

Materie-SDA

Recursivitate

-**functie direct recursive**=contine in interiorul sau un apel catre ea insasi

-**functie indirect recursive**= F(f de care forbim) ---contine---apel catre G---care contine---apel catre F

-**Reguli:** -trebuie sa existe cazuri elementare(sau sa se ajunga la unul)=**cazuri de baza\punkte fixe**

-**se aloca memorie pe stiva** pentru: val parametrilor si variabilelor locale (!memoria ramane alocata pe tot parcursul executiei apelului)

```
int fibo(int n)
{
    if (n==0)
        return 1;
    else if (n==1)
        return 1;
    else
        return fibo(n-1)+fibo(n-2);
}

int cate(int c, int nr)
{
    if(nr==0)
        return 0;
    else if(nr%10<c)
        return 1+cate(c,nr/10);
    else
        return cate(c,nr/10);
}
```

*Algoritm recursive de cautare binara

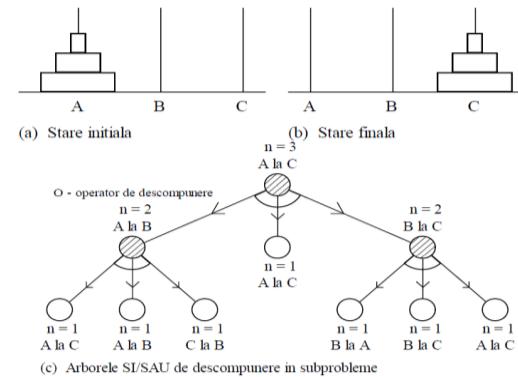
-gasirea unui nr dintr-un sir (crescator sau descrescator) printr-un nr minim de pasi

-se imparte sirul in 2 jumatati, jumatarea buna, in alte 2 jumatati si tot asa pana la gasirea nr cautat

```
1 #include <stdio.h>
2 int binarySearch(int *v, int x, int left, int right)
3 {
4     int mij=(left+right)/2;
5     if(v[mij]==x)
6         return 1;
7     else if(left>right)
8         return 0;
9     else if(x<v[mij])
10        return binarySearch(v,x,left,mij-1);
11     else
12        return binarySearch(v,x,mij+1,right);
13 }
14
15 int main()
16 {
17     int v[100],n,cf,i;
18     scanf("%d", &n);
19     for(i=0; i<n; i++)
20         scanf("%d", &v[i]);
21     scanf("%d", &cf);
22     printf("%d\n", binarySearch(v,cf,0,n-1));
23 }
```

->cautare binara pe un sir ordonat

->cautarea binara recursiva este un exemplu de **“Divide et Impera”**(metoda care presupune impartirea problemei in probleme mai mici si rezolvarea lor pe rand)



->exemplu: **Problema turnurilor din Hanoi**

```

#include <stdio.h>
void swap(int *x, int *y)
{
    int t=*x;
    *x=*y;
    *y=t;
}
int partition(int *v, int st, int dr)
{
    int pivot=v[dr];
    int i=st-1;
    for (int j=st; j<dr; j++){
        if(v[j]<=pivot){
            i++;
            swap(&v[i], &v[j]);
        }
    }
    swap(&v[i+1], &v[dr]);
    return i+1;
}
void quickSort(int *v, int st, int dr)
{
    if(st<dr){
        int mid=partition(v,st,dr);
        quickSort(v,st,mid-1);
        quickSort(v,mid+1,dr);
    }
}
int main()
{
    int n;
    int v[100];
    scanf("%d", &n);
    for(int i=0; i<n; i++)
        scanf("%d", &v[i]);
    quickSort(v,0,n-1);
    for(int i=0; i<n; i++)
        printf("%d ", v[i]);
}

```

QuickSort

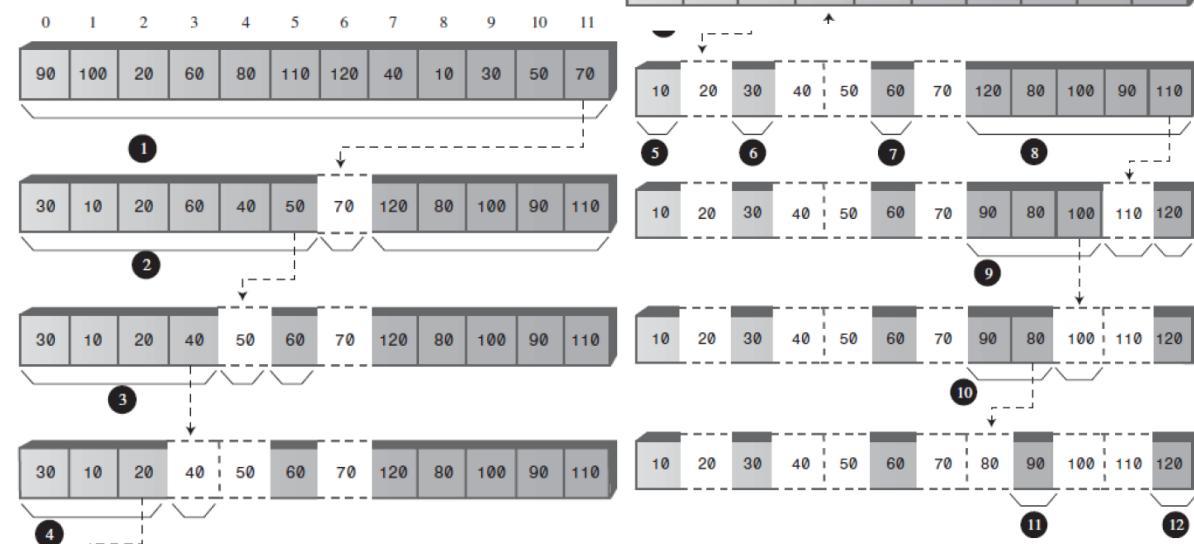
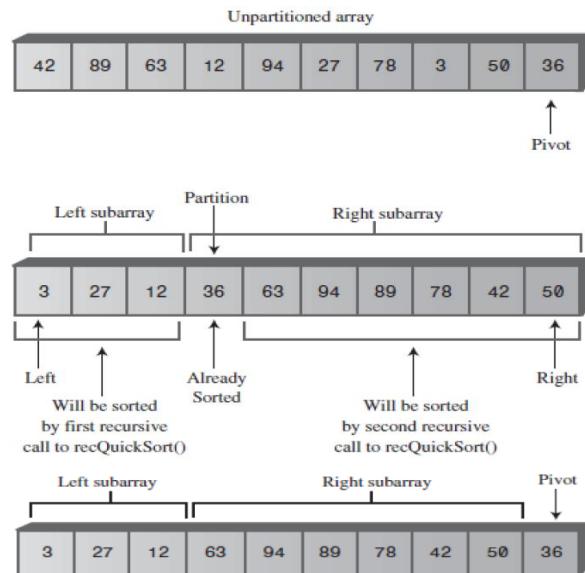
-mecanism de baza pentru sortarea rapida

-Partitionarea datelor=impartirea acestora in 2 multimi, cu valori mai mici (intr-o parte) si mai mari (in cealalta) fata de o valoare precizata. (!dupa partitonare nu se sorteaza, datele doar sunt impartite in 2 clase)

-QuickSort=cel mai rapid algoritm de sortare, in majoritatea cazurilor

-temp de executie O(N²)

-funtionare Quicksort recursive (gandire)



MergeSort=interclasarea 2 vectorilor deja sortati

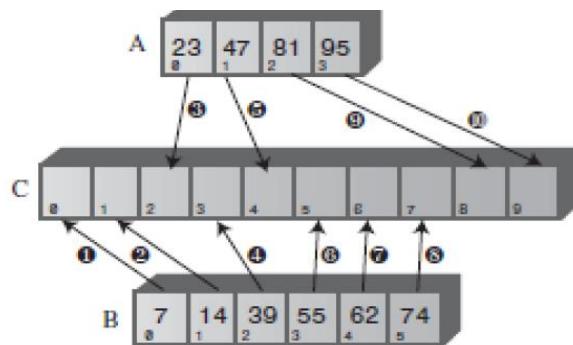
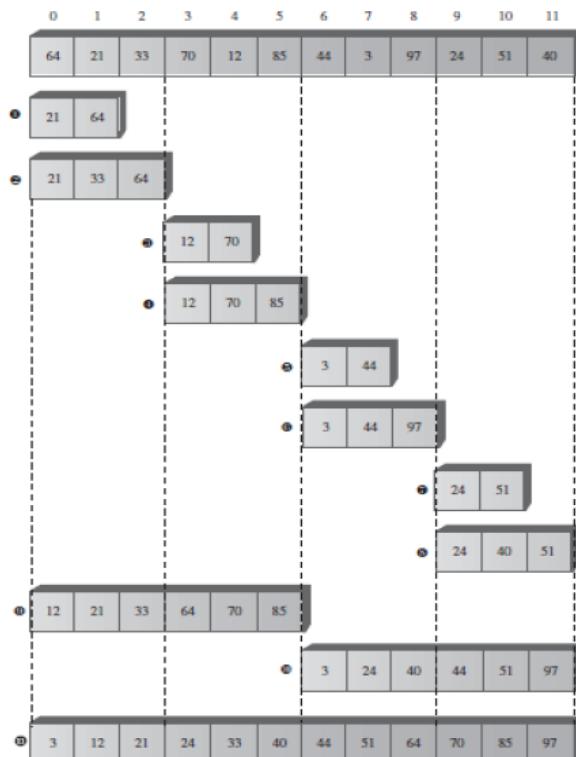
-algoritm recursiv de sortare

-dovadă conceptual mai simplu ca QuickSort

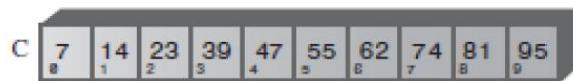
-dezavantaj = are nevoie de un tablou suplimentar

Sortare prin interclasare

-imparte vectorul in 2 parti => le sortezi => le interclasazezi (se face acelasi lucru si cu



a) Before Merge



b) After Merge

jumataatile)

-cel mai simplu caz este cel cand vectorul initial este o putere a lui 2

-in caz contrar se por la interclasarea vectorilor de marimi diferite

-toti sub-vectorii intermediari se memoreaza intr-un vector de lucru care este de marimea vectorului initial (la pozitiile lor corespunzatoare)

-dupa fiecare operatie de interclasare se copiaza vectorul de lucru in vectorul original

Introducere in Complexitatea algoritmilor

-eficiența=resursele consumate de algoritmul in executie:

->temp de executie=eficienta timpului

->memorie consumata=eficienta spatiului

*Dimensiunea datelor de prelucrat

-probleme de sortare -> numarul de elemente de sortat

-inmultirea a 2 numere -> numarul de biti din reprezentarea numerelor

*Temp de executie = numarul de operatii primitive ("pasi" executati)

-analizam mereu cazul cel mai nefavorabil

-in cazul timpului aflam limita superioara a timpului de executie

-timpul de executie mediu este adesea apropiat de ce mai devaforabil

-**Analiza algoritmica** = comportamentul algoritmului la executie

-**Complexitatea algoritmului** = numarul de operatii elementare in functie de dimensiunea problemei(volumul datelor de intrare)

- $n_{op} = T(N)$

- Functia T poate fi:

 - Constantă $T(N) = c_0$

 - Liniară $T(N) = c_0N + c_1$

 - Pătratică $T(N) = c_0N^2 + c_1N + c_2$

 - Polinomială $T(N) = c_0N^p + \dots + c_p$ unde $p > 2$

 - Exponențială $T(N) = c_0a^N$ unde $a > 1$

-exemple pentru cazul $N=100$

-**performanata calculatorului** este dependenta de complexitatea alg si independent de alg in sine

-> se considera ca **o operatie elementara dureaza 10^{-6} sec**

Complexitate	Valoare
$\log_2 N$	$6,65 * 10^{-6}$ sec
N	10^{-4} sec
$N * \log_2 N$	$6,65 * 10^{-4}$ sec
N^2	10^{-2} sec
N^3	1 sec
2^N	$2^{100} * 10^{-6}$ sec $\cong 10^{24}$ sec

-**complexitatea alg** depinde de configuratia datelor de intrare

*comportamente ale Alg

-**cea mai buna** = gasirea valorii dupa o singura comparatie

-**medie** = mai greu de aflat (se folosesc distributii statistice ale datelor)

În cazul căutării secvențiale, aceasta este:

$$(1 + 2 + \dots + N) / N = (N + 1) / 2 = O(N)$$

-in **situatia cea mai nefavorabila** = gasirea valorii pe ultima pozitie (complexitate $O(N)$)

*Limitari in analiza algoritmica

-imposibil de aflat pt algoritmi complicati

-greu de aflat cazul tipic

- **O** nu poate surprinde micile diferente dintre algoritmi si nu e concludenta pentru volume mici de date

Structuri de date si tipuri de date abstracte

-**structura de data**=mod de organizare a datelor in memorie (pt accesare eficienta)

-**ex**: vectori, matrici, structure, liste

-**structura de date abstracta**=descrierea unui mod de organizare a datelor + operatiile specific

-> permite abstractizarea datelor fara a tine cont de implementare

->ex: multime, stiva, coada

-**colectie**=grup de elem de ac tip (pot fi duplicate)

-**multime**=colectie fara duplicate

-**colectie/multime**=structura de date abstracta, descrisa prin operatii de baza care pot fi executata asupra ei

->operatiile de baza sunt independente de tipul elem si de reprezentarea in memorie

->pt implementare e necesara reprezentarea datelor (o structura de date)

*Structuri de date fundamentale

-**vector**=colectie de date de acelasi tip

-structura=grupare cu mai multe date de tipuri diferite

-structura de date -> caracterizata prin relatiile dintre elementele colectiei si operatiile posibile dintre acestea

-colectii liniare (secvente, liste)= elem cu un singur successor

-colectii arborescente (ierarhice)=elem pot avea mai multi succesi (mai multi-fii, un parinte)

-tip abstract de date=definit prin operatiile asociate

Liste inlantuite

-colectie de alemente, alocate dinamic, dispersate in memorie si legate intre ele prin pointeri

-este o structura dinamica flexibila (se poate extinde continuu)

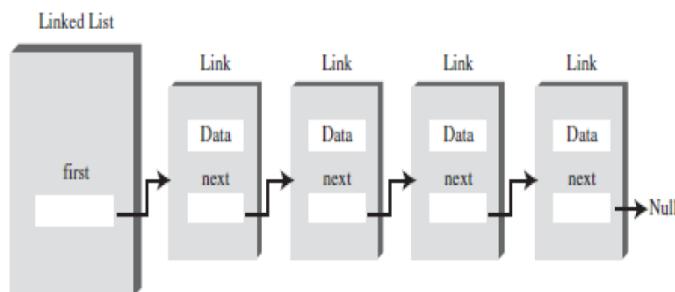
-limitare = marimea zonei "heap" din care se aloca memorie

*Liste simplu inlantuite

-fiecare element din lista contine adresa elementului urmator

-primul elem=head

-ultimul element contine NULL sau adresa primului element (pt o lista circulara) sau adresa unui elem terminator cu valoare speciala



```
typedef struct node {
    int *data;
    struct node *next;
} node_t;

typedef struct sl_list {
    node_t *head;
    node_t *tail;
    size_t len;
} sl_list_t;
```

exemple pe liste fara santinela

```
sl_list_t *init_list()
{
    sl_list_t *lista = NULL;
    lista = malloc(sizeof(*lista));
    lista->head = NULL;
    lista->tail = NULL;
    lista->len = 0;
    return lista;
}
```

```
node_t *init_node(int value)
{
    node_t *nod = NULL;
    nod = malloc(sizeof(*nod));
    nod->data = malloc(sizeof(*nod->data));
    *nod->data = value;
    nod->next = NULL;
    return nod;
}
```

```
void destroy_list(sl_list_t *list)
{
    if(list->head == NULL){
        free(list);
        return;
    }
    node_t *aux;
    for(aux = list->head; list->head->next != NULL; list->head = list->head->next)
        free_node(aux);
    free(list);
}
```

```
void free_node(node_t *node)
{
    free(node->data);
    free(node);
}
```

```

int insert_node(sl_list_t *list, int value, uint position)
{
    if(position < 0 || position > list->len - 1)
        return -1;
    node_t *curr = init_node(value);
    if(position == 0 || position == list->len - 1){
        int ok = 0;
        if(position == 0 && list->len == 0){
            list->head = curr;
            list->tail = curr;
            list->len++;
        }
        else if(position == 0){
            curr->next = list->head;
            list->head = curr;
            list->len++;
        }
        else if(position == list->len - 1){
            list->tail->next = curr;
            list->tail = curr;
            list->len++;
        }
    }
    else{
        node_t *aux = list->head;
        for(int i = 0; i < position-1; i++)
            aux = aux->next;
        curr->next = aux->next;
        aux->next = curr;
        list->len++;
    }
    return 0;
}

```

```

void print_list_int(sl_list_t *list)
{
    node_t *aux = list->head;
    printf("%-20s", "Lista construita:");
    while (aux->next != NULL) {
        printf("%d->", *aux->data);
        aux = aux->next;
    }
    printf("%d\n", *aux->data);
}

```

```

int remove_node_by_key(sl_list_t *list, int value)
{
    node_t *curr = list->head;
    node_t *ante = NULL;
    while(curr->next != NULL){
        if(*curr->data == value)
            break;
        ante = curr;
        curr = curr->next;
    }
    if(curr == list->tail && *list->tail->data != value)
        return -1;
    if(curr == list->tail && *list->tail->data == value){
        free_node(curr);
        list->tail = ante;
        list->tail->next = NULL;
        list->len--;
        return 0;
    }
    if(ante == NULL){
        node_t *aux = list->head;
        list->head = list->head->next;
        free_node(aux);
        list->len--;
        return 0;
    }
    ante->next = curr->next;
    free_node(curr);
    list->len--;
    return 0;
}

```

```

int remove_node_at_position(sl_list_t *list, uint position)
{
    if(position < 0 || position > list->len - 1)
        return -1;
    if(position == 0){
        node_t *aux = list->head;
        list->head = list->head->next;
        free_node(aux);
        list->len--;
        return 0;
    }
    node_t *curr = list->head;
    node_t *ante = NULL;
    for(int i = 0; i < position; i++){
        ante = curr;
        curr = curr->next;
    }
    if(curr == list->tail){
        list->tail = ante;
        list->tail->next = NULL;
        free_node(curr);
        list->len--;
        return 0;
    }
    list->len--;
    ante->next = curr->next;
    free_node(curr);
    return 0;
}

```

```

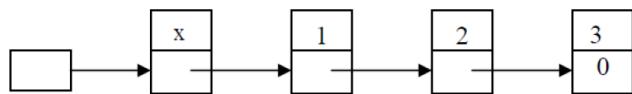
int insert_sorted_desc(list_t *list, int value) {
    if(list->head==NULL){
        node_t *nod=init_node(&value);
        list->head=nod;
        list->tail=nod;
        list->len++;
    }
    else if(*list->head->data < value){
        insert_node(list,value,0);
    }
    else if(*list->tail->data > value){
        insert_node(list,value,list->len);
    }
    else{
        node_t *nod=list->tail;
        while(*nod->data < value)
            nod=nod->prev;
        node_t *aux=init_node(&value);
        aux->next=nod->next;
        aux->prev=nod;
        nod->next->prev=aux;
        nod->next=aux;
        list->len++;
    }
    /* TODO 2
     * Se vor trata urmatoarele cazuri:
     * - Lista este vida
     * - Inserarea se face la inceputul listei
     * - Inserarea se face la sfarsitul listei
     */
    return 0;
}

```

->exemplu de lista auto-organizata

*Lista cu santinela

-contine cel putin un element (santinela), e creat la initializarea listei si ramane mereu la inceputul ei



-operatiile sunt mai simple (nu trebuie tratat cazul modificarii primului elem din lista, pentru ca primul e santinela si nu se schimba)

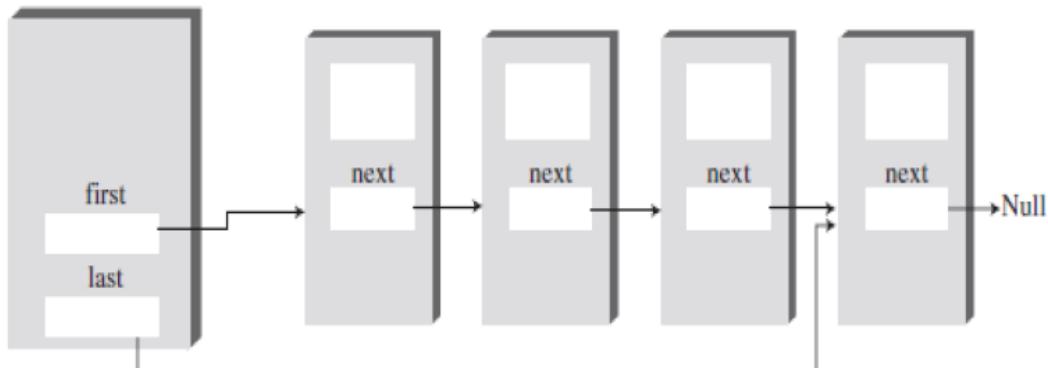
*Lista cu 2 capete

-are in plus pointer la ultimul elem din lista

-permite inserarea mai usoara a unui elem nou in caputul listei (la liste obisnuite trebuie parcursa

toata lista inaintea adaugarii elem la final)

