

Draft Version

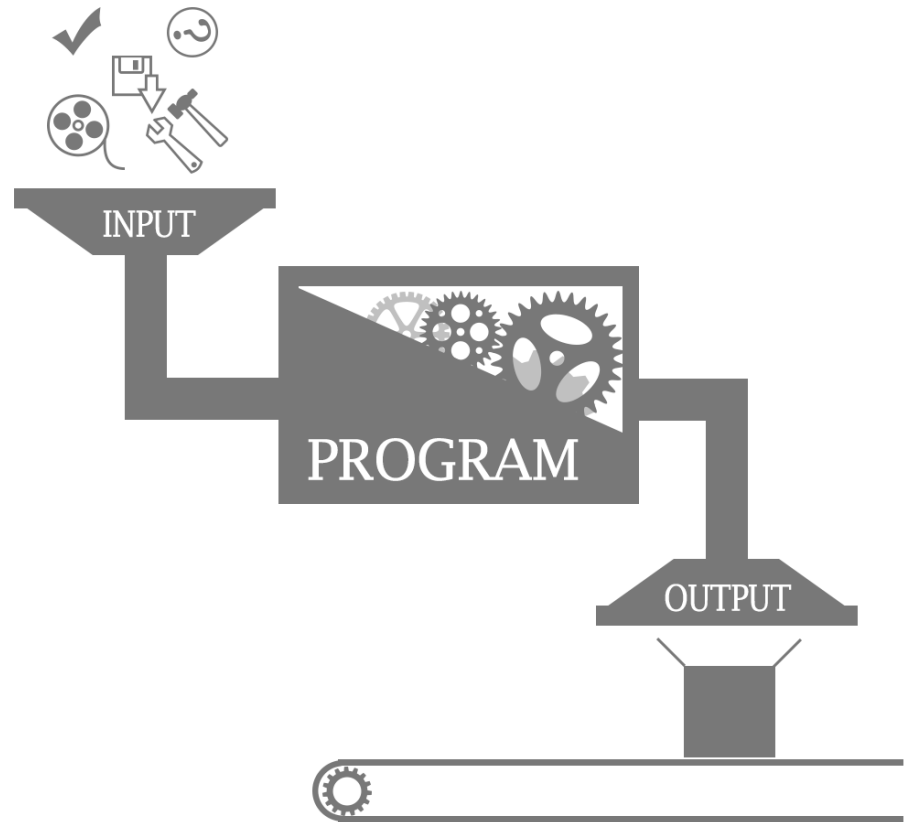
DSA\300 – Data Structures and
Algorithms with C#

Data Structures & Algorithms with C# – Lecture #4

Ahmed Mohyeldin



Please consider the environment before printing this material.



Lecture #4

SEARCHING ALGORITHMS

Searching a List of Data Elements for a Particular Value 😊

Search Algorithms – The *Definition* of Searching?

- *Searching* is the process of locating a given item in a data collection.
- *Searching Methods* can be divided into two categories:

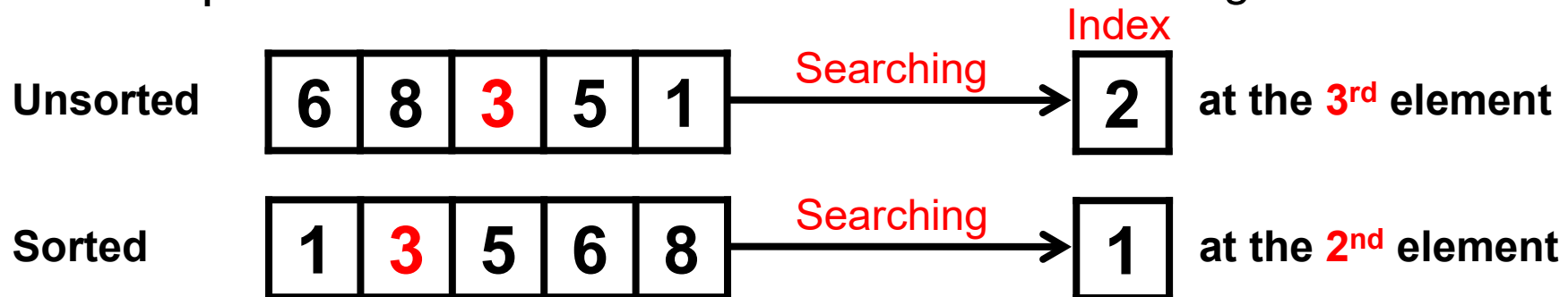
- ☐ Searching methods for unsorted as well as sorted lists.

