ADT Seznam

dynamická množina

prvky vkládáme a vyjímáme na libovolné pozici (místě)
v jejich posloupnosti (sekvenci)

zásobník a fronta vs. seznam

Příklad

řádek znaků na obrazovce (*model*)

ab<u>c</u>d
↑ - okamžitá, nynější pozice v seznamu

vlož *x* (za okamžitou pozici):

ab<u>x</u>cd

vlož *y* (za okamžitou pozici):

abx<u>y</u>cd

posun na další prvek:

abxy<u>c</u>d

vyber prvek (na okamžité pozici):

abxy<u>d</u>

Rozhraní ADT seznam

```
// rozhrani ADT seznam
Seznam()                    //  vytvoření prázdného seznamu
boolean jePrazdny( )        //  test je-li seznam prázdný
void tiskSeznamu( )         //  tisk prvků seznamu
void naZacatek( )           //  nastavení okamžité pozice na
                            // první prvek
boolean jePosledni( )       //  test je-li okamžitá pozice
                            // nastavena na poslední prvek
void naDalsiPrvek( )        //  nastavení okamžité pozice na
                            //  pozici následujícího prvku
int ctiKlic( )              //  přečti klíč prvku na okamžité
                            //  pozici
void vloz(int i)            //  vlož prvek
int vyber( )                //  vyber prvek
```

odstranění všech výskytů prvku **`int x`** v **`Seznamu s`**

```
if(!s.jePrazdny()) {
  s.naZacatek();
  boolean konec=false;
  while (!konec) {
    konec = s.jePosledni();
    if(s.ctiKlic() == x)
      s.vyber();
    else
      s.naDalsiPrvek();
  }
}
```

neznáme implementaci !

Implementace ADT seznam

polem - operace vložení a výběru uvnitř seznamu jsou ***O(n)***

spojovou strukturou - všechny operace vložení a výběru jsou ***O(1)***

http://java.sun.com/javase/6/docs/api/java/util/List.html

http://java.sun.com/javase/6/docs/api/java/util/ArrayList.html

http://java.sun.com/javase/6/docs/api/java/util/LinkedList.html

List (interface)	Order is the most important feature of a **List**; it promises to maintain elements in a particular sequence. **List** adds a number of methods to **Collection** that allow insertion and removal of elements in the middle of a **List**. (This is recommended only for a **LinkedList**.) A **List** will produce a **ListIterator**, and using this you can traverse the **List** in both directions, as well as insert and remove elements in the middle of the **List**.
ArrayList*	A **List** implemented with an array. Allows rapid random access to elements, but is slow when inserting and removing elements from the middle of a list. **ListIterator** should be used only for back-and-forth traversal of an **ArrayList**, but not for inserting and removing elements, which is expensive compared to **LinkedList**.
LinkedList	Provides optimal sequential access, with inexpensive insertions and deletions from the middle of the **List**. Relatively slow for random access. (Use **ArrayList** instead.) Also has **addFirst( )**, **addLast( )**, **getFirst( )**, **getLast( )**, **removeFirst( )**, and **removeLast( )** (which are not defined in any interfaces or base classes) to allow it to be used as a stack, a queue, and a deque.

Bruce Eckel: Thinking in Java. Second Edition. President, MindView, Inc.

http://www.planetpdf.com/developer/article.asp?ContentID=6632

Implementace spojovou strukturou

prvek seznamu

```
class Prvek {
  ... klic;
  Prvek dalsi;
}
```

seznam reprezentován referenční proměnnou **`prvni`** typu **`Prvek`**

inicializace: **`prvni = null;`**

test je-li prázdný: **`if(prvni == null);`**

okamžitá pozici reprezentována referenční proměnnou **`nynejsi`** (pozice), typu **`Prvek`**

nastaveni na začátek: **`nynejsi = prvni;`**

test poslední pozice: **`if (nynejsi.dalsi == null);`**

posun na další prvek: **`nynejsi = nynejsi.dalsi;`**

procházení seznamem:

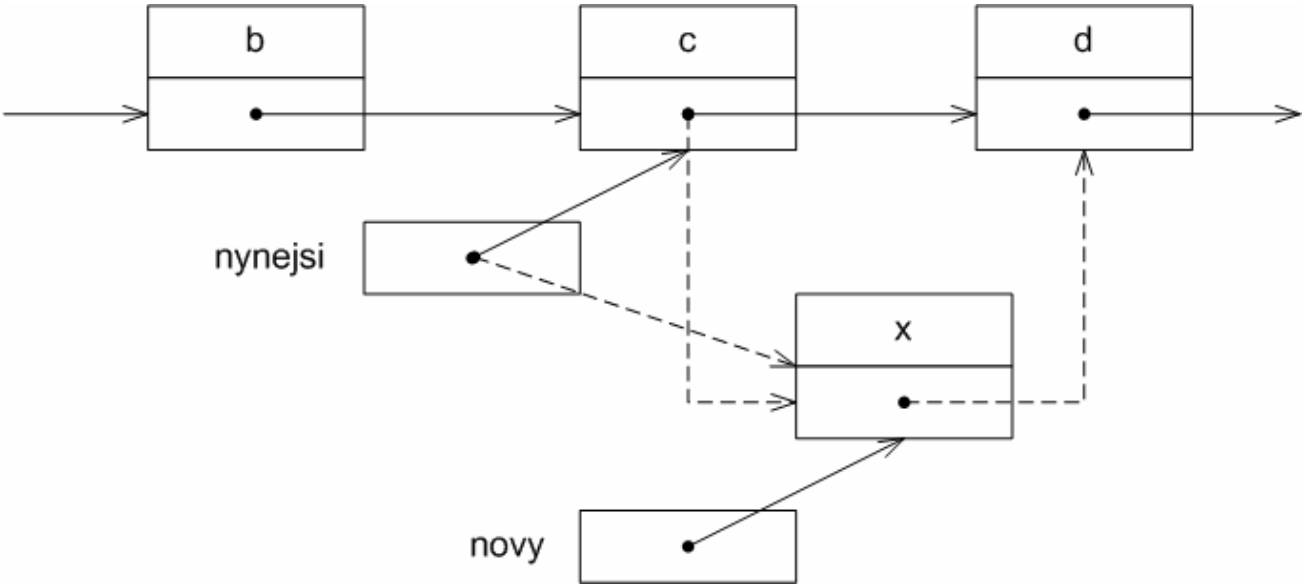
```
for (nynejsi = prvni; nynejsi != null;
        nynejsi = nynejsi.dalsi)
```

