Vježba 1: Analiza algoritama

(predavanja)

Zadaci za vježbu

1. Sljedećim funkcijama dodijelite odgovarajuću vremensku složenost O(1), O(n) ili O(n^2)?

```
def f1(lista1, lista2):
   for n in range(0, len(lista1)):
      for m in range(0, 100):
         print(n, m)

def f2(lista):
   for i in range(0, 5):
      print(lista[i])

def f3(lista1, lista2):
   # neka liste lista1 i lista2 imaju jednak broj elemenata
   for n in range(0, len(lista1)):
      for m in range(0, len(lista2)):
         print(n, m)

def f4(lista):
   for i in range(0, len(lista)):
      print(lista[i])

def f5(lista):
   for i in range(0, len(lista) / 2):
      print(lista[i])

def f6(lista):
   for i in range(len(lista) - 1):
      print(lista[i])

def f7(lista1, lista2):
   # liste lista1 i lista2 imaju jednak broj elemenata
   svi_elementi = lista1 + lista2
   for i in range(0, len(svi_elementi)):
      print(svi_elementi[i])
```

2. U funkciji *zbroji* ispod neka je parametar *p* lista cijelih brojeva. Da li je moguće napisati ovu funkciju tako da zbraja sve elemente liste *p* u vremenu O(1)?

```
def zbroji(p):
   ...
```

3. Napišite funkciju koja sortira elemente tako da uzastopno traži najmanji element u listi (pod uvjetom da je prethodni najmanji element uklonjen iz liste). Koja je vremenska složenost ove funkcije?

4. Napišite funkciju koja preokreće elemente liste (na primjer, za listu [1, 2, 3] daje listu [3, 2, 1]). Koja je vremenska složenost vaše funkcije?

Vježba 2: Rekurzija

2.1 Uvod

S obzirom da su operacije osnovni elementi gotovo svakog programa, one kao takve određuju zakonitosti o tome kako se neki kompjuterski proces odvija na razini tih operacija. One određuju kako se jedna faza tog procesa nadograđuje na prethodnu. U programiranju postoje dva osnovna načina kojima se definira takva "evolucija" nekog procesa: *iteracija* (petlje) i *rekurzija*. Do sada smo petlje koristili kao osnovni mehanizam za iteraciju, odnosno ponavljanje jednog niza naredbi. Rekurzija se, kao i iteracija, također odnosi na ponavljanje, ali se za to ne koristi petlja nego rekurzivni poziv operacije. Operacija koja u svojoj definiciji sadrži jedan ili više takvih poziva zove se *rekurzivna operacija*. Općenito, rekurzivna operacija u jednoj osnovnoj formi je ona koja poziva samu sebe s jednog ili više mjesta unutar svoje definicije:

```
def rek(x):
   ...
   ... rek(n) ...
   ...
```

Svaki postupak koji se može definirati iteracijom može se također definirati rekurzijom i obratno. Oba ova načina imaju prednosti i mana, a koji od njih treba koristiti zavisi, kao uvijek, od prirode problema koji rješavamo.
Kao jednu usporedbu iteracije i rekurzije proučit ćemo jednostavnu matematičku operaciju koja se zove *faktorijel* i koja daje umnožak svih (cijelih) brojeva u nekom intervalu *1..n*. Ova se operacija u matematici označava sa *n!* i definirana je na ovaj način:

$$n! = n * (n - 1) * (n - 2) * (n - 3) * ... * 1.$$

Na primjer, *5! = 4 * 3 * 2 * 1 = 24*. Iterativna varijanta operacije, koju ćemo zvati *fakt_iter*, može se napisati ovako:

```
def fakt_iter(n):
    rezultat = 1
    while n > 0:
        rezultat *= n
        n -= 1

    return rezultat

fakt_iter(4)
=> 24
```

Ako u svakom koraku zamijenimo varijable *rezultat* i *n* s njihovom vrijednošću u tom koraku možemo jasno vidjeti kako se došlo do rezultata:

1. *rezultat = 1 (početna vrijednost)*
2. *rezultat = 1 * 4, n = 4*
3. *rezultat = 4 * 3, n = 3*
4. *rezultat = 12 * 2, n = 2*
5. *rezultat = 24 * 1, n = 1*

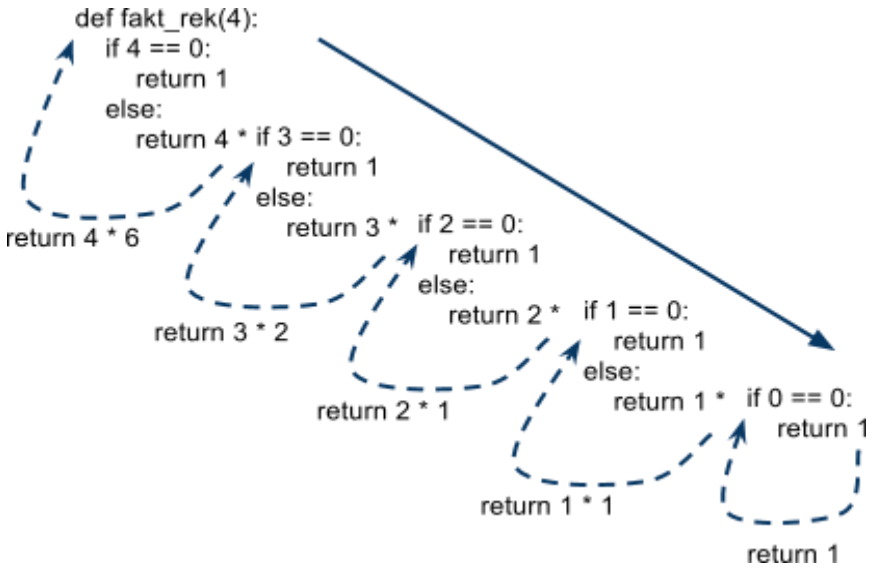
Vidimo da se ovaj postupak odvija tako da je u svakom koraku vrijednost varijabli *rezultat* i *n* bliža nekoj konačnoj vrijednosti. Isto tako, uvjet petlje *n > 0* osigurava njegovo prekidanje kada *n* dosegne vrijednost 0.
Faktorijel se često definira i na ovaj način:

$$n! = \begin{cases} 1 & \text{ako je } n = 0 \\ n \times (n-1)! & \text{ako je } n > 0 \end{cases}$$

U drugom redu vidimo da je u ovoj definiciji faktorijel definiran pomoću faktorijela, ali za manju vrijednost, odnosno za *n - 1*. Ovo se naziva *rekurzivnom definicijom* i za nju možemo napisati rekurzivnu varijantu ove operacije, koju ćemo zvati *fakt_rek*:

```
def fakt_rek(n):
    if n == 0:
        return 1
    else:
        return n * fakt_rek(n - 1)

fakt_rek(4)
=> 24
```

Prvo što ovdje treba primijetiti je to da u ovoj operaciji ne koristimo petlju nego pozivamo operaciju *fakt_rek* unutar te iste operacije! Ovdje se odmah možemo pitati sa čime množimo *n* ako svaki put idemo na početak ove operacije baš kada nam treba neki broj? S tehničkog gledišta, rekurzivni poziv operacije odvija se na potpuno isti način kao i svaki drugi poziv, odnosno ne postoji nikakav "specijalni" mehanizam koji je ugrađen u Pythonov interpreter i koji bi se koristio za rekurzivne operacije. Rekurzivni poziv *fakt_rek(n - 1)* u stvari daje umnožak brojeva u intervalu *1 .. n - 1* koji onda pomnožimo sa *n*. Drugim riječima, konačni rezultat dobijemo tako da svaki broj *k* pomnožimo s umnoškom brojeva na manjem intervalu, *1 .. k - 1*. Kao što smo iterativnu varijantu ove operacije prikazali kao jedan niz naredbi koji se ponavlja više puta, njenu rekurzivnu varijantu prikazujemo kao jedan izraz koji se sastoji od više podizraza. Operaciju *fakt_rek* možemo zamisliti tako da na svako mjesto gdje se nalazi *fakt_rek(n - 1)* stavimo cijelu definiciju ove operacije s tim da varijablu *n* zamijenimo s njenom trenutnom vrijednošću, kako je pokazano na slici 2.4.



Slika 2.4 – Izvršavanje izraza *fakt_rek(4)*.

Ako izdvojimo samo izraz *return*-naredbe dobili bi ovakav niz koraka:

1. *return 4 * fakt_rek(4 - 1)*
2. *return 4 * 3 * fakt_rek(3 - 1)*
3. *return 4 * 3 * 2 * fakt_rek(2 - 1)*
4. *return 4 * 3 * 2 * 1 * fakt_rek(1 - 1)*
5. *return 4 * 3 * 2 * 1 * 1*
6. *return 4 * 3 * 2 * 1*
7. *return 4 * 3 * 2*
8. *return 4 * 6*
9. *return 24*

U svakom koraku imamo isti izraz: *n * fakt_rek(n - 1)*. U prvom koraku, prema tome, rješavamo početni izraz *fakt_iter(4)*, čiji je rezultat *4 * fakt_rek(4 - 1)*. U sljedećem koraku rješavamo izraz iz prethodnog koraka, odnosno *fakt_rek(3)*, čiji je rezultat *3 * fakt_rek(3 - 1)*. Ovdje, međutim, ne možemo zaboraviti na 4 iz prethodnog koraka jer s tom vrijednošću moramo pomnožiti rezultat ovog idućeg koraka, to jest rezultat poziva *fakt_rek(4 - 1)*, što je *3 * fakt_rek(3 - 1)*, i tako dalje. Množenje s brojem 4 i drugim brojevima koji slijede do koraka 5 je odgođeno. Tek kada se dosegne korak 5 mogu se početi množiti vrijednosti jer su tek tada sve one na raspolaganju. Izraz *if n == 0: return 1* osigurava prekidanje ovog procesa kada *n* dosegne vrijednost 0. To se vidi nakon koraka 4, gdje više ne pozivamo operaciju *fakt_iter* sa sljedećim manjim argumentom jer smo "došli do kraja", to jest došli smo do *fakt_iter(0)* za koji znamo da je 1, pa prema tome nema potrebe dalje pozivati ovu operaciju. Nakon toga, u koraku 5 imamo kompletan izraz množenja koji se postupno izračunava do konačnog rezultata. Koracima 5 do 8 se u stvari vraćamo iz svih ovih prethodnih poziva - korakom 5 vraćamo se iz poziva u koraku 4, korakom 6 iz poziva u koraku 3, korakom 7 iz poziva u koraku 2, i na kraju korakom 8 se vraćamo iz poziva u prvom koraku, čime dobivamo rezultat ove operacije.
Da bi promatrali izvršavanje ove operacije možemo je napisati tako da odvojimo rekurzivni poziv operacije od množenja, te da ispišemo vrijednost varijable *n* prije i poslije svakog

poziva. Svaku ćemo vrijednost od *n* prije poziva sačuvati u jednoj tekstualnoj varijabli tako da se vidi sa čime množimo rezultat tog poziva. Operaciju, koju ćemo zvati *fakt_rek_koraci*, možemo napisati ovako:

```
def fakt_rek_koraci(n, s):
    if n == 0:
        return 1
    else:
        s += str(n) + ' * '     # sačuvaj trenutnu vrijednost od n
        print(s, 'fakt_rek_koraci(', n - 1, ')')
        t = fakt_rek_koraci(n - 1, s)
        print(s, t)
        r = n * t
        return r

fakt_rek_koraci(4, '')
=> 4 *  fakt_rek_koraci( 3 )
   4 * 3 *  fakt_rek_koraci( 2 )
   4 * 3 * 2 *  fakt_rek_koraci( 1 )
   4 * 3 * 2 * 1 *  fakt_rek_koraci( 0 )
   4 * 3 * 2 * 1 *  1
   4 * 3 * 2 *  1
   4 * 3 *  2
   4 *  6
   24
```

Operacija *fakt_rek_koraci* radi na isti način kao i *fakt_rek*, samo što ovdje rezultat rekurzivnog poziva smještamo u varijablu prije nego što ga pomnožimo sa *n*.
Sada se vidi razlika između iterativnog i rekurzivnog procesa. Rad iterativne operacije bio je baziran na postepenim promjenama vrijednosti varijabli *rezultat* i *n* - varijablu *rezultat* povećavali smo u svakom koraku tako da smo je pomnožili sa *n*, a varijabla *n* nam je istovremeno služila kao brojač koraka. Kaže se da su *rezultat* i *n varijable stanja* ovog procesa. To znači da je u svakom koraku njegov progres bio određen tim dvjema varijablama. Općenito, iterativni proces je onaj čije je stanje u svakom koraku određeno varijablama stanja i pravilom kako se mijenjaju vrijednosti tih varijabli iz jedne iteracije u iduću. Kod našeg primjera možemo reći da svaki put nakon što smo varijablu *rezultat* postavili na *rezultat * n* i varijablu *n* na *n - 1*, ovaj proces je prešao na sljedeći korak.
Rekurzivna operacija radila je tako da se svaka vrijednost i operacija (kao što je množenje) morala pamtiti da bi na kraju izračunali konačni rezultat. Operacije množenja ovdje su bile "odgođene" dok nismo dobili kompletan izraz. Ovo odgađanje operacija nije ništa drugo nego određivanje vrijednosti argumenata koje mora prethoditi pozivu dotične operacije. S obizrom da je sam argument rezultat te iste operacije, dobivamo ovaj tipični "kružni" proces koji eventualno završava nekom konkretnom vrijednošću za taj argument (kao što je u našem primjeru broj 1 kada je *n* došao do 0). To se vidi na gornjem nizu koraka kao izraz koji se širi nakon čega slijedi skupljanje (kao posljedica vraćanja iz poziva operacije) i takva je forma tipična za rekurzivne procese. Općenito, proces koji je baziran na takvom nizu odgođenih operacija zove se *rekurzivan proces*.

Rekurzivne operacije mogu biti kompleksnije, gdje se ista operacija može koristiti kao argument na više mjesta. Kao jedan primjer proučit ćemo još jednu matematičku operaciju koja izračunava vrijednosti takozvanog *Fibonaccijevog niza*, koji izgleda ovako:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

Ovaj niz počinje s 0, 1, a svaki idući broj je zbroj prethodna dva. Prema tome, operacija *fib(n)* kojom se izračunava vrijednost ovog niza na poziciji *n* (gdje je 0 početna pozicija) može se matematički definirati ovako:

$$fib(n) = \begin{cases} 0, \text{ za } n = 0 \\ 1, \text{ za } n = 1 \\ fib(n - 1) + fib(n - 2), \text{ za } n > 1 \end{cases}$$

Ovu definiciju možemo lako prevesti u Python:

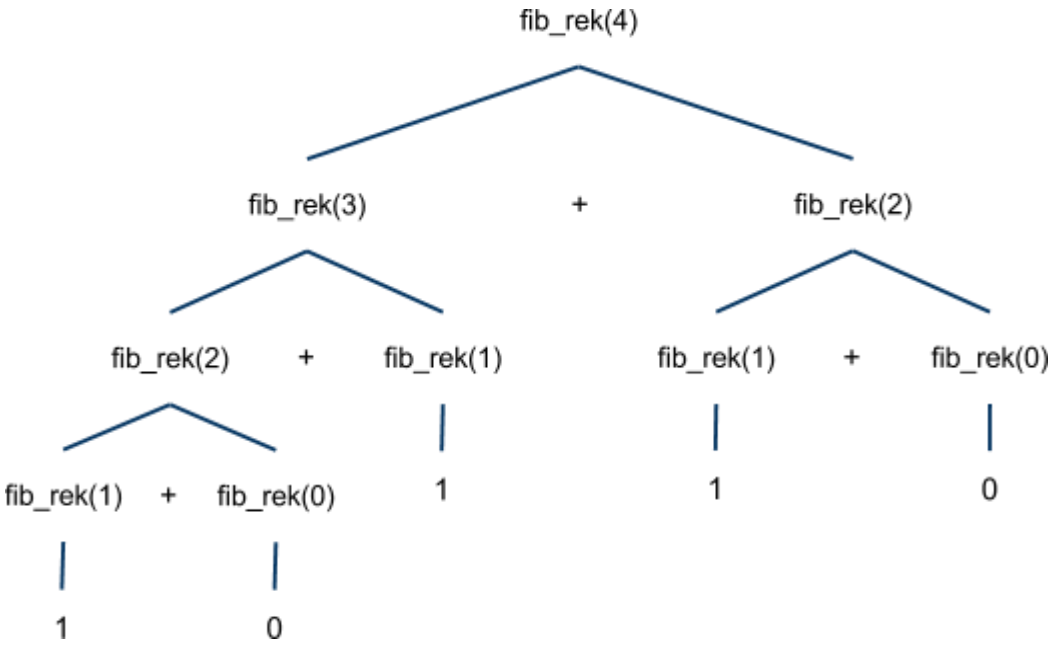
```
def fib_rek(n):
    if n in {0, 1}:
        return n
    else:
        return fib_rek(n - 1) + fib_rek(n - 2)

fib_rek(4)
=> 3
```

Sada možemo ovaj proces prikazati na isti način kao u prethodnom primjeru, za izraz *fib_rek(4)*:

1. *return fib_rek(4 - 1) + fib_rek(4 - 2)*
2. *return fib_rek(3 - 1) + fib_rek(3 - 2) + fib_rek(4 - 2)*
3. *return fib_rek(2 - 1) + fib_rek(2 - 2) + fib_rek(3 - 2) + fib_rek(4 - 2)*
4. *return 1 + fib_rek(2 - 2) + fib_rek(3 - 2) + fib_rek(4 - 2)*
5. *return 1 + 0 + fib_rek(3 - 2) + fib_rek(4 - 2)*
6. *return 1 + 0 + 1 + fib_rek(4 - 2)*
7. *return 1 + 0 + 1 + fib_rek(2 - 1) + fib_rek(2 - 2)*
8. *return 1 + 0 + 1 + 1 + fib_rek(2 - 2)*
9. *return 1 + 0 + 1 + 1 + 0*
10. *return 1 + 1 + 1*
11. *return 2 + 1*
12. *return 3*

Grananje izraza ovog procesa dobijemo tako da svaki poziv *fib_rek* zamijenimo s dva nova poziva jer se rezultat operacije *fib_rek* sastoji od dva rekurzivna poziva (za *n > 1*). Zbog toga je oblik izraza koje generira ovaj proces takav da se oni šire i skupljaju više puta. Takav oblik izraza generiranih nekom rekurzivnom operacijom zove se *rekurzivno stablo* zato jer ima formu stabla. Takvo stablo za izraz *fib_rek(4)* pokazano je na slici 2.5, gdje vidimo da

se svaki poziv ove operacije grana na dvije strane zato jer u ovoj operaciji imamo dva rekurzivna poziva spojena operatorom “+”. Na krajevima ovog stabla promatranog odozgo prema dole nalaze se brojevi koji zbrojeni daju rezultat ove operacije.



Slika 2.5 - Rekurzivno stablo izraza *fib_rek(4)*.

U rekurzivnom stablu ovog procesa možemo vidjeti da je dio tog stabla za *n* = 2 ponovljen dva puta. Prvi dio bio je formiran pozivom *fib(n - 2)* za *n* = 4, a drugi pozivom *fib(n - 1)* za *n* = 3, što se lako vidi na slici. Iz ovoga možemo zaključiti da rekurzivni procesi mogu biti neefikasni u nekim slučajevima jer se tada, kao u ovom slučaju, vrijednost za jedan te isti izraz izračunava više puta.
Iterativna varijanta za fibonaccijev niz po strukturi je slična operaciji *fakt_iter*:

```
def fib_iter(n):
    prethodni = 0
    sljedeći = 1
    while n > 0:
        t = prethodni + sljedeći
        prethodni = sljedeći
        sljedeći = t
        n -= 1

    return prethodni

fib_iter(4)
=> 3
```

Kao što je bio slučaj kod *fakt_iter*, proces za *fib_iter(4)* izgleda puno jednostavnije od rekurzivnog procesa:

1. ***prethodni = 0**, sljedeći = 1 (početne vrijednosti)*
2. *t = 1, **prethodni = 1**, sljedeći = 1, n = 4*
3. *t = 2, **prethodni = 1**, sljedeći = 2, n = 3*
4. *t = 3, **prethodni = 2**, sljedeći = 3, n = 2*
5. *t = 5, **prethodni = 3**, sljedeći = 5, n = 1*

Ovdje vidimo kako se varijabla *prethodni* postepeno približavala rezultatu, a varijabla *sljedeći* je sadržavala vrijednost koja dolazi nakon one u varijabli *prethodni*. Ovaj proces, kao i onaj za *fakt_iter*, nema nikakvog odgođenog računanja i svaki korak je isključivo određen vrijednostima varijabli *t*, *prethodni*, *sljedeći* i *n*.

Grananje koje je tipično za rekurzivne procese ima jednu, u nekim situacijama negativnu posljedicu, a to je potreba za većom količinom memorije u odnosu na iterativne procese. U primjeru operacija *fakt_rek* i *fib_rek* vidjeli smo da izraz, to jest broj vrijednosti koje treba "pamtiti", sve više raste, pa su i potrebe za memorijom tokom izvršavanja tih operacija sve veće. To nije bio slučaj kod njihovih iterativnih varijanti jer smo tamo koristili varijable stanja u kojima smo držali sve privremene vrijednosti dok nismo došli do konačnog rezultata. Općenito, kod rekurzivne varijante što je *n* bio veći trebalo je više memorije, dok je kod iterativne varijante potreba za memorijom bila konstantna, to jest nije ovisila o *n*. U primjeru fibonaccijevog niza operacija *fib_rek* imala je, osim memorije, jedan dodatni problem, a to je da je neke izraze izračunavala više puta, što bi kod većih vrijednosti za *n* rezultiralo značajnim rastom vremena potrebnog da dođe do rezultata. Iz ovoga možemo zaključiti da su iterativna rješenja u nekim slučajevima efikasnija od rekurzivnih, sa stajališta veličine potrebne memorije i brzine izvršavanja. To, međutim, nije uvijek slučaj, odnosno njihova efikasnost može biti i podjednaka. Kada budemo govorili o strukturama podataka vidjet ćemo da u nekim slučajevima kod obje vrste procesa količina potrebne memorije raste proporcionalno s veličinom ulaznih podataka i da im je efikasnost podjednaka.

