce mění se v každé rundě
$\oplus$ – operace sčítání mod 2^{32}
W_t – vstupní zpráva, 32b
K_t – konstanta, 32b
A,B,C,D,E – vnitřní stavy - 32bitů (každý)
- celkem 160 bitů
- na konci procesu tam je hash
- na počátku konstanty



SHA-1 expanze bloku

- vstupní blok M_i má délu 512 bitů
- před vstupem do vlastního algoritmu je rozdělen a expandován:
  1) rozdělení na 32 bitové části (16 různých bloků) označených $W_0 \ldots W_{15}$
  2) expanzní funkce $E\{0,1\}^{512} \rightarrow E\{0,1\}^{2560}$ těchto 16 částí rozšíří na 80 podle schématu

$$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) <<< 1$$

t= 16..79

Funkce *F* v jednotlivých rundách algoritmu SHA-1

$$f(t,B,C,D) = (B \wedge C) \vee (\neg B \wedge D) \qquad (0 \le t \le 19)$$
$$f(t,B,C,D) = B \oplus C \oplus D \qquad (20 \le t \le 39)$$
$$f(t,B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) \qquad (40 \le t \le 59)$$
$$f(t,B,C,D) = B \oplus C \oplus D \qquad (60 \le t \le 79)$$

Konstanta K_t

K_t = 0x5A827999 (0 ≤ t ≤ 19)
K_t = 0x6ED9EBA1 (20 ≤ t ≤ 39)
K_t = 0x8F1BBCDC (40 ≤ t ≤ 59)
K_t = 0xCA62C1D6 (60 ≤ t ≤ 79)

Počáteční hodnoty registrů A,B,C,D,E

A = 0x67452301
B = 0xEFCDAB89
C = 0x98BADCFE
D = 0x10325476
E = 0xC3D2E1F0

- dosud se pokládala za bezpečnou
- není garantována bezpečnost po roce 2010 (tzn. vhodné pouze pro krátkodobou bezpečnost)
- NIST doporučuje ukončit používání SHA-1 , nejpozději do konce r. 2010
- Než budou představeny zcela nové HF, používat SHA-2
- 2005 - navržen hardwarový SHA-1 Cracker (podobně jako u DESu)
  - 303 PC
  - v každém PC 16 desek
  - na každé desce 32 jader
  - cena: 1.000.000 $
  - doba prolomení SHA-1 2 dny

http://csrc.nist.gov/CryptoToolkit/tkhash.htm

Elektronický popis – algoritmus

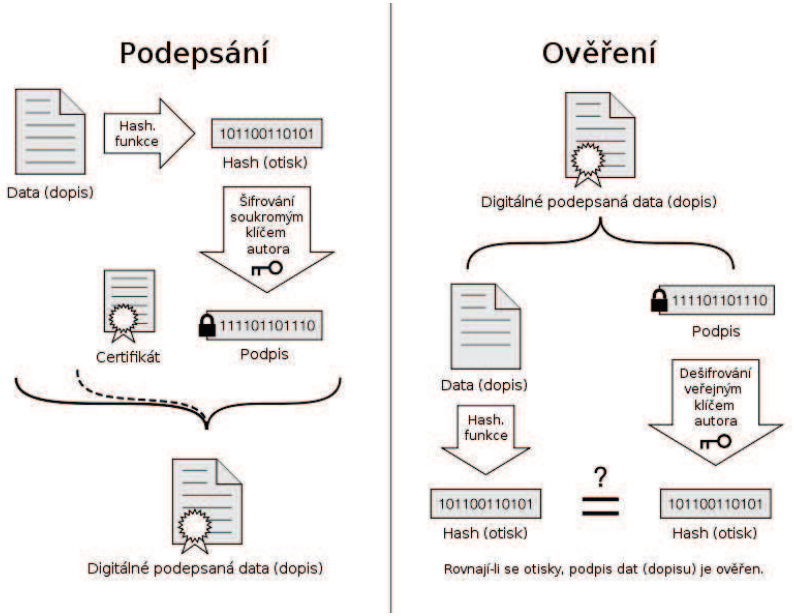
- Vybere se kryptografická hašovací funkce.
- Dále se rozhodne o parametrech L a N, které určují délku klíče. V původní verzi DSS (Digital Signature Standard) byla volba L omezena na násobky 64 v rozsahu 512 až 1024 včetně. Doporučují se dvojice L a N (1024,160), (2048,224), (2048,256) a (3072,256).
- Dále se vybere N-bitové prvočíslo q. Délka N musí být alespoň taková, jako délka výstupu použité hašovací funkce.
- Dále se vybere L-bitové prvočíslo p takové, že p-1 je násobek q.
- Nakonec se vybere g jako takové číslo, jehož multiplikativní řád modulo p je právě q. Toho lze dosáhnout dosazováním do vzorce $\boldsymbol{g=h^{(p-1)/q} \bmod p}$ pro náhodná h (kde $1< h < p$-1), dokud výsledek není různý od jedné. Většina náhodných voleb h uspěje, nejčastěji se používá h=2.
- Všechny výše zmíněné hodnoty mohou být sdílený více uživateli a nejsou tajné. Následuje vytvoření samotných klíčů.
- Nejdříve se náhodně vybere x v rozsahu $0<x<q$.
- pak se spočítá $y=g^x \bmod p$
- Veřejný klíč je pak dán jako čtveřice $\boldsymbol{(p,q,g,y)}$, soukromý klíč je dán jako x.

