Structuri de date

Tema 4

David Andreea Raluca – Grupa 10LF221

1. **Arbore sintactic.** Se citeşte din fişier o expresie aritmetică formată din numere, variabile şi operatorii de bază (+, -, *, /) şi paranteze.

   a. Să se construiască un arbore sintactic corespunzător expresiei. (2p)

   b. Să se afişeze arborele pe niveluri.(0.5p)

```
#include <iostream>
#include <fstream>
#include <vector>
#include <stack>
#include <string>
#include <queue>

class Nod
{
public:
	char informatie;
	Nod* parinte = nullptr;
	Nod* fiuSt = nullptr;
	Nod* fiuDr = nullptr;
};

class ArboreSintactic
{
	int nrNoduri = 0;
	Nod* radacina = new Nod;
public:
	void setRadacina(char informatie)
	{
		radacina->informatie = informatie;
	}
	Nod* getRadacina()
	{
		return radacina;
	}
	void Inserare(char informatie,Nod* parinte)
	{
		Nod* NewNode = new Nod;
		NewNode->informatie = informatie;
		NewNode->parinte = parinte;
		if (parinte->fiuDr == nullptr)
			parinte->fiuDr= NewNode;
```

```
                else
                        parinte->fiuSt = NewNode;
                nrNoduri++;
        }
};

void CitireDate(std::string& expresie)
{
        std::ifstream fin("Fisier.in");
        fin >> expresie;
        fin.close();
}

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

void FormareFormaPoloneza(std::string& formaPoloneza,std::string expresie)
{
        std::stack<char>operatii;
        std::stack<char>fPoloneza;
        for (char caracter : expresie)
        {
                if(('0' <= caracter) && (caracter <=
'9'))||('a'<=caracter)&&(caracter<='z'))||('A'<=caracter)&&(caracter<='Z'))
                {
                        fPoloneza.push(caracter);
                }
                else
                {
                        if (caracter == '(')
                        {
                                operatii.push(caracter);
                        }
                        else
                        {
                                if (caracter == ')')
                                {
                                        while (!(operatii.empty()) && (operatii.top() != '('))
                                        {
                                                fPoloneza.push(operatii.top());
                                                operatii.pop();
                                        }
                                        if (!(operatii.empty()))
                                                operatii.pop();
```

```
                                        }
                                        else
                                        {
                                                while (!(operatii.empty()) && (Prioritate(operatii.top()) >=
Prioritate(caracter)))
                                                {
                                                        fPoloneza.push(operatii.top());
                                                        operatii.pop();
                                                }
                                                operatii.push(caracter);
                                        }
                                }
                        }
                }
                while (!(operatii.empty()))
                {
                        fPoloneza.push(operatii.top());
                        operatii.pop();
                }
                char caracterCurent = fPoloneza.top();
                while (!(fPoloneza.empty()))
                {
                        formaPoloneza.push_back(caracterCurent);
                        fPoloneza.pop();
                        if (!(fPoloneza.empty()))
                                caracterCurent = fPoloneza.top();
                }
}

void ConstruireArbore(std::string formaPoloneza,ArboreSintactic& arbore)
{
        arbore.setRadacina(formaPoloneza[0]);
        Nod* nodCurent = arbore.getRadacina();
        for (int index = 1; index < formaPoloneza.size(); index++)
        {
                while((nodCurent->fiuDr) && (nodCurent->fiuSt))
                {
                        nodCurent = nodCurent->parinte;
                }
                if (!(nodCurent->fiuDr))
                {
                        arbore.Inserare(formaPoloneza[index], nodCurent);
                        if ((formaPoloneza[index] == '+') || (formaPoloneza[index] == '-') ||
(formaPoloneza[index] == '/') || (formaPoloneza[index] == '*'))
                        {
                                nodCurent = nodCurent->fiuDr;
                        }
                }
                else if (!(nodCurent->fiuSt))
                {
                        arbore.Inserare(formaPoloneza[index], nodCurent);
```

```
                        if ((formaPoloneza[index] == '+') || (formaPoloneza[index] == '-') ||
(formaPoloneza[index] == '/') || (formaPoloneza[index] == '*'))
                        {
                                nodCurent = nodCurent->fiuSt;
                        }
                }
        }
}

void AfisarePeNivele(Nod* radacina)
{
        std::queue<Nod*> Coada;
        Nod* nodCurent;
        Coada.push(radacina);
        while (!(Coada.empty()))
        {
                int nrNoduri = Coada.size();
                for (int index = 0; index < nrNoduri; index++)
                {
                        nodCurent = Coada.front();
                        Coada.pop();
                        std::cout << nodCurent->informatie << " ";
                        if (nodCurent->fiuSt != nullptr)
                                Coada.push(nodCurent->fiuSt);
                        if (nodCurent->fiuDr != nullptr)
                                Coada.push(nodCurent->fiuDr);
                }
                std::cout << std::endl;
        }
}

int main()
{
        std::string expresie,formaPoloneza;
        ArboreSintactic arbore;
        CitireDate(expresie);
        FormareFormaPoloneza(formaPoloneza,expresie);
        ConstruireArbore(formaPoloneza,arbore);
        AfisarePeNivele(arbore.getRadacina());
}
```

