Strukture podataka i algoritmi u Pythonu

Barukčić, Maja Marija

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Maja Marija Barukčić

STRUKTURE PODATAKA I ALGORITMI U PYTHONU

Diplomski rad

Voditelj rada:
doc. dr. sc. Vedran Čačić

Zagreb, rujan 2019.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Često u programu imamo grupu podataka koju bismo htjeli organizirati na neki način, na primjer uvesti neki redoslijed među njima. Isti problem javlja se i u svakodnevnom životu.

Uzmimo za primjer košarice u supermarketu. Košarice nisu razbacane po podu supermarketa nego su složene jedna u drugu pokraj ulaznih vrata. Kupac će uzeti košaricu s vrha, a ako primijeti da mu košarica nije potrebna, vratit će je na vrh. U programiranju postoji sličan princip rada s podacima gdje se uvijek obrađuje i briše zadnji dodani podatak. Kako bismo efikasno obrađivali podatke po tom principu većina programskih jezika ima strukturu podataka koju zovemo stog (*stack*) koja to omogućuje. Tijekom vremena uočeno je mnogo principa rada s podacima koji se često pojavljuju te danas većina programskih jezika implementira tipove kao što su stog, red i slični, o kojima ćemo više doznati u sljedećim poglavljima.

Kako bi se studente upoznalo s najčešće korištenim tipovima podataka, na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta drži se kolegij *Strukture podataka i algoritmi*. Službena literatura za taj kolegij je knjiga [3] te je ovaj rad nastao po uzoru na tu knjigu.

Prvo poglavlje uvodi osnovne pojmove koje ćemo koristiti u nastavku rada. Definirat ćemo pojmove strukture podataka, apstraktnog tipa podataka i algoritma.

U drugom poglavlju govorit ćemo o apstraktnim tipovima podataka kod kojih se podaci pohranjuju u linearnom slijedu, a međusobno se razlikuju po tome kojim redoslijedom dohvaćamo podatke koje smo spremili. To su lista, vezana lista, stog, red i dvostrani red.

U trećem poglavlju govorit ćemo o dvama vrlo sličnim pojmovima: općim stablima i binarnim stablima (uključujući binarna stabla traženja). Također ćemo navesti tri algoritma za obilazak stabla.

U četvrtom poglavlju upoznat ćemo se s prioritetnim redom, apstraktnim tipom podataka koji je sličan linearno poredanim strukturama iz drugog poglavlja.

U zadnjem, petom poglavlju upoznajemo se sa skupovima, rječnicima, preslikavanjima i binarnim relacijama.

Za apstraktne tipove podataka napravit ćemo nekoliko implementacija koristeći različite strukture podataka te komentirati njihove prednosti i nedostatke. Također, uz neke od njih napravit ćemo i analizu složenosti algoritama korištenih za implementaciju operacija nad apstraktnim tipovima podataka.

Kao osnova prilikom definicije apstraknih tipova podataka i odabira imena za metode korištena je knjiga [3], uz prilagodbe imena funkcija / metoda stilu koji odgovara programskom jeziku Python i objektno orijentiranom stilu programiranja.

Poglavlje 1

Osnovni pojmovi

1.1 Strukture podataka i apstraktni tipovi podataka

U ovom poglavlju definirat ćemo osnovne pojmove koje ćemo koristiti u nastavku.

Definicija 1.1. *Tip podataka* je skup vrijednosti koje neki podatak može poprimiti (domena) i sa skupom operacija koje se mogu izvršavati nad skupom vrijednosti.

Primjeri nekih tipova podataka su:

- cijeli broj: $2, -3, 6, -1356, \ldots$
- tekst: „s”, „Ovo je rečenica.”, „234”, ...
- istinitost: istina, laž

Definicija 1.2. *Struktura podataka* je skupina varijabli u nekom programu zajedno s vezama između tih varijabli. Stvorena je s namjerom da omogući pohranjivanje određenih podataka te efikasno izvršavanje određenih operacija nad tim podacima (upisivanje, promjena, čitanje, traženje po nekom kriteriju, ...).

Svaka struktura podataka građena je od jednostavnijih dijelova:

- *ćelija*: varijabla koju promatramo kao nedjeljivu cjelinu. U jednom kontekstu se nešto može promatrati kao ćelija, a u drugom se može promatrati unutarnja građa te cjeline.

- *polje*: više ćelija istog tipa. Ćelije koje grade polje nazivaju se *elementi polja* i označavamo ih indeksima 0, 1, ..., $N - 1$, gdje je N unaprijed zadani broj ćelija (veličina polja).
- *zapis*: skupina ćelija (ne nužno istog tipa) koje čine jednu cjelinu. Broj, redoslijed i tip ćelija je unaprijed zadan i nepromjenjiv. U objektno orijentiranim jezicima zapisi se uglavnom reprezentiraju pomoću klasa.

Definicija 1.3. *Apstraktni tip podataka* (*Abstract Data Type*), skraćeno ATP, je tip podataka definiran od strane programera koji specificira skup vrijednosti i kolekciju dobro definiranih operacija koje se izvode na tim vrijednostima.
ATP možemo definirati i kao matematički model strukture podataka koji specificira koji tip podataka je pohranjen, operacije koje možemo izvoditi na tim podacima i parametre za te operacije.

U definiciji ATP-a navodimo što koja operacija radi, ali ne i način na koji radi: kažemo da su definicije *neovisne o implementaciji*. ATP uglavnom koristimo preko njegovog sučelja (*interface*), skupa operacija definiranih nad njim. Skup operacija možemo podijeliti u 4 grupe:

- Konstruktori stvaraju i inicijaliziraju novu instancu ATP-a.
- Accessori nam vraćaju podatke sadržane u ATP-u ne mijenjajući ih.
- Mutatori modificiraju vrijednosti u ATP-u.
- Iteratori slijedno obrađuju vrijednosti iz ATP-a.

Implementacija ATP-a sastoji se od definicije strukture podataka koja prikazuje podatke iz ATP-a te potprograma kojima su realizirane operacije ATP-a pomoću odabranih algoritama. Implementacije istog ATP-a mogu se razlikovati po odabiru strukture podataka koja će se koristiti i potprogramima koji se koriste za operacije.

1.1.1 Odabir pogodne strukture podataka

Prvi korak kod implementacije ATP-a je odabir strukture podataka. No kako odabrati pogodnu strukturu podataka?
Pri odabiru si možemo pomoći postavljanjem sljedećih pitanja:

1. *Zadovoljava li struktura podataka sve zahtjeve za memorijom koji su zadani domenom ATP-a?*

Svaki ATP ima domenu. Odabrana struktura podataka mora biti u mogućnosti pohraniti sve moguće vrijednosti iz te domene.

2. *Pruža li struktura podataka mogućnost pristupa i manipulacije podacima koji su potrebni za implementaciju ATP-a?*
   Svaki ATP ima skup operacija. Struktura mora omogućiti implementaciju svih operacija bez potrebe izlaganja korisnika implementacijskim detaljima.

3. *Omogućuje li struktura podataka efikasnu implementaciju ATP-a?*
   Kod implementacije ATP-a bitno je pružiti efikasno rješenje. Nijedna struktura podataka nije primjerena za implementaciju svih ATP-a; također treba paziti na to da ovisno o načinu uporabe, u jednoj situaciji prva implementacija može biti efikasnija od druge, dok u drugoj situaciji može biti obrnuto.

Primjer 1.1. Pogledajmo sada primjer toka misli prilikom odabira strukture podataka za implementaciju ATP-a. Zamislimo da trebamo implementirati ATP za spremanje podataka o pedeset studenata. Za svakog studenta trebamo pamtiti JMBAG, ime, prezime i ocjenu iz kolegija SPA. Treba omogućiti (često) ispisivanje ocjena svih studenata, te povremeno sortiranje studenata po ocjenama.

Odlučili smo se za polje duljine 50. Svaki element polja će biti instanca klase `Student` koja ima tri člana tipa `str` i jedan tipa `int`.

Sada možemo provjeriti odgovara li odabrana struktura podataka našim zahtjevima. To ćemo učiniti postavljanjem navedena 3 pitanja.

1. *Zadovoljava li struktura podataka sve zahtjeve za memorijom koji su zadani domenom ATP-a?*
   Potrebno je spremiti podatke za pedeset studenata, a mi možemo spremiti 50 elemenata u polje. Također, svaka instanca klase ima 3 varijable tipa `str` i jednu tipa `int`. U varijablu tipa `int` možemo spremiti ocjenu, a u varijable tipa `str` možemo spremiti JMBAG, ime i prezime studenta. Na temelju toga možemo pozitivno odgovoriti na prvo pitanje.

2. *Pruža li struktura podataka mogućnost pristupa i manipulacije podacima koji su potrebni za implementaciju ATP-a?*
   Pošto smo podatke spremili u polje možemo im pristupiti i ispisati ih u bilo kojem trenutku. Također na polju možemo implementirati algoritam koji će sortirati studente po ocjenama.

3. *Omogućuje li struktura podataka efikasnu implementaciju ATP-a?*
   Kao osnovnu strukturu podataka smo koristili polje. Najčešće korištena operacija je dohvaćanje / ispis podataka, a ona se svodi na dohvaćanje jednog po jednog elementa

polja. Sortiranje studenata po ocjenama se svodi na sortiranje polja. Za implementaciju te operacije možemo koristiti neki od poznatih algoritama za sortiranje.

Kako smo na sva tri pitanja odgovorili pozitivno, možemo zaključiti da je odabrana struktura podataka pogodna za implementaciju ATP-a.

1.2 Algoritmi

U prethodnom poglavlju govorili smo o tome da ATP-i imaju operacije: na primjer, sortiranje liste ili obilazak čvorova u stablu. Kao što ćemo vidjeti u poglavlju 3, postoji nekoliko algoritama za obilazak svih čvorova u stablu. No što je zapravo algoritam?.

Definicija 1.4. *Algoritam* je metoda, proces, postupak ili pravilo za rješavanje neke klase problema (obično na računalu) koje se sastoji od konačnog niza naredbi (koje se izvršavaju određenim redom, svaka nula ili više puta).

Osnovna svojstva algoritma:

- *Ulaz*: svaki algoritam ima nula, jedan ili više (konačno mnogo) ulaznih podataka. Ulazne vrijednosti definiraju konkretnu instancu problema koju trebamo riješiti.
- *Izlaz*: svaki algoritam mora imati barem jedan izlazni podatak. Iz njega čitamo rješenje instance problema koju rješavamo.
- *Konačnost*: algoritam mora stati u konačno mnogo koraka za svaki ulaz, odnosno za svaku instancu problema.
- *Definiranost i nedvosmislenost*: svaka osnovna naredba mora biti jednoznačno definirana.
- *Efikasnost*: algoritam mora završiti u „razumnom" vremenu i uz korištenje „razumne" količine memorije.

Pogledajmo svojstva algoritama na primjeru konkretnog algoritma, Euklidovog algoritma za nalaženje najveće zajedničke mjere dva prirodna broja.

Primjer 1.2. Euklidov algoritam možemo zapisati na sljedeći način. Kako bismo odredili najveći zajednički djelitelj (skraćeno *NZD*) brojeva A i B, radimo sljedeće:

- ako je $A = 0$, tada je $NZD(A, B) = B$ i algoritam završava;

- ako je $B = 0$, tada je $NZD(A, B) = A$ i algoritam završava;
- zapišemo A kao $A = q \cdot B + r$, pri čemu je $0 \leq r < B$;
- odredimo $NZD(B, r)$ korištenjem Euklidovog algoritma;
- koristimo svojstvo $NZD(A, B) = NZD(B, r)$.

Vidimo da algoritam ima dva ulazna podatka: prirodne brojeve A i B. Izlaz algoritma je prirodni broj $NZD(A, B)$. Kako su A i B prirodni brojevi, algoritam će stati u konačnom vremenu. Svaki korak algoritma je nedvosmisleno definiran. Algoritam za izvršavanje u svakom trenutku treba u memoriji držati dva cijela broja. Također, za broj koraka j u algoritmu vrijedi $j < 2 \log_2 \min\{A, B\}$. Dokazi svih iskazanih tvrdnji o Euklidovom algoritmu mogu se pronaći u [1].

Kada rješavamo problem najčešće postoji nekoliko načina / algoritama koje možemo koristiti. Kako bismo utvrdili koji je najbolji, moramo prvo odlučiti po kojem kriteriju ćemo rangirati algoritme (na primjer po vremenu potrebnom za izvršavanje, po potrebnoj memoriji, ...). U tu svrhu uvodimo pojam složenosti algoritma.

Definicija 1.5. *Složenost algoritma* je cijena korištenja tog algoritma za rješavanje jednog problema iz zadane klase, iskazana kao funkcija veličine problema. Obično mjeri vrijeme izvođenja ili potrebnu memoriju.

Kako bismo ocjenjivali različite algoritme, definiramo notaciju „veliko O".

Definicija 1.6. Neka su $f, g : \mathbb{N} \to \mathbb{R}_0^+$ dvije funkcije. Kažemo da je funkcija g *asimptotska gornja međa* za funkciju f ako postoje $c > 0$ i $n_0 \in \mathbb{N}$ tako da za svaki $n \geq n_0$ vrijedi $f(n) \leq c \cdot g(n)$. Oznaka: $f(n) = O(g(n))$.

Neka svojstva O-notacije:

- ako vrijedi $f_1(n) = O(g_1(n))$ i $f_2(n) = O(g_2(n))$
  tada je $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$
- ako vrijedi $f_1(n) = O(g_1(n))$ i $f_2(n) = O(g_2(n))$ tada je $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$
- ako je $p : \mathbb{N} \to \mathbb{R}_0^+$ polinom stupnja k tada vrijedi $p(n) = O(n^k)$.

Prije daljnje rasprave pogledajmo sljedeći primjer.

Primjer 1.3. Neka su $f_1(x) = x^2$, $f_2(x) = 2x^2$ i $f_3(x) = x^3$. Prema definiciji 1.6 vrijedi:

1. neka je $c = 1$ i $n_0 = 1$, tada za svaki $n \geq n_0$ vrijedi $f_1(x) = x^2 \leq 1 \cdot x^4$ pa je $f_1(x) = O(x^4)$
2. neka je $c = 2$ i $n_0 = 1$, tada za svaki $n \geq n_0$ vrijedi $f_2(x) = 2x^2 = c \cdot x^2 \leq c \cdot x^4$ pa je $f_2(x) = O(x^4)$
3. neka je $c = 1$ i $n_0 = 1$, tada za svaki $n \geq n_0$ vrijedi $f_3(x) = x^3 \leq c \cdot x^3$ pa je $f_3(x) = O(x^3)$

Kao što vidimo iz navedenog primjera, funkcije f_1 i f_2 su $O(x^4)$ dok je $f_3 = O(x^3)$. Na temelju tih zaključaka mogli bismo krivo zaključiti da funkcije f_1 i f_2 rastu brže od funkcije f_3, iako znamo da x^3 raste brže od x^2. Iz tog razloga uvodimo oznaku Θ.

Definicija 1.7. Neka su $f, g : \mathbb{N} \to \mathbb{R}_0^+$ dvije funkcije. Kažemo da su funkcije f i g *istog reda veličine*, i pišemo $f(x) = \Theta(g(x))$, ako postoje $c_1 > 0$, $c_2 > 0$ i $n_0 > 0$ tako da za svaki $n \geq n_0$ vrijedi $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

Primjer 1.4. Vratimo li se funkcijama f_1, f_2 i f_3 iz primjera 1.3 možemo zaključiti:

1. neka je $c_1 = 1$, $c_2 = 1$ i $n_0 = 1$, tada za svaki $n \geq n_0$ vrijedi $c_1 \cdot x^2 \leq f_1(x) = x^2 \leq c_2 \cdot x^2$ pa je $f_1(x) = \Theta(x^2)$
2. neka je $c_1 = 1$, $c_2 = 2$ i $n_0 = 1$, tada za svaki $n \geq n_0$ vrijedi $c_1 \cdot x^2 \leq f_2(x) = 2x^2 \leq c_2 \cdot x^2$ pa je $f_2(x) = \Theta(x^2)$
3. neka je $c_1 = 1$, $c_2 = 1$ i $n_0 = 1$, tada za svaki $n \geq n_0$ vrijedi $c_1 \cdot x^3 \leq f_3(x) = x^3 \leq c_2 \cdot x^3$ pa je $f_3(x) = \Theta(x^3)$

Za razliku od O-notacije, Θ-notacija nam daje i gornju i donju ocjenu brzine rasta funkcije. Zato ćemo prilikom usporedbe algoritama koristiti Θ, a ne O.

U daljnjim poglavljima za računanje složenosti algoritma brojit ćemo osnovne operacije koje algoritam napravi kako bi od ulaza duljine n generirao izlaznu vrijednost.

Osnovne operacije bit će nam:

- pridruživanje;
- usporedba dva elementa;

- računske operacije.

