ICE-2231
(Data Structures and Algorithms)

Lecture on
Chapter-2: Arrays, Records, Pointers

By

Dr. M. Golam Rashed

(golamrashed@ru.ac.bd)

Department of Information and Communication Engineering (ICE)
University of Rajshahi, Rajshahi-6205, Bangladesh

Data structures are classified as either *Linear* or *Nonlinear*.

- A data structure is said to be *Linear* if its elements forms a sequence, or a linear list.
- There are **TWO** basic ways of representing such *linear structures* in memory.
  - One ways is to have the linear relationship between the elements represented by means of sequential memory locations. (For example, ARRAYS).
  - The other ways is to have the linear relationship between the elements represented by mean of pointers or links. (For example, linked lists)

- *Nonlinear data structures (For example, tress and graphs) discussed later.*

Operations on Linear Structure

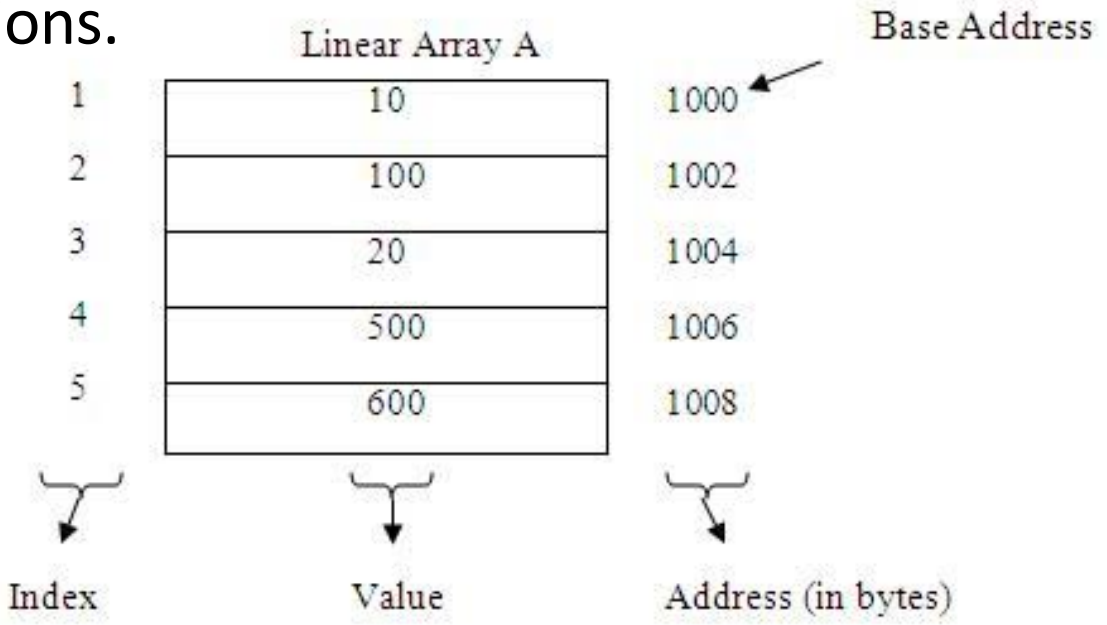
The operations one normally performed on any linear structure include the following:

- *Traversing*-Processing each element in the list
- *Search* – Finding the location of the element with a given value or the record with a given key.
- *Insertion*-Adding a new element to the list.
- *Deletion*-Removing an element from the list.
- *Sorting*-Arranging the element in some type of order.
- *Merging*-Combining two list into a single list.

Linear Arrays

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

- The elements of the array are referenced respectively by an index set consisting of *n* consecutive numbers.
- The elements of the array are stored respectively in successive memory locations.



Linear Arrays

- The number *n* of elements is called **the length or size of the array**.
- In general, the length or the number of data elements of the array can be obtained from the index set by the formula.

$$\text{Length} = \text{UB} - \text{LB} + 1$$

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

Note that, Length=UB when LB=1.

Linear Arrays: Example

AMPLE 4.1

Let DATA be a 6-element linear array of integers such that

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

Sometimes we will denote such an array by simply writing

DATA: 247, 56, 429, 135, 87, 156

The array DATA is frequently pictured as in Fig. 4-1(*a*) or Fig. 4-1(*b*).

