

Katedra informatiky  
Přírodovědecká fakulta  
Univerzita Palackého v Olomouci

# BAKALÁŘSKÁ PRÁCE

Demonstrace práce s datovými strukturami



2018

Vedoucí práce: Mgr. Tomáš Kühn,  
Ph.D.

Patrik Becher

Studijní obor: Aplikovaná informatika,  
prezenční forma

## **Bibliografické údaje**

Autor:	Patrik Becher
Název práce:	Demonstrace práce s datovými strukturami
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2018
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Tomáš Kühn, Ph.D.
Počet stran:	57
Přílohy:	1 CD/DVD
Jazyk práce:	český

## **Bibliographic info**

Author:	Patrik Becher
Title:	Data Structure Demonstration
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2018
Study field:	Applied Computer Science, full-time form
Supervisor:	Mgr. Tomáš Kühn, Ph.D.
Page count:	57
Supplements:	1 CD/DVD
Thesis language:	Czech

## Anotace

*Cílem bakalářské práce bylo vytvořit nástroj pro podporu výuky algoritmizace, konkrétně práce se základními stromovými datovými strukturami (binární vyhledávací stromy, AVL stromy, červeno-černé stromy). Výsledná aplikace podporuje vizualizaci vybraných datových struktur, včetně názorné demonstrace běžně prováděných operací s těmito datovými strukturami se souběžným zobrazením pseudokódu prováděné operace.*

## Synopsis

**Klíčová slova:** Binární vyhledávací stromy, Binární strom, AVL strom, Červeno-černý strom, Stromové animace Java, JavaFX

**Keywords:** Binary search trees, Binary tree, AVL tree, Red-black tree, Tree animations, Java, JavaFX

Rád bych poděkoval panu Mgr. Tomáši Kührovi Ph.D. za vedení této bakalářské práce a panu RNDr. Arnoštu Večerkovi za odbornou pomoc a poskytnuté materiály k práci. Dále bych chtěl poděkoval mé rodině a přítelkyni za podporu při tvorbě.

*Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.*

datum odevzdání práce

podpis autora

# Obsah

<b>1</b>	<b>Úvod</b>	<b>8</b>
<b>2</b>	<b>Stromy</b>	<b>9</b>
2.1	Binární strom . . . . .	11
2.2	Binární vyhledávací strom . . . . .	11
2.2.1	Vyhledávání . . . . .	12
2.2.2	Vkládání . . . . .	14
2.2.3	Odebírání . . . . .	17
2.3	AVL strom . . . . .	19
2.3.1	Rotace . . . . .	22
2.3.2	Vyhledávání . . . . .	23
2.3.3	Vkládání . . . . .	23
2.3.4	Odebírání . . . . .	24
2.4	Červeno-černý strom . . . . .	26
2.4.1	Transformace . . . . .	27
2.4.2	Vyhledávání . . . . .	29
2.4.3	Vkládání . . . . .	29
2.4.4	Odebírání . . . . .	30
<b>3</b>	<b>Programátorská dokumentace</b>	<b>36</b>
3.1	Programovací jazyk a použité technologie . . . . .	36
3.1.1	Java . . . . .	36
3.1.2	JavaFX . . . . .	36
3.1.3	FXML . . . . .	36
3.2	Architektura programu . . . . .	36
3.2.1	Balík <i>application</i> . . . . .	37
3.2.2	Balík <i>trees</i> . . . . .	38
3.2.3	Balík <i>graphic</i> . . . . .	42
<b>4</b>	<b>Uživatelská příručka</b>	<b>49</b>
4.1	Minimální požadavky aplikace . . . . .	49
4.2	Spuštění . . . . .	49
4.3	Okno aplikace . . . . .	49
4.3.1	Popis okna aplikace . . . . .	50
4.3.2	Klávesové zkratky . . . . .	53
	<b>Závěr</b>	<b>54</b>
	<b>Conclusions</b>	<b>55</b>
<b>A</b>	<b>Obsah přiloženého CD/DVD</b>	<b>56</b>
	<b>Literatura</b>	<b>57</b>

## Seznam obrázků

1	Příklady neorientovaných stromů . . . . .	9
2	Popis stromu . . . . .	10
3	Binární strom . . . . .	11
4	Rozdílné výšky stromu . . . . .	12
5	Postup vyhledávání hodnoty 5 . . . . .	13
6	Postup vkládání hodnoty 6 - hledání . . . . .	15
7	Postup vkládání hodnoty 6 - vložení . . . . .	15
8	Postup odebírání hodnoty 4 . . . . .	18
9	Vyvážený strom . . . . .	20
10	Výpočet faktoru vyvážení pro uzel $u$ . . . . .	20
11	Ukázka vyváženého AVL stromu . . . . .	21
12	Rotace RR . . . . .	22
13	Rotace LL . . . . .	22
14	Rotace RL . . . . .	23
15	Rotace LR . . . . .	23
16	Postup vkládání hodnoty 5 . . . . .	24
17	Postup odebírání hodnoty 1 . . . . .	26
18	Ukázka vyváženého ČČ stromu . . . . .	26
19	Rotace RR . . . . .	27
20	Rotace LL . . . . .	27
21	Rotace RL . . . . .	28
22	Rotace LR . . . . .	28
23	Přebarvení 1 . . . . .	29
24	Přebarvení 2 . . . . .	29
25	Postup vkládání hodnoty 10 . . . . .	30
26	Odstranění dvojité obarveného uzlu – jednoduchá rotace . . . . .	32
27	Odstranění dvojité obarveného uzlu – dvojitá rotace . . . . .	32
28	Odstranění dvojité obarveného uzlu – přebarvení . . . . .	33
29	Odstranění dvojité obarveného uzlu – přebarvení a přesun označení . . . . .	33
30	Rotace s dvojitě obarveným uzlem – jednoduchá rotace . . . . .	34
31	Ukázka odebírání – 1) odebrání 8 . . . . .	34
32	Ukázka odebírání – 2) vložení NULL list . . . . .	35
33	Ukázka odebírání – 3) případ 4 . . . . .	35
34	Ukázka odebírání – 4) případ 2 . . . . .	35
35	Hlavní okno programu. . . . .	49
36	Menu programu. . . . .	51
37	Dialogové okno při změně stromu. . . . .	52
38	Dialogové okno kliknutí na <i>Nový</i> v menu. . . . .	52
39	Dialogové okno při načítání stromu. . . . .	52
40	Chyba při načítání uloženého stromu. . . . .	52
41	Chyba při ukládání stromu. . . . .	53
42	Dialogové okno navázané na tlačítko <i>Nový</i> .... . . . .	53

43	Dialogové okno pro náhodný strom. . . . .	53
----	-------------------------------------------	----

## Seznam zdrojových kódů

1	search . . . . .	13
2	Result - Třída . . . . .	15
3	search - úprava . . . . .	16
4	insert . . . . .	16
5	delete . . . . .	19
6	computeFactor . . . . .	21
7	styles.css - ukázka . . . . .	38
8	countChildren() . . . . .	44
9	Ukázka použití konstruktoru třídy DrawingTree . . . . .	45
10	Ukázka určení souřadnic nově vloženého uzlu. . . . .	47
11	Zkrácená ukázka nextAnimation() . . . . .	48
12	Ukázka výpočtu správného umístění uzlu. . . . .	48

# 1 Úvod

Tato aplikace vznikla za účelem výuky základních binárních stromů. Obsahuje podporu pro *Binární vyhledávací*, *AVL* a *Červenočerné stromy*. Program pomocí animací zobrazuje operace: *Vyhledávání*, *Vkládání* a *Odebírání* prvků ze stromů. Souběžně s animací zobrazuje stručný pseudokód aktuálně prováděné operace. Dále umožňuje *Opakovat poslední operaci* a *Generování náhodných stromů*.

Text samotné práce se dělí na tři části: *Teoretickou část* – zabývá se teorií vybraných stromů, *programovací část* – popisuje architekturu a samotnou implementaci programu, a *část s uživatelskou příručkou* – zde je vysvětleno uživatelské prostředí a funkcionality.

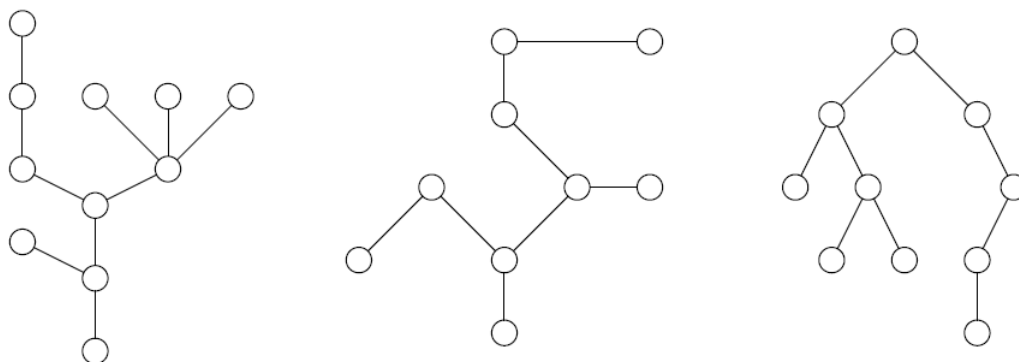


## 2 Stromy

V kapitole jsou vysvětleny základní pojmy, které jsou nezbytné k pochopení vlastností *stromů* obsažených v aplikaci. V podkapitolách jsou osvětleny principy pro tvorbu a následnou práci s konkrétními *binárními vyhledávacími stromy*. Využitím těchto principů byl naprogramován tento výukový nástroj.

### Definice 1 (Strom)

*Strom* je neorientovaný <sup>1</sup> souvislý <sup>2</sup> graf bez kružnic <sup>3</sup> [1].



Obrázek 1: Příklady neorientovaných stromů

Strom je datová struktura, která představuje stromovou strukturu propojených *uzlů* <sup>4</sup>. Uzly jsou mezi sebou vzájemně spojeny pomocí *hran* <sup>5</sup>. Strom složený z uzlů  $U$  má  $|U - 1|$  hran.

### Definice 2 (Kořenový strom)

*Kořenový strom* je strom, ve kterém je vybrán jeden vrchol (kořen). Může to být kterýkoliv vrchol. Bývá to ale vrchol, který je v nějakém smyslu na vrcholu hierarchie objektů, která je stromem reprezentována. [1]

---

<sup>1</sup>Mezi každými dvěma vrcholy existuje právě jedna cesta.

<sup>2</sup>Vynecháním libovolné hrany vznikne nesouvislý graf.

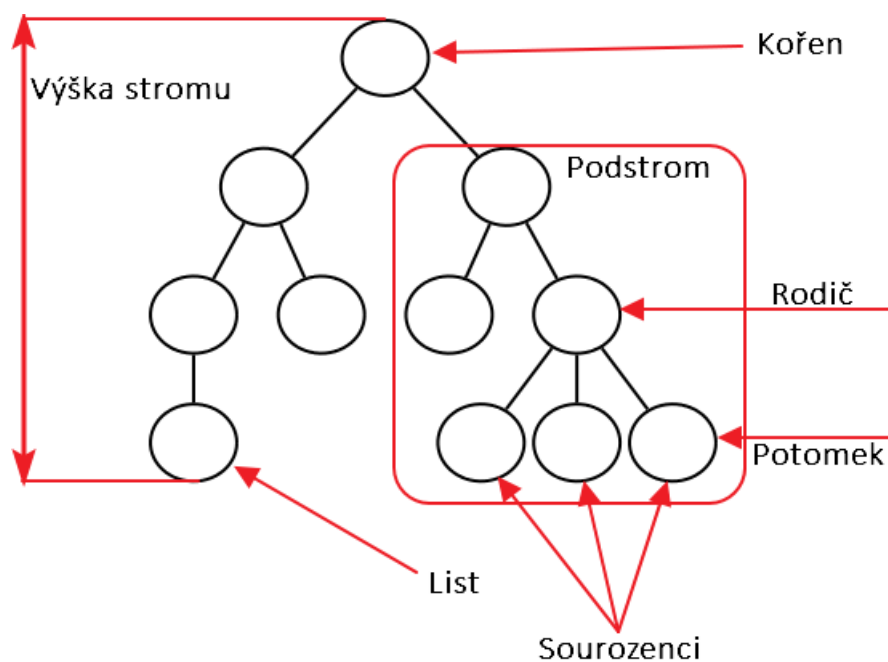
<sup>3</sup>Přidáním jakékoli hrany vznikne graf s kružnicí.

<sup>4</sup>Prvek obsahující hodnotu.

<sup>5</sup>Představuje cestu mezi spojenými uzly.

### Důležité pojmy:

- **Uzel** – Jednoduše jakýkoliv prvek stromu.
- **Kořen** – Jeden konkrétní uzel, který se nachází na vrcholu stromu. Pouze tento uzel nemá *rodiče*.
- **Potomek, následník** – Uzel, který je hranou přímo připojen k jinému uzlu, cestou od kořene.
- **Rodič, předchůdce** – Uzel, který má alespoň jednoho potomka.
- **Sourozenci** – Skupina uzlů, které mají stejného rodiče.
- **Podstrom** – Část stromu, která je úplným stromem s tím, že kořen tohoto podstromu má svého rodiče.
- **Koncový uzel, list** – Uzel bez potomků.
- **Výška stromu** – Nejdelší délka cesty od kořene k uzlu.



Obrázek 2: Popis stromu

### Definice 3 ( $m$ -ární stromy)

Kořenový strom se nazývá  $m$ -ární, právě když každý jeho vrchol má nejvýše  $m$  potomků. 2-ární strom se nazývá binární. Kořenový strom se nazývá úplný  $m$ -ární, právě když každý jeho vrchol nemá buď žádného nebo má právě  $m$  potomků.[2]

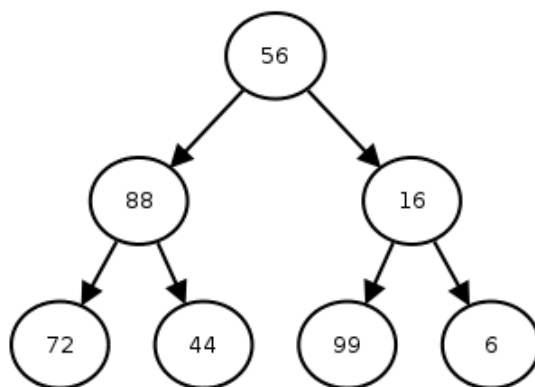
## 2.1 Binární strom

### Definice 4 (Binární strom)

*Binární strom* je typ kořenových stromů, ve kterém každý obsažený uzel má maximálně 2 potomky.

Každý uzel obsahuje vlastnosti:

- **Klíč** – Hodnota uložená v uzlu.
- **Ukazatel na levého potomka**
- **Ukazatel na pravého potomka**
- **Ukazatel na jednoho rodiče** – Tento ukazatel není povinný.



