Book

A Simplified Approach to Data Structures

Prof.(Dr.)Vishal Goyal, Professor, Punjabi University Patiala

Dr. Lalit Goyal, Associate Professor, DAV College, Jalandhar

Mr. Pawan Kumar, Assistant Professor, DAV College, Bhatinda

Shroff Publications and Distributors

Edition 2014

ONE DIMENSIONAL ARRAYS

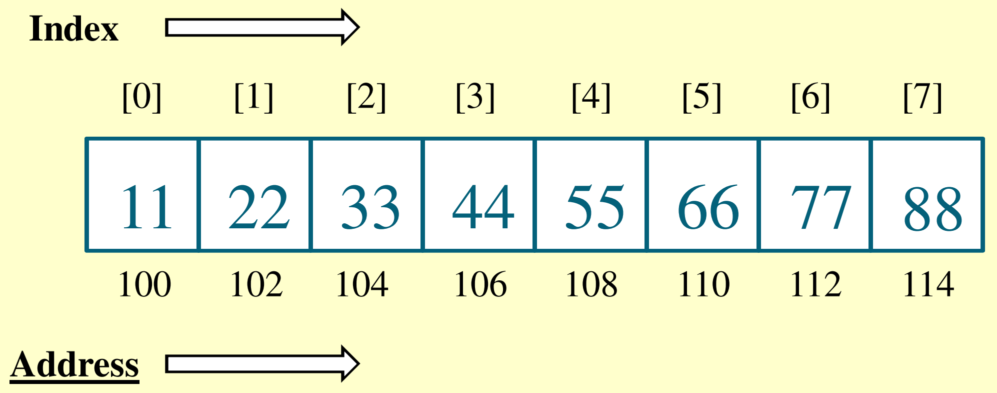
OUTLINE

- Introduction to Arrays
- One Dimensional Arrays
- Memory representation of one dimensional arrays
- Operations performed on arrays

INTRODUCTION TO ARRAYS

- An array is the linear collection of **finite number** of **homogeneous** data elements.
- Although some programming languages accept arrays in which elements are of different types.
- Elements of the array are referenced by an index set consisting of 'n' consecutive numbers. We can refer to the elements in the array as the first element, the second element and so forth, until we get to the last element.
- An array is the most efficient data structure for storing and accessing a sequence of objects.
- Array elements are stored in successive memory locations.
- The array are also popular with the name **Vectors** and **Tables** in mathematics.

INTRODUCTION TO ARRAYS



Array with 8 elements

- Indices are used to access the stored values in an array.
- Let’s say we have an array of “n” integers, then its first integer element is indexed with the “0” value and the last integer element will be represented with “n-1” indexed value.

Example: if we have 8 elements then the first element would be referenced by a[0] i.e. The first index value ,then other distinct values in the array are each referenced as:

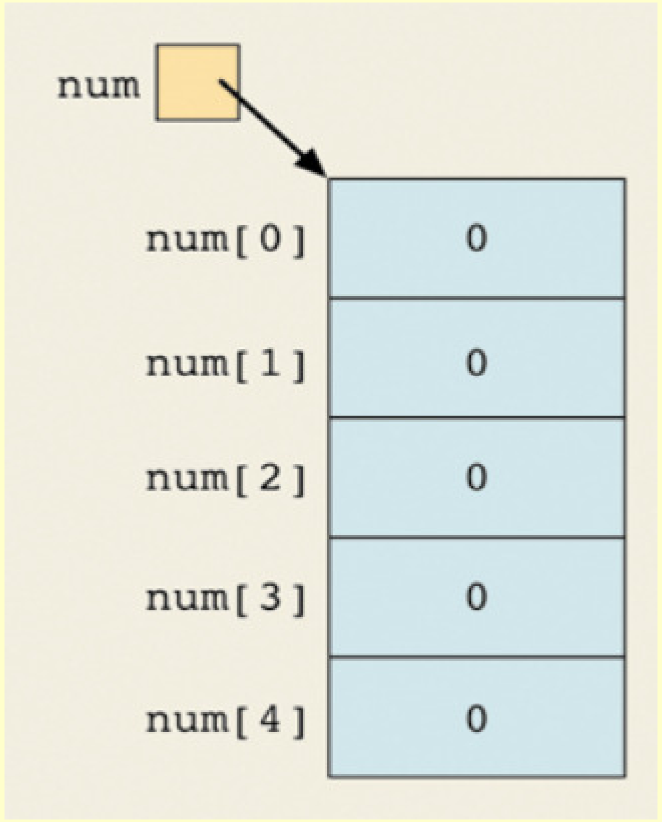
a[0]=1, a[1]=2, ...a[n-1]5=n i.e. a[7]=8 where “n-1” becomes 7 where “n” was 8 .

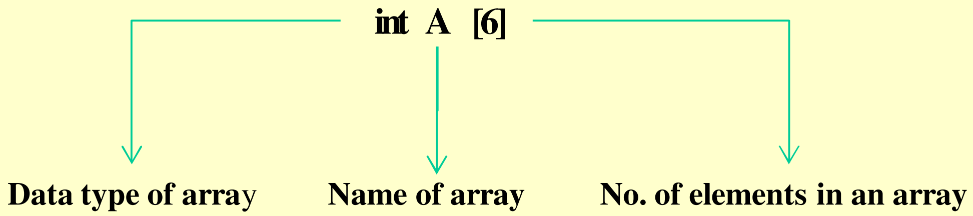
Example of Array

- **Syntax to declare an array:**

dataType[ ] arrayName;
arrayName = new dataType[N]

- **int[ ] num = new int[5];**





ONE DIMENSIONAL ARRAY

- One dimensional array or linear array is a list of a finite number *n* of homogeneous data elements such that:

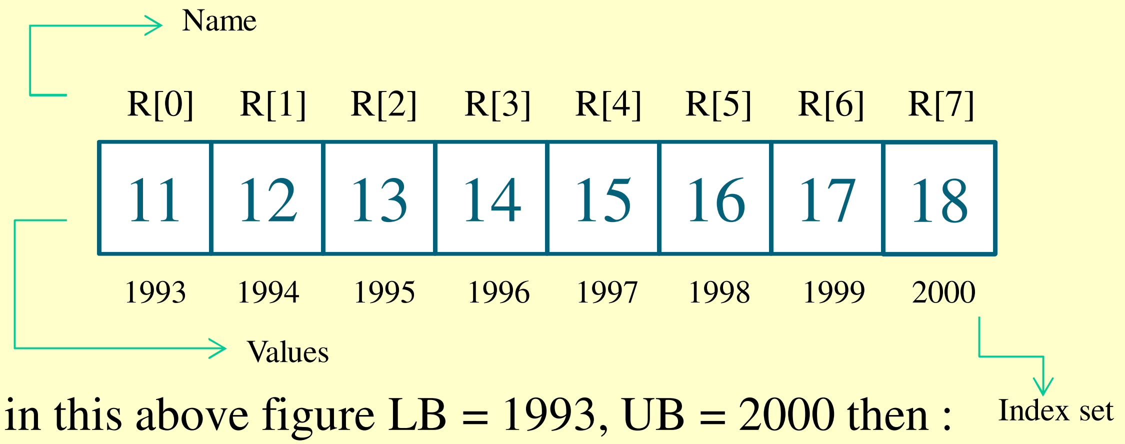
  a) The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.
  b) The elements of the array are stored respectively in successive contiguous memory locations.

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

$$A \text{ (length of array)} = UB - LB + 1$$

(UB is the largest index, LB is the smallest index of the array.)

ONE DIMENSIONAL ARRAY



Example: in this above figure LB = 1993, UB = 2000 then :

Size of the array :

UB - LB + 1

=2000 - 1993 + 1

=7 + 1

=8

Therefore length or size of an array is 8 .

In this, **int** specifies the type of the variable. In this example, an integer ,and

R specifies the name of the variable , number of brackets []denotes the size of an array.

Arrays are also known as vectors and tables in mathematics.

Syntax of 1-d array: data_type array_name [no. of elements].

MEMORY REPRESENTATION OF ONE DIMENSIONAL ARRAY

- Let LA is a linear array in the memory of the computer.
- Memory of computer is simply a sequence of addressed locations therefore, the elements of LA are stored in the successive memory cells.

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

- Accordingly, the computer needs to keep track only of the address of the first element of LA called the base address of LA (denoted by Base (LA)).
- Using base address the computer calculates the address of any element of LA by the following formula:

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

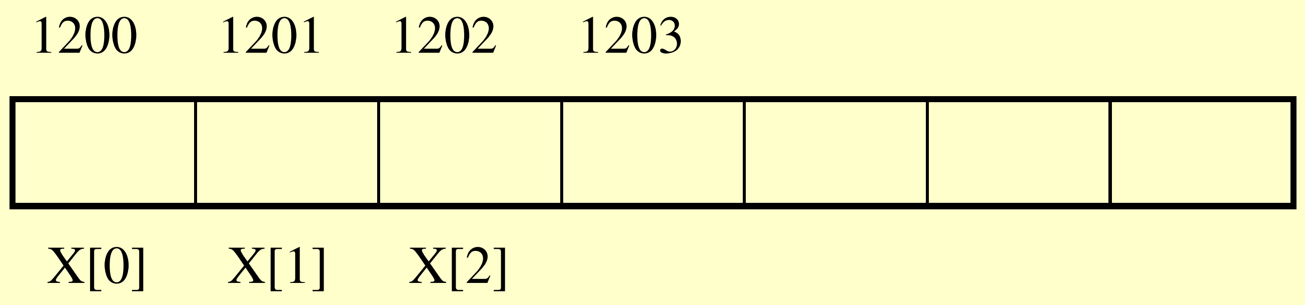
Where w is the number of words per memory cell for the array LA.

MEMORY REPRESENTATION OF ONE DIMENSIONAL ARRAY

Example: char X[100];

- Let *char* uses 1 location storage.
- If the base address is 1200 then the next element is in 1201. Index Function is written as:

Loc (X[i]) = Loc(X[0]) + i , *i* is subscript and LB = 0



MEMORY REPRESENTATION OF ONE DIMENSIONAL ARRAY

- In general, index function:

$$Loc(X[i]) = Loc(X[LB]) + w*(i-LB);$$

where w is length of memory location required.
For real number: 4 byte, integer: 2 byte and character: 1 byte.

