**Algorithms and data**

**structures**

***Completed: Mishanya03***

***2023***

**Оглавление**

[**Работа с множеством** 3](#_Toc136527464)

[**Сортировки** 4](#_Toc136527465)

[**Сортировка пузырьком** 4](#_Toc136527466)

[**Сортировка - Поиск максимального элемента** 5](#_Toc136527467)

[**Сортировка методом вставки** 6](#_Toc136527468)

[**Сортировка Shell** 6](#_Toc136527469)

[**Сортировка Radix** 7](#_Toc136527470)

[**Сортировка слиянием с рекурсией и без** 8](#_Toc136527471)

[**Сортировка QuickSort** 9](#_Toc136527472)

[**Списки** 10](#_Toc136527473)

[**Односвязный список** 10](#_Toc136527474)

[**Двусвязный список** 12](#_Toc136527475)

[**Хэш-таблицы** 14](#_Toc136527476)

[**Граф** 15](#_Toc136527477)

[**Список литературы** 17](#_Toc136527478)

# **Работа с множеством**

class Set<T> where T : IComparable

public int GetSize() { return n; }

public int GetCount() { return size; }

public int Count { get { return size; } }

public int GetIndex(T value)

public bool IsContains(T element)

public void Add(T value)

public void RemoveAt(int index)

public void Remove(T value)

public T GetValue(int index)

public override string ToString()

public Set<T> Union(Set<T> ss)

public Set<T> Intersection(Set<T> ss)

public Set<T> Addition(Set<T> ss)

public static Set<T> operator +(Set<T> s1, Set<T> s2) => s1.Union(s2);

public static Set<T> operator \*(Set<T> s1, Set<T> s2) => s1.Intersection(s2);

public static Set<T> operator -(Set<T> s1, Set<T> s2) => s2.Addition(s1);

public List<Set<T>> subSets()

public List<Set<T>> Permutations()

Множество представляет собой структуру данных, которая содержит уникальные элементы в произвольном порядке. Множество отличается от других структур данных, таких как массив или список, тем, что не может содержать повторяющиеся элементы.

Множества обладают следующими свойствами:

1. Уникальность элементов: Множество не может содержать повторяющиеся элементы. Каждый элемент в множестве является уникальным.
2. Неупорядоченность: Элементы в множестве не имеют определенного порядка. Порядок, в котором элементы были добавлены в множество, не гарантируется и может быть изменен.
3. Доступ и модификация: Элементы в множестве доступны для чтения, но они не могут быть прямо изменены. Чтобы изменить множество, необходимо выполнить операции добавления или удаления элементов.
4. Проверка принадлежности: Множество позволяет эффективно проверять принадлежность элемента. Это одно из основных применений множеств - быстрая проверка наличия элемента в множестве.
5. Операции над множествами: Множество поддерживает операции объединения, пересечения, разности и симметрической разности, которые позволяют комбинировать и сравнивать множества.
6. Итерация: Множество можно перебирать в цикле, получая доступ к каждому элементу.
7. Размер: Множество имеет размер, который определяется количеством уникальных элементов в нем.

Метод Union(Set<T> ss) возвращает новое множество, содержащее все элементы, которые присутствуют как в текущем множестве, так и в множестве ss.

Метод Intersection(Set<T> ss) возвращает новое множество, содержащее только те элементы, которые присутствуют как в текущем множестве, так и в множестве ss.

Метод Addition(Set<T> ss) возвращает новое множество, содержащее все элементы, которые присутствуют только в текущем множестве, но не в множестве ss.

Операторы +, \*, и - являются перегрузками операторов и обеспечивают альтернативный способ вызова методов Union, Intersection и Addition соответственно.

Метод subSets() возвращает список всех подмножеств текущего множества.

Метод Permutations() генерирует все возможные перестановки элементов множества.

Сначала в методе создается массив nums, содержащий индексы элементов текущего множества. Затем определяется максимальное количество перестановок для данного множества, которое равно факториалу размера множества.

Затем метод Permute(nums) вызывается рекурсивно для получения всех возможных перестановок элементов множества. Метод Permute(nums) работает следующим образом:

* Если массив nums пустой, то создается новый пустой список и добавляется в результирующий список.
* Если массив nums не пустой, то для каждого элемента в массиве выполняются следующие шаги:
  + Элемент сохраняется в переменной currentNum.
  + Создается новый массив remainingNums, который содержит все элементы из nums, за исключением currentNum.
  + Рекурсивно вызывается метод Permute() для массива remainingNums.
  + К каждой полученной перестановке добавляется currentNum в начало списка.
  + Перестановки добавляются в результирующий список.

После выполнения Permute(nums), полученные перестановки сохраняются в списке listIndex. Затем создается пустой список result, который будет содержать все перестановки множества. Для каждой перестановки из listIndex создается новое множество, которое заполняется элементами из текущего множества data, используя соответствующие индексы из перестановки. Созданное множество добавляется в список result.

По завершении работы метода Permutations(), список result содержит все возможные перестановки элементов множества.

Метод Permute является вспомогательной функцией, которая используется в методе Permutations, который возвращает список всех перестановок элементов в заданном множестве.

Permute работает рекурсивно. Она принимает на вход массив nums, который содержит целые числа, представляющие индексы элементов в множестве. Этот массив сначала проверяется на пустоту. Если он пустой, то создается и добавляется пустой список в список результатов и возвращается этот список.

Если массив не пустой, то происходит итерация по всем его элементам. Для каждого элемента создается новый массив remainingNums, который содержит все элементы nums кроме текущего. Затем для каждого такого массива remainingNums вызывается рекурсивно Permute, чтобы получить список всех перестановок элементов, которые остались после удаления текущего элемента.

Далее каждая перестановка в списке remainingPermutations обновляется путем добавления текущего элемента в начало этой перестановки, а затем добавляется в список результатов. В конце метод Permute возвращает список всех перестановок элементов в множестве.

Все методы основаны на обходе и модификации данных в текущем множестве и множестве-аргументе, а также на создании новых множеств с помощью операций над элементами.

