3 Stiva. Coada. Liste Dublu Inlantuite

3.1 Obiective

Scopul acestei sesiuni de laborator este de a ne familiariza cu implementarea operatiilor pe tipurile de date abstracta stiva si coada, respectiv cu implementarea operatiilor pe liste dublu inlantuite.

3.2 Notiuni teoretice

3.2.1 Stive

Stiva este o lista cu o politica de acces speciala: adaugarea sau stergerea unui element se face la un singur capat al listei, numit **varful stivei**. Elementul introdus primul in stiva poarta numele de **baza stivei**. Stiva se poate asemana unui vraf de farfurii asezat pe o masa: modalitatea cea mai comoda de a pune o farfurie fiind in varful stivei, si tot de aici e cel mai simplu sa se ia o farfurie.

Datorita locului unde se actioneaza asupra stivei, aceste structuri se mai numesc structuri de tip **LIFO (Last In First Out)**, adica *ultimul venit - primul iesit*. Modelul unei **stive** implementat prin strategia inlatuita este dat de Figura 3.1, iar modelul de implementare prin vector este schitat in Figura 3.2.

Principalele operatii pe o stiva sunt urmatoarele:

push – adaugarea unui element in varful stivei ($insert\_first$ – lista inlantuita, sau $insert\_last$ – vector);
pop – stergerea unui element din varful stivei ($delete\_first$ – lista inlantuita, sau $delete\_last$ – vector); operatia poate sa si returneze elementul sters (i.e. nu il sterge fizic, doar il elimina din structura)

Pe langa aceste operatii, se mai poate specifica o operatie de initializare a stivei, respectiv una care doar returneaza primul element din stiva, fara a-l sterge (de regula denumita *top*).

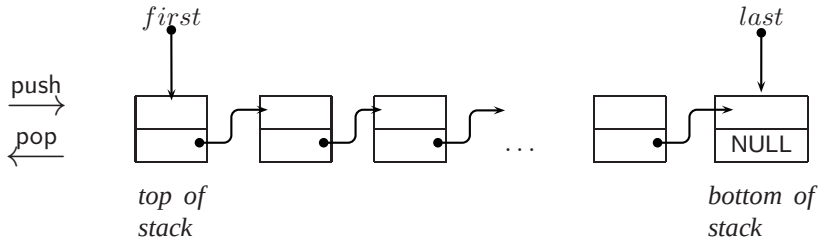

Figure 3.1: Modelul unei stive, implementata prin lista simplu inlantuita.

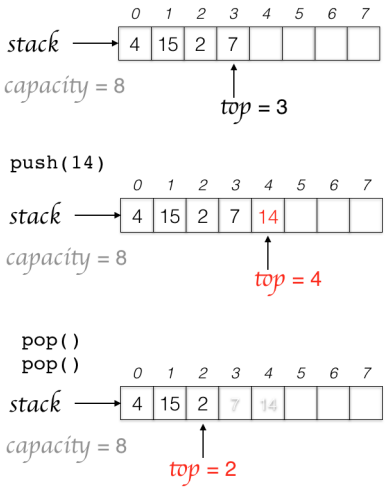

Figure 3.2: Modelul unei stive, implementata prin vector.

Ex. 1 — Implementati operatiile fundamentale pe stiva – **void push(NodeT** stack, int key)** si **NodeT* pop(NodeT** stack)**, utilizand lista simplu inlantuita ca si structura de baza. Totodata, implementati o functie de initializare a stivei, si una de afisare a elementelor acesteia. Testati operatiile implementate.

3.2.2 Cozi

Coada reprezinta o alta categorie speciala de lista, in care elementele se adauga la un capat (sfarsit) si se sterg de la celalalt capat (inceput). Aceasta politica de acces este cunoscuta sub numele de **FIFO (First In First Out)**, adica *primul venit - primul servit*.

