Tema 2 - Stive, cozi, liste înlănţuite

1. **Implementarea unei liste dublu înlănţuite**. Să se implementeze o listă dublu înlănţuită cu funcţionalităţile descrise în continuare. Se cere utilizarea unei structuri *node* care are trei câmpuri: un câmp pentru informaţie (de tip int) şi două câmpuri de tip pointer la *node* pentru legăturile către elementele precedent şi următor. Se cere utilizarea unei structuri *List* care are ca membri două variabile de tip $node*$ reprezentând primul respectiv ultimul element din listă, o variabilă de tip int reprezentând numărul de elemente din listă şi funcţiile:

   - $push\_front(key)$ - adaugă cheia *key* în capul listei (0.20 p)
   - $push\_back(key)$ - adaugă cheia *key* la finalul listei (0.20 p)
   - $pop\_front()$ - şterge primul element din listă (0.20 p)
   - $pop\_back()$ - şterge ultimul element din listă (0.20 p)
   - $find(key)$ - caută o cheie *key* în listă - returnează pointer la nodul cu cheia *key* sau NULL (0.20 p)
   - $erase(node*Nod)$ -şterge un element Nod din listă (NU implică căutare). Nodul transmis ca parametru a fost în prealabil găsit cu $find$ sau identificat prin parcurgerea listei. (0.20 p)
   - $remove(int\ key)$ - şterge toate aparitiile cheii *key* (implică căutare) (0.20 p)
   - $insert(node*Nod, int\ val)$ - inserează un element cu cheia *val* înainte de nodul indicat de Nod. (0.5 p)
   - $empty()$ - verifică dacă lista e vidă (0.20 p)
   - $clear()$ - goleşte lista. (0.20 p)
   - funcţia $print()$ - afişează elementele listei (0.20 p)
   - funcţia $size()$ - returnează numărul de elemente din listă.

   De asemenea să se implementeze următoarele funcţii, care nu fac parte din structură:

- *palindrom*(*List L*) - verifică dacă lista este palindrom (0.5 p)
- *compare*(*List L*1, *ListL*2) - returnează 1 dacă $L1$ şi $L2$ sunt identice şi 0 altfel. (0.20 p)

În funcţia *main* realizaţi un meniu cu ajutorul unei instrucţiuni *switch*, prin care se oferă opţiuni, corespunzătoare fiecărei funcţionalităţi, precum şi o opţiune de EXIT. Într-o instrucţiune *while*, se citesc şi se execută opţiuni până la alegerea opţiunii de EXIT.

ATENŢIE: Nici o funcţie nu trebuie să dea eroare de execuţie, dacă se apelează pe o listă vidă!!!

2. **Inversare elemente vector**. Să se inverseze elementele unui vector utilizând o stivă. Primul element se va intershimba cu ultimul, al doilea cu penultimul, etc. (1p)

3. **Implementare coadă**. Să se implementeze o coadă utilizând liste înlănţuite. Vă trebuie:

   - o structură *node* cu două câmpuri - un câmp pentru informaţie (de tipul cerut de problema curentă) şi un câmp de tip pointer la *node* pentru legătura la elementul următor.
   - o structură *Queue* cu
     - două câmpuri de tip pointer la nod, pentru primul şi ultimul element iniţializate cu NULL (nullptr).
     - un câmp nr_elem de tip int - numărul de elemente din coada.
     - funcţia *push*(*elem*) - pune *elem* la sfarsitul cozii
     - funcţia *pop*() - elimină elementul de la începutul cozii
     - funcţile $front()$ şi *back*() returnează primul respectiv ultimul element din coadă
     - funcţia *empty*() - verifică dacă coada este vidă.
     - funcţia *clear*() - goleşte coada
     - funcţia *size*() - returnează numărul de elemente din coada

   Această coadă va fi utilizată în următoarea problemă:

   La un examen se pot prezenta candidaţi pe durata a două zile. În fiecare zi timpul alocat pentru examinare este de t ore ($t \leq 6$). La examen se înscriu n candidaţi. Se citesc din fişier t, n precum şi candidaţii cu numele (de tip **std::string**). Ei vor fi introduşi într-o coadă, de unde vor fi extraşi pe rând

pentru examinare. Pentru fiecare candidat, care este la rând, se generează o durată aleatorie cu o valoare între 5 minute şi 15 minute. În momentul în care timpul t s-a terminat, deci se încheie prima zi de evaluare, candidaţii care au rămas în coada vor fi extraşi pe rând şi trecuţi într-un fişier de ieşire, care va reprezenta lista candidaţilor pentru ziua a doua de examinare.

