Министерство науки и высшего образования Российской Федерации

федеральное государственное бюджетное образовательное учреждение высшего образования «Казанский национальный исследовательский технический университет им. А.Н. Туполева-КАИ»

(КНИТУ-КАИ)

Институт Компьютерных технологий и защиты информации

Кафедра Динамики процессов и управления

**ОТЧЕТ**

**по лабораторным работам**

**по**

**алгоритмам и структуре данных**

Направление подготовки/специальность: 09.03.03 «Прикладная информатика»

Выполнил:

Шарипов Р.Р. 4216

Проверил:

Семенов П.К.

**Казань 2023**

1. Множества class Set<T> where T: IComparable

Описание класса: Класс Set<T> представляет собой реализацию множества в C#. Он обеспечивает хранение и операции с элементами, которые должны быть сравнимы с помощью интерфейса IComparable.

Класс предоставляет возможности добавления, удаления, проверки наличия элемента, получения элемента по индексу, а также выполнения операций объединения, пересечения и разности множеств.

Поля класса:

* int size: размер массива данных.
* int n: текущее количество элементов в множестве.
* T[] data: массив данных, представляющих элементы множества.
* bool done: флаг, указывающий на успешное выполнение операции.

Конструктор:

* public Set(int n): создает экземпляр класса Set<T> с указанным начальным размером n. Если n больше нуля, создается массив данных размером n.

Методы класса:

* public int GetSize(): возвращает размер массива данных.
* public int GetCount(): возвращает текущее количество элементов в множестве.
* public int Count { get { return size; } }: свойство, представляющее количество элементов в множестве.
* public int GetIndex(T value): возвращает индекс первого вхождения указанного элемента value в множество. Если элемент не найден, возвращает -1.
* public bool IsContains(T element): проверяет наличие указанного элемента element в множестве и возвращает true, если элемент найден, и false в противном случае.
* public T this[int index]: индексатор, позволяющий получить элемент множества по указанному индексу.
* public bool Exists(T value): проверяет наличие указанного элемента value в множестве и возвращает true, если элемент найден, и false в противном случае.
* public void Add(T value): добавляет указанный элемент value в множество. Если элемент уже присутствует, он не будет добавлен повторно.
* public void RemoveAt(int index): удаляет элемент множества по указанному индексу.
* public void Remove(T value): удаляет первое вхождение указанного элемента value из множества.
* public T GetValue(int index): возвращает элемент множества по указанному индексу.
* public override string ToString(): возвращает строковое представление множества, заключенное в фигурные скобки и разделенное запятыми.
* public Set<T> Union(Set<T> ss): выполняет операцию объединения текущего множества с указанным множеством ss и возвращает новое множество, содержащее все элементы из обоих множеств.
* public static Set<T> operator +(Set<T> s1, Set<T> s2): перегруженный оператор "+", позволяющий объединить два множества.
* public List<Set<T>> subSets(): возвращает список подмножеств текущего множества.
* public Set<T> Interesection(Set<T> s2): выполняет операцию пересечения текущего множества с указанным множеством s2 и возвращает новое множество, содержащее только общие элементы.
* public static Set<T> operator *(Set<T> s1, Set<T> s2): перегруженный оператор "*", позволяющий выполнить операцию пересечения двух множеств.
* public Set<T> Addition(Set<T> s2): выполняет операцию разности текущего множества с указанным множеством s2 и возвращает новое множество, содержащее элементы, которые есть в текущем множестве, но отсутствуют во втором множестве.
* public static Set<T> operator -(Set<T> s1, Set<T> s2): перегруженный оператор "-", позволяющий выполнить операцию разности двух множеств.
* private bool CheckCombIndex(int[] index, int k): вспомогательный метод для операции генерации комбинаций.
* public List<Set<T>> Combination(int k): возвращает список всех комбинаций без повторений размера k из элементов множества.

1. Сортировки class Sort

Класс "Sort" представляет собой коллекцию статических методов, реализующих различные алгоритмы сортировки

