DELTA – Střední škola informatiky a ekonomie, Základní škola a Mateřská škola, s.r.o.

Ke Kamenci 151, PARDUBICE

MATURITNÍ PROJEKT

IMPLEMENTACE VYBRANÝCH PROBLÉMŮ ABSTRAKTNÍCH DATOVÝCH TYPŮ

Šimon Hyksa

4.B

Informační technologie 18-20-M/01

2022/2023

Zadání maturitního projektu z informatických předmětů

Jméno a příjmení: Šimon Hyksa

Pro školní rok: 2022/2023

Třída: 4.

Obor: Informační technologie 18-20-M/01

Téma práce: Řešení vybraných problémů abstraktních datových typů

Vedoucí práce: Mgr. Josef Horálek, Ph.D.

Způsob zpracování, cíle práce, pokyny k obsahu a rozsahu práce:

Cílem maturitního projektu je podrobně popsat a realizovat vybrané problémy v oblasti algoritmiky abstraktních datových typů (dále ADT) zaměřených na ADT strom a halda. Autor zpracuje problematiku algoritmické reprezentace ADT a základní práce s těmito ADT. Dále autor zpravuje vybrané metody práce s ADT zejména s binárním stromem a haldou. V praktické části pak autor vytvoří sadu podrobně popsaných implementací vybraných problémů ve vybraném objektově orientovaném jazyce.

Stručný časový harmonogram (s daty a konkretizovanými úkoly):

09/2022 – 10/2022 Analýza problematiky ADT

11/2022 – 12/2022 Popis principů a využití ATD halda a strom

12/2022 – 02/2023 Implementace vybraných problémů

02/2023 – 03/2023 Testování vybraných řešení

03/2023 Finalizace textového znění maturitního projektu

Prohlašuji, že jsem maturitní projekt vypracoval samostatně, výhradně s použitím uvedené literatury.

V Pardubicích 31. 3. 2023 …….....

PODĚKOVÁNÍ

Chci poděkovat Mgr. Josefu Horálkovi, Ph.D. za odborné vedení maturitního projektu a dodání potřebných studijních materiálů.

RESUMÉ

Tato maturitní práce popisuje vybrané problémy abstraktních datových typů se zaměřením na haldu a strom. V teoretické části se budu zabývat samotnými datovými typy a použitými algoritmy k řešení vybraných problémů.

V praktické části ukážu, jak lze implementovat vybrané problémy za použití programovacího jazyka Rust.

KLÍČOVÁ SLOVA

TUI, garbage collector, IDE, PATH, Git, GZip, rustc, cargo, hashmap, borrow checker, rust-analyzer, Visual Studio Code, node, root, heapsort, heapify, JPEG, MP3, char, i32, Traversal, bootstrap

ANOTATION

This senior thesis describes selected problems of abstract data types with a focus on heap and tree. In the theoretical part I will discuss the data types themselves and the algorithms used to solve the selected problems.

In the practical part I will show how the selected problems can be implemented using the Rust programming language.

KEY WORDS

TUI, garbage collector, IDE, PATH, Git, GZip, rustc, cargo, hashmap, borrow checker, rust-analyzer, Visual Studio Code, node, root, heapsort, heapify, JPEG, MP3, char, i32, Traversal, bootstrap

Obsah

1. Úvod do maturitní práce

Cílem maturitní práce je podrobně popsat a realizovat vybrané problémy v oblasti algoritmiky abstraktních datových typů (dále ADT) zaměřených na ADT binární strom a haldu.

Tento maturitní projekt bude využit jako doprovodný studijní materiál předmětu algoritmy, jehož cílem je seznámit studenty se základními principy struktury, využití a charakteristik vybraných algoritmických problémů. Mezi obtížnější části problematiky algoritmů patří práce s abstraktními datovými typy. Pro naplnění cílů této práce jsou teoreticky popsány principy fungování abstraktního datového typu haldy a binárního stromu včetně vybraných metod a postupů pro jejich vyhledávání.

Praktická realizace celého projektu je navržena v programovacím jazyce Rust, jehož kódová část je podrobně popsána jako součást dokumentace. Výstupem práce je aplikace s jednotným TUI rozhraním, které umožňuje uživateli (studentovi) výběr konkrétního algoritmu a jeho implementaci.

2. Technologie

Pro implementaci jednotlivých algoritmů jsem využil programovací jazyk Rust. Výběr programovacího jazyka Rust jsem zvolil pro jeho výkonnost ale i bohatý typový ekosystém. V neposlední řadě moje volba jazyku byla ovlivněna jeho popularitou mezi vývojáři, která má stoupající tendenci. Umožňuje uživateli využívat jazyk k různým činnostem, jako je vývoj webových aplikací nebo programů pro ovládání nízko-úrovňových prvků. Dále jsem do svého maturitního projektu přidal podprojekt, který funguje jako „bootstrap“ a díky němu moje implementace řešení fungují jako jednotlivé spustitelné podprogramy.