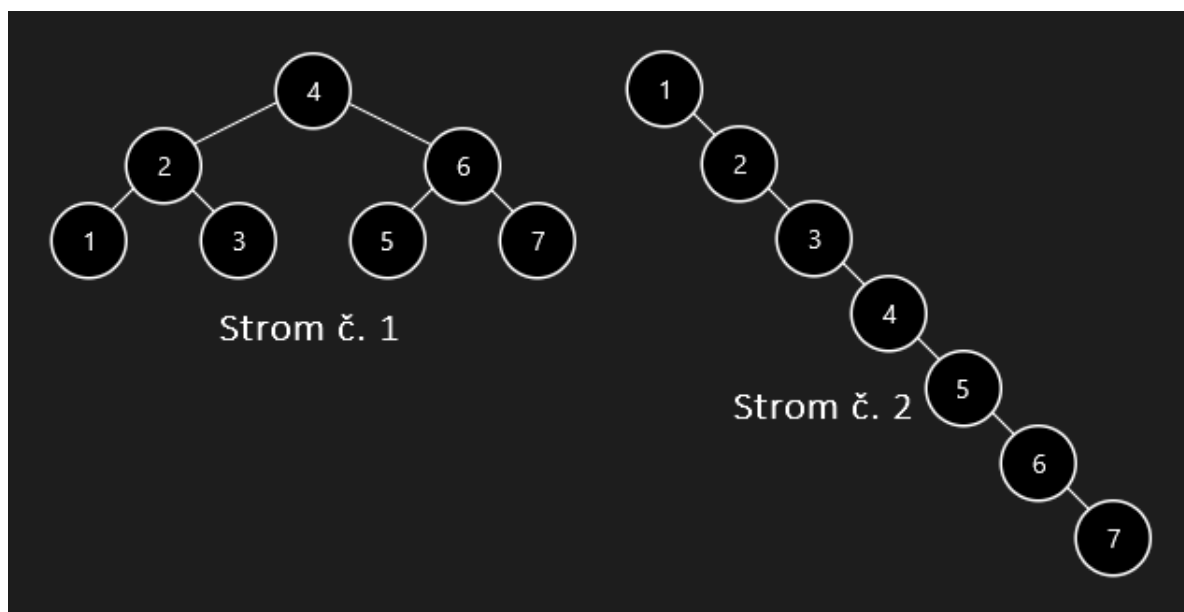
Obrázek 3: Binární strom

## 2.2 Binární vyhledávací strom

Binární *vyhledávací* strom (zkratka BVS) je speciální typ binárního stromu, kde platí následující:

- Každý **pravý** potomek  $P$  rodiče  $R$  má vyšší hodnotu  $h$  než jeho rodič. Platí tedy:  $P.h > R.h \Rightarrow$  Pravý podstrom uzlu  $R$  obsahuje pouze uzly, které mají vyšší hodnotu než uzel  $R$ .
- Každý **levý** potomek  $L$  rodiče  $R$  má nižší hodnotu  $h$  než jeho rodič. Platí tedy:  $P.h > L.h \Rightarrow$  Levý podstrom uzlu  $R$  obsahuje pouze uzly, které mají nižší hodnotu než uzel  $R$ .
- Ve stromě se nenachází dva uzly se stejnou hodnotou.

Toto uspořádání uzlů v BVS usnadňuje vyhledávání. Operace nad BVS stromem s výškou  $h$  mají časovou složitost  $\theta(h)$ . V nejhorším případě může mít BVS výšku rovnu  $n - 1$ , kde  $n$  je počet uzlů. Oba případy jsou zobrazeny na obrázku 4.



Obrázek 4: Rozdílné výšky stromu

### 2.2.1 Vyhledávání

Operace *vyhledávání* patří k nejčastěji používané operaci s BVS. Při *vyhledávání* je potřeba zadat hodnotu  $x$ , kterou chceme vyhledat. Postupně dochází k porovnávání hodnot uzlů s  $x$ . Výsledkem vyhledávání je buď uzel, který obsahuje hodnotu  $x$  nebo takový uzel neexistuje.

#### Přesný postup vyhledávání:

##### 1. krok – počáteční

Na začátku vyhledávání je třeba určit aktuální uzel, který označíme  $u$ . Hledanou hodnotu označíme  $x$ .

V tomto kroku  $u$  bude kořen stromu, pokud strom nemá kořen, hodnota  $x$  je nenalezena a tím vyhledávání končí.

##### 2. krok – průběžný

Zde dochází k porovnávání  $x$  a hodnoty aktuálního uzlu  $u$ . Hodnotu  $u$  označíme  $u.h$ .

Pokud je  $u$  prázdný, vyhledávání končí neúspěchem.

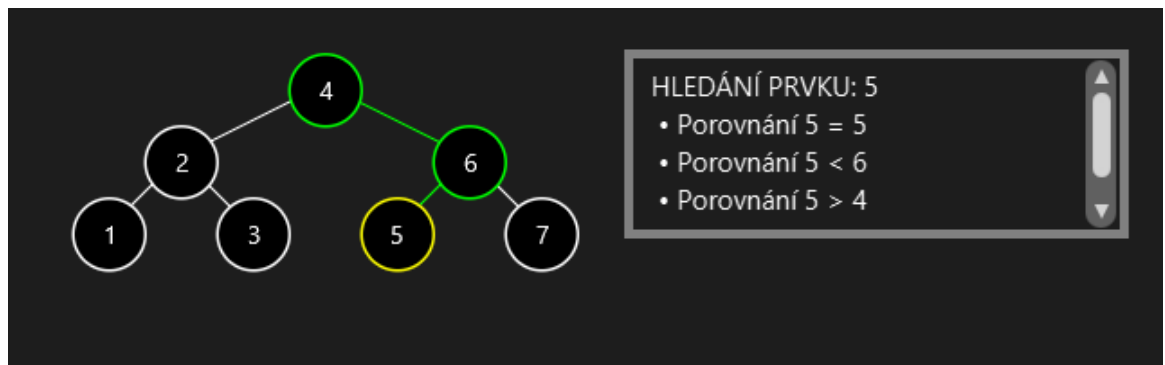
#### Při porovnávání mohou nastat tyto možnosti:

- $x > u.h$

V tomto případě jako  $u$  nastavíme pravého potomka  $u$ . A znovu provedeme 2. krok.

- $x < u.h$   
V tomto případě jako  $u$  nastavíme levého potomka  $u$ . A znovu provedeme 2. krok.
- $x = u.h$   
Hledaná hodnota  $x$  byla nalezena v  $u$ . Vyhledávání tedy končí.

Na obrázku 5 je zvýrazněna cesta průchodů stromem při vyhledávání uzlu s hodnotou 5. Na obrázku je i zaznamenána historie porovnávání.



Obrázek 5: Postup vyhledávání hodnoty 5

### Zdrojový kód vyhledávání v jazyku Java:

```

1 Node search(int value, Node node) { //value je hledaná hodnota, node
    je při prvním volání kořen
2     if (node == null) {
3         return null; //uzel nebyl nalezen
4     }
5
6     if (value > node.getValue()) { //pokud je hledaná hodnota vyšší než
        má aktuální uzel
7         search(value, node.getRight()); //nastavím uzel na pravého potomka
8     } else if (value < node.getValue()) { //pokud je hledaná hodnota
        vyšší než má aktuální uzel
9         search(value, node.getLeft()); //nastavím uzel na levého potomka
10    } else { //pokud není vyšší ani nižší, tak se musí rovnat
11        return node; //vrátím nalezený uzel
12    }
13 }
```

Zdrojový kód 1: search

### 2.2.2 Vkládání

Při  *vkládání*  zadané hodnoty  $x$  je nejprve nutné prohledat strom, jestli se zde  $x$  již nenachází. Pokud je nalezen uzel s hodnotou  $x$ , je výsledkem operace nalezený uzel. V případě, že daný strom tento uzel neobsahuje, je jako potomek posledního prohledávaného uzlu vložen nový uzel s hodnotou  $x$ .

#### Přesný postup vkládání:

**1. krok** – počáteční

Na začátku vkládání je třeba určit aktuální uzel, který označíme  $u$ . Vkládanou hodnotu označíme  $x$ .

V tomto kroku  $u$  bude kořen stromu. Pokud strom nemá kořen, vytvoříme nový uzel s hodnotou  $x$  a ten vložíme do stromu. Uzel se stane kořenem stromu, čímž vkládání končí.

**2. krok** – vyhledávání, vkládání

Před vkládáním je nejprve nutno ověřit, zda se ve stromu nenachází uzel s hodnotou  $x$ .

#### Vyhledávání může dopadnout těmito způsoby:

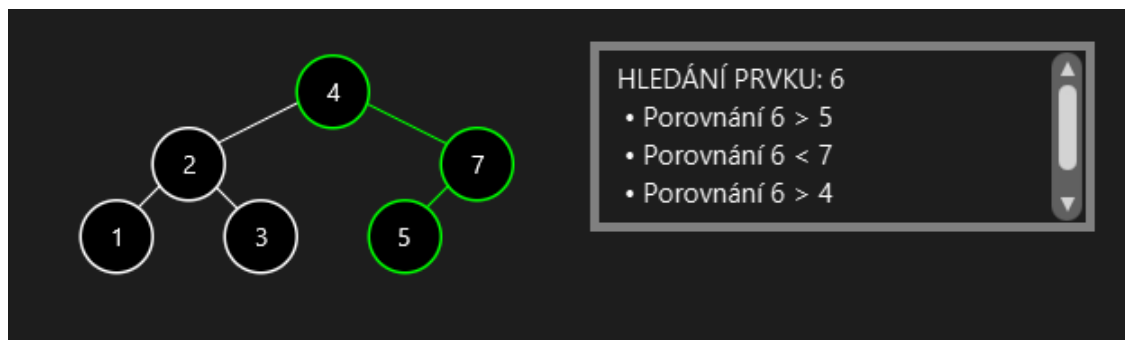
- Prvek byl nalezen.  
Pokud byl uzel s hodnotou  $x$  nalezen, vkládání končí neúspěchem.
- Prvek nebyl nalezen, přičemž platí  $x > u.h^6$ .  
Vytvoříme nový uzel s hodnotou  $x$  a vložíme jako pravého potomka  $u^7$ .
- Prvek nebyl nalezen, přičemž platí  $x < u.h^6$ .  
Vytvoříme nový uzel s hodnotou  $x$  a vložíme jako levého potomka  $u^7$ .

Na obrázku 6 je zvýrazněna cesta průchodů stromem při vyhledávání uzlu s hodnotou 6. Samotné vložení je na obrázku 7.

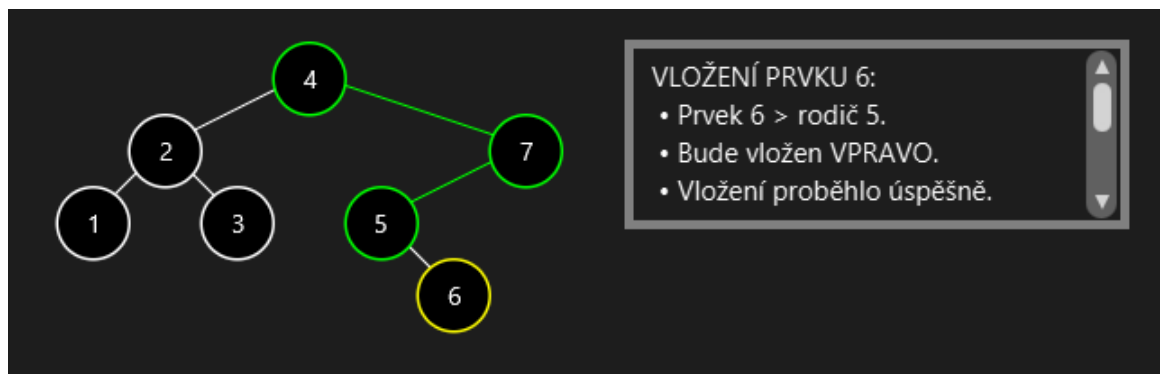
---

<sup>6</sup>Hodnota posledního navštíveného uzlu při hledání.

<sup>7</sup>Poslední navštívený uzel při hledání.



Obrázek 6: Postup vkládání hodnoty 6 - hledání



Obrázek 7: Postup vkládání hodnoty 6 - vložení

### Zdrojový kód vkládání v jazyku Java:

Před samotnou funkcí insert je třeba vytvořit třídu Result, která bude dále použita:

```

1 Class Result() {
2     private Node node; //poslední navštívený uzel (hledaný/rodič)
3     private boolean isFind; //atribut pro určení zda byl uzel nalezen
4
5     public Result(Node node, boolean isFind) { //konstruktor
6         ...
7     }
8     ...
9 }

```

Zdrojový kód 2: Result - Třída

Dále je potřeba trochu poupravit již známou funkci `search` 1:

```
1 Result search(int value, Node node) { //value je hledaná hodnota,  
    node je při prvním volání kořen  
2     if (value > node.getValue()) {  
3         if (node.getRight() != null) {  
4             search(value, node.getRight()); //pokud má pravého potomka  
5         } else {  
6             return new Result(node, false); //pokud nemá pravého potomka,  
                node = rodič vkládaného  
7         }  
8     } else if (value < node.getValue()) {  
9         if (node.getLeft() != null) {  
10            search(value, node.getLeft()); //pokud má levého potomka  
11        } else {  
12            return new Result(node, false); //pokud nemá levého potomka,  
                node = rodič vkládaného  
13        }  
14    } else {  
15        return new Result(node, true);  
16    }  
17 }
```

Zdrojový kód 3: `search` - úprava

Nyní funkce `insert`:

```
1 Node insert(int value) {  
2     Result result = search(value); //nejprve vyhledáme value  
3     if (result.isFind()) { //pokud byl uzel s danou hodnotou nalezen  
4         return result.getNode();  
5     } else { //dále vkládáme nově vytvořený uzel:  
6         if (value > result.getNode().getValue()) {  
7             result.getNode().setRight(new Node(value)); //bude pravý potomek  
8         } else {  
9             result.getNode().setLeft(new Node(value)); //bude levý potomek  
10        }  
11    }  
12    return null;  
}
```

Zdrojový kód 4: `insert`



### 2.2.3 Odebírání

Při *odebírání* zadané hodnoty  $x$  je stejně jako u vkládání nejprve nutné prohledat strom, zda se odebíraný uzel s hodnotou  $x$  ve stromě nachází. Pokud je uzel nalezen, je následně smazán a struktura stromu případně upravena.

#### Přesný postup odebírání:

**1. krok** – počáteční

Na začátku odebírání je třeba určit aktuální uzel, který označíme  $u$ . Odebíranou hodnotu označíme  $x$ .

V tomto kroku  $u$  bude kořen stromu. Pokud strom nemá kořen odebírání končí, uzel s hodnotou  $x$  nebyl nalezen.

**2. krok** – vyhledávání

Před odebíráním je nejprve nutno uzel s hodnotou  $x$  vyhledat.

#### Vyhledávání může dopadnout těmito způsoby:

- Prvek nebyl nalezen.  
Pokud se uzel s hodnotou  $x$  ve stromě nenachází, odebírání končí neúspěchem.
- Prvek byl nalezen.  
Nyní můžeme nalezený uzel  $u$  odebrat.

**3. krok** – odebírání

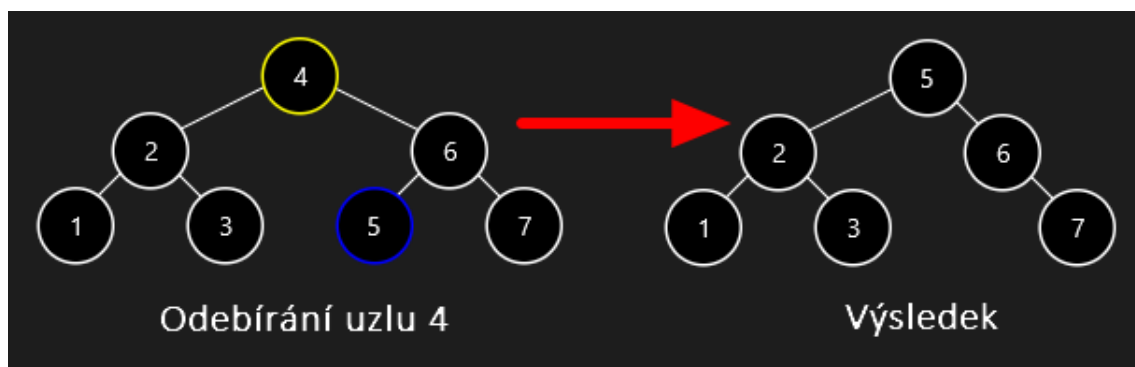
#### Odebírání $u$ má tyto možnosti:

- Pokud  $u$  je list<sup>8</sup>.  
List  $u$  může být odebrán.
- $u$  má jednoho potomka.  
 $u$  bude nahrazen podstromem potomka.
- $u$  má dva potomky.
  - $u$  bude nahrazen *nejlevějším* prvkem z *pravého* podstromu.
  - $u$  bude nahrazen *nejpravější* prvkem z *levého* podstromu.