# **Сортировки**

class Sort

## **Сортировка пузырьком**

public static void BubbleSort<T>(T[] list) where T : IComparable

Идея сортировки пузырьком заключается в сравнении и перестановке соседних элементов списка до тех пор, пока список не будет отсортирован. Она получила название "пузырьком" из-за свойства меньших элементов "всплывать" к верхней части списка по мере прохождения через него.

Вот основные шаги, описывающие идею сортировки пузырьком:

1. Проходим по списку элементов, начиная с начала.
2. Сравниваем текущий элемент с его соседом справа.
3. Если текущий элемент больше следующего, меняем их местами.
4. Повторяем шаги 2-3 для каждой пары соседних элементов, пока не пройдем весь список.
5. После первого прохода самый большой элемент окажется в конце списка.
6. Повторяем шаги 1-5 для оставшейся части списка (без последнего элемента).
7. Продолжаем повторять шаги 1-6, пока список не будет полностью отсортирован.

В результате каждого прохода по списку наибольший элемент "всплывает" к правой части, что приводит к постепенному упорядочиванию всех элементов списка.

Сортировка пузырьком является простым, но не самым эффективным алгоритмом сортировки. В некоторых случаях он может быть заметно медленнее более эффективных алгоритмов, особенно для больших списков данных.

Алгоритм сортировки пузырьком заключается в последовательном проходе по массиву и сравнении каждого элемента с его соседом справа, и если они не упорядочены, то происходит обмен этих элементов местами. Этот процесс продолжается до тех пор, пока в массиве не останется элементов, которые нужно отсортировать.

Метод принимает массив объектов типа T и сравнивает элементы в массиве, используя метод CompareTo. Для того чтобы использовать этот метод, объекты типа T должны реализовывать интерфейс IComparable.

Переменная N хранит длину массива, а переменная numberOfPairs уменьшается на 1 после каждого прохода по массиву. Переменная swappedElements равна true, если произошла перестановка элементов в массиве, и false в противном случае.

Алгоритм завершается, когда swappedElements равна false, что означает, что произошло полное прохождение по массиву без перестановок элементов.

В итоге, метод "BubbleSort" сортирует переданный ему массив объектов типа T в порядке возрастания путем сравнения каждого элемента с его соседом и перестановки их местами при необходимости.

## **Сортировка - Поиск максимального элемента**

public static void SearchMaxElement<T>(T[] list) where T : IComparable

Идея сортировки поиска максимального элемента состоит в том, чтобы последовательно находить максимальный элемент в неотсортированной части массива и перемещать его в конец этой части. Таким образом, массив сортируется путем построения отсортированной части справа направо.

Вот шаги этого алгоритма:

1. Начинаем с неотсортированного массива.
2. Ищем максимальный элемент в неотсортированной части массива.
3. Перемещаем найденный максимальный элемент в конец неотсортированной части.
4. Сокращаем неотсортированную часть массива, исключая последний элемент, который уже отсортирован.
5. Повторяем шаги 2-4, пока неотсортированная часть массива не будет пустой.

Этот процесс повторяется, пока не весь массив не будет отсортирован. После завершения алгоритма элементы будут упорядочены в порядке возрастания.

Метод поиска максимального элемента на каждом шаге является одним из способов выбора элемента для упорядочивания. Однако, этот подход не является самым эффективным с точки зрения временной сложности сортировки. На практике обычно используются более оптимизированные алгоритмы сортировки, такие как быстрая сортировка (quicksort), сортировка слиянием (merge sort) или сортировка пузырьком (bubble sort).

Метод "SearchMaxElement" реализует алгоритм сортировки выбором, который находит наибольший элемент в списке и перемещает его в конец, затем повторяет этот процесс для оставшихся элементов.

Аргумент метода "list" представляет собой массив элементов типа T, где T реализует интерфейс "IComparable", который определяет порядок сравнения элементов.

В начале метода определяется длина массива "N" и переменная "numberOfPairs", которая инициализируется значением "N". Затем цикл "for" выполняется "N" раз, каждый раз уменьшая значение "numberOfPairs".

Внутри цикла "for" происходит поиск наибольшего элемента среди первых "numberOfPairs" элементов массива. При этом переменная "maxElement" инициализируется значением первого элемента в массиве, а "maxElementIndex" устанавливается в 0. Затем происходит сравнение каждого элемента массива с "maxElement", и если текущий элемент больше "maxElement", то "maxElement" обновляется и "maxElementIndex" устанавливается на индекс этого элемента.

После прохода по всем элементам массива, если "maxElementIndex" не равен "numberOfPairs", то наибольший элемент меняется местами с последним элементом, используя временную переменную "tmp".

Таким образом, на каждом шаге цикла "for" наибольший элемент перемещается в конец массива, пока все элементы не будут отсортированы в порядке возрастания.

## **Сортировка методом вставки**

public static void InsertionSort<T>(T[] list) where T : IComparable

Идея сортировки методом вставки (Insertion Sort) заключается в следующем:

1. Сортировка начинается с первого элемента массива (или списка) и предполагается, что этот элемент уже отсортирован.
2. Берется следующий элемент и вставляется в отсортированную часть массива, сдвигая все большие элементы вправо, чтобы освободить место для вставки.
3. Этот процесс повторяется для каждого следующего элемента до тех пор, пока весь массив не будет отсортирован.

Иллюстрация шагов сортировки методом вставки:

1. Исходный массив: [5, 2, 4, 6, 1, 3]
2. Начиная со второго элемента (2), он сравнивается с предыдущим элементом (5). Поскольку 2 меньше 5, элементы сдвигаются вправо, и 2 вставляется на его место.

Массив после первого шага: [2, 5, 4, 6, 1, 3]

1. Следующий элемент (4) сравнивается с предыдущими элементами (5 и 2). 4 вставляется на свое место между 2 и 5.

Массив после второго шага: [2, 4, 5, 6, 1, 3]

1. Аналогично, элементы 6, 1 и 3 сравниваются и вставляются на свои места.

