

# Spojové struktury

Jan Faigl

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 10

**A0B36PR1 – Programování 1**

# Část 1 – Spojové struktury (seznamy)

Spojové struktury

Spojový seznam

Spojový seznam s odkazem na konec seznamu

Příklad použití

Vložení/odebrání prvku

Kruhový spojový seznam

Obousměrný seznam

# Část I

## Spojové struktury

## Kolekce prvků (položek)

- V programech je velmi běžný požadavek na uchování seznamu (množiny) prvků (proměnných/objektů)

*Viz příklad geometrických objektů v 8. a 9. přednášce*

- Základní příkladem je pole (statické délky)

*V Javě definováno jménem typu a [], například **double[]***

- Jedná se o kolekci položek (proměnných) stejného typu
- + Umožňuje jednoduchý přístup k položkám indexací prvku
  - Položky jsou stejného typu (velikosti)*
- Velikost pole je určena při vytvoření pole *Operátor **new***
  - Velikost (maximální velikost) musí být známa v době vytváření
  - Změna velikost v podstatě není přímo možná
    - Nutná nové vytvoření (alokace paměti)*
  - Využití pouze malé části pole je mrháním paměti
- V případě třídění pole přesouváme položky
  - Vložení prvku a především mazání prvku vyžaduje kopírování položek

## Seznam – list

- Seznam (proměnných nebo objektů) patří mezi základní datové struktury

Základní **ADT** – *Abstract Data Type*

- Seznam zpravidla nabízí sadu základních operací:

- Vložení prvku (**insert**)
- Odebrání prvku (**remove**)
- Vyhledání prvku
- Aktuální počet prvku v seznamu

- Implementace seznamu může být různá:

- Pole statické délky
  - Indexování je velmi rychlé
  - Vložení prvků může být pomalé (nová alokace a kopírování)

Viz například **GeomObjectArray** z 9. přednášky.

[lec09/GeomObjectArray.java](#)

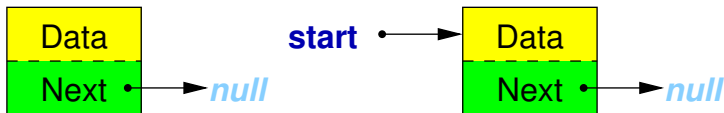
- **Spojové seznamy**

## Spojové seznamy

- Datová struktura realizující seznam variabilní délky
- Každý prvek seznamu obsahuje
  - Datovou část (hodnota proměnné / objekt)
  - Referenci na další prvek v seznamu
 

*null v případě posledního prvku z seznamu.*
- První prvek seznamu se zpravidla označuje jako *head* nebo *start*

*Realizujeme jej jako referenční proměnnou odkazující na první prvek seznamu*



## Základní operace se spojovým seznamem

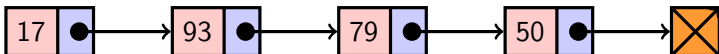
- Vložení prvku
  - Předchozí prvek odkazuje na nový prvek
  - *Nový prvek může odkazovat na předchozí prvek, který na něj odkazuje* *Tzv. obousměrný spojový seznam*
- Odebrání prvku
  - Předchozí prvek aktualizuje hodnotu odkazu na následující prvek
  - Předchozí prvek tak nově odkazuje na následující hodnotu, na kterou odkazoval odebíraný prvek
- Základní implementací spojového seznamu je tzv.  
**jednosměrný spojový seznam**

## Jednosměrný spojový seznam

- Příklad spojového seznamu pro uložení číselných hodnot



- Přidání prvku 50 na konec seznamu



- Odebrání prvku 79



1. Nejdříve sekvenčně najdeme prvek s hodnotou 79
2. Následně vyjmeme a napojíme prvek 93 na prvek 50

*Hodnotu reference next prvku 93 nastavíme na hodnotu reference next odebíraného prvku, tj. na prvek 50*



# Spojový seznam v Javě

- Seznam tvoří dvě třídy
  - Zapouzdření vlastního listu
  - Třída pro uložení prvku seznamu
    - Vlastní data prvku
    - Odkaz (reference) na další prvek
- Příklad tříd pro uložení spojového seznamu celých čísel

```
class ListNode {
    int value;
    ListNode next;
}

public class LinkedList {
    ListNode start;
    ...
}
```