- If LB = 5, Loc(X[LB]) = 1200, and w = 4, find Loc(X[8]) ?

$$\begin{aligned} Loc(X[8]) &= Loc(X[5]) + 4*(8 - 5) \\ &= 1212 \end{aligned}$$

ONE DIMENSIONAL ARRAY

200	300	250	302	400	202	500	100
1995	1996	1997	1998	1999	2000	2001	2002

Example : **lower bound(lb)=1995**
Upper bound(ub)=2002 and as integer occupies 2 bytes of memory therefore
W= 2 and the first element is at memory location 1000 therefore **base(s)=1500**.

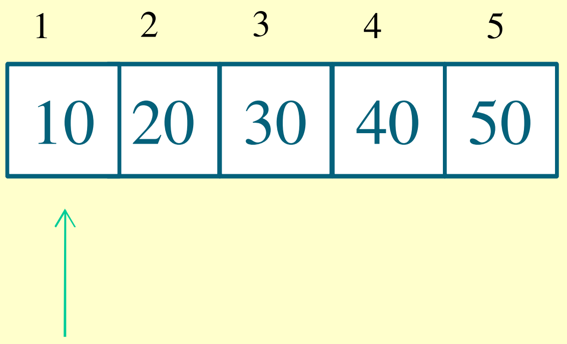
$$
\begin{aligned}
Loc(S_k) &= base(s) + w\,(k - lb) \\
Loc(S_{1998}) &= 1500 + 2\,(1998-1995) \\
Loc(S_{1998}) &= 1500 + 2\,(3) \\
&= 1506
\end{aligned}
$$

OPERATIONS PERFORMED ON ARRAYS

- ***TRAVERSAL*** : Processing each element in the list.
- ***INSERTION*** : Adding a new element to the list.
- ***DELETION*** : Removing an element from the list.
- ***SEARCHING***: Finding a location of an element in the given value.
- ***SORTING*** : Arranging the elements in proper order.
- ***MERGING*** : Combining two list into a single list.

Traversing

- Definition: traversing accesses each data item exactly once.
- **Example** : suppose there is a linear array a as:



Traversing of an Array

- In traversing ,we start from beginning and visit till last element.
- In this we access value of each element exactly once like :

a[1] =10 ,a[2] =20 ,a[3] =30 ,a[4] =40 ,a[5] =50 .

Traversal

- Traversing operation means visiting every node element once. e.g. to print, etc.

- <u>Example algorithm:</u>

```
1.  [Assign counter]
   K=LB   // LB = 0
2. Repeat step 2.1 and 2.2 while K <= UB   // If LB = 0
    2.1   [visit element]
     do PROCESS on LA[K]
          2.2        [add counter]
           K=K+1
3. end repeat step 2
4. exit
```

Insertion

<u>Definition</u>: adds a new data item in given collection of data items.

Example: consider 5 names in list namely Amit, Aditi, Anil, Pawan and Naveen.

Amit	Aditi	Anil	Pawan	Naveen			
1	2	3	4	5	6	7	8

Array ‘S’ with 5 element

If we want to insert a new element in the array at 4th position with its value Ravi.

So Pawan, Naveen would be shifted by one position like :

Amit	Aditi	Anil	Pawan **Ravi**	Naveen			
1	2	3	4	5	6	7	8

Array ‘S’ with new element inserted at 4th Position

Insertion

- Insert item at the back is easy if there is a space. Insert item in the middle requires the movement of all elements to the right as in figure shown below.

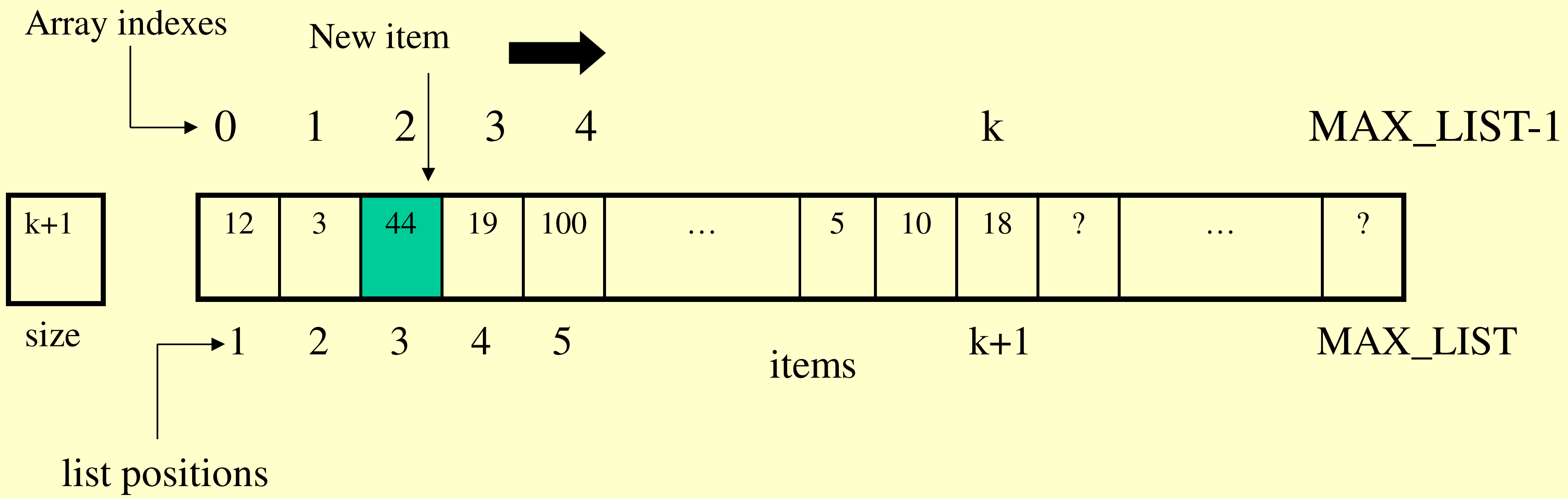


Figure 3: Shifting items for insertion at position 3

Insertion Algorithm

- Example algorithm:

```
INSERT(LA, N, K, ITEM)
//LA is a linear array with N element
//K is integer positive where K < N and LB = 0
//Insert an element, ITEM in index K
    1. [Assign counter]
       J = N – 1;    // LB = 0
    2. Repeat step 2.1 and 2.2 while J >= K
       2.1 [shift to the right all elements from J]
            LA[J+1] = LA[J]
       2.2 [decrement counter]   J = J – 1
    3. [Stop repeat step 2]
    4. [Insert element]   LA[K] = ITEM
    5. [Reset N]   N = N + 1
    6. Exit
```

Deletion

Definition : Removing an existing data item from the given collection of data items.

Example : Consider 6 names in list namely Amit, Aditi, Anil , Ravi, Pawan and Naveen .

Amit	Aditi	Anil	Ravi	Pawan	Naveen		
1	2	3	4	5	6	7	8

Array ‘A’ with 6 elements

If we want to delete Anil from list which is at location 3,then Ravi , Pawan , Naveen would be shifted one step forward that is s

Amit	Aditi	Anil	Ravi	Pawan	Naveen		
1	2	3	4	5	6	7	8

Array ‘S’ with 5 elements after deleting element from 3rd position

Deletion

- Delete item

(a)

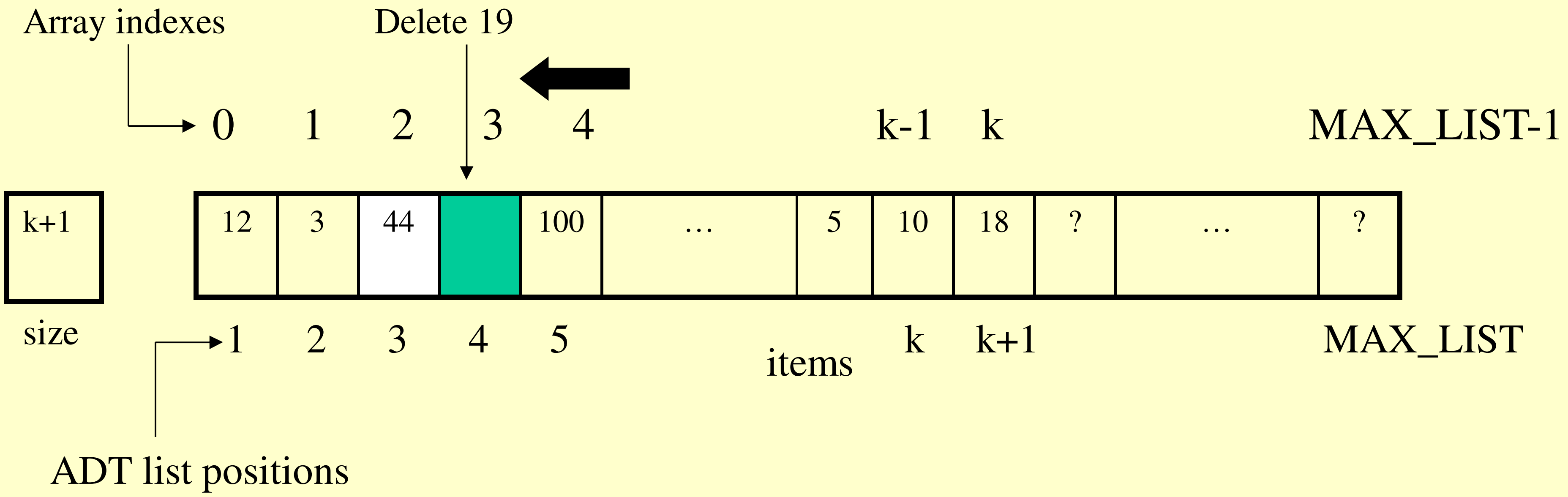


Figure 4: Deletion causes a gap

Deletion

(b)

