

21



Sortovi

↳ Selection sort → pronađi najmanji el. niza i zamjeni ga s prvim nesortiranim

```
for (int i=0; i<n; i++) {
    int min = i;
    for (int j=i+1; j<n; j++) {
        if (A[j] < A[min]) {
            min = j;
        }
    }
    Swap (&A[i], &A[min]);
}
```

$\Theta(n^2)$ ne stabilan

4	?	3	2
1	4	3	2
1	2	3	4
1	2	3	4
1	2	3	4

- najmanji
- sortirani dio

↳ Bubble sort → zamjeni susjedne elemente ako nisu dobro poređani

→ ubrzanje: ako u prolasku kroz cijeli niz nije bilo zamjene, niz je sortiran

```
for (int i=0; i<n-1; i++) {
    bool swapHappened = 0;
    for (int j=0; j<n-i-1; j++) {
        if (A[j+1] < A[j]) {
            Swap (&A[j], &A[j+1]);
            swapHappened = 1;
        }
    }
}
```

stabilan

$\Theta(n^2) \rightarrow$ obični

$\Omega(n) ; O(n^2)$
↳ poboljšani

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

- sortirani dio
- mješuric

(neću dalje raspisivati jer je sortiran do kraja)

↳ Insertion sort → sortirani i nesortirani dio niza, svakim korakom el. iz nesortiranog dijela ubacujemo na odgovarajuće mjesto u sortiranom dijelu

```
for (int i=1; i<n; i++) {
    int temp = A[i];
    for (int j=i; j>=1 && A[j-1] > temp; j--) {
        A[j] = A[j-1];
    }
    A[j] = temp;
}
```

stabilan

$\Omega(n) ; O(n^2)$
sortiran niz naopako sortiran niz

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

- sortirani dio
- el. koji ubacujemo

↳ Shellov sort → najstariji brzi algoritam, modificirani insertion sort (zadnji $h=1 \rightarrow$ ponara se kao insertion sort tada)
 → koristi se inkrementalni sljed brojeva $h_1, h_2, h_3, \dots, h_t$; izbor sljeda je jako bitan za učinkovitost algoritma

```
for(int step = n/2; step > 0; step = step/2){  
    for(int i = step; i < n; i++){  
        temp = A[i];  
        for(int j = i; j >= step && A[j-step] > temp; j = j-step){  
            A[j] = A[j-step];  
        }  
        A[j] = temp;  
    }  
}
```

ne stabilan

$O(n^{3/2})$

primjer sortiranja u prez., str. 79.

↳ Merge sort → niz se podijeli na dva jednaka dijela, taj postupak se ponavlja dok ne dobijemo niz duljine 1 (sortiran), nakon toga dva sortirana podniza spajamo u sortirani niz i tako dok ne sortiramo cijelo polje
 stabilan

$\Omega(1)$, $\Theta(n \log_2 n)$, $O(n \log_2 n)$

↳ ako je ulazni niz sortiran, ne radi ništa

↳ Quick sort → rekurzija: „podijeli pa vladaj“
 ne stabilan
 $\Theta(n \log n)$ $O(n^2)$

Stabla

↳ ful slično ko diskretna (usmjereni stabla)

↳ stupanj stabla: maks stupanj čvora (vrha), broje se samo "izlazni" vrhovi (neposredni potomci)

↳ razina (level) → korijen je razine 1 → razine djece čvora razine k su jednaki $k+1$

↳ dubina → max level stabla

↳ binarno stablo → stablo koje se sastoji od nijednog, jednog ili više čvorova drugog stupnja

→ razlikujemo lijevo i desno podstabla svakog čvora

→ maks broj čvorova na k-toj razini: 2^{k-1}

→ stablo dubine k s $2^k - 1$ elementima → puno (full) binarno stablo

→ stablo s n čvorova i dubine k je potpuno akko mu čvorovi odgovaraju čvorovima punog bin. stabla dubine k koji su numerirani od 1 do n

→ prikaz bin. stabla jednodimenzionalnim poljem
↳ problem: neefikasno umetanje i brisanje
↳ pravila za i-ti čvor stabla s n čvorova

- roditelj(i) = $\lfloor i/2 \rfloor$ za $i \neq 1$
- lijevoDijete(i) = 2^*i , $2^*i \leq n$, ako je $2^*i > n$, čvor nema lijevo dijete
- desnoDijete(i) = $2^*i + 1$, $2^*i + 1 \leq n$, ako je $2^*i + 1 > n$, čvor nema desno dijete

→ prikaz dinamičkom strukturom

```
template <typename T> struct Node {  
    T item;  
    shared_ptr<Node<T>> left;  
    shared_ptr<Node<T>> right;  
    Node(const T &newItem): item(newItem), left(nullptr), right(nullptr) {}  
};
```

↳ k-stabla → prirodna generalizacija bin. stabala.

