

Softverski algoritmi u sistemima automatskog upravljanja

Strukture podataka 2

Stack

Stack je apstraktna struktura popodataka koja radi po principu **LIFO** (*Last In, First Out*) —
poslednji elemnt koji je dodat je prvi koji se uklanja



Analogija iz stvarnog sveta:

- ▶ Slaganje tanjira — **poslednji tanjur** koji je stavljen na vrh je **prvi koji će biti uzet**

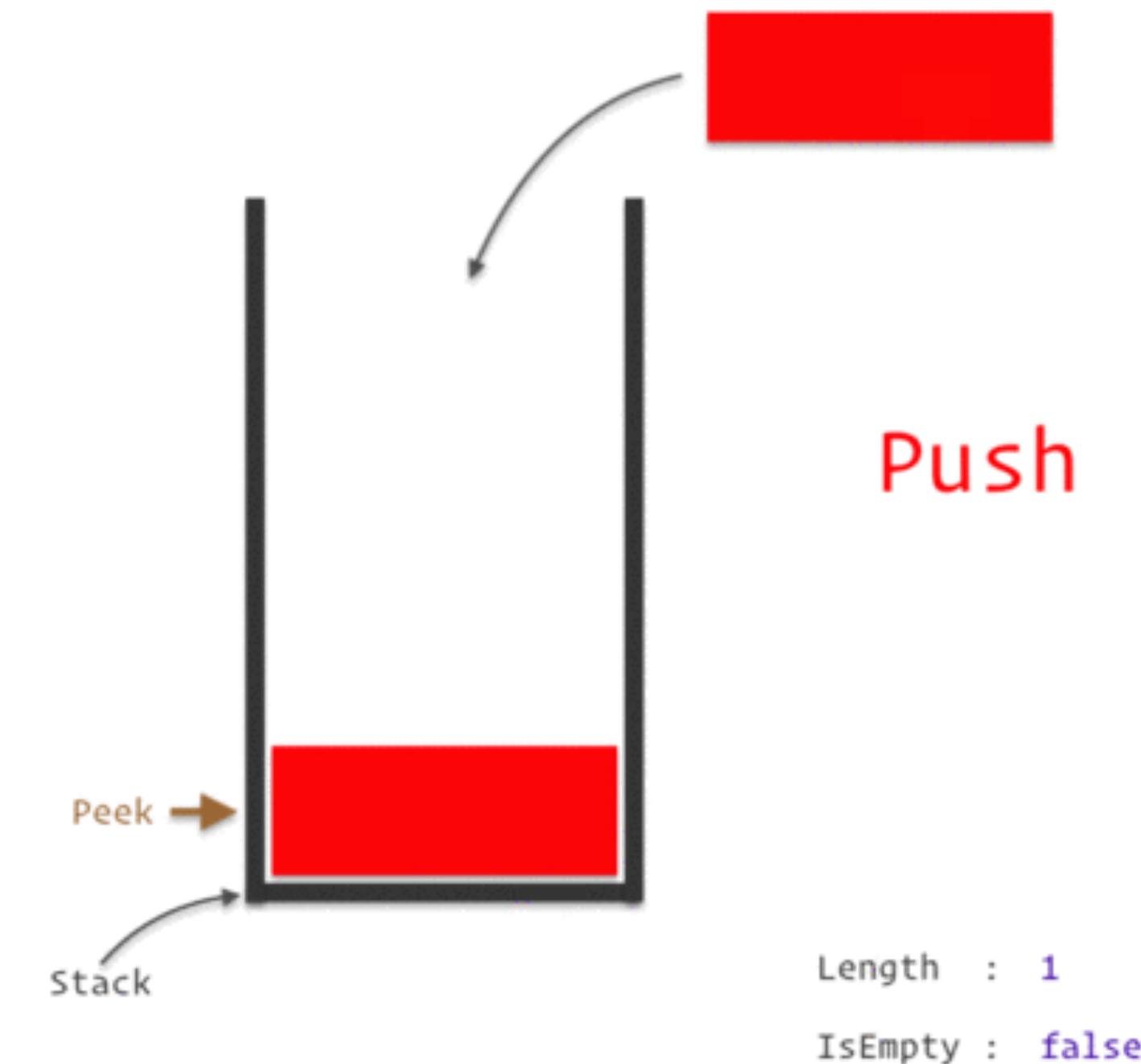


Primene u programiranju

- ▶ Praćenje poziva funkcija (*call stack*)
- ▶ Implementacija *undo/redo*
- ▶ Parsiranje izraza i balansiranje zagrada

