

# ALGORITMI IN PODATKOVNE STRUKTURE

## REKURZIJA

### OSNOVNE INFORMACIJE:

- ponavljanje lahko izvajamo z **iteracijo** ali pa **rekurzijo**
- **iteracija**: zanke
- **rekurzija**: program kliče samega sebe

### PRIMER: Fibonacci

```
fib(1) = 1
fib(2) = 1
fib(n) = fib(n - 1) + fib(n - 2) za n > 2
```

### PRIMER: izračun izraza

- vzamemo **binarno izrazno drevo**
- rekurzivna spremenljivka: **koren**
- **robni pogoj**: list število vrnemo

$$v(\text{operator}) = v(T_1) \text{ operator } v(T_2)$$

### PRIMER: izračun fakultete

- rekurzivna spremenljivka: **n**
- **robni pogoj**: **n = 0**
- **splošni primer**:  $I_n = n \cdot (n - 1)!$

### PRIMER: potenciranje števila

$$\begin{aligned} \text{potenca}(x, 0) &= 1 \\ \text{potenca}(x, p) &= x * \text{potenca}(x, p-1) \text{ za } p > 0 \end{aligned}$$

### PRIMER: maksimalno število

- rekurzivna spremenljivka: **n**
- **robni pogoj**: **n = 1**
- **splošni primer**:  $\max(1) = \text{element}[1]$
- $\max(n) = \text{maximum}(\max(n-1), \text{element}[n])$

### PRIMER:

#### višina binarnega drevesa

- rekurzivna spremenljivka: **koren**
- **robni pogoj**: koren je null
- ko koren null vrnemo 0
- **splošni pogoj**:

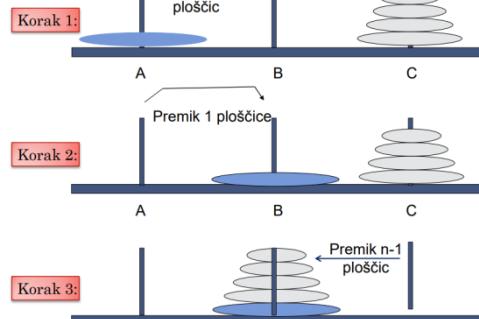
$$v(T) = \max(v(T_1), v(T_2)) + 1$$

### PRIMER: Hanojski stolpi

- vse diske postavimo iz **a** na **b** palico
- rekurzivna spremenljivka: število diskov na **a**
- **robni primer**: **a** ima 0 diskov
- **splošni**: najprej vse na **c** in potem na **b**

Nobena ploščica ne sme biti nikoli na vrhu manjše ploščice.  
Naenkrat smemo premikati le po eno ploščico.

Vsako ploščico moramo vedno odložiti na eno od palic, nikoli ob strani.  
Premaknemo lahko vedno le ploščico, ki je na vrhu nekega stolpča.



### sklad in rekurzija

- **dodajamo na vrh**
- **brišemo iz vrha**

- MAKENULL(S) – naredi prazen sklad
- EMPTY(S) – ali je sklad prazen
- TOP(S) – vrne vrhni element sklada
- PUSH(x, S)– vstavi element x na vrh sklada
- POP(S) – zbrise vrhni element sklada

vsako repno rekurzijo lahko  
zamenjamo z iterativno zanko !

### Kaj ima to z rekurzijo dude?

- **pomnilniško** rekurzija **bolj** zahtevna
- shranjujemo na **skladu**
- potrebnata velikost sklada je **globina**
- rekurzivna koda **krajša**

### REPNA REKURZIJA:

- rekurzija se pojavi **na koncu** metode
- ko se izvajanje rekurzije konča se **konča** tudi izvajanje **procedure**
- **iteracija**: samo cela od začetka

```
static public void hanoi(char A, char B, char C,int n) {
    if(n>0) {
        hanoi(A,C,B,n-1);
        System.out.println("premik iz " + A + " na " + B);
        hanoi(C,B,A,n-1);
    } // if
} // hanoi
    prevedba repne
    rekurzije na iteracijo
```

```
static public void hanoi0tail(char A, char B, char C,int n) {
    char T;
    while(n>0) {
        hanoi(A,C,B,n-1);
        System.out.println("premik iz " + A + " na " + B);
        // priprava argumentov za ponovitev (rekurzivni klic)
        T=A; // zamenjamo zelbla A in C
        A = C ;
        C = T ;
        n=n-1 ;
    } // while
} // hanoi0tail
```

to po navadi  
naredijo boljši  
prevajalniki

na sklad in general spravimo  
**argumente**, **lokalne spremenljivke**  
in **naslov** za nadaljevanje

počnemo samo če lahko algoritem **optimiziramo**  
**boljše** kot prevajalnik

## ČASOVNA KOMPLEKSNOT

### OSNOVNI POJMI:

- **asimptotična zgornja meja** **O**
- **spodnja meja** **Ω**
- **stroga meja** **θ**

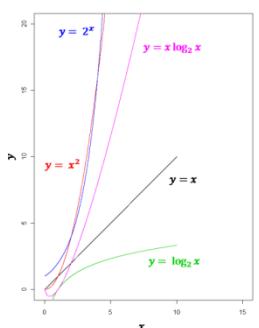
### zgornja meja časovne kompleksnosti:

$$T(n) = O(g(n)) \Leftrightarrow \exists c, n_0 > 0 : n > n_0 \rightarrow c \cdot g(n) \geq T(n) \geq 0$$

pri dovolj velikem **n** je kompleksnost našega

programa **navzgor** omejena s funkcijo **g(n)** in **c** je poljubna **konstanta**

$$\log n, n, n \log n, n^2, n^3, n^4, 2^n, n!, n^n$$



## računanje zgornje meje

eliminacija konstante

$$c > 0 \rightarrow O(c \cdot f(n)) = O(f(n))$$

vsota

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

prevladujoča funkcija

$$\forall n > n_0 : f(n) > g(n) \rightarrow O(f(n)) + O(g(n)) = O(f(n))$$

produkt

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

tranzitivnost

$$f(n) = O(g(n)), g(n) = O(h(n)) \rightarrow f(n) = O(h(n))$$

refleksivnost

$$f(n) = O(f(n))$$

Ocena dejanskega časa izvajanja?

→ pri **oceni velikostnega reda** časovne zahtevnosti zanemarimo vse člene **nižjega reda** kot tudi vse **konstante** ki lahko v realnosti zelo spremenijo stanje

→ predpostavimo neko **odvisnost** med oceno in dejanskim časom izvajanja

$$T(n) = a * O(g(n)) + c$$

### PRIMERI:

$$17n^3 + 93n^2 + n = O(n^3)$$

$$2n^2 + n \log n = O(n^2)$$

$$3n + n \log n = O(n \log n)$$

$$2^n + 739n^9 = O(2^n)$$

$$(n + \sqrt{n}) \cdot (n \log n + n + 3) = O(n^2 \log n)$$

1. osnovne operacije:  $O(1)$

2. pri zaporedju ukazov seštevamo zahtevnosti

3. pri pogojih štejemo kompleksnost izračuna pogoja in maksimum vseh možnih izbir

4. pri zankah seštejemo kompleksnost izračuna pogoja in enkratne izvedbe zanke ter pomnožimo s številom izvajanja zanke

5. pri rekurziji izrazimo zahtevnost kot rekurenčno enačbo

ampak če je n manjši od 10000 je drugi algoritmom primernejši

polinomsko:  $T(n) = c n^e$   
eksponentno:  $T(n) = c 2^{en}$

### PRIMER:

velikost podatkov	5	6	7	8	9	10
čas	13.8	83.6	388.6	1796.2	8753.6	44421.4

→ zanima nas kako hitro raste funkcija  
→ iščemo konstante ki so vsaj **do neke mere konstante**  
→ vidimo da narašča **eksponentno**

f(n)	f(n+1) - f(n) = O(g(n))
$\log n$	$\log(n+1) - \log n = O\left(\frac{1}{n}\right)$
$n$	$1 = O(1)$
$n \log n$	$(n+1) \log(n+1) - n \log n = O(\log n)$
$n^2$	$2n+1 = O(n)$
$n^3$	$3n^2 + 3n + 1 = O(n^2)$
$n^4$	$4n^3 + 6n^2 + 4n + 1 = O(n^3)$
$2^n$	$2^n = O(2^n)$
$n!$	$n \times n! = O((n+1)!)$

### NEENAKOSTI

$$a > b > 0, c > 0 \rightarrow n^a > cn^b$$

$$a > 0, b > 1, c > 0 \rightarrow n^a > c \log_b n$$

$$a > 1, b > 0, c > 0 \rightarrow a^n > cn^b$$

$$a > 1, c > 0 \rightarrow n! > ca^n$$

$$c > 0 \rightarrow n^n > cn!$$

f(n)	f(n+1) - f(n) = O(g(n))	povečanje velikosti rešljivega problema v danem času z $10 \times$ hitrejšim rač.
$\log n$	$\log(n+1) - \log n = O\left(\frac{1}{n}\right)$	$n^{10}$
$n$	$1 = O(1)$	$10 n$
$n \log n$	$(n+1) \log(n+1) - n \log n = O(\log n)$	$< 10n$
$n^2$	$2n+1 = O(n)$	$3.16n$
$n^3$	$3n^2 + 3n + 1 = O(n^2)$	$2.15n$
$n^4$	$4n^3 + 6n^2 + 4n + 1 = O(n^3)$	$1.78n$
$2^n$	$2^n = O(2^n)$	$\leq n+4$
$n!$	$n \times n! = O((n+1)!)$	$\leq n+1$

## ABSTRAKTNI PODATKOVNI TIP

### PODATKOVNI TIP:

- **podatkovni tip** definira množico možnih **vrednosti** objektov tega tipa in vse možne **operacije**
- pri **strukturiranih** podatkovnih objektih se operacije na delih strukturi dedujejo
- prožno programiranje
- nadzor programerja ne prevajalnika, ker se rabimo zavedati, kako zadeva **implementirana**
- veliko napak zaradi **nenadzorovanih** dostopov do podatkov

**seznam (list)** – zbirka elementov, ki se lahko *ponavlja*; vrstni red elementov je *pomemben*

**množica (set)** – zbirka elementov, kjer vrstni red *ni pomemben*; elementi se *ne ponavljajo*

**vrsta (queue)** – zbirka, kjer elemente vedno dodajamo na konec vrste in jih vedno brišemo na začetku vrste

**sklad (stack)** – zbirka, kjer se elementi dodajajo in brišejo vedno na vrhu (enem koncu) sklada

**preslikava (map)** – vsakemu elementu  $d$  iz domene priredi ustrezni element  $r$  iz zaloge vrednosti

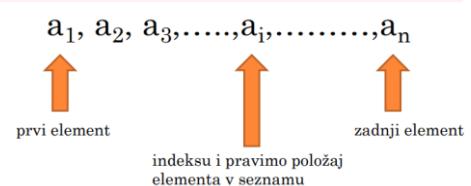
### ABSTRAKTNI PODATKOVNI TIP:

- definira **strukturo podatkovnega objekta**
- množico možnih **vrednosti** za vsak del strukture
- vse možne **operacije** na objektih tega tipa
- ni **avtomsatskega** dedovanja
- **prednosti**: varnost in hitrost programiranja
- modul, paket, razred objektov
- 