Za kraj je ostalo još jedno pitanje vezano za ove dvije vrste procesa: Kada neki proces, odnosno operaciju treba definirati iterativno, a kada rekurzivno? Odgovor najčešće ovisi o tome koja je najprirodnija definicija problema koji rješavamo; međutim, ni ta odluka nije uvijek jednostavna. U našim primjerima s izračunavanjem faktorijela i fibonaccijevog niza lako smo mogli definirati oba problema na iterativan ili rekurzivan način. U takvim slučajevima možemo uzeti u obzir efikasnost izvršavanja, tako da bi se u oba ova slučaja opredijelili za iterativnu varijantu. S druge strane, kod nekih problema rekurzija je prirodno i prihvatljivo riješenje. Pretpostavimo da želimo napisati program koji traži neku zadanu vrijednost u nizu, s tim da sam taj niz može imati druge nizove kao elemente. Za početak možemo definirati predikat koji jednostavno pretražuje niz za zadanu vrijednost, ne uzimajući u obzir činjenicu da elementi tog niza mogu biti drugi nizovi:

```
def postoji_p(elem, niz):
    for e in niz:
        if e == elem:
            return True

    return False
```

```
postoji_p('plavo', ['crveno', 'zeleno', 'žuto', 'plavo',
                    'ljubičasto', 'sivo'])
=> True
```

Ova operacija radi ispravno ako se element koji tražimo nalazi u "glavnom" nizu, to jest ne nalazi se u nekom unutrašnjem nizu. Na primjer, za sljedeći ulazni niz ova operacija ne pronalazi traženi element:

```
postoji_p('plavo', ['crveno', 'zeleno', ['žuto', 'plavo'],
                    'ljubičasto', 'sivo'])
=> False
```

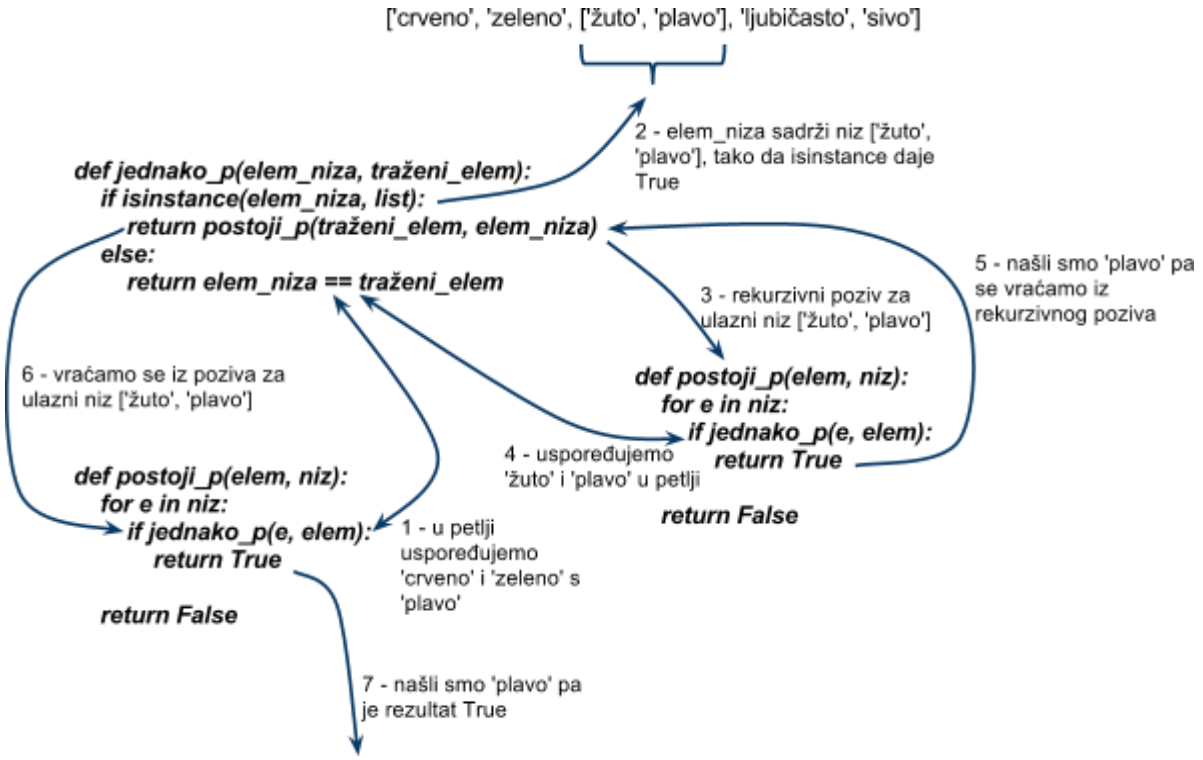
To je zato jer *'plavo'* nije isto što i *['plavo']* - u prvom slučaju imamo tekstualnu vrijednost, a u drugom niz, pa s obzirom da su to dvije različite vrste podataka ne možemo ih uspoređivati, to jest ne mogu biti jednaki. Da bi ova operacija radila s unutrašnjim nizovima moramo joj omogučiti da "uđe" u jedan takav niz ako dođe do takvog elementa, te da nastavi traženje unutar tog niza, počevši od prvog njegovog elementa. Takav program možemo napisati ovako:

```
def jednako_p(elem_niza, traženi_elem):
    if isinstance(elem_niza, list):      # je li elem_niza niz?
        return postoji_p(traženi_elem, elem_niza)
    else:
        return elem_niza == traženi_elem

def postoji_p(elem, niz):
    for e in niz:
        if jednako_p(e, elem):
            return True

    return False

postoji_p('plavo', ['crveno', 'zeleno', ['žuto', 'plavo'],
                    'ljubičasto', 'sivo'])
=> True
```

Dio unutar petlje radi gotovo identično prethodnoj varijanti, ali umjesto operatora "==" koristimo predikat *jednako_p*. Ovaj predikat prvo provjeri je li element koji uspoređujemo sa zadanom vrijednošću u stvari sam niz.[1] Ako nije onda ga jednostavno usporedimo s tom vrijednošću kao što smo radili u prethodnoj varijanti. Međutim, ako je, onda predikat *jednako_p* ponovo koristi predikat *postoji_p* s tim nizom kao ulaznim nizom! Prema tome, pretraživanje unutrašnjih nizova također obavlja operacija *postoji_p* čiji rezultat kaže da li traženi element postoji u unutrašnjem nizu. Taj rezultat koristi ta ista operacija dok

[1]Operacija *isinstance(x, t)* daje True ako je vrijednost *x* tipa *t*. U Pythonu nizovi su tipa *list*, tako da ako je *x* neki niz onda *isinstance(x, list)* daje True.

pretražuje vanjski niz. Ovaj je proces prikazan na slici 2.6. Ovdje se također radi o rekurziji, samo što operacija ne poziva samu sebe direktno, nego preko neke druge operacije. U ovom primjeru operacija *postoji_p* poziva operaciju *jednako_p* koja poziva operaciju *postoji_p*. Takve se operacije zovu *uzajamno rekurzivne operacije*.



Slika 2.6 - Izvršavanje izraza postoji_p('plavo', ['crveno', 'zeleno', ['žuto', 'plavo'], 'ljubičasto', 'sivo']).

Na ovaj način možemo naći bilo koji element, bez obzira na to koliko se on duboko nalazi u unutrašnjim nizovima. Na primjer,

```
postoji_p('zeleno', ['crveno', [['zeleno', 'žuto']]], ['plavo'],
                     'ljubičasto', 'sivo'])
=> True

postoji_p('zeleno', ['crveno', [['zeleno']], ['žuto']],
                    ['plavo'], 'ljubičasto', 'sivo'])
=> True

postoji_p('bijelo', ['crveno', [['zeleno', 'žuto']]], ['plavo'],
                     'ljubičasto', 'sivo'])
=> False
```

Operaciju *postoji_p* mogli smo napisati i ovako:

```
def postoji_p(elem, niz):
    for e in niz:
        if isinstance(e, list):
```

```
            if postoji_p(elem, e):
                return True
        else:
            if e == elem:
                return True

    return False
```

Ova varijanta radi isto što i prethodna, ali koristi direktnu rekurziju. Međutim, prethodna je varijanta jasnije napisana zato jer su prolaz kroz niz i uspoređivanje (operacijom *jednako_p*) odvojeni; ako pogledamo definiciju operacije *postoji_p* odmah je jasno kako ona radi zato jer nije "opterećena" time kada i kako treba napraviti rekurzivni poziv.

Ako pogledamo neki niz koji sadrži unutrašnje nizove, svaki od tih nizova možemo pretraživati na isti način kao i vanjski niz, to jest koristeći istu operaciju. Ulazni niz za tu operaciju upravo je taj unutrašnji niz. U ovakvim slučajevima upotreba rekurzije je izrazito korisna tehnika jer nam omogučava rad s podacima koji su organizirani po nekom određenom principu u jednu strukturu koja se sastoji od više takvih istih, ali manjih struktura! Na primjer, niz se sastoji od elemenata koji mogu biti drugi nizovi. Za svaki od tih nizova također važi isto pravilio, odnosno i oni se sastoje od elemenata koji mogu biti drugi nizovi, i tako dalje. Ovdje treba uočiti to da smo za definiciju niza koristili pojam niza: Jedan niz može sadržavati drugi niz. Ova sama definicija je, prema tome, rekurzivna, pa smo i problem pretraživanja niza riješili koristeći rekurziju. Kao što je već rečeno, međutim, u nekim situacijama rekurzija nije najefikasnije rješenje. Čak i ako je problem definiran rekurzivno potrebno je imati uvid u svojstva takvih procesa (kao što smo napravili kod operacije *fib_rek*) prije nego što se odlučimo na iteraciju ili rekurziju, a isto tako može postojati i više rekurzivnih rješenja od kojih neka mogu biti efikasnija od drugih. Rekurzija općenito nije efikasnija od iteracije (iako može biti podjednako efikasna), ali operaciju može učiniti jasnijom, pogotovo ako direktno odgovara definiciji problema, kao što je bio slučaj s pretraživanjem nizova.

Zadaci za vježbu

1. Napišite rekurzivnu funkciju koja vraća zbroj svih vrijednosti u listi koja sadrži samo brojeve.

2. Napišite rekurzivnu funkciju *preokreni* koja preokreće listu.

```
preokreni([1, 2, 3, 4, 5])
=> [5, 4, 3, 2, 1]
```

3. Napišite rekurzivnu funkciju *palindrom* koja vraća True ako je zadani string palindrom (isti s lijeva i zdesna), u suprotnom vraća False.

4. Napišite funkciju *postoji* koja pronalazi zadani element u listi koja može sadržavati druge liste i vraća True ako taj element postoji, u suprotnom vraća False.

```
postoji('c', ['a', [['b', 'c'], 'd', ['e']], 'f'])
=> True
```

5. Napišite rekurzivnu funkciju *dubina* koja utvrđuje maksimalnu dubinu liste. Primjerice, lista `['a', ['b', 'c', [['d']], '[[[X]]]']]` ima dubinu 3 (na kojoj se nalazi element 'd'; element 'a' je na dubini 0, element `'[[[X]]]'` je string, ne lista).

Vježba 3: Vezana lista

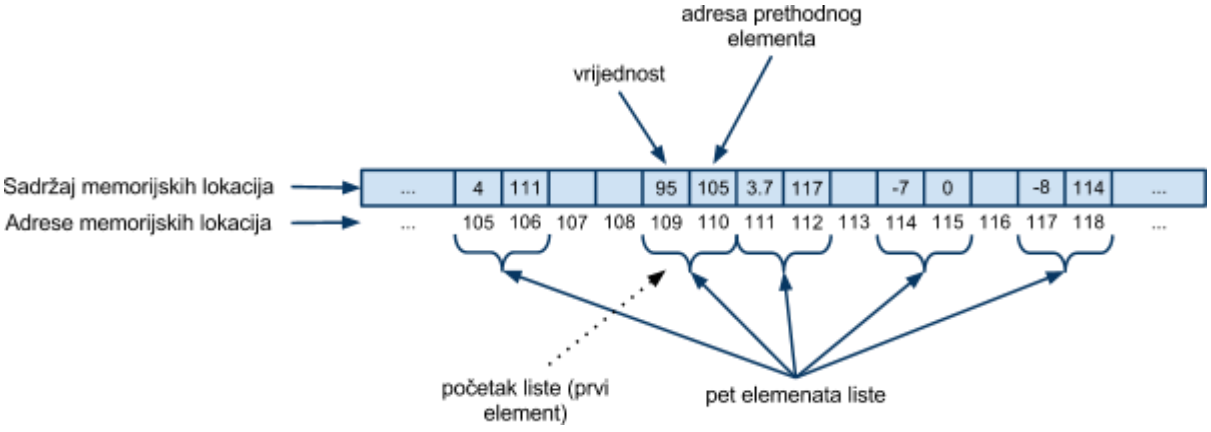
3.1 Uvod

Ako nam je efikasno dodavanje i uklanjanje elemenata važnije od efikasnog indeksiranja onda treba uzeti u obzir takozvane *vezane strukture podataka*. Kod takvih struktura podataka elementi nisu poslagani u memoriji jedan iza drugoga, niti je jedan dio memorije unaprijed rezerviran za određeni broj elemenata, nego se memorija rezervira samo za svaki element posebno. Nadalje, svaki od tih elemenata sadrži jednu ili više adresa na kojima se nalaze elementi te strukture koji su s njim povezani. Jedna takva struktura podataka je *vezana lista*.

Ova se struktura podataka zove tako zato jer predstavlja jednu listu elemenata, slično kao i nizovi, ali koji su organizirani na drugačiji način da bi se omogučilo efikasno umetanje i uklanjanje elemenata. Koncept vezane liste koja sadrži brojeve 95, 4, 3.7, -8 i -7 prikazan je na slici 5.5.

Slika 5.5 - Vezana lista.

Kod jednostavnih vezanih lista svaki je element jedan par, odnosno sastoji se od dva dijela: Prvi dio je vrijednost koju želimo smjestiti u listu, a drugi je adresa idućeg ili prethodnog elementa te liste. Ako se sada spustimo na razinu memorije, lista sa slike 5.5 mogla bi izgledati kako je prikazano na slici 5.6.



Slika 5.6 - Primjer vezane liste na razini memorije: Svaki se element sastoji od vrijednosti i adrese na kojoj se nalazi idući element liste.

Na toj slici treba primijetiti da elementi u memoriji nisu obavezno poslagani u istom redoslijedu u kojem se pojavljuju u listi. Jedini redoslijed koji je fiksan je redoslijed podatka i adrese idućeg elementa, i te dvije vrijednosti su u ovom primjeru postavljene na susjedne memorijske adrese. U principu, mora se znati gdje se nalazi podatak i adresa idućeg elementa liste, a na koji način ćemo to osigurati zavisi od detalja implementacije

same liste. Nadalje, vidimo da jedna adresa predstavlja početak liste, to jest samu listu, slično kao što je adresa prvog elementa u nizu predstavljala taj niz. Ako sada pratimo adrese, počevši od 109 na kojoj se nalazi prvi element, dobit ćemo ispravni redoslijed elemenata ove liste. Na primjer, kod elementa na paru adresa 109/110 vidimo da se idući element nalazi na adresi 105, a na paru adresa 105/106 vidimo da se idući element nalazi na adresi 111, nakon čega slijedi 117, i tako dalje. Prema tome, niz adresa koje ove podatke povezuju u listu su 109 (početna adresa) -> 105 -> 111 -> 117 -> 114 -> 0. Kod mnogih sistema adresa 0 (ili nulta adresa) se ne koristi za smještanje podataka, tako da ovdje može poslužiti kao oznaka za kraj liste. Nulta adresa se također često koristi kao specijalna konstanta koja označava da neka varijabla ne sadrži neku određenu adresu.

3.2 Implementacija vezane liste

Vezanu listu možemo implementirati u Pythonu na prilično jednostavan način. Definirat ćemo dvije klase, *Element* i *VezanaLista*. Objekti klase *Element* sadržavat će vrijednost i adresu elementa koji je prethodno bio dodan u listu, dok će objekt klase *VezanaLista* predstavljati samu tu strukturu podataka, s operacijama pristupa, dodavanja i uklanjanja elemenata. Klasa *Element* može se definirati ovako:

```
class Element:
    def __init__(self, podatak):
        self._prethodni = None
        self._podatak = podatak
```

Objekti ove klase služe nam samo kao nosioci podataka, bez ikakvih operacija. Za klasu *VezanaLista* definirat ćemo za početak četiri operacije, *dodaj*, za dodavanje novog elementa na kraj liste, *posljednji*, koja će uvijek davati posljednji dodani element, *prethodni*, koja nam daje prethodno dodani element liste počevši od nekog zadanog elementa, te *podatak*, koja nam daje vrijednost koji smo smjestili u listu:

```
class VezanaLista:
    def __init__(self):
        self._posljednji = None

    def dodaj(self, podatak):
        novi = Element(podatak)
        novi._prethodni = self._posljednji
        self._posljednji = novi

    def posljednji(self):
        return self._posljednji

    def prethodni(self, element):
        if element != None:
            return element._prethodni
        else:
            return None
```

```
    def podatak(self, e):
        return e._podatak
```

Operacija *dodaj* dodaje novi element na kraj liste. S obzirom da u listu želimo dodavati podatke bilo kakvog tipa, te podatke moramo "upakirati" s informacijama o adresi prethodnog elementa, pa zbog toga u prvom redu formiramo objekt klase *Element* koja će nositi taj podatak. U srednjem redu te operacije povezujemo prethodni element s ovim novim tako da varijabla *_prethodni* novog elementa ima adresu elementa koji je prije toga bio dodan, ili *None* ako se radi o elementu koji je prvi dodan u listu. Varijabla *_posljednji* uvijek sadrži adresu elementa koji je posljednji bio dodan u listu. Operaciju *dodaj* možemo ilustrirati ovako: Pretpostavimo da lista sadrži jedan ili više elemenata, te da se posljednji dodani element nalazi na adresi 125. Pretpostavimo sada da se novi element koji želimo dodati u listu nalazi na adresi 318 nakon izvršenja prvog reda ove operacije. Da bi ga dodali u listu moramo ga povezati s posljednjim elementom te liste, a taj se nalazi u varijabli *_posljednji*. Prema tome, varijabla *_prethodni* objekta klase *Element*, odnosno tog novog elementa, mora se postaviti na 125 jer to je adresa na kojoj se nalazi posljednji dodani element liste i to je ono što radimo u drugom redu operacije *dodaj*. U posljednjem, trećem, redu taj novi element sada postaje posljednji element liste, pa ga zato pridružujemo varijabli *_posljednji*. Ovdje treba napomenuti da ovakvu listu promatramo u obrnutom redoslijedu od onoga kojim smo u nju dodavali elemente, to jest posljednji dodani element je u stvari prvi element liste. Ako se vratimo na sliku 5.6 možemo zamisliti kako bi dodali element 95 u listu. Posljednji dodani element je 4 na adresi 105, pa prema tome varijabla *_posljednji* sadrži adresu 105. Varijabla *_prethodni* novog elementa mora sadržavati adresu 105 da bi se povezala s tim posljednjim elementom, pa joj zato u drugom redu pridružujemo *_posljednji*. Nakon toga ovaj novi element (na adresi 109) postaje posljednji dodani element pridruživanjem njegove adrese varijabli *_posljednji* u trećem redu.
Operacije *posljednji* i *prethodni* sačinjavaju jedan par operacija koji nam omogučava pristup elementima liste. Operacija *posljednji* jednostavno daje element koji je posljednji dodan u listu i pomoću njega možemo doći do svakog drugog elementa liste. Operacija *prethodni* za zadani element *e* daje prethodni element s kojim je ovaj direktno povezan preko njegove adrese (koja se nalazi u varijabli *_prethodni* elementa *e*).
Objekt klase *VezanaLista* sada možemo definirati na uobičajen način:

```
lista = VezanaLista()
```

Operacijom *dodaj* možemo dodati nekoliko elemenata u listu:

```
lista.dodaj('jedan')
lista.dodaj('dva')
lista.dodaj('tri')
```

Sada operacije *posljednji* i *prethodni* možemo koristiti za ispisivanje svih elemenata liste *lista*:

```
def ispiši_elemente(lista):
    e = lista.posljednji()
```

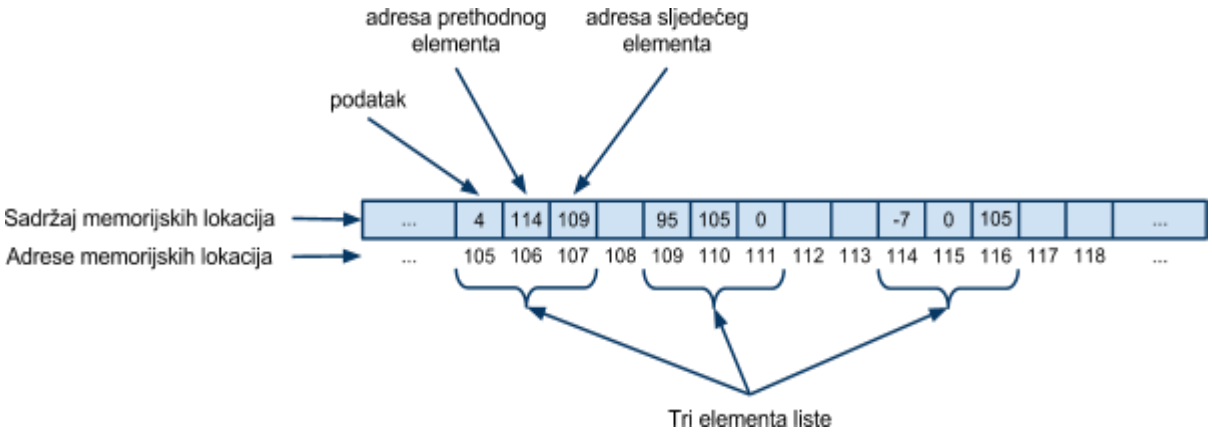
```
    while e != None:
        print(lista.podatak(e))
        e = lista.prethodni(e)

ispiši_elemente(lista)
=> tri
   dva
   jedan
```

Vidimo da su elementi ispisani u obrnutom redoslijedu od onog po kojem su dodani u listu. To je zato jer u klasi *VezanaLista* svaki novi element povezujemo s onim prethodno dodanim, odnosno novi element sadrži adresu onog prethodnog, tako da uvijek počinjemo od elementa koji je posljednji bio dodan u listu. Isto tako, petlja *while e != None*: ... kruži sve dotle dok operacija prethodni ne vrati *None* kao rezultat, što bi značilo da smo došli do elementa koji je prvi dodan u listu, pa bi stoga adresa za njegov prethodni element bila postavljena na *None* u varijabli *_prethodni* elementa *e*. Time bi varijabli *e* bila pridružena vrijednost *None*, što bi prekinulo izvršavanje ove petlje.

Kao jedna sporedna napomena, u klasama *Element* i *VezanaLista* varijable *_prethodni* i *_posljednji* počinju znakom "_". Taj je znak dio naziva varijable i ponekad se koristi za označavanje varijabli koje se koriste samo unutar klase, ali nisu predviđene za korištenje putem objekta te klase, kao što je *lista._posljednji*. Iako u ovom jednostavnom primjeru to ne dolazi do izražaja, kompleksnije klase često sadrže mnoge varijable i operacije koje se koriste za funkcioniranje same klase, ali koje ne predstavljaju dio njenog sučelja i koje bi u mnogim situacijama bilo nepreporučljivo koristiti direktno. U primjeru klase *VezanaLista* ne želimo da korisnik te klase direktno koristi varijablu *_prethodni* objekta klase *Element* jer na taj način nije osigurana provjera da taj element nije *None*. Sigurnije je koristiti operaciju *prethodni* klase *VezanaLista* jer ta operacija obavlja odgovarajuću provjeru. Isto tako, operacija *podatak* omogućava nam to da ne moramo znati koje su varijable definirane za klasu *Element*, odnosno koja od njih sadrži vrijednost koju smo dodali u listu. Na kraju ovog poglavlja vidjet ćemo bolju tehniku pristupa elementima liste i ostalih struktura podataka.

3.3 Varijanta vezane liste: Dvostruko-vezana lista

Lista kakvu smo implementirali klasom *VezanaLista* često se naziva *jednostruko-vezanom listom* zato jer svaki element povezujemo samo s jednim drugim elementom, to jest onim prethodnim. To znači da ako imamo adresu nekog elementa ne možemo direktno doći do sljedećeg elementa u listi, nego moramo krenuti ispočetka od onog posljednjeg. Općenito, kod jednostruko-vezane liste elementima možemo pristupati samo u jednom smjeru. Da bi omogućili dvosmjerno kretanje kroz listu svaki element mora imati dvije adrese - jednu za prethodni i drugu za sljedeći element. Ovo je ilustrirano na slici 5.7, gdje vidimo da je broj 4 srednji element ispred kojeg se nalazi broj -7 (na adresi 114), a broj koji ga slijedi je 95 (na adresi 109). Očito je da je broj -7 prvi element liste jer mu je adresa prethodnog elementa 0, dok je broj 95 posljednji element jer mu je adresa sljedećeg elementa 0.



Slika 5.7 - Primjer dvostruko-vezane liste, gdje se svaki element sastoji od podatka i dvije adrese - jedna za prethodni i druga za idući element liste.

