

Rândul 1.

1. Pentru a cumpăra cât mai puține cutii trebuie să le cumpărăm pe cele cu număr mare de șuruburi. Vom sorta vectorul ns descrescător, pentru a avea la începutul vectorului valori mari. Dacă ns este sortat, pur și simplu calculăm câte elemente de la început trebuie să adăugăm pentru a avea $s * k$ șuruburi.

funcție șuruburi ($s, k, nrc : \text{intreg}$, $ns : \text{intreg}[]$) este

```
// bubble-sort
```

```
schimb : boolean
```

```
i : intreg
```

```
schimb = true
```

```
cât timp schimb == true executa
```

```
    schimb = false
```

```
    pentru i = 0, nrc-1, 1 executa
```

```
        dacă  $ns[i] < ns[i+1]$  atunci
```

```
            temp : intreg
```

```
            temp =  $ns[i]$ 
```

```
             $ns[i] = ns[i+1]$ 
```

```
             $ns[i+1] = temp$ 
```

```
            schimb = true
```

suruburi : întreg

suruburi = 0

index : întreg

index = 0

cât timp $\text{index} < \text{nr}$ și $\text{suruburi} < s * k$ execută

$\text{suruburi} = \text{suruburi} + \text{nr}[\text{index}]$

$\text{index} = \text{index} + 1$


// index e prima poziție pe care nu am mai adăugat și
// este și numărul de cutii adunate

returnează index

Complexitate :

- bubble_sort : $O(n^2)$

- restul : $O(n)$

} $O(n^2)$ 

2. Cele 2 cicluri pentru:

$$\sum_{i=1}^n \sum_{j=1}^i 1 = \sum_{i=1}^n i = \frac{n \cdot (n+1)}{2}$$

ciclul câțimp:

$$k = n^2$$

$$k > n$$

k scade cu 2

avem aproximativ $\frac{n^2 - n}{2}$ operații

$$\text{Complexitate: } \frac{n^2 + n}{2} + \frac{n^2 - n}{2} \in \Theta(n^2)$$

3. 1. X trebuie să fie mai mare ca 30

X trebuie să fie mai mare decât 37, 40, 43

X trebuie să fie mai mic decât 50, 60

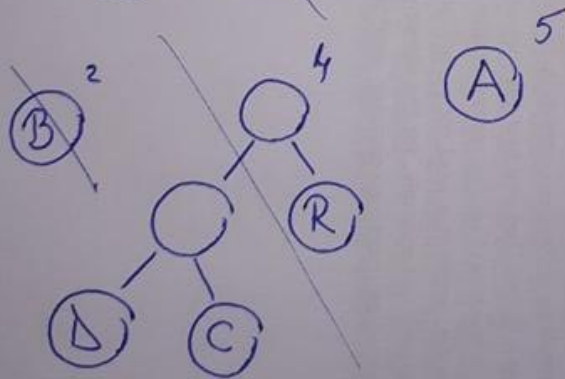
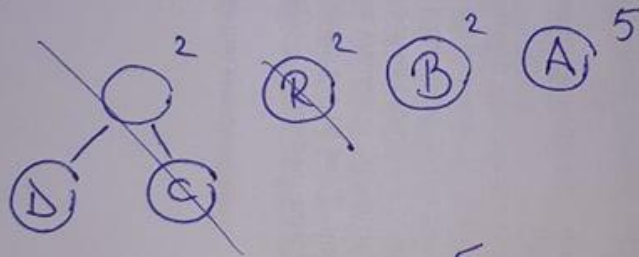
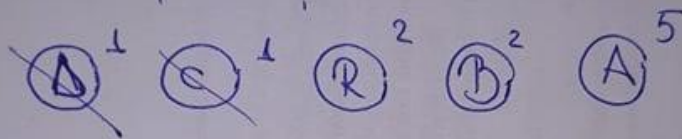
\Rightarrow valori posibile pentru X: 44, 45, 46, 47, 48, 49

