Structuri de date

Tema 2

David Andreea Raluca – Grupa 10LF221

1. **Implementarea unei liste dublu înlănţuite**. Să se implementeze o listă dublu înlănţuită cu funcţionalităţile descrise în continuare. Se cere utilizarea unei structuri *node* care are trei câmpuri: un câmp pentru informaţie (de tip int) şi două câmpuri de tip pointer la *node* pentru legăturile către elementele precedent şi următor. Se cere utilizarea unei structuri *List* care are ca membri două variabile de tip *node** reprezentând primul respectiv ultimul element din listă, o variabilă de tip int reprezentând numărul de elemente din listă şi funcţiile:

   - $push\_front(key)$ - adaugă cheia *key* în capul listei (0.20 p)
   - $push\_back(key)$ - adaugă cheia *key* la finalul listei (0.20 p)
   - $pop\_front()$ - şterge primul element din listă (0.20 p)
   - $pop\_back()$ - şterge ultimul element din listă (0.20 p)
   - $find(key)$ - caută o cheie *key* în listă - returnează pointer la nodul cu cheia *key* sau NULL (0.20 p)
   - $erase(node*Nod)$ -şterge un element *Nod* din listă (NU implică căutare). Nodul transmis ca parametru a fost în prealabil găsit cu *find* sau identificat prin parcurgerea listei. (0.20 p)
   - $remove(int\ key)$ - şterge toate apariţiile cheii *key* (implică căutare) (0.20 p)
   - $insert(node*Nod, int\ val)$ - inserează un element cu cheia *val* înainte de nodul indicat de *Nod*. (0.5 p)
   - $empty()$ - verifică dacă lista e vidă (0.20 p)
   - $clear()$ - goleşte lista. (0.20 p)
   - funcţia $print()$ - afişează elementele listei (0.20 p)
   - funcţia $size()$ - returnează numărul de elemente din listă.

   De asemenea să se implementeze următoarele funcţii, care nu fac parte din structură:

   - $palindrom(List\ L)$ - verifică dacă lista este palindrom (0.5 p)
   - $compare(List\ L1, ListL2)$ - returnează 1 dacă $L1$ şi $L2$ sunt identice şi 0 altfel. (0.20 p)

   În funcţia *main* realizaţi un meniu cu ajutorul unei instrucţiuni *switch*, prin care se oferă opţiuni, corespunzătoare fiecărei funcţionalităţi, precum şi o opţiune de EXIT. Într-o instrucţiune *while*, se citesc şi se execută opţiuni până la alegerea opţiunii de EXIT.

