Cozi şi stive

1. Noțiuni generale

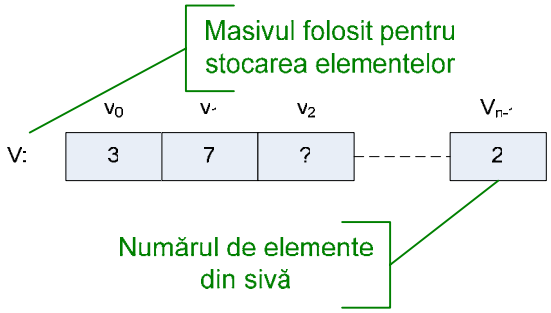
Cozile şi stivele sunt structuri de date **logice** (implementarea este făcută utilizând alte structuri de date) şi **omogene** (toate elementele sunt de acelaşi tip). Ambele structuri au două **operații de bază**: adăugarea şi extragerea unui element. În afara acestor operații se pot implementa şi alte operații utile: test de structură vidă, obținerea primului element fără extragerea acestuia, … Diferența fundamentală între cele două structuri este **disciplina de acces**. Stiva foloseşte o disciplină de acces de tip LIFO (Last In First Out), iar coada o disciplină de tip FIFO (First In First Out).

2. Moduri de implementare

Stivele şi cozile pot fi implementate în mai multe moduri. Cele mai utilizate implementări sunt cele folosind masive şi liste. Ambele abordări au avantaje şi dezavantaje:

	Avantaje	Dezavantaje
Masive	• implementare simplă • consum redus de memorie • viteză mare pentru operații	• numărul de elemente este limitat • risipă de memorie în cazul în care dimensiunea alocată este mult mai mare decât numărul efectiv de elemente
Liste	• număr oarecare de elemente	• consum mare de memorie pentru memorarea legăturilor

Pentru implementarea unei stive folosind masive avem nevoie de un masiv V de dimensiune n pentru memorarea elementelor. Ultimul element al masivului va fi utilizat pentru memorarea numărului de elemente ale stivei.

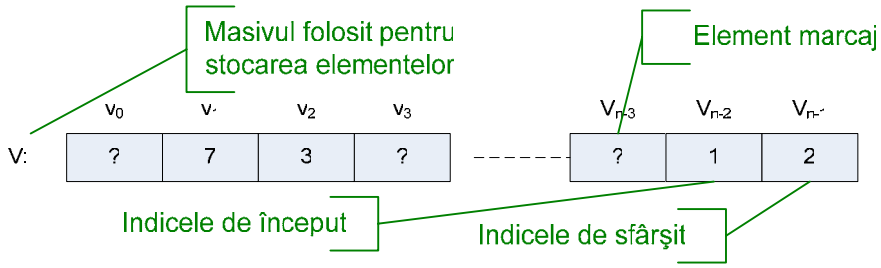


În cazul în care stiva este vidă, atunci elementul V_{n-1} va avea valoarea 0. Folosind această reprezentare, operațiile de bază se pot implementa în timp constant ($O(1)$).

Algoritmii pentru implementarea operațiilor de bază (în pseudocod) sunt:

```
adaugare(elem, V, n)
if v[n-1] = n-1                 //verificam dacă stiva nu e plină
      return "stiva plina"
v[v[n-1]] = elem                //adaugăm elementul în masiv
v[n-1] = v[n-1] + 1             //incrementăm numărul de elemente
return "succes"

stergere(V, n)
if v[n-1] = 0                   //verificam dacă stiva nu e goală
      return "stiva goala"
elem = v[v[n-1] - 1]            //extragem elementul din masiv
v[n-1] = v[n-1] + 1             //decrementăm numărul de elemente
return elem
```

Coada se poate implementa folosind un vector circular de dimensiune *n* (după elementul *n-4* urmează elementul 0). Ultimele două elemente conțin indicii de start şi sfârşit ai cozii, iar antepenultimul element este un marcaj folosit pentru a putea diferenția cazurile de coadă goală şi coadă plină.