Modelul intuitiv al acestei structuri este coada care se formeaza la un magazin: lumea se aseaza la coada la sfarsitul ei, cei care se gasesc la inceputul cozii sunt serviti, parasind apoi coada.

Figura 3.3 prezinta modelul de implementare inlantuita a unei cozi, respectiv Figura 3.4 pe cel de vector. Operatiile principale sunt:

enqueue – introducerea unui element in coada – $insert\_last$;
dequeue – scoaterea unui element din coada – $delete\_first$; operatia poate sa si returneze elementul sters (i.e. nu il sterge fizic, doar il elimina din lista)

Pe langa aceste operatii, se mai poate specifica o operatie de initializare a cozii, respectiv una care doar returneaza primul element din coada, fara a-l sterge (de regula denumita $front$).

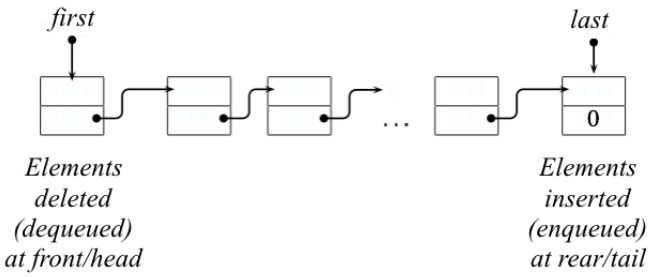

Figure 3.3: Modelul unei cozi, implementata ca si lista simplu inlantuita.

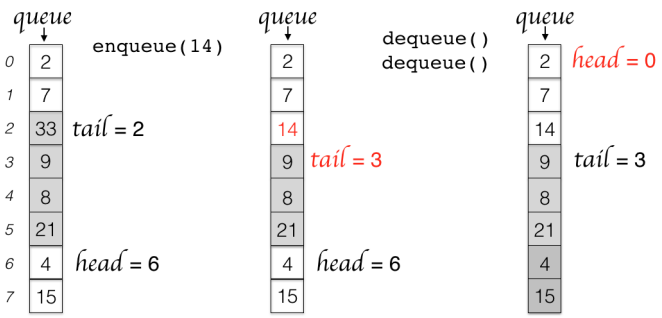

Figure 3.4: Modelul unei cozi implementate secvential (cu vector). $tail$ reprezinta urmatoarea pozitie pentru inserare, respectiv $head$ reprezinta pozitia primului element din coada. Continutul cozii se afla intre $head$ (inclusiv) si $tail$ (exclusiv), interpretate circular. Initial, coada contine elementele 4, 15, 2, 7; dupa enqueue(14): 4, 15, 2, 7, 14; dupa cele doua operatii de dequeue(): 2, 7, 14

Ex. 2 — Implementati operatiile fundamentale pe coada – **void enqueue(struct queue *my_queue, int key)** si **int dequeue(struct queue *my_queue)** (impreuna cu o functie de initializare a cozii si una de afisare a elementelor acesteia), utilizand un **vector** ca si structura de baza.
Utilizati exemplele furnizate in Figura 3.4 si pseudocodul disponibil in Th. Cormen et. al, "Introduction to Algorithms" (pag 235, sect 10.1).
Implementarea voastra ar trebui sa considere cazurile de $overflow$ la inserare (coada este plina, nu se mai poate insera), respectiv $underflow$ la stergere (coada este goala, nu se poate sterge element). Cate elemente poate contine maxim o coada implementata asa, daca vectorul de baza are dimensiune $CAPACITY$? **Sugestie**: Structura $queue$ ar trebui sa contina un vector de dimens $CAPACITY$ (data, constanta), si cei doi indici - $head$, respectiv $tail$.

```
#define CAPACITY  10
typedef struct _queue
{
        int vec[CAPACITY];
        int size;
        int head, tail;
}queue
```

3.2.3 Lista dublu înlănţuită

Lista *dublu înlănţuită* este lista dinamică între nodurile căreia s-a definit o dublă relaţie: de succesor si de predecesor. Modelul unei astfel de liste este dat în figura 3.5.

