PT&SCHE

Introduction of part-time and short cycle studies in Serbia

Funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UNIVERZITET METROPOLITAN BEOGRAD

KI201 - JAVA 4: STRUKTURE PODATAKA I ALGORITMI – DEO A

Sortiranje

Lekcija 06

PRIRUČNIK ZA STUDENTE

KI201 - JAVA 4: STRUKTURE PODATAKA I ALGORITMI – DEO A

Lekcija 06

SORTIRANJE

- Sortiranje
- Poglavlje 1: Uvod u sortiranje
- Poglavlje 2: Sortiranje mehurom – bubble sort
- Poglavlje 3: Sortiranje spajanjem – merge sort
- Poglavlje 4: Brzo sortiranje – quick sort
- Poglavlje 5: Sortiranje pomoću hipa – heap sort
- Poglavlje 6: Segmentno (bucket) i radiks (radix) sortiranje
- Poglavlje 7: Spoljašnje sortiranje – extern sort
- Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

Uvod

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

U okviru ove lekcije studenti se upoznaju sa:

- Osnovima sortiranja
- Najprostijim kvadratnim algoritmima sortiranja (selection sort, bubble sort)
- Metodama sortiranja baziranim na umetanju elemenata (insertion sort, shell sort)
- Metodama sortiranja logaritamske složenosti (merge sort, quick sort i heap sort)
- Metodama segmentnog sortiranja (bucket sort i radix sort)
- Spoljašnjim sortiranjem (extern sort)

Sortiranje je proces preuređivanja elemenata nekog skupa u određenom poretku. Sortiranje skupa podataka je preduslov za njegovo efikasno pretraživanje. Podaci koje treba sortirati mogu biti celi brojevi, brojevi dvostruke dužine, karakteri ili objekti.

Podela algoritama se vrši na osnovu globalnog metoda sortiranja i čine je dve grupe:

- algoritmi sortiranja poređenjem
- algoritmi sortiranja bez poređenja.

Sortiranje je od velikog značaja za izradu velikog broja aplikacija. Izrada i primena najboljeg algoritma u velikoj meri utiče na brzinu rada aplikacija koje operišu sa ogromnim brojem podataka. Stoga ćemo se u okviru ove lekcije baviti raznim metodama sortiranja i anlizirati njihove prednosti i mane, a biće date i preporuke koji je algoritam sortiranja najpogodniji za određeni vid problema.

Poglavlje 1

Uvod u sortiranje

UVODNA RAZMATRANJA

Sortiranje je proces preuređivanja elemenata nekog skupa u određenom poretku. Sortiranje skupa podataka je preduslov za njegovo efikasno pretraživanje

Sortiranje je proces preuređenja elemenata nekog skupa u određeni poredak. Najčešće se koriste numerički i leksikografski poredak. Sortiranje skupa podataka je preduslov za njegovo efikasno pretraživanje. Sortiranje je takođe vrlo korisno pri utvrđivanju jednakosti dva skupa podataka. Za izlazni skup podataka, tj. sortirani skup važi da predstavlja permutaciju ulaznog skupa i da je svaki element izlaznog skupa manji ili jednak prethodnom elementu po odabranom poretku.

Procenjuje se da se 25 ~ 50 % ukupne računarske snage koristi za operacije sortiranja. Mogući razlozi:

- mnoge aplikacije zahtevaju sortiranje
- mnoge aplikacije izvršavaju sortiranje i kada ne moraju to da rade
- mnoge aplikacije koriste neefikasne algoritme sortiranja.

Sortiranje je klasičan predmet u računarstvu. **Prvo**, sortiranje se koristi za rešavanje drugih problema. **Drugo**, algoritmi za sortiranje su dobri za vežbanje osnovnih tehnika programiranja uz korišćenje izjava, petlji, metoda i nizova. **Treće**, algoritmi sortiranja su odlični primeri za poređenje performansi algoritama.

Podaci koje treba sortirati mogu biti: celi brojevi, brojevi dvostruke dužine, karakteri ili objekti. U nastavku lekcije biće prikazane razne metode sortiranja numeričkih vrednosti.

Kod metoda ćemo usvojiti sledeće pretpostavke:

- podaci za sortiranje su celi brojevi
- podaci se sortiraju u rastućem i opadajućem redosledu.
- podaci se čuvaju u nizovima ili povezanim listama (mogu se koristiti i *ArrayList* ili *LinkedList* za sortiranje).

KLASIFIKACIJA ALGORITAMA

Dele se na algoritme sortiranja poređenjem (poređenje elemenata prema nekom ključu) i na one bez poređenja. Algoritmi sortiranja bez poređenja mogu dostići linearnu složenost

Podela algoritama se vrši na osnovu globalnog metoda sortiranja i čine je dve grupe:

- Algoritmi sortiranja poređenjem
- Algoritmi sortiranja bez poređenja.

Algoritmi sortiranja poređenjem

Kod ove grupe algoritama sortiranja, elementi ulaznog skupa se prerasporeduju na osnovu nekog ključa. Ključ predstavlja posebno izabranu vrednost koja određuje da li je neki element veći, manji ili jednak nekom drugom elementu ulaznog skupa. Najveći broj algoritama pripada ovoj grupi, a oni se dalje mogu klasifikovati na sledeći način:

a) Algoritmi umetanja:

- sortiranje umetanjem (insertion sort)
- metoda sortiranja po Šelu (Shell sort).

b) Algoritmi izbora:

- sortiranje selekcijom (ili izborom – selection sort)
- sortiranje korišćenjem hipa (heap sort).

c) Algoritmi zamene:

- sortiranje mehurom (bubble sort).
- brzo sortiranje (quick sort)

Algoritmi sortiranja bez poređenja

Ovi algoritmi koriste druge metode osim poređenja ključeva elemenata da bi došli do željenog redosleda elemenata. Osnovni razlog za razvoj ovih algoritama je što algoritmi sortiranja poređenjem u najboljem slučaju imaju složenost O (n log n), dok algoritmi sortiranja bez poređenja mogu dostići linearnu složenost.

Najznačajniji algoritmi iz ove grupe su:

- bucket sort (segmentno sortiranje)
- counting sort (sortiranje brojanjem)
- radix sort .

U okviru ove lekcije od algoritama bez poređenja će biti obrađeni bucket sort i radix sort algoritmi. Sortiranje brojanjem će biti obrađeno u okviru lekcije o algoritmima za rad sa stringovima.

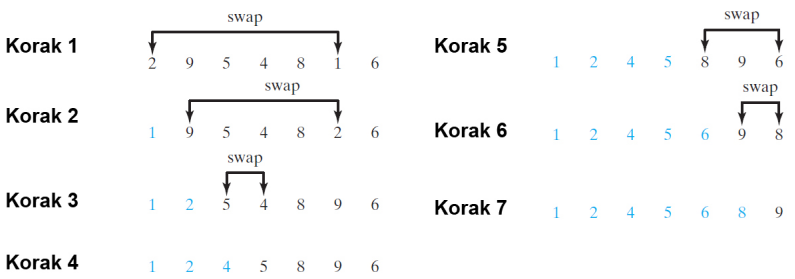
SORTIRANJE SELEKCIJOM

Metoda selekcije pronalazi najmanji element u listi i zamenjuje ga sa prvim elementom. Zatim se u ostatku liste ponovo pronalazi najmanji element i zamenjuje sa drugim članom liste

Sortiranje, kao i pretraživanje, je veoma čest zadatak u programiranju. Mnogo različitih algoritama je razvijeno u svrhu sortiranja. U nastavku ćemo se upoznati sa jednim veoma prostim i intuitivnim algoritmom a to je metoda selekcije, odnosno selection sort.

Metoda selekcije - Selection Sort

Pretpostavimo da želimo da sortiramo listu brojeva u rastućem poretku. Metoda selekcije pronalazi najmanji element u listi i zamenjuje ga sa prvim elementom. Zatim se u ostatku liste, bez prvog člana, ponovo pronalazi najmanji element i zamenjuje sa drugim članom liste. Postupak se ponavlja sve dok se prođe kroz sve pozicije u listi (Slika- 1) .



Slika 1.1 Postupak sortiranja metodom selekcije

Koraci u izvršavanju algoritma su:

Korak 1: Izabrati 1 (najmanji) i zameniti ga sa 2 (prvi) u listi.

Korak 2: Broj 1 je sada na pravilnoj poziciji i ne treba više ništa razmatrati. Izabrati broj 2 (najmanji) i zameniti mesta sa brojem 9 (prvi) u ostatku liste.

Korak 3: Broj 2 je sad postavljen na korektno mesto, i završeno je razmatranje za ovu poziciju. Izabrati 4 (najmanji) i zameniti mu mesta sa 5 (prvi) u ostatku liste.

Korak 4: Broj 4 je na pravoj poziciji i njega više ne poredimo sa drugim brojevima. Proveravamo broj 5, ali je on najmanji među elementima u ostatku. Nema nikakve zamene u ovom slučaju.

Korak 5: Broj 5 je na korektnoj poziciji, prelazimo stoga na sledeću poziciju. Izaberemo broj 6 (najmanji) i zamenimo mu mesta sa brojem 8 (prvi) u ostatku liste.

Korak 6: Broj 6 je sada na korektnoj poziciji i više ga ne uzimamo u razmatranje. Zatim izaberemo broj 8 (najmanji) i zamenimo mu mesta sa 9 (prvi) u ostatku liste.

IMPLEMENTACIJA METODE SELEKCIJE – SELECTION SORT

Metod selekcije je najjednostavniji, ali i najsporiji metod sortiranja nizova. Sastoji se iz dve ugneždene petlje koje se maksimalno izvršavaju n puta, gde je n dimenzija niza

Sada kada znamo kako funkcioniše *selection sort* možemo da ga implementiramo u nekom programskom jeziku (C ili Java na primer). Početnicima u programiranju je teško da iz prvog pokušaja razviju kompletno rešenje problema. Stoga, krenućemo sa pisanjem koda prve iteracije sortiranja sa ciljem da se pronađe najmanji element i da se zameni sa prvim elementom, a zatim ćemo razmisliti šta će biti različito za drugu iteracij, šta za treću itd. Uvid u svaki pojedinačni korak će vam omogućiti da generalizujete algoritam i da napišete opšti algoritam za bilo koju iteraciju sortiranja. Rešenje može biti predstavljeno na sledeći način:

```
for (int i = 0; i < list.length - 1; i++)
{
 select the smallest element in list[i..list.length-1];
 swap the smallest with list[i], if necessary;
 // list[i] is in its correct position.
 // The next iteration apply on list[i+1..list.length-1]
}
```

U nastavku je listing kompletnog programa u Java-i. Tu imamo metod *selectionSort(double[] list)* koji sortira bilo koji niz realnih brojeva tipa ***double***. Metod je implementiran korišćenjem ugneždene *for* petlje.

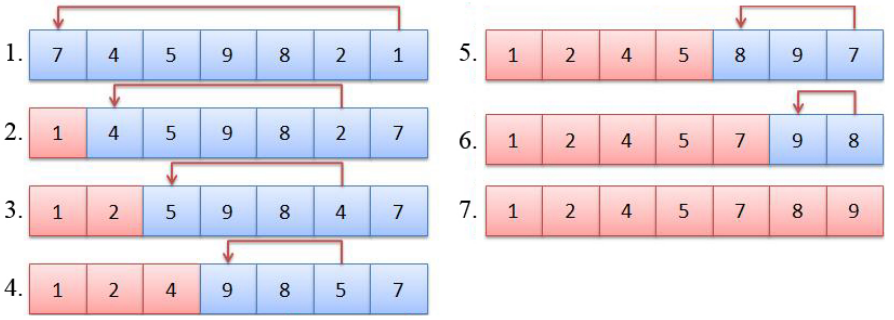
```
public class SelectionSort {
  public static void selectionSort(double[] list) {
    for (int i = 0; i < list.length - 1; i++) {
    // Find the minimum in the list[i..list.length-1]
    double currentMin = list[i];
    int currentMinIndex = i;

    for (int j = i + 1; j < list.length; j++) {
        if (currentMin > list[j]) {
        currentMin = list[j];
        currentMinIndex = j;
        }
    }

    if (currentMinIndex != i) {
        list[currentMinIndex] = list[i];
        list[i] = currentMin;
    }
    }
  }
}
```

Spoljašnja petlja (brojač *i*) iteriše sa ciljem da se pronađe najmanji element u listi koja obuhvata elemente od *list[i]* do*list[list.length-1]*, a zatim da taj element zameni sa članom *list[i]*. Promenljiva ***i*** je inicijalno jednaka *0*. Nakon svake iteracije spoljašnje petlje, mi postavljamo odgovarajući član na pravo mesto, a to je mesto *list[i].* Konačno, svi elementi se postavljaju na prava mesta i dobijamo sortiranu listu.

ANALIZA SELECTION SORT ALGORITMA

Pošto se sortiranje selekcijom sastoji iz dve ugnežđene petlje koje se u najgorem slučaju izvršavaju n-1 puta, može se zaključiti da se radi o algoritmu kvadratne složenosti

Algoritam sortiranja selekcijom je opisan u prethodnoj lekciji, i on pronalazi najmanji element u listi, a zatim menja mesta najmanjem i prvom članu liste. Zatim u ostatku liste (bez prvog člana, koji je sada najmanji) ponovo pronalazi najmanji i menja mu mesta sa prvim članom te preostale liste, i tako dalje dok sa desne strane ne ostane lista sa samo jednim elementom u njoj (Slika-2).



Slika 1.2 Postupak sortiranja selekcijom (selection sort algoritam)

Broj poređenja je *n – 1* za prvu iteraciju, *n-2* za drugu, i tako dalje. Označimo sa *T(n)* vremensku složenost *selection sort* algoritma a sa *c* ukupan broj ostalih operacija dodele i dodatnog poređenja u svakoj iteraciji. U tom slučaju imamo (Slika-3):

$$
\begin{aligned}
T(n) &= (n-1) + c + (n-2) + c + \ldots + 2 + c + 1 + c \\
&= \frac{(n-1)(n-1+1)}{2} + c(n-1) = \frac{n^2}{2} - \frac{n}{2} + cn - c \\
&= O(n^2)
\end{aligned}
$$

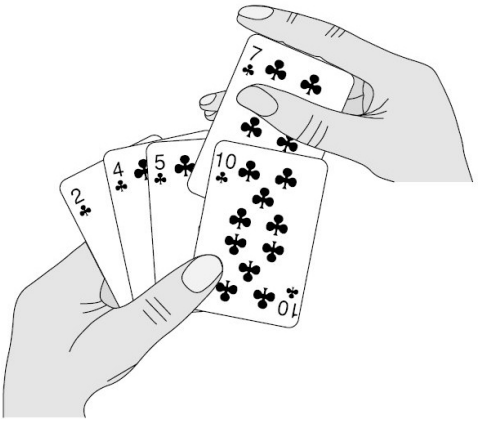
Slika 1.3 Vremenska složenost selection sort algoritma

Stoga, ako ostavimo samo najdominantniji član polinoma, složenost *selection sort* algoritma je $O(n^2)$.

IDEJA O SORTIRANJU UMETANJEM

U okviru metoda sortiranja umetanjem dve najpoznatije metode su: metoda direktnog umetanja (insertion sort) i metoda sortiranja po Šelu (Shell sort)

Jedan od primera ovakve grupe sortiranja je metod baziran na igri karata gde igrač pokušava da sortira karte u ruci kada uzima novu kartu i pokušava da stavi kartu na pravo mesto (Slika-4) .



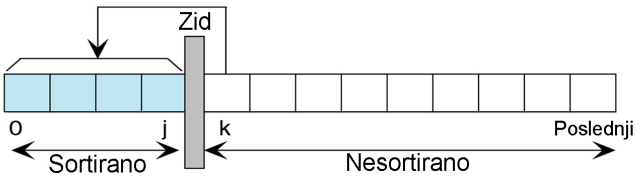
Slika 1.4 Ideja o umetanju elemenata u niz na određeno mesto

Ako je dat niz x_n sa elementima nekog uređenog tipa T, koji treba urediti u neopadajući poredak, ova metoda sortiranja polazi od pretpostavke da imamo uređen početni deo niza, (to svakako važi za $i = 2$, jer je podniz sa jednim elementom uređen) i u svakom koraku, počevši od $i = 2$ i povećanjem i, i-ti element se stavlja na pravo mesto u odnosu na prvih (uređenih) i-1.