## Zobecnění pro uložení objektů

- V Javě jsou všechny typy (kromě primitivních typů) odvozeny od třídy **Object**
- Můžeme tak zobecnit spojový seznam pro uložení libovolných objektů

```
class ListNode {
    Object item;
    ListNode next;
}

public class LinkedList {
    ListNode start;
    ...
}
```

## Příklad – LinkedList

- Prvky zapouzdříme uvnitř třídy `LinkedList`

```
public class LinkedList {  
    class ListNode { //inner class  
        Object item;  
        ListNode next;  
        ListNode(Object item) {  
            this.item = item;  
            next = null;  
        }  
    }  
    private ListNode start;  
  
    public LinkedList() {  
        start = null;  
    }  
}
```

## LinkedList – push

- Přidání prvku na začátek

```
public void push(Object obj) {  
    ListNode node = new ListNode(obj);  
    if (start == null) { // 1st element  
        start = node;  
    } else {  
        node.next = start; // update the reference  
        start = node;     // set new start  
    }  
}
```

- Přidání prvku není závislé na počtu prvků v seznamu

*Konstantní složitost operace push –  $O(1)$*

## LinkedList – pop

- Odebrání prvního prvku ze seznamu

```
public Object pop() {  
    Object ret = null;  
    if (start != null) {  
        ret = start.item;  
        start = start.next; //can be next item or null  
    }  
    return ret;  
}
```

- Odebrání prvku není závislé na počtu prvků v seznamu

*Konstantní složitost operace pop –  $O(1)$*

## LinkedList – size

- Zjištění počtu prvků v seznamu vyžaduje projít seznam až k zarážce `null`, tj. položka `next` objektu typu `ListNode` je `null`

```
public int size() {  
    int count = 0;  
    ListNode cur = start;  
    while(cur != null) {  
        cur = cur.next;  
        count++;  
    }  
    return count;  
}
```

- Pro zjištění počtu prvků v seznamu musíme projít kompletní seznam, tj.  $n$  položek

*Lineární složitost operace size –  $O(n)$*

## LinkedList – back

- Vrácení hodnoty posledního prvku ze seznamu

```
public Object back() {  
    ListNode end = start;  
    while(end != null && end.next != null) {  
        end = end.next;  
    }  
    return end == null ? null : end.item;  
}
```

- Pro vrácení hodnoty posledního prvku v seznamu musíme projít všechny položky seznamu

*Lineární složitost operace back –  $O(n)$*

## LinkedList – zrychlení operací **size** and **back**

- Operace **size** a **back** procházejí kompletní seznam
- Operaci **size** můžeme urychlit pokud budeme udržovat aktuální počet položek v seznamu
  - Zavedeme datovou položku **int count**
  - Počet prvků inkrementujeme při každém přidání prvku a dekrementuje při každém odebrání prvku
- Operaci **back** můžeme urychlit referenční proměnou odkazující na poslední prvek

```
public class LinkedList { ...  
    private ListNode end;  
    ... }
```

- V případě přidání prvku na začátek, aktualizujeme pouze pokud byl seznam doposud prázdný
- Aktualizujeme v případě přidání prvku na konec
- Nebo při vyjmutí posledního prvku



## LinkedList – urychlený size

```

public class LinkedList {
    private int count;
    public LinkedList() { ...
        count = 0;
    }
    public int size() { return count; }
}

```

```

public void push(Object obj)
{
    ListNode node = new
    ListNode(obj);
    count++;
    ...
}

```

```

public Object pop() {
    Object ret = null;
    if (start != null) {
        count--;
        ... }
    return ret;
}

```

## LinkedList – push s odkazem na konec seznamu

```
public void push(Object obj) {  
    ListNode node = new ListNode(obj);  
    if (start == null) {  
        start = node;  
        end = start; //update tail reference  
    } else {  
        node.next = start;  
        start = node;  
    }  
}
```

*Hodnotu referenční proměnné **end** nastavujeme pouze pokud byl seznam prázdný, protože prvky přidáváme na začátek.*

## LinkedList – pop s odkazem na konec seznamu

```
public Object pop() {  
    Object ret = null;  
    if (start != null) {  
        ret = start.item;  
        start = start.next;  
    }  
    if (start == null) {  
        end = null; // update the tail reference  
    }  
    return ret;  
}
```

