International Islamic University Chittagong
Department of Computer Science & Engineering
Autumn - 2022

Course Code: CSE-2321
Course Title: Data Structures

Mohammed Shamsul Alam
Professor, Dept. of CSE, IIUC

Lecture – 3

Arrays

Linear Arrays

A **linear array** is a list of a ***finite*** number n of ***homogeneous*** data elements (i.e., data elements of the same type) such that:

a) The elements of the array are ***referenced*** respectively by an ***index set*** consisting of n ***consecutive numbers***.

b) The elements of the array are ***stored*** respectively in ***successive memory locations***.

- The number n of elements is called the ***length*** or ***size*** of the array. If not explicitly stated, we will assume the index set consists of the integers 1, 2, …. N. In general the length or the number of data elements can be obtained from the index set by the formula

  Length = UB – LB + 1

  Where **UB** is the largest index, called the ***upper bound*** and **LB** is the smallest index, called the ***lower bound*** of the array.

- The elements of an array may be denoted by the bracket notation (used in Pascal and C/C++)

  A[1], A[2] , A[3], …… , A[n]

  The number **K** in **A[K]** is called a ***subscript*** or an ***index*** and **A[K]** is called a ***subscripted variable***.

Linear Arrays

- *Example 4.1*: Let DATA be a 6-element array of integers such that

  DATA[1] = 247 DATA[2] = 56 DATA[3] = 429 DATA[4] = 135
  DATA[5] = 87 DATA[6] = 156

- The array DATA is frequently pictured as in

DATA

1	247
2	56
3	429
4	135
5	87
6	156

(a)

DATA

247	56	429	135	87	156
1	2	3	4	5	6

(b)

Representation of Linear Arrays in Memory

Let **LA** be a linear array in the memory of the computer. We know, the memory of the computer is simply a sequence of addressed locations. Let us use the notation:

LOC (LA[K]) = address of the element LA[K] of the array LA

As previously noted, the elements of LA are stored in successive memory cells. Accordingly, the computer does not need to keep track of the address of every elements of LA, but needs to keep track only of the address of the first element of LA, denoted by ***BASE (LA)*** and called the ***based address*** of LA. Using this address BASE (LA), the computer calculates the address of any element of LA by the following formula:

$$\mathbf{LOC\ (LA[K]) = BASE(LA) + w\ (\ K - LB)}$$

Where **w** is the number of words per memory cell for the array LA.

- Note that the time to calculate LOC (LA[K)) is essentially the same for any value of **K**. Furthermore, given any subscript K, one can locate and access the content of LA[K] without scanning any other element of LA.

Representation of Linear Arrays in Memory

Example 4.4

Consider the array AUTO in Example 4.1(b), which records the number of automobiles sold each year from 1932 through 1984. Suppose AUTO appears in memory as pictured in Fig. 4.4. That is, *Base*(AUTO) = 200, and w = 4 words per memory cell for AUTO. Then

LOC(AUTO[1932]) = 200, LOC(AUTO[1933]) = 204, LOC(AUTO[1934]) = 208, ...

The address of the array element for the year K = 1965 can be obtained by using Eq. (4.2):

LOC(AUTO[1965]) = *Base*(AUTO) + w(1965 − lower bound)
= 200 + 4(1965 − 1932) = 332

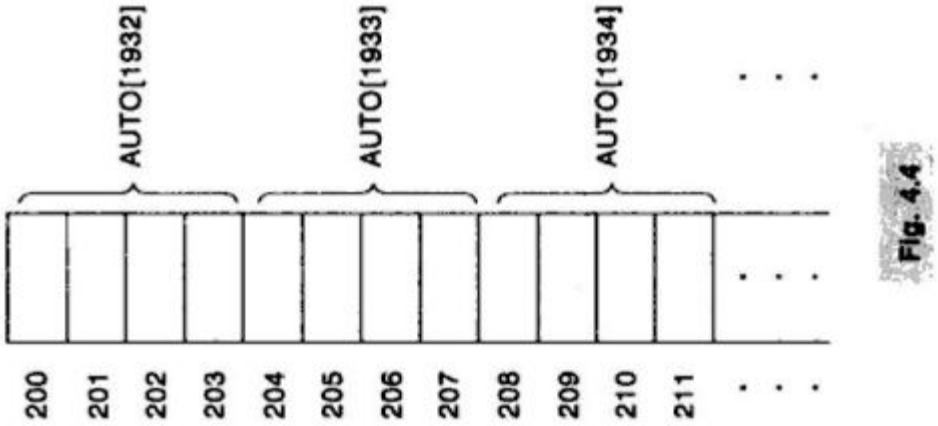


Fig. 4.4

Traversing Linear Arrays

Let ***A*** be a collection of data elements stored in the memory of the computer. Suppose we want to print the contents of each element of ***A*** or suppose we want to count the number of elements of ***A*** with the given property. This can be accomplished by traversing ***A***, by accessing and processing (frequently called *visiting*) each element of ***A*** exactly once.