→ k je stupanj stabla; $k \geq 2$, s istim mogućnostima prikazivanja

→ općenita stabla s raznim stupnjevima se mogu transformirati u bin. stabla → manji i učinkovitiji algoritmi, manja potreba za memorijom

!!! ostatak: pre 2 str. 76 na dalje !!!

Gomila

- ↳ prioritetski red \rightarrow struktura podataka kao obični red, samo što se ne skida podatak koji je prvi dodan, nego onaj koji ima najveću vrijednost (prioritet)
- ↳ prikazivanje pomoću sortirane verzane liste, sortiranog binarnog stabla ili gomile (najprirodnije i najčinkovitije)
- ↳ gomila je potpuno binarno stablo u kojem se čvorovi mogu uspoređivati nekom uređenom relacijom (npr. \leq) i gdje bilo koji čvor u smislu te relacije veći ili jednak od svoje djece (ako postoji)
- ↳ n elemenata gomile koji su prikazani potpunim binarnim stablom (i druga stabla mogu zadovoljiti)
 - \rightarrow prikazivanje ostvareno ubacivanjem jednog po jednog elementa u gomilu, čuvajući svojstvo gomile
 - \rightarrow na "duo" (list/gomile) dolje se član koji se uspoređuje sa svojim "predcima" sve dok ne postane manji ili jednak jednoj od tih vrijednosti

$$\hookrightarrow \underline{k} = \lceil \log_2(n+1) \rceil$$

↳ broj razina binarnog stabla

↳ $O(n \log n)$, $\Theta(n)$

↳ gomila s relacijom veći od \rightarrow max heap

↳ gomila s relacijom manji od \rightarrow min heap

↳ Heap sort \rightarrow element s vrha se zamjenjuje s posljednjim elementom polja \rightarrow gomila se skraćuje za 1 element i padašava
 \rightarrow to se ponavlja n puta, te s obzirom da za podešavanje vrijedi $O(\log n)$ \rightarrow sort: $O(n \log n)$

Grafovi

↳ diskretna!

↳ red grafa \rightarrow broj vrhova

↳ veličina grafa \rightarrow broj bridova

↳ orijentirani graf \rightarrow usmjereni graf (digraf) u kojem između dva vrha postoji samo jedan brid

↳ multigraf \rightarrow jednogstavan graf koji dozvoljava višestruke bridove, ali ne i petlje

↳ pseudografi \rightarrow multigrat koji dozvoljava i petlje

↳ obilazak grafa u dubini (DFS - depth first search)

\rightarrow složenost $\Theta(|E| + |V|)$

\rightarrow rekursivan algoritam grafa G krećeći od vrha v

procedura $DFSR(G, v)$

označi v kao obidan

ispisi v

za sve neobidene vrhove w susjedne vrhu v

$DFSR(G, w)$

\rightarrow nerekursivni algoritam (zahtjeva vlastitu implementaciju stoga) $\rightarrow \Theta(|E| + |V|)$

procedura $DFS(G, v)$

stavi v na stog S

dok ima elemenata na stogu S

skinji v sa stoga S

ako v nije obidan

označi v kao obidan

ispisi v

za sve neobidene w susjedne s v

stavi w na stog

↳ obilazak grafa u širinu (BFS - breadth first search)

→ složenost $\Theta(|E| + |V|)$

→ nerekurzivni algoritam

procedura BFS(G, v)

označi v kao obidan

ispisi v

stavi v u red Q

dok ima elemenata u redu Q

uzmi v iz reda Q

za sve neobidene w susjedne v

ako w nije obidan

označi w kao obidan

ispisi w

stavi w u red Q

↳ Dijkstra algoritam

↳ za zadani težinski graf G i početni vrh v_0 pronađi najkraći put do svih ostalih vrhova ili do određenog vrha v