32. ABRACADABRA

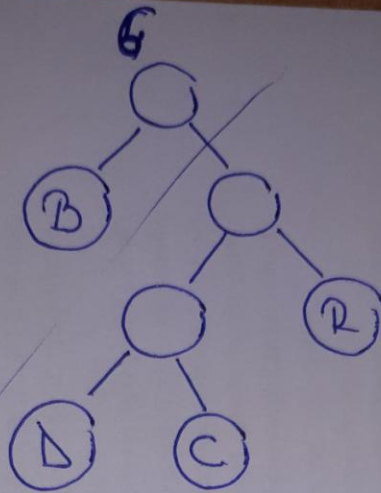
Frecvențe: A - 5
 B - 2
 R - 2
 C - 1
 D - 1



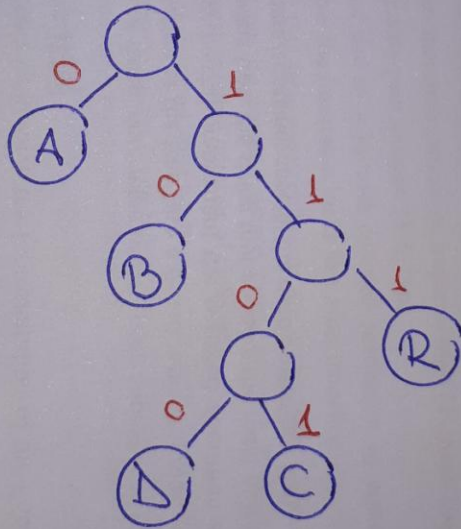
Coadă cu priorități: (Mai multe variante sunt posibile)



5
A



Arborele final



A - 0

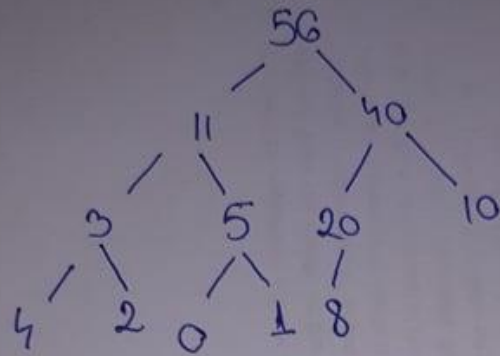
B - 10

D - 1100

C - 1101

R - 111

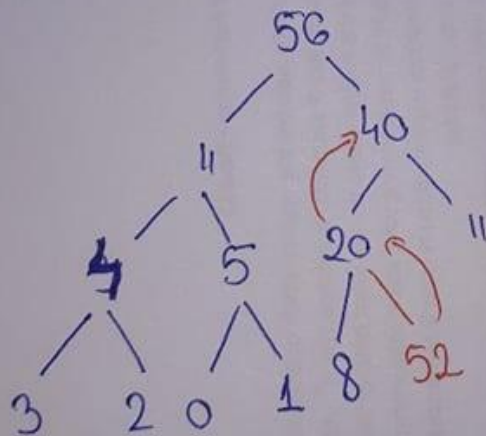
3.3. Desenăm vectorul ca ansamblu



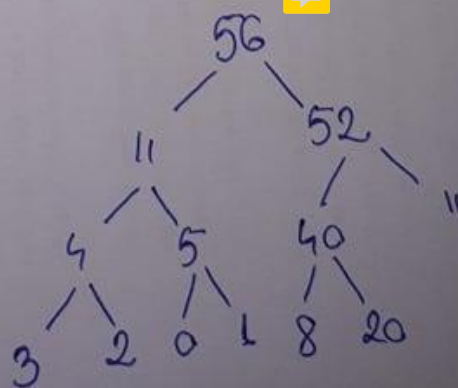
Pare a fi Max-heap
dar avem problema
la 3 și 4.

NU este ansamblu.

Interschimbăm 3 cu 4



Adăugăm 52.