Dvostruko-vezanu listu možemo implementirati s nekoliko jednostavnih promjena u klasama *Element* i *VezanaLista*. Kao što je rečeno, svaki element imat će dvije adrese - jednu za prethodni element i jednu za sljedeći:

```
class Element:
    def __init__(self, podatak):
        self._podatak = podatak
        self._prethodni = None
        self._sljedeći = None
```

U klasi *VezanaLista* moramo modificirati operaciju *dodaj* tako da veže novi element sa prethodnim (ako on postoji), ali isto tako da prethodni veže s novim. Prethodni element vezujemo s novim tako da mu postavimo varijablu *_sljedeći* na novi element, a novi vezujemo s prethodnim na isti način kao i do sada. Još jedna promjena biti će u operaciji (konstruktoru) *__init__* gdje ćemo dodati varijablu *_prvi* koja će se odnositi na prvi element liste. Ta će nam varijabla omogućiti da elementima liste pristupamo u istom redoslijedu po kojem su u nju dodavani. Uz to, vodit ćemo evidenciju o ukupnom broju elemenata u listi koristeći varijablu *_broj_elemenata* koju ćemo povećati za 1 kod dodavanja novog elementa i smanjiti za 1 kod uklanjanja nekog postojećeg elementa liste. Informacija o broju elemenata u listi trebat će nam kasnije za definiciju operacije indeksiranja liste. Nadalje, kao što imamo par operacija *posljednji* i *prethodni*, tako ćemo dodati novi par operacija, *prvi* i *sljedeći*, koji ćemo koristiti na isti način, ali sa svrhom prolaska kroz listu u suprotnom smjeru. Klasa *VezanaLista* sada izgleda ovako (pokazane su samo nove i modificirane operacije):

```
class VezanaLista:
    def __init__(self):
        self._prvi = None
        self._posljednji = None
        self._broj_elemenata = 0

    def dodaj(self, podatak):
```

```
        novi = Element(podatak)

        # veži novi element s prethodnim
        novi._prethodni = self._posljednji

        # ako već postoji jedan element veži ga s ovim novim
        if self._posljednji:
            self._posljednji._sljedeći = novi

        # ako je lista prazna upamti koji je prvi dodani element
        if self._prvi == None:
            self._prvi = novi

        # upamti posljednji element
        self._posljednji = novi

        # jedan element više u listi
        self._broj_elemenata += 1

        return novi     # za slučaj da nam treba ovaj novi objekt

    def prvi(self): return self._prvi

    def sljedeći(self, element):
        if element != None:
            return element._sljedeći
        else:
            return None
    …
```

Sada možemo napisati operaciju *ispiši_elemente_po_redu* koja ispisuje sve elemente zadane liste po redu po kojem su bili u nju dodavani:

```
def ispiši_elemente_po_redu(lista):
    e = lista.prvi()
    while e != None:
        print(lista.podatak(e))
        e = lista.sljedeći(e)

ispiši_elemente_po_redu(lista)
=> jedan
   dva
   tri
```

Ono što je još važnije je to da sada preko jednog elementa možemo direktno doći do prethodnog i sljedećeg elementa u listi. Uzmimo u obzir sljedeću operaciju koja daje element s traženim podatkom:

```
def nađi_element(lista, podatak):
    e = lista.prvi()
    while e:
        if lista.podatak(e) == podatak:
            return e
        e = lista.sljedeći(e)

    return None
```

Ovom operacijom možemo tražiti element u našoj listi koji sadrži podatak “dva”:

```
element = nađi_element(lista, 'dva')
```

Ako takav element postoji onda preko njega možemo lako doći do prethodnog ili sljedećeg elementa (ako takvi također postoje). Na primjer:

```
if element != None:
    if element._prethodni != None:
        # izluči podatak iz prethodnog elementa
        print('Prethodni element:', element._prethodni._podatak)
    else:
        print('Element "', element._podatak, '" je prvi element')

    if element._sljedeći != None:
        # izluči podatak iz sljedećeg elementa
        print('Sljedeći element:', element._sljedeći._podatak)
    else:
        print('Element "', element._podatak, '" je posljednji element')
```

Za objekt *element* koji sadrži tekstualnu vrijednost “dva” ovaj bi niz naredbi ispisao sljedeće:

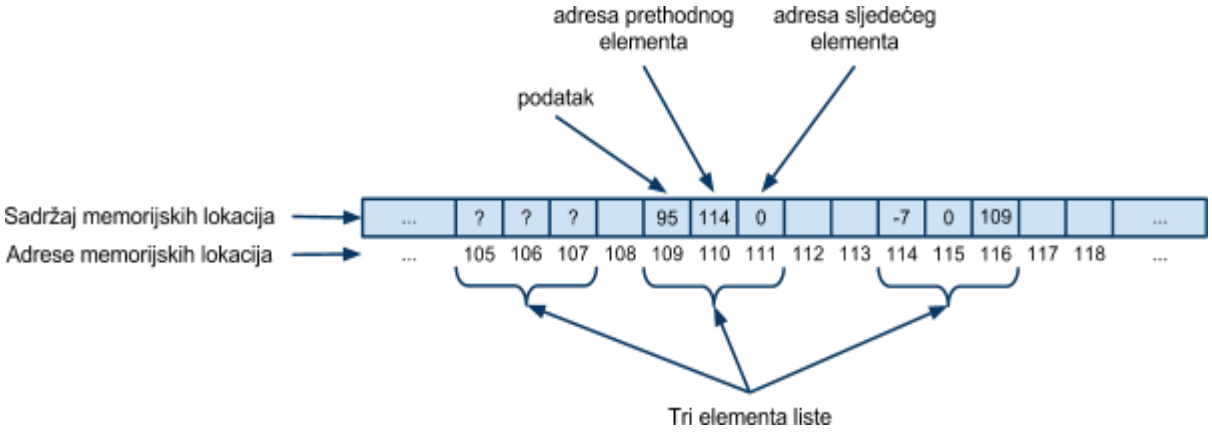
```
Prethodni element: jedan
Sljedeći element: tri
```

U gornjim *if* naredbama morali smo koristiti varijable *_prethodni*, *_sljedeći* i *_podatak* koje, kao što je prethodno rečeno, nismo namjeravali koristiti ovako direktno preko objekta dotične klase. Za sada, međutim, to će nam biti prihvatljivo za ilustraciju principa o kojima je ovdje riječ.

Iz gornjeg primjera vidimo da je dvostruko-vezana lista općenito fleksibilnija od jednostruko-vezane liste, uz manji nedostatak da zauzima nešto više memorije za isti broj elemenata zato jer svaki se od njih sastoji od tri umjesto dva dijela.

3.4 Uklanjanje elemenata iz vezane liste

Kod operacije *dodaj* u klasi *VezanaLista* možemo primijetiti da dodavanje novog elementa u listu ni na koji način na ovisi o veličini te liste. Za razliku od umetanja elemenata u nizove, kod vezanih lista ne moramo pomicati preostale elemente da bi napravili mjesto za novi element - novi element moramo samo povezati s njemu susjednim elementima. Drugim riječima, bez obzira na veličinu liste dodavanje novog elementa u vezanu listu uvijek zahtijeva isti broj operacija, a time i konstantno vrijeme.
Kao što je slučaj s dodavanjem novih elemenata u vezanu listu, isto važi i za uklanjanje postojećih elemenata iz takve liste. Uklanjanje nekog elementa iz vezane liste općenito zahtijeva samo to da "prespojimo" adrese tako da povežemo susjedne elemente.

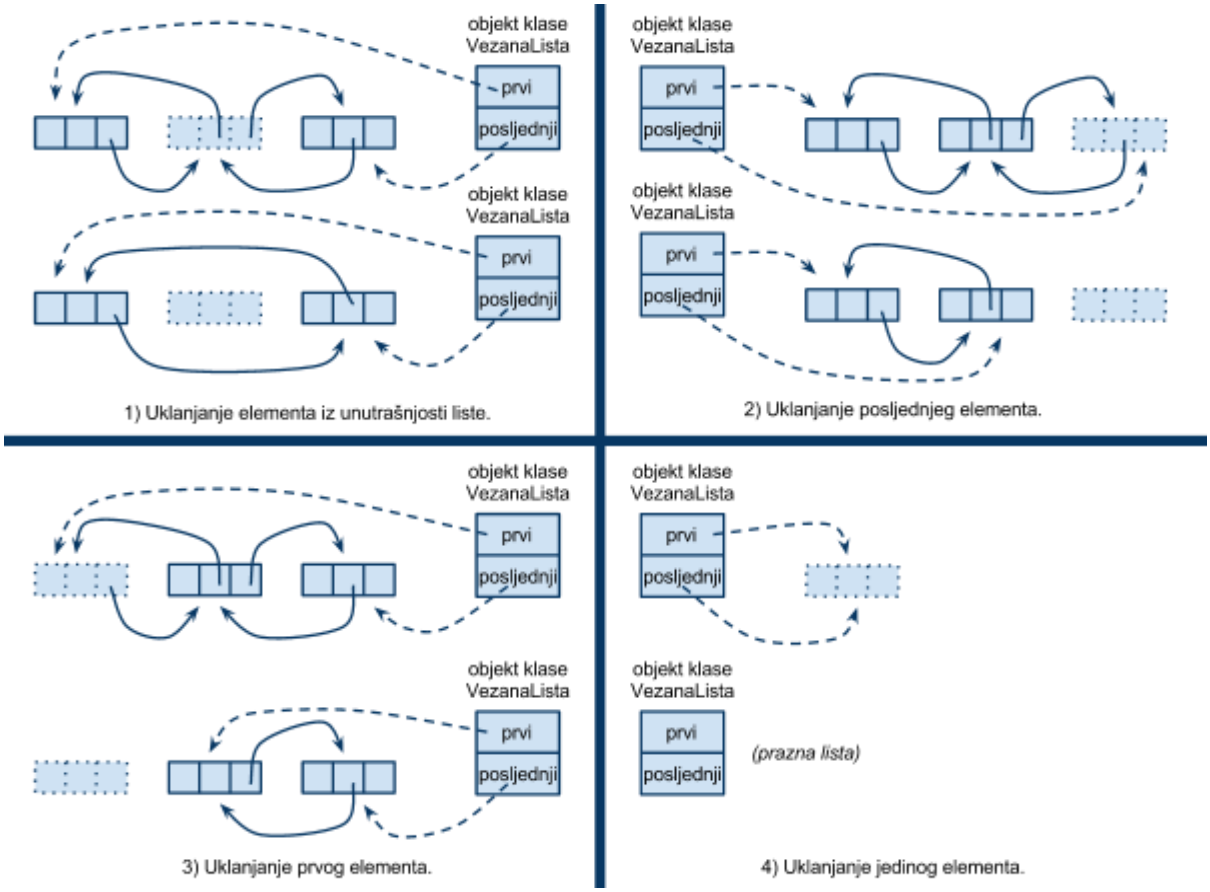


Slika 5.8 – Sadržaj memorije nakon što je uklonjen srednji element liste s adrese 105.

Na slici 5.8 vidimo rezultat uklanjanja broja 4 iz liste, gdje su brojevi -7 i 95 sada direktno povezani. U stvari, da bi uklonili bilo koji element iz vezane liste postoje četiri slučaja koja moramo uzeti u obzir, zavisno od toga na kakvoj se poziciji u listi taj element nalazi:

1. Ako element nije prvi niti posljednji onda njegov prethodni element moramo povezati s njemu sljedećim elementom;
2. Ako uklanjamo posljednji element onda njemu prethodni element postaje posljednji element liste;
3. Ako uklanjamo prvi element onda njemu sljedeći element postaje prvi element liste;
4. Ako uklanjamo jedini element u listi onda jednostavno varijable za prvi i posljednji element liste postavimo na *None*, što daje praznu listu.

Na slici 5.9 ilustrirana su ova četiri slučaja, gdje vidimo kako su veze među elementima modificirane nakon uklanjanja nekog elementa iz liste.



Slika 5.9 - Primjer uklanjanja elemenata iz liste: Gornja slika svakog od četiri dijela pokazuje veze elemenata prije, a donja poslije uklanjanja. Element koji se uklanja označen je točkastim crtama. Svaki se element sastoji od tri dijela - prvi s lijeva je podatak, drugi je adresa prethodnog, a treći adresa sljedećeg elementa.

Na osnovu ova četiri slučaja možemo za klasu *VezanaLista* definirati operaciju *ukloni*:

```
class VezanaLista:
    …
    def ukloni(self, element):
        # 1. uklanjamo element iz untrašnjosti liste
        if element._prethodni != None and element._sljedeći != None:
            element._prethodni._sljedeći = element._sljedeći
            element._sljedeći._prethodni = element._prethodni

        # 2. uklanjamo posljednji element
        elif element._prethodni != None:
            element._prethodni._sljedeći = None
            self._posljednji = element._prethodni

        # 3. uklanjamo prvi element
        elif element._sljedeći != None:
            element._sljedeći._prethodni = None
```

```
            self._prvi = element._sljedeći

        # 4. uklanjamo jedini element liste
        else:
            self._prvi = None
            self._posljednji = None

        # jedan element manje u listi
        self._broj_elemenata -= 1
```

Kod ove operacije treba uočiti to da se slučaj za uklanjanje elementa iz unutrašnjosti liste mora nalaziti na početku jer tu provjeravamo da li postoji i prethodni i sljedeći element, a nakon toga, to jest za ostale slučajeve, provjeravamo pojedinačno za prethodni i za sljedeći element, te na kraju slučaj za jedini element. Ako bi, na primjer, zamijenili redoslijed slučajeva 1 i 2 ova operacija ne bi radila dobro jer bi tada slučaj 2 bio korišten i za slučaj uklanjanja elementa iz unutrašnjosti liste koji također ima prethodni element.
Operacijom *ukloni* sada možemo ukloniti bilo koji element iz liste. Za početak možemo napraviti novu listu:

```
lista = VezanaLista()
lista.dodaj('jedan')
lista.dodaj('dva')
lista.dodaj('tri')

ispiši_elemente_po_redu(lista)
=> jedan
   dva
   tri
```

S obzirom da je svaki element ove liste objekt klase *Element* moramo imati varijablu koja sadrži taj objekt, odnosno njegovu adresu. Za tu svrhu koristit ćemo operaciju *nađi_element*:

```
element = nađi_element(lista, 'dva')
```

Varijabla *element* sada sadrži element koji možemo ukloniti, a on sadrži informacije o tome koji mu je prethodni i sljedeći element. Pomoću tih će informacija operacija *ukloni* na odgovarajući način povezati njemu susjedne elemente (ako oni postoje) i po potrebi odrediti koji je novi prvi ili posljednji element:

```
lista.ukloni(element)
```

Ovim je zadani element uklonjen iz liste, što vidimo ako ponovo ispišemo sve elemente ove liste:

```
ispiši_elemente_po_redu(lista)
=> jedan
```

```
tri
```

Iz ovih primjerima možemo vidjeti da je dodavanje i uklanjanje elemenata iz vezane liste vrlo efikasno. To znači da je vezana lista bolji izbor od nizova kada nam treba struktura podataka u koju ćemo često dodavati ili iz nje uklanjati elemente na početku ili oko sredine te liste.

3.5 Indeksiranje vezane liste

Ako način na koji su element organizirani u vezanoj listi usporedimo s nizovima, vidimo da je osnovna razlika u tome da su elementi u nizu poslagani u memoriji po redu, što omogučava interpreteru da na jednostavan način izračuna gdje se nalazi *n*-ti element. Kod vezanih lista to nije moguće zato jer elementi praktički mogu biti "porazbacani" posvuda po memoriji - jedino što ih povezuje su adrese prethodnog i sljedećeg elementa. Zbog toga, da bi do nekog elementa vezane liste došli pomoću njegovog indeksa, odnosno pozicije u listi, moramo slijediti niz adresa počevši od prvog elementa, s tim da brojimo koliko smo elemenata prošli. Operaciju indeksiranja, dakle, koju ćemo zvati *nti_element* možemo za našu klasu definirati ovako:

```
class VezanaLista:
    …
    def nti_element(self, n):
        if n + 1 > self._broj_elemenata or n < 0:
            raise GreškaIndeksiranjaListe()

        e = self._prvi
        while n > 0:
            n -= 1
            e = e._sljedeći

        return e

class GreškaIndeksiranjaListe(Exception): pass
```

Kod ove operacije prvo moramo osigurati to da indeks nije negativan broj ili veći od broja elemenata u listi, u kojem slučaju bi signalizirali grešku tipa *GreškaIndeksiranjaListe*, za što smo definirali odgovarajuću klasu. Ako je indeks ispravan onda za svaki element, počevši od prvog, jednostavno slijedimo adresu za sljedeći element sve dotle dok nam indeks ne padne na 0, čime efektivno brojimo elemente. Element na nekom indeksu u ovoj listi sada možemo dobiti ovako:

```
element = lista.nti_element(2)
lista.podatak(element)
=> tri
```

Da bi bili konzistentni u odnosu na nizove, elemente brojimo od 0, a ne 1, tako da je element na nultom indeksu u stvari prvi element.

Iz operacije *nti_element* vidimo na indeksiranje liste nije efikasna operacija, kao što je to slučaj kod nizova, jer kod vezane liste na osnovu zadanog indeksa ne možemo jednostavno izračunati gdje se nalazi dotični element, nego do njega moramo doći postepeno kroz druge elemente. To implicira da se operacija indeksiranja vezane liste ne izvršava u konstantnom vremenu, nego općenito ovisi o broju elemenata u listi. Prema tome, ako nam treba struktura podataka koju ćemo često indeksirati onda vezana lista nije dobar izbor.

3.6 Pretraživanje vezane liste

Iako su vezane liste po strukturi drugačije od nizova, one ne podrazumijevaju nikakav specifičan poredak elemenata koje sadrže, baš kao što je to slučaj i s nizovima. Vezane liste, dakle, također podržavaju linearno pretraživanje, koje smo već implementirali operacijom *nađi_vrijednost*. Ovdje ćemo takvu operaciju definirati kao dio klase *VezanaLista*:

```
class VezanaLista:
    …
    def nađi_vrijednost(self, v):
        e = self._prvi
        while e != None:
            if e._podatak == v:
                return e
            else:
                e = e._sljedeći

        return None
```

Kao što smo već vidjeli, nedostatak vezanih lista je u efikasnosti indeksiranja, gdje su nizovi puno efikasniji, tako da bi i binarno pretraživanje vezane liste bilo manje efikasno od takvog pretraživanja nizova (jer kod binarnog pretraživanja niza radimo s indeksima). Korištenje vezanih lista nam, prema tome, za pretraživanje ne nudi prednost u odnosu na nizove, čak i ako su elementi sortirani po veličini.
Još jedna korisna operacija koju možemo definirati za liste je pretraživanje elemenata na osnovu nekog zadanog kriterija, za što ćemo definirati operaciju *nađi*:

```
class VezanaLista:
    …
    def nađi(self, opr):
        e = self._prvi
        while e != None:
            if opr(e._podatak):
                return e
            else:
                e = e._sljedeći

        return None
```

Ovo je operacija višeg reda koja kao argument uzima operaciju koja je predviđena da služi kao kriterij za element koji tražimo. Ova operacija radi slično kao operacije *prvi* i *selekcija* koje smo vidjeli u drugom poglavlju i koristimo je na isti način. Na primjer, da bi u listi našli vrijednost *'dva'* ili *'tri'* sada možemo pisati ovako:

```
x = lista.nađi(lambda v: v == 'dva' or v == 'tri')
```

Ako takva vrijednost postoji onda će varijabli *x* biti pridružen prvi element koji sadrži jednu ili drugu vrijednost.
S obzirom da je korisno znati koliko elemenata sadrži neka lista možemo dodati jednostavnu operaciju *veličina* koja daje broj elemenata u listi:

```
class VezanaLista:
    …
    def veličina(self): return self._broj_elemenata
```

Iako je varijabla *_broj_elemenata* dostupna direktno preko objekta klase *VezanaLista*, na ovaj se način jasnije vidi koje su operacije definirane za objekte ove klase. Kao što je već rečeno, varijable kao što su *_broj_elemenata*, *_sljedeći*, *_prethodni* i slično, koristit ćemo uglavnom u operacijama koje su definirane unutar dotične klase, dok će nam operacije kao što je *veličina* predstavljati sučelje za rad s objektima takvih klasa.

Zadaci za vježbu

1. Implementirajte *cirkularnu* (kružnu) jednostruko vezanu listu kod koje posljednji element ima vezu s onim prvim. Neka vaša klasa ima metodu *iduci* koja slijedi veze među elementima kružne liste i koja uvijek daje iduci element liste (to jest, nikad ne dolazi do kraja).

2. Napišite funkciju *postoji_ciklus* koja vraća True ako u jednostruko vezanoj listi postoji ciklus, u suprotnom vraća False.

3. Napišite funkciju *prva_tri* koja u obliku ntorke vraća prva tri elementa jednostruko vezane liste i funkciju *zadnja_tri* koja vraća zadnja tri elementa jednostruko vezane liste (pod pretpostavkom da lista ima dovoljan broj elemenata). Ako lista nema dovoljan broj elemenata treba vratiti ntorku s dva, jednim ili 0 elemenata (ako je lista prazna).

4. Implementirajte dvostruko vezanu listu s metodom *indeks(n)* koja vraća element na indeksu *n* te liste. Nadalje, s obzirom da je indeksiranje neefikasno kod vezanih lista neka metoda *indeks* „kešira“ (sprema) dohvaćene vrijednosti tako da ako se od te metode traži element na ponovljenom indeksu onda ona treba samo vratiti prethodno spremljeni element (bez da ponovo prolazi kroz listu).