*Hodnotu referenční proměnné **end** nastavujeme pouze pokud byl odebrán poslední prvek, protože prvky odebíráme ze začátku.*

## LinkedList – back s odkazem na konec seznamu

- Proměnná **end** je buď **null** nebo odkazuje na poslední prvek seznamu

```
public Object back() {  
    return end != null ? end.item : null;  
}
```

- Udržováním hodnoty proměnné **end** jsme snížili časovou náročnost operace **back** z lineární složitosti na počtu prvků v seznamu  $O(n)$  na konstantní složitost  $O(1)$

## LinkedList – pushEnd

- Přidání prvku na konec seznamu

```
public void pushEnd(Object obj) {  
    ListNode node = new ListNode(obj);  
    if (end == null) { // adding the 1st element  
        start = end = node; //update both start and end  
    } else {  
        end.next = node; // update next of the previous end  
        end = node;    // set new end  
    }  
}
```

- Na asymptotické složitost metody přidání dalšího prvku (na konec seznamu) se nic nemění, je nezávislé na aktuálním počtu prvků v seznamu

## LinkedList – popEnd

- Odebrání prvku z konce seznamu

```
public Object popEnd() {
    if (start == null) { return null; }
    Object ret = end.item;
    if (start == end) {    // the last item is
        start = end = null; // removed from the list
    } else { // there is also penultimate item
        ListNode cur = start; // that needs to be
        while(cur.next != end) { // updated (its
            cur = cur.next;      // reference to the
        }                       // the next item)
        end = cur; // the penultimate is the new end
        end.next = null; // the end does not have next
    }
    return ret;
}
```

*Složitost je  $O(n)$ , protože musíme aktualizovat předposlední prvek. Alternativně lze řešit přes obousměrný spojový seznam.*

## Výpis položek seznamu – `print`

- Seznam postupně procházíme
- Seznam je obecný a pro tisk se využívá metody `toString`

```
public void print() {  
    ListNode cur = start;  
    while(cur != null) {  
        System.out.print(  
            cur.item +  
            (cur.next == null ? "\n" : " ")  
        );  
        cur = cur.next;  
    }  
}
```

`lec10/LinkedListEnd.java`

## Příklad – seznam celých čísel **Integer**

### ■ Příklad použití na seznam objektu třídy **Integer**

```

LinkedListEnd lst = new LinkedListEnd();
lst.push(10).push(5).pushEnd(17).push(7).pushEnd(21);
lst.print();

System.out.println("Pop 1st item: " + lst.pop());
System.out.print("Lst: "); lst.print();

System.out.println("Back of the list: " + lst.back());
System.out.println("Pop from the end: " + lst.popEnd());
System.out.print("Lst: "); lst.print();

```

### ■ Výstup programu

```

java DemoIntLinkedList
7 5 10 17 21
Pop 1st item: 7
Lst: 5 10 17 21
Back of the list: 21
Pop from the end: 21
Lst: 5 10 17

```

lec10/DemoIntLinkedList.java



## LinkedList – vložení prvku do seznamu

- Vložení do seznamu:
  - na začátek – modifikujeme referenční proměnnou **start** (metoda `push`)
  - na konec – modifikujeme referenční proměnnou předposledního prvku a nastavujeme nový konec (metoda `pushEnd`)
  - obecně – potřebujeme hodnotu referenční proměnné prvku, za který chceme nový prvek vložit (**node**)

```
ListNode newNode = new ListNode(obj);  
newNode.next = node.next;  
node.next = newNode;
```

- Do seznamu můžeme chtít prvek vložit na příslušné pořadí, tj. podle indexu v seznamu

*Případně můžeme také požadovat vložení podle hodnoty prvku, tj. vložit před prvek s příslušnou hodnotu.*

## LinkedList – insertAt

- Vložení nového prvku na pozici *index* v seznamu

```
public void insertAt(Object obj, int index) {
    if (index < 0) { return; } //only positive position
    ListNode newNode = new ListNode(obj);
    ListNode node = getNode(index - 1);
    if (node == start) { // the replacement node can
        start = newNode; // be the start node
    }
    if (node != null) { // node can be null for the 1st
        // node, i.e., the situation start == node == null
        newNode.next = node.next;
        node.next = newNode;
    }
}
```

