**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по курсовой работе**

**по дисциплине «Программирование»**

Тема: бинарное дерево.

|  |  |  |
| --- | --- | --- |
| Студент гр. 2372 |  | Гечис В.Р. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2023

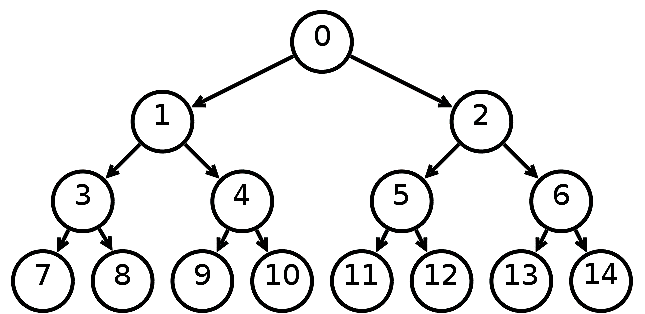
**Цель работы.**

изучение свойств и организации бинарных деревьев; получение практических навыков в работе с бинарными деревьями; определение преимуществ и недостатков подобных структур данных; проведение сравнительной характеристики скорости вставки, удаления и поиска элемента в бинарных деревьях.

## Понятие бинарного дерева

Бинарное дерево – это динамическая структура данных, состоящая из узлов, каждый из которых содержит кроме данных не более двух ссылок на различные бинарные деревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется корнем дерева.

Узел, не имеющий поддеревьев, называется листом. Исходящие узлы называются предками, входящие – потомками. Высота дерева определяется количеством уровней, на которых располагаются его узлы.



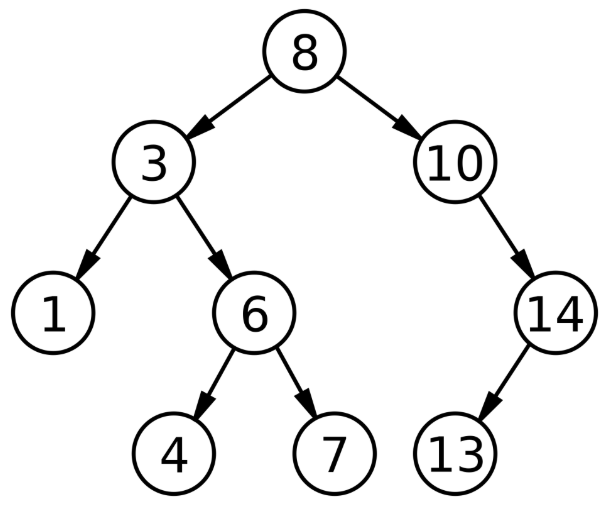
Дерево является рекурсивной структурой данных, поскольку каждое поддерево также является деревом. Действия с такими структурами лучше всего описывать с помощью рекурсивных алгоритмов.

Бинарные дерева нужны:

1. Если необходимо расположить информацию, связанную между собой некой иерархией. Примером является файловая система компьютера.
2. Если необходимо хранить данные, составленные в виде определённой структуры. Тогда хранение в виде бинарного дерева позволяет уменьшить скорость поиска данных и доступа к хранимой информации.
3. Если необходима высокая скорость добавления или удаления данных.
4. Если заранее неизвестен хранимый объем данных. Бинарные деревья, также, как и связанные списки, не имеют ограничения на количество узлов, поскольку узлы связаны указателями.

## Понятие бинарного дерева поиска

Бинарное дерево поиска – бинарное дерево, которое обладает дополнительными свойствами. Если дерево организовано таким образом, что для каждого узла все ключи левого поддерева меньше ключа этого узла, а все ключи его правого поддерева больше, то его можно называть бинарным деревом поиска.  
Иными словами, данные в бинарном дереве поиска хранятся в отсортированном виде. Одинаковые ключи не допускаются.



При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем.

Если искомое значение больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

Поскольку каждый корень дерева должен иметь три части, логичнее всего представить его в виде следующей рекурсивной структуры:

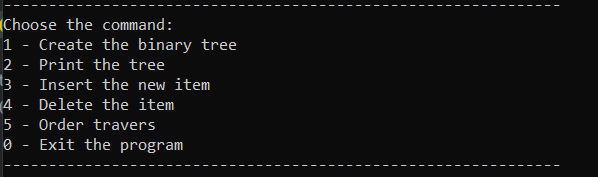
Сбалансированное бинарное дерево поиска – это бинарное дерево поиска с логарифмической высотой.