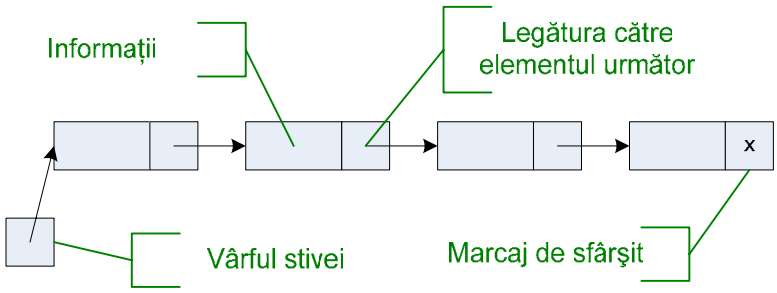


Algoritmii care implementează operațiile de bază pentru o coadă memorată in forma prezentată sunt:

```
adaugare(elem, V, n)
v[n-2] = (v[n-2] + 1) mod (n-2)   //deplasăm capul cozii
if v[n-1] = v[n-2]                //verificare coada plină
      return "coada plina"
V[V[n-1]] = elem                  //adăugăm elementul
return "succes"

stergere(V,n)
if v[n-1] = v[n-2]                //verificare coadă goală
      return "coada goala"
v[n-1] = (v[n-1] + 1) mod (n-2)   //deplasăm indicele de sfârşit
return V[V[n-1]]                  //întoarcem elementul
```

Cea de-a doua modalitate de implementare a stivelor şi cozilor este cea folosind liste alocate dinamic.

În cazul stivei, vom folosi o listă simplu înlănțuită organizată ca în figura următoare:



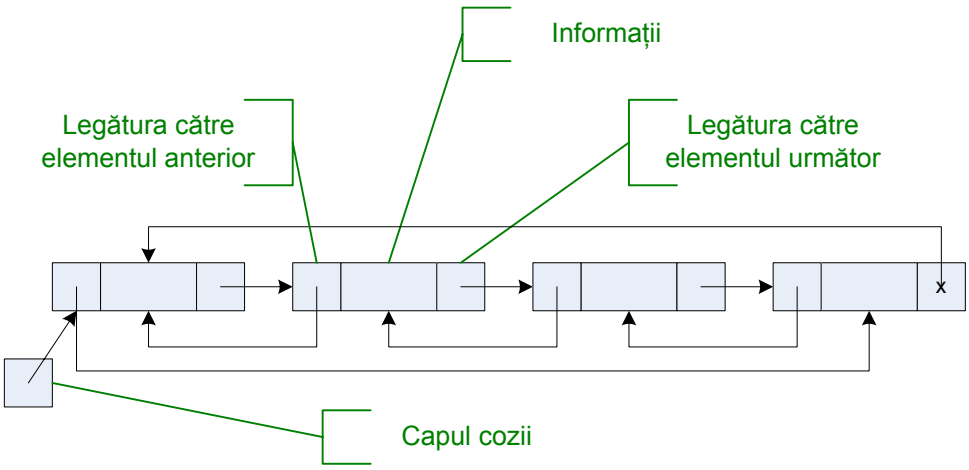
Fiecare nod este format din informațiile utile şi o legătură către elementul următor. Tipul informațiilor stocate în stivă este indicat de utilizator prin definirea tipului *TipStiva*. Stiva vidă este reprezentată printr-un pointer nul. Elementele sunt adăugate înaintea primului element (cu deplasarea varfului stivei). Extragerea se face tot din vărful stivei.

