Tema 4 - Arbori rădăcină. Heap-uri. Cozi de prioritate.

Arbori

1. **Arbore sintactic.** Se citeşte din fişier o expresie aritmetică formată din numere, variabile şi operatorii de bază (+, -, *, /) şi paranteze.

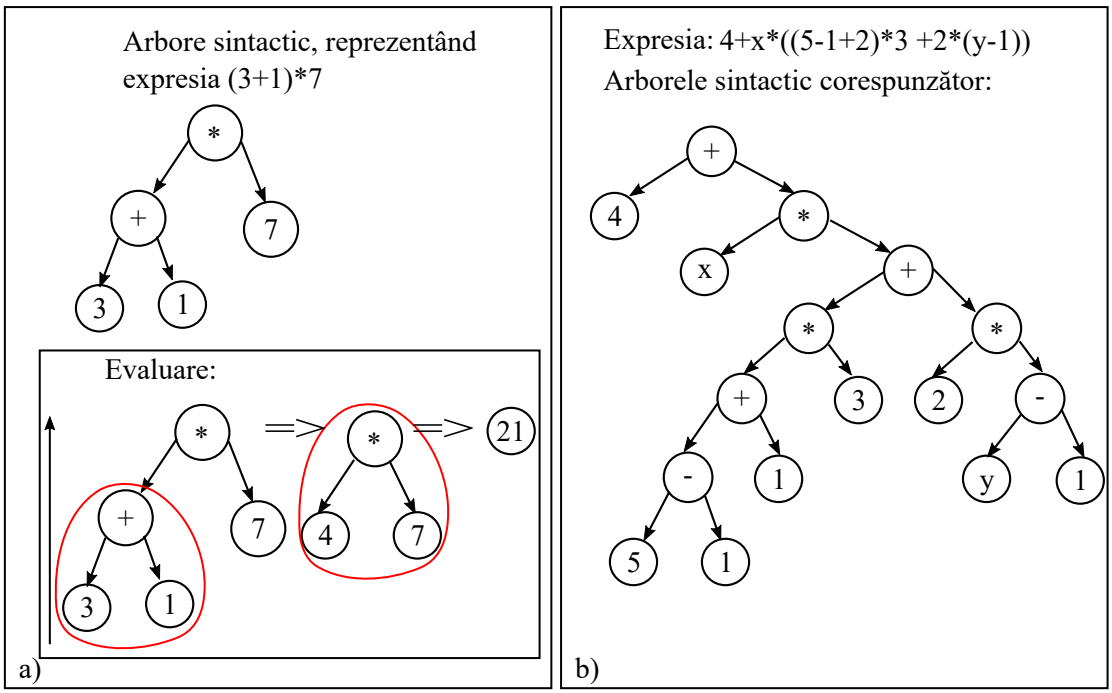


Figure 1: a) Exemplu de arbore sintactic pentru o expresie aritmetică simplă împreună cu modul de evaluare. b) Expresia aritemetică împreună cu arborele sintactic corespunzător.

a. Să se construiască un arbore sintactic corespunzător expresiei. (2p)

b. Să se afişeze arborele pe niveluri.(0.5p)

c. Pentru variabile să se ceară valori (double) şi apoi să se evalueze expresia. Permiteţi repetarea acestui pas. (acelaşi arbore dar valori diferite pentru variabile) (2p)

Observaţie: într-un astfel de arbore sintactic, care este arbore binar, nodurile interne conţin operatorii aritmetici, iar frunzele conţin operanzii. O operaţie dintr-un nod intern se efectuează între valorile reprezentate de rezultatele subarborilor copii ai acestui nod. Arborele sintactic permite evaluarea expresiei aritmetice în manieră *bottom-up*. Acest lucru înseamnă că operatorii care se vor aplica mai întâi se află pe niveluri mai mari decât cei care se vor aplica mai apoi. Un exemplu este prezentat în figura 1 a.

Exemplu: Pentru expresia aritmetică 4+x*((5-1+2)*3 +2*(y-1)) arborele corespunzător este cel din figura 1b. Indicaţie: se poate folosi forma poloneză postfixată pentru construcţia arborelui.

