11. STIVE ŞI COZI

11.1 Consideraţii privind structurilor de date de tip stivă şi coadă

O *stivă*, *stack în limba engleză,* este o structură de tip LIFO (Last In First Out =ultimul intrat primul ieşit) şi este un caz particular al listei liniare în care toate inserările (depunerile –în engleză *push*) şi ştergerile (sau extragerile -în engleză *pop*) (în general orice acces) sunt făcute la unul din capetele listei, numit vârful stivei. Acest nod poate fi citit, poate fi şters sau în faţa lui se poate insera un nou nod care devine cap de stivă.

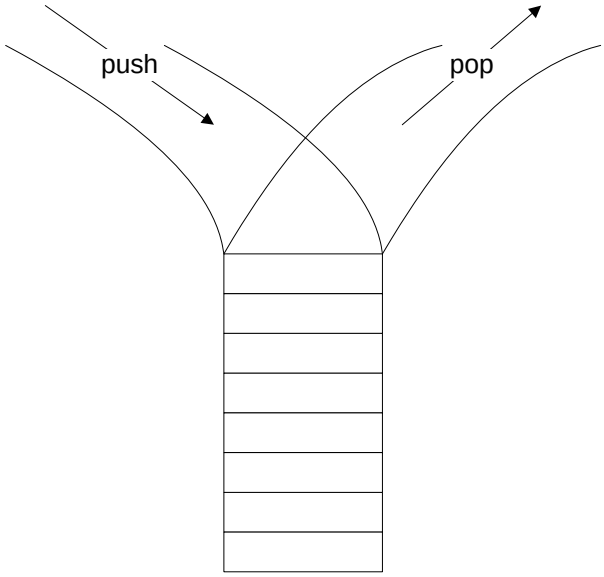

Figura 11.1 *Mecanismul de stivă*

Pentru înţelegerea mecanismului unei stive, se poate folosi reprezentarea manevrării într-un depou a vagoanelor de cale ferată sau a unui vraf de farfurii din care putem depune şi extrage doar în şi din vârful vrafului. Stivele sunt legate şi de algoritmii recursivi, la care salvarea variabilelor dintr-o funcţie care se apelează recursiv se face având la bază un mecanism de tip stivă.

O stivă poate fi implementată ca o listă liniară pentru care operaţiile de acces (inserare, ştergere, accesare element) sunt restricţionate astfel:

- inserarea se face doar în faţa primului element al listei, capul listei;
- accesarea respectiv ştergerea acţionează doar asupra primului element al listei.

O stivă poate fi implementată folosind o structură de date statică sau dinamică.

În abordarea statică, stiva poate fi organizată pe un spaţiu de memorare de tip tablou unidimensional.

În abordarea dinamică, implementarea unei stive se poate face folosind o structură de tip listă liniară simplu înlănţuită în care inserarea se va face tot timpul în capul listei iar ştergerea de asemenea se va face în capul listei.

O *coadă* (în engleză *queue*) este o structură de tip FIFO (First In First Out = Primul venit primul servit), în care toate inserările se fac la un capăt al ei (numit capul cozii) iar ştergerile (extragerile) (în general orice acces) se fac la celălalt capăt (numit sfârşitul cozii). În cazul cozii, avem nevoie de doi pointeri, unul către primul element al cozii (capul cozii), iar altul către ultimul său element (sfârşitul cozii). Există şi o variantă de coadă circulară, în care elementele sunt legate în cerc, iar cei doi pointeri, indicând capul şi sfârşitul cozii, sunt undeva pe acest cerc.

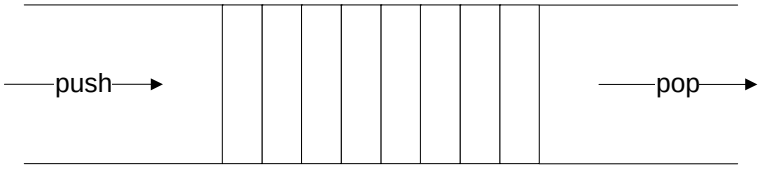

Figura 11.2 *Mecanismul de coadă*

Se face o analogie cu o cale ferată pe un singur sens sau cu obişnuita coadă la un ghişeu oarecare la care primul venit este şi primul servit.

