SORTIRANJE

Sortiranje se moze definisati kao proces preuredjivanja skupa podataka po nekom utvrdjenom poretku.

Jedna od osnovnih svrha sortiranja je omogucavanje efikasnijeg pretrazivanja. Sortiranje je korisno ako treba proveriti jednakost dva skupa podataka.

Zbog velike vaznosti, razvijen je citav spektar metoda razlicite efikasnosti. Posebna paznja se posvecuje vremenskoj slozenosti koja se krece u relativno sirokom opsegu, od O(n) do $O(n^2)$.

Izbor algoritma sortiranja uzima u obzir i prostornu slozenost jer neki algoritmi zahtevaju veci ili manji dodatni prostor da bi se sortiranje obavilo.

U skupu kljuceva koji se sortira nije nuzno da sve vrednosti budu razlicite. ***Za metod sortiranja se kaze da je stabilan ako pocetni poredak zapisa sa istom vrednoscu kljuca ostane ocuvan u sortirnom nizu.***

Po mestu sortiranja gde se nalaze podaci koji se uredjuju postoje dve grupe metoda:

- **Unutrasnje sortiranje –** se koristi za podatke koji mogu da stanu u operativnu memoriju, obicno u formi niza ili tabele
- **Spoljasnje sortiranje –** uredjuje velike skupove podataka koji se nalaze na spoljasnjim memorijama organizovane u datoteke

12.1 Sortiranje poređenjem

Svi metodi ovoga tipa su zasnovani isključivo na međusobnom poređenju vrednosti ključeva odgovarajućih zapisa koji se sortiraju. **Pokazuje se da su najbolje performanse kod ovih metoda u srednjem i najgorem slučaju ograničene na $O(n \log n)$.**

Glavni pristupi se mogu svrstati u četiri grupe:

- **metodi umetanja (INSERTION-SORT, SHELL-SORT)**
- **metodi selekcije (SELECTION-SORT, stablo binarnog pretrazivanja, stablo selekcije, HEAPSORT)**
- **metodi zamene (BUBBLESORT, SHAKESORT, QUICKSORT, pobitno razdvajan**
- **metodi spajanja (COUNTING-SORT)**

12.1.1 Metodi umetanja

Ova grupa metoda se zasniva na principu postepenog uređivanja niza, tako što se u svakom trenutku održava uređeni i neuređeni deo. U svakom koraku se uzima jedan element iz neuređenog dela i umeće na odgovarajuće mesto u uređenom delu, koji na taj način raste. Predstavnici ove grupe metoda su: **direktno umetanje i umetanje sa smanjenjem inkrementa.**

Direktno umetanje (Zadatak 1, 54.pitanje)

Algoritam za sortiranje direktnim umetanjem je realizovan procedurom **INSERTION-SORT**. Ulaz u proceduru predstavlja neuređeni niz ključeva $a[1{:}n]$, koji ona na kraju napravi uređenim.

INSERTION-SORT(*a*)
for $i = 2$ **to** n **do** // indeksi elemenata iz neuredjenog dela
$K = a[i]$
$j = i - 1$
while $(j > 0)$ and $(a[j]) > K)$ **do**
$a[j + 1] = a[j]$
$j = j - 1$
end_while
$a[j + 1] = K$

end_for

opis rada algoritma:

U početku se uređeni deo sastoji samo od prvog elementa niza, a u neuređeni deo spadaju svi ostali elementi (n-1).

U svakoj iteraciji u okviru unutrašnje petlje uzima se tekući element kao prvi element iz neuređenog dela, pa se upoređuje redom sa elementima uređenog dela, prvo sa poslednjim, pa sa pretposlednjim, itd., sve dok se ne naiđe na prvi element u uređenom delu koji nije veći od tekućeg elementa. Pritom se elementi uređenog dela koji su veći od tekućeg elementa pomeraju za po jedno mesto naviše i tako prave mesto za njegovo umetanje. Sortiranje se završava kad nema više elemenata u neuređenom delu.

Metod je stabilan jer kad ključ koji se umeće dođe do jednakog ključa u uređenom delu, on se stavlja neposredno iza njega.

Performanse

Najbolje performanse direktno umetanje postiže kada je niz već uređen. Tada je vremenska složenost je $O(n)$.

Najgori slučaj predstavlja niz sortiran u obratnom poretku, pa su i broj poređenja i broj premeštanja reda veličine

$$O(\sum_{i=2}^{n}(i-1)) = O(n^2).$$

Prosečni slučaj je bolji od najgoreg samo za konstantan faktor 1/2 tako da performanse ostaju reda $O(n^2)$.

Poboljsanja:

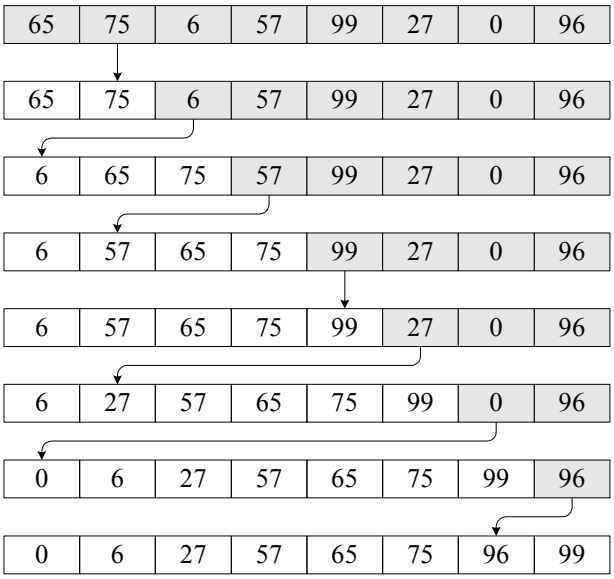
Osnovne operacije direktnog umetanja su pretraživanje uređenog dela i umetanje tekućeg elementa na odgovarajuće mesto. Činjenica da je donji deo, u koji se umeće tekući element, već sortiran, omogućava da se optimizuje prva od ove dve operacije. Tako se, umesto sekvencijalnim pretraživanjem kao u osnovnoj varijanti, mesto umetanja može naći efikasnijim, binarnim pretraživanjem. Ova varijanta se naziva ***binarnim umetanjem***.

Sada se prosečan broj poređenja u svakoj iteraciji umesto ***i*/2** može smanjiti na približno **log *i***, što smanjuje ukupan broj poređenja na ***O*(*n* log *n*).** Međutim, kada se nađe mesto umetanja, broj premeštanja elemenata ostaje isti, pa on dominantno određuje složenost, tako da ona ipak ostaje $O(n^2)$.

U nekim posebnim slučajevima, čak, performanse mogu i da se pogoršaju. Na primer, u slučaju već uređenog niza kada je mesto umetanja uvek na kraju uređenog dela, direktno umetanje ga pronalazi sa samo jednim poređenjem, a binarnom umetanju treba **log *i*** poređenja.

Broj premestanja bi mogli da smanjimo ako bi koristili jednostruko ulančanu listu umesto niza. Kod ove varijante postoji i dodatni vektor *next* koji simulira pokazivače. Na ovaj nacin bi smanjili broj premestanja, ali bi nam bilo kriticno poredjenje tako da i u ovoj varijanti vremenska složenost metoda i dalje ostaje $O(n^2)$. Nedostatak je svakako i potreban dodatni prostor za pokazivače.

Primer:



Slika 12.3 Direktno umetanje

1. Kljuc 65 je u uredjenom delu, a ostali kljucevi su u neuredjenom delu. U prvom koraku se krene od kljuca 75, on se uporedi sa kljucem 65 i ispostavilo se da je veci i da ga ne treba dirati. **Sada uredjen deo cine 65, 75, a neuredjen (6, 57, 99, 27, 0, 96).**

2. Kljuc 6 je manji od 75, manji je od 65 pa ga stavljamo na pocetak **uredjenog niza (6, 65, 75), a neuredjeni niz cine: 57, 99, 27, 0, 96.**

3. Kljuc 57 sada umecemo u uredjeni niz izmedju 6 i 65. **Uredjeni niz cine: 6, 57, 65, 75. Neuredjeni niz cine: 99, 27, 0, 96.**

4. Kljuc 99 je veci od kljuca 75 pa ga stavljamo na kraj **uredjenog niza (6, 57, 65, 75, 99), a neuredjen niz cine: 27, 0, 96.**

5. Kljuc 27 iz neuredjenog niza, smestamo u uredjeni niz izmedju kljuceva 6 i 57. Sada uredjeni niz cine: **6, 27, 57, 65, 75, 99, a neuredjen niz cine: 0, 96.**

6. Kljuc 0 iz neuredjenog niza, smestamo na pocetak uredjenog niza. **Sada uredjeni niz cine: 0**, **6, 27, 57, 65, 75, 99, a neuredjen niz cini: 96.**

7. U poslednjoj iteraciji kljuc 96 se smesta izmedju kljuceva 75 i 99. Niz je sada uredjen: **0**, **6, 27, 57, 65, 75, 96, 99.**

Umetanje sa smanjenjem inkrementa (Zadaci 2 i 3, 55.pitanje)

Pokazuje se da veliki broj premeštanja u metodu direktnog umetanja dolazi zbog činjenice da se premeštaju samo susedni elementi. Bolji rezultati bi se mogli očekivati ako se umesto kratkih koraka omoguće skokovi na većoj distanci.

Metod prvo razdvoji početni niz na grupe tako što u svaku grupu svrstava elemente na ekvidistantnim pozicijama u nizu. Ovaj razmak između elemenata u grupi se naziva *inkrementom* h_1. Broj grupa, naravno, odgovara vrednosti inkrementa. Zatim se ove grupe posebno sortiraju primenom metoda direktnog umetanja i niz postaje h_1-sortiran.

U narednom koraku u nizu koji je nastao posle prvog koraka inkrement se smanjuje na $h_2 < h_1$ i tako formira manji broj grupa sa većim brojem elemenata, pa se one opet nezavisno sortiraju.

U svakom sledećem koraku isti postupak se ponavlja sa smanjenim inkrementom. Na kraju se postupak završava svođenjem inkrementa na 1 što znači da je čitav niz jedna grupa čijim se sortiranjem dobija konačan rezultat.

