Laborator 4 - Stivă şi Coadă

Responsabili:

- Gabriel Bercaru [mailto:mailto:gabriel.bercaru10@gmail.com]
- Alexandru-Mihai Stroie [mailto:mailto:andistroie@gmail.com]
- Cosmin Petrișor [mailto:mailto:cosmin.ioan.petrisor@gmail.com]

Obiective

În urma parcurgerii acestui laborator studentul va fi capabil să:

- înțeleagă principiul de funcționare al structurilor de date stivă şi coadă
- implementeze o stivă şi o coadă folosind un vector pentru stocarea elementelor
- transforme o expresie din forma infixată în formă postfixată
- să evalueze o expresie în formă postfixată
- implementeze algoritmul de sortare Radix Sort

Stivă (Stack)

O stivă este o instanță a unui tip de date abstract ce formalizează conceptul de colecţie cu acces restricționat. Restricția respectă regula LIFO (Last In, First Out).

Accesul la elementele stivei se face doar prin vârful acesteia.

Operații:

- *void push(E element)* – adaugă un element (entitate) în stivă. Adăugarea se poate face doar la vârful stivei.
- *E pop()* – șterge un element din stivă și îl returnează. Ștergerea se poate face doar la vârful stivei.
- *E peek()* – consultă (întoarce) elementul din vârful stivei fără a efectua nicio modificare asupra acesteia.
- *bool isEmpty()* – întoarce *1* dacă stiva este goală; *0* dacă are cel puțin un element

Implementare

O structură de date definește un set de operații și funcționalitatea acestora.
Implementarea efectivă a unei structuri de date poate fi realizată în diverse moduri, cât timp funcționalitatea este păstrată.

O stivă poate fi implementată cu ajutorul unui **vector** sau cu **liste înlănțuite**.

În cadrul acestui laborator, ne vom concentra asupra implementării unei stive cu ajutorul unui vector de stocare.

Reprezentare internă cu vector

La nivel de implementare, stiva este reprezentată printr-o clasă ce folosește (pe lângă operațiile ce pot fi efectuate asupra ei) un vector de stocare (*stackArray*) de o dimensiune maximă dată (*NMAX*) și un indice ce indică vârful stivei(*topLevel*).

Stack.h

```
#ifndef __STACK__H__
#define __STACK__H__

// Primul argument al template-ului este tipul de date T
// Al doilea argument este dimensiunea maxima a stivei N
template<typename T, int N>
class Stack {
private:
    // Vectorul de stocare
    T stackArray[N];

    // Pozitia in vector a varfului stivei
    int topLevel;

public:
    // Constructor
    Stack() {
        // TODO: initializari
    }

    // Destructor
    ~Stack() {
        // TODO: eliberare resurse, daca este cazul
    }

    // Operator de adaugare
    void push(T x) {
        // TODO: verificari, implementare
    }

    // Operatorul de stergere
    void pop() {
        // TODO: verificari, implementare
    }

    // Operatorul de consultare
    T peek() {
        // TODO: verificari, implementare
    }

    // Operatorul de verificare dimensiune
    bool isEmpty() {
        // TODO: implementare
    }
};

#endif //__STACK__H__
```

Exemplu de utilizare

Forma poloneză inversă

Forma poloneză inversă [http://en.wikipedia.org/wiki/Reverse_Polish_notation] este o notație matematică în care fiecare operator urmează dupa toți operanzii săi.

Cel mai simplu exemplu de notație postfixată este cel pentru doi operanzi și un operator:

5 + 4	**se scrie sub forma**	**5 4 +**

În cazul în care există mai multe operații, operatorul apare imediat după cel de-al doilea operand:

2 + 4 - 5	**se scrie sub forma**	**2 4 + 5 -**

Avantajul major al formei poloneze inverse este faptul că elimină parantezele din cadrul expresilor:

5 + (1 + 4)	**se scrie sub forma**	**5 1 4 + +**

Algoritmul de conversie a unei expresii din formă infixată în formă postfixată

```
1. cât timp există elemente de citit
    1.1 citește un element
    1.2 dacă elementul este un număr, afișare (se adaugă la forma
        postfixată)
    1.3 dacă elementul este o paranteză stângă, adaugă-l în stivă
    1.4 dacă elementul este o paranteză dreaptă, extrage operatorii din
        stivă și adaugă-i la forma postfixată până când vârful stivei
        ajunge o paranteză stângă (care este extrasă, dar nu este
        adăugată la forma postfixată).

    !!! dacă stiva s-a golit fără să fie găsită o paranteză stângă,
        înseamnă că expresia inițială avea paranteze greșite
    1.5 dacă elementul este un operator (fie el O1)
        1.5.1 cât timp există un alt operator în vârful stivei (fie el O2)
              ȘI precedența lui O1 este MAI MICA SAU EGALA decât cea a lui O2, extrage O2
              din stivă, afișare (se adaugă la forma postfixată)
        1.5.2 adaugă O1 în stivă
2. când nu mai există elemente de citit, extrage toate elementele rămase
   în stivă și adaugă-le la forma postfixată (elementele trebuie să fie
   numai operatori; dacă este extrasă o paranteză stângă expresia inițială
   avea parantezele greșite).
```

Exemplu

Fie expresia:

```
1 - 7 * 2 /(3 + 5)^2^5
```

Element	Acțiune	Forma postfixată	Stiva	Observaţii
1	Adaugă element la forma postfixată	1		
-	Pune elementul în stivă	1	-	
7	Adaugă element la forma postfixată	1,7	-	
*	Pune elementul în stivă	1,7	* -	* are precedență mai mare decât -
2	Adaugă element la forma postfixată	1,7,2	* -	
/	Extrage element din stivă	1,7,2*	-	/ și * au aceeași prioritate
	Pune elementul în stivă		/ -	/ are precedență mai mare decât -
(	Pune elementul în stivă	1,7,2*	(/ -	
3	Adaugă element la forma postfixată	1,7,2*3	(/ -	
+	Pune elementul în stivă	1,7,2*3	+ (/ -	
5	Adaugă element la forma postfixată	1,7,2*3,5	+ (/ -	
)	Extrage element din stivă	1,7,2*3,5+	(/ -	Se repeta până când se întâlnește (
	repetă		/ -	(a fost ignorat
^	Pune elementul în stivă	1,7,2*3,5+	^ / -	^ are precedență mai mare decât /

2	Adaugă element la forma postfixată	1,7,2*3,5+2	^ / -	
^	Pune elementul în stivă	1,7,2*3,5+2	^ ^ / -	^ este considerat asociativ-dreapta
5	Adaugă element la forma postfixată	1,7,2*3,5+2,5	^ ^ / -	
Final	Extrage toate elementele din stivă	1,7,2*3,5+2,5^^/-		

Algoritmul de evaluare a unei expresii în formă postfixată

```
1. cât timp există elemente de citit
    1.1 citește un element
    1.2 dacă elementul este o valoare
            1.2.1 pune elementul în stivă
        altfel (elementul este un operator)
            1.2.2 extrage 2 operanzi din stivă
            1.2.3 dacă nu există 2 operanzi în stivă
                EROARE: forma postfixată nu este corectă
            1.2.4 evaluează rezultatul aplicării operatorului asupra celor doi
                  operanzi
            1.2.5 pune rezultatul în stivă
2. dacă există o singură valoare în stivă
        2.1 afișează valoarea ca rezultat final al evaluării expresiei
    altfel
        EROARE: forma postfixată nu este corectă
```

Coadă (Queue)

O coadă este o structură de date organizată după modelul FIFO (First In, First Out): primul element introdus va fi primul eliminat din buffer.

Metode generale disponibile pentru o clasă Queue:

- *void enqueue(E element)* – adaugă elementul ”element” la sfârșitul cozii. Adăugarea se face doar la sfârșitul cozii.
- *E dequeue()* – șterge un element din coadă și îl returnează. Ștergerea se poate face doar la începutul cozii.
- *E peek()* – întoarce primul element din coada fără a-l scoate din aceasta.
- *bool isEmpty()* – întoarce *true* dacă coada este goală; *false* dacă are cel puțin un element

Variante de implementare

O coadă se poate poate implementa folosind pe post de container **lista înlănțuită**, **array de dimensiune fixă**, **array circular**.