Sortiranje umetanjem je jednostavan algoritam za sortiranje koji generiše krajnji sortirani niz stavku po stavku.

Mnogo je manje efikasan sa većim listama od mnogo složenijih algoritama kao što su algoritam brzog sortiranja (quicksort), hipsort i algoritam sortiranja spajanja (mergesort).

Koristi dva skupa podataka za sortiranje: sortirane i nesortirane (Slika-5). U svakom prolazu se prvi element u nesortiranoj podlisti prenosi u sortiranu podlistu umetanjem na odgovarajuće mesto.



Slika 1.5 Postupak kod sortiranja direktnim umetanjem

Studenti mogu pogledati animaciju sortiranja umetanjem na sledećem web linku: http://www.cs.armstrong.edu/liang/animation/web/InsertionSort.html. U okviru materijala za

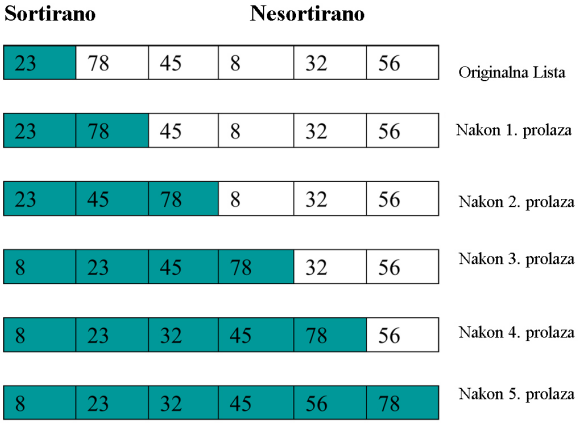
vežbe se nalazi primer u Java-i koji kreira jednu ovakvu animaciju, što će i biti obrađeno na vežbama.

PRIMERI SORTIRANJA DIREKTNIM UMETANJEM

U svakom prolazu sortiranja direktnim umetanjem, prvi element u nesortiranoj podlisti se prenosi u sortiranu podlistu umetanjem na odgovarajuće mesto

Primer 1

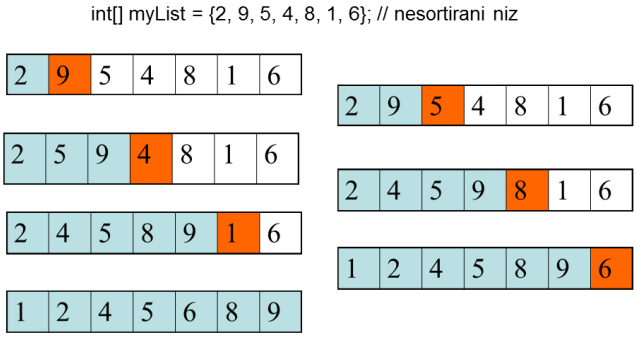
Slika- 6 daje primer sortiranja direktnim umetanjem.



Slika 1.6 Sortiranje direktnim umetanjem. Postupak prebacivanja elementa direktnim umetanjem iz nesortiranog u sortirani deo

Primer 2

U nastavku je još jedan primer sortiranja umetanjem (Slika- 7) .



Slika 1.7 Sortiranje direktnim umetanjem. Postupak pomeranja zida koji razdvaja sortirani od nesortiranog dela

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

IMPLEMENTACIJA I ANALIZA ALGORITMA

Algoritam se sastoji iz dve ugnježdene petlje. Spoljašnja petlja ide po elementima nesortirane grupe elemenata, a unutrašnja ubacuje taj element na odgovarajuće mesto u sortiranom delu

Kod za deirektno umetanje može biti napisan na sledeći način:

```
public static void insertionSort(Item a[], int n)
{
 for (int i = 1; i < n; i++) {
  Item tmp = a[i];
  for (int j=i; j>0 &amp;amp;&amp;amp; tmp < a[j-1]; j--)
   a[j] = a[j-1];
  a[j] = tmp;
 }
}
```

U okviru spoljašnje petlje se vrši ubacivanje svakog od *n* elemenata. U proseku, postoji *n/2* elemenata koji su već sortirani. Unutrašnja petlja pomera pola od ovih elemenata, što znači da ih ima *n/4*. Zbog toga je vreme potrebno za sortiranje oko $n^2/4$. Zanemarivanjem konstanti dobija se $O(n^2)$.

Najbolji slučaj: > O(n)

- niz je već sortiran u rastućem redosledu
- unutrašnja petlja se neće izvršiti
- broj pomeranja: 2*(n-1) > O(n)
- broj poređenja ključeva: (n-1) > O(n)

Najgori slučaj: > O(n2)

- niz je inverzno uređen
- unutrašnja petlja se izvršava i-1 puta, za i = 2,3, ..., n
- broj pomeranja: 2*(n-1)+(1+2+...+n-1)= 2*(n-1)+ n*(n-1)/2 > O(n2)
- broj poređenja ključeva: (1+2+...+n-1)= n*(n-1)/2 > O(n2)

Prosečan slučaj: > $O(n^2)$

Sortiranje umetanjem je stoga složenosti $O(n^2)$.

Prednosti algoritma umetanja:

- Jednostavna primena
- Algoritam efikasan na malim skupovima podataka
- Prilagodljiv za skupove podataka koji su već značajno sortirani
- Efikasniji u praksi od većine drugih kvadratnih, tj. $O(n^2)$ algoritama, kao što su sortiranje izborom (selection sort) ili algoritam sortiranja mehurom (bubble sort)
- Trenutan (online, može da sortira listu, odmah pri primanju).

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Na osnovu Slike 1 primeniti sortiranje selekcijom na sledećim brojevima {3.4, 5, 3, 3.5, 2.2, 1.9, 2}.

Zadatak 2. Na osnovu Slike 5 primeniti sortiranje umetanjem na sledeće brojeve {3.4, 5, 3, 3.5, 2.2, 1.9, 2}.

Zadatak 3. Kako se može algoritam sortiranja selekcijom modifikovati tako da se sortiranje vrši u opadajućem redosledu?

Zadatak 4. Kako se može algoritam sortiranja umetanjem modifikovati tako da se sortiranje vrši u opadajućem redosledu?

Zadatak 5. Sortiranje selekcijom uzima najmanji element niza i zamenjuje ga sa prvim. Izmeniti implementaciju algoritma tako da algoritam zamenjuje najveći element sa poslednjim. Napisati test program koji će učitati 10 brojeva, pokrenuti izmenjeni algoritam i ispisati rezultat.

Zadatak 6. Implementirati sortiranje umetanjem generičkog niza brojeva.

```
public static <E extends Comparable<E>>
    void insertionSort(E[] list)
```

Zadatak 7. Implementirati sortiranje umetanjem primenom rekurzije.

Poglavlje 2

Sortiranje mehurom – bubble sort

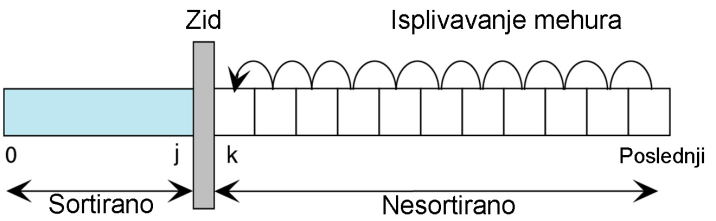
OSNOVI O SORTIRANJU MEHUROM

Sortiranje mehurom ispituje niz od početka do kraja, upoređujući elemente u toku prolaza. Cilj je da se u toku procesa poređenja vrši isplivavanje najlakšeg mehura na jednu stranu niza

Sortiranje mehurom ispituje niz od početka do kraja upoređujući elemente u toku prolaza. Svaki put kada naiđe veći element pre manjeg elementa, zamenjuju se ova dva elementa. Na ovaj način se veći elementi prenose prema kraju niza. Najveći (ili najmanji) element *"ispliva"* na jedan kraj niza. Onda se ponavlja proces za nesortirani deo niza sve dok se čitav niz ne sortira.

Primer sortiranja mehurom

- Lista se deli na dva dela: sortirani i nesortirani deo (Slika- 1) .
- Najmanji element ispliva iz nesortiranog dela u sortirani deo.
- Zid se pomera za jedan element udesno.
- Sortiranje zahteva *n-1* prolaza za sortiranje niza.



Slika 2.1 Postupak sortiranja mehurom

Osnovna analiza algoritma

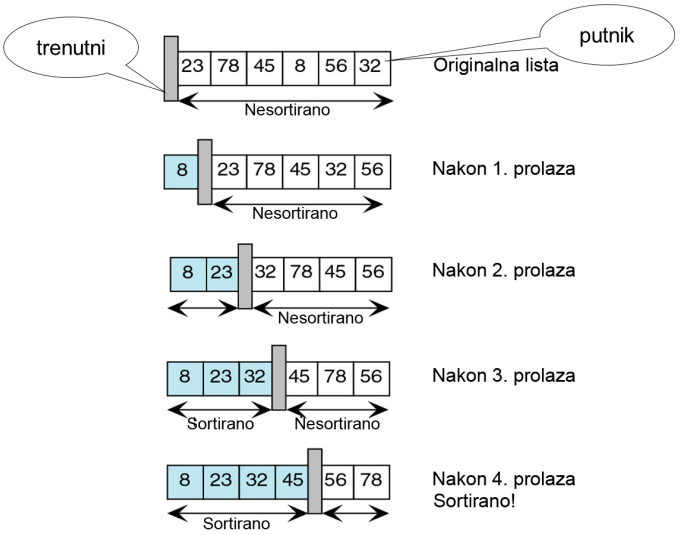
Prosečno i najgore vreme izvršavanja ovog algoritma je $O(n^2)$, pa se veoma retko koristi za velike i neuređene nizove. Sortiranje mehurom može biti korišćeno za mali skup podataka. On može biti efikasan i za nizove koji su skoro ili delimično sortirani. Na primer, ako je neki broj elemenata samo za jednu poziciju pomeren u odnosu na poziciju koju bi imao nakon sortiranja (npr. 0123546789 i 1032547698), zamena kod sortiranja mehurom će već u prvom prolazu da ih dovede do odgovarajuće pozicije, dok u drugom prolazu će urediti sve elemente.

PRIMERI SORTIRANJA MEHUROM

U toku jedne faze sortiranja mehurom, jedan mehur isplivava sve dok ne naiđe na lakši od sebe i tu staje, a dalje se nastavlja isplivavanje lakšeg mehura do samog vrha ili do prvog lakšeg

Primer 1

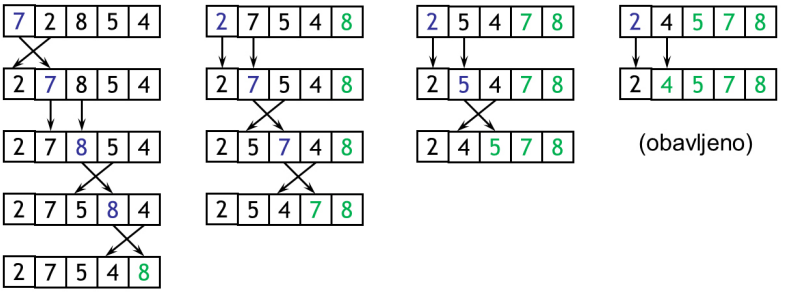
Slika- 2. daje primer sortiranja mehurom.



Slika 2.2 Primer sortiranja mehurom. Pomeranje zida koje razdvaja sortirani i nesortirani deo

Primer 2

Pogledajmo još jedan primer (Slika- 3) :



Slika 2.3 Sortiranje mehurom. Postupak zamene elemenata na odgovarajućim pozicijama, i pomeranje zida između sortiranog i nesortiranog dela niza

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

IMPLEMENTACIJA I ANALIZA ALGORITMA

Svaki put kada naiđe veći element pre manjeg elementa, zamenjuju se ova dva elementa. Najlakši ispliva, a proces se ponavlja za nesortirani deo niza sve dok se čitav niz ne sortira

U nastavku je dat kod u programskom jeziku Java:

```
public static void bubbleSort(int[] a)
{
 int outer, inner;
 for (outer = a.length - 1; outer > 0; outer--)
 {
  //counting down
  for (inner = 0; inner < outer; inner++)
  {
   // bubbling up
   if (a[inner] > a[inner + 1])
   { // if out of order...
    int temp = a[inner];
    // ...then swap
    a[inner] = a[inner + 1];
    a[inner + 1] = temp;
   }
  }
 }
}
```

Neka je veličina niza *n = a.length*. Spoljašnja petlja se izvršava *n-1* puta (može se reći *n*, dovoljno tačno). Svaki put kada se izvrši spoljašnja petlja (outer loop), unutrašnja petlja (inner loop) se takođe izvrši. Unutrašnja petlja se izvrši *n-1* puta u prvom koraku spoljašnje petlje, linearno opadajući do izvršavanja samo jednom u poslednjem koraku.

U proseku, unutrašnja petlja se izvršava oko *n/2* puta za svako izvršavanje spoljašnje petlje. U unutrašnjoj petlji, uvek se izvršavaju poređenje (konstantno vreme) i zamena (swap) (takođe konstatno vreme).

Rezultat je: n * n/2 * k, ili, $O(n^2/2+k)=O(n^2)$

Najbolji slučaj: > O(n)

- niz je već sortiran u rastućem redosledu.
- broj pomeranja: 0 > O(1)
- broj poređenja ključeva: (n-1) > O(n)

Najgori slučaj: > $O(n^2)$

- niz je u reverznom redosledu
- spoljašnja petlja se izvršava n-1 puta
- broj poređenja: (1+2+...+n-1)= n*(n-1)/2 > $O(n^2)$

Prosečan slučaj: > O(n^2)

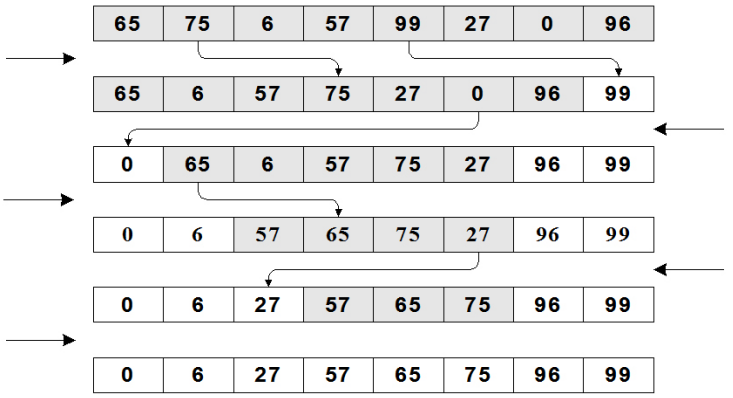
- pogledati sve moguće početne vrednosti podataka

Dakle, sortiranje mehurom je algoritam vremenske složenosti O(n^2) .

SHAKER SORT – POBOLJŠANI BUBBLE SORT

Ovaj algoritam je u stvari verzija sortiranja mehurom, pri čemu se isplivavanje najlakšeg i potanjanje najtežeg mehura obavlja naizmenično u okviru jedne faze

Shaker sort (cocktail sort, shake sort) je stabilan algoritam za sortiranje kvadratne složenosti O(n^2). To je u stvari verzija sortiranja mehurom, pri čemu se isplivavanje najlakšeg mehura i potanjanje najtežeg naizmenično obavlja u okviru jedne faze. Stoga, sortiranje *n* elemenata se obavlja u *n/2* faza. Svaka iteracija se sastoji iz dve faze: isplivavanje najmanjeg mehura *"sa leva na desno"* kome prethodi potanjanje najvećeg mehura *"s desna na levo"*. Pogledajmo sliku (Slika-4):



Slika 2.4 Shaker sort – specijalan slučaj sortiranja mehurom

Na ovaj način se otklanja mana sortiranja mehurom, tj. takozvani problem *zeca i kornjače*.