*Neřeší aktualizaci **end** (odkaz na konec seznamu)*

*Pro napojení spojového seznamu potřebuje položku **next**, proto hledáme prvek na pozici  $index - 1$*

## LinkedList – getNode

- Nalezení prvku na pozici **index**
- Pokud je **index** vyšší než počet prvků v poli, návrat posledního prvku

```
private ListNode getNode(int index) {  
    ListNode cur = start;  
    int i = 0;  
    while(i < index && cur != null && cur.next != null) {  
        cur = cur.next;  
        i++;  
    }  
    return cur;  
}
```

*Pokud je seznam prázdný vrátí **null**.*

## Příklad vložení prvků do seznamu (**LinkedList**)

### ■ Příklad vložení do seznam čísel **Integer**

```
LinkedList lst = new LinkedList();  
lst.push(10).push(5).push(17).push(7).push(21);  
lst.print();  
  
lst.insertAt(55, 2);  
lst.print();  
  
lst.insertAt(0, 0);  
lst.print();  
  
lst.insertAt(100, 10);  
lst.print();
```

### ■ Výstup programu

```
java DemoInsertAt  
21 7 17 5 10  
21 7 55 17 5 10  
0 7 55 17 5 10  
0 7 55 17 5 10 100
```

lec10/DemoInsertAt.java

## LinkedList – getAt(int index)

- Nalezení prvků v seznamu podle pozice v seznamu
- V případě „adresace“ mimo rozsah seznamu vrátí **null**

```
public Object getAt(int index) {  
    if (index < 0 || start == null) { return null;}  
    ListNode cur = start;  
    int i = 0;  
    while(i < index && cur != null && cur.next != null) {  
        cur = cur.next;  
        i++;  
    }  
    return (cur != null && i == index) ?  
        cur.item : null;  
}
```

*Složitost operace je v nejnepříznivějším případě  $O(n)$  (v případě pole je to  $O(1)$ )*

## Příklad použití `getAt(int index)`

- Příklad vypsání obsahu seznamu metodou `getAt` v cyklu

```

LinkedList lst = new LinkedList();
lst.push(10).push(5).push(17).push(7).push(21);
lst.print();
for(int i = 0; i < 7; ++i) {
    System.out.println("lst[" + i + "]: " + lst.getAt(i));
}

```

- Výstup programu

```
javac DemoGetAt.java && java DemoGetAt
```

```

21 7 17 5 10
lst[0]: 21
lst[1]: 7
lst[2]: 17
lst[3]: 5
lst[4]: 10
lst[5]: null
lst[6]: null

```

lec10/DemoGetAt.java

*V tomto případě v každém běhu cyklu je složitost metody `get`  $O(n)$  a tedy výpis obsahu seznamu má složitost  $O(n^2)$ !*

## LinkedList – removeAt(int index)

- Odebrání prvku na pozici `int index` a navážeme seznam
- Pro navázání seznamu potřebujeme prvek na pozici `index - 1`

```
public void removeAt(int index) {
    if (index < 0 || start == null) {
        return;
    }
    if (index == 0) {
        pop(); //call the pop function to handle start
    } else {
        ListNode node = getNode(index - 1);
        if (node.next != null) {
            node.next = node.next.next;
        }
    }
}
```

*Složitost v nejnepríznivější případě  $O(n)$  (nejdříve musíme najít prvek).*

## Příklad použití `removeAt(int index)`

```

LinkedList lst = new LinkedList();
lst.push(10).push(5).push(17).push(7).push(21);
lst.print();

System.out.println("Remove item at 3 (" + lst.getAt(3) + ")");
lst.removeAt(3);
lst.print();

System.out.println("Remove item at 3 (" + lst.getAt(3) + ")");
lst.removeAt(3);
lst.print();

System.out.println("Remove item at 0 (" + lst.getAt(0) + ")");
lst.removeAt(0);
lst.print();

```

### ■ Výstup programu

```

javac DemoRemoveAt.java && java DemoRemoveAt
21 7 17 5 10
Remove item at 3 (5)
21 7 17 10
Remove item at 3 (10)
21 7 17
Remove item at 0 (21)
7 17

```

[lec10/DemoRemoveAt.java](#)



## Vyhledání prvku v seznamu podle obsahu – `indexOf`

- Vrátí číslo pozice prvního výskytu prvku v seznamu
- Pokud není prvek v seznamu nalezen vrátí `-1`

```
public int indexOf(Object obj) {  
    int count = 0;  
    ListNode cur = start;  
    boolean found = false;  
    while(cur != null && !found) {  
        found = cur.item.equals(obj);  
        cur = cur.next;  
        count++;  
    }  
    return found ? count-1 : -1;  
}
```

- Porovnání hodnot objektů metodou `equals`!