2.1. Úvod do jazyka Rust

[3] Hlavním důvodem, proč jsem si vybral tento jazyk, jsou jeho dominanty jako je výkon, typová bezpečnost a souběžnost s ostatními programovacími jazyky. Jeho vysoký výkon spočívá v absenci „garbage collectoru“ a nahrazuje ho „borrow checker“, který sleduje životnost objektu všech odkazů v programu během kompilace. Jazyk prochází pravidelnými aktualizacemi, díky kterým nezaostává za konkurenčními programovacími jazyky jako je například C/C++. Typovou bezpečností jazyku se rozumí zamezení operací vedoucích k chybám.

2.2. Instalace jazyka pro Windows 10

Pro instalaci jazyka budeme nejdříve potřebovat nainstalovat nástroj „rustup“, který nám bude nadále spravovat celý jazyk včetně nástrojů „cargo“ a „rustc“.

URL ke stažení: https://www.rust-lang.org/tools/install.

Dále se musíme ujistit, že adresář `~/.cargo/bin` se nachází v proměnné prostředí PATH.

Jako poslední krok je potřeba nastavit naše IDE, například Visual Studio Code, tak, aby nám při vývoji v Rustu bylo co nejvíce užitečné. Visual Studio Code nabízí přímo rozšíření s názvem [12] „rust-analyzer“, které pomáhá s debugováním. Využívá [13] Language Server Protocol (LSP), který nám přidává užitečné funkce jako je automatické doplňování datových typů nebo vysvětlující popis pro určitou položku v kódu.

2.3. Spuštění projektu

Ke spuštění budeme potřebovat verzovací nástroj Git.

URL ke stažení: https://git-scm.com/download/win

K ověření úspěšné instalace spustíme příkaz `git --version`

Poté skrze příkaz *`git clone https://github.com/Werdix/adt_bootstrap.git`* si naklonujeme repozitář.

Finální krok bude sestavení „bootstrapu“. Přesuneme se do složky „cursive_cli“ skrze příkaz *`cd ./cursive_cli/`* a provedeme příkaz na kompilaci a spuštění `cargo run`.

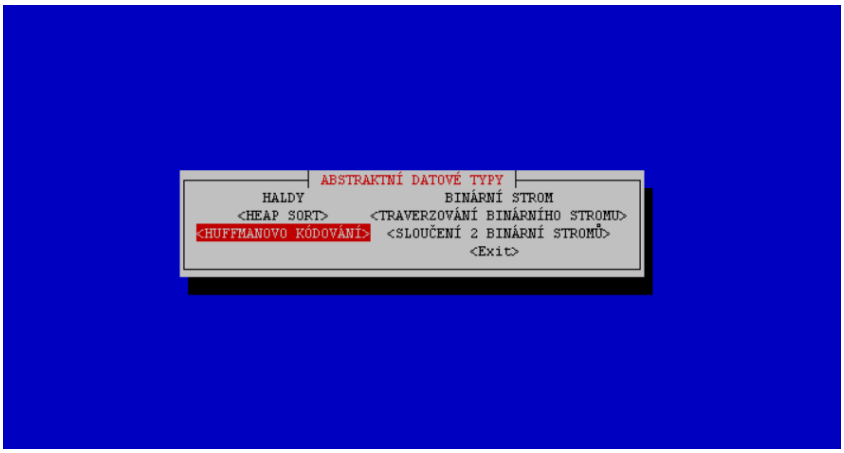
2.4. Externí knihovny

K vytvoření uživatelského prostředí byla použita knihovna [11] „Cursive“, která primárně slouží k vytváření TUI aplikací. Na rozdíl od knihovny [14] „tui-rs“ je její syntaxe výrazně jednodušší. Ve většině případů se jedná o aplikace pro operační systém Linux.

2.5. Aplikace

Po spuštění aplikace se zobrazí menu, kde si uživatel může vyzkoušet mé implementace řešení. Po kliknutí na vybrané řešení se ukáže dialogové okno, do kterého se budou zapisovat vstupní parametry. Pro všechny řešení, kromě „Huffmanova kódování“, je potřeba prvky vstupních parametrů oddělovat středníkem (;).

Ukázka validního vstupu: 18;457;-666;221;54;32;-15;69;47



Obrázek 1: Hlavní menu aplikace

3. Úvod do abstraktních datových typů

Abstraktní datový typ je datová struktura, která je implementována programátorem, aby usnadnila práci s daty a zlepšila čitelnost kódu.