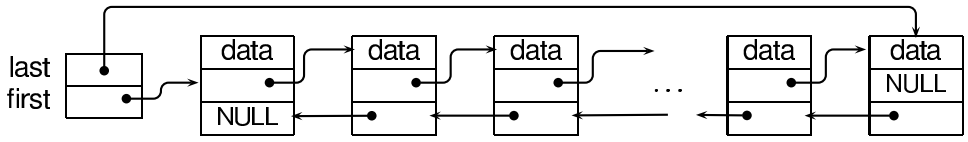

Figure 3.5: Modelul unei liste dublu înlănţuite.

Structura de nod intr-o listă dublu înlănţuită se poate defini astfel:

```
typedef struct node_type
{
  KeyT key; /* optional */
  ValueT value;
  /* pointer to next node */
  struct node_type *next;
  /* pointer to previous node */
  struct node_type *prev;
} NodeDL;
```

In exemplul de mai sus, $KeyT$, respectiv $ValueT$ reprezinta valori generice pentru tipul campurilor key, respectiv $value$. In implementarea efectiva, acestea se vor inlocui cu tipul corespunzator cerintei problemei.
Principalele operaţii cu liste dublu înlăţuite sunt următoarele:

- inserarea unui nod (la inceput, la sfarsit, inainte/dupa un anumit nod);
- ştergerea unui nod (de la inceput, de la sfarsit, nodul avand o anumita cheie),
- cautarea nodului cu o anumita informatie/cheie;

Aditional, mai putem specifica operatii de creare si stergere a intregii liste, operatie de calculare a dimensiunii listei, etc.

3.2.4 Implementarea operaţiilor cu liste dublu înlănţuite

Vom identifica lista este dată de o structura de tip $list\_header$, care contine cei doi pointeri spre inceputul si sfarsitul listei (i.e. incapsulam informatia de inceput/sfarsit a listei intr-o structura noua):

```
/* list header structure */
struct list_header
{
  NodeDL *first;
  NodeDL *last;
};
```

```
/* the list is declared as a list header structure*/
struct list_header L = {NULL, NULL};
```

Aceasta grupare a celor doi pointeri sub o structura noua ne permite accesarea acestora printr-o singura variabila, fiind o modalitate usor mai eleganta de a specifica lista.

Multumita celor doua referinte mentinute la nivelul fiecarui nod, o listă dublu înlănţuită poate fi parcursa in doua directii asa cum arata tabelul 3.1.

Pargurgere secvenţial înainte	Parcurgere secvenţial înapoi

`for ( p = L.first; p != NULL; p = p->next )`
`{`
`  /*perform some operation on current node, p*/`
`}`
	`for ( p = L.last; p != NULL; p = p->prev )`
	`{`
	`  /*perform some operation on current node, p*/`
	`}`

Table 3.1: Parcurgerile pentru o lista dublu inlantuita

Inserarea unui nod într-o listă dublu înlănţuită

Ca si la lista simplu inlantuita, pentru a insera un nod nou in structura, se alocă mai intai spaţiu pentru nodul nou şi se populează câmpurile de date din nod (setam si campurile $prev$ si $next$ la valoarea $NULL$).

Din nou, ca si la lista simplu inlantuita, la operatia de inserare trebuie sa verificam daca lista nu este goala; daca lista este goala, nodul nou va reprezenta atat inceputul, cat si sfarsitul listei:

```
if ( L->first == NULL )
{ /* the list is empty */
  L->first = L->last = p;
}
```

Observatie: Avand in vedere ca o functie de inserare poate modifica valoarea lui L – fie inceputul, fie sfarsitul listei – lista trebuie transmisa prin referinta, prin urmare in corpul functiei accesul la campurile $first$ si $last$ se face fie utilizand *L->*, sau *(*L).* (i.e. *L->first*, sau *(*L).first*).