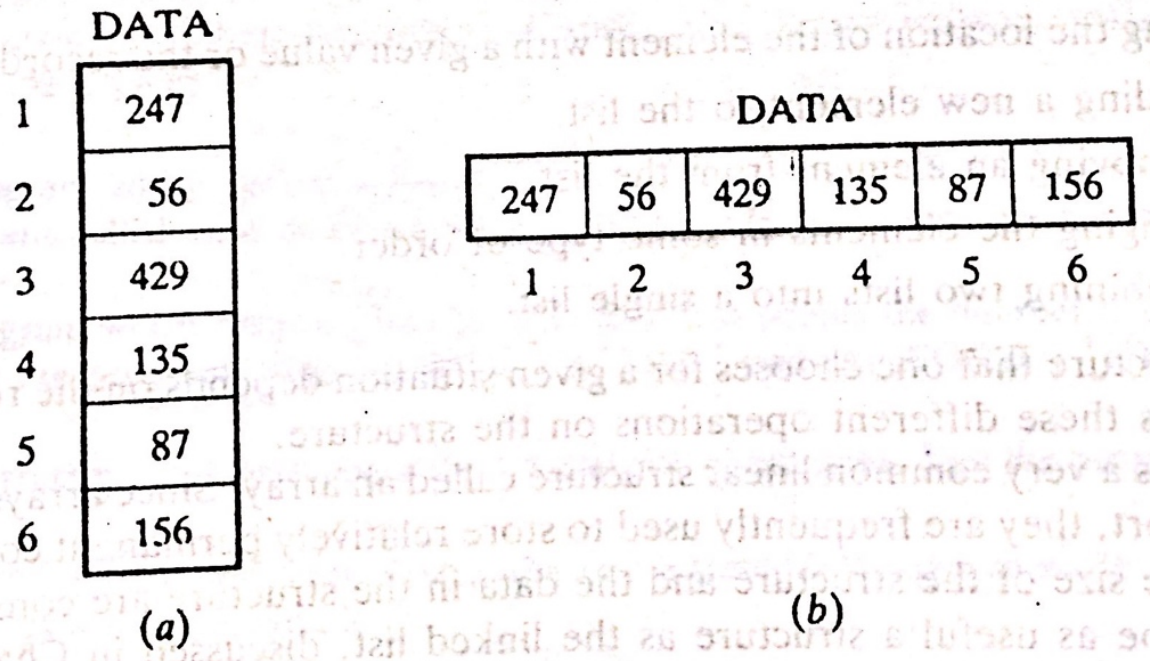


Fig. 4-1

Linear Arrays: Indexing Example

- An automobile company uses an array AUTO to record the number of automobiles sold each year from 1930 through 1984.

- Rather than beginning the index set with 1, it is more useful to begin the index set with 1932 so that

  AUTO[K]=Number of automobiles sold in the year K,

  Then, LB=1932 and UB=1984 of AUTO

  Length=UB-LB+1
  =1984-1930+1=55

- On implementation, each programming language has its own rules for declaring arrays. Each such declaration must give, implicitly , THREE items of information,
  - The **NAME** of the array,
  - The **DATA TYPES** of the array
  - The **INDEX SET** of the array

Representation of LA in Memory

- ✓ Let LA be a linear array in the memory of the computer.

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

- ✓ The elements of LA are stored in successive memory cells.

- ✓ Accordingly, the computer does not need to keep track of the address of every element of LA, But needs to keep track only the address of the first element of LA.
- ✓ Denoted by

Base*(LA)*-called the base address of LA

Representation of LA in Memory

- ✓ Using the base address of LA, the computer calculates the address of any element of LA by the following formula:

