EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 17300/B/2011/1541413018

Štruktúrované programovanie v jazyku C

Bakalárska práca

2011 **Bohuslav Kolek**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Štruktúrované programovanie v jazyku C

Bakalárska práca

Študijný program: Hospodárska informatika a účtovníctvo (Medziodborové štúdium, bakalársky I. st., externá forma)

Študijný odbor: 6292 7 03 Hospodárska informatika a účtovníctvo

Školiace pracovisko: KAI FHI - Katedra aplikovanej informatiky

Školiteľ: Ing. Ján Hanák, PhD.

Bratislava 2011 **Bohuslav Kolek**

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bohuslav Kolek

Študijný program: Hospodárska informatika a účtovníctvo (Medziodborové štúdium, bakalársky I. st., externá forma)

Študijný odbor: 6292 7 03 Hospodárska informatika a účtovníctvo

Typ záverečnej práce: Bakalárska záverečná práca

Jazyk záverečnej práce: slovenský

Názov: Štruktúrované programovanie v jazyku C

Anotácia: Teoreticko-praktický rozbor základných princípov štruktúrovaného programovania v jazyku C.

Vedúci: Ing. Ján Hanák, PhD.

Katedra: KAI FHI - Katedra aplikovanej informatiky FHI

Vedúci katedry: doc. Ing. Gabriela Kristová, CSc.

Dátum schválenia: 19.10.2010

....

študent vedúci

Čestné vyhlásenie

Čestne vyhlasujem, že záverečnú prácu som vypracoval samostatne a že som uviedol všetku použitú literatúru.

Dátum: ....
(podpis študenta)

Ďakujem svojmu vedúcemu bakalárskej práce, Ing. Jánovi Hanákovi, PhD. za cenné rady a pripomienky pri písaní tejto práce.

ABSTRAKT

KOLEK, Bohuslav: *Štruktúrované programovanie v jazyku C.* – Ekonomická univerzita v Bratislave. Fakulta hospodárskej informatiky; Katedra aplikovanej informatiky – Vedúci záverečnej práce: Ing. Ján Hanák, PhD. – Bratislava: FHI, 2011, 45s.

Cieľom záverečnej práce bolo poskytnúť základné informácie o triedení (usporiadaní) množiny údajov a o základných dátových štruktúrach (lineárne a stromové). Práca je rozdelená do štyroch kapitol. Obsahuje 4 tabuľky a 14 obrázkov. Prvá kapitola je venovaná popísaniu základných vlastností triediacich algoritmov a dátových štruktúr. Druha a tretia kapitola popisujú cieľ práce a metodiku práce. V štvrtej kapitole sme popísali prácu vybraných triediacich algoritmov a dátových štruktúr a implementovali ich v jazyku C. Výsledkom riešenia danej problematiky sú algoritmy v jazyku C.

Kľúčové slová:
Dátové štruktúry, triedenie, lineárny zoznam, binárny vyhľadávací strom

ABSTRACT

KOLEK, Bohuslav: Structured programming in C. – University of Economics in Bratislava. Faculty of Business Informatics; Department of Applied Computer Science – Final work chief: Ing. Ján Hanák, PhD. – Bratislava: FHI, 2011, 45p.

The main aim of bachelor´s work was to provide basic information about sorting of a list of elements and about data structures (linear and tree). The work is divided into four chapters. It contains of 4 tables and 14 images. The first chapter is devoted to describing the basic properties of sorting algorithms and data structures. The second and third chapter describes the objective of the work and methodology of the work. In the fourth chapter we describe how the selected sorting algorithms works and we also describe data structures and implement them in C. The results of solving the problems are the algorithms in C.

Keywords:
Data structures, sorting, linear list, binary search tree

O B S A H str.

Úvod **8**

1 Súčasný stav riešenia problematiky **9**

1.1. Triedenie 9
1.2. Dátové štruktúry 11
1.2.1. Lineárne dátové štruktúry 12
1.2.2. Binárne stromy 15

2 Cieľ práce **19**

3 Metodika práce **20**

4 Výsledky práce **21**

4.1. Triedenie (triediace algoritmy) 21
4.1.1. Bubblesort 21
4.1.2. Shakersort (Cocktailsort) 23
4.1.3. Combsort 24
4.1.4. Shellsort 25
4.1.5. Insertion sort 26
4.1.6. Výsledky porovnávania 27
4.2. Lineárne dátové štruktúry 29
4.2.1. Základné operácie vykonávane nad obojsmerným lineárnym zoznamom 30
4.2.2. Vytvorenie zoznamu a vkladanie prvkov na koniec zoznamu 31
4.2.3. Oprava aktuálneho uzla 32
4.2.4. Vymazanie aktuálneho uzla zo zoznamu 32
4.2.5. Pohyb v zozname (zmena aktuálneho uzla) 33
4.2.6. Výpis zoznamu 34
4.2.7. Zotriedenie zoznamu 34
4.2.8. Vymazanie celého zoznamu 35
4.3. Binárne stromy 36
4.3.1. Vytvorenie binárneho vyhľadávacieho stromu 37
4.3.2. Prechod binárnym vyhľadávacím stromom 38

4.3.3. Mazanie vrcholov 39

Záver **44**

Použitá literatúra **45**

Úvod

Programovací jazyk C vznikol na začiatku sedemdesiatych rokov v Bellových laboratóriách firmy AT&T. Autorom tohto jazyka je Denis Ritchie. C patrí medzi štruktúrované programovacie jazyky. To znamená, že v algoritmoch sa používajú riadiace štruktúry, ktoré určujú postup pri riadení jednotlivých krokov algoritmu. Sú to :

1. Postupnosť (sekvencia príkazov)
2. Vetvenie
3. Cyklus

Všetky tieto štruktúry sú popísané v mnohých príručkách o programovaní v jazyku C. Preto sa v tejto práci nebudeme venovať popisu týchto štruktúr, ale použijeme ich pri riešení vybraných problémov. Prvá problematika, ktorej sa budeme venovať je triedenie, pretože táto problematika patrí k základným problémom pri programovaní. Popíšeme si postup usporiadania pri vybraných algoritmoch, implementujeme ich v jazyku C a na záver ich porovnáme. Druhou problematikou, ktorou sa budeme zaoberať sú dátové štruktúry, ktoré spolu s algoritmami tvoria samotné programy. Tak ako pri triedení aj tu si vyberieme niektoré štruktúry a bližšie ich popíšeme a implementujeme vybrané operácie s týmito štruktúrami v jazyku C.

1 Súčasný stav riešenia problematiky

1.1. Triedenie

Triedenie (usporiadanie) množiny prvkov je proces preusporiadania danej množiny v špecifickom poradí (podľa určitej relácie). Je to jedna z najčastejšie sa vyskytujúcich operácií pri spracovaní údajov, resp. dátových štruktúr. Usporiadanie množiny prvkov výrazne zvyšuje efektívnosť vyhľadávania prvkov. Typickými príkladmi triedenia je spracovanie rôznych zoznamov, registrov, skladov, slovníkov a pod.

Problematika triedenia je veľmi prepracovaná. Venovalo sa jej množstvo autorov. Existuje celý rad metód, ktoré sa líšia rýchlosťou, pamäťovou a operačnou zložitosťou algoritmu, princípom triedenia, prípustným typom triedených prvkov a spôsobom prístupu k triedeným prvkom.

V matematike a informatike na riešenie takýchto úloh používame triediace algoritmy. Tieto majú nasledujúce vlastnosti:

- časová zložitosť
- stabilita
- typ triedenia

Časová zložitosť

Je základnou vlastnosťou triediacich algoritmov. V prípade triedenia vyjadruje, aký čas je potrebný na zotriedenie n-prvkového zoznamu. Jedná sa o takzvanú O-notáciu, ktorá predstavuje horné ohraničenie časovej zložitosti. Typické horné ohraničenie pri triediacich algoritmoch je $O(n \log n)$ alebo v horších prípadoch $O(n^2)$. Ideálnym prípadom je $O(n)$ (pri ktorom je potrebné každý prvok prečítať aspoň raz), avšak tento výsledok sa dosahuje iba v ideálnych prípadoch. V priemerných hodnoteniach nie je možný.

Stabilita

Ako uvádza Wirth (2, s.46) triedený zoznam môže obsahovať dáta s rovnakými hodnotami. Podľa toho, či dochádza k zmene vzájomnej polohy týchto prvkov pred a po zoradení, ich delíme na stabilné a nestabilné.

Stabilný algoritmus zachováva vzájomnú polohu. To znamená, že keď zoradím prvky zoznamu najprv podľa mena a následne podľa priezviska, pri použití stabilných algoritmov dostanem zoznam zoradený podľa priezviska a pri rovnakých priezviskách zoradený podľa mena.

Nestabilné triedenie môže zmeniť vzájomnú polohu prvkov.

Napríklad ak potrebujem zotriediť nasledujúce prvky:

(Malý, Peter) (Veľký, Boris) (Malý, Ján) (Dlhý, Karol).

Po zotriedení týchto prvkov podľa priezviska môžeme dostať tieto výsledky:

(Dlhý, Karol) (Malý, Ján) (Malý, Peter) (Veľký, Boris) – poradie zmenené

(Dlhý, Karol) (Malý, Peter) (Malý, Ján) (Veľký, Boris) – poradie zachované.

Pri stabilnom triedení je možné prvky triediť viackrát. Každé triedenie podľa iného kľúča. V tomto prípade je treba aplikovať triedenie podľa kľúčov s rastúcou prioritou:

(Malý, Peter) (Veľký, Boris) (Malý, Ján) (Dlhý, Karol) – originál

(Veľký, Boris) (Malý, Ján) (Dlhý, Karol) (Malý, Peter) – po triedení podľa mena

(Dlhý, Karol) (Malý, Ján) (Malý, Peter) (Veľký, Boris) – po triedení podľa priezviska.