Codul sursă pentru biblioteca care implementează operațiile pe stiva alocată dinamic este:

```
#ifndef STIVA_H
#define STIVA_H

// Un element din stiva
struct NodStiva
{
	TipStiva Date;		// tipul definit de utilizator
	NodStiva *Urmator;	// legatura catre elementul urmator

	// constructor pentru initializarea unui nod
	NodStiva(TipStiva date, NodStiva *urmator = NULL) :
		Date(date), Urmator(urmator){}
};

// Stiva este memorata ca un
// pointer catre primul element
typedef NodStiva* Stiva;

// Creaza o stiva vida
Stiva StCreare()
{
	return NULL;
}

// Verivica daca o stiva este vida
bool StEGoala(Stiva& stiva)
{
	return stiva == NULL;
}

// Adauga un element in stiva
void StAdauga(Stiva& stiva, TipStiva date)
{
	stiva = new NodStiva(date, stiva);
}

// Intoarce o copie a varfului stivei
TipStiva StVarf(Stiva& stiva)
{
	// Caz 1: stiva vida
	if (StEGoala(stiva))			// daca stiva e goala, atunci
		return TipStiva();		// intoarcem valoarea implicita pentru
tipul stivei

	// Caz 2: stiva nevida
	return stiva->Date;			// intoarcem varful stivei
}

// Extrage elementul din varful stivei
TipStiva StExtrage(Stiva& stiva)
```

```
{
	// Caz 1: stiva vida
	if (StEGoala(stiva))			// daca stiva e goala, atunci
		return TipStiva();		// intoarcem valoarea implicita pentru
tipul stivei

	// Caz 2: stiva nevida
	NodStiva *nodDeSters = stiva; // salvam o referinta la nodul de sters
	TipStiva rez = stiva->Date;   // salvam datele de returnat

	stiva = stiva->Urmator;       // avansam capul listei

	delete nodDeSters;            // stergem nodul de sters

	return rez;                   // intoarcem rezultatele
}

#endif //STIVA_H
```

Coada poate fi implementată folosind o listă circulară dublu înlanțuită de forma:



Ca şi în cazul stivei, tipul informațiilor stocate este indicat de către utilizator prin definirea tipului de date *TipCoada*. Coada goală este reprezentată printr-un pointer nul. Adăugarea elementelor se face la sfârşitul listei, iar extragerea se face de la începutul acesteia.

Codul sursă care implementează operațiile pe structura de date prezentată este:

```
#ifndef COADA_H
#define COADA_H

// Un element din coada
struct NodCoada
{
	// tipul definit de utilizator
	TipCoada Date;

	// legaturile catre elementul
	// urmator si anterior
	NodCoada *Urmator, *Anterior;

	// constructor pentur initializarea unui nod
	NodCoada(TipCoada date,
		NodCoada *urmator = NULL, NodCoada *anterior = NULL) :
		Date(date), Urmator(urmator), Anterior(anterior)
	{}
};

// Coada este memorata ca un
```

```
// pointer catre primul element
typedef NodCoada* Coada;

// Creaza o coada vida
Coada CdCreare()
{
	return NULL;
}

// Testeaza daca o coada este vida
bool CdEGoala(Coada& coada)
{
	return coada == NULL;
}

// Adauga un element la sfarsitul cozii
void CdAdauga(Coada& coada, TipCoada date)
{
	if (CdEGoala(coada))
	{
		// Cazul 1: Coada vida
		coada = new NodCoada(date);
		coada->Anterior = coada->Urmator = coada;
	}
	else
	{
		// Cazul 2: Coada cu cel putin un element
		coada->Anterior->Urmator = new NodCoada(date, coada, coada->Anterior);
		coada->Anterior = coada->Anterior->Urmator;
	}
}

// Obtine o copie a primului element din coada
TipCoada CdVarf(Coada& coada)
{
	// Caz 1: coada vida
	if (CdEGoala(coada))		// daca coada e goala, atunci
		return TipCoada();		// intoarcem valoarea implicita pentru
tipul stivei

	// Caz 2: coada nevida
	return coada->Date;			// intoarcem varful cozii
}

// Extrage primul element din coada
TipCoada CdExtrage(Coada& coada)
{
	TipCoada rez;

	// Caz 1: coada vida
	if (CdEGoala(coada))
		return TipCoada();

	// Caz 2: coada cu un singur element
	if (coada->Anterior == coada)
	{
		rez = coada->Date;
		delete coada;
		coada = NULL;

		return rez;
	}

	// Caz 3: coada cu mai multe elemente
	NodCoada *nodDeSters = coada;
	rez = coada->Date;

	coada = coada->Urmator;
	coada->Anterior = coada->Anterior->Anterior;
	coada->Anterior->Urmator = coada;
	delete nodDeSters;

	return rez;
}

#endif //COADA_H
```

