**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ**

Практическая работа №3

по дисциплине

«Алгоритмы и структуры данных»

на тему

«Коллекция данных – Сбалансированное дерево поиска»

Группа: АВТ-043

Факультет: АВТФ

Студент: Сивирилова М.К., Миллер В.В.

Преподаватель: Дыбко М.А.

Новосибирск 2022

**ОГЛАВЛЕНИЕ**

[1. Введение 3](#_Toc118635074)

[2. Основная часть 5](#_Toc118635075)

[Формат АТД «BST-дерево» 5](#_Toc118635076)

[Формат АТД «Прямой итератор списка» 10](#_Toc118635077)

[Формат АТД «Обратный итератор списка» 12](#_Toc118635078)

[Определение шаблонного класса для коллекции «Дерево поиска» 14](#_Toc118635079)

[Тестирование трудоемкости 17](#_Toc118635080)

[3. Заключение 21](#_Toc118635081)

1. Введение

**Цель работы:**

Изучение и исследование методов балансировки двоичных деревьев поиска на примере АТД «Сбалансированное дерево поиска». Освоение методики модификации коллекций с помощью механизма наследования классов.

**Задание:**

1. Спроектировать, реализовать и провести тестовые испытания АТД «Сбалансированное дерево поиска» для коллекции, содержащей данные произвольного типа. Тип данных задаётся клиентской программой. АТД «Сбалансированное дерево поиска» представляет собой модифицированную версию АТД «BST-дерево» с трудоёмкостью операций .

Интерфейс АТД «Сбалансированное дерево поиска» включает следующие операции:

* конструктор,
* конструктор копирования,
* деструктор,
* опрос размера дерева,
* очистка дерева,
* проверка дерева на пустоту,
* доступ по чтению/записи к данным по ключу,
* включение данных с заданным ключом,
* удаление данных с заданным ключом,
* запрос прямого итератора, установленного на узел дерева с минимальным ключом begin(),
* запрос обратного итератора, установленного на узел дерева с максимальным ключом rbegin(),
* запрос «неустановленного» прямого итератора end(),
* запрос «неустановленного» обратного итератора rend(),

Операции прямого и обратного итератора:

* операция доступа по чтению и записи к данным текущего узла \* ,
* операция перехода к следующему (для обратного – к предыдущему) по ключу узлу в дереве ++ ,
* операция перехода к предыдущему (для обратного – к следующему) по ключу узлу в дереве -- ,
* проверка равенства однотипных итераторов == ,
* проверка неравенства однотипных итераторов != .

Для тестирования коллекции интерфейс АТД «Сбалансированное дерево поиска» включает дополнительные операции:

* вывод структуры дерева на экран (для узлов отображать ключи и параметр для балансировки),
* опрос числа просмотренных предыдущей операцией узлов дерева.

2. Выполнить отладку и тестирование всех операций АТД «Сбалансированное дерево поиска» с помощью меню операций.

3. Выполнить сравнительное тестирование трудоёмкости операций вставки, удаления и поиска для коллекций «BST-дерево» и «Сбалансированное дерево поиска» для случайной и вырожденной структуры дерева.

4. Выполнить сравнительный анализ теоретических и экспериментальных показателей трудоёмкости операций.

1. Основная часть

Формат АТД «Сбалансированное 2-3 дерево поиска»

**АТД Tree**

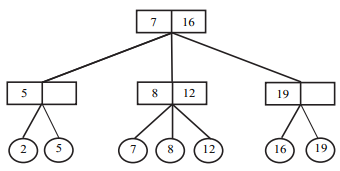
**Общая характеристика:**

Сбалансированным является такое дерево, высота которого логарифмически зависит от количества элементов в дереве n. К наиболее известным сбалансированным деревьям относятся рандомизированное дерево, AVL-дерево, красно-черное дерево (RB-дерево). Также известны 2-3-дерево и 2-3-4-дерево, которые являются идеально сбалансированными.

В сбалансированные деревья введён механизм локальной балансировки, который контролирует и выравнивает высоты поддеревьев в процессе вставок и удалений узлов.

2-3-дерево не является двоичным деревом. Его узлы могут иметь двух или трёх сыновей. Все пути от корня до любого листа имеют одинаковую длину. Данные с ключами располагаются только в листьях дерева по возрастанию ключей.

2-3-дерево (рис. 1) соответствует полному, идеально сбалансированному BST-дереву, если все его узлы являются 2-узлами. В этом случае его высота равна . Если же все узлы в 2-3-дереве – 3-узлы, то высота 2-3-дерева равна . Таким образом, длина пути от корня дерева к листьям с данными и, следовательно, трудоёмкость операций лежат в пределах – .



*Рисунок 1. Структура 2-3-дерева*

**Данные:**

***Параметры:***

***T*** – тип элементов, хранящихся в двоичном дереве.

***counter*** – размер 2-3 дерева.

***prevNodeCounter*** – счётчик использований узлов для подсчёта трудоёмкости.

***root*** – указатель на корень 2-3 дерева.

***Структура хранения коллекции:*** 2-3 дерево с элементами типа T.

**Операции:**

***Конструктор Tree()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** создание пустого 2-3 дерева

**Выход:** нет

**Постусловия:** создано пустое 2-3 дерево с числом элементов *counter* = 0 и с указателем на корень *root* = nullptr.

***Конструктор Tree(size)***

**Вход:** *size* – размер создаваемого 2-3 дерева

**Предусловия:** нет

**Процесс:** создание 2-3 дерева размером *size*, заполненного случайными значениями

**Выход:** нет

**Постусловия:** создано 2-3 дерево с числом элементов *size* со случайными значениями элементов.

***Конструктор копирования Tree(other)***

**Вход:** *other* – ссылка на копируемое 2-3 дерево

**Предусловия:** нет

**Процесс:** создание копии 2-3 дерева *other*

**Выход:** нет

**Постусловия:** создана копия 2-3 дерева с числом элементов *counter* = *other.counter* и счётчиком использований узлов *prevNodeCounter* = *other.prevNodeCounter*.

