



Милош Ђекић

ФОН, Београд

# Алгоритми и Структуре података

- са примерима у програмском језику Јава -

Август 2007

# **Алгоритми и Структуре података**

**- скрипта -**



# САДРЖАЈ

<b>САДРЖАЈ</b>	<b>3</b>
<b>УВОД</b>	<b>5</b>
<b>ЛИНЕАРНЕ СТРУКТУРЕ ПОДАТАКА</b>	<b>6</b>
<b>НИЗОВИ</b>	<b>6</b>
<b>ВЕКТОРИ</b>	<b>6</b>
<b>ЛИСТЕ</b>	<b>7</b>
Уланчане листе	7
Кружне листе	8
Двоструко уланчане листе	9
<b>ЗАДАЦИ #1 (Листе)</b>	<b>10</b>
<b>СТЕКОВИ</b>	<b>10</b>
<b>РЕДОВИ</b>	<b>12</b>
Секвенцијална реперезентација реда	12
Уланчана реперезентација реда	13
Приоритетни ред	13
<b>ЗАДАЦИ #2 (Линеарне Структуре)</b>	<b>14</b>
<b>НЕЛИНЕАРНЕ СТРУКТУРЕ ПОДАТАКА</b>	<b>16</b>
<b>СТАБЛА</b>	<b>16</b>
Уланчана реперезентација стабла	17
Секвенцијална реперезентација стабла	17
Бинарна стабла	18
Начини обиласка бинарног стабла	19
Обилажење бинарног стабла - рекурзивно	20
Стабла вишег реда (m)	21
<b>ЗАДАЦИ #3 (Стабла)</b>	<b>21</b>
<b>АЛГОРИТМИ ПРЕТРАЖИВАЊА</b>	<b>25</b>
<b>СЕКВЕНЦИЈАЛНО ПРЕТРАЖИВАЊЕ</b>	<b>25</b>
Пребацивање на почетак	26
Транспозиција	26
Претраживање сортирање датотеке	26
<b>БИНАРНО ПРЕТРАЖИВАЊЕ</b>	<b>27</b>
Интерполационо претраживање	27
Робусно интерполационо претраживање - Fast Search	28
Индекс-секвенцијално претраживање	28
<b>СТАБЛА ПРЕТРАЖИВАЊА</b>	<b>29</b>
<b>БСТ СТАБЛА</b>	<b>29</b>
Уметање чвора у BST стабло	29
Брисање чвора из BST стабла	30
<b>АВЛ СТАБЛА</b>	<b>31</b>
Уметање чвора у AVL стабло	31
<b>СТАБЛА ОПШТЕГ ПРЕТРАЖИВАЊА - ВСТ</b>	<b>33</b>
<b>Б СТАБЛА</b>	<b>35</b>
<b>Б* СТАБЛА</b>	<b>37</b>
<b>Б+ СТАБЛА</b>	<b>38</b>
<b>ЗАДАЦИ #4 (Стабла претраживања)</b>	<b>40</b>

<b>ХЕШИРАЊЕ - Hashing</b>	<b>42</b>
<b>НЕЗАВИСНЕ ХЕШ ФУНКЦИЈЕ</b>	<b>43</b>
Метод дељења	43
Метод множења (Мултипликативна хеш функција)	43
Метод средине квадрата	43
Метод склапања	43
Метод конверзије основе	43
Метод алгебарског кодовања	44
<b>ЗАВИСНЕ ХЕШ ФУНКЦИЈЕ</b>	<b>44</b>
Метод анализе цифара	44
Директна кумулативна функција расподеле	44
Апроксимација - Сегментна линеарна функција	44
<b>РАЗРЕШАВАЊЕ КОЛИЗИЈА</b>	<b>45</b>
Отворено адресирање	45
Уланчавање	46
<b>СОРТИРАЊЕ</b>	<b>47</b>
<b>МЕТОДЕ УМЕТАЊА</b>	<b>47</b>
Директно уметање - Insertion Sort	47
Shell Sort	48
<b>МЕТОДЕ СЕЛЕКЦИЈЕ</b>	<b>48</b>
Директна селекција - Selection Sort	48
Сортирање помоћу BST	49
HEAP Sort	49
<b>МЕТОДЕ ЗАМЕНЕ</b>	<b>50</b>
Bubble Sort	50
Quick Sort	50
Побитно раздвајање	50
<b>МЕТОДЕ СПАЈАЊА</b>	<b>51</b>
<b>МЕТОДЕ ЛИНЕАРНЕ СЛОЖЕНОСТИ</b>	<b>51</b>
Counting Sort	51
Radix Sort	51
<b>ИЗВОРИ - ЛИТЕРАТУРА - НАПОМЕНЕ</b>	<b>52</b>
<b>ДОДАТАК - РАЗНИ ЗАДАЦИ</b>	<b>53</b>



## УВОД

ALGORITHMS + DATA STRUCTURES = PROGRAMS

опис начина обраде података

опис начина организације података

Алгоритми имају један или више улаза и један или више излаза. Морају бити недвосмислени и оствариви, тј. да имају коначно време извршења. Неки алгоритми су *дејтерминистички* (уvek дају исти резултат), док постоје и *псеудоалгоритми* (не дају увек исти резултат - *генератори*). Алгоритми се најчешће пишу у псеудојезику, па се после имплементирају у одговарајућем програмском језику.

**ЗАПИС** - логичка целина, састављена од елемената различитог типа (поља). Најчешће се користи у имплементацији динамичких структура и датотека. Операције се најчешће врше над пољима.

Запис у меморијској локацији може бити смештен на 2 начина:

- 1) *континуална алокација поља*
  - штеди се меморија на рачун времена, тако што се делови записа слажу у меморији један за другим, без остављања празног простора (неки податак може почињати у средини неке речи и наставити се у следећу реч).
- 2) *везивањем поља за почетак речи*
  - штеди се време на рачун меморије, тако што сваки део записа почиње на почетку речи, тј. ако се не заврши на крају, тај простор до краја се оставља празним.

Подела структура:

- 1) *по релацијама елемената*
  - **линеарне** (1 елемент у вези са још највише 2)
  - **нелинеарне** (1 елемент може бити у вези са више елемената)
- 2) *по могућности промене величине*
  - **статичке** (низови, са претходно дефинисаним бројем елемената)
  - **динамичке** (показивачи - референце)
- 3) *по месту чувања*
  - **унутрашње** (у оперативној меморији)
  - **спољашње** (у датотекама)

Меморијске репрезентације:

- 1) *секвенцијални*
  - физички и логички поредак исти
  - приступ елементима директан
  - обично се користи за линеарне структуре
- 2) *уланчани*
  - физички и логички поредак различити
  - приступ елементу индиректан
  - обично се користи за нелинеарне структуре
  - често се користи и показивачки механизам

## ЛИНЕАРНЕ СТРУКТУРЕ ПОДАТАКА

Линеарна структура може бити:

- *секвенцијална* - низ (вектор)
- *уланчана* - уланчана листа

Својство линеарних листи је једнодимензионалност. Неке од операција које се над њима врше су: обилазак по поретку, претраживање, приступ елементу, уметање, брисање...

### НИЗОВИ

Низови су хомогени типови, у којима сваки елемент има своје место у низу (индекс) и приступа му се директно. Једнодимензионални низови су *вектори*, дводимензионални су *матрице*, док постоје и вишедимензионални низови.

Као пример може се навести меморија рачунара. Она је, у ствари, један једнодимензионални низ меморијских локација. За сваку меморијску локацију у меморији се зна које су претходна и наредна меморијска локација. Уколико се зада неки низ, он ће, у меморији, имати тачно дефинисану локацију, коју ће одређивати управо претходна и следећа локација. Тако, када је програмеру потребно да приступи неком елементу низа, а позната му је његова локација у низу, он приступа адреси низа, која је, у ствари, почетна адреса низа у меморији (адреса првог податка). Затим, од првог елемента низа броји помераје, тако да број помераја буде једнак индексу који му је познат:

```
System.out.println(niz[1]);
```

адреса низа +  $n$  \* величина типа податка

По меморији се креће померајима величине бајта (на пр.), а по низу се креће померајима величине типа података низа.

Низови се, као тип податка, користе када је јако битна брзина приступа податку, а јако ретко се врши уметање или брисање елемената (изузетак је када се брисање врши са почетка или краја). У меморији су сви подаци на једном месту, низ је беспрекорно запакован скуп података са јединственом локацијом.

### ВЕКТОРИ

У меморији, вектори се представљају тако што се не раздваја логички концепт од имплементације. Смештање вектора у меморијске локације се може извршити на више начина:

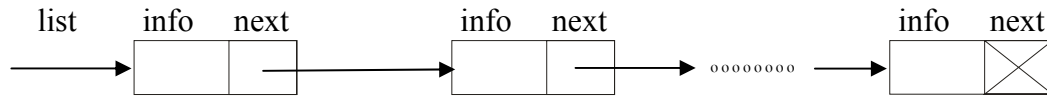
- 1) *континуирано* (фино запаковано, без остављања празног простора)
  - просторно ефикасно, али неефикасан приступ
- 2) *са допунвањем* ("**padding**" - ако један вектор заузима више од једне меморијске локације, оставља се празан простор до првог следећег празног меморијског места)
  - ефикаснији приступ, али просторно неефикасно

Понекад се деси да се више вектора може сместити у једну реч (меморијску локацију). То се може урадити само ако једна реч има простора да у њу стану барем две целе речи или више целих речи.

# ЛИСТЕ

## Уланчане листе

Код уланчаних листа, процеси уметања и брисања елемената су релативно лаки (знатно брже него код низова). Користи се показивачки механизам, а сваки елемент листе се назива **чвор**. Уланчана листа је уланчана имплементација линеарне листе, где сваки чвор има свог претходника и свог следбеника (осим првог и последњег чвора).



Листе могу бити:

- 1) *по организацији*
  - уређене (реастуће, опадајуће)
  - неуређене
- 2) *по начину повезаности:*
  - једноструко уланчане
  - двоструко уланчане
  - кружне (последњи чвор показује на први)
  - некружне

Једноструко уланчана листа:

```
public class Lista {  
    private Element first, last; //pokazivači na prvi i poslednji element liste  
    private int br; //broj članova liste  
  
    public Lista() { first = last = null; br = 0; }  
}
```

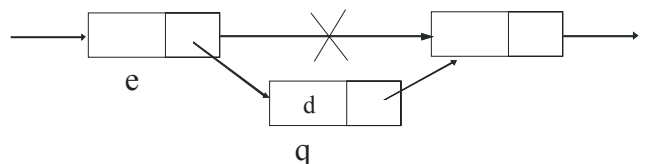
Елемент листе:

```
public class Element {  
    private int data; //podatak, koji čuva Element  
    private Element next; //pokazivač na sledeći element u listi  
  
    public Element(int x) { data = x; next = null; } //KONSTRUKTOR 1  
    public Element() {} //NO-ARG KONSTRUKTOR  
    public Element(Element e) { data=e.getData(); next=e.getNext(); } //KONSTRUKTOR 2  
  
    public int getData() { return data; }  
    public Element getNext() { return next; }  
  
    public void setNext(Element e) { next = e; }  
    public void setData(int d) { data = d; }  
}
```

Операције над једноструко уланчаном листом:

- 1) Уметање иза задатог чвора:

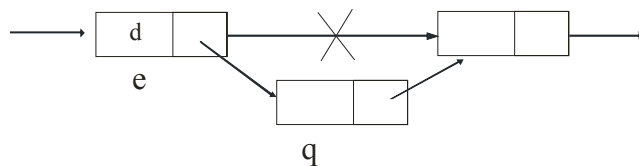
```
public void Insert_After(Element e, int d) {  
    Element q = new Element();  
    q.setData(d);  
    q.setNext(e.getNext());  
    e.setNext(q);  
}
```





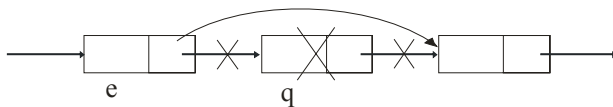
## 2) Уметање испред задатог чвора:

```
public void Insert_Before(Element e, int d) {
    Element q = new Element();
    q.setNext(e.getNext());
    q.setData(e.getData());
    e.setData(d);
    e.setNext(q);
}
```



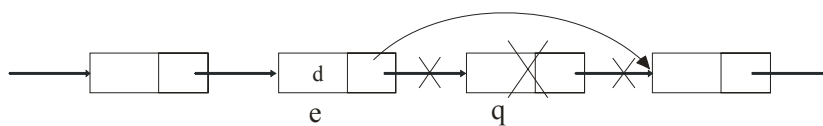
## 3) Брисање иза задатог чвора:

```
public void Delete_After(Element e) {
    Element q = e.getNext();
    e.setNext(q.getNext());
    q = null;
}
```



## 4) Брисање задатог чвора:

```
public void Delete(Element e) {
    Element q = e.getNext();
    e.setNext(q.getNext());
    e.setData(q.getData());
    q = null;
}
```



## 5) Претраживање неуређене листе:

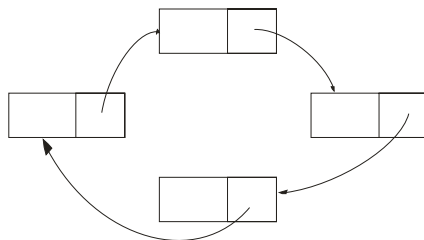
```
public Element Search(int d) {
    Element e = this.first;
    while (e != null && e.getData() != d) e = e.getNext();
    return e;
}
```

## 6) Претраживање уређене листе:

```
public Element Search_ord(int d) {
    Element e = this.first;
    while (e != null && e.getData() != d) e = e.getNext();
    if (e != null && e.getData() > d) e = null;
    return e;
}
```

## Кружне листе

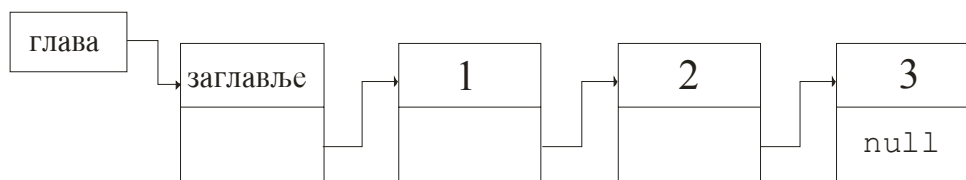
То су такве листе, у којима последњи чвор листе има показивач на први чвор листе. Обично се узима да спољашњи показивач на листу показује на последњи елемент, јер је тако лакше и брже убацивање елемената на почетак и крај листе.



### Алгоритам за убацивање елемента после неког елемента e:

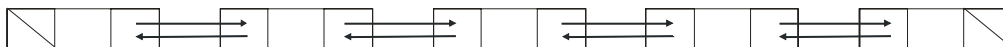
```
public void InsertAfter(Element e, int d) {  
    Element q = new Element();  
    q.setData(d);  
    q.setNext(e.getNext());  
    e.setNext(q);  
    if (this.first == e) this.first = q;  
}
```

**Заглавље листе** је посебан чвор који указује на први елемент листе и обично, поред показивача, садржи неку информацију о самој листи. Листа која има заглавље никада није празна.



### Двоструко уланчане листе

То су листе у којима сваки елемент показује и на свог претходника и на свог следбеника (осим првог и последњег, уколико листа уједно није и кружна). Предност је лако кретање по листи.

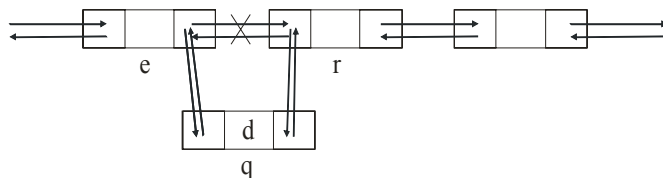


За двоструко уланчане листе увек важи релација:

```
(e.getPrev()).getNext() = e = (e.getNext()).getPrev();
```

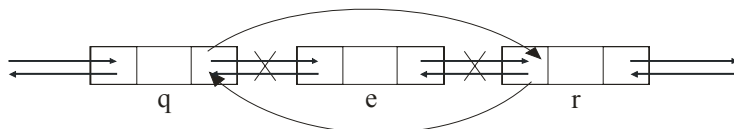
#### 1) Уметање чвора:

```
public void Insert_After(Element e, int d) {  
    Element q = new Element();  
    q.setData(d);  
    Element r = e.getNext();  
    q.setPrev(e);  
    q.setNext(r);  
    e.setNext(q);  
    r.setPrev(q);  
}
```



#### 2) Брисање чвора:

```
public void Delete(Element e) {  
    Element q = e.getPrev();  
    Element r = e.getNext();  
    q.setNext(r);  
    r.setPrev(q);  
    e = null;  
}
```



## ЗАДАЦИ #1 (Листе)

1) Реализовати операцију **Invert()**, која мења поредак чворова у листи, тако да први чвор постане последњи, други постане предпоследњи, итд.

```
public void Invert() {
    Element p = this.first;
    Element q = null;
    Element r; // čuva referencu
    while (p != null) {
        r = q; q = p;
        p = p.getNext();
        q.setNext(r);
    }
    this.first = q;
}
```

2) Реализовати операцију **Concatenate(Lista l)**, која надовезује листу на листу ако су:

- а) листе једноструко уланчане
- б) листе кружно уланчане (последњи елемент показује на први елемент листе)

а)

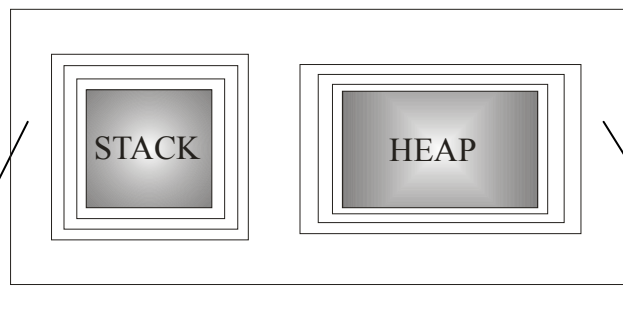
```
public Lista Concatenate(Lista l) {
    Element e = this.first;
    while (e.getNext() != null) e = e.getNext();
    e.setNext(l.first);
    return this;
}
```

б)

```
public Lista Concatenate (Lista l) {
    this.last.setNext(l.first);
    l.last.setNext(this.first);
    return this;
}
```

## СТЕКОВИ

Стекови су линеарне листе са **LIFO** системом приступа (**Last In First Out**). Једини део стека коме се приступа је врх стека (крај), на који се елементи стека слажу и одакле се узимају.



- пре свега се складиште статички подаци;
- складиштење на врх стека, по LIFO принципу;
- никада не сме доћи до тога да се са стека узме више него што је претходно стављено (дошло би до појаве изузетка - грешке, која се зове **Underflow** - *попкорачење*).

