Dynamické struktury a Abstraktní Datový Typy (ADT)

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 11

A0B36PR1 – Programování 1

Část 1 – Spojové struktury (stromy)

Část 2 – Abstraktní datový typ

Datové struktury

Zásobník

Fronta

Část 3 – Příklad ADT – Prioritní fronta

Popis

Implementace spojovým seznamem

Binárním strom a halda

Část I

Stromy

Lineární a nelineární spojové struktury

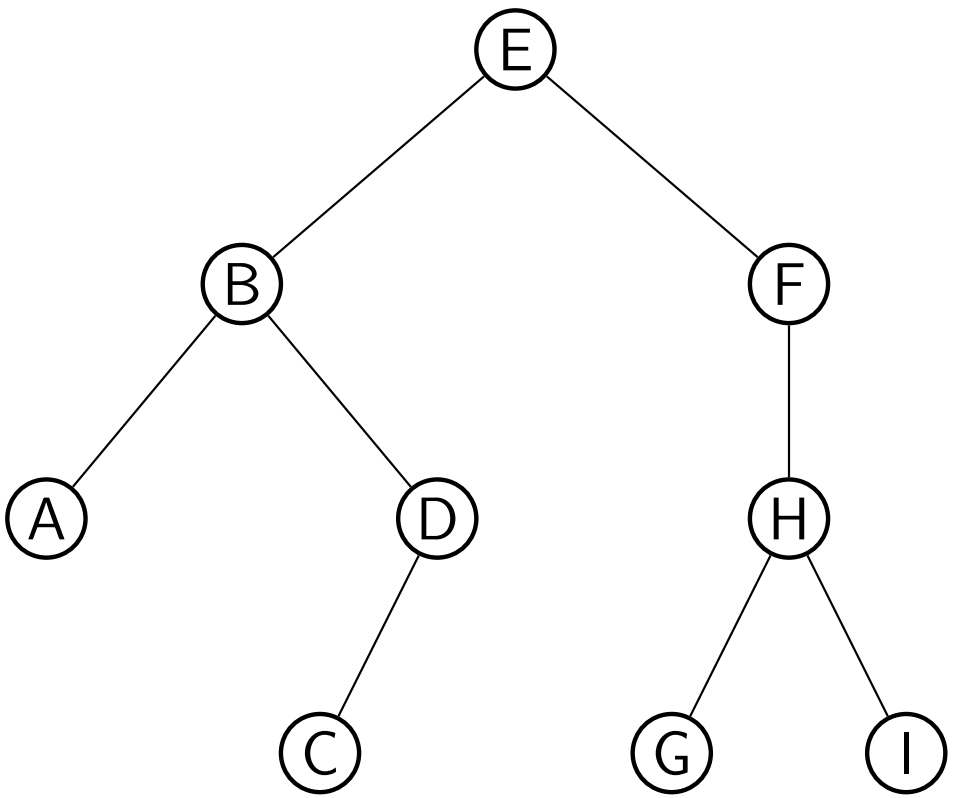
- Spojové seznamy představují lineární spojovou strukturu

  Každý prvek má nejvýše jednoho následníka

- Nelineární spojové struktury (např. stromy)

  Každý prvek může mít více následníků

- **Binární strom**: každý prvek (uzel) má nejvýše dva následníky



- **kořen stromu**
- **list**
- **levý podstrom**
- **pravý podstrom**

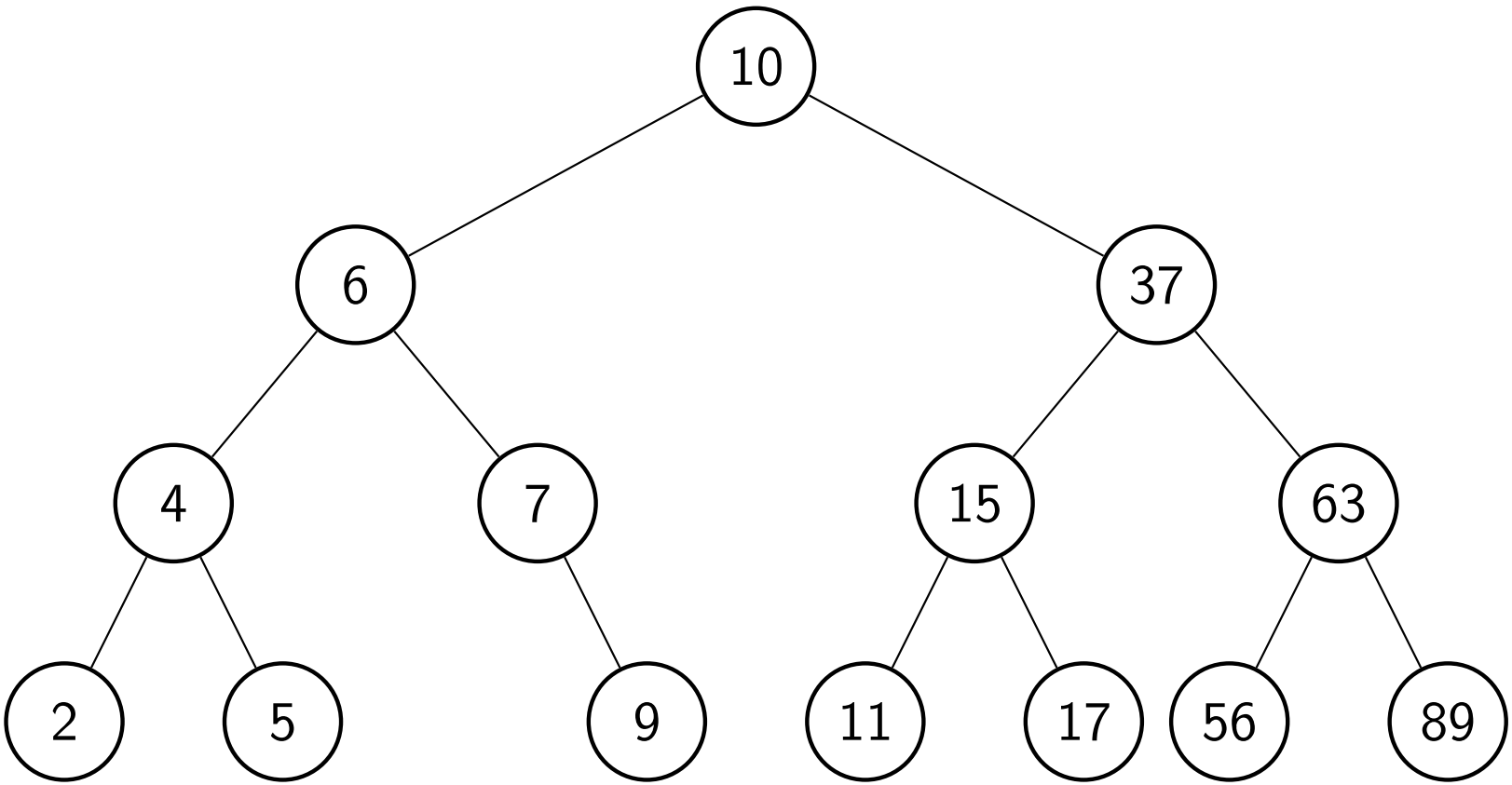
Binární strom

```
public class BinaryTree {
    class Node {
        Object value;
        Node left;
        Node right;
        Node(Object obj) {
            value = obj;
            left = right = null;
        }
    }
    private Node root;

    public BinaryTree() {
        root = null;
    }
}
```

