**Восьмая Задача**

**Оглавление**

1. Глава 1
   1. Условия
   2. Теорема
2. Глава 2
   1. Реализация
3. Глава 3
   1. Список Литературы

**Глава 1**

1. Условия: Моделирование Помощника
   1. Реализовать набор классов для splay-дерева
   2. Реализовать операции создаяния дерева из множества элементов, операций поворотов, добакления и удаления узла, слияния деревьев.
   3. Перегрузить операторы ввода/вывода в поток >>, << так, чтобы можно было вводить выводить информацию о дереве и из файла, и из консоли.
   4. Произвести обзор других алгоритмов и структур данных для решения задач, которые эффективно обрабатываются с помощью splay-дерева.
2. Теорема
   1. Вступление: Структуры данных и алгоритмы являются фундаментальными понятиями в информатике, которые создают основу для разработки эффективного программного обеспечения. В этом отчете мы сосредоточимся на двух самонастраивающихся деревьях бинарного поиска - AVL-деревьях и Splay-деревьях.
   2. Splay-дерева: Основным теоретическим обоснованием эффективности расщепленных деревьев является "Теорема расщепленного дерева" или "Теорема баланса". Эта теорема утверждает, что общая стоимость "m" операций с разветвленным деревом пропорциональна "m", умноженному на логарифм числа узлов в дереве. Это означает, что каждая операция, в среднем, занимает O(log n) времени.
   3. AVL дерева: Деревья AVL, названные в честь их изобретателей Адельсона-Вельского и Лэндиса, являются одними из самых ранних самобалансирующихся деревьев бинарного поиска. В дереве AVL высоты двух дочерних поддеревьев любого узла отличаются не более чем на единицу. Если в какой-либо момент они отличаются более чем на единицу, выполняется перебалансировка для восстановления свойства. Деревья AVL гарантируют, что дерево остается сбалансированным, что приводит к O (log n) высотам дерева и гарантирует, что все основные операции (вставка, удаление, поиск) могут быть выполнены за O (log n) времени.
   4. В контексте древовидной структуры данных, такой как AVL-дерево или Splay-дерево, "node" является фундаментальной частью структуры, которая содержит ключевые данные и указатели на другие узлы. Каждый узел содержит фрагмент данных (значение "key"). Эти данные используются для упорядочения узлов в дереве. В двоичном дереве, таком как AVL-дерево или Splay-дерево, каждый node содержит указатели на два других nodes: "left" дочерний и "right" дочерний. Эти указатели создают древовидную структуру.

**Глава 2**

**Реализация**

1. Класс Node:
   1. Исходный код “Node”(и операторы):

template<class T>

class Node {

public:

T key;

Node\* left;

Node\* right;

Node<T>(T key): key(key), left(NULL), right(NULL){}

Node<T>() {

key = 0;

left = right = NULL;

}

~Node<T>() {

left = right =nullptr;

}

template<class U>

friend ostream& operator << (ostream& out, const Node<U>& node);

template<class U>

friend istream& operator >> (istream& in, Node<U>& node);

};

template<class T>

ostream& operator<<(ostream& out, const Node<T>& node) {

out << "Key: " << node.key;

if (node.left != nullptr)

out << " | Left child: " << node.left->key;

if (node.right != nullptr)

out << " | Right child: " << node.right->key;

return out;

}

template<class T>

istream& operator>>(istream& in, Node<T>& node) {

in >> node.key;

return in;

