

# Structuri de Date și Algoritmi

## Recapitulare

**Mihai Nan**

Departamentul de Calculatoare  
Facultatea de Automatică și Calculatoare  
Universitatea POLITEHNICA din București



**Anul Universitar 2022–2023**

# Problema 1

Se dă o tablă 2D ( $m \times n$ ) care conține 'X' și 'O'. Se cere să se implementeze o funcție prin care să cucerim o zonă delimitată de 'X' pornind dintr-un punct dat. O zonă e cucerită de 'X' atunci când toate 'O'-urile sunt transformate în 'C' în zona delimitată de 'X'.

*Input:*

X	X	X	X
X	O	O	X
X	X	O	X
X	O	X	X

**start:** (2,2)

*Output:*

X	X	X	X
X	C	C	X
X	X	C	X
X	O	X	X

# Problema 1

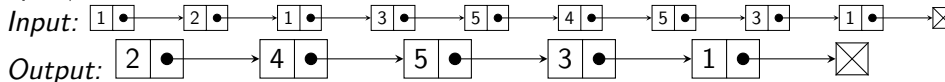
```
int checkCoordinates(int x, int y, int n, int m) {  
    if (x < 0 || y < 0)  
        return 0;  
    if (x >= n || y >= m)  
        return 0;  
    return 1;  
}  
  
int checkCell(char **grid, int n, int m, int x, int y) {  
    if (!checkCoordinates(x, y, n, m))  
        return 0;  
    return grid[x][y] == '0';  
}
```

# Problema 1

```
void captures(char **grid, int n, int m, int x, int y) {  
    if (grid[x][y] == 'O') {  
        grid[x][y] = 'C';  
        if (checkCell(grid, n, m, x - 1, y))  
            captures(grid, n, m, x - 1, y);  
        if (checkCell(grid, n, m, x, y - 1))  
            captures(grid, n, m, x, y - 1);  
        if (checkCell(grid, n, m, x + 1, y))  
            captures(grid, n, m, x + 1, y);  
        if (checkCell(grid, n, m, x, y + 1))  
            captures(grid, n, m, x, y + 1);  
    }  
}
```

## Problema 2

Scrieți o funcție **recursivă** care să elimine dintr-o listă liniară simplu înlănțuită elementele duplicate. Pentru elementele duplicate, se va păstra ultima apariție din listă.



# Problema 2

```
int contains(List list, int value) {  
    List tmp;  
    tmp = list;  
    while (tmp != NULL) {  
        if (tmp->value == value)  
            return 1;  
        tmp = tmp->next;  
    }  
    return 0;  
}
```

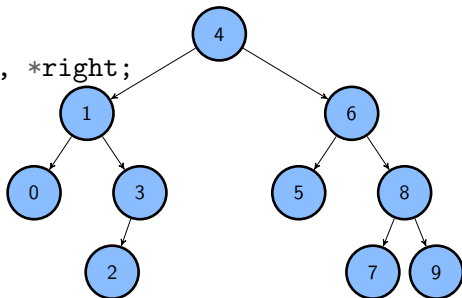
## Problema 2

```
List convert(List l, List tmp) {  
    if (l == NULL)  
        return tmp;  
    if (!contains(l->next, l->value))  
        tmp = insertLast(tmp, l->value);  
    return convert(l->next, tmp);  
}
```

# Problema 3

Se dă un arbore binar de căutare. Să se implementeze o funcție ce primește ca parametru un nod din arbore și returnează următorul nod considerând parcurgerea în inordine. Structura arboreului este definită astfel:

```
typedef struct tree{  
    int key;  
    struct tree *parent, *left, *right;  
} tree_t;
```



*Input: 3 Output: 4*

*Input: 1 Output: 2*

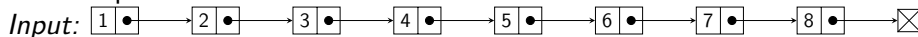


# Problema 3

```
int getSuccessor(Tree node) {
    Tree tmp;
    int pred = 0;
    if (node->right != NULL) {
        tmp = node->right;
        while (tmp->left != NULL) {
            tmp = tmp->left;
        }
        return tmp->value;
    } else {
        tmp = node->parent;
        if (tmp != NULL) {
            pred = tmp->value;
            while (tmp->parent != NULL && pred < tmp->parent->value) {
                tmp = tmp->parent;
            }
            return tmp->value;
        }
        return -1;
    }
}
```

# Problema 4

Implementați o funcție care primește ca argumente o listă liniară simplu înlănțuită cu elemente de tip numere întregi și o valoare numerică  $K$ . Această funcție va determina nodul de pe poziția  $K$  începând numerotarea de la ultimul nod spre primul și considerând ultimul ca fiind nodul ce trebuie returnat pentru  $K = 1$ .



și  $K = 3$

*Output:* 6

## Problema 4

```
List getNth(List head, int k) {  
    List iter1, iter2;  
    iter1 = head;  
    iter2 = head;  
    int i;  
    for (i = 1; i <= k; i++) {  
        iter1 = iter1->next;  
    }  
    while (iter1 != NULL) {  
        iter1 = iter1->next;  
        iter2 = iter2->next;  
    }  
    return iter2;  
}
```

