SVEUČILIŠTE U SPLITU
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

ALGORITMI I STRUKTURE PODATAKA

IZVJEŠTAJI S LABORATORIJSKIH VJEŽBI

Zvonimir Kujundžić

Datum predaje: 11.2.2024.

U ovom dokumentu teorijski su potkrijepljeni, odnosno objašnjeni, postupci i načini rješavanja i izvođenja algoritama korištenih u praktičnoj izvedbi vježbi.

Riješene vježbe su na linku: https://github.com/zkujun25/AlgoritmiIStrukturePodataka

Sadržaj - Vježbe

1. Methods and Criteria

Implementirati ćemo usporedbu dvaju članova koja se odvija prilikom soritranja. Odnosno, cilj je omogućiti korištenje različitih kriterija za usporedbu, odnosno sortiranje objekata.

1.1 Passing by Refference

Zadatak 1.1 je najjednostavniji primjer zamjene dvaju članova. Za izvedbu zadatka bilo je potrebno implementirati funckiju *Swap* koja prima dva parametra koristeći ključnu riječ ***ref*** (odnosno refference).

Naime, zamjena vrijednosti dvaju članova funkcionira na principu dodavanja trećeg privremenog člana koji čuva vrijednost jednog od druga dva člana prilikom zamjene. Izvedba toga unutar funkcije nije problem. Međutim, funkcija ima mogućnost vraćanja samo jedne vrijednosti, odnosno samo jednom možemo pisati *return*. U principu bismo mogli vratiti niz, ali to ne želimo.

Stoga, koristeći se ključnom riječi *ref* naglašavamo da ćemo primiti adresu varijable, a ne vrijednost poslane varijable. Važno je napomenuti da se u pozivu funkcije ispred naziva varijable koju šaljemo stavlja simbol ***&***¸ isto tako za razliku od C-a, u C# nije potrebno koristiti * kao naznaku da se koristi vrijednost varijable, već se koristi samo ime varijable. S obzirom na poslanu adresu, a ne vrijednost, logično je da se zamjena unutar funkcije zapravo odvija na memorijskim lokacijama dviju poslanih/primljenih varijabli, a ne s lokalnim varijablama u funkciji. Shodno tome, funkcija neće imati return jer je zamjena vidljiva na globalnoj razini programa (dogodila se unutar memorije koja se koristi u cijelom programu).

1.2 Sorting via *IComparable*

Zadatak 2 koristi se interfaceom ***IComparable*** koji je *built in* u System namespace-u. Najčešća primjena IComparable-a je korištenje njegove *CompareTo* metode. CompareTo radi na sljedeći način:

Uspoređuje trenutnu instancu sa drugim objektom istog tipa i vraća cjelobrojnu vrijednost koja indicita je li vrijednost trenutne instance veća, jednaka ili manja od vrijednosti drugog poslanog objekta. Koristiti ćemo CompareTo tako da na osnovu te vrijednosti vršimo, ili ne, zamjenu.

Vrijednosti koje vraća CompareTo su sljedeće:

- Negativan broj – vrijednost prvog objekta, manja je od vrijednosti drugog
- Nula – vrijednosti su iste
- Pozitivan broj – vrijednost prvog objekta veća je od vrijednosti drugog

Problem koji nastaje kod CompareTo je taj što se prilikom overrideanja CompareTo metode unutar objekta vežemo za kriterij po kojem uspoređujemo, odnosno ne možemo ga mijenjati. Taj problem otkloniti ćemo u narednom zadatku.

Za demonstraciju rada CompareTo u sklopu ovog zadatka definirali smo klasu Student koji ima ime i prosjek ocjena. CompareTo metodu smo napravili na način da uspoređuje ocjene studenata. Sortiranje studenata obavili smo pomoću odvojene klase Bubble koja *BubbleSort* algoritmom sortira niz studenata (parametar tipa IComparable jer se koristi CompareTo) koristeći napisanu CompareTo metodu koja je napisana i to na način da se unutar for petlji obavlja zamjena samo ako je vrijednost *array[i].CompareTo(array[j]) > 0*. Odnosno, sortirati ćemo ih silazno. Naravno, zamjena se vrši nad cijelim članom niza, odnosno nad cijelim Studentom, tako da se ne zamjene samo ocjene već cijeli student (ocjena i ime) zamjeni poziciju u nizu s drugim studentom.

1.3 Sorting via *IComparer*

Kao što je prethodno spomenuto, CompareTo je ograničen na samo jedan kriterij. U sklopu ove vježbe, kreirati ćemo kriterij tipa *enumeration* koji će sadržati moguće vrijednosti po kojima se vrši usporedba (u našem slučaju, za studenta, to su Name i Grade).