2. **Operaţii pe un arbore binar**. Se citeşte dintr-un fişier un arbore binar. (1p). Modul de citire este la alegere.Variante propuse:

   - se citesc 2 parcurgeri din care se construieşte arborele
   - se citesc: 3 vectori - unul cu cheile arborelui, unul cu copii stângi pentru fiecare nod, unul cu copii drepţi pentru fiecare nod
   - citire pe niveluri (în lăţime)
   - printr-un vector de taţi

   Să se implementeze următoarele funcţii pentru arbore:

   - O funcţie ce calculează înălţimea unui subarbore (0.5p).
   - Funcţii pentru parcurgerea arborelui (RSD, SRD, SDR, Niveluri) (0.5p).
   - O funcţie care afişează frunzele de la stânga la dreapta (0.5p).
   - O funcţie care verifică dacă doi arbori sunt identici (1p).
   - O funcţie care verifică dacă un arbore e complet (0.5p).
   - O funcţie care determină adâncimea unui nod (0.5p).

3. **Arbore genealogic.** Se consideră arborele genealogic al unei familii. Construiţi o structură corespunzătoare, care să conţină: (2p)

   - o funcţie de citire a arborelui din fişier
   - o funcţie de afişare pe niveluri, astfel încât la afişare nivelurile să fie separate prin 2 rânduri libere şi pentru fiecare persoană să se afişeze numele său, numele părintelui şi numărul de copii. (Pentru o afisare grafică a arborelui se oferă suplimentar 2p).

- o funcţie, care are ca parametru un nume şi afişază numele tatălui şi numele copiilor perosanei respective, dacă aceasta se găseşte în arbore. Altfel semnalează faptul, că nu a fost găsită persoana.
- o funcţie care afişază numărul de generaţii reprezentate în arbore

Heap. Coadă de prioritate

4. **Sortare** Să se implementeze algoritmul **Heap-Sort**. Să se sorteze un vector de numere. (1p)

5. **Prority queue**. Implementaţi o coadă de priorităţi folosind o structură (clasă) PRIORITY_QUEUE, care să aibă un câmp DATA de tip vector de întregi, care să stocheze elementele cozii sub forma unui heap max şi un câmp SIZE - nr. de elemente stocate în coadă. În plus structura trebuie să aibă metodele:

   - INSERT - inserează un nou nod în coadă
   - EXTRACT_MAX - extrage elementul de prioritate maximă din coadă
   - MAX_ELEMENT - returnează elementul de prioritate maximă
   - INCREASE_KEY - creşte prioritatea unui nod
   - MAX_HEAPFY (sau SIFT_DOW) - funcţia care coboară o cheie pe poziţia corespunzătoare din heap

   În funcţia *main* se declară o variabilă de tip PRIORITY_QUEUE şi se foloseşte un *menu* implementat cu ajutorul unei instrucţiuni *switch*, prin care utilizatorul să poată selecta oricare dintre operaţiile de inserţie, extragerea maximului, obţinerea maximului şi afişarea elementelor din heap. (2p)

   Observaţie: daca pentru stocarea datelor se foloseşte **std::vector**, atunci NU trebuie câmpul **size**. Se va folosi *size*-ul vectorului. Adaptaţi corespunzător funcţiile pentru heap.

6. **Codificarea Huffmann**. Se citeşte un text dintr-un fişier. Să se construiască arborele de codificare Huffmann corespunzător. Să se afişeze codul corespunzător fiecărui caracter şi să se codifica textul. **std::priority_queue** (min). (3p)

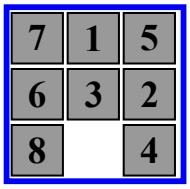
7. **Interclasarea optimală**. Se consideră n vectori de numere întregi sortaţi crescător. Să se interclasesez aceşti vectori într-unul singur cu număr minim de comparaţii. (2p)