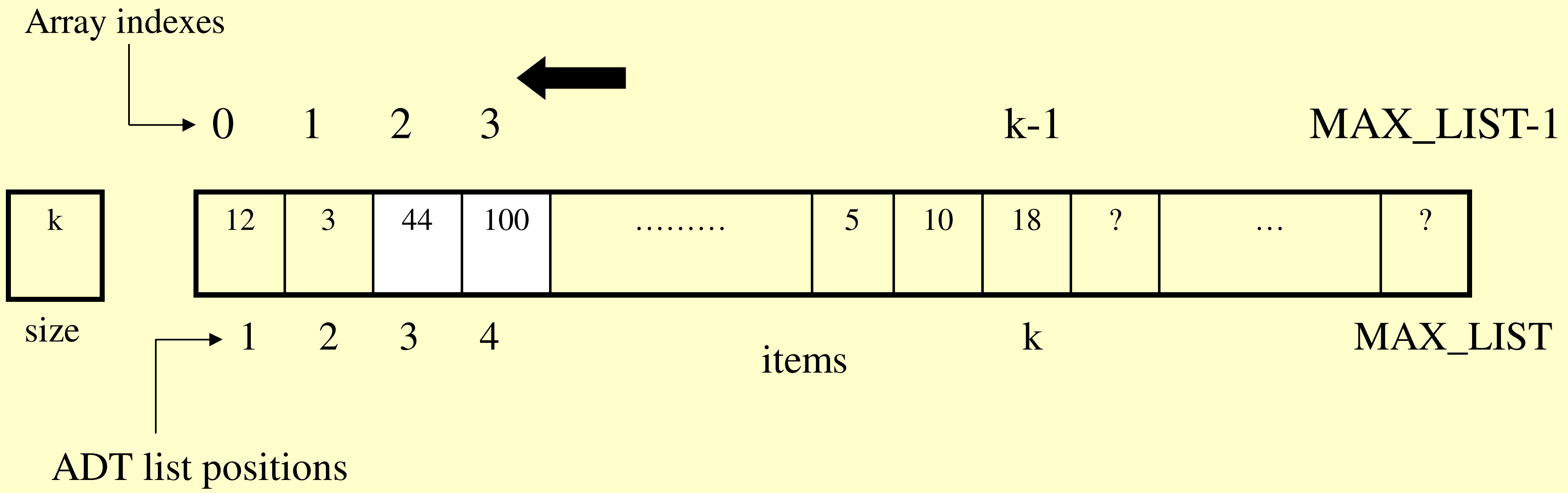


Figure 5: Fill gap by shifting

Deletion Algorithm

Example algorithm:

```
DELETE(LA, N, K, ITEM)
  1.  ITEM = LA[K]
  2.  Repeat for I = K to N–2     // If LB = 0
      2.1      [Shift element ke J + 1, forward]
               LA[I] = LA[I+1]
  3.  [end of loop]
  4.  [Reset N in LA]
      N = N – 1
  5.  Exit
```

Searching

- Searching is the process of determining whether or not a given value exists in a data structure or a storage media.
- We will study two searching algorithms
  - Linear Search
  - Binary Search

Linear Search

- The linear (or sequential) search algorithm on an array is:

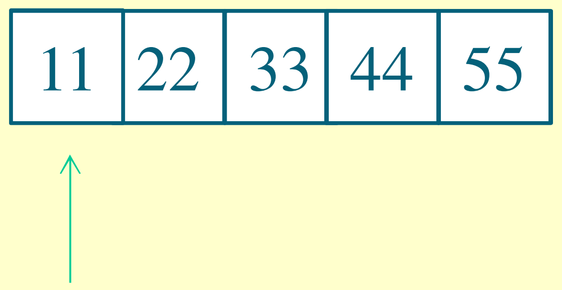
  Start from beginning of an array/list and continues until the item is found or the entire array/list has been searched.
- Sequentially scan the array, comparing each array item with the searched value.
- Linear search algorithm has complexity of O(n).

  (Note: linear search can be applied to both sorted and unsorted arrays.)

Linear Search

- The elements of the array need not in sorted order.
- It can be applied on any linear data structures even if elements of data structures don't occupy the contiguous memory locations.
- Start from beginning and compare with each element and continues until element is found or searched is made.
- The **<u>complexity</u>** of linear search is **Big O(n).**

Example : consider a linear array a as



If(Data==Item)
(data==66)

Array 'A' with 5 elements

Linear Search Algorithm

Searching the position of given element “Data” in an array ‘S’ having ‘n’ elements.

```
1.  Repeat steps 2 and 3 for i=1 to n
2. If S[i] = Data    then
              Print “ Element is found at position “: I
              Exit
      [End If]
3.  Set i= i+1
    [End loop]
4. Print: “ Desired element Data is found in the array”
5.  Exit
```

Binary Search

- Binary search algorithm is efficient if the array is sorted.
- A binary search is used whenever the list starts to become large.
- The binary search starts by testing the data in the element at the middle of the array to determine if the target is in the first or second half of the list.
- If it is in the first half, we do not need to check the second half. If it is in the second half, we do not need to test the first half. In other words we eliminate half the list from further consideration. We repeat this process until we find the target or determine that it is not in the list.
- To find the middle of the list, we need three variables, one to identify the beginning of the list, one to identify the middle of the list, and one to identify the end of the list.

Binary Search

- Assume we want to find 22 in a sorted list as follows:

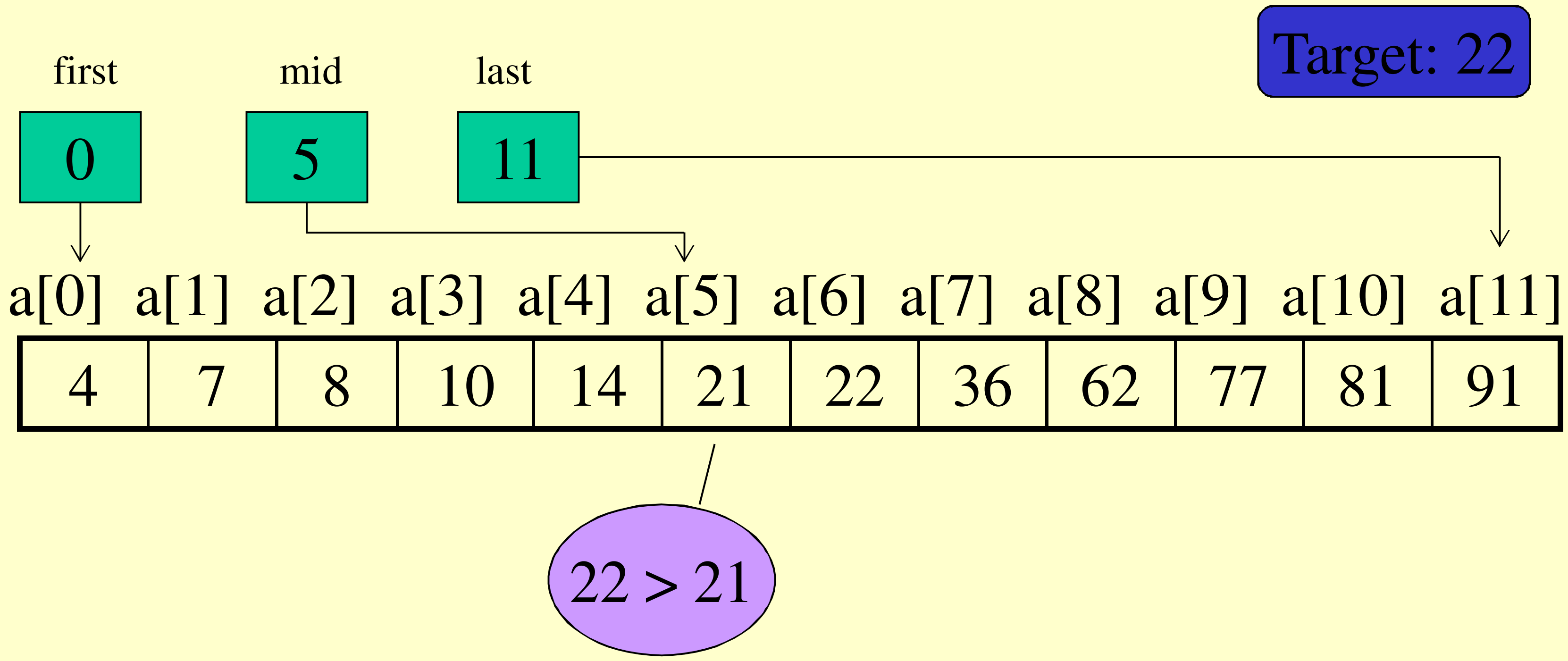
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	7	8	10	14	21	22	36	62	77	81	91

- The three indexes are first, mid and last. Given first as 0 and last as 11, mid is calculated as follows:

$$\text{mid} = (\text{first} + \text{last}) / 2$$
$$\text{mid} = (0 + 11) / 2 = 11 / 2 = 5$$

