MONOIDUL COZILOR ÎN RAPORT CU OPERAȚIA DE CONCATENARE

Lucia Bitcovschi
Universitatea de Stat din Moldova
bitco@mail.md

Abstract. *The purpose of this paper ist o investigate the tails in terms of algebraic structures. Will be considered the most important operations such as concatenation of queues, inserting an item in the queue. Queue elements are considered as abstract data types, which have contributed to their implementation through template classes.*

Cuvinte-cheie: *coadă, clasă template, concatenare.*

I. Introducere

În ceea ce urmează vom încerca să studiem mulțimea cozilor cu diferite operații din punctul de vedere al structurilor algebrice. Vor fi examinate cele mai importante operații asupra cozii, cum sunt: inserarea unui element în coadă, concatenarea a două cozi.

Vom cerceta pentru care dintre aceste operații o mulțime de cozi formează *monoid*.

Elementele cozii reprezintă un tip abstract de date, pe care-l vom nota prin *T*. Acest tip va servi drept parametru la realizarea cozii prin clase *template*.

Coada, ca şi stiva, este o formă de listă cu acces restrictiv. Spre deosebire de stivă, la care adăugarea şi ştergerea elementelor se realizau pe la acelaşi capăt al listei, în cazul cozii toate inserările se fac la un capăt al cozii, iar toate ştergerile la celălalt capăt.

Conceptul de coadă apare în orice sistem în care obiectele sunt servite în ordinea sosirii. Cele două capete ale cozii îşi iau numele din această paralelă cu rândul de aşteptare. Mai precis, capătul de la care sunt eliminate elementele se numeşte *cap* (în engleză – *head*), iar capătul la care se adaugă elementele se numeşte *coadă* (în engleză – *tail*).

Pentru a evita confuziile, vom folosi, în acest scop, termenii de *început* şi *sfârşit*. Aşa cum am mai arătat, terminologia evoluează în timp.

Definiția clasică a cozii este aceea a unei structuri stricte de tip ***FIFO***, dar termenul este folosit şi pentru alte structuri, în care elementele sunt eliminate pe la un singur capăt al şirului, indiferent de momentul inserării. Această definiție mai relaxată include structuri în care noile elemente pot fi inserate în fața altora, pentru a se păstra ordinea alfabetică sau pentru a se acorda anumitor elemente o prioritate mai mare. În aceşti termeni, şi stiva este considerată un caz particular de coadă.

II. Operațiile pentru coadă

Definiția 1. *Coada este o structură de date (SD), pentru care se cunoaşte poziția elementului care se adaugă în coadă şi poziția elementului care se scoate din coadă. Coada este o structură de date de tip* ***FIFO*** *(First In, First Out), elementul care este extras din coadă fiind întotdeauna cel care a stat mai mult în coadă* [1].

Ordinea de extragere din coadă este similară cu ordinea de introducere în coadă, ceea ce face utilă o coadă în aplicațiile unde ordinea de servire este aceeaşi cu ordinea de sosire: procese de tip „vânzător – client" sau „producător – consumator". În astfel de situații, coada de aşteptare este necesară pentru a acoperi o diferență temporară între ritmul de servire şi ritmul de sosire, deci pentru a memora temporar cereri de servire (mesaje) care nu pot fi deocamdată prelucrate.

Operațiile asociate tipului abstract de date *coadă* sunt:

- **enqueue (ob):** adaugă un obiect la sfârşitul cozii (*baza*);.
- **dequeue ():** elimină obiectul din fața cozii (*cap*);
- **front ():** returnează o referință către *cap*, fără a elimina obiectul;
- **back ():** returnează o referință către *baza*, fără a elimina obiectul;
- **size ():** returnează numărul obiectelor din coadă;
- **isEmpty ():** returnează **true** dacă coada este vidă, altfel **false**;
- Concatenarea a două cozi.

Ca şi alte liste abstracte, cozile pot fi realizate ca vectori sau ca liste înlănțuite, cu condiția suplimentară că durata operațiilor **enqueue (ob)** şi **dequeue ()** să fie minimă (O(1)).