Pentru gestiunea unei cozi sunt necesari doi indicatori:

- un indicator care indică primul element ce urmează a fi scos;
- un indicator care indică unde va fi pus următorul element adăugat la coadă.

Într-o abordare statică, o coadă poate fi implementată folosind un spaţiu de memorare de tip tablou unidimensional. În acest caz, adăugări şi ştergeri repetate în coadă deplasează conţinutul cozii la dreapta, faţă de începutul vectorului. Pentru evitarea acestei deplasări, o soluţie este ca fiecare operaţie de extragere din coadă să fie acompaniată de deplasarea la stânga a conţinutului cozii cu o poziţie. În acest caz, primul element care urmează să fie eliminat din coadă va fi întotdeauna pe prima poziţie, indicatorul care să indice elementul ce urmează să fie scos pierzându-şi utilitatea. Dezavantajul acestei soluţii îl reprezintă necesitatea deplasării întregului set de elemente din coadă rămase cu o poziţie.

Într-o abordare dinamică, o coadă poate fi implementată folosind o listă liniară simplu înlănţuită la care operaţiile de acces sunt restricţionate corespunzător. Există două abordări:

-una în care adăugarea se face tot timpul la începutul listei liniare simplu înlănţuite iar extragerea se face de la sfârşitul listei

-cea de a doua, în care adăugarea se face la sfârşitul listei iar extragerea se face din capul listei.

Pentru implementarea unei cozi, vom folosi aspectele tratate în capitolul de liste liniare.

11.2 Caracteristicile structurilor de date de tip stivă

Caracteristicile unei stive sunt puse în evidenţă prin comparaţie cu structura de date de tip listă:

- în timp ce pentru gestionarea listei se vehiculează cu variabile de tip T_p, care stochează adresa componentei (Z_1, θ_1) a listei, componenta numită cap de listă, în cazul stivei este memorată adresa ultimei componente (Z'_n, θ'_n), numită vârful stivei;
- în timp ce cont $cont(\theta_n) = NULL$ în cazul listei, $cont(\theta'_1) = NULL$ în cazul stivei;
- în timp ce $succ(Z_j) = Z_{j+1}$ în cazul listei, în cazul stivei $succ(Z'_j)=Z'_{j-1}$;
- în timp ce $pred(Z_j) = Z_{j-1}$ în cazul listei, în cazul stivei $pred(Z'_j)=Z'_{j+1}$;
- în tip ce parcurgerea în cazul listei este de la (Z_1, θ_1) spre (Z_n, θ_m), în cazul stivei, parcurgerea este de la (Z'_n, θ'_n) spre (Z'_1, θ'_1);
- în timp ce disciplina lucrului cu elementele listei este primul venit – primul servit (FIFO), în cazul stivei regula de vehiculare a elementelor, este ultimul venit – primul servit (LIFO).

Se observă că stiva este o listă inversă. Totuşi, multe dintre tehnicile de rezolvare a problemelor vehiculează stive şi nu liste, datorită disciplinei de gestionare a elementelor.

Regulile de creare şi accesare la nivel de stivă, prevăd că fiecărui element care se adaugă, i se alocă o zonă de memorie şi acest element devine vârf al stivei.

Există procese în care informaţiile se stochează în ordinea opusă momentului prelucrării, iar odată folosită, aceasta devine necesară. Stocând informaţiile într-o stivă, după utilizarea informaţiilor conţinute în vârful stivei, zona de memorie alocată este eliberată (dealocată), vârful stivei mutându-se cu o componentă spre baza stivei.

Baza stivei este considerat elementul (Z'_1, θ'_1). În funcţie de cerinţele de prelucrare, se obţine o fluctuaţie a poziţiei vârfului stivei, ceea ce creează o imagine mai bună pentru dinamica prelucrării datelor. Astfel, stivele stau la baza gestionării adreselor parametrilor ce se transmit în funcţii, implementării mecanismelor de recursivitate din limbaje de programare.

