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: Бинарные деревья

Студент гр. 9304 _____ Цаплин И.В.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

Цель работы.

Ознакомиться с понятием бинарного дерева. Реализовать бинарное дерево на языке программирования C++.

Задание.

Вариант 3.

Для заданного бинарного дерева b типа BT с произвольным типом элементов:

- напечатать элементы из всех листьев дерева b;

- подсчитать число узлов на заданном уровне n дерева b (корень считать узлом 1-го уровня).

Описание алгоритма работы.

Программа считывает строку treeString и число levelToCount из стандартного потока ввода. Затем с помощью функции isCorrect устанавливается, является ли строка корректным представлением бинарного дерева. Если строка не является корректным представлением бинарного дерева, программа выводит сообщение об этом и завершает работу. Если строка корректна, создаётся дерево binTree. Для дерева binTree вызывается метод printLeaves, который печатает все элементы из листьев дерева. Затем вызывается метод countLevel, который подсчитывает количество элементов на уровне levelToCount дерева. Результат работы метода levelToCount выводится в стандартный поток вывода.

Формат входных и выходных данных.

На вход программе подаётся скобочкая запись бинарного дерева и число. Скобочкая запись дерева имеет вид (<Root>(<Left>)(<Right>)), где <Root> корневой элемент, <Left> - левый элемент или левое поддерево, <Right> - правый элемент или правое поддерево. () - обозначение пустого поддерева.

Описание основных структур данных и функций.

1. Class BinTreeNode – узел бинарного дерева:
   - std::shared_ptr<BinTreeNode> left — указатель на левое поддерево
   - std::shared_ptr<BinTreeNode> right — указатель на правое поддерево
   - std::weak_ptr<BinTreeNode> parent — указатель на родителя

- T data — данные типа T

2. Class BinTree – бинарное дерево:
   - std::shared_ptr<BinTreeNode<T>> head – указатель на корень дерева
   - copyBinTree() – метод копирования дерева.

Принимает указатель на узел другого дерева и его родителя, возвращает указатель на копию узла. Метод работает рекурсивно. Сначала создаётся копия текущего узла, затем метод вызывается для правого и левого поддеревьев. Данный метод используется конструктором копирования и оператором копирования.

- insert() – метод вставки элемента в дерево.

Осуществляет вставку в дерево на наименьшую глубину, начиная с левого поддерева. Метод записывает вершины каждого уровня в очередь, начиная с левой. В тоже время осуществляется поиск пустого поддерева. При нахождении пустого поддерева происходит вставка элемента, метод завершает работу. Таким образом, метод просматривает уровень за уровнем в поиске пустого места, при этом первым рассматривается левое поддерево.

- find() – метод поиска элемента по значению.

Метод работает рекурсивно. Сначала метод вызывается для левого поддерева, затем для правого, если элемент не был найден. В последнюю очередь проверяется корневой элемент. Если элемент найден, метод возвращает указатель на него, в противном случае возвращается nullptr.

- deleteNode() – метод удаления элемента по значению.

Использует метод find() для нахождения элемента. Если элемент найден, удаляет сам элемент и всё его поддерево.

- printTree() – вывод дерева.

Метод работает рекурсивно. Сначала метод вызывается для левого поддерева. Затем печатается корневой элемент. Затем метод вызывается для правого поддерева.

- printLeaves() – вывод листьев.

Метод работает рекурсивно. Если элемент – лист, данные элемента выводятся в стандартный поток вывода. Если элемент – не лист, метод вызывается для левого и правого поддеревьев.

- countLevel() - подсчет элементов на заданном уровне.

Если текущий узел находится на заданном уровне, возвращает единицу. Если текущий уровень меньше заданного, вызывает метод для для левого и правого поддеревьев.

- StrToBin() - создание бинарного дерево по заданной строке.

Метод создаёт корневой элемент и записывает в него данные. Затем, если встречен символ открывающейся скобки, метод вызывается для левого поддерева, затем для правого. Метод возвращает указатель на корневой элемент дерева и используется в конструкторе класса.

- getHead() - возвращает указатель на голову списка.

3. Функция isCorrect() - проверяет, что строка является корректным представлением бинарного дерева. С помощью стека проверяет корректность расставленных скобок. Также проверяет, что после каждой открывающейся скобки стоят данные, а после данных стоит открывающаяся или закрывающаяся скобка. Возвращает true, если строка корректна, и false, если строка некорректна.

Разработанный код см. в приложении А.

Тестирование.

Для проведения тестирования был написан bash-скрипт tests_script. Скрипт запускает программу с определёнными входными данными и сравнивает полученные результаты с готовыми ответами. Для каждого теста выводится сообщение TestX <входные данные> passed или TestX <входные данные> failed. Для каждого теста выводится ожидаемый результат и полученный. Полученные в ходе работы файлы с выходными данными удаляются.