3. Aplicație – Evaluarea expresiilor aritmetice

Pentru exemplificarea utilizării structurilor de tip stivă şi coadă vom construi o aplicație pentru evaluarea expresiilor aritmetice. Expresiile acceptate pot conține numere naturale, operații aritmetice de bază (adunare, scădere, înmulțire, împărțire) şi paranteze.

Pentru evaluarea expresiei vom folosi forma poloneză postfixată. Această formă de scriere a expresiei permite o evaluarea rapidă într-o singură parcurgere.

Evaluarea expresiei primite sub forma unui şir de caractere se face în trei faze:

1. Se transformă şirul primit într-o coadă de elemente numite atomi care conțin informațiile necesare transformării şi evaluării expresiei (tip, prioritate şi valoare).
2. Expresia stocată ca o coadă de atomi este rescrisă în forma poloneză postfixată.
3. Se evaluează expresia.

Structurile folosite pentru memorarea expresiei sunt:

```
// Tipurile de atomi acceptate de aplicatie
enum TipAtom
{
	Termen,				// un numar
	DeschidereParanteza,		// )
	InchidereParanteza,		// (
	Plus,					// +
	Minus,				// -
	Inmultire,				// *
	Impartire				// /
};

// Informatiile referitoare la un atom
struct Atom
{
	TipAtom Tip;
	int Prioritate;
	double Valoare;

	// Constructorul pentru initializarea
	// unui atom
	Atom(TipAtom tip = Termen,
		int prioritate = 0, double valoare = 0)
	{
		Tip = tip;
		Prioritate = prioritate;
		Valoare = valoare;
	}

};
```

Algoritmii folosesc bibliotecile pentru manipularea stivelor şi cozilor prezentate în secțiunea precedentă:

```
// Stivele si cozile folosite vor
// avea ca informatie utila atomi
typedef Atom TipStiva;
typedef Atom TipCoada;

#include "stiva.h"

#include "coada.h"
```

Prima fază a aplicației este transformarea expresiei într-o coadă de atomi. Funcția care implementează operația este:

```
// Transforma expresia intr-o coada de atomi
Coada ParsareSir(char *expresie)
{
	Coada coada = CdCreare();

	while (*expresie != '\0')
	{
		switch(*expresie)
		{
			case '+': CdAdauga(coada, Atom(Plus, 1)); break;
			case '-': CdAdauga(coada, Atom(Minus, 1)); break;
			case '*': CdAdauga(coada, Atom(Inmultire, 2)); break;
			case '/': CdAdauga(coada, Atom(Impartire, 2)); break;
			case '(': CdAdauga(coada, Atom(DeschidereParanteza, 0)); break;
			case ')': CdAdauga(coada, Atom(InchidereParanteza,  0)); break;
			default:
				// termen (numar intreg)
				if (*expresie > '0' && *expresie <= '9')
				{
					// construim termenul
					double valoare = 0;
					while (*expresie >= '0' && *expresie <= '9')
					{
						valoare = valoare*10 + (*expresie - '0');
						expresie++;
					}

					// trebuie sa revenim la primul caracter
					// de dupa numar
					expresie--;

					// adaugam termenul in coada
					CdAdauga(coada, Atom(Termen,  0, valoare));
				}
		}

		// avansam la urmatorul caracter din expresie
		expresie++;
	}

	return coada;
}
```

Transformarea expresiei din forma normală infixată în forma poloneză postfixată se face folosind algoritmul lui Dijkstra. Acest algoritm utilizează o stivă în care sunt păstrați operatorii şi din care sunt eliminați şi transferați în scrierea postfixată (o coadă).