Vježba 4: Stogovi i redovi

4.1 Uvod

Vezana lista je jedna općenita struktura podataka koja može poslužiti kao osnova za neke druge strukture podataka, od kojih ćemo ovdje spomenuti dvije: *stog* i *red*.[2] Stog je struktura podataka koja se, kao i vezana lista ili niz, sastoji od jednog niza podataka, ali kod koje elemente možemo dodavati i uklanjati samo na jednom kraju. Ova struktura podataka karakterizirana je s dvije specifične operacije koje se zovu *push*, za dodavanje elemenata, i *pop*, za uklanjanje. Kod stoga element koji je posljednji bio umetnut je prvi koji se može iz njega izvaditi, odnosno posljednji element koji je ušao je prvi koji ide van. Ova struktura podataka, iako vrlo jednostavna, ima veliku važnost u programiranju, što ćemo vidjeti kasnije u nekim primjerima pretraživanja.
Stog možemo implementirati na više načina, na primjer, kao niz, jednostruko-vezanu ili dvostruko-vezanu listu. Za našu ćemo implementaciju koristiti vezanu listu. Ovdje je važno imati u vidu to da vezanu listu već samu po sebi možemo koristiti kao stog, koristeći operacije *dodaj* i *ukloni*. Ono po čemu se stog razlikuje od vezane liste su operacije koje su za njega definirane, a ne način na koji povezujemo elemente. Mogučnosti stoga kao strukture podataka više su ograničene u odnosu na vezanu listu, ali što se tiče unutrašnjeg mehanizma po kojem on radi vezanu listu i dalje možemo koristiti kao osnovu za stog. Klasu *Stog*, prema tome, možemo definirati ovako (zajedno s klasom za grešku):

```
class Stog:
    def __init__(self):
        self._lista = VezanaLista()

    def push(self, element):
        self._lista.dodaj(element)

    def pop(self):
        e = self._lista.posljednji()
        if e == None:
            raise PrazanStog()
        else:
            self._lista.ukloni(e)
            return e._podatak

    def vrh(self):
        if self.prazno_p():
            raise PrazanStog()
        else:
            return self._lista.posljednji()._podatak

    def prazno_p(self):
```

[2]Eng. stog je *stack*, a red je *queue*.

```
        return self._lista.veličina() == 0


class PrazanStog(Exception): pass
```

Kao što smo napravili kod vezane liste, u nekim slučajevima moramo signalizirati grešku i kod stogova. Ako pokušavamo izvući element iz praznog stoga ili dobiti element na vrhu stoga koji je prazan onda signaliziramo grešku tipa *PrazanStog*. U klasi *Stog* vidimo da za sve operacije u stvari koristimo samo operacije vezane liste, ali da ostale operacije vezane liste nisu na raspolaganju kod ove strukture podataka. Drugim riječima, klasom *Stog* samo smo ograničili i prilagodili funkcionalnost vezane liste za neku drugu, sličnu strukturu podataka, gdje se tada takva struktura podataka, kao što je *Stog*, ponekad naziva *adapterom*.
S obzirom da se podaci u stog umeću i vade samo na jednom kraju, njima možemo pristupati samo u redoslijedu obrnutom od onoga u kojem su bili dodavani. Na primjer, definirat ćemo novi objekt klase *Stog* u koji ćemo dodati tri elementa:

```
stog = Stog()
stog.stavi('jedan')
stog.stavi('dva')
stog.stavi('tri')
```

Sada iz tog stoga možemo vaditi jedan po jedan element, sve dotle dok u stogu ima elemenata:

```
while not stog.prazno_p():
    e = stog.makni()
    print(e)
```

Ovom petljom dobili bi sljedeće:

```
tri
dva
jedan
```

Operacijom *vrh* možemo dobiti element koji je posljednji bio stavljen na stog, bez da ga time uklonimo.
Jedno pitanje koje se često nameće u ovakvim slučajevima je to da li je ovdje trebalo koristiti nasljeđivanje ili kompoziciju objekata (kao što smo ovdje napravili). Specifično, da li bi bilo bolje da klasa *Stog* nasljeđuje od klase *VezanaLista*, čime bi ih postavili u odnos "stog *je* vezana lista", ili da smo koristili odnos "stog *sadrži* vezanu listu" u kakvom su sada? Ovdje je dovoljno usporediti funkcionalnost, odnosno operacije koje su definirane za ove dvije strukture podataka da bi vidjeli da se u stvari radi o dva različita koncepta - u suprotnom stog ne bi niti postojao kao zasebna struktura podataka. Pored, možda, operacije kojom dodajemo elemente na kraj stoga i predikata za prazan stog, ova struktura podataka nema ništa zajedničko s vezanom listom, kao što je pretraživanje, indeksiranje, uklanjanje bilo kojeg elementa, prethodni i sljedeći element i slično. Dakle, očito je da stog nije

varijanta vezane lista, a činjenica da za implementaciju stoga ovdje koristimo vezanu listu ne mijenja ništa - isto smo tako mogli koristiti i niz, što ne bi značilo da je stog niz jer se operacije za nizove i dalje razlikuju od onih za stogove. Da smo klasu *Stog* definirali tako da nasljeđuje od klase *VezanaLista* tada bi naš stog podržavao sve ove druge operacije koje konceptualno nisu definirane za stogove. Kao što je rečeno na početku, ovdje moramo uzeti u obzir konceptualne karakteristike struktura podataka, odnosno operacije koje su za njih definirane, a ne način na koje su te strukture podataka implementirane.

Još jedna jednostavna struktura podataka je *red*, koja je slična stogu, ali je razlika u tome da elemente dodajemo na jednom kraju, a vadimo ih na drugom. Za redove su definirane dvije operacije:

- dodavanje u red, koja se označava sa *enqueue*
- vađenje iz reda, koja se označava sa *dequeue*

Isto kao što smo napravili kod stoga, za signalizaciju grešaka ovdje ćemo koristiti klasu *PrazanRed*, koju ćemo signalizirati u slučaju pokušaja vađenja elementa i ispitivanja sljedećeg elementa (odnosno onog na početku) praznog reda. Isto tako, red možemo implementirati koristeći vezanu listu:

```
class Red:
    def __init__(self):
        self._lista = VezanaLista()

    def enqueue(self, element):
        self._lista.dodaj(element)

    def dequeue(self):
        e = self._lista.prvi()
        if e == None:
            raise PrazanRed()
        else:
            self._lista.ukloni(e)
            return e._podatak

    def vrh(self):
        if self.prazno_p():
            raise PrazanRed()
        else:
            return self._lista.prvi()._podatak

    def prazno_p(self):
        return self._lista.veličina() == 0


class PrazanRed(Exception): pass
```

Vidimo da je jedina razlika u odnosu na klasu *Stog* u operaciji *dequeue*, gdje ovdje uvijek uzimamo prvi element liste, a ne posljednji. To znači da će element koji je posljednji dodan u red biti i posljednji koji će iz njega izaći. Na primjer:

```
red = Red()
red.enqueue('jedan')
red.enqueue('dva')
red.enqueue('tri')

while not red.prazno_p():
    e = red.dequeue()
    print(e)
```

Gornja petlja ispisuje elemente po redu po kojem su bili dodani:

```
jedan
dva
tri
```

Operacija *vrh* je logički ista kao i istoimena operacija klase *Stog*, to jest ona daje element koji će sljedeći izaći iz reda, odnosno onaj sljedeći koji bi dala operacija *dequeue*.
U idućem ćemo dijelu vidjeti neke primjene stogova i redova.

Zadaci za vježbu

1. Implementirajte stog pomoću jednostruko-vezane liste.

2. Implementirajte red pomoću dvostruko-vezane liste.

3. Ovaj zadatak demonstrira jednu upotrebu stoga kod izračunavanja aritmetičkih izraza. Treba napisati program koji izračunava aritmetičke izraze kao što je 2+3*7. Radi jednostavnosti možete pretpostaviti da nema zagrada i da su brojevi jednoznamenkasti cijeli brojevi, te da su dozvoljene samo operacije +, -, *, / i ^ (potenciranje), s tim da program mora voditi računa o prioritetu operacija: najviši prioritet ima ^, slijede * i /, te na kraju + i -. Program treba raditi tako da prvi izraz transformira u takozvani *postfiksni oblik*, što bi u gornjem slučaju bilo [2, 3, 7, *, +]. Sada se izraz u ovakvom obliku može izračunati u dva koraka (počevši od polaznog postfiksnog oblika): [2, 3, 7, *, +] => [2, 21, +] => [23]. Dakle, ako idemo s lijeva na desno u ovakvoj listi, za svaki operator koji s lijeva ima barem dvije vrijednosti izračunamo njegov rezultat. Taj postupak ponavljamo dok u listi ne ostane samo jedna vrijednost, što je rezultat cijelog izraza.
Postupak transformiranja izraza u posfiksni oblik može se napraviti upotrebom stoga za operatore. Sada prolaskom kroz zadani izraz s lijeva na desno svaku znamenku dodamo u Pythonovu listu. Kada naiđemo na operator provjerimo prvi element na stogu operatora. Ako je taj stog prazan dodamo trenutni operator do kojeg smo došli. Ako stog nije prazan onda usporedimo prioritete: Ako operator na stogu ima niži prioritet od trenutnog operatora onda trenutni operator dodamo na stog. Ako operator na stogu ima veći ili jednak prioritet

kao i trenutni operator onda sve operatore sa stoga dodamo u listu, a trenutni operator stavimo na stog. Ovaj postupak ponavljamo dok ne dođemo do kraja izraza. Ako na kraju izraza stog operator nije prazan onda te operatore dodamo na kraj liste u redoslijedu u kojem se nalaze na stogu. Kao primjer, neka je izraz 5+2^2-3*2. Ispod su koraci koji pokazuju promjene stoga, izlazne liste i preostalog (nepročitanog) dijela zadanog izraza.

Stog: [] *vrh stoga je posljednji element*
Preostali izraz: "5+2^2-3*2"
Izlazna lista: []

Stog: []
Preostali izraz: "+2^2-3*2"
Izlazna lista: [5]

Stog: [+]
Preostali izraz: "2^2-3*2"
Izlazna lista: [5]

Stog: [+]
Preostali izraz: "^2-3*2"
Izlazna lista: [5, 2]

Stog: [+, ^] *operacija "^" ima viši prioritet od „+" pa ide na stog*
Preostali izraz: "2-3*2"
Izlazna lista: [5, 2]

Stog: [+, ^]
Preostali izraz: "-3*2"
Izlazna lista: [5, 2, 2]

Stog: [-] *operacija "-" ima niži prioritet od "^" pa sve sa stoga ide u listu, a „-„ ide na stog*
Preostali izraz: "3*2"
Izlazna lista: [5, 2, 2, ^, +]

Stog: [-]
Preostali izraz: "*2"
Izlazna lista: [5, 2, 2, ^, +, 3]

Stog: [-, *]
Preostali izraz: "2"
Izlazna lista: [5, 2, 2, ^, +, 3]

Stog: [-, *]
Preostali izraz: ""
Izlazna lista: [5, 2, 2, ^, +, 3, 2]

Stog: [] *došli smo do kraja izraza pa sve sa stoga ide u listu*
Preostali izraz: ""
Izlazna lista: [5, 2, 2, ^, +, 3, 2, *, -]

Na kraju izlazna lista sadrži zadani izraz u postfiksnom obliku, stog je prazan i cijeli izraz je pročitan. Sada bi konačni rezultat dobili redukcijom izlazne liste: [5, 2, 2, ^, +, 3, 2, *, -] => [5, 4, +, 3, 2, *, -] => [9, 3, 2, *, -] => [9, 6, -] => [3].

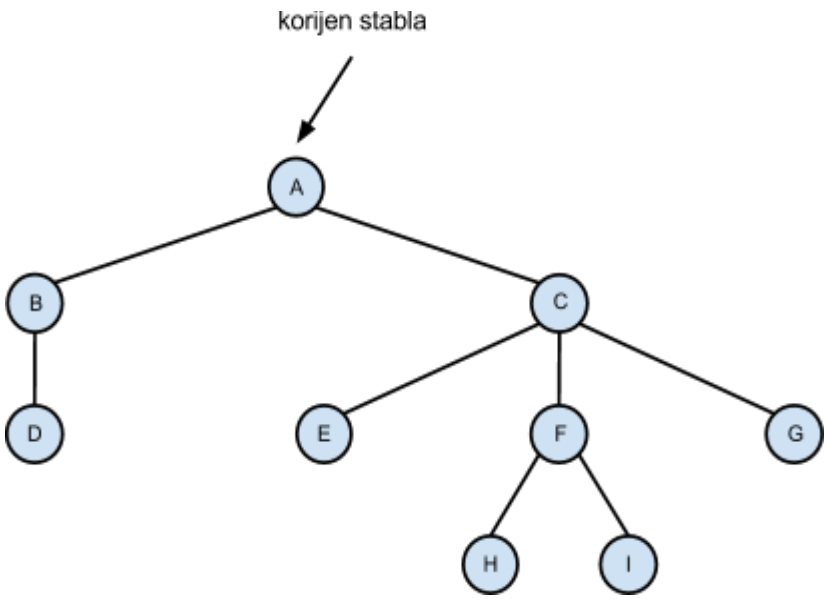
Vježba 5: Stabla

5.1 Uvod

Kao što smo vidjeli, binarno pretraživanje nizova ima značajno bolje performanse od linearnog. Međutim, dodavanje i uklanjanje elemenata otežava činjenica da su nizovi kompaktna struktura podataka koja zahtijeva da se postojeći elementi često pomiču na druge pozicije unutar niza. Vezane liste, s druge strane, omogučavaju efikasno dodavanje i uklanjanje elemenata, ali s cijenom neefikasnog indeksiranja koje usporava binarno pretraživanje.

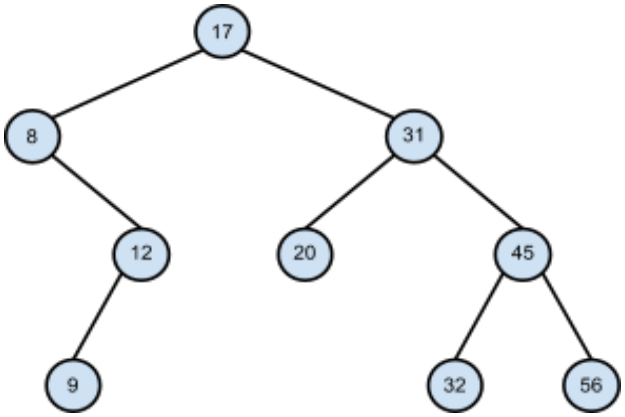
Jedna struktura podataka koja je pogodna za binarno pretraživanje, a istovremeno ima dobre performanse dodavanja i uklanjanja elemenata zove se *stablo* i njena je implementacija zasnovana na sličnim tehnikama koje smo koristili kod vezane liste. Ova se struktura podataka tako zove zato jer po svom izgledu podsjeća na stablo, što vidimo na slici 5.10. Uobičajeno je da se struktura stabla prikazuje s korijenom na vrhu, tako da se ono grana odozgo prema dole. Iako po svojoj formi jednostavna, ova struktura podataka ima iznenađujuće veliki značaj u informatici i računarstvu. U drugom smo poglavlju, na primjer, vidjeli kako se pozivi operacija mogu prikazati pomoću stabla kada smo govorili o rekurzivnim operacijama. Općenito, stablo se može koristiti i kao model nekog procesa ili odnosa među objektima i podacima, a ne samo kao kolekcija podataka. U ovom ćemo dijelu strukturu stabla proučavati isključivo iz perspektive organiziranja i pretraživanja podataka.

Na slici 5.10 vidimo stablo koje se sastoji od jednog *čvora* A koji predstavlja *korijen* tog stabla i skupine drugih čvorova koji se iz njega neposredno ili posredno granaju. Također vidimo da se stablo u stvari sastoji od više drugih, manjih stabala, gdje svakog od njih nazivamo *podstablom*. Na primjer, čvor C i čvorovi koji se iz njega granaju predstavljaju jedno podstablo, a isto važi i za čvor F, ali i za krajnje čvorove kao što su D, E, H, I i G, koji se nazivaju *listovima* ovog stabla. Ako se neki čvor grana u dva podstabla kaže se da taj čvor ima *lijevo* i *desno* podstablo, kao čvorovi A i F na slici. Čvorovi koji se granaju iz jednog čvora nazivaju se *sljedbenicima* tog čvora, a sam taj čvor je njihov *prethodnik*. Na primjer, čvorovi H i I su sljedbenici čvora C i kaže se da su oni *direktni* sljedbenici čvora F. Isto tako, čvor F je direktan prethodnik čvorova H i I.



Slika 5.10 – Stablo s devet čvorova.

Jedna vrsta stabla koja je od posebnog značaja u programiranju je takozvano *binarno stablo*. Kod takvog se stabla svaki čvor grana u *najviše* dva čvora. Na primjer, stablo na slici 5.10 nije binarno zato jer se čvor C grana u tri čvora - E, F i G. Binarna stabla se često koriste kao struktura podataka pogodna za binarno pretraživanje. Binarno stablo koje se koristi u svrhu binarnog pretraživanja zove se *stablo za binarno pretraživanje*. Ako zamislimo da svi čvorovi binarnog stabla sadrže podatke koji se međusobno mogu postaviti u nekakav poredak, onda svako stablo za binarno pretraživanje mora zadovoljavati sljedeće svojstvo: Ako je P neki čvor takvog stabla onda za svaki sljedbenik S u lijevom podstablu tog čvora vrijedi $S \leq P$, a za svaki sljedbenik S u desnom podstablu tog čvora vrijedi $S \geq P$. Drugim riječima, za svaki čvor stabla za binarno pretraživanje s njegove su lijeve strane svi elementi koji nisu veći od njega, a s njegove desne strane svi oni koji nisu manji. Na slici 5.11 prikazano je jedno stablo za binarno pretraživanje. Vidimo da su čvorovi koji su manji od korijena s njegove lijeve strane, a svi oni veći s njegove desne strane. To također važi i za sve ostale čvorove ovakvog stabla, odnosno za svako podstablo. Ako, na primjer, pogledamo čvor 31 onda vidimo da je 20 s njegove lijeve strane, a 32, 45 i 56 s njegove desne strane. Ako sada zamislimo kako bi tražili neku vrijednost u ovakvom stablu možemo uočiti sličnost s binarnim pretraživanjem koje smo primijenili kod nizova. Princip binarnog pretraživanja kod ovakvog stabla je isti. Ako, na primjer, tražimo vrijednost 32 onda počnemo od korijena, i s obzirom da je $32 > 17$ nastavljamo tražiti u desnom podstablu. Sljedeća vrijednost je 31, tako da opet nastavljamo u desnom podstablu čvora 31, gdje se nalazi vrijednost 45. Sada tražimo dalje u lijevom podstablu čvora 45 zato jer je $32 < 45$, čime dolazimo do tražene vrijednosti 32.



Slika 5.11 – Stablo za binarno pretraživanje.

Binarno stablo možemo implementirati slično kao i vezanu listu, koristeći jedan objekt koji će predstavljati čvorove stabla i drugi koji će sve te objekte objedinjavati u jednu cjelovitu strukturu. Za čvorove stabla koristit ćemo klasu *Čvor*:

```
class Čvor:
    def __init__(self, podatak):
        self._podatak = podatak
```

```
        self._prethodnik = None
        self._lijevo = None
        self._desno = None
```

Ova je klasa slična klasi *Element* koju smo koristili kod vezanih lista; varijabla *_podatak* predstavlja vrijednost za dotični čvor i možemo joj pridružiti bilo kakvu vrijednost koju možemo uspoređivati s drugim vrijednostima u stablu, a varijable *_prethodnik*, *_lijevo* i *_desno* predstavljaju veze čvora sa svojim direktnim prethodnikom, te direktnim lijevim i desnim sljedbenikom. Objekte ove klase definiramo na uobičajeni način:

```
čvor = Čvor('tekstualna vrijednost')
```

Za objekt koji će predstavljati stablo koristit ćemo klasu *Stablo*:

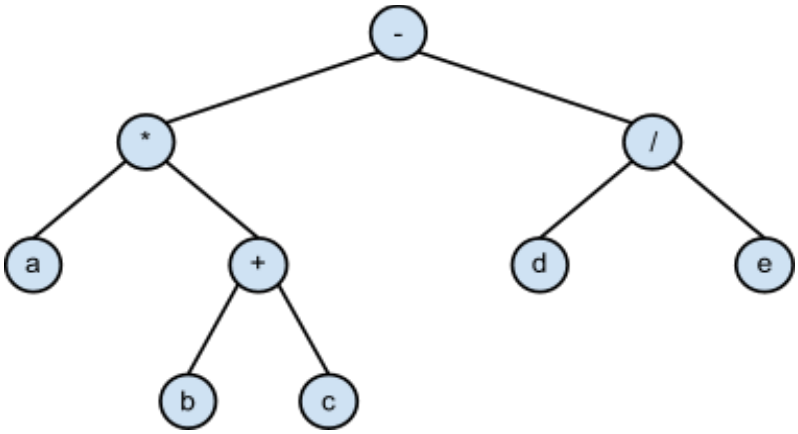
```
class Stablo:
    def __init__(self):
        self._korijen = None
```

Operacija inicijalizacije je jednostavna: Korijen stabla na početku sadrži *None*, što označava prazno stablo. Novo stablo definiramo ovako:

```
stablo = Stablo()
```

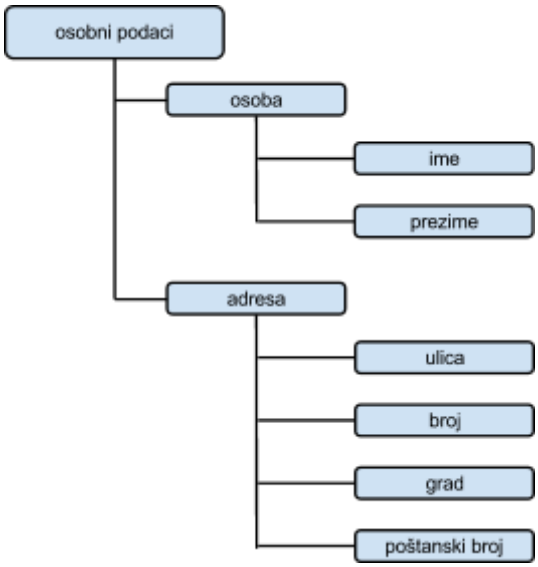
5.2 Dvije osnovne tehnike prolaženja kroz stablo

Iako se stabla za binarno pretraživanje koriste prvenstveno radi efikasnog pretraživanja podataka, stablo kao struktura podataka može se koristiti (i često se koristi) i u druge svrhe. Na primjer, aritmetički izraz *a*(b+c)-d/e* može se prikazati stablom kao na slici 5.16. Takvo se stablo zove *apstraktno sintaksno stablo* i ima veliku primjenu kod izrade prevodioca i interpretera za programske jezike.



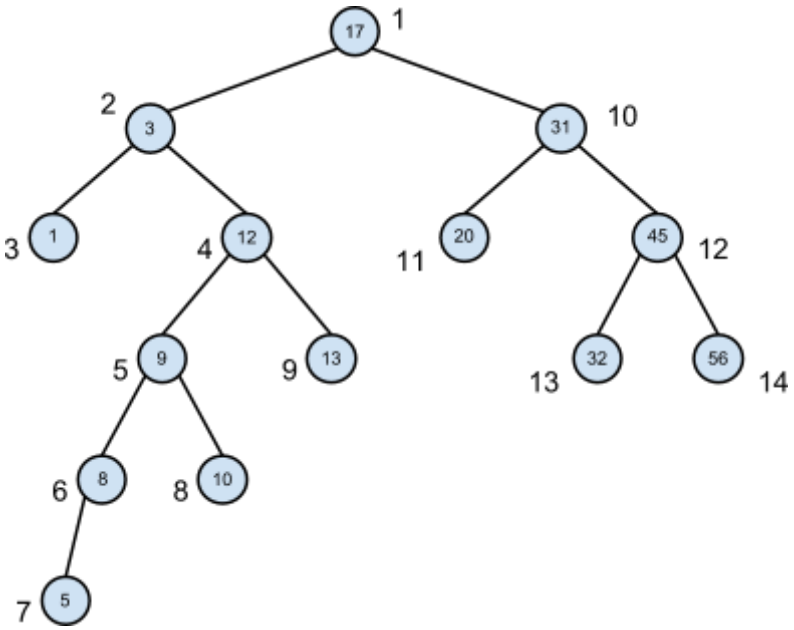
Slika 5.16 - Apstraktno sintaktičko stablo izraza a*(b+c)-d/e.

Vidimo da je svaki list u ovom stablu neka varijabla ili vrijednost ovog izraza, a unutrašnji čvorovi predstavljaju operacije. Ovakvo stablo, prema tome, ne sadrži vrijednosti koje su u njemu raspoređene po veličini (kao što je to bio slučaj kod binarnog stabla za pretraživanje), nego su te vrijednosti raspoređene prema odnosima u kojima se međusobno nalaze. Na primjer, operacija "+" odnosi se na vrijednosti *b* i *c*, to jest *b* i *c* su dva operanda operacije "+", a rezultat te operacije je operand za operaciju "*", zajedno s vrijednošću *a*. Nadalje, rezultat operacija "*" i "/" (s operandima *d* i *e*) su operandi operacije "-" čiji je rezultat ujedno i rezultat ovog izraza. Sada vidimo zbog čega je struktura stabla izuzetno pogodna za prikaz ovakvih izraza: Svaki čvor predstavlja jedan podizraz (s tim da su i vrijednosti *a*, *b*, *c*, …, također trivijalni izrazi) čiji rezultat je operand za širi izraz čiji je on dio, kao što je podizraz *b + c* dio podizraza *a * (b + c)*. Općenito, stablo je često idealna struktura podataka za prikaz odnosa među podacima gdje se jedan veći dio sastoji od jednog ili više manjih dijelova koji imaju istu strukturu. Kao još jedan primjer, osobni podaci mogu se prikazati kao na slici 5.17 na kojoj je stablo prikazano tako da se širi odozgo prema dole, a produbljuje s lijeva na desno.



Slika 5.17 – Struktura osobnih podataka.