Algoritam sortiranja sa smanjenjem inkrementa je realizovan procedurom **SHELL-SORT.** Pored pretpostavke, kao i ranije, da se podaci za sortiranje nalaze u nizu $a[1:n]$, ovde se pretpostavlja da se vrednosti inkremenata nalaze u nizu $h[1:t]$.

```
SHELL-SORT(a, h)
for i = 1 to t do
        inc = h[i]
        for j = inc + 1 to n do
                y =a[j]
                k = j – inc
                while (k ≥ 1) and (y < a[k]) do
                        a[k + inc] = a[k]
                        k = k – inc
                end_while
                a[k + inc] = y
        end_for
end_for
```

Na početku je inkrement h veliki, pa su grupe male. Pokazuje se da svaki kasniji korak zadržava uređenost iz prethodnog koraka i još je povećava, što znači da h_{i+1}-sortiran niz ostaje i h_i-sortiran. Prema tome, u kasnijim koracima kad se grupe povećaju, one su već dosta uređene kao rezultat prethodnih koraka, pa se stvaraju uslovi za efikasniji rad direktnog umetanja u okviru grupe.

Performanse:

Efikasnost algoritma prvenstveno zavisi od broja inkremenata i njihovih vrednosti. U principu, može se izabrati bilo koja sekvenca inkremenata h_1, h_2, ..., h_t takva da je:

$h_{i+1} < h_i, \quad 1 \leq i < t$

$h_t = 1$.

Knuth predlaže izbor:

$h_{i-1} = 3h_i + 1$

$h_t = 1$

$t = \lfloor \log_3 n \rfloor - 1$

što daje sekvencu ...., 121, 40, 13, 4, 1.

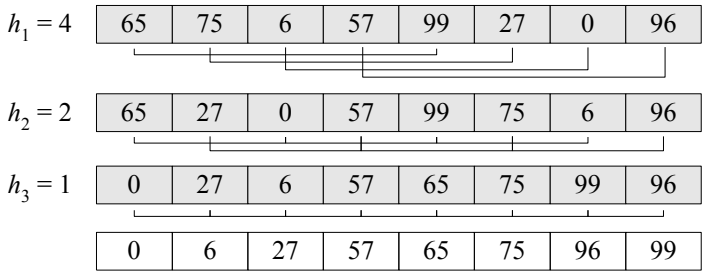
Primećeno je da je bolje ako su inkrementi međusobno prosti, jer to omogućava mešanje elemenata iz više grupa. Interakcija između grupa je bolja u slučaju uzajamno prostih inkremenata, a time se u ranijim koracima, kad su grupe manje, postiže bolja uređenost. **Pokazano je da se vremenska složenost algoritma može aproksimirati sa $O(n(\log n)^2)$. Empirijski rezultati pokazuju da se vreme izvršavanja može predstaviti sa an^b gde je a između 1.1 i 1.7, a b približno 1.26. Ovo je značajno poboljšanje u odnosu na direktno umetanje ($O(n^{1.3})$ umesto $O(n^2)$),** pa se, u opštem slučaju, ovaj metod preporučuje za nizove veličine od nekoliko stotina elemenata.

Primer:

U prvom koraku inkrement je 4. Ovde cemo uociti 4 podniza: 1-5, 2-6, 3-7, 4-8. U ovoj fazi se vrsi sortiranje u okviru ova 4 podniza. Imacemo nakon sortiranja po parovima: 65-99, 27-75, 0-6, 57-96. Citav niza nakon ove faze je: 65, 27, 0, 57, 99, 75, 6, 96.

U drugom koraku inkrement smanjio na 2. Ovde cemo uociti 2 podniza: 1-3-5-7, 2-4-6-8. U ovoj fazi se vrsi sortiranje u okviru ova 2 podniza. Nakon sortiranja, prvi podniz je: 0, 6, 65, 99. Nakon sortiranja, drugi podniz je: 27, 57, 75, 96. Citav niza nakon ove faze je: 0, 27, 6, 57, 65, 75, 99, 96.

U trećem, poslednjem koraku inkrement je 1. Sada se na citav niz dobijen u prethodnom koraku primeni metod direktnog umetanja i na taj nacin se dobije kompletno sortiran niz: 0, 6, 27, 57, 65, 75, 96, 99.



Slika 12.4 Umetanje sa smanjenjem inkrementa

12.1.2 Metodi selekcije

Princip ovih metoda je da se selektuje najmanji element iz neuredjenog dela i stavlja se na kraj uredjenog dela. Tipični primeri iz grupe metoda selekcije su: **direktna selekcija, sortiranje pomoću binarnog stabla i *heapsort*.**

Direktna selekcija (Zadatak 4, 61.pitanje)

Ovde se tokom rada algoritma održava uređeni i neuređeni deo niza. U početku čitav niz je neuređeni deo. Zatim se sekvencijalnim pretraživanjem ovog dela nađe najmanji element, pa on zameni mesto sa prvim elementom. Sada je taj element u uredjenom delu, a ostali elementi su u neuredjenom delu. U svakoj narednoj iteraciji se trazi najmanji element iz neuredjenog dela i on se postavlja na kraj uredjenog dela. Postupak se završava kad se neuređeni deo svede samo na jedan element, čime i on postaje uređen.

Algoritam je realizovan procedurom SELECTION-SORT. Algoritam se može ravnopravno implementirati tako da selektuje najveći element, pa onda uređeni deo raste od kraja niza ka početku.

```
SELECTION-SORT(a)
for i = 1 to n - 1 do
        min = a[i]
        pos = i
        for j = i + 1 to n
do
                if (a[j] >
min) then

        min = a[j]

        pos = j
                end_if
        end_for
        a[pos] = a[i]
        a[i] = min
end_for
```

Direktna selekcija pokazuje neke sličnosti, ali i razlike u poređenju sa direktnim umetanjem. Oba metoda održavaju uređeni i neuređeni deo, pri čemu se prvi povećava, a drugi smanjuje za po jedan element u svakom koraku. Međutim, dok direktno umetanje vrši poređenja u uređenom delu tražeći mesto umetanja za jedan element neuređenog dela, direktna selekcija poredi sve elemente u neuređenom delu. Prema tome, dok kod direktnog umetanja elementi mogu da dolaze jedan za drugim, direktna selekcija ne može da počne ako svi elementi nisu prisutni.

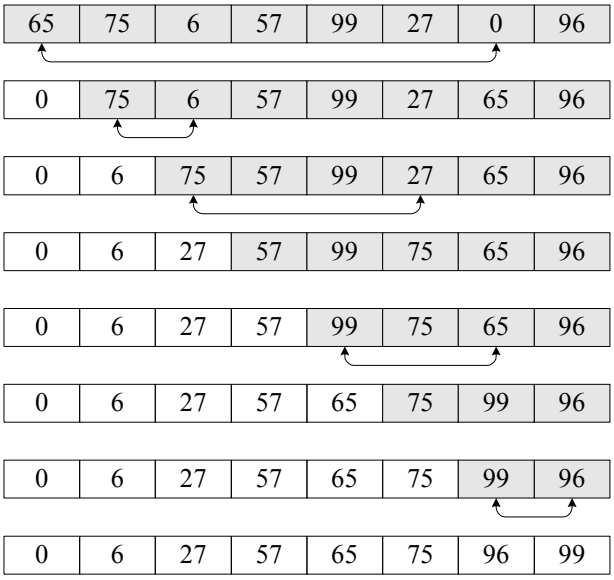
Performanse:

Složenost metoda je diktirana brojem poređenja kojih u svakom koraku ima $n - i$, pa je njihov ukupan broj

$$\sum_{i=1}^{n-1}(n-i) = n(n - 1)/2$$

što daje složenost $O(n^2)$, pa metod nije primenljiv za veće skupove podataka. **Broj poređenja ne zavisi od stanja prethodne uređenosti niza, pa se ne može razlikovati najbolji i najgori slučaj.**

Primer:



Slika 12.5 Direktna selekcija

1. U pocetku je citav niz neuredjen deo. Prodje se kroz citav niz i nadje najmanji clan, 0, koji zameni mesto sa prvim elementom, 65. **Sada uredjeni deo cini 0, a ostali elementi su u neuredjenom delu.**

2. Sada se prodje kroz neuredjen deo pa se najmanji element, 6, postavi na prvo mesto. Sada uredjen deo cine 0 i 6, a ostali elementi su u neuredjenom delu. Tako se u svakoj iteraciji uredjen deo povecava za jedan, a neuredjen smanjuje za 1.

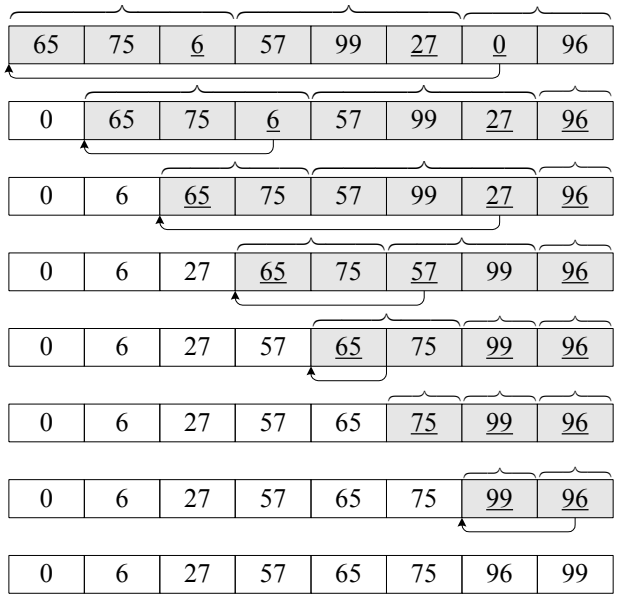
Kvadratna selekcija

Pošto je broj poređenja kritičan, poboljšanja su usmerena u pravcu smanjenja ovog broja. Jedan način da se izbegnu ponovna poređenja je organizovanje neuređenog dela u više grupa i nalaženje najmanjeg elementa u svakoj grupi - **lokalnog minimuma.**

Onda se selekcija globalnog minimuma svodi na nalaženje najmanjeg od lokalnih minimuma. Kada se nađe globalno najmanji element, on se ukloni iz svoje grupe i zatim se nađe novi lokalni minimum te grupe. Glavna ušteda je što u narednom koraku ne treba da se vrše ponovna poređenja u drugim grupama, jer njihovi lokalni minimumi ostaju isti, već je dovoljno samo poređenje lokalnih minimuma grupa.

Varijanta ***kvadratne selekcije*** deli početni niz u $\sqrt{n}$ grupa od po $\sqrt{n}$ elemenata. **Na ovaj način se red složenosti od $O(n^2)$ može popraviti na $O(n\sqrt{n})$.**

Primer:



Slika 12.6 Kvadratna selekcija

1. Najpre formiramo 3 grupe od po 3,3,i 2 elementa. Pronadjemo minimume u svakoj grupi. Ti minimumi su takozvani lokani minimumi, 6, 27, 0. Najmanji od ova tri minimuma je 0 koja kao globalni minimum odlazi u uredjen deo.
2. U drugoj iteraciji nastavljamo postupak s tim sto sada prva grupa ima 3 elementa, sa lokalnim minimumom 6, druga grupa ima 3 elementa sa lokalnim minimumom 27, a treca grupa ima samo jedan element, 96, koji je ujedno i lokalni minimum. Najmanji od ova tri elementa je 6 koji kao globalni minimum odlazi u uredjen deo odmah iza nule. U svakoj narednoj iteraciji se nalazi lokalni minimum pa minimum tih minimuma kao globalni minimum odlazi u uredjen deo.