V následujícím obrázku 8 je ukázka mazání uzlu s hodnotou 4, který je následně nahrazen uzlem s hodnotou 5, který je *nejlevější* prvek z *pravého* podstromu.

---

<sup>8</sup>Nemá žádné potomky.



Obrázek 8: Postup odebírání hodnoty 4

Zdrojový kód<sup>9</sup> odebírání v jazyku Java:

---

<sup>9</sup>Tento kód je pouze zjednodušená implementace k lepšímu pochopení odebírání.

```

1 boolean delete(int value) {
2     Node removeNode = result.getNode();
3     Node helpNode; //pomocná proměnná
4
5     Result result = search(value); //vyhledám hodnotu
6
7     if (!result.isFind()) { //pokud ho nenajdu
8         return false;
9     }
10
11     if ((removedNode.getLeft() != null) && (removedNode.getRight() !=
12         null)) { //Pokud má dva potomky
13         helpNode = removedNode.getRight(); //dosadím pravého potomka
14
15         while (helpNode.getLeft() != null) { //a hledám nejlevějšího
16             helpNode = helpNode.getLeft();
17         }
18         removeNode.setNode(helpNode);
19     } else if (removedNode.getLeft() != null) { //pouze levý potomek
20         removeNode.setNode(removedNode.getLeft());
21     } else if (removedNode.getRight() != null) { //pouze pravý potomek
22         removeNode.setNode(removedNode.getRight());
23     } else { //pokud je to list
24         removeNode.delete(); //odstraním list ze stromu
25     }
26
27     return true;
28 }

```

Zdrojový kód 5: delete

## 2.3 AVL strom

Určitým způsobem by se dalo tvrdit, že *AVL strom* je binární vyhledávací strom, který má ale jednu zásadní odlišnost. Složitost všech operací u BVS je závislá na výšce stromu, je tedy žádoucí udržovat výšku stromu co nejnižší. *AVL strom* je tedy vyvážený binární vyhledávací strom.

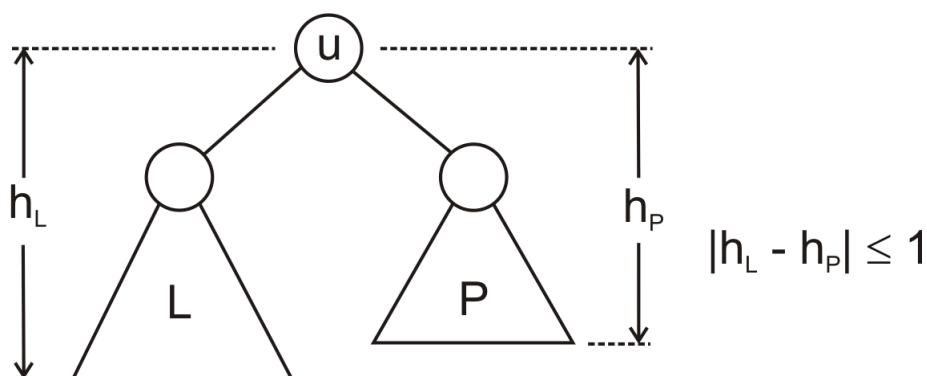
### Definice 5 (Vyvážený strom)

Strom je vyvážený tehdy a jen tehdy, je-li rozdíl výšek každého uzlu nejvýše 1. [6][4]

Kvůli udržování vyváženosti stromu mají operace *ukládání* a *odebírání* složitost  $\theta(\log(n))$ .

V následujícím obrázku 9 je předchozí definice o vyváženém stromu názorně vysvětlena. Nejprve je třeba zavést pojmy:

- $u$  – aktuální uzel.
- $L$  – levý podstrom  $u$ .
- $P$  – pravý podstrom  $u$ .
- $h_L$  – výška levého podstromu.
- $h_P$  – výška pravého podstromu.

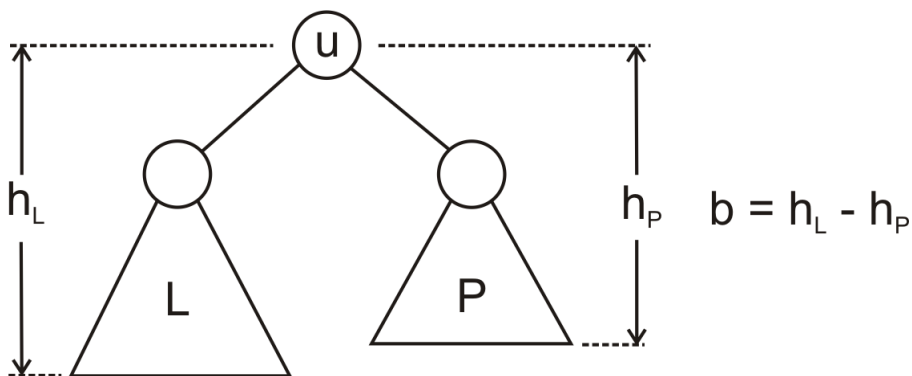


Obrázek 9: Vyvážený strom

### Faktor vyvážení

Pro kontrolu dodržení pravidla o vyvážení je třeba pro každý uzel zavést novou vlastnost *faktor vyvážení uzlu*. Tuto vlastnost budeme značit  $b$  (anglicky *balance*).  $b$  obsahuje informaci o aktuálním vyvážení daného podstromu.

Nabývá hodnot  $\langle -2, 2 \rangle$ , přičemž uzel je vyvážený pokud jeho faktor  $b$  je 1, 0 nebo -1. Při operaci přidávání nebo odebírání může být  $b$  2 nebo -2, pak je ale nutné provést transformaci, která dosáhne vyvážení daného uzlu.

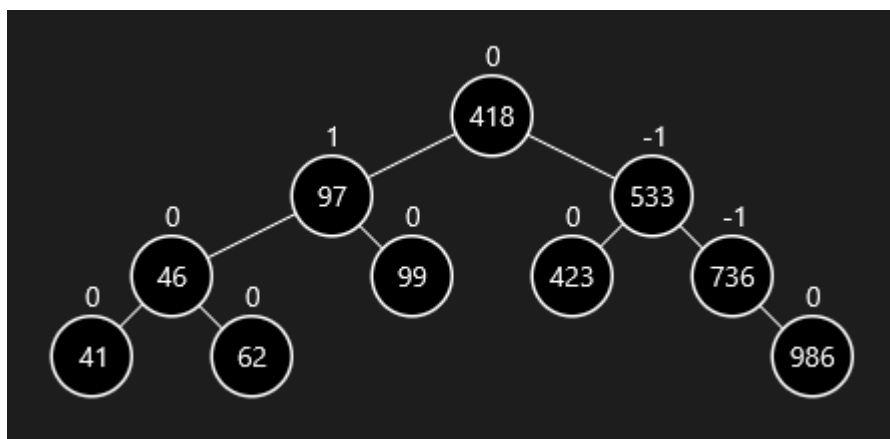


Obrázek 10: Výpočet faktoru vyvážení pro uzel  $u$

## Zdrojový kód rekurzivního výpočtu faktoru v jazyku Java:

```
1 int computeFactor() {
2     int lHeight = 0; //výška levého podstromu
3     int rHeight = 0; //výška pravého podstromu
4
5     if (left != null) { //pokud má levý podstrom
6         lHeight = left.computeFactor();
7     }
8
9     if (right != null) { //pokud má pravý podstrom
10        rHeight = right.computeFactor();
11    }
12
13    factor = lHeight - rHeight;
14
15    return Math.max(rHeight, lHeight) + 1;
16 }
```

Zdrojový kód 6: computeFactor



Obrázek 11: Ukázka vyváženého AVL stromu

Každý uzel ALV stromu obsahuje tyto vlastnosti:

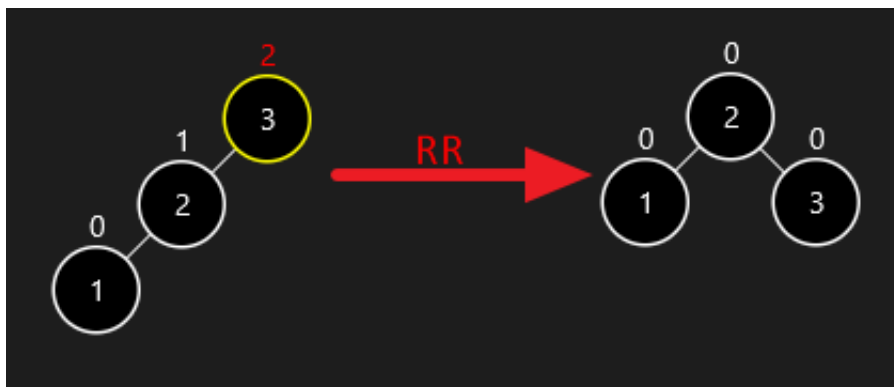
- **Klíč** – Hodnota uložená v uzlu.
- **Faktor vyvážení** – Faktor vyvážení daného uzlu.
- **Ukazatel na levého potomka**
- **Ukazatel na pravého potomka**
- **Ukazatel na jednoho rodiče**

### 2.3.1 Rotace

K obnově vyváženosti uzlů se používají *rotace*. *Rotace* pouze změní ukazatele uzlů v nevyvážené části stromu, aby došlo k opětovnému vyvážení. V AVL stromu se k vyvažování podle typu nevyvážené části stromu používají tyto rotace:

#### Jednoduchá rotace RR

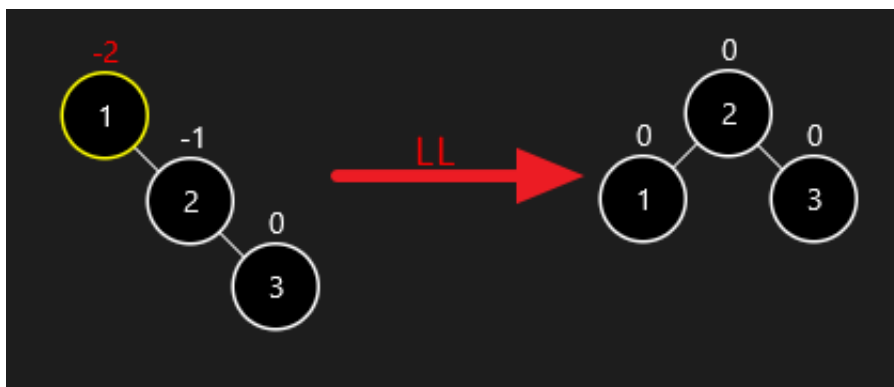
Nevyvážený uzel má  $b = 2$ , jeho *levý potomek* má  $b = 0$  nebo  $b = 1$ .



Obrázek 12: Rotace RR

#### Jednoduchá rotace LL

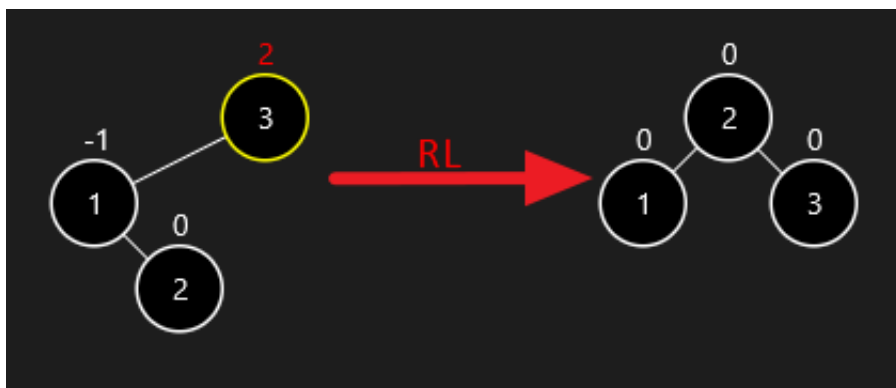
Nevyvážený uzel má  $b = -2$ , jeho *pravý potomek* má  $b = -1$  nebo  $b = 0$ .



Obrázek 13: Rotace LL

### Dvojitá rotace RL

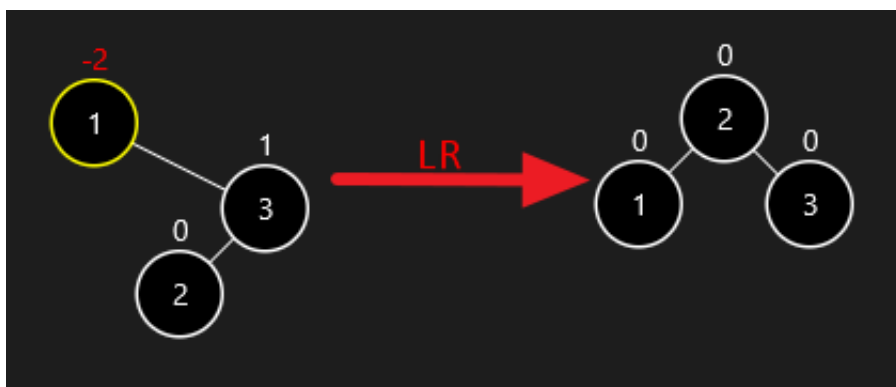
Nevyvážený uzel má  $b = 2$ , jeho *levý potomek* má  $b = 1$ .



Obrázek 14: Rotace RL

### Dvojitá rotace LR

Nevyvážený uzel má  $b = -2$ , jeho *pravý potomek* má  $b = 1$ .



Obrázek 15: Rotace LR

## 2.3.2 Vyhledávání

*Vyhledávání* u AVL stromu se nijak neliší od toho v BVS.

## 2.3.3 Vkládání

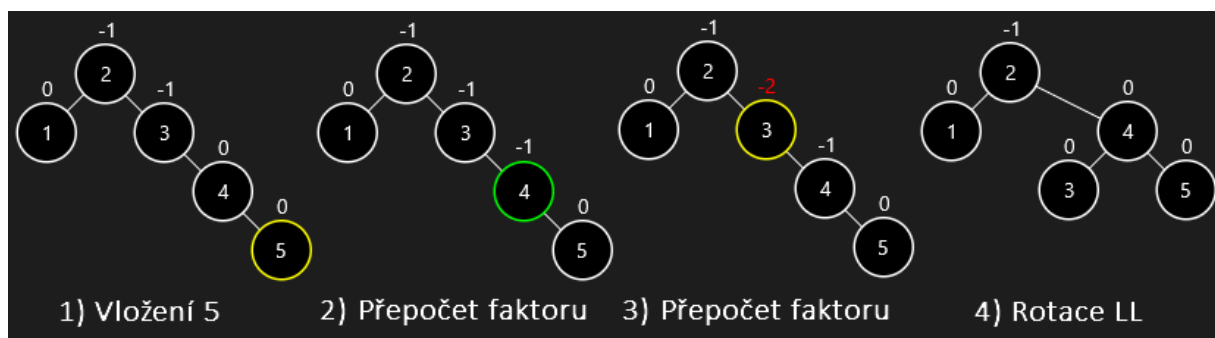
*Vkládání* je založeno na *vkládání* u BVS. Navíc je zde přidán krok aktualizace faktoru vyvážení, kde od nově vloženého uzlu postupně u určitých uzlů přepočítáváme  $b$ . Pokud se strom kvůli vložení nového uzlu stává nevyváženým, provedeme jednu z rotací, čímž daný strom vyvážíme.