-foloseste la implementarea unei cozi



*Eficienta listelor inlantuite

-inserare si stergere rapide, la inceput si final => O(1)

- cautarea, stergerea si inserarea unui elem in medie, presupune jumataate din parcurgerea totala a listei => O(N)

-eficienta crescuta fata de tablouri, deoarece nu mai este necesara mutarea elem dupa stergere sau inserare

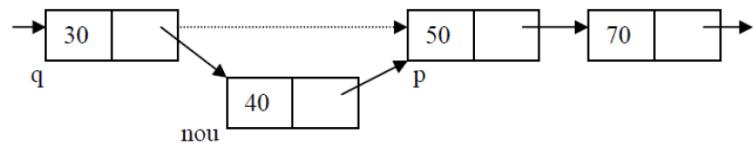
-listele nu folosesc mai multa memorie decat au nevoie

Liste inlantuite ordonate

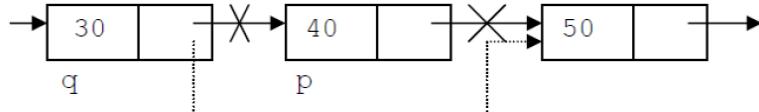
-aici nu se poate aplica cautarea binara (e necesara trecerea prin fiecare elem pentru a-gasi pe cel cautat)

-mai intai se cauta locul unde trebuie pus elem => apoi este inserat

-elem poate fi **inserat la inceput, mijloc sau final**



stergerea dintr-o lista ordonata



-pentru modificarea primului element **se va modifica si head-ul**

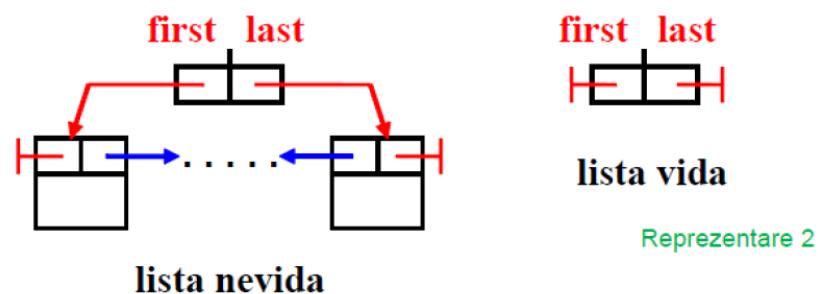
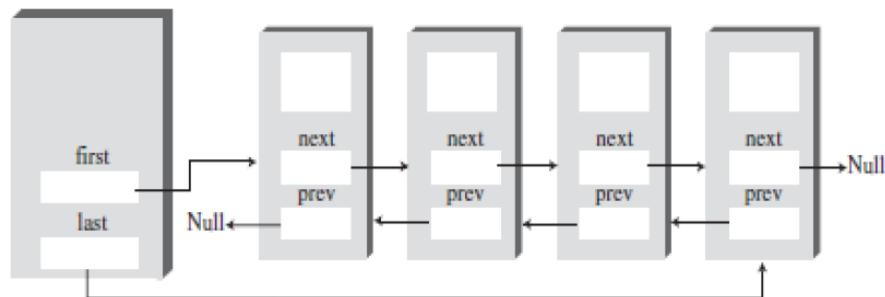
-la fiecare modificare se **va schimba len-ul**

Liste dublu inlantuite

-fiecare elem contine **2 adrese de legatura** (catre elem precedent si catre elem urm)

-acces mai rapid la elem precedent

-lista se poate parcurge in ambele sensuri



Reprezentare 2

```

typedef struct node {
    int *data;
    struct node *next;
    struct node *prev;
} node_t;

typedef struct dl_list {
    node_t *head;
    node_t *tail;
    int len;
} list_t;

node_t *init_node(int *value) {
    node_t *new_node = malloc(sizeof(node_t));
    new_node->data = malloc(sizeof(int));
    *new_node->data = *value;
    new_node->next = NULL;
    new_node->prev = NULL;
    return new_node;
}

list_t *init_list() {
    list_t *new_list = malloc(sizeof(list_t));
    new_list->head = NULL;
    new_list->tail = NULL;
    new_list->len = 0;
    return new_list;
}

void free_node(node_t *nod)
{
    free(nod->data);
    free(nod);
}

void destroy_list(list_t *list) {
    node_t *aux;
    /* Se sterg toate nodurile din lista, incepand de la coada */
    while (list->len > 0) {
        aux = list->tail;           // Se retine un pointer catre ultimul nod din lista
        list->tail = list->tail->prev; // Se muta pointerul cozii cu un nod mai in spate
        free_node(aux);            // Se elibereaza memoria ocupata de nodul eliminat
        --list->len;               // Se decrementeaza lungimea listei
    }
    /* Se elibereaza memoria ocupata de structura listei */
    free(list);
}
}

void print_list(list_t *list) {
    node_t *aux = list->head;

    printf("%-25s", "Lista construita (>):");

    if (aux == NULL) {
        // Listavida
        printf("NULL\n");
        return;
    }

    while (aux != list->tail) {
        printf("%d -> ", *aux->data);
        aux = aux->next;
    }
    printf("%d\n", *aux->data);
}

void print_list_reversed(list_t *list) {
    node_t *aux = list->tail;
    printf("%-25s", "Lista construita (>):");
    if (aux == NULL) {
        printf("NULL\n");
        return;
    }
    while (aux != list->head) {
        printf("%d -> ", *aux->data);
        aux = aux->prev;
    }
    printf("%d\n", *aux->data);
    /* TODO l.a.
     * Se va porni de la ultimul nod din lista si se va parcurge lista
     * pana cand se va ajunge la primul nod
     */
}

```

```

int insert_node(list_t *list, int value, int position) {
    if(position<0 || position>list->len)
        return -1;
    if(list->head==NULL){
        node_t *node=init_node(&value);
        list->head=node;
        list->tail=node;
        list->len=1;
    }
    else if(position==0){
        node_t *node=init_node(&value);
        node->next=list->head;
        list->head->prev=node;
        list->head=node;
        list->len++;
    }
    else if(position==list->len){
        node_t *node=init_node(&value);
        node->prev=list->tail;
        list->tail->next=node;
        list->tail=node;
        list->len++;
    }
    else{
        if(position<=list->len/2){
            node_t *aux=list->head;
            for(int i=0; i<position-1; i++)
                aux=aux->next;
            node_t *nod=init_node(&value);
            nod->next=aux->next;
            aux->next->prev=nod;
            nod->prev=aux;
            aux->next=nod;
            list->len++;
        }
        else{
            node_t *aux=list->tail;
            for(int i=list->len-1; i>position; i--)
                aux=aux->prev;
            node_t *nod=init_node(&value);
            nod->prev=aux->prev;
            aux->prev->next=nod;
            nod->next=aux;
            aux->prev=nod;
            list->len++;
        }
    }
    /* TODO 1.b.
     * Se vor trata urmatoarele cazuri:
     * - Pozitia nu este in intervalul [0, len] => return -1
     * - Inserarea se face intr-o lista vida
     * - Inserarea se face la inceputul listei
     * - Inserarea se face la sfarsitul listei
     * - Inserarea se face intr-o pozitie intermediara:
     *     - Pozitia este mai aproape de inceputul listei => parcurgerea se va face de la inceputul listei
     *     - Pozitia este mai aproape de sfarsitul listei => parcurgerea se va face de la sfarsitul listei
     */
    return 0;
}

int insert_sorted_desc(list_t *list, int value) {
    if(list->head==NULL){
        node_t *nod=init_node(&value);
        list->head=nod;
        list->tail=nod;
        list->len++;
    }
    else if(*list->head->data < value){
        insert_node(list,value,0);
    }
    else if(*list->tail->data > value){
        insert_node(list,value,list->len);
    }
    else{
        node_t *nod=list->tail;
        while(*nod->data < value)
            nod=nod->prev;
        node_t *aux=init_node(&value);
        aux->next=nod->next;
        aux->prev=nod;
        nod->next->prev=aux;
        nod->next=aux;
        list->len++;
    }
    /* TODO 2
     * Se vor trata urmatoarele cazuri:
     * - Lista este vida
     * - Inserarea se face la inceputul listei
     * - Inserarea se face la sfarsitul listei
     */
    return 0;
}

```

```

int remove_node(list_t *list, int position) {
    if(position<0 || position>=list->len)
        return -1;
    if(position==0){
        node_t *aux;
        aux=list->head;
        list->head=aux->next;
        list->head->prev=NULL;
        list->len--;
        free(aux);
    }
    else if(position==list->len-1){
        node_t *aux;
        aux=list->tail;
        list->tail=aux->prev;
        list->tail->next=NULL;
        list->len--;
        free(aux);
    }
    else{
        if(position<=list->len/2){
            node_t *aux=list->head;
            for(int i=0; i<position-1; i++)
                aux=aux->next;
            node_t *nod=aux->next;
            nod->next->prev=aux;
            aux->next=nod->next;
            free(nod);
            list->len--;
        }
        else{
            node_t *aux=list->tail;
            for(int i=list->len-1; i>position; i--)
                aux=aux->prev;
            node_t *nod=aux->prev;
            nod->prev->next=aux;
            aux->prev=nod->prev;
            free(nod);
            list->len--;
        }
    }
    /* TODO 1.c.
     * Se vor trata urmatoarele cazuri:
     * - Pozitia nu este in intervalul [0, len - 1] sau lista este vida (len == 0) => return -1
     * - Stergerea se face de la inceputul listei
     * - Stergerea se face de la sfarsitul listei
     * - Stergerea se face dintr-o pozitie intermediara:
     *     - Pozitia este mai aproape de inceputul listei => parcurgerea se va face de la inceputul listei
     *     - Pozitia este mai aproape de sfarsitul listei => parcurgerea se va face de la sfarsitul listei
     */
    return 0;
}

int insert_sorted_desc(list_t *list, int value) {
    if(list->head==NULL){
        node_t *nod	init_node(&value);
        list->head=nod;
        list->tail=nod;
        list->len++;
    }
    else if(*list->head->data < value){
        insert_node(list,value,0);
    }
    else if(*list->tail->data > value){
        insert_node(list,value,list->len);
    }
    else{
        node_t *nod=list->tail;
        while(*nod->data < value)
            nod=nod->prev;
        node_t *aux	init_node(&value);
        aux->next=nod->next;
        aux->prev=nod;
        nod->next->prev=aux;
        nod->next=aux;
        list->len++;
    }
    /* TODO 2
     * Se vor trata urmatoarele cazuri:
     * - Lista este vida
     * - Inserarea se face la inceputul listei
     * - Inserarea se face la sfarsitul listei
     */
    return 0;
}

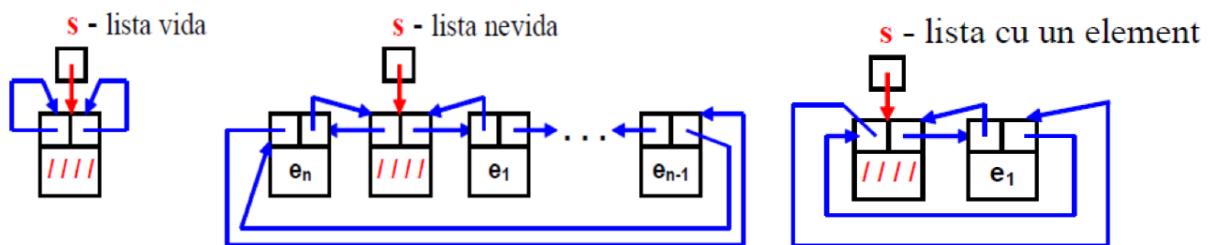
```

```

list_t *remove_duplicates(list_t *list) {
    list_t *new_list = init_list();
    for(node_t *aux=list->head; aux !=NULL; aux=aux->next){
        int val=*aux->data;
        for(node_t *aux2=aux->next; aux2 != NULL; aux2=aux2->next){
            int val2=*aux2->data;
            if(val<val2){
                *aux->data=val2;
                *aux2->data=val;
                int ct;
                ct=val2;
                val2=val;
                val=ct;
            }
        }
        node_t *aux=list->head;
        insert_node(new_list,*aux->data,0);
        for(aux=list->head->next; aux != NULL; aux=aux->next){
            if(*aux->data != *aux->prev->data)
                insert_node(new_list,*aux->data,new_list->len);
        }
    /*
     * TODO 3
     * Pornind de la lista primita ca parametru, se construieste
     * o noua lista care va contine doar valori unice.
     * Lista finala va fi sortata descrescator
     */
    return new_list;
}

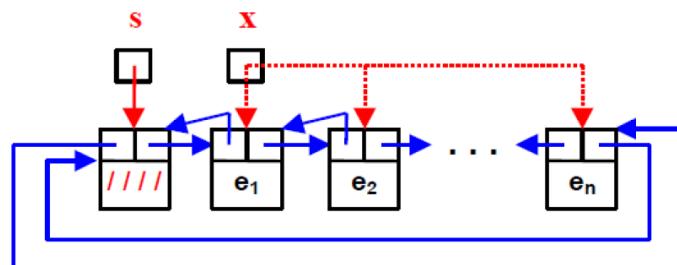
```

Liste dublu inlantuite circulare cu santinela



-parcurgerea de la inceputul listei

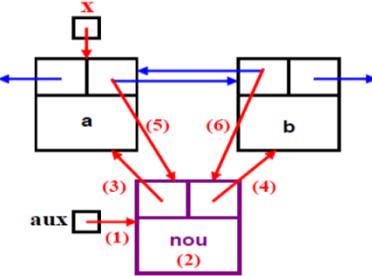
- `for (x = s->next; x != s; x = x->next)`



-parcurgerea de la sfarsitul listei `for (x = s->prev; x != s; x = x->prev)`

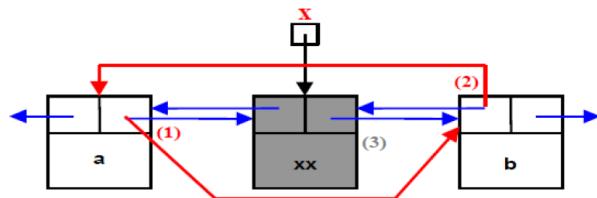
-inserare dupa celula cu adresa x

```
(1) aux = (TListNode) malloc(sizeof(ListNode));
(2) aux->elem = value;
(3) aux->prev = x;
(4) aux->next = x->next;
(5) x->next = aux;
(6) aux->next->prev = aux;
```



-eliminarea celulei de la adresa x ($x \neq s$)

```
(1) x->prev->next = x->next;
(2) x->next->prev = x->prev;
(3) free(x);
```



*Liste Skip

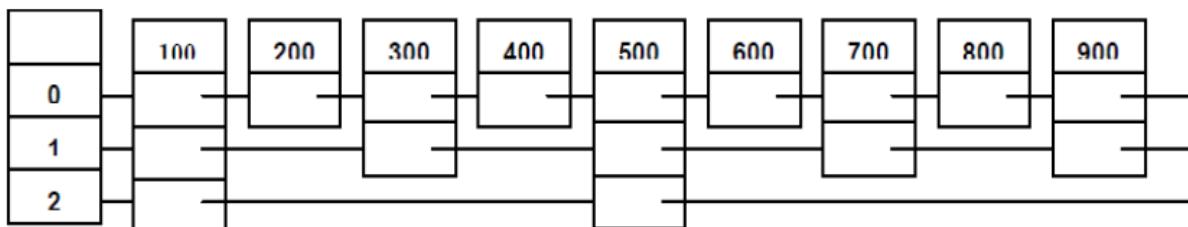
-dezavantaj la liste=timpul de cautare a unei valori (timp proportional cu lungimea)

-**Lista Skip** (lista cu salturi) = liste ordonate cu timp de cautare mic

-adresele de legatura intre elemente sunt situate pe niveluri

-o lista cu salturi poate fi privita ca fiind formata din mai multe liste paralele cu anumite elem comune

Adresele de legatură între elemente sunt situate pe câteva niveluri: pe nivelul 0 este legătura la elementul imediat următor din listă, pe nivelul 1 este o legătură la un element aflat la o distanță d_1 , pe nivelul 2 este o legătură la un element aflat la o distanță $d_2 > d_1$ și.a.m.d.



-cautarea incepe la nivelul maxim si se opreste la elem cu valoare mai mica decat elem cautat

*Liste neliniare

-o lista care poate contine subliste, pe oricate niveluri de adancime

-**limbajul Lisp** ("List Processing")=listele se reprezinta prin expresii (in Lisp pot fi memorate expresii aritmetice, propozitii, fraze)

(- 5 3) 5 - 3
 o expresie cu 3 atomi

(+ 1 2 3 4) 1 + 2 + 3 + 4

-**CAR**=primul elem din lista (de obicei o functie sau un operand)

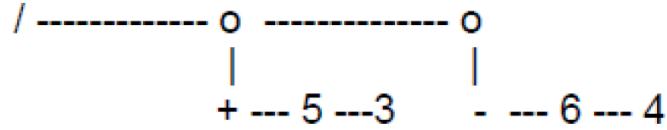
-CDR=celelalte elemente sau restul listei (operanzi)

-**expresie LISP** ca lista neliniara cu 2 subliste

-pt o implementare eficienta->se folosesc 2 tipuri de noduri

->noduri cu date = pointeri

->noduri cu adresa unei subliste

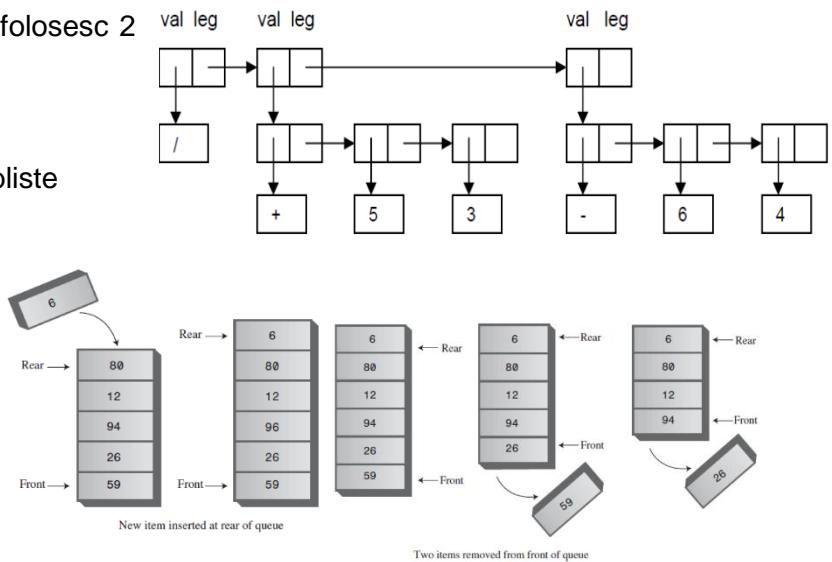


Cozi (FIFO)

-structura de date abstracta

-adaugarea se face la final

-extragerea la inceput



*Cozi cu prioritati

-elem sunt ordonate dupa valoarea unei chei

-elem cu cheia minima/maxima se afla in fata

-elementele sunt inserate in ordinea corespunzatoare

-coada de prioritati ordonata descrescator => elemental cu cheie maxima are cea mai mare prioritate

-coada de prioritati ordonata crescator => elemental cu cheie minima are cea mai mare prioritate

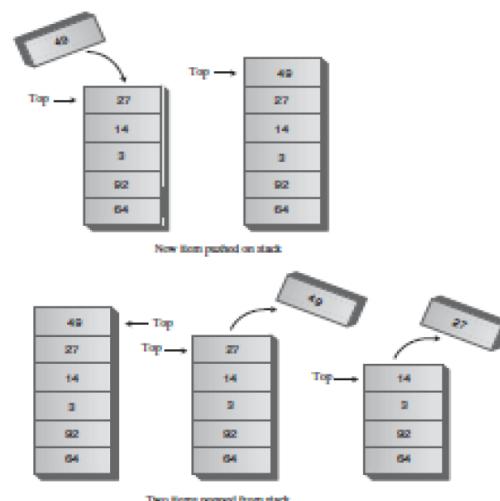
Stive(LIFO)

-structura de date abstracta care permite accesul la un singur elem (ultimul)

-permite accesul la elementul inserat

-**operatii principale:** introducerea unui element in varful stivei si extragerea lui

-poate fi implementata prin liste sau array-uri



Arbore

-**avantaje:** tablouri+liste

-**structuri liniare:** liste, stive, cozi

-**structuri ierarhice:** arbori

-Arborii ofera : ->inserare si stergere rapida

->cautare rapida

-Arbore=colectie de noduri si arce(legaturi dintre noduri) in care exista o singura cale intre oricare doua noduri

-Calea=secventa de noduri si arce care conecteaza aceste noduri

-Radacina=primul nod din arbore

-Arbore= radacina + nr finit de arbori(subarbori)

-Arce=modul in care nodurile sunt conectate

-Tipuri de arbori: ->**Binari** (arbore cu cel mult 2 fii)

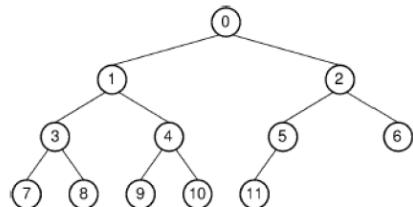
->**Multicai/n-ari** (nodurile pot avea mai multi fii)

*Arbore binar

-orice nod poate avea cel mult 2 fii (fiu stang si fiu drept)