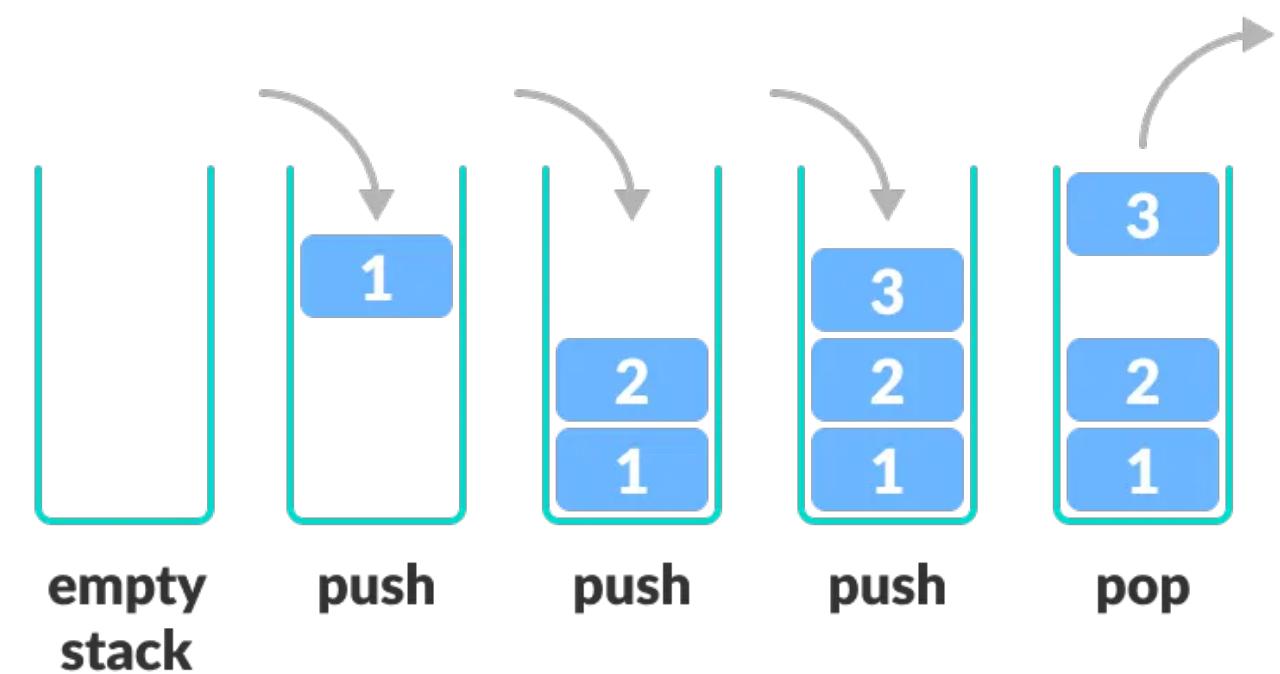
! Zašto je bitan?

- ▶ **Jednostavan** je za implementaciju
- ▶ **Efikasan** — osnovne operacije su
- ▶ Pojavljuje se u **brojnim algoritima** i **problemstkim zadacima**



Kako implementirati Stack u Pythonu?

