Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету "Дискретный анализ" №2

Студент: Шипилов К. Ю.

Преподаватель: Макаров Н. К.

Группа: М8О-203Б-22

Дата: 18.05.2024

Оценка:

Подпись:

Москва 2024 г.

Оглавление

Цель работы

Приобретение практических навыков в реализации и использовании сбалансированных деревьев.

Постановка задачи

Необходимо создать программную библиотеку, реализующую указанную структуру данных, на основе которой разработать программу-словарь. В словаре каждому ключу, представляющему из себя регистронезависимую последовательность букв английского алфавита длиной не более 256 символов, поставлен в соответствие некоторый номер, от 0 до 2^{64} - 1. Разным словам может быть поставлен в соответствие один и тот же номер.

Программа должна обрабатывать строки входного файла до его окончания. Каждая строка может иметь следующий формат:

+ word 34 — добавить слово «word» с номером 34 в словарь. Программа должна вывести строку «OK», если операция прошла успешно, «Exist», если слово уже находится в словаре.

- word — удалить слово «word» из словаря. Программа должна вывести «OK», если слово существовало и было удалено, «NoSuchWord», если слово в словаре не было найдено.

word — найти в словаре слово «word». Программа должна вывести «OK: 34», если слово было найдено; число, которое следует за «OK:» — номер, присвоенный слову при добавлении. В случае, если слово в словаре не было обнаружено, нужно вывести строку «NoSuchWord».

! Save /path/to/file — сохранить словарь в бинарном компактном представлении на диск в файл, указанный парамером команды. В случае успеха, программа должна вывести «OK», в случае неудачи выполнения операции, программа должна вывести описание ошибки (см. ниже).

! Load /path/to/file — загрузить словарь из файла. Предполагается, что файл был ранее подготовлен при помощи команды Save. В случае успеха, программа должна вывести строку «OK», а загруженный словарь должен заменить текущий (с которым происходит работа); в случае неуспеха, должна быть выведена диагностика, а рабочий словарь должен остаться без изменений. Кроме системных ошибок, программа должна корректно обрабатывать случаи несовпадения формата указанного файла и представления данных словаря во внешнем файле.

Для всех операций, в случае возникновения системной ошибки (нехватка памяти, отсутсвие прав записи и т.п.), программа должна вывести строку, начинающуюся с «ERROR:» и описывающую на английском языке возникшую ошибку.

Различия вариантов заключаются только в используемых структурах данных:

Красно-чёрное дерево.

Общие сведения о программе

Программа представлена файлом **– main.cpp**.

Словарь реализован с помощью класса **Dictionary**, имеющего поле класса **RedBlackTree**, методы, соответствующие командам, поступающим на ввод, а также метод для печати дерева, необходимый для проверки и отладки программы.

Внутри класса **RedBlackTree** объявлены поля **nullNode** и **root** — указатель на лист и корень дерева соответственно.

Все узлы дерева реализуются структурой **Node** со следующими полями:

key – ключ элемента (строка);

data – значение элемента (число);

color — цвет узла (тип задается с помощью **enum**);

left, right, parent — левый, правый сын и родитель соответственно (указатели на **Node**)

Для класса **RedBlackTree** реализовано правило пяти, а также методы поиска, вставки и удаления элемента (**find, insert, erase**), поиска минимального и максимального элемента в поддереве (**min, max**), очистки дерева (**clear**), предикаты, сравнивающие текущий узел с **nullNode** (**isNull**), с левым или правым сыном другого узла (**isLeftSon**, **isRightSon**), методы загрузки дерева из файла (**load**), сохранения в файл (**save**) и печати (**printTree**).

Для реализации этих методов пришлось также реализовать следующие приватные методы:

- **leftRotate** и **rightRotate** — левый и правый поворот;

- **InsertFixup** и **eraseFixup** — перекраска узлов дерева после вставки и удаления, чтобы сохранить свойства красно-черного дерева;
- **replace** — замена одного поддерева другим;
- **deleteSubtree** — удаление поддерева;
- **copy** — копирование одного поддерева в другое;
- **saveNodes** — рекурсивное сохранение поддерева в файл.