Problem je naime u tome jer se kod sortiranja mehurom laki mehuri (zečevi) veoma brzo uzdižu na vrh, a teški mehuri (kornjače) se spuštaju samo za jednu poziciju na dole u svakoj iteraciji.

Kod u programskom jeziku Java može biti napisan kao:

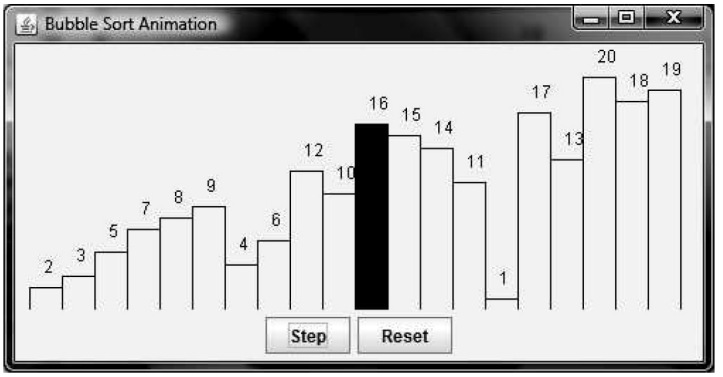
```
public static void shakerSort(int[] array) {
    for (int i = 0; i < array.length/2; i++) {
        boolean swapped = false;
        for (int j = i; j < array.length - i - 1; j++) {
            if (array[j] < array[j+1]) {
                int tmp = array[j];
                array[j] = array[j+1];
```

```
                array[j+1] = tmp;
                swapped = true;
            }
        }
        for (int j = array.length - 2 - i; j > i; j--) {
            if (array[j] > array[j-1]) {
                int tmp = array[j];
                array[j] = array[j-1];
                array[j-1] = tmp;
                swapped = true;
            }
        }
        if(!swapped) break;
    }
}
```

ZADATAK: ANIMACIJA BUBBLE SORT-A

Cilj zadatka je kreiranje programa koji vrši animaciju postupka sortiranja niza primenom metode sortiranja mehurom

Napisati Java aplet koji animira bubble sort. Kreirati niz koji se sastoji od 20 brojeva, nasumično (random) raspoređenih u opsegu od 1 do 20. Elementi niza su prikazani u histogramu kao na Slici- 1. Klikom na dugme *Step* se ostvaruje izvršavanje jedne operacije poređenja u algoritmu i vrši se prefarbavanje histograma za tekući korak sortiranja. Obojiti dirku histograma koja se odnosi na element koji se trenutno zamenjuje. Kada se algoritam završi, prikazati dijalog koji informiše korisnika o kraju sortiranja. Klikom na dugme *Reset* kreira se novi nasumični niz i započinje njegovo sortiranje.



Slika 2.5 Animacija bubble sort algoritma

Rešenje je dato u nastavku:

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
```

```
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class Exercise23_16 extends Application {
  double radius = 2;
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    HistogramPane pane = new HistogramPane();
    pane.setStyle("-fx-border-color: black");

    Button btStep = new Button("Step");
    Button btReset = new Button("Reset");

    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(btStep, btReset);
    hBox.setAlignment(Pos.CENTER);

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(pane);
    borderPane.setBottom(hBox);

    Label lblStatus = new Label();
    borderPane.setTop(lblStatus);
    BorderPane.setAlignment(lblStatus, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 400, 250);
    primaryStage.setTitle("Exercise23_16"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    StepControl control = new StepControl();
    pane.setNumbers(control.getArray());

    pane.widthProperty().addListener(ov -> pane.repaint());
    pane.heightProperty().addListener(ov -> pane.repaint());

    btStep.setOnAction(e -> {
      if (control.step())
        pane.setColoredBarIndex(control.getCurrentIndex());
      else
        lblStatus.setText("The array is already sorted");
    });

    btReset.setOnAction(e -> {
      control.reset();
      pane.setColoredBarIndex(-1);
      lblStatus.setText("");
```

```
    });
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
   */
  public static void main(String[] args) {
    launch(args);
  }

  public final static int ARRAY_SIZE = 20;

  class StepControl {
    private int[] list = new int[ARRAY_SIZE];

    public int[] getArray() {
      return list;
    }

    StepControl() {
      initializeNumbers();
    }

    public void initializeNumbers() {
      for (int i = 0; i < list.length; i++) {
        list[i] = i + 1;
      }

      // Random shuffle
      for (int i = 0; i < list.length; i++) {
        int index = (int) (Math.random() * ARRAY_SIZE);
        int temp = list[i];
        list[i] = list[index];
        list[index] = temp;
      }
    }

    private int i = 1;
    private int j = 0;

    public int getCurrentIndex() {
      return j;
    }

    public void reset() {
      i = 1;
      j = 0;
      initializeNumbers();
    }

    public boolean step() {
```

```
      if (i >= list.length)
        return false;

      if (j < list.length - i) {
        if (list[j] > list[j + 1]) {
          // Swap list[i] with list[i + 1]
          int temp = list[j];
          list[j] = list[j + 1];
          list[j + 1] = temp;
        }
        j++;
      }
      else {
        i++;
        j = 0;
      }

      return true;
    }
  }

  class HistogramPane extends Pane {
    private int[] numbers;
    private int coloredBarIndex = -1;

    public void setNumbers(int[] numbers) {
      this.numbers = numbers;
      repaint();
    }

    public void setColoredBarIndex(int index) {
      coloredBarIndex = index;
      repaint();
    }

    public void repaint() {
      // Find maximum integer
      int max = numbers[0];
      for (int i = 1; i < numbers.length; i++) {
        if (max < numbers[i]) {
          max = numbers[i];
        }
      }

      this.getChildren().clear();

      double height = getHeight() * 0.88;
      double width = getWidth() - 20;
      double unitWidth = width * 1.0 / numbers.length;
      for (int i = 0; i < numbers.length; i++) {
        Rectangle bar =  new Rectangle(i * unitWidth + 10, getHeight()
          - (numbers[i] * 1.0 / max) * height, unitWidth, (numbers[i] * 1.0 / max)
* height);
```

```
        bar.setFill(Color.WHITE);
        bar.setStroke(Color.BLACK);
        this.getChildren().add(bar);
        this.getChildren().add(new Text(i * unitWidth + 10 + 10,
          getHeight() - (numbers[i] * 1.0 / max) * height - 10,
          numbers[i] + ""));
      }

      if (coloredBarIndex >= 0) {
        int i = coloredBarIndex;
        Rectangle filledRectangle = new Rectangle(i * unitWidth + 10, getHeight()
          - (numbers[i] * 1.0 / max) * height, unitWidth, (numbers[i] * 1.0 / max)
* height);
        filledRectangle.setFill(Color.RED);
        this.getChildren().add(filledRectangle);
      }
    }
  }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Pomoću slike 2 primeniti bubble sort na sledeće brojeve {45, 11, 50, 59, 60, 2, 4, 7, 10}.

Zadatak 2. Ako je lista već sortirana koliko će bubble sort uraditi poređenja?

Zadatak 3. Napisati program koji će zahtevati unos određenog broja studenata, za svakog studenta se traži ime i ocena. Sortirati listu studenata opadajuće po oceni.
Zadatak 4. Dve niske su anagrami ako se sastoje od istog broja istih karaktera. Napisati program koji proverava da li su dve niske karaktera anagrami.
Niske se zadaju sa standardnog ulaza i neće biti duže od 127 karaktera. Uputstvo: Napisati funkciju koja sortira slova unutar niske karaktera, a zatim za sortirane niske proveriti da li su identične.

Zadatak 5. U datom nizu brojeva treba pronaći dva broja koja su na najmanjem rastojanju. Niz se zadaje sa standardnog ulaza, sve do kraja ulaza, ali neće sadržati više od 256 i manje od 2 elemenata. Na izlaz ispisati razliku pronađena dva broja. Uputstvo: Prvo sortirati niz.

Zadatak 6. Implementirati sledeće generičke metode upotrebom bubble sorta. Prvi metod koristi Comparable interfejs, a drugi Comparator interfejs.

```
public static <E extends Comparable<E>>
    void bubbleSort(E[] list)
public static <E> void bubbleSort(E[] list,
    Comparator<? super E> comparator)
```

Poglavlje 3

Sortiranje spajanjem – merge sort

OSNOVI SORTIRANJA SPAJANJEM

Sortiranje spajanjem ili merge sort je rekurzivni algoritam sortiranja zasnovan na poređenju. Predstavlja primer algoritamske strategije podeli i osvoji

Sortiranje spajanjem ili "merge sort" je algoritam sortiranja zasnovan na poređenju. To je rekurzivni algoritam. Njegova vremenska složenost proporcionalna je sa O(n log n), a u srednjem slučaju je uvek efikasniji od algoritma brzog sortiranja (quick sort).

U većini implementacija je stabilna, što znači da zadržava početni redosled jednakih elemenata u sortiranom nizu.

Predstavlja primer algoritamske paradigme"*podeli i osvoji*". Konstruisao ga je Džon fon Nojman (John von Neumann) 1945. godine. Konceptualno, algoritam sortiranja spajanjem se izvršava na sledeći način:

a) Ako niz ima nula ili jedan element, onda je već sortiran.

b) U slučaju da niz ima više elemenata:

- Podeliti nesortirani niz u dva podniza približno jednake dužine
- Sortirati svaki podniza rekurzivno ponovnom primenom algoritma sortiranja spajanjem
- Spojiti dva sortirana podniza u jedan sortirani niz.

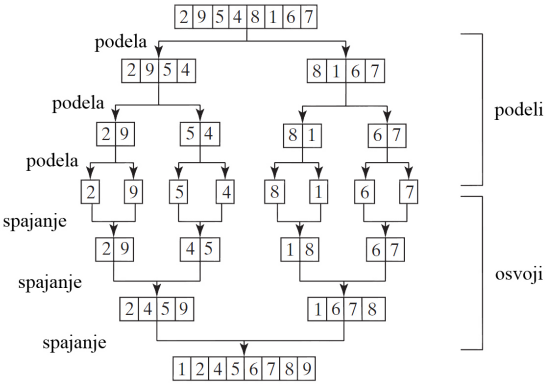
Algoritam sortiranja spajanjem uključuje dva važna principa kojima poboljšava (smanjuje) vremena izvršavanja:

- kratki niz je moguće sortirati u manjem broju koraka nego dugački (osnova za deljenje niza na dva podniza)
- manje koraka je potrebno za konstrukciju sortiranog niza od dva sortirana podniza nego od dva nesortirana podniza (osnova za spajanje).

ZAŠTO SORTIRANJE SPAJANJEM?

Sastoji se iz deljenja nesortiranog niza u dva podniza približno jednake dužine, a zatim nakon rekurzivnog sortiranja svake od polovina vrši se spajanje u uređen niz

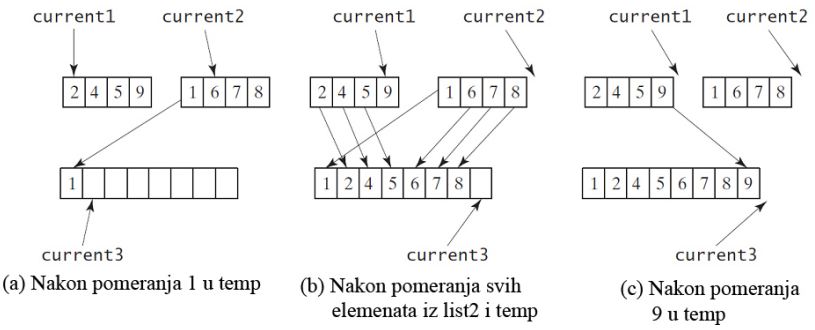
Da bi smo razumeli kako sortiranje spajanjem radi pogledajmo sledeću sliku (Slika- 1) .



Slika 3.1 Ilustracija sortiranja spajanjem koje koristi "podeli i osvoji" princip

Slika-1 il ustruje algoritam spajanjem koji je primenjen na nizu od 8 elemenata (2 9 5 4 8 1 6 7). Originalni niz delimo (divide) na dva dela (2 9 5 4) i (8 1 6 7). Zatim rekurzivno primenjujemo merge sort na svaki od podnizova, sa ciljem da podelimo podniz (2 9 5 4) na dva nova dela (2 9) i (5 4), odnosno drugi podniz (8 1 6 7) na delove (8 1) i (6 7). Ovaj proces se ponavlja sve dok u podnizu ne ostane samo jedan element. Na primer, niz (2 9) delimo na dva podniza (2) i (9).

Pošto sada niz (2) sadrži samo jedan element, on ne može dalje biti podeljen. Sada spojimo (merge) podnizove (2) i (9) u novi sortirani podniz (2 9); spojimo (5) i (4) u novi sortirani podniz (4 5). Nakon toga spojimo (2 9) sa (4 5) i dobijemo sortirani podniz (2 4 5 9), i konačno spojimo (2 4 5 9) i (1 6 7 8) u sortirani niz (1 2 4 5 6 7 8 9). Da bi smo bolje razumeli kako se vrši implementiranje algoritma spajanja pogledajmo Sliku-2:



Slika 3.2 Dva sortirana podniza se spajaju u jedan niz

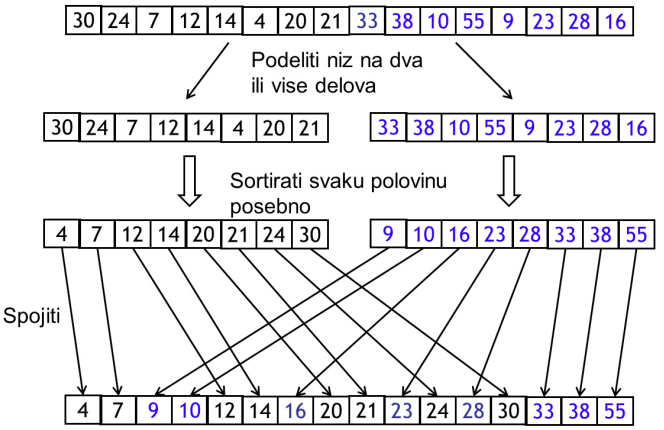
Kao što vidimo, imamo dva podniza *list1* i *list2*, a rezultat njihovog spajanja pakujemo u niz *temp*. Imamo brojače *current1* i *current2* za svaki od podnizova, preko kojih se vrši poređenje i odluka koji sledeći element ide u *temp*. Pri svakoj odluci uvećavamo *current1* ili *current2* za jedan, a onda ponovo upoređujemo elementa, i sve tako dok ne potrošimo sve elemente.

PRIMERI SORTIRANJA SPAJANJEM

Sortiranje spajanjem koristi princip da je manje koraka potrebno za konstrukciju sortiranog niza od dva sortirana podniza nego od dva nesortirana podniza

Primer 1

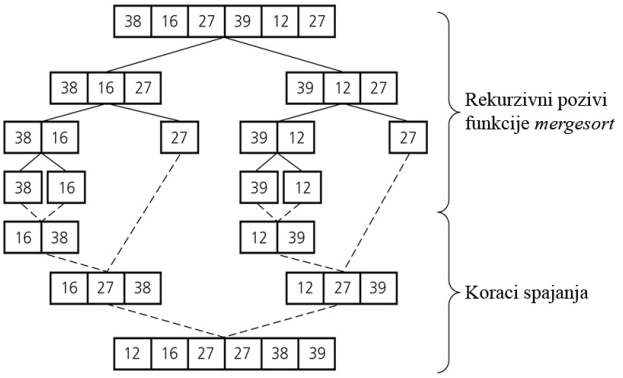
Slika- 3 daje primer sortiranja spajanjem:



Slika 3.3 Sortiranje spajanjem. Postupak podele niza i redosled kako elementi ulaze u spojeni niz

Primer 2

Pogledajmo još jedan primer (Slika- 4) :



Slika 3.4 Primer sortiranja spajanjem

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

IMPLEMENTACIJA SORTIRANJA SPAJANJEM

Kod se sastoji iz rekurzivne funkcije mergesort u kojoj se vrši rekurzivna podela niza na polovine (a zatim i na pola od polovine) i funkcije merge koja zatim spaja sortirane polovine

Pseudokod algoritma sortiranja spajanjem se uprošćeno može napisati na sledeći način:

```
mergeSort(list):
   firstHalf = mergeSort(firstHalf);
   secondHalf = mergeSort(secondHalf);
   list = merge(firstHalf, secondHalf);
```

Kod koji vrši sortiranje spajanjem, na osnovu pretodnog pseudokoda može biti napisan na sledeći način:

```
public static void mergesort(DataType theArray[], int first, int last)
{
 if (first < last)
 {
  int mid = (first + last)/2; // index of midpoint
  mergesort(theArray, first, mid);
  mergesort(theArray, mid+1, last);

  // merge the two halves
  merge(theArray, first, mid, last);
 }
} // end mergesort
```

Ostaje naravno da se implementira metoda merge koja spaja dva podniza u jedan sortirani niz. Postavlja se pitanje kako to uraditi.