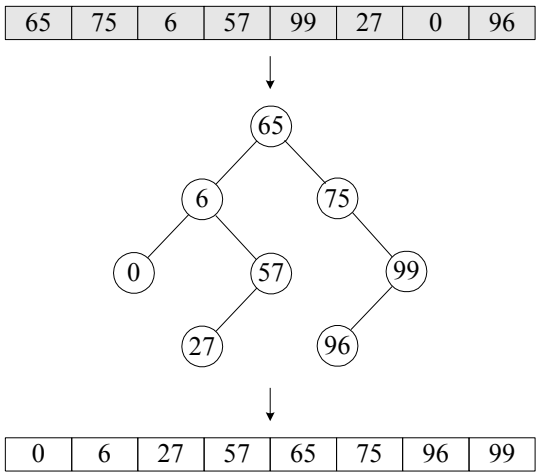
Sortiranje pomocu stabla binarnog pretrazivanja (Zadatak 5, 53.pitanje)

U prvoj fazi elementi neuređenog niza se uzimaju sekvencijalno i umeću u stablo binarnog pretraživanja. Prvi element postaje koren, a ostali dolaze na mesta saglasno svojoj vrednosti.

Jedini problem predstavljaju elementi sa jednakim vrednostima jer oni u izvornoj definiciji stabla binarnog pretraživanja nisu dozvoljeni. Kako sortiranje dozvoljava postojanje istih elemenata, onda se može usvojiti konvencija da se element sa manjom vrednošću stavlja u levo podstablo, a ako je veći ili jednak odlazi u desno podstablo.

Drugi način da se reši ovaj slučaj je da se za svaki čvor održava lista zapisa sa ključevima iste vrednosti. **Oba načina tretiranja istih ključeva omogućavaju stabilnost metoda**.

Kada se generiše stablo koje uključuje sve elemente zadatog niza, onda se njihov sortirani poredak može dobiti obilaskom stabla po *inorder* poretku. Ovaj obilazak praktično realizuje fazu selekcije.



Slika 12.7 Sortiranje pomoću stabla binarnog pretraživanja

Performanse

Performanse ovog metoda zavisi od poretka i odnosa vrednosti u početnom nizu. **Najgori slučaj** se javlja kad je ulazni niz već potpuno sortiran u neopadajućem ili nerastućem poretku. Tada se dobija degenerisano stablo u vidu linearne liste od *n* elemenata. **Red složenosti je $O(n^2)$.**

U najboljem slučaju, koji rezultira balansiranim stablom, performanse generisanja su reda $O(n \log n)$.

Prosečan slučaj je mnogo bliži najboljem slučaju nego najgorem, pa su prosečne performanse takođe reda $O(n \log n)$**, ali sa većim konstantnim faktorom**.

Dodatni *inorder* obilazak stabla je reda $O(n)$, što ne povećava red složenosti. Sa aspekta korišćenja prostora, ovaj metod ima nedostatak jer zahteva dodatni prostor za generisanje stabla. Ovaj algoritam je pogodan za dinamicke strukture podataka.

Sortiranje pomocu stabla selekcije (Zadatak 6, 56.pitanje)

Pored povećanog prostora koji zahteva, najveći nedostatak prethodnog metoda su slabe performanse za monotone ulazne nizove. Zato se koriste i druge vrste binarnih stabala. Evo jednog, koje se naziva ***stablo selekcije*** koje resava problem garantovane performanse.

U prvoj fazi se od elemenata neuređenog niza formiraju grupe od po dva elementa koji se mogu smatrati listovima stabla. Veći iz svakog para ide na viši nivo i postaje čvor grananja stabla, gde se opet vrši uparivanje susednih čvorova i princip generisanja stabla se rekurzivno nastavlja dok najveći element od svih ne postane koren stabla.

U fazi selekcije najveći element se bira vrlo jednostavno, jer se on nalazi u korenu, pa ga treba ukloniti iz stabla i smestiti u uređeni deo niza.

U narednom koraku se radi isti postupak, jedino se umesto prethodno izabranog najveceg elementa stavlja -∞ čime se praktično drugi element iz para upućuje na prazno mesto na višem nivou i upoređuje sa svojim parnjakom radi popunjavanja praznog mesta na sledećem nivou. Ovo se ponavlja sve dok drugi najveći element ne ispliva na vrh.

Performanse:

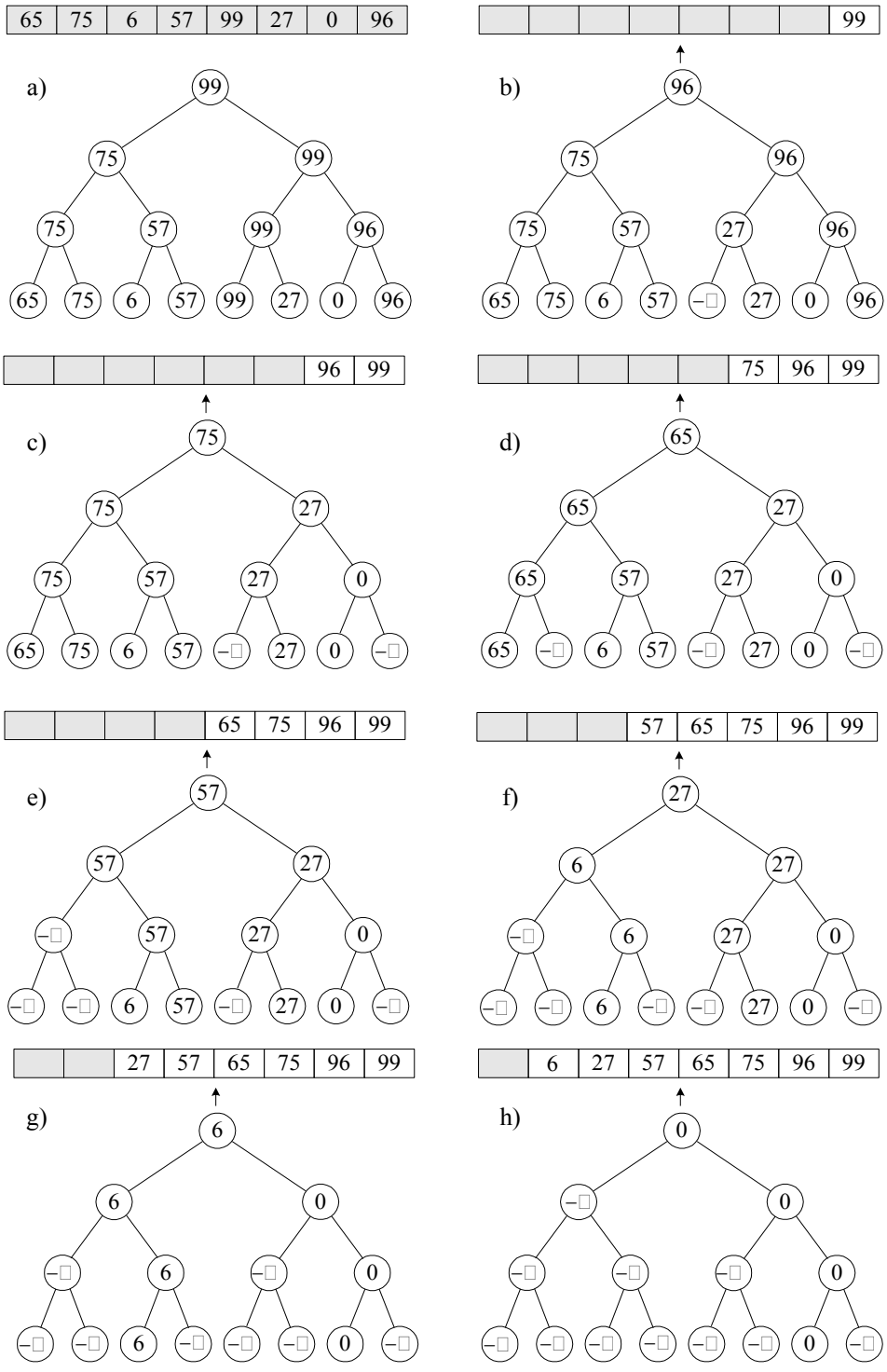
Faza generisanja stabla zahteva $O(n)$ poređenja, a faza selekcije najviše $O(n \log n)$ pa je ukupna vremenska složenost ***sortiranja pomoću stabla selekcije*** reda ***O*(*n* log *n*).** Kao i sortiranje pomoću stabla binarnog pretraživanja, **ovaj metod zahteva dodatni prostor *O*(*n*)** za formiranje stabla.

Primer:

Kljuceve iz nesortiranog niza smo smestili u listove pa smo poredili parove kljuceve, dva po dva kao kod kup takmicenja. U nasem primeru imamo 8 kljuceva, kao ¼ finale u takmicenju. Pobednici odlaze u ½ finale. To su 75 i 57, odnosno 99 i 96. Popodnici polufinala odlaze u finale. To su 75 i 99. Pobednik iz finala, kljuc 99, odlazi u uredjen niz na poslednje mesto.

U narednoj iteraciji, se sve isto radi, jedino se u listu umesto 99 stavlja -∞. Tako da sada u ¼ finalu imamo parove 75 i 57, odnosno, 27 i 96. Pobednici odlaze u ½ finale. To su 75 i 96. Popbednik iz finala, kljuc 96, odlazi u uredjen niz na pretposlednje mesto.

Postupak se nastavlja na isti nacin. Sada se umesto kljuceva 99 i 96 stavlja u listovima -∞.



Slika 12.8 - Sortiranje pomoću stabla selekcije: a) početno stanje, b)-h) iteracije sortiranja

Heapsort (Zadatak 7, 51.pitanje)

Pored zahteva za dodatnim prostorom za stablo selekcije, nedostatak mu je i to što se ono u fazi selekcije sve više popunjava sa elementima $-\infty$. Zbog toga se rade mnoga nepotrebna poređenja što umanjuje efikasnost. Prema tome, cilj je realizovati metod koji će zadržati prednosti, a otkloniti nedostatke osnovnog metoda preko stabla selekcije. Upravo te osobine su postignute kod algoritma ***heapsort*** koji zato predstavlja jedan od najboljih poznatih algoritama za sortiranje.