### SEZNAM

#### SEZNAM BASICS:

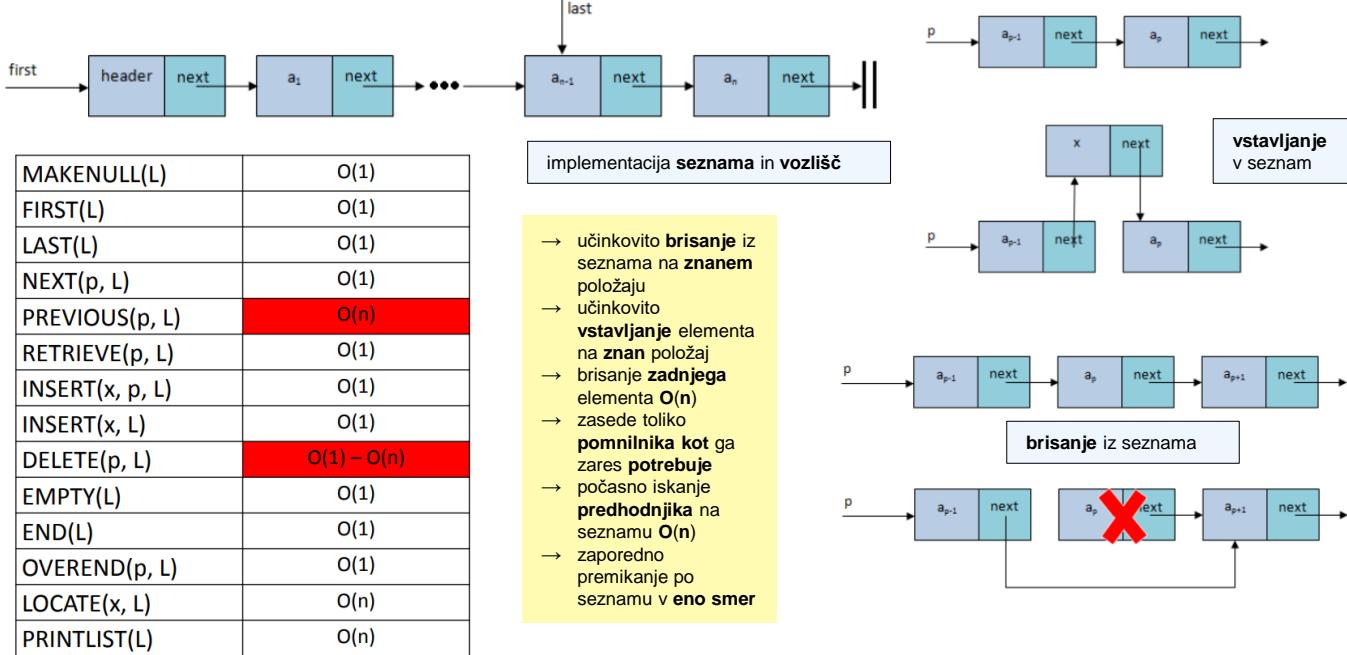
- zaporedje **0** ali **več** elementov
- vrstni red elementov je **pomemben**
- elementi v seznamu se lahko **ponavlja**



#### IMPLEMENTACIJE:

- enosmerni seznam s kazalci
- dvosmerni seznam s kazalci
- implementacija s poljem
- z indeksnimi kazalci

- MAKENULL(L) – naredi prazen seznam L
- FIRST(L) – vrne položaj prvega elementa v seznamu
- LAST(L) – vrne položaj zadnjega elementa v seznamu
- NEXT(p, L) – vrne naslednji položaj položaja p
- PREVIOUS(p, L) – vrne predhodni položaj položaja p
- RETRIEVE(p, L) – vrne element  $a_p$  na položaju p
- INSERT(x, p, L) – vstavi element x na položaj p
- INSERT(x, L) – vstavi element x na poljuben položaj
- DELETE(p, L) – zbrise element  $a_p$  na položaju p
- EMPTY(L) – preveri, če je seznam prazen
- END(L) – vrne položaj, ki sledi zadnjemu elementu se
- OVEREND(p, L) – preveri, če je p = END(L)
- LOCATE(x, L) – poišči položaj elementa x v seznamu
- PRINTLIST(L) – po vrsti izpiše vse elemente seznama



## dvosmerni seznam

- učinkovito **iskanje predhodnika** O(n)  
→ ima **2 kazalca**, na **predhodni** in **naslednji** element  
→ **nima zamknjenih** kazalcev

MAKENULL(L)	O(1)
FIRST(L)	O(1)
LAST(L)	O(1)
NEXT(p, L)	O(1)
PREVIOUS(p, L)	O(1)
RETRIEVE(p, L)	O(1)
INSERT(x, p, L)	O(1)
INSERT(x, L)	O(1)
DELETE(p, L)	O(1)
EMPTY(L)	O(1)
END(L)	O(1)
OVEREND(p, L)	O(1)
LOCATE(x, L)	O(n)
PRINTLIST(L)	O(n)

**class ListTwoWayLinkedNode {**  
Object element;  
**ListTwoWayLinkedNode next, previous;**

```
public class ListTwoWayLinked {
    protected ListTwoWayLinkedNode first, last;
    ...
}
```

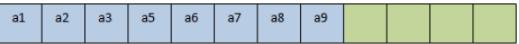
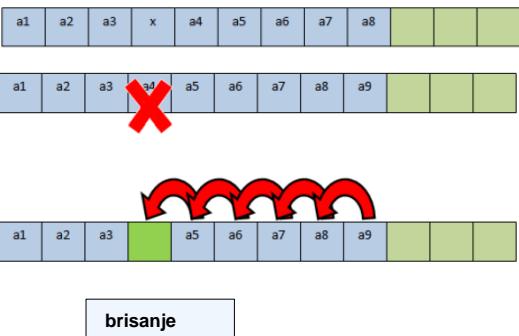
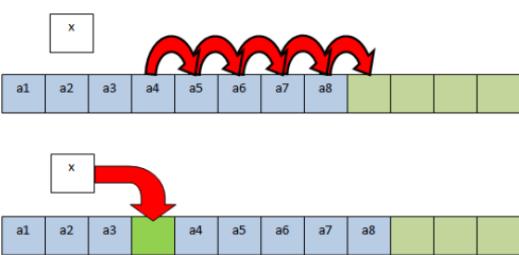
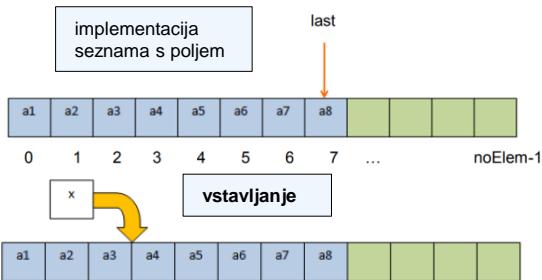
first → prev a1 next → prev a2 next → ... → prev ap next → last

**seznam s poljem**

- uporablja polje za shranjevanje elementov
- položaj elementa z **indeksom** polja
- imamo indeks **zadnjega** elementa v seznamu

```
public class ListArray {
    private Object elements[];
    private int lastElement;
    ...
    public ListArray(int noElem) {
        elements = new Object[noElem];
    }
}
```

→ primerna za implementacije kjer se seznam **zelo malo spreminja**  
→ učinkovito če sprostimo zahtevo o **vrstnem redu** elementov



MAKENULL(L)	O(1)
FIRST(L)	O(1)
LAST(L)	O(1)
NEXT(p, L)	O(1)
PREVIOUS(p, L)	O(1)
RETRIEVE(p, L)	O(1)
INSERT(x, p, L)	O(n)
INSERT(x, L)	O(1)
DELETE(p, L)	O(n)
EMPTY(L)	O(1)
END(L)	O(1)
OVEREND(p, L)	O(1)
LOCATE(x, L)	O(n)
PRINTLIST(L)	O(n)

Prednosti:

- preprosta implementacija
- načeloma dobre časovne zahtevnosti

Slabosti:

- neučinkovito vstavljanje
- neučinkovito brisanje
- omejena dolžina
- ves čas zaseda maksimalno dolžino pomnilnika

### seznam z indeksnimi kazalci

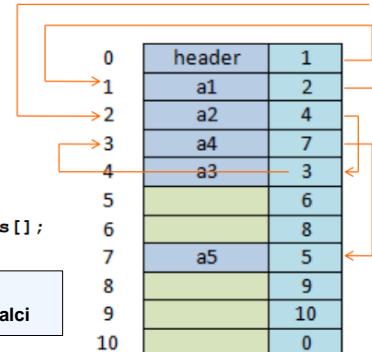
- uporabljamo tam kjer niso omogočene dinamične podatkovne strukture
- vsaka celica polja je sestavljena iz elementa in iz indeksa naslednjega elementa

```
class ListCursorNode {
    Object element;
    int next;
    ...
}
```

položaj je spet zamknjen

```
public class ListCursor {
    private ListCursorNode cells[];
    private int last;
}
```

programer skrbi za dodeljevanje in čiščenje pomnilnika drugače podobno kot seznam z kazalci



imamo dva seznama, prvi ima vse elemente in drugi povezuje vse prazne celice v polju

- učinkovito brisanje iz seznama na znanem položaju
- učinkovito vstavljanje elementa na znan položaj
- brisanje zadnjega elementa O(n)
- zasede toliko pomnilnika kot ga zares potrebuje
- počasno iskanje predhodnika na seznamu O(n)
- zaporedno premikanje po seznamu v eno smer

- počasna inicializacija O(n)
- programer sam skrbi
- maksimalna velikost mora bit podana v naprej
- zaseda maksimalno količino pomnilnika

## MNOŽICA

- MAKENULL(S) – naredi prazno množico S
- FIRST(S) – vrne položaj prvega elementa v množici S
- NEXT(p, S) – vrne naslednji položaj položaja p
- RETRIEVE(p, S) – vrne element  $a_p$  na položaju p
- INSERT(x, S) – vstavi element x v množico S brez podvajanja
- DELETE(p, S) – zbrise element  $a_p$  na položaju p
- EMPTY(S) – preveri, če je množica prazna
- OVEREND(p, S) – preveri, če je p položaj, ki sledi zadnjemu elementu množice
- LOCATE(x, S) – poiše položaj elementa x v množici S
- PRINTSET(S) – po vrsti izpiše vse elemente množice S
- UNION(S1, S) – v množico S doda (brez podvajanja) vse elemente iz množice S1
- INTERSECTION(S1, S) – iz množice S zbrise vse elemente, ki se ne nahajajo tudi v S1

MNOŽICA BASICS:

- zaporedje 0 ali več elementov
- vrstni red elementov ni pomemben
- elementi v seznamu se ne ponavljajo

MAKENULL(L)	O(cells.length)
FIRST(L)	O(1)
LAST(L)	O(1)
NEXT(p, L)	O(1)
PREVIOUS(p, L)	O(n)
RETRIEVE(p, L)	O(1)
INSERT(x, p, L)	O(1)
INSERT(x, L)	O(1)
DELETE(p, L)	O(1) ... O(n)
EMPTY(L)	O(1)
END(L)	O(1)
OVEREND(p, L)	O(1)
LOCATE(x, L)	O(n)
PRINTLIST(L)	O(n)

Eno dodajanje elementa v S brez podvajanja: O(n)  
m dodajanj elementa v S brez podvajanja: m X O(n) = O(mn)

- implementacija lahko z enosmernimi seznamom s kazalci
- ohranimo kazalec last ne rabimo tho
- ohranimo položaj elementa čeprav ne rabimo again
- pri dodajanju pazimo da niso podvojeni

Eno brisanje elementa: element poiščemo O(n) plus zbrisemo O(1)...O(n)  
m brisanj elementa: m krat iskanje plus min(m,n) krat brisanje:  
 $m \times O(n) + \min(m,n) \times [O(1) ... O(n)] = O(mn) ... O((m+n)n)$

MAKENULL(S)	O(1)
FIRST(S)	O(1)
NEXT(p, S)	O(1)
RETRIEVE(p, S)	O(1)
INSERT(x, S)	O(n)
DELETE(p, S)	O(1) ... O(n)
EMPTY(S)	O(1)
OVEREND(p, S)	O(1)
LOCATE(x, S)	O(n)
PRINTSET(S)	O(n)
UNION(S1, S)	O(mn)
INTERSECTION(S1, S)	O(min) ... O((m+n)n)



- insert in locate učinkoviti  $O(\log n)$
- union:  $O(m \cdot \log n)$
- intersection:  $O(m \cdot \log n)$
- hitre operacije vezane na urejenost

- insert:  $O(n)$
- locate:  $O(n)$
- union in intersection:  $O(mn)$
- neučinkovite operacije vezane na urejenost elementov

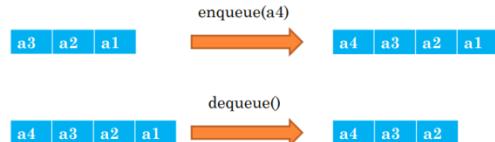
- insert in locate učinkoviti  $O(1)$
- union:  $O(m)$
- intersection:  $O(m)$
- vse velja če zgoščevalna funkcija enakomerno razvrsti elemente

## VRSTA

### VRSTA BASICS:

- dodajamo na konec
- brišemo na začetku
- sistem **FIFO**: first in first out

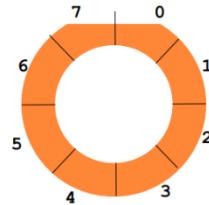
- MAKENULL(Q) – naredi prazno vrsto
- EMPTY(Q) – ali je vrsta prazna
- FRONT(Q) – vrne prvi element v vrsti
- ENQUEUE(x, Q) - vstavi element x na konec vrste
- DEQUEUE(Q) – zbrise prvi element iz vrste