Typ triedenia

Podľa Wirtha (2, s.46) triedenie môžeme rozdeliť do troch hlavných kategórii podľa ich základnej metódy. A to triedenie:

Výberom – selection

V nezotriedenom zozname nájdeme prvok s minimálnou hodnotou a uložíme ho na začiatok zoznamu.

```
6 1 4 5 3 2
1.6 4 5 3 2
1 2.6 4 5 3
1 2 3.6 4 5
1 2 3 4.6 5
1 2 3 4 5.6
1 2 3 4 5 6
```

Vkladaním – insertion

V nezotriedenom zozname postupne berieme jeden prvok za druhým a kladieme ho na správne miesto v zoradenom zozname.

```
6 1 4 5 3 2
6.1 4 5 3 2
1 6.4 5 3 2
1 4 6.5 3 2
1 4 5 6.3 2
1 3 4 5 6.2
1 2 3 4 5 6
```

Zámenou – exchanging

V nezotriedenom zozname na základe metódy konkrétneho algoritmu nájdeme dvojicu prvkov, ktoré sú nezotriedené, a vzájomne ich vymeníme.

```
6 1 4 5 3 2
1 4 5 3 2 6
1 4 3 2 5 6
1 3 2 4 5 6
1 2 3 4 5 6
```

1.2. Dátové štruktúry

Každý program má dve základné časti, ktoré profesor Wirth zhrnul do známej rovnice

Programy = Dátové štruktúry + Algoritmy

Podľa Horovčáka a Podlubného (4) sa zdá, že jedným zo základných princípov moderných technológií vytvárania programov je práve oddelenie časti, zaoberajúcej sa údržbou informácií v dátových štruktúrach od vlastného algoritmu riešenia úlohy .

Obidve časti môžu byť v programe v rôznej miere rozptýlené. Sústredením pamäťových prostriedkov a kódu pre manipuláciu s nimi do samostatných modulov získame prehľadnejšie, čitateľnejšie, modifikovateľnejšie, lepšie dokumentované a predovšetkým spoľahlivejšie programy, ktoré sú aj efektívnejšie, lebo nedochádza napr. k opakovaniu kódu. Dá sa povedať, že vhodným návrhom dátovej štruktúry pre danú úlohu je spravidla aj algoritmus riešenia úlohy jednoduchší a prehľadnejší.

V ďalšej časti sa budeme zaoberať základnými dátovými štruktúrami. A to lineárnymi zoznamami a stromovými štruktúrami.

1.2.1. Lineárne dátové štruktúry

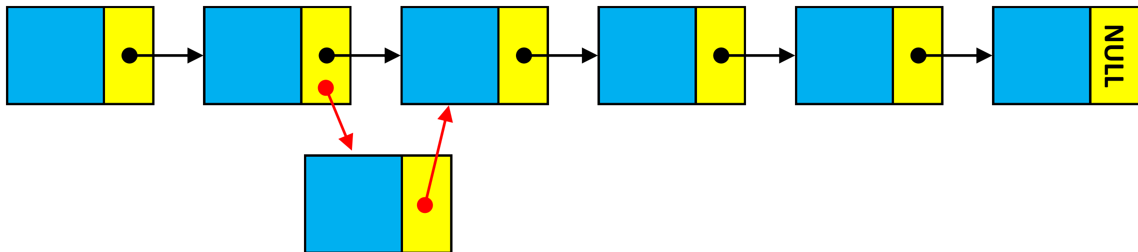
Slovo lineárny udáva, že počet predchodcov aj nasledníkov hociktorej položky zoznamu je maximálne 1. To znamená, že v lineárnom zozname existuje maximálne jedna predchádzajúca a jedna nasledujúca položka. Z toho dôvodu lineárne zoznamy slúžia pre reprezentáciu úplného usporiadania. Podľa spôsobu vkladania a vyberania položiek rozoznávame tri typy lineárnych dátových štruktúr: front, zásobník a zoznam(4).

Front je spôsob organizácie, ktorý používame vtedy, ak potrebujeme spracovať položky v takom poradí, v akom prichádzajú. Spôsob vkladania a vyberania položiek sa označuje ako FIFO (first-in, first-out).

Zásobník je front typu LIFO (last-in first-out) - posledná vložená položka bude vybraná ako prvá. Používame tam, kde potrebujeme odložiť informácie a cestou späť sa k nim vrátiť (stack pri volaní funkcií, riešenie rekurzívnych problémov).

Zoznam predstavuje takú postupnosť, do ktorej môžeme pridávať, resp. rušiť položky na ľubovoľnom mieste, ktoré je označené premennou index (tzv. ukazovateľ, ktorý označuje ľubovoľné miesto práce so zoznamom od začiatku až po koniec). Tým sa zoznam líši od frontu, resp. zásobníka, kde je možná aktualizácia iba na koncoch postupnosti. Zoznam je teda všeobecným prípadom lineárnej dátovej štruktúry. Realizácia zoznamu v programe je možná tak ako pri fronte aj zásobníku dvoma spôsobmi. Použitím poľa alebo dynamických štruktúr. Podrobnejšie si popíšeme dynamické štruktúry.

Ideálne riešenie je údajová štruktúra vytvorená pomocou ukazovateľov. Každý prvok "zreťazeného" zoznamu je uložený v samostatnom objekte (dynamickej premennej), ktorého súčasťou je okrem hodnoty prvku taktiež položka obsahujúca ukazovateľ na objekt, v ktorom je uložený ďalší prvok zoznamu.



Obrázok 1 – Vloženie prvku do lineárneho zoznamu

Na obrázku 1 vidíme zobrazenie operácie vloženia nového prvku. Realizuje sa takto:

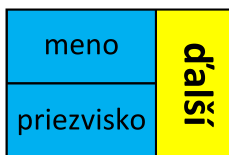
- Vytvoríme objekt, ktorý obsahuje hodnotu nového prvku a ukazovateľ na objekt, pred ktorý sa zaraďuje
- Zmeníme ukazovateľ v tom objekte, za ktorý sa nový prvok zaraďuje

Rozsah takto reprezentovanej dynamickej štruktúry (t.j. v našom prípade počet prvkov zoznamu) je možné v programe ľubovoľne meniť - prvky (dynamické premenné) do zoznamu môžeme podľa momentálnej potreby pridávať a odoberať. Vyhneme sa tak zbytočnému zaťažovaniu pamäte.

Ako vo svojej práci uvádza Stoffová (5), dynamické premenné, ktoré sú navzájom prepojené ukazovateľmi a tvoria tak dynamickú štruktúru sa nazývajú tiež uzly. Jednotlivé uzly nesú vždy istú hodnotu (obsah) a sprístupňujú iné uzly hodnotou príslušných ukazovateľov.

Každý uzol obsahuje preto dve časti:

- **hodnotovú časť**, ktorá vyjadruje vlastnú hodnotu (obsah) uzla (modrá časť v obrázku 2)
- **spojovaciu časť** (väzobnú) tvorenú hodnotou ukazovateľa na iný uzol (resp. hodnotami ukazovateľov na iné uzly). (žltá časť v obrázku 2)



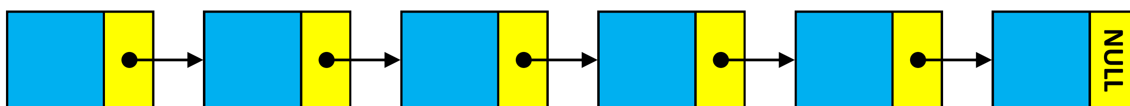
Obrázok 2 – Grafické znázornenie uzla

Preto je vhodným údajovým typom používaným pre definíciu uzla dynamickej štruktúry záznam - v jazyku C nazvaný **štruktúra** - `struct`.

Ako vo svojej práci uvádza Knuth (6, s. 251-295), poznáme 4 typy lineárnych zoznamov.

- **Jednosmerný lineárny zoznam**

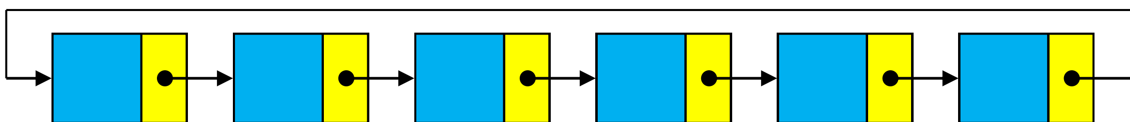
Jednosmerný lineárny zoznam obsahuje prvky (uzly), z ktorých každý má jednu položku ukazovateľ. Tieto ukazovatele sú nastavené tak, aby ukazovali na prvok nasledujúci v zozname. Posledný uzol ,keďže nemá na koho ukazovať má hodnotu NULL.



Obrázok 3 – Grafické znázornenie jednosmerného lineárneho zoznamu

- **Jednosmerný cyklický zoznam**

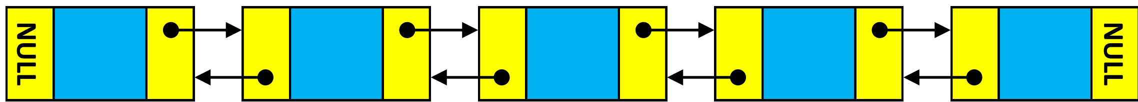
V tomto prípade sa jedná o lineárny zoznam, ktorého uzly tvoria kruh. Ukazovateľ posledného uzla v tomto prípade nemá hodnotu NULL, ale má hodnotu prvého uzla v zozname.



Obrázok 4 – Grafické znázornenie jednosmerného cyklického zoznamu

- **Obojsmerný lineárny zoznam**

Každý uzol obsahuje dva ukazovatele. Na predchádzajúci aj nasledujúci uzol. Samozrejme prvý uzol nemá predchodcu, takže hodnota ukazovateľa bude NULL a posledný, keďže nemá nasledovníka ma tiež v príslušnom ukazovateli hodnotu NULL.



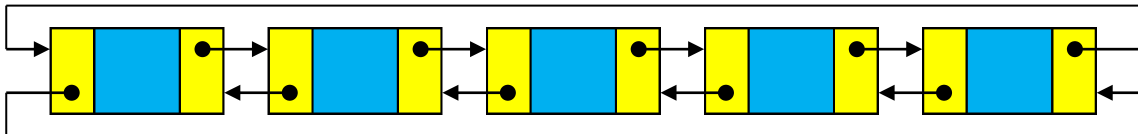
Obrázok 5 – Grafické znázornenie obojsmerného lineárneho zoznamu

- **Obojsmerný cyklický zoznam**

Ako v predchádzajúcom type, ale namiesto hodnôt NULL v prvom a poslednom uzle sú vložené nasledujúce:

- Pri prvom uzle adresa posledného v ukazovateľovi na predchádzajúci uzol
- Pri poslednom uzle adresa prvého uzla v ukazovateľovi na nasledujúci uzol

Takto sa vlastne obojsmerne uzavrie.