Результаты тестировния см. в приложении Б.

Выводы.

Было изучено понятие бинарного дерева. Реализовано бинарное дерево на языке программирования C++.

Реализована программа, котороя создаёт бинарное дерево по его скобочному представлению, выводит данные в его листах и количество элементов на заданном уровне. Проведено тестирование работы программы.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab3.cpp

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

template<typename T>
class BinTree;

template<typename T>
class BinTreeNode {
    std::shared_ptr<BinTreeNode> left;
    std::shared_ptr<BinTreeNode> right;
    std::weak_ptr<BinTreeNode> parent;
    T data;
    friend class BinTree<T>;
public:
    BinTreeNode(std::shared_ptr<BinTreeNode> left,
std::shared_ptr<BinTreeNode> right,
std::shared_ptr<BinTreeNode<T>> parent , T data):
left(std::move(left)), right(std::move(right)),
parent(std::move(parent)), data(data){
    }
};

template<typename T>
class BinTree {
public:
    BinTree(std::string& str){
        auto iterator = str.cbegin();
        head = strToBinTree(iterator, nullptr);
    }

    ~BinTree() = default;

    BinTree(const BinTree<T> &other) {
        head = copyBinTree(other.head, nullptr);
    }

    BinTree<T>& operator=(const BinTree<T> &other) {
        head = copyBinTree(other.head);
        return *this;
    }

    BinTree(BinTree<T> &&other) {
        head = std::move(other.head);
    }
```

```
        BinTree<T>& operator= (BinTree<T> &&other) {
            head = std::move(other.head);
            return *this;
        }

        std::shared_ptr<BinTreeNode<T>>
copyBinTree(std::shared_ptr<BinTreeNode<T>> otherHead,
std::shared_ptr<BinTreeNode<T>> headParent) {
            if (otherHead == nullptr) {
                return nullptr;
            }

            if (otherHead == head) {
                return head;
            }

            std::shared_ptr<BinTreeNode<T>> curHead =
std::make_shared<BinTreeNode<T>>(nullptr, nullptr, headParent,
otherHead->data);
            if (otherHead->left != nullptr) {
                curHead->left = copyBinTree(otherHead->left,
curHead);
            }

            if (otherHead->right != nullptr) {
                curHead->right = copyBinTree(otherHead->right,
curHead);
            }

            return curHead;
        }

        void insert(T dataToInsert){
            if (head == nullptr) {
                head = std::make_shared<BinTreeNode<T>>(nullptr,
nullptr, nullptr, dataToInsert);
                return;
            }
            std::queue<std::shared_ptr<BinTreeNode<T>>> queue;
            queue.push(head);
            while (!queue.empty()) {
                std::shared_ptr<BinTreeNode<T>> temp =
queue.front();
                queue.pop();
                if (temp->left != nullptr)
                    queue.push(temp->left);
                else {
                    temp->left =
std::make_shared<BinTreeNode<T>>(nullptr, nullptr, temp,
dataToInsert);
                    return;
                }
                if (temp->right != nullptr)
```

```
                    queue.push(temp->right);
                else {
                    temp->right =
std::make_shared<BinTreeNode<T>>(nullptr, nullptr, temp,
dataToInsert);
                    return;
                }
            }
        }

        std::shared_ptr<BinTreeNode<T>>
find(std::shared_ptr<BinTreeNode<T>> curNode, T dataToFind){
            if(!curNode){
                return nullptr;
            }
            std::shared_ptr<BinTreeNode<T>> foundNode = nullptr;
            if(curNode->left != nullptr){
                foundNode = find(curNode->left, dataToFind);
            }
            if(foundNode != nullptr){
                return foundNode;
            }
            if(curNode->right != nullptr){
                foundNode = find(curNode->right, dataToFind);
            }
            if(foundNode != nullptr){
                return foundNode;
            }
            if(curNode->data == dataToFind){
                return curNode;
            }
            return nullptr;
        }

        void deleteNode(T dataToDelete){
            std::shared_ptr<BinTreeNode<T>> NodeToDelete =
find(head, dataToDelete);
            if(!NodeToDelete){
                return;
            }
            std::shared_ptr<BinTreeNode<T>> NodeParent =
NodeToDelete->parent.lock();
            if(NodeParent == nullptr){
                head = nullptr;
                return;
            }
            if(NodeParent->left != nullptr){
                if(NodeParent->left->data == dataToDelete){
                    NodeParent->left = nullptr;
                    return;
                }
            }
            if(NodeParent->right != nullptr){
```

