МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Государственное автономное образовательное учреждение высшего образования

«Новосибирский государственный технический университет»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Отчет по контрольной работе №1

Коллекция данных – дерево поиска

Вариант 8

Выполнила студентка группы ДТ-160

Буянкина Елизавета Алексеевна

Проверил преподаватель

Романенко Т.А.

Новосибирск – 2024

# Задание

Спроектировать и реализовать АТД «BST – дерево» для коллекции, содержащей ключи и данные произвольного типа. Типы ключей и данных задаются клиентской программой в виде параметров шаблонного класса «BST – дерево».

Интерфейс АТД «BST – дерево» включает следующие операции:

• опрос размера дерева (количества узлов),

• очистка дерева (удаление всех узлов),

• проверка дерева на пустоту,

• поиск данных с заданным ключом,

• включение в дерево нового узла с заданным ключом и данными,

• удаление из дерева узла с заданным ключом,

• обход узлов в дереве по схеме, заданной в варианте задания, и вывод ключей в порядке обхода,

• дополнительная операция, заданная в варианте задания.

• операция создания итератора

• операции итератора \*, ++, ==

• вывод структуры дерева на экран.

Вариант 8

• Алгоритмы основных операций АТД (вставки, удаления и поиска) реализуются в рекурсивной форме.

• Создание и операции обратного итератора.

• Схема операции обхода: Lt → Rt → t.

• Дополнительная операция: поиск для заданного ключа предыдущего по значению ключа в дереве.

# Формат АТД

Двоичное бинарное дерево поиска  (BST - дерево) - упорядоченное дерево, каждая вершина (узел)  которого имеет не более двух потомков, причем каждый из потомков считается либо левым сыном,  либо правым сыном  своего родителя.

Как абстрактный тип данных, BST-дерево предусматривает операции поиска, вставки и удаления элементов по ключу. Используя эти операции можно построить любое бинарное дерево. Операции вставки, удаления и поиска элементов для BST-дерева используют правило двоичного поиска при доступе к элементу с заданным значением ключа.

## Данные

**Параметры:**

***KeyType*** – тип ключей, хранящихся в дереве

***DataType*** – тип данных, хранящихся в дереве

***size*** – количество элементов, хранящихся в дереве

**Структура хранения коллекции:**

Связная структура дерева на базе адресных указателей. Каждый узел дерева размещается в динамической памяти и содержит помимо ключа типа ***KeyType*** и данных типа ***DataType*** два указателя на левого и правого сыновей.

## Операции:

1. Конструктор BSTree()

**Вход**: нет

**Предусловия**: нет

**Процесс:** создание пустого дерева

**Выход**: нет

**Постусловия:** создано пустое дерево с числом элементов ***size*** = 0

1. Опрос размера дерева getSize()

**Вход**: нет

**Предусловия**: нет

**Процесс:** возврат текущего количества значения ***size***, хранящихся в дереве

**Выход**: размер дерева ***size***

**Постусловия:** нет

1. Очистка дерева clear()

**Вход**: нет

**Предусловия**: нет

**Процесс:** удаление всех узлов

**Выход**: нет

**Постусловия:** удалены все элементы, число элементов ***size*** = 0

1. Проверка дерева на пустоту isEmpty()

**Вход**: нет

**Предусловия**: нет

**Процесс:** проверка наличия хотя бы одного элемента

**Выход**: булевое значение true или false

**Постусловия:** нет

1. Поиск данных с заданным ключом find(const KeyType& key)

**Вход**: key – ключ элемента

**Предусловия**: ключ существует

**Процесс:** поиск ключа

**Выход**: данные, хранящиеся по ключу или генерация исключения при невыполнении предусловия

**Постусловия**: нет

1. Включение в дерево нового узла с заданным ключом и данными insert(const KeyType& key, const DataType& value)

**Вход**: key – ключ элемента, value – значение элемента

**Предусловия**: ключ key не существует

**Процесс:** добавление в дерево узла с ключом key

**Выход**: нет

**Постусловия**: элемент с ключом key добавлен, ***size = size +* 1**

1. Удаление из дерева узла с заданным ключом removeByKey(const KeyType& key)

