*  Алгоритм -описание последовательности действий, строгое выполнение которых приводит к решению поставленной задачи за конечное число шагов.
*  Дискретность. Под дискретностью понимается описание последовательности шагов обработки, организованной таким образом, что в начальный момент задаётся исходная ситуация, а после каждого следующего шага ситуация преобразуется на основе данных, полученных в предшествующих шагах. Дискретность алгоритма означает, что он исполняется по шагам: каждое действие, предусмотренное алгоритмом, исполняется только после того, как закончилось исполнение предыдущего.
*  Детерминированность (определенность)— каждое правило алгоритма должно быть четким, однозначным. Благодаря этому свойству выполнение алгоритма носит механический характер и не требует никаких дополнительных указаний или сведений о решаемой задаче. К примеру, в кулинарном рецепте сказано: слегка потрясите, чтобы смесь стала немного комковатой. Подогрейте молоко в маленькой кастрюльке и влейте её в смесь. Формальному исполнителю здесь неясно, требуется ли трясти смесь, пока она вся не станет комом, и какой всё-таки величины кастрюля. Большая или маленькая? И до какой температуры надо греть молоко. Так что такой алгоритм любому исполнителю выполнить довольно трудно, практически невозможно. Можно сказать, что в алгоритме не должны присутствовать не определённые слова: немного, чуть-чуть, слегка и т. д.
* Конечность. Алгоритм должен заканчиваться после конечного числа шагов, причем число шагов является критическим параметром, определяющим эффективность и сложность алгоритма.
*  Массовость. Алгоритм решения задачи всегда разрабатывается в общем виде, поэтому он должен быть применим для некоторого класса задач, различающихся только исходными данными. При этом исходные данные могут выбираться из некоторой области, которая называется областью применимости алгоритма.
*  Результативность. Это свойство подразумевает, что каждый шаг и алгоритм в целом после своего порождает результат, в котором все имеющиеся объекты однозначно определены. Если это по каким - либо причинам невозможно, то алгоритм должен сообщать, что решение задачи не существует.
*  Линейный алгоритм – тип алгоритма, в котором последовательность действий не меняется в процессе его выполнения.
*  Разветвляющий алгоритм – тип алгоритма, в котором существует хотя бы одно условие, в зависимости от которого ЭВМ осуществляет переход на одну из двух возможных команд.
*  Циклический алгоритм — тип алгоритма, предусматривающий многократное повторение одних и тех же операций над новыми исходными данными. К циклическим алгоритмам сводится большинство методов вычислений, перебора вариантов. Цикл программы — последовательность команд (серия, тело цикла), которая может выполняться многократно (для новых исходных данных) до удовлетворения некоторого условия.
*  Временная сложность алгоритма - это один из критериев оценки алгоритма. Под ней понимается зависимость итераций алгоритма от размера входных данных. Определяется функцией T(n), где n это количество входных данных.
*  Асимптотическую сложность алгоритма оценивают по порядку роста функции времени T(n). Алгоритм имеет сложность O(f(n)), если при увеличении размерности входных данных n, время выполнения алгоритма возрастает с той же скоростью, что и функция f(n). При работе с матрицей n x n во время каждой из n итераций внешнего цикла, внутренний цикл тоже выполняется n раз. Общее количество итераций внутреннего цикла равно n\*n. Это определяет сложность алгоритма как O(n 2 ). Оценивая порядок сложности алгоритма, используют только ту часть, которая возрастает быстрее всего. Предположим, что рабочий цикл описывается выражением n 3 +n. В таком случае его сложность будет равна O(n 3 ). Рассмотрение быстро растущей части функции позволяет оценить поведение алгоритма при увеличении n. Например, при n=100, разница между n 3 +n=1000100 и n 3 =1000000 равна всего лишь 100, что составляет 0,01%. При вычислении *O* можно не учитывать постоянные множители в выражениях. Алгоритм с временной сложностью 3n 3 имеет асимптотическую сложность O(n 3 ). Это делает зависимость отношения O(n) от изменения размера задачи более очевидной.
*  Временная сложность может быть константой или описываться функциями: *O*(n), *O*(log n), *O*(n log n), *O*(n 2 ), *O*(n 3 ), 2 n .
*  Структура данных (англ. *data structure*) — программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных в вычислительной технике. Для добавления, поиска, изменения и удаления данных структура данных предоставляет некоторый набор функций, составляющих её интерфейс.
* Список:
*  Массив.
*  Связанный список.
*  Ассоциативный массив.
*  Хеш-таблица.
*  Стек.
*  Очередь.
*  Граф
*  Деревья.
*  В информатике списком называют абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза. Экземпляр списка является компьютерной реализацией математического понятия конечной последовательности — кортежа. Экземпляры значений, находящихся в списке, называются элементами списка, если значение встречается несколько раз, то каждое считается отдельным элементом.
*  Односвязный список представляет собой список, каждый элемент которого содержит информацию необходимую для нахождения следующего элемента списка (узла).
*  Двухсвязные списки применяются при необходимости перемещения по списку как в прямом, так и в обратном направлении, когда заданному элементу требуется найти предыдущий и следующий за ним элемент.

