

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(ка) гр. 9381

Шахин Н.С

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Ознакомится с бинарными деревьями и реализовать функции для работы с ними.

Задание.

Вариант 18д

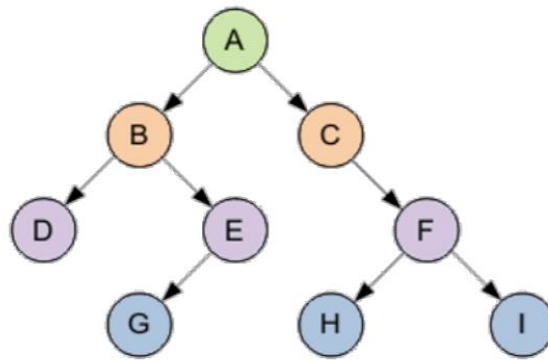
18. Бинарное дерево называется бинарным деревом поиска, если для каждого его узла справедливо: все элементы правого поддеревя больше этого узла, а все элементы левого поддеревя – меньше этого узла. Бинарное дерево называется пирамидой, если для каждого его узла справедливо: значения всех потомков этого узла не больше, чем значение узла.

Для заданного бинарного дерева с числовым типом элементов определить, является ли оно бинарным деревом поиска и является ли оно пирамидой.

Основные теоретические положения.

Дерево – структура данных, представляющая собой древовидную структуру в виде набора связанных узлов. Бинарное дерево — это конечное множество элементов, которое либо пусто, либо содержит элемент (корень), связанный с двумя различными бинарными деревьями, называемыми левым и правым поддеревьями. Каждый элемент бинарного дерева называется узлом. Связи между узлами дерева называются его ветвями.

Способ представления бинарного дерева:



A — корень дерева

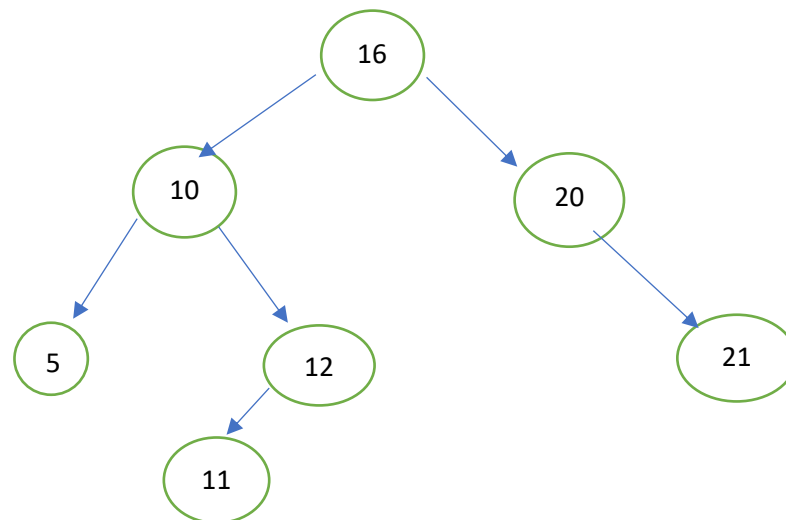
B — корень левого поддерева

C — корень правого поддерева

Обход дерева осуществляется в порядке КЛП. Обход дерева сверху вниз
(в прямом порядке)

Пример дерева, обрабатываемого программой:

(рис.1)



(16(10(5)(12(11)))(20()(21)))

Описание работы алгоритма.

Для начала программа должна считать данные и создать бинарное дерево. Дерево реализуется на базе указателей: в полях каждого узла должен

храниться указатель на левый и правый элемент узла. Если узел в дереве пустой, то хранится указатель `nullptr`.

Нахождение бинарного дерева поиска.

На вход алгоритму даётся дерево. Обход дерева осуществляется в порядке КЛП. Сначала проверяется левое поддерево, если его значение меньше корня, то возвращается `true`, затем проверяется правое поддерево, если его значение больше, чем корень, то возвращается `true`. По этому принципу рекурсивно проходим по дереву.

Для наглядности работы алгоритма опишу его на примере дерева изображённого на рисунке 1 стр 3. Сначала идём в лево от корня 16 к корню 10. $10 < 16$, значит мы проверили корень и можем идти в левое поддерево. Идем от 10 в лево в 5. $5 < 10$, проверили корень. У 5 нет поддеревьев значит возвращаем `true` и идём на уровень выше. От 10 идём в право в 12. $12 > 10$ и меньше 16, проверили корень, теперь проверяем левое поддерево. Из 12 идём в 11, $11 > 10$ и $11 < 12$, у 11 нет поддеревьев, значит идём на уровень выше и возвращаем `true`. Таким образом вы завершили обход КЛП для вершины 10, функции для проверки левого и правого поддерева вернули `true`, значит возвращаемся на уровень выше в корень дерева и возвращаем `true`. Теперь нужно обойти правое поддерево у корня. Идём в право в 20, $20 > 16$. У 20 нет левого поддерева, значит сразу идём в право в 22, $22 > 20$, возвращаем `true`. Заканчиваем обход правого поддерева и возвращаем `true` и поднимаемся на уровень корня. Функции для обхода правого и левого поддерева вернули `true`, таким образом значит функция для проверки дерева тоже возвращает значение `true`.