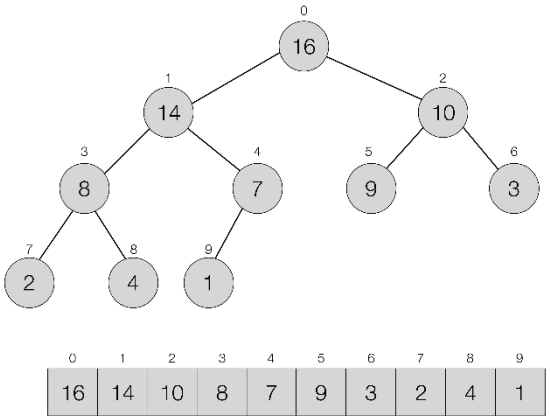
Existují různé implementace abstraktních datových typů, mezi které patří seznamy, fronty, množiny, zásobníky, stromy a grafy. Například zásobník umožňuje vkládat hodnoty za sebe tak, že poslední vkládaný prvek bude vyjmut jako první. Fronta zase ukládá hodnoty tak, aby první prvek byl vyjmut jako první. To například využívá aplikace Spotify, kde si uživatel může vybrané skladby vložit do fronty a budou se postupně přehrávat.

Já se budu v této práci zabývat haldou a binárním stromem. I když jsou si tyto dva abstraktní datové typy podobné, jsou mezi nimi podstatné rozdíly. Například binární strom nemůže být reprezentovaný jako pole, zatímco halda může, protože u binárního stromu nelze přesně určit indexy jeho uzlů.

4. Halda

[4] Halda je výjimečná datová struktura založená na principu stromu. Každá položka v haldě je nazývána „uzel“ (*Node*). Jako „kořenový uzel“ (*Root*) označujeme ten uzel, který nemá žádného rodiče. Kořenem se rozumí uzel, který má největší hodnotu.

K vizualizaci haldy můžeme využít jedno-rozměrné pole nebo stromovou strukturu. Na rozdíl od stromové vizualizace, není v poli jednoznačná spojitost mezi jednotlivými „uzly“. Vezmeme uzel s indexem *i*, který bude reprezentovat rodičovský uzel. V haldě dále platí pravidlo, že uzel může mít maximálně dva potomky, levý a pravý. Index levého potomka je tedy potom *2*i + 1* a index pravého potomka *2*i + 2*.

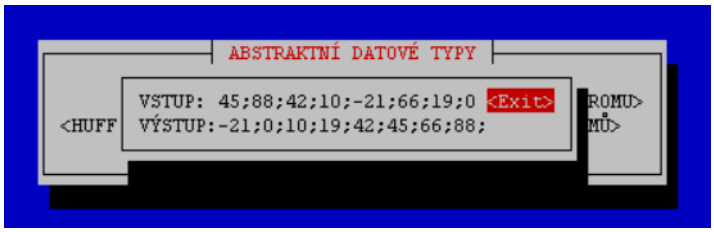


Obrázek 2: Vyobrazení haldy

4.1. Heapsort

[6] „Třídění haldou“ neboli Heapsort patří mezi nejefektivnější třídící algoritmy. Jeho časová složitost je O(n log n), což znamená, že je velmi efektivní pro řazení velkých seznamů dat. Algoritmus dělíme na dvě části: „Třídění“ (heapsort) a „Zhaldování“ (heapify).

Obrázek 3: Vstupní parametry (Heapsort)



Obrázek 4: Výstup (Heapsort)

4.1.1. Průběh „Zhaldování“ (heapify)

Funkce „Zhaldování“ upraví naše pole tak, aby splňovalo podmínky být haldou, tedy přesune největší prvek na index kořenu haldy.

```
    fn max_heapify(arr: &mut [i32], mut root: usize) {
        /*Zhalduje pole, neboli převede největší položku do kořenu */
        let last: usize = arr.len() - 1;
        loop {
            let left = 2 * root + 1; //index levého childu
            if left > last {
                break;
            }
            let right = left + 1; //index pravého childu

            //Podmínka pro zjištění, zda je potomek levý nebo pravý
            let child = if right <= last && arr[right] > arr[left] {
                right
            } else {
                left
            };

            //pokud má potomek větší hodnotu než kořen, tak si vymění
místo
            if arr[child] > arr[root] {
                arr.swap(root, child);
            }
            root = child;}}
```

Kód 1: Funkce heapify ("Zhaldování")

4.1.2. Průběh „Třídění“ (heapsort)

Poté co naše pole splňuje podmínku být haldou, přesuneme kořen haldy (největší prvek) na úplný konec pole a snížíme délku haldy o 1. Tím jsme seřadili poslední prvek, se kterým již nebudeme nadále pracovat.