Массив после третьего шага: [1, 2, 3, 4, 5, 6]

1. В итоге, весь массив отсортирован в порядке возрастания.

Сортировка методом вставки проста в реализации и хорошо работает на небольших массивах или в случаях, когда массив уже частично отсортирован. Однако, для больших массивов она может быть неэффективной, так как имеет временную сложность O(n^2) в худшем случае.

Метод "InsertionSort" реализует алгоритм сортировки вставками для массива элементов типа T, который должен быть сравнимым (реализовывать интерфейс IComparable).

Алгоритм сортировки вставками проходится по массиву, начиная со второго элемента и до последнего. На каждой итерации алгоритма текущий элемент сравнивается со всеми предыдущими элементами, которые стоят перед ним. Если текущий элемент меньше предыдущего, то предыдущий элемент сдвигается на одну позицию вправо, чтобы освободить место для текущего элемента. Этот процесс продолжается, пока не будет найдено место для текущего элемента. Затем текущий элемент вставляется на найденную позицию.

Таким образом, на каждой итерации алгоритма увеличивается упорядоченная часть массива, а неупорядоченная часть уменьшается, пока вся последовательность не будет упорядочена.

В итоге, после выполнения метода "InsertionSort", элементы в переданном массиве будут отсортированы в порядке возрастания, если тип T реализует интерфейс IComparable и имеет реализацию метода CompareTo.

## **Сортировка Shell**

public static void ShellSort<T>(T[] list) where T : IComparable

Сортировка Shell (или сортировка Шелла) является модификацией сортировки вставками, которая улучшает её производительность путем упорядоченной предварительной сортировки элементов внутри массива. Идея сортировки Shell состоит в сравнении и перемещении элементов, находящихся на определенном расстоянии друг от друга.

Вот основная идея сортировки Shell:

1. Задается интервал (шаг) сортировки. Этот интервал определяет, на каком расстоянии друг от друга сравниваются и перемещаются элементы в массиве.
2. Начиная с самого большого значения интервала, происходит сравнение элементов, находящихся на расстоянии интервала друг от друга, и при необходимости они меняются местами.
3. Затем интервал сортировки сокращается, и процесс сравнения и перемещения элементов повторяется.
4. Шаги 2-3 повторяются до тех пор, пока интервал сортировки не станет равным 1. В этот момент выполняется обычная сортировка вставками для упорядочивания оставшихся элементов.

Идея сортировки Shell заключается в том, чтобы "предварительно" упорядочить элементы массива с большими шагами, что помогает уменьшить количество сравнений и перемещений, которые требуются для финальной сортировки вставками.

Сортировка Шелла является алгоритмом с переменным интервалом и зависит от выбора последовательности интервалов. Существует несколько различных последовательностей интервалов, которые могут быть использованы в сортировке Шелла, такие как последовательность Шелла, последовательность Пратта и др.

Сортировка Шелла обеспечивает более быструю сортировку по сравнению с обычной сортировкой вставками, особенно для больших массивов. Однако её анализ и аналитические оценки эффективности являются сложными задачами из-за зависимости от выбора последовательности интервалов.

Метод "ShellSort" реализует алгоритм сортировки Шелла. Этот алгоритм является усовершенствованной версией сортировки вставками. Он работает следующим образом:

1. На первом шаге, величина шага (step) равна длине массива деленной на 2.
2. Затем сравниваются элементы массива на расстоянии шага друг от друга. Если элементы стоят в неправильном порядке, они меняются местами.
3. После этого шаг уменьшается вдвое и процесс повторяется до тех пор, пока шаг не станет равен 1.
4. На последнем шаге происходит сортировка вставками, т.е. каждый элемент сравнивается с предыдущим, и если элемент стоит не на своем месте, то он меняется местами с предыдущим элементом.

Таким образом, сортировка Шелла использует идею сортировки вставками, но меняет порядок элементов до того, как происходит финальная сортировка вставками. Это позволяет сортировать массивы более эффективно.

## **Сортировка Radix**

public static void RadixSort(int[] list)

Сортировка по основанию (Radix sort) является алгоритмом сортировки, который сортирует элементы путем последовательной сортировки по разрядам чисел или других элементов данных. Он основан на принципе сортировки от младшего разряда к старшему.

Идея сортировки Radix заключается в том, чтобы сначала сортировать элементы по наименее значащему разряду (обычно самому правому разряду) и продолжать сортировку по более значащим разрядам до наиболее значащего разряда.

Шаги сортировки Radix:

1. Изначально формируется массив или список элементов для сортировки.
2. Начиная с наименее значащего разряда, выполняется сортировка элементов по этому разряду. Это может быть выполнено с помощью стабильной сортировки, такой как сортировка подсчетом или сортировка вставками.
3. После сортировки по текущему разряду, элементы распределяются в соответствующие группы или корзины, основываясь на значении текущего разряда.
4. Затем элементы из корзин объединяются в один массив или список в порядке их появления в корзинах.
5. Процесс повторяется для следующего разряда, пока не будут отсортированы все разряды элементов.
6. По окончании сортировки всех разрядов элементы находятся в правильном порядке.

Важным аспектом сортировки Radix является использование стабильных сортировок на каждом шаге для сохранения порядка элементов с одинаковыми значениями разряда. Количество проходов по всем разрядам зависит от количества разрядов в элементах и основания, которое используется (обычно основание равно 10 для десятичных чисел).

Сортировка по основанию эффективна для сортировки чисел или других элементов с фиксированным количеством разрядов. Однако она может потребовать больше памяти, если элементы занимают много места, и может быть неэффективной для элементов с переменной длиной, таких как строки.

Общая сложность сортировки Radix составляет O(d \* (n + b)), где d - количество разрядов, n - количество элементов, b - основание.

Этот метод называется "RadixSort" (сортировка по разрядам) и представляет собой алгоритм сортировки целочисленного массива. Он работает следующим образом:

1. Находится максимальное число в массиве, чтобы определить количество цифр в наибольшем числе.

2. Проверяется, есть ли отрицательные числа в массиве.

3. Если отрицательных чисел нет, то начинается процесс сортировки.

