Pb1:

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

class Node {
public:
	char val;
	Node* st, * dr;
	Node(char v) {
		val = v;
		st = nullptr;
		dr = nullptr;
	}
};

void formaPoloneza(std::string expr, std::vector<char>& vector) {
	std::stack<char> stiva;
	for (int i = 0; i < expr.length(); i++) {
		if (isdigit(expr[i]))
			vector.push_back(expr[i]);
		else {
			if (stiva.empty() || expr[i] == '(')
				stiva.push(expr[i]);
			else {
				if (stiva.top() == '+') {
					if (expr[i] == '-') {
						vector.push_back(stiva.top());
						stiva.pop();
					}
					if (expr[i] != ')')
						stiva.push(expr[i]);
				}
				else if (stiva.top() == '-') {
					if (expr[i] == '+') {
						vector.push_back(stiva.top());
						stiva.pop();
					}
					if (expr[i] != ')')
						stiva.push(expr[i]);
				}

				else if (stiva.top() == '*') {
					if (expr[i] == '+' || expr[i] == '-' || expr[i] == '/') {
						vector.push_back(stiva.top());
						stiva.pop();
						if (stiva.top() == '+' || stiva.top() == '-') {
							vector.push_back(stiva.top());
							stiva.pop();
						}
					}
					if (expr[i] != ')')
						stiva.push(expr[i]);

				}
				else if (stiva.top() == '/') {
```

```
                                        if (expr[i] == '+' || expr[i] == '-' || expr[i] == '*') {
                                                vector.push_back(stiva.top());
                                                stiva.pop();
                                                if (stiva.top() == '+' || stiva.top() == '-') {
                                                        vector.push_back(stiva.top());
                                                        stiva.pop();
                                                }
                                        }
                                        if (expr[i] != ')')
                                                stiva.push(expr[i]);
                                }

                                else if (stiva.top() == '(') {
                                        stiva.push(expr[i]);
                                }

                                if (expr[i] == ')') {
                                        while (stiva.top() != '(') {
                                                vector.push_back(stiva.top());
                                                stiva.pop();
                                        }
                                        stiva.pop();
                                }

                        }
                }

        }
        while (!stiva.empty()) {
                vector.push_back(stiva.top());
                stiva.pop();
        }

        for (int i = 0; i < vector.size(); i++) {
                std::cout << vector[i];
        }
        std::cout << std::endl;
}

Node* buildTree(std::vector<char>expr) {
        std::stack<Node*> stack;
        for (int i = 0; i < expr.size(); i++) {
                if (isdigit(expr[i])) {
                        Node* n = new Node(expr[i]);
                        stack.push(n);
                }
                else {
                        Node* n = new Node(expr[i]);
                        n->dr = stack.top();
                        stack.pop();
                        n->st = stack.top();
                        stack.pop();
                        stack.push(n);
                }
        }
        return stack.top();
}

void afisare(Node* root) {
```

```
	if (root == NULL)
		return;
	std::queue<Node*>q;
	q.push(root);
	while (!q.empty()) {
		int size = q.size();
		for (int i = 0; i < size; i++) {
			Node* current = q.front();
			std::cout << current->val << " ";
			q.pop();
			if (current->st != nullptr)
				q.push(current->st);
			if (current->dr != nullptr)
				q.push(current->dr);