(**Traversing a Linear array**) Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. Set K := LB
2. Repeat Steps 3 and 4 while K<= UB
   Apply PROCESS to LA[K]
3. Set K := K+1
   [End of Step 2 Loop]
4. Exit.

- We also state an alternative form of the algorithm which uses a repeat-for loop instead of the repeat-while loop.

(**Traversing a Linear array**) This algorithm traverses linear array LA with lower bound LB and upper bound UB.

1. Repeat for K = LB to UB
   Apply PROCESS to LA[K]
2. Exit.

Traversing Linear Arrays

- *Caution:* the operation PROCESS in the traversal algorithm may use certain variables which must be initialized before PROCESS is applied to any of the elements in the array. Accordingly the algorithm may need to be processed by such an initialized step.
- *Example:* Consider the array AUTO, which records the number of automobiles sold each year from 2000 through 2019. Each of the following modules, which carry out the given operation, involves traversing AUTO.

a. **Find the number NUM of years during which more than 300 automobiles were sold**.

1. Set NUM:=0.
2. Repeat for K =2000 to 2019:
   If AUTO[K] > 300, then: Set NUM:-NUM+1.
3. Return.

b. **Print each year and the number of automobiles sold in that year.**

1. Repeat for K=2000 to 2019:
   Write: K, AUTO[K].
2. Return.

Sessional: Write a program to create an array of **n** elements and then separately write the **odd** and **even** elements of the list.

Inserting into a Linear Array

Let ***A*** be a collection of data elements in the memory of the computer. "***Inserting***" refers to the operation of adding another element to the collection ***A***.

Inserting an element at the "end" of a linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array. Then, on the average, half of the elements must be moved downward to new locations to accommodate the new element and keep the order of the other elements.

(**Inserting into a Linear Array**) INSERT (LA, N, K, ITEM)

Here **LA** is a linear array with **N** elements and **K** is a positive integer such that **K ≤ N**. This algorithm inserts an element **ITEM** into the **Kth** position in **LA**.

1. Set J := N
2. Repeat Steps 3 and 4 while J ≥ K
3. Set LA [J+1] := LA [J]
4. Set J := J - 1
5. Set LA [K] := ITEM
6. Set N := N + 1
7. Exit.

Deleting from a Linear Array

Let ***A*** be a collection of data elements in the memory of the computer. "***Deleting***" refers to the operation of removing one of the elements from ***A***.

Deleting an element at the "end" of an array presents no difficulties, but deleting an element somewhere in the middle of the array would require that each subsequent element be moved one location upward in order to "fill up" the array.

(**Deleting from a Linear Array**) DELETE (LA, N, K. ITEM)

Here **LA** is a linear array with **N** elements and **K** is a positive integer such that **K ≤ N**. This algorithm deletes the **Kth** clement from **LA.**

I. Set ITEM := LA [K]

2. Repeat for J = K to N - 1:

    Set LA[J] := LA[J + 1]

3. Set N := N - 1.

4. Exit.

Sessional:

1. Write a program to create an array of n elements and then insert an element to the list.
2. Write a program to create an array of n elements and then delete an element from the list.

Sorting: Bubble Sort

Let A be a list of *n* numbers. ***Sorting*** A refers to the operation of rearranging the elements of A so they are in increasing order, i.e., so that

A[1] < A[2] < A[3] < . . . < A[N]