Příklad – Binární vyhledávací strom

- Pro každý prvek (uzel) platí, že hodnota potomka vlevo je menší (nebo null) a hodnota potomka vpravo je větší (nebo null)



Průchod binárním vyhledávacím stromem

- Při hledání prvku konkrétní hodnoty klíče se postupně zanořujeme hlouběji do stromu, přičemž může nastat jedna z následujících situací:
  1. Aktuální prvek má hledanou hodnotu klíče, hledání je ukončeno
  2. Hodnota klíče je menší než hodnota aktuálního prvku, pokračujeme v hledání v další úrovni levého potomka
  3. Hodnota klíče je větší než hodnota aktuálního prvku, pokračujeme v hledání v další úrovni pravého potomka
  4. Aktuální prvek má hodnotu **null**, hledání je ukončeno, prvek ve stromu není
- Při průchodu stromem postupujeme rekurzivně tak, že nejdříve navštěvujeme levé potomky a následně pak pravé potomky

  Pokud budeme při takovém průchodu vypisovat hodnoty v levém podstromu, pak hodnotu prvku a následně hodnoty v pravém podstromu, vypíšeme hodnoty uložené ve stromu uspořádaně (sestupně nebo vzestupně

Příklad binárního stromu textových řetězců

```
public class BinaryStringTree {
  class Node {
     final String value;
     Node left;
     Node right;
     Node(String s) {
        value = s;
        left = right = null;
     }
  }
  private Node root;
  private int counter; //for counting traversed nodes
  public BinaryStringTree() {
     root = null;
  }
  ...
```

BinaryStringTree – insert 1/2

- Při vložení prvku rekurzivně procházíme strom
- Prvek zařadíme tak, aby platila podmínka vyhledávacího stromu

Tj. hodnoty všech uzlů v levém podstromu jsou menší a hodnoty všech uzlů v pravé podstromu jsou větší než hodnota prvku každého uzlu.

```
public void insert(String s) {
    if (root == null) {
        root = new Node(s);
    } else {
        insert(s, root);
    }
}
private void insert(String s, Node node) {
    ...
}
```

BinaryStringTree – insert 2/2

```
private void insert(String s, Node node) {
    if (node.value.compareTo(s) > 0) {
        if (node.left == null) {
            node.left = new Node(s);
        } else {
            insert(s, node.left);
        }
    } else {
        if (node.right == null) {
            node.right = new Node(s);
        } else {
            insert(s, node.right);
        }
    }
}
```

BinaryStringTree – traverse 1/2

- Při průchodu stromu postupujeme nejdříve do levého podstromu a až pak do pravého
- V proměnné **counter** počítáme pořadí navštívených uzlů

```
public void traverse() {
    counter = 0;
    traverse(root);
}

private void traverse(Node node) {
    ...
}
```

BinaryStringTree – traverse 2/2

```
private void traverse(Node node) {
    if (node != null) {
        if (node.left != null) {
            traverse(node.left);
        }
        System.out.print(counter + ": " + node.value + " ");
        counter++;
        if (node.right != null) {
            traverse(node.right);
        }
    }
}
```

Rekurzivní průchod stromem!

