MATERIJAL ZA STUDENTE · OSNOVE PROGRAMIRANJA

Uvod u Python

Od prvog programa do funkcija: sintaksa, promenljive, operatori, tipovi podataka, uslovi, petlje i funkcije, uz primere koda i izlaza.

Kako čitati ovaj materijal

```
print("Ovako izgleda Python kôd")
```

```
Ovako izgleda izlaz programa
```

> **Napomena:** Žuti okviri ističu važne napomene i savete, a plavi objašnjenja primera.

Sadržaj

1. Uvod u Python (Python Introduction) 3
2. Ulaz i izlaz u Python-u (Input and Output in Python) 5
3. Promenljive u Python-u (Python Variables) 8
4. Operatori u Python-u (Python Operators) 16
5. Ključne reči u Python-u (Python Keywords) 24
6. Tipovi podataka u Python-u (Python Data Types) 27
7. Uslovne naredbe u Python-u (Conditional Statements) 33
8. Petlje u Python-u (Loops in Python) 45
9. Funkcije u Python-u (Python Functions) 52
10. Polja u Python-u (Arrays) 61
11. Lančane liste u Python-u (Linked Lists) 92
12. Lančane liste: vežbe (Linked Lists Exercises) 130
13. Stek i red (Stacks and Queues) 166

POGLAVLJE 1

Uvod u Python (Python Introduction)

Python je programski jezik visokog nivoa poznat po svojoj jednostavnoj i čitljivoj sintaksi. Poseduje sledeće karakteristike:

- **Omogućava pisanje čistog koda** sa manjim brojem linija.
- **Podržava više programskih paradigmi**, uključujući objektno-orijentisano, funkcionalno i proceduralno programiranje.
- **Široko se koristi** u veb razvoju (*web development*), automatizaciji, analizi podataka, veštačkoj inteligenciji i mnogim drugim oblastima.
- **Dinamički je tipiziran** (*dynamically typed*) i ima automatsko upravljanje memorijom / sakupljanje smeća (*automatic garbage collection*).

Program "Hello World" (Hello World Program)

Program *Hello World* prikazuje osnovnu strukturu Python programa:

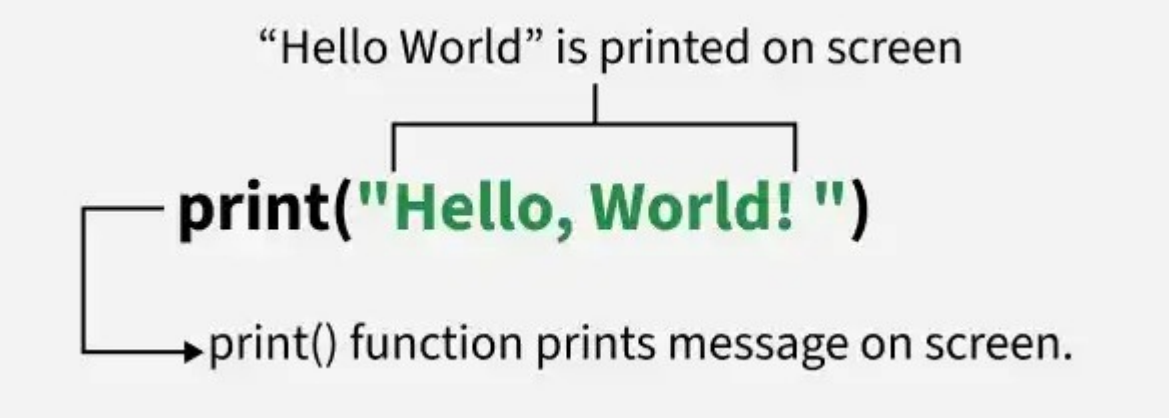
```
# Ovo je komentar. On se neće izvršavati.
print("Hello, World!")
```

IZLAZ (OUTPUT):

```
Hello, World!
```

Kako ovo radi (How does this work):

1. `# Ovo je komentar`: Linije koje počinju sa znakom `#` predstavljaju komentare u Python-u. Interpretator ih potpuno ignoriše prilikom izvršavanja koda i služe isključivo ljudima za dokumentovanje i lakše razumevanje koda.
2. `print(...)`: Ugrađena funkcija u Python-u koja služi za ispisivanje podataka na standardni izlaz (ekran/konzolu).
3. `"Hello, World!"`: Tekst unutar navodnika predstavlja **nisku** (*string*). Funkcija `print()` preuzima ovu nisku i prikazuje je na ekranu.



Slika 1. *Delovi naredbe print("Hello, World!")*

- `print()` je ugrađena Python funkcija koja daje instrukciju računaru da prikaže tekst na ekranu.
- `"Hello, World!"` je tekstualni niz (*string*) obavijen navodnicima (mogu se koristiti i jednostruki `'` i dvostruki `"` navodnici).
- **Python koristi uvlačenje koda / indentaciju** (razmake ili tabulatore) za definisanje blokova koda, umesto vitičastih zagrada `{}` koje se koriste u jezicima C, C++ i Java.
- **Znak `#`** se koristi za pisanje komentara u Python-u. Komentari se ignorišu tokom izvršavanja programa.

POGLAVLJE 2

Ulaz i izlaz u Python-u (Input and Output in Python)

Funkcija `print()` se koristi za izlaz (prikaz) u različitim formatima, dok funkcija `input()` omogućava interakciju sa korisnikom.

Unos podataka pomoću `input()`

Funkcija `input()` u Python-u služi za preuzimanje korisničkog unosa. Podrazumevano, ona vraća uneti podatak u obliku **niske (string-a)**.

```
name = input("Enter your name: ")
print("Hello,", name, "! Welcome!")
```

IZLAZ (OUTPUT):

```
Enter your name: GeeksforGeeks
Hello, GeeksforGeeks ! Welcome!
```

Kôd traži od korisnika da unese svoje ime, čuva ga u promenljivoj `name`, a zatim ispisuje poruku dobrodošlice adresiranu na uneto ime.

Štampanje izlaza pomoću `print()`

Funkcija `print()` nam omogućava prikaz tekstualnih poruka, promenljivih i izraza na konzoli. U donjem primeru, `"Hello, World!"` je tekstualni literal obavijen dvostrukim navodnicima. Kada se izvrši, ova naredba ispisuje tekst na konzolu.

```
print("Hello, World!")
```

IZLAZ (OUTPUT):

```
Hello, World!
```

Ispisivanje promenljivih

Funkciju `print()` možemo koristiti za ispisivanje jedne ili više promenljivih. Više promenljivih ispisujemo tako što ih razdvojimo zarezima.

```
s = "Brad"
print(s)

s = "Anjelina"
age = 25
city = "New York"
print(s, age, city)
```

IZLAZ (OUTPUT):

```
Brad
Anjelina 25 New York
```

Takođe možemo preuzeti više unosa odjednom u jednoj liniji, pri čemu se vrednosti koje je korisnik uneo dele u zasebne promenljive pomoću metode `.split()`:

```
x, y = input("Enter two numbers: ").split()
print(x, y)
```

IZLAZ (OUTPUT):

```
Enter two numbers: 10 20
10 20
```

Preuzimanje različitih tipova korisničkog unosa

Podrazumevani tip korisničkog unosa (string) možemo promeniti u bilo koji drugi tip (`int`, `float`, itd.) pomoću konverzije tipa (*typecasting*).

```
i = int(input("How old are you?: "))
f = float(input("Evaluate 7/2: "))
print(i, f)
```

IZLAZ (OUTPUT):

```
How old are you?: 23
Evaluate 7/2: 3.5
23 3.5
```

POGLAVLJE 3

Promenljive u Python-u (Python Variables)

Promenljive se koriste za skladištenje podataka koji se mogu referencirati i modifikovati tokom izvršavanja programa. Promenljiva je u suštini naziv (ime) koje je dodeljeno određenoj vrednosti.

Za razliku od Java-e i mnogih drugih jezika, promenljive u Python-u ne zahtevaju eksplicitnu deklaraciju tipa. Tip promenljive se **zaključuje na osnovu dodeljene vrednosti**.

```
x = 5
name = "Alex"
print(x)
print(name)
```

IZLAZ (OUTPUT):

```
5
Alex
```

Pravila za imenovanje promenljivih (Rules for Naming Variables)

Da biste pravilno koristili promenljive, potrebno je poštovati sledeća pravila imenovanja:

- Nazivi mogu sadržati **slova, cifre i donje crte (`_`)**.
- Prvi karakter **ne može biti cifra**.
- Nazivi su **osetljivi na velika i mala slova** (*case-sensitive*), pa se `myVar` i `myvar` tretiraju kao dve različite promenljive.
- **Rezervisane reči** (*keywords*) kao što su `if`, `else` i `for` ne mogu se koristiti kao nazivi promenljivih.

Važeći (ispravni) nazivi:

```
age = 21
_colour = "lilac"
total_score = 90
```

Nevažeći (neispravni) nazivi:

```
1name = "Error"     # Počinje cifrom
class = 10          # "class" je rezervisana reč
user-name = "Doe"   # Sadrži crticu (-)
```

Dodeljivanje vrednosti promenljivim (Assigning Values to Variables)

1. **Osnovno dodeljivanje:** Vrednosti se dodeljuju promenljivim pomoću operatora dodeljivanja `=`.

```
x = 5
y = 3.14
z = "Hi"
```

2. **Dinamičko tipiziranje (*Dynamic Typing*):** Python je dinamički tipiziran jezik, što znači da ista promenljiva tokom izvršavanja programa može skladištiti podatke različitih tipova.

```
x = 10              # x je ceo broj (int)
x = "Now a string"  # x sada postaje tekstualni niz (string)
```

3. **Dodeljivanje iste vrednosti:** Ista vrednost se može dodeliti višestrukim promenljivim u jednoj liniji koda.

```
a = b = c = 100
print(a, b, c)
```

IZLAZ (OUTPUT):

```
100 100 100
```

4. **Dodeljivanje različitih vrednosti:** Većem broju promenljivih mogu se dodeliti različite vrednosti u jednoj liniji koda.

```
x, y, z = 1, 2.5, "Python"
print(x, y, z)
```

IZLAZ (OUTPUT):

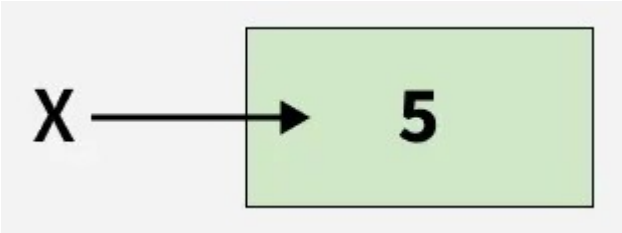
```
1 2.5 Python
```

Koncept referenciranja objekata (Concept of Object Reference)

Razmotrimo dodeljivanje vrednosti promenljivoj `x`:

```
x = 5
```

Kada se linija `x = 5` izvrši, Python u memoriji kreira **objekat** koji predstavlja vrednost `5` i usmerava promenljivu `x` da **referencira** taj objekat.



Slika 2. *Promenljiva x referencira objekat 5*

Sada, dodelimo drugu promenljivu `y` promenljivoj `x`.

```
y = x
```

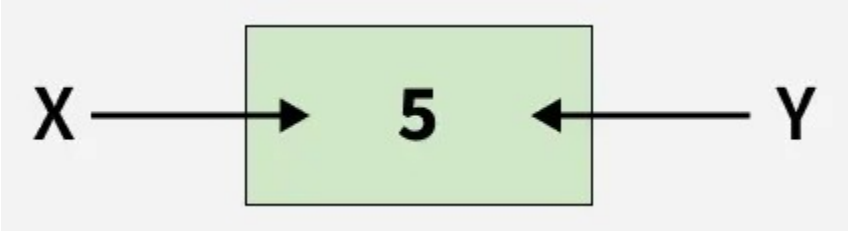
Ova naredba kreira promenljivu `y` i usmerava je da referencira **isti objekat** kao i `x`, a ne samu promenljivu `x`. Ovo se naziva **deljeno referenciranje (*Shared Reference*)**, gde više promenljivih pokazuje na isti objekat u memoriji.

Kako ovo izgleda u memoriji?

```
x ─┐
   ├──> [ Objekat: 5 ]
y ─┘
```

Kada promenite vrednost jedne promenljive dodeljivanjem nove vrednosti (npr. `x = 10`), Python **ne menja** sam objekat `5` (jer su celi brojevi nemutabilni / neizmenjivi), već kreira novi objekat `10` i preusmerava `x` da pokazuje na njega. Promenljiva `y` i dalje ostaje vezana za objekat `5`.

```
x = 5
y = x

print(x, y)  # Izlaz: 5 5

x = 10       # x sada referencira novi objekat 10

print(x, y)  # Izlaz: 10 5
```



Slika 3. *Deljena referenca: x i y pokazuju na isti objekat 5*

Sada, ako napišemo:

```
x = 'Geeks'
```

Python kreira novi objekat za vrednost `"Geeks"` i usmerava promenljivu `x` da referencira ovaj novi objekat.

Stanje u memoriji nakon ove naredbe

Prethodno su i `x` i `y` pokazivali na objekat sa vrednošću `5`. Nakon izvršavanja naredbe `x = 'Geeks'`:

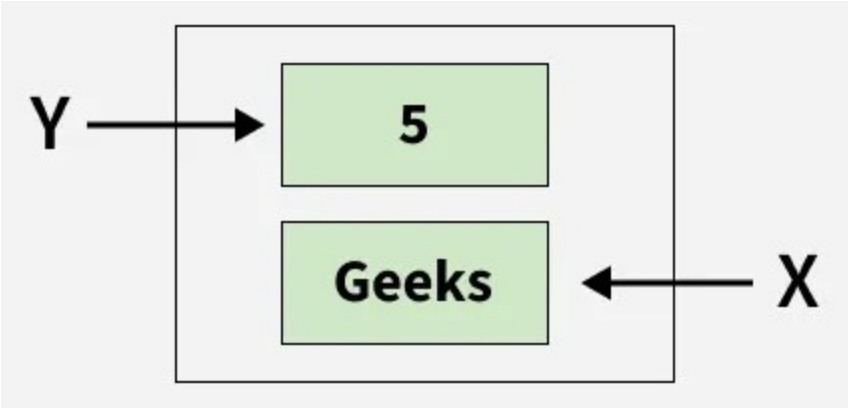
1. **`x`** sada referencira novi tekstualni objekat (string) `'Geeks'`.
2. **`y`** i dalje nastavlja da referencira originalni objekat `5`.

```
x ——> [ Objekat: 'Geeks' ]

y ——> [ Objekat: 5 ]
```

Primer u kodu:

```
x = 5
y = x       # y i x referenciraju isti objekat (5)

x = 'Geeks' # x sada referencira novi objekat ('Geeks')

print(x)    # Izlaz: Geeks
print(y)    # Izlaz: 5
```

Ovo jasno ilustruje da promenljive u Python-u funkcionišu kao **labele ili pokazivači na objekte u memoriji**, a ne kao fiksne memorijske lokacije u koje se podaci upisuju.



Slika 4. *Posle x = 'Geeks' promenljiva x dobija novi objekat, y ostaje vezana za 5*

Promenljiva `y` ostaje nepromenjena, i dalje referencirajući originalni objekat `5`. Sada, ako dodelimo novu vrednost promenljivoj `y`:

```
y = "Computer"
```

Python kreira još jedan novi objekat u memoriji za tekstualnu vrednost `"Computer"` i usmerava promenljivu `y` da referencira taj novi objekat.

Šta se desilo sa originalnim objektom 5?

Nakon što su i `x` i `y` preusmereni na nove objekte (`'Geeks'` i `'Computer'`), na objekat `5` više **ne ukazuje nijedna promenljiva**.

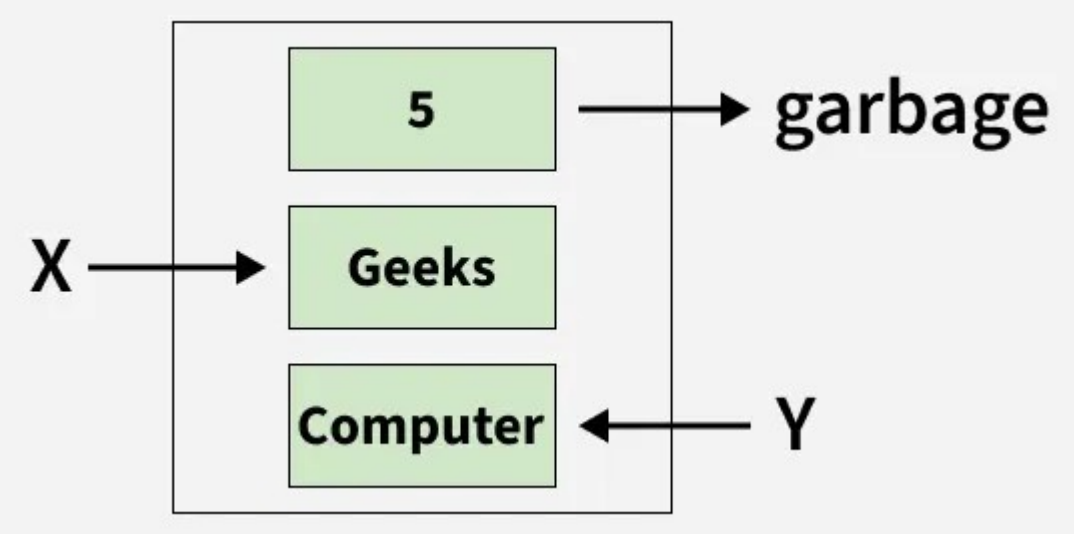
```
x ——> [ Objekat: 'Geeks' ]

y ——> [ Objekat: 'Computer' ]

       [ Objekat: 5 ]  <—— (Nema referenci! Sakupljač smeća ga uklanja)
```

U ovom trenutku nastupa Python-ov mehanizam za **automatsko sakupljanje smeća (*Automatic Garbage Collection*)**:

- Brojač referenci (*reference count*) za objekat `5` pada na **0**.
- Sakupljač smeća prepoznaje da je objekat nedostupan i oslobađa memoriju koju je on zauzimao.

Kompletan prikaz u kodu:

```
x = 5
y = x          # Obje promenljive pokazuju na objekat 5

x = 'Geeks'    # x dobija novu referencu ('Geeks')
y = 'Computer' # y dobija novu referencu ('Computer')

print(x)  # Izlaz: Geeks
print(y)  # Izlaz: Computer
```



Slika 5. *Objekat 5 bez referenci postaje kandidat za sakupljanje smeća*

Python promenljive skladište **reference na objekte**, a ne same vrednosti. Kada se promenljivoj dodeli nova vrednost, ona počinje da referencira novi objekat, dok stari objekat na koji više niko ne ukazuje postaje kandidat za automatsko sakupljanje smeća (*garbage collection*).

Doseg (opseg) promenljive (*Scope*) određuje gde se u Python programu toj promenljivoj može pristupiti (na primer, unutar funkcije ili u celom programu).

Razumevanje ponovnog dodeljivanja vrednosti (Understanding Variable Reassignment)

U ovom primeru proveravamo da li izmena jedne promenljive utiče na drugu kada obe u početku referenciraju isti objekat.

```
x = 1
y = x
y = y + 1

print(x)
print(y)
```

IZLAZ (OUTPUT):

```
1
2
```

Objašnjenje: U početku, i `x` i `y` referenciraju objekat `1`. Nakon izvršavanja `y = y + 1`, promenljiva `y` počinje da referencira novi objekat `2`, dok `x` i dalje referencira objekat `1`. Zbog toga izmena promenljive `y` ne utiče na `x`.

Brisanje promenljive (Deleting a Variable)

Ključna reč `del` koristi se za brisanje promenljive iz memorije. Nakon brisanja, promenljivoj se više ne može pristupiti.

```
x = 10
del x
print(x)
```

IZLAZ (OUTPUT):

```
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 3, in <module>
NameError: name 'x' is not defined
```

> **Objašnjenje:** Naredba `del x` briše promenljivu `x`. Pristupanje promenljivoj `x` nakon brisanja izaziva grešku `NameError` jer ta promenljiva više ne postoji u opsegu.

Praktični primeri (Practical Examples)

1. Zamena vrednosti dve promenljive (Swapping Two Variables)

Korišćenjem višekratnog dodeljivanja u jednoj liniji, možemo zameniti vrednosti dve promenljive bez potrebe za privremenom promenljivom (*temporary variable*).

```
a, b = 5, 10
a, b = b, a
print(a, b)
```

IZLAZ (OUTPUT):

```
10 5
```

2. Prebrojavanje karaktera u stringu (Counting Characters in a String)

Dodelite rezultat operacije nad stringom promenljivoj u jednoj liniji koda.

```
word = "Python"
length = len(word)
print("Length of the word:", length)
```

IZLAZ (OUTPUT):

```
Length of the word: 6
```

POGLAVLJE 4

Operatori u Python-u (Python Operators)

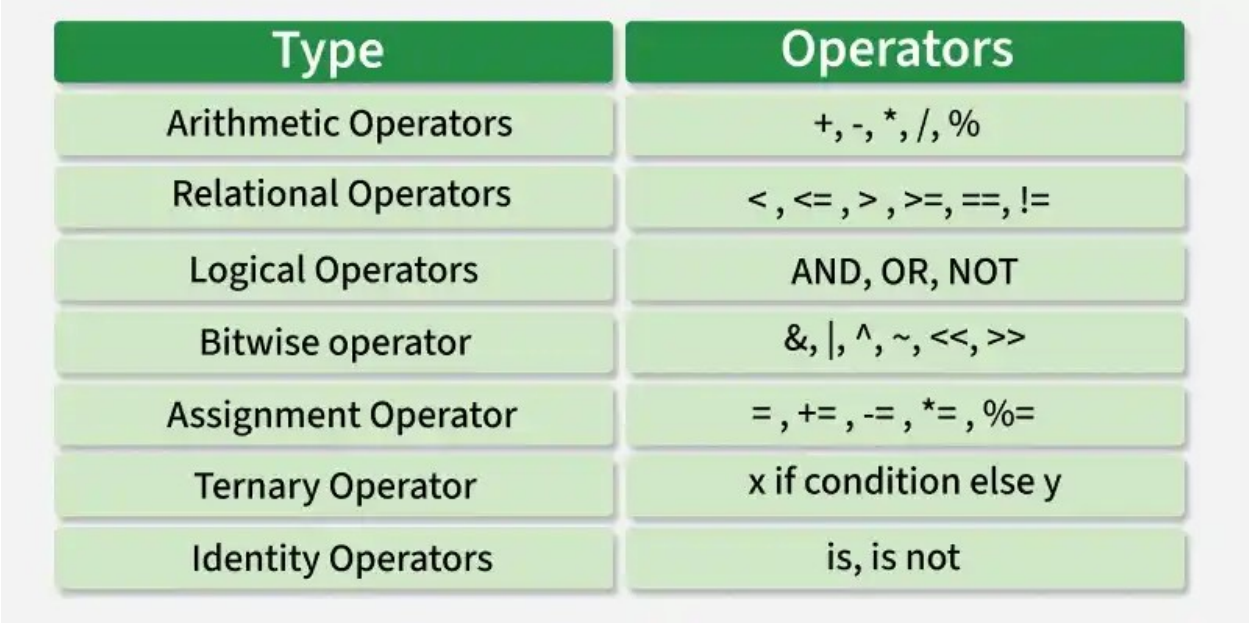
U Python programiranju, operatori se generalno koriste za izvođenje operacija nad vrednostima i promenljivim.

- **Operatori (*Operators*):** Specijalni simboli kao što su `+`, `-`, `*`, `/` i drugi.
- **Operandi (*Operands*):** Vrednosti ili promenljive nad kojima se operator primenjuje (npr. u izrazu `a + b`, `a` i `b` su operandi).

Vrste operatora u Python-u (Types of Operators in Python)

Python podržava sledeće glavne kategorije operatora:

1. **Aritmetički operatori** (*Arithmetic Operators*) – za osnovne matematičke račune (`+`, `-`, `*`, `/`, `%`, `**`, `//`).
2. **Operatori poređenja** (*Comparison / Relational Operators*) – za poređenje vrednosti (`==`, `!=`, `>`, `<`, `>=`, `<=`).
3. **Logički operatori** (*Logical Operators*) – za spajanje uslova (`and`, `or`, `not`).
4. **Operatori dodeljivanja** (*Assignment Operators*) – za dodeljivanje i ažuriranje vrednosti (`=`, `+=`, `-=`, `*=`, `/=`, itd.).
5. **Bitovni operatori** (*Bitwise Operators*) – za rad na nivou bitova (`&`, `|`, `^`, `~`, `<<`, `>>`).
6. **Operatori pripadnosti** (*Membership Operators*) – za proveru prisustva elementa u sekvenci (`in`, `not in`).
7. **Operatori identiteta** (*Identity Operators*) – za proveru da li dve promenljive referenciraju isti objekat u memoriji (`is`, `is not`).



Type	Operators
Arithmetic Operators	+, -, *, /, %
Relational Operators	< , <= , > , >=, ==, !=
Logical Operators	AND, OR, NOT
Bitwise operator	&, |, ^, ~, <<, >>
Assignment Operator	= , += , -= , *= , %=
Ternary Operator	x if condition else y
Identity Operators	is, is not

Slika 6. *Pregled vrsta operatora u Python-u*

1. Aritmetički operatori (Arithmetic Operators)

Aritmetički operatori se koriste za izvođenje osnovnih matematičkih operacija kao što su sabiranje, oduzimanje, množenje i deljenje.

U Python-u, operator običnog deljenja (`/`) uvek vraća rezultat u vidu broja sa pokretnim zarezom (*float*), dok deljenje bez ostatka (`//`, *floor division*) vraća ceo broj (*integer*).

```
a = 15
b = 4

print("Addition:", a + b)        # Sabiranje
print("Subtraction:", a - b)     # Oduzimanje
print("Multiplication:", a * b)  # Množenje
print("Division:", a / b)        # Deljenje (vraca float)
print("Floor Division:", a // b) # Celobrojno deljenje
print("Modulus:", a % b)         # Ostatak pri deljenju
print("Exponentiation:", a ** b) # Stepenovanje
```

IZLAZ (OUTPUT):

```
Addition: 19
Subtraction: 11
Multiplication: 60
Division: 3.75
Floor Division: 3
Modulus: 3
Exponentiation: 50625
```

2. Operatori poređenja (Comparison Operators)

Operatori poređenja (ili relacioni operatori) porede dve vrednosti i vraćaju bulevsku vrednost — `True` (Tačno) ili `False` (Netačno), u zavisnosti od uslova.

```
a = 13
b = 33

print(a > b)   # Veće od
print(a < b)   # Manje od
print(a == b)  # Jednako
print(a != b)  # Različito
print(a >= b)  # Veće ili jednako
print(a <= b)  # Manje ili jednako
```

IZLAZ (OUTPUT):

```
False
True
False
True
False
True
```

3. Logički operatori (Logical Operators)

Logički operatori izvode logičke operacije **AND** (I), **OR** (ILI) i **NOT** (NE). Koriste se za spajanje više uslovnih iskaza.

Redosled prioriteta logičkih operatora u Python-u je:

1. `not` (Logičko NE)
2. `and` (Logičko I)
3. `or` (Logičko ILI)

```
a = True
b = False

print(a and b)  # Tačno samo ako su oba True
print(a or b)   # Tačno ako je bar jedan True
print(not a)    # Negira vrednost
```

IZLAZ (OUTPUT):

```
False
True
False
```

4. Bitovni operatori (Bitwise Operators)

Bitovni operatori rade direktno nad pojedinačnim bitovima vrednosti i izvode operacije bit-po-bit. Koriste se za rad sa binarnim brojevima.

Pregled bitovnih operatora:

- `~` Bitovno NE (*Bitwise NOT*)
- `<<` i `>>` Bitovno pomeranje ulevo/udesno (*Bitwise Shift*)
- `&` Bitovno I (*Bitwise AND*)
- `^` Bitovno ekskluzivno ILI (*Bitwise XOR*)
- `|` Bitovno ILI (*Bitwise OR*)

```
a = 10  # U binarnom: 1010
b = 4   # U binarnom: 0100

print(a & b)   # Bitovno AND
print(a | b)   # Bitovno OR
print(~a)      # Bitovno NOT (Komplement dvojke: -(a + 1))
print(a ^ b)   # Bitovno XOR
print(a >> 2)  # Pomeranje udesno za 2 mesta
print(a << 2)  # Pomeranje ulevo za 2 mesta
```

IZLAZ (OUTPUT):

```
0
14
-11
14
2
40
```

5. Operatori dodeljivanja (Assignment Operators)

Operatori dodeljivanja koriste se za dodeljivanje vrednosti promenljivima. Operator dodeljuje vrednost sa desne strane izraza operandu na levoj strani.

```
a = 10
b = a
print(b)

b += a   # b = b + a
print(b)

b -= a   # b = b - a
print(b)

b *= a   # b = b * a
print(b)

b <<= a  # b = b << a (bitovno pomeranje ulevo)
print(b)
```

IZLAZ (OUTPUT):

```
10
20
10
100
102400
```

6. Operatori identiteta (Identity Operators)

`is` i `is not` su operatori identiteta koji se koriste za proveru da li dve promenljive **referenciraju istu memorijsku lokaciju**. To što su dve promenljive jednake po vrednosti (`==`), ne znači nužno da su i identične u memoriji (`is`).

- **`is`**: Vraća `True` ako operandi referenciraju isti objekat.
- **`is not`**: Vraća `True` ako operandi ne referenciraju isti objekat.

```
a = 10
b = 20
c = a

print(a is not b)
print(a is c)
```

IZLAZ (OUTPUT):

```
True
True
```

7. Operatori pripadnosti (Membership Operators)

`in` i `not in` su operatori pripadnosti koji se koriste za proveru da li se određena vrednost nalazi unutar sekvence (npr. liste, stringa, torke ili skupa).

- **`in`**: Vraća `True` ako se vrednost nalazi u sekvenci.
- **`not in`**: Vraća `True` ako se vrednost ne nalazi u sekvenci.

```
x = 24
y = 20
my_list = [10, 20, 30, 40, 50]

if (x not in my_list):
    print("x is NOT present in given list")
else:
    print("x is present in given list")

if (y in my_list):
    print("y is present in given list")
else:
    print("y is NOT present in given list")
```

IZLAZ (OUTPUT):

```
x is NOT present in given list
y is present in given list
```

8. Ternarni operator (Ternary Operator)

Ternarni operator (poznat i kao uslovni izraz) procenjuje izraz na osnovu toga da li je uslov tačan ili netačan. Dodat je u Python u verziji 2.5.

Omogućava testiranje uslova u jednoj liniji koda i zamenjuje višeličnu `if-else` strukturu, čineći kod kompaktnijim.

Sintaksa:

```
[vrednost_if_true] if [uslov] else [vrednost_if_false]

a, b = 10, 20
min_val = a if a < b else b

print(min_val)
```

IZLAZ (OUTPUT):

```
10
```

Prioritet i asocijativnost operatora (Precedence and Associativity)

Prioritet i asocijativnost operatora određuju redosled kojim se operacije izvršavaju u složenim izrazima.

Prioritet operatora (Operator Precedence)

Kada izraz sadrži više operatora sa različitim prioritetom, prioritet određuje koja će se operacija prva izvršiti (npr. množenje ima veći prioritet od sabiranja, a `and` ima veći prioritet od `or`).

```
expr = 10 + 20 * 30  # Prvo se vrši množenje (20 * 30 = 600), pa sabiranje (+ 10)
print(expr)

name = "Alex"
age = 0

# Prvo se evaluira 'and' (name == "John" and age >= 2 -> False)
# Zatim se evaluira 'or' (name == "Alex" or False -> True)
if name == "Alex" or name == "John" and age >= 2:
    print("Hello! Welcome.")
else:
    print("Good Bye!!")
```

IZLAZ (OUTPUT):

```
610
Hello! Welcome.
```

Asocijativnost operatora (Operator Associativity)

Ako izraz sadrži dva ili više operatora **istog prioriteta**, asocijativnost određuje redosled izvršavanja — sa leve na desnu stranu (*Left-to-Right*) ili sa desne na levu stranu (*Right-to-Left*).

- Većina operatora (deljenje, sabiranje, oduzimanje...) ima asocijativnost **sleva na desno**.
- Operator stepenovanja (`**`) ima asocijativnost **sdesna na levo**.

```
print(100 / 10 * 10)  # Sleva na desno: (100 / 10) * 10 = 10.0 * 10 = 100.0
print(5 - 2 + 3)      # Sleva na desno: (5 - 2) + 3 = 3 + 3 = 6
print(5 - (2 + 3))    # Zagrade menjaju prioritet: 5 - 5 = 0
print(2 ** 3 ** 2)    # Sdesna na levo! Prvo 3**2 = 9, zatim 2**9 = 512
```

IZLAZ (OUTPUT):

```
100.0
6
0
512
```

POGLAVLJE 5

Ključne reči u Python-u (Python Keywords)

Ključne reči (*keywords*) su specijalne rezervisane reči koje čine osnovu samog programskog jezika. One definišu pravila i sintaksnu strukturu Python programa, što znači da se ne mogu koristiti kao nazivi za promenljive, funkcije, klase ili bilo koje druge identifikatore.

Dobijanje liste svih Python ključnih reči

Sve ključne reči dostupne u trenutnoj verziji Python-a možete izlistati pomoću ugrađenog modula `keyword`:

```
import keyword

print("Lista ključnih reči u Python-u:")
print(keyword.kwlist)
```

IZLAZ (OUTPUT):

```
Lista ključnih reči u Python-u:
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',
'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not',
'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Kako prepoznati ključne reči

Neki od načina da prepoznate ključne reči u kodu su:

1. **Isticanje sintakse (*Syntax Highlighting*):** Većina okruženja za razvoj (IDE-ovi kao što su PyCharm, VS Code) automatski boji ključne reči drugačijom bojom ili stilom.
2. **Greške u sintaksi (*SyntaxError*):** Ako pogrešno upotrebite ključnu reč (npr. kao ime promenljive ili funkcije), Python će prijaviti sintaksnu grešku.

Ključne reči kao imena promenljivih

Ako pokušate da iskoristite ključnu reč kao ime promenljive, Python će izbaciti `SyntaxError`.

```
for = 10
print(for)
```

IZLAZ (OUTPUT):

```
  File "main.py", line 1
    for = 10
        ^
SyntaxError: invalid syntax
```

Kategorizacija ključnih reči po kontekstu

Radi lakšeg razumevanja, sve ključne reči možemo podeliti u sledeće kategorije:

Kategorija	Ključne reči
Vrednosti (*Value Keywords*)	`True`, `False`, `None`
Operatori (*Operator Keywords*)	`and`, `or`, `not`, `is`, `in`
Upravljanje tokom (*Control Flow*)	`if`, `else`, `elif`, `for`, `while`, `break`, `continue`, `pass`, `try`, `except`, `finally`, `raise`, `assert`
Funkcije i klase	`def`, `return`, `lambda`, `yield`, `class`
Upravljanje kontekstom	`with`, `as`
Uvoz i moduli	`import`, `from`
Opseg i imenski prostor (*Scope*)	`global`, `nonlocal`
Asinhrono programiranje	`async`, `await`

Ključne reči vs Identifikatori

Ključne reči (Keywords)	Identifikatori (Identifiers)
Rezervisane reči u Python-u koje imaju specifično značenje.	Imena koja daje programer za promenljive, funkcije, klase itd.
Ne mogu se koristiti kao imena promenljivih.	Mogu se proizvoljno imenovati dokle god nisu ključne reči.
Primeri: `if`, `else`, `for`, `while`	Primeri: `x`, `broj`, `suma`, `rezultat`
Sastavni su deo sintakse Python-a.	Korisnički definisana, smislena imena u kodu.
Ne mogu se redefinisati niti menjati.	Programer ih može proizvoljno definisati i menjati.

Promenljive vs Ključne reči

Promenljive (Variables)	Ključne reči (Keywords)
Služe za skladištenje podataka u memoriji.	Rezervisane reči sa unapred definisanim značenjem u jeziku.
Programer ih može praviti, menjati i brisati.	Ne mogu se menjati niti koristiti kao imena promenljivih.
Primeri: `x`, `godine`, `ime`	Primeri: `if`, `while`, `for`
Sadrže vrednosti kojima se manipuliše u programu.	Definišu samu strukturu i logiku Python koda.
Nazivi moraju pratiti pravila imenovanja, ali su fleksibilni.	Fiksirane su samim Python jezikom i ne mogu se menjati.

POGLAVLJE 6

Tipovi podataka u Python-u (Python Data Types)

Tipovi podataka u Python-u definišu vrstu vrednosti koja je sačuvana u promenljivoj i određuju koje se operacije nad tim podacima mogu izvršavati. Pošto Python tretira sve kao **objekat**, svaka vrednost je povezana sa određenim tipom podataka.

Zašto su tipovi podataka važni:

- **Efikasno skladištenje:** Pomažu interpretatoru da razume kako da skladišti i procesira različite vrste podataka na najefikasniji način.
- **Ispravnost operacija:** Omogućavaju izvođenje ispravnih operacija osiguravajući da se vrše samo dozvoljene akcije nad kompatibilnim tipovima podataka (npr. ne možete sabrati broj i tekst bez prethodne konverzije).

Osnovni ugrađeni tipovi podataka u Python-u

Python ima nekoliko ugrađenih kategorija tipova podataka:

Kategorija	Tip podataka	Opis / Primer
Numerički tipovi	`int`, `float`, `complex`	Celobrojne vrednosti (`10`), brojevi sa pokretnim zarezom (`3.14`), kompleksni brojevi (`2 + 3j`)
Tekstualni tip	`str`	Niske karaktera / stringovi (`"Zdravo, svet!"`)
Bulevski tip	`bool`	Logičke vrednosti: `True` (Tačno) ili `False` (Netačno)
Sekvencijalni tipovi	`list`, `tuple`, `range`	Uređeni nizovi podataka (npr. `[1, 2, 3]`, `(10, 20)`)
Tipovi skupova	`set`, `frozenset`	Neuređene kolekcije jedinstvenih elemenata (`{1, 2, 3}`)

Kategorija	Tip podataka	Opis / Primer
Mape / Rečnici	`dict`	Kolekcije parova ključ-vrednost (`{"ime": "Ana", "godine": 25}`)
Tip bez vrednosti	`NoneType`	Predstavlja odsustvo vrednosti (`None`)

Provera tipa podatka (`type()`)

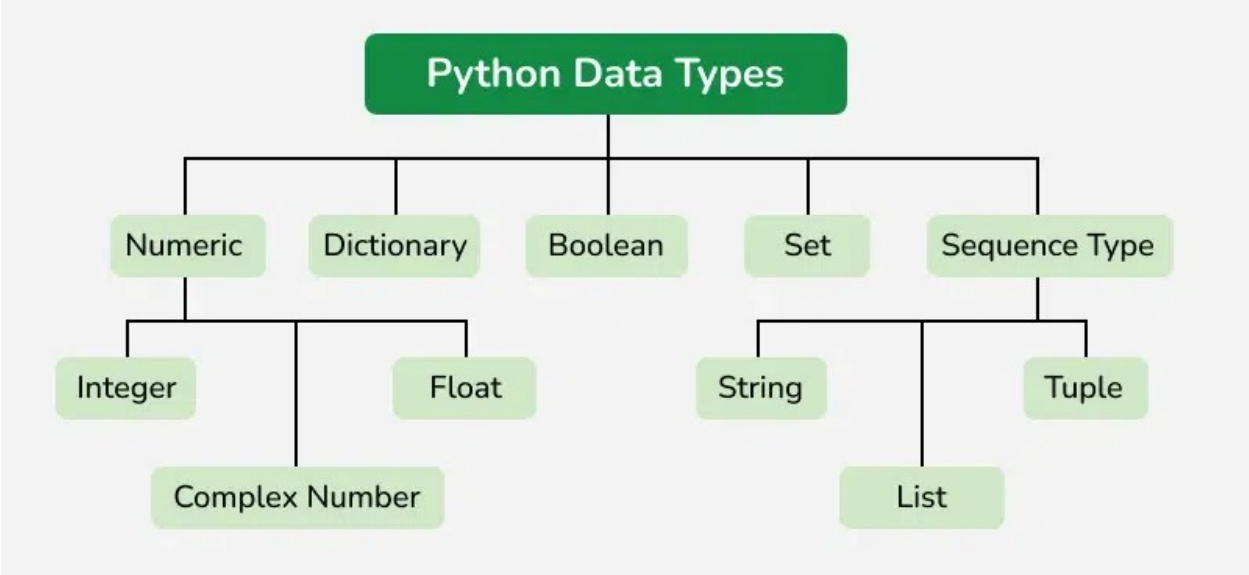
Tip bilo koje promenljive ili vrednosti možete proveriti pomoću ugrađene funkcije `type()`:

```
x = 42
y = 3.14
poruka = "Python"
is_active = True

print(type(x))          # <class 'int'>
print(type(y))          # <class 'float'>
print(type(poruka))     # <class 'str'>
print(type(is_active))  # <class 'bool'>
```

Dinamičko tipiziranje (Dynamic Typing)

Python je **dinamički tipiziran jezik**, što znači da ne morate eksplicitno navoditi tip podatka pri definisanju promenljive. Tip se određuje automatski u trenutku dodeljivanja vrednosti u izvršavanju programa:

```
# Promenljiva x je prvo int
x = 10
print(type(x))  # <class 'int'>

# Istoj promenljivoj dodeljujemo string
x = "Sada sam string"
print(type(x))  # <class 'str'>
```



Slika 7. *Hijerarhija ugrađenih tipova podataka u Python-u*

Pregled tipova podataka u Python-u

Kod u nastavku prikazuje dinamičko dodeljivanje vrednosti promenljivoj `x`. Sa svakim novim dodeljivanjem, prethodna vrednost se zamenjuje, a `x` menja svoj tip podataka:

```
x = 50                          # int
x = 60.5                        # float
x = "Hello World"               # str
x = ["geeks", "for", "geeks"]   # list
x = ("geeks", "for", "geeks")   # tuple
```

1. Numerički tipovi podataka (Numeric Data Types)

Numerički tipovi služe za skladištenje brojčanih vrednosti. Python podržava tri glavna numerička tipa:

- **Integers (`int`):** Celobrojne vrednosti (pozitivni ili negativni celi brojevi bez razlomka ili decimalnog mesta).
- **Float (`float`):** Realni brojevi predstavljeni zapisom sa pokretnim zarezom (sadrže decimalnu tačku).
- **Complex (`complex`):** Kompleksni brojevi koji imaju realni i imaginarni deo (npr. `2 + 3j`).

```
a = 5
b = 5.0
c = 2 + 4j

print(type(a))  # <class 'int'>
print(type(b))  # <class 'float'>
print(type(c))  # <class 'complex'>
```

2. Sekvencijalni tipovi podataka (Sequence Data Types)

Sekvenca je uređena kolekcija elemenata koji mogu biti istog ili različitog tipa. Elementima sekvence pristupa se pomoću **indeksiranja** (prvi element je na indeksu `0`, dok negativni indeksi broje sa kraja: `-1` je poslednji element).

String (`str`)

Služi za skladištenje tekstualnih podataka. Pravi se pomoću jednostrukih, dvostrukih ili trostrukih navodnika:

```
s = 'Welcome to the Geeks World'
print(s)        # Welcome to the Geeks World
print(type(s))  # <class 'str'>

# Pristup karakterima preko indeksa
print(s[1])     # 'e' (drugi karakter)
print(s[-1])    # 'd' (poslednji karakter)
```

Lista (`list`)

Uređena i **promenljiva** (*mutable*) kolekcija koja se koristi za skladištenje više stavki u jednoj promenljivoj. Elementi mogu biti različitih tipova:

```
a = [1, 2, 3]
print(a)  # [1, 2, 3]

b = ["Geeks", "For", "Geeks", 4, 5]
print(b[3])   # 4
print(b[-3])  # 'Geeks'
```

Torka / Tupla (`tuple`)

Uređena i **nepromenljiva** (*immutable*) kolekcija. Nakon što se kreira, njeni elementi se ne mogu menjati niti brisati:

> **Napomena:** Torka sa jednim elementom **mora** sadržati zarez na kraju (npr. `(1,)`), inače će je Python tretirati kao običnu vrednost u zagradama, a ne kao torku.

```
t1 = (1,)
print(type(t1))  # <class 'tuple'>

t2 = ('Geeks', 'For', 'Geeks', 1, 2)
print(t2[3])   # 1
print(t2[-3])  # 'Geeks'
```

3. Bulevski tip podataka (Boolean Data Type)

Predstavlja jednu od dve logičke vrednosti: `True` (Tačno) ili `False` (Netačno). Koristi se uglavnom u uslovima i poređenjima (klasa `bool`).

```
print(type(True))   # <class 'bool'>
print(type(False))  # <class 'bool'>
```

Truthy i Falsy vrednosti

U Python-u se sve vrednosti u bulevskom kontekstu (npr. unutar `if` uslova) procenjuju kao istinite (*truthy*) ili neistinite (*falsy*):

- **Falsy vrednosti:** `0`, `0.0`, `""` (prazan string), `[]` (prazna lista), `()` (prazna torka), `{}` (prazan rečnik), `None`, `False`.
- **Truthy vrednosti:** Sve ostale vrednosti (npr. brojevi različiti od nule, neprazne sekvence).

```
if 1:
    print("1 is truthy")

if not 0:
    print("0 is falsy")
```

4. Skupovi (Set Data Type)

Skupovi (`set`) su **neuređene** i **promenljive** kolekcije koje skladište **isključivo jedinstvene elemente** (automatski uklanjaju duplikate).

> **Napomena:** Pošto su neuređeni, elementima skupova **se ne može pristupati preko indeksa** (`s[0]` vrši grešku). Prolazak kroz skup se obično radi petljom `for`, a redosled pri štampanju može varirati.

```
s1 = {"a", "a", "b", "c", "b"}
print(s1)  # {'b', 'a', 'c'} — duplikati su uklonjeni

s2 = {"Geeks", "For", "Geeks"}
for i in s2:
    print(i)
```

5. Rečnici / Mape (Dictionary Data Type)

Rečnici (`dict`) služe za skladištenje podataka u obliku parova **ključ:vrednost** (`key:value`).

- Svaki ključ u rečniku mora biti **jedinstven**.
- Vrednostima se pristupa pomoću pripadajućeg ključa u uglastim zagradama `d[ključ]` ili metodom `d.get(ključ)`.
- **Ključevi su osetljivi na velika i mala slova** (*case-sensitive*): ključ `Key` i `key` se tretiraju kao dva potpuno različita ključa.