Heapsort je zasnovan na specifičnoj vrsti binarnog stabla koja se naziva *heap*.

***Heap** se definiše kao kompletno ili skoro kompletno binarno stablo koje poseduje svojstvo uređenosti takvo da je sadržaj čvora oca uvek veći ili jednak od sadržaja oba sina. Iz definicije sledi da koren stabla obavezno predstavlja najveći element u stablu. Svaka putanja koja vodi od korena do lista predstavlja nerastuću sekvencu elementa.*

Grub opis rada algoritma:

U prvoj fazi, na osnovu vrednosti elemenata neuređenog niza, generiše heap. Zatim, u fazi selekcije, u svakom koraku jednostavno izabere koren kao najveći element, ukloni ga iz heap-a i ubaci ga na početak uređenog dela niza. Naravno, pre sledećeg koraka treba reorganizovati smanjeno stablo da dobije novi koren, ali tako da opet zadovoljava svojstvo heap-a.

Pošto skoro kompletno stablo ima sve popunjene nivoe osim poslednjeg, a poslednji nivo je sukcesivno popunjen od levog kraja donekle, već ranije je naglašeno da se ono može efikasno implementirati u vidu niza. Ovakva sekvencijalna reprezentacija otklanja potrebu za pokazivačima. Cvorovi kompletnog ili skoro kompletnog stabla se u niz smestaju kad se ono obilazi po ***level-orderu***.

Po definiciji *heap*-a smeštenog u nizu $a[1:n]$, važi:

$$a[i] \geq a[2i] \text{ i } a[i] \geq a[2i+1], \quad 1 \leq i < 2i < 2i+1 \leq n .$$

Heap može sasvim ravnopravno da se definiše i organizuje tako da otac ima manju ili jednaku vrednost u poređenju sa sinovima, pa se u korenu pojavljuje najmanji element. Zbog svojih osobina, ***heap*** **je izuzetno pogodna struktura za implementaciju prioritetnog reda**, kako nerastućeg tako i neopadajućeg. Analiza performansi operacija pokazuje da je to jedna od najefikasnijih implementacija prioritetnog reda.

Heapsort **se sastoji od dve faze:**

- **generisanja *heap*-a**
- **selekcije elemenata sa ponovnim procesiranjem *heap*-a.**

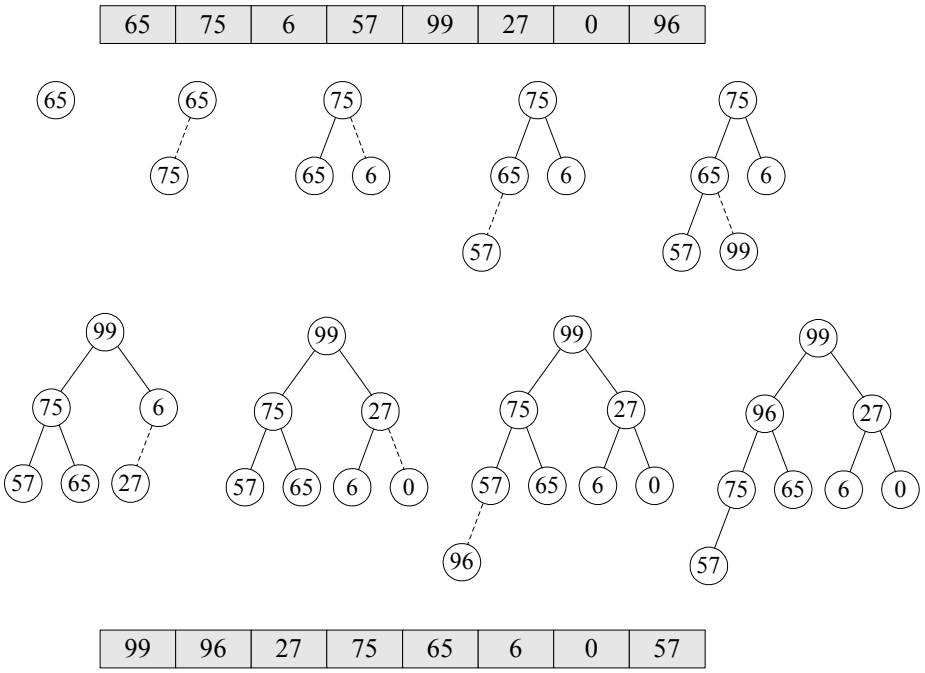
Primer:

Heap se generiše na mestu ulaznog niza sukcesivnim ubacivanjem po jednog člana, počevši od elementa *a*[1] koji za sebe predstavlja inicijalni *heap*. Posle *i* - 1 koraka deo niza *a*[1]..*a*[*i* - 1] predstavlja *heap*. Zatim se, u *i*-tom koraku, stablu priključuje element *a*[*i*] kao novi element *heap*-a *nhe* na mestu lista u skoro kompletnom stablu. Na osnovu njegovog indeksa *s* izračunava se pozicija oca *f* celobrojnim deljenjem sa 2.

Ako pri ubacivanju sin nije veći od oca, svojstvo *heap*-a je odmah zadovoljeno i ovaj korak je završen.

Ako je pri ubacivanju sin veći od oca, onda oni zamenjuju mesto i postupak se dalje ponavlja na višim nivoima sve dok na putu ka korenu element ne dođe do mesta u stablu gde je njegov otac veći ili jednak njemu ili se ne dođe do samog korena. Ovim je element zaustavljen u svojoj propagaciji naviše na odgovarajućem mestu čime je obezbeđeno da povećano stablo i dalje ima svojstvo *heap*-a.

Kada se uključe svi elementi ulaznog niza, formiran je konačan *heap* od *n* elemenata i može da počne faza selekcije.



Slika 12.9 Generisanje *heap*-a

1. Na pocetku imamo neuredjen niz od 8 elemenata. Prvo se izdvoji prvi element, 65, i on je heap sam za sebe.

2. Potom stize kljuc 75. Heap se prosiri za jedno mesto. Medjutim, kako je novouvedeni element veci od svog oca, 65, oni moraju da zamene mesta kako bi zadovoljili svojstvo heap-a.
3. Potom stize kljuc 6, on je manji od svog oca, kljuca 75 pa je sve u redu.
4. Potom stize kljuc 57, on je manji od svog oca, kljuca 65 pa je sve u redu.
5. Potom stize kljuc 99 i on je veci od svog oca, kljuca 65. Sa njim menja mesto, medjutim, on je opet veci od svoga oca, kljuca 75 pa i sa njim menja mesto.
6. Stize kljuc 27, on je veci od svoga oca, kljuca 6, pa sa njim menja mesto.
7. Sada stize kljuc 0, on je manji od svog oca, kljuca 27 pa je sve u redu.
8. Stize kljuc 96, on je veci od svoga oca, kljuca 57 pa sa njim zameni mesto. Medjutim, on je veci ponovo od svoga oca, kljuca 75 pa i sa njim zameni mesto. Kako nije veci od kljuca 99, ostaje tu gde jeste. Konacan izgled heap-a je dat na slici, skroz desno, dole.

   Ovim smo dobili jedan prioritetan red koji garantuje da je najveci element u korenu.

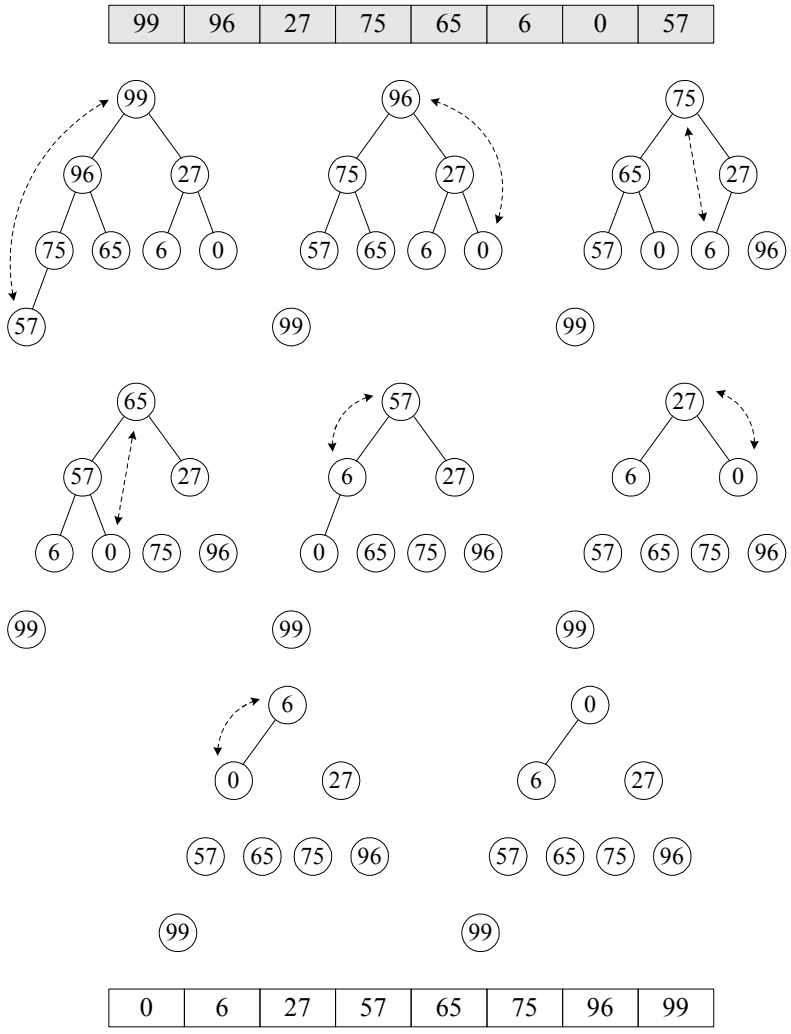
2.FAZA, FAZA SELEKCIJE :

Pošto se najveći element nalazi u korenu, selekcija se svodi na njegovo uklanjanje sa *heap*-a i prebacivanje u sortirani deo. Pošto je cilj da se sortiran niz ostavi na mestu ulaznog niza, ovaj element treba prebaciti na krajnju poziciju istiskujući tako element *heap*-a *last* koji se tamo nalazi. Pošto je mesto korena ostalo upražnjeno, na njega treba staviti element *last* ako se time zadovoljava uređenost *heap*-a, što znači da element *last* nije manji od sinova korena.

Ako ovaj uslov nije ispunjen, onda se element *last* zameni sa svojim većim sinom, koji postaje koren, a element *last* ide na nivo 1. Postupak se ponavlja i element *last* "pada" zamenjujući se sa većim sinom sve dok ne dođe na mesto gde nije manji od oba sina ili ne dođe na mesto lista.