Для данного случая это правило необязательно, в отличие от AVL или красно-черных деревьев, где строгое определение оперирует разницей глубины самого глубоко и самого неглубокого листа и отношением глубины самого глубоко и самого неглубокого листа соответственно.

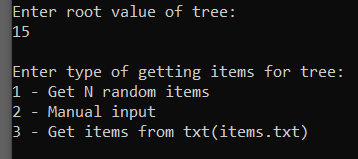
Сбалансированное бинарное дерево поиска применяется, когда необходимо осуществлять быстрый поиск элементов, чередующийся со вставками новых элементов и удалением существующих.

**Выполнение работы.**

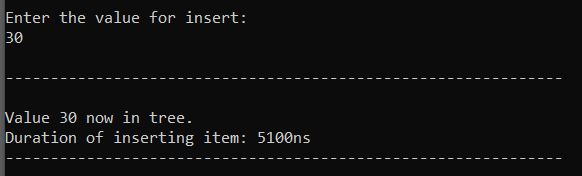
1. Выбор типа заполнения и заполнения дерева:



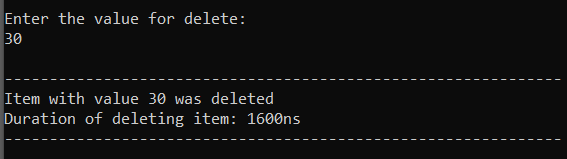
1. Формирование дерева:



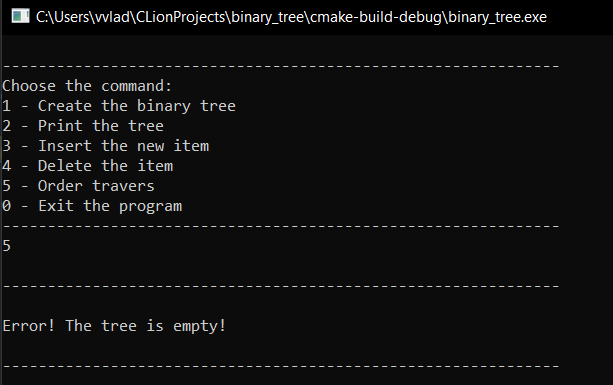
1. Вставка нового элемента:



1. Удаление элемента и дальнейшая балансировка:



5.Проверка на ошибки



**Полный код программы:**

#include <iostream>  
#include <windows.h>  
#include <fstream>  
#include <sstream>  
#include <chrono>  
  
using namespace std;  
  
  
struct Tree{  
 int data;  
 Tree\* left;  
 Tree\* right;  
};  
  
  
void insert(int key, Tree\* root) {  
 if (key < root->data)  
 {  
 if (root->left)  
 insert(key, root->left);  
 else  
 {  
 root->left = new Tree;  
 root->left->data = key;  
 root->left->left = NULL;  
 root->left->right = NULL;  
 }  
 }  
 else if (key >= root->data) {  
  
 if (root->right)  
 insert(key, root->right);  
 else  
 {  
 root->right = new Tree;  
 root->right->data = key;  
 root->right->left = NULL;  
 root->right->right = NULL;  
 }  
}  
}  
  
Tree\* getFreeTree(int &size) {  
 Tree\* root = (Tree\*)malloc(sizeof(Tree));  
 root->left = root->right = NULL;  
 int value;  
 cout << "\nEnter root value of tree:\n";  
 cin >> value;  
 root->data = value;  
  
 int userChoice;  
 cout << "\nEnter type of getting items for tree:\n"  
 "1 - Get N random items\n"  
 "2 - Manual input\n"  
 "3 - Get items from txt(items.txt)\n";  
 cin >> userChoice;  
 if(userChoice ==1){  
 cout << "\nEnter count of items for tree:\n";  
 cin >> size;  
 size+=1;  
 for(int i = 0; i <size; i++){  
 insert((-99 + (std::rand() % 199)),root);  
 }  
 return root;  
 }  
 if(userChoice == 2) {  
 int userValue=1;  
 cout << "\nEnter items for tree: (Enter 0 for stop)\n";  
 while (userValue!=0) {  
 cin >> userValue;  
 if(userValue) {  
 insert(userValue, root);  
 size += 1;  
 }  
// }  
 }  
 size+=1;  
 return root;  
 }  
  
 if(userChoice==3){  
 ifstream database;  
 database.open("C:\\Users\\vvlad\\CLionProjects\\binary\_tree\\items.txt");  
 if (!database.is\_open()) {  
 cout << endl << "Error! File isn't open";  
 }  
 else{  
 string data;  
 while(getline(database,data)){  
// cout << stoi(data) << endl;  
 insert(stoi(data),root);  
 size +=1;  
 }  
 }  
 size+=1;  
 database.close();  
 return root;  
 }  
  
