20. Programovací techniky: Abstraktní datový typ, jeho specifikace a implementace. Datový typ zásobník, fronta, tabulka, strom, seznam. Základní algoritmy řazení a vyhledávání. Složitost algoritmů.

Abstraktní datový typ

Pokud nabízíme omezené rozhraní datového typu pomocí specifikované sady operací, nazýváme takový typ abstraktní datový typ (ADT). Vzhledem k tomu, že odpovídající „konkrétní“ datový typ pak není uživateli dostupný a jedinou možností jak s hodnotami pracovat je právě prostřednictvím nabízených operací, získáme na tento typ mnohem abstraktnější pohled.
Další výhodou použití abstraktních datových typů, a tedy především oddělení rozhraní datového typu od jeho implementace, je to, že můžeme vytvořit různé implementace téhož datového typu, aniž se tato změna projeví při jeho používání. Pro vytváření různých implementací můžeme mít více důvodů, např.:

- Implementace se mohou od sebe lišit efektivitou jednotlivých operací; uživatel si zvolí tu implementaci, která je z hlediska efektivity v dané situaci nejvýhodnější.
- Může jít o různé vývojové verze téže implementace.
- Může jít o implementace poskytnuté různými autory.

V prostředí objektově orientovaných programovacích jazyků jsou pro definici abstraktních datových typů k dispozici třídy a v některých jazycích máme dokonce přímo konstrukce, umožňující definovat abstraktní rozhraní.

V případě jazyka Java bychom abstraktní datový typ Time reprezentující čas mohli definovat pomocí třídy takto:

```
public class Time {
   public Time(int hours, int minutes) {
      this.hours = hours; this.minutes = minutes;
   }

   public int getHours() { return hours; }

   public int getMinutes() { return minutes; }

   public void increment(int delta) {
      int newMins = minutes + delta;
      minutes = newMins % 60;
      hours  += newMins / 60;
   }
   private int hours;
   private int minutes;
}
```

Povšimněte si, že třída Time obsahuje ve svých veřejných členech pouze metody getHours(), getMinutes() a increment(). Konstruktor createTime() je nahrazen skutečným konstruktorem a samotná hodnota je vnitřně reprezentovaná dvojicí proměnných hours a minutes pro počet hodin a minut. Rozdíl ve vnitřní reprezentaci proti předchozí ukázce se navenek nijak viditelně neprojeví, neboť uživatel pracuje pouze s veřejnými metodami objektu.

Abstraktní datové typy můžeme použít pro nejrůznější účely a jejich struktura se bude lišit podle konkrétní aplikace. Ovšem při vytváření programů se s některými abstrakcemi setkáváme relativně často - například s různými kolekcemi hodnot, grafovými strukturami nebo vyhledávacími tabulkami. V následujících částech této kapitoly se budeme těmito často se vyskytujícími abstrakcemi zabývat podrobněji. Ukážeme si, jakým způsobem je můžeme implementovat, případně jak můžeme využít nabízené standardní knihovny, které jejich podporu již často obsahují.

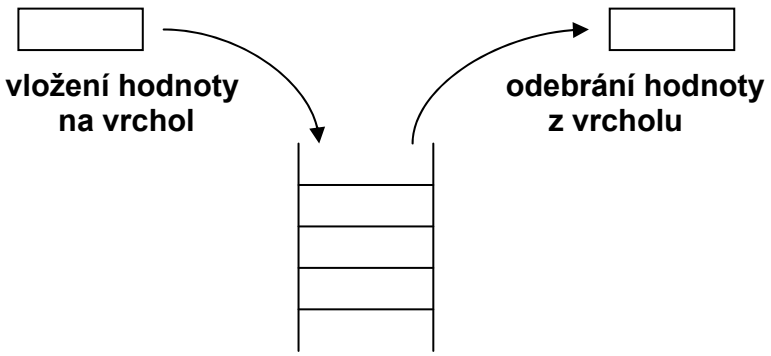
K základním abstraktním datovým typům můžeme zařadit následující konstrukce (v závorkách jsou uvedeny i obvykle používané anglické termíny):

- zásobník (stack),
- fronta (queue),
- seznam (list),
- množina (set),
- strom (tree),
- zobrazení (map).

