PONAVLJANJE GRADIVA PROGRAMIRANJA

⇒ mehanizam alokacije memorije:
- ↳ rezervacija memorije
- ↳ oslobadanje memorije

```
int *y;
y = malloc(1000);
```
→ rezervira mjesto za 1000 bajtova na mjestu u memoriji gdje je najbolje

↳ PIŠI OVAKO:

```
y = malloc(1000 * sizeof(int))
```
↳ rezervira mjesto za 1000 cijelih brojeva

⇒ kada želimo osloboditi svu memoriju koju smo rezervirali, pišemo:

```
free(y);
```

⇒ ako želimo proširit rezervirani blok, pišemo funkciju:

```
y = realloc(y, 1000);
```
↳ oslobadta još 1000 bajtova u nastavku bloka „y" na istom mjestu (ako nema mjesta, „y" će promjenit mjesto)

⇒ sve ove funkcije, malare se u knjižnici:

```
#include <malloc.h>
```

↳ NAPOMENA: tijekom alociranja memorije, u kod
moramo pratit koliko smo memorije
rezervirali jer tu veličinu ne možemo
dobit natrag
↳ ako pišemo „sizeof(Y)", dobit
ćemo veličinu pokazivača „Y", a n
veličinu rezerviranog bloka označenog s „Y

⇒ primjena alokacije javlja se kod želje da rezerviram
velik dio memorije - ako napišemo:

```
int X[1000 000];
```

↳ program se ruši
zbog stack overflow-a

⇒ FORMATIRANA DATOTEKA ⇒ tekstualna datoteka
⇒ NEFORMATIRANA DATOTEKA ⇒ binarna datoteka

POLJA POKAZIVAČA

⇒ recimo da imamo datoteku (formatiranu) i želimo svaki redak pohraniti na drugo mjesto u memoriji

↳ maksimalan broj redaka je 10 !

```
char* p;
char* pc[10];
    // prebrojimo broj redaka i broj znakova u svakom retku
p = (char*) malloc((maxznakured+1)*sizof(char));
    // prekopiramo jedan red na "heap" memoriju na mjesto gdje pokazuje "p"
pc[1] = p;
```

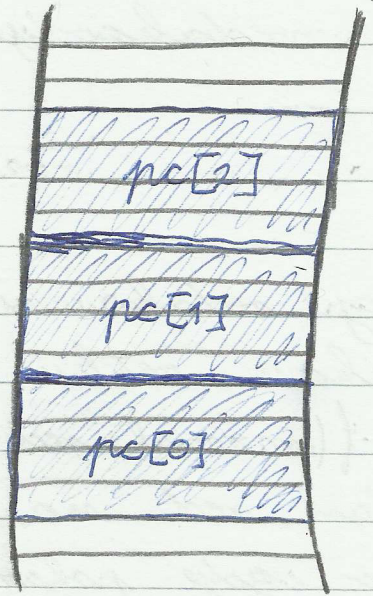
ponovimo postupak tako da se zavrtimo u petlju

⇒ NAPOMENA: ako nemamo zadan maksimalan broj redaka, valja koristiti DVOSTRUKE POKAZIVAČE (ovo dolazi u drugom dijelu semestra)

```
char** pp;
```

⇒ pomoću polja pokazivača, možemo pokazivat na različite lokacije u „heap" (dinamičkoj) memoriji

↳ to je korisno ako iz datoteke želimo podatke pospremit na način koj nam odgovara za daljnju obradu



char *pc[3];

POZIV FUNKCIJE

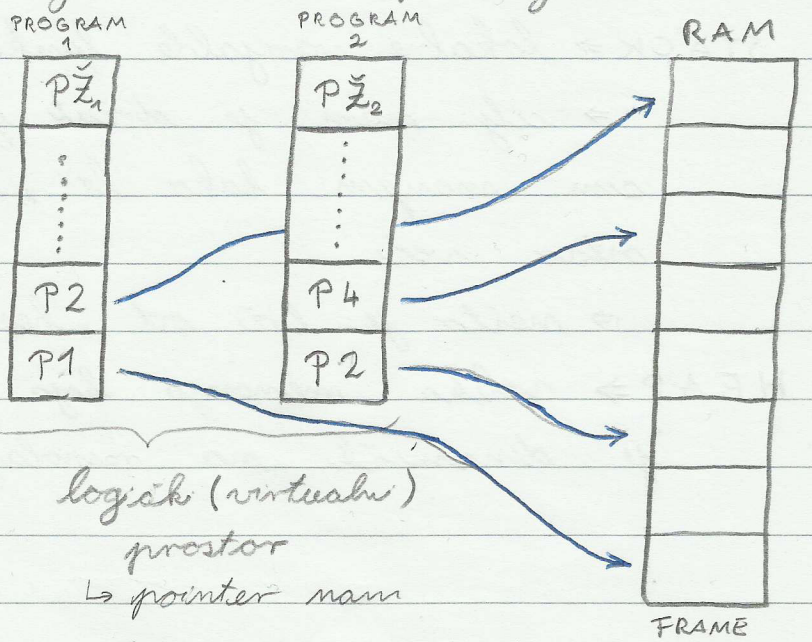
⇒ kad se pokrene nek program (funkcija), pridjeli mu se dio fizičke memorije

↳ to je VIRTUALNA MEMORIJA - program ima dojam da mu je dodjeljena sva računalna memorija

↳ program zna samo početnu adresu i veličinu memorije

↳ program ne zna za virtualne memorije ostalih programa (kad bi i znao, ne može im pristupiti - OS mu brani)

⇒ o dodjel i poveranost fizičke i virtualne memorije brine operacijski sustav



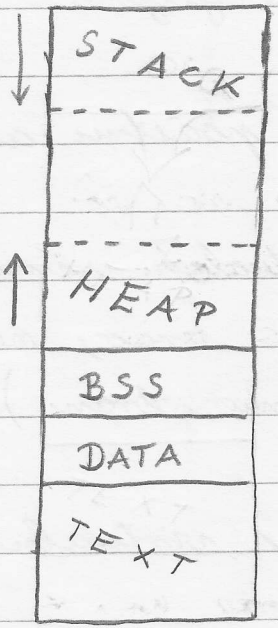
logički (virtualni) prostor

↳ pointer nam vraća adresu od tud

FRAME - fizička raspodjela rama kojima se dodjeljuju stranice (P1, P2, ...) logičke memorije