4. **Sortare** Să se implementeze algoritmul **Heap-Sort**. Să se sorteze un vector de numere. (1p)

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>

void CitireDate(std::vector<int>& vector, int& n)
```

```
{
	std::cin >> n;
	int element;
	for (int index = 0; index < n; index++)
	{
		std::cin >> element;
		vector.push_back(element);
	}
}

void CreareHeap(std::vector<int> vector, std::priority_queue<int, std::vector<int>, std::greater<int>>&
heapMax)
{
	for (int index = 0; index < vector.size(); index++)
		heapMax.push(vector[index]);
}

void HeapSort(std::vector<int>& vector, std::priority_queue<int, std::vector<int>, std::greater<int>>&
heapMin)
{
	vector.clear();
	while (!heapMin.empty())
	{
		vector.push_back(heapMin.top());
		heapMin.pop();
	}
}

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

int main()
{
	std::vector<int> vector;
	int n;
	std::priority_queue<int,std::vector<int>,std::greater<int>> heapMin;
	CitireDate(vector, n);
	CreareHeap(vector, heapMin);
	HeapSort(vector,heapMin);
	AfisareVector(vector);
	return 0;
}
```

5. **Prority queue**. Implementaţi o coadă de priorităţi folosind o structură (clasă) PRIORITY_QUEUE, care să aibă un câmp DATA de tip vector de întregi, care să stocheze elementele cozii sub forma unui heap max şi un câmp SIZE - nr. de elemente stocate în coadă. În plus structura trebuie să aibă metodele:

   - INSERT - inserează un nou nod în coadă
   - EXTRACT_MAX - extrage elementul de prioritate maximă din coadă
   - MAX_ELEMENT - returnează elementul de prioritate maximă
   - INCREASE_KEY - creşte prioritatea unui nod
   - MAX_HEAPFY (sau SIFT_DOW) - funcţia care coboară o cheie pe poziţia corespunzătoare din heap

   În funcţia *main* se declară o variabilă de tip PRIORITY_QUEUE şi se foloseşte un *menu* implementat cu ajutorul unei instrucţiuni *switch*, prin care utilizatorul să poată selecta oricare dintre operaţiile de inserţie, extragerea maximului, obţinerea maximului şi afişarea elementelor din heap. (2p)

```
#include <iostream>

class PRIORITY_QUEUE
{
	int* DATA;
	int SIZE=0;
	int CAPACITY = 10;
public:
	PRIORITY_QUEUE()
	{
		DATA = new int[CAPACITY];
	}

	void MAX_HEAPFY(int poz)
	{
		int st = 2 * poz + 1;
		int dr = 2 * poz + 2;
		int pozMax = poz;
		if ((st < SIZE) && (DATA[st] > DATA[pozMax]))
			pozMax = st;
		if ((dr < SIZE) && (DATA[dr] > DATA[pozMax]))
			pozMax = dr;
		if (pozMax != poz)
		{
			std::swap(DATA[poz],DATA[pozMax]);
			MAX_HEAPFY(pozMax);
		}
	}
	void EXTRACT_MAX()
	{
```

```
		DATA[0] = DATA[SIZE - 1];
		SIZE = SIZE - 1;
		MAX_HEAPFY(0);
}
int MAX_ELEMENT()
{
		return DATA[0];
}
void INCREASE_KEY(int poz,int value)
{
		int p;
		if (value > DATA[poz])
		{
			DATA[poz] = value;
			p = (poz - 1) / 2;
			while ((poz > 0) && (DATA[p] < value))
			{
				DATA[poz] = DATA[p];
				poz = p;
				p = (poz - 1) / 2;
			}
			DATA[poz] = value;
		}
}
void RESIZE()
{
		CAPACITY = CAPACITY * 2;
		int* newDATA = new int[CAPACITY];
		for (int index = 0; index < SIZE; index++)
		{
			newDATA[index] = DATA[index];
		}
		delete[] DATA;
		DATA = newDATA;
}
void INSERT(int value)
{
		if (SIZE == CAPACITY)
			RESIZE();
		DATA[SIZE] = 0;
		SIZE++;
		INCREASE_KEY(SIZE - 1, value);
}
void PRINT()
{
		for (int index = 0; index < SIZE; index++)
			std::cout << DATA[index] << ' ';
}