- део меморије на којем се динамички алоцирају подаци;
- мањи део меморије (Garbage Collector у Јави ће брже одрадiti свој посао на мањем делу меморије);
- може бити потпуно "разбијен" у меморији (рачунар зна који су положаји у питању);
- HEAP је оптималне величине за дати програм

Када је реч о секвенцијалној репрезентацији стека, говори се о једном, два или више стекова који се смештају у један низ. Постоји показивач на врх стека (`stek.top`), а садржај стека су елементи низа:

```
public class Stek {  
  
    private Object[] stek; //niz koji sadrži stek  
    private int top; //pokazuje na vrh steka (stack-pointer)  
    private int broj; //broj elemenata steka  
  
    public Stek(int broj) {  
        this.broj = broj;  
        top = 0;  
        stek = new Object[broj];  
    }  
}
```

#### 1) Провера да ли је стек празан:

```
public boolean Stack_Empty() {  
    if (top == 0) return true;  
    else return false;  
}
```

#### 2) Уметање у стек:

```
public Stek Push(Object o) {  
    if (top == broj) {  
        System.out.println("Overflow"); //Niz nije dovoljno veliki  
        return;  
    }  
    else {  
        top += 1;  
        stek[top] = o;  
    }  
    return this;  
}
```

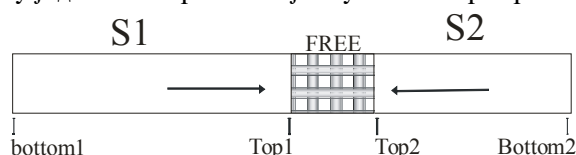
#### 3) Уклањање са стека:

```
public Object Pop() {  
    if (Stack_Empty()) { System.out.println("Underflow"); return; } //Stek je prazan  
    else return stek[top--];  
    return null;  
}
```

#### 4) Читање без уклањања:

```
public Object Top() {  
    if (Stack_Empty()) { System.out.println("Underflow"); return; }  
    else return stek[top];  
    return null;  
}
```

Често се користи имплементација 2 стека у један вектор. Стекови расту један према другом и по потреби узимају слободан простор, који се налази између њих (врши се релокација простора између стекова). Могућа је имплементација више стекова у један вектор. Постоји и уланчана репрезентација стека (први чвор је врх).



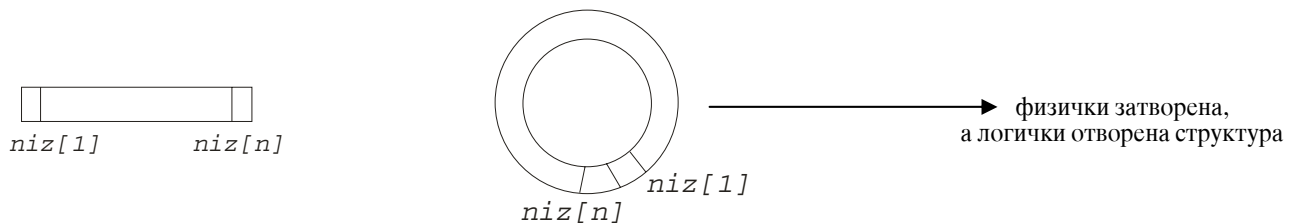
## РЕДОВИ

Редови су линеарне листе са **FIFO** режимом приступа подацима (**First In First Out**). Имају два краја - **чело** и **зачеље**. То су динамичке и хомогене структуре (елементи су истог типа). Као и стекови, могу се представљати секвенцијално и уланчано.



### Секвенцијална реперезентација реда

Ред се секвенцијално представља у облику низа ( $niz[1, n]$ ), који садржи показиваче **head** и **tail**. Када су **head** и **tail** једнаки нули, ред је празан. Да би, након уклањања елемента из реда, тај простор био поново доступан за неки нови елемент, користи се кружни бафер (Bounded Buffer) за представљање реда, у коме су први и последњи елемент низа спојени.



### Имплементација реда:

```
public class Red {  
    private Object[] bafer; //niz, sekvencijalna implementacija reda (Bounded Buffer)  
    private int head, tail;  
    private int kap; //ukupni kapacitet bafera (niza)  
    private int tmp; //trenutni kapacitet bafera (niza)  
  
    public Red(int kap) { this.kap = kap; bafer = new Object[kap]; }  
}
```

### Убацавање елемента у ред:

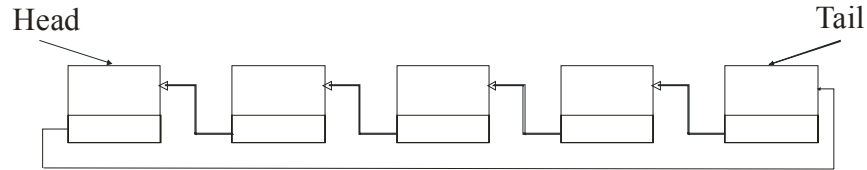
```
public void append(Object o) {  
    if(kap == tmp) {  
        System.out.println("Bafer je pun");  
        return;  
    }  
    bafer[(head++) % kap] = o; //% je kao mod u Paskalu (11mod10 = 1)  
    tmp++;  
}
```

### Узимање елемента из реда:

```
public Object take() {  
    if(tmp == 0) {  
        System.out.println("Bafer je prazan!");  
        return null;  
    }  
    tmp--;  
    return bafer[(tail++) % kap];  
}
```

## Уланчана реперезентација реда

Ред се уланчано може представити преко једноструко уланчане листе и преко кружне листе. У случају кружне листе, **head** показује на први елемент листе, а **tail** на последњи.



### Имплементација реда:

```
public class URed {  
    private Element head, tail;  
    public URed() {  
        head = null;  
        tail = null;  
    }  
}
```

### Убацивање елемента у ред:

```
public void InsertLast(int data) {  
    Element tmp = new Element(data);  
    tmp.setNext(null); // zato što će to biti poslednji element u listi  
    if (this.tail == null) head = tmp;  
    else this.tail.setNext(tmp);  
    this.tail = tmp;  
}
```

### Узимање елемента из реда:

```
public int TakeFirst() {  
    Element tmp = null;  
    int data = 0;  
    if (this.head == null) {  
        System.out.println("Underflow");  
        return data;  
    }  
    else {  
        tmp = this.head;  
        this.head = tmp.getNext();  
        data = tmp.getData();  
        tmp = null;  
    }  
    return data;  
}
```

## Приоритетни ред

Код ове врсте имплементације реда поредак у реду није одређен временом уласка у ред, већ по неком другом критеријуму приоритета. Поредак уметања је значајан само код елемената са истим приоритетом, а сам приоритетни ред се имплементира као еквивалентно растућа или опадајућа уређена листа.



## ЗАДАЦИ #2 (Линеарне Структуре)

1) Дата је двоструко спрегнута листа целих бројева сортирана у растућем редоследу и показивач који показује на први елемент у листи. Написати методу која пребројава колико има елемената који су већи од просека целе листе.

```
public int Zadatak1(Lista lista) {
    int broj,prosek = 0;
    int brojac = 1;

    //prvi deo: odrediti prosek
    Element tmp = lista.first;
    broj = tmp.data;
    while (tmp.next != null) {
        tmp = tmp.next;
        broj = broj + tmp.data;
        brojac++;
    }
    prosek = broj/brojac;

    //drugi deo: vratiti broj elemenata
    brojac = 0;
    while (tmp.prev != null) {
        if(tmp.data > prosek) brojac++; else return brojac;
        tmp = tmp.prev;
    }
    return brojac;
}
```

2) Дат је показивач на почетни чвор двоструко спрегнуте листе сортиране у растућем редоследу која садржи позитивне целе бројеве. Написати функцију која ће између свих оних елемената листе који се по вредности разликују за више од 1 убацити у дату листу нове елементе тако да листа после позива операције има у себи сукцесивне целе бројеве. На пример: ако листа садржи {3, 5, 8}, након позива ове функције садржаће {3, 4, 5, 6, 7, 8}.

```
public Lista Zadatak2(Lista lista) {
    Element tmp = lista.first; //recimo da je to dati pokazivač

    while (tmp != null) {
        Element n;
        if((tmp.next != null) && (tmp.data < tmp.next.data-1)) {
            n = new Element(tmp.data+1);
            n.prev = tmp; n.next = tmp.next;
            tmp.next.prev = n; tmp.next = n;
            tmp = n;
        }
        else tmp = tmp.next;
    }
    return lista;
}
```

3) Написати функцију **Transformisi(Lista lista, int[] stek)** која ће од стека који је имплементиран као једноструко спрегнута листа формирати нови стек који је имплементиран преко низа.

```
public int[] Transformisi(Lista lista, int[] stek) {
    Element tmp = lista.first;
    int brojac = 0;
    while(tmp != null) { brojac++; tmp = tmp.next; }
    tmp = lista.first; stek = new int[brojac];
    for (int i=0; i<brojac; i++) { stek[i] = tmp.data; tmp = tmp.next; }
    return stek;
}
```

4) Дат је показивач на неки чвор двоструко спрегнуте листе целих бројева која сигурно садржи најмање 4 елемента. Написати функцију `ubaciInt(Element e, int a, int n)` која ће убацити нови елемент са садржајем `a` и то тако да он буде на `n`-тој позицији од почетка.

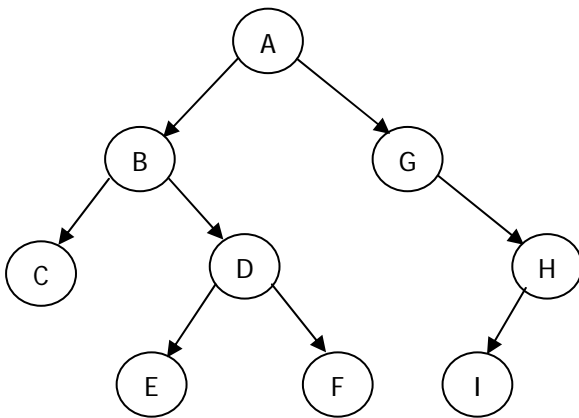
```
public void ubaciInt(Element e, int a, int n) {
    Element tmp = new Element(a);
    while(e.prev != null) { e = e.prev; } //sada je e prvi element liste
    for (int i=0; i<n-1; i++) { e = e.next; } //e - element na n-tom položaju u listi
    tmp.prev = e.prev;
    tmp.next = e;
    e.prev = tmp;
}
```

# НЕЛИНЕАРНЕ СТРУКТУРЕ ПОДАТАКА

Када један елемент структуре може бити у вези са више од два друга елемента, таква структура података је нелинеарна (вишедимензионална) структура. Меморијска реперезентација оваквих структура је најчешће уланчана, али може бити и секвенцијална. Нелинеарне структуре су *сјабла* и *графови*.

## СТАБЛА

Стабло је коначан, непразан скуп елемената (*чворова*), који се састоји од првог чвора (*корена*) и од осталих чворова, који се могу раздвојити у своје подскупове (*подстабла*).



На слици се може уочити корен А, који се грана у два чвора (В и Г). Линије које повезују чворове у стаблу су **џране**. **Улазни сџејен** (ред) стабла је број грана које улазе у чвор, док је **излазни сџејен** број грана које из њега излазе.

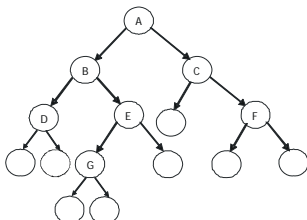
Чворови могу бити **нейћерминални** (имају излазне гране) и **ћерминални** (немају излазне гране), који се називају **лисћови** (на слици су то чворови Е, F и I). **Оћац** је чвор из којег се гранају неки чворови (на слици: А у односу на В и G), док су **синови** чворови који потичу иу неког оца чвора. **Браћа** су синови чворови који потичу од истог оца. Појам **ћредака** и **ћоћомка** је везан за једну повезану контуру (на слици, А је предак за I, док је I потомак од А).

**Пуџ** је скуп грана које су линеарно повезане (на слици : А-Г-Н-І), док је **дужина њуџа** број грانا на путу (на слици - 3). **Ниво стабла** (висина или дубина стабла) је број нивоа, тј. укупан број чворова у најдужем путу – 1 (јер је први чвор на нивоу 0).

Стабла могу бити:

- *слична* - исте топологије (распоред и повезаност чворова)
- *еквивалентна* - слична стабла која имају исти садржај чворова
- *уређена* - познато је који је први син а који је други
- *позiciona* - важна је позиција сваког чвора

Неопходно је разликовати појмове *интерних* и *екстерних* чворова.



- интерни чворови су прави (постојећи) чворови неког стабла (на слици су то чворови A,B,C,D,E,F,G)
- екстерни чворови су чворови који се додају да би сваки чвор имао исти број синова (на слици су представљени као празни кругови).

Максимални број интерних чворова:

$$n_{\max} = \frac{m^{h+1}}{m-1}$$

h - дубина (ниво) стабла

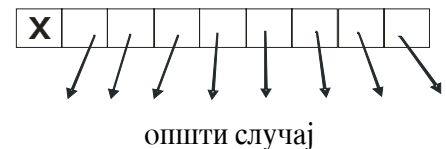
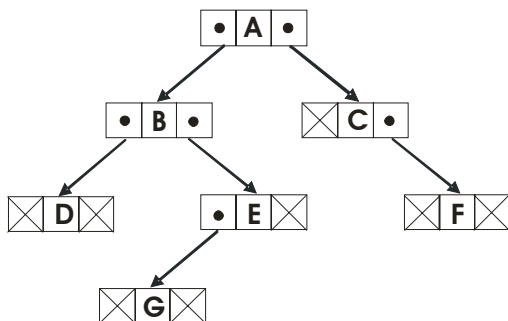
m - број излазних грана неког чвора

Број екстерних чворова:

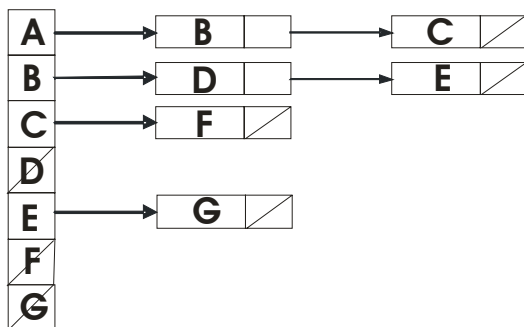
$$l = n(m-1) + 1$$

тако да важи релација:  $n + l = 2^l - 1$  где је n број интерних чворова

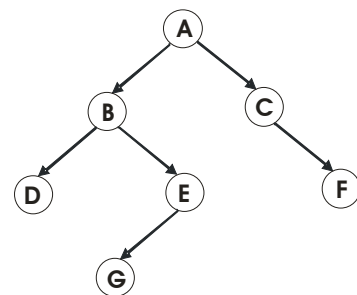
### Уланчана реперезентација стабла



Уланчана реперезентација је погодна за рад са стаблима (кретање, проналажење, брисање...), поготово ако се уведе додатни показивач са сина на оца, међутим, искоришћење простора је изузетно неефикасно (велики број неискоришћених показивача). Зато се често користи просторно прихватљивија уланчана реперезентација која има само n неискоришћених показивача.



$\Leftrightarrow$



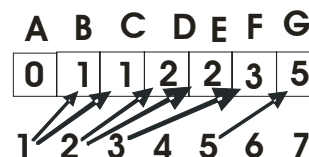
У примеру на слици се може приметити да постоји тачно 7 неискоришћених показивача.

- A показује на B и C
- B показује на D и E
- C показује на F
- E показује на G

### Секвенцијална реперезентација стабла

1	A	2	3
2	B	4	5
3	C	0	6
4	D	0	0
5	E	7	0
6	F	0	0
7	G	0	0

- A показује на 2 и 3, тј. на B и C
- B показује на 4 и 5, односно на D и E



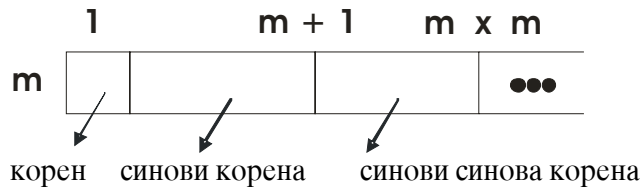
За чвор  $k$  у вектору важи:

ОТАЦ  $\rightarrow \lfloor (k+m-2)/m \rfloor = \lfloor (k-1)/m \rfloor$

СИНОВИ  $\rightarrow m(k-1) + 2, m(k-1) + 3, \dots, mk + 1$

Ове релације се могу користити за испитивање да ли је чвор отац неког чвора  $k$ , или син, или ниједно од та два.

Меморијска имплементација:

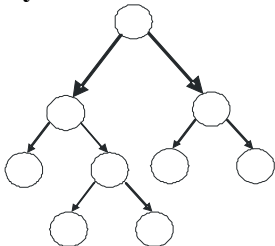


## Бинарна стабла

Бинарно стабло је коначан скуп чворова који је или празан, или се састоји од корена са два посебна подстабла (левим и десним), која су такође бинарна стабла (сваки члан је повезан са два члана) - највише два члана. Овакво стабло је различито од уређеног стабла  $m=2$ .

Врсте бинарних стабала:

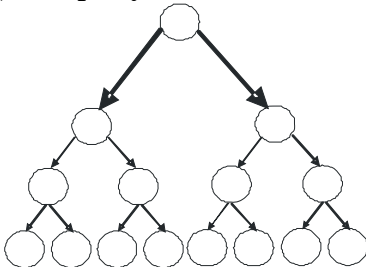
### 1) пуно стабло



- сви чворови имају степен 2

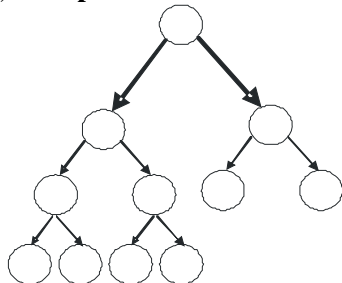
- број чворова у пуном стаблу је увек непаран, због корена стабла

### 2) скоро пуно стабло



Скоро пуно стабло је пуно стабло у којем се сви листови налазе на истом нивоу.

### 3) скоро комплетно стабло



Скоро комплетно стабло не мора бити пуно стабло. Листови су секвенцијално попуњени са леве на десну страну.

Максималан број чворова у бинарном стаблу је  $n_{\max} = 2^{l+1} - 1$ . Минимална дубина, тј. ниво бинарног стабла је  $h_{\min} = \lceil \log_2(n+1) \rceil - 1$ .

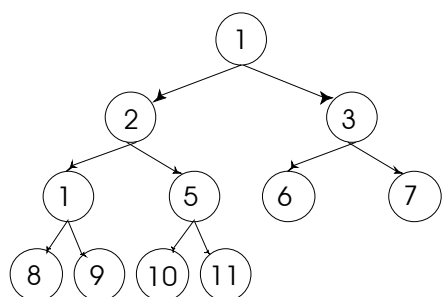
Чворови се код бинарног стабла смештају у меморију на сл. начин:



Оваква секвенцијална имплементација омогућава 100 % меморијско искоришћење и често се примењује код бинарних стабала. Проналажење сродства код овакве имплементације је следећи:

- отац:  $\lfloor i/2 \rfloor \rightarrow \text{idiv}2$
- леви син:  $2i$
- десни син:  $2i+1, 2i+1 \leq n$

пример:



$i = 5$

Отац:  $\lfloor i/2 \rfloor = 2$

Леви син:  $2i = 10$

Десни син:  $2i+1 = 11$

### Начини обиласка бинарног стабла

Постоји више начина да се обиђе неко стабло. Неки од њих су **preorder**, **inorder**, **postorder**, **levelorder** итд.