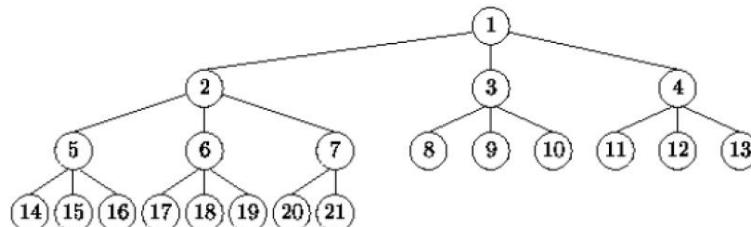
-Arbore vid=nu contine niciun nod

-Arbore binar-noduri cu cel mult 2 fii (subarboare stang, subarboare drept)



*Arbore multicai

-mai mult de 2 subarbori



*Terminologie

-Stramos=orice nod aflat pe calea de la radacina la nodul x

-Descendent=orice nod aflat in arborele cu radacina x

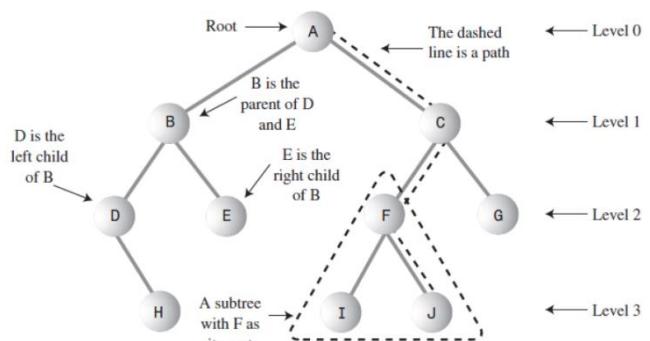
-Tata/fie=noduri aflate la distanta 1 (stramos/descendent direct)

-Frati=fii aceluiasi nod

-Nivel/adancime=numarul de arce de la radacina la nod (Radacina e pe nivelul 0)

-Inaltime: ->val maxima de pe nivelurile nodurilor terminale

->nivelul maxim din arbore
(adancimea frunzei de nivel maxim)



H, E, I, J, and G are leaf nodes

-Ordin(grad)=numarul de descendenți

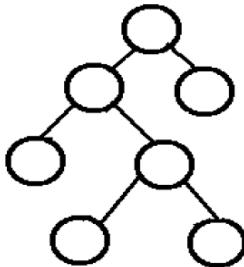
directi ai unui nod (valoarea maxima luata de gradul unui nod component al arborelui)

-**noduri interne**= noduri care au subarbore

-**noduri externe**=Frunze

*Arbore binar plin

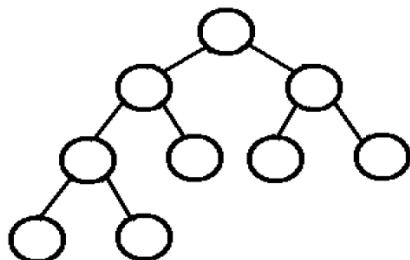
-fiecare nod are exact 0 sau 2 fii



Arbore binar plin

*Arbore binar complet

-arbore binar complet umplut cu exceptia ultimului nivel care este umplut de la stanga la dreapta



Arbore binar complet

*Arbore binari plini

-N noduri interne => N+1 noduri externe

-N noduri interne => 2N arce/legaturi ->N-1 legaturi intre noduri interne

->N+1 legaturi la nodurile externe

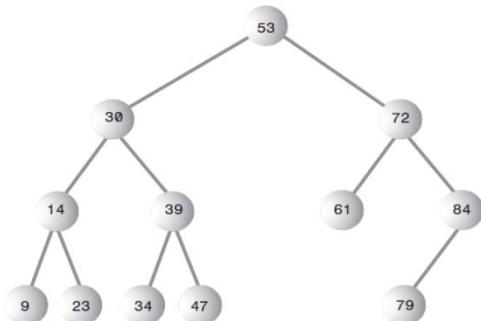
*Arbore binar de cautare

-arbore binar care are o cheie asociata fiecarui nod

-cheia din nod este mai mare ca oricare cheie din subarborele stang

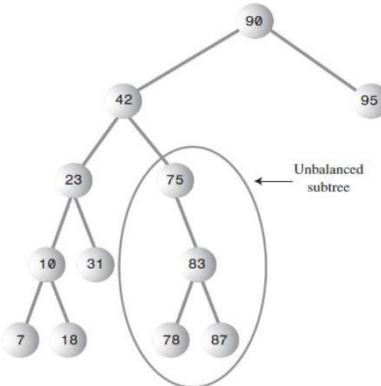
-cheia din nod este mai mica ca oricare cheie din subarborele drept

```
typedef struct BTNode {
    int data;
    struct BTNode *parent;
    struct BTNode *left;
    struct BTNode *right;
} BTNode;
```



*Arboree neechilibrați

-majoritatea nodurilor sunt situate pe o singura parte a radacini



Forma postfixata a unei expresii

-**notatie postfixata**(forma poloneza inversa)

-**forma infixata** -> forma postfixata => necesitatea folosirii unei stive

*Infixata->Postfixata

-**notatie postfixata** ex: $A+B \Rightarrow AB+$

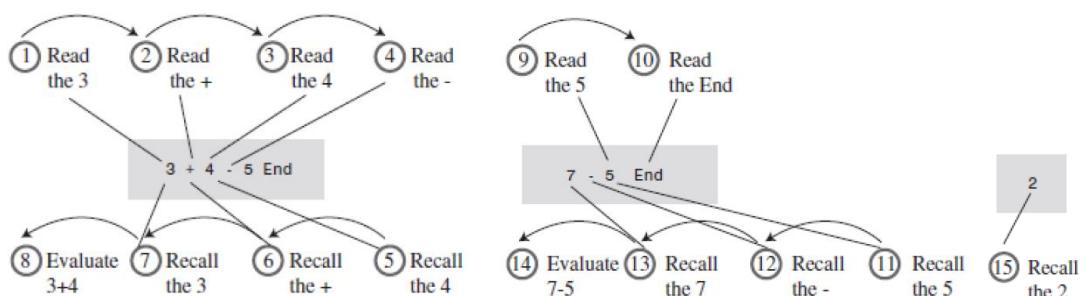
$A/B \Rightarrow AB/$

TABLE 4.2 Infix and Postfix Expressions

Infix	Postfix
$A+B-C$	$AB+C-$
A^B/C	$AB^C/$
$A+B*C$	$ABC*+$
$A*B+C$	$AB*C+$
$A*(B+C)$	$ABC+*$
$A*B+C*D$	$AB*CD*+$
$(A+B)^(C-D)$	$AB+CD-^*$
$((A+B)^C)-D$	$AB+C*D-$
$A+B*(C-D)/(E+F)$	$ABCDEF+/-+*$

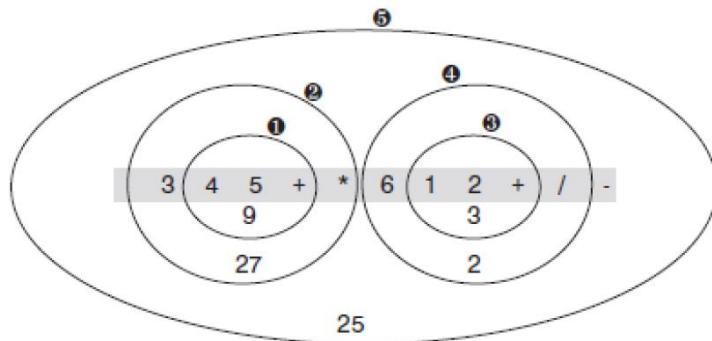
-**notatie prefixata** ex: $+AB$ in loc de $AB+$ (operatorii se scriu inaintea operanzilor)

Item Read	Expression Parsed So Far	Comments
3	3	
+	3+	
4	3+4	
-	7	When you see the -, you can evaluate 3+4.
5	7-5	
End	2	When you reach the end of the expression, you can evaluate 7-5.



Character Read from Infix Expression	Infix Expression Parsed so Far	Postfix Expression Written So Far	Character Read from Infix Expression	Infix Expression Parsed So Far	Postfix Expression Written So Far	Stack Contents
A	A	A	A	A	A	
*	A*	A	+	A+	A	+
(A*(A	B	A+B	AB	+
B	A*(B	AB	*	A+B*	AB	*+
+	A*(B+	AB	(A+B*(AB	+*(
C	A*(B+C	ABC	C	A+B*(C	ABC	+*(
)	A*(B+C)	ABC+	-	A+B*(C-	ABC	+*(-
	A*(B+C)	ABC+*	D	A+B*(C-D	ABCD	+*(-
End	A*(B+C)	ABC+*)	A+B*(C-D)	ABCD-	+*(
				A+B*(C-D)	ABCD-	+*
				A+B*(C-D)	ABCD-	+*
				A+B*(C-D)	ABCD-*	+
				A+B*(C-D)	ABCD-*+	

-Evaloarea expresiei postfixate



Radix Sort

-algoritm de sortare pentru numere naturale(dupa ranguri)

```
void radixSort(int *v, int n) {
    queue_t **vect=malloc(10*sizeof(queue_t *));
    for(int i=0; i<10; i++){
        vect[i]=initList();
    }
    int max=0;
    for(int i=0; i<n; i++){
        if(max<v[i])
            max=v[i];
    }
    int ct=0;
    while(max){
        ct++;
        max=max/10;
    }
    int impartit=1;
    for(int k=0; k<ct; k++){
        for(int i=0; i<n; i++){
            if(v[i]<impartit){
                enqueue(vect[0],v[i]);
            }else{
                int cifra=(v[i]/impartit)%10;
                enqueue(vect[cifra],v[i]);
            }
        }
        int status,j=0;
        for(int i=0; i<10; i++){
            status=0;
            while(status==0){
                int nr=dequeue(vect[i],&status);
                if(status==0){
                    v[j]=nr;
                    j++;
                }
            }
        }
        impartit*=10;
    }
}
```

- Tablou nesortat cu 7 elemente
- 421 240 035 532 305 430 124
- Sortare după cifra unităților
- (240 430) (421) (532) (124) (035 305)
- Sortare după cifra zecilor
- (305) (421 124) (430 532 035) (240)
- Sortare după cifra sutelor
- (035) (124) (240) (305) (421 430) (532)
- Tabloul sortat

-are aceeasi complexitate ca QuickSort dar foloste de doua ori mai multa memorie

Arborei parcurgeri

-parcurge=vizitarea tuturor nodurilor dintr-un arbore

-parcurgeri: ->parcure in adancime

->parcure pe nivel (in latime)

*Parcurea in Adancime

Algoritm Parcure_in_Adancime(arbore):

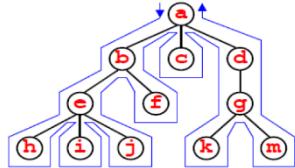
daca arbore vid atunci revenire;

prelucreaza informatia din radacina;

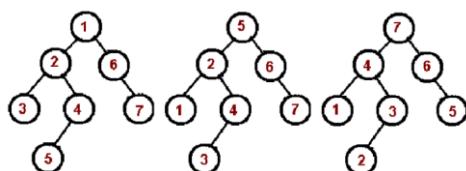
pentru fiecare subarbore repeta

 Parcure_in_Adancime(subarbore);

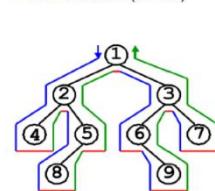
sfarsit



- Preordine: Rădăcină, Stânga, Dreapta (**RSD**)
- Inordine: Stânga, Rădăcină, Dreapta (**SRD**)
- Postordine: Stânga, Dreapta, Rădăcină (**SDR**)

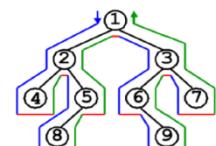


• Preordine (RSD)



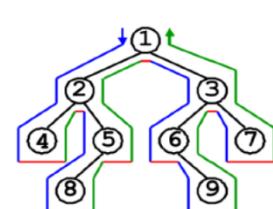
1 2 4 5 8 3 6 9 7

• Inordine (SRD)

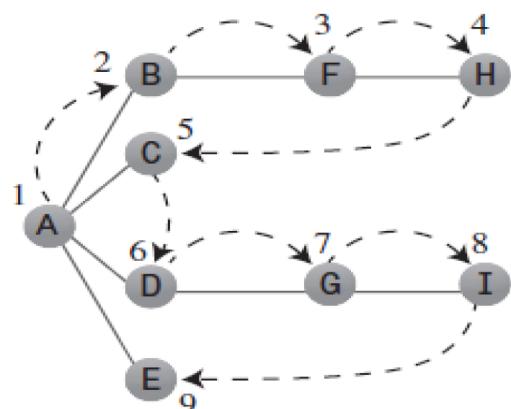


4 2 8 5 1 6 9 3 7

• Postordine (SDR)



4 8 5 2 9 6 7 3 1



```
void dfs(graph_mat G, int i, int **vis){
    (*vis)[i] = 1;
    printf("%d ", i);
    for(int j = 0; j < G.nr_nodes; j++)
        if(G.mat_adj[i][j] && !(*vis)[j])
            dfs(G, j, vis);
}
```

-Parcurea in inordine(arbore binar)=>parcurea in ordine crescatoare

*Reprezentarea arborilor binari prin vectori

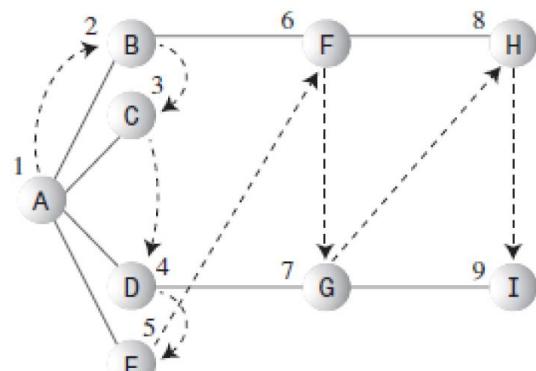
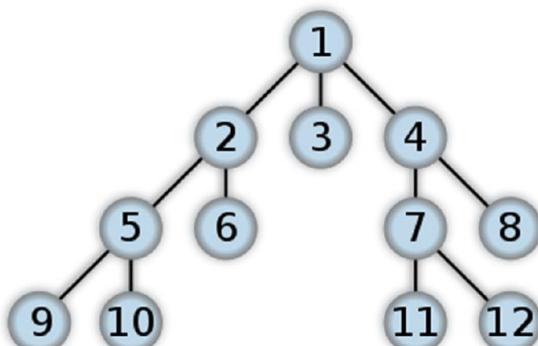
Reprezentarea arborilor binari prin vectori

- Radacina – indice $i = 1$
- Nodul cu indice i are subarborii la pozitiile $2*i$ si $2*i+1$
- Arbore de inaltimea h are $2^{(h+1)} - 1$ celule
- Daca cunosc indexul unui nod i , obtin indexul tatalui
 $i \text{ div } 2$



Reprezentare potrivita in cazul in care arborele este construit de la inceput echilibrat
si nu se mai modifica

*Parcursarea pe nivel (In Latime)

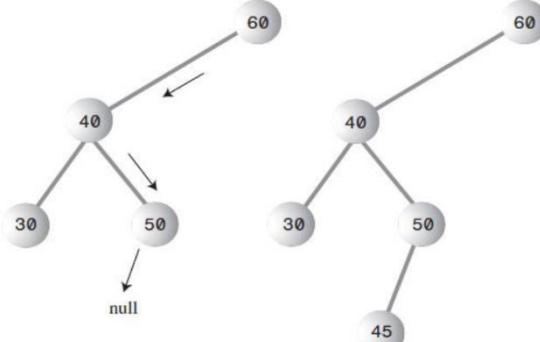


```
void bfs(graph_mat G, int i, int **vis){
    int *queue;
    queue = calloc(10*G.nr_nodes, sizeof(int));
    set_zeros(&queue, G.nr_nodes);
    int p = 0, u = 0;
    queue[p] = i;
    u++;
    (*vis)[i] = 1;
    while(p < u){
        printf("%d ", queue[p]);
        for(int j = 0; j < G.nr_nodes; j++)
            if(G.mat_adj[queue[p]][j] && (*vis)[j] == 0){
                queue[u++] = j;
                (*vis)[j] = 1;
            }
        p++;
    }
}
```

Arbore binary de cautare

-timpul necesar gasirii unui nod depinde de numarul de niveluri parcuse pana la gasire

-timpul este d.p. cu log binar al nr de noduri



a) Before insertion

b) After insertion

*Stergerea unui nod

1. nodul este frunza

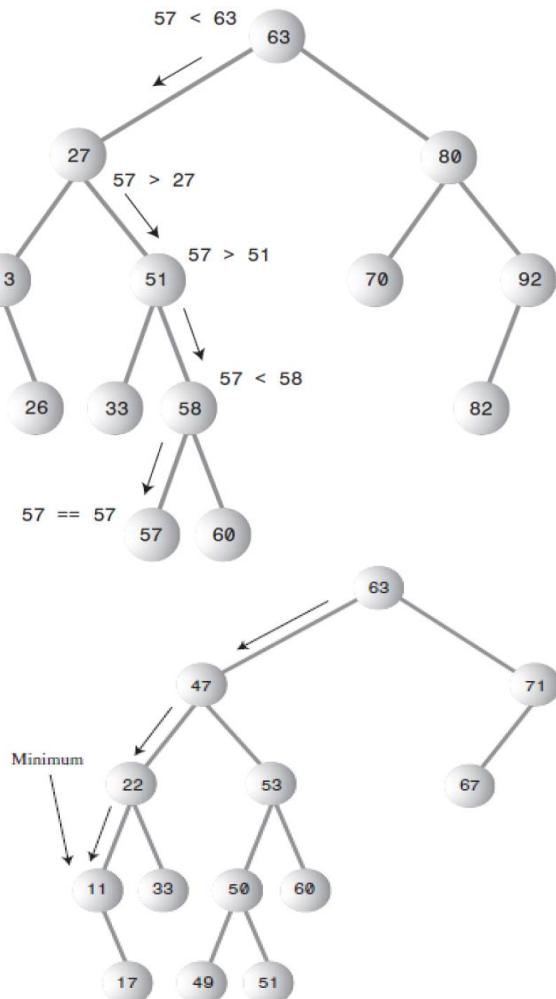
2. nodul are un singur fiu

3. nodul are 2 fii

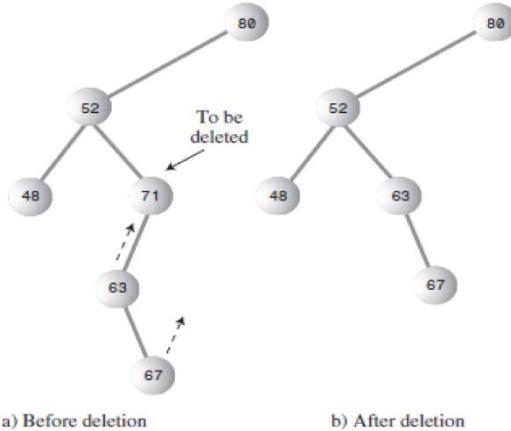
*nodul este frunza

-pointerul catre el se va face null (cel al parintelui)

-free(node)



a) Before deletion b) After deletion



a) Before deletion

b) After deletion

*nodul are un fiu

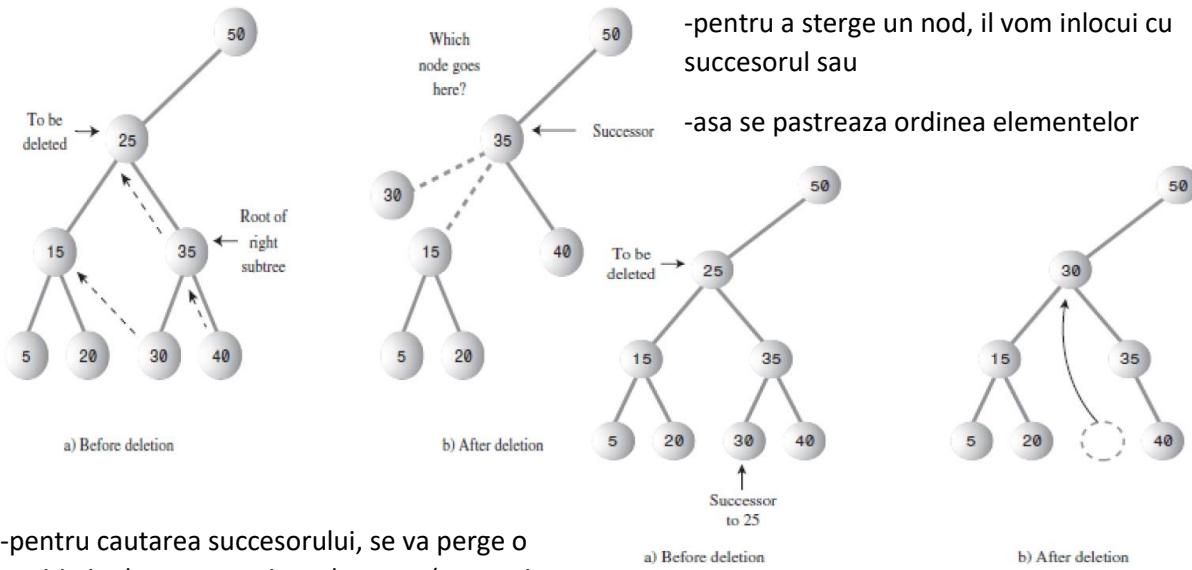
-nodul are un parinte si un fiu

-legam parintele de fiu (fiul nodului curent)