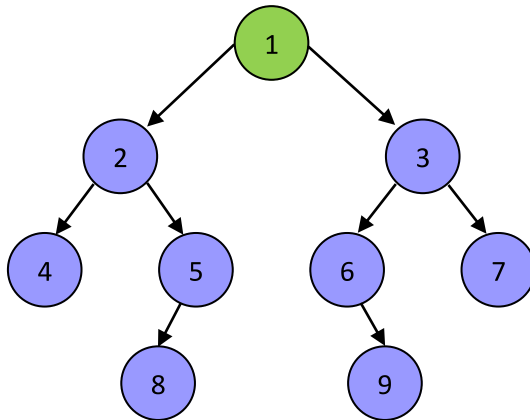


Obrázok 6 – Grafické znázornenie obojsmerného cyklického zoznamu

1.2.2. Binárne stromy

Vo všeobecnosti sú pre programátora stromy ďalšou dynamickou údajovou štruktúrou. So stromami sa stretávame pomerne často. Uvedieme niekoľko príkladov: strom adresárov na disku počítača, rozhodovací strom (od jednoduchého binárneho vetvenia až po strom všetkých možností), strom odvodenia a pod. Pomocou stromu vždy vyjadrujeme určitú hierarchickú štruktúru, vzájomné vzťahy objektov. Dobrým príkladom zo života môže byť strom, v ktorom sa nachádzajú všetci zamestnanci firmy - v koreni je generálny riaditeľ, nižšiu úroveň tvoria vedúci oddelení, každému z nich podliehajú ďalší pracovníci atď.

V teórii grafov stromom rozumieme súvislý orientovaný graf, v ktorom neexistujú cykly. Ako vidíme na obrázku 7, strom obsahuje množinu prvkov (vrcholov) s jedným špeciálnym prvkom, ktorý nazývame **koreň stromu**. Je to jediný vrchol, do ktorého nevstupuje žiadna hrana. Do všetkých ostatných vstupuje práve jedna hrana. Pre jednotlivé vrcholy sa používa rodinná terminológia:



Obrázok 7 – Grafické znázornenie stromu

- Ak z vrcholu v idú hrany do vrcholov w1 a w2, tak hovoríme, že w1 a w2 sú bratia a sú synmi vrcholu v (je ich otec)
- Ak existuje cesta z vrcholu v do vrcholu w, tak vrchol v nazývame predkom vrcholu w a vrchol w nazývame potomkom vrcholu v
- Každý vrchol so všetkými svojimi potomkami tvorí podstrom (s koreňom v)
- Vrcholy, ktoré majú synov, sa nazývajú vnútorné, vrcholy bez synov sú listy

V "našom" strome sú napr. vrcholy 4, 5 bratia, a zároveň sú synmi vrcholu 2. Vrchol 8 je potomkom vrcholu 2, koreň stromu - vrchol 1 je predkom všetkých ostatných vrcholov. Vrcholy 4, 8, 9, 7 sú listy.

Podľa charakteru jednotlivých problémových aplikácií možno špecifikovať rôzne špeciálne prípady stromu.

Usporiadaný strom je taký, v ktorom synovia každého vrcholu tvoria usporiadanú množinu (je medzi nimi stanovené poradie).

Počet priamych potomkov vnútorného vrcholu nazývame jeho stupňom. Maximálny stupeň spomedzi všetkých vrcholov určuje stupeň stromu.

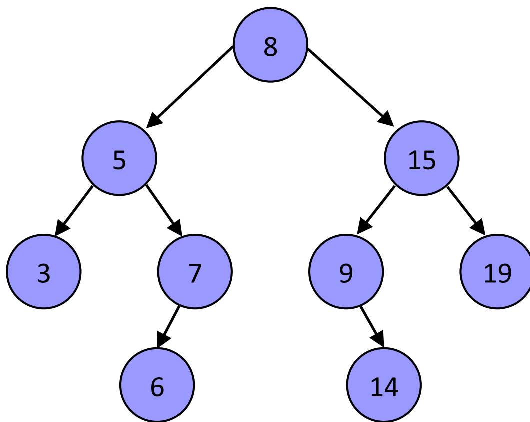
Viaccestný strom je strom so stupňom vyšším ako 2.

Binárny strom (binary tree), je taký strom, v ktorom má každý vrchol najviac dvoch synov, pričom je určené, ktorý syn je ľavý a ktorý pravý. Binárny strom je teda usporiadaný strom druhého stupňa. My sa budeme v ďalšej časti zaoberať binárnym stromom.

Špeciálnym a podľa Stoffovej (5) aj veľmi užitočným prípadom binárneho stromu je takzvaný binárny vyhľadávací strom. Vrcholy v tomto strome sú ohodnotené hodnotou h(V) pričom platí:

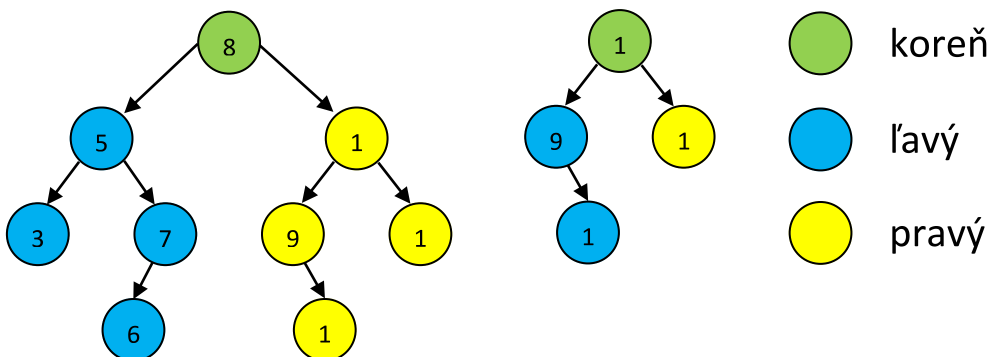
1. $h(U)<h(V)$ pre každý vrchol U z ľavého podstromu stromu s koreňom V
2. $h(U)>h(V)$ pre každý vrchol U z pravého podstromu stromu s koreňom V
3. pre každý m patriace M existuje práve jeden vrchol V taký, že $h(V)=m$

Inými slovami, pre všetky vrcholy platí, že v ľavom synovi je uložená hodnota menšia a v pravom väčšia ako v danom vrchole. Z toho vyplýva, že všetky hodnoty v ľavom podstrome sú menšie a v pravom podstrome väčšie ako v koreni stromu.



Obrázok 8 – Grafické zobrazenie binárneho vyhľadávacieho stromu

Ďalšou vlastnosťou binárneho stromu je, že sa naňho môžeme pozerať ako na rekurzívnu štruktúru. To znamená, že ho môžeme definovať pomocou seba samého. Každý strom je tvorený ďalšími stromami. Toto môžeme vidieť na obrázku 9:



Obrázok 9 – Grafické znázornenie rekurzie v strome

Tu je vidieť, že každý strom sa skladá z ľavého a pravého podstromu. Ale aj tieto podstromy sa dajú interpretovať, ako strom s koreňom a pravým aj ľavým podstromom. Tak ako to vidíme na obrázku. Takto môžeme strom rozobrať až po listy, ktoré sú v postate taktiež stromy, ale zložené len s koreňa s prázdnymi podstromami. Túto vlastnosť využijeme aj pri tvorbe algoritmu na vytvorenie binárneho stromu.

2 Cieľ práce

Cieľom tejto práce je poskytnúť základné informácie o problematike triedenia a dátových štruktúrach. Tieto problematiky sú dosť obsiahle, takže sme si z nich vybrali iba určité špecifické časti, ktoré rozpracujeme podrobnejšie. Keďže témou práce je štruktúrované programovanie v jazyku C, následne implementujeme riešenie pre jednotlivé problémy pomocou algoritmov v jazyku C, ktoré využívajú princípy a prostriedky štruktúrovaného programovania.

V časti venovanej triedeniu je naším ďalším cieľom porovnať jednotlivé algoritmy podľa ich efektivity v rôznych situáciách.

3 Metodika práce

Pri zostavovaní tejto práce sme použili viacero krokov:

- v prípravnej fáze štúdium odbornej literatúry a dostupných elektronických zdrojov, ktoré sú uvedené v zozname použitej literatúry
- výber problematiky, ktorej sa v ďalších kapitolách venujeme
- príprava teoretického základu, ktorý slúži k pochopeniu problému
- implementácia riešenia v jazyku C
- popis jednotlivých algoritmov

Pri časti venovanej triedeniu zoznamu prvkov sme do jednotlivých algoritmov zakomponovali aj pomocné premenné, ktoré nám pomôžu pri porovnaní týchto algoritmov z hľadiska rýchlosti a počtu krokov potrebných k riešeniu úlohy. Na porovnanie sme použili viacero úloh, ktoré nám ukážu výhody aj nevýhody jednotlivých algoritmov.

Pri dátových štruktúrach sme si vybrali dva typy týchto štruktúr a to konkrétne obojsmerný lineárny zoznam a binárny vyhľadávací strom. Pre tieto štruktúry sme v jazyku C implementovali základné operácie, ktoré s nimi môžeme vykonávať.