1. public static void BubbleSort<T>(T[] list) where T : IComparable
   * Описание: реализует сортировку пузырьком. Перебирает элементы массива и сравнивает каждую пару соседних элементов. Если элементы не упорядочены, они меняются местами. Процесс повторяется до тех пор, пока все элементы не станут упорядоченными.
   * Параметры:
     + **list**: Массив элементов типа T, который требуется отсортировать. Тип T должен реализовывать интерфейс **IComparable** для сравнения элементов.
   * Возвращаемое значение: Метод не возвращает значения. Он изменяет переданный массив **list**, сортируя его по возрастанию.
2. public static void MaxSort<T>(T[] list) where T : IComparable
   * Описание: реализует сортировку выбором. Ищет максимальный элемент в массиве и помещает его в конец. Затем повторяет этот процесс для оставшихся элементов до тех пор, пока весь массив не будет упорядочен.
   * Параметры:
     + **list**: Массив элементов типа T, который требуется отсортировать. Тип T должен реализовывать интерфейс **IComparable** для сравнения элементов.
   * Возвращаемое значение: Метод не возвращает значения. Он изменяет переданный массив **list**, сортируя его по возрастанию.
3. public static void InsertionSort<T>(T[] list) where T : IComparable
   * Описание: реализует сортировку вставками. Перебирает элементы массива и вставляет каждый элемент в правильную позицию в отсортированной части массива слева от текущего элемента.
   * Параметры:
     + **list**: Массив элементов типа T, который требуется отсортировать. Тип T должен реализовывать интерфейс **IComparable** для сравнения элементов.
   * Возвращаемое значение: Метод не возвращает значения. Он изменяет переданный массив **list**, сортируя его по возрастанию.
4. public static void ShellSort<T>(T[] list) where T : IComparable
   * Описание: реализует сортировку Шелла. Использует последовательность шагов сортировки вставками для упорядочивания элементов массива. Шаги уменьшаются до 1, чтобы окончательно отсортировать массив.
   * Параметры:
     + **list**: Массив элементов типа T, который требуется отсортировать. Тип T должен реализовывать интерфейс **IComparable** для сравнения элементов.
   * Возвращаемое значение: Метод не возвращает значения. Он изменяет переданный массив **list**, сортируя его по возрастанию.
5. public static void CountingSort(int[] list, int maxValue)
   * Описание: реализует сортировку подсчетом. Создает массив **array** для подсчета количества каждого значения в массиве **list**. Затем восстанавливает упорядоченный массив из массива подсчета.
   * Параметры:
     + **list**: Массив целых чисел, который требуется отсортировать.
     + **maxValue**: Максимальное значение в массиве **list**.
   * Возвращаемое значение: Метод не возвращает значения. Он изменяет переданный массив **list**, сортируя его по возрастанию.
6. public static void QuickSort2<T>(T[] list, int left, int right) where T : IComparable
   * Описание: реализует быструю сортировку (QuickSort) с использованием рекурсии. Выбирает опорный элемент (пивот) и разделяет массив на две части: элементы меньше пивота и элементы больше пивота. Затем рекурсивно сортирует обе части массива.
   * Параметры:
     + **list**: Массив элементов типа T, который требуется отсортировать. Тип T должен реализовывать интерфейс **IComparable** для сравнения элементов.
     + **left**: Индекс левой границы сортируемой части массива.
     + **right**: Индекс правой границы сортируемой части массива.
   * Возвращаемое значение: Метод не возвращает значения. Он изменяет переданный массив **list**, сортируя его по возрастанию.
7. public static void MergeSort<T>(T[] list, int left, int right) where T : IComparable
   * Описание: реализует сортировку слиянием (MergeSort) с использованием рекурсии. Разделяет массив на две половины, рекурсивно сортирует обе половины, а затем сливает их в один отсортированный массив.
   * Параметры:
     + **list**: Массив элементов типа T, который требуется отсортировать. Тип T должен реализовывать интерфейс **IComparable** для сравнения элементов.
     + **left**: Индекс левой границы сортируемой части массива.
     + **right**: Индекс правой границы сортируемой части массива.
   * Возвращаемое значение: Метод не возвращает значения. Он изменяет переданный массив **list**, сортируя его по возрастанию.
8. public static void MergeSortNR<T>(T[] list) where T : IComparable
   * Описание: реализует сортировку слиянием (MergeSort) с использованием итераций (без рекурсии). Разделяет массив на подмассивы фиксированного размера, сортирует их с помощью слияния и объединяет в один отсортированный массив.
   * Параметры:
     + **list**: Массив элементов типа T, который требуется отсортировать. Тип T должен реализовывать интерфейс **IComparable** для сравнения элементов.
   * Возвращаемое значение: Метод не возвращает значения. Он изменяет переданный массив **list**, сортируя его по возрастанию.
9. Односвязный список class SingleLinkedList<K, T> where K : IComparable where T : IComparable

Класс **SingleNode<K, T>** представляет собой узел односвязного списка и является наследником класса **Item<K, T>**. Этот класс определен как обобщенный, принимая два параметра типа **K** и **T**, которые обозначают типы ключа и значения узла соответственно.