Reprezentare internă cu listă înlănțuită

- *void enqueue(E element)* - echivalent cu operația de addLast() efectuată pe listă. Complexitate: timp constant - O(1)
- *E dequeue()* echivalent cu operația de removeFirst(). Complexitate: timp constant - O(1)
- *E peek()* presupune returnarea valorii stocate in `head`-ul listei. Complexitate: timp constant - O(1)
- *bool isEmpty()* - la fel ca în cazul listei. Complexitate: timp constant - O(1)

Reprezentare internă cu vector

Vom avea doi indici (**head** și **tail**) ce vor reprezenta începutul, respectiv sfârșitul cozii în cadrul vectorului. Apare însă următoarea problemă din punctul de vedere al spațiului neutilizat: întotdeauna spațiul de la **0** la **head-1** va fi nefolosit, iar numărul de elemente ce pot fi stocate în coadă va scădea (având inițial N elemente ce pot fi stocate, după ce se extrage prima oară un element, mai pot fi stocate doar N-1 elemente). Vrem ca întotdeauna să putem stoca maxim N elemente.

Soluția: vector circular.

- *void enqueue(E element)* - presupune adăugarea noului element la sfârșitul vectorului. Se verifică în prealabil dacă dimensiunea vectorului mai permite adăugarea unui element. Se incrementează **tail**. Complexitate: O(1)
- *E dequeue()* - "șterge" și întoarce primul element din vector. Se incrementează **head**. Complexitate: O(1)
- *E peek()* - întoarce primul element din vector. Complexitate: O(1)
- *bool isEmpty()* - *true* dacă vectorul nu conține niciun element, *false* în caz contrar. Complexitate: O(1)

Reprezentare internă cu vector circular

La nivel de implementare, coada este reprezentată printr-o clasă template ce folosește (pe lângă operațiile ce pot fi efectuate asupra ei) un vector de stocare (*queueArray*) de o dimensiune maximă specificată ca al doilea argument al template-ului (*N*), doi indici ce indică începutul (*head*) şi sfârşitul cozii (*tail*). De asemenea, se reţine şi dimensiunea curentă a cozii (*size*) pentru a putea spune când aceasta este plină sau vidă.

- *void enqueue(E element)* - adaugă noul element la sfârșitul vectorului. Se incrementează modulo (dimensiune container) indicele **tail**. Complexitate: O(1)
- *E dequeue()* - șterge și întoarce primul element din vector. Se incrementează modulo (dimensiune container) indicele **head**. Complexitate: O(1)
- *E peek()* - întoarce primul element din vector. Complexitate: O(1)
- *bool isEmpty()* - la fel ca în celelalte cazuri. Complexitate: O(1)
- *bool isFull()* - se verifică dacă următorul index după **tail** (circular) este egal cu **head**. Dacă da, returnează *true*, returnează *false* în caz contrar. Complexitate: O(1)

Queue.h

```
template <typename T, int N>
class Queue {
private:
    int head;
    int tail;
    int size;
    T queueArray[N];

public:
    // Constructor
    Queue() {
        // TODO
    }

    // Destructor
    ~Queue() {
        // TODO
    }

    // Adauga in coada
    void enqueue(T e) {
        // TODO
```

```
    }

    // Extrage din coada
    void dequeue() {
        // TODO
    }

    // Afla primul element
    T front() {
        // TODO
    }

    bool isEmpty() {
        // TODO
    }
};
```

Alte tipuri de coadă

Double ended queue - Dequeue

Într-o coadă obișnuită accesul la elemente este de tip FIFO - elementele sunt introduse pe la un capăt și scoase la celălalt capăt. În cazul unei Dequeue, se permit ambele operații, la ambele capete. Astfel, în capătul **head** se pot atât introduce, cât și extrage elemente. La fel și în cazul capătului **tail**. Se observă că cele două structuri prezentate în acest laborator (stiva și coada) sunt particularizări ale structurii de date Dequeue. Dintre cele 4 operații de adaugare/ștergere puse la dispoziție de o dequeue, atât stiva cât și coada folosesc doar 2 (addFront() și removeFront() în cazul stivei, respectiv addRear() și removeFront() în cazul cozii).