Ovoga puta, umjesto ComapreTo koristimo Compare, koji zapravo nasljeđuje CompareTo ali ga koristi više puta ovisno o kriteriju. Unutar studenta raspišemo dvije vrste usporedbi u ovisnosti o kriteriju koji je tipa enumeracije koju smo prethodno deklarirali, odnosno primati će vrijednost Name ili Grade. Ovisno o kriteriju uspoređivati ćemo ocjene ili imena.

Još jedna značajna razlika u izvedbi je prilikom poziva, ovoga puta se u Bubble ne poziva CompareTo kao što je već rečeno već Compare, i to na sljedeći način: comparer.Compare(array[i], array[j]). Vidimo da se ovoga puta oba studenta šalju kao argumenti a tip komparacije sadržan je u compareru, u kojem smo definirali Compare.

U principu funkcionalnost je ista kao u zadatku 1.2, ali poboljšana jer smo uz male izmjene omogućili više mogućih načina uspoređivanja. Dodatno bi se moglo poboljšati tako da dopustimo korisniku da unese kriterij umjesto da ga hardkodiramo kao što smo to radili u ovom zadatku u vidu demonstracije.

1.4 Using Delegates

Ovaj zadatak služi isključivo za demonstraciju rada i svrhe delegata. Delegati su zapravo posebni tipovi podataka koji omogućuju reference na metode, odnosno, pokazivači na funkcije. Bitno je napomenuti da je važno definirati koje argumente prima metoda na koju delegat pokazuje. Metoda na koju delegat pokazuje se definira tako što se pridjeli naziv metode vrijednosti delegata. A pozivom delegata vrši se izvršavanje metode koju on predstavlja, tj na koju ima referencu.

U našem slučaju imali smo delegat Invoker koji je primao cjelobrojni argument. Kroz zadatak smo mijenjali vrijednost delegata koristeći 4 napisane metode i pozivali ga.

Isto tako, prikazali smo labda funkciju, odnosno arrow (zbog =>) funkciju unutar C#. Lambda expressions su korisni pri definiranju funkcija koje su kratke i jednostavne, odnosno da bi izbjegli stvaranje metode.

1.5 Sorting via Delegates

U ovom zadatku smo zadatak 1.3 preradili koristeći znanje stečeno zadatkom 1.4. Odnosno, umjesto definiranja enumeracije Name, Grade i korištenja Compare, što je djelomično zakompliciralo zadatak u odnosu na 1.2, koristili smo delegat na funkciju. Točnije, definirali smo dvije različite metode, jednu koja uspoređuje po kriteriju name i drugu koja uspoređuje po kriteriju

grade. Unutar klase Bubble pri pozivu sorta sada šaljemo niz kojeg sortiramo i metodu koju želimo koristiti za sortiranje, a sort je tako definiran da će tu metodu primiti u svoj delegat. Zatim, korištenjem tog delegata unutar *if statement-a* slanjem array[i] i array[j] kao argumente delegata izvršavati će se poslata metoda za zamjenu. S obzirom da ta metoda radi na osnovu CompareTo može se reći da je ovaj način jednostavniji od onog u 1.3.

2. Searching

Cilj ove vježbe je prikazati mogućnosti rekurzivnih metoda u odnosu na iterativne, prikazati i upotrijebiti sekvencijalno i binarno pretraživanje te napraviti pametni niz i primjetiti razlike u odnosu na obični niz.

2.1 Recursive Methods

Rekurzivne metode su one metode koje obavljaju poziv na sebe direktno ili indirektno. U praksi, rekurzivne metode imaju dva slučaja:

1. Osnovni slučaj – slučaj kojeg metoda može riješiti i vratiti
2. Rekurzivni slučaj – slučaj u kojem metoda poziva samu sebe

Rekurzivne metode rade na način da prva poziva drugu, druga treću i tako dok jedna ne pozove metodu koja će izvršiti osnovni slučaj. Nakon što posljednje pozvana metoda vrati vrijednost osnovnog slučaja, ta vrijednost ide u onu metodu koja ju je pozvala (prethodna metoda), te se vrijednosti shodno tome izračunaju sve do prve metode.

Generalno gledano, rekurzivne metode trebaju više vremena za izvođenje od iterativnih, jer koriste više memorije u stacku (svaka iteracija funkcije postoji u stacku i čeka rezultat od funckije koju je pozvala), ali su rekurzivne metode jednostavnije za implementiranje.