4 Výsledky práce

4.1. Triedenie (triediace algoritmy)

Jedným zo základných problémov, ktorými sa zaoberá informatika je triedenie (usporiadanie) zoznamu prvkov. Tento najčastejšie zoraďujeme, buď abecedne, alebo podľa numerickej veľkosti čísel

V tejto časti práce si popíšeme princíp práce rôznych algoritmov triedenia. Tiež si ich popíšeme vlastnosťami algoritmov a implementujeme daný algoritmus v jazyku C. Do týchto algoritmov vložíme aj dve pomocné premenné, ktoré nám pomôžu pri vzájomnom porovnaní efektívnosti algoritmov a to:

`poc_porovnani` – v tejto premennej budeme počítať počet porovnaní, ktoré vykoná algoritmus pri triedení zoznamu
`poc_vymen` – v tejto premennej budeme počítať počet výmien prvkov v algoritme

4.1.1. Bubblesort

Bublinové triedenie patrí k najjednoduchším spôsobom triedenia, aj keď jeho efektivita je veľmi nízka. Je to klasický výučbový algoritmus. Princípom tohto triedenia je vzájomné porovnávanie susedných prvkov v zozname. Začíname porovnávaním prvých dvoch. Ak je prvý prvok väčší ako druhý, tak ich vymeníme. Takto postupujeme celým zoznamom. Po prvom prechode celého zoznamu sa najväčší prvok dostáva na jeho koniec. Tento postup sa opakujeme na celý zoznam, pokiaľ počas jeho prechodu dochádza k výmene prvkov. Ak nedôjde k žiadnej výmene zoznam je zotriedený. Problémom pri tomto triedení je pohyb prvkov v zozname. Sú to takzvané korytnačky a zajace. Zajace sú prvky v vysokými hodnotami, ktoré sa pomerne rýchlo presúvajú na koniec zoznamu (najväčší prvok sa ocitá na svojom mieste po prvom prechode zoznamom, aj keď bol na začiatku na prvom mieste). Opačné sú prvky s malými hodnotami. Tie sa smerom dopredu posúvajú veľmi pomaly. Napríklad najmenší prvok sa v každom prechode zoznamom posunie iba o jednu pozíciu. Ak je úplne na konci potrebujeme *n-1* prechodov zoznamom, aby sa dostal na začiatok.

Typ triedenia: zámenou

Stabilita: stabilný

Časová zložitosť

Priemerná aj najhoršia časová zložitosť bublinového triedenia je $O(n^2)$. Tento algoritmus je jedným z najpomalších, aj v porovnaní s algoritmami s rovnakou časovou zložitosťou. Je nevýhodný pre použitie na triedenie rozsiahlych nezotriedených zoznamov. Lepšie výsledky však dosahuje pri zoradení zoznamu, kde je iba jeden prvok, ktorý nie je na svojom mieste. V tomto prípade je zložitosť $2n$ a v prípade dvoch nezotriedených prvkov je to $3n$.

Implementácia

```
 1.  do
 2.  {
 3.    vymena = NIE;
 4.    for(x = 0; x < velkost_pola-1; x++)
 5.    {
 6.      poc_porovnani += 1;
 7.      if(pole[x] > pole[x+1])
 8.      {
 9.        vymen(&pole[x], &pole[x+1]);
10.        poc_vymen += 1;
11.        vymena = ANO;
12.      }
13.    }
14.  }while (vymena);
```

Pomocou malých úprav sa dá rýchlosť tohto algoritmu upraviť. Stačí ak využijeme to, že pri každom prechode sa najväčší prvok dostáva na koniec zoznamu a budeme prechádzaný zoznam pri každom prechode skracovať o jeden prvok. Tým sa výrazne zmenší počet porovnaní, ktoré sú potrebné k zotriedeniu zoznamu. Aj keď počet výmen zostane zachovaný.

```
 1.  do
 2.  {
 3.    vymena = NIE;
 4.    for(x = 0; x < velkost_pola-1-y; x++)
 5.    {
 6.      poc_porovnani += 1;
 7.      if(pole[x] > pole[x+1])
 8.      {
 9.        vymen(&pole[x], &pole[x+1]);
10.        poc_vymen += 1;
11.        vymena = ANO;
12.      }
13.    }
14.    y+=1;
15.  }while (vymena);
```

4.1.2. *Shakersort (Cocktailsort)*

Toto triedenie vychádza s bublinového triedenia. V podstate je to obojsmerné bublinové triedenie. Implementačne je náročnejšie. Snažíme sa odstrániť problém „korytnačiek“ v triedenom zozname a to tým, že porovnávame prvky v oboch smeroch. Čiže po prvom prechode zoznamom, keď sa najväčší prvok dostáva na svoju pozíciu nepokračujeme v porovnávaní od začiatku, ale sa vraciame v zozname smerom na začiatok. V prvom kroku spätného chodu porovnávame prvok *n-1* s prvkom *n-2*. Potom *n-2* s *n-3* a takto postupujeme až na začiatok zoznamu. Po tomto prechode sa na prvú pozíciu v zozname dostáva najmenší prvok. Tým eliminujeme “korytnačku“. V ďalšom prechode porovnávaný zoznam skracujeme o dva prvky (prvý a posledný, ktoré sú už na svojich miestach). Týmto sme zmenšili počet výmen aj počet porovnaní.

Typ triedenia: zámenou

Stabilita: stabilný

Časová zložitosť

Najhoršia aj priemerná zložitosť tohto triedenia je $O(n^2)$. Ale v prípade ak je zoznam takmer zotriedený tak sa blíži k $O(n)$. Napríklad ak všetky prvky sú v polohe, ktorej vzdialenosť od zotriedeného stavu je k (k≥1), tak časová zložitosť tohto zoznamu je $O(k * n)$.

Implementácia

```
 1.  for(x = 0; x < velkost_pola/2; x++)
 2.  {
 3.    vymena = NIE;
 4.    for(y = (0+x); y < velkost_pola-1-x; y++)
 5.    {
 6.      poc_porovnani += 1;
 7.      if(pole[y] > pole[y+1])
 8.      {
 9.        vymen(&pole[y], &pole[y+1]);
10.        poc_vymen += 1;
11.        vymena = ANO;
12.      }
13.    }
14.    for(y = velkost_pola-2-x; y > (0+x); y--)
15.    {
16.      poc_porovnani += 1;
17.      if(pole[y] < pole[y-1])
18.      {
19.        vymen(&pole[y], &pole[y-1]);
20.        poc_vymen += 1;
```

```
21.         vymena = ANO;
22.       }
23.     }
24.     if (!vymena) break;
25.  }
```

4.1.3. Combsort

Tento algoritmus bol vynájdený v roku 1980 Wlodzimierzom Dobosiewiczom. Neskôr ho znovuobjavili a spopularizovali Steven Lacey a Richard Box v článku uverejnenom Byte Magazine v apríli 1991. Combsort upravuje bublinové triedenie. Základnou myšlienkou je odstránenie „korytnačiek“, ktoré spomaľujú toto triedenie. Pri porovnávaní prvkov v zozname porovnávame prvky, ktoré sú vedľa seba, čiže ich rozstup je 1. Ideou combsortu je porovnávať prvky, ktorých rozstup je väčší. Na začiatku prvky zotriedime a to tak, že dĺžku zoznamu delíme tzv. shrink faktorom. Najčastejšie ide o hodnotu 1,3. Výsledok zaokrúhlime smerom nadol na celé číslo (aby sme vedeli indexovať prvky zoznamu uložené v poli). Toto číslo nám udáva krok pre prvé porovnávanie. Po prechode zoznamom s týmto krokom, krok opäť delíme shrink faktorom. Takto sa to opakuje až kým sa krok nerovná 1. Ďalej pokračujeme ako pri bublinovom triedení. V pôvodnej verzii sa ako shrink faktor požíva hodnota 1,3. Ale po testovaní viacerých hodnôt sa ako najefektívnejšia ukázala hodnota 1,247330950103979.

Typ triedenia: zámenou

Stabilita: nestabilný

Časová zložitosť

Najhoršia zložitosť tohto triedenia je $O(n^2)$.

Implementácia

```
1.  while ((krok > 1) || vymena)
2.  {
3.    if (krok > 1) krok = (int) (krok / shrink_faktor);
4.    vymena = NIE;
5.    x = 0;
6.    while ((krok + x) < velkost_pola)
7.    {
8.      poc_porovnani += 1;
9.      if (pole[x] > pole[x + krok])
10.     {
11.       vymen(&pole[x], &pole[x + krok]);
```

```
12.         poc_vymen += 1;
13.         vymena = ANO;
14.       }
15.       x++;
16.     }
17.   }
```

4.1.4. Shellsort

Shellovo triedenie bolo vytvorené Donaldom Shellom v roku 1959. Je to v podstate zlepšená verzia triedenia priamym vkladaním. Toto pozostáva v triedení prvkov po skupinách so zmenšovaním kroku, pričom sa začíname s najväčším krokom a končíme krokom 1. Podobne ako v Comb sorte. Obvykle ako postupnosť krokov uvažujeme rad mocnín 2, napr.: 4,2,1, ale nie je to podmienkou. Voľbou postupnosti krokov však môžeme podstatne ovplyvniť efektívnosť triedenia. Pri rade mocnín napríklad nedochádza k vzájomnému porovnávaniu prvkov na párnych a nepárnych miestach. Tie porovnávame až v poslednom kroku, keď sa krok rovná jednej.

Typ triedenia: vkladaním

Stabilita: nestabilný

Časová zložitosť

Pri použití Shellovho pôvodného kroku, teda začínajúceho hodnotou n/2 a postupného delenia na polovicu, pokiaľ nedosiahne hodnotu 1 dosahujeme najhoršiu zložitosť tohto triedenia $O(n^2)$. Pri Hibbardovej sekvencii 2^k-1 je zložitosť $O(n^{4/3})$. Najlepšie výsledky sa podarilo dosiahnuť pri použití sekvencie nájdenej Marcinom Ciurom. Táto je 1, 2, 10, 23, 57, 132, 301, 701, 1750. V tomto prípade ide o zložitosť $O(n \log n)$.