Общий алгоритм решения

1. **Поиск** в красно-черном дереве работает так же, как и в бинарном:
   - начинается из корневого элемента;
   - если он меньше искомого, идем в правое поддерево, а если больше — в левое;
   - если текущий элемент — лист, тогда искомого элемента в дереве нет.
2. **Вставка.** Новый узел вставляется как в красно-черное дерево, как в бинарное, его цвет красный.
   - Если узел вставляется в корень, то он перекрашивается в черный.
   - Если родитель нового узла черный, то свойства красно-черного дерева не нарушены.
   - Если и родитель и дядя нового узла — красные, то они оба перекрашиваются в чёрный, а их предок (дедушка нового узла) становится красным. Процедура исправления свойств красно-черного дерева рекурсивно запускается из дедушки.
   - Если родитель является красным, но дядя — чёрный, текущий узел — левый сын и родитель — левый сын, тогда выполняется правый поворот дерева из дедушки.
   - Если родитель является красным, но дядя — чёрный, а также, текущий узел — правый сын, а родитель в свою очередь — левый сын, тогда может быть произведен поворот дерева, который меняет роли текущего узла и его родителя. Теперь задача сводится к предыдущему случаю.

В симметричных случаях вставка выполняется по симметричному алгоритму.

3. **Удаление.** Находим узел для удаления (аналогично поиску). Если у удаляемого узла нет потомков или только один потомок, просто удаляем его. Если у удаляемого узла два потомка, находим его преемника (например, наименьший ключ в правом поддереве) и заменяем удаляемый узел преемником.

   Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной.

   Рассмотрим ребёнка удалённой вершины:

   - Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца - в красный цвет.
   - Если брат текущей вершины был чёрным, то получаем три случая:
     - Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины.
     - Если у брата правый ребёнок чёрный,а левый красный, то перекрашиваем брата и его левого сына и делаем вращение.
     - В же у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца - в чёрный, делаем вращение и выходим из алгоритма.
   - Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева.