- ▶ U Pythonu ne postoji posebna **ugrađena struktura** podataka stack, ali je lako možemo implementirati korišćenjem **klasa** i **OOP pristupa**
- ▶ Koristićemo klasu **Stack** u kojoj ćemo definisati osnovne metode za rad sa stekom:
 - ▶ `push()` — dodavanje elementa
 - ▶ `pop()` — uklanjanje poslednjeg elementa
 - ▶ `peek()` — uvid u vrh steka
 - ▶ `is_empty()` — provera da li je stek prazan
- ▶ Unutrašnje skladištenje elemenata ćemo rešiti pomoću obične **liste (list)**, jer nudi efikasne operacije na kraju



Ovako dobijamo čist, organizovan i proširiv kod koji olakšava kasniju upotrebu i testiranje.

Kreiranje klase i konstruktor

```
class Stack:  
    def __init__(self):  
        self.items = []
```



Objašnjenje:

- ▶ Prilikom kreiranja objekta klase `Stack`, inicializujemo praznu listu `items` koja će sadržati elemente steka



Zašto ovako?

- ▶ Lista omogućava efikasne operacije na kraju $O(1)$), a enkapsulacijom u klasu olakšavamo dodavanje dodatnih funkcionalnosti kasnije

Osnovne metode

```
def push(self, item):  
    self.items.append(item)
```

- ▶ Dodaje novi element na vrh steka (kraj liste)
- ▶ Vremenska složenost: $O(1)$

```
def peek(self):  
    if not self.is_empty():  
        return self.items[-1]  
    return None
```

- ▶ Vraća element sa vrha steka bez uklanjanja
- ▶ Ako je stek prazan vraća None
- ▶ Vremenska složenost: $O(1)$

```
def pop(self):  
    if not self.is_empty():  
        return self.items.pop()  
    return None
```

- ▶ Uklanja i vraća poslednji ubačeni element

- ▶ Ako je stek prazan vraća None
- ▶ Vremenska složenost: $O(1)$

```
def is_empty(self):  
    return len(self.items) == 0
```

- ▶ Proverava da li je stek prazan
- ▶ Vremenska složenost: $O(1)$

Zadatak: Validacija izraza sa zagradama

Zadatak

- ▶ Napisati program koji simulira izvršavanje zadatka preko reda čekanja (Queue

Podržane zgrade

- ▶ **Okrugle:** ()
- ▶ **Uglaste:** []
- ▶ **Vitičaste:** { }

Cilj

- ▶ Svaka **otvorena** zgrada mora imati **odgovarajuću zatvorenu**
- ▶ **Redosled uparivanja** mora biti ispravan (*LIFO* princip)
- ▶ Na kraju obrade **stek mora biti prazan**

Queue

Queue je apstraktna struktura podataka koja radi po principu **FIFO** (*First In, First Out*) —
prvi elemnt koji je dodat je prvi koji se uklanja



Analogija iz stvarnog sveta:

- ▶ Red u banci, supermarketu ili kol-centru — **prvi koji stigne je prvi koji se uslužuje**