### Přesný postup vkládání:

- 1. krok** – počáteční  
Stejně jako u BVS.
- 2. krok** – vyhledávání, vkládání  
Novému uzlu po vložení nastavíme faktor vyvážení na 0 a  $u$  nastavíme na předchůdce nově vloženého uzlu.  
Pokud nově vložený uzel je kořen, vkládání úspěšně končí.
- 3. krok** – výpočet faktoru vyvážení  
V tomto kroku v aktuálním uzlu  $u$  aktualizujeme faktor vyvážení  $b$ .

### Po aktualizaci $u.b$ mohou nastat tyto případy:

- $u.b = 1$  nebo  $u.b = -1$   
Pokud  $u$  je kořen, vkládání úspěšně končí. Jinak  $u$  nastavíme na rodiče  $u$  a znovu opakujeme krok 3.
- $u.b = 2$  nebo  $u.b = -2$   
Provede se příslušná rotace.  
Pokud po provedení rotace je  $u.b = 0$  nebo je nyní  $u$  kořen, vkládání úspěšně končí. Jinak  $u$  nastavíme na předchůdce  $u$  a znovu opakujeme krok 3.
- $u.b = 0$   
Vkládání úspěšně končí.



Obrázek 16: Postup vkládání hodnoty 5

### 2.3.4 Odebírání

*Odebírání* je založeno na stejnojmenné operaci u BVS. Navíc je zde jako u *vkládání* přidán krok aktualizace faktoru vyvážení, kde od odstraněného uzlu postupně u určitých uzlů přepočítáváme  $b$ . Pokud se strom kvůli odebrání uzlu stává nevyváženým, tak provedeme jednu z rotací, čímž daný strom vyvážíme.



### Přesný postup odebírání:

1. **krok** – počáteční  
Stejně jako u BVS.
2. **krok** – vyhledávání  
Stejně jako u BVS.
3. **krok** – odebírání

#### Odebírání $u$ má tyto možnosti:

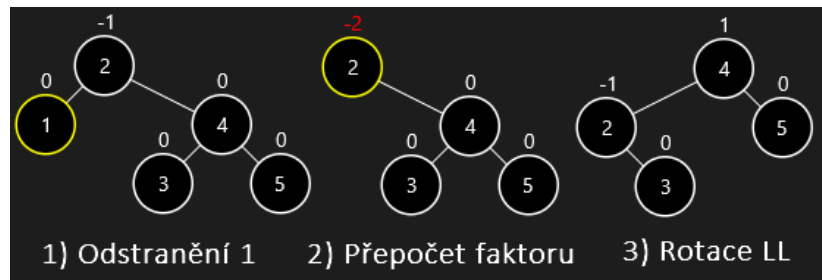
- Pokud  $u$  je list.  
List  $u$  může být odebrán.  
Pokud byl  $u$  kořen, odebírání úspěšně končí. Jinak nastavíme  $u$  na rodiče  $u$ .
- $u$  má jednoho potomka.  
 $u$  bude nahrazen podstromem potomka. Pokud byl  $u$  kořen, odebírání úspěšně končí. Jinak nastavíme  $u$  na rodiče  $u$ .
- $u$  má dva potomky.
  - $u$  bude nahrazen *nejlevějším* prvkem z *pravého* podstromu, zároveň tento prvek získá faktor vyvážení z  $u$ .
  - $u$  bude nahrazen *nejpravějším* prvkem z *levého* podstromu, zároveň tento prvek získá faktor vyvážení z  $u$ .

Rodiče uzlu, který nahradil  $u$ , uložíme do  $u$ .

4. **krok** – výpočet faktoru vyvážení  
V tomto kroku v aktuálním uzlu  $u$  aktualizujeme faktor vyvážení  $b$ .

#### Po aktualizaci $u.b$ mohou nastat tyto případy:

- $u.b = \langle -1, 1 \rangle$   
Pokud  $u$  je kořen, odebírání úspěšně končí. Jinak  $u$  nastavíme na rodiče  $u$  a znovu opakujeme krok 4.
- $u.b = 2$  nebo  $u.b = -2$   
Provede se příslušná rotace.  
Pokud po provedení rotace je  $u.b = 0$  nebo je nyní  $u$  kořen, odebírání úspěšně končí. Jinak  $u$  nastavíme na předchůdce  $u$  a znovu opakujeme krok 4.



Obrázek 17: Postup odebrání hodnoty 1

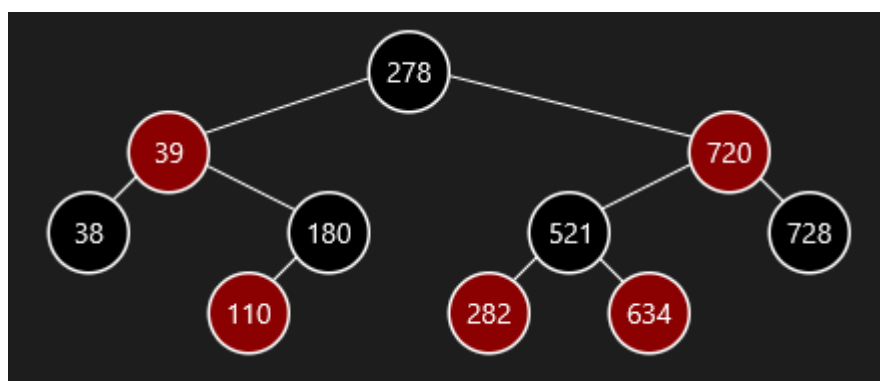
## 2.4 Červeno-černý strom

*Červeno-černý strom* (zkratka ČČ) vyvážený BVS. Na rozdíl od AVL stromu je ČČ strom vyvažován na základě barevného označení uzlů. Tímto vyvažováním má časovou složitost operací *ukládání* a *odebírání*  $\theta(\log(n))$ .

„Červeno-černý strom zajišťuje, že žádná cesta z kořene do libovolného listu stromu nebude dvakrát delší než kterákoli jiná, to znamená, že strom je přibližně vyvážený.“<sup>[4]</sup>

**ČČ strom musí splňovat tyto vlastnosti:**

- Každý uzel má červenou nebo černou barvu.
- Kořen stromu má vždy barvu černou.
- Ve stromu se nenacházejí dva po sobě jdoucí červené uzly.
- Každý červený uzel, který není list, má dva černé potomky.
- Cesta od kořene ke všem listům obsahuje vždy stejný počet černých uzlů.<sup>10</sup>



Obrázek 18: Ukázka vyváženého ČČ stromu

<sup>10</sup>Je zde započítán i samotný list, pokud má černou barvu.

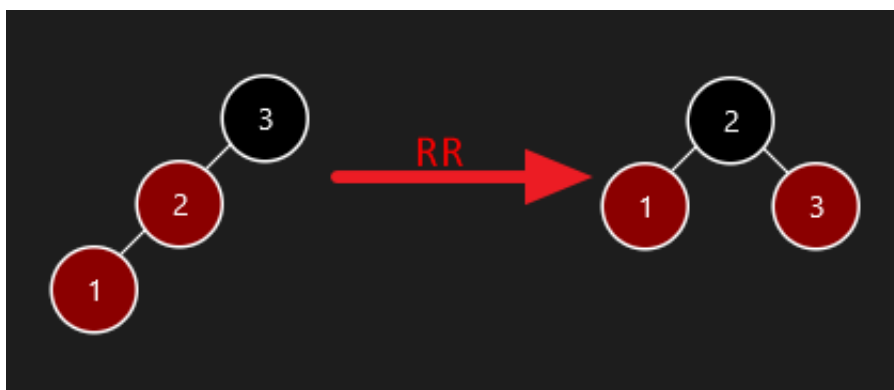
Každý uzel ČČ stromu obsahuje tyto vlastnosti:

- Klíč – Hodnota uložená v uzlu.
- Obarvení uzlu – Barva uzlu.
- Ukazatel na levého potomka
- Ukazatel na pravého potomka
- Ukazatel na jednoho rodiče

#### 2.4.1 Transformace

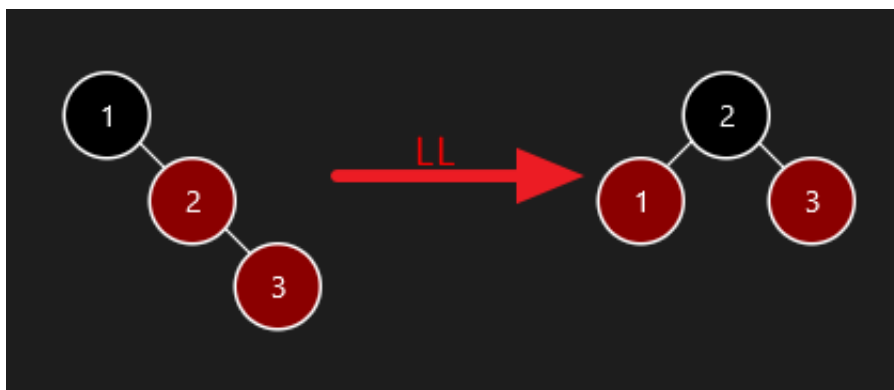
K obnově vyváženosti uzlů se používají *transformace*. Používají se zde stejně jako u AVL stromu *Rotace* a navíc je zde nová transformace *přebarvení*. Tyto *transformace* se používají k opětovnému vyvážení nevyváženého stromu.

##### Jednoduchá rotace RR



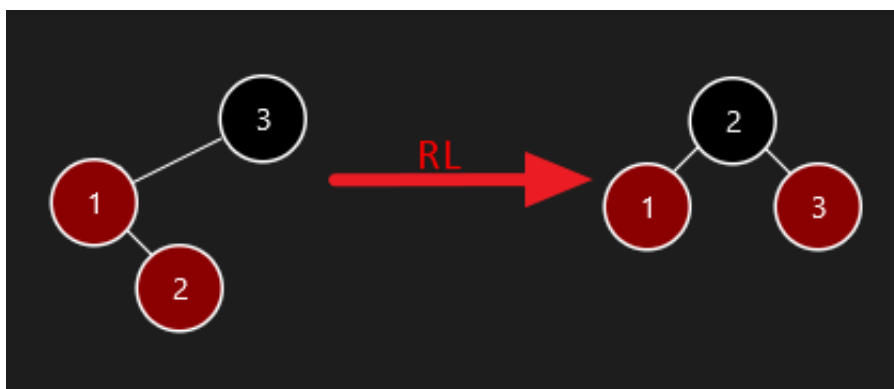
Obrázek 19: Rotace RR

##### Jednoduchá rotace LL



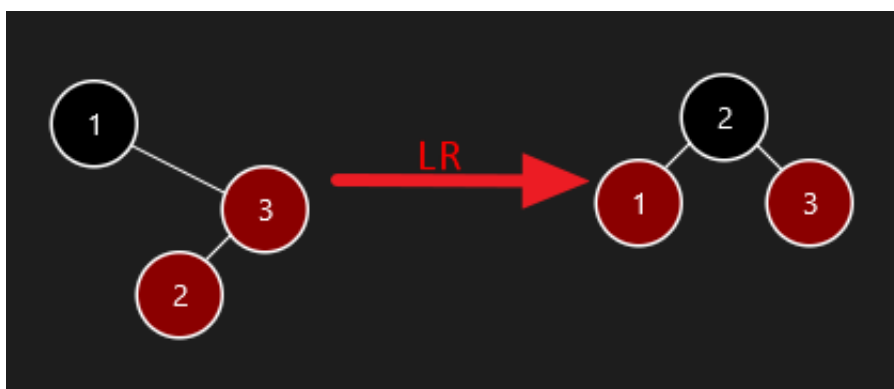
Obrázek 20: Rotace LL

### Dvojitá rotace RL



Obrázek 21: Rotace RL

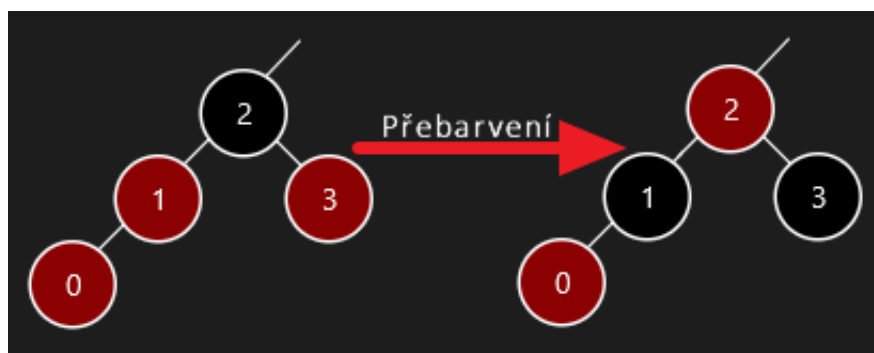
### Dvojitá rotace LR



Obrázek 22: Rotace LR

### Přebarvení + symetrické případy

Pokud má ten vyšší z dvojice červených po sobě jdoucích uzlů červeného sourozence, tak on i jeho sourozenec získá černou barvu a jejich rodič, pokud se nejedná o kořen, získá červenou barvu.



Obrázek 23: Přebarvení 1



Obrázek 24: Přebarvení 2

### 2.4.2 Vyhledávání

*Vyhledávání* u ČČ stromu se nijak neliší od toho v BVS.

### 2.4.3 Vkládání

*Vkládání* je založeno na *vkládání* u BVS. Navíc je zde kvůli vyvážení stromu přidán krok kontroly barev. Pokud se ve stromu po vložení objeví dva po sobě jdoucí uzly s červenou barvou, provedeme příslušné transformace, aby strom splňoval vlastnosti ČČ stromu.

Každý nový uzel má při vkládání automaticky červenou barvu.

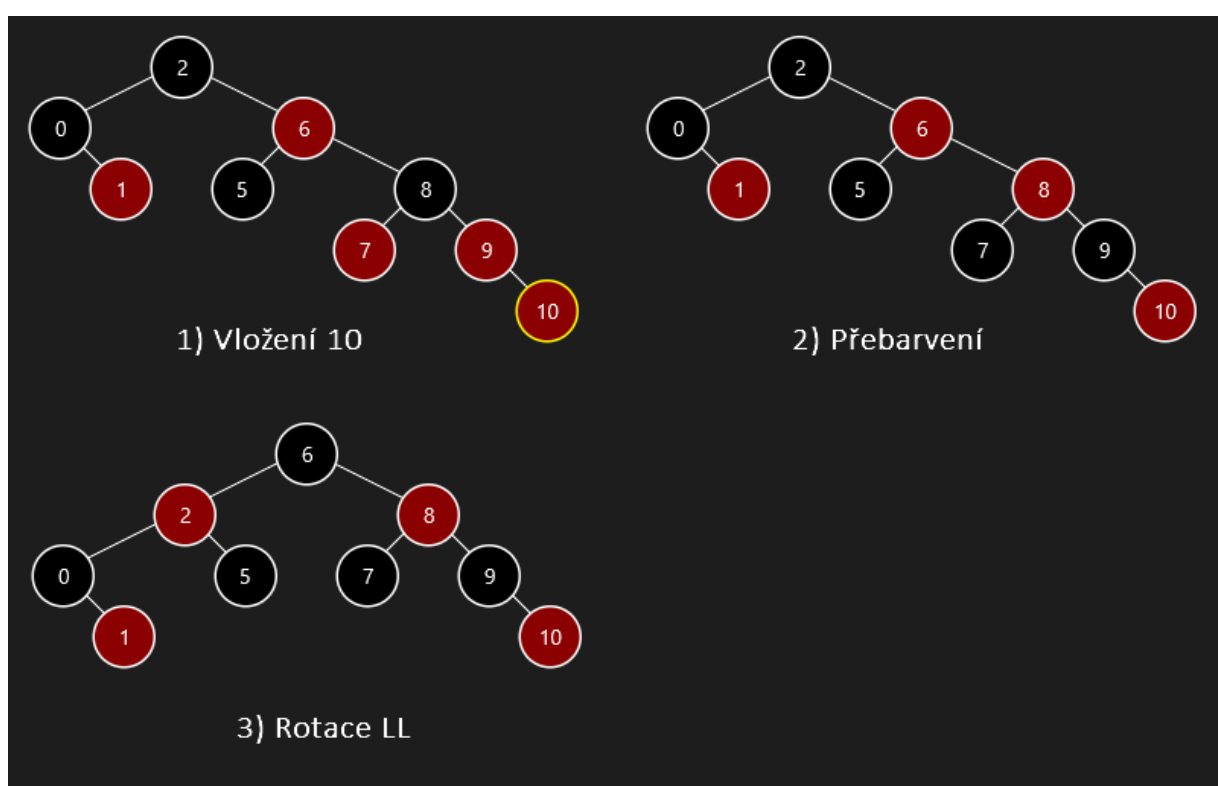
#### Přesný postup vkládání:

1. **krok** – počáteční  
Stejně jako u BVS.
2. **krok** – vyhledávání, vkládání  
Vložíme nový uzel  $u$  na příslušné místo a  $u$  nastavíme na předchůdce tohoto uzlu, je-li nový uzel kořen, nastavíme mu černou barvu a vkládání úspěšně končí.