- e.g., *Sequential Search*.

- ☐ Searching methods for sorted lists.

- e.g., *Binary Search*.

- Example: Search for element value = 3 in the following lists?



Search Algorithms – The *Benefits* of Searching?

- Searching algorithms has many practical applications in computing and science:
 - Finding a specific piece of information in digital data storage is all about searching!
 - Large a mount of Internet time is spent in searching large variety of databases.
 - RDBMS queries.
 - File systems search.
 - *...etc.*

Search Algorithms –The *Methods* of Searching?

- The searching techniques differ with data structures and data organization.

1 *Array* Search Techniques

- Two basic search methods are used to search arrays:
 - *Linear* Search.
 - *Binary* Search.
 - *Also*, variants of binary search method exist, e.g.,
 - *Interpolation* Search, *Exponential* Search.

2 *Tree* Search Techniques

- The *binary search tree* and *B-tree data structures* are based on binary search.
- Two advanced search methods are used to search trees and graphs:
 - *Breadth First* Search (BFS) and *Depth First* Search (DFS)

Searching Algorithms – *Performance Analysis*?

- **Judging the Performance** of Searching Algorithm
 - Many different algorithms exist for carrying out searching either sorted and/or unsorted lists.
 - Each algorithm has its merits, but the general criteria for judging a searching algorithm are based on the following questions:

- 1 **How fast**, can it find a match in an average case?
- 2 **How fast** is its best and worst case of finding match?
- 3 **Does it** support searching only sorted lists.
- 4 **Does it** support searching both sorted and unsorted lists?

Searching Algorithms – The *Linear* Search

Linear Search

Linear Search – The *What?*

- *Linear Search* is also known as *Sequential Search*.
- *Linear Search* can be used to search both *sorted* and *unordered* lists.
- *Linear Search* operates by checking every element of a list one at a time in sequence until a match is found.

Linear Search – The *How*?

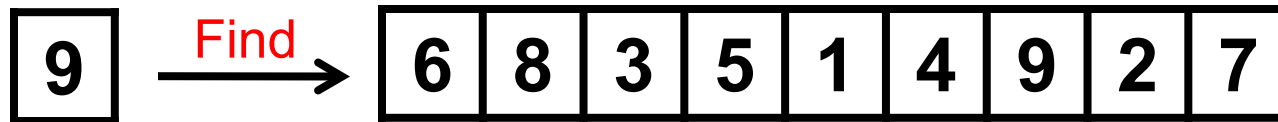
■ Algorithm Procedure:

- 1 For each item in the list, check if the item we are looking for matches the item in the list as follows:
 - i If it matches, return the location where the item is found(*i.e.*, the item index) and end the search.
 - ii Otherwise, continue searching until a match is found or the end of the list is reached.
- 2 If the end of the list is reached without finding a match, then the item does not exist in the list.

Linear Search – *Step-by-Step* Example?

■ *Step-by-Step* Example:

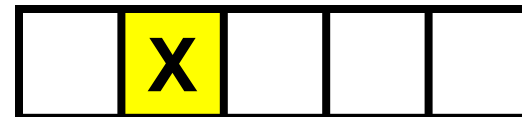
- Search the following list for the value of 9 using the Linear (*i.e.*, Sequential) Search algorithm:



□ *Notes:*

- Symbols used in the solution steps:

Current element to be compared with the key.



Match is found.



Linear Search – *Step-by-Step* Example?

Initial State:

9



6

8

3

5

1

4

9

2

7

Is 1st element = 9? → False

6

8

3

5

1

4

9

2

7

Is 2nd element = 9? → False

6

8

3

5

1

4

9

2

7

Is 3rd element = 9? → False

6

8

3

5

1

4

9

2

7

Is 4th element = 9? → False

6

8

3

5

1

4

9

2

7

Is 5th element = 9? → False

6

8

3

5

1

4

9

2

7

Is 6th element = 9? → False

6

8

3

5

1

4

9

2

7

Is 7th element = 9? → True

6

8

3

5

1

4

9

2

7

Match is found at 7th element

6

8

3

5

1

4

9

2

7

Finally, match is found at the 7th element, and the algorithm can terminate.

