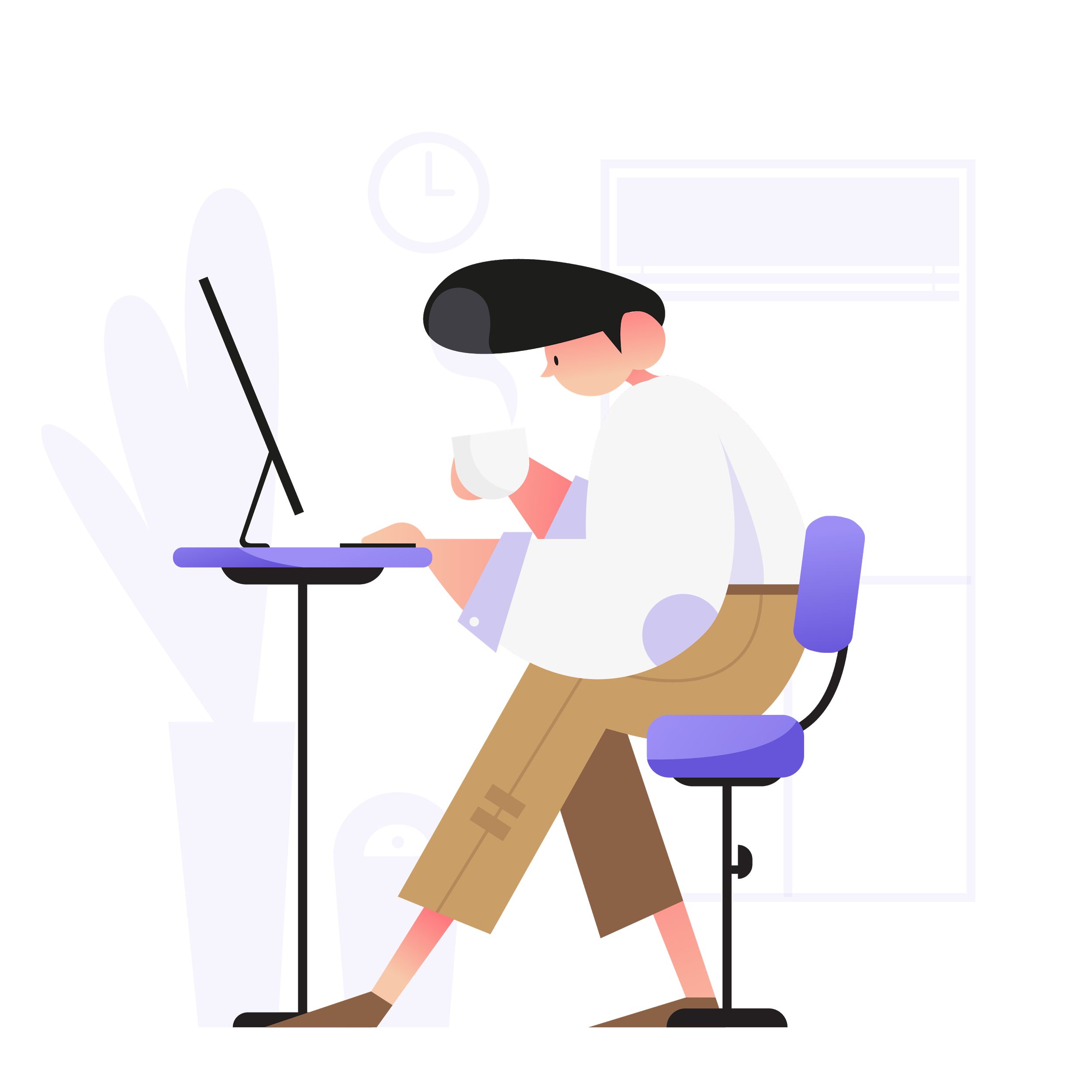
Алгоритмы и структуры данных на С#

Массив, список, поиск

Netcore 3.1



# 

**На этом уроке**

1. Узнаем, что такое массив и связный список.
2. Рассмотрим применимость связного списка и массива.
3. Выясним, что такое алгоритм сортировки, линейный поиск, бинарный поиск, интерполяционный поиск.