Primjer 1.5. Pogledajmo sada primjer brojanja operacija. Imamo dva isječka koda. Oba isječka računaju zbroj elemenata u pojedinom retku kvadratne matrice A reda n, i zbroj svih elemenata u matrici A.

```
totalSum = 0
rowSum = [0] * n
for i in range(n):
    for j in range(n):
        rowSum[i] = rowSum[i] + A[i][j]
        totalSum = totalSum + A[i][j]
```

Možemo primijetiti da imamo dvije petlje. Tijelo vanjske petlje izvršit će se n puta. U svakom prolazu kroz vanjsku petlju, tijelo unutarnje petlje izvršit će se n puta. Zaključujemo da ćemo n^2 puta proći kroz unutarnju petlju, a pri svakom prolasku izvršit će se 4 operacije što znači da će se ukupno izvršiti $4 \cdot n^2$ osnovnih operacija.

```
totalSum = 0
rowSum = [0] * n
for i in range(n):
    for j in range(n):
        rowSum[i] = rowSum[i] + A[i][j]
    totalSum = totalSum + rowSum[i]
```

U unutarnjoj petlji pojavljuju se dvije operacije (jedno zbrajanje i jedno pridruživanje) koje se izvršavaju n puta, tj. ukupno $2 \cdot n$ operacija pri svakom izvršavanju unutarnje petlje. Tijelo vanjske petlje se izvršava n puta, i pri svakom izvršavanju imamo $2n$ operacija iz unutarnje petlje, te jedno zbrajanje i jedno pridruživanje. Ukupno imamo

$$n \cdot (2n + 2) = 2n^2 + 2n$$

operacija.

Vidimo da, iako oba isječka računaju istu stvar, to rade sa različitim brojem operacija. Pogledajmo sada složenost isječaka.
Za prvi isječak možemo uzeti $c_1 = c_2 = 4$ i $n_0 = 1$. Tada ćemo imati

$$4 \cdot n^2 \leq 4n^2 \leq 4 \cdot n^2$$

iz čega možemo, prema definiciji 1.7, zaključiti da je $4n^2 = \Theta(n^2)$.
Za drugi isječak možemo uzeti $c_1 = 2$, $c_2 = 4$ i $n_0 = 1$. Iz

$$2 \cdot n^2 \leq 2n^2 + 2n \leq 2 \cdot n^2 + 2 \cdot n^2 \leq 4 \cdot n^2$$

slijedi da je $2n^2 + 2n = \Theta(n^2)$.

Iako će prvi isječak trebati više operacija za svaki $n > 1$, razlika nije značajno velika jer je u oba primjera broj operacija reda veličine $\Theta(n^2)$.

Primjer 1.6. Pogledajmo primjer funkcije koja vraća indeks elementa u polju, odnosno -1 ako polje nema element s tom vrijednošću.

```
def find(arr, value) -> int:
    idx = 0
    for x in arr:
        if x == value:
            return idx
        idx = idx + 1
    return -1
```

Neka je n duljina ulaznog polja *arr*. Vidimo da funkcija ima jednu petlju koja prolazi elementima polja. Razlika, u odnosu na isječke iz prethodnog primjera, je da se neće uvijek izvršiti cijela petlja. Ovisno o ulaznim podacima, tijelo petlje se može izvršiti jednom, dva puta, ... ili n puta.

Neka je $arr = [0, 1, 2, 3, 4, 5]$. Pogledajmo različite pozive funkcije *find* i broj broj osnovnih operacija koje su izvršene:

- $find(arr, 0)$ ... tijelo petlje izvršeno je jednom i ukupno imamo jedno izvršavanje osnovnih operacija
- $find(arr, 1)$ ... tijelo petlje izvršeno je dva puta i ukupno imamo dva izvršavanja osnovnih operacija
- $find(arr, 5)$ ... tijelo petlje izvršeno je šest puta i ukupno imamo šest izvršavanja osnovnih operacija
- $find(arr, 7)$ ... tijelo petlje izvršeno je šest puta i ukupno imamo sedam izvršavanja osnovnih operacija

Općenito, broj izvršenih operacija je jednak indeksu elementa u polju, odnosno $n + 1$ ako se element ne nalazi u polju.

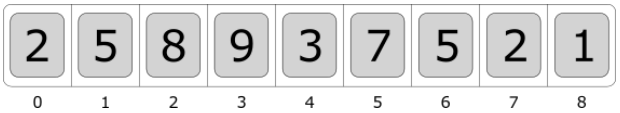
U primjeru 1.6 vidimo da broj izvršenih operacija može ovisiti o ulaznim podacima. U tom slučaju ne možemo odrediti točan broj operacija i na temelju njega odrediti složenost funkcije. Zbog toga ćemo kod određivanja složenosti gledati najgori slučaj (*worst case*). U primjeru 1.6 to je slučaj kada se traženi element ne nalazi u polju i tada imamo $n + 1$ operacija. Na temelju toga zaključujemo da je složenost $\Theta(n)$.

Poglavlje 2

Liste

2.1 Polje

Najjednostavniji tip podataka za pohranu i pristup kolekciji podataka je polje (*array*). Riječ je o sekvencijalnom tipu podataka zadane duljine (nakon što je polje stvoreno nije mu moguće mijenjati duljinu). Polje se često koristi kao osnovna struktura za gradnju složenijih tipova, kao što su višedimenzionalna polja, liste, ...



Slika 2.1: Primjer jednodimenzionalnog polja s devet elemenata

Iako se polje u mnogo programskih jezika može naći kao osnovni tip, Python uobičajeno koristi listu za stvaranje nizova promjenjive duljine. Promatrajući različite implementacije Pythona možemo vidjeti sljedeće:

- u implementaciji CPython `list` je implementiran korištenjem polja
- u implementaciji JPython `list` je implementiran korištenjem ArrayList
- u implementaciji IronPython `list` je implementiran korištenjem Array
- ...

Strukture podataka koje su korištene prilikom implementacije klase `list` su reprezentacije polja u odgovarajućim programskim jezicima. Iz tog razloga ćemo u daljnjim poglavljima, kada se pojavi potreba za poljem, koristiti Pythonov `list` uz ograničenje duljine.

2.2 Lista

Nešto složeniji tip od polja su liste. Za razliku od polja, duljina liste je promjenjiva. Općenito listu možemo definirati na sljedeći način:

Definicija 2.1. *Lista* (*list*) je konačni niz (nula ili više) podataka istog tipa. Podaci koji čine listu nazivaju se njezini *elementi*. Listu obično zapisujemo na način

$$[a_1, a_2, a_3, \ldots, a_n]$$

pri čemu $n \geq 0$ predstavlja *duljinu liste*. U slučaju kada je $n = 0$ kažemo da je lista prazna.

Napomena: S obzirom da indeksiranje u Pythonu kreće od 0, i mi ćemo liste indeksirati na sljedeći način:

$$[a_0, a_1, a_2, \ldots, a_{n-1}]$$

ATP *Lista* možemo definirati na sljedeći način:

Lista

elementtype ... bilo koji tip
List ... konačni niz (ne nužno različitih) podataka tipa *elementtype*
position ... tip podataka koji služi za zadavanje pozicije u listi. Ako imamo listu $[a_0, a_1, a_2, \ldots, a_n]$ onda su definirane pozicije koje odgovaraju prvom (a_0), drugom (a_1), ..., n-tom (a_{n-1}) elementu, i pozicija na kraju liste (neposredno nakon n-tog elementa).
first(L) ... vraća prvu poziciju u listi L. Ako je L prazna vraća $end(L)$.
end(L) ... vraća poziciju na kraju liste L
make_null(L) ... pretvara listu L u praznu listu
insert(L, x, p) ... ubacuje podatak x na poziciju p u listi L. Pri tome se elementi, koji su do tada bili na pozicijama p i dalje, pomiču za jednu poziciju dalje. Ako smo na početku imali listu $L = [a_0, a_1, a_2, \ldots, a_{n-1}]$ i a_p se nalazi na poziciji p tada nakon poziva funkcije $insert(L, x, p)$ imamo $L = [a_0, a_1, a_2, \ldots, a_{p-1}, x, a_p, \ldots, a_{n-1}]$. U slučaju $p = end(L)$ dobijemo $L = [a_0, a_1, a_2, \ldots, a_{n-1}, x]$.
append(L, x) ... ubacuje element x na kraj liste L. Odgovara pozivu metode $insert(L, x, end(L))$.

delete(L, p) ... briše element na poziciji p u listi L. Ako smo na početku imali listu $L = [a_0, a_1, a_2, \ldots, a_{n-1}]$ i a_p se nalazi na poziciji p, nakon poziva funkcije $delete(L, p)$ imat ćemo $L = [a_0, a_1, a_2, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{n-1}]$.
next(L, p) ... vraća poziciju nakon pozicije p u listi L. Nije definirana ako je $p = end(L)$.
previous(L, p) ... vraća poziciju prije pozicije p u listi L. Nije definirana ako je $p = first(L)$.
retrieve(L, p) ... vraća element na poziciji p u listi L. Nije definirana ako je $p = end(L)$.
set_value(L, x, p) ... postavlja vrijednost elementa na poziciji p u listi L na x. Nije definirana ako je $p = end(L)$.
Metode *insert*, *delete*, *next*, *previous*, *retrieve* i *set_value* nisu definirane ako L nema poziciju p.

Pogledajmo sada implementaciju liste. Kao i uvijek, postoji više mogućnosti.

1. Fizički redoslijed elemenata u memoriji podudara se s logičkim redoslijedom u listi. U tom slučaju nam kao struktura podataka najbolje odgovara polje. Kako Python nema ugrađen tip polje, koristit ćemo `list` kao što smo napomenuli u točki 2.1.

2. Elementi liste pohranjeni su na proizvoljnim lokacijama u memoriji. U tom slučaju se fizički redoslijed u memoriji ne podudara s logičkim redoslijedom u listi. Kako bismo zadali logički redoslijed u listi koristimo neku vrstu reference. Takve liste se često nazivaju *vezanim listama* i njihovu implementaciju dat ćemo u točki 2.3.

2.2.1 Implementacija liste pomoću polja

U ovom dijelu pogledat ćemo detalje vezane uz implementaciju liste pomoću polja (tj. Pythonov `list`)

```
1 from dataclasses import dataclass
2
3 elementtype = str
4 MAXLENGTH = 9
5
6 @dataclass
7 class List:
8     position = int
9     last: position
```

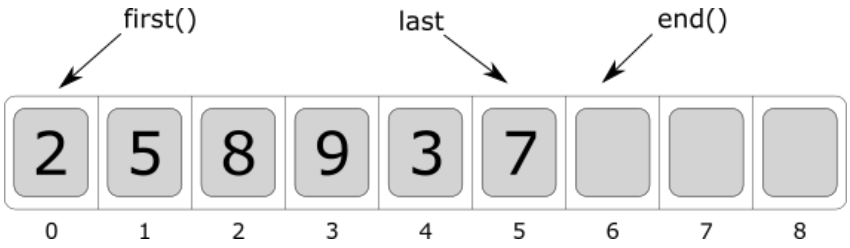
```
10      elements: [elementtype]
11
12      @classmethod
13      def make_null(cls) -> 'List':
14          return cls(last=-1, elements=[None] * MAXLENGTH)
15
16      def first(L) -> position:
17          return 0
18
19      def end(L) -> position:
20          return L.last + 1
21
22      def next(L, p: position) -> position:
23          if not L.first() <= p < L.end():
24              raise ValueError(f'Position {p} out of range')
25          return p + 1
26
27      def previous(L, p: position) -> position:
28          if p == L.first():
29              raise ValueError('First position')
30          if not L.first() < p <= L.end():
31              raise ValueError(f'Position {p} out of range')
32          return p - 1
33
34      def retrieve(L, p: position) -> elementtype:
35          if not L.first() <= p < L.end():
36              raise ValueError(f'Position {p} out of range')
37          return L.elements[p]
38
39      def set_value(L, x: elementtype, p: position) -> None:
40          if not L.first() <= p < L.end():
41              raise ValueError(f'Position {p} out of range')
42          L.elements[p] = x
43
44      def insert(L, x: elementtype, p: position) -> None:
45          if L.last >= MAXLENGTH - 1:
46              raise MemoryError('Full list')
47          if not L.first() <= p <= L.end():
48              raise ValueError(f'Position {p} out of range')
```

```
49          for q in range(L.last, p - 1, -1):
50              L.elements[q + 1] = L.elements[q]
51          L.elements[p] = x
52          L.last += 1
53
54      def append(L, x: elementtype) -> None:
55          L.insert(x, L.end())
56
57      def delete(L, p: position) -> None:
58          if not L.first() <= p < L.end():
59              raise ValueError(f'Position {p} out of range')
60          for q in range(p, L.end()):
61              L.elements[q] = L.elements[q + 1]
62          L.last -= 1
```

Isječak 2.1: Implementacija ATP-a List pomoću polja

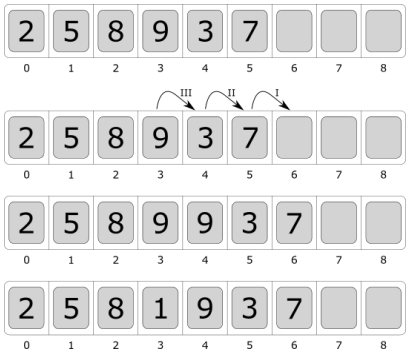
Na početku primijetimo da smo, prije definicije same klase, definirali tip *elementtype* kao `str`, te postavili duljinu polja *MAXLENGTH* na devet.

Vidimo da imamo jedan klasni atribut *position*, koji predstavlja tip pozicije. U našem slučaju tip je `int`, jer u ovoj implementaciji pozicije su indeksi. Također imamo dvije članske varijable. Varijabla *last* predstavlja poziciju zadnjeg elementa u polju, a *elements* je polje duljine *MAXLENGTH* u koje spremamo elemente tipa *elementtype*.



Slika 2.2: Primjer polja sa označenim vrijednostima

Pogledajmo prvo metodu *insert*. Metoda radi tako da prvo provjeri je li lista puna i pozicija p valjana. Nakon toga pomiče sve elemente na pozicijama većim ili jednakim p za jedno mjesto te postavlja vrijednost elementa na poziciji p na x. Elementi se pomiču zdesna nalijevo; prvo se pomakne zadnji element, pa predzadnji, ...



Slika 2.3: Primjer rada metode *insert*. Rimskim brojevima označen je redoslijed pomicanja elemenata.

Metoda *delete* radi po sličnom principu. Prvo provjeri valjanost pozicije p, a zatim pomiče elemente polja. Za razliku od *insert*, ovdje se pomicanje izvršava slijeva nadesno.

Pogledajmo sada ostale metode. Metoda *make_null* vraća praznu listu tako da alocira polje duljine *MAXLENGTH* i postavi *last* na -1. Metoda *first* vraća 0, a *end* poziciju nakon zadnjeg elementa. Metoda *next* vraća poziciju nakon, a *previous* prije pozicije koju smo predali metodi. Metoda *retrieve* vraća element na zadanoj poziciji, a *set_value* postavlja vrijednost elementa. Metoda *append* poziva metodu *insert* sa pozicijom *end()*.

Pogledajmo sada složenosti metoda. Označimo s n broj elemenata u listi.
Metoda *insert* će u najgorem slučaju, kada dodajemo element na početak liste, morati pomaknuti sve elemente liste za jedno mjesto. U tom slučaju ćemo imati n pridruživanja. Također će povećati *last* za 1 i pridružiti novu vrijednost prvom elementu što znači da imamo ukupno $n + 2$ pridruživanja. Dakle, složenost operacije *insert* je $\Theta(n)$.
Istim razmišljanjem možemo zaključiti da je složenost operacije *delete* $\Theta(n)$.
Metoda *append* se svodi na poziv metode *insert*, ali s pozicijom *end()*. Iz tog razloga nikada neće ući u petlju te će ukupno biti dvije operacije, dakle operacija *append* ima složenost $\Theta(1)$.
Kod ostalih metoda vidimo da broj operacija ne ovisi o n pa možemo zaključiti da je njihova složenost $\Theta(1)$.

2.2.2 Nedostaci implementacije liste pomoću polja

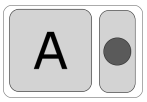
Kod ubacivanja novih ili brisanja starih elemenata liste implementirane pomoću polja potrebno je pomicati elemente kako bismo napravili mjesta za nove elemente ili uklonile rupe nakon brisanja.

Veličina polja je fiksna i nije ju moguće mijenjati. Iako duljina liste nije fiksna, zbog implementacije pomoću polja, svaki puta kada lista previše naraste, potrebno je „ručno" stvoriti novo dovoljno veliko polje i prepisati sve elemente u njega.

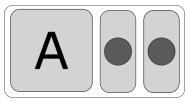
Kako su elementi polja spremljeni na uzastopnim adresama u memoriji, kod svakog stvaranja polja potrebno je naći dovoljno veliki komad memorije. To može predstavljati problem kod duljih polja.