-free(node)

*nodul are 2 fi

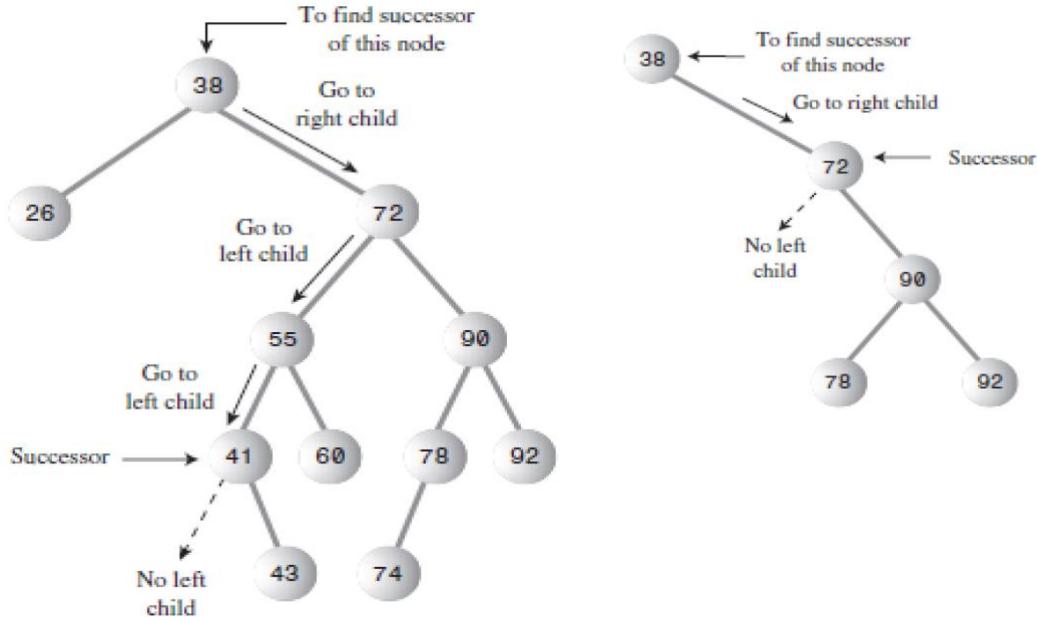
-nu poate fi inlocuit pur si simplu



-pentru cautarea succesorului, se va perge o pozitie in dreapta, apoi total stanga (asa gasim cel mai mic numar mai mare ca nodul pe care vrem sa il stergem)

-mutam valoare gasita in nodul pe care vrem sa il inlocuim

-stergem nodul pe care l-am mutat mai sus



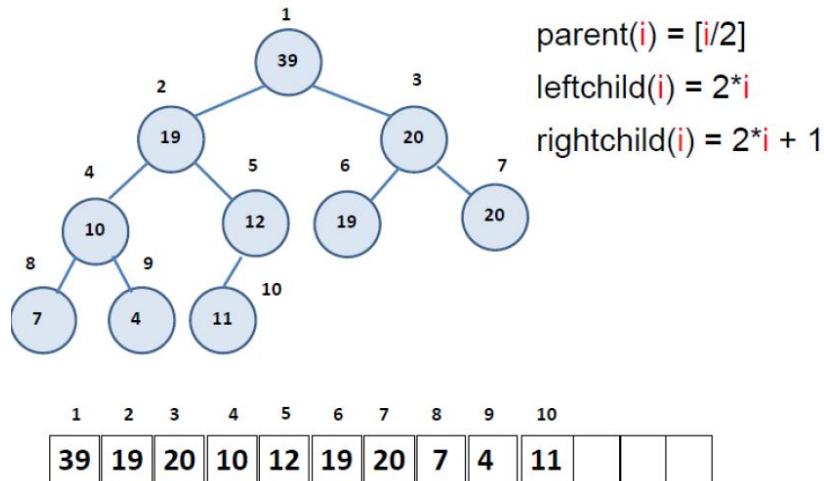
Movile

-un tip special de arbore

-permite implementarea eficienta a operatiilor cu cozi

-**Movina**=arbore binar complet

-Conditia de movila=nodul respectiv trebuie sa aiba o cheie mai mare sau egala decat oricare dintre fii sai (Max-Heap) ---min sau egala pt (MinHeap)



-tabloul=structura in care se gaseste fizic in memorie movila

-Movila=o reprezentare conceptuala

-intr-o movila Max-Heap, fiecare nod are o cheie mai mica decat parintele sau si mai mare ca fii sai

-este slab ordonata in comparative cu un arbore binar de cautare

-reprezentarea movilei cu elemente intregi

```
typedef struct {
    int v[M];
    int n;
} heap;
```

•Se definește o movilă cu maxim **M** elemente întregi și dimensiunea efectivă a tabloului **n**

*Stergerea

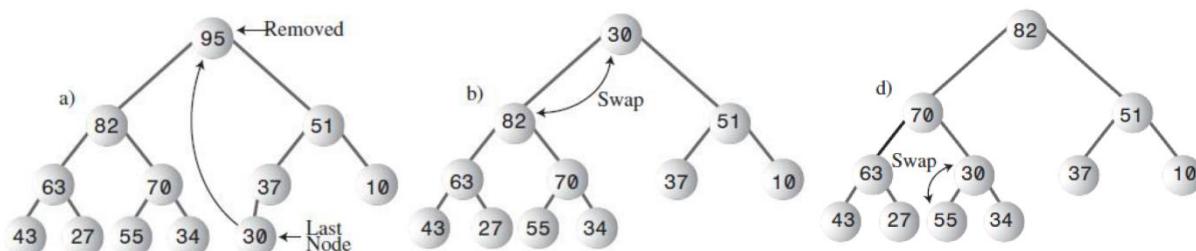
-stergerea elementului cu cheie maxima (max-Heap), celui cu cheie minima (min-heap)

-acest element reprezinta radacina arborelui

-dupa stergere arborele nu ramene complet

-**ultimul nod**=nodul cel mai din dreapta de pe ultimul nivel (cel care corespunde ultimei cellule din vector)

- 1. **Stergerea** propriu-zisă a rădăcinii
- 2. Deplasarea **ultimoiu nod** din arbore în nodul **rădăcină**
- 3. “**Filtrarea**” acestui nod în jos, până când ajunge dedesubtul unui nod mai mare decât el și deasupra unuia mai mic



*Algoritm de stergere (Heapify)

```

typedef struct{
    int v[10];
    int n;
}heap;
void swap(heap h, int i, int j)
{
    int t;
    t=h.v[i];
    h.v[i]=h.v[j];
    h.v[j]=t;
}
void heapify(heap h, int i)
{
    int st,dr,m,aux;
    st=2*i;
    dr=st+1;
    if(st<=h.n && h.v[st]>h.v[i])
        m=st;
    else m=i;
    if(dr<=h.n && h.v[dr]>h.v[m])
        m=dr;
    if(m!=i){
        swap(h,i,m);
        heapify(h,m);
    }
}

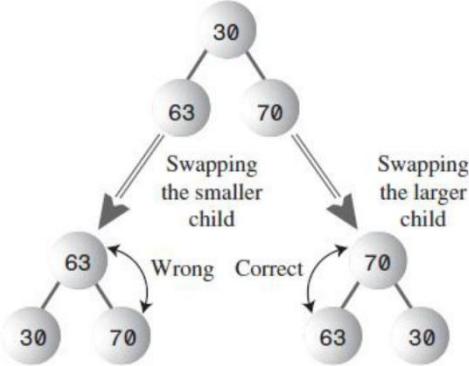
```

-reface o movila dintr-un arbore, la care elementul nu respecta conditia de movila

-se ajunge aici in cazul stergerii sau modificarii unei valori din radacina movilei

-pentru fiecare pozitie algoritmul determina care dintre fii acestuia este mai mare

-alg permute apoi nodul cu cel mai mare dintre fii



*Transformare tablou->movila

-se face treptat pornind de la Frunze spre radacina, cu ajustare la fiecare element

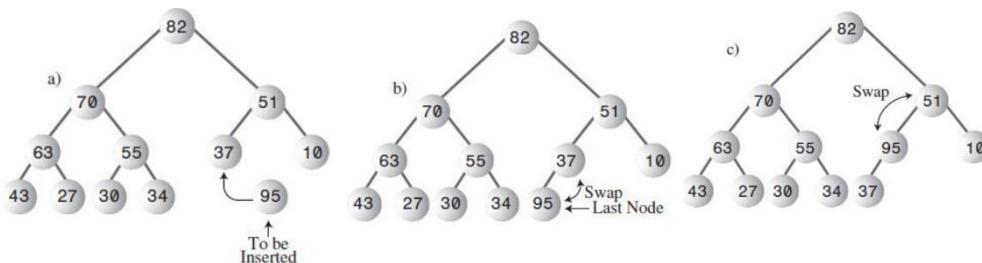
Operatie	Tablou	Arbore
initializare	1 2 3 4 5 6	1 2 3 4 5 6
heapify(3)	1 2 6 4 5 3	1 2 6 4 5 3
heapify(2)	1 5 6 4 2 3	1 2 6 4 5 3
heapify(1)	6 5 3 4 2 1	1 2 3 4 6 5 3 6 1

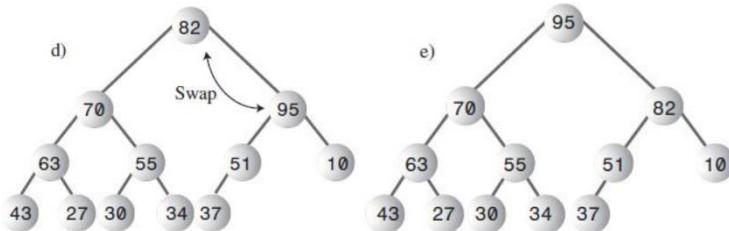
*Inserare

-se utilizeaza o filtrare in sus

-nodul este inserat in prima pozitie disponibila de la sfarsitul tabloului

-apoi trebuie facut heapify pt a se respecta conditia de movila





-Exemplu de inserare

Se inserează valoarea $val=7$ în tabloul $a=[8,5,6,3,2,4,1]$

$i=8$, $a[8]=7$ $a=[8,5,6,3,2,4,1,7]$

$i=8$, $a[4]=3 < 7$, $a[8]$ cu $a[4]$ $a=[8,5,6,7,2,4,1,3]$

$i=4$, $a[2]=5 < 7$, $a[4]$ cu $a[2]$ $a=[8,7,6,5,2,4,1,3]$

$i=2$, $a[1]=8 > 7$ $a=[8,7,6,5,2,4,1,3]$

-filtrarea in sus este mai usoara ca filtrarea in jos deoarece se va compara doar cu parintele, cu cei 2 fii

*Eficiența algoritmului HeapSort

-eficienta structurii de movila ofera posibilitatea folosirii acestui algoritm	După citire tablou	4,2,6,1,5,3
-se insereaza elemente neordonate intr-o movila	După makeheap	6,5,4,1,2,3
-la final se obtine un tablou ordonat crescator	După schimbare 6 cu 3	3,5,4,1,2,6
-pasii ordonare sir =>	După heapify	5,3,4,1,2,6
	După schimbare 5 cu 2	2,3,4,1,5,6
	După heapify	4,3,2,1,5,6
	După schimbare 4 cu 1	1,3,2,4,5,6
	După heapify	3,1,2,4,5,6
	După schimbare 3 cu 2	2,1,3,4,5,6
	După heapify	2,1,3,4,5,6
	După schimbare 2 cu 1	1,2,3,4,5,6

-timp de executie $O(N \cdot \log N)$

-are independent relative fata de distributia initiala a elementelor

Grafuri

-o modalitate de a reprezenta relatiile care exista intre obiecte

-**Graf**=o pereche de 2 multimi, cea a varfurilor si cea a legaturilor lor

-tipuri de grafuri:

->**neorientat** (undirected graph)

->**orientat** (directed graph, diagraph)

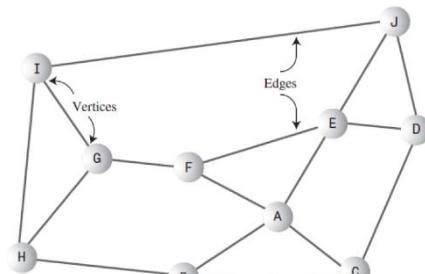
->**neorientat si ponderat** (undirected weighted graph)

->**orientat si ponderat** (directed weighted graph)

-varf + muchie => graf neorientat

-nod + arc => graf orientat

-capetele unei muchii=cele 2 vf legate de o muchie



- varfuri adiacent**=daca exista o muchie intre ele
- varfuri adiacente cu un varf**=vecinii varfului dat
- muchie (u,v) incidenta intr-un varf**=varful este u sau v
- graf orientat (u,v)** => v=destinatia, u=originea
- arce de iesire** ale unui nod v=arce orientate care au v origine
- arce de intrare** ale unui nod v= arce orientate care au v destinatie
- gradul unui nod**=nr de arce incidente ale nodului $d(v)$
- nod izolat**= $d(v)=0$
- gradul interior**(de intrare) al unui nod $d'(v)$ = nr arcelor care intra in nod
- gradul exterior**(de iesire) al unui nod $d''(v)$ =nr arcelor care ies din nod
- ordinul** unui graf= nr de varfuri din graf $n=|V|$
- dimensiunea** unui graf= nr de muchii $m=|E|$
- o cale (drum)** de lungime k=succesiune de noduri
- exista **cale a->b= b este accesibil din a**
- cale simpla**=are noduri distincte
- ciclu**=un drum circular
- o bucla**=un ciclu de luncime 1 (a,a)
- ciclu simplu**=toate nodurile distincte(exceptand nodul de plecare) si nu contine bucle
- graf aciclic**=nu are cicluri
- graf neorientat conex**=oricare 2 varfuri pot fi unite printr-o cale
- componente conexe**=clasele de echivalenta ale varfurilor prin relatia "este accesibil din"
- graf neconex**=alcatuit din mai multe componente conexe
- graf neorientat conex**=o singura componenta conexa
- graf dens** = $E \sim O(|V|^2)$
- graf rar** = $E \sim O(|V|)$
- graf orientat tare conex(puternic conectat)**=oricare doua noduri u si v, (1,2), (2,3), (3,4), (4,1) u e accesibil din v si v e accesibil din u
- graf orientat** (u,v) = $u \rightarrow v$, $(v,u)=v \rightarrow u$
- **noduri vecine sau adiacente** = muchii intre care exista un arc
- succesorii si predecesorii** unui nod=arce care ies si arce care intra in nod

Un **subgraf** al unui graf $G = (V, E)$ este un graf $G' = (V', E') \subseteq G$ cu $E' \subseteq E$ și $V' \subseteq V$, unde E' conține toate muchiile din E care au extremitățile în V' (se mai numește subgraf induș de V' în G)

->**subgraf**

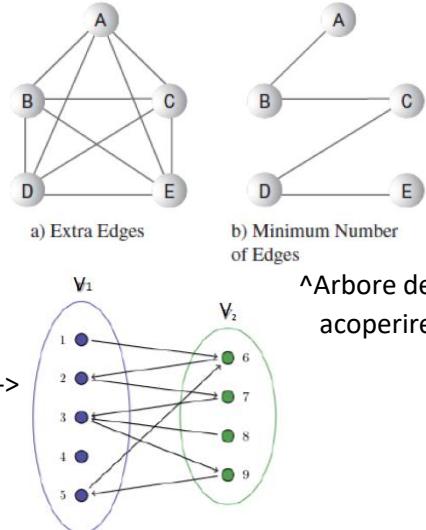
Graful $G' = (V, E')$ este un **subgraf de acoperire** (spanning subgraph) al grafului $G = (V, E)$ dacă $E' \subseteq E$ (se mai numește graf parțial)

->**subgraf de acoperire** (spanning subgraph)

-arbore liber=graf neorientat conex și aciclic

-padure=graf neorientat neconex și aciclic

-arbore de acoperire=subgraph de acoperire care este un arbore liber (arbore liber)



^Arbore de acoperire

-graf bipartite=graf în care multimea nodurilor se partionează ->

-graf regulat=toate varfurile au același grad

-graf k-regulat=toate varfurile au gradul k

-graf neorientat complet=oricare 2 varfuri sunt adiacente

-graf orientat complet=oricare 2 noduri sunt adiacente

-ciclu Hamiltonian=ciclu simplu în care se vizitează fiecare varf din graf **exact o dată**

-cale Hamiltoniana= cale simplă într-un graf

-ciclu Eulerian = ciclu care vizitează fiecare muchie din graf **exact o dată**

-cale Euleriana= cale care vizitează fiecare muchie din graf **exact o dată**

-graf neorientat cu m muchii => gradul varfurilor= $2*m$

P1: Fie un **graf neorientat** G cu m muchii și multimea de varfuri V

$$\sum_{v \in V} d(v) = 2 * m$$

-graf orirntat cu m arce => gradul nodurilor de intrare = g.n. ieșire= m

P2: Fie un **graf orientat** G cu m arce și multimea de noduri V

$$\sum_{v \in V} d^-(v) = \sum_{v \in V} d^+(v) = m$$

P3: Fie un **graf** G cu n varfuri și m muchii

Dacă G este **neorientat** atunci $m \leq n*(n-1)/2$

Dacă G este **orientat** atunci $m \leq n*(n-1)$

P4: Fie un **graf** G cu n varfuri și m muchii

Dacă G este **conex** atunci $m \geq n-1$

Dacă G este **arbore** atunci $m = n-1$

Dacă G este o **pădure** atunci $m \leq n-1$

*Operări de bază cu grafuri

initGraph – inițializare graf

numV – întoarce numărul de varfuri din graf

numE – întoarce numărul de muchii din graf

getEdge(u,v) – întoarce 1 dacă există muchia dintre u și v , altfel întoarce 0

deg(v) – întoarce gradul varfului v

indeg(v) – întoarce gradul interior (de intrare) al nodului v

outdeg(v) – întoarce gradul exterior (de ieșire) al nodului v

insertVertex(x) – creează și întoarce un varf nou ce memorează elementul x

removeVertex(v) – elimină varful v și toate muchiile incidente lui

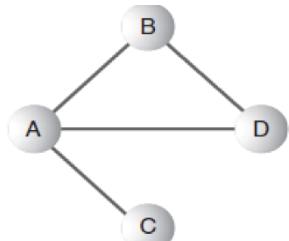
endVertices(e) – întoarce un vector ce conține cele 2 varfurile care definesc muchia e ; dacă graful este orientat, primul nod este originea, cel de al doilea este destinația

insertEdge(u,v,x) – creează și întoarce o muchie nouă de la varful v la varful u și o etichetează cu x ; eroare dacă muchia există deja

removeEdge(e) – elimină muchia e din graf

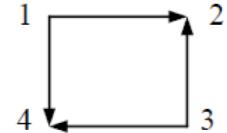
*Modalitati de reprezentare

-matrice de adiacenta (N noduri => N*N elem are matricea)



	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

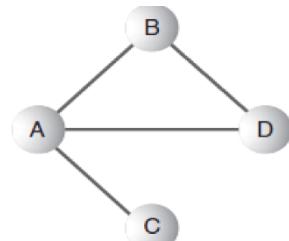
1	2	3	4
1	0	1	0
2	0	0	0
3	0	1	0
4	0	0	0



->e dezavantajoasa daca nr de noduri este mult mai mare ca nr de arce

->**matrice rara**=cu peste jumata elem nenule

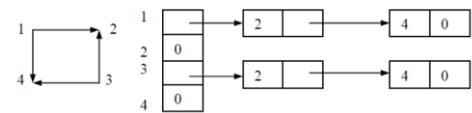
-lista de adiacenta



Vertex	List Containing Adjacent Vertices
A	B—>C—>D
B	A—>D
C	A
D	A—>B

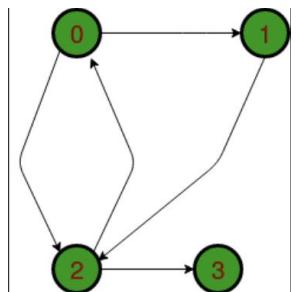
->ordinea varfurilor dintr-o lista de adiacenta nu este importanta

- Reprezentarea grafului orientat (1,2),(1,4),(3,2),(3,4) printr-un tabou de pointeri la liste de adiacenta



Algoritmul lui Warshall

-ne arata daca exista un drum intre 2 noduri, care nu sunt direct legate

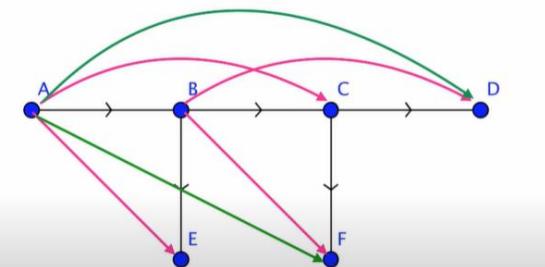


-1 este legat de 2, dar 2 nu este legat de 1

-din 2 nu poti ajunge direct in 1, dar 2->0->1

-2 de 1 este legat prin 0

Warshall's Algorithm: The Concept



-meru se cauta conectari de grad 2 (ex 2->0->1), apoi se marcheaza in matrice cu 1 2->1, apoi se cauta alte conectari de grad 2 si tot asa pentru fiecare nod pana nu mai exista

-pasul asta se repeat de cate ori se vrea

Graf neorientat ->gasirea varfurilor conectate <- cu DFS sau BFS

Graf orientat ->gasirea tuturor nodurilor conectate<- DFS pornind, pe rand, din fiecare nod (asa alegem drumul care ne convine)

Pasi:

1.copiem matricea de adiacenta

2.cautam daca exista vreo relatie de genul A->B->C (practice un nod la care se poate ajunge tracand prin alt nod, practice un drum de 2 pasi) => daca exista => il notam cu 1 ca si cum ar fi direct

3. se repeat pasul pentru fiecare nod in parte de cate ori este nevoie pana nu mai exista cai de genul (sau de cate ori se doreste)

```

int max(int,int);
void warshall(int p[10][10],int n) {
    int i,j,k;
    for (k=1;k<=n;k++)
        for (i=1;i<=n;i++)
            for (j=1;j<=n;j++)
                p[i][j]=max(p[i][j],p[i][k]&&p[k][j]);
}
int max(int a,int b) {
    ;
    if(a>b)
        return(a); else
        return(b);
}

Warshall(A[1...n, 1...n]) // A is the adjacency matrix
R(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R(k)[i, j] ← R(k-1)[i, j] or (R(k-1)[i, k] and R(k-1)[k, j])
return R(n)

```

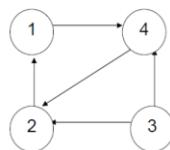
DFS(Depth First Search)-adancime

-Stiva(LIFO)

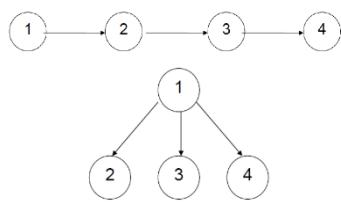
-timp de executie: → liste de adiacenta => O(V+E) V=varfuri, E=arce

→ matrice de adiacenta => O(V²)

Explorarea DFS din nodul 3 a grafului orientat produce secvență de noduri 3, 2, 1, 4, iar arborele de acoperire este format din arcele 3-2, 2-1 și 1-4



Același arbore de acoperire la explorarea DFS și BFS



-algoritm iterativ

-pentru un rezultat bun al DFS-ului iterative, succesorii nodului vor fi pusii in ordine descrescatoare a valorilor lor si extragerea lor de pe stiva se va face in ordine inversa

```

pune nodul de start în stivă
repeta cât timp stiva nu e vidă
    scoate din stivă nodul x
    afişare și marcare nodul x
    pune în stivă orice succesor nevizitat y al lui x

```

Presupunem că graful are n noduri și este prezentat prin matricea de adiacență a .