In functie de pozitia din lista unde dorim sa inseram nodul nou, intalnim situatiile de inserare descrise in tabelul 3.2:

Inaintea primului nod	După ultimul nod	După un nod de cheie dată $afterKey$, presupunând că acesta există şi are adresa q:

`/* the list is not empty */`
`p->next = L->first;`
`L->first->prev = p;`
`L->first = p;`
	`/* the list is not empty */`
	`p->prev = L->last;`
	`L->last->next = p;`
	`L->last = p;`
		`p->prev = q;`
		`p->next = q->next;`
		`if ( q->next != NULL )`
		`  q->next->prev = p;`
		`q->next = p;`
		`if ( L->last == q )`
		`  L->last = p;`

Table 3.2: Inserare inainte si dupa primul nod

Ştergerea unui nod dintr-o listă dublu înlănţuită

Intr-o lista dublu inlantuita se poate sterge primul element, ultimul element, un element care are o cheie data. Tabelul 3.3 prezinta primele 2 cazuri, iar pentru ultimul caz se da pseudocodul.

Ştergerea primului nod:	Ştergerea ultimului nod:

`p = L->first;`
`L->first = L->first->next;`
`free( p ); /* free memory */`
`if ( L->first == NULL )  L->last == NULL;`
`else L->first->prev = NULL;`
	`p = L->last;`
	`L->last = L->last->prev;`
	`if ( L->last == NULL ) L->first = NULL;`
	`else L->last->next = NULL;`
	`free( p );`

Table 3.3: Stergerea primului / ultimului nod din lista dublu inlantuita

La stergerea unui nod precizat prin cheia $givenKey$, presupunem că nodul există şi are adresa p (gasita apeland căutarea dupa cheie). Se da mai jos pseudocodul pentru stergerea nodului p, urmand ca implementarea efectiva sa fie lasata ca si exercitiu:

function DELETE(L, p)
  if $p.prev \neq NIL$ **then**
    $p.prev.next \leftarrow p.next$
  else
    $L.first \leftarrow p.next$
  end if
  if $p.next \neq NIL$ **then**
    $p.next.prev \leftarrow p.prev$
  else
    $L.last \leftarrow p.prev$
  end if
  $free(p)$
end function

Căutarea unui element într-o listă dublu înlănţuită

Căutarea unui nod dupa cheie se face identic ca si pentru o lista simplu înlănţuită (vezi Laborator 2).

Ex. 3 — Implementati urmatoarele functii pentru lista dublu inlantuita:

- Inserare: **void insert_first(struct list_header *L, int givenKey)**, **void insert_last(struct list_header *L, int givenKey)** si **void insert_after_key(struct list_header *L, int afterKey, int givenKey)**.
- Parcurgere: **void print_forward(struct list_header *L)** si **void print_backward(struct list_header *L)** care afiseaza continutul listei dublu inlantuite de la primul la ultimul element, respectiv in ordine inversa.
- Cautare: **NodeDL* search(struct list_header *L, int givenKey)** care cauta in lista $*L$ nodul care are cheia $givenKey$. Functia returneaza adresa nodului, respectiv $NULL$ daca acesta nu exista.
- Stergere: **void delete_first(struct list_header *L)**, **void delete_last(struct list_header *L)** si **delete_key(struct list_header *L, int givenKey)** pentru lista dublu inlantuita.

3.3 Mersul lucrarii

Studiati codul prezentat în laborator si utilizati acest cod pentru rezolvarea exercitiilor obligatorii, prezentate pe parcursul lucrarii. La finalul sesiunii de laborator, este obligatoriu ca fiecare student sa prezinte codul (compilabil, executabil) cerut in exercitiile de pe parcursul lucrarii de laborator.

3.3.1 Probleme

1. Rezolvati exercitiile obligatorii din laborator - marcate cu chenar gri !

2. Implementati operatiile fundamentale pe stiva (**void push(int *stack, int *top, int key)** si (**int pop(int *stack, int *top)**), utilizand un vector ca si structura de baza.

