**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по курсовой работе**

**по дисциплине «Программирование»**

Тема: АВЛ-дерево.

|  |  |  |
| --- | --- | --- |
| Студент гр. 2372 |  | Братко В.В. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2023

**Цель работы.**

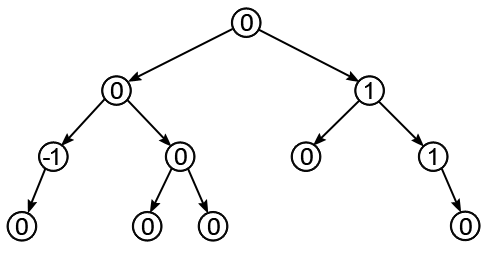
изучение свойств и организации сбалансированных деревьев; получение практических навыков в работе с АВЛ-деревьями; определение преимуществ и недостатков подобных структур данных; проведение сравнительной характеристики скорости вставки, удаления и поиска элемента в АВЛ-деревьях.

**Основные теоретические положения.**

## Понятие АВЛ-дерева

Сбалансированным называется такое двоичное дерево поиска, в котором высота каждого из поддеревьев, имеющих общий корень, отличается не более чем на некоторую константу, и при этом выполняются условия, характерные для двоичного дерева.

АВЛ-дерево – сбалансированное двоичное дерево поиска с константой отличия, равной 1. Для узлов АВЛ-дерева определен коэффициент сбалансированности. Это разность высот правого и левого поддеревьев, принимающая одно значения из множества {–1, 0, 1}. Вместо привычных значений узла отображены соответствующие коэффициенты сбалансированности.



Отличительной особенностью сбалансированного дерева является эффективность в обработке. Действительно, ведь максимальное количество шагов, которое необходимо для обнаружения нужного узла, равно количеству уровней самого бинарного дерева поиска.

Так как поддеревья практически симметричны, высота дерева сводится к оптимальному минимуму. Это позволяет ускорить поиск, что сказывается на общей производительности.

Для АВЛ-деревьев определены следующие основные операции:

1)  добавление узла в дерево;  
2)  поиск по дереву;  
3)  удаление узла;  
4)  балансировка дерева.

## Балансировка АВЛ-дерева

В процессе обработки АВЛ-дерева балансировка может нарушиться, тогда потребуется осуществить операцию балансировки. Если после выполнения операций вставки или удаления коэффициент сбалансированности какого-либо узла АВЛ-дерева становиться равен 2, то необходимо выполнить операцию балансировки.

Операция балансировки осуществляется путем поворота узлов (изменения связей в поддереве). Вращения не меняют свойств бинарного дерева поиска. Различают несколько типов вращения:

1)  малый правый поворот;  
2)  большой правый поворот;  
3)  малый левый поворот;  
4)  большой левый поворот.

Оба типа больших поворотов являются комбинацией малых вращений (право-левым или лево-правым поворотом).

## Добавление и удаление узлов

Операция вставки нового узла в АВЛ-дерево выполняется рекурсивно. По ключу данного узла производится поиск места вставки: спускаясь по дереву вниз, алгоритм сравнивает ключ добавляемого узла со встречающимися ключами, далее происходит вставка нового элемента.

По возвращению из рекурсии выполняется балансировка. Для осуществления балансировки следует знать, с каким из рассмотренных ранее случаев дисбаланса имеется дело.

Именно поэтому процесс вставки узла в АВЛ-дерево отличается от добавления узла в обычное бинарное дерево. Общий алгоритм выглядит следующим образом:

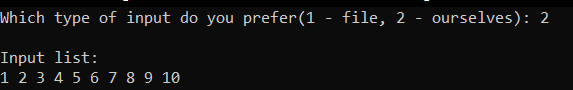
1)  поиск места, где должен будет находиться новый узел;  
2)  добавление узла в дерево на найденную позицию;  
3)  перерасчет коэффициентов сбалансированности для узлов, находящихся выше добавленного узла;  
4)  если дерево стало несбалансированным, то необходимо выполнить балансировку.

