ZALG 3. cvičení

Podmínky udělení zápočtu

- https://github.com/martinnovaak/ZALGcv/blob/main/README.md#podm%C3%ADnky-ud%C4%9Blen%C3%AD-z%C3%A1po%C4%8Dtu

Binární strom složitost operací

- Metody najdi, vlož a smaž lze implementovat pomocí rekurze nebo iterativně (s využitím pomocných ukazatelů a cyklů)
- Varianty:
  - Bez rekurze: prostorová složitost $\mathrm{T}(n) = O(1)$ (paměť pouze pro pomocné ukazatele)
  - S rekurzí: prostorová složitost $T(n) = O(h)$ (h – hloubka rekurze = hloubka větve), výrazně stoupá i časová složitost
- Doporučení: Při implementaci, pokud možno vždy volíme řešení bez rekurze

Vyhledání vrcholu ve stromě

```
Pomocná proměnná:
  x = root
[Postup] Dokud není x == nullptr opakuj:
    a) je-li x->data = key, pak je x hledaný vrchol a cyklus končí
    b) je-li x->data > key, pak x := x->left
    c) je-li x->data < key, pak x := x->right
[Návrat] return x (je-li x nullptr znamená to, že daný prvek se ve stromě nenachází)
```

Vkládání vrcholu s hodnotou `key` do stromu

```
Pomocná proměnná:
  x := root - právě zkoumaný prvek
  inserted := false
[Postup stromem] Opakuj:
    1) je-li x->data = key, pak se již key ve stromě nachází ==> ukonči cyklus
    2) je-li x->data > key, pak:
          2a) Pokud x->left == nullptr ==> x->left = new vrchol ==> inserted := true a ukonči cyklus
          2b) Pokud x->left != nullptr ==> x := x->left ==> jdi na další iteraci (na začátek cyklu)
    3) je-li x->data < key, pak x := x->right
          3a) Pokud x->right == nullptr ==> x->right = new vrchol ==> inserted := true a ukonči cyklus
          3b) Pokud x->right != nullptr ==> x := x->right ==> jdi na další iteraci (na začátek cyklu)
[Návrat] return inserted (prvek byl úspěšně vložen)
```

Jednoduchý výpis stromu

- Binární vyhledávácí strom je setříděná datová struktura
- Dá se proto použít pro setřídění posloupnosti klíčů
- Naším cílem je vypsat na řádce klíče vzestupně od nejmenšího po největší

- 3 Postupy:
  - 1) Rekurzivní
  - 2) Se zásobníkem
  - 3) Použití parent pointerů

Výpis stromu rekurze

```
[Parametr] branch
Není-li branch nullptr:
    Vypiš podstrom branch->left
    Vypiš vrchol branch
    Vypiš podstrom branch->right
```

Výpis stromu zásobník

```
z := zásobník
pom := root   - aktuálně navštívený vrchol
[Cyklus] Dokud není pom nullptr a dokud zásobník není prázdný
    [Cyklus 2] Dokud pom není nullptr
        vlož pom do zásobníku
        pom := pom->left
    [Konec cyklu]
    pom := z.top(), z.pop()
    vypiš pom
    pom := pom->right
[Konec cyklu]
```