Linear Search – *Pseudocode*(Basic Algorithm)!

```
1 // Linear Search – Pseudocode #1: Basic Algorithm
2
3 procedure LinearSearch( A  : list of n items,
4                        key: value to be searched for)
5 defined as:
6     n = length( A )
7
8     for i = 0 to n-1 do // Comparison i<n is required.
9         if key == A[i] then
10             return i and stop // Match is found.
11         end if
12     end for
13
14     return -1 // Match is not found.
15 end procedure
16 /* Pseudocode #1: The basic algorithm above makes two
17 comparisons per iteration: one to check if A[i] equals
18 key, and the other to check if i still points to a
19 valid index of the list. */
```

Linear Search – *C# Implementation* #1

```
1 // LinearSearch<T>() - Searches the entire List<T> for an element
2 // (Basic Algorithm) using the default comparer & "Linear (i.e.,
3 // Sequential) Search" algorithm.
4 static int LinearSearch<T>(T[] list, T item)
5     where T : System.IComparable<T>
6 {
7     int n = list.Length;
8
9     for (int i = 0; i < n; i++)
10    {
11        if (item.CompareTo(list[i]) == 0) // (item == list[i])?
12        {
13            return i; // Match is found at index i.
14        }
15    }
16
17    return -1; // No match is found.
18 }
19
20 /* Note: Return Value = The zero-based index of item in the
21    List<T>, if item is found; otherwise, a negative number. */
22
23
24
```

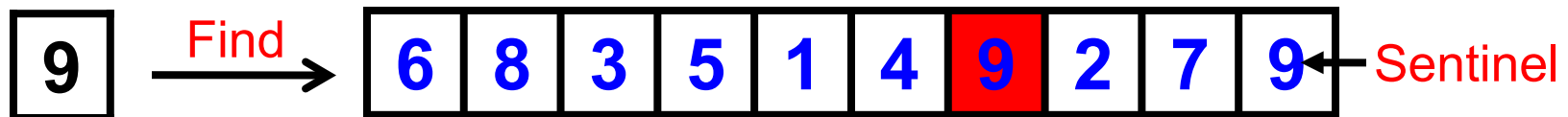
Linear Search – *C# Implementation* #1 (Alt.)

```
1 // LinearSearch<T>() - Searches the entire List<T> for an element
2 // (Basic Algorithm) using the default comparer & "Linear (i.e.,
3 // Sequential) Search" algorithm.
4 static int LinearSearch<T>(T[] list, T item)
5     where T : System.IComparable<T>
6 {
7     int n = list.Length;
8     int retValue = -1;
9     for (int i = 0; i < n; i++)
10    {
11        if (item.CompareTo(list[i]) == 0) // (item == list[i])?
12        {
13            retValue = i; // Match is found at index i.
14            break;
15        }
16    }
17
18    return retValue;
19 }
20
21 /* Note: Return Value = The zero-based index of item in the
22    List<T>, if item is found; otherwise, a negative number. */
23
24
```

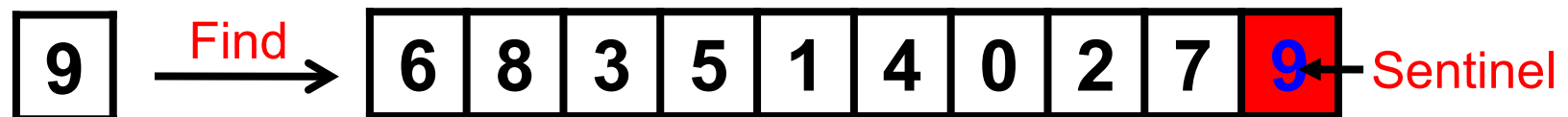
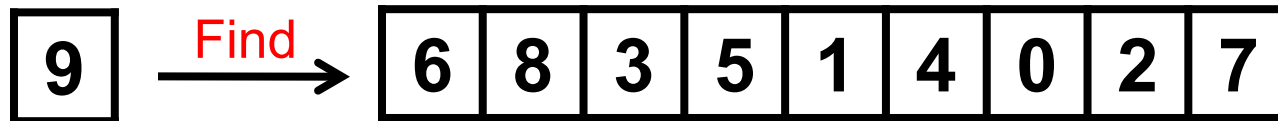
Linear Search – *Pseudocode*(With a Sentinel)!

- **Sentinel Optimization Idea:** By adding an extra record to the list (a **sentinel value**) that equals the target (**search key**), the second comparison, to check if the end of the list is reached, can be eliminated until the end of the search, making the algorithm faster.