Основные члены класса **SingleNode<K, T>**:

1. **next** (приватное поле): представляет ссылку на следующий узел в односвязном списке.
2. Конструкторы:
   * **SingleNode(K key, T value)**: Создает новый экземпляр узла с указанным ключом и значением, наследуя конструктор класса **Item<K, T>**. Устанавливает значение поля **next** в **null**.
   * **SingleNode()**: Создает новый экземпляр узла без параметров, наследуя конструктор класса **Item<K, T>**. Устанавливает значение поля **next** в **null**.
3. **Next** (свойство): обеспечивает доступ к полю **next**. Позволяет получить или задать ссылку на следующий узел в списке.
4. **ToString()**: Переопределяет метод **ToString()** класса **Item<K, T>**, чтобы предоставить строковое представление узла.

Класс **SingleNode<K, T>** представляет базовый строительный блок односвязного списка и содержит информацию о ключе и значении узла. Он также обладает ссылкой на следующий узел, что обеспечивает связь между узлами списка.

Класс SingleLinkedList<K, T>:

Класс **SingleLinkedList<K, T>** представляет собой односвязный список и является обобщенным (generic). Он параметризуется типами **K** и **T**, которые обозначают типы ключа и значения, хранящихся в узлах списка, соответственно. Класс также ограничен ограничением **where K: IComparable where T : IComparable**, что означает, что типы **K** и **T** должны реализовывать интерфейс **IComparable**.

Основные члены класса **SingleLinkedList<K, T>**:

1. **first** (приватное поле): Ссылка на начальный (первый) узел списка.
2. **pos** (приватное поле): хранит количество узлов в списке.
3. **First** (свойство): обеспечивает доступ к полю **first** и возвращает ссылку на начальный узел списка.
4. **Count** (свойство): возвращает количество узлов в списке.
5. Конструкторы:
   * **SingleLinkedList()**: Создает новый пустой экземпляр односвязного списка.
6. Методы:
   * **AddBegin(K key, T value)**: Добавляет новый узел с указанным ключом и значением в начало списка. Возвращает новую позицию в списке.
   * **AddEnd(K key, T value)**: Добавляет новый узел с указанным ключом и значением в конец списка. Если список пустой, устанавливает первый узел в новый узел. Возвращает новую позицию в списке.
   * **Clear()**: Очищает список путем установки первого узла в **null** и сбрасывает значение счетчика **pos**.
   * **ContainsValue(T value)**: Проверяет, содержит ли список узел с указанным значением. Возвращает **true**, если узел найден, иначе **false**.
   * **ContainsKey(K key)**: Проверяет, содержит ли список узел с указанным ключом. Возвращает **true**, если узел найден, иначе **false**.
   * **getNode(int k)**: Возвращает узел по указанной позиции **k** в списке. Если список пуст или указана недопустимая позиция, возвращает **null**.
   * **InsertAfterNode(K key, K newKey, T Value)**: Вставляет новый узел с указанным ключом и значением после узла с указанным ключом **key**. Если узел с ключом **key** не найден, вставка не выполняется. Возвращает новую позицию в списке.
   * **InsertBeforeNode(K key, K newKey, T Value)**: Вставляет новый узел с указанным ключом и значением перед узлом с указанным ключом **key**. Если узел с ключом **key** не найден, вставка выполняется в конец списка. Возвращает новую позицию в списке.
   * **RemoveFirstNode()**: Удаляет первый узел из списка.
   * **RemoveLastNode()**: Удаляет последний узел из списка.
   * **RemoveAt(int index)**: Удаляет узел по указанному индексу **index** в списке. Если указан недопустимый индекс, удаление не выполняется.
   * **Remove(K key)**: Удаляет узел с указанным ключом **key** из списка. Если узел с ключом **key** не найден, ничего не происходит.
   * **WriteList()**: Выводит содержимое списка, перебирая все узлы и выводя их ключи и значения в консоль.

Данный метод **Reverse()** используется для разворота односвязного списка. Давайте рассмотрим его работу по шагам:

1. Создаются три переменные: **prev** (предыдущий узел), **current** (текущий узел) и **next** (следующий узел). Изначально **prev** и **next** инициализируются как **null**, а **current** получает значение первого узла (**first**) списка.
2. Запускается цикл **while**, который выполняется до тех пор, пока **current** не станет **null**, то есть пока не пройдем по всем узлам списка.
3. Внутри цикла сохраняется ссылка на следующий узел в переменную **next**, чтобы не потерять доступ к следующему узлу после изменения ссылок.
4. Далее происходит изменение ссылки **current.Next**. Она указывает на предыдущий узел (**prev**), что обеспечивает разворот связи узлов.
5. Переменная **prev** обновляется, чтобы указывать на текущий узел (**current**), так как мы перемещаемся дальше по списку.
6. Переменная **current** обновляется, чтобы указывать на следующий узел (**next**), чтобы продолжить итерацию по списку.
7. Цикл повторяется, пока не будут пройдены все узлы списка.
8. После окончания цикла переменная **first** обновляется, чтобы указывать на новый головной узел (бывший последний узел), то есть переменная **prev**.
9. Stack public class Stack<T>

**Стек** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *stack* — стопка; читается *стэк*) — [абстрактный тип данных](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), представляющий собой [список элементов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), организованных по принципу [*LIFO*](https://ru.wikipedia.org/wiki/LIFO) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *last in — first out*, «последним пришёл — первым вышел»).