***Деструктор ~Tree()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** удаление всех элементов из 2-3 дерева

**Выход:** нет

**Постусловия:** 2-3 дерево удалено

***Копирование элементов copy(node)***

**Вход:** *node* – указатель на узел поддерева, который необходимо скопировать в текущее дерево

**Предусловия:** нет

**Процесс:** копирование узла и всех его последующих детей в текущее 2-3 дерево

**Выход:** нет

**Постусловия:** в текущее 2-3 дерево скопировались все элементы поддерева с узлом *node*.

***Полная очистка поддерева clear(node)***

**Вход:** *node* – указатель на узел поддерева, который необходимо очистить

**Предусловия:** нет

**Процесс:** полная очистка указанного поддерева.

**Выход:** нет

**Постусловия:** в текущем 2-3 дереве очищены элементы, начиная с указанного узла.

***Опрос размера бинарного дерева getCounter()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** возврат текущего количества элементов *counter,* хранящихся в 2-3 дереве

**Выход:** размер 2-3 дерева

**Постусловия:** нет

***Проверка на пустоту isEmpty()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** проверка 2-3 дерева на пустоту

**Выход:** наличие какого-либо элемента в массиве в виде булевой переменной

**Постусловия:** нет

***Опрос количества использований узлов getPrevNodeCounter()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** возврат текущего количества использований узлов *prevNodeCounter*

**Выход:** значение искомого количества использований узлов *prevNodeCounter*

**Постусловия:** нет

***Сброс количества использований узлов resetPrevNodeCounter()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** сброс текущего количества использований узлов *prevNodeCounter*

**Выход:** нет

**Постусловия:** текущее количество использований узлов *prevNodeCounter* сбросилось до нуля

***Форматированный вывод бинарного дерева print(root, level, x, y, c)***

**Вход:** *root* – корень дерева для вывода*, level* – уровень дерева (где корень – первый уровень, его дети – второй и т.д.)*, x* – координата курсора в строке консоли (отступ от начала в символах)*, y* – координата курсора в консоли (отступ от первой строки)*, с* – параметр, отвечающий за различие корня (с=0), левого ребёнка (с=1), правого ребёнка (с=2)

**Предусловия:** 1 *c* *3, a 0, root != nullptr*

**Процесс:** вывод дерева в консоль

**Выход:** нет

**Постусловия:** увеличение количества изменений узлов *prevNodeCounter*++

***Добавление значения по значению insert(p, key)***

**Вход:** *p* – указатель на узел дерева, в который произойдёт вставка, *key –* значение вставляемого элемента

**Предусловия:** *node == nullptr, counter == 0, !p, p && p - лист*

**Процесс:** вставка элемента *key* по заданному узлу *node*

**Выход:** всегда возвращаем корень дерева, т.к. он может меняться

**Постусловия:** увеличение размера дерева *counter++,* увеличение количества изменений узлов *prevNodeCounter*++

***Разделение вершины при переполнении split(elem)***

**Вход:** *elem* – указатель на текущий узел дерева

**Предусловия:** нет

**Процесс:** создаем две новые вершины, которые имеют такого же родителя, как и разделяющийся элемент, дальше происходит своеобразная сортировка ключей при разделении.

**Выход:** возврат родителя отправляемого узла *elem*, либо сам же этот элемент, если их кол-во равно трём

**Постусловия:** увеличение количества изменений узлов *prevNodeCounter*++

***Опрос наличия элемента с заданным значением search(p, key)***

**Вход:** *p* – указатель на узел дерева, в котором происходит поиск, *key –* значение искомого элемента

**Предусловия:** нет

**Процесс:** поиск элемента с заданным значением key в 2-3 дереве по заданному узлу *p*

**Выход:** указатель на узел искомого значения

**Постусловия:** увеличение количества изменений узлов *prevNodeCounter*++

***Поиск минимального элемента в поддереве searchMin(p)***

**Вход:** *p* – указатель на узел дерева, который является корнем и с которого начнётся поиск

**Предусловия:** нет

**Процесс:** поиск самого минимального значения дерева

**Выход:** указатель на узел с минимальным значением дерева

**Постусловия:** увеличение количества изменений узлов *prevNodeCounter*++

***Удаление элемента по значению remove(p, key)***

**Вход:** *p* – указатель на узел дерева, с которого начнётся удаление*, key* – значение удаляемого элемента

**Предусловия:** *node == nullptr*

**Процесс:** удаление элемента по заданному значению

**Выход:** указатель на узел, удаляемого значения

**Постусловия:** уменьшение размера дерева *counter--,* увеличение количества изменений узлов *prevNodeCounter*++

***Исправление дерева fix(leaf)***

**Вход:** *leaf* – указатель на листок дерева

**Предусловия:** нет

**Процесс:** Исправление дерева для того, чтобы соблюдались все свойства сбалансированного 2-3 дерева. Возможны 3 случая: когда единственный ключ в вершине; когда вершина, в которой удалили ключ, имела два ключа; когда достаточно перераспределить ключи в дереве; когда нужно произвести склеивание и пройтись вверх по дереву как минимум на ещё одну вершину