4. В цикле по количеству цифр в наибольшем числе происходит следующее:

- Создаются 10 ведер (buckets), каждое из которых соответствует определенной цифре (от 0 до 9).

- Каждое число из массива разбивается на цифры по порядку (начиная с младшего разряда) и помещается в соответствующее ведро.

- Числа из ведер собираются обратно в массив в порядке от младших разрядов к старшим.

- После прохода по всем цифрам увеличивается делитель del на порядок 10.

5. Если в массиве были отрицательные числа, то метод завершает работу без сортировки.

Таким образом, RadixSort сортирует массив по каждой цифре числа, начиная с младшей и заканчивая старшей, используя ведерную сортировку. Это позволяет сортировать массивы с очень большим количеством элементов за время O(nk), где n - количество элементов в массиве, а k - максимальное количество цифр в числе.

## **Сортировка слиянием с рекурсией и без**

public static void MergeSort<T>(T[] list, int left, int right) where T : IComparable

public static void MergeSortNR<T>(T[] list) where T : IComparable

Идея сортировки слиянием (Merge Sort) состоит в разделении массива на две половины, рекурсивной сортировке каждой половины, а затем слиянии отсортированных половин в один отсортированный массив.

Вот основные шаги алгоритма сортировки слиянием:

1. Разделение: Разделите исходный массив на две примерно равные половины.
2. Рекурсивная сортировка: Рекурсивно примените сортировку слиянием к каждой половине массива, пока не достигнете базового случая, когда размер массива становится равным 1 или 0 (считается, что одиночный элемент или пустой массив уже отсортированы).
3. Слияние: Слейте отсортированные половины массива в один новый массив, сохраняя упорядоченность элементов. При слиянии сравнивайте элементы из каждой половины и помещайте их в правильном порядке в новый массив. После этого скопируйте оставшиеся элементы из каждой половины, если таковые имеются, в новый массив.
4. Возврат результата: Верните отсортированный массив.

Преимуществом сортировки слиянием является ее стабильность (порядок равных элементов сохраняется) и эффективность для больших массивов. Она обладает асимптотической сложностью O(n log n) в худшем и среднем случаях, что делает ее хорошим выбором для сортировки больших объемов данных.

Первый метод "MergeSort" представляет рекурсивную версию сортировки. Он принимает массив элементов и два индекса - левый (left) и правый (right) границы массива. Первоначально проверяется условие, что левая граница меньше правой, чтобы убедиться, что массив не пустой. Затем вычисляется средняя точка и вызывается рекурсивно тот же метод для двух половинок массива, слева и справа от средней точки. Наконец, вызывается метод "MergeLists", который объединяет две половинки массива в отсортированный массив.

Второй метод "MergeSortNR" представляет итеративную версию сортировки. Он также принимает массив элементов, но не принимает границы. В этом методе используются два цикла - внешний цикл для увеличения размера текущей группы элементов (начинается с единицы и удваивается на каждой итерации), а внутренний цикл для объединения двух подмассивов. В цикле объединения используется метод "MergeLists".

Метод "MergeLists" принимает три индекса - левый (left), средний (middle) и правый (right) - которые определяют границы двух подмассивов. Внутри метода создаются два новых массива - для левой и правой половинок, заполняются элементами из исходного массива. Затем происходит объединение двух массивов в один отсортированный массив, путем сравнения элементов исходных массивов и помещения их в правильном порядке в результирующий массив. В конце метода, если один из массивов был полностью обработан, оставшиеся элементы копируются в результирующий массив.

Во всех методах используется ограничение на тип элементов (where T : IComparable), чтобы элементы можно было сравнивать между собой.

## **Сортировка QuickSort**

public static void QuickSort<T>(T[] list, int leftIndex, int rightIndex) where T : IComparable

QuickSort (Быстрая сортировка) - это один из самых известных алгоритмов сортировки, основанный на принципе "разделяй и властвуй". Он основан на сравнении элементов массива и их последующей перестановке таким образом, чтобы элементы слева от выбранного опорного элемента были меньше него, а элементы справа - больше него. После этого процесс повторяется для обоих получившихся подмассивов, пока весь массив не будет отсортирован.

Идея алгоритма QuickSort:

1. Выбирается опорный элемент из массива. Это может быть любой элемент, например, первый, последний или случайно выбранный элемент массива.
2. Все элементы, меньшие или равные опорному элементу, перемещаются налево от него, а все элементы, большие опорному элементу, перемещаются направо от него. Этот процесс называется разделением (partitioning) массива.
3. Применяется рекурсивно QuickSort к левому подмассиву (элементы слева от опорного элемента) и правому подмассиву (элементы справа от опорного элемента).
4. Рекурсия продолжается до тех пор, пока подмассивы не будут состоять из одного элемента или будут пустыми.
5. После завершения рекурсии весь массив будет отсортирован.

Процесс разделения (partitioning) основан на прохождении двух указателей через массив в противоположных направлениях. Один указатель идет с начала массива, а другой - с конца. При нахождении элементов, которые находятся не на своих местах, они меняются местами. Это продолжается, пока указатели не встретятся.

QuickSort обладает хорошей производительностью в среднем случае, но в худшем случае может иметь квадратичную сложность времени. Однако, благодаря случайному выбору опорного элемента, вероятность худшего случая крайне мала, и обычно QuickSort работает очень быстро на практике.

Метод "QuickSort" реализует алгоритм быстрой сортировки, который принимает массив элементов типа T, левый и правый индексы.

Алгоритм быстрой сортировки работает следующим образом: сначала выбирается опорный элемент (pivot), который используется для разделения массива на две части - элементы, которые меньше опорного, и элементы, которые больше опорного. Затем элементы каждой из частей сортируются рекурсивно, пока не будет достигнут конечный результат.

Код метода "QuickSort" начинается с инициализации переменных. Затем определяется опорный элемент tmp, который равен левому элементу в списке. Далее, в цикле while, происходит сравнение элементов в списке с опорным элементом tmp. Если элементы в списке меньше tmp, то i увеличивается на 1, если же элементы в списке больше tmp, то j уменьшается на 1. Если i <= j, то происходит обмен местами элементов i и j, а затем i увеличивается на 1 и j уменьшается на 1.