Связный список

public class LinkedListNode<T>

{

// Создание нового узла со специальным значением.

public LinkedListNode(T value) // конструктор

{

Value = value;

}

// Значение узла.

public T Value // свойство

{

get;

internal set;

}

// Установка следующего значения для узла (null если последний).

public LinkedListNode<T> Next

{

get;

internal set;

}

}

public class LinkedList<T> : System.Collections.Generic.IEnumerable<T>

{

public LinkedListNode<T> \_head;

public LinkedListNode<T> \_tail;

public int Count

{

get;

private set;

}

#region Метод Add добавляет элемента в конец связаного списока

// 1) Создание нового узла

// 2) Нахожение последнего узла

// 3) Создание указателя с предыдущего узла на новый.!!!!

// Когда указатель указывает на начало списка, то при добавлении элемента приходится перебирать все его узлы, что бы достичь конечного,

// при этом степень роста алгоритма составляет О(n). Лудше когда указатель установлен на конец списка, тогда при добавлении элемента, стпень

// роста равна константе О(1).

public void Add(T value)

{

LinkedListNode<T> node = new LinkedListNode<T>(value);

if (\_head == null)

{

\_head = node;

\_tail = node;

}

else

{

\_tail.Next = node;

\_tail = node;

}

Count++;

}

#endregion

#region Метод Remove удаляет из списка указанный элемент.

//Метод возвращает true, если элемент найден и удален.

public bool Remove(T item)

{

LinkedListNode<T> previous = null;

LinkedListNode<T> current = \_head;

while (current != null)

{

if (current.Value.Equals(item)) // Определяет, равен ли заданный объект текущему объекту.

{

if (previous != null)

{

// Смена указателя Next. из предыдущего на следующий

// ДО:

// head

// +-----+------+ +-----+------+ +-----+-----+ +-----+-----+

// | 12 | \*---+--->| 15 | \*--+--->| 20 | \*--+--->| 25 | \*--+--->

// +-----+------+ +-----+------+ +-----+-----+ +-----+-----+

previous.Next = current.Next;

// ПОСЛЕ:

// head

// +-----+------+ +-----+------+ +-----+-----+

// | 12 | \*---+--->| 15 | \*--+-------------------->| 25 | \*--+--->

// +-----+------+ +-----+------+ +-----+-----+

// Определение конца списка

if (current.Next == null)

{

\_tail = previous;

}

}

else

{

// Удаление первого элемента списка

\_head = \_head.Next;

// Если список пуст и был удален единственный элемент

if (\_head == null)

{

\_tail = null;

}

}

Count--;

return true;

}

previous = current;

current = current.Next;

}

return false;

}

#endregion

#region Метод Contains возвращает значение true, если элемент пренадлежит списку

public bool Contains(T item)

{

LinkedListNode<T> current = \_head;

while (current != null)

{

if (current.Value.Equals(item))

{

return true;

}

current = current.Next;

}

return false;

}

#endregion

#region Метод получает нумератор для колекции

// Возвращает экземпляр интерфейса IEnumerator <T>, который позволяет пронумеровать элементы связанного списка,

// от первого до последнего.

public System.Collections.Generic.IEnumerator<T> GetEnumerator()

{

LinkedListNode<T> current = \_head;

while (current != null)

{

yield return current.Value;

current = current.Next;

}

}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()

{

return ((IEnumerable<T>)this).GetEnumerator();

}

#endregion

#region Метод Clear очищает список

public void Clear()

{

// Метод Clear очищет список, устанавливая \_head и \_tail в null. Достаточно удалить ссылки на элементы и сборщик мусора,

// самостоятельно сотрет список.