U našem primjeru, napravili smo metodu za računanje faktorijela rekurzivnim načinom, i to tako da ako je vrijednost parametra <= 1 imamo osnovni slučaj i vraćamo 1 (ne smijemo vratiti nula,

jer bi konačni rezultat tako bio nula, radimo množenje). U svim ostalim slučajevima (parametar je veći od 1) imamo rekurzivni slučaj gdje vraćamo parametar pomnožen s rekurzivnim pozivom u kojem šaljemo parametar umanjen za 1 kao argument.

U drugom dijelu zadatka izveli smo računanje Fibonnacija pomoću dviju metoda, jedne iterativne i jedne rekurzivne. Samim gledanjem u napisani kod (u prilogu na gitu) očigledno je koliko je kraća i jednostavnija implementacija rekurzivnim načinom.

2.2 Sequential and Binary Search

Pretraga se odnosi na traženje nekog člana unutar određene podatkovne strukture. Pri pisanju koda cilj nam je pretragu svesti na što manju kompleksnost (O(1)). U ovom zadatku prikazati ćemo dvije vrste pretrage, sekvencijalnu i binarnu.

Sekvencijalna pretraga koristi se za pretragu unutar nizova koji nisu sortirani. Ako element ima n članova, kažemo da je kompleksnost ove pretrage O(n) jer moramo prijeći kroz sve članove i usporediti ju s vrijednosti koju tražimo. Nekada ćemo imati sreće i naći element pri početku pretraživanja, ali dugoročno gledano trebamo minimalno n/2 za pronaći, a s obzirom da je ½ konstanta, ne gledamo ju pri kompleksnosti, zato kažemo da je O(n).

Binarna pretraga koristi se nad nizovima koji su već sortirani. Ako element ima n članova kažemo da je kompleksnost ove pretrage O(log(n)). To je zato što binarna pretraga radi na sljedeći način:

- Podijelimo niz u dva niza oko središnjeg elementa
- Ako je traženi element manji od središnjeg elementa odabiremo lijevi niz (za slučaj da je originalni niz uzlazno sortiran), a ukoliko je traženi element veči od središnjeg elementa, odabiremo desni niz
- Ponovno obavljamo prve dvije točke nad odabranim nizom sve dok ne dođemo do elementa kojeg tražimo

Konstantim dijeljenjem na pola dolazimo do toga da kompleksnost ovisi o logaritmu. Ova metoda je puno brža od sekvencijalne, ali je uvjet taj da je niz već sortiran.

U sklopu ovog zadatka definirali smo tri metode, jednu za sekvencijalnu i dvije za binarnu pretragu. Sekvencijalna pretraga nije ništa drugo nego prolazak kroz cijeli niz dok ne pronađemo element. Obje metode za binarnu pretragu rade isto, samo što jedna prima 2 parametra, niz i element koji se traži, dok druga prima 4 parametra, niz, traženi element te low i high index. Pomoću high i low indexa uzimamo samo porciju niza za pretraživanje a ne cijeli niz. U našem slučaju smo ju implementirali, ali smo pretragu svejedno izvršili nad cijelim nizom (kao low poslali smo 0, a kao high poziciju zadnjeg elementa niza). Da smo poslali druge brojeve pretraga bi se vršila samo na rasponu pozicija između low i high.

2.3 Smart Arrays

U posljednjem zadatku ove vježbe cilj je bio napraviti pametni niz. Za razliku od običnih nizova gdje se memorija alocira pri definiranju samog niza, pod pojmom pametni niz misli se na niz u koji se može naknadno dodavati članove u bilo kojem trenutku, pa čak i, ukoliko za time bude potrebe, alocirati dodatno memorije.

SmartArray nije ništa drugo nego objekt koji će imati u sebi definiran niz, veličinu tog niza i mjesto na kojem je upisan posljednji član. Kako bismo mogli dohvaćati elemente niza spremljenog unutar klase SmartArray potrebno je definirati *Indexer.* Indexer nije ništa drugo nego član klase koji omogućava pristup članovima klase pomoću indeksa, u našem slučaju niza. Indexer se definira koristeći dva svojstva, ***get*** i ***set*** (u žargonu geter i seter). Prilikom korištenja indexera ne naglašava se koristi li se get ili set svojstvo već se u ovisnosti s koje strane se nalazi član s indexom poziva get, tj set. Odnosno, ukoliko imamo array[i] = a, očigledno imamo set, jer članu pridružujemo nešto, to jest, ako imamo ispis array[i] ili a = array[i], očigledno dohvaćamo vrijednost niza na određenom indexu i, samim time koristi se get.

S obzirom na to da SmartArray nije tipični niz nego klasa/objekt, potrebno je definirati i metode za rad s nizom unutar njega, a to bi bila Add, Remove i SmartEnumerator (služi za iteraciju).