После завершения цикла while, вызываются рекурсивно методы QuickSort для левой и правой части списка, если левый индекс меньше j, и правый индекс больше i соответственно. Это продолжается до тех пор, пока не будет отсортирован весь массив.

Таким образом, метод QuickSort выполняет быструю сортировку элементов массива, разделяя его на две части и рекурсивно вызывая сам себя для каждой из этих частей, пока не будет достигнут конечный результат.

# **Списки**

## **Односвязный список**

class singleNode<K, T> : Item<K, T>

class singleLinkedList<K, T> where K : IComparable where T : IComparable

Односвязный список (Single linked list) - это структура данных, состоящая из узлов, каждый из которых содержит какое-то значение (например, число, строку или ссылку на другой объект) и ссылку на следующий узел в списке. Последний узел списка имеет ссылку на NULL или пустую ссылку.

Односвязный список является динамической структурой данных, что означает, что его размер может изменяться во время выполнения программы, и узлы могут быть добавлены или удалены по мере необходимости.

Идея односвязного списка заключается в том, что каждый узел содержит ссылку только на следующий узел в списке, а не на предыдущий. Это отличает его от двусвязного списка, который содержит ссылки на оба соседних узла.

В односвязном списке элементы располагаются последовательно, и каждый элемент имеет указатель на следующий элемент. Таким образом, элементы не обязательно расположены в памяти последовательно, что позволяет динамически изменять размер списка.

Односвязный список используется для решения различных задач, таких как реализация стека или очереди, поиска элементов, упорядочивания и т. д.

Класс singleNode<K, T> представляет узел односвязного списка и содержит свойства и методы для работы с узлом. Класс основан на классе Item<K, T>, который является информационной частью узла односвязного списка.

Методы класса singleNode<K, T>:

* Конструктор singleNode(K key, T value): создает новый объект singleNode<K, T> с заданным ключом key и значением value.
* Конструктор singleNode(): создает новый пустой объект singleNode<K, T>.
* Свойство Next: получает или устанавливает ссылку на следующий узел в списке.
* Метод ToString(): возвращает строковое представление узла.

Класс singleLinkedList<K, T> представляет односвязный список и содержит свойства и методы для работы со списком.

public int AddBegin(K key, T value)

public int AddEnd(K key, T value)

public void Clear()

public bool ContainsValue(T value)

public bool ContainsKey(K key)

public singleNode<K, T> getNode(int k)

public int InsertAfterNode(K key, K newKey, T Value)

public int InsertBeforeNode(K key, K newKey, T Value)

public void RemoveFirstNode()

public void RemoveLastNode()

public void RemoveAt(int index)

public void Remove(K key)

public void PrintList()

Методы класса singleLinkedList<K, T>:

* Конструктор singleLinkedList(): создает новый пустой объект singleLinkedList<K, T>.
* Свойство First: получает ссылку на первый узел в списке.
* Свойство Count: получает количество узлов в списке.
* Метод AddBegin(K key, T value) добавляет новый узел в начало односвязного списка. Создается новый узел типа singleNode<K, T> с переданными ключом key и значением value. Затем устанавливается ссылка Next нового узла на текущий первый узел списка, а ссылка first устанавливается на новый узел. В конце метод возвращает новую позицию в списке.
* Метод AddEnd(K key, T value) добавляет новый узел в конец односвязного списка. Создается новый узел типа singleNode<K, T> с переданными ключом key и значением value. Если список пустой, то ссылка first устанавливается на новый узел и метод возвращает новую позицию в списке. В противном случае, происходит поиск последнего узла в списке, затем устанавливается ссылка Next последнего узла на новый узел. В конце метод возвращает новую позицию в списке.
* Метод Clear() очищает односвязный список, устанавливая ссылку first на null и сбрасывая позицию списка pos на 0.
* Метод ContainsValue(T value) проверяет, содержится ли значение value в списке. Если список не пустой, происходит перебор узлов до тех пор, пока не будет найден узел со значением, равным переданному value. Если такой узел найден, метод возвращает true, в противном случае возвращается false.
* Метод ContainsKey(K key) проверяет, содержится ли ключ key в списке. Если список не пустой, происходит перебор узлов до тех пор, пока не будет найден узел с ключом, равным переданному key. Если такой узел найден, метод возвращает true, в противном случае возвращается false.
* Метод getNode(int k) возвращает узел списка с заданным индексом k. Если список пуст или индекс k больше или равен количеству узлов в списке, метод вернет null. В противном случае метод будет последовательно проходить по списку, начиная с первого узла, пока не достигнет узла с индексом k и вернет его.
* Метод InsertAfterNode(K key, K newKey, T Value) вставляет новый узел после узла с заданным ключом key. Если список пуст или ключ key не найден в списке, новый узел будет добавлен в конец списка. В противном случае метод будет последовательно проходить по списку, начиная с первого узла, пока не найдет узел с ключом key. Затем он вставляет новый узел после найденного узла и возвращает новую позицию нового узла в списке.
* Метод InsertBeforeNode(K key, K newKey, T Value) вставляет новый узел перед узлом с заданным ключом key. Если список пуст или ключ key не найден в списке, новый узел будет добавлен в конец списка. Если узел с ключом key является первым узлом списка, новый узел будет добавлен в начало списка. В противном случае метод будет последовательно проходить по списку, начиная с первого узла, пока не найдет узел с ключом key. Затем он вставляет новый узел перед найденным узлом и возвращает новую позицию нового узла в списке.
* Метод RemoveFirstNode() удаляет первый узел из списка. Если первый узел равен null, то метод ничего не делает. В противном случае, метод обновляет first на first.Next и уменьшает значение pos на 1.
* Метод RemoveLastNode() удаляет последний узел из списка. Если первый узел равен null, то метод ничего не делает. Если первый узел имеет следующий узел равный null, то first устанавливается в null. В противном случае, метод проходит по всем узлам списка до последнего и обновляет Next узла перед последним на null. Затем метод уменьшает значение pos на 1.
* Метод RemoveAt(int index) удаляет узел из списка по его индексу. Если индекс меньше 0, то метод ничего не делает. В противном случае, метод проходит по всем узлам списка до узла с индексом index - 1. Если currentNode.Next равен null, то метод ничего не делает. В противном случае, метод обновляет Next узла currentNode на currentNode.Next.Next и уменьшает значение pos на 1.
* Метод Remove(K key) удаляет узел из списка по его ключу. Если ключ первого узла равен key, то вызывается метод RemoveFirstNode(). Если ключ узла, следующего за currentNode, равен key, то метод обновляет Next узла currentNode на currentNode.Next.Next. Если узел со значением key является последним узлом, то вызывается метод RemoveLastNode(). Если узел со значением key не найден в списке, то метод ничего не делает.
* Метод PrintList() выводит содержимое списка на консоль. Он проходит по всем узлам списка, начиная с первого узла, и выводит информацию о каждом узле на консоль.