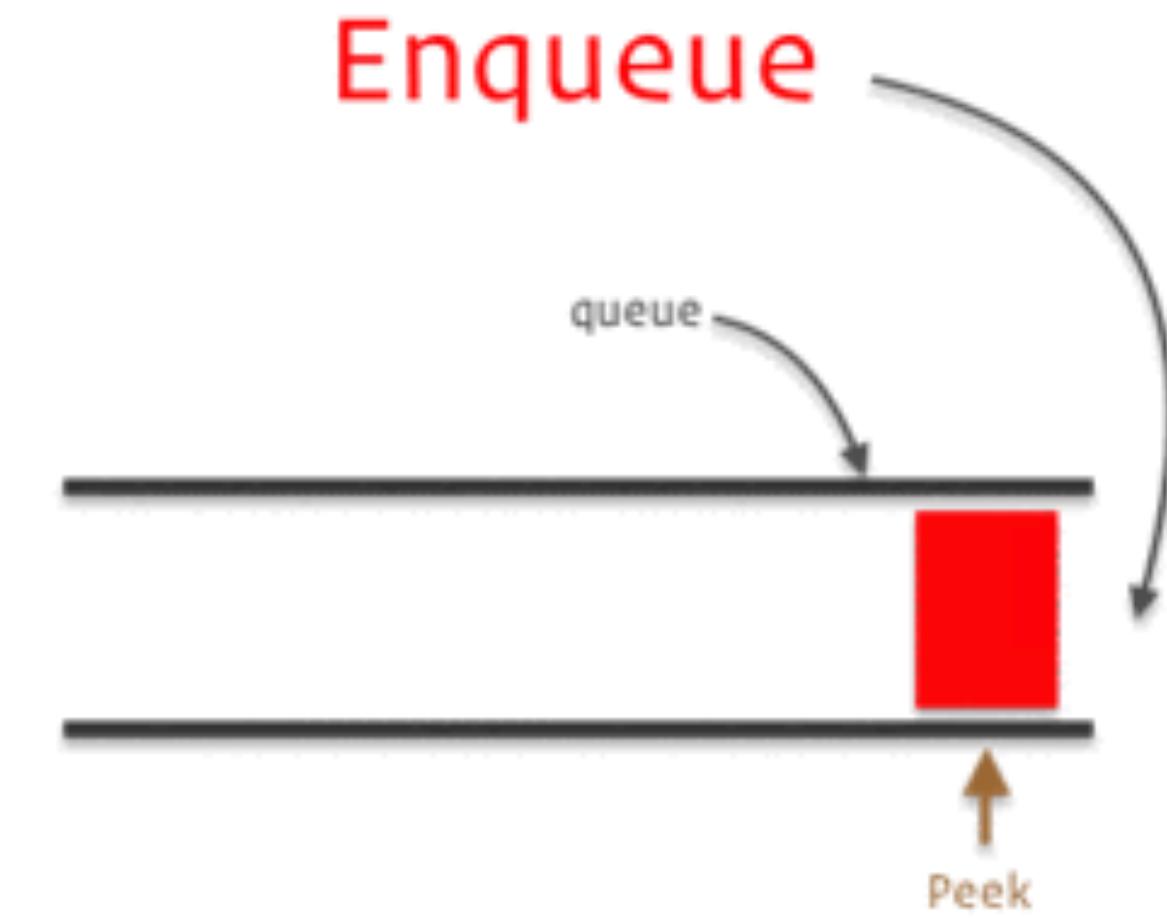


Primene u programiranju

- ▶ **Planiranje procesa** u OS
- ▶ **BFS** algoritam u grafovima
- ▶ **Kontrola toka podataka** u mrežama

! Zašto je bitan?

- ▶ Omogućava *fer redosled obrade*
- ▶ Ima **efikasne operacije** kad se korisit deque
- ▶ Neophodan u mnogim **algoritmima i simulacijama**



IsEmpty : false

Length : 1

Kako implementirati Queue u Pythonu?