3.4. $m=13$

16	32		42	30		19			136		50	
0	1	2	3	4	5	6	7	8	9	10	11	12

$$d(30, 0) = (30 \div 13 + 0 + 0) \div 13 = 4$$

$$d(19, 0) = (19 \div 13 + 0 + 0) \div 13 = 6$$

$$d(42, 0) = (42 \div 13 + 0 + 0) \div 13 = 3$$

$$d(50, 0) = (50 \div 13 + 0 + 0) \div 13 = 11$$

$$d(16, 0) = (16 \div 13 + 0 + 0) \div 13 = 3 \text{ -occupat}$$

$$d(16, 1) = (16 \div 13 + 1 \cdot 1 + 2 \cdot 1^2) \div 13 = 3 + 1 + 2 = 6 \text{ occupat}$$

$$d(16, 2) = (16 \div 13 + 1 \cdot 2 + 2 \cdot 2^2) \div 13 = 3 + 2 + 8 = 13 \div 13 = 0$$

$$d(136, 0) = (136 \div 13 + 0 + 0) \div 13 = 6 \text{ -occupat}$$

$$d(136, 1) = (136 \div 13 + 1 \cdot 1 + 2 \cdot 1^2) \div 13 = 6 + 1 + 2 = 9$$

$$d(32, 0) = (32 \div 13 + 0 + 0) \div 13 = 6 \text{ -occupat}$$

$$d(32, 1) = (32 \div 13 + 1 \cdot 1 + 2 \cdot 1^2) \div 13 = 6 + 1 + 2 = 9 \text{ -occupat}$$

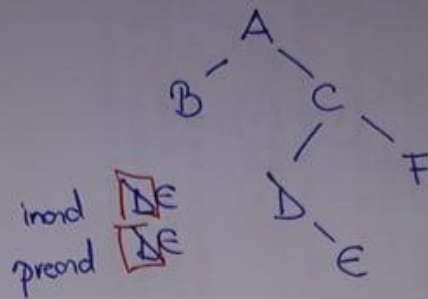
$$d(32, 2) = (32 \div 13 + 1 \cdot 2 + 2 \cdot 2^2) \div 13 = (6 + 2 + 8) \div 13 = 3 \text{ occupat}$$

$$d(32, 3) = (32 \div 13 + 1 \cdot 3 + 2 \cdot 3^2) \div 13 = (6 + 3 + 18) \div 13 = 27 \div 13 = 1$$

4.1.

inordine
preordine

$B \boxed{A} D E C F$
 $\boxed{A} B C D E F$



Frunte: B, E, F Răspuns: b, e, f

4.2.

e. oricâte

Pe fiecare poziție avem o listă înălțurată
în care putem adăuga oricâte elemente.

4.3 e. niciuna

-adăugăm pe ultima poziție, ștergem
de pe ultima poziție, toate operațiile au
complexitate $\Theta(1)$

4.4. Folosim o Stivă:

$\underline{2} \underline{3} \underline{4} * + \underline{6} \underline{2} \underline{1} + / +$

s: ~~2~~ ~~3~~ ~~4~~ 12 14 ~~6~~ ~~2~~ ~~1~~ 8 2 16

e: între 15 și 100

4.5. c. ambele

Cada fiind vidă, elementul adăugat va fi și primul (front) și ultimul (end) nod.

4.6. c \Rightarrow este $\Theta(1)$ la LSI și $\Theta(n)$ la VD

5. MultiDictionar pe LSI. Reprezentare

Nod:

cheie: TCheie

valoare: TValoare

urm: \uparrow Nod

ant: \uparrow Nod

MultiDictionar

prim: \uparrow Nod

ultim: \uparrow Nod

Specificare:

sterge(c, v)

- c: TCheie, v: TValoare

- returnează True dacă a găsit și a șters perechea (c, v), fals dacă perechea (c, v) nu e în

MultiDictionar

funcție șterge (c, v) este:

nodCurrent : ↑ Nod

nodCurrent = this.prim

căttimp nodCurrent != NULL și nodCurrent.cheie ≠ c și
nodCurrent.valoare ≠ v execută

nodCurrent = [nodCurrent].urm

dacă nodCurrent == NULL atunci // nu este în Multidirecțional

returnează false

altfel // ștergem nodCurrent. Poate fi prim și/sau ultim sau de la mijloc.