```
d = {1: 'Geeks', 2: 'For', 3: 'Geeks'}

print(d[1])      # 'Geeks'
print(d.get(2))  # 'For'
```

POGLAVLJE 7

Uslovne naredbe u Python-u (Conditional Statements)

Uslovne naredbe se koriste za upravljanje tokom izvršavanja programa na osnovu određenih uslova. Redom procenjuju uslove i omogućavaju programu da izvrši različite blokove koda u zavisnosti od toga da li je uslov tačan (`True`) ili netačan (`False`).

Naredba `if` (If Statement)

Naredba `if` je najosnovniji oblik uslovne logike. Koristi se za izvršavanje bloka koda **isključivo kada je navedeni uslov tačan (`True`)**.

Sintaksa

```
if uslov:
    # blok koda koji se izvršava ako je uslov True
```

> **Važna napomena (Uvlačenje koda / Indentation):** Python koristi uvlačenje koda (razmake ili tabulatore) za definisanje blokova koda. Svi redovi unutar `if` bloka moraju biti uvučeni podjednako (preporuka PEP 8 je 4 razmaka).

Primer upotrebe

```
starost = 18

if starost >= 18:
    print("Punoletni ste.")
```

IZLAZ (OUTPUT):

```
Punoletni ste.
```

Ako uslov iznad vrati `False` (npr. ako je `starost = 15`), Python će jednostavno preskočiti kod unutar `if` bloka i nastaviti sa izvršavanjem sledećih linija izvan njega.

Proširenje: `if-else` i `if-elif-else`

Za potpuniju kontrolu toka najčešće se kombinuju sledeće ključne reči:

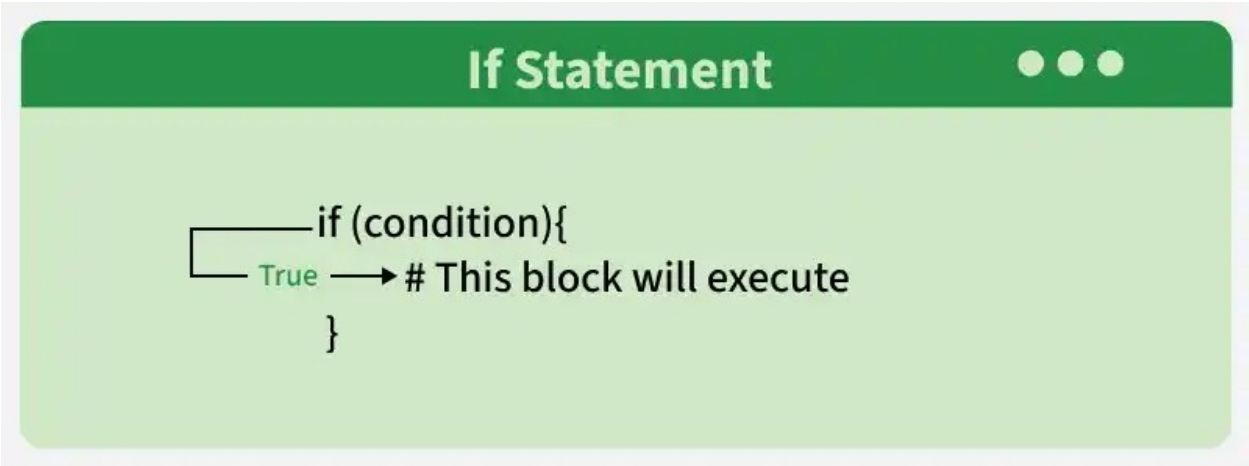
1. **`else`:** Definiše rezervni blok koda koji se izvršava kada je `if` uslov **netačan (`False`)**.
2. **`elif`** (skraćeno od *else if*): Omogućava proveru više uzastopnih uslova. Čim se naiđe na prvi uslov koji je `True`, njegov blok se izvršava, a svi ostali se preskaču.

Primer sa više uslova

```
poeni = 75

if poeni >= 90:
    print("Ocena: 10")
elif poeni >= 80:
    print("Ocena: 9")
elif poeni >= 70:
    print("Ocena: 8")
else:
    print("Ocena: ispod 8")
```

IZLAZ (OUTPUT):

```
Ocena: 8
```



Slika 8. *Struktura if naredbe*

```
age = 20
if age >= 18:
    print("Eligible to vote.")
```

IZLAZ (OUTPUT):

```
Eligible to vote.
```

Kratka `if` naredba (Short Hand if)

Kratki oblik `if` naredbe koristi se za pisanje uslovnog iskaza u jednoj liniji koda. Koristan je kada treba izvršiti samo jednu naredbu.

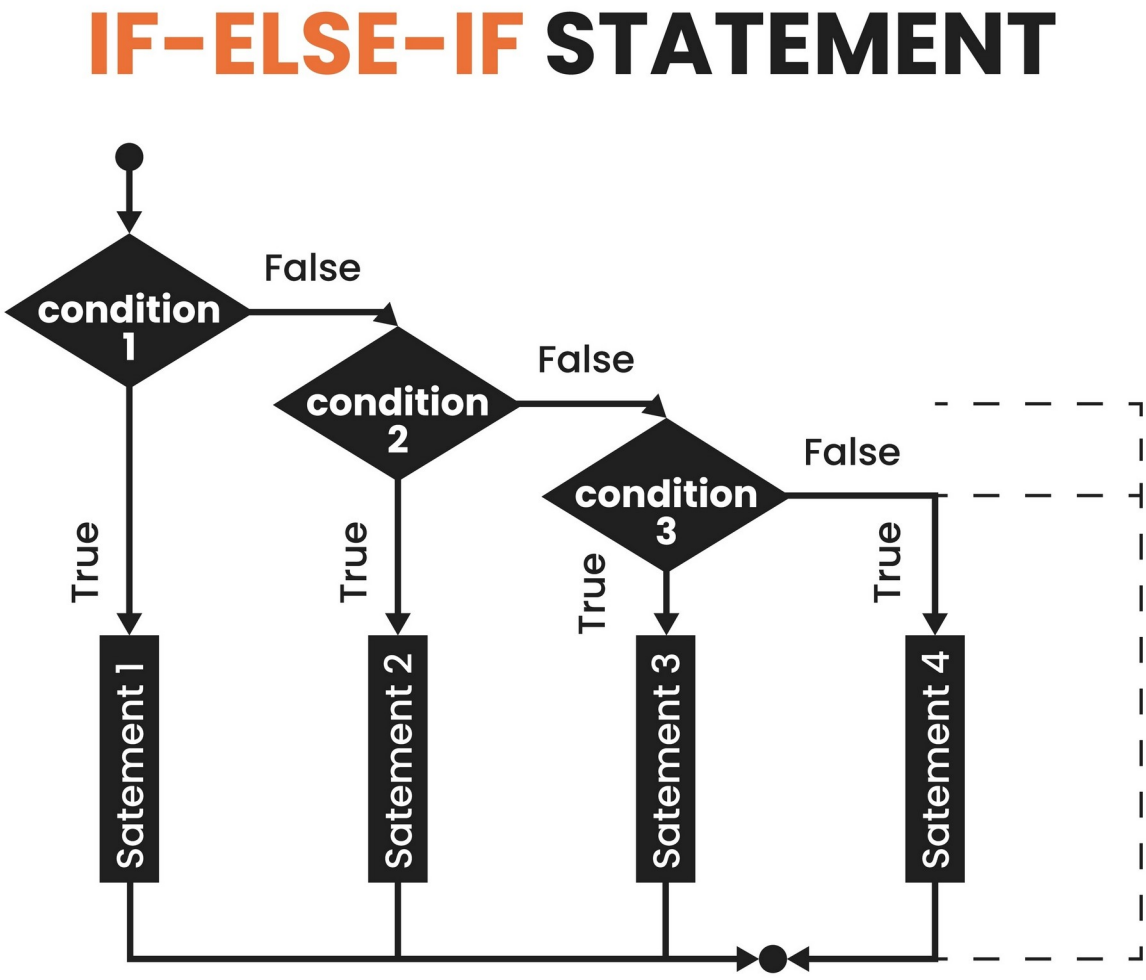
```
age = 19
if age > 18: print("Eligible to Vote.")
```

IZLAZ (OUTPUT):

```
Eligible to Vote.
```

Naredba `if-else` (If else Statement)

Naredba `if-else` koristi se za izvršavanje jednog bloka koda kada je uslov **tačan (`True`)**, a drugog bloka koda kada je uslov **netačan (`False`)**.



Slika 9. *Dijagram toka if-elif-else strukture (izvor: Shutterstock)*

Sintaksa

```
if uslov:
    # blok koda ako je uslov True
else:
    # blok koda ako je uslov False
```

Primer upotrebe

```
age = 16

if age >= 18:
    print("Eligible to vote.")
else:
    print("Not eligible to vote.")
```

IZLAZ (OUTPUT):

```
Not eligible to vote.
```

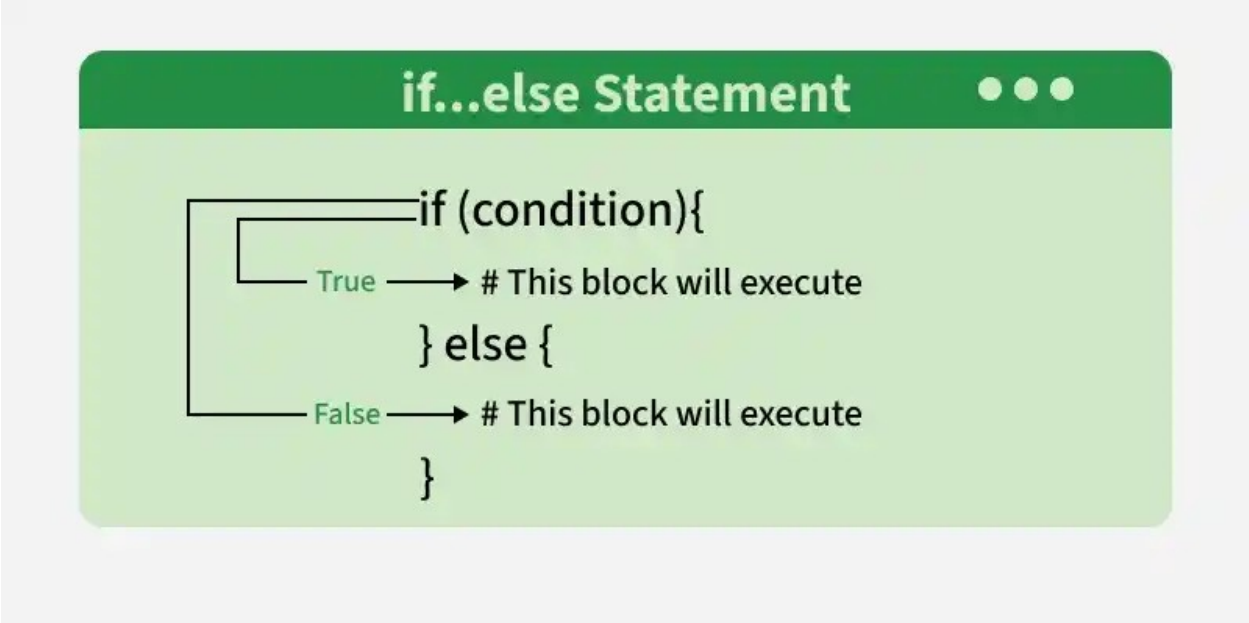
Kratka `if-else` naredba (Short Hand if-else / Ternarni operator)

Kao i kod običnog `if`-a, i `if-else` se može napisati u jednoj liniji koda. Ova forma se u Python-u naziva **ternarni operator** ili uslovni izraz.

```
age = 20
status = "Adult" if age >= 18 else "Minor"
print(status)
```

IZLAZ (OUTPUT):

```
Adult
```



Slika 10. *Struktura if-else naredbe*

```
age = 10
if age <= 12:
    print("Travel for free.")
else:
    print("Pay for ticket.")
```

IZLAZ (OUTPUT):

```
Travel for free.
```

Naredba `if-elif-else` (If-elif-else Statement)

Naredba `elif` (skraćeno od *else if*) koristi se za proveru više uzastopnih uslova u programu. Ona izvršava svoj blok koda kada je njen uslov **tačan (`True`)**, a svi prethodni uslovi su bili **netačni (`False`)**.

Sintaksa

```
if uslov_1:
    # Izvršava se ako je uslov_1 True
elif uslov_2:
    # Izvršava se ako je uslov_1 False, a uslov_2 True
elif uslov_3:
    # Izvršava se ako su uslov_1 i uslov_2 False, a uslov_3 True
else:
    # Izvršava se ako nijedan od prethodnih uslova nije True
```

Primer upotrebe

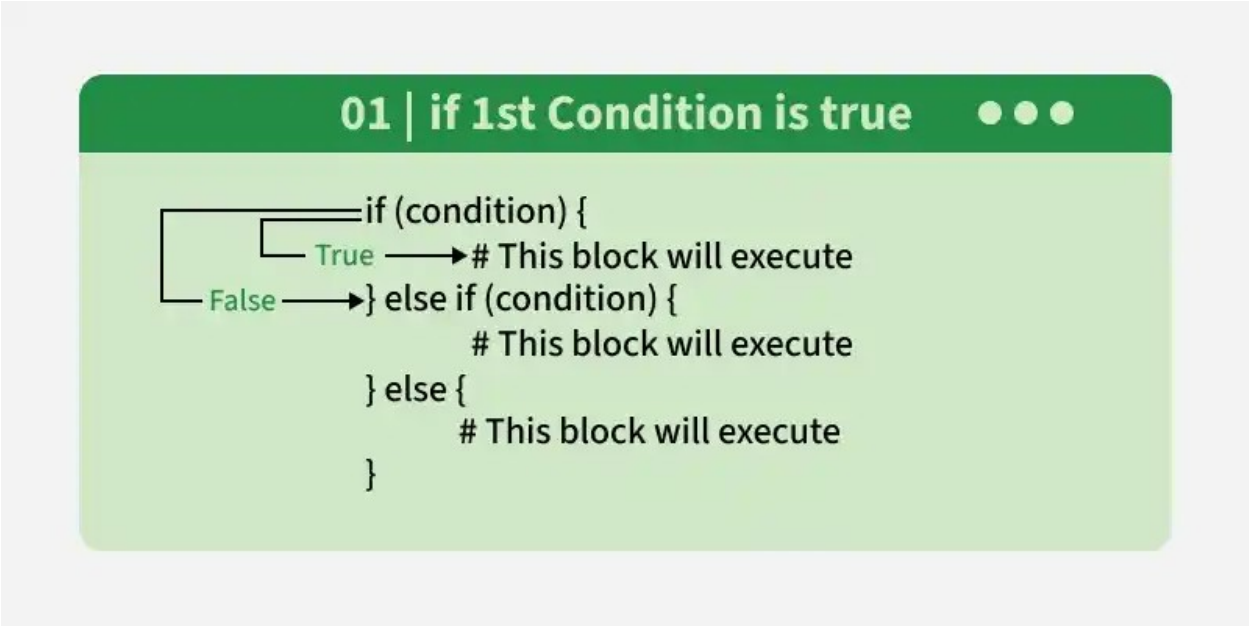
```
age = 20

if age <= 12:
    print("Travel for free.")
elif age <= 18:
    print("Discounted ticket price.")
elif age < 65:
    print("Standard ticket price.")
else:
    print("Senior discount.")
```

IZLAZ (OUTPUT):

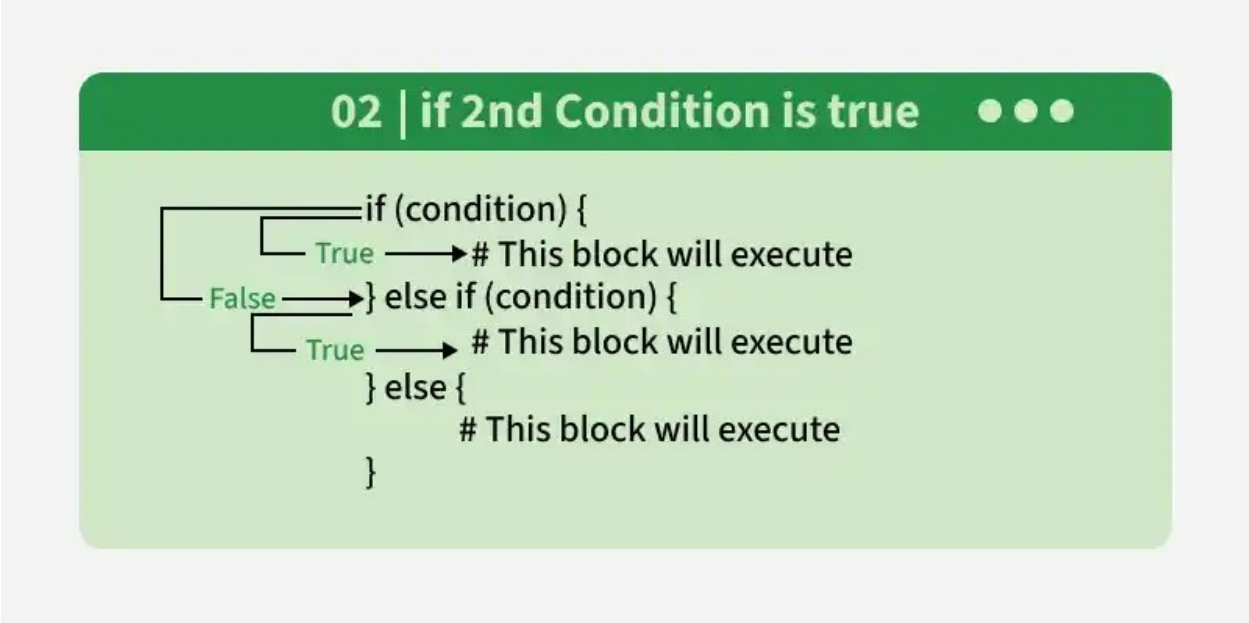
```
Standard ticket price.
```

Objašnjenje rada:

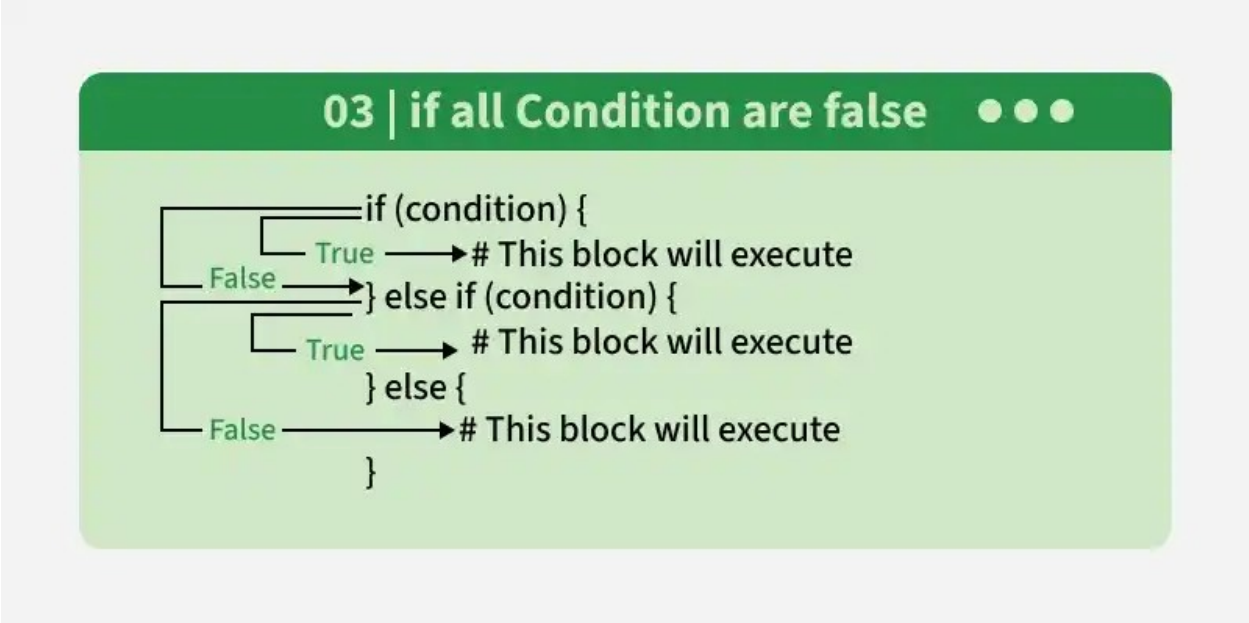
1. Prvi uslov `age <= 12` (10 <= 12) vraća `False` za `age = 20`.
2. Drugi uslov `age <= 18` (20 <= 18) takođe vraća `False`.
3. Treći uslov `age < 65` (20 < 65) vraća `True` — izvršava se njegov `print` blok.
4. Preostali `else` blok se automatski preskače.



Slika 11. *if-elif-else: izvršava se blok prvog tačnog uslova*



Slika 12. *if-elif-else: prvi uslov netačan, drugi tačan*



Slika 13. *if-elif-else: svi uslovi netačni, izvršava se else blok*

```
age = 25

if age <= 12:
    print("Child.")
elif age <= 19:
    print("Teenager.")
elif age <= 35:
    print("Young adult.")
else:
    print("Adult.")
```

IZLAZ (OUTPUT):

```
Young adult.
```

Objašnjenje: Pošto je `age = 25`, kod preskače prva dva uslova (`age <= 12` i `age <= 19`) jer su netačni, dok je treći uslov (`age <= 35`) tačan (`True`), pa se ispisuje `"Young adult."`.

Ugnježđena `if-else` naredba (Nested if-else Statement)

Ugnježđena `if-else` naredba predstavlja `if` ili `else` blok smešten unutar drugog `if` ili `else` bloka. Koristi se za proveru sekundarnih uslova tek nakon što je primarni uslov ispunjen.

Sintaksa

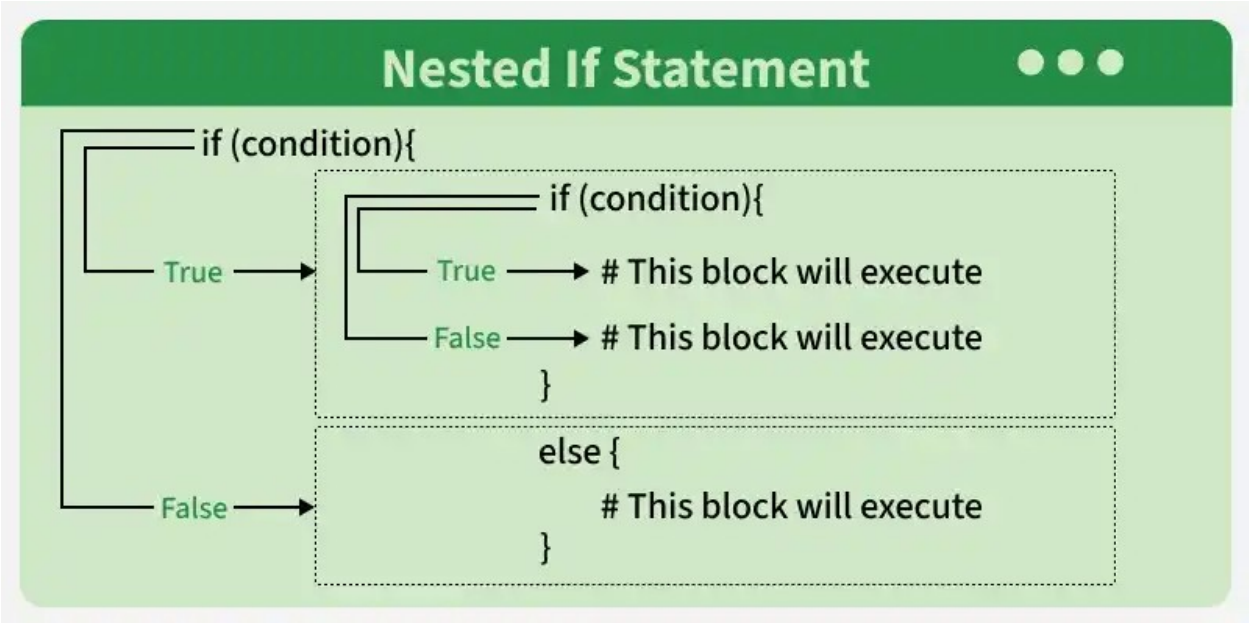
```
if primarni_uslov:
    # Izvršava se ako je primarni uslov True
    if sekundarni_uslov:
        # Izvršava se ako su I primarni I sekundarni uslov True
    else:
        # Izvršava se ako je primarni True, a sekundarni False
else:
    # Izvršava se ako je primarni uslov False
```

Primer upotrebe

```
age = 20
has_license = True

if age >= 18:
    if has_license:
        print("Možete voziti automobil.")
    else:
        print("Punoletni ste, ali nemate vozačku dozvolu.")
else:
    print("Maloletni ste i ne možete voziti.")
```

IZLAZ (OUTPUT):

```
Možete voziti automobil.
```



Slika 14. *Struktura ugnježđene if-else naredbe*

```
age = 70
is_member = True

if age >= 60:
    if is_member:
        print("30% senior discount!")
    else:
        print("20% senior discount.")
else:
    print("Not eligible for a senior discount.")
```

IZLAZ (OUTPUT):

```
30% senior discount!
```

Uslovni izraz / Ternarni operator (Conditional Expression)

Uslovni izraz (ternarni operator) predstavlja sažet način za pisanje `if-else` strukture u jednoj liniji koda. Koristi se kada na osnovu uslova treba izabrati i dodeliti jednu od dve vrednosti.

Sintaksa

```
vrednost_ako_je_tačno if uslov else vrednost_ako_je_netačno
```

Primer upotrebe

```
age = 20
s = "Adult" if age >= 18 else "Minor"
print(s)
```

IZLAZ (OUTPUT):

```
Adult
```

> **Objašnjenje:** Pošto je uslov `age >= 18` tačan (`True`), promenljivoj `s` se dodeljuje vrednost `"Adult"`. U suprotnom, bila bi dodeljena vrednost `"Minor"`.

Naredba `match-case` (Match-Case Statement)

Naredba `match-case` (uvedena u Python-u 3.10 kao *Structural Pattern Matching*) koristi se za poređenje vrednosti sa više različitih uzoraka (*patterns*) i izvršavanje

odgovarajućeg bloka koda. Slična je `switch-case` naredbi iz drugih programskih jezika (C++, Java, JavaScript).

Karakteristike:

- `case _:` Služi kao podrazumevani (*default*) slučaj (odgovara `else` bloku ili `default:` u drugim jezicima).
- **Vertikalna crta (`|`):** Omogućava spajanje više vrednosti u jedan `case` (OR logika).

Primer upotrebe

```
number = 2

match number:
    case 1:
        print("One")
    case 2 | 3:
        print("Two or Three")
    case _:
        print("Other number")
```

IZLAZ (OUTPUT):

```
Two or Three
```

POGLAVLJE 8

Petlje u Python-u (Loops in Python)

Petlje se koriste za višekratno izvršavanje bloka koda sve dok se ne ispuni određeni uslov ili dok se ne obrade svi elementi u sekvenci.

Glavni tipovi petlji u Python-u su:

- **`for` petlja:** Koristi se za iteraciju kroz sekvence (liste, torke, stringove, opsege).
- **`while` petlja:** Izvršava kod sve dok je zadati uslov tačan (`True`).

`for` petlja (For Loop)

`for` petlja se koristi za prolazak (iteraciju) kroz sekvencu kao što su lista, torka, string ili `range` objekat. Izvršava blok koda po jednom za svaki element u sekvenci.

Primer upotrebe

```
n = 4
for i in range(0, n):
    print(i)
```

IZLAZ (OUTPUT):

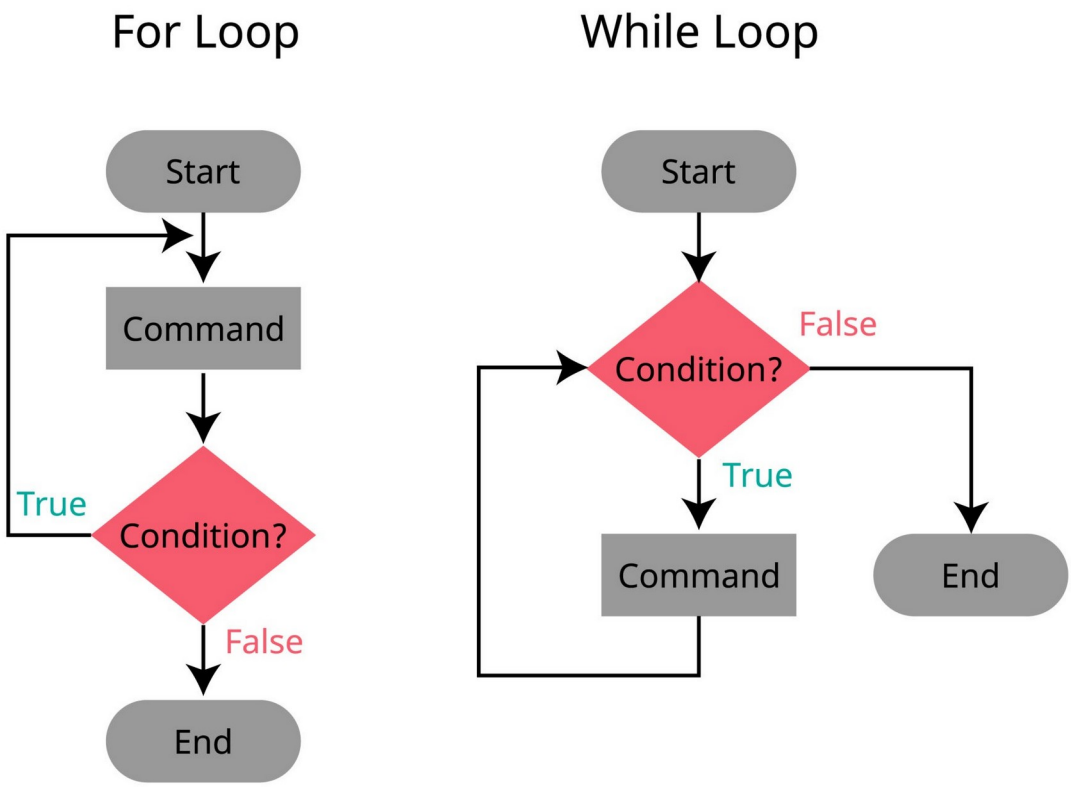
```
0
1
2
3
```

Objašnjenje:

- `n = 4` definisuje krajnju granicu za petlju.
- `range(0, n)` generiše niz brojeva od `0` do `3` (gornja granica `n` nije uključena).
- `for i in range(0, n):` prolazi kroz svaki broj iz generisanog opsega i dodeljuje ga promenljivoj `i`.

Dijagram toka `for` petlje (Flowchart)

Slika ispod prikazuje logički tok izvršavanja `for` i `while` petlji:

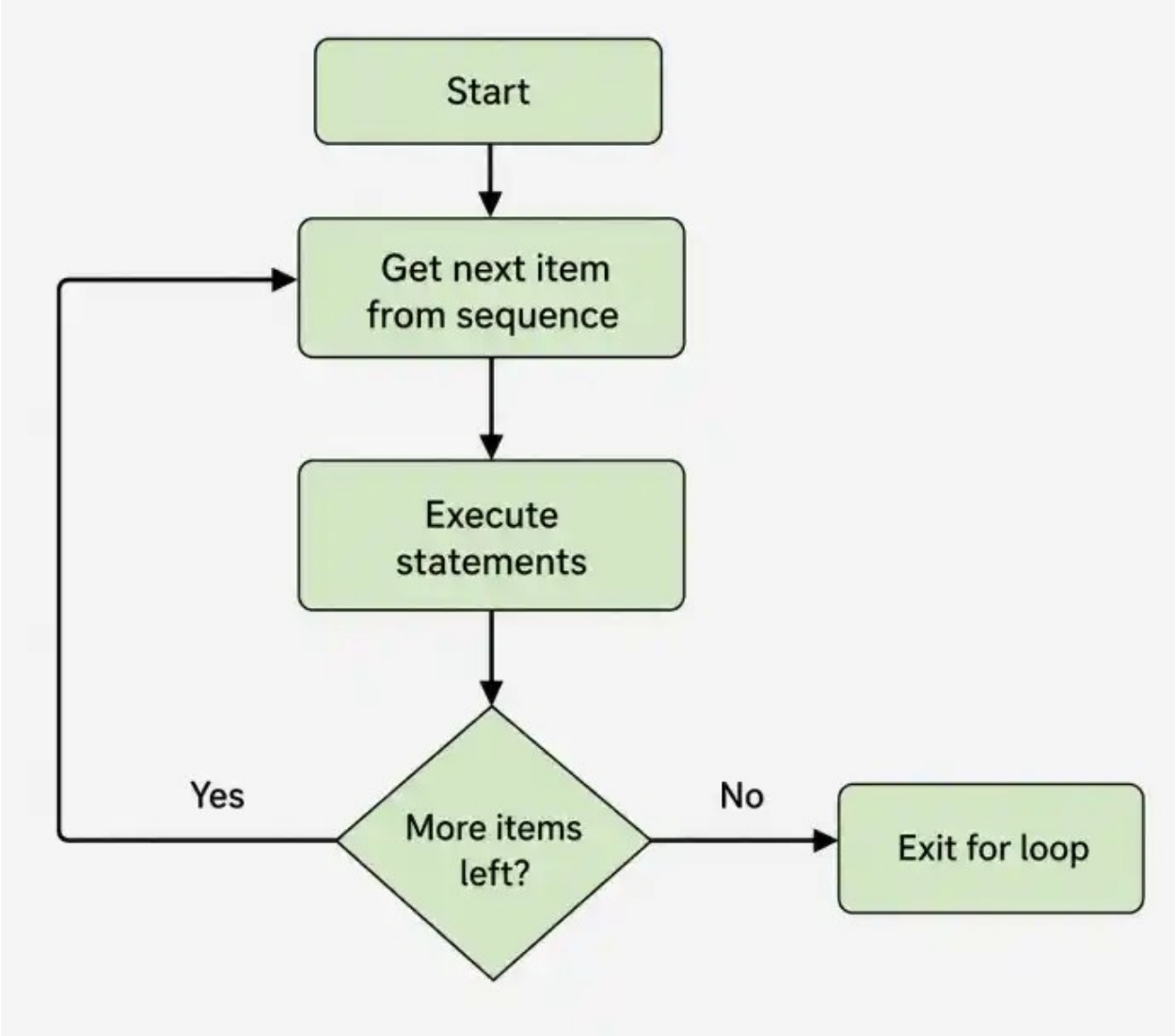


Slika 15. *Dijagram toka for i while petlje (izvor: Piscine / Getty Images)*

Proširenje: Rad sa `range()` funkcijom i sekvencama

Funkcija `range()` može primiti do tri argumenta: `range(start, stop, step)`.

```
# Iteracija kroz opseg sa korakom (step)
for i in range(1, 10, 2):
    print(i)  # Ispisuje neparne brojeve: 1, 3, 5, 7, 9

# Iteracija direktno kroz listu
voce = ["jabuka", "banana", "grožđe"]
for x in voce:
    print(x)
```



Slika 16. *Dijagram toka for petlje kroz sekvencu*

Iteracija pomoću indeksa sekvence (Iterating by Index)

`for` petlja može prolaziti kroz elemente sekvence i pomoću njihovih indeksnih vrednosti, uz pomoć funkcija `len()` i `range()`.

```
a = ["geeks", "for", "geeks"]
for idx in range(len(a)):
    print(a[idx])
```

IZLAZ (OUTPUT):

```
geeks
for
geeks
```

Objašnjenje:

- `len(a)` vraća ukupni broj elemenata u listi (u ovom slučaju `3`).
- `range(len(a))` generiše indekse od `0` do `2` (`0, 1, 2`).
- `a[idx]` pristupa elementu liste preko trenutnog indeksa.

> **Savet za Python:** Umesto ručnog korišćenja `range(len(a))`, u Python-u je elegantnije koristiti ugrađenu funkciju `enumerate()`, koja istovremeno vraća i indeks i sam element:

```
for idx, val in enumerate(a):
    print(idx, val)
```

`while` petlja (While Loop)

`while` petlja višekratno izvršava blok koda sve dok je zadati uslov **tačan (`True`)**. Kada uslov postane **netačan (`False`)**, petlja se prekida i program nastavlja sa izvršavanjem linija koda koje se nalaze odmah nakon petlje.

```
cnt = 0
while (cnt < 3):
    cnt = cnt + 1
    print("Hello Geek")
```

IZLAZ (OUTPUT):

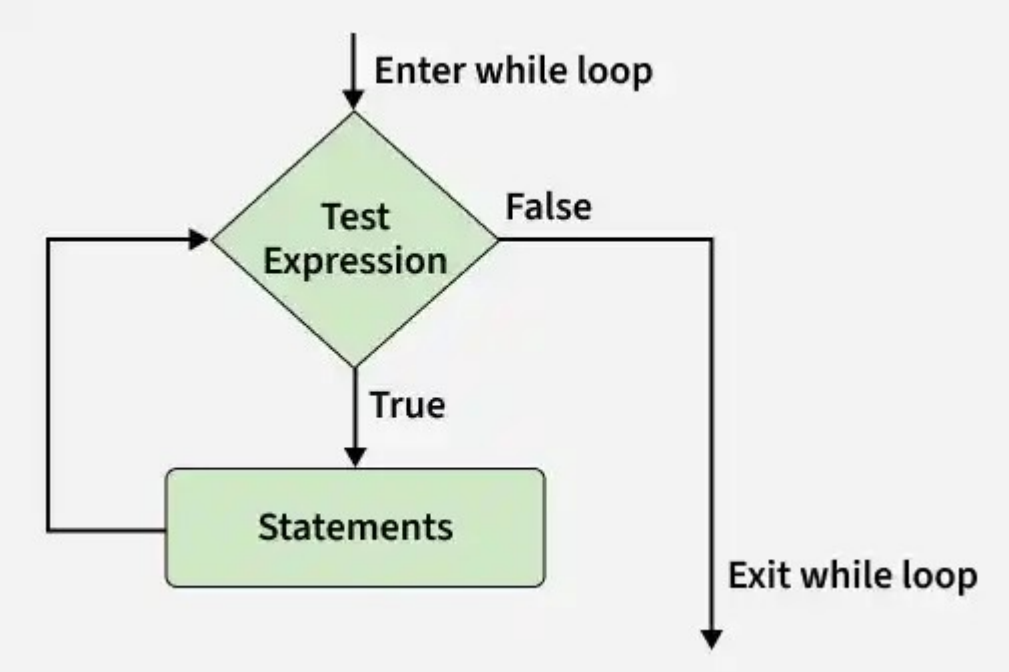
```
Hello Geek
Hello Geek
Hello Geek
```

Objašnjenje:

1. `cnt = 0` inicijalizuje brojačku promenljivu.
2. `while (cnt < 3):` proverava uslov. Petlja se izvršava dok god je `cnt` manji od `3`.
3. `cnt = cnt + 1` uvećava brojač za `1` u svakoj iteraciji (može se napisati i skraćeno kao `cnt += 1`).

Dijagram toka `while` petlje (Flowchart)

Logika rada `while` petlje svodi se na proveru uslova pre svakog ulaska u blok koda:



Slika 17. *Dijagram toka while petlje*

Beskonačna `while` petlja (Infinite While Loop)

Beskonačna `while` petlja neprekidno izvršava kod unutar sebe jer je njen uslov uvek tačan (`True`).

```
while (True):
    print("Hello Geek")
```

Objašnjenje:

- `while (True):` kreira petlju čiji je uslov uvek ispunjen.
- `print("Hello Geek")` se izvršava bez prestanka unutar petlje.

> **Napomena:** Ne preporučuje se korišćenje ovakvih petlji bez odgovarajućeg mehanizma za prekid (kao što je `break`), jer stvaraju nikad završavajući proces koji primorava korisnika da nasilno prekine rad programa.

Ugnježđene petlje (Nested Loops)

Ugnježđena petlja je petlja koja se nalazi unutar druge petlje. Unutrašnja petlja se **u potpunosti izvršava iznova za svaku pojedinačnu iteraciju spoljašnje petlje**.

```
for i in range(1, 5):
    for j in range(i):
        print(i, end=' ')
    print()
```

IZLAZ (OUTPUT):

```
1
2 2
3 3 3
4 4 4 4
```

Objašnjenje:

- **Spoljašnja petlja** `for i in range(1, 5)` kontroliše broj redova (od 1 do 4).
- **Unutrašnja petlja** `for j in range(i)` se izvršava `i` puta za svaki red.
- `print(i, end=' ')` ispisuje vrednost `i` u istom redu sa razmakom na kraju, dok prazan `print()` prebacuje ispis u sledeći red nakon što se unutrašnja petlja završi.

Razlika između `for` i `while` petlje

Postoji ključna razlika u njihovoj nameni:

- **`for` petlja** se koristi kada je broj iteracija unapred poznat ili kada se prolazi kroz definisanu sekvencu ili opseg.
- **`while` petlja** se koristi kada broj iteracija nije unapred poznat, već zavisi od ispunjenosti nekog dinamičkog uslova.

POGLAVLJE 9

Funkcije u Python-u (Python Functions)

Python funkcije su blokovi koda namenjeni ponovnoj upotrebi koji služe za izvršavanje određenog zadatka. Pomažu u organizaciji programa na manje, preglednije celine i omogućavaju izvršavanje iste logike kad god je to potrebno, jednostavnim pozivanjem funkcije.

Prednosti korišćenja funkcija:

- **Ponovna upotrebljivost koda (*Reusability*):** Napišete kod jednom i pozivate ga više puta sa različitim ulaznim podacima.
- **Modularnost i čitljivost:** Razbijanje složenog koda na manje, logičke celine lakše je za održavanje i razumevanje.
- **Manje grešaka:** Izmena logike vrši se na jednom mestu unutar funkcije, umesto na više mesta u programu.

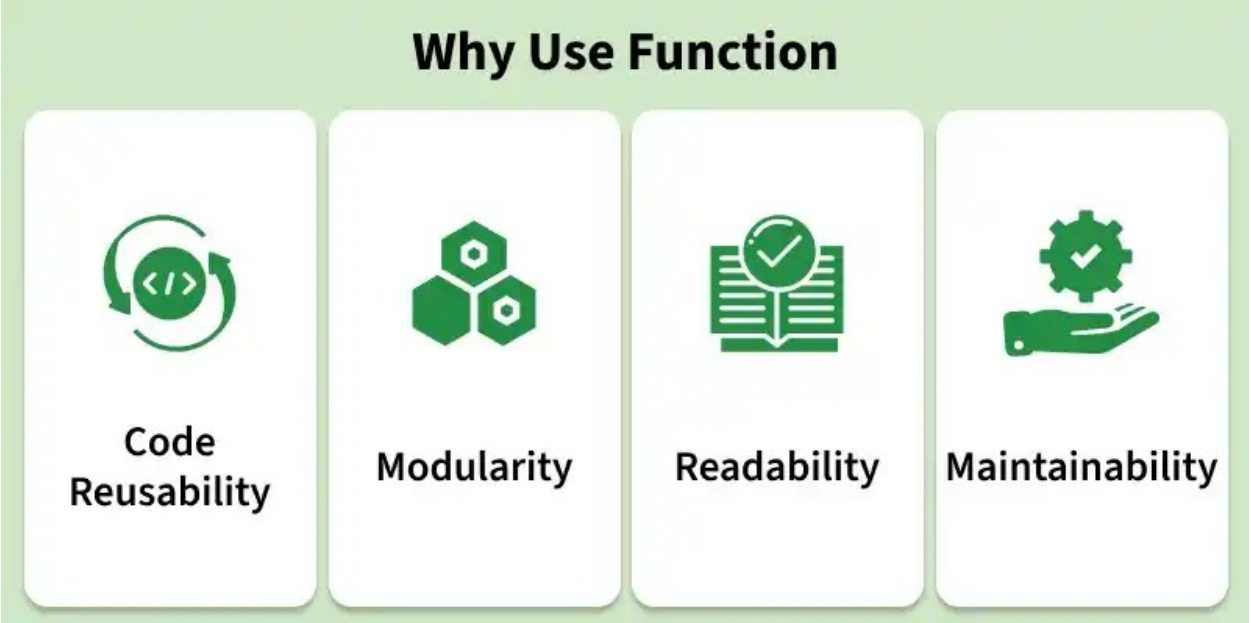
Osnovna sintaksa i primer

Funkcija se definiše pomoću ključne reči `def`, nakon čega sledi naziv funkcije, zagrade `()` sa opcionim parametrima i dvotačka `:`. Unutrašnji kod funkcije mora biti uvučen (*indented*).

```
# Definisanje funkcije
def pozdrav(ime):
    """Ova funkcija ispisuje pozdravnu poruku."""
    print(f"Zdravo, {ime}!")