BinaryStringTree – Příklad použití

```
String[] strings = {"ZZZ", "DDD", "AAA",  "YYY", "BBB",
   "QQQ", "CCC"};

BinaryStringTree tree = new BinaryStringTree();

System.out.println("Print array:");
for(int i = 0; i < strings.length; ++i) {
   System.out.print(i +": " + strings[i] + " ");
   tree.insert(strings[i]);
}
System.out.println("\n\nTraverse tree");
tree.traverse();
```

- Výstup programu

```
javac DemoBinaryStringTree.java && java DemoBinaryStringTree
Print array:
0: ZZZ 1: DDD 2: AAA 3: YYY 4: BBB 5: QQQ 6: CCC

Traverse tree
0: AAA 1: BBB 2: CCC 3: DDD 4: QQQ 5: YYY 6: ZZZ
```

lec11/BinaryStringTree.java, lec11/DemoBinaryStringTree.java

Stromové struktury

- Stromové struktury jsou významné datové struktury pro vyhledávání

  Složitost vyhledávání je úměrná hloubce stromu.
- Binární stromy – každý uzel má nejvýše dva následníky
  - Hloubku stromu lze snížit tzv. vyvažováním stromu

    Georgy Adelson-Velsky a Landis
    - AVL stromy
    - Red-Black stromy
  - Plný binární strom – každý vnitřní uzel má dva potomky a všechny uzly jsou co nejvíce vlevo
    - Můžeme efektivně reprezentovat polem (pro daný maximální počet uzlů)
    - Lze použít pro efektivní implementaci prioritní fronty (Heap – halda)
    - Základem třídicího algoritmu *Heap Sort*
- Vícecestné stromy – např. B–strom (Bayer tree) pro ukládání utříděných záznamů

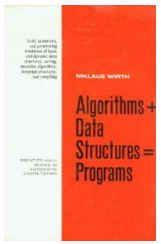
  Informativní více v Algoritmizaci

Část II

Abstraktní datový typ

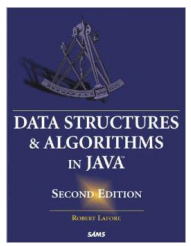
Zdroje

Algorithms + Data Structures = Programs, *Niklaus Emil Wirth* Prentice-Hall 1975, ISBN 0-13-022418-9

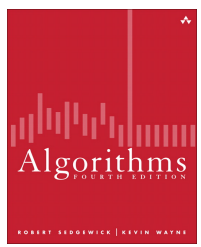


Update 2004, http://www.ethoberon.ethz.ch/WirthPubl/AD.pdf

Data Structures and Algorithms in Java (2nd Edition) *Robert Lafore* Sams, 2002, ISBN: 978-0672324536



Algorithms (4th Edition) *Robert Sedgewick and Kevin Wayne* Addison-Wesley Professional, 2011, ISBN: 978-0321573513



http://algs4.cs.princeton.edu/home

- Algorithms and Data Structures with implementations in Java and C++
  http://www.algolist.net
- Algorithms and Data Structures
  http://introcs.cs.princeton.edu/java/40algorithms

Datové struktury a abstraktní datový typ

- **Datová struktura** (typ) je množina dat a operací s těmito daty
- **Abstraktní datový typ** formálně definuje data a operace s nimi
  - Fronta (Queue)
  - Zásobník (Stack)
  - Pole (Array)
  - Tabulka (Table)
  - Seznam (List)
  - Strom (Tree)
  - Množina (Set)

Nezávislé na konkrétní implementaci

Abstraktní datový typ

- Množina druhů dat (hodnot) a příslušných operací, které jsou přesně specifikovány a to **nezávisle na konkrétní implementaci**
- Můžeme definovat:
  - Matematicky – signatura a axiomy
  - Rozhraním (interface) a popisem operací, kde rozhraní poskytuje:
    - Konstruktor vracející odkaz (na objekt nebo strukturu)

      Objektově orientovaný i procedurální přístup
    - Operace, které akceptují odkaz na argument (data) a které mají přesně definovaný účinek na data

Příklad matematického popisu ADT – Datový typ Boolean

- Syntax popisuje, jak správně vytvořit logický výraz:
  1. `true` a `false` jsou logické výrazy
  2. Jestliže x a y jsou logické výrazy, pak
     - i **!(x)** – negace
     - ii **(x & y)** – logický součin `and`
     - iii **(x | y** – logický součet `or`
     - iv **(x==y)**, **(x != y)** – relační operátory

     jsou logické výrazy.

Pokud se chceme vyhnout psát u každé operace závorky, musíme definovat priority operátorů.

Příklad matematického popisu ADT – Datový typ Boolean

- Sémantika popisuje význam jednotlivých operací, můžeme definovat axiomy:
  - **true == true** : `true`
  - **false == false** : `true`
  - **true == false** : `false`
  - **false == true** : `false`
  - **!(true)** : `false`
  - **!(false)** : `true`
  - **x & false** : `false`
  - **x & true** : **x**
  - **x & y** : **y & y**
  - **x | y** : **y | y**
  - **x | false** : **x**
  - **x | true** : `true`

Abstraktní datový typ (ADT) – Vlastnosti

- Počet datových položek může být
  - Neměnný – **statický datový typ** – počet položek je konstantní

    Např. pole, řetězec, třída
  - Proměnný – **dynamický datový typ** – počet položek se mění v závislosti na provedené operaci

    Např. vložení nebo odebrání určitého prvku
- Typ položek (dat):
  - **Homogenní** – všechny položky jsou stejného typu
  - **Nehomogenní** – položky mohou být různého typu
- Existence bezprostředního následníka
  - **Lineární** – existuje bezprostřední následník prvku, např. pole, fronta, seznam, ...
  - **Nelineární** – neexistuje přímý jednoznačný následník, např. strom