Класс представляет реализацию стека с параметризованным типом **T**.

* + Содержит приватные поля **\_Size**, **\_Array** и **\_Top**.
    1. **\_Size** представляет размер стека.
    2. **\_Array** представляет массив для хранения элементов стека.
    3. **\_Top** представляет индекс верхнего элемента стека.
  + Объявляет конструктор, который создает стек с заданным размером и инициализирует поля.
  + Объявляет свойство **Top**, позволяющее получить индекс верхнего элемента стека.
  + Объявляет свойство **Size**, позволяющее получить размер стека.
  + Реализует метод **IsFull()**, который проверяет, заполнен ли стек.
  + Реализует метод **IsEmpty()**, который проверяет, пуст ли стек.
  + Реализует метод **Push(T Element)**, который добавляет элемент в стек.
  + Реализует метод **Peek()**, который возвращает верхний элемент стека без его удаления.
  + Реализует метод **Pop()**, который удаляет и возвращает верхний элемент стека.
  + Включает приватный метод **Resize(int newSize)**, который изменяет размер массива **\_Array**, если необходимо.
    1. Метод создает новый массив с новым размером, копирует элементы из старого массива и обновляет **\_Array**, **\_Size** и **\_Top** соответственно.

Этот класс представляет реализацию стека с основными операциями, такими как добавление, удаление и получение элемента вершины стека. Класс также автоматически изменяет размер стека при необходимости, чтобы обеспечить его эффективность и гибкость.

1. HashTable
2. class HashItem<K, T> : Item<K, T>
   * Этот класс представляет элемент хэш-таблицы с ключом типа **K** и значением типа **T**.
   * Содержит защищенные поля **key** и **value**, представляющие ключ и значение элемента соответственно.
   * Определяет свойства **Value** и **Key**, позволяющие получить и установить значение и ключ элемента соответственно.
   * Переопределяет метод **ToString()**, который возвращает строковое представление элемента в формате "(key=value)".
   * Объявляет конструкторы:
     + Пустой конструктор, инициализирующий **key** и **value** значениями по умолчанию.
     + Конструктор, принимающий **key** и **value** и инициализирующий соответствующие поля.
3. class HashTableOpen<K, T> where K : IComparable where T : IComparable

Класс HashTableOpen<K, T> представляет собой реализацию хэш-таблицы с использованием метода цепочек (открытое хэширование). Хэш-таблица позволяет хранить элементы данных, связанные с ключами. Если несколько элементов имеют одинаковый хеш, они объединяются в связанный список.

Основные члены класса HashTableOpen<K, T>:

* + - 1. capacity (приватное поле): Ёмкость (размер) хэш-таблицы, задается при создании объекта.
      2. size (приватное поле): Количество элементов в хэш-таблице.
      3. items (приватное поле): Словарь, содержащий списки элементов хэш-таблицы. Ключ словаря - хеш-значение элементов, значение словаря - список элементов с одинаковым хеш-значением.
      4. Item (свойство только для чтения): позволяет получить все элементы хэш-таблицы в виде коллекции пар "хеш-значение" - "список элементов".

Конструкторы:

1. HashTableOpen(int capacity): Создает новый экземпляр хэш-таблицы с заданной ёмкостью.

Методы:

* 1. GetHash(K key): Возвращает хеш-значение для заданного ключа. Хеш-значение вычисляется с использованием метода GetHashCode() для ключа и приводится к положительному значению в диапазоне от 0 до capacity.
  2. Resize(int newCapacity): изменяет ёмкость хэш-таблицы на новое значение newCapacity. При изменении ёмкости перехэширование выполняется путем перераспределения элементов в новые списки с учетом нового размера.
  3. Insert(K key, T value): Вставляет новый элемент с заданным ключом и значением в хэш-таблицу. Если элемент с таким же ключом уже существует, выводится сообщение об ошибке. При вставке элемента вычисляется его хеш-значение, и элемент добавляется в соответствующий список элементов с тем же хеш-значением.
  4. Delete(K key): Удаляет элемент с заданным ключом из хэш-таблицы. Если элемент с таким ключом не существует, ничего не происходит.
  5. Search(K key): Поиск элемента с заданным ключом в хэш-таблице. Возвращает найденный элемент или null, если элемент не найден.
  6. View(): Выводит содержимое хэш-таблицы на экран. Перебирает все элементы в словаре items и выводит хеш-значение каждого элемента вместе с ключами и значениями элементов.