2.3 Vezana lista

Vezana lista je struktura koja je građena od čvorova koji su povezani tako da grade linearnu strukturu. Svaki *čvor* sadrži vrijednost i barem jednu referencu / pointer na drugi čvor. Primjeri čvorova prikazani su na slici 2.4.



(a) Čvor s podatkom „A" i jednom referencom na drugi čvor

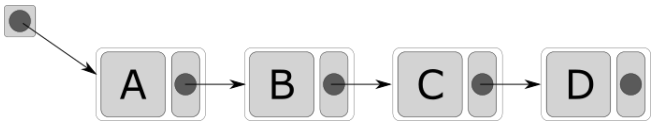


(b) Čvor s podatkom „A" i dvije reference na čvorove

Slika 2.4: Čvorovi u vezanim listama

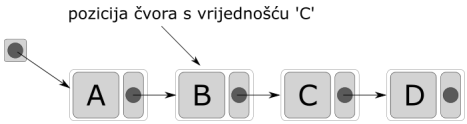
2.3.1 Implementacija liste kao jednostruko vezane liste

U vezanoj listi, osim čvorova, postoji i referenca na prvi čvor u listi (*head*). Kod jednostruko vezanih listi svaki čvor ima jednu referencu koju ćemo označavati s *next*. Kraj liste označen je tako što zadnji čvor ima nul-referencu kao vrijednost od *next*. Vezana lista može biti prazna i u tom slučaju je *head* nul-referenca.



Slika 2.5: Jednostruko vezana lista s 4 čvora

Napomena 2.1. Kod jednostruko vezanih listi je pozicija čvora *N* definirana kao pointer na čvor čiji pointer *next* pokazuje na *N*. Ilustracija je vidljiva na slici 2.6.



Slika 2.6: Pozicija čvora s vrijednošću „C”

Pogledajmo sad implementaciju.
Na isječku 2.2 vidimo da klasa ima atribut: *position*, koji predstavlja poziciju elementa i u ovom je slučaju jednaka samoj klasi jer je imati referencu na čvor isto što i imati referencu na listu. Također imamo dvije članske varijable: *element* je varijabla u koju spremamo vrijednost tipa *elementtype*, a *nextp* je referenca na sljedeći element.

```
6 class List:
7     position = 'List'
8     element: elementtype
9     nextp: position
```

Isječak 2.2: Atribut i varijable u klasi `List`

Pogledamo li isječak 2.3 vidimo da metoda *end* u najgorem slučaju mora proći kroz sve elemente polja. Iz toga možemo zaključiti da je njezina složenost $\Theta(n)$.

```
18     def end(L) -> position:
19         p = L
20         while p.nextp:
21             p = p.nextp
22         return p
```

Isječak 2.3: Implementacija metode *end*

Također, metoda *previous* (vidi isječak 2.4) mora, kako bi vratila poziciju prije p, proći kroz sve elemente polja prije p. U najgorem slučaju, kada je $p = end()$, morat će proći kroz sve elemente polja te je i njezina složenost $\Theta(n)$.

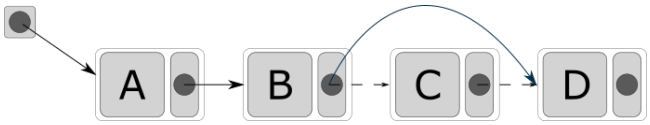
```
27     def previous(L, p: position) -> position:
28         q = L
29         while q.nextp != p:
30             q = q.nextp
31         return q
```

Isječak 2.4: Implementacija metode *previous*

Iako je složenost prethodno spomenutih metoda gora nego složenost istih metoda kod implementacije pomoću polja, postoje i metode koje imaju bolju složenost. Jedna od njih je *delete*. Pogledamo li isječak 2.5 vidimo da se brisanje elementa svodi na jednu usporedbu i jedno pridruživanje te je složenost ove metode $\Theta(1)$. Ovakva složenost omogućena je dobrim odabirom definicije za poziciju elementa (vidi napomenu 2.1). Sam rad metode ilustriran je slikom 2.7.

```
46      def delete(L, p: position) -> None:
47          if p.nextp:
48              p.nextp = p.nextp.nextp
```

Isječak 2.5: Implementacija metode *delete*

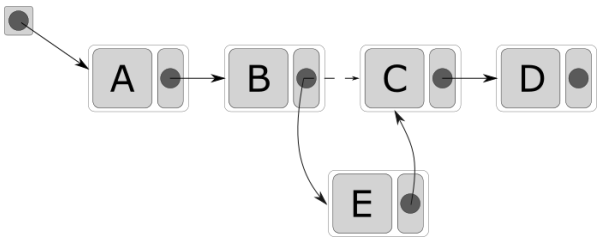


Slika 2.7: Brisanje čvora s vrijednošću „C”

Još jedna metoda čiji rad možda nije jasan na prvi pogled je metoda *insert*. Kao kod *delete*, ni ovdje nema potrebe za pomicanjem elemenata nego se sve svodi na stvaranje novog čvora i postavljanje pointera *nextp* (vidi sliku 2.8). Zbog toga je njezina složenost $\Theta(1)$.

```
39      def insert(L, x: elementtype, p: position) -> None:
40          temp = p.nextp
41          p.nextp = List(element=x, nextp=temp)
```

Isječak 2.6: Implementacija metode *insert*



Slika 2.8: Dodavanje čvora u sredini liste

Implementacije ostalih metoda su intuitivne te ih nećemo pobliže objašnjavati. Samo ćemo napomenuti da je njihova složenost $\Theta(1)$.

2.3.2 Usporedba implementacija liste pomoću polja i vezane liste

Metoda	polje	vezana lista
first	$\Theta(1)$	$\Theta(1)$
end	$\Theta(1)$	$\Theta(n)$
make_null	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(n)$	$\Theta(1)$
append	$\Theta(1)$	$\Theta(1)$
delete	$\Theta(n)$	$\Theta(1)$
next	$\Theta(1)$	$\Theta(1)$
previous	$\Theta(1)$	$\Theta(n)$
retrieve	$\Theta(1)$	$\Theta(1)$
set_value	$\Theta(1)$	$\Theta(1)$

Tablica 2.1: Usporedba složenosti metoda u različitim implementacijama liste

Pogledamo li tablicu 2.1 vidimo da se složenosti većine metoda podudaraju, no ne svih. Dok implementacija pomoću polja ima bolje performanse kod metoda *previous* i *end*, implementacija pomoću vezane liste ima bolju složenost kod metoda *insert* i *delete*.

Zbog toga ne postoji jedinstveni odgovor na pitanje koja je implementacija bolja. Implementaciju ćemo birati ovisno o tome kako namjeravamo koristiti listu. Ako imamo puno dodavanja i brisanja elemenata s početka ili sredine liste, odabrali bismo implementaciju pomoću vezane liste jer ćemo kod tih operacija imati bolje performanse. S druge strane, ako imamo uglavnom dodavanja i brisanja s kraja liste (vidi poglavlje 2.4) mogli bismo odabrati i implementaciju pomoću polja.

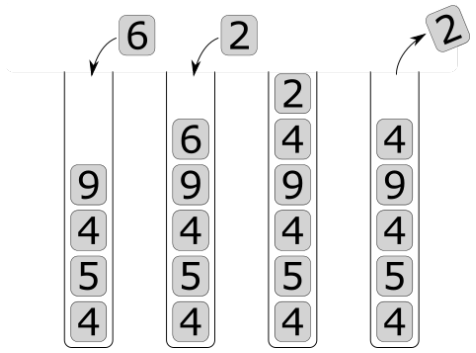
Važno je istaknuti i nedostatke pojedinih implementacija. Najveći nedostatak implementacije pomoću polja, uz složenost metoda *insert* i *delete*, je to da moramo odabrati fiksnu duljinu polja s kojom radimo i ne možemo ju mijenjati. To može predstavljati problem ako ne znamo unaprijed koliko elemenata će naša lista imati. Ako primijetimo da nam treba više prostora, moramo napraviti novo veće polje, te prepisati sve podatke u njega. Naravno, ako nam je maksimalna duljina liste unaprijed poznata i duljina ne varira previše, tada možemo zanemariti ovaj nedostatak implementacije.

S druge strane, kod implementacije pomoću vezane liste nemamo ograničenja na duljinu (osim memorije samog računala), no kod ove implementacije javlja se drugi problem. Za svaki element, osim vrijednosti, moramo pamtiti dodatni podatak, referencu na sljedeći element. Samim time trošimo više memorije po elementu.

2.4 Stog

Stog je kolekcija objekata koji se ubacuju i izbacuju po principu *last in, first out (LIFO)*. Objekti se mogu ubacivati u bilo kojem trenutku, ali se može pristupiti samo zadnjem ubačenom objektu, za kojeg kažemo da se nalazi na *vrhu* stoga.

Princip rada stoga je prikazan na slici 2.9.



Slika 2.9:
(a) dodavanje vrijednosti 6 na stog
(b) dodavanje vrijednosti 2 na stog
(c) stanje stoga nakon što smo dodali 6 i 2
(d) uklanjanje vrijednosti s vrha stoga

Primjer 2.2. Pogledajmo sada nekoliko primjera stoga iz svakodnevnog života:

- „Toranj" tanjura na polici možemo gledati kao stog. Kada pospremamo tanjure stavit ćemo ih na vrh „tornja". Kada trebamo tanjur uzet ćemo ga s vrha.
- Još jedan poznati primjer stoga su dispenzeri poznatih PEZ®-bombona. Kada stavljamo bombone u dispenzer uvijek ih dodajemo na vrh, a kada ih vadimo skidamo ih s vrha.
- Povijest posjećenih stranica u preglednicima interneta može raditi po principu stoga. Kada posjetimo neku *web*-stranicu, prethodna stranica se dodaje na vrh stoga. Klikom na gumb za povratak, skida se stranica s vrha stoga i otvara u pregledniku.

Pogledajmo definiciju ATP-a Stog:

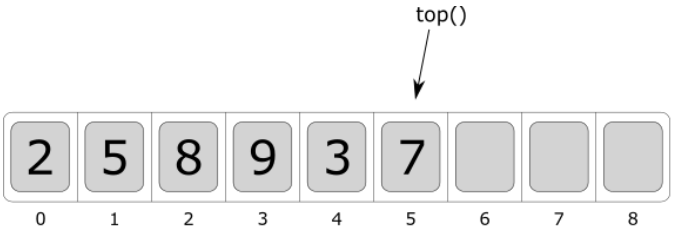
Stog

elementtype ... bilo koji tip
Stack ... konačni niz (ne nužno različitih) podataka tipa *elementtype*
make_null(S) ... pretvara stog S u prazan stog
empty(S) ... vraća *True* ako je stog prazan, inače vraća *False*
push(S, x) ... ubacuje element x na vrh stoga S
pop(S) ... ukloni i vrati element s vrha stoga S. Nije definirana ako je S prazan.
top(S) ... vrati element s vrha stoga S i pri tome ne mijenja stog. Nije definirana ako je S prazan.

Pošto je stog posebni slučaj liste, kao implementaciju stoga možemo koristiti bilo koju implementaciju liste. Ali zbog manjeg broja operacija moguće je optimizirati neke metode, zbog čega u nastavku dajemo dvije implementacije stoga.

2.4.1 Implementacija stoga pomoću polja

Struktura podataka koju koristimo za implementaciju stoga prikazana je na slici 2.10. Na toj slici vidimo da je vrh stoga na zadnjem elementu liste zbog čega se operacije *push* i *pop* svode na dodavanje i micanje elemenata s kraja liste. Implementaciju tih metoda možemo vidjeti na isječcima 2.7 i 2.8.



Slika 2.10: Polje korišteno kao stog

```
18      def push(S, x: elementtype) -> None:
19          if S.topp == MAXLENGTH - 1:
20              raise MemoryError('Full stack')
21          S.topp += 1
22          S.elements[S.topp] = x
```

Isječak 2.7: Implementacija metode *push*

```
24      def pop(S) -> elementtype:
25          if S.empty():
26              raise ValueError('Empty stack')
27          S.topp -= 1
28          return S.elements[S.topp + 1]
```

Isječak 2.8: Implementacija metode *pop*

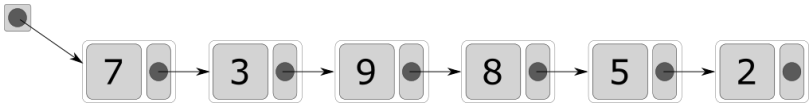
Iz prethodna dva isječka vidimo da je složenost metoda *push* i *pop* $\Theta(1)$. Metoda *top* svodi se na vraćanje zadnjeg elementa u polju, dok metoda *empty* vraća *True* ako i samo ako je „pozicija zadnjeg elementa" jednaka -1. Vidimo da je u ovoj implementaciji stoga složenost svih metoda $\Theta(1)$.

2.4.2 Implementacija stoga pomoću vezane liste

Sljedeća implementacija stoga koju ćemo pogledati je implementacija pomoću vezane liste. Kao i kod implementacije pomoću polja, možemo koristiti standardnu implementaciju liste. Ali i u ovom slučaju možemo postići bolju složenost ako uzmemo u obzir operacije koje ćemo obavljati nad ATP-om. Najbitnije je primijetiti dvije stvari:

- kod stoga se elementi uvijek dodaju i uklanjaju s iste strane
- dodavanje / uklanjanje elemenata na kraju vezane liste je skupa operacija

Iz tog tog razloga ćemo vrh stoga staviti na početak liste. Tada će stog sa slike 2.10 izgledati kao na slici 2.11 i izbjeći ćemo dodavanje / izbacivanje elemenata na kraju vezane liste.



Slika 2.11: Stog implementiran pomoću vezane liste

```
23      def push(S, x: elementtype) -> None:
24          S.topp = cell(element=x, nextp=S.topp)
```

Isječak 2.9: Implementacija metode *push*

```
26      def pop(S) -> elementtype:
27          if S.empty():
28              raise ValueError('Empty stack')
```

```
29          top = S.topp.element
30          S.topp = S.topp.nextp
31          return top
```

Isječak 2.10: Implementacija metode *pop*

Pogledamo li isječke 2.9 i 2.10 možemo vidjeti da su složenosti obje metode $\Theta(1)$.

2.4.3 Usporedba implementacija stoga pomoću polja i vezane liste

Složenosti svih metoda su $\Theta(1)$ kod obje implementacije. Možemo li na temelju toga zaključiti da su jednako dobre i da nije bitno koju koristimo? Ne. Ovisno o situaciji u kojoj ćemo koristiti stog preferirat ćemo jednu implementaciju.

Iako nema velike razlike u vremenskoj složenosti između implementacija, postoji razlika u količini korištene memorije.

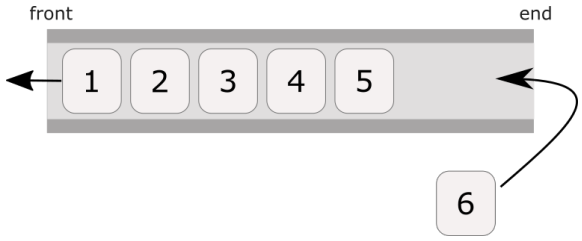
Implementacija pomoću polja je pogodnija ako neće biti velikih varijacija u duljini polja. Razlog tome je da je duljina polja fiksna te, u slučaju da trebamo više elemenata nego što stane u polje, moramo sve elemente prepisati u veće polje. Također, ako u jednom trenutku trebamo jako veliki stog, a nakon toga uklonimo većinu elemenata iz stoga, imat ćemo veliku količinu rezervirane a neiskorištene memorije.

S druge strane, kod implementacije pomoću vezane liste, za svaki element osim same vrijednosti imamo i pointer na sljedeći element. Zbog toga će ova implementacija trošiti više memorije po elementu stoga, ali nemamo ograničenja na ukupnu duljinu stoga (osim, naravno, memorije samog računala).

2.5 Red

U svakodnevnom životu *red* je poznat kao skupina ljudi koji stoje jedan iza drugoga i čekaju da budu posluženi. Uvijek se poslužuje prva osoba u redu, a ako nova osoba želi biti poslužena stat će na kraj reda, iza zadnje osobe.

U programiranju, red je vrsta liste u kojoj se ubacivanje i izbacivanje elemenata odvija po principu *first in, first out (FIFO)*: dodavanje elemenata je moguće u bilo kojem trenutku, ali je moguće ukloniti samo onaj element koji je najduže u redu. Obično kažemo da elemente dodajemo na kraj, a uklanjamo s početka reda.