În general, ideea de stivă sugerează prelucrări care au un caracter simetric. De exemplu, se consideră o prelucrare completă privind:

A. încărcarea componentelor C_1 pentru disponibilizarea de resurse ale unui sistem de calcul;
B. încărcarea componentei C_2 care este destinată lansării în execuţie a programului utilizatorului;
C. efectuarea deschiderii de fişiere în programul utilizatorului – C_3;
D. prelucrări de date şi afişări de rezultate – C_4;
E. închiderea fişierelor;
F. ieşirea din programul utilizatorului sub supravegherea componentei C_3;
G. ieşirea din lucru sub componenta C_2;

H. dezactivarea resurselor sistemului de calcul prin componenta C_1.
Se construieşte stiva din figura 11.3.

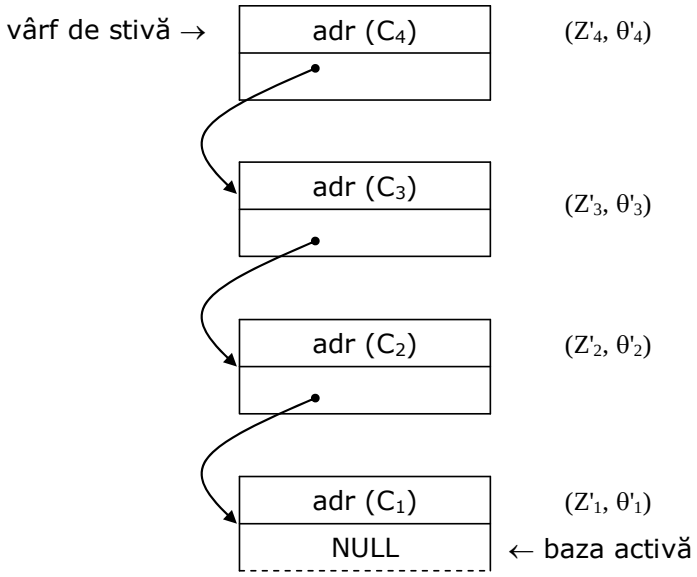

Figura 11.3 *Structura unei stive*

După parcurgerea pasului D, vârfului stivei coboară la (Z'_3, θ'_3) şi componenta (Z'_4, θ'_4) este dealocată, deci distrusă.

Folosind informaţiile stocate în pasul C, se efectuează ştergerea fişierelor (pasul E). Vârful stivei se coboară la componenta (Z'_2, θ'_2), iar componenta (Z'_3, θ'_3) este eliberată.

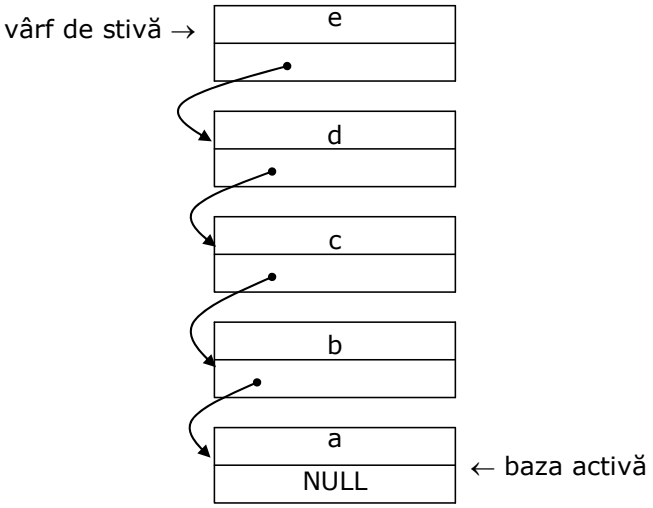

Figura 11.4 *Structura unei stive pentru inversarea caracterelor unui şir*

Se execută pasul F, care este simetricul pasului C şi apoi vârful coboară spre (Z'_1, θ'_1) şi se activează componentele C_1 pentru eliberarea resurselor pas simetric pasului A.

Astfel, de exemplu, dacă dorim să inversăm poziţia elementelor unui şir {a, b, c, d, e}, prin crearea unei stive care să conţină aceste elemente, cu modelul grafic din figura 11.4 şi prin apelarea unei funcţii de parcurgere cu imprimarea elementelor identificate, se obţine exact şirul { e, d, c, b, a }.