3. Stiva permite inserarea si stergerea la un singur capat, pe cand coada permite inserarea la un capat si stergerea de la celalalt capat. O coada cu doua capete (*en. double-ended queue*) permite inserarea si stergerea de la ambele capete. Implementati cele patru operatii de inserare/stergere pentru o astfel de structura, utilizand un vector ca si structura de baza. Operatiile voastre ar trebui sa aiba complexitate $O(1)$.

4. (*) Să se implementeze operatiile pe o listă dublu înlănţuită de tip XOR. Pentru o astfel de listă nu avem pointerii $prev$ sau $next$, ci se foloseşte un singur pointer care reprezintă şi pointerul anterior şi pe cel următor. Acest lucru este posibil folosind operaţia XOR, şi următoarele consideraţii:

   ```
   A XOR B = C
   C XOR A = B
   C XOR B = A
   ```

   Când se crează o astfel de listă se realizează un XOR între $next$ şi $prev$, şi se salvează pointerul la primul element. Să presupunem că A = $prev$, B = $next$, C = valoarea stocată.

```
prev XOR next = valoarea stocata
valoarea stocata XOR prev = next
valoarea stocata XOR next = prev
```

Dacă se traversează lista XOR dublu înlănţuită, se cunoaşte elementul curent şi elementul anterior, asa că utilizând operaţiile de mai sus se poate calcula adresa elementului următor din listă.

5. Se da un garaj pentru camioane. Drumul de access al garajului poate sa contina oricâte camioane. Garajul are o singura usa astfel încât doar ultimul camion care a intrat pote sa iasa primul (conform modelului stiva). Fiecare camion este identificat de un numar întreg pozitiv (truck_id). Scrieti un program care sa trateze diferite tipuri de mutari ale camioanelor, si sa permita urmatoarele comenzi:

   a) `Pe_drum` (truck_id);
   b) `Intra_in_garaj` (truck_ id);
   c) `Iese_din_garaj` (truck_id);
   d) `Afiseaza_camioane` (`garaj` sau `drum`);

   Daca se doreste scoaterea unui camion care nu este cel mai aproape de intrarea garajului se va afisa un mesaj de eroare `Camionul` x `nu este la usa garajului.`

6. Se considera problema anterioara a camioanelor dintr-un garaj, dar de aceasta data garajul are doua usi legate printr-un drum circular. O usa este folosita doar pentru intrarea camioanelor în garaj, iar alta usa este folosita pentru iesirea din garaj. Camioanele pot iesi din garaj doar în ordinea în care au intrat (conform modelului coada).

   Date de intrare:

```
Pe_drum(2)
Pe_drum(5)
Pe_drum(10)
Pe_drum(9)
Pe_drum(22)
Afiseaza_camioane(drum)
Afiseaza_camioane(garaj)
Intra_in_garaj(2)
Afiseaza_camioane(drum)
Afiseaza_camioane(garaj)
Intra_in_garaj(10)
Intra_in_garaj(25)
Iese_din_garaj(10)
Iese_din_garaj(2)
Afiseaza_camioane(drum)
Afiseaza_camioane(garaj)
```

   Date de iesire:

```
drum:␣2␣5␣10␣9␣22
garaj:␣nimic
drum:␣5␣10␣9␣22
garaj:␣2
error:␣25␣nu␣este␣pe␣drum!
error:␣10␣nu␣este␣la␣iesire!
drum:␣2␣5␣9␣22
garaj:␣10
```

7. De la tastatură se citesc n cuvinte; să se creeze o listă dublu înlănţuită, care să conţină în noduri cuvintele distincte şi frecvenţa lor de apariţie. Lista va fi ordonată alfabetic. Se vor afişa cuvintele şi frecvenţa lor de apariţie a) în ordine alfabetică crescătoare şi b) în ordine alfabetică descrescătoare.