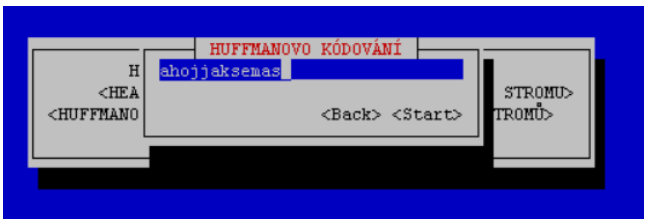
```
fn heap_sort(arr: &mut [i32]) {
    if arr.len() <= 1 {
        return;
    }

    //Zhaldění
    let last_parent = (arr.len() - 2) / 2;
    for i in (0..=last_parent).rev() {
        max_heapify(arr, i);
    }
    //Řazení
    for end in (1..arr.len()).rev() {
        arr.swap(0, end);
        max_heapify(&mut arr[..end], 0);}}
```

Kód 2: Funkce heapsort ("Třídění")

4.2. Huffmanovo kódování

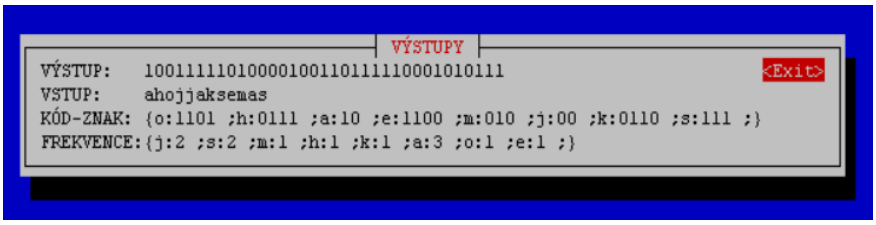
Je jednoduchý bezztrátový kompresní algoritmus. Byl vyvinut americkým průkopníkem počítačové vědy Davidem Albertem Huffmanem (*1925 – 1999). Jeho algoritmus je využíván například programem GZip nebo kodeky multimédií jako JPEG a MP3.

K vytvoření „kompresního kódu“ využívá abstraktní datový typ halda, kde každý prvek má znak ze vstupního parametru, jeho frekvenci vyjádřenou číslem a samotný binární kód. Za protikladný algoritmus, ve smyslu způsobu vytvoření „kompresního kódu“, můžeme považovat algoritmus Shannonovo-Fanovo kódování. Jeho „kompresní kód“ je sestaven



Obrázek 5: Vstupní parametr (Huffmanovo kódování)

od kořene a dále pokračuje k dalším listům, zatím co Huffmanovo kódování začíná u listů a pokračuje dál ke kořeni. Díky své jednoduchosti je Huffmanovo kódování obecně rychlejší.



Obrázek 6: Výstup (Huffmanovo kódování)

4.2.1. Frekvence

Prvotně je potřeba zjistit frekvenci všech znaků v řetězci, který nám byl dán. Dále zjištěné informace budeme ukládat do hashmapy s klíčem typu char a hodnotou typu i32.

```
    /*Funkce pro zjištění frekvence jednotlivých charakterů v řetězci
*/
    pub fn frequency(s: &str) -> HashMap<char, i32> {
        let mut h = HashMap::new();
        /*cyklus pro procházení řetězce znak po znaku */
        for chr in s.chars() {
            /*Pokud se klíč(znak) již je v Hashmapě nachází, tak se
jeho hodnota(frekvence) zvýší o 1*/
            let count = h.entry(chr).or_insert(0);
            *count += 1;
        }
        return h;}
```

Kód 3: Frekvence znaků

4.2.2. Huffmanův strom

Pro každou položku v hashmapě vytvoříme uzel (*Node*), který poté vložíme do pole. Následně vybereme z pole dva uzly s největší frekvencí. Tyto dva uzly budou následně potomci uzlu, který bude mít uložený součet frekvencí těchto dvou uzlů.

```
    struct Node {
        ch: Option<char>,
        freq: i32,
        left: Option<Box<Node>>,
        right: Option<Box<Node>>,}
```

Kód 4: Struktura Node

```
    let freq_map: HashMap<char, i32> = frequency(to_encode);
    let mut p: Vec<Box<Node>> = freq_map
            .iter()
            .map(|x| new_box(new_node(*(x.1), Some(*(x.0)))))
            .collect();
        /*Vytvoření Huffmanova stromu */
        while p.len() > 1 {
            p.sort_by(|a, b| (&(b.freq)).cmp(&(a.freq)));
            let a = p.pop().unwrap();
            let b = p.pop().unwrap();
            let mut c = new_box(new_node(a.freq + b.freq, None));
            c.left = Some(a);
            c.right = Some(b);
            p.push(c);
        }
        let root = p.pop().unwrap();
```