U nasem primeru, kljuc iz korena zameni mesto sa kljucem 57 koji odlazi u koren, a kljuc 99 odlazi u sortirani deo. Kljuc 57 se poredi sa vecim od sinova, to je kljuc 96 pa „tone“ na tu stranu, a kljuc 96 odlazi u koren. Konacno, kljuc 57 se poredi sa kljucem 75 kao vecim sinom pa kako je i od njega manji, menjaju mesta.

U narednom koraku, kljuc 96 zameni mesto sa kljucem 0. Kljuc 96 odlazi u sortirani deo, a kljuc nula se usporedi sa vecim od sinova. Kako je manji od kljuca 75, sa njim zameni mesto, a kljuc 75 odlazi u koren. On je manji sada od kljuca 65 pa i sa njim zameni mesto. Ovde smo potpuno azurirali heap, a u sortiranom delu su za sada 99 i 96. Postupak se tako nastavlja sve dok se svi kljucevi ne nadju u sortiranom delu. Sve je prikazano na narednoj slici.



Slika 12.10 Selekcija i procesiranje *heap*-a

Postoji i alternativna implementacija sa procedurom ADJUST koja stablo, čiji je koren čvor *i*, a nijedan čvor nema indeks veći od *n*, pretvara u *heap*, pretpostavljajući da levo i desno podstablo čvora *i* već zadovoljavaju osobine *heap*-a.

Performanse:

U fazi generisanja *heap*-a, pri ubacivanju novog elementa, on može od lista najviše da se "izdigne" do korena. Tako on, u najgorem slučaju, može da pređe put jednak visini stabla zahtevajući samo po jedno poređenje i zamenu po nivou. Pošto je visina skoro kompletnog binarnog stabla $O(\log n)$, a ova operacija se ponavlja za n elementa, **gornja granica vremenske složenosti faze generisanja *heap*-a je $O(n \log n)$. Sa ADJUST procedurom slozenost faze generisanja je $O(n)$.**

U fazi selekcije, pri preuređivanju *heap*-a, kada krajnji element "propada" kroz *heap* do svog mesta, on prolazi kroz najviše onoliko nivoa kolika je visina stabla, **dakle opet *O*(log *n*).** Pritom se vrše najviše dva poređenja i jedna zamena po nivou, što za *n* čvorova **daje *O*(*n* log *n*), pa je to i ukupna vremenska složenost *heapsort*-a.**

Maksimalna vremenska složenost je određena topologijom skoro kompletnog binarnog stabla i ne zavisi od vrednosti i poretka elemenata u ulaznom nizu, tako da se performansa reda O(n log n) uvek može garantovati. Prema tome, velika prednost heapsort-a jer to što daje dobru performansu i u najgorem slučaju. U prosečnom slučaju performansa je, takođe, reda O(n log n), ali sa nešto većim konstantnim faktorom od particijskog sortiranja.

Algoritam nije stabilan.

Pored garantovane perfromanse najgoreg slučaja *heapsort* je od velike praktične važnosti u primenama koje ne traže potpuno sortiranje čitavog niza od *n* elemenata, već samo manji broj najvećih elementa. Neka se traži *k* elemenata pri čemu je $k << n$. Kao što je pokazano, generisanje *heap*-a se može obaviti u $O(n)$. Zatim se vrši samo *k* koraka za prvih *k* najvećih elemenata u fazi selekcije koji traju po $O(\log n)$. Ukupno trajanje je onda $O(n + k \log n)$, pa ako je $k < n/\log n$ onda je zahtevano vreme blisko $O(n)$.

12.1.3 Metodi zamene

U ovoj grupi metoda zamena je glavni mehanizam na kojem se zasniva sortiranje. Zamena dva elementa se obavlja uvek kada se utvrdi da oni nisu u pravilnom poretku. U zavisnosti od toga da li se zamenjuju susedni elementi ili udaljeni elementi i na koji način, razlikuju se **metode direktne zamene, particijskog sortiranja i pobitnog razdvajanja.**

Direktna zamena - Bubblesort

Algoritam više puta sekvencijalno prolazi kroz niz i pritom upoređuje svaki element sa narednim u nizu, pa ako ova dva elementa nisu u pravilnom poretku, zamene im se mesta.

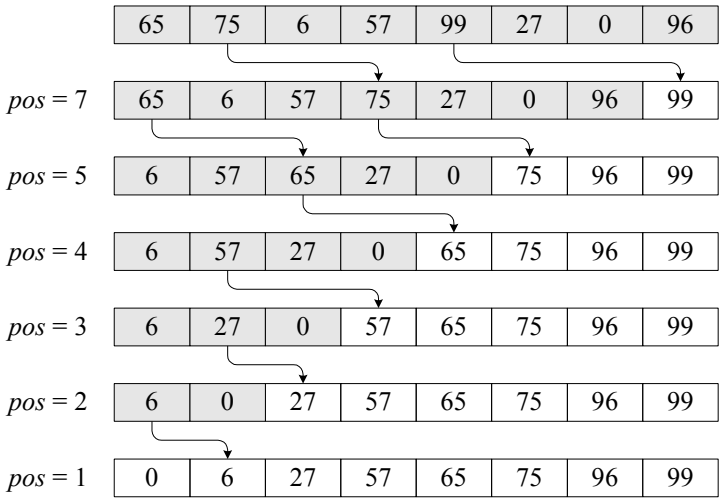
U prvom koraku najveći element sigurno dođe na poslednje mesto i tako u svakom prolazu bar jedan element dođe na svoje mesto. Kada $n - 1$ viših elemenata budu na svojim mestima, tada je i najmanji element na svom mestu, pa je zato potrebno najviše $n - 1$ prolaza za sortiranje čitavog niza.

Osnovna varijanta metoda direktne zamene može da se optimizuje u cilju skraćenja vremena sortiranja. Može da se primeti da u svakom koraku ne treba vršiti proveru svih elemenata, jer su posle *i*-tog koraka *i* najvećih elemenata na svojim mestima. Tako je svaki naredni korak sve kraći. Pored toga, ukoliko se zapamti najviša pozicija u nizu na kojoj je izvršena zamena, pošto su očigledno svi elementi iznad te pozicije u pravilnom poretku, u narednom koraku ne mora da se vrši provera tih elemenata. Prema tome, postupak može da se završi i u manje od $n - 1$ koraka. Štaviše, ako u nekom koraku nije izvršena nijedna zamena, to znači da ovaj korak nije doprineo sortiranju, jer su svi elementi na svojim mestima, pa naredni

koraci nisu ni potrebni. Ove optimizacije algoritma su implementirane u proceduri BUBBLESORT.

Promenljiva *pos* pamti u svakom koraku najvišu poziciju sa koje je neki element zamenio mesto sa svojim desnim susedom, pa u narednom koraku ne treba ići preko te pozicije jer je gornji deo niza već uređen. Kako se na početku svakog koraka ova promenljiva postavlja na 0, ako takva ostane i na kraju koraka, onda zamena nije ni bilo i algoritam završava rad.

Primer:



Slika 12.11 Direktna zamena

1. 65 se uporedi sa 75, u dobrom su poredku pa se staje. 75 je veci od 6 i 57, ali je manji od kljuca 99 tako da ga zaustavljamo odmah iza 99. Drugim recima, kljuc se probija ka kraju sve dok ne dodje do kljuca koji je veci od njega. Tu staje, a onda na isti nacin taj, veci kljuc nastavlja da se probija ka kraju dok ne dodje do kljuca koji je veci od njega ili dok ne dodje do samog kraja niza. Na taj nacin, nakon prvog koraka, najveci kljuc sigurno dolazi do kraja. To je u nasem primeru kljuc 99.
2. Kljuc 65 ide do prvog veceg, to je kljuc 75. 75 se pomera do prvog veceg, to je kljuc 96. Kako je kljuc 96 manji od kljuca 99 koji je vec na svome mestu, nakon ovog koraka su kljucevi 75, 96 i 99 zauzeli svoja mesta na samom kraju. Proces se tako nastavlja do se niz kompletno ne sortira.

Performanse:

Najbolji slučaj - kada je ulazni niz već sortiran. Tada se sve završava u jednom koraku sa $C_{min} = n - 1$ poređenja i $M_{min} = 0$ zamena, pa je složenost ***O*(*n*).**

Najgori slučaj - kada je niz sortiran u obratnom poretku. Tada je potrebno $n - 1$ koraka sa $C_{max} = 0.5(n^2 - n)$ poređenja i isto toliko zamena, $M_{max} = 0.5(n^2 - n)$, pa je složenost $\boldsymbol{O(n^2)}$.

Prosečan slučaj - prosečan broj poređenja je $C_{ave} = 0.5(n^2 - n \ln n)$, a prosečan broj zamena $M_{ave} = 0.25(n^2 - n)$. **Dakle, u svakom slučaju složenost ovog algoritma je $\boldsymbol{O(n^2)}$.**

Shakersort

Metod direktne zamene pokazuje izvesnu nesimetriju u ponašanju zavisno od rasporeda ključeva. Na primer, niz

99 0 6 27 57 65 75 96

će biti sortiran u samo jednom koraku, jer veći ključevi brzo idu ka svojim mestima udesno. Međutim, za sortiranje niza

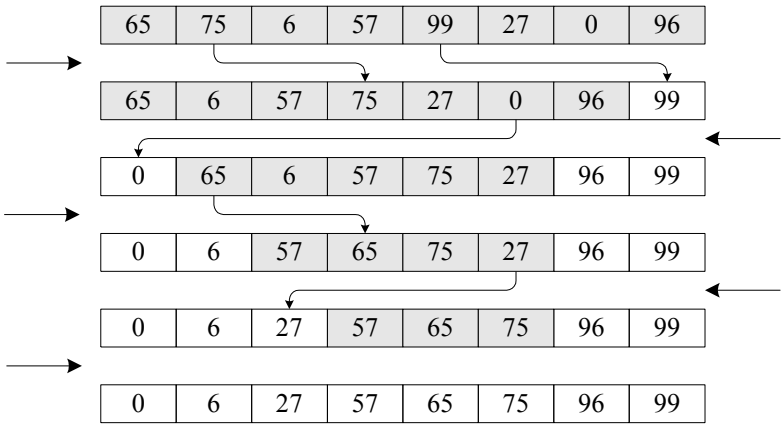
6 27 57 65 75 96 99 0

će biti potrebno 7 koraka, ***jer se manji ključevi sporije probijaju ka svojim mestima ulevo. Naime, manji ključevi mogu da se pomeraju ulevo samo za po jedno mesto po koraku. Ova pojava nameće ideju da se smer u kojem se vrše poređenja alternativno menja iz koraka u korak, ujednačavajući brzinu probijanja manjih i većih ključeva do svojih mesta. Ova varijanta direktne zamene se naziva shakersort.***

Performanse:

Analize pokazuju da se broj poređenja neznatno smanjuje i da u svakom slučaju performansa ostaje istog reda složenosti, $O(n^2)$.