Binary Search

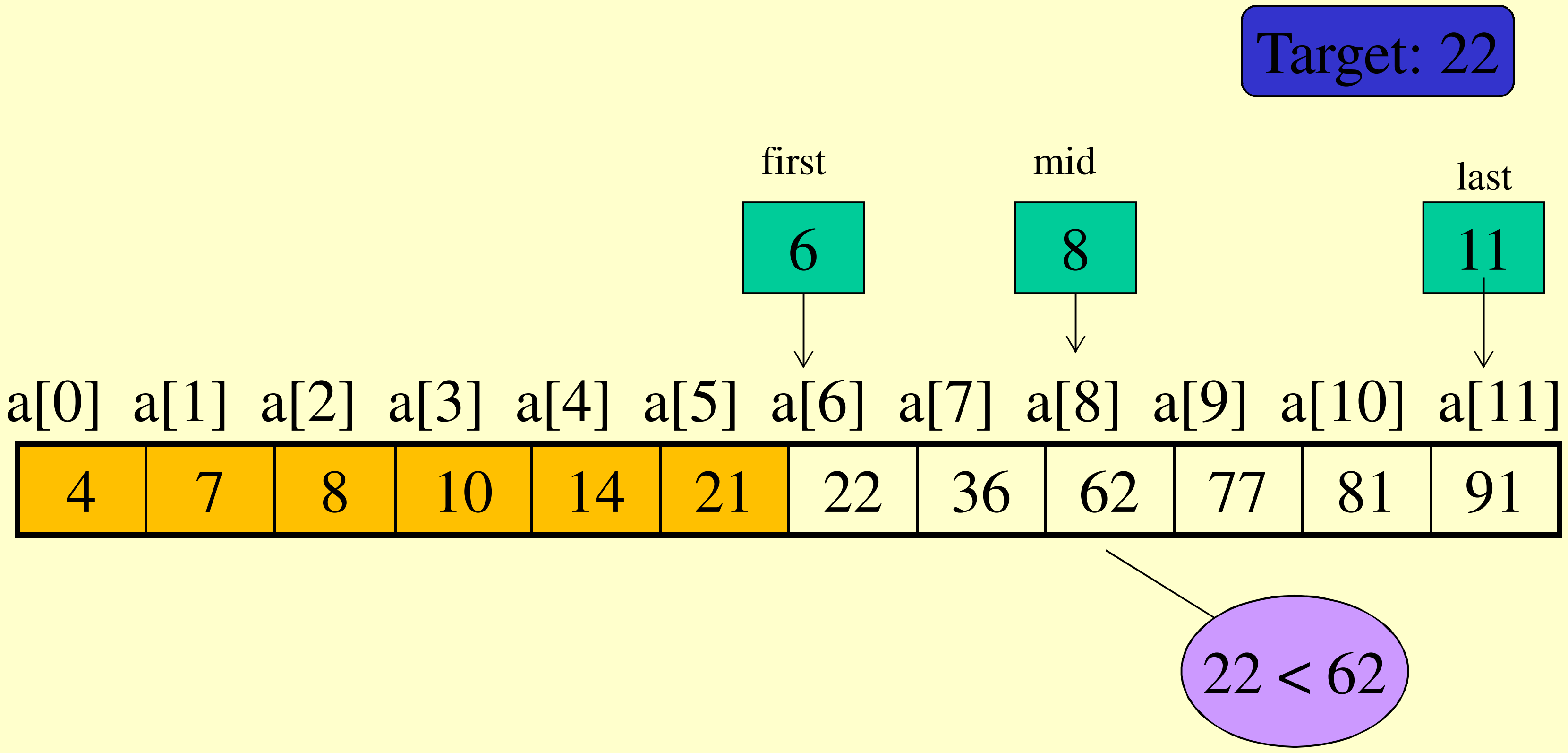
- At index location 5, the target is greater than the list value (22 > 21). Therefore, eliminate the array locations 0 through 5 (mid is automatically eliminated). To narrow our search, we assign mid + 1 to first and repeat the search.



Binary Search

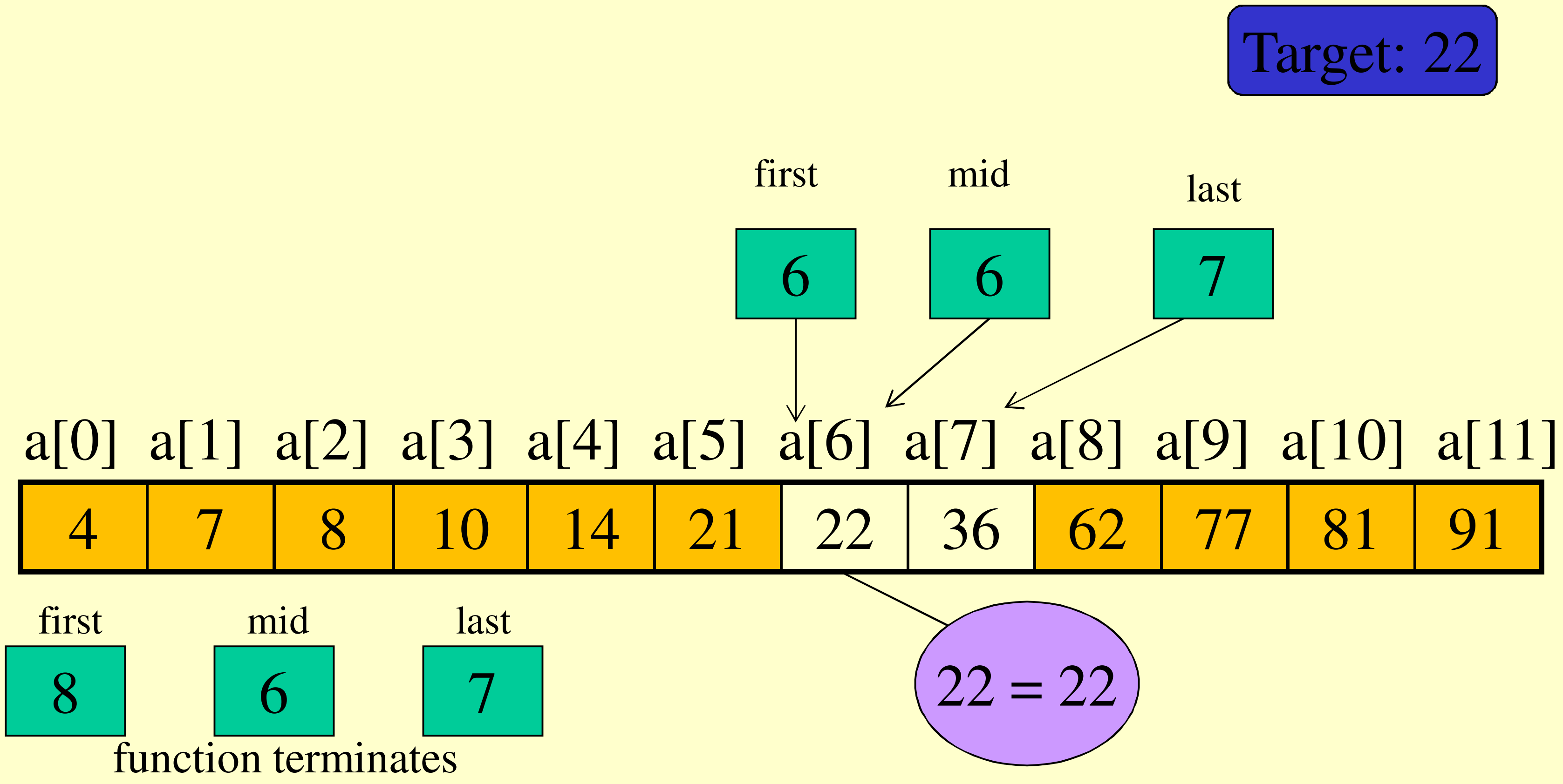
- The next loop calculates mid with the new value for first and determines that the midpoint is now 8 as follows:

$$mid = (6 + 11) / 2 = 17 / 2 = 8$$



Binary Search

- When we test the target to the value at mid a second time, we discover that the target is less than the list value (22 < 62). This time we adjust the end of the list by setting last to mid – 1 and recalculate mid. This step effectively eliminates elements 8 through 11 from consideration. We have now arrived at index location 6, whose value matches our target. This stops the search.



Binary Search Algorithm

```
BINARY(DATA,LB,UB,ITEM,LOC)
1.[Initialize segment variables.]
Set BEG = LB,END =UB and MID = INT(BEG + END)/2).
2.Repeat Steps 3 and 4 while $BEG \leq END$ and $DATA[MID] \neq ITEM$.
3.If ITEM< DATA[MID],then:
Set END = MID -1    else
Set BEG =MID +1.(end of if structure)
```

Binary Search Algorithm

```
4. Set MID =INT((BEG + END)/2).
(End of step 2 loop.)
5.If DATA[MID]=ITEM,then:
Set LOC= MID.
Else:
Set LOC= NULL.
(end of if structure)
6.Exit.
```

Complexity Analysis of Binary Search

The best case for binary search occurs when the element being searched for is exactly at the middle of the sorted array. In this case only comparison is required to find that desired element giving a best case runtime of **Big O(1)**.

Worst case for the binary search occurs when the element is not found in the array. since binary search halves the sorted array in each step until there are no values that can be halved. The efficiency of binary search in this case can be expressed as logarithmic function. For calculating the worst complexity, the number of elements in the array as power of two (i.e. $n=2^x$).

After 1st comparison, number of elements remains$=n/2^1 =n/2$

After 2nd comparison, number of elements remains$=n/2^2 =n/2$

After 3rd comparison, number of elements remains$=n/2^3 =n/2$

:

:

After xth comparison, number of elements remains$=n/2^{x=}1$

Total number of maximum comparisons **$= x =\log_2 n$**

Multi- Dimensional Arrays

- The above table tells the score of five students in the class. First column specifies the roll number of students and second columns specifies the marks of particular student whose roll number is mentioned in the row.
- The elements of two dimensional array a can be referenced using any of the notations shown below:

A_{ij} OR A[i][j] OR A(i,j)

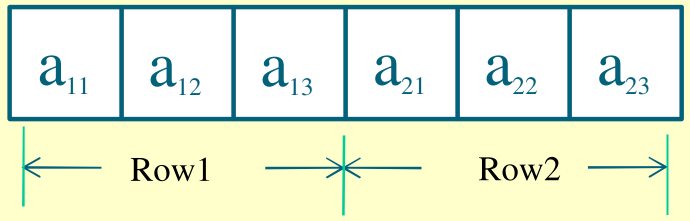
Consider the example of two dimensional array A of order 2*3. Here array A has 2 rows and 3 columns. Total numbers of elements in arrays A is 2*3 i.E 6

$$A = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ 1 & a_{11} & a_{12} & a_{13} \\ 2 & a_{21} & a_{22} & a_{23} \end{array}$$

A Two Dimensional array with 2 rows and 3 columns

Memory Representation Of Two Dimensional Array

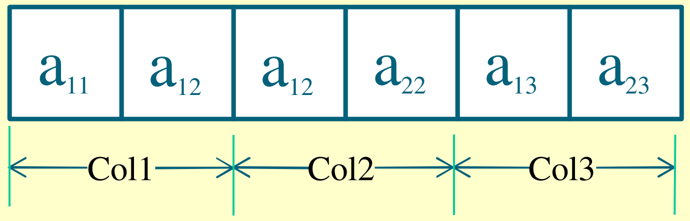
- Consider a two dimensional array of order r*c. While representing a two dimensional array on paper ,we write its elements in rows and columns. But when such an array is stored into the computer memory ,its r*c elements will occupy r*c consecutive memory locations.
- The choice between the two ways for storing the dimensional array into the computer memory depends on the programming language. The representation of two dimensional array using both the way. Row major order and column major are shown below for an array a of order 2*3.



Row major order representation of two dimension array ‘A’

Memory Representation of Two Dimensional Array

As shown in the figure above the elements of the first row of the array are stored first and after that elements of the second row are stored.



Column major order representation of two dimension array ‘A’

As shown in the above figure for column major order ,the element of the first column of the array are stored first and after that the element of the second column are stored and so on.