Starea unui vîrf (vizitat sau nu) este memorată în vectorul caracteristic v . Toate

-alg recursiv

aceste variabile sunt globale.

```

void dfs(int k)
{
    v[k]=1; //vizitam varful curent x
    for(int i=1;i<=n;i++) // determinam vecinii nevizitati ai lui x
        if(a[k][i]==1 && v[i]==0)
        {
            dfs(i); // continuam parcurgerea cu vecinul curent i
        }
}

```

c

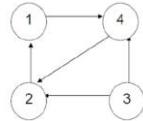
Funcția recursivă de explorare DFS

```
// explorare DFS dintr-un nod dat v
void dfs (Graf g, int v, int vazut[]) {
    int w, n = g.n; // n = numar noduri din graful g
    vazut[v] = 1; // marcare v ca vizitat
    printf("%d ", v); // afisare (sau memorare)
    for (w = 1; w <= n; w++)
        // repeta pentru fiecare vecin posibil w
        if (getEdge(g, v, w) && vazut[w] == 0)
            // daca w este un vecin nevizitat al lui v
            dfs(g, w, vazut); // continua explorarea din w
}
```

Funcția iterativă de explorare DFS

```
void dfs (Graf g, int v, int vazut[]) {
    int x, y;
    Stack s; // s este o stiva de numere naturale
    s = initStack(); // initializare stiva
    push(s, v); // pune nodul v pe stiva
    while (!isEmptyStack(s)) {
        x = pop(s); // scoate din stiva nodul x
        vazut[x] = 1; // marcheaza x ca vizitat
        printf("%d ", x); // afiseaza nodul x
        for (y = g.n; y >= 1; y--)
            // cauta un vecin al lui x, care este nevizitat
            if (getEdge(g, x, y) && !vazut[y]) {
                vazut[y] = 1;
                push(s, y); // pune nodul y pe stiva }
    }
```

Explorarea BFS din nodul 3 a grafului
conduce la secvența de noduri 3, 2, 4, 1 și
la arborele de acoperire 3-2, 3-4, 2-1,
dacă se folosesc succesorii în ordinea
crescătoare a valorilor acestora



Funcția de explorare în lățime

```
// explorare in latime dintr-un nod dat v
void bfs (Graf g, int v, int vazut[]) {
    int x, y;
    Queue q; // q este o coada de numere naturale
    q = initQueue();
    vazut[v] = 1; // marcare v ca vizitat
    enqueue(q, v); // pune pe v in coada
```

*Drum minim

-drumul care foloseste cel mai mic numar de muchii

->pentru afisarea drumului parcurs de la v la x,
trebuie parcurs in sens invers tabloului p (de la
ultimul element la primul)

$x \rightarrow p[x] \rightarrow p[p[x]] \rightarrow \dots \rightarrow v$

Funcția pentru vizitarea tuturor nodurilor

```
// explorare graf prin vizitarea tuturor nodurilor
// se porneste din primul nod
void df (Graf g) {
    int vazut[M] = {0}; // multime noduri vizitate
    int v;
    for (v = 1; v <= g.n; v++)
        if (!vazut[v])
            printf("\n Explorare din nodul %d \n", v);
            dfs(g, v, vazut);
    }
```

BFS(Breadth First Search)-latime

-coada(FIFO)

-timp de executie:->liste de adiacenta => O(V+E)
V=varfuri, E=arce

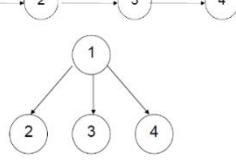
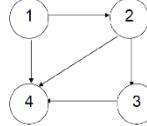
->matrice de adiacenta => O(V²)

Exemplu

Același arbore de acoperire la explorarea DFS și BFS

- Se consideră graful orientat cu arcele:

- (1,2), (1,4), (2,3), (2,4), (3,4)
- Explorarea BFS este: 1 → 2 → 4 → 3



```
while (!isEmptyQueue(q)) {
    x = dequeue(q); // scoate pe x din coada
    for (y = 1; y <= g.n; y++)
        // repeta pentru fiecare potential vecin cu x
        if (getEdge(g, x, y) && vazut[y] == 0) {
            // daca y este vecin cu x si nevizitat
            printf("%d - %d \n", x, y);
            // scrie muchia x-y
            vazut[y] = 1; // y vizitat
            enqueue(q, y); // pune y in coada
        }
    }
```

Calcul distanța minimă

```
//distanta minima de la v la toate celelalte noduri din graf
void dist_min (Graf g, int v, int vazut[], int d[], int p[]) {
    int x, y;
    Queue q;
    q = initQueue();
    vazut[v] = 1;
    d[v] = 0;
    p[v] = 0;
    enqueue(q, v); // pune pe v in coada
    while (!isEmptyQueue(q)) {
        x = dequeue(q); // scoate pe x din coada
        for (y = 1; y <= g.n; y++)
            if (getEdge(g, x, y) && vazut[y] == 0) {
                // daca exista arc intre x si y si y nu a fost vizitat
                vazut[y] = 1;
                d[y] = d[x] + 1;
                // y este un nivel mai jos decat x
                p[y] = x;
                // x este predecesorul lui y pe drumul minim
                enqueue(q, y);
            }
    }
```

Grafuri Ponderate Algoritmul lui Prim

- arbore de acoperire(spanning tree) = subgraph de acoperire care e un arbore liber
- arbore de acoperire de cost minim= intr-un graf ponderat sa fie un arbore de acoperire cu suma costurilor muchiilor mai mica sau egala cu suma costurilor muchiilor din orice alt arbore de acoperire din graf
- graf conex cu n varfuri => arborii de acoperire au exact n-1 muchii

*Algoritmul lui Prim

- determina arborele de acoperire de cost minim
- porneste de la arborele vid si incerca sa adauge cate o muchie
- cat timp arborele nu contine toate muchiile din graf=>se alerge una cu cost minim legata la arborele parcial construit (daca nu formeaza cicluri)

```
1  #include<stdio.h>           // Prim's Algorithm in C
2  #include<stdbool.h>
3  #define INF 9999999
4  #define V 5                  // number of vertices in graph
5  int G[V][V] = {              //matrice de costuri
6      {0, 9, 75, 0, 0},
7      {9, 0, 95, 19, 42},
8      {75, 95, 0, 51, 66},
9      {0, 19, 51, 0, 31},
10     {0, 42, 66, 31, 0}};
11
12 int main() {
13     int no_edge;             // number of edge
14     int selected[V];        //vector de vizitati
15     memset(selected, false, sizeof(selected));
16     no_edge = 0;             //inceudem cu nodul 0
17     selected[0] = true;      //il marcam ca vizitat
18     int x;                  // row number
19     int y;                  // col number
20     printf("Edge : Weight\n"); // print for edge and weight
21     while (no_edge < V - 1) { //For every vertex in the set S, find the all adjacent vertices
22         int min = INF;       // , calculate the distance from the vertex selected at step 1.
23         x = 0;               // if the vertex is already in the set S, discard it otherwise
24         y = 0;               //choose another vertex nearest to selected vertex at step 1
25         for (int i = 0; i < V; i++)
26             if (selected[i])
27                 for (int j = 0; j < V; j++)
28                     if (!selected[j] && G[i][j]) // not in selected and there is an edge
29                         if (min > G[i][j]) {
30                             min = G[i][j];
31                             x = i;
32                             y = j;
33                         }
34                     printf("%d - %d : %d\n", x, y, G[x][y]);
35                     selected[y] = true;
36                     no_edge++;
37     }
38 }
39 }
```

Sortare topologica a grafurilor orientate

- algoritm care poate fi modelat cu ajutorul grafurilor
- util pentru cand este necesara disponerea intr-o anumita ordine a elementelor
- Scop:** Ordonarea elem astfel incat fiecare sa fie precedat de elemental de care e conditionat.

-nu e posibila: -> daca graful contine un ciclu

->nu exista niciun element fara conditionari

-uneori este posibila doar o sortare topologica parcial

-analiza a drumului critic= modelarea planificarii unor activitati cu ajutorul grafurilor

-Solutii:

->pronind de la un element fara predecesori

->pornind de la elemente fara predecesori

->folosind DFS, cu afisarea nodurilor din care incepe exploatarea, dup ace s-au exploatat toate celelalte noduri

*Sortare topologica cu liste de predecesori

repetă
 cauta un nod nemarcat si fara predecesori
dacă s-a gasit **atunci**
 afiseaza nod si marcheaza nod
 sterge nod marcat din graf
până când nu mai sunt noduri fara predecesori
dacă ramâne noduri nemarcate **atunci**
 nu este posibilă sortarea topologică

Se obtine sortarea topologica: 2, 1, 4, 3, 5



```
// functia determina numarul de conditionari ale nodului v
int nrcond (Graf g, int v) {
    int j, cond = 0; // cond = numar de conditionari
    for (j = 1; j <= g.n; j++)
        if (getEdge(g, j, v))
            cond++;
    return cond;
}

// functia de sortare topologica si afisarea rezultatului
void topsort (Graf g) {
    int i, j, n = g.n, ns, gasit, sortat[50] = {0};
    ns = 0; // noduri sortate si afisate
    do {
        gasit = 0;
        // cauta un nod nesortat, fara conditionari
        for (i = 1; i <= n && !gasit; i++)
            if (!sortat[i] && nrcond(g, i) == 0) { // i fara conditionari
                gasit = 1;
                sortat[i] = 1;
                ns++; // noduri sortate
                printf("%d ", i); // afiseaza nod gasit
                removeVertex(g, i); // elimina nodul i din graf
            }
    } while (gasit);
    if (ns != n) printf("\n Sortare topologica imposibila !");
}
```

*Sortare topologica cu liste de succesori

repetă
 cauta un nod fara succesori
 pune nod gasit in stiva si marcheaza ca sortat
 elimină nod marcat din graf
până când nu mai există noduri fara succesori
dacă nu mai sunt noduri nemarcate **atunci**
repetă
 scoate nod din stiva si afiseaza nod
până când stiva e goală

La extragerea din stiva se obtine: 2, 4, 1, 3, 5

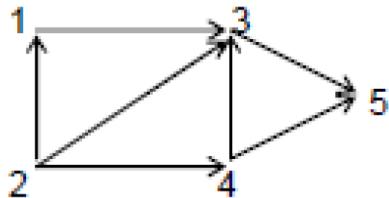
*Sortare topologica folosind exploatarea in adancime (derivate din DFS)

```
Stack s; // stiva se foloseste in doua functii
// sortare topologica pornind dintr-un nod v
void ts (Graf g, int v) {
    vazut[v] = 1;
    for (int w = 1; w <= g.n; w++)
        if (getEdge(g, v, w) && !vazut[w])
            ts(g, w);
    push(s, v);
}
```

```
// sortare topologica a grafului g
int main () {
    int i, j, n; Graf g;
    readG(g); n = g.n;
    for (j = 1; j <= n; j++)
        vazut[j] = 0;
    s = initStack();
    for (i = 1; i <= n; i++)
        if (vazut[i] == 0)
            ts(g, i);
    while (!isEmptyStack(s)) { // scoate din stiva si afiseaza
        i = pop(s);
        printf("%d ", i);
    }
}
```

Exemplu

- Secvență de apeluri și evoluția stivei pentru graful orientat:
- $(2,1), (1,3), (2,3), (2,4), (4,3), (3,5), (4,5)$

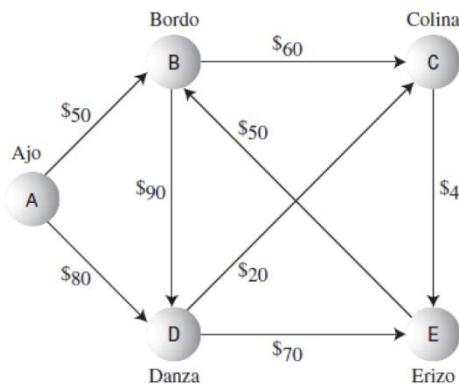


Apel	Stiva	Din
ts(1)		main()
ts(3)		ts(1)
ts(5)		ts(3)
push(5)	5	ts(5)
push(3)	5, 3	ts(3)
push(1)	5, 3, 1	ts(1)
ts(2)		main()
ts(4)		ts(2)
push(4)	5, 3, 1, 4	ts(4)
push(2)	5, 3, 1, 4, 2	ts(2)

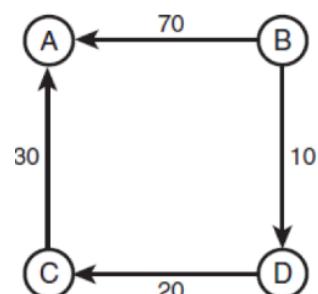
Algoritmul lui Floyd

-aflarea costului minim din orice nod catre orice alt nod pentru un graf orientat si ponderat

-all-pairs shortest path



	A	B	C	D	E
A	—	50	100	80	140
B	—	—	60	90	100
C	—	90	—	180	40
D	—	110	20	—	60
E	—	50	110	140	—



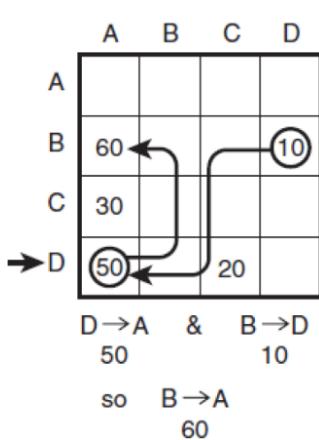
A	B	C	D
A			
B	70		
C	30		
D			20

A	B	C	D
A			
B	70		
C	30		
D			20

$\rightarrow C \xrightarrow{30} A$ & $D \xrightarrow{20} C$

$30 \xrightarrow{} D \xrightarrow{50} A$

$50 \xrightarrow{} B \xrightarrow{60} A$



-in cazul cailor multiple de cost diferit, se va allege costul ce la mic

-ex: aici B->A costul e de 70 dar B->D->A costul e de 60 =>

Pentru B->A avem cost de 70 si 60, $60 < 70 \Rightarrow$ inclocuim casuta costului B->A cu 60 pt ca e costul de drum minim

```

int min(int,int);
void floyds(int p[10][10],int n) {
    int i,j,k;
    for (k=1;k<=n;k++)
        for (i=1;i<=n;i++)
            for (j=1;j<=n;j++)
                if(i==j)
                    p[i][j]=0; else
                    p[i][j]=min(p[i][j],p[i][k]+p[k][j]);
}
int min(int a,int b) {
    if(a<b)
        return(a); else
        return(b);
}

```

-se va face pe o matrice de costuri, unde

$i=j \Rightarrow 0$

$i \rightarrow j$ există cost $\Rightarrow \text{cost}[i,j]$

$i \rightarrow j$ nu există cost $\Rightarrow \text{inf}$ (valoare arbitrară mare)

```

AlgoritmFloyd() {
    pentru toate liniile i execută
        pentru toate coloanele j execută
            A[i,j] ← cost[i,j]
    pentru toate liniile i execută
        A[i,i] ← 0
    pentru k de la 1 la n execută
        pentru toate liniile i execută
            pentru toate coloanele j execută
                dacă A[i,k] + A[k,j] < A[i,j] atunci
                    A[i,j] ← A[i,k] + A[k,j]
}

```

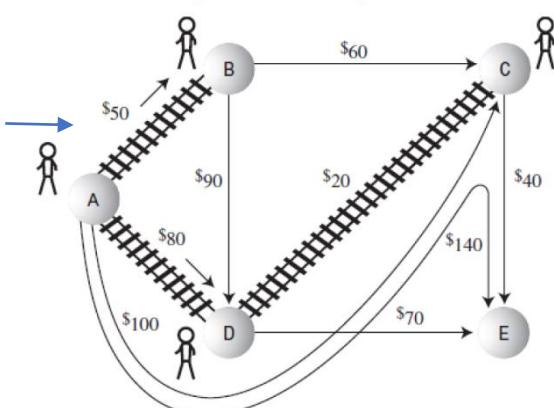
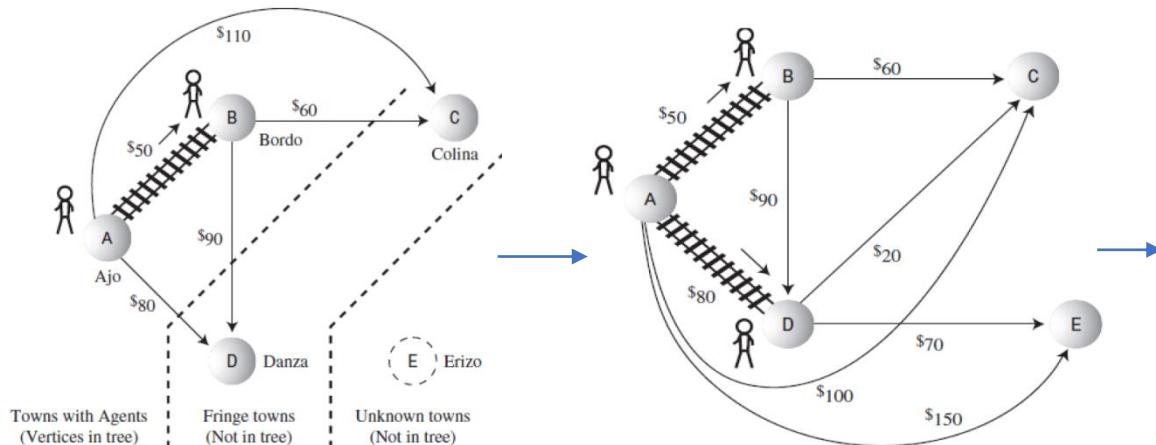
Algoritmul lui Dijkstra (problema drumului minim)

-aplicatie pentru **grafuri orientate si ponderate** (pot fi conexe sau neconexe)

-determinarea **celui mai scurt drum** dintre 2 noduri

-**drum minim**=cea mai ieftina ruta

-Regula: Mereu se merge in "Orasul" in care se ajunge din "Start" cu cem mai mic cost



-se allege nodul de start

-algoritmul se termina cand se cunosc costurile din A catre fiecare oras

```

void Dijkstra(float **Mat,int n,int start, int end, float *total)      //algoritmul Dijkstra
{
    int ctl=0;
    int ct2=0;
    printf("%d\n", end);
    int *path=malloc(2*n*sizeof(int));
    int pas=0;
    while(pas<2){
        float cost[n][n],distance[n],mindistance;
        int pred[n];
        int visited[n],count,nextnode,i,j;
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                if(Mat[i][j]==0)
                    cost[i][j]=INFINITY;
                else cost[i][j]=Mat[i][j];
        for(i=0;i<n;i++){
            distance[i]=cost[start][i];
            pred[i]=start;
            visited[i]=0;
        }
        distance[start]=0;
        visited[start]=1;
        count=1;
        while(count<n-1){
            mindistance=INFINITY;
            for(i=0;i<n;i++)
                if(distance[i]<mindistance&&!visited[i]){
                    mindistance=distance[i];
                    nextnode=i;
                }
            visited[nextnode]=1;
            for(i=0;i<n;i++)
                if(!visited[i])
                    if(mindistance+cost[nextnode][i]<distance[i]){
                        distance[i]=mindistance+cost[nextnode][i];
                        pred[i]=nextnode;
                    }
            count++;
        }
    }
}

```

Ideile algoritmului lui Dijkstra

- 1. De fiecare dată când ne aflăm într-un oraș nou, actualizăm lista de costuri
- În listă reținem numai drumul de **cost minim** (cunoscut până în momentul curent) dintre punctul de pornire și un alt oraș precizat
- 2. Mergem întotdeauna în orașul care are calea cea mai ieftină față de punctul de pornire

Algoritmul lui Kruskal

-gaseste arborele minim de acoperire pentru un graf conex ponderat

-gaseste submultimea muchiilor, care formeaza un **arbore** (include toate varfurile) cu un **cost minimal**

-graful nu e conex => face doar un arbore parțial

-este un algoritm **greedy**

-**utilizat mai ales cand numarul de muchii este mic**

-pentru n^2 muchii => se recomanda Algoritmul Prim

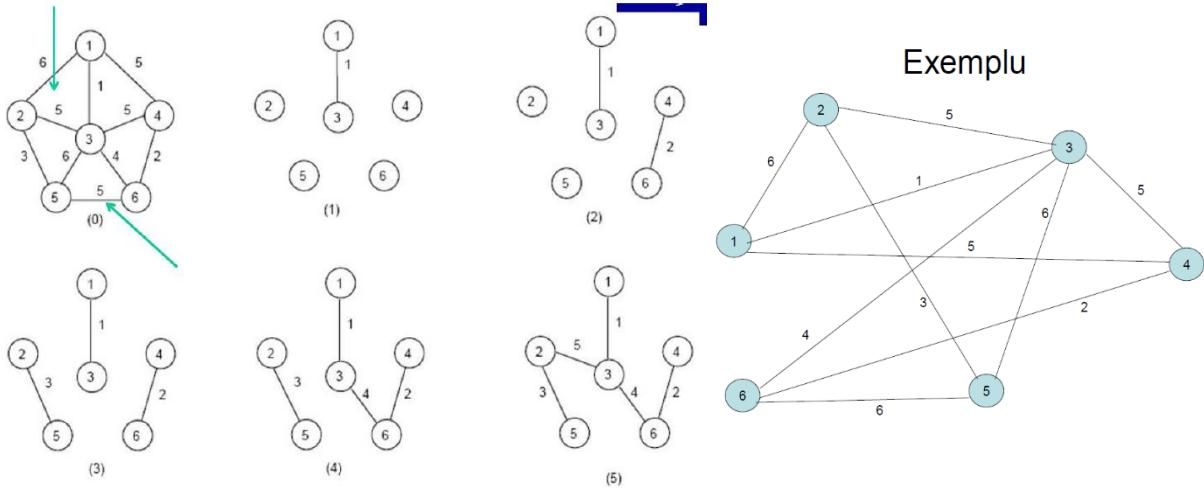
-Pasi:

1.Fiecare varf va fi un arbore.

