ARRAYS

- Linear arrays: Memory representation
- Traversal
- Insertion
- Deletion
- Linear Search
- Binary Search
- Merging
- 2D Array : Memory representation

CONTENTS

2.1 Introduction

- Data Structure can be classified as:
  - ✓ linear
  - ✓ non-linear
- Linear (elements arranged in sequential in memory location) i.e. array & linear link-list
- Non-linear such as a tree and graph.
- Operations:
  - ✓ Traversing, Searching, Inserting, Deleting, Sorting, Merging
- Array is used to store a fix size for data and a link-list the data can be varies in size.

2.1 Introduction

- Advantages of an Array:
  - Very simple
  - Economy – if full use of memory
  - Random accessed at the same time

- Disadvantage of an Array:
  - wasting memory if not fully used

2.2 Linear Array

- Homogeneous data:

  a) Elements are represented through indexes.

  b) Elements are saved in sequential in memory locations.

- Number of elements, N –> length or size of an array.

  If:

  UB : upper bound (the largest index)

  LB : lower bound (the smallest index)

  Then: N = UB – LB + 1

  Length = N = UB when LB = 1

2.2 Linear Array

- All elements in A are written symbolically as, 1 .. n is the subscript.

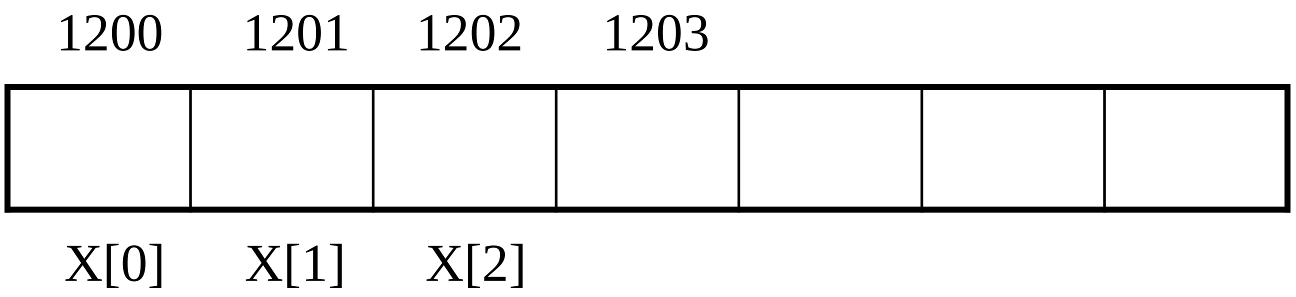
  A1, A2, A3, ... , An
- In FORTRAN and BASIC → A(1), A(2), ..., A(N)
- In Pascal, C/C++ and Java → A[0], A[1], ..., A[N-1]
- subscript starts from 0

  LB = 0, UB = N–1

2.2.1 Representation of Array in a Memory

- The process to determine the address in a memory:
  a) First address – base address.
  b) Relative address to base address through index function.

  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



2.2.1 Representation of Array in a Memory

- In general, index function:

  LOC (LA[K]) = BASE(LB) + w*(K-LB);

  where w is number of words per memory cell for the array.
  For real number: 4 byte, integer: 2 byte and character: 1 byte.

- <u>Example:</u>

  If LB = 5, BASE[LB]) = 1200, and w = 4, find Loc(LA[8]) ?
  Loc(X[8])= Loc(X[5]) + 4*(8 – 5)
  = 1212

2.2.2 Traversing Algorithm

- Traversing operation means visit every element once. e.g. to print, etc.
- Example algorithm:LA is a linear array with lower bound LB and Upper Bound UB

```
1.  [Assign counter]
   K=LB
2. Repeat step 3 and 4 while K <= UB
3. [visit element]
    do PROCESS on LA[K]
4.  [add counter]
        Set K=K+1
5. exit
```

2.2.3 Insertion Algorithm

- 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 1.