dacă nodCurrent == this.prim atunci

dacă nodCurrent == this.ultim atunci // un singur
nod

this.prim = NULL

this.ultim = NULL

altfel

this.prim = [this.prim].urm

[this.prim].ant = NULL

altfel dacă nodCurrent == this.ultim atunci

$\text{this.ultim} = [\text{this.ultim}].\text{ant}$

$[\text{this.ultim}].\text{urm} = \text{NULL}$

altfel // de la mijloc

$[\text{nodCurrent}].\text{ant}.\text{urm} = [\text{nodCurrent}].\text{urm}$

$[\text{nodCurrent}].\text{urm}.\text{ant} = [\text{nodCurrent}].\text{ant}$

returnează true

Complexitate: $O(n)$

- caz favorabil (ștergere prim): $\Theta(1)$
- caz ~~defavorabil~~ defavorabil (ștergere ultim): $\Theta(n)$
- caz mediu: $\Theta(n)$

Reprezentare iterator

Iterator MD:

md : MultiDictionary

current : \uparrow Nod

Operații iterator:

Subalg. creează (md : MultiDictionary) este:

$\text{this.md} = \text{md}$

$\text{this.current} = \text{md.prim}$

Subalg. urmator () este:

```
┌   this.current = [this.current].urm  
└
```

functie element () este:

```
┌   cheie = [this.current].cheie  
   val = [this.current].valoare  
   returnează <cheie, val>  
└
```

functie valid () este:

```
┌   dacă this.current == null atunci  
   │   returnează false  
   │  
   │   altfel  
   │       returnează true  
└
```


6. Colectie pe TD + Li

Nod:

elem: TElem

prev: întreg

urm: ↑Nod

Colectie:

m: întreg

elems: ↑Nod[]

adauga(e: TElem)

- se adauga e in Colectie
- nu se returneaza nimic

Subalg. adauga(e: TElem) este:

poz: întreg

poz = hashCode(e) % this.m // sau doar poz = e / this.m

nodC: ↑Nod

nodC = this.elems[poz]

gasit: boolean

gasit = false

// verificam daca avem deja un nod cu e

cat timp gasit == false si nodC != null execută

daca [nodC].elem == e atunci

gasit = true

altfel

nodC = [nodC].urm

// dacă am găsit nod, incrementăm frecvența

dacă găsit == true atunci

[nodCurrent].frecu = [nodCurrent].frecu + 1

altfel // adăugăm nod nou cu frecvența 1

nodNou: ↑ Nod

[nodNou].elem = e

[nodNou].frecu = 1

[nodNou].urm = NULL

dacă this.elems[pos] == NULL atunci

this.elems[pos] = nodNou

altfel

[nodNou].urm = this.elems[pos]

this.elems[pos] = nodNou

Complexitate $O(n)$

Caz favorabil

$O(1)$

—> prim nodul pe prima pos, sau nu sunt noduri la pos

Caz defavorabil $O(n)$ — toate elementele într-o listă

Caz mediu $O(n)$

Rândul 2

1. Dacă vrem diferență minimă, cea mai simplă variantă este să sortăm vectorul și să ne uităm la diferența de înălțime la K copii consecutive din vector.

functie diferență ($k, n: \text{int}$, $\text{inaltimi}: \text{int}[]$) este

// sortăm vectorul înălțimi

aux, index: int

schimbat: boolean

schimbat = true

index = 0

cat timp schimbat == true executa

schimbat = false

pentru index = 0, $n-k$, 1 executa

dacă $\text{inaltimi}[\text{index}] > \text{inaltimi}[\text{index}+k]$ atunci

aux = $\text{inaltimi}[\text{index}]$

$\text{inaltimi}[\text{index}] = \text{inaltimi}[\text{index}+k]$

$\text{inaltimi}[\text{index}+k] = \text{aux}$

schimbat = true

prim, ultim, dif, mindif: int

prim = 0 // indexul primului copil

ultim = $k-1$ // indexul ultimului copil din grupa de K

mindif = $\text{inaltimi}[0]$


```

    ciclul cãtimp ultim < n executã
    dif = înãlțimi[ultim] - înãlțimi[prim]
    datã dif < mindif atunci
    |   mindif = dif
    |   prim = prim + 1
    |   ultim = ultim + 1
    |
    |   returnează mindif
  