Implementácia

Podľa shella

```
1.  krok = velkost_pola / 2;
2.  while (krok > 0)
3.  {
4.    for (x = krok; x < velkost_pola; x++)
5.    {
6.      y = x;
7.      while ((y >= krok) && (pole[y-krok] > pole[y]))
8.      {
9.        vymen(&pole[y], &pole[y-krok]);
```

```
10.         poc_vymen += 1;
11.         poc_porovnani += 1;
12.         y-=krok;
13.       }
14.       poc_porovnani += 1;
15.     }
16.     if (krok == 2)
17.       krok = 1;
18.     else
19.       krok = (int) (krok / 2);
20.   }
```

S využitím postupnosti kroku od Martina Ciuru

```
1.  int kroky[14] = { 171228, 68491, 27396, 10958, 4383, 1753, 701,
    301, 132, 57, 23, 10, 4, 1 };
2.  for ( z = 0; z < 14; z++)
3.  {
4.    krok = kroky[z];
5.    for (x = krok; x < velkost_pola; x++)
6.    {
7.      y = x;
8.      while ((y >= krok) && (pole[y-krok] > pole[y]))
9.      {
10.       vymen(&pole[y], &pole[y-krok]);
11.       poc_vymen += 1;
12.       poc_porovnani += 1;
13.       y-=krok;
14.     }
15.     poc_porovnani += 1;
16.   }
17. }
```

4.1.5. Insertionsort

Insertionsort alebo triedenie priamym vkladaním. V každom kroku tohto algoritmu vyberáme prvok z neutriedenej časti zoznamu a ukladáme ho na správnu pozíciu do zotriedenej časti zoznamu. Začíname prvými dvoma prvkami, ktoré porovnávame a dávame do správneho poradia. V ľavej časti zoznamu sa takto vytvára zotriedená časť zoznamu. V ďalšom kroku porovnávame tretí prvok (ten je prvý prvok v nezotriedenej časti zoznamu) postupne, najprv s druhým prvkom. Ak je menší vymeníme ich, ak nie (to znamená, že poradie prvkov je správne) pokračujeme ďalším krokom, čiže berieme ďalší prvok z nezotriedenej časti. Pokiaľ sme prvky vymenili, potom pokračujeme v porovnávaním s prvým prvkom. Celý tento cyklus sa opakuje *n-1* krát. Takto zotriedime celý zoznam.

Typ triedenia: vkladaním

Stabilita: stabilný

Časová zložitosť

Najlepší prípad nastáva, keď je pole už zotriedené. Vtedy je výsledok lineárny $O(n)$ (porovnávame len posledný prvok zo zoradenej časti s prvým prvkom nezotriedenej). Najhorší prípad je ak je zoznam zotriedený v opačnom poradí. Vtedy je výsledok kvadratický $O(n^2)$. Priemerná hodnota je takisto $O(n^2)$.

Implementácia

```
 1.  for (x = 1; x < velkost_pola; x++)
 2.  {
 3.    y = x;
 4.    while ((y-1 >= 0) && (pole[y-1] > pole[y]))
 5.    {
 6.      vymen(&pole[y], &pole[y-1]);
 7.      poc_vymen += 1;
 8.      poc_porovnani += 1;
 9.      y-=1;
10.    }
11.    poc_porovnani += 1;
12.  }
```

4.1.6. Výsledky porovnávania

Pre porovnanie jednotlivých algoritmov z hľadiska počtu porovnaní, výmen a rýchlosti sme triedili pole so 100 000 prvkami. V prvom prípade to boli hodnoty z rozsahu 0 až 100. Čiže sme triedili veľké množstvo prvkov s rovnakými hodnotami. To nám znižuje počet výmen v porovnaní s nasledujúcim pokusom. Keďže prvky v rovnakou hodnotou sa nevymieňajú. Výsledky tohto triedenia vidíme v tabuľke 1. Pri bubblesort triedení máme dva algoritmy. BUBBLESORT (viď algoritmus na strane 18) a BUBBLESORT2 (viď upravený algoritmus na strane 18). Podobne aj pri triedení shellsorte. SHELLSORT (originálny algoritmus podľa Shella) a SHELLSORT2 (s využitím postupnosti kroku od Martina Ciuru). Testovanie sme robili na notebooku TOSHIBA Satellite A665-148, procesor Intel Core i7 1,73GHz, 4GB RAM, OS Windows 7 64-bit.

Tabuľka 1 - výsledky triedenia poľa o veľkosti 100 000 prvkov (hodnoty prvkov 0 - 100)

100 000 prvkov	Počet porovnaní	Počet výmen	Čas (s)
BUBBLESORT	9 884 801 151	2 473 081 212	153.5640
BUBBLESORT2	4 999 288 175	2 473 081 212	132.3410
SHAKERSORT	3 753 572 372	2 473 081 212	117.9600
COMBSORT	4 395 790	**291 109**	**0.0400**
INSERTIONSORT	2 473 181 211	2 473 081 212	104.8810
SHELLSORT	2 579 501	1 079 495	0.0670
SHELLSORT2	**1 794 289**	608 499	**0.0400**

V tomto prípade je jednoznačne najhorší algoritmus BUBBLESORT, ktorý potrebuje najviac porovnaní, výmen aj času. Najefektívnejšie sa javia COMBSORT a SHELLSORT2, ktoré sú časovo približne rovnaké. Pričom na počet porovnaní je na tom lepšie SHELLSORT2, ale na počet výmen COMBSORT.

V druhom triedení sme použili hodnoty prvkov od 0 po 1 000 000. Výsledky tohto triedenia vidíme v tabuľke 2.

Tabuľka 2 - výsledky triedenia poľa o veľkosti 100 000 prvkov (hodnoty 0 - 1 000 000)

100 000 prvkov	Počet porovnaní	Počet výmen	Čas (s)
BUBBLESORT	9 970 400 295	2 504 098 947	168.6490
BUBBLESORT2	4 999 906 635	2 504 098 947	129.7760
SHAKERSORT	3 757 264 269	2 504 098 947	115.8670
COMBSORT	4 495 789	**712 105**	**0.0630**
INSERTIONSORT	2 504 198 946	2 504 098 947	104.5310
SHELLSORT	4 196 541	2 696 535	0.1450
SHELLSORT2	**2 538 418**	1 352 628	0.0770

V porovnaní s prvým meraním sa vo všetkých algoritmoch zvýšil počet výmen aj porovnaní. To súvisí s väčším oborom hodnôt triedeného zoznamu. Čiže máme menej prvkov s rovnakou hodnotou, čo nám zvyšuje počet výmen a tiež sa zoznam neskôr stáva zotriedeným, čo zvyšuje počet porovnaní. Porovnanie jednotlivých algoritmov je také isté ako v prvom prípade, akurát s vyšším počtom výmen a porovnaní sa prejavil časový rozdiel medzi COMBSORT a SHELLSORT2 algoritmom. Takže v tomto prípade je combsort efektívnejší.

V treťom prípade sme triedenie spustili na zotriedenom zozname. V tomto prípade nedochádza k výmene prvkov, lebo pole je už zotriedené. Zaujímavý údaj je však počet porovnaní. Ako vidíme v tabuľke 3, oba typy bubblesortu, a insertionsort

potrebujú iba jeden prechod zoznamom, aby zistil, že je zotriedený. Combsort vykoná takmer celý algoritmus triedenia a zastaví sa, až keď je krok rovný 1 a prebehne prvé kolo bubblesortu. Výsledky tohto triedenia vidíme v tabuľke 3.

Tabuľka 3 - výsledky triedenia zotriedeného poľa o veľkosti 100 000 prvkov

100 000 prvkov	Počet porovnaní	Počet výmen	Čas (s)
BUBBLESORT	**99 999**	0	0.0010
BUBBLESORT2	**99 999**	0	0.0010
SHAKERSORT	199 997	0	0.0010
COMBSORT	4 295 791	0	0.0280
INSERTIONSORT	**99 999**	0	0.0010
SHELLSORT	1 500 006	0	0.0130
SHELLSORT2	1 185 790	0	0.0100

V poslednom prípade sme triedili opačne zotriedené pole. Výsledky tohto triedenia vidíme v poslednej tabuľke 4.

Tabuľka 4 - výsledky triedenia opačne zotriedeného poľa o veľkosti 100 000 prvkov

100 000 prvkov	Počet porovnaní	Počet výmen	Čas (s)
BUBBLESORT	9 999 900 000	4 999 950 000	230.2390
BUBBLESORT2	4 999 950 000	4 999 950 000	252.9790
SHAKERSORT	4 999 950 000	4 999 950 000	253.6420
COMBSORT	4 395 790	**232 796**	0.0430
INSERTIONSORT	5 000 049 999	4 999 950 000	252.4630
SHELLSORT	2 344 566	844 560	0.0520
SHELLSORT2	**1 699 006**	513 216	**0.0340**

V tomto najťažšom prípade si COMBSORT a SHELLSORT2 vymenili poradie. Aj keď ich vzájomné poradie v počte porovnaní a výmen zostalo zachované, v časovej efektívnosti v tomto prípade vyšiel lepšie algoritmus SHELLSORT2.

4.2. Lineárne dátové štruktúry

Na implementáciu týchto dátových štruktúr v jazyku C sa využíva dátový typ `struct`.

Podrobnejšie sa budeme venovať obojsmernému lineárnemu zoznamu.