**Вход**: нет

**Предусловия**: ключ key существует

**Процесс:** удаление из дерева узла с ключом key

**Выход**: нет

**Постусловия**: элемент с ключом key удален, ***size = size -* 1**

1. Вывод структуры дерева printTree()

**Вход**: нет

**Предусловия**: нет

**Процесс:** вывод структуры дерева на экран

**Выход**: нет

**Постусловия:** нет

1. Обход узлов в дереве Lt\_Rt\_t()

**Вход**: нет

**Предусловия**: нет

**Процесс:** обход узлов в дереве по схеме Lt Rt t и вывод ключей в порядке обхода

**Выход**: нет

**Постусловия:** нет

1. Поиск для заданного ключа предыдущего по значению ключа в дереве predecessorKey(const KeyType& key)

**Вход**: нет

**Предусловия**: нет

**Процесс:** обход узлов в дереве по схеме Lt Rt t и вывод ключей в порядке обхода

**Выход**: нет

**Постусловия:** нет

1. Запрос обратного итератора rbegin( )

**Вход**: нет

**Предусловия**: ***size* ≠ 0**

**Процесс:** формирование итератора, установленного на максимальный элемент дерева

**Выход:** итератор для доступа к элементам ***rbegin***( ) или «неустановленный» итератор ***rend***( ) при невыполнении предусловия

**Постусловия:** нет

1. Запрос «неустановленного» обратного итератора rend( )

**Вход**: нет

**Предусловия**: нет

**Процесс:** формирование «неустановленного» обратного итератора, указывающего на позицию после наименьшего элемента дерева.

**Выход:** «неустановленный» итератор произвольного доступа *rend*( )

**Постусловия:** нет

# Cправочное определение класса для коллекции «BST – дерево»

template <class KeyType, class DataType>

class BSTree {

public:

BSTree(); // конструктор

size\_t getSize() const; // опрос размера дерева

void clear(); // очистка дерева

bool isEmpty() const; // проверка дерева на пустоту

const DataType& find(const KeyType& key) const; // поиск данных с заданным ключом

void insert(const KeyType& key, const DataType& value); // включение в дерево нового узла с заданным ключом и данными

void removeByKey(const KeyType& key); // удаление из дерева узла с заданным ключом

void printTree(); // вывод структуры дерева

void Lt\_Rt\_t(); // обход узлов в дереве

const KeyType& predecessorKey(const KeyType& key); // поиск для заданного ключа предыдущего по значению ключа в дереве

class rIterator { // Обратный итератор

public:

DataType& operator\* (); // доступ к данным текущего элемента

rIterator& operator ++(); //оператор инкремента итератора

bool operator == (const rIterator& other); // проверка равенства

bool operator != (const rIterator& other); // проверка неравенства

};

rIterator rbegin(); // запрос обратного итератора

rIterator rend(); // запрос неустановленного обратного итератора

};

# Выводы

Был спроектирован и реализован АТД «BST – дерево» для коллекции, содержащей ключи и данные произвольного типа. Реализация «BST – дерева» предоставляет эффективную структуру для хранения данных с возможностью быстрого поиска, вставки и удаления элементов. Использование итератора позволяет удобно обходить элементы в порядке, установленном структурой дерева.

# Список использованной литературы

1. Альфред Ахо, Джон Э. Хопкрофт, Д. Ульман Структуры данных и алгоритмы. - М. - СПб – Киев: «Вильямс», 2000 г. – 384 с.

2. Фрэнк М. Каррано, Джанет Дж. Причард. Абстракция данных и решение задач на С++. Стены и зеркала. - М. - СПб – Киев: «Вильямс», 2003 г. – 848 с.

3. Т. Кормен, Ч. Лейзерсон, Р. Ривест Алгоритмы. Анализ и построение. - М: «БИНОМ», 2000 г. – 960 с.

4. Кубенский А.А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на С++. – СПб.: БХВ-Петербург, 2004 г. – 464 с.

5. Коллинз У.Дж. Структуры данных и стандартная библиотека шаблонов. – М.: ООО «Бином-Пресс», 2004. – 624 с.

6. Роберт Сэджвик. Фундаментальные алгоритмы на С++. Части 1-5. - М: «DiaSoft», 2001 г. – 688 с.