**Выход:** либо ничего, либо следующий листок

**Постусловия:** уменьшение размера дерева *counter--,* увеличение количества изменений узлов *prevNodeCounter*++

***Перераспределение элементов redistribute(leaf)***

**Вход:** *leaf* – указатель на листок дерева

**Предусловия:** нет

**Процесс:** при цикличном сдвиге ключей между родителем и сыновьями также нужно будет перемещать и внуков родителя. Перераспределять ключи можно из любого брата, но удобнее всего из ближнего, который имеет 2 ключа, при этом мы циклично сдвигаем все ключи

**Выход:** либо ничего, либо следующий листок

**Постусловия:** уменьшение размера дерева *counter--,* увеличение количества изменений узлов *prevNodeCounter*++

***Слияние элементов merge(leaf)***

**Вход:** *leaf* – указатель на листок дерева

**Предусловия:** нет

**Процесс:** перенос меньшего ключа родителя в поддерево с меньшими ключами либо большего ключа в поддерево с большими ключами, и удаление вершины без ключей.

**Выход:** либо ничего, либо следующий листок

**Постусловия:** уменьшение размера дерева *counter--,* увеличение количества изменений узлов *prevNodeCounter*++

**Конец АТД Tree**

Формат АТД «Прямой итератор списка»

**АТД Iterator**

**Общая характеристика**

Итератор представляет собой объект, реализующий методы доступа по чтению/записи элементов связанного с ним списка, дерева. Имеется возможность установки на начало и конец списка (дерева), перемещение к следующему и предыдущему элементу, сравнение итераторов между собой.

**Данные:**

***Параметры:***

*cur* – текущий индекс элемента

*tree* – указатель на дерево, с которым связан итератор

*counter* – счётчик элементов

**Операции:**

***Конструктор Iterator(tree)***

**Вход:** *tree* – указатель на дерево

**Предусловия:** нет

**Процесс:** создание итератора, связанного с *tree*

**Выход:** нет

**Постусловия:** создан итератор

***Установка в начало массива begin(node)***

**Вход:** *node* – указатель на проверяемый узел

**Предусловия:** нет

**Процесс:** установка итератора в начало дерева (самое маленькое/левое значение)

**Выход:** указатель на итератор

**Постусловия:** итератор установлен в начало дерева

***Установка в конец массива end(node)***

**Вход:** *node* – указатель на проверяемый узел

**Предусловия:** нет

**Процесс:** установка итератора в конец дерева

**Выход:** указатель на итератор

**Постусловия:** итератор установлен в конец дерева

***Переход к следующему значению operator++()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** переход на следующий элемент дерева или в начало, если достигнут конец массива

**Выход:** указатель на итератор

**Постусловия:** итератор установлен на следующий элемент дерева

***Переход к предыдущему значению operator--()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** переход на предыдущий элемент дерева или в конец, если достигнуто начало дерева

**Выход:** указатель на итератор

**Постусловия:** итератор установлен на предыдущий элемент дерева

***Доступ по чтению и записи operator\*()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** возврат значения *key* текущего элемента

**Выход:** значение *key* типа *Т*

**Постусловия:** нет

***Опрос равенства значений operator==(i)***

**Вход:** *i* – итератор, с которым происходит сравнение

**Предусловия:** нет

**Процесс:** сравнение равенства значений элементов, на которые указывают итераторы

**Выход:** подтверждение/опровержение равенства в виде булевой переменной

**Постусловия:** нет

***Опрос неравенства значений operator!=(i)***

**Вход:** *i* – итератор, с которым происходит сравнение

**Предусловия:** нет

**Процесс:** сравнение неравенства значений элементов, на которые указывают итераторы

**Выход:** подтверждение/опровержение неравенства в виде булевой переменной

**Постусловия:** нет

**Конец АТД Iterator**

Формат АТД «Обратный итератор списка»

**АТД ReverseIterator**

**Общая характеристика**

Обратный итератор представляет собой объект, реализующий методы доступа по чтению/записи элементов связанного с ним списка, дерева в обратном порядке. Имеется возможность установки на начало и конец списка (дерева), перемещение к следующему и предыдущему элементу, сравнение обратных итераторов между собой.

**Данные:**

***Параметры:***

*cur* – текущий индекс элемента

*tree* – указатель на дерево, с которым связан обратный итератор

*counter* – счётчик элементов

**Операции:**

***Конструктор ReverseIterator(tree)***

**Вход:** *tree* – указатель на дерево

**Предусловия:** нет

**Процесс:** создание обратного итератора, связанного с *tree*

**Выход:** нет

**Постусловия:** создан обратный итератор

***Установка в конец массива rbegin(node)***

**Вход:** *node* – указатель на проверяемый узел

**Предусловия:** нет

**Процесс:** установка обратного итератора в конец дерева

**Выход:** указатель на обратный итератор

**Постусловия:** обратный итератор установлен в конец дерева

***Установка в начало массива rend(node)***

**Вход:** *node* – указатель на проверяемый узел

**Предусловия:** нет

**Процесс:** установка обратного итератора в начало дерева

**Выход:** указатель на обратный итератор

**Постусловия:** обратный итератор установлен в начало дерева

***Переход к следующему значению operator++()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** переход на предыдущий элемент дерева или в конец, если достигнута начало дерева

**Выход:** указатель на обратный итератор

**Постусловия:** обратный итератор установлен на предыдущий элемент дерева

***Переход к предыдущему значению operator--()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** переход на следующий элемент дерева или в начало, если достигнут конец дерева