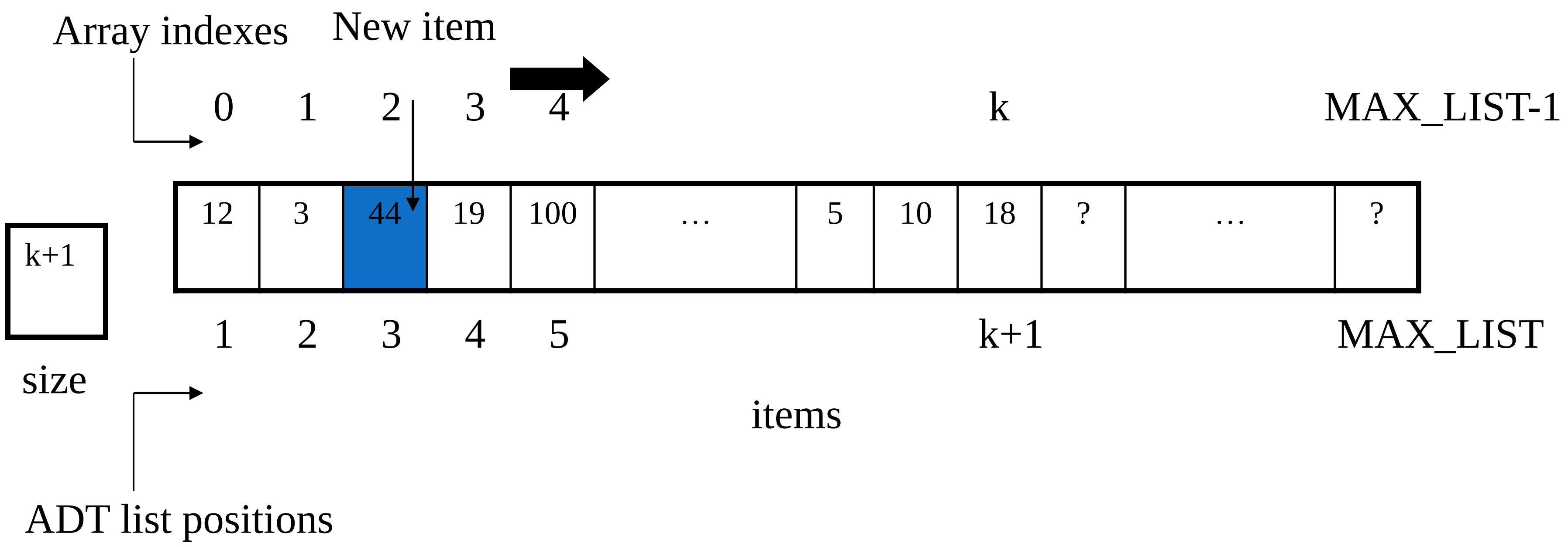


Figure 1: Shifting items for insertion at position 3

2.2.3 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 = 1
//Insert an element, ITEM in index K
    1. [Assign counter]
       J = N ;
    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
```

2.2.4 Deletion Algorithm

- Delete item.

(a)

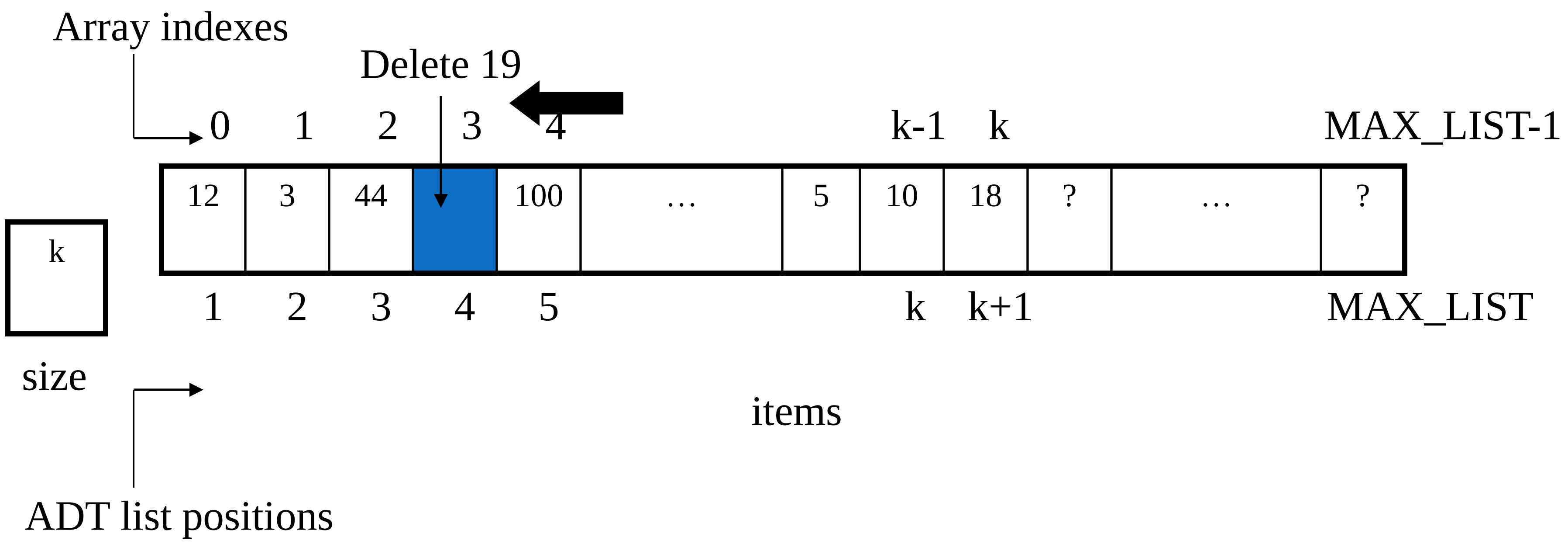


Figure 2: Deletion causes a gap