⇒ o raspodjeli stranica u frame-u brine operacijski sustav i MMU (memory management unit) koji je ustvari jedna brza Look-up tablica

⇒ SEGMENTI LOGIČKE MEMORIJE:



TEXT ⇒ ovdje je pohranjen kod programa

DATA ⇒ inicijalizirane globalne i statičke lokalne varijable

↳

BSS ⇒ neinicijalizirane globalne i statičke lokalne varijable

↳

STACK ⇒ lokalne varijable funkcije
⇒ cilj nam je držat ga čim manjim kako bi „heap" ostao veći
⇒ nešto je brži od „heap"-a

HEAP ⇒ velika memorija koja nam je dinamički na raspolaganju

⇒ STACK - služi za privremeno smještaj varijabli i povratnih adresa
- namjerno ograničen na određeni s vrha kako bi izbjegli programerske pogreške
↳ stavi na stack je PUSH
↳ uzmi sa stack-a je POP

⇒ STACK FRAME - ćelija koja se stavlja na stack
↳ sadrži:
a) povratnu adresu - adresa na koju se treba vratiti po završetku funkcije
- sprema se u procesor u PROGRAM COUNTER
b) lokalne varijable
c) argumente (parametre) funkcije
d) registre procesora koje može koristiti

⇒ napomena: stack je vrlo često malen (oko 1 MB) što je za „dobre" programere dovoljno

SLOŽENOST ALGORITAMA

⇒ matematičar Al Kovvarizmi:
↳ nastojao raščlaniti problem u jasne matematičke korake
↳ otac algoritama

⇒ ALGORITAM:
↳ precizan opis rješavanja nekog problema
↳ moraju biti definirani ulazni (početni) objekt
↳ algoritam mora biti:
a) EFEKTIVAN - daje rezultat
b) EFIKASAN - „što brže uz što manje utrošene memorije"

⇒ PROGRAM:
↳ implementacija algoritma u neko programsko jeziku koja jednoznačno opisuje što računalo radi

algoritmi
\+ strukture podataka
= program

⇒ postupci izrade algoritama zahtijevaju kreativnost, pa se sama izrada ne može algoritmizirati

⇒ ANALIZA SLOŽENOSTI ALGORITMA:

↳ dvije analize:

a) A PRIORI - prije pisanja koda, analiziramo efikasnost algoritma

b) A POSTERIORI - testiramo efikasnost implementiranog algoritma

⇒ O-notacija

↳ najrašireniji metoda provjere složenost algoritama

↳ nalazi funkcijsku ovisnost vremena izvođenja o broju ulaznih objekata

↳ radi se „a priori"

↳ vrijeme gledamo u odnosu na neku osnovnu operaciju (npr. x++; traje - jednu jedinicu vremena)

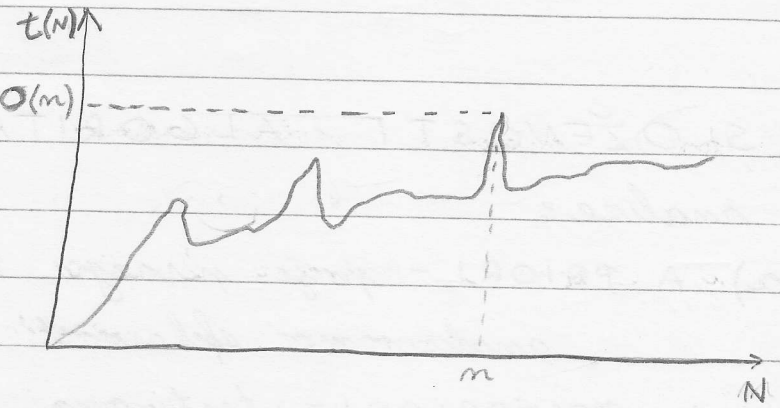
$$f(n) = O(g(n))$$

↳ ako postoje dvije pozitivne konstante c i n_0 takve da vrijedi:

$$|f(n)| \leq c \cdot |g(n)| \quad \text{za sve } n \geq n_0$$

↳ tražimo najmanji $g(n)$

↳ O - notacija prikazuje najgori slučaj



⇒ Ω - notacija

↳ isto kao i O - notacija samo što gleda najbolji slučaj

⇒ asimptotska notacija

$$f(n) \sim g(n) \text{ ako je } \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$$

↳ preciznija od O - notacije jer znamo red veličine i konstanta koja ga množi

TEHNIKE ADRESIRANJA

⇒ VRSTE KLJUČEVA:

a) primarni ključ: jednoznačno odredituje neki zapis

b) ulančani ključevi: potrebni ključevi za jednoznačno određivanje zapisa

c) sekundarni ključ: ne mora jednoznačno odredivat zapis, al ukazuje na neki atribut zapisa

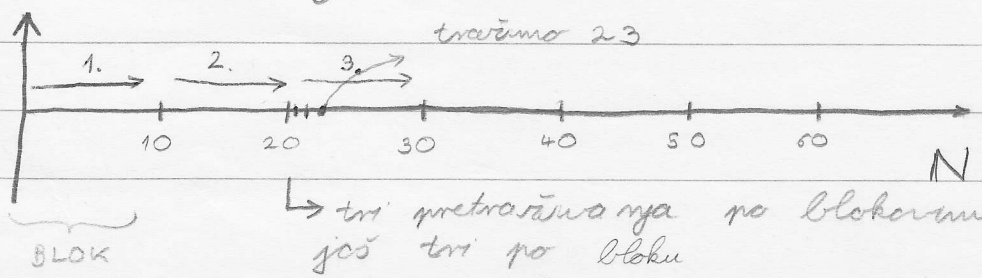
↳ mi ćemo se baviti samo primarnim ključevima

⇒ <u>traženje podatka u (sortiranoj) direktnoj datoteci:</u>

a) pomičemo se za neku veličinu bloka, npr 10, po podacima i gledamo nalazi li se traženi ključ ispred il iza naše pozicije

