Laborator 04 - Cozi

Responsabili:

- Claudia Cârdei
- Alex Fărcășanu

Obiective

În urma parcurgerii acestui laborator studentul va fi capabil să:

- înțeleagă principiul de funcționare al unei cozi
- implementeze o coadă folosind un vector pentru stocarea elementelor
- implementeze algoritmul de sortare Radix Sort

Ce este o coadă?

O coadă este o structură de date ce modelează un buffer de tip FIFO (First In, First Out). Astfel, primul element introdus în coadă va fi și primul care va fi scos din coadă.

Operații:

- *enqueue* – adaugă un element (entitate) în coadă. Adăugarea se poate face doar la sfârșitul cozii.
- *dequeue* – șterge un element din coadă și îl returnează. Ștergerea se poate face doar la începutul cozii.
- *front* – consultă (întoarce) elementul din capul cozii fără a efectua nicio modificare asupra acesteia.
- *isEmpty* – întoarce *1* dacă coada este goală; *0* dacă are cel puțin un element

Dequeue

Dequeue (sau coadă cu dublu access) este o structură de tip coadă în care însă accesul (introducere / extragere de elemente) se poate realiza "prin ambele capete". De cele mai multe ori sunt implementate folosind liste dublu înlănţuite (vor fi studiate în cadrul laboratorului 5). Dintr-un anume punct de vedere, se poate considera că atât stiva cât şi coada clasică sunt specializări ale tipului abstract **dequeue** întrucât ambele se pot implementa folosind dequeue (şi restrângând operaţiile ce se realizează asupra sa).

Priority queue

Coada prioritară reprezintă un tip de coadă în care fiecare element are asociată o anume prioritate. În aceste condiţii, operaţiile de bază asupra cozii devin:

- *enqueue* - adaugă la coadă un element cu prioritatea specificată
- *dequeue* - extrage elementul cu cea mai mare prioritate
- *front* - examinează elementul cu cea mai mare prioritate fără a-l extrage din coadă

Cum se implementează?

La fel ca și stivele, cozile se pot implementa cu ajutorul unui **vector** sau cu **liste înlănțuite**.

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

Search

- Reguli generale și de notare
- Catalog
- Concursuri
- Calendar

Laboratoare

- Laborator 1 - Introducere in C++
- Laborator 2 - Noțiuni de C++
- Laborator 3 - Stive
- Laborator 4 - Cozi
- Laborator 5 - Liste generice
- Laborator 6 - HashTable
- Laborator 7 - Grafuri
- Laborator 8 - Arbori Binari
- Laborator 9 - Arbori Binari de Căutare
- Laborator 10 - Heap-uri
- Laborator 11 - Treap-uri
- Laborator 12 - Mulțimi Disjuncte

Teme

- Tema 1
- Tema 2
- Tema 3
- Tema 4

Resurse

- Debugging
- Data Structure Visualization

Table of Contents

- Laborator 04 - Cozi
  - Obiective
  - Ce este o coadă?
    - Dequeue
    - Priority queue
    - Cum se implementeaz
    - Implementare cu vector
    - Implementare cu vector circular
  - Radix Sort
  - Exerciţii
  - Interviu
  - Resurse

Implementarea 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.

Implementarea 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.

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 la coada
 void enqueue(T e) {
  // TODO
 }

 // Extrage din coada
 T dequeue() {
  // TODO:
 }

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

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

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)
```

Exerciţii

1) [**3p**] Pornind de la header-ul definit anterior realizaţi implementarea structurii de date *coadă*.

- constructor și destructor
- [**1p**] metoda enqueue
- [**1p**] metoda dequeue
- [**0.5p**] metoda front
- [**0.5p**] metoda isEmpty
-

2) [**3p**] Implementaţi Radix Sort (sortare descrescătoare).

3) [**2p**] Verificaţi dacă un număr este palindrom folosind o stivă şi o coadă.

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. 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.
2. Implementaţi operaţiile elementare ale unei cozi folosind două stive.
3. 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**.
4. 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.
5. Cum s-ar implementa o stivă folosind o coadă de priorităţi?
6. De câte cozi este nevoie ca să se poată implementa o coadă de priorităţi?

Resurse

[1] - Array Queue Visualization

[2] - Linked List Queue Visualization

[3] - Queue STL Implementation

[4] - Priority Queue STL Implementation