U narednom listingu je prikazan Java kod koji radi po principu prethodnih algoritama, s tim što je metoda merge implementirana po principu opisanom na Slici-2:

```
public class MergeSort {
  public static void mergeSort(int[] list) {
    if (list.length > 1) {
    // Merge sort the first half
    int[] firstHalf = new int[list.length / 2];
    System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
    mergeSort(firstHalf);

    // Merge sort the second half
    int secondHalfLength = list.length - list.length / 2;
    int[] secondHalf = new int[secondHalfLength];
    System.arraycopy(list, list.length / 2,
        secondHalf, 0, secondHalfLength);
    mergeSort(secondHalf);
```

```
    // Merge firstHalf with secondHalf into list
    merge(firstHalf, secondHalf, list);
    }
  }

  public static void merge(int[] list1, int[] list2, int[] temp) {
    int current1 = 0; // Current index in list1
    int current2 = 0; // Current index in list2
    int current3 = 0; // Current index in temp

    while (current1 < list1.length &amp;amp;&amp;amp; current2 < list2.length) {
    if (list1[current1] < list2[current2])
        temp[current3++] = list1[current1++];
    else
        temp[current3++] = list2[current2++];
    }

    while (current1 < list1.length)
    temp[current3++] = list1[current1++];

    while (current2 < list2.length)
    temp[current3++] = list2[current2++];
  }

  public static void main(String[] args) {
    int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
    mergeSort(list);
    for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  }
}
```

ANALIZA ALGORITMA

Vremenska složenost algoritma je O(n log n). Ovaj algoritam je stoga bolji od svih kvadratnih algoritama kao što su selection sort, insertion sort, i bubble sort

Neka T(n) predstavlja vreme potrebno za sortiranje niza od *n* elemenata korišćenjem sortiranja spajanjem. Pretpostavimo da je *n* stepen broja 2. Merge sort algoritam deli niz na dva podniza, sortira podnizove korištenjem rekurzije i onda spaja dva podniza. Dakle:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + VremeSpajanja(merge)$$

Prvo T(n/2) je vreme za sortiranje prve polovine niza, a drugo T(n/2) je vreme za sortiranje druge polovine niza. Za spajanje podnizova je potrebno najviše *n-1* poređenja elemenata ova dva podniza i *n* pomeranja elemenata u privremenu listu. Ukupno vreme je *2n-1*. Imamo da je složenost ovog algoritma:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 2n - 1 = O(nlogn)$$

Ovaj algoritam je stoga bolji od svih kvadratnih algoritama kao što su selection sort, insertion sort, i bubble sort.

Najbolji slučaj:

- svi elementi u prvom nizu su manji (ili veći), nego svi elementi u drugom nizu.

- broj pomeranja: 2k + 2k
- broj poređenja: k

Najgori slučaj:

- broj pomeranja: 2k + 2k
- broj poređenja: 2k-1

Merge sort je jako efikasan algoritam u odnosu na vreme, i najgori i prosečan slučaj su O(n $\log_2$ n). Ali, merge sort zahteva dodatni niz čija je veličina jednaka veličini originalnog niza. Ako se koristi povezana lista (linked list), o kojoj ćemo pričati u nekoj od narednih lekcija, ne treba dodatni niz:

- treba prostor za linkove
- teže je podeliti listu na polovine (O(n)).

U slučaju da želimo da spajamo velike skupove podataka, neophodno je:

- pročitati što je moguće više podataka iz memorije
- sortirati ih koristeći brzi algoritam sortiranja (npr. quicksort)
- upisati sortirane podatke u novu datoteku.

ZADATAK: GENERIČKI MERGE SORT

Cilj zadatka je kreiranje generičkih metoda koje vrše sortiranje korišćenjem metode spajanja

Napisati sledeća dva generička metoda korišćenjem metode spajanja - merge sort. Prvi metod sortira elemente korišćenjem interfejsa Comparable a drugi koristi interfejs Comparator.

```
public static <E extends Comparable<E>>
void mergeSort(E[] list)
public static <E> void mergeSort(E[] list,
Comparator<? super E> comparator)
```

Rešenje je dato u nastavku:

```
import java.util.Comparator;

public class Zadatak2 {
  /** The method for sorting the numbers */
  public static <E extends Comparable<E>> void mergeSort(E[] list) {
    if (list.length > 1) {
```

```
  // Merge sort the first half
  E[] firstHalf = (E[])new Comparable[list.length / 2];
  System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
  mergeSort(firstHalf);

  // Merge sort the second half
  int secondHalfLength = list.length - list.length / 2;
  E[] secondHalf = (E[])new Comparable[secondHalfLength];
  System.arraycopy(list, list.length / 2,
                  secondHalf, 0, secondHalfLength);
  mergeSort(secondHalf);

  // Merge firstHalf with secondHalf
  E[] temp = merge(firstHalf, secondHalf, list);
  System.arraycopy(temp, 0, list, 0, temp.length);
  }
}

public static<E extends Comparable<E>> E[]
  merge(E[] list1, E[] list2, E[] temp) {

  int current1 = 0; // Index in list1
  int current2 = 0; // Index in list2
  int current3 = 0; // Index in temp

  while (current1 < list1.length &amp;amp;&amp;amp; current2 < list2.length) {
  if (list1[current1].compareTo(list2[current2]) < 0) {
      temp[current3++] = list1[current1++];
  }
  else {
      temp[current3++] = list2[current2++];
  }
  }

  while (current1 < list1.length) {
  temp[current3++] = list1[current1++];
  }

  while (current2 < list2.length) {
  temp[current3++] = list2[current2++];
  }

  return temp;
}

public static void main(String[] args) {
  Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
  mergeSort(list);
  for (int i = 0; i < list.length; i++) {
  System.out.print(list[i] + " ");
  }

  System.out.println();
```

```
  Circle[] list1 = {new Circle(2), new Circle(3), new Circle(2),
                    new Circle(5), new Circle(6), new Circle(1), new Circle(2),
                    new Circle(3), new Circle(14), new Circle(12)};
  mergeSort(list1, new GeometricObjectComparator());
  for (int i = 0; i < list1.length; i++) {
  System.out.print(list1[i] + " ");
  }

}

public static <E> void mergeSort(E[] list,
  Comparator<? super E> comparator) {
  if (list.length > 1) {
  // Merge sort the first half
  E[] firstHalf = (E[])new Object[list.length / 2];
  System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
  mergeSort(firstHalf, comparator);

  // Merge sort the second half
  int secondHalfLength = list.length - list.length / 2;
  E[] secondHalf = (E[])new Object[secondHalfLength];
  System.arraycopy(list, list.length / 2,
                    secondHalf, 0, secondHalfLength);
  mergeSort(secondHalf, comparator);

  // Merge firstHalf with secondHalf
  E[] temp = merge1(firstHalf, secondHalf, comparator);
  System.arraycopy(temp, 0, list, 0, temp.length);
  }
}

private static<E> E[]
  merge1(E[] list1, E[] list2, Comparator<? super E> comparator) {
  E[] temp = (E[])new Object[list1.length + list2.length];

  int current1 = 0; // Index in list1
  int current2 = 0; // Index in list2
  int current3 = 0; // Index in temp

  while (current1 < list1.length &amp;amp;&amp;amp; current2 < list2.length) {
  if (comparator.compare(list1[current1], list2[current2]) < 0) {
      temp[current3++] = list1[current1++];
  }
  else {
      temp[current3++] = list2[current2++];
  }
  }

  while (current1 < list1.length) {
  temp[current3++] = list1[current1++];
  }

  while (current2 < list2.length) {
```

```
    temp[current3++] = list2[current2++];
    }

    return temp;
  }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Primeniti sortiranje spajanjem na osnovu Slike 1 na sledeće brojeve {45, 11, 50, 59, 60, 2, 4, 7, 10}.

Zadatak 2. Šta se dešava ako se linije 6–15 klase MergeSort zamene sledećim kodom?

```
int[] firstHalf = new int[list.length / 2 + 1];
System.arraycopy(list, 0, firstHalf, 0, list.length / 2 + 1);
mergeSort(firstHalf);

// Merge sort the second half
int secondHalfLength = list.length - list.length / 2 - 1;
int[] secondHalf = new int[secondHalfLength];
System.arraycopy(list, list.length / 2 + 1,
    secondHalf, 0, secondHalfLength);
mergeSort(secondHalf);
```

Zadatak 3. Napisati program koji pronalazi broj koji se najviše puta pojavljivao u datom nizu. Niz se zadaje sa standardnog ulaza sve do kraja ulaza i neće biti duži od 256 i kraći od jednog elemenata. Uputstvo: Prvo sortirati niz, a zatim naći najdužu sekvencu jednakih elemenata.

Poglavlje 4

Brzo sortiranje – quick sort

UVOD U BRZO SORTIRANJE

Ideja algoritma sastoji se u podeli niza prema odabranom elementu za podelu koji se dovodi na pravo mesto, i u primeni algoritma brzog sortiranja na svaku od dva dobijena podniza

Ovo je najčešće upotrebljavan algoritam sortiranja. Nije težak za implementaciju, a koristi manje resursa (vremena i prostora) nego bilo koji drugi algoritam sortiranja, u većini slučajeva.

Algoritam ne zahteva dodatnu memoriju, samo n*log(n) operacija u proseku za sortiranje *n* elemenata, i ima izuzetno kratku unutrašnju petlju. Loše strane algoritma su što je rekurzivan (nerekurzivna varijanta je mnogo složenija), u najgorem slučaju izvršava oko n^2 operacija.

Algoritam je vrlo osetljiv na implementaciju (efikasnost se može narušiti lošim izborom u implementaciji).

Ideja algoritma sastoji se u podeli niza prema odabranom elementu za podelu koji se dovodi na pravo mesto, i u primeni algoritma brzog sortiranja na svaku od dva dobijena podniza.

Rekurzivni poziv se završava kada se primeni na podniz sa manje od dva elementa.

Procedura

Procedura je data na Slici- 1 . Algoritam se sastoji iz sledećih koraka:

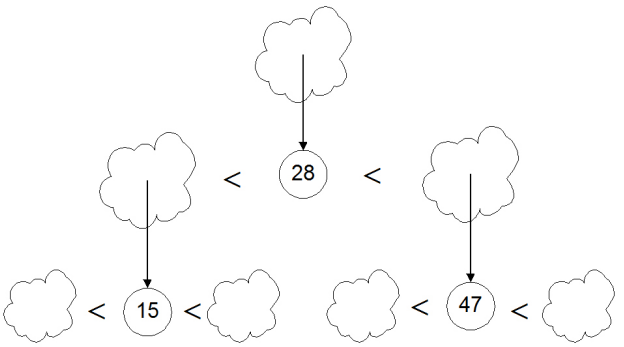
a) Izabrati *“pivot”*

b) Podeliti listu na dve liste:

- jednu sa vrednostima *“manjim ili jednakim”* od vrednosti pivota
- jednu sa vrednostima *većim* od vrednosti pivota

c) Sortirati potproblem rekurzivno

Rešenje se dobija spajanjem (engl. concatenation) dva podniza



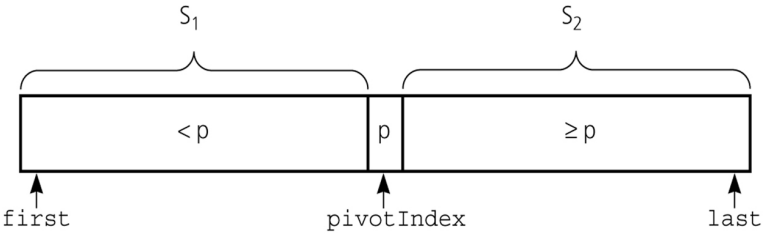
Slika 4.1 Postupak brzog sortiranja

POSTUPAK SORTIRANJA – PODELA I IZBOR PIVOTA

Prvi korak u fazi sortiranja je izbor pivota. Kao pivot se najčešće bira prvi, srednji ili poslednji element trenutnog podniza

Podela

Podela stavlja pivot na pravo mesto u okviru niza (Slika- 2).



Slika 4.2 Pravilna podela niza na dva dela koje deli pivot

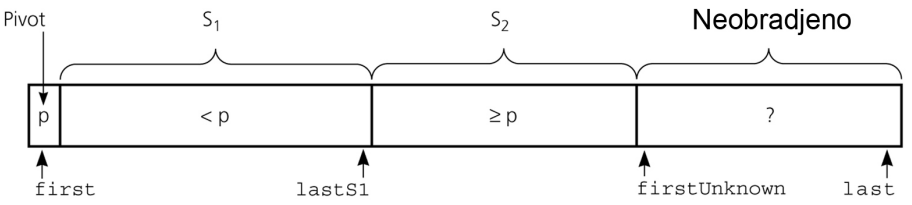
Uređivanje elemenata niza oko pivota p generiše dva manja problema pri sortiranju. To su sortiranje levog dela niza i sortiranje desnog dela niza. Kada se ova dva manja problema rešavaju rekurzivno, veći problem je time rešen.

Izbor pivota

Prvo, mora se izabrati pivot element među elementima datog niza, zatim staviti ovaj pivot na prvu lokaciju niza pre podele. Koji element niza treba da bude izabaran kao pivot?

Ako su elementi niza slučajno odabrani, izabrati pivot nasumice. Može se odabrati prvi ili poslednji element kao pivot,ali možda se tako ne dobije dobra podela. Mogu se koristiti druge metode izbora pivota.

Invarijanta za algoritam podele (Slika-3).

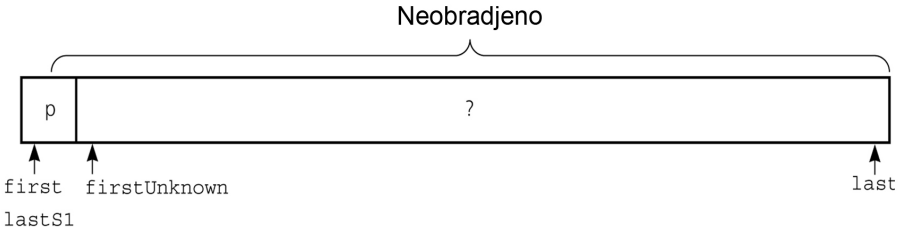


Slika 4.3 Postupak u toku izvršavanja podele niza. Pivot je inicijalno na prvom članu. Koristimo repere lastS1 i firstUnknown da bi smo tačno razdovojili podeljene grupe podnizova

POSTUPAK SORTIRANJA – ZAMENA PIVOTA

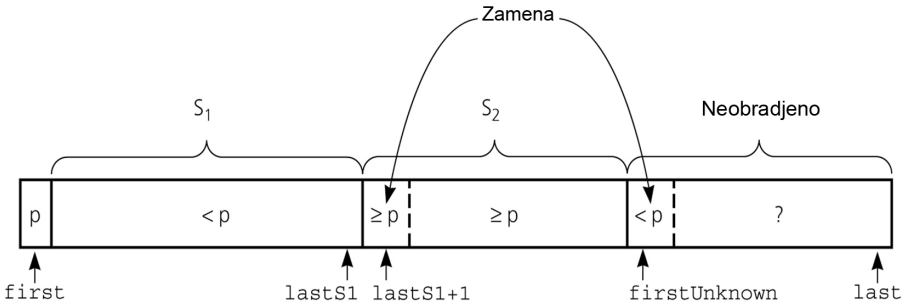
Nakon što se izvrši podela članova, pivot, koji je inicajlno bio na početku niza, se prebacuje na mesto koje deli elemente koji su manji od njega u odnosu na elemente koji su veći od njega

Početno stanje (Slika- 4) .



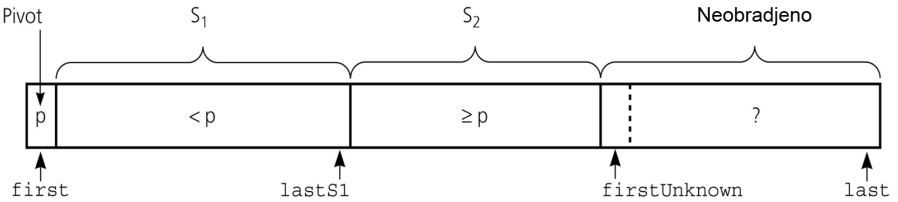
Slika 4.4 Početno stanje podele. Pivot je na prvom članu.

Pomeranje *theArray[firstUnknown]* u *S1* zamenom sa *theArray[lastS1+1]* i inkrementiranje *lastS1* i *firstUnknown* (Slika- 5) .



Slika 4.5 Postupak gde element na indeksu firstUnknown upoređujemo sa pivotom i onda ga pozicioniramo na određeno mesto u obrađenom delu niza.

Pomeranje *theArray[firstUnknown]* u *S2* i inkrementiranje *firstUnknown* (Slika- 6) .



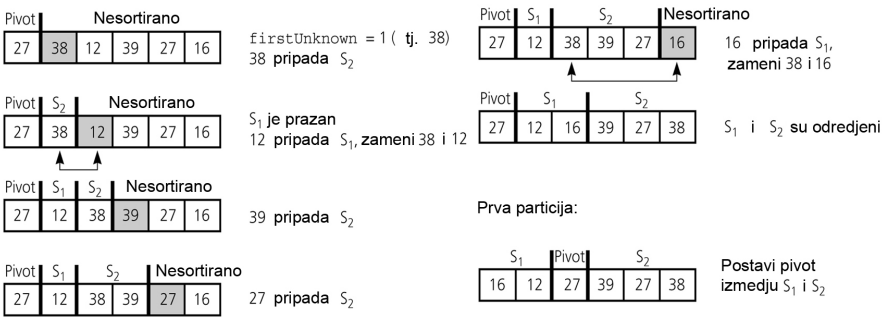
Slika 4.6 Ukoliko je element sa indeksom veći od pivota onda ostaje gde jeste. Indeks firstUnknown zatim uvećavamo za 1.

PRIMERI BRZOG SORTIRANJA

Postupak se sastoji iz izbora pivota, podele liste na dva dela prema vrednosti pivota, a zatim na svaku od polovina rekurzivno primeniti identičan postupak

Primer 1

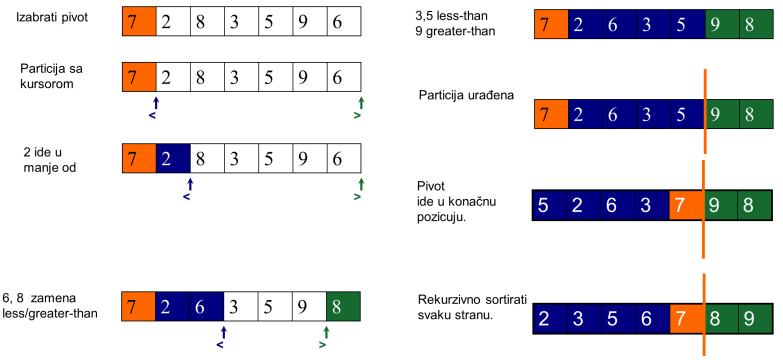
Slika- 7. daje primer brzog sortiranja.



Slika 4.7 Primer brzog sortiranja. Postupak podele niza na dva dela, u zavisnosti od veličine pivota

Primer 2

Pogledajmo još jedan primer (Slika- 8) :



Slika 4.8 Postupak pomeranja granica u obrađenom delu niza. Granica deli niz na one koji su manji od pivota i veći od pivota. Takođe imamo i granicu između obrađenog i neobrađenog dela

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

IMPLEMENTACIJA ALGORITMA

Implementacija funkcije quicksort se uglavnom sastoji iz funkcije partition koja pozicionira pivot, i dva rekurzivna poziva iste quicksort funkcije koje rade nad polovinama početnog niza

Pseudokod brzog sortiranja ima sledeći oblik:

```
quicksort(A, lo, hi):
    if lo < hi:
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)
// lo is the index of the leftmost element of the subarray
// hi is the index of the rightmost element of the subarray (inclusive)
```

pri čemu operacija *partition* može da se predstavi kao:

```
partition(A, lo, hi)
    pivotIndex := choosePivot(A, lo, hi)
    pivotValue := A[pivotIndex]
    // put the chosen pivot at A[hi]
    swap A[pivotIndex] and A[hi]
    storeIndex := lo
    // Compare remaining array elements against pivotValue = A[hi]
    for i from lo to hi-1, inclusive
        if A[i] < pivotValue
            swap A[i] and A[storeIndex]
            storeIndex := storeIndex + 1
    swap A[storeIndex] and A[hi] // Move pivot to its final place
    return storeIndex
```

Ovaj pseudokod, u nešto izmenjenom obliku, možemo napisati na sledeći način, u programskom jeziku Java:

```
private static void quickSort(int[] list, int first, int last) {
 if (last > first) {
  int pivotIndex = partition(list, first, last);
  quickSort(list, first, pivotIndex - 1);
  quickSort(list, pivotIndex + 1, last);
 }
}
/** Partition the array list[first..last] */
private static int partition(int[] list, int first, int last) {
 int pivot = list[first]; // Choose the first element as the pivot
 int low = first + 1; // Index for forward search
 int high = last; // Index for backward search
```

```
 while (high > low) {
  // Search forward from left
  while (low <= high &amp;amp;&amp;amp; list[low] <= pivot)
   low++;

  // Search backward from right
  while (low <= high &amp;amp;&amp;amp; list[high] > pivot)
   high--;

  // Swap two elements in the list
  if (high > low) {
   int temp = list[high];
   list[high] = list[low];
   list[low] = temp;
  }
 }

 while (high > first &amp;amp;&amp;amp; list[high] >= pivot)
  high--;

 // Swap pivot with list[high]
 if (pivot > list[high]) {
  list[first] = list[high];
  list[high] = pivot;
  return high;
 }
 else {
  return first;
 }
}
```

ANALIZA ALGORITMA

Za najgori slučaj ovaj algoritam je kvadratne složenosti, dok je u najboljem slučaju složenosti O(n lon n). U najboljem slučaju ovaj algoritam je bolji i efikasniji od merge sort algoritma

Da bi se izvršila podela niza od *n* elemenata, to zahteva *n* poređenja i *n* pomeranja u najgorem slučaju. Stoga vreme neophodno za podelu niza je O(n).

Najgori slučaj. Neka je pivot prvi element. U najgorem slučaju, on svaki put deli niz na dve podliste, pri čemu je jedna od *n-1* elemenata, a druga je prazna. Veličina većeg podniza je za jedan manja od veličine niza pre podele:

$T(n)=n-1+n-1+...+1=O(n^2)$.

Stoga je quick sort vremenske složenosti $O(n^2)$ za najgori slučaj.

Najbolji slučaj. U najboljem slučaju, pivot deli niz na dva podniza približno jednake veličine, oko n/2. Neka sa T(n) označimo vreme neophodno za sortiranje niza od n elemenata korišćenjem brzog sortiranja. Imaćemo:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n = O(logn)$$

Prosečan slučaj. U prosečnom slučaju, pivot neće podeliti niz na dva dela koji su ili približno jednaki ili je jadna potpuno prazna. Statistički, veličine oba podniza su veoma približne. Stoga, vreme izvršavanja prosečnog slučaja je O(n logn), ali matematička analiza prosečnog slučaja nije predmet ove lekcije.

Rezime

Izbor pivota zahteva konstatno vreme, dok deljenje (partitioning): linearno vreme

Rekurzija: vreme sortiranja leve particije (npr. veličine **i**) + vreme za desnu (veličine **N-i-1**) + vreme da se kombinuje rešenje.

T(1) = b

T(N) = T(i) + T(N-i-1) + cN, ovde je i broj elemenata koji su manji od pivota

Izbori pivota:

- Izabrati nasumice pivot - dobro teoretski, ali poziv generatora slučajnih brojeva može biti skup.
- Pametni izbor pivota - *"Median-of-3"* pravilno računa srednju vrednost (prvi, srednji, poslednji) elemenata

I quick sort i merge sort su algoritmi tipa *„podeli i osvoji"*. Za merge sort, najveći deo posla odlazi na spajanje dva niza. Za quick sort, najveći deo vremena odlazi na podelu niza na dva dela. merge sort je mnogo efikasniji u najgorem slučaju, dok je quick sort skoro jednako efikasan u prosečnom slučaju. merge sort zahteva dodatan prostor, pa je quick sort efikasniji u najboljem slučaju.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Zašto je quick sort prostorno efikasniji od sortiranja spajanjem?

Zadatak 2. Pomoću slike 7 primeniti quick sort na sledeći niz brojeva {45, 11, 50, 59, 60, 2, 4, 7, 10}.

Zadatak 3. (Poboljšani quick sort) Quick sort algoritam koji smo obradili na predavanjima bira prvi element kao pivot. Izmeniti program tako da može da se izabere prvi, srednji i poslednji element kao pivot.

Zadatak 4. Napisati funkciju koja proverava da li u datom nizu postoje dva elementa čiji zbir je jednak zadatom celom broju. Napisati i program koji testira ovu funkciju. U programu se prvo učitava broj, a zatim i niz. Elementi niza se unose sve do kraja ulaza. Pretpostaviti da u niz neće biti uneto više od 256 brojeva. Prvo sortirati niz..

Zadatak 5. Implementirati sledeće generičke metode primenom quick sorta. Prva metoda sortira elemente pomoću Comparable, a druga metoda pomoću Comparator interfejsa.

```
public static <E extends Comparable<E>>
    void quickSort(E[] list)
public static <E> void quickSort(E[] list,
    Comparator<? super E> comparator)
```

Poglavlje 5

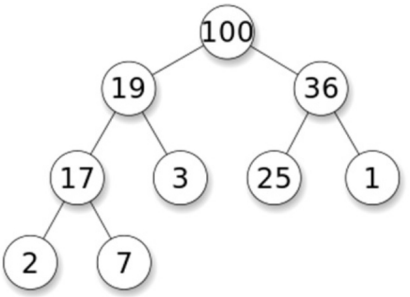
Sortiranje pomoću hipa – heap sort

OSNOVI O HIP SORTIRANJU

Hip je struktura podataka organizovana po principu stabla koje mora da zadovoljava uslov: ključ svakog čvora je veći ili jednak od ključeva njihovih sinova

Hip (heap) je struktura podataka organizovana po principu stabla koje mora da zadovoljava uslov: *ključ svakog čvora je veći ili jednak od ključeva njihovih sinova*.

Hip je veoma korisna struktura podataka kada treba da se ukloni čvor najvišeg ili najnižeg prioriteta. Primer hip stabla se nalazi na Slici-1.



Slika 5.1 Struktura hip stabla

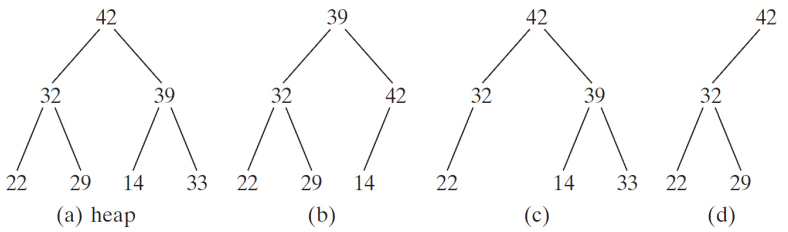
Binarno stablo je hijerarhijska struktura. Ono je ili prazno ili sadrži jedan element koji se naziva koren (root), i dve odvojene grane, koje se nazivaju *levo podstablo* i *desno podstablo*. O stablima će biti puno više reči u lekcijama L8 i L9.

Binarni hip je binarno stablo koje ima sledeća svojstva:

- svojstvo oblika: Hip je kompletno binarno stablo.
- svojstvo hipa: Svaki čvor je veći ili jednak od njegovih naslednika.

Binarno stablo je kompletno ako je svaki nivo stabla ispunjen osim zadnjeg nivoa koji ne mora biti ispunjen i svi listovi u zadnjem nivou su postavljeni s leva na desno.

Na primer, na sledećoj Slici-2, b inarna stabla u (a) i (b) su kompletna, ali binarna stabla u (c) i (d) nisu kompletna. Takođe, binarno stablo u (a) je hip, ali (b) nije hip, jer je koren (39) manji od desnog deteta (42) :



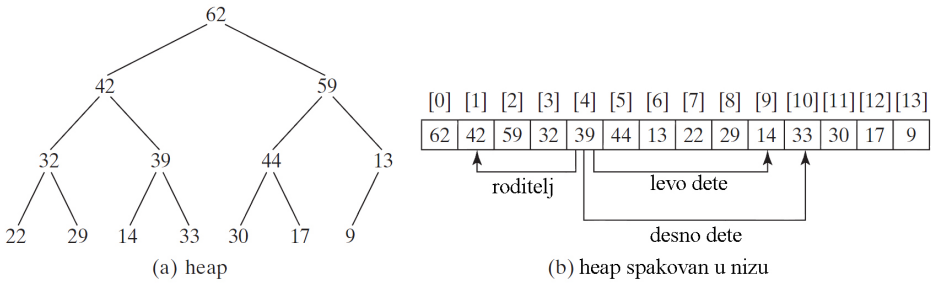
Slika 5.2 Kompletna i nekompletna binarna stabla

PREDSTAVLJANJE HIP STRUKTURE

Hip se obično čuva u nizu, pri čemu se za čvor na poziciji i, njegovo levo dete nalazi na poziciji 2i+1, njegovo desno dete na poziciji 2i+2, a njegov roditelj na indeksu (i-1)/2

Hip na Slici-3a može biti čuvan u nizu prikazanom na Slici-3b. Za čvor na poziciji i, njegovo levo dete je na poziciji 2i+1, njegovo desno dete je na poziciji 2i +2, a njegov roditelj je na indeksu (i-1)/2.

Na primer, element 39 je na poziciji 4, tako da je njegovo levo dete (element 14) na poziciji 9=2*4+1, njegovo desno dete (element 33) na poziciji 10 (2*4+2), a roditelj (element 42) je na poziciji 1 ((4-1)/2).



Slika 5.3 Predstavljanje hip strukture

U cilju boljeg uvida u sortiranje korišćenjem hipa, predlaže se poseta sledećeg linka: http://www.cs.armstrong.edu/liang/animation/web/Heap.html.

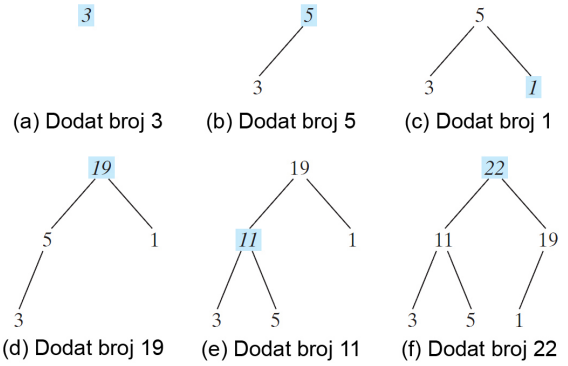
DODAVANJE ELEMENATA

Da bi ste dodali novi čvor u hip, prvo ga dodajte na kraj hipa, a zatim reorganizujte stablo tako što metodom isplivavanja novi član putuje ka korenu dok ne naiđe na prvi veći od sebe

Da bi ste dodali novi čvor u hip, prvo ga dodajte na kraj hipa, a zatim reorganizujte stablo na sledeći način:

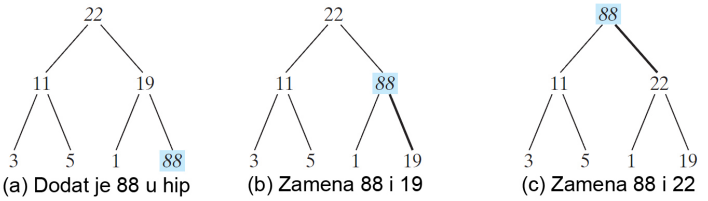
```
Let the last node be the current node;
while (the current node is greater than its parent) {
    Swap the current node with its parent;
    Now the current node is one level up;
}
```

Pretpostavimo da je struktura inicijalno prazna. Nakon dodavanja elemenata 3, 5, 1, 19, 11 i 22 u hip strukturu, ona dobija sledeći oblik (Slika-4):



Slika 5.4 Dodavanje elemenata u hip strukturu - Elementi 3, 5, 1, 19, 11, i 22 su ubačeni u hip

Razmotrimo sada dodavanje elementa 88 u postojeću hip strukturu. Reorganizacija se odvija prema sledećoj slici (Slika-5):



Slika 5.5 Dodavanje elementa 88 u hip strukturu

Postavimo novi čvor 88 na kraj hip stabla, kao što je prikazano na Slici- 5a . normalno, 88 i 19 menjaju mesta, kao što je prikazano na Slici- 5 b.

Pošto je 88 veće od 22, stoga oni takođe menjaju mesta, kao što je prikazano na Slici- 5 c.

UKLANJANJE KORENA I REGENERISANJE STABLA

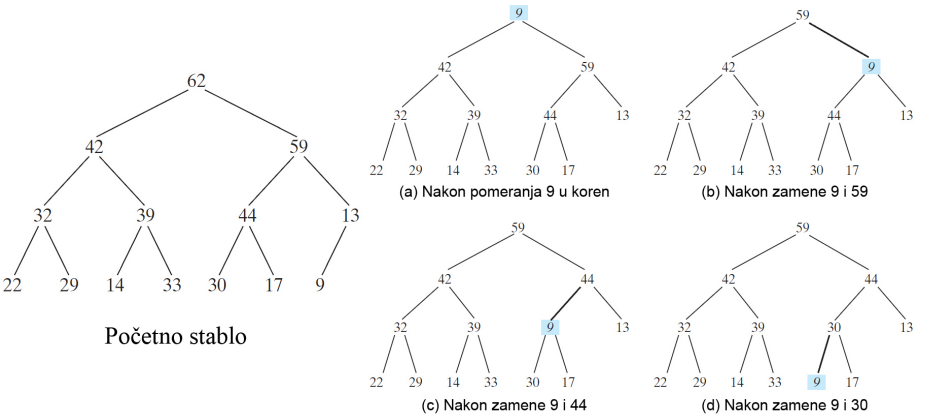
Nakon uklanjanja korena, poslednji element iz hip-a se postavlja u koren. Zatim se vrši reorganizacija hip-a tako što element iz korena potanja ka dnu što je dublje moguće

Često je neophodno obrisati maksimalni element koji se nalazi u korenu hip stabla. Nakon što je koren obrisan, stablo mora biti reorganizovano da bi zadržalo svojstvo hipa. Algoritam za reorganizaciju stabla može biti opisan na sledeći način:

```
Move the last node to replace the root;
Let the root be the current node;
while (the current node has children and the current node is
        smaller than one of its children) {
    Swap the current node with the larger of its children;
    Now the current node is one level down;
}
```

Na Slici- 6 je prikazan postupak reorganizacije hip stabla nakon što se koren 62 obriše. Postupak reorganizacije je sledeći:

- Pomeranje elementa 9 u koren, Slika- 6a .
- Zamena elemenata 9 i 59, Slika- 6b .
- Zamena 9 sa 44, Slika- 6c .
- Zamena 9 sa 30, Slika- 6d .



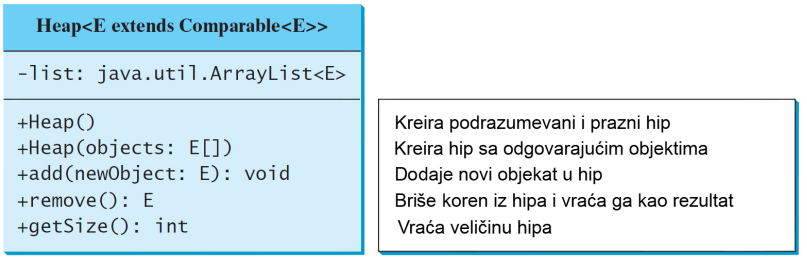
Slika 5.6 Uklanjanje korena 62 iz hipa

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

IMPLEMENTACIJA ALGORITMA

Implementacija se sastoji iz funkcije add koja dodaje novi element u stablo, i funkcije remove koja briše koren, a zatim i reorganizuje stablo

Sada možemo da implementiramo klasu koja će da služi za hip strukturu. Dijagram je prikazan na Slici- 7.



Slika 5.7 Klasni dijagram za Heap sort

Dodavanje elemenata može biti ostvareno na sledeći način:

```
public void add(E newObject) {
    list.add(newObject); // Append to the heap
    int currentIndex = list.size() - 1; // The index of the last node

    while (currentIndex > 0) {
        int parentIndex = (currentIndex - 1) / 2;
        // Swap if the current object is greater than its parent
        if (list.get(currentIndex).compareTo(
                list.get(parentIndex)) > 0) {
            E temp = list.get(currentIndex);
            list.set(currentIndex, list.get(parentIndex));
            list.set(parentIndex, temp);
        }
        else
            break; // the tree is a heap now

        currentIndex = parentIndex;
    }
}
```

Funkcija za brisanje elemenata može da ima sledeći oblik:

```
public E remove() {
    if (list.size() == 0) return null;

    E removedObject = list.get(0);
    list.set(0, list.get(list.size() - 1));
    list.remove(list.size() - 1);

    int currentIndex = 0;
    while (currentIndex < list.size()) {
        int leftChildIndex = 2 * currentIndex + 1;
        int rightChildIndex = 2 * currentIndex + 2;

        // Find the maximum between two children
        if (leftChildIndex >= list.size()) break; // The tree is a heap
        int maxIndex = leftChildIndex;
        if (rightChildIndex < list.size()) {
            if (list.get(maxIndex).compareTo(
                    list.get(rightChildIndex)) < 0) {
                maxIndex = rightChildIndex;
            }
        }

        // Swap if the current node is less than the maximum
        if (list.get(currentIndex).compareTo(
                list.get(maxIndex)) < 0) {
            E temp = list.get(maxIndex);
            list.set(maxIndex, list.get(currentIndex));
```

```
            list.set(currentIndex, temp);
            currentIndex = maxIndex;
        }
        else
            break; // The tree is a heap
    }

    return removedObject;
}
```

ANALIZA ALGORITMA

Vremenska složenost kreiranja hipa je O(logn), dok je vremenska složenost kreiranja sortiranog niza iz hipa O(n log n). Hip sort ne zahteva nikakav dodatni prostor pa je bolji od merge sort-a

Označimo sa *h* visinu hipa koji ima *n* elemenata. Visina hipa je u stvari broj čvorova u najdužoj putanji od korena do lista. Pošto je hip kompletno binarno stablo, prvi nivo ima 1 čvor, drugi nivo ima 2 čvora, a k-ti nivo ima 2^{k-1} čvorova, (h - 1)-vi nivo ima 2^{h-2} čvorova, a h-ti nivo ima najmanje 1, a najviše 2^{h-2} čvorova. Stoga važi:

$$1 + 2 + \ldots + 2^{h-2} \leq 1 + 2 + \ldots + 2^{h-2} + 2^{h-1}$$

pa je:

$$2^{h-1} - 1 < n \leq 2^h - 1$$

$$2^{h-1} < n + 1 \leq 2^h$$

$$h - 1 < log(n + 1) \leq h$$

Dakle važi da je h < log(n + 1) + 1 i log(n + 1) ≤ h. Stoga je log(n + 1) ≤ h < log(n + 1) + 1. Zaključujemo da je vremenska složenost hip sort algoritma O(logn).
Pošto metod *add* traži put od lista do korena, potrebno mu je najviše *h* koraka da bi dodao novi element u hip. Stoga, ukupno vreme za konstruisanje inicijalnog hipa je O(n logn) za niz od *n* elemenata.

Pošto se metod *remove* poziva *n* puta, ukupno vreme za stvaranje sortiranog niza iz hipa je O(n logn).

Kao što se može primetiti, i heap sort i merge sort zahtevaju vreme *O*(*n* log*n*). Merge sort zahteva pomoćni niz za spajanje dva podniza, dok heap sort ne zahteva nikakav dodatni prostor. Stoga je heap sort mnogo efikasniji od merge sort algoritma.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

ZADATAK: IMPLEMENTACIJA HEAP SORT-A

Cilj zadatka je implementacija hip sort-a koji smo obradili na predavanjima

a) Generisanje Hip stabla:

```
public class Heap<E extends Comparable<E>> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<E>();

  /** Create a default heap */
  public Heap() {
  }

  /** Create a heap from an array of objects */
  public Heap(E[] objects) {
    for (int i = 0; i < objects.length; i++)
    add(objects[i]);
  }

  /** Add a new object into the heap */
  public void add(E newObject) {
    list.add(newObject); // Append to the heap
    int currentIndex = list.size() - 1; // The index of the last node

    while (currentIndex > 0) {
    int parentIndex = (currentIndex - 1) / 2;
    // Swap if the current object is greater than its parent
    if (list.get(currentIndex).compareTo(
        list.get(parentIndex)) > 0) {
        E temp = list.get(currentIndex);
        list.set(currentIndex, list.get(parentIndex));
        list.set(parentIndex, temp);
    }
    else
        break; // the tree is a heap now

    currentIndex = parentIndex;
    }
  }

  /** Remove the root from the heap */
  public E remove() {
    if (list.size() == 0) return null;

    E removedObject = list.get(0);
    list.set(0, list.get(list.size() - 1));
    list.remove(list.size() - 1);

    int currentIndex = 0;
    while (currentIndex < list.size()) {
```

```
    int leftChildIndex = 2 * currentIndex + 1;
    int rightChildIndex = 2 * currentIndex + 2;

    // Find the maximum between two children
    if (leftChildIndex >= list.size()) break; // The tree is a heap
    int maxIndex = leftChildIndex;
    if (rightChildIndex < list.size()) {
        if (list.get(maxIndex).compareTo(
            list.get(rightChildIndex)) < 0) {
        maxIndex = rightChildIndex;
        }
    }

    // Swap if the current node is less than the maximum
    if (list.get(currentIndex).compareTo(
        list.get(maxIndex)) < 0) {
        E temp = list.get(maxIndex);
        list.set(maxIndex, list.get(currentIndex));
        list.set(currentIndex, temp);
        currentIndex = maxIndex;
    }
    else
        break; // The tree is a heap
    }

    return removedObject;
  }

  /** Get the number of nodes in the tree */
  public int getSize() {
    return list.size();
  }
}
```

b) Sortiranje korišćenjem Hip stabla:

```
public class HeapSort {
  /** Heap sort method */
  public static <E extends Comparable<E>> void heapSort(E[] list) {
    // Create a Heap of integers
    Heap<E> heap = new Heap<E>();

    // Add elements to the heap
    for (int i = 0; i < list.length; i++)
    heap.add(list[i]);

    // Remove elements from the heap
    for (int i = list.length - 1; i >= 0; i--)
    list[i] = heap.remove();
  }

  /** A test method */
  public static void main(String[] args) {
```

```
    Integer[] list = {-44, -5, -3, 3, 3, 1, -4, 0, 1, 2, 4, 5, 53};
    heapSort(list);
    for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Šta je povratna vrednost metode *remove* u slučaju da je hip prazan?

Zadatak 2. Dodati redom elemente 4, 5, 1, 2, 9, and 3 u hip. Nacrtati dijagrame koji prikazuju hip strukturu posle svakog dodavanja.

Zadatak 3. Prikazati hip strukturu nakon brisanja korena 59 sa hipa na slici 6d.

Zadatak 4. Prikazati korake kreiranja hip strukture pomoću sledećih brojeva {45, 11, 50, 59, 60, 2, 4, 7, 10}.

Zadatak 5. Koje od sledećih naredbi su pogrešne?

```
Heap<Object> heap1 = new Heap<Object>();
Heap<Number> heap2 = new Heap<Number>();
Heap<BigInteger> heap3 = new Heap<BigInteger>();
Heap<Calendar> heap4 = new Heap<Calendar>();
Heap<String> heap5 = new Heap<String>();
```

Zadatak 6. (*Sortiranje korišćenjem hipa*) Implementirati sledeći metod za sortiranje korišćenjem hipa.

```
public static <E extends Comparable<E>> void sort(E[] list)
```

Zadatak 7. Napisati sledeće metode upotrebom hip sortiranja. Prva metoda sortira elemente pomoću Comparable, a druga metoda pomoću Comparator interfejsa.

```
public static <E extends Comparable<E>>
    void heapSort(E[] list)
public static <E> void heapSort(E[] list,
    Comparator<? super E> comparator)
```

Zadatak 8. Napisati Java aplikaciju koja vizuelno prikazuje hip strukturu. Aplikacija treba da omogući dodavanje i brisanje elemenata.

Poglavlje 6

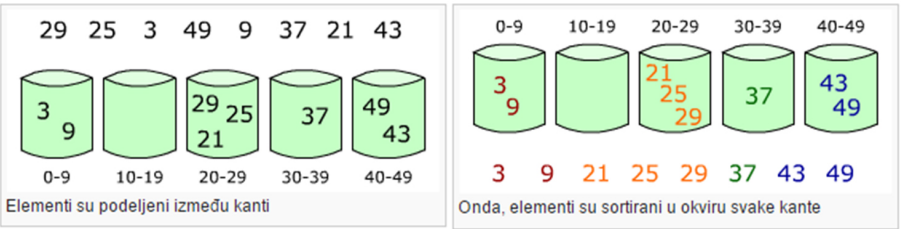
Segmentno (bucket) i radiks (radix) sortiranje

UVOD U SEGMENTNO SORTIRANJE – BUCKET SORT

Segmentno sortiranje je algoritam za sortiranje koji deli niz na određeni broj kofa. Svaka kofa se onda sortira pojedinačno, koristeći neki drugi algoritam za sortiranje

Svi algoritmi sortiranja koje smo opisali do sada (selection, insertion, quick merge, heap sort) su opšti algoritmi koji rade sa bilo kojim tipom podataka (tj. celim brojevima, stringovima, i bilo kojim objektima koji su uporedivi). Ovi algoritmi sortiraju elemente upoređivanjem njihovih ključeva. Donja granica za opšte algoritme sortiranja je bila $O(n \log n)$.

Međutim, ukoliko su ključevi koje poredimo u stvari mali celi brojevi, moguće je koristiti segmentno sortiranje bez potrebe za poređenjem ključeva. Primer segmentnog sortiranja je prikazan na Slici-1:

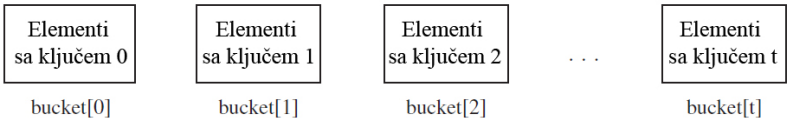


Slika 6.1.1 Primer segmentnog sortiranja

Segmentno sortiranje je algoritam za sortiranje koji deli niz na određeni broj kanti (bucket). Svaka kanta se onda sortira pojedinačno - korišćenjem nekog drugog algoritma za sortiranje.