For example, suppose A originally is the list

8, 4, 19, 2, 7, 13, 5, 16

After sorting, A is the list

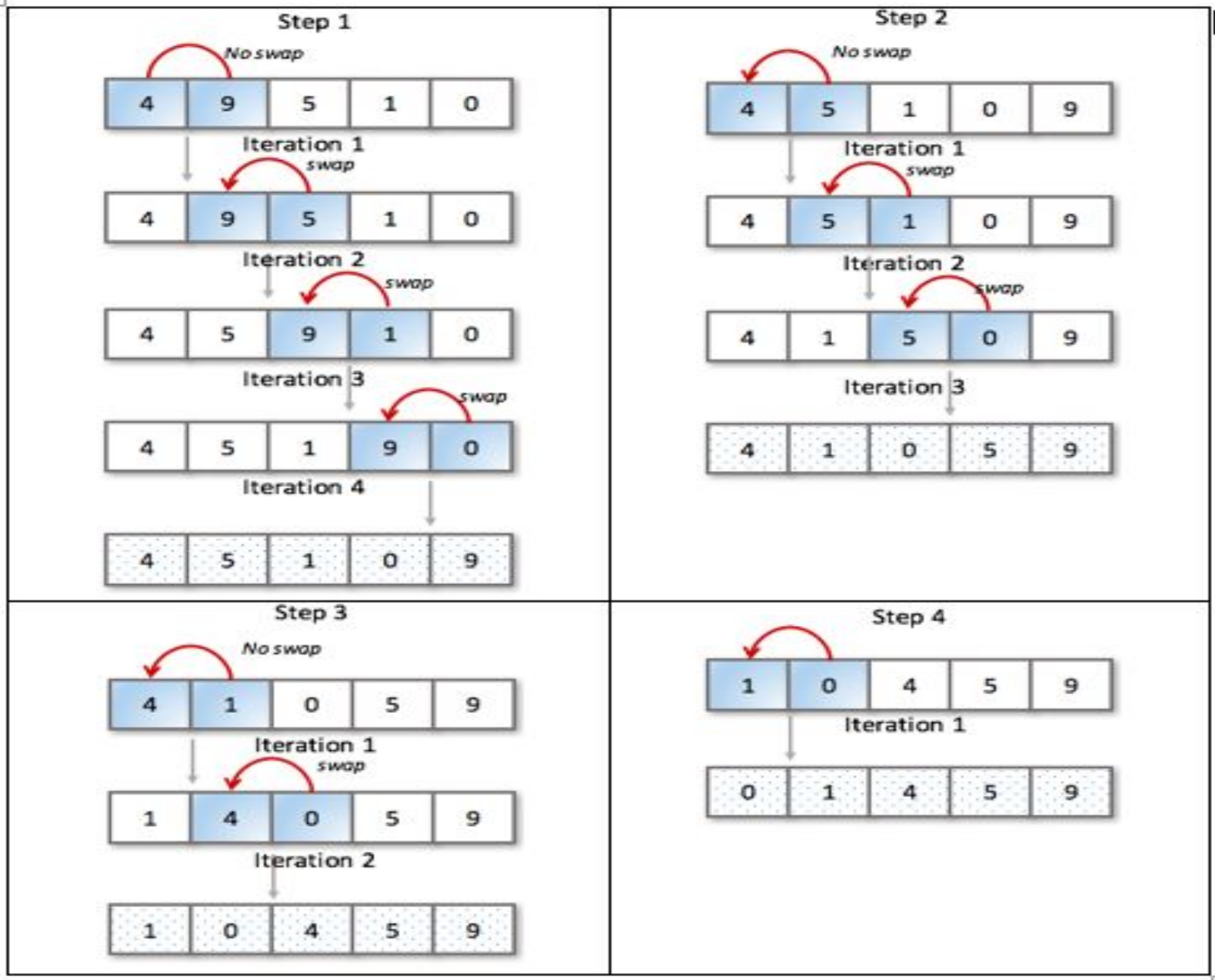
2, 4, 5, 7, 3, 13, 16, 19.

(**Bubble Sort**) BUBBLE (DATA, N)

Here **DATA** is an array with **N** elements. This algorithm sorts the elements in **DATA**.

1. Repeat Steps 2 and 3 for K = 1 to N - 1
2. Set PTR := 1
3. Repeat while PTR <= N - K:
   (a) If DATA[PTR] > DATA[PTR+1], then:
   Interchange DATA[PTR] and DATA[PTR+1]
   (b) Set PTR := PTR+1
4. Exit.