}

* 1. Члены Класса “Node”:
     1. “T key” Это данные, хранящиеся в узле. "T" - это заполнитель для типа, который может содержать данные любого типа (int, string и т.д.). "Key"обычно используется для организации nodes в дереве.
     2. “Node\* left;”, “Node\* right;” Это указатели на левый и правый дочерние узлы текущего узла соответственно. В двоичном дереве каждый node может иметь до двух дочерних элементов. Если дочерний node не существует (например, у leaf node нет дочерних элементов), эти pointers будут иметь значение nullptr.
  2. Методы класса “Node”:
     1. “Node<T>(T key)”, “Node<T>()” и “~Node<T>()”: Они конструктор и деструктор. В конструкторе “Node<T>(T key)”, он принимает аргумент "T key", который является key для вновь созданного node. Этот ключ присваивается "key" node, а его левый и правый дочерние элементы инициализируются как " NULL ". И в конструкторе “Node<T>()”, Он устанавливает аргумент "key" равным 0, а правому и левому дочерним элементам присваивает значение "NULL".
     2. “friend ostream& operator <<” и “friend istream& operator >>”, Это перегрузки операторов вставки потока (<<) и извлечения (>>) соответственно.

template<class T>

ostream& operator<<(ostream& out, const Node<T>& node) {

out << "Key: " << node.key;

if (node.left != nullptr)

out << " | Left child: " << node.left->key;

if (node.right != nullptr)

out << " | Right child: " << node.right->key;

return out;

}

template<class T>

istream& operator>>(istream& in, Node<T>& node) {

in >> node.key;

return in;

}

Функция “friend ostream& operator <<” выводит ключ "Node" в объект "stream", который может быть выводом консоли или файлом.

Функция “friend istream& operator >>” позволяет вам напрямую вводить значение в объект "Node" из потока. Он используется, когда вы хотите принять пользовательский ввод или прочитать из файла непосредственно в объект "Node".

1. Класс “SplayTree”:
   1. Исходный код класса “SplayTree”(и операторы):

template<class T>

class SplayTree {

protected:

void savetofilehelper(Node<T>\* node, ofstream& file) {

if (node) {

savetofilehelper(node->left, file);

file << node->key << endl;

savetofilehelper(node->right, file);

}

}

void deleteTree(Node<T>\* node) {

if (node == nullptr) {

return;

}

deleteTree(node->left);

deleteTree(node->right);

delete node;

}

public:

Node<T>\* root;

SplayTree(): root(nullptr) {}

~SplayTree() {

deleteTree(root);

}

Node<T>\* rightrotate(Node<T>\* node) {

if (node == nullptr) {

throw Exception("cannot rotate a null node");

}

Node<T>\* newNode = node->left;

if (newNode == nullptr) {

throw Exception("Cannot rotate a node, because it has no leftchild");

}

else {

node->left = newNode->right;

newNode->right = node;

return newNode;

}

}

Node<T>\* leftrotate(Node<T>\* node) {

if (node == nullptr) {

throw Exception("Cannot rotate a null node");

}

Node<T>\* newNode = node->right;

if (newNode == nullptr) {

throw Exception("Cannot rotate a node, because it has no right child");

}

else {

node->right = newNode->left;

newNode->left = node;

return newNode;

}

}

Node<T>\* splay(T key, Node<T>\* root) {

if (!root) return NULL;

Node<T> header;

header.left = header.right = NULL;

Node<T>\* LeftTreeMax = &header;

Node<T>\* RightTreeMin = &header;

while (1) {

if (key < root->key) {

if (!root->left) {

break;

}

if (key < root->left->key) {

root = rightrotate(root);

if (!root->left) break;

}

RightTreeMin->left = root;

RightTreeMin = RightTreeMin->left;

root = root->left;

RightTreeMin->left = NULL;

}

else if (key > root->key) {

if (!root->right) break;

if (key > root->right->key) {

root = leftrotate(root);

if (!root->right) break;

}

LeftTreeMax->right = root;

LeftTreeMax = LeftTreeMax->right;

root = root->right;

LeftTreeMax->right = NULL;

}

else break;

}

LeftTreeMax->right = root->left;

RightTreeMin->left = root->right;

root->left = header.right;

root->right = header.left;

return root;

}

Node<T>\* New\_Node(T key) {

Node<T>\* p\_node = new Node<T>;

if (!p\_node) {

throw Exception("Out of Memory");

}

p\_node->key = key;

p\_node->left = p\_node->right = NULL;

return p\_node;