Kód 5: Vytvoření Huffmanova stromu

4.2.3. Generování binárního kódu

Pro další krok ke kódování je potřeba vytvořit druhou hashmapu, do které budeme ukládat znak jako klíč a jeho kompresní kód jako hodnotu. Rekurzivním způsobem projdeme celý Huffmanův strom. Kompresní kód bude vytvořen v závislosti na poloze uzlu, tedy zda je levý nebo pravý. Čím menší má znak frekvenci, tím delší bude jeho kompresní kód.

```
    /*Generování binárního kódu pro node*/
    fn generate_codes(p: &Box<Node>, h: &mut HashMap<char, String>, s:
String) {
        if let Some(ch) = p.ch {
            h.insert(ch, s);
        } else {
            if let Some(ref l) = p.left {
                generate_codes(l, h, s.clone() + "0");
            }
            if let Some(ref r) = p.right {
                generate_codes(r, h, s.clone() + "1");
            }
        }
    }
```

Kód 6: Generování binárního kódu

4.2.4. Komprese vstupu

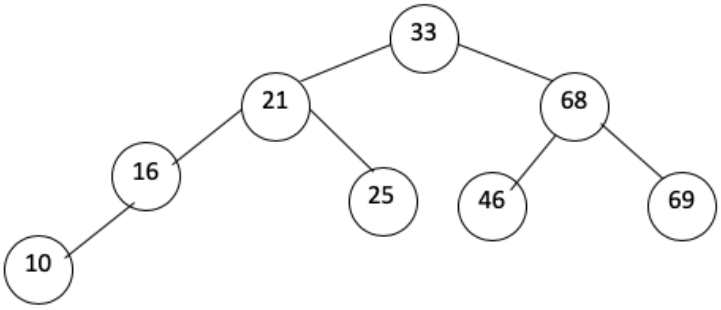
Funkce, která nakonec „zakóduje“ vstupní hodnotu, nám projde vstupní parametr znak po znaku a najde k němu příslušnou hodnotu v hashmapě. Hodnotou se rozumí přidělený kompresní kód. Tento kód se následně „vsune“ do proměnné *r*, kterou nám funkce vrátí.

```
    fn encode_string(s: &str, h: &HashMap<char, String>) -> String {
        let mut r = "".to_string();
        let mut t: Option<&String>;

        for ch in s.chars() {
            t = h.get(&ch);
            r.push_str(t.unwrap());} return r;}
```

Kód 7: Funkce pro kompresi vstupu

5. Binární strom

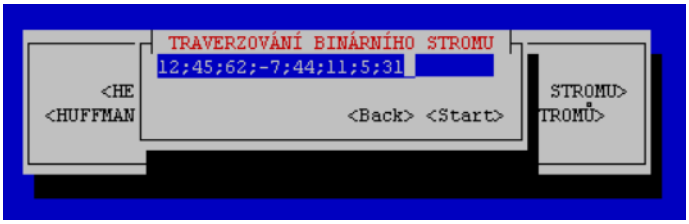
[2] Je stromová datová struktura, kde každý uzel (Node) má nejvýše dva potomky, kteří se dělí na levý a pravý. Platí také podmínka, že levý potomek je vždy menší než jeho rodič a pravý potomek je vždy větší než jeho rodič. Na následujícím obrázku je vyobrazení jednoduchého binárního stromu.



Obrázek 7: Vyobrazení binárního stromu

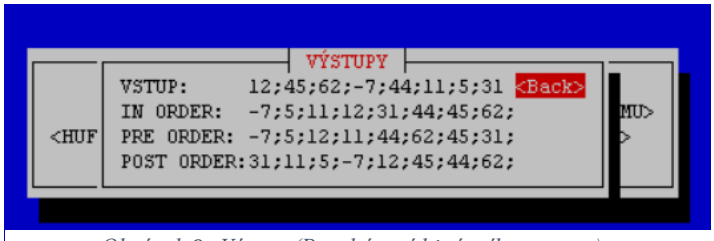
5.1. Procházení binárním stromem

K sestavení binárního stromu je potřeba všechny prvky pole seřadit vzestupně.



Obrázek 8: Vstupní parametry (Procházení binárního stromu)

Následně získáme index prvku, který se nachází na středu pole, pomocí něhož můžeme rozdělit na pravý a levý podstrom. Následující funkce vytvoří uzel, který bude mít parametr *right* a *left* jako odkaz na další uzly.