#### 1) PREORDER обилазак:

- посети се корен
- обиђе се лево подстабло на preorder начин
- обиђе се десно подстабло на preorder начин

резултат: ABCDEFG → први чвор је корен

#### 2) INORDER обилазак:

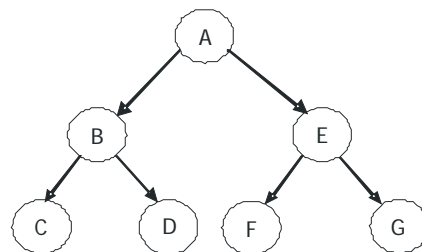
- посети се лево подстабло на inorder начин
- посети се корен
- посети се десно стабло на inorder начин

резултат: CBDAFEG → први чвор је најлевљи чвор

#### 3) POSTORDER обилазак

- обиђе се лево подстабло на postorder начин
- обиђе се десно подстабло на postorder начин
- обиђе се корен

резултат: CDBFGEA → први чвор је најлевљи чвор





## Обилажење бинарног стабла - рекурзивно

Чвор стабла:

```
public class Cvor {  
    public char data;  
    public Cvor levi;  
    public Cvor desni;  
  
    public Cvor(char data) { this.data = data; }  
}
```

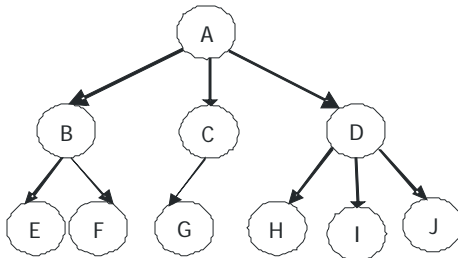
Стабло са рекурзивним методама обиласка:

```
public class BinarnoStablo {  
    public Cvor koren;  
  
    public BinarnoStablo(Cvor koren) { this.koren = koren; }  
  
    public static void Preorder(Cvor koren) {  
        if (koren != null) System.out.println(koren.data);  
        if (koren.levi != null) Preorder(koren.levi);  
        if (koren.desni != null) Preorder(koren.desni);  
    }  
  
    public static void Inorder(Cvor koren) {  
        if (koren.levi != null) Inorder(koren.levi);  
        if (koren != null) System.out.println(koren.data);  
        if (koren.desni != null) Inorder(koren.desni);  
    }  
  
    public static void Postorder(Cvor koren) {  
        if (koren.levi != null) Postorder(koren.levi);  
        if (koren.desni != null) Postorder(koren.desni);  
        if (koren != null) System.out.println(koren.data);  
    }  
  
    public static void main(String[] args) {  
        BinarnoStablo stablo = new BinarnoStablo(new Cvor('A'));  
        stablo.koren.levi = new Cvor('B');  
        stablo.koren.desni = new Cvor('E');  
        stablo.koren.levi.levi = new Cvor('C');  
        stablo.koren.levi.desni = new Cvor('D');  
        stablo.koren.desni.levi = new Cvor('F');  
        stablo.koren.desni.desni = new Cvor('G');  
  
        System.out.println("Preorder obilazak stabla:");  
        BinarnoStablo.Preorder(stablo.koren);  
        System.out.println("-----");  
        System.out.println("Inorder obilazak stabla:");  
        BinarnoStablo.Inorder(stablo.koren);  
        System.out.println("-----");  
        System.out.println("Postorder obilazak stabla:");  
        BinarnoStablo.Postorder(stablo.koren);  
        System.out.println("-----");  
    }  
}
```

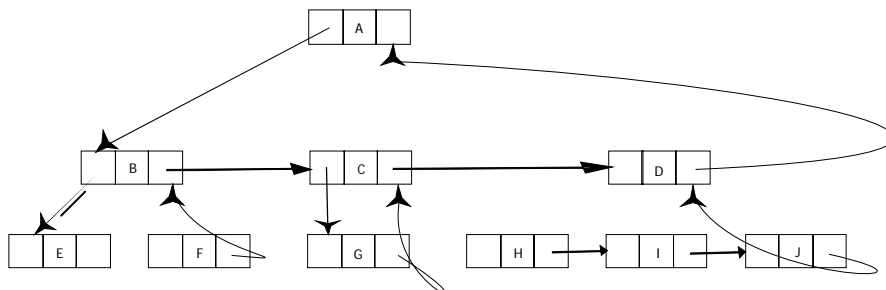
## Стабла вишег реда (m)

Проблем стабала вишег реда је неефикасно коришћење простора у уланчаној репрезентацији ( $n(m-1)+1$  неискоришћених показивача, а само  $m-1$  искоришћених показивача). Решење овог проблема било би неко бинарно стабло исте семантике (распореда чворова...).

Овакво стабло се заснива на релацији **“најлевљи син - десни браћ”**, а сви синови истог оца се налазе у уланчаној листи.



Дато стабло изгледа овако:



Често се неискоришћени показивач најдеснијег сина враћа на оца.

## ЗАДАЦИ #3 (Стабла)

1) Дат је показивач на корен бинарног стабла, чији су елементи цели бројеви. Написати методу која ће вратити минималну вредност чворова стабла.

```
public static int VратиMinimum(Cvor koren, int min) {
    int tmp = 0;
    if (koren != null) {
        if (koren.data <= min)
            min = koren.data;
    }
    if (koren.levi != null) {
        tmp = VратиMinimum(koren.levi, min);
        if (tmp <= min)
            min = tmp;
    }
    if (koren.desni != null) {
        tmp = VратиMinimum(koren.desni, min);
        if (tmp <= min)
            min = tmp;
    }
    return min;
}
```

Напомена: методи се, на почетку, прослеђује корен стабла и `Integer.MAX_VALUE`.

2) Написати методу која враћа укупан број чворова у стаблу.

```
public static int IzbrojCvorove(Cvor koren) {
    if (koren == null) return 0;
    return 1 + IzbrojCvorove(koren.levi) + IzbrojCvorove(koren.desni);
}
```

3) Написати методу која враћа висину стабла.

```
public static int VратиVisinu(Cvor koren) {
    if (koren == null) return 0;
    return 1 + Math.max(VратиVisinu(koren.desni), VратиVisinu(koren.levi));
}
```

4) Два бинарна стабла су идентична ако су иста по структури и садржају, што значи да оба корена имају исти садржај и њихова одговарајућа подстабла су идентична. Написати методу која ће проверити да ли су два бинарна стабла идентична.

```
public static boolean IdenticnaStabla(Cvor koren1, Cvor koren2) {
    if (koren1 == null && koren2 == null) return true;
    if (koren1 != null && koren2 != null) {
        if (koren1.data == koren2.data) {
            return IdenticnaStabla(koren1.levi, koren2.levi) &&
                IdenticnaStabla(koren1.desni, koren2.desni);
        }
    }
    return false;
}
```

5) Дата је референца на корен неког бинарног стабла. Чворови стабла садрже целе бројеве. Написати методу која ће вратити број чворова који су по садржају већи од својих потомака.

```
public static int BrojCvorova(Cvor koren) {
    if (koren == null) return 0;
    if (koren.levi != null && koren.desni != null) {
        if (koren.data > koren.levi.data && koren.data > koren.desni.data)
            return 1 + BrojCvorova(koren.levi) + BrojCvorova(koren.desni);
        else return BrojCvorova(koren.levi) + BrojCvorova(koren.desni);
    }
    if (koren.levi != null && koren.desni == null) {
        if (koren.data > koren.levi.data)
            return 1 + BrojCvorova(koren.levi);
        else return BrojCvorova(koren.levi);
    }
    if (koren.levi == null && koren.desni != null) {
        if (koren.data > koren.desni.data)
            return 1 + BrojCvorova(koren.desni);
        else return BrojCvorova(koren.desni);
    }
    return 0;
}
```

У овом задатку се морају пратити четири случаја: у првом случају постоје два потомка, у следећа два постоји само по један потомак (леви или десни), а четврти случај (када нема потомака) је сервисан последњом наредбом у методи. Само на овај начин ће метода радити у свим случајевима, јер унапред није познато каквог је изгледа стабло.

6) Написати методу која прихвата референцу на корен бинарног стабла и референцу на неки чвор, а враћа референцу на родитеља датог чвора, односно **null** ако родитељ не постоји (што значи да чвор није из датог стабла). Напомена: чвор стабла има референце само на своје потомке.

```
public static Cvor VратиRoditelja(Cvor koren, Cvor cvor) {
    if (koren == null || koren == cvor) return null;
    if (koren.levi == cvor || koren.desni == cvor) return koren;
    Cvor tmp;
    tmp = VратиRoditelja(koren.levi, cvor);
    if (tmp != null) return tmp;
    tmp = VратиRoditelja(koren.desni, cvor);
    if (tmp != null) return tmp;
    return null;
}
```

7) Бинарно стабло се назива HEAP ако за сваки чвор важи да је његов садржај већи од садржаја свих осталих чворова у његовим подстаблима. Написати методу која ће проверити да ли је задато стабло HEAP.

```
public static boolean DaLiJeHeap(Cvor koren) {
    if (koren == null) return true;
    if ((koren.levi != null && koren.data <= koren.levi.data) ||
        !DaLiJeHeap(koren.levi))
        return false;
    if ((koren.desni != null && koren.data <= koren.desni.data) ||
        !DaLiJeHeap(koren.desni))
        return false;
    return true;
}
```

8) Написати методу која ће вратити родитеља датог чвора (разређен код у односу на задатак 6).

```
public static Cvor VратиRoditelja(Cvor koren, Cvor cvor) {
    if (koren == null) return null;
    if (koren == cvor) return null;
    if (koren.levi != null && koren.levi == cvor) return koren;
    if (koren.desni != null && koren.desni == cvor) return koren;
    Cvor tmp = null;
    if (koren.levi != null) tmp = VратиRoditelja(koren.levi, cvor);
    if (tmp != null) return tmp;
    if (koren.desni != null) tmp = VратиRoditelja(koren.desni, cvor);
    if (tmp != null) return tmp;
    return null;
}
```

9) Написати методу која прима референцу на корен бинарног стабла целих бројева и враћа минималну вредност садржаја чворова стабла.

```
public static int vratiMin(Cvor koren, int min) {
    if (koren != null && koren.data <= min) min = koren.data;
    int minL;
    if (koren.levi != null) {
        minL = Math.min(min, vratiMin(koren.levi, min));
    } else minL = min;
    int minD;
    if (koren.desni != null) {
        minD = Math.min(min, vratiMin(koren.desni, min));
    } else minD = min;
    return Math.min(minL, minD);
}
```

10) Написати методу која прима референцу на корен бинарног стабла и штампа садржај свих чворова на путу од корена до чвора са најмањом вредношћу у стаблу.

```
private static Cvor VratiMinCvor(Cvor koren, int min) {
    if (koren == null) return null;
    if (koren.data == min) return koren;
    Cvor tmp = null;
    tmp = VratiMinCvor(koren.levi, min);
    if (tmp != null) return tmp;
    tmp = VratiMinCvor(koren.desni, min);
    if (tmp != null) return tmp;
    return null;
}

public static void StampaJDoMinCvora(Cvor koren) {
    if (koren == null) return;
    Cvor tmp = VratiMinCvor(koren, vratiMin(koren, Integer.MAX_VALUE));
    ArrayList lista = new ArrayList();
    while (tmp != null) {
        lista.add(Integer.toString(tmp.data));
        tmp = VratiRoditelja(koren, tmp);
    }
    Object[] niz = lista.toArray();
    for (int i=niz.length-1; i>-1; i--) System.out.println(niz[i].toString());
}
```

11) Дата је референца на корен бинарног стабла чији чворови садрже целе бројеве. Написати методу која ће вратити референцу на чвор датог стабла код кога је најмањи производ садржаја чворова из његовог десног подстабла.

```
// metoda koja vraća proizvod sadržaja čvorova stabla
private static int VratiProizvod(Cvor koren) {
    if (koren == null) return 1;
    return koren.data * VratiProizvod(koren.levi) * VratiProizvod(koren.desni);
}

// metoda koja radi posao
private static Object[] VC(Cvor koren, Cvor ref, int proizvod, int min) {
    if (koren == null) {
        Object[] ret = new Object[2];
        ret[0] = ref; ret[1] = new Integer(min);
        return ret;
    }
    int p = Integer.MAX_VALUE;
    if (koren.desni != null) p = VratiProizvod(koren.desni);
    if (p < min) {
        min = p;
        ref = koren;
    }
    Object[] levo, desno;
    levo = VC(koren.levi, ref, p, min); Integer l = (Integer)levo[1];
    desno = VC(koren.desni, ref, p, min); Integer d = (Integer)desno[1];
    if (l.intValue() < d.intValue()) return levo;
    else return desno;
}

// omotačka metoda
public static Cvor VratiCvor(Cvor koren) {
    Object[] ret = VC(koren, null, 1, Integer.MAX_VALUE);
    return (Cvor)ret[0];
}
```

## АЛГОРИТМИ ПРЕТРАЖИВАЊА

Претраживање представља налажење жељеног податка у структури података, у циљу приступа, на основу неке идентификације (**кључа**). Ова активност је изузетно честа, па је зато њена сложеност битна.

Ако неку структуру података назовемо **табела** или **дајтошека**, то је група (скуп) података, од којих се сваки назива **запис (чвор)**. Кључеви, као идентификација записа, могу бити :

- **интерни** - у оквиру записа, као његов део (поље)
- **екстерни** - скуп кључева који садрже показиваче на записе

Према информацији коју садрже, кључеви могу бити :

- **примарни** - јединствени кључеви, не постоје два записа са истим кључем
- **секундарни** - вредности кључева нису нужно јединствене

Процес претраживања се може дефинисати као **алгоритам који прима аргумент х и проналази један или више чворова - записа у датотеци или табели чија је вредност кључа х**. Према исходу, претраживање може бити успешно и неуспешно. Постоје варијације алгоритама за претраживање, па је, у неким случајевима, погодно убацити кључ у датотеку или табелу, уколико је резултат претраживања неуспех.

Према месту претраживања, претраживање може бити **унутрашње** (у меморији) и **спољашње** (у датотекама на екстерним медијумима). Скуп података који се претражује може бити:

- **статичан и динамичан** - статичан није склон мењањима, динамичан јесте
- **уређен и неуређен** - за неуређене је могуће само секвенцијално претраживање

## СЕКВЕНЦИЈАЛНО ПРЕТРАЖИВАЊЕ

Секвенцијално претраживање представља претраживање < елемент по елемент > и самим тим се за њега имплементира најпростији алгоритам, међутим поседује највећу сложеност ( $O(n)$ ). Ово је једини могући начин претраживања листа и неуређених скупова.

Само претраживање се изводи тако што се анализира елемент по елемент структуре, користећи се методама самих структура за прелазак са податка на податак, редоследом који је задат самом структуром. Када се дође до жељеног записа, он се враћа, или се враћа грешка (нема елемента).

Елемент који се претражује и грешка:

```
public class Element {
    public int kljuc;
    public Object vrednost;
}

public class Greska extends Exception {
    public String toString() { return " *** Element Ne Postoji! ***"; }
}
```

Секвенцијално претраживање низа на кључ:

```
public Object Search(Element[] niz, int kljuc ) throws Greska {
    int duzina = niz.length;
    for (int i=0; i<duzina; i++) {
        if (niz[i].kljuc == kljuc) return niz[i].vrednost;
    }
    throw new Greska();
}
```



## Секвенцијално претраживање листе на кључ:

```
public Object Search (Lista lista, int kljuc) throws Greska {
    Element tmp = lista.first;
    while ((tmp = tmp.getNext()) != null) {
        if (tmp.kljuc == kljuc) return tmp.vrednost;
    }
    throw new Greska();
}
```

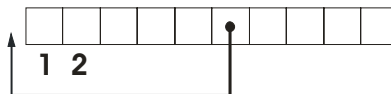
Уколико је потребно да, ако елемент не постоји у датотеци, буде креиран нови елемент датотеке, уместо операције `throw new Greska()` извршиће се уметање новог елемента у листу или низ.

Претходно је поменуто да је ефикасност алгорита за секвенцијално претраживање  $O(n)$ , што значи да ће метода која га имплементира  $n$  пута дохватити елементе структуре података, све док не пронађе елемент са одговарајућим кључем. Уколико је елемент на првом месту, јасно је да ће се операција брзо извршити, међутим, ако је близу последњег места, брзина је драстично умањена.

Секвенцијално претраживање је могуће оптимизовати, ако су познате вероватноће наласка кључева, тако што ће кључеви са већим вероватноћама наласка бити на првим местима у низу или листи. Међутим, проблеми са вероватноћама су управо непознавање истих, као и њихове сталне промене. Најпознатије методе оптимизације помоћу вероватноћа су *пребацавање на почетак* и *метод транспозиције*.

### Пребацавање на почетак

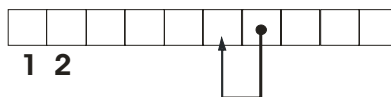
Овај метод оптимизације допуњује алгоритам претраживања, тако да се кључ, чим се пронађе, ставља на прво место у скупу кључева, под претпоставком да ће му се у блиској будућности опет приступити.



Оптимизација методом пребацавања за почетак се користи за спрегнуте листе. Проблем методе су велике осцилације у перформансама (у неким случајевим је боље користити, у неким не).

### Транспозиција

Метод транспозиције је оптимизација секвенцијалног претраживања која допуњује алгоритам претраживања на тај начин што се, када се кључ нађе, помери за једно место ближе почетку датотеке.



Уколико се дати кључ често користи, у једном моменту ће вероватно завршити на самом почетку датотеке и тако оправдати помераје. Ова оптимизација није тако радикална као претходна, али има стабилније перформансе.

### Претраживање сортиране датотеке

Уколико је датотека коју треба претражити реализована на такав начин, да су елементи уређени, тј. сортирани унутар ње, сам процес претраживања је веома олакшан. Сортирана датотека је или у опадајућем или у растућем редоследу, што смањује сложеност алгорита за претраживање.

Ако је, на пример, датотека сортирана у растућем редоследу и програм покушава да је претражи на кључ  $x$ , метода може претраживања може бити реализована на следећи начин: чим се наиђе на запис чији је кључ већи од  $x$ , метода враћа грешку, јер сигурно нема тог елемента.

## БИНАРНО ПРЕТРАЖИВАЊЕ

Како секвенцијално претраживање споро конвергира ка траженом елементу, за уређене низове се користи рекурзивна тактика < divide & conquer >, коју имплементира алгоритам за бинарно претраживање, који се своди на половљење интервала претраживања.

Методи се прослеђује кључ који се тражи. Он се пореди са кључем записа који се налази у средини на пр. растуће сортиране табеле. Уколико су једнаки, претрага се успешно завршава. Ако је кључ који се тражи мањи, претражује се прва половина, поново поређењем са кључем записа који се налази у средини прве половине. Овакав поступак омогућен је применом рекурзије:

Бинарно претраживање низа на кључ:

```
public Object Search(Element[] niz, int kljuc, int low, int high) throws Greska {
    if (low > high) throw new Greska();

    int mid = (low + high) / 2;
    if (niz[mid].kljuc == kljuc) return niz[mid].vrednost;
    else if (low == high) throw new Greska();
    else if (niz[mid].kljuc < kljuc) return Search(niz, kljuc, mid+1, high);
    else return Search(niz, kljuc, low, mid-1);
}
```

Сложеност овог алгоритма је мања од сложености алгоритма за секвенцијално претраживање и износи  $O(\log_2 n)$ . Бинарно претраживање је непримењиво за уланчане листе, веће се може применити искључиво на сортиране низове. Практична примена је отежана, јер је код низова теже убацивање елемената, када их алгоритам за претраживање не пронађе.

Када је ефикасност претраживања примарна, спорост рекурзије захтева нерекурзивну имплементацију алгоритма за бинарно претраживање:

```
public Object Search2(Element[] niz, int kljuc) {
    Object rezultat = null;
    int low = 0;
    int high = niz.length-1;
    int mid;

    while (high >= low) {
        mid = (low + high) / 2;
        if (niz[mid].kljuc > kljuc) high = mid-1;
        else if (niz[mid].kljuc < kljuc) low = mid+1;
        else { rezultat = niz[mid].vrednost; break; }
    }

    return rezultat;
}
```

## Интерполационо претраживање

Уколико је потребно реализовати алгоритам претраживања структуре података чије су вредности кључева униформно распоређене (фактор повећања вредности кључева је константан) између минималне и максималне вредности, које су унапред познате, постиже се претраживање које је брже од бинарног. То је интерполационо претраживање.