}  
  
void destroyTree(Tree \*root)  
{  
 if(root!=NULL)  
 {  
 destroyTree(root->left);  
 destroyTree(root->right);  
 delete root;  
 }  
}  
void sizeTree(Tree\* root, int &size\_count) {  
 if (root) {  
 size\_count++;  
 sizeTree(root->left,size\_count);  
 sizeTree(root->right,size\_count);  
 }  
}  
void setPositionXY (short x, short y)  
{  
 HANDLE StdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);  
 COORD coord = {x, y};  
 SetConsoleCursorPosition(StdOut, coord);  
}  
  
void megaline(int startX,int startY, int finishX){  
 if(finishX>startX) {  
 for (int x = startX ; x < finishX; x++) {  
 setPositionXY(x, startY);  
 cout << "-";  
 }  
 }  
 else{  
 for (int x = startX ; x > finishX; x--) {  
 setPositionXY(x, startY);  
 cout << "-";  
 }  
 }  
 setPositionXY(startX,startY);  
}  
void print (Tree \* root, short x, short y, short height, char side)  
{  
 if (root) {  
 if (height > 0 && side != 'k') {  
 if (side == 'l') {  
 x -= (height + 1) \* 6;  
 } else {  
 x += (height + 1) \* 6;  
 }  
  
 } else if (side != 'k')  
 if (side == 'l'){  
 x -= 4;  
 }  
 else {  
 x += 4;  
 }  
  
 setPositionXY(x,y+=2);  
  
 height--;  
 cout << root->data;  
 print (root->left,x,y,height,'l');  
 print (root->right,x,y,height,'r');  
 }  
}  
  
void preOrderTravers(Tree\* root) {  
 if (root) {  
 std::cout << root->data << " ";  
 preOrderTravers(root->left);  
 preOrderTravers(root->right);  
 }  
}  
void inOrderTravers(Tree\* root) {  
 if (root) {  
 inOrderTravers(root->left);  
 std::cout << root->data << " ";  
 inOrderTravers(root->right);  
 }  
}  
void postOrderTravers(Tree\* root) {  
 if (root) {  
 postOrderTravers(root->left);  
 postOrderTravers(root->right);  
 std::cout << root->data << " ";  
 }  
}  
int heightOfTree(Tree \* root)  
{  
 if(root == 0)  
 return 0;  
 int left, right;  
 if (root->left) {  
 left = heightOfTree(root->left);  
 }else  
 left = -1;  
 if (root->right) {  
 right = heightOfTree(root->right);  
 }else  
 right = -1;  
 int max = left > right ? left : right;  
 return max+1;  
}  
Tree \*getItem(Tree\* root, int userValue){  
 if (root) {  
 if(root->data==userValue){  
 return root;  
 }  
 else if(root->data>userValue) {  
 getItem(root->left, userValue);  
 }  
 else {  
 getItem(root->right, userValue);  
 }  
 }  
}  
Tree \*getParent(Tree\* root, int userValue){  
 if (root) {  
 if(root->left->data==userValue || root->right->data==userValue){  
 return root;  
 }  
 else if (root->data>userValue) {  
 getParent(root->left, userValue);  
 }  
 else {  
 getParent(root->right, userValue);  
 }  
  