```

Complexitate $O(n^2)$

- sortare $O(n^2)$

- ciclul cãtimp după sortare : $\Theta(n)$

2. ciclul cãtimp: $k = n^2$ k este împãrțit la 2 $\Rightarrow \log_2 n^2 = 2 \cdot \log_2 n$

index este nr de repetiți pentru ciclul cãtimp $\Rightarrow 2 \cdot \log_2 n$

ramura dacã $\Rightarrow 2 \cdot \log_2 n$

ramura altfel $\Rightarrow n$

$2 \cdot \log_2 n + \text{Max}(2 \cdot \log_2 n + n) = \log_2 n + n = \Theta(n)$

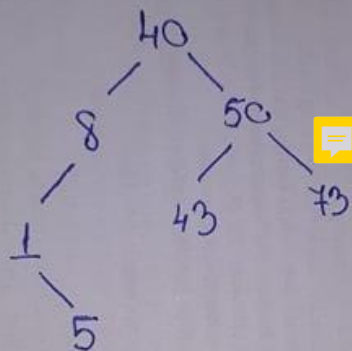
3.1. \in un max-heap

X e mai mic ca 18

X e mai mare ca 6 si 9

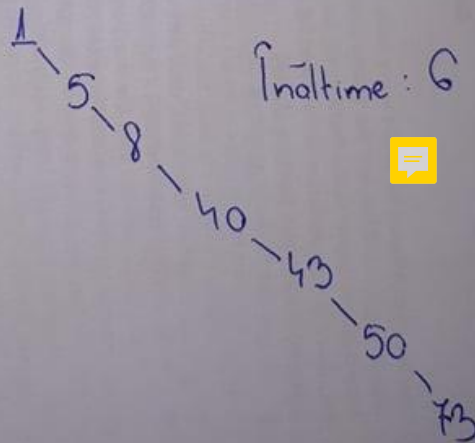
Valori posibile pentru X : 11, 12, 13, 14, 15, 16, 17

3.2.



Înălțime 3

O variantă:



Înălțime: 6

3.3. $d(c) = c \% m$

$$d(7) = 7 \% 9 = 7$$

$$d(19) = 19 \% 9 = 1$$

$$d(66) = 66 \% 9 = 3$$

$$d(3) = 3 \% 9 = 3$$

$$d(23) = 23 \% 9 = 5$$

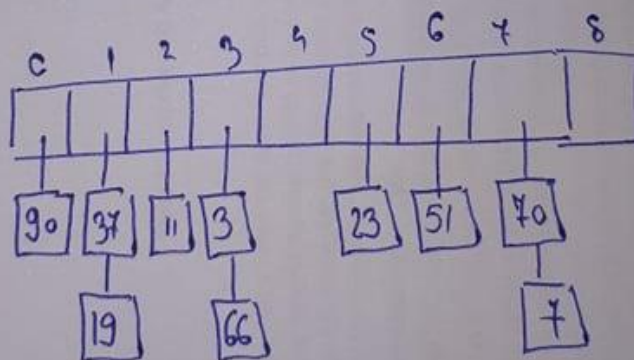
$$d(37) = 37 \% 9 = 1$$

$$d(51) = 51 \% 9 = 6$$

$$d(90) = 90 \% 9 = 0$$

$$d(70) = 70 \% 9 = 7$$

$$d(11) = 11 \% 9 = 2$$

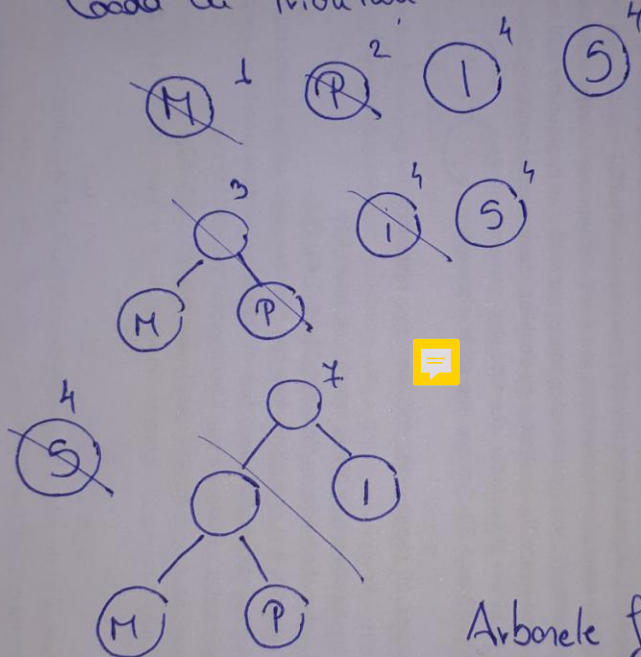


3.4. MISSISSIPPI

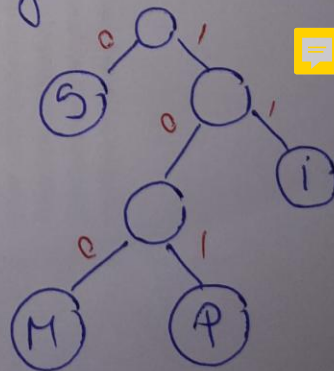
Frecvențe:

M - 1
 i - 4
 S - 4
 P - 2

Cuadru cu Priorități



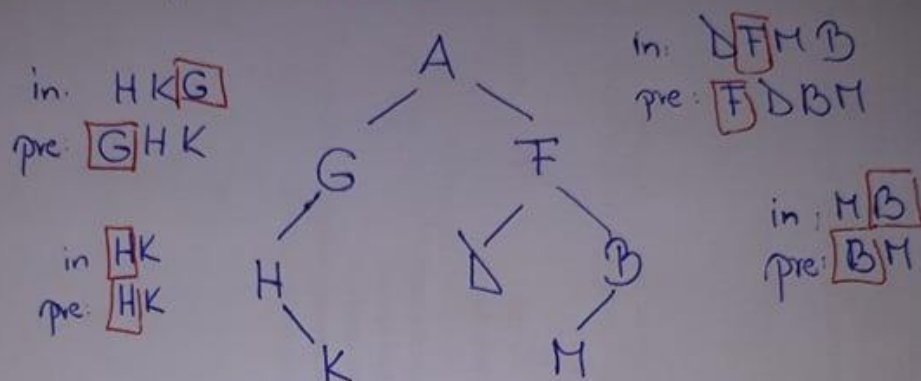
Arborele final



S - 0
 M - 100
 P - 101
 i - 11

4.1. inordine HKGA Δ FM Δ B

preordine Δ AGHK Δ FBM



parcurgerea pe niveluri: AG Δ FH Δ BK Δ M = a.

4.2. b. șterge o pereche de pe o poziție pt că nu există poziția într-un dicționar

4.3 Folosim o stivă: $\begin{array}{ccccccc} 4 & 5 & 1 & * & - & 6 & 3 & + & 2 & * & + \\ \hline & & & & & & & & & & \end{array}$

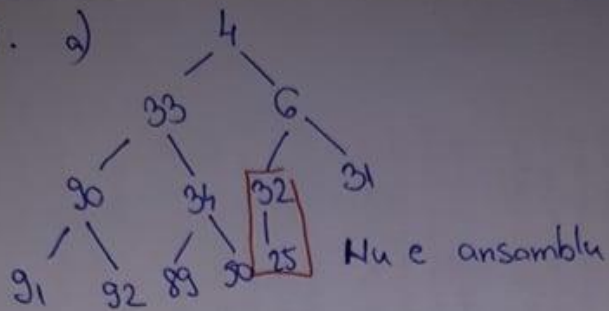
5: ~~4~~ ~~5~~ ~~1~~ ~~5~~ ~~1~~ ~~8~~ ~~8~~ ~~2~~ ~~18~~ 17

e \Rightarrow 15 și 100

4.4. c \Rightarrow la stivă avem vârf - acolo adăugăm, de acolo ștergem

\Rightarrow la coadă avem front - de acolo ștergem și end - acolo adăugăm

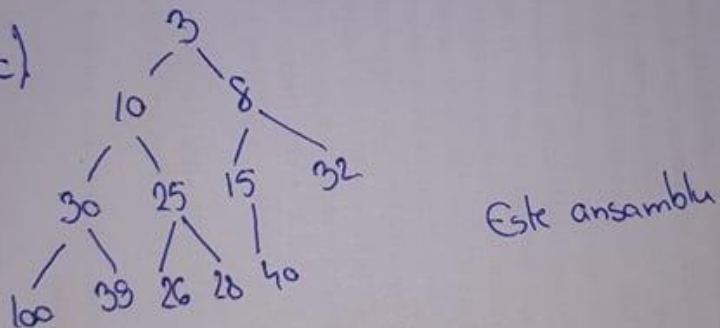
4.5. a)



