INFORMATICĂ · CLASA A XI-A · STRUCTURI DE DATE (INTENSIV) · LECȚIA 17

Structuri de date liniare: stiva, coada și algoritmul lui Lee

Durată: 60 min Nivel: intensiv informatică / olimpiadă (apare și la neintensiv, indirect) Înainte: lecțiile x-02, xi-11

CE VEI ȘTI DUPĂ LECȚIE

- Implementezi o stivă (LIFO) și o coadă (FIFO) cu vectori și cu STL.
- Verifici parantezările corecte cu o stivă.
- Folosești coada pentru algoritmul lui Lee (drum minim în matrice).

1. Stiva: ultimul venit, primul plecat (LIFO)

Imaginează-ți un teanc de farfurii: pui o farfurie **deasupra** și iei tot de **deasupra**. Singurul element la care ai acces este cel din **vârf**.



Stiva

Operațiile de bază:

Operație	Efect	Cu vector (`st[]` , `vf`)	Cu STL
push x	adaugă x în vârf	`st[++vf] = x;`	`s.push(x)`
pop	scoate vârful	`vf--;`	`s.pop()`
top	citește vârful	`st[vf]`	`s.top()`
vidă?	stiva nu are elemente	`vf == 0`	`s.empty()`

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int st[100001], vf;                  // vf = câte elemente are stiva

int main()
{
    int q;
    cin >> q;
    while (q--)
    {
        char op[10];
        cin >> op;
        if (strcmp(op, "push") == 0)
        {
            int x;
            cin >> x;
            st[++vf] = x;            // întâi crește vf, apoi punem
        }
        else if (strcmp(op, "pop") == 0) vf--;
        else if (strcmp(op, "top") == 0) cout << st[vf] << '\n';
    }
    return 0;
}
```

DE CE AȘA?

De ce `++vf` la push și `vf--` la pop? `vf` este indicele ultimului element. La push facem loc (vf crește), apoi scriem; la pop nu trebuie să ștergem nimic: scăzând `vf`, elementul de deasupra „nu mai există" pentru program. **Ce se întâmplă la pop pe stivă goală?** `vf` devine −1 și programul citește din afara vectorului: verifică `vf > 0` înainte.

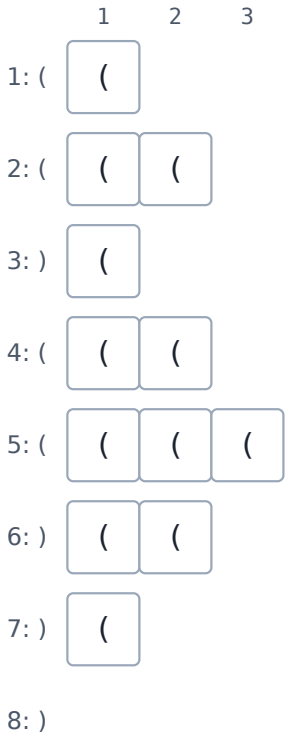
DE ȘTIUT

Stiva este chiar mecanismul din spatele recursivității: fiecare apel de funcție se pune pe stivă, iar la `return` se scoate (lecțiile XI.3 și XI.11).

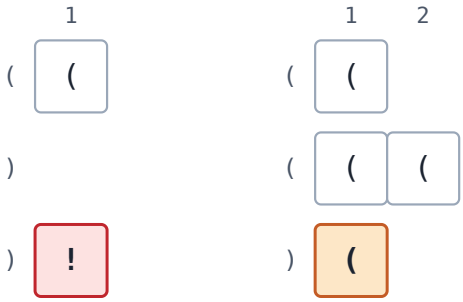
2. Verificarea parantezelor cu stiva

Un șir de paranteze este corect dacă fiecare paranteză care se închide corespunde ultimei deschise și nu rămâne niciuna deschisă. „Ultima deschisă" înseamnă exact **vârful stivei**.

Regula: la `(` → push; la `)` → dacă stiva e goală, șirul e greșit, altfel pop; la final stiva trebuie să fie goală.



se termină cu stiva goală și nu a existat niciun) pe stivă goală → șir corect