# Pozivanje funkcije
pozdrav("Ana")
pozdrav("Marko")
```

IZLAZ (OUTPUT):

```
Zdravo, Ana!
Zdravo, Marko!
```



Slika 18. *Prednosti korišćenja funkcija*

Definisanje funkcije (Defining a Function)

Funkcija se u Python-u definiše pomoću ključne reči `def`. U nastavku je prikazana osnovna sintaksa za definisanje funkcije:

Sintaksa

```
def naziv_funkcije(parametri):
    # Telo funkcije (kod koji se izvršava)
    return vrednost
```

Struktura funkcije

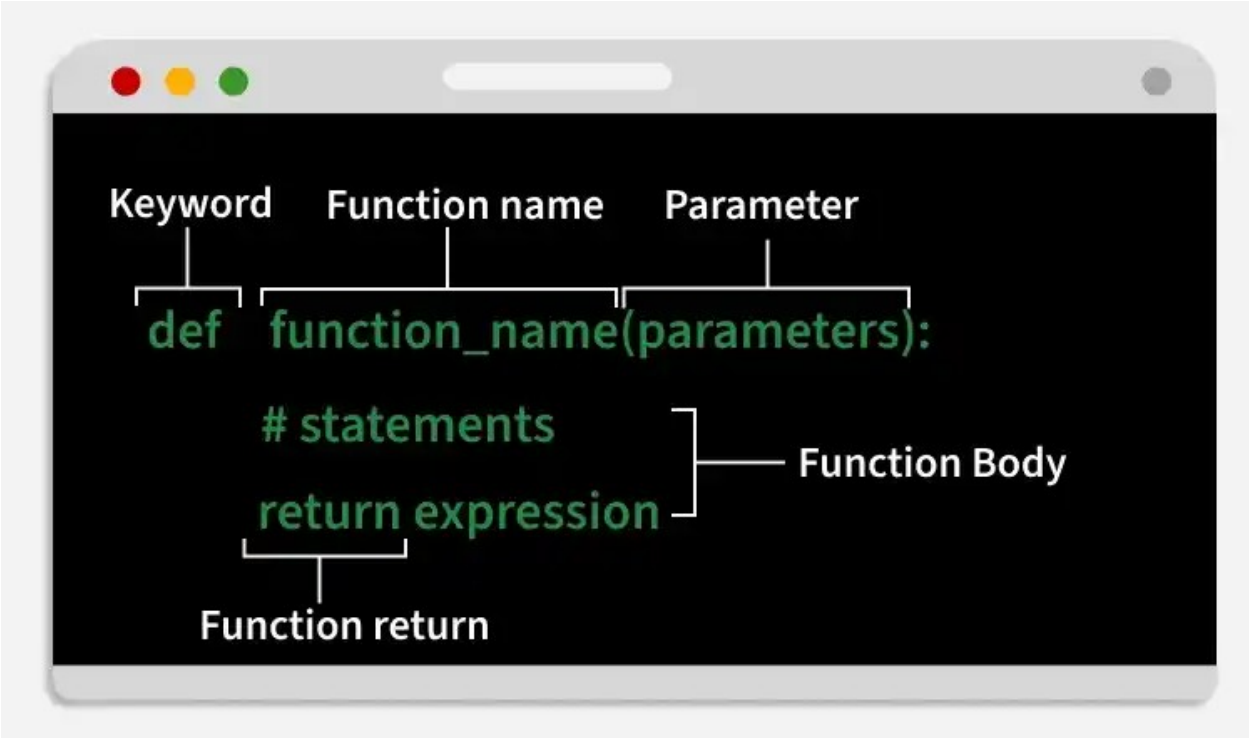
- **`def`:** Ključna reč koja označava početak definicije funkcije.
- **`naziv_funkcije`:** Jedinstveno ime funkcije po kom se ona poziva (sledi standardna pravila za imenovanje promenljivih).
- **`parametri` (opciono):** Ulazni podaci koje funkcija prima unutar zagrada `()`. Ako funkcija ne prima podatke, zagrade ostaju prazne.
- **Dvotačka (`:`):** Označava početak bloka koda funkcije.
- **Uvučeni kod (*Indented block*):** Telo funkcije u kome se nalazi logika. Uvlačenje (obično 4 razmaka) je obavezno u Python-u.
- **`return` (opciono):** Naredba koja vraća rezultat izvršavanja nazad pozivaocu. Ako se izostavi, funkcija podrazumevano vraća `None`.

Primer definisanja i pozivanja

```
def saberi(a, b):
    rezultat = a + b
    return rezultat

# Pozivanje funkcije
zbir = saberi(5, 3)
print(zbir)
```

IZLAZ (OUTPUT):

```
8
```



Slika 19. *Delovi definicije funkcije*

Definisanje i pozivanje funkcije (Defining and Calling a Function)

Funkcija se definiše pomoću ključne reči `def`. Nakon kreiranja, poziva se navođenjem njenog naziva sa zagradama `()`, unutar kojih se prosleđuju argumenti ukoliko ih funkcija zahteva.

```
def fun():
    print("Welcome to GFG")

fun()
```

IZLAZ (OUTPUT):

```
Welcome to GFG
```

> **Napomena:** U Python-u je funkcija objekat prve klase (*first-class object*), što znači da se funkcija može dodeliti promenljivoj i pozivati preko nje (npr. `g = fun; g()`).

Argumenti funkcije (Function Arguments)

Argumenti su vrednosti koje se prosleđuju funkciji prilikom poziva. Omogućavaju funkciji da primi ulazne podatke i obradi ih.

```
def evenOdd(x):
    if (x % 2 == 0):
        return "Even"
    else:
        return "Odd"

print(evenOdd(16))
print(evenOdd(7))
```

IZLAZ (OUTPUT):

```
Even
Odd
```

Tipovi argumenata u Python-u

Python podržava 4 osnovna tipa argumenata u zavisnosti od načina prosleđivanja:

1. Podrazumevani argumenti (Default Arguments)

Koriste unapred definisanu vrednost ako se tokom poziva funkcije ne prosledi odgovarajući argument.

```
def myFun(x, y=50):
    print("x: ", x)
    print("y: ", y)

myFun(10)
```

IZLAZ (OUTPUT):

```
x:  10
y:  50
```

Objašnjenje: Parametar `y` ima podrazumevanu vrednost `50`. Pošto pri pozivu `myFun(10)` prosleđujemo samo jedan argument, `y` poprima podrazumevanu vrednost `50`.

2. Imenovani argumenti (Keyword Arguments)

Vrednosti se prosleđuju uz eksplicitno navođenje imena parametara, pa redosled prosleđivanja nije bitan.

```
def student(fname, lname):
    print(fname, lname)

student(fname='Geeks', lname='Practice')
student(lname='Practice', fname='Geeks')
```

IZLAZ (OUTPUT):

```
Geeks Practice
Geeks Practice
```

3. Pozicioni argumenti (Positional Arguments)

Vrednosti se dodeljuju parametrima tačno po redosledu po kom su prosleđene.

```
def nameAge(name, age):
    print("Hi, I am", name)
    print("My age is ", age)

print("Case-1:")
nameAge("Olivia", 27)

print("Case-2:")
nameAge(27, "Olivia")
```

IZLAZ (OUTPUT):

```
Case-1:
Hi, I am Olivia
My age is  27
Case-2:
Hi, I am 27
My age is  Olivia
```

> **Objašnjenje:** U slučaju 1, vrednosti odgovaraju pravom redosledu parametara. U slučaju 2, zamena mesta argumenata dovodi do pogrešne dodele vrednosti po poziciji.

4. Proizvoljni argumenti (Arbitrary Arguments: *args i **kwargs)

Omogućavaju funkciji da prihvati proizvoljan broj argumenata:

- `*args` prikuplja dodatne **pozicione** argumente u vidu torke (*tuple*).
- `**kwargs` prikuplja dodatne **imenovane** argumente u vidu rečnika (*dictionary*).

```
def myFun(*args, **kwargs):
    print("Non-Keyword Arguments (*args):")
    for arg in args:
        print(arg)

    print("Keyword Arguments (**kwargs):")
    for key, value in kwargs.items():
        print(f"{key} == {value}")

myFun('Hey', 'Welcome', first='Geeks', mid='for', last='Geeks')
```

IZLAZ (OUTPUT):

```
Non-Keyword Arguments (*args):
Hey
Welcome
Keyword Arguments (**kwargs):
first == Geeks
mid == for
last == Geeks
```

Ugnježđene funkcije (Function within Functions)

Funkcija definisana unutar druge funkcije naziva se unutrašnja (*inner* ili *nested*) funkcija. Koristi se za organizaciju povezane logike i ima pristup promenljivim iz spoljašnje funkcije (*closures*).

```
def f1():
    s = 'I love GeeksforGeeks'
    def f2():
        print(s)

    f2()

f1()
```

IZLAZ (OUTPUT):

```
I love GeeksforGeeks
```

Naredba `return` (Return Statement)

Naredba `return` prekida izvršavanje funkcije i vraća rezultat pozivaocu. Može vratiti bilo koji tip podataka, pa i više vrednosti odjednom (spakovane u torku). Ako se `return` izostavi, funkcija podrazumevano vraća `None`.

```
def sq_value(num):
    return num**2

print(sq_value(2))
print(sq_value(-4))
```

IZLAZ (OUTPUT):

```
4
16
```

Prenos po referenci i prenos po vrednosti u Python-u (Pass by Object Reference)

Promenljive u Python-u zapravo predstavljaju reference na objekte. Ponašanje funkcije zavisi od toga da li je prosleđeni objekat **promenljiv (*mutable*)** ili **nepromenljiv (*immutable*)**:

- **Promenljivi objekti** (poput lista ili rečnika): izmene napravljene unutar funkcije utiču na originalni objekat izvan funkcije.
- **Nepromenljivi objekti** (poput celih brojeva, float-ova, stringova ili torki): izmena vrednosti unutar funkcije kreira novi objekat lokalno, pa originalni objekat van funkcije ostaje nepromenjen.

```
# Primer sa promenljivim objektom (lista)
def myFun(x):
    x[0] = 20

b = [10, 11, 12, 13]
myFun(b)
print(b)

# Primer sa nepromenljivim objektom (int)
def myFun2(x):
    x = 20

a = 10
myFun2(a)
print(a)
```

IZLAZ (OUTPUT):

```
[20, 11, 12, 13]
10
```

Zaključak: Python koristi mehanizam koji se naziva **"Pass-by-Object-Reference"** (ili *Pass-by-Assignment*), gde funkcija dobija referencu na postojeći objekat, ali mogućnost njegove modifikacije zavisi isključivo od mutabilnosti samog tipa podataka.

POGLAVLJE 10

Polja u Python-u (Arrays)

Polje (engl. *array*) predstavlja skup elemenata iste vrste kojima se može direktno pristupati korišćenjem numeričkog indeksa. Polja su osnovna struktura podataka: na njima se grade matrice, tabele, redovi, stekovi i mnoge druge strukture.

Ovo poglavlje prati prezentaciju *Polja* iz predmeta Strukture podataka. Primeri koji su u originalu napisani u jeziku C++ ovde su prevedeni u Python, a uz svaki primer objašnjeno je šta se dešava u memoriji i po čemu se Python razlikuje od C++-a.

Polja u Python-u

Python nema ugrađen tip koji se zove „polje“ u smislu jezika C, ali nudi tri strukture koje tu ulogu igraju:

- **list** je ugrađena lista. U pozadini je dinamičko polje referenci, pa elementi mogu biti različitog tipa, a lista može da raste i da se smanjuje.
- **array.array** iz standardne biblioteke čuva vrednosti jednog numeričkog tipa (npr. celih brojeva) kompaktno, jednu do druge, baš kao polje u C-u.
- **numpy.ndarray** iz spoljne biblioteke NumPy je višedimenzionalno polje namenjeno numeričkim proračunima.

```
from array import array

a = [3, 5, 2, 7]                # lista
b = array('i', [3, 5, 2, 7])    # polje celih brojeva ('i' = int)

print(a[2], b[2])
print(b.itemsize)               # veličina jednog elementa u bajtovima
```

IZLAZ (OUTPUT):

```
2 2
4
```

U ostatku poglavlja kao polje uglavnom koristimo listu, jer je to struktura sa kojom se u Python-u najčešće radi.

> **Napomena:** Lista ne čuva same vrednosti, već reference na objekte. Zato `[1, "dva", 3.0]` radi bez greške, dok bi `array('i', [1, "dva"])` izazvao grešku TypeError.

Moduli koji se koriste u primerima

Neki primeri koriste dodatne module. Moduli iz standardne biblioteke dolaze uz svaku instalaciju Python-a i dovoljno ih je uključiti, dok se spoljne biblioteke moraju prvo instalirati:

Modul	Odakle dolazi	Kako se uključuje	Instalacija
array	standardna biblioteka	`from array import array`	`nije potrebna`
abc	standardna biblioteka	`from abc import ABC, abstractmethod`	`nije potrebna`
numpy	spoljna biblioteka (PyPI)	`import numpy as np`	`pip install numpy`

- `import modul` uvozi ceo modul, pa se imena pišu sa prefiksom: `import array`, zatim `array.array('i', [1, 2])`.
- `from modul import ime` uvozi samo navedena imena, koja se zatim koriste direktno: `from array import array`, zatim `array('i', [1, 2])`.

> **Napomena:** Sve algoritme u ovom poglavlju (traženje, promenu veličine, kopiranje, sortiranje) pišemo sami, bez ugrađenih funkcija koje obavljaju isti posao. Cilj je razumeti kako algoritam radi, a ne samo pozvati gotovu funkciju.

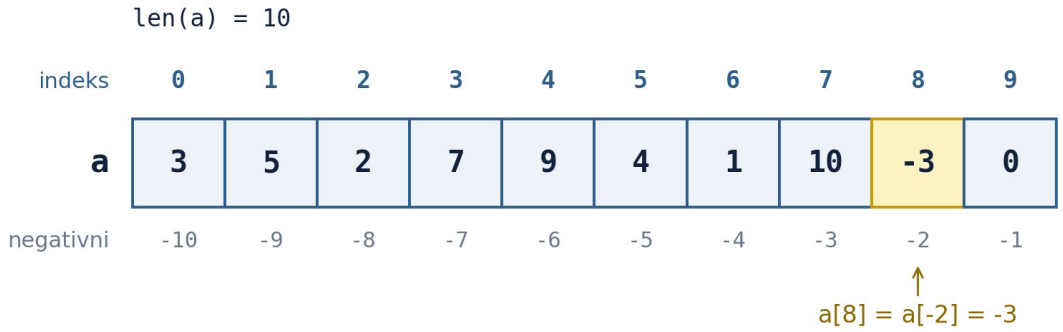
Tipovi polja

Polja se dele prema broju indeksa koji je potreban da bi se došlo do elementa:

- **Jednodimenzionalna polja – vektori.** Elementu se pristupa jednim indeksom: `a[i]`.
- **Dvodimenzionalna polja – matrice.** Potrebna su dva indeksa, vrsta i kolona: `M[i][j]`.
- **Višedimenzionalna polja.** Potrebno je N indeksa, npr. `A[i][j][k]`.

Jednodimenzionalno polje

Na slici je prikazano celobrojno polje `a` od deset elemenata. Iznad svake ćelije je njen indeks, a ispod negativni indeks, koji broji od kraja polja. Vrednost `a[8]` je -3.



Slika 20. *Jednodimenzionalno polje a sa pozitivnim i negativnim indeksima*

```
a = [3, 5, 2, 7, 9, 4, 1, 10, -3, 0]

print(a[0], a[8])     # pristup preko indeksa
print(a[-1], a[-2])   # negativni indeksi broje od kraja
print(len(a))         # broj elemenata
```

IZLAZ (OUTPUT):

```
3 -3
0 -3
10
```

Pokušaj pristupa indeksu koji ne postoji, na primer `a[10]`, izaziva grešku IndexError. Python, za razliku od C++-a, uvek proverava granice polja.

Pristup elementima

Elementi polja su u memoriji smešteni jedan do drugog. Zato se adresa bilo kog elementa može izračunati direktno, bez prolaska kroz prethodne elemente. Ako je:

- **α** adresa prvog elementa vektora A = (a[δ], a[δ+1], ..., a[n]),
- **σ** veličina jednog elementa,
- **δ** početna vrednost indeksa,

onda je adresa elementa sa indeksom i:

```
adresa(a[i]) = α + σ · (i − δ)
```

Kako je u Python-u (kao i u C/C++) δ = 0, formula se svodi na:

```
adresa(a[i]) = α + σ · i
```

> **Objašnjenje:** Ako polje počinje na adresi 1000, a svaki element zauzima 4 bajta, element `a[3]` nalazi se na adresi 1000 + 4 · 3 = 1012. Za ovaj račun potrebno je samo jedno množenje i jedno sabiranje, bez obzira na to koliko je polje veliko. Zato je pristup elementu polja složenosti **O(1)**.

Kod liste σ je veličina jedne reference (pokazivača), jer lista čuva reference. Kod `array.array` σ je veličina same vrednosti, što pokazuje atribut `itemsize`.

Rad sa poljima: C++ i Python

Sledeća tabela pokazuje kako se osnovne deklaracije iz C/C++ prevode u Python:

Operacija	C/C++	Python
Statička deklaracija	`int a[10];`	`a = [0] * 10`
Dinamička deklaracija	`int* a = new int[len];`	`a = [0] * n`
Dealokacija	`delete [] a;`	`del a`
Deklaracija sa vrednostima	`int a[10] = {1,2,…,0};`	`a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]`
Polje jednog tipa	`int a[10];`	`array('i', [0] * 10)`

```
n = 50
a = [0] * n          # polje od n nula
a.append(1)          # lista može da raste
print(len(a))

b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
del b                # uklanja ime b
```

IZLAZ (OUTPUT):

```
51
```

> **Napomena:** U Python-u nema ručne dealokacije memorije. Naredba `del` uklanja samo ime promenljive, a memoriju oslobađa sakupljač smeća (*garbage collector*) kada na objekat više ne pokazuje nijedna referenca.

Operacije nad poljem

Nad poljem se najčešće izvode sledeće operacije:

- **obilazak polja** (engl. *traversal*): pristupa se svakom elementu polja i nad njim vrši neka obrada (na primer, provera vrednosti ili ažuriranje),
- **traženje** (engl. *search*): traži se element zadate vrednosti,
- **umetanje** (engl. *insertion*): dodaje se novi element na zadatu lokaciju,
- **brisanje** (engl. *deletion*): iz polja se uklanja element zadate vrednosti ili sa zadate lokacije,
- **sortiranje** (engl. *sorting*): elementi se reorganizuju prema nekoj vrsti uređenosti (kod numeričkih polja to je obično rastući ili opadajući redosled, kod alfanumeričkih abecedni ili leksikografski).

```
a = [3, 5, 2, 7, 9]

for x in a:                  # obilazak
    print(x * 2, end=" ")
print()

print(7 in a, a.index(7))    # traženje

a.insert(2, 99)              # umetanje na poziciju 2
print(a)

a.remove(99)                 # brisanje po vrednosti
x = a.pop(0)                 # brisanje po poziciji
print(x, a)
```

IZLAZ (OUTPUT):

```
6 10 4 14 18
True 3
[3, 5, 99, 2, 7, 9]
3 [5, 2, 7, 9]
```

Traženje i sortiranje ne radimo ugrađenim funkcijama, već ih implementiramo sami u nastavku poglavlja (binarno traženje i četiri algoritma sortiranja).

Složenost ovih operacija nad listom od n elemenata:

Operacija	Python	Složenost
Pristup elementu	`a[i]`	O(1)
Traženje (redom)	`x in a`	O(n)

Operacija	Python	Složenost
Binarno traženje	`binary_search(a, x, 0, n - 1)`	O(log n)
Umetanje na kraj	`a.append(x)`	O(1) u proseku
Umetanje u sredinu	`a.insert(i, x)`	O(n)
Brisanje	`a.remove(x)`, `a.pop(i)`	O(n), pop() sa kraja O(1)
Sortiranje	`bubble_sort(a)` … `merge_sort(…)`	O(n²) do O(n log n)

> **Objašnjenje:** Umetanje i brisanje u sredini polja su skupi jer se svi elementi iza zadate pozicije moraju pomeriti za jedno mesto.

Dijagrami toka

Algoritmi u ovom poglavlju prikazani su i **dijagramima toka** (engl. *flowchart*). Dijagram toka crta korake algoritma pomoću standardnih simbola povezanih strelicama koje pokazuju redosled izvršavanja:



Slika 21. *Osnovni simboli dijagrama toka*

- **Početak / kraj** (zaobljeni pravougaonik): na početku se navodi naziv funkcije sa parametrima.
- **Obrada** (pravougaonik): naredba dodele vrednosti ili poziv funkcije.
- **Uslov** (šestougao): iz njega izlaze dve grane, *da* i *ne*. Petlja se crta kao uslov čija jedna grana vodi nazad, ispred uslova.
- **Ulaz / izlaz** (paralelogram): vraćanje rezultata ili podizanje izuzetka.
- **Spajanje** (mala tačka): mesto gde se dve grane ponovo spajaju.

Kada se piše kôd na osnovu dijagrama, svaki uslov postaje `if` ili `while`, a svaka grana nazad postaje petlja.

Binarno traženje

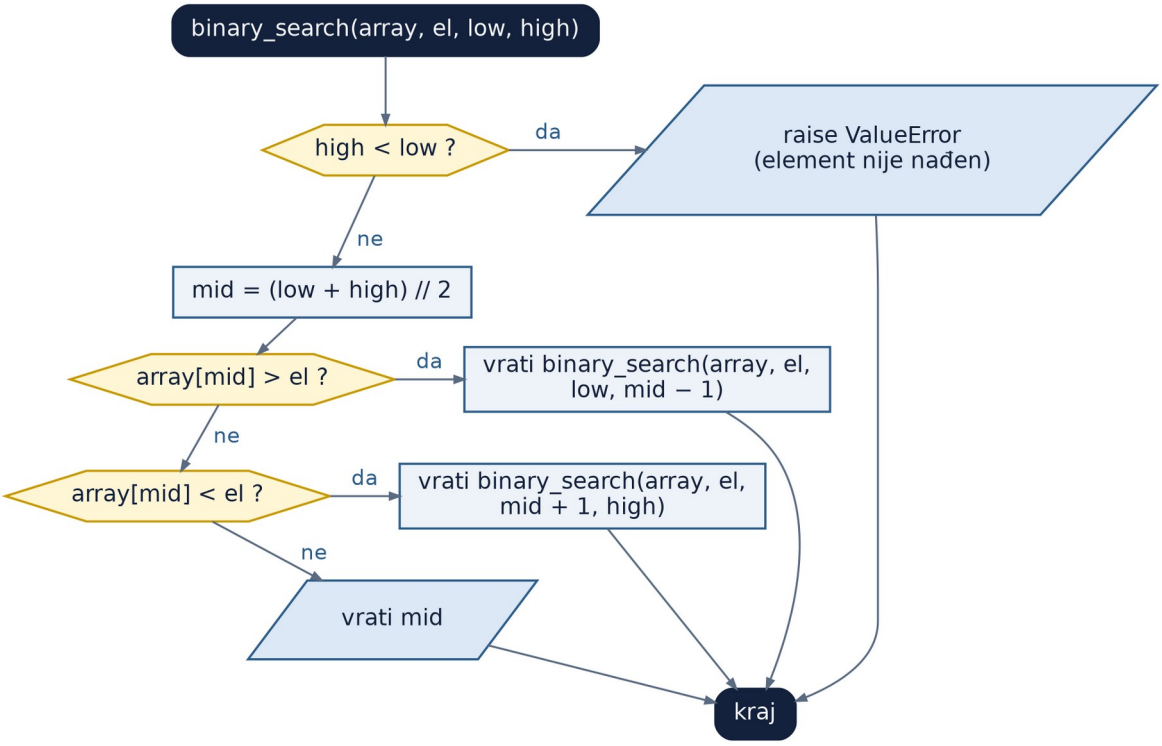
Kada je polje **sortirano**, element se može naći mnogo brže nego proverom redom. Binarno traženje poredi traženu vrednost sa srednjim elementom opsega. Ako je tražena vrednost manja, pretraga se nastavlja u levoj polovini, a ako je veća, u desnoj. Tako se opseg u svakom koraku prepolovi.

Na slici se u polju od 16 elemenata traži vrednost 76. Do rezultata se stiže u četiri koraka.



Slika 22. *Koraci binarnog traženja vrednosti 76 u sortiranom polju*

Dijagram toka



Slika 23. *Dijagram toka binarnog traženja*

Rekurzivna implementacija

```
def binary_search(array, el, low, high):
    """U listi array traži element el u opsegu indeksa od low do high."""
    if high < low:
        raise ValueError("Element nije nađen")
    mid = (low + high) // 2
    if array[mid] > el:
        return binary_search(array, el, low, mid - 1)
    elif array[mid] < el:
        return binary_search(array, el, mid + 1, high)
    else:
        return mid

data = [2, 9, 11, 15, 28, 33, 40, 47, 51, 64, 76, 78, 82, 85, 91, 99]
print(binary_search(data, 76, 0, len(data) - 1))
```

IZLAZ (OUTPUT):

```
10
```

> **Objašnjenje:** Ako je `high < low`, opseg je prazan i element ne postoji, pa funkcija podiže izuzetak ValueError (u C++ verziji to je bio SBPException). Operator `//` je celobrojno deljenje, pa je `mid` uvek ceo broj. Pošto se opseg stalno polovi, broj koraka je najviše oko $\log_2 n$, pa je složenost **O(log n)**.

> **Napomena:** Binarno traženje radi ispravno samo nad sortiranim poljem. Nad nesortiranim poljem rezultat nema smisla.

Dinamička polja

Dinamička polja su polja koja mogu menjati veličinu tokom izvršavanja programa. U prezentaciji je opisana klasa koja pored podataka čuva i **osnovu** (engl. *base*), odnosno vrednost prvog dozvoljenog indeksa. Tako se mogu napraviti polja čiji indeksi počinju, na primer, od -1 ili od 1.

Python lista je već dinamičko polje, ali ćemo klasu napisati sami, da bi se videlo kako takva struktura radi iznutra.

Klasa Array

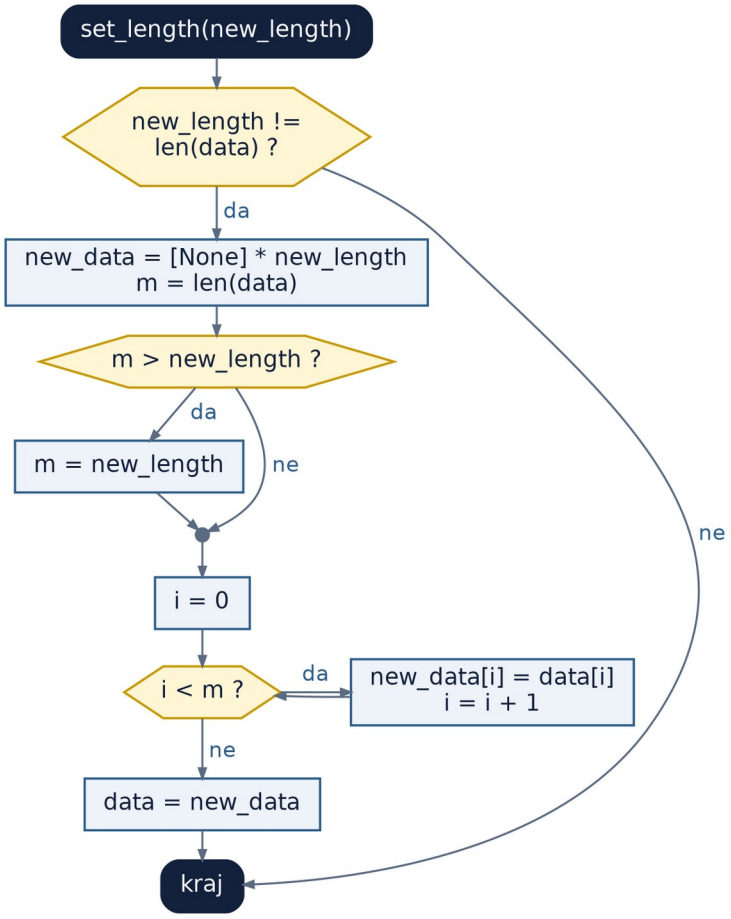
```
class Array:
    """Jednodimenzionalno polje sa proizvoljnom osnovom indeksa."""

    def __init__(self, length, base=0):
        self.base = base             # osnova (prvi indeks)
        self.data = [None] * length  # vektor podataka

    def __len__(self):
        return len(self.data)

    def _offset(self, index):
        offset = index - self.base
        if not 0 <= offset < len(self.data):
            raise IndexError("Indeks van granica!")
        return offset

    def __getitem__(self, index):
        return self.data[self._offset(index)]

    def __setitem__(self, index, value):
        self.data[self._offset(index)] = value

    def print(self):
        for i in range(len(self.data)):
            print(f"data[{i + self.base}]={self.data[i]}")
```

Klasa koristi specijalne metode Python-a, koje zamenjuju preopterećenje operatora iz C++-a:

- `__len__` omogućava da se dužina polja dobije funkcijom `len(p)`,
- `__getitem__` se poziva pri čitanju `p[i]` (u C++ to su bili `operator []` i `getAt`),
- `__setitem__` se poziva pri upisu `p[i] = x` (u C++ `setAt`),
- `_offset` pretvara spoljašnji indeks u poziciju u listi oduzimanjem osnove i proverava granice.

Promena veličine polja

Metoda `set_length` pravi novo polje zadate dužine, kopira u njega onoliko elemenata koliko staje i zamenjuje staro polje novim. Dijagram toka:



Slika 24. *Dijagram toka metode set_length (promena veličine polja)*

```
def set_length(self, new_length):
    if new_length != len(self.data):
        new_data = [None] * new_length
        m = len(self.data)
        if m > new_length:      # m je manja od dve dužine
            m = new_length
        for i in range(m):
            new_data[i] = self.data[i]
        self.data = new_data

Array.set_length = set_length
```

1. Ako je nova dužina ista kao stara, ništa se ne radi.
2. Pravi se novo polje `new_data` tražene dužine.
3. Za m se uzima manja od dve dužine: pri povećanju kopiraju se svi elementi, pri smanjenju samo prvih `new_length`.
4. Staro polje se zamenjuje novim; staro briše sakupljač smeća, pa nije potrebno `delete [] data`.

Kopiranje sadržaja polja

U C++ verziji kopiranje je rađeno operatorom dodele `=`. U Python-u operator dodele samo pravi novu referencu na isti objekat, pa kopiranje pišemo kao posebnu metodu. Kao u C++ verziji, ako se dužine razlikuju, prvo se pravi novo polje odgovarajuće dužine, a zatim se elementi kopiraju jedan po jedan:

```
def copy_from(self, other):
    if other is not self:          # ne kopira se samo u sebe
        if len(self.data) != len(other.data):
            self.data = [None] * len(other.data)
        for i in range(len(other.data)):
            self.data[i] = other.data[i]
        self.base = other.base

Array.copy_from = copy_from
```

Primer korišćenja

```
p = Array(3, base=-1)
p[-1] = 3
p[0] = 5
p[1] = 2
p.print()

p.set_length(5)
print(len(p), p.data)

q = Array(1)
q.copy_from(p)
print(q.base, q.data)

try:
    p[10] = 1
except IndexError as e:
    print("Greška:", e)
```

IZLAZ (OUTPUT):

```
data[-1]=3
data[0]=5
data[1]=2
5 [3, 5, 2, None, None]
-1 [3, 5, 2, None, None]
Greška: Indeks van granica!
```

Objašnjenje: Polje `p` ima osnovu -1, pa su dozvoljeni indeksi -1, 0 i 1. Posle promene dužine na 5 dodata su dva prazna mesta. Upis na indeks 10 izlazi van granica, pa se podiže IndexError, koji se hvata blokom `try/except` (u C++ `try/catch`).

Napomena: Ugrađena lista se sama proširuje: kada se popuni, Python rezerviše malo više mesta nego što je trenutno potrebno. Zato je `append` u proseku O(1), iako povremeno mora da prepiše celo polje na novo mesto.

Višedimenzionalna polja

Višedimenzionalno polje dimenzije N je kolekcija podataka kojima se pristupa preko N indeksnih izraza (npr. `A[i][j]`). U memoriji se ono ipak smešta kao jednodimenzionalno polje, red po red (engl. *row-major*), pa je potrebno N indeksa pretvoriti u jedan pomeraj:

$$\text{adresa}(a[i_0, i_1, \ldots, i_{n-1}]) = \alpha + \sigma \cdot \Sigma\ \text{faktori}[k] \cdot i_k$$

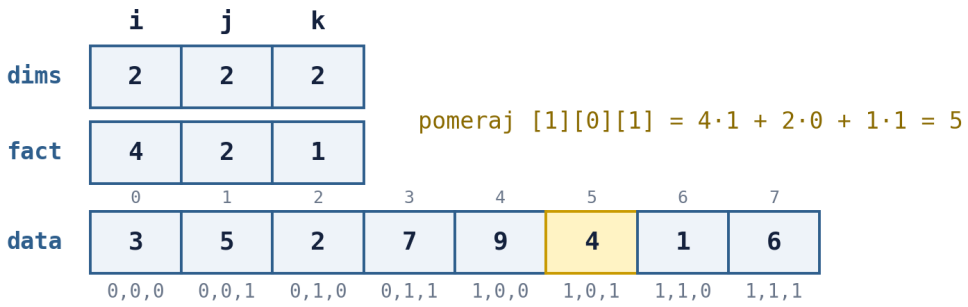
gde su faktori:

- `faktori[k] = 1`, za k = N − 1 (poslednja dimenzija),
- `faktori[k] = dim[k+1] · dim[k+2] · … · dim[N−1]`, za k < N − 1.

> **Objašnjenje:** Faktor dimenzije govori koliko elemenata treba preskočiti kada se taj indeks poveća za jedan. Za polje dimenzija 2 × 3 × 4 faktori su [12, 4, 1], pa je element [1, 2, 3] na poziciji 12 · 1 + 4 · 2 + 1 · 3 = 23.

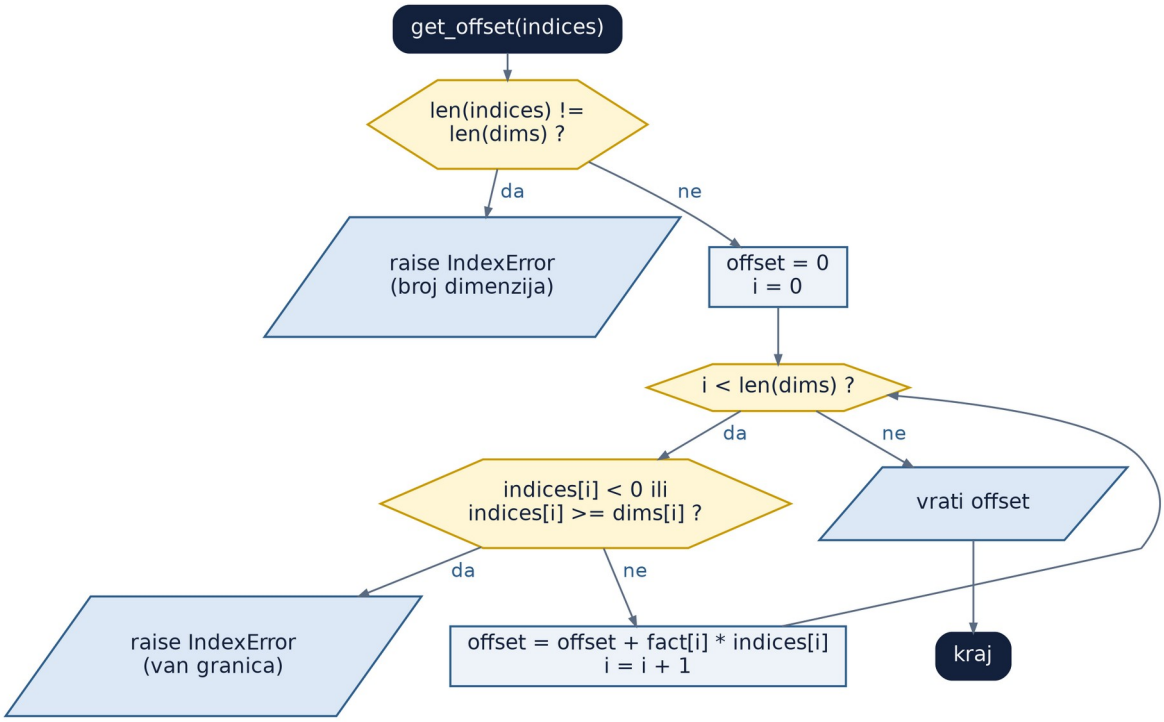
Memorijska reprezentacija

Slika prikazuje polje dimenzija 2 × 2 × 2. Pored samih podataka čuvaju se dimenzije i unapred izračunati faktori.



Slika 25. *Memorijska reprezentacija trodimenzionalnog polja 2 × 2 × 2*

Dijagram toka određivanja pomeraja



Slika 26. *Dijagram toka metode get_offset (određivanje pomeraja)*

Klasa MultiDimensionalArray

```
class MultiDimensionalArray:
    def __init__(self, *dims):
        self.dims = dims                    # dimenzije (torka)
        self.fact = [0] * len(dims)         # faktori
        product = 1
        for i in range(len(dims) - 1, -1, -1):
            self.fact[i] = product
            product *= dims[i]
        self.data = [None] * product        # vektor podataka

    def get_offset(self, indices):
        if len(indices) != len(self.dims):
            raise IndexError("Broj dimenzija je netačan!")
        offset = 0
        for i in range(len(self.dims)):
            if indices[i] < 0 or indices[i] >= self.dims[i]:
                raise IndexError("Indeks je van granica!")
            offset += self.fact[i] * indices[i]
        return offset

    def __getitem__(self, indices):
        return self.data[self.get_offset(indices)]

    def __setitem__(self, indices, value):
        self.data[self.get_offset(indices)] = value

m = MultiDimensionalArray(2, 3, 4)
print(m.fact, len(m.data))
m[1, 2, 3] = 12
print(m.get_offset((1, 2, 3)), m[1, 2, 3])
```

IZLAZ (OUTPUT):

```
[12, 4, 1] 24
23 12
```

Objašnjenje: Konstruktor prolazi kroz dimenzije od poslednje ka prvoj i usput množi proizvod, pa svaka dimenzija dobija proizvod svih dimenzija desno od sebe. Taj proizvod je na kraju i ukupan broj elemenata ($2 \cdot 3 \cdot 4 = 24$). Metoda `get_offset` proverava broj indeksa i njihove granice, pa sabira proizvode faktora i indeksa. Kada se piše `m[1, 2, 3]`, Python prosleđuje indekse kao torku `(1, 2, 3)`.

Liste listi

U svakodnevnom radu matrica se u Python-u najčešće pravi kao lista listi, gde je svaka vrsta posebna lista:

```
M = [[0] * 4 for _ in range(3)]   # 3 vrste, 4 kolone
M[1][2] = 7
print(M)
```

IZLAZ (OUTPUT):

```
[[0, 0, 0, 0], [0, 0, 7, 0], [0, 0, 0, 0]]
```

Česta greška je da se matrica napravi množenjem liste:

```
bad = [[0] * 3] * 2
bad[0][0] = 1
print(bad)
```

IZLAZ (OUTPUT):

```
[[1, 0, 0], [1, 0, 0]]
```

> **Napomena:** Izraz `[[0] * 3] * 2` ne pravi dve vrste, već dve reference na **istu** listu. Izmena jedne „vrste“ zato menja obe. Svaka vrsta mora da se napravi posebno, kao u prvom primeru.

Matrice

Matrice su dvodimenzionalna polja realnih brojeva. Prvi indeks označava vrstu, a drugi kolonu. Matrice se dele:

- po obliku na **kvadratne** (engl. *square matrix*), sa istim brojem vrsta i kolona, i **pravougaone** (engl. *rectangular matrix*),
- po popunjenosti na **retko posednute** (engl. *sparse matrix*), kod kojih je većina elemenata nula, i **guste** (engl. *dense*).

Specijalne matrice

Kod **gornje trougaone** matrice svi elementi ispod glavne dijagonale su nule, a kod **donje trougaone** svi elementi iznad nje. Kvadratna trougaona matrica reda n ima najviše

```
1 + 2 + … + n = n · (n + 1) / 2
```

elemenata različitih od nule, pa je dovoljno sačuvati samo njih. Matrica je **simetrična** ako je a[i][j] = a[j][i]; i tada je dovoljno memorisati samo elemente donje ili gornje trougaone matrice.

Ako se elementi donje trougaone matrice smeštaju red po red u jednu listu, element (i, j), za j ≤ i, nalazi se na poziciji:

```
def lower_index(i, j):
    """Pozicija elementa (i, j) donje trougaone matrice, j <= i."""
    return i * (i + 1) // 2 + j

print([lower_index(i, j) for i in range(4) for j in range(i + 1)])
```

IZLAZ (OUTPUT):

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Objašnjenje: Pre vrste i nalazi se 1 + 2 + … + i = i(i + 1)/2 elemenata, a u samoj vrsti element je j-ti. Za matricu 4 × 4 čuva se 10 umesto 16 elemenata.

Apstraktna klasa matrice

U C++ verziji zajednički interfejs svih matrica opisan je klasom sa virtuelnim funkcijama. U Python-u se za to koristi modul `abc` (*abstract base classes*):

```
from abc import ABC, abstractmethod

class Matrix(ABC):
    @abstractmethod
    def get(self, i, j): ...          # čitanje vrednosti

    @abstractmethod
    def put(self, i, j, value): ...   # dodela vrednosti

class DenseMatrix(Matrix):
    def __init__(self, rows, cols):
        self.rows = rows              # broj vrsta
        self.cols = cols              # broj kolona
        self.data = [0.0] * (rows * cols)

    def get(self, i, j):
        return self.data[i * self.cols + j]

    def put(self, i, j, value):
        self.data[i * self.cols + j] = value

    def transpose(self):
        t = DenseMatrix(self.cols, self.rows)
        for i in range(self.rows):
            for j in range(self.cols):
                t.put(j, i, self.get(i, j))
        return t

d = DenseMatrix(2, 3)
d.put(0, 2, 5.0)
print(d.get(0, 2), d.transpose().get(2, 0))
```

IZLAZ (OUTPUT):

```
5.0 5.0
```

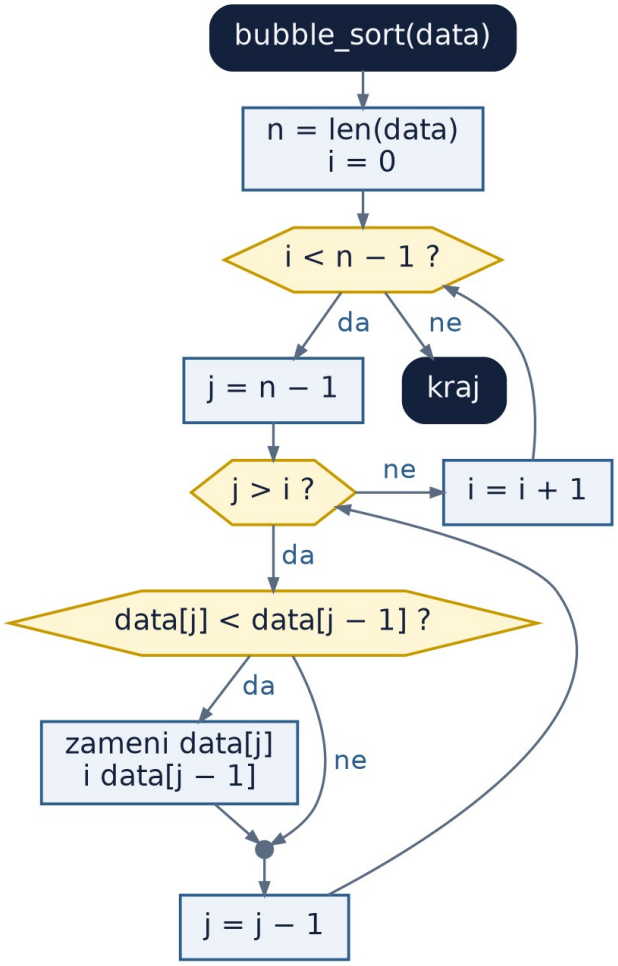
Objašnjenje: Gusta matrica čuva sve elemente u jednoj listi dužine `rows * cols`, a pozicija elementa (i, j) je `i * cols + j`, što je formula za višedimenzionalno polje sa faktorima [cols, 1]. Pokušaj da se napravi objekat `Matrix()` izaziva TypeError, jer apstraktna klasa ne može da se instancira. Operacije množenja (`times`) i sabiranja (`plus`) dodaju se na isti način kao `transpose`.

Primena polja: sortiranje

Sortiranje je jedna od najčešćih operacija nad poljima. Obradićemo četiri klasična algoritma. Za svaki je dat opis, dijagram toka i implementacija, a svi primeri sortiraju isto polje `[3, 5, 2, 7, 9, 4, 1, 10, -3, 0]`.

Bubble sort (metod mehurova)

Upoređuju se dva po dva susedna elementa i manji ide napred, tako da u prvom prolazu najmanji element „isplivava“ na prvu poziciju, u drugom drugi najmanji na drugu, itd.



Slika 27. *Dijagram toka bubble sort-a*

```
def bubble_sort(data):
    n = len(data)
    for i in range(n - 1):
        for j in range(n - 1, i, -1):
            if data[j] < data[j - 1]:
                data[j], data[j - 1] = data[j - 1], data[j]

a = [3, 5, 2, 7, 9, 4, 1, 10, -3, 0]
bubble_sort(a)
print(a)
```

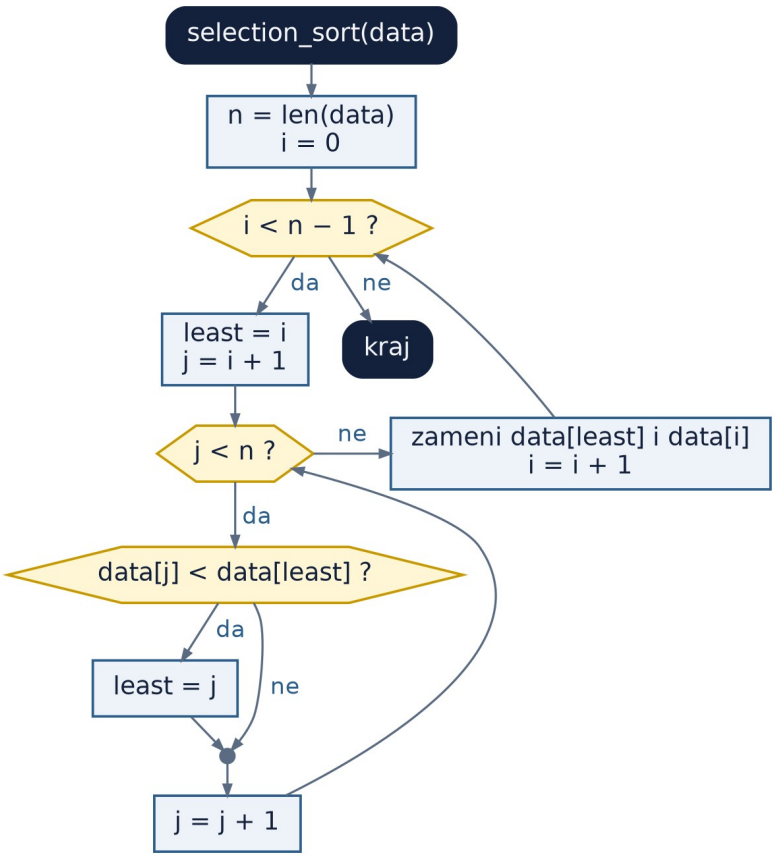
IZLAZ (OUTPUT):

```
[-3, 0, 1, 2, 3, 4, 5, 7, 9, 10]
```

Složenost: **O(n²)**. Zamena `a, b = b, a` zamenjuje pomoćnu funkciju `swap` iz C++-a.

Selection sort (metod selekcije)

Traži se indeks najmanjeg elementa u celom vektoru i on menja mesto sa prvim elementom. Nakon toga traži se najmanji u preostalom delu vektora i stavlja na drugo mesto, itd.



Slika 28. *Dijagram toka selection sort-a*