**Оглавление**

[Введение](#_vgv4rv859aam)

[Массив](#_mp5jhwgivq0l)

[Что такое массив](#_wz371bvslc14)

[Массив в памяти](#_vo5l6rrtqlsd)

[Связный список](#_8p26d8v13u6b)

[Что такое связный список](#_slbti4nvr524)

[Операции со списком](#_3znysh7)

[Двусвязный список](#_x3sr3joeszyp)

[Сравнение](#_dtykjqqiy08x)

[Сортировка](#_w4w8qjm1ytak)

[Поиск](#_qpskiwr4ikr)

[Линейный поиск](#_mk8db3caa4cm)

[Двоичный (бинарный) поиск](#_wifmay98s63y)

[Интерполяционный поиск](#_gaoh2ycf6vfg)

[Заключение](#_phthbruomjj1)

[Практическое задание](#_ehqy05xrcasa)

[1. Двусвязный список](#_c9dtpfeb0upc)

[2. Двоичный поиск](#_rjljchac4kow)

[Дополнительные материалы](#_2jxsxqh)

[Используемые источники](#_7c9jrnh1eqkx)

# 

# Введение

Массив и связный список — одни из базовых структур данных. Их принципы часто используются в более продвинутых структурах, только в ином виде. Ссылки на дочерние элементы в деревьях — пример того, как принцип использования ссылок из связного списка при немного другой организации может дать нам уже другую структуру данных. О массивах и списках, а также поиске в них мы узнаем на этом уроке.

# 

# Массив

## Что такое массив

Массив — это последовательный набор данных одного типа. К примеру, массив элементов типа int или string. Каждый элемент массива доступен по его индексу или, проще говоря, номеру. **Нумерация в массиве начинается с нуля**.

Доступ к элементу массива или изменение существующего стоит O(1), потому что мы заранее знаем позицию элемента в памяти.

Вставка нового элемента или удаление существующего стоит O(N), потому что для расширения или уменьшения размера массива нужно создать новый массив и скопировать элементы из текущего.

В C# массив записывается через [] — например, int[]. Также в C# есть динамический массив List<T>, он скрывает за собой работу, связанную с изменением размера, вставкой и удалением элементов. О нём поговорим подробнее на следующих занятиях.

Ниже приведён пример простой работы с массивом:

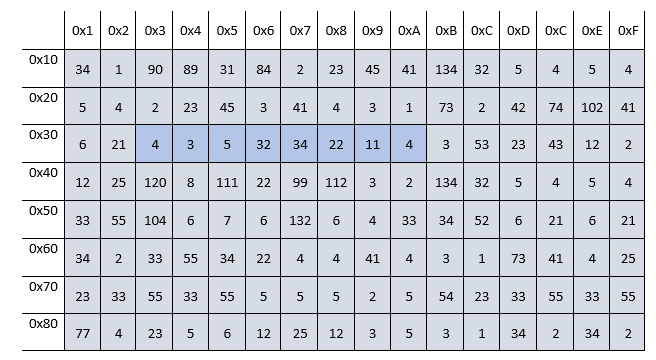
|  |
| --- |
| using System;  namespace GeekBrainsAlgos {  class Program  {  static void Main(string[] args)  {  int[] intArray = new int[] {1, 6, 77, 123, 444 , 555 ,660 }; *//Пример массива типа int*  string[] stringArray = new string[] { "Dog", "ate", "my", "homework" }; *//Пример массива типа string*   int x = intArray[0]; *// получить первый элемент массива и положить его в переменную*  Console.WriteLine(x);   PrintArray(stringArray); *//вывести массив*   stringArray[stringArray.Length - 1] = "slippers"; *// присвоить новое значение последнему элементу массива*   PrintArray(stringArray); *//вывести массив*  }   private static void PrintArray(string[] stringArray)  {  for (int i = 0; i < stringArray.Length; i++)  {  Console.Write(stringArray[i]);  Console.Write(" ");  }  Console.Write("\n\r");  }  } } |

## Массив в памяти

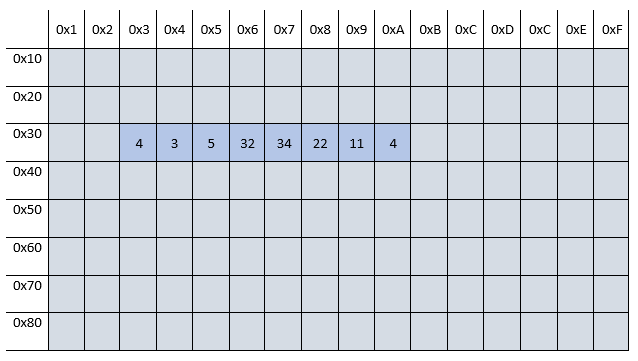
Элементы одного массива для значимых типов, таких как byte, short, int, long, float, double или struct, расположены в памяти последовательно. Эта особенность позволяет знать позицию любого элемента массива в памяти, так как в классическом понимании при работе с массивом мы всегда знаем адрес первого элемента и, соответственно, можем получить позицию другого, просто посчитав смещение относительно первого.