### IMPLEMENTACIJA:

- enosmerni seznam kjer položaj ni zamaknjen
- krožno polje: vrsta se krožno premika po polju

problem krožnega polja da velikost omejena in ves čas zaseda maksimalno pomnilnika



MAKENULL(Q)	O(1)
EMPTY(Q)	O(1)
FRONT(Q)	O(1)
ENQUEUE(x, Q)	O(1)
DEQUEUE(Q)	O(1)

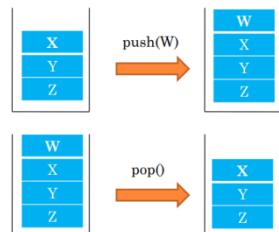
## SKLAD

### SKLAD BASICS:

- dodajamo na vrh
- brišemo iz vrha
- sistem **LIFO**: last in first out

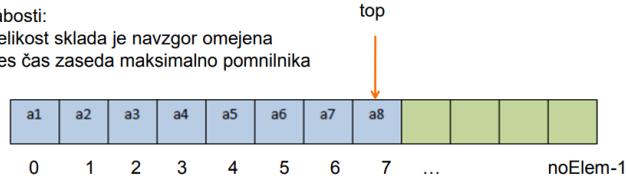
- MAKENULL(S) – naredi prazen sklad
- EMPTY(S) – ali je sklad prazen
- TOP(S) – vrne vrhni element sklada
- PUSH(x, S) - vstavi element x na vrh sklada
- POP(S) – zbrise vrhni element sklada

MAKENULL(S)	O(1)
EMPTY(S)	O(1)
TOP(S)	O(1)
PUSH(x, S)	O(1)
POP(S)	O(1)



Slabosti:

- velikost sklada je navzgor omejena
- ves čas zaseda maksimalno pomnilnika



### IMPLEMENTACIJA:

- enosmerni seznam kjer položaj ni zamaknjen
- polje: dodajamo in brišemo na koncu oz. vrhu

## PRESKOČNI SEZNAM

→ želimo nadgradit povezan seznam, da imamo hitrejsše iskanje

→ naredimo ekspresivni nivo

→ dobimo čas iskanja  $2\sqrt{n}$

→ lahko dodamo še en nivo

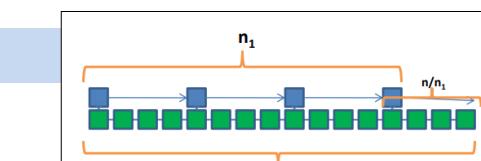
→ koliko nivojev dodamo?

$$k = \log_2 n$$

### brisanje:

→ poiščemo in zbrisemo na vseh nivojih

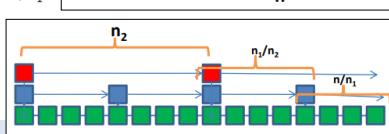
→ enaka časovna kompleksnost kot vstavljanje



$$\text{Število korakov} = n_2 + n_1/n_2 + n/n_1$$

Minimizirajmo!

$$\text{Rešitev: } n_1 = \frac{n}{n_2} = \sqrt{n}$$



vsak element ima polje kazalcev  
višina novega elementa naključna!

### vstavljanje:

→ poiščemo element

→ za vsak nivo si zapomnimo zadnji obiskani element

→ vstavimo v prvi nivo

→ ponavljamo

→  $O(\log n)$  da najdemo in še enkrat toliko da vstavimo

# PRESLIKAVA

## MAP BASICS:

- vsakemu elementu iz **domene** priredi ustrezni element iz **zaloge** vrednosti
- lahko uporabimo **seznam**: časovna zahtevnost iskanja parov je po navadi previsoka **O(n)**
- uporabimo **iskalno drevo**: **O(log n)**
- uporabimo **polje**: indeksi so **domene**, hitre **dodajanje, iskanje in brisanje O(1)**

**MAKENULL(M)** - inicializira prazno preslikavo

**ASSIGN(M, d, r)** - definira, da je  $M(d) = r$

**COMPUTE(M, d)** - vrne vrednost  $M(d)$ , če je definirana, sicer **null**

## zgoščena tabela

### HASH MAP:

- domena lahko **infinite**
- **zgoščevalna funkcija** slika **originalno domeno** v **manjšo**
- **razprši** elemente

### HASH FUNCTION:

- **velikost polja**: ne sme zasest **preveč** pomnilnika vendar mora biti **dovolj velika**
- po navadi po **modusu**
- idealno da je **injektivna**

## PRIMER: hash function

Lahko uporabimo zgoščevalno funkcijo:  
 $h(\text{tel\_st}) = \text{tel\_st} \bmod 17$

Telefonska številka	Lastnik računa
031 - 456 876	Janez N.
031 - 345 987	Miha K.
041 - 237442	Tine B.
040 - 327 896	Janez N.
070 - 213 445	Tina Z.
040 - 743 210	Marko S.
070 - 236 580	Teja B.

h(031456876) = 8
h(031345987) = 10
h(041237442) = 15
h(040327896) = 3
h(070213445) = 11
h(040743210) = 7
h(070236580) = 9

z uporabo zgoščevalne funkcije preslikamo indekse iz originalne domene v indekse manjše domene

0	-
...	-
3	Ana H.
...	-
7	Marko S.
8	Janez N.
9	Teja B.
10	Miha K.
11	Tina Z.
...	-
15	Tine B.
16	-

ne more biti **dinamična** struktura, ker je zgoščevalna funkcija **fiksna**

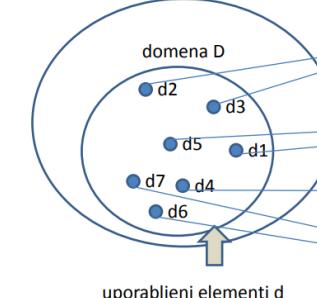
najslabši primer je če vsi elementi sovpadajo in imamo **seznam**

$$h(x) = x \bmod m$$

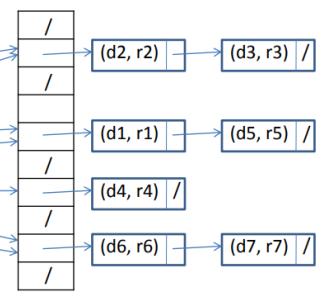
m je  
praštevilo  
večje od  
potenčne  
množice

reševanje omejenosti:  $d_1 \neq d_2 \wedge h(d_1) = h(d_2)$

- ko se napolni: **novi večjo** z novo **funkcijo**
- **reshashing**  $O(n)$
- **sovpadanje** v praksi neizogibno
- **zaprti zgoščeni tabeli**: če funkcija sovпадa gremo na drugo funkcijo, lahko počasno
- **odprtih**: v polju hranimo kazalce na pare ki sovpadajo



uporabljeni elementi d



odprtih tabel

Pričakovana zahtevnost iskanja elementa je  $O(n/m)$ .

če hash funkcija dobro zbrana je zadeva really fast skoraj vse operacije  $O(1)$

$n$  - število vstavljenih elementov  
 $m$  - velikost zgoščene tabele.

- ✓ **Zaprti zgoščeni tabela** zaseda manj pomnilnika
- ✓ **Odprtih zgoščeni tabela** je bolj dinamična/fleksibilna
- ✓ vstavljanje v **odprtih zgoščeni tabeli** vedno  $O(1)$

## JAVANSKE ZBIRKE

### implementacija

vrmesnik	seznam	hashset	hashmap
	zaprti zgoščeni tabela, ponovno zgoščanje	ArrayList	polje
Seznam		(primerno, ko se seznamami malo spreminja in potrebujemo pogoste neposredne dostope do elementov)	povezan seznam s kazalci
Množica	Hashset NEUREJENA (primerno, ko potrebujemo hitre operacije vstavljanja, brisanja in iskanja elementov)	ArrayList	rdeče-črno drevo
Preslikava	HashMap NEUREJENA (primerno, ko potrebujemo hitre operacije vstavljanja, brisanja in iskanja elementov)	TreeSet UREJENA (primerno, ko potrebujemo operacije vezane na urejenost elementov)	TreeMap UREJENA (primerno, ko potrebujemo operacije vezane na urejenost elementov)

### IMPLEMENTACIJA:

- **ArrayList**: seznam s **poljem**
- **LinkedList**: seznam s **kazalci**
- **HashSet**: neurejena množica implementirana z **zaprti zgoščeno tabelo** s ponovnim zgoščanjem
- **TreeSet**: urejena množica implementirana z **rdeče črnim drevesom**
- **HashMap**: preslikava in ista struktura za implementacijo kot **Hashset**
- **TreeMap**: preslikava implementirana kot **TreeSet**

# DREVO

## TREE BASICS:

- zbirka hierarhično **urejenih vozlišč**
- vozlišča imajo relacijo **oče sin**
- **vozlišče**: oznaka, oče, če obstaja sin je to levi sin
- **koren** nima očeta
- implementacija z **polji** ali **kazalci**

če imamo **izrojeno** drevo:  
Stopnja = 1, višina = n

- PARENT( $n, T$ ) - vrne očeta vozlišča  $n$  v drevesu  $T$
- LEFTMOST\_CHILD( $n, T$ ) - vrne najbolj levega sina vozlišča  $n$
- RIGHT\_SIBLING( $n, T$ ) - vrne desnega brata vozlišča  $n$
- LABEL( $n, T$ ) - vrne oznako vozlišča  $n$
- ROOT( $T$ ) - vrne koren drevesa  $T$
- MAKENULL( $T$ ) - naredi prazno drevo  $T$
- CREATE( $r, v, T_1, \dots, T_k$ ) - generira drevo s korenom  $r$  z oznako  $v$  in stopnjo  $k$  s poddrevesi  $T_1, \dots, T_k$

## terminologija dreves

notranje vozlišče (*internal node*):  
vozlišče z vsaj enim sinom

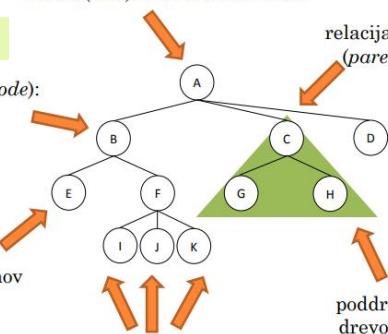
vrstica: **nivo**

list (*leaf*): vozlišče brez sinov

če imamo **uravnoteženo** drevo:  
Stopnja =  $k \rightarrow$  višina =  $\log_k n$

koren (*root*): vozlišče brez očeta

relacija oče-sin (*parent-child*)



poddrevo (*subtree*):  
drevo s korenom v danem vozlišču

**stopnja (degree) vozlišča**: število sinov vozlišča (npr. stopnja vozlišča C je 2)

**stopnja drevesa**: največje št. sinov nekega vozlišča (npr. stopnja drevesa na sliki je 3)

**pot (path)**: zaporedje  $n_1, n_2, \dots, n_k$ , pri čemer velja za vsak  $1 \leq i < k$ , da je  $n_i$  oče  $n_{i+1}$   
(npr. A, B, F, J)

( $j > i$ ):  $n_j$  je prednik (*ancestor*) vozlišča  $n_j$ , oz.  $n_j$  so nasledniki (*descendants*) vozlišča  $n_j$ . Dolžina poti (*path length*) je enaka številu vozlišč na poti.