Raspršeno adresiranje

- ↳ funkcija $hash(k)$ mapira vrijednost ključa u indeks pod kojim se ključ nađe u hash tablici
- ↳ pri pristupanju ključu se računa $hash(k) \Rightarrow$ složenost $O(1)$
- ↳ kolizija → više ključeva s istom hash vrijednosti
- ↳ izbjegavanje kolizija → direktno adresiranje - svaki ključ ima jedinstveno mjesto u tablici (velika neiskorištenost memorije)
 - savršena hash funkcija - svaki ključ ima jedinstveno mjesto u tablici (teška za konstruirati)
 - ublažavanje - elemente na istoj lokaciji u hash tablici preslikavamo u listu
 - vrijeme dodavanja $O(1)$, ali ne i za pristupanje
 - dobar način rješavanja kolizija ako ograničimo veličinu liste pokazivača
 - faktor opterećenja $\lambda = \text{broj ključeva} / \text{veličina tablice} \Rightarrow$ očekivano vrijeme pretrage $O(\lambda + 1)$
- otvoreno adresiranje - ne koristi se memorija izvan hash tablice → više memorije → veća hash tablica
 - manu → ograničena veličina tablice, $\lambda < 1$
 - pri ubacivanju el. u tablicu, slijedno se ispituje tablica dok se ne pronađe prazno mjesto
 - ↳ sljed lokačija ovisi o ključu
 - ↳ proširuje se hash funkcija slijednim brojem $hash(k, n), n \in [0, m]$

tehnike otvorenog adresiranja

- linearno ispitivanje - $h(k, i) = (h'(k) + i) \bmod m$
 - ↳ pomocna hash f-ja
 - u sljednoj sekvenci se pomičemo po susjednim mjestima dok ne nademo prazno mjesto
 - problem je brisanje - ako prilikom brisanja nađemo na prazno mjesto, ne možemo zaključiti da ne postoji taj element, rješeno ispitivanjem n elemenata ($n = \text{broj najduže sekvence}$)
 - problem linearne grupiranja
- kvadratno adresiranje - $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m; c_1, c_2 \neq 0, \text{const}$
 - rješen problem linearne grupiranja, ali postoji problem kvadratnog grupiranja: ako je $h(k_1, 0) = h(k_2, 0), \text{ slijedi } h(k_1, 1) = h(k_2, 1) \dots$, manji problem od lin. grupiranja

- \rightarrow dvostruko raspršeno adresiranje - $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$; h_1 i h_2 neovisne pomoćne f-je
 - izbjegnut problem grupiranja, permutacije dvostrukog raspršenosti imaju karakteristiku nasumičnosti
 - $h_2(k)$ mora biti relativno prost broj s obzirom na m (jedini zajednički djelitelj im je 1), kako bi cijela tablica mogla biti pretrazena, npr. m je potencija br. 2, a h_2 uvek vraca neparne brojere

\hookrightarrow analiza otvorenenog adresiranja

- $\rightarrow \lambda \leq 1$, a ako dodajemo element, $\lambda < 1$, jer $\lambda = 1$ znači da je tablica puna
- \rightarrow pretpostavljamo uniformno raspršeno adresiranje (svaka sekvenca $h(k, 0), \dots, h(k, m-1)$ je jednakog vjerojatna za svaki k (juč)
- \rightarrow očekivani broj ispitivanja prilikom neuspješne pretrage iznosi najviše $1/(1-\lambda)$

mali λ ($\lambda=0.1$)	$\lambda=0.5$	veliki λ ($\lambda=0.9$)
$\frac{1}{1-0.1} = \frac{1}{0.9} \approx 1$	$\frac{1}{1-0.5} = 2$	$\frac{1}{1-0.9} = \frac{1}{0.1} = 10$

\Rightarrow s obzirom da rezultati ovise o λ , dobra praksa držati λ između 0.5 i 0.6

\hookrightarrow karakteristike dobre hash f-je

- \rightarrow izlazna vrijednost ovisi SAMO o ulaznom podatku
 - \rightarrow f-ja koristi sve podatke (kada ih ne bi sve koristila, uz male varijacije ulaznih podataka bi dobivali iste izlazne vrijednosti (narušava se razdoblja))
 - \rightarrow jednolikost rasporedjuje ulazne vrijednosti;
 - \rightarrow za slične ulazne podatke daje vrlo različite izlazne vrijednosti (ulazni podaci su često slični, želimo ih ravnomjerno rasporediti)
- \hookrightarrow transformacija ključa u adresu u tri koraka

- ① ako ključ nije numerički, transformirati ga u broj (bez gubitka informacije)
- ② nad ključem upotrijebiti algoritam za transformaciju, što ravnomjerije, u pseudo slučajni broj reda veličine prethinca
- ③ rezultat se množi s odgovarajućom konstantom ≤ 1 , zbog transformacije u interval relativnih adresa (jednak broju pretinaca)

→ idealna transformacija - vjerojatnost da dva različita ključa u tablici veličine M daju istu adresu je $1/M$