- **Case #1:** Search **key exists** in the list



- **Case #2:** Search **key does not exist** in the list



Linear Search – *Pseudocode*(With a Sentinel)!

```
1 // Linear Search – Pseudocode#2: With a Sentinel Value
2 procedure LinearSearch( A : list of n items,
3                       key: target value)
4 defined as:
5   Append key to A    // Add a sentinel = target.
6   i = 0, n = length( A )
7   while A[i]!=key do // Comparison i<n is not required.
8     i = i + 1
9   end while
10  if i < n-1 then // Is the sentinel not reached yet?
11    return i      // Yes, match is found at i < n - 1.
12  else
13    return -1     // No, match is not found.
14 end procedure
15 /* Pseudocode #2: By adding an extra record to the list (a
16 sentinel value) that equals the target, the second
17 comparison can be eliminated until the end of the search,
18 making the algorithm faster. The search will reach the
19 sentinel if the target is not contained within the list. */
```




TECHNICAL NOTE: The Sentinel Overhead?

- **Tack Care:** Using a sentinel value in linear search reduces the number of comparisons by half. However, it places possible high overhead on the execution time performance! This depends on both the *data type* and *data structure* of the list's internal storage as follows:

Data Structure


- *Ordinary Array*: A completely new array of size = the original size + 1 has to be allocated. This is followed by a time-consuming copy operations of the entire old list elements to their new counterparts – **Considerable overhead! X ☹**
- *Linked List*: Only one new node (*i.e.*, item) has to be allocated, one copy operation to be assigned to the target value, and appended to the list – **Relatively accepted overhead. OK ☺**

Data Type

- Handling lists of objects incurs much higher copy operations overhead than lists of references only, especially with large-size objects. **Therefore, efficient list data structure implementation should only keep references to the items instead of using imbedded copies of them.**

Linear Search – *C# Implementation* #2

```
1 // LinearSearch<T>() - Searches the entire List<T> for an element
2 // (With a Sentinel) using the default comparer & "Linear (i.e.,
3 // Sequential) Search" algorithm.
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20 // TODO: Write a C# modified version of the LinearSearch<T>( ),
21 // which uses the "Basic Linear Search" method to use the
22 // "Sentinel Value" approach instead. Then, compare the run-
23 // time performance of both versions using a large data set.
24
```



Linear Search – *C# Test Driver* (Code)

```
1 // Linear Search Algorithm - LinearSearch<T>( ) Test Driver.
2 using System;
3
4 namespace DSA
5 {
6     class Program
7     {
8         // main() - Program entry point.
9         static void Main(string[] args)
10        {
11            Console.WriteLine("-----");
12            Console.WriteLine("  Testing DSA.LinearSearch<T>( ) ");
13            Console.WriteLine("-----");
14
15            int n, match;
16            double [] list;
17            double key;
18            string input;
19            bool check;
20            Console.Write("Enter the number of list elements: ");
21            input = Console.ReadLine();
22            check = int.TryParse(input, out n);
23            if (!check) return;                                // continued...
24
```

Linear Search – *C# Test Driver* (Code)

(...cont'd)

```
25         list = new double[n]; // Create a list[n].
26
27         for (int i = 0; i < n; i++) // Enter the list elements:
28         {
29             do
30             {
31                 Console.Write("Enter element no.{0}: ", i + 1);
32                 input = Console.ReadLine();
33                 check = double.TryParse(input, out list[i]);
34             } while (!check);
35         }
36
37         Console.Write("Enter the search key: ");
38         input = Console.ReadLine();
39         check = double.TryParse(input, out key);
40         if (!check) return;
41
42         match = LinearSearch(list, key); // Perform the search.
43
44         if (match < 0)
45             Console.WriteLine("No match is found.");
46
47                                     // continued...
48
```

Linear Search – *C# Test Driver* (Code)

(...cont'd)

```
49
50         else
51             Console.WriteLine("Match is found at element no.{0}",
52                               match + 1);
53
54             Console.WriteLine("Press any key to continue...");
55             Console.ReadKey(true);
56     }
57
58     /* Note: Return Value = The zero-based index of item in the
59        List<T>, if item is found; otherwise, a negative number. */
60
61                                     // continued...
62
63
64
65
66
67
68
69
70
71
72
```

Linear Search – *C# Test Driver* (Code)

(...cont'd)

```
73 // LinearSearch<T>() - Searches the entire List<T> for an element
74 // (Basic Algorithm) using the default comparer & "Linear (i.e.,
75 // Sequential) Search" algorithm.
76 static int LinearSearch<T>(T[] list, T item)
77     where T : System.IComparable<T>
78 {
79     int n = list.Length;
80
81     for (int i = 0; i < n; i++)
82     {
83         // (item == list[i])?
84         if (item.CompareTo(list[i]) == 0)
85         {
86             return i; // Match is found at index i.
87         }
88     }
89
90     return -1; // No match is found.
91 }
92
93 /* Note: Return Value = The zero-based index of item in the
94 List<T>, if item is found; otherwise, a negative number. */
95
96
```