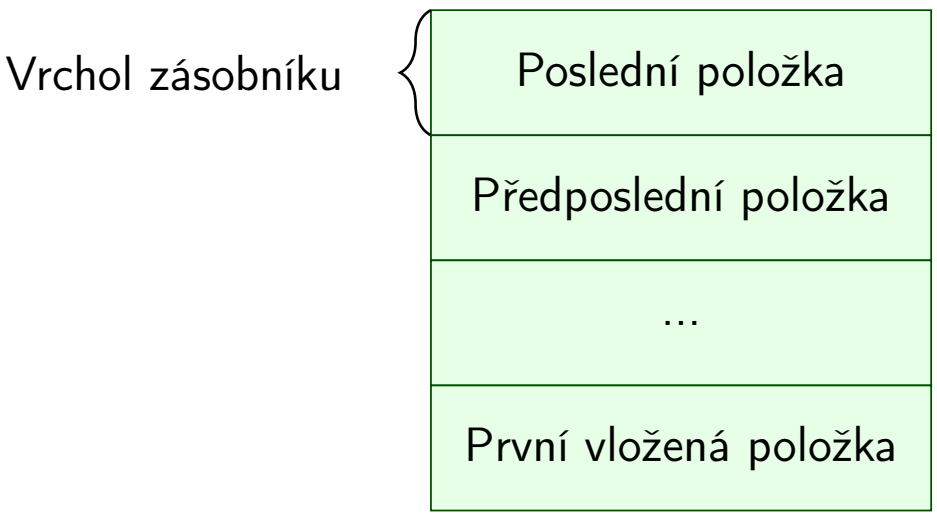
Příklad ADT – Zásobník

Zásobník je dynamická datová struktura umožňující vkládání a odebírání hodnot tak, že naposledy vložená hodnota se odebere jako první

LIFO – Last In, First Out

Základní operace:

- Vložení hodnoty na vrchol zásobníku
- Odebrání hodnoty z vrcholu zásobníku
- Test na prázdnost zásobníku



Příklad ADT – Operace nad zásobníkem

Základní operace nad zásobníkem

- **push** – vložení prvku na vrchol zásobníku
- **pop** – vyjmutí prvku z vrcholu zásobníku
- **isEmpty** – test na prázdnost zásobníku

Další operace nad zásobníkem mohou být

- **peek** – čtení hodnoty z vrcholu zásobníku

*alternativně také třeba **top***

- *search* – vrátí pozici prvku v zásobníku

Pokud se nachází v zásobníku, jinak -1

- *size* – vrátí aktuální počet hodnot v zásobníku

Zpravidla není potřeba

Příklad ADT – Rozhraní zásobníku 1/2

- V objektově orientovaném programování můžeme s výhodou využít principu **zapouzdření**
- Zásobník můžeme definovat rozhraním, bez konkrétní implementace

```
public interface Stack {
    public Object push(Object item);
    public Object pop();
    public boolean isEmpty();

    public Object peek();
}
```

- Součástí definice rozhraní ADT je však také přesný popis operací

Definice může být textový popis, např. v Javě lze využít javadoc.

Příklad ADT – Rozhraní zásobníku 2/2

```
public interface Stack {
    /**
     * Add the item if it is not null
     *
     * @param item
     * @return the added item or null if object has not
      been added to the stack
     */
    public Object push(Object item);
    public Object pop();
    public boolean isEmpty();

    public Object peek();
};
```

https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html

Implementace zásobníku polem 1/3

```
public class StackArray implements Stack {
    private final int MAX_SIZE = 10;
    private Object[] stack;
    private int count;

    public StackArray() { ... }
    @Override
    public Object push(Object item) { ... }
    @Override
    public Object pop() { ... }
    @Override
    public boolean isEmpty() { ... }
    @Override
    public Object peek() { ... }
}
```

lec11/StackArray.java

Implementace zásobníku polem 2/3

```
public StackArray() {
    stack = new Object[MAX_SIZE];
    count = 0;
}

@Override
public Object push(Object item) {
    if (item == null || count == stack.length) {
        return null;
    }
    stack[count++] = item;
    return item;
}
```

Implementace zásobníku polem 3/3

```
@Override
public Object pop() {
   return count > 0 ? stack[--count] : null;
}

@Override
public boolean isEmpty() {
   return count == 0;
}

@Override
public Object peek() {
   return count > 0 ? stack[count - 1] : null;
}
```

Proč v metodě `pop` *používáme* `--count` *a v* `peek` `count - 1`*?*

Zásobník – Příklad použití

- Uloží do zásobníku pouze `MAX_SIZE` (10) položek

```
Stack stack = new StackArray();
for (int i = 0; i < 15; ++i) {
   stack.push((int) (Math.random() * 50));
   System.out.println(
         "Add " + i + " item to the stack: " +
         stack.peek()
         );
}
System.out.println("\nPop the items from the stack:");
while (stack.isEmpty() == false) {
   System.out.print(stack.pop() + " ");
}
System.out.println("");
```

lec11/DemoStack.java

Implementace zásobníku rozšiřitelným polem 1/2

- V případě naplnění pole vytvoříme nové o „něco" větší pole
- Zavedeme položku `MAX_RESIZE`

```
public class StackResizeArray implements Stack {
    private final int RESIZE = 10;
    private Object[] stack;
    private int count;

    public StackResizeArray() {
        stack = new Object[RESIZE];
        count = 0;
    }
    ...
}
```