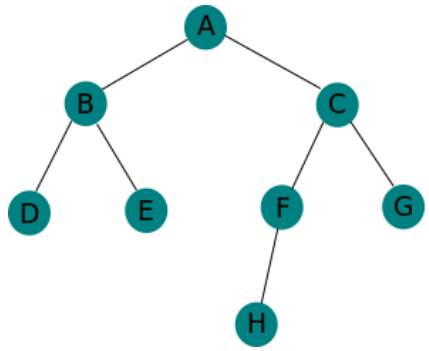
Obrázek 9: Výstup (Procházení binárního stromu)

```
    struct Node {
        item: i32,
        left: Option<Box<Node>>,
        right: Option<Box<Node>>,}

    /*Strom můžeme sestavit pouze v případě, že je pole seřazeno
vzestupně*/
    fn build_tree(vals: Vec<i32>) -> Option<Box<Node>> {
        if vals.is_empty() {
            return None;
        }
        let mid = vals.len() / 2;
        let item = vals[mid];

        let left = build_tree(vals[0..mid].to_vec());
        let right = build_tree(vals[mid + 1..].to_vec());
        return Some(Box::new(Node { item, left, right }));}
```

Kód 8: Sestavení binárního stromu – funkce build_tree

5.1.1. IN - ORDER Traversal

IN – ORDER (v pořadí) Traversal začíná v levém podstromu, přesune se do rodičovského uzlu a následně do pravého podstromu. Výsledkem bude tedy setříděný výpis uzlů.



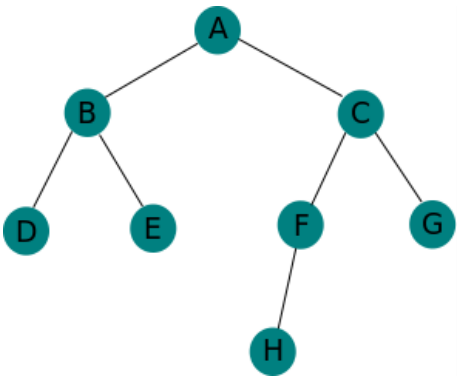
Obrázek 10: Binární strom - IN ORDER

Postup průchodu: D, B, E, A, H, F, C, G

```
    /*začíná v nejmenším, končí v největším */
    fn in_order_traversal(root: &Option<Box<Node>>) -> String {
        let mut tree: String = String::new();
        if let Some(node) = root {
            tree.push_str(&in_order_traversal(&node.left));
            tree.push_str((node.item.to_string()).as_str());
            tree.push(';');
            tree.push_str(&in_order_traversal(&node.right));
        }return tree;}
```

Kód 9: IN - ORDER Traversal

5.1.2. POST – ORDER Traversal



Obrázek 11: Binární strom - POST ORDER

POST – ORDER Traversal funguje na podobném principu jako algoritmus „prohledávání do hloubky“. Začne v kořeni stromu a postupuje dále do levého podstromu. Až s ním skončí přesune se na pravý podstrom.

Postup průchodu: A, B, D, E, C, F, H, G

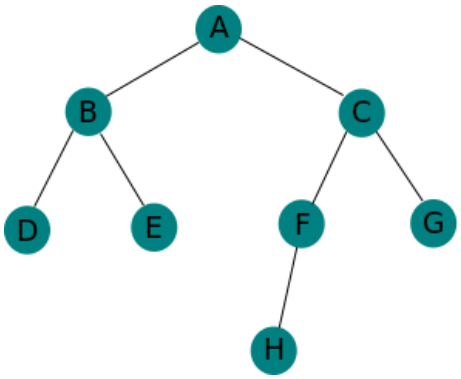
```
    /*Funguje na podobném principu jako DFS (začíná v rootu, končí ve
spodním pravém uzlu */
    fn post_order_traversal(root: &Option<Box<Node>>) -> String {
        let mut tree: String = String::new();
        if let Some(node) = root {
            tree.push_str(&post_order_traversal(&node.left));
            tree.push_str(&post_order_traversal(&node.right));
            tree.push_str((node.item.to_string()).as_str());
            tree.push(';');
        }
        return tree;
    }
```

Kód 10: POST - ORDER Traversal

5.1.3. PRE – ORDER Traversal

PRE – ORDER Traversal začíná v levém podstromu, pak se přesune do pravého podstromu a poté do rodičovského uzlu.