O coadă înlănțuită poate fi definită prin:

Adresa de început a cozii, iar pentru adăugare să se parcurgă toată coada pentru a găsi ultimul element (durata operației **enqueue(ob)** va fi O(n)).

Adresele primului şi ultimului element, pentru a elimina timpul de parcurgere a cozii la adăugare.

Adresa ultimului element, care conține adresa primului element (coadă circulară).

Descrierea operațiilor pentru coadă

În cele ce urmează vom analiza modul în care putem implementa o structură de tip coada în memoria calculatorului.

Dorim să efectuăm operații la ambele extremități ale structurii, aşa că va trebui ca în loc de un *pointer* să folosim doi: un *pointer* de început şi un *pointer* de sfârşit. La început coada este goală, deci ambii *pointeri* trebuie să indice către aceeaşi locație de memorie.

Coada este un *container* de obiecte, acest lucru înseamnă că ar trebui că putem declara cozi de orice tip*: int, char, bool,* etc. sau un tip definit de utilizator. Deci vom implementa o clasă **generică** *Coada*, cu parametrul generic **T**.

```
template<class T>
class Coada
{
public:
    Coada(); // Constructor
    ~Coada(); // Destructor
    void enqueue(const T& ob); // Adaug un element in coada
    void dequeue(); // Elimin obiectul din fata
    const T& front() const; // Returnez o referinta catre cap
    const T& back() const; // Returnez o referinta catre baza
    int size() const; // Returnez numarul de elemente
    isEmpty() const; // Este coada vida?
private:
    struct Element
    {
        T data;
        Element * next;
    };
    Element * cap;
    Element * baza;
```

```
    int count; // Numarul elementelor
};
```

Observăm că la unele funcții apare referința constantă. Pur şi simplu, se returnează un obiect **T** şi nu o referință datorită eficienței. Atunci când se returnează, o referință nu se mai alocă, temporar, spațiu pentru obiectul returnat sau pentru parametrul formal, deoarece pot accesa direct variabila referită. Referința este constantă tocmai pentru că mai vrem să modificăm din greşeală argumentul sau obiectul returnat, în plus că nu am putea transmite valori concrete (de ex. 1, 2, 'c', 4.6, etc.) funcției **enqueue ()**, dacă referința nu ar fi constantă.

Alocarea înlănțuită se aplică deosebit de convenabil în cazul cozilor. În acest caz, este uşor de observat că referințele se vor deplasa din partea din față spre spatele cozii, astfel încât când este extras un nod din față, noul nod de început este specificat în mod direct. Vom folosi pointerul *cap* şi *baza* pentru fața şi, respectiv, spatele cozii.

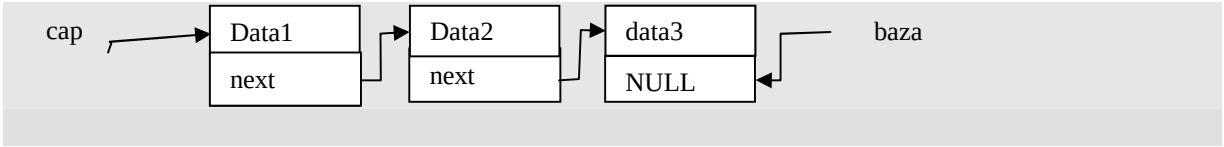

Fig. 1. O coadă implementată ca listă simplă

Inițializarea cozii

Inițializarea cozii presupune crearea cozii vide, adică elementele **cap** şi **baza** indică **NULL**. Coada este inițializată în constructor:

```
template<class T> Coada<T>::Coada()
{
      cap = baza = NULL; // coada vida
     count = 0; // nici un element
}
```

Coada vidă

Coada este vidă atunci când **cap** este **NULL**, deci algoritmul va fi următorul:

```
template<class T>  Coada<T>::isEmpty() const
{
    return cap == NULL;
}
```

Calificatorul **const** informează compilatorul că respectiva funcție nu modifică obiectele cozii, nu modifică datele membre.