```
def selection_sort(data):
    n = len(data)
    for i in range(n - 1):
        least = i
        for j in range(i + 1, n):
            if data[j] < data[least]:
                least = j
        data[least], data[i] = data[i], data[least]

a = [3, 5, 2, 7, 9, 4, 1, 10, -3, 0]
selection_sort(a)
print(a)
```

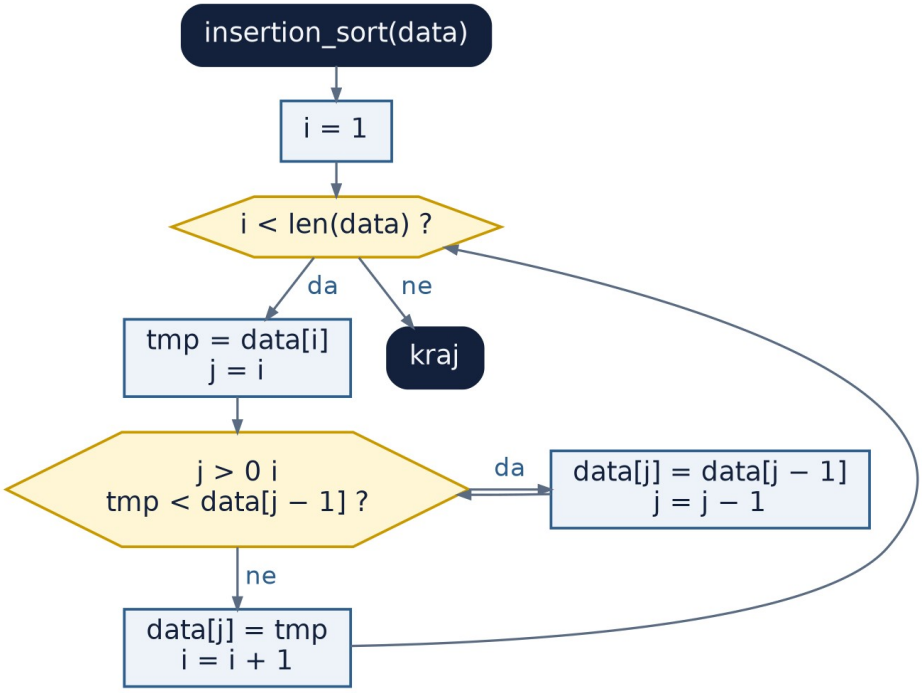
IZLAZ (OUTPUT):

```
[-3, 0, 1, 2, 3, 4, 5, 7, 9, 10]
```

Složenost: **$O(n^2)$**, uvek $n(n-1)/2$ poređenja, ali najviše $n-1$ zamena.

Insertion sort (metod umetanja)

Polazeći od i-tog elementa vektora, pomeraju se svi elementi levo od njega koji su veći od njega za po jedno mesto udesno, da bi se napravilo mesto za i-ti element. Levo od pozicije i tako je uvek sortiran deo polja.



Slika 29. *Dijagram toka insertion sort-a*

```
def insertion_sort(data):
    for i in range(1, len(data)):
        tmp = data[i]
        j = i
        while j > 0 and tmp < data[j - 1]:
            data[j] = data[j - 1]
            j -= 1
        data[j] = tmp

a = [3, 5, 2, 7, 9, 4, 1, 10, -3, 0]
insertion_sort(a)
print(a)
```

IZLAZ (OUTPUT):

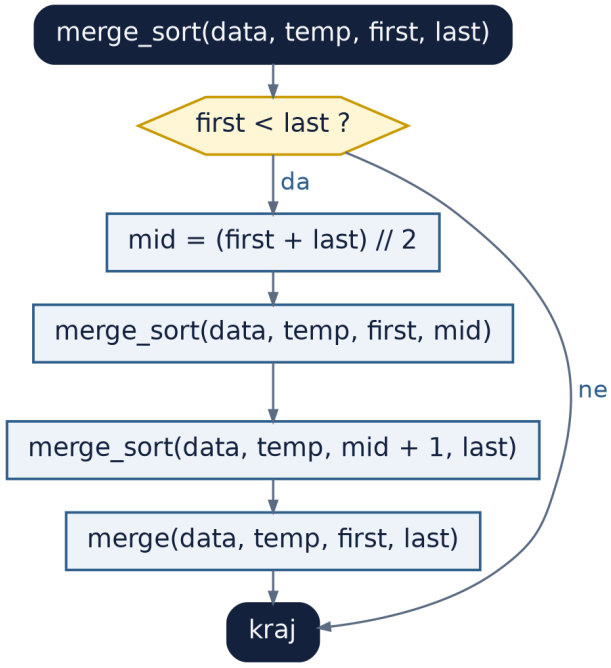
```
[-3, 0, 1, 2, 3, 4, 5, 7, 9, 10]
```

Složenost: **O(n²)**, ali za već sortirano polje samo n − 1 poređenja, odnosno O(n).

Merge sort (metod spajanja)

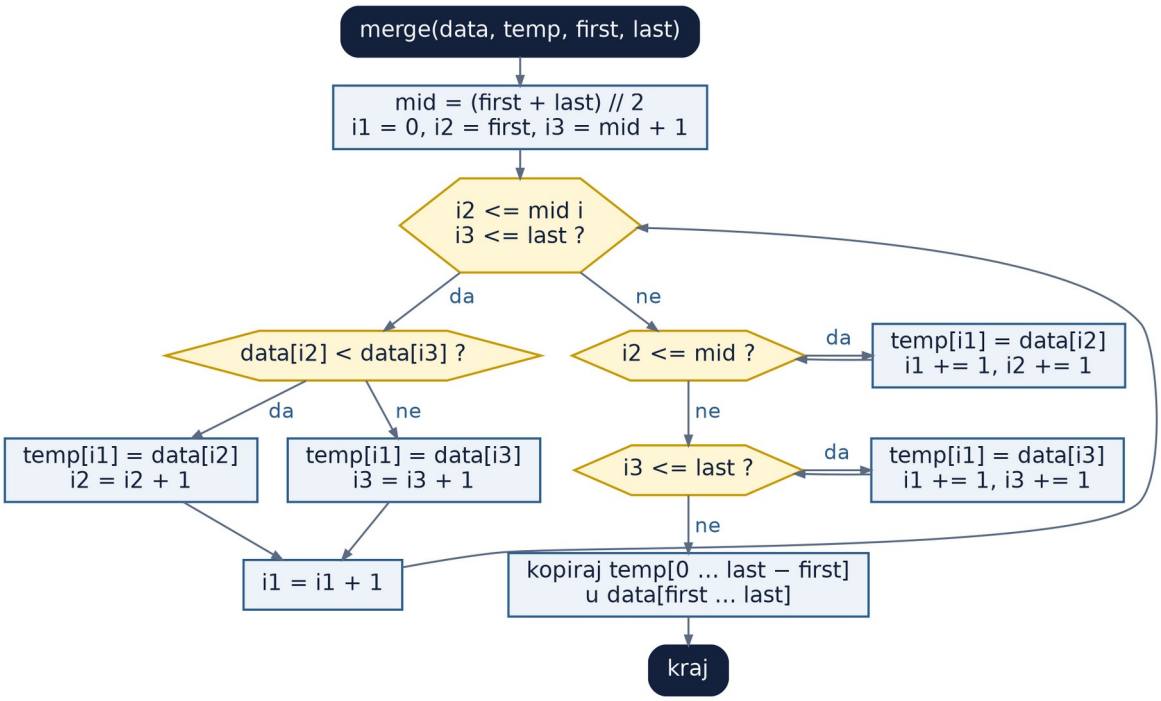
Polje se deli na dve polovine, svaka polovina se rekurzivno sortira, a zatim se dve sortirane polovine spajaju u jednu. Kao u C++ verziji, koristi se pomoćno polje `temp` iste veličine kao polje koje se sortira, i funkcije rade sa granicama opsega `first` i `last`.

Rekurzivna funkcija deli opseg na pola:



Slika 30. *Dijagram toka rekurzivne funkcije merge_sort*

Funkcija `merge` spaja dve sortirane polovine opsega `first … last` preko polja `temp`:



Slika 31. *Dijagram toka funkcije merge*

```
def merge_sort(data, temp, first, last):
    if first < last:
        mid = (first + last) // 2
        merge_sort(data, temp, first, mid)
        merge_sort(data, temp, mid + 1, last)
        merge(data, temp, first, last)

def merge(data, temp, first, last):
    mid = (first + last) // 2
    i1, i2, i3 = 0, first, mid + 1
    while i2 <= mid and i3 <= last:     # obe polovine imaju elemente
        if data[i2] < data[i3]:
            temp[i1] = data[i2]
            i2 += 1
        else:
            temp[i1] = data[i3]
            i3 += 1
        i1 += 1
    while i2 <= mid:                    # ostatak leve polovine
        temp[i1] = data[i2]
        i1 += 1
        i2 += 1
    while i3 <= last:                   # ostatak desne polovine
        temp[i1] = data[i3]
        i1 += 1
        i3 += 1
    for k in range(last - first + 1):   # vrati spojeni deo u data
        data[first + k] = temp[k]

a = [3, 5, 2, 7, 9, 4, 1, 10, -3, 0]
temp = [0] * len(a)
merge_sort(a, temp, 0, len(a) - 1)
print(a)
```

IZLAZ (OUTPUT):

```
[-3, 0, 1, 2, 3, 4, 5, 7, 9, 10]
```

> **Objašnjenje:** Indeks `i2` prolazi kroz levu polovinu (`first … mid`), `i3` kroz desnu (`mid + 1 … last`), a `i1` je sledeće slobodno mesto u `temp`. Manji od dva elementa se upisuje u `temp`, a kada se jedna polovina isprazni, ostatak druge se prepisuje. Na kraju se spojeni deo vraća iz `temp` u `data`.

Složenost: **O(n log n)**. Algoritam se može ubrzati ako se rekurzija zameni iteracijom. Loša strana je što zahteva dodatni memorijski prostor (polje `temp`) iste veličine kao vektor koji se uređuje.

Poređenje efikasnosti

Algoritam	Najbolji slučaj	Srednji slučaj	Najgori slučaj
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Tabela je preuzeta iz prezentacije. Kod selection sort-a najbolji slučaj O(n) odnosi se na broj zamena (n − 1), dok je broj poređenja uvek n(n − 1)/2.

Zadaci za proveru razumevanja gradiva

1. Izračunati koliko operacija (množenja i sabiranja) se štedi kod pristupa M-dimenzionalnom polju korišćenjem unapred izračunatih faktora. Uporediti broj operacija prilikom pristupa na „konvencionalan“ način i sa unapred izračunatim faktorima.
2. Predložiti način za pamćenje trougaone matrice, tako da budu smešteni samo nenulti elementi. Izvesti izraz za pristup elementima gornje trougaone matrice i izraz za pristup elementima donje trougaone matrice.
3. Projektovati klasu za rad sa donjom trougaonom matricom i implementirati metode za upis i čitanje elementa: `get(i, j)` i `put(i, j, value)`. Kao polaznu tačku iskoristiti funkciju `lower_index`.
4. Izračunati koliko memorijskog prostora takva matrica zauzima u odnosu na standardnu „gustu“ matricu i koliko je pristup elementima složeniji.

Zadaci za vežbu sa rešenjima

Sledećih pet zadataka utvrđuje gradivo ovog poglavlja. Pokušajte da svaki zadatak prvo rešite sami, pa tek onda pogledajte rešenje. Kao i u ostatku poglavlja, rešenja ne koriste ugrađene funkcije koje obavljaju isti posao (npr. `min`, `max`, `insert`, `sort`), već je svaki algoritam napisan ručno.

Zadatak 1: Umetanje i brisanje sa pomeranjem elemenata

Napisati funkcije `insert_at(a, n, index, value)` i `delete_at(a, n, index)` za polje `a` fiksnog kapaciteta, u kome je popunjeno prvih `n` mesta. Funkcija za umetanje pomera elemente udesno i upisuje novu vrednost, a funkcija za brisanje pomera elemente ulevo. Obe funkcije vraćaju novi broj popunjenih mesta i podižu izuzetak ako operacija nije moguća.

Rešenje:

```
def insert_at(a, n, index, value):
    if n == len(a):
        raise IndexError("Polje je puno!")
    if index < 0 or index > n:
        raise IndexError("Pozicija van granica!")
    for i in range(n, index, -1):    # pomeranje udesno, od kraja
        a[i] = a[i - 1]
    a[index] = value
    return n + 1

def delete_at(a, n, index):
    if index < 0 or index >= n:
        raise IndexError("Pozicija van granica!")
    for i in range(index, n - 1):    # pomeranje ulevo
        a[i] = a[i + 1]
    a[n - 1] = None
    return n - 1

a = [3, 5, 2, 7, None, None]         # kapacitet 6, popunjeno 4
n = 4
n = insert_at(a, n, 1, 99)
print(a, n)
n = delete_at(a, n, 3)
print(a, n)
```

IZLAZ (OUTPUT):

```
[3, 99, 5, 2, 7, None] 5
[3, 99, 5, 7, None, None] 4
```

> **Objašnjenje:** Pri umetanju se pomeranje radi **od kraja ka poziciji umetanja**, jer bi se pomeranjem od početka elementi međusobno prepisali. Pri brisanju je obrnuto: ide se od pozicije brisanja ka kraju. Obe operacije u najgorem slučaju pomeraju svih n elemenata, pa su složenosti O(n).

Zadatak 2: Najmanji i najveći element u jednom prolazu

Napisati funkciju `min_max(a)` koja u jednom prolazu kroz polje pronalazi najmanji i najveći element i njihove indekse. Za prazno polje funkcija podiže ValueError.

Rešenje:

```
def min_max(a):
    if len(a) == 0:
        raise ValueError("Polje je prazno!")
    i_min = 0
    i_max = 0
    for i in range(1, len(a)):
        if a[i] < a[i_min]:
            i_min = i
        elif a[i] > a[i_max]:
            i_max = i
    return a[i_min], i_min, a[i_max], i_max

a = [3, 5, 2, 7, 9, 4, 1, 10, -3, 0]
mn, i_mn, mx, i_mx = min_max(a)
print("Najmanji:", mn, "na poziciji", i_mn)
print("Najveći:", mx, "na poziciji", i_mx)
```

IZLAZ (OUTPUT):

```
Najmanji: -3 na poziciji 8
Najveći: 10 na poziciji 7
```

Objašnjenje: Pamte se indeksi trenutnog minimuma i maksimuma, a ne same vrednosti, pa se na kraju dobijaju i vrednosti i pozicije. Grana `elif` je dovoljna, jer element koji je manji od trenutnog minimuma ne može istovremeno biti veći od trenutnog maksimuma. Složenost je O(n), sa najviše 2(n − 1) poređenja.

Zadatak 3: Klasa za donju trougaonu matricu

Projektovati klasu `LowerTriangularMatrix` koja pamti samo elemente donje trougaone matrice reda n u jednodimenzionalnom polju. Implementirati metode `get(i, j)` i `put(i, j, value)`. Čitanje elementa iznad dijagonale vraća 0, a upis vrednosti različite od nule iznad dijagonale podiže izuzetak. Izračunati uštedu memorije za n = 4.

Rešenje:

```
class LowerTriangularMatrix:
    def __init__(self, n):
        self.n = n
        self.data = [0] * (n * (n + 1) // 2)

    def _index(self, i, j):
        if i < 0 or i >= self.n or j < 0 or j >= self.n:
            raise IndexError("Indeks van granica!")
        return i * (i + 1) // 2 + j

    def get(self, i, j):
        if j > i:                    # iznad dijagonale
            self._index(i, j)        # samo provera granica
            return 0
        return self.data[self._index(i, j)]

    def put(self, i, j, value):
        if j > i:
            if value != 0:
                raise ValueError("Iznad dijagonale mora biti 0!")
            return
        self.data[self._index(i, j)] = value

m = LowerTriangularMatrix(4)
k = 1
for i in range(4):
    for j in range(i + 1):
        m.put(i, j, k)
        k += 1

for i in range(4):
    for j in range(4):
        print(m.get(i, j), end=" ")
    print()

print("Smešteno elemenata:", len(m.data), "umesto", 4 * 4)
```

IZLAZ (OUTPUT):

```
1 0 0 0
2 3 0 0
4 5 6 0
7 8 9 10
Smešteno elemenata: 10 umesto 16
```

Objašnjenje: Pre vrste i nalazi se 1 + 2 + … + i = i(i + 1)/2 elemenata, pa je element (i, j) na poziciji i(i + 1)/2 + j. Matrica reda n zauzima n(n + 1)/2 mesta umesto n^2, što je za veliko n skoro dvostruko manje. Cena je jedno množenje,

> jedno sabiranje i jedno deljenje više pri svakom pristupu, a složenost pristupa ostaje O(1).

Zadatak 4: Množenje matrica

Klasi `DenseMatrix` iz ovog poglavlja dodati metodu `times(other)` koja vraća proizvod dve matrice. Ako broj kolona prve matrice nije jednak broju vrsta druge, metoda podiže ValueError.

Rešenje:

```
def times(self, other):
    if self.cols != other.rows:
        raise ValueError("Dimenzije matrica se ne slažu!")
    result = DenseMatrix(self.rows, other.cols)
    for i in range(self.rows):
        for j in range(other.cols):
            s = 0.0
            for k in range(self.cols):
                s += self.get(i, k) * other.get(k, j)
            result.put(i, j, s)
    return result

DenseMatrix.times = times

A = DenseMatrix(2, 3)
B = DenseMatrix(3, 2)
vrednosti_a = [1, 2, 3, 4, 5, 6]
vrednosti_b = [7, 8, 9, 10, 11, 12]
for i in range(2):
    for j in range(3):
        A.put(i, j, vrednosti_a[i * 3 + j])
for i in range(3):
    for j in range(2):
        B.put(i, j, vrednosti_b[i * 2 + j])

C = A.times(B)
for i in range(C.rows):
    for j in range(C.cols):
        print(C.get(i, j), end=" ")
    print()
```

IZLAZ (OUTPUT):

```
58.0 64.0
139.0 154.0
```

> **Objašnjenje:** Element C[i][j] je zbir proizvoda i-te vrste matrice A i j-te kolone matrice B. Zato su potrebne tri ugnježđene petlje, a složenost za matrice dimenzija m × p i p × n je O(m · p · n), odnosno $O(n^3)$ za kvadratne matrice.

Zadatak 5: Iterativno binarno traženje sa brojanjem poređenja

Napisati iterativnu verziju binarnog traženja (bez rekurzije) koja pored indeksa vraća i broj izvršenih poređenja. Uporediti broj poređenja sa linearnim traženjem za polje od 16 elemenata.

Rešenje:

```
def binary_search_iter(array, el):
    low = 0
    high = len(array) - 1
    steps = 0
    while low <= high:
        mid = (low + high) // 2
        steps += 1
        if array[mid] > el:
            high = mid - 1
        elif array[mid] < el:
            low = mid + 1
        else:
            return mid, steps
    return -1, steps                # -1: element nije nađen

def linear_search(array, el):
    steps = 0
    for i in range(len(array)):
        steps += 1
        if array[i] == el:
            return i, steps
    return -1, steps

data = [2, 9, 11, 15, 28, 33, 40, 47, 51, 64, 76, 78, 82, 85, 91, 99]
for x in (76, 99, 50):
    print(x, "binarno:", binary_search_iter(data, x),
          "linearno:", linear_search(data, x))
```

IZLAZ (OUTPUT):

```
76 binarno: (10, 4) linearno: (10, 11)
99 binarno: (15, 5) linearno: (15, 16)
50 binarno: (-1, 4) linearno: (-1, 16)
```

Objašnjenje: Rekurzivni poziv je zamenjen petljom koja menja granice `low` i `high`. Petlja se završava kada se element nađe ili kada opseg postane prazan (`low > high`). Za 16 elemenata binarno traženje izvrši najviše 5 koraka ($\lfloor \log_2 16 \rfloor + 1$), dok linearno u najgorem slučaju izvrši 16. Iterativna verzija ne troši memoriju na stek poziva, pa je u praksi i brža od rekurzivne.

POGLAVLJE 11

Lančane liste u Python-u (Linked Lists)

Lančane liste (engl. *linked lists*) su linearne strukture kod kojih svaki element ukazuje na svoje susede u okviru te strukture. Element lančane liste naziva se **čvor**: on čuva podatak i pokazivač (u Python-u referencu) na susedni čvor.

Ovo poglavlje prati prezentaciju *Lančane liste* iz predmeta Strukture podataka. Naslovi i dijagrami odgovaraju prezentaciji, a sav kôd je napisan u Python-u. Kao i u prethodnom poglavlju, liste pravimo sami od čvorova i pokazivača, bez ugrađenih struktura koje bi obavile isti posao.

Polje ili lančana lista

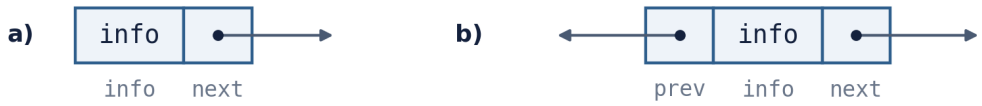
Polje i lančana lista čuvaju niz elemenata, ali na potpuno različit način:

- **Polje** čuva elemente jedan do drugog u memoriji. Zato se adresa i-tog elementa računa direktno i pristup je O(1). Cena toga je što umetanje ili brisanje u sredini pomera sve elemente iza te pozicije.
- **Lančana lista** čuva čvorove bilo gde u memoriji i povezuje ih pokazivačima. Umetanje i brisanje ne pomeraju ostale elemente, već se samo prevežu pokazivači. Cena je što se do i-tog čvora stiže samo prolaskom od početka, pa je pristup O(n).

U Python-u je pokazivač obična referenca na objekat čvora, a prazan pokazivač (u C++ `NULL`) je `None`.

Čvorovi

Čvor jednostruko ulančane liste ima dva polja: `info` (podatak) i `next` (pokazivač na sledeći čvor). Čvor dvostruko ulančane liste ima još i polje `prev`, pokazivač na prethodni čvor.



Slika 32. *Struktura čvora lančane liste: a) čvor jednostruko ulančane liste, b) čvor dvostruko ulančane liste*

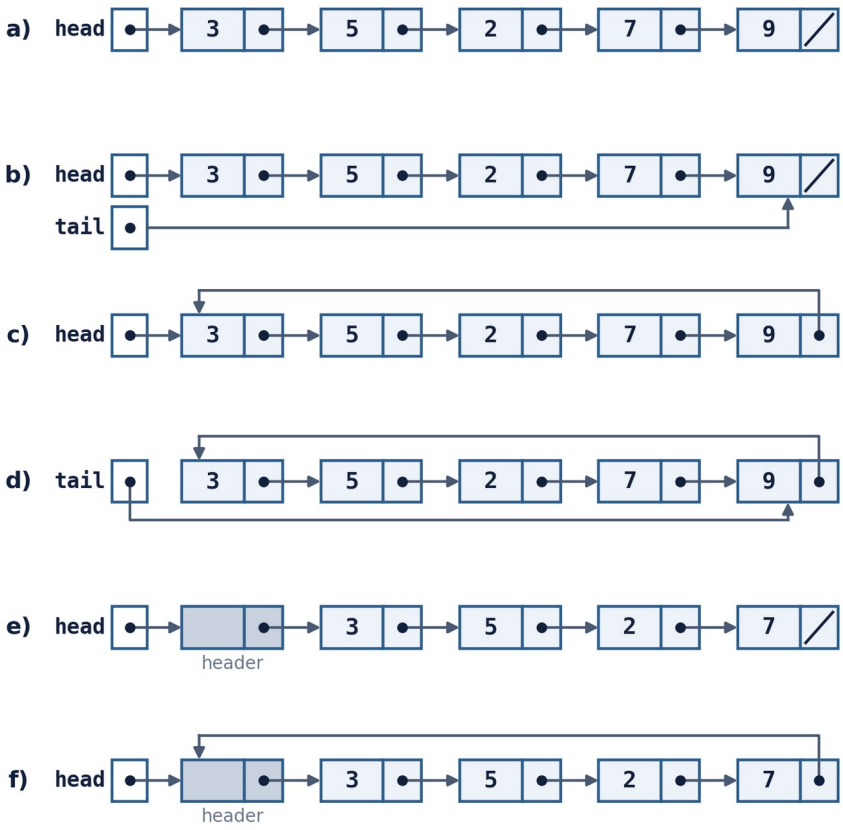
Lančana lista može biti:

- **jednostruko ulančana** (engl. *singly linked list*), gde svaki čvor zna samo svog sledbenika, ili
- **dvostruko ulančana** (engl. *doubly linked list*), gde svaki čvor zna i prethodnika i sledbenika.

Na svim crtežima u ovom poglavlju kosa crta u polju pokazivača označava `None`, odnosno da pokazivač ne pokazuje ni na šta.

Memorijska reprezentacija jednostruko ulančane liste

Ista jednostruko ulančana lista može se organizovati na nekoliko načina, zavisno od toga koje pokazivače lista čuva i da li je poslednji čvor povezan sa prvim:



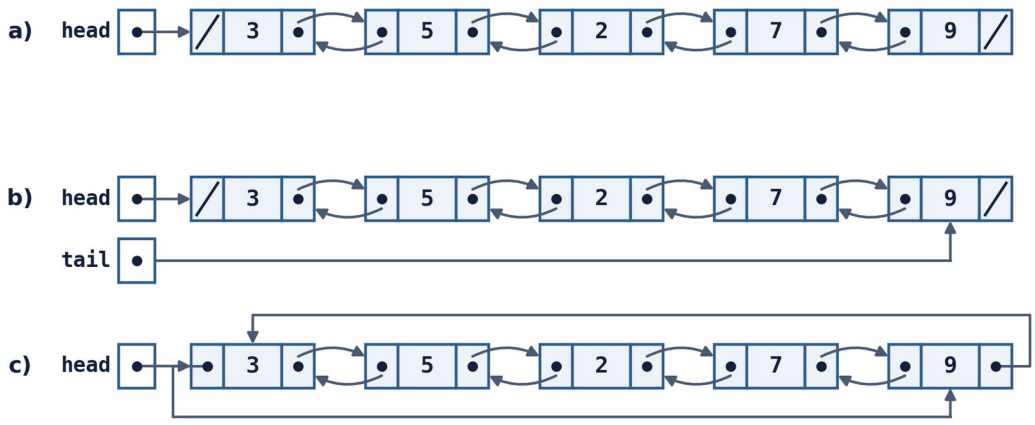
Slika 33. *Jednostruko ulančane liste: a) sa pokazivačem na početak, b) sa pokazivačima na početak i kraj, c) ciklična lista sa pokazivačem na početak, d) ciklična lista sa pokazivačem na kraj, e) lista sa zaglavljem, f) ciklična lista sa zaglavljem*

- **a) Sa pokazivačem na početak (head).** Najjednostavnija varijanta. Dodavanje na početak je O(1), ali dodavanje na kraj zahteva prolazak kroz celu listu.
- **b) Sa pokazivačima na početak i kraj (head i tail).** Pokazivač `tail` omogućava dodavanje na kraj u O(1). Ovu varijantu implementiramo u nastavku.
- **c) Ciklična lista sa pokazivačem na početak.** Poslednji čvor ne pokazuje na `None`, već nazad na prvi čvor.
- **d) Ciklična lista sa pokazivačem na kraj.** Dovoljan je samo `tail`, jer je prvi čvor uvek `tail.next`. Tako se sa jednim pokazivačem dobija brz pristup i početku i kraju.

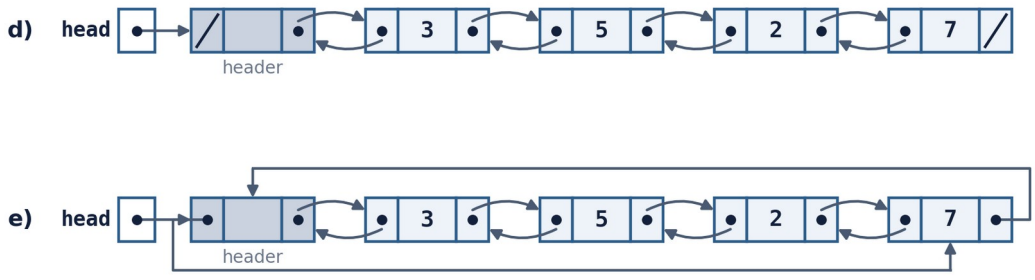
- **e) Lista sa zaglavljem (header).** Prvi čvor je pomoćni čvor bez podatka. Lista zato nikad nema „prazan početak“, pa operacije umetanja i brisanja nemaju poseban slučaj za prvi čvor.
- **f) Ciklična lista sa zaglavljem.** Kombinacija prethodne dve.

Dvostruko ulančane liste

Kod dvostruko ulančane liste svaki čvor ima pokazivač i na sledbenika i na prethodnika, pa se lista može obilaziti u oba smera. Varijante su iste kao kod jednostruko ulančane liste:



Slika 34. *Dvostruko ulančane liste: a) samo sa pokazivačem na početak, b) sa pokazivačima na početak i kraj, c) ciklična dvostruko ulančana lista*



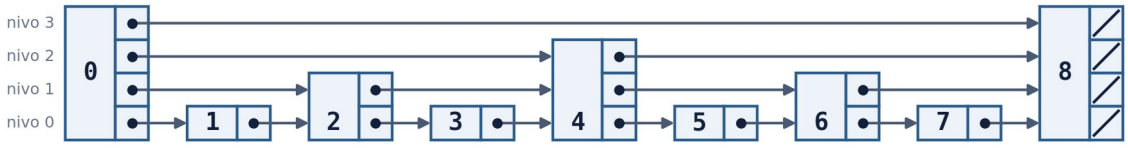
Slika 35. *Dvostruko ulančane liste: d) lista sa zaglavljem i e) ciklična dvostruko ulančana lista sa zaglavljem*

Kod prvog čvora `prev` je `None`, a kod poslednjeg je `next` `None`. Kod ciklične liste prvi čvor preko `prev` pokazuje na poslednji, a poslednji preko `next` na prvi.

> **Napomena:** Dodatni pokazivač troši memoriju (jedan pokazivač više po čvoru), ali donosi dve važne prednosti: brisanje sa kraja je O(1) i čvor se može obrisati bez traženja njegovog prethodnika.

Liste sa preskokom (skip lists)

Traženje u običnoj lančanoj listi je sporo, jer se mora proći čvor po čvor. **Lista sa preskokom** dodaje čvorovima pokazivače koji „preskaču“ više čvorova odjednom, kao brze trake na autoputu:



Slika 36. *Lista sa preskokom: svaki drugi čvor pokazuje dve pozicije napred, svaki četvrti četiri pozicije napred*

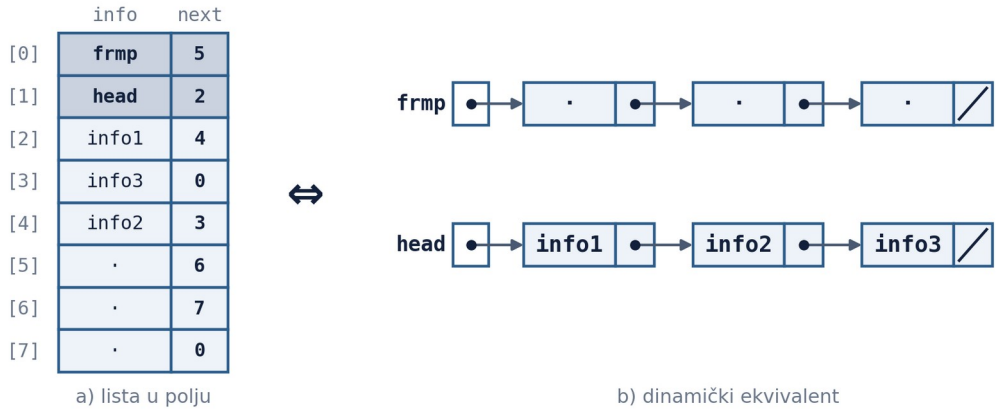
U listi sa preskokom sa n čvorova, za svako k ($1 \le k \le \lfloor \log_2 n \rfloor$) i i ($1 \le i \le \lfloor n / 2^{k-1} \rfloor - 1$), čvor na poziciji $i \cdot 2^{k-1}$ ukazuje na čvor na poziciji $(i + 1) \cdot 2^{k-1}$. Naime, svaki drugi čvor ukazuje na čvor za dve pozicije ispred njega, svaki četvrti za četiri pozicije ispred itd. Najveći broj nivoa je:

$$\text{max_level} = \lfloor \log_2 n \rfloor + 1$$

> **Objašnjenje:** Na slici ima 9 čvorova (0 do 8), pa je max_level = $\lfloor \log_2 9 \rfloor + 1 = 4$. Nivo 0 je obična lista. Na nivou 1 vezani su čvorovi 0, 2, 4, 6 i 8, na nivou 2 čvorovi 0, 4 i 8, a na nivou 3 čvorovi 0 i 8. Traženje počinje na najvišem nivou i spušta se, pa u svakom koraku odbacuje oko polovine preostalih čvorova, kao binarno traženje. Zato je složenost O(log n).

Statička memorijska reprezentacija lančanih listi

Lančana lista se može napraviti i bez dinamičkih čvorova, pomoću polja. Svaki red polja ima podatak i **indeks** sledećeg elementa, koji zamenjuje pokazivač:



Slika 37. *Statička reprezentacija jednostruko ulančane liste: a) lista predstavljena poljem, b) dinamički ekvivalent*

Mesta 0 i 1 u polju čuvaju početke dve liste: `head` je početak liste sa podacima, a `frmp` početak liste slobodnih mesta. Pošto mesto 0 nije element liste, vrednost 0 u koloni `next` označava kraj liste. Obilazak u Python-u:

```
info = ["frmp", "head", "info1", "info3", "info2", None, None, None]
nxt  = [5, 2, 4, 0, 3, 6, 7, 0]

i = nxt[1]                  # početak liste podataka (head)
while i != 0:               # 0 označava kraj liste
    print(info[i], end=" ")
    i = nxt[i]
print()

i = nxt[0]                  # početak liste slobodnih mesta (frmp)
while i != 0:
    print(i, end=" ")
    i = nxt[i]
print()
```

IZLAZ (OUTPUT):

```
info1 info2 info3
5 6 7
```

Objašnjenje: Od mesta 1 (head) čita se da lista počinje na mestu 2 (info1). Kolona next na mestu 2 pokazuje na mesto 4 (info2), a mesto 4 na mesto 3 (info3), čiji je next 0, pa je to kraj. Redosled u listi zato ne zavisi od redosleda u polju. Novi element uzima prvo slobodno mesto iz liste frmp, a obrisani element se vraća u nju.

Ostale definicije i pojmovi

- Lančana lista se naziva **uređena lančana lista** ako njeni elementi slede neki poredak (alfabetski, leksikografski, uređenje u rastući ili opadajući redosled i sl.).
- Ako lančana lista zadržava uređenost nakon svake operacije, naziva se **sortirana lančana lista**. Umetanje u nju je obrađeno u trećem zadatku za vežbu.
- **Samoorganizujuća lista** je lista koja preuređuje svoje čvorove, koristeći neku heuristiku, kako bi poboljšala efikasnost pristupa.

Reorganizacija samoorganizujuće liste se najčešće svodi na jednu od dve operacije:

- **Pomeranje na početak:** čvor kome je poslednji put pristupano premešta se na početak liste, tako da sledeći pristup tom čvoru ne zahteva prolazak kroz listu. Primenjeno je u prvom zadatku za vežbu.
- **Transpozicija:** čvor kome je pristupano menja mesto sa svojim prethodnikom. Često traženi čvorovi tako postepeno putuju ka početku.

Osnovne operacije

Osnovne operacije za rad sa lančanom listom su:

- **traverse:** obilazak lančane liste,
- **find:** pronalazi zadati čvor, ukoliko se on nalazi u listi,
- **insert:** umeće novi čvor u listu: na početak (*addHead*), na kraj (*addTail*), iza zadatog čvora (*insertAfter*) i ispred zadatog čvora (*insertBefore*),
- **delete:** briše zadati čvor iz liste.

Osim ovih operacija, često se implementiraju i **sort** (uređuje elemente liste po zadatom kriterijumu), **copy** (kopira elemente jedne liste u drugu), **concatenate** (nadovezuje jednu listu na kraj druge) i **split** (cepa jednu lančanu listu na dve zasebne liste). Poslednje dve su obrađene u petom zadatku za vežbu.

Složenost osnovnih operacija za listu od n elemenata:

Operacija	Polje	SLL (head i tail)	DLL (head i tail)
Pristup i-tom elementu	`O(1)`	`O(n)`	`O(n)`
Dodavanje na početak	`O(n)`	`O(1)`	`O(1)`

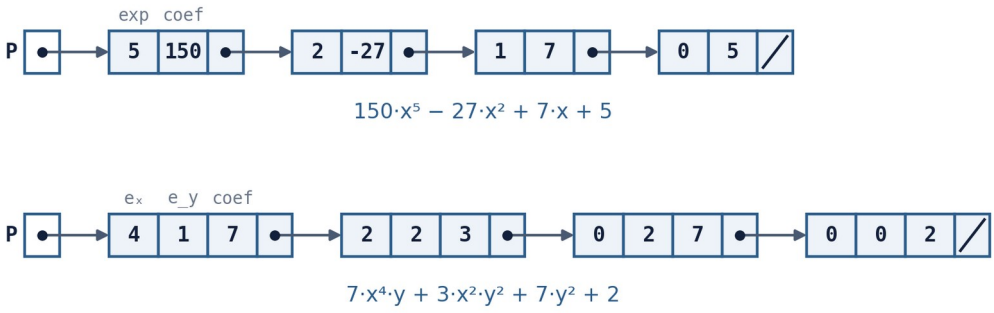
Operacija	Polje	SLL (head i tail)	DLL (head i tail)
Dodavanje na kraj	`O(1)*`	`O(1)`	`O(1)`
Brisanje sa početka	`O(n)`	`O(1)`	`O(1)`
Brisanje sa kraja	`O(1)`	`O(n)`	`O(1)`
Traženje elementa	`O(n)`	`O(n)`	`O(n)`

ako u polju ima slobodnog mesta. Jedina operacija u kojoj je jednostruko ulančana lista sporija od dvostruko ulančane je brisanje sa kraja, jer poslednji čvor ne zna svog prethodnika

Primene lančanih listi

Lančana reprezentacija polinoma

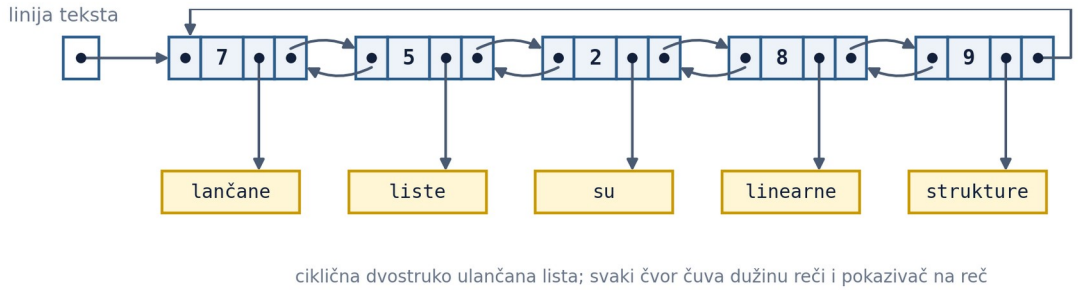
Polinom $P_n(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_i \cdot x^i + \ldots + a_1 \cdot x + a_0$ može se predstaviti listom u kojoj svaki čvor čuva eksponent i koeficijent jednog člana. Čuvaju se samo članovi čiji koeficijent nije nula, poređani po opadajućem eksponentu, pa polinom $x^{1000} + 1$ zauzima samo dva čvora umesto polja od 1001 elementa.



Slika 38. *Polinomi $150 \cdot x^5 - 27 \cdot x^2 + 7 \cdot x + 5$ i $7 \cdot x^4 \cdot y + 3 \cdot x^2 \cdot y^2 + 7 \cdot y^2 + 2$ predstavljeni jednostruko ulančanim listama*

Za polinom sa dve promenljive čvor čuva dva eksponenta, za x i za y. Sabiranje dva polinoma svodi se na spajanje dve sortirane liste i obrađeno je u četvrtom zadatku za vežbu.

Lančana reprezentacija teksta

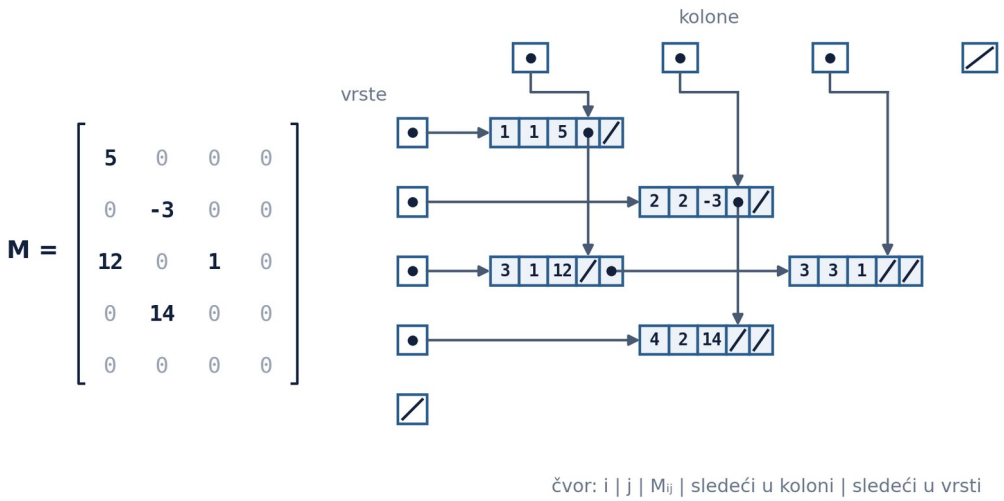


Slika 39. *Linija teksta predstavljena cikličnom dvostruko ulančanom listom*

Linija teksta je ciklična dvostruko ulančana lista u kojoj svaki čvor čuva dužinu reči i pokazivač na samu reč. Umetanje ili brisanje reči u sredini linije menja samo pokazivače susednih čvorova, a ostatak teksta se ne pomera, kao što bi bio slučaj kada bi tekst bio jedno dugačko polje znakova.

Lančana reprezentacija retko posednute matrice

Retko posednuta matrica ima mnogo nula. Umesto cele matrice čuvaju se samo nenulti elementi, svaki kao čvor sa pet polja: vrsta i, kolona j, vrednost M_{ij}, pokazivač na sledeći element u istoj koloni i pokazivač na sledeći element u istoj vrsti. Svaka vrsta i svaka kolona ima svoj početni pokazivač:



Slika 40. *Lančana reprezentacija retko posednute matrice dimenzija 5 × 4*

> **Objašnjenje:** Matrica na slici ima 20 elemenata, a samo 5 različitih od nule. Lančana reprezentacija čuva 5 čvorova i 9 početnih pokazivača. Kroz vrstu 3

> prolazi se horizontalnim pokazivačima (12, pa 1), a kroz kolonu 1 vertikalnim (5, pa 12). Za velike retke matrice, na primer 10 000 × 10 000 sa nekoliko hiljada nenultih elemenata, ušteda memorije je ogromna.

Jednostruko ulančana lista u Python-u

Klasa čvora

Čvor jednostruko ulančane liste čuva podatak `info` i pokazivač `next` na sledbenika:

```
class SLLNode:
    def __init__(self, info=None, next_node=None):
        self.info = info          # podatak
        self.next = next_node     # pokazivač na sledbenika

    def is_equal(self, el):
        return self.info == el
```

U C++ verziji postoje tri konstruktora (bez argumenata, sa podatkom i sa podatkom i sledbenikom). U Python-u ih zamenjuje jedan `__init__` sa podrazumevanim vrednostima: `SLLNode()`, `SLLNode(5)` i `SLLNode(5, drugi_cvor)` su sva tri ispravna poziva. Destruktor nije potreban. Parametar se zove `next_node` da se ne bi poklopio sa ugrađenom funkcijom `next`.

Klasa liste i dodavanje elemenata

Lista čuva samo dva pokazivača: `head` na prvi i `tail` na poslednji čvor. Prazna lista ima oba pokazivača jednaka `None`:

```
class SLList:
    def __init__(self):
        self.head = None          # pokazivač na početak liste
        self.tail = None          # pokazivač na kraj liste

    def is_empty(self):
        return self.head is None

    def add_to_head(self, el):
        self.head = SLLNode(el, self.head)
        if self.tail is None:
            self.tail = self.head

    def add_to_tail(self, el):
        if not self.is_empty():
            self.tail.next = SLLNode(el)
            self.tail = self.tail.next
        else:
            self.head = self.tail = SLLNode(el)
```

- `add_to_head` pravi novi čvor čiji je `next` stari prvi čvor, i proglašava ga za `head`. Ako je lista bila prazna, novi čvor je ujedno i poslednji, pa se postavlja i `tail`.
- `add_to_tail` kači novi čvor iza starog poslednjeg (`tail.next`) i pomera `tail` na njega. Ako je lista bila prazna, `head` i `tail` pokazuju na isti, jedini čvor.
- Obe operacije imaju složenost O(1), jer ne prolaze kroz listu.

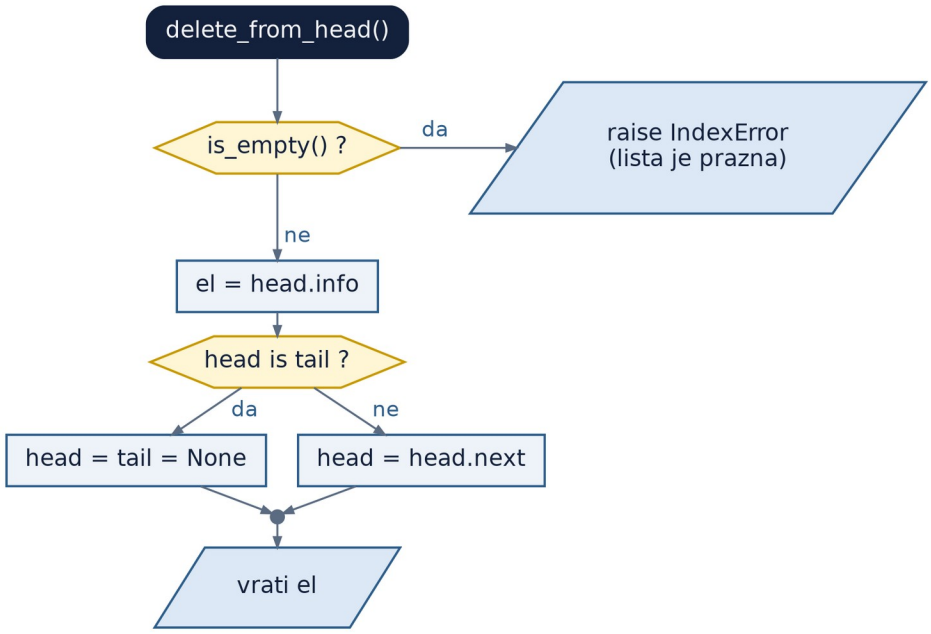
Obilazak i traženje