Slika 2.12: Princip rada reda

Primjer 2.3. Pogledajmo sada nekoliko primjera reda iz svakodnevnog života:

- Najpoznatiji primjer je red za blagajnu. Novi kupci dolaze na kraj reda, a poslužuje se kupac na početku reda.
- Mrežni pisač će ispisati dokumente onim redoslijedom kojim ih je zaprimio. Novi dokumenti će se dodati na kraj reda.
- Server ima red upita koje je zaprimio od klijenata. Kada zaprimi novi upit, taj upit se stavlja na kraj reda. Server uvijek odgovara na prvi upit u redu, a nakon što odgovori upit se ukloni iz reda.

poziv metode	**povratna vrijednost**	**red**[1]
D.enqueue(2)	-	[2]
D.enqueue(4)	-	[2, 4]
D.dequeue()	2	[4]
D.empty()	*False*	[4]
D.dequeue()	4	[]
D.empty()	*True*	[]
D.dequeue()	error: „Empty queue”	[]
D.enqueue(7)	-	[7]
D.enqueue(5)	-	[7, 5]
D.enqueue(4)	-	[7, 5, 4]
D.dequeue()	7	[5, 4]

Tablica 2.2: Primjer rada reda

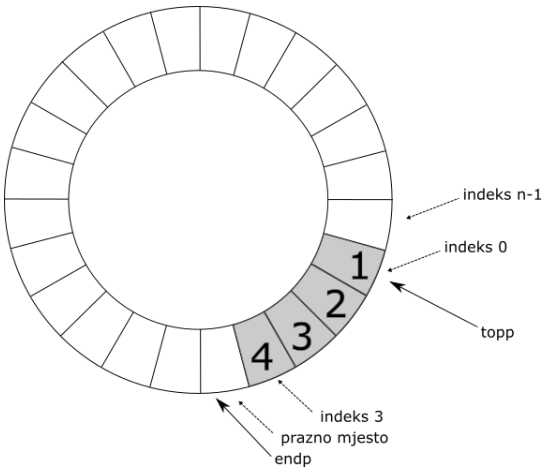
[1] Stanje reda nakon izvršavanja metode; početak reda nalazi se s lijeve, a kraj s desne strane.

Red

elementtype ... bilo koji tip
Queue ... konačni niz (ne nužno različitih) podataka tipa *elementtype*
make_null(Q) ... pretvara red *Q* u prazni red
empty(Q) ... vraća *True* ako je red *Q* prazan. Inače vraća *False*.
enqueue(x, Q) ... ubacuje element *x* na kraj reda *Q*
dequeue(Q) ... izbacuje i vraća element s početka reda *Q*
front(Q) ... vraća element s početka reda *Q*, pri tome ne mijenja *Q*

2.5.1 Implementacija reda pomoću cirkularnog polja

U ovom poglavlju koristimo varijantu polja, tzv. *cirkularno* polje. Riječ je o polju koje možemo prikazati kao na slici 2.13. Kako dodajemo i brišemo elemente, početak reda se pomiče.



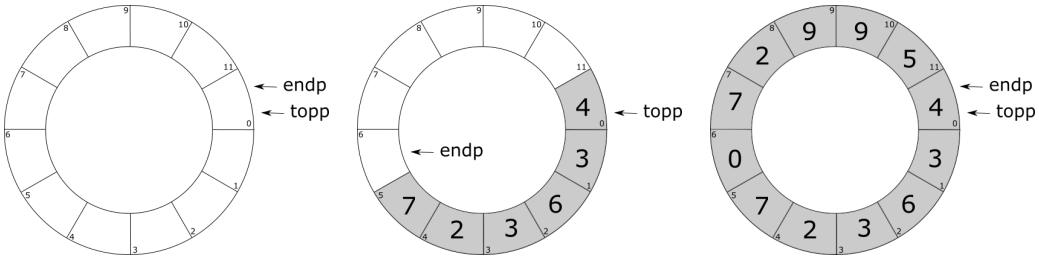
Slika 2.13: Cirkularno polje

```
7  class Queue:
8      elements: [elementtype]
9      position = int
10     endp: position
11     topp: position
```

Isječak 2.11: Atribut i varijable u klasi `Queue`

Kao što vidimo na isječku 2.11, `Queue` ima polje *elements* u koje spremamo elemente tipa *elementtype*, varijablu *topp* u kojoj pamtimo indeks prvog elementa i varijablu *endp* u kojoj pamtimo indeks nakon zadnjeg elementa.

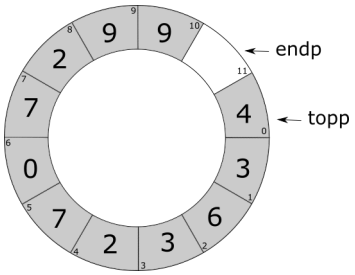
Kako kod implementacije koristimo cirkularno polje moramo posebnu pažnju posvetiti rubnim slučajevima, kada je red prazan i kada je pun. Ti slučajevi prikazani su na slici 2.14.



(a) Prazni red (b) Red s elementima (c) Puni red

Slika 2.14: Različita stanja reda

Pogledamo li pobliže slike 2.14a i 2.14c možemo primijetiti da u oba slučaja *topp* i *endp* imaju jednake vrijednosti. U tom slučaju ne možemo znati je li red pun ili prazan. Kako bismo izbjegli slučaj sa slike 2.14c, tj. razlikovali prazni od punog reda, dogovorit ćemo se da između zadnjeg i prvog elementa u redu uvijek mora biti barem jedno slobodno mjesto. Nakon tog dogovora puno polje će izgledati kao na slici 2.15.



Slika 2.15: Cirkularno polje

Sada možemo pogledati implementaciju metode *enqueue* u isječku 2.12. U liniji 21 provjeravamo je li indeks nakon *endp* jednak *topp*, odnosno je li red pun. Vidimo da je složenost metode $\Theta(1)$.

```
20      def enqueue(Q, x: elementtype) -> None:
21          if Q.topp == Q._add_one(Q.endp):
22              raise MemoryError('Full queue')
23          Q.elements[Q.endp] = x
24          Q.endp = Q._add_one(Q.endp)
```

Isječak 2.12: Implementacija metode *enqueue*

U liniji 21 možemo uočiti poziv metode *_add_one(position)* koja vraća sljedbenika modulo *MAXLENGTH*:

```
44      def _add_one(Q, p: position) -> position:
45          return (p + 1) % MAXLENGTH
```

Isječak 2.13: Implementacija privatne metode *_add_one*

Već smo vidjeli da je složenost metode *enqueue* $\Theta(1)$, no istu složenost imaju i *dequeue*, *front* i *empty*.

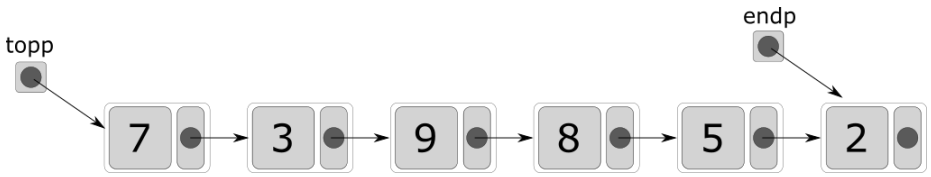
2.5.2 Implementacija reda pomoću jednostruko vezane liste

Kao i kod implementacije liste pomoću vezane liste, svaka ćelija osim vrijednosti elementa ima jednu referencu *nextp* na sljedeći element. Razlika u odnosu na listu je što, osim reference *topp* na početni element, imamo i referencu *endp* na zadnji element u redu (za razliku od implementacije pomoću cirkularnog polja).

```
11 class Queue:
12     topp: cell
13     endp: cell
```

Isječak 2.14: Varijable u klasi `Queue`

Kod implementacije stoga smo elemente dodavali na početak vezane liste. Kod implementacije reda ćemo elemente dodavati na kraj vezane liste, a brisati s početka vezane liste.

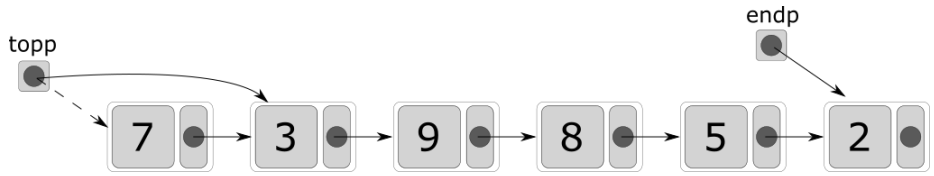


Slika 2.16: Implementacija reda pomoću vezane liste

Implementaciju metode *dequeue* možemo vidjeti u isječku 2.15. Kao što je ilustrirano na slici 2.17 metoda postavlja *topp* na vrijednost reference *nextp* starog *topp*. Kako kod svakog nepraznog reda imamo točno tri izvođenja osnovnih operacija možemo zaključiti da je složenost ove metode $\Theta(1)$.

```
30      def dequeue(Q) -> elementtype:
31          if Q.empty():
32              raise ValueError('Empty queue')
33          else:
34              temp = Q.topp.element
35              Q.topp = Q.topp.nextp
36              return temp
```

Isječak 2.15: Implementacija metode *dequeue*



Slika 2.17: Ilustracija rada metode *dequeue*

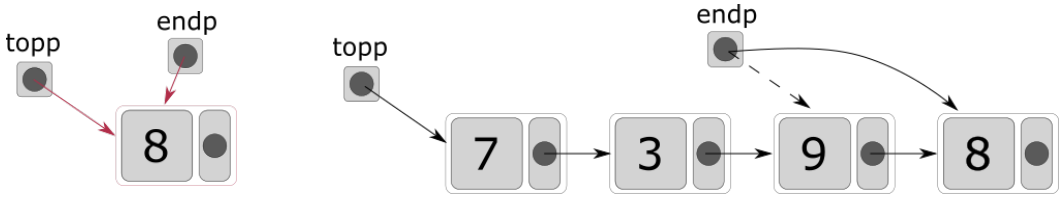
Metoda *enqueue* je nešto složenije jer ima različito ponašanje ovisno o tome dodajemo li novi element u prazni ili neprazni red.

```
22      def enqueue(Q, x: elementtype) -> None:
23          if Q.empty():
24              Q.endp = cell(element=x, nextp=None)
25              Q.topp = Q.endp
26          else:
27              Q.endp.nextp = cell(element=x, nextp=None)
28              Q.endp = Q.endp.nextp
```

Isječak 2.16: Implementacija metode *enqueue*

Ako imamo neprazni red, metoda stvara novu ćeliju te postavlja vrijednosti *topp* i *nextp* prvog elementa na tu ćeliju (vidi sliku 2.18b).
U slučaju da je red bio prazan, metoda stvara novu ćeliju te postavlja *topp* i *endp* na tu ćeliju.

Neovisno o slučaju, uvijek ćemo imati konstantni broj osnovnih operacija iz čega slijedi da je složenost metode $\Theta(1)$.



(a) red je bio prazan

(b) red nije bio prazan

Slika 2.18: Dodavanje elementa u red

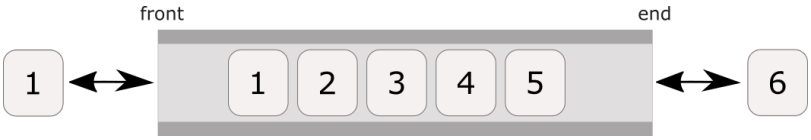
2.5.3 Usporedba implementacija reda

U obje navedene implementacije, sve metode imaju istu složenost $\Theta(1)$. Kao i kod svih implementacija pomoću polja, najveći nedostatak implementacije pomoću cirkularnog polja je ograničenje duljine koju zadajemo na početku.

Najveći nedostatak implementacije pomoću jednostruko vezane liste je trošenje dodatne memorije za pohranjivanje referenci *nextp*.

2.6 Dvostrani red (*deque*)

U ovom dijelu promatrat ćemo strukturu sličnu redu, koja omogućuje ubacivanje i izbacivanje elemenata na oba kraja, kao što je prikazano na slici 2.19.



Slika 2.19: Princip rada dvostranog reda

Dvostrani red je općenitiji od reda i stoga, što može biti korisno u nekim primjenama. Na primjer, opisujemo red čekanja u restoranu. Povremeno može doći do situacije kada će gosti biti odvedeni s početka reda, samo da bi se ustanovilo da nema slobodnih stolova; tada će se gosti vratiti na početak reda. Također, može se dogoditi da gosti s kraja reda odluče otići. (Da bismo opisali slučaj kada gosti iz sredine reda odlaze treba nam općenitiji ATP.)[2]

[2]Primjer je iz [2].

Mogući primjer poziva metoda možemo vidjeti u tablici 2.3.

poziv metode	**povratna vrijednost**	**red**[3]
D.enqueue_front(2)	-	[2]
D.enqueue_front(4)	-	[4, 2]
D.enqueue_back(3)	-	[4, 2, 3]
D.enqueue_front(1)	-	[1, 4, 2, 3]
D.dequeue_back()	3	[1, 4, 2]
D.dequeue_front()	1	[4, 2]
D.dequeue_back()	2	[4]
D.empty()	*False*	[4]
D.dequeue_back()	4	[]
D.empty()	*True*	[]
D.dequeue_back()	error: „Empty deque”	[]
D.enqueue_front(7)	-	[7]

Tablica 2.3: Primjer rada dvostranog reda

Dvostrani red

elementtype ... bilo koji tip
Deque ... konačni niz (ne nužno različitih) podatataka tipa *elementtype*
make_null(D) ... pretvara dvostrani red *D* u prazni dvostrani red
empty(D) ... vraća *True* ako je dvostrani red *D* prazan. Inače vraća *False*.
enqueue_front(x, D) ... ubacuje element *x* na početak dvostranog reda *D*
enqueue_back(x, D) ... ubacuje element *x* na kraj dvostranog reda *D*
dequeue_front(D) ... izbacuje i vraća element s početka dvostranog reda *D*
dequeue_back(D) ... izbacuje i vraća element s kraja dvostranog reda *D*
front(D) ... vraća element s početka dvostranog reda *D*, pri tome ne mijenja *D*
back(D) ... vraća element s kraja dvostranog reda *D*, pri tome ne mijenja *D*

2.6.1 Implementacija dvostranog reda pomoću cirkularnog polja

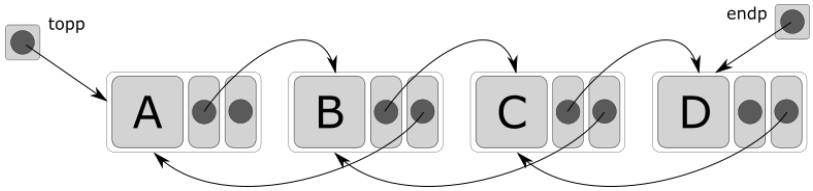
Kod implementacije dvostranog reda koristimo istu strukturu podataka kao kod implementacije reda pomoću cirkularnog polja. Same implementacije su slične, samo što osim *enqueue_back* i *dequeue_front* (ekvivalenti metodama *enqueue* i *dequeue* kod reda) imamo i metode *enqueue_front* i *dequeue_back*.

[3] Stanje dvostranog reda nakon izvršavanja metode; početak reda nalazi se s lijeve, a kraj s desne strane.

2.6.2 Implementacija dvostranog reda pomoću dvostruko vezane liste

Kod reda smo uspješno izbjegli prolaženje po svim elementima vezane liste, no kod dvostranog reda moramo omogućiti brisanje elemenata s oba kraja. Zato kod implementacije ovog ATP-a koristimo dvostruko vezanu listu umjesto jednostruko vezane.

U dvostruko vezanoj listi svaki čvor osim reference *nextp* ima i referencu *previousp* kao što je ilustrirano na slici 2.4b. Referenca *nextp* pokazuje na sljedeći čvor, a *previousp* na prethodni. Ilustraciju možemo vidjeti na slici 2.20.



Slika 2.20: Ilustracija implementacije dvostranog reda

Ovdje smo, kako bismo imali bolju vremensku složenost, odlučili utrošiti dodatnu memoriju za dvije reference po elementu. Od dodatnih referenci najviše profitiraju metode *dequeue_front* i *dequeue_back*. Da nemamo dodatnu referencu, jedna od tih metoda (ovisno kako postavimo kraj dvostranog reda) bi imala složenost $\Theta(n)$. Dodavanjem reference *previousp* uspjeli smo postići da obje metode imaju složenost $\Theta(1)$.

2.6.3 Usporedba implementacija dvostranog reda

U obje implementacije sve metode imaju iste vremenske složenosti $\Theta(1)$. Kao i prije, implementacija pomoću cirkularnog polja ima nedostatak da je kapacitet reda ograničen s *MAXLENGTH*, dok implementacija pomoću dvostruko vezane liste nema ograničen broj elemenata, ali zauzima više memorije po elementu.

Poglavlje 3

Stabla

Do sada smo upoznali nekoliko linearnih struktura podataka kao što su liste, stogovi i redovi. U ovom poglavlju uvodimo prvu nelinearnu strukturu podataka, *stabla*. Riječ je o strukturi koja podatke organizira u hijerarhiju. Sama organizacija podataka slična je onoj u obiteljskom stablu, s pojmovima poput: „dijete", „roditelj", „predak", ...

Podaci su spremljeni u čvorove, a parovi čvorova su povezani usmjerenim bridovima. Bridovi predstavljaju odnose među čvorovima.