nalezení prvku s určitou hodnotou:

```
if(nynejsi.klic = hodnota);
```

vlož prvek za nynejsi:

```
if (prvni == null) {
  prvni = novy;
  prvni.dalsi = null;
}
else {
  novy.dalsi = nynejsi.dalsi;
  nynejsi.dalsi = novy;
}
nynejsi = novy;
```



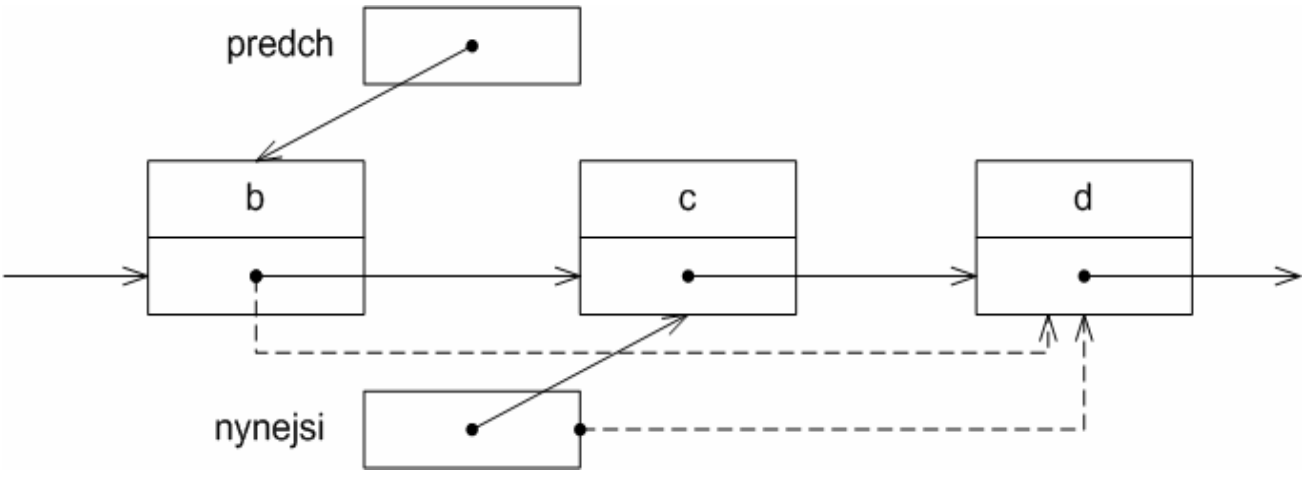
pro výběr (nynějšího) prvku musíme znát odkaz na předcházející prvek !

predch udržuje odkaz na prvek předcházející okamžité pozici

je-li okamžitou pozicí první prvek má proměnná **predch** hodnotu **null**

vyber nynejsi:

```
if(predch == null) {
  prvni = nynejsi.dalsi;
  nynejsi = prvni;
}
else {
  predch.dalsi = nynejsi.dalsi;
  if (nynejsi.dalsi == null) {   //byl poslední
    nynejsi = prvni;
    predch = null;
  }
  else
    nynejsi = nynejsi.dalsi;
```



Poznámka: v případě, že jsme odstranili poslední prvek, nastavíme okamžitou pozici v seznamu na první prvek