```
    def print_all(self):
        tmp = self.head
        while tmp is not None:
            print(tmp.info, end=" ")
            tmp = tmp.next
        print()

    def find_node(self, el):
        tmp = self.head
        while tmp is not None and not tmp.is_equal(el):
            tmp = tmp.next
        return tmp

    def is_in_list(self, el):
        return self.find_node(el) is not None

    def clear(self):
        while not self.is_empty():
            self.delete_from_head()
```

Obilazak liste uvek ima isti oblik: pomoćni pokazivač `tmp` kreće od `head` i prelazi na `tmp.next` dok ne dođe do `None`. U C++ verziji to je bila petlja `for (tmp = head; tmp != NULL; tmp = tmp->next)`.

> **Napomena:** Python nema destruktor u smislu C++-a. Čvor se automatski briše kada na njega više ne pokazuje nijedna referenca. Metoda `clear()` prazni listu brisanjem čvor po čvor, kao destruktor iz prezentacije, ali bi bilo dovoljno i samo postaviti `head` i `tail` na `None`.

Brisanje elementa sa početka



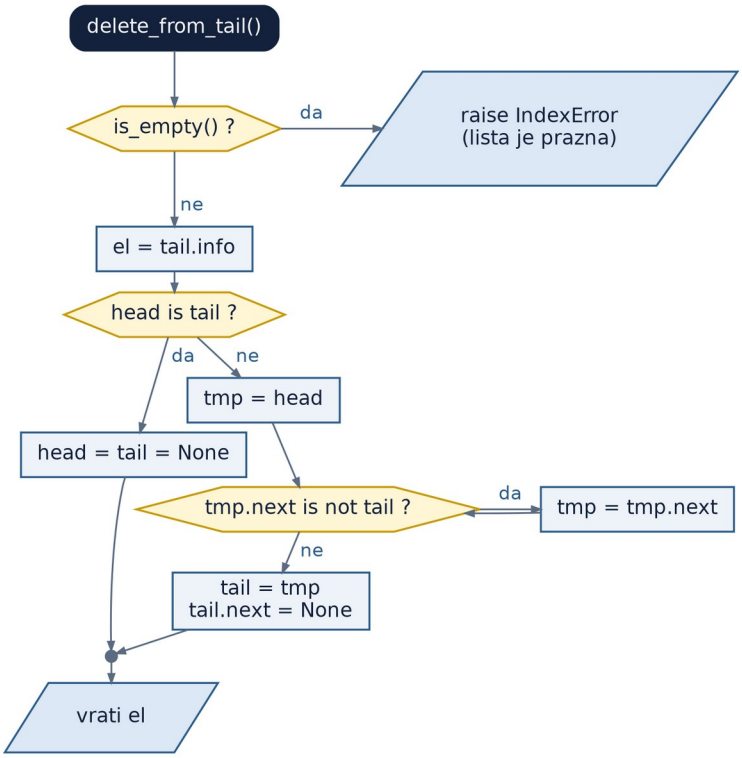
Slika 41. *Dijagram toka metode delete_from_head*

```
    def delete_from_head(self):
        if self.is_empty():
            raise IndexError("Lista je prazna!")
        el = self.head.info
        if self.head is self.tail:
            self.head = self.tail = None
        else:
            self.head = self.head.next
        return el
```

> **Objašnjenje:** Ako je lista prazna, podiže se izuzetak (u C++ SBPException). Inače se pamti podatak iz prvog čvora i `head` se pomera na drugi čvor. Ako je čvor bio jedini (`head is tail`), lista postaje prazna. Operator `is` proverava da

li su dve promenljive isti objekat, a ne samo da li su jednake. Umesto `delete tmp` dovoljno je izgubiti referencu na stari prvi čvor. Složenost je O(1).

Brisanje elementa sa kraja

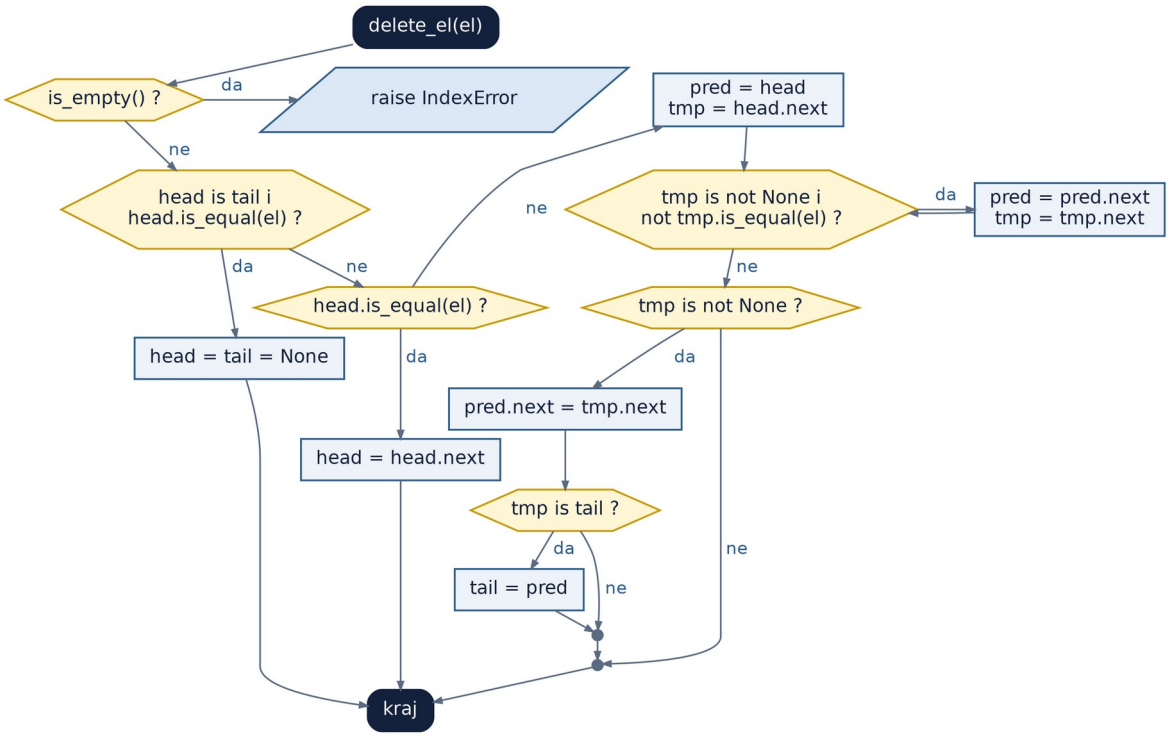


Slika 42. *Dijagram toka metode delete_from_tail*

```
    def delete_from_tail(self):
        if self.is_empty():
            raise IndexError("Lista je prazna!")
        el = self.tail.info
        if self.head is self.tail:
            self.head = self.tail = None
        else:
            tmp = self.head
            while tmp.next is not self.tail:
                tmp = tmp.next
            self.tail = tmp
            self.tail.next = None
        return el
```

Objašnjenje: Posle brisanja poslednjeg čvora novi `tail` mora biti pretposlednji čvor. Pošto čvor jednostruko ulančane liste ne zna svog prethodnika, pretposlednji se traži petljom od početka, dok `tmp.next` ne postane `tail`. Zbog toga je složenost **O(n)**, i to je glavna slabost jednostruko ulančane liste.

Brisanje zadatog elementa



Slika 43. *Dijagram toka metode delete_el jednostruko ulančane liste*

```
    def delete_el(self, el):
        if self.is_empty():
            raise IndexError("Lista je prazna!")
        if self.head is self.tail and self.head.is_equal(el):
            self.head = self.tail = None
        elif self.head.is_equal(el):
            self.head = self.head.next
        else:
            pred = self.head
            tmp = self.head.next
            while tmp is not None and not tmp.is_equal(el):
                pred = pred.next
                tmp = tmp.next
            if tmp is not None:
                pred.next = tmp.next
                if tmp is self.tail:
                    self.tail = pred
```

Metoda razlikuje četiri slučaja:

1. lista je prazna, pa se podiže izuzetak;
2. lista ima jedan čvor i on sadrži traženi element, pa lista postaje prazna;
3. traženi element je u prvom čvoru, pa se `head` pomera na drugi čvor;

4. u ostalim slučajevima dva pokazivača, `pred` i `tmp`, idu zajedno kroz listu. Kada `tmp` nađe element, `pred.next` se preusmerava na `tmp.next`, pa se `tmp` preskače. Ako je obrisan poslednji čvor, `tail` se pomera na `pred`.

Ako element nije u listi, petlja se završava sa `tmp is None` i lista ostaje nepromenjena. Složenost je O(n).

Primer korišćenja

```
lista = SLList()
for x in [3, 5, 2, 7, 9]:
    lista.add_to_tail(x)
lista.add_to_head(1)
lista.print_all()

print(lista.delete_from_head(), lista.delete_from_tail())
lista.delete_el(2)
lista.print_all()
print(lista.is_in_list(7), lista.is_in_list(2))
print(lista.tail.info)
```

IZLAZ (OUTPUT):

```
1 3 5 2 7 9
1 9
3 5 7
True False
7
```

Dvostruko ulančana lista u Python-u

Klasa čvora

```
class DLLNode:
    def __init__(self, info=None, prev=None, next_node=None):
        self.info = info
        self.prev = prev          # pokazivač na prethodnika
        self.next = next_node     # pokazivač na sledbenika

    def is_equal(self, el):
        return self.info == el
```

Klasa liste i dodavanje elemenata

```
class DLList:
    def __init__(self):
        self.head = None
        self.tail = None

    def add_to_head(self, el):
        if not self.is_empty():
            self.head = DLLNode(el, None, self.head)
            self.head.next.prev = self.head
        else:
            self.head = self.tail = DLLNode(el)

    def add_to_tail(self, el):
        if not self.is_empty():
            self.tail = DLLNode(el, self.tail, None)
            self.tail.prev.next = self.tail
        else:
            self.head = self.tail = DLLNode(el)
```

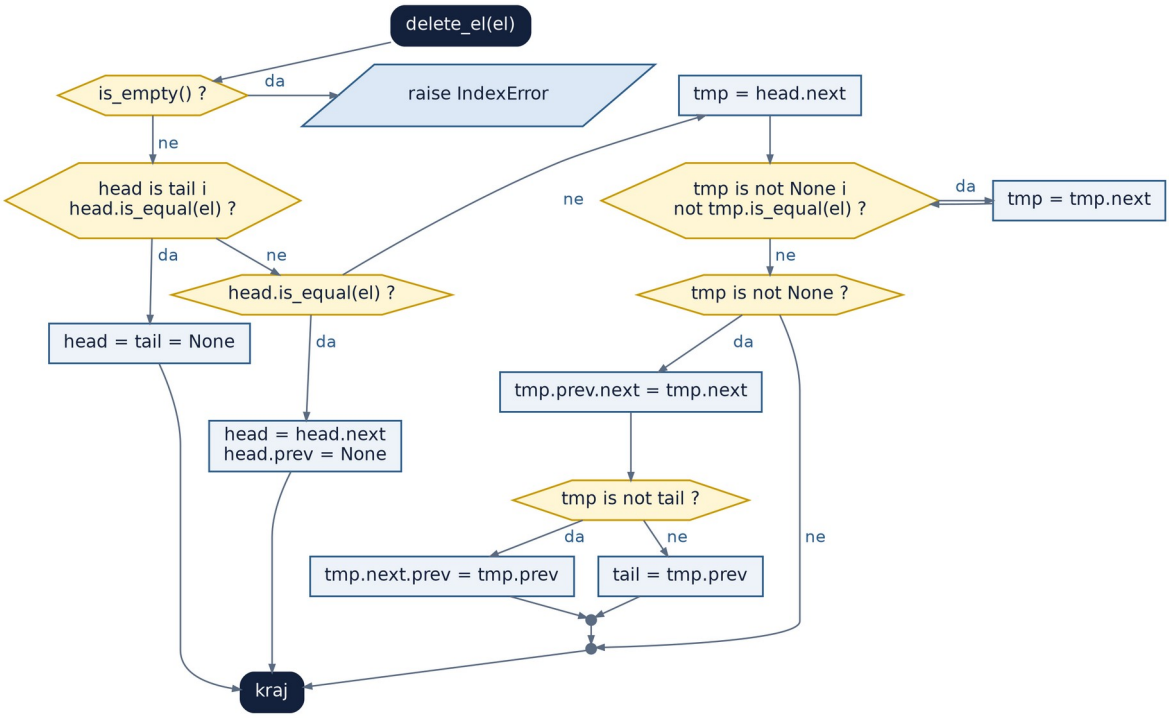
Objašnjenje: Pored novog čvora mora se ažurirati i njegov sused. Kod dodavanja na početak novi čvor dobija `next` na stari prvi čvor, a stari prvi čvor dobija `prev` na novi (`self.head.next.prev = self.head`). Kod dodavanja na kraj je obrnuto: novi čvor dobija `prev` na stari poslednji, a stari poslednji dobija `next` na novi.

Brisanje sa početka i sa kraja

```
    def delete_from_head(self):
        if self.is_empty():
            raise IndexError("Lista je prazna!")
        el = self.head.info
        if self.head is self.tail:
            self.head = self.tail = None
        else:
            self.head = self.head.next
            self.head.prev = None
        return el

    def delete_from_tail(self):
        if self.is_empty():
            raise IndexError("Lista je prazna!")
        el = self.tail.info
        if self.head is self.tail:
            self.head = self.tail = None
        else:
            self.tail = self.tail.prev
            self.tail.next = None
        return el
```

Za razliku od jednostruko ulančane liste, brisanje sa kraja ne traži pretposlednji čvor: to je `tail.prev`. Obe operacije su zato O(1). Novi prvi čvor dobija `prev = None`, a novi poslednji `next = None`.

Brisanje zadatog elementa



Slika 44. *Dijagram toka metode delete_el dvostruko ulančane liste*

```
    def delete_el(self, el):
        if self.is_empty():
            raise IndexError("Lista je prazna!")
        if self.head is self.tail and self.head.is_equal(el):
            self.head = self.tail = None
        elif self.head.is_equal(el):
            self.head = self.head.next
            self.head.prev = None
        else:
            tmp = self.head.next
            while tmp is not None and not tmp.is_equal(el):
                tmp = tmp.next
            if tmp is not None:
                tmp.prev.next = tmp.next
                if tmp is not self.tail:
                    tmp.next.prev = tmp.prev
                else:
                    self.tail = tmp.prev
```

> **Objašnjenje:** Nije potreban pokazivač `pred`, jer je prethodnik već poznat: `tmp.prev`. Kada se nađe čvor `tmp`, prevezuju se oba suseda. Prethodnik dobija `next` na sledbenika (`tmp.prev.next = tmp.next`). Ako `tmp` nije poslednji, sledbenik dobija `prev` na prethodnika (`tmp.next.prev = tmp.prev`); ako jeste, `tail` se pomera na `tmp.prev`.

Primer korišćenja

```
    def print_reverse(self):
        tmp = self.tail
        while tmp is not None:
            print(tmp.info, end=" ")
            tmp = tmp.prev
        print()

d = DLList()
for x in [3, 5, 2, 7, 9]:
    d.add_to_tail(x)
d.delete_el(2)
d.print_all()
d.print_reverse()
print(d.delete_from_tail(), d.tail.info)
```

IZLAZ (OUTPUT):

```
3 5 7 9
9 7 5 3
9 7
```

Metoda `print_reverse` obilazi listu unazad, od `tail` preko pokazivača `prev`, što kod jednostruko ulančane liste nije moguće bez dodatne memorije.

Skip lista u Python-u

Čvor i klasa skip liste

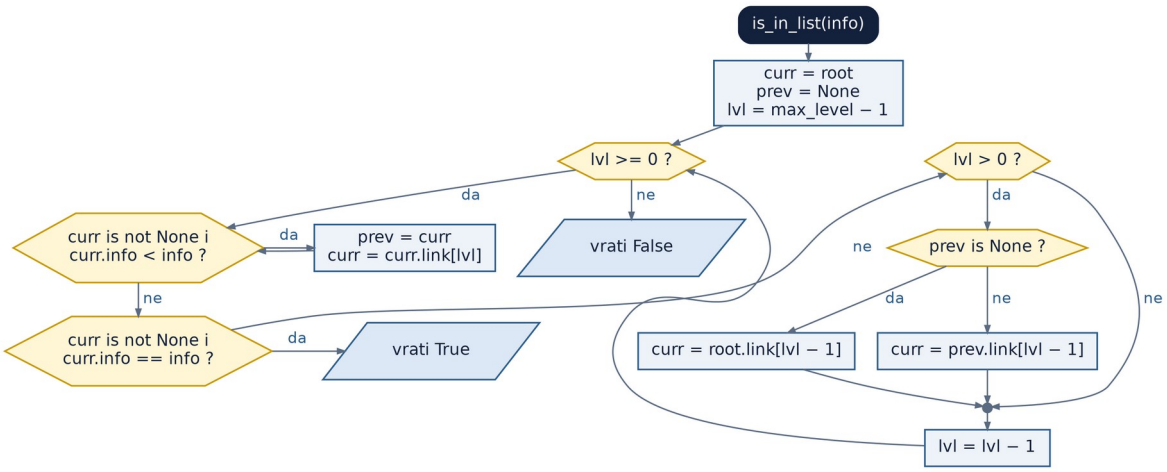
```
class SkipListNode:
    def __init__(self, info=None, level=0):
        self.info = info
        self.level = level
        self.link = [None] * level    # pokazivači, po jedan za svaki nivo
```

Čvor nivoa `level` ima `level` pokazivača: `link[0]` pokazuje na sledeći čvor (obična lista), `link[1]` preskače na čvor dve pozicije napred, `link[2]` četiri pozicije napred, i tako dalje.

```
class SkipList:
    def __init__(self, max_level):
        self.max_level = max_level    # maksimalni nivo čvora u listi
        self.root = None              # početak liste
        self.no_elem = 0              # broj čvorova u listi

    def is_empty(self):
        return self.root is None

    def build(self, values):
        """Pravi savršenu skip listu od sortiranog polja values."""
        n = len(values)
        nodes = [None] * n
        for p in range(n):
            level = 1
            step = 2
            while level < self.max_level and p % step == 0:
                level += 1
                step *= 2
            nodes[p] = SkipListNode(values[p], level)
        for lvl in range(self.max_level):
            last = None
            for p in range(n):
                if nodes[p].level > lvl:
                    if last is not None:
                        last.link[lvl] = nodes[p]
                    last = nodes[p]
        if n > 0:
            self.root = nodes[0]
        self.no_elem = n
```

U originalnoj prezentaciji čvorovi se dodaju metodom `skipListInsert`, koja nivo novog čvora bira slučajno. Mi zbog jasnoće koristimo metodu `build`, koja od sortiranog polja pravi „savršenu“ skip listu po pravilu sa slike 36: čvor na poziciji p dobija jedan nivo više za svaki stepen dvojke kojim je p deljiv, a prvi čvor dobija sve nivoe. Druga petlja zatim, za svaki nivo, povezuje redom sve čvorove koji imaju taj nivo.

Traženje u skip-listi



Slika 45. *Dijagram toka metode is_in_list skip liste*

```
    def is_in_list(self, info):
        if self.is_empty():
            return False
        curr = self.root
        prev = None
        lvl = self.max_level - 1
        while lvl >= 0:
            while curr is not None and curr.info < info:
                prev = curr                    # manji je: pređi na sledeći
čvor
                curr = curr.link[lvl]
            if curr is not None and curr.info == info:
                return True                    # postoji čvor sa tim info
poljem
            if lvl > 0:                        # pređi na niži nivo
                if prev is None:               # polazeći od root-a
                    curr = self.root.link[lvl - 1]
                else:                          # ili od prethodnog čvora
                    curr = prev.link[lvl - 1]
            lvl -= 1
        return False
```

Objašnjenje: Traženje kreće od najvišeg nivoa. Na svakom nivou ide se napred dok je sledeći čvor manji od traženog, a `prev` pamti poslednji takav čvor. Ako se naiđe na traženu vrednost, vraća se `True`. Inače se spušta na niži nivo i nastavlja od `prev`. Kada se ni na nivou 0 ne nađe element, vraća se `False`.

```
sk = SkipList(4)
sk.build([0, 1, 2, 3, 4, 5, 6, 7, 8])
print(sk.is_in_list(5), sk.is_in_list(10), sk.is_in_list(-1))
```

IZLAZ (OUTPUT):

```
True False False
```

Traženje broja 5 ide ovako: na nivou 3 iz čvora 0 sledeći je 8, što je veće od 5, pa se spušta. Na nivou 2 iz čvora 0 ide se do 4, jer je 4 < 5, a zatim je sledeći 8, pa se spušta. Na nivou 1 iz čvora 4 sledeći je 6, pa se spušta. Na nivou 0 iz čvora 4 sledeći je 5, i element je nađen. Umesto šest koraka kroz običnu listu bilo je potrebno samo nekoliko poređenja.

Zadaci za proveru razumevanja gradiva

1. Projektovati klasu SLList1 za rad sa jednostruko ulančanom listom čiji čvorovi sadrže po dva stringa (prvi je „nepoznata“ reč, a drugi njeno objašnjenje) i implementirati funkcije *explain(nepoznata_rec)*, koja objašnjava zadatu reč tako što pronalazi njeno objašnjenje u listi i premešta pronađeni čvor na početak liste da bi ubrzala sledeće traženje te reči, i *update(druga_lista)*, koja u tekuću listu dodaje sve reči i odgovarajuća objašnjenja iz liste druga_lista, ako ne postoje u tekućoj listi.
2. Nacrtati algoritam za traženje u skip-listi.
3. Odrediti funkciju složenosti traženja u skip listi f(n, l), gde je n broj elemenata liste, a l nivo liste (tj. maksimalni nivo čvora).
4. Uporediti veličinu i brzinu pretrage skip-liste u odnosu na „običnu“ jednostruko ulančanu listu.

Prvi zadatak je rešen među zadacima za vežbu, a dijagram toka za drugi zadatak dat je na slici 45.

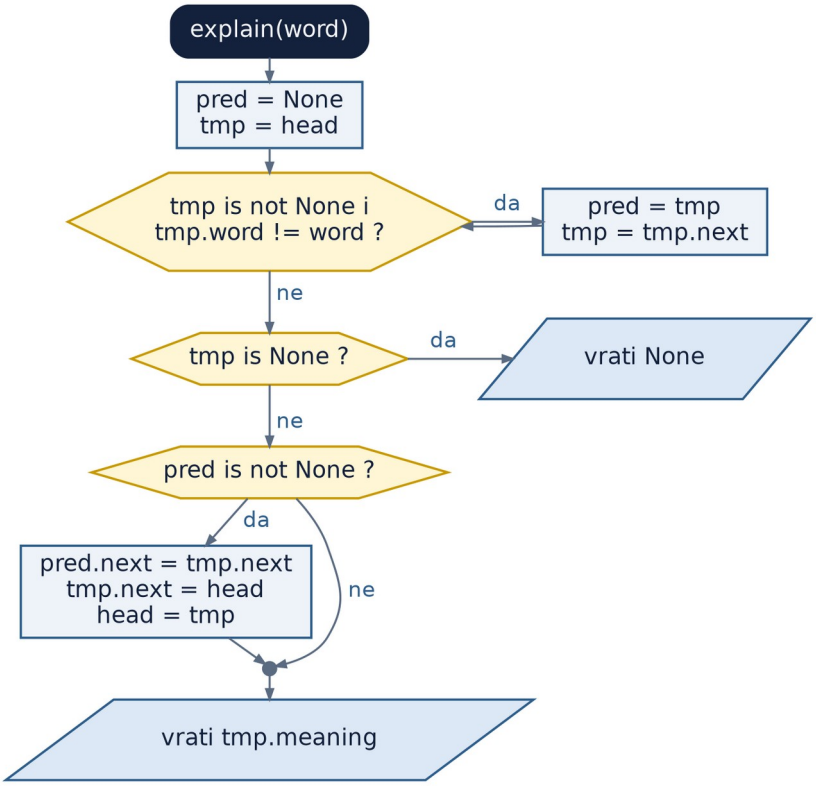
Zadaci za vežbu sa rešenjima

Za svaki zadatak dat je algoritam u obliku dijagrama toka, kompletno rešenje u Python-u, izlaz programa i objašnjenje. Rešenja koriste klase `SLLNode` i `SLList` iz ovog poglavlja i ne koriste ugrađene strukture koje obavljaju isti posao. Pokušajte da svaki zadatak prvo rešite sami.

Zadatak 1: Rečnik sa pomeranjem na početak (SLList1)

Projektovati klasu `SLList1` čiji čvorovi sadrže reč i njeno objašnjenje. Implementirati metodu `explain(word)`, koja vraća objašnjenje reči i premešta pronađeni čvor na početak liste (samoorganizujuća lista, pomeranje na početak), i metodu `update(other)`, koja dodaje sve reči iz liste `other` koje već ne postoje u tekućoj listi.

Algoritam metode explain:



Slika 46. *Dijagram toka metode explain (pomeranje pronađenog čvora na početak)*

Rešenje:

1. Klase potrebne za rešenje: čvor rečnika `DictNode` (reč, objašnjenje i pokazivač na sledbenika) i lista `SLList1` sa svim metodama. Rešenje ne koristi druge klase iz poglavlja:

```
class DictNode:
    def __init__(self, word, meaning, next_node=None):
        self.word = word
        self.meaning = meaning
        self.next = next_node

class SLList1:
    def __init__(self):
        self.head = None

    def add(self, word, meaning):
        self.head = DictNode(word, meaning, self.head)

    def explain(self, word):
        pred = None
        tmp = self.head
        while tmp is not None and tmp.word != word:
            pred = tmp
            tmp = tmp.next
        if tmp is None:
            return None                  # reč nije u rečniku
        if pred is not None:             # premesti čvor na početak
            pred.next = tmp.next
            tmp.next = self.head
            self.head = tmp
        return tmp.meaning

    def contains(self, word):
        tmp = self.head
        while tmp is not None:
            if tmp.word == word:
                return True
            tmp = tmp.next
        return False

    def update(self, other):
        tmp = other.head
        while tmp is not None:
            if not self.contains(tmp.word):
                self.add(tmp.word, tmp.meaning)
            tmp = tmp.next

    def print_words(self):
        tmp = self.head
        while tmp is not None:
            print(tmp.word, end=" ")
            tmp = tmp.next
        print()
```

2. Primer korišćenja (nastavak istog programa):

```
r = SLList1()
r.add("stek", "struktura LIFO")
r.add("red", "struktura FIFO")
r.add("čvor", "element lančane liste")
r.print_words()

print(r.explain("stek"))
r.print_words()                 # "stek" je sada na početku

d = SLList1()
d.add("graf", "skup čvorova i grana")
d.add("red", "struktura FIFO")
r.update(d)                     # dodaje se samo "graf"
r.print_words()
print(r.explain("nepoznata"))
```

IZLAZ (OUTPUT):

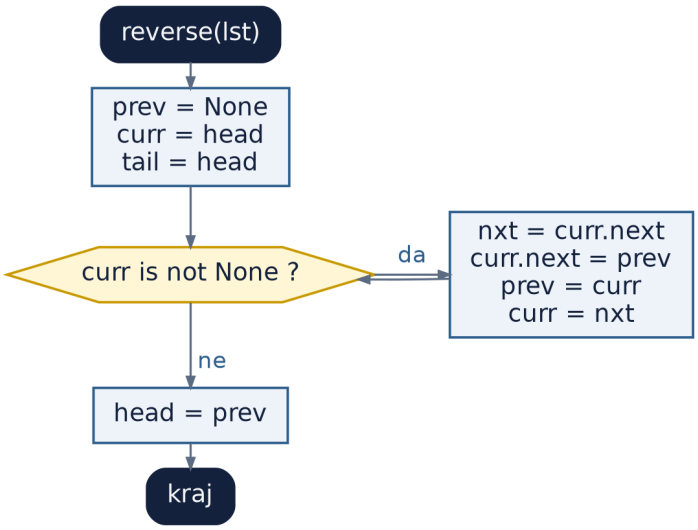
```
čvor red stek
struktura LIFO
stek čvor red
graf stek čvor red
None
```

Objašnjenje: Metoda `explain` traži reč sa dva pokazivača, `pred` i `tmp`, kao `delete_el`. Kada nađe čvor, isključi ga iz liste (`pred.next = tmp.next`) i ubaci ga na početak (`tmp.next = head`, `head = tmp`). Ako je čvor već prvi (`pred is None`), ništa se ne pomera. Često tražene reči tako ostaju blizu početka, pa se sledeći put nalaze brže. Metoda `update` za svaku reč iz druge liste proverava da li već postoji (`contains`) i dodaje samo nove, pa je njena složenost O(n · m).

Zadatak 2: Obrtanje liste u mestu

Napisati funkciju `reverse(lst)` koja obrće redosled čvorova jednostruko ulančane liste, bez pravljenja novih čvorova, samo preusmeravanjem pokazivača.

Algoritam:



Slika 47. *Dijagram toka funkcije reverse*

Rešenje:

1. Klase potrebne za rešenje. Čvor `SLLNode` i jednostruko ulančana lista `SLList` sa metodama `is_empty`, `add_to_tail` i `print_all`. Kod je isti kao u odeljku „Jednostruko ulančana lista u Python-u“, pa se program može pokrenuti samostalno:

```
class SLLNode:
    def __init__(self, info=None, next_node=None):
        self.info = info          # podatak
        self.next = next_node     # pokazivač na sledbenika

    def is_equal(self, el):
        return self.info == el


class SLList:
    def __init__(self):
        self.head = None          # pokazivač na početak liste
        self.tail = None          # pokazivač na kraj liste

    def is_empty(self):
        return self.head is None

    def add_to_tail(self, el):
        if not self.is_empty():
            self.tail.next = SLLNode(el)
            self.tail = self.tail.next
        else:
            self.head = self.tail = SLLNode(el)

    def print_all(self):
        tmp = self.head
        while tmp is not None:
            print(tmp.info, end=" ")
            tmp = tmp.next
        print()
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def reverse(lst):
    prev = None
    curr = lst.head
    lst.tail = lst.head          # stari prvi postaje poslednji
    while curr is not None:
        nxt = curr.next          # zapamti ostatak liste
        curr.next = prev         # okreni pokazivač unazad
        prev = curr
        curr = nxt
    lst.head = prev              # stari poslednji postaje prvi

lista = SLList()
for x in [1, 2, 3, 4, 5]:
    lista.add_to_tail(x)
reverse(lista)
lista.print_all()
print(lista.head.info, lista.tail.info)
```

IZLAZ (OUTPUT):

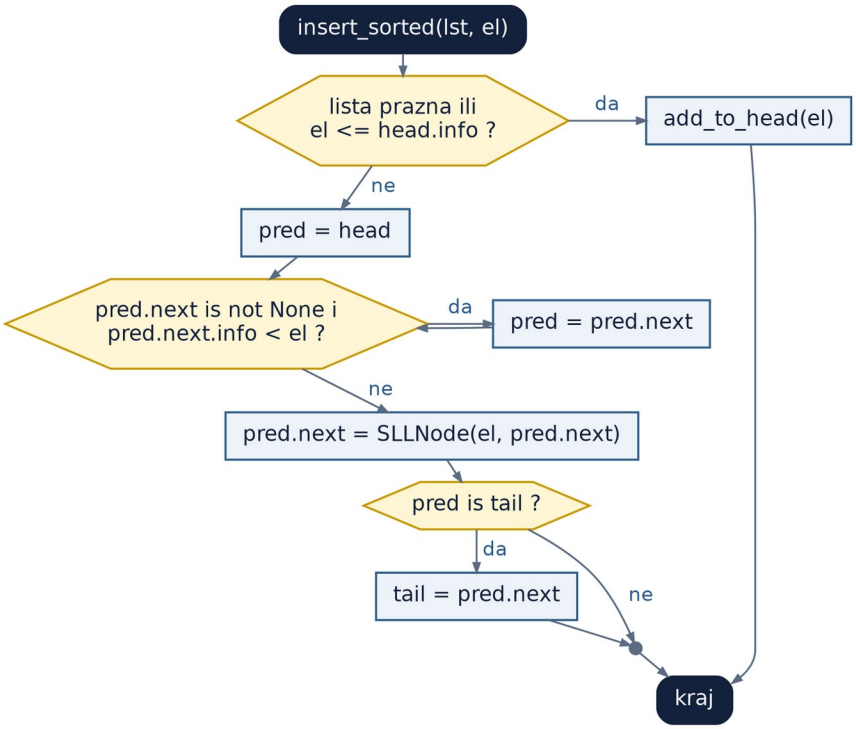
```
5 4 3 2 1
5 1
```

Objašnjenje: Funkcija prolazi kroz listu jednom i svakom čvoru okreće pokazivač `next` tako da pokazuje na prethodni čvor. Pre okretanja mora se zapamtiti `curr.next` u `nxt`, jer bi se inače izgubio ostatak liste. Kada se petlja završi, `prev` pokazuje na stari poslednji čvor, koji postaje novi `head`. Složenost je O(n), a dodatna memorija O(1), samo tri pokazivača.

Zadatak 3: Umetanje u sortiranu lančanu listu

Napisati funkciju `insert_sorted(lst, el)` koja umeće element u rastuće sortiranu listu tako da lista ostane sortirana.

Algoritam:



Slika 48. *Dijagram toka funkcije insert_sorted*

Rešenje:

1. Klase potrebne za rešenje. Čvor `SLLNode` i jednostruko ulančana lista `SLList` sa metodama `is_empty`, `add_to_head` i `print_all`. Kod je isti kao u odeljku „Jednostruko ulančana lista u Python-u“, pa se program može pokrenuti samostalno:

```
class SLLNode:
    def __init__(self, info=None, next_node=None):
        self.info = info          # podatak
        self.next = next_node     # pokazivač na sledbenika

    def is_equal(self, el):
        return self.info == el


class SLList:
    def __init__(self):
        self.head = None          # pokazivač na početak liste
        self.tail = None          # pokazivač na kraj liste

    def is_empty(self):
        return self.head is None

    def add_to_head(self, el):
        self.head = SLLNode(el, self.head)
        if self.tail is None:
            self.tail = self.head

    def print_all(self):
        tmp = self.head
        while tmp is not None:
            print(tmp.info, end=" ")
            tmp = tmp.next
        print()
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def insert_sorted(lst, el):
    if lst.is_empty() or el <= lst.head.info:
        lst.add_to_head(el)              # novi najmanji element
        return
    pred = lst.head
    while pred.next is not None and pred.next.info < el:
        pred = pred.next
    pred.next = SLLNode(el, pred.next)   # umetni iza pred
    if pred is lst.tail:
        lst.tail = pred.next

lista = SLList()
for x in [5, 1, 9, 3, 7, 3]:
    insert_sorted(lista, x)
    lista.print_all()
```

IZLAZ (OUTPUT):

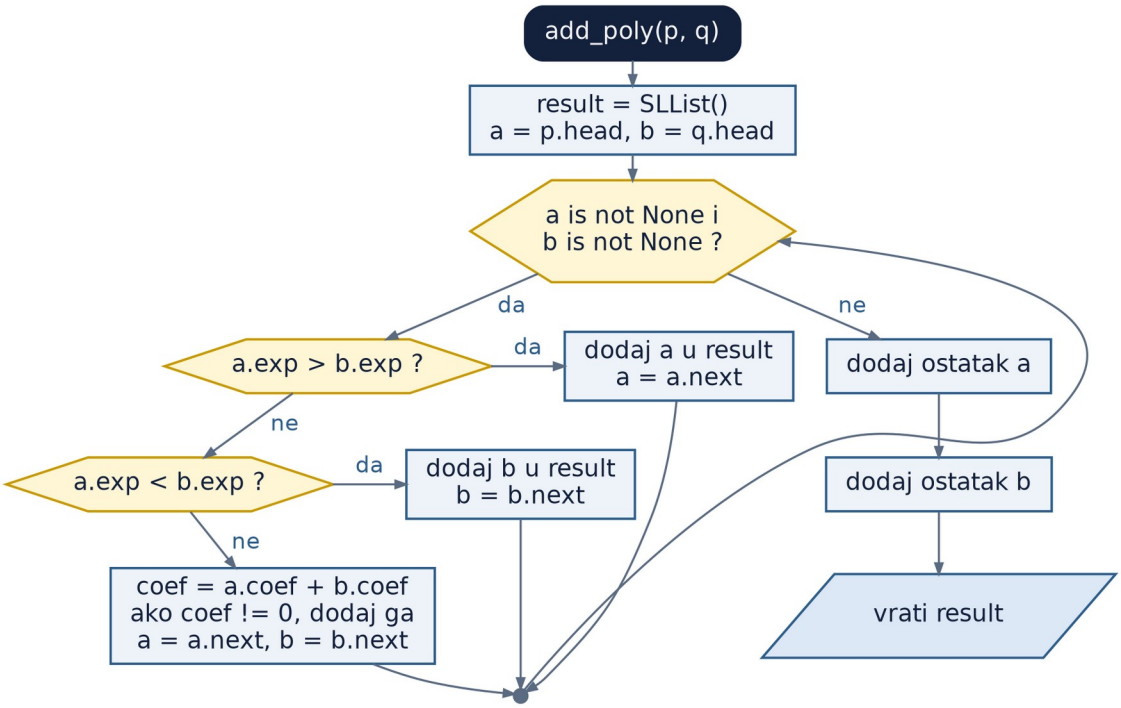
```
5
1 5
1 5 9
1 3 5 9
1 3 5 7 9
1 3 3 5 7 9
```

Objašnjenje: Ako je lista prazna ili je novi element najmanji, on ide na početak. Inače `pred` ide kroz listu dok je sledeći element manji od novog, pa se novi čvor umeće iza `pred`: dobija `next = pred.next`, a `pred.next` postaje novi čvor. Ako je `pred` bio poslednji, novi čvor postaje `tail`. Ovo je operacija *insertAfter*. Složenost je O(n), ali se, za razliku od polja, nijedan element ne pomera.

Zadatak 4: Sabiranje polinoma

Polinomi su predstavljeni listama čvorova `(eksponent, koeficijent)`, sortiranim po opadajućem eksponentu. Napisati funkciju `add_poly(p, q)` koja vraća novu listu sa zbirom polinoma, i funkciju za ispis polinoma.

Algoritam:



Slika 49. *Dijagram toka funkcije add_poly*

Rešenje:

1. Klase potrebne za rešenje. Čvor `SLLNode` i jednostruko ulančana lista `SLList` sa metodama `is_empty` i `add_to_tail`. Kod je isti kao u odeljku „Jednostruko ulančana lista u Python-u“, pa se program može pokrenuti samostalno:

```
class SLLNode:
    def __init__(self, info=None, next_node=None):
        self.info = info          # podatak
        self.next = next_node     # pokazivač na sledbenika

    def is_equal(self, el):
        return self.info == el


class SLList:
    def __init__(self):
        self.head = None          # pokazivač na početak liste
        self.tail = None          # pokazivač na kraj liste

    def is_empty(self):
        return self.head is None

    def add_to_tail(self, el):
        if not self.is_empty():
            self.tail.next = SLLNode(el)
            self.tail = self.tail.next
        else:
            self.head = self.tail = SLLNode(el)
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def add_poly(p, q):
    result = SLList()
    a = p.head
    b = q.head
    while a is not None and b is not None:
        if a.info[0] > b.info[0]:            # veći eksponent u p
            result.add_to_tail(a.info)
            a = a.next
        elif a.info[0] < b.info[0]:          # veći eksponent u q
            result.add_to_tail(b.info)
            b = b.next
        else:                                # isti eksponent
            coef = a.info[1] + b.info[1]
            if coef != 0:
                result.add_to_tail((a.info[0], coef))
            a = a.next
            b = b.next
    while a is not None:                     # ostatak p
        result.add_to_tail(a.info)
        a = a.next
    while b is not None:                     # ostatak q
        result.add_to_tail(b.info)
        b = b.next
    return result

def print_poly(lst):
    tmp = lst.head
    while tmp is not None:
        print("%+d*x^%d" % (tmp.info[1], tmp.info[0]), end=" ")
        tmp = tmp.next
    print()

def make_poly(terms):
    lst = SLList()
    for i in range(len(terms)):
        lst.add_to_tail(terms[i])
    return lst

p = make_poly([(5, 150), (2, -27), (1, 7), (0, 5)])
q = make_poly([(3, 4), (2, 27), (0, -2)])
print_poly(p)
print_poly(q)
print_poly(add_poly(p, q))
```

IZLAZ (OUTPUT):

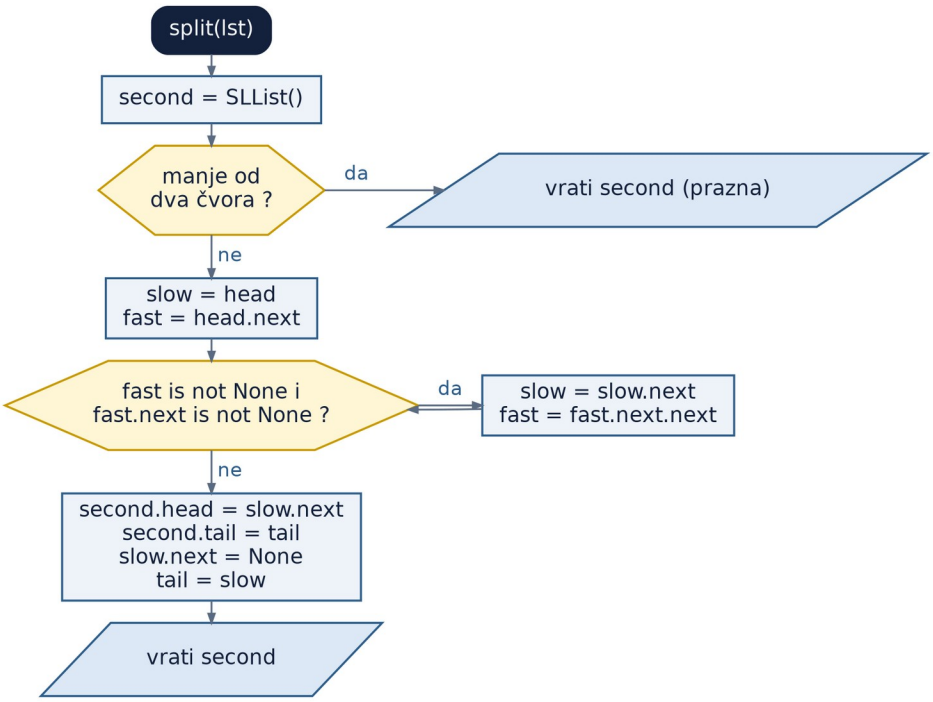
```
+150*x^5 -27*x^2 +7*x^1 +5*x^0
+4*x^3 +27*x^2 -2*x^0
+150*x^5 +4*x^3 +7*x^1 +3*x^0
```

> **Objašnjenje:** Pošto su obe liste sortirane po opadajućem eksponentu, sabiranje je spajanje dve sortirane liste, kao funkcija `merge` kod merge sort-a. Član sa većim eksponentom se prepisuje u rezultat. Kod jednakih eksponenata sabiraju se koeficijenti, a član se izostavlja ako je zbir nula: ovde $-27 \cdot x^2 + 27 \cdot x^2$ nestaje. Na kraju se prepisuje ostatak liste koja nije ispražnjena. Složenost je O(m + n), gde su m i n brojevi članova polinoma.

Zadatak 5: Cepanje i nadovezivanje lista (split i concatenate)

Napisati funkciju `split(lst)` koja cepa listu na dve polovine i vraća drugu polovinu kao novu listu (ako je broj čvorova neparan, prva polovina ima jedan čvor više), i funkciju `concatenate(first, second)` koja nadovezuje drugu listu na kraj prve.

Algoritam funkcije split:



Slika 50. *Dijagram toka funkcije split*

Rešenje:

1. Klase potrebne za rešenje. Čvor `SLLNode` i jednostruko ulančana lista `SLList` sa metodama `is_empty`, `add_to_tail` i `print_all`. Kod je isti kao u odeljku „Jednostruko ulančana lista u Python-u“, pa se program može pokrenuti samostalno:

```
class SLLNode:
    def __init__(self, info=None, next_node=None):
        self.info = info          # podatak
        self.next = next_node     # pokazivač na sledbenika

    def is_equal(self, el):
        return self.info == el


class SLList:
    def __init__(self):
        self.head = None          # pokazivač na početak liste
        self.tail = None          # pokazivač na kraj liste

    def is_empty(self):
        return self.head is None

    def add_to_tail(self, el):
        if not self.is_empty():
            self.tail.next = SLLNode(el)
            self.tail = self.tail.next
        else:
            self.head = self.tail = SLLNode(el)

    def print_all(self):
        tmp = self.head
        while tmp is not None:
            print(tmp.info, end=" ")
            tmp = tmp.next
        print()
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def split(lst):
    second = SLList()
    if lst.head is None or lst.head.next is None:
        return second                    # manje od dva čvora
    slow = lst.head
    fast = lst.head.next
    while fast is not None and fast.next is not None:
        slow = slow.next                 # jedan korak
        fast = fast.next.next            # dva koraka
    second.head = slow.next
    second.tail = lst.tail
    slow.next = None                     # preseci listu
    lst.tail = slow
    return second

def concatenate(first, second):
    if second.is_empty():
        return
    if first.is_empty():
        first.head = second.head
    else:
        first.tail.next = second.head    # poveži kraj prve sa početkom druge
    first.tail = second.tail
    second.head = second.tail = None     # druga lista ostaje prazna

lista = SLList()
for x in [1, 2, 3, 4, 5, 6, 7]:
    lista.add_to_tail(x)
druga = split(lista)
lista.print_all()
druga.print_all()

concatenate(lista, druga)
lista.print_all()
print(druga.is_empty(), lista.tail.info)
```

IZLAZ (OUTPUT):