Реализация

main.cpp

```
#include <fstream>
#include <iostream>
#include <sstream>
```

```
#include <string>

enum colorType { red, black };

struct Node {
colorType color;
std::string key;
u_int64_t data;
Node* left;
Node* right;
Node* parent;

Node()
: color(black),
key(std::string()),
data(u_int64_t()),
left(nullptr),
right(nullptr),
parent(nullptr) {}

Node(colorType colorValue, const std::string& keyValue,
const u_int64_t& dataValue, Node* leftChild, Node* rightChild,
Node* parentNode)
: color(colorValue),
key(keyValue),
data(dataValue),
left(leftChild),
right(rightChild),
parent(parentNode) {}
};

class RedBlackTree {
private:
Node* nullNode;
Node* root;
size_t elementsCount;

void leftRotate(Node* oldRoot) {
if (isNull(oldRoot) || isNull(oldRoot->right)) {
return;
}
Node* newRoot = oldRoot->right;
oldRoot->right = newRoot->left;
if (!isNull(newRoot->left)) {
newRoot->left->parent = oldRoot;
}
newRoot->parent = oldRoot->parent;
if (isNull(oldRoot->parent)) {
this->root = newRoot;
```

```
} else if (isLeftSon(oldRoot)) {
oldRoot→parent→left = newRoot;
} else {
oldRoot→parent→right = newRoot;
}
newRoot→left = oldRoot;
oldRoot→parent = newRoot;
}

void rightRotate(Node* oldRoot) {
if (isNull(oldRoot) || isNull(oldRoot→left)) {
return;
}
Node* newRoot = oldRoot→left;
oldRoot→left = newRoot→right;
if (!isNull(newRoot→right)) {
newRoot→right→parent = oldRoot;
}
newRoot→parent = oldRoot→parent;
if (isNull(oldRoot→parent)) {
this→root = newRoot;
} else if (isLeftSon(oldRoot)) {
oldRoot→parent→left = newRoot;
} else {
oldRoot→parent→right = newRoot;
}
newRoot→right = oldRoot;
oldRoot→parent = newRoot;
}

void insertFixup(Node* brokenNode) {
while (!isNull(brokenNode→parent) && brokenNode→parent→color ≠
black) {
Node* uncle = isLeftSon(brokenNode→parent)
? brokenNode→parent→parent→right
: brokenNode→parent→parent→left;
if (uncle→color == red) {
brokenNode→parent→color = black;
uncle→color = black;
brokenNode→parent→parent→color = red;
brokenNode = brokenNode→parent→parent;
} else {
if (isLeftSon(brokenNode→parent) && isRightSon(brokenNode)) {
brokenNode = brokenNode→parent;
leftRotate(brokenNode);
} else if (isRightSon(brokenNode→parent) && isLeftSon(brokenNode)) {
brokenNode = brokenNode→parent;
rightRotate(brokenNode);
}
```

```
brokenNode→parent→color = black;
brokenNode→parent→parent→color = red;
if (isLeftSon(brokenNode→parent)) {
rightRotate(brokenNode→parent→parent);
} else {
leftRotate(brokenNode→parent→parent);
}
}
}
root→color = black;
}

void replace(Node* oldSubtree, Node* newSubtree) {
if (isNull(oldSubtree)) {
return;
}
if (isNull(oldSubtree→parent)) {
root = newSubtree;
} else if (isLeftSon(oldSubtree)) {
oldSubtree→parent→left = newSubtree;
} else {
oldSubtree→parent→right = newSubtree;
}
if (!isNull(newSubtree)) {
newSubtree→parent = oldSubtree→parent;
}
}

void eraseFixup(Node* brokenNode, Node* parent) {
while (brokenNode ≠ root && brokenNode→color == black) {
Node* brother =
isLeftSon(brokenNode, parent) ? parent→right : parent→left;
if (brother→color == red) {
brother→color = black;
parent→color = red;
if (isLeftSon(brokenNode, parent)) {
leftRotate(parent);
brother = parent→right;
} else {
rightRotate(parent);
brother = parent→left;
}
}
if (brother→left→color == black && brother→right→color == black) {
brother→color = red;
brokenNode = parent;
parent = parent→parent;
} else {
```

```
if (isLeftSon(brokenNode, parent)) {
if (brother->right->color == black) {
brother->left->color = black;
brother->color = red;
rightRotate(brother);
brother = parent->right;
}
brother->color = parent->color;
parent->color = black;
brother->right->color = black;
leftRotate(parent);
} else {
if (brother->left->color == black) {
brother->right->color = black;
brother->color = red;
leftRotate(brother);
brother = parent->left;
}
brother->color = parent->color;
parent->color = black;
brother->left->color = black;
rightRotate(parent);
}
brokenNode = root;
parent = nullNode;
}
}
brokenNode->color = black;
// printTree();
}

void deleteSubtree(Node* root) {
if (isNull(root)) {
return;
}
if (!isNull(root->left)) {