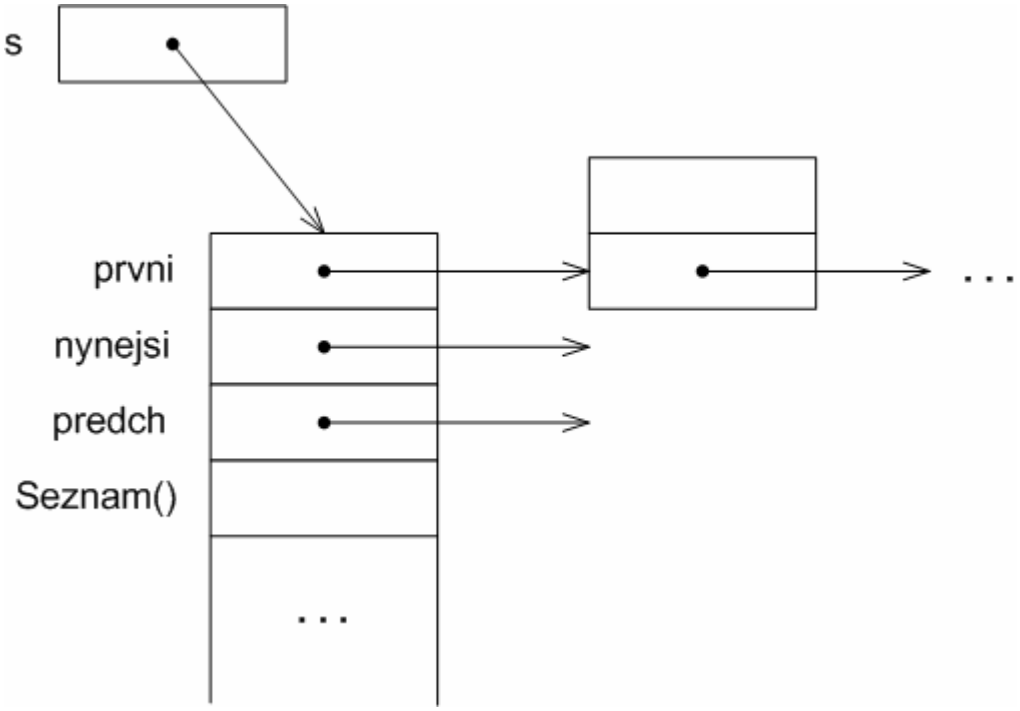
Poznámka: uvedená implementace operace pro vložení prvku neumožňuje vložit do neprázdného seznamu prvek na první místo

- můžeme doplnit operace o operaci *vlož na začátek*
- s pomocí proměnné **`predch`** můžeme napsat implementaci obecnější operace *vlož před*, která vloží prvek před okamžitou pozici

Poznámka: pozorujte nutnost odlišnosti vybíráme-li první prvek a poslední prvek

http://www.cs.usask.ca/resources/tutorials/csconcepts/index.html

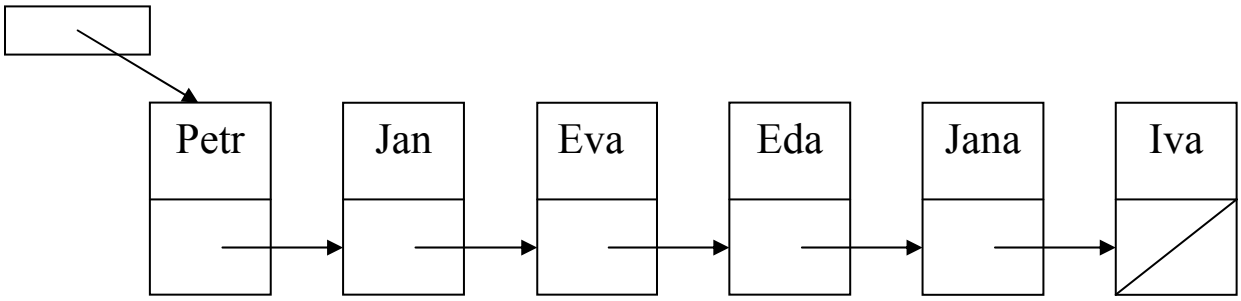
Referenční proměnná `s` třídy `Seznam` a její instance



Úkol (*motivační*)

Seznam hochů a dívek vytiskněte tak, aby v levém sloupci byli pod sebou jména všech hochů a v pravém sloupci jména všech dívek.

V seznamu je stejný počet dívek a hochů a byli do něj vloženi náhodně.



Petr	Eva
Jan	Jana
Eda	Iva

Algoritmus (*myšlenka)*

- najdi prvního hocha a vytiskni jmeno
- najdi první dívku a vytiskni jmeno
- přejdi na nový řádek

- najdi dalšího hocha a vytiskni jmeno
- najdi další dívku a vytiskni jmeno
- přejdi na nový řádek

- skonči, není-li v seznamu dalsi hoch ani dalsi divka

Implementace ?