Kod stabala općenito postoje dvije osnovne tehnike kojima možemo doći do svakog čvora u stablu. Jedna od tih tehnika zove se *pretraživanje u dubinu* kod koje dolazimo do čvorova, počevši od korijena prema onim dubljim, s lijeva na desno. Na slici 5.18 prikazano je stablo koje pokazuje redoslijed po kojem dolazimo do čvorova pretraživanjem u dubinu. Vidimo da se prvo spuštamo po lijevoj strani, u dubinu dokle god možemo, nakon čega se vraćamo na najbliži čvor koji je imao i desnu stranu, te ovaj postupak ponavljamo za taj čvor, i tako dok ne prođemo sve čvorove u stablu.



Slika 5.18 - Pretraživanje u dubinu.

Operacija kojom možemo implementirati pretraživanje stabla u dubinu jako je slična operaciji kojom smo ispisivali vrijednosti stabla u sortiranom redoslijedu:

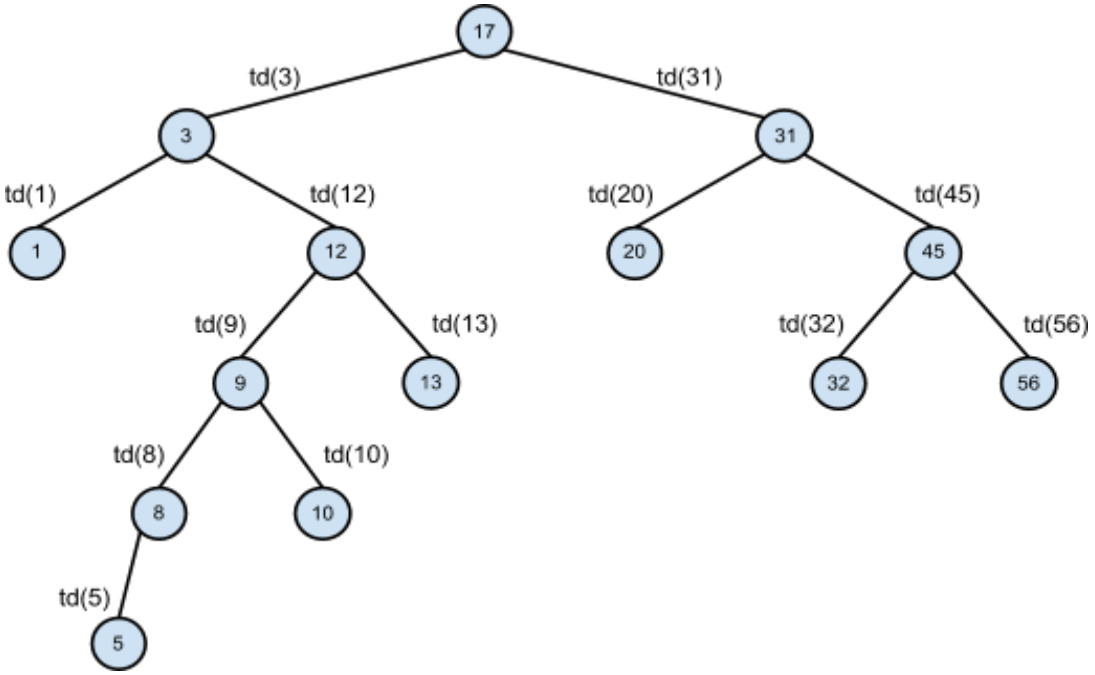
```
def prođi_u_dubinu_rek(čvor):
    print('*', čvor._podatak, end = '   ')
    if čvor._lijevo:
        prođi_u_dubinu_rek(čvor._lijevo)
    if čvor._desno:
        prođi_u_dubinu_rek(čvor._desno)

prođi_u_dubinu_rek(st._korijen)
=> 17   3   1   12   9   8   5   10   13   31   20   45   32   56
```

U operaciji *prođi_u_dubinu_rek* vidimo da prvo idemo u lijevo dokle možemo, nakon toga u desno, čime je opet moguće da idemo u lijevo podstablo tog desnog čvora, i tako dalje. Prvi rekurzivan poziv, *prođi_u_dubinu_rek(čvor._lijevo)* vodit će na lijevu stranu, a drugi poziv, *prođi_u_dubinu_rek(čvor._desno)*, vodit će na desnu stranu kada se vratimo iz onog prvog poziva. Općenito, za svaki rekurzivan poziv za lijevu stranu pamtimo jedan rekurzivan poziv za desnu stranu. Ako ime ove operacije skratimo na *td* onda na slici 5.19 vidimo kako su se odvijali njeni rekurzivni pozivi. Kod poziva *td(3)* sljedeći na redu bio je poziv za desno podstablo, *td(31)*, a kod poziva *td(1)* sljedeći na redu bio je poziv za desno podstablo, *td(12)*. S obzirom da je poziv *td(12)* bio posljednji dodan na takvu "listu čekanja", to je bio prvi poziv kojim se nastavilo nakon poziva *td(1)*. Ovaj proces ilustriran je donjom tabelom. Stupac označen sa *td(čvor._lijevo)* pokazuje za koji se čvor izvodi rekurzivni poziv, a stupac označen sa *td(čvor._desno)* pokazuje koji rekurzivni poziv će uslijediti nakon povratka iz prethodnog poziva, odnosno poziva za lijevi čvor. Vidimo da broj poziva u tom stupcu u stvari raste - to je zato jer svaki put kada idemo u lijevo podstablo moramo se vratiti u najbliže desno podstablo nakon što više ne možemo ići ulijevo. S obzirom da je na putu po

lijevoj strani stabla bilo više desnih čvorova, budući pozivi za njih su se na taj način nagomilali. Na primjer, nakon čvora 17 išlo se na lijevi čvor, 3, ali se morao "upamtiti" čvor 31, koji je desni čvor čvora 17. Nakon što se došlo u čvor 3, išlo se dalje u čvor 1, ali se morao upamtiti čvor 12, koji je desni čvor čvora 3. Nakon čvora 1 više se nije moglo ići u lijevo, pa se išlo na posljednji upamćeni čvor, 12. S obzirom da je čvor 12 sada iskorišten (odnosno izveden je rekurzivan poziv *td(12)*), on se više ne pamti, pa se tako više i ne nalazi u desnom stupcu. S tog čvora opet se išlo u lijevo na čvor 9, a pamtio se čvor 13 koji je desni čvor čvora 12, i tako dalje. U čvorovima 9, 8 i 5 ništa se ne mijenja zato jer možemo ići samo na lijevu stranu. Nakon čvora 5 posljednji upamćeni čvor je 10, nakon kojeg se opet išlo na prije njega upamćeni čvor, 13. Nakon 13 također nemamo kuda, pa se ide na prethodno upamćeni čvor, 31, koji je u stvari bio upamćen kao nastavak nakon čvora 17. Desnu stranu stabla sada prolazimo na isti način, gdje na kraju dolazimo do čvora 56. Nakon tog čvora nemamo kuda, a isto tako nema više desnih čvorova kojima bi se moglo nastaviti dalje, tako da time završava ovaj proces prolaženja kroz stablo. Posljednji stupac ove tabele pokazuje iz kojeg se poziva operacija vraća kada se više ne može ići ni lijevo ni desno, odnosno kada se dođe do lista stabla.
Kod ovog postupka treba imati u vidu to da su sva ova "pamćenja" desnih čvorova bila izvedena automatski, koristeći mahanizam poziva operacija koji smo opisali u drugom poglavlju.

korak	***td(čvor._lijevo)***	***čvor***	***td(čvor._desno)***	***ako smo došli do lista stabla***
1	*td(17)*	*17*	*td(31)*	
2	*td(3)*	*3*	*td(12), td(31)*	
3	*td(1)*	*1*	***td(12)**, td(31)*	*vraćamo se iz poziva td(1)*
4	***td(12)***	*12*	*td(13), td(31)*	
5	*td(9)*	*9*	*td(10), td(13), td(31)*	
6	*td(8)*	*8*	*td(10), td(13), td(31)*	
7	*td(5)*	*5*	***td(10)**, td(13), td(31)*	*vraćamo se iz poziva td(5) i td(8)*
8	***td(10)***	*10*	***td(13)**, td(31)*	*vraćamo se iz poziva td(10) i td(9)*
9	***td(13)***	*13*	***td(31)***	*vraćamo se iz poziva td(13), td(12) i td(3)*
10	***td(31)***	*31*	*td(45)*	
11	*td(20)*	*20*	***td(45)***	*vraćamo se iz poziva td(20)*
12	***td(45)***	*45*	*td(56)*	
13	*td(32)*	*32*	***td(56)***	*vraćamo se iz poziva td(32)*
14	***td(56)***	*56*		*vraćamo se iz poziva td(56), td(45), td(31)*

U stupcu označenom *td(čvor._desno)* treba uočiti redoslijed u kojem su budući pozivi nanizani. Njih dodajemo s lijeve strane, a s te iste strane ih i uzimamo, odnosno uvijek uzimamo onaj prvi slijeva. Ako se prisjetimo strukture podataka koju zovemo stog, možemo vidjeti da buduće pozive u ovom stupcu tretiramo kao da su dodavani u stog. Tehnički, unutar Pythonovog interpretera ovi pozivi i jesu "poslagani" na način koji odgovara stogu, to jest na isti način koji je ilustriran u dijelu 2.5. Sada ćemo vidjeti da za pretraživanje stabla možemo stog kao strukturu podataka koristiti direktno, odnosno u njemu možemo eksplicitno pamtiti čvorove stabla od kojih se pretraživanje treba nastaviti.



Slika 5.19 - Pretraživanje u dubinu.

Iako je rekurzija prirodno riješenje za ovaj postupak (jer se jedno stablo sastoji od više manjih stabala), ona nije neophodna. U prethodnoj smo tabeli vidjeli da svaki put kada se išlo na lijevi čvor, pamtio se poziv za desni čvor jer je taj poziv slijedio onaj za lijevi čvor u operaciji *prođi_u_dubinu_rek*. U toj smo operaciji u stvari iskoristili postojeći mehanizam Pythonovog interpretera (mehanizam koji odgovara pravilima poziva operacija) koji vodi evidenciju o tome kako teku ovi pozivi operacija, to jest kako se iz jednog poziva vraćamo u prethodni. Taj mehanizam, naravno, radi isto za bilo koji poziv operacije, bez obzira na to je li taj poziv rekurzivan ili nije. Ovdje se sada možemo pitati kako bi napisali operaciju za prolaženje kroz stablo bez korištenja rekurzije? Takva operacija je prilično jednostavna ako slijedimo sličan postupak koji je prikazan gornjom tabelom, ali umjesto da se pamte pozivi operacije za desni čvor mi ćemo pamtiti sam taj desni čvor koristeći stog kao strukturu podataka koju smo prethodno definirali. Ovu operaciju, koju ćemo zvati *prođi_u_dubinu_iter*, definirat ćemo ovako:

```
def prođi_u_dubinu_iter(čvor):
    stog = Stog()
    stog.push(čvor)     # dodaj početni čvor
    while not stog.prazno_p():
        x = stog.pop()       # uzmi sljedeći čvor
        print(x._podatak, end = '   ')
        if x._desno: stog.push(x._desno)
        if x._lijevo: stog.push(x._lijevo)

prođi_u_dubinu_iter(stablo._korijen)
=> 17   3   1   12   9   8   5   10   13   31   20   45   32   56
```

U ovoj operaciji koristimo petlju gdje u svakoj iteraciji izvadimo čvor sa stoga, ispišemo vrijednost koju sadrži, i zatim na stog stavimo njegov desni i lijevi čvor (ako ti čvorovi postoje). Ovaj postupak ponavljamo sve dotle dok stog ne postane prazan. Stog na početku sadrži samo početni čvor koji je zadan (što je u ovom primjeru korijen) zato jer moramo početi s nekim čvorom.
U primjeru ovih dviju operacija, *prođi_u_dubinu_rek* i *prođi_u_dubinu_iter*, treba dobro uočiti i razumijeti važnost stoga kao strukture podataka. Njegovo osnovno svojstvo je to da je posljednji element koji je dodan onaj prvi koji ide van. Na tom principu je baziran mehanizam poziva operacija, kao što smo vidjeli u drugom poglavlju, iako tamo nismo govorili o stogu kao strukturi podataka. Na primjer, u lancu poziva operacija $a \Rightarrow b \Rightarrow c$, nakon *c* moramo se vratiti u *b*, a nakon *b* vraćamo se u *a*. Ako sada zamislimo da kod svakog poziva operacije na stog stavimo operaciju u koju se moramo vratiti iz tog poziva, možemo vidjeti da ćemo dobiti takav isti proces. U ovom primjeru, prije poziva *b* na stog dodamo *a*, te prije poziva *c* na stog dodamo *b*. S obzirom da je *b* posljednji dodan na stog, on prvi ide van, to jest po povratku iz operacije *c* vraćamo se prvo u operaciju *b*. Sada je na stogu ostala samo operacija *a*, pa se prema tome po povratku iz *b* vraćamo u *a*. Upravo je to mehanizam koji nam omogućava pretraživanje stabla u dubinu, zato jer kada više ne možemo ići na lijevu stranu idemo na najbližu desnu stranu koju smo prethodno prošli, a poziv za tu desnu stranu bio bi onaj koji je na vrhu stoga. Kao što smo vidjeli, u gornjoj tabeli stupac *td(čvor._desno)* u stvari sadrži pozive operacija koji su dodavani kao na stog, to jest prvi poziv s lijeva je onaj koji je sljedeći na redu. U operaciji *prođi_u_dubinu_iter* koristimo taj isti princip, samo što ovdje eksplicitno dodajemo i uklanjamo čvorove sa stoga. Ono što je važno kod ove operacije je to da koristimo stog, pa zbog toga dobijamo isti rezultat kao i kod operacije *prođi_u_dubinu_rek*.

Još jedan način prolaženja kroz stablo je *prolaženje u širinu*, što je ilustrirano na slici 5.20 na kojoj vidimo da se takvim pretraživanjem spuštamo po "slojevima" stabla. Jedna, na prvi pogled iznenađujuća činjenica vezana za ovakvo pretraživanje je ta da ga možemo implementirati na gotovo isti način kao i iterativno pretraživanje u dubinu, ali s jednom izmjenom: Umjesto stoga koristimo red! Ovo je pokazano operacijom *prođi_u_širinu*:

```
def prođi_u_širinu(čvor):
    red = Red()
    red.enqueue(čvor)
    while not red.prazno_p():
        x = red.dequeue()
        print(x._podatak, end = '   ')
        if x._lijevo: red.enqueue(x._lijevo)
        if x._desno: red.enqueue(x._desno)

prođi_u_širinu(stablo._korijen)
=> 17   3   31   1   12   20   45   9   13   32   56   8   10   5
```

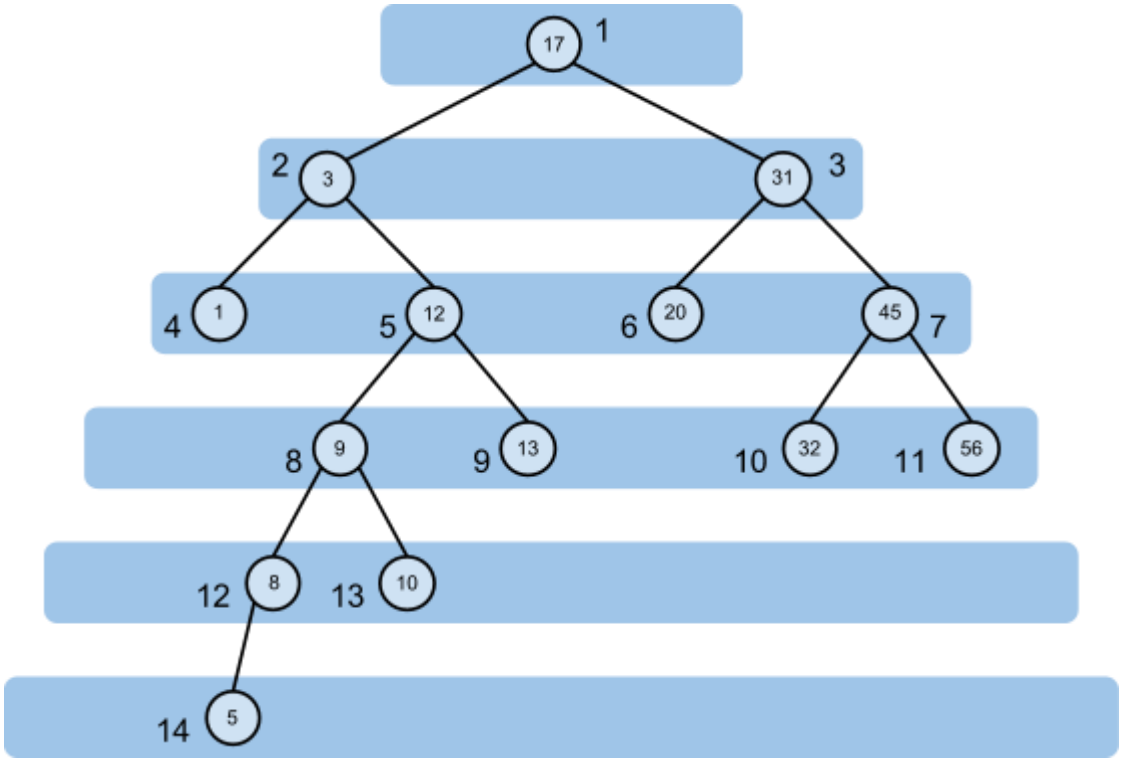
U ovoj smo operaciji također zamijenili redoslijed dviju *if* naredbi zato da bi čvorovi stabla bili ispisani s lijeva na desno (umjesto s desna na lijevo), ali to u principu nije važno. Kao što je kod stoga posljednji dodani element onaj koji prvi ide van, kod reda je obratno: Posljednji dodani element posljednji ide van. Na primjer, kada dođemo u čvor 3 u red

dodajemo lijevi i desno čvor, odnosno 1 i 12. Međutim, dok smo bili u čvoru 17 u red smo dodali čvorove 3 i 31, pa će prema tome čvor 31 doći na red prije čvorova 1 i 12, što ne bi bio slučaj kod stoga, jer ako su čvorovi 1 i 12 posljednji dodani u red, onda će oni i posljednji doći na red. U sljedećoj tabeli ilustriran je ovaj postupak.

korak	***ispisani čvor***	***sadržaj reda***
1	17	17, 3, 31
2	3	3, 31, 1, 12
3	31	31, 1, 12, 20, 45
4	1	1, 12, 20, 45
5	12	12, 20, 45, 9, 13
6	20	20, 45, 9, 13
7	45	45, 9, 13, 32, 56
8	9	9, 13, 32, 56, 8, 10
9	13	13, 32, 56, 8, 10
10	32	32, 56, 8, 10
11	56	56, 8, 10
12	8	8, 10, 5
13	10	10, 5
14	5	5

U stupcu *sadržaj reda* nalaze se čvorovi koje dodajemo na kraj reda, a prvi čvor s lijeva je onaj koji je sljedeći na redu. Na primjer, za čvor 17 dodajemo njegov lijevi, a zatim desni sljedbenik, to jest 3 i 31, dok za čvor 3 dodajemo čvorove 1 i 12, i tako dalje.

Na primjerima operacija *prođi_u_dubinu_iter* i *prođi_u_širinu* dobro se vidi razlika između stoga i reda. Stog kao koncept ima na neki način temeljnu važnost u računarstvu, ne samo kao struktura podataka nego i kao mehanizam na kojem je zasnovan princip izvršavanja programa na kompjuteru, a time i jedan od osnovnih principa programskih jezika. Kao što smo vidjeli u drugom poglavlju, pozivi operacija rade na principu stoga, što je realizirano na razini interpretera ili prevodioca za dotični programski jezik.



Slika 5.20 - Pretraživanje u širinu.

5.3 Dodavanje novih elemenata u stablo za binarno pretraživanje

Dvije osnovne operacije svake strukture podataka su dodavanje novih elemenata i pretraživanje, pa ćemo početi s operacijom *dodaj*:

```
class Stablo:
    …

    def dodaj(self, v):
        čvor = Čvor(v)
        c = self._korijen
        if c == None:
            # Stablo je prazno pa je prvi dodani čvor automatski
            # i korijen tog stabla.
            self._korijen = čvor
        else:
            # dođi do odgovarajućeg lista stabla i s njim poveži novi čvor

            while c != None:
                p = c
                if v < c._podatak:
                    c = c._lijevo
                else:
                    c = c._desno

            čvor._prethodnik = p    # p je prethodnik novog čvora
            if v < p._podatak:
                # novi čvor je lijevi sljedbenik svog prethodnika
                p._lijevo = čvor
            else:
                # novi čvor je desni sljedbenik svog prethodnika
                p._desno = čvor
```

U ovoj operaciji prvo definiramo novi objekt za čvor koji će nositi vrijednost koju želimo dodati u stablo. S obzirom da u stablu već mogu postojati čvorovi koji su prethodno dodani, da bi dodali novi čvor moramo pronaći odgovarajući list u stablu na koji ćemo ga nadovezati. Na primjer, ako u stablo prikazano na slici 5.11 želimo dodati vrijednost 14 onda bi se ta vrijednost nadovezala na čvor s brojem 12 kojem bi bila desni sljedbenik. Radimo, dakle, isto kao da tražimo broj 12, uzimajući u obzir njegovu vrijednost na osnovu koje u svakom čvoru odlučujemo nastavljamo li dalje s lijevim ili desnim podstablom, i tako dok ne dođemo do kraja, za što je zadužena *while*-petlja. Varijabla *p* sadrži prethodnik svakog čvora do kojeg se došlo i nju koristimo nakon petlje da bi novi čvor povezali s njegovim prethodnikom, a isto tako da bi taj prethodnik povezali s njegovim novim sljedbenikom.
Da bi vidjeli kako stablo izgleda definirat ćemo jednu jednostavnu operaciju koja će stablo prikazivati vertikalno, s tim da će svaki čvor uvijek imati dva direktna sljedbenika. Ako takav sljedbenik ne postoji za neki čvor na njegovom će se mjestu nalaziti znak “*”. S obzirom da je svako podstablo također stablo, operaciju *ispiši* možemo definirati rekurzivno:

```
def ispiši(cvor, n = 1):
    c = čvor
    print(' ' * n, c._podatak)
    if c._desno:
        ispiši(c._desno, n + 4)
    else:
        print(' ' * (n + 4), '*')

    if c._lijevo:
        ispiši(c._lijevo, n + 4)
    else:
        print(' ' * (n + 4), '*')
```

Ova operacija nije dio klase *Stablo* (iako bi, naravno, mogla biti). Sada u naše stablo možemo dodati vrijednosti koje smo koristili za stablo na slici 5.11:

```
for vrijednost in [17, 8, 31, 12, 20, 45, 9, 32, 56]:
    st.dodaj(vrijednost)
```

Operacija *ispiši* pokazuje nam kako ovo stablo izgleda:

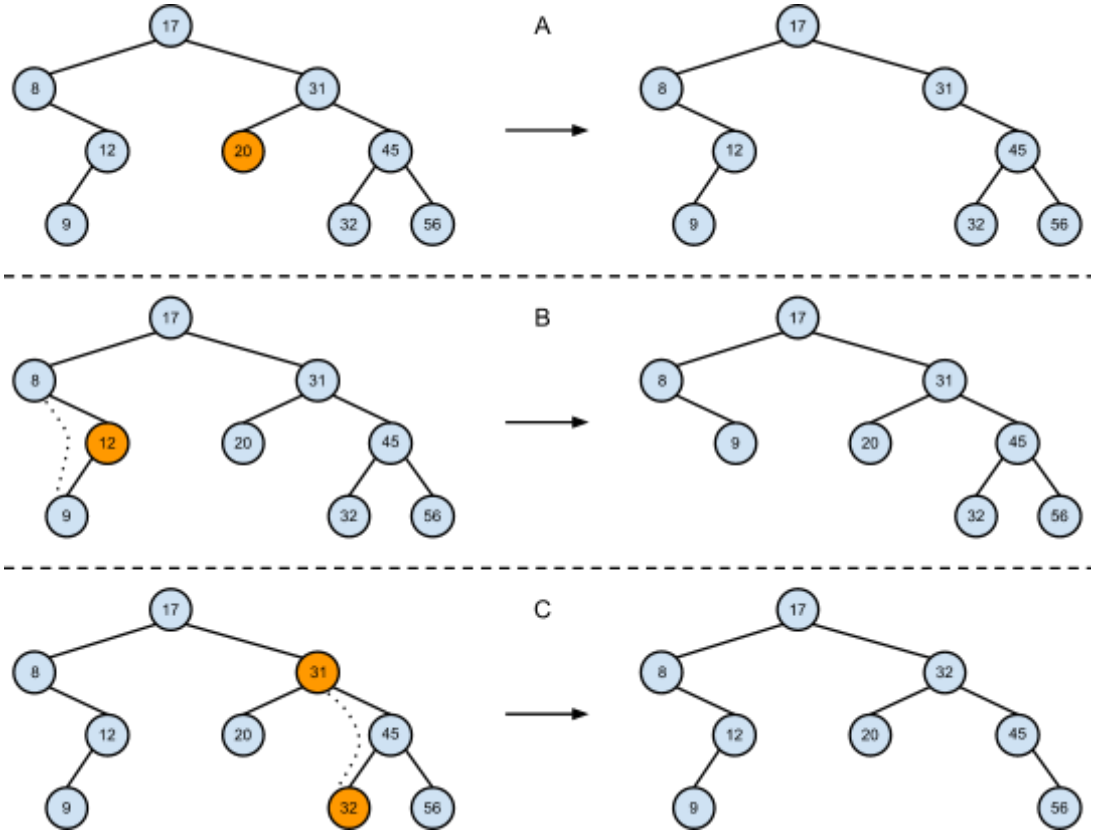
```
ispiši(st._korijen)
=> 17
      31
          45
              56
                  *
                  *
              32
                  *
                  *
          20
              *
              *
      8
          12
              *
              9
                  *
                  *
          *
```

Vidimo da je to isto stablo kao ono na slici 5.11, samo je prikazano tako da se širi vertikalno, a produbljuje horizontalno.