Postup průchodu: D, E, B, H, F, G, C, A



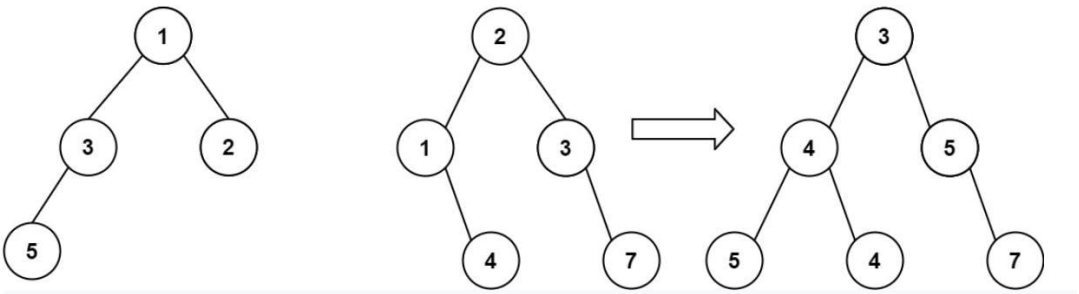
Obrázek 12: Binární strom - PRE ORDER

```
fn pre_order_traversal(root: &Option<Box<Node>>) -> String {
    let mut tree: String = String::new();
    if let Some(node) = root {
        tree.push_str((node.item.to_string()).as_str());
        tree.push(';');
        tree.push_str(&pre_order_traversal(&node.left));
        tree.push_str(&pre_order_traversal(&node.right));
    }
    return tree;}
```

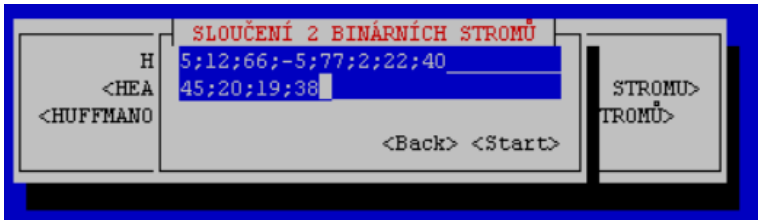
Kód 11: PRE - ORDER Traversal

5.2. Sloučení dvou binárních stromů

[5] Princip sloučení dvou binárních stromů je prostý. Poté co se vytvoří dva binární stromy, zkontroluje se, zda se některé uzly překrývají. Pokud se překrývají, sečtou se jejich hodnoty a jsou vloženy do nového uzlu v novém binárním stromu. V případě, že se uzly nepřekrývají, vloží se do nového binárního stromu jako uzel s aktuální hodnotou. Zde je vyobrazen příklad jednoduchého sloučení dvou binárních stromů.

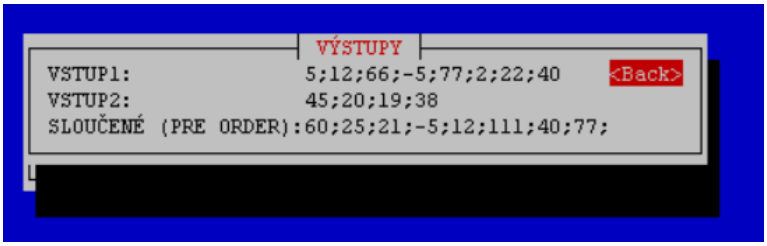


Obrázek 13: Vyobrazení sloučení dvou binárních stromů



Obrázek 14: Vstupní parametry (Sloučení dvou binárních stromů)

Součástí algoritmu pro sloučení dvou binárních stromů využijeme funkci *build_tree* pro vytvoření binárního stromu z předchozího řešení.



Obrázek 15: Výstup (Sloučení dvou binárních stromů)

Funkce *merge_trees* dostane dva vstupní parametry, které budou kořenové uzly již sestavených stromů. Pokud oba stromy obsahují uzel na stejné pozici, například „kořen“, vytvoří nový uzel, který bude obsahovat součet hodnot „kořenů“ z prvního a druhého stromu.

```
     /*Funkce pro sloučení binární stromů*/
     fn merge_trees(tree1: Option<Box<Node>>, tree2: Option<Box<Node>>)
-> Option<Box<Node>> {
         /*Rekurzivní průchod */
         match (tree1, tree2){
             /*Pokud oba stromy obsahují uzel na stejné pozici, jejich
hodnota se sečte do nového uzlu na stejné pozici v */
             (Some(node1), Some(node2)) => {
                 return Some(Box::new(Node {
                     item: node1.item + node2.item,
                     left: merge_trees(node1.left, node2.left),
                     right: merge_trees(node1.right, node2.right),
                 }))},
             /*Pokud se uzel s touto pozicí nachází pouze v jednom ze
stromů, přidá se jako nový uzel do nového stromu na stejnou pozici*/
             (None, Some(node2)) => return Some(node2),
             (Some(node1), None) => return Some(node1),
     (None, None) => return None,}}
```

Kód 12: Sloučení dvou binárních stromů

Závěr

Cílem mého maturitního projektu, bylo podrobně popsat a realizovat vybrané problémy v oblasti algoritmů abstraktních datových typů (dále ADT) zaměřených na ADT binární strom a halda. Pro naplnění cílů této práce byly teoreticky popsány principy fungování abstraktního datového typu halda a binární strom včetně vybraných metod a postupů pro jejich vyhledávání. Samotné téma ADT je obsahově velmi rozsáhlé v oblasti programování. Ve svém projektu jsem se věnoval základům ADT, které mi pomohly objasnit jejich funkčnost.