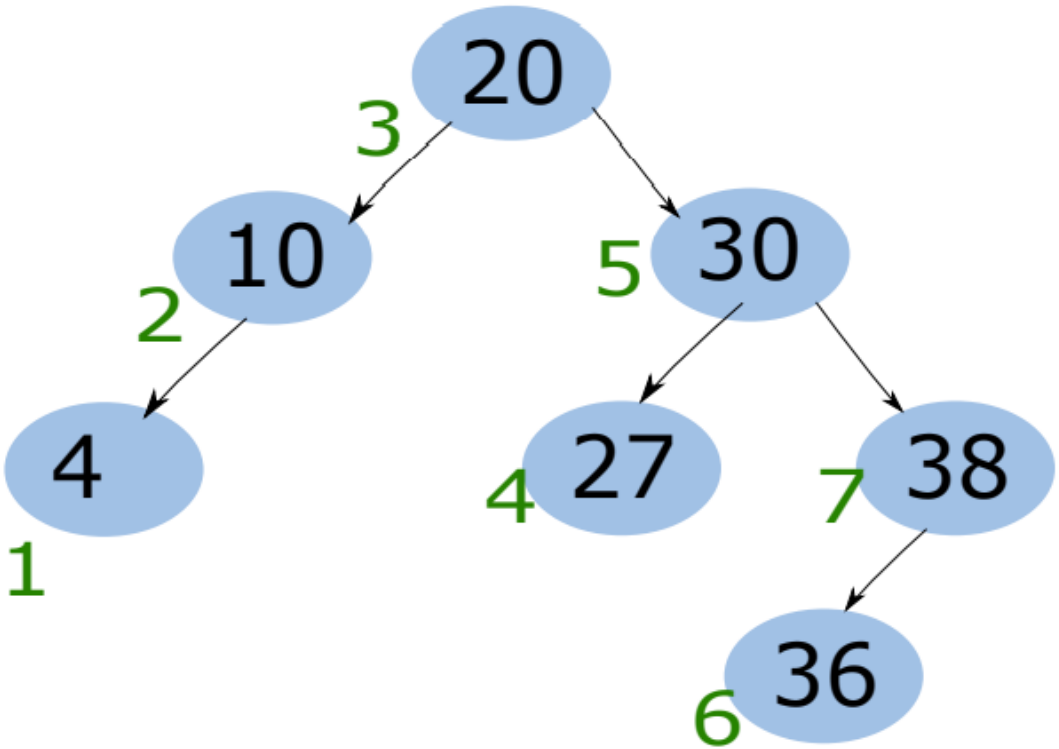
Výpis bez zásobníku i rekurze

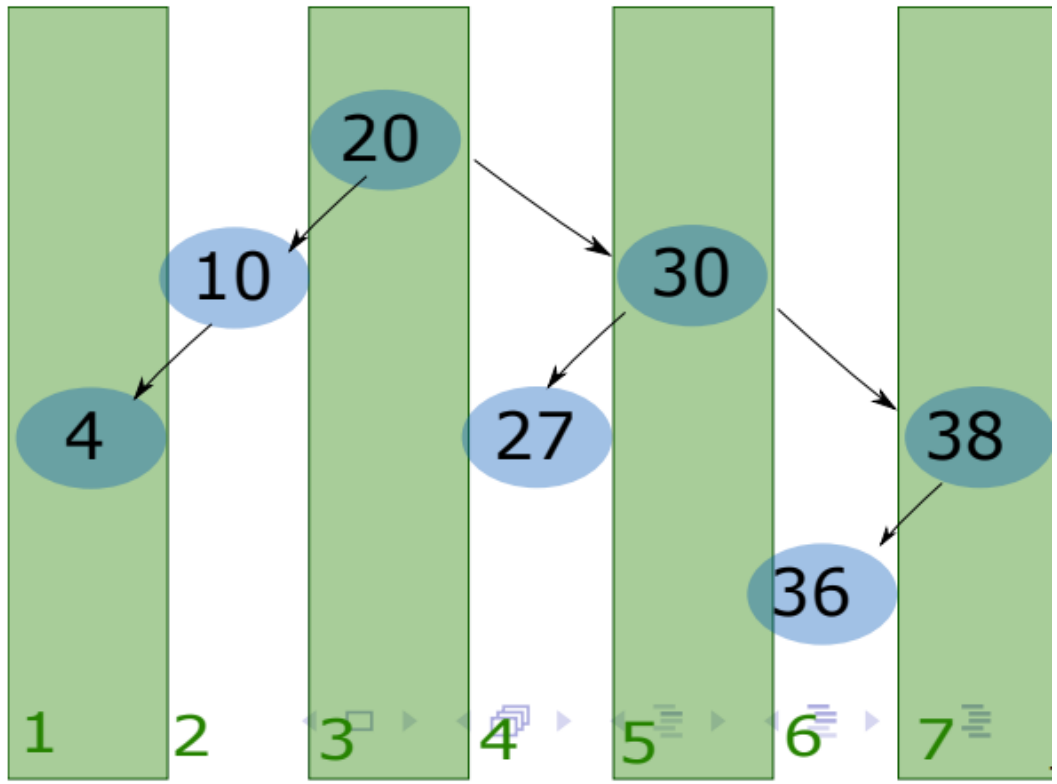
```
Pokud je strom prázdný ==> konec
vrchol pom = min(root)  - do pom ulož nejmenší vrchol ve stromě

[Cyklus] Dokud není pom nullptr
    vypiš pom
    Pokud má pom pravého potomka ==> pom := min(pom->right)
    Jinak:
        father := pom->parent
        [Cyklus] Dokud rodič není nullptr && dokud je pom pravym synem otce
            pom := father;
            father := pom->parent;
        [Konec cyklu]
        pom := father;
[Konec cyklu]
```