Primer:



Slika 12.12 - Varijanta direktne zamene – *shakersort*

1. Krecemo sleva udesno. Kljuc 65 poredimo sa kljucem 75, kako je manji od njega, nediramo ga. Sada kljuc 75 pomeramo udesno sve dok ne dodjemo do kljuca koji je veci od njega. To je kljuc 99. Sada kljuc 99 pomeramo udesno i prakticno dolazimo do kraja jer je on najveci.
2. Sa krecemo od kraja, od kljuca 96 koji poredimo sa kljucem 0, pa kako je veci, njega ostavljamo tu, a nastavljamo od kljuca 0 koji pomeramo u levo sve dok ne dodjemo do kljuca koji je manji od njega ili dok ne dodjemo do pocetka niza. Ispostavlja se da je kljuc 0 najmanji pa ga stavljamo na pocetak.

3. U trece koraku, krecemo s desna u levo od kljuca 65. Njega pomeramo u levo sve dok ne dodjemo do kljuca koji je veci od njega. To je kljuc 75. Sada kljuc 75 pomeramo u desno sve dok ne dodjemo do kljuca koji je veci od njega, a to je kljuc 96. Konacno, ispred kljuca 96 je samo kljuc 99 koji je vec u sortiranom delu pa je i kljuc 96 sada u sortiranom delu. Na ovaj nacin se nastavlja sve dok se niz skroz ne sortira.

Particijsko sortiranje - Quicksort (Zadatak 8, 59. & 60. pitanje)

Može se zaključiti da je broj premeštanja elemenata kod direktne zamene kritičan. Zato je, kao i kod ostalih direktnih metoda, performansa ograničena na $O(n^2)$. Prirodno se javlja ideja slična onoj u *shellsort*-u da se pokuša zamena između udaljenih elemenata da bi elementi brže dolazili na svoje mesto.

Particijsko sortiranje se zasniva na principu zamene na većoj udaljenosti uz dinamičko prilagođavanje sekvence poređenja.

Algoritam počinje tako što na određen način izabere jedan element niza koji se naziva razdvojnim elementom ili ***pivotom***. Zatim se niz reorganizuje tako da pivot dođe na poziciju j u nizu, a svi ostali elementi se razvrstaju u dve nesortirane particije koje ispunjavaju sledeće uslove:

- donju particiju čine elementi koji su manji od vrednosti pivota ili jednaki sa njom, $a[i] \leq pivot$, $1 \leq i \leq j-1$,
- gornju particiju čine elementi koji su veći od vrednosti pivota ili jednaki sa njom, $a[i] \geq pivot$, $j+1 \leq i \leq n$.

S obzirom na kriterijum razdvajanja elemenata, može se zaključiti da je u ovom koraku pivot došao na svoju konačnu poziciju koju će imati u sortiranom nizu. Problem sortiranja niza je sveden na problem sortiranja dva podniza. Postupak se rekurzivno nastavlja sve dok svaka rezultujuća particija ne dođe do jedinične veličine, čime se dobija sortirani niz.

Primer:

Usvojicemo biranje pivota kao prvog elementa u particiji. U nasem primeru je pivot kljuc 65. Onda uzmemo dva pointera (celobrojna pokazivaca). To su **i (krece odozdo)** i **j (krece odozgo).** Prvi pokazivac, i, prelazi preko svih elemenata koji zasluzuju da ostanu u donjoj particiji. To su svi elementi koji su manji od pivota. Kada dodje do vecega, onda zastane. U nasem slucaju, i je zastao vec na prvom koraku mkod kljuca 75 koji je veci od kljuca 65.

Onda odemo gore gde se pokazivac j spusta na dole i prelazi preko svih elemenata koji zasluzuju da ostanu u gornjoj particiji. On zastane kada naidje na kljuc koji je manji od kljuca 65. U nasem primeru zastaje vec na prvom koraku kod kljuca 0. Kada se ovo desi, kljucevi 75 i 0 zamene mesta.

Pokazivaci nastavljaju da „klize“, i ide na gore, a j na dole. Pokazivac i prelazi preko kljuceva 6 i 57 jer oni zasluzuju da ostanu u donjoj particiji, ali zastaje na kljucu 99 koji je veci od pivota, kluca 65.

Pokazivac j zastaje kada dodje do kljuca 27 koji je manji od pivota pa samim tim ne zasluzuje da bude u gornjoj particiji. Nakon toga kljucevi 27 i 99 zamene mesta (4. red na slici).

Ovo je sada granica izmedju gornje i donje particije. Ocigledno je da donju particiju cine kljucevi: 0, 6, 57, 27. Gornju particiju cine kljucevi: 99, 75, 96.

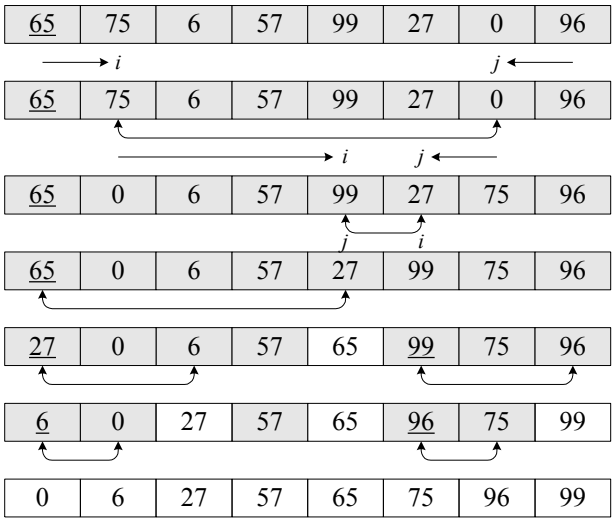
Neophodno je da pivot sada dodje na svoju konacnu particiju. To ce se desiti kada pivot, kljuc 65, zameni mesto sa poslednjim elementom donje particije, kljucem 27 (5. red na slici).

Sada smo sortiranje niza sveli na sortiranje dva podniza. Izmedju njih je pivot 65 koji je na svojoj konacnoj poziciji. Prvi podniz cine: 27, 0, 6, 57. Drugi pod niz cine: 99, 75, 96.

Prvi podniz: 27, 0, 6, 57.

Za pivota uzimamo prvi element, kljuc 27. Pokazivac i krece od dole, od kljuca 0, a pokazivac j krece od gore na dole, od kljuca 57. Pokazivac i se zaustavlja na kljucu 57, a pokazivac j se zaustavlja na kljucu 6. **Oni su se mimoisli, a nije bilo zamene pa sada pivot i j-ti element menjaju mesto**, tj. kljucevi 27 i 6. Kljuc 27 je sada na konacnoj poziciji (6.red na slici).

Posmatramo sada particiju koju cine kljuc 6 koji je sada pivot i kljuc 0. Pokazivac i krece od kljuca 0, a pokazivac j je vec na kljucu 0. **U ovoj situaciji su se pokazivaci susreli pa kako nije bilo zamena, j-ti element i pivot menjaju mesto**, tacnije kljucevi 0 i 6. Sada su kljucevi 0, 6, 27 i 65 na konacnim pozicijama. Sada imamo dve particije. Prvu cini kljuc 42, a drugu cine: 96, 75, 99. U prvoj particiji su i i j na istom kljucu, susreli su se, a nije bilo zamene pa je ocigledno da kljuc 42 ostaje na toj poziciji. Konacni sortiran niz je dat u 7.redu na slici.



Slika 12.13 Particijsko sortiranje

Implementacija:

Osnovni deo implementacije ovog algoritma predstavlja funkcija **PARTITION.** Ova funkcija se sastoji iz tri ciklusa:

1. spoljasnji koji se ponavlja dok se indeksi i i j ne susretnu (i polazi od pocetka particije, a j od kraja)
2. i dva unutrasnja ciklusa, od kojih jedan inkrementira i sve dok je i-ti element manji ili jednak pivotu, a drugi inkrementira j sve dok je j-ti element veci od pivota.

Ovde sada ima par scenarija:

1. i-ti element je veci od pivota, a j-ti element je manji od pivota, a i je manje od j (jos uvek se nisu susreli), tada kljucevi sa indeksima i i j menjaju mesta. Nakon ucinjene zamene, proces konvergiranja i i j se nastavlja sve do susretanja ili mimoilazenja. Kada god se desi da je i-ti element veci od pivota, a j-ti manji, oni zamene mesta
2. kada se i i j susretnu ili mimoidju, pivot i j-ti element zamene mesto. Pivot se sada nalazi na svojoj konacnoj poziciji. Svi elementi levo od njega su manji, a desno od njega veci. Na ovaj nacin smo dobili dve neuredjene particije. Ovim se zavrsava funkcija PARTITION, koja kao rezultat vraca indeks j gde je smesten pivot. Citav proces se ponavlja za donju, a zatim za gornju particiju.
3. i i j su se mimoisli (nije bilo zamena): pivot i j-ti element menjaju mesto
4. i i j su se susreli (nije bilo zamena): pivot i j-ti element menjaju mesto

Performanse:

Najbolji slucaj: za idelno polovljenje particija – $O(n \log n)$

Najbolji slučaj se javlja kada se obrađivana particija u svakom koraku prepolovi na dve jednake particije. Tada se broj koraka polovljenja izračunava iz uslova $n/2^k = 1$, što daje $k = \log n$. U svakom koraku se obradi najvise n elemenata. Prema tome, vremenska složenost u najboljem slučaju idealnog polovljenja particija je reda $O(n \log n)$.

Najgori slucaj: za neravnomerno deljenje particija – $O(n^2)$

To je slucaj kada se za pivota bira najmanji ili najveci element. Ostali elementi predju ispod ili iznad njega, a druge particije nema. Paradoksalno je da bi najgoru performansu dobili ako bi krenuli od skroz uredjenog niza. Neka je to rastuci niz sa pocetnim elementom 0. I sada ako uzmemo da je pivot upravo 0, mi imamo samo jednu particiju jer nema elemenata manjih od 0.

Prosecan slucaj: gori od najboljeg za 38% – $O(n \log n)$

O(*n log n*) se garantuje ako se particije dele u nekom odnosu cak i ako je on izuzetno nepovoljan, npr. 10:1. Vazno je izbeci da se krnji samo jedna particija.