Punctajul pentru problemă: 1p pentru implementarea cozii + 1p pentru rezolvarea problemei. Folosiţi nume semnificative pentru variabilele folosite (chiar dacă în enunţ s-au folosit denumiri precum t şi n)!

4. **Parantezare corectă**: Se dă un şir de paranteze deschise şi închise de tip (,), [,], {, }. Să se verifice dacă şirul este corect. Folosiţi o stiva (std::stack) pentru rezolvare. **Exemplu**: şirul [()()] este corect, şirul ([]) nu este corect, şirul ()] (nu este corect. (1p)

5. **Simularea unei stive prin două cozi**. Rezolvaţi problema parantezării, simulând funcţionarea stivei cu ajutorul a două cozi. Puteţi folosi *queue* din STL. Cerinţă: definirea unei structuri *stiva*, care în interior să dispună drept containere de două elemente de tip *queue* şi de funcţii de tip *push* şi *pop*, care apelează însă doar operaţiile corespunzătoare ale cozilor din structură. Elementele care se introduc în stivă vor fi stocate în cozile de care dispune stiva. Explicitaţi într-un comentariu complexitatea operaţiilor de adăugare şi extragere din stiva dvs. (1p)

6. **Evaluarea expresiilor aritmetice**. Se citeşte dintr-un fişier un şir de expresii aritmetice alcătuite din numere întregi fără semn, operatorii aritmetici +, −, ∗, /, ^ (reprezentând ridicare la putere) şi paranteze rotunde. Expresiile pot fi despărţite între ele prin ; sau prin trecere la rând nou. **Atenţie:** algoritmul trebuie să funcţioneze corect şi dacă există caractere "albe" în expresiile considerate (spaţii, tab-uri) şi dacă NU există! Spaţiile albe se ignoră la analizarea acestor expresii.

   Pentru fiecare expresie:

   - Să se afişeze întâi expresia aritmetică.
   - Să se construiască forma poloneză postfixată şi să se afişeze. (1p)
   - Să se evalueze expresia şi să se afişeze rezultatul. (1p)
   - Să se semnaleze erori în expresie (de parantezare, de operatori, de caractere nepermise). Dacă într-o expresie se găsesc erori se întrerupe analiza/evaluarea acesteia după mesajul de eroare şi se trece la următoarea expresie. (1p)

Pentru citirea şi procesarea conform enunţului a mai multor expresii dintr-un fişier - 0.5p.

Puncte suplimentare

- Pentru funcţionarea algoritmului şi cu numere întregi de mai multe cifre (1p)
- Pentru funcţionarea algoritmului şi cu numere reale (1p)

Utilizaţi stive din STL. Folosiţi funcţii separate pentru construirea formei poloneze pentru o expresei aritmetică stocată într-un **string** şi pentru evaluarea pornind de la o formă poloneză.

Exemple:

- $2 + + + 3$ nu este o expresie corectă. Programul semnalează că există prea mulţi operatori.
- $2 * ((3 + 4)$ nu este o expresie corectă. Programul semnalează faptul că parantezarea nu este corectă.
  Observaţie: chiar dacă numărul de paranteze deschise este egal cu cel de paranteze închise, parantezarea poate să nu fie corectă. De exemplu))((sau ()))((.
- $23\# + a$ nu este o expresie corectă. Programul semnalează faptul că apar caractere nepermise $\#$, a.
- $3 * 4 - 3 * (24 - 12) - 7$. Programul afişază valoarea -31. Forma poloneză postfixată este $34 * 32412 - * - 7-$

Algoritmul este descris în documentaţia de pe e-learning.

