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

1. 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.

2. 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. . Положение каждого узла в BST-дереве определяется правилом: ключевое значение левого сына текущего узла меньше ключевого значения текущего узла; ключевое значение правого сына текущего узла больше ключевого значения текущего узла.

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.

2.1 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 элемент дерева размещается в динамической памяти и содержит помимо ключа типа ***KeyType*** и данных типа ***DataType*** два указателя на левого и правого сыновей.

2.2 Операции:

1) Конструктор BSTree()

Вход: нет

Предусловия: нет

Процесс: создание пустого дерева

Выход: нет

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

2) Опрос размера дерева getSize()

Вход: нет

Предусловия: нет

Процесс: возврат текущего количества элементов, хранящихся в дереве

Выход: размер дерева ***size***

Постусловия: нет

3) Очистка дерева clear()

Вход: нет

Предусловия: нет

Процесс: удаление всех элементов

Выход: нет

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

4) Проверка дерева на пустоту isEmpty()

Вход: нет

Предусловия: нет

Процесс: проверка наличия хотя бы одного элемента

Выход: булевое значение true, если дерево пустое, иначе false

Постусловия: нет

5) 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

6) Включение в дерево нового элемента 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** или генерация исключения при невыполнении предусловия

7) Удаление из дерева элемента 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** или генерация исключения при невыполнении предусловия

8) Вывод структуры дерева printTree()

Вход: нет

Предусловия: нет

Процесс: вывод структуры дерева на экран

Выход: нет

Постусловия: нет

9) 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

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

Вход: нет

Предусловия: меньший ключ существует

Процесс: поиск наибольшего ключа, который меньше по значению, чем заданный

Выход: ключ, соответствующий предшественнику заданного ключа или генерация исключения при невыполнении предусловия

Постусловия: нет

11) Запрос обратного итератора rbegin( )

Вход: нет

Предусловия: нет

Процесс: формирование итератора, установленного на максимальный элемент дерева

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

Постусловия: нет

12) Запрос «неустановленного» обратного итератора rend( )

Вход: нет

Предусловия: нет

Процесс: формирование «неустановленного» обратного итератора, указывающего на позицию перед наименьшим элементом дерева.

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

Постусловия: нет

3. Справочное 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 элемента с заданным ключом и данными
     void removeByKey(const KeyType& key); // удаление из дерева элемента 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
      };
```

4. 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.

5. 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 C++. 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 C++. – 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 C++. Части 1-5. - М: «DiaSoft», 2001 г. – 688 с.

6. Приложение с текстами программ

6.1 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:
		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* () { // /доступ к данным текущего элемента
			if (current == nullptr) {
				throw exception("Вышли за пределы дерева");
			}
			return current->value;
		}
		rIterator& operator ++() { // оператор инкремента итератора
			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);
		}
	}
	Node* findMaxNode(Node* node) const { // Ищем максимальный ключ в заданном узле или его детях
		if (node == nullptr) {
			throw exception("Узел не существует");
		}
		Node* buf = node;
		while (buf->rightPtr != nullptr) { // Итерируемся пока не найдем крайний правый узел
			buf = buf->rightPtr;
		}
		return buf;
	}
	Node* rParent(Node* node, const Node* nodeByKey) { // Ищем родителя заданного узла
		if (node == nodeByKey) { // 2. Нашли родителя, возвращаемся
			return nullptr;
		}
		else if (nodeByKey->key > node->key) {
			Node* rp = rParent(node->rightPtr, nodeByKey);
			if (rp != nullptr) {
				return rp;  // 3. Нашли родителя, возвращаемся
			}
			else {
				return node; // 1. Нашли родителя, возвращаемся
			}
		}
		else {
			return rParent(node->leftPtr, nodeByKey);
		}
	}
	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;
}
```

6.2 Menu.cpp

```
#include <iostream>
#include "BSTree.h"
```

```
using namespace std;
int main() {
	setlocale(LC_ALL, "Russian");
	BSTree<int, int> tree = BSTree<int, int>();
	auto it = tree.rend();
	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. Поиск по ключу" << endl;
		cout << "10. Обнулить итератор" << endl;
		cout << "11. Показать содержание итератора" << endl;
		cout << "12. Перевести итератор на следующий элемент" << endl;
		cout << "13. Присвоить итератору новое значение" << endl;
		cout << endl;
		cin >> n;
		cout << endl;
		try {
			switch (n) {
			case 1: {
				int key;
				int 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: {
				int key;
				cin >> key;
				cout << tree.find(key) << endl;
```

```
                        break;
                }
                case 10: {
                        it = tree.rbegin(); break;
                }
                case 11: {
                        cout << *it << endl; break;
                }
                case 12: {
                        ++it; break;
                }
                case 13: {
                        cin >> *it; break;
                }
                default: {
                        cout << "Попробуйте еще раз" << endl; break;
                }
                }
        }
        catch (exception e) {
                cout << e.what() << endl;
        }
    }
}
```