## **Двусвязный список**

class doubleNode<K, T>

class doubleLinkedList<K, T> where K : IComparable where T : IComparable

Двусвязный список (или двунаправленный список) - это структура данных, состоящая из узлов, каждый из которых содержит значение и две ссылки (указателя) на предыдущий и следующий узлы.

Идея двусвязного списка заключается в том, что каждый узел имеет две ссылки: одну на предыдущий узел и одну на следующий узел. Это позволяет обойти список в обоих направлениях, начиная с любого узла. Первый и последний узлы в списке могут быть определены как граничные, так как они имеют ссылки на предыдущий и следующий узлы соответственно, но могут также иметь значение NULL, если список пуст.

Преимущества двусвязного списка:

1. Возможность обращаться к элементам списка в обоих направлениях.
2. Вставка и удаление элементов выполняются быстро, так как для них не требуется перестраивание ссылок на другие элементы.
3. Можно легко удалять последний элемент списка.

Недостатки двусвязного списка:

1. Расход памяти. Каждый узел двусвязного списка содержит две ссылки, что требует дополнительной памяти по сравнению с односвязным списком.
2. Дополнительная сложность при реализации и поддержке структуры данных.

Двусвязные списки часто применяются, когда требуется эффективное вставление и удаление элементов в середине списка или при необходимости обхода списка в обоих направлениях.

Класс doubleLinkedList реализует двусвязный список (double linked list), который состоит из объектов типа doubleNode<K,T>. Класс имеет методы добавления элементов в начало и конец списка, вставки элемента после заданного элемента, а также методы для проверки наличия значения и ключа в списке.

Класс doubleNode<K,T> представляет узел двусвязного списка и имеет ссылки на следующий и предыдущий узлы, а также значение типа T и ключ типа K.

public int AddBegin(K key, T value)

public int AddEnd(K key, T value)

public void Clear()

public bool ContainsValue(T value)

public bool ContainsKey(K key)

public doubleNode<K, T> getNode(int k)

public int InsertAfterNode(K key, K newKey, T Value)

public int InsertBeforeNode(K key, K newKey, T Value)

public void RemoveFirstNode()

public void RemoveLastNode()

public void RemoveAt(int index)

public void Remove(K key)

public void PrintDoubleNodeNext()

public void PrintDoubleNodePrev()

В конструкторе класса doubleLinkedList инициализируются ссылки на первый и последний узлы и переменная pos, которая хранит количество элементов в списке.

* Метод AddBegin добавляет новый узел в начало списка. Если список не пустой, то новый узел становится первым элементом, а ссылка на предыдущий узел первого элемента указывает на новый узел. Если список пустой, то новый узел становится первым и последним элементом.
* Метод AddEnd добавляет новый узел в конец списка. Если список не пустой, то новый узел становится последним элементом, а ссылка на следующий узел последнего элемента указывает на новый узел. Если список пустой, то новый узел становится первым и последним элементом.
* Метод Clear очищает список.
* Метод ContainsValue проверяет, содержит ли список указанное значение.
* Метод ContainsKey проверяет, содержит ли список указанный ключ.
* Метод getNode возвращает узел списка по указанному индексу.
* Метод InsertAfterNode вставляет новый узел после указанного узла. Если список пустой, то новый узел становится первым и последним элементом. Если указанный узел последний в списке, то новый узел становится последним элементом. Если указанный узел не найден, метод возвращает -1.
* Метод InsertAfterNode(K key, K newKey, T Value) вставляет новый узел с ключом newKey и значением Value после узла с заданным ключом key. Сначала создается новый узел e, затем происходит поиск узла с ключом key. Если такой узел найден, то происходит вставка нового узла после него. Если узел с ключом key не найден, то новый узел добавляется в конец списка с помощью метода AddEnd. После вставки индекс списка pos увеличивается на 1.
* Метод InsertBeforeNode(K key, K newKey, T Value) работает аналогично предыдущему методу, но вставляет новый узел перед узлом с заданным ключом key.
* Метод RemoveFirstNode() удаляет начальный узел списка. Если список пуст, то метод ничего не делает. Если список не пуст, то начальный узел удаляется, первый узел списка становится следующим за удаленным, и индекс списка pos уменьшается на 1.
* Метод RemoveLastNode() удаляет конечный узел списка. Если список пуст, то метод ничего не делает. Если список не пуст, то конечный узел удаляется, последний узел списка становится предыдущим от удаленного, и индекс списка pos уменьшается на 1.
* Метод RemoveAt удаляет узел по заданному индексу index. Если индекс меньше 0, метод просто возвращает управление. Если индекс больше длины списка, метод также возвращает управление. Затем он перебирает узлы списка, начиная с первого, пока не достигнет узла с индексом index - 1. Если следующий узел после текущего узла не равен null, то текущий узел ссылается на узел, следующий за следующим узлом, и следующий узел, следующий за удаляемым узлом, ссылается на текущий узел. Длина списка уменьшается на 1.
* Метод Remove удаляет узел с указанным ключом key. Если первый и последний узлы списка имеют заданный ключ, они удаляются с помощью методов RemoveFirstNode и RemoveLastNode. Затем метод перебирает узлы списка, начиная с первого, пока не найдет узел со значением ключа, равным key. Если такой узел найден, он удаляется из списка таким же образом, как и в методе RemoveAt. Если удаляемый узел был последним узлом списка, он удаляется с помощью метода RemoveLastNode. Длина списка уменьшается на 1.
* Метод PrintDoubleNodeNext выводит список в прямом порядке, начиная с первого узла. Он перебирает узлы списка, начиная с первого, пока не достигнет конца списка. Для каждого узла метод выводит ссылку на него и переходит к следующему узлу.
* Метод PrintDoubleNodePrev выводит список в обратном порядке, начиная с последнего узла. Он перебирает узлы списка, начиная с последнего, пока не достигнет начала списка. Для каждого узла метод выводит ссылку на него и переходит к предыдущему узлу.