2.(pas care se executa de $n-1$ ori) Se cauta muchia de cost minim care unește varfurile ale 2 arbori diferiti. Se selecteaza muchia.

->Dupa selectarea ultimei muchii ($n-1$) se extrage muchia de cost minim

->Nu se doreste formarea unui ciclu



Exemplu

```

1 // Kruskal's algorithm in C
2 #include <stdio.h>
3 #define MAX 30
4 typedef struct edge {
5     int u, v, w;
6 } edge;
7 typedef struct edge_list {
8     edge data[MAX];
9     int n;
10 } edge_list;
11 edge_list elist;
12 int Graph[MAX][MAX], n;
13 edge_list spanlist;
14
15 void kruskalAlgo() {
16     int belongs[MAX], i, j, cno1, cno2;
17     elist.n = 0;
18     for (i = 1; i < n; i++)
19         for (j = 0; j < i; j++)
20             if (Graph[i][j] != 0) {
21                 elist.data[elist.n].u = i;
22                 elist.data[elist.n].v = j;
23                 elist.data[elist.n].w = Graph[i][j];
24                 elist.n++;
25             }
26     sort();
27     for (i = 0; i < n; i++)
28         belongs[i] = i;
29     spanlist.n = 0;
30     for (i = 0; i < elist.n; i++) {
31         cno1 = find(belongs, elist.data[i].u);
32         cno2 = find(belongs, elist.data[i].v);
33         if (cno1 != cno2) {
34             spanlist.data[spanlist.n] = elist.data[i];
35             spanlist.n = spanlist.n + 1;
36             applyUnion(belongs, cno1, cno2);
37         }
38     }
39 }
```

// Applying Kruskal Algo

```

int find(int belongs[], int vertexno) {
    return (belongs[vertexno]);
}
void applyUnion(int belongs[], int c1, int c2) {
    int i;
    for (i = 0; i < n; i++)
        if (belongs[i] == c2)
            belongs[i] = c1;
}
void sort() {           // Sorting algo
    int i, j;
    edge temp;
    for (i = 1; i < elist.n; i++)
        for (j = 0; j < elist.n - 1; j++)
            if (elist.data[j].w > elist.data[j + 1].w) {
                temp = elist.data[j];
                elist.data[j] = elist.data[j + 1];
                elist.data[j + 1] = temp;
            }
}
```

Tabele de dispersie

- structuri ce ofera inserare si cautare rapida (timp O(1))
- dispersia datelor**=transformarea unui domeniu de valori ale unei anumite chei intr-un domeniu de indicia dintr-un tablou
- hash function**=functie care face dispersia datelor
- unele chei nu au nevoie de aplicarea acestei functii, ele pot fi utilizate direct ca indicii in tablou
- fuctia de dispersie** -> converteste un nr dintr-un domeniu mai mare intr-un nr cu domeniu mai mic
- corespondenta** -> se face printre-un tablou
- tablou de dispersie**=tablou in care sunt inserate elemente utilizand o functie de dispersie
- restrangerea domeniului** -> poate face ca functia de dispersie sa nu fie injective (si se pot incurca elemente)
- Solutia:** (stim ca tabloul are de 2 ori mai multe celule decat elemente)

1.la aparitia unei coliziuni => se cauta in tablou (**intr-un mod stabilit**) o celula libera=>se insereaza elementul nou in celula, in locul celei obtinute prin aplicarea functiei de dispersie (**Adresare deschisa**)

2.crearea unei liste inlantuite, la intalnirea aceluiasi indice cuvantul va fi adaugat la lista cu indicele dorit (**inlantuire separate**)

*Adresare deschisa

- Tipuri: ->**sondaj linear** (linear probing)
- >**sondaj patratnic** (quadratic probing)
- >**dubla dispersie** (double hashing)

*Sondaj liniar

Într-un sondaj liniar, dacă indicele inițial de dispersie este x , celulele sondate ulterior vor fi $x+1, x+2, x+3, \dots$

-**cauta secvential o celula libera** (indexul creste pana gasim ceva liber)

-**secventa ocupata** (filled sequence) = secventa de cellule ocupate dintr-o tabela de dispersie

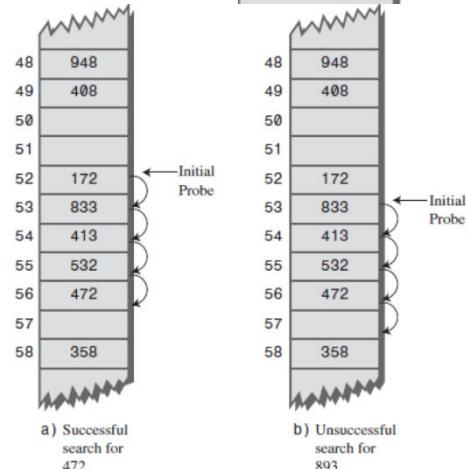
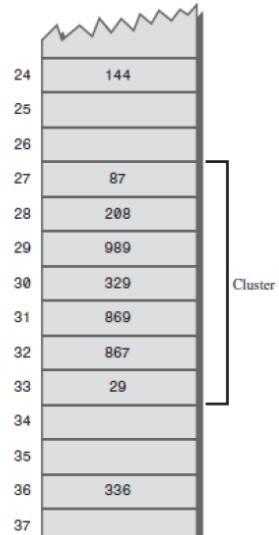
-**clustering**=ocuparea din ce in ce mai mult a secventelor (lungirea lor)

-**umplerea unei tabele** de dispersie este ineficienta

-la **umplerea ei**, niciun algoritm nu mai functioneaza

-la **cautarea unei chei** => se aplica functia de dispersie => se obtine indicele din tablou

-**celula dorita este ocupata** => coliziune => alg va cauta in continuare



-**sondaj** (probing)= process de cautare a celulei potrivite

-la gasirea unei cellule libere inaintea cheii cautate => operatie esuata

-lungimea sondajului (**probe length**)=numarul pasilor efectuati pana la gasirea celulei

-**coeficient de incarcare**=raportul dintre nr de elem din tabela si dimensiunea ei maxima

-apar clustere => performanta scazuta

O tabelă cu 10.000 de celule și 6.667 de elemente are un coeficient de încărcare de 2/3

$$\text{loadFactor} = \text{noItems} / \text{arraySize}$$

*Sondaj Patratic

Sondajul patratic presupune examinarea celulelor $x+1, x+4, x+9, x+16, \dots$

Distanța dintre acestea și celula inițială este egală cu pătratul numărului iterătiei curente

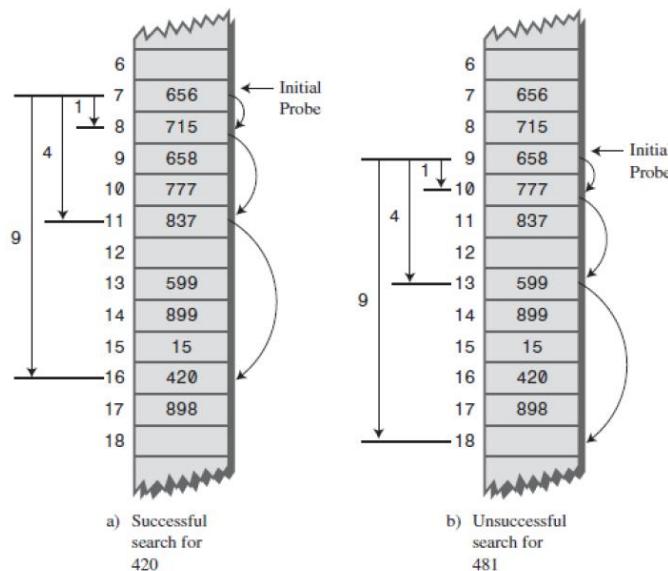
-sondarea celulelor pe o arie mai mare

-mareste pasul de cautare la fiecare iteratie

-se recomanda ca dimensiunea tableei sa fie un numar prim (pentru a evita aparitia unei secvente infinite de iteratii)

-rezolva problema clusterelor din Sondajul linear (**acumulare primara**)

-Dezavantaj: -> produc alt tip de acumulari => **acumulare secundara** (daca adaugam elemente cu acelasi indice va creste foarte mult tabela si va sari peste multe locuri libere inutil)



*Dubla dispersie

-rezolva problemele de mai sus

-este necesara generarea unei secvente de sondaj, care depinde de fiecare cheie in parte

-valori diferite asociate aceluiasi indice vor utiliza secvente de sondaj diferite

-cheia trece prin functie de dispersie => apoi trece prin alta functie de dispersie=> rezultatul obtinut se utilizeaza ca pas de deplasare

-pentru o cheie pasul va ramane acelasi pt sondaj=> se modifica alta cheie

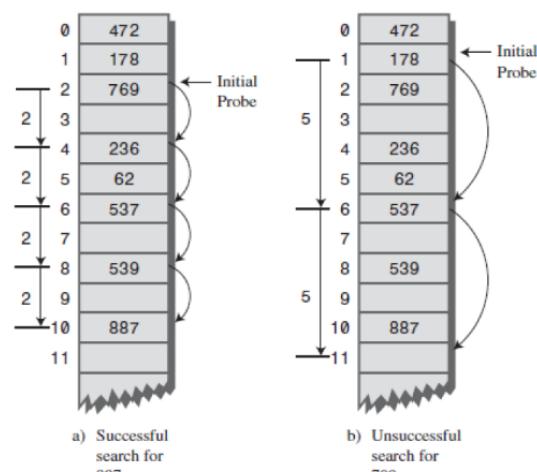
-**functia secundara** nu trebuie sa coincide cu cea primara

Exemplu:

$$\text{stepSize} = 5 - (\text{key} \% 5)$$

sau

$$\text{stepSize} = 7 - (\text{key} \% 7)$$



- rezultatul ei **nu trebuie sa fie 0** (pentru ca sondajele sa nu stationeza ce aceeasi celula)
- dimensiunea tabelei trebuie sa fie un **nr prim**

*Inlantuire separata(tratare coliziuni)

- fiecare celula din tabela sa contine o inlantuire separate
- nu este necesara cautarea unei alte cellule
- o tabela cu N celule are cel putin N elemente
- multe elemente** => timp crescut de cauta (necesita parcurgerea, in medie, a jumătate din lista)
- gasirea initiala** => $O(1)$
- cautarea in lista** => $O(M)$
- coeficient >=2/3** => performanta redusa

-**pentru gasirea unui element** => parcurgerea listei cu indicele corespunzator

- inserare liste nesortate $1+L$
- inserare liste sortate $1+L/2$

*Functii de dispersie

- trebuie sa fie **simpla**
- functie lenta** => performanata redusa

*Eficiența tabelelor de dispersie

- inserare si cautare(minm, fara coliziuni) => $O(1)$
- timp de inserare si cautare => d.p. lungimea sondajului efectuat +timpul calculului functiei

*Adresare deschisa(tartare coliziuni)

- pierdere de eficeinta
- cauatriile nereusite dureaza mai mult
- trebuie sa parcurga toata secventa
- gasit => se opreste
- parcurge fara garantia ca va gasi ceva

-**L optim=0.5**

*Sondaj linear

- P**=lungimea sondajului
- L**=coeficientul de incarcare

sondajului liniar, pentru o operatie reusita, este:

$$P = (1 + 1 / (1 - L)) / 2$$

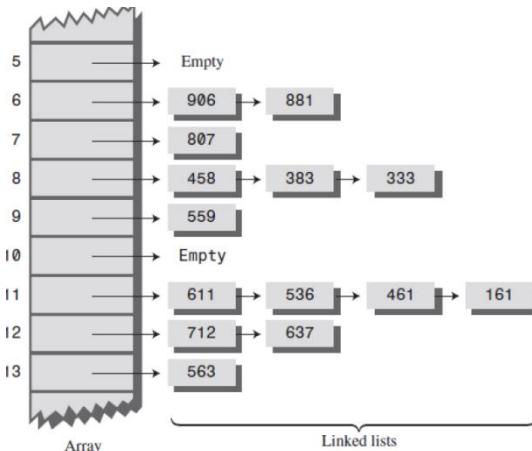
Pentru o operatie nereusita, avem:

$$P = (1 + 1 / (1 - L)^2) / 2$$

Dacă funcția secundară de dispersie întoarce valoarea s, secvența de sondaj este:

$$x, x+s, x+2s, x+3s, x+4s, \dots$$

unde valoarea s depinde de cheie, dar rămâne constantă pe parcursul sondajului



-L nu trebuie sa depaseasca 2/3, daca este posibil ½

-L mai mic => memorie mai multa ocupata cf

-valoare optima=compromise intre eficienta memoriei si viteza

*Sondaj patratice si dubla dispersie

-performante descrise prin ecuatii commune

-putin mai bun ca sondajul linear

-permite valori mai mari pentru L, fara sa scada performanta

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define H 13           //dimensiune tabela hash

typedef struct nod {    //nod din lista de cuvinte
    char *cuv;          //adresa unui cuvant
    int nr;             //numar de aparitii cuvant
    struct nod *next;   //legatura la nodul urmator
} Nod;

typedef Nod* Map[H]; //tipul dictionar
//functie de dispersie
int hash (char *s) {
    int i, sum = 0;
    for (i = 0; i < strlen(s); i++)
        sum = sum + (i + 1)*s[i];
    return sum % H;
}

//initializare tabela de dispersie
//se initializeaza cu NULL fiecare element din tabela
void initD (Map d) {
    int i;
    for (i = 0; i < H; i++)
        d[i] = NULL;
}

//afisare dictionar (lista dupa lista)
//pentru fiecare cuvant din text se afiseaza numarul de
//aparitii al cuvantului din text
void printD (Map d) {
    int i;
    Nod *p;
    for (i = 0; i < H; i++) {
        p = d[i];
        while (p != NULL) {
            printf("%20s %4d\n", p->cuv, p->nr);
            p = p->next;
        }
    }
}

//cautare (localizare) cuvant in dictionar
//pozitia cheii c in tabela de dispersie este aflata folosind
//functia de dispersie
Nod* cautD (Map d, char *c) {
    Nod *p;
    int k;
    k = hash(c);      //pozitia cheie c in vector
    p = d[k];         //adresa listei de cuvinte
    while (p != NULL && strcmp(p->cuv, c)) //cauta cheia c in lista
        p = p->next;
    return p;          //p=NULL daca c nu este gasit
}

//adauga o pereche cheie-valoare in dictionar
//pozitia cheii c in tabela de dispersie este aflata folosind
//functia de dispersie
void putD (Map d, char *c, int nr) {
    Nod *p, *pn;
    int k;
    k = hash(c);
    if ((p = cautD(d,c)) != NULL)
        //daca cheia c exista in nodul p
        p->nr = nr;
    else { //daca cheia c nu exista in tabela de dispersie
        pn = (Nod*) malloc(sizeof(Nod));
        //creare nod nou
        pn->cuv = c;    //completare nod cu cheia c
        pn->nr = nr;    //completare nod cu valoarea nr
        pn->next = d[k];
        d[k] = pn; //adaugare la inceputul listei de cuvinte
    }
}

//extrage valoarea asociata unei chei date
int getD (Map d, char *c) {
    Nod *p;
    if ((p = cautD(d,c)) != NULL)
        return p->nr; //daca exista anterior
    else
        return 0;
}