Datové struktury

Zásobník (Stack)

- Je to dynamická datová struktura, umožňující vkládání a odebírání hodnot, přičemž naposledy vložená hodnota se odebere jako první
- Je to paměť typu LIFO (zkratka z angl. Last-In First-Out, poslední dovnitř, první ven)



- Základní operace:
  - vložení hodnoty na vrchol zásobníku
  - odebrání hodnoty z vrcholu zásobníku
  - test na prázdnost zásobníku
- Příklad třídy realizující zásobník znaků:

```
class ZasobnikZnaku {
  public ZasobnikZnaku() {...}
  public void vloz(char z) { ... }
  public char odeber() { ... }
  public boolean jePrazdny() { ... }
  ...
}
```

- Poznámky

- v angličtině se operace nad zásobníkem obvykle jmenují *push*, *pop* a *isEmpty*
- pro zásobník může být definována ještě operace čtení hodnoty z vrcholu zásobníku bez jejího odebrání (v angl. *top* či *peek*)

Implementace zásobníku polem

```
public void vloz(char z) {
    if ( vrchol==MAX_INDEX )
      throw new RuntimeException( "plný zásobník" );
    pole[++vrchol] = z;
  }
  public char odeber() {
    if ( vrchol<0 )
      throw new RuntimeException( "prázdný zásobník" );
    return pole[vrchol--];
  }
  public boolean jePrazdny() {
    return vrchol<0;
  }
}
```

Implementace zásobníku spojovým seznamem

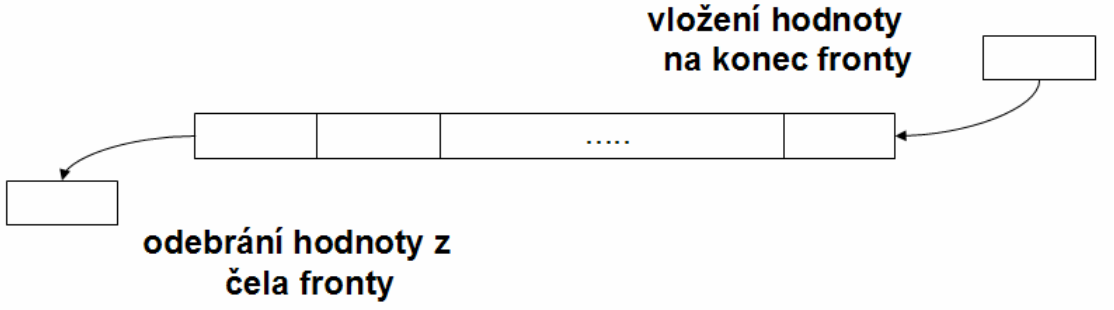
```
class Prvek {
  char hodn;
  Prvek dalsi;
  public Prvek(char h, Prvek p) {
    hodn = h; dalsi = p;
  }
}
class ZasobnikZnaku {
  private Prvek vrchol;
  public ZasobnikZnaku() {
    vrchol = null;
  }
}
public void vloz(char z) {
    vrchol = new Prvek(z, vrchol);
  }

  public char odeber() {
    if (vrchol==null)
      throw new RuntimeException( "prazdny zasobnik" );
    char vysl = vrchol.hodn;
    vrchol = vrchol.dalsi;
    return vysl;
  }

  public boolean jePrazdny() {
    return vrchol==null;
  }
}
```

Fronta (Queue)

- Je to datová struktura v níž se hodnoty odebírají v tom pořadí, v jakém byly vloženy
- Je to paměť typu FIFO (zkratka z angl. First-In First-Out, první dovnitř, první ven)