Definujeme nový dátový typ:

```
typedef struct uzol {
      int cislo;
      struct uzol *nasl;
      struct uzol *pred;
} UZOL;
```

Hodnotová časť uzla je v našom príklade celočíselná hodnota typu integer. Tá nám predstavuje, samotné údaje, uložené v zozname. Ďalšie dve položky tvoria spojovaciu časť a sú to ukazovatele na štruktúru UZOL. Konkrétne *nasl na nasledujúci a *pred na predchádzajúci uzol v zozname. V prípade ak by sme chceli uschovať v zozname údaje o mene, priezvisku a veku osoby. Tak by sme premennú číslo nahradili napríklad nasledovne:

```
typedef struct osoba {
      char meno[15];
      char priezvisko[20];
      int vek;
      struct osoba *nasl;
      struct osoba *pred;
} OSOBA;
```

4.2.1. Základné operácie vykonávane nad obojsmerným lineárnym zoznamom

Pre implementáciu základných operácii nad zoznamom použijeme štruktúru UZOL, ktorú sme si zadefinovali vyššie. Plus pomocnú štruktúru, kde si budeme odkladať informácie potrebné pre jednotlivé operácie.

```
typedef struct pom {
      UZOL *akt;
      UZOL *pomocny;
      UZOL *prvy;
      UZOL *posledny;
      int pocet_uzlov;
} POM;
```

*akt – ukazovateľ na aktuálny uzol v zozname

*pomocný – pomocný ukazovateľ

*prvy, *posledny – ukazovateľ na prvý a posledný uzol zoznamu

pocet_uzlov – počet uzlov v zozname

Na začiatku si ju inicializujeme s nasledujúcimi hodnotami:

```
POM pom={NULL,NULL,NULL,NULL,0};
```

4.2.2. Vytvorenie zoznamu a vkladanie prvkov na koniec zoznamu

Realizujeme pomocou podprogramu `vytvor`.

```
 1.  void vytvor(POM *p)
 2.  {
 3.    UZOL *novy;
 4.    int x;
 5.    if ((novy = (UZOL *)malloc(sizeof(UZOL))) == NULL)
 6.    {
 7.      printf("Malo pameti\n");
 8.    };
 9.    if (p->prvy == NULL)
10.    {
11.      p->prvy = novy;
12.      p->posledny = novy;
13.      novy->nasl = NULL;
14.      novy->pred = NULL;
15.    }
16.    else
17.    {
18.      novy->nasl = NULL;
19.      novy->pred = p->posledny;
20.      p->posledny->nasl = novy;
21.      p->posledny = novy;
22.    };
23.    system("cls");
24.    printf ("Zadaj hodnotu uzla a stlac Enter\n");
25.    scanf("%d",&x);
26.    novy->cislo = x;
27.    p->akt = novy;
28.    p->pocet_uzlov++;
29.  }
```

5.-8. - alokácia pamäte pre nový uzol + test či prebehla úspešne.

9. - test či sa jedná o prvý uzol.

11.-14. - prvý uzol nemá predchodcu ani nasledovníka, takže vkladáme NULL + nastavenie pomocných údajov.

16.-22. - ak sa nejedná o prvý uzol, tak vkladáme adresy predchádzajúceho a nasledujúceho uzla a aktualizujeme pomocné údaje.

24.-26. – načítanie a uloženie hodnoty do uzla.

27.-28. – aktualizácia pomocných údajov.

4.2.3. Oprava aktuálneho uzla

```
 1.  void uprav(POM *p)
 2.  {
 3.  int x;

 4.  system("cls");
 5.  printf ("Zadaj hodnotu uzla a stlac Enter\n");
 6.  scanf("%d",&x);
 7.  p->akt->cislo = x;
 8.  }
```

5.-6. – načítanie a uloženie novej hodnoty do aktuálneho uzla.

4.2.4. Vymazanie aktuálneho uzla zo zoznamu

```
 1.  void vymaz(POM *p)
 2.  {
 3.    if (p->akt == p->prvy)
 4.    {
 5.      p->pomocny = p->akt->nasl;
 6.      if ( p->pomocny != NULL ) p->pomocny->pred = NULL;
 7.      free((void*)p->akt);
 8.      p->prvy = p->pomocny;
 9.    }
10.    else if (p->akt == p->posledny)
11.    {
12.      p->pomocny = p->akt->pred;
13.      p->pomocny->nasl = NULL;
14.      free((void*)p->akt);
15.      p->posledny = p->pomocny;
16.    }
17.    else
18.    {
19.      p->pomocny = p->akt->pred;
20.      p->pomocny->nasl = p->akt->nasl;
21.      p->pomocny = p->akt->nasl;
22.      p->pomocny->pred = p->akt->pred;
23.      free((void*)p->akt);
24.    }
25.    p->akt = p->pomocny;
26.    p->pocet_uzlov--;
27.  }
```

3. - test či je aktuálny uzol prvý.

5.-8. - nastavenie pomocného uzla na nasledujúci, ak je pomocný rôzny od NULL t.j. zoznam má viac ako 1 uzol nahradí sa ukazovateľ na prvý uzol hodnotou NULL (pomocný sa stane novým prvým uzlom). Následne sa uvoľní pamäť s aktuálnym uzlom (vymaže sa z pamäte) a aktualizuje sa ukazovateľ na prvý uzol.

10. – test či je aktuálny uzol posledný.

12.-15. - nastavenie pomocného uzla na predchádzajúci. Keďže sa po vymazaní stane posledným, tak do ukazovateľa na nasledujúci vkladáme NULL. Následne sa uvoľní pamäť s aktuálnym uzlom (vymaže sa z pamäte) a aktualizuje sa ukazovateľ na posledný uzol.

19.-23. – vymazanie uzla, ktorý je niekde vo vnútri zoznamu. Nastavenie pomocného na predchádzajúci uzol. Potom nastavenie jeho ukazovateľa na nasledujúci uzol za aktuálnym uzlom. Nastavenie pomocného na nasledujúci uzol. A nastavenie jeho ukazovateľa na predchádzajúci uzol na predchádzajúci aktuálneho uzla. Nakoniec sa vymaže aktuálny.

25.-26. - aktualizácia pomocných údajov.

4.2.5. Pohyb v zozname (zmena aktuálneho uzla)

```
 1.  void predchadzajuci(POM *p)
 2.  {
 3.    system("cls");
 4.    if (p->akt->pred != NULL) p->akt = p->akt->pred;
 5.    else
 6.    {
 7.      printf ("Si na zaciatku zoznamu");
 8.      printf("\nStlac lubovolnu klavesu");
 9.      getchar();
10.    }
11.  }
```

```
 1.  void nasledujuci(POM *p)
 2.  {
 3.    system("cls");
 4.    if (p->akt->nasl != NULL) p->akt = p->akt->nasl;
 5.    else
 6.    {
 7.      printf ("Si na konci zoznamu");
 8.      printf("\nStlac lubovolnu klavesu");
 9.      getchar();
10.    }
11.  }
```

4. – test či nie sme na začiatku (na konci) zoznamu. Ak nie, do aktuálneho uzla sa vloží predchádzajúci (nasledujúci) uzol.

5.-11. – ak sme na začiatku (konci) vypíše sa upozornenie.

4.2.6. Výpis zoznamu

Od začiatku

```
1.  void vypis (POM *p)
2.  {
3.    system("cls");
4.    for (p->pomocny = p->prvy; p->pomocny != NULL; p->pomocny =
    p->pomocny->nasl ) printf ("%d ", p->pomocny->cislo);
5.    printf("\nStlac lubovolnu klavesu");
6.    getchar();
7.  }
```

Od konca

```
1.  void vypis_odzadu (POM *p)
2.  {
3.    system("cls");
4.    for (p->pomocny = p->posledny; p->pomocny != NULL; p->pomocny
    = p->pomocny->pred ) printf ("%d ", p->pomocny->cislo);
5.    printf("\nStlac lubovolnu klavesu");
6.    getchar();
7.  }
```

5 – cyklus, ktorý prechádza zoznam od začiatku (od konca) a vypisuje hodnoty uzlov na obrazovku.

4.2.7. Zotriedenie zoznamu

Na triedenie zoznamu sme použili triedenie Shakersort.

```
1.  void zotried(POM *p)
2.  {
3.    int x,y,z,vymena;

4.    p->pomocny = p->prvy;
5.    x = p->pocet_uzlov-1;

6.    do
7.    {
```

```
 8.      vymena = NIE;

 9.      for (y=1;y<=x;y++)
10.      {
11.        p->akt = p->pomocny;
12.        p->pomocny = p->akt->nasl;
13.        if (p->akt->cislo > p->pomocny->cislo)
14.        {
15.          z = p->akt->cislo;
16.          p->akt->cislo = p->pomocny->cislo;
17.          p->pomocny->cislo = z;
18.          vymena = ANO;
19.        }
20.      }
21.      x--;
22.      p->pomocny = p->akt;
23.      for (y=1;y<=x;y++)
24.      {
25.        p->akt = p->pomocny;
26.        p->pomocny = p->akt->pred;
27.        if (p->akt->cislo < p->pomocny->cislo)
28.        {
29.          z = p->akt->cislo;
30.          p->akt->cislo = p->pomocny->cislo;
31.          p->pomocny->cislo = z;
32.          vymena = ANO;
33.        }
34.      }
35.      p->pomocny = p->akt;
36.      x--;
37.    }while (vymena);
38.  }
```

4.-5. – nastavenie pomocných premenných.

6. – cyklus s výstupnou podmienkou. Opakujeme pokiaľ telo neprebehne bez výmeny.

9.-20. – cyklus, ktorý prechádza zoznam zľava doprava a porovnáva susedné uzly v zozname. Ak je ľavý väčší ako pravý, vymení ich (13.-19. riadok).

23.-34. – to isté ako predchádzajúci cyklus, ale sprava doľava.

37. – test na výmenu.

4.2.8. Vymazanie celého zoznamu

```
1.  void koniec (POM *p)
2.  {
3.    int x;

4.    p->akt = p->prvy;
5.    for (x = 1; x <= p->pocet_uzlov; x++)
6.    {
```

```
 7.        p->pomocny = p->akt->nasl;
 8.        free((void*)p->akt);
 9.        p->akt = p->pomocny;
10.    }
11.  }
```

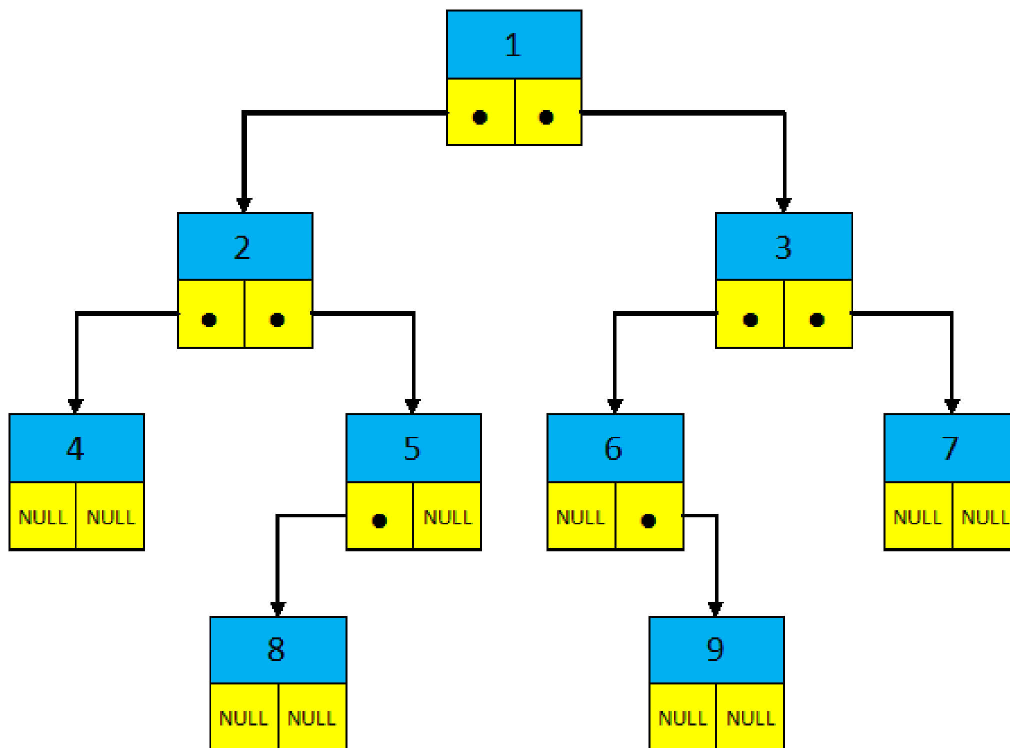
5.-10. – cyklus, ktorý prejde celým zoznamom a postupne vymazáva všetky uzly.