Elektronický podpis

Elektronický podpis jsou elektronické identifikační údaje autora (odesílatele) elektronického dokumentu, připojené k tomuto dokumentu.

Zaručený elektronický podpis je elektronický podpis v takové formě, která zaručuje (zpravidla použitím kryptografických metod):

- autenticitu– lze ověřit původnost (identitu) subjektu, kterému patří elektronický podpis),
- integritu – lze prokázat, že po podepsání nedošlo k žádné změně, soubor není úmyslně či neúmyslně poškozen,
- nepopiratelnost – autor nemůže tvrdit, že podepsaný elektronický dokument nevytvořil (např. nemůže se zříct vytvoření a odeslaní výhružného dopisu),
- může obsahovat časové razítko, které prokazuje datum a čas podepsání dokumentu.



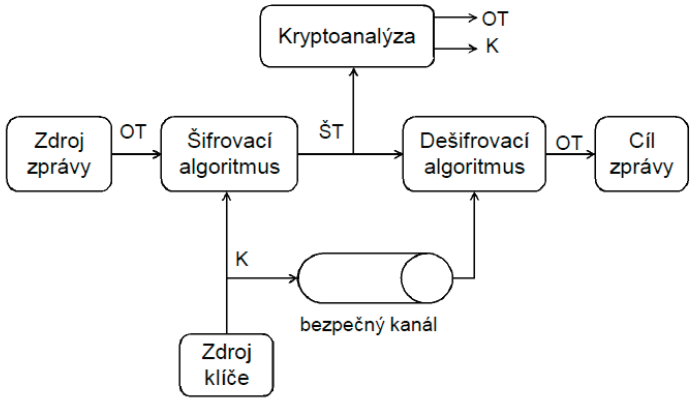
Podepisování

Při označení hašovací funkce písmenem H a zprávy písmenem z probíhá podepisování takto:

- pro danou zprávu se vybere náhodná hodnota k v rozsahu $0<k<q$
- spočítá se $r=(g^k \bmod p) \bmod q$
- spočítá se $s=(k^{-1}(H(z)+x\times r)) \bmod q$
- v nepříliš pravděpodobném případě, že je $r=0$ nebo $s=0$ se výpočet opakuje od začátku
- jinak je podpisem dvojice (r,s)

Ověřování podpisu

- pokud neplatí $0< r <q$ a $0< s <q$ pak je podpis automaticky zamítnut.
- jinak se spočítá $w = (s)^{-1} \bmod q$
- dále se spočítá $u1 = (H(z)*w) \bmod q$
- dále se spočítá $u2 = (r*w) \bmod q$
- nakonec se spočítá $v = ((g^{u1}*y^{u2}) \bmod p) \bmod q$
- Podpis platí, pokud platí $v = r$



Kvantová kryptografie

- Informace v tradičním systému (ne-kvantovém)
  - je možné ji neomezeně kopírovat
  - je možné vytvářet identické kopie zprávy
  - je možné ji měřit s libovolně malou chybou
- Informace v kvantovém systému
  - vychází z Heisenbergova principu neurčitosti
  - nelze ji libovolně kopírovat
  - těžk se uchovává, zpracovává
  - nelze vytvářet identické kopie
  - nelze kopírovat neznámý kvantový stav
  - pokus o zjištění přesné hodnoty způsobí změnu naměřené hodnoty

