**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 поиска.

|  |  |  |
| --- | --- | --- |
| Студентка гр. 2372 |  | Соколовский В.Д. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2023

**Цель работы.**

Изучение свойств и организации деревьев как структуры данных; получение практических навыков в работе с бинарным деревом поиска; определение преимуществ и недостатков структуры данных вида дерева; проведение сравнительной характеристики скорости вставки, удаления и поиска элемента в различных структурах данных.

**Основные теоретические положения.**

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.

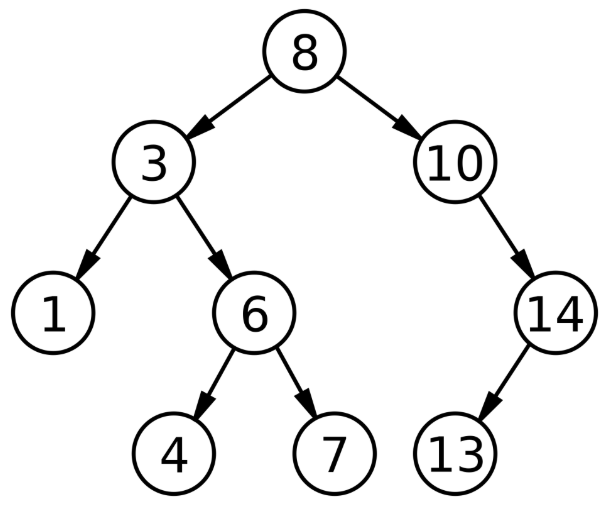
**Бинарные дерева нужны:**

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.



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.

## **Добавление и удаление узлов**

Если дерево пустое, то создается новый узел, который добавляется в дерево. Если дерево не пустое, то значение ключа добавляемого элемента сравнивается со значением ключа в узле, начиная от корня.

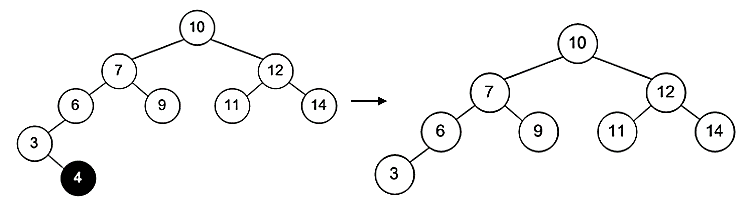
Так как одинаковые ключи не допускаются, перед вставкой нового элемент нужно проверить, нет ли уже в бинарном дереве такого элемента. Для этого необходимо начать обход дерева с корневого узла и проверить, не превосходит ли значение корневого узла добавляемого значения.

Если корневой узел больше добавляемого элемента, то необходимо переместиться в левое дочернее дерево. В противном случае – в правое. После выполнения данной проверки можно переходить к добавлению узла в бинарное дерево.

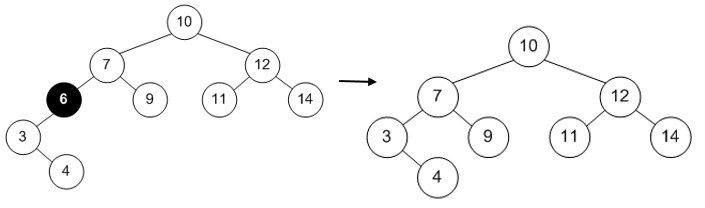
С удалением узла дело обстоит немного иначе. Существует несколько возможных ситуаций:

1)      у узла нет наследников (узел является листом);  
 2)      у узла имеется наследник, притом только один;  
 3)      у узла есть оба наследника.

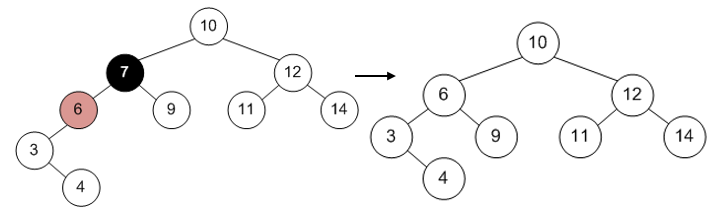
Если у узла нет наследника, то необходимо просто удалить этот узел, а у его родителя обнулить указатель на него. Это самая простая ситуация.



Все слегка усложняется, когда у удаляемого узла имеются наследники.