Add služi za dodavanje novih članova i to na način da, ukoliko je niz popunjen, radimo novi niz s duplo više elemenata od trenutnog, u kojeg počevši od nulte pozicije kopiramo naš niz, te potom brišemo naš niz.

Remove metoda služi za brisanje članova niza, važno je voditi računa o tome da pratimo gdje je last, kako bismo mogli dodavati nove članove.

Enumerator nije ništa drugo nego vlastita implementacija foreach petlje nad nizom u objektu pomoću metode MoveNext.

Koristeći se napisanim metodama demonstrirali smo rad nad pametnim nizom.

3. Sorting

U trećoj vježbi cilj je implementirati i uočiti razlike različitih vrsta sort algoritama nad nizom podataka.

3.1 Selection Sort

Selection sort je posebna vrsta sortiranja u kojoj se kroz niz iterira više puta. Prilikom svake iteracije traži se najmanji (ili najveći) član te se mijenja s pozicijom na prvom mjestu. Zatim se iterira kroz niz počevši od drugog elementa. Postupak se ponavlja sve dok se ne dođe do kraja niza. Ovaj postupak sortiranja niza jako je jednostavan za implementaciju ali nije najjefikasniji jer je kompleksnost izvođenja $O(n^2)$ – kroz niz od n članova prolazimo n puta. Kod većih nizova izvođenje postaje predugo, tako da je pogodan isključivo za manje nizove i, eventualno, nizove koji su gotovo već sortirani. Pri implementaciji smo napravili metodu i omogućili smo primanje parametra koji nam govori odakle krećemo sa sortiranjem.

3.2 Quick Sort

Qucik Sort algoritam jedan je od najefikasnijih načina za sortiranje nizova. Njegova primjena podsjeća na BinarySearch kojeg smo prethodno odradili. Naime, QS funkcionira na principu da se odabere jedan element koji predstavlja ***pivot-a***, zatim se niz dijeli na dva niza, jedan s elementima

manjim od pivota, a drugim s elementima većim od pivota. Unutar oba niza se ponavlja postupak s pivotom i podjelom sve dok se ne dođe do toga da se niz ne može podijeliti. Očigledno je da se koristi rekurzivna metoda za primjenu.

Pri izvedbi QS-a treba paziti na sljedeće:

- **Pivot** – važno je što bolje odabrati pivota, to je najčešće prvi ili posljednji element niza, ali problem nastane kada je taj element najveći ili najmanji unutar niza. Idealno bi bilo da je pivot zapravo srednja vrijednost unutar niza.
- **Zamjena članova niza** – Da bismo omogućili podjelu niza u dva, onog s elementima manjima, i elementima većim od pivota potrebno je koristiti se s dva ***flag-a***. Na primjer leftmark (last u našoj vježbi) i rightmark (right u našoj vježbi). Leftmark kreće s prve pozicije iza pivota (neka je pivot na početku našeg niza), a rightmark je na kraju niza. Krećemo s uspoređivanjem s pivotom tako da usporedimo pivot i element koji je na leftmarku. Dokle god je element na leftmarku manji od pivota leftmark povećavamo za 1, tj pomičemo ga udesno. Kada dođemo do toga da je element na leftmarku veći od pivota, zaustavljamo leftmark i provjeravamo rightmark. Ako je element na rightmarku manji od pivota, zamijenimo element s leftmarka i rightmarka i nastavimo rad s leftmarkom kao i dosad. Ukoliko je element na rightmarku veći od pivota, rightmark pomićemo ulijevo (smanjujemo za 1) dok ne dođemo do elementa manjeg od pivota. Ovaj postupak ponavljamo dok se leftmark i rightmark ne prijeđu (dok leftmark ne bude veći za 1 od rightmarka). U tom trenutku zamjenjujemo pivot sa elementom na kojeg pokazuje prvi pokazivač odnosno rightmark u ovom slučaju (jer su se presreli leftmark i rightmark). Tada je pivot zasigurno na točnoj poziciji među nizovima. Tada možemo rekurzivno primjeniti QS nad nizovima od početne pozicije do pivota, i od pivota do kraja niza.

Najveći problem kod korištenja Quick Sort-a nastaje kada je niz već sortiran jer koji god element da je uzet za pivota uvijek će biti besmisleno particiranje.

U rješenju vježbe prikazali smo implementaciju Quick Sort algoritmima kako za usporedbu nad ocjenama studenta, tako i za usporedbu nad imenima studenata.