### 3. krok – kontrola obarvení stromu

Pokud má  $u$  černou barvu, vkládání úspěšně končí. Jinak provedeme příslušnou transformaci:

- Rotaci  
Po provedení rotace se strom stává vyváženým a vkládání úspěšně končí.
- Přebarvení  
Po přebarvení nastavíme  $u$  na předchůdce  $u$ .  
Pokud je nyní  $u$  černý, tak se jedná o kořen a vkládání úspěšně končí.  
Jinak nastavíme  $u$  na rodiče  $u$  a znovu provedeme krok 3.



Obrázek 25: Postup vkládání hodnoty 10

#### 2.4.4 Odebírání

*Odebírání* u ČČ stromu je o trochu složitější než u předchozích stromů. Je to způsobeno tím, že musíme zaručit dodržování počtu a umístění černých uzlů, aby po odstranění byla dále splněna vlastnost: *cesta od kořene ke všem listům obsahuje vždy stejný počet černých uzlů*.

U *odebírání* tedy musíme sledovat barvu odebíraného uzlu i těch, které nahrazují odebraný uzel. Pokud odebraný (přesunutý) černý uzel nahradí červený, získá

černou barvu a strom zůstává vyvážený. Jinak na jeho původní místo vložíme do stromu *dvojitě obarvený černý uzel s NULL hodnotou*. Tento uzel i samotné *dvojitě obarvení* nám značí, že je daný strom nevyvážený a pomocí transformací strom upravíme, aby jsme *NULL uzel* i *dvojitě označení* odstranili a tím strom vyvážili.

### Přesný postup odebírání:

1. **krok** – počáteční  
Stejně jako u BVS.
2. **krok** – vyhledávání  
Stejně jako u BVS.
3. **krok** – odebírání

#### Odebírání $u$ má tyto možnosti:

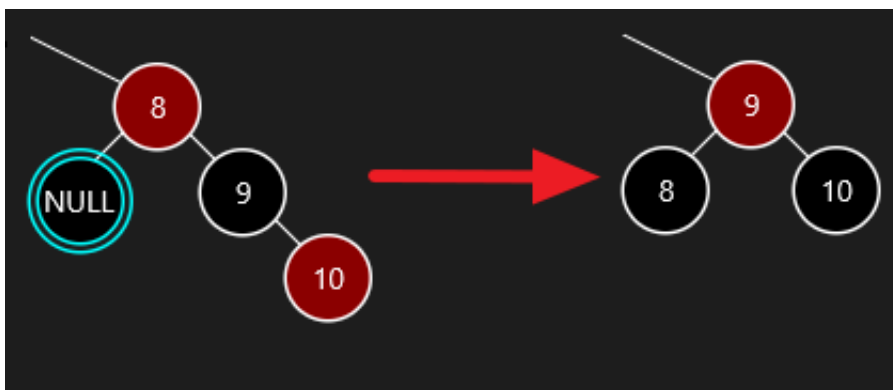
- Pokud  $u$  je list.  
List  $u$  odebereme.  
Pokud  $u$  neměl černou barvu nebo se jednalo o kořen, odebírání úspěšně končí.
- $u$  má jednoho potomka.  
Do  $u$  dosadíme hodnotu potomka. Pokud je potomek červený můžeme ho odstranit a odebírání úspěšně končí. Jinak  $u$  nastavíme na tohoto potomka a znovu zopakujeme krok 3.
- $u$  má dva potomky.
  - do  $u$  dosadíme hodnotu z *nejlevějšího* prvku z *pravého* podstromu. Pokud je tento prvek červený, odstraníme ho a odebírání úspěšně končí. Jinak  $u$  nastavíme na tento prvek a znovu zopakujeme krok 3.
  - do  $u$  dosadíme hodnotu z *nejpravějšího* prvku z *levého* podstromu. Pokud je tento prvek červený, odstraníme ho a odebírání úspěšně končí. Jinak  $u$  nastavíme na tento prvek a znovu zopakujeme krok 3.
- 4. **krok** – Odstranění NULL listu  
V tomto kroku na místo smazaného uzlu  $u$  vložíme dvojitě obarvený NULL list. Dále pokračujeme odstraňováním dvojitě obarveného černého uzlu, které je vysvětleno níže.

## Odstranění dvojité obarveného černého uzlu:

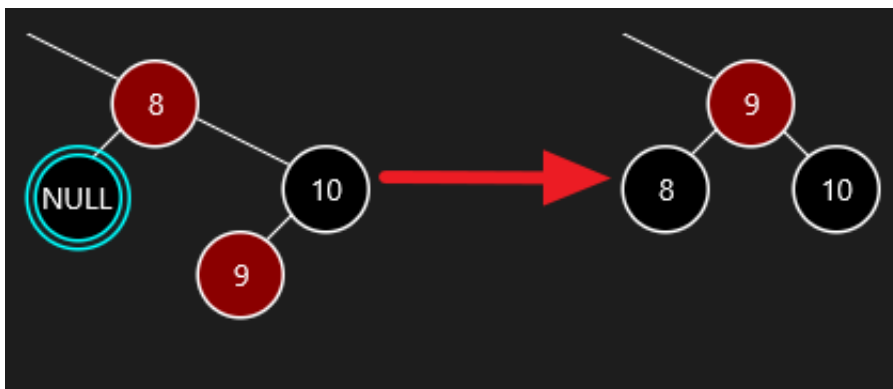
### ***Případ 1** – rotace + symetrické případy:*

Pokud má dvojité obarvený uzel černého sourozence, který má pravého nebo levého potomka červené barvy. Předchůdce dvojité obarveného uzlu může mít jakoukoliv barvu a tuto barvu pak získá uzel, který se po dokončení rotace dostane na jeho místo.

V obrázku 26 a 27 může před rotací uzel s hodnotou 8 mít černou barvu. Tuto barvu by po rotaci pak získal uzel s hodnotou 9.



Obrázek 26: Odstranění dvojité obarveného uzlu – jednoduchá rotace

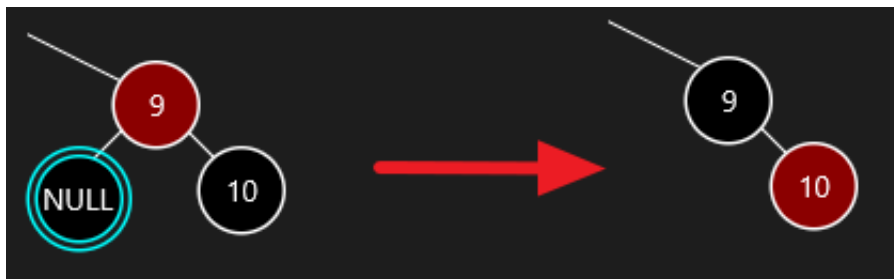


Obrázek 27: Odstranění dvojité obarveného uzlu – dvojitá rotace



**Případ 2 – přebarvení + symetrický případ:**

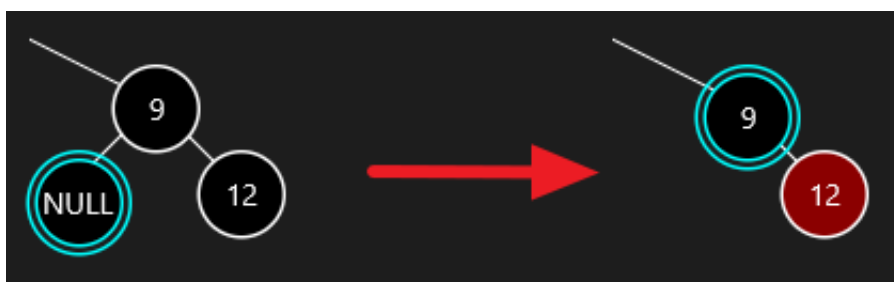
Pokud sourozenec dvojité obarveného uzlu je černý, ale nemá červené potomky a zároveň dvojité obarvený uzel má červeného předchůdce.



Obrázek 28: Odstranění dvojité obarveného uzlu – přebarvení

**Případ 3 – přebarvení a přesun označení + symetrický případ:**

Pokud sourozenec dvojité obarveného uzlu je černý, ale nemá červené potomky a zároveň dvojité obarvený uzel nemá červeného předchůdce.

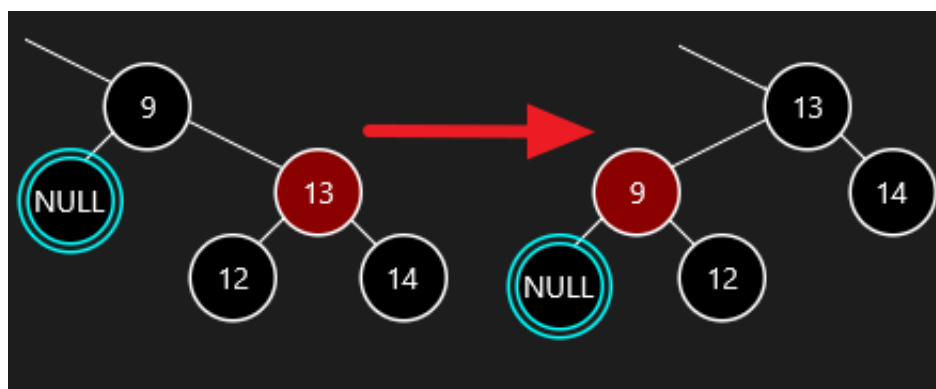


Obrázek 29: Odstranění dvojité obarveného uzlu – přebarvení a přesun označení

Pokud předchůdce dvojité obarveného uzlu není kořen, tak získá dvojité obarvení a dále pokračujeme s odstraněním tohoto označení.

**Případ 4 – rotace (bez odstranění dvojité obarveného uzlu) + symetrický případ:**

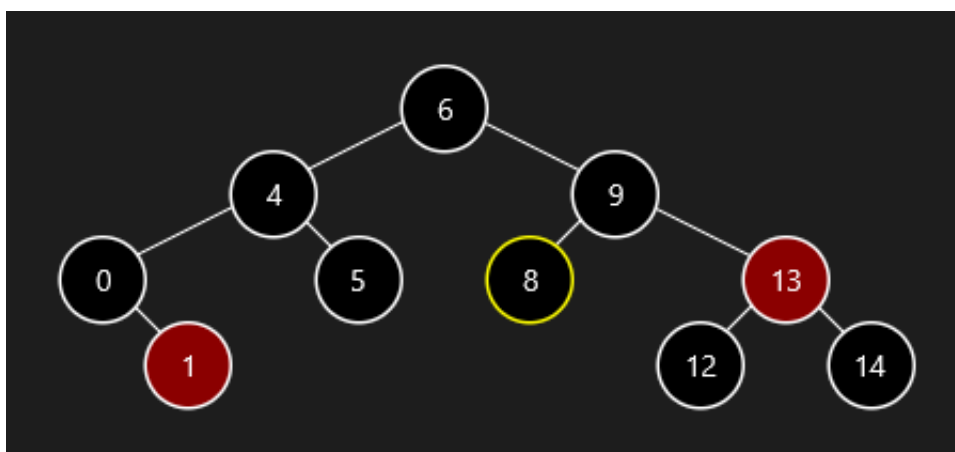
Pokud sourozenec dvojité obarveného uzlu je červený.



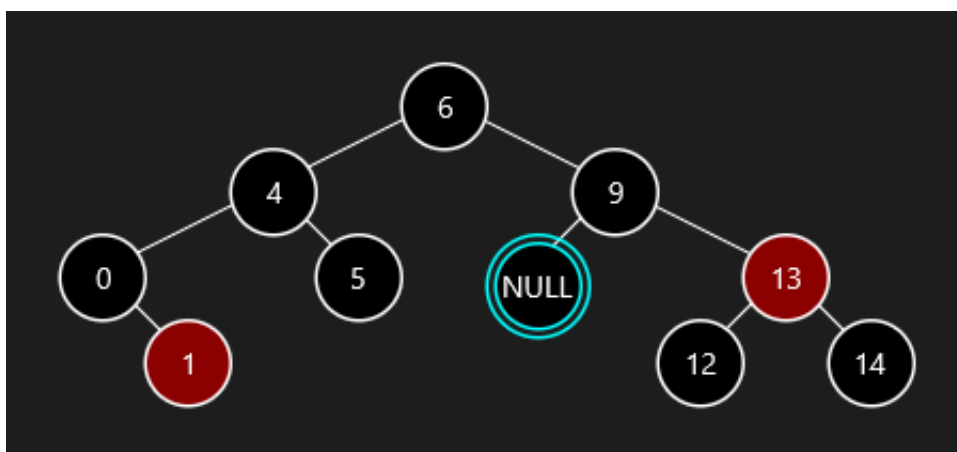
Obrázek 30: Rotace s dvojité obarveným uzlem – jednoduchá rotace

Po rotaci dále pokračujeme s odstraněním dvojité obarveného uzlu.