Example of Bubble Sort



Sorting: Bubble Sort

Example 4.7

Suppose the following numbers are stored in an array A:

32, 51, 27, 85, 66, 23, 13, 57

We apply the bubble sort to the array A. We discuss each pass separately.
Pass 1. We have the following comparisons:

(a) Compare A_1 and A_2. Since 32 < 51, the list is not altered.
(b) Compare A_2 and A_3. Since 51 > 27, interchange 51 and 27 as follows:

32, (27, (51, 85, 66, 23, 13, 57

(c) Compare A_3 and A_4. Since 51 < 85, the list is not altered.
(d) Compare A_4 and A_5. Since 85 > 66, interchange 85 and 86 as follows:

32, 27, 51, (66, (85, 23, 13, 57

(e) Compare A_5 and A_6. Since 85 > 23, interchange 85 and 23 as follows:

32, 27, 51, 66, (23, (85, 13, 57

(f) Compare A_6 and A_7. Since 85 > 13, interchange 85 and 13 to yield:

32, 27, 51, 66, 23, (13, (85, 57

(g) Compare A_7 and A_8. Since 85 > 57, interchange 85 and 57 to yield:

32, 27, 51, 66, 23, 13, (57, (85

At the end of this first pass, the largest number, 85, has moved to the last position. However, the rest of the numbers are not sorted, even though some of them have changed their positions.

For the remainder of the passes, we show only the interchanges.

Pass 2. (27, 33,) 51, 66, 23, 13, 57, 85

27, 33, 51, (23, 66,) 13, 57, 85

27, 33, 51, 23, (13, 66,) 57, 85

27, 33, 51, 23, 13, (57, 66,) 85

At the end of Pass 2, the second largest number, 66, has moved its way down to the next-to-last position.

Pass 3. 27, 33, (23, 51,) 13, 57, 66, 85

27, 33, 23, (13, 51,) 57, 66, 85

Pass 4. 27, (23, 33,) 13, 51, 57, 66, 85

27, 23, (13, 33,) 51, 57, 66, 85

Pass 5. (23, 27,) 13, 33, 51, 57, 66, 85

23, (13, 27,) 33, 51, 57, 66, 85

Pass 6. (13, 23,) 27, 33, 51, 57, 66, 85

Pass 6 actually has two comparisons, A_1 with A_2 and A_2 and A_3. The second comparison does not involve an interchange.

Pass 7. Finally, A_1 is compared with A_2. Since $13 < 23$, no interchange takes place.

Since the list has 8 elements; it is sorted after the seventh pass. (Observe that in this example, the list was actually sorted after the sixth pass. This condition is discussed at the end of the section.)

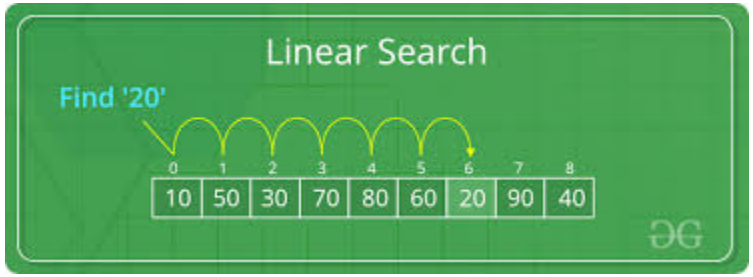
Searching

Let DATA be a collection of data elements in memory, and suppose a specific ITEM of information is given. *Searching* refers to the operation of finding the location LOC of ITEM in DATA, or printing some message that ITEM does not appear there. The search is said to be ***successful*** if ITEM does appear in DATA and ***unsuccessful*** otherwise.

There are many different searching algorithms. The algorithm that one chooses generally depends on the way the information in DATA is organized. Here we will discus two algorithms called **linear search** and **binary search**.

Linear Search

Suppose DATA is a linear array with n elements. Given no other information about DATA, the most intuitive way to search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one. That is, first we test whether DATA[1] = ITEM, and then we test whether DATA[2] = ITEM, and so on. This method, which traverses DATA sequentially to locate ITEM, is called ***linear search*** or ***sequential search***.



Linear Search

(**Linear Search**) A linear array **DATA** with **N** elements and a specific **ITEM** of information are given. This algorithm finds the location **LOC** of **ITEM** in the array **DATA** or sets **LOC=0**.

I. Set *K := 1* and LOC := 0
2. Repeat Steps 3 arid 4 while LOC = 0 and *K* <= *N*
3. If ITEM = DATA [K], then: Set LOC := K
4. Set K := K + 1
5. If LOC = 0, then:
   Write: ITEM is not in the array DATA
   Else:
   Write: LOC is the location of ITEM
6. Exit.

Linear Search

The following algorithm is suitable when one want to add the element ITEM to DATA after an unsuccessful search for ITEM in DATA.

(**Linear Search**) LINEAR (DATA, N. ITEM, LOC)

Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA or sets LOC = 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA] Set DATA [N + 1] := ITEM
2. Set LOC :=1
3. Repeat while DATA [LOC] ≠ ITEM:
   Set LOC:= LOC + 1.
4. [Successful?] If LOC=N+ 1, then: Set LOC := 0
5. Exit.