General Multidimensional Array

- An array can be of three dimensional, four dimensional ,..N-dimensional. The concept of referencing an element of n-dimensional array can be extended from one dimensional or two dimensional arrays.
- Generally speaking ,total number of elements of elements in a n dimensional array can be calculated by multiplying the length of each dimension. Here to reference any element of the n-dimensional array, n subscript will be required. An n dimensional array **B** can be declared as:

  B[lb_1: ub_1 , lb_2 : ub_2 , lb_3 : ub_3 , ……., lb_n : ub_n]

- Here $\mathbf{lb_i}$ is the lower index and $\mathbf{ub_i}$ is the upper index of the **i**th dimension.
- Length of **i**th dimension can be calculated as:
- Length of **i**th dimension $(l_i)=ub_i - lb_i + 1$ where $1 \leq i \leq n$
- Each element is referenced by using n subscript $k_1,k_2,k_3,\ldots\ldots.K_n$ each with the property as $lb_1 \leq k_1 \leq ub_1,\ldots..Lb_2 \leq k_2 \leq ub_2,\ldots..Lb_n \leq k_n \leq ub_n$

General Multidimensional Array

- The of storage for multidimensional array is same as for two dimensional array i.e. Either in row major order or column major order.
- Consider a three dimensional array t(2:5,1:3,4:6) which is a collection of 36(4*3*3) elements.

$$\begin{tikzcd}
\boxed{t_{2.1.4}} \arrow[r] & \boxed{t_{2.1.5}} \arrow[r] & \boxed{t_{2.1.6}} \arrow[r] & \boxed{t_{2.2,4}} \arrow[r] & \boxed{t_{2.2.5}} \arrow[r] & \boxed{t_{2.2.6}} \arrow[r] & \boxed{t_{2.3.4}} \arrow[r] & \boxed{t_{2.3.5}} \arrow[r] & \boxed{t_{2.3.6}} \arrow[dllllllll] \\
\boxed{t_{3.1.4}} \arrow[r] & \boxed{t_{3.1.5}} \arrow[r] & \boxed{t_{3.1.6}} \arrow[r] & \boxed{t_{3.2.4}} \arrow[r] & \boxed{t_{3.2.5}} \arrow[r] & \boxed{t_{3.2.6}} \arrow[r] & \boxed{t_{3.3.4}} \arrow[r] & \boxed{t_{3.3.5}} \arrow[r] & \boxed{t_{3.3.6}} \arrow[dllllllll] \\
\boxed{t_{4.1.4}} \arrow[r] & \boxed{t_{4.1.5}} \arrow[r] & \boxed{T_{4,1.6}} \arrow[r] & \boxed{t_{4.2,4}} \arrow[r] & \boxed{t_{4.2.5}} \arrow[r] & \boxed{T_{4,2.6}} \arrow[r] & \boxed{t_{4.3.4}} \arrow[r] & \boxed{T_{4,3,5}} \arrow[r] & \boxed{t_{4.3.6}} \arrow[dllllllll] \\
\boxed{t_{5.1.4}} \arrow[r] & \boxed{t_{5.1.5}} \arrow[r] & \boxed{t_{5.1.6}} \arrow[r] & \boxed{t_{5.2,.4}} \arrow[r] & \boxed{t_{5.2,.5}} \arrow[r] & \boxed{t_{5.2,6}} \arrow[r] & \boxed{t_{5.3.4}} \arrow[r] & \boxed{t_{5.3.5}} \arrow[r] & \boxed{t_{5.3,6}}
\end{tikzcd}$$

Sequence of elements of a three dimensional array t in row major order

General Multidimensional Array

Consider a three dimensional array A having LB and UB as lower index and upper index of first dimension ,lb and ub as lower index and upper index of second dimension ,lb and ub as lower index and upper index of third dimension .The length of each dimension (l,l,l respectively)

$L_1 = ub_1 - lb_1 + 1$	Length of the First Dimension
$L_2 = ub_2 - lb_2 + 1$	Length of the Second Dimension
$L_3 = ub_3 - lb_3 + 1$	Length of the Third Dimension

We can calculate the address of any element A[i,j,k] (such that $lb_1 \leq i \leq ub_1$, $lb_2 \leq j \leq ub_2$, $lb_3 \leq k \leq ub_3$) using the following formula:

For Row Major Order

$$\text{Address of}(A[i][j][k]) = \text{Base}(A) + w[l_2\ l_3 (i - lb_1) + l_3 (j - lb_2) + (k - lb_3)]$$

For Column Major Order

$$\text{Address of}(a[i][j][k]) = \text{Base}(A) + w[(i - lb_1) + l_1 (j - lb_2) + l_1\ l_2 (k - lb_3)]$$

Sparse Arrays

- As mentioned earlier ,matrices are two dimensional arrays in which elements are arranged into rows and columns. A matrix of order r*c is collection of r*c elements which are arranged in r rows and c columns .
- Each element of a two dimensional array is referenced by using two subscript ,1st subscript to represent row index and 2nd subscript to represent column index.

Sparse Matrix

- A matrix M said to be sparse matrix if majority of its elements are meaningless. Such kind of matrices contains high density of meaningless elements or we can say that the sparse matrix has a very few elements which are significant.
- If we consider zero as meaningless elements then the sparse matrix which has majority of zero elements .The below shown array 6*5 is a sparse matrix because maximum of its elements are zero.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 7 & 0 & 0 & 5 \end{bmatrix}$$

Diagonal Matrix

- A matrix M is said to b diagonal matrix if and only if M[I,j] =0 for i!= j. that is all the non –diagonal elements are zero.

$$\begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

A Diagonal matrix

Upper Triangular Matrix

- A matrix m is called upper triangular matrix if and only if [i,j]=0 for i>j.
- That is, all the elements of the matrix below the diagonal elements are zero .
- Matrix is said to be strictly upper triangular if all the diagonal elements are also
  zero along with the elements below the diagonal

$$\begin{bmatrix} 5 & 8 & 7 & 9 \\ 0 & 3 & 2 & 16 \\ 0 & 0 & 5 & 7 \\ 0 & 0 & 0 & 8 \end{bmatrix}$$

An Upper Triangular matrix

$$\begin{bmatrix} 0 & 8 & 7 & 9 \\ 0 & 0 & 2 & 16 \\ 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

A Strictly Upper Triangular matrix

Lower Triangular Matrix

$$\begin{bmatrix} 5 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 8 & 9 & 5 & 0 \\ 3 & 2 & 7 & 8 \end{bmatrix}$$

A lower Triangular matrix

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 8 & 9 & 0 & 0 \\ 3 & 2 & 7 & 0 \end{bmatrix}$$

A Strictly Lower Triangular matrix

- A matrix M is called lower triangular matrix if and only if [i,j]=0 for i<j.
- That is, all the elements of the matrix above the diagonal elements are zero .
- Matrix is said to be strictly lower triangular if all the diagonal elements are also zero

  along with the elements above the diagonal.

Memory Representation Of Special Kind Of Matrices

- Generally to store an ordinary n*n matrix into memory ,we need n^2 memory locations. But in case of special matrices which are discussed above the memory requirements can be reduced up to some extent.
- Some of the techniques for storing these special matrix are discussed in the next slide.

Method of linearization

- In this technique the element of matrix are stored in a one dimensional array.

  Only the non-zero element of the matrix are stored and zero entries are discarded.

- Let us store a diagonal matrix M of order n*n in a one dimensional array A. There will be at most n non-zero elements at the diagonal position of matrix . The element $m_{1,1}$ will be stored as element $a_{1,}$ element $m_{2,2}$ will be stored as element a_2 and so on up to the element $m_{n,n}$ which will be stored as element a_n.

$$\begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 9 \end{bmatrix}$$

$$\begin{array}{|c|c|c|c|} \hline 5 & 7 & 8 & 9 \\ \hline \multicolumn{1}{c}{a_1} & \multicolumn{1}{c}{a_2} & \multicolumn{1}{c}{a_3} & \multicolumn{1}{c}{a_4} \end{array}$$

A matrix ‘M’ of order 4*4 and array representation of the matrix

Method Of Vector Representation

$$\begin{bmatrix} 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 5 & 0 & 0 & 3 \\ 0 & 0 & 0 & 8 & 0 \end{bmatrix}$$

- In this technique ,the non-zero elements of the sparse matrix are stored in the array along with its row id and column id. The non zero elements are stored in the row major order by discarding all the zero entries.
- For example ,using the method of vector representation, a 4*65 sparse array which is shown below will be store by vector V as:

Method Of Vector Representation

V[1]	**1**	**1**	**6**
V[2]	2	3	2
V[3]	3	2	5
V[4]	3	5	3
V[5]	4	4	8

Vector representation of non-zero elements of a sparse matrix

However in vector representation ,we are storing row id and column id along with the element itself, but this approach avoids storage of the zero entries.

Advantages Of Array

- Array is the simple kind of data structure which is very easy to implementation.
- Address of any element of the array can be calculated very easily as elements are stored in contiguous memory locations.
- Array can be used to implement other data structure such as stack ,queue, trees and graph.
- If elements of an array are stored in some logical order then binary search can be applied to search an element in the array efficiently.

Limitations Of Array

- Array is the static kind of data structure. Memory used by array cannot be increased or decreased whether it is allocated at run time or compile time.
- Insertion and deletion of elements are very time consuming in array.
- Only homogeneous elements can be stored in the array. Therefore , in case we want to store the data of mixed type, the array cannot be used.