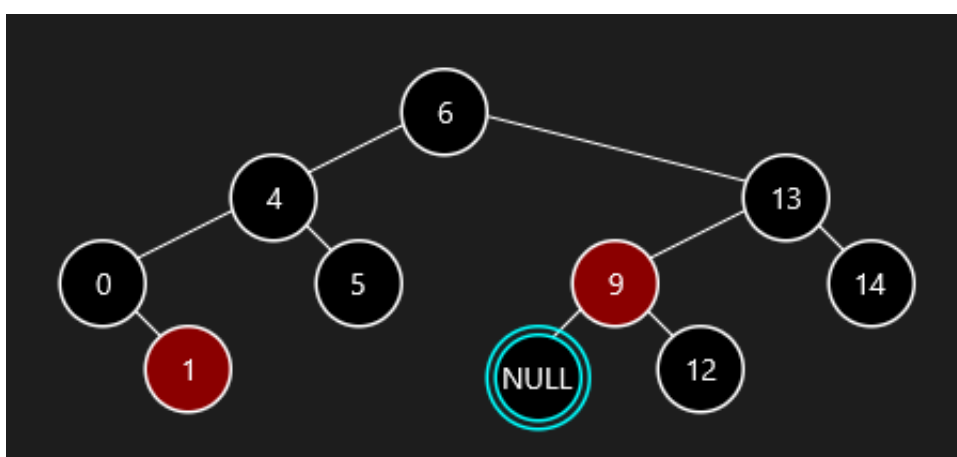
**Ukázka odebírání v ČČ stromu:**



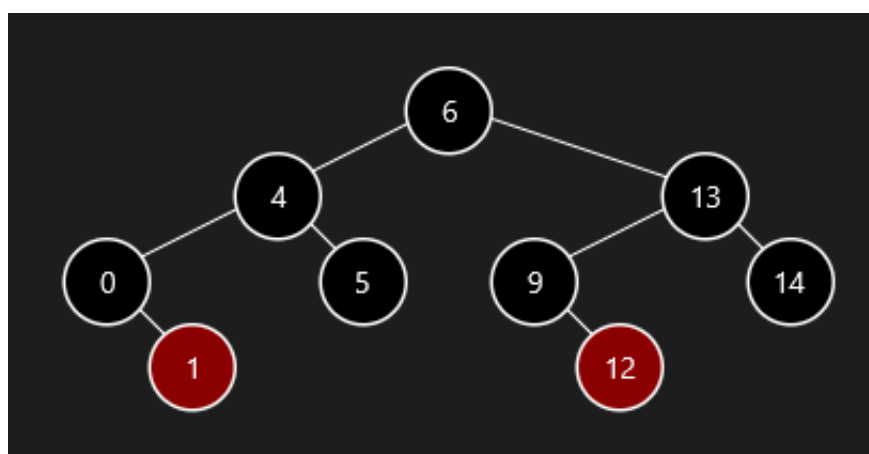
Obrázek 31: Ukázka odebírání – 1) odebrání 8



Obrázek 32: Ukázka odebírání – 2) vložen NULL list



Obrázek 33: Ukázka odebírání – 3) případ 4



Obrázek 34: Ukázka odebírání – 4) případ 2

## 3 Programátorská dokumentace

V této kapitole budou popsány technologie, které byly použity pro tvorbu aplikace. Dále zde bude popsána architektura a implementace programu.

### 3.1 Programovací jazyk a použité technologie

Program je napsán v jazyce Java, přesněji ve verzi 8. Na uživatelské prostředí je použita platforma JavaFX. Při výběru vhodného programového jazyka pro tuto aplikaci byla pro mě důležitá zkušenost s tímto jazykem a taky přenositelnost mezi operačními systémy, protože aplikaci jsem psal střídavě ve Windows 10 a macOS 10.13 High Sierra. Pro psaní zdrojového kódu jsem použil vývojové prostředí Eclipse Oxygen.

#### 3.1.1 Java

*Java* je objektově orientovaný programovací jazyk, který vznikl v roce 1995. Nyní je nově (od březnu roku 2018) ve verzi 10. Dle *TIOBY indexu* se Java v poslední době nachází stále na 1. místě nejpoužívanějších programovacích jazyků.<sup>[12]</sup> Program v Javě je možné spustit i bez instalace, stačí mít nainstalovanou správnou verzi JVM<sup>11</sup>.

#### 3.1.2 JavaFX

Pro tvorbu grafického uživatelského prostředí (zkratka GUI) aplikace jsem využil *JavaFX*. Je to platforma pro tvorbu GUI v programech Java, která je součástí knihovny Java od verze 8. Nahrazuje zastaralý *Swing*. *JavaFX* navíc umožňuje tvorbu animací a podporuje stylování pomocí CSS<sup>12</sup>.

#### 3.1.3 FXML

*FXML* je značkovací jazyk založen na jazyku XML<sup>13</sup>. Tento jazyk se používá k návrhu GUI v JavaFX.

### 3.2 Architektura programu

Program obsahuje tyto balíky:

- application
- trees
- graphic

---

<sup>11</sup>Java Virtual Machine.

<sup>12</sup>Cascading Style Sheets.

<sup>13</sup>Extensible Markup Language.

### 3.2.1 Balík *application*

Tento balík vytváří a má kontrolu nad GUI. Dále zprostředkovává komunikaci mezi logickou a grafickou částí programu.

#### Seznam souborů v balíku:

- Main
- WindowController
- styles – typu CSS
- Window – typu FXML

#### Třída *Main*

Třída dědí z třídy `Application`, která umožňuje vytvořit JavaFX aplikaci. Tato třída obsahuje funkci `main(String[] args)`, což je hlavní funkce programu, která se spustí hned při spuštění aplikace. Poté se spustí funkce `start(Stage primaryStage)`, která inicializuje GUI.

#### Třída *WindowController*

Tato třída obsluhuje všechny prvky okna vytvořené v FXML. Pomocí anotace `@FXML` získá tato třída reference na prvky GUI. Její hlavní činnost spočívá ve vykonávání metod, které jsou v souboru `Window.fxml` definovány pro prvky okna. Mezi hlavní metody navázané na prvky okna patří:

- `changeTree(ActionEvent event)` – metoda pro změnu aktuálního stromu, ta volá další funkce, které připraví prostředí pro zvolený strom.
- `searchNumber()`, `insertNumber()`, `deleteNumber()` – metody pro práci s aktuálním stromem, tyto metody zavolají příslušnou obsluhu aktuálního stromu `tree`, což je objekt typu `ITree` (3.2.2). Tento objekt provede operaci s daným stromem a vrátí výsledek typu `Result` (3.2.2). Ten je dále předán příslušné metodě objektu `graphicTree` typu `DrawingTree` (3.2.3).
- `repeatLastAnimation` – zaručuje obnovení předchozího stromu a zopakování poslední operace.
- `dialogNewTree` – obsluha pro tlačítko *Nový...*, které vyvolá dialog pro možnosti nového stromu.
- `checkEnableButtons` – funkce, která aktivuje a deaktivuje tlačítka podle logiky jejich možnosti použití (např. pokud je aktuální strom prázdný, je deaktivováno tlačítko *Vyhledat* a *Smazat*).

Její další činnost spočívá v komunikaci mezi logickou a grafickou částí programu. Což hlavně zajišťují již zmíněné metody: `searchNumber()`, `insertNumber()`, `deleteNumber()`. Navíc jsou zde funkce pro generování náhodného stromu: `newRandomTree()` a `generateRandomTreeList()` nebo pro ukládání předchozího stromu a jeho následné obnovení: `createHistory()`, `createHistoryRecursion(Object object)` (*pomocná metoda pro rekurzivní uložení rozložení hodnot v aktuálním stromu*) a `getHistoryTree()`.

### Soubor kaskádových stylů *styles*

Obsahuje popis způsobu zobrazování elementů v okně aplikace. Tyto styly jsou zapsány podobně jako CSS styly pro jazyk *HTML*, s tím rozdílem, že se před každou vlastnost známou ze stylování HTML stránek dává prefix `-fx-` např.:

```
1 .menu-text, Label {
2     -fx-font-size: 14.0pt;
3     -fx-font-family: "Segoe UI Light";
4     -fx-fill: white;
5 }
```

Zdrojový kód 7: `styles.css` - ukázka

Celá podrobná dokumentace CSS pro JavaFX aplikace je dostupná na adrese: <https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>.

### Soubor FXML *Window*

V tomto FXML souboru je napsána celá struktura GUI aplikace, která je navázána na příslušný controller `application.WindowController`. Jsou zde na jednotlivé elementy okna navázány příslušné metody pro obsluhu akcí s těmito elementy (*např. pokud uživatel klikne na tlačítko Vložit, je zavolána metoda `insertNumber()` z třídy `WindowController`*).

#### 3.2.2 Balík *trees*

Tento balík se stará o logiku stromů. Jsou zde implementovány všechny stromy a jejich příslušné operace, které byly výše popsány.

#### Seznam souborů v balíku:

- `ITree` – interface<sup>14</sup>
- `INode` – interface

---

<sup>14</sup>Rozhraní Java.

- `BinaryTree`
- `AVLTree`
- `RedBlackTree`
- `BinaryNode`
- `AVLNode`
- `RedBlackNode`
- `RecordOfAnimation`
- `Result`
- `AnimatedAction` – enum<sup>15</sup>
- `Color` – enum
- `Side` – enum

### Rozhraní *ITree*

Toto rozhraní zobecňuje práci se stromy, kdy každý typ stromu implementuje toto rozhraní a tím prakticky většina metod nerozlišuje s jakým typem stromu pracuje, ale pouze používá objekt typu `ITree` a volá metody, které mají pro všechny typy stromů stejný název, ale rozdílnou implementaci. Kvůli tomuto rozhraní je zdrojový kód programu velmi zjednodušen a ušetřen o spoustu řádků, které by musely řešit o jaký strom se jedná a teprve potom volat příslušné metody. Například již zmíněné metody třídy `WindowController`: `searchNumber()`, `insertNumber()`, `deleteNumber()` (3.2.1) pracují pouze s objektem typu `ITree`.

Hlavní definované metody rozhraní jsou:

- `search(int value)`, `insert(int value)`, `delete(int value)` – definice metod pro základní operace se stromy. Metody vrací objekt typu `Result`.
- `getRoot()` – metoda vrátí kořen daného stromu typu `INode`.

### Třídy *BinaryTree*, *AVLTree*, *RedBlackTree*

Jsou to třídy, které reprezentují typy stromů. Implementují rozhraní `ITree`, což zaručuje, že každá instance těchto tříd obsahuje všechny základní operace. Kvůli rozdílnému fungování těchto stromů musí každá z těchto tříd tyto operace implementovat po svém. `BinaryTree` prakticky implementuje pouze metody definované v rozhraní `ITree`, zatímco `AVLTree` a `RedBlackTree` obsahují metody navíc.

---

<sup>15</sup>Výčtový typ Javy.

Vybrané metody `AVLTree`:

- `balanceTree(Result result, AVLNode startNode)` – metoda, která se volá kvůli vyvážení AVL stromu. Tato metoda ověří faktory vyvážení příslušných uzlů, jestliže je strom nevyvážený volá potřebnou rotaci. Metoda vrací objekt typu `Result`.
- `llBalance(Result result, AVLNode nodeB),`  
`rrBalance(Result result, AVLNode nodeB),`  
`lrBalance(Result result, AVLNode nodeC),`  
`rlBalance(Result result, AVLNode nodeC)` – metody, které provádí rotace. Vrací objekt typu `Result`.

Vybrané metody `RedBlackTree`:

- `balanceTree(Result result, RedBlackNode startNode)` – metoda, která se volá kvůli vyvážení ČČ stromu. V této metodě se ověří vyváženost stromu, případně je zde provedeno přebarvení, pokud je potřeba jsou volány i metody rotace. Metoda vrací objekt typu `Result`.
- `llBalance(Result result, RedBlackNode nodeB),`  
`rrBalance(Result result, RedBlackNode nodeB),`  
`lrBalance(Result result, RedBlackNode nodeC),`  
`rlBalance(Result result, RedBlackNode nodeC)` – metody, které provádí rotace a s tím související přebarvení. Vrací objekt typu `Result`.
- `doubleBlack(Result result, RedBlackNode parent, Side side)` – tato funkce řeší odstranění dvojité obarvených uzlů, které se mohou vyskytnout při operaci odebírání. Funkce vrací objekt typu `Result`.

### Rozhraní *INode*

Rozhraní `INode` obsahuje definice všech důležitých `get` a `set` metod, které jsou potřeba k práci s jakýmkoliv uzlem. Je zde definována i metoda `equals(Object obj)`, která slouží k porovnávání uzlů. Toto rozhraní zobecňuje práci s uzly stromů a umožňuje měnit a číst jejich atributy, aniž by muselo být známo o jaký typ stromu, a tedy i uzlu, se jedná.

### Třídy *BinaryNode*, *AVLNode*, *RedBlackNode*

Reprezentují uzly stejnojmenných stromů. Implementují rozhraní `INode`. Třídy obsahují tyto atributy:

- `value` – číselná hodnota uzlu.
- `parent` – ukazatel na rodiče.
- `left` – ukazatel na levého potomka.
- `right` – ukazatel na pravého potomka.



- `graphicNode` – grafická reprezentace uzlu.
- `factor`(pouze u `AVLNode`) – číselná hodnota faktoru vyvážení.
- `color`(pouze u `RedBlackNode`) – barva uzlu. Reprezentována pomocí výčtového typu `Color`.

Třída `AVLNode` navíc implementuje zmíněnou rekurzivní funkci `computeFactor()` [6](#).

### **Třída *RecordOfAnimation***

Tato třída slouží k zaznamenávání animací, které budou následně zpracovány v grafické části aplikace.

Třída obsahuje tyto atributy:

- `action` – typ animace reprezentovaný výčtovým typem `AnimatedAction`.
- `node` – ukazatel na grafický uzel, kterého se daná animace týká. Tato proměnná je typu `IGraphicNode` ([3.2.3](#)).
- `object` – pomocná proměnná, která je používána na různé typy objektu podle druhu animace.

### **Třída *Result***

Objekty této třídy jsou používány jako návratové hodnoty všech logických operací se stromy. Tento objekt má v sobě uloženy důležité informace pro následné zpracování všech grafických operací včetně animací. Ve třídě `WindowController` metody: `searchNumber()`, `insertNumber()`, `deleteNumber()` ([3.2.1](#)) nejprve zavolají příslušnou metodu u aktuálního logického stromu typu `ITree`. Ta vytvoří objekt typu `Result`, do kterého postupně uloží všechny animace potřebné ke grafickému zobrazení volané operace a tento objekt na závěr vrátí. Tyto metody vrácený výsledek dále předávají objektu typu `DrawingTree` ([3.2.3](#)), který strom vykreslí a vykoná příslušné animace.

Třída obsahuje tyto atributy:

- `node` – výsledný uzel dané operace reprezentován pomocí `INode`.
- `side` – pozice výsledného uzlu (*při vkládání je to například strana na kterou má být uzel vložen*), která je výčtového typu `Side`.
- `way` – seznam všech navštívených uzlů při provádění dané operace. Je typu `ArrayList<IGraphicNode>`. Využívá se při animování vyhledávání uzlu.
- `recordOfAnimations` – seznam záznamů všech animací. Je typu `ArrayList<RecordOfAnimation>`.

Vybrané metody:

- `addNodeToWay(IGraphicNode node)` – přidává do kolekce `way` navštívený uzel.
- `addAnimation(AnimatedAction action, IGraphicNode node, Object object)` – metoda, která přidá do `recordOfAnimations` nový záznam animace.

### Výčtový typ *AnimatedAction*

Obsahuje například hodnoty: `SEARCH`, `INSERT`, `DELETE`, `MOVEVALUE`, `MOVENODE`, `RR`, `LL`, `RL`, `LR`... a další.

### Výčtový typ *Color*

Obsahuje hodnoty: `RED`, `BLACK`;

### Výčtový typ *Side*

Obsahuje hodnoty: `LEFT`, `RIGHT`, `NONE`;

## 3.2.3 Balík *graphic*

Tento balík se stará o grafické zobrazování stromů, animování všech operací a zobrazování zjednodušeného popisu.

### Seznam souborů v balíku:

- `IGraphicNode` – interface
- `AVLGraphicNode`
- `BinaryGraphicNode`
- `RedBlackGraphicNode`
- `DrawingTree`

### Rozhraní *IGraphicNode*

Toto rozhraní definuje všechny důležité `get`, `set` a další metody, které jsou potřeba pro práci s jakýmkoliv objektem reprezentující grafický uzel zmíněných stromů. Je ním zobecněna práce třídy `DrawingTree`, která může vykreslovat a provádět animace aniž by znala typ stromu.