 }  
}  
Tree\* findMaximum(Tree\* node){  
 while(node->right){  
 node = node->right;  
 }  
 return node;  
}  
Tree\* deleteItem(Tree\* root, int value){  
 if(!root){  
 return root;  
 }  
  
 if(value < root->data){  
 root->left = deleteItem(root->left, value);  
 }  
 else if(value > root->data){  
 root->right = deleteItem(root->right, value);  
 }  
 else{  
 if(!root->left && !root->right){  
 delete root;  
 root = nullptr;  
 }  
 else if(!root->left){  
 Tree\* temp = root;  
 root = root->left;  
 delete temp;  
 }  
 else if(!root->right){  
 Tree\* temp = root;  
 root = root->left;  
 delete temp;  
 }  
 else{  
 Tree\* temp = findMaximum(root->left);  
 root->data = temp->data;  
 root->left = deleteItem(root->left, temp->data);  
 }  
 }  
 return root;  
}  
  
void outputFile(Tree \* root, short x, short y, short height, char side)  
{  
 ofstream database;  
 database.open("C:\\Users\\vvlad\\CLionProjects\\binary\_tree\\output\_tree.txt");  
 if(database.is\_open()){  
 cout << "\nopen";  
 }  
 if (!database.is\_open()) {  
 cout << '\n' << "Saving error!";  
 }  
 if (root) {  
 if (height > 0 && side != 'k') {  
 if (side == 'l') {  
 x -= (height + 1) \* 6;  
 } else {  
 x += (height + 1) \* 6;  
 }  
  
 } else if (side != 'k')  
 if (side == 'l'){  
 x -= 4;  
 }  
 else {  
 x += 4;  
 }  
  
 setPositionXY(x,y+=2);  
 for(int i = 0; i<y;i++){  
 database << "\n";  
 }  
 for (int i = 0; i <x;i++){  
 database << " ";  
 }  
 height--;  
  
 database << root->data;  
 database.close();  
 outputFile(root->left,x,y,height,'l');  
 outputFile(root->right,x,y,height,'r');  
 }  
  
}  
int main(){  
  
 Tree \* root = new Tree;  
 root = 0;  
 int size=0;  
 int userValue;  
 int height = 0;  
 int cycleMain;  
 while(true) {  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Choose the command:"  
 "\n1 - Create the binary tree"  
 "\n2 - Print the tree"  
 "\n3 - Insert the new item"  
 "\n4 - Delete the item"  
 "\n5 - Order travers"  
 "\n0 - Exit the program";  
 cout << "\n--------------------------------------------------------------\n";  
 cin >> cycleMain;  
 if (!cycleMain) {  
 destroyTree(root);  
 return 0;  
 }  
 if (cycleMain == 1) {  
 system("cls");  
 root = getFreeTree(size);  
 height = heightOfTree(root);  
 system("cls");  
 print(root, 80, 3, height, 'k');  
 setPositionXY(1, 20);  
 } else if (root) {  
 if (cycleMain == 2) {  
 system("cls");  
// height = heightOfTree(root);  
 print(root, 80, 0, height, 'k');  
 setPositionXY(1, 20);  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Binary tree contains " << size << " items. Height of the tree: " << height;  
 cout << "\n--------------------------------------------------------------\n";  
  
 }  
 if (cycleMain == 3) {  
 system("cls");  
 cout << "\nEnter the value for insert:\n";  
 cin >> userValue;  
 auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 insert(userValue, root);  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "\nValue " << userValue << " now in tree.\n";  
 cout << "Duration of inserting item: " << end - start << "ns";  
 cout << "\n--------------------------------------------------------------\n";  
 }  
 if (cycleMain == 4) {  
 system("cls");  
 cout << "\nEnter the value for delete:\n";  
 cin >> userValue;  
 auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 deleteItem(root, userValue);  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Item with value " << userValue << " was deleted";  
 cout << "\nDuration of deleting item: " << end - start << "ns";  
 cout << "\n--------------------------------------------------------------\n";  
 }  
 if (cycleMain == 5) {  
 system("cls");  
 print(root, 80, 3, height, 'k');  
 setPositionXY(1, 20);  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Straight travel in tree: ";  
 auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 preOrderTravers(root);  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration: " << end - start << "ns";  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Symmetric travel in tree: ";  
 start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 inOrderTravers(root);  
 end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration: " << end - start << "ns";  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Reversed travel in tree: ";  
 start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 postOrderTravers(root);  
 end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration: " << end - start << "ns";  
 cout << "\n--------------------------------------------------------------\n";  
  
 }  
 if(cycleMain==6){  
 system("cls");  
 outputFile(root, 80, 3, height, 'k');  
 cout << "\nTree was saved";  
 setPositionXY(1, 20);  
 }  
 }  
 else {cout << "\n--------------------------------------------------------------\n";  
 cout << "\nError! The tree is empty!\n";  
 cout << "\n--------------------------------------------------------------\n";  
 }  
 }  
}