Пермский филиал федерального государственного автономного образовательного учреждения высшего образования  
«Национальный исследовательский университет   
Высшая школа экономики»

*Факультет социально-экономических и компьютерных наук*

Мусихин Данил Михайлович

**Управление памятью**

*Лабораторная работа № 2*

студента образовательной программы «Разработка информационных систем для бизнеса» по направлению подготовки *09.03.04 Программная инженерия*

Руководитель, к.т.н., Доцент кафедры ИТБ.

Л.Н. Лядова

Пермь, 2023 год

**Оглавление**

[1 Постановка задачи 3](#_Toc150020129)

[2 Анализ 4](#_Toc150020130)

[3 Проектирование 5](#_Toc150020131)

[4 Код программы 6](#_Toc150020132)

[5 Тестирование 7](#_Toc150020133)

# Постановка задачи

## Задание 1

Разработайте приложение для создания линейных списков, позволяющее строить списки с различной организацией и выполнять операции над ними:

Создание однонаправленного списка, с дисциплиной LIFO – с вставкой нового элемента в начало вписка.

Создание однонаправленного списка, с дисциплиной FIFO – с вставкой нового элемента в конец вписка.

Создание однонаправленного списка, с сортировкой элементов по ключу: новый элемент вставляется в соответствии с правилом сортировки списка по возрастанию (неубыванию) значений ключевых полей элементов.

Создание двунаправленного списка, с сортировкой элементов по ключу: новый элемент вставляется в соответствии с правилом сортировки списка по возрастанию (неубыванию) значений ключевых полей элементов.

Создание линейного списка, с сортировкой элементов по нескольким (как минимум, по двум) ключам: в определении типа – структуры элементов задаётся несколько полей (атрибутов); новый элемент вставляется в соответствии с правилом сортировки списка по возрастанию (неубыванию) значений двух ключевых полей (для сортировки используется две ссылки, определяющие связи между элементами, установленные в нужном порядке, по каждому ключу). Например: выполнить сортировку списка студентов по ФИО (по алфавиту) и по рейтингу (по невозрастанию).

Для каждого типа списков должны быть реализованы операции

1. Вставки нового элемента в список в соответствии с заданной (см. выше) дисциплиной;

Просмотра списка в двух направлениях (от головы (первого элемента) к концу и с последнего

Элемента до первого элемента списка – до головы) с выводом всех элементов;

Поиска элементов с заданным значением и подсчёта их количества в списке;

Удаления из списка элемента с заданным значением (или всех элементов с заданным значением, если их несколько в списке.

## Задание 2

Разработайте приложение для построения бинарных деревьев, позволяющее строить деревья и выполнять операции над ними:

1. Вставки нового элемента в бинарное дерево;

Вывода всех значений, записанных во все элементы (вершины) бинарного дерева в заданном порядке:

* 1. В порядке возрастания значений,
  2. В порядке убывания значений;

Поиска элемента (вершины, содержащей заданное значение;

Удаления из дерева элемента с заданным значением;

Уничтожения дерева – удаления всех вершин.

# Анализ

Для выполнения анализа задачи предлагается разбить ее на несколько компонентов, такие как структуры данных, операции и алгоритмы, которые могут быть использованы для выполнения каждой из подзадач. В данном случае нам нужно создать два разных приложения: одно для линейных списков и другое для бинарных деревьев.

## Разработка приложения для линейных списков

**Структура данных:**

1. Однонаправленный список с дисциплиной LIFO (Last-In-First-Out).

Однонаправленный список с дисциплиной FIFO (First-In-First-Out).

Однонаправленный список с сортировкой элементов по ключу.

Двунаправленный список с сортировкой элементов по ключу.

Линейный список с сортировкой элементов по нескольким ключам.

**Операции:**

1. Вставка нового элемента в список в соответствии с заданной дисциплиной.

Просмотр списка в двух направлениях.

Поиск элементов с заданным значением и подсчет их количества в списке.

Удаление из списка элемента с заданным значением (или всех элементов с заданным значением, если их несколько в списке).

**Алгоритмы:**

1. Для сортировки элементов по ключу можно использовать алгоритм вставки или сортировку слиянием.

Для поиска элементов и удаления из списка можно использовать линейный поиск или более эффективные алгоритмы, такие как бинарный поиск.

## Задание 2: Разработка приложения для бинарных деревьев

**Структура данных:**

1. Бинарное дерево.

**Операции:**

1. Вставка нового элемента в бинарное дерево.

Вывод всех значений в заданном порядке (возрастание и убывание).

Поиск элемента в дереве.

Удаление элемента из дерева.

Уничтожение дерева (удаление всех вершин).

**Алгоритмы:**

1. Для вставки элемента в бинарное дерево можно использовать алгоритмы вставки в бинарное дерево поиска (BST) или AVL-дерево, если необходимо балансировать дерево.

Для вывода значений в заданном порядке можно использовать обход дерева в инфиксном порядке (для возрастания) или обратном инфиксном порядке (для убывания).

Для поиска элемента и его удаления можно использовать рекурсивные алгоритмы поиска и удаления в бинарном дереве.

Для уничтожения дерева можно выполнить обход дерева и удаление каждой вершины.