- Implementace fronty:
  - pomocí pole
  - pomocí spojového seznamu

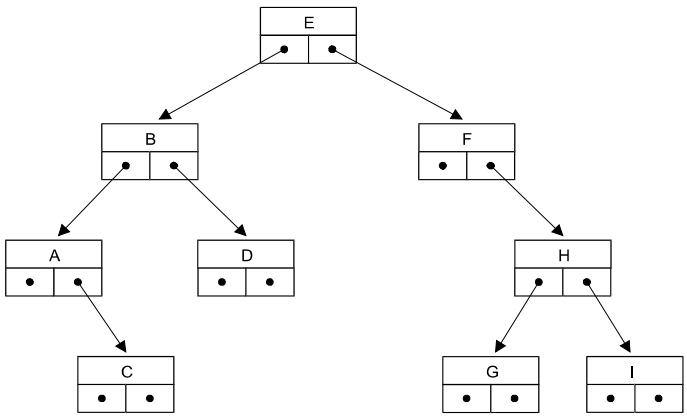
Implementace fronty spojovým seznamem

```
class PrvekFronty {
  PrvekFronty dalsi;
  int hodn;
}

public class FrontaCisel {
  private PrvekFronty celo;
  private PrvekFronty volny;
public FrontaCisel() {
    celo = new PrvekFronty();
    volny = celo;
  }
  public void vloz(int x) {
    volny.hodn = x;
    volny.dalsi = new PrvekFronty();
    volny = volny.dalsi;
  }
  public int odeber() {
    if (jePrazdna())
      throw new RuntimeException( " prazdna fronta" );
    int vysl = celo.hodn;
    celo = celo.dalsi;
    return vysl;
  }
  public boolean jePrazdna() {
    return celo == volny;
  }
}
```

Stromy

- Lineární spojová struktura (spojový seznam)
  - každý prvek má nanejvýš jednoho následníka
- Nelineární spojová struktura (strom):
  - každý prvek může mít více následníků
- Binární strom: každý prvek (uzel) má nanejvýš dva následníky



- Některé pojmy: kořen stromu, levý podstrom, pravý podstrom, list

Pole jako tabulka

- Pole lze použít též pro realizaci tabulky (zobrazení), která hodnotám typu indexu (v jazyku Java to je pouze interval celých čísel počínaje nulou) přiřazuje hodnoty nějakého typu
- Příklad: přečíst řadu čísel zakončených nulou a vypsat tabulku četnosti čísel od 1do 100 (ostatní čísla ignorovat). Tabulka četnosti bude pole 100 prvků typu *int*, počet výskytů čísla *x*, kde $1 \leq x \leq 100$, bude hodnotou prvku s indexem *x*-1

Seznam

- obsahuje prvky uspořádané.

Seznam (list) je datová struktura, která umožňuje kromě základních operací nad obecnou kolekcí také přistupovat k libovolnému prvku na základě pořadového čísla (indexu) a vkládat a rušit prvky na libovolné pozici. Představuje vlastně nejobecnější kolekci, pomocí které můžeme implementovat kolekce další včetně zásobníku a fronty. Z tohoto důvodu jsou v mnoha programovacích jazycích seznamy k dispozici jako součást standardních knihoven (Java, C++, C#).
Více na http://www.cs.vsb.cz/benes/vyuka/pte/texty/adt/ch01s04.html

Časová složitost algoritmů

- Důležitou vlastností algoritmu je časová náročnost výpočtů provedené podle daného algoritmu
- Ta se nezískává měřením doby výpočtu pro různá data, ale analýzou algoritmu, jejímž výsledkem je časová složitost algoritmu
- Časová složitost algoritmu vyjadřuje závislost času potřebného pro provedení výpočtu na rozsahu (velikosti) vstupních dat
- Čas se však neměří sekundách, ale počtem provedených operací, přičemž trvání každé operace se chápe jako bezrozměrná jednotka
- Příklad: součet prvků pole

```
static int soucet(int[] pole) {
  int s = 0;
  for (int i=0; i<pole.length; i++ ) s = s + pole[i] ;
  return s;
}
```

- Považujme za operace podtržené konstrukce, pak časová složitost je:

$C(n) = 2 + (n+1) + n + n = 3 + 3n$

kde *n* je počet prvků pole.