Интерполационо претраживање се остварује када је разлика између елемената датотеке приближно константна:

1	3	5	8	10	14	16	18	21	24
0	1	2	3	4	5	6	7	8	9

$$\text{mid} = \text{low} + (\text{high} - \text{low}) * \frac{\text{ključ} - \text{n}[\text{low}]}{\text{n}[\text{high}] - \text{n}[\text{low}]}$$

← позиција првог елемента
← позиција последњег елемента
← max вредност
← min вредност

Поступак претраживања је следећи: вредност **low** поставимо на 0, а вредност **high** на **n.length-1**. Позиција записа са траженим кључем се одреди приближно (интерполира) по датаој формули. Уколико на тој позицији није запис са задатим кључем, прелази се на псеудо-бинарно претраживање, у смислу да се рекурзивно понавља процес уз промене **high** и **low** елемената.

Уколико су кључеви заиста униформно распоређени, сложеност алгоритма који имплементира интерполационо претраживање је  $O(\log_2(\log_2 n))$ , што је боље од бинарног претраживања. Међутим, ако кључеви нису униформно распоређени, ефикасност алгоритма се знатно смањује. У пракси, честа је ситуација да се кључеви групишу око неке вредности. Тада се користи **fast search** претраживање.

### Робусно интерполационо претраживање - Fast Search

**Fast search** претраживање покушава да поправи перформансе интерполационог претраживања и у случају неуниформне расподеле кључева. Ово се постиже увођењем локалне променљиве у методи која имплементира алгоритам интерполационог претраживања. То је променљива **razmak**, за коју важи да је увек мања од **index(ključ) - low** или **high - index(ključ)**.

На почетку, размак је постављен на вредност  $\text{razmak} = \sqrt{(\text{high} - \text{low} + 1)}$ . Помоћна променљива је постављена на вредност:

$$\text{pom} = \text{low} + \frac{\text{ključ} - \text{n}[\text{low}]}{\text{high} - \text{low}} * (\text{high} - \text{low})$$

Тако је **index(ključ)** једнак:

$$\text{index}(\text{ključ}) = \min(\text{high} - \text{razmak}, \max(\text{pom}, \text{low} + \text{razmak}))$$

Овиме се гарантује да је **index(ključ)** следећа позиција која ће бити испитана, која је најмање **razmak** позиција удаљена од крајева интервала **low** и **high**. Када се открије да се кључ који тражимо налази у већем делу интервала вредност променљиве **razmak** се дуплира. Тако се избегавају груписања кључева са приближним вредностима. Кда је **ključ** у мањем делу интервала, тада се **razmak** враћа на квадратни корен новог интервала (рекурзивно).

Када је реч о ефикасности робусног интерполационог претраживања, оно у просеку има ефикасност  $O(\log_2(\log_2 n))$  као интерполационо. У најгорем случају, ефикасност је  $O(\log_2 n)$ .

### Индекс-секвенцијално претраживање

Метод индекс-секвенцијалног претраживања користи додатну помоћну табелу. Датотека је сортирана, као предуслов овог претраживања и величине је **n**, а подељена у **k** блокова са по **m** записа. Највећи кључ из сваког блока је смештен у табелу индекса, заједно са показивачем на блок из кога потиче.

Табела индекса је такође сортирана. Претраживање почиње од табеле, тако што се тражи први кључ који има вредност једнаку или већу од траженог кључа. Помоћу показивача који се налазе у табели приступа се блоку порекла нађеног кључа. Уколико је нађену кључ једнак кључу који тражимо, враћа се тај кључ (први из блока), а ако не, секвенцијално се претражи блок на кључ и запис се проналази са највише **m** поређења.

Просечан број поређења приликом претраживања датотеке је једнак просечном броју поређења код претраживања табеле индекса сабраним са просечним бројем поређења код секвенцијалног претраживања блока:

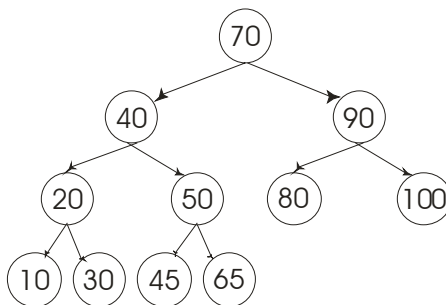
$$P = k/2 + m/2 = (k + n/k)/2$$

## СТАБЛА ПРЕТРАЖИВАЊА

Када је датотека организована као низ или листа, користе се већ поменуте методе претраге ових структура - секвенцијално, бинарно, интерполационо... За имплементацију датотеке са ефикасним претраживањем често се користи **BST стабло** (бинарно стабло тражења).

### BST СТАБЛА

BST стабло је бинарно стабло, које је уређено на такав начин, да сваки чвор садржи један кључ и показиваче на своја два сина (као обично бинарно стабло), с тим што је увек задовољен услов да леви син садржи кључ који је мањи од кључа оца, а да је кључ десног сина већи од кључа оца. Исто тако, сви чворови левог подстабла имају мање кључеве од кључа корена, а сви кључеви десног подстабла имају веће кључеве од кључа корена.



Најефикасније претраживање је када је BST стабло балансирано, тј. када су му сви листови на истој висини. Сложеност оваквог претраживања је  $\log_2 n$  и једнака је висини стабла. Проблем настаје приликом убацивања нових чворова, јер они могу уништити балансираност и самим тим смањити ефикасност претраге стабла и до  $n$ . Овај проблем се решава применом **AVL стабала**, која одржавају балансираност приликом убацивања нових чворова извођењем одговарајућих ротација чворова.

### Уметање чвора у BST стабло

Операцији уметања новог чвора у BST стабло претходи бинарно претраживање на кључ који би требало унети. Само ако је претага неуспешна (дати кључ не постоји у стаблу), нови чвор се може уметнути у стабло и то на место листа, што значи да се структура стабла неће много променити.

BST стабло:

```
public class BST {  
    public Cvor koren;  
  
    public BST() {}  
    public BST(Cvor koren) { this.koren = koren; }  
}
```

Рескурзивно уметање чвора:

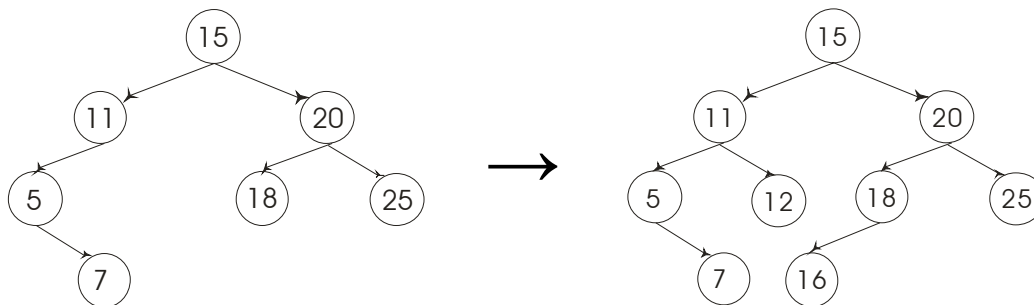
```
public void InsertR(Cvor cvor, Cvor koren) {  
    if (koren == null) { koren = cvor; return; }  
    if (cvor.data == koren.data) {  
        System.out.print("Kljuc vec postoji!");  
        return;  
    }  
    if (cvor.data < koren.data) InsertR(cvor, koren.levi);  
    else InsertR(cvor, koren.desni);  
}
```

## Итеративно уметање чвора:

```
public void InsertI(Cvor cvor) {  
    Cvor tmp = this.koren;  
    Cvor roditelj = null;  
    while (tmp != null) {  
        roditelj = tmp;  
        if (tmp.data == cvor.data) System.out.print("Kljuc vec postoji!");  
        else if (cvor.data < tmp.data) tmp = tmp.levi;  
        else tmp = tmp.desni;  
    }  
    if (cvor.data < roditelj.data) roditelj.levi = cvor;  
    else roditelj.desni = cvor;  
}
```

Помоћна варијабла **roditelj** нам користи да увек буде корак иза варијабле **tmp** и да представља његовог родитеља. То нам омогућава да, на крају, уметнемо нови чвор као лист, јер када **tmp** не постоји, јасно је да је **roditelj** лист на који треба закачити нови чвор.

Пр: уметање чворова са кључевима 12 и 16:



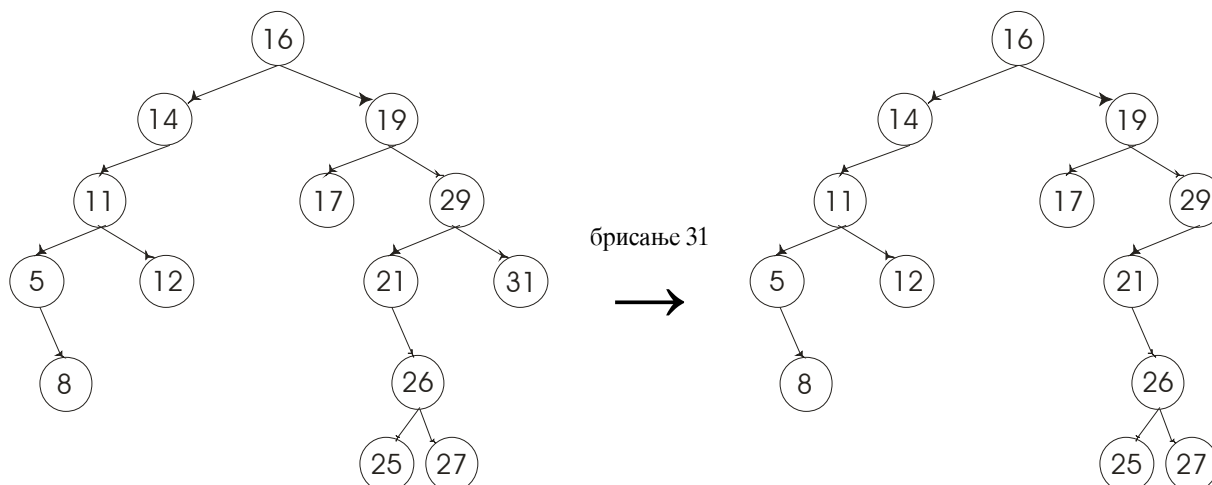
Овај алгоритам се, уз додатак могућности дуплирања кључева, може ефикасно користити и за сортирање.

## Брисање чвора из BST стабла

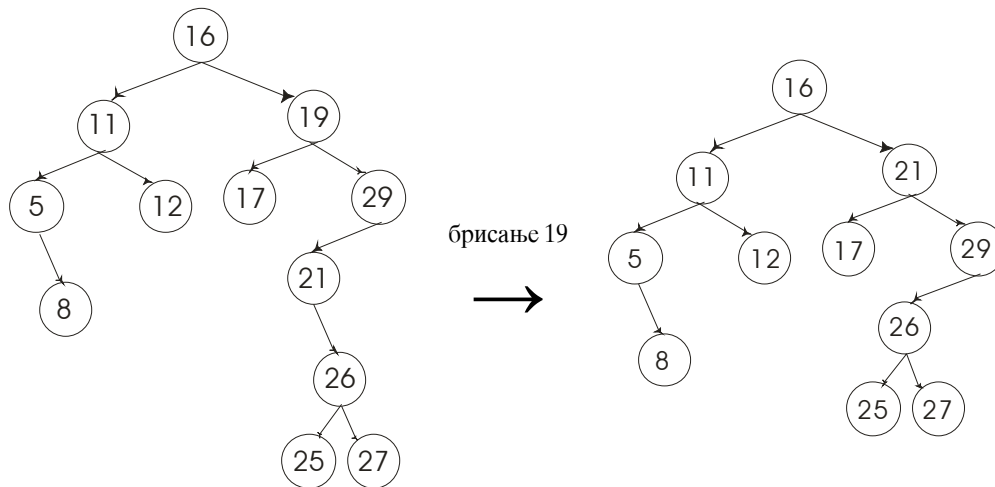
Брисање чвора из BST стабла је сложенији поступак од уметања. Разликују се три случаја:

- 1) *брисање листа* (слично уметању - структура стабла се не мења)
- 2) *брисање чвора са једним подстаблом* (отац преузима његовог следбеника)
- 3) *брисање чвора са два подстабла* (заменају се следбеник или претходник)

Пример брисања чворова :



Следећи корак је брисање чвора са кључем 14:



Код трећег случаја, треба комбиновати замењивање са следбеником и претходником, како би се очувао правилан изглед стабла.

## AVL СТАБЛА

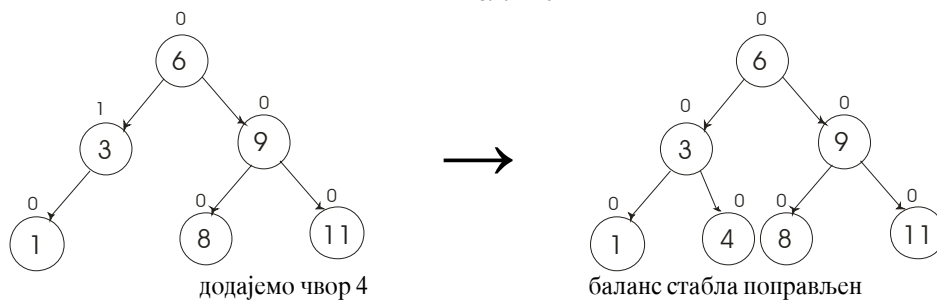
AVL стабло је добило име по *Аделсон-Велски-ју* и *Ландис-у*. Баланс неког стабла се обележава формулом :  $h = h_l - h_r$  и представља разлику висина левог и десног подстабла неког чвора. AVL стабло бинарног претраживања представља стабло коме је за сваки чвор баланс  $|b| \leq 1$ , тј. Разлика између левог и десног постабла у висини не сме бити већа од 1. То је **балансирано BST стабло**.

### Уметање чвора у AVL стабло

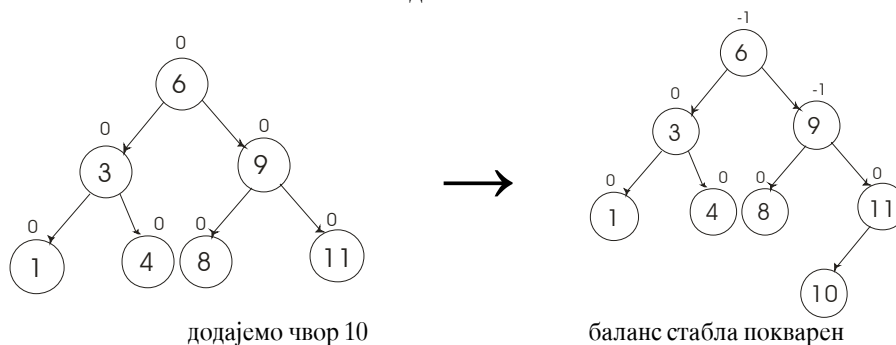
За уметање чвора у AVL стабло постоје три случаја:

- 1) Баланс стабла се поправља
- 2) Баланс се погоршава, али је стабло и даље AVL ( $|b| \leq 1$ )
- 3) Стабло више није AVL ( $|b| > 1$ )

- ПРВИ СЛУЧАЈ -



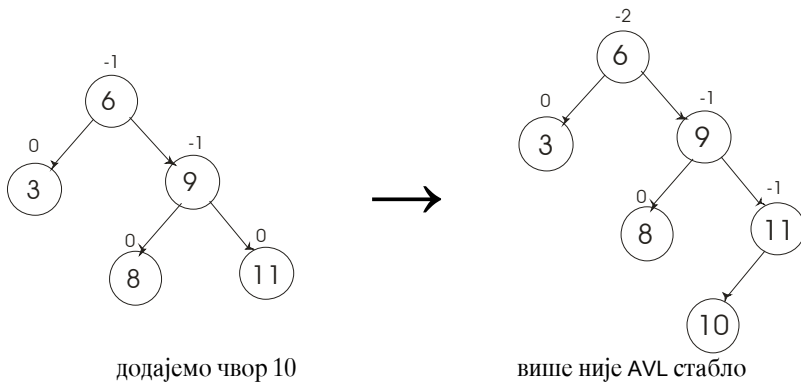
- ДРУГИ СЛУЧАЈ -



- Алгоритми и Структуре података -

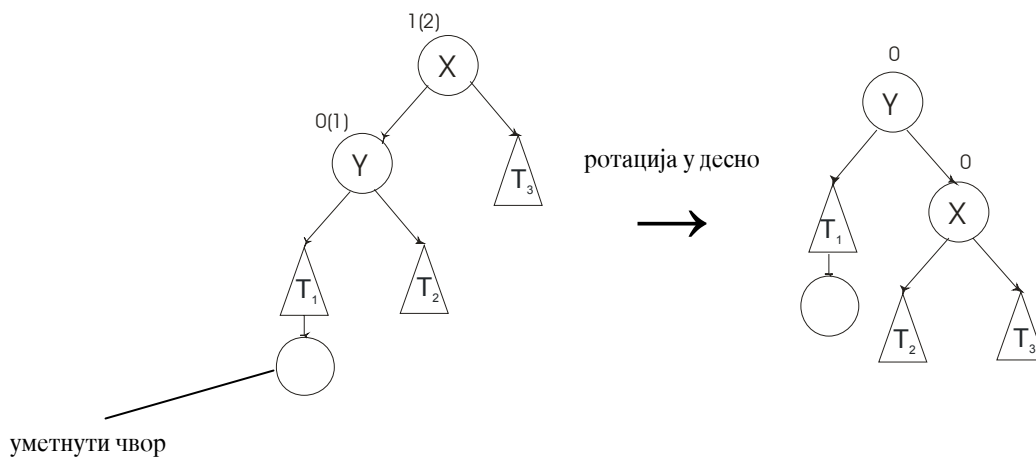


- ТРЕЋИ СЛУЧАЈ -



Приликом уметања чворова, први чвор одоздо-на-горе који има баланс  $|b| > 1$  се назива **критични чвор**. Како би се он вратио у дозвољени баланс, користи се принцип једноструке ротације око критичног чвора. Постоје два карактеристична случаја:

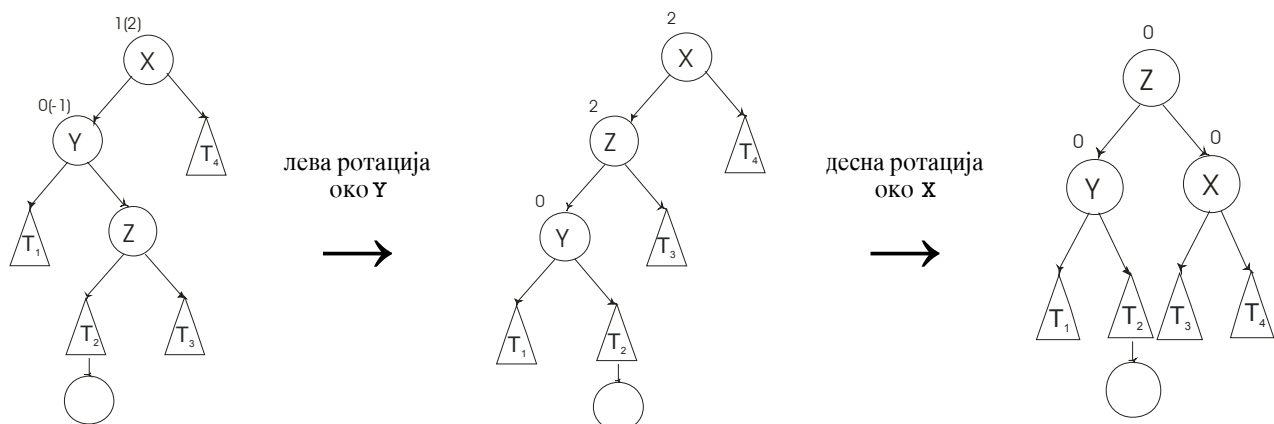
- 1) Критични чвор и његов син на страни уметања се нагињу на исту страну (потребна једна ротација):



```
private void RotirajDesno(Cvor x) {
    Cvor y = x.levi;
    Cvor tmp = y.desni;
    y.desni = x;
    x.levi = tmp;
}
```

```
private void RotirajLevo(Cvor x) {
    Cvor y = x.desni;
    Cvor tmp = y.levi;
    y.levi = x;
    x.desni = tmp;
}
```

- 2) Критични чвор и његов син на страни уметања се нагињу на различиту страну (потребне су две ротације - прво се син првог небалансираног чвора доведе на исту страну као његов отац, а онда се отац балансира као у првом случају)



Ротације око чворова исправљају баланс чворова, али такође и одржавају inorder поредак (BST).