Если же у удаляемого два наследника, то узел не удаляется, а его значение заменяется на максимум левого поддерева. После этого удаляется максимум левого поддерева.



Максимум левого поддерева имеет не более одного наследника, так что он удаляется просто, аналогично ситуации, рассмотренной выше. Функция вставки аналогична функции поиска: необходимо пройти по дереву и вставить узел в нужное место. Для этого следует выбрать левое или правое поддерево корневого узла, а затем рекурсивно перемещаться по выбранному поддереву до тех пор, пока не будет найдена позиция для вставки узла.

 Если же необходимо удалить дерево, то необходимо реализовать рекурсивную функцию удаления.

**Обходы дерева**

Дерево – это связный граф без циклов. Связность означает, что из любой вершины в любую другую существует путь по ребрам. Отсутствие циклов означает, что данный путь – единственный.

Обход графа – это систематическое посещение всех его вершин по одному разу каждой. Существует два вида обхода графа:

 1) поиск в глубину;

 2) поиск в ширину.

Обходу в ширину в графе соответствует обход по уровням бинарного дерева. При данном обходе идет посещение узлов по принципам сверху вниз и слева направо. Обходу в глубину в графе соответствуют три вида обходов бинарного дерева: прямой, симметричный и обратный.

**Постановка задачи.**

Необходимо реализовать программу, которая выполняет следующие действия.

1.   Формирование бинарного дерева из *N* элементов:

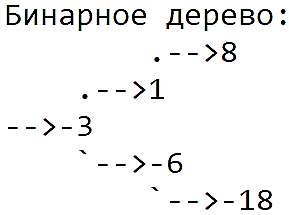
a) пользователь вводит количество элементов *N* бинарного дерева, которое автоматически заполняется случайными числами (–99 до 99);

б) пользователь вводит в консоль элементы массива, *N* определяется автоматически по количеству введенных элементов;

в) элементы считываются с файла, в котором хранится массив из чисел, *N* – количество элементов в файле.

Определение скорости формирования бинарного дерева.

2.   Вывод в консоль и файл tree бинарного дерева. Бинарное дерево должно иметь подобный вид:



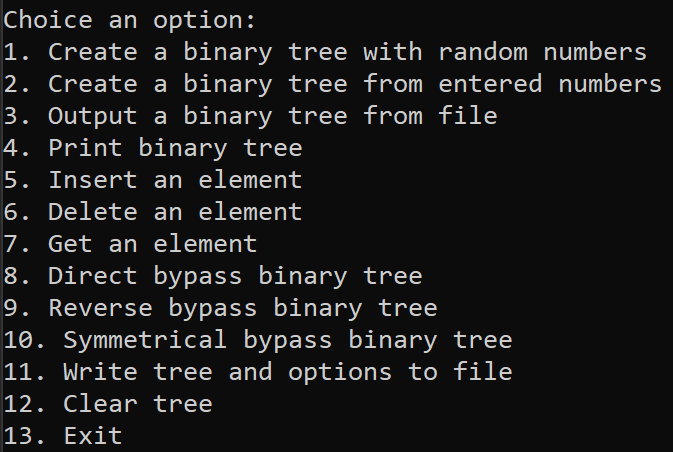
3.   Определение скорости вставки, удаления и получения элемента дерева. В отчете сравните скорость работы бинарного дерева с линейной структурой (двусвязным списком или динамическим массивом) и сделайте выводы.

4. Прямой обход, обратный обход и обход в ширину бинарного дерева.

5.  Генерация заданий к практической работе по бинарным деревьям. Необходимо сгенерировать задания в файл output\_task в количестве вариантов, которые введет пользователь. В файл output\_key необходимо вывести короткие ответы к заданиям. В файл output\_ans необходимо вывести развернутые ответы к заданиям. Должны быть представлены задания на создание бинарного дерева поиска из массива значений, удаление и вставка элементов в бинарное дерево поиска.