Нахождение пирамиды.

Алгоритм нахождения пирамиды аналогичен нахождению бинарного дерева поиска, за исключением того, что элементы правого поддерева должны быть меньше или равны корня.

Функции и структуры данных.

Для хранения элементов дерева реализован класс Node. У класса Node есть три приватных поля `int data` – хранит значение корня, `Node* left` – указатель на левое поддерево, `Node* right` – указатель на правое поддерево. У класса Node реализованы методы для работы с полями класса:

`Node* getLeft()` – возвращает указатель на левое поддерево.

`Node* getRight()` - возвращает указатель правое поддерево.

`int getData() const` – возвращает значение поля `data`.

`void setLeft(Node* l)` – присваивает значение полю `left`.

`void setRight(Node* r)` - присваивает значение полю `right`.

`void setData(int d)` – присваивает значение полю `data`.

Для хранения дерева и функций для его обработки создан класс `BinaryTree`. У класса есть поля `Node* tree` – указатель на корень дерева и `bool errorFlag` – флаг отвечающий за корректность дерева . Для работы с деревом реализованы следующие методы:

`BinaryTree(string& str)` – конструктор класса. В конструктор передаётся `string& str` – исходная строка с данными.

`Node* scanTree(string& str, int& pos)` – рекурсивная функция, которая вызывается в конструкторе, для создания листьев дерева. В функцию передаются `string& str` – исходная строка, `int& pos` – положение в строке. Функция возвращает указатель на листок.

`~BinaryTree()` – деструктор. В деструкторе вызывается рекурсивная функция `void destroy(Node*& buf)` для отчистки памяти.

`void destroy(Node*& buf)` – функция для отчистки памяти. В функцию передаётся указатель на лист по ссылке. Функция проходит по дереву в порядке ЛПК и отчищает память.

bool checkBST() – функция для проверки дерева на соответствие бинарному дереву поиска.

bool checkPiramid() – функция для проверки дерева на соответствие пирамиде.

Тестирование.

№	Входные данные	Результат
1	(10(6(4)(3))(5))	check Binary Search Tree Root: 10 go left to the 6; 6<10 OK Leaf: 6 go left to the 4; 4<6 OK go right to the 3; 3<6 ERROR Check Piramid Root: 10 go to the 6; 6<=10 OK Leaf: 6 go to the 4; 4<=6 OK go to the 3; 3<=6 OK go to the 5; 5<=10 OK Tree is a piramid
2	(5(4()(2))(6))	Tree is not BST or Piramid
3	(8(6)(12))	Tree is BST
4	(16(10(4(2)(6))(12(11)(14(13)(15)))) (22(18(17)(20))(24(23)(25))))	Tree is BST
5	(13(11(10(-1)(2))(5(4)))(13(8)(6)))	Tree is a piramid
6	(16(10(4)(12))(22(15)(24)))	Tree is not BST or Piramid
7	(1(2)(3))	Tree is not BST or Piramid
8	(5(-8(-12)(-4))(20(10)))	Tree is BST
9	1 2 3	error 1 2 3 is not a Tree or correct tree
10	(!(4))	not integer in tree

		error (! (4)) is not a Tree or correct tree
11	(1(2)(3))345	error (1(2)(3))345 is not a Tree or correct tree
12	((345)))	error ((345))) is not a Tree or correct tree

Вывод.

Были освоены принципы работы с бинарным деревом, и реализована данная структура данных на языке программирования C++.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ.

Файл structs.h

```
#ifndef AISD_LB3_STRUCTS_H
#define AISD_LB3_STRUCTS_H

#include <iostream>
#include <string>
#include <cstring>
#include <unistd.h>
#include <fstream>

using namespace std;

#endif //AISD_LB3_STRUCTS_H
```