- Modifikujeme metodu **push**

Implementace zásobníku rozšiřitelným polem 2/2

```
@Override
public Object push(Object item) {
  if (item == null) { return null; }
  if (count == stack.length) {
    Object[] newStack = new Object[stack.length + RESIZE];
    System.arraycopy(stack, 0, newStack, 0, stack.length);
    stack = newStack;
  }
  return stack[count++] = item;
}
```

lec11/StackResizeArray.java

- Použití identické jako pro StackArray, tentokrát je možné uložit do zásobníku více položek

```
Stack stack = new StackResizeArray();
for (int i = 0; i < 15; ++i) {
    ...
```

lec11/DemoStack.java

Implementace zásobníku spojovým seznamem 1/3

- Zásobník také můžeme implementovat spojovým seznamem

```
public class StackLinkedList implements Stack {
   private class ListNode {
      Object item;
      ListNode next;
      ListNode(Object item) {
         this.item = item;
         next = null;
      }
   }
   private ListNode start;
   public StackLinkedList() { start = null; }
   public Object push(Object item) { ... }
   public Object pop() { ... }
   public boolean isEmpty() { ... }
   public Object peek() { ... }
}
```

lec11/StackLinkedList.java

Implementace zásobníku spojovým seznamem 2/3

```
@Override
public Object push(Object item) {
    if (item == null) { return null; }
    ListNode node = new ListNode(item);
    if (start == null) {
        start = node;
    } else {
        node.next = start;
        start = node;
    }
    return item;
}
```

Implementace zásobníku spojovým seznamem 3/3

```
@Override
public Object pop() {
   Object ret = null;
   if (start != null) {
      ret = start.item;
      start = start.next;
   }
   return ret;
}
@Override
public boolean isEmpty() {
   return start == null;
}
@Override
public Object peek() {
   return start != null ? start.item : null;
}
```

ADT – Zásobník příklad použití různých implementací

```
final int TYPE = 3;
Stack stack =
    (TYPE == 1) ?
    new StackArray() : (TYPE == 2) ?
    new StackResizeArray() : new StackLinkedList();

System.out.println("stack type " + stack);

for (int i = 0; i < 15; ++i) {
   stack.push((int) (Math.random() * 50));
}
System.out.println("\nPop items from the stack:");
while (stack.isEmpty() == false) {
   System.out.print(stack.pop() + " ");
}
System.out.println("");
```

Zásobník **StackArray** – vložení prázdného prvku

- Operaci **push** jsme definovali tak, že nelze vložit prázdný (**null**) prvek
- Nedokážeme rozlišit, zdali návratová hodnota **null** metody **push** znamená:
  1. Pokus o vložení prázdného prvku
  2. Indikaci plného zásobníku
- Přeplnění zásobníku můžeme indikovat mechanismem výjimek

```
public Object push(Object item) {
    if (count == stack.length) {
        throw new RuntimeException("Stack full");
    }
    if (item == null) { return null; }
    stack[count++] = item;
    return item;
}
```

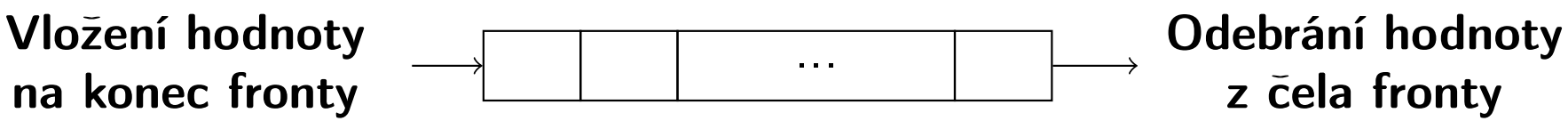
Výjimku `RuntimeException` *není třeba deklarovat v hlavičce metody, více v PR2*

ADT – Zásobník a vložení prázdného prvku

- Obecně není nutné rozlišovat, zdali je vkládaný prvek prázdný (null) či nikoliv
- Je jen na uživateli (programátorovi) jak zásobník používá a jaké prvky ukládá
- Podobně můžeme využít mechanismu výjimek při pokusu vyjmout prvek metodou **pop** z prázdného zásobníku
  - Uživatel má k dispozici metodu **isEmpty** pro ověření, zdali je zásobník prázdný či nikoliv

Příklad ADT – Fronta

- **Fronta** je dynamická datová struktura, kde se odebírají prvky v tom pořadí, v jakém byly vloženy
- Jedná se o strukturu typu FIFO (First In, First Out)



- Implementace
  - Pole

    Pamatujeme si pozici začátku a konce fronty v poli
  - Spojovým seznamem

    Pamatujeme si ukazatel na začátek a konec fronty

https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

ADT – **Fronta** – Příklad rozhraní

```
public interface Queue {
    /**
     * Add item to the queue
     *
     * @param item
     * @return the added item
     */
    public Object push(Object item);

    public Object pop();
    public boolean isEmpty();

    public Object peek();
    public Object back();
}
```

Implementace fronty polem

```
public Object push(Object item) {
    if (count == queue.length) {
        throw new RuntimeException("Queue full");
    }
    queue[end] = item;
    end = (end + 1) % queue.length;
    count++;
}
public Object pop() {
    if (count == 0) {
        throw new RuntimeException("Queue empty");
    }
    Object ret = queue[start];
    start = (start + 1) % queue.length;
    count--;
    return ret;
}
```

Implementace fronty spojovým seznamem 1/4

```
public class QueueLinkedList implements Queue {
    private class ListNode {
        Object item;
        ListNode next;
        ListNode(Object item) {
            this.item = item;
            next = null;
        }
    }
    private ListNode start;
    private ListNode end;
    public QueueLinkedList() {
        start = end = null;
    }
    public Object push(Object item) { ... }
    ...
}
```