3.3 Priority Queues

Priority queue funkcionira na principu Heap-a što je zapravo parcijalno uređena stablasta struktura podataka. Kažemo da je djelomično uređena zato što nisu svi elementi sortirani, ali je zasigurno osigurano da su svi elementi iznad određenog veći od njega, odnosno svi ispod manji. Drugim riječima, u *root-u* će se zasigurno nalaziti najveći član. Moguća je i obrnuta primjena, kada je u korijenu najmanji član. U sklopu ove vježbe radili smo sa najvećim elementom u korijenu.

Priority queue radi na principu higher i lower order elemenata. Naime, element koji je u rootu ima najveću vrijednost, te ga se prvog briše prilikom brisanja. Element na dnu desno ima najnižu vrijednost. Prilikom ispisa niza uvijek se ispisuje element s najvišom vrijednošću, odnosno root, a njemu možemo pristupiti direktno.

Prilikom dodavanja novih elemenata, dodajemo novi element na kraj, a zatim provodimo *BubbleUp* algoritam u kojem mijenjamo mjesto s parentom ukoliko je manji od childrena (trenutnog elementa) i tako dok ne postignemo početnu uređenost. Taj se postupak često naziva i *Heapify*.

Prilikom brisanja elemenata, briše se root element, a u root se dovodi element sa dna, te se zatim provodi *BubbleDown* algoritam, slično kao *BubbleUp* dok se ne zadovolji uvjet uređenja (da su parents veći od children).

Važno je napomenuti da je rad s binarnim stablom lagan jer uvijek možemo lako pristupati roditelju i children elementu (i/2, i/2+1, i*2, i*2+1) i td..

U kontekstu ove vježbe napunili smo priority queue, ispisali ga, te potom brisali članove jednog po jednog te ispisvali ostatak queuea.

3.4 Heap Sort

Heap sort je zapravo primjena priority queue-a ali za sortiranje i sastoji se od 2 koraka. Prvi je napraviti heap od niza (*Heapify*) a potom iterativno uklanjamo elemente iz heapa, tj u ovom slučaju ih vraćamo u niz. Samim time, s obzirom da je Heap uređena struktura postižemo uzlazno ili

silazno sortiran niz, jer je u rootu uvijek pozicioniran najmanji, tj najveći element. Složenost heap sort algoritma bila bi (n logn) jer mora kroz cijeli niz proći jednom (n) da bi ga heapifyali, a zatim imamo logn.

Heap sort nije efikasan kod nizova koji imaju puno jednakih članova.

4. Lists

U ostatku vježbi baviti ćemo se različitim strukturama podataka (ne nizovima). Općenita podjela bila bi:

- Linearne strukture podataka (1-1)
- Hijerarhijske strukture podataka (1-N)
- Grafovi (N-N)
- Setovi (elementi koji nemaju međusobne poveznice, te ne postoje duple vrijednosti unutar jednog seta)

U okviru ove vježbe upoznati ćemo se s linearnom strukturom podataka, to jest dva tipa listi, jednostruko i dvostruko povezanih. Kroz zadatke ćemo prikazati implementaciju i primjenu od obje vrste.

4.1 Single Linked Lists

Općenito, liste su linearna struktura podataka koja se sastoji od više čvorova, od kojih svaki ima podatke i referencu na sljedeći čvor. Osim čvorova, važno je napomenuti da svaka lista mora imati Head, to jest referncu na prvi čvor u listi, a posljednji čvor pokazuje na null, to jest označava kraj liste. U određenim implementacijama postoji Tail, tj. referenca na kraj liste.

U radu s listama postoji mnogo operacija, važno je izdvojiti:

- Dodavanje na početak – referenca čvora kojeg umećemo mora pokazivati na element na kojeg pokazuje Head, zatim Head pokazuje na novi čvor.
- Dodavanje na kraj – referenca posljednjeg čvora pokazuje na novi čvor, a njegova referenca na null. Tu je važno napomenuti da je implementacija ovakvog dodavanja vremenski zahtjevna ukoliko ne postoji Tail jer je potrebno proći kroz cijelu listu svaki put kako bismo našli zadnji čvor.
- Brisanje s početka – postavimo da Head pokazuje na isti čvor na kojeg pokazuje čvor na kojeg pokazuje head. Time prvi čvor nestaje, tj. biva izbrisan.
- Brisanje s kraja – moramo prošetati kroz cijelu listu kako bismo pronašli čvor (predzadnji čvor) koji pokazuje na čvor čija referenca pokazuje na null. Postavljamo referencu predzadnjeg čvora na null. Kao i dodavanje na kraj i ovo je vremenski zahtjevno, ali ovaj put i ukoliko opostoji Tail, jer ne tražimo zadnji, već predzadnji čvor.
- Pretraga članova – napravimo referencu za iteratiranje i postavimo je na Head, zatim koristeći referencu na sljedeći čvor pomićemo referencu za iteratiranje dok ne pronađemo traženi element

U sklopu ovog zadatka napravili smo listu te dodavali i brisali članove s početka i kraja postupcima koji su upravo objašnjeni.