Одржавање AVL стабла је јако једноставно - највише једна двострука ротација приликом уметања и највише једна ротација код брисања чворова. Загарантована је логаритамска перформанса само за нијансу лошија од оптималне.

Најлошији облик AVL стабла је **Фибоначијево сџабло**, које се добија рекурзивно, тако што се прави нови чвор чије је лево подстабло претходно сџабло, а десно подстабло претходно сџабло претходног стабла. Нови чворови који се креирају имају бројеве Фибоначијевог низа (1, 2, 3, 5, 8, ...).

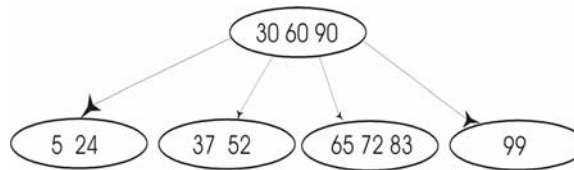
## **СТАБЛА ОПШТЕГ ПРЕТРАЖИВАЊА - VST**

Када се врши претраживање на спољашњој меморији (пр: магнетним дисковима), BST стабла не дају добре перформансе, пре свега због велике висине која доводи до много поређења (обраћања диску - довлачења података са магнетне траке). На екстерној меморији довлачење података је јако споро, тј. знатно спорије од оперативне меморије или кеша.

С обзиром да је екстерна меморија, због великог капацитета и перманентности записа (подаци се не губе гашењем рачунара), веома широко коришћена, јако је значајан био проналазак ефикасном приступу подацима. Пошто је диск електротехнички уређај коме је приступ доста спорији од приступа оперативној меморији, често се тражени податак довлачи у оперативну меморију у виду пакета, тј. блока података. Тако се очекује да ше неки од тих података бити поново претраживан у блиској будућности (циљ је смањење броја приступа диску).

Стабло бинарног претраживања обезбеђује флексибилност, али је неефикасно због великог броја приступа (велика висина). Решење је **сџабло *m*-арног претраживања\***, које има највише ***m*-1** кључева у чвору, чиме обезбеђује мању висину, тј. мањи број приступа диску, а већи број приступа оперативној меморији (из стабла се довлачи читав чвор, тј. блок од више кључева). Претраживање у оквиру чвора се врши у оперативној меморији.

Ова вишегранска стабла имају највише ***m*-1** кључева и ***m*** показивача у чвору, при чему сваки кључ ***k*** раздваја два показивача, а чворови су поређани по растућем редоследу. Подстабла су такође VST стабла. Претраживање у чвору се може вршити секвенцијално или бинарно, а неуспешно претраживање се завршава у листу (преци листа морају бити пуни, а листови не). Листови не морају бити на истом нивоу, што је уједно и недостатак VST стабла.



Из примера са слике је јасно да ће inorder пролазак кроз стабло дати кључеве сортиране по растућем редоследу. Чвор који садржи максимални број кључева се назива **пун чвор** (корен стабла са слике или његов трећи син с лева на десно). Чвор који има бар једно празно подстабло је **полулист**. Уколико су уви полулистови на истој висини **VST стабло је балансирано**.

### Операције у *m*-арном стаблу претраживања:

#### 1) *уметање чвора (кључа)*

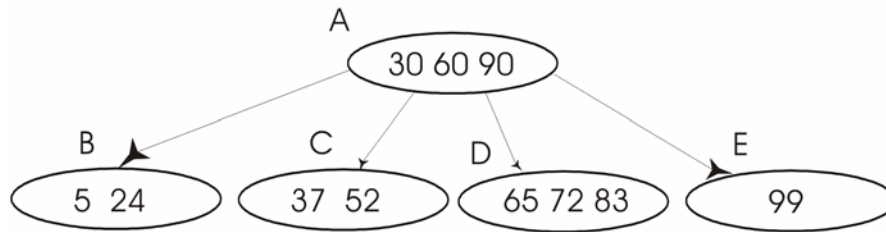
- уметнути чвор ће увек бити лист
- ако чвор у који кључ треба да се уметне није пун, у њега ће се уметнути кључ уз померање већих кључева за једно место

#### 2) *брисање чвора (кључа)*

- ако је кључ у листу, уклања се кључ и сажима се лист (нпр. 3-чвор постаје 2-чвор), а ако је обрисани кључ био једини у листу, брише се цео лист
- ако се кључ не налази у листу, након његовог уклањања у чвору фали један кључ (сви чворови осим листова морају бити пуни) и тај кључ се допуњава из подстабла.

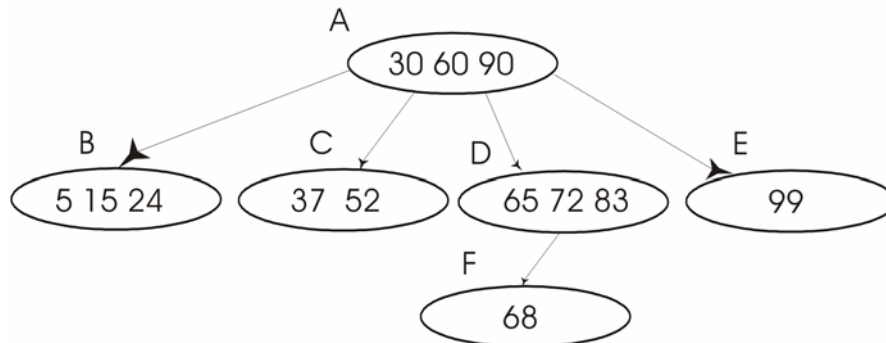
\* вишегранско стабло претраживања - **VST**

Ако почетно VST стабло изгледа овако:

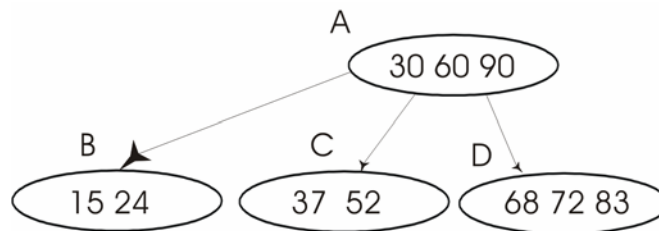


стабло мора да задржи **top-down** критеријум организације (сви преци листова морају бити пуни, тј. попуњавање се врши одозго на доле).

Ако додамо кључеве 15 и 68, стабло ће изгледати овако:



И ако, на крају, обришемо кључеве 5, 65 и 99, крајњи изглед је:



Чвор VST стабла:

```
public class Cvor {  
    int[] kljucevi; // niz ključeva  
    Cvor[] sinovi; // niz pokazivača (referenci) na sinove  
  
    public Cvor() {  
        this.kljucevi = new int[3];  
        this.sinovi = new Cvor[4];  
    }  
}
```

Претраживање VST стабла:

```
public static Cvor Pretrazi(Cvor koren, int kljuc) {  
    if (koren == null) return null;  
    int pozicija = 0;  
    for (int i=0; i<3; i++) {  
        if (koren.kljucevi[i] == 0) break;  
        if (koren.kljucevi[i] == kljuc) return koren;  
        if (koren.kljucevi[i] < kljuc) pozicija = i+1;  
        else pozicija = i;  
    }  
    return Pretrazi(koren.sinovi[pozicija], kljuc);  
}
```

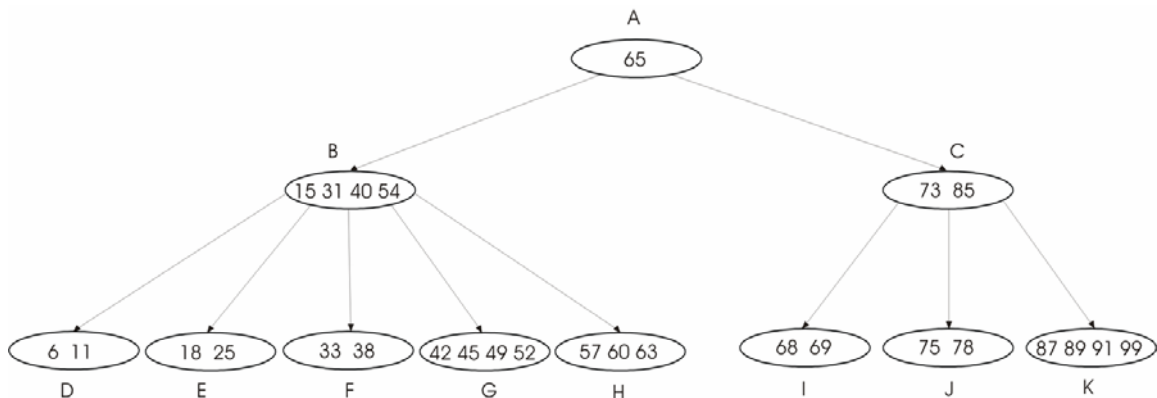
## В СТАБЛА

Стабла  $m$ -арног претраживања су слабо попуњене и небалансиране структуре и самим тим нису најбоље решење за претраживање. Из овог разлога се уводи В стабло, које гарантује да поменутих проблема неће бити приликом претраживања.

В стабло је VST са следећим особинама:

- корен, ако није лист, има најмање два подстабла
- чворови гранања имају најмање  $\lceil m/2 \rceil$  подстабала
- листови имају најмање  $\lceil m/2 \rceil - 1$  кључева (попуњености)
- сви листови су на истом нивоу (балансираности)

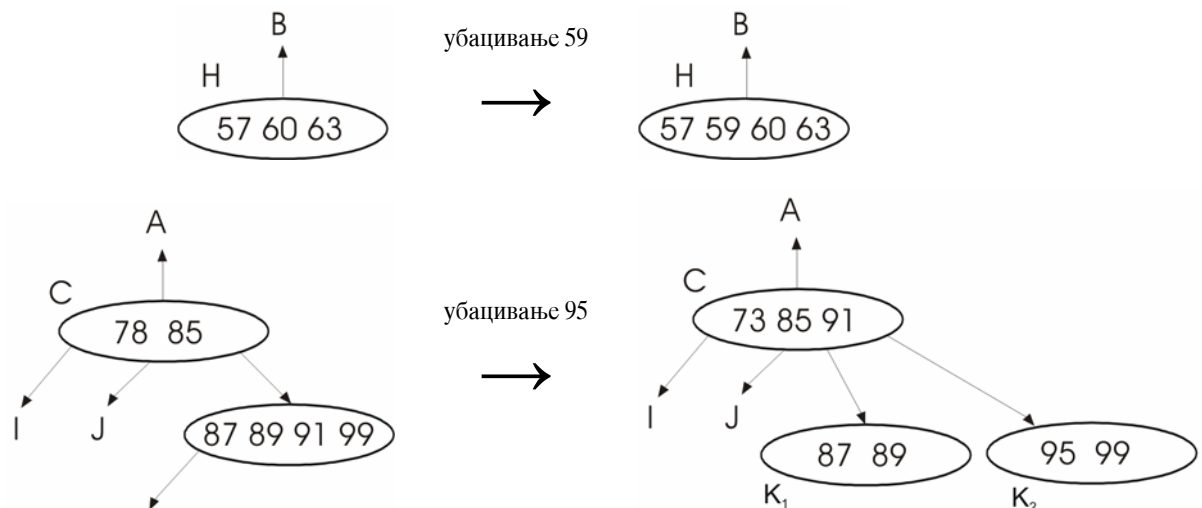
В стабла гарантују 50% попуњеност (најчешће 70 - 80%) и балансираност уз релативно јефтину цену одржавања. Претраживање, као и код VST стабала може да се заврши на било ком нивоу, а неуспешно увек у листовима..

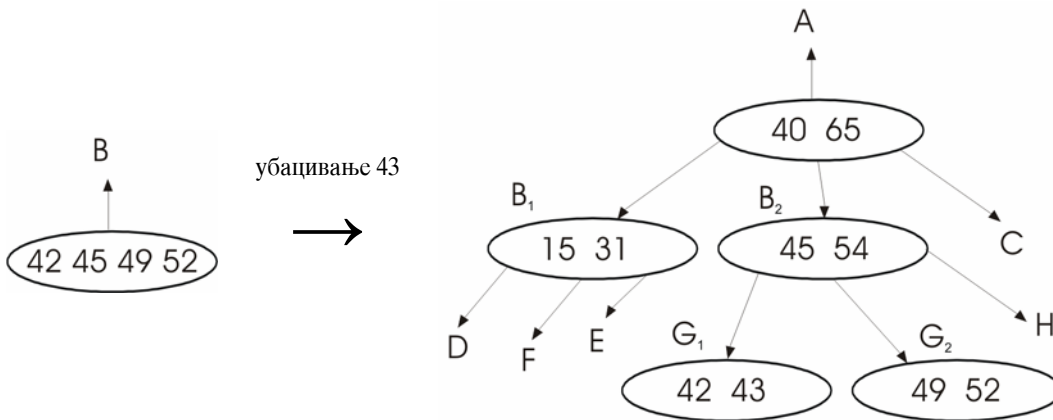


Алгоритам за уметање чвора у В стабло такође гарантује 50% попуњеност и балансираност стабла. Само уметање се врши на следећи начин:

- 1) покушава се уметање чвора у лист, а ако лист није пун врши се уметање кључа у лист уз померање већих кључева за по једно место
- 2) уколико је лист пун, врши се прелом, тако да мањи кључеви иду у стари чвор (лист у који је покушано уметање), већи кључеви иду у нови чвор, а средњи кључ мигрира у оца.
- 3) ако је и отац пун, поступак се понавља све док се евентуално корен не преломи (стабло расте top-down, обрнуто од  $m$ -арних)

Примери:



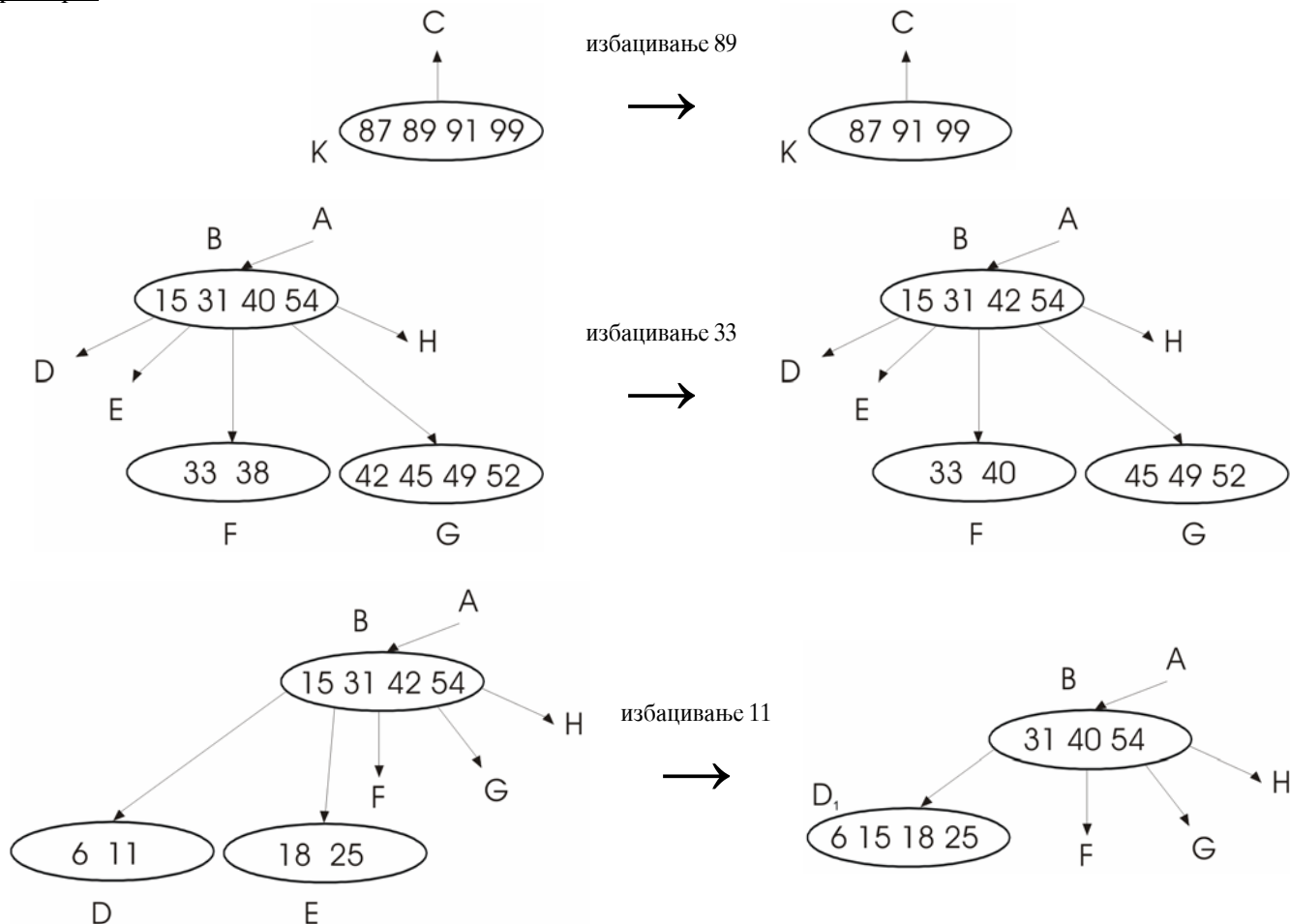


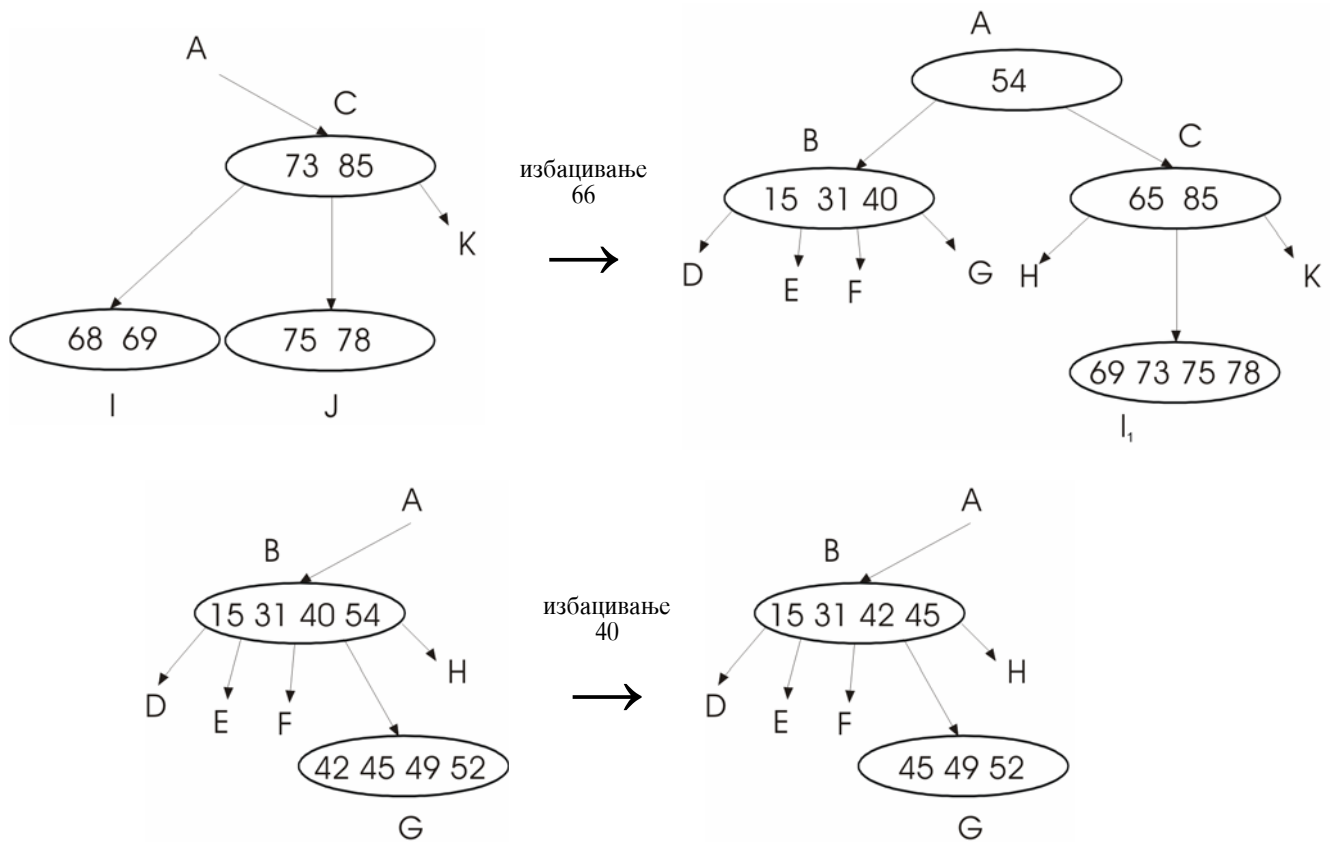
За разлику од стабла  $m$ -арног претраживања, B стабло расте одозго на горе, па су обично пунију чворови на доњим нивоима (мало дуже претраживање), али су перформансе загарантоване.