```cpp
#include <iostream>
using namespace std;

int main()
{
    char s[1005];
    cin >> s;
    int vf = 0;
    bool corect = true;
    for (int i = 0; s[i] && corect; i++)
        if (s[i] == '(') vf++;                // push (conținutul nu contează: doar câte sunt)
        else if (vf == 0) corect = false;     // ) fără ( de închis
        else vf--;                            // pop
    if (vf != 0) corect = false;              // au rămas ( neînchise
    cout << (corect ? "DA" : "NU");
    return 0;
}
```

DE CE AȘA?

De ce ajunge un contor pentru un singur fel de paranteze? Toate sunt `(`; nu mai trebuie să știm „care” e în vârf, doar „câte”. Pentru **mai multe feluri** (`()`, `[]`, `{}`) stiva trebuie să rețină tipul, ca mai jos.

Trei feluri de paranteze

Acum închiderea trebuie să se potrivească cu **tipul** vârfului: `)` cu `(`, `]` cu `[`, `}` cu `{`.

```cpp
#include <iostream>
using namespace std;

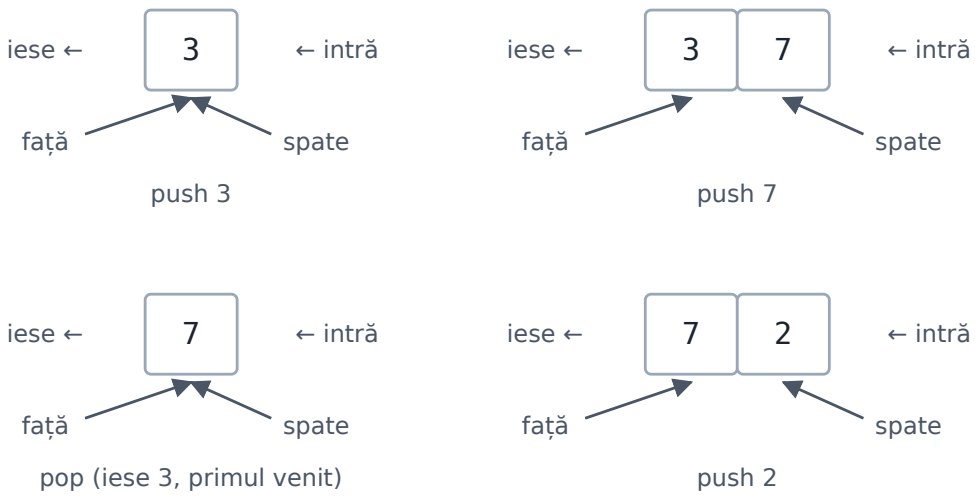
int main()
{
    char s[1005], st[1005];
    cin >> s;
    int vf = 0;
    bool corect = true;
    for (int i = 0; s[i] && corect; i++)
    {
        char c = s[i];
        if (c == '(' || c == '[' || c == '{') st[++vf] = c;
        else
        {
            if (vf == 0) { corect = false; break; }
            char v = st[vf--];                          // paranteza din vârf
            if ((c == ')' && v != '(') || (c == ']' && v != '[') || (c == '}' && v != '{'))
                corect = false;
        }
    }
    if (vf != 0) corect = false;
    cout << (corect ? "DA" : "NU");
    return 0;
}
```

DE CE AȘA?

De ce `([)]` este greșit deși are câte două din fiecare? La `)` în vârf este `[`, nu `(`: închiderea nu se potrivește cu ultima deschisă. Numărarea simplă a perechilor n-ar prinde asta; **stiva** ține minte ordinea.

3. Coada: primul venit, primul plecat (FIFO)

Un rând la casă: intri la **sfârșit**, pleci de la **început**.



Coada

Operație	Cu vector (`q[]` , `prim` , `ultim`)	Cu STL
push x	`q[++ultim] = x;`	`q.push(x)`
pop	`prim++;`	`q.pop()`
primul element	`q[prim]`	`q.front()`
vidă?	`prim > ultim`	`q.empty()`

Cu vector, coada pornește cu `prim = 1` , `ultim = 0` . Spațiul „din față” rămâne nefolosit, dar dacă fiecare element intră o singură dată (ca la BFS), un vector de n + 1 elemente ajunge.

Numărătoarea (problema lui Josephus)

n elevi stau în cerc; se numără din k în k și cel de-al k-lea iese. Cine iese, în ordine? Coada simulează cercul: „rotește” k − 1 elevi (din față în spate), apoi scoate-l pe următorul.



se rotește k − 1 elemente (din față în spate), apoi se elimină următorul

Numărătoare

```cpp
#include <iostream>
using namespace std;