Priority Queue

Este o coadă în care un dequeue() / peek() va întoarce primul element din acea coadă în funcție de un anumit criteriu. Exemplu: pentru o coadă cu priorități care organizează elementele în funcție de valoarea maximă, un peek() va întoarce valoarea maximă stocată. Similar, în cazul unei cozi cu priorități de minim, peek() va întoarce valoarea minimă stocată.

Exemple de utilizare

Radix Sort

Radix Sort este un algoritm de sortare care ţine cont de cifre individuale ale elementelor sortate. Aceste elemente pot fi nu doar numere, ci orice altceva ce se poate reprezenta prin întregi. Majoritatea calculatoarelor digitale reprezintă datele în memorie sub formă de numere binare, astfel că procesarea cifrelor din această reprezentare se dovedeşte a fi cea mai convenabilă. Există două tipuri de astfel de sortare: LSD (least significant digit) şi MSD (most significant digit). LSD procesează reprezentările dinspre cea mai puţin semnificativă cifră spre cea mai semnificativă, iar MSD invers.

O versiune simplă a radix sort este cea care foloseşte 10 cozi (câte una pentru fiecare cifră de la 0 la 9). Aceste cozi vor reţine la fiecare pas numerele care au cifra corespunzătoare rangului curent. După această împărţire, elementele se scot din cozi în ordinea crescătoare a indicelui cozii (de la 0 la 9), şi se reţin într-un vector (care devine noua secvenţă de sortat). Exemplu:

Secvenţa iniţială:

```
170, 45, 75, 90, 2, 24, 802, 66
```

Numere sunt introduse în 10 cozi (într-un vector de 10 cozi), în funcţie de cifrele de la dreapta la stânga fiecărui număr.

Cozile pentru prima iteraţie vor fi:

```
  * 0: 170, 090
  * 1: nimic
  * 2: 002, 802
  * 3: nimic
  * 4: 024
  * 5: 045, 075
  * 6: 066
  * 7 - 9: nimic
```

a. Se face dequeue pe toate cozile, în ordinea crescătoare a indexului cozii, şi se pun numerele într-un vector, în ordinea astfel obţinută:

Noua secvenţă de sortat:

```
170, 090, 002, 802, 024, 045, 075, 066
```

b. A doua iteraţie:

Cozi:

```
  * 0: 002, 802
  * 1: nimic
  * 2: 024
  * 3: nimic
  * 4: 045
  * 5: nimic
  * 6: 066
  * 7: 170, 075
  * 8: nimic
  * 9: 090
```

Noua secvenţă:

```
002, 802, 024, 045, 066, 170, 075, 090
```

c. A treia iteraţie:

Cozi:

```
  * 0: 002, 024, 045, 066, 075, 090
  * 1: 170
  * 2 - 7: nimic
  * 8: 802
  * 9: nimic
```

Noua secvenţă:

```
002, 024, 045, 066, 075, 090, 170, 802 (sortată)
```

Schelet

Schelet

Exerciţii

Fiecare laborator va avea unul sau doua exerciții publice si un pool de subiecte ascunse, din care asistentul poate alege cum se formeaza celelalte puncte ale laboratorului.

Atenție! Pentru acest laborator asistentul poate înlocui exercițiile publice complet.

1) [**3p**] Implementare stivă.

2) [**3p**] Implementare coadă.

3) [**2p**] Determinați dacă un șir format din caracterele (, [, {,),], } este corect parantezat.

4) [**2p**] Implementaţi Radix Sort (sortare crescătoare).