# Приложение с текстами программ

## BSTree.h

#include<iostream>

using namespace std;

template <class KeyType, class DataType>

class BSTree {

private:

size\_t size = 0; // Количество элементов в дереве

protected:

class Node { // Узел дерева

public:

KeyType key; // Ключ

DataType value; // Значение

Node\* leftPtr = nullptr, // Указатель на левого ребенка

\* rightPtr = nullptr; // Указатель на правого ребенка

Node(const KeyType& key, const DataType& value) : key(key), value(value) {}

};

Node\* head = nullptr; // Корень дерева

public:

BSTree(); // конструктор пустого дерева

~BSTree(); // деструктор

size\_t getSize() const; // опрос размера дерева (количества узлов)

void clear(); // очистка дерева (удаление всех узлов)

bool isEmpty() const; // проверка дерева на пустоту

const DataType& find(const KeyType& key) const; // поиск данных с заданным ключом

void insert(const KeyType& key, const DataType& value); // включение в дерево нового узла с заданным ключом и данными

void removeByKey(const KeyType& key); // удаление из дерева узла с заданным ключом

void printTree(); // вывод структуры дерева на экран

void Lt\_Rt\_t(); // обход узлов в дереве по схеме Lt Rt t, и вывод ключей в порядке обхода

const KeyType& predecessorKey(const KeyType& key); // поиск для заданного ключа предыдущего по значению ключа в дереве

class rIterator { // Обратный итератор

private:

const Node\* current; // Указатель на текущий узел

BSTree\* tree; // Указатель на родительское дерево

public:

rIterator(BSTree\* tree) { // Создаем указатель для указанного дерева

this->tree = tree;

if (tree == nullptr) {

current = nullptr;

}

else {

current = tree->findMaxNode(tree->head); // Находим наибольший ключ дерева

}

}

DataType& operator\* () {

if (current == nullptr) {

throw exception("Вышли за пределы дерева");

}

return current->value;

}

rIterator& operator ++() { // поиск для заданного ключа предыдущего по значению

if (current != nullptr) {

current = tree->predecessor(current);

}

return \*this;

}

bool operator == (const rIterator& other) { // проверка равенства

return current == other.current;

}

bool operator != (const rIterator& other) { // проверка неравенства

return current != other.current;

}

};

rIterator rbegin() {

return rIterator(this);

}

rIterator rend() {

return rIterator(nullptr);

}

private:

void clear(Node\*& node); // Удаляем узел

const Node\* findNode(const Node\* node, const KeyType& key) const { // Ищем рекурсивно ключ в заданном узле или его детях

if (node == nullptr) {

throw exception("Ключ не существует");

}

if (node->key == key) {

return node;

}

else if (key < node->key) {

return findNode(node->leftPtr, key);

}

else {

return findNode(node->rightPtr, key);

}

}

const Node\* findMaxNode(const Node\* node) const { // Ищем максимальный ключ в заданном узле или его детях

if (node == nullptr) {

throw exception("Узел не существует");

}

while (node->rightPtr != nullptr) { // Итерируемся пока не найдем крайний правый узел

node = node->rightPtr;

}

return node;

}

const Node\* rParent(const Node\* node, const Node\* nodeByKey) { // Ищем родителя заданного узла

if (node == nodeByKey) { // 2. Нашли родителя, возращаемся

return nullptr;

}

else if (nodeByKey->key > node->key) {

const Node\* rp = rParent(node->rightPtr, nodeByKey);

if (rp != nullptr) {

return rp; // 3. Нашли родителя, возращаемся

}

else {

return node; // 1. Нашли родителя, возращаемся

}

}

else {

return rParent(node->leftPtr, nodeByKey);

}

}

const Node\* predecessor(const Node\* node) { // Предыдущий по значению ключа узел для заданного узла

if (node->leftPtr != nullptr) {

return findMaxNode(node->leftPtr); // Ищем максимальный ключ в заданном узле или его детях