deleteSubtree(root->left);
}
if (!isNull(root->right)) {
deleteSubtree(root->right);
}
if (isNull(root->parent)) {
this->root = nullNode;
} else if (isLeftSon(root)) {
root->parent->left = nullNode;
} else {
root->parent->right = nullNode;
}
delete root;
```

```
root = nullptr;
}

void copy(Node*& thisNode, const Node* otherNode,
const RedBlackTree& otherTree) {
if (otherTree.isNull(otherNode)) {
if (!isNull(thisNode)) {
deleteSubtree(thisNode->left);
deleteSubtree(thisNode->right);
}
thisNode = nullNode;
return;
}
if (isNull(thisNode)) {
thisNode = new Node();
}
thisNode->parent = nullNode;
thisNode->color = otherNode->color;
thisNode->key = otherNode->key;
thisNode->data = otherNode->data;
copy(thisNode->left, otherNode->left, otherTree);
if (otherNode->left != otherTree.nullNode) {
thisNode->left->parent = thisNode;
}
copy(thisNode->right, otherNode->right, otherTree);
if (otherNode->right != otherTree.nullNode) {
thisNode->right->parent = thisNode;
}
}

void saveNodes(std::ofstream& outputFile, const Node* node) const {
if (isNull(node)) {
return;
}
size_t length = node->key.size();
outputFile.write(reinterpret_cast<const char*>(&length),
sizeof(size_t));
outputFile.write(reinterpret_cast<const char*>(node->key.c_str()),
length * sizeof(char));
outputFile.write(reinterpret_cast<const char*>(&node->data),
sizeof(u_int64_t));
saveNodes(outputFile, node->left);
saveNodes(outputFile, node->right);
}

public:
RedBlackTree() {
nullNode = new Node();
root = nullNode;
```

```
root→parent = nullNode;
elementsCount = 0;
}

RedBlackTree(const RedBlackTree& other) : RedBlackTree() {
copy(root, other.root, other);
elementsCount = other.elementsCount;
}

RedBlackTree(RedBlackTree&& other)
: nullNode(other.nullNode),
root(other.root),
elementsCount(other.elementsCount) {
other.nullNode = other.root = nullptr;
other.elementsCount = 0;
}

virtual ~RedBlackTree() {
deleteSubtree(root);
delete nullNode;
nullNode = root = nullptr;
}

RedBlackTree& operator=(const RedBlackTree& other) {
copy(root, other.root, other);
elementsCount = other.elementsCount;
return *this;
}

RedBlackTree& operator=(RedBlackTree&& other) {
deleteSubtree(root);
delete nullNode;
nullNode = other.nullNode;
root = other.root;
elementsCount = other.elementsCount;
other.nullNode = other.root = nullptr;
other.elementsCount = 0;
return *this;
}

Node* find(const std::string& key) const {
Node* foundNode = root;
while (!isNull(foundNode) && foundNode→key ≠ key) {
foundNode = key < foundNode→key ? foundNode→left : foundNode→right;
}
return foundNode;
}

void insert(const std::string& key, const u_int64_t& data) {
```

```
Node* parent = nullNode;
Node* child = root;
while (!isNull(child)) {
parent = child;
if (key < child→key) {
child = child→left;
} else if (key > child→key) {
child = child→right;
} else {
throw std::runtime_error("Exist");
}
}
Node* newChild = new Node(red, key, data, nullNode, nullNode, parent);
if (isNull(parent)) {
root = newChild;
} else if (key < parent→key) {
parent→left = newChild;
} else {
parent→right = newChild;
}
insertFixup(newChild);
++elementsCount;
}

Node* min(Node* root) {
if (isNull(root)) {
return root;
}
while (!isNull(root→left)) {
root = root→left;
}
return root;
}

Node* max(Node* root) {
if (isNull(root)) {
return root;
}
while (!isNull(root→right)) {
root = root→right;
}
return root;
}

void erase(Node* target) {
colorType erasedColor = target→color;
Node *movedNode, *parent;
if (isNull(target→left)) {
movedNode = target→right;
```

```
parent = target->parent;
replace(target, target->right);
} else if (isNull(target->right)) {
movedNode = target->left;
parent = target->parent;
replace(target, target->left);
} else {
Node* rightSubtreeMinimum = min(target->right);
erasedColor = rightSubtreeMinimum->color;
movedNode = rightSubtreeMinimum->right;
parent = rightSubtreeMinimum;
if (rightSubtreeMinimum->parent != target) {
parent = rightSubtreeMinimum->parent;
replace(rightSubtreeMinimum, rightSubtreeMinimum->right);
rightSubtreeMinimum->right = target->right;
rightSubtreeMinimum->right->parent = rightSubtreeMinimum;