5) [**2p**] Implementaţi problema Turnurilor din Hanoi folosind explicit stive. Jocul este format din trei tije și un număr variabil de discuri, de diferite mărimi, care pot fi poziționate pe oricare din cele trei tije. Jocul începe având discurile așezate în stivă pe prima tijă, în ordinea mărimii lor, astfel încât să formeze un turn. Scopul jocului este acela de a muta întreaga stivă de pe o tijă pe alta, respectând următoarele reguli:

- doar un singur disc poate fi mutat, la un moment dat.
- fiecare mutare constă în luarea celui mai de sus disc de pe o tija și glisarea lui pe o altă tijă, chiar și deasupra altor discuri care sunt deja prezente pe acea tijă.
- un disc mai mare nu poate fi poziționat deasupra unui disc mai mic.

6) [**2p**] Realizaţi conversia unui număr din baza 10 în baza 2, folosind stive.

7) [**2p**] Verificaţi dacă un număr este palindrom folosind un deque.

8) [**2p**] Implementaţi găsirea celui mai dens interval de puncte unidimensionale.

9) [**2p**] Inversaţi ordinea elementelor într-o coadă. Exemplu: dacă în coadă se află iniţial elementele [4 7 10], după inversare, coada va conţine [10 7 4].

10) [**2p**] Evaluare expresie aritmetică.

11) [**2p**] Primul caracter care nu se repetă într-un stream de caractere.

12) [**2p**] Implementaţi o coadă folosind două stive.

Observaţie Puteţi aborda problema în două feluri:

- unul în care metoda **enqueue** este mai costisitoare
- unul în care metoda **dequeue** este mai costisitoare

13) [**2p**] Implementaţi o stivă folosind două cozi.

Observaţie: Puteţi aborda problema în două feluri:

- unul în care metoda **push** este mai costisitoare
- unul în care metoda **pop** este mai costisitoare

Interviu

Această secțiune nu este punctată și încearcă să vă facă o oarecare idee a tipurilor de întrebări pe care le puteți întâlni la un job interview (internship, part-time, full-time, etc.) din materia prezentată în cadrul laboratorului.

1. Implementați, folosind un singur vector, 3 stive
2. Scrieți un program cu ajutorul căruia să sortați o stivă. Nu aveți acces decât la operațiile push(), pop(), top() și isEmpty()

3. Adăugaţi structurii de stivă o nouă funcţie numită min, funcţie care returnează cel mai mic element. Push, pop şi min trebuie să aibă complexitate O(1)
4. Descrieți cum este folosită stiva sistemului în cazul transmiterii parametrilor apelului unei funcții
5. Implementaţi operaţiile elementare ale unei stive folosind două cozi . Problema admite două versiuni: una în care operaţia **pop** este eficientă, iar cealaltă în care operaţia **push** este eficientă.
6. Implementaţi operaţiile elementare ale unei cozi folosind două stive.
7. Presupunând că avem o coadă ce conţine un număr mare de elemente, coada neputând fi ţinută în memorie. Prezentaţi o modalitate de a implementa operaţiile **enqueue** şi **dequeue**.
8. Să se implementeze o coadă ce are şi operaţia **findmax**, pe lângă operaţiile **enqueue** şi **dequeue**. **Findmax** trebuie să returneze cea mai mare valoare aflată în coadă la momentul respectiv. Oferiţi o implementare eficientă.
9. Cum s-ar implementa o stivă folosind o coadă de priorităţi?
10. De câte cozi este nevoie ca să se poată implementa o coadă de priorităţi?

Bibliografie

1. C++ Reference [http://www.cplusplus.com]
2. Array Stack Visualization [http://www.cs.usfca.edu/~galles/visualization/StackArray.html]
3. Linked List Stack Visualziation [http://www.cs.usfca.edu/~galles/visualization/StackLL.html]
4. Array Queue Visualization [http://www.cs.usfca.edu/~galles/visualization/QueueArray.html]
5. Linked List Queue Visualization [http://www.cs.usfca.edu/~galles/visualization/QueueLL.html]
6. Queue STL Implementation [http://www.cplusplus.com/reference/stl/queue/]
7. Priority Queue STL Implementation [http://www.cplusplus.com/reference/stl/priority_queue/]
8. CLRS - *Introduction to Algorithms, 3rd edition*, capitol 10.1 - Stacks and queues

sd-ca/2018/laboratoare/lab-04.txt · Last modified: 2019/02/01 13:23 by teodora.serbanescu