↳ kada se traženi ključ nalazi iza naše pozicije, onda idemo unutar bloka dok ne nađemo željeni podatak

↳ **optimalna veličina bloka:** $\sqrt{N}$, ako je N broj podataka



↳ tri pretraživanja po blokovima i još tri po bloku

⇒ izvod optimalnog bloka:

$$\left(\frac{N}{2B} + \frac{B}{2}\right)' = 0$$

prosječan broj pretraživanja po podatku

$$-\frac{N}{2B^2} + \frac{1}{2} = 0 \quad \Rightarrow \quad B = \sqrt{N}$$

↳ složenost ovog algoritma:

$$2\sqrt{N} = O(\sqrt{N}) = O(N^{1/2})$$

↓

vidimo da je bolje od traženja podatak po podatak koj ima složenost $O(N)$

↳ ovo se naziva:

"BLOCK SEARCH"

b) dijelimo datoteku na pola i gledamo je li traženi ključ više il niže od polovice

↳ postupak ponavljamo, pa imamo podatke koji dijelimo ovako:

$$N,\ N/2,\ N/4,\ \ldots\ldots,\ 1$$

učinili smo K koraka

$$f(N) = \log_2 N$$

↳ stoga, složenost ovog algoritma je:

$$O(\log N)$$

↳ ovo se naziva:

„BINARY SEARCH"

POBOLJŠANJE BRZINE PRETRAŽIVANJA

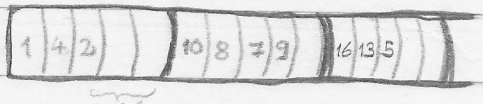
⇒ cilj nam je doći jednaku brzinu pretraživanja za bilo kakvu veličinu podataka

↳ složenost: O(1)

⇒ indeksno - slijedne datoteke:

↳ ideja je da svaki ključ ne sjedi baš na svoje mjesto u polju (datoteci) već smo datoteku podijelili na blokove do kojih se možemo direktno pozicionirat

↳ time smo smanjili zalihost, ali se datoteke trebaju održavat kada se prostor u određenom bloku popuni



zalihost - tu nadopunjavamo datoteku

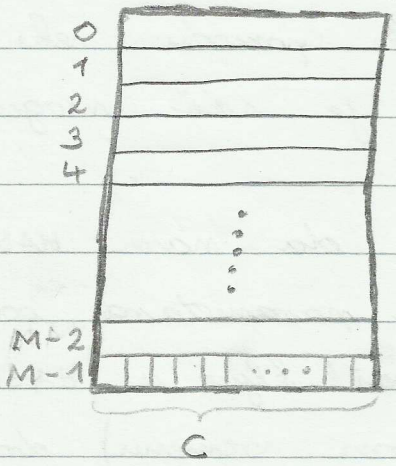
↳ ovo nije naše rješenje, ali može pomoć u pronalaženju rješenja

⇒ indeksno - neslijedne datoteke:

↳ sada nemamo ključ, već sortiramo pomoću neke funkcije ključa

↳ ovo se naziva raspršeno adresiranje ili HASHING

↳ imamo M pretinaca u HASH-tablici:



→ ako nam se u ključu promjeni 1 bit, želimo da naša HASH-funkcija na pseudo-slučajno (ne blisko) mjesto smjesti naš novi ključ u HASH-tablici

C → „kapacitet pretinca"

M → „broj pretinaca"

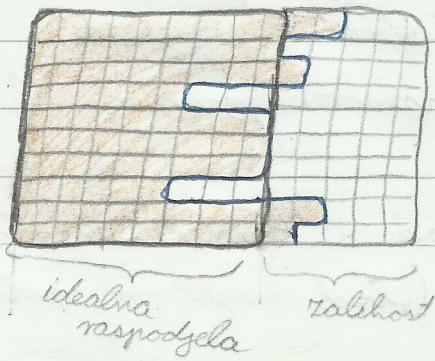
↳ O nije funkcija od N, već je funkcija od C, a C je konstanta

↳ želimo da naša HASH-funkcija ravnomjerno raspoređuje podatke po pretincima, tj. da ne prenapuni neke pretince, a neke ostavi gotovo prazne

↳ HASH-funkcija vraća adresu pretinca u koji upisujemo podatak

$$\text{gustoća zapisa} = \frac{N}{C \cdot M}$$

↳ na HASH-tablici, obično se uzima rablost od 30% idealne



— realna raspodjela

⇒ PRINCIP HASHINGA:

1) raspodjelimo podatke pomoću neke HASH-funkcije nasumično, ali što je više moguće jednoliko

↳ poželjno je da nam HASH-funkcija koristi što više parametara od podatka

2) sada, uvrstimo naš ključ u HASH-funkciju kako bi (u konstantnom vremenu) doznali u kojem je pretincu naš podatak

3) nadalje, po pretincu slijedno pretražujemo dok ne nađemo željeni podatak -

↳ primjetimo da nam vrijeme koje je potrebno da nađemo podatak u pretincu, ovisno samo o veličini pretinca C (uz mala odstupanja), a ne o broju podataka N

⇒ NAPOMENA: između dva realna pretinca, općenito se nalazi nešto praznog prostora jer to poboljšava performanse čitanja

⇒ ako je pretinac koji HASH-funkcija prilikom raspoređivanja podataka odabere popunjen, onda:

a) možemo pomoću „realloc" funkcije povećat odabran pretinac

b) staviti podatak u sljedeći slobodni pretinac („bad neighbour policy")

REKURZIJA

⇒ rekurzija je procedura (funkcija) koja poziva samu sebe

↳ napomena: rekurzija mora imati zaustavak

⇒ rekurzivno zadavanje funkcija u matematici:

npr. $f(n) = n \cdot f(n-1)$

$f(0) = 1 \; ; \; f(1) = 1$

rekurzivna definicija faktorijela

⇒ OSNOVNI SLUČAJ:

↳ „u kojem slučaju evaluiramo vrijednost bez ulaženja u daljnju rekurziju"

↳ svaka (ispravna) rekurzija mora imati osnovni slučaj

⇒ algoritmi složenosti $O(N)$ nisu pogodni za izvedbu rekurzijama jer će brzo probiti stack