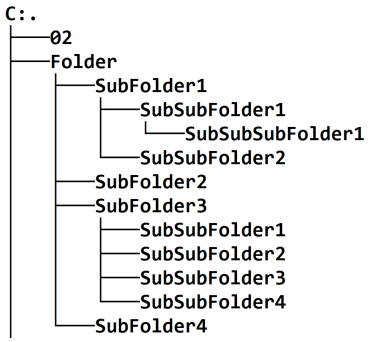
Formalno, stablo definiramo na sljedeći način.

Definicija 3.1. *Stablo* je neprazni skup podataka istog tipa, koje zovemo *čvorovi*. Skup čvorova zadovoljava ova dva uvjeta:

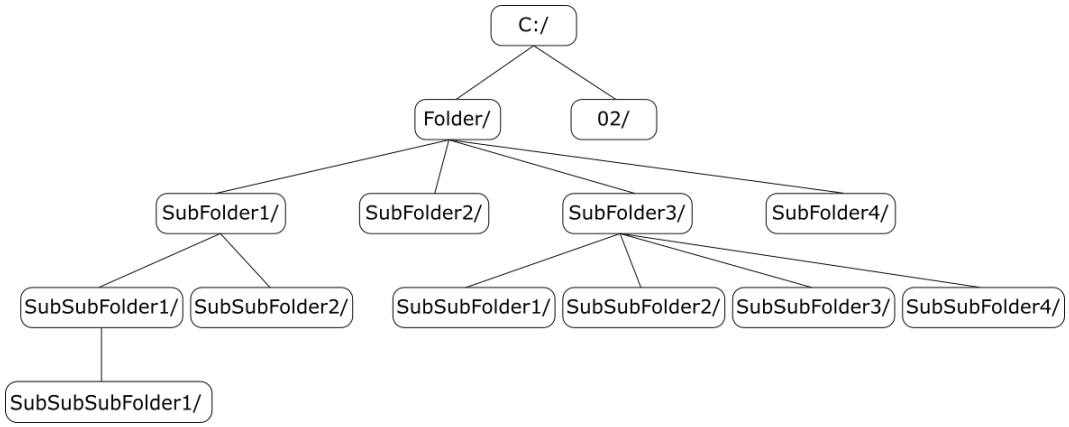
- postoji istaknuti čvor r, kojeg nazivamo *korijen*;
- ostali čvorovi grade konačni niz $(T_1, T_2, \ldots, T_n)$ od nula, jednog ili više disjunktnih stabala, koja nazivamo *podstabla korijena r*

Primijetimo da je definicija stabala rekurzivna jer se stablo T definira pomoću manjih stabala $T_1, T_2, \ldots, T_n$.

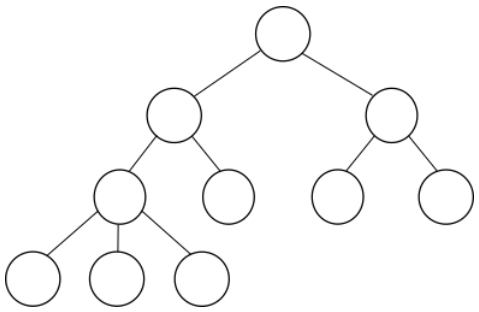
Jedan primjer strukture stabla je reprezentacija nad-mapa i pod-mapa u sustavu.



Slika 3.1: Primjer stabla u sustavu direktorija



Slika 3.2: Prikaz stabla sa slike 3.1



Slika 3.3: Primjer stabla bez oznaka

Definirajmo sada pojmove vezane uz stabla.

Definicija 3.2.

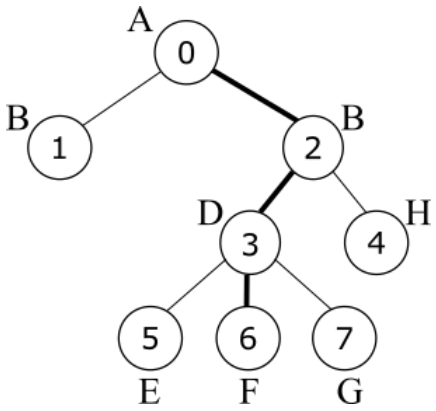
- *Korijen r* je najviši čvor stabla, odnosno jedini čvor u stablu koji nema roditelja.
- Korijeni $r_1, r_2, \ldots, r_n$ od $T_1, T_2, \ldots, T_n$ su *djeca* čvora r, a čvor r je njihov *roditelj*.
- *Put od* i_1 *do* i_m je niz čvorova $i_1, i_2, i_3, \ldots, i_m$ takvih da je i_p roditelj od i_{p+1} za sve $p \in \{1, \ldots, m-1\}$. *Duljina* tog puta je $m-1$. Ako postoji put od čvora i do čvora j kažemo da je čvor i *predak* čvora j, a čvor j *potomak* čvora i.
- *Braća* su djeca istog roditelja.
- *Razina* ili *nivo s* je skup svih čvorova sa svojstvom da je put od korijena do tog čvora duljine s. Korijen se nalazi na razini 0, njegova djeca na razini 1, njihova djeca na razini 2, itd.
- Ako čvor ima djecu zovemo ga *unutrašnji čvor*, a inače *list*.
- *Visina stabla* je redni broj posljednje neprazne razine.

Definicija 3.3. *Označeno stablo* je stablo kod kojega je svakome čvoru pridružen dodatni podatak koji nazivamo *oznaka*. Pri tome razlikujemo ime čvora od njegove oznake. Ime čvora je jedinstveno i služi za identifikaciju, dok oznaka čvora ne mora biti jedinstvena: moguće je u jednom stablu imati više čvorova s istom oznakom. Imena čvorova imaju analognu ulogu kao pozicije u listi, a oznake kao elementi liste.

U nastavku ćemo ime čvora upisivati unutar kružića, a njegovu oznaku pored njega. Na slici 3.4 vidljiv je taj način označavanja stabla. Korijen stabla ima ime 0 i oznaku „A". Čvorovi s imenima 1 i 2 imaju oznaku „B", čvor s imenom 3 ima oznaku „D" itd.

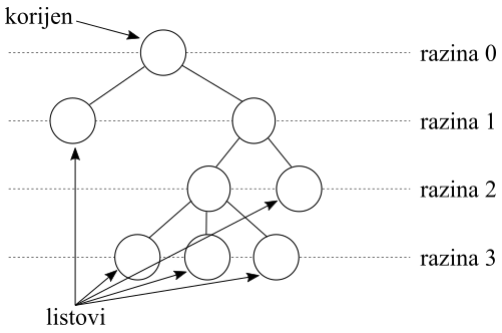
Čvor 2 je roditelj čvorovima 3 i 4, a čvorovi 3 i 4 su djeca čvora 2.

Vidimo da postoji put (označen podebljanom crtom) od čvora 0 do čvora 6 pa možemo zaključiti da je čvor 0 predak čvora 6, a čvor 6 potomak čvora 0.



Slika 3.4: Primjer označenog stabla

Pogledajmo sada ostale pojmove vezane uz stabla na primjeru stabla sa slike 3.5. Vidimo da stablo ima 4 razine s čvorovima, pa je njegova visina 3. Nadalje, to stablo ima 5 listova.



Slika 3.5: Još neki pojmovi vezani uz stablo

U daljnjem tekstu ćemo, ako nije drugačije naznačeno, pod stablom podrazumijevati označeno stablo.

3.1 Opća stabla

U ovom poglavlju pogledajmo ATP Stablo.

Stablo

Node ... bilo koji tip podataka. Tip *Node* koristimo za imena čvorova. Nul-referencu `None` koristimo kao ime nepostojećeg čvora.
labeltype ... bilo koji tip podataka koji koristimo za oznake čvorova
Tree ... uređeno stablo čiji čvorovi su podaci tipa *Node* (međusobno različiti i različiti od `None`) kojima su pridružene oznake tipa *labeltype*
make_root(T, label) ... pretvara stablo *T* u stablo koje se sastoji samo od korijena s oznakom *label*. Vraća čvor koji predstavlja korijen stabla *T*.
insert_child(T, label, node) ... ubacuje čvor s oznakom *label* u stablo *T* tako da bude prvo dijete čvora *node*. Vraća čvor koji je ubačen (njegovo ime).
insert_sibling(T, label, node) ... ubacuje čvor s oznakom *label* u stablo *T* tako da bude idući po redu brat čvora *node*. Vraća čvor koji je ubačen (njegovo ime).
delete(T, node) ... izbacuje list s imenom *node* iz stabla *T*. Nije definirana ako *node* nije list ili je *node* korijen.
root(T) ... vraća korijen stabla *T*
first_child(T, node) ... vraća prvo dijete (njegovo ime) čvora *node*. Ako je *node* list (nema djece) vraća `None`.
next_sibling(T, node) ... vraća sljedećeg brata čvora *node*. Ako čvor *node* nema sljedećeg brata, vraća `None`.
parent(T, node) ... vraća roditelja čvora *node*. Ako je *node* korijen stabla, vraća `None`.
change_label(T, label, node) ... postavlja oznaku čvora *node* na *label*
label(T, node) ... vraća oznaku čvora *node*
Metode *insert_child*, *insert_sibling*, *delete*, *first_child*, *next_sibling*, *parent*, *change_label* i *label* nisu definirane ako čvor *node* ne pripada stablu *T*.

U zadnjem dijelu sljedećeg poglavlja pogledat ćemo implementaciju stabla, ali prije toga pogledajmo nekoliko načina na koje možemo proći po svim čvorovima stabla.

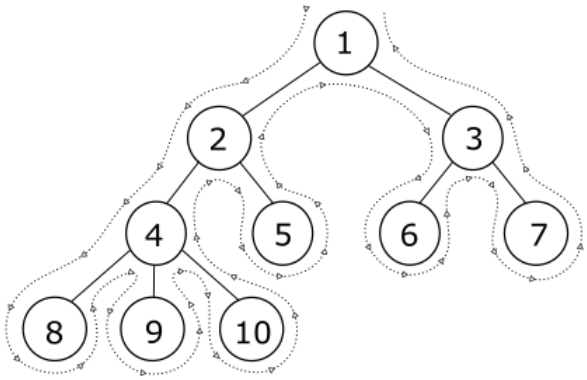
3.1.1 Obilasci stabla

Obilazak stabla T je sustavni način pristupanja svim čvorovima („posjećivanja" svih čvorova) stabla *T*, tako da svaki čvor posjetimo točno jednom. Pod *posjećivanjem čvora* podrazumijevat ćemo bilo koju radnju nad tim čvorom: čitanje oznake, mijenjanje oznake ili neku

složeniju operaciju. Ukupno ćemo pogledati četiri najpoznatija algoritma za obilazak stabla: *Preorder*, *Inorder*, *Postorder* i *pretraživanje u širinu*.

Kako je definicija stabla rekurzivna, prirodno je za očekivati da će i obilasci biti rekurzivni. Svaki od obilazaka ima svoje prednosti u određenim primjenama te će i o njima biti riječi u tekstu.

Kao primjer koristit ćemo stablo sa slike 3.6. Vidimo da ono ima 10 čvorova.

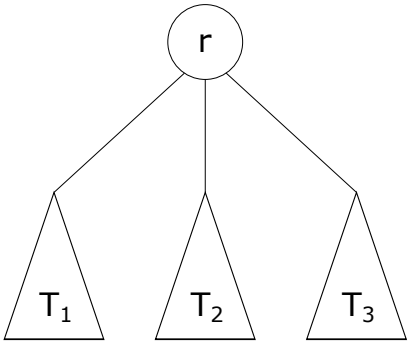


Slika 3.6: Primjer stabla na kojem ćemo proučavati obilaske

3.1.1.1 Preorder

Obilazak počinje posjetom korijenu, jer ga smatramo „ulazom" u stablo. Nakon njega obilazimo njegovo prvo podstablo, zatim drugo, sve dok ne dođemo do zadnjeg podstabla. Kad obiđemo zadnje podstablo obilazak je završen.

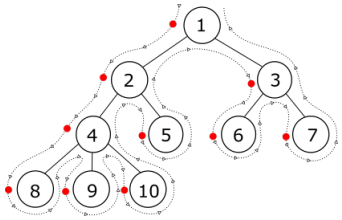
Pogledamo li stablo sa slike 3.7, obilazak bi išao sljedećim redosljedom: prvo bismo posjetili korijen r, a zatim bismo rekurzivno primjenili obilazak *Preorder* na stablo T_1, zatim na stablo T_2 i na kraju na stablo T_3. Osnovni slučaj rekurzije nastupa kada dođemo do lista.



Slika 3.7: Stablo

Konkretnije, ako pogledamo stablo sa slike 3.6, čvorovi bi bili posjećeni u sljedećem redoslijedu: 1, 2, 4, 8, 9, 10, 5, 3, 6, 7.

Obilazak možemo vizualizirati pomoću crtkane linije na slici. Linija kreće s lijeve strane korijena i obilazi oko svih čvorova u stablu. Čvor *i* ćemo posjetiti kada se, krećući se po crtkanoj liniji, nađemo na lijevoj strani čvora *i* (vidi sliku 3.8).



Slika 3.8: Vizualizacija obilaska Preorder

Algoritam možemo jednostavno realizirati kao rekurzivnu funkciju koja kao argumente prima stablo *T* i njegov čvor (podrazumijevano korijen) *node*.

```
1 def Preorder(T, node: Node = None):
2     if node is None:
3         node = T.root()
4     print(T.label(node), end=' ')
```

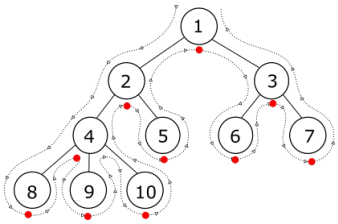
```
5     child = T.first_child(node)
6     while child:
7         Preorder(T, child)
8         child = T.next_sibling(child)
```

Isječak 3.1: Preorder

3.1.1.2 Inorder

Sljedeći po redu obilazak je *Inorder*. Kao i Preorder, i on se definira rekurzivno: prvo korištenjem obilaska Inorder obiđemo prvo podstablo, zatim posjetimo korijen, te obilazimo drugo podstablo, treće podstablo, . . . , sve dok ne završimo s obilaskom zadnjeg podstabla. I ovdje je osnovni slučaj rekurzije slučaj kada je korijen podstabla koje obilazimo list.

Kod obilaska stabla sa slike 3.6 to znači da čvor posjetimo svaki puta kada se, krećući se po isprekidanoj crti, prvi puta nađemo s donje strane čvora (slika 3.9). Čvorovi stabla bit će posjećeni u sljedećem redoslijedu : 8, 4, 9, 10, 2, 5, 1, 6, 3, 7.



Slika 3.9: Vizualizacija obilaska Inorder

Implementacija algoritma vidljiva je u isječku 3.2.

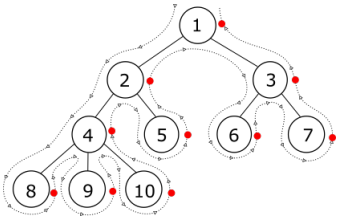
```
1 def Inorder(T, node: Node = None):
2     if node is None:
3         node = T.root()
4     if T._is_leaf(node):
5         print(T.label(node), end=' ')
6     else:
7         child = T.first_child(node)
8         Inorder(T, child)
9         print(T.label(node), end=' ')
10        child = T.next_sibling(child)
```

```
11          while child:
12              Inorder(T, child)
13              child = T.next_sibling(child)
```

Isječak 3.2: Inorder

3.1.1.3 Postorder

Obilazak Postorder prvo obilazi sva podstabla korijena, a zatim posjećuje korijen. Ako se krećemo oko stabla to znači da čvor posjetimo kada se nađemo s njegove desne strane. Čvorovi sa stabla na slici 3.10 bit će posjećeni u redoslijedu: 8, 9, 10, 4, 5, 2, 6, 7, 3, 1.



Slika 3.10: Vizualizacija obilaska Postorder

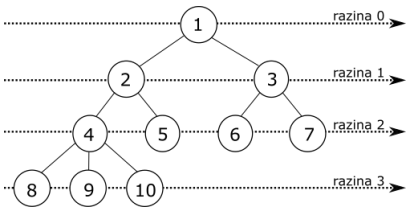
I ovaj obilazak lako je implementirati kao funkciju koja kao argument prima stablo. Implementacija je vidljiva na isječku 3.3.

```
1 def Postorder(T, node: Node = None):
2     if node is None:
3         node = T.root()
4     child = T.first_child(node)
5     while child:
6         Postorder(T, child)
7         child = T.next_sibling(child)
8     print(T.label(node), end=' ')
```

Isječak 3.3: Postorder

3.1.1.4 Pretraživanje u širinu (BFS)

Ovaj obilazak obilazi čvorove razinu po razinu: prvo razinu 0 (korijen), zatim razinu 1, itd. Čvorovi sa slike 3.11 bit će posjećeni u sljedećem redoslijedu: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.