		}
		std::cout << std::endl;
	}
}

int main()
{
	std::string expresie = "(1+(2*3))+(4-(5*1))";
	std::vector<char> vector;
	Node* n;
	formaPoloneza(expresie, vector);
	n = buildTree(vector);
	afisare(n);
	return 0;
}
```

Pb2:

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

struct TreeNode {
	int val;
	TreeNode* left;
	TreeNode* right;
};

void inaltimeSubarbore(TreeNode* root, int h, int& h_max) {
	if (root == nullptr)
		return;
	if (h > h_max)
		h_max = h;
	inaltimeSubarbore(root->left, h + 1, h_max);
	inaltimeSubarbore(root->right, h + 1, h_max);
}

void SRD(TreeNode* root) {
	if (root == nullptr)
		return;
```

```
	SRD(root->left);
	std::cout << root->val << " ";
	SRD(root->right);
}

void RSD(TreeNode* root) {
	if (root == nullptr)
		return;
	std::cout << root->val << " ";
	RSD(root->left);
	RSD(root->right);
}

void SDR(TreeNode* root) {
	if (root == nullptr)
		return;
	SDR(root->left);
	SDR(root->right);
	std::cout << root->val << " ";
}

void creareLegaturi(TreeNode* root, std::vector<TreeNode*> nodes) {
	for (int i = 0; i < nodes.size(); i++) {
		if (root->left == nullptr && root->right == nullptr)
			return;
		if (root->left != nullptr && root->left->val == nodes[i]->val) {
			root->left = nodes[i];
			creareLegaturi(root->left, nodes);
		}
		else if (root->right != nullptr && root->right->val == nodes[i]->val) {
			root->right = nodes[i];
			creareLegaturi(root->right, nodes);
			return;
		}
	}

}

void creareArbore(TreeNode*& root) {
	std::vector<int> keys, leftChildren, rightChildren;
	std::ifstream fin("arbore1.txt");
	int key, leftChild, rightChild;
	while (fin >> key >> leftChild >> rightChild) {
		keys.push_back(key);
		leftChildren.push_back(leftChild);
		rightChildren.push_back(rightChild);
	}
	fin.close();
	std::vector<TreeNode*> nodes(keys.size());
	for (int i = 0; i < keys.size(); i++) {
		nodes[i] = new TreeNode;
		nodes[i]->val = keys[i];
		nodes[i]->left = nullptr;
		nodes[i]->right = nullptr;
	}

	for (int i = 0; i < leftChildren.size(); i++) {
		if (leftChildren[i] != -1) {
			TreeNode* newNode = new TreeNode;
```

```
                        newNode->val = leftChildren[i];
                        newNode->left = nullptr;
                        newNode->right = nullptr;
                        nodes[i]->left = newNode;
                }
                else {
                        nodes[i]->left = nullptr;
                }
                if (rightChildren[i] != -1) {
                        TreeNode* newNode = new TreeNode;
                        newNode->val = rightChildren[i];
                        newNode->left = nullptr;
                        newNode->right = nullptr;
                        nodes[i]->right = newNode;
                }
                else {
                        nodes[i]->right = nullptr;
                }
        }

        root = nodes[0];
        creareLegaturi(root, nodes);
}

void afisarePeNiveluri(TreeNode* root) {
        if (root == nullptr)
                return;
        std::queue<TreeNode*> queue;
        queue.push(root);
        while (!queue.empty()) {
                int size = queue.size();
                for (int i = 0; i < size; i++) {
                        TreeNode* current = queue.front();
                        std::cout << current->val << " ";
                        queue.pop();
                        if (current->left)
                                queue.push(current->left);
                        if (current->right)
                                queue.push(current->right);
                }
                std::cout << std::endl;
        }
}

void afisareFrunze() {
        std::vector<int> keys, leftChildren, rightChildren;
        std::ifstream fin("arbore1.txt");
        int key, leftChild, rightChild;
        while (fin >> key >> leftChild >> rightChild) {
                keys.push_back(key);
                leftChildren.push_back(leftChild);
                rightChildren.push_back(rightChild);
        }
        fin.close();
        for (int i = 0; i < keys.size(); i++) {
                if (leftChildren[i] == -1 && rightChildren[i] == -1)
                        std::cout << keys[i] << " ";
        }
        std::cout << std::endl;
```

```
}

bool arboriIdentici() {
	std::vector<int> keys1, leftChildren1, rightChildren1;
	std::ifstream fin("arbore1.txt");
	int key1, leftChild1, rightChild1;
	while (fin >> key1 >> leftChild1 >> rightChild1) {
		keys1.push_back(key1);
		leftChildren1.push_back(leftChild1);
		rightChildren1.push_back(rightChild1);
	}
	fin.close();
	std::vector<int> keys2, leftChildren2, rightChildren2;
	std::ifstream fiin("arbore2.txt");
	int key2, leftChild2, rightChild2;
	while (fiin >> key2 >> leftChild2 >> rightChild2) {
		keys2.push_back(key2);
		leftChildren2.push_back(leftChild2);
		rightChildren2.push_back(rightChild2);
	}
	fiin.close();

	if (keys1.size() != keys2.size())
		return false;

	for (int i = 0; i < keys1.size(); i++) {
		if (keys1[i] != keys2[i] || leftChildren1[i] != leftChildren2[i] || rightChildren1[i] != rightChildren2[i])
			return false;
	}
	return true;
}

bool arboreComplet(TreeNode* root, int k, int inaltime) {
	if (root == nullptr)
		return false;
	std::queue<TreeNode*> queue;
	queue.push(root);
	while (!queue.empty() && k < inaltime) {
		int size = queue.size();
		if (size != pow(2, k))
			return false;
		for (int i = 0; i < size; i++) {
			TreeNode* current = queue.front();
			queue.pop();
			if (current->left)
				queue.push(current->left);
			if (current->right)
				queue.push(current->right);
		}
		k++;
	}
	return true;
}

void adancimeNod(TreeNode* root, int k, int valoare) {
	if (root == nullptr)
		return;
	std::queue<TreeNode*> queue;
	bool gasit = false;
```

```
	queue.push(root);
	while (!queue.empty()) {
		int size = queue.size();
		for (int i = 0; i < size; i++) {
			TreeNode* current = queue.front();
			if (current->val == valoare) {
				std::cout << "Adancimea nodului " << valoare << " este: " << k;
				gasit = true;
				return;
			}
			queue.pop();
			if (current->left)
				queue.push(current->left);
			if (current->right)
				queue.push(current->right);
		}
		k++;
	}
	if (!gasit)
		std::cout << "Nodul " << valoare << " nu exista in arbore.";
}


int main() {

	int h_max = -1;
	TreeNode* root;
	creareArbore(root);
	inaltimeSubarbore(root, 0, h_max);
	std::cout << "Inaltime: " << h_max << std::endl;
	std::cout << "SRD: ";
	SRD(root);
	std::cout << std::endl;
	std::cout << "RSD: ";
	RSD(root);
	std::cout << std::endl;
	std::cout << "SDR: ";
	SDR(root);
	std::cout << std::endl;
	std::cout << "Afisare pe niveluri: " << std::endl;
	afisarePeNiveluri(root);
	std::cout << "Frunzele arborelui sunt: ";
	afisareFrunze();
	std::cout << "Arbori identici: " << arboriIdentici() << std::endl;
	std::cout << "Arbore complet: " << arboreComplet(root, 0, h_max) << std::endl;
	adancimeNod(root, 0, 5);
	return 0;
}
```

