МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

Тема: Программирование алгоритмов с бинарными деревьями

Студент гр. 9381 _____ Камакин Д.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы

Познакомиться с представлением и реализацией такой нелинейной структуры данных как бинарное дерево, изучить способы её реализации, получить навыки решения задач обработки бинарных деревьев.

Постановка задачи

Вариант 6-д.

Задано бинарное дерево b типа BT с произвольным типом элементов.

Используя очередь и операции над ней (см.п.2), напечатать все элементы дерева b по уровням: сначала - из корня дерева, затем (слева направо) - из узлов, сыновних по отношению к корню, затем (также слева направо) - из узлов, сыновних по отношению к этим узлам, и т.д.

Описание алгоритма

За основу главного алгоритма вывода дерева по уровням был взят алгоритм поиска в ширину. Описание алгоритма:

1. Поместить узел, с которого начинается поиск, в изначально пустую очередь.

2. Извлечь из начала очереди узел и пометить его как развёрнутый. Еесли узел является целевым узлом и очередь пуста, то завершить поиск с результатом «успех». В противном случае, в конец очереди добавляются все преемники узла , которые ещё не развёрнуты и не находятся в очереди.

3. В случае, если уровень текущего элемента увеличился, тогда необходимо добавить в вывод перенос строки

4. Вернуться к п. 2.

Описание структур:

BinaryTree — класс для бинарного дерева. В приватном поле int level хранится уровень глубины текущего звена дерева при считывании, int readIndex хранит номер текущего символа в строке, TreeNode *root — указатель на корень дерева.

TreeNode — класс для звена бинарного дерева. Приватные поля TreeNode *left и TreeNode *right указывают на левое и правое звено по отношению к текущему, char data хранит информацию звена, int level уровень глубины.

TreeQueue — класс для структуры очереди, приватное поле TreeNode **data — массив для очереди, int currElem — индекс вершины в массиве.

Описание методов класса BinaryTree:

- *void stringToTree(std::string input)* — принимает строку с записью исходного дерева. Запускает функцию для рекурсивного обхода input recTreeInit.
- *void print()* - запускает функцию для рекурсивного обхода дерева и вывода его данных recTreePrint.
- *TreeNode* getRoot()* - возвращает TreeNode* - указатель на корень дерева.
- *void deleteTree(TreeNode *node)* — принимает TreeNode *node — указатель на корень дерева, которое требуется удалить. Производит его рекурсивный обход и очищает память, выделенную для каждого звена.
- *void recTreePrint(TreeNode *node)* — принимает TreeNode *node — указатель на текущий элемент. Производит рекурсивный обход дерева и вывод данных.
- *TreeNode* recTreeInit(std::string input)* — принимает string input — исходную строку с записью дерева, возвращает TreeNode* - звено

дерева. Производит рекурсивный обход строки и формирует дерево.

Описание методов класса TreeNode:

- *TreeNode* getLeft()* - возвращает TreeNode* - указатель на левое плечо текущего звена.
- *void setLeft(TreeNode *l)* — принимает TreeNode *l - указатель на звено, которое требуется установить в качестве леовго плеча.
- *void setRight(TreeNode *r)* — принимает TreeNode *r — указатель на звено, которое требуется установить в качестве правого плеча.
- *TreeNode* getRight()* - возвращает TreeNode* - указатель на правое плечо текущего звена.
- *char getData()* - возвращает char — символ текущего звена.
- *int getLevel()* - возвращает int — уровень глубины текущего звена.
- *void setLevel(int l)* — принимает int l — уровень глубины текущего звена.

Описание методов класса TreeQueue:

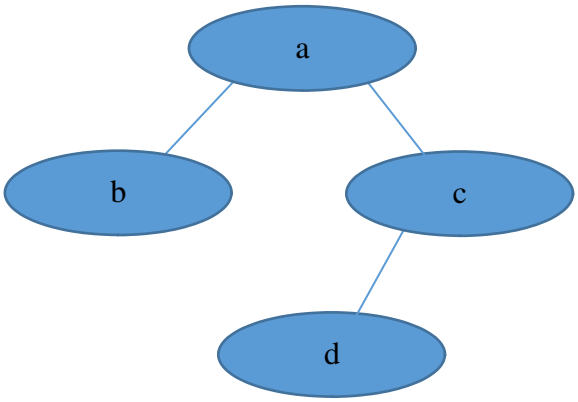
- *void extendMemory()* - увеличивает вместимость очереди на 1 элемент.
- *void add(TreeNode* value)* — принимает TreeNode *value — указатель на звено дерева, которое требуется добавить в очередь.
- *TreeNode* pop()* - возвращает TreeNode* - указатель на текущий элемент и удаляет его из очереди.
- *TreeNode* top()* - возвращает TreeNode* - указатель на текущий элемент в очрееди.
- *int size()* - возвращает int — текущий размер очереди.