2.2.4 Deletion Algorithm

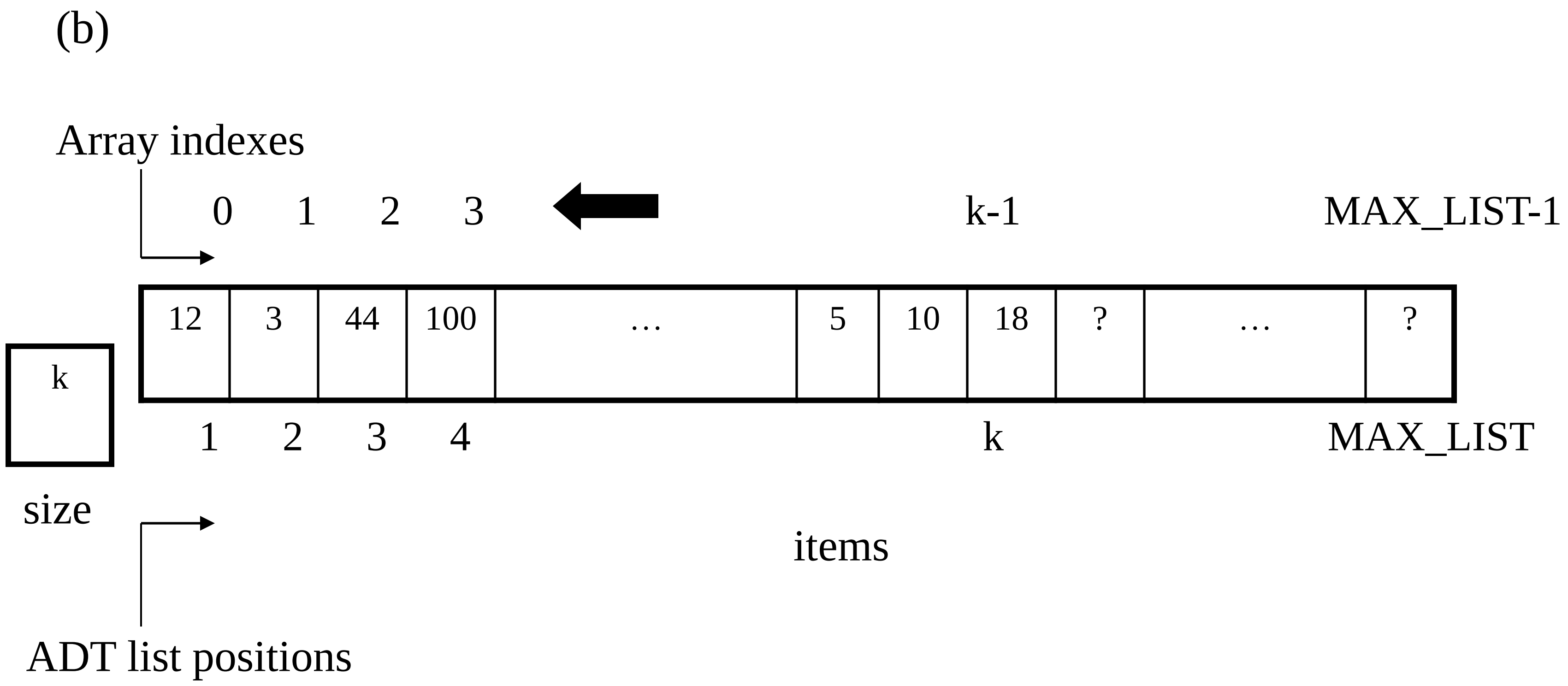


Figure 3: Fill gap by shifting

2.2.4 Deletion Algorithm

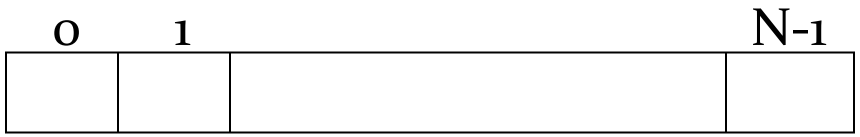
- Example algorithm:

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

2.2.5 Sequential Search

Compare successive elements of a given list with a search ITEM until

1. either a match is encountered
2. or the list is exhausted without a match.



Algorithm:

```
SequentialSearch(LA, N, ITEM, LOC)
  1. I = 0 // If LB = 0
  2. Repeat step 2.1 while (i<N and LA[I] != ITEM )
       2.1 I=I+1
  3. If  LA[I]==ITEM then
     Return found at LOC=I
     Else
     Return not found
  4. Exit
```

2.2.5 Binary Search Algorithm

- Binary search algorithm is efficient if the array is sorted.
- A binary search is used whenever the list starts to become large.
- Consider to use binary searches whenever the list contains more than 16 elements.
- 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.

2.2.5 Binary Search Algorithm

- 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.
- We analyze two cases here: the target is in the list (target found) and the target is not in the list (target not found).

2.2.5 Binary Search Algorithm

- **Target found case**: 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

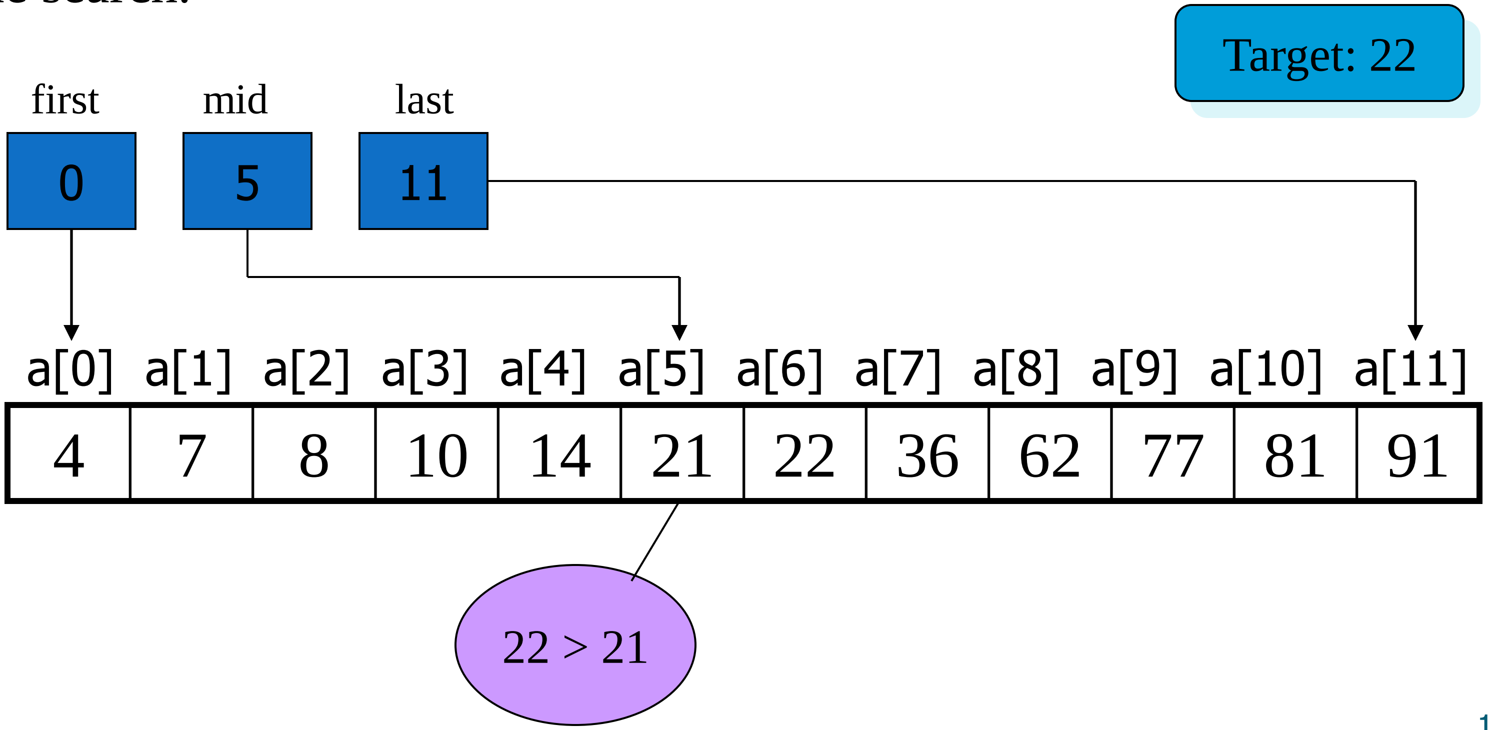
- The three indexes are first, mid and last. Given first as 0 and last as 11, mid is calculated as follows:

  mid = (first + last) / 2

  mid = (0 + 11) / 2 = 11 / 2 = 5

2.2.5 Binary Search Algorithm

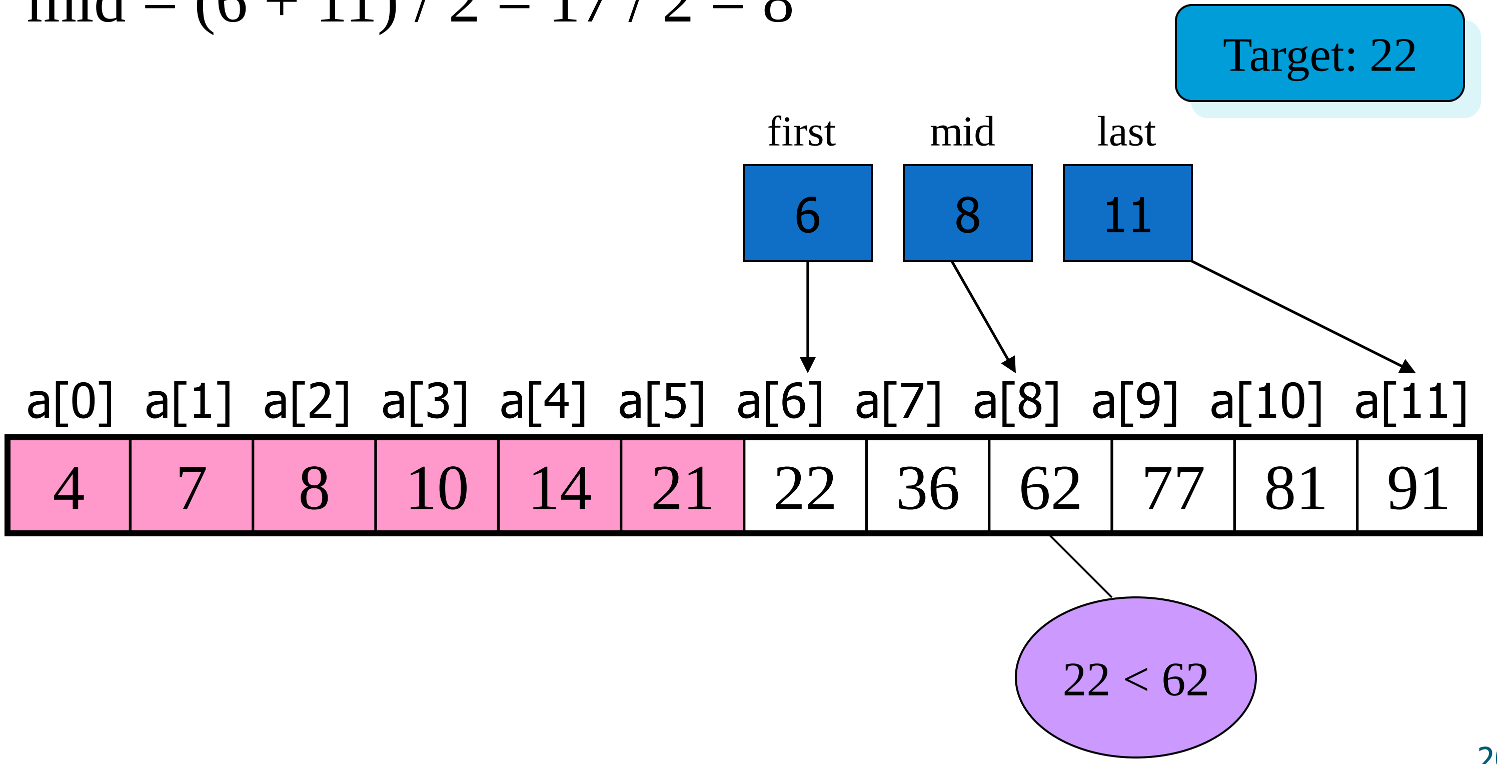
- At index location 5, the target is greater than the list value (22 > 21). Therefore, eliminate the array locations 0 through 5 (mid is