Izbor pivota:

Za efikasnost je izuzetno bitno kako se pivot bira. Izbor pivota određuje podelu particija:

- **ne birati prvi ili poslednji element za pivota**: da bi se izbegla paradoksalna situacija sa uredjenim nizom, nikada se ne bira prvi ili poslednji element
- **slucajan izbor**: slucajan izbor nas ne stiti od najgoreg slucaja
- **srednji od tri (ili vise) elementa particije**: nacin koji nas stiti od najgoreg slucaja je da uzmemo tri elementa particije pa onda srednji od tih elemenata da nam bude razdvojni element, tj. pivot. U tom slucaju se garantuje da ce jedna particija da ima bar jedan element. Nekada se ovaj izbor vrsi uzimanjem 5 elementa sto garantuje jos simetricnije deljenje particije.
- **srednja vrednost (meansort)**: postoji mogucnost da se pivot uzima kao srednji element nekih slucajno izabranih elemenata. Ova varijanta se zove ***meansort***. Kod ovog metoda se pri kreiranju particija izračunava srednja vrednost elemenata u njoj, pa se ona zatim koristi kao pivot. Ovde pivot ne mora da bude član particije, tako da on ne ide na svoje mesto između particija, pa se one "dodiruju".

Pobitno razdvajanje – radix exchange (Zadatak 9, 52. pitanje)

Princip razbijanja na particije se koristi i u metodu ***pobitnog razdvajanja*** (*radix exchange*), ali se umesto korišćenja pivota uvodi drugačiji kriterijum podele na particije. Ovaj metod podrazumeva korišćenje binarne reprezentacije ključa (*radix* = 2) da bi se donosile binarne odluke u podeli na particije.

Neka binarna reprezentacija ključa ima m bitova $(b_{m-1}\ b_{m-2}\ ....b_0)_2$, gde je b_{m-1} najstariji, a b_0 najmlađi bit. U prvom koraku se analizira najstariji bit b_{m-1} i stvaraju se dve particije: gornja u kojoj su svi ključevi sa $b_{m-1} = 1$ i donja u kojoj su svi ključevi sa $b_{m-1} = 0$. Zatim se u obe particije isti postupak ponavlja konsultujući bit b_{m-2}. Postupak se nastavlja do particija jedinične veličine ili dok se ne ispitaju svi bitovi.

Opis algoritma:

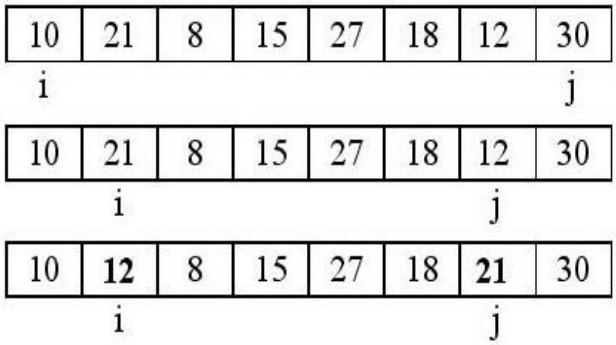
Postupak razdvajanja se vrši veoma slično kao u quicksort-u. Za particiju koja se obrađuje uvedu se dva tekuća indeksa: i koji se na početku postavi na donju granicu particije i j na gornju granicu particije. Zatim se i inkrementira sve dok se ne nađe ključ kod koga je tekući bit odlučivanja 1, a j se dekrementira sve dok se ne nađe ključ kod kojeg je taj isti bit 0, pa ova dva ključa zamene mesta. Ovo se radi sve dok ne postane i veće od j. Kada i postane vece od j, tj. kada se mimoidju, j predstavlja gornju granicu elemenata kod kojih posmatrani bit ima vrednost 0, a i predstavljadonju granicu za one kod kojih on ima vrednost 1. Time su razdvojene dve particije.

Nakon razdvajanja na particije, ista procedura se primenjuje za svaku od particija, s tim sto se sada posmatra naredni bit, manje tezine. Vremenska složenost je reda O(n log n).

Primer:

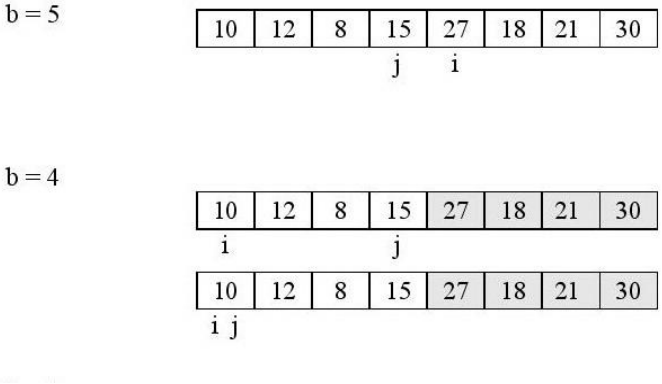
Demonstrirati postupak na primeru neuredjenog niza: **10, 21, 8, 15, 27, 18, 12, 30.**

10	01010
21	10101
8	01000
15	01111
27	11011
18	10010
12	01100
30	11110



U prvom koraku, indeks i se povecava, prelazeci preko kljuceva kod kojih je najstariji bit 0. Zaustavlja se kada naidje na kljuc kod koga je najstariji bit 1. U nasem primeru, to je kljuc 21.

Indeks j se smanjuje, prelazeci preko kljuceva kod kojih je najstariji bit 1, a zaustavlja se kada naidje na kljuc kod koga je najstariji bit 0. U nasem primeru, to je kljuc 12 (3.red slike). Nakon toga kljucevi 21 i 12 zamene mesta, a indeksi i i j nastave da se povecavaju, odnosno smanjuju.



Nakon mimoilaženja **i** i **j**, **j** predstavlja gornju granicu elemenata kod kojih posmatrani bit ima vrednost 0, a **i** predstavlja donju granicu za one kod kojih on ima vrednost 1. Time su razdvojene dve particije.

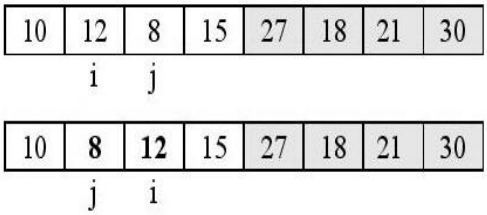
Nakon razdvajanja na particije, ista procedura se primenjuje na svaku od particija, s tim što se tada posmatra naredni bit, manje težine.

U ovom slučaju, bit 4 svih elemenata donje particije ima vrednost 1, pa se ova particija ne redukuje.

Sada treba razmatrati naredni bit, bit 3 donje particije. Indeks i se zaustavlja na kljucu 12 kod koga je ovaj bit 1, a indeks j se zaustavlja na kljucu 8 kod koga je ovaj bit 0. Sada je neophodno da kljucevi 12 i 8 zamene mesta, a indeksi i i j su se mimoisli, pa **j** predstavlja gornju granicu elemenata kod kojih posmatrani bit ima vrednost 0, to su elementi 10 i 8. Indeks **i** predstavlja donju granicu za one kod kojih on ima vrednost 1. To su elementi 12 i 15.

10	01010
21	10101
8	01000
15	01111
27	11011
18	10010
12	01100
30	11110

b = 3

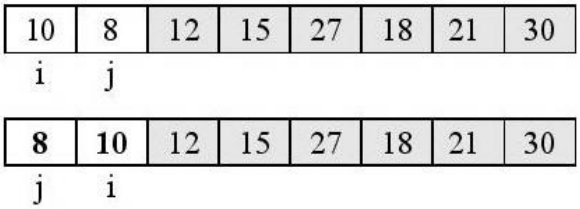


Sada uslovno imamo 3 particije. U prvoj su elementi 10 i 8, a tu analizu nastavljamo od bita b=2. Drugu particiju cine elementi 12 i 15, a tu analizu nastavljamo od bita b=2. Trecu particiju cine elementi 27, 18, 21, 30. Tu analizu nastavljamo od bita b=4.

Particija ciji su elementi 10 i 8:

Posmatramo bit b=2. Indeks i krece od kljuca 10, ali on se vec tu zaustavlja jer je bit 2 kod ovog kljuca 1. Indeks j krece od kljuca 8 i on se vec tu zaustavlja jer je bit 2 kod ovog kljuca 0.

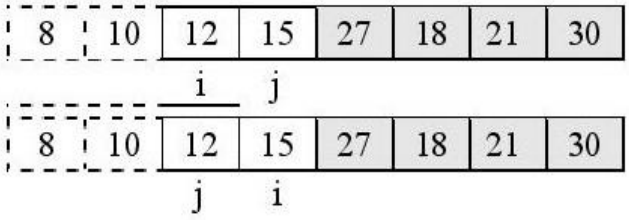
b = 2



Particija ciji su elementi 12 i 15:

Posmatramo bit b=2. Indeks **i** krece od kljuca 12, pa prelazi na kljuc 15 gde staje jer je kraj particije, a i bit 2 kod kljuca 15 je 1. Indeks j krece od kljuca 15, pa prelazi na kljuc 12 gde staje je je kraj particije, a i bit 2 kljuca 12 je 0.

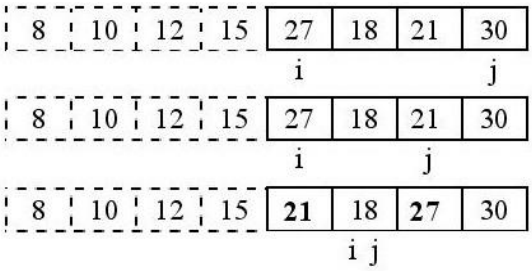
b = 2



Particija ciji su elementi 27, 18, 21, 30:

10	01010
21	10101
8	01000
15	01111
27	11011
18	10010
12	01100
30	11110

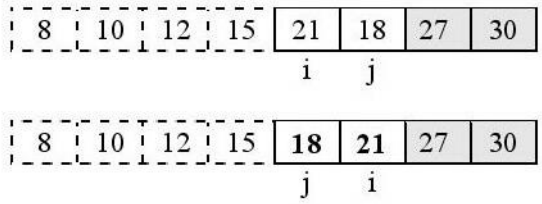
b = 4



Analizu ove particije pocinjemo od bita b=4. **Indeks i** polazi od kljuca 27. Medjutim, bit 4 ovog kljuca ima vrednost 1, pa on tu zastaje. **Indeks j** polazi od kljuca 30 i zastaje na kljucu 21 jer bit 4 ima vrednost 0. Sada je neophodno da kljucevi 21 i 27 zamene mesta, a indeksi se susrecu na kljucu 18. Indeks j tu zastaje jer je bit 4 kljuca 18 nula, a indeks i dolazi do kljuca 27 i tu zastaje jer je bit 4 jednak 1. Dakle, indeksi su se mimoisli sto prakticno znaci da smo sada ovu particiju podelili na dve nove particije. Prvu cine kljucevi 21 i 18, a drugu cine kljucevi 27 i 30. Sada analizu nastavljamo za svaku particiju posebno gledajuci bit 3.