În cazul calculării lui *n!*, se foloseşte formula de recurenţă:

$$P(n) = P(n-1) * n \qquad (11.1)$$

cu $P(1) = 1$, se creează o stivă care conţine expresiile ce sunt evaluate cu valorile descrescătoare ale lui n.

Se construieşte stiva care are în informaţia utilă, de la bază spre vârf: *n, n – 1, n – 2, …, 3, 2, 1*.

Valoarea iniţială a funcţiei recursive $P(1) = 1$, permite prin parcurgerea stivei de la vârf spre bază, obţinerea rând pe rând a rezultatelor: $P(2), P(3),\ldots,P(n-2), P(n-1), P(n)$ şi întâlnindu-se NULL la baza stivei, procesul se încheie.

11.3 Operaţii de bază cu stive

Ştergerea unei stive, revine la dealocarea componentelor care formează stiva. Se realizează prin parcurgerea integrală a stivei, cu dealocarea fiecărui vârf.

Adăugarea unui element în stivă, se face de regulă cu mutarea poziţiei vârfului stivei pe componenta adăugată.

Ştergerea unui element din stivă, se efectuează de regulă asupra elementului din vârful stivei, aceasta coborând pe componenta ce îl precede.

Parcurgerea stivei, se efectuează de la vârf către bază.

Operaţiile de inserare, de concatenare, de sortare a stivelor, nu diferă mult de aceleaşi operaţii din interiorul unei liste, dar frecvenţa lor de utilizare este mult mai redusă.

Programul următor, exemplifică unele dintre operaţiile care se execută cu structuri de date de tip stivă.

```
#include <iostream.h>
#include <malloc.h>

class elementstiva
{
public:
      int info;
      elementstiva *prec;
      elementstiva()
      {
            prec=NULL;
      }
};

class stiva
{
```

```
public:
	elementstiva *vs; //varful stivei

	stiva()
	{
		vs=NULL;
	}
	void sterg()
	{
		elementstiva* aux=vs;
		if(vs == NULL) cout<<"\n Stiva este vida!";
		else
		{
			aux = vs->prec;
			free(vs);
			while(aux !=NULL)
			{
				vs=aux;
				aux=aux->prec;
				delete vs;
			}
			vs = NULL;
		}

	}//
	void tipar()
	{
		elementstiva* aux;
		if(vs == NULL) cout<<"\n Stiva este vida!";
		else
		{
			aux = vs;
			while(aux !=NULL)
			{
				cout<<aux->info;
				aux=aux->prec;
			}
		}

	}

	void inserare(int el)
	{
		elementstiva *aux;
		aux = new elementstiva();
		aux->info = el;
		aux->prec = vs;
		vs = aux;

	}//

	int extrag()
	{
		elementstiva* aux;
		int el;
		if (vs == NULL) return -1;
		else
```

```
            {
                  aux = vs->prec;
                  el = vs->info;
                  delete vs;
                  vs = aux;
                  return el;
            }

      }

};

void main()
{
      stiva st;
      char opt;
      int x,el;

      do
      {
            cout<<"\n Optiuni de lucru cu stiva ";
            cout<<"\n P - Tiparire stiva";
            cout<<"\n A - Adaugare element în stiva";
            cout<<"\n E - Extragere element din stiva";
            cout<<"\n T - Terminare lucru";
            cin>>opt;
            switch(opt)
            {
            case 'a':
                  cout<<"element:";cin>>el;
                  st.inserare(el);
                  break;
            case 'e':
                  {
                        x = st.extrag();
                        if (x==-1)
                              cout<<"\n Stiva este vida!";
                        else
                              cout<<"\n Element extras"<<x;
                  }
                  break;
            case 's':
                  st.sterg();
                  break;
            case 'p':
                  st.tipar();
                  break;
            default:
                  cout<<"\n Nu exista optiunea!";
            }
      }while (opt!='t');
}
```

Pe baza meniului afişat prin program, se activează funcţiile în orice ordine, iar situaţia de stivă vidă, este marcată în mod corespunzător. Crearea stivei se realizează prin adăugări succesive, care presupun alocări dinamice de

memorie pentru elementele componente. Procedura de ştergere eliberează toată zona de memorie alocată stivei, iar funcţia de extragere, eliberează zona de memorie ocupată de vârful stivei şi iniţializează cu adresa noului vârf al stivei, variabila VS.