automatically eliminated). To narrow our search, we assign mid + 1 to first and repeat the search.



2.2.5 Binary Search Algorithm

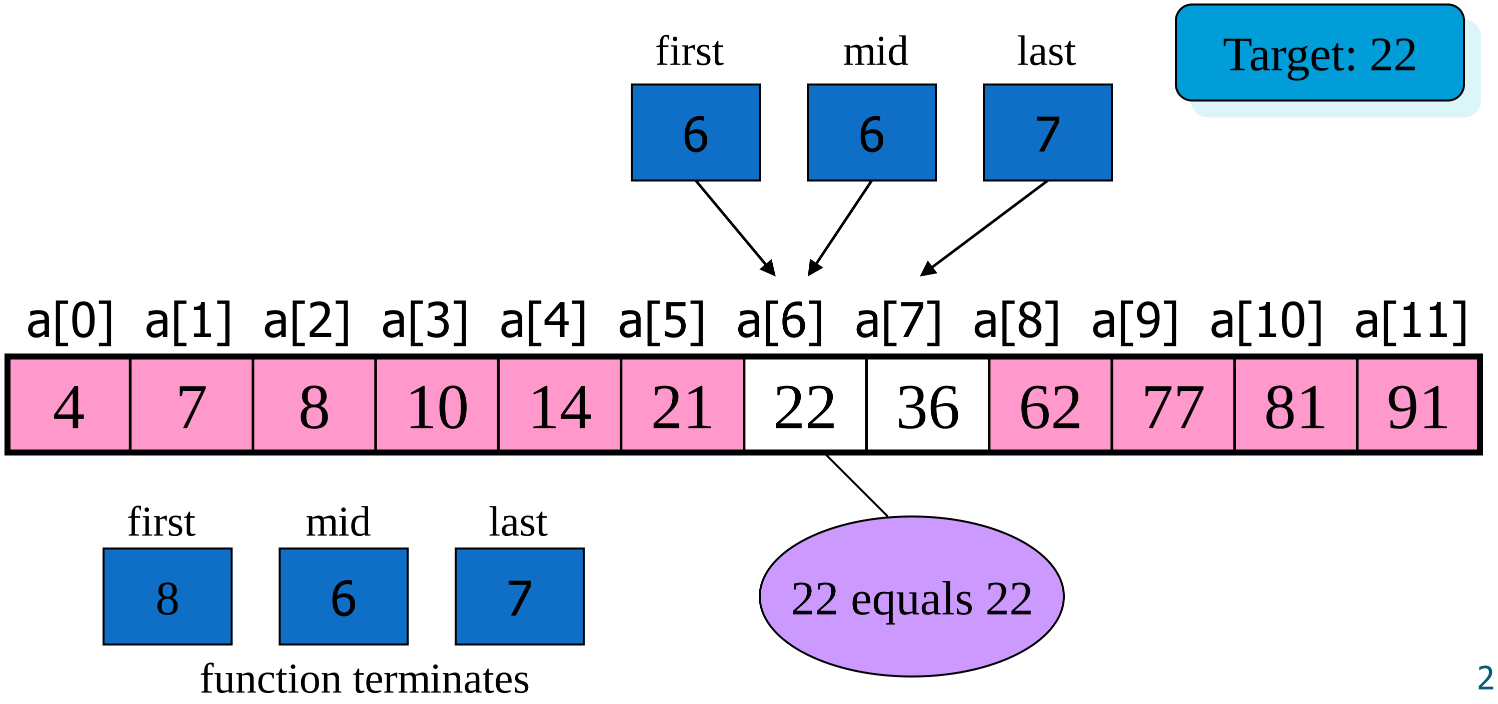
- The next loop calculates mid with the new value for first and determines that the midpoint is now 8 as follows:

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



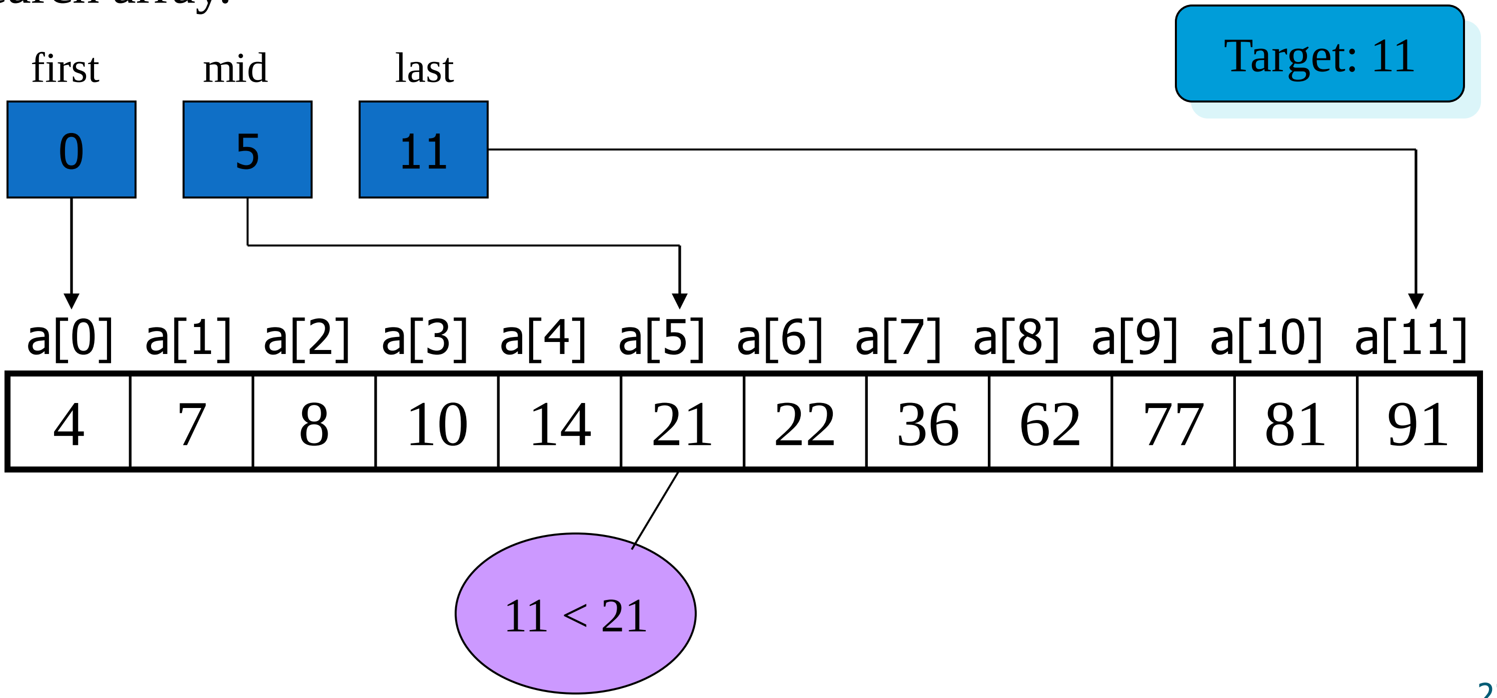
2.2.5 Binary Search Algorithm