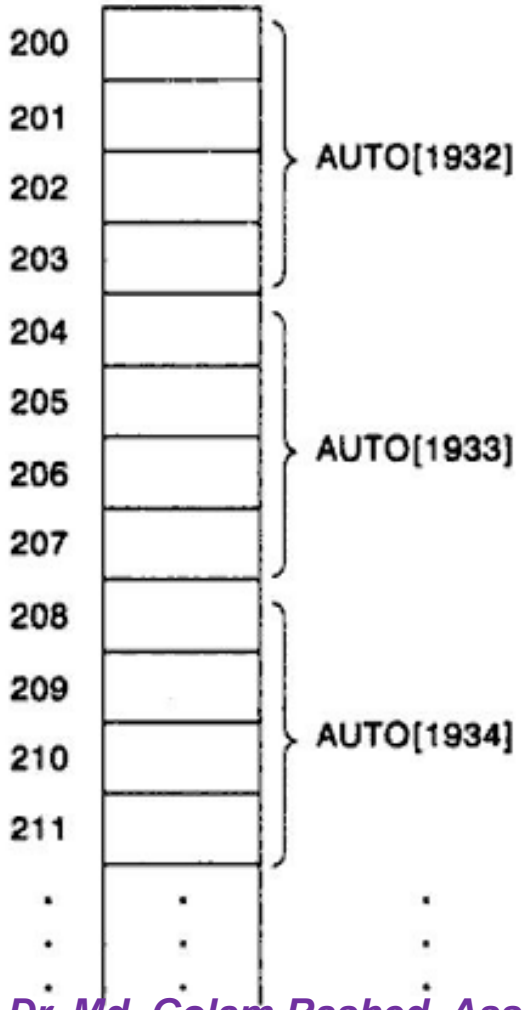
$$LOC(LA[K]) = Base(LA) + w(K\text{-lower bound})$$

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

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

Example: Representation of LA in Memory

- Consider the array AUTO which records the number of automobiles sold each year from 1932 through 1984.
- Suppose AUTO appears in memory as pictured below



- Here, Base (AUTO)=200, and
- w=4 words per memory cell for AUTO.

LOC (AUTO[1932])= 200

LOC (AUTO[1933])= 204

LOC (AUTO[1934])= 208

- Address of the array element for the year K=1965 can be obtained:

LOC(AUTO[1965])=**Base(AUTO)+w(1965-lower bound)**

= 200+4(1965-1932)=332

LOC(LA[K])=*Base*(LA)+w(K-lower bound)

Can Linear Arrays be indexed?

- A collection A of data elements is said to be indexed if any element of A, which we shall call A_k , can be located and processed in the time that is independent of K.
- This is very important property of linear arrays

Traversing Linear Arrays A: Algorithm

A is a linear array with LB and UB

Step 1. [Initialize counter] Set K:=LB
Step 2. Repeat Step 3 and 4 while K<=UB [Repeat while loop]
Step 3. [Visit element] Apply PROCESS to A[K].
Step 4. [Increase counter] Set K:=K+1.
[End of Step 2 loop.]
Step 5. Exit.

Step 1. Repeat for K=LB to UB [Repeat for loop]
[Visit element] Apply PROCESS to A[K].
[End of Step 2 loop.]
Step 2. Exit.

Traversing Linear Arrays: Example

Consider the array AUTO in Example 4.1(*b*), which records the number of automobiles sold each year from 1932 through 1984. 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.

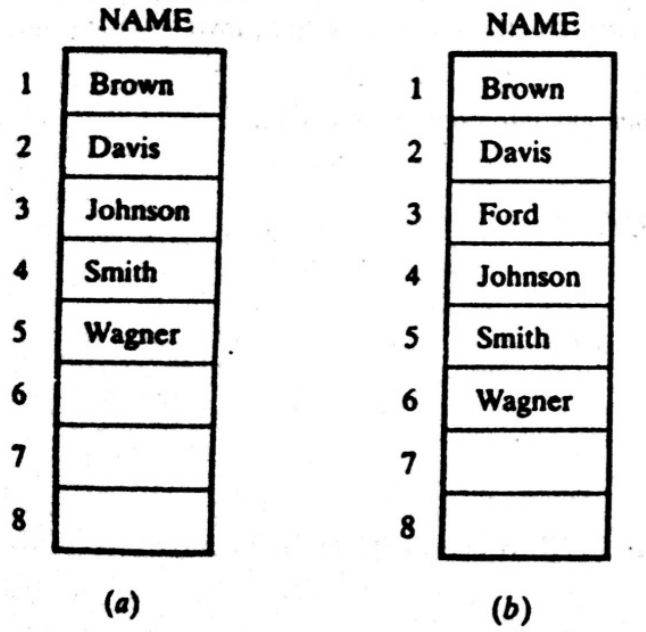
1. [Initialization step.] Set NUM := 0.
2. Repeat for K = 1932 to 1984:
   If AUTO[K] > 300, then: Set NUM := NUM + 1.
   [End of loop.]
3. Return.