}

Node<T>\* Insert(T key, Node<T>\* root) {

static Node<T>\* p\_node = NULL;

if (!p\_node) p\_node = New\_Node(key);

else p\_node->key = key;

if (!root) {

root = p\_node;

p\_node = NULL;

return root;

}

root = splay(key, root);

if (key < root->key) {

p\_node->left = root->left;

p\_node->right = root;

root->left = NULL;

root = p\_node;

}

else if (key > root->key) {

p\_node->right = root->right;

p\_node->left = root;

root->right = NULL;

root = p\_node;

}

else return root;

p\_node = NULL;

return root;

}

Node<T>\* Delete(T key, Node<T>\* root) {

Node<T>\* temp;

if (!root) return NULL;

root = splay(key, root);

if (key != root->key) return root;

else {

if (!root->left) {

temp = root;

root = root->right;

}

else {

temp = root;

root = splay(key, root->left);

root->right = temp->right;

}

delete temp;

return root;

}

}

Node<T>\* search(T key, Node<T>\* root) {

return splay(key, root);

}

void InOrder(Node<T>\* root) {

if (root) {

InOrder(root->left);

cout << "key: " << root->key;

if (root->left) cout << " | left child: " << root->left->key;

if (root->right) cout << " | right child: " << root->right->key;

cout << "\n";

InOrder(root->right);

}

}

Node<T>\* merge(Node<T>\* t1, Node<T>\* t2) {

if (t1 == nullptr) return t2;

if (t2 == nullptr) return t1;

t1->key += t2->key;

t1->left = merge(t1->left, t2->left);

t2->right = merge(t1->right, t2->right);

return t1;

}

void save(const char\* c) {

ofstream infile(c);

if (!infile.is\_open()) {

throw Exception("Failed to open file");

}

savetofilehelper(root, infile);

infile.close();

}

Node<T>& load(ifstream& in) {

if (in) {

T key;

while (in >> key) {

root = Insert(key, root);

}

}

return \*this;

}

template<class U>

friend ostream& operator << (ostream& out, const SplayTree<U>& st);

template<class U>

friend istream& operator >> (istream& in, SplayTree<U>& st);

};

template<class T>

void writeKeysInOrder(Node<T>\* node, ostream& out) {

if (node) {

writeKeysInOrder(node->left, out);

out << node->key;

if (node->left) out << " | left child: " << node->left->key;

if (node->right) out << " | right child: " << node->right->key;

out << endl;

writeKeysInOrder(node->right, out);

}

}

template<class T>

ostream& operator<<(ostream& out, const SplayTree<T>& st) {

writeKeysInOrder(st.root, out);

return out;

}

template<class T>

istream& operator >> (istream& in, SplayTree<T>& st) {

T key;

while (in >> key) {

st.root = st.Insert(key, st.root);

}

return in;