- prvotní myšlenka – Richard Feynman
- simulace kvantových systémů na klasickém počítači mají často exponenciálně rostoucí nároky na výpočetní čas v závislosti na délce vstupu
- nápad – nešlo by to využít obráceně -> urychlení některých algoritmů
- definice kvantového počítače - 1985 - David Deutsch
- Kvantový počítač využívá
  - principu superpozice
  - linearity kvantové mechaniky
  - jeho činnost je popsána unitárními operátory (z toho mj. plyne, že všechny s kvantovým počítačem jsou vratné)

Vernamova šifra

- one-time pad
- objevena v roce 1917 (Gilbert Vernam)
- jediný absolutně bezpečný kryptosystém
- nelze ho prolomit ani hrubou silou (brute-force attack)
- matematický důkaz provedl v roce 1949 C. E. Shannon
- problém s generováním a distribucí klíče

Použití:

- pro zabezpečení horké linky mezi Moskvou a Washingtonem
- projekt VENONA - http://en.wikipedia.org/wiki/Venona_project

Vernamova šifra

Požadavky nutné pro správnou funkci

- **Klíč je minimálně stejně dlouhý jako přenášená zpráva.**
  - jiné šifrovací systémy používají kratší klíče, což znamená, že počet možných klíčů je menší než počet možných zpráv
  - kratší klíč umožňuje útok hrubou silou
- **Klíč je dokonale náhodný.**
  - nelze použít klasické počítačové generátory pseudonáhodných posloupností
  - nejvhodnější je užití fyzikálních metod, například tepelného šumu nebo ještě lépe kvantových procesů (poločas rozpadu atd.)

Vernamova šifra

Šifrování: Znak otevřeného textu se přičítá na znak hesla pomocí operace XOR

Dešifrování: Znak šifrového textu se přičítá na znak hesla pomocí operace XOR

X	Y	X ⊗ Y	(X ⊗ Y) ⊗Y = X
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Pravdivostní tabulka pro šifrování a dešifrování pomocí funkce XOR

Nepodmíněná bezpečnost

Systém, který nelze prolomit bez ohledu na dostupné množství výpočetního výkonu, protože ŠT neposkytuje dostatek informací nutných k jednoznačnému rozpoznání odpovídajícího OT

Systém s nepodmíněnou bezpečností bude funkční i ve věku kvantových počítačů.

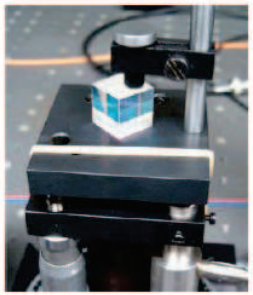
Vernamova šifra dokáže zajistit nepodmíněnou bezpečnost.

Kvantová kryptografie

- existující protokoly využívající kvantových principů spoléhají na nemožnost tvorby identických kopií neznámého kvantového stavu.
- obvykle protokoly pro bezpečnou výměnu klíče
- schopnost detekce odposlechu
- zatím se nepoužívá pro uchovávání šifrovaných informací – obtížnost dlouhodobého zamezení interakce kvantového systému s okolním prostředím
- klasická kryptografie se spoléhá na výpočetní složitost

Polarizační kódování

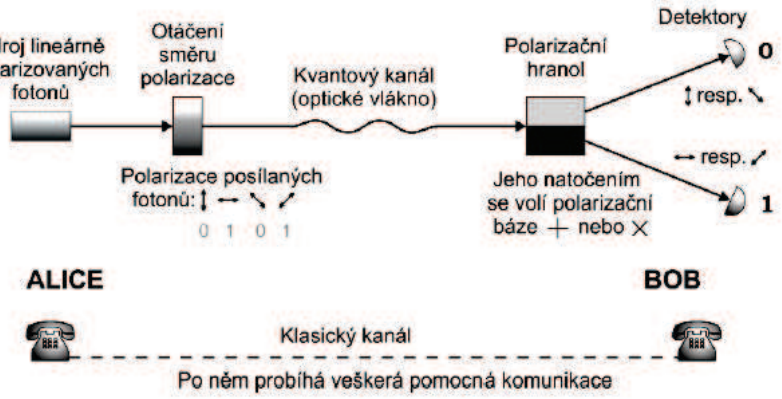
- Praktická realizace – hranol z islandského vápence
- rozdělí paprsek obsahující fotony s různou polarizací na dvě kolmé složky



Obr. a foto – Laboratoř optiky, Univerzita Palackého, Olomouc

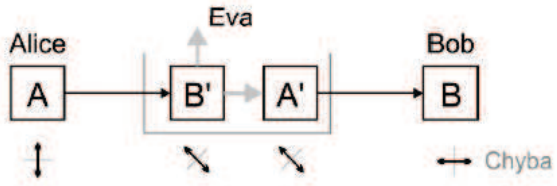
Foton se šikmou polarizací se buď odrazí nebo projde.
Po odrazu bude polarizován vodorovně, po průchodu svisle.