Adăugarea unui element în coadă

Algoritmul este simplu. Alocăm memorie pentru un obiect de tip **Element,** scriu informația în obiect. Succesorul acestui obiect este **baza.** Noul element devine **baza**. Se incrementează **count**. Funcția **enqueue()** va implementa această operație.

```
template<class T> void Coada<T>::enqueue(const T& ob)
{
    if(isEmpty()) // Daca coada este vida
    {
        cap = new Element;
        cap->data = ob;
        cap->next = NULL; // Fiind singurul element, succesorul
      //este NULL
```

```
        baza = cap;
        count = 1;
    }
    else
    {
          Element * p = new Element;
        p->data = ob;
        p->next = NULL; // Devine noul element baza
        baza->next = p; // Fosta baza se leaga de noua baza
        baza = p; // p devine baza
        ++count;  // S-a mai adaugat un element
    }
}
```

Grafic coada va arăta astfel:

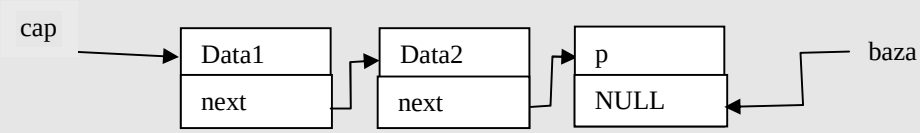

Fig. 2. Coada după inserarea elementului p

Eliminarea unui element din coadă

Se poate elimina numai elementul din față. Se salvează adresa capului -> Elementul următor devine cap -> Se eliberează memoria fostului cap. Se decrementează **count**. Funcția **dequeue()** va implementa această operație.

```
template<class T> void Coada<T>::dequeue()
{
    if(isEmpty()) cout<< "Eroare! Coada Vida!";
    Element * q = cap; // Salvez elementul din cap
    cap = cap->next; // Elementul urmator devine cap
    delete q;
    --count;
}
```

Operația de ştergere în cazul cozilor se deduce în mod similar. Dacă fig.2 reprezintă situația înainte de ştergere, situația după aceasta va fi.

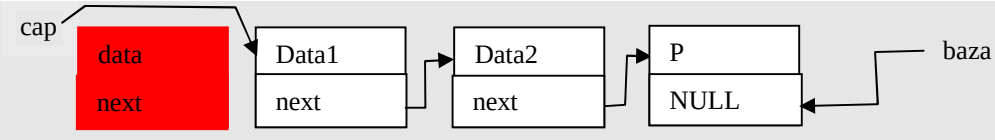

Fig 3. Coada după ştergerea primului element

Implementarea unei cozi printr-un vector circular (numit şi buffer circular) limitează numărul maxim de valori ce pot fi memorate temporar în coadă.

Concatenarea cozilor

Concatenarea este operația prin care din două cozi C_1 şi C_2 cu elemente de acelaşi tip se obține a treia coadă formată astfel: la coada C_1 sunt alipite elementele cozii C_2. Concatenarea cozii C_1 cu coada C_2, presupune că pointerul *next* la baza cozii C_1, trebuie să-şi schimbe valoarea din NULL cu adresa din capul cozii C_2 (a se vedea Fig. 4).

```
template <class T>
 void Coada<T>::concatcoada(coada * cap1,coada * cap2)
   {
      Element * p; p=cap1.cap->next;
      for(p=cap1; p->urm!=NULL; p=p->urm)
      {
        p->next=cap2.cap->next;
        cap2.cap->next=NULL;
      }
}
```

Grafic coada va arăta astfel:

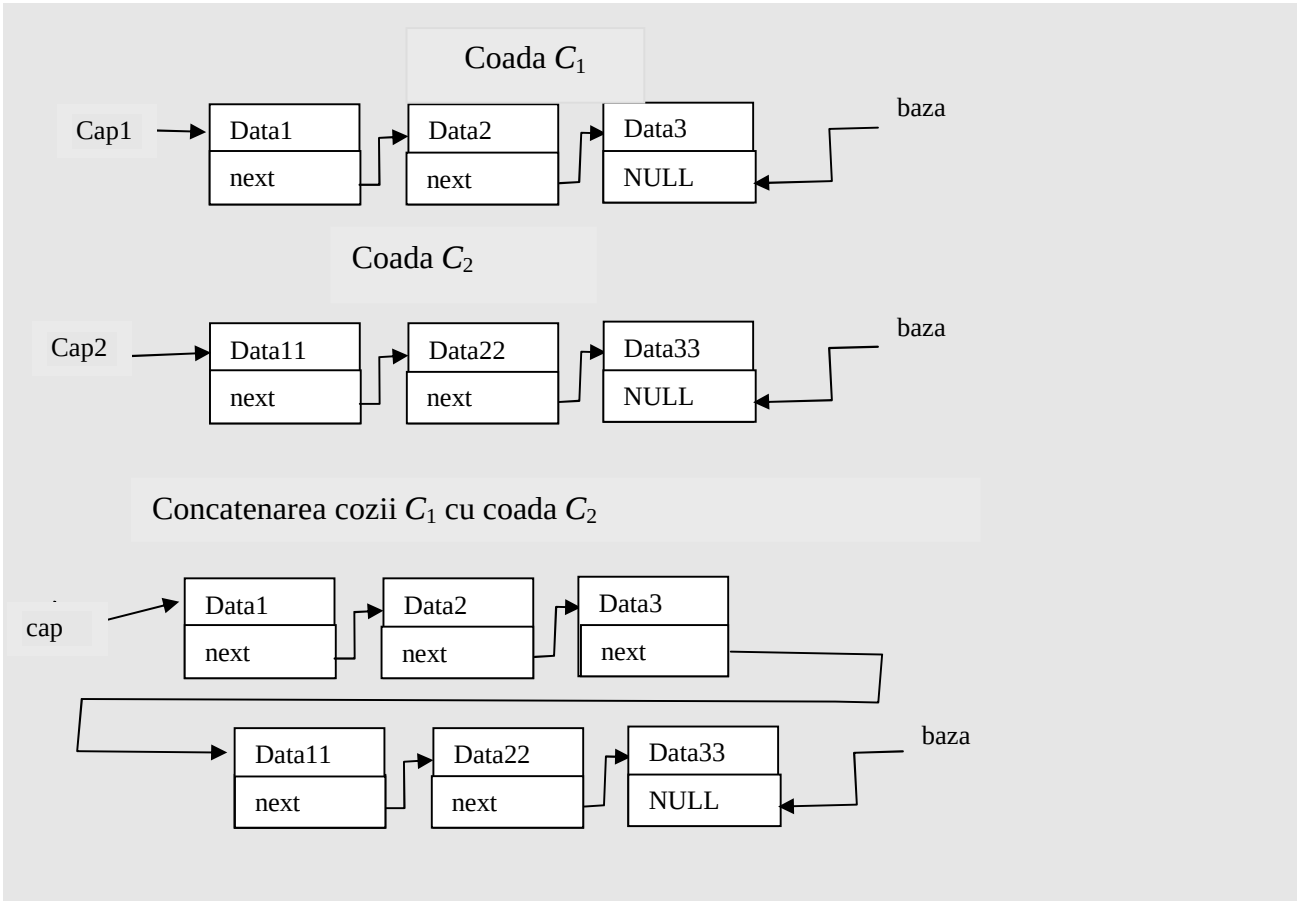

Fig 4. Concatenarea a două cozi

III. Structura algebrică a cozilor

Fie $E=\{e_1,e_2,...,e_n\}$ o mulțime de elemente de natură arbitrară de unul și același tip.