Implementace fronty spojovým seznamem 2/4

```
@Override
public Object push(Object item) {
    ListNode node = new ListNode(item);
    if (start == null) {
        start = end = node;
    } else {
        end.next = node;
        end = end.next;
    }
    return item;
}
```

Implementace fronty spojovým seznamem 3/4

```
@Override
public Object pop() {
    if (start == null) {
        throw new RuntimeException("Queue empty");
    }
    Object ret = start.item;
    start = start.next;
    if (start == null) {
        end = null;
    }
    return ret;
}
```

Implementace fronty spojovým seznamem 4/4

```
@Override
public boolean isEmpty() {
    return start == null;
}

@Override
public Object peek() {
    return start != null ? start.item : null;
}

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

ADT – **Fronta** spojovým seznamem – příklad použití

```
Queue queue = new QueueLinkedList();
for (int i = 0; i < 15; ++i) {
   queue.push((int) (Math.random() * 50));
   System.out.println(
         "Add " + i +
         " front: " + queue.peek() +
         " back:" + queue.back());
}
System.out.println("\nPop the items from the queue:");
while (queue.isEmpty() == false) {
   System.out.print(queue.pop() + " ");
}
System.out.println("");
```

lec11/QueueLinkedList.java, lec11/DemoQueue.java

Část III

ADT – Prioritní fronta a možné implementace

Prioritní fronta

- Fronta
  - První vložený prvek je první odebraný prvek

    FIFO
- Prioritní fronta
  - Některé prvky jsou při vyjmutí z fronty preferovány

    Některé vložené objekty je potřeba obsloužit naléhavěji, např. fronta pacientů u lékaře.
  - Operace **pop** odebírá z fronty prvek s nejvyšší prioritou

    Vrchol fronty je prvek s nejvyšší prioritou.
- Rozhraní prioritní fronty může být identické jako o běžné fronty, avšak specifikace upřesňuje chování dílčích metod

`https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html`

Prioritní fronta – specifikace rozhraní

Prioritní frontu můžeme implementovat různě složitě a také s různými výpočetními nároky, např.

- Polem nebo spojovým seznamem s modifikací metod **push** nebo **pop** a **peek**
- S využitím pokročilé datové struktury pro efektivní vyhledání prioritního prvku (halda)

Prioritní prvek může být ten s nejmenší hodnotou, pak

- Metody **pop** a **peek** vrací nejmenší prvek dosud vložený do fronty
- Prvky potřebujeme porovnávat, proto nestačí „jen" vkládat prvky typu `Object`, ale potřebujeme je také porovnávat, například metodou **compareTo** rozhraní **Comparable**

`https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html`

Prioritní fronta – příklad rozhraní

```
public interface PriorityQueue {
    public Comparable push(Comparable item);
    /**
      Remove the head of the priority queue, where
      the is the least element in the queue

      @return the removed least element in the queue
    */
    public Comparable pop();
    public Comparable peek();
    public boolean isEmpty();
}
```

Prioritní fronta spojovým seznamem 1/4

```
public class PriorityQueueLinkedListNaive implements
    PriorityQueue {
   private class ListNode {
      Comparable item;
      ListNode next;
      ListNode(Comparable item) {
         this.item = item;
         next = null;
      }
   }
   private ListNode start;
   public PriorityQueueLinkedListNaive() {
      start = null;
   }
   ...
}
```

Prioritní fronta spojovým seznamem 2/4

- Nedovolíme vkládat prázdný prvek

```
@Override
public Comparable push(Comparable item) {
    if (item == null) {
        throw new RuntimeException("Adding null ...");
    }
    ListNode node = new ListNode(item);
    if (start == null) {
        start = node;
    } else {
        node.next = start;
        start = node;
    }
    return item;
}
```

Prioritní fronta spojovým seznamem 3/4

- Metoda **peek** vrací nejmenší prvek ve frontě, který musíme nejdříve najít

```
@Override
public Comparable peek() {
    if (start == null) { return null; }
    ListNode ret = start;
    ListNode cur = start.next;
    while (cur != null) {
        if (cur.item.compareTo(ret.item) < 0) {
            ret = cur;
        }
        cur = cur.next;
    }
    return ret.item;
}
```

V nejnepříznivějším případě procházíme celou frontu $\sim O(n)$

Prioritní fronta spojovým seznamem 4/4

```
public Comparable pop() {
  if (start == null) { return null; }
  ListNode prevRet = null; //remember the previous of the ret
  ListNode ret = start;
  ListNode prev = start;
  ListNode cur = start.next;
  while (cur != null) {
     if (cur.item.compareTo(ret.item) < 0) {
       ret = cur;
       prevRet = prev;
     }
     prev = cur;
     cur = cur.next;
  }
  if (prevRet == null) {
    start = start.next;
  } else {
    prevRet.next = ret.next;
  }
  return ret.item;
}
```

Musíme najít nejmenší prvek a zároveň zajistit napojení spojového seznamu po jeho odebrání ze seznamu, proto potřebujeme také předcházející prvek v seznamu.

Prioritní fronta spojovým seznamem – Příklad použití

```
PriorityQueue queue = new PriorityQueueLinkedListNaive();
for (int i = 0; i < 15; ++i) {
   int v = (int) (Math.random() * 50);
   queue.push(v);
   System.out.println("Add " + v + " current peek is " +
    queue.peek());
}
System.out.println("\nPop the items from the priority
    queue:");
while (queue.isEmpty() == false) {
   System.out.print(queue.pop() + " ");
}
System.out.println("");
```

lec11/PriorityQueueLinkedListNaive.java, lec11/DemoPriorityQueue.java

- Příklad výstupu:

```
Add 43 current peek is 43
Add 27 current peek is 27
Add 26 current peek is 26
Add 20 current peek is 20

Pop the items from the priority queue:
20 26 27 43
```

Prioritní fronta spojovým seznamem a výpočetní náročnost

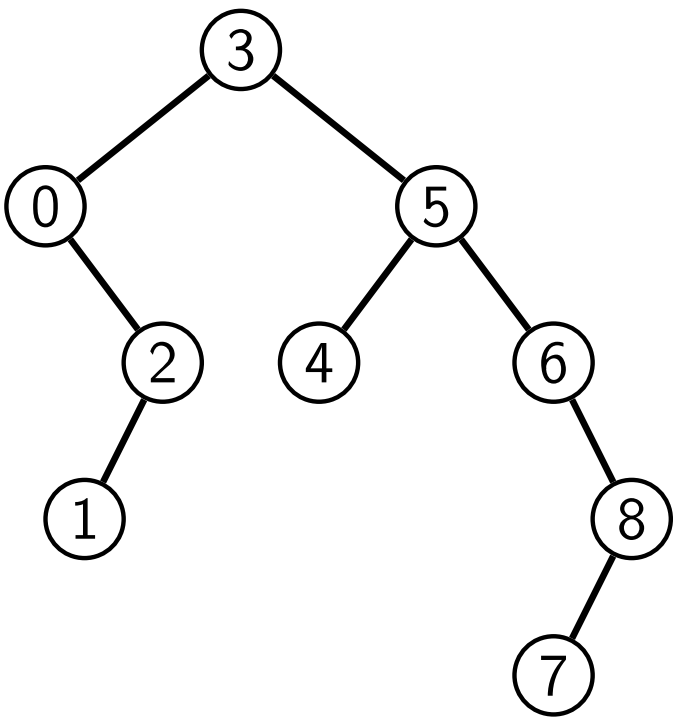
- V naivní implementaci prioritní fronty jsme zohlednění priority „odložili" až do doby, kdy potřebujeme odebrat prvek z fronty
- Při odebrání (nebo vrácení) nejmenšího prvku v nejhorším případě musíme projít celý seznam
- To může být v řadě praktických aplikací nedostatečné a raději bychom chtěli „udržovat" nejmenší prvek připravený
  - Můžeme to například udělat zavedením položky **head**, ve které bude aktuálně nejmenší vložený prvek do fronty
  - Prvek **head** aktualizujeme v metodě **push** porovnáním hodnoty aktuálně vkládaného prvku
  - Tím zefektivníme operaci **peek**
  - V případě odebrání nejmenšího prvku, však musíme frontu znovu projít a najít nový nejmenší prvek

Alternativně můžeme použít sofistikovanější datovou strukturu, která nám umožní efektivně udržovat hodnotu nejmenšího prvku a to jak při operaci vložení (**push**) tak při operaci vyjmutí (**pop**) prvku z prioritní fronty.

Binární vyhledávací strom a halda

Binární vyhledávací strom

- Může obsahovat prázdná místa
- Hloubka stromu se může měnit
- Strom můžeme procházet

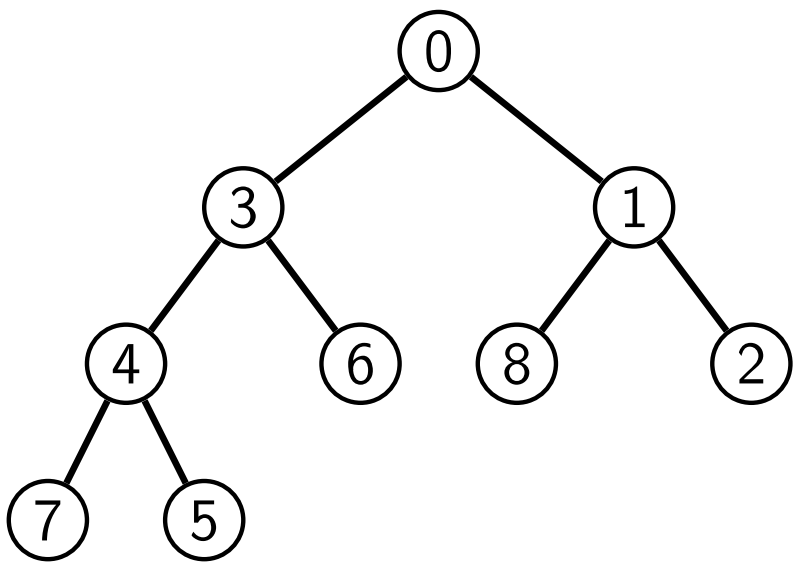


Halda

- Binární plný strom

  Hloubka stromu vždy $\lfloor \log_2(n) \rfloor$
- Kořen stromu je vždy prvek s nejmenší hodnotou
- Splňuje vlastnost haldy

Heap property



Halda

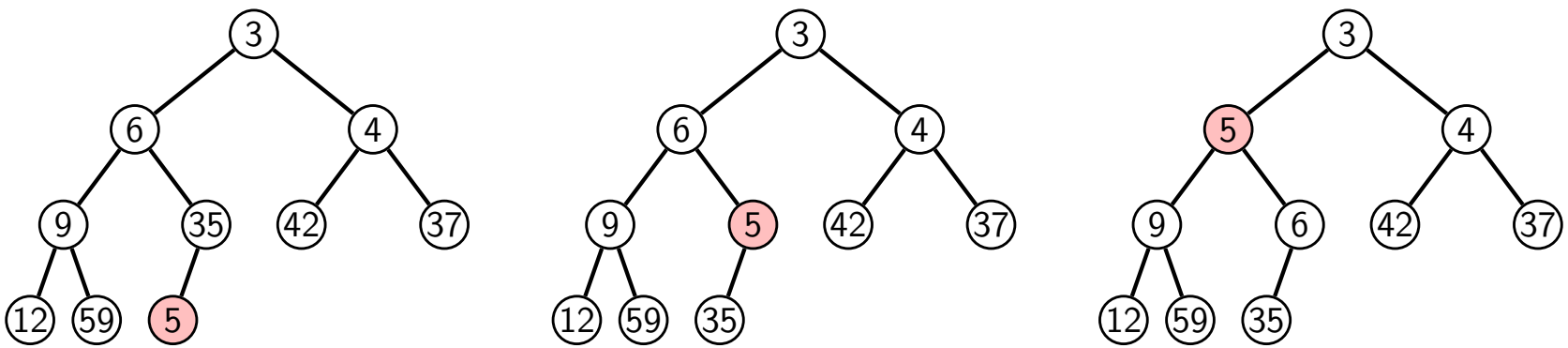
- Halda je dynamická datová struktura, která má „tvar" binárního stromu a uspořádání fronty
- Každý prvek haldy obsahuje hodnotu a dva potomky, podobně jako binární strom
- Vlastnosti haldy:
  - Hodnota každého prvku je menší než hodnota libovolného potomka