```
1 2 3 4
5 6 7
1 2 3 4 5 6 7
True 7
```

Objašnjenje: Funkcija `split` koristi dva pokazivača koji idu različitom brzinom: `slow` za jedan, a `fast` za dva čvora. Kada `fast` stigne do kraja, `slow` je tačno na sredini, pa se dužina liste ne mora unapred brojati. Lista se cepa presecanjem pokazivača `slow.next`. Funkcija `concatenate` je O(1) jer zahvaljujući pokazivaču `tail` ne prolazi kroz prvu listu: dovoljno je povezati

> `first.tail.next` sa `second.head`. Kod liste bez `tail` ova operacija bi bila O(n).

POGLAVLJE 12

Lančane liste: vežbe (Linked Lists Exercises)

Ovo poglavlje prati vežbe o lančanim listama. Rešavamo zadatke u kojima se čvorovi liste **premeštaju** prevezivanjem pokazivača: zamenu mesta čvorovima, tri algoritma sortiranja lančane liste, umetanje u sortiranu listu, bucket sort i nalaženje preseka pravougaonika. Za svaki zadatak dati su tekst, skica rada na maloj listi, kôd u Python-u, dijagram toka i objašnjenje.

Sve funkcije su metode klase `SLList` iz prethodnog poglavlja (lista sa pokazivačima `head` i `tail`), proširene novim metodama. Kompletan kôd svih potrebnih klasa ponovljen je u rešenjima zadataka za vežbu na kraju poglavlja.

Kako čitamo skice

Svaka skica prikazuje listu kroz više stanja, jedno ispod drugog. Desno od svakog reda je naredba iz koda koja je napravila to stanje. Oznake na skicama su iste u celom poglavlju:

- **Čvor** je pravougaonik sa dva polja: levo je podatak `info`, desno pokazivač `next`. Kosa crta u polju pokazivača znači `None`.
- **Žuta strelica** je pokazivač koji se menja upravo u tom koraku.
- **Žuti čvorovi** su čvorovi koje u tom koraku poredimo ili premeštamo, a **plavi** čvorovi su deo liste koji je već sortiran.
- **Crvene oznake** iznad čvorova (`prev`, `ptr`, `ptr1`...) su pomoćni pokazivači iz koda.

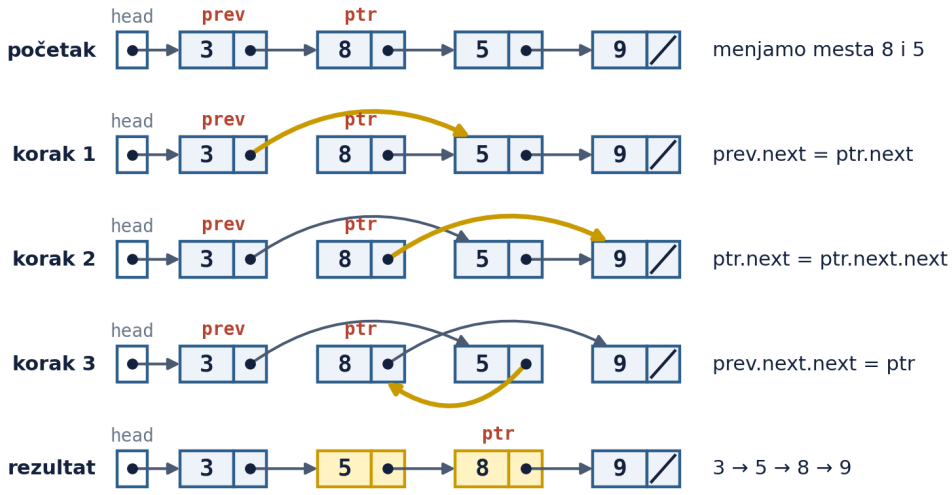
> **Napomena:** U svim zadacima menjamo **pokazivače**, a ne podatke u čvorovima. Zamena samo vrednosti `info` bila bi jednostavnija, ali u stvarnim programima podatak može biti veliki objekat, a drugi delovi programa mogu čuvati referencu na određeni čvor. Premeštanjem čvorova te reference ostaju ispravne.

Zamena mesta susedima u listi

Zadatak: napisati metodu `swap_neighbors(ptr, prev)` koja zamenjuje mesta čvoru `ptr` i njegovom sledbeniku. Parametar `prev` je prethodnik čvora `ptr`, ili `None` ako je `ptr` prvi čvor liste.

Pre zamene lista izgleda ovako: `prev → ptr → sled → ostatak`. Posle zamene treba da bude `prev → sled → ptr → ostatak`. Menjaju se tačno tri pokazivača: pokazivač prethodnika, pokazivač čvora `ptr` i pokazivač čvora `sled`.

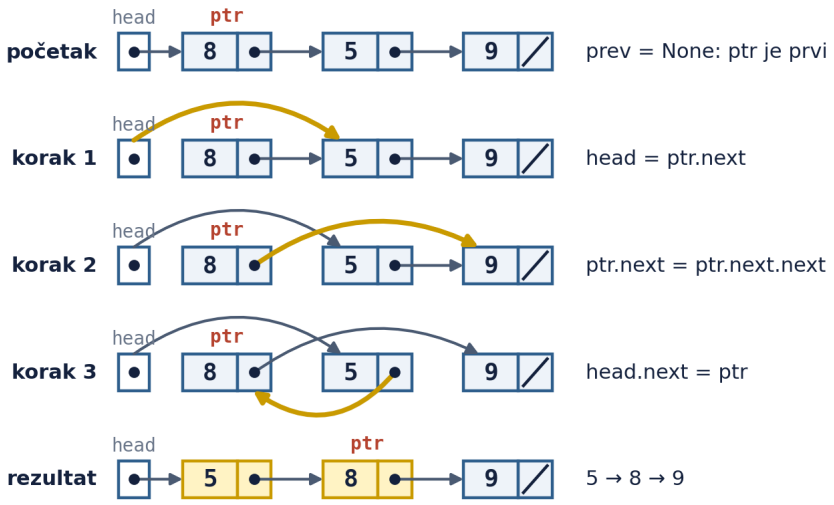
Slučaj kada prethodnik postoji



Slika 51. *Zamena susednih čvorova 8 i 5 kada prethodnik postoji*

1. `prev.next = ptr.next`: prethodnik (3) sada pokazuje na sledbenika (5). Čvor 8 je privremeno „izbačen“, ali ga i dalje imamo preko `ptr`.
2. `ptr.next = ptr.next.next`: čvor 8 pokazuje na 9, odnosno na ono na šta je pokazivao 5.
3. `prev.next.next = ptr`: `prev.next` je sada čvor 5, pa njegov `next` postavljamo na 8.

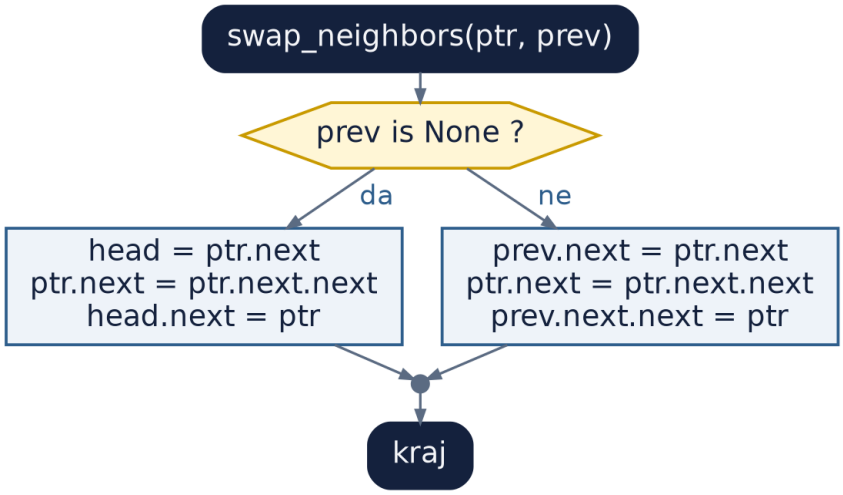
Slučaj kada je ptr prvi čvor



Slika 52. *Zamena prva dva čvora liste*

Kada `ptr` nema prethodnika, njegovu ulogu preuzima pokazivač `head`: umesto `prev.next` menja se `head`. Koraci su isti: `head = ptr.next`, `ptr.next = ptr.next.next` i `head.next = ptr`.

Kôd i dijagram toka



Slika 53. *Dijagram toka metode swap_neighbors*

```
    def swap_neighbors(self, ptr, prev):
        if prev is None:                    # ptr je prvi čvor
            self.head = ptr.next
            ptr.next = ptr.next.next
            self.head.next = ptr
        else:
            prev.next = ptr.next
            ptr.next = ptr.next.next
            prev.next.next = ptr

lista = SLList()
for x in [3, 8, 5, 9]:
    lista.add_to_tail(x)
prev = lista.head               # čvor 3
ptr = lista.head.next           # čvor 8
lista.swap_neighbors(ptr, prev)
lista.print_all()

lista.swap_neighbors(lista.head, None)   # zameni prva dva
lista.print_all()
```

IZLAZ (OUTPUT):

```
3 5 8 9
5 3 8 9
```

Objašnjenje: Metoda pretpostavlja da `ptr` ima sledbenika, jer se on menja sa `ptr`. Redosled naredbi je bitan: posle prvog koraka sledbenik je dostupan preko `prev.next` (odnosno `head`), pa ga treći korak može dohvatiti. Metoda ne menja `tail`; ako se zamenom poslednji čvor promeni, pozivalac ga ažurira metodom `fix_tail`.

```
    def fix_tail(self):
        tmp = self.head
        while tmp is not None and tmp.next is not None:
            tmp = tmp.next
        self.tail = tmp
```

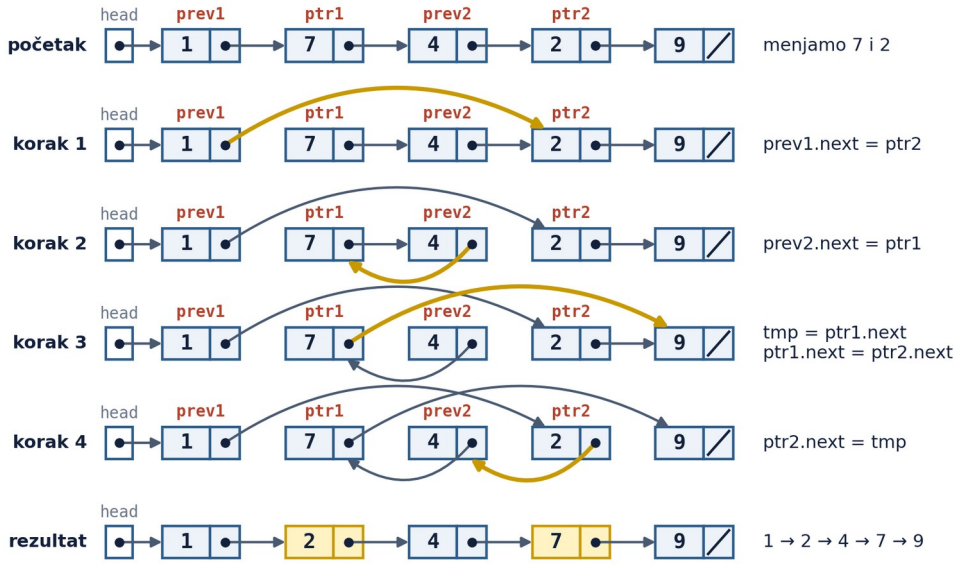
Pomoćna metoda `fix_tail` prolazi kroz listu i postavlja `tail` na poslednji čvor. Koriste je algoritmi sortiranja na kraju rada, jer tokom sortiranja čvorovi menjaju mesta.

Zamena mesta za dva elementa u listi

Zadatak: napisati metodu `swap(ptr1, prev1, ptr2, prev2)` koja zamenjuje mesta dvama proizvoljnim čvorovima liste. `prev1` i `prev2` su njihovi prethodnici (`None` za prvi čvor).

Postoje tri slučaja:

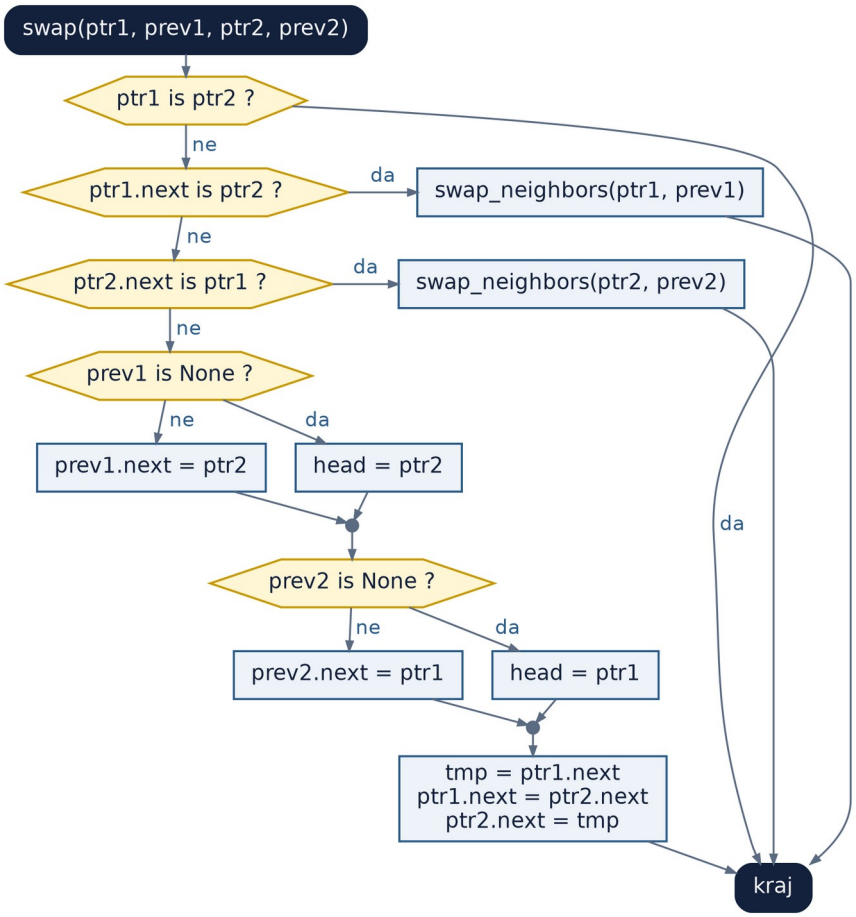
1. **Isti čvor** (`ptr1 is ptr2`): nema šta da se radi.
2. **Susedi** (jedan je odmah iza drugog): poziva se `swap_neighbors`. Ovaj slučaj se mora rešiti posebno, jer bi opšti postupak napravio čvor koji pokazuje sam na sebe.
3. **Čvorovi nisu susedi:** prvo se preusmere prethodnici, pa se zamene sledbenici.



Slika 54. *Zamena čvorova 7 i 2 koji nisu susedi*

U opštem slučaju (slika) koraci su:

1. `prev1.next = ptr2`: čvor ispred 7 sada pokazuje na 2 (ako `prev1` ne postoji, menja se `head`);
2. `prev2.next = ptr1`: čvor ispred 2 sada pokazuje na 7;
3. `tmp = ptr1.next` i `ptr1.next = ptr2.next`: čvor 7 dobija sledbenika koji je imao 2;
4. `ptr2.next = tmp`: čvor 2 dobija stari sledbenik od 7. Pomoćna promenljiva `tmp` je potrebna jer korak 3 prepisuje `ptr1.next`.



Slika 55. *Dijagram toka metode swap*

```
    def swap(self, ptr1, prev1, ptr2, prev2):
        if ptr1 is ptr2:
            return
        elif ptr1.next is ptr2:              # susedi: ptr1, pa ptr2
            self.swap_neighbors(ptr1, prev1)
        elif ptr2.next is ptr1:              # susedi: ptr2, pa ptr1
            self.swap_neighbors(ptr2, prev2)
        else:
            if prev1 is None: self.head = ptr2
            else: prev1.next = ptr2
            if prev2 is None: self.head = ptr1
            else: prev2.next = ptr1
            tmp = ptr1.next                  # zameni sledbenike
            ptr1.next = ptr2.next
            ptr2.next = tmp
```

```
lista = SLList()
for x in [1, 7, 4, 2, 9]:
    lista.add_to_tail(x)
n1 = lista.head             # 1
n7 = n1.next                # 7
n4 = n7.next                # 4
n2 = n4.next                # 2
lista.swap(n7, n1, n2, n4)
lista.print_all()
```

IZLAZ (OUTPUT):

```
1 2 4 7 9
```

Objašnjenje: Naredba `if uslov: naredba` u jednom redu je ispravan Python i ovde skraćuje kôd. Slučajevi suseda proveravaju se u oba smera (`ptr1.next is ptr2` i `ptr2.next is ptr1`), jer pozivalac ne mora znati koji je čvor prvi u listi.

Sortiranje lančane liste

Algoritmi sortiranja koje smo upoznali nad poljem rade i nad lančanom listom, ali uz četiri važne razlike:

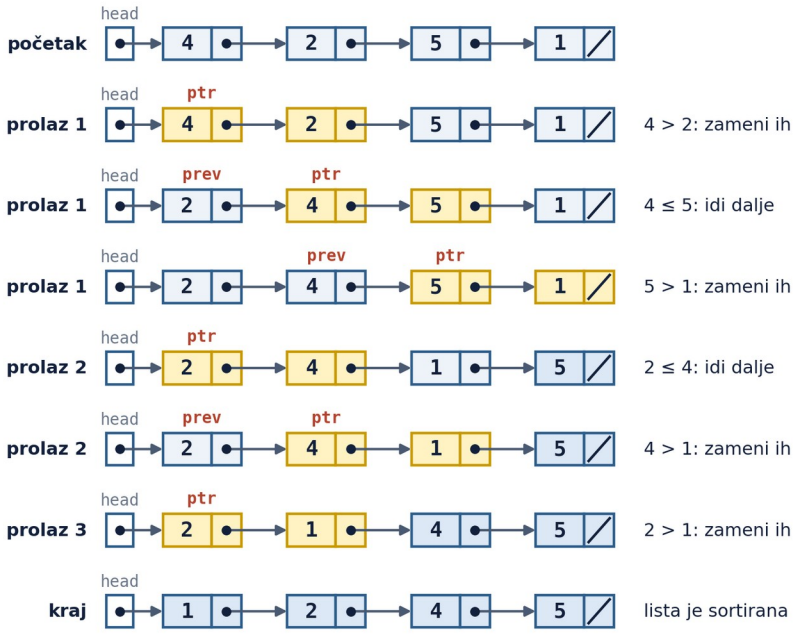
- **Nema indeksa.** Ne postoji `a[i]`; do čvora se stiže samo prolaskom od `head` preko pokazivača `next`.
- **Pomeraju se čvorovi.** Umesto zamene vrednosti koristimo `swap_neighbors` i `swap`.
- **Pamti se prethodnik.** Uz svaki pokazivač koji se kreće kroz listu nosimo i pokazivač na čvor ispred njega, jer je potreban za zamenu.
- **Na kraju se ažurira `tail`.** Posle premeštanja poslednji čvor se promenio, pa ga `fix_tail` ponovo nalazi.

Sva tri algoritma sortiraju rastuće i imaju složenost $O(n^2)$. U nastavku je za svaki prvo opisana ideja na običnom nizu brojeva, a zatim rad na listi.

Bubble sort

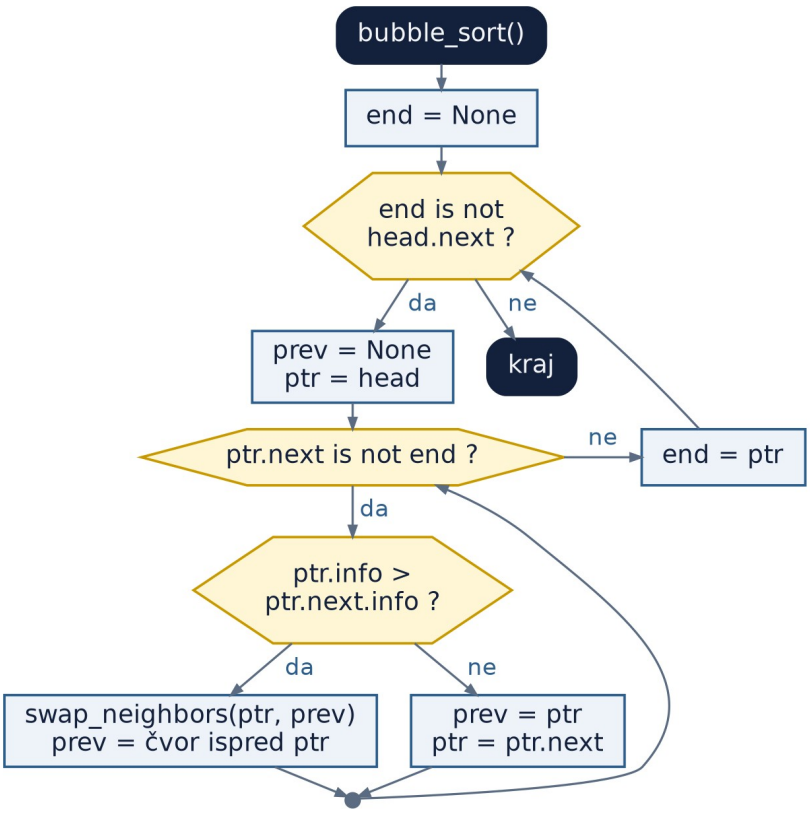
Ideja: prolazi se kroz listu i porede se dva susedna elementa. Ako je levi veći od desnog, oni menjaju mesta. Posle prvog prolaza najveći element se nalazi na kraju, jer je „isplivao“ kao mehur. Posle drugog prolaza na pretposlednjem mestu je drugi najveći, i tako dalje. Svaki sledeći prolaz je za jedan element kraći, jer se sortiran kraj više ne gleda.

Na nizu 4 2 5 1: prvi prolaz daje 2 4 1 5 (5 je na mestu), drugi daje 2 1 4 5, a treći 1 2 4 5.



Slika 56. *Bubble sort na listi 4, 2, 5, 1: svako poređenje u posebnom redu*

Pokazivač `ptr` je levi element para koji poredimo, a `prev` čvor ispred njega. Kada se `ptr` i `ptr.next` zamene, čvor `ptr` se pomerio jedno mesto napred zajedno sa zamenom, pa se on **ne pomera** dalje. Pomera se samo `prev`, na čvor koji je došao ispred `ptr`. Ako nije bilo zamene, pomeraju se oba pokazivača.



Slika 57. *Dijagram toka bubble sort-a na lančanoj listi*

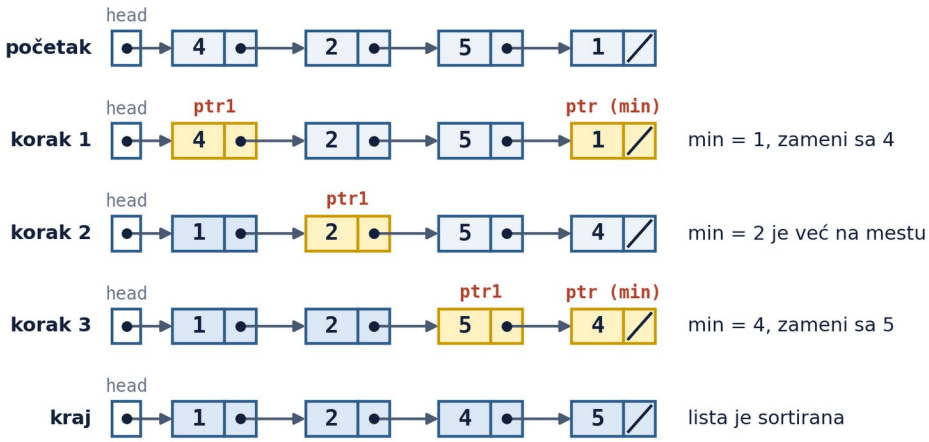
```
    def bubble_sort(self):
        if self.head is None or self.head.next is None: return
        end = None                          # od end nadalje je sortirano
        while end is not self.head.next:
            prev = None
            ptr = self.head
            while ptr.next is not end:
                if ptr.info > ptr.next.info:
                    self.swap_neighbors(ptr, prev)
                    if prev is None: prev = self.head
                    else: prev = prev.next
                else:
                    prev = ptr
                    ptr = ptr.next
            end = ptr                       # najveći je stigao na kraj
        self.fix_tail()
```

Objašnjenje: Promenljiva `end` pokazuje na prvi čvor sortiranog dela na kraju liste (na početku `None`, jer sortiranog dela još nema). Unutrašnja petlja ide dok `ptr.next` ne stigne do `end`, a posle nje je `ptr` najveći element nesortiranog dela, pa postaje novi `end`. Spoljna petlja se završava kada je `end` drugi čvor liste: tada je nesortiran samo prvi element, a on je najmanji.

Selection sort

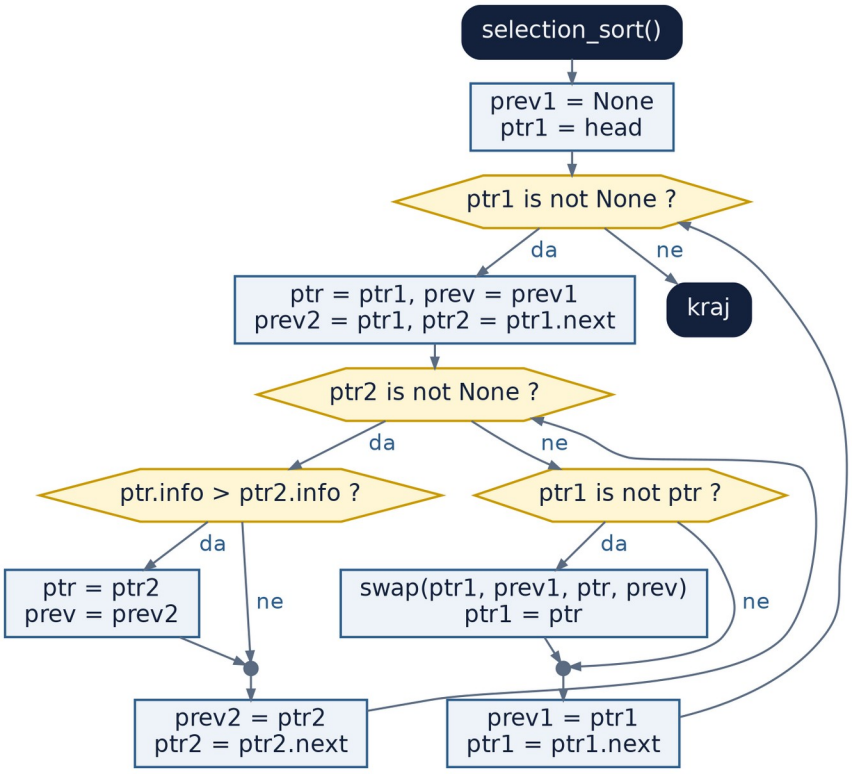
Ideja: u nesortiranom delu se pronađe najmanji element i stavi na prvo mesto tog dela. Zatim se ponovi za ostatak. Svaki korak radi samo jednu zamenu, pa je zamena najviše n − 1, ali poređenja ima n(n − 1)/2.

Na nizu 4 2 5 1: najmanji je 1 i menja mesto sa 4 (1 2 5 4); zatim je najmanji u ostatku 2, i on je već na mestu; zatim 4 menja mesto sa 5 (1 2 4 5).



Slika 58. *Selection sort na listi 4, 2, 5, 1*

`ptr1` pokazuje na mesto koje popunjavamo, a `ptr` na najmanji element pronađen u ostatku liste. Pošto najmanji element najčešće nije sused od `ptr1`, koristi se opšta metoda `swap`.



Slika 59. *Dijagram toka selection sort-a na lančanoj listi*

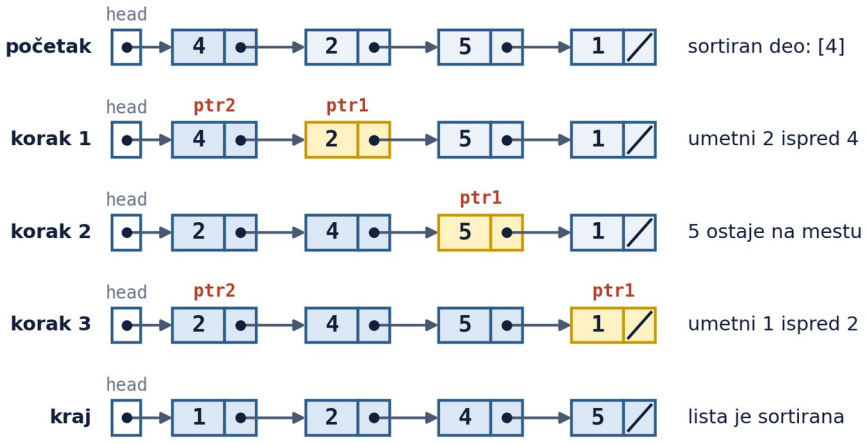
```
    def selection_sort(self):
        prev1, ptr1 = None, self.head
        while ptr1 is not None:
            ptr, prev = ptr1, prev1          # najmanji do sada
            prev2, ptr2 = ptr1, ptr1.next
            while ptr2 is not None:          # traži najmanji u ostatku
                if ptr.info > ptr2.info:
                    ptr, prev = ptr2, prev2
                prev2, ptr2 = ptr2, ptr2.next
            if ptr1 is not ptr:
                self.swap(ptr1, prev1, ptr, prev)
                ptr1 = ptr                   # najmanji je sada na mestu
            prev1, ptr1 = ptr1, ptr1.next
        self.fix_tail()
```

Objašnjenje: Pokazivači uvek idu u parovima (`ptr1`/`prev1`, `ptr`/`prev`, `ptr2`/`prev2`), jer `swap` traži prethodnike. Dodela `a, b = x, y` postavlja dve promenljive odjednom. Posle zamene najmanji čvor `ptr` je na mestu koje smo popunjavali, pa se `ptr1` postavlja na njega i od njega se ide dalje.

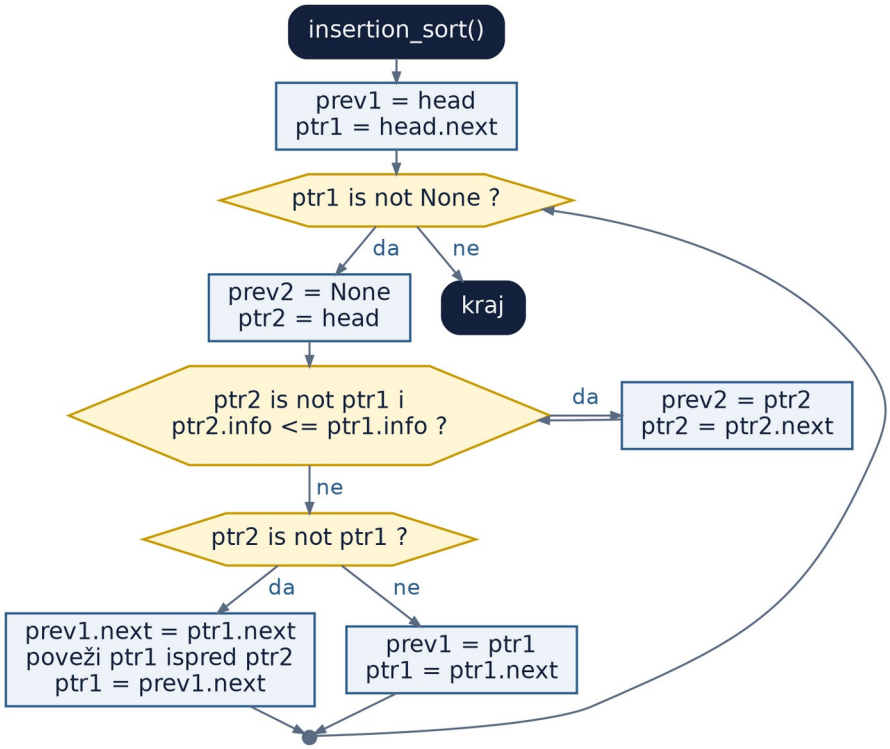
Insertion sort

Ideja: kao kada slažemo karte u ruci. Levo je sortiran deo; uzima se sledeći element i umeće na pravo mesto u sortiranom delu. Kod lančane liste umetanje je jeftino, jer se elementi ne pomeraju kao u polju: čvor se izvadi i prevežu se tri pokazivača.

Na nizu 4 2 5 1: 2 ide ispred 4 (2 4 5 1), 5 ostaje na mestu, a 1 ide na početak (1 2 4 5).



Slika 60. *Insertion sort na listi 4, 2, 5, 1*



Slika 61. *Dijagram toka insertion sort-a na lančanoj listi*

```
    def insertion_sort(self):
        if self.head is None: return
        prev1, ptr1 = self.head, self.head.next
        while ptr1 is not None:
            prev2, ptr2 = None, self.head
            while ptr2 is not ptr1 and ptr2.info <= ptr1.info:
                prev2, ptr2 = ptr2, ptr2.next    # traži mesto za ptr1
            if ptr2 is not ptr1:                 # umetni ptr1 ispred ptr2
                prev1.next = ptr1.next
                if prev2 is None: self.head = ptr1
                else: prev2.next = ptr1
                ptr1.next = ptr2
                ptr1 = prev1.next
            else:                                # ptr1 je već na mestu
                prev1, ptr1 = ptr1, ptr1.next
        self.fix_tail()
```

Objašnjenje: `ptr1` je čvor koji umećemo, a `prev1` čvor ispred njega. `ptr2` kreće od početka liste i traži prvi element veći od `ptr1`; ako stigne do samog `ptr1`, čvor je već na pravom mestu. Inače se `ptr1` izvadi (`prev1.next = ptr1.next`) i ubaci ispred `ptr2`. Posle umetanja `prev1` ostaje isti, pa je sledeći čvor za obradu `prev1.next`. Mesto se traži od početka liste, jer u jednostruko ulančanoj listi ne možemo da idemo unazad.

> **Napomena:** U originalnom C++ kodu sa vežbi uslov petlje je `ptr1->info < ptr2->info`, što sortira listu **opadajuće**. Ovde je uslov okrenut (`ptr2.info <= ptr1.info`) da bi sva tri algoritma sortirala rastuće. Znak `<=` čuva redosled jednakih elemenata, pa je sortiranje stabilno.

Primer korišćenja

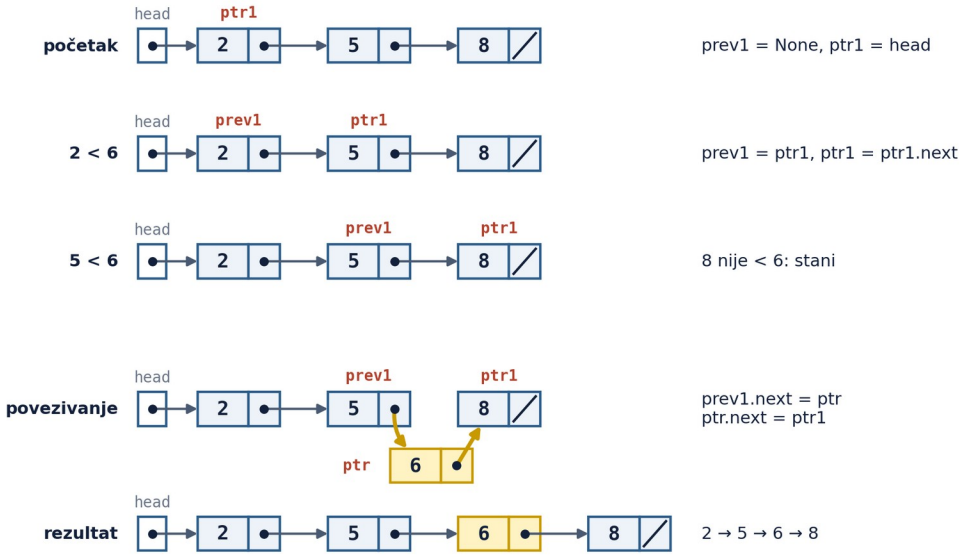
```
for metoda in ["bubble_sort", "selection_sort", "insertion_sort"]:
    lista = SLList()
    for x in [4, 2, 5, 1, 3, 2]:
        lista.add_to_tail(x)
    if metoda == "bubble_sort":
        lista.bubble_sort()
    elif metoda == "selection_sort":
        lista.selection_sort()
    else:
        lista.insertion_sort()
    print(metoda, end=": ")
    lista.print_all()
```

IZLAZ (OUTPUT):

```
bubble_sort: 1 2 2 3 4 5
selection_sort: 1 2 2 3 4 5
insertion_sort: 1 2 2 3 4 5
```

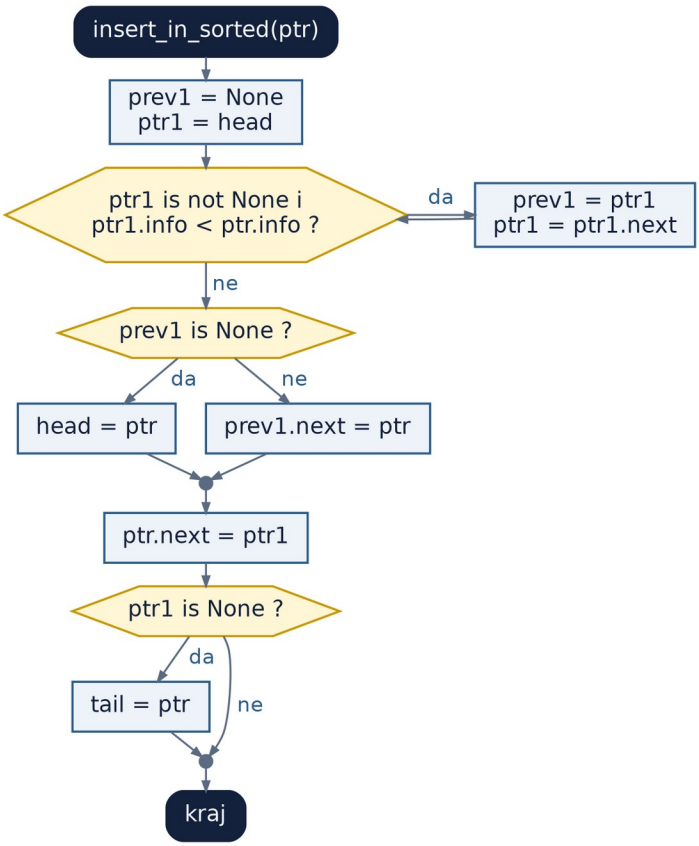
Umetanje u sortiranu listu

Zadatak: napisati metodu `insert_in_sorted(ptr)` koja čvor `ptr` umeće u rastuće sortiranu listu tako da lista ostane sortirana.



Slika 62. *Umetanje čvora 6 u sortiranu listu 2, 5, 8*

Lista se obilazi sa dva pokazivača dok je element manji od novog. Kada se petlja završi, `prev1` je poslednji manji element, a `ptr1` prvi veći ili jednak (ili `None` ako takvog nema). Novi čvor se umeće između njih: `prev1.next = ptr` i `ptr.next = ptr1`. Ako je `prev1` `None`, novi čvor je najmanji i postaje `head`; ako je `ptr1` `None`, novi čvor je najveći i postaje `tail`.



Slika 63. *Dijagram toka metode insert_in_sorted*

```
    def insert_in_sorted(self, ptr):
        prev1 = None
        ptr1 = self.head
        while ptr1 is not None and ptr1.info < ptr.info:
            prev1 = ptr1
            ptr1 = ptr1.next
        if prev1 is None:
            self.head = ptr
        else:
            prev1.next = ptr
        ptr.next = ptr1
        if ptr1 is None:
            self.tail = ptr
```

```
lista = SLList()
for x in [5, 2, 8, 6, 1, 9]:
    lista.insert_in_sorted(SLLNode(x))
    lista.print_all()
```

IZLAZ (OUTPUT):

```
5
2 5
2 5 8
2 5 6 8
1 2 5 6 8
1 2 5 6 8 9
```

Ova metoda je osnova za bucket sort i za presek pravougaonika. Poređenje `ptr1.info < ptr.info` radi za brojeve, ali i za bilo koje objekte koji definišu operator `<` (metodu `__lt__`), kao što ćemo videti kod pravougaonika.

> **Napomena:** I ovde je u originalnom C++ kodu uslov `ptr->info < ptr1->info`, koji daje opadajući redosled. Za bucket sort i presek pravougaonika potreban je rastući redosled, pa je uslov okrenut.

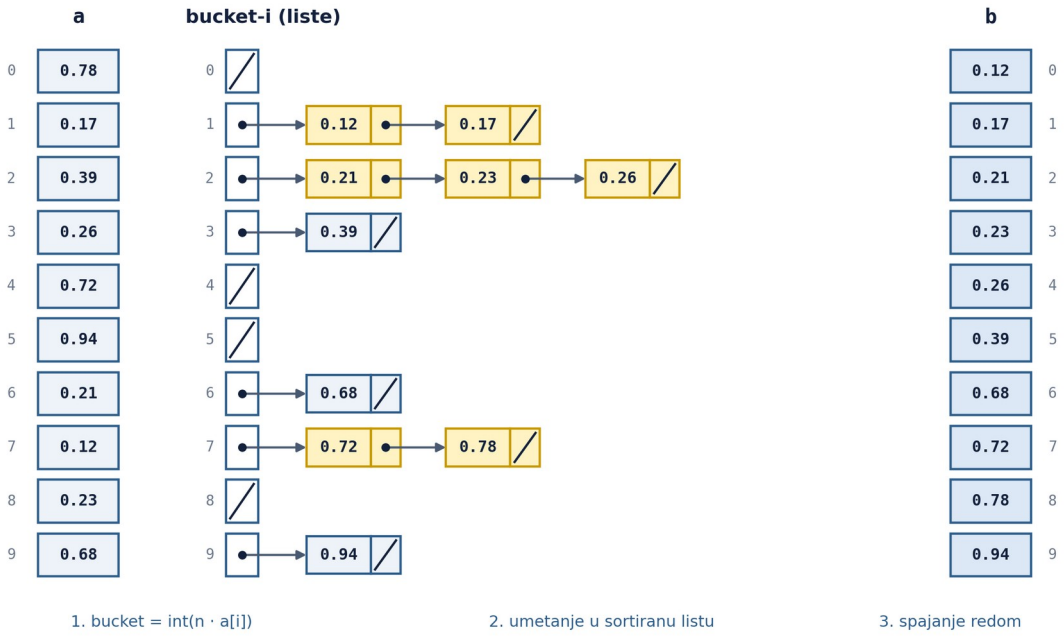
Bucket sort

Algoritmi sortiranja zasnovani na poređenju ne mogu biti brži od O(n log n). **Bucket sort** može biti linearan, O(n), ali samo uz dodatne pretpostavke o podacima:

- vrednosti elemenata su u opsegu [0, 1);
- elementi su **uniformno raspoređeni** u tom opsegu;
- potreban je dodatni memorijski prostor veličine n.

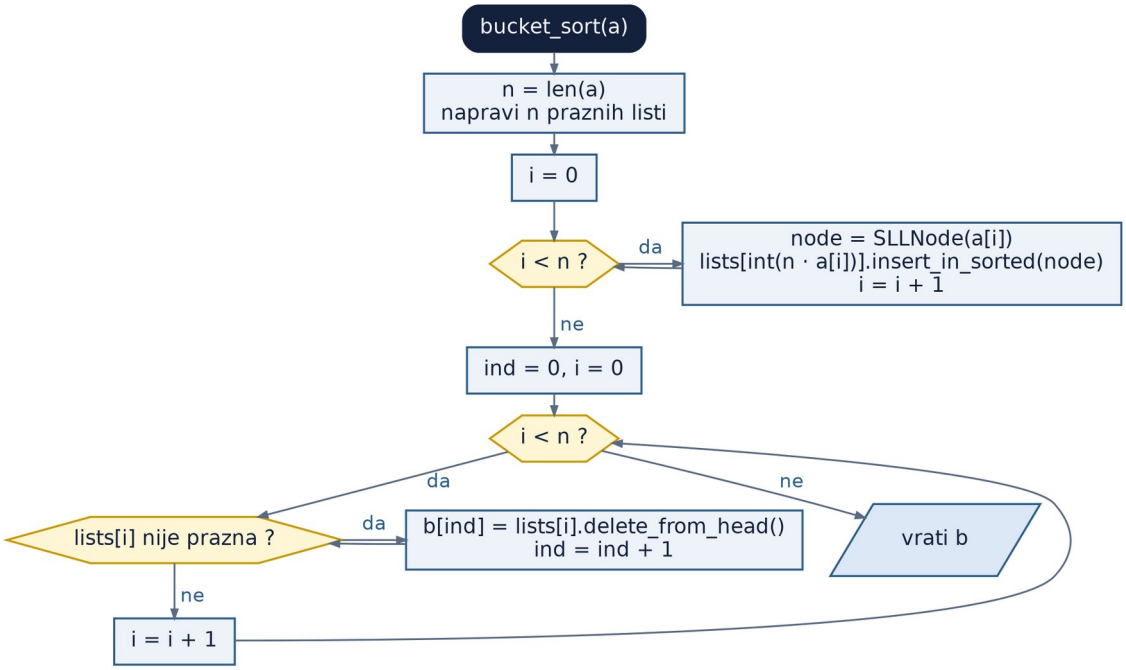
Ideja:

1. Opseg [0, 1) se podeli na n jednakih podopsega (engl. *bucket*, „kofa“). Bucket sa indeksom i sadrži vrednosti iz [i/n, (i+1)/n).
2. Svaki element se ubaci u svoj bucket. Indeks bucket-a za vrednost x je `int(n * x)`; na primer, za n = 10 vrednost 0.78 ide u bucket 7. Svaki bucket je lančana lista, a element se umeće metodom `insert_in_sorted`, pa je svaki bucket uvek sortiran.
3. Bucket-i se spoje redom, od 0 do n − 1, u uređen niz.



Slika 64. *Bucket sort za deset vrednosti: raspoređivanje u bucket-e i spajanje*

Na slici se vidi da su u bucket-u 2 vrednosti 0.26, 0.21 i 0.23, koje su umetnute redom, pa je lista 0.21 → 0.23 → 0.26. Na kraju se bucket-i prazne redom, uvek sa početka liste.



Slika 65. *Dijagram toka bucket sort-a*