# Проектирование

## Задание 1: Разработка приложения для линейных списков

**Структура данных:**

1. Однонаправленный список:
   1. Содержит ссылку на первый элемент (голову).
   2. Каждый элемент хранит значение и ссылку на следующий элемент.

Двунаправленный список:

* 1. Содержит ссылки на первый и последний элементы (голову и хвост).
  2. Каждый элемент хранит значение и ссылки на следующий и предыдущий элементы.

**Операции:**

1. Вставка нового элемента:
   1. Для LIFO и FIFO добавление элементов в начало и конец списка соответственно.
   2. Для сортированных списков вставка элементов в соответствии с правилами сортировки.

Просмотр списка:

* 1. Просмотр списка в двух направлениях, от головы к хвосту и от хвоста к голове.

Поиск элементов:

* 1. Поиск элементов с заданным значением в списке.

Удаление элементов:

* 1. Удаление элемента с заданным значением или всех элементов с заданным значением.

**Алгоритмы:**

1. Для сортировки элементов в сортированных списках можно использовать алгоритм сортировки вставкой или сортировку слиянием, в зависимости от требований к производительности.

Для поиска элементов можно использовать линейный поиск или, при наличии сортировки, бинарный поиск.

## Задание 2: Разработка приложения для бинарных деревьев

**Структура данных:**

1. Бинарное дерево:
   1. Каждый узел содержит значение, ссылки на левого и правого потомка.

**Операции:**

1. Вставка нового элемента:
   1. Реализовать функцию вставки, сравнивая значение с корнем и перемещаясь влево или вправо в зависимости от результата сравнения.

Вывод значений:

* 1. Вывод значений в заданном порядке (возрастание и убывание), используя обход дерева.

Поиск элемента:

* 1. Поиск элемента, начиная с корня и перемещаясь по дереву влево или вправо.

Удаление элемента:

* 1. Реализовать функцию удаления элемента. Это может потребовать учета разных случаев, включая случай, когда у узла есть два потомка.

Уничтожение дерева:

* 1. Рекурсивно удалять все узлы, начиная с корня.

**Алгоритмы:**

1. Для вставки элемента в бинарное дерево можно использовать рекурсивный алгоритм, который сравнивает значение с корнем и определяет, в какую сторону двигаться.

Для вывода значений можно использовать рекурсивные обходы дерева в инфиксном и обратном инфиксном порядке.

Для поиска элемента можно использовать аналогично рекурсивный алгоритм.

Для удаления элемента можно рассмотреть разные случаи, включая случай, когда у узла есть два потомка. Тут также может потребоваться рекурсивный подход.

**Балансировка дерева:**

1. Понятие высоты дерева:
   1. Высота дерева - это длина самого длинного пути от корня до одного из его листьев.

Рассмотрите разницу в высоте (баланс-фактор) между левым и правым поддеревьями каждого узла. Баланс-фактор узла равен разнице высоты его левого и правого поддеревьев.

* 1. Если баланс-фактор узла больше 1 (что означает, что левое поддерево выше), или меньше -1 (что означает, что правое поддерево выше), выполните вращение, чтобы восстановить баланс. Существует несколько типов вращений, включая:
  2. Левое вращение (LL): Используется, когда баланс-фактор левого поддерева больше 1, и разница высот между левым и правым поддеревом больше 1. Это вращение выполняется для узла, который находится в левом поддереве своего родителя и имеет левого потомка.
  3. Правое вращение (RR): Используется, когда баланс-фактор правого поддерева больше 1, и разница высот между левым и правым поддеревом больше 1. Это вращение выполняется для узла, который находится в правом поддереве своего родителя и имеет правого потомка.

Лево-правое вращение (LR) и Право-левое вращение (RL): Используются, когда разница высот между левым и правым поддеревом больше 1, и узел имеет соответственно правого потомка с левым поддеревом или левого потомка с правым поддеревом.

Повторяйте шаги 2 и 3 до тех пор, пока дерево не станет сбалансированным.

# Код программы

## Структура LIFO

#include <iostream>

using namespace std;

template <typename T>

class Stack

{

    private:

        struct Node

        {

            T data;

            Node\* next;

            Node(T val): data(val), next(nullptr) {}

        };

        Node\* head;

        /\*

            @brief Рекурсивно выводает список с хвоста

            @param Нода для взятия из нее данных для вывода

            @return Строка с элементами стэка с хвоста

        \*/

        void showFromTailRecursive(Node\* node) const

        {

            if (node == nullptr)

                return;

            showFromTailRecursive(node->next);

            cout << node->data << " ";

        }

    public:

        Stack(): head(nullptr) {}

        /\*

            @brief Добавление нового элемента

            @param Значение, которое будет хранить новый элемент

        \*/

        void add(T value)

        {

            Node\* newNode = new Node(value);

            if (head == nullptr)

                head = newNode;

            else

            {

                newNode -> next = head; // присваиваем следующее значение next для head

                head = newNode; // а новое значение head примет next

                // это соответствует структуре LIFO

            }

        }

        /\*

            @brief Выдает список с головы

        \*/

        void showFromHead() const

        {

            if (head == nullptr)

            {

                cout << "stack is empty" << endl;

                return;

            }

            Node\* current = head;

            while (current != nullptr)

            {

                cout << current->data << " ";

                current = current -> next;

            }

            cout << endl;