Pb4:

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

void maxHeapify(std::vector<int>& arr, int n, int i) {
	int largest = i;
	int left = 2 * i + 1;
```

```
	int right = 2 * i + 2;
	if (left<n && arr[left]>arr[largest])
		largest = left;
	if (right<n && arr[right]>arr[largest])
		largest = right;
	if (largest != i) {
		std::swap(arr[i], arr[largest]);
		maxHeapify(arr, n, largest);
	}

}

void heapSort(std::vector<int>& arr, int n) {
	for (int i = n / 2 - 1; i >= 0; i--) {
		maxHeapify(arr, n, i);
	}
	for (int i = n - 1; i >= 0; i--) {
		std::swap(arr[0], arr[i]);
		maxHeapify(arr, i, 0);
	}
}

void print(std::vector<int>arr, int n) {
	std::cout << "Elementele vectorului dupa sortare: ";
	for (int i = 0; i < arr.size(); i++) {
		std::cout << arr[i] << " ";
	}
}

int main()
{
	std::vector<int> arr = { 64, 25, 12, 22, 11 };
	int n = arr.size();
	heapSort(arr, n);
	print(arr, n);
	return 0;
}
```

Pb5:

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

class priorityQueue {
public:
	std::vector<int> data;

	void maxHeapify(int i) {
		int largest = i;
		int left = 2 * i + 1;
		int right = 2 * i + 2;
		if (left<data.size() && data[left]>data[largest])
			largest = left;
		if (right<data.size() && data[right]>data[largest])
			largest = right;
		if (largest != i) {
			std::swap(data[largest], data[i]);
			maxHeapify(largest);
		}
	}

	void extractMax() {
		data[0] = data[data.size() - 1];
		data.pop_back();
		maxHeapify(0);
	}

	int maxElement() {
		return data[0];
	}

	void increasePriority(int i, int val) {
		if (val > data[i]) {
			data[i] = val;
			int p = (i - 1) / 2;
			while (i > 0 && val > data[p]) {
				data[i] = data[p];
				i = p;
				p = (i - 1) / 2;
			}
			data[i] = val;
		}
	}

	void insert(int val) {
		data.push_back(0);
		increasePriority(data.size() - 1, val);
	}

	void print() {
		for (int i = 0; i < data.size(); i++) {
			std::cout << data[i] << " ";
		}
		std::cout << std::endl;
	}

};
```

```cpp
int main()
{
	priorityQueue p;
	int optiune;
	std::cout << "Comanda 1 pentru inserarea unui nou element;" << std::endl;
	std::cout << "Comanda 2 pentru extragerea elementului maxim;" << std::endl;
	std::cout << "Comanda 3 pentru afisarea elementului prioritate maxima;" << std::endl;
	std::cout << "Comanda 4 pentru afisarea;" << std::endl;
	std::cout << "Comanda 0 pentru iesire." << std::endl;

	do {
		std::cout << "Alege o optiune: ";
		std::cin >> optiune;
		switch (optiune) {
		case 0:
			break;
		case 1:
			int valoare;
			std::cout << "Inserati: ";
			std::cin >> valoare;
			p.insert(valoare);
			break;
		case 2:
			p.extractMax();
			break;
		case 3:
			std::cout << p.maxElement();
			std::cout << std::endl;
			break;
		case 4:
			p.print();
			break;
		default:
			std::cout << "Optiune invalida." << std::endl;
			break;
		}
	} while (optiune != 0);

	return 0;
}
```