Výpis stromu
A) Kompaktní výpis

- Každý vrchol bude vypsán ve svém sloupečku šířky n
- Pro každý vrchol si spočteme pořadí zleva
- i-tý vrchol zleva odsadíme o $n \cdot i$ mezer
- Pořadí vrcholů zleva si musíme předpočítat a uložit jako atribut vrcholu (rozroste se struktura vrcholu)


20
3
10
5
30
2
4
27
38
4
7
1
36
6


20
10
30
4
27
38
36
1
2
3
4
5
6
7

B) Široký výpis

- Výpis stromu jako by byl dokonalý
- Strom bude symetrický s prázdnými sloupečky pro chybějící vrcholy
- Ve stromě o hloubce h může být až $2^h - 1$ vrcholů

Hloubka stromu

```
x := root
q := queue - fronta obsahující i-tou úroveň
height := 0
Je-li root == nullptr return hloubka := 0

vlož kořen do fronty queue

[Cyklus] Opakuj dokud není queue prázdná
 level_size := queue.size()
   [Cyklus] Pro každý prvek v úrovni h
      pom := q.front() a q.pop()
      vlož potomky pom do queue
   [Konec cyklu]
  height = height + 1
[Konec cyklu]
[Návrat] Vrať hloubku stromu
```

Široký výpis stromu

```
h := hloubka stromu
space := 2^(h-1)  - Mezera na řádku i
queue - Fronta obsahující i-tou úroveň

[Cyklus1] Opakuj pro i = 0 do h (i < h)
    level_size := queue.size()
    [Cyklus2] Pro každý prvek v úrovni h
        pom := queue.front a queue.pop()
        vlož potomky pom do queue
        vypiš space-1 mezer
        vypiš pom
        vypiš space mezer
    [Konec cyklu2]
    space = space / 2
    přejdi na další řádek
[Konec cyklu1]
```

STL knihovny s binárními stromy

- std::set – setříděný kontejner obsahující množinu jedinečných objektů typu klíč
- std::map - setříděný kontejner obsahující dvojice klíč-hodnota

Nevýhody

- Při vkládání klíčů v nevhodném pořadí mohou operace stromu mít asymptotickou složitost $O(n)$ místo chtěných $O(logn)$

- Jak zajistit operace $O(logn)$ za každé situace?

Samovyvažovací stromy (Self-balancing trees)

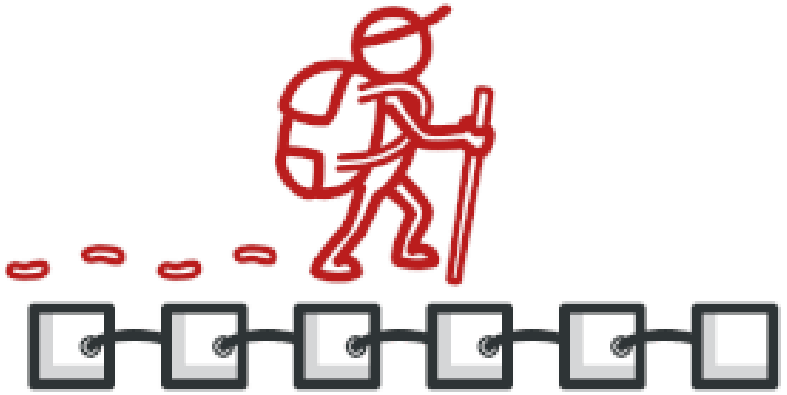
3. Zápočtová úloha – AVL strom

4. Zápočtová úloha – Červeno-černý strom

- Red-Black Tree

Iterátor

- Iterátor je návrhový vzor chování, který umožňuje procházet prvky kolekce, aniž by byla odhalena její základní reprezentace (seznam, zásobník, strom atd.).



```
class iterator
{
    vertex * current;
public:
    explicit iterator(vertex * start) : current(start) {}
    const V& operator*() {return current->data;}
    iterator& operator++() {
        if (current->right != nullptr) {
            current = min( branch: current->right);
        } else {
            vertex * parent = current->parent;
            while (parent != nullptr && current == parent->right) {
                current = parent;
                parent = parent->parent;
            }
            current = parent;
        }
        return *this;
    }
    bool operator!=(const iterator& other) const {
        return current != other.current;
    }
};

iterator begin() {return iterator( start: min( branch: root));}
iterator end() {return iterator( start: nullptr);}
```