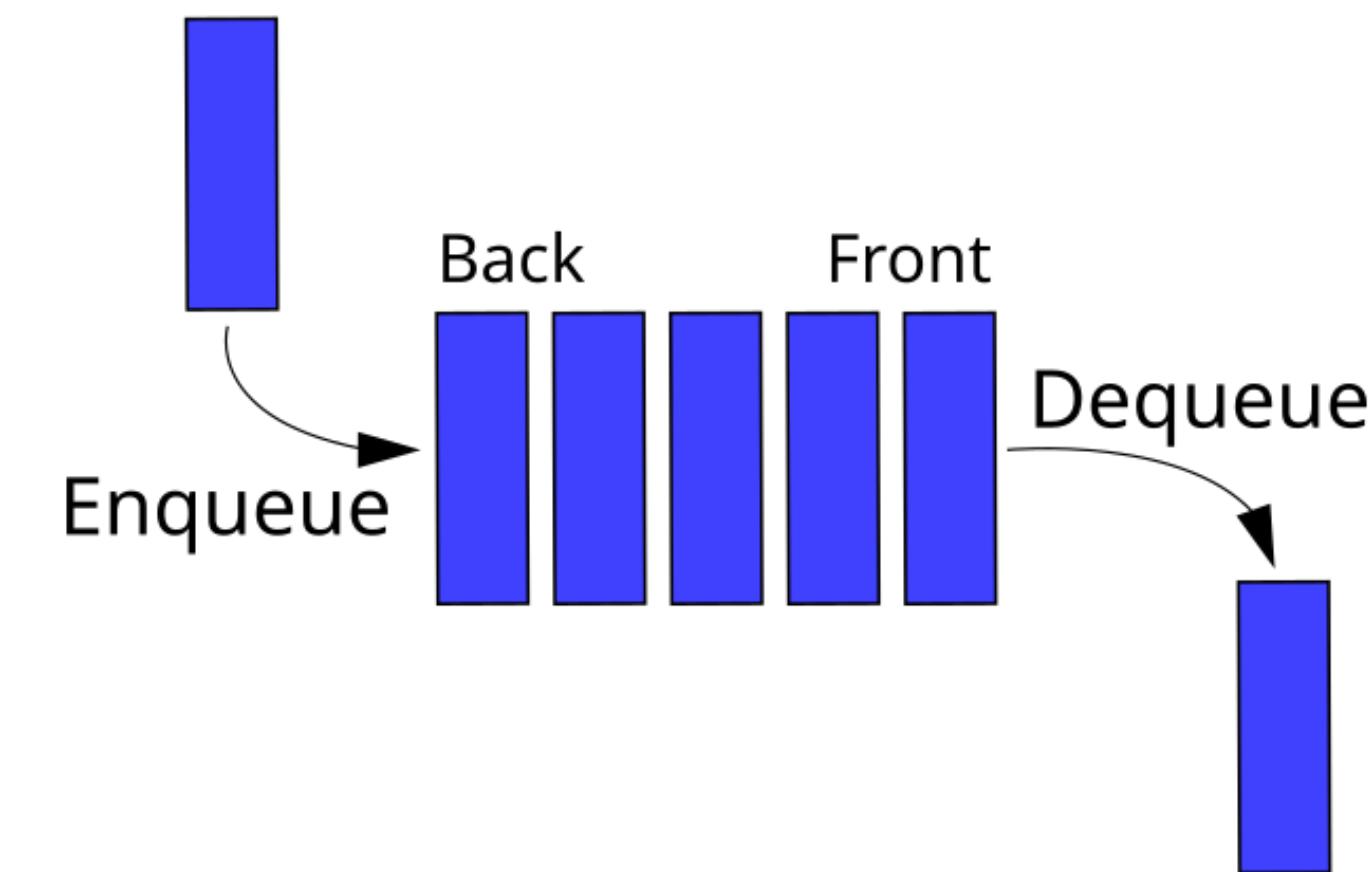
🚫 Zašto nećemo koristiti listu?

- ▶ Iako `list.append()` ima vremensku složenost $O(1)$, metoda `pop(0)` ima $O(n)$ jer mora pomeriti sve elemente uлево.
- ▶ To čini listu **neefikasnom za uklanjanje sa početka reda** (što je suština queue-a).

✓ Rešenje: `collections.deque`

```
from collections import deque  
  
queue = deque()
```

- ▶ `deque.append()` → dodavanje na kraj reda: $O(1)$
- ▶ `deque.popleft()` → uklanjanje sa početka: $O(1)$



`deque` (*double-ended queue*) podržava efikasne operacije i na početku i na kraju kolekcije

Kreiranje klase i konstruktor

```
from collections import deque

class Queue:
    def __init__(self):
        self.items = deque()
```



Objašnjenje:

- ▶ U konstruktoru (`__init__`) pravimo promenljivu `self.items` koja je instanca `deque`
- ▶ `deque` je **dvostruko povezani red** (*double-ended queue*) koji omogućava:
 - ▶ **Brze operacije** dodavanja i uklanjanja elemenata **sa obe strane**
 - ▶ **Bolju efikasnost** od list za red



Zašto ovako?