*Porovnání operátorem `==` porovná hodnoty referenčních proměnných (adresy), nikoliv obsah.*

## Příklad použití `indexOf` 1/3

```
LinkedList lst = new LinkedList();  
lst.push(10).push(5).push(17).push(7).push(21);  
lst.print();  
  
Integer i = 5;  
System.out.println("Index of (" + i + ") is " + lst.indexOf(i));  
  
i = i + 12;  
System.out.println("Index of (" + i + ") is " + lst.indexOf(i));  
  
i = 3;  
System.out.println("Index of (" + i + ") is " + lst.indexOf(i));
```

### ■ Výstup programu

```
java DemoIndexOfInt  
21 7 17 5 10  
Index of (5) is 3  
Index of (17) is 2  
Index of (3) is -1
```

[lec10/DemoIndexOfInt.java](#)

## Příklad použití `indexOf` 2/3

```
LinkedList lst = new LinkedList();  
lst.push("FEE").push("CTU").push("PR1").push("Lecture10");  
lst.print();  
  
String s = "PR1";  
System.out.println("Index of (" + s + ") is " + lst.indexOf(s));  
  
s = "Fee";  
System.out.println("Index of (" + s + ") is " + lst.indexOf(s));
```

### ■ Výstup programu

```
javac DemoIndexOfString.java && java DemoIndexOfString  
Lecture10 PR1 CTU FEE  
Index of (PR1) is 1  
Index of (Fee) is -1  
  
lec10/DemoIndexOfString.java
```

## Příklad použití `indexOf` 3/3

```

LinkedList lst = new LinkedList();
lst.push(10).push(5).push("FEE").push(7).push("CTU");
lst.print();

Integer i = 5;
System.out.println("Index of (" + i + ") is " + lst.indexOf(i));

String s = "FEE";
System.out.println("Index of (" + s + ") is " + lst.indexOf(s));

```

### ■ Výstup programu

```

javac DemoIndexOfMix.java && java DemoIndexOfMix
CTU 7 FEE 5 10
Index of (5) is 3
Index of (FEE) is 2

```

### ■ Využití objektového přístup, dědičnosti a polymorfismu

[lec10/DemoIndexOfMix.java](#)

*Se zavedením generických typů, je i v Javě kladen důraz na statickou typovou kontrolu, tj. kontrolu typů při překladu. Jsou upřednostňovány parametrizované datové struktury pro konkrétní typy (např. Integer nebo String).*

## Odebrání prvku ze seznam podle jeho obsah

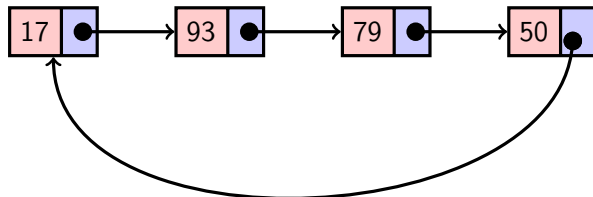
- Podobně jako vyhledání prvku podle obsahu můžeme prvky odebrat
- Můžeme implementovat přímo nebo s využitím již existujících metod `indexOf` a `removeAt`
- Příklad implementace

```
public void remove(Object obj) {  
    int idx = indexOf(obj);  
    while(idx != -1) {  
        removeAt(idx);  
        idx = indexOf(obj);  
    }  
}
```

*Odebíráme všechny výskyty objektu v seznamu.*

## Kruhový spojový seznam

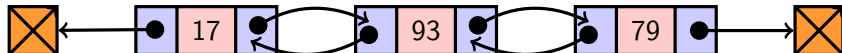
- Položka **next** posledního prvku může odkazovat na první prvek
- Tak vznikne kruhový spojový seznam