Sorting

- Sorting is the process of rearranging your data elements/Item in ascending or descending order

Unsorted Data

4	3	2	7	1	6	5	8	9

Sorted Data (Ascending)

1	2	3	4	5	6	7	8	9

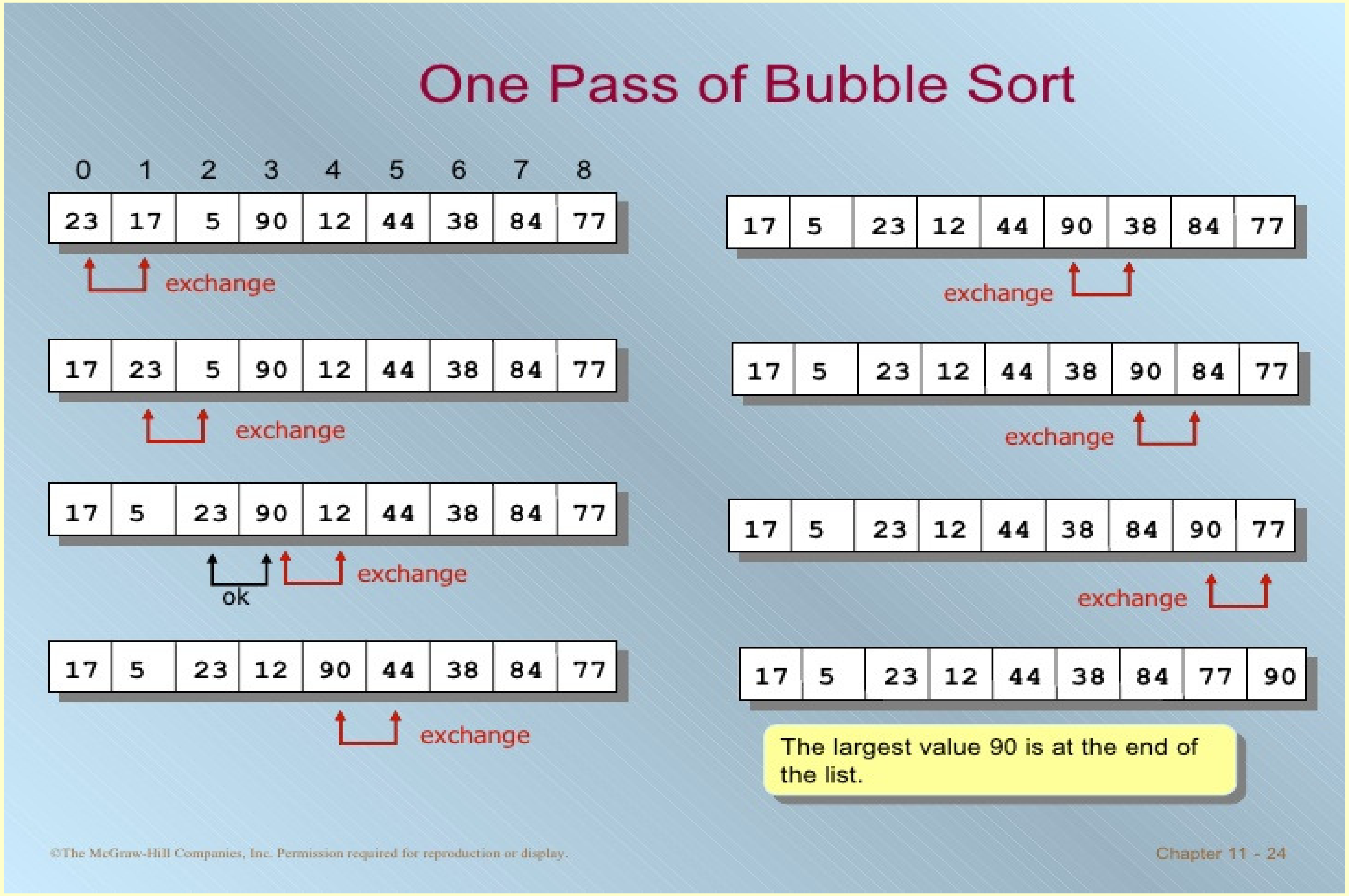
Sorted Data (Descending)

9	8	7	6	5	4	3	2	1

Bubble sort

- With the selection sort, we make one exchange at the end of one pass.
- The bubble sort improves the performance by making more than one exchange during its pass.
- By making multiple exchanges, we will be able to move more elements toward their correct positions using the same number of comparisons as the selection sort makes.
- The key idea of the bubble sort is to make pairwise comparisons and exchange the positions of the pair if they are out of order.

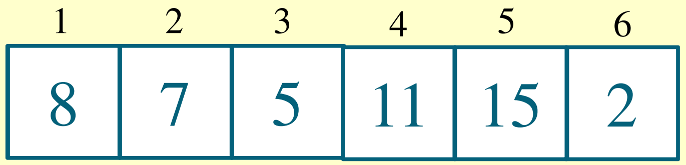
Bubble Sort



One Pass of Bubble Sort
0 1 2 3 4 5 6 7 8
23 17 5 90 12 44 38 84 77
exchange
17 23 5 90 12 44 38 84 77
exchange
17 5 23 90 12 44 38 84 77
ok
exchange
17 5 23 12 90 44 38 84 77
exchange
17 5 23 12 44 90 38 84 77
exchange
17 5 23 12 44 38 90 84 77
exchange
17 5 23 12 44 38 84 90 77
exchange
17 5 23 12 44 38 84 77 90
The largest value 90 is at the end of the list.

Bubble Sort

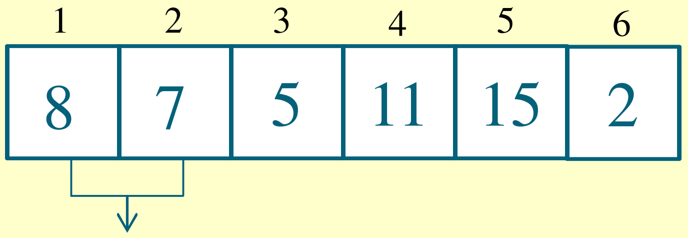
Example : We have an unsorted array list ‘S’ having 6 elements as shown below:



An Unsorted Array ‘S’ with 6 elements.

Various Passes takes place to sort the list .

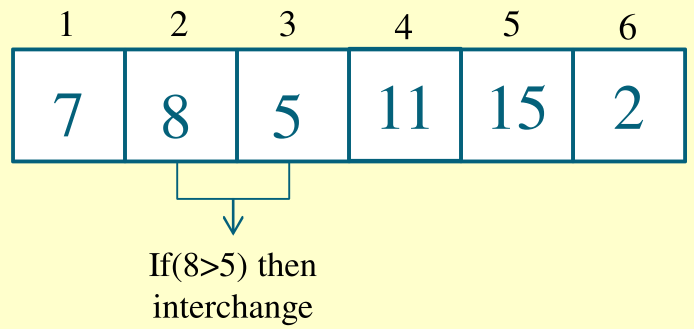
Pass 1 : Number of steps= **n-1 =6 -1 = 5.**



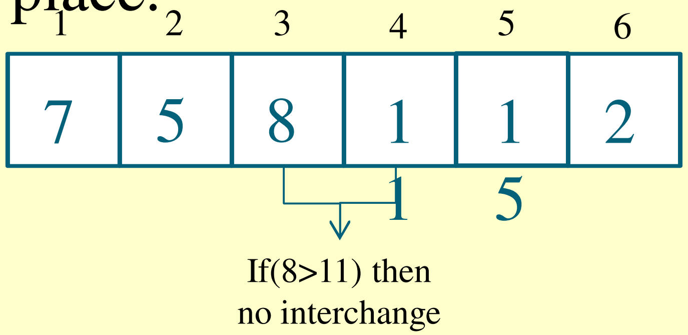
If(8>7) then interchange

S[1] is compared with S[2] ; as S[1]>S[2] then exchange takes place.

Bubble Sort



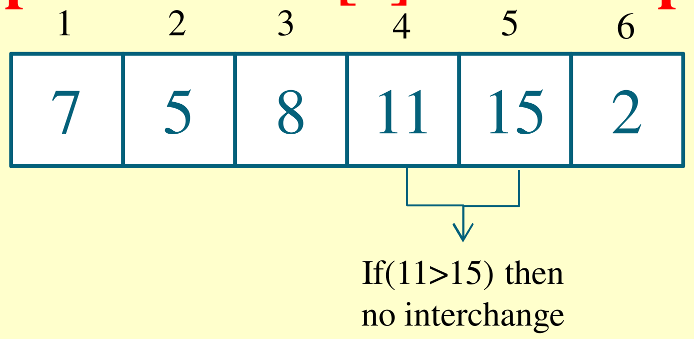
S[2] is compared with S[3];as S[2]>S[3],then exchange takes place.



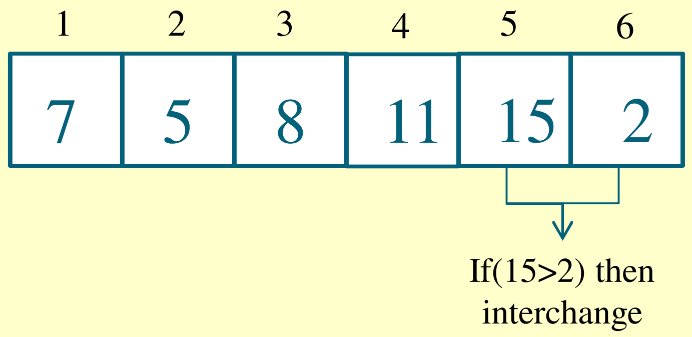
S[3] is compared with S[4];as S [3]< S[4],then no exchange takes place.

Bubble Sort

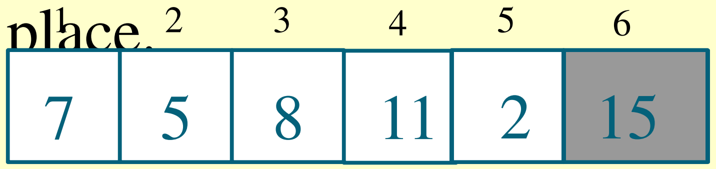
The 1st largest element 15 has obtained it’s proper positioned S[6] in 5 comparison



S[4] is compared with S[5];as S[4]< S[5],then no exchange takes place.

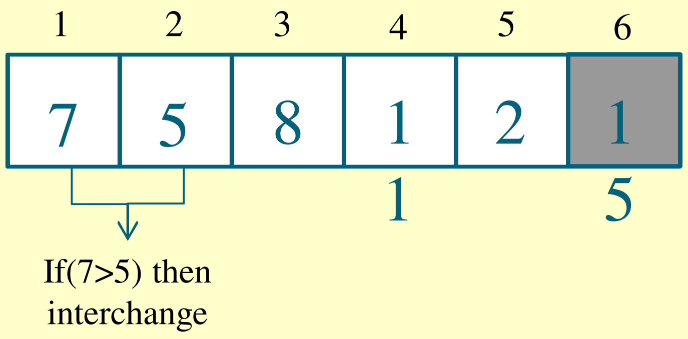


S[5] is compared with S[6];as S[5]> S[6],then exchange takes place.