↳ algoritmi složenosti $O(N \log N)$ i $O(\log N)$ već jesu

⇒ FIBONACCIJEV NIZ:

↳ svaki član niza je zbroj prethodna dva člana

$$F_i = F_{i-1} + F_{i-2}$$

```
int F(int n)
{ if (n<=1)
    return 1;
  else
    return F(n-1) + F(n-2);
}
```

↳ kod je jednostavan, al je složenost ovog algoritma 2^{n-1} što je prilično loše jer je ovo eksponencijalna složenost

⇒ NAJVEĆA ZAJEDNIČKA MJERA:

↳ najveći zajednički djelitelj

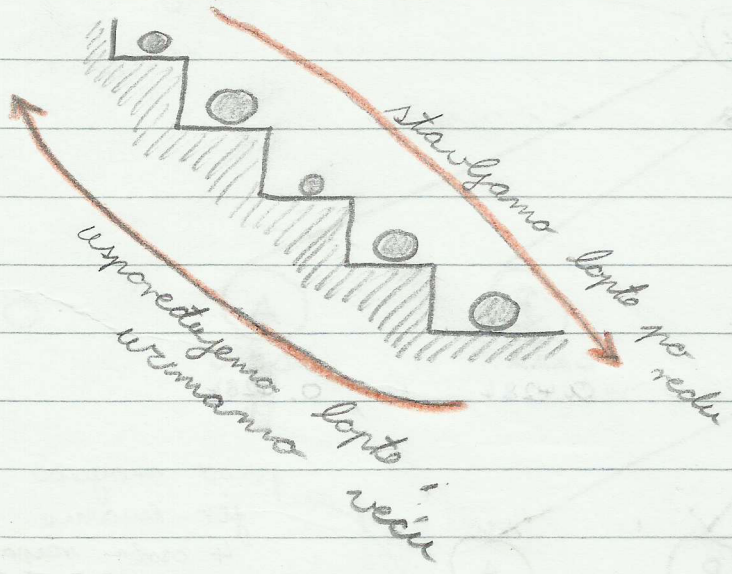
```
int nzm(int a, int b)
{ if (b==0)
    return a;
  return nzm(b, a%b);
}
```

⇒ KAKO PRONAĆI NAJVEĆI ELEMENT U POLJU POMOĆU REKURZIJA:

ISPIT!

↳ proglasimo prvi za najveći
↳ ratim, smanjujemo polje i opet proglasimo prvi za najveći sve dok ne dođemo do zadnjeg
↳ kada se vraćamo gore, uspoređujemo elemente te tako na poslijetku imamo maksimalan

⟶ analogija sa stubištem:



najčešći način rješavanja rekurzija

⇒ pokušaj realizirat „binary-search" pomoću rekurzija

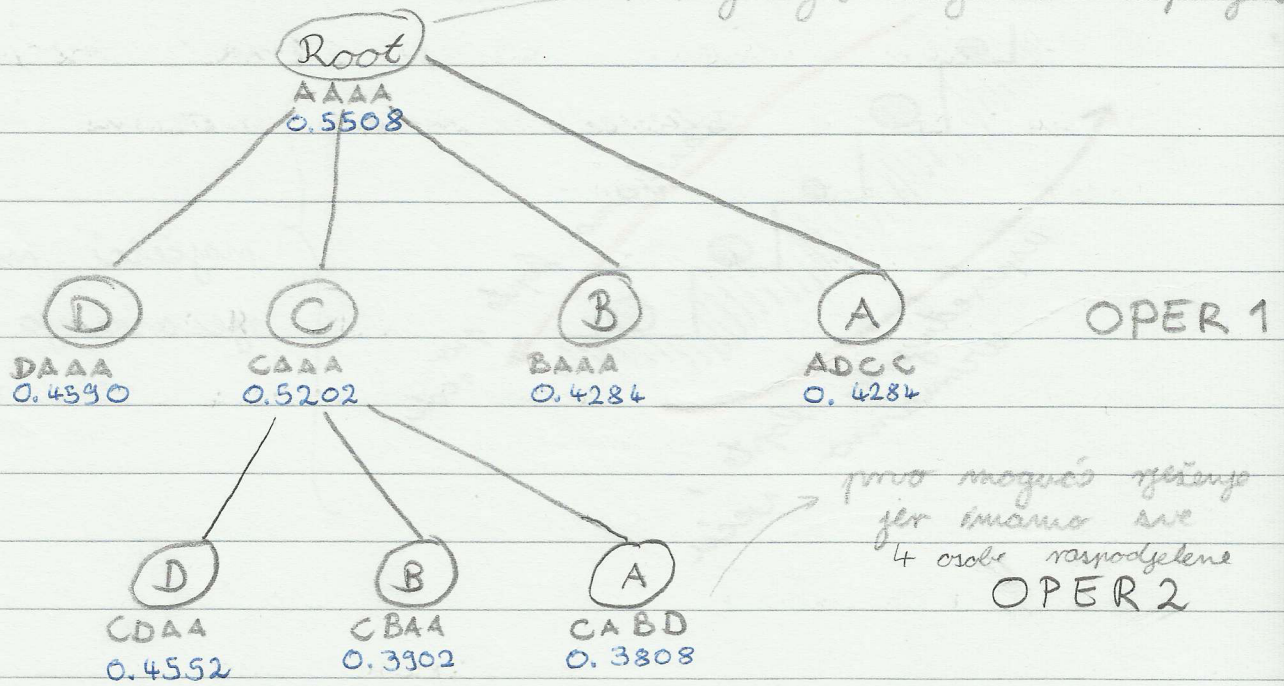
⇒ PROBLEM:

↳ imamo podatke za radnike o uspješnost obavljanja pojedinog posla

↳ trebamo najbolje raspodjelit poslove da uspješnost bude najbolja

		OPERATION			
		1	2	3	4
TEAM	A	0.9	0.8	0.9	0.85
	B	0.7	0.6	0.8	0.7
	C	0.85	0.7	0.85	0.8
	D	0.75	0.7	0.75	0.7

→ najbolja moguća raspodjela



vraćamo se do kraja i dobivamo rješenje

SORTIRANJA

⇒ SELECTION SORT:

↳ nađi najmanji element i zamijeni ga s prvim elementom polja

↳ taj postupak ponavljaj i tako smanjuj nesortiran dio polja

↳ složenost: $O(n^2)$

⇒ BUBBLE SORT:

↳ radi zamjenu susjednih elemenata ako nisu dobro poredani

↳ tako dobijemo najveći element polja na kraju te u sljedećem prolazu možemo raditi jedan korak manje

↳ složenost: $O(n^2)$

↳ ubrzanje: ako pri prolasku kroz niz nije bila zamjena, niz je sortiran

⇒ INSERTION SORT:

↳ na ispravno mjesto u sortiran niz umetni element iz nesortiranog niza

↳ složenost: $O(n^2)$

↳ dobar algoritam, ali smo još uvijek na istoj složenosti

⇒ SHELL SORT:

↳ uvjet za „k"-sortirana polja

PR: tri-sortirano polje:

↳ to je modifikacija insertion sort-a

↳ složenost: $O(n^2)$

⇒ MERGE SORT:

↳ rekurzivno „podijeli pa vladaj"

↳ složenost: $O(n \log_2 n)$

↳ cijena bržeg sortiranja je memorija jer stvaramo pomoćno polje

⇒ QUICK SORT:

↳ najbrži poznat algoritam sortiranja

```
QS(int A[], int L, int R)
{ if(R > L)
    i = partition(L, R)
    QS(a, L, i-L)
    Qs(a, i+1, R)
}
```

↳ kako ovo radi:

⇒ partition: sve elemente manje od stožera stavi s lijeve strane stožera

↳ metoda odabira stožera je proizvoljna

↳ naravno, odabir stožera može utjecat na performanse algoritma

PR: [5] 3 1 2 7 9 10

↳ stožer → utvrdi njegovu poziciju

3 1 2 (5) 7 9 10

L' R' L'' R''

ponovi postupak — ponovi postupak

↳ ako dobijemo već sortirani niz i te ako imamo loš izbor stožera, složenost je $O(N^2)$!

PR: 1 2 3 4 5 6 [7] — N

↳ stožer

N ratova

1 2 3 4 5 [6] — N - 1

↳ stožer

1 2 3 4 [5] — N - 2

↳ stožer

$$(N+1) \cdot \frac{N}{2} = \frac{N^2}{2} + \frac{N}{2}$$

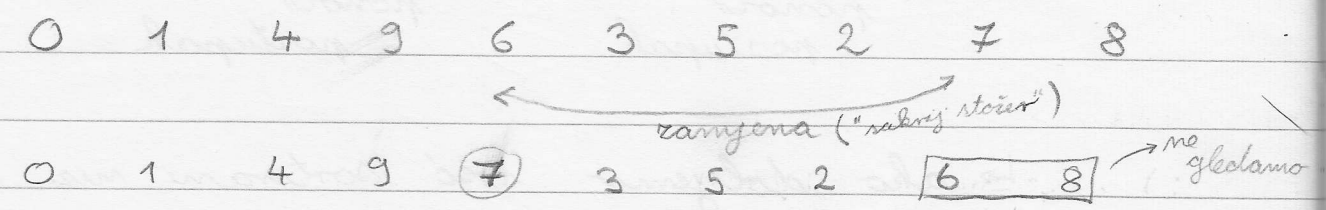
$$O(N^2)$$

↳ kako pravilno izabrati stožer?

↳ PROCJENA MEDIJANA:

↳ uzmemo prvi, srednji i zadnji element polja, tj.

↳ ta tri elementa sortiramo i uzimamo element koji se nalazi u sredini polja za stožerni element



↳ krenemo s lijeve strane i gledamo je li neki broj na krivoj strani polja (> 6)

↳ kada je, krenemo s desne strane i tražimo broj koji bi trebao biti na lijevoj strani (< 6) kako bi ga zamijenili

↳ ako pravilno odaberemo stožer (najbolja je metoda procjene medijana), šanse da se dogodi složenost $O(N^2)$ su zanemarivo male, pa je nova složenost $O(n \log_2 N)$

STOG

⇒ struktura podataka kod koje se posljednji pohranjeni podatak uzima prvi za obradu

↳ prednosti ⇒ dodavanje i brisanje traje isto bez obzira na broj pohranjenih podataka

⇒ pristup samo posljednjem podatku (manja mogućnost pogreške)

⇒ IMPLEMENTACIJA:

a) <u>statičkim poljem</u>

↳ ažurira se vrijednost varijable koja predstavlja vrh stoga

```
typedef struct
  { int vrh, polje [MAXSTOG];
    } Stog;
```

⇒ u radu sa stogom, definiramo tri funkcije:

1) `void stog_init (Stog* stog)`

```
{ stog -> vrh = -1 ;   // dereferenciranje pointera na strukturu
/* ↳ možemo pisat i ovako:
   (*stog). vrh = -1 ; */ }
```

2) `int dodaj (int element, Stog *stog)`

3) `int uzmi (int element, Stog *stog)`

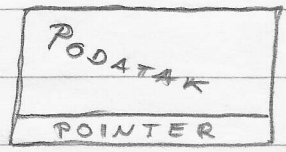
⇒ bolja realizacija stoga je dinamičnim poljem:

```
#define INC 10

typedef struct {
    { int vrh, size;
      int *polje; } Stog;

void init (Stog *s)
    { s->vrh = -1;
      s->polje = (int *) malloc (INC * sizeof (int));
      s->size = INC; }

int doday (int el, Stog *s)
    { if (s->vrh == (s->size - 1))
        { s->polje = (int *) realloc (s->polje, s->size + INC);
          s->size += INC; }
    }
```

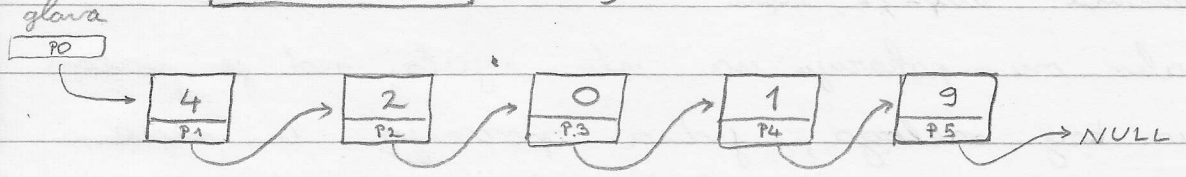
↳ treba dodati i provjeru je li uspjela dinamička alokacija memorije