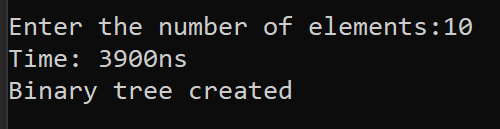
**Выполнение работы.**

1. Пользователю предоставляется интерактивное меню:

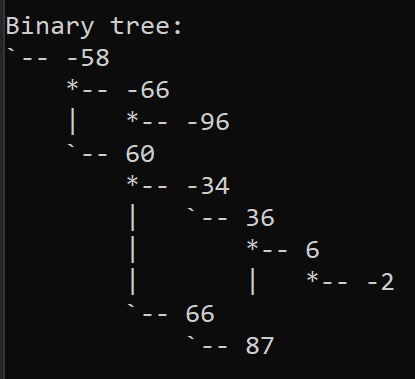


1. Формирование бинарного дерева размерности N, где пользователь вводит количество элементов, которые будут автоматически заполнять случайными числами (0 до 99);

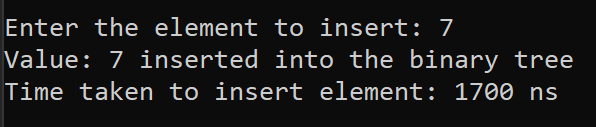
Определение скорости его создания;



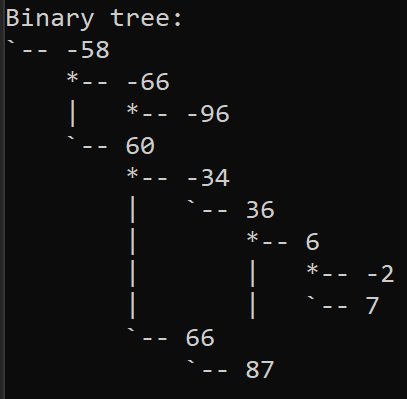
1. Вывод дерева:



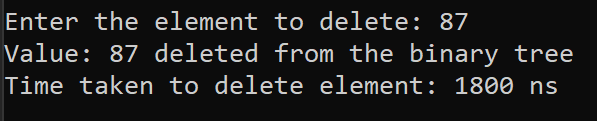
1. Вставка нового элемента и определение его скорости:



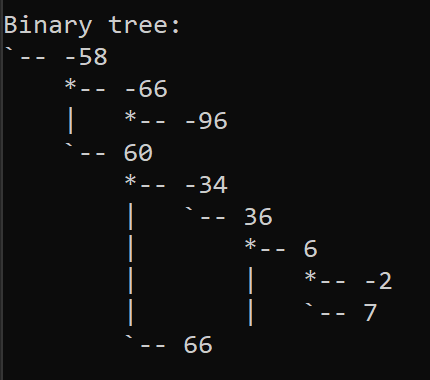
Вывод бинарного дерева:



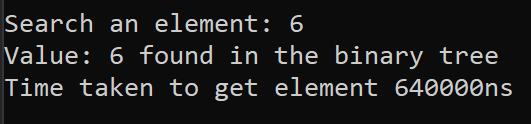
1. Удаление элемента и определение его скорости:



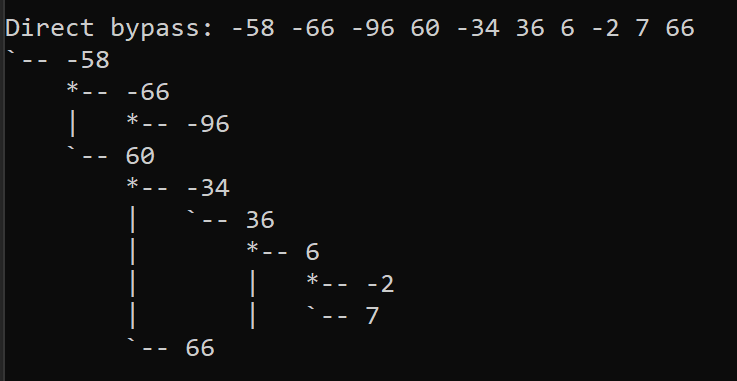
Вывод бинарного дерева:



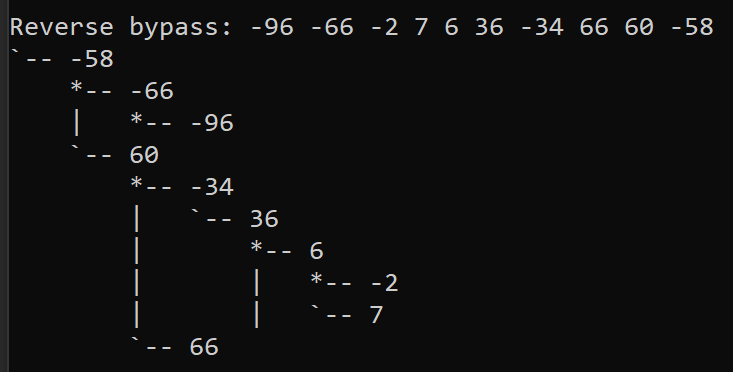
1. Поиск элемента и определение его скорости:



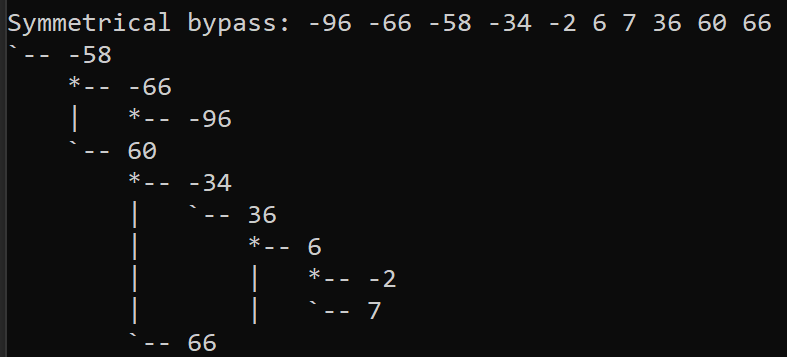
1. Прямой обход дерева:



1. Обратный обход:



1. Симметричный обход:



**Вывод:**

В ходе работы были изучены свойства и организации деревьев как структуры данных; определены преимущества и недостатки структуры данных вида дерева; проведены сравнительной характеристики скорости вставки, удаления и поиска элемента в различных структурах данных.

Приложение А

Полный код программы

#include <iostream>  
#include <cstdlib>  
#include <fstream>  
#include <vector>  
#include <chrono>  
  
using namespace std;  
using namespace chrono;  
  
  
  
struct Tree{  
 int key;  
 Tree\* left;  
 Tree\* right;  
  
 Tree(int value){  
 key = value;  
 left = nullptr;  
 right = nullptr;  
 }  
};  
  
  
  
Tree\* insertNode(Tree\* root, int value){  
 if(root == nullptr){  
 return new Tree(value);  
 }  
  
 if(value < root->key){  
 root->left = insertNode(root->left, value);  
 }  
 else if (value > root->key){  
 root->right = insertNode(root->right, value);  
 }  
  
 return root;  
}  
  
  
  
Tree\* createTree(const vector<int>& array){  
 Tree\* root = nullptr;  
 for(int num : array) {  
 root = insertNode(root, num);  
 }  
 return root;  
}  
  
  
  
Tree\* searchNode(Tree \*root, int value){  
 if(root == nullptr || root->key == value){  
 return root;  
 }  
 if(value < root->key){  
 return searchNode(root->left, value);  
 }  
 else{  
 return searchNode(root->right, value);  
 }  
}  
  
  
  
Tree\* findMinimum(Tree\* node){  
 while(node->left != nullptr){  
 node = node->left;  
 }  
 return node;  
}  
  
  
  
Tree\* deleteNode(Tree\* root, int value){  
 if(root == nullptr){  
 return root;  
 }  
  
 if(value < root->key){  
 root->left = deleteNode(root->left, value);  
 }  
 else if(value > root->key){  
 root->right = deleteNode(root->right, value);  
 }  
 else{  
 if(root->left == nullptr && root->right == nullptr){  
 delete root;  
 root = nullptr;  
 }  
 else if(root->left == nullptr){  
 Tree\* temp = root;  
 root = root->left;  
 delete temp;  
 }  
 else if(root->right == nullptr){  
 Tree\* temp = root;  
 root = root->left;  
 delete temp;  
 }  
 else{  
 Tree\* temp = findMinimum(root->right);  
 root->key = temp->key;  
 root->right = deleteNode(root->right, temp->key);  
 }  
 }  
 return root;  
}  
  
  
  
void printTree(Tree\* root, string indent = "", bool isLeft = false){  
 if(root == nullptr){  
 return;  
 }  
  
 cout << indent;  
  
 if(isLeft){  
 cout << "\*-- ";  
 indent += "| ";  
 }  
 else{  
 cout << "`-- ";  
 indent += " ";  
 }  
  
 cout << root->key << endl;  
  
 printTree(root->left, indent, true);  
 printTree(root->right, indent, false);  
}  
  