Алгоритам за брисање кључа из B стабла је следећи:

- 1) уклања се кључ и ако остане бар  $\lceil m/2 \rceil - 1$  кључева врши се само померање
- 2) ако не остане довољно кључева, проба се преузимање једног кључа од десног или левог брата, посредно преко низа (слично ротацији)
- 3) ако је позајмица од браће немогућа, врши се спајање: **лисџ + леви (или десни) браћ + раздвојени кључ из оца**
- 4) уколико се врши уклањање кључа из чвора гранања, а не из листа, кључ мења место са својим следбеником и проблем се своди на брисање кључа из листа

Примери:





Перформансе брисања су исте као код уметања кључева. Могућа је оптимизација брисања без физичког уклањања (помоћу флегова). Искоришћење простора код В стабла је око 70%. В стабло је pogodно као начин динамичког организовања података на екстерној меморији.

## В\* СТАБЛА

Прелом чвора у В стаблу приликом уметања кључева је скупа операција, у смислу перформанси, па је зато настала идеја да се тај прелом што више одложи. То се постиже редистрибуцијом кључева међу браћом, тј. уместо да се преломи чвор, гледају се леви и десни брат и уколико је могуће, умеће се у њих. Тако је настало В\* стабло, које је стабло  $m$ -арног претраживања са следећим особинама:

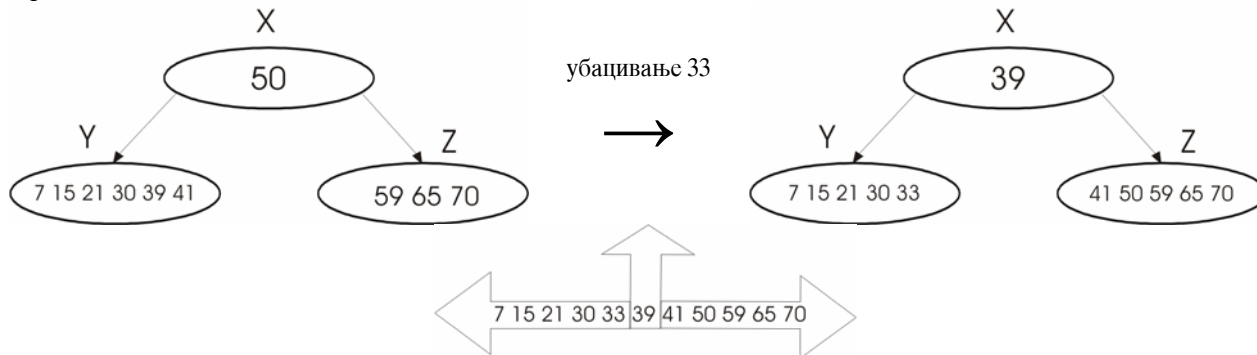
- сваки чвор, осим корена, има највише  $m$  подстабала
- сваки чвор, осим корена и листова, има најмање  $(2m-1)/3$  подстабала (форсира се  $2/3$  попуњеност)
- корен има 2 до  $2 \lfloor (2m-2)/3 \rfloor + 1$  подстабала
- сви листови су на истом нивоу
- чвор, ако није лист, има са  $k$  подстабала  $k-1$  кључева

Из списка особина се види да је попуњеност чворова двотрећинска ( $2/3$ ). Претраживање на кључ је исто као у В стаблу.

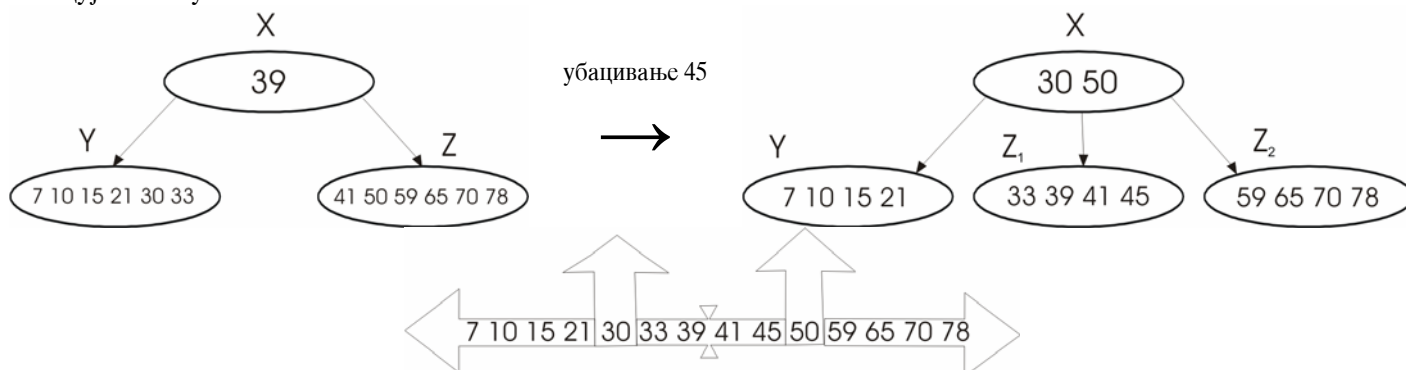
Алгоритам за уметање чвора у В\* стабло:

- 1) када је лист пун, покушава се **ипреливање** са десним или левим братом, тако што се сви кључеви из левог и десног брата, раздвојни кључ и кључ који се умеће ( $m-1+j+1+1$ ) ставе у уређени низ. Затим ће се мањих  $\lfloor (m+j+1)/2 \rfloor$  кључева задржати у чвору који је био пун, а преостали већи кључеви ће ићи у десног брата.
- 2) ако преливање не успе, врши се прелом 2 чвора (полазни чвор и његов брат) на 3 чвора, тако што у први иде  $\lfloor (2m-2)/3 \rfloor$  кључева, у други  $\lfloor (2m-1)/3 \rfloor$  кључева, а у трећи  $\lfloor 2m/3 \rfloor$  кључева, док у чвор родитељ иду два раздвојна кључа
- 3) овај принцип се пропaгира до корена, уколико је то потребно

### Примери:



Убацујемо кључеве 10 и 78:



Приликом брисања кључа врши се спајање 3 чвора у 2 (ако је потребно). У односу на В стабла, В<sup>+</sup> стабла имају бољу попуњеност, мању висину и ефикасније операције.

## **В<sup>+</sup> СТАБЛА**

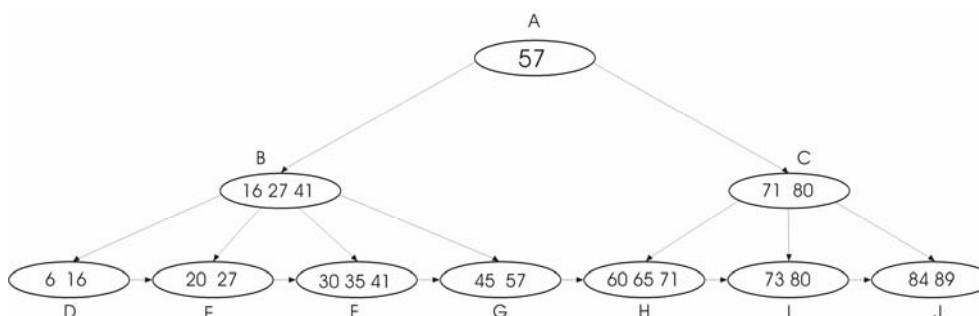
Проблем и главни недостатак В и В<sup>+</sup> стабала је тај, што је секвенцијални приступ кључевима отежан или немогућ (структура није погодна), а у неким случајевима је то веома потребно. Зато је настало В<sup>+</sup> стабло, које омогућава, поред директног приступа и секвенцијални приступ кључевима. Основни принцип је убрзање рада у системима код којих време приступа чвору далеко превазилази време обраде чвора.

В<sup>+</sup> стабло им следеће особине:

- сваки кључ раздваја два показивача, али нема физичких адреса записа уз кључ
- корен може имати минимално два подстабла, а остали чворови од  $\lfloor m/2 \rfloor$  до  $m$
- кључеви у оквиру чвора су уређени ( $K_i < K_j$  за свако  $i < j$ )
- у сваком подстаблу за кључ важи  $K_i < K_x \leq K_{i+1}$

Листови имају следећа својства:

- сваки лист има најмање  $\lfloor m/2 \rfloor$  кључева у уређеном поретку и уз кључеве се налазе физичке адресе записа
- сви листови су на истој дубини и увезани су у уланчану листу по поретку К



**Индексни део** је чвор А са чворовима В и С, а **присвојни део** је на пример чвор I.

Сваки кључ из индексног дела се реплицира у листу као највећи у левом подстаблу. В\* стабла имају јако ефикасан секвенцијални приступ (улево до најмањег листа, па онда по уланчаној листи), налажење следбеника (у истом или суседном листу), као и приступ записима са кључевима у задатом опсегу. Операције су сличне као у В стаблу.

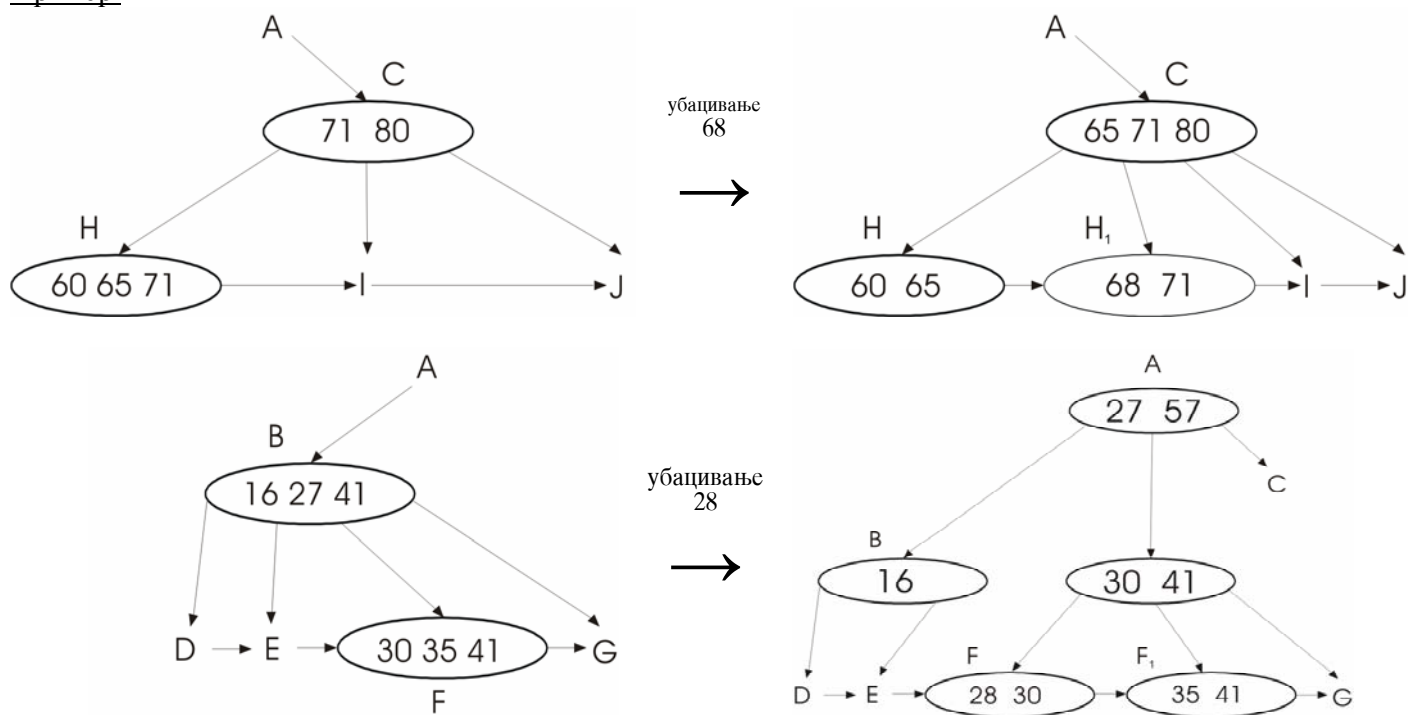
#### Претраживање:

- и успешно и неуспешно претраживање се завршавају у листу
- када се наиђе на кључ у индексном делу иде се у лево подстабло и наставља се претрага
- перформансе су загарантоване

#### Алгоритам за уметање кључа:

- 1) уколико, код уметања у лист, лист није пун, кључ се само убаци у лист тако да структура и даље остане уређена
- 2) ако је лист пун, десиће се прелом, тако да мањи кључеви и средњи кључ (позиција  $\lceil m/2 \rceil$ ) остају у старом листу, већи кључеви иду у нови лист, а средњи кључ ( $\lceil m/2 \rceil$ ) се реплицира у оцу
- 3) ако се прелом деси у оцу или било ком другом чвору у индексном делу, он ће се решити исто као у В стаблу

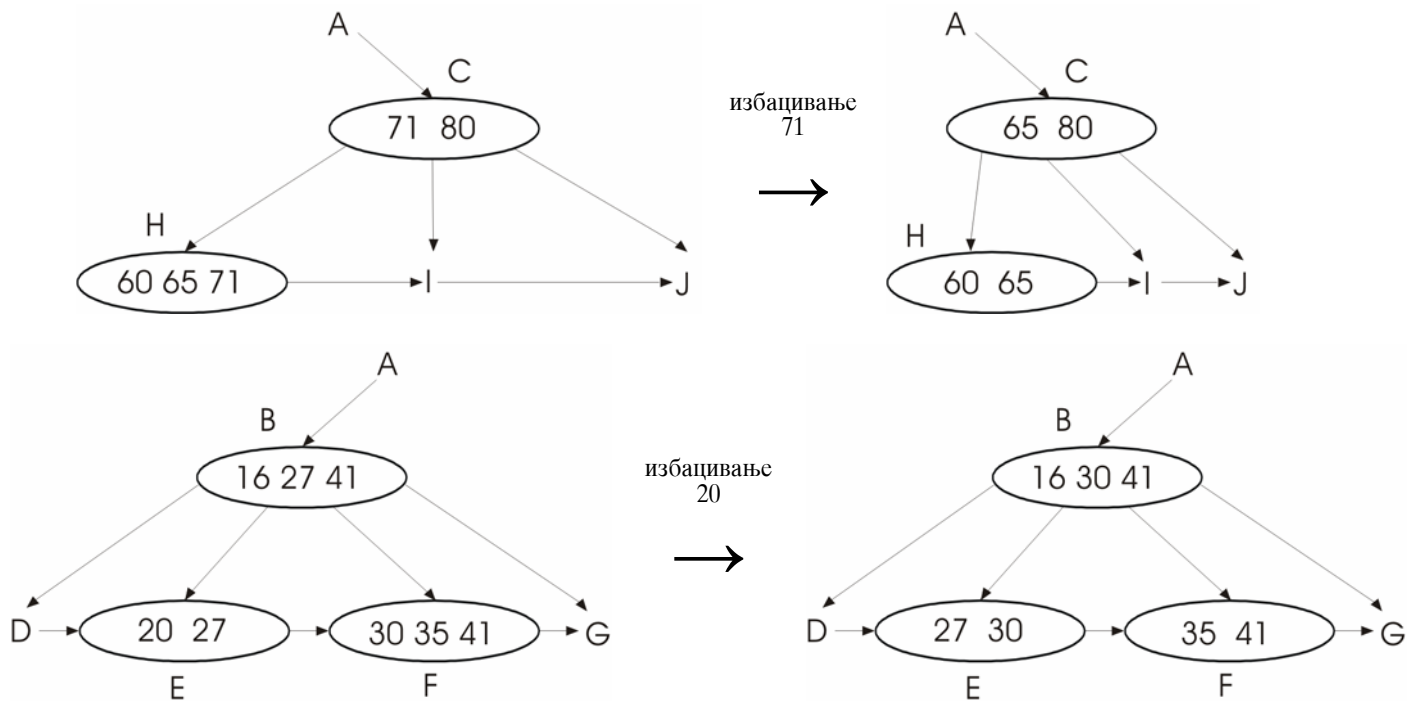
#### Пример:



#### Алгоритам за брисање кључа:

- 1) кључ се обавезно уклања из листа
- 2) ако је кључ најдеснији, треба га уклонити из чвора гранања тако што га замењује претходник
- 3) ако чвор падне испод минимума, проба се преузимање од браће, а ако то не успе врши се спајање





Попуњеност чворова је слична као у В стаблу с тим што је за исту величину чвора већи степен гранања (чвор уз кључеве садржи или показиваче на подстабла или на записе, а не оба као у В стаблу). Структура чвора у индексном делу је иста као структура чвора у приступном делу. Примена В<sup>+</sup> стабала је у индексно-секвенцијалним датотекама.