- Logické 0 a 1 jsou kódovány do dvou navzájem kolmých lineárních polarizací ze dvou polarizačních bází
- Báze jsou vůči sobě pootočeny o 45° (viz. Blochova koule)



Systém může fungovat i s jednou polarizací, ale případný útočník pak může s pravděpodobností ½ uhodnout polarizaci a odposlouchávat komunikaci.

Odposlech musí být aktivní – 1 foton nelze rozdělit na menší kvanta ani vytvořit jeho přesnou kopii. Útočník musí foton zachytit, změřit stejným zařízením jako má Bob a rychle ho znovu vyslat stejným zařízením jako má Alice.



Obrana proti odposlechu:

- polarizace pro každý foton se náhodně mění (na obou stranách - nezávisle na sobě)
- po přenosu si jiným kanálem sdělí jaké polarizace v daném kroku použili
- pokud oba použili stejnou bázi – bity si ponechají
- pokud použili různé báze – bity zahodí

- útočník tak má v každém kroku pravděpodobnost ½ , že zvolí bázi špatně
- při volbě špatné báze dojde s pravděpodobností ½ ke změně polarizace a tedy celkem ¼ přijatých bitů bude chybná

Alice a Bob srovnají část přenesených bitů -> zvýšená chybovost = odposlech

- srovnání 128 bitů – pravděpodobnost odhalení odposlechu

P = 1- (1-0,25)128 ~ 1-1,018.10-16 =0,9999999999999998

Protokol BB84

- 1984
- Benett-Brassard
- slouží k bezpečné výměně klíče
- dohodnutý klíč je poté použit pro Vernamovu šifru
- založen na využití Heisenbergova principu neurčitosti ve spojení s polarizačním kódováním
- nejznámější kryptografický protokol využívající kvantové principy
- používán dodnes (s drobnými obměnami)
- využívá dvou kanálů
  - kvantový kanál slouží k přenosu šifrovacího klíče
  - jiný telekomunikační kanál slouží k přenosu šifrované zprávy
  - autentizace na druhém (ne-kvantovém) kanále se neřeší !

- kvantový kanál je realizován optickými vlákny
- vysoké nároky na kvalitu přenosové cesty

- minimální útlum
- malá disperze

Zpráva je přenášena pomocí fotonů s různou polarizací.

Rovina polarizace je pro každý bit volena absolutně náhodně a nezávisle.

V případě, že příjemce zvolí jinou bázi než odesílatel dostane náhodný výsledek s pravděpodobností ½

Čtyři možné roviny polarizace (viz diagram rovin).

Existují i protokoly se 6 nebo 2 polarizacemi.

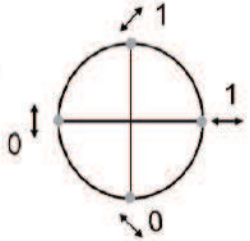


Diagram rovin

K testování polarizace slouží měření ve dvou různých bázích.

Směr polarizace	Hodnota bitu	Báze
→	1	+
↑	0	+
↗	1	×
↖	0	×

1.	1	0	1	1	1	0	0	1	0	0	0	0	1	1	0	0

Strana A generuje náhodné bity.

1.	1	0	1	1	1	0	0	1	0	0	0	0	1	1	0	0
2.	+	+	+	×	×	+	×	+	×	+	+	×	×	+	+	+

Strana A náhodně volí báze.

Strana A kóduje bity do polarizace fotonů a odesílá je příjemci B.

1.	1	0	1	1	1	0	0	1	0	0	0	0	1	1	0	0
2.	+	+	+	×	×	+	×	+	×	+	+	×	×	+	+	+
3.	→	↑	→	↗	↗	↑	↖	→	↖	↑	↑	↖	↗	→	↑	↑

1.	1	0	1	1	1	0	0	1	0	0	0	0	1	1	0	0
2.	+	+	+	×	×	+	×	+	×	+	+	×	×	+	+	+
3.	→	↑	→	↗	↗	↑	↖	→	↖	↑	↑	↖	↗	→	↑	↑
4.	×	×	+	×	+	+	+	+	+	×	×	×	×	+	+	+

Příjemce B náhodně volí báze.