Slika 3.11: Pretraživanje u širinu

Jedna moguća implementacija algoritma dana je sljedećom funkcijom:

```
def BFS(T):
    queue = Queue.make_null()
    node = T.root()
    while node:
        print(T.label(node), end=' ')
        for child in children(T, node):
            queue.enqueue(child)
        if queue.empty():
            return
        node = queue.dequeue()

def children(T: Tree, node: Node) -> Generator[Node]:
    child = T.first_child(node)
    while child:
        yield child
        child = T.next_sibling(child)
```

Isječak 3.4: Pretraživanje u širinu

3.1.2 Implementacija stabla pomoću pointera

U ovom poglavlju pogledat ćemo implementaciju koja se temelji na korištenju pointera.

```
@dataclass
class Node(List):
    parentp: 'Node'
    children: 'Node'
    previousp: 'Node'
```

Isječak 3.5: Implementacija klase `Node`

Pogledamo li isječak 3.5 možemo vidjeti implementaciju klase `Node` koja predstavlja čvor u stablu. Svaki čvor ima referencu na roditelja *parentp*, listu svoje djece *children* i referencu na prethodni element *previousp* (u listi). Također nasljeđuje klasu `List`. Koristimo implementaciju liste pomoću pointera iz poglavlja 2.3.

Nakon što smo vidjeli implementaciju klase `Node`, u isječku 3.6 vidimo da klasa `Tree` ima jednu varijablu *rootp* tipa `Node` koja je referenca na korijen.

```
14 class Tree:
15     rootp: Node
```

Isječak 3.6: Varijabla u klasi `Tree`

Metode *make_root*, *root*, *change_label*, *label* i *parent* su jednostavne te ćemo njihovu implementaciju preskočiti. Samo ćemo napomenuti da su im složenosti $\Theta(1)$. Metoda *insert_child* se svodi na dodavanje novog elementa na početak liste *children* i dodavanje atributa *parentp*, *children* i *previousp*, a metoda *first_child* vraća prvi element iz te liste te je i njihova složenost $\Theta(1)$.

```
22     def insert_child(T, x: labeltype, parent: Node) -> Node:
23         parent.children.insert(x, parent.children.first())
24         p = parent.children.first().nextp
25         p.parentp = parent
26         p.children = List.make_null()
27         p.previousp = parent.children.first()
28         return p
```

Isječak 3.7: Metoda *insert_child*

```
30     def delete(T, node: Node):
31         if not T._is_leaf(node):
32             raise ValueError('Can only delete leafs')
33         if node == T.root():
34             raise ValueError("Can't delete root")
35         parent = node.parentp
36         temp = node.nextp
37         position = node.previousp
38         parent.children.delete(node.previousp)
39         if temp:
40             temp.previousp = position
```

Isječak 3.8: Metoda *delete*

U isječku 3.8 vidimo da metoda *delete* briše element *node* iz liste djece roditelja od *node* te možemo zaključiti da je složenost ove metode $\Theta(1)$.

```
42     def insert_sibling(T, x: labeltype, node: Node) -> Node:
43         if node == T.root():
44             raise ValueError("Can't add sibling to root")
45         parent = node.parentp
46         old_sibling = node.nextp
47         parent.children.insert(x, node)
48         node.nextp.parentp = node.parentp
49         node.nextp.children = List.make_null()
50         node.nextp.previousp = node
51         if old_sibling:
52             old_sibling.previousp = node.nextp
53         return node.nextp
```

Isječak 3.9: Metoda *insert_sibling*

U isječku 3.9 vidimo implementaciju metode *insert_sibling*. Vidimo da metoda dodaje element s vrijednošću *x* u listu *parentp.children* te nakon toga postavlja dodatne atribute za čvor u stablu. Kako koristimo implementaciju ATP-a *Lista* pomoću vezane liste, složenost metode *inset_sibling* je $\Theta(1)$.

```
63     def next_sibling(T, node: Node) -> Node:
64         if node == T.root():
65             return None
66         return node.nextp
```

Isječak 3.10: Metoda *next_sibling*

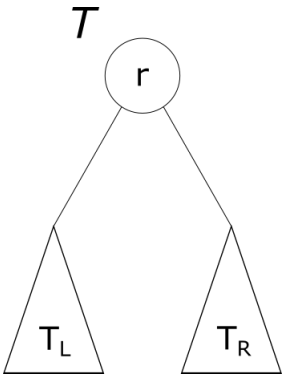
3.2 Binarno stablo

U ovom dijelu promatramo ATP sličan stablu, *binarno stablo*. Riječ je o ATP-u kod kojeg svaki čvor ima najviše dvoje djece. Formalnije binarno stablo možemo definirati sljedećom definicijom.

Definicija 3.4. *Binarno stablo T* je konačan skup podataka istog tipa koje zovemo *čvorovi*. Pri tome vrijedi:

- T je prazan skup, ili
- postoji istaknuti čvor r koji se zove *korijen* od T, a ostali čvorovi grade uređeni par (T_L, T_R) disjunktnih (manjih) binarnih stabala.

Definicija se može ilustrirati slikom 3.12.



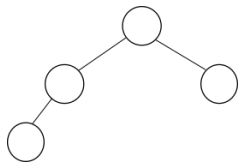
Slika 3.12: Neprazno binarno stablo T

Definicija binarnog stabla je, kao i definicija stabla, rekurzivna. Korijen podstabla T_L (ako postoji) nazivamo *lijevo dijete*, korijen podstabla T_R (ako postoji) *desno dijete*, a čvor r je njihov roditelj.

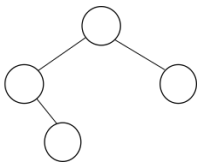
Primijetimo razliku u odnosu na definiciju stabla:

- binarno stablo može biti prazno;
- razlikujemo situaciju kada je jedino dijete čvora lijevo ili desno (slika 3.13).

Na slici 3.13 su dva prikaza istog stabla, ali dva različita binarna stabla.



(a) Čvor ima lijevo dijete



(b) Čvor ima desno dijete

Slika 3.13: Dva različita binarna stabla

Binarno stablo

Node ... bilo koji tip podataka. Tip *Node* koristimo za imena čvorova. Nul-referencu `None` koristimo kao ime nepostojećeg čvora.
labeltype ... bilo koji tip podataka koji koristimo za oznake čvorova
BinaryTree ... binarno stablo čijim čvorovima su pridružene oznake tipa *labeltype*
make_null(T) ... stvara prazno binarno stablo T
empty(T , x) ... vraća *True* ako je binarno stablo T prazno. Inače vraća *False*.
create(x, TL, TR) ... stvara binarno stablo T koje ima korijen s oznakom x te lijevo podstablo TL i desno podstablo TR
left_subtree(T, node) ... vraća lijevo podstablo čvora *node* u stablu T
right_subtree(T, node) ... vraća desno podstablo čvora *node* u stablu T
insert_left_child(T, node, x) ... čvoru *node* stabla T dodaje lijevo dijete s oznakom x i vraća to dijete. Nije definirana ako *node* već ima lijevo dijete.
insert_right_child(T, node, x) ... čvoru *node* stabla T dodaje desno dijete s oznakom x i vraća to dijete. Nije definirana ako *node* već ima desno dijete.
delete(T, node) ... uklanja list *node* iz stabla T
change_label(T, x, node) ... postavlja oznaku čvora *node* na x
label(T, node) ... vraća oznaku čvora *node*
Metode *left_subtree*, *right_subtree*, *insert_left_child*, *insert_right_child*, *delete*, *change_label* i *label* nisu definirane ako čvor *node* ne pripada stablu T.

3.2.1 Implementacija binarnog stabla pomoću pointera

Kod implementacije binarnog stabla pomoću pointera imamo, kao i kod stabala, klasu `Node` koja reprezentira čvor binarnog stabla. Kao što vidimo u isječku 3.11 svaki čvor ima referencu na roditelja *parentp*, reference na djecu *left_childp* i *right_childp* te *label* u kojem čuvamo oznaku čvora.

```
7 class Node:
8     parentp:'Node'
9     left_childp: 'Node'
10    right_childp: 'Node'
11    label: labeltype
```

Isječak 3.11: Klasa Node

Ako želimo stvoriti novo binarno stablo, to možemo napraviti korištenjem metode *make_null* ili korištenjem metode *create*.

Metoda *left_subtree* (*right_subtree*), kao što vidimo na isječku 3.12, vraća binarno stablo kojemu je korijen lijevo (desno) dijete od *node*.

Metoda *insert_left_child(T, node, x)* (*insert_right_child(T, node, x)*) dodaje lijevo (desno) dijete čvoru *node*, ako *node* nema lijevo (desno) dijete.

```
36    def left_subtree(T, node: Node) -> 'BinaryTree':
37        return BinaryTree(rootp=node.left_childp)
```

Isječak 3.12: Metoda *left_subtree*

Složenost svih metoda je $\Theta(1)$.

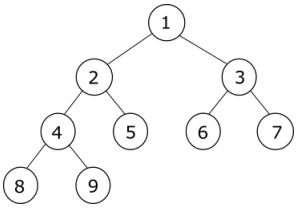
3.2.2 Posebna binarna stabla

Posebna vrsta binarnog stabla je *potpuno binarno stablo*, koje se može definirati na sljedeći način. (Definicija je preuzeta iz [3].)

Definicija 3.5. *Potpuno binarno stablo* građeno je od n čvorova s imenima $0, 1, 2, \ldots, n-1$. Pritom vrijedi:

- lijevo dijete čvora i je čvor $2i + 1$ (ako je $2i + 1 \geq n$ tada čvor i nema lijevo dijete);
- desno dijete čvora i je čvor $2i + 2$ (ako je $2i + 2 \geq n$ tada čvor i nema desno dijete).

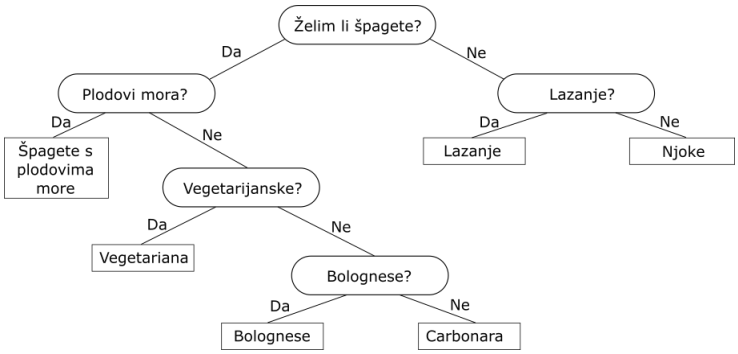
Neformalno, potpuno binarno stablo možemo definirati kao binarno stablo kojemu su sve razine osim zadnje popunjene, a svi čvorovi na zadnjoj razini su pomaknuti sasvim ulijevo.



Slika 3.14: Potpuno binarno stablo

Primjer 3.1. Jedan bitni primjer korištenja binarnih stabala su *stabla odlučivanja*. Ona se koriste kada želimo prikazati niz mogućih ishoda kod odgovaranja na niz da/ne pitanja. Svakom unutarnjem čvoru pridruženo je pitanje, a svakom listu je pridružen jedan mogući krajnji ishod. Lijevo podstablo predstavlja slučaj kada odgovorimo „da", a desno kad odgovorimo „ne".

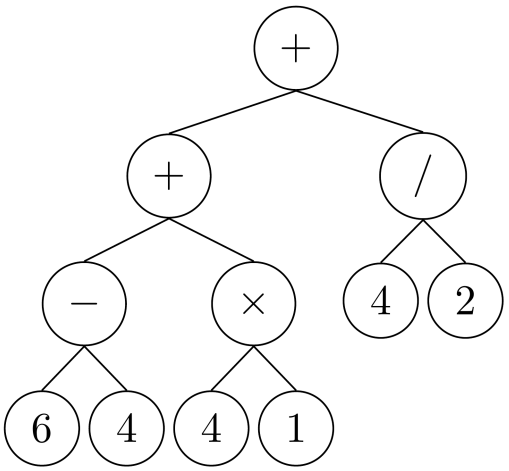
Slika 3.15 pokazuje jedno takvo stablo odlučivanja kod procesa naručivanja hrane iz lokalnog restorana.



Slika 3.15: Stablo odlučivanja

Primjer 3.2. Aritmetički izrazi (s binarnim operacijama) također se mogu prikazati kao binarna stabla. Na slici 3.16 prikazano je jedno takvo stablo, koje prikazuje izraz

$$((6 - 4) + (4 \times 1)) + (4 \div 2).$$



Slika 3.16: Binarno stablo koje prikazuje aritmetički izraz

Još jedna važna vrsta binarnog stabla je *hrpa (heap)* koja se definira na sljedeći način.

Definicija 3.6. Potpuno binarno stablo T je *hrpa*, ako su ispunjeni sljedeći uvjeti:

- Čvorovi od T su označeni podacima nekog tipa na kojem je definiran totalni uređaj.
- Neka je i bilo koji čvor od T. Tada je oznaka od i manja ili jednaka oznaci bilo kojeg djeteta od i.

Iz definicije slijedi da je korijen čvor s najmanjom oznakom. To svojstvo će nam biti korisno kod implementacije prioritetnog reda pomoću hrpe.

3.2.3 Implementacija hrpe pomoću polja

Prije same implementacije pogledajmo definiciju ATP-a Hrpa.

Hrpa

elementtype ... bilo koji tip podataka
Heap ... hrpa čijim čvorovima su pridružene oznake tipa *elementtype*
make_null(H) ... stvara praznu hrpu H
insert(H, x) ... dodaje čvor s oznakom x u hrpu H
del_min(H) ... uklanja čvor s najmanjom vrijednosti iz hrpe H i vraća ga
empty(H) ... vraća *True* ako je hrpa H prazna, inače vraća *False*

Kao i kod ostalih implementacija pomoću polja, i ovdje imamo varijablu *MAXLENGTH* kojom određujemo duljinu polja koje koristimo. Kod implementacije koristimo svojstvo potpunog binarnog stabla iz definicije 3.5.

```
@dataclass
class Heap:
    elements: [elementtype]
    _count: int
```

Isječak 3.13: Varijable u klasi `Heap`

Na isječku 3.13 vidimo da klasa `Heap` ima dvije članske varijable: polje *elements* u koje spremamo elemente hrpe, i varijablu *_count* koja odgovara broju elemenata u hrpi.

Prva metoda koju ćemo pobliže pogledati je *insert* koju možemo vidjeti na isječku 3.14. Metoda dodaje element *x* u polje *elements* te poziva pomoćnu metodu *_propagate_up* koja prebacuje novo dodani element na pravu poziciju.

```
    def insert(H, x: elementtype):
        if H._count == MAXLENGTH:
            raise ValueError('Full heap')
        H._count += 1
        H.elements[H._count - 1] = x
        H._propagate_up(H._count - 1)
```

Isječak 3.14: Metoda *insert*

Pogledajmo sada pobliže metodu *_propagate_up* na isječku 3.15.

```
    def _propagate_up(H, i):
        while i > 0 and H.elements[i] < H.elements[_up(i)]:
            temp = H.elements[_up(i)]
            H.elements[_up(i)] = H.elements[i]
            H.elements[i] = temp
            i = _up(i)
```

Isječak 3.15: Metoda *_propagateUp*

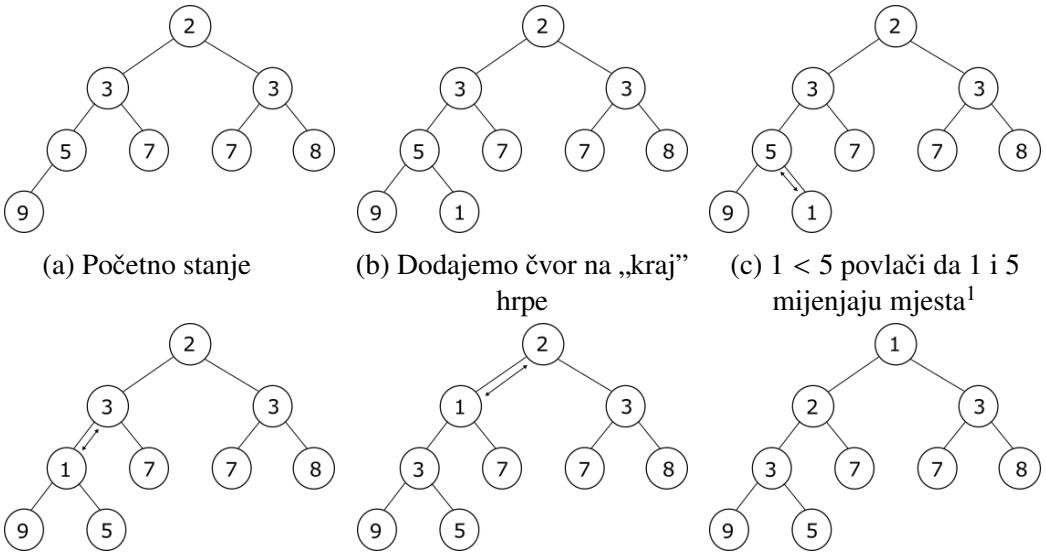
```
    def _up():
        return (i-1)//2
```

Isječak 3.16: Pomoćna metoda *_up*

Metoda *_propagate_up(H,i)* prima dva parametra: hrpu *H* i indeks *i* elementa koji propagira. Sve dok vrijedi

- indeks je veći od 0 (čvor nije korijen), i
- čvor je manji od svog roditelja,

zamijeni čvor i roditelja. Cijeli proces ubacivanja novog elementa prikazan je na slici 3.17.