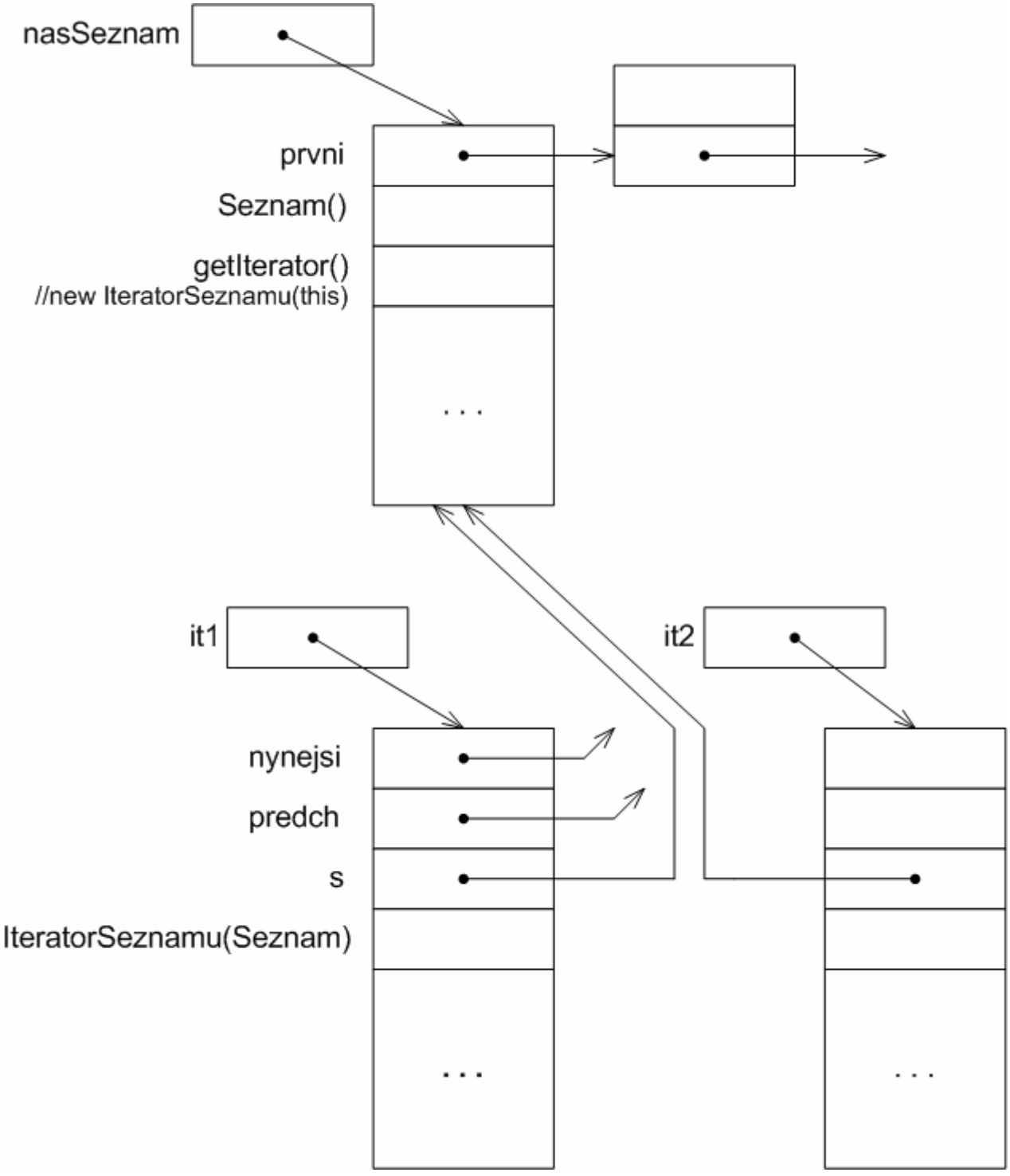
oddělíme
- data objektu implementujícího **seznam** - proměnná **`prvni`**
- práci s ním, proměnné **`nynejsi`** a **`predch`**, implementuje **iterátor**

objekty třídy **`Seznam`** metodou **`getIterator()`**, získají referenci na své objekty třídy **`IteratorSeznamu`**

```
class Seznam {
  private Prvek prvni;

  Seznam();
  ...
  IteratorSeznamu getIterator() {
    return new IteratorSeznamu(this);
  }
  ...
}

class IteratorSeznamu {
  private Prvek nynejsi;
  private Prvek predch;
  private Seznam s;

  IteratorSeznamu(Seznam s) {
    this.s = s;
    naZacatek();
  }
  ...
}
```

každý iterátor spravuje vlastní okamžitou pozici daného seznamu

```
public static void main (...) {
   ...
   Seznam nasSeznam = new Seznam();
   IteratorSeznamu it1 = nasSeznam.getIterator();
   IteratorSeznamu it2 = nasSeznam.getIterator();
```



```
class Seznam {       // rozhraní
  Seznam() {
  Prvek getPrvni()
  void setPrvni(Prvek)
  boolean jePrazdny()
  IteratorSeznamu getIterator()
  void tiskSeznamu ()
}


class IteratorSeznamu {        // rozhraní
    IteratorSeznamu(Seznam)
    void naZacatek()
    boolean jePosledni()
    void naDalsiPrvek()
    int ctiKlic()
    void vloz(int)
    int vyber
}
```

Implementace metod tříd **Seznam** a **IteratorSeznamu**

```
class Prvek {
  int klic;
  Prvek dalsi;

  Prvek (int klic) {
    this.klic = klic;
  }
  void tiskPrvku() {
    System.out.print(klic + " ");
  }
}

class Seznam {
  private Prvek prvni;

  Seznam() {
    prvni = null;
  }
  Prvek getPrvni() {
    return prvni;
  }
  void setPrvni(Prvek ref) {
    prvni = ref;
  }
  boolean jePrazdny() {
    return (prvni == null);
  }
  IteratorSeznamu getIterator() {
    return new IteratorSeznamu(this);
  }
  void tiskSeznamu() {
    for (Prvek x = prvni; x != null; x = x.dalsi)
      x.tiskPrvku();
    System.out.println("");
  }
}
```

```
class IteratorSeznamu {

  private Prvek nynejsi;
  private Prvek predch;
  private Seznam s;

  IteratorSeznamu(Seznam s) {
    this.s = s;
    naZacatek();
  }
  void naZacatek() {
    nynejsi = s.getPrvni();
    predch = null;
  }
  boolean jePosledni() {
    return (nynejsi.dalsi == null);
  }
  void naDalsiPrvek() {
    predch = nynejsi;
    nynejsi = nynejsi.dalsi;
  }
  int ctiKlic() {
    return nynejsi.klic;
  }
  void vloz(int i) {
    Prvek novy = new Prvek(i);
    if (s.jePrazdny()) {
      s.setPrvni(novy);
      nynejsi = novy;
    }
    else {
      novy.dalsi = nynejsi.dalsi;
      nynejsi.dalsi = novy;
      naDalsiPrvek();
    }
  }
```