Файл main.cpp

```
#include "Tree.h"
#include "InOut.h"

int main (int argc, char** argv){
    string input = cuinput(argc, argv);
    if(input.empty()){
        cout<<"ERROR";
        return 0;
    }
    string copy = string(input);
    BinaryTree tree = BinaryTree(copy);
    if(!tree.isErr() && copy.empty()) {
        cout<<input<<endl;
        if (tree.checkBST()) {
            cout << "Tree is BST\n";
            if (argc > 1) {
                ofstream outfile(optarg, ios::app);
                outfile << "Tree is BST\n";
            }
        }
    }
}
```



```

    } else if (tree.checkPiramid()) {
        cout << "Tree is a piramid\n";
        if (argc > 1) {
            ofstream outfile(optarg, ios::app);
            outfile << "Tree is a piramid\n";
        }
    } else {
        cout << "Tree is not BST or Piramid\n";
        if (argc > 1) {
            ofstream outfile(optarg, ios::app);
            outfile << "Tree is not BST or Piramid\n";
        }
    }
} else{
    cout<<"error\n"<<input<<" is not a Tree or correct tree\n";
}
return 0;
}

```

Файл InOut.h

```

#ifndef AISD_LB3_INOUT_H
#define AISD_LB3_INOUT_H

```

```

#include "structs.h"

```

```

string cuinput(int argc, char** argv);

```

```

#endif //AISD_LB3_INOUT_H

```

Файл InOut.cpp

```

#include "InOut.h"

```

```

string cuinput(int argc, char** argv){
    if(argc == 1) {
        cout<<"Write a statement: ";
        string res;
        getline(cin, res);
        return res;
    }
}

```

```

int option = 0;
while ((option = getopt(argc,argv,"hf:"))!=-1){
    switch (option) {
        case 'h': cout<<"If you want read from file use flaf -
f<filename>\n"; return "HELP";

        case 'f': cout<<"read from file - "<<optarg<<endl;
            ifstream infile(optarg);
            if (!infile) {
                cout << "> File can't be open!" << endl;
                return "";
            }
            string str;
            getline(infile, str);
            return str;
        }
    }
    return "";
}

```

Файл Node.h

```

#ifndef AISD_LB3_NODE_H
#define AISD_LB3_NODE_H

class Node {
private:
    Node* right;
    int data;
    Node* left;
public:
    Node(): right(nullptr), data(0), left(nullptr){}

    Node* getLeft();

    Node* getRight();

    int getData() const;

```

```

void setLeft(Node* l);

void setRight(Node* r);

void setData(int d);

};

```

Файл Node.cpp

```

#include "Node.h"

int Node::getData() const {
    return data;
}

Node* Node::getLeft() {
    return left;
}

Node* Node::getRight() {
    return right;
}

void Node::setData(int d) {
    data = d;
}

void Node::setLeft(Node *l) {
    left = l;
}

void Node::setRight(Node *r) {
    right = r;
}

```

```
}
```

Файл Tree.h

```
#ifndef AISD_LB3_TREE_H
#define AISD_LB3_TREE_H

#include "Node.h"
#include "structs.h"
#define MIN -30000
#define MAX 30000

class BinaryTree{
public:
    explicit BinaryTree(string& str);
    bool isErr() const;
    bool checkBST();
    bool checkPiramid();
    ~BinaryTree();

private:
    Node* tree;
    void skip (string& str, int& pos, int n = 1);
    int getNum(string& input, int& pos);
    Node* scanTree(string& str, int& pos);
    void destroy(Node*& buf);

    bool checkBst_left(Node* node, int min = MIN , int max = MAX, int indent
= 0);
    bool checkBst_right(Node* node, int min = MIN, int max = MAX, int indent
= 0);

    bool checkPiramid_leaf(Node* node, int max, int indent);
    bool errorFlag = false;
};

#endif //AISD_LB3_TREE_H
```

Файл Tree.cpp

```
#include "Tree.h"
#include "InOut.h"
```

```

BinaryTree::BinaryTree(string &str){
    int pos = 0;
    tree = scanTree(str, pos);
}

BinaryTree::~~BinaryTree() {
    destroy(this->tree);
}

bool BinaryTree::isErr() const {
    return errorFlag;
}

void BinaryTree::skip(string& str, int& pos, int n){
    if (str.length() >= n) {
        str = str.substr(n);
        pos++;
    }
}

int BinaryTree::getNum(string& str, int& pos){
    string strNum;
    while (isdigit(str[0]) || (strNum.length() == 0 && str[0] == '-')) {
        strNum += str[0];
        skip(str, pos, 1);
    }
    return stoi(strNum);
}

Node *BinaryTree::scanTree(string& str, int& pos) {
    if(str[0] == '(') {
        skip(str, pos, 1);
        if(str[0] == ')') {
            skip(str, pos, 1);
            return nullptr;
        }
    }
}