- Doba výpočtu obvykle nezávisí jen na rozsahu vstupních dat, ale též na konkrétních hodnotách
- Obecně proto rozlišujeme časovou složitost v nejlepším, nejhorším a průměrném případě
- Příklad: sekvenční hledání prvku pole s danou hodnotou

```
static int hledej(int[] pole, int x) {
  for (int i=0; i<pole.length; i++ )
      if ( x==pole[i] ) return i;
  return -1;
}
```

- Analýza:
  - nejlepší případ: první prvek má hodnotu *x*
    $Cmin(n) = 3$
  - nejhorší případ: žádný prvek nemá hodnotu *x*
    $Cmax(n) = 1 + (n+1) + n + n = 2 + 3n$

– průměrný případ
$Cprum(n) = 2.5 + 1.5n$

- Přesné určení počtu operací při analýze složitosti algoritmu bývá velmi složité
- Zvlášť komplikované, ba i nemožné, bývá určení počtu operací v průměrném případě; proto se většinou omezujeme jen na analýzu nejhoršího případu
- Zpravidla nás nezajímají konkrétní počty operací pro různé rozsahy vstupních dat *n*, ale tendence jejich růstu při zvětšujícím se *n*
- Pro tento účel lze výrazy udávající složitost zjednodušit: stačí uvažovat pouze složky s nejvyšším řádem růstu a i u nich lze zanedbat multiplikativní konstanty
- Příklad: řád růstu časové složitosti předchozích algoritmů je *n* (časová složitost je lineární)
- Časovou složitost vyjadřujeme pomocí tzv. asymptotické notace:

O dvou funkcích *f* a *g* definovaných na množině přirozených čísel a s nezáporným oborem hodnot říkáme, že *f* roste řádově nejvýš tak rychle, jako *g* a píšeme

$$f(n) = O(g(n))$$

pokud existují přirozená čísla K a $n1$ tak, že platí

$$f(n) \le K.g(n) \quad \text{pro všechna } n > n1$$

Algoritmy řazení pole

- Algoritmy řazení pole jsou algoritmy, které přeskupí prvky pole tak, aby upravené pole bylo seřazené

- Pole *p* je vzestupně seřazené, jestliže platí

$p[i\text{-}1] <= p[i]$ pro $i = 1$ … počet prvků pole – 1

- Pole *p* je sestupně seřazené, jestliže platí

$p[i\text{-}1] >= p[i]$ pro $i = 1$ … počet prvků pole – 1

- Principy některých algoritmů řazení ukažme na řazení pole prvků typu *int*

Následující metody vzestupně řadí všechny prvky pole daného parametrem:
bubbleSort( )
selectSort( )
insertSort( )
mergeSort( )

Řazení zaměňováním (Bubble Sort)

- Při řazení zaměňováním postupně porovnáváme sousední prvky a pokud jejich hodnoty nejsou v požadované relaci, vyměníme je; to je třeba provést několikrát
- Hrubé řešení:

```
for ( n=a.length-1; n>0; n-- )
  for ( i=0; i<n; i++ )
    if ( a[i]>a[i+1] ) "vyměň a[i] a a[i+1]"
```

- Podrobné řešení :

```
static void bubbleSort(int[] a) {
  int pom, n, i;
  for ( n=a.length-1; n>0; n-- )
    for ( i=0; i<n; i++ )
      if ( a[i]>a[i+1] ) {
        pom = a[i]; a[i] = a[i+1]; a[i+1] = pom;
      }
}
```

- Časová složitost je O($n2$)

Řazení výběrem (Select Sort)

- Při řazení výběrem se opakovaně hledá nejmenší prvek
- Hrubé řešení:

```
for (i=0; i<a.length-1; i++) {
  "najdi nejmenší prvek mezi a[i] až a[a.length-1]";
  "vyměň hodnotu nalezeného prvku s a[i]";
}
```

- Podrobné řešení:

```
public static void selectSort(int[] a) {
    int i, j, imin, pom;
    for (i=0; i<a.length-1; i++) {
      imin = i;
      for (j=i+1; j<a.length; j++)
          if (a[j]<a[imin]) imin = j;
      if (imin!=i) {
        pom = a[imin]; a[imin] = a[i]; a[i] = pom;
      }
    }
  }
```

- Časová složitost algoritmu SelectSort: O(*n*2)

Řazení vkládáním (Insert Sort)

- Pole lze seřadit opakovaným vkládání prvku do seřazeného úseku pole
- Hrubé řešení:

```
for (n=1; n<a.length; n++) {
  " úsek pole od a[0] do a[n-1] je seřazen "
  " vlož do tohoto úseku délky n hodnotu a[n] "
}
```