- Při přidání prvku na začátek je nutné aktualizovat hodnotu položky **next** posledního prvku

## Obousměrný spojový seznam

- Každý prvek obsahuje odkaz na následující a předchozí položku v seznamu, položky **prev** a **next**
- První prvek má nastavenou položku **prev** na hodnotu **null**
- Poslední prvek má **next** nastavenou na **null**
- Příklad obousměrného seznamu celých čísel



## Příklad – DoubleLinkedList

```
public class DoubleLinkedList {
    class ListNode { //inner class
        ListNode prev;
        ListNode next;
        Object item;
        ListNode(Object item) {
            this.item = item;
            prev = null;
            next = null;
        }
    }
    private ListNode start;
    public LinkedList() {
        start = null;
    }
}
```



## DoubleLinkedList – vložení prvku

### ■ Vložení prvku před prvek **cur**:

1. Napojení vloženého prvku do seznamu, hodnoty **prev** a **next**
2. Aktualizace **next** předchozí prvku k prvku **cur**
3. Aktualizace **prev** proměnné prvku **cur**

```
public void insert(Object obj, ListNode cur) {  
    ListNode newNode = new ListNode(obj);  
    newNode.next = cur;  
    newNode.prev = cur.prev;  
    if (cur.prev != null) {  
        cur.prev.next = newNode;  
    }  
    cur.prev = newNode;  
}
```

## DoubleLinkedList – přidání prvku na začátek seznamu

### push

```
public DoubleLinkedList push(Object obj) {  
    ListNode node = new ListNode(obj);  
    if (start == null) {  
        start = end = node;  
    } else {  
        node.next = start;  
        start.prev = node;  
        start = node;  
    }  
    return this;  
}
```

lec10/DoubleLinkedList.java

## DoubleLinkedList – tisk seznamu `print` a `printReverse`

```
public void print() {
    ListNode cur = start;
    while(cur != null) {
        System.out.print(cur.item +
            (cur.next == null ? "\n" : " "));
        cur = cur.next;
    }
}

public void printReverse() {
    ListNode cur = end;
    while(cur != null) {
        System.out.print(cur.item +
            (cur.prev == null ? "\n" : " "));
        cur = cur.prev;
    }
}
```

## Příklad použití

```

LinkedListEnd lst = new LinkedListEnd();
lst.push(10).push(5).pushEnd(17).push(7).pushEnd(21);
lst.print();

System.out.println("Pop 1st item: " + lst.pop());
System.out.print("Lst: "); lst.print();

System.out.println("Back of the list: " + lst.back());
System.out.println("Pop from the end: " + lst.popEnd());
System.out.print("Lst: "); lst.print();

```

### ■ Výstup programu

```

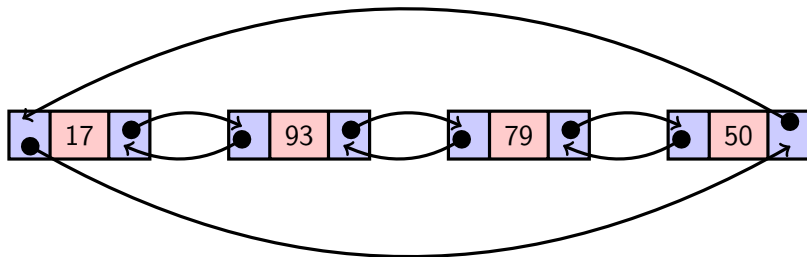
javac DemoDoubleLinkedList.java && java DemoDoubleLinkedList
Regular print:
DDDD CCC BB A
Revert print:
A BB CCC DDDD

```

[lec10/DemoDoubleLinkedList.java](#)

## Kruhový obousměrný seznam

- Položka **next** posledního prvku odkazuje na první prvek
- Položka **prev** prvního prvku odkazuje na poslední prvek



# Shrnutí přednášky

## Diskutovaná témata

- Spojové seznamy – lineární spojové struktury
- Jednosměrný spojový seznam
  - Operace vkládání a odebrání prvku
  - Průchod seznamem
  - Vyhledávání prvku v seznamu
- Kruhový jednosměrný spojový seznam
- Obousměrný spojový seznam
- Kruhový obousměrný spojový seznam
- **Příště: Nelineární spojové struktury a abstraktní datový typ**