5.4 Uklanjanje elemenata iz stabla za binarno pretraživanje

Operaciju *sljedeći* sada možemo koristiti u definiciji operacije *ukloni* koja uklanja zadani čvor. Ova operacija je kompleksnija od ostalih koje smo do sada definirali zato jer uklanjanje nekog čvora iz stabla zahtjeva da neke ostale čvorove povežemo na odgovarajući način tako da osnovno svojstvo stabla za binarno pretraživanje ostane sačuvano. Na slici

5.12 vidimo tri slučaja koja moramo uzeti u obzir: Slučaj A odnosi se na uklanjanje elementa koji nema sljedbenika, kao što je čvor 20 na slici, odnosno elementa koji je list stabla. U tom slučaju jednostavno uklonimo taj čvor, s tim da kod njegovog prethodnika moramo ukinuti vezu s njim. U slučaju B uklanjamo čvor koji ima samo jednog sljedbenika, kao što je čvor 12. U tom slučaju povežemo njegovog prethodnika s njegovim sljedbenikom. U slučaju C uklanjamo čvor koji ima dva sljedbenika, kao što je čvor 31. Ovaj slučaj je interesantan zbog toga što ovdje u stvari ne uklanjamo čvor 31 nego čvor koji nosi prvu sljedeću vrijednost po veličini, a to je čvor 32. U ovom slučaju samo kopiramo podatke iz čvora 32 u čvor 31, a zatim uklonimo čvor 32, što ima isti učinak kao da smo uklonili čvor 31.



Slika 5.12 - Uklanjanje elementa iz stabla za binarno pretraživanje. Na slici A uklanjamo element 20, na slici B element 12, a na slici C element 31.

Sada na osnovu ovih pravila možemo definirati operaciju *ukloni*:

```
class Stablo:
    …
    def ukloni(self, čvor):
        sljedbenik = None
        uklonjeni_čvor = None

        # Nađi čvor koji treba ukloniti - taj čvor je onaj koji je zadan
        # (ako on ima najviše jednog direktnog sljedbenika) ili čvor koji
        # nosi sljedeću vrijednost po veličini (ako zadani čvor ima dva
```

```
        # direktna sljedbenika).
        if čvor._lijevo == None or čvor._desno == None:
            uklonjeni_čvor = čvor
        else:
            uklonjeni_čvor = self.sljedeći(čvor)

        # poveži ostale čvorove kako treba

        if uklonjeni_čvor._lijevo != None:
            sljedbenik = uklonjeni_čvor._lijevo
        else:
            sljedbenik = uklonjeni_čvor._desno

        # Čvor koji slijedi uklonjeni čvor (ako postoji) mora dobiti
        # istog prethodnika koji je bio prethodnik uklonjenom čvoru.
        if sljedbenik != None:
            sljedbenik._prethodnik = uklonjeni_čvor._prethodnik

        # Ako uklonjeni čvor nema prethodnika to znači da uklanjamo
        # korijen stabla, tako da prethodno ustanovljeni sljedbenik
        # postaje novi korijen.
        if uklonjeni_čvor._prethodnik == None:
            self._korijen = sljedbenik
        else:
            # Ako uklanjamo lijevi čvor onda je njegov lijevi sljedbenik
            # novi lijevi sljedbenik njegovog prethodnika, inače njegov
            # desni sljedbenik postaje desni sljedbenik njegovog
            # prethodnika.
            if uklonjeni_čvor == uklonjeni_čvor._prethodnik._lijevo:
                uklonjeni_čvor._prethodnik._lijevo = sljedbenik
            else:
                uklonjeni_čvor._prethodnik._desno = sljedbenik

        # Ako smo morali ukloniti neki drugi čvor od onoga
        # koji je zadan u parametru onda moramo kopirati podatke
        # iz uklonjenog čvora u onaj zadani.
        if uklonjeni_čvor != čvor:
            čvor._podatak = uklonjeni_čvor._podatak

        # dobro je znati koji čvor je u stvari bio uklonjen
        return uklonjeni_čvor
```

Iako ovu operaciju nismo rastavili na tri dijela koja direktno odgovaraju pojedinačnim slučajevima sa slike 5.12, komentari u kodu govore dovoljno o tome kako ona radi. Sljedeći primjer pokazuje rezultat uklanjanja čvora 17 iz stabla na slici 5.11:

```
st.ukloni(st.nađi(17))
```

```
=> 20
      31
          45
              56
                  *
                  *
              32
                  *
                  *
          *
      8
          12
              *
              9
                  *
                  *
          *
```

5.5 Pretraživanje stabla za binarno pretraživanje

Operacija za pronalaženje vrijednosti u stablu za binarno pretraživanje je jednostavna i zasnovana je na sličnom postupku koji smo koristili za operaciju *dodaj*. Ovu ćemo operaciju zvati *nađi*:

```
class Stablo:
    …
    def nađi(self, v):
        čvor = self._korijen
        while čvor != None:
            if čvor._podatak == v:
                return čvor
            elif v < čvor._podatak:
                čvor = čvor._lijevo
            else:
                čvor = čvor._desno

        return None
```

Za ovu operaciju koristimo prethodno opisani princip binarnog pretraživanja: Počnemo od korijena s kojim uspoređujemo traženu vrijednost. Na osnovu te usporedbe nastavljamo s lijevim ili desnim podstablom. Ovo ponavljamo sve dotle dok ne naiđemo na traženi element ili dok ne dođemo do kraja stabla, u kojem slučaju operacija vraća *None* kao rezultat, što znači da traženi element ne postoji u stablu. Ako nađemo čvor koji sadrži traženu vrijednost onda kao rezultat dajemo sam taj čvor zato da bi ga (ako bude potrebno) mogli koristi za druge operacije sa stablom, kao što je uklanjanje.
Na slici 5.11 možemo vidjeti da je kod stabla za binarno pretraživanje lako pronaći najveći i najmanji element. Da bi došli do najvećeg elementa jednostavno slijedimo isključivo desnu stranu, počevši od korijena, dok za najmanji element slijedimo isključivo lijevu stranu. Ovim postupkom lako saznamo da je najmanji element ovog stabla 8, a najveći 56. Operacije za pronalaženje najvećeg ili najmanjeg elementa u stablu mogu biti korisne, a i poslužit će nam kod operacije uklanjanja, pa ćemo ih stoga ovdje definirati:

```
class Stablo:
    …
    def najmanji(self, čvor):
        sljedbenik = čvor

        # slijedi samo lijevu stranu
        while sljedbenik._lijevo != None:
            sljedbenik = sljedbenik._lijevo

        return sljedbenik

    def najveći(self, čvor):
        sljedbenik = čvor
```

```
        # slijedi samo desnu stranu
        while sljedbenik._desno != None:
            sljedbenik = sljedbenik._desno

        return sljedbenik
```

Ako nas, na primjer, zanima koja je najmanja i najveća vrijednost u stablu možemo koristiti ove dvije operacije tako da im zadamo korijen stabla kao početni čvor:

```
st.najmanji(st._korijen)._podatak
=> 8

st.najveći(st._korijen)._podatak
=> 56
```

Još jedna korisna operacija koju ćemo kasnije koristiti daje kao rezultat element koji po svojoj veličini prvi slijedi element koji joj je zadan. Ovu operaciju nazvat ćemo *sljedeći*:

```
class Stablo:
    …
    def sljedeći(self, čvor):
        if čvor._desno != None:
            return self.najmanji(čvor._desno)
        else:
            c = čvor._prethodnik
            sljedbenik = čvor
            while c != None and sljedbenik == c._desno:
                sljedbenik = c
                c = c._prethodnik

            return c
```

Operacija *sljedeći* ima dva dijela. Prvi se dio odnosi na slučaj kada zadani čvor ima desno podstablo, gdje je tada sljedeći element onaj najmanji u tom podstablu. Drugim riječima, sljedeći element je onaj koji dobijemo tako da slijedimo samo lijevu stranu tog podstabla. Na primjer, koristeći ovo pravilo na slici 5.11 možemo vidjeti da je sljedeći element čvora 17 čvor 20, a čvora 31 čvor 32. U drugom slučaju, ako zadani čvor nema desno podstablo onda je sljedeći element neki čvor *c* koji je najmanji prethodnik zadanog čvora čiji je lijevi čvor također prethodnik zadanog čvora. Na slici 5.11 sljedeći element od čvora 12 je 17 zato jer je čvor 8 prethodnik čvora 12, a isto tako je i lijevi sljedbenik čvora 17. Općenito, za ovaj se slučaj jednostavno "penjemo" po stablu počevši od zadanog čvora, dok ne naiđemo na čvor koji je direktni lijevi sljedbenik svog prethodnika. U sljedećem primjeru koristimo ovu operaciju da bi saznali koja vrijednost dolazi nakon 12:

```
st.sljedeći(st.nađi(12))._podatak
=> 17
```

Ovdje smo prvo koristili operaciju *nađi* da bi dobili objekt za taj čvor koji smo zatim koristili za operaciju *sljedeći*.

5.6 Performanse stabla za binarno pretraživanje

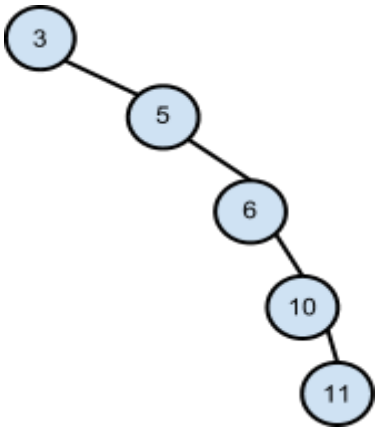
Postupak pronalaženja nekog elementa u stablu za binarno pretraživanje ima iste performanse kao i binarno pretraživanje niza, to jest vremenska složenost je i ovdje $O(log_2 n)$, gdje je n broj čvorova (elemenata) u stablu. S obzirom na spomenute nedostatke nizova, stablo za binarno pretraživanje je dobar izbor za slučajeve u kojima nam ne treba indeksiranje, odnosno za slučajeve gdje elemente stabla želimo tražiti na osnovu nekakve vrijednosti kojom su identificirani. Ovakva stabla, međutim, također imaju nedostatke. Kao jedan primjer, pogledajmo stablo na slici 5.13. Vidimo da to stablo u stvari izgleda kao lista zato jer su mu svi elementi na jednoj strani. Za takvo se stablo kaže da je *nebalansirano* kada se govori o stablu za binarno pretraživanje. Nebalansirano stablo može poprimiti razne oblike, od kojih je ovo na slici 5.13 jedan od krajnjih slučajeva. Ako zamislimo da imamo velik broj elemenata u takvom stablu onda binarno pretraživanje ne bi bilo ništa efikasnije od linearnog zato jer za svaki sljedeći element moramo (i možemo) ići samo na desnu stranu, čime bi prošli svaki čvor takvog stabla. Na primjer, ako tražimo broj 25 onda, kao i obično, počinjemo od korijena gdje se nalazi 3. S obzirom da je 25 > 3 nastavljamo s desnim podstablom gdje se nalazi 5. Sada je opet 25 > 5, tako da nastavljamo s desnim podstablom, i tako dalje, dok ne dođemo do kraja. Vidimo da postupkom binarnog pretraživanja u ovom slučaju nismo izbjegli niti jedan element, što praktički eliminira prednost korištenja stabla za binarno pretraživanje. Do ovakvih nebalansiranih oblika stabala za binarno pretraživanje može se doći postepenim uklanjanjem i umetanjem elemenata. Sljedećim primjerom iz stabla na slici 5.11 dobijamo stablo koje je u stvari lista:

```
for vrijednost in [17, 8, 12, 9, 32]:
    st.ukloni(st.nađi(vrijednost))

for vrijednost in [57, 60, 65]:
    st.dodaj(vrijednost)
```

Operacija *ispiši* pokazuje da stablo sada ima samo desnu stranu:

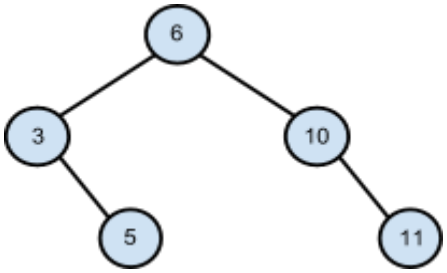
```
ispiši(st._korijen)
=> 20
      31
          45
              56
                  57
                      60
                          65
                              *
                              *
                          *
                      *
                  *
              *
          *
      *
```

Da bi izbjegli to da stablo za binarno pretraživanje dobije oblik u kojem ima previše čvorova na jednoj strani stabla, koriste se tehnike *balansiranja* takvog stabla. Rezultat balansiranja stabla prikazanog na slici 5.13 može rezultirati stablom na slici 5.14, gdje vidimo da imamo jednak broj čvorova na lijevoj i desnoj strani.



Slika 5.13 – Nebalansirano stablo za binarno pretraživanje.

Tehnike balansiranja stabla ovdje nećemo proučavati. Ono što je važno je to da za mnoge programske jezike postoje gotove komponente za rad sa stablima za binarno pretraživanje i one su često napravljene tako da stablo održavaju u balansiranom obliku.[3]



Slika 5.14 – Balansirano stablo za binarno pretraživanje.

Balansirano stablo za binarno pretraživanje ne mora imati jednak broj čvorova na svakoj strani zato jer to nije uvijek moguće, ali zavisno od tehnike balansiranja ta razlika je dovoljno mala da bi imala negativan utjecaj na performanse pretraživanja takvog stabla.

[3]Jedna vrsta stabla za binarno pretraživanje zove se *crveno-crno stablo* (eng. *red-black tree*) koje ima određena dodatna svojstva koja moraju biti zadovoljena da bi stablo bilo balansirano.

5.7 Sortiranje pomoću stabla za binarno pretraživanje

Svojstvo stabla za binarno pretraživanje koje smo na početku spomenuli, a koje kaže da su s lijeve strane svakog čvora elementi koji nisu veći, a s desne oni koji nisu manji od njega, implicira to da su elementi u takvom stablu u stvari sortirani. To mora biti slučaj jer u suprotnom ne bi mogli efikasno pretraživati takvo stablo, isto kao što su elementi niza morali biti sortirani da bi mogli primijeniti binarno pretraživanje. Da bi iz takvog stabla ispisali sve elemente po veličini dovoljno je doći do svakog elementa tako da idemo od krajnjeg lijevog prema krajnjem desnom, odozgo prema dole i s lijeva na desno. Operacija za to je iznenađujuće jednostavna:

```
def ispiši_sortirano(čvor):
    if čvor._lijevo:
        ispiši_sortirano(čvor._lijevo)
    print(čvor._podatak, end = '   ')
    if čvor._desno:
        ispiši_sortirano(čvor._desno)
```

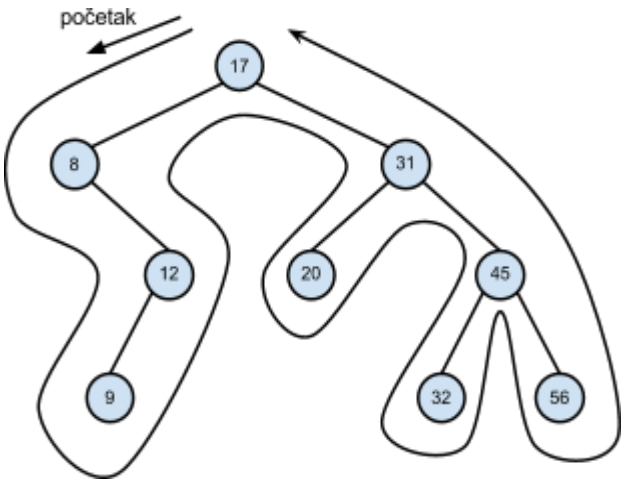
Ako ovoj operaciji zadamo korijen stabla dobit ćemo sve elemente tog stabla ispisane po veličini, od najmanjeg prema najvećem:

```
ispiši_sortirano(st._korijen)
=> 8   9   12   17   20   31   32   45   56
```

Po redoslijedu rekurzivih poziva ove operacije vidimo da elemente ispisujemo s lijeva na desno, jer je prvi po redu poziv za lijevi čvor, a potom za desni. Ako bi ovaj redoslijed preokrenuli elementi bi bili ispisani od najvećeg prema najmanjem:

```
def ispiši_sortirano_od_najvećeg(čvor):
    if čvor._desno:
        ispiši_sortirano_od_najvećeg(čvor._desno)
    print(čvor._podatak, end = '   ')
    if čvor._lijevo:
        ispiši_sortirano_od_najvećeg(čvor._lijevo)

ispiši_sortirano_od_najvećeg(st._korijen)
=> 56   45   32   31   20   17   12   9   8
```

Kod ovih dviju operacija možemo zamisliti kao da se drugi poziv pamti kod svakog prvog poziva. Na primjer, kod operacije *ispiši_sortirano* za svaki poziv *ispiši_sortirano(čvor._lijevo)* pamti se poziv *ispiši_sortirano(čvor._desno)* koji će biti izvršen nešto kasnije, odnosno kada se dođe do čvora od kojeg se više ne može ići ulijevo. U tom bi slučaju uvjet *if* naredbe bio *False* i vratili bi se na mjesto iza rekurzivnog poziva *ispiši_sortirano(čvor._lijevo)*, nakon čega bi došli do poziva *ispiši_sortirano(čvor._desno)*. Redoslijed kojim operacija *ispiši_sortirano* prolazi kroz čvorove stabla prikazan je na slici 5.15.



Slika 5.15 - Prolaz kroz stablo.

Zadaci za vježbu

1. Napišite funkciju *je_bsp* koja vraća True ako je zadano stablo binarno stablo pretraživanja ili False ako nije. Na primjer,

```
je_bsp([50, [44, 30, 48], [80, [66, 60, 67], 88]])
=> True

je_bsp([50, [44, 30, 48], [49, [66, 60, 67], 88]])
=> False
```

2. Napišite funkciju *napravi_binarno_stablo* koja za niz numeričkih vrijednosti vraća (Python) listu koja predstavlja binarno stablo tih vrijednosti. Na primjer,

```
napravi_binarno_stablo([17, 8, 31, 12, 9, 45, 20, 32, 56])
=> [17, [8, None, [12, 9, None]], [31, 20, [45, 32, 56]]]
```

Ove vrijednosti ne moraju nužno biti u ovom poretku u stablu, zavisno od toga kojim su redoslijedom dodavane.

3. Napišite funkciju *dubina_bsp* koja vraća dubinu zadanog binarnog stabla. Na primjer, ako se korijen stabla nalazi na dubini 0 onda

```
dubina_bsp([17, [8, None, [12, 9, None]], [31, 20, [45, 32, 56]]])
=> 3
```

Vježba 6: Gomila (*heap*)

Zadaci za vježbu

1. ...

Vježba 7: Algoritmi - sporo sortiranje

Zadaci za vježbu

1. Algoritam „bubble sort“ radi tako da uzastopno zamijenjuje dva susjedna elementa ako su u pogrešnom redoslijedu. Slika ispod prikazuje rad algoritma „bubble sort“. Napišite funkciju *bubble_sort* koja sortira vrijednosti prema tom algoritmu.

		0	1	2	3	4	5	6	7
i = 0	j								
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i=1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i=2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i=3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i=4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i=5	0	1	2	3	4				
	1	1	2	3					
i=6	0	1	2	3					
		1	2						

2. Slika ispod prikazuje rad algoritma „insertion sort“. Napišite funkciju *insertion_sort* koja sortira vrijednosti prema tom algoritmu.

3. Napišite program koji uspoređuje vremena izvršavanja ovih dvaju algoritama na nizovima od 10000 elemenata.

4. Proširite funkciju *bubble_sort* tako da može sortirati niz koji se sastoji od bilo kakvih objekata, gdje se funkcija usporedbe dvaju elemenata može zadati kao parametar.