(a) Početno stanje

(b) Dodajemo čvor na „kraj" hrpe

(c) 1 < 5 povlači da 1 i 5 mijenjaju mjesta[1]

(d) 1 < 3 povlači da 1 i 3 mijenjaju mjesta

(e) 1 < 2 povlači da 1 i 2 mijenjaju mjesta

(f) Propagiranje je završilo (došli smo do korijena)

Slika 3.17: Vizualizacija procesa ubacivanja elementa u hrpu

Osim *insert* i *make_null*, `Heap` ima i metodu *del_min*. Metoda prebacuje element s „kraja" hrpe u korijen te ga nakon toga pomoću pomoćne metode *_propagate_down* propagira prema dolje. Metoda *_propagate_down* zamijeni čvor s manjim djetetom, ako je veći od manjeg djeteta.

```
22      def del_min(H):
23          if H._count == 0:
24              raise ValueError('Empty heap')
25          minval = H.elements[0]
26          H.elements[0] = H.elements[H._count - 1]
27          H._count = H._count - 1
```

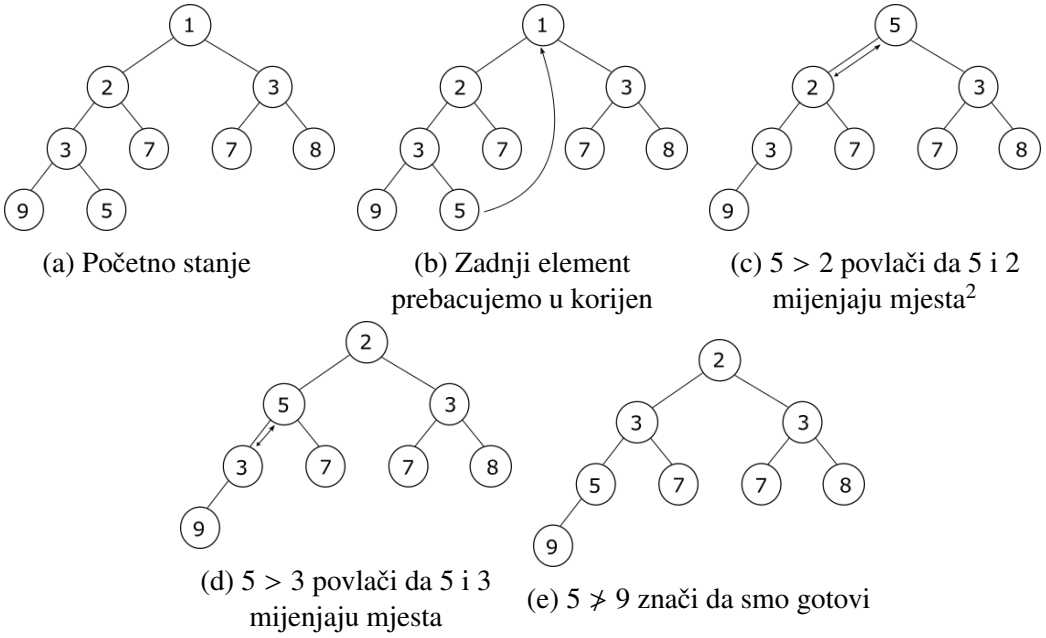
[1] započinje propagiranje novog elementa prema gore

```
28          H.elements[H._count] = None
29          H._propagate_down(0)
30          return minval
```

Isječak 3.17: Metoda *del_min*

```
39      def _propagate_down(H, i):
40          while 2*i + 2 <= H._count:
41              mc = H._min_child_index(i)
42              if H.elements[i] > H.elements[mc]:
43                  tmp = H.elements[i]
44                  H.elements[i] = H.elements[mc]
45                  H.elements[mc] = tmp
46              if i == mc:
47                  break
48              i = mc
49
50      def _min_child_index(H, i):
51          if 2*i + 2 >= H._count:
52              return 2*i + 1
53          elif H.elements[2*i + 1] < H.elements[2*i + 2]:
54              return 2*i + 1
55          else:
56              return 2*i +2
```

Isječak 3.18: Metode *propagate_down* i *_min_child_index*



(a) Početno stanje

(b) Zadnji element prebacujemo u korijen

(c) $5 > 2$ povlači da 5 i 2 mijenjaju mjesta[2]

(d) $5 > 3$ povlači da 5 i 3 mijenjaju mjesta

(e) $5 \not> 9$ znači da smo gotovi

Slika 3.18: Vizualizacija procesa uklanjanja najmanjeg elementa u hrpi

Nakon što smo pogledali implementacije metoda, možemo proučiti njihovu složenost. Metoda *make_null* ima složenost $\Theta(1)$.
Neka je n broj elemenata u hrpi. Metoda *insert* pri svakom pozivu ima jednu usporedbu, jedno izvršavanje operacije zbrajanja, jedno pridruživanje te jedan poziv metode *_propagate_up*. Metoda *_propagate_up* ima dvije usporedbe i dvije računske operacije u uvjetu petlje `while`. U tijelu petlje imamo četiri pridruživanja i šest računskih operacija. Znamo da je visina potpunog binarnog stabla, pa time i hrpe, $\lfloor \log_2(n+1) \rfloor - 1$. Kako metoda *_propagate_up* prolazi putem od novog elementa (koji je na početku list) do korijena (u najgorem slučaju) znamo da u najgorem slučaju imamo $\lfloor \log_2(n+1) \rfloor - 1$ izvršavanja tijela petlje. Zbrojimo li sada sve operacije imamo:

$$1+1+1+(\lfloor \log_2(n+1) \rfloor - 1)\cdot(2+2+4+6) = 3+14\cdot\lfloor \log_2(n+1) \rfloor - 14 = 14\cdot\lfloor \log_2(n+1) \rfloor - 11$$

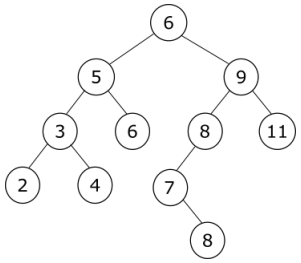
operacija. Iz gornjeg računa možemo zaključiti da je složenost metode *insert* $\Theta(\log n)$.
Slično možemo zaključiti da metoda *del_min* ima složenost $\Theta(\log n)$.

[2] započinje propagiranje novog elementa prema gore

3.3 Binarno stablo traženja (BST)

Definicija 3.7. *Binarno stablo traženja* (BST) je binarno stablo kod kojeg svaki čvor k ima oznaku d takvu da:

1. čvorovi u lijevom podstablu od k imaju oznaku manju ili jednaku d
2. čvorovi u desnom podstablu od k imaju oznaku veću od d



Slika 3.19: Primjer binarnog stabla traženja

Propozicija 3.1. Obilazak Inorder posjećuje čvorove binarnog stabla traženja u (ne strogo) rastućem redoslijedu njihovih oznaka.

Dokaz. Dokaz provodimo indukcijom po visini binarnog stabla traženja.
Ako je (pod)stablo prazno tada su oznake trivijalno posjećene u rastućem položaju.

Neka je $n \in \mathbb{N}_0$. Pretpostavimo da za sva binarna stabla traženja visine manje ili jednake n vrijedi da obilazak Inorder posjećuje čvorove u rastućem redoslijedu njihovih oznaka.

Neka je T binarno stablo traženja visine $n + 1$ i označimo s T_L i T_D njegovo lijevo i desno podstablo. Obilazak Inorder (pod)stabla se sastoji od rekurzivnog obilaska lijevog podstabla, zatim posjeta korijenu, i na kraju rekurzivnog obilaska desnog podstabla.
Iz pretpostavke indukcije slijedi da će obilazak T_L posjetiti čvorove u rastućem redoslijedu oznaka. Po definiciji binarnog stabla traženja su sve oznake čvorova iz T_L manje od oznake korijena, tako da se posjetom korijena nakon lijevog podstabla, ne kvari rastući poredak oznaka do sada posjećenih čvorova.
Također, iz pretpostavke indukcije slijedi da će obilazak T_D posjetiti čvorove u rastućem redoslijedu oznaka. Kako je iz definicije binarnog stabla traženja slijedi da su sve oznake u T_D veće od korijenove, posjetom desnog podstabla, nakon korijena, opet se ne kvari rastući poredak oznaka posjećenih čvorova.
Po principu matematičke indukcije slijedi tvrdnja. □

Posljedica ove propozicije (i toga da je obilazak Inorder izvršiv u vremenu $\Theta(n)$) je da u vremenu $\Theta(n)$ možemo iterirati kroz ključeve preslikavanja (preslikavanje ćemo definirati u točki 5.3), ako je preslikavanje implementirano pomoću binarnog stabla traženja.

Binarno stablo traženja

labeltype ... bilo koji tip podataka koji koristimo za oznake čvorova
Node ... bilo koji tip podataka. Tip *Node* koristimo za imena čvorova. Nul-referencu `None` koristimo kao ime nepostojećeg čvora.
BinarySearchTree ... binarno stablo traženja čijim čvorovima su pridružene oznake tipa *labeltype*
make_null(T) ... stvara prazno BST *T*
empty(T) ... vraća je li BST *T* prazno
left_subtree(T, node) ... vraća lijevo podstablo čvora *node* u BST-u *T*
right_subtree(T, node) ... vraća desno podstablo čvora *node* u BST-u *T*
insert(T, x) ... dodaje čvor s oznakom *x* u BST *T* i vraća taj čvor
delete(T, node) ... uklanja čvor *node* iz BST-a *T*
first(T) ... vraća čvor s najmanjom oznakom
last(T) ... vraća čvor s najvećom oznakom
before(T, node) ... vraća čvor za koji vrijedi da mu je oznaka najveća takva da je manja ili jednaka oznaci čvora *node* (taj čvor bi bio posjećen neposredno prije čvora *node* u obilasku Inorder). Vraća `None` ako je *node* jednak *first(T)*.
after(T, node) ... vraća čvor za koji vrijedi da mu je oznaka najmanja takva da je veća od oznake čvora *node* (taj čvor bi bio posjećen neposredno nakon čvora *node* u obilasku Inorder). Vraća `None` ako je *node* jednak *last(T)*.
label(T, node) ... vraća oznaku čvora *node*
Metode *left_subtree*, *right_subtree*, *delete*, *before*, *after* i *label* nisu definirane ako čvor *node* ne pripada binarnom stablu traženja *T*.

Vrijednost *first(T)* možemo odrediti tako da počnemo u korijenu, i idemo do lijevog djeteta sve dok ono postoji. Kada dođemo do čvora koji nema lijevo dijete, našli smo „najmanji" čvor. Zbog simetrije, vrijednost *last(T)* dobijemo tako da idemo od korijena udesno.

Prethodnika, tj. *before(T, node)* određujemo algoritmom danim u isječku 3.19, dok sljedbenika, tj. *after(T, node)*, određujemo algoritmom danim u isječku 3.20.

```
52     def before(T, node: Node) -> Node:
53         if node.left_childp:
54             return BinarySearchTree(rootp = node.left_childp
    ).last()
55         else:
```

```
56              nodeParent = node.parentp
57              while nodeParent and node == nodeParent.
    left_childp:
58                  node = nodeParent
59                  nodeParent = node.parentp
60              return nodeParent
```

Isječak 3.19: Metoda *before*

```
62      def after(T, node: Node) -> Node:
63          if node.right_childp:
64              return BinarySearchTree(rootp = node.
    right_childp).first()
65          else:
66              nodeParent = node.parentp
67              while nodeParent and node == nodeParent.
    right_childp:
68                  node = nodeParent
69                  nodeParent = node.parentp
70              return nodeParent
```

Isječak 3.20: Metoda *after*

Rad tih algoritama temelji se na radu obilaska Inorder te je njihova valjanost dokazana propozicijom 3.1.

Pogledajmo sada složenost metode *before*: Označimo sa h visinu stabla. Imamo dva slučaja:

1. čvor *node* ima lijevo dijete: U ovom slučaju metoda poziva metodu *last(T)* na lijevom podstablu od T. Metoda *last* ima složenost $\Theta(h)$.

2. čvor *node* nema lijevo dijete: U ovom slučaju izvršavamo *while*-petlju. Svakim izvršavanjem tijela petlje podižemo se za jedan nivo u BST-u zbog čega imamo najviše h izvršavanja tijela. Jer tijelo petlje ima dva izvršavanja operacija i jednu usporedbu u uvjetu petlje, u ovom slučaju imamo najviše $1 + h \cdot 3$ operacija.

Iz ta dva slučaja možemo zaključiti da metoda *before* ima složenost $\Theta(h)$.
Analogno možemo zaključiti da *after* također ima složenost $\Theta(h)$.

3.3.1 Pretraživanje BST-a

Najvažnija posljedica definicije BST-a je njegov algoritam pretraživanja. Algoritam se temelji na tome da BST gledamo kao stablo odlučivanja, uz to da je postavljeno pitanje „Je

li vrijednost traženog ključa manja od trenutnog ključa ili je veća?". U slučaju da je manja idemo lijevo, a ako je veća idemo desno. Ako su vrijednosti jednake stajemo jer smo našli traženi čvor. Implementacija algoritma prikazana je na isječku 3.21.

```
72      def find(T, x: labeltype) -> Node:
73          current_node = T.rootp
74          while current_node:
75              if x < current_node.element:
76                  current_node = current_node.left_childp
77              elif x == current_node.element:
78                  return current_node
79              elif x > current_node.element:
80                  current_node = current_node.right_childp
```

Isječak 3.21: Metoda *find*

Pri svakom izvršavanju tijela petlje *while* se, ako nismo pronašli element, pomičemo jednu razinu niže u stablu. Zaključujemo da ćemo imati najviše h prolaza kroz petlju. Na temelju toga možemo zaključiti da *find* ima složenost $\Theta(h)$.

3.3.2 Ubacivanje čvorova

Prilikom dodavanja elementa moramo paziti da očuvamo svojstvo BST-a. Ako je stablo prazno, tj. dodajemo prvi čvor, ne moramo brinuti o čuvanju svojstva. Ako BST nije prazno, pozivamo rekurzivnu metodu *_insert*. Metoda *_insert*, slično kao *find* gleda da li je x veći ili manji (ili jednak) oznaci čvora *node* i ovisno o slučaju radi sljedeće:

1. Ako je x manji ili jednak oznaci čvora *node*:
   Bazni slučaj rekurzije je kada *node* nema lijevo dijete. U tom slučaju čvoru *node* dodajemo lijevo dijete s oznakom x. Ukoliko *node* već ima lijevo dijete pozivamo *_insert(x, node.left_childp)*

2. Ako je x veći od oznake čvora *node*:
   Bazni slučaj rekurzije je kada *node* nema desno dijete. U tom slučaju čvoru *node* dodajemo desno dijete s oznakom x. Ukoliko *node* već ima desno dijete pozivamo *_insert(x, node.right_childp)*

Slično kao i metoda *find(T,x)* metoda *inset(t,x)* ima složenost $\Theta(h)$.

```
82      def insert(T, x: labeltype) -> Node:
83          if T.empty():
84              T.rootp = Node(parentp = None, left_childp =
    None, right_childp = None, element = x)
```

```
85              return T.rootp
86          return T._insert(x, T.rootp)
87
88      def _insert(T, x: labeltype, node: Node, parent = None):
89          if x <= node.element:
90              if node.left_childp == None:
91                  node.left_childp = Node(parentp = node,
    left_childp = None, right_childp = None, element = x)
92                  return node.left_childp
93              else:
94                  return T._insert(x, node.left_childp)
95          else:
96              if node.right_childp == None:
97                  node.right_childp = Node(parentp = node,
    left_childp = None, right_childp = None, element = x)
98                  return node.right_childp
99              else:
100                 return T._insert(x, node.right_childp)
```

Isječak 3.22: Metoda *insert*

3.3.3 Brisanje čvorova

Za razliku od dodavanja gdje smo dodavali samo listove, kod brisanja možemo uklanjati i čvorove koji nisu listovi. Označimo s *node* čvor koji želimo obrisati, a s T_{node} podstablo kojemu je *node* korijen. Ukupno imamo nekoliko slučajeva ovisno o tome koliko čvor *node* ima djece:

1. Čvor *node* nema djece: Možemo ga samo obrisati.

2. Čvor *node* ima jedno dijete: Zamijenimo ga s njegovim djetetom. Time nismo narušili svojstvo BST-a.