   ATENŢIE: Nici o funcţie nu trebuie să dea eroare de execuţie, dacă se apelează pe o listă vidă!!!

```
#include <iostream>

struct node
```

```
{
	int informatie;
	node* prev=nullptr, * next=nullptr;
};

struct List
{
	node* head = nullptr, * tail = nullptr;
	int nrElemente = 0;
	void push_front(int key)
	{
		node* newNode = new node;
		newNode->informatie = key;
		newNode->next = head;
		if (head)
			head->prev = newNode;
		else
			tail = newNode;
		head = newNode;
		nrElemente++;
	}
	void push_back(int key)
	{
		node* newNode = new node;
		newNode->informatie = key;
		newNode->prev = tail;
		if (tail)
			tail->next = newNode;
		else
			head = newNode;
		tail = newNode;
		nrElemente++;
	}
	void pop_front()
	{
		if (!head)
		{
			std::cout << "Lista este vida" << std::endl;
			return;
		}
		node* firstNode = head;
		head = head->next;
		if (head)
			head->prev = nullptr;
		nrElemente--;
		delete firstNode;
	}
	void pop_back()
	{
		if (!tail)
		{
```

```
                std::cout << "Lista este vida" << std::endl;
                return;
        }
        node* lastNode = tail;
        tail = tail->prev;
        if (tail)
                tail->next = nullptr;
        nrElemente--;
        delete lastNode;
}
node* find(int key)
{
        if (!(head) && !(tail))
        {
                std::cout << "Lista este vida!" << std::endl;
                return nullptr;
        }
        node* nodCurent = head;
        while ((nodCurent != nullptr) && (nodCurent->informatie != key))
        {
                nodCurent = nodCurent->next;
        }
        return nodCurent;
}
void erase(node* Nod)
{
        if (!(head) && !(tail))
        {
                std::cout << "Lista este vida!" << std::endl;
                return;
        }
        if (Nod == nullptr)
        {
                std::cout << "Nodul nu exista!" << std::endl;
                return;
        }
        else if (Nod == head)
        {
                pop_front();
                return;
        }
        else if (Nod == tail)
        {
                pop_back();
                return;
        }
        node* nodDeSters = Nod;
        node* nodAnterior = Nod->prev;
        Nod = Nod->next;
        nodAnterior->next = Nod;
        Nod->prev = nodAnterior;
```

```
		nrElemente--;
		delete nodDeSters;
}
void remove(int key)
{
		if (!(head) && !(tail))
		{
			std::cout << "Lista este vida!" << std::endl;
			return;
		}
		node* nodCurent = head;
		node* nodUrmator;
		while (nodCurent != nullptr)
		{
			nodUrmator = nodCurent->next;
			if (nodCurent->informatie == key)
			{
				erase(nodCurent);
			}
			nodCurent = nodUrmator;
		}
}
void insert(node* Nod, int val)
{
		if (Nod == head)
			push_front(val);
		else
		{
		  node* newNode = new node;
		  newNode->informatie = val;
		  newNode->next = Nod;
		  Nod->prev = newNode;
		  node* nodAnterior = Nod->prev;
		  nodAnterior->next = newNode;
		  newNode->prev = nodAnterior;
		  nrElemente++;
	}
}
void print()
{
		if (!(head) && !(tail))
		{
			std::cout << "Lista este vida!" << std::endl;
			return;
		}
		node* curent = head;
		while (curent)
		{
			std::cout << curent->informatie<<' ';
			curent = curent->next;
		}
```

```
		std::cout <<std:: endl;
	}
	void empty()
	{
		if (!(head) && !(tail))
			std::cout << "Lista este vida!" << std::endl;
		else
			std::cout << "Lista nu este vida!" << std::endl;
	}
	int size()
	{
		return nrElemente;
	}
	void clear()
	{
		if (!(head) && !(tail))
		{
			std::cout << "Lista este vida!" << std::endl;
			return;
		}
		while (nrElemente)
		{
			pop_front();
		}
		tail = nullptr;
	}
};

void palindrom(List L)
{
	if (!(L.head) && !(L.tail))
	{
		std::cout << "Lista este vida!" << std::endl;
		return;
	}
	node* lastNode=L.tail;
	node* firstNode=L.head;
	int jumatate = 1;
	while (jumatate<=L.size()/2)
	{
		if (lastNode->informatie != firstNode->informatie)
		{
			std::cout << "Lista nu este palindrom" << std::endl;
			return;
		}
		lastNode = lastNode->prev;
		firstNode = firstNode->next;
		jumatate++;
	}
	std::cout << "Lista este palindrom" << std::endl;
}
```

```
bool compare(List L1, List L2)
{
	node* nodCurent1 = L1.head;
	node* nodCurent2= L2.head;
	if ((nodCurent1 == nullptr) || (nodCurent2 == nullptr))
	{
		std::cout << "Una sau ambele liste sunt vide!" << std::endl;
		return false;
	}
	if (L1.size() != L2.size())
		return false;
	while ((nodCurent1) && (nodCurent2))
	{
		if (nodCurent1->informatie != nodCurent2->informatie)
			return false;
		nodCurent1 = nodCurent1->next;
		nodCurent2 = nodCurent2->next;
	}
	return true;
}

void AfisareComenzi()
{
	std::cout << "Comenzile sunt:" << std::endl;
	std::cout << "1 - adaugare element la inceputul listei" << std::endl;
	std::cout << "2 - adaugare element la sfarsitul listei" << std::endl;
	std::cout << "3 - stergere primul element" << std::endl;
	std::cout << "4 - stergere ultimul element" << std::endl;
	std::cout << "5 - cautare cheie in lista" << std::endl;
	std::cout << "6 - stergere un anumit nod" << std::endl;
	std::cout << "7 - sterge toate aparitiile unei chei" << std::endl;
	std::cout << "8 - inserare element inaintea unui anumit nod" << std::endl;
	std::cout << "9 - verificare lista" << std::endl;
	std::cout << "10 - golire lista" << std::endl;
	std::cout << "11 - afisare lista" << std::endl;
	std::cout << "12 - dimensiune lista" << std::endl;
	std::cout << "13 - verificare palindrom" << std::endl;
	std::cout << "14 - comparare doua liste" << std::endl;
	std::cout << "Orice inafara intervalului [1,14] - EXIT" << std::endl;
	std::cout<<std::endl;
}

int main()
{
	List lista1,lista2;
	int comanda=0,nr=0,nr2=0,dimensiune=0,ok=1;
	AfisareComenzi();
	while (ok!=0)
	{
		std::cout << "Introduceti comanda:" << std::endl;
```

```
		std::cin >> comanda;
		switch (comanda)
		{
		case 1:
			std::cout << "Introduceti elementul pe care vreti sa il adaugati in capul listei:" <<
std::endl;
			std::cin >> nr;
			lista1.push_front(nr);
			break;
		case 2:
			std::cout << "Introduceti elementul pe care vreti sa il adaugati la finalul listei:" <<
std::endl;
			std::cin >> nr;
			lista1.push_back(nr);
			break;
		case 3:
			std::cout << "Ati ales sa stergeti primul element din lista!" << std::endl;
			lista1.pop_front();
			break;
		case 4:
			std::cout << "Ati ales sa stergeti ultimul element din lista!" << std::endl;
			lista1.pop_back();
			break;
		case 5:
			std::cout << "Introduceti elementul pe care vreti sa il cautati in lista:" << std::endl;
			std::cin >> nr;
			if (lista1.find(nr) != nullptr)
				std::cout << "Elementul se afla la nodul " << lista1.find(nr) << std::endl;
			else
				std::cout << "Elementul nu exista!" << std::endl;
			break;
		case 6:
			std::cout << "Introduceti ce element vreti sa fie sters din lista:" << std::endl;
			std::cin >> nr;
			if (lista1.find(nr) != nullptr)
				lista1.erase(lista1.find(nr));
			else
				std::cout << "Elementul nu exista!" << std::endl;
			break;
		case 7:
			std::cout << "Introduceti ce cheie vreti sa fie stearsa din toata lista:" << std::endl;
			std::cin >> nr;
			lista1.remove(nr);
			break;
		case 8:
			std::cout << "Introduceti ce element vreti sa introduceti si unde:" << std::endl;
			std::cin >> nr >> nr2;
			lista1.insert(lista1.find(nr2), nr);
			break;
		case 9:
			lista1.empty();
```

```
                break;
            case 10:
                std::cout << "Ati ales sa goliti lista!" << std::endl;
                lista1.clear();
                break;
            case 11:
                lista1.print();
                break;
            case 12:
                std::cout << "Lista are " << lista1.size() << " elemente" << std::endl;
                break;
            case 13:
                palindrom(lista1);
                break;
            case 14:
                std::cout << "Introduceti a doua lista si dimensiunea sa:" << std::endl;
                std::cin >> dimensiune;
                for (int index = 0; index < dimensiune; index++)
                {
                    std::cin >> nr;
                    lista2.push_back(nr);
                }
                if (compare(lista1, lista2))
                    std::cout << "Listele sunt identice!" << std::endl;
                else
                    std::cout << "Listele nu sunt identice!" << std::endl;
                break;
            default:
                ok = 0;
                std::cout << "EXIT" << std::endl;
                break;
            }
        }
        return 0;
}
```