7. **Eliminarea parantezelor redundante**. Se citeşte dintr-un fişier un şir de caractere reprezentând o expresie aritmetică formată din numere, operatori (+, - /, *), litere (reprezentând variabile), paranteze rotunde, eventual spaţii albe. Se cere elminarea parantezelor redundante. (3p)

   Exemple:

   (a) Pentru expresia: $((x + y) * 3 + ((z)))$ rezultatul algoritmului ar trebui sa fie: $((x + y) * 3 + z)$.

   (b) Pentru expresia: $(((x+y))*3+((z-x)))$ rezultatul algoritmului ar trebui sa fie: $((x + y) * 3 + (z - x))$ sau mai bine $(x + y) * 3 + (z - x)$

8. **Deque.** Implementaţi o structură de tip *deque*. Această structură trebuie să dispună de un container secvenţial în care se vor stoca elementele, de funcţii de tip *push-back*, *push-front*, *pop-back*, *pop-front*, *size*, *empty*, toate în complexitate constantă. De asemenea trebuie să permită accesul prin poziţie la elemente (de exemplu printr-o funcţie *at*) şi funcţii de inserare / ştergere de pe o anumită poziţie.

   Există următoarele variante de implementare:

   (a) Folosind o coadă circulară care are la bază un vector (implementat cu un vector alocat dinamic). În acest caz trataţi situaţia umplerii containerului prin realocare. (2p)

   (b) Folosind un vector de array-uri (blocuri de memorie de dimensiune fixă). Pentru acest lucru consultaţi materialul de curs sau puneţi întrebări suplimentare la laborator. (5p)

   Observaţie: Se punctează doar una dintre abordări, nu amândouă.

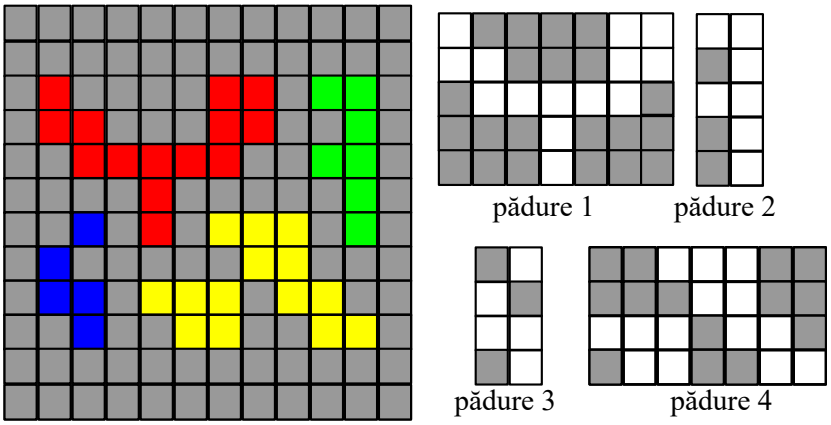

Figure 1: Exemplu de imagine conţinând 4 zone de pădure. În matricea corespunzătoare poziţiile colorate sunt pădure.

9. **Problema urşilor**. La Brasov pe Tâmpa s-au aventurat doi urşi, care reprezintă un pericol pentru populaţie. Primăria a hotărât relocarea lor în altă zonă împădurită. În acest scop este studiată harta unei regiuni potrivite cu acest scop. Harta este reprezentată printr-o matrice (dimensiune minimă 12×12), în care fiecare poziţie reprezintă un hectar de teren. Poziţiile de pădure sunt marcate cu P, celelalte cu $-$. O zonă împădurită compactă este reprezentată de mulţimea tuturor celulelor marcate cu P conectate.

   Se va alege pentru relocarea urşilor acea zonă de pădure care:

- are suprafaţa mai mare decât un parg dat T (citit din fişier).
- are compactitatea (raportul dintre suprafaţa sa şi suprafaţa dreptunghiului în care este inscrisă) maximă.