}

else {

return rParent(head, node); // Ищем родителя заданного узла

}

}

void insert(Node\*& node, const KeyType& key, const DataType& value); // включение нового узла с заданным ключом и данными в качестве ребенка заданного родительского узла или его детей

void removeByKey(Node\*& node, const KeyType& key); // удаление заданного узла или его ребенка с заданным ключом

void printNode(const Node\* node, const size\_t& level); // вывод на экран заданного узла

void Lt\_Rt\_t(const Node\* node); // обход заданного узла и его детей в дереве по схеме Lt Rt t, и вывод ключей в порядке обхода

};

template<class KeyType, class DataType>

void BSTree<KeyType, DataType>::clear(Node\*& node) { // удаление заданного узла и его детей

if (node == nullptr) {

return;

}

clear(node->leftPtr); // Удаляем детей слева

clear(node->rightPtr); // Удаляем детей справа

delete node; // Удаляем сам узел

node = nullptr; // Удаляем ссылку на очищенный узел

}

template<class KeyType, class DataType>

void BSTree<KeyType, DataType>::insert(Node\*& node, const KeyType& key, const DataType& value) {// включение нового узла с заданным ключом и данными в качестве ребенка заданного родительского узла или его детей

if (node == nullptr) { // Когда нашли свободное место для узла

node = new Node(key, value);

return;

}

if (node->key == key) {

throw exception("Ключ уже существует");

}

else if (key < node->key) {

insert(node->leftPtr, key, value);

}

else {

insert(node->rightPtr, key, value);

}

}

template<class KeyType, class DataType>

void BSTree<KeyType, DataType>::removeByKey(Node\*& node, const KeyType& key) { // удаление заданного узла или его ребенка с заданным ключом

if (node == nullptr) {

throw exception("Ключ не найден");

}

if (node->key == key) { // Нашли узел для удаления

if (node->leftPtr == nullptr && node->rightPtr == nullptr) { // Если нет детей - просто удаляем

delete node;

node = nullptr;

}

else if (node->leftPtr == nullptr) { // Если нет ребенка слева - копируем на место удаляемого узла правого ребенка

Node\* temp = node;

node = node->rightPtr;

delete temp;

}

else if (node->rightPtr == nullptr) { // Если нет ребенка справа - копируем на место удаляемого узла левого ребенка

Node\* temp = node;

node = node->leftPtr;

delete temp;

}

else { // Оба ребенка присутствуют

const Node\* temp = findMaxNode(node->leftPtr); // Находим максимального ребенка удаляемого узла

node->value = temp->value;

node->key = temp->key;

removeByKey(node->leftPtr, temp->key); // Удаляем найденного ребенка

}

}

else if (key < node->key) {

removeByKey(node->leftPtr, key);

}

else {

removeByKey(node->rightPtr, key);

}

}

template<class KeyType, class DataType>

void BSTree<KeyType, DataType>::printNode(const Node\* node, const size\_t& level) { // вывод на экран заданного узла

if (node == nullptr) {

return;

}

printNode(node->rightPtr, level + 1);

for (int i = 0; i < level; i++) {

cout << " ";

}

cout << node->key << endl;

printNode(node->leftPtr, level + 1);

}

template<class KeyType, class DataType>

void BSTree<KeyType, DataType>::Lt\_Rt\_t(const Node\* node) {

if (node == nullptr) {

return;

}

Lt\_Rt\_t(node->leftPtr); // Сначала все узлы слева

Lt\_Rt\_t(node->rightPtr); // Потом все узлы справа

cout << node->key << " "; // Выводим текущий узел

}

template <class KeyType, class DataType>

BSTree<KeyType, DataType>::BSTree() {}

template <class KeyType, class DataType>

BSTree<KeyType, DataType>::~BSTree() {

clear();

}

template<class KeyType, class DataType>

size\_t BSTree<KeyType, DataType>::getSize() const {// опрос размера дерева (количества узлов)

return size;

}

template<class KeyType, class DataType>

void BSTree<KeyType, DataType>::clear() { // очистка дерева (удаление всех узлов)

size = 0;

clear(head); // Удаляем корневой узел и его детей

}

template <class KeyType, class DataType>

bool BSTree<KeyType, DataType>::isEmpty() const { // проверка дерева на пустоту

return head == nullptr; // Проверяем наличие хотя бы одного узла (корневого)