Vježba 8: Algoritmi – brzo sortiranje

8.1 Binarno pretraživanje

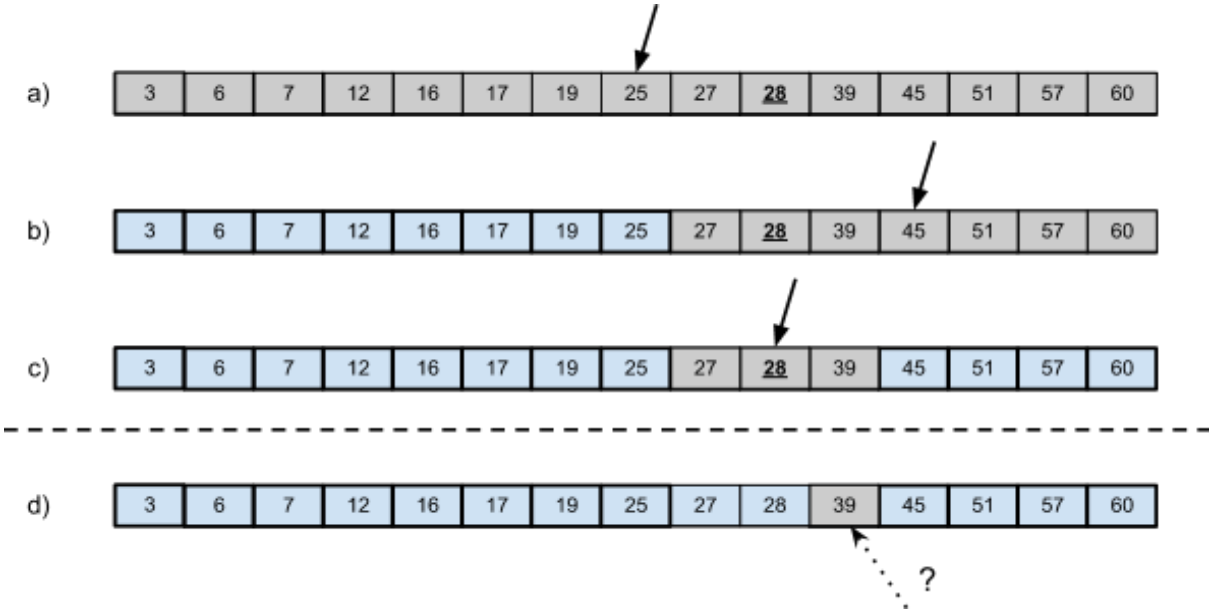
Da bi vidjeli da li neki element postoji u nizu moramo u najjednostavnijem slučaju ispitati svaki indeks tog niza, po redu, dok ne naiđemo na traženi element ili dok ne dođemo do kraja niza. Možemo reći, prema tome, da je vrijeme potrebno za takvu operaciju *O(n)*, gdje je *n* broj elemenata u nizu, zato jer ako traženi element ne postoji u nizu morali bi pretražiti cijeli niz. Ovakvo se pretraživanje zove *linearno* ili *sekvencijalno pretraživanje* i najjednostavnija je tehnika pretraživanja koju smo već koristili u prvom poglavlju. Kao što smo već vidjeli, operaciju za ovakvo pretraživanje, koju ćemo zvati *linearno_pretraživanje*, možemo definirati na ovaj način:

```
def linearno_pretraživanje(e, niz):
    for i in range(0, len(niz)):
        if e == niz[i]:
            return i

    return None
```

Ovim, međutim, podrazumijevamo da elementi niza nisu uređeni ni po kakvom određenom poretku, pa zbog toga ne možemo pretpostaviti gdje se u nizu može nalaziti traženi element, čime bi izbjegli ispitivanje svakog indeksa tog niza i poboljšali performanse ovakve operacije. Jedan način na koji možemo urediti elemente niza za efikasnije pretraživanje je da ih sortiramo po veličini. Sortiranje u ovom kontekstu ne uključuje samo brojeve, nego bilo koji tip vrijednosti koje možemo postaviti u nekakav njima prirodni poredak. Na primjer, tekstualne vrijednosti možemo sortirati po abecedi. Ako pretražujemo niz u kojem su vrijednosti sortirane onda je jedno očito poboljšanje performansi takve operacije pretraživanja to da prekinemo pretraživanje čim dođemo do prvog elementa u nizu koji je veći od onog koji tražimo (za slučaj gdje traženi element ne postoji). Ovu operaciju zvat ćemo *sortirano_pretraživanje* i kod nje pretpostavljamo da su elementi u zadanom nizu sortirani u rastućem poretku:

```
def sortirano_pretraživanje(e, niz):
    for i in range(0, len(niz)):
        if e == niz[i]:
            return i
        elif niz[i] > e:
            return None

    # ako je traženi element veći od posljednjeg u nizu
    return None
```

Iako je ova operacija poboljšanje u odnosu na operaciju *linearno_pretraživanje*, činjenica da su elementi sortirani omogučava nam značajno bolju tehniku pretraživanja koja se zove *binarno pretraživanje*. Ova tehnika pretraživanja bazirana je na jednoj jednostavnoj obzervaciji, a ta je da ako počnemo tražiti od bilo kojeg indeksa u sortiranom nizu onda uvijek znamo da li se traženi element nalazi negdje prije ili poslije tog indeksa. Ovo je slično situaciji u kojoj tražimo određenu stranicu u nekoj knjizi: Ako tražimo stranicu 720, a knjigu smo otvorili na stranici 521 onda moramo nastaviti s traženjem samo među ostatkom stranica, a ne prije toga, jer znamo da se stranica 720 ne može nalaziti prije stranice 521. Na ovaj način u svakom koraku preskačemo velik broj stranica dok ne dođemo do one koju tražimo. Na sličan način možemo definirati operaciju pretraživanja niza, pod pretpostavkom da su njegovi elementi sortirani. Ako niz zamislimo kao da su elementi poslagani s lijeva na desno onda ova operacija radi tako da počne od sredine niza, gdje ispita element na tom indeksu. Ako taj element nije onaj koji tražimo onda uzmemo u obzir to da li se traženi element nalazi negdje lijevo ili desno od tog elementa. Na osnovu te informacije ponavljamo ovaj postupak počevši od indeksa koji se nalazi u sredini lijevog ili desnog segmenta niza u odnosu na taj element, i tako dalje. Na ovaj način od preostalih elemenata svaki put preskačemo pola njih. Ovaj je postupak prikazan na slici 5.3, gdje vidimo da smo za traženi element 28 morali ispitati svega tri elementa od njih petnaest - 25, 45 i 28 (slike *a*, *b* i *c*). Ako bi, na primjer, tražili broj 40 koji ne postoji, onda bi morali ispitati samo četiri elementa, gdje bi posljednji ispitani element bio broj 39 (slika *d*), za razliku od svih petnaest elemenata koje bi morali ispitati linearnim pretraživanjem.

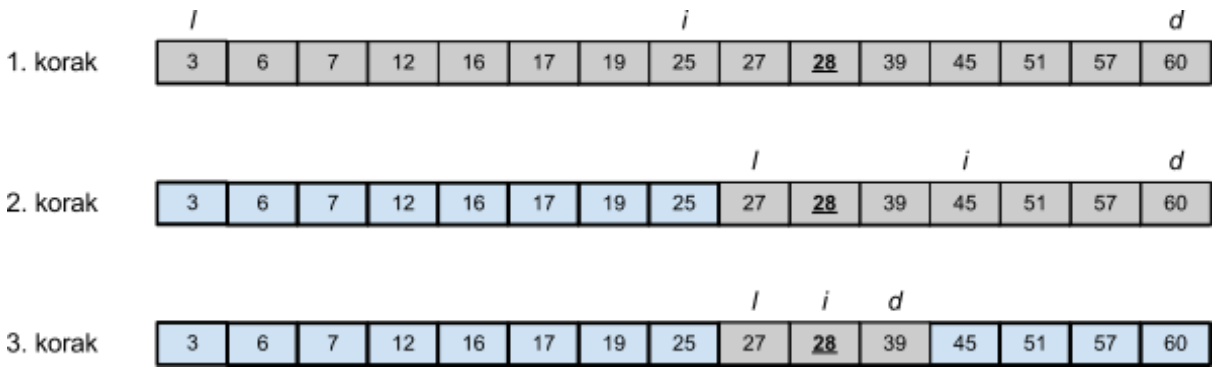


Slika 5.3 - Binarno pretraživanje niza od 15 elemenata, gdje tražimo broj 28. Na slici a) počinjemo na polovici niza, gdje se nalazi broj 25. S obzirom da je 28 veći od 25 nastavljamo s desnom polovicom na čijoj se sredini nalazi broj 45, što je prikazano na slici b). Broj 28 je manji od 45, pa nastavljamo na lijevoj polovici od prethodne polovice, gdje dolazimo do traženog broja 28 (slika c)). Da smo tražili broj 40 ovim bi postupkom došli do broja 39 (zato jer je 40 manji od 45, a veći od 28) i tu bi stali jer bi time došli do segmenta koji više ne možemo rastaviti na lijevu i desnu stranu (sastoji se od samo jednog elementa). Ovo je pokazano na slici d).

Ovaj postupak ćemo sada upotrijebiti za operaciju *binarno_pretraživanje*:

```
def binarno_pretraživanje(e, niz):
    l = 0
    d = len(niz) - 1
    i = len(niz) // 2
    while l <= d:
        if e == niz[i]:
            return i
        else:
            if e < niz[i]:
                d = i - 1
            else:
                l = i + 1

            i = (l + d) // 2

    return None     # traženi element ne postoji u nizu
```

U ovoj operaciji varijable *l* i *d* određuju segment niza na čijoj sredini ispitujemo element. Način na koji ona radi prikazan je na slici 5.4 gdje tražimo broj 28. U prvom koraku počinjemo na sredini niza. S obzirom da je 28 veći od 25 moramo nastaviti s desnom stranom niza, pa se prema tome izvršava naredba *l = i + 1*, tako da je sada novi segment onaj pokazan u drugom koraku. Indeks *i* je sada na vrijednosti 45 koja se nalazi na sredini tog segmenta i koja je veća od 28, što znači da moramo preći na lijevi segment koji se nalazi između *l* i *i - 1*, odnosno izvršava se naredba *d = i - 1*, nakon čega dobijamo segment pokazan u trećem koraku. Na sredini ovog segmenta nalazi se tražena vrijednost 28.



Slika 5.4 - Rad operacije *binarno_pretraživanje*.

Jedno svojstvo binarnog pretraživanja je to da za niz od *N* vrijednosti treba ispitati najviše $log_2N + 1$ vrijednosti. To znači da, na primjer, za niz koji sadrži milijardu vrijednosti (pod uvjetom da je taj niz sortiran) nikada ne moramo ispitati više od 30 vrijednosti, što je drastična razlika u odnosu na jednostavno linearno pretraživanje. Ovo se može činiti kao mali broj za toliko vrijednosti, ali ako uzmemo u obzir da kod svakog neuspješnog uspoređivanja vrijednosti njihov broj praktički prepolovimo onda se lako vidi da u svakom koraku odbacujemo velik broj vrijednosti, odnosno pola od broja vrijednosti u prethodnom

koraku. U slučaju milijarde vrijednosti, nakon prvog koraka taj bi broj pao na 500.000.000, nakon drugog na 125.000.000, trećeg na 62.500.000, i tako dalje, tako da bi ovim postupkom u 30 koraka došli do 1. Vremenska složenost binarnog pretraživanja je, dakle, $O(log_2N)$. Ako usporedimo rast funkcija $f(n) = n$ (linearno pretraživanje) i $f(n) = log_2n$ (binarno pretraživanje) možemo vidjeti da prva funkcija, koja karakterizira linearno pretraživanje, raste znatno brže od druge, što znači da vrijeme potrebno za pronalaženje nekog elementa linearnim pretraživanjem raste brže s povećanjem broja elemenata u nizu. Iz ovoga možemo zaključiti da binarno pretraživanje ima znatno bolje performanse od linearnog. Ovdje je važno ponovo istaknuti to da kada se radi o pretraživanju niza onda se binarno pretraživanje može koristiti samo ako su vrijednosti u tom nizu sortirane po veličini.
Da bi preciznije utvrdili performanse binarnog pretraživanja u odnosu na linearno, možemo ih izmjeriti. Kao prvi korak definirat ćemo niz koji se sastoji od 100000 slučajnih brojeva čije su vrijednosti u intervalu između 0 i 1000:

```
import random

niz = []
for i in range(0, 100000):
    niz += [random.randint(0, 1000)]
```

U idućem koraku koristit ćemo Pythonov modul *timeit* koji se koristi za mjerenje vremena izvršavanja kratkih programa:

```
from timeit import Timer
```

Koristeći klasu *Timer* ovog modula pozivat ćemo operaciju *linearno_pretraživanje* za broj -5 koji ne postoji u gornjem nizu slučajnih brojeva (ovdje je cilj demonstrirati vrijeme pretraživanja cijelog niza, pa zato tražimo nepostojeći broj):

```
lt = Timer('linearno_pretraživanje(-5, niz)',
           'from __main__ import linearno_pretraživanje, niz')
```

Prvi argument za klasu *Timer* je poziv operacije *linearno_pretraživanje* zadan kao tekstualna vrijednost, a drugi je neophodan za izvršavanje tog poziva, ali nam detalji ovdje nisu važni. Sada ćemo izmjeriti koliko traje 1000 gore zadanih poziva, to jest 1000 poziva *linearno_pretraživanje(-5, niz)*:

```
lt.timeit(1000)
=> 12.75668044476587
```

Rezultat pokazuje da je 1000 ovih poziva trajalo nešto više od 12 sekundi. Da bi isto napravili za binarno pretraživanje, gornji ćemo niz prvo sortirati:

```
niz.sort()
```

Sada ćemo definirati objekt klase *Timer* za poziv *binarno_pretraživanje(-5, niz)*, isto kao što smo napravili za linearno pretraživanje:

```
bt = Timer('binarno_pretraživanje(-5, niz)',
           'from __main__ import binarno_pretraživanje, niz')
```

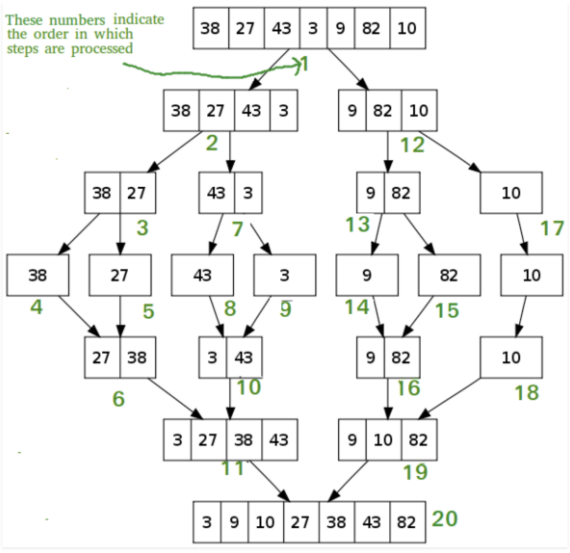
Za isti broj poziva dobijamo sljedeći rezultat:

```
bt.timeit(1000)
=> 0.008883819980997743
```

Vidimo da je 1000 poziva za binarno pretraživanje trajalo manje od jedne sekunde, odnosno oko devet tisućina sekunde, što je drastična razlika u odnosu na linearno pretraživanje. Ovdje, naravno, moramo uzeti u obzir to da smo niz prije toga morali sortirati, što također zahtijeva dodatno vrijeme. Kako smo prethodno vidjeli, binarno stablo je jedna struktura podataka čiji se elementi tokom dodavanja i uklanjanja mogu efikasno održavati u sortiranom redoslijedu, što je čini vrlo pogodnom strukturom za binarno pretraživanje.

Zadaci za vježbu

1. Algoritam „merge sort“ radi tako da niz vrijednosti podijeli na pola i za svaku od tih polovina ponovo poziva sebe, gdje na kraju spoji sve te sortirane podnizove. To je jedan od algoritama koji koristi tehniku „podijeli-pa-riješi“ (*divide-and-conquer*). Na slici ispod prikazan je način rada ovog algoritma.

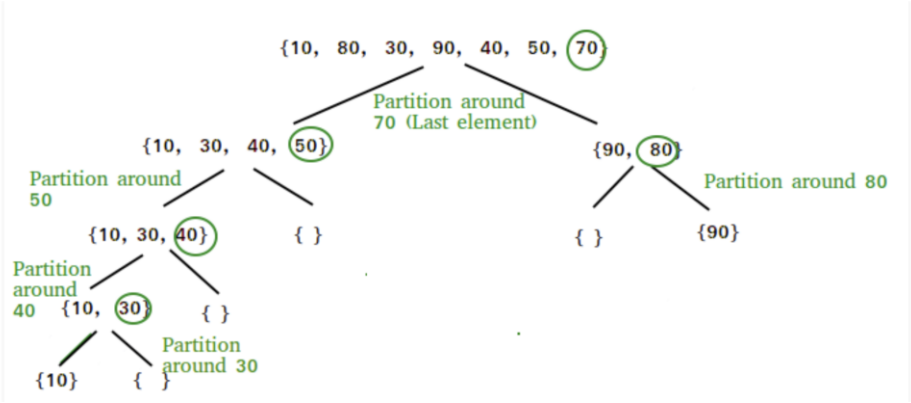


a) Napište funkciju *merge_sort* koja ovaj algoritam implementira rekurzivno.
b) Proširite gornju implementaciju tako da daje ukupan broj usporedbi vrijednosti. Isto napravite i za vašu implementaciju *bubble sort* algoritma i usporedite i komentirajte rezultate.
c) Nađite veličinu niza kod koje funkcija *bubble_sort* sortira brže od funkcije *merge_sort*. Objasnite zašto je to tako.

2. Algoritam *quick sort*, kao i *merge sort*, radi po principu „podijeli-pa-riješi“. Za razliku od *merge sort*a, ovaj algoritam dijeli niz na osnovu jednog elementa *P*, tako da se niz dijeli u podnizove, gdje jedan podniz sadrži sve elemente manje od *P*, a drugi sve elemente veće od *P*. Algoritam nakon toga poziva samog sebe za te podnizove koje na kraju spaja u konačni sortirani niz. Element *P* može se odabrati na nekoliko načina:

* Uvijek odaberi prvi element
* Uvijek odaberi posljednji element
* Odaberi element slučajnim odabirom
* *P* je srednja vrijednost niza

Na slici ispod prikazan je način rada ovog algoritma:



a) Napišite funkciju *quick_sort* koja ovaj algoritam implementira rekurzivno.
b) Generirajte 10 nizova koji se sastoje od 10.000 slučajnih brojeva. Za svaki niz pozovite *quick_sort* funkciju i izmjerite vrijeme izvršavanja. Sada generirajte jedan niz od 10.000 brojeva sortiranih u rastućem poretku i izmjerite vrijeme izvršavanja, te komentirajte rezultat.
c) Za svaki od gore generiranih nizova pozovite i vašu funkciju za merge-sort i usporedite vremena te funkcije s ovom za quick-sort.

Vježba 9: Hash-tehnike adresiranja

9.1 Uvod

U radu s nizovima koristili smo indekse za pristup elementima. Iako je to za mnoge aplikacije dovoljno, često se javlja potreba za pristup podacima pomoću nekakvog složenijeg identifikatora od jednostavne numeričke vrijednosti kao što je indeks. Ako bi, na primjer, željeli implementirati strukturu podataka koja odgovara telefonskom imeniku onda pristup telefonskim brojevima na osnovu indeksa ne bi bio praktičan zato jer je svaki taj broj pridružen jednom imenu, a ne indeksu. To znači da bi nam u tom slučaju trebala jedna struktura podataka koju možemo koristiti slično kao i niz, ali kod koje umjesto indeksa koristimo neku složeniju vrijednost kao što je tekst. Takva struktura podataka jednoj vrijednosti pridružuje neku drugu vrijednost koja se zove *ključ*. Ključ ima istu ulogu kao i indeks kod nizova, to jest na osnovu njega možemo dobiti vrijednost koja je pridružena tom ključu, isto kao što na osnovu indeksa dobijemo vrijednost koja se nalazi na toj poziciji u nizu. Niz u kojem je svaka vrijednost pridružena jednom takvom ključu zove se *asocijativan niz*. Način na koji koristimo asocijativne nizove može izgledati ovako:

```
telefonski_broj['Charlie Chaplin'] = '1234-567'

telefonski_broj['Charlie Chaplin']
=> 1234-567
```

U ovom primjeru asocijativan niz *telefonski_broj* koristimo slično kao i običan niz, ali umjesto indeksa koristimo tekstualnu vrijednost. U prvom redu vrijednost *'1234-567'* pridružujemo ključu *'Charlie Chaplin'*, a u drugom ispisujemo vrijednost pridruženu tom ključu. Asocijativni nizovi često se nazivaju *mapama*, *tabelama*, a ponekad i *rječnicima* zbog očite sličnosti.
Mapu možemo implementirati slično kao i stablo ili vezanu listu, koristeći dvije klase - jednu koja sadrži ključ i vrijednost, te drugu na osnovu koje ćemo instancirati objekte koji predstavljaju mapu. Što se tiče strukture podataka koju ćemo koristiti za smještanje i pretraživanje elemenata mape, možemo koristiti stablo za binarno pretraživanje. To znači da ovdje jednu strukturu podataka koristimo da bi implementirali drugu, isto kao što smo vezanu listu koristili za implementaciju stoga i reda. Elemente mape držat ćemo u objektu klase *ElementMape* čiji konstruktor će izgledati ovako:

```
class ElementMape:
    def __init__(self, k, v = ''):
        self._ključ = k
        self._vrijednost = v
```

Svaki element mape sadržavat će ključ i vrijednost pridruženu tom ključu. Parametru za vrijednost, *v*, pridružili smo početnu vrijednost, što znači da njega nije obavezno zadati (u kojem slučaju će sadržavati ovu početnu vrijednost). Da bi mogli koristiti stablo za binarno pretraživanje svaki objekt koji u to stablo želimo dodati mora zadovoljavati dva osnovna uvjeta, odnosno za njega moraju biti definirane dvije operacije: Operacija *jednako* (==) i

operacija *manje-od* (<). To možemo vidjeti u operacijama *dodaj* i *nađi*, gdje neki objekt uspoređujemo da bi znali da li je jednak traženom objektu (operacija "==") ili se nalazi na lijevoj odnosno desnoj strani nekog čvora (operacija "<"). S obzirom da Pythonov interpreter ne može znati šta nama znači da su dva objekta klase *ElementMape* jednaka ili da je jedan manji od drugog, za objekte ove klase moramo definirati te dvije operacije:

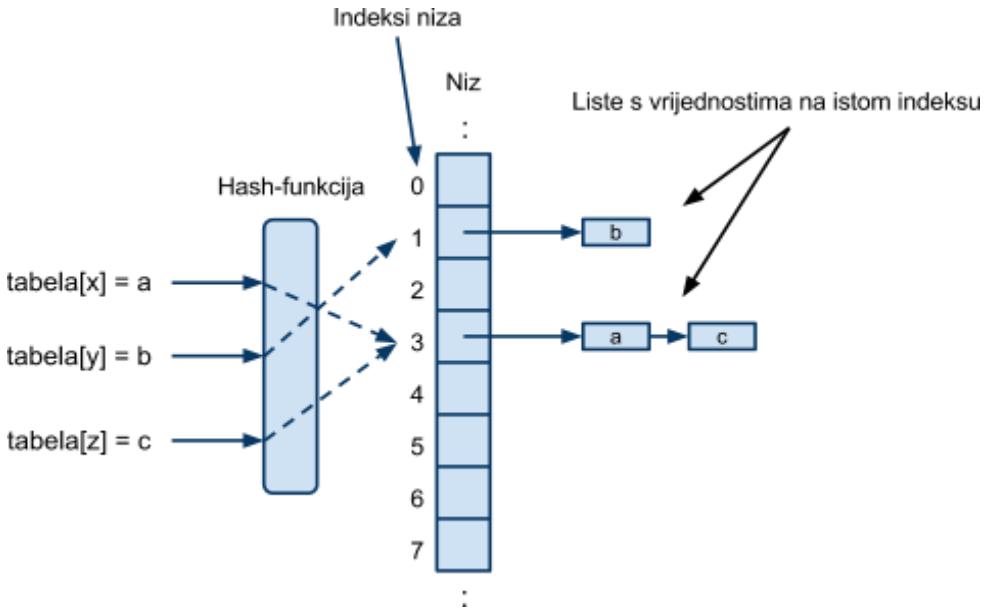
```
class ElementMape:
    …
    def __eq__(self, x):
        return self._ključ == x._ključ

    def __lt__(self, x):
        return self._ključ < x._ključ
```

Ovo smo već vidjeli u trećem poglavlju; za dva objekta *a* i *b* klase *ElementMape* izraz *a == b* interpretira se kao *ElementMape.__eq__(a, b)*, a izraz *a < b* kao *ElementMape.__lt__(a, b)*. To znači da bi se parametar *self* u gornjim definicijama ove dvije operacije odnosio na *a*, a parametar *x* na *b*. Sada imamo sve što nam treba za korištenje stabla za binarno pretraživanje kao osnovu za mapu, tako da možemo definirati klasu *Mapa*:

```
class Mapa:
    def __init__(self):
        self._stablo = Stablo()

    def __setitem__(self, ključ, vrijednost):
        self._stablo.dodaj(ElementMape(ključ, vrijednost))

    def __getitem__(self, ključ):
        e = self._stablo.nađi(ElementMape(ključ))
        if e == None:
            raise PogrešanKljuč()
        else:
            return e._podatak._vrijednost

class PogrešanKljuč(Exception): pass
```