# **Хэш-таблицы**

class HashTableArray<K, T> where K : IComparable where T : IComparable

class HashTableList<K, T> where K : IComparable where T : IComparable

Хэш-таблица - это структура данных, используемая для хранения и организации пар ключ-значение. Она обеспечивает эффективный доступ и быстрые операции вставки, удаления и поиска элементов.

Идея хэш-таблицы заключается в использовании хэш-функции для преобразования ключей в индексы массива. Хэш-функция принимает ключ и вычисляет хэш-код - числовое значение, которое служит в качестве индекса в массиве. Хэш-код используется для определения места, где будет храниться значение в массиве, называемом бакетом или ячейкой хэш-таблицы.

Преимущество хэш-таблицы заключается в том, что она позволяет выполнять операции вставки, удаления и поиска элементов в среднем за постоянное время O(1), независимо от размера данных. Однако в худшем случае (когда возникают коллизии) время выполнения операций может быть O(n), где n - количество элементов в хэш-таблице.

Классы HashTableArray и HashTableList представляют реализацию хэш-таблиц, которые используются для хранения пар ключ-значение.

Класс HashTableArray использует массив для хранения элементов таблицы. В конструкторе создается массив заданного размера, который инициализируется null. Каждый элемент массива - объект типа Item, который содержит пару ключ-значение. Класс имеет методы для добавления элементов, поиска элементов по ключу, удаления элементов и изменения размера таблицы. При достижении определенного количества элементов, размер таблицы увеличивается в два раза.

Класс HashTableList использует массив списков для хранения элементов таблицы. В конструкторе создается массив заданного размера, который инициализируется пустыми списками. Каждый элемент списка - объект типа Item, который содержит пару ключ-значение. Класс имеет методы для добавления элементов, поиска элементов по ключу и изменения размера таблицы. При добавлении нового элемента, происходит поиск по списку с ключами, если ключ уже существует, значение заменяется.

Оба класса используют хэш-функцию для вычисления индекса массива или списка, в который будет добавлен новый элемент. Метод GetIndex принимает ключ и возвращает индекс в массиве или списке. Хэш-функция вычисляется с помощью метода GetHashCode для ключа.

Оба класса используют интерфейс IComparable для сравнения ключей и значений. Это позволяет классу определять порядок следования элементов и правильно выполнять поиск и сортировку.

Оба класса содержат свойства Count и Size, которые возвращают количество элементов и размер таблицы соответственно.

Класс HashTableArray подходит для малых таблиц, где количество элементов не превышает размера таблицы. Класс HashTableList подходит для больших таблиц, где количество элементов может превышать размер таблицы.

public int HashCode(K key)

public int GetIndex(K key)

public int SearchByKey(K key)

public Item<K, T> SearchItemByKey(K key)

public int Add(K key, T value)

public bool DeleteByKey(K key)

public void Resize(int newsize)

public Item<K, T>[] ToArray()

public void Print()

Методы обоих классов реализуют основные операции, которые можно производить с хэш-таблицами:

1. HashCode(K key): вычисляет хэш-код для заданного ключа key.
2. GetIndex(K key): вычисляет индекс элемента в хэш-таблице на основе его хэш-кода.
3. SearchByKey(K key): ищет элемент с заданным ключом key в хэш-таблице.
4. SearchItemByKey(K key): возвращает элемент с заданным ключом key из хэш-таблицы.
5. Add(K key, T value): добавляет элемент с ключом key и значением value в хэш-таблицу.
6. DeleteByKey(K key): удаляет элемент с заданным ключом key из хэш-таблицы.
7. Resize(int newsize): изменяет размер хэш-таблицы.
8. ToArray(): возвращает массив элементов хэш-таблицы.
9. Print(): выводит на экран все элементы хэш-таблицы.

Класс HashTableArray<K, T> реализует хэш-таблицу на основе массива. Он использует массив Item<K, T>[] list для хранения элементов хэш-таблицы и методы HashCode(K key) и GetIndex(K key) для вычисления индекса элемента в массиве. Метод SearchByKey(K key) ищет элемент в массиве с помощью цикла, перебирая элементы от индекса, вычисленного с помощью GetIndex(K key), до конца массива, а затем от начала массива до индекса. Метод Add(K key, T value) добавляет новый элемент в массив в первый свободный слот. Если массив заполнен, то вызывается метод Resize(int newsize), который увеличивает размер массива и копирует в него все элементы из старого массива.

Класс HashTableList<K, T> реализует хэш-таблицу на основе списка. Он использует массив списков List<Item<K, T>>[] lists для хранения элементов хэш-таблицы и методы HashCode(K key) и GetIndex(K key) для вычисления индекса списка, в котором хранится элемент. Метод Add(K key, T value) добавляет новый элемент в список с помощью цикла foreach, перебирая элементы списка и проверяя, есть ли уже элемент с заданным ключом. Если такой элемент найден, его значение заменяется на новое значение. Если элемент не найден, он добавляется в список. Методы Remove(K key) и Get(K key) удаляют и возвращают элемент с заданным ключом соответственно. Они также используют методы HashCode(K key) и GetIndex(K key) для поиска правильного индекса списка и цикл foreach для перебора элементов в списке. Методы Count и Clear возвращают количество элементов в хэш-таблице и очищают хэш-таблицу соответственно.