În exemplul din figura 1 se consideră pragul $T = 10$. Regiunea roşie şi regiunea galbenă satisfac ambele cerinţa ca suptafaţa să fie mai mare ca T. Compactitatea zonei de pădure roşu este $14/35 = 0.4$, iar a celei galbene este $14/28 = 0.5$. Deci urşii vor fi relocaţi în zona galbenă. Rezultatul va fi indicat prin perechile de coordonate ale colţului stânga-sus şi ale colţului drepata-jos corespunzătoare dreptunghiului în care este înscrisă zona. În exemplul de mai sus $(4, 6)$ şi $(10, 9)$ (coordonate ecran!)

Punctaj:

- identificarea corectă a suprafţelor de pădure - 1.5p
- selectarea corectă suprafeţei cerute - 1.5p

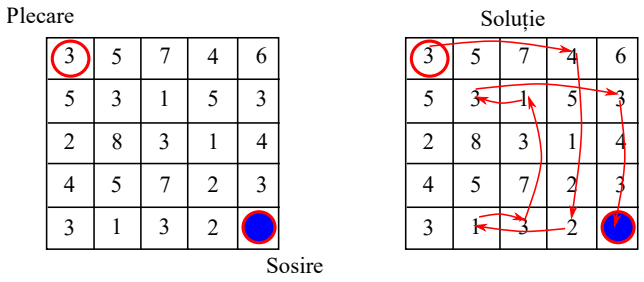

Figure 2: Joc numere exemplu de rezolvare

10. **Joc numere**. Se consideră o matrice $n \times n$ de numere întregi pozitive. Un personaj pleacă din colţul stânga-sus şi trebuie să ajungă în colţul drepta jos. La fiecare mutare personajul poate să se deplaseze în sus, în jos, în stânga sau în dreapta cu exact atâtea poziţii câte indică numărul din căsuţa pe care se află la momentul curent. Nu are voie însă să depăşească marginile matricei. Să se determine numărul minim de mutări necesare pentru a ajunge din colţul stânga sus în colţul dreapta jos. Dacă nu este posibil, să se semnaleze acest lucru.Un exemplu este prezentat în figura 2. [1]

11. **Problema bazinelor**: Un piscicultor dispune de un teren dreptunghiular (dimensiune minimă 15×15), pe care sunt amplasate un număr N (care iniţial

[1] problemă preluată din "Algorithms" de Jeff Erickson, http://jeffe.cs.illinois.edu/teaching/algorithms/#book

nu este cunoscut) bazine pentru peşti. Terenul este reprezentat de o matrice de numere, în care fiecare poziţie reprezintă un metru pătrat de tren, iar valoarea reprezintă adâncimea în metri. Valorile mai mici decât 0 reprezintă regiuni care vor face parte dintr-un bazin, iar valorile mai mari ca 0 reprezintă regiuni uscate. Un bazin este reprezentat de toate poziţiile din matrice înconjurate complet de o regiune uscată.

- Să se determine câte bazine naturale există pe acest teren (etichetând pe fiecare în mod corespunzător). (1.5p)
- Să se pună în evidenţă marginile fiecărui bazin printr-o afişare corespunzătoare pe ecran a matricei. Marginile sunt pixeli care însă nu fac parte din bazin (deci au valori ≥ 0) (1p)
- Pentru a putea creşte peşti de dimensiuni mai mari, piscicultorul are nevoie de un bazin cu o regiune compactă de adâncime cel puţin h şi un volum de cel puţin V. Înţelegem prin regiune compactă o mulţime de poziţi pentru care oricare două poziţii (x_1, y_1) şi (x_2, y_2) pot fi conectate între ele printr-ul şir de poziţii intermediare vecine două câte două (considerând vecini orizontali şi verticali). Verificaţi dacă dispune de unul / mai multe astfel de bazine şi în caz contrat puneţi în evidenţă bazinele respective. (2p)
  Citirea matricii ce reprezintă terenul se realizează dintr-un fişier.

Observaţii: Pentru rezolvarea problemelor

- NU utilizaţi funcţii recursive pentru rezolvarea problemelor. Puteţi folosi STL (*vector*, *pair*, *tuple*, *list*, *stack*, *queue* etc.)
- Pentru rezolvarea eficientă cu seturi disjuncte a ultimelor două probleme se oferă punctaj suplimentar.