}

* 1. Члены класса “SplayTree”:
     1. Node<T>\* root – Начальный Node
  2. Методы класса “SplayTree”:
     1. “SplayTree(): root(nullptr)” и “~SplayTree()”: Конструктор и Деструктор.
     2. “void savetofilehelper(Node<T>\* node, ofstream& file)”: Это вспомогательная функция, используемая для сохранения "keys" "SplayTree" в файл в порядке обхода. Если текущий "Node" существует, он рекурсивно сохраняет левое поддерево, записывает ключ текущего "Node" в файл, затем рекурсивно сохраняет правое поддерево.
     3. “void deleteTree(Node<T>\* node)”: Эта функция используется для удаления всех "Nodes" в дереве и освобождения памяти.
     4. “Node<T>\* rightrotate(Node<T>\* node)”: Этот метод выполняет поворот вправо для данного "Node". Он используется для балансировки дерева во время операций "splay". Он генерирует "Exception", если "Node" или его левый дочерний элемент равен "nullptr".
     5. “Node<T>\* leftrotate(Node<T>\* node)”: Этот метод выполняет поворот влево для данного "Node".
     6. “Node<T>\* splay(T key, Node<T>\* root)”: Эта функция перемещает "node" с заданным "key" в "root" дерева, используя серию операций поворота. Если "key" не существует в дереве, последний доступный "node" будет перемещен в "root". Эта операция оптимизирует дерево для последовательных обращений к одному и тому же "key".
     7. “Node<T>\* New\_Node(T key)”: Эта функция создает новый "Node" с заданным "Key" и возвращает указатель на него.
     8. “Node<T>\* Insert(T key, Node<T>\* root)”: Эта функция вставляет в дерево новый "node" с заданным "key". Он использует функцию "splay", чтобы гарантировать, что вновь вставленный "node" будет перемещен в "root" дерева.
     9. “Node<T>\* Delete(T key, Node<T>\* root)”: Эта функция удаляет "node "с заданным "key" из дерева. Сначала он воспроизводит дерево в удаляемом "node", а затем удаляет "root node". Если "key" не существует в дереве, функция ничего не делает.
     10. “Node<T>\* search(T key, Node<T>\* root)”: Эта функция выполняет поиск "node" с заданным "key" в дереве. Он возвращает "root" дерева после операции "splay". Если "key" существует, возвращаемый "root" - это "node" с "key". Если этого не происходит, возвращаемый "root" является последним доступным "node".
     11. “void InOrder(Node<T>\* root)”: Эта функция печатает "keys" дерева в порядке обхода по порядку.
     12. “Node<T>\* merge(Node<T>\* t1, Node<T>\* t2)”: Этот метод объединяет два дерева t1 и t2, предполагая, что t1 и t2 являются максимальными кучами.
     13. “void save(const char\* c)”: Эта функция сохраняет "keys" дерева в файл. Он открывает файл, записывает "keys" с помощью "savetofilehelper", затем закрывает файл.
     14. “Node<T>& load(ifstream& in)”: Эта функция загружает "keys" из файла в дерево. Он считывает "keys" из файла и вставляет их в дерево. Он возвращает ссылку на дерево.
     15. “friend ostream& operator << (ostream& out, const SplayTree<U>& st)” и “friend istream& operator >> (istream& in, SplayTree<U>& st)”: Они являются функцией friend в классе, поэтому они могут получить защищенный доступ. "operator <<" позволяет напечатать объект "SplayTree". "operator >>" позволяет загружать "keys" в "SplayTree".
     16. “void writeKeysInOrder(Node<T>\* node, ostream& out)”: Эта функция выводит "keys" "node" и его дочерних элементов в порядке обхода в output stream.

1. Класс “AVLTree” и “AVLNode”:
   1. Исходный код класса “AVLNode”:

template<class T>

class AVLNode : public Node<T> {

public:

int height;

AVLNode(T key): Node<T>(key), height(1) {}

AVLNode<T>() {

key = 0;

left = right = nullptr;

height = 0;

}

template<class U>

friend ostream& operator << (ostream& out, const AVLNode<U>& avlnode);

template<class U>

friend istream& operator >> (istream& in, AVLNode<U>& avlnode);

}

* 1. Члены класса “AVLNode”:
     1. “int height”: Он представляет собой длину самого длинного пути от "node" к "leaf". Он используется для определения "баланса" каждого "node", то есть того, отличаются ли левое и правое поддеревья каждого "node" по высоте не более чем на единицу.
  2. Методы классы “AVLNode”:
     1. “AVLNode(T key): Node<T>(key), height(1)” и “~AVLNode<T>()” – Конструктор и Деструктор.
     2. “friend ostream& operator << (ostream& out, const AVLNode<U>& avlnode)” и “friend istream& operator >> (istream& in, AVLNode<U>& avlnode)”: Это дружественные функции, которые перегружают операторы вставки и извлечения для класса "AVLNode"
  3. Исходный код класса “AVLTree”:

template<class T>

class AVLTree {

protected:

void deleteTree(AVLNode<T>\* avlnode) {

if (avlnode == nullptr) {

return;