- ▶ Dizajniran je upravo za ovakve slučajeve korišćenja
- ▶ Ponaša se kao red sa obe strane: možemo lako implementirati običan `queue`, `stack`, pa čak i `priority queue` varijante

Osnovne metode

```
def enqueue(self, item):  
    self.items.append(item)
```

- ▶ Dodaje novi element na kraj reda
- ▶ Vremenska složenost: $O(1)$

```
def peek(self):  
    if self.is_empty():  
        return None  
    return self.items[0]
```

- ▶ Vraća element na početku reda bez uklanjanja
- ▶ Vremenska složenost: $O(1)$

```
def dequeue(self):  
    if self.is_empty():  
        return None  
    return self.items.popleft()
```

- ▶ Uklanja i vraća prvi dodat element
- ▶ Vremenska složenost: $O(1)$

```
def is_empty(self):  
    return len(self.items) == 0
```

- ▶ Proverava da li red ima elemenata
- ▶ Vremenska složenost: $O(1)$

Zadatak: Upravljanje redom zadataka

Zadatak

- ▶ Napisati program koji simulira **izvršavanje zadataka preko reda čekanja** (Queue)
- ▶ Svaki zadatak ima **naziv** i **trajanje u sekundama**

Ulazni podaci

- ▶ Zadatak se unosi kao **par (naziv, trajanje)**
- ▶ Zadatke dodajemo u **red po FIFO principu**

```
[("Backup fajlova", 3), ("Slanje izveštaja", 2), ("Restart sistema", 1)]
```

Cilj

- ▶ **Dodati** sve zadatke u red koristeći enqueue()
- ▶ **Izvršavati** ih redom koristeći dequeue()
- ▶ Ispisati poruku: "Izvršava se zadatak: <naziv> (traje <n> sekundi)"
- ▶ Na kraju, red mora biti **prazan**

Linked List

Povezana lista je linearna struktura podataka u kojoj su elementi (čvorovi) međusobno povezani pokazivačima.

Za razliku od nizova, **elementi nisu smešteni u susetnim memorijskim lokacijama**.



Analogija iz stvarnog sveta:

- ▶ Vagoni — svaki vagon zna gde je sledeći



Primene u programiranju

- ▶ **Dinamičko** upravljanje memorijom
- ▶ Implementacija **složenijih struktura**
- ▶ Često **dodavanje/uklanjanje** elemenata



! Zašto je bitan?

- ▶ Nema potrebe za **predefinisanim kapacitetom**
- ▶ Dodavanje i uklanjanje elemnata sa **početka ili sredine**
- ▶ Odlična za probleme sa **promejnjivim brojem elemenata**

Kako implementirati Linked List u Pythonu?

- ▶ U Pythonu nemamo ugrađenu LinkedList strukturu — zato je sami pravimo koristeći **klase**
- ▶ Napravimo dve klase:
 - 🔗 Node – predstavlja **jedan čvor liste** (vrednost + veza ka sledećem)
 - 📦 LinkedList – **upravlja celom listom** (čuva početni čvor i metode)

📌 Šta radimo u konstruktorima?

Node klasa:

```
class Node:  
    def __init__(self, value):  
        self.value = value      # vrednost čvora  
        self.next = None         # veza ka sledećem čvoru
```

LinkedList klasa:

```
class LinkedList:  
    def __init__(self):  
        self.head = None          # početak liste
```



Prednosti ovakve implementacije:

- ▶ Možemo lako **dodavati ili brisati čvorove bez pomeranja ostalih**
- ▶ **Modularna struktura** — svaki Node je nezavisna jedinica
- ▶ **Pogodno za proširivanje** (npr. dvosmerna lista, kuržna lista)

Osnovne metode

```
def append(self, value):
    new_node = Node(value)
    if not self.head:
        self.head = new_node
    else:
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
```

- ▶ Dodavanje čvora na kraj liste
- ▶ Vremenska složenost: $O(n)$

```
def find(self, value):
    current = self.head
    while current:
        if current.value == value:
            return current
        current = current.next
    return None
```