LISTE

⇒ strukture podataka koje se sastoji od uredenog niza elemenata



} jedan element liste (ATOM)



↳ nedostatak je veće zauzeće memorije (malo), te je ŕetanje kroz listu u $O(n)$

↳ prednost povezane liste je dodavanje i brisanje elemenata u $O(1)$ jednostavnom igrom s pointerima

↳ lista je gotova uvijek dinamički realizirana

⇒ REALIZACIJA:

```
struct at
  { int element;
    struct at *sljedi; };

typedef struct at atom;
```

} najbolja realizacija

↳ imamo tipizirani pokazivač koji je lagano dereferencirati (umjesto „void" pointera koji uvijek očekuje cast-ing)

```
int main(void)
  { atom * glava;
    glava = NULL;
    ⋮
```

REDOVI

⇒ linearna lista kod koje se umetanje u listu izvodi na jednom, a brisanje na drugom kraju

↳ načelo First In First Out

⇒ imamo pokazivače ulaz i izlaz

↳ ako on pokazuje na isto mjesto, red je prazan

↳ iz tog razloga, jedna pozicija u redu mora biti prazna kako bi razlikovali pun red od praznoga

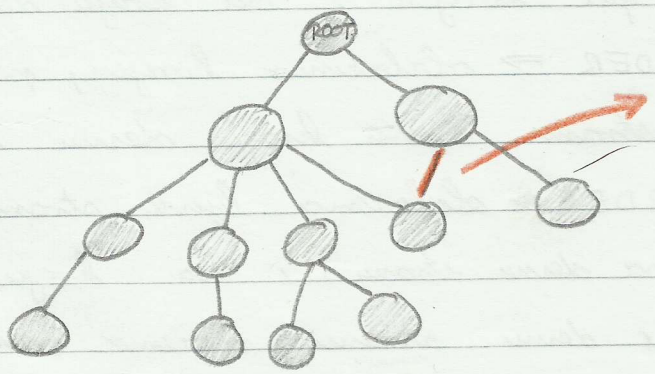
STABLA

⇒ najbolji primjer stabla u računarstvu je datotečni sustav

⇒ **stablo** - struktura podataka koja se sastoji od konačnog broja čvorova
↳ postoji **korijen** (root) što je ustvari samo jedan poseban čvor
↳ ostali čvorovi su podijeljeni u „k" podskupova od kojih je svaki također stablo (tzv. podstabla)

⇒ glavna razlika između stabla i liste je nelinearnost - stabla su nelinearna („kada smo odabrali jedan put niz stablo, moramo se vratiti prema „gore" kako bi izabrali neki drugi put")

⇒ bitno je da su svi od svih „k" podskupova disjunktni, tj. da se ne preklapaju



ovo nije stablo
(to je općenitija struktura podataka koja se naziva graf)

⇒ BINARNO STABLO ⇒ stablo koje se sastoji od čvorova drugog stupnja

⇒ POTPUNO STABLO ⇒ stablo popunjeno do kraja, osim radnje razine s tim da se radnja razina puni s lijeva na desno

⇒ PUNO STABLO ⇒ potpuno popunjeno stablo (uključujući i radnju razinu)

⇒ napomena: u obradi podataka strukturiranih u stablo, vrlo često se upotrebljavaju rekurzije

⇒ napomena: uvijek je bolje da neka funkcija koja radi s standardnim strukturama podataka (npr. stog, lista, stabla, ...) automatski mijenja vrijednost pokazivača (i ostale vrijednosti koje očekujemo promijenjene) bez oslanjanja na samu povratnu vrijednost funkcije

⇒ standardne obilasci stabla:

a) IN-ORDER ⇒ obilazimo lijevu stranu, pa korijen i na kraju desnu stranu

b) PRE-ORDER ⇒ obilazimo korijen, pa lijevu stranu i na kraju desnu stranu

c) POST-ORDER ⇒ obilazimo lijevu stranu, pa desnu stranu i na kraju korijen

↳ naravno, lijevu i desnu stranu možemo zamijeniti (još uvijek je to standardno ispis)

⇒ BRISANJE IZ STABLA:

↳ slučaj kada brišemo čvor koj ima dvoje djece je prilično ramrošen

↳ bitno je da očuvamo konzistentnost strukture (npr. manji su uvijek na lijevoj strani)

HEAP

⇒ prioritetni red ⇒ struktura podataka koja je realizirana na principu „First In, First Out", ali poruka s najvišim prioritetom se uvijek nalazi na vrhu iz naše strukture

↳ prioritetni red nije sortirana lista zbog toga što bi tada složenost bila $O(n)$, nego se samo na vrhu nalazi najprioritetnija poruka

⇒ heap (gomila) nam nudi realizaciju prioritetnog reda sa složenostima operacija:

a) ADD ⇒ $O(\log N)$

b) REMOVE ⇒ $O(\log N)$

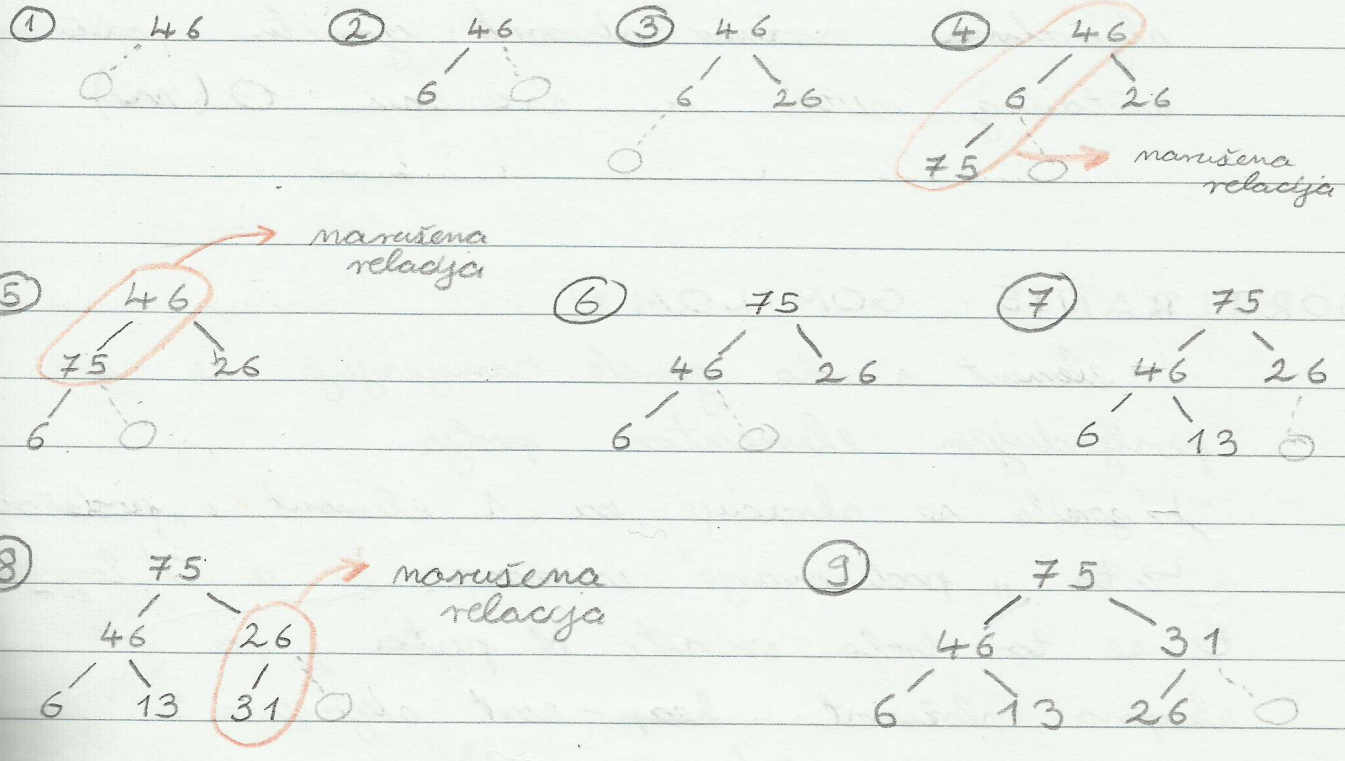
⇒ HEAP ⇒ potpuno binarno stablo gdje se čvorovi mogu usporedivati nekom relacijom

⇒ <u>oblikovanje gomile:</u>

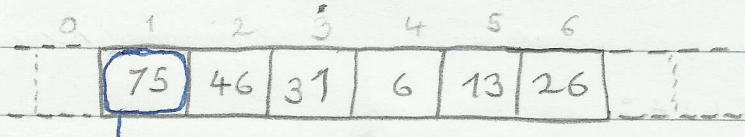
↳ čitamo elemente koje stavljamo na heap na slijedeći način i stavljamo ih tako na potpuno stablo da ne narušimo svojstva potpunog stabla

↳ ako smo narušili relaciju kojom uspoređujemo, trebamo to popraviti i to je moguće učiniti u $O(\log_2 N)$ koraka

PR: Dodajmo elemente 46, 6, 26, 75, 13, 31 u strukturu podataka heap na način da je roditeljski čvor uvijek veći od oba djeteta.



↳ na kraju, naše jednodimenzionalno polje izgleda ovako:



najveći element je na vrhu naše strukture, dok nam za ostale nije bitno u kakvom su redoslijedu

⇒ <u>zaključak</u> ⇒ stvaranje gomile od N elemenata zahtjeva složenost $N \cdot O(\log N)$

↳ postoji poseban slučaj koji ima $O(N)$

⇒ ako imamo već radan niz podataka, njih možemo jednostavno prepisati u niz podataka u našem programu

↳ nakon toga, pomoću „poseb nog" algoritma možemo stvoriti gomilu podešavanjem učitanog niza u vremenu $O(n)$

⇒ SORTIRANJE GOMILOM:

↳ element s vrha gomile zamjenjuje se s posljednjim elementom polja

↳ gomila se skraćuje za 1 element i „podešava"

↳ to „podešavanje" izvršava se u $O(\log_2 N)$

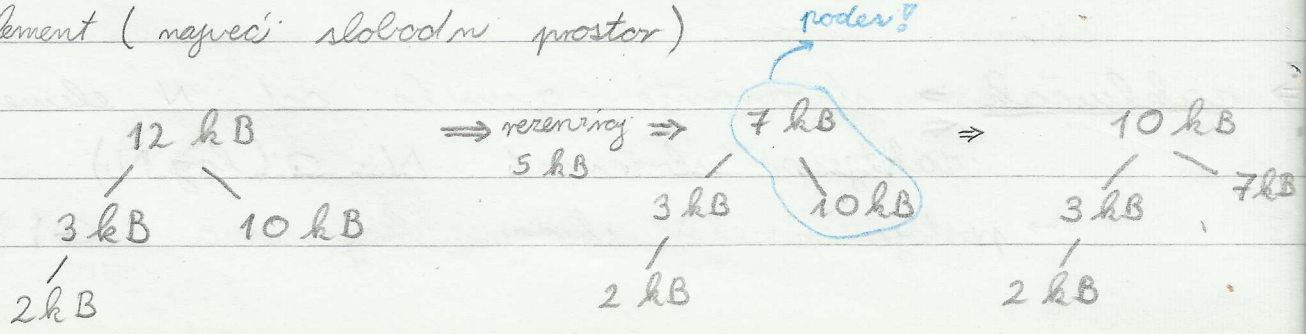
te se to treba izvršiti N puta, pa je ukupna složenost heap-sort algoritma:

$$O(N \log_2 N)$$

⇒ HEAP MEMORIJA:

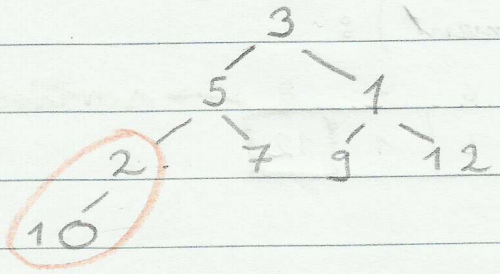
↳ radi na principu da vraća pokazivač iz najvećeg slobodnog dijela memorije (funkcija „malloc")