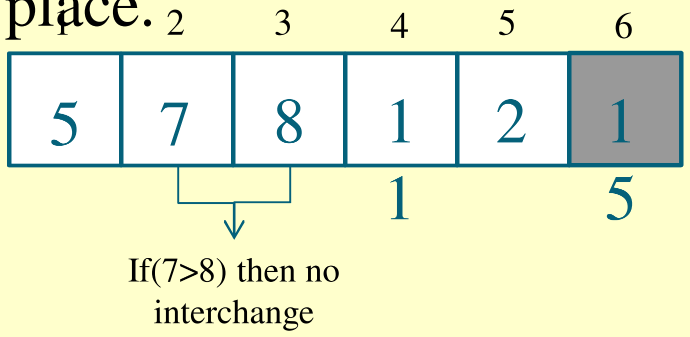


Bubble Sort

<u>Pass 2</u>: n-2 = 6-2 = 4

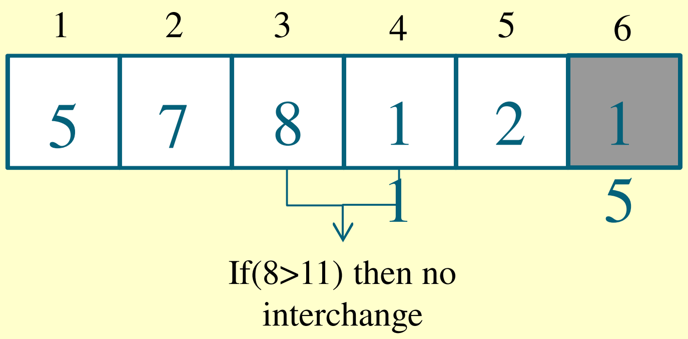


S[1] is compared with S[2];as S[1]> S[2],then exchange takes place.

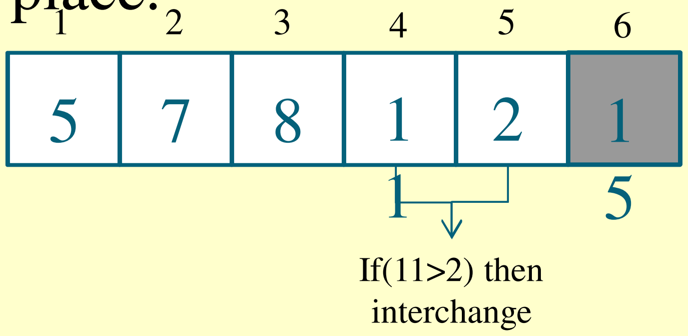


S[2] is compared with S[3];as S[2]<S[3],then no exchange takes place.

Bubble Sort

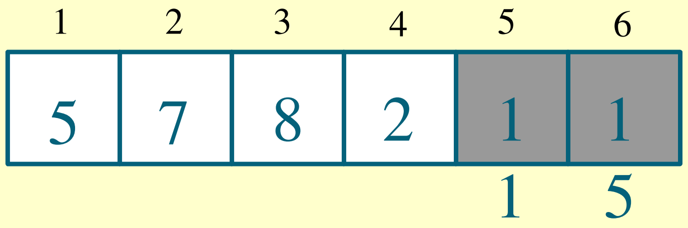


S[3] is compared with S[4];as S[3]<S[4],then no exchange takes place.



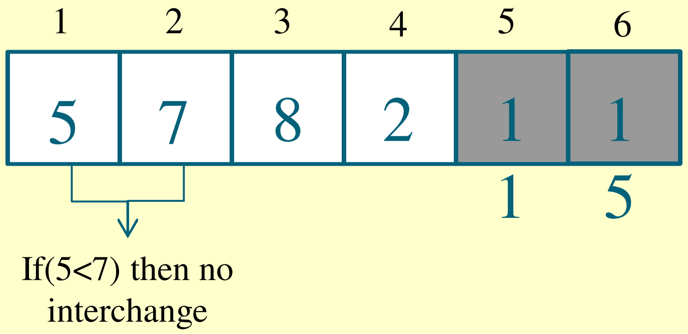
S[4] is compared with S[5];as S[4]>S[5],then exchange takes place.

Bubble Sort



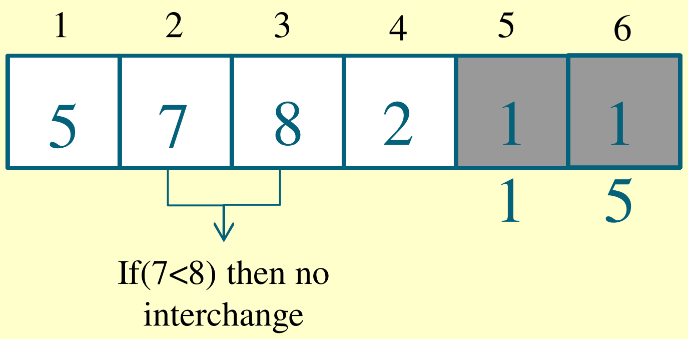
The 2nd largest element 11 has obtained it's proper positioned S[5] in 4 comparison

<u>Pass 3</u>: n-3 = 6-3 = 3

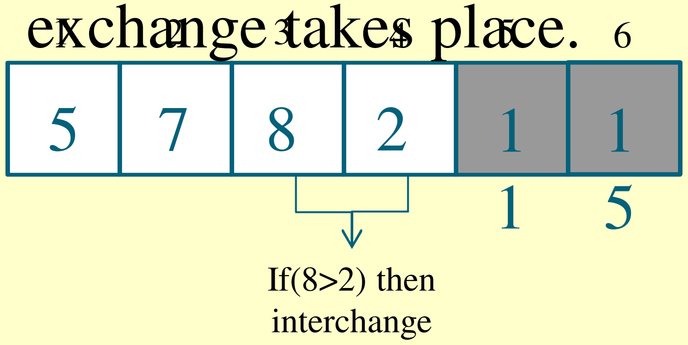


S[1] is compared with S[2];as S[1]<S[2],then no exchange takes place.

Bubble Sort

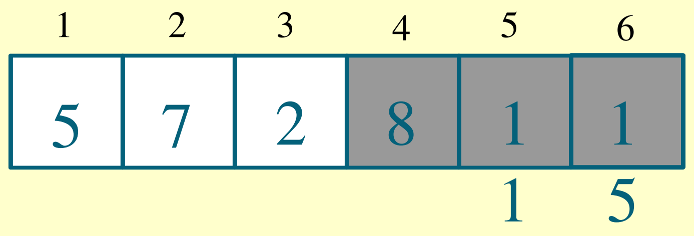


S[2] is compared with S[3];as S[2]<S[3],then no exchange takes place.



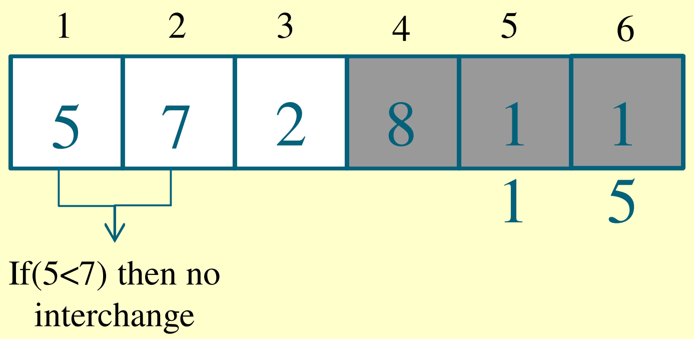
S[3] is compared with S[4];as S[3]>S[4],then exchange takes place.

Bubble Sort



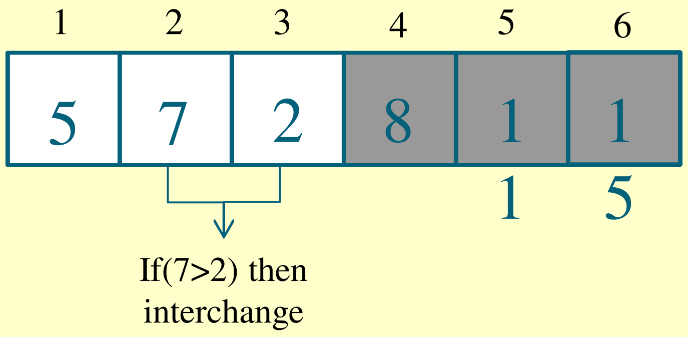
The 3rd largest element 8 has obtained its proper position S[4] in 3 comparisons.

<u>Pass 4</u>: n-4 = 6-4 = 2

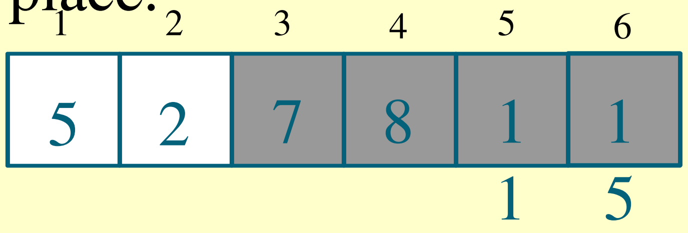


S[1] is compared with S[2];as S[1]<S[2],then no exchange takes place.

Bubble Sort



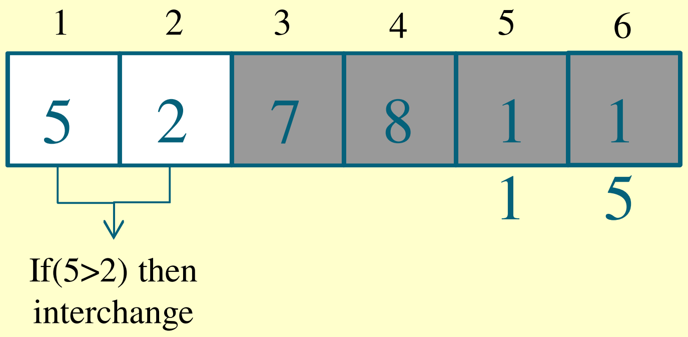
S[2] is compared with S[3];as S[2]>S[3],then exchange takes place.