Метод **Insert(K key, T value)** служит для вставки нового элемента с заданным ключом и значением в хэш-таблицу. Вот подробное описание этого метода:

1. Создается новый экземпляр **HashItem<K, T>** с заданным ключом и значением.
2. Получается хеш-значение для ключа с помощью метода **GetHash(key)**.
3. Проверяется, существует ли уже список элементов с таким хеш-значением в словаре **items**.
4. Если список уже существует, получается ссылка на него.
5. Проверяется, существует ли элемент с таким же ключом в списке элементов.
   * Если элемент уже существует, выводится сообщение об ошибке ("Такое значение существует") и метод завершается.
   * Если элемент с таким ключом не найден, новый элемент добавляется в список элементов с соответствующим хеш-значением.
6. Если список элементов с заданным хеш-значением не существует, создается новый список **hashTableItem** и в него добавляется новый элемент **item**.
7. Созданный список **hashTableItem** добавляется в словарь **items** с ключом, равным хеш-значению элементов.

Метод **Delete(K key)** служит для удаления элемента с заданным ключом из хэш-таблицы. Вот подробное описание этого метода:

1. Получается хеш-значение для заданного ключа с помощью метода **GetHash(key)**.
2. Если элемент с данным хеш-значением не существует в словаре **items**, метод завершается.
3. Получается список элементов, соответствующих заданному хеш-значению, из словаря **items**.
4. Ищется элемент с заданным ключом в списке элементов.
5. Если элемент с заданным ключом найден, он удаляется из списка элементов.
6. class HashTableClose<K, T> where K : IComparable where T : IComparable

Класс **HashTableClose<K, T>** представляет реализацию хэш-таблицы с закрытым хэшированием. Вот его описание:

1. Приватные поля:
   * **capacity** - представляет ёмкость хэш-таблицы (начальное количество ячеек).
   * **size** - представляет количество элементов в хэш-таблице.
   * **items** - массив элементов **Item<K, T>**.
2. Метод **GetHash(K key)**:
   * Приватный метод, который вычисляет хеш-код для заданного ключа.
   * Использует функцию **Math.Abs(key.GetHashCode()) % capacity** для получения хеш-кода.
   * Возвращает вычисленное хеш-значение.
3. Свойство **Size**:
   * Позволяет получить только для чтения количество элементов в хэш-таблице.
4. Свойство **Capacity**:
   * Позволяет получить только для чтения ёмкость хэш-таблицы.
5. Метод **IsEmpty()**:
   * Проверяет, пуста ли хэш-таблица.
   * Возвращает **true**, если **size** равно 0, иначе возвращает **false**.
6. Конструктор **HashTableClose(int capacity)**:
   * Инициализирует **capacity** и **size**.
   * Создает массив **items** с указанной ёмкостью.
7. Метод **Insert(K key, T value)**:
   * Вставляет элемент с заданным ключом и значением в хэш-таблицу.
   * Если текущий размер хэш-таблицы равен ёмкости, вызывает метод **Resize()** для увеличения ёмкости в два раза.
   * Вычисляет хеш-значение для ключа.
   * Если текущий индекс занят и ключи не совпадают, ищет свободный индекс для вставки элемента.
   * Вставляет элемент по найденному индексу в массив **items**.
8. Метод **Search(K key)**:
   * Ищет элемент с заданным ключом в хэш-таблице.
   * Вычисляет хеш-значение для ключа.
   * Если текущий индекс занят и ключи не совпадают, ищет индекс элемента с заданным ключом.
   * Если элемент не найден, выбрасывает исключение **KeyNotFoundException**.
   * Иначе возвращает значение элемента.
9. Метод **View()**:
   * Выводит содержимое хэш-таблицы на экран.
   * Перебирает все элементы в массиве **items**.
   * Если элемент не равен **null**, выводит ключ и значение элемента.
10. Метод **Delete(K key)**:
    * Удаляет элемент с заданным ключом из хэш-таблицы.
    * Вычисляет хеш-значение для ключа.
    * Если текущий индекс занят и ключи не совпадают, ищет индекс элемента с заданным ключом.
    * Если элемент не найден, возвращает **false**.
    * Иначе удаляет элемент из массива **items** и возвращает **true**.
11. В методе **Insert(K key, T value)** после вставки элемента в хэш-таблицу, увеличивает **size** на 1.
12. Метод **Resize(int newCapacity)**:

* Создает новый массив элементов **newItems** с новой ёмкостью **newCapacity**.
* Перебирает все элементы в текущей таблице и перехеширует их в новую таблицу.
* Для каждого элемента в текущей таблице:
  + Получает ключ и значение элемента.
  + Вычисляет новый хеш для ключа в новой таблице.
  + Если текущий индекс занят и ключи не совпадают, ищет свободный индекс для вставки элемента в новую таблицу.
  + Вставляет элемент в новую таблицу по найденному индексу.
* Обновляет **capacity** и **items** текущей таблицы.