```

```

    }

    if(!(isdigit(str[0]) || str[0] == '-')){
        errorFlag = true;
        cout<<"not integer in tree\n";
        return nullptr;
    }

    Node *buf = new Node();
    buf->setData(getNum(str, pos));
    if(str[0] == ' '){
        skip(str, pos, 1);
        buf->setLeft(nullptr);
        buf->setRight(nullptr);
        return buf;
    }

    if(str[0] == '('){
        buf->setLeft(scanTree(str, pos));
        if(str[0] == '('){
            buf->setRight(scanTree(str, pos));
            if(str[0] == ')') {
                skip(str, pos, 1);
                return buf;
            } else {cout<<"error\n"; errorFlag = true;}
        }else if(str[0] == ')'){
            skip(str, pos, 1);
            return buf;
        }
    }

    }

    }

    else{ cout<<"error\n"; errorFlag = true;}
    return nullptr;
}

void BinaryTree::destroy(Node*& buf){
    if(buf != nullptr) {

```

```

        Node* left = buf->getLeft();
        Node* right = buf->getRight();
        destroy(left);
        destroy(right);
        delete buf;
    }
}

bool BinaryTree::checkBST(){
    if(tree != nullptr && !errorFlag){
        cout<<"check Binary Search Tree\n";
        cout<<"Root: "<< tree->getData()<<endl;
        int indent = 0;
        if(checkBst_left(tree->getLeft(), MIN, tree->getData(), indent) &&
            checkBst_right(tree->getRight(), tree->getData(), MAX, indent))
            return true;
        else return false;
    }
    return false;
}

bool BinaryTree::checkBst_left(Node* node, int min, int max, int indent){
    if(node != nullptr){
        for(int i = 0; i < indent; i++){cout<<" ";}
        cout<<"go left to the "<<node->getData()<<" ";
        if(node->getData() > min && node->getData() < max){
            if(min == MIN){
                cout<<node->getData()<<" "<<max<<" OK "<<endl;
            }else cout<<min<<" "<<node->getData()<<" "<<max<<" OK "<<endl;
            if(node->getLeft() != nullptr && node->getRight() != nullptr) {
                for (int i = 0; i <= indent; i++) { cout << " "; }
                cout << "Leaf: " << node->getData() << endl;
            }
            int old_max = max;
            max = node->getData();
            int old_min = min;

```

```

        min = node->getData();

        if(checkBst_left(node->getLeft(), old_min, max, indent+1) &&
checkBst_right(node->getRight(), min, old_max, indent+1)){

            return true;

        } else return false;

    } else{

        cout<<max<<" "<<node->getData()<<" ERROR"<<endl;

        return false;

    }

}

return true;

}

bool BinaryTree::checkBst_right(Node* node, int min, int max, int indent){

    if(node != nullptr){

        for(int i = 0; i < indent; i++){cout<<" ";}

        cout<<"go right to the "<<node->getData()<<" ";

        if(node->getData() > min && node->getData() < max){

            if(max == MAX){

                cout<<min<<" "<<node->getData()<<" OK "<<endl;

            }else cout<<min<<" "<<node->getData()<<" "<<max<<" OK "<<endl;

            if(node->getLeft() != nullptr && node->getRight() != nullptr) {

                for (int i = 0; i <= indent; i++) { cout << " "; }

                cout << "Leaf: " << node->getData() << endl;

            }

            int old_max = max;

            max = node->getData();

            int old_min = min;

            min = node->getData();

            if(checkBst_left(node->getLeft(), old_min, max, indent+1) &&
checkBst_right(node->getRight(), min, old_max, indent+1)){

                return true;

            } else return false;

        } else{

            cout<<node->getData()<<" "<<min<<" ERROR"<<endl;

            return false;

        }

    }

}

```



```

    }

    return true;

}

bool BinaryTree::checkPiramid(){
    if(tree!= nullptr && !errorFlag){
        cout<<"Check Piramid\n";
        cout<<"Root: "<< tree->getData()<<endl;
        int indent = 0;
        if(checkPiramid_leaf(tree->getLeft(), tree->getData(), indent) &&
checkPiramid_leaf(tree->getRight(), tree->getData(), indent)){
            return true;
        } else return false;
    }
    return false;
}

bool BinaryTree::checkPiramid_leaf(Node* node, int max, int indent){
    if(node != nullptr){
        for(int i = 0; i < indent; i++){cout<<" ";}
        cout<<"go to the "<<node->getData()<<" ";
        if(node->getData() <= max){
            cout<<node->getData()<<"<="<<max<<" OK "<<endl;
            if(node->getLeft() != nullptr && node->getRight() != nullptr) {
                for (int i = 0; i <= indent; i++) { cout << " "; }
                cout << "Leaf: " << node->getData() << endl;
            }
            max = node->getData();
            if(checkPiramid_leaf(node->getLeft(), max, indent+1) &&
checkPiramid_leaf(node->getRight(), max, indent+1)) {
                return true;
            } else return false;
        } else {
            cout<<node->getData()<<">="<<max<<" ERROR "<<endl;
            return false;
        }
    }
}

```

```
        }  
    }  
    return true;  
}
```