**nivo (level) vozlišča**: dolžina poti od korena do vozlišča (npr. nivo vozlišča E je 3)

**višina (height) drevesa**: dolžina najdaljše poti od korena do lista (npr. za zgornje drevo je 4)

## OBHOD DREVESA:

- **preorder** oz. **prem**: oznako korena pred oznakami poddreves
- **postorder** oz. **obratni**: obratno najprej oznake poddreves nato oznake korena
- **inorder** oz. **vmesni**: levo poddrevo, koren, desno
- **nivojski**: izpišemo po nivojih

### premi obhod

```
public void preorder(TreeNode r) {
    if (r != null) {
        // najprej izpišemo oznako korena
        r.writeLabel(); System.out.print(", ");
        TreeNode n = leftmostChild(r);
        while (n != null) {
            preorder(n);
            n = rightSibling(n);
        } // while
    } // if
} // preorder
```

### obratni obhod

```
public void postorder(TreeNode r) {
    if (r != null) {
        TreeNode n = leftmostChild(r);
        while (n != null) {
            postorder(n);
            n = rightSibling(n);
        } // while
        // nazadnje izpišemo oznako korena
        r.writeLabel(); System.out.print(", ");
    } // if
} // postorder
```

### vmesni obhod

```
public void inorder(TreeNode r) {
    if (r != null) {
        TreeNode n = leftmostChild(r);
        inorder(n);
        // za levim poddrevesom izpišemo oznako korena
        r.writeLabel(); System.out.print(", ");
        n = rightSibling(n);
        while (n != null) {
            inorder(n);
            n = rightSibling(n);
        } // while
    } // if
} // inorder
```

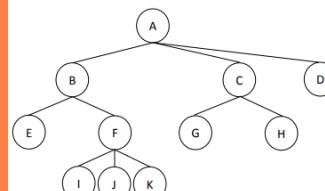
### obhod po nivojih

```
private boolean printLevel(int l, TreeNode r) {
    if (r == null)
        return false;
    else if (l == 1) {
        r.writeLabel(); System.out.print(", ");
        return true;
    }
    else {
        TreeNode n = leftmostChild(r);
        boolean existsLevel = false;
        while (n != null) {
            // izpis nivoja je možen v vsaj enem poddrevesu
            existsLevel |= printLevel(l-1, n);
            n = rightSibling(n);
        }
        return existsLevel;
    }
} // printLevel
```

## IMPLEMENTACIJA Z POLJEM:

- vsako vozlišče hrani **število sinov**
- celo drevo hrani **število vozlišč**
- koren drevesa je **prvi element** polja
- sledijo **vsa vozlišča** levega poddrevesa
- redko se uporablja ker ni **dinamično**
- problematično **spreminjanje**
- operacije **O(1)** razen ko ustvarimo drevo in ko iščemo **desnega brata O(n)** in **O(k)** kjer **k** število vozlišč **poddrevesa**
- operacije učinkovite za **levo poravnano drevo**

povečujemo nivo dokler se še kaj izpiše,  
števec teče po **koren** in **nivoju**



**A B E F I J K C G H D**  
**E I J K F B G H C D A**  
**E B I F J K A G C H D**  
**A B C D E F G H I J K**

## binarno drevo

- **vozlišče**: stopnja največ 2
- lahko ima tudi samo **desnega** sina
- **največ vozlišč** možnih:  $2^{višina} - 1$
- **n** vozlišč ima **n + 1** praznih poddreves

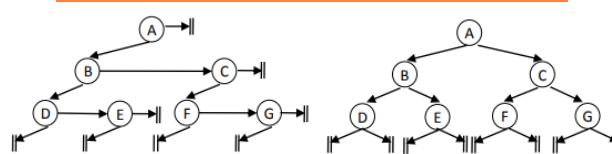
$n \geq v \geq \lceil \log_2(n+1) \rceil$  ocena **višine** drevesa

A	3
B	2
E	0
F	3
I	0
J	0
K	0
C	2
G	0
H	0
D	0

**koren**  
**prvo poddrevo**  
**drugo poddrevo**  
**tretje poddrevo**

**IMPLEMENTACIJA Z KAZALCI:**

- vsako vozlišče vsebuje kazalca na **levega otroka** in **desnega brata**
- vsako vozlišče vsebuje kazalce na **vse otroke**, po navadi za **binarna** drevesa
- drevo podano s kazalcem na **koren** drevesa
- vse operacije imajo **O(1)**



## IZRAZNA DREVESA:

- poznati moramo strukturo vseh možnih izrazov
- **leva asociativnost**
- **prioriteta operatorjev**

$$a+b+c = (a+b)+c \quad a+b*c = a+(b*c)$$

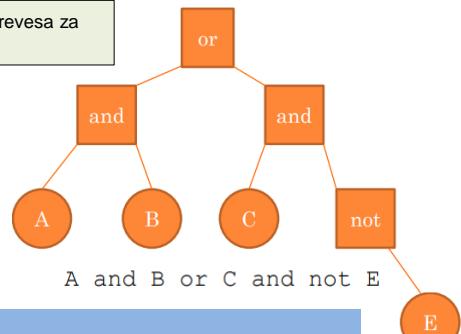
**kontekstno neodvisne gramatike:**

- opisuje vse možne aritmetične izraze
- množica pravil, ki bodo iz začetne spremenljivke generirala poljuben aritmetični izraz

### sintaktična analiza:

- izraz beremo na vhodu od leve proti desni
- beremo simbol po simbol
- simbol je lahko: operator, oklepaj ali število
- za vsako spremenljivko uvedemo eno metodo
- med sintaktično analizo lahko gradimo izrazno drevo
- vhod v metodo dobri tudi dele za izgradnjo poddrevesa in vrne koren za to drevo

primer izraznega drevesa za logične izraze



## SLOVAR

### SLOVAR BASICS:

- poseben primer množice
- omogoča samo vstavljanje, brisanje, iskanje
- zahteva učinkovito iskanje
- iskanje je razlika med slovarjem in seznamom
- urejenost bodisi na elementih ali pa na ključih
- minimalni, maksimalni element
- predhodnik ali pa izpis na intervalu funkcije

• MAKENULL(D) – naredi prazen slovar D

- MEMBER(x, D) – preveri, če je element x v slovarju D
- INSERT(x, D) – vstavi element x v slovar D
- DELETE(x, D) – zbriše element x iz slovarja D

### BST: binarno iskalno drevo

- najbolj simple drevesna implementacija slovarja
- rekurzivna definicija
- prazno drevo je **BTS**
- drevo z oznako x in levim in desnim poddrevesom R in L je **BTS** če sta L in R **BTS** in velja zgornja formula

## IMPLEMENTACIJA

### ZGOŠČENA TABELA

**Prednost:** časovna kompleksnost vseh operacij je pod določenimi pogoji reda O(1)

#### Slabosti:

- fiksna podatkovna struktura,
- fiksna zgoščevalna funkcija (zaradi sovpadanja se lahko izrodi),
- neučinkovite operacije, ki temeljijo na urejenosti elementov po ključih

### DREVESNE STRUKTURE

#### Lastnosti:

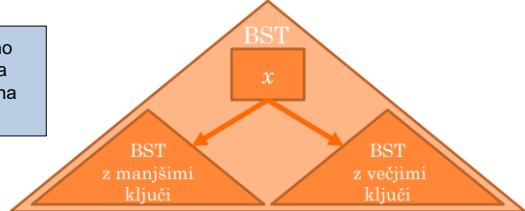
- časovna kompleksnost osnovnih operacij je reda  $O(\log n)$
  - časovna kompleksnost operacij na podlagi urejenosti elementov slovarja je reda  $O(n \log n)$
- pri čemer je  $n$  število elementov slovarja.

```

public class BSTreeNode {
    Comparable key;           // ključ
    BSTreeNode left;          // referenca na levo poddrevo
    BSTreeNode right;         // referenca na desno poddrevo
    ...
}
  
```

#### implementacija vozlišča

drevo podano kot referenca na vozlišče na korenju



### rotacija:

→ zrcalimo drevo po simetriji

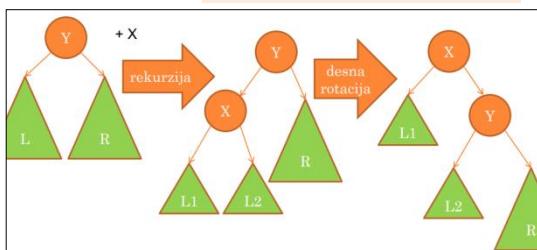
```

protected BSTreeNode rightRotation
    BSTreeNode temp = node;
    node = node.left;
    node.left = temp;
    temp.right = node;
    return node;
}
  
```

```

protected BSTreeNode leftRotation
    BSTreeNode temp = node;
    node = node.right;
    temp.right = node.left;
    node.left = temp;
    return node;
}
  
```

#### dodajanje manjšega elementa



### DODAJANJE ELEMENTA V KOREN:

- če je nov element manjši od korena potem rekurzivno dodamo element v koren levega poddrevesa, imamo desno rotacijo
- če je večji od korena ravno obratno v koren desnega, imamo levo rotacijo

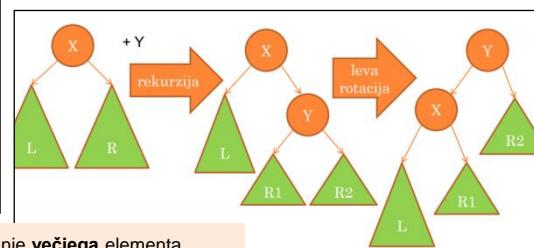
### iskanje v BTS:

- največ možnih korakov je višina drevesa
- pri poravnanim drevesu O(log n)
- pri izrojenem drevesu O(n)

```

private boolean member(Comparable x, BSTreeNode node) {
    if (node == null)
        return false;
    else if (x.compareTo(node.key) == 0)
        return true;
    else if (x.compareTo(node.key) < 0)
        return member(x, node.left);
    else
        return member(x, node.right);
}
  
```

#### dodajanje večjega elementa



```

private BSTreeNode insertRoot(Comparable x, BSTreeNode node) {
    if (node == null)
        node = new BSTreeNode(x);
    else if (x.compareTo(node.key) < 0) {
        node.left = insertRoot(x, node.left);
        node = rightRotation(node);
    } else if (x.compareTo(node.key) > 0) {
        node.right = insertRoot(x, node.right);
        node = leftRotation(node);
    } else
        ; // duplicate; do nothing or throw exception
    return node;
} // insertRoot

```

implementacija vstavljanja v koren pri BST

### brisanje v BTS:

- ne smemo zbrisati cele veje
- **posodobimo**
- poiščemo element
- ga nadomestimo z **minimalnim** elementom v **desnem** poddrevesom
- lahko z **maksimalnem** v **levem**
- glede časovne zahtevnosti je isto kakor za vstavljanje

// za prenos minimalnega ključa iz desnega poddrevesa

```

private Comparable minNodeKey() {
    public BSTreeNode delete(Comparable x, BSTreeNode node) {
        if (node == null)
            if (x.compareTo(node.key) == 0) { // delete node
                if (node.left == null) // empty left
                    node = node.right;
                else if (node.right == null) // empty right
                    node = node.left;
                else {
                    node.right = deleteMin(node.right); // delete min from right
                    node.key = minNodeKey(); // put it into the node
                }
            }
        else if (x.compareTo(node.key) < 0)
            node.left = delete(x, node.left);
        else
            node.right = delete(x, node.right);
        } // if (node != null)
        return node;
    } // delete
}

```

## RBT: rdeče črno iskalno drevo

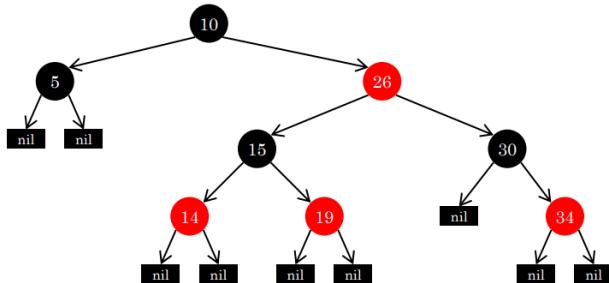
### lomljena drevesa:

- pri vsaki operaciji se drevo lomi
- **dvojne rotacije** za dvig elementa v koren
- pri iskanju se najdeni element dvigne v koren
- pri vstavljanju vstavimo kot list in dvignemo v koren
- pri brisanju dvignemo v koren, dvignemo najmanjši element v **levem** poddrevesu in nadomestimo koren

random informativen fact idk man

### RBT BASICS:

- vozlišče **rdeče** ali **črne** barve
- vsako **rdeče** ima lahko samo **črna sinova**
- vsaka **pot** od **vozlišča** do **praznega** poddrevesa vsebuje enako število **črnih** vozlišč
- če **n** vozlišč višina največ  $2 \log_2(n + 1)$
- drevo je **vedno delno poravnano**



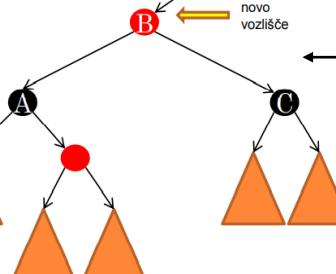
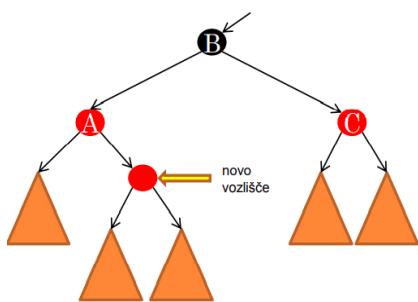
### delna poravnovanost:

- višina drevesa je največ **dvakrat večja** od **poravnovanega** drevesa z istim številom vozlišč

**2. očeta in strica**  
pobarvamo v **črno** starega očeta pa v **rdečo**

### IMPLEMENTACIJA:

- k navadnemu **BST** vozlišču dodamo **očeta** in **barvo**
- vse osnovne operacije reda **O(log n)**
- **dodajanje**: dodamo **rdeči list** in mogoče **popravljamo** v najslabšem primeru do **korena**
- **brisanje**: minimalni iz desnega poddrevesa **nadomesti** element, če je **črn** mogoče potrebno popravljanje again najslabše do **korena**

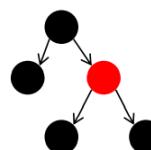
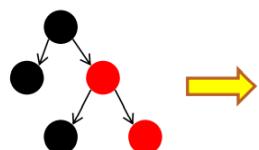
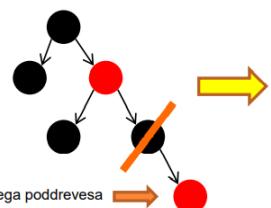


### popravljanje pri brisanju:

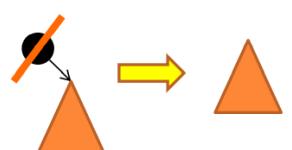
najprej brišemo kot navadno **BST drevo**, torej če je element list drevesa, ga enostavno **zbrisemo**, če ima samo **enega** sin, ga **zbrisemo** ter na njegovo mesto postavimo njegovega sina in če ima **dva** sina, zbrisemo **največji** element iz **levega** poddrevesa ali **najmanjši** element iz **desnega** poddrevesa, ki **nadomesti** zbrisano vozlišče

če smo zbrisali **črno** vozlišče se **zmanjša črna višina** za ena kar je treba popraviti.