}

deleteTree(avlnode->left);

deleteTree(avlnode->right);

delete avlnode;

}

void savetofilehelper(AVLNode<T>\* avlnode, ofstream& file) {

if (avlnode) {

savetofilehelper(avlnode->left, file);

file << avlnode->key << endl;

savetofilehelper(avlnode->right, file);

}

}

template<class U>

void writeKeysInOrder(AVLNode<U>\* avlnode, ostream& out) {

if (avlnode) {

writeKeysInOrder(avlnode->left, out);

out << avlnode->key;

if (avlnode->left) out << " | left child: " << avlnode->left->key;

if (avlnode->right) out << " | right child: " << avlnode->right->key;

out << endl;

writeKeysInOrder(avlnode->right, out);

}

}

public:

AVLNode<T>\* root;

AVLTree() : root(nullptr) {}

~AVLTree() {

deleteTree(root);

}

int height(AVLNode<T>\* avlnode) {

if (avlnode == nullptr) return 0;

return avlnode->height;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

AVLNode<T>\* newNode(T key) {

AVLNode<T>\* node = new AVLNode<T>();

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return node;

}

AVLNode<T>\* rightRotate(AVLNode<T>\* y) {

if (y == nullptr) throw Exception("Cannot rotate a null node\n");

AVLNode<T>\* x = y->left;

if (x == nullptr) throw Exception("Cannot rotate a node, because it has no left child\n");

AVLNode<T>\* t2 = x->right;

x->right = y;

y->left = t2;

y->height = max(height(y->left), height(y->right) + 1);

x->height = max(height(x->left), height(x->right) + 1);

return x;

}

AVLNode<T>\* leftRotate(AVLNode<T>\* x) {

if (x == nullptr) {

throw Exception("Cannot rotate a null node");

}

AVLNode<T>\* y = x->right;

if (y == nullptr) {

throw Exception("Cannot rotate a node, because it has no right child");

}

AVLNode<T>\* t2 = y->left;

y->left = x;

x->right = t2;

x->height = max(height(x->left), height(x->right) + 1);

y->height = max(height(y->left), height(y->right) + 1);

return y;

}

int getBalance(AVLNode<T>\* avlnode) {

if (avlnode == NULL) return 0;

return height(avlnode->left) - height(avlnode->right);

}

AVLNode<T>\* search(T key, AVLNode<T>\* root) {

if (root == nullptr || root->key == key)

return root;

if (root->key < key)

return search(key, root->right);

return search(key, root->left);

}

AVLNode<T>\* insert(AVLNode<T>\* avlnode, int key) {

if (avlnode == NULL) return(newNode(key));

if (key < avlnode->key) avlnode->left = insert(avlnode->left, key);

else if (key > avlnode->key) avlnode->right = insert(avlnode->right, key);

else return avlnode;

avlnode->height = 1 + max(height(avlnode->left), height(avlnode->right));

int balance = getBalance(avlnode);

if (balance > 1 && key > avlnode->left->key) {

avlnode->left = leftRotate(avlnode->left);

return rightRotate(avlnode);

}

if (balance < -1 && key < avlnode->right->key) {

avlnode->right = rightRotate(avlnode->right);

return leftRotate(avlnode);

}

return avlnode;

}

AVLNode<T>\* minValueAVLN(AVLNode<T>\* avlnode) {

AVLNode<T>\* current = avlnode;

while (current->left != nullptr) current = current->left;

return current;

}

AVLNode<T>\* deletenode(AVLNode<T>\* root, int key) {

if (root == NULL) return root;

if (key < root->key) root->left = deletenode(root->left, key);

else if (key > root->key) root->right = deletenode(root->right, key);

else {

if ((root->left == NULL) || (root->right == NULL)) {

AVLNode<T>\* temp = root->left ?

root->left :

root->right;

if (temp == NULL) {

temp = root;

root = NULL;

}

else \*root = \*temp;

delete temp;

