**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”

|  |  |  |
| --- | --- | --- |
| Студент гр. 2372 |  | Юрин А. А. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2023

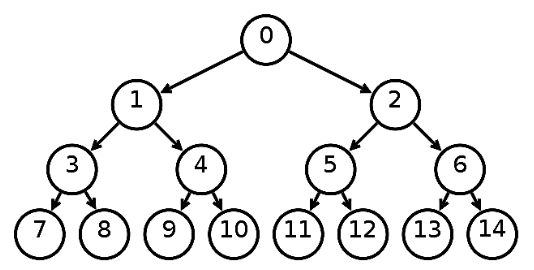
**Цель работы.**

изучение свойств и организации АВЛ-Деревьев и деревьев бинарного поиска; получение практических навыков в работе с АВЛ и БП-деревьями; определение преимуществ и недостатков подобных структур данных; проведение сравнительной характеристики скорости вставки, удаления и поиска элемента в БП дереве.

**Основные теоретические положения.**

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:

Если необходимо расположить информацию, связанную между собой некой иерархией. Примером является файловая система компьютера.

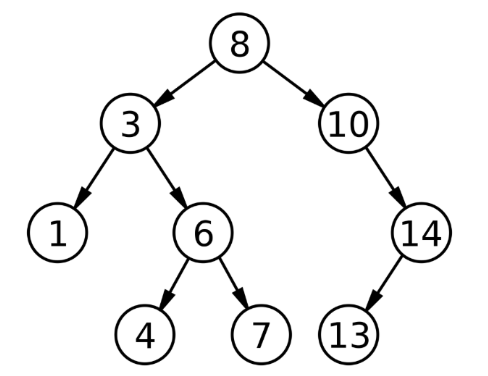
Если необходимо хранить данные, составленные в виде определённой структуры. Тогда хранение в виде бинарного дерева позволяет уменьшить скорость поиска данных и доступа к хранимой информации.

Если необходима высокая скорость добавления или удаления данных.

Если заранее неизвестен хранимый объем данных. Бинарные деревья, также, как и связанные списки, не имеют ограничения на количество узлов, поскольку узлы связаны указателями.

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 вставке элемента в бинарное дерево поиска необходимо учитывать два случая:

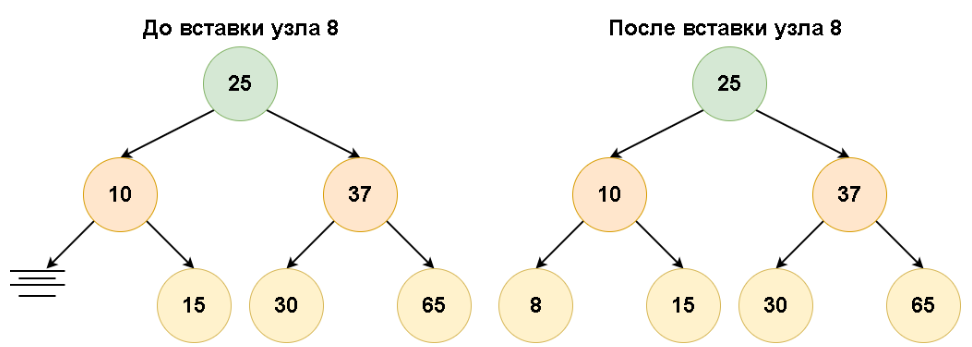
Дерево пустое;

Дерево непустое.

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.



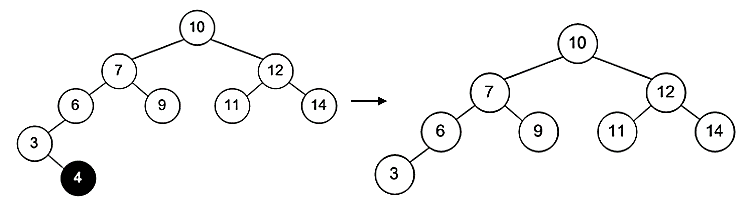
С удалением узла дело обстоит немного иначе. Существует несколько возможных ситуаций:

1) у узла нет наследников (узел является листом);

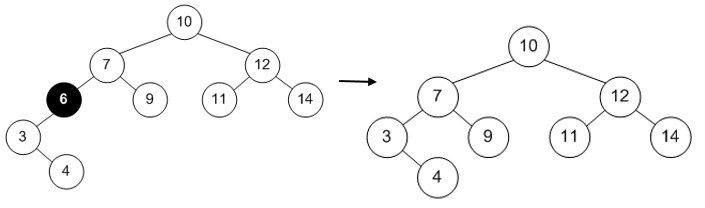
2) у узла имеется наследник, притом только один;

3) у узла есть оба наследника.