Linear Search – *C# Test Driver* (Output)

(...cont'd)

- **Output:**

Testing DSA.LinearSearch<T>() (Generic Version)

Enter the number of list elements: 9 ←

Enter element no.1: 6 ←

Enter element no.2: 8 ←

Enter element no.3: 3 ←

Enter element no.4: 5 ←

Enter element no.5: 1 ←

Enter element no.6: 4 ←

Enter element no.7: 9 ←

Enter element no.8: 2 ←

Enter element no.9: 7 ←

Enter the search key: 9 ←

Match is found at element no.7

Press any key to continue...

- **Note:**

← Return Key required following a user input value.

Linear Search – The *Time* Analysis?

■ Linear Search Utilizes a Single Loop.

□ *Theoretically*, the comparison loop runs at most n times. In each iteration, it compares the search key with the element at index i , where $i \in [0 \text{ to } n - 1]$.

□ No. of Comparisons:

- Worst Case = n (All n elements will be tested).
- Average Case = $n/2$ (half of the elements will be tested).
- Best Case = 1 (Only, one element will be tested).
- If the information is stored on disk, the search time can be very long, but if the data is unsorted, this is the only method available.
- **Note**, Looking more precisely at the practical implementation details, another comparison is needed at each loop cycle to test if the end of list is reached. Therefore, the actual number of comparisons are: $2n$, n , 2 for worst, average and best cases, respectively.

Linear Search – The *Performance* Summary?

■ *Time Analysis:*

- Worst Case $O(n)$ ← $O(n)$ comparisons
- Average Case $O(n)$ ← $O(n)$ comparisons
- Best Case $O(1)$ ← $O(1)$ comparisons
 - Where, n is the number of items being searched.
- The *Linear Search* is the *Simplest* and *Worst* of searching routines! However, it is the only method that *fits both sorted and unsorted lists*.

■ *Space Analysis:*

- Worst Case Space = $O(1)$ iterative
 - *i.e.*, only requires a constant amount of additional memory space.
 - Where, n is the number of items being searched.

Binary Search

Binary Search – The *What?*

- *Binary Search* is fast searching algorithm, but it can be used only to search *sorted* lists.
- The *Binary Search* method uses the “*divide-and-conquer*” approach.
- The *binary search tree* and *B-tree* data structures are based on binary search.

Binary Search – The *How*?

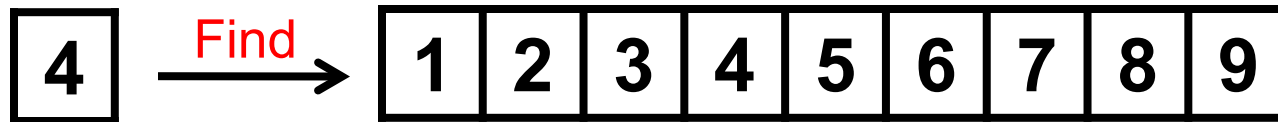
■ Algorithm Procedure:

- 1 First, test the middle element;
 - i If the element is larger than the key, then test the middle element of the first half;
 - ii Otherwise, test the middle element of the second half.
 - 2 Repeat this process until either a match is found, or there are no more elements to test.
- **Remember:** The Binary Search Algorithm can only be applied to a *sorted* list.

Binary Search – *Step-by-Step* Example?

■ *Step-by-Step* Example:

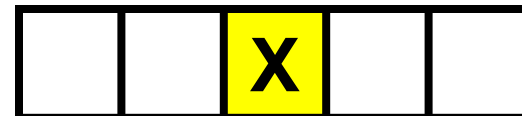
- Search the following sorted list for the value of 4 using the Binary Search algorithm:



□ *Notes:*

- Symbols used in the solution steps:

Middle value of the list.



White Cells: remaining list cells.

Black Cells: discarded cells.

Red Cell : Match is found.



Binary Search – *Step-by-Step* Example?

(...cont'd)

Initial State:

4

Find →

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Mid. Value = 5 > 4

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Consider only right half

1	2	3	4	5				
---	---	---	---	---	--	--	--	--

Mid. Value = 3 < 4

1	2	3	4	5				
---	---	---	---	---	--	--	--	--

Consider only left half

		3	4	5				
--	--	---	---	---	--	--	--	--

Mid. Value = 4 (match is found)

		3	4	5				
--	--	---	---	---	--	--	--	--

Finally, match is found at the 4th element, and the algorithm can terminate.