~PRIORITY_QUEUE()
{
```

```
        delete[] DATA;
    }
};

void InterfataComenzi()
{
    std::cout << "Comenzile disponibile sunt:" << std::endl;
    std::cout << "1 - INSERT" << std::endl;
    std::cout << "2 - EXTRACT_MAX" << std::endl;
    std::cout << "3 - MAX_ELEMENT" << std::endl;
    std::cout << "4 - INCREASE_KEY" << std::endl;
    std::cout << "5 - PRINT" << std::endl;
    std::cout << "Orice numar inafara intervalului [1,5] - EXIT" << std::endl;
    std::cout << std::endl;
}

int main()
{
    PRIORITY_QUEUE Heap;
    int comanda,valoare,pozitie;
    bool ok=true;
    InterfataComenzi();
    while (ok)
    {
        std::cout << "Introduceti comanda:" << std::endl;
        std::cin >> comanda;
        switch (comanda)
        {
          case 1:
          {
              std::cout << "Introduceti ce nod doriti sa il inserati:" << std::endl;
              std::cin >> valoare;
              Heap.INSERT(valoare);
              break;
          }
          case 2:
          {
              std::cout << "Ati ales sa extrageti elementul cu prioritatea maxima!" << std::endl;
              Heap.EXTRACT_MAX();
              break;
          }
          case 3:
          {
              std::cout << "Elementul cu prioritatea maxima este " << Heap.MAX_ELEMENT() <<
std::endl;
              break;
          }
          case 4:
          {
              std::cout << "Introduceti pozitia si valoarea nodului pe care doriti sa ii cresteti
prioritatea:" << std::endl;
```

```
                    std::cin >>pozitie>>valoare;
                Heap.INCREASE_KEY(pozitie, valoare);
                    break;
                }
                case 5:
                {
                    std::cout << "Ati ales sa afisati elementele heap-ului!" << std::endl;
                    Heap.PRINT();
                    break;
                }
                default:
                {
                    ok = false;
                    break;
                }
            }
            std::cout << std::endl;
        }
        return 0;
}
```

6. **Codificarea Huffmann**. Se citeşte un text dintr-un fişier. Să se construiască arborele de codificare Huffmann corespunzător. Să se afişeze codul corespunzător fiecărui caracter şi să se codifica textul. **std::priority_queue** (min). (3p)

```
#include <iostream>
#include <fstream>
#include <string>
#include <unordered_map>
#include <queue>
#include <functional>

class Nod
{
public:
    Nod* fiuSt=nullptr, *fiuDr=nullptr;
    std::pair<char, int> informatie;
    Nod(char caracter,int frecventa)
    {
        informatie.first = caracter;
        informatie.second = frecventa;
    }
};

class Comparator
{
public:
    bool operator()(Nod*& nr1, Nod*& nr2) const
    {
```

```
		return nr1->informatie.second > nr2->informatie.second;
	}
};

void CitireText(std::string& text)
{
	std::ifstream fin("Fisier.in");
	std::getline(fin, text);
	fin.close();
}

void AflareFrecventaCaractere(std::string& text, std::unordered_map<char, int>& frecventa)
{
	for (char caracter : text)
	{
		if (frecventa.find(caracter) == frecventa.end())
			frecventa.insert({ caracter, 1 });
		else
			frecventa.find(caracter)->second++;
	}
}

void CreareHeapMin(std::unordered_map<char, int>& frecventa, std::priority_queue<Nod*, std::vector<Nod*>,
Comparator>& heapMin)
{
	for (auto iterator : frecventa)
	{
		Nod* newNode=new Nod(iterator.first,iterator.second);
		heapMin.push(newNode);
	}
}

void CreareArboreHuffmann(std::priority_queue<Nod*, std::vector<Nod*>, Comparator>& heapMin)
{
	while (heapMin.size() > 1)
	{
		Nod* Nod1 = heapMin.top();
		heapMin.pop();
		Nod* Nod2 = heapMin.top();
		heapMin.pop();
		Nod* newNode = new Nod('*', Nod1->informatie.second + Nod2->informatie.second);
		newNode->fiuSt = Nod1;
		newNode->fiuDr = Nod2;
		heapMin.push(newNode);
	}
}

void ParcurgereArboreHuffmann(std::priority_queue<Nod*, std::vector<Nod*>, Comparator>& heapMin,Nod*
nodCurent,std::string prefix, std::unordered_map<char,std::string>& codificare)
{
	if (nodCurent == nullptr)
```

```
		return;
	if (nodCurent->informatie.first != '*')
	{
		std::cout << nodCurent->informatie.first << " = " << prefix << std::endl;
		codificare.insert({nodCurent->informatie.first,prefix });
	}
	ParcurgereArboreHuffmann(heapMin, nodCurent->fiuSt, prefix+'0',codificare);
	ParcurgereArboreHuffmann(heapMin, nodCurent->fiuDr, prefix+'1',codificare);
}

void CodificareText(std::unordered_map<char, std::string> codificare, std::string text)
{
	std::cout << std::endl;
	std::cout << "Textul codificat este:" << std::endl;
	for (char caracter : text)
	{
		std::cout << codificare.find(caracter)->second << ". ";
	}
}

int main()
{
	std::string text;
	std::unordered_map<char, int> frecventa;
	std::unordered_map<char, std::string> codificare;
	std::priority_queue<Nod*, std::vector<Nod*>,Comparator> heapMin;
	CitireText(text);
	AflareFrecventaCaractere(text, frecventa);
	CreareHeapMin(frecventa, heapMin);
	CreareArboreHuffmann(heapMin);
	std::string prefix ="";
	std::cout << "Codul corespunzator fiecarui caracter este:" << std::endl;
	ParcurgereArboreHuffmann(heapMin, heapMin.top(),prefix,codificare);
	CodificareText(codificare, text);
	return 0;
}
```