2. **Algoritmul Bucket-Sort**. Scrieţi o funcţie **Bucket-Sort**, care are ca parametru un vector de nr numere reale din intervalul $[0, 1)$, repartizate uniform în acest interval. Algoritmul sortează vectorul folosind metoda $Bucket - Sort$ descrisă în Cormen. Pentru $bucket$-uri folosiţi liste. În funcţia main citiţi din fişier un vector de numere cu valori din intervalul $[0, 1)$ pe care apoi îl sortaţi cu funcţia implementată şi îl afişaţi pe ecran. (1p)

```
#include <iostream>
#include <vector>
#include <fstream>

struct node
{
```

```
	float informatie;
	node* next = nullptr,* prev = nullptr;
};

struct List
{
	int nrElemente;
	node* head = nullptr, * tail = nullptr;
	void push_backk(float valoare)
	{
		node* newNode = new node;
		newNode->informatie = valoare;
		newNode->prev=tail;
		if (tail)
			tail->next = newNode;
		else
			head = newNode;
		tail = newNode;
		nrElemente++;
	}
};

void BucketSort(std::vector<float>& vector)
{
	std::vector<List> vector2(vector.size());
	for (int index = 0; index < vector.size(); index++)
	{
		vector2[int(vector.size() * vector[index])].push_backk(vector[index]);
	}
	node* nodCurent, * nodCurent2;
	for (int index = 0; index < vector.size(); index++)
	{
		nodCurent = vector2[index].head;
		while (nodCurent!=nullptr)
		{
			nodCurent2 = nodCurent->next;
			while (nodCurent2!=nullptr)
			{
				if (nodCurent->informatie > nodCurent2->informatie)
				{
					std::swap(nodCurent->informatie, nodCurent2->informatie);
				}
				nodCurent2 = nodCurent2->next;
			}
			nodCurent = nodCurent->next;
		}
	}
	int dimensiune = vector.size();
	vector.clear();
	for (int index = 0; index < dimensiune; index++)
	{
```

```
                nodCurent = vector2[index].head;
                while (nodCurent)
                {
                        vector.push_back(nodCurent->informatie);
                        nodCurent = nodCurent->next;
                }
        }
}

void AfisareVector(std::vector<float> vector)
{
        for (int index = 0; index < vector.size(); index++)
                std::cout << vector[index] << ' ';
}

int main()
{
        std::vector<float> vector;
        int nr;
        std::ifstream fin("Fisier.in");
        fin >> nr;
        float element;
        for(int index = 0; index < nr; index++)
        {
                fin >> element;
                vector.push_back(element);
        }
        BucketSort(vector);
        AfisareVector(vector);
        fin.close();
        return 0;
}
```