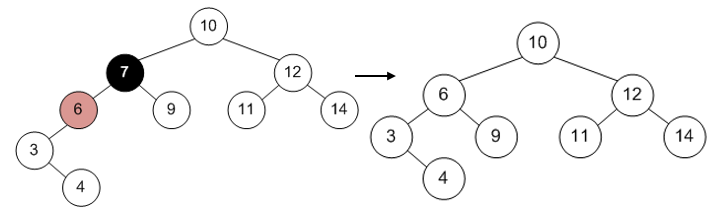
Если у узла нет наследника, то необходимо просто удалить этот узел, а у его родителя обнулить указатель на него. Это самая простая ситуация.



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.

**Постановка задачи.**

Необходимо реализовать программу, которая выполняет следующие действия.

Для БП деревьев:

Необходимо реализовать программу, которая выполняет следующие действия.

1. Формирование бинарного дерева из N элементов:

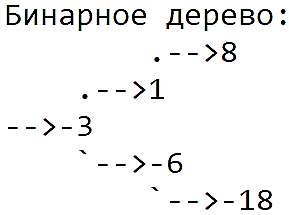
a) пользователь вводит количество элементов N бинарного дерева, которое автоматически заполняется случайными числами (–99 до 99);

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.**

Структура Бинарного дерева и функции:

#include <iostream>  
#include <string>  
#include <chrono>  
  
using namespace std;  
using namespace chrono;  
  
struct Node  
{  
 int value;  
 Node\* left;  
 Node\* right;  
};  
  
Node\* search(int value, Node\* root) {  
 if (root != nullptr)  
 {  
 if (value == root->value)  
 return root;  
 if (value < root->value)  
 return search(value, root->left);  
 else  
 return search(value, root->right);  
 }  
 else return nullptr;  
}  
  
Node\* create(int value) {  
 Node\* root = new Node;  
 root->left = nullptr;  
 root->right = nullptr;  
 root->value = value;  
 return root;  
}  
  
Node\* getMin(Node\* root) {  
 while (root->left)  
 root = root->left;  
 return root;  
}  
  
Node\* deleteB(int value, Node\* root) {  
 if (root == nullptr)  
 return nullptr;  
 if (value < root->value)  
 root->left = deleteB(value, root->left);  
 else if (value > root->value)  
 root->right = deleteB(value, root->right);  
 else {  
 if (root->left == nullptr && root->right == nullptr)  
 root = nullptr;  
 else if (root->left == nullptr || root->right == nullptr)  
 root = root->left ? root->left : root->right;  
 else {  
 Node\* minnode = getMin(root->right);  
 root->value = minnode->value;  
 root->right = deleteB(minnode->value, root->right);  
 }  
 }  
 return root;  
}  
  
void insert(int value, Node\* root) {  
 if (!root) return;  
 if (value < root->value)  
 {  
 if (root->left != nullptr)  
 insert(value, root->left);  
 else {  
 root->left = new Node;  
 root->left->value = value;  
 root->left->left = nullptr;  
 root->left->right = nullptr;  
 }  
 } else if (value >= root->value) {  
 if (root->right != nullptr)  
 insert(value, root->right);  
 else {  
 root->right = new Node;  
 root->right->value = value;  
 root->right->left = nullptr;  
 root->right->right = nullptr;  
 }  
 }  
}  
  
void destroy(Node\* root) {  
 if(root!=nullptr) {  
 destroy(root->left);  
 destroy(root->right);  
 delete root;  
 }  
}  
  
void print(Node\* root, string indent = "", bool isLeft = false){  
 if(root == nullptr){  
 return;  
 }  
 cout << indent;  
 if(isLeft){  
 cout << "\*-- ";  
 indent += "| ";  
 }  
 else{  
 cout << "`-- ";  
 indent += " ";  
 }  
  
 cout << root->value << endl;  
  
 print(root->left, indent, true);  
 print(root->right, indent, false);  
}  
  
void add(int value, Node\*& root) {  
 if (root == nullptr)  
 root = create(value);  
 else  
 insert(value, root);  
}  
  
void strw(Node\* root) {  
 if(root){  
 std::cout << root->value << ' ';  
 strw(root->left);  
 strw(root->right);  
 }  
}  
  
void revw(Node\* root) {  
 if(root){  
 revw(root->left);  
 revw(root->right);  
 std::cout << root->value << ' ';  
 }  
}  
  