↳ nakon toga, heap ponovno podesuje svoju heap strukturu kako bi korijen ponovno bio najveći element (najveći slobodni prostor)

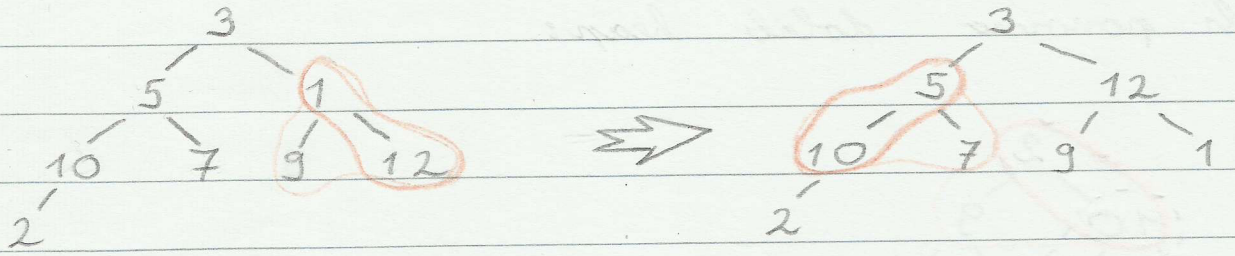


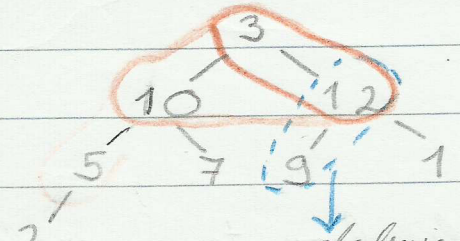
PR: Opiši sortiranje niza 3, 5, 1, 2, 7, 9, 12, 10 pomoću algoritma **heap-sort**:

↳ prvo, zadani niz interpretiramo kao statičko stablo:

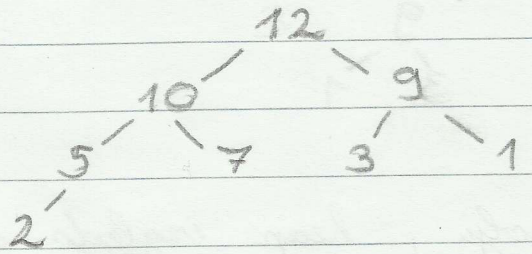


→ ovo nije heap, pa treba poderit stablo kako bi roditelj bio već od svoje djece





dalnje podešavanje broja 3 kako bi na broju dobili heap:

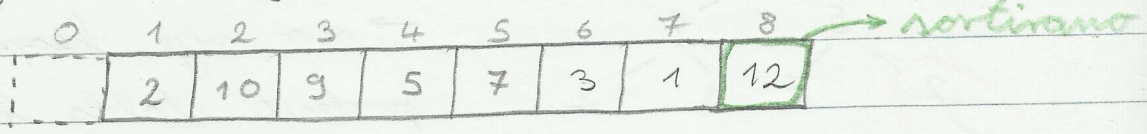


↳ naše statičko polje sada izgleda ovako:

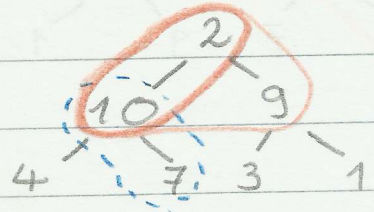
0	1	2	3	4	5	6	7	8
	12	10	9	5	7	3	1	2

OKRENI

↳ sada to polje sortiramo na način da zamijenimo prvi i zadnji element našeg polja (najveći element dolazi na kraj i njega smatramo sortiranim, pa sada našu gomilu smanjujemo za jedan element):

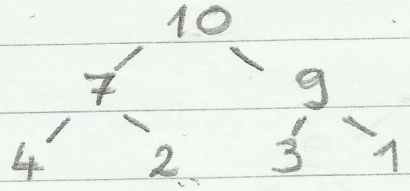


↳ nadalje, trebamo podesiti naše stablo kako bi ponovno dobili heap:

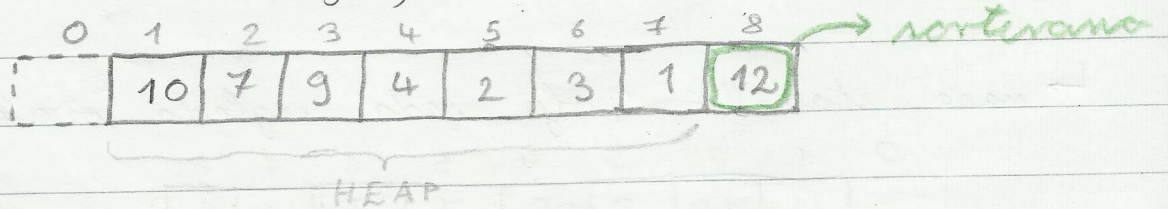


dalnje podešavanje glave
(maksimalna složenost $O(\log_2 N)$)

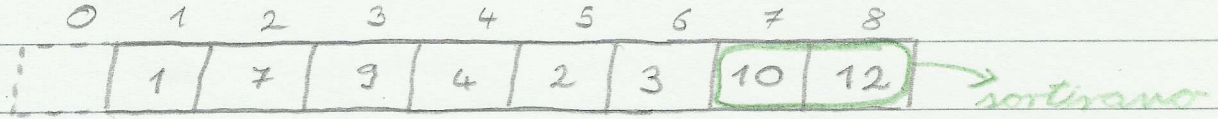
↳ sada smo uspješno stvorili heap te on izgleda ovako:



↳ u statičkom polju, heap izgleda ovako:



↳ sada samo prvih 7 elemenata tretiramo kao heap, te mijenjamo prvi i zadnji element heapa:



↳ vidimo da smo povećali sortirani dio, a smanjili nesortirani, pa ovaj postupak ponavljamo u petlji (ne rekurzivno!) dok nam je veličina polja veća od 1 (ako imamo jedan element, polje je već sortirano)