Algoritmul este:

1. se inițializează stiva şi scrierea postfixată;
2. atât timp cât nu s-a ajuns la sfârşitul expresiei matematice:
    a. se citeşte următorul element din expresie;
    b. dacă este valoare se adaugă în scrierea postfixată;
    c. dacă este *„(”* se introduce în stivă;
    d. dacă este *)* se transferă elemente din stivă în scrierea postfixată până la*(*
    e. altfel:
        i. atât timp cât ierarhia operatorului din vârful stivei este mai mare ierarhia operatorului curent, se trece elementul din vârful stivei în scrierea postfixată;

ii. se introduce operatorul curent în stivă.

3. se trec toți operatorii rămaşi pe stivă în scrierea postfixată.

Algoritmul este implementat folosind următoarea funcție:

```
// Transforma o expresie din forma normala
// infixata in forma poloneza postfixata
Coada FormaPoloneza(Coada& expresie)
{
	// stiva folosita pentru stocarea temporara
	Stiva stiva = StCreare();

	// coada pentru stocarea rezultatului (forma poloneza)
	Coada formaPoloneza = CdCreare();

	Atom atomStiva;

	// parcurgem expresia initiala
	while (!CdEGoala(expresie))
	{
		Atom atom = CdExtrage(expresie);

		switch (atom.Tip)
		{
		case Termen:
		// se adauga direct in sirul rezultat
			CdAdauga(formaPoloneza, atom);
			break;

		case DeschidereParanteza:
		// se adauga in stiva
			StAdauga(stiva, atom);
			break;

		case InchidereParanteza:
		// se muta din stiva in rezultat toti operatorii
		// pana la deschiderea parantezei
			atomStiva = StExtrage(stiva);
			while (atomStiva.Tip != DeschidereParanteza)
			{
				CdAdauga(formaPoloneza, atomStiva);
				atomStiva = StExtrage(stiva);
			}
			break;
		default: // operator
			while (!StEGoala(stiva) && StVarf(stiva).Prioritate >
atom.Prioritate)
				CdAdauga(formaPoloneza, StExtrage(stiva));

			StAdauga(stiva, atom);
		}
	}

	// mutam toate elementele ramase in rezultat
	while (!StEGoala(stiva))
		CdAdauga(formaPoloneza, StExtrage(stiva));

	return formaPoloneza;

}
```

Evaluarea expresiei în forma poloneză se face folosind stivă pentru operanzi după algoritmul următor:

1. se inițializează stiva;
2. atât timp cât nu s-a ajuns la sfârşitul scrierii postfixate:
   a. se citește următorul element;
   b. dacă este valoare se depune pe stivă;

c. altfel, respectiv cazul în care este operator:
   i. se extrage din stivă elementul *y*;
   ii. se extrage din stivă elementul *x*;
   iii. se efectuează operația *x operator y*;
   iv. se depune rezultatul pe stivă;
3. ultima valoare care se află pe stivă este rezultatul expresiei.

Funcția care implementează algoritmul este:

```
// Evaluaeaza o expresie aflata in
// forma poloneza postfixata
double Evaluare(Coada& formaPoloneza)
{
	// stiva de termeni folosita pentru evaluare
	Stiva stiva = StCreare();

	// parcurgem expresia in forma poloneza
	while (!CdEGoala(formaPoloneza))
	{
		Atom atom = CdExtrage(formaPoloneza);

		if (atom.Tip == Termen)
			// daca avem un termen atunci il adaugam in stiva
			StAdauga(stiva, atom);
		else
		{
			// daca avem un operator atunci scoatem
			// ultimii doi termeni din stiva,  efectuam operatia
			// si punem rezultatul pe stiva
			Atom termen1 = StExtrage(stiva);
			Atom termen2 = StExtrage(stiva);

			switch (atom.Tip)
			{
			case Plus:
				StAdauga(stiva, Atom(Termen, 0, termen1.Valoare +
termen2.Valoare));
				break;
			case Minus:
				StAdauga(stiva, Atom(Termen, 0, termen2.Valoare -
termen1.Valoare));
				break;
			case Inmultire:
				StAdauga(stiva, Atom(Termen, 0, termen1.Valoare *
termen2.Valoare));
				break;
			case Impartire:
				StAdauga(stiva, Atom(Termen, 0, termen2.Valoare /
termen1.Valoare));
				break;
			}
		}
	}

	// rezultatul expresiei este valoarea
	// ultimului termen ramas in stiva
	return StExtrage(stiva).Valoare;
}
```