void symw(Node\* root) {  
 if(root){  
 symw(root->left);  
 std::cout << root->value << ' ';  
 symw(root->right);  
 }  
}  
  
void walk(Node\*& root) {  
 int choice;  
 cout <<  
 "Walk:\n"  
 "1. Straight walk\n"  
 "2. Reverse walk\n"  
 "3. Symmetrical walk\n\n";  
 cout << "Type a number to continue: ";  
 cin >> choice;  
 switch(choice) {  
 case 1:  
 strw(root);  
 cout << endl;  
 break;  
 case 2:  
 revw(root);  
 cout << endl;  
 break;  
 case 3:  
 symw(root);  
 cout << endl;  
 break;  
 }  
}  
  
void fill(Node\*& root) {  
 system("cls");  
 cout <<  
 "Fill:\n"  
 "1. Create with random values\n"  
 "2. Create with manual values\n\n";  
 int choice;  
 cout << "Type a number to continue: ";  
 cin >> choice;  
 switch(choice) {  
 case 1: {  
 destroy(root);  
 root = nullptr;  
 int n;  
 cout << "Input value: ";  
 cin >> n;  
 for(int i = 0; i < n; i++){  
 int r = rand() % 200 - 99;  
 add(r, root);  
 }  
 break;  
 }  
 case 2: {  
 destroy(root);  
 root = nullptr;  
 cout << "Input numbers:\n";  
 int val;  
 while (true) {  
 cin >> val;  
 if (cin.fail())  
 break;  
 add(val, root);  
 }  
 cin.clear();  
 cin.ignore(numeric\_limits<streamsize>::max(), '\n');  
 break;  
 }  
 }  
}  
  
void menu() {  
 Node\* bt = nullptr;  
 while (true) {  
 system("cls");  
 cout <<  
 "Choose a category:\n"  
 "1. Create Binary Tree\n"  
 "2. Print Binary Tree\n"  
 "3. Insert element into Binary Tree\n"  
 "4. Delete element from Binary Tree\n"  
 "5. Find element in Binary Tree\n"  
 "6. Binary Tree walk\n\n";  
 int choice;  
 cout << "Type a number to continue: ";  
 cin >> choice;  
 cout << endl;  
 switch (choice) {  
 case 0:  
 return;  
 case 1:  
 fill(bt);  
 break;  
 case 2:  
 print(bt);  
 break;  
 case 3: {  
 int n;  
 cout << "Input a number to insert: ";  
 cin >> n;  
 auto t1 = steady\_clock::now();  
 add(n, bt);  
 auto t2 = steady\_clock::now();  
 auto result = duration\_cast<nanoseconds>(t2 - t1);  
 cout << "Inserted in " << result.count() << " nanoseconds." << endl;  
 break;  
 }  
 case 4: {  
 int n;  
 cout << "Input a number to delete: ";  
 cin >> n;  
 auto t1 = steady\_clock::now();  
 deleteB(n, bt);  
 auto t2 = steady\_clock::now();  
 auto result = duration\_cast<nanoseconds>(t2 - t1);  
 cout << "Deleted in " << result.count() << " nanoseconds." << endl;  
 break;  
 }  
 case 5: {  
 int n;  
 cout << "Input a number to search: ";  
 cin >> n;  
 auto t1 = steady\_clock::now();  
 Node\* node = search(n, bt);  
 auto t2 = steady\_clock::now();  
 auto result = duration\_cast<nanoseconds>(t2 - t1);  
 if (node != nullptr)  
 cout << "Element found: " << node->value << " (" << result.count() << " nanoseconds)" << endl;  
 else  
 cout << "Element not found (" << result.count() << " nanoseconds)" << endl;  
 break;  
 }  
 case 6:  
 walk(bt);  
 break;  
 default:  
 cout << "\nCategory with number " << choice << " does not exist." << endl;  
 break;  
 }  
 system("pause");  
 }  
}  
  
int main()  
{  
 srand(time(nullptr));  
 while (true) {  
 system("cls");  
 cout <<  
 "Choose tree type from below:\n"  
 "0. Exit\n"  
 "1. Binary Search Tree\n\n";  
 int choice;  
 cout << "Type a number to continue: ";  
 cin >> choice;  
 cout << endl;  
 switch (choice) {  
 case 0:  
 return 0;  
 case 1:  
 menu();  
 break;  
 default:  
 cout << "\nCategory with number " << choice << " does not exist." << endl;  
 break;  
 }  
 }  
}

**Вывод.**

Я научился работать с Бинарным деревом поиска и изучил его строение.