### **ЗАДАЦИ #4 (Стабла претраживања)**

1) Написати методу која враћа ниво чвора BST стабла који садржи елемент који се тражи.

```
public static int VратиNivoCvoraBST(Cvor koren, int data) {
    int nivo = 0; // inicijalno je to nivo korena
    Cvor cvor = koren;
    while (koren != null) {
        if (cvor.data == data) return nivo;
        if (cvor.data < data) cvor = cvor.levi;
        else cvor = cvor.desni;
        nivo++;
    }
    return nivo;
}
```

2) Написати методу која штампа садржај свих чворова AVL стабла у опадајућем редоследу.

```
public static void StampajAVLOpadajuce(Cvor koren) {
    if (koren != null) {
        System.out.println(koren.data);
        StampajAVLOpadajuce(koren.desni);
        StampajAVLOpadajuce(koren.levi);
    }
    return;
}
```

3) Написати методу која ће одштампати садржај свих чворова на путањи од корена до задатог чвора AVL стабла целих бројевва.

```
public static void StampajDo(Cvor koren, Cvor cvor) {
    if (koren == null) return;
    Cvor tmp = null;
    System.out.println(koren.data);
    if (koren.data < cvor.data) tmp = koren.levi;
    else tmp = koren.desni;
    while (tmp != cvor) {
        System.out.println(tmp.data);
        if (tmp.data < cvor.data) tmp = tmp.levi;
        else tmp = tmp.desni;
    }
}
```

4) Написати програм који убацује нови елемент у BST и јавља да ли је стабло AVL.

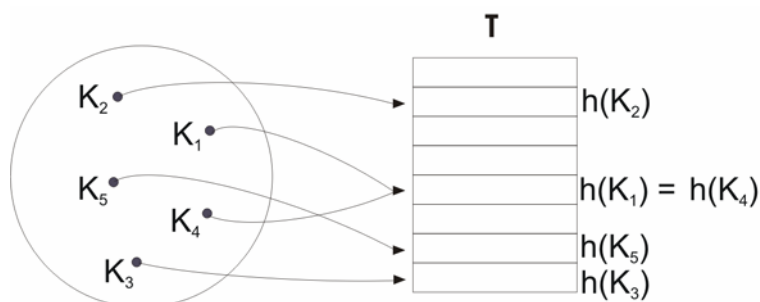
```
// metoda koja vraca visinu stabla/podstabla
private static int visina(Cvor cvor) {
    if (cvor == null) return 0;
    return 1 + Math.max(visina(cvor.desni), visina(cvor.levi));
}
// metoda koja proverava da li je stablo balansirano
private static boolean balansirano(Cvor koren) {
    if (koren == null) return true;
    if (Math.abs(visina(koren.levi) - visina(koren.desni)) <= 1)
        return balansirano(koren.levi) && balansirano(koren.desni);
    return false;
}
// metoda koja umece cvor
public static boolean Insert(Cvor koren, int data) {
    Cvor cvor = new Cvor(data);
    if (koren == null) { koren = cvor; return true; }
    Cvor tmp = koren, roditelj = null;
    while (tmp != null) {
        if (data > tmp.data) tmp = tmp.desni;
        else tmp = tmp.levi;
        roditelj = tmp;
    }
    if (data > roditelj.data) roditelj.desni = cvor;
    else roditelj.levi = cvor;
    return balansirano(koren);
}
```

## ХЕШИРАЊЕ - Hashing

Шта би било идеално претраживање? Наравно, када би постојао директан приступ запису на основу кључа без поређења са другим кључевима, тј. перформансе не би зависиле од броја кључева у скупу. Идеално претраживање је могуће остварити са довољно великом табелом која би морала да покрије све случајеве, што није практично за природне кључеве (пр: ако је кључ матични број, табела мора имати  $10^{19}$  поља).

Проблем непрактичности табеле решава се увођењем функције која трансформише кључ у цео број, који приступа опсегу индекса у табели (идеално би било да сваки кључ добије јединствену позицију у табели). Оваква функција, која врши мапирање кључева у опсег индекса, назива се **хеш функција**, а поступак се зове **хеширање**.

Нека је  $T[i]$ ,  $0 \leq i \leq n-1$  хеш табела са  $n$  улаза и нека кључеви  $K$  припадају неком скупу  $S$  (кеш). Ако је  $h$  нека хеш функција, тада се матична адреса кључа  $i$  добија као:  $i = h(K)$ .

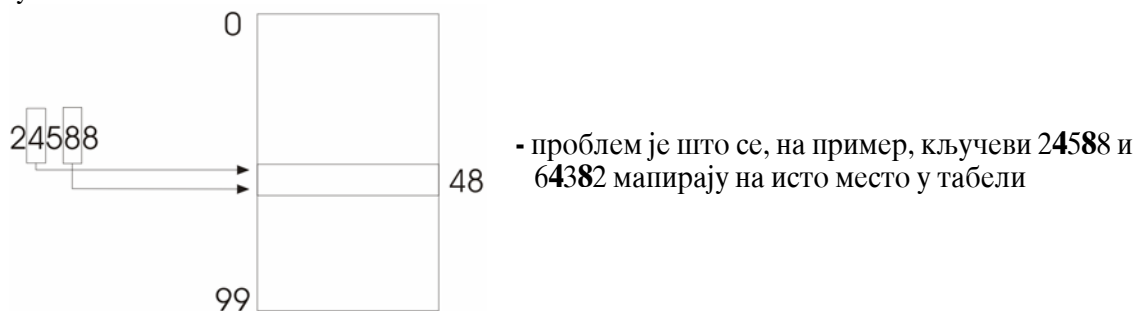


Задатак хеш функције је да компресује скуп кључева на мањи опсег у табели. Два кључа не могу бити смештена на исто место у табели ( $K_1$  и  $K_2$ ), па овај случај представља проблем који се назива **колизија**, а та два кључа су **синоними** (скуп синонима је **класа еквиваленције**).

Пожељне особине хеш функција су **униформност** (за произвољан кључ  $K$  постоји једнака вероватноћа да се он мапира у било коју матичну адресу -  $P(i=h(K)) = 1/n$ ,  $0 \leq i \leq n-1$ ) и **одржавање њорейка** (секвенцијални приступ, тј.  $h(K_i) > h(K_j)$  за  $K_i < K_j$ ). С обзиром да су ове особине често контрапродуктивне, узима се само услов што боље униформности.

Код технике хеширања треба изабрати функцију која је ефикасна, као и ефикасан метод за разрешавање колизије. Хеш функција је ефикасна ако је једноставна (брзо се израчунава) и ако је униформна (што ређе колизије). Кључеви су најчешће нумерички, али могу бити и алфанумерички и алфабетски.

**Метод екстракције** представља метод којим се издваја довољан број цифара из кључа за одређивање његове позиције у табели:



Пожељно је да хеш функција зависи од свих знакова кључа, као и њихових позиција (пр: да би број 12345 био различит од 54321). Хеш функције могу бити:

- независне од расподеле кључева (чешће)
- зависне од расподеле кључева (могуће за унапред познату расподелу кључева)

## НЕЗАВИСНЕ ХЕШ ФУНКЦИЈЕ

### Метод дељења

Резултат ове хеш функције је остатак при дељењу целобројног кључа неким бројем  $n$ , који је већи или једнак величини хеш табеле:  $h(K) = K \bmod n$ . Ово је једноставан и често коришћени метод.

Није препоручљиво да  $n$  буде паран број, зато што мапира парне у парне и непарне у непарне бројеве што не ваља за претежно парне/непарне бројеве. Такође није препоручљиво да  $n$  буде степен броја 2 или 10 јер не зависи од свих цифара кључа ( $K \bmod 100$  се своди на само последње две цифре). Исто важи и за случај кључева конгруентних по модулу  $d$ , када  $n$  не треба да буде узајамно прост број са  $d$  (за конгруентност по модулу 5 и дилац 75, кључеви се мапирају на само 15 позиција). У пракси се препоручује да  $n$  буде прост број, не превише близу степена броја 2.

### Метод множења (Мултипликативна хеш функција)

Функција изгледа овако:  $h(K) = \lfloor n \cdot (c \cdot K \bmod 1) \rfloor$ ,  $0 < c < 1$ , где је  $c$  реална константа. Значи, узимају се само цифре иза децималне тачке и множе се са величином табеле.

Избор  $n$  није критичан (може и број 2), а за  $c$  би било најбоље да буде златни пресек ( $c \approx 0.61803$ ).

### Метод средине квадрата

Овај метод функционише тако што се нумеричка репрезентација кључа помножи сама са собом, па се са средњих позиција (јер све цифре учествују у њиховом добијању) узме онолико цифара колико је потребно за мапирање у табелу.

Пример:     $K = 5894$      $5894 * 5894$   
                                  23576  
                                  53046  
                                  47152  
                                  29470  
                                  34739236  
                                  [39]

### Метод склапања

Склапање се заснива на деоби кључа на делове исте дужине, која одговара броју цифара потребних за адресирање табеле и њиховом сабирању, са игнорисањем преноса.

Пример:     $K = 19653014$     19    19  
                                  65    56  
                                  30    30  
                                  14    41  
                                  128    146  
                                  [28]    [46]  
                                  са померањем    са обртањем

### Метод конверзије основе

Овај метод се заснива на прављење новог кључа. За кључ у систему са основом  $p$  прави се нови са основом ( $q > p$ ), где  $q$  треба да буде узајамно прост број са  $p$ .

Пример:    за 6154 ( $p = 10$ )  $\rightarrow$  13420 ( $q = 13$ )    - узима се потребан број цифара са десне стране или се примењује дељење  
                                   $6 \cdot 13^3 + 1 \cdot 13^2 + 5 \cdot 13^1 + 4 \cdot 13^0 = 13420$

## Метод алгебарског кодовања

Метод се заснива на аритметици по модулу 2 са полиномима и предложен је за хардверску имплементацију. Кључ са бинарном представом од  $r$  бита ( $K_{r-1} \dots K_0$ ) се третира као полином  $K(x)$ , где важи да је:  $K(x) = K_{r-1}x^{r-1} + \dots + K_0$ . Ако је табела реда  $m$ , изабере се полином реда  $m$  ( $P(x) = x^m + P_{m-1}x^{m-1} + \dots + P_0$ ) и израчуна се  $K(x) \bmod P(x) = h_{m-1}x^{m-1} + \dots + h_0$ . Коефицијенти добијеног полинома  $h_{m-1} \dots h_0$  представљају резултат хеш функције.

## ЗАВИСНЕ ХЕШ ФУНКЦИЈЕ

### Метод анализе цифара

Овај метод анализира скуп нумеричких вредности свих кључева и прави табелу са бројем појављивања појединих цифара на одређеним позицијама. Затим селекује онолико позиција колико је потребно за адресирање хеш табеле, при чему се бирају колоне са најмањим осцилацијама појављивања различитих цифара (најуниформније).

### Директна кумулативна функција расподеле

$F_z(x) = P(z \leq x)$  за познату расподелу вредности кључева из  $S$  (случајна променљива  $z$  није већа од неке вредности  $x$ ). Уколико скуп  $S$  садржи  $m$  кључева  $K_1 < K_2 < \dots < K_m$ , онда ова функција има дискретну унутрашњу расподелу на  $1/m, 2/m, \dots, 1$ , што значи да је  $P(F_z(z) \leq i/m) = i/m$  за  $0 \leq i \leq m$ , из чега следи да је:

$$F_z(K_1) = 1/m, F_z(K_2) = 2/m, \dots, F_z(K_m) = 1$$

Пошто је циљ да  $h(k)$  буде униформна у опсегу табеле, тј.

$$P(h(K) = i) = 1/n \text{ за } 0 \leq i \leq n-1 \rightarrow h(K) = \lceil nF_z(K) \rceil - 1$$

Пример: кључеви 6, 7, 12, 14, 15, 16 у табели са 8 улаза

	$T$
$h(K_1) = \lceil 8 \cdot 1/6 \rceil - 1 = 1$	0
$h(K_2) = \lceil 8 \cdot 2/6 \rceil - 1 = 2$	1
$h(K_3) = \lceil 8 \cdot 3/6 \rceil - 1 = 3$	2
$h(K_4) = \lceil 8 \cdot 4/6 \rceil - 1 = 5$	3
$h(K_5) = \lceil 8 \cdot 5/6 \rceil - 1 = 6$	4
$h(K_6) = \lceil 8 \cdot 1 \rceil - 1 = 7$	5
	6
	7

- ова функција има ретку особину да одржава кључеве у сортираном поретку у зависности од вредности кључева

### Апроксимација - Сегментна линеарна функција

У многим случајевима дискретна кумулативна расподела кључева није позната па је потребно проценити и апроксимирати. Ако је дат скуп од  $m$  целобројних кључева вредности у интервалу од  $a$  до  $d$ , тада овај интервал треба поделити на  $j$  једнаких подинтервала дужине  $l = (d-a)/j$ . За сваки кључ  $K$ , редни број подинтервала  $i$  се одређује као  $i = 1 + \lfloor (K-a)/l \rfloor$ .

$N_i$  — број кључева у подинтервалу  $i$  (фреквенција)

$G_i$  — број кључева у подинтервалима 1 до  $i$  (кумулативна фреквенција)

На основу  $N_i$  и  $G_i$  се одређује линеарна апроксимација кумулативне функције расподеле за подинтервал  $i$  као:

$$P_i(K) = (G_i + ((K-a)/(l-i)) \cdot N_i) / m$$

На крају, хеш функција за кључ К у подинтервалу  $I_i$  се добија као:

$$I_i(K) = \lceil n \cdot P_i(K) \rceil - 1 \text{ за } 1 \leq i \leq j$$

**Пример:** Нека је дато  $m = 20$  кључева у опсегу од 0 до 100 и нека су подељени у пет интервала. Тада је дужина једног подинтервала  $l = 20$ , а анализом кључева су добијене вредности  $N_i$  и  $G_i$ .

$i$	$N_i$	$G_i$
1	2	2
2	0	2
3	5	7
4	9	16
5	4	20

Уколико је дата нека хеш табела са  $n = 25$  улаза и кључ  $K = 51$ , прво се израчуна одговарајући подинтервал:

$$i = 1 + \lfloor (51 - 0) / 20 \rfloor = 3$$

а затим:

$$I_3(51) = \lceil 25 \cdot (7 + ((51 - 0) / 20 - 3) \cdot 5) / 20 \rceil - 1 = 5$$

Дакле, иако је вредност кључа 51 око половине опсега вредности кључева, он се смешта на адреси 5 јер су кључеви на већим вредностима много вероватнији.

## РАЗРЕШАВАЊЕ КОЛИЗИЈА

Број колизија се увек може смањити повећањем табеле, али се тада лоше користи простор, јер је попуњеност табеле мања. **Фактор попуњености табеле**  $\alpha = m/n$  представља однос заузетих улаза табеле и целокупне величине табеле.

Најчешће методе разрешавања колизија су **отворено адресирање** и **уланчавање**.

### Отворено адресирање

Овај принцип функционише тако што, уколико је адреса кључа заузета, тражи се нека друга локација у табели. **Испитни низ** је низ адреса у табели које се проверавају при уметању или претраживању (пожељно је да буде пермутација - 0 до  $n-1$ ). Генерисање испитног низа се зове **поновно хеширање (rehashing)**.

Проблем код брисања кључа је то што се прекида испитни низ. Тај проблем се може решити постављањем флегова DELETED или селективним померањем испитног низа. **Варијанте отвореног адресирања су:**

- линеарно претраживање
- случајно претраживање
- квадратно претраживање
- двоструко хеширање

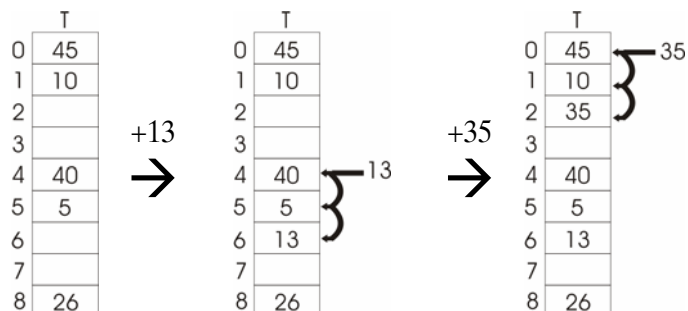
#### Линеарно претраживање

Приликом колизије се генерише испитни низ сукцесивних локација, а када се дође до краја табеле, враћа се на почетак.

$$h_i(K) = (h_0(K) + i) \bmod n ; i=1,2,\dots,n-1$$

$$\text{односно } h_i(K) = (h_{i-1}(K) + 1) \bmod n$$

**Пример:** у Т убацујемо кључеве 13 и 35



При брисању кључева могуће је секвенцијално померање. За брисање на адреси  $i$  тражи се прва слободна адреса  $j$ . Провера почиње од адресе  $i+1$  до  $j$  и ако то није матична адреса кључа, могуће је померање. Проблем линеарног претраживања је примарно груписање тј. стварање група заузетих локација за већу попуњеност. Ово доводи до неједнаке вероватноће попуњавања слободних локација. Примарно груписање настаје када за кључеве  $K_1$  и  $K_2$  важи:

$$h_{i+1}(K_1) = h_r(K_2) \text{ за } r = 0, 1, 2, \dots$$

То значи да испитни низ за  $K_2$  почиње од неке адресе у испитном низу за кључ  $K_1$ , па се од тада испитни низови за  $K_1$  и  $K_2$  поклапају.

#### Случајно претраживање

$$h_i(K) = (h_0(K) + P_i) \bmod n ; i = 1, 2, \dots, n-1$$

$P_i$  је псеудониз као једна пермутација бројева из опсега од 1 до  $n-1$ . Један од ефикаснијих начина за генерисање ове суме је помоћу помераачких регистара (што није могуће у програмском језику Java).

#### Квадратно претраживање

$$h_i(K) = (h_0(K) + r i^2) \bmod n ; i = 1, 2, \dots$$

Испитни низ се генерише преко квадрата броја неуспелих покушаја или без квадрирања. Проблем је што испитни низ не мора садржати све адресе табеле.

Уколико је  $n$  прост број, гарантује се да испитни низ садржи барем  $n/2$  адреса из табеле. Ако је  $n$  прост број облика  $4j+3$  гарантује се да ће испитни низ имати све адресе из табеле (потпун испитни низ).

#### Двоструко хеширање

Секундарно груписање се јавља када два кључа имају исту матичну адресу и за њих се генеришу потпуно исти испитни низови ( $h_i(K_1) = h_i(K_2)$  за  $i = 0, 1, 2, \dots$ ). Двоструко хеширање драстично смањује појављивање секундарног груписања јер користи две независне хеш функције (примарну и секундарну).

$$h_i(K) = (h_0(K) + i \cdot g(K)) \bmod n ; i = 1, 2, \dots, n-1$$

односно

$$h_i(K) = (h_{i-1}(K) + g(K)) \bmod n$$

#### Уланчавање

Два основна проблема која се јављају код отвореног адресирања су сложено брисање кључева и то што се испитни низ не састоји само од синонима. Идеја за решавање овог проблема је уланчавање синонима у листе ван области табеле, што побољшава операције претраживања, уметања и брисања. Дакле, нема поновног хеширања и број кључева није ограничен величином табеле. Једина лоша страна оваквог уланчавања је додатно коришћење меморије.

## СОРТИРАЊЕ

Сортирање представља преуређивање скупа података по неком утврђеном редоследу. Ово је једна од честих активности. Постоји велики број алгоритама за сортирање који, у зависности од ситуације, имају сложеност од  $O(n)$  до  $O(n^2)$ . Циљ сортирања може бити ефикасније претраживање, провера једнакости два скупа података, систематизовани приказ...