**Выход:** указатель на обратный итератор

**Постусловия:** обратный итератор установлен на следующий элемент дерева

***Доступ по чтению и записи operator\*()***

**Вход:** нет

**Предусловия:** нет

**Процесс:** возврат значения *key* текущего элемента

**Выход:** значение *key* типа *Т*

**Постусловия:** нет

***Опрос равенства значений operator==(i)***

**Вход:** *i* – обратный итератор, с которым происходит сравнение

**Предусловия:** нет

**Процесс:** сравнение равенства значений элементов, на которые указывают обратные итераторы

**Выход:** подтверждение/опровержение равенства в виде булевой переменной

**Постусловия:** нет

***Опрос неравенства значений operator!=(i)***

**Вход:** *i* – обратный итератор, с которым происходит сравнение

**Предусловия:** нет

**Процесс:** сравнение неравенства значений элементов, на которые указывают обратные итераторы

**Выход:** подтверждение/опровержение неравенства в виде булевой переменной

**Постусловия:** нет

**Конец АТД ReverseIterator**

Определение шаблонного класса для коллекции «Сбалансированное 2-3 дерево поиска»

template <typename T>

class Tree {

public:

Tree();

Tree(int size);

Tree(const Tree<T>& other);

~Tree();

class Node {

public:

T keys[3];

Node\* first;

Node\* second;

Node\* third;

Node\* fourth;

Node\* parent;

int size; // Кол-во занятых ключей

bool contain(T key) { // Этот метод возвращает true, если ключ key находится в вершине, иначе false.

for (int i = 0; i < size; ++i)

if (keys[i] == key) return true;

return false;

}

void swap(T& x, T& y) {

T r = x;

x = y;

y = r;

}

void sort2(T& x, T& y) {

if (x > y) swap(x, y);

}

void sort3(T& x, T& y, T& z) {

if (x > y) swap(x, y);

if (x > z) swap(x, z);

if (y > z) swap(y, z);

}

void sort() { // Ключи в вершинах должны быть отсортированы

if (size == 1) return;

if (size == 2) sort2(keys[0], keys[1]);

if (size == 3) sort3(keys[0], keys[1], keys[2]);

}

void insertToNode(T key) { // Вставляем ключ key в вершину (не в дерево)

keys[size] = key;

size++;

sort();

}

void removeFromNode(T key) { // Удаляем ключ key из вершины (не из дерева)

if (size >= 1 && keys[0] == key) {

keys[0] = keys[1];

keys[1] = keys[2];

size--;

}

else if (size == 2 && keys[1] == key) {

keys[1] = keys[2];

size--;

}

}

void becomeNode2(T key, Node\* f, Node\* s) { // Преобразовать в 2-вершину.

keys[0] = key;

first = f;

second = s;

third = nullptr;

fourth = nullptr;

parent = nullptr;

size = 1;

}

bool isLeaf() { // Является ли узел листом; проверка используется при вставке и удалении.

return (first == nullptr) && (second == nullptr) && (third == nullptr);

}

};

// Вспомогательные функции

Node\* getRoot() { // Опрос корня дерева

return root;

};

int getCounter() {

return counter;

}

int getPrevNodeCounter() {

return prevNodeCounter;

}

void resetPrevNodeCounter() {

prevNodeCounter = 0;

}

bool isEmpty() {

return counter == 0;

}

Tree<T>& operator=(const Tree<T>& other);

// Основные функции

void print(Node\* root, int level = 0, int x = 50, int y = 0, int c = 0) { // c = 0 - корень, c = 1 - first, c = 2 - second, с = 3 - third

if (root) {

if (level == 0) system("cls");

if (c != 0) {

if (c == 1) x -= 40 - level \* 15;

else if (c == 3) x += 30 - level \* 10;

}

GoToXY(x, y++);

if (c == 1) printf("/");

if (c == 2) printf("|");

if (c == 3) printf("\\");

if (c == 1) GoToXY(x - 2, y++);

else GoToXY(x, y++);

printf("%d", root->keys[0]);

if (root->size == 2) printf(" %d", root->keys[1]);

prevNodeCounter++;

level++;

if (root->first) print(root->first, level, x, y, 1);

if (root->second) print(root->second, level, x, y, 2);

if (root->third) print(root->third, level, x, y, 3);

}

}

// Вставка в дерево

Node\* insert(Node\* p, T key) { // вставка ключа key в дерево с корнем p; всегда возвращаем корень дерева, т.к. он может меняться

prevNodeCounter++;

if (counter == 0) {

root = (Node\*)malloc(sizeof(Node)); // если дерево пусто, то создаем первую 2-3-вершину (корень)

root->keys[0] = key;

root->first = nullptr;

root->second = nullptr;

root->third = nullptr;

root->fourth = nullptr;

root->parent = nullptr;

root->size = 1;

counter++;

return root;

}

else if (!p) {

Node\* newNode = (Node\*)malloc(sizeof(Node)); // создаем новую вершину

newNode->keys[0] = key;

newNode->first = nullptr;

newNode->second = nullptr;

newNode->third = nullptr;

newNode->fourth = nullptr;

newNode->parent = nullptr;

newNode->size = 1;

counter++;

return newNode;

}

if (p && p->isLeaf()) {

p->insertToNode(key);

counter++;

}

else if (key <= p->keys[0]) insert(p->first, key);

else if ((p->size == 1) || ((p->size == 2) && key <= p->keys[1])) insert(p->second, key);

else insert(p->third, key);

return split(p);

}

// Метод для разделение вершины при переполнении

Node\* split(Node\* elem) {

if (elem->size < 3) return elem;

// Создаем две новые вершины, которые имеют такого же родителя, как и разделяющийся элемент.