4.2 Double Linked Lists

Za razliku od jednostrukih listi, dvostruke liste, nemaju samo pokazivač na sljedeći element u listi već i na prethodni. Ovdje također imamo Head koji pokazuje na prvi element, čiji pokazivač na sljedeći čvor (u nastavku *next*) i pokazivač na prethodni čvor (u nastavku *prev*) pokazuju na samog sebe kada je on jedini. Kada postoje 3 čvora, Head pokazuje na prvi, next od prvog pokazuje na drugi, next od drugog pokazuje na treći, a next od trećeg na prvi, prev pokazivači pokazuju isto samo u drugom smjeru.

Prednost dvostruko povezanih listi je u tome što za razliku od jednostrukih, brisanje i dodavanje na kraj obavlja jednako brzo kao i dodavanje i brisanje s kraja. Međutim, mana je u tome što više

opterećuju memoriju zbog 2 reference, te je implementacija teža jer treba voditi računa o prev i next referencama priliko brisanja, dodavanja i umetanja.

U sklopu ovog zadatka napravili smo double linked listu sa studentima te demonstrirali dodavanje, brisanje i pretragu.

5. Stacks and Queues

U sklopu ove vježbe koristiti ćemo jednostruku listu iz prethodne vježbe kako bismo implementirali Stack i Queue.

5.1 Stacks

Stack je posebna izvedba liste koja funkcionira na FILO (First-in-last-out) principu. Algoritamski gledano, stack nije ništa drugo nego izvedba liste u kojoj se elementi dodaju na početak te brišu s početka, što su zapravo dvije jednostavne operacije za izvedbu (opisane u prethodnoj vježbi).

U ovom zadatku napunili smo stack (***push***) **s** određenim podacima te smo ih brisali (***pop***) *s* početka (te ispisivali nakon svakog brisanja) sve dok nismo ostali bez elemenata u listi.

5.2 Queues

Queue, odnosno red, posebna je izvedba liste koja funkcionira na principu FIFO (First-in-first-out). Drugim riječima, u izvedbi, to bi bila lista u koju se elementi dodaju na početak a brišu s kraja. S obzirom na kompleksnost brisanja s kraja (vježba 4) u odnosu na kompleksnost dodavanja na kraj, manje kompleksno je izvesti obrnuti redoslijed (LILO). Odnosno dodavati elemente na kraj, a brisati s početka. Redoslijed isipisanih članova ostale isti, a algoritam smo poboljšali.

U ovom zadatku dodavali smo elemente na kraj liste, te smo ih brisali s početka. To smo demonstrirali korištenjem pop, push i display metoda.

7. Hash Tables

Hash tablice su posebna struktura podataka koja kombinira liste i nizove kako bi omogućila direktan pristup podacima (bez iteratiranja kroz podatke).

7.1 Defining a Hash Table

Ključ dobre Hash tablice je u Hash funkciji, odnosno načinu hashiranja podataka. Naime, koristit ćemo se čvorovima koji imaju name i value svojstva te referencu na sljedeći čvor. Ti čvorovi će na osnovu hash funckije biti raspodijeljeni u određenu poziciju niza čiji element ima referencu na čvor. Drugim riječima, neka imamo niz od 10 članova, nad imenom unutar čvora primjenimo određenu matematičku funkciju te sa cjelobrojnim ostatkom sa 10 dobijemo broj od 1 do 10. Tada taj čvor dodjeljujemo poziciji u nizu koja je rezultat operacije. Taj element niza zapravo od tog trena na dalje pokazuje na upisani čvor.

Ukoliko je hash funkcija dobra, svi čvorovi će imati jedinstvene indexe te će svaki index niza pokazivati na isključivo jedan čvor (idealna izvedba). Inače, na istom elementu niza možemo dobiti listu.

7.2 Inserting Name-Value pairs

Dodavanje čvorova unutar hash tablice vrši se tako da se hashira ime te se pridjeljuje određenoj vrijednosti niza. Kao što je već opisano, ukoliko hash funkcija nije savršena na određenoj poziciji dobiti ćemo listu.

7.3 Searching for Names

Prilikom pretrage članova, hashiramo ime, automatski dobijemo poziciju na kojoj se čvor nalazi. Međutim, ukoliko je riječ o tome da imamo liste na određenim članovima moramo iteratirati kroz listu prethodno poznatim operacijama (vježbe 4 i 5).