1. sin korena  
**rdeč** potem ga  
pobarvamo  
**črno** in basta

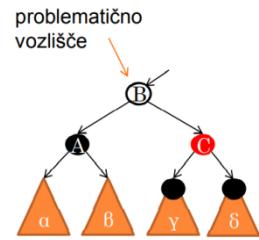
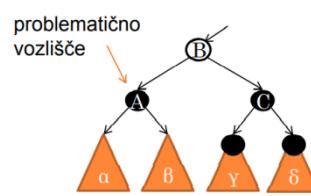
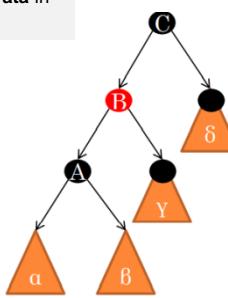
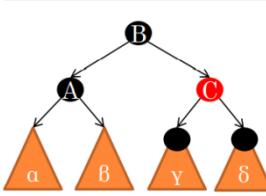


2. če je imelo vozlišče samo **eno**  
**poddrevo** je vse good

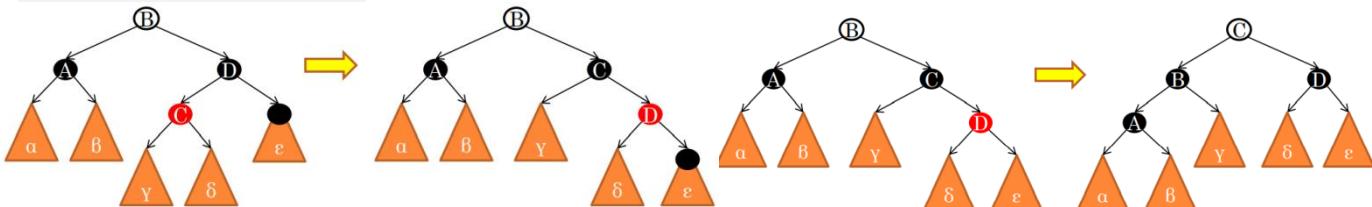
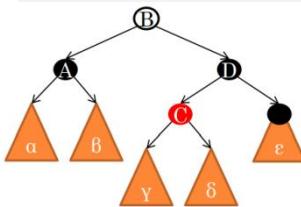


3. če je **koren** problematičnega vozlišča **črn** in njegov **brat rdeč** potem je **oče črn** zato najprej **rotiramo brata in očeta**. dobimo **črnega brata** in nadaljujemo v 4.

4. če je je **brat C črn** in sta nečaka črna, pobarvamo **brata v rdeče** in rekurzivno ponovimo celoten postopek z novim **vozliščem B**



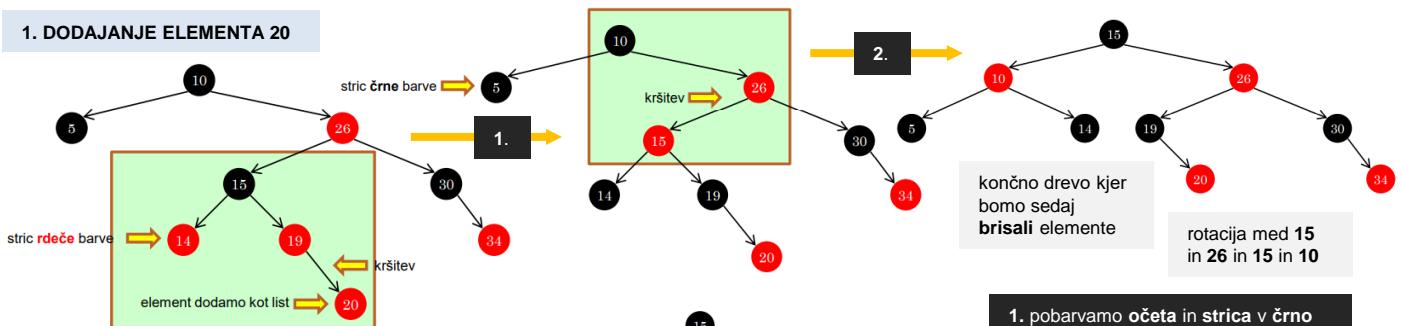
5. Če je **brat D črn** in je zunanji **nečak črn**, rotiramo **brata** in **notranjega nečaka** in zamenjamo **njeni barvi** in nadaljujemo v 6



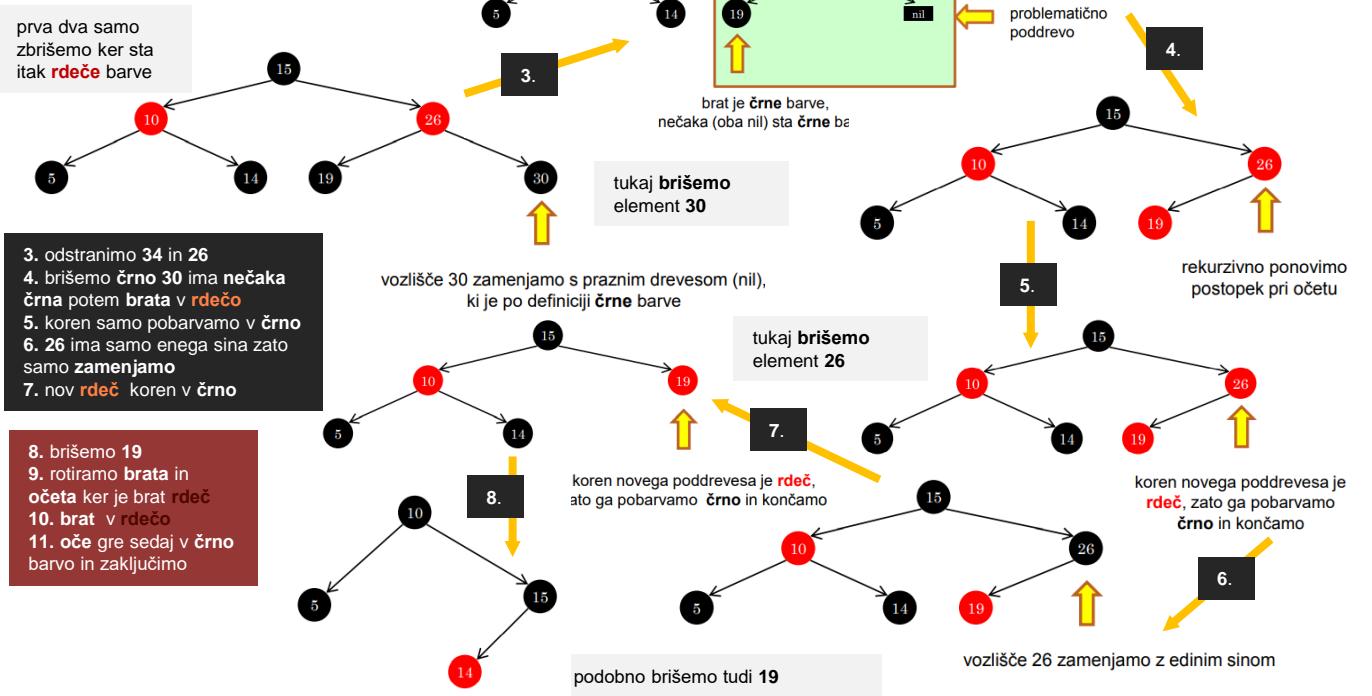
6. Če je **brat C črn** in je **zunanji nečak rdeč**, izvedemo **enojno rotacijo** med **očetom** in **bratom** in končamo, ker se je črna višina problematičnega poddravesa **povečala** za ena.

## PRIMER

### 1. DODAJANJE ELEMENTA 20



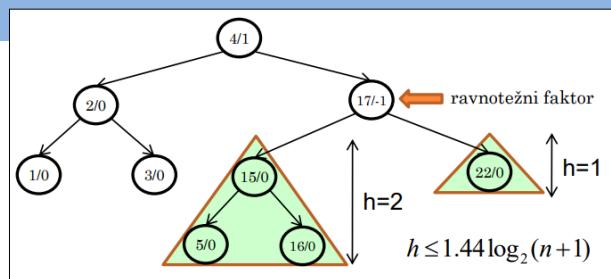
### 2. BRISANJE ELEMENTOV 34, 20, 30, 26, 19



## AVL: Adelson Velskii in Landis drevo

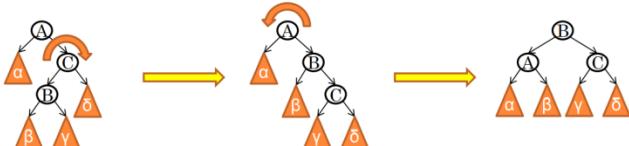
### OBHOD DREVESA:

- **delno poravnano** binarno iskalno drevo
- višini obeh **poddreves** razlike **največ 1** za vsako vozlišče
- zahtevnost operacij je  $O(\log n)$
- **ravnotežni faktor** vozlišča je **razlika višin** poddreves
- višina praznega drevesa je **0**
- ravnotežni faktor dodamo **BST** vozlišču

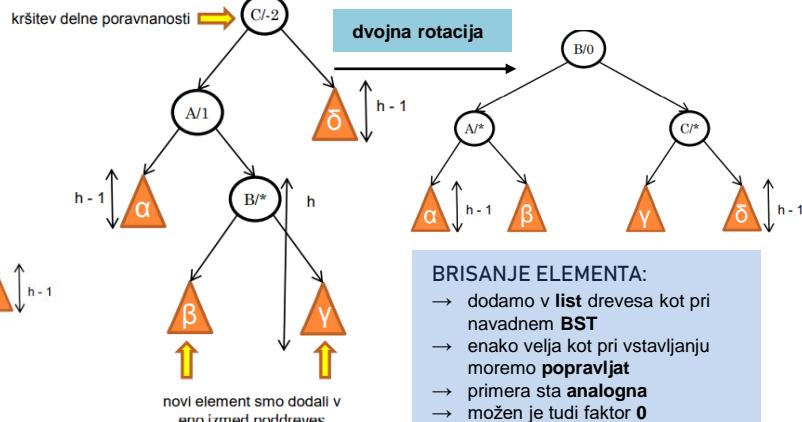
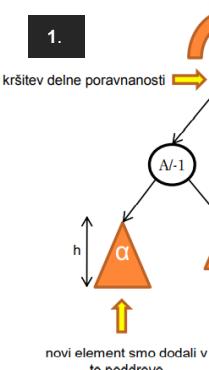
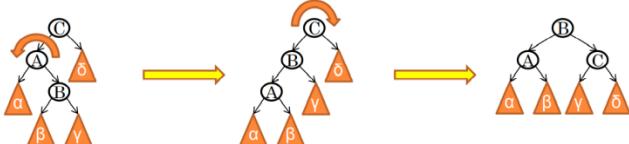


