**Linear search in c#**

**Linear search implementation in c#**

Linear search, also known as sequential search, is a simple algorithm used to find a specific value in an array or list. It works by iterating through each element of the array or list and comparing it to the target value. If the target value is found, the index of the element is returned. If the target value is not found, the algorithm will return -1. So let’s take a look at linear search in c#.

In C#, the linear search can be easily implemented using a for loop. Here is an example of how to implement a linear search algorithm in C#:

static int LinearSearch(int[] numbers, int target)

{

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

{

if (numbers[i] == target)

{

return i;

}

}

return -1;

}

int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int index = LinearSearch(numbers, 5);

Console.WriteLine("The number 5 is at index " + index);

This example creates an array of integers and searches for the number 5. The LinearSearch method takes two parameters, an array of integers and the target value. The method iterates through the array using a for loop, and compares each element to the target value. If the element is equal to the target value, the index of the element is returned. If the target value is not found, the method will return -1. The output of this program will be “The number 5 is at index 4”, which is the index of the number 5 in the array.

**The time complexity**

Linear search has a time complexity of O(n), which means that the time it takes to find a specific value increases linearly with the size of the array or list. This makes linear search less efficient than other algorithms, such as binary search, when searching large arrays or lists.

If you would know more about time complexity, take a look at [this](https://tutorials.eu/csharp-time-complexity/) article.

Below you can see the graph that shows how the complexity grows along with the data:

A graph of a function

Description automatically generated

**Pros and cons of linear search**

Pros:

* Simple to understand and implement: Linear search is a basic algorithm that is easy to understand and implement, making it suitable for beginners.
* No need to sort the data: Linear search does not require the data to be sorted, which makes it more versatile than other search algorithms, such as binary search.
* Can be used with any data type: Linear search can be used with any data type, including arrays and lists of integers, strings, and custom objects.

Cons:

* The time complexity of O(n).
* Inefficient for large datasets: Because linear search has a time complexity of O(n), it becomes less efficient when searching large datasets. This is because it has to iterate through every item in the dataset in order to find the target value.
* Not suitable for large-scale or real-time applications: Due to its inefficiency for large datasets, linear search is not suitable for large-scale or real-time applications where performance is critical.

Binary Search using C#

Binary Search Algorithm. It is one of the Divide and conquer algorithms types, where it halves the number of elements it has to search in each step, making the average time complexity O (log n). It works on a sorted array. Given below are the steps/procedures of the Binary Search algorithm.

* Each step compares the search key with the value of the middle element of the array.
* The keys matching in step 1 means a matching element has been found, and its index (or position) is returned. Else step 3 or 4.
* If the search key is less than the middle element, then the algorithm repeats its action on the sub-array to the left of the middle element or,
* If the search key is greater than the middle element, then the algorithm repeats its action on the sub-array to the right of the middle element.
* If the search key does not match any of the subsequent left or right arrays, it means that the key is not present in the array, and a special "Nil" indication can be returned.

We will first have a look at the C# implementation using an iterative approach.

public static object BinarySearchIterative(int[] inputArray, int key)

{

int min = 0;

int max = inputArray.Length - 1;

while (min <=max)

{

int mid = (min + max) / 2;

if (key == inputArray[mid])

{

return ++mid;

}

else if (key < inputArray[mid])

{

max = mid - 1;

}

else

{

min = mid + 1;

}

}

return "Nil";

}

C#

Copy

And here is the recursive one. Please note that in recursive, you need to pass min as 0 and max as inputArray.Length - 1

public static object BinarySearchRecursive(int [] inputArray, int key, int min, int max)

{

if (min > max)

{

return "Nil";

}

else

{

int mid = (min+max)/2;

if (key == inputArray [mid])

{

return ++mid;

}

else if (key < inputArray [mid])

{

return BinarySearchRecursive(inputArray, key, min, mid - 1);

}

else

{

return BinarySearchRecursive(inputArray, key, mid + 1, max);

}

}

}

C#

Copy

The above code is a recursive implementation of the binary search algorithm. Binary search is a divide-and-conquer algorithm that can be used to search for a value in a sorted array.

The code first checks if the minimum value is greater than the maximum value. If it is, then the value is not found in the array, and the function returns "Nil". Otherwise, the code calculates the middle index of the array and compares the value to the element at the middle index.

If the value is equal to the element at the middle index, then the function returns the index of the element. Otherwise, the code recursively calls itself on the half of the array that does not contain the value.

The recursive call to the function continues until the value is found or the minimum value is greater than the maximum value.

Here is an explanation of the code step-by-step:

1. The function takes the input array, the key to search for, and the minimum and maximum values of the array as input.
2. The function checks if the minimum value is greater than the maximum value. If it is, then the value is not found in the array, and the function returns "Nil".
3. Otherwise, the function calculates the middle index of the array.
4. The function compares the value to the element at the middle index.
5. If the value is equal to the element at the middle index, then the function returns the index of the element.
6. Otherwise, the function recursively calls itself on the half of the array that does not contain the value.
7. The recursive call to the function continues until the value is found or the minimum value is greater than the maximum value.
8. //using System;
9. //using System.Collections.Generic;
10. //using System.Linq;
11. //using System.Text;
12. //using System.Threading.Tasks;
13. //namespace ConsoleApp\_Wipro2024.Week\_2.Day\_4
14. //{
15. // class linear\_binary\_seaches
16. // {
17. // static void Main(string[] args)
18. // {
19. // int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
20. // int index = LinearSearch(numbers, 5);
21. // Console.WriteLine("The number 5 is at index " + index);
22. // Console.ReadLine();
23. // }
24. // static int LinearSearch(int[] numbers, int target)
25. // {
26. // for (int i = 0; i < numbers.Length; i++)
27. // {
28. // if (numbers[i] == target)
29. // {
30. // return i;
31. // }
32. // }
33. // return -1;
34. // }
35. // }
36. //}
37. // binary search demo
38. using System;
39. class MyClass
40. {
41. static void Main(string[] args)
42. {
43. int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
44. BinarySearchDisplay(arr, 9);
45. Console.ReadLine();
46. }
47. public static object BinarySearchDisplay(int[] arr, int key)
48. {
49. int minNum = 0; //0
50. int maxNum = arr.Length - 1; //10-1 =9
51. while (minNum <= maxNum) // 0<9
52. {
53. int mid = (minNum + maxNum) / 2; // mid= 0+9/2= 4
54. if (key == arr[mid]) // 9 == arr[4]=5
55. {
56. return ++mid;
57. }
58. else if (key < arr[mid])
59. {
60. max = mid - 1;
61. }
62. else
63. {
64. min = mid + 1;
65. }
66. }
67. return "None";
68. }