7.4 Deleting Name-Value pairs

Brisanje članova jako je slično kao i pretraga samo ovog puta vodimo računa o prethodnom čvoru. Naime, ukoliko na poziciji npr. 2 imamo listu. Hashiranjem imena dobijemo poziciju 2, napravimo referencu na 2. Ukoliko referenca pokazuje na čvor čija je vrijednost ime koje želimo izbrisati postavljamo vrijednost naše reference na referencu čvora na kojeg naša referenca već pokazuje.

7.5 Testing Hash Tables

U našem zadatku napravili smo hash tablicu dužine 8, upisali u nju 8 parova ime-vrijednost. Vršili smo proces pretrage te brisanja kako bismo prikazali mogućnosti hash tablice.

8. Trees

Stabla su posebna hijerarhijska struktura podataka (1-N). U sklopu ove vježbe baviti ćemo se binarnim stablima, što znači da svaki čvor može imati max 2 podređena čvora (djeteta).

8.1 Defining Binary Search Trees

Binarno stablo sastoji se od n čvorova, i to na način da svaki čvor sadrži vrijednost te referencu na lijevi i referencu na desni child čvor. Definirano je tako da je za svaki čvor zagarantirano da su mu svi elementi lijevo od njega nužno i manji od njega, a svi elementi desno od njega nužno veći od njega.

8.2 Inserting in BSTs

Prilikom umetanja čvorova u binarno stablo važno je voditi računa o pozicioniranju elemenata, odnosno tome da ustrojstvo stabla ostane nepromijenjeno.

Način dodavanja elemenata je sljedeći:

- Ukoliko je stablo prazno – čvor postaje korijen stabla (element na vrhu)
- Kada stablo nije prazno – ukoliko je vrijednost manja od vrijednosti u korijenu idemo u lijevo podstablo, inače idemo u desno podstablo. Rekurzivno obavljamo postupak umetanja nad podstablom dokle god left ili right node nisu null, u tom trenu left/right referenci pridružujemo novi čvor.

8.3 Searching in BSTs

Pretraga u binarnom stablu ima kompleksnost O(logn) zato što je stablo sortirano i pri svakoj provjeri eliminiramo pola članova koje ne treba pretraživati.

Postupak pretraživanja je jednostavan. Krećemo od root-a. Ukoliko je vrijednost root, već smo ju našli. Inače, iterativno obavljamo sljedeći postupak: ukoliko je element manji od trenutnog (root u prvom trenutku) idi u lijevo podstablo, a ukoliko je veći idi u desno podstablo. Pri odlasku u novo podstablo ponovno provjerimo s novim rootom.

8.4 Deleting in BSTs

Brisanje u binarnom stablu se sastoji od različitog broja slučajeva. Najjednostavniji je kada čvor kojeg brišemo nema djece, tada ga samo izbrišemo. Ukoliko element ima jedno dijete zamijenimo čvor kojeg brišemo sa djetetom. Postupak brisanja postaje kompliciran kada čvor ima dvoje djece. U tom slučaju treba pronaći nasljednika, to jest čvor koji treba doći na izbrisano mjesto.

Kada brišemo root nasljednik je zasigurno najljeviji čvor u desnom podstablu (jer je to sljedeći srednji član). Odnosno, općenito gledano nasljednik je najljeviji čvor u desnom podstablu (iako, u teoriji to može biti i najdesniji čvor u lijevom podstablu). Kada god se briše čvor koji ima dvoje djece potrebno je zamjeniti čvor koji želimo izbrisati sa najljevijim čvorom desnog podstabla i onda nakon zamjene izbrisati najljeviji čvor (prethodni root). Brisanje s kraja je jednostavno jer tada čvor nema djece.

Pod najljevijim čvorom desnog podstabla podrazumijeva se da se uzme right podstablo od čvora, te se od novog roota ide left dokle god ne dođemo do čvora koji pokazuje null. Taj čvor je najljeviji.

8.5 Traversing in BSTs

Prilikom ispisa stabla postoje određeni načini za prolazak kroz stablo. U ovoj vježbi implementirati ćemo *InOrder*, *PreOrder* i *PostOrder* tip prolaska kroz stablo.

InOrder funkcionira je takav način prolaska kroz stablo da daje sortirani ispis. Odnosno, prvo ispisuje najljeviji element lijevog stabla, pa njegov parent, pa desni child od parenta, pa parent od parenta, pa desno podstablo i td. Na kraju podaci budu sortirano ispisani. Iako je riječima teško opisati postupak, implementacija nije ništa drugo nego postupak rekurzivnih poziva.