}

else {

AVLNode<T>\* temp = minValueAVLN(root->right);

root->key = temp->key;

root->right = deletenode(root->right, temp->key);

}

}

if (root == NULL) return root;

root->height = 1 + max(height(root->left), height(root->right));

int balance = getBalance(root);

if (balance > 1 && getBalance(root->left) >= 0) return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

if (balance < -1 && getBalance(root->right) > 0) {

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

void preOrder(AVLNode<T>\* root) {

if (root != NULL) {

cout << root->key << " ";

preOrder(root->left);

preOrder(root->right);

}

}

void save(const char\* c) {

ofstream outfile(c);

if (!outfile.is\_open()) {

throw Exception("Failed to open file");

}

savetofilehelper(root, outfile);

outfile.close();

}

AVLNode<T>\* load(ifstream& in) {

if (!in.is\_open()) {

throw Exception("Could not open file");

}

int key;

while (in >> key) {

root = insert(root, key);

}

return root;

}

template<class U>

friend ostream& operator << (ostream& out, AVLTree<U>& avltree);

template<class U>

friend istream& operator >> (istream& in, AVLTree<U>& avltree);

};

* 1. Члены Класса “AVLTree”:
     1. “AVLNode<T>\* root” Начальный “node”
  2. Методы класса “SplayTree”:
     1. “AVLTree(): root(nullptr)” и “~AVLTree()”: Конструктор и Деструктор.
     2. “void savetofilehelper(AVLNode<T>\* node, ofstream& file)”: Это вспомогательная функция, используемая для сохранения "keys" "AVLTree" в файл в порядке обхода.
     3. “void deleteTree(AVLNode<T>\* node)”: Эта функция используется для удаления всех "Nodes" в дереве и освобождения памяти.
     4. “void writeKeysInOrder(AVLNode<U>\* avlnode, ostream& out)”: Эта функция записывает ключи дерева AVL в outputstream способом обхода по порядку.
     5. “AVLNode<T>\* rightRotate(AVLNode<T>\* y)” и “AVLNode<T>\* leftRotate(AVLNode<T>\* x)”: Эти функции выполняют правый и левый повороты на заданных "nodes" соответственно. Эти вращения используются для балансировки дерева.
     6. “int height(AVLNode<T>\* avlnode)”: Эта функция возвращает высоту данного "node".
     7. “int max(int a, int b)”: Простая функция для получения максимального значения из двух целых чисел.
     8. “AVLNode<T>\* newNode(T key)”: Эта функция создает новый "AVLNode" с заданным "Key".
     9. “int getBalance(AVLNode<T>\* avlnode)”: Эта функция возвращает коэффициент баланса "node", который представляет собой разницу в высоте между его левым и правым дочерними элементами.
     10. “AVLNode<T>\* insert(AVLNode<T>\* avlnode, int key)”: Эта функция вставляет новый "node" с заданным "key" в дерево "AVL", сохраняя баланс дерева.
     11. “AVLNode<T>\* minValueAVLN(AVLNode<T>\* avlnode)”: Эта функция возвращает "node" с минимальным "key" в поддереве, имеющем "root" в данном "node".
     12. “AVLNode<T>\* deletenode(AVLNode<T>\* root, int key)”: Эта функция удаляет "node" с заданным "key" из дерева "AVL", сохраняя баланс дерева.
     13. “AVLNode<T>\* search(T key, AVLNode<T>\* root)”: Эта функция выполняет поиск "key" в дереве "AVL".
     14. “void preOrder(AVLNode<T>\* root)”: Эта функция печатает "keys" дерева в порядке обхода по порядку.
     15. “void save(const char\* c)”: Эта функция сохраняет "keys" дерева AVL в файл.
     16. “AVLNode<T>& load(ifstream& in)”: Эта функция загружает "keys" из файла в дерево AVL.
     17. “friend ostream& operator << (ostream& out, const AVLTree<U>& st)” и “friend istream& operator >> (istream& in, AVLTree<U>& st)”: Это дружественные функции, которые перегружают операторы вставки и извлечения для класса "AVLTree", обеспечивая удобный способ печати и загрузки "AVLTree" соответственно.