}

template<class KeyType, class DataType>

const DataType& BSTree<KeyType, DataType>::find(const KeyType& key) const { // поиск данных с заданным ключом

return findNode(head, key)->value;

}

template<class KeyType, class DataType>

void BSTree<KeyType, DataType>::insert(const KeyType& key, const DataType& value) { // включение в дерево нового узла с заданным ключом и данными

insert(head, key, value);

size++;

}

template<class KeyType, class DataType>

void BSTree<KeyType, DataType>::removeByKey(const KeyType& key) { // удаление из дерева узла с заданным ключом

removeByKey(head, key);

}

template<class KeyType, class DataType>

void BSTree<KeyType, DataType>::printTree() { // вывод структуры дерева на экран

printNode(head, 0);

}

template<class KeyType, class DataType>

void BSTree<KeyType, DataType>::Lt\_Rt\_t() { // обход узлов в дереве по схеме Lt Rt t, и вывод ключей в порядке обхода

Lt\_Rt\_t(head);

cout << endl;

}

template<class KeyType, class DataType>

const KeyType& BSTree<KeyType, DataType>::predecessorKey(const KeyType& key) { // поиск для заданного ключа предыдущего по значению ключа в дереве

const Node\* nodeForKey = findNode(head, key); // Находим узел по ключу

const Node\* foundNode = predecessor(nodeForKey); // Находим предшественника для найденного узла

if (foundNode == nullptr) {

throw exception("Меньший ключ не существует");

}

return foundNode->key;

}

## Menu.cpp

#include <iostream>

#include "BSTree.h"

using namespace std;

int main() {

setlocale(LC\_ALL, "Russian");

BSTree<int, string> tree = BSTree<int, string>();

int n;

while (true) {

cout << endl << "Меню. Выберите действие:" << endl << endl;

cout << "1. Вставка" << endl;

cout << "2. Удаление" << endl;

cout << "3. Размер дерева" << endl;

cout << "4. Очистка" << endl;

cout << "5. Проверка на пустоту" << endl;

cout << "6. Обход: Lt -> Rt -> t" << endl;

cout << "7. Дополнительная операция: поиск для заданного ключа предыдущего по значению ключа в дереве" << endl;

cout << "8. Печать" << endl;

cout << "9. Сгенерировать дерево" << endl;

cout << "10. Поиск по ключу" << endl;

cout << "11. Проитерироваться с помощью обратного итератора" << endl;

cout << "12. Сравнить обратные итераторы" << endl;

cout << endl;

cin >> n;

cout << endl;

try {

switch (n) {

case 1: {

int key;

string value;

cin >> key >> value;

tree.insert(key, value);

break;

}

case 2: {

int key;

cin >> key;

tree.removeByKey(key);

break;

}

case 3: {

cout << tree.getSize() << endl;

break;

}

case 4: {

tree.clear();

break;

}

case 5: {

cout << tree.isEmpty() << endl;

break;

}

case 6: {

cout << endl;

tree.Lt\_Rt\_t();

break;

}

case 7: {

int key;

cin >> key;

cout << tree.predecessorKey(key);

break;

}

case 8: {

cout << endl;

tree.printTree();

break;

}

case 9: {

tree.clear();

tree.insert(10, "a");

tree.insert(5, "b");

tree.insert(2, "c");

tree.insert(8, "d");

tree.insert(15, "e");

tree.insert(9, "f");

tree.insert(11, "g");

break;

}

case 10: {

int key;

cin >> key;

cout << tree.find(key) << endl;

break;

}

case 11: {

auto it = tree.rbegin();

while (it != tree.rend()) {

cout << \*it << " ";

++it;

}

cout << endl;

break;

}

case 12: {

BSTree<int, string>::rIterator it1 = tree.rbegin();

BSTree<int, string>::rIterator it2 = tree.rbegin();

cout << \*it1 << " == " << \*it2<<" ? -> " << (it1 == it2) << endl;

++it2;

cout << \*it1 << " == " << \*it2 << " ? -> " << (it1 == it2) << endl;

break;

}

default: {

cout << "Попробуйте еще раз" << endl;

break;

}

}

}

catch (exception e) {

cout << e.what() << endl;

}

}

}