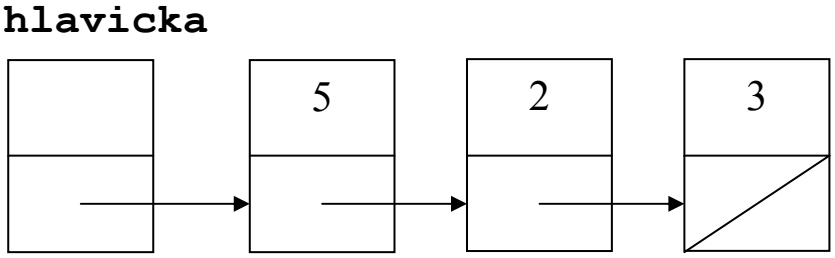
```
  int vyber() {
    int i = nynejsi.klic;
    if(predch == null) {
      s.setPrvni(nynejsi.dalsi);
      naZacatek();
    }
    else {
      predch.dalsi = nynejsi.dalsi;
      if (jePosledni())
        naZacatek();
      else
        nynejsi = nynejsi.dalsi;
    }
    return i;
  }
}
```

opět v implementaci některých operací musíme rozlišovat, je-li okamžitá pozice první resp. poslední prvek

řešení – hlavička seznamu

Hlavička seznamu

odkaz na první prvek seznamu je v položce **`dalsi`** v zdánlivém (dummy) prvku seznamu - v hlavičce



`hlavicka` nahradí proměnnou **`prvni`**

třída **`Seznam`**

inicializace: **`hlavicka = new Prvek();`**

test je-li prázdný: **`if (hlavicka.dalsi == null)`**

`getHlavicka()` bude vracet referenci na hlavičku seznamu

třída **`IteratorSeznamu`**

nastavení na začátek:

```
if(s.jePrazdny()) {
  nynejsi = s.getHlavicka();
  predch = null;
}
else {
  nynejsi = s.getHlavicka().dalsi;
  predch = s.getHlavicka();
}
```

vlož prvek za nynejsi:

```
novy.dalsi = nynejsi.dalsi;
nynejsi.dalsi = novy;
naDalsiPrvek(); //opravit v eKnize
```

vyber nynejsi:

```
predch.dalsi = nynejsi.dalsi;
if (jePosledni())
  naZacatek();
else
  nynejsi = nynejsi.dalsi;
```

Kruhový spojový seznam

v položce **`dalsi`** posledního prvku namísto hodnoty **`null`** uchovávame odkaz na první prvek

Obousměrný spojový seznam

```
class Prvek {
   ...
   Prvek dalsi;
   Prvek predch;
}
```

Koncový prvek

konec seznamu - objekt **`z`** třídy **`Prvek`**, kterého položka **`dalsi`** odkazuje na sebe

inicializace:

```
hlavicka = new Prvek();
z = new Prvek();
hlavicka.dalsi = z;
z.dalsi = z;
```

http://www.cs.usask.ca/resources/tutorials/csconcepts/index.html

Implementace spojového seznamu pomocí paralelních polí

```
char[] klic = new char[max];
int[] dalsi = new int[max];
```

prvek seznamu – dvojice prvků se stejným indexem

`dalsi[i]` - ukazatel na následující prvek seznamu
`dalsi[i] == -1 -` poslední prvek

Příklad

```
int jmeno = 3;
klic[3] = 'P';  dalsi[3] = [4];
klic[4] = 'A';  dalsi[4] = [7];
klic[7] = 'V';  dalsi[7] = [2];
klic[2] = 'E';  dalsi[2] = [6];
klic[6] = 'L';  dalsi[6] = [-1];


int jinejmeno = 9;
klic[9] = 'D';  dalsi[9] = ...;
```

správa volných prvků

zásobník všech volných prvků
-spojené prvky pole **dalsi**
-vrchol v proměnné **volne**

```
int volne;
```

přidělení indexu pro další prvek seznamu

```
int pridelIndex() {
  if (volne == -1)
    "neni volny index"
  else {
    int i = volny;
    volny = dalsi[i];
    return i;
}
```

vrácení indexu mezi volné

```
void uvolniIndex(i) {
  dalsi[i] = volne;
  volne = i;
}
```

Poznámka: pole je dobrým modelem hlavní paměti počítače

Rekurze

Pes jitrničku sežral...

Pes jitrničku sežral
docela maličkou,
spatřil ho při tom kuchař,
klepl ho paličkou.

Plakali všichni psové
kopali jemu hrob,
na desce mramorové
stál nápis těchto slov:

něco částečně složeno nebo definováno pomocí sebe samého

písnička ≡ „Pes jitrničku … těchto slov:“ + písnička

```
String Pisnicka( ) {
  return ("Pes jitrničku … těchto slov" +
Pisnicka( ));
}
```

„zpívání“ s x opakováními

```
class Pisnicka {


  public static void main(String[] args) {
    int x = Integer.parseInt(args[0]);
    System.out.println(Pisnicka(x));
  }

  static String Pisnicka(int x) {
    if (x == 0)
      return "";
    else
      return ("Pes jitrnicku ... techto slov:" +
                Pisnicka(--x));
  }
}
```

Co je ***něco*** ?