Operacije *__getitem__* i *__setitem__* su specijalne operacije koje interpreter za Python automatski poziva kod izraza indeksiranja, kao što je *a[i]*, gdje je *a* objekt klase u kojoj su ove operacije definirane (ili samo jedna od njih). Te nam dvije operacije samo omogučavaju da objekte klase *Mapa* koristimo kao nizove, umjesto da definiramo obične operacije kao što su *dodaj* i *nađi* kakve koristimo kod klasa *Stablo* ili *VezanaLista*. Izraz *m[k]*, gdje je *m* objekt klase *Mapa*, sada se interpretira kao *Mapa.__getitem__(m, k)*, a izraz *m[k] = v* kao *Mapa.__setitem__(m, k, v)*, to jest *__getitem__* se poziva samo kod čitanja vrijednosti pridruženoj ključu *k*, a *__setitem__* kod pridruživanja vrijednosti *v* ključu *k*. U operaciji *__setitem__* u stablo dodajemo ključ i vrijednost kao novi objekt klase *ElementMape*, dok u operaciji *__getitem__* jednostavno tražimo zadani ključ u tom stablu i, ako ga nađemo,

rezultat ove operacije je vrijednost pridružena zadanom ključu, u suprotnom signaliziramo grešku tipa *PogrešanKljuč* koja označava to da traženi ključ ne postoji u mapi.
Iako klasa *Mapa* u osnovi radi ono što nam treba, način na koji pronalazi vrijednost pridruženu nekom ključu nije efikasan u odnosu na obične nizove. Razlog tome je očit - umjesto da izračunamo gdje se nalazi vrijednost za neki zadani ključ, mi taj ključ tražimo. Na primjer, za milion vrijednosti u ovoj mapi morali bi uspoređivati oko dvadesetak ključeva (pod uvjetom da je stablo balansirano) da bi saznali da neki ključ ne postoji. S obzirom da u ovoj mapi koristimo stablo za binarno pretraživanje vremenska složenost je $O(log_2 n)$, što je slabo u odnosu na $O(1)$ za obične nizove. Način na koji možemo poboljšati ovu karakteristiku je taj da na osnovu ključa izračunamo gdje se (otprilike) nalazi vrijednost koja mu je pridružena. Ako, na primjer, koristimo tekstualne ključeve onda možemo uzeti u obzir broj znakova u tom ključu, odnosno njegovu duljinu, tako da su sve vrijednosti koje su pridružene nekom ključu s određenom duljinom na jednom mjestu. Na primjer, sve vrijednosti koje su pridružene ključu koji se sastoji od 10 znakova mogu biti smještene u jednoj vezanoj listi, gdje bi za svaku takvu vrijednost postojao podatak o tome kojem je specifičnom ključu od 10 znakova ona pridružena. Tada, da bi našli odgovarajuću vrijednost, pretražujemo samo taj skup ključeva (onih koji se sastoje od 10 znakova), a ne sve njih, što bi ubrzalo pretraživanje. Ako se, na primjer, od milion ključeva njih 5 sastoji od 10 znakova tada bi u najgorem slučaju morali pretražiti samo 0.0005% ključeva, što je značajno poboljšanje. Ovo bi realizirali tako da u mapi grupiramo vrijednosti na osnovu nekih karakteristika ključa kojem su one pridružene, tako da bi (za potrebe pretraživanja) mapa na osnovu ključa odredila na koju se grupu vrijednosti on odnosi, to jest koju grupu vrijednosti treba pretraživati. Ovdje već možemo vidjeti da efikasnost pretraživanja na ovaj način ovisi o načinu raspodjele vrijednosti u takve grupe. Na primjer, za milion vrijednosti koje su raspoređene u samo dvije grupe od 500,000 vrijednosti pretraživanje neće biti efikasno kao kada bi bile raspoređene u 20.000 grupa od 50 vrijednosti, jer bi u ovom drugom slučaju, nakon što bi izračunali u kojoj se grupi nalazi vrijednost sa zadanim ključem, pretraživali puno manji skup vrijednosti. Nadalje, kako su vrijednosti raspodijeljene kroz sve te grupe također utječe na efikasnost pretraživanja - ako je od 20.000 grupa 95% vrijednosti nakupljeno u svega nekoliko grupa onda će pretraživanje biti sporo jer će se uglavnom odvijati u tih nekoliko grupa u kojima se nalazi velika većina vrijednosti.
Grupiranje podataka, dakle, igra glavnu ulogu kod ovakvih mapa i zasnovano je na nečemu što se zove *hash-funkcija*. Ako pretpostavimo da koristimo jedan niz koji sadrži ovakve grupe vrijednosti, svrha hash-funkcije je da nam kaže na kojem se indeksu u nizu nalazi grupa u kojoj se nalazi podatak s traženim ključem. U ovom slučaju pretpostavljamo da se na jednom indeksu niza nalazi neki objekt koji sadrži više podataka, kao što je vezana lista, stablo ili niz. Na tom se indeksu nalaze sve vrijednosti za čiji je ključ hash-funkcija dala isti broj, tako da su sve one bile smještene na isti indeks u nizu. Ovdje treba istaknuti da sve te vrijednosti koje se nalaze na istom indeksu niza imaju različite ključeve, ali je za te ključeve rezultat hash-funkcije bio isti, kao što možemo imati puno različitih tekstualnih ključeva duljine 10 znakova. Ako hash-funkcija uzima u obzir samo duljinu ključa onda će njen rezultat za sve ove ključeve od 10 znakova biti 10, što bi značilo da će se svi oni nalaziti na indeksu 10 u nizu (odnosno nalazit će se u grupi vrijednosti na tom indeksu). Mape kod kojih se podaci pretražuju pomoću hash-funkcije zovu se *hash-tabele*. Struktura hash-tabele ilustrirana je na slici 5.21.



Slika 5.21 - Ilustracija hash-tabele: Hash-funkcija za oba ključa *x* i *z* daje isti broj, 3, što znači da na indeksu 3 imamo dvije vrijednosti: Vrijednost *a* pridruženu ključu *x* i vrijednost *c* pridruženu ključu *z*, dok ova funkcija za ključ *y* daje 1, što je indeks na kojem se nalazi vrijednost *b* pridružena ključu *y*.

Hash-funkcije implementiramo jednostavno pomoću operacija. Na primjer, hash-funkciju koja radi s tekstualnim ključevima i koja na osnovu duljine ključa daje indeks na kojem se nalazi vrijednost ili grupa kojoj pripada vrijednost pridružena tom ključu može, u trivijalnom slučaju, izgledati ovako:

```
def hash_funkcija(ključ): return len(ključ)
```

Ovo je prilično naivna hash-funkcija zato jer u slučaju da postoji puno ključeva iste duljine imali bi puno vrijednosti koje bi dijelile jedan te isti indeks. Jedan bolji način bio bi taj da na osnovu znakova tesktualnog ključa izračunamo indeks tako da, na primjer, zbrojimo numeričke kodove svih znakova, pa da onda za taj zbroj odredimo ostatak dijeljenja s veličinom niza:

```
def hash_funkcija(ključ, m):
    x = 0
    for znak in ključ:
        x += ord(znak)

    return x % m        # ostatak dijeljenja
```

Ovdje ostatak dijeljenja spriječava to da je rezultat hash-funkcije veći od veličine samog niza u koji smještamo vrijednosti zato jer *x % m* ne može nikada biti veći od *m*. Sada za dva ključa iste duljine možemo dobiti različite vrijednosti, pod uvjetom da ta dva ključa ne koriste iste znakove:

```
hash_funkcija('plava', 101)
```

```
=> 27

hash_funkcija('trava', 101)
=> 37
```

U ova dva primjera vrijednost 101 za *m*, odnosno veličinu tabele (niza), je arbitrarno izabran prost broj[4]. Problem kod ove hash-funkcije je u tome da za ključeve koji se sastoje od istih znakova (koji mogu biti nanizani u drugačijem redoslijedu) ova funkcija daje isti indeks. Jedan bolji način je taj da indeks za svaki ključ izračunamo tako da ključ prikažemo kao broj s nekom bazom, kao što je 128, slično kao što smo decimalne brojeve prikazivali u bazama 2 i 16 u drugom poglavlju, gdje bi svaki znak ključa predstavljao broj koji odgovara njegovom ASCII kodu. Na primjer, ključ OCEAN prikazali bi ovako:

$79 \cdot 128^4 + 67 \cdot 128^3 \cdot 69 \cdot 128^2 + 65 \cdot 128^1 + 78 \cdot 128^0$

Da bi izbjegli potenciranje, gornji izraz možemo izračunati koristeći takozvanu *Hornerovu metodu*:

$(((79 \cdot 128 + 67) \cdot 128 + 69) \cdot 128 + 65) \cdot 128 + 78$

Ovaj bi ključ, dakle, odgovarao broju 21348049102. Sada za hash-funkciju *h* i ključ *k* možemo izračunati indeks po formuli $h(k) = k \bmod m$, gdje je *m* veličina hash-tabele. Operaciju za ovakvu hash-funkciju možemo napisati ovako:

```
def hash_funkcija(ključ, m):
    k = ord(ključ[0])
    for i in range(1, len(ključ)):
        znak = ord(ključ[i])
        k = k * 128 + znak

    return k % m
```

Ovdje više nemamo problem s ključevima koji sadrže iste znakove ili koji su iste duljine:

```
hash_funkcija('OCEAN', 101)
=> 80

hash_funkcija('NAECO', 101)
=> 79

hash_funkcija('ABCDE', 101)
=> 10
```

[4]Jedan poželjan uvjet za ovakvu funkciju je taj da *m* bude prost broj, jer bi u suprotnom mogli dobiti situaciju da rezultat hash-funkcije ovisi samo o posljednjem znaku ključa.

Hash-tabelu ćemo ovdje implementirati klasom *HashTabela*. Prvo ćemo definirati klasu *Vrijednost* koja će sadržavati ključ i vrijednost; objekte ove klase smještat ćemo u hash-tabelu, slično kao što smo objekte klase *ElementMape* smještali u mapu:

```
class Vrijednost:
    def __init__(self, k, v):
        self._ključ = k
        self._vrijednost = v
        self._element_liste = None
```

Objekte ove klase također ćemo dodavati u vezanu listu zato da bi imali popis svih elemenata hash-tabele. U klasi *VezanaLista* vidjeli smo da operacija *dodaj* kao rezultat vraća dodani objekt klase *Element* - taj ćemo objekt pridružiti varijabli *_element_liste* definiranoj u klasi *Vrijednost* i koristit ćemo ga kod uklanjanja elemenata iz hash-tabele.
Objekti klase *Vrijednost* sadrže ključ za vrijednost koja mu je pridružena, tako da ćemo preko tog ključa tražiti vrijednost u listi koja sadrži sve vrijednosti koje su bile postavljene na isti indeks. Klasu *HashTabela* definirat ćemo postepeno, počevši s konstruktorom:

```
class HashTabela:
    def __init__(self, h, veličina = 101, opterećenje = 0.75,
                                                    proširenje = 1.6):
        self._fn = h
        self._veličina = veličina
        self._vrijednosti = [None] * veličina
        self._elementi = VezanaLista()      # lista svih elemenata
        self._opterećenje = opterećenje
        self._proširenje = proširenje
```

Vidimo da konstruktor uzima četiri argumenta, od kojih su posljednja tri neobavezna, jer je svaki od ova tri parametra postavljen na početnu vrijednost u slučaju da mu nije zadan argument. Parametar *h* bit će postavljen na operaciju koja služi za izračunavanje hash-funkcije. U trećem redu formiramo niz od elemenata *None* čiji je broj zadan parametrom *veličina*. Ako koristimo početnu vrijednost ovog parametra onda će ovaj niz u početku sadržavati 101 *None* vrijednost. O ostalim parametrima govorit ćemo kasnije. Objekt za hash-tabelu možemo instancirati ovako:

```
ht = HashTabela(hash_funkcija)
```

Operaciju *hash_funkcija* prethodno smo definirali i ovdje želimo da je hash-tabela koristi za izračunavanje indeksa.
Jedna nepoželjna karakteristika hash-tabela je ta da im performanse padaju kada broj elemenata poraste do te mjere da je tabela puna ili skoro puna. To je, u stvari, sasvim logično jer to znači da (skoro) svaki indeks niza sadrži najmanje jedan element i da će ostali elementi koje dodajemo u hash-tabelu morati dijeliti isti indeks s nekim postojećim elementima. To nadalje znači da će sve više vremena biti utrošeno u pretraživanju lista koje sadrže te elemente. Da bi ublažili negativne efekte ove pojave, hash-tabelu povremeno treba *proširiti* tako da joj povećamo niz u kojem smješta elemente, a nakon toga postojeće

elemente treba ponovo rasporediti u tom novom nizu po istom principu, koristeći njihov ključ i postojeću hash-funkciju. To ćemo proširenje obaviti u toku dodavanja novog elementa, za što će biti zadužena operacija *__setitem__*:

```
class HashTabela:
    …
    def __setitem__(self, ključ, vrijednost):
        # ako je broj elemenata prešao zadani prag tabelu treba proširiti
        if self._elementi.veličina() > self._veličina * self._opterećenje:
            self.proširi()

        indeks = self._fn(ključ, self._veličina)

        # ako na ovom indeksu nema ništa dodaj praznu vezanu listu
        if self._vrijednosti[indeks] == None:
            self._vrijednosti[indeks] = VezanaLista()
        else:
            # Ako već postoji vrijednost s ovim ključem onda
            # tu vrijednost moramo ukloniti prije nego što dodamo
            # novu za isti ključ...
            lista = self._vrijednosti[indeks]
            e = self._nađi_vrijednost(lista, ključ)
            if e != None:
                self._ukloni_element(lista, e)

        # dodaj novi element u listu
        v = Vrijednost(ključ, vrijednost)
        self._vrijednosti[indeks].dodaj(v)

        # dodaj novi element u listu svih elemenata
        e = self._elementi.dodaj(v)
        v._element_liste = e
```

Na početku provjeravamo da li je broj elemenata u tabeli dostigao neki određeni postotak (koji je zadan u konstruktoru i koji možemo promijeniti u toku definiranja novog objekta ove klase). U ovom primjeru, ako taj postotak ostavimo kako je unaprijed zadan onda on iznosi 75%, što znači da će niz biti proširen kada broj elemenata pređe 75% veličine niza. U operaciji *proširi* jednostavno formiramo novi niz koji je za određeni faktor veći od postojećeg (ovdje je taj faktor 1.6, što je unaprijed zadano u konstruktoru), nakon čega svaki postojeći element hash-tabele ponovo dodamo u hash-tabelu koja sada koristi taj veći niz:

```
class HashTabela:
    …
    def proširi(self):
        # Operacija math.ceil(x) daje najmanji cijeli broj veći ili
        # jednak x.
```

```
        self._veličina = math.ceil(self._veličina * self._proširenje)

        # novi, veći niz za vrijednosti
        self._vrijednosti = [None] * self._veličina

        # Moramo sačuvati prethodnu listu svih elemenata, inače bi je
        # izgubili nakon sljedećeg reda.
        lista = self._elementi

        # nova lista svih vrijednosti
        self._elementi = VezanaLista()

        # Ovdje prolazimo kroz postojeću listu svih elemenata i svaki od
        # njih ponovo dodajemo u hash-tabelu koja sada koristi veći
        # niz.
        e = lista.prvi()
        while e != None:
            v = e._podatak
            self[v._ključ] = v._vrijednost
            e = lista.sljedeći(e)
```

U nastavku operacije *__setitem__* izračunamo indeks na koji treba smjestiti zadanu vrijednost koristeći hash-funkciju koju smo pridružili varijabli *_fn* u konstruktoru. S obzirom da se više vrijednosti može nalaziti na istom indeksu hash-tabele, na svakom tom indeksu nalazit će se vezana lista koja sadrži te vrijednosti. Prema tome, ako se na nekom indeksu niza nalazi vrijednost *None* (kojom smo taj niz inicijalizirali u konstruktoru) onda na taj indeks prvo moramo postaviti vezanu listu, a zatim u nju dodati vrijednost koja pripada tom indeksu. Ako se, međutim, na tom indeksu već nalazi vezana lista onda moramo provjeriti da li se vrijednost s tim ključem već nalazi u toj listi, zato jer se vrijednost pridružena jednom ključu može promijeniti. Na primjer,

```
ht['Isaac Newton'] = 'fizičar'
ht['Isaac Newton'] = 'matematičar'
```

U ovom slučaju za drugo pridruživanje moramo iz vezane liste ukloniti vrijednost *'fizičar'* s ključem *'Isaac Newton'*, odnosno dotični objekt klase *Vrijednost*, da bi je zamijenili s vrijednošću *'matematičar'* s istim ključem.
U posljednja dva reda operacije *__setitem__* vrijednost koju smo dodali u hash-tabelu dodajemo u listu koja sadrži sve elemente hash-tabele. Operacija *dodaj* klase *VezanaLista* vraća objekt klase *Element* (to su objekti koje smještamo u vezanu listu). Taj objekt u posljednjem redu pridružujemo elementu koji smo upravo dodali u hash-tabelu tako da on praktički sadrži element vezane liste koji sadrži njega samog! Ovdje se treba podsjetiti na činjenicu da svaka varijabla u Pythonu u stvari sadrži *adresu* nekog objekta, kao što smo vidjeli u trećem poglavlju. Prema tome, varijabla *_element_liste* sadržavat će adresu objekta klase *Element* koji je jedan od elemenata vezane liste. Taj će element, nadalje, u varijabli *_podatak* sadržavati adresu objekta klase *Vrijednost* koji je dodan u hash-tabelu. Razlog iz kojeg nam trebaju ove dvije posljednje naredbe je taj da kada uklanjamo neki

element iz hash-tabele taj element također moramo ukloniti i iz liste svih elemenata, što je lako kada imamo objekt koji je element te liste jer se operaciji *ukloni* klase *VezanaLista* taj objekt mora zadati kao argument. Prema tome, element vezane liste pridružujemo varijabli *_element_liste* objekta klase *Vrijednost* koji dodajemo u hash-tabelu, tako da ako poslije želimo ukloniti tu vrijednost iz hash-tabele onda odmah imamo element koji je sadrži u ovoj vezanoj listi i koji možemo dati kao argument za operaciju *ukloni*. U operaciji *__setitem__* koristimo pomoćnu operaciju *_ukloni_element* kojom ujedno uklanjamo taj element iz liste na indeksu niza na kojem se on nalazi i iz liste svih elemenata. Ovu ćemo operaciju također koristiti u glavnoj operaciji za uklanjanje elementa iz hash-tabele:

```
class HashTabela:
    …
    def _ukloni_element(self, ht_lista, e):
        ht_lista.ukloni(e)     # ukloni iz hash-tabele

        # ukloni iz liste svih elemenata
        self._elementi.ukloni(e._podatak._element_liste)
```

Sljedeća operacija je *__getitem__* koja daje vrijednost pridruženu zadanom ključu:

```
class HashTabela:
    …
    def __getitem__(self, ključ):
        indeks = self._fn(ključ, self._veličina)
        lista = self._vrijednosti[indeks]
        e = self._nađi_vrijednost(lista, ključ)
        if e != None:
            return e._podatak._vrijednost
        else:
            # zadani ključ ne postoji u hash-tabeli
            raise PogrešanKlјuč()
```

Ova operacija je jednostavnija od operacije *__setitem__* jer ovdje moramo samo izračunati indeks pomoću hash-funkcije i zatim u listi koja se nalazi na tom indeksu pronaći vrijednost sa zadanim ključem. Ako vrijednost s tim ključem ne postoji, ili na tom indeksu uopće nema liste (jer se na njemu ne nalazi niti jedna vrijednost), onda ova operacija signalizira grešku tipa *PogrešanKlјuč*.
U ovim operacija koristimo pomoćnu operaciju *_nađi_vrijednost* koju ćemo također definirati kao člana ove klase:

```
class HashTabela:
    …
    def _nađi_vrijednost(self, lista, ključ):
        if lista != None:
            e = lista.nađi(lambda v: v._ključ == ključ)
            return e
        else:
```

```
            return None
```

Još jedna operacija koju treba definirati je operacija za uklanjanje vrijednosti iz hash-tabele, koju ćemo zvati *ukloni*:

```
class HashTabela:
    …
    def ukloni(self, ključ):
        indeks = self._fn(ključ, self._veličina)
        lista = self._vrijednosti[indeks]
        e = self._nađi_vrijednost(lista, ključ)
        if e != None:
            self._ukloni_element(lista, e)
        else:
            # zadani ključ ne postoji u hash-tabeli
            raise PogrešanKlјuč()
```

Vidimo da ova operacija u prvom dijelu radi isto kao i *__getitem__*, odnosno prvo izračuna indeks i zatim pokušava naći vrijednost pridruženu zadanom ključu u listi na tom indeksu. Ako takva vrijednost postoji onda je jednostavno uklonimo iz liste hash-tabele i iz liste svih elemenata, koristeći operaciju *_ukloni_element*, u suprotnom signaliziramo grešku.

9.2 Performanse hash-tabela

Pod uvjetom da imamo dobru hash-funkciju, odnosno takvu koja raspoređuje ključeve u hash-tabeli tako da u većini slučajeva samo jedan ključ odgovara jednom indeksu, performanse hash-tabela uvelike nadmašuju binarno pretraživanje. Kao što je već rečeno, performanse indeksiranja hash-tabele (to jest traženja vrijednosti) u tom je slučaju blizu *O(1)*, što je znatno bolje od $O(log_2 n)$. Bez dobre hash-funkcije, međutim, hash-tabele mogu imati i puno lošije performanse od binarnog pretraživanja, kao što je ilustrirano na početku ovog dijela. Općenito, ako ne možemo pronaći dobru hash-funkciju za vrijednosti s kojima radimo bolje je koristiti binarno pretraživanje. U nekim slučajevima, međutim, nismo sigurni u performanse strukture podataka koju koristimo, gdje nam tada preostaje to da njihove performanse jednostavno mjerimo eksperimentiranjem i na osnovu toga zaključimo koja je efikasnija.

Zadaci za vježbu

1. Upotrebom mape napišite funkciju *frekv* koja vraća broj pojavljivanja zadanog elementa u listu. Primjerice, broj 1 se pojavljuje četiri puta a listi:

```
frekv([1, 2, 1, 2, 3, 1, 1, 4], 1)
=> 4
```

2. Upotrebom mape napišite funkciju *grupe* koja grupira iste elemente u listu ili ntorku. Na primjer,

```
grupe([1, 2, 1, 2, 3, 1, 1, 4])
=> [(1, 1, 1, 1), (2, 2), (3,), (4,)]
```

3. Upotrebom mape napišite program koji daje riječ koja se najčešće pojavljuje u nekom zadanom tekstu. Možete iskoristiti program iz zadatka 1.

4. Implementirajte jednostavnu hash-tabelu klasom *Mapa* kod koje ključ može biti samo string i koja se upotrebljava na sljedeći način:

```
m = Mapa()  # instanciranje

# postavi(k, v) postavlja vrijednost kljuca k na vrijednost v
m.postavi("a", 10)

# kljuc(k) vraca vrijednost kljuca k
v = m.kljuc("a")  # v je 10
v = m.kljuc("b")  # iznimka!

# m.postoji(k) vraca True ako kljuc k postoji u mapi m, ili False ako ne postoji
m.postoji("a")  # vraca True
m.postoji("test")  # vraca False
```

U slučaju da ključ ne postoji u mapi, metoda *kljuc* treba signalizirati iznimku tipa *NepostojeciKljuc*.
Za hash-funkciju treba koristiti i implementirati sljedeći algoritam:

```
hash(s):
   h = 7
   for c in s:
      h = h*31+ord(c)  # ord(c) vraca ASCII kôd znaka c
   return h % 10  # % daje ostatak cjelobrojnog dijeljenja
```

Za slučajeve kolizija treba koristiti niz (Pythonova lista, [...]).
Vaša implementacija ne smije upotrebljavati Pythonovu mapu ili bilo koju biblioteku koja već implementira hash-tabelu (kao što je *collections.OrderedDict* i sl.)