PreOrder je takav način prolaska kroz stablo da prvo ispisujemo root pa lijevu djecu dokle ih ima, a tek onda desnu djecu. Također, implementacija se svodi na poretku rekurzivnih poziva za ispis.

PostOrder je takav način prolaska kroz stablo gdje se prvo ispisuje najljeviji čvor, onda desni čvor od istog parenta a tek onda parent. Root se ispiše tek na kraju. Implementacija se svodi na poredak rekurzivnih ispisa.

8.6 Testing BSTs

U sklopu ove vježbe napravili smo binarno stablo, napunili ga podacima, pretraživali podatke, brisali podatke, implementirali sva 3 objašnjena traversal tipa te ih koristili za ispisivanje stabla.

9. Graphs

Grafovi su apstraktna struktura podataka koja se sastoji od skupa čvorova/vertex i skupa bridova/edge koji povezuju čvorove.

9.1 Defining an Edge

U kontekstu ove vježbe definiramo brid na način da ima dvije vrijednosti, destination i cost. Koristeći te dvije vrijednosti kasnije će biti izvedivo pronaći najkraći put, što je i cilj rada s grafom.

9.2 Defining an Vertex

U ovom zadatku radimo na implementaciji čvora/vertex-a. Prilikom implementacije čvora definirali smo broj susjednih bridova te informacije o čvoru (index, source i distance). Također, definirali smo i metodu AddEdge koja dodaje novi brid u listu susjednih bridova čvoru. Također,

kao i kod definicije brida, definicija čvora će nam pomoći pri implementaciji pretraživanja najkraćeg puta.

9.3 Graph as an Adjacency List

Tek u ovom zadatku zapravo implementiramo strukturu grafa koristeći se listom susjedstava. Graf se sastoji od vertices, odnosno niza koji sadrži sve čvorove grafa, te pot-a koji će se koristiti parcijalno uređenim stablom. Isto tako, postoji i mogućnost dodavanja novih bridova u grafu koji mogu biti usmjereni ili neusmjereni. Usmjereni bridovi su oni bridovi koji točno definiraju smjer povezivanja jednog čvora na drugi, te ih se ne može koristiti u suprotnom smjeru. Neusmjereni brid je brid koji povezuje dva čvora nebitno o smjeru, to jest može se koristiti u oba smjera. Također, definirali smo enumerator i display metode kako bismo mogli iteratirati kroz graf to jest ispisivati vrijednosti čvorova iz grafa.

9.4 Partially Order Tree

U ovom zadatku smo definirali parcijalno uređeno stablo koje nam služi za pohranu udaljenosti (priorities) čvorova od izvorišta grafa. U ovisnosti od udaljenosti čvora od izvorišta moramo koristit BubbleUp i BubbleDown metode kako bismo dobili uređeno stablo. Kažemo da je parcijalno uređeno jer nisu svi elementi sortirani, već su sortirane korelacije parent-child (slično kao heap). Partially order tree će nam služiti za izradu algoritma za pronalazak najkraćeg puta.

9.5 Finding Shortest Path

Za pronalazak najkraćeg puta koristi se algoritam kojeg je izmislio Djikstra. On se koristi kako bismo našli najkraći put između dva čvora u grafu koji ima bridove s numeričkim vrijednostima. Radi na sljedeći način:

1. Postavimo početni čvor i vrijednost najkraćeg puta do njega na 0, a vrijednosti puteva do ostalih čvorova na beskonačno.
2. Iteracijskim pristupom prolazimo kroz sve čvorove dok ih sve ne obradimo i to koristeći pot. Pot, odnosno parcijalno uređeno stablo nam pomaže pri tome tako što se pri svakoj iteraciji odabire čvor s najmanjom udaljenošću od početnog (iz pot-a) te se taj čvor označava kao obrađen a udaljenosti do njegovih susjednih čvorova se ažuriraju ako je moguće naći kraći put preko tog čvora.
3. Za svaki neobrađeni susjedni čvor algoritam provjerava ukupnu vrijednost puta koja se postiže putem od trenutnog do tog susjednog čvora. Ako je ta težina manja od postojeće ažurira ju se.
4. Ponavljaju se koraci 2 i 3 dok nema neobrađenih čvorova.

Nakon ova 4 koraka poznati su najkraći putevi od početnog čvora svih čvorova.

9.6 Testing the Graph

Implementirali smo tri načina popunjavanja grafa, graf sa samo neusmjerenim bridovima, graf sa samo usmjerenim bridovima, te graf sa miješanim bridovima. Nad grafom je izvršen postupak pronalaska najkraćeg puta za sve čvorove te je obavljen ispis u kojem su vidljivi svi čvorovi s njihovim informacijama te njima najkraći putevi.