11.4 Evaluarea expresiilor matematice cu ajutorul stivei şi cozii

Pentru evaluarea expresiilor matematice există diferiţi algoritmi. Unul dintre aceştia foloseşte ca structură principală de date stiva.

Acest algoritm presupune rearanjarea expresiei într-o anumită formă astfel încât ordinea operaţiilor să fie clară şi evaluarea să necesite o singură parcurgere a expresiei. Pentru aceasta se poate folosi forma poloneză, inventată de matematicianul de origine poloneză Jan Lukasiewicz. Acesta presupune scrierea operatorilor înaintea operanzilor. Această formă mai are o variantă numită scrierea poloneză inversă în care operatorii sunt scrişi în urma operanzilor.

Tabelul nr. 11.1 *Forme ale scrierii unei expresii matematice*

Expresia matematică (scriere infixată)	**Expresia în forma poloneză (scriere prefixată)**	**Expresia în forma poloneză inversă (scriere postfixată)**
4 + 5	**+ 4 5**	**4 5 +**
4 + 5 * 5	**+ 4 * 5 5**	**4 5 5 * +**
4 * 2 + 3	**+ * 4 2 3**	**4 2 * 3 +**
4 + 2 + 3	**+ + 4 2 3**	**4 2 + 3 +**
4 * (2 + 3)	*** 4 + 2 3**	**4 2 3 + ***

După cum se vede din tabelul 11.1, ordinea operanzilor nu se schimbă, ei găsindu-se în aceeaşi ordine ca în expresia matematică.

Forma poloneză inversă are avantaje faţă de scrierea prefixată sau infixată:

- ordinea în care se efectuează operaţiile este clară;
- parantezele nu mai sunt necesare;
- evaluările sunt uşor de efectuat cu ajutorul calculatorului.

Un algoritm de transformare din expresie matematică în scriere postfixată a fost dezvoltat de către Edsger Dijkstra (algoritmul macazului al lui Dijkstra – Dijkstra Shunting Algorithm). Acest algoritm utilizează o stivă în care sunt păstraţi operatorii şi din care sunt eliminaţi şi transferaţi în scrierea postfixată. Fiecare operator are atribuită o ierarhie după cum este prezentat în tabelul 11.2.

Tabelul nr. 11.2 *Ierarhia operatorilor*

Operator	Ierarhie
([{	1
)] }	2
+ -	3
* /	4

Algoritmul este:

- se iniţializează stiva şi scrierea postfixată;
- atât timp cât nu s-a ajuns la sfârşitul expresiei matematice:
  - se citeşte următorul element din expresie;
  - dacă este valoare se adaugă în scrierea postfixată;
  - dacă este *„(”* se introduce în stivă;
  - dacă este *„)”* se transferă elemente din stivă în scrierea postfixată până la *„(”*;
  - altfel:
    - atât timp cât ierarhia operatorului din vârful stivei este mai mare ierarhia operatorului curent, se trece elementul din vârful stivei în scrierea postfixată;
    - se introduce operatorul curent în stivă.
- se trec toţi operatorii rămaşi pe stivă în scrierea postfixată.

Având expresia sub această formă, se face evaluarea ei. Algoritmul de evaluare foloseşte tot o stivă pe care sunt păstraţi de această dată operanzii. Algoritmul este:

- se iniţializează stiva;
- atât timp cât nu s-a ajuns la sfârşitul scrierii postfixate:
  - se citeşte următorul element;
  - dacă este valoare se depune pe stivă;
  - altfel (este operator):
    - se extrage din stivă elementul y;
    - se extrage din stivă elementul x;
    - se efectuează operaţia x operator y;
    - se depune rezultatul pe stivă;
- ultima valoare care se află pe stivă este rezultatul expresiei.