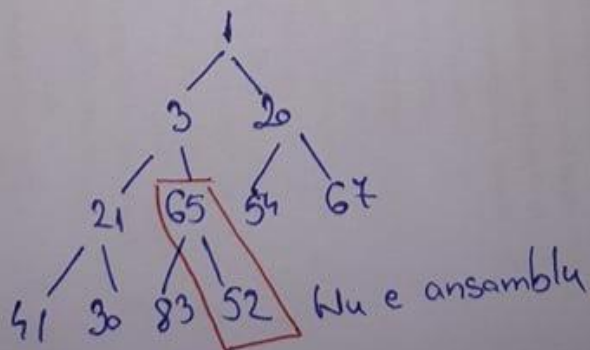
b)



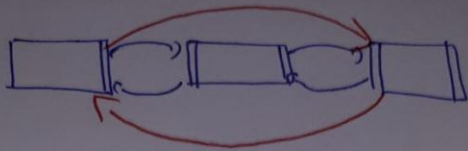
c)



d)



4.6.



b. Fals. Modificăm doar 2 legături

5. Multime pe VD.

Multime:


cap: întreg

len: întreg

elemente: TElem[]


Specificată: adauga(e)

- e este TElem

- e se adaugă în Mult  numai dacă nu există
dijă. Dacă se adaugă returnează true, altfel returnează false

funcție adauga(e: TElem) este:

// verificăm dacă există deja e

găsit: boolean 

index: întreg

găsit = false

index = 0

0

ca timp $g\ddot{a}sit == false$ si $index < this.len$ execută

dacă $this.elemente[index] == e$ atunci

$g\ddot{a}sit = true$

$index = index + 1$

dacă $g\ddot{a}sit == true$ atunci

returnează $false$

altfel // nu există în mulțime. Adăugăm

dacă $this.len == this.cap$ atunci

$elemente$: @ vector cu $this.cap * 2$ elemente

i : întreg

pentru $i = 0, this.len, 1$ execută

$elemente[i] = this.elemente[i]$

$this.elemente = elemente$

$this.cap = this.cap * 2$

$this.elemente[this.len] = e$

$this.len = this.len + 1$

returnează $true$

Complexitate: $O(n)$

Caz favorabil: $O(1)$ - elementul se găsește pe prima poziție

Caz defavorabil $O(n)$ - elementul nu se găsește în mulțime

Caz mediu: $O(n)$

Reprezentare iterator:

IteratorMulTime

mul: MulTime

current: Intreg

Subalgoritm crearea (~~m~~ m: MulTime) este:

this.mul = m

this.current = 0

subalg. următor() este

this.current = this.current + 1

subalg. element() este

returnează this.mul.elemente[this.current]

~~functie~~ valid() este

dacă this.current < this.mul.len atunci

returnează true

altfel returnează false

6. Do-ABC

Reprezentare:

Nod:

c: TChie

v: TValoare

st: \uparrow Nod

dr: \uparrow Nod

DictionarOrdonat

rădăcina: \uparrow Nod

rel: Relatie

cauta(c)

- caută o cheie în DictionarOrdonat, c este TChie
- dacă c este în Dictionar, returnează valoarea asociată, altfel

Null.

funcția cauta(c) este:

nodCurrent: \uparrow Nod

nodCurrent = this.rădăcina

găsit: boolean

găsit = false

cât timp găsit == false și nodCurrent \neq Null se execută

dacă [nodCurrent].c == c atunci

găsit = true

altfel dacă this.rel(c, [nodCurrent].c) atunci
nodCurrent \leftarrow [nodCurrent].st

altfel
nodCurrent ← [nodCurrent].dz
b
dacă gaură == true atunci
returnează [nodCurrent].v
altfel
returnează null
b

Complexitate: $O(n)$ 