        }

        /\*

            @brief Выдает элементы с хвоста

        \*/

        void showFromTail() const

        {

            if (head == nullptr)

            {

                cout << "stack is empty" << endl;

                return;

            }

            showFromTailRecursive(head);

            cout << endl;

        }

        /\*

            @brief Удаляет элементы с заданным значение

            @param Значение элемента для удаления

        \*/

        void remove(T value)

        {

            Node\* current = head;

            Node\* prev = nullptr;

            while (current != nullptr)

            {

                if (current -> data == value)

                {

                    if (prev == nullptr) // если текущий элемент - это голова

                    {

                        head = head -> next;

                        delete current;

                        current = head;

                    }

                    else // есди текущий элемент это все последующие

                    {

                        prev -> next = current -> next;

                        delete current;

                        current = prev -> next;

                    }

                }

                else

                {

                    prev = current;

                    current = current -> next;

                }

            }

        }

        /\*

            @brief Считает элементы с заданным значением

            @param Значение

            @return Количество элементов

        \*/

        int count (T value) const

        {

            int count = 0;

            Node\* current = head;

            while (current != nullptr)

            {

                if (current -> data == value)

                    count++;

                current = current -> next;

            }

            return count;

        }

};

int main()

{

    Stack<bool> stack;

    stack.add(true);

    stack.add(true);

    stack.add(false);

    stack.showFromTail();

    cout << stack.count(true) << endl;

    stack.remove(true);

    stack.showFromTail();

}

## Структура FIFO

#include <iostream>

using namespace std;

template <typename T>

class Queue

{

    private:

        struct Node

        {

            T data;

            Node\* next;

            Node(T val): data(val), next(nullptr) {}

        };

        Node\* head;

        Node\* tail;

        /\*

            @brief Рекурсивно выводает список с хвоста

            @param Нода для взятия из нее данных для вывода

        \*/

        void showFromTailRecursive(Node\* node) const

        {

            if (node == nullptr)

                return;

            showFromTailRecursive(node -> next);

            cout << node -> data << " ";

        }

    public:

        Queue(): head(nullptr), tail(nullptr) {}

        /\*

            @brief Добавление нового элемента

            @param Значение, которое будет хранить новый элемент

        \*/

        void add(T value)

        {

            Node\* newNode = new Node(value);

            if (head == nullptr)

            {

                head = newNode;

                tail = newNode;

            }

            else

            {

                tail -> next = newNode;

                tail = newNode;

            }

        }

        /\*

            @brief Удаляет элементы с заданным значение

            @param Значение элемента для удаления

        \*/

        void remove(T value)

        {

            Node\* current = head;

            Node\* prev = nullptr;

            while (current != nullptr)

            {

                if (current -> data == value)

                {

                    if (prev == nullptr) // если текущий элемент - это голова

                    {

                        head = head -> next;

                        delete current;

                        current = head;

                    }

                    else // есди текущий элемент это все последующие

                    {

                        prev -> next = current -> next;

                        delete current;

                        current = prev -> next;

                    }

                }

                else

                {

                    prev = current;

                    current = current -> next;

                }

            }

        }

        /\*

            @brief Выдает список с головы

        \*/

        void showFromHead() const

        {

            if (head == nullptr)

            {

                cout << "queue is empty" << endl;

                return;

            }

            Node\* current = head;

            while (current != nullptr)

            {

                cout << current -> data << " ";

                current = current -> next;

            }

            cout << endl;

        }

        /\*

            @brief Выдает элементы с хвоста

        \*/

        void showFromTail() const

        {

            if (head == nullptr)

            {

                cout << "queue is empty" << endl;

                return;

            }

            showFromTailRecursive(head);

            cout << endl;

        }

        /\*

            @brief Считает элементы с заданным значением

            @param Значение

            @return Количество элементов

        \*/

        int count (T value) const

        {

            int count = 0;

            Node\* current = head;

            while (current != nullptr)

            {

                if (current -> data == value)

                    count++;

                current = current -> next;

            }

            return count;

        }

};

int main()

{

    Queue<int> queue;

    queue.add(1);

    queue.add(1);

    queue.add(3);

    queue.showFromHead();

    cout << queue.count(1) << endl;

    queue.remove(1);

    queue.showFromHead();

}

## Сортированный список

#include <iostream>

using namespace std;

template <typename T>

class SortedList

{

    private:

        struct Node

        {

            T key;

            Node\* next;

            Node(T val): key(val), next(nullptr) {}

        };

        Node\* head;

        /\*

            @brief Рекурсивно выводает список с хвоста

            @param Нода для взятия из нее данных для вывода

            @return Строка с элементами стэка с хвоста

        \*/

        void showFromTailRecursive(Node\* node) const

        {

            if (node == nullptr)

                return;

            showFromTailRecursive(node -> next);

            cout << node -> key << " ";

        }

    public:

        SortedList(): head(nullptr) {}

        /\*

            @brief Добавление нового элемента

            @param Значение, которое будет хранить новый элемент

        \*/

       void add(T value)

       {

            Node\* newNode = new Node(value);

            if (!head || value < head->key) // встает перед головой

            {

                newNode->next = head;

                head = newNode;

            }

            else // встает после головы

            {

                Node\* current = head;

                while (current->next && current->next->key <= value) // проходим до тех пор, пока не встретим большее значение, это будет местом, где должен будет стоять наш элемент

                    current = current->next;