Codul sursă complet al aplicației este:

```
#include <iostream>
using namespace std;

const int DIM_MAX_EXPRESIE = 1024;

// Tipurile de atomi acceptate de aplicatie
```

```
enum TipAtom
{
	Termen,					// un numar
	DeschidereParanteza,	// )
	InchidereParanteza,		// (
	Plus,					// +
	Minus,					// -
	Inmultire,				// *
	Impartire				// /
};

// Informatiile referitoare la un atom
struct Atom
{
	TipAtom Tip;
	int Prioritate;
	double Valoare;

	// Constructorul pentru initializarea
	// unui atom
	Atom(TipAtom tip = Termen,
		int prioritate = 0, double valoare = 0)
	{
		Tip = tip;
		Prioritate = prioritate;
		Valoare = valoare;
	}
};

// Stivele si cozile folosite vor
// avea ca informatie utila atomi
typedef Atom TipStiva;
typedef Atom TipCoada;

#include "stiva.h"
#include "coada.h"

// Transforma expresia intr-o coada de atomi
Coada ParsareSir(char *expresie)
{
	Coada coada = CdCreare();

	while (*expresie != '\0')
	{
		switch(*expresie)
		{
			case '+': CdAdauga(coada, Atom(Plus, 1)); break;
			case '-': CdAdauga(coada, Atom(Minus, 1)); break;
			case '*': CdAdauga(coada, Atom(Inmultire, 2)); break;
			case '/': CdAdauga(coada, Atom(Impartire, 2)); break;
			case '(': CdAdauga(coada, Atom(DeschidereParanteza, 0)); break;
			case ')': CdAdauga(coada, Atom(InchidereParanteza,  0)); break;
			default:
				// termen (numar intreg)
				if (*expresie > '0' && *expresie <= '9')
				{
					// construim termenul
					double valoare = 0;
					while (*expresie >= '0' && *expresie <= '9')
					{
						valoare = valoare*10 + (*expresie - '0');
						expresie++;
					}

					// trebuie sa revenim la primul caracter
					// de dupa numar
					expresie--;

					// adaugam termenul in coada
					CdAdauga(coada, Atom(Termen,  0, valoare));
				}
		}

		// avansam la urmatorul caracter din expresie
		expresie++;
	}
```

```
	return coada;
}

// Transforma o expresie din forma normala
// infixata in forma poloneza postfixata
Coada FormaPoloneza(Coada& expresie)
{
	// stiva folosita pentru stocarea temporara
	Stiva stiva = StCreare();

	// coada pentru stocarea rezultatului (forma poloneza)
	Coada formaPoloneza = CdCreare();

	Atom atomStiva;

	// parcurgem expresia initiala
	while (!CdEGoala(expresie))
	{
		Atom atom = CdExtrage(expresie);

		switch (atom.Tip)
		{
		case Termen:
		// se adauga direct in sirul rezultat
			CdAdauga(formaPoloneza, atom);
			break;

		case DeschidereParanteza:
		// se adauga in stiva
			StAdauga(stiva, atom);
			break;

		case InchidereParanteza:
		// se muta din stiva in rezultat toti operatorii
		// pana la deschiderea parantezei
			atomStiva = StExtrage(stiva);
			while (atomStiva.Tip != DeschidereParanteza)
			{
				CdAdauga(formaPoloneza, atomStiva);
				atomStiva = StExtrage(stiva);
			}
			break;
		default: // operator
			while (!StEGoala(stiva) && StVarf(stiva).Prioritate >
atom.Prioritate)
				CdAdauga(formaPoloneza, StExtrage(stiva));

			StAdauga(stiva, atom);
		}
	}

	// mutam toate elementele ramase in rezultat
	while (!StEGoala(stiva))
		CdAdauga(formaPoloneza, StExtrage(stiva));

	return formaPoloneza;
}

// Evaluaeaza o expresie aflata in
// forma poloneza postfixata
double Evaluare(Coada& formaPoloneza)
{
	// stiva de termeni folosita pentru evaluare
	Stiva stiva = StCreare();

	// parcurgem expresia in forma poloneza
	while (!CdEGoala(formaPoloneza))
	{
		Atom atom = CdExtrage(formaPoloneza);

		if (atom.Tip == Termen)
			// daca avem un termen atunci il adaugam in stiva
			StAdauga(stiva, atom);
		else
		{
			// daca avem un operator atunci scoatem
			// ultimii doi termeni din stiva,  efectuam operatia
```

```
                        // si punem rezultatul pe stiva
                        Atom termen1 = StExtrage(stiva);
                        Atom termen2 = StExtrage(stiva);

                        switch (atom.Tip)
                        {
                        case Plus:
                                StAdauga(stiva, Atom(Termen, 0, termen1.Valoare +
termen2.Valoare));
                                break;
                        case Minus:
                                StAdauga(stiva, Atom(Termen, 0, termen2.Valoare -
termen1.Valoare));
                                break;
                        case Inmultire:
                                StAdauga(stiva, Atom(Termen, 0, termen1.Valoare *
termen2.Valoare));
                                break;
                        case Impartire:
                                StAdauga(stiva, Atom(Termen, 0, termen2.Valoare /
termen1.Valoare));
                                break;
                        }
                }
        }

        // rezultatul expresiei este valoarea
        // ultimului termen ramas in stiva
        return StExtrage(stiva).Valoare;
}

void main()
{
        // alocare spatiu pentru expresia citita
        char *sir = new char[DIM_MAX_EXPRESIE];

        // citire expresie de la tastatura
        cout << "Expresia:";
        cin.getline(sir, 10000);

        // Faza 1: Construirea cozii de atomi
        Coada expresie = ParsareSir(sir);

        // Faza 2: Transformarea in forma poloneza
        Coada formaPoloneza = FormaPoloneza(expresie);

        // Faza 3: Evaluarea expresiei
        double rezultat = Evaluare(formaPoloneza);

        // afisarea retzultatului
        cout << "Valoare: " << rezultat << endl;

}
```