Objekty typu `IGraphicNode` reprezentují obecný grafický uzel všech stromů a jsou prakticky používány ve všech částech programu. Výjimku tvoří funkce, které jsou specifické pro určitý typ stromu, ty si objekty typu `IGraphicNode` přetypují do `BinaryGraphicNode`, `AVLGraphicNode` nebo `RedBlackGraphicNode` (Například u AVL stromu je potřeba změnit atribut `factor`, který obsahuje a má k němu přístup pouze `AVLGraphicNode`).

Hlavní definované metody jsou:

- `createStackPaneNode()` – vytvoří grafickou reprezentaci listu (*vytvoří kruh a doprostřed vloží hodnotu listu, to celé pak vloží do StackPane*).
- `highlightFindNode()` – metoda pro zvýraznění nalezeného grafického uzlu.
- `countChildren()` – metoda, které vypočítá a uloží počet všech pravých a levých potomků daného uzlu. Metoda vrací součet všech potomků. Používá se pro rozpočítání vzdáleností mezi uzly, aby se ve vykresleném stromu žádné nepřekrývaly. Implementace metody je zde [8](#).

### Třída *BinaryGraphicNode*

Třída implementuje rozhraní `IGraphicNode`. Reprezentuje grafický uzel stejnojmenného stromu. Pouze implementuje všechny metody z rozhraní. Obsahuje tyto atributy:

- `radiusSize` – velikost vykreslených uzlů. Hodnota je typu `final int` inicializovaná na 20.
- `parent` – ukazatel na rodiče uzlu. Typu `IGraphnicNode`.
- `left` – ukazatel na levého potomka. Typu `IGraphnicNode`.
- `right` – ukazatel na pravého potomka. Typu `IGraphnicNode`.
- `side` – pozice vůči rodiči. Výčtového typu `Side`.
- `leftChildrenCount` – počet všech levých potomků.
- `rightChildrenCount` – počet všech pravých potomků.
- `value` – ukazatel na hodnotu uzlu. Typu `Text`.
- `circle` – grafický prvek kruh o poloměru `radiusSize`. Typu `Circle`.
- `stacPaneNode` – `StackPane`, který představuje grafický uzel. V něm je vložen objekt `value` a `circle`.
- `x` – souřadnice `stacPaneNode` x. Typu `DoubleProperty`.
- `y` – souřadnice `stacPaneNode` y. Typu `DoubleProperty`.
- `branch` – představuje větev<sup>16</sup> mezi tímto uzlem a jeho rodičem. Typu `Line`.

---

<sup>16</sup>Spoj mezi uzlem a jeho rodičem.

```

1  @Override
2  public int countChildren() {
3      if (left != null) {
4          leftChildrenCount = 1 + left.countChildren();
5      } else {
6          leftChildrenCount = 0;
7      }
8
9      if (right != null) {
10         rightChildrenCount = 1 + right.countChildren();
11     } else {
12         rightChildrenCount = 0;
13     }
14
15     return leftChildrenCount + rightChildrenCount;
16 }

```

Zdrojový kód 8: countChildren()

### **Třídy *AVLGraphicNode* a *RedBlackGraphicNode***

Tyto třídy dědí ze třídy `BinaryGraphicNode` => zároveň pro ně automaticky platí, že implementují rozhraní `IGraphicNode`. Tyto třídy reprezentují grafické uzly svých stejnojmenných stromů, ale prakticky se od `BinaryGraphicNode` příliš neliší.

`AVLGraphicNode` pouze přidává atribut `factor` a k němu příslušnou `get` a `set` metodu. Faktor vyvážení je reprezentován objektem typu `Text`. Tento atribut následně vloží do zděděného atributu `stackNode`.

`RedBlackGraphicNode` používá červenou a černou barvu výplně zděděného atributu `circle`, kvůli tomu je přidán atribut `color` výčtového typu `Color` a jeho příslušnou `get` a `set` metodu. Dále přidává metodu pro označení dvojité obarveného listu.

### **Třída *DrawingTree***

Nejrozsáhlejší třída aplikace, která ovládá grafické zobrazování všech stromů, animací a zobrazování popisku aktuálně prováděných operací. `WindowController` při vytvoření nového stromu vytvoří objekt `graphicTree`, který je instancí `DrawingTree`. Objektu `graphicTree` jsou pak pomocí metod předávány výsledky (typu `Result`) zpracovaných operací v logické reprezentaci stromu. Tyto výsledky jsou následně graficky zpracovány a po provedení poslední animace je program připraven na přijímání dalších požadavků od uživatele.

Parametry konstruktoru:

1. `paneTree` – ukazatel na prvek okna typu `Pane`, do kterého se stromy budou kreslit.

2. `animationSpeed` – ukazatel na property<sup>17</sup> prvku okna typu `Slider`. Parametr je typu `DoubleProperty`. Tato hodnota reprezentuje aktuální zvolenou hodnotu uživatelem pomocí *slideru* pro nastavení rychlosti animace.
3. `stageWidthProperty` – ukazatel na property šířku okna aplikace. Je typu `ReadOnlyDoubleProperty`.
4. `stageHeightProperty` – ukazatel na property výšku okna aplikace. Je typu `ReadOnlyDoubleProperty`.
5. `WindowController` – ukazatel na samotný objekt `WindowController`, který tento konstruktor volá. To proto, aby tato třída měla přístup k metodám pro deaktivování prvků okna při průběhu animace.

```

1 g = new DrawingTree(paneTree, sliderSpeed.valueProperty(), window.
    widthProperty(), window.heightProperty(), this);
2 /**
3  * window - je okno aplikace
4  * this - WindowController ve kterém je tento konstruktor volán
5  */

```

#### Zdrojový kód 9: Ukázka použití konstruktoru třídy `DrawingTree`

Třída má spoustu atributů, které jsou hlavně využívány funkcemi týkajícími se animací. Animace po skončení automaticky volají funkce, které nemají přístup k lokálně uloženým datům a proto musí pracovat s atributy třídy, ke kterým je přístup ze všech funkcí dané třídy. Některé atributy třídy:

- `value` – hodnota vložená uživatelem načtená z `TextArea`.
- `newIGraphicNode` – ukazatel na nově vkládaný uzel.
- `graphicNodeSize` – rozměr grafického uzlu. Při vkládání kořene je do tohoto atributu vložen jeho rozměr.
- `xAnimatedNode`, `yAnimatedNode` – `DoubleProperty` sloužící k ukládání, které jsou dále používány v animacích.
- `DOWNMARGIN` – představuje velikost zobrazené mezery mezi potomkem a rodičem. Je typu `final int` a má hodnotu 40.
- `listGraphicNodes` – seznam všech aktuálně zobrazených uzlů. Typu `ArrayList<IGraphicNode>`.

---

<sup>17</sup>Vlastnost objektu v JavaFX, které lze *naslouchat* a reagovat na změny. Tyto vlastnosti jsou obvykle typovány na typ objektu, který uchovávají např. `DoubleProperty`. Jdou propojit (*nabíndovat*) s jinými property stejného typu => pokud se změní hodnota v property, tak se změní i hodnota ve všech, které jsou s ní propojeny.

- `recordOfAnimations` – seznam aktuálně prováděných animací. Typu `ArrayList<RecordOfAnimation>`.
- `animationIndex` – index poslední provedené animace.
- `wayList` – aktuální seznam všech navštívených uzlů, pro animování vyhledávání. Typu `ArrayList<IGraphicNode>`.
- `wayIndex` – index posledního zvýrazněného uzlu.
- `balanceIndex` – index aktuálně kontrolovaného uzlu, používá se ve funkci `balanceTreeNext()`.
- Dále obsahuje všechny uložené parametry z konstruktoru jako je: `paneTree`, `animationSpeed`, `stageWidthProperty`, `stageHeightProperty`, `windowController`.

Vybrané metody:

- `insertRoot(INode root)` – metoda animuje vložení kořene. Kvůli skutečnosti, že uživatel může měnit velikost okna aplikace, musí mít kořen souřadnice, které se přizpůsobují aktuální šířce okna. K tomu slouží již zmíněná `stageWidthProperty`. S tímto atributem propojím `x` (`DoubleProperty`) kořene, navíc ji vydělím dvěma, abych pozicoval uzel přímo doprostřed okna. Tím zaručím, že kořen bude vždy uprostřed okna.
- `insertNode(Result result, int value)` – metoda je volána z `WindowController` při kliknutí uživatele na tlačítko *Vložit*. Nejprve se vkládanému uzlu uloženého v `result` nastaví počáteční souřadnice a je vložen do okna, zároveň je uložen do `newIGraphicNode`. Poté je zavolána metoda `startAnimation(result.getRecordOfAnimations())`.
- `deleteNode(Result result, int value)` – pokud je mazaný uzel nalezen volá `startAnimation(result.getRecordOfAnimations())`. Při nenalezení pouze zobrazí animaci hledání.
- `searchNode(Result result, int value)` – připraví prostředí na hledání a zavolá `startAnimation(result.getRecordOfAnimations())`.
- `startAnimation(ArrayList<RecordOfAnimation> recordOfAnimations)` – tato funkce pouze připraví prostředí pro spuštění animací. Tato příprava spočívá v nastavení všech potřebných atributů na výchozí hodnoty, určení rychlosti animace, uložení aktuálního seznamu animací, atd.). Dále volá funkci `nextAnimation()`.
- `nextAnimation()` – funkce má na starost postupné vykonávání animací ze seznamu `recordOfAnimations`. Obsahuje `switch`, který podle typu animace zavolá příslušnou funkci. Ukázka této funkce naleznete zde [11](#).

- `insertNodeAnimation()` – zde je prováděna animace vkládání nového uzlu. Souřadnice nově umístěného uzlu určím pomocí tohoto kódu 10. Po dokončení animace je zavolána funkce `insertNodeAnimationFinished()`, která tyto souřadnice uloží do vloženého uzlu a zařídí pokračování dalších animací, tak že navýší `animationIndex` o 1 a zavolá funkci `nextAnimation()`.
- `createBranch(IGraphicNode node)` – funkce vytvoří pro daný uzel *větev*, nebo-li hranu mezi tímto uzlem a jeho rodičem.
- `checkBranches()` – funkce ověří, zda všechny vykreslené uzly, mají zobrazené příslušné *větve*. Pokud se tam nějaká větev nenachází, tak jí zobrazí. Neobsahuje-li uzel větev, která by se měla zobrazit, tak je nejprve vytvořena a pak i zobrazena.
- `balanceTree()` a `balanceTreeNext()` – funkce, která zajišťuje korektní zobrazení stromů. `balanceTree()` je volána při dokončení všech animací. Uvnitř funkce se pomocí rekurzivní funkce `countChildren()` aktualizuje počet potomků u všech uzlů. Pak je zavolána funkce `balanceTree()` a ta pomocí rekurze postupně prochází `listGraphicNodes`. Pro každý uzel vypočítá správné souřadnice a porovná je s aktuálním umístěním uzlu. Pokud se souřadnice liší, tak je potřeba uzel posunout na správné místo. Tímto postupem je zaručeno, že se žádný uzel nebude překrývat. Ukázka funkce je zde 12.
- Dále jsou zde implementovány metody na příslušné animace:  
`deleteNodeAnimation()`, `moveNodeAnimation()`, `moveValueAnimation()`,  
`updateFactor()`, `rrAnimation()`, `rlAnimation()`, `llAnimation()`,  
`lrAnimation()`.

```

1 DoubleProperty x = new SimpleDoubleProperty();
2 DoubleProperty y = new SimpleDoubleProperty();
3
4 if (result.getSide() == Side.LEFT) {
5     x.bind(new IGraphicNode.getParent().getX().subtract(graphicNodeSize
6         ));
7 } else {
8     x.bind(new IGraphicNode.getParent().getX().add(graphicNodeSize));
9 }
10 y.bind(new IGraphicNode.getParent().getY().add(DOWNMARGIN));
11 new IGraphicNode.setX(x);
12 new IGraphicNode.setY(y);

```

Zdrojový kód 10: Ukázka určení souřadnic nově vloženého uzlu.

```

1 private void nextAnimation() {
2     //pokud už proběhly všechny animace
3     if (animationIndex >= recordOfAnimations.size()) {
4         balanceTree();
5         return;
6     }
7
8     switch (recordOfAnimations.get(animationIndex).getAction().) {
9     case SEARCH:
10         wayIndex = 0;
11         nextSearchNode();
12         break;
13
14     case INSERT:
15         insertNodeAnimation();
16         break;
17
18     case DELETE:
19         deleteNodeAnimation();
20         break;
21
22     case MOVENODE:
23         moveNodeAnimation();
24         break;
25     ...

```

Zdrojový kód 11: Zkrácená ukázka nextAnimation()

```

1 xAnimatedNode = new SimpleDoubleProperty();
2 IGraphicNode iGraphicNode = listGraphicNodes.get(balanceIndex);
3
4 if (iGraphicNode.getSide() == Side.LEFT) {
5     xAnimatedNode.bind(iGraphicNode.getParent().getX().subtract(
6         graphicNodeSize).subtract(graphicNodeSize * iGraphicNode.
7         getRightChildrenCount()));
8
9 } else {
10     xAnimatedNode.bind(iGraphicNode.getParent().getX().add(
11         graphicNodeSize).add(graphicNodeSize * iGraphicNode.
12         getLeftChildrenCount()));
13 }
14
15 //pokud uzel nemá správné souřadnice provedu posunutí
16 if (iGraphicNode.getX().get() != xAnimatedNode.get()) {
17     ...
18 }

```

Zdrojový kód 12: Ukázka výpočtu správného umístění uzlu.



## 4 Uživatelská příručka

Uživatelská příručka obsahuje návod, který popisuje jak aplikaci spustit a její funkčnost. Jsou zde popsány i všechny komponenty okna a práce s nimi.

### 4.1 Minimální požadavky aplikace

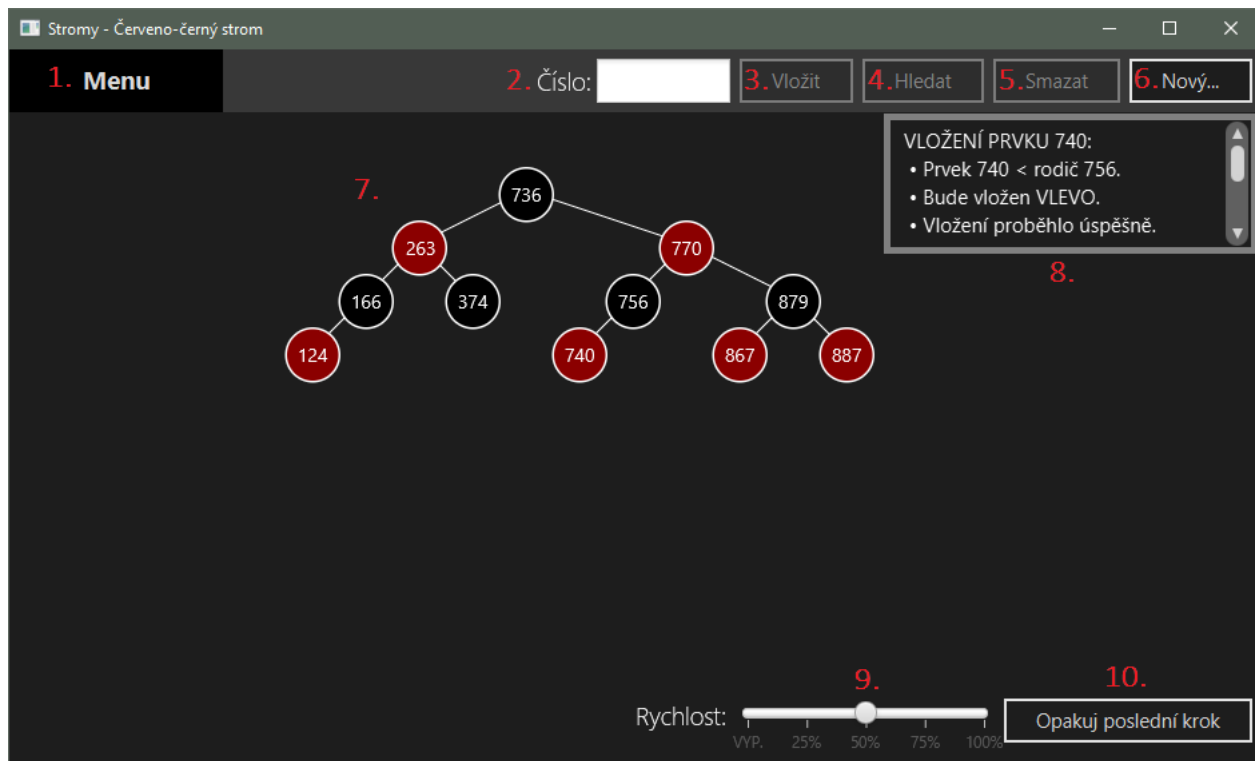
- Windows 10 nebo macOS 10.12 Sierra.
- JVM verze 8.
- Rozlišení obrazovky aspoň 1366×768.