//citire cuvinte, creare si afisare lista de cuvinte
int main() {
    char numef[20], buf[128], *q;
    Map dc;
    int nra;
    FILE *f;
    printf("Introduceti numele fisierului: ");
    scanf("%s", numef);
    f = fopen(numef, "r");
    initD(dc);
    while (fscanf(f, "%s", buf) > 0) {
        q = strdup(buf); //creare adresa pentru sirul citit
        nra = getD(dc, q); //calculeaza nr de aparitii cuvant
        if (nra == 0) //daca e prima aparitie
            putD(dc, q, 1); //pune cuvant in tabela
        else
            putD(dc, q, nra + 1); //modifica numar aparitii cuvant
    }
    printD(dc); //afisare tabela de dispersie
}
```

În cazul unei căutări reușite, avem:

$$P = -\log_2(1 - L) / L$$

În cazul unei căutări nereușite, avem:

$$P = 1 / (1 - L)$$

Algoritm Ford-Fulkerson

-alg greedy care calculeaza fluxul maxim intr-o retea de transport

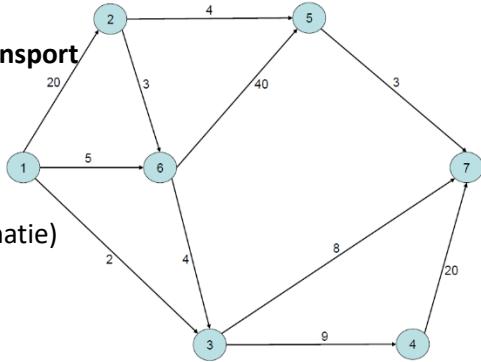
-reteea de transport=graf orientat:

->exista doar un nod cu grad interior 0 (nod sursa)

->exista doar un nod cu grad exterior 0 (nod destinatie)

->**graf conex**, cu drum de la **sursa->destinatie**

-> fiecare **arc are o capacitate**



-**flux**=functie care indeplineste:

->**conditia de marginire a fluxului**=pt orice arc valoarea nu poate depasi capacitatea lui

->**conditia de conservare a fluxului**=suma fluxurilor arcelor care intra in x=suma fluxurilor care ies din x (exceptie sursa si destinatia)

-**idee**:

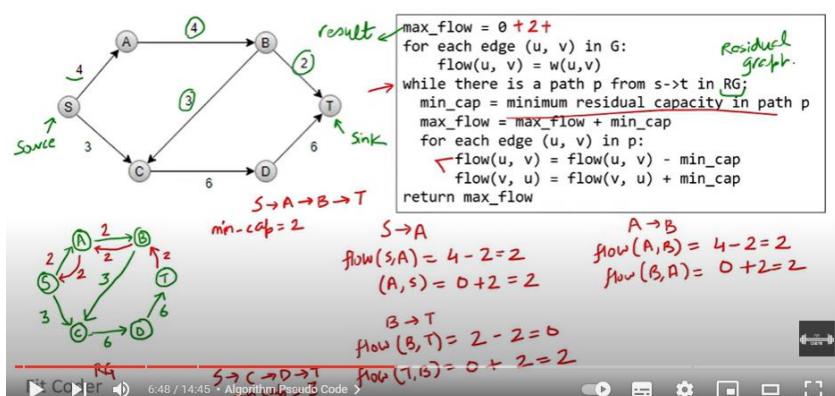
1.se determina un drum de ameliorare a fluxului

2.se amelioreaza fluxul de-a lungul acestui drum

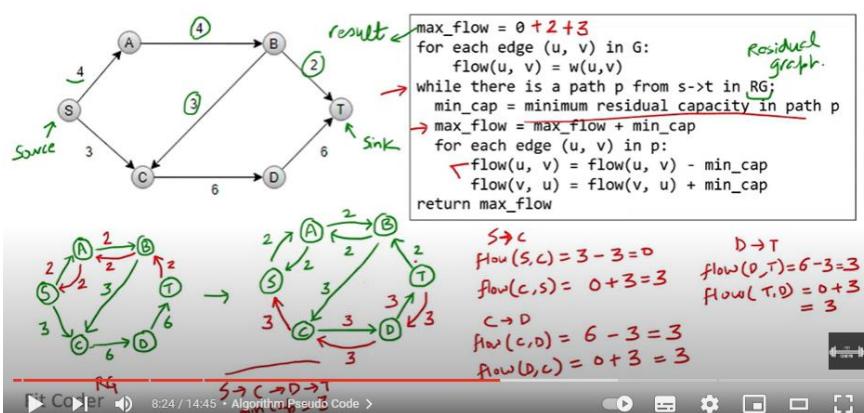
-**drum de ameliorare**= determinat prin parcurgerea in latime a grafului incepand cu sursa

-complexitate= $O(n^*m^2)$

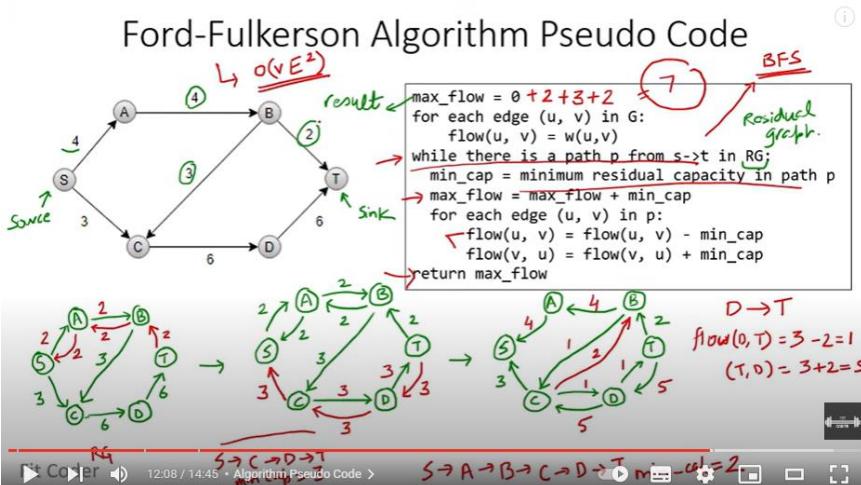
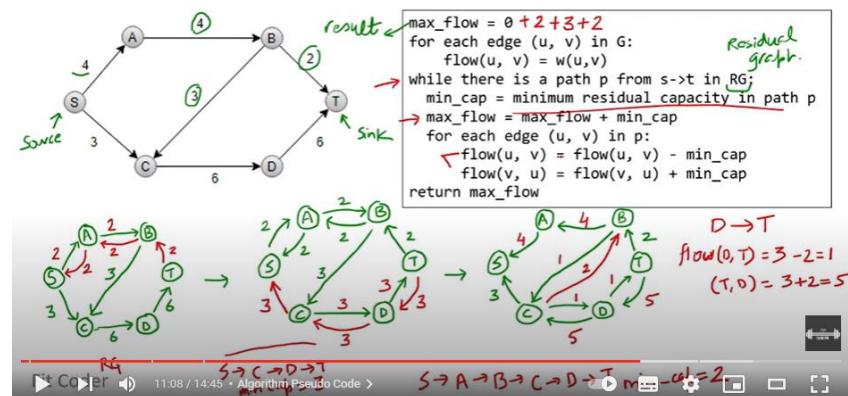
Ford-Fulkerson Algorithm Pseudo Code



Ford-Fulkerson Algorithm Pseudo Code



Ford-Fulkerson Algorithm Pseudo Code



Algoritmul Lempel-Ziv-Welch

<https://www.youtube.com/watch?v=RV5aUr8sZD0&t=3s>

-cea mai folosita metoda de compresie a datelor

-tabela de dispersie->asociaza unor siruri de

caracter->coduri numerice

-marime uzuala= $4096 = 2^{12}$ ----- $256 = 2^8$ pozitii contin

caractere individuale

Compresia LZW

Initializare dicționar cu toate caracterele din sir și codurile asociate
 $P = \{\}$; $C =$ caracterul curent din sir
căt timp C nu este ultimul caracter **repetă**
 citește un caracter C
 dacă $P+C$ există în dicționar **atunci** $P=P+C$
 altfel /* $P+C$ nu este în dicționar */
 determină codul lui P din dicționar și
 depune-l în sirul de ieșire
 adaugă $P+C$ în dicționar
 $P=C$
 C trece la caracterul următor
 Depune în sirul de ieșire codul asociat lui P
 Sirul de ieșire este codificarea dorită

Tabela de dispersie va conține:

$0=a / 1=b / 2=0b (ab) / 3=1b (bb) / 4=1a (ba) /$

$5=0a (aa) / 6=2b (abb) / 7=4a (baa)$

$8=2a (aba) / 9=6a (abba) / 10=5a (aaa) /$

$11=5b (aab) / 12=7b (baab) / 13=3a (bba)$

Intrare: $w \ a \ b \ b \ a \ w \ a \ b \ b \ a$

1 a

2 b

3 w

$P = \{\} \ C = w \ P+C = w$

$P = w \ C = a \ P+C = wa$

adaug 4 wa output 3

$P = a \ C = b \ P+C = ab$

adaug 5 ab output 1

$P = b \ C = b \ P+C = bb$

adaug 6 bb output 2

$P = b \ C = a \ P+C = ba$

adaug 7 ba output 2

$P = a \ C = w \ P+C = aw$

adaug 8 aw output 1

$P = w \ C = a \ P+C = wa$

adaug 9 wab output 4

$P = b \ C = b \ P+C = wb$

adaug 10 bba output 6

$P = a$

output 1

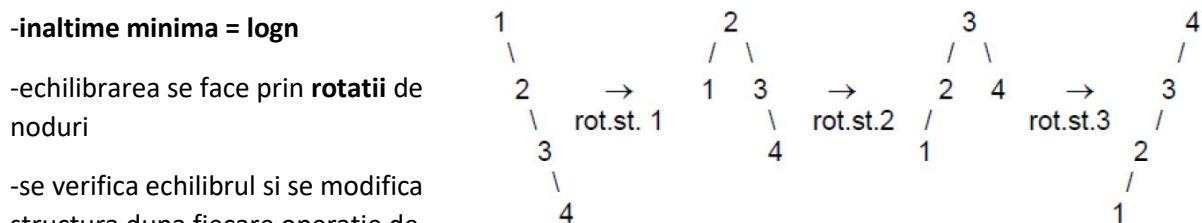
Ieșire: 3 1 2 2 1 4 6 1

Arbore echilibrati

-**arboare binary echilibrat** => cautare eficientă

-**arbore echilibrat** => inaltime minima

-inaltime minima = logn



-se verifica echilibrul si se modifica structura dupa fiecare operatie de inserare sau stergere

-Criterii: ->**deterministe**

->probabiliste

*Criterii deterministe

-au efectul de a reduce sau menține înaltimea arborelui:

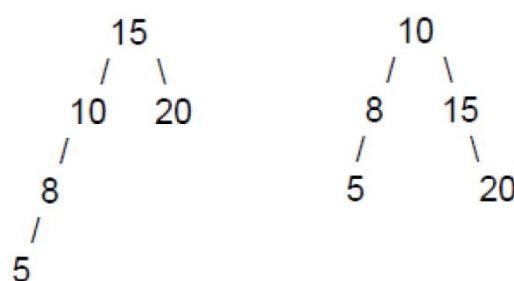
- 1.diferenta dintre inaltimele celor doi subarbori ai fiecarui nod (arborii AVL)
2.diferenta dintre cea mai lunga si cea mai scurta cale de la radacina la Frunze (arborii Red-Black)

Structura de date se retine : **inaltimea nodului(1)** si **culoarea lui(2)**

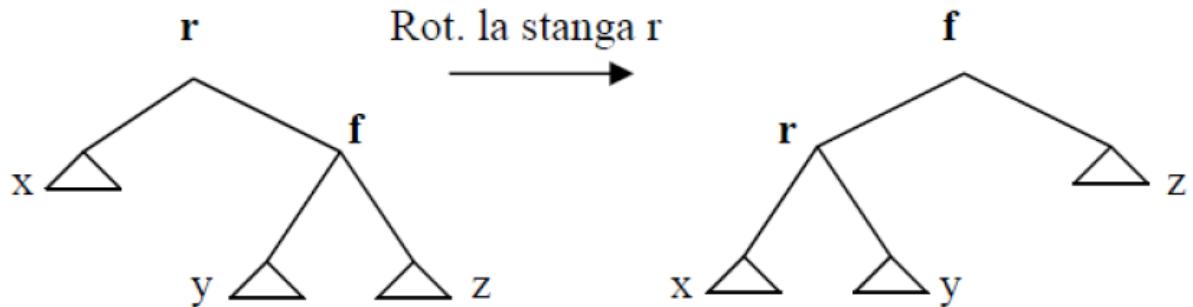
*Criterii probabiliste
-pornesc de la observarea efectului unor modificari asupra reducerii inaltimii arborilor binary de

- memoreaza in fiecare nod inaltimea si numarul de noduri din subarborele a carui radacina este
 - restructurarea se face doar dupa un nr de inserari sau stergeri
 - stergerea doar marcheaza nodul ca fiind invalid (nu e efectiva)
 - stergerea efectiva se face doar cand mai mult de jumata sunt marcate ca invalide
 - la fiecare inserare se actualizeaza informatiile pentru celelalte noduri => apoi se verifica echilibrul
=>se restructureaza doar subarborele care a produs dezechilibru

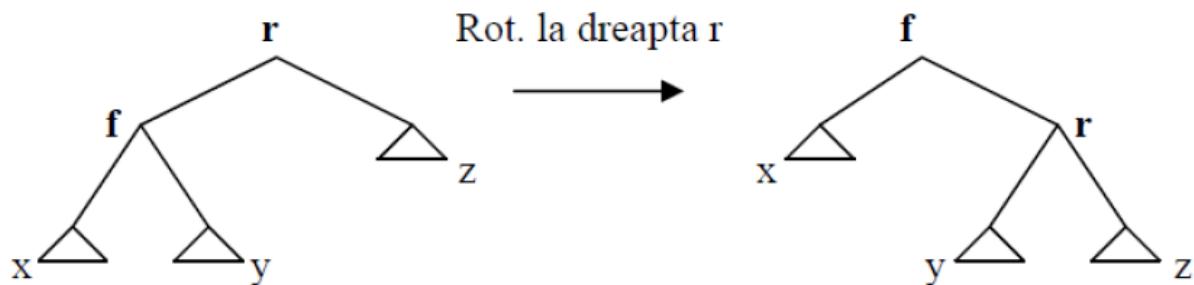
- Initial sunt doar 15-10-20
 - Se adauga 8, totul este ok
 - Se adauga 5, echilibrul e stricat =>refacem arborele----->



- dupa fiecare rotatie, relatiile dintre valorile nodurilor sunt mentinute
- rotatie la stanga**



- rotatie la dreapta**



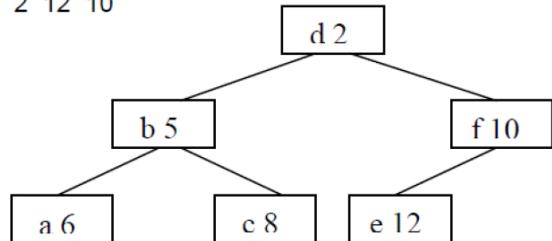
Arborei Splay si Treap

Cheie	a	b	c	d	e	f
Prior	6	5	8	2	12	10

-sta la baza memorie cache

-nodul introdus este ridicat mereu in radacina (Splay) sau pana cand este indeplinita o conditie(Treap)

-**Arborei Splay**=aduce ultima valoare accesata in radacina, prin rotatii, dupa cautarea sau adaugarea unui nou nod



->**pentru stergere**, se adduce in radacina nodul respectiv, apoi este sters

- >**metode:**
 - ridicarea treptata a nodului n, prin rotatii, pana cand ajunge radacina
 - ridicarea parintelui sau, apoi ridicarea lui**(echilibrare mai eficienta)

-**Arborei Treap**=fiecare nod are o prioritate, arborele este obligat sa **respecte valorile nodurilor** dar si **conditia de heap fata de prioritatile nodurilor** (pentru **movila Min-Heap**)

->**nu e o movila**

->**inaltimea** nu depaseste $2 \cdot \log(n)$

->folosesc rotatii simple si analogii cu structura de movila

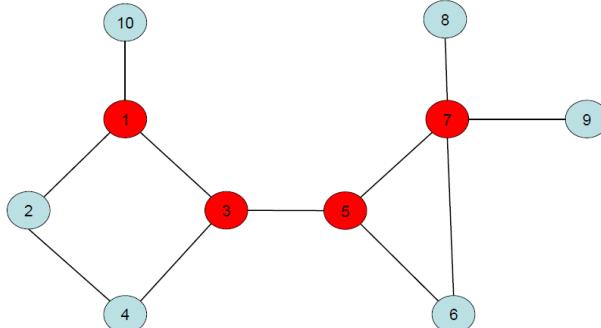
->**Treap=Tree Heap** (structura+caracteristicile arborelui binary+cele ale movilei)

->**fiecare nod = valoare+prioritate**

Biconexitate

-**punct de articulație**=daca subgraful obtinut prin eliminarea acestui varf si a muchiilor incidente cu acesta nu mai este conex

- Punctele de articulație sunt 1, 3, 5, 7



-**componenta biconexă a unui graf**=subgraf biconex maximal cu aceasta proprietate

-**descompunere in componente biconexe** <=DFS

-se pleaca din 3

-**muchii negre**=muchii obtinute prin parcurgere DFS

-**muchii rosii**=muchii de revenire

-**punct de articulație**= are cel putin 2 descendenți intre subarbori diferiti ai radacinii, neexistand muchii

-**nu e punct de articulație**=daca din descendantul y al lui x, poate fi atins un stramos al lui x pe un drum format din descendenți ai lui x si o muchie de revenire

-numar de ordine al vf x= dfn(x)

-x stramosul lui y => dfn(x)<dfn(y)

-low(x)

$low(x) = \min\{dfn(x), \min\{low(y) | y \text{ fiu al lui } x, \min\{dfn(y) | (x,y) \text{ muchie de revenire}\}\}$

Se folosesc 3 vectori de noduri:

1. $dfn[x]$ este momentul vizitării (descoperirii) vârfului x în explorarea DFS

2. $p[x]$ este predecesorul vârfului x în arborele de explorare DFS

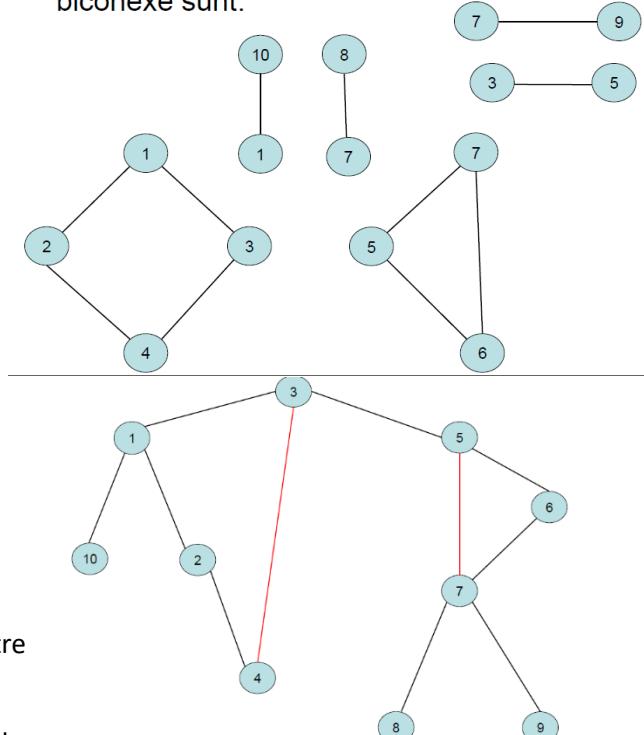
3. $low[x] = \min\{dfn[x], \min\{low[y] | y \text{ fiu al lui } x, \min\{dfn[y] | (x,y) \text{ muchie de revenire}\}\}$

Vectorul low se determină la vizitarea DFS

Funcția ce determină **punctele de articulație** verifică, pe rând, pentru fiecare vârf din graf, ce statut are în arborele DFS

-**biconex**=nu are puncte de articulație

- Pentru graful precedent, componentele biconexe sunt:



x	1	2	3	4	5	6	7	8	9	10
dfn(x)	2	4	1	5	6	7	8	9	10	3
low(x)	1	1	1	1	6	6	6	9	10	3

Funcție care numără fii lui x în arborele descris prin vectorul de predecesori p

```

int fii (int x, int p[], int n) {
    int i, m = 0;
    for (i = 1; i <= n; i++)
        if (i != x && p[i] == x) // dacă i are ca părinte pe x
            m++;
    return m;
}

//Funcție de parcurgere în adâncime din vârful x, cu
//crearea vectorilor dfn, p, low
void dfs (Graf g, int x, int t, int dfn[], int p[], int low[]) {
    int w;
    low[x] = dfn[x] = ++t;
    for (w = 1; w <= g.n; w++) {
        if (g.a[x][w]) // dacă w este vecin cu x
            if (dfn[w] == 0) { // dacă w nevizitat
                p[w] = x; // w are ca predecesor pe x
                dfs(g,w,t,dfn,p,low); //continuă vizitarea din w
                low[w]=min(low[x],low[w]); //actualizare low[w]
            }
        else // dacă w deja vizitat
            if (w != p[x]) // dacă muchie de revenire (x,w)
                low[x]=min(low[x],dfn[w]); // actualizare low[x]
    }
}

```

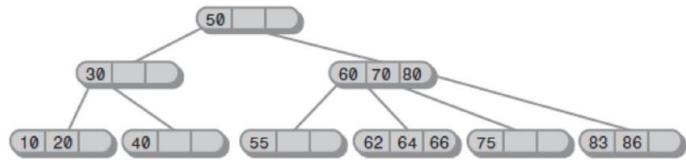
```