Node\* x = (Node\*)malloc(sizeof(Node));

x->keys[0] = elem->keys[0];

x->first = elem->first;

x->second = elem->second;

x->third = nullptr;

x->fourth = nullptr;

x->parent = elem->parent;

x->size = 1;

prevNodeCounter++;

Node\* y = (Node\*)malloc(sizeof(Node));

y->keys[0] = elem->keys[2];

y->first = elem->third;

y->second = elem->fourth;

y->third = nullptr;

y->fourth = nullptr;

y->parent = elem->parent;

y->size = 1;

prevNodeCounter++;

if (x->first) x->first->parent = x; // Правильно устанавливаем "родителя" "сыновей".

if (x->second) x->second->parent = x; // После разделения, "родителем" "сыновей" является "дедушка",

if (y->first) y->first->parent = y; // Поэтому нужно правильно установить указатели.

if (y->second) y->second->parent = y;

prevNodeCounter++;

if (elem->parent) {

elem->parent->insertToNode(elem->keys[1]);

if (elem->parent->first == elem) elem->parent->first = nullptr;

else if (elem->parent->second == elem) elem->parent->second = nullptr;

else if (elem->parent->third == elem) elem->parent->third = nullptr;

// Дальше происходит своеобразная сортировка ключей при разделении.

if (elem->parent->first == nullptr) {

elem->parent->fourth = elem->parent->third;

elem->parent->third = elem->parent->second;

elem->parent->second = y;

elem->parent->first = x;

}

else if (elem->parent->second == nullptr) {

elem->parent->fourth = elem->parent->third;

elem->parent->third = y;

elem->parent->second = x;

}

else {

elem->parent->fourth = y;

elem->parent->third = x;

}

Node\* temp = elem->parent;

free(elem);

return temp;

}

else {

x->parent = elem; // Так как в эту ветку попадает только корень,

y->parent = elem; // то мы "родителем" новых вершин делаем разделяющийся элемент.

elem->becomeNode2(elem->keys[1], x, y);

return elem;

}

}

// Поиск в дереве

Node\* search(Node\* p, T key) { // Поиск ключа key в 2-3 дереве с корнем p.

prevNodeCounter++;

if (!p) return nullptr;

if (p->contain(key)) return p;

else if (key < p->keys[0]) return search(p->first, key);

else if ((p->size == 2) && (key < p->keys[1]) || (p->size == 1)) return search(p->second, key);

else if (p->size == 2) return search(p->third, key);

}

// Поиск минимального элемента в поддереве

Node\* searchMin(Node\* p) { // Поиск узла с минимальным элементов в 2-3-дереве с корнем p.

prevNodeCounter++;

if (!p) return p;

if (!(p->first)) return p;

else return searchMin(p->first);

}

// Удаление из дерева

Node\* remove(Node\* p, T key) { // Удаление ключа key в 2-3-дереве с корнем p.

Node\* elem = search(p, key); // Ищем узел, где находится ключ key

prevNodeCounter++;

if (!elem) return p;

counter--;

Node\* min = nullptr;

if (elem->keys[0] == key) min = searchMin(elem->second); // Ищем эквивалентный ключ

else min = searchMin(elem->third);

if (min) { // Меняем ключи местами

T& z = (key == elem->keys[0] ? elem->keys[0] : elem->keys[1]);

elem->swap(z, min->keys[0]);

elem = min; // Перемещаем указатель на лист, т.к. min - всегда лист

}

elem->removeFromNode(key); // И удаляем требуемый ключ из листа

return fix(elem); // Вызываем функцию для восстановления свойств дерева.

}

// Используется после удаления для возвращения свойств дереву (использует merge или redistribute)

Node\* fix(Node\* leaf) {

prevNodeCounter++;

if (leaf->size == 0 && leaf->parent == nullptr) { // Случай 0, когда удаляем единственный ключ в вершине

free(leaf);

return nullptr;

}

if (leaf->size != 0) { // Случай 1, когда вершина, в которой удалили ключ, имела два ключа

if (leaf->parent) return fix(leaf->parent);

else return leaf;

}

Node\* parent = leaf->parent;

if (parent->first->size == 2 || parent->second->size == 2 || parent->size == 2) leaf = redistribute(leaf); // Случай 2, когда достаточно перераспределить ключи в дереве

else if (parent->size == 2 && parent->third->size == 2) leaf = redistribute(leaf); // Аналогично

else leaf = merge(leaf); // Случай 3, когда нужно произвести склеивание и пройтись вверх по дереву как минимум на еще одну вершину

return fix(leaf);

}

// Перераспределение также используется при удалении

Node\* redistribute(Node\* leaf) {

prevNodeCounter++;

Node\* parent = leaf->parent;

Node\* first = parent->first;

Node\* second = parent->second;

Node\* third = parent->third;

if ((parent->size == 2) && (first->size < 2) && (second->size < 2) && (third->size < 2)) {

if (first == leaf) {

parent->first = parent->second;

parent->second = parent->third;

parent->third = nullptr;

parent->first->insertToNode(parent->keys[0]);

parent->first->third = parent->first->second;

parent->first->second = parent->first->first;

if (leaf->first != nullptr) parent->first->first = leaf->first;

else if (leaf->second != nullptr) parent->first->first = leaf->second;

if (parent->first->first != nullptr) parent->first->first->parent = parent->first;

parent->removeFromNode(parent->keys[0]);

free(first);