Удаление узла из АВЛ-дерева сложнее аналогичной операции для простого двоичного дерева и включает в себя следующие этапы:

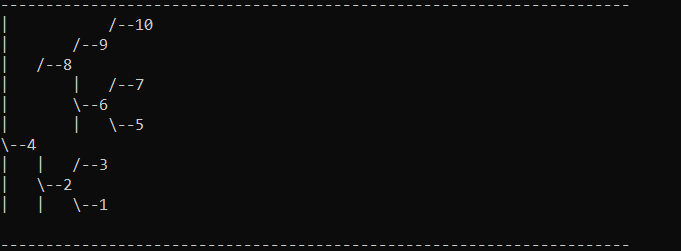
1)  поиск узла, который требуется удалить (в процессе поиска запоминаются пройденные узлы для выполнения последующей балансировки);  
2)  удаление искомого узла и обновление списка пройденных узлов;  
3)  выполнение балансировки и перерасчет коэффициентов сбалансированности.

**Выполнение работы.**

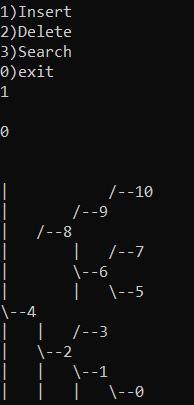
1. Выбор типа заполнения и заполнения дерева:



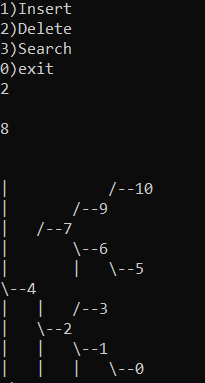
1. Формирование дерева:



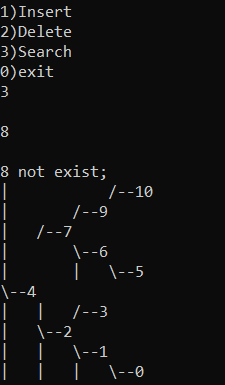
1. Вставка нового элемента:



1. Удаление элемента и дальнейшая балансировка:



1. Поиск элемента:



6.Проверка на ошибки



**Полный код программы:**

#include <iostream>  
#include <sstream>  
#include <stack>  
#include <fstream>  
#include <chrono>  
  
using namespace std;  
  
int checkInput(){  
 int input;  
 try {  
 cin >> input;  
 if (cin.fail()) {  
 throw 1;  
 }  
 } catch (int exeption) {  
 cout << "ERROR!!!";  
 exit(0);  
 }  
 cin.sync();  
 cout << "\n";  
 return input;  
}  
  
struct Node{  
 int key;  
 int height = 1;  
 Node \*left = nullptr;  
 Node \*right = nullptr;  
};  
  
int height(Node\* node) {  
 if (node != nullptr) {  
 return node->height;  
 } else {  
 return 0;  
 }  
}  
  
int balanceFactor(Node\* node) {  
 if (node == nullptr) {  
 return 0;  
 } else {  
 return (height(node->right) - height(node->left));  
 }  
}  
  
void fixHeight(Node\* node) {  
 int left = height(node->left);  
 int right = height(node->right);  
 if (left > right) {  
 node->height = left + 1;  
 } else {  
 node->height = right + 1;  
 }  
}  
  
Node\* rotateLeft(Node\* node) {  
 Node\* p = node->right;  
 node->right = p->left;  
 p->left = node;  
 fixHeight(node);  
 fixHeight(p);  
 return p;  
}  
  
Node\* rotateRight(Node\* node) {  
 Node\* p = node->left;  
 node->left = p->right;  
 p->right = node;  
 fixHeight(node);  
 fixHeight(p);  
 return p;  
}  
  