  
  
void writeTree(Tree\* root, ofstream& outFile, string indent, bool isLeft){  
 if(root == nullptr){  
 return;  
 }  
  
 outFile << indent;  
  
 if(isLeft){  
 outFile << "\*-- ";  
 indent += "| ";  
 }  
 else{  
 outFile << "`-- ";  
 indent += " ";  
 }  
  
 outFile << root->key << endl;  
  
 writeTree(root->left, outFile, indent, false);  
 writeTree(root->right, outFile, indent, true);  
}  
  
  
  
void writeOperations(Tree\* root, const string& filename, int insertedValue, int deletedValue, int searchValue){  
 ofstream outFile(filename, ios::app);  
 if(!outFile){  
 cout << "Failed to open file: " << filename << endl;  
 return;  
 }  
  
  
 outFile << "Inserted value: " << insertedValue << endl;  
 outFile << "Deleted value: " << deletedValue << endl;  
 outFile << "Searched value: " << searchValue << endl;  
 outFile << "\nBinary tree:" << endl;  
  
 writeTree(root, outFile, "", true);  
 outFile << "\n------------------------------------------------------------------------------------\n\n";  
 outFile.close();  
  
 cout << "Binary tree written to file: " << filename << endl;  
}  
  
  
  
void bypassDirect(Tree\* root){  
 if(root == nullptr){  
 return;  
 }  
  
 cout << root->key << " ";  
 bypassDirect(root->left);  
 bypassDirect(root->right);  
}  
  
  
  
void bypassReverse(Tree\* root){  
 if(root == nullptr){  
 return;  
 }  
  
 bypassReverse(root->left);  
 bypassReverse(root->right);  
 cout << root->key << " ";  
}  
  
  
  
void bypassSymmetrical(Tree\* root){  
 if(root == nullptr){  
 return;  
 }  
 bypassSymmetrical(root->left);  
 cout << root->key << " ";  
 bypassSymmetrical(root->right);  
}  
  
  
  
int main() {  
  
 Tree\* root = nullptr;  
 vector<int> elements;  
 int insertedValue, deletedValue, searchValue;  
  
 while (true) {  
 int choice;  
 cout << "\nChoice an option:\n";  
 cout << "1. Create a binary tree with random numbers\n";  
 cout << "2. Create a binary tree from entered numbers\n";  
 cout << "3. Output a binary tree from file\n";  
 cout << "4. Print binary tree\n";  
 cout << "5. Insert an element\n";  
 cout << "6. Delete an element\n";  
 cout << "7. Get an element\n";  
 cout << "8. Direct bypass binary tree\n";  
 cout << "9. Reverse bypass binary tree\n";  
 cout << "10. Symmetrical bypass binary tree\n";  
 cout << "11. Write tree and options to file\n";  
 cout << "12. Clear tree\n";  
 cout << "13. Exit\n";  
 cout << "Choice: ";  
 cin >> choice;  
 cout << "\n------------------------------------------------------------------------------------\n";  
 switch (choice) {  
 case 1: {  
 int numElements;  
 cout << "Enter the number of elements:";  
 cin >> numElements;  
  
 for(int i = 0; i < numElements; i++){  
 int num = rand() % 199 - 99;  
 elements.push\_back(num);  
 }  
 auto start = high\_resolution\_clock::now();  
  
 root = createTree(elements);  
  
 auto end = high\_resolution\_clock::now();  
  
 auto duration = duration\_cast<nanoseconds>(end - start);  
 cout << "Time: " << duration.count() << "ns\n";  
  
 cout << "Binary tree created\n";  
 cout << "\n------------------------------------------------------------------------------------\n";  
 break;  
 }  
 case 2: {  
 int num, numElements;  
 cout << "Enter the number of elements:\n";  
 cin >> numElements;  
  
 cout << "Enter elements:";  
 for(int i = 0; i < numElements; i++){  
 cin >> num;  
 elements.push\_back(num);  
 }  
 auto start = high\_resolution\_clock::now();  
  
 root = createTree(elements);  
  
 auto end = high\_resolution\_clock::now();  
  
