UNIT 5

Linear Search Algorithm(Sequential Search)

What is Search?

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Linear Search Algorithm (Sequential Search Algorithm)

- Linear search algorithm finds given element in a list of elements with O(n) time complexity where n is total number of elements in the list.
- This search process starts comparing of search element with the first element in the list.
- If both are matching then results with element found otherwise search element is compared with next element in the list.
- If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list.
- if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

Linear search is implemented using following steps...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the first element in the list.

Step 3: If both are matching, then display "Given element found!!!" and terminate the function

Step 4: If both are not matching, then compare search element with the next element in the list.

Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in the list.

Step 6: If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

Example

Consider the following list of element and search element...

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

search element **12**

Step 1:

search element (12) is compared with first element (65)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
	12							

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
		12						

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
			12					

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
				12				

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
					12			

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
						12		

Both are matching. So we stop comparing and display element found at index 5.

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

Program:

```
#include<stdio.h>
#include<conio.h>

void main(){
  int list[20],size,i,sElement;

  printf("Enter size of the list: ");
  scanf("%d",&size);

  printf("Enter any %d integer values: ",size);
  for(i = 0; i < size; i++)
    scanf("%d",&list[i]);

  printf("Enter the element to be Search: ");
  scanf("%d",&sElement);

  // Linear Search Logic
  for(i = 0; i < size; i++)
  {
     if(sElement == list[i])
     {
        printf("Element is found at %d index", i);
        break;
     }
  }
  if(i == size)
     printf("Given element is not found in the list!!!");
  getch();
}
```

Binary Search Algorithm

What is Search?

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Binary Search Algorithm

- Binary search algorithm finds given element in a list of elements with O(log n) time complexity where n is total number of elements in the list.
- The binary search algorithm can be used with only sorted list of element.
- That means, binary search can be used only with list of element which are already arraged in a order.
- The binary search can not be used for list of element which are in random order.
- This search process starts comparing of the search element with the middle element in the list.
- If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list.
- If the search element is smaller, then we repeat the same process for left sublist of the middle element.
- If the search element is larger, then we repeat the same process for right sublist of the middle element.
- We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps...

Step 1: Read the search element from the user

Step 2: Find the middle element in the sorted list

Step 3: Compare, the search element with the middle element in the sorted list.

Step 4: If both are matching, then display "Given element found!!!" and terminate the function

Step 5: If both are not matching, then check whether the search element is smaller or larger than middle element.

Step 6: If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

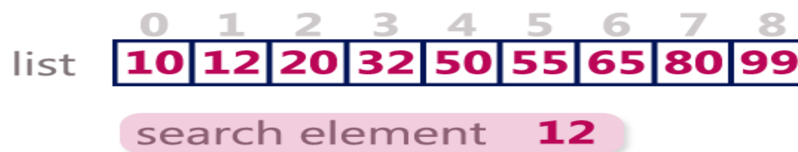
Step 7: If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

Step 8: Repeat the same process until we find the search element in the list or until sublist contains only one element.

Step 9: If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

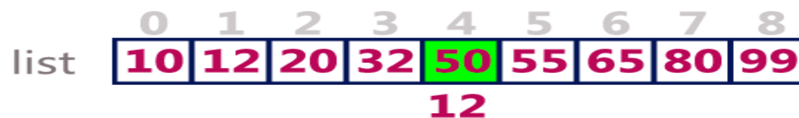
Example

Consider the following list of element and search element...