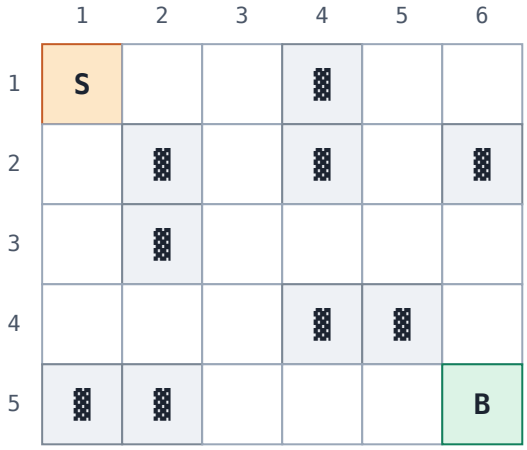
int main()
{
    int n, k, q[100001], prim = 1, ultim = 0, total;
    cin >> n >> k;
    for (int i = 1; i <= n; i++) q[++ultim] = i;
    total = n;
    while (prim <= ultim)
    {
        for (int pas = 1; pas < k; pas++)          // k-1 elevi trec din față în spate
        {
            q[++ultim] = q[prim];
            prim++;
        }
        cout << q[prim] << ' ';                     // iese al k-lea
        prim++;
    }
    return 0;
}
```

DE CE AȘA?

De ce vectorul are 100 001 elemente deși sunt doar n elevi? Fiecare „rotire” scrie un element nou la sfârșit, deci `ultim` crește chiar dacă în coadă sunt puțini elevi. Pentru mult spațiu se folosește coadă circulară (`% n`) sau `queue` din STL.

4. Algoritmul lui Lee: drumul minim într-un labirint

Ai o matrice cu 0 = liber și 1 = zid și vrei numărul **minim** de pași de la start la destinație. Este **BFS** pe matrice: fiecare celulă are cel mult 4 vecini (sus, dreapta, jos, stânga), iar coada păstrează „frontul undei”.



harta: S = start, B = destinație, ▓ = zid

Harta

Pornim din S cu distanța 0. Celulele vecine primesc distanța 1, apoi vecinii lor 2, și tot așa: unda se extinde exact ca o undă pe apă. Prima dată când ajunge la B, a ajuns pe cel mai scurt drum.



culoarea se schimbă la fiecare 2 pași: BFS parcurge „inele” de distanță crescătoare

Unda

Vectorii `di = {-1, 0, 1, 0}` și `dj = {0, 1, 0, -1}` codifică cele 4 direcții: pentru direcția k, vecinul lui (i, j) este `(i + di[k], j + dj[k])`.

Trace pentru primii pași (start = (1,1); coada ca perechi (linie, coloană)):

Pas	scot	vecini liberi și nevizitați (ordine: sus, dreapta, jos, stânga)	coada după
0			(1,1) d=0
1	(1,1)	(1,2) d=1, (2,1) d=1	(1,2) (2,1)

Pas	scot	vecini liberi și nevizitați (ordine: sus, dreapta, jos, stânga)	coada după
2	(1,2)	(1,3) d=2 ((2,2) este zid)	(2,1) (1,3)
3	(2,1)	(3,1) d=2	(1,3) (3,1)
4	(1,3)	(2,3) d=3 ((1,4) este zid)	(3,1) (2,3)

(Atenție: ordinea în care apar vecinii depinde de ordinea direcțiilor din `di`, `dj`. Distanțele nu depind de ordine.)

Pentru a afișa și drumul, reținem la fiecare celulă „de unde am venit” și reconstruim de la destinație:

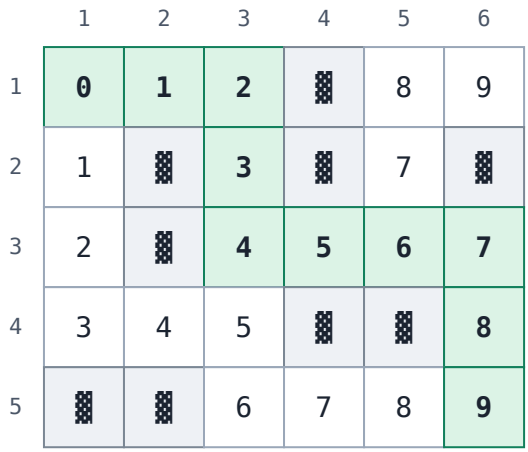
```cpp
#include <iostream>
using namespace std;

int a[102][102], d[102][102], pi[102][102], pj[102][102], n, m;
int qi[10005], qj[10005];
int di[] = {-1, 0, 1, 0}, dj[] = {0, 1, 0, -1};

void afisDrum(int i, int j)
{
    if (pi[i][j] != 0) afisDrum(pi[i][j], pj[i][j]);   // întâi drumul până la părinte
    cout << i << ' ' << j << '\n';
}

int main()
{
    int si, sj, fi, fj;
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) cin >> a[i][j];
    cin >> si >> sj >> fi >> fj;