}

else if (second == leaf) {

first->insertToNode(parent->keys[0]);

parent->removeFromNode(parent->keys[0]);

if (leaf->first != nullptr) first->third = leaf->first;

else if (leaf->second != nullptr) first->third = leaf->second;

if (first->third != nullptr) first->third->parent = first;

parent->second = parent->third;

parent->third = nullptr;

free(second);

}

else if (third == leaf) {

second->insertToNode(parent->keys[1]);

parent->third = nullptr;

parent->removeFromNode(parent->keys[1]);

if (leaf->first != nullptr) second->third = leaf->first;

else if (leaf->second != nullptr) second->third = leaf->second;

if (second->third != nullptr) second->third->parent = second;

free(third);

}

}

else if ((parent->size == 2) && ((first->size == 2) || (second->size == 2) || (third->size == 2))) {

if (third == leaf) {

if (leaf->first != nullptr) {

leaf->second = leaf->first;

leaf->first = nullptr;

}

leaf->insertToNode(parent->keys[1]);

if (second->size == 2) {

parent->keys[1] = second->keys[1];

second->removeFromNode(second->keys[1]);

leaf->first = second->third;

second->third = nullptr;

if (leaf->first != nullptr) leaf->first->parent = leaf;

}

else if (first->size == 2) {

parent->keys[1] = second->keys[0];

leaf->first = second->second;

second->second = second->first;

if (leaf->first != nullptr) leaf->first->parent = leaf;

second->keys[0] = parent->keys[0];

parent->keys[0] = first->keys[1];

first->removeFromNode(first->keys[1]);

second->first = first->third;

if (second->first != nullptr) second->first->parent = second;

first->third = nullptr;

}

}

else if (second == leaf) {

if (third->size == 2) {

if (leaf->first == nullptr) {

leaf->first = leaf->second;

leaf->second = nullptr;

}

second->insertToNode(parent->keys[1]);

parent->keys[1] = third->keys[0];

third->removeFromNode(third->keys[0]);

second->second = third->first;

if (second->second != nullptr) second->second->parent = second;

third->first = third->second;

third->second = third->third;

third->third = nullptr;

}

else if (first->size == 2) {

if (leaf->second == nullptr) {

leaf->second = leaf->first;

leaf->first = nullptr;

}

second->insertToNode(parent->keys[0]);

parent->keys[0] = first->keys[1];

first->removeFromNode(first->keys[1]);

second->first = first->third;

if (second->first != nullptr) second->first->parent = second;

first->third = nullptr;

}

}

else if (first == leaf) {

if (leaf->first == nullptr) {

leaf->first = leaf->second;

leaf->second = nullptr;

}

first->insertToNode(parent->keys[0]);

if (second->size == 2) {

parent->keys[0] = second->keys[0];

second->removeFromNode(second->keys[0]);

first->second = second->first;

if (first->second != nullptr) first->second->parent = first;

second->first = second->second;

second->second = second->third;

second->third = nullptr;

}

else if (third->size == 2) {

parent->keys[0] = second->keys[0];

second->keys[0] = parent->keys[1];

parent->keys[1] = third->keys[0];

third->removeFromNode(third->keys[0]);

first->second = second->first;

if (first->second != nullptr) first->second->parent = first;

second->first = second->second;

second->second = third->first;

if (second->second != nullptr) second->second->parent = second;

third->first = third->second;

third->second = third->third;

third->third = nullptr;

}

}

}

else if (parent->size == 1) {

leaf->insertToNode(parent->keys[0]);

if (first == leaf && second->size == 2) {

parent->keys[0] = second->keys[0];

second->removeFromNode(second->keys[0]);

if (leaf->first == nullptr) leaf->first = leaf->second;

leaf->second = second->first;

second->first = second->second;

second->second = second->third;

second->third = nullptr;

if (leaf->second != nullptr) leaf->second->parent = leaf;

}

else if (second == leaf && first->size == 2) {

parent->keys[0] = first->keys[1];

first->removeFromNode(first->keys[1]);

if (leaf->second == nullptr) leaf->second = leaf->first;

leaf->first = first->third;

first->third = nullptr;

if (leaf->first != nullptr) leaf->first->parent = leaf;

}

}

return parent;

}

// Слияние используется при удалении

Node\* merge(Node\* leaf) {

prevNodeCounter++;

Node\* parent = leaf->parent;

if (parent->first == leaf) {

parent->second->insertToNode(parent->keys[0]);

parent->second->third = parent->second->second;

parent->second->second = parent->second->first;

if (leaf->first != nullptr) parent->second->first = leaf->first;

else if (leaf->second != nullptr) parent->second->first = leaf->second;

if (parent->second->first != nullptr) parent->second->first->parent = parent->second;

parent->removeFromNode(parent->keys[0]);

free(parent->first);

parent->first = nullptr;

}

else if (parent->second == leaf) {

parent->first->insertToNode(parent->keys[0]);

if (leaf->first != nullptr) parent->first->third = leaf->first;

else if (leaf->second != nullptr) parent->first->third = leaf->second;

if (parent->first->third != nullptr) parent->first->third->parent = parent->first;

parent->removeFromNode(parent->keys[0]);

free(parent->second);

parent->second = nullptr;

}

if (parent->parent == nullptr) {

Node\* temp = nullptr;

if (parent->first != nullptr) temp = parent->first;

else temp = parent->second;

temp->parent = nullptr;

free(parent);

return temp;

}

return parent;