Definiția 2. *Prin coadă asupra mulțimii* $E=\{e_1,e_2,...,e_n\}$, *notată* $C(e_{i_1},e_{i_2},...,e_{i_k})$, *vom înțelege mulțimea ordonată a elementelor* $\{e_{i_1},e_{i_2},...,e_{i_k}\}$, *unde* $e_{i_j}\in E$, $j=1,2,...,k$, *precum* k – ***este numărul de elemente în coada*** $C(e_{i_1},e_{i_2},...,e_{i_k})$, *iar elementul* e_{i_k} – *se numește* ***ultimul element al cozii***.

Notă. Coada poate fi și vidă, dacă nu are nici un element. Vom nota coadă vidă prin „$\varnothing$", adică C() = $\varnothing$. Prin $C(e)$ vom nota coada dintr-un singur element $e\in E$.

Notăm prin $C=\{C_1,C_2,\ldots,C_m,\ldots\}$ – mulțimea tuturor cozilor asupra mulțimii E. Introducem un set de operații pentru coadă.

Inserarea unui element în coadă

Vom introduce operația care ne va da posibilitatea de a înscrie (adăuga) elemente noi în coadă pe care vom numi-o „*adăugare*" și o vom nota prin semnul "+".

Definiția 3. *Operația "+", care adăugă în coadă* $C(e_{i_1},e_{i_2},...,e_{i_k})$ *elementul* $e_{i_{k+1}}$, *este operația care preface coada* $C(e_{i_1},e_{i_2},...,e_{i_k})$ *în coada* $C(e_{i_1},e_{i_2},...,e_{i_k},e_{i_{k+1}})$ *astfel încât elementul* $e_{i_{k+1}}$ *devine ultimul element al cozii, adică* $C(e_{i_1},e_{i_2},...,e_{i_k})+e_{i_{k+1}}=C(e_{i_1},e_{i_2},...,e_{i_k},e_{i_{k+1}})$.

Notă. Deoarece un singur element $e_{i_j}\in E$ îl putem considera ca o coadă dintr-un singur element $C(e_{i_j})$, vom scrie că $C(e_{i_1},e_{i_2},...,e_{i_k})+e_{i_{k+1}}=C(e_{i_1},e_{i_2},...,e_{i_k})+C(e_{i_{k+1}})=C(e_{i_1},e_{i_2},...,e_{i_k},e_{i_{k+1}})$.

Definiția 4. Un cuplu (M,O) format dintr-o mulțime nevidă M și o operație algebrică "O", definită pe M, se numește ***grupoid*** [3].

Extindem operația „*adăugare*" pentru mai multe elemente, care ne va da posibilitatea să concatenăm cozi.

Definiția 5. *Operația "+", care concatenează coada* $C_i=C(e_{i_1},e_{i_2},...,e_{i_k})$ *cu coada* $C_j=C(e_{i_{k+1}},e_{i_{k+2}},...,e_{i_{k+m}})$, *este operația* $(...((C(e_{i_1},e_{i_2},...,e_{i_k})+e_{i_{k+1}})+e_{i_{k+2}})+...)+e_{i_{k+m}}$ *care are ca rezultat coada* $C(e_{i_1},e_{i_2},...,e_{i_k},e_{i_{k+1}},e_{i_{k+2}},...,e_{i_{k+m}})$, *astfel încât elementul* $e_{i_{k+m}}$ *este ultimul element al cozii* C_i+C_J, *pentru orice* $C_i, C_j\in C$.

Astfel mulțimea tuturor cozilor asupra mulțimii E cu operația extinsă „*adăugare*" reprezintă un grupoid notat $(C,+)$.

Definiția 6. *Vom spune că operația algebrică " O ", definită pe mulțimea M, posedă element neutru, dacă există un element* $\mathbf{l}\in M$, *astfel încât pentru orice element* $e\in M$ $eO\mathbf{l}=\mathbf{l}Oe=e$ [3].