- 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.



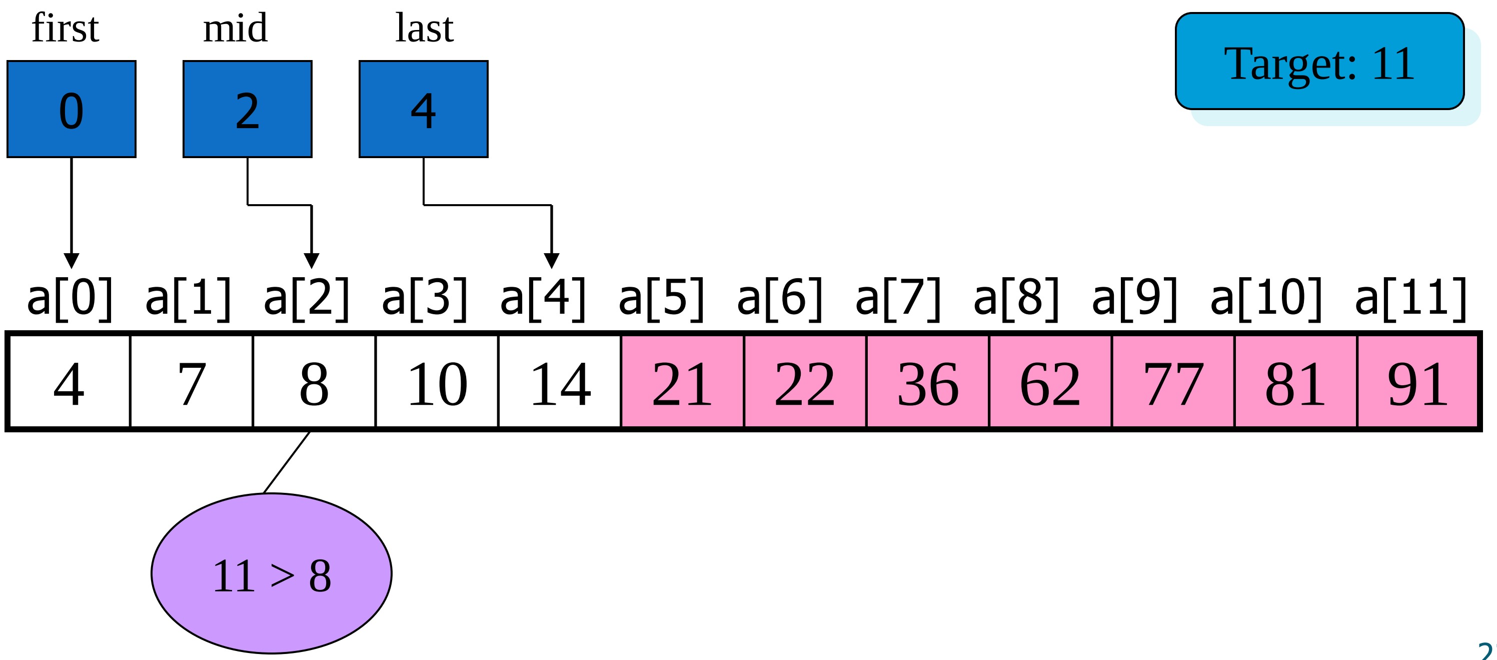
2.2.5 Binary Search Algorithm

- Target not found case: This is done by testing for first and last crossing: that is, we are done when first becomes greater than last. Two conditions terminate the binary search algorithm when (a) the target is found or (b) first becomes larger than last. Assume we want to find 11 in our binary search array.