# Problema 5

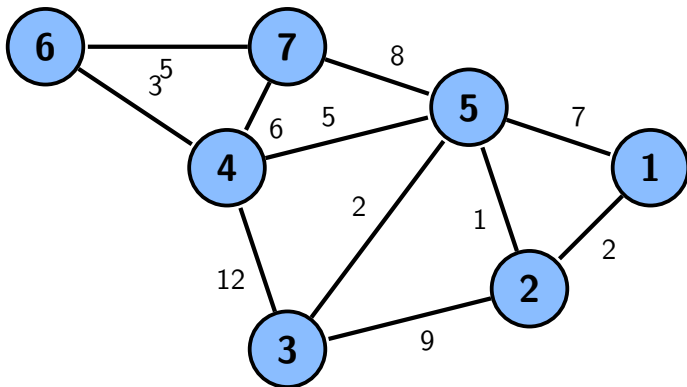
Implementați o funcție care primește ca parametru un graf neorientat reprezentat prin liste de adiacență și verifică dacă acel graf este sau nu un arbore. Fie  $G = (V, E)$  un graf neorientat, unde  $V$  este mulțimea nodurilor și  $E$  este cea a muchiilor. Graful  $G$  este arbore dacă și numai dacă:

- 1  $G$  este **minimal conex** (dacă i se elimină orice muchie, se obține un graf neconex);
- 2  $G$  este **maximal fără cicluri** (dacă i se adaugă orice muchie, se obține un graf care are măcar un ciclu).

# Problema 6

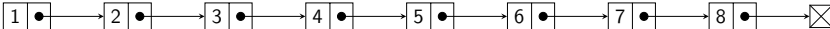
Considerăm că avem o hartă ce este reprezentată sub forma grafului din figura de mai jos. Fiecare nod din graf corespunde unui oraș, iar fiecare muchie corespunde unui drum bidirecțional ce are asociat un cost ce reprezintă timpul de care are nevoie o mașină autonomă pentru a învăța să parcurgă drumul. Problema care apare este că durează foarte mult ca mașina să fie învățată să parcurgă toate drumurile pe care le avem la dispoziție. De aceea, este nevoie să folosim unul dintre algoritmii studiați în cadrul cursului pentru a proiecta o mașină care să fie capabilă să ajungă în orice oraș (indiferent de orașul din care pornește), iar timpul necesar învățării acestei mașini să fie minim. Specificați ce algoritm ar putea să fie folosit pentru a determina timpul necesar învățării și drumurile pe care mașina ajunge să le învețe. Aplicați acest algoritm, pas cu pas, pentru graful dat.

# Problema 6



# Problema 7

Implementați o funcție care primește ca argumente o listă liniară simplu înlănțuită cu elemente de tip numere întregi și o valoare numerică  $K$ . Această funcție va determina nodul de pe poziția  $K$  în parcurgerea în Zig-Zag a listei.

*Input:*   
și  $K = 3$

*Parcurgerea în Zig-Zag:*  $1 \rightarrow 8 \leftarrow 2 \rightarrow 7 \leftarrow 3 \rightarrow 6 \leftarrow 4 \rightarrow 5$

*Output:* 2

# Problema 7

```
List zig_zag(List head, int k) {
    List first = head;
    List last = head;
    while (last->next != NULL)
        last = last->next;
    for (int i = 0; i < k; i++) {
        if (i % 2 == 0)
            head = head->next;
        else {
            last->next = NULL;
            last = head;
            while (last->next != NULL)
                last = last->next;
        }
    }
    if (k % 2 == 0) {
        return last;
    } else {
        return head;
    }
}
```



# Problema 8

Implementați o funcție care primește ca parametru un graf orientat aciclic reprezentat prin liste de adiacență și determină o sortare topologică validă pentru acest graf.

# Problema 8

Implementați o funcție care primește ca parametru un graf orientat aciclic reprezentat prin liste de adiacență și determină o sortare topologică validă pentru acest graf.

Pot aplica un DFS modificat (sortare descrescătoare după timpul de finalizare) sau algoritmul lui Kahn.

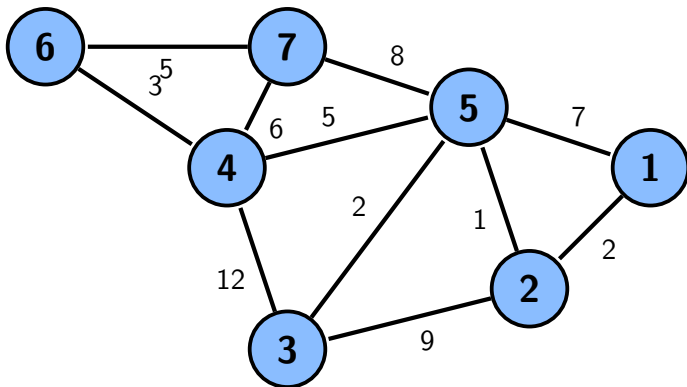
# Problema 8

```
void TopSortDFS(Graph g, int start, int *visited, Stack sort)
{
    visited[start] = 1;
    List list = g->adjList[start];
    while (list != NULL) {
        if (!visited[list->value])
            TopSortDFS(g, list->value, visited, sort);
        list = list->next;
    }
    sort = push(sort, start);
}
```

# Problema 9

Considerăm că avem o hartă ce este reprezentată sub forma grafului din figura de mai jos. Fiecare nod din graf corespunde unui stand, iar fiecare muchie corespunde unui drum bidirecțional ce are asociat un cost ce reprezintă cantitatea de energie ce trebuie consumată de către roboțelul Pepper pentru a se muta de la un stand la altul. Știind că roboțelul Pepper este situat inițial la standul 1, determinați cantitatea minimă de energie pe care trebuie să o consume pentru a ajunge la cel mai îndepărtat stand de el. Specificați ce algoritm ar putea să fie folosit pentru a determina cantitatea de energie și drumurile pe care roboțelul ajunge să le parcurgă. Aplicați acest algoritm, pas cu pas, pentru graful dat.

# Problema 9



*Vă mulțumesc pentru atenție!*