}
replace(target, rightSubtreeMinimum);
rightSubtreeMinimum->left = target->left;
rightSubtreeMinimum->left->parent = rightSubtreeMinimum;
rightSubtreeMinimum->color = target->color;
}
if (erasedColor == black) {
eraseFixup(movedNode, parent);
}
delete target;
--elementsCount;
}

void clear() {
deleteSubtree(root);
elementsCount = 0;
}

bool isNull(const Node* checkedNode) const {
return checkedNode == nullptr || checkedNode == nullNode;
}

bool isLeftSon(const Node* child, const Node* parent = nullptr) const {
if (!parent) {
parent = child->parent;
}
return !isNull(parent) ? child == parent->left : false;
}

bool isRightSon(const Node* child, const Node* parent = nullptr) const
{
if (!parent) {
parent = child->parent;
```

```
}
return !isNull(parent) ? child == parent->right : false;
}

void save(std::ofstream& outputFile) {
outputFile.write(reinterpret_cast<const char*>(&elementsCount),
sizeof(size_t));

saveNodes(outputFile, root);
}

void load(std::ifstream& inputFile) {
clear();
size_t count{0};
inputFile.read(reinterpret_cast<char*>(&count), sizeof(size_t));
std::string key;
u_int64_t data;
size_t length;
for (size_t i{0}; i < count; ++i) {
inputFile.read(reinterpret_cast<char*>(&length), sizeof(size_t));
key.resize(length);
inputFile.read(reinterpret_cast<char*>(key.data()),
length * sizeof(char));
inputFile.read(reinterpret_cast<char*>(&data), sizeof(u_int64_t));
insert(key, data);
}
}

void printTree(size_t depth = 0, Node* node = nullptr) {
if (!node) {
node = root;
std::cout << "new tree" << std::endl;
}
if (isNull(node)) {
std::cout << std::endl;
return;
}
std::cout << std::endl;
printTree(depth + 1, node->right);
for (size_t i{0}; i < 2 * depth; ++i) {
std::cout << "\t";
}
if (node->color == red) {
std::cout << "red: ";
} else {
std::cout << "black: ";
}
// std::cout << node->key;
std::cout << std::endl;
```

```
printTree(depth + 1, node->left);
std::cout << std::endl;
}
};

std::string& toLowerCase(std::string& word) {
for (size_t i{0}; i < word.size(); ++i) {
word[i] = std::tolower(word[i]);
}
return word;
}

class Dictionary {
private:
RedBlackTree tree;

public:
Dictionary() = default;

void insert(std::string& key, const u_int64_t& value) {
try {
tree.insert(toLowerCase(key), value);
// std::cout << "inserted ";
std::cout << "OK" << std::endl;
} catch (std::runtime_error& error) {
std::cout << error.what() << std::endl;
}
}

void at(std::string& key) {
Node* foundNode = tree.find(toLowerCase(key));
if (!tree.isNull(foundNode)) {
// std::cout << "found ";
std::cout << "OK: " << foundNode->data << std::endl;
} else {
std::cout << "NoSuchWord" << std::endl;
}
}

void erase(std::string& key) {
Node* foundNode = tree.find(toLowerCase(key));
if (!tree.isNull(foundNode)) {
tree.erase(foundNode);
// std::cout << "erased ";
std::cout << "OK" << std::endl;
} else {
std::cout << "NoSuchWord" << std::endl;
}
}
```

```
void save(std::string& path) {
std::ofstream outputFile;
outputFile.open(path, std::ios::trunc | std::ios::out |
std::ios::binary);
tree.save(outputFile);
outputFile.close();
// std::cout << "saved ";
std::cout << "OK" << std::endl;
}

void load(std::string& path) {
std::ifstream inputFile;
inputFile.open(path, std::ios::binary | std::ios::in);
tree.load(inputFile);
inputFile.close();
// std::cout << "loaded ";
std::cout << "OK" << std::endl;
}

void print() { tree.printTree(); }
};

int main() {
freopen("input.txt", "r", stdin);
Dictionary dictionary;
std::string input;
std::string line;
while (std::getline(std::cin, line)) {
std::cin.clear();
std::istringstream read{line};
read >> input;
if (input == "+") {
std::string word;
u_int64_t number;
read >> word >> number;
dictionary.insert(word, number);
} else if (input == "-") {
std::string word;
read >> word;
dictionary.erase(word);
} else if (input == "!") {
std::string operation, path;
read >> operation >> path;
if (operation == "Save") {
dictionary.save(path);
} else {
dictionary.load(path);
}
```

```
// } else if (input == "#") {
// dictionary.print();
} else {
dictionary.at(input);
}
}
}
```

