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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

Тема: Деревья

Студент гр. 9382 _____ Субботин М. О.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2020

Цель работы.

Познакомиться с одной из часто используемых на практике нелинейных конструкций, способами её организации и рекурсивной обработки. Получить навыки решения задач обработки деревьев, как с использованием базовых функций их рекурсивной обработки, так и без использования рекурсии.

Задание

4. Для заданного бинарного дерева b типа BT с произвольным типом элементов определить, есть ли в дереве b хотя бы два одинаковых элемента.

При выполнении упражнений следует использовать вариант реализации бинарного дерева (см. 3.5), соответствующий варианту.

Для представления деревьев во входных данных рекомендуется использовать скобочную запись, кроме случаев, специально оговоренных в условии задачи. В выходных данных рекомендуется представлять дерево (лес) в горизонтальном виде (т. е. с поворотом на 90°), а бинарные деревья - в виде уступчатого списка.

В заданиях 1 - 5, в зависимости от варианта, предлагается реализовать рекурсивные или не рекурсивные процедуры (функции); в последнем случае следует использовать стек и операции над ним.

Описание структуры дерева, которая используется в программе.

Структура дерева реализовывалась через динамическую память.

```
 template<typename Elem>
class BT{
public:
   Elem value;
   BT *left;
   BT *right;
   BT(Elem val, BT * l, BT * r);
   BT * enter();
   void checkForEqualElements(BT * root);
   void outBT(BT * root);
};
```

Elem value – элемент, который находится в вершине

BT* left – указатель на левого ребенка вершины

BT* right – указатель на правого ребенка вершины

Описание методов:

BT* enter() – метод, который вводит дерево из файла и возвращает указатель на корень этого дерева.

void checkForEqualElements(BT* root) – метод, который реализует алгоритм поиска дублирующих вершин.

void outBT(*BT root) – выводит дерево на консоль в таком же виде в каком и вводил с файла.

Описание алгоритма:

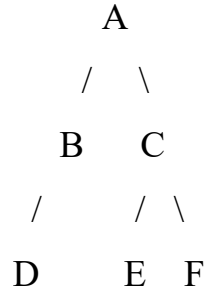
От нас требуется определить какие элементы вершин дублируются.

Мы просто проходимся по всем вершинам дерева, не важно каким способом (в нашей программе обходим мы в ширину), смотрим, если этот элемент мы раньше не встречали, запоминаем его, если мы встречали раньше этот элемент – выводим информацию об этом.

Обход в ширину осуществляется так: с начала корень дерева мы записываем в очередь. Затем пока очередь не опустеет мы будем проделывать следующие действия. Достаем из очереди элемент, проделываем нужные нам действия для решения задачи, потом проверяем, существует ли у этого элемента левый ребенок, если да, то записываем левого ребенка в очередь, затем проверяем, существует ли правый ребенок, если да, то тоже записываем его в очередь.

Наш обход по дереву можно назвать обходом по его уровням. То есть, каждый раз, записывая потомков очередной вершины в конец очереди мы не продвинемся “вглубь”, пока не пройдем все вершины на текущем уровне.

Давайте разберем наш обход на маленьком примере. Вот у нас есть дерево:



Мы записываем наш корень в очередь: A.

Затем вытаскиваем вершину из очереди и проверяя каждого из двух детей заносим их в очередь: B C. Выводим при этом A.

Затем, вытаскиваем первую занесенную вершину – т.е. B и проделываем ту же операцию для нее. Т.к. у нее только левый ребенок, наша очередь будет выглядеть так: C D. И в выводе будут уже две вершины A B.

Вытаскиваем первую вершину из очереди – C, проделываем те же операции, в очереди теперь находятся D E F, и в выводе A B C.

Снова вытаскиваем первую вершину из очереди – D, у нее нет детей, поэтому в очередь не добавятся элементы, а в выводе будут A B C D.

Вытаскиваем E из очереди, у нее тоже нет детей, поэтому очередь будет состоять только из элемента F, а в выводе будут A B C D E.

И наконец вытаскиваем вершину F, у нее нет детей, значит в очередь мы больше не добавим элементом, очередь оказывается пуста, и в выводе будут A B C D E F. Т.к. очередь пуста, и мы не добавили в нее новых элементов, наш алгоритм обхода завершился. Если посмотреть на наш вывод, то можно заметить, что вершины в нем расположены так, как мы бы смотрели на них слева направо по уровням 0 уровень : A, 1 уровень: B C, 2 уровень: D E F.