Описание функций:

- *int getAction()* - возвращает int — выбор пользователя, считанный с клавиатуры.
- *bool stepByStepAlg(int& lev, string &out, TreeQueue* queue, bool flag)* — возврщает bool — текущее состояние в рекурсии. Принимает int &lev — глубина звена, string &out — результирующая строка, TreeQueue *queue — указатель на очередь, bool flag — текущее состояние.

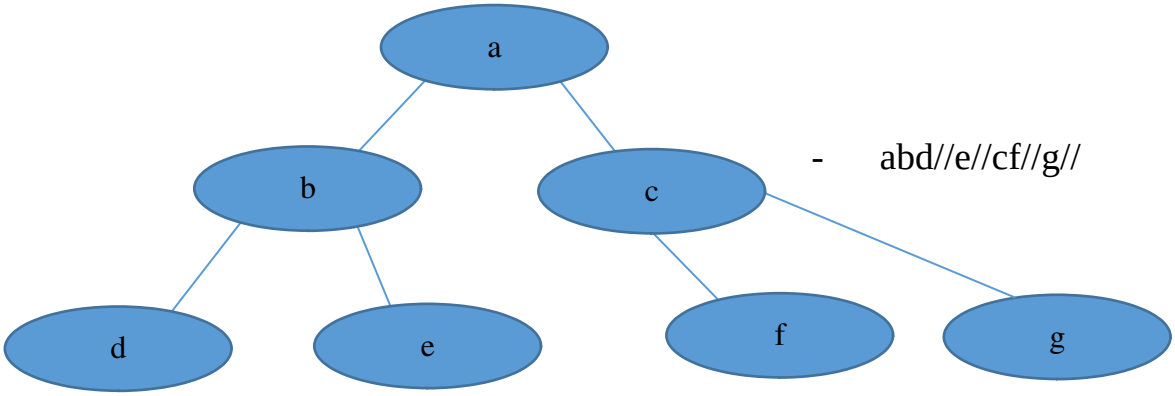
Примеры деревьев и их представление:

Пример 1:



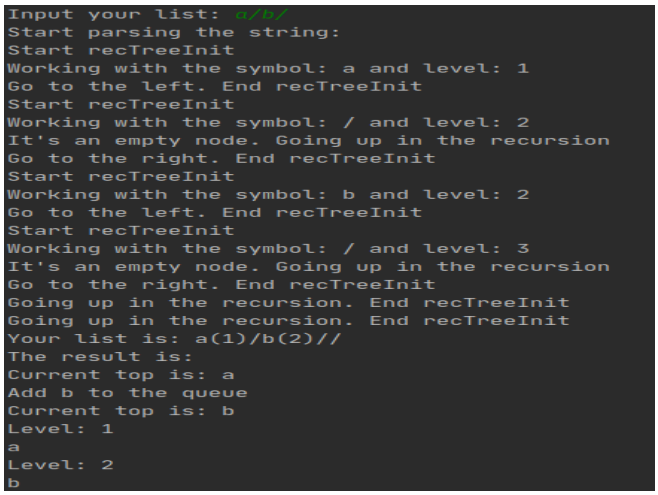
- ab//cd///

Пример 2:



- abd//e//cf//g//

Тестирование

Входные данные	Исходные данные
1. abd//e//cf//g//	Level: 1 a Level: 2 b c Level: 3 d e f g
2. a/b//	
3. a//	Level: 1 a
4. a/b//	Level: 1 a Level: 2 b

5. cd/b//	Level: 1 c Level: 2 d Level: 3 b

Выводы

В ходе работы были приобретены навыки работы с бинарными деревьями, изучены методы обхода бинарных деревьев и реализация деревьев на базе указателей на структуру

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

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

using namespace std;

// Node of the binary tree.
class TreeNode {
    TreeNode *left = nullptr;
    TreeNode *right = nullptr;
    char data = '\0';
    int level = 1;

public:
    TreeNode() = default;

    // Constructor with data
    explicit TreeNode(char symbol) : data(symbol) {};

    // Return the left node
    TreeNode* getLeft() {
        return left;
    }

    // Set the left node
    void setLeft(TreeNode *l) {
        left = l;
    }

    // Set the right node
    void setRight(TreeNode *r) {
        right = r;
    }

    // Return the right node
    TreeNode* getRight() {
```

```
            return right;
        }

        // Return the data
        char getData() {
            return data;
        }

        // Return the level
        int getLevel() {
            return level;
        }

        // Set the level
        void setLevel(int l) {
            level = l;
        }
    };


    // Class of the binary tree. Based on TreeNode
    class BinaryTree {
        TreeNode *root = nullptr;
        int readIndex = 0;
        int level = 1;

    public:
        BinaryTree() : root(nullptr) {};

        ~BinaryTree() {
            deleteTree(root);
        }

        // Start recursion init of the tree
        void stringToTree(std::string input) {
            if (input.empty() || input == "\n") // Check for incorrect
string
                return;
```

```
            root = new TreeNode();
            std::cout << "Start parsing the string: " << '\n';
            root = recTreeInit(std::move(input));
        }

        // Start recursion print the tree
        void print() {
            recTreePrint(root);
        }