4.3. Binárne stromy

Podľa Stoffovej (5), na reprezentáciu binárneho stromu je najvýhodnejšie použitie zoznamu. Na jeho vytvorenie si zadefinujeme nasledujúcu štruktúru:

```
typedef struct vrchol {
      int cislo;
      struct vrchol *lavy;
      struct vrchol *pravy;
} VRCHOL;
```

Takže v každom, vrchole budú okrem samotnej hodnoty vrcholu aj dve ukazovatele na jeho synov. A to na pravého aj ľavého syna. Ak vrchol nemá syna, alebo synov, v ukazovateli uložíme hodnotu NULL. Na sprístupnenie a manipuláciu so stromom potrebujeme vedieť adresu koreňa a adresu aktuálneho vrcholu s ktorým pracujeme.



Obrázok 10 – Zobrazenie binárneho stromu pomocou štruktúry vrchol

Takže si deklarujeme ukazovateľ na koreň:

```
VRCHOL *koren, *akt;
```

4.3.1. Vytvorenie binárneho vyhľadávacieho stromu

```
 1.  if ((akt = (VRCHOL *)malloc(sizeof(VRCHOL))) == NULL)
 2.  {
 3.    printf("Malo pameti\n");
 4.  };
 5.  akt->lavy = akt->pravy = NULL;
 6.  akt->cislo = x;
 7.  vloz(&koren, akt);
 8.
 9.  void vloz(VRCHOL **koren, VRCHOL *x)
10.  {
11.    if(!(*koren))
12.    {
13.      *koren = x;
14.      return;
15.    }
16.    if (x->cislo < (*koren)->cislo)
17.    vloz (&(*koren)->lavy, x);
18.    else if (x->cislo > (*koren)->cislo)
19.    vloz (&(*koren)->pravy, x);
20.  }
```

1.-4. – alokácia pamäte pre aktuálny vrchol + test či prebehla úspešne.

5. – nastavenie smerníkov aktuálneho vrcholu na hodnoty NULL.

6. – vloženie predtým načítanej hodnoty s premennej x do hodnoty vrcholu.

7. – volanie funkcie `vloz`, ktorá vloží aktuálny vrchol do stromu, pričom ako parametre funkcie posielame ukazovateľ na ukazovateľ `koren` (ten má na začiatku hodnotu NULL) a ukazovateľ na aktuálny vrchol.

9.-20. – telo funkcie `vloz`.

11.-15. – test či je v premennej `koren` hodnota rôzna od NULL:

- ak áno, preskočí sa telo podmienky,

- ak nie, do ukazovateľa na koreň sa uloží hodnota aktuálneho vrcholu a opúšťa sa funkcia.

16. – test či hodnota aktuálneho vrcholu je menšia ako hodnota v koreni.

17. – ak áno, aktuálny vrchol pôjde do ľavého podstromu, čo zabezpečíme rekurzívnym volaním funkcie `vloz` s parametrami ukazovateľ na ukazovateľ ľavého syna aktuálneho „koreňa“ a ukazovateľ na aktuálne vkladaný vrchol.

18. – ak neprejde testom v riadku 16, test či hodnota aktuálneho vrcholu je väčšia ako hodnota v koreni.

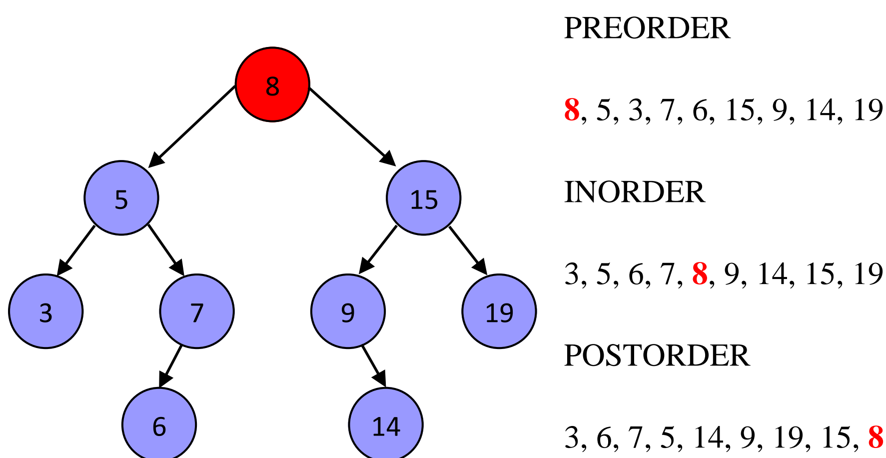
19. – ak áno, to isté ako v riadku 17, ale s tým, že vytvárame pravý podstrom.

4.3.2. Prechod binárnym vyhľadávacím stromom

Ďalšou operáciou po vytvorení stromu je jeho prechod. Knuth vo svojej práci uvádza tri typy rekurzívnych metód prechádzania binárnym stromom (6, s. 315-316). A to sú tieto:

- Priamy prechod (PREORDER) – strom sa v tomto prípade prehľadáva v poradí koreň, ľavý, pravý podstrom,
- Stredový prechod (INORDER) – strom sa prehľadáva v poradí ľavý podstrom, koreň, pravý podstrom,
- Spätný prechod (POSTORDER) – strom sa prehľadáva v poradí ľavý, pravý podstrom, koreň.

Pri prechádzaní nášho vzorového stromu, dostaneme nasledovné výsledky:



Obrázok 11 – Prechádzanie stromu rôznymi metódami

Ako môžeme vidieť, pri prechode stromom metódou INORDER dostaneme výpis hodnôt zotriedených od najmenšieho po najväčší.

```
1.  void preorder(VRCHOL *koren)
2.  {
3.    if (koren != NULL)
4.    {
5.      printf("%d,", koren->cislo);
6.      preorder(koren->lavy);
7.      preorder(koren->pravy);
8.    }
9.  }
```

```
1.  void inorder(VRCHOL *koren)
2.  {
3.    if (koren != NULL)
4.    {
5.      inorder(koren->lavy);
6.      printf("%d,", koren->cislo);
7.      inorder(koren->pravy);
8.    }
9.  }
```

```
1.  void postorder(VRCHOL *koren)
2.  {
3.    if (koren != NULL)
4.    {
5.      postorder(koren->lavy);
6.      postorder(koren->pravy);
7.      printf("%d,", koren->cislo);
8.    }
9.  }
```

3. – test či je hodnota koreňa rôzna od NULL.

5.-7. – výpis hodnoty aktuálneho koreňa, a rekurzívne volanie výpisu v poradí podľa jednotlivých metód prechodu stromom.

4.3.3. Mazanie vrcholov

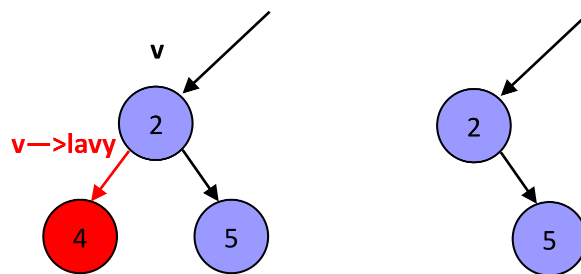
Mazanie vrcholov binárneho stromu, je už trochu zložitejšie. Môžu nastať tri rôzne prípady.