### OPERACIJE:

- **iskanje**: kot pri običajnem
- **dodajanje**: dodamo list in preračunamo **ravnotežnostne faktore** navzgor, enkrat potrebna **rotacija**
- **brisanje**: nadomestimo element z minimalnim iz desnega poddrevesa in preračunamo **ravnotežnostne faktore** navzgor in nato potrebne rotacije



### DVOJNE ROTACIJE



**BRISANJE ELEMENTA:**

- dodamo v list drevesa kot pri navadnem BST
- enako velja kot pri vstavljanju moremo **popravljati**
- primera sta **analogni**
- možen je tudi faktor **0**

### POPOLNOMA PORAVNANA DREVESA : vsi listi so na istem nivoju

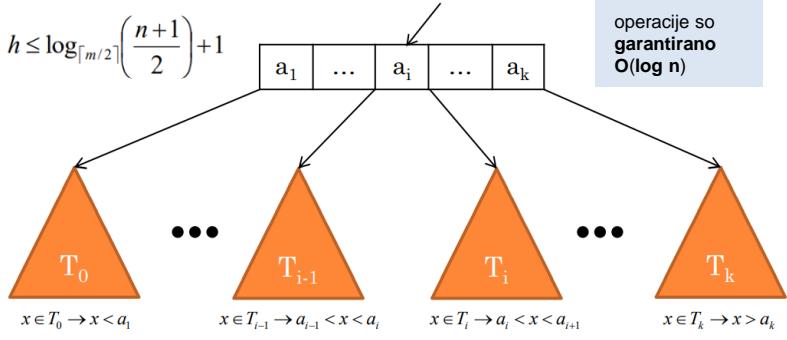
#### 2 3 DREVESA:

- vozlišče 2 ali 3 sinove
- vozlišče 1 ali 2 ključa
- **1. možnost**: elementi se nahajajo v listih in je ključ v **notranjem** vozlišču enak **najmanjšemu** ključu v **desnem** poddrévesu
- **2. možnost**: listi so prazni in so elementi v notranjih vozliščih, **pospološitev BST**
- zahtevnost operacij je  $O(\log n)$

again fucking random

- vsako notranje vozlišče **reda m** ima od **navzgor zaokrožene polovice m** do **m sinov** in en ključ manj kot ima sinov
- izjema: **koren drevesa** lahko od **2** do **m sinov**
- elementi so shranjeni v **notranjih vozliščih**
- **listi so prazna** poddrévesa
- pospološitev binarnih iskalnih dreves

#### B drevesa



- Kje se uporablja B drevo?
- velikost glavnega pomnilnika ni dovolj za **velike baze podatkov**
  - uporabimo **zunanji disk**
  - **počasne** povezave
  - **minimiziramo število dostopov** do zunanjega pomnilnika
  - uporabljamo **B drevo**

### 1. ISKANJE

- začnemo v **korenju**
- **primerjamo** element
- če ga najdemo **zaključimo**
- če naletimo na element ki je **večji** potem nadaljujemo iskanje v **poddrevesu** z istim indeksom
- časovna zahtevnost iskanja je  **$O(m \log n)$**
- če je ključev ogromno potem **bisekcija**
- **$O(\log m \cdot \log n)$**

### OPERACIJE IN IMPLEMENTACIJA

```
public class BTreenode {
    int count; // 0..m-1 stevilo kljucev v vozlišcu
    Comparable keys[]; // array [0..m-1]
    BTreenode children[]; // array [0..m]: m+1 možnih otrok
```

### 3. BRISANJE:

- element **nadomestimo** z
- **maksimalnim iz desnega ali minimalnim iz levega**
- vedno izbrisani v listu
- če je **premal** elementov v listu, ga **zdržimo** z bratom in ustreznim ključem, ki ga rekurzivno **zbrisemo** pri **očetu**

```
Min = indeks prvega (najmanjšega) ključa - 1
Max = indeks zadnjega (največjega) ključa + 1;
ponavljaj
    če Max-Min = 1 potem poddrevo najdeno;
    sicer
        i = (Max-Min)/2;
        če Element < ključ(i) potem Max = i;
        sicer
            če Element > ključ(i) potem Min = i;
            sicer element najden; // ključ(i) == Element
            dokler ne najdeš elementa ali ustreznega poddreveta;
```

### 3. BRISANJE

- brišemo element iz vozlišča na **zadnjem nivoju**
- če element **ni** na zadnjem nivoju ga **nadomestimo** z **največjim** elementom ustreznega **levega poddreveta** in tega dejansko **izbrisemo**
- če število elementov ne pada pod dovoljeno zaključim
- če **pade pod m / 2 navzgor – 1 popravljamo**
- postopek rekurzivno ponavljamo pri očetu

če eden izmed **bratov** vsebuje **dovolj** elementov od brata **vzame enega ali več** elementov skupaj z ustreznimi **poddrevesi** in z **zamenjavo** ustreznega elementa pri **očetu**

### 1. ISKANJE:

- **pospolitev BST**
- element bodisi **najdemo** ali pa **ustrezeno poddrevo**

### 2. DODAJANJE

- če pravo vozlišče vsebuje **manj** kot **m - 1** elementov **dodamo** in **končamo**
- če ni prostora **razbijemo** na **2** vozlišči
- sredinski je navzgor **zaokrožen m / 2**
- do sredinskega na **levo**
- od sredinskega **desno**
- sredinskega rekurzivno **očetu prejšnjega**
- **$O(\log n)$**

če eden **nobeden** izmed bratov dovolj elementov jih **zdržimo**

## PRIORITETNA VRSTA

### Kaj je priority queue?

- vsak element **oznako prioritete**, ki določa **vrstni red** brisanja elementov iz vrste
- **ne velja FIFO!**
- **nizja** prioriteta pomeni da gre element **prej** iz vrste

	MAKENULL	EMPTY	INSERT	DELETEMIN
neurejeni seznam	$O(1)$	$O(1)$	$O(1)$	$O(n)$
urejeni seznam	$O(1)$	$O(1)$	$\leq O(n)$	$O(1)$
BST	$O(1)$	$O(1)$	$\leq O(n)$	$\leq O(\log n)$
AVL, RB-drevo	$O(1)$	$O(1)$	$= O(\log n)$	$= O(\log n)$
kopica	$O(1)$	$O(1)$	$\leq O(\log n)$	$\leq O(\log n)$

učinkovitost operacij z različnimi implementacijami

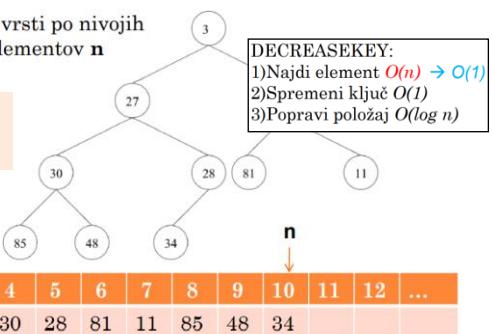
```
public interface PriorityQueue {
    public void makenull();
    public void insert(Comparable x);
    public Comparable deleteMin();
    public boolean empty();
} // PriorityQueue
```

### KOPICA:

- **binarno drevo**
- **levo poravnano**: elementi na najnižjem nivoju lahko manjkajo samo na desni strani
- **delno urejeno**: v korenu **najmanjši** element tega poddreveta

vozlišča hranimo po vrsti po nivojih hranimo še število elementov **n**

### implementacija kopice z poljem:



v vozliščih ne potrebujemo dodatnih indeksov, saj jih lahko sproti izračunamo:

če z **i** označimo indeks vozlišča, potem velja:

- $2 * i$  je indeks levega sina
- $2 * i + 1$  je indeks desnega sina
- $i / 2$  je indeks očeta

**DECREASEKEY**: vsak element hrani kazalec na svoj **polozaj** v kopici ker potrebujemo direkten **dostop** !

### IZGRADNJA:

- kopico z **n elementi** zgradimo v času reda  **$O(n \log n)$**  če **n** krat uporabimo **insert**
- **$O(n)$** , če so vsi elementi podani na začetku in jih poljubno postavimo v kopico, da je **levo poravnana** in potem **urejamo**

Za algoritme na grafih potrebujemo še operacijo:

- **DECREASEKEY(x,k,Q)** : elementu **x** v kopici zmanjša ključ na **k**

# DISJUNKTNE MNOŽICE

- množice **disjunktne** glede na neko relacijo
- gradimo tako da vsak element **podmnožica**
- **manjše** združujemo v **večje** če so elementi iz ene in druge v **relaciji**
- za vsak element **vemo kateri** podmnožici **pripada**

```
public interface DisjointSet {
    public abstract void makenull();
    public abstract DisjointSubset makeset(Object x);
    public abstract void union(DisjointSubset a1, DisjointSubset a2);
    public abstract DisjointSubset find(DisjointSubset x);
} // interface DisjointSet
```

terminologija in logika  
behind it basically from DS2

## GRAFI

### TERMINOLOGIJA:

- množica **vozlišč** in **povezav**  
talking about  
usmerjeni grafi
- **povezava** je urejen **par** vozlišč
- **izstopna stopnja**: število povezav ki se **začnejo** tu
- **vstopna stopnja**: število povezav ki se **končajo** tu
- **poln graf**: vsako vozlišče z vsemi in sabo
- **pot**: zaporedje vozlišč od A do B
- **enostavna pot**: ni ciklov
- **drevo**: graf brez ciklov
- **vpeto drevo**: podgraf grafa ki je drevo lol
- **n** je število **vozlišč**
- **m** je število **povezav**

**IMPLEMENTACIJA Z SEZNAMOM SOSEDNOSTI:**

- **vozlišča** hranimo v **seznamu**
- vsako vozlišče ima **seznam povezav** iz njega
- vsaka **povezava** kaže na **konec** povezave
- **časovna zahtevnost** vseh operacij reda **O(1)**

### NEUSMERJENI GRAFI:

- **povezava** ni usmerjen ampak **neusmerjen** par
- stopnja samo **število povezav**
- implementiramo isto samo da je seznam  
**dvosmeren** imamo dvoje kazalcev

**MAKENULL(S)** generira prazno množico množic  $S$ .

**MAKESET(x, S)** tvori novo množico  $\{x\}$  in jo doda v  $S$ .  $O(1)$

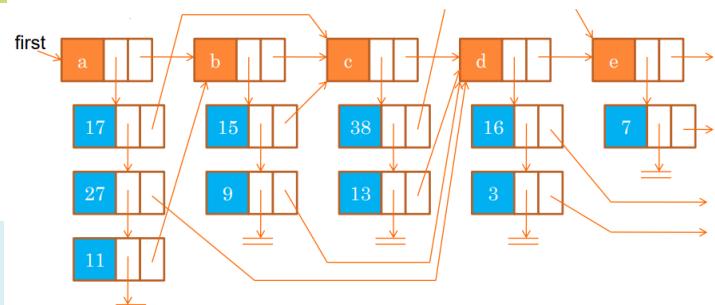
**UNION(A<sub>1</sub>, A<sub>2</sub>, S)** združi dve disjunktni podmnožici  $A_1$  in  $A_2$  v novo podmnožico  $O(1)$

**FIND(x, S)** vrne podmnožico, katere element je  $x$ .

### IMPLEMENTACIJA Z GOZDOM:

- vsaka **množica** je **drevo**
- vsak element kaže na **očeta** v **drevesu**
- **koren** kaže **sam nase**
- množica **identificirana** z korenom
- ko iščemo hkrati privežemo **vsak element** na **poti** na **koren**  
zato da ne pride do **izrojenosti**
- **unija** priveže koren **manjše** na koren **večje** množice

- **MAKENULL(G)** – naredi prazen usmerjen graf  $G$
- **INSERT\_VERTEX(v, G)** – doda vozlišče  $v$  v graf  $G$
- **INSERT\_EDGE(v<sub>1</sub>, v<sub>2</sub>, G)** – doda povezavo  $<v_1, v_2>$  v grafu  $G$
- **FIRST\_VERTEX(G)** – vrne prvo vozlišče v grafu  $G$
- **NEXT\_VERTEX(v, G)** – vrne naslednje vozlišče danega vozlišča  $v$  po nekem vrstnem redu
- **FIRST\_EDGE(v, G)** – vrne prvo povezavo v grafu  $G$  z začetkom  $v$
- **NEXT\_EDGE(e, v, G)** – vrne naslednjo povezavo dane povezave  $e$  z začetkom  $v$  po nekem vrstnem redu
- **END\_POINT(e, G)** – vrne konec povezave  $e$  v grafu  $G$