Remark: Here, Step-1 guarantees that the loop in step 3 must terminate.

Binary Search

Suppose DATA is an array which is sorted in increasing numerical order or, equivalently, alphabetically. Then there is an extremely efficient searching algorithm, called ***binary search***, which can be used to find the location LOC of a given ITEM of information in DATA.

(**Binary Search**) BINARY (DATA, LB, UB, ITEM, LOC)

Here DATA is a sorted array with lower bound LB and upper bound UB and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC=NULL.

1. Set BEG := LB, END := UB and MID := INT((BEG+END)/2)
2. Repeat Steps 3 and 4 while BEG ≤ END and DATA [MID] ≠ ITEM
3. If ITEM < DATA [MID], then:
   Set END := MID-1
   Else:
   Set BEG := MID+1
4. Set MID := INT((BEG+END)/2)
5. If DATA [MID] = ITEM, then:
   Set LOC := MID
   Else:
   Set LOC := NULL
6. Exit.

Binary Search

Example 4.9

Let DATA be the following sorted 13-element array:

DATA: 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

We apply the binary search to DATA for different values of ITEM.

(a) Suppose ITEM = 40. The search for ITEM in the array DATA is pictured in Fig. 4.6, where the values of DATA[BEG] and DATA[END] in each stage of the algorithm are indicated by circles and the value of DATA[MID] by a square. Specifically, BEG, END and MID will have the following successive values:

1. Initially, BEG = 1 and END = 13. Hence
   MID = INT[(1 + 13)/2] = 7 and so DATA[MID] = 55
2. Since 40 < 55, END has its value changed by END = MID − 1 = 6. Hence
   MID = INT[(1 + 6)/2] = 3 and so DATA[MID] = 30
3. Since 40 > 30, BEG has its value changed by BEG = MID + 1 = 4. Hence
   MID = INT[(4 + 6)/2] = 5 and so DATA[MID] = 40

We have found ITEM in location LOC = MID = 5.

(1) (11) 22, 30, 33, 40, 44, [55], 60, 66, 77, 80, 88, (99)
(2) (11) 22, [30], 33, 40, (44) 55, 60, 66, 77, 80, 88, 99
(3) 11, 22, 30, (33) [40], (44) 55, 60, 66, 77, 80, 88, 99 [Successful]

Fig. 4.6 *Binary Search for ITEM = 40*

Binary Search

(b) Suppose ITEM = 85. The binary search for ITEM is pictured in Fig. 4.7. Here BEG, END and MID will have the following successive values:

1. Again initially, BEG = 1, END = 13, MID = 7 and DATA[MID] = 55.
2. Since 85 > 55, BEG has its value changed by BEG = MID + 1 = 8. Hence

   MID = INT[(8 + 13)/2] = 10 and so DATA[MID] = 77

3. Since 85 > 77, BEG has its value changed by BEG = MID + 1 = 11. Hence

   MID = INT[(11 + 13)/2] = 12 and so DATA[MID] = 88

4. Since 85 < 88, END has its value changed by END = MID − 1 = 11. Hence

   MID = INT[(11 + 11)/2] = 11 and so DATA[MID] = 80

   (Observe that now BEG = END = MID = 11.)

Since 85 > 80, BEG has its value changed by BEG = MID + 1 = 12. But now BEG > END. Hence ITEM does not belong to DATA.

(1) (11), 22, 30, 33, 40, 44, [55], 60, 66, 77, 80, 88, (99)

(2) 11, 22, 30, 33, 40, 44, 55, (60), 66, [77], 80, 88, (99)

(3) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, (80), [88], (99)

(4) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, (80), 88, 99 [Unsuccessful]

Fig. 4.7 *Binary Search for ITEM = 85*

Remark: Whenever ITEM does not appear in DATA, the algorithm eventually arrives at the stage that BEG = END = MID. Then the next step yields END < BEG, and control transfers to Step 5 of the algorithm.

Binary Search

Limitations of the Binary Search Algorithm

a. The list must be sorted and

b. One must have direct access to the middle element to any sublist.

This means that one must essentially use a sorted array to hold the data. But keeping data in a sorted array is normally very expensive when there are many insertions and deletions. Accordingly, in such situations, one may use a different data structure, such as a linked list or a binary search tree, to store the data.

Sessional:

1. Write a program to sort n numbers using Bubble Sort algorithm.
2. Write a program to search an element from a list of n numbers using Linear Search algorithm.
3. Write a program to search an element from a list of n numbers using Binary Search algorithm.