\_head = null;

\_tail = null;

Count = 0;

}

#endregion

#region Метод CopyTo копирует содержимое списка в массив

public void CopyTo(T[] array, int arrayIndex)

{

LinkedListNode<T> current = \_head;

while (current != null)

{

array[arrayIndex++] = current.Value;

current = current.Next;

}

}

#endregion

}

class Program

{

public static void Display(LinkedList<int> words, string test)

{

Console.WriteLine(test);

foreach (int word in words)

{

Console.Write(word + " ");

}

Console.WriteLine();

Console.WriteLine();

}

static void Main(string[] args)

{

LinkedList<int> instance = new LinkedList<int> { };

#region Добавление элементов в список

instance.Add(12);

instance.Add(15);

instance.Add(20);

instance.Add(25);

Display(instance, "List");

#endregion

#region Удаление элемента списка

instance.Remove(20);

Display(instance, "20 was removed");

#endregion

#region Копирование списка в массив

Console.WriteLine("Копирование списка в массив");

int [] arr = new int [5];

instance.CopyTo(arr, 2);

for (int i = 0; i < arr.Length; i++)

{

Console.WriteLine(arr[i]);

}

#endregion

#region Удаление списка

instance.Clear();

Display(instance, "List is cleared");

#endregion

}

}

Реализация ArrayList

public class ArrayList<T> : IEnumerable

{

T[] \_items;

// Конструктор по умолчанию

public ArrayList(): this(0)

{

}

// Конструктор с параметром

public ArrayList(int length)

{

if (length < 0)

{

throw new ArgumentException("Error length");

}

\_items = new T[length];

}

// Конструктор с параметром

public ArrayList(ICollection<T> list)

{

int index = 0;

\_items = new T[list.Count];

foreach (var element in list)

{

Count++;

\_items[index++] = element;

}

}

// Подсчет элементов в массиве

public int Count

{

get;

internal set;

}

// Метод возврящает перечислитель, осуществляет перебор коллекции.

public System.Collections.Generic.IEnumerator<T> GetEnumerator()

{

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

{

yield return \_items[i];

}

}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

}

public static void PrintValues(IEnumerable obj)

{

Console.WriteLine("Dinemic array");

foreach (var item in obj)

Console.Write(" {0}", item);

Console.WriteLine();

}

\*\*\*

Метод Add

// Метод добавляет новый элемент в конец массива

public void Add(T item)

{

{

if (\_items.Length == Count)

{

GrowArray();

}

\_items[Count++] = item;

}

}

// Расширение массива

private void GrowArray()

{

int newLength = \_items.Length == 0 ? 4 : \_items.Length << 1; // 0000 0100 => 0000 1000

T[] newArray = new T[newLength];

\_items.CopyTo(newArray, 0);

\_items = newArray;

}

\*\*\*

Вставка по индексу Insert

public void Insert(int index, T item)

{

if (index > Count)

{

throw new IndexOutOfRangeException();

}

//Расширение массива.

if (\_items.Length == Count)

{

GrowArray();

}

// Сдвиг всех элементов, после вставки элемента, в право.

Array.Copy(\_items, index, \_items, index + 1, Count - index);

\_items[index] = item;

Count++;

}

\*\*\*

Удаление Remove

// Удаление элемента по индексу

public void RemoveAt(int index)

{

if (index >= Count)

{

throw new IndexOutOfRangeException();

}

int shiftStart = index + 1;

if (shiftStart < Count) // если удаляется не последний элемент массива

{

// Сдвиг массива на один элемент влево

Array.Copy(\_items, shiftStart, \_items, index, Count - shiftStart);

}

Count--;

}

// Удаление указаного элемента

public bool Remove(T item)

{

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

{

if (\_items[i].Equals(item))

{

RemoveAt(i);

return true;

}

}

return false;

}

\*\*\*

Множество

public class Set<T> : IEnumerable<T> where T : IComparable<T>

{

// Создание экземпляра класса List<T>

private readonly List<T> \_items = new List<T>();

#region Добавление элементов в множество

public void Add(T item)

{

// Если такой элемент уже существует в множестве, выдать исключение

if (Contains(item))

{

throw new InvalidOperationException("Такое значение уже содержится в множестве");

}

\_items.Add(item);

}

#endregion

#region Проверка наличия элемента в множестве

public bool Contains(T item)

{

return \_items.Contains(item);

}

#endregion

#region Количество элементов в множестве

public int Count

{

get

{

return \_items.Count;

}

}

#endregion

#region Нумератор

public IEnumerator<T> GetEnumerator()

{

return \_items.GetEnumerator();

}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()

{

return \_items.GetEnumerator();

}

#endregion

}

Bubble Sort

// Метод реализует сортировку пузырьком

public static void BubbleSort( ref int [] items)

{

bool swapped;

do

{

swapped = false;

for (int i = 1; i < items.Length; i++)

{

if (items[i - 1].CompareTo(items[i]) > 0)

{

Swap(items, i - 1, i);

swapped = true;

}

}

}

while (swapped != false);

}