}

void clear(Node\* node) {

if (node == root) root = nullptr;

if (node) {

prevNodeCounter++;

clear(node->first);с

clear(node->second);

clear(node->third);

counter--;

if (node->size == 2) counter--;

free(node);

node = nullptr;

}

//if (counter == 0) root = nullptr;

}

void copy(Node\* node) {

if (node) {

prevNodeCounter++;

insert(this->root, node->keys[0]);

if (node->size == 2) insert(this->root, node->keys[1]);

if (node->first) copy(node->first);

if (node->second) copy(node->second);

if (node->third) copy(node->third);

}

}

class Iterator // внутренний класс - итератор

{

private:

Node\* cur = nullptr; // текущий узел

T key;

Tree<T>\* tree;

int counter = 1;

public:

Iterator(Tree\* tree) { // конструктор итератора

this->tree = tree;

};

Iterator begin(Node\* node) { // установка на первое значение в массиве

if (node->first)

begin(node->first);

else {

cur = node;

key = node->keys[0];

}

counter = 1;

return \*this;

}

Iterator end(Node\* node) { // установка на последнее значение в массиве

if (node->third)

end(node->third);

else if (node->second)

end(node->second);

else {

cur = node;

if (node->size == 2) key = node->keys[1];

else key = node->keys[0];

}

counter = tree->counter;

return \*this;

}

Iterator& operator++(int) { // перегрузка перемещения вперед

if (tree->counter != counter) {

if (cur->isLeaf()) {

if (cur == cur->parent->first) {

if (cur->size == 1 || key == cur->keys[1]) {

cur = cur->parent;

key = cur->keys[0];

return \*this;

}

}

if (cur->size == 2 && key == cur->keys[0]) {

key = cur->keys[1];

}

else {

if ((cur == cur->parent->second && !cur->parent->third) || cur == cur->parent->third) {

while (cur->parent) cur = cur->parent;

if (key > cur->keys[0] && cur->size == 2)

key = cur->keys[1];

else

key = cur->keys[0];

}

else if (cur == cur->parent->second && cur->parent->third) {

cur = cur->parent;

key = cur->keys[1];

}

}

}

else {

if (key == cur->keys[1] && cur->size == 2) {

cur = cur->third;

}

else {

cur = cur->second;

}

while (cur->first) cur = cur->first;

key = cur->keys[0];

}

counter++;

}

else

begin(tree->root);

return \*this;

};

Iterator& operator--(int) { // перегрузка перемещения назад

if (tree->counter > 0) {

if (cur->isLeaf()) {

if (cur == cur->parent->third || cur == cur->parent->second) {

if (cur->size == 1 || key == cur->keys[0]) {

if (cur->parent->size == 2 && cur == cur->parent->third) {

cur = cur->parent;

key = cur->keys[1];

}

else {

cur = cur->parent;

key = cur->keys[0];

}

return \*this;

}

}

if (cur->size == 2 && key == cur->keys[1]) {

key = cur->keys[0];

}

else {

if ((cur == cur->parent->second && !cur->parent->third) || cur == cur->parent->third) {

cur = cur->parent;

if (key > cur->keys[0])

key = cur->keys[1];

else

key = cur->keys[0];

}

else if (cur == cur->parent->first) {

while (cur->parent) cur = cur->parent;

if (key < cur->keys[1]) key = cur->keys[0];

else key = cur->keys[1];

}

}

}

else {

if (key == cur->keys[1] && cur->size == 2) {

cur = cur->second;

}

else {

cur = cur->first;

}

while (cur->third) cur = cur->third;

while (cur->second) cur = cur->second;

if (cur->size == 2) key = cur->keys[1];

else key = cur->keys[0];

}

counter--;

}

else

end(tree->root);

return \*this;

};

T& operator\*() { // перегрузка доступа по чтению/записи

return key;

}

bool operator==(const Iterator&); // перегрузка равенства

bool operator!=(const Iterator&); // перегрузка неравенства

};

class ReverseIterator // внутренний класс - итератор

{

private:

Node\* cur = nullptr; // текущий индекс

Tree<T>\* tree;

T key;

int counter = 1;

public:

ReverseIterator(Tree\* tree) { // конструктор итератора

this->tree = tree;

};

ReverseIterator rbegin(Node\* node) { // установка на последнее значение в массиве

if (node->third)

rbegin(node->third);

else if (node->second)

rbegin(node->second);

else {

cur = node;

if (node->size == 2) key = node->keys[1];

else key = node->keys[0];

}

counter = tree->counter;

return \*this;

}

ReverseIterator rend(Node\* node) { // установка на первое значение в массиве

if (node->first)

rend(node->first);

else {

cur = node;

key = node->keys[0];

}

counter = 1;

return \*this;

}

ReverseIterator& operator--(int) { // перегрузка перемещения назад

if (tree->counter != counter) {

if (cur->isLeaf()) {

if (cur == cur->parent->first) {

if (cur->size == 1 || key == cur->keys[1]) {

cur = cur->parent;

key = cur->keys[0];

return \*this;

}

}

if (cur->size == 2 && key == cur->keys[0]) {

key = cur->keys[1];

}

else {

if ((cur == cur->parent->second && !cur->parent->third) || cur == cur->parent->third) {

while(cur->parent) cur = cur->parent;

if (key > cur->keys[0])

key = cur->keys[1];

else

key = cur->keys[0];

}

else if (cur == cur->parent->second && cur->parent->third) {

cur = cur->parent;

key = cur->keys[1];

}

}

}

else {

if (key == cur->keys[1] && cur->size == 2) {

cur = cur->third;

}

else {

cur = cur->second;