1. class BinaryTree<T>

Класс class Node<T>также имеет следующие методы:

* **Node(T value, int key)**: Конструктор класса, который инициализирует новый узел с заданным значением и ключом.
* **Node()**: Конструктор класса, который инициализирует новый узел со значениями по умолчанию.
* **ToString()**: Переопределенный метод, возвращающий строковое представление узла, содержащее информацию о ключе, значении и ссылках на родителя, левого и правого потомков.

Класс BinaryTree<T> представляет собой реализацию двоичного дерева поиска, где T - тип данных, хранящихся в узлах дерева. Дерево представлено в виде ссылочной структуры, где каждый узел содержит ссылки на левого и правого потомков, а также ссылку на родительский узел.

1. **Поля класса:**
   * **root**: Ссылка на корневой узел дерева.
2. **Методы класса:**
   * public void Insert(T value, int key) Метод для вставки нового узла с указанным значением и ключом в дерево. Если дерево пустое, новый узел становится корневым. Если узел с таким ключом уже существует в дереве, его значение обновляется.
   * public Node<T> Find(int key) Метод для поиска узла с заданным ключом в дереве. Возвращает найденный узел или **null**, если узел не найден.
   * public void Delete(int key) Метод для удаления узла с заданным ключом из дерева. В зависимости от количества потомков удаляемого узла, применяется соответствующая логика удаления.
   * public void ViewTree() Метод для просмотра содержимого дерева. Обход дерева выполняется в порядке возрастания ключей.
   * public Node<T> FindMin(Node<T> node)Вспомогательный метод для поиска узла с минимальным значением ключа, начиная с указанного узла.
   * public Node<T> FindMax(Node<T> node)Вспомогательный метод для поиска узла с максимальным значением ключа, начиная с указанного узла.
   * public Node<T> Next(Node<T> t) Метод для поиска узла со следующим значением ключа после указанного узла.
   * public Node<T> Prev(Node<T> node)Метод для поиска узла с предыдущим значением ключа перед указанным узлом.
   * public void ViewFromMin()Метод для просмотра содержимого дерева, начиная с узла с минимальным значением ключа.
   * public void ViewFromMax()Метод для просмотра содержимого дерева, начиная с узла с максимальным значением ключа.

Метод public void Insert(T value, int key)предназначен для вставки нового узла в дерево. Он принимает значение **value** (данные узла) и ключ **key** (используется для упорядочивания узлов в дереве). Вставка происходит следующим образом:

1. Создается новый узел **node** с переданными значением и ключом.
2. Если дерево пустое (корень **root** равен **null**), то новый узел становится корневым узлом и операция завершается.
3. Если в дереве уже есть узел с таким же ключом (**Find(node.key) != null**), то значение этого узла обновляется на новое переданное значение, и операция завершается.
4. Иначе происходит поиск места для вставки нового узла в дерево:
   * Стартовая позиция - корневой узел **currentNode**.
   * Если ключ текущего узла больше ключа вставляемого узла, двигаемся влево:
     + Если левый потомок **currentNode.left** отсутствует, то новый узел **node** становится левым потомком **currentNode**. Устанавливается ссылка на родительский узел **node.parent = currentNode**, и операция завершается.
     + Если левый потомок **currentNode.left** существует, двигаемся дальше влево, обновляя **currentNode = currentNode.left**.
   * Если ключ текущего узла меньше ключа вставляемого узла, двигаемся вправо:
     + Если правый потомок **currentNode.right** отсутствует, то новый узел **node** становится правым потомком **currentNode**. Устанавливается ссылка на родительский узел **node.parent = currentNode**, и операция завершается.
     + Если правый потомок **currentNode.right** существует, двигаемся дальше вправо, обновляя **currentNode = currentNode.right**.