4. Probleme

1. Să se scrie funcția de inversare a unui şir folosind o stivă alocată dinamic.
2. Să se scrie funcția de adăugare pentru o stivă alocată ca vector.
3. Să se scrie funcția de extragere a unui element dintr-o stivă alocată ca vector.
4. Să se descrie modul de organizare a unei cozi alocate ca vector şi să se scrie funcția de adăugare a unui element.
5. Să se descrie modul de organizare a unei cozi alocate ca vector şi să se scrie funcția de ştergere a unui element.
6. Să se scrie funcția de concatenare a două stive alocate dinamic S_1 şi S_2 folosind doar operațiile de bază (adăugare, extragere, testare stivă vidă).

Rezultatul trebuie să conțină elementele din cele două stive în ordinea inițială.

Indicație: Se va folosi o stivă temporară.

7. Să se scrie funcția de conversie a unei stive în listă simplu înlănțuită.
8. Se consideră un şir de numere întregi. Să se scrie funcția care construieşte două stive (una cu numerele negative şi una cu cele pozitive) ce conțin numerele în ordinea inițială folosind doar structuri de tip stivă.

Indicație: Se adaugă toate elementele într-o stivă temporară după care se extrag elementele din aceasta şi se introduc în stiva corespunzătoare.

9. Scrieți funcția recursivă pentru ştergerea tuturor elementelor dintr-o stivă alocată dinamic.
10. Scrieți funcția pentru ştergerea tuturor elementelor dintr-o coadă alocată dinamic.