```
                if(NodeParent->right->data == dataToDelete){
                    NodeParent->right = nullptr;
                }
            }
        }

        void printTree(std::shared_ptr<BinTreeNode<T>> curNode) {
            if (!curNode) {
                return;
            }
            printTree(curNode->left);
            std::cout << curNode->data << ' ';
            printTree(curNode->right);
        }

        void printLeaves(std::shared_ptr<BinTreeNode<T>> curNode)
{
            if (!curNode) {
                return;
            }
            if ((curNode->left == nullptr) && (curNode->right ==
nullptr)){
                std::cout << curNode->data << ' ';
            }else{
                printLeaves(curNode->left);
                printLeaves(curNode->right);
            }
        }

        int countLevel (std::shared_ptr<BinTreeNode<T>> curNode,
int curLevel, int levelToCount) {
            if (curNode == nullptr){
                return 0;
            }
            if (curLevel == levelToCount){
                return 1;
            }
            if (curLevel < levelToCount) {
                return countLevel(curNode->left, curLevel + 1,
levelToCount) +
                       countLevel(curNode->right, curLevel + 1,
levelToCount);
            }
         return 0;
        }

        std::shared_ptr<BinTreeNode<T>>
strToBinTree(std::string::const_iterator& iterator,
std::shared_ptr<BinTreeNode<T>> parent){
            std::shared_ptr<BinTreeNode<T>> root = nullptr;
            iterator++;
            if (*iterator == ')'){
                return  root;
```

```
            }
            root = std::make_shared<BinTreeNode<T>>(nullptr,
nullptr, parent, *iterator);
            iterator++;
            if (*iterator == '('){
                root->left = strToBinTree(iterator, root);
                iterator++;
            }
            if (*iterator == '('){
                root->right = strToBinTree(iterator, root);
                iterator++;
            }
            return  root;
        }

        std::shared_ptr<BinTreeNode<T>> getHead(){
            return head;
        }

    private:
        std::shared_ptr<BinTreeNode<T>> head;
    };


    bool isCorrect(const std::string& str){
        std::stack<char> Stack;
        if (str[0] != '('){
            return false;
        }
        for (auto iter = str.cbegin();iter != str.cend()-1; iter+
+){
            if((*iter == '(') && (*(iter+1) == ')') {
                return false;
            }
            if(*iter != '(' && *iter != ')' && *(iter+1)  != '('
&& *(iter+1)  != ')')
                return false;
        }
        for (char i : str){
            if (i == '('){
                Stack.push(i);
            }
            if (i == ')'){
                if (Stack.empty()){
                    return false;
                }
                Stack.pop();
            }
        }
        return Stack.empty();
    }

    int main() {
```

```
        std::string treeString;
        std::getline (std::cin, treeString);
        int levelToCount;
        std::cin >> levelToCount;
        if (!isCorrect(treeString)) {
            std::cout << "Tree is not correct\n";
            return 0;
        }
        BinTree<char> binTree(treeString);
        binTree.printLeaves(binTree.getHead());
        std::cout << "\n" << binTree.countLevel(binTree.getHead(),
1, levelToCount) << std::endl;

    }
```

Название файла: tests_script

```
#! /bin/bash

printf "\nRunning tests...\n\n"
for n in {1..8}
do
    ./lab3 < "./Tests/tests/test$n.txt" > "./Tests/out/out$n.txt"
    printf "Test$n: "
    cat "./Tests/tests/test$n.txt" | tr -d '\n'
    if cmp "./Tests/out/out$n.txt"
"./Tests/true_results/true_out$n.txt" > /dev/null; then
            printf " - Passed\n"
    else
            printf " - Failed\n"
    fi
    printf "Desired result:\n"
    cat "./Tests/true_results/true_out$n.txt"
    printf "Actual result:\n"
    cat "./Tests/out/out$n.txt"
    printf "\n"
done
rm ./Tests/out/out*
```

Название файла: Makefile

```
lab3: Source/lab3.cpp
```

```
        g++ -Wall -std=c++17 Source/lab3.cpp -o lab3

run_tests: lab3
        ./tests_script
```

ПРИЛОЖЕНИЕ Б
РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ

№	Входные данные	Выходные данные	Комментарии
1	(A(B(C)(D))E) 3	C D 2	Дерево с двумя элементами на третьем уровне
2	() 1	0	Пустое дерево. Нет листьев. Ноль элементов на первом уровне
3	(A(B(C(D(E))))) 5	E 1	Дерево только с левыми потомками
4	(A(B(D)(E))(C(F)(G))) 3	D E F G 4	Максимально разветвлённое дерево
5	(A()(C()(D()(F)))) 5	F 0	Дерево только с правми потомками. Заданный для подсчёта уровень лежит вне дерева
6	(A(B()(D))(C(E))) 3	D E 2	Дерево с двумя элементами на третьем уровне
7	(A(B)(CE)) 2	Tree is not correct	Некорректная запись дерева. Два элемента в

			одном узле
8	(A(B)(C) 2	Tree is not correct	Некорректная запись дерева. Пропущена последняя скобка