Метод public void Delete(int key)предназначен для удаления узла с заданным ключом из дерева. Он работает следующим образом:

1. Находится узел **delete** с заданным ключом с помощью метода **Find(key)**.
2. Если узел **delete** не найден, операция завершается.
3. Если узел **delete** найден:
   * Если у узла **delete** нет потомков (левого и правого), то удаляем его, обновляя ссылку у родительского узла:
     + Если узел **delete** является корневым узлом, то устанавливаем **root = null**.
     + Иначе, проверяем, является ли узел **delete** левым или правым потомком родительского узла. Обновляем ссылку родительского узла на **null** соответствующего потомка.
   * Если у узла **delete** есть только один потомок (левый или правый), то заменяем узел **delete** его единственным потомком:
     + Если узел **delete** является корневым узлом, то устанавливаем **root** на его единственного потомка.
     + Иначе, проверяем, является ли узел **delete** левым или правым потомком родительского узла. Обновляем ссылку родительского узла на его единственного потомка.
   * Если у узла **delete** есть оба потомка (левый и правый), то находим наибольший узел в левом поддереве (узел с максимальным ключом):
     + Находим максимальный узел **maxNode** в левом поддереве узла **delete** с помощью метода **FindMaxNode(delete.left)**.
     + Копируем значение и ключ узла **maxNode** в узел **delete**.
     + Рекурсивно вызываем метод **Delete(maxNode.key)** для удаления узла **maxNode** из левого поддерева узла **delete**.
4. class Graph

Класс Graph является реализацией структуры данных для моделирования и представления графов. Он содержит ряд методов и свойств, позволяющих работать с вершинами и ребрами графа. Давайте рассмотрим основные аспекты класса Graph:

1. Поля класса Vertex:
   * **private static int IDV** - статическое поле для уникального идентификатора вершин.
   * **private int ID** - уникальный идентификатор текущей вершины.
   * **public string label** - метка или имя вершины.
   * **private List<Edge> edges** - список ребер, связанных с данной вершиной.
   * **public double sumdistance** - сумма расстояний.
   * **public COLORS\_VERTEX color** - цвет вершины, представленный в виде перечисления.
   * **public Vertex prevvertex** - ссылка на предшествующую вершину.
   * **public bool visited** - флаг, указывающий, была ли вершина посещена.
2. Поля класса Edge:
   * **public Vertex BeginPoint** - начальная вершина ребра.
   * **public Vertex EndPoint** - конечная вершина ребра.
   * **public double distance** - длина ребра.
3. Поля класса Graph:
   * **public List<Vertex> allVertexs** - список всех вершин в графе.
   * **public List<Edge> allEdges** - список всех ребер в графе.
4. Конструкторы:
   * **public Vertex(string label)** - создает новый экземпляр вершины с указанной меткой и инициализирует остальные поля.
   * **public Edge(Vertex begin, Vertex end, double d)** - создает новое ребро между начальной и конечной вершинами с указанной длиной.
   * **public Graph()** - создает новый пустой граф, инициализируя списки вершин и ребер.
5. Методы класса Vertex:
   * **public int GetID()** - возвращает уникальный идентификатор вершины.
   * **public List<Edge> GetEdges()** - возвращает список ребер, связанных с данной вершиной.
   * **public void ViewEdges()** - выводит информацию о ребрах, связанных с данной вершиной.
   * **public bool AddEdge(Edge edge)** - добавляет новое ребро к вершине, если оно корректно.
6. Методы класса Graph:
   * **public bool AddEdge(Vertex v1, Vertex v2, double d)** - добавляет новое ребро между двумя вершинами в граф, если они существуют.

Метод public void BFS(Vertex s)

1. Метод **BFS** принимает в качестве аргумента стартовую вершину **s** для обхода графа в ширину.
2. Создается пустая очередь **Q**, которая будет использоваться для хранения вершин, которые нужно обработать.
3. Инициализируются начальные значения для всех вершин графа:
   * Каждая вершина получает цвет **COLORS\_VERTEX.WHITE**, обозначающий, что она еще не была посещена.
   * Расстояние до каждой вершины устанавливается на бесконечность или на некоторое большое значение, чтобы показать, что они еще не были достигнуты.
   * Предыдущая вершина (**prevvertex**) устанавливается на **null**.
4. Стартовая вершина **s** помещается в очередь **Q**.
5. Начинается основной цикл, который будет выполняться, пока очередь **Q** не станет пустой.
6. На каждой итерации цикла извлекается вершина **u** из очереди **Q**.
7. Для каждой смежной вершины **v** для вершины **u** выполняются следующие действия:
   * Если вершина **v** еще не была посещена (её цвет равен **COLORS\_VERTEX.WHITE**), то она помечается как обрабатываемая (цвет **COLORS\_VERTEX.GRAY**).
   * Обновляется расстояние до вершины **v** (если требуется) и предыдущая вершина **prevvertex**, чтобы отразить путь от стартовой вершины **s** до вершины **v**.
   * Вершина **v** добавляется в очередь **Q** для дальнейшей обработки.
8. После перебора всех смежных вершин, текущая вершина **u** помечается как посещенная (цвет **COLORS\_VERTEX.BLACK**).
9. Цикл повторяется с шага 6, пока в очереди **Q** остаются вершины для обработки.
10. Когда все вершины графа были посещены, алгоритм завершается.