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

1. Repeat for K = 1932 to 1984:
   Write: K, AUTO[K].
   [End of loop.]
2. Return.

Inserting elements in a Linear Arrays:

- ✓ Let A be a collection of data elements in the computer memory.
- ✓ Inserting refer to the operation of **ADDING** another element to the collection.
- ✓ Inserting an element at the "end" of the linear array can be **EASILY** done provided the memory space allocating for the array is large enough to accommodate the additional element.
- ✓ Inserting an element in the middle of the array is **RELATIVELY COMPLICATED TASK.**
- ✓ On an average, half of the elements must be moved downward to new locations to accommodate the elements and keep the order of the other elements.



Inserting elements in a LA: Algorithm

INSERT(LA, N, K, ITEM) [**inserts an element ITEM into Kth position in LA**]

1. [Initialize counter] Set J:=N
2. Repeat Step 3 and 4 while J>=K.
3. [Move J^{th} element downward.] Set LA[J+1]:=LA[J]
4. [Decrease counter] Set J:=J-1.
   [End of Step 2 loop.]
5. [Insert element.] Set LA[K]:=ITEM.
6. [Reset N.] Set N:=N+1.
7. Exit.

- The elements are moved in reverse order . First LA[N], then LA[N-1],……and last LA[K]; otherwise data might be erased.

Instant Test-1:

Consider the linear arrays:

XXX (10 : 55),
YYY (-10 : 15), and
ZZZ (25)

a) Find the number of element in each array.

We know Length=UB-LB+1

Accordingly, Length (XXX) = 55 - 10 + 1= 46
Length (YYY) = 15-(-10)+1 = 26
Length (ZZZ) = 25-1+1= 25

Instant Test-2:

Consider the linear arrays:

XXX (10 : 65),
YYY (-10 : 15), and
ZZZ (25)

b) Suppose Base (XXX) = 300 and w= 4 words per memory cell for XXX.

- **Find the address of XXX [15]**
  XXX [35]
  XXX [75]

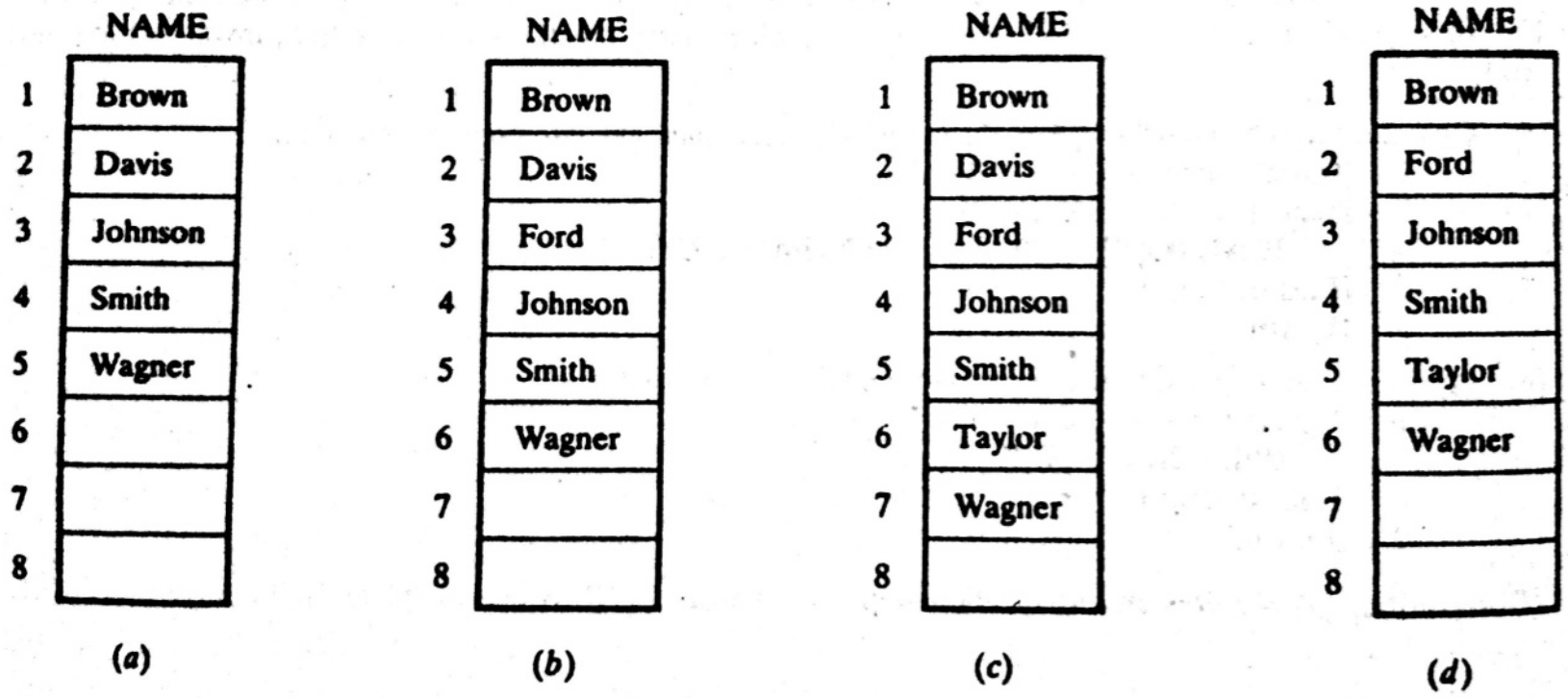
We know the formula: LOC (XXX[k])= Base (XXX)+w (k-LB)