3. **Implementare coadă**. Să se implementeze o coadă utilizând liste înlănţuite. Vă trebuie:

   - o structură *node* cu două câmpuri - un câmp pentru informaţie (de tipul cerut de problema curentă) şi un câmp de tip pointer la *node* pentru legătura la elementul următor.
   - o structură *Queue* cu
     - două câmpuri de tip pointer la nod, pentru primul şi ultimul element iniţializate cu NULL (nullptr).
     - un câmp *nr_elem* de tip int - numărul de elemente din coada.
     - funcţia *push*(*elem*) - pune *elem* la sfarsitul cozii
     - funcţia *pop*() - elimină elementul de la începutul cozii
     - funcţile *front*() şi *back*() returnează primul respectiv ultimul element din coadă
     - funcţia *empty*() - verifică dacă coada este vidă.
     - funcţia *clear*() - goleşte coada
     - funcţia *size*() - returnează numărul de elemente din coada

   Această coadă va fi utilizată în următoarea problemă:

La un examen se pot prezenta candidaţi pe durata a două zile. În fiecare zi timpul alocat pentru examinare este de t ore ($t \leq 6$). La examen se înscriu n candidaţi. Se citesc din fişier t, n precum şi candidaţii cu numele (de tip **std::string**). Ei vor fi introduşi într-o coadă, de unde vor fi extraşi pe rând pentru examinare. Pentru fiecare candidat, care este la rând, se generează o durată aleatorie cu o valoare între 5 minute şi 15 minute. În momentul în care timpul t s-a terminat, deci se încheie prima zi de evaluare, candidaţii care au rămas în coada vor fi extraşi pe rând şi trecuţi într-un fişier de ieşire, care va reprezenta lista candidaţilor pentru ziua a doua de examinare.