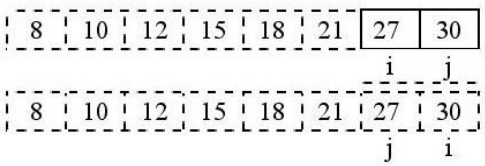
Particija ciji su elementi 21, 18:

b = 3



Bit 3 kljuca 21 je 1, a bit 3 kljuca 18 je 0, pa je neophodno ove kljuceve zameniti. Kljucevi 27 i 30 su vec u pravom poretku.

b = 3



Primetiti da je ovde sortiranje završeno pre nego što su provereni svi bitovi u binarnoj reprezentaciji ključeva.

Metodi spajanja

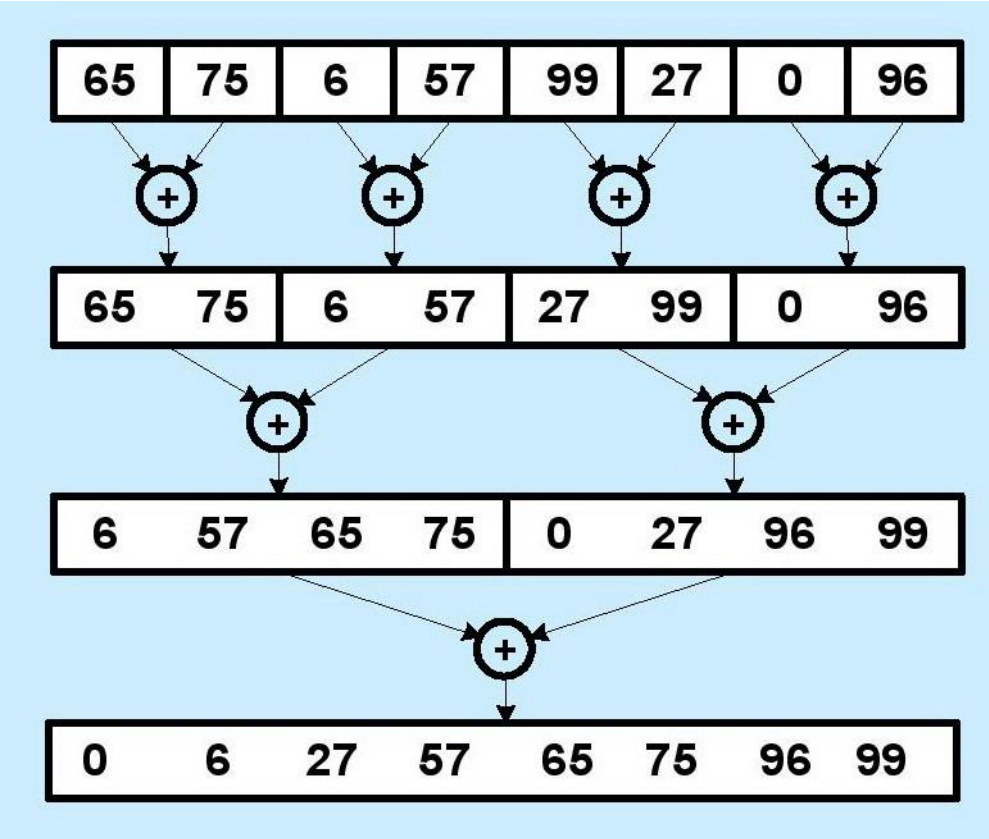
Ove metode su pretezno karakteristicne za spoljasnje sortiranje mada se mogu koristiti i kod unutrasnjeg sortiranja.

Opis algoritma:

U pocetku imamo niz od n elementata koji su sami po sebi uredjeni. To je zapravo n grupa od po jednog elementa. Zatim spajamo po dve takve grupe u jednu. Sada zapravo dva elementa spojimo na uredjen nacin i tako formiramo grupe od po dva elementa.

U narednom koraku grupe od po dva elementa spajamo u grupe od 4 elementa unutar kojih su elementi uredjeni. Pa grupe od 4 u grupe od 8 i tako redom dok ne sortiramo citav niz.

Primer:



Performanse:

Ukupan broj koraka je log n, a u svakom koraku poredimo n elemenata pa je ukupna slozenost **O(n log n).**

Sortiranje brojanjem - counting sort (Zadatak 10, 58. pitanje)

Metodi pod određenim pretpostavkama o vrednostima koje se sortiraju, mogu da postignu linearnu složenost.

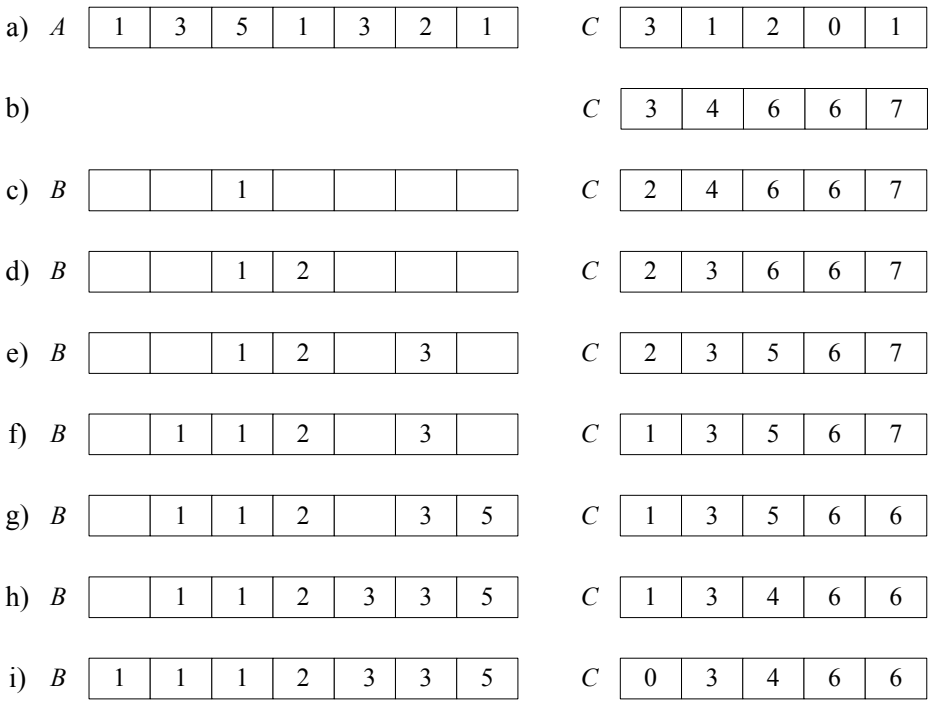
Opis algoritma:

Sortiranje brojanjem (counting sort) ***podrazumeva da su ključevi iz ulaznog niza celobrojne vrednosti u opsegu*** $1 \leq a[i] \leq k$. ***Osnovna ideja ovog metoda je da za svaki element ulaznog niza odredi broj ostalih elemenata iz niza koji su manji od njega. Na primer, ako je 10 elemenata niza manje od nekog elementa, onda on u sortiranom nizu treba da se smesti na poziciju 11. Naravno, osnovna ideja se mora nadgraditi imajući u vidu i mogućnost jednakih ključeva, jer oni ne mogu da se smeste na istu poziciju. Algoritam je stabilan jer zadrzava isti poredak elemenata sa istom vrednoscu nakon sortiranja.***

Performanse

Vremensku složenost ovog metoda nije teško odrediti. Prva i treća petlja su složenosti $O(k)$, dok su druga i četvrta petlja složenosti $O(n)$. Prema tome, ukupna vremenska složenost je $O(n + k)$. Pošto se u praksi ovaj metod koristi kada je $k = O(n)$, onda je i **ukupna složenost $O(n)$,** pa je po tom kriterijumu ovaj metod svrstan u linearne metode.

Primer:



Slika 12.16 Sortiranje brojanjem

U nasem primeru, kljucevi leze u opsegu od 1 do 5. Sada je neophodno implementirati jedan brojacki niz C[1:k] koji ima onoliko elemenata kolika je vrednost najveceg elementa u nizu koji sortiramo. U nasem primeru je to 5. On nam je neophodan da bi mogli da prebrojimo elemente 1, 2, 3, 4, 5.

Posle inicijalizacije brojača *C* na nulu, prođe se ulazni niz i za svaki element inkrementira se odgovarajući brojač *C*[*i*]. Utvrdicemo da nas polazni niz ima tri jedinice, jednu dvojku, dve trojke, nijednu cetvrku, jednu peticu.

Treća petlja sukcesivno od početka niza *C* uvećava vrednost tekućeg brojača prethodnim, tako da brojač *C*[*i*] tada označava broj elemenata koji su manji ili jednaki vrednosti *i* (slika 12.16b). **Ova slika nam pokazuje:**

C[1] = 3: da imamo 3 elementa koji su manji ili jednaki 1. To su tri jedinice.

C[2] = 4: da imamo 4 elemetna koji su manji ili jednaki 2. To su tri jedinice i jedna dvojka.

C[3] = 6: da imamo 6 elemenata koji su manji ili jednaki 3. To su tri jedinice, jedna dvojka i dve trojke.

C[4] = 6: da imamo 6 elemenata koji su manji ili jednaki 4. To su tri jedinice, jedna dvojka, dve trojke, nijednu cetvorku.

C[5] = 7: da imamo 7 elemenata koji su manji ili jednaki 5. To su tri jedinice, jedna dvojka, dve trojke, nijednu cetvorku, jedna petica.

Kada smo izvrsili odbrojavanje, elemente mozemo da stavimo na svoja mesta. Krecemo odpozadi pocetnog niza: 1, 3, 5, 1, 3, 2, 1:

1: jedinicu treba smestiti na poziciju 3, a brojac C[1] se umanjiti na 2.

2: dvojku treba smestiti na poziciju na koju pokazuje brojac C[2], a to je pozicija 4. Sada je neophodno brojac C[2] umanjiti na 3.

3: trojku treba smestiti na poziciju na koju pokazuje brojac C[3], a to je pozicija 6. Sada je neophodno brojac C[3] umanjiti na 5.

1: jedinicu treba smestiti na poziciju na koju pokazuje brojac C[1], a to je pozicija 2. Sada je neophodno brojac C[1] umanjiti na 1.

5: peticu treba smestiti na poziciju na koju pokazuje brojac C[5], a to je pozicija 7. Sada je neophodno brojac C[5] umanjiti na 6.

3: trojku treba smestiti na poziciju na koju pokazuje brojac C[3], a to je pozicija 5. Sada je neophodno brojac C[3] umanjiti na 4.

1: jedinicu treba smestiti na poziciju na koju pokazuje brojac C[1], a to je pozicija 1.