1. “int main()” функция:
   1. Исходный код:

int main() {

AVLTree<int> avl;

ifstream fin("test.txt");

if (fin) {

fin >> avl;

fin.close();

}

else {

cout << "Couldn't open the file. \n";

return 1;

}

try {

avl.root = avl.rightRotate(avl.root);

}

catch (const Exception& e) {

cout << "Exception was caught: " << endl;

e.print();

}

avl.preOrder(avl.root);

try {

avl.load(fin);

}

catch (const Exception& e) {

e.print();

}

ofstream fout("result.txt");

if (fout) {

fout << avl;

fout.close();

}

AVLNode<int>\* result = avl.search(20, avl.root);

if (result != nullptr) cout << "Found\n";

else cout << "Not Found\n";

SplayTree<int> st;

ifstream fin("test.txt");

if (fin) {

fin >> st;

fin.close();

}

else {

cout << "Couldn't open the file";

return 1;

}

st.root = st.Insert(10, st.root);

st.root = st.Insert(20, st.root);

st.root = st.Insert(30, st.root);

try {

st.root = st.rightrotate(st.root);

}

catch (const Exception &e) {

cout << "Exception was caught: " << endl;

e.print();

}

st.InOrder(st.root);

try {

st.root = st.leftrotate(st.root);

st.root = st.leftrotate(st.root);

}

catch (const Exception& e) {

e.print();

}

cout << "The key 90 is deleted\n";

st.Delete(90, st.root);

st.InOrder(st.root);

return 0;

}

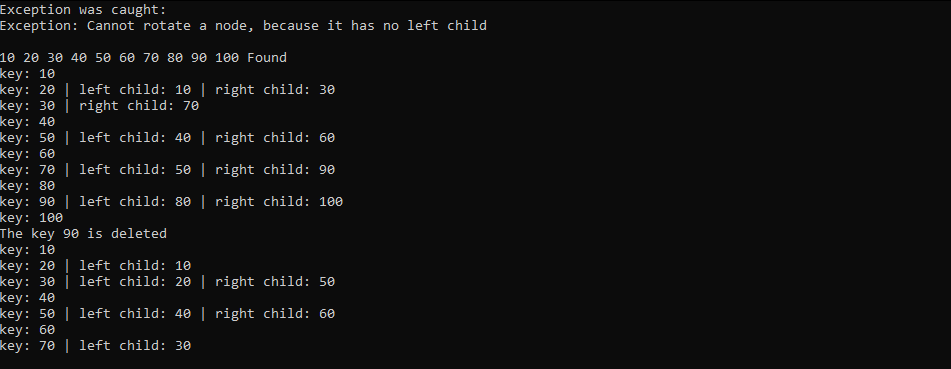
* 1. Объяснение:

Программа запускается с создания дерева AVL и дерева Splay, оба для целых чисел.

Для дерева AVL он считывает данные из "test.txt ", выполняет правый поворот на крыше и загружает больше данных. Затем он записывает данные в "result.txt " и ищет число 20.

Для дерева Splay он также считывает данные из "test.txt ", вставляет три целых числа (10, 20, 30), выполняет операции вращения, а затем пытается удалить число 90.

Обе операции с деревом включают обработку исключений для поворотов. Данные деревьев выводятся на консоль до и после поворотов. Если какие-либо файловые операции завершаются неудачей, отображаются соответствующие сообщения об ошибках, и программа завершается с кодом возврата, равным 1. При успешном выполнении программа завершается с кодом возврата, равным 0.

* 1. Пример того, как работает программа:

**Глава 3**

**Список Литературы**

1. [Splay-Дерево](https://neerc.ifmo.ru/wiki/index.php?title=Splay-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE)
2. [SplayTree in Python](https://habr.com/ru/companies/JetBrains-education/articles/210296/)
3. [AVLTree](https://www.programiz.com/dsa/avl-tree)