Punctajul pentru problemă: 1p pentru implementarea cozii + 1p pentru rezolvarea problemei. Folosiţi nume semnificative pentru variabilele folosite (chiar dacă în enunţ s-au folosit denumiri precum t şi n)!

```
#include <iostream>
#include <fstream>
#include <time.h>
#include <stdlib.h>

struct node
{
	std::string nume;
	node* next;
};

struct Queue
{
	node* firstNode = nullptr;
	node* lastNode = nullptr;
	int nr_elem=0;
	void push(std::string elem)
	{
		node* newNode = new node;
		newNode->nume = elem;
		newNode->next = nullptr;

		if (lastNode)
			lastNode->next = newNode;
		else
			firstNode = newNode;
		lastNode = newNode;
		nr_elem++;
	}
	void pop()
	{
		if (firstNode == nullptr)
		{
			std::cout << "Coada este vida!" << std::endl;
			return;
		}
		node* nodDeSters = firstNode;
		firstNode = firstNode->next;
```

```
		if (firstNode == nullptr)
			lastNode = nullptr;
		delete nodDeSters;
		nr_elem--;
	}
	std::string front()
	{
		if (firstNode == nullptr)
		{
			std::cout << "Coada este vida!" << std::endl;
			return "Nu avem cadidati";
		}
		return firstNode->nume;
	}
	std::string back()
	{
		if (lastNode == nullptr)
		{
			std::cout << "Coada este vida!" << std::endl;
			return "Nu avem cadidati!";
		}
		return lastNode->nume;
	}
	void empty()
	{
		if ((firstNode == nullptr) && (lastNode == nullptr))
			std::cout << "Coada este vida!" << std::endl;
		else
			std::cout << "Coada nu este vida!" << std::endl;
	}
	void clear()
	{
		node* curent = firstNode;
		while (curent)
		{
			curent = curent->next;
			pop();
		}
	}
	int size()
	{
		return nr_elem;
	}
};

void CitireDate(Queue& coada,int& nrCandidati,int& timp)
{
	std::ifstream fin("Fisier.in");
	fin >> timp >> nrCandidati;
	std::string candidat;
	for (int index = 0; index < nrCandidati; index++)
```

```
	{
		fin >> candidat;
		coada.push(candidat);
	}
	fin.close();
}

void TrecereTimp(Queue coada,int timp)
{
	timp = timp * 60;
	int timpCandidat;
	node* candidatCurent = coada.firstNode;
	srand((unsigned)time(NULL));
	while ((candidatCurent!=nullptr) && (timp>0))
	{
		timpCandidat = rand() % 11 + 5;
		std::cout << timpCandidat << ' ' << candidatCurent->nume << std::endl;
		if (timp - timpCandidat >= 0)
		{
			timp = timp - timpCandidat;
			coada.pop();
			candidatCurent = coada.firstNode;
		}
		else
			timp = 0;
	}
	if (candidatCurent == nullptr)
		std::cout << "S-au examinat toti candidatii in prima zi" << std::endl;
	else
	{
		std::ofstream fout("Fisier.out");
		while (candidatCurent)
		{
			fout << candidatCurent->nume << std::endl;
			candidatCurent = candidatCurent->next;
		}
	fout.close();
	}
}

int main()
{
	Queue coada;
	int nrCandidati, timp;
	CitireDate(coada, nrCandidati,timp);
	TrecereTimp(coada, timp);
	return 0;
}
```

4. **Parantezare corectă**: Se dă un şir de paranteze deschise şi închise de tip (,), [,], {, }. Să se verifice dacă şirul este corect. Folosiţi o stiva (std::stack) pentru rezolvare. **Exemplu**: şirul [()()] este corect, şirul ([]) nu este corect, şirul ()] (nu este corect. (1p)

```
#include <iostream>
#include <stack>

bool verificareSir(std::string paranteze)
{
	std::stack<char> stiva;
	for (char caracter : paranteze)
	{
		if (caracter == '{')
		{
			if (!(stiva.empty()))
				return false;
			stiva.push(caracter);
		}
		else if(caracter == '}')
		{
			if ((stiva.empty())||(stiva.top()!='{'))
				return false;
			stiva.pop();
		}
		else if (caracter == '[')
		{
			if (!(stiva.empty())&&(stiva.top() == '('))
				return false;
			stiva.push(caracter);
		}
		else if(caracter==']')
		{
			if ((stiva.empty())|| (stiva.top()!='['))
				return false;
			stiva.pop();
		}
		else if (caracter == '(')
			stiva.push(caracter);
		else
		{
			if ((stiva.empty())||(stiva.top() != '('))
				return false;
			stiva.pop();
		}
	}
	if (stiva.empty())
		return true;
	return false;
```

```
}

int main()
{
	std::string paranteze;
	std::cin >> paranteze;
	if (verificareSir(paranteze))
		std::cout << "Este corect!" << std::endl;
	else
		std::cout << "Nu este corect!" << std::endl;
	return 0;
}
```