                // по итогу current - это элемент, который меньше или равен, которому вставляем, но после которого идет эл-т больше вставляемого

                newNode->next = current->next;

                current->next = newNode;

                // после всех вставок должно получится

                // current -> newNode -> current->next

            }

       }

       /\*

            @brief Выдает список с головы

        \*/

        void showFromHead() const

        {

            if (head == nullptr)

            {

                cout << "sorted list is empty" << endl;

                return;

            }

            Node\* current = head;

            while (current != nullptr)

            {

                cout << current -> key << " ";

                current = current -> next;

            }

            cout << endl;

        }

        /\*

            @brief Выдает элементы с хвоста

        \*/

        void showFromTail() const

        {

            if (head == nullptr)

            {

                cout << "sorted list is empty" << endl;

                return;

            }

            showFromTailRecursive(head);

            cout << endl;

        }

        /\*

            @brief Удаляет элементы с заданным значение

            @param Значение элемента для удаления

        \*/

        void remove(T value)

        {

            Node\* current = head;

            Node\* prev = nullptr;

            while (current != nullptr)

            {

                if (current -> key == value)

                {

                    if (prev == nullptr) // если текущий элемент - это голова

                    {

                        head = head -> next;

                        delete current;

                        current = head;

                    }

                    else // есди текущий элемент это все последующие

                    {

                        prev -> next = current -> next;

                        delete current;

                        current = prev -> next;

                    }

                }

                else

                {

                    prev = current;

                    current = current -> next;

                }

            }

        }

        /\*

            @brief Считает элементы с заданным значением

            @param Значение

            @return Количество элементов

        \*/

        int count (int value) const

        {

            int count = 0;

            Node\* current = head;

            while (current != nullptr)

            {

                if (current -> key == value)

                    count++;

                current = current -> next;

            }

            return count;

        }

};

int main()

{

    SortedList<bool> list;

    list.add(true);

    list.add(true);

    list.add(false);

    list.showFromHead();

    cout << list.count(true) << endl;

    list.remove(true);

    list.showFromTail();

}

## Сортированный двунаправленный список

#include <iostream>

using namespace std;

template <typename T>

class SortedBidirectionalList

{

    private:

        struct Node

        {

            T key;

            Node\* next;

            Node\* prev;

            Node(T val): key(val), next(nullptr), prev(nullptr) {}

        };

        Node\* head;

    public:

        SortedBidirectionalList(): head(nullptr) {}

        /\*

            @brief Добавление нового элемента

            @param Значение, которое будет хранить новый элемент

        \*/

       void add(T value)

       {

            Node\* newNode = new Node(value);

            if (!head || value < head->key) // встает перед головой

            {

                newNode->next = head;

                if (head)

                    head->prev = newNode;

                head = newNode;

            }

            else // встает после головы

            {

                Node\* current = head;

                while (current->next && current->next->key <= value) // проходим до тех пор, пока не встретим большее значение, это будет местом, где должен будет стоять наш элемент

                    current = current->next;

                // по итогу current - это элемент, который меньше или равен, которому вставляем, но после которого идет эл-т больше вставляемого

                newNode->next = current->next;

                if (current->next)

                    current->next->prev = newNode;

                current->next = newNode;

                newNode->prev = current;

                // после всех вставок должно получится

                // current <-> newNode <-> current->next

            }

       }

       /\*

            @brief Выдает список с головы

        \*/

        void showFromHead() const

        {

            if (head == nullptr)

            {

                cout << "sorted list is empty" << endl;

                return;

            }

            Node\* current = head;

            while (current != nullptr)

            {

                cout << current->key << " ";

                current = current -> next;

            }

            cout << endl;

        }

        /\*

            @brief Выдает элементы с хвоста

        \*/

        void showFromTail() const

        {

            if (head == nullptr)

            {

                cout << "sorted list is empty" << endl;

                return;

            }

            Node\* current = head;

            while (current->next != nullptr)

                current = current->next; // доходим до конца

            while (current != nullptr)

            {

                cout << current->key << " ";

                current = current -> prev;

            }

            cout << endl;

        }

        /\*

            @brief Удаляет элементы с заданным значение

            @param Значение элемента для удаления

        \*/

        void remove(T value)

        {

            Node\* current = head;

            while (current != nullptr)

            {

                if (current -> key == value)

                {

                    if (current->prev == nullptr) // если текущий элемент - это голова

                    {

                        head = head -> next;

                        head->prev = nullptr;

                        delete current;

                        current = head;

                    }

                    else // если текущий элемент это все последующие

                    {

                        current->prev -> next = current->next;

                        current->next -> prev = current->prev;

                        Node\* remember = current->next;

                        delete current;

                        current = remember;

                        remember = nullptr;

                    }

                }

                else

                    current = current -> next;

            }

        }

        /\*

            @brief Считает элементы с заданным значением

            @param Значение

            @return Количество элементов

        \*/

        int count (int value) const

        {

            int count = 0;

            Node\* current = head;

            while (current != nullptr)

            {

                if (current -> key == value)

                    count++;

                current = current -> next;

            }

            return count;

        }

};

int main()

{

    SortedBidirectionalList<int> list;

    list.add(6);

    list.add(10);

    list.add(15);

    list.add(10);

    list.showFromHead();

    list.showFromTail();

}

## Сортированный по 2 ключам список

#include <iostream>

using namespace std;

template <typename T1, typename T2>

class SortedBy2KeysList

{

    private:

        struct Node

        {

            T1 key1;

            T2 key2;

            Node\* next;

            Node(T1 val1, T2 val2): key1(val1), key2(val2), next(nullptr) {}

        };

        Node\* head;

        /\*

            @brief Рекурсивно выводает список с хвоста

            @param Нода для взятия из нее данных для вывода

        \*/

        void showFromTailRecursive(Node\* node) const

        {

            if (node == nullptr)

                return;

            showFromTailRecursive(node -> next);

            cout << node->key1 << ":" << node->key2 << endl;

        }

    public:

        SortedBy2KeysList(): head(nullptr) {}

        /\*

            @brief Добавление нового элемента

            @param Значение, которое будет хранить новый элемент

        \*/

       void add(T1 value1, T2 value2)

       {

            Node\* newNode = new Node(value1, value2);

            if (!head || value1 < head->key1 && value2 < head->key2) // встает перед головой

            {

                newNode->next = head;

                head = newNode;

            }

            else // встает после головы

            {

                Node\* current = head;

                while (current->next && current->next->key1 <= value1 && current->next->key2 <= value2) // проходим до тех пор, пока не встретим большее значение, это будет местом, где должен будет стоять наш элемент

                    current = current->next;

                // по итогу current - это элемент, который меньше или равен, которому вставляем, но после которого идет эл-т больше вставляемого

                newNode->next = current->next;

                current->next = newNode;

                // после всех вставок должно получится

                // current -> newNode -> current->next

            }

       }

       /\*

            @brief Выдает список с головы

        \*/

        void showFromHead() const

        {

            if (head == nullptr)

            {

                cout << "list is empty";

                return;

            }

            Node\* current = head;

            while (current != nullptr)

            {

                cout << current->key1 << ":" << current->key2 << endl;

                current = current -> next;

            }

            cout << endl;

        }

        /\*

            @brief Выдает элементы с хвоста

            @return Строка с элементами с хвоста

        \*/

        void showFromTail() const

        {

            if (head == nullptr)

            {

                cout << "list is empty";

                return;

            }

            showFromTailRecursive(head);

            cout << endl;

        }

        /\*

            @brief Удаляет элементы с заданным значение

            @param Значение элемента для удаления

        \*/

        void remove(T1 value1, T2 value2)

        {

            Node\* current = head;

            Node\* prev = nullptr;

            while (current != nullptr)

            {

                if (current->key1 == value1 && current->key2 == value2)

                {

                    if (prev == nullptr) // если текущий элемент - это голова

                    {

                        head = head -> next;

                        delete current;

                        current = head;

                    }

                    else // есди текущий элемент это все последующие

                    {

                        prev -> next = current -> next;

                        delete current;

                        current = prev -> next;

                    }

                }

                else

                {

                    prev = current;

                    current = current -> next;

                }

            }

        }

        /\*

            @brief Считает элементы с заданным значением

            @param Значение

            @return Количество элементов

        \*/

        int count (T1 value1, T2 value2) const

        {

            int count = 0;

            Node\* current = head;

            while (current != nullptr)

            {

                if (current -> key1 == value1 && current->key2 == value2)

                    count++;

                current = current -> next;

            }

            return count;

        }

};

int main()

{

    SortedBy2KeysList<string, int> list;

    list.add("Мусихин", 7);

    list.add("Иванов", 10);

    list.add("Иванов", 3);

    list.add("Шаравьев", 15);

    list.add("Мусихин", 6);

    list.add("Худеньких", 10);

    list.showFromHead();

    list.showFromTail();

}

## Бинарное дерево

// Создание бинарного дерева и работа с ним

#include <iostream>

using namespace std;

template <typename T>

class BinaryTree

{

private:

    struct Node

    {

        T data;

        Node\* right;

        Node\* left;

        int height;

        Node(T val): data(val), right(nullptr), left(nullptr), height(1) {}

    };

    Node\* root;

    // для добовления

    Node\* addRecursive(Node\* current, T value)

    {

        if (current == nullptr)

            return new Node(value);

        if (value < current->data)

            current->left = addRecursive(current->left, value);

        else if (value > current->data)

            current->right = addRecursive(current->right, value);

        return balance(current);

    }

    // для вывода

    void showRecursive(Node\* current, int l) const

    {

        if (current == nullptr)

            return;

        showRecursive(current->left, l+1);

        for (int i = 0; i < l; i++)

            cout << "\t";

        cout << current->data << endl;

        showRecursive(current->right, l+1);

        cout << endl;

    }

    void showRightToLeftRecursive(Node\* current, int l) const

    {

        if (current == nullptr)

            return;

        showRecursive(current->right, l+1);

        for (int i = 0; i < l; i++)

            cout << "\t";

        cout << current->data << endl;

        showRecursive(current->left, l+1);

        cout << endl;

    }

    // балансировка

    Node\* balance(Node\* node)

    {

        updateHeight(node);

        int balanceFactor = height(node->left) - height(node->right);

        // проверяем нарушение баланса

        if (balanceFactor > 1)

        {

            if (height(node->left->left) >= height(node->left->left))

                return rightRotate(node);

            else

            {

                leftRotate(node);

                return rightRotate(node);

            }

        }

        else if (balanceFactor < -1)

        {

            if (height(node->right->left) <= height(node->right->right))

                return leftRotate(node);

            else

            {

                rightRotate(node);

                return leftRotate(node);

            }

        }

        return node;

    }

    int height(Node\* node) const

    {

        if (node)

            return node->height;

        else

            return 0;

    }

    void updateHeight(Node\* node)

    {

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

    }

    Node\* leftRotate(Node\* y)

    {

        Node\* x = y->right;

        Node\* T2 = x->left;

        x->left = y;

        y->right = T2;

        updateHeight(y);

        updateHeight(x);

        return x;

    }

    Node\* rightRotate(Node\* y)

    {

        Node\* x = y->left;

        Node\* T2 = x->right;

        x->right = y;

        y->left = T2;

        updateHeight(y);

        updateHeight(x);

        return x;

    }

    // удаление

    Node\* findMin(Node\* current)

    {

        while (current->left)

            current = current->left;

        return current;

    }

    Node\* removeRecursive(Node\* current, T value)

    {

        if (current == nullptr)

            return current;

        if (value < current->data)

            current->left = removeRecursive(current->left, value);

        else if (value > current->data)

            current->right = removeRecursive(current->right, value);

        else

        {

            Node\* T2;

            if (current->left == nullptr)

            {

                T2 = current->right;

                delete current;

                return T2;

            }

            if (current->right == nullptr)

            {

                T2 = current->left;

                delete current;

                return T2;

            }

            T2 = findMin(current->right);

            current->data = T2->data;

            removeRecursive(current->right, T2->data);

        }

        return balance(current);

    }

    void delRecursive(Node\* current)

    {

        if (current == nullptr)

            return;

        delRecursive(current->left);

        delRecursive(current->right);

        delete current;

    }

    // поиск

    Node\* findRecursive(Node\* current, T value) const

    {

        if (current == nullptr)

            return current;

        Node\* T2;

        if (value < current->data)

        {

            T2 = findRecursive(current->left, value);

            if (T2)

                current = T2;

        }

        else if (value > current->data)

        {

            T2 = findRecursive(current->right, value);

            if (T2)

                current = T2;

        }

        return current;

    }

public:

    BinaryTree(): root(nullptr) {}

    /\*

        @brief Добавляет новый элемент

        @param Значение, которое будет хранить переменная

    \*/

    void add(T value)

    {

        root = addRecursive(root, value);

    }

    /\*

        @brief Показ дерева слева-направо

    \*/

    void show() const

    {

        if (root)

            showRecursive(root, 0);

        else

            cout << "tree is empty" << endl;

    }

    /\*

        @brief Показ дерева справа-налево

    \*/

    void showRightToLeft() const

    {

        if (root)

            showRightToLeftRecursive(root, 0);

        else

            cout << "tree is empty" << endl;

    }

    /\*

        @brief Удаление элемента

        @param Значение для элемента, который удаляем

    \*/

    void remove(T value)

    {

        root = removeRecursive(root, value);

    }

    /\*

        @brief Удаление всего дерева

    \*/

    void del()

    {

        delRecursive(root);

        root = nullptr;

    }

    /\*

        @brief Поиск элемента

        @param Значение для элемента, который ищем

    \*/

    void find(T value) const

    {

        Node\* node = findRecursive(root, value);

        if (node->data == value)

            showRecursive(node, 0);

        else

            cout << "there is no such element with value " << value << endl;

    }

};

int main()

{

    BinaryTree<int> tree;

    tree.add(1);

    tree.add(4);

    tree.add(7);

    tree.add(10);

    tree.show();

    tree.showRightToLeft();

    tree.add(-1);

    tree.add(-4);

    tree.show();

    tree.find(7);

    tree.remove(7);

    tree.show();

    tree.find(7);

    tree.del();

    tree.show();

}

# Тестирование

Для начала были проведены тесты для структуры LIFO, увидеть их можно в таблице 1.

Таблица 1 – LIFO тесты

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тесты** | **Входные данные** | **Ожидаемы результат** | **Полученный результат** | **Успешность** |
| **Т1** | вызов печати стека с головы | stack is empty | stack is empty | + |
| **Т2** | вызов печати стека с хвоста | stack is empty | stack is empty | + |
| **Т3** | добавление числа 10 вывести  удалить число 10 вывести | 10 stack is empty | 10 stack is empty | + |
| **Т4** | добавление 1.1 добавление 1.5 вывод поиск 2 | 1.1 1.5 0 | 1.1000 1.5000 0 | + |
| **Т5** | добавление hi, user, bay вывод с хвоста поиск hi удаление i вывод с хвоста | hi user bay 1 hi user bay | hi user bay 1 hi user bay | + |
| **Т6** | добавление правды добавление правды добавление лжи вывод с хвоста поиск правды удаление правды вывод с хвоста | правда правда ложь 2 ложь | 1 1 0 2 0 | + |

Покрытие этих тестов можно увидеть в черном ящике в таблице 2.

Таблица 2 - LIFO чя

| **Категории** | Т1 | Т2 | Т3 | Т4 | Т5 | Т6 |
| --- | --- | --- | --- | --- | --- | --- |
| **Длина стека** | | | | | | |
| нулевая | + | + |  |  |  |  |
| единичная |  |  | + |  |  |  |
| > 1 |  |  |  | + | + | + |
| **Элементы стека** | | | | | | |
| Целые числа |  |  | + |  |  |  |
| Символы |  |  |  |  | + |  |
| Строки |  |  |  |  | + |  |
| Булевы значения |  |  |  |  |  | + |
| **Выходные данные** | | | | | | |
| "stack is empty" | + | + |  |  |  |  |
| печать стека с головы |  |  | + | + |  |  |
| печать стека с хвоста |  |  |  |  | + | + |
| печать количества элементов с заданным параметром |  |  |  | + | + | + |
| **Поиск элемента** | | | | | | |
| не найден |  |  |  | + |  |  |
| найден 1 |  |  |  |  | + |  |
| найдено > 1 |  |  |  |  |  | + |
| **Удаление элемента** | | | | | | |
| Элемент в голове |  |  | + |  |  | + |
| Элемент в хвосте |  |  | + |  |  |  |
| Элемент посередине |  |  | + |  |  | + |
| Элемента нет |  |  |  |  | + |  |
| **Сколько удаляется за раз** |  |  |  |  |  |  |
| 0 |  |  |  |  | + |  |
| 1 |  |  | + |  |  |  |
| > 1 |  |  |  |  |  | + |
| **Печать стека** | | | | | | |
| с головы | + |  | + | + |  |  |
| с хвоста |  | + | + |  | + | + |

Далее были проведены тесты для структуры FIFO, смотреть в таблицу 3.

Таблица 3 - FIFO тесты

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тесты** | **Входные данные** | **Ожидаемы результат** | **Полученный результат** | **Успешность** |
| Т1 | вывод с начала поиск элемента 1 удаление элемента 1 вывод с конца | queue is empty 0 queue is empty | queue is empty 0 queue is empty | + |
| Т2 | добавление 1 вывод поиск 1 удаление 1 вывод | 1 1 queue is empty | 1 1 queue is empty | + |
| Т3 | добавление 1 добавление 1 добавление 3 вывод поиск 1 удаление 1 вывод | 1 1 3 2 3 | 1 1 3 2 4 | + |

И покрытие этих тестов по черному ящику в таблице 4. Важно отметить, что в этом черном ящике и последующих уже не учитываются переменные разных типов, так как хранение их взаимодействие с ними происходит через вывод, а в выводы в каждом из классов устроены одинаково.

Таблица 4 - FIFO чя

| **Категории** | Т1 | Т2 | Т3 |
| --- | --- | --- | --- |
| **Длина очереди** | | | |
| нулевая | + |  |  |
| единичная |  | + |  |
| > 1 |  |  | + |
| **Выходные данные** | | | |
| вывод очереди |  | + | + |
| количество найденных элементов | + | + | + |
| "queue is empty" | + |  |  |
| **Поиск элемента** | | | |
| элемент не найден | + |  |  |
| найден 1 |  | + |  |
| найдено > 1 |  |  | + |
| **Удаление элемента** | | | |
| в начале |  | + | + |
| в середине |  | + | + |
| в конце |  | + |  |
| элемента нет | + |  |  |
| **Сколько удаляется за раз** | | | |
| 0 | + |  |  |
| 1 |  | + |  |
| > 1 |  |  | + |
| **Печать стека** | | | |
| с начала | + | + |  |
| с хвоста | + |  |  |

Далее в таблицах 5–12 пойдут тесты и их покрытия по черным ящикам для сортированного списка, двунаправленного сортированного списка, сортированного по 2 ключам списка и идеально-сбалансированно идеального дерева.

Таблица 5 - SortedList тесты

| **Тесты** | **Входные данные** | **Ожидаемы результат** | **Полученный результат** | **Успешность** |
| --- | --- | --- | --- | --- |
| Т1 | вывести с начала найти 1 удалить 1 вывести с конца | sorted list is empty 0 sorted list is empty | sorted list is empty 0 sorted list is empty | + |
| Т2 | добавить а вывести с начала найти а удалить а удалить а | а 1 sorted list is empty | а 1 sorted list is empty | + |
| Т3 | добавить правду правду ложь вывести с начала найти правду удалить правду вывести с конца | 0 1 1 2 0 | 0 1 1 2 1 | + |

В таблице 6 в данном черном ящике снова учитывается, что переменные могут иметь разные типы, так как нужно убедиться, что сравнения работают корректно для любого типа, чтобы убедиться, что список правда будет сортированным. Далее это снова не будет учитываться, так как механизм работы точно такой же, как и в классе SortedList.

Таблица 6 - SortedList чя

| **Категории** | Т1 | Т2 | Т3 |
| --- | --- | --- | --- |
| **Длина массива** | | | |
| нулевая | + |  |  |
| единичная |  | + |  |
| > 1 |  |  | + |
| **Выходные данные** | | | |
| вывод массива | + | + | + |
| количество найденных элементов | + | + | + |
| "list is empty" | + | + | + |
| **Поиск элемента** | | | |
| элемент не найден | + |  |  |
| найден 1 |  | + |  |
| найдено > 1 |  |  | + |
| **Удаление элемента** | | | |
| в начале |  | + |  |
| в середине |  | + | + |
| в конце |  | + | + |
| элемента нет | + |  |  |
| **Сколько удаляется за раз** | | | |
| 0 | + |  |  |
| 1 |  | + |  |
| > 1 |  |  | + |
| **Печать массива** | | | |
| с начала | + | + | + |
| с хвоста | + | + | + |
| **Элементы массива** | | | |
| числа | + |  |  |
| символы |  | + |  |
| логические переменные |  |  | + |

Таблица 7 - SortedBidirectionalList тесты

| **Тесты** | **Входные данные** | **Ожидаемы результат** | **Полученный результат** | **Успешность** |
| --- | --- | --- | --- | --- |
| Т1 | вывести с начала найти 1 удалить 1 вывести с конца | sorted list is empty 0 sorted list is empty | sorted list is empty 0 sorted list is empty | + |
| Т2 | добавить 1 вывести с начала найти 1 удалить 1 вывести с конца | 1 1 sorted list is empty | 1 1 sorted list is empty | + |
| Т3 | добавить 1 1 0 вывести с начала найти 1 удалить 1 вывести с конца | 0 1 1 2 0 | 0 1 1 2 1 | + |

Таблица 8 - SortedBidirectionalList чя

|  |  |  |  |
| --- | --- | --- | --- |
| **Категории** | Т1 | Т2 | Т3 |
| **Длина массива** | | | |
| нулевая | + |  |  |
| единичная |  | + |  |
| > 1 |  |  | + |
| **Выходные данные** | | | |
| вывод массива | + | + | + |
| количество найденных элементов | + | + | + |
| "list is empty" | + | + | + |
| **Поиск элемента** | | | |
| элемент не найден | + |  |  |
| найден 1 |  | + |  |
| найдено > 1 |  |  | + |
| **Удаление элемента** | | | |
| в начале |  | + |  |
| в середине |  | + | + |
| в конце |  | + | + |
| элемента нет | + |  |  |
| **Сколько удаляется за раз** | | | |
| 0 | + |  |  |
| 1 |  | + |  |
| > 1 |  |  | + |
| **Печать массива** | | | |
| с начала | + | + | + |
| с хвоста | + | + | + |

Таблица 9 - SortedBy2KeysList тесты

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тесты** | **Входные данные** | **Ожидаемы результат** | **Полученный результат** | **Успешность** |
| Т1 | вывести с начала найти М 1 удалить М 1 вывести с конца | sorted list is empty 0 sorted list is empty | sorted list is empty 0 sorted list is empty | + |
| Т2 | добавить М 1 вывести с начала найти М 1 удалить М 1 вывести с конца | М: 1 1 sorted list is empty | М: 1 1 sorted list is empty | + |
| Т3 | добавить М: 1 М: 1 А: 0 вывести с начала найти М: 1 удалить М: 1 вывести с конца | А: 0  М: 1  М: 1 2 А: 0 | А: 0  М: 1  М: 1 2 А: 1 | + |

Таблица 10 - SortedBy2KeysList чя

| **Категории** | Т1 | Т2 | Т3 |
| --- | --- | --- | --- |
| **Длина массива** | | | |
| нулевая | + |  |  |
| единичная |  | + |  |
| > 1 |  |  | + |
| **Выходные данные** | | | |
| вывод массива | + | + | + |
| количество найденных элементов | + | + | + |
| "list is empty" | + | + | + |
| **Поиск элемента** | | | |
| элемент не найден | + |  |  |
| найден 1 |  | + |  |
| найдено > 1 |  |  | + |
| **Удаление элемента** | | | |
| в начале |  | + |  |
| в середине |  | + | + |
| в конце |  | + | + |
| элемента нет | + |  |  |
| **Сколько удаляется за раз** | | | |
| 0 | + |  |  |
| 1 |  | + |  |
| > 1 |  |  | + |
| **Печать массива** | | | |
| с начала | + | + | + |
| с хвоста | + | + | + |

Таблица 11 - BinaryTree тесты

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тесты** | **Входные данные** | **Ожидаемы результат** | **Полученный результат** | **Успешность** |
| Т1 | вывести слева найти 1 удалить 1 вывести справа | tree is empty 0 sorted list is empty | sorted list is empty 0 sorted list is empty | + |
| Т2 | добавить 1 вывести слева найти 1 удалить 1 вывести слева | 1 1 tree is empty | 1 1 tree is empty | + |
| Т3 | добавить 1 2 3 вывести слева найти 1 удалить 1 вывести справа | 1 2  3 1  3 2  1 | 1 2  3 1  3 2  2 | + |

Таблица 12 - BinatyTree чя

|  |  |  |  |
| --- | --- | --- | --- |
| **Категории** | Т1 | Т2 | Т3 |
| **Длина дерева** | | | |
| нулевая | + |  |  |
| единичная |  | + |  |
| > 1 |  |  | + |
| **Выходные данные** | | | |
| вывод дерева | + | + | + |
| найденный элемент | + | + | + |
| "tree is empty" | + | + | + |
| **Поиск элемента** | | | |
| элемент не найден | + |  |  |
| найден 1 |  | + |  |
| **Удаление элемента** | | | |
| в начале |  | + |  |
| в середине |  | + | + |
| в конце |  | + | + |
| элемента нет | + |  |  |
| **Печать дерева** | | | |
| слева направо | + | + | + |
| справа налево | + | + | + |