Исходный код:

```
#include <iostream>
#include <fstream>
#include <queue>
#include <set>
using namespace std;

ifstream infile ("KLP.txt");

/*
 * class BT - класс, реализующий бинарное дерево
 * Elem value - элемент, содержащийся в вершине
 * BT* left - указатель на левого ребенка вершины
 * BT* right - указатель на правого ребенка вершины
 */
```

```
template<typename Elem>
class BT{
public:
    Elem value;
    BT *left;
    BT *right;
    BT(Elem val, BT * l, BT * r);
    BT * enter();
    void checkForEqualElements(BT * root);
    void outBT(BT * root);
};
/*
 * BT(Elem val) - конструктор бинарного дерева
 * Elem val - элемент в будущей вершине
 * BT * l - левый ребенок в будущей вершине
 * BT * r - правый ребенок в будущей вершине
 */
template <typename Elem>
BT<Elem>::BT(Elem val,BT * l, BT * r) {
    this->value = val;
    this->left = l;
    this->right = r;
}

/*
 * BT<Elem>* enter() - метод для ввода дерева, возвращает указатель на
корень дерева
 */
template<typename Elem>
BT<Elem>* BT<Elem>::enter() {
    Elem val;
    infile >> val;
    if(val == '/') return nullptr;
    else {
        BT<Elem> * l = enter();
        BT<Elem> * r = enter();
        return new BT<Elem>(val,l,r);
    }
}
/*
 * void outBT(BT<Elem> *root) - метод для вывода дерева в консоль
 * BT<Elem> * root - указатель на корень дерева
```

```
 */
template<typename Elem>
void BT<Elem>::outBT(BT<Elem> *root) {
    if(root!=nullptr){
        cout << root->value;
        outBT(root->left);
        outBT(root->right);
    } else {
        cout << "/";
    }

}

/*
 * void checkForEqualElements(BT* root) - метод, определяющий существуют ли одинаковые элементы в дереве
 * BT* root - указатель на корень дерева
 */
template<typename Elem>
void BT<Elem>::checkForEqualElements(BT * root) {
    if(root == nullptr){
        cout << "\nНекорректное дерево!\n";
        return;
    }
    //создаем очередь для итеративного обхода по дереву в ширину
    std::queue<BT<Elem>*> queueBFS;

    //заносим в очередь корень дерева
    queueBFS.push(root);
    cout << "Заносим корень дерева '" << root->value << "' в очередь \n";

    //создаем множество встречающихся во время итерации вершин
    std::set<Elem> previousNodes;

    //создаем множество вершин-дубликатов
    std::set<Elem> duplicateNodes;

    //обходим дерево в ширину
    while(!queueBFS.empty()){
        //достаем из очереди вершину
        BT<Elem> *tempNode = queueBFS.front();
        queueBFS.pop();
```

```
            cout << "Достаем вершину '" << tempNode->value << "' из очереди
\n";

            //проверяем эта вершина дубликат или нет
            if(previousNodes.count(tempNode->value) == 1){
                cout << "Вершина '" << tempNode->value << "' уже встречалась в
дереве!\n";
                //т.к. дубликат, то вставляем ее в множество дубликатов
                duplicateNodes.insert(tempNode->value);
            }
            else {
                //т.к. не дубликат, вставляем ее в множество пройденных вершин
                previousNodes.insert(tempNode->value);
                cout << "Вершины '" << tempNode->value << "' еще не было в
дереве, запоминаем ее\n";
            }
            if(tempNode->left!=nullptr){
                //вставляем, если существует, левого ребенка с очередь
                queueBFS.push(tempNode->left);
                cout << "Левого ребенка '"<< tempNode->left->value <<"'
текущей вершины '" << tempNode->value << "' записываем в очередь\n";
            }
            else {
                cout << "У текущей вершины '" << tempNode->value << "' нет
левого ребенка!\n";
            }
            if(tempNode->right!=nullptr){
                //вставляем, если существует, правого ребенка в очередь
                queueBFS.push(tempNode->right);
                cout << "Правого ребенка '"<< tempNode->right->value<<"'
текущей вершины '" << tempNode->value << "' записываем в очередь\n";
            }
            else {
                cout << "У текущей вершины '" << tempNode->value << "' нет
правого ребенка!\n";
            }

        }
        //если существуют дубликаты, то выведем их
        if(!duplicateNodes.empty()) {
            cout << "\nВершины, которые встречаются в дереве более одного
раза: \n";
```

```
            typename std::set<Elem>::iterator it = duplicateNodes.begin();
            while (it != duplicateNodes.end()) {
                cout << *it << " ";
                it++;
            }
        }
        //если дубликатов не существуют, выведем сообщение об этом
        else {
            cout << "\nВ данном дереве все элементы вершин различны";
        }
    }

    int main() {
        BT<char> * tree;
        tree = tree->enter();
        cout << "Рассматриваемое дерево: \n";
        tree->outBT(tree);
        cout << "\nПроходимся по дереву в ширину и ищем одинаковые элементы:
\n";

        tree->checkForEqualElements(tree);
        cout << "\n";
        return 0;
    }
```

Тестирование.

№	Входные данные	Выходные данные	Результат
1	/	Некорректное дерево!	Правильно
2	abd/g///cq//yi//fk///	В данном дереве все вершины различны	Правильно
3	abd/g///cq//ya//fk///	a	Правильно
4	acd/g///cq//ya//fk///	a c	Правильно
5	aaa/a///aa//aa//aa///	a	Правильно
6	b//	В данном дереве все вершины различны	Правильно
7	a/b/a//	a	Правильно
8	c/b/a//	В данном дереве все вершины различны	Правильно

Обработка результатов тестирования.

Программа выдает корректные результаты на всех тестах. Рассматривается также отдельно случай, когда мы вводим пустое дерево.

Выводы.

Я ознакомился с такой структурой данных, как дерево и научился обрабатывать его.