Метод public void DFS(Vertex startVertex)

1. Метод **DFS** принимает в качестве аргумента стартовую вершину **startVertex** для обхода графа в глубину.
2. Инициализируются цвета всех вершин графа:
   * Каждая вершина получает цвет **COLORS\_VERTEX.WHITE**, обозначающий, что она еще не была посещена.
3. Вызывается вспомогательная рекурсивная функция private void DFSVisit(Vertex currentVertex)с аргументом **startVertex** для начала обхода графа.
4. В функции **DFSVisit**:
   * Текущая вершина **currentVertex** помечается как посещаемая (цвет **COLORS\_VERTEX.GRAY**).
   * Происходит перебор всех ребер, выходящих из текущей вершины.
   * Для каждой следующей вершины **nextVertex** выполняются следующие действия:
     + Если вершина **nextVertex** еще не была посещена (её цвет равен **COLORS\_VERTEX.WHITE**), то она помечается как посещаемая (цвет **COLORS\_VERTEX.GRAY**).
     + Устанавливается текущая вершина **currentVertex** как предыдущая для **nextVertex**.
     + Рекурсивно вызывается функция **DFSVisit** для **nextVertex**.
   * После перебора всех ребер, текущая вершина **currentVertex** помечается как полностью посещенная (цвет **COLORS\_VERTEX.BLACK**).
5. Когда вся рекурсия завершается, алгоритм обхода графа в глубину завершается.

Метод public void Dijkstra(Vertex s)

1. Метод **Dijkstra** принимает в качестве аргумента стартовую вершину **s** для поиска кратчайшего пути в графе.
2. Инициализируется расстояние до всех вершин как бесконечность, кроме начальной вершины **s**, для которой расстояние устанавливается равным 0.
3. Создается множество **S** для хранения посещенных вершин.
4. Создается множество **V**, содержащее все вершины графа.
5. Алгоритм выполняется до тех пор, пока множество **S** не содержит все вершины.
6. На каждой итерации алгоритма выбирается вершина **u** из **V**, которая не принадлежит **S** и имеет минимальное расстояние.
7. Затем пересчитывается расстояние до всех соседних вершин **v**, которые не принадлежат **S**, через вершину **u**.
8. Если полученное расстояние меньше текущего расстояния до вершины **v**, то расстояние обновляется, а предыдущая вершина **prevvertex** для **v** устанавливается как **u**.
9. После пересчета расстояний для всех соседних вершин, текущая вершина **u** добавляется в множество **S**.
10. Шаги 6-9 повторяются до тех пор, пока все вершины не будут добавлены в множество **S**.
11. Когда все вершины были посещены и расстояния до них были определены, алгоритм Дейкстры завершается.

Метод public int CountSubgraphs()

1. Метод **CountSubgraphs** не принимает аргументов.
2. Инициализируется счетчик **subgraphsCount** на 0. Этот счетчик будет использоваться для подсчета количества подграфов в графе.
3. Создается пустой список **visitedVertices**, который будет использоваться для хранения уже посещенных вершин.
4. Происходит перебор всех вершин графа.
5. Для каждой текущей вершины **cv** выполняются следующие действия:
   * Если текущая вершина **cv** не содержится в списке **visitedVertices**, это означает, что текущая вершина еще не была посещена, и мы не начали обход из нее.
   * Вызывается метод **BFS(cv)** для запуска алгоритма обхода графа в ширину (BFS) из текущей вершины.
   * Все посещенные вершины в текущем запуске BFS добавляются в список **visitedVertices**.
   * Счетчик **subgraphsCount** увеличивается на 1, так как мы нашли новый подграф.
6. После завершения перебора всех вершин метод возвращает значение **subgraphsCount**, которое представляет количество подграфов в графе.
7. Алгоритм **CountSubgraphs** завершается.

**Список литературы**

1.C#. Алгоритмы и структуры данных. Н. А. Тюкачев, В. Г. Хлебостроев

2.Вирт Н. АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ. М.Мир 1989

3.Алгоритмы. Справочник. С примерами на C, C++, Java и Python. Джордж Хайнеман, Гэри Поллис, Стэнли Селков

4. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. / М.: Издательский дом «Вильямс»

5. Pikabu // Pikabu URL: https://pikabu.ru/story/kheshtablitsa\_hashtable\_na\_yazyike\_c\_5805617 (дата обращения: 10.04.2023).