При последовательном проходе по элементам массива мы можем получить преимущество, так как в кеш процессора будет попадать сразу несколько нужных нам элементов массива, что уменьшит количество обращений в оперативную память. Соответственно, мы можем получить количество обращений в оперативную память меньшее, чем N, где N — количество обращений к элементам массива.

Пример расположения массива в памяти из 8 элементов со значениями {4, 3 ,5, 32, 34, 22, 11, 4}:



Если убрать для наглядности не интересующие нас ячейки памяти:

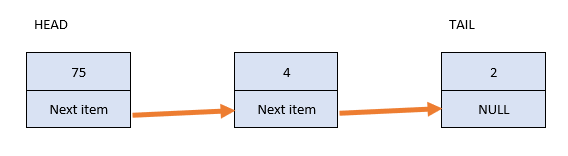


# Связный список

## Что такое связный список

В отличие от массива, связный список не хранится монолитным участком в памяти. Каждый его элемент может находиться в любом месте памяти, доступной приложению, и содержит ссылку на следующий элемент. За счёт этого при добавлении или удалении элемента не нужно его пересоздавать, достаточно поменять ссылки. Получить N-й элемент стоит O(N), потому что нельзя, как в массиве, заранее вычислить адрес N-го элемента, и нужно будет пройтись по всему списку до него.

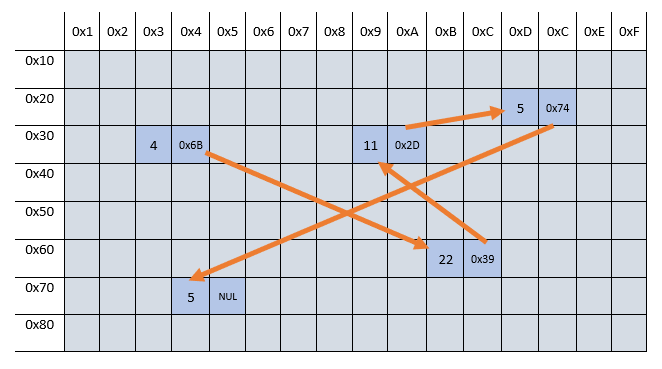
Пример списка:



Элемент списка в коде C#:

|  |
| --- |
| public sealed class Node  {  public int Value { get; set; }  public Node NextItem { get; set; }  } |

Пример: расположение ссылки в памяти. Как вы можете заметить, элементы могут располагаться в памяти непоследовательно.

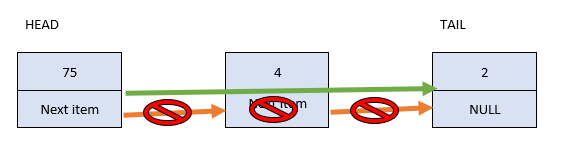


## Операции со списком

Вставка нового элемента, если известна ссылка на предыдущий элемент, будет стоить O(1), так как произойдёт только смена ссылок у элементов. Под стоимостью подразумевается асимптотическая сложность. Если ссылка неизвестна, то вставка будет стоить O(N), так как нужно будет ещё пройтись по списку и найти этот элемент. Аналогично с удалением.

Чтобы удалить элемент из примера выше, в данном случае ссылка на следующий элемент в удаляемом элементе заменяется на NULL. А ссылка на следующий элемент подставляется в предыдущий. Иными словами, в предыдущем элементе ставим ссылку на следующий после удаляемого, а в удаляемом ставим NULL.

Визуализированный пример удаления:



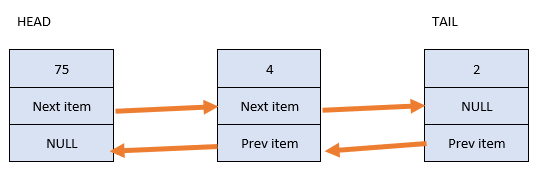
Пример кода для работы со списком:

|  |
| --- |
| public static void AddAfterItem(Node node, int value) *// добавление после любого элемента*  {  var newNode = new Node { Value = value };  var nextItem = node.NextItem;  node.NextItem = newNode;  newNode.NextItem = nextItem;  }   public static void AddItem(Node startNode, int value) *//добавление в конец, если не знаем конечный элемент*  {  var node = startNode;   while(node.NextItem != null)  {  node = node.NextItem;  }   var newNode = new Node { Value = value };  node.NextItem = newNode;  }   public static void RemoveNextItem(Node node) *//удаление следующего примера*  {  if (node.NextItem == null)  return;   var nextItem = node.NextItem.NextItem;  node.NextItem = nextItem;  }   public static Node RemoveItemByIndex(Node startNode, int itemIndex) *//удаление элемента с определённым номером; так как ссылка на начало может поменяться, метод возвращает ссылку на актуальное начало списка*   {  if(itemIndex == 0)  {  var newStartNode = startNode.NextItem;  startNode.NextItem = null;  return newStartNode;  }   int currentIndex = 0;  var currentNode = startNode;  while (currentNode != null)  {  if(currentIndex == itemIndex - 1)  {  RemoveNextItem(currentNode);  return startNode;  }   currentNode = currentNode.NextItem;  currentIndex++;  }   return startNode;  }   public static Node FindNodeByValue(Node startNode, int value) *//поиск элемента с определённым значением*  {  var currentNode = startNode;   while(currentNode != null)  {  if (currentNode.Value == value)  return currentNode;   currentNode = currentNode.NextItem;  }   return null; *// если ничего не нашли, то null*  } |

## 

## Двусвязный список

Как видно из кода выше, удаление элемента требует знания о предыдущем элементе, чтобы поменять ссылки. В двусвязном списке можно обойти эту проблему, так как каждый элемент содержит ссылку на предыдущий. Это решение увеличивает затраты памяти, так как каждый элемент теперь содержит не одну ссылку, а две, но упрощает работу со списком и даёт возможность пройти по нему в двух направлениях.



|  |
| --- |
| public sealed class NodeTwoLinked {  public int Value { get; set; }  public Node NextItem { get; set; }  public Node PrevItem { get; set; } } |

## 

## Сравнение

Список в сравнении с массивом больше подходит для случаев, когда происходит частая вставка и удаление элементов не с конца и не нужно иметь частый произвольный доступ к элементам. Массив же больше подходит для случаев, когда операций вставок и удалений очень мало и нужен частый произвольный доступ или же структура используется очень редко и её производительность не важна.

# 

# Сортировка

Сортировка даёт нам структуру, в которой элементы расставлены в порядке возрастания или убывания. Для понимания блока о поиске нужно затронуть сортировку в массиве. Пока мы рассмотрим простую сортировку пузырьком, а с остальными познакомимся на дальнейших занятиях.

Код сортировки пузырьком, вычислительная сложность O(N2):

|  |
| --- |
| static void BubbleSort(int[] array)  {  for (int i = 0; i < array.Length; i++)  {  for (int j = 0; j < array.Length - 1; j++)  {  if (array[j] > array[j + 1])  {  var temp = array[j + 1];  array[j + 1] = array[j];  array[j] = temp;  }  }  }  } |

Для простоты можете использовать встроенную функцию сортировки в C# Array.Sort():

|  |
| --- |
| static void Main(string[] args)  {  var array = new int[] { 55, 11, 77, 22 };  Array.Sort(array);  } |

# 

# Поиск

## Линейный поиск

Простой поиск, в котором мы проходим по всей структуре и сравниваем элемент с искомым. Его сложность — O(N).

Для массива:

|  |
| --- |
| public static int FindIndexByValue(int[] array, int value) *//поиск элемента с определённым значением*  {  for(int i = 0; i < array.Length; i++)  {  if (array[i] == value)  return i;  }   return -1;  } |

Для списка:

|  |
| --- |
| public static Node FindNodeByValue(Node startNode, int value) *//поиск элемента с определённым значением*  {  var currentNode = startNode;   while(currentNode != null)  {  if (currentNode.Value == value)  return currentNode;   currentNode = currentNode.NextItem;  }   return null; *// если ничего не нашли, то null*  } |

## 

## Двоичный (бинарный) поиск

Ускорить поиск невозможно, если нет дополнительных данных. Хорошо известно, что поиск может быть гораздо более эффективным, если данные упорядочены. Достаточно представить себе телефонный справочник.

Выбираем центральный элемент массива с индексом mid, сравниваем его с искомым значением:

* если он равен искомому значению, то поиск прекращается;
* если он меньше искомого значения, то заключаем, что все элементы с индексами, равными или меньшими mid, можно игнорировать в дальнейшем поиске;
* если он больше искомого значения, то можно игнорировать все значения индекса, большие или равные mid.