- **funkce:**

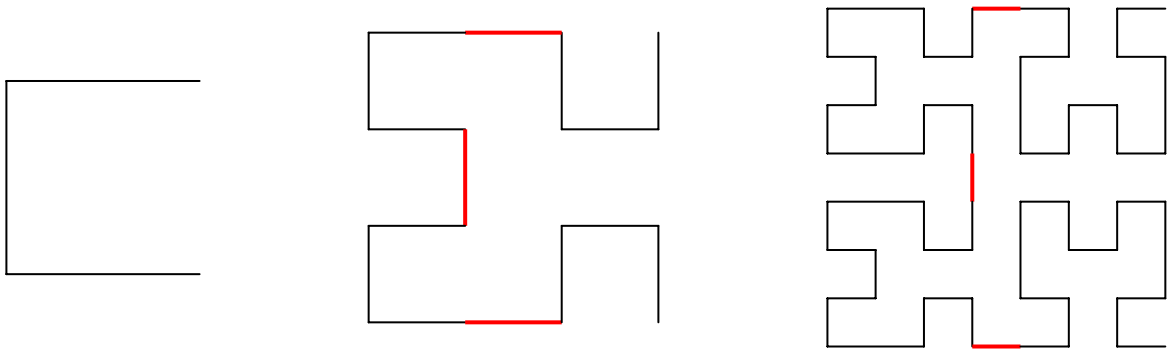
faktoriál
a) 0! = 1,
b) n! = n.(n-1)!, pro n >0

- **množiny:**

seznam
a) ( ) je seznam
b) je-li e hodnota prvku seznamu její přidání k seznamu vytvoří seznam

- geometrické útvary:

Hilbertovy křivky



H_1 H_2 H_3

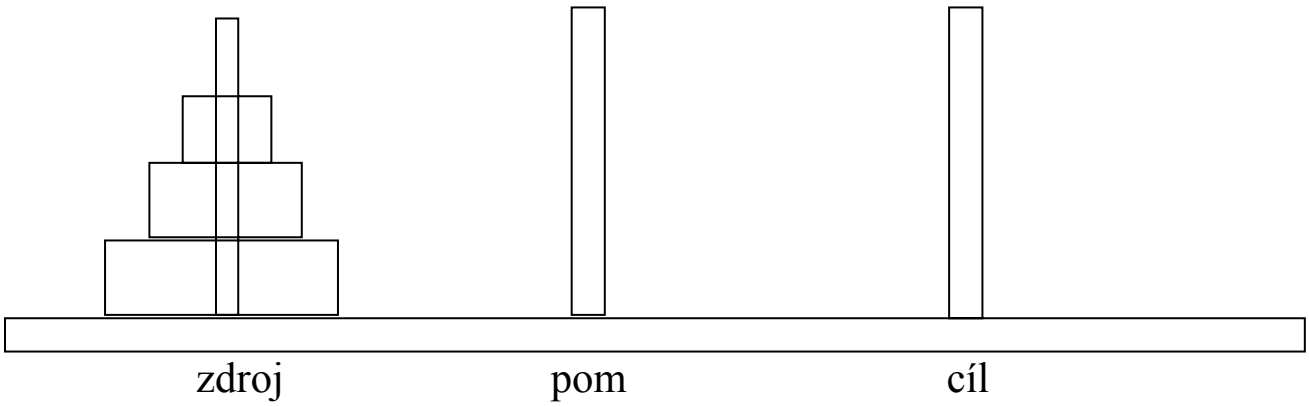
- algoritmy

nalezení nejmenšího prvku pole

a) nejmenší prvek pole s jedním prvkem je tento prvek
b) nejmenší prvek pole s více než jedním prvkem je menší z nejmenších prvků levé a pravé „poloviny“ pole. V případě lichého prvku se jejich velikost liší o 1.

hanojské věže

a) má-li věž velikost jednoho kotouče, přenes kotouč ze zdrojové na cílovou tyč
b) má-li věž velikost **n>1** kotoučů,
přenes věž o velikosti **n-1** kotoučů na pomocnou tyč;
přenes kotouč ze zdrojové na cílovou tyč;
přenes věž o velikosti **n-1** kotoučů z pomocné tyče na cílovou



http://ksvi.mff.cuni.cz/~kryl/Avyuka/20067/animacePRM/hanoi.swf

Rekurzivní schéma:

aspoň jeden případ, který není definován samým sebou – a)

ostatní případy vyjádřeny případy menší velikosti - b)

Schémata jsou nezávislá na počítačích a programování.

Můžeme je vyjádřit pomocí rekurzivních programů.

Faktoriál

```
class Faktorial {
  public static void main (String[] args) {
    int n = Integer.parseInt(args[0]);
    System.out.println(n+"! = "+f(n));
  }

  static int f(int n) {
    if (n > 1)
      return n * f(n - 1);
    else
      return 1;
  }
}
```

Čas výpočtu:

$T(n) = c_{por}$, pro $n \leq 1$

$T(n) = c_{por} + c_{nas} + T(n-1)$, pro $n > 1$

$$T(n) = c_{por} + c_{nas} + T(n-1) = c_{por} + c_{nas} + (c_{por} + c_{nas} + T(n-2)) =$$
$$c_{por} + c_{nas} + (c_{por} + c_{nas} + \ldots + (c_{por} + c_{nas} + T(1)) =$$
$$(n-1)\,(c_{nas} + c_{por}) + c_{por} = n\,c_{por} + (n-1)\,c_{nas}$$