- ▶ Pronalaženje elementa u listi
- ▶ Vremenska složenost: $O(n)$

```
def prepend(self, value):
    new_node = Node(value)
    new_node.next = self.head
    self.head = new_node
```

- ▶ Dodavanje čvora na početak liste
- ▶ Vremenska složenost: $O(1)$

```
def delete(self, value):
    current = self.head
    if current and current.value == value:
        self.head = current.next
        return
    while current and current.next:
        if current.next.value == value:
            current.next = current.next.next
            return
        current = current.next
```

- ▶ Brisanje prvog čvora sa datom vrednošću
- ▶ Vremenska složenost: $O(n)$

Zadatak: Josephus Problem

Zadatak

- ▶ Zamislimo **n osoba** koje stoje u krugu i broje u krug do broja **k**. Svaka **k-ta osoba biva eliminisana** iz kruga, i brojanje kreće iznova od sledeće osobe. Proces se ponavlja dok ne ostane **samo jedan preživeli**.

Ulazni podaci

- ▶ n – broj osoba u krugu
- ▶ k – broj nakon kojeg se osoba eliminiše

Izlaz:

- ▶ Rednosled eliminisanih osoba
- ▶ Broj preživelih osoba

Cilj

- ▶ Implementirati **cirkularnu jednostruko povezanu listu** i koristiti je za rešavanje Josephus problema.
Prikazati redosled eliminacija i odrediti konačnog pobjednika.

Tree

Stablo je nelinearna struktura podataka koja se sastoji od čvorova povezanih relacijom roditelj–dete.

Počinje od **korena** (*root*), a svaki čvor može imati više potomaka.



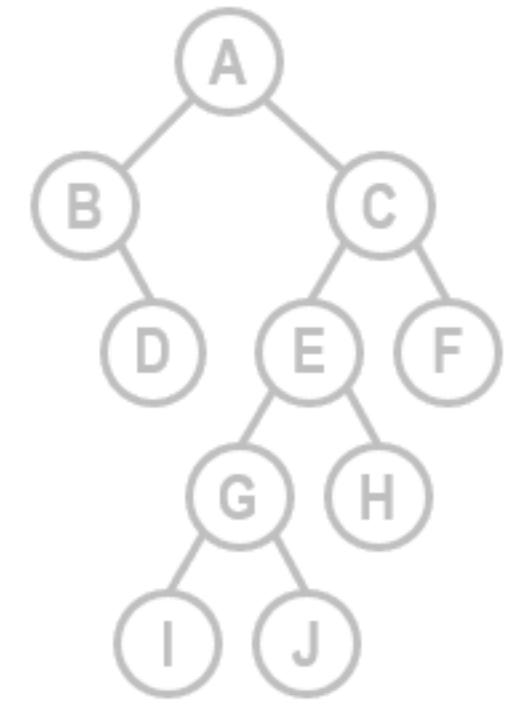
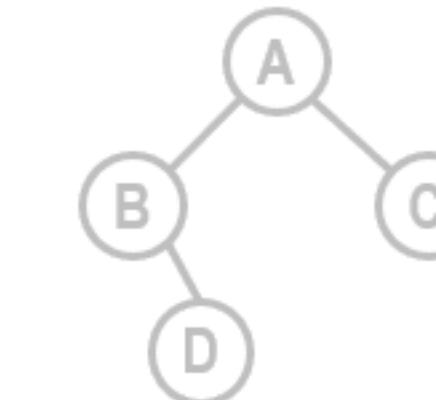
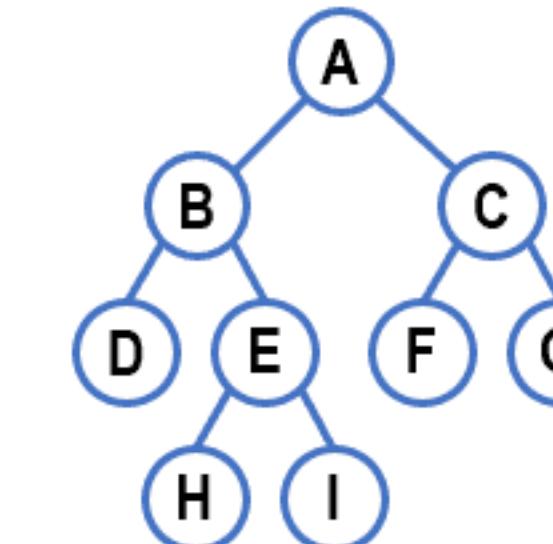
Analogija iz stvarnog sveta:

- ▶ Porodično stablo, Hijerarhija u firmi, Fajl sistem (folderi unutar foldera)



Primene u programiranju

- ▶ **Algoritmi odlučivanja** (*decision trees*)
- ▶ Brza **pretraga i sortiranje**
- ▶ **Organizacija podataka** u bazama



! Zašto je bitno?

- ▶ Omogućava **efikasnu hijerarhijsku organizaciju podataka**
- ▶ Osnova za **mnoge napredne sturkture** (*heap, tree, ...*)
- ▶ Ima **prirodnu primenu** u rešavanju mnogih realnih problema

Kako implementirati Tree u Pythonu?

- ▶ Koristićemo **binarno stablo**, gde svaki čvor može imati **najviše dva potomka**: levog i desnog

Node klasa:

```
class Node:  
    def __init__(self, value):  
        self.value = value  
        self.left = None  
        self.right = None
```

- ▶ Svaki čvor čuva vrednost (value) i reference ka **levom** i **desnom detetu**
- ▶ Prazni pokazivači (None) znače da čvor nema potomke

BinaryTree klasa:

```
class BinaryTree:  
    def __init__(self, root_value):  
        self.root = Node(root_value)
```

- ▶ Klasa BinaryTree služi za **rad sa stablom kao celinom**
- ▶ Može sadržati metode za **umetanje, pretragu, obilazak** itd.

Zašto koristimo ovu strukturu?

- ▶ Intuitivna reprezentacija **hijerarhijskih odnosa**
- ▶ Laka nadogradnja **dodatnim metodama** (npr. preorder obilazak)
- ▶ Osnova za složenije strukture: **BST, Heap, AVL...**

Osnovne metode

```
def insert(self, value):
    def _insert(node, value):
        if not node:
            return Node(value)
        if value < node.value:
            node.left = _insert(node.left, value)
        else:
            node.right = _insert(node.right, value)
        return node

    self.root = _insert(self.root, value)
```

- ▶ **Dodaje čvor** na odgovarajuću poziciju u skladu sa BST pravilima
- ▶ Vremenska složenost: $O(\log n)$ (prosečno)

```
def preorder(self, node):
    if node:
        print(node.value)
        self.preorder(node.left)
        self.preorder(node.right)
```

```
def search(self, value):
    current = self.root
    while current:
        if value == current.value:
            return current
        elif value < current.value:
            current = current.left
        else:
            current = current.right
    return None
```

- ▶ **Pronalaženje vrednosti** u stablu
- ▶ Vremenska složenost: $O(\log n)$

- ▶ Obliazak u redu: **korenski → levo → desno**
- ▶ Koristi se često za serijalizaciju stabla
- ▶ Vremenska složenost: $O(n)$

Zadatak: Sortirani telefonski kontakti

Zadatak

- ▶ Napravi program koji čuva kontakte kao stringove (*imena*) koristeći **BST**.
- ▶ Umetni imena u BST, a zatim ih **ispisi po abecednom redu** koristeći **inorder obilazak**.

Koraci

1. Kreiraj klasu `Node` sa vrednošću, levim i desnim detetom.
2. Napravi klasu `BinaryTree` sa metodom `insert()` koja umeće nova imena u *BST*.
3. Napravi metodu `inorder()` koja obilazi stablo i ispisuje imena **po abecedi**.

Složenost:

- ▶ Umetanje: $O(\log n)$ u proseku
- ▶ Inorder obilazak: $O(n)$