//Funcție de găsire a punctelor de articulație
void articulatie (Graf g, int dfn[], int p[], int low[]) {
    int x, w, t = 0; // t = moment vizitare (descoperire vârf)
    dfs(g, 1, t, dfn, p, low); // vizitare din 1 (graf conex)
    for (x = 1; x <= g.n; x++) {
        if (p[x] == 0) {
            if (fii(x, p, g.n) > 1) //dacă rădăcină cu cel puțin 2 fiu
                printf("%d ", x); //este punct de articulație
        }
        else // dacă nu e rădăcina
        for (w = 1; w <= g.n; w++) {
            // dacă x are un fiu w în arborele DFS
            if (p[w]==x && low[w]>=dfn[x]) // cu low[w]>=dfn[x]
                printf("%d ", x); // atunci x este punct de articulație
    } } }

```

Arborei 2-3-4

-**arbori multicai(de ordin al 4-lea)**, unde fiecare nod are cel mult 4 fiu



-**sunt arbori echilibrativ**

-fiecare **nod poate contine 1,2 sau 3 elemente**

-**toate frunzele se află pe același nivel**

-**2-3-4= numarul de legaturi catre fiu pe care le poate avea un nod**

-un nod **1 element=> 2 fii**

-un nod **2 elemnte => 3 fii**

-un nod **3 elemnte => 4 fii**

-un nod care nu este frunza=>cu un fiu mai mult decat numarul de elemente pe care le contine(**L=nr legaturi, D=nr elemente => L=D+1**) –frunzele se exclud de la regula (nu au fiu)

-**arbore binary ⇔ arbore multicai de ordin 2**

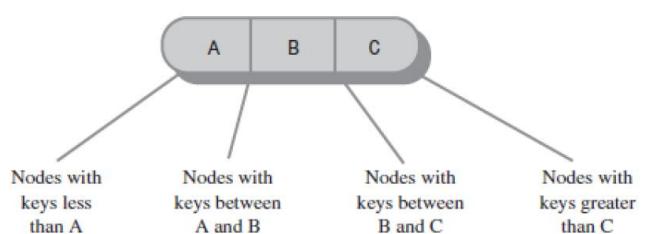
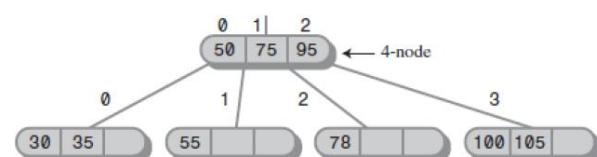
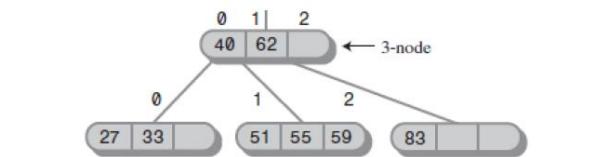
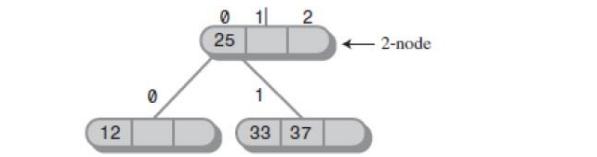
-**nu este permisa existenta nodurilor cu un fiu**

-**elementele din nod sunt dispuse in ordine crescatoare**

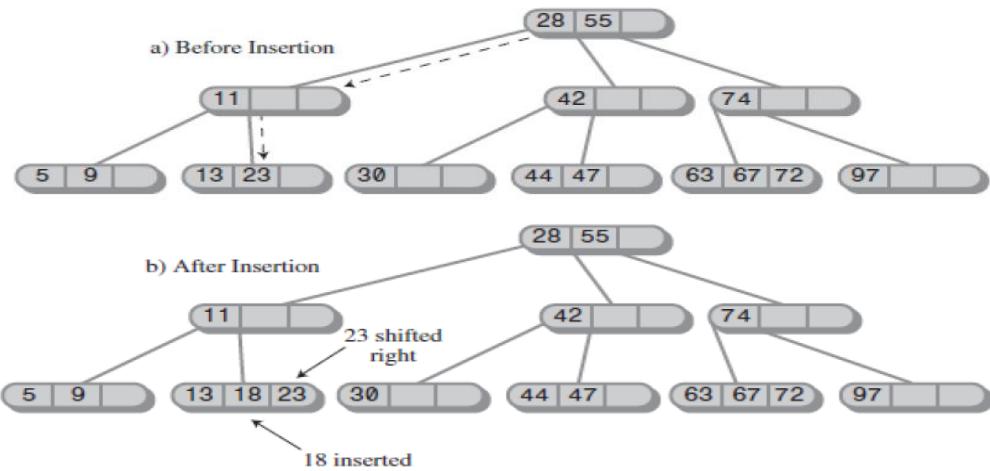
-**Reguli:**

-noile elemente sunt mereu inserate in frunze (se pot muta valori pentru a nu se schimba ordinea)

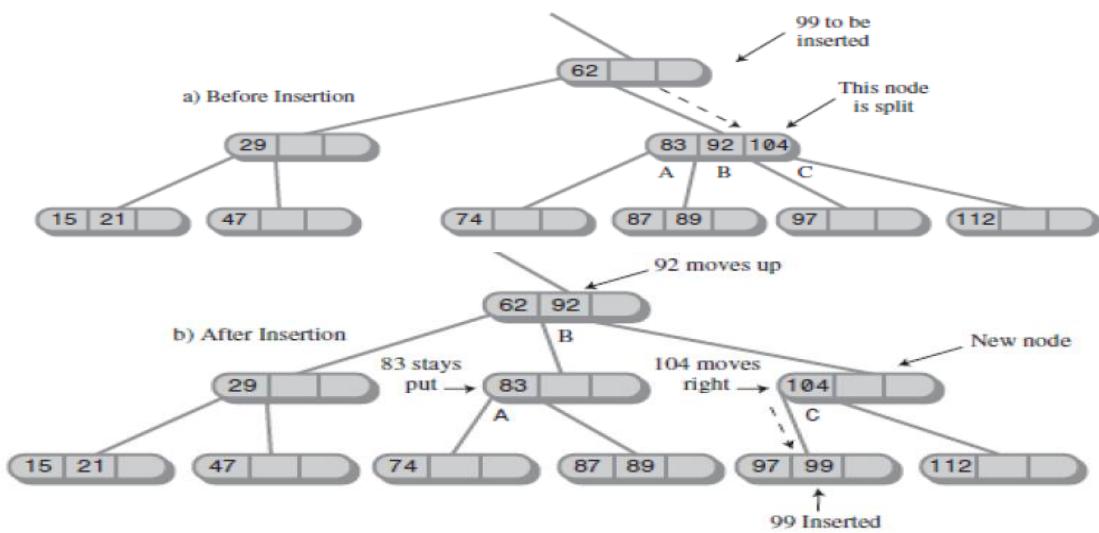
Un nod cu 2 legături se numește **2-nod**, cu 3 legături **3-nod**, iar cu 4 legături **4-nod**



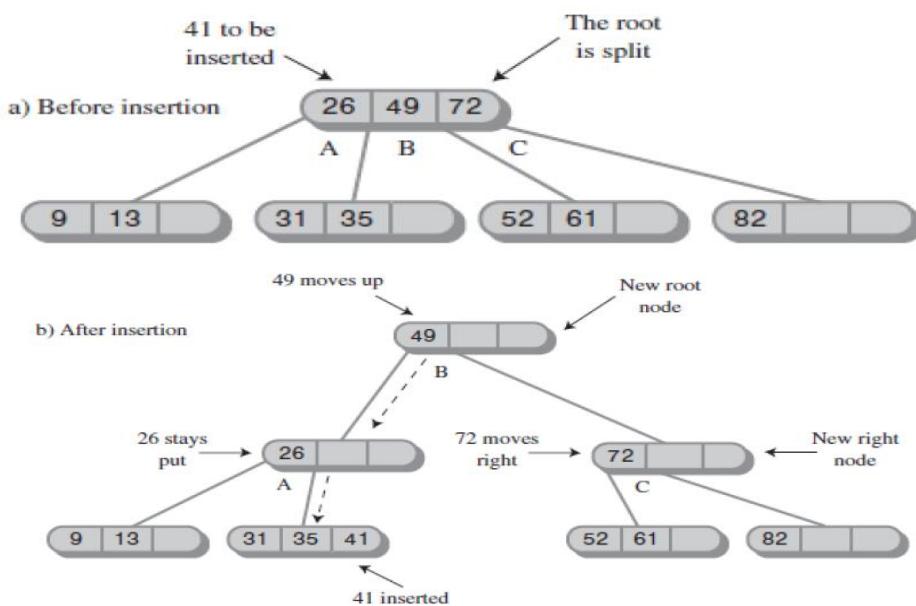
*Divizarea nodurilor



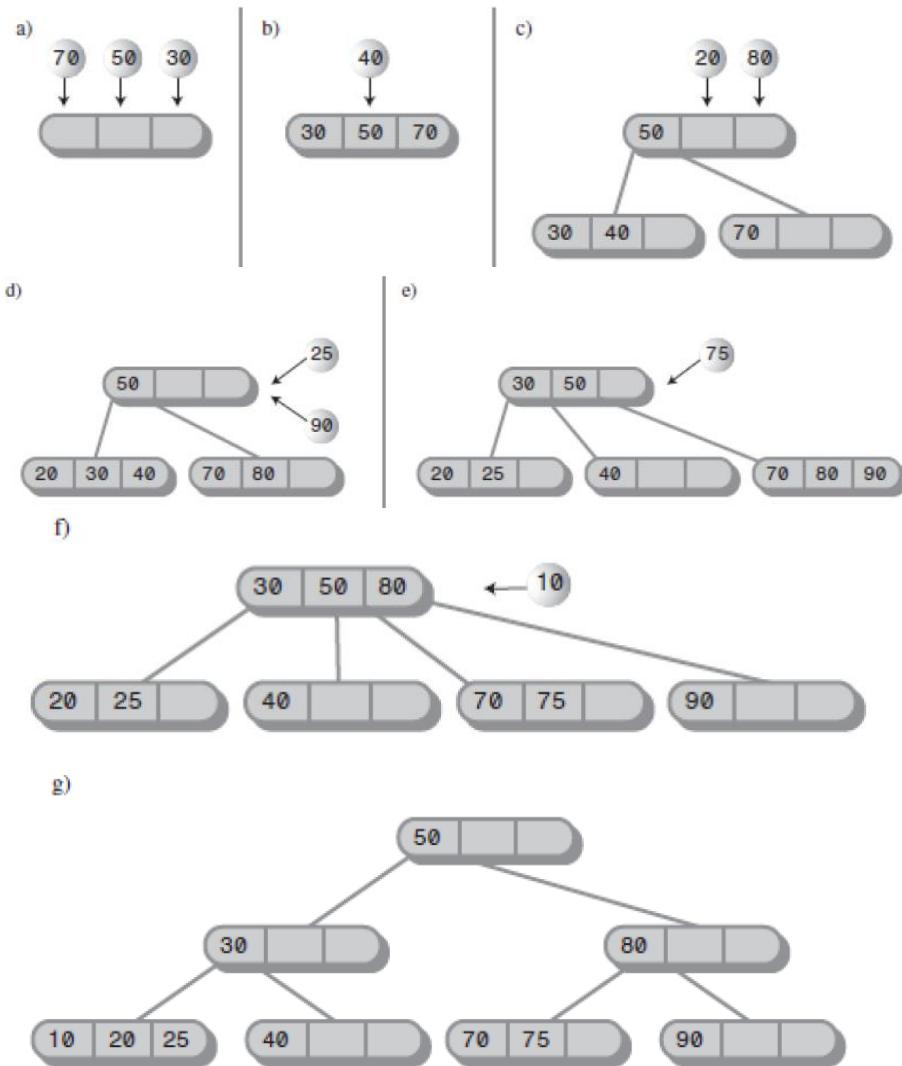
-la intalnirea unui nod complet la inserare



*Divizarea radacinii



*Divizarea la parcurgere descendenta



-inaltimea unui arbore poate creste oar in urma divizarii

Arbore AVL

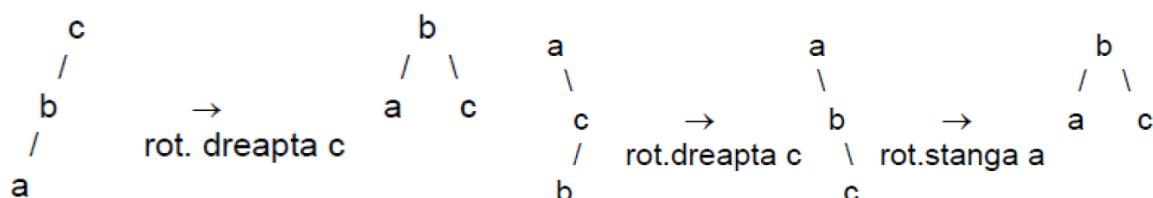
-arbore binary de cautare in care fiecare subarbore e echilibrat in inaltime

-se introduce fiecarui nod un camp suplimentar pt inaltimea nodului/diferenta dintre inalimi

a celor 2 arbori

-echilibrarea se face prin rotatii simple sau duble insotite de recalcularea inaltimei, parcurgand de jos in sus arborele

nod (-1, 0, 1 pentru noduri "echilibrate" și -2 sau +2 la producerea unui dezechilibru) factorul de echilibru al unui nod interior se poate modifica la -2 (adăugare la subarborele din stânga) sau la +2 (adăugare la subarborele din dreapta),



-Reguli: ----->

Inserarea în subarborele din dreapta al unui fiu drept necesită o **rotație simplă la stânga**

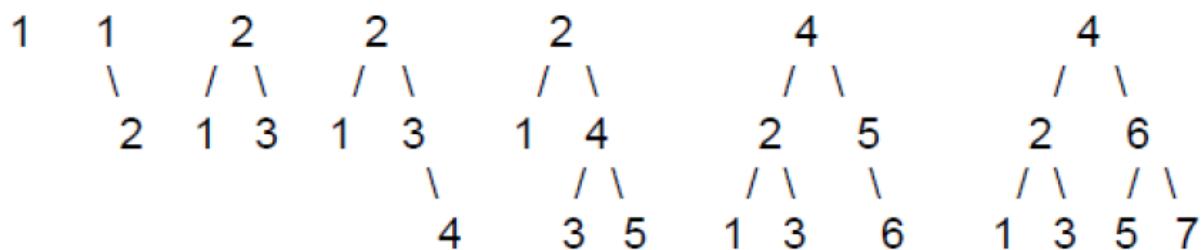
--

Inserarea în subarborele din stânga al unui fiu stâng necesită o **rotație simplă la dreapta**

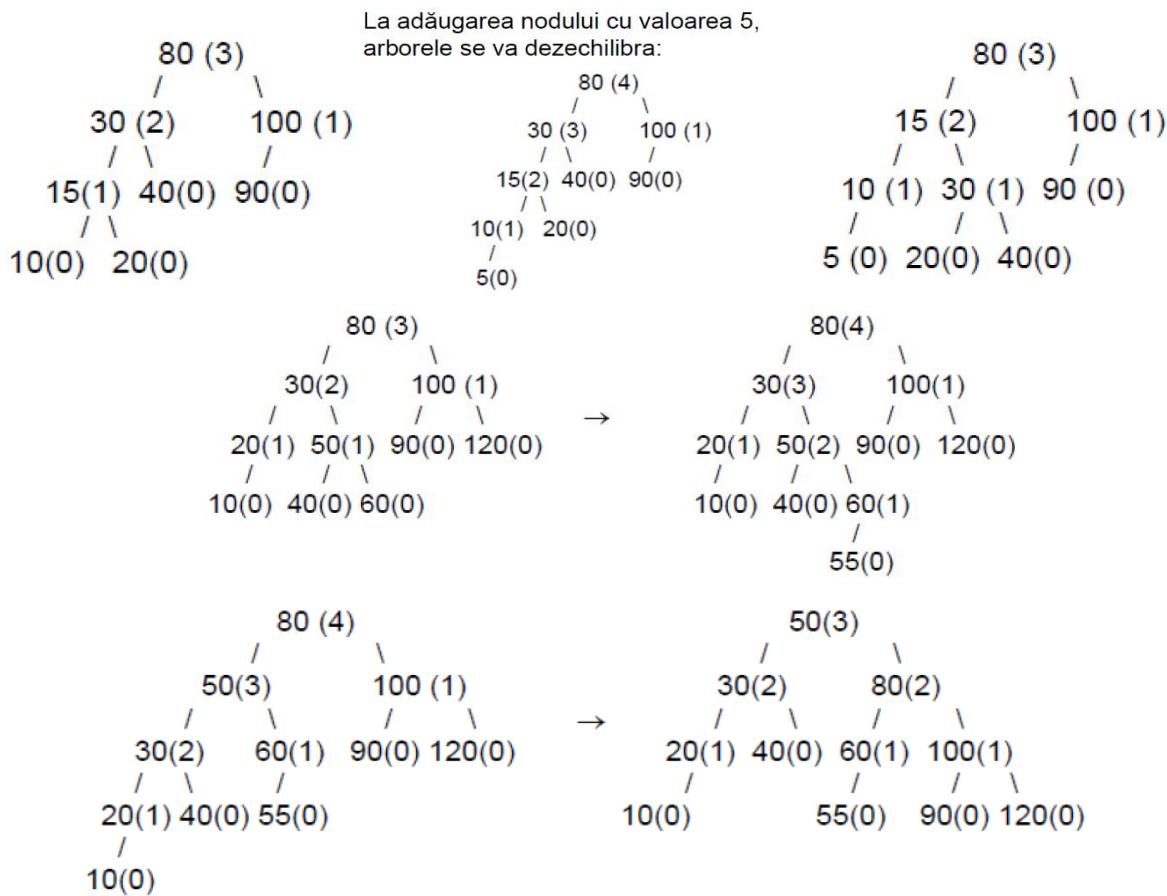
Inserarea în subarborele din stânga al unui fiu drept necesită o **rotație dublă la stânga**

Inserarea în subarborele din dreapta al unui fiu stâng necesită o **rotație dublă la dreapta**

- Se inserează valorile 1, 2, 3, 4, 5, 6, 7:



-se memoreaza in fiecare nod din arbore inaltimea sa (nod inexistent => -1)



Arbore de regasire (trie<-retrieval=regasire)

-arbori multicai

cana, cant, casa, dop, mic, minge

-memoreaza siruri de caractere sau siruri de biti care au in comun unele subsiruri

-nodurile pot contine sau nu date

-sir= cale radacina->nod frunza/nod interior

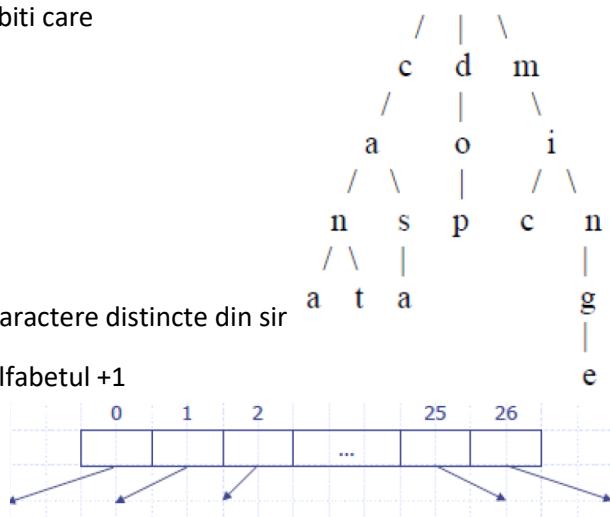
-siruri pe biti=>arbore binary

-siruri de caractere=>nr de succesi=nr de caractere distincte din sir

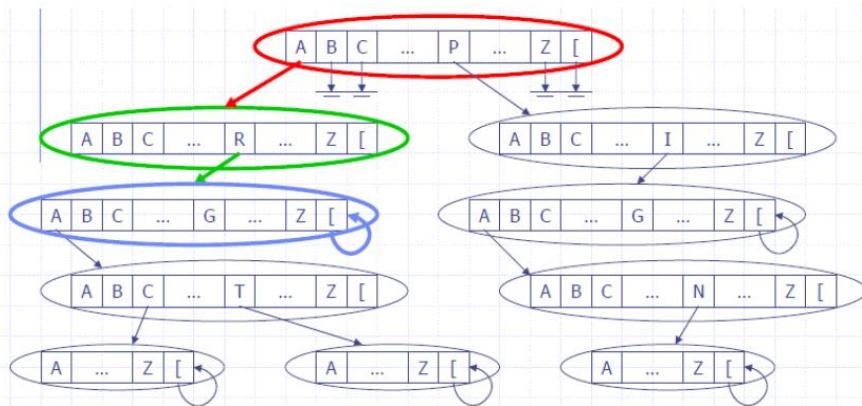
-fiecare nod are -> subarbori cate litere are alfabetul +1

- un nod al arborelui de regasire =>

-locul 26 este pentru "[", urm character din ascii dupa Z, el tine locul caracterului terminator



AR, ARAC, ARAT, PI și PIAN



-inserare si stergere cum am facut la lab

-ocupa mult spatiu in memorie (o buna parte din el fiind nefolosit)

- nu este o varianta buna daca sunt putine cuvinte

-cand ai un alphabet intreg nu mai este un dezavantaj asa mare

Coda (Queue)-algoritmi

```
typedef struct QNode {
    BTNode *data;
    struct QNode *next;
    struct QNode *prev;
} QNode_t;

typedef struct queue {
    QNode_t *head;
    QNode_t *tail;
    size_t len;
} queue_t;
```

```
QNode_t *initQueueNode(BTNode *data) {
    QNode_t *newNode = malloc(sizeof(QNode_t));
    newNode->data = data;
    newNode->prev = newNode->next = NULL;
    return newNode;
}

queue_t *initQueue() {
    queue_t *newQueue = malloc(sizeof(queue_t));
    newQueue->head = newQueue->tail = NULL;
    newQueue->len = 0;
    return newQueue;
}
```

```

void enqueue(queue_t *queue, BTNode *data) {
    if (data == NULL)
        return;
    QNode_t *new = initQueueNode(data);
    if (queue->len == 0) {           // Coada goala
        queue->head = new;
        queue->tail = new;
    } else {                         // Introducere la inceput de coada
        new->next = queue->head;
        queue->head->prev = new;
        queue->head = new;
    }
    ++queue->len;
}

BTNode *dequeue(queue_t *queue) {
    BTNode *ret;
    if (queue->len == 0)           // Coada este goala
        return NULL;
    QNode_t *aux = queue->tail;   // Elimina nod de la final
    if (queue->len == 1) {         // Coada are un singur element
        queue->tail = NULL;
        queue->head = NULL;
    } else {                      // Coada are mai multe elemente
        queue->tail = aux->prev;
        queue->tail->next = NULL;
    }
    --queue->len;

    ret = aux->data;           // Copiez valoarea nodului
    free(aux);                 // Setez statusul pe 0
    return ret;
}

BTNode *peek(queue_t *queue) {
    if ([queue->len == 0])
        return NULL;
    return queue->tail->data;
}

```

Stiva(Stack)-algoritmi

```
SNode_t *initStackNode(BTNode *value) {
    SNode_t *newNode = malloc(sizeof(SNode_t));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

stack_t *initStack() {
    stack_t *newStack = malloc(sizeof(stack_t));
    newStack->top = NULL;
    newStack->len = 0;
    return newStack;
}

void push(stack_t *stack, BTNode *data) {
    if (data == NULL)
        return;
    SNode_t *new_node = initStackNode(data);
    new_node->next = stack->top;
    stack->top = new_node;
    ++stack->len;
}

BTNode *pop(stack_t *stack) {
    SNode_t *aux = stack->top;
    BTNode *ret;
    if (stack->len == 0)          // Stiva e goala
        return NULL;
    ret = aux->data;             // Salvez valoarea nodului
    stack->top = aux->next;      // Elimin nodul din stiva
    --stack->len;
    free(aux);
    return ret;
}

BTNode *top(stack_t *stack) {
    if ([stack->len == 0])
        return NULL;
    return stack->top->data;
}
```

```
typedef struct SNode {
    BTNode *data;
    struct SNode *next;
} SNode_t;

typedef struct stack {
    SNode_t *top;
    SNode_t *len;
} stack_t;
```