Step 1:

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

search element **80**

Step 1:

search element (80) is compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

80

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 2:

search element (80) is compared with middle element (65)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

80

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 3:

search element (80) is compared with middle element (80)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

80

Both are not matching. So the result is “Element found at index 7”

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

Program:

```
#include<stdio.h>
#include<conio.h>

void main()
{
  int first, last, middle, size, i, sElement, list[100];
  clrscr();

  printf("Enter the size of the list: ");
  scanf("%d",&size);

  printf("Enter %d integer values in Assending order\n", size);

  for (i = 0; i < size; i++)
    scanf("%d",&list[i]);

  printf("Enter value to be search: ");
  scanf("%d", &sElement);

  first = 0;
  last = size - 1;
  middle = (first+last)/2;

  while (first <= last) {
    if (list[middle] < sElement)
      first = middle + 1;
    else if (list[middle] == sElement) {
      printf("Element found at index %d.\n",middle);
      break;
    }
    else
      last = middle - 1;

    middle = (first + last)/2;
  }
  if (first > last)
    printf("Element Not found in the list.");
  getch();
}
```

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

Insertion Sort

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

- Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Step by Step Process

The insertion sort algorithm is performed using following steps...

Step 1: Assume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.

Step 2: Consider first element from the unsorted list and insert that element into the sorted list in order specified.

Step 3: Repeat the above process until all the elements from the unsorted list are moved into the sorted list.

Sorting Logic

Following is the sample code for insrtion sort...

```
//Insertion sort logic

for i = 1 to size-1 {
  temp = list[i];
  j = i;
  while ((temp < list[j]) && (j > 0)) {
    list[j] = list[j-1];
    j = j - 1;
  }
  list[j] = temp;
}
```

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45

Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Sorted | Unsorted

15	20	10	30	50	18	5	45

Move the first element 15 from unsorted portion to sorted portion of the list.

Sorted | Unsorted

15	20	10	30	50	18	5	45

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

Sorted | Unsorted

15	20	10	30	50	18	5	45

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

Sorted | Unsorted

10	15	18	20	30	50	5	45

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

Sorted | Unsorted

5	10	15	18	20	30	50	45

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.

Sorted | Unsorted

5	10	15	18	20	30	45	50

Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...

5	10	15	18	20	30	45	50

Program:

Selection Sort

- Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending).
- In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list.
- If any element is smaller than the selected element (for Ascending order), then both are swapped.
- Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.

Step by Step Process

The selection sort algorithm is performed using following steps...

Step 1: Select the first element of the list (i.e., Element at first position in the list).
Step 2: Compare the selected element with all other elements in the list.
Step 3: For every comparision, if any element is smaller than selected element (for Ascending order), then these two are swapped.
Step 4: Repeat the same procedure with next position in the list till the entire list is sorted.

Sorting Logic

Following is the sample code for selection sort...

```
//Selection sort logic

for(i=0; i<size; i++){
    for(j=i+1; j<size; j++){
        if(list[i] > list[j])
            {
            temp=list[i];
            list[i]=list[j];
            list[j]=temp;
        }
    }
}
```

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

Example:

Consider the following unsorted list of elements...

Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

15 > 20
FALSE

15 > 10
TRUE
SWAP

10 > 30
FALSE

10 > 50
FALSE

10 > 18
FALSE

10 > 5
TRUE
SWAP

5 > 45
FALSE

List after 1st iteration

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration

5	10	20	30	50	18	15	45

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

5	10	15	30	50	20	18	45

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 4th iteration

5	10	15	18	50	30	20	45

Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 5th iteration

5	10	15	18	20	50	30	45

Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 6th iteration

5	10	15	18	20	30	50	45

Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration

5	10	15	18	20	30	45	50

Final sorted list

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

Complexity of the Insertion Sort Algorithm:

- To sort a unsorted list with 'n' number of elements we need to make ((n-1)+(n-2)+(n-3)+......+1) = (n (n-1))/2 number of comparisons in the worst case. If the list already sorted, then it requires 'n' number of comparisons.

Worst Case : O(n2)
Best Case : Ω(n2)
Average Case : Θ(n2)

Program:

Basic concept of order of complexity:

Performance analysis of an algorithm is the process of calculating space required by that algorithm and time required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures...

1).Space required to complete the task of that algorithm (Space Complexity). It includes program space and data space

2).Time required to complete the task of that algorithm (Time Complexity).

Space Complexity:

What is Space complexity?