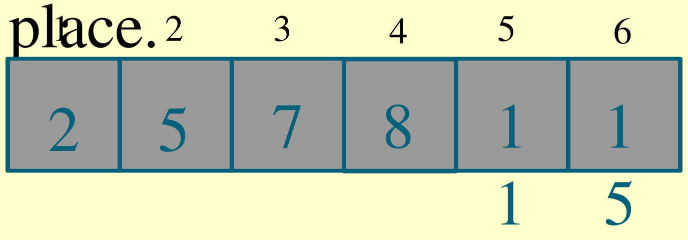
The 4th largest element 7 has obtained its proper position S[3] in 2 comparisons.

Bubble Sort

<u>Pass 5</u>: n-5 = 6-5 = 1



S[1] is compared with S[2];as S[1]>S[2],then exchange takes place.



The S[1] and S[2] have obtained its proper position in just a single comparison.

Bubble sort Algorithm

Sorting an array ‘S’ of size ‘n’ in increasing order using the bubble sort technique.

```
1.  Repeat for p = 1 to n-1
2.  For i= 1 to n-p
3.  If S[i] > S[i+1]    then
                    Exchange S[i] with S[i+1]
                    [End If]
                    [End Loop]
                    [End Loop]
4.   Exit
```

Complexity of Bubble Sort Algorithm

The complexity of any sorting algorithm is analyzed through the number of Comparisons required during the sorting procedure. In this algorithm we can easily Determine the total number of comparisons. As we have already observed that an Array of size **n** gets sorted after **n-1** passes. In the bubble sort n-1 comparisons take Place during the **1st** pass which places the largest element of the array on the last Position, **n-2** comparisons take place in the **2nd** pass which places the second largest Element of the array on the second last position, **kth** pass requires **n-k** comparisons Which places kth largest element at **(n-k+1)th** position of the array and the last pass Requires only one comparison. Total number of comparison will be:

$$
\begin{aligned}
F(n) &= (n-1)+(n-2)+(n-3)+\ldots\ldots\ldots\ldots+(n-k)+2+1 \\
&= (n-1)+(n-2)+\ldots\ldots\ldots\ldots+2+1 \\
&= ((n-1)*n)/2
\end{aligned}
$$

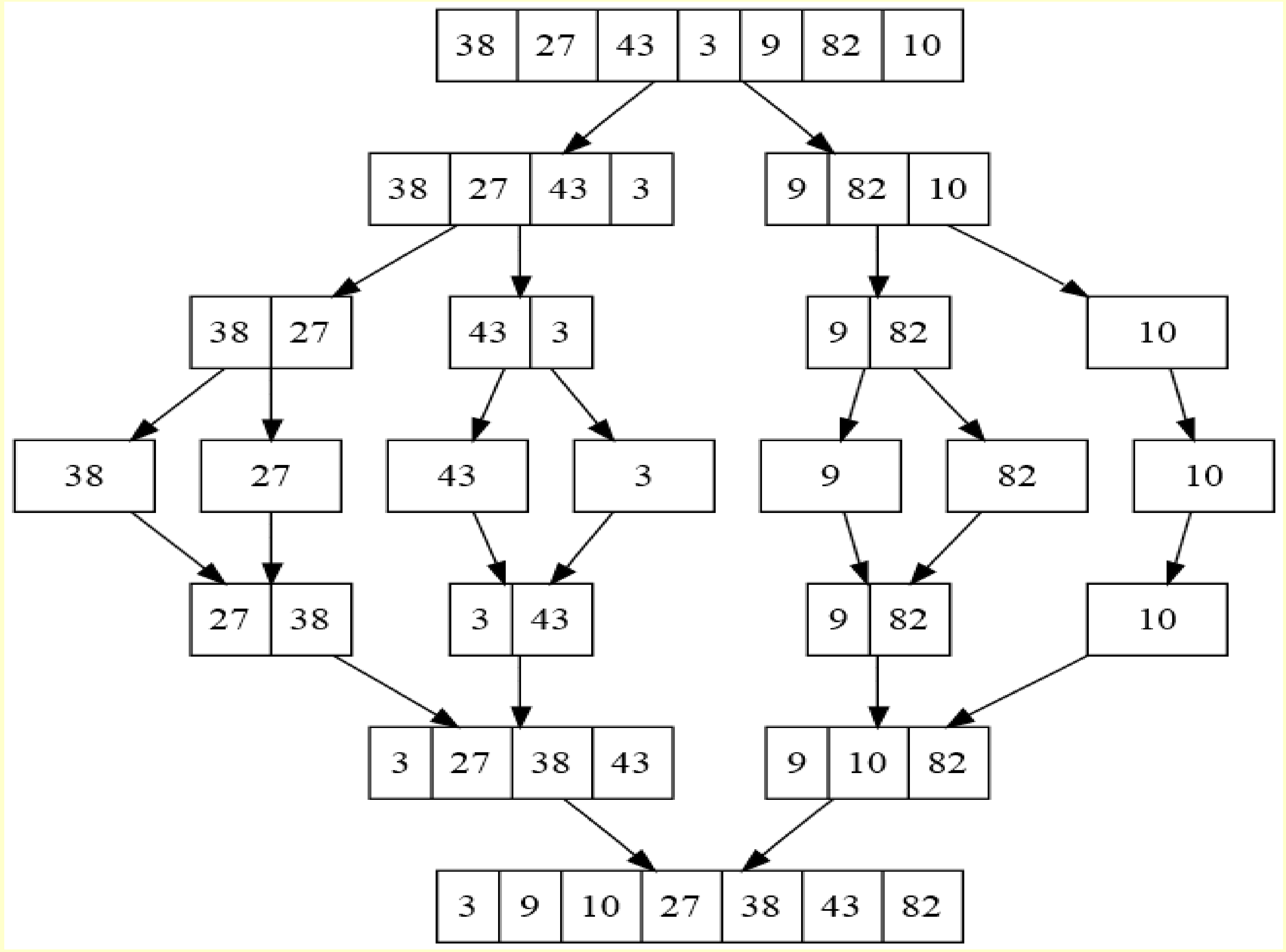
The complexity of bubble sort algorithm will be **Big O(n^2)**

Merging

- Merging of arrays refers to combining the elements of two linear arrays into a single array.
- The following is the input and output of the MERGE procedure :
- **INPUT**: Array A and indices p, q, r such that $p \leq q \leq$ r and subarray $A[p \mathinner{..} q]$ is sorted and subarray $A[q + 1 \mathinner{..} r]$ is sorted. By restrictions on p, q, r, neither subarray is empty.
- **OUTPUT**: The two subarrays are merged into a single sorted subarray in $A[p \mathinner{..} r]$.

We implement it so that it takes $\Theta(n)$ time,
where $n = r - p + 1$, which is the number of elements being merged.

Merging


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merging Algorithm

```
MERGE ($A$, $p$, $q$, $r$ )
    1.  $n_1 \leftarrow q - p + 1$
     $n_2 \leftarrow r - q$
     Create arrays L[1 . . $n_1$ + 1] and R[1 . . $n_2$ + 1]
    2.  **FOR** $i \leftarrow 1$ **TO** $n_1$
          **DO**    $L[i] \leftarrow A[p + i - 1]$
       **FOR** $j \leftarrow 1$ **TO** $n_2$
          **DO**    $R[j] \leftarrow A[q + j]$
    3. $L[n_1 + 1] \leftarrow \infty$
       $R[n_2 + 1] \leftarrow \infty$
       $i \leftarrow 1$
       $j \leftarrow 1$
```

Merging Algorithm

```
4. FOR $k \leftarrow p$ TO $r$
       DO IF L[$i$ ] $\leq$ R[ $j$]
              THEN A[$k$] $\leftarrow$ L[$i$]
                     $i \leftarrow i + 1$
              ELSE A[k] $\leftarrow$ R[j]
                     $j \leftarrow j + 1$
5. Exit
```

Merging

- Definition: merging of array refers to combining the elements of two linear arrays into single array.
- When elements of the arrays are not sorted and merged array the it needn't be in sorted order.
- If the elements in given arrays are sorted and we want to merge them into third array which is also required to be in sorted order.

Case 1: an unsorted array 'A1' with 7 elements

1	2	3	4	5	6	7
8	7	90	75	2	32	25

Unsorted array 'A2' with 6 elements

1	2	3	4	5	6
15	45	21	9	99	3

After merging A1 & A2 ; merged array 'M' is :

8	7	90	75	2	32	25	15	45	21	9	99	3
1	2	3	4	5	6	7	8	9	10	11	12	13

Merging

Case 2: sorted array A1 with 3 elements.

2	7	8
1	2	3

Sorted array A2 with 4 elements.

3	9	15	21
1	2	3	4

2	3	7	8	9	15	21

During merging a1[1]is compared with A2[1].As A1[1]<A2[1] so shift A1[1] i.e. 2 to first location of the 3rd array A3 at 1st position

Now a1[2]is compared with A2[1].As A2[1]<A1[2] so shift A2[1] will be placed at 2nd position in 3rd array.

Now a1[2]is compared with A2[2].As A1[2]<A2[2] So shift A1[2] will be placed at 3rd position in 3rd array.

Now a1[3]is compared with A2[2].As A1[3]<A2[2] so shift A1[3] will be placed at 4th position in 3rd array

Now the remaining elements are copied as such as they are already in sorted order.

Now all the elements in sorted order.

Merging

Algorithm: Merges two sorted arrays and (in ascending order) into a third sorted array (in ascending order). and are the lower bounds of given arrays and respectively and and are the upper bounds of the given arrays and respectively.

```
1. Set i =lb1 , j=lb2 , K =1.
2. While i≤ ub1 and j ≤ ub2.
3.   If A1[i]<A2[j] then
          Set A3[K]=A1[i]
          Set i=i+1.
          Set k=k+1.
```

Merging

```
Else
            Set A3[K]=A2[j]
            Set j=j+1
            Set k=k+1
        [End if]
    [End loop]
4. If i>ub1 then
        While j≤ub2
            Set A3[K] =A2[j]
            Set j=j+1
            Set k=k+1.
```

Multi- Dimensional Arrays

- The elements of one dimensional array are referenced using only single subscript. This is the reason that arrays are also popular with the name of single dimensional array.
- Multidimensional array is the array in which elements are referenced by using 2 or more subscript.
- The array whose elements are referenced by using 2 or more subscript. The array whose elements are referenced by using 3 subscript is known as three dimensional array

Two Dimensional Array

Consider an example of a class result as also shown below

101201	78
101202	86
101203	45
101204	34
101205	90