### 4.2 Spuštění

Aplikaci není třeba instalovat. Před samotným spuštěním je pouze potřeba mít nainstalovaný JVM verze 8, který dostupný z adresy: <https://java.com/en/download/>. Samotnou aplikaci STROMY.JAR() stačí zkopírovat z adresáře bin/ a spustit.

### 4.3 Okno aplikace

Po spuštění se zobrazí hlavní okno programu viz. obrázek 35. V titulku okna je název aktuálně zvoleného stromu.



Obrázek 35: Hlavní okno programu.

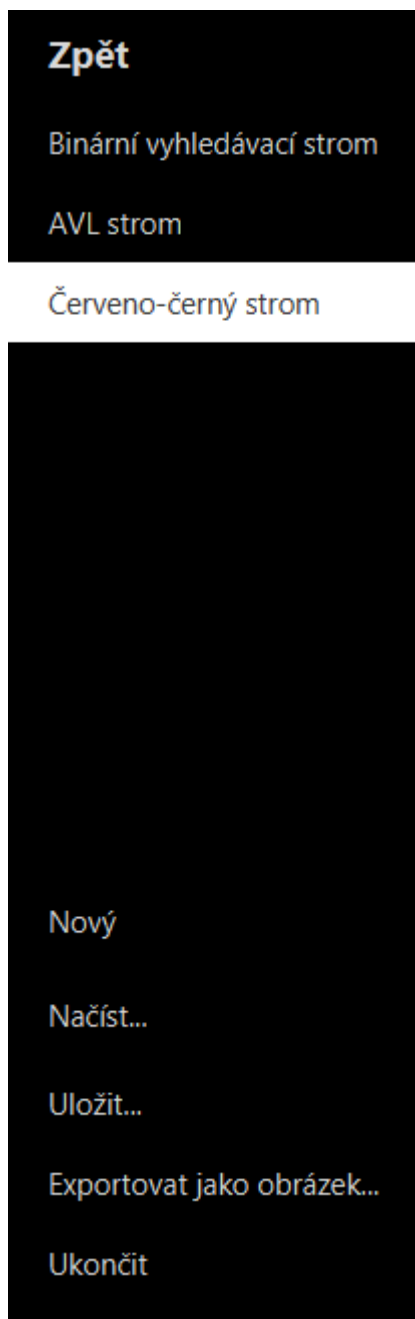
### 4.3.1 Popis okna aplikace

1. Menu – slouží k výběru stromu viz. obrázek 36. Aktuální strom je v menu zvýrazněn bílou barvou. Při vybrání stromu se objeví dialogové okno 37. Menu skryjeme po kliknutí na položku *Zpět*.

Další funkce menu:

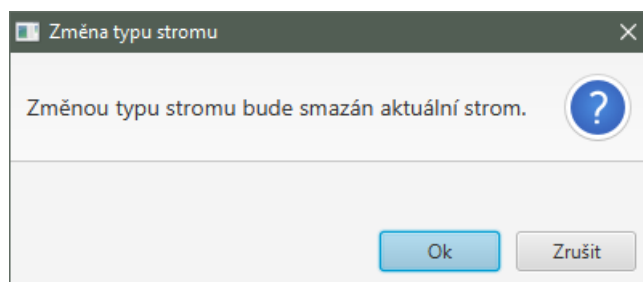
- *Nový* – Smaže aktuální strom, může zobrazit dialogové okno 38.
  - *Načíst...* – Zobrazí standardní okno pro výběr umístění uloženého souboru *název.tree*. Načítání může zobrazit dialog 39 nebo chybovou hlášku 40.
  - *Uložit...* – Zobrazí standardní okno pro výběr umístění a názvu ukládaného souboru. Na dané místo uloží aktuální strom jako *název.tree*. Ukládání může zobrazit chybovou hlášku 41.
  - *Exportovat jako obrázek...* – Zobrazí standardní okno pro výběr umístění a názvu obrázku. Na dané místo uloží obrázek aktuálního stromu jako *název.png*.
  - *Ukončit* – Ukončí aplikaci.
2. Textové pole – pro uživatelské zadávání číselných hodnot. Rozmezí čísla je 0 až 10 000. Při zadání čísla se automaticky tlačítka, které je možné v daném případě použít stávají aktivními.
  3. Tlačítko *Vložit* – slouží ke vkládání uživatelem zadané hodnoty do aktuálního zobrazeného stromu. Je aktivní vždy, pokud uživatel zadá nějakou hodnotu.
  4. Tlačítko *Hledat* – slouží k hledání uživatelem zadané hodnoty v aktuálním zobrazeném stromu. Je aktivní, pokud strom má alespoň kořen a textové pole obsahuje nějakou uživatelem zadanou hodnotu.
  5. Tlačítko *Smazat* – slouží k odebrání uzlu s uživatelem zadanou hodnotou v aktuálně zobrazeném stromu. Je aktivní, pokud strom má alespoň kořen a textové pole obsahuje nějakou uživatelem zadanou hodnotu
  6. Tlačítko *Nový...* – po kliknutí zobrazí dialog 42. V tomto dialogu je možno aktuální strom vymazat nebo vytvořit náhodný strom. Při výběru *Nový náhodný...* se zobrazí dialog 43, kde zadáme počet hodnot ve stromě a po potvrzení je strom vytvořen.
  7. Kreslicí plocha – zde je vykreslován strom, jsou zde i prováděny animace.
  8. Popis operací – stručný popis aktuálně prováděných operací, který si lze pročíst i po provedení animace. Tento popis lze přesunout, pomocí chycnutí šedého okraje, kamkoliv po kreslicí ploše programu.

9. Rychlost animací – před prováděním animace, lze vybrat její rychlost, případně i animace vypnout.
10. Tlačítko *Opakuj poslední krok* – po kliknutí na tlačítko se zopakuje poslední provedená operace. Toto tlačítko je aktivní, pokud je uložena nějaká předchozí operace.

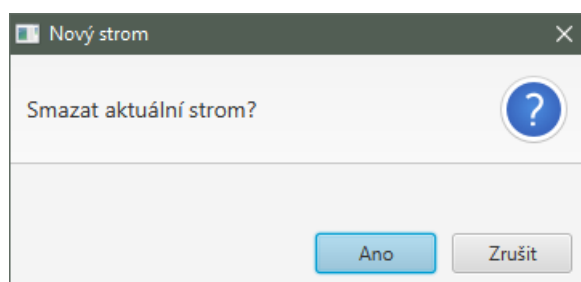


Obrázek 36: Menu programu.

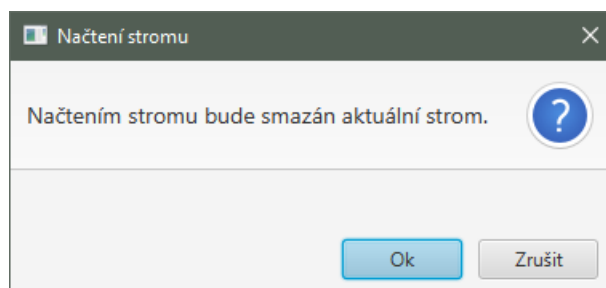
Dialogové okna aplikace:



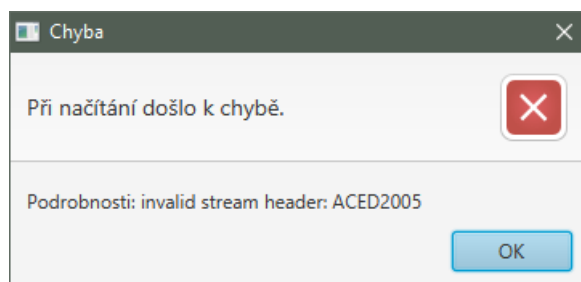
Obrázek 37: Dialogové okno při změně stromu.



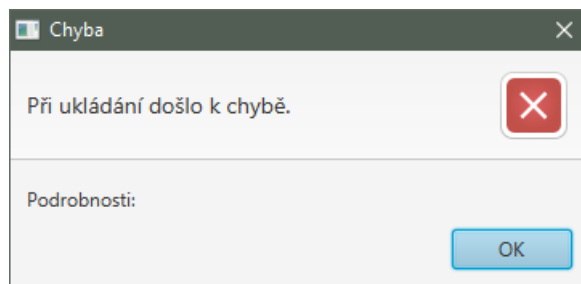
Obrázek 38: Dialogové okno kliknutí na *Nový* v menu.



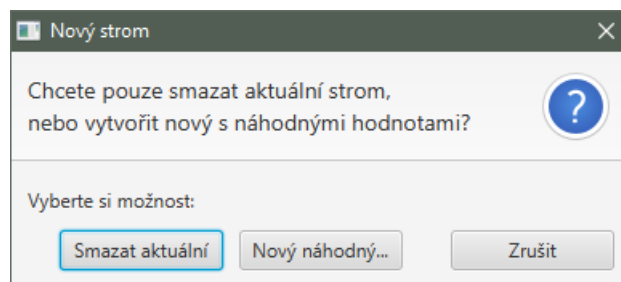
Obrázek 39: Dialogové okno při načítání stromu.



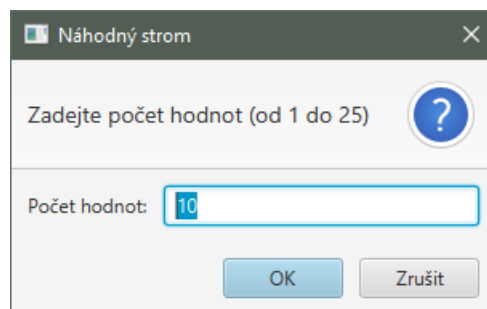
Obrázek 40: Chyba při načítání uloženého stromu.



Obrázek 41: Chyba při ukládání stromu.



Obrázek 42: Dialogové okno navázané na tlačítko *Nový...*



Obrázek 43: Dialogové okno pro náhodný strom.

#### 4.3.2 Klávesové zkratky

Při zadávání hodnoty do textového pole, je možné volat operace pomocí klávesových zkratk:

- *ENTER* – zavolá operaci *Vložit*.
- *F* – zavolá operaci *Hledat*.
- *DELETE* – zavolá operaci *Smazat*.

## Závěr

Výsledkem bakalářské práce je aplikace STROMY, která může sloužit k potřebě výuky *Binárních vyhledávacích*, *AVL* a *červeno-černých stromů*. Aplikace názorně animuje operace *Vyhledávání*, *Vkládání* a *Odebírání*. Tyto operace jsou zároveň stručně popsány, navíc je možnost je znovu zopakovat, což umožňuje lepší pochopení dané operace.

Aplikace je naprogramována s důrazem na jednoduchost a obecnost. Toto zobecnění hlavních částí programu umožňuje v budoucnu doprogramovat další typy stromů, aniž by se musela předělávat celá aplikace.

## Conclusions

Thesis conclusions in “English”.

## A Obsah přiloženého CD/DVD

Na samotném konci textu práce je uveden stručný popis obsahu přiloženého CD/DVD, tj. jeho závazné adresářové struktury, důležitých souborů apod.

### **bin/**

Program STROMY, spustitelné přímo z CD/DVD.

### **doc/**

Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

### **src/**

Kompletní zdrojové kódy programu STROMY.

### **readme.txt**

Instrukce pro instalaci a spuštění programu STROMY, včetně všech požadavků pro jeho bezproblémový provoz.

Navíc CD/DVD obsahuje:

### **literature/**

Vybrané položky bibliografie, příp. jiná užitečná literatura vztahující se k práci.

U veškerých cizích převzatých materiálů obsažených na CD/DVD jejich zahrnutí dovolují podmínky pro jejich šíření nebo přiložený souhlas držitele copyrightu. Pro všechny použité (a citované) materiály, u kterých toto není splněno a nejsou tak obsaženy na CD/DVD, je uveden jejich zdroj (např. webová adresa) v bibliografii nebo textu práce nebo v souboru `readme.txt`.



## Literatura

- [1] BĚLOHLÁVEK, Radim. Algoritmická matematika 2 – část 1 [online]. 2012-05-15. [cit. 2018-07-07]. Dostupné z: <http://belohlavek.inf.upol.cz/vyuka/algoritmicka-matematika-2-1.pdf>
- [2] BĚLOHLÁVEK, Radim; VYCHODIL, Vilém. Diskrétní matematika pro informatiky II [online]. 2010-10-16. [cit. 2018-07-07]. Dostupné z: <http://belohlavek.inf.upol.cz/vyuka/dm2.pdf>
- [3] VEČERKA, Arnošt. Studijní materiály ALM-2
- [4] DVORSKÝ, Jiří. Algoritmy I.[online]. 2007-02-27. [cit. 2018-07-07]. Dostupné z: <http://www.cs.vsb.cz/dvorsky/Download/SkriptaAlgoritmy/Algoritmy.pdf>
- [5] FINLAYSON, Ian. Binary Search Trees [online]. [cit. 2018-07-07]. Dostupné z: <http://cs.umw.edu/~finlayson/class/fall12/cpsc230/notes/17-binary-search-trees.html>
- [6] ADELSON-VELSKII, G. M.; LANDIS, E. M. An algorithm for the organization of information. Soviet Mathematics Doklady, 3:1259–1263, 1962. [online]. [cit. 2018-07-07] Dostupné z: <http://professor.ufabc.edu.br/~jesus.mena/courses/mc3305-2q-2015/AED2-10-avl-paper.pdf>
- [7] CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Introduction to Algorithms, 978-0-262-03384-8, 1990. [online]. 2011-04-04 [cit. 2018-07-07]. Dostupné z: <http://belohlavek.inf.upol.cz/vyuka/Cormen-RB-trees.pdf>
- [8] GALLES, David. Data Structure Visualizations [online]. [cit. 2018-07-07] Dostupné z: <https://www.cs.usfca.edu/galles/visualization/Algorithms.html>
- [9] ORACLE Java documentation [online]. [cit. 2018-07-07] Dostupné z: <https://docs.oracle.com/javase/8/javase-clienttechnologies.htm>
- [10] BLOCH, Joshua. Effective Java – Second Edition, 978-0321356680, 2001.
- [11] POMALORI, Andreas. JavaFX Programming Cookbook [online]. [cit. 2018-07-07] Dostupné z: <https://www.javacodegeeks.com/minibook/javafx-programming-cookbook>
- [12] TIOBY [online]. [cit. 2018-07-07] Dostupné z: <https://www.tiobe.com/tiobe-index/>