  - Každá úroveň haldy je plná, kromě poslední úrovně, která je zaplněna zleva doprava

    Binární plný strom
  - Prvky mohou být odebrány pouze přes kořenový uzel
- Vlastnost haldy zajišťuje, že kořen je vždy nejmenší prvek
- Nejmenší prvek je tedy první hodnota, kterou z haldy odebereme

Halda – přidání prvku **push**

- Po každém provedení operace **push** musí být splněny vlastnosti haldy
- Prvek přidáme na konec haldy, tj. na první volnou pozici (vlevo) na nejnižší úrovni haldy
- Zkontrolujeme, zdali je splněna podmínka haldy, pokud ne, zaměníme prvek s nadřazeným prvkem (předkem)

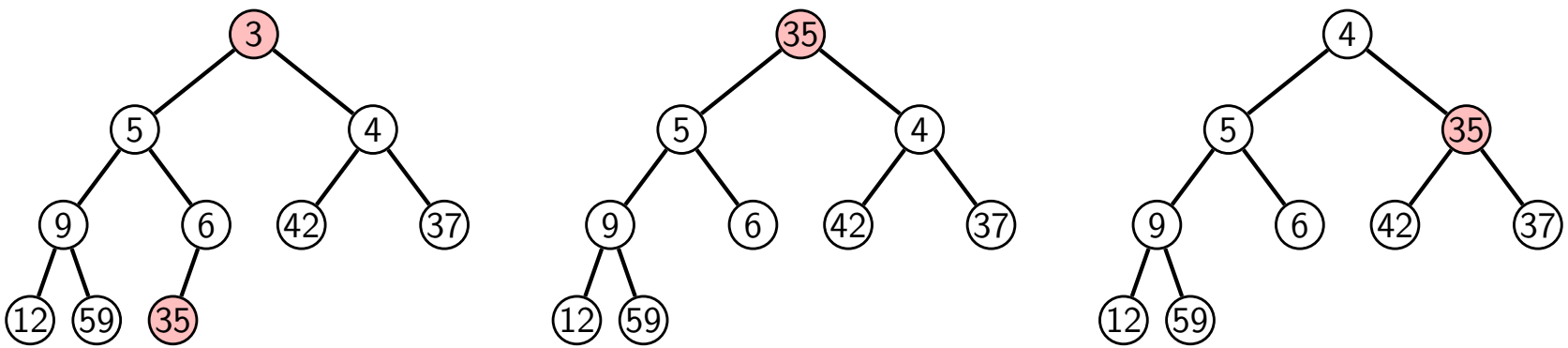
V nejhorším případě prvek „probublá" až do kořene stromu



Halda – odebrání prvku **pop**

- Při operaci **pop** odebereme kořen stromu
- Prázdné místo nahradíme nejpravějším listem (uzlem v poslední úrovni)
- Zkontrolujeme, zdali je splněna podmínka haldy, pokud ne, zaměníme prvek s potomkem a postup opakujeme

V nejhorším případě prvek „probublá" až do listu stromu



Prioritní fronta haldou

- Prvky ukládáme do haldy a při každém vložení / odebrání zajišťujeme, aby platily vlastnosti **haldy**
- Operace **peek** má konstantní složitost a nezáleží na počtu prvku ve frontě, nejmenší prvek je vždy kořen

  Asymptotická složitost v notaci velké O je $O(1)$.
- Operace **push** a **pop** udržují vlastnost haldy záměnami prvku až do hloubky stromu

  Pro binární plný strom je hloubka stromu $\log_2(n)$, *kde n je aktuální počet prvků ve stromu, odtud složitost operace* $O(log(n))$.

Informativní více v Algoritmizaci

Shrnutí přednášky

Diskutovaná témata

- Nelineární spojové struktury
  - Stromy a binární strom
- Abstraktní datový typ (ADT)
  - Datové struktury a jejich popis
  - Zásobník
  - Fronta
- Příklad ADT – prioritní fronta

Informativní

- Příště: Soubory a proudy, soubor jako posloupnost bytů