}

while (cur->first) cur = cur->first;

key = cur->keys[0];

}

counter++;

}

else

rend(tree->root);

return \*this;

};

ReverseIterator& operator++(int) { // перегрузка перемещения вперёд

if (tree->counter > 0) {

if (cur->isLeaf()) {

if (cur == cur->parent->third || cur == cur->parent->second) {

if (cur->size == 1 || key == cur->keys[0]) {

if (cur->parent->size == 2 && cur == cur->parent->third) {

cur = cur->parent;

key = cur->keys[1];

}

else {

cur = cur->parent;

key = cur->keys[0];

}

return \*this;

}

}

if (cur->size == 2 && key == cur->keys[1]) {

key = cur->keys[0];

}

else {

if ((cur == cur->parent->second && !cur->parent->third) || cur == cur->parent->third) {

cur = cur->parent;

if (key > cur->keys[0])

key = cur->keys[1];

else

key = cur->keys[0];

}

else if (cur == cur->parent->first) {

while (cur->parent) cur = cur->parent;

if (key < cur->keys[1]) key = cur->keys[0];

else key = cur->keys[1];

}

}

}

else {

if (key == cur->keys[1] && cur->size == 2) {

cur = cur->second;

}

else {

cur = cur->first;

}

while (cur->third) cur = cur->third;

while (cur->second) cur = cur->second;

if (cur->size == 2) key = cur->keys[1];

else key = cur->keys[0];

}

counter--;

}

else

rbegin(tree->root);

return \*this;

};

T& operator\*() { // перегрузка доступа по чтению/записи

return key;

}

bool operator==(const ReverseIterator&); // перегрузка равенства

bool operator!=(const ReverseIterator&); // перегрузка неравенства

};

protected:

Node\* root = nullptr; // Корень дерева

int counter = 0;

int prevNodeCounter = 0;

};

template <typename T>

Tree<T>::Tree() {

root = nullptr;

counter = 0;

}

template <typename T>

Tree<T>::Tree(int size) {

srand(time(NULL));

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

T key = rand() % 100 - 50;

Node\* temp = search(root, key);

while (temp && (temp->keys[0] == key || temp->keys[1] == key)) {

key = rand() % 100 - 50;

}

root = insert(root, key);

}

}

template<typename T>

Tree<T>::Tree(const Tree<T>& other) {

copy(other.root);

prevNodeCounter = other.prevNodeCounter;

//counter = other.counter;

}

template <typename T>

Tree<T>::~Tree() {

clear(getRoot());

}

template<typename T>

Tree<T>& Tree<T>::operator=(const Tree<T>& other) {

clear(root);

copy(other.root);

prevNodeCounter = other.prevNodeCounter;

//counter = other.counter;

return \*this;

}

template<typename T>

bool Tree<T>::Iterator::operator==(const Iterator& i) {

return this.cur == i.cur;

}

template<typename T>

bool Tree<T>::Iterator::operator!=(const Iterator& i) {

return this.cur != i.cur;

}

template<typename T>

bool Tree<T>::ReverseIterator::operator==(const ReverseIterator& i) {

return this.cur == i.cur;

}

template<typename T>

bool Tree<T>::ReverseIterator::operator!=(const ReverseIterator& i) {

return this.cur != i.cur;

}

Тестирование трудоемкости

Способ тестирования:

В дереве N элементов, программа выполняет операции вставки, удаления, поиска, запоминает количество проходов по узлам дерева и считает среднюю арифметическую трудоёмкость для вырожденного дерева и логарифмическую трудоёмкость для случайного дерева. Индекс выбирается случайным образом в диапазоне [0; N]. Взяты значения N, равные 100, 500, 1000, 1500, 3000.

Теоретическая трудоемкость данных операций для случайного дерева равна О() - О(), а для вырожденного дерева O().

Графики для дерева поиска и 2-3 дерева:

*Рисунок 2. График трудоёмкости для случайного дерева поиска и случайного сбалансированного 2-3 дерева поиска при операции вставки*

*Рисунок 3. График трудоёмкости для вырожденного дерева поиска и вырожденного сбалансированного 2-3 дерева поиска при операции вставки*

*Рисунок 4. График трудоёмкости для случайного дерева поиска и случайного сбалансированного 2-3 дерева поиска при операции удаления по значению*

*Рисунок 5. График трудоёмкости для вырожденного дерева поиска и вырожденного сбалансированного 2-3 дерева поиска при операции удаления по значению*

*Рисунок 6. График трудоёмкости для случайного дерева поиска и случайного сбалансированного 2-3 дерева поиска при операции поиска по значению*

*Рисунок 7. График трудоёмкости для вырожденного дерева поиска и вырожденного сбалансированного 2-3 дерева поиска при операции поиска по значению*

Как видно по рисункам 2-7, экспериментальная трудоемкость в основном меньше, чем теоретическая. Это объясняется тем, что взятая теоретическая трудоемкость справедлива для крайнего худшего случая.

1. Заключение

В ходе работы был спроектирован и реализован АТД «Сбалансированное дерево поиска» для сбалансированного 2-3 дерева поиска, содержащего данные произвольного типа, а также АТД «Прямой итератор» и АТД «Обратный итератор».

Для отладки и тестирования всех операций АТД «Сбалансированное дерево поиска» и итераторов создано меню операций.

Экспериментально получена трудоемкость выполнения операций по вставке, удалению и поиску элемента в бинарном дереве. Произведено сравнение экспериментальной и теоретической трудоемкости.

Все задания выполнены. Цель работы достигнута.