```
def bucket_sort(a):
    n = len(a)
    lists = [None] * n
    for i in range(n):
        lists[i] = SLList()               # n praznih bucket-a
    for i in range(n):
        node = SLLNode(a[i])
        lists[int(n * a[i])].insert_in_sorted(node)
    b = [0.0] * n
    ind = 0
    for i in range(n):
        while not lists[i].is_empty():
            b[ind] = lists[i].delete_from_head()
            ind += 1
    return b

a = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]
print(bucket_sort(a))
```

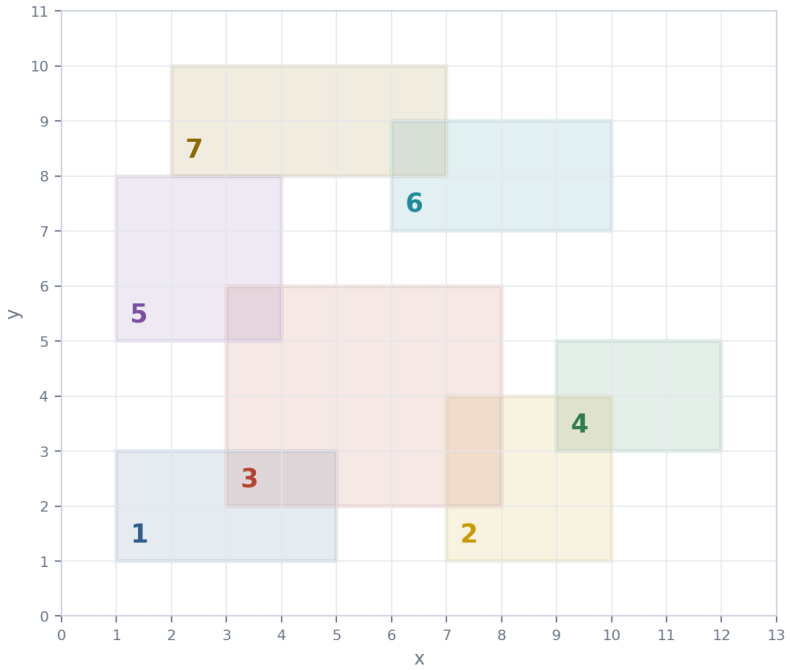
IZLAZ (OUTPUT):

```
[0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]
```

Objašnjenje: Zašto je složenost linearna: zbog uniformne raspodele, u svakom bucket-u se u proseku nalazi jedan element, pa je umetanje u sortiranu listu praktično O(1). Raspoređivanje n elemenata je zato O(n), a spajanje takođe O(n), jer se ukupno izvadi tačno n elemenata. Ugnježđena petlja u drugom delu ne menja ovu ocenu. Ako podaci nisu uniformni (na primer, svi su u [0, 0.1)), svi završe u istom bucket-u, pa se algoritam svodi na insertion sort i složenost postaje $O(n^2)$.

Presek uređenih pravougaonika

Zadatak: pravougaonici imaju ivice paralelne x i y osi i pamte se kao (xmin, ymin, xmax, ymax). Ulaz je niz pravougaonika uređen po minimalnoj y koordinati (ymin). Izlaz su svi parovi pravougaonika koji se seku.



Slika 66. *Sedam pravougaonika numerisanih po rastućem ymin*

Trivijalno rešenje upoređuje svaki pravougaonik sa svakim, što je n(n − 1)/2 poređenja. Bolje rešenje koristi uređenost ulaza da suzi skup pravougaonika sa kojima se novi poredi.

Ideja: lista aktivnih pravougaonika

- Pravougaonike obrađujemo redom po ymin, kao da horizontalna linija klizi odozdo nagore.
- Čuvamo **listu aktivnih** pravougaonika: onih koje je linija već prošla, a koji još mogu da seku nove. Lista je sortirana po **ymax**, rastuće.
- Kada stigne novi pravougaonik, sa početka liste se izbacuju svi čiji je ymax manji od njegovog ymin. Oni se završavaju ispod njega, pa ne mogu da seku ni njega ni bilo koji kasniji (jer kasniji imaju još veći ymin). Zato je dovoljno izbacivati sa početka.
- Svi preostali u listi preklapaju se sa novim po y: njihov ymin je manji ili jednak (obrađeni su ranije), a ymax je veći ili jednak ymin novog. Zato je dovoljno proveriti preklapanje po x.
- Novi pravougaonik se umeće u listu metodom `insert_in_sorted`.

Dva intervala $[a_1, b_1]$ i $[a_2, b_2]$ se preklapaju ako i samo ako je $a_1 \le b_2$ i $b_1 \ge a_2$. Za pravougaonike po x: `rc.xmin <= q.xmax and rc.xmax >= q.xmin`.

Skica rada

Korak	ymin	Izbačeni	Preseci	Lista posle umetanja: broj(ymax)
1	1	—	—	`1(3)`
2	1	—	—	`1(3) → 2(4)`
3	2	—	3–1, 3–2	`1(3) → 2(4) → 3(6)`
4	3	—	4–2	`1(3) → 2(4) → 4(5) → 3(6)`
5	5	1, 2	5–3	`4(5) → 3(6) → 5(8)`
6	7	4, 3	—	`5(8) → 6(9)`
7	8	—	7–5, 7–6	`5(8) → 6(9) → 7(10)`

> **Objašnjenje:** U koraku 5 novi pravougaonik ima ymin = 5. Pravougaonici 1 i 2 imaju ymax 3 i 4, pa se izbacuju sa početka liste. Od preostalih (4 i 3) samo se 3 preklapa po x, pa je pronađen par 5–3. U koraku 6 izbacuju se 4 i 3 (ymax 5 i 6 < 7). Ukupno je pronađeno šest parova, a broj poređenja je mnogo manji nego kod rešenja „svaki sa svakim“.

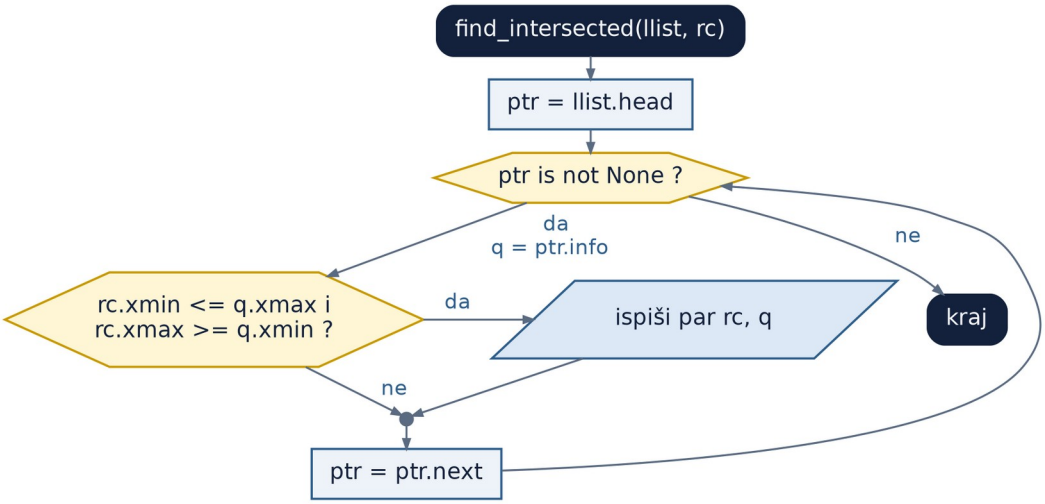
Klasa RectXY

```
class RectXY:
    def __init__(self, xmin=0, ymin=0, xmax=0, ymax=0):
        self.xmin = xmin
        self.ymin = ymin
        self.xmax = xmax
        self.ymax = ymax

    def __lt__(self, other):              # poređenje po ymax
        return self.ymax < other.ymax

    def __gt__(self, other):
        return self.ymax > other.ymax

    def __str__(self):
        r = (self.xmin, self.ymin, self.xmax, self.ymax)
        return "(%d %d %d %d)" % r
```

Specijalne metode zamenjuju C++ operatore: `__lt__` je operator `<`, a `__gt__` operator `>`, i oba porede pravougaonike po ymax. Zahvaljujući metodi `__lt__`, ista

metoda `insert_in_sorted` radi i sa pravougaonicima. Metoda `__str__` zamenjuje `Print()`: `print(rc)` ispisuje koordinate.

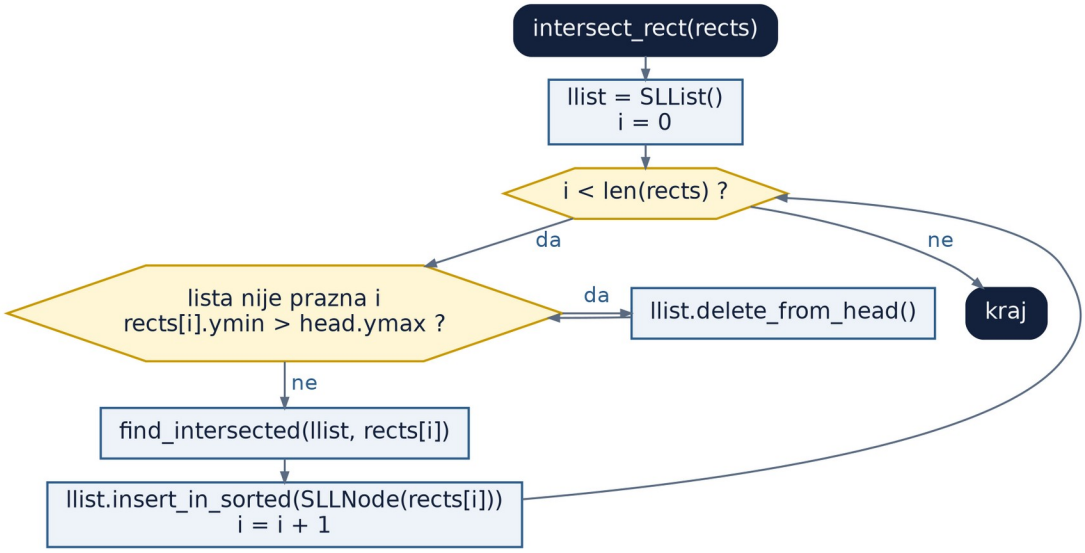
Traženje preseka u listi



Slika 67. *Dijagram toka funkcije find_intersected*

```
def find_intersected(llist, rc):
    ptr = llist.head
    while ptr is not None:
        rc_list = ptr.info
        if rc.xmin <= rc_list.xmax and rc.xmax >= rc_list.xmin:
            print(rc, rc_list)
        ptr = ptr.next
```

Glavna funkcija



Slika 68. *Dijagram toka funkcije intersect_rect*

```
def intersect_rect(rects):
    llist = SLList()
    for i in range(len(rects)):
        stop = False
        while not stop and not llist.is_empty():
            if rects[i].ymin > llist.get_head_el().ymax:
                llist.delete_from_head()  # ne može više da seče nijedan
            else:
                stop = True
        find_intersected(llist, rects[i])
        llist.insert_in_sorted(SLLNode(rects[i]))

rects = [RectXY(1, 1, 5, 3), RectXY(7, 1, 10, 4), RectXY(3, 2, 8, 6),
         RectXY(9, 3, 12, 5), RectXY(1, 5, 4, 8), RectXY(6, 7, 10, 9),
         RectXY(2, 8, 7, 10)]
intersect_rect(rects)
```

IZLAZ (OUTPUT):

```
(3 2 8 6) (1 1 5 3)
(3 2 8 6) (7 1 10 4)
(9 3 12 5) (7 1 10 4)
(1 5 4 8) (3 2 8 6)
(2 8 7 10) (1 5 4 8)
(2 8 7 10) (6 7 10 9)
```

Objašnjenje: Za svaki novi pravougaonik prvo se izbace oni koji su završeni, zatim se pronađu preseci sa preostalima, i tek onda se novi ubaci u listu.

> Redosled je važan: novi pravougaonik ne treba porediti sa samim sobom. Funkcija `get_head_el` vraća podatak iz prvog čvora liste.

```
    def get_head_el(self):
        return self.head.info
```

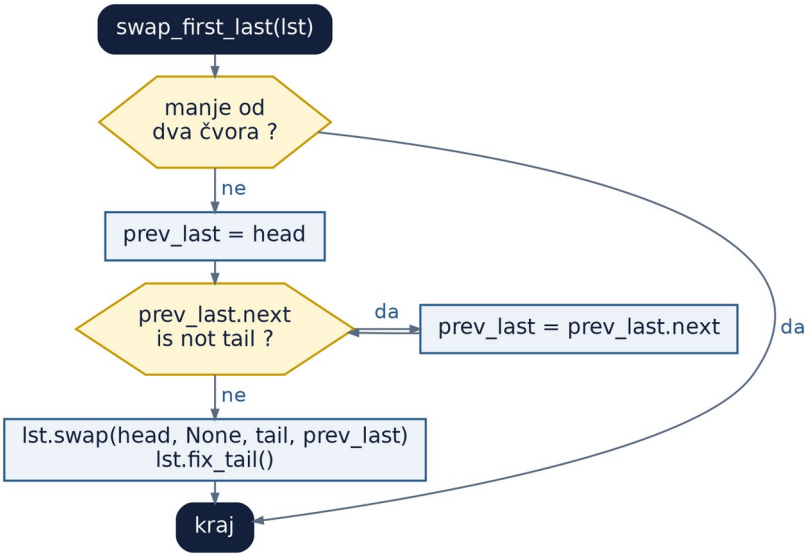
Zadaci za vežbu sa rešenjima

Svaki zadatak ima tekst, algoritam u obliku dijagrama toka, kompletno rešenje i objašnjenje. Rešenje je samostalan program: prvi deo sadrži implementaciju svih klasa koje su potrebne (čvor i lista sa potrebnim metodama), a drugi deo funkcije rešenja i primer korišćenja.

Zadatak 1: Zamena prvog i poslednjeg čvora

Napisati funkciju `swap_first_last(lst)` koja zamenjuje mesta prvom i poslednjem čvoru jednostruko ulančane liste, koristeći metodu `swap`. Lista sa manje od dva čvora ostaje nepromenjena.

Algoritam:



Slika 69. *Dijagram toka funkcije swap_first_last*

Rešenje:

1. Klase potrebne za rešenje: čvor `SLLNode` i lista `SLList` sa metodama `is_empty`, `add_to_tail`, `print_all`, `fix_tail`, `swap_neighbors`, `swap`. Kôd je isti kao u ovom i prethodnom poglavlju:

```
class SLLNode:
    def __init__(self, info=None, next_node=None):
        self.info = info          # podatak
        self.next = next_node     # pokazivač na sledbenika


class SLList:
    def __init__(self):
        self.head = None          # pokazivač na početak liste
        self.tail = None          # pokazivač na kraj liste

    def is_empty(self):
        return self.head is None

    def add_to_tail(self, el):
        if not self.is_empty():
            self.tail.next = SLLNode(el)
            self.tail = self.tail.next
        else:
            self.head = self.tail = SLLNode(el)

    def print_all(self):
        tmp = self.head
        while tmp is not None:
            print(tmp.info, end=" ")
            tmp = tmp.next
        print()

    def fix_tail(self):
        tmp = self.head
        while tmp is not None and tmp.next is not None:
            tmp = tmp.next
        self.tail = tmp

    def swap_neighbors(self, ptr, prev):
        if prev is None:                  # ptr je prvi čvor
            self.head = ptr.next
            ptr.next = ptr.next.next
            self.head.next = ptr
        else:
            prev.next = ptr.next
            ptr.next = ptr.next.next
            prev.next.next = ptr

    def swap(self, ptr1, prev1, ptr2, prev2):
        if ptr1 is ptr2:
            return
        elif ptr1.next is ptr2:              # susedi: ptr1, pa ptr2
            self.swap_neighbors(ptr1, prev1)
        elif ptr2.next is ptr1:              # susedi: ptr2, pa ptr1
            self.swap_neighbors(ptr2, prev2)
        else:
            if prev1 is None: self.head = ptr2
            else: prev1.next = ptr2
```

```
            if prev2 is None: self.head = ptr1
            else: prev2.next = ptr1
            tmp = ptr1.next                  # zameni sledbenike
            ptr1.next = ptr2.next
            ptr2.next = tmp
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def swap_first_last(lst):
    if lst.head is None or lst.head.next is None:
        return                               # manje od dva čvora
    prev_last = lst.head                     # traži pretposlednji čvor
    while prev_last.next is not lst.tail:
        prev_last = prev_last.next
    lst.swap(lst.head, None, lst.tail, prev_last)
    lst.fix_tail()

lista = SLList()
for x in [1, 7, 4, 2, 9]:
    lista.add_to_tail(x)
swap_first_last(lista)
lista.print_all()
print(lista.head.info, lista.tail.info)

dva = SLList()
dva.add_to_tail(3)
dva.add_to_tail(8)
swap_first_last(dva)
dva.print_all()
```

IZLAZ (OUTPUT):

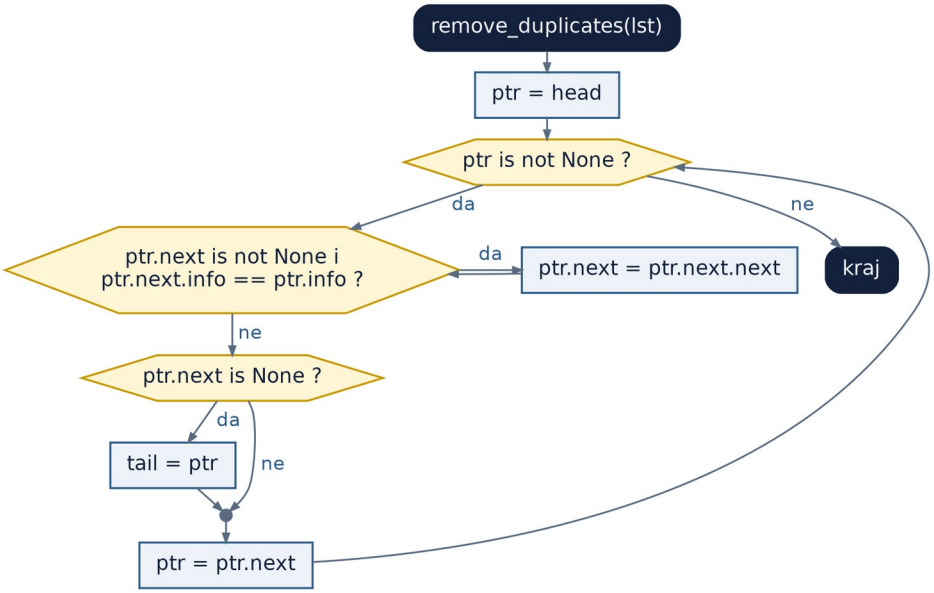
```
9 7 4 2 1
9 1
8 3
```

Objašnjenje: Metoda `swap` traži prethodnike oba čvora. Prvi čvor nema prethodnika (`None`), a prethodnik poslednjeg se nalazi prolaskom kroz listu. Ako lista ima samo dva čvora, prethodnik poslednjeg je prvi čvor, pa `swap` sam prepoznaje da su susedi i poziva `swap_neighbors`. Na kraju `fix_tail` postavlja `tail` na novi poslednji čvor. Složenost je O(n) zbog traženja pretposlednjeg čvora.

Zadatak 2: Uklanjanje duplikata iz sortirane liste

Napisati funkciju `remove_duplicates(lst)` koja iz rastuće sortirane liste uklanja ponovljene vrednosti, tako da svaka vrednost ostane tačno jednom.

Algoritam:



Slika 70. *Dijagram toka funkcije remove_duplicates*

Rešenje:

1. Klase potrebne za rešenje: čvor `SLLNode` i lista `SLList` sa metodama `is_empty`, `add_to_tail`, `print_all`. Kôd je isti kao u ovom i prethodnom poglavlju:

```
class SLLNode:
    def __init__(self, info=None, next_node=None):
        self.info = info          # podatak
        self.next = next_node     # pokazivač na sledbenika


class SLList:
    def __init__(self):
        self.head = None          # pokazivač na početak liste
        self.tail = None          # pokazivač na kraj liste

    def is_empty(self):
        return self.head is None

    def add_to_tail(self, el):
        if not self.is_empty():
            self.tail.next = SLLNode(el)
            self.tail = self.tail.next
        else:
            self.head = self.tail = SLLNode(el)

    def print_all(self):
        tmp = self.head
        while tmp is not None:
            print(tmp.info, end=" ")
            tmp = tmp.next
        print()
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def remove_duplicates(lst):
    ptr = lst.head
    while ptr is not None:
        while ptr.next is not None and ptr.next.info == ptr.info:
            ptr.next = ptr.next.next        # preskoči duplikat
        if ptr.next is None:
            lst.tail = ptr                  # ptr je sada poslednji
        ptr = ptr.next

lista = SLList()
for x in [1, 1, 2, 3, 3, 3, 5, 8, 8]:
    lista.add_to_tail(x)
remove_duplicates(lista)
lista.print_all()
print(lista.tail.info)
```

IZLAZ (OUTPUT):

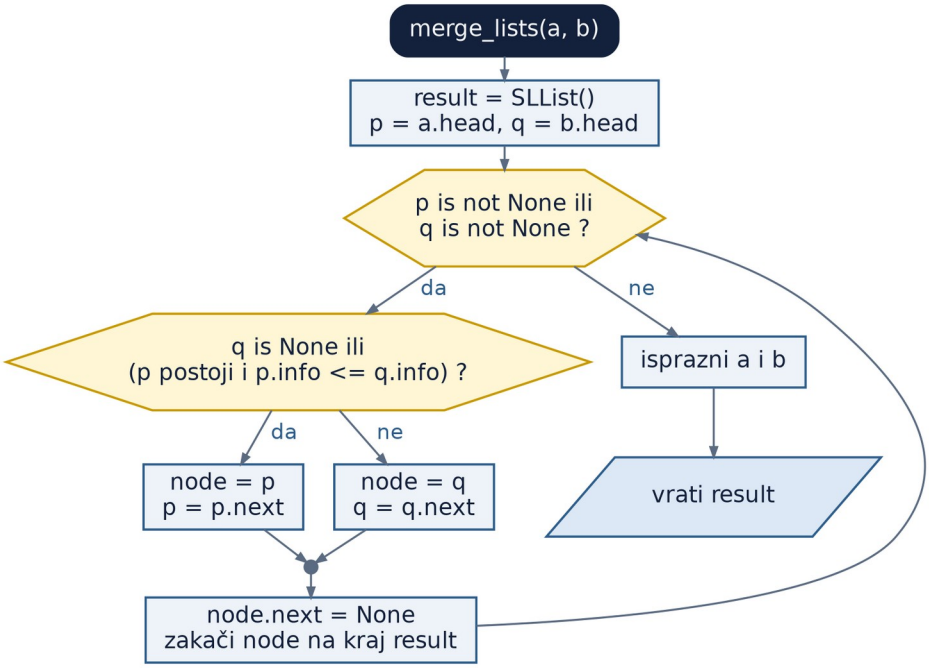
```
1 2 3 5 8
8
```

> **Objašnjenje:** U sortiranoj listi jednake vrednosti su jedna do druge. Zato je dovoljno za svaki čvor preskočiti sve sledbenike sa istom vrednošću (`ptr.next = ptr.next.next`). Unutrašnja petlja se ponavlja dok sledbenik ima istu vrednost, pa se uklanja i više uzastopnih duplikata. Ako je `ptr` postao poslednji čvor, ažurira se `tail`. Složenost je O(n), jer se svaki čvor obradi jednom.

Zadatak 3: Spajanje dve sortirane liste

Napisati funkciju `merge_lists(a, b)` koja od dve rastuće sortirane liste pravi jednu sortiranu listu, **bez pravljenja novih čvorova**: postojeći čvorovi se prevezuju. Posle spajanja liste `a` i `b` su prazne.

Algoritam:



Slika 71. *Dijagram toka funkcije merge_lists*

Rešenje:

1. Klase potrebne za rešenje: čvor `SLLNode` i lista `SLList` sa metodama `is_empty`, `add_to_tail`, `print_all`. Kôd je isti kao u ovom i prethodnom poglavlju:

```
class SLLNode:
    def __init__(self, info=None, next_node=None):
        self.info = info          # podatak
        self.next = next_node     # pokazivač na sledbenika


class SLList:
    def __init__(self):
        self.head = None          # pokazivač na početak liste
        self.tail = None          # pokazivač na kraj liste

    def is_empty(self):
        return self.head is None

    def add_to_tail(self, el):
        if not self.is_empty():
            self.tail.next = SLLNode(el)
            self.tail = self.tail.next
        else:
            self.head = self.tail = SLLNode(el)

    def print_all(self):
        tmp = self.head
        while tmp is not None:
            print(tmp.info, end=" ")
            tmp = tmp.next
        print()
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def merge_lists(a, b):
    result = SLList()
    p = a.head
    q = b.head
    while p is not None or q is not None:
        if q is None or (p is not None and p.info <= q.info):
            node = p                        # uzmi čvor iz prve liste
            p = p.next
        else:
            node = q                        # uzmi čvor iz druge liste
            q = q.next
        node.next = None                    # otkači ga od stare liste
        if result.is_empty():
            result.head = result.tail = node
        else:
            result.tail.next = node         # zakači ga na kraj rezultata
            result.tail = node
    a.head = a.tail = None                  # čvorovi sada pripadaju rezultatu
    b.head = b.tail = None
    return result

a = SLList()
for x in [1, 4, 6, 9]:
    a.add_to_tail(x)
b = SLList()
for x in [2, 3, 6, 10, 12]:
    b.add_to_tail(x)
c = merge_lists(a, b)
c.print_all()
print(a.is_empty(), b.is_empty(), c.tail.info)
```

IZLAZ (OUTPUT):

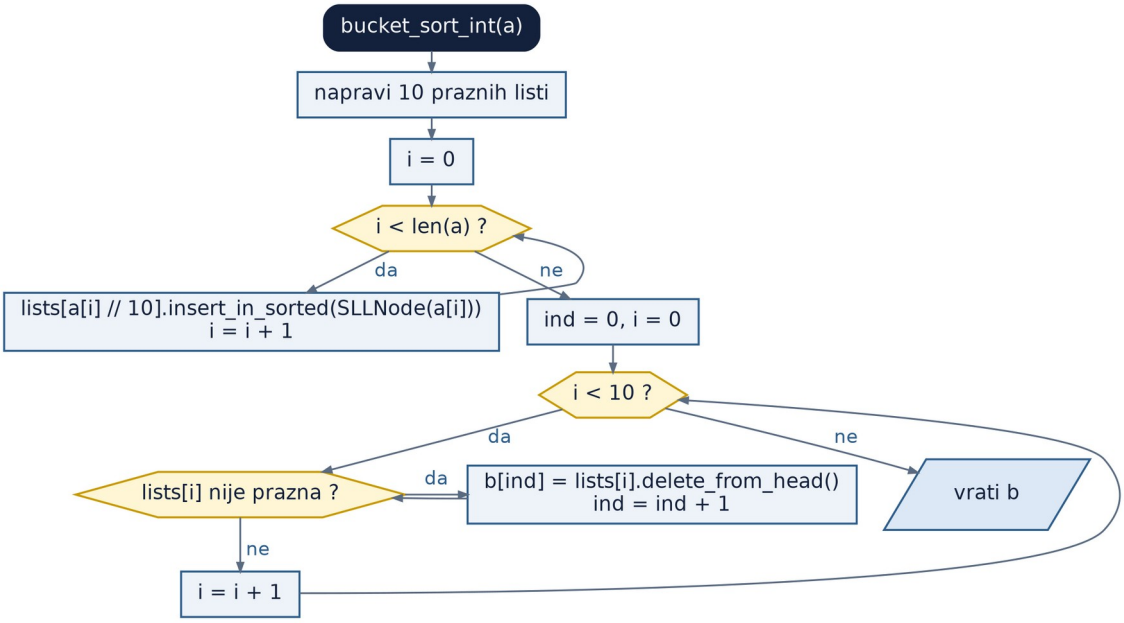
```
1 2 3 4 6 6 9 10 12
True True 12
```

Objašnjenje: Pokazivači `p` i `q` idu kroz prvu i drugu listu. U svakom koraku se uzima manji od dva čvora (iz prve liste ako su jednaki, pa je spajanje stabilno), otkači se od svoje liste i zakači na kraj rezultata preko `result.tail`. Kada se jedna lista isprazni, uslov `q is None` (ili `p is None`) uzima sve preostale čvorove druge. Pošto se ne prave novi čvorovi, dodatna memorija je O(1), a složenost O(m + n). Ovo je operacija `merge` iz merge sort-a, napisana za lančanu listu.

Zadatak 4: Bucket sort za cele brojeve

Napisati funkciju `bucket_sort_int(a)` koja sortira niz celih brojeva iz opsega [0, 100) (na primer, poene sa ispita) pomoću 10 bucket-a, po jedan za svaku deseticu.

Algoritam:



Slika 72. *Dijagram toka funkcije bucket_sort_int*

Rešenje:

1. Klase potrebne za rešenje: čvor `SLLNode` i lista `SLList` sa metodama `is_empty`, `delete_from_head`, `insert_in_sorted`. Kôd je isti kao u ovom i prethodnom poglavlju:

```
class SLLNode:
    def __init__(self, info=None, next_node=None):
        self.info = info          # podatak
        self.next = next_node     # pokazivač na sledbenika


class SLList:
    def __init__(self):
        self.head = None          # pokazivač na početak liste
        self.tail = None          # pokazivač na kraj liste

    def is_empty(self):
        return self.head is None

    def delete_from_head(self):
        if self.is_empty():
            raise IndexError("Lista je prazna!")
        el = self.head.info
        if self.head is self.tail:
            self.head = self.tail = None
        else:
            self.head = self.head.next
        return el

    def insert_in_sorted(self, ptr):
        prev1 = None
        ptr1 = self.head
        while ptr1 is not None and ptr1.info < ptr.info:
            prev1 = ptr1
            ptr1 = ptr1.next
        if prev1 is None:
            self.head = ptr
        else:
            prev1.next = ptr
        ptr.next = ptr1
        if ptr1 is None:
            self.tail = ptr
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def bucket_sort_int(a):
    k = 10                                  # broj bucket-a: desetice
    lists = [None] * k
    for i in range(k):
        lists[i] = SLList()
    for i in range(len(a)):
        lists[a[i] // 10].insert_in_sorted(SLLNode(a[i]))
    b = [0] * len(a)
    ind = 0
    for i in range(k):
        while not lists[i].is_empty():
            b[ind] = lists[i].delete_from_head()
            ind += 1
    return b

ocene = [73, 15, 98, 42, 7, 77, 45, 13, 64, 99, 70]
print(bucket_sort_int(ocene))
```

IZLAZ (OUTPUT):

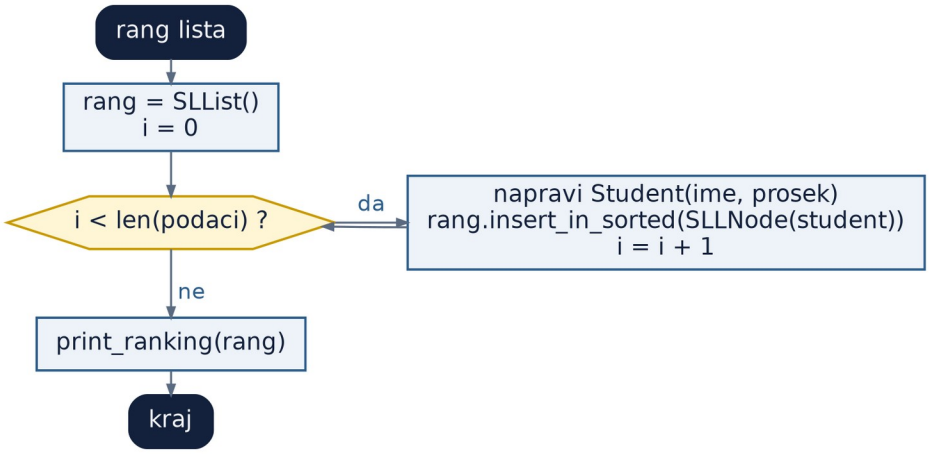
```
[7, 13, 15, 42, 45, 64, 70, 73, 77, 98, 99]
```

Objašnjenje: Indeks bucket-a je `a[i] // 10`, odnosno desetica broja: 73 ide u bucket 7, a 7 u bucket 0. Unutar bucket-a brojevi se umeću u sortiranu listu, pa je svaki bucket uređen. Spajanjem bucket-a od 0 do 9 dobija se sortiran niz. Ovo je isti algoritam kao bucket sort sa vežbi, samo je pravilo za izbor bucket-a prilagođeno celim brojevima.

Zadatak 5: Rang lista studenata

Napraviti klasu `Student` (ime i prosek) i rang listu studenata kao sortiranu lančanu listu, od najboljeg proseka ka najslabijem. Studenti se dodaju jedan po jedan metodom `insert_in_sorted`. Ispisati rang listu sa rednim brojevima.

Algoritam:



Slika 73. *Dijagram toka pravljenja rang liste*

Rešenje:

1. Klase potrebne za rešenje: čvor `SLLNode` i lista `SLList` sa metodama `is_empty`, `insert_in_sorted`. Kôd je isti kao u ovom i prethodnom poglavlju:

```
class SLLNode:
    def __init__(self, info=None, next_node=None):
        self.info = info          # podatak
        self.next = next_node     # pokazivač na sledbenika


class SLList:
    def __init__(self):
        self.head = None          # pokazivač na početak liste
        self.tail = None          # pokazivač na kraj liste

    def is_empty(self):
        return self.head is None

    def insert_in_sorted(self, ptr):
        prev1 = None
        ptr1 = self.head
        while ptr1 is not None and ptr1.info < ptr.info:
            prev1 = ptr1
            ptr1 = ptr1.next
        if prev1 is None:
            self.head = ptr
        else:
            prev1.next = ptr
        ptr.next = ptr1
        if ptr1 is None:
            self.tail = ptr
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
class Student:
    def __init__(self, ime, prosek):
        self.ime = ime
        self.prosek = prosek

    def __lt__(self, other):                # "manji" je onaj sa VEĆIM
prosekom,
        return self.prosek > other.prosek   # pa bolji studenti idu na početak

    def __str__(self):
        return "%-8s %.2f" % (self.ime, self.prosek)

def print_ranking(lst):
    rank = 1
    ptr = lst.head
    while ptr is not None:
        print(rank, ptr.info)
        rank += 1
        ptr = ptr.next

rang = SLList()
podaci = [("Ana", 8.75), ("Marko", 9.40), ("Jovana", 7.90),
          ("Nikola", 9.40), ("Sara", 8.10)]
for i in range(len(podaci)):
    ime, prosek = podaci[i]
    rang.insert_in_sorted(SLLNode(Student(ime, prosek)))
print_ranking(rang)
```

IZLAZ (OUTPUT):

```
1 Nikola   9.40
2 Marko    9.40
3 Ana      8.75
4 Sara     8.10
5 Jovana   7.90
```

Objašnjenje: Metoda `insert_in_sorted` poredi čvorove operatorom `<`. Zato je dovoljno da klasa `Student` definiše `__lt__` tako da je „manji“ student onaj sa **većim** prosekom: takvi studenti dolaze na početak liste. Pri istom proseku kasnije unet student dolazi ispred ranije unetog (Nikola ispred Marka), jer `insert_in_sorted` umeće ispred prvog elementa koji nije manji. Za stabilan redosled pri jednakim prosecima uslov u `insert_in_sorted` treba promeniti u `ptr1.info <= ptr.info`, odnosno u `not (ptr.info < ptr1.info)`.

POGLAVLJE 13

Stek i red (Stacks and Queues)

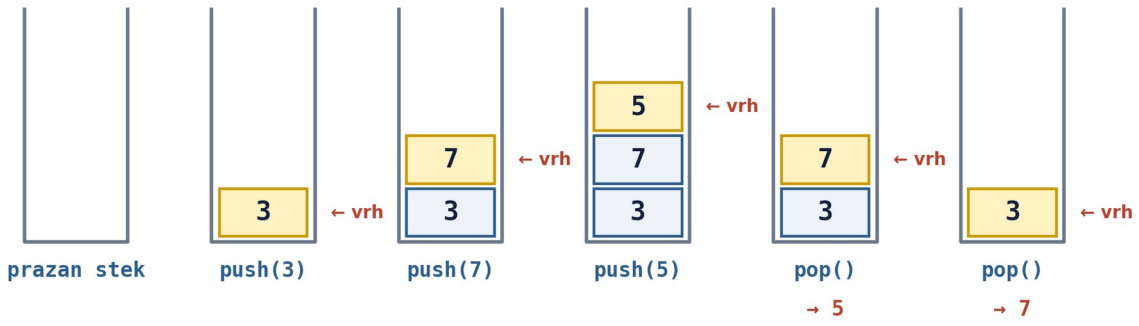
Stek (magacin) i **red** su dve najčešće korišćene strukture podataka. Obe su zapravo ograničene liste: elementi se ne mogu dodavati i uzimati bilo gde, već samo na tačno određenim mestima. Upravo zbog tih ograničenja su korisne, jer prirodno opisuju mnoge situacije: poništavanje izmena, vraćanje unazad kroz lavirint, čekanje na štampač ili procesor.

U ovom poglavlju prvo obnavljamo ideju i operacije obe strukture, zatim pravimo sopstvene klase nad lančanom listom, pa gledamo šta Python već nudi. Na kraju rešavamo dva primera sa vežbi: put kroz lavirint (stek) i izvršavanje poslova (red), i pet zadataka za vežbu sa potpunim rešenjima.

Stek (magacin)

Stek je struktura u kojoj se elementi dodaju i uzimaju **samo na jednom kraju**, koji se zove **vrh** (engl. *top*). Element koji je poslednji stavljen prvi se uzima. Ovo pravilo se zove **LIFO** (engl. *last in, first out*: poslednji unutra, prvi napolje).

Najbolja slika steka je gomila tanjira: novi tanjir se stavlja na vrh, a uzima se uvek onaj sa vrha. Do tanjira na dnu ne može se doći dok se ne sklone svi iznad njega.



Slika 74. *Stek kroz niz operacija: tri push operacije, pa dve pop operacije*

Na slici se na prazan stek redom stavljaju 3, 7 i 5. Prvi `pop` vraća 5, jer je poslednji stavljen, a drugi vraća 7. Na steku ostaje samo 3.

Operacije nad stekom

Operacija	Šta radi	Složenost
`push(el)`	stavlja element na vrh steka	O(1)
`pop()`	skida element sa vrha i vraća ga	O(1)
`get_top()`	vraća element sa vrha, ali ga ne skida	O(1)
`is_empty()`	proverava da li je stek prazan	O(1)

Pozivanje `pop` ili `get_top` nad praznim stekom je greška, pa se pre toga proverava `is_empty`. Elementima ispod vrha nije moguće pristupiti direktno.

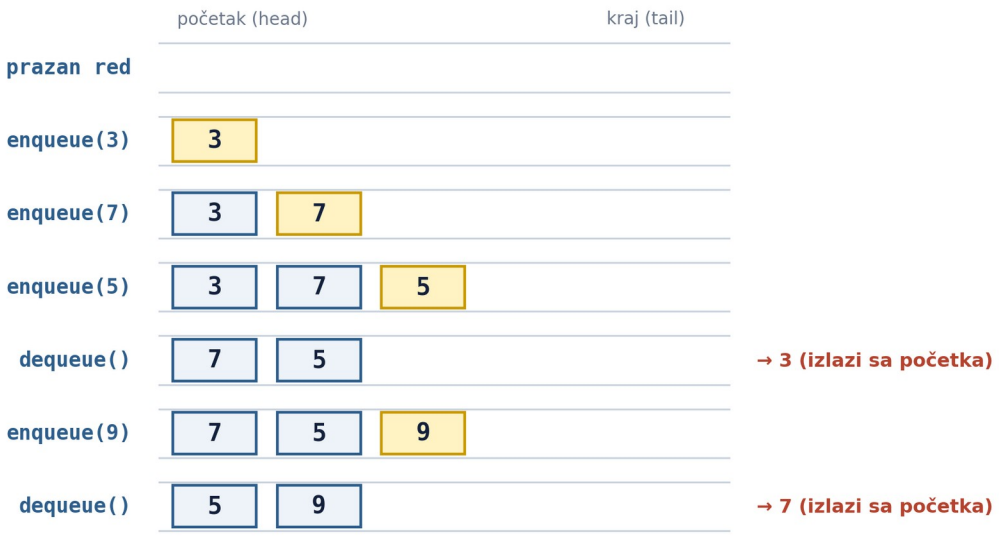
Gde se stek koristi

- **Pozivi funkcija:** kada funkcija pozove drugu funkciju, računar na stek stavlja mesto na koje treba da se vrati. Kada se pozvana funkcija završi, sa vrha uzima adresu povratka. Odatle i naziv greške *stack overflow* kod beskonačne rekurzije.
- **Poništavanje izmena (Ctrl + Z)** i dugme **„Nazad“** u pregledaču: poslednja akcija se prva poništava.
- **Provera zagrada** u izrazu i **izračunavanje izraza** (zadaci 1 i 2 za vežbu).
- **Vraćanje unazad** (engl. *backtracking*): traženje izlaza iz lavirinta, rešavanje sudoku-a i slično.

Red

Red je struktura u kojoj se elementi **dodaju na jednom kraju**, a **uzimaju na drugom**. Element koji je prvi stavljen prvi se i uzima. Ovo pravilo se zove **FIFO** (engl. *first in, first out*: prvi unutra, prvi napolje). Mesto sa koga se uzima je **početak** reda (engl. *head* ili *front*), a mesto na koje se dodaje je **kraj** reda (engl. *tail* ili *rear*).

Slika reda je red na kasi u prodavnici: ko je prvi došao, prvi je uslužen, a novi kupci staju na kraj.



Slika 75. *Red kroz niz operacija enqueue i dequeue*

Operacije nad redom

Operacija	Šta radi	Složenost
`enqueue(el)`	dodaje element na kraj reda	O(1)
`dequeue()`	uzima element sa početka reda i vraća ga	O(1)
`front()`	vraća prvi element, ali ga ne uzima	O(1)
`is_empty()`	proverava da li je red prazan	O(1)

Gde se red koristi

- **Štampač** štampa dokumente redom kojim su poslati.
- **Operativni sistem** deli procesor između programa koji čekaju u redu (primer sa izvršavanjem poslova u ovom poglavlju).
- **Obrada zahteva** na serveru, poruka u aplikacijama, poziva u call centru.
- **Pretraga u širinu** (BFS): nalaženje najkraćeg puta kroz lavirint ili mrežu (zadatak 5 za vežbu).

Kako napraviti stek i red

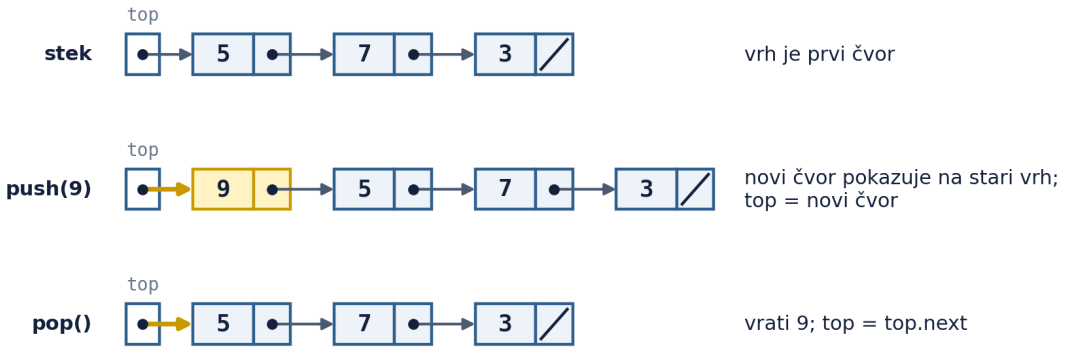
Stek i red se mogu napraviti pomoću polja ili pomoću lančane liste:

- **Pomoću polja:** stek je jednostavan, jer se dodaje i uzima na kraju polja, za O(1). Kod reda je problem uzimanje sa početka: ako se svi elementi pomeraju za jedno mesto, to je O(n). Rešenje je kružni bafer, polje u kome se početak i kraj „vrte u krug“, ali je tada veličina reda ograničena.
- **Pomoću lančane liste:** sve operacije su O(1), a veličina nije ograničena. Stek je lista u kojoj se koriste samo dodavanje i brisanje na početku. Red je lista sa pokazivačima `head` i `tail`, u kojoj se dodaje na kraj, a briše sa početka.

Klase `DStack` i `DQueue` iz originalnih vežbi su **dinamičke** (otuda slovo D), napravljene nad lančanom listom. Isto radimo i mi, koristeći znanje o čvorovima iz prethodnih poglavlja.

Stek nad lančanom listom

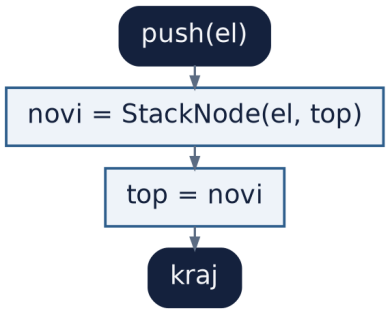
Vrh steka je prvi čvor liste, a pokazivač na njega zove se `top`. Operacija `push` je dodavanje čvora na početak liste, a `pop` brisanje čvora sa početka. Pokazivač na kraj liste nije potreban.



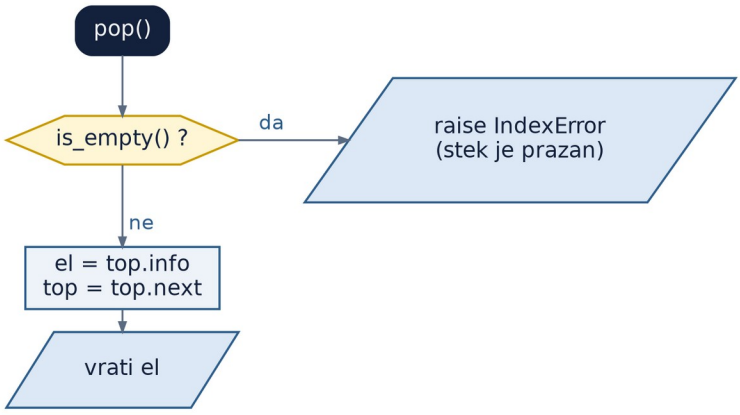
Slika 76. *Stek kao lančana lista: push dodaje čvor na početak, pop ga uklanja*

```
class StackNode:
    def __init__(self, info, next_node=None):
        self.info = info
        self.next = next_node     # čvor ispod u steku


class DStack:
    def __init__(self):
        self.top = None           # pokazivač na vrh steka

    def is_empty(self):
        return self.top is None

    def push(self, el):
        self.top = StackNode(el, self.top)

    def pop(self):
        if self.is_empty():
            raise IndexError("Stek je prazan!")
        el = self.top.info
        self.top = self.top.next
        return el

    def get_top(self):
        if self.is_empty():
            raise IndexError("Stek je prazan!")
        return self.top.info
```

Dijagrami toka operacija `push` i `pop`:



Slika 77. *Dijagram toka metode push*



Slika 78. *Dijagram toka metode pop*

```
s = DStack()
for x in [3, 7, 5]:
    s.push(x)
print(s.get_top())
print(s.pop(), s.pop())
print(s.is_empty(), s.pop(), s.is_empty())
```

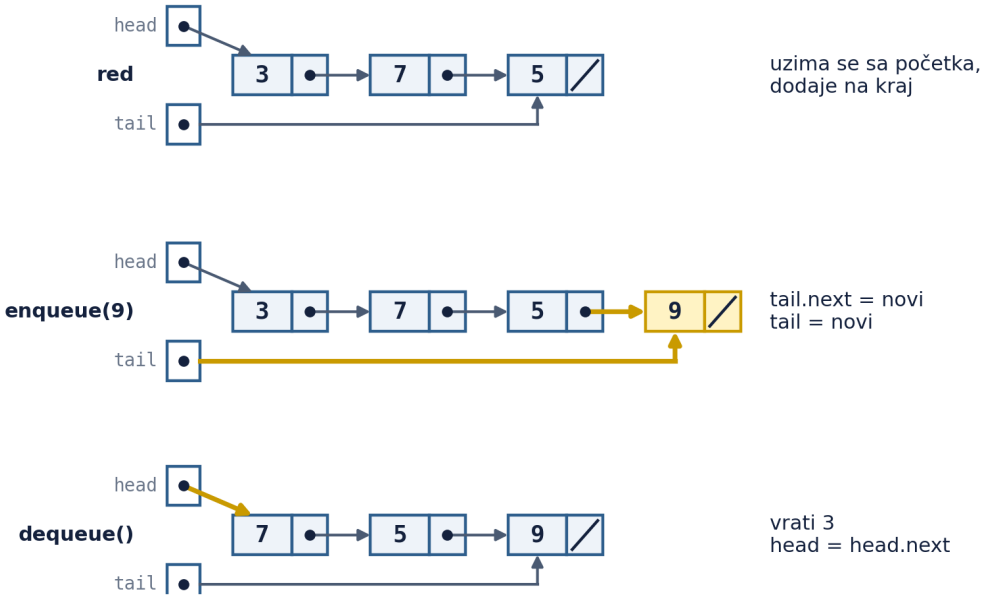
IZLAZ (OUTPUT):

```
5
5 7
False 3 True
```

Objašnjenje: `push` pravi novi čvor čiji je `next` dosadašnji vrh, i taj čvor postaje novi vrh: `self.top = StackNode(el, self.top)`. Desna strana se izračuna pre dodele, pa `self.top` unutar zagrade još znači stari vrh. `pop` pamti podatak iz vrha i pomera `top` na čvor ispod njega; stari vrh briše sakupljač smeća. Obe operacije imaju složenost O(1), bez obzira na veličinu steka.

Red nad lančanom listom

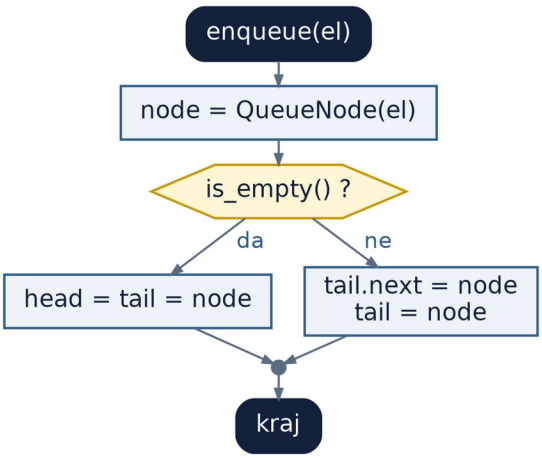
Red čuva dva pokazivača: `head` na prvi čvor, sa koga se uzima, i `tail` na poslednji čvor, na koji se dodaje.



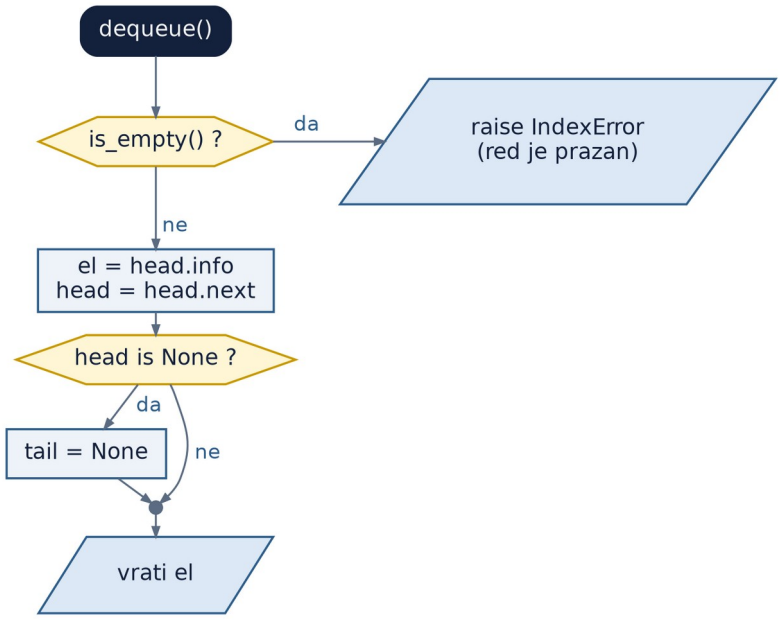
Slika 79. *Red kao lančana lista sa pokazivačima head i tail*

Zašto se dodaje na kraj, a uzima sa početka, a ne obrnuto? Brisanje sa kraja jednostruko ulančane liste je O(n), jer treba naći pretposlednji čvor. Dodavanje na kraj (preko `tail`) i brisanje sa početka su oba O(1).