    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) d[i][j] = -1;      // -1 = nevizitat
    int prim = 1, ultim = 1;
    qi[1] = si; qj[1] = sj; d[si][sj] = 0;
    while (prim <= ultim)
    {
        int i = qi[prim], j = qj[prim];
        prim++;
        for (int k = 0; k < 4; k++)
        {
            int ni = i + di[k], nj = j + dj[k];
            if (ni >= 1 && ni <= n && nj >= 1 && nj <= m && a[ni][nj] == 0 && d[ni][nj] ==
-1)
            {
                d[ni][nj] = d[i][j] + 1;
                pi[ni][nj] = i; pj[ni][nj] = j;           // de unde am venit
                qi[++ultim] = ni; qj[ultim] = nj;
            }
        }
    }
    cout << d[fi][fj] << '\n';
    if (d[fi][fj] != -1) afisDrum(fi, fj);
    return 0;
}
```



d[i][j] = pași de la start (undă care se extinde) · verde: drumul minim, 9 pași

Distanțele și drumul

DE CE AȘA?

De ce verificăm limitele (`ni >= 1 && …`) înaintea lui `a[ni][nj]` ? Altfel am citi în afara matricei (de ex. `a[0][j]`), ceea ce poate da rezultate ciudate sau erori. Alternativă: bordezi harta cu 1 (zid) pe margine și nu mai testezi limitele. **De ce `d[ni][nj] == -1` ca „nevizitat"?** Distanța 0 este validă (startul). **De ce BFS și nu backtracking?** Backtracking-ul ar încerca toate drumurile (număr exponențial); BFS atinge fiecare celulă o singură dată: O(n · m).

Intrare / ieșire

Intrare	Ieșire
harta 5 × 6 din figură, start `1 1` , destinație `5 6`	`9` , apoi cele 10 celule ale drumului
harta 3 × 3 în care startul este înconjurat de ziduri	`-1`

5. Greșeli frecvente

Greșeala	Efect	Corect
pop pe stivă / coadă goală	citire în afara vectorului	verifici `vf > 0` / `prim <= ultim`
`st[vf++] = x`	scrii în poziția 0, nu 1	`st[++vf] = x` (dacă indicii încep de la 1)
paranteze: contor pentru mai multe tipuri	`([)]` ar ieși corect	stivă cu tipul parantezei
Lee: marchezi vecinul la scoatere, nu la adăugare	celule puse în coadă de mai multe ori	marchezi când adaugi
Lee: `d` inițializat cu 0	0 înseamnă și „nevizitat", și „start"	−1 pentru nevizitat
coadă de mărime mică la Lee	depășire	`n · m + 1` elemente

6. La BAC

> **LA BAC**
>
> La Subiectul I apar grile cu o **stivă și o coadă** în care se introduc numere în ordine și se mută elemente de la una la alta („S→C"): desenează cele două structuri pas cu pas. Ideea de bază: stiva inversează ordinea, coada o păstrează. La Subiectul III, aceste structuri apar mai rar direct, dar parantezele corecte și drumurile minime în matrice (Lee) sunt probleme foarte frecvente la clasă și la olimpiadă.

7. Exersează pe hârtie

1. Ce conține stiva după operațiile: push 3, push 7, pop, push 2, push 9, pop, push 4? Care este vârful?

2. Simulează verificarea parantezelor pentru `(()(()))` și pentru `())(` pe hârtie (stiva după fiecare caracter).

3. Într-o stivă se introduc, în ordine, numerele 1, 2, 3, 4, 5. Care este singura ordine posibilă de scoatere dacă nu ai voie să le scoți decât când sunt în vârf și le scoți pe toate la final?

4. O coadă și o stivă conțin inițial, în aceeași ordine, numerele 1, 2, 3, 4 (în stivă 4 este în vârf; în coadă 1 este în față). Se scoate câte un element din fiecare și se adaugă în cealaltă structură, o dată. Ce se află în fiecare după această operație?

5. (BAC) O coadă conține, în această ordine, valorile 5, 8, 3. Se efectuează: `AD 2`, `EX`, `AD 7`, `EX`. Care este conținutul cozii?

6. Pentru `n = 5, k = 3` în problema numărătorii, în ce ordine ies elevii?

7. Pe harta Lee din lecție, care este distanța până la celula (3,5)? Dar până la (5,3)?

8. Verifică dacă `[({})]` și `[(])` sunt corecte, cu stiva.

9. Scrie un program care citește n numere și le afișează în ordine inversă, folosind o stivă.

10. Pe aceeași hartă, dacă zidul din (1,4) ar dispărea, cum s-ar schimba distanța până la B?

8. Mai departe (pentru cine vrea mai mult)

Evaluarea expresiilor în scriere postfixată (poloneză). `3 4 + 2 *` înseamnă (3 + 4) · 2. Se citește de la stânga la dreapta: număr → push; operator → pop două numere, calculează, push rezultat.