Пример работы

Test 1

Input	Output
+ a 1 + A 2 + aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaa 18446744073709551615 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaa A - A a	OK Exist OK OK: 18446744073709551615 OK: 1 OK NoSuchWord

Test 2

Input	Output

+ qhbzvuziy 18446744071656752066	OK
+ ngcfxhzxwssbnfabxqsqbeybfudrceyhtlondsslaqtayqlptjllsunqffswmuucoyprobsdiyndzexazdwnhwnsnodufkrafgwlm 18446744073589059394	OK
+ cjkcgwtsiqbbqzduhteamirgohenlbxsdwvftzjlbvghrvpjfpmuxucewsmbdjvykpjuemhxdbihgrnsvrnfqfqhguffktuhrkwbuuks 18446744072112168448	OK
+ tyepiggrerbdsiragmrknhrhaoixzmgcpknppikqvmbblympufnnjmqtqvqujkllpveuhprseijtdknhozyddeamwnxgzotjsdltwdcujxhbkijzpqphtodfbaruyncnshfvbwiyxykcheadgqejohvxozhvycudqakhbcabfom 18446744072698716530	OK
+ omybshzqxiihdenrnbdmfvqyyjsettlxwonvnrmtprxvvvxzwbrlxxndyrguscfwmzfswzxqufhqvkyzhqxjfokjugqsfhhkqlemyulxeimnzphrifabstwijlpvdwizptuuneroelyjleisbcmcxikz 584590942	OK
! Save qwerty	Exist
cjkcgwtsiqbbqzduhteamirgohenlbxsdwvftzjlbvghrvpjfpmuxucewsmbdjvykpjuemhxdbihgrnsvrnfqfqhguffktuhrkwbuuks	Exist
- cjkcgwtsiqbbqzduhteamirgohenlbxsdwvftzjlbvghrvpjfpmuxucewsmbdjvykpjuemhxdbihgrnsvrnfqfqhguffktuhrkwbuuks	Exist
qhbzvuziy	OK
- qhbzvuziy	OK
tyepiggrerbdsiragmrknhrhaoixzmgcpknppikqvmbblympufnnjmqtqvqujkllpveuhprseijtdknhozyddeamwnxgzotjsdltwdcujxhbkijzpqphtodfbaruyncnshfvbwiyxykcheadgqejohvxozhvycudqakhbcabfom	OK
- tyepiggrerbdsiragmrknhrhaoixzmgcpknppikqvmbblympufnnjmqtqvqujkllpveuhp	OK
+	OK
+	OK
+	OK
+	OK: 18446744072112168448
+	OK
+	OK: 18446744071656752066
+	OK
+	OK: 18446744072698716530
+	OK
+	OK: 584590942
+	OK
+	OK: 18446744073589059394
+	OK
+	OK