De exemplu, pentru expresia în scriere postfixată: 4 8 2 3 * - 2 / +, algoritmul se execută ca în figura 11.5:

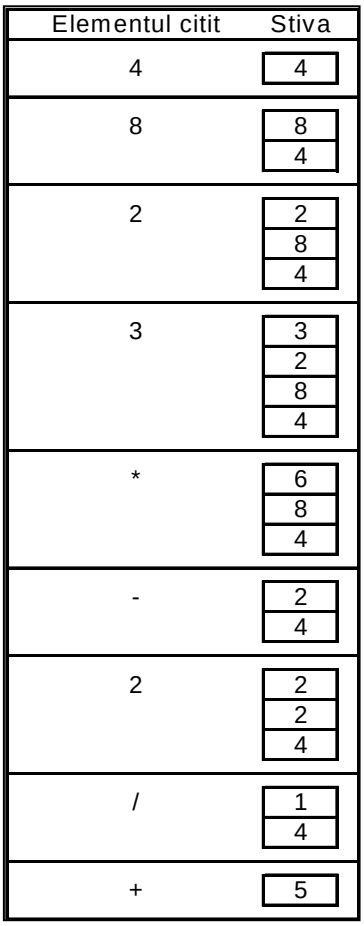

Figura 11.5 *Execuţia algoritmului*

Programul care realizează evaluarea expresiilor matematice prin algoritmii prezentaţi a fost realizat folosind programarea orientată pe obiecte. Astfel există o clasă numită *stiva* care are rolul de a implementa operaţiile cu stiva necesare atât la generarea expresiei în scriere postfixată cât şi la evaluarea expresiei scrisă în această formă. Acest obiect are la bază clasa *deque* (double ended que – coadă cu 2 capete – listă dublu înlănţuită) care se găseşte în biblioteca standard de C++. Fiind nevoie de două tipuri de stivă (una care păstrează operanzi şi una care păstrează valori reale) s-a construit o clasă stivă template.

Pentru păstrarea formei poloneze inverse se utilizează o structură de date numită coadă care este implementată tot printr-un obiect care are la bază clasa *deque*.

În cadrul acestei clase sunt implementate funcţii care ajută la programarea algoritmului. Există funcţii care verifică dacă primul element din coadă reprezintă o valoare (*eNumar()*), extrag primul element ca valoare (*getNumar()*).

Mai există o clasă *fisier* care are rolul de a parcurge fişierul în care se află expresie aritmetică ce trebuie evaluată.