Výstupem práce je aplikace s jednotným TUI rozhraním, která umožňuje uživateli (studentovi) výběr konkrétního algoritmu a jeho implementaci. Aplikaci je možno dále rozšiřovat o další algoritmy, které mohou pomoci uživateli (studentovi) pochopit jejich principy.

Seznam obrázků

Citovaná literatura

[1] Huffman Coding. *Programiz: Learn to Code for Free* [online]. Parewa Labs Pvt. [cit. 2023-03-22]. Dostupné z: https://www.programiz.com/dsa/huffman-coding

[2] GeeksforGeeks. Binary Tree Data Structure [online]. 2020-12-14 [2023-03-22]. Dostupné z: https://www.geeksforgeeks.org/binary-tree-data-structure/

[3] Přispěvatelé Wikipedie, *Rust (programovací jazyk)* [online], Wikipedie: Otevřená encyklopedie, c2023, Datum poslední revize 11. 03. 2023, 14:42 UTC, [citováno 22. 03. 2023] https://cs.wikipedia.org/w/index.php?title=Rust_(programovac%C3%AD_jazyk)&oldid=22535208

[4] GeeksforGeeks. Heap Data Structure [online]. 2019-11-18 [2023-03-22]. Dostupné z: https://www.geeksforgeeks.org/heap-data-structure/

[5] GeeksforGeeks. Merge two binary trees by doing node sum (recursive and iterative) [online]. 2019-01-23 [2023-03-22]. Dostupné z: https://www.geeksforgeeks.org/merge-two-binary-trees-node-sum/

[6] ITnetwork.cz. Algoritmus Heap sort - třídění čísel podle velikosti [online]. 2014 [2023-03-22]. Dostupné z: https://www.itnetwork.cz/algoritmy/razeni/algoritmus_heap_sort_trideni_cisel_podle_velikosti

[7] WRÓBLEWSKI, Piotr. *Algoritmy*. Brno: Computer Press, 2015. ISBN 978-80-251-4126-7.

[8] LeetCode. Merge two binary trees by doing node sum (recursive and iterative) [online]. 2021-02-05 [2023-03-22]. Dostupné z: https://assets.leetcode.com/uploads/2021/02/05/merge.jpg

[9] Sergey, Ilya. Heaps [online]. 2019 [2023-03-22]. Dostupné z: https://ilyasergey.net/YSC2229/_images/heaps.png

[10] Chegg Study. Binary Tree [online]. 2018 [2023-03-22]. Dostupné z: https://media.cheggcdn.com/media/223/2237fccf-d3ee-42b5-a2ff-7ab01ace5cb8/phpwbvLrs.png

[11] Gyscos. Cursive [softwarová knihovna]. 2021-03-10 [2023-03-22]. Dostupné z: https://github.com/gyscos/cursive

[12] Rust-analyzer.github.io. [online]. [cit. 2023-03-27]. Dostupné z: https://rust-analyzer.github.io/

[13] Microsoft.github.io. [online]. [cit. 2023-03-27]. Dostupné z: https://microsoft.github.io/language-server-protocol/

[14] Fdehau. tui-rs [softwarová knihovna]. [cit. 2023-03-27]. Dostupné z: https://github.com/fdehau/tui-rs

[15] Algoritmus: Strom [online]. www.algoritmy.net: Jan Neckář, 2016 [cit. 2023-03-28]. Dostupné z: https://algoritmy.net/article/104/Strom

[16] Binary Search Tree (BST) Traversals – Inorder, Preorder, Post Order - GeeksforGeeks [online]. 2023 [cit. 2023-03-28]. Dostupné z: https://www.geeksforgeeks.org/binary-search-tree-traversal-inorder-preorder-post-order/

[17] Binary data structures: an intro to trees and heaps in JavaScript [online]. 2019 [cit. 2023-03-28]. Dostupné z: https://www.freecodecamp.org/news/binary-data-structures-an-intro-to-trees-and-heaps-in-javascript-962ab536cb42/

[18] The Rust Programming Language - The Rust Programming Language: an intro to trees and heaps in JavaScript [online]. 2022 [cit. 2023-03-28]. Dostupné z: https://doc.rust-lang.org/stable/book/

[19] Heap Data Structure: Programiz: Learn to Code for Free [online]. Parewa Labs Pvt. [cit. 2023-03-28]. Dostupné z: https://www.programiz.com/dsa/heap-data-structure

[20] Easy Rust [online]. 2020 [cit. 2023-03-28]. Dostupné z: https://github.com/Dhghomon/easy_rust