7. **Evaluarea expresiilor aritmetice**. Se citeşte dintr-un fişier un şir de expresii aritmetice alcătuite din numere întregi fără semn, operatorii aritmetici $+$, $-$, $*$, $/$, $\hat{}$ (reprezentând ridicare la putere) şi paranteze rotunde. Expresiile pot fi despărţite între ele prin ; sau prin trecere la rând nou. **Atenţie:** algoritmul trebuie să funcţioneze corect şi dacă există caractere "albe" în expresiile considerate (spaţii, tab-uri) şi dacă NU există! Spaţiile albe se ignoră la analizarea acestor expresii.

   Pentru fiecare expresie:

   - Să se afişeze întâi expresia aritmetică.
   - Să se construiască forma poloneză postfixată şi să se afişeze. (1p)
   - Să se evalueze expresia şi să se afişeze rezultatul. (1p)

   - Să se semnaleze erori în expresie (de parantezare, de operatori, de caractere nepermise). Dacă într-o expresie se găsesc erori se întrerupe analiza/evaluarea acesteia după mesajul de eroare şi se trece la următoarea expresie. (1p)

   Pentru citirea şi procesarea conform enunţului a mai multor expresii dintr-un fişier - 0.5p.

   Puncte suplimentare

   - Pentru funcţionarea algoritmului şi cu numere întregi de mai multe cifre (1p)
   - Pentru funcţionarea algoritmului şi cu numere reale (1p)