- **ADJACENT\_POINT(e, v, G)** – vrne drugi konec povezave  $e$  v grafu  $G$  z enim koncem v  $v$

## kritična pot

### KAKO NAJDEM KRITIČNO POT:

- najdaljša pot od **začetka** do **konca**
- painfully **časovno zahtevno** z navadno rekurzijo
- povezave imajo neko **ceno** oz. **čas**
- graf pregledujemo od **začetka** proti **koncu**
- hranimo **pregledana** vozlišča, katerih **naslednikov** še **nismo** pregledali
- za vsako vozlišče hranimo **maksimalno** pot
- hranimo **vstopno stopnjo** vozlišča ki se **manjša** za 1 vsakič ko vozlišče **pregledamo**
- če želimo izpis shranimo še **predhodnika**
- vse začetne čase damo na 0 ko začnemo
- priprava grafa **zahtevnost O(n + m)**
- časovna **zahtevnost** za iskanje poti **O(m)**

priprava  
graфа

- izračunamo vstopne stopnje vseh vozlišč
- postavimo začetne čase za vsa vozlišča na 0
- prehod preko vseh vozlišč (n) in vseh povezav (m)

## najkrajša pot

### KAKO NAJDEM NAJKRAJŠO POT:

- spremenimo algoritem z **dinamičnim programiranjem** za kritično
- zahteva **graf brez ciklov** in eno zaključno vozlišče
- lahko dobimo **vpeto drevo** najkrajših poti z **algoritmom Dijkstra**
- deluje tudi če **imamo cikle** pazit moramo da ni **negativnih cen**

### DIJKSTRA:

- iz **začetnega** vozlišča **proti listom**
- **požrešni** algoritem: iz množice vozlišč ki še **niso** v drevesu izberemo tisto ki je **najkrajša** od začetka
- ko vozlišče **dodamo** pregledamo **naslednike**

potrebujemo operacijo  
za **zmanjšanje**  
prioritet

če naslednik v  
drevesu ga  
ignoriramo

če v prioritetni vrsti  
zmanjšamo  
prioritet

če ni v prioritetni vrsti  
ga dodamo

## DECREASE KEY (x, New, Q)

- zmanjša prioriteto elementa  $x$  na New
- v kopici operacijo implementiramo tako, da element z zmanjšano prioriteto zamenjujemo z očetom
- postopek se ustavi, bodisi če je oče manjši od elementa ali če element pride v koren kopice
- časovna zahtevnost je reda  $O(\log n)$  pod pogojem, da imamo direkten dostop do elementa v kopici
- vsako vozlišče hrani svoj položaj (indeks) v kopici

### KURSKALOV ALGORITEM:

- na začetku vsako vozlišče svoje drevo
- združujemo drevesi v eno tako da ju povežemo z najkrajšo povezavo
- na začetku vse povezave damo v prioritno vrsto
- drevo je disjunktna množica vozlišč



### primerjava:

- Primov algoritem deluje le za povezane grafe Kruskalov pa tudi za nepovezane
- v Primovem so v prioritetti vrsti vozlišča v Kruskalovem pa povezave

**požrešen algoritem:** algoritem pri katerem naredimo lokalno optimalno potezo na vsakem koraku in ne zagotavlja nujno optimalnosti as a whole

- vsako vozlišče dodamo in izbrisemo iz prioritete vrste torej dvakrat
- **n** operacij INSERT in DELETEN
- časovna kompleksnost če vrsta s kopico najslabše  $O((n + m) \log n)$
- ker graf povezan velja  $m \geq n - 1$
- red  $O(m \log n)$

## minimalno vpeto drevo

### PRIMOV ALGORITEM:

- požrešni algoritem: isto ko Dijkstra samo da je graf neusmerjen
- gradimo od poljubnega začetnega vozlišča
- izmed množice vozlišč ki še niso v drevesu izberemo tisto z najkrajšo povezavo do nekega vozlišča
- spet pogledamo sosedje in naredimo isto kot pri Dijkstri

### ADT GRAPH spremenim → ADT KGRAPH:

MAKENULL(G) naredi prazen graf  $G$ .

INSERT.VERTEX(v, G) doda vozlišče  $v$  v graf  $G$ .

FIRST.VERTEX(G) vrne prvo vozlišče v grafu  $G$ .

NEXT.VERTEX(v, G) vrne naslednje vozlišče danega vozlišča  $v$  po nekem vrstnem redu v grafu  $G$ .

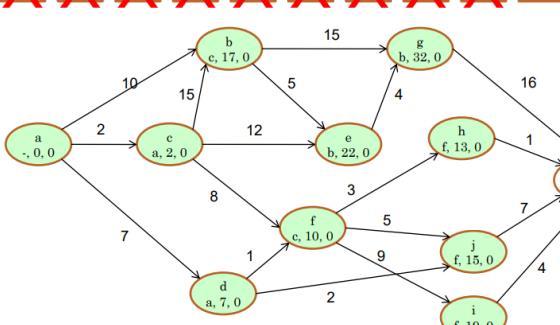
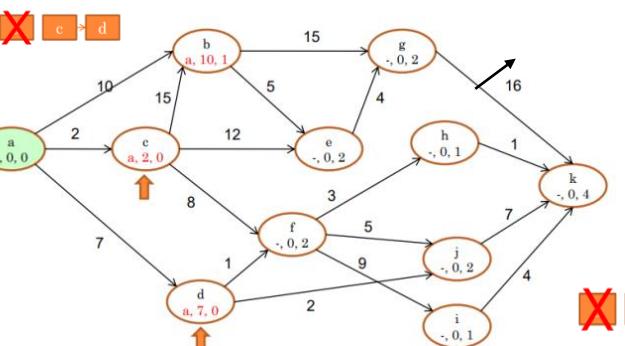
\*FIRST.EDGE(G) vrne prvo povezavo v grafu  $G$ .

\*NEXT.EDGE(e, G) vrne naslednjo povezavo dane povezave  $e$  v grafu  $G$  po nekem vrstnem redu.

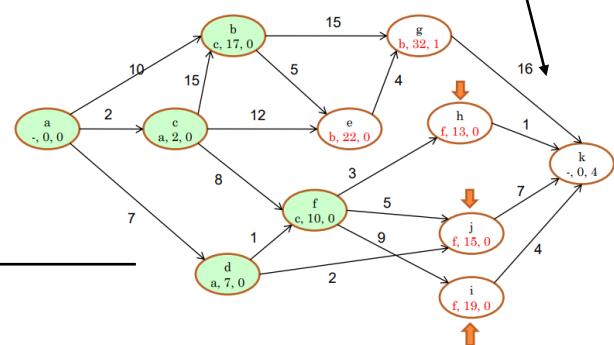
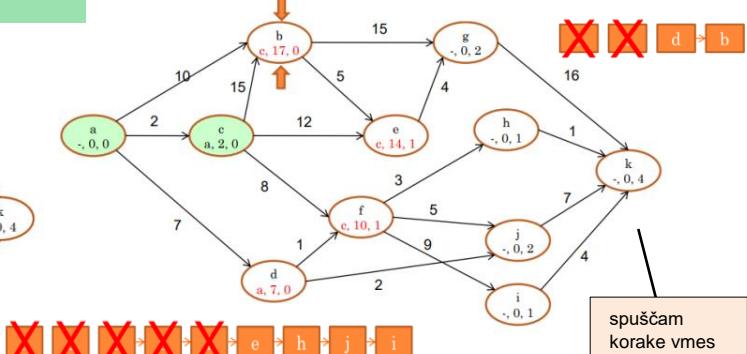
\*ENDPOINTS(e, G, v1, v2) vrne oba konca,  $v1$  in  $v2$ , povezave  $e = < v1, v2 >$  v grafu  $G$ .

## PRIMERI ISKANJA POTI:

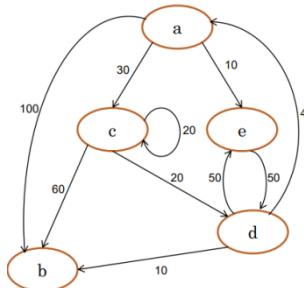
### 1. kritična pot



našli smo daljšo pot: razdalja vozlišča 'c' + povezava od 'c' do 'b'



## 2. najkrajša pot

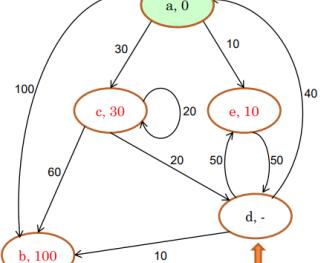


Prioritetna vrsta (kopica):



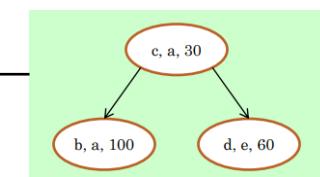
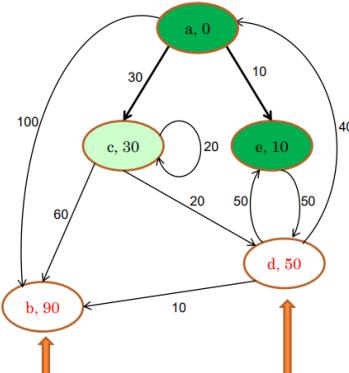
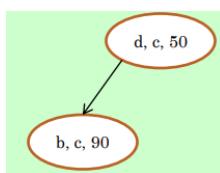
zapis vsebuje:  
 • ime vozlišča  
 • ime očeta v vpetem drevesu  
 • dolžina najkrajše znane poti od začetnega vozlišča

odstranjeni element iz prioritetne vrste vsebuje rezultat



še vedno neobiskano vozlišče

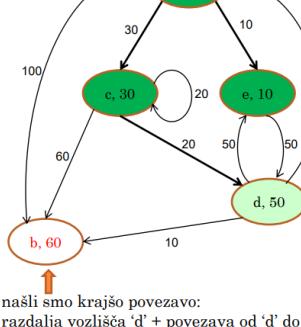
razdalja vozlišča 'a' + povezava od 'a' do 'b'



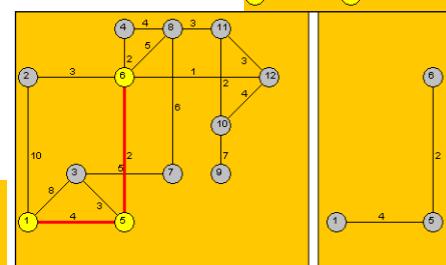
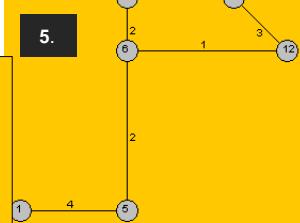
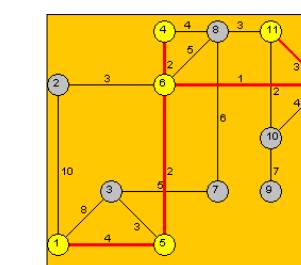
3. minimalno vpeto drevo

c ne dodamo v prioriteten vrsto, ker je razdalja 30 + 20 daljša od najkrajše znane razdalje 30

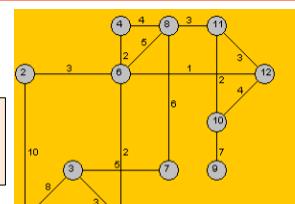
končamo ko je prioriteta vrsta prazna



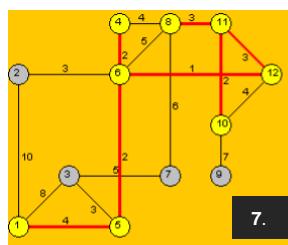
a ne dodamo v pr. vrsto, ker je razdalja 50 + 40 daljša od najkrajše znane razdalje 0 tako rešimo cikle



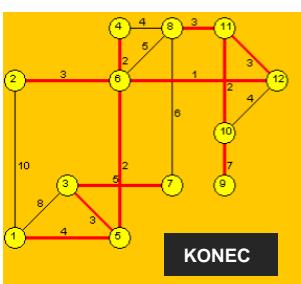
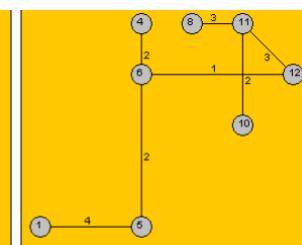
1.