Такой поиск называется двоичным (binary search). В нём используются две индексные переменные min и max, отмечающие в массиве левый и правый концы отрезка, в котором искомое значение всё ещё может быть найдено. Главное предусловие для данного алгоритма — массив должен быть отсортирован. Если говорить проще, то делим массив пополам, смотрим, в какой половине может находиться искомый элемент, и повторяем это уже в искомой половине.

|  |
| --- |
| public static int BinarySearch(int[] inputArray, int searchValue) {  int min = 0;  int max = inputArray.Length - 1;  while (min <= max)  {  int mid = (min + max) / 2;  if (searchValue == inputArray[mid])  {  return mid;  }  else if (searchValue < inputArray[mid])  {  max = mid - 1;  }  else  {  min = mid + 1;  }  }  return -1; } |

## Интерполяционный поиск

Если при бинарном поиске целевой элемент массива каждый раз ищется в середине раздела, то интерполяционный поиск призван ускорить процесс: он старается угадать расположение целевого элемента в массиве по его значению. Предположим, в массиве содержится 1000 элементов со значениями от 1 до 100. Если наша цель — найти число 30, то его нужно искать в районе первой трети массива, где-то рядом с индексом 300. Общее распределение чисел не всегда позволяет получить результат с прицельной точностью, но он может оказаться довольно близким.

|  |
| --- |
| public static int InterpolationSearch(int[] a, int length, int value) {  int min = 0;  int max = length - 1;  while (min <= max)  {  *// Находим разделяющий элемент*  int mid = min + (max - min) \* (value - a[min]) / (a[max] - a[min]);  if (a[mid] == value)  return mid;  else if (a[mid] < value)  min = mid + 1;  else if (a[mid] > value)  max = mid - 1;  }  return -1; *// Элемент не найден* } |

В этом алгоритме некоторые проблемы остаются нерешёнными. Например, расчёт mid может привести к выходу за пределы массива или к тому, что его значение не будет находиться между min и max.

Самая сложная часть приведённого кода — оператор для расчёта mid. Чтобы найти его значение, к текущему значению min добавляется расстояние от min до max, масштабируемое по ожидаемой доле расстояния от a[min] до a[max], где должно находиться искомое значение values.

Например, если значение a[min] равно 100, a[max] — 200, а values — 125, определить место для поиска целевого элемента можно путём таких расчётов:

(values – a[min]) / (a[max] – a[min]) = (125 – 100) / (200 – 100) = 25 / 100 = 0,25.

Это значит, что новое значение mid должно располагаться на четверти пути от min до max. Если данные распределены очень неравномерно и вы ищете наихудшее целевое значение, этот алгоритм обладает производительностью O(N). Если распределение относительно равномерное, ожидаемая производительность составит O(log (log N)).

# 

# Заключение

Мы рассмотрели массив и связный список, выяснили, в какие моменты стоит использовать список, а в какие — массив. Узнали, что массив хранится в памяти монолитом, а элементы связного списка разбросаны по памяти, но добавить элементы в связный список проще, чем в массив. Также узнали о методах поиска: можно искать не только линейно в массиве, но и, если массив отсортирован, с некоторым шагом, пропуская не интересующие элементы.

# 

# Практическое задание

## 1. Двусвязный список

Требуется реализовать класс двусвязного списка и операции вставки, удаления и поиска элемента в нём в соответствии с интерфейсом.

|  |
| --- |
| namespace GeekBrainsTests {  public class Node  {  public int Value { get; set; }  public Node NextNode { get; set; }  public Node PrevNode { get; set; }  }   *//Начальную и конечную ноду нужно хранить в самой реализации интерфейса*  public interface ILinkedList  {  int GetCount(); *// возвращает количество элементов в списке*  void AddNode(int value); *// добавляет новый элемент списка*  void AddNodeAfter(Node node, int value); *// добавляет новый элемент списка после определённого элемента*  void RemoveNode(int index); *// удаляет элемент по порядковому номеру*  void RemoveNode(Node node); *// удаляет указанный элемент*  Node FindNode(int searchValue); *// ищет элемент по его значению*  }  } |

## 2. Двоичный поиск

Требуется написать функцию бинарного поиска, посчитать его асимптотическую сложность и проверить работоспособность функции.

# Дополнительные материалы

1. [Сортировка пузырьком](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BF%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%BC).
2. [Сортировка выбором](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%8B%D0%B1%D0%BE%D1%80%D0%BE%D0%BC).
3. [Сортировка перемешиванием](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D1%88%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%D0%BC).
4. [Вычислительная сложность](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C).

# Используемые источники

1. Р. Стивенс. Алгоритмы. Теория и практическое применение. М.: Издательство «Э», 2016.
2. Н. Вирт. Алгоритмы и структуры данных. Новая версия для Оберона. М.: ДМК-пресс, 2010.
3. Д. Рихтер CLR via C# М.: Питер, 2019.