   Utilizaţi stive din STL. Folosiţi funcţii separate pentru construirea formei poloneze pentru o expresei aritmetică stocată într-un **string** şi pentru evaluarea pornind de la o formă poloneză.

```
#include <iostream>
#include <stack>
```

```
#include <fstream>
#include <string>
#include <vector>

void CitireExpresii(std::stack<std::string>& expresii)
{
	std::ifstream fin("Fisier.in");
	std::string expresi,expresie;
	while (!fin.eof())
	{
		std::getline(fin,expresi);
		for (char caracter : expresi)
		{
			if (caracter == ';')
			{
				if(!expresie.empty())
				expresii.push(expresie);
				expresie.clear();
			}
			else if((caracter!=' ')&&(caracter!=9))
			{
				expresie.push_back(caracter);
			}
		}
		if (!expresie.empty())
		{
			expresii.push(expresie);
			expresie.clear();
		}
		expresi.clear();
	}
}

int Prioritate(char caracter)
{
	if (caracter == '(')
		return 0;
	else if ((caracter == '-') || (caracter == '+'))
		return 1;
	else if ((caracter == '*') ||(caracter == '/'))
		return 2;
	else if(caracter=='^')
	   return 3;
}

void FormaPoloneza(std::stack<char>& formaPoloneza,std::string expresie, std::vector<int>& numere)
{
	std::stack<char>operatii;
	double numar = 0;
	int ant = 0;
	for (char caracter:expresie)
```

```
	{
		if (('0' <= caracter) && (caracter <= '9'))
		{
		  formaPoloneza.push(caracter);
		  numar = numar * 10 + int(caracter - 48);
		  ant = 1;
		}
		else
		{
			if (ant == 1)
			{
				numere.push_back(numar);
				numar = 0;
				ant = 0;
			}
			if (caracter == '(')
			{
				operatii.push(caracter);
			}
			else
			{
				if (caracter == ')')
				{
					while (!(operatii.empty()) && (operatii.top() != '('))
					{
						formaPoloneza.push(operatii.top());
						operatii.pop();
					}
					if (!(operatii.empty()))
						operatii.pop();
				}
				else
				{
					while (!(operatii.empty()) &&
(Prioritate(operatii.top())>=Prioritate(caracter)))
					{
						formaPoloneza.push(operatii.top());
						operatii.pop();
					}
					operatii.push(caracter);
				}
			}
		}
	}
	if(ant==1)
	 numere.push_back(numar);
	while (!(operatii.empty()))
	{
		formaPoloneza.push(operatii.top());
		operatii.pop();
	}
```

```
}

std::string CreareFormaPoloneza(std::stack<char>& formaPoloneza)
{
	std::string expresieNoua;
	char parte = formaPoloneza.top();
	while (!(formaPoloneza.empty()))
	{
		expresieNoua.push_back(parte);
		formaPoloneza.pop();
		if(!(formaPoloneza.empty()))
		parte = formaPoloneza.top();
	}
	reverse(expresieNoua.begin(), expresieNoua.end());
	return expresieNoua;
}

int nrCifre(int numar)
{
	if (numar == 0)
		return 1;
	int nr = 0;
	while (numar != 0)
	{
		nr++;
		numar = numar / 10;
	}
	return nr;
}

bool EvaluareExpresie(std::string expresie,double& rezultat,std::vector<int>& numere)
{
	std::stack<double> calcul;
	int nr = 0;
	for (char caracter : expresie)
	{
		if (('0' <= caracter) && (caracter <= '9'))
		{
			if (nr == 0)
			{
				calcul.push(numere[0]);
				nr = nrCifre(numere[0]);
				numere.erase(numere.begin());
			}
				nr--;
		}
		else if ((caracter == '^') || (caracter == '*') || (caracter == '/') || (caracter == '+') || (caracter == '-
'))
		{
			if (calcul.empty())
			{
```

```
                                        std::cout << "Expresia are prea multi operatori!" << std::endl;
                                        return false;
                                }
                                rezultat = calcul.top();
                                calcul.pop();
                                if (calcul.empty())
                                {
                                        std::cout << "Expresia are prea multi operatori!" << std::endl;
                                        return false;
                                }
                                if (caracter == '^')
                                {
                                        rezultat = std::pow(calcul.top(),rezultat);
                                }
                                else if (caracter == '*')
                                {
                                        rezultat = calcul.top()* rezultat;
                                }
                                else if (caracter == '/')
                                {
                                        rezultat = calcul.top() / rezultat;
                                }
                                else if (caracter == '+')
                                {
                                        rezultat = calcul.top() + rezultat;
                                }
                                else if (caracter == '-')
                                {
                                        rezultat = calcul.top() - rezultat;
                                }
                                calcul.pop();
                                calcul.push(rezultat);
                        }
                        else if ((caracter == ')') || (caracter == '('))
                        {
                                std::cout << "Expresia are paranteze eronate!" << std::endl;
                                return false;
                        }
                        else
                        {
                                std::cout << "Expresia are caractere nepermise!" << std::endl;
                                return false;
                        }
                }
                calcul.pop();
                return true;
}

void ParcurgereExpresii(std::stack<std::string>expresii)
{
        std::string expresie,expresieNoua;
```

```
	double rezultat;
	while (!(expresii.empty()))
	{
		expresie = expresii.top();
		std::cout << "Expresia este:" << std::endl;
		std::cout << expresie << std::endl;
		std::stack<char> formaPoloneza;
		std::vector<int> numere;
		FormaPoloneza(formaPoloneza, expresie,numere);
		expresieNoua=CreareFormaPoloneza(formaPoloneza);
		rezultat = 0;
		if (EvaluareExpresie(expresieNoua,rezultat,numere))
		{
			std::cout << "Forma poloneza postfixata este:" << std::endl;
			std::cout << expresieNoua << std::endl;
			std::cout << "Rezultatul este: " << rezultat << std::endl;
		}
		expresieNoua.clear();
		expresii.pop();
		std::cout << std::endl;
	}
}

int main()
{
	std::stack<std::string> expresii;
	CitireExpresii(expresii);
	ParcurgereExpresii(expresii);
	return 0;
}
```