1.	1	0	1	1	1	0	0	1	0	0	0	0	1	1	0	0
2.	+	+	+	×	×	+	×	+	×	+	+	×	×	+	+	+
3.	→	↑	→	↗	↗	↑	↖	→	↖	↑	↑	↖	↗	→	↑	↑
4.	×	×	+	×	+	+	+	+	+	×	×	×	×	+	+	+
5.	↗	↖	→	↗	→	→	→	→	↑	↖	↗	↖	↖	↑	→	↑

Příjemce B měří přijaté fotony v předem zvolených bázích.

1.	1	0	1	1	1	0	0	1	0	0	0	0	1	1	0	0
2.	+	+	+	×	×	+	×	+	×	+	+	×	×	+	+	+
3.	→	↑	→	↗	↗	↑	↖	→	↖	↑	↑	↖	↗	→	↑	↑
4.	×	×	+	×	+	+	+	+	+	×	×	×	×	+	+	+
5.	↗	↖	→	↗	→	→	→	→	↑	↖	↗	↖	↖	↑	→	↑
6.	1	0	1	1	1	1	1	1	0	0	1	0	0	0	1	0

Příjemce B dekóduje bity.

1.	1	0	1	1	1	0	0	1	0	0	0	0	1	1	0	0
2.	+	+	+	×	×	+	×	+	×	+	+	×	×	+	+	+
3.	→	↑	→	↗	↗	↑	↖	→	↖	↑	↑	↖	↗	→	↑	↑
4.	×	×	+	×	+	+	+	+	+	×	×	×	×	+	+	+
5.	↗	↖	→	↗	→	→	→	→	↑	↖	↗	↖	↖	↑	→	↑
6.	1	0	1	1	1	1	1	1	0	0	1	0	0	0	1	0
7.			ok	ok				ok				ok				ok

Strany A a B se veřejně domluví, na kterých bázích se shodli.

1.	1	0	1	1	1	0	0	1	0	0	0	0	1	1	0	0
2.	+	+	+	×	×	+	×	+	×	+	+	×	×	+	+	+
3.	→	↑	→	↗	↗	↑	↖	→	↖	↑	↑	↖	↗	→	↑	↑
4.	×	×	+	×	+	+	+	+	+	×	×	×	×	+	+	+
5.	↗	↖	→	↗	→	→	→	→	↑	↖	↗	↖	↖	↑	→	↑
6.	1	0	1	1	1	1	1	1	0	0	1	0	0	0	1	0
7.			ok	ok				ok				ok				ok
8.			1									0				

Strany A a B obětují libovolné (dohodnuté) množství bitů na detekci odposlechu.

1.	1	0	1	1	1	0	0	1	0	0	0	0	1	1	0	0
2.	+	+	+	×	×	+	×	+	×	+	+	×	×	+	+	+
3.	→	↑	→	↗	↗	↑	↖	→	↖	↑	↑	↖	↗	→	↑	↑
4.	×	×	+	×	+	+	+	+	+	×	×	×	×	+	+	+
5.	↗	↖	→	↗	→	→	→	→	↑	↖	↗	↖	↖	↑	→	↑
6.	1	0	1	1	1	1	1	1	0	0	1	0	0	0	1	0
7.			ok	ok				ok				ok				ok
8.			1									0				
9.			ok									ok				

Pokud je většina kontrolovaných bitů shodná, znamená to, že nedošlo k odposlechu.

1.	1	0	1	1	1	0	0	1	0	0	0	0	1	1	0	0
2.	+	+	+	×	×	+	×	+	×	+	+	×	×	+	+	+
3.	→	↑	→	↗	↗	↑	↖	→	↖	↑	↑	↖	↗	→	↑	↑
4.	×	×	+	×	+	+	+	+	+	×	×	×	×	+	+	+
5.	↗	↖	→	↗	→	→	→	→	↑	↖	↗	↖	↖	↑	→	↑
6.	1	0	1	1	1	1	1	1	0	0	1	0	0	0	1	0
7.			ok	ok				ok				ok				ok
8.			1									0				
9.			ok									ok				
10.				1				1								0

Zbylé bity tvoří klíč, který bude použit pro vlastní šifrování (v tomto případě 110).

Dnes se navíc provádí tzv. *zesílení soukromí* (*privacy amplification*).
Z množství chyb detekovaných během přenosu se určí jak moc byl kanál odposloucháván resp. kolik informace z něj mohl útočník získat.

Strany A a B pak předem definovaným způsobem převedou dohodnutý klíč na jiný kratší např. pomocí hashovací funkce.

Cílem je minimalizovat útočníkovi informace o klíči.