1) **Zrušenie vrcholu bez synov (listu)**

   To sa spravíme jednoducho odstránením vrcholu zo stromu a upravením ukazovateľa na syna v otcovi na hodnotu NULL. V našom príklade vymažeme

vrchol s hodnotou 4, a do ukazovateľa na ľavého syna vo vrchole 2 vložíme hodnotu NULL.

```
free((void*)v->lavy);
v->lavy = NULL;
```

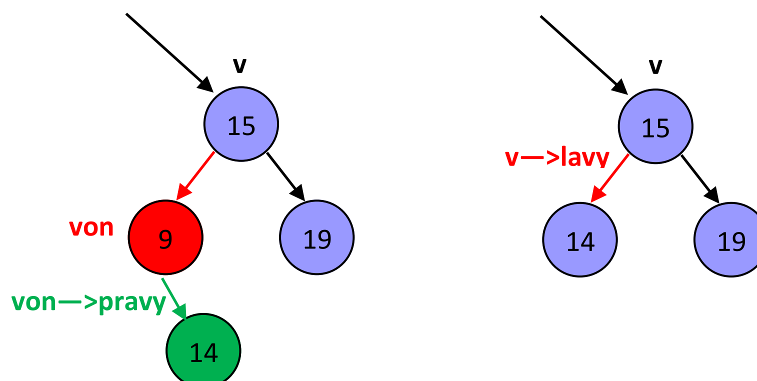


Obrázok 12 – Grafické znázornenie zrušenia vrcholu bez synov

2) Zrušenie vrcholu s jedným synom

Odstránime vrchol, ktorý chceme odstrániť, pričom do ukazovateľa na syna otca zmazaného vrcholu dáme adresu syna mazaného vrcholu. V našom príklade sa vymaže vrchol 9 a do ukazovateľa na ľavého syna vo vrchole 15 sa vložíme adresu vrcholu 14.

```
von = v->lavy;
v->lavy = von->pravy
free((void*)von);
```



Obrázok 13 – Grafické znázornenie zrušenia vrcholu s jedným synom

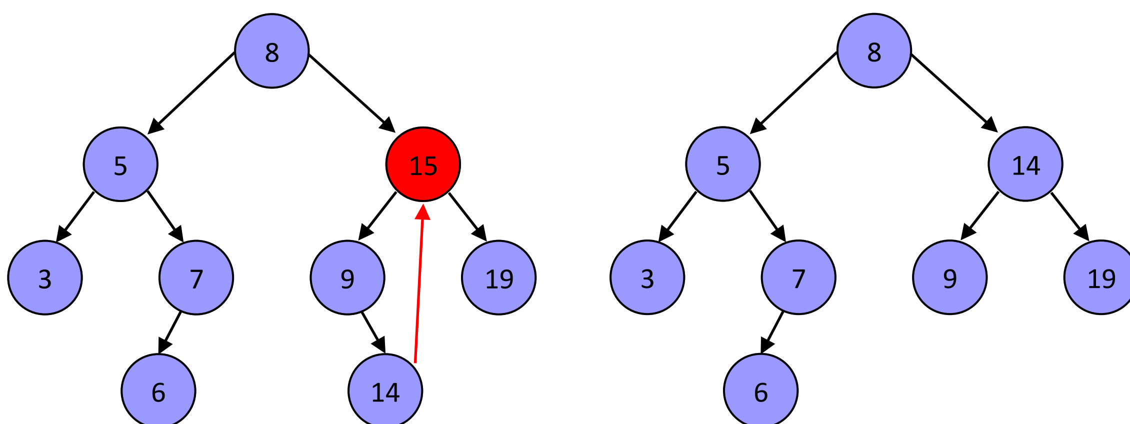
3) **Zrušenie vrcholu s dvoma synmi**

Toto je najzložitejší prípad. V tomto prípade vrchol, ktorý ideme zrušiť nahradíme, a to buď najpravejším prvkom ľavého podstromu, alebo najľavejším prvkom pravého podstromu. V našom príklade vymažeme vrchol 15, nájdeme najpravejší vrchol z ľavého podstromu, čo je vrchol 14. Jeho hodnotu 14 prepíšeme do mazaného vrcholu. Následne ešte upravíme ukazovatele vo vrchole 9. A to na hodnotu NULL, alebo na hodnotu ľavého syna najpravejšieho vrcholu, ktorého hodnotu sme prepisovali do mazaného vrcholu. K tomuto sme vytvorili dve funkcie a to `vymaz` a `prepis`.

```
 1.  void vymaz(VRCHOL **koren, int x)
 2.  {
 3.    VRCHOL *von;

 4.    if (*koren == NULL) printf("Take cislo v strome nie je\n");
 5.    else
 6.    {
 7.      if (x < (*koren)->cislo) vymaz(&(*koren)->lavy, x);
 8.      else if (x > (*koren)->cislo) vymaz(&(*koren)->pravy, x);
 9.      else
10.      {
11.        von = *koren;
12.        if (von->pravy == NULL) *koren = von->lavy;
13.        else if (von->lavy == NULL)  *koren = von->pravy;
14.        else prepis(&von->lavy, &von);
15.        free((void*)von);
16.      }
17.    }
18.  }
```

```
 1.  void prepis(VRCHOL **v, VRCHOL **von)
 2.  {
 3.    if ((*v)->pravy != NULL) prepis(&(*v)->pravy, &(*von));
 4.    else {
 5.      (*von)->cislo = (*v)->cislo;
 6.      *von = *v;
 7.      *v = (*v)->lavy;
 8.    }
 9.  }
```



Obrázok 14 – Grafické znázornenie zrušenia vrcholu s dvomi synmi

Funkcia `vymaz`

1. – hlavička funkcie, kde prichádza ukazovateľ na ukazovateľ na koreň stromu, a hodnota, ktorú chceme vymazať.

3. – pomocná premenná von, ktorá je ukazovateľom na typ VRCHOL.

4. – test na to, či sa koreň nachádza v strome, ak áno pokračujeme, ak nie tak výpis informácie o tomto stave.

7.-8. – porovnanie hodnoty aktuálneho koreňa s hľadanou hodnotou. Ak nie je rovná, rekurzívne ho hľadáme v pravom, alebo ľavom podstrome. Ak je rovná hľadanej hodnote, pokračujeme na ďalší riadok.

9.-18. – vymazanie nájdeného vrcholu.

11. - priradenia hodnoty ukazovateľa z predchádzajúceho vrcholu, ktorú budeme vymazávať do pomocnej premennej.

12. – ak vymazávaný vrchol (von) nemá pravého syna, priradíme do jeho otcovského vrcholu (v) ukazovateľ na jeho ľavého syna (von->lavy).

13. - ak vymazávaný vrchol (von) nemá ľavého syna, priradíme do jeho otcovského vrcholu (v) ukazovateľ na jeho pravého syna (von->pravy).

Riadky 12 a 13 nám pokrývajú prípad 1) a 2).

14. – ak má vymazávaný vrchol (von) oboch synov, riešime prípad 3). Pre tento prípad požijeme funkciu (podprogram) `prepis`, kde pošleme ukazovateľ na ľavý podstrom vymazávaného vrcholu a ukazovateľ na vymazávaný vrchol (von).

15. – uvoľnenie pamäte s vymazávaným prvkom.

Funkcia `prepis`

1. - hlavička funkcie, kde prichádza ukazovateľ na ukazovateľ na ľavého syna vymazávaného vrchol stromu (pretože budeme hľadať najpravejší prvok ľavého podstromu), a ukazovateľ na ukazovateľ na vymazávaný vrchol.

3. – hľadáme najpravejšieho syna, takže ak má aktuálny vrchol v pravého syna, rekurzívne voláme funkciu `prepis`.

4. – pokiaľ už vrchol v pravého syna nemá, to znamená, že on je najpravejší syn podstromu, pokračujeme na výmenu hodnôt vrcholov.

5. – do hodnoty vrcholu von načítame hodnotu najpravejšieho syna v.

6. – ukazovateľ na vrchol, ktorý chceme vymazať nahradíme adresou najpravejšieho syna.

7. – ak by mal najpravejší syn ešte ľavého syna, jeho adresa sa vloží do ukazovateľa na pravého syna.

Záver

V práci sa nám podarilo splniť ciele, ktoré sme si pri jej písaní dali. V časti venovanej triedeniu, sme rozobrali niekoľko základných postupov pri usporiadaní množiny prvkov. Každý z týchto algoritmov sme spracovali v jazyku C a tieto sme následne použili na porovnanie ich efektívnosti. Ako máme možnosť vidieť z výsledkov porovnania v kapitole 4.1.6. v rôznych situáciách sa môže ako najvýhodnejší použiť iný algoritmus. To je ale už úlohou programátorov, aby pri vytváraní softvéru vybrali najvhodnejší typ triedenia podľa úloh, ktoré potrebujú riešiť. Použitie týchto algoritmov v praxi je veľmi časté. V podstate sa používajú vo všetkých programoch, ktoré spracúvajú a zobrazujú dáta. Tieto je potrebné spracovávať a výstup sa zoraďuje podľa určitého kľúča. Pri veľkých informačných systémoch sa jedná o veľké množstvá dát, ktorých zoradenie pri výbere nevhodného algoritmu môže v porovnaní s iným trvať niekoľkonásobne dlhšie (ako to môžeme vidieť aj pri našich testoch). A takto môže negatívne ovplyvniť beh iných procesov v informačnom systéme, ktoré sú závislé na výsledku triedenia.

Podobné je to aj pri výbere vhodnej dátovej štruktúry, ktorú programátori zvolia pre danú aplikáciu. Pri výbere vhodnej štruktúry sa šetrí miesto v pamäti počítača. Môžeme ňou ovplyvniť rýchlosť a efektívnosť algoritmov a tak celkovo vplývať na efektivitu programu, alebo informačného systému.

Použitá literatúra

1. KNUTH, D.E. 1973. *The Art of Computer Programming, Vol.3 Sorting and Searching*. Addison-Wesley, 1973. 735s.

2. WIRTH, N. 1985. *Algorithm & Data Structures (Oberon version 2004).* [online]. 183s. [cit. 2011.04.04.]. Dostupné na internete: <http://www-old.oberon.ethz.ch/WirthPubl/AD.pdf>

3. KERNIGHAM, B.W. – RITCHIE, D.M. *Programovací jazyk C*. 1978. Bratislava: Alfa, 1989. 256s. ISBN 80-05-00154-1

4. HOROVČÁK, P. – PODLUBNÝ, I. *Úvod do programovania v jazyku C*. [online]. [cit. 2011.04.04.]. Dostupné na internete: <http://www.tuke.sk/podlubny/C>

5. STOFFOVÁ, V. *Dynamické údajové štruktúry: Hypertextová učebnica*. [online]. [cit. 2011.04.04.]. Dostupné na internete: <http://cec.truni.sk/stoffov/dynamicke-udajove-struktury/start.html>

6. KNUTH, D.E. 1968. *The Art of Computer Programming, Vol.1 Fundamental Algorithm*. druhé vydanie. Addison-Wesley, 1969. 636s.

7. HEROUT, P. 1994. *Učebnice jazyka C*. Č. Budejovice: Kopp, 1998. 3. Upravené vydanie. 269s. ISBN 80-85828-21-9.

8. HEROUT, P.1995. *Učebnice jazyka C 2.díl*. Č. Budejovice: Kopp, 1998. 236s. ISBN 80-85828-50-2.

9. Sorting algorithm http://en.wikipedia.org/wiki/Sorting_algorithm [online]. [cit. 2011.04.04.]