```
class QueueNode:
    def __init__(self, info):
        self.info = info
        self.next = None          # čvor iza u redu


class DQueue:
    def __init__(self):
        self.head = None          # početak reda: odavde se uzima
        self.tail = None          # kraj reda: ovde se dodaje

    def is_empty(self):
        return self.head is None

    def enqueue(self, el):
        node = QueueNode(el)
        if self.is_empty():
            self.head = self.tail = node
        else:
            self.tail.next = node
            self.tail = node

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Red je prazan!")
        el = self.head.info
        self.head = self.head.next
        if self.head is None:     # red je ostao prazan
            self.tail = None
        return el

    def front(self):
        if self.is_empty():
            raise IndexError("Red je prazan!")
        return self.head.info
```



Slika 80. *Dijagram toka metode enqueue*



Slika 81. *Dijagram toka metode dequeue*

```
q = DQueue()
for x in [3, 7, 5]:
    q.enqueue(x)
print(q.front())
print(q.dequeue(), q.dequeue())
q.enqueue(9)
print(q.dequeue(), q.dequeue(), q.is_empty())
```

IZLAZ (OUTPUT):

```
3
3 7
5 9 True
```

Objašnjenje: Prazan red je poseban slučaj u obe operacije. Kada se u prazan red dodaje prvi element, on je istovremeno i prvi i poslednji, pa se postavljaju i `head` i `tail`. Kada se iz reda uzme poslednji element, `head` postaje `None`, a tada i `tail` mora postati `None`. Inače bi `tail` i dalje pokazivao na uklonjeni čvor, i sledeći `enqueue` bi zakačio novi čvor na čvor koji više nije u redu.

Stek i red u Python-u

Python nema posebne klase koje se zovu „stek“ i „red“, ali ima strukture koje se koriste u te svrhe:

Struktura	Python	Dodavanje	Uzimanje	Prvi element
stek	`list`	`s.append(x)`	`s.pop()`	`s[-1]`
red	`collections.deque`	`q.append(x)`	`q.popleft()`	`q[0]`
red za niti	`queue.Queue`	`q.put(x)`	`q.get()`	—

```
stek = []
stek.append(1)
stek.append(2)
stek.append(3)
print(stek[-1], stek.pop(), stek)

from collections import deque
red = deque()
red.append(1)
red.append(2)
red.append(3)
print(red[0], red.popleft(), list(red))
```

IZLAZ (OUTPUT):

```
3 3 [1, 2]
1 1 [2, 3]
```

> **Napomena:** Python lista se **ne koristi kao red**. Naredba `lista.pop(0)` uzima prvi element, ali zatim pomera sve ostale elemente za jedno mesto ulevo, pa je O(n). Klasa `deque` (engl. *double-ended queue*, red sa dva kraja) dodaje i uzima sa oba kraja za O(1). Moduli `collections` i `queue` su deo standardne biblioteke, pa se samo uključe naredbom `from collections import deque`, bez instalacije. Klasa `queue.Queue` je namenjena programima sa više niti (engl. *threads*); za obične programe koristi se `deque`.

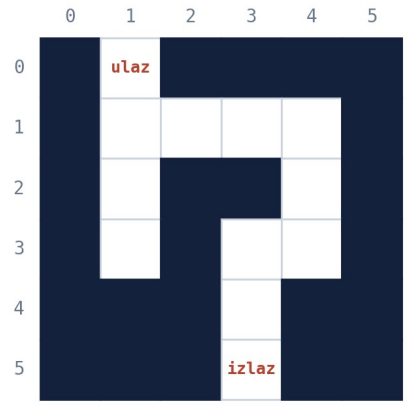
Kada šta koristiti: za učenje pišemo sopstvene klase, jer se tako vidi kako struktura radi iznutra. U stvarnim programima koriste se `list` za stek i `deque` za red.

Primena steka: određivanje puta kroz lavirint

Zadatak: odrediti put kojim miš treba da prođe da bi izašao iz lavirinta. Lavirint je zadat kao matrica prepreka. Miš može da pređe na susedno polje sa leve, desne,

gornje i donje strane. Ulaz u lavirint je u zadatoj koloni prve vrste, a izlaz je bilo koje polje u poslednjoj vrsti.

Lavirint zapisujemo kao listu stringova, u kojoj je `#` zid, a `.` prolaz. Polje je par (vrsta, kolona).



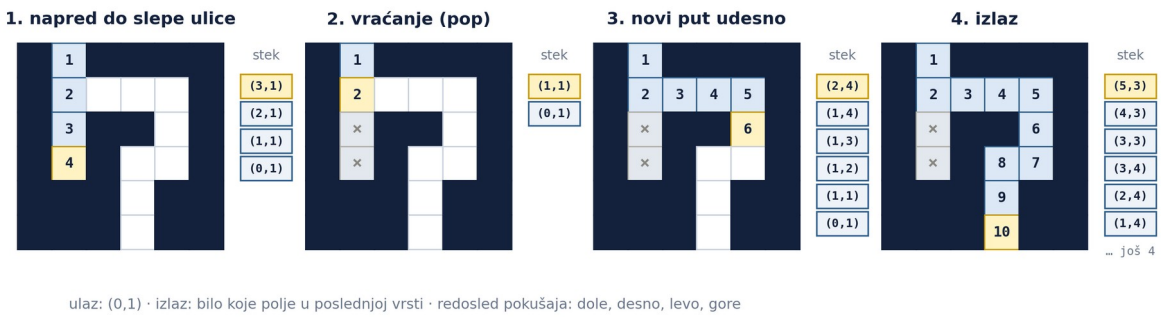
Slika 82. *Lavirint 6 × 6 sa ulazom u polju (0, 1) i izlazom u poslednjoj vrsti*

Ideja

Miš ne zna put unapred, pa ga traži pokušajima. Stek pamti put kojim je miš prošao od ulaza do trenutnog polja:

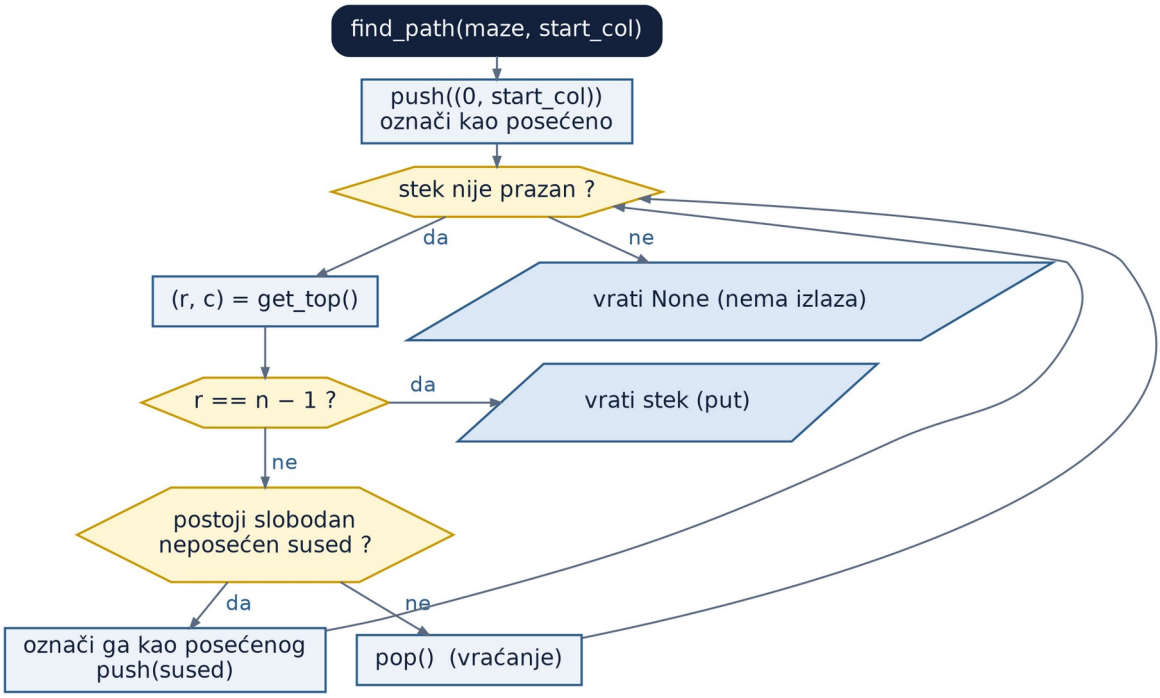
- Kada miš pređe na novo polje, to polje se stavlja na stek (**korak napred = push**).
- Kada miš dođe u slepu ulicu, odnosno nema nijedno susedno polje u koje nije već ulazio, polje se skida sa steka (**korak nazad = pop**). Miš se vraća na prethodno polje i odatle pokušava drugi smer.
- Polja u kojima je miš već bio se pamte (matrica `visited`), da ne bi ulazio dva puta u isto polje i vrteo se u krug.
- Kada je vrh steka u poslednjoj vrsti, miš je izašao, a stek sadrži ceo put od ulaza do izlaza. Ako se stek isprazni, miš se vratio do ulaza i izlaz ne postoji.

Ovo je kao Arijadnin konac u mitu o Tezeju: dok ide napred, Tezej odmotava konac; kada naiđe na slepu ulicu, namotava ga do poslednjeg račvanja i pokušava drugi hodnik. Konac je stek.



Slika 83. *Četiri stanja lavirinta i steka: napred do slepe ulice, vraćanje, novi put i izlaz*

Redosled pokušaja je: dole, desno, levo, gore. Miš prvo ide dole do polja (3, 1), koje je slepa ulica. Dva `pop`-a ga vraćaju na (1, 1), odakle ide udesno, pa dole, i stiže do izlaza (5, 3). Polja (2, 1) i (3, 1) ostaju označena kao posećena, pa miš u njih više ne ulazi.



Slika 84. *Dijagram toka funkcije find_path*

```
def is_free(maze, r, c):
    return 0 <= r < len(maze) and 0 <= c < len(maze[0]) and maze[r][c] == "."


def find_path(maze, start_col):
    n = len(maze)
    visited = [[False] * len(maze[0]) for _ in range(n)]
    moves = [(1, 0), (0, 1), (0, -1), (-1, 0)]    # dole, desno, levo, gore
    stack = DStack()
    stack.push((0, start_col))
    visited[0][start_col] = True
    while not stack.is_empty():
        r, c = stack.get_top()
        if r == n - 1:                            # stigli smo u poslednju
vrstu
            return stack
        moved = False
        for dr, dc in moves:
            nr, nc = r + dr, c + dc
            if is_free(maze, nr, nc) and not visited[nr][nc]:
                visited[nr][nc] = True
                stack.push((nr, nc))              # korak napred
                moved = True
                break
        if not moved:
            stack.pop()                           # slepa ulica: vrati se
    return None


def print_path(stack):
    path = []
    while not stack.is_empty():
        path.append(stack.pop())
    for i in range(len(path) - 1, -1, -1):        # od ulaza ka izlazu
        print(path[i], end=" ")
    print()

lavirint = ["#.####",
            "#....#",
            "#.##.#",
            "#.#..#",
            "###.##",
            "###.##"]
put = find_path(lavirint, 1)
if put is None:
    print("Nema izlaza iz lavirinta!")
else:
    print("Izlaz iz lavirinta:")
    print_path(put)
```

IZLAZ (OUTPUT):

```
Izlaz iz lavirinta:
(0, 1) (1, 1) (1, 2) (1, 3) (1, 4) (2, 4) (3, 4) (3, 3) (4, 3) (5, 3)
```

> **Objašnjenje:** Funkcija `is_free` proverava da li je polje unutar matrice i da nije zid. U glavnoj petlji gleda se polje na vrhu steka (`get_top`, ne `pop`, jer miš ostaje na njemu). Ako je u poslednjoj vrsti, put je pronađen. Inače se redom pokušavaju četiri smera, i prvo slobodno neposećeno susedno polje se stavlja na stek. Ako takvog nema, radi se `pop`. Funkcija `print_path` prazni stek; pošto je na vrhu izlaz, polja se prvo skupe u listu, pa ispišu unazad, od ulaza ka izlazu.

> **Napomena:** Originalni C++ kôd bira sledeći smer funkcijom `FindNextMove`, koja okreće smer kretanja u odnosu na polje iz koga je miš došao, i ne koristi matricu posećenih polja. Ta varijanta je teža za praćenje i, u lavirintima sa petljama, miš može da obilazi isti krug. Verzija sa matricom `visited` je jednostavnija i sigurno se završava. U originalu takođe konstruktor `PointInt(int tmpx, int tmpy)` postavlja x i y na 0 umesto na prosleđene vrednosti, što je greška; u Python-u polje predstavljamo običnim parom (torkom) `(vrsta, kolona)`. Stek pronalazi **neki** put, ne obavezno najkraći; najkraći put se nalazi pomoću reda (zadatak 5).

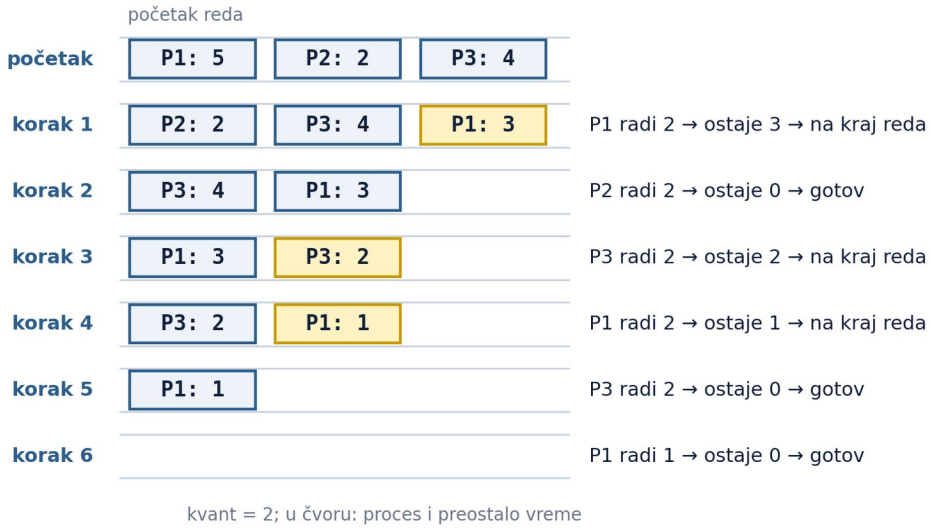
Primena reda: izvršavanje poslova

Zadatak: svaki od poslova ima određeni kvant vremena koji mu je potreban da se izvrši. Da bi izvršavanje teklo kontinualno za sve procese, svakom se dodeljuje kvant vremena, nakon čega se on zaustavlja i kreće sledeći proces. Obezbediti da procesi dobijaju kvante vremena redosledom u kom su se pojavili.

Ideja

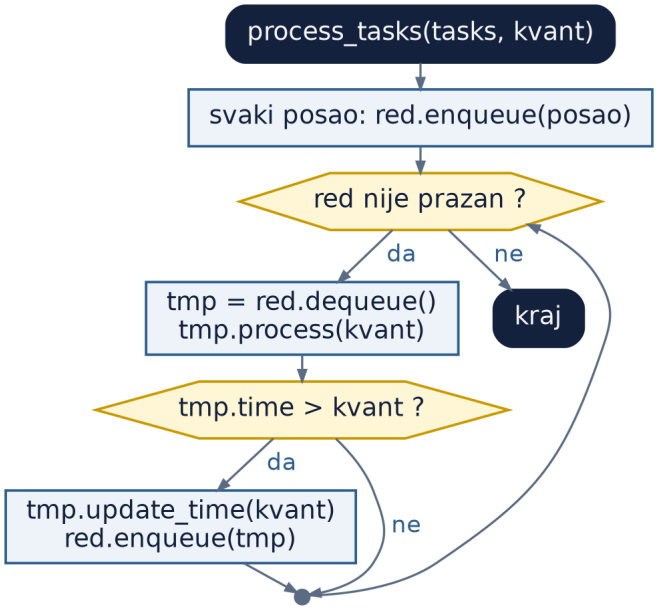
Ovo je raspoređivanje „u krug“ (engl. *round robin*), način na koji operativni sistem deli procesor između programa. Svi poslovi čekaju u redu. U svakom koraku:

1. uzima se posao sa početka reda (`dequeue`);
2. on radi jedan kvant vremena (ili manje, ako mu treba manje);
3. ako posao nije gotov, vraća se **na kraj reda** sa umanjenim preostalim vremenom (`enqueue`); ako je gotov, više se ne vraća.



Slika 85. *Stanje reda poslova po koracima, za kvant 2*

Na početku P1 treba 5 jedinica vremena, P2 treba 2, a P3 treba 4. P1 radi 2, ostaje mu 3, pa ide na kraj reda. P2 radi 2 i završava se. P3 radi 2 i vraća se sa 2 preostale jedinice. Postupak se ponavlja dok se red ne isprazni.



Slika 86. *Dijagram toka funkcije process_tasks*

```
class Task:
    def __init__(self, id_task, time):
        self.id_task = id_task
        self.time = time              # preostalo vreme

    def update_time(self, tm):
        self.time -= tm

    def process(self, kvant):
        rad = kvant if self.time > kvant else self.time
        print("Proces", self.id_task, "radi", rad, "- ostaje", self.time -
rad)


def process_tasks(tasks, kvant):
    red = DQueue()
    for i in range(len(tasks)):
        red.enqueue(tasks[i])
    while not red.is_empty():
        tmp = red.dequeue()
        tmp.process(kvant)
        if tmp.time > kvant:
            tmp.update_time(kvant)
            red.enqueue(tmp)              # nije gotov: na kraj reda

poslovi = [Task(1, 5), Task(2, 2), Task(3, 4)]
process_tasks(poslovi, 2)
```

IZLAZ (OUTPUT):

```
Proces 1 radi 2 - ostaje 3
Proces 2 radi 2 - ostaje 0
Proces 3 radi 2 - ostaje 2
Proces 1 radi 2 - ostaje 1
Proces 3 radi 2 - ostaje 0
Proces 1 radi 1 - ostaje 0
```

Objašnjenje: Red garantuje pravednost: svaki posao dobija kvant redom, i nijedan ne čeka beskonačno. Posao koji se vrati na kraj čeka dok svi ostali ne dobiju svoj kvant. U originalu metoda `process` samo ispisuje broj procesa; ovde ispisuje i koliko je posao radio i koliko mu je ostalo, da bi se pratilo izvršavanje.

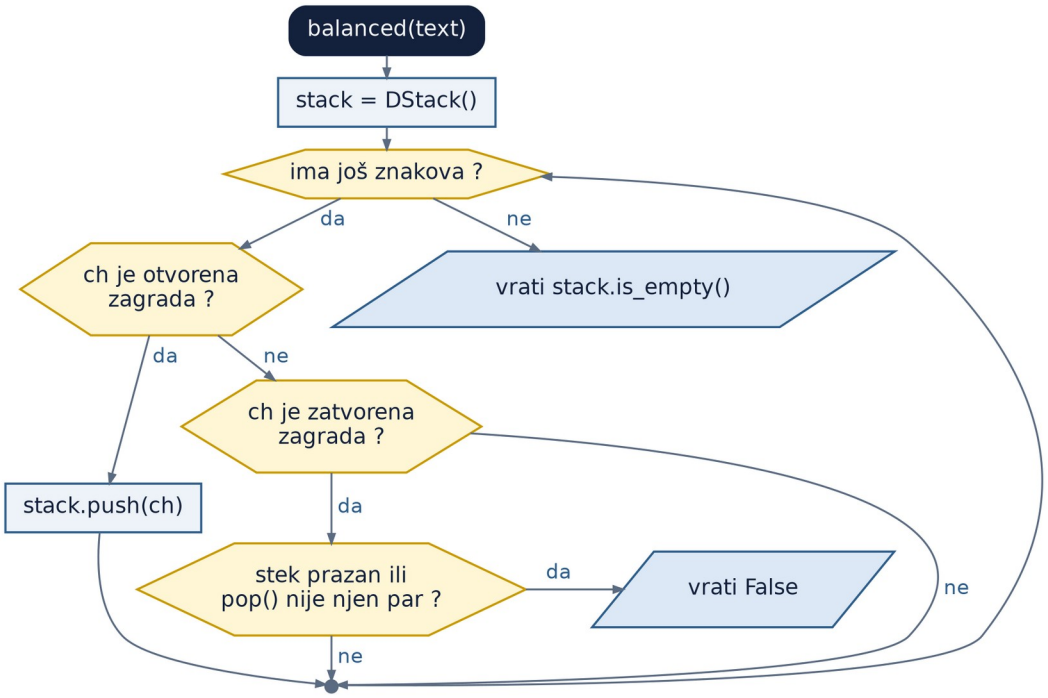
Zadaci za vežbu sa rešenjima

Svaki zadatak ima tekst, algoritam u obliku dijagrama toka, kompletno rešenje i objašnjenje. Rešenje je samostalan program: prvi deo sadrži klase koje su potrebne (stek `DStack` ili red `DQueue` nad lančanom listom), a drugi deo funkcije rešenja i primer korišćenja.

Zadatak 1: Provera uparenosti zagrada

Napisati funkciju `balanced(text)` koja proverava da li su zagrade `()`, `[]` i `{}` u izrazu pravilno uparene: svaka otvorena zagrada mora biti zatvorena odgovarajućom zatvorenom, i to obrnutim redom otvaranja.

Algoritam:



Slika 87. *Dijagram toka funkcije balanced*

Rešenje:

1. Klase potrebne za rešenje: čvor `StackNode` i stek `DStack`. Kôd je isti kao u ovom poglavlju:

```
class StackNode:
    def __init__(self, info, next_node=None):
        self.info = info
        self.next = next_node     # čvor ispod u steku


class DStack:
    def __init__(self):
        self.top = None           # pokazivač na vrh steka

    def is_empty(self):
        return self.top is None

    def push(self, el):
        self.top = StackNode(el, self.top)

    def pop(self):
        if self.is_empty():
            raise IndexError("Stek je prazan!")
        el = self.top.info
        self.top = self.top.next
        return el

    def get_top(self):
        if self.is_empty():
            raise IndexError("Stek je prazan!")
        return self.top.info
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def balanced(text):
    pairs = {")": "(", "]": "[", "}": "{"}
    stack = DStack()
    for i in range(len(text)):
        ch = text[i]
        if ch == "(" or ch == "[" or ch == "{":
            stack.push(ch)                    # otvorena: čeka svoj par
        elif ch == ")" or ch == "]" or ch == "}":
            if stack.is_empty() or stack.pop() != pairs[ch]:
                return False                  # nema para ili pogrešan par
    return stack.is_empty()                   # ostale neuparene otvorene?

for izraz in ["(a + b) * [c - d]", "{[()()]}", "(a + b]", "((x)", "x) + (y"]:
    print("%-18s %s" % (izraz, balanced(izraz)))
```

IZLAZ (OUTPUT):

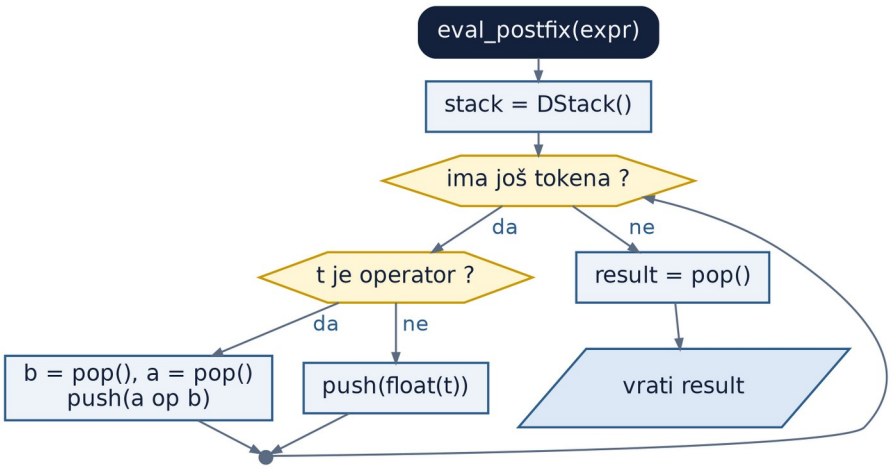
```
(a + b) * [c - d]  True
{[()()]}           True
(a + b]            False
((x)               False
x) + (y            False
```

> **Objašnjenje:** Otvorene zagrade se stavljaju na stek i čekaju svoj par. Kada naiđe zatvorena zagrada, ona mora da se upari sa **poslednjom** otvorenom, a to je upravo vrh steka. Zato je stek prirodan izbor: zagrade se zatvaraju obrnutim redom od otvaranja (LIFO). Greška je ako je stek prazan (nema otvorene zagrade za par) ili ako je na vrhu pogrešna vrsta zagrade. Na kraju stek mora biti prazan; inače je neka zagrada ostala otvorena. Rečnik `pairs` za svaku zatvorenu zagradu daje odgovarajuću otvorenu. Složenost je O(n).

Zadatak 2: Izračunavanje izraza u postfiksnom zapisu

U postfiksnom zapisu operator stoji iza operanada: izraz (3 + 4) · 2 piše se `3 4 + 2 *`. Ovaj zapis nema zagrade i ne zahteva pravila prioriteta. Napisati funkciju `eval_postfix(expr)` koja izračunava izraz u postfiksnom zapisu; elementi izraza su razdvojeni razmacima.

Algoritam:



Slika 88. *Dijagram toka funkcije eval_postfix*

Rešenje:

1. Klase potrebne za rešenje: čvor `StackNode` i stek `DStack`. Kôd je isti kao u ovom poglavlju:

```
class StackNode:
    def __init__(self, info, next_node=None):
        self.info = info
        self.next = next_node     # čvor ispod u steku


class DStack:
    def __init__(self):
        self.top = None           # pokazivač na vrh steka

    def is_empty(self):
        return self.top is None

    def push(self, el):
        self.top = StackNode(el, self.top)

    def pop(self):
        if self.is_empty():
            raise IndexError("Stek je prazan!")
        el = self.top.info
        self.top = self.top.next
        return el

    def get_top(self):
        if self.is_empty():
            raise IndexError("Stek je prazan!")
        return self.top.info
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def eval_postfix(expr):
    stack = DStack()
    tokens = expr.split()
    for i in range(len(tokens)):
        t = tokens[i]
        if t == "+" or t == "-" or t == "*" or t == "/":
            b = stack.pop()                    # drugi operand je na vrhu
            a = stack.pop()
            if t == "+":
                stack.push(a + b)
            elif t == "-":
                stack.push(a - b)
            elif t == "*":
                stack.push(a * b)
            else:
                stack.push(a / b)
        else:
            stack.push(float(t))               # broj ide na stek
    result = stack.pop()
    if not stack.is_empty():
        raise ValueError("Neispravan izraz!")
    return result

print(eval_postfix("3 4 + 2 *"))               # (3 + 4) * 2
print(eval_postfix("5 1 2 + 4 * + 3 -"))       # 5 + (1 + 2) * 4 - 3
print(eval_postfix("10 2 8 * + 3 -"))          # 10 + 2 * 8 - 3
```

IZLAZ (OUTPUT):

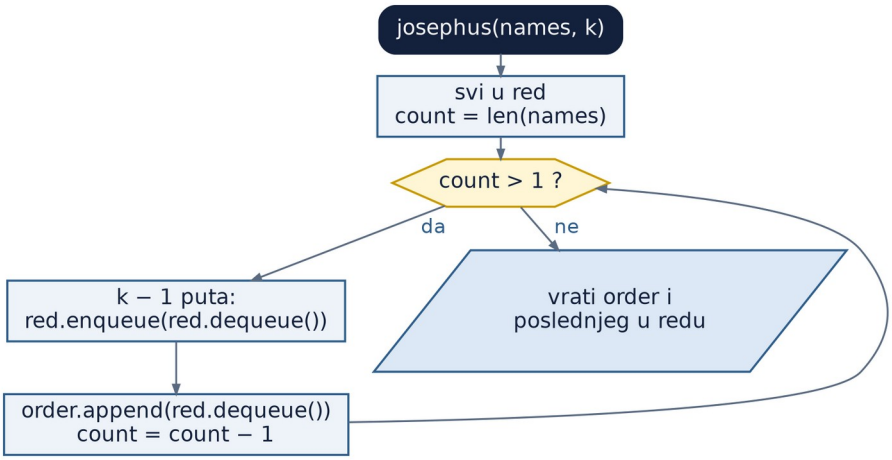
```
14.0
14.0
23.0
```

Objašnjenje: Izraz se čita sleva nadesno. Broj se stavlja na stek. Kada naiđe operator, sa steka se skidaju dva broja, primeni se operator i rezultat se vrati na stek. Važno je redosled: prvi `pop` daje **drugi** operand (b), a drugi `pop` prvi operand (a), što je bitno kod oduzimanja i deljenja. Na kraju na steku mora ostati tačno jedan broj, rezultat. Kalkulatori i kompajleri izraze prvo prevode u postfiksni zapis, pa ih ovako izračunavaju. Metoda `split()` deli string na delove po razmacima.

Zadatak 3: Josifov problem

N osoba stoji u krugu. Broji se od prve osobe, i svaka k-ta osoba ispada iz kruga; brojanje se nastavlja od sledeće. Napisati funkciju `josephus(names, k)` koja vraća redosled ispadanja i poslednju preostalu osobu.

Algoritam:



Slika 89. *Dijagram toka funkcije josephus*

Rešenje:

1. Klase potrebne za rešenje: čvor `QueueNode` i red `DQueue`. Kôd je isti kao u ovom poglavlju:

```
class QueueNode:
    def __init__(self, info):
        self.info = info
        self.next = None          # čvor iza u redu


class DQueue:
    def __init__(self):
        self.head = None          # početak reda: odavde se uzima
        self.tail = None          # kraj reda: ovde se dodaje

    def is_empty(self):
        return self.head is None

    def enqueue(self, el):
        node = QueueNode(el)
        if self.is_empty():
            self.head = self.tail = node
        else:
            self.tail.next = node
            self.tail = node

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Red je prazan!")
        el = self.head.info
        self.head = self.head.next
        if self.head is None:     # red je ostao prazan
            self.tail = None
        return el

    def front(self):
        if self.is_empty():
            raise IndexError("Red je prazan!")
        return self.head.info
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def josephus(names, k):
    red = DQueue()
    for i in range(len(names)):
        red.enqueue(names[i])
    order = []
    count = len(names)
    while count > 1:
        for i in range(k - 1):
            red.enqueue(red.dequeue())          # prvih k-1 ide na kraj reda
        order.append(red.dequeue())             # k-ti ispada
        count -= 1
    return order, red.dequeue()

ispali, pobednik = josephus(["Ana", "Bojan", "Ceca", "Dule", "Ema", "Filip"],
3)
print("Redosled ispadanja:", ispali)
print("Pobednik:", pobednik)
```

IZLAZ (OUTPUT):

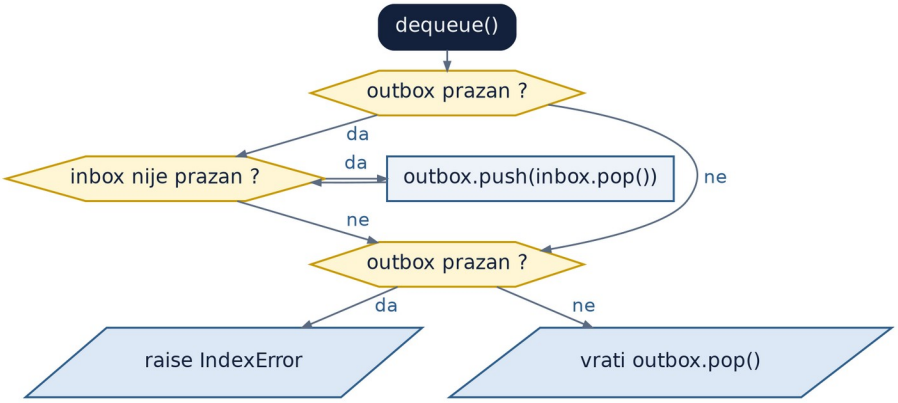
```
Redosled ispadanja: ['Ceca', 'Filip', 'Dule', 'Bojan', 'Ema']
Pobednik: Ana
```

Objašnjenje: Krug se predstavlja redom: osoba na početku reda je ona koja je na redu za brojanje. Prvih k − 1 osoba se uzme sa početka i vrati na kraj reda (one su „izbrojane“, ali ostaju u krugu), a k-ta se uzme i ne vraća (ispada). Tako red stalno „kruži“, kao krug ljudi. Za 6 osoba i k = 3 ispadaju redom Ceca, Filip, Dule, Bojan i Ema, a pobeđuje Ana. Složenost je O(n · k).

Zadatak 4: Red napravljen od dva steka

Napraviti klasu `QueueTwoStacks` koja se ponaša kao red (FIFO), a interno koristi samo dva steka (LIFO), bez drugih struktura.

Algoritam:



Slika 90. *Dijagram toka metode dequeue reda od dva steka*

Rešenje:

1. Klase potrebne za rešenje: čvor `StackNode` i stek `DStack`. Kôd je isti kao u ovom poglavlju:

```
class StackNode:
    def __init__(self, info, next_node=None):
        self.info = info
        self.next = next_node     # čvor ispod u steku


class DStack:
    def __init__(self):
        self.top = None           # pokazivač na vrh steka

    def is_empty(self):
        return self.top is None

    def push(self, el):
        self.top = StackNode(el, self.top)

    def pop(self):
        if self.is_empty():
            raise IndexError("Stek je prazan!")
        el = self.top.info
        self.top = self.top.next
        return el

    def get_top(self):
        if self.is_empty():
            raise IndexError("Stek je prazan!")
        return self.top.info
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
class QueueTwoStacks:
    def __init__(self):
        self.inbox = DStack()      # ovde stižu novi elementi
        self.outbox = DStack()     # odavde se uzimaju, u obrnutom redu

    def is_empty(self):
        return self.inbox.is_empty() and self.outbox.is_empty()

    def enqueue(self, el):
        self.inbox.push(el)

    def dequeue(self):
        if self.outbox.is_empty():
            while not self.inbox.is_empty():    # prespi inbox u outbox
                self.outbox.push(self.inbox.pop())
        if self.outbox.is_empty():
            raise IndexError("Red je prazan!")
        return self.outbox.pop()

q = QueueTwoStacks()
for x in [1, 2, 3]:
    q.enqueue(x)
print(q.dequeue(), q.dequeue())
q.enqueue(4)
q.enqueue(5)
while not q.is_empty():
    print(q.dequeue(), end=" ")
print()
```

IZLAZ (OUTPUT):

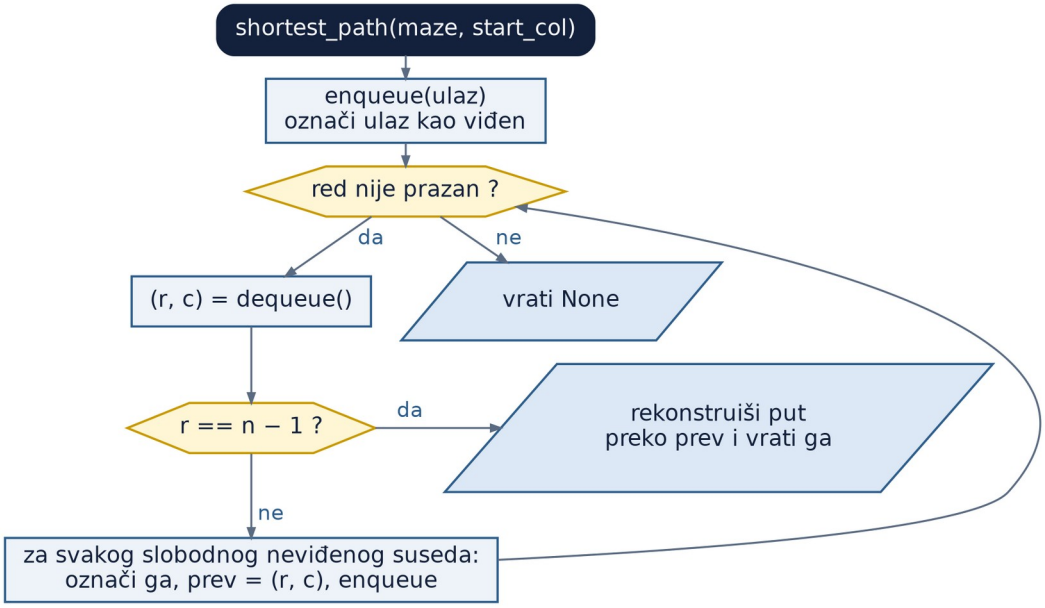
```
1 2
3 4 5
```

Objašnjenje: Novi elementi idu na stek `inbox`. Kada treba uzeti element, a stek `outbox` je prazan, svi elementi se prespu iz `inbox` u `outbox`. Prespi obrne redosled, pa je najstariji element na vrhu `outboxa`. Elementi se zatim uzimaju sa `outboxa` sve dok se ne isprazni, i tek onda se ponovo presipa. Svaki element se presipa najviše jednom, pa je prosečna (amortizovana) složenost svake operacije O(1), iako jedno presipanje može biti O(n). Ovaj zadatak pokazuje da se jedna struktura može napraviti od druge.

Zadatak 5: Najkraći put kroz lavirint

Napisati funkciju `shortest_path(maze, start_col)` koja nalazi **najkraći** put kroz lavirint iz primera sa vežbi, koristeći red umesto steka (pretraga u širinu, engl. *breadth-first search*, BFS).

Algoritam:



Slika 91. *Dijagram toka funkcije shortest_path*

Rešenje:

1. Klase potrebne za rešenje: čvor `QueueNode` i red `DQueue`. Kôd je isti kao u ovom poglavlju:

```
class QueueNode:
    def __init__(self, info):
        self.info = info
        self.next = None          # čvor iza u redu


class DQueue:
    def __init__(self):
        self.head = None          # početak reda: odavde se uzima
        self.tail = None          # kraj reda: ovde se dodaje

    def is_empty(self):
        return self.head is None

    def enqueue(self, el):
        node = QueueNode(el)
        if self.is_empty():
            self.head = self.tail = node
        else:
            self.tail.next = node
            self.tail = node

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Red je prazan!")
        el = self.head.info
        self.head = self.head.next
        if self.head is None:     # red je ostao prazan
            self.tail = None
        return el

    def front(self):
        if self.is_empty():
            raise IndexError("Red je prazan!")
        return self.head.info
```

2. Funkcije rešenja i primer korišćenja (nastavak istog programa):

```
def shortest_path(maze, start_col):
    n = len(maze)
    m = len(maze[0])
    prev = [[None] * m for _ in range(n)]      # odakle smo došli u polje
    seen = [[False] * m for _ in range(n)]
    moves = [(1, 0), (0, 1), (0, -1), (-1, 0)]
    red = DQueue()
    red.enqueue((0, start_col))
    seen[0][start_col] = True
    while not red.is_empty():
        r, c = red.dequeue()
        if r == n - 1:                         # prvi stigao u poslednju vrstu
            path = []
            while (r, c) != (0, start_col):
                path.append((r, c))
                r, c = prev[r][c]
            path.append((0, start_col))
            return path[::-1]
        for dr, dc in moves:
            nr, nc = r + dr, c + dc
            if 0 <= nr < n and 0 <= nc < m and maze[nr][nc] == "." and not
seen[nr][nc]:
                seen[nr][nc] = True
                prev[nr][nc] = (r, c)
                red.enqueue((nr, nc))
    return None

lavirint = ["#.####",
            "#....#",
            "#.##.#",
            "#.#..#",
            "#...##",
            "###.##"]
put = shortest_path(lavirint, 1)
print("Dužina puta:", len(put))
print(put)
```

IZLAZ (OUTPUT):

```
Dužina puta: 8
[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (4, 2), (4, 3), (5, 3)]
```

Objašnjenje: Stek vodi miša što dublje u jedan hodnik pre nego što proba drugi, pa nalazi neki put, ali ne nužno najkraći. Red obrađuje polja „talasima“: prvo sva polja na udaljenosti 1 od ulaza, pa sva na udaljenosti 2, i tako dalje. Zato je prvo polje iz poslednje vrste koje se izvadi iz reda ujedno i najbliže ulazu. Matrica `prev` pamti iz kog polja smo došli u svako polje, pa se put rekonstruiše unazad od izlaza do ulaza i na kraju obrne (`path[::-1]`). U ovom lavirintu postoje dva puta; najkraći ima 8 polja, a duži, desni put ima 10.