Codul sursă al programului este:

```
#pragma warning(disable:4786)

#include <deque>
#include <iostream>
#include <string>

using namespace std;

#include "ierarhie.h"
#include "stiva.h"
#include "fisier.h"
#include "coada.h"

////////////////////Variabile globale
fisier f;
stiva <char> OPER;
stiva <double> EVAL;
coada POSTFIX;

void eroare(const char * text,int nr){
      cout<<"\n"<<text<<"\n";
      exit(nr);
}

void scrierePOSTFIX(string &op)
{
      char opc;
      if(op=="")
            return;
      int ierOp=ierarhie(op);
      //daca e valoare, se trece direct in scrierea postfixata
      if(!ierOp)
            POSTFIX.adauga(op);
      else
      {
            opc=op[0];
            switch(ierOp)
            {     //daca e paranteza deschisa, se introduce pe stiva
                  case PD:  //([{
                        OPER.push(opc);
                        break;
            //daca e paranteza inchisa, se extrag toate elementele
            //de pe stiva pana la intalnirea unei paranteze deschise
                  case PI:  //)]}
                        while(ierarhie(OPER.top())!=PD)
                              POSTFIX.adauga(OPER.pop());
                        OPER.pop();
                        break;
            //daca e alt operator, se extrag elemente de pe stiva atat
            //timp cat stiva nu este goala si ierarhia operatorului
            //din varful stivei este mai mare sau egala decat ierahia
            //operatorului curent
            //la sfarsit operatorul curent se depune pe stiva
                  default:

      while((!OPER.eGoala())&&ierarhie(OPER.top())>=ierOp)
```

```
                              POSTFIX.adauga(OPER.pop());
                        OPER.push(opc);
                        break;
            }
      }
}

void evalPOSTFIX()
{
      string op;
      char opc;
      double t1, t2, rez;
      //atat timp cat nu s-a ajuns la sfarsitul expresiei postfixate
      while(!POSTFIX.eGoala())
      {
            //daca elementul curent este numar acesta se depune pe stiva
            while(POSTFIX.eNumar())
                  EVAL.push(POSTFIX.getNumar());
            //se extrag 2 valori de pe stiva
            t2=EVAL.pop();
            t1=EVAL.pop();
            op=POSTFIX.extrage();
            opc=op[0];
            //se efectueaza operatia dintre cele 2 valori
            switch(opc)
            {
                  case '+':
                        rez=t1+t2;
                        break;

                  case '-':
                        rez=t1-t2;
                        break;

                  case '*':
                        rez=t1*t2;
                        break;

                  case '/':
                        rez=t1/t2;
                        break;

                  default:
                        eroare("Operator necunoscut!", 1);

            }
            //rezultatul operatiei se depune pe stiva
            EVAL.push(rez);
      }
}

void main(int argc, char* argv[])
{
      //se primeste ca parametru numele fisierului in care se
      //gaseste expresia matematica ce se doreste a fi evaluata
      string numefis;
      if(argc!=2)
```

```
      {     cout<<"Specificati numele unui fisier pentru care doriti
scrierea postfixata!";
            exit(1);
      }

      numefis=argv[1];

      f.open(numefis.c_str());


      if(f.bad())
      {
            cout<<"Fisierul "<<numefis<<" nu exista.\n";
            exit(2);
      }

      cout<<"Lucrez cu fisierul "<<numefis<<".\n";


      //parcurge fisierul primit ca parametru
      string op;
      while(!f.eof())
      {
            op=f.citeste();
            scrierePOSTFIX(op);
      }
      f.close();

      //se extrag toti operatorii ramasi pe stiva
      while(!OPER.eGoala())
            POSTFIX.adauga(OPER.pop());

      //afiseaza expresia din fisier in forma postfixata
      cout<<"\n\nExpresia in scriere postfixata\n";
      for(int i=0;i<POSTFIX.size();i++)
            cout<<POSTFIX[i]<<" ";

      //efectueaza calculul expresiei
      evalPOSTFIX();

      //afiseaza valoarea expresiei (ultima valoare ramasa pe stiva)
      cout<<"\nValoarea expresiei este: "<<EVAL.pop()<<"\n";
}

//ierarhie.h - informatii despre ierarhia operatorilor
#define PD 1
#define PI 2
#define PLUSMINUS 3
#define INMIMP 4

int ierarhie(char c)
{
      switch (c)
      {
            case '+':
            case '-':
                  return PLUSMINUS;
```

```
                  break;
            case '*':
            case '/':
                  return INMIMP;
                  break;
            case '(':
            case '[':
            case '{':
                  return PD;
                  break;
            case ')':
            case ']':
            case '}':
                  return PI;
                  break;
            default:
                  return 0;
                  break;
      }
};

int ierarhie(string & s)
{
      char c;
      c=s[0];
      return ierarhie(c);
};

/////coada.h - Clasa coada
void eroare(const char * text,int nr);

class coada{

private:
      //implementarea cozii cu o clasa template double ended que din
      //C++ Standard Template Library
      deque <string> s;
public:

      //verifica daca coada e goala
      bool eGoala()
      {
            return s.empty();
      };


      //adauga un element in coada
      void adauga(const string & str)
      {
            s.push_back(str);
      };


      //adauga un element in coada
      void adauga(char c)
      {
            string str;
```

```
            str=c;
            s.push_back(str);
      };

      //intoarce si extrage primul element din coada
      string extrage()
      {
            string t;
            if(!s.empty())
            {
                  t=s.front();
                  s.pop_front();
            }
            else eroare("Eroare de sintaxa.",2);
            return t;
      };

      //verifica daca primul element din coada reprezinta un numar
      bool eNumar()
      {
            char *stop;
            string st=s.front();
            strtod(st.c_str(), &stop);
            return (*stop) == 0;
      };

      //extrage elementul din coada ca numar
      double getNumar()
      {
            char *stop;
            double v;
            string st;
            st=s.front();
            s.pop_front();
            v=strtod(st.c_str(), &stop);
            return v;
      };

      //intoarce a i-lea element din coada (0 - primul element)
      string operator [] (unsigned int i)
      {
            if((i>=0) && (i<s.size()))
                  return s[i];
            eroare("Eroare de sintaxa.",4);
            return "";
      };

      //intoarce numarul de elemente din coada
      int size()
      {
            return s.size();
      };
};
/////stiva.h - Clasa template stiva

void eroare(const char * text,int nr);
```

```
template <class T>
class stiva{

private:
	//implementarea stivei cu o clasa template double ended que din
	//C++ Standard Template Library
	deque <T> s;
public:

	//verifica daca stiva e goala
	bool eGoala()
	{
		return s.empty();
	};

	//introduce un element in stiva
	void push(T str)
	{
		s.push_back(str);
	};

	//intoarce si extrage elementul din stiva
	T pop()
	{
		T t;
		if(!s.empty())
		{
			t=s.back();
			s.pop_back();
		}
		else eroare("Eroare de sintaxa.",5);
		return t;
	};

	//intoarce elementul din stiva
	T top()
	{
		if (!s.empty())
			return s.back();
		eroare("Eroare de sintaxa.",6);
		return NULL;
	};
};

/////fisier.h - clasa fisier

#include <string>
#include <fstream>

using namespace std;

class fisier
{
public:
	//citeste o noua secventa de caractere din fisierul de date
	string citeste();
	//deschide fisierul
```

```
	void open(const char * numefis);
	//verifica daca fisierul a fost deschis cu succes
	bool bad();
	//verifica daca s-a ajuns la sfarsitul fisierului
	bool eof();
	//inchide fisierul deschis
	void close();

private:
	// fisierul din care se citesc datele
	ifstream fis;
	// verifica daca parametrul primit este separator sau operator
	bool isSeparator(string s);
	// verifica daca parametrul primit este separator sau operator
	bool isSeparator(char c);
	//scoate spatiile albe din datele de intrare (tab, spatiu, sfarsit
de linie)
	void eatWhite();
};


/////fisier.cpp - clasa fisier

#include "fisier.h"

// verifica daca parametrul primit este separator sau operator
bool fisier::isSeparator(string s)
{
	char c=s[0];
	return (c=='+') || (c=='-') || (c=='*') || (c=='/') || (c=='(') ||
		(c=='[') || (c=='{') || (c==')') || (c==']') || (c=='}')
||
		(c=='\n') || (c=='\t') || (c==' ');
};

// verifica daca parametrul primit este separator sau operator
bool fisier::isSeparator(char c)
{
	return (c=='+') || (c=='-') || (c=='*') || (c=='/') || (c=='(') ||
		(c=='[') || (c=='{') || (c==')') || (c==']') || (c=='}')
||
		(c=='\n') || (c=='\t') || (c==' ');
}

//scoate spatiile albe din datele de intrare (tab, spatiu, sfarsit de
linie)
void fisier::eatWhite()
{
	while((fis.peek()==' ') || (fis.peek()=='\n') ||
(fis.peek()=='\t'))
		fis.get();
}

//citeste o noua secventa de caractere din fisierul de date
string fisier::citeste()
{
	string rez="";
```

```
	rez=fis.get();

	if(!isSeparator(rez))
		while(!isSeparator(fis.peek()) && !fis.eof())
			rez+=fis.get();

	eatWhite();

	return rez;
}

//deschide fisierul
void fisier::open(const char * numefis)
{
	fis.open(numefis);
	eatWhite();
}

//verifica daca fisierul a fost deschis cu succes
bool fisier::bad()
{
	return fis.bad();
}

//verifica daca s-a ajuns la sfarsitul fisierului
bool fisier::eof()
{
	return fis.eof();
}

//inchide fisierul deschis
void fisier::close()
{
	fis.close();
}
```