- Podrobné řešení:

```
 private static void vloz(int[] a, int n, int x) {
   int i;
   for (i=n-1; i>=0 && a[i]>x; i--) a[i+1]=a[i]; // odsun
   a[i+1] = x; // vlozeni
 }

 public static void insertSort(int[] a) {
   for (int n=1; n<a.length ; n++)
     vloz(a, n, a[n]);
 }
```

- Časová složitost algoritmu InsertSort: O(*n*2)

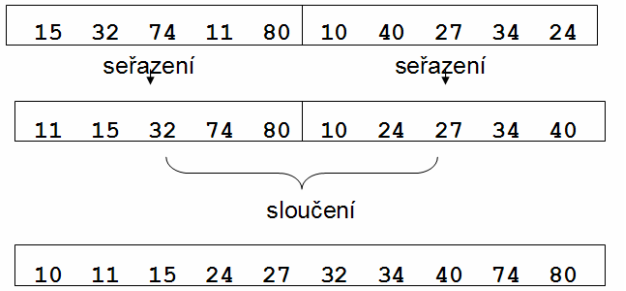
Řazení slučováním (Merge Sort)

- Problém slučování lze obecně formulovat takto:
  - ze dvou seřazených (monotonních) posloupností *a* a *b* máme vytvořit novou posloupnost obsahující všechny prvky z *a* i *b*, která je rovněž seřazená
- Příklad:

a: 2 3 6 8 10 34
b: 3 7 12 13 55
výsledek: 2 3 3 6 7 8 10 12 13 34 55

- Princip slučování:
  - postupně porovnáváme prvky zdrojových posloupností a do výsledné posloupnosti přesouváme menší z nich
  - nakonec zkopírujeme do výsledné posloupnosti zbytek první nebo druhé posloupnosti
- Efektivnější algoritmy řazení mají časovou složitost O(*n log a*)

- Jedním z nich je algoritmus řazení slučováním (MergeSort), který je založen na opakovaném slučování seřazených úseků do úseků větší délky
- Lze jej popsat rekurzívně:
  - řazený úsek pole rozděl na dvě části
  - seřaď levý úsek a pravý úsek
  - přepiš řazený úsek pole sloučením levého a pravého úseku



- Rekurzivní funkce řazení úseku pole:

```
private static void mergeSort( int[] a, int[] pom,
                               int prvni, int posl ) {
   if ( prvni<posl ) {
     int stred = (prvni+posl)/2;
     mergeSort(a, pom, prvni, stred);
     mergeSort(a, pom, stred+1, posl);
     merge(a, pom, prvni, stred+1, posl);
     for (int i=prvni; i<=posl; i++) a[i] = pom[i];
   }
 }
```

- Výsledná funkce:

```
public static void mergeSort(int[] a) {
   int[] pom = new int[a.length];
   mergeSort(a, pom, 0, a.length-1);
 }
```

- Rekurzívní MergeSort se volá rekurzívně tak dlouho, dokud délka úseku pole není 1; pak začne slučování sousedních úseků do dvakrát většího úseku v pomocném poli, který je třeba zkopírovat do původního pole
- Rozdělení pole na úseky, které se postupně slučují, je dáno postupným půlením úseků pole shora dolů.
- Nerekurzívní (iterační) algoritmus MersgeSort postupuje zdola nahoru:
- pole *a* se rozdělí na dvojice úseků délky 1, které se sloučí do seřazených úseků délky 2 v pomocném poli *pom*
- dvojice sousedních seřazených úseků délky 2 v poli *pom* se sloučí do seřazených úseků délky 4 v poli *a*
- dvojice sousedních úseků délky 4 v poli *a* se sloučí do seřazených úseků délky 8 v poli *pom*
- atd.
- tento postup se opakuje, pokud délka úseku je menší než velikost pole
- skončí-li slučování tak, že výsledek je v pomocném poli *pom*, je třeba jej zkopírovat do původního pole *a*

Zdroje:

[1]Algoritmizace – přednášky - http://service.felk.cvut.cz/courses/X36ALG/alg_prednasky/

[2]Abstraktní datové typy - http://www.cs.vsb.cz/benes/vyuka/pte/texty/adt/index.html