Bucket sort algoritam radi na sledeći način:

Pretpostavimo da su ključevi u opsegu od **0** do **t**. Potrebne su nam **t + 1** kante označene sa **0**, **1**, . . . , i **t**. Ukoliko je ključ nekog elementa **i**, onda se element pakuje u kantu sa indeksom **i**. Svaka kanta sadrži elemente sa istim ključem (Slika- 2) .



Slika 6.1.2 Postupak rada algoritma segmentnog sortiranja

Treba napomenuti da u slučaju da je **t** veoma veliko, korišćenje segmentnog sortiranja nije poželjno. U tom slučaju treba koristiti radix sort. Radix sort je baziran na segmentnom sortiranju, s tim što radix sort koristi samo deset kanti. O radix sort algoritmu će detaljno biti reči kasnije.

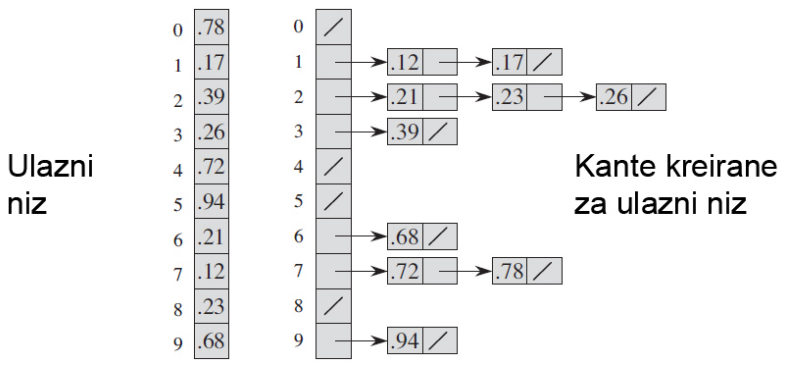
IMPLEMENTACIJA SEGMENTNOG SORTIRANJA

Za niz kofa potrebno je kreirati odgovarajući dinamički ili statički niz. U okviru jedne kofe elementi se mogu čuvati ili u povezanim listama ili u dinamičkom nizu

Pseudokod segmentnog sortiranja može biti napisan kao:

```
function bucketSort(array, n) is
    buckets ← new array of n empty lists
    for i = 0 to (length(array)-1) do
        insert array[i] into buckets[msbits(array[i], k)]
    for i = 0 to n - 1 do
        nextSort(buckets[i])
    return the concatenation of buckets[0], ...., buckets[n-1]
```

Sledeći dijagram demonstrira moguću implementaciju algoritma(Slika-3):



Slika 6.1.3 Primer implementacije kofe pomoću dinamičkih struktura

Ako su ključevi od 1…K, postoji niz od K kanti. Unutar jedne kante možemo koristiti povezane liste ili dinamičke nizove za podatke (Slika- 3) .

Kod može biti napisan na sledeći način:

```
public static void bucketSort(int input[], int n)
    {
        List[] buckets = new List[n];
        for (int i = 0; i < n; i++) {
            buckets[i] = new ArrayList<Integer>();
        }

        // 2) Put array elements in different buckets
        for (int i=0; i<n; i++)
```

```
        {
            int bi = (int) (n*input[i]); // Index in bucket
            buckets[bi].add(input[i]);
        }

        // 3) Sort individual buckets
        for (List bucket : buckets) {
            Collections.sort(bucket);
        }

        // 4) Concatenate all buckets into arr[]
        int ndx = 0;
        // merge the buckets: O(n)
        for (int b = 0; b < buckets.length; b++) {
            for (int v : buckets[b]) {
                input[ndx++] = v;
            }
        }
    }
```

ANALIZA SEGMENTNOG SORTIRANJA

Segmentno sortiranje je stabilno - ako 2 elementa u početnoj listi imaju isti ključ njihov redosled neće biti izmenjen u sortiranoj listi. Složenost algoritma u najboljem slučaju je O(n+k)

Segmentno sortiranje radi na sledeći način:

- Kreirati niz kanti koje su inicijalno prazne.
- Obići originalni niz, u smestiti svaki od članova u njemu odgovarajuću kantu.
- Sortirati svaku nepraznu kantu.

Kako se dobija sortirani niz:

- Posetiti redom svaku od kanti, pokupiti elemente iz njih i napuniti originalni niz.

Jasno, potrebno je $O(n + t)$ vremena da se sortira lista i potreban je prosor $O(n + t)$, gde je n veličina liste.

Najgori slučaj: $O(n^2)$

Najbolji slučaj: O(n+k)

Prosečan slučaj: O(n+k)

Korisno je za kraj napomenuti da je segmentno sortiranje stabilno, što znači da ako dva elementa u originalnoj listi imaju isti ključ, njihov redosled neće biti promenjen u sortiranoj listi. To znači da, ako element e_1 i element e_2 imaju isti ključ i e_1 prethodi e_2 u originalnoj listi, e_1 će prethoditi e_2 i u sortiranoj listi.

6.1 Radix sort algoritam

POSTAVKA RADIX SORTIRANJA

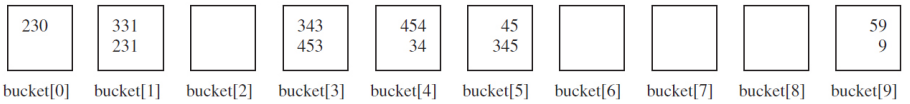
Radix sortiranje je specijalan slučaj segmentnog sortiranja, s tim što radix sort koristi samo deset kanti. Svodi se na ponavljanje segmentnog sortiranja za svaku od pozicija osnove broja

Radix sortiranje je specijalan slučaj segmentnog sortiranja.

Analizirajmo sledeći primer: Pretpostavimo da su ključevi sa kojima radimo pozitivni celi brojevi. Ideja radix sort-a je da se ključevi podele u podgrupe prema njihovoj brojnoj osnovi (radix poziciji), tj. jedinici, desetici, stotici broja, itd.... Ovo sortiranje se svodi na stalno ponavljanje segmentnog sortiranja za svaku od pozicija osnove broja, počevši od sortiranja prema jedinici broja. Analizirajmo sortiranje sledećeg niza:

331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9

Primenimo segmentno sortiranje tako što za ključ uzmemo jedinicu broja. Elementi će biti smešteni u kante na sledeći način (Slika- 1) :



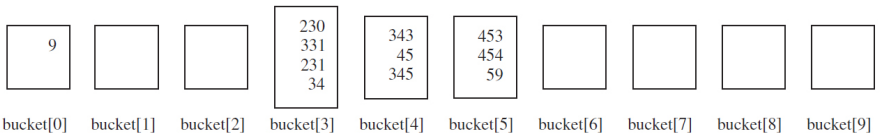
Slika 6.2.1 Sortiranje prema jedinici broja

Nakon toga vraćamo elemente u niz. Redosled nakon prvog koraka će biti:

230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 9

Sada primenimo segmentno sortiranje tako što za ključ uzmemo deseticu broja.

Elementi se u kante pakuju na sledeći način (Slika-2):

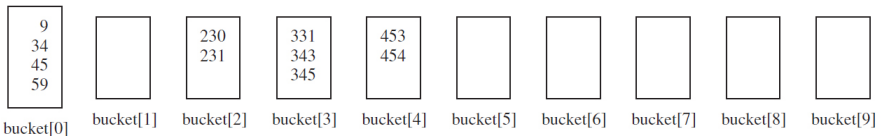


Slika 6.2.2 Sortiranje prema desetici broja

Ponovo vraćamo elemente u niz i imamo sledeći raspored:

9, 230, 331, 231, 34, 343, 45, 345, 453, 454, 59

Na kraju, ostalo je da primenimo segmentno sortiranje tako što za ključ uzmemo stoticu broja. Elementi se u kante pakuju prema Slici-3:



Slika 6.2.3 Sortiranje prema stotici broja

Konačno, iz kanti vraćamo elemente nazad u niz i dobijamo sledeći redosled (niz je sortiran):

9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454

PRIMERI RADIX SORTIRANJA

Radix sort se osim sortiranja celih brojeva veoma uspešno koristi kod sortiranja kraćih stringova i kodova

Primer 1. Sortiranje stringova

Radix sort prvo grupiše elemente podataka prema njihovom krajnje desnom karakteru i postavlja ove grupe u redosled u odnosu na krajnji desni karakter. Onda kombinuje te grupe.

Ponavlja se grupisanje i operacija spajanja za sve druge pozicije karaktera, od kranje desnih do krajnje levih pozicija karaktera.

Na kraju operacija sortiranje će biti završeno (Slika- 4) .

mom, dad, god, fat, bad, cat, mad, pat, bar, him	originalna lista
(dad,god,bad,mad) (mom,him) (bar) (fat,cat,pat)	grupisati po prvom s' desna
dad,god,bad,mad,mom,him,bar,fat,cat,pat	spajanje grupa
(dad,bad,mad,bar,fat,cat,pat) (him) (god,mom)	grupisati po srednjem slovu
dad,bad,mad,bar,fat,cat,pat,him,god,mom	spajanje grupa
(bad,bar) (cat) (dad) (fat) (god) (him) (mad,mom) (pat)	grupisati po prvom s' leva
bad,bar,cat,dad,fat,god,him,mad,mom,par	spajanje grupa (SORTIRANO)

Slika 6.2.4 Primer sortiranja stringova korišćenjem radix sort-a

U cilju boljeg uvida u sortiranje korišćenjem radix sort algoritma, predlaže se poseta sledećeg linka: http://www.cs.armstrong.edu/liang/animation/web/RadixSort.html.

Primer 2. Sortiranje kodova

U nastavku je dat još jedan primer sortiranja kodova korišćenjem radix sort-a (Slika-5):

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Originalni niz
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Grupisani po 4. cifri
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Spojeni
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grupisani po 3. cifri
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Spojeni
(0**0**04, 1**0**61) (0**1**23, 2**1**50, 2**1**54) (0**2**22, 0**2**83) (1**5**60)	Grupisani po 2. cifri
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Spojeni
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grupisani po 1. cifri
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Spojeni (sortirani)

Slika 6.2.5 Primer sortiranja kodova primenom radix sort-a

IMPLEMENTACIJA ALGORITMA

Implementacija se sastoji iz kreiranja 10 inicijalno praznih kofa koje se po potrebi popunjavaju elementima. U okviru jedne kofe elementi se mogu čuvati ili u povezanim listama ili u nizu

Pseudo-kod algortma radix sort-a može biti napisan na sledeći način:

```
radixSort(inout theArray:ItemArray, in n:integer, in d:integer)
// sort n d-digit integers in the array theArray
    for (j=d down to 1) {
        Initialize 10 groups to empty
        Initialize a counter for each group to 0
        for (i=0 through n-1) {
            k = jth digit of theArray[i]
            Place theArrar[i] at the end of group k
            Increase kth counter by 1
        }
        Replace the items in theArray with all the items in group 0,
        followed by all the items in group 1, and so on.
    }
```

Implementacija u jezika Java je data u nastavku. Za svaku kantu se kreira lista, tako da ćemo imati 10 listi tipa ArrayList.

```
public void radixsort(int[] input) {
    final int RADIX = 10;
    // declare and initialize bucket[]
    List<Integer>[] bucket = new ArrayList[RADIX];
    for (int i = 0; i < bucket.length; i++) {
        bucket[i] = new ArrayList<Integer>();
    }

    // sort
    boolean maxLength = false;
    int tmp = -1, placement = 1;
    while (!maxLength) {
        maxLength = true;
```

```
        // split input between lists
        for (Integer i : input) {
            tmp = i / placement;
            bucket[tmp % RADIX].add(i);
            if (maxLength &amp;amp;&amp;amp; tmp > 0) {
                maxLength = false;
            }
        }
        // empty lists into input array
        int a = 0;
        for (int b = 0; b < RADIX; b++) {
            for (Integer i : bucket[b]) {
                input[a++] = i;
            }
            bucket[b].clear();
        }
        // move to next digit
        placement *= RADIX;
    }
}
```

ANALIZA ALGORITMA

Ukoliko imamo n brojeva koji se sastoje iz maksimalno k cifara, radix sortiranje će se izvršiti za vreme O(n · k)

Radix sort je algoritam koji sortira kodove procesiranjem pojedinačnih cifara broja ili karaktera stringa. Ukoliko imamo *n* brojeva koji se sastoje iz maksimalno *k* cifara, radix sortiranje će se izvršiti za vreme O(n * k).

Radix sort algoritam zahteva *2*n*d* pomeranja za sortiranje *n* stringova svaki sa *d* karaktera.

Možemo usvojiti da je Radix Sort vremenske složenosti O(n) .

Mada je radix sort složenosti O(n) on nije pogodan kao algoritam sortiranja za opšte namene.

Zahtev za memoriju je *d* * original size of data (svaka grupa treba da bude dovoljno velika da sadrži originalni skup podataka). Na primer, treba 27 grupa za sortiranje stringa sa velikim slovima. Kod radix sort-a je pogodnije koristiti povezane liste (engl. linked list) nego nizove. U ovom slučaju nije potrebna velika memorija.

Radix sort može da procesira broj počevši ili od najmanje cifre (least significant digit- LSD), što je ustvari jedinica broja, ili počevši od najveće cifre (most significant digit- MSD).

U lekciji L13 će detaljnije biti opisani LSD i MSD radix sort algortmi.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Prikazati postupak radix sortiranja na sledećim brojevima 454, 34, 23, 43, 74, 86, and 76.

Zadatak 2. (Radix sort) Napisati program koji nasumično generiše niz od 1,000,000 celih brojeva i sortira ih korišćenjem radix sort algoritma.

Zadatak 3. Napisati program koji će dati vreme izvršavanja sledećih algoritama sortiranja: selection sort, bubble sort, merge sort, quick sort, heap sort, and radix sort na slučajno generisanim nizovima dužine 50,000, 100,000, 150,000, 200,000, 250,000 i 300,000.

Poglavlje 7

Spoljašnje sortiranje – extern sort

UVOD U SPOLJAŠNJE SORTIRANJE. PRVA FAZA SORTIRANJA

Za sortiranje ogromnih podataka iz spoljašnjih fajlova treba prvo preneti deo podataka u memoriju, zatim ih interno sortirati i na kraju odštampati sortirane podatke u privremene fajlove

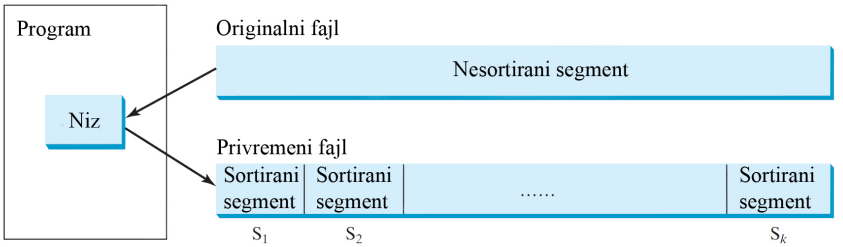
Svi prethodno pomenuti algoritmi sortiranja pretpostavljaju da su svi podaci, koji treba da budu sortirani, istovemeno smešteni u internoj memoriji, u odgovarajućem nizu. Za sortiranje podataka u spoljašnjim datotekama (obično na hard disku), treba prvo preneti podatke u memoriju, a zatim ih interno sortirati. Međutim, ako je datoteka prevelika, svi podaci u datoteci se ne mogu odmah dovesti u memoriju.

Rešenje je korišćenje spoljašnjeg sortiranja – extern sort.

Spoljašnje sortiranje se sastoji iz dve faze.

Faza I

Uzastopno prebacivanje dela podataka iz datoteke u niz, a zatim uređivanje niza korišćenjem internog algoritma sortiranja, i na kraju prenošenje podataka iz niza u privremenu datoteku (Slika- 1).