Binary Search – *Pseudocode*

```
1 // Binary Search – Pseudocode:
2
3 procedure BinarySearch( A : sorted list of n items,
4                       key: value to be searched for)
5 defined as:
6     n = length( A )
7
8     set low = 0, high = n-1
9     while low <= high do
10         mid = (low + high) / 2
11         if key < A[mid] then
12             high = mid-1
13         else if key > A[mid] then
14             low = mid+1
15         else
16             return mid and stop // Match is found.
17     end while
18     return -1 // Match is not found.
19 end procedure
```

Binary Search – *C# Implementation*

```
1 // BinarySearch<T>() - Searches the entire sorted List<T> for an
2 //                       element using the default comparer & "Binary
3 //                       Search" algorithm.
4 static int BinarySearch<T>(T[] list, T item)
5     where T : System.IComparable<T>
6 {
7     int low, high, mid;
8     low = 0;
9     high = list.Length - 1;
10
11     while (low <= high)
12     {
13         mid = (low + high) / 2;
14         if (item.CompareTo(list[mid]) < 0)      // (item<list[mid])?
15             high = mid - 1;
16         else if (item.CompareTo(list[mid]) > 0) // (item>list[mid])?
17             low = mid + 1;
18         else
19             return mid; // Match is found at mid.
20     }
21     return -1; // No match is found.
22 } /* Note: Return Value = The zero-based index of item in the
23    List<T>, if item is found; otherwise, a negative number. */
24
```


Binary Search – *C# Test Driver* (Code)

```
1  // Binary Search Algorithm - BinarySearch<T>() Test Driver.
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20 // TODO: Write a C# "Test Driver" for the BinarySearch<T>( ).
21 //      a simple C# console program similar to the one showed
22 //      previously for LinearSort<T>( ).
23
24                                     // continued...
```

Binary Search – *C# Test Driver* (Output)

(...cont'd)

- **Output:**

Testing DSA.BinarySearch<T>() (Generic Version)

Enter the number of list elements: 9 ←

Enter element no.1: 1 ←

Enter element no.2: 2 ←

Enter element no.3: 3 ←

Enter element no.4: 4 ←

Enter element no.5: 5 ←

Enter element no.6: 6 ←

Enter element no.7: 7 ←

Enter element no.8: 8 ←

Enter element no.9: 9 ←

Enter the search key: 4 ←

Match is found at element no.4

Press any key to continue...

- **Note:**

← Return Key required following a user input value.

Binary Search – The *Time* Analysis?

- Binary Search Utilizes a Single Loop.

- *Theoretically*, the comparison loop runs at most $\lfloor \log_2(n + 1) \rfloor$ times. In each iteration, it makes one or two comparisons, checking if the middle element is equal to the key value in each iteration.
- *What is the expected runtime in worst, average and best case scenarios?*

■ *Worst Case Analysis:*

- In binary search, the search is confined to a sub-array which is half of the size of the original array.
- At each step, search key is compared with middle element of the sub-array (either 1 or 2 comparisons are made per iteration).

□ *No. of Comparisons* = $\log_2(n + 1)$

- In general, the total number of comparisons is equal to the total number of binary partitions of the array.
- Since, the partition sizes can be 1, 2, 4, ..., if there are at most P partitions then
- $2^0 + 2^1 + 2^2 + \dots + 2^{P-1} = 2^P - 1$
- Taking logarithm: $P = \log_2(n + 1)$
- Therefore, $\log_2(n + 1)$ comparisons are made.

□ *The Total Running Time* = $O(\log_2(n + 1))$ or $O(\log_2(n))$

- Where, n is the number of items being searched.

Binary Search – The *Time* Analysis?

(...cont'd)

■ *Average Case Analysis:*

- **No. of Comparisons** = the number is somewhat lower. $\log_2(n - 1)$ is about the expected number of probes in an average successful search.
- **The Total Running Time** = $O(\log_2(n - 1))$ or $O(\log_2(n))$

■ *Best Case Analysis:*

- **No. of Comparisons** = 1 (Only, one element will be tested).
- **The Total Running Time** = $O(1)$

■ **Note:**

Looking more precisely at the practical implementation details, another comparison is needed at each loop cycle to test if the remaining list is diminished to single element. Therefore, 2 or 3 comparisons are made in each iteration of the comparison loop.

■ Binary Search vs. Linear Search:

- In terms of iterations, no search algorithm that is based solely on comparisons can exhibit better average and worst-case performance than binary search.
- Binary search algorithm is more efficient compared to linear search
 - To search array of one million items, we make around 20 comparisons.
 - Linear search would involve about 50,000 comparisons.
- For small number of data items, there is no much difference between linear and binary search times.