Node\* balance(Node\* root) {  
 fixHeight(root);  
 if (balanceFactor(root) == 2) {  
 if (balanceFactor(root->right) < 0) {  
 root->right = rotateRight(root->right);  
 }  
 return rotateLeft(root);  
 }  
 if (balanceFactor(root) == -2) {  
 if (balanceFactor(root->left) > 0) {  
 root->left = rotateLeft(root->left);  
 }  
 return rotateRight(root);  
 }  
 return root;  
}  
Node\* insert(Node\* node, int k) {  
 if (!node) {  
 Node\* newNode = new Node;  
 newNode->key = k;  
 return newNode;  
 }  
 if (k < node->key) {  
 node->left = insert(node->left, k);  
 } else if (k > node->key) {  
 node->right = insert(node->right, k);  
 }  
 return balance(node);  
}  
  
Node\* search(Node \*node, int k, stack<Node\*> &stack){  
 if(!node){  
 return nullptr;  
 }else{  
 if(k == node->key){  
 return node;  
 }  
 stack.push(node);  
 if(k > node->key){  
 return search(node->right, k, stack);  
 }else{  
 return search(node->left, k, stack);  
 }  
 }  
}  
  
Node\* search(Node \*node, int k){  
 if(!node){  
 return nullptr;  
 }else{  
 if(k == node->key){  
 return node;  
 }  
 if(k > node->key){  
 return search(node->right, k);  
 }else{  
 return search(node->left, k);  
 }  
 }  
}  
  
Node\* getMax(Node \*node, stack<Node\*> &stack){  
 Node \*prev = node;  
 while (node->right) {  
 stack.push(node);  
 prev = node;  
 node = node->right;  
 }  
 //надо написать, чтобы максимальный элемент удалялся и у его родителя->right = null  
 if(node->left && node != prev){  
 prev->right = node->left;  
 }else{  
 prev->right = nullptr;  
 }  
 return node;  
}  
  
Node\* deleteNode(Node \*node, int k){  
 stack<Node\*> stack;  
 Node \*deleteElement = search(node, k, stack);  
 if(deleteElement){  
 if(deleteElement->left && deleteElement->right){  
 stack.push(deleteElement);  
 Node \*max = getMax(deleteElement->left, stack);  
 if(deleteElement->left == max){  
 deleteElement->left = max->left;  
 }  
 deleteElement->key = max->key;  
 delete max;  
 }else if(deleteElement->left){  
 stack.top()->left = deleteElement->left;  
 delete deleteElement;  
 }else if(deleteElement->right){  
 stack.top()->right = deleteElement->right;  
 delete deleteElement;  
 }else{  
 if(stack.top()->left == deleteElement){  
 stack.top()->left = nullptr;  
 delete deleteElement;  
 }else {  
 stack.top()->right = nullptr;  
 delete deleteElement;  
 }  
 }  
 }else {  
 return node;  
 }  
 while (!stack.empty()){  
 Node \*help = stack.top();  
 if(stack.size() == 1){  
 node = balance(stack.top());  
 break;  
 }  
 stack.pop();  
 stack.top()->left == help ? stack.top()->left = balance(help) : stack.top()->right = balance(help);  
 }  
 return node;  
}  
  
void printTree(Node\* root, ostream& outputFile, std::string prefix = "", bool isLeft = true) {  
 if (root == nullptr) {  
 return;  
 }  
 printTree(root->right, outputFile, prefix + (isLeft ? "| " : " "), false);  
 outputFile << prefix;  
 outputFile << (isLeft ? "\\--" : "/--");  
 outputFile << root->key << endl;  
 printTree(root->left,outputFile, prefix + (isLeft ? "| " : " "), true);  
}  
  
void action(Node \*&root, string str, int type, ofstream& output\_key, ofstream& output\_ans){  
 istringstream iss(str); int k;  
 while (iss >> k) {  
 switch (type) {  
 case 0:  
 case 1:  
 root = insert(root, k);  
 break;  
 case 2:  
 if (!search(root, k)) output\_key << k << " is not exist!\n";  
 else{  
 root = deleteNode(root, k);  
 }  
 break;  
 case 3:  
 if (search(root, k)) output\_ans << k << " exist; ";  
 else output\_ans << k << " not exist; ";  
 break;  
 default:  
 output\_key << "Error input!";  
 }  
 output\_key << type <<":\n";  
 printTree(root, output\_key);  
 }  
 output\_ans << "\n";  
 switch (type) {  
 case 0:  
 output\_ans << "Tree after creation:\n";  
 break;  
 case 1:  
 output\_ans << "Tree after add elements:\n";  
 break;  
 case 2:  
 output\_ans << "Tree after delete elements:\n";  
 break;  
 case 3:  
 output\_ans << "Final result:\n";  
 break;  
 }  
 printTree(root, output\_ans);  
}  
  