Hence, LOC (XXX[15])= 300 + 4 (15-10) = 320
LOC (XXX [35])= 300 + 4 (35-10)= 400

AAA [75] is not an element of XXX, since 75 exceeds UB=65

Deleting elements from a Linear Arrays:

- ✓ Deleting refers to the operation of removing one of the elements from A.
- ✓ Deleting an element at the "end" of the linear array present no difficulties (**EASILY).**
- ✓ But deleting an element somewhere in the middle of the array would require that each of the subsequent element be moved one location upward in order to fill-up the array.



Deleting elements from a LA: Algorithm

DELETE(LA, N, K, ITEM)

(This algorithm deletes the Kth element from LA)

1. Set ITEM:=LA[K]
2. Repeat for J=K to N-1:

   [Move J+1th element upward.] Set LA[J]:=LA[J+1]

   [End of Step 2 loop.]
3. [Reset the number N of element in LA] Set N:=N-1.
4. Exit.

Sorting

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[a]<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, 8, 13, 16, 19

Sorting: BUBBLE SORT

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 66 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 the first pass, the largest number ,85 has moved to the last position.

Rest of the number are not sorted.

Sorting: BUBBLE SORT

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 positi

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 involve an interchange.

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

...after the seventh pass. (Observe that in this example, the list was act

Since the list has 8 elements, it is sorted after the seventh pass.

Bubble Sort: Algorithm

BUBBLE (DATA, N)

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

Step 1. Repeat Steps 2 and 3 for K=1 to N-1.

Step 2. Set PTR:=1 [Initialize pass pointer PTR]

Step 3. Repeat while PTR<=N-K [Execute pass.]

(a) If DATA[PTR] > DATA[PTR+1], then:

Interchange Data[PTR] and DATA[PTR+1].

[End of IF Structure]

(b) Set PTR:=PTR+1.

[End of inner loop.]

[End of Step 1. outer loop.]

Step 4. Exit.

Complexity of BUBBLE SORT

- Traditionally, the time for this sorting algorithm is measured in terms of the number of comparisons.
- The number f(n) of comparisons in the bubble sort is easily computed.
- Specifically, there are n-1 comparisons during the first pass, which placed the largest element to the last position; there are n-2 comparisons in the second step, which placed the second largest element in the next-to-the last position, and so on. Thus.

$$F(n)=(n-1)+(n-2)+....+2+1$$
$$=n(n-1)/2=n^2/2+O(n)$$
$$=O(n^2)$$

Searching

- ✓ 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 is DATA is organized.
- ✓ A simple searching algorithm: Linear Search Algorithm
- ✓ The well known algorithm: Binary search Algorithm

Linear Search Algorithm

- ✓ Suppose DATA is a linear array with n elements. Given no other information about DATA.
- ✓ Simple way to search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one.
- ✓ Suppose we want to know whether Jhon appears in the array or not.
- ✓ Again, Suppose, we want to know whether Moon appears in the array or not.