rseijtdknhozyddeamwnxgzotjsdltwdcuj xhbkijzpqphtodfbaruyncnshfvbwiyxykc headgqejohvxozhvycudqakhbcabfom omybshzqxiihdenrnbdmfvqyyjsettlxwon vnrmtprxvvvxzwbrlxxndyrguscfwmzfswz xqufhqvkyzhqxjfokjugqsfhhkqlemyulxe imnzphrifabstwijlpvdwizptuuneroelyj leisbcmcxikz – omybshzqxiihdenrnbdmfvqyyjsettlxwon vnrmtprxvvvxzwbrlxxndyrguscfwmzfswz xqufhqvkyzhqxjfokjugqsfhhkqlemyulxe imnzphrifabstwijlpvdwizptuuneroelyj leisbcmcxikz ngcfxhzxwssbnfabxqsqbeybfudrceyhtlo ndsslaqtayqlptjllsunqffswmuucoyprob sdiyndzexazdwnhwnsnodufkrafgwlm – ngcfxhzxwssbnfabxqsqbeybfudrceyhtlo ndsslaqtayqlptjllsunqffswmuucoyprob sdiyndzexazdwnhwnsnodufkrafgwlm ! Load qwerty + qhbzvuziy + ngcfxhzxwssbnfabxqsqbeybfudrceyhtlo ndsslaqtayqlptjllsunqffswmuucoyprob sdiyndzexazdwnhwnsnodufkrafgwlm + cjkcgwtsiqbbqzduhteamirgohenlbxsdwv ftzjlbvghrvpjfpmuxucewsmbdjvykpjuem hxdbihgrnsvrnfqfqhguffktuhrkwbuuks + tyepiggrerbdsiragmrknhrhaoixzmgcpkn ppikqvmbblympufnnjmqtqvqujkllpveuhp rseijtdknhozyddeamwnxgzotjsdltwdcuj xhbkijzpqphtodfbaruyncnshfvbwiyxykc headgqejohvxozhvycudqakhbcabfom + omybshzqxiihdenrnbdmfvqyyjsettlxwon vnrmtprxvvvxzwbrlxxndyrguscfwmzfswz xqufhqvkyzhqxjfokjugqsfhhkqlemyulxe imnzphrifabstwijlpvdwizptuuneroelyj	

leisbcmcxikz - qhbzvuziy - ngcfxhzxwssbnfabxqsqbeybfudrceyhtlondsslaqtayqlptjllsunqffswmuucoyprobsdiyndzexazdwnhwnsnodufkrafgwlm - cjkcgwtsiqbbqzduhteamirgohenlbxsdwvftzjlbvghrvpjfpmuxucewsmbdjvykpjuemhxdbihgrnsvrnfqfqhguffktuhrkwbuuks - tyepiggrerbdsiragmrknhrhaoixzmgcpknppikqvmbblympufnnjmqtqvqujkllpveuhprseijtdknhozyddeamwnxgzotjsdltwdcujxhbkijzpqphtodfbaruyncnshfvbwiyxykcheadgqejohvxozhvycudqakhbcabfom - omybshzqxiihdenrnbdmfvqyyjsettlxwonvnrmtprxvvvxzwbrlxxndyrguscfwmzfswzxqufhqvkyzhqxjfokjugqsfhhkqlemyulxeimnzphrifabstwijlpvdwizptuuneroelyjleisbcmcxikz	

Test 3

Input	Output
+ QBvzYnCXZ 312096615 + sBfBqQeBUreHlNslQaQPJlUQfwucYrbdYDEAdNWSOUKAglhjcWsQbZUtaIghNBSWFZLvHVJpUuesBJyPuMXbhRSRffhUftHkbuStEiGEBsrgrNRaIzgpNpkvBlMUNjQQqjLPeHreJdNOYdaWxzTSLWcJHkJpPtDbrYCSFbixkhAgEOVOHyUqKbAfmoYSzxIdNNdFqYsTLwNNMPXVXWRXNYgsfMfWxUhvYhXfKuqFhQey 18446744073356597424 + xiNPrfbtILVwZtUEolJeScCiziepwXvOFslNFEvBZTvZhQLCrmIgDwBcBizWIwyqNxFDJHTfSPRTGwyydQyPlmXgPRclWWHGYFAUDLsZrFVslfAeqbDLsHhEoRPSydyjTGwNuCDtPGXRxlCrnnTfCETHeAJXGhQ 18446744072158200836	OK OK OK OK OK: 18446744072158200836 OK OK: 312096615 OK OK: 18446744073356597424 OK OK