void consoleMod(Node \*&root){  
 cout << "Input list: \n";  
 string stringList;  
 getline(cin, stringList);  
 istringstream iss(stringList);  
 int num;  
 while (iss >> num) {  
 root = insert(root, num);  
 printTree(root, cout);  
 cout << "\n" << "----------------------------------------------------------------------" << "\n";  
 }  
 int type = 1;  
 while(type) {  
 cout << "1)Insert\n"  
 "2)Delete\n"  
 "3)Search\n"  
 "0)exit\n";  
 type = checkInput();  
 if(type == 0) break;  
 int k = checkInput();  
 switch (type){  
 case 1:  
 root = insert(root, k);  
 break;  
 case 2:  
 if (!search(root, k)) cout << k << " is not exist!\n";  
 else{  
 root = deleteNode(root, k);  
 }  
 break;  
 case 3:  
 if(search(root, k)) cout << k << " exist;";  
 else cout << k << " not exist;";  
 break;  
 default:  
 cout << "Error input!";  
 }  
 cout << "\n";  
 printTree(root, cout);  
 }  
}  
  
void fileMod(Node \*&root){  
 ifstream input\_task("C:\\FirstCursProgramm\\C++\\coursework2\\input\_task.txt");  
 ofstream output\_key("C:\\FirstCursProgramm\\C++\\coursework2\\output\_key.txt");  
 ofstream output\_ans("C:\\FirstCursProgramm\\C++\\coursework2\\output\_ans.txt");  
 if (input\_task.is\_open() && output\_ans.is\_open() && output\_key.is\_open()) {  
 string line, creatLine, deleteLine, addLine, searchLine;  
 for(int i = 0; i < 5; i++) getline(input\_task, line);  
 while (getline(input\_task, line)) {  
 getline(input\_task, creatLine);  
 getline(input\_task, deleteLine);  
 getline(input\_task, addLine);  
 getline(input\_task, searchLine);  
 auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 action(root, creatLine, 0, output\_key, output\_ans);  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 output\_key << "time spent to creat tree in nanoseconds: " << end - start << "ns\n";  
 start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 action(root, deleteLine, 2, output\_key, output\_ans);  
 end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 output\_key << "time spent to delete elements tree in nanoseconds: " << end - start << "ns\n";  
 start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 action(root, addLine, 1, output\_key, output\_ans);  
 end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 output\_key << "time spent to add elements tree in nanoseconds: " << end - start << "ns\n";  
 start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 action(root, searchLine, 3, output\_key, output\_ans);  
 end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 output\_key << "time spent to search elements tree in nanoseconds: " << end - start << "ns\n";  
 output\_ans << "----------------------------";  
 output\_key << "----------------------------";  
 root = nullptr;  
 }  
 input\_task.close();  
 output\_ans.close();  
 output\_key.close();  
 } else {  
 std::cout << "Could not open the file!!!" << std::endl;  
 }  
}  
  
int main() {  
 Node \*root = nullptr;  
 cout << "Which type of input do you prefer(1 - file, 2 - ourselves): ";  
 int inputType = checkInput();  
 if(inputType == 1){  
 fileMod(root);  
 }else if(inputType == 2){  
 consoleMod(root);  
 }else{  
 cout << "Wrong input!\n";  
 exit(1);  
 }  
 getchar();  
 return 0;  
}

**Вывод:**

Проведя всю работу, я могу сказать, что АВЛ-дерево лучше всего применять для данных, в которых необходим постоянный поиск, т.к. балансировка гарантирует в худшем случае сложность алгоритма поиска O(log(n)). Добавление и удаление элементов тоже делается недолго, но из-за постоянной балансировки бывает случаи, когда такая частая балансировка немного избыточна.