Поредак по коме се сортира је одређен вредношћу поља кључа и може бити опадајући и растући. Дакле, кључева  $K_1 \dots K_n$  треба довести у поредак  $K_{p_1} \leq \dots \leq K_{p_n}$ , где је  $p_1 \dots p_n$  једна пермутација од 1 до  $n$ . Сортирање је стабилно ако се поредак истих кључева задржава и у сортираном низу. Према месту операције, сортирања се деле на унутрашња и спољашња.

Перформансе алгоритама сортирања зависе од:

- 1) броја корака
- 2) броја поређења кључева
- 3) броја премештања кључева

Алгоритми сортирања се деле на оне који сортирање врше методом поређења и остале. Сортирање поређењем се искључиво заснива на поређењу кључева и врши се:

- методом уметања
- методом селекције
- методом замене
- методом спајања

## МЕТОДЕ УМЕТАЊА

Сортирање методом уметања функционише по принципу: по један елемент из неуређеног дела умеће се у уређени део.

### Директно уметање - Insertion Sort

```
public static int[] InsertionSort(int[] niz) {  
    for (int i=1; i<niz.length; i++) {  
        int j = i-1;  
        int k = niz[i];  
        int poz = i;  
        while (j >= 0 && niz[j] > k) {  
            poz = j;  
            niz[j+1] = niz[j];  
            j -= 1;  
        }  
        niz[poz] = k;  
    }  
    return niz;  
}
```

Сви елементи лево од  $i$ -тог су уређени. Померамо  $i$ -ти елемент улево све док не дође на своје место у уређеном низу. Сада су првих  $i$  елемената сортирани. Понављамо поступак у **for** петљи.

65	76	6	57	99	27	0	96	6	57	65	76	99	27	0	96
65	76	6	57	99	27	0	96	6	27	57	65	76	99	0	96
6	65	76	57	99	27	0	96	0	6	27	57	65	76	99	96
6	57	65	76	99	27	0	96	0	6	27	57	65	76	96	99



Проблем овог алгоритма је велики број поређења и премештања. Једно од побољшања за велике низове било би бинарно претраживање уређеног дела, али то би смањило само број поређења, не и премештања (за уређење низове даје чак лошије перформансе). Друго побољшање била би једноструко уланчана листа уместо низа, што би смањило број премештања, али захтева додатни простор.

## Shell Sort

Ово је алгоритам који представља побољшање Insertion Sort-a. Другачије се зове уметање са смањењем инкремента `inc`. Он сортира групе елемената на еквидистантном размаку `inc` методом директног уметања, а затим се инкремент смањи, све док, у последњем кораку, не постане 1.

```
public static int[] ShellSort(int[] niz, int[] ninc) {
    int inc = 0;
    for (int i=0; i<ninc.length; i++) {
        inc = ninc[i]; // inc је инкремент, из низа инкремената
        for (int j=inc; j<niz.length; j++) {
            int y = niz[j]; // узет други element из групе
            int k = j-inc; // чува индекс prethodnog elementa групе
            while (k>=0 && niz[k]>y) {
                niz[k+inc] = niz[k];
                k = k-inc;
            }
            niz[k+inc] = y;
        }
    }
    return niz;
}
```

На почетку, алгоритму се даје, као инпут, низ који треба сортирати и низ инкремената, за које се препоручује да буду узајамно прости бројеви. Сложеност алгоритма је  $O(n(\log_2 n)^2)$ .

## МЕТОДЕ СЕЛЕКЦИЈЕ

Сортирање методом селекције има следећи принцип: селекује се најмањи елемент из неуређеног дела и ставља се на крај уређеног дела. Понекада се неуређени део реализује као приоритетни ред ради лакше селекције. Поређење у неуређеном делу не може да почне док се не прикупе сви подаци од низу.

### Директна селекција - Selection Sort

```
public void SelectionSort(int[] niz) {
    int index = 0; // pokazivač na prvi element neuređenog niza
    for (int i=0; i<niz.length; i++) {
        int min = niz[i], ind = i;
        for (int j=index; j<niz.length; j++)
            if (niz[j] < min) {
                min = niz[j];
                ind = j; // ind чува poziciju minimalnog elementa
            }
        int tmp = niz[index];
        niz[index] = min;
        niz[ind] = tmp;
        index++;
    }
}
```

65	76	6	57	99	27	0	96
0	76	6	57	99	27	65	96
0	6	76	57	99	27	65	96

0	6	27	57	99	76	65	96
0	6	27	57	65	76	99	96
0	6	27	57	65	76	99	96

Код метода селекције постоји једна замена и  $n-1$  поређења по кораку. Значи, једини проблем је број поређења. Једно од могућих побољшања је квадратна селекција (врши се селекција по групама и налазе се локални минимуми из којих се налази глобални).

### Сортирање помоћу BST

Елементи неуређеног низа се смештају у бинарно стабло претраживања. Затим се стабло обилази Inorder, што даје сортирани поредак (за случај сортирања у растућем поретку). Због очувања стабилности, исти кључеви се смештају у десно подстабло или се уланчавају у листу.

Сложеност је  $O(n \log_2 n)$ , али није гарантована. Уколико би стабло било организовано као AVL, била би загарантована оваква логаритамска сложеност. Ово сортирање је погодно за динамичке скупове података.

1) генерисање BST стабла :

```

graph TD
    65((65)) --> 6((6))
    65 --> 76((76))
    6 --> 0((0))
    6 --> 57((57))
    57 --> 27((27))
    76 --> 99((99))
    99 --> 96((96))

```

2) Inorder обилазак стабла даје сортирани низ :

0	6	27	57	65	76	96	99
---	---	----	----	----	----	----	----

```

public void BSTSort(int[] niz) {
    Cvor koren = new Cvor(niz[0]);
    for (int i=1; i<niz.length; i++) {
        Cvor tmp = koren;
        Cvor roditelj = null;
        while (tmp != null) {
            if (niz[i] > tmp.data) {
                roditelj = tmp;
                tmp = tmp.desni;
            }
            else {
                roditelj = tmp;
                tmp = tmp.levi;
            }
        }
        if (roditelj.data > niz[i])
            roditelj.levi = new Cvor(niz[i]);
        else roditelj.desni = new Cvor(niz[i]);
    }
    Inorder(koren);
}

```

### HEAP Sort

Проблем сортирања помоћу BST је незагарантована перформанса, због небалансираности. Ово се може исправити такозваним **стаблом селекције**, међутим, недостатак овог стабла је додатни простор и велики број непотребних поређења.

Структура селекције HEAP представља комплетно или скоро комплетно бинарно стабло у коме је отац већи или једнак са оба сина (за опадајуће сортирање је мањи или једнак). Ово стабло се не имплементира физички, већ само логички секвенцијално у низу.

Процес сортирања се врши у два основна корака: генерисање HEAP-а и процесирање HEAP-а. Након тога се Levelorder методом прође кроз стабло да би се добио сортирани низ. Све време се мора пазити на услов да стабло буде комплетно или скоро комплетно и да је чвор отац већи од оба сина (или једнак).

HEAP сортирање гарантује сложеност  $O(n \log_2 n)$ , за разлику од сортирања помоћу BST. За сортирање  $k$  од  $n$  елемената, при чему је  $k \ll n$ , добија се линеарна сложеност.

## МЕТОДЕ ЗАМЕНЕ

Принцип метода замене је замена места два елемента која нису у правилном поретку (примењује се и у другим методама). Познате су методе: Bubble Sort, Quick Sort и Побитно раздвајање.

### Bubble Sort

Овај метод се другачије назива *Директна замена*. Заснива се на замени места два суседна елемента тако да највећи елемент изађе на почетак уређеног дела. Могућа је оптимизација, тако да се замена која се врши са елементом на највишој позицији у пролазу запамти, па се елементи изнад тога не проверавају јер су сортирани. Такође, ако у неком кораку није извршена ниједна замена, низ је већ сортиран.

```
public static int[] BubbleSort(int[] niz) {
    int poz = niz.length-1;
    while (poz != 0) {
        int granica = poz;
        poz = 0;
        for (int i=0; i<granica; i++) {
            if (niz[i] > niz[i+1]) {
                int tmp = niz[i];
                niz[i] = niz[i+1];
                niz[i+1] = tmp;
                poz = i;
            }
        }
    }
    return niz;
}
```

Унапређење овог алгоритма је Shake Sort, који алтернативно мења смер проласка.

### Quick Sort

Ово је рекурзиван метод. Поређења и замене се врше на већој удаљености, а сортирање се врши парцијално. Две партиције раздваја *пивот*, који се налази на првом месту када се низ сортира. Партиције се рекурзивно деле док се не дође до јединичне партиције.

```
public static void QuickSort(int[] niz, int low, int high) {
    int j = partition(niz, low, high); // partition() sortira i vraća sredinu (pivot)
    QuickSort(niz, low, j-high);
    QuickSort(niz, j+1, high);
}
```

Могућа је и итеративна реализација помоћу стека. Просечна сложеност је за 38% гора од најбољег случаја, што је импозантно (најгори случај је  $O(n^2)$ , а просечан  $O(n \log_2 n)$ ).

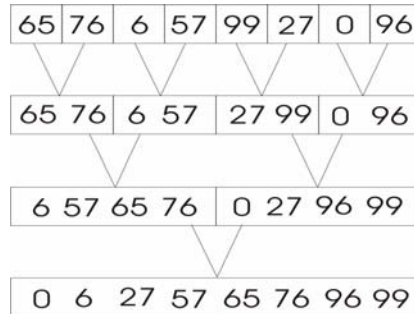
Избор пивота знатно утиче на перформансе, па се зато, да не би дошло до формирања само једне партиције, за пивота често бира случајни елемент, или средњи од њих неколико.

### Побитно раздвајање

Посматра се бинарна репрезентација кључева. Поређење кључева се врши на основу одговарајућег бита (почиње се од најстаријег), тј. ако је  $b_0 = 0$ , елемент иде у доњу, а ако је  $b_0 = 1$  у горњу партицију. Сложеност је иста као и код Quick Sort-a.

## МЕТОДЕ СПАЈАЊА

Код директног спајања, суседни елементи се спајају у уређених двојке, па четворке... Код имплементације са низом је потребан помоћни низ, док код имплементације са листама то није потребно (а и избегавају се премештања). Перформансе су загарантоване -  $O(n \log_2 n)$ .



## МЕТОДЕ ЛИНЕАРНЕ СЛОЖЕНОСТИ

Када су могуће одређене претпоставке о кључевима структуре (низа) која треба да буде сортирана, омогућена је примена метода линеарне сложености.

### Counting Sort

Када је познато да су целобројни кључеви у опсегу 1..k, за сваки кључ се одређује број мањих и једнаких кључева. Метод је стабилан и има сложеност  $O(n)$ .

```
public static int[] CountingSort(int[] niz, int k) {
    int[] c = new int[k]; int[] b = new int[niz.length];
    for (int i=0; i<niz.length; i++) c[niz[i]-1] = c[niz[i]-1]+1;
    for (int i=1; i<k; i++) c[i] = c[i] + c[i-1];
    for (int i=niz.length-1; i>=0; i--) {
        b[c[niz[i]-1]-1] = niz[i];
        c[niz[i]-1] = c[niz[i]-1]-1;
    }
    return b;
}
```

### Radix Sort

Слично дигиталном стаблу, разврставање се врши на основу знакова кључа. Почиње се са најмлађим знаком. Метод је стабилан и сложеност је  $O(k_n)$ , где је k број знакова кључа. Погодан је за предсортирање.

на основу цифре јединица:

Q<sub>0</sub> 0  
Q<sub>1</sub>  
Q<sub>2</sub>  
Q<sub>3</sub>  
Q<sub>4</sub>  
Q<sub>5</sub> 65 75  
Q<sub>6</sub> 6 96  
Q<sub>7</sub> 57 27  
Q<sub>8</sub>  
Q<sub>9</sub> 99  
0 65 75 6 96 57 27 99



на основу цифре десетица:

Q<sub>0</sub> 0 6  
Q<sub>1</sub>  
Q<sub>2</sub> 27  
Q<sub>3</sub>  
Q<sub>4</sub>  
Q<sub>5</sub> 57  
Q<sub>6</sub> 65  
Q<sub>7</sub> 75  
Q<sub>8</sub>  
Q<sub>9</sub> 96 99  
сортиран низ: 0 6 27 57 65 75 96 99

## ИЗВОРИ - ЛИТЕРАТУРА - НАПОМЕНЕ

- СТРУКТУРЕ ПОДАТАКА И АЛГОРИТМИ -скрипта- (Синиша Нешковић, ФОН 2006.)
- АЛГОРИТМИ И СТРУКТУРЕ ПОДАТАКА 1 -скрипта- (Дарко Ђуришић, ЕТФ 2005.)
- АЛГОРИТМИ И СТРУКТУРЕ ПОДАТАКА 2 -скрипта- (Дарко Ђуришић, ЕТФ 2005.)
- ЗАДАЦИ СА ИСПИТНИХ РОКОВА НА ФОН-у
- ПРИМЕРИ И ЗАДАЦИ СА ПРЕДАВАЊА И ВЕЖБИ НА ЕТФ-у

Похваљујем ауторе скрипти из овог предмета, нарочито свог најбољег пријатеља и будућег колегу **Дарка Ђуришића**, који ме је научио најбитнијим чињеницама ове области и који ми је уступио своју литературу на коришћење.

Приликом програмерског рада је веома битно разумети елементарне ствари, што се на ФОН-у јако тешко остварује, с обзиром на збрку предмета на смеру Информациони Системи и Технологије. Зато сам се трудио да елементарне појмове разјасним примерима и напоменама.

Ова збирка података је намењена апсолутно свима који желе да уче о алгоритмима и структурама података, јер садржи велики дијапазom података (више него предавања на ФОН-у и мало мање него предавања на ЕТФ-у). С обзиром на велику количину материјала коју сам сретао сурфујући Интернетом, није било лако направити јединствену скрипту, али сам се трудио да у њој нема грешака, што је и највећи проблем код осталих скрипти доступних на сајтовима везаним за ову област.

**Напомена:** Сви примери кода су рађени у програмском језику Java. Код сам писао ја лично. Уколико има неких грешака у коду, унапред се извињавам и молио бих вас да ми пријавите било какав недостатак на моју електронску адресу: [mdjekic@gmail.com](mailto:mdjekic@gmail.com).

**Реклама 1:** Уколико су вам потребни часови из програмског језика Java, без обзира на обим знања који је потребан, било да су у питању елементарне ствари или озбиљно програмирање, јавите се на e-mail. Цена по договору и у зависности од потреба. Могу објашњавати и АСП, мада мислим да је овај документ сасвим довољан (за ФОН-овце). За сада је поприличан број људи положио програмирање на конто мојих часова. Такође, уколико вам могу нешто помоћи успут, можете послати питање и ја ћу радо одговорити, наравно бесплатно.

**Реклама 2:** Израда било каквих врста програма, web страна и осталих презентација уз договор о цени. За мање корисничке апликације заиста нећу наплаћивати много, док за неке напредније апликације или информационе системе, цена сигурно неће бити мала. Јавите се на e-mail и направимо калкулацију. Моје референце су израда програма за Техничку службу Војномедицинске Академије (ВМА) и неколико web презентација, које нећу наводити из практичних разлога (не рекламирам бивше послодавце, осим по потреби или поручбини).

Милош Ђекић  
Август 2007. године



## ДОДАТАК - РАЗНИ ЗАДАЦИ

1) Уколико је неки String имплементиран као двоструко спрегнута листа, имплементирати методу која ће проверити да ли је задати String палиндром. Палиндром је реч или реченица која се на исти начин чита и с лева на десно и с десна на лево (пр: <Ана воли Милована> или <капак>).

```
public static boolean IsPalindrome(LString lista) {
    CElement e1 = lista.first;
    CElement e2 = lista.last;
    while(true) {
        while (e1.data == ' ') {
            e1 = e1.next;
            if (e1 == e2) return true;
        }
        while (e2.data == ' ') {
            e2 = e2.prev;
            if (e1 == e2) return true;
        }
        if (Character.toLowerCase(e1.data) != Character.toLowerCase(e2.data)) break;
        else {
            e1 = e1.next;
            if (e1 == e2) return true; // ukoliko je paran broj ne-blanko karaktera
            e2 = e2.prev;
            if (e1 == e2) return true; // ukoliko je neparan
        }
    }
    return false;
}
```

2) Написати методу која ће вратити референцу на лист бинарног стабла целих бројева који се налази на највећој дубини.

```
// metoda koja radi posao
private static Object[] Rek(Cvor koren, Cvor ref, int dubina, int max) {
    if (koren == null) {
        Object[] ret = new Object[2];
        ret[0] = ref; ret[1] = new Integer(dubina);
        return ret;
    }
    if (dubina > max) {
        max = dubina;
        ref = koren;
    }
    Object[] levo, desno;
    levo = Rek(koren.levi, ref, dubina+1, max); Integer l = (Integer)levo[1];
    desno = Rek(koren.desni, ref, dubina+1, max); Integer d = (Integer)desno[1];
    if (l.intValue() > d.intValue()) return levo;
    else return desno;
}
// omotačka metoda
public static Cvor VратиCvor(Cvor koren) {
    Object[] ret = Rek(koren, koren, 1, 0);
    return (Cvor)ret[0];
}
```

3) Написати методу која ће иштампати садржај свих чворова бинарног стабла на путањи од корена до најдубљег листа..

```
public static void print(Cvor koren) {
    Cvor cvor = VratiCvor(koren); // prethodni zadatak
    ArrayList lista = new ArrayList();
    while (cvor != null) {
        lista.add(cvor);
        cvor = cvor.roditelj;
    }
    Object[] niz = lista.toArray();
    for (int i=niz.length-1; i>-1; i--) {
        Cvor c = (Cvor)niz[i];
        System.out.println(c.data);
    }
}
```

4) Написати методу која пореди два String-а имплементирана преко једноструко спрегнуте цикличне листе и враћа -1 ако је први String мањи од другог, 0 ако су једнаки и 1 ако је други већи.

```
public static int uporedi(LString s1, LString s2) {
    int b1 = 0; int b2 = 0;
    Element e1 = s1.first; Element e2 = s2.first;
    while (e1.next != s1.first) { b1++; e1 = e1.next; }
    while (e2.next != s2.first) { b2++; e2 = e2.next; }
    if (b1 < b2) return -1;
    if (b1 == b2) return 0;
    else return 1;
}
```

5) Дата је референца на први елемент једноструко спрегнуте листе целих бројева, која је сортирана у опадајућем редоследу. Без промене структуре листе и без коришћења других структура података, имплементирати методу која ће одштампати елементе листе у растућем редоследу.

```
public static void printList(Element prvi) {
    int broj = 1;
    Element tmp1 = prvi;
    while (tmp1.next != null) { broj++; tmp1 = tmp1.next; }
    System.out.println(broj);
    // sada tmp1 pokazuje na poslednji element liste
    Element tmp2 = prvi;
    for (int i=0; i<broj-1; i++) {
        System.out.println(tmp1.data);
        while (tmp2.next != tmp1) tmp2 = tmp2.next;
        tmp1 = tmp2; tmp2 = prvi;
    }
    System.out.println(tmp1.data);
}
```

6) Два стабла су <слична као у огледалу> ако су оба празна или ако је лево подстабло једног идентично десном стаблу другог и обратно. Написати методу која ће проверити да ли су два стабла таква.

```
public static boolean ogledalo(Cvor c1, Cvor c2) {
    if (c1 == null && c2 == null) return true;
    if (c1 != null && c2 != null && c1.data == c2.data)
        return ogledalo(c1.levi, c2.desni) && ogledalo(c1.desni, c2.levi);
    return false;
}
```