! Save qwerty	Exist
xiNPrfbtILVwZtUEolJeScCiziepwXvOFsl NFEvBZTvZhQLCrmIgDwBcBizWIwyqNxFDJH TfSPRTGwyydQyPlmXgPRclWWHGYFAUDLsZr FVslfAeqbDLsHhEoRPSydyjTGwNuCDtPGXR xlCrnnTfCETHeAJXGhQ	Exist
- xiNPrfbtILVwZtUEolJeScCiziepwXvOFsl NFEvBZTvZhQLCrmIgDwBcBizWIwyqNxFDJH TfSPRTGwyydQyPlmXgPRclWWHGYFAUDLsZr FVslfAeqbDLsHhEoRPSydyjTGwNuCDtPGXR xlCrnnTfCETHeAJXGhQ	Exist
QBvzYnCXZ	OK
- QBvzYnCXZ	OK
sBfBqQeBUreHlNslQaQPJlUQfwucYrbdYDE AdNWSOUKAglhjcWsQbZUtaIghNBSWFZLvHV JpUuesBJyPuMXbhRSRffhUftHkbuStEiGEB srgrNRaIzgpNpkvBlMUNjQQqjLPeHreJdNO YdaWxzTSLWcJHkJpPtDbrYCSFbixkhAgEOV OHyUqKbAfmoYSzxIdNNdFqYsTLwNNMPXVXW RXNYgsfMfWxUhvYhXfKuqFhQey	OK
- sBfBqQeBUreHlNslQaQPJlUQfwucYrbdYDE AdNWSOUKAglhjcWsQbZUtaIghNBSWFZLvHV JpUuesBJyPuMXbhRSRffhUftHkbuStEiGEB srgrNRaIzgpNpkvBlMUNjQQqjLPeHreJdNO YdaWxzTSLWcJHkJpPtDbrYCSFbixkhAgEOV OHyUqKbAfmoYSzxIdNNdFqYsTLwNNMPXVXW RXNYgsfMfWxUhvYhXfKuqFhQey	
! Load qwerty	
+ qbvzyncxz	
+ sbfbqqeburehlnslqaqpjluqfwucyrbdyde adnwsoukaglhjcwsqbzutaighnbswfzlvhv jpuuesbjypumxbhrsrffhufthkbusteigeb srgrnraizgpnpkvblmunjqqqjlpehrejdno ydawxztslwcjhkjpptdbrycsfbixkhageov ohyuqkbafmoyszxidnndfqystlwnnmpxvxw rxnygsfmfwxuhvyhxfkuqfhqey	
+ xinprfbtilvwztueoljesccziepwxvofsl nfevbztvzhqlcrmigdwbcbizwiwyqnxfdjh tfsprtgwyydqyplmxgprclwwhgyfaudlszr	

fvslfaeqbdlshheorpsydyjtgwnucdtpgxr xlcrnntfcetheajxghq - qbvzyncxz - sbfbqqeburehlnslqaqpjluqfwucyrbdyde adnwsoukaglhjcwsqbzutaighnbswfzlvhv jpuuesbjypumxbhrsrffhufthkbusteigeb srgrnraizgpnpkvblmunjqqqjlpehrejdno ydawxztslwcjhkjpptdbrycsfbixkhageov ohyuqkbafmoyszxidnndfqystlwnnmpxvxw rxnygsfmfwxuhvyhxfkuqfhqey - xinprfbtilvwztueoljesccciziepwxvofsl nfevbztvzhqlcrmigdwbcbizwiwyqnxfdjh tfsprtgwyydqyplmxgprclwwhgyfaudlszr fvslfaeqbdlshheorpsydyjtgwnucdtpgxr xlcrnntfcetheajxghq	

Вывод

Выполняя лабораторную работу я изучил сбалансированные деревья, а также алгоритмы поиска, вставки и удаления элементов из них, и на практике реализовал красно-черное дерево. Сбалансированные деревья имеют преимущество перед обычными. Высота в них $h \sim \log(n)$, а для обычного дерева в худшем случае $h = n$. Так как операции поиска, вставки и удаления в бинарном дереве работают со сложностью $O(h)$, то использование сбалансированных деревьев, например красно-черного, будет намного эффективнее.