Adams
Charlie
Rasha
Moon
Rock
Smith

Adams
Charlie
Rasha
Moon
Rock
Smith
Jhon

Adams
Charlie
Rasha
Moon
Rock
Smith
Moon

Linear Search Algorithm: Example

search element **12**

Step 1:

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

Both are not matching. So move to next element

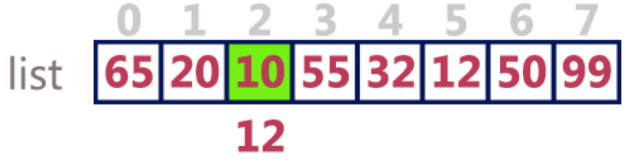
Step 2:

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

Both are not matching. So move to next element

Step 3:

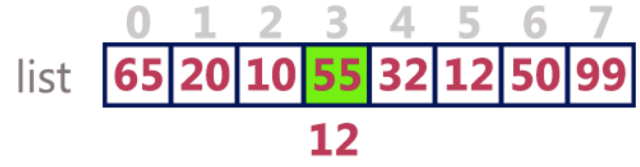
search element (12) is compared with next element (10)



Both are not matching. So move to next element

Step 4:

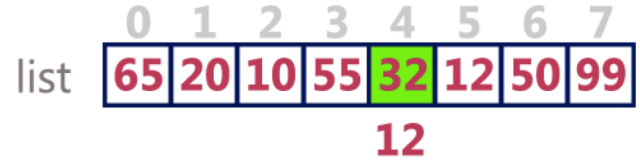
search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

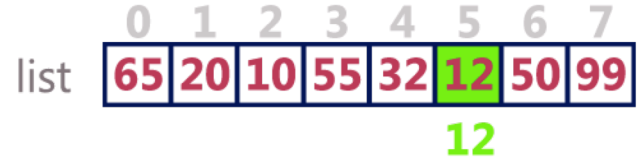
search element (12) is compared with next element (32)



Both are not matching. So move to next element

Step 6:

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



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

Searching: Linear Search Algorithm

LINEAR (DATA, N, ITEM, LOC)

Step 1. [Insert ITEM at the end of DATA] Set DATA[N+1]:=ITEM

Step 2. [Initialize counter] Set LOC:=1.

Step 3. [Search for ITEM.]

Repeat while Data [LOC]$\neq ITEM$:

Set LOC:=LOC+1.

[End of loop.]

Step 4. [Successful?] IF LOC=N+1, then: Set LOC:=0;

Step 5. Exit.

Linear Search Algorithm: Complexity

- ✓ The complexity of this algorithm is measured by the number f(n) of comparison required to find ITEM where DATA contains n elements.
- ✓ Two important cases to consider are:
  - ✓ **The average case**, and
  - ✓ **The worst case**

The running time of the average case uses the probabilistic notation of expectation.

- ✓ Suppose, p_k is the probability that ITEM appears in DATA[K], and
- ✓ suppose, q is the probability that ITEM does not appear in DATA.
- ✓ Since, the algorithm uses k comparisons when ITEM appears in DATA [K], the average number of comparisons is given by

$$f(n)= 1.p_1+2.p_2+.....n.p_n+(n+1)q$$

- ✓ In particular, q is very small, and ITEM appears with equl probability in each element of DATA. Then $q \approx 0$ and each $p_i=1/n$.
- ✓ Accordingly $f(n) = 1.\frac{1}{n} + 2.\frac{1}{n} + 3.\frac{1}{n} + \ldots\ldots + n.\frac{1}{n} + (n+1).0$
- ✓ $$= (1+2+\ldots+n).\frac{1}{n} = \frac{n(n+1)}{2}.\frac{1}{n} = \frac{n+1}{2}$$

Linear Search Algorithm: Complexity

The worst case occurs when one must search through the entire array DATA, when ITEM does not appear in DATA.

Algorithm Requires f(n)= *n+1* comparison.

Thus, in the worst case, the running time is proportional to *n*