8. **Problema excursionistului.** Un drumeţ trebuie sa ajungă de la localitatea $A(x_0, y_0)$ la localitatea $B(x_1, y_1)$ parcurgând un relief variat. Excursionistul caută drumul cel mai convenabil din punct de vedere al efortului depus. Harta este reprezentată de o grilă dreptunghiulară de numere, reprezentând denivelarea punctului respectiv faţă de nivelul mării (0). Costul deplasării de la un pătrat al grilei la altul este dat de diferenţa de nivel + 1. În plus, drumeţul nu poate depăşi o diferntă de nivel mai mare de 2 unităţi. De asemenea nu poate trece prin locuri mai adânci de -2. Deplasarea se face sus-jos-stânga-dreapta (nu pe diagonală). Reprezentaţi pe hartă un traseu optim pentru drumeţ. Folosiţi algoritmul A^* (4p)

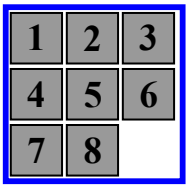
   Exemplu: se consideră harta 10×12 din figură. Drumul este marcat cu roşu de la punctul de start (roşu) la puncrul de sosire (verde).

$$\begin{pmatrix}
1 & 2 & 2 & 3 & 1 & 0 & 3 & 2 & 2 & 0 & 1 & 1 \\
1 & 2 & 3 & 4 & 4 & 3 & 3 & 2 & 0 & 1 & 2 & 1 \\
0 & 3 & 4 & 7 & 6 & 3 & 3 & 2 & -1 & -1 & -3 & 1 \\
-1 & 2 & 3 & 6 & 5 & 1 & 3 & 1 & -1 & -2 & -3 & 1 \\
1 & 2 & 3 & 1 & 2 & 1 & -2 & -1 & 1 & 2 & -3 & 1 \\
2 & 3 & 4 & 7 & 7 & -1 & -3 & -1 & 1 & 0 & 1 & 1 \\
1 & 3 & 6 & 4 & 3 & -1 & -3 & 0 & 1 & 0 & 1 & 1 \\
1 & 3 & 5 & 4 & 2 & -1 & -2 & 0 & 2 & 0 & 1 & 1 \\
2 & 3 & 3 & 5 & 4 & 1 & 3 & 0 & 3 & 0 & 1 & 1 \\
2 & 2 & 3 & 2 & 2 & 1 & 3 & 0 & 7 & 0 & 1 & 1
\end{pmatrix}$$

9. **Rezolvare puzzle**: Se consideră o jocul următor. Se consideră 8 plăcuţe pătrate numerotate de la 1 la 8, plasate într-o ramă pătrată de dimensiune 3×3. O poziţie este liberă. Orice plăcuţă vecină cu poziţia liberă poate fi glisată pe această poziţie. Se cere şirul de configuraţii (afişat), care duc la aranjarea numerelor în ordine crescătoare, cu poziţia liberă în colţul dreapta-jos al cadrului. Un exemplu este prezentat în figură. (4p)



Configurație inițială



Configurație finală

Folosiţi algoritmul A*, precum şi structurile de date din stl potrivite (priority_queue, unordered_map, vector etc.)
Costul unei configuraţii = numărul de mutări care s-a efectuat din configuraţia iniţială până la aceasta.
Euristica 1: - numărul de plăcuţe care încă nu se află pe poziţia potrivită.
Păstraţi configuraţiile deja generate împreună cu o legătură (definită în mod potrivit) către configuraţia din care s-a obţinut (parintele) într-un unordered_map, pentru a putea apoi genera şirul de configuraţii de la cea iniţială la soluţie. Pentru exemplul din fighură $h_1 = 9$
Euristica 2: suma distanţelor de la fiecare piesă la poziţia corectă (city-block distance). Pentru exemplul din figură avem $h_2 = 3+1+2+2+2+2+1+3 = 18$. Euristica 2 este mai eficientă decât 1, pentru ca se apropie mai bine de costul real.
Indicaţie - pentru simplificare, consideraţi la fiecare pas, că se mută poziţia liberă sus, jos, la stânga sau la dreapta.

Pentru A* vă recomand următorul site:
https://www.redblobgames.com/pathfinding/a-star/introduction.html