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

- Memory required to store program instructions
- Memory required to store constant values
- Memory required to store variable values
- And for few other things

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

Instruction Space: It is the amount of memory used to store compiled version of instructions.

Environmental Stack: It is the amount of memory used to store information of partially executed functions at the time of function call.

Data Space: It is the amount of memory used to store all the variables and constants.

To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following...

2 bytes to store Integer value,

4 bytes to store Floating Point value,

1 byte to store Character value,

6 (OR) 8 bytes to store double value

Example 1

Consider the following piece of code...

```
int square(int a)

{

        return a*a;

}
```

In above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value.

That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be Constant Space Complexity.

Constant Space Complexity :If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be **Constant Space Complexity**

Example 2

Consider the following piece of code...

```
int sum(int A[], int n)
{
  int sum = 0, i;
  for(i = 0; i < n; i++)
    sum = sum + A[i];
  return sum;
}
```

In above piece of code it requires

'n*2' bytes of memory to store array variable 'a[]'

2 bytes of memory for integer parameter 'n'

4 bytes of memory for local integer variables 'sum' and 'i' (2 bytes each)

2 bytes of memory for return value.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the amount of memory depends on the input value of 'n'. This space complexity is said to be Linear Space Complexity.

Linear Space Complexity :If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be **Linear Space Complexity**

Time Complexity

What is Time complexity?

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.

Time complexity of an algorithm can be defined as follows...

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, running time of an algorithm depends upon the following...

- Whether it is running on Single processor machine or Multi processor machine.
- Whether it is a 32 bit machine or 64 bit machine
- Read and Write speed of the machine.
- The time it takes to perform Arithmetic operations, logical operations, return value and assignment operations etc.,
- Input data

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because, the configuration changes from one system to another system. To solve this problem, we must assume a model machine with specific configuration. So that, we can able **to calculate generalized time complexity according to that model machine**.

To calculate time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. Single processor machine
2. 32 bit Operating System machine

3. It performs sequential execution
4. It requires 1 unit of time for Arithmetic and Logical operations
5. It requires 1 unit of time for Assignment and Return value
6. It requires 1 unit of time for Read and Write operations

Now, we calculate the time complexity of following example code by using the above defined model machine...

Example 1

Consider the following piece of code...

```
int sum(int a, int b)
{
  return a+b;
}
```

In above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires same amount of time i.e. 2 units.

Constant Time Complexity.: If any program requires fixed amount of time for all input values then its time complexity is said to be **Constant Time Complexity.**

Example 2

Consider the following piece of code...

```
int sum(int A[], int n)
{
  int sum = 0, i;
  for(i = 0; i < n; i++)
    sum = sum + A[i];
  return sum;
}
```

For the above code, time complexity can be calculated as follows...

Code	Cost (Time require for line (Units))	Repeatation (No. of Times Executed)	Total (Total Time required in worst case)
int sumOfList(int A[ ], int n)			
{			
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1 + 1 + 1	1 + (n+1) + n	2n + 2
sum = sum + A[i];	2	n	2n
return sum;	1	1	1
}			
			4n + 4 (Total Time required)

In above calculation

Cost is the amount of computer time required for a single operation in each line.

Repeatation is the amount of computer time required by each operation for all its repeatations.

Total is the amount of computer time required by each operation to execute.

So above code requires **'4n+4'** Units of computer time to complete the task. Here the exact time is not fixed. And it changes based on the n value. If we increase the n value then the time required also increases linearly.

Totally it takes '4n+4' units of time to complete its execution and it is Linear Time Complexity.

Linear Time Complexity :If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be **Linear Time Complexity**

Best Case,Worst Case and Average Case Efficiencies:

Best Case: It is the minimum number of steps that can be executed for a given problem is know as **Best Case.**

Worst Case: It is the maximum number of steps that can be executed for a given problem is know as **Worst case.**

Average Case: It is the Average number of steps that can be executed for a given problem is know as **Average Case.**