■ Binary Search Limitations:

- A major limitation of binary search method is that *the array should be pre-sorted*. This is time-consuming if array contains a very large number of data items.

- Another limitation is that binary search *can not be used with a data structure in which the middle element can not be accessed directly*.
 - For example, it is *not applicable to linked lists*.

Binary Search – The *Performance* Summary?

■ *Time Analysis:*

- Worst Case $O(\log_2 n)$ ← $O(\log_2 n)$ comparisons
- Average Case $O(\log_2 n)$ ← $O(\log_2 n)$ comparisons
- Best Case $O(1)$ ← $O(1)$ comparisons
 - Where, n is the number of items being searched.
- Since logarithmic function grows at a much smaller rate compared to linear function, *Binary Search* is *more efficient than linear search*. However, it only *fits sorted lists*.

■ *Space Analysis:*

- Worst Case Space = $O(1)$ iterative
 - *i.e.*, only requires a constant amount of additional memory space.
 - Where, n is the number of items being searched.

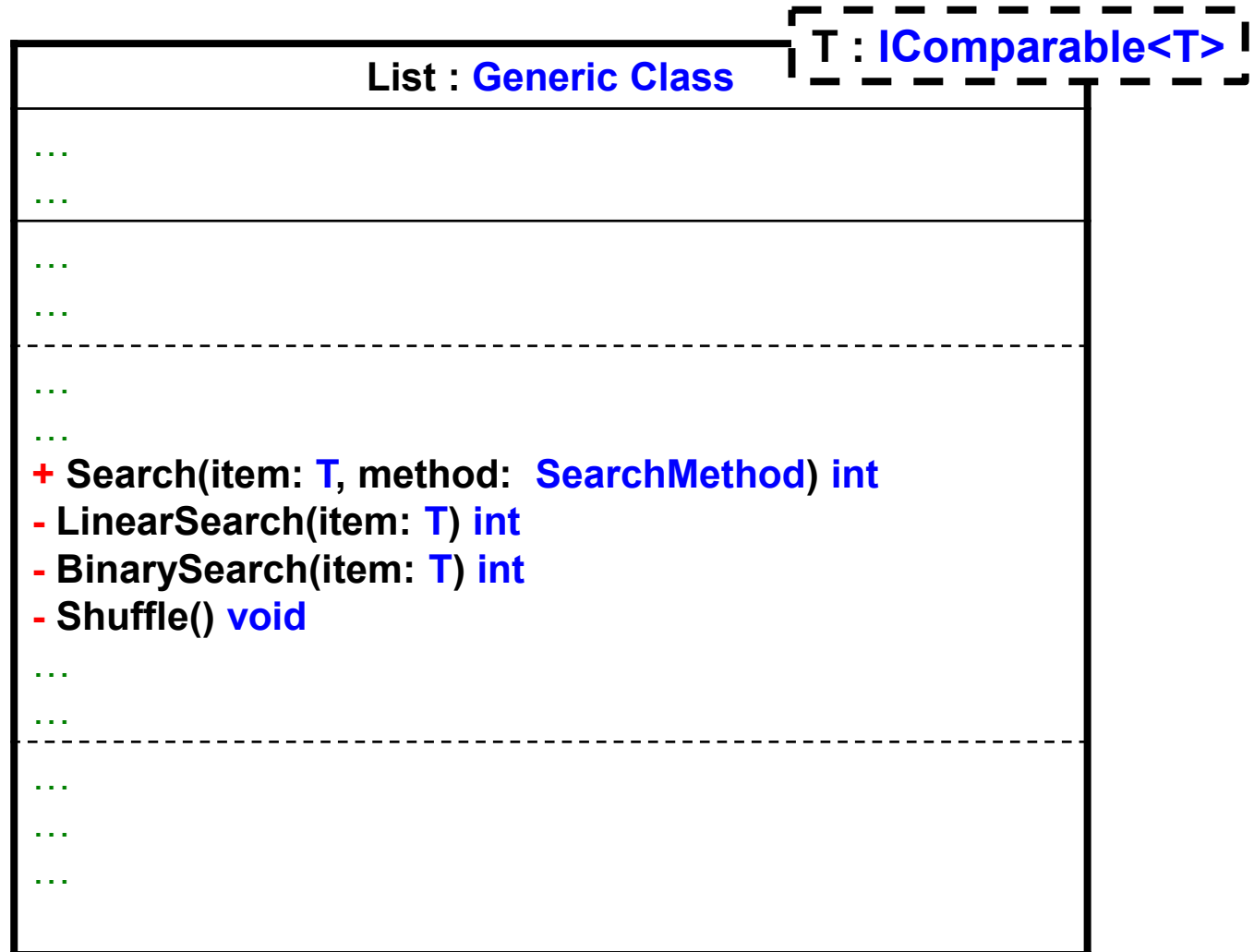
Searching Algorithms – The *List<T>* Class