```cpp
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int st[1005], vf = 0;
    string t;
    while (cin >> t)
    {
        if (t == "+" || t == "-" || t == "*")
        {
            int b = st[vf--], a = st[vf--];            // întâi b (vârful), apoi a
            if (t == "+") st[++vf] = a + b;
            else if (t == "-") st[++vf] = a - b;
            else st[++vf] = a * b;
        }
        else st[++vf] = stoi(t);
    }
    cout << st[vf];
    return 0;
}
```

Stiva monotonă: următorul element mai mare. Pentru fiecare element, aflii primul element mai mare din dreapta lui, în O(n): parcurgi de la stânga la dreapta; cât timp vârful stivei (indice) are valoare mai mică decât elementul curent, acela își găsește răspunsul.

```cpp
#include <iostream>
using namespace std;

int main()
{
    int n, a[1001], r[1001], st[1001], vf = 0;
    cin >> n;
    for (int i = 1; i <= n; i++) { cin >> a[i]; r[i] = -1; }
    for (int i = 1; i <= n; i++)
    {
        while (vf > 0 && a[st[vf]] < a[i])
            r[st[vf--]] = a[i];                        // a[i] este următorul mai mare
        st[++vf] = i;
    }
    for (int i = 1; i <= n; i++) cout << r[i] << ' ';
    return 0;
}
```

Coadă dublă (deque) și fereastră glisantă. Maximul în fiecare fereastră de lungime k, în O(n), păstrând indici în ordine descrescătoare a valorilor.

Stiva și recursivitatea. Orice program recursiv poate fi rescris cu o stivă explicită (de exemplu DFS iterativ); util când adâncimea recursivității este foarte mare.

Rezumat

- Stiva: LIFO, operații la vârf; coada: FIFO, intri la sfârșit, ieși din față.

- Paranteze corecte: stivă (cu tipul parantezei când sunt mai multe feluri).
- Lee = BFS pe matrice cu `di, dj`; distanța se pune la **adăugarea** în coadă; −1 = nevizitat; parent pentru drum.
- STL: `stack` și `queue` (`push`, `pop`, `top` / `front`, `empty`).

Probleme de lucru (pbinfo)

Începe cu acestea, în ordine: sunt alese să urmeze pașii din lecție.

- [ ] #875 **stiva** — ușoară
  https://www.pbinfo.ro/probleme/875/stiva
- [ ] #876 **coada** — ușoară
  https://www.pbinfo.ro/probleme/876/coada
- [ ] #848 **paranteze1** — ușoară
  https://www.pbinfo.ro/probleme/848/paranteze1

Temă (pbinfo)

Trimite soluțiile pe pbinfo și notează punctajul obținut. Bifează ce ai terminat.

- [ ] #852 **paranteze3** — ușoară
  https://www.pbinfo.ro/probleme/852/paranteze3
- [ ] #877 **cuburi2** — ușoară
  https://www.pbinfo.ro/probleme/877/cuburi2
- [ ] #864 **roboti** — ușoară
  https://www.pbinfo.ro/probleme/864/roboti
- [ ] #866 **acces** — ușoară
  https://www.pbinfo.ro/probleme/866/acces
- [ ] #3277 **lee** — medie
  https://www.pbinfo.ro/probleme/3277/lee
- [ ] #1884 **uemm1** — medie
  Stivă monotonă: următorul element mai mare.
  https://www.pbinfo.ro/probleme/1884/uemm1

Pentru cei care vor mai mult

Probleme mai grele sau de concurs, pentru cine vrea să meargă dincolo de BAC.

- [ ] #2638 **eval-exp** — medie
  https://www.pbinfo.ro/probleme/2638/eval-exp

Grile din subiectele de BAC (pbinfo)

Exerciții de tip grilă din variantele de Bacalaureat, rezolvate online pe pbinfo, cu verificare automată a răspunsului.

- [ ] `#123` **Exercițiul #123**
  stivă și coadă cu aceleași numere: ce se obține după mutarea S→C