Čas výpočtu je ***O(n).***

- přímočaré použití rekurze může být velice nevhodné

Fibonacciho čísla

$F_0 = 0$,

$F_1 = 1$,

$F_i = F_{i-1} + F_{i-2}$ pro $i \geq 2$

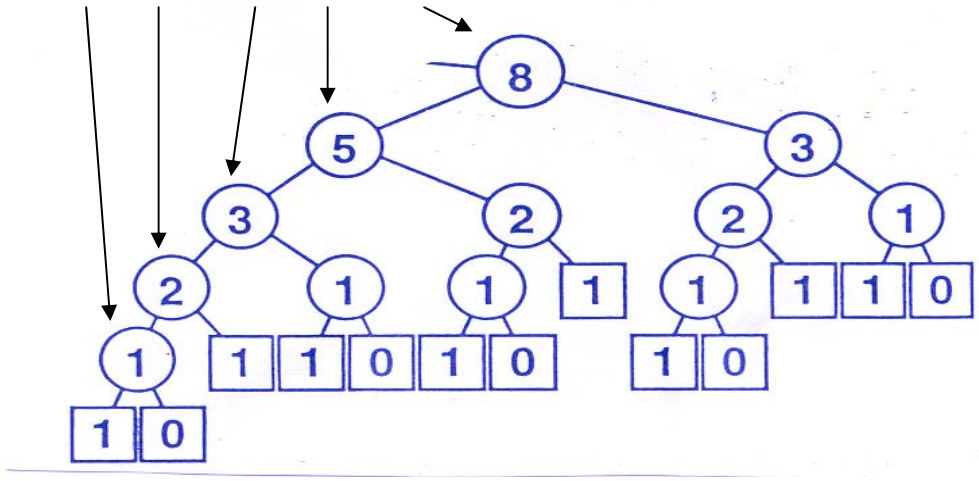
0, 1, 1, 2, 3, 5, 8, …

Rekurzivní výpočet

```
static int F(int i) {
  if (i < 1) return 0;
  if (i == 1) return 1;
  return F(i-1) + F(i-2);
}
```

Rekurzivní volání pro F_6

F_0 F_1 F_2 F_3 F_4 F_5 F_6
0, 1, 1, 2, 3, 5, 8, …



Počet rekurzivních volání pro výpočet F_i

F_2 : 2

F_3: 2 + 1 = 3

F_4: (2 + 1) + (2) = 5

F_5: (2 + 1 + 2) + (2 + 1) = 8

…

Počet rekurzivních volání pro výpočet F_n je F_{n+1} (indukcí)

F_n je nejbližší celé číslo k $\varphi^n / \sqrt{5}$, kde φ je zlatý poměr - 1.618… .

Čas výpočtu roste exponenciálně !

Poznámka: 1 : x = x : (1 – x), 1/x je zlatý poměr

Seznam

přímý průchod seznamem s tiskem hodnot procházených prvků

```
void pruchod(Prvek x) {
  if (x == null)
     return;
  x.tiskPrvku( );
  pruchod(x.dalsi);
}
```

obrácený průchod seznamem s tiskem hodnot procházených prvků

```
void pruchodR(Prvek x) {
  if (x == null)
    return;
  pruchodR(x.dalsi);
  x.tiskPrvku ( );
}
```

koncová rekurze
- rekurzivní výpočet faktoriálu
- přímý průchod seznamem

- je-li posledním krokem rekurzivní metody ***M(x)***, volání ***M(y)***, můžeme volání ***M(y)*** nahradit přiřazením ***x = y*** a skokem na začátek metody ***M***

```
void pruchod(Prvek x) {
  if (x == null)
     return;
  x.tiskPrvku( );
  pruchod(x.dalsi);
}

for (Prvek x = prvni; x != null; x = x.dalsi)
      x.tiskPrvku();
```

Poznámka: metoda pruchodR není takto přímočaře transformovatelná na iterační program

Hilbertovy křivky

Hilbertova křivka prvního řádu vznikla spojením čtyř Hilbertových křivek nultého řádu (bodů) orientovanými úsečkami ← , ↓ , →.

Hilbertova křivka řádu *i+1* vznikne spojením čtyř vhodně rotovaných Hilbertových křivek řádu *i* poloviční velikosti – **A, B, C, D**.

A ≡ D ← A ↓ A → B
B ≡ C ↑ B → B ↓ A
C ≡ B → C ↑ C ← D
D ≡ A ↓ D ← D ↑ C

Hilbertovu křivku H_i nakreslíme voláním metody *A(i)*.

```
A(i) ≡ if (i > 0) {
         D(i-1) ← A(i-1) ↓ A(i-1) → B(i-1)
       }
```

metoda *A* nevolá jenom sama sebe nýbrž i metody *D* a *B*, které opět volají metodu *A* - nepřímá rekurze