Slika 7.1 Sortiranje originalnog niza u segmentima – Uzastopno prebacivanje dela podataka iz datoteke u niz, sortiranje niza korišćenjem internog algoritma i štampanje sortiranog niza u privremenu datoteku

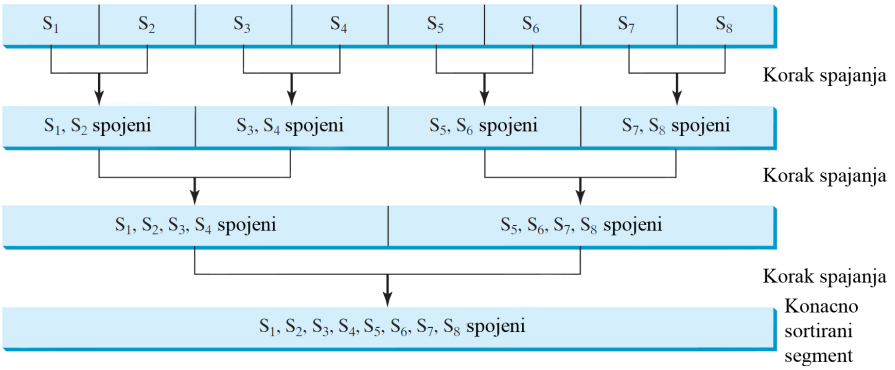
Pretpostavimo da niz može maksimalno da primi 100 000 celobrojnih vrednosti koje trebaju biti sortirane. nakon sortiranja kreiraju se fajlovi S_1, S_2, ..., S_k, gde svaki od fajlova sadrži 100 000 sortiranih vrednosti, a poslednji fajl S_k može da ima i manje od 100 000 vrednosti.

DRUGA FAZA SPOLJAŠNJEG SORTIRANJA

U drugoj fazi treba uzeti par sortiranih segmenata iz privremenih datoteka, spojiti ih i ponovo zapisati u neku privremenu datoteku

Faza II

Spojiti par sortiranih segmenata (npr., S_1 sa S_2, S_3 sa S_4, ..., itd) u veći sortirani segment i sačuvati novi segment u novu privremenu datoteku. Nastaviti isti proces sve dok se ne generiše jedan sortirani segment (Slika-2) .

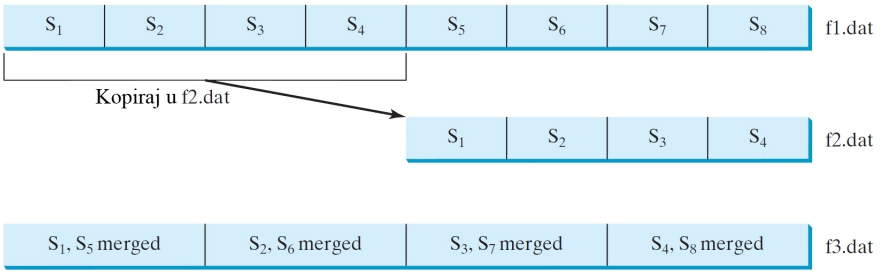


Slika 7.2 Sortirani segmenti se iterativno spajaju, najčešće korišćenjem metode spajanja preuzete iz merge sort algoritma

Implementacija Faze II

U svakom koraku, spajaju se dva sortirana segmenta u cilju kreiranja novog segmenta. Veličina novog segmenta je duplirana. Tako je broj segmenata smanjen za pola nakon svakog koraka spajanja.

Segment je suviše veliki da bi bio doveden u memoriju. Da bi se implementirao korak spajanja, treba kopirati polovinu segmenata iz datoteke *f1.dat* u privremenu datoteku*f2.dat*. Onda spojiti prvi dostupni segment iz *f1.dat* sa prvim segmentom iz *f2.dat* u privremenu datoteku *f3.dat* (Slika-3).



Slika 7.3 Implementacija Faze II. Spajanje delova fajlova

Implementacija algoritma spoljašnjeg sortiranja je obrađena u okviru materijala za vežbe.

ANALIZA SPOLJAŠNJEG SORTIRANJA

Broj ulazno/izlaznih operacija u svakom koraku spajanja je O(n). Ukupan broj koraka spajanja je log (n/c). Stoga, složenost algoritma spoljašnjeg sortiranja je O(n logn)

Najveći troškovi se kod spoljašnjeg sortiranja odnose na procese učitavanja iz datoteke i upisivanja u dateteke (I/O troškovi). Pretpostavimo da imamo n elemenata u datoteci. U **Fazi I** je n elemenata niza učitano iz datoteke, a zatim i upisano u privremene datoteke. Stoga, ulazno/izlazni troškovi **Faze I** su $O(n)$.

U **Fazi II**, pre prvog koraka spajanja, broj sortiranih segmenata je n/c, gde je c jednako maksimalnoj veličini niza (MAX_ARRAY_SIZE). U svakom koraku spajanja broj segmenata se smanji za polovinu. Stoga je nakon prvog koraka spajanja broj segmenata jednak n/2c . Nakon drugog spajanja broj segmenata je $n/2^2c$, a nakon trećeg spajanja broj segmenata je $n/2^3c$. Nakon log(n/c) koraka spajanja, broj segmenata će biti 1. Stoga će ukupan broj koraka spajanja biti log(n/c) .

U svakom koraku spajanja se polovina segmenata čita iz datoteke *f1*, a zatim upisuje u privremenu datoteku *f2*. Ostatak segmenata iz *f1* se spaja sa segmentima iz *f2*. Broj ulazno/izlaznih operacija u svakom koraku spajanja je O(n). Pošto je ukupan broj koraka spajanja log(n/c) stoga je ukupan broj ulazno/izlaznih operacija:

O(n)x log(n/c)=O(nlogn)

Stoga, vremenska složenost algoritma spoljašnjeg sortiranja je O(n logn).

ZADATAK: IMPLEMENTACIJA EXTERN SORT-A

Cilj zadatka je implementacija algoritma spoljašnjeg sortiranja koji smo obradili na predavanjima

Kreiranje ogromnog spoljašnjeg fajla:

```
import java.io.*;

public class CreateLargeFile {
  public static void main(String[] args) throws Exception {
    DataOutputStream output = new DataOutputStream(
    new BufferedOutputStream(
    new FileOutputStream("largedata.dat")));

    for (int i = 0; i < 800004; i++)
    output.writeInt((int)(Math.random() * 1000000));

    output.close();
```

```
    // Display first 100 numbers
    DataInputStream input =
    new DataInputStream(new FileInputStream("largedata.dat"));
    for (int i = 0; i < 100; i++)
    System.out.print(input.readInt() + " ");

    input.close();
  }
}
```

U nastavku je dat kod za spoljašnje sortiranje:

```
import java.io.*;

public class SortLargeFile {
  public static final int MAX_ARRAY_SIZE = 43;
  public static final int BUFFER_SIZE = 100000;

  public static void main(String[] args) throws Exception {
    // Sort largedata.dat to sortedfile.dat
    sort("largedata.dat", "sortedfile.dat");

    // Display the first 100 numbers in the sorted file
    displayFile("sortedfile.dat");
  }

  /** Sort data in source file and into target file */
  public static void sort(String sourcefile, String targetfile)
    throws Exception {
    // Implement Phase 1: Create initial segments
    int numberOfSegments =
    initializeSegments(MAX_ARRAY_SIZE, sourcefile, "f1.dat");

    // Implement Phase 2: Merge segments recursively
    merge(numberOfSegments, MAX_ARRAY_SIZE,
    "f1.dat", "f2.dat", "f3.dat", targetfile);
  }

  /** Sort original file into sorted segments */
  private static int initializeSegments
    (int segmentSize, String originalFile, String f1)
    throws Exception {
    int[] list = new int[segmentSize];
    DataInputStream input = new DataInputStream(
    new BufferedInputStream(new FileInputStream(originalFile)));
    DataOutputStream output = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream(f1)));

    int numberOfSegments = 0;
    while (input.available() > 0) {
    numberOfSegments++;
    int i = 0;
    for ( ; input.available() > 0 &amp;amp;&amp;amp; i < segmentSize; i++) {
```

```
        list[i] = input.readInt();
    }

    // Sort an array list[0..i-1]
    java.util.Arrays.sort(list, 0, i);

    // Write the array to f1.dat
    for (int j = 0; j < i; j++) {
        output.writeInt(list[j]);
    }
    }

    input.close();
    output.close();
    return numberOfSegments;
  }

  private static void merge(int numberOfSegments, int segmentSize,
    String f1, String f2, String f3, String targetfile)
    throws Exception {
    if (numberOfSegments > 1) {
    mergeOneStep(numberOfSegments, segmentSize, f1, f2, f3);
    merge((numberOfSegments + 1) / 2, segmentSize * 2,
        f3, f1, f2, targetfile);
    }
    else { // Rename f1 as the final sorted file
    File sortedFile = new File(targetfile);
    if (sortedFile.exists()) sortedFile.delete();
    new File(f1).renameTo(sortedFile);
    }
  }

  private static void mergeOneStep(int numberOfSegments,
    int segmentSize, String f1, String f2, String f3)
    throws Exception {
    DataInputStream f1Input = new DataInputStream(
    new BufferedInputStream(new FileInputStream(f1), BUFFER_SIZE));
    DataOutputStream f2Output = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream(f2), BUFFER_SIZE));

    // Copy half number of segments from f1.dat to f2.dat
    copyHalfToF2(numberOfSegments, segmentSize, f1Input, f2Output);
    f2Output.close();

    // Merge remaining segments in f1 with segments in f2 into f3
    DataInputStream f2Input = new DataInputStream(
    new BufferedInputStream(new FileInputStream(f2), BUFFER_SIZE));
    DataOutputStream f3Output = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream(f3), BUFFER_SIZE));

    mergeSegments(numberOfSegments / 2,
    segmentSize, f1Input, f2Input, f3Output);
```

```
  f1Input.close();
  f2Input.close();
  f3Output.close();
}


/** Copy first half number of segments from f1.dat to f2.dat */
private static void copyHalfToF2(int numberOfSegments,
  int segmentSize, DataInputStream f1, DataOutputStream f2)
  throws Exception {
  for (int i = 0; i < (numberOfSegments / 2) * segmentSize; i++) {
  f2.writeInt(f1.readInt());
  }
}

/** Merge all segments */
private static void mergeSegments(int numberOfSegments,
  int segmentSize, DataInputStream f1, DataInputStream f2,
  DataOutputStream f3) throws Exception {
  for (int i = 0; i < numberOfSegments; i++) {
  mergeTwoSegments(segmentSize, f1, f2, f3);
  }

  // If f1 has one extra segment, copy it to f3
  while (f1.available() > 0) {
  f3.writeInt(f1.readInt());
  }
}

/** Merges two segments */
private static void mergeTwoSegments(int segmentSize,
  DataInputStream f1, DataInputStream f2,
  DataOutputStream f3) throws Exception {
  int intFromF1 = f1.readInt();
  int intFromF2 = f2.readInt();
  int f1Count = 1;
  int f2Count = 1;

  while (true) {
  if (intFromF1 < intFromF2) {
      f3.writeInt(intFromF1);
      if (f1.available() == 0 || f1Count++ >= segmentSize) {
      f3.writeInt(intFromF2);
      break;
      }
      else {
      intFromF1 = f1.readInt();
      }
  }
  else {
      f3.writeInt(intFromF2);
      if (f2.available() == 0 || f2Count++ >= segmentSize) {
      f3.writeInt(intFromF1);
```

```
        break;
        }
        else {
        intFromF2 = f2.readInt();
        }
    }
    }

    while (f1.available() > 0 &amp;amp;&amp;amp; f1Count++ < segmentSize) {
    f3.writeInt(f1.readInt());
    }

    while (f2.available() > 0 &amp;amp;&amp;amp; f2Count++ < segmentSize) {
    f3.writeInt(f2.readInt());
    }
  }

  /** Display the first 100 numbers in the specified file */
  public static void displayFile(String filename) {
    try {
    DataInputStream input =
        new DataInputStream(new FileInputStream(filename));
    for (int i = 0; i < 100; i++)
        System.out.print(input.readInt() + " ");
    input.close();
    }
    catch (IOException ex) {
    ex.printStackTrace();
    }
  }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Deset brojeva {2, 3, 4, 0, 5, 6, 7, 9, 8, 1} se čuva u spoljašnjem tekstualnom fajlu *podaci.dat.* Pratiti izvršavanje *SortLargeFile* programa ručno korak po korak za MAX_ARRAY_SIZE = 2.

Zadatak 2. Napisati program koji će prikazati vreme izvršavanja algoritma spoljašnjeg sortiranja za datoteku sa slučajno generisanim brojevima veličine 5,000,000, 10,000,000, 15,000,000, 20,000,000, 25,000,000 i 30,000,000 brojeva.

Zaključak

REZIME

Na osnovu svega obrađenog možemo zaključiti sledeće:

Sortiranje je proces preuređivanja elemenata nekog skupa u određenom poretku. Algoritmi sortiranja se dele na: algoritme sortiranja poređenjem (poređenje elemenata prema nekom ključu) i na one bez poređenja. Algoritmi sortiranja bez poređenja mogu dostići linearnu složenost.

Metoda selekcije (selection sort) je najprostiji ili i najsporiji algoritam sortiranja. Spada u algoritme kvadratne složenosti.

Sortiranje mehurom (bubble sort) ispituje niz od početka do kraja, upoređujući elemente u toku prolaza, pri čemu se vrši isplivavanje najlakšeg mehura na jednu stranu niza. Spada u grupu algoritama kvadratne složenosti.

U okviru metoda sortiranja umetanjem dve najpoznatije metode su: metoda direktnog umetanja (insertion sort) i metoda sortiranja po Šelu (Shell sort). Oba algoritma su kvadratne složenosti.

Sortiranje spajanjem ili merge sort je rekurzivni algoritam sortiranja zasnovan na poređenju. Predstavlja primer algoritamske strategije "podeli i osvoji“. U ovu grupu još spadaju i algoritmi hip sort i quick sort.

Ideja quick sort algoritma sastoji se u podeli niza prema odabranom elementu za podelu koji se dovodi na pravo mesto, i u primeni algoritma brzog sortiranja na svaku od dva dobijena podniza.

Heap sort funkcioniše po principu hip stabla, ali se naime njegovi elementi pakuju u niz.

Odlika hipa je da se za svaki čvor tačno znaju indeksi njegovog roditelja odnosno indeksi levog i desnog potomka, što olakšava sortiranje podataka.

Segmentno sortiranje je algoritam za sortiranje koji deli niz na određeni broj kanti. Svaka kanta se onda sortira pojedinačno. Radix sortiranje je specijalan slučaj segmentnog sortiranja, s tim što radix sort koristi samo deset kanti.

Spoljašnje sortiranje je efikasan algoritam koji sortira ogroman broj podataka koji se čuvaju na spoljačnjem medijumu, zbog nemogućnosti da se ceo niz smesti u RAM memoriju.

Slika-1 daje poređenje algoritama sortiranja.

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	n * log n	n * log n
Quicksort	n^2	n * log n
Radix sort	n	n
Treesort	n^2	n * log n
Heapsort	n * log n	n * log n

Slika-1 Poređenje algoritama prema najboljem i najgorem slučaju izvršavanja

REFERENCE

Korišćena literatura

[1] D. Liang, Introduction to Java Programming, Comprehensive Version, deseto izdanje, Prentice Hall, 2014

[2] M.A. Weiss, Data Structures and Problem Solving Using Java, treće izdanje, Addison Wesley, 2005.

[3] Nenad Filipovic, Algoritmi i strukture podataka, Masinski fakultet u Kragujevcu, 2010.

[4] R. Sedgewick, K.Wayne, Algorithms, 4. Izdanje, Pearson Education, 2011.

[5] https://en.wikipedia.org/wiki/Insertion_sort

[6] https://en.wikipedia.org/wiki/Bubble_sort

[7] https://en.wikipedia.org/wiki/Quicksort

[8] https://en.wikipedia.org/wiki/Merge_sort