↳ osmišljavanje hash f-ja

→ metoda dijeljenja - $h(k) = k \bmod m$

- m takav da nije potencija broja dva (ako je 2^r , onda hash f-ja ovisi samo o najniših r bitova ključa), t.j. izabiremo ga da bude prost broj koji nije potencijama brojera $10 ; 2$

→ metoda množenja - $h(k) = Lm(kA \bmod 1)$

- $A \rightarrow$ konstanta

- uzimamo ostatak dijeljenja kA s 1 (odnosi se dio za decimalne točke), množimo s M ; uzimamo njiveće cijelo te vrijednosti

- m ne mora biti prosti broj, najčešće potencija broja 2

- ako naše računalo radi s rječima širine w bita, ograničavamo A da bude $s/2^w$; $s \in \{0, 1, 2, \dots, 2^w\}$, te množimo ključ k sa s ; produkt je $2w$ bitna riječ $r_1 \times 2^w + r_0$; r_1 - prvih w znamenki, r_0 - drugih w znamenki

- željena p -bitna hash vrijednost sadrži p vodećih dijekti r_0 dijelci produkta

→ korijen iz srednjih znamenki kvadrata ključa

- ključ se prvo kvadrira, a zatim se uzme korijen iz srednjih $2n$ znamenki, ako je m n -terožnamenkasti broj

→ dijeljenje - ključ se podijeli prostim brojem približnom veličini pretinca, ostatak je adresa pretinca

→ preklapanje - npr. za ključ $172 \ 407 \ 359 \Rightarrow 407 + 953 + 277 = 7637$

→ izmjenica baze brojenja - ključ se prebaci u drugu bazu, uzme se potreban broj najmanje značajnih znamenki i transformira u raspon adresa

↳ kapacitet pretinca

→ ako je pretinac kapaciteta 1, česti preljevi

→ što veći pretinac, manja vjerojatnost preljeva, ali čitanje i pisanje duže traje

→ gustoća pakiranja = $\frac{N}{M} / C$

broj zapisa koje treba pohraniti \downarrow broj pretinaca \uparrow broj zapisa u jednom pretincu

→ postupak s preljevom

- koristenje primarnog područja - ako je pretinac pun, koristi sljedeći;

- efikasan kod veličine pretinca iznad 10

- ulančavanje - pretinci organizirani kao linearne liste

↳ koristenje raspršenog adresiranja nije prikladno ako:

- se podatke pretražuje po vrijednosti koja nije ključ
- se trazi da podaci budu sortirani

Pretraživanje znakovnih nizova

↳ osnovni algoritam

→ brute-force

→ $O(n*m)$; n - duljina ulaznog niza, m - duljina uzorka

```
for (int j=0; j<=n-m; j++) {  
    for (int i=0; i<m && pattern[i]==A[i+j]; i++) {  
        }  
        if (i==m) {  
            return j  
        }  
    }
```

↳ algoritam Rabin-Karp

→ algoritam računa sažetke podnizova

→ za svaki podniz ulaznog niza (veličine uzorka) računa sažetak, ako se sažetci podudaraju, tek onda uspoređuje uzorak i podniz

→ $O(n*m)$, pretprocesiranje $\Theta(m)$

→ 2 petlje, vanjska prolazi elementima ulaznog niza, unutar nje za prolaz po m znakova uzorka i usporedbu (ako se sažetci podudaraju)

→ za izračunavanje sažetka se koristi poseban algoritam; prez. str. 10.

↳ algoritam Knuth-Morris-Pratt

→ slično kao i osnovni algoritam, samo što još preskace usporedbе koje su višak, viškovi se određuju pomoću polja LPS (Longest proper prefix which is also suffix) koje ovise o uzorku

→ građuje LPS polja; prez str. 74.

→ pretprocesiranje $\Theta(m)$, usporedba $\Theta(n)$

Dotički

↳ Sortovi - Bubble (troboljšani), Insertion i Selection → kod

- crtanje svih, najčešće Shell, Quick ; Heap

sve elemente

sortiranog bin.
stabla

↳ Stabla - kod (standardne f-je: inorder, preorder, postorder, zbroji, pomnoži, dodavanje, razmjenje)
- crtanje sortiranog bin. stabla

najčešće

↳ Grafovi - DFS, BFS → crtanje + kod !!!

- Dijkstra → crtanje + NADOPUNJAVANJE koda ili jednostavne f-je

↳ Hash - uklanjanje i otvoreno adresiranje (samo kod)

↳ Gomila - 2 algoritma $\Theta(n)$; $\Theta(n \log n)$

↳ nadopuni kod

- crtanje → najčešće

↳ Stringovi - naivni → crtanje + kod

- RK → crtanje

- KMP → crtanje, LPS → kod i crtanje