H_3:

$$\begin{array}{l} A(2) \leftarrow D(2) \\ \quad\downarrow \\ A(2) \rightarrow B(2) \end{array}$$

$$\begin{array}{ll} C(1) & A(1) \\ \uparrow & \downarrow \\ D(1) \leftarrow & D(1) \end{array}$$

$$\begin{array}{l} A(1) \leftarrow D(1) \\ \quad\downarrow \\ A(1) \rightarrow B(1) \end{array}$$

$$\begin{array}{ll} B(1) \rightarrow & B(1) \\ \uparrow & \downarrow \\ C(1) & A(1) \end{array}$$

$$\begin{array}{cccc} A(1) & & B(1) & \\ \leftarrow & & \rightarrow & \\ \downarrow & & \uparrow \quad \downarrow & \\ \rightarrow & & & \end{array} \qquad \begin{array}{cccc} C(1) & & & D(1) \\ \leftarrow & & & \\ \quad\uparrow & & & \uparrow \quad \downarrow \\ \rightarrow & & & \quad\leftarrow \end{array}$$

$$\begin{array}{l} A \equiv D \leftarrow A \downarrow A \rightarrow B \\ B \equiv C \uparrow B \rightarrow B \downarrow A \\ C \equiv B \rightarrow C \uparrow C \leftarrow D \\ D \equiv A \downarrow D \leftarrow D \uparrow C \end{array}$$

```
public class Hilbert {

  static int x0,y0;
  static int x,y;
  static int h0=512;
  static int n=5;
  static int h;
  static DrawingTool dt; //(0,0)- vlevo nahoře

  static void  doleva( ) {
    int xl = x-h;
    dt.line(x, y, xl, y);
    x = xl;
  }

  static void  dolu( ) {
    int yd = y+h;
    dt.line(x, y, x, yd);
    y = yd;
  }

  static void  doprava( ) {
    int xp = x+h;
    dt.line(x, y, xp, y);
    x = xp;
  }

  static void  nahoru( ) {
    int yh = y-h;
    dt.line(x, y, x, yh);
    y = yh;
  }
```

```
static void  a(int i) {
  if (i>0) {
    d(i-1); doleva( );
    a(i-1); dolu( );
    a(i-1); doprava( );
    b(i-1);
  }
}

static void  b(int i) {
  if (i>0) {
    c(i-1); nahoru( );
    b(i-1); doprava( );
    b(i-1); dolu( );
    a(i-1);
  }
}

static void  c(int i) {
  if (i>0) {
    b(i-1); doprava( );
    c(i-1); nahoru( );
    c(i-1); doleva( );
    d(i-1);
  }
}

static void  d(int i) {
  if (i>0) {
    a(i-1); dolu( );
    d(i-1); doleva( );
    d(i-1); nahoru( );
    c(i-1);
  }
}
```

```
  public static void main(String [] args) {

    int width=600; int height=600;
    dt=new DrawingTool(width,height);
    h=h0; x0=width/2; y0=height/2;
    for (int i=1; i<=n; i++) {
      h = h/2;
      x0 = x0 + h/2; y0 = y0 - h/2;
      x = x0; y = y0;
      a(i);
    }
  }

}
```

Poznámky:

Krok kreslení h křivky H_1 je h0/2, křivky H_2 je h0/4, ..., křivky H_n je $h0/2^n$
Pro vykreslení křivky H_n musí být $h0 = 2^k$, $k \geq n$ (mezi pixely nemůžeme)

Šířky (výšky) křivek jsou
H_1 h0/2
H_2 (2*1 + 1)h0/4 = h0/2 + h0/4
...
H_n h0/2 + h0/4 + ... + $h0/(2^n)$

$\lim_{n\to\infty}$ (h0/2 + h0/4 + ... + $h0/(2^n)$)= h0/2 / (1-1/2) = h0

Všechny křivky se vejdou do čtverce o straně h0
// opravit eKnihu

Nalezení nejmenšího prvku pole

```
class RekMinimum {

  public static void main (String [] args) {
    int[] a = {5,7,3,9,8,2};
    System.out.println(min(a,0,a.length-1));
  }

  static int min(int[] a, int l, int p) {
    if (l==p) return a[l];
    int s = (l+p)/2;
    int minl = min(a,l,s);
    int minp = min(a,s+1,p);
    if (minl < minp) return minl;
    else return minp;
  }

}
```

rekurzivní metoda rozdělí pole (problém) o velikosti *n* na dvě části o velikostech *m* a *n-m.*

počet operací porovnání prvků:

$T(n) = T(m) + T(n-m) + 1$, pro $n > 1$ a $T(1) = 0$

$T(n) = n - 1$.

Důkaz (indukcí):

$T(1) = 1 – 1 = 0$

je-li pro $k < n$ $T(k) = k – 1$
$T(n) = (m – 1) + (n – m – 1) + 1 = n - 1$

Hanojské věže

```
class Veze {

  public static void main(String[] args) {

  int pocetKotoucu=3;

  prenesVez(pocetKotoucu, "zdroj", "pom  ", "cil  ");

  }

  static void prenesVez(int velikost,
                        String z, String p, String c) {
    if(velikost == 1)
      System.out.println("kotouc 1 z "+z+" na "+c);
    else {
      prenesVez(velikost-1, z, c, p);   //zdroj -> pom
      System.out.println("kotouc "+velikost+
                         " z "+z+" na "+c);
      prenesVez(velikost-1, p, z, c);   //pom -> cil
    }
  }

}
```

Počet přenosů kotoučů pro velikost věže *n* je:

$T(n) = 2T(n-1) + 1,$ pro $n \geq 2$ a $T(1) = 1$

$T(n) = 2^n - 1.$

Důkaz (indukcí):

$T(1) = 2^1 - 1 = 1$

je-li pro $k < n$ $T(k) = 2^k - 1$
$T(n) = 2(2^{n-1} - 1) + 1 = 2^n - 1$