3. Čvor *node* ima dva djeteta: Kako *node* ima djecu ne možemo ga samo obrisati jer bi se time izgubila veza sa ostalim čvorovima u T_{node}. Također, ne možemo ga samo zamijeniti njegovim djetetom kao u slučaju 2 jer ima dvoje djece. Zbog toga ćemo *node* zamijeniti s najmanjim elementom u desnom podstablu od T_{node}. Označimo taj čvor s $node_2$.
   Pokažimo sada da time nismo narušili svojstva iz definicije 3.7:

- Kako je čvor $node_2$ element desnog podstabla od T_{node} znači da je njegova oznaka veća od oznake čvora *node*, pa samim time i od svih oznaka u lijevom podstablu od T_{node}.
- Budući da smo uzeli najmanji element u desnom podstablu od T_{node} vrijedi da su svi ostali elementi tog podstabla veći od njega.

```
102     def delete(T, x: labeltype):
103         node = T.find(x)
104         if node:
105             T._delete(node)
106 
107     def _delete(T, node: Node):
108         parent = node.parentp
109         if node.left_childp == None: #nema lijevo dijete
110             T._update_pointers(node, node.right_childp)
111         elif node.right_childp == None: #nema desno dijete
112             T._update_pointers(node, node.left_childp)
113         else: #ima oba djeteta
114             node2 = BinarySearchTree(rootp=node.right_childp
    ).first()
115             node.element = node2.element
116             T._delete(node2)
```

Isječak 3.23: Metoda *delete*

Metoda *delete(T, x)* prvo poziva metodu *find(T, x)* koja pronalazi čvor s vrijednošću *x*. Zatim ako takav čvor postoji poziva metodu *_delete(T, node)* koja briše čvor kao što je opisano u prethodno navedenim slučajevima. Metode *find* i *_delete* imaju složenost $\Theta(h)$ zbog čega metoda *delete* ima složenost $\Theta(h) + \Theta(h) = \Theta(h)$.

Poglavlje 4

Prioritetni red

U ovom poglavlju uvodimo ATP Prioritetni red (*priority queue*). To je kolekcija elemenata kojima su pridruženi prioriteti i koja omogućava ubacivanje elemenata u bilo kojem trenutku i izbacivanje elementa s najnižom vrijednosti prioriteta. Prilikom dodavanja elementa u prioritetni red, pridružuje mu se prioritet. Kao prioritet moguće je koristiti bilo koji tip na kojem postoji totalni uređaj.

U knjizi [2] možemo pronaći sljedeći primjer korištenja prioritetnog reda. Pretpostavimo da je neki let popunjen jedan sat prije polijetanja. Zbog mogućnosti otkazivanja, zrakoplovna tvrtka ima red putnika koji čekaju u nadi da dobiju mjesto na letu. Iako je vrijeme dolaska putnika utječe na to hoće li dobiti mjesto u zrakoplovu, postoje i drugi čimbenici kao što su cijena koju je putnik platio i *frequent flyer status*. Zbog toga je moguće da mjesto u zrakoplovu dobije putnik koji je došao kasnije, ako je tom putniku dodijeljen bolji prioritet.

Prioritetni red

prioritytype ... bilo koji tip podataka na kojem postoji totalni uređaj
valuetype ... bilo koji tip podataka
elementtype ... skup svih parova (p, x) gdje je p tipa *prioritytype*, a x tipa *valuetype*
PriorityQueue ... konačni niz (ne nužno različitih) podatataka tipa *elementtype*
make_null(P) ... pretvara prioritetni red P u prazni prioritetni red
empty(P) ... vraća *True* ako je prioritetni red P prazan. Inače vraća *False*.
insert(P, p, x) ... ubacuje uređeni par (p, x) u prioritetni red P, gdje je x element koji želimo ubaciti, a p njegov prioritet.
delete_min(P) ... izbacuje element s najmanjim prioritetom iz prioritetnog reda P. Vraća uređeni par (p, x), pri čemu je p prioritet, a x vrijednost elementa. Nije definirana u slučaju da je prioritetni red P prazan.

4.1 Implementacija prioritetnog reda pomoću sortirane liste

U ovoj implementaciji koristimo listu kao temeljnu strukturu podataka. U svakom trenutku elementi u listi su nam sortirani padajuće prema prioritetu. Zbog tog svojstva, brisanje elementa s najmanjim prioritetom svodi se na brisanje zadnjeg elementa u listi, zbog čega metoda *delete_min* ima složenost $\Theta(1)$. Kada dodajemo element u prioritetni red moramo odrediti poziciju elementa i ubaciti element u listu zbog čega metoda *insert* ima složenost $\Theta(n)$. Metoda *empty*, dobrim odabirom implementacije klase `List`, ima složenost $\Theta(1)$.

4.2 Implementacija prioritetnog reda pomoću hrpe

U ovom poglavlju za implementaciju koristimo hrpu. Konkretno, koristimo implementaciju hrpe iz poglavlja 3.2.3.

```
8  @dataclass
9  class PriorityQueue:
10     heap: Heap
11
12     @classmethod
13     def make_null(cls):
14         return cls(heap= Heap.make_null())
```

Isječak 4.1: Varijabla i konstruktor klase `PriorityQueue`

Na isječku 4.2 vidimo da se metode *insert* i *delete_min* svode na pozive odgovarajućih metoda iz klase `Heap`. Iz toga možemo zaključiti da obe metode imaju složenost $\Theta(n)$.

```
19     def insert(Q, priority: prioritytype, x: valuetype):
20         Q.heap.insert([priority, x])
21
22     def delete_min(Q) -> 'elementtype':
23         if Q.empty():
24             raise ValueError('Empty PriorityQueue')
25         return Q.heap.del_min()
```

Isječak 4.2: Metode *insert* i *delete_min* za Prioritetni red

4.3 Implementacija prioritetnog reda pomoću BST-a

Kao strukturu koristit ćemo binarno stablo traženja. Implementacije metoda *empty* i *insert* se svode na pozive odgovarajućih metoda iz BST-a. Metoda *delete_min* (slika 4.3) prvo pronalazi najmanji element i zatim ga briše.

```
22      def delete_min(Q) -> 'elementtype':
23          if Q.empty():
24              raise ValueError('Empty PriorityQueue')
25          min_element = Q.elements.first()
26          if min_element:
27              Q.elements.delete(Q.elements.label(min_element))
28              return  Q.elements.label(min_element)
```

Isječak 4.3: Implementacija prioritetnog reda pomoću BST-a

Kako metode *insert*, *first* i *delete* u implementaciji ATP-a BST pomoću pointera imaju složenost $\Theta(h)$ (gdje je h visina BST-a), možemo zaključiti da i metode *insert* i *delete_min* u ovoj implementaciji ATP-a Prioritetni red imaju složenost $\Theta(h)$.

Poglavlje 5

Neuređene strukture podataka

Do sada smo upoznali brojne apstraktne tipove podataka. Zajedničko im je što su svi uređeni: na primjer, liste linearno, a stabla hijerarhijski. U ovom poglavlju definirat ćemo nekoliko neuređenih apstraktnih tipova podataka. Smatramo da je čitatelj u prethodnim poglavljima stekao dovoljno iskustva te implementacije ostavljamo čitatelju za vježbu.

5.1 Skup

U ovoj točki upoznajemo ATP *Skup* koji otprilike odgovara matematičkom pojmu skupa. Riječ je o kolekciji međusobno različitih elemenata između kojih ne postoji nikakav uređaj. Pretpostavit ćemo da na skupu vrijednosti, koje mogu poprimiti elementi skupa, postoji totalni uređaj $\leq$.

Skup

elementtype ... bilo koji skup na kojem postoji totalni uređaj ($\leq$)
Set ... konačni skup čiji elementi su međusobno različiti podaci tipa *elementtype*
make_null(A) ... pretvara skup A u prazan skup
insert(A, x) ... ubacuje element x u skup A ako $x \notin A$. Ako je $x \in A$, ne mijenja A.
delete(A, x) ... izbacuje element x iz skupa A. Ako $x \notin A$, ne radi ništa.
member(A, x) ... vraća *True* ako je $x \in A$. Inače vraća *False*.
min(A) ... vraća element skupa A s najmanjom vrijednosti, u smislu uređaja $\leq$. Nije definirana ako je A prazan skup.
max(A) ... vraća element skupa A s najvećom vrijednosti, u smislu uređaja $\leq$. Nije definirana ako je A prazan skup.
subset(A, B) ... vraća *True* ako je $A \subseteq B$. Inače vraća *False*.
union(A, B) ... stvara i vraća skup $C = A \cup B$. Ne mijena skupove A i B.
intersect(A, B) ... stvara i vraća skup $C = A \cap B$. Ne mijenja skupove A i B.

difference(A, B) ... stvara i vraća skup $C = A \setminus B$. Ne mijenja skupove A i B.

Neke moguće strukture podataka koje se mogu koristiti za implementaciju ATP-a Skup su *bit*-vektor i sortirana vezana lista.

5.2 Rječnik

Rječnik (*dictionary*) je posebna vrsta skupa kod kojeg nije potrebno obavljanje složenijih operacija kao što su unija i presjek. Obavljaju se samo operacije ubacivanja i izbacivanja elemenata, te provjere je li neki element u skupu, tj. rječniku.

Rječnik

elementtype ... bilo koji skup na kojem postoji totalni uređaj ($\leq$)
Dictionary ... konačni skup čiji elementi su međusobno različiti podaci tipa *element-type*
make_null(D) ... pretvara rječnik D u prazan rječnik
add(x, D) ... ubacuje element x u rječnik D ako x nije u D. Ako je x već sadržan u D, tada ne radi ništa.
remove(x,D) ... izbacuje element x iz rječnika D. Ako D ne sadrži x, ne radi ništa.
member(x, D) ... vraća *True* ako je $x \in D$. Inače vraća *False*.

Kako je ATP Rječnik jednostavniji od ATP-a Skup, i nije potrebno izvršavati složene skupovne operacije, moguće je napisati efikasniju implementaciju. Kao strukture podataka mogu se koristiti na primjer *hash*-tablice ili binarno stablo traženja.

5.3 Preslikavanja

Još jedan važan ATP koji je vezan uz pojam skupa je Preslikavanje (*mapping*).

Definicija 5.1. *Preslikavanje* M je skup uređenih parova (d, r), gdje su svi d-ovi podaci jednog tipa (*domena* ili područje definicije), a svi r-ovi podaci drugog tipa (*kodomena* ili područje vrijednosti). Pri tome za svaki d postoji najviše jedan par (d, r) u M.

U uređenom paru (d, r), d nazivamo *ključ* (*key*), a r nazivamo *vrijednost* (*value*). Ako za zadani d preslikavanje M sadrži par (d, r) tada pišemo $r = M(d)$ i kažemo da je $M(d)$ definirano. Ako za zadani d preslikavanje M ne sadrži par (d, r) ni za koji r kažemo da $M(d)$ nije definirano.

Neki primjeri preslikavanja su:

- Svaki student, prilikom prvog upisa na fakultet, dobije JMBAG (jedinstveni matični broj akademskog građanina). Nakon toga informacijski sustav na fakultetu može koristiti JMBAG kao jedinstveni identifikator studenta. U ovom slučaju JMBAG je ključ, a podaci o studentu vrijednost.
- Prilikom registracije na portalu ili forumu potrebno je upisati adresu elektroničke pošte, koju vlasnik portala može koristiti kao ključ pod kojim će spremati sve podatke vezane uz tog korisnika.

Preslikavanje

domain ... bilo koji tip (domena)
range ... bilo koji tip (kodomena)
Mapping ... preslikavanje čiju domenu čine podaci tipa *domain*, a kodomenu podaci tipa *range*
make_null(M) ... pretvara preslikavanje M u nul-preslikavanje, koje nije nigdje definirano
assign(M, d, r) ... definira $M(d)$ tako da vrijedi $M(d) = r$, bez obzira na to je li $M(d)$ prije bilo definirano
deassign(M, d) ... iz preslikavanja M izbacuje uređeni par kojemu je ključ jednak d. Ako $M(d)$ nije definirano metoda ne mijenja M.
compute(M,d) ... vraća vrijednost $M(d)$ ako je $M(d)$ definiran. Nije definirana ako $M(d)$ nije definiran.

Primjeri struktura podataka koje se mogu koristiti za implementaciju ATP-a Preslikavanje su lista, *hash*-tablica i binarno stablo traženja.

5.4 Binarne relacije

Definicija 5.2. *Binarna relacija*, ili kraće *relacija* (*relation*) je podskup R Kartezijevog produkta $A \times B$, gdje A zovemo *prva domena*, a B *druga domena*. Neka su $d_1 \in A$ i $d_2 \in B$. Ako relacija R sadrži uređeni par (d_1, d_2) tada kažemo da je d_1 u relaciji sa d_2 i pišemo $d_1\ R\ d_2$. U protivnom kažemo da d_1 nije u relaciji sa d_2 i pišemo $d_1 \not R\ d_2$.

Primjeri relacija su:

- Korisnici se mogu pretplatiti na različite pakete televizijskih programa. Svaki korisnik može imati pretplatu na više različitih paketa. Prvu domenu, u ovom primjeru, čini skup svih korisnika, a drugu skup svih paketa. Kada se korisnik pretplati na novi

paket, u relaciju dodajemo novi uređeni par (*korisnik*, *paket*), a kada odjavi paket uklanjamo odgovarajući uređeni par. U svakom trenutku možemo dobiti popis svih paketa na koje je određeni korisnik pretplaćen, ali i popis svih korisnika koji su se pretplatili na određeni paket.

- Isti proizvod moguće je kupiti u više trgovina, a svaka trgovina ima više različitih proizvoda. Prva domena je skup svih proizvoda, a druga skup svih trgovina. Ako trebamo određeni proizvod, možemo dobiti popis svih trgovina koje ga imaju u ponudi. Ako planiramo ići u trgovinu, možemo dobiti popis svih proizvoda koje možemo kupiti u toj trgovini.

Relacija

domain1 ... bilo koji tip (prva domena)
domain2 ... bilo koji tip (druga domena)
Relation ... binarna relacija s prvom domenom *domain1* i drugom domenom *domain2*
make_null(R) ... pretvara R u nul-relaciju u kojoj nijedan element prve domene nije u relaciji ni s jednim elementom iz druge domene
relate(R, d_1, d_2) ... postavlja $d_1\ R\ d_2$, neovisno o tome jesu li već u relaciji ili ne
unrelate(R, d_1, d_2) ... postavlja $d_1\ \not{R}\ d_2$, neovisno jesu li u relaciji ili ne
compute2(R, d_1) ... vraća skup $\{d_2 \mid d_1\ R\ d_2\}$
compute1(R, d_2) ... vraća skup $\{d_1 \mid d_1\ R\ d_2\}$

Za implementaciju ATP-a Relacija moguće je koristiti iste strukture kao za ATP Rječnik pri čemu metode *make_null*, *relate* i *unrelate* odgovaraju metodama *make_null*, *insert* i *delete* u rječniku.

Bibliografija

[1] A. Dujella. *Uvod u teoriju brojeva, skripta*.

[2] M.T. Goodrich, R. Tamassia i M.H. Goldwasser. *Data Structures and Algorithms in Python*. Wiley, 2013.

[3] R. Manger. *Strukture podataka i algoritmi*. Element, 2014.

Sažetak

U programima i u svakodnevnom životu često imamo podatke koje trebamo organizirati. Prilikom programiranja to činimo korištenjem apstraktnih tipova podataka.

U prvom poglavlju definiramo pojmove strukture podataka, apstraktnog tipa podataka i algoritma.

U drugom poglavlju definiramo i implementiramo listu, vezanu listu, stog, red i dvostrani red te komentiramo složenosti njihovih metoda.

U trećem poglavlju govorimo o stablima, binarnim stablima i binarnim stablima traženja.

U četvrtom poglavlju upoznat ćemo i implementirati apstraktni tip podataka prioritetni red, koji je sličan linearnim strukturama iz drugog poglavlja.

U zadnjem, petom poglavlju, definiramo neuređene strukture podataka: skup, rječnik, preslikavanje i binarne relacije.

Summary

In computer programs and everyday life, we have data that we need to organize. While programming, we do that by using abstract data types.

In the first chapter, we define data structures, abstract data types, and algorithms.

In the second chapter, we define and implement a list, linked list, stack, queue, and deque and comment on the complexity of their methods.

In the third chapter, we talk about trees, binary trees, and binary search trees.

In the fourth chapter, we get to know and implement the abstract data type priority queue, which is similar to the linear structures from the second chapter.

In the last, fifth chapter, we define the unordered data structures: set, dictionary, mapping, and relation.

Životopis

Rođena sam 14. lipnja 1995. u Grazu. Osnovnu školu završila sam u Cestici. 2013. godine završila sam srednjoškolsko obrazovanje u Drugoj gimnaziji u Varaždinu (smjer opća gimnazija). Tijekom osnovne i srednje škole sudjelovala sam na nizu natjecanja iz područja matematike, fizike, tehničke kulture, likovnog i njemačkog jezika.

Nakon srednjoškolskog obrazovanja, godine 2013. upisala sam Prirodoslovno-matematički fakultet u Zagrebu, smjer Matematika. Nakon tri godine upisala sam na istome fakultetu diplomski studij Računarstvo i matematika.