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=123

- [ ] `#125` **Exercițiul #125**
  stivă: câte elemente trebuie scoase ca să ajungi la o valoare
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=125

- [ ] `#134` **Exercițiul #134**
  stivă + coadă, a doua variantă a operației de extragere
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=134

- [ ] `#135` **Exercițiul #135**
  care structură aplică principiul FIFO
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=135

- [ ] `#141` **Exercițiul #141**
  stivă cu regulă de adăugare: ce valori pot fi introduse
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=141

- [ ] `#160` **Exercițiul #160**
  coadă: ultima valoare scoasă după o serie de operații
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=160

- [ ] `#183` **Exercițiul #183**
  coadă cu operațiile AD / EX: starea finală
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=183

- [ ] `#279` **Exercițiul #279**
  stivă și coadă împreună: conținutul final
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=279

Citește și exersează în plus

- Structuri de date liniare https://www.pbinfo.ro/articole/13230/structuri-de-date-liniare
- Stiva https://www.pbinfo.ro/articole/19577/stiva
- Coada https://www.pbinfo.ro/articole/19579/coada
- Algoritmul lui Lee https://www.pbinfo.ro/articole/18589/algoritmul-lui-lee
- VisuAlgo — Stack / Queue — operații push, pop, enqueue, dequeue animate (engleză) https://visualgo.net/en/list
- cppreference — std::stack / std::queue — documentația oficială pentru containerele STL (engleză) https://en.cppreference.com/w/cpp/container/stack

- USACO Guide — Stacks — stiva monotonă și alte aplicații cu probleme de concurs (engleză)
  https://usaco.guide/silver/stacks

Răspunsuri

Exersează pe hârtie

1. push 3 → [3]; push 7 → [3, 7]; pop → [3]; push 2 → [3, 2]; push 9 → [3, 2, 9]; pop → [3, 2]; push 4 → [3, 2, 4]. Stiva conține **3, 2, 4** (de la bază la vârf), vârful este **4**.

2. `(()(()))`: stiva după fiecare caracter are 1, 2, 1, 2, 3, 2, 1, 0 paranteze: corect. `())(`: după `(` → 1; după `)` → 0; al doilea `)` vine pe stivă goală → **greșit**.

3. Când scoți toate elementele, ies în ordine inversă: **5, 4, 3, 2, 1** (singura ordine, dacă nu amesteci introducerile cu scoaterile).

4. Din coadă iese 1 (primul venit), din stivă iese 4 (vârful). 1 se adaugă în stivă, 4 se adaugă în coadă. Coada devine **2, 3, 4, 4** (față → spate), iar stiva **1, 2, 3, 1** (bază → vârf).

5. `AD 2` → 5, 8, 3, 2; `EX` scoate 5 → 8, 3, 2; `AD 7` → 8, 3, 2, 7; `EX` scoate 8 → **3, 2, 7**.

6. Coada 1 2 3 4 5: se rotesc 1, 2 și iese 3 → coada 4 5 1 2; se rotesc 4, 5 și iese 1 → 2 4 5; se rotesc 2, 4 și iese 5 → 2 4; se rotesc 2, 4 și iese 2 → 4; iese 4. Ordinea: **3, 1, 5, 2, 4**.

7. Din figura distanțelor: (3,5) are **6**; (5,3) are **6**.

8. `[({})]`: corect (fiecare închidere potrivește vârful). `[(])`: la `]` vârful este `(`, deci **greșit**.

9. Citești numerele cu `st[++vf] = x`, apoi `while (vf > 0) cout << st[vf--] << ' ';`.

10. Distanța până la B rămâne **9**: drumul prin (3,5) nu se scurtează. Se scurtează doar distanțele până la celulele din colțul de sus: (1,4) devine 3, (1,5) devine 4, (1,6) devine 5, iar (2,5) devine 5.