OOP Implementation

Searching Algorithms – The *List*<T> Class



List <T> Class Diagram

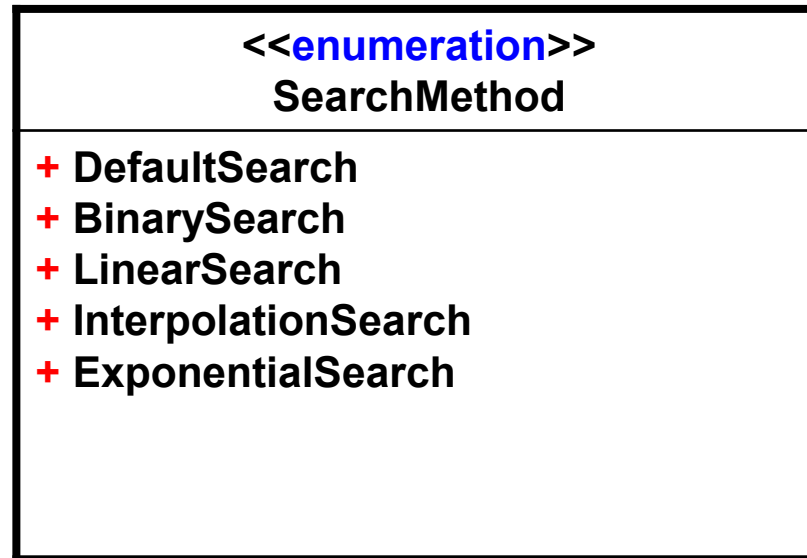


NOTE: The **List** <T> class was introduced earlier in Lecture #3: Sorting Algorithms. Hence, only new/modified elements are shown here.

Searching Algorithms – The *List<T>* Class (...cont'd)




Enumerated Types Diagram



Searching Algorithms – The *List<T>* Class (...cont'd)

```
1  // Searching Algorithms - List<T>() Generic Class.
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20 // TODO: Modify the C# List<T> generic class to support searching
21 //      by adding the items shown in the UML class diagram. Then,
22 //      write a complete "Test Driver" to demonstrate using the
23 //      Search( ) method.
24
```



DEMO



Object-Oriented Implementation of Searching Algorithms
in C# – The List<T> Generic Class.

Lab Assignments

Searching Algorithms – *Lab Assignments*

■ Structured Programming Implementation:

1. Create a ***Search<T>()*** generic static class method that implements one of the studied search techniques on an array of any data type ***T*** that implements the ***Comparable<T>*** interface.

- ***Bonus:*** Create a ***Point3D(x, y, z)*** structure that implements the ***Comparable<T>*** interface. Then, use the ***Search<T>()*** method to search a list of 3D points for the following cases:

1. Key is a specific point $P(x_p, y_p, z_p)$.
2. Key is a specific point that has distance D from the origin.

■ Object-Oriented Programming Implementation:

2. Modify the ***List<T>*** generic class to support the studied searching algorithms on an array member variable of any data type ***T*** that ***Comparable<T>*** interface

- ***Bonus:*** Implement another search technique.

SUMMARY – Lecture Title.

- **Searching** → is the process of locating a given item in a data collection.
- **Searching Methods** → Two classes of methods exist:
 - **Sequential (Linear)** searches, and
 - **Binary (Divide-and-Conquer)** searches.
- **Time Analysis** → $T(n)_{\text{Total}} \approx T(n)_{\text{Comparisons}}$
 - **Sequential Search** → Slowest $O(n)$, but valid for both sorted and unsorted lists .
 - **Binary Search** → Fast $O(\log_2 n)$, but limited to sorted lists.
- **Space Analysis** → Generally, $O(1)$.



Please consider the environment before printing this material.

Now, let's go to the DSA programming lab 😊



Lecture #4: Searching Algorithms

Searching a List of Data for a Particular Value 😊

SUMMARY – Q & A

Feedbacks: email to ameldin@gmail.com. We value your feedback!
(Please include the following prefix in the subject field: **[DSA-C#]**)