 auto duration = duration\_cast<nanoseconds>(end - start);  
 cout << "Time: " << duration.count() << "ns\n";  
  
 cout << "Binary tree created\n";  
 cout << "\n------------------------------------------------------------------------------------\n";  
 break;  
 }  
 case 3: {  
 ifstream File;  
 File.open("C:\\Users\\User\\CLionProjects\\coursework2\\output\_tree.txt");  
 if(File.is\_open()){  
 char ch;  
 while(File.get(ch)){  
 cout << ch;  
 }  
 }  
 else{  
 cout << "Error";  
 }  
 File.close();  
 break;  
 }  
 case 4: {  
 if(root == nullptr){  
 cout << "Binary tree is empty\n";  
 }  
 else{  
 cout << "Binary tree: \n";  
 printTree(root);  
 cout << "\n------------------------------------------------------------------------------------\n";  
 }  
 break;  
 }  
 case 5: {  
 int value;  
 cout << "Enter the element to insert: ";  
 cin >> value;  
  
 auto start = high\_resolution\_clock::now();  
  
 insertNode(root, value);  
 insertedValue = value;  
  
 auto end = high\_resolution\_clock::now();  
  
 cout << "Value: " << value << " inserted into the binary tree\n";  
  
 auto duration = duration\_cast<nanoseconds>(end - start);  
 cout << "Time taken to insert element: " << duration.count() << " ns\n";  
  
 cout << "\n------------------------------------------------------------------------------------\n";  
 break;  
 }  
 case 6: {  
 int value;  
 cout << "Enter the element to delete: ";  
 cin >> value;  
  
 auto start = high\_resolution\_clock::now();  
 deleteNode(root, value);  
 deletedValue = value;  
  
 auto end = high\_resolution\_clock::now();  
  
 cout << "Value: " << value << " deleted from the binary tree\n";  
  
 auto duration = duration\_cast<nanoseconds>(end - start);  
 cout << "Time taken to delete element: " << duration.count() << " ns\n";  
  
 cout << "\n------------------------------------------------------------------------------------\n";  
 break;  
 }  
 case 7: {  
 int value;  
 cout << "Search an element: ";  
 cin >> value;  
  
 auto start = high\_resolution\_clock::now();  
  
 Tree \*foundNode = searchNode(root, value);  
 if(foundNode){  
 searchValue = value;  
 cout << "Value: " << value << " found in the binary tree\n";  
 }  
 else{  
 cout << "Value: " << value << " not found in the binary tree\n";  
 }  
 auto end = high\_resolution\_clock::now();  
  
 auto duration = duration\_cast<nanoseconds>(end - start);  
 cout << "Time taken to get element " << duration.count() << "ns\n";  
 cout << "\n------------------------------------------------------------------------------------\n";  
 break;  
 }  
 case 8: {  
 cout << "Direct bypass: ";  
 bypassDirect(root);  
 cout << endl;  
 printTree(root);  
 cout << "\n------------------------------------------------------------------------------------\n";  
 break;  
 }  
 case 9: {  
 cout << "Reverse bypass: ";  
 bypassReverse(root);  
 cout << endl;  
 printTree(root);  
 cout << "\n------------------------------------------------------------------------------------\n";  
 break;  
 }  
 case 10: {  
 cout << "Symmetrical bypass: ";  
 bypassSymmetrical(root);  
 cout << endl;  
 printTree(root);  
 cout << "\n------------------------------------------------------------------------------------\n";  
 break;  
 }  
 case 11: {  
 string path = "C:\\Users\\User\\CLionProjects\\coursework2\\tree.txt";  
  
 ofstream fout;  
  
 fout.open(path, ofstream::app);  
  
 if(!fout.is\_open()){  
 cout << "File opening error\n";  
 }  
 else{  
 writeOperations(root, path, insertedValue, deletedValue, searchValue);  
 }  
 break;  
 }  
 case 12: {  
 elements.clear();  
 cout << "Tree is clean\n";  
 cout << "\n------------------------------------------------------------------------------------\n";  
 break;  
 }  
 case 13: {  
 return 0;  
 }  
 default: {  
 cout << "Invalid option. Please try again\n";  
 cout << "\n------------------------------------------------------------------------------------\n";  
 break;  
 }  
 }  
 }  
}