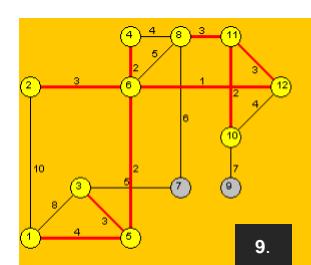
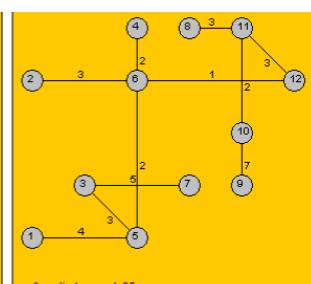
2.



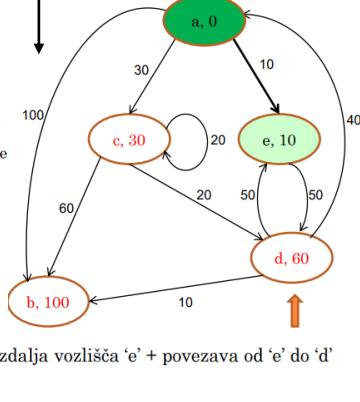
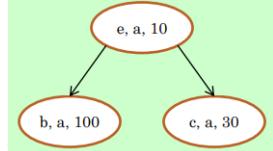
7.



KONEC



9.



# REŠEVANJE PROBLEMOV

## 1. Predpogoji ?

- sposobnost **pomnjenja**
- sposobnost **procesiranja**
- sposobnost **učenja**
- **naravna in umetna inteligencija**

## kratkoročni spomin:

- hiter vendar potrebovno **obnavljati**
- omejen na 5, 7 ali 9 kazalcev

## dolgoročni spomin:

- počasen
- neomejen
- povezave med nevroni, jakost povezav, proteini
- zapomnimo si vse, samo **dostopa** nimamo ?

## zunanji spomin:

- zunanji disk...
- zahteva **notacijo**
- izbira notacije ključna za uspeh

## zaporedno procesiranje:

- **počasno**
- zavestno oz. **logično**
- lahko **opisemo**

## vzporedno procesiranje:

- zelo **hitro**
- podzavestno oz. **numerično**
- brez opisa

**znanje:** interpretacija informacije iz podatkov

**učenje:** sprememba v sistemu, ki mu omogoča, da naslednjič **enako** ali podobno nalogu reši **bolje**

- **implicitno:** podzavestno, iz izkušenj, neopisljivo
- **eksplicitno:** logično, zavestno, opisljivo
- eksplicitno zahteva **notacijo**

**zavest:** rezultat **kompleksnosti** sistema in je nastala v nekem časovnem trenutku evolucije

- **nemerljiva**
- ne znamo jo definirat
- **neopisljiva**
- doživljamo **subjektivno**

imamo naravno in strojno **učenje** in naravno in umetno **inteligenco**

- preko **100** vrst
- spominska, numerična, besedna, logična, prostorska, čustvena...
- ni brez sposobnosti **pomnjenja, procesiranja in učenja**
- količina je **merljiva**

## osnovni principi reševanja problemov

### UMETNA INTELIGENCA:

- **teorija izračunljivosti:** malo problemov rešljivih algoritmično
- matematična **logika**, računalniški **programski jeziki**, **rekurzivne** funkcije in **formalne gramatike** so po izrazni moči ekvivalentne

### breadth first search:

- garantirano **najkrajša** pot do rešitve
- z globino **ekspONENTNO** narašča prostorska zahtevnost

### depth first search:

- prostorska zahtevnost **linearno**
- lahko zgreši rešitev blizu **začetnemu** stanju
- lahko se **zaciKla**

### iterative deepening:

- najde **najkrajšo** pot
- ni težav z ciklanjem
- prostorska zahtevnost **linearna**
- preiščemo cel prostor ki smo ga že preiskali torej **casovna zahtevnost eksponentna**

### branch and bound z best first:

- zelo dobra strategija
- **eksponentna** prostorska zahtevnost

### 1. RAZUMEVANJE:

- razjasnitev zahtev
- **implicitne** podatke izrazimo **eksplicitno**
- identificiramo **nepOMEMBNE** podatke
- nedvoumna definicija problema
- opis **dejstev**
- opis **pravil**
- opis **ciljev**

### 2. ABSTRAKCIJA:

- **identificiramo** in **poimenujemo** pomembne objekte
- določimo **relacije** med objekti in **operacije**
- s tem definiramo **problemski prostor**

### 4. PODPROBLEMI:

- od **zgoraj dol**
- od **spodaj gor**
- od spodaj gor je na primer Fibonacci
- lahko poenostavimo in najdemo **izomorfni** primer

### 3. IZBIRA NOTACIJE:

- izberemo **zapis**

#### formalni sistemi:

- **algebra**
- **geometrija**
- teorija **množic**
- **izjavni** račun
- **predikativni** račun 1. reda
- **programski jezik**

### 5. PREISKOVALNA STRATEGIJA:

- problemski prostori so **zelo veliki**
- izberemo glede na **velikost** in **lastnosti** prostora
- iščemo lahko **optimalno** ali **približno** rešitev
- optimalno: **izčrpno** in omejeno izčrpno
- približno: **požrešno**...

## OPTIMALNE STRATEGIJE

### greedy search:

- izredno hiter
- veliko **izpustimo**
- v **praksi** pogosto dovolj

### iskanje v snopu:

- **pospolištev** požrešnega iskanja
- večkrat ponoviš iskanje **iz naključnega** začetnega stanja
- določimo **koliko** vej pogledamo

### lokalna optimizacija:

- **pospolištev** požrešnega iskanja
- večkrat ponoviš iskanje **iz naključnega** začetnega stanja
- super za **velike** prostore

### gradientno iskanje:

- **pospolištev** požrešnega iskanja
- za **vezne** prostore

## SUBOPTIMALNE STRATEGIJE

## reševanje problemov z algoritmi

„Algoritem je **končno zaporedje** natančno določenih **ukazov**, ki opravijo neko nalogu v **končnem številu korakov**. Algoritem sprejme **vhodne podatke** in vrne **rezultat**. „

### 2. ZASNOVA:

- naredimo **neformalni opis** oziroma **matematični model**
- **abstraktни podatkovni tipi** in **psevdokoda**
- **podatkovne strukture** in **program**

### 4. PREVERJANJE:

- **branje**
- **testiranje**: dokaže nepravilnosti ne pravilnosti
- **formalno dokazovanje pravilnosti!**

- izomorfni problem je **dobro barvanje** grafa
- uporabljamo **požrešno barvanje** iz DS
- približna rešitev

### PRIMER: križišče

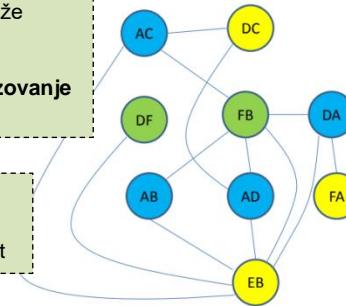
- **objekti**: ovinki
- **relacija**: taki ki jih lahko izvedemo naenkrat in taki ki jih ne moremo
- **dobo čakanja** želimo zmanjšati na minimum
- cikel **preklapljanja semaforjev** na minimum čim več kompatibilnih ovinkov naenkrat zeleno

### 1. IZRAŽANJE:

- diagram poteka
- **psevdokoda**
- programski jezik

### 3. ANALIZA:

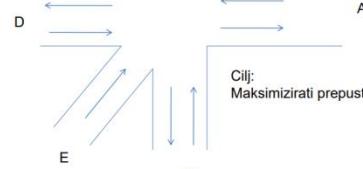
- časovna zahtevnost
- prostorska zahtevnost



POMEMBNI objekti:  
Ovinki-npr: AB, AC, CD, EF itd.

Relacija:  
Kompatibilni npr. AB, AC in DF  
Nekompatibilna npr. AC in FB

C  
B



Cilj:  
Maksimizirati prepustnost

## FORMALNO DOKAZOVANJE PRAVILNOSTI

- pomembno za algoritme ki jih ni mogoče **intuitivno** preveriti dokažemo pravilnost
- če je **nepravilno** delovanje res tragično i guess moramo preveriti formalno

iz pogojev **P** ki veljajo **pred izvršitvijo** stavka izpeljujemo pogoje **Q** ki veljajo **po izvršitvi**

### Zanka:

//  $P_1(y)$   
while (Pogoj(y)) {

// za  $i \rightarrow$  izvajanje zanke:  $P_i(y) \& Pogoj(y)$   
S;

} // while  
//  $P_k(y) \& !Pogoj(y)$

Pri čemer velja

//  $P_i(y) \& Pogoj(y)$   
S  
//  $P_{-(i+1)}(y)$

definiramo **zančno invarianto** torej pogoj I ki je resničen prej vmes in na koncu

//  $I(y)$   
while (Pogoj(y)) {  
//  $I(y) \& Pogoj(y)$   
S;  
//  $I(y)$   
} // while  
//  $I(y) \& !Pogoj(y)$

in je pogoj  $P_k(y)$  odvisen od števila izvajanj zanke.

### 1. PRAVILA IZPELJEVANJA POGOJEV:

- program definiramo kot **preslikavo**
- elementi množice x morajo izpolnjevati **vhodni** pogoj tisti v z pa **izhodni**
- program je **parcialno pravilen**: program se ustavi in izhodni podatki **izpolnjujejo** pogoj
- **totalno pravilen**: je parcialno pravilen in se ustavi po **končnem številu korakov**

$$f: X \longrightarrow Z$$

$$\phi(x_1, \dots, x_n)$$

vhodni in izhodni pogoj

$$\Psi(z_1, \dots, z_m, x_1, \dots, x_n)$$

### Izbira:

//  $P(y)$   
if (Pogoj(y))  
//  $P(y) \& Pogoj(y)$   
...;  
else  
//  $P(y) \& !Pogoj(y)$   
...;

problem ker izstopni pogoj odvisen od števila izvajanj zanke ki ga ne vemo v naprej, ne sme biti trivialen omogočati mora izpeljavo končnega pogoja

dokazati moramo tudi da se zanka resnično ustavi zato definiramo zančno spremenljivko I, dobro urejeno množico, ki je po navadi množica **naravnih števil** in zančno **invarianto** ki pripada množici.

dokazati moramo da zančna **invariante** vedno resnična in da se I zmanjša pri vsaki ponovitvi zanke.

vrednost spremenljivke med izvajanjem označimo z II

torej izpeljujemo pogoje iz pogojev, iz začetnega hočemo po pravilih izpeljati končnega

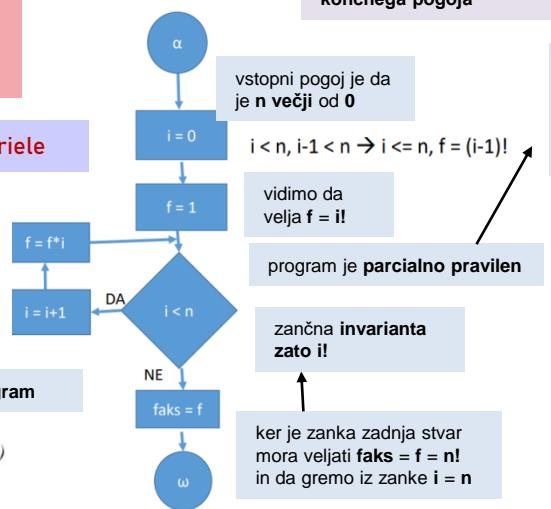
```
// I pripada D
while (Pogoj(y)) {
    // I pripada D & Pogoj(y)
    S;
    // I pripada D & (II < I)
} // while
```

problem ker izstopni pogoj odvisen od števila izvajanj zanke ki ga ne vemo v naprej, ne sme biti trivialen omogočati mora izpeljavo končnega pogoja

### PRIMER: izračun faktorielle

```
static public int faks(int n) {
// f(n) = (n >= 0)
    int i=0,f=1;

    while (i < n) {
        i++;
        f *= i;
    } // while
    return f;
// psi(faks, n) = (faks = n!)
} // faks
```



dokazat moramo **ustavljenost** programa:

- Dobro utemeljena množica: N (z 0)
- Zančna spremenljivka: I = n - i
- Zančna invarianta: n - i ∈ N

dokažemo da se zančna spremenljivka zmanjšuje

### Zaporedje:

```
// P0(y)
{ S1; S2; ...; Sk }
// Pk(y)
pri čemer velja
// P_{-(i-1)}(y)
Si;
// Pi(y)
```

### Združitev več poti izvajanja:

```
if (...) { ... }
// PI(y)
else { ... }
// P2(y)
; // PI(y) or P2(y)
```

### Prireditev:

```
// P(izraz)
y = izraz;
// P(y)
```