Notă. Elementul neutru se notează de obicei cu 0 și se numește *element zero.*

Teorema 1. *Grupoidul* $(C,+)$ *posedă element neutru.*

Demonstrație:

În calitate de element neutru se va lua coada vidă $\varnothing$. Luăm orice coadă arbitrară $C_i\in X$. Fie $C_i=C(e_{i_1},e_{i_2},...,e_{i_k})$, atunci $C_i+\varnothing=C(e_{i_1},e_{i_2},...,e_{i_k})+\varnothing=\varnothing+C(e_{i_1},e_{i_2},...,e_{i_k})=C(e_{i_1},e_{i_2},...,e_{i_k})=C_i$. Teorema este demonstrată.

Teorema 2. *Grupoidul* $(C,+)$ *este semigrup.*

Demonstrație:

Fie C_i, C_j, C_r trei cozi arbitrare din *C*. Demonstrăm că $(C_i + C_j) + C_r = C_i + (C_i + C_r)$.

Fie $C_i = C(e_{i_1}, e_{i_2}, ..., e_{i_{k_i}})$, $C_j = C(e_{j_1}, e_{j_2}, ..., e_{j_{k_i}})$, $C_r = C(e_{r_1}, e_{r_2}, ..., e_{r_{k_i}})$. Atunci

$$(C_i + C_j) + C_r = (C(e_{i_1}, e_{i_2}, ..., e_{i_{k_i}}) + C(e_{j_1}, e_{j_2}, ..., e_{j_{k_i}})) + C(e_{r_1}, e_{r_2}, ..., e_{r_{k_i}}) = C(e_{i_1}, e_{i_2}, ..., e_{i_{k_i}}) +$$

$$+ (C(e_{j_1}, e_{j_2}, ..., e_{j_{k_i}}) + C(e_{r_1}, e_{r_2}, ..., e_{r_{k_i}})) = C_i + (C_j + C_r).$$

pentru orice $C_i, C_j, C_r \in X$. Teorema este demonstrată.

Teorema 3. *Grupoidul* (*C*,+) *este monoid.*

Demonstrație:

Din teorema 1 şi teorema 2 rezultă că grupoidul (*C*,+) este semigrup. Trebuie să demonstrăm că grupoidul (*C*,+) este monoid.

Fie o coadă arbitrară $C_i \in C$. Demonstrăm că $C_i + \varnothing = \varnothing + C_i = C_i$. Fie $C_i = C(e_{i_1}, e_{i_2}, ..., e_{i_k})$, atunci $C_i + \varnothing = C(e_{i_1}, e_{i_2}, ..., e_{i_k}) + \varnothing = \varnothing + C(e_{i_1}, e_{i_2}, ..., e_{i_k}) = C(e_{i_1}, e_{i_2}, ..., e_{i_k}) = C_i$.

Deci, grupoidul (*C*,+) este monoid.

IV. Concluzii

În articolul prezentat a fost descrisă în detalii una dintre cele mai importante şi utilizate structuri dinamice de date, – coada înlănțuită. Au fost introduse definițiile formalizate pentru structura dată şi noțiunile legate de ea. Au fost cercetate proprietățile algebrice ale cozilor şi ale operațiilor asupra lor. S-a demonstrat că totalitatea cozilor cu elementele de unul şi acelaşi tip *T*, dotată cu operația de concatenare este monoid. A fost propusă realizarea în limbajul C++ a unei clase generice parametrizate cu clasa abstractă arbitrară *T*, care permite modelarea lucrului cu cozile din punctul de vedere al structurilor algebrice.

V. Referințe:

1. Knuth D.E. Tratat de programare a calculatoarelor. - Bucureşti: Editura Tehnică, 1974. – 676 p.
2. Zaharia M.D. Structuri de date şi algoritmi, exemple în limbajele C şi C++.– Cluj-Napoca, 2002. – 252 p.
3. Goian I., Sârbu P., Topală A. Grupuri şi inele. – Chişinău: CEP USM, 2005. - 244 p.
4. Bitcovschi L. Cercetarea listelor ca structuri algebrice. – Materialele Conferinței ştiințifice internaționale: „Modelarea matematică, Optimizare şi Tehnologii informaționale". – Chişinău: 24-26 martie 2010, p. 384-391.