        // Return the root
        TreeNode* getRoot() {
            return root;
        }

    private:
        // Recursion tree memory free
        void deleteTree(TreeNode *node) {
            if (!node)
                return;

            deleteTree(node->getLeft()); // Go to left
            deleteTree(node->getRight()); // Go to right
            delete node;
        }

        // Recursion output the tree
        void recTreePrint(TreeNode *node) {
            if (!node) {
                std::cout << '/';
                return;
            }

            std::cout << node->getData() << '(' << node->getLevel() <<
')';
            recTreePrint(node->getLeft()); // Go to left
            recTreePrint(node->getRight()); // Go to right
        }
```

```
        // Recursion read the tree
        TreeNode* recTreeInit(std::string input) {
            if (input.length() <= readIndex)
                return nullptr;
            std::cout << "Start recTreeInit" << '\n';
            char symbol = input[readIndex];
            readIndex++;
            std::cout << "Working with the symbol: " << symbol << " and
level: " << level << '\n';

            if (symbol == '/') {
                std::cout << "It's an empty node. Going up in the
recursion" << '\n';
                level--;
                return nullptr;
            } else {
                auto buf = new TreeNode(symbol);
                buf->setLevel(level);
                level++;
                std::cout << "Go to the left. End recTreeInit" << '\n';
                buf->setLeft(recTreeInit(input));
                level++;
                std::cout << "Go to the right. End recTreeInit" << '\
n';
                buf->setRight(recTreeInit(input));
                level--;
                std::cout << "Going up in the recursion. End
recTreeInit" << '\n';
                return buf;
            }
        }
    };

    // Queue for the tree
    class TreeQueue {
        TreeNode** data = nullptr;
        int currElem = 0;

    public:
```

```
        TreeQueue() = default;;

        // Reallocate the array and adds one element
        void extendMemory() {
            auto **buffer = new TreeNode*[currElem + 1];

            for (int i = 0; i < currElem; i++)
                buffer[i] = data[i];

            delete [] data;
            data = buffer;
        }

        // Start reallocate
        void add(TreeNode* value) {
            extendMemory();
            data[currElem++] = value;
        }

        // Delete the top
        TreeNode* pop() {
            TreeNode *elem = data[0];

            for (int i = 1; i < currElem + 1; i++) // Move elements in
the array
                data[i - 1] = data[i];

            currElem--;
            return elem;
        }

        // Return top
        TreeNode* top() {
            return data[0];
        }

        // Return size
        int getSize() {
            return currElem;
```

```
        }
    };

    // The main algorithm based on queue.
    bool stepByStepAlg(int& lev, string &out, TreeQueue *queue, bool
flag) {
        for (;;) {
            TreeNode *current = queue->top();
            cout <<  "Current top is: " << current->getData() << '\n';
            queue->pop();

            if (lev != current->getLevel()) {
                lev++;
                if (lev == 1) {
                    out += "Level: ";
                    out += std::to_string(lev);
                    out += '\n';
                } else {
                    out += "\nLevel: ";
                    out += std::to_string(lev);
                    out += '\n';
                }
            }

            out += current->getData();
            out += '\t';

            if (!current->getLeft() && !current->getRight() && !queue-
>getSize())
                return true;

            if (current->getLeft()) {
                cout << "Add " << current->getLeft()->getData() << " to
the queue" << '\n';
                queue->add(current->getLeft());
            }

            if (current->getRight()) {
```

```
                cout << "Add " << current->getRight()->getData() << "
to the queue" << '\n';
                queue->add(current->getRight());
            }

            if (flag)
                return false;
        }
    }

    int getAction() {
        int action = 0;

        cout << "Choose one of the following options: " << '\n' <<
             "1. Read from the keyboard" << '\n' <<
             "2. Read from the file" << '\n' <<
             "3. Exit" << '\n' <<
             "Your choice: ";
        cin >> action;

        return action;
    }

    int main() {
        int action;
        string input;
        ifstream file;
        string fileName;

        while((action = getAction()) != 3) {
            string out;
            int level = 0;
            auto *tree = new BinaryTree();
            auto *treeQueue = new TreeQueue();

            switch(action) {
                case 1:
                    cout << "Input your list: ";
                    cin >> input;
```

```
            break;
        case 2:
            cout << "Input the path to your file: ";
            cin >> fileName;
            file.open(fileName);

            if (!file.is_open()) {
                cout << "Wrong file" << '\n';
                continue;
            }

            file >> input;
            file.close();
            break;
        default:
            cout << "Wrong input. Try again" << '\n';
            return 0;
    }

    tree->stringToTree(input);

    if (!tree->getRoot()) {
        cout << "That's an empty tree" << "\n\n";
        continue;
    }

    treeQueue->add(tree->getRoot());
    cout << "Your list is: ";
    tree->print();
    cout << '\n';
    cout << "The result is: " << '\n';
    stepByStepAlg(level, out, treeQueue, false);
    cout << out;
    cout << "\n\n";

    delete tree;
    delete treeQueue;
}
```

```
    cout << "Exiting the program" << '\n';
    return 0;
}
```