Хэш-таблица на основе списка обычно используется, когда неизвестно, сколько элементов будет храниться в хэш-таблице, или когда количество элементов может изменяться со временем. В отличие от хэш-таблицы на основе массива, список может динамически изменяться, что позволяет ему расти и уменьшаться по мере необходимости.

Однако хэш-таблицы на основе списка могут быть медленнее, чем хэш-таблицы на основе массива, поскольку доступ к элементам в списке происходит последовательно. Кроме того, они могут занимать больше места в памяти, так как каждый элемент списка должен содержать дополнительную информацию о связи с другими элементами.

# **Граф**

Граф - это абстрактная структура данных, используемая для моделирования отношений между объектами. Граф состоит из вершин (или узлов) и ребер (или связей), которые соединяют эти вершины.

Идея графа заключается в представлении объектов (вершин) и связей между ними (ребер) в виде абстрактной структуры. Вершины графа могут представлять различные сущности, такие как люди, места, веб-страницы, компьютерные узлы и т. д. Ребра графа представляют отношения или связи между этими сущностями.

Графы являются мощным инструментом для моделирования и анализа сложных систем, таких как социальные сети, транспортные сети, графики зависимостей программ, графики сетей и многое другое. Они позволяют наглядно представить и анализировать взаимосвязи и взаимодействия между объектами.

Графы могут быть ориентированными или неориентированными. В ориентированном графе ребра имеют направление, показывающее однонаправленные связи между вершинами. В неориентированном графе ребра не имеют направления и представляют двусторонние связи между вершинами.

Графы могут быть представлены и храниться в различных формах, таких как матрицы смежности, списки смежности или смежные структуры данных. Каждая форма имеет свои преимущества и недостатки в зависимости от задачи и требований к эффективности операций.

Класс Vertex представляет вершину графа, а класс Edge - ребро. Класс Graph содержит списки всех вершин и ребер, а также методы для добавления ребер и обхода графа в ширину и глубину.

В классе Vertex определены следующие поля и методы:

* private static int IDV - статическое поле, содержащее количество созданных вершин
* private int ID - идентификатор текущей вершины
* public string label - метка (имя) вершины
* private List<Edge> edges - список ребер, связанных с вершиной
* public double sumdistance - сумма расстояний
* public COLORS\_VERTEX color - цвет вершины (используется при обходе графа)
* public Vertex prevvertex - ссылка на предшественника (используется при обходе графа)
* public bool visited - флаг, указывающий, посещалась ли вершина при обходе графа
* public Vertex(string label) - конструктор, инициализирующий поля label, edges, sumdistance, color и prevvertex
* public int GetID() - метод, возвращающий идентификатор вершины
* public List<Edge> GetEdges() - метод, возвращающий список ребер, связанных с вершиной
* public override string ToString() - метод, возвращающий строковое представление вершины (метку и идентификатор)
* public void ViewEdges() - метод, выводящий на консоль список ребер, связанных с вершиной
* public bool AddEdge(Edge edge) - метод, добавляющий ребро к списку ребер, связанных с вершиной

В классе Edge определены следующие поля и методы:

* public Vertex BeginPoint - начальная вершина ребра
* public Vertex EndPoint - конечная вершина ребра
* public double distance - длина ребра
* public Edge(Vertex begin, Vertex end, double d) - конструктор, инициализирующий поля BeginPoint, EndPoint и distance
* public override string ToString() - метод, возвращающий строковое представление ребра

public bool AddEdge(Vertex v1, Vertex v2, double d)

Граф представлен списками вершин и ребер, которые могут быть добавлены в граф с помощью метода AddEdge. Метод принимает две вершины и расстояние между ними в качестве аргументов и добавляет новое ребро в список ребер, а также обновляет список ребер для каждой из вершин.

public int BFS(Vertex s)

Метод BFS (Breadth First Search) - поиск в ширину. Этот метод ищет кратчайший путь от вершины s до всех других вершин графа. Он использует очередь для хранения вершин, которые еще не были посещены. На каждом шаге из очереди извлекается вершина и для каждого ее непосещенного соседа добавляется расстояние до него и переключается его цвет на серый. После того, как все соседи вершины были обработаны, она переключается в черный цвет.

public int DFS(Vertex s)

Метод DFS (Depth First Search) - поиск в глубину. Этот метод ищет путь от вершины s до всех других вершин графа. Он использует стек для хранения вершин, которые еще не были посещены. На каждом шаге из стека извлекается вершина и для каждого ее непосещенного соседа переключается его цвет на серый и добавляется в стек. После того, как все соседи вершины были обработаны, она переключается в черный цвет.

public List<Vertex> Get\_Path(Vertex s, Vertex v)

Метод Get\_Path - возвращает список вершин, которые образуют кратчайший путь между вершинами s и v. Этот метод используется после выполнения BFS для поиска пути между двумя вершинами. Он начинает с вершины v и идет назад по цепочке предшествующих вершин, пока не достигнет вершины s.

public int CountConnectedComponents()

Метод CountConnectedComponents - определяет количество связных компонентов в графе. Для этого метод запускает BFS из каждой непосещенной вершины графа и увеличивает счетчик каждый раз, когда запускается новый BFS. Количество связных компонентов в графе равно количеству запусков BFS.

public void ViewGraph()

public void ViewAllEdges()

public void ViewAllVertexes()

Класс также содержит несколько методов для просмотра содержимого графа, таких как ViewAllVertexes, ViewAllEdges и ViewGraph, которые позволяют просмотреть все вершины, все ребра и список ребер для каждой вершины, соответственно.

# **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. Ахо, Хопркрофт, Уллман - Структуры.данных.и.алгоритмы – 2003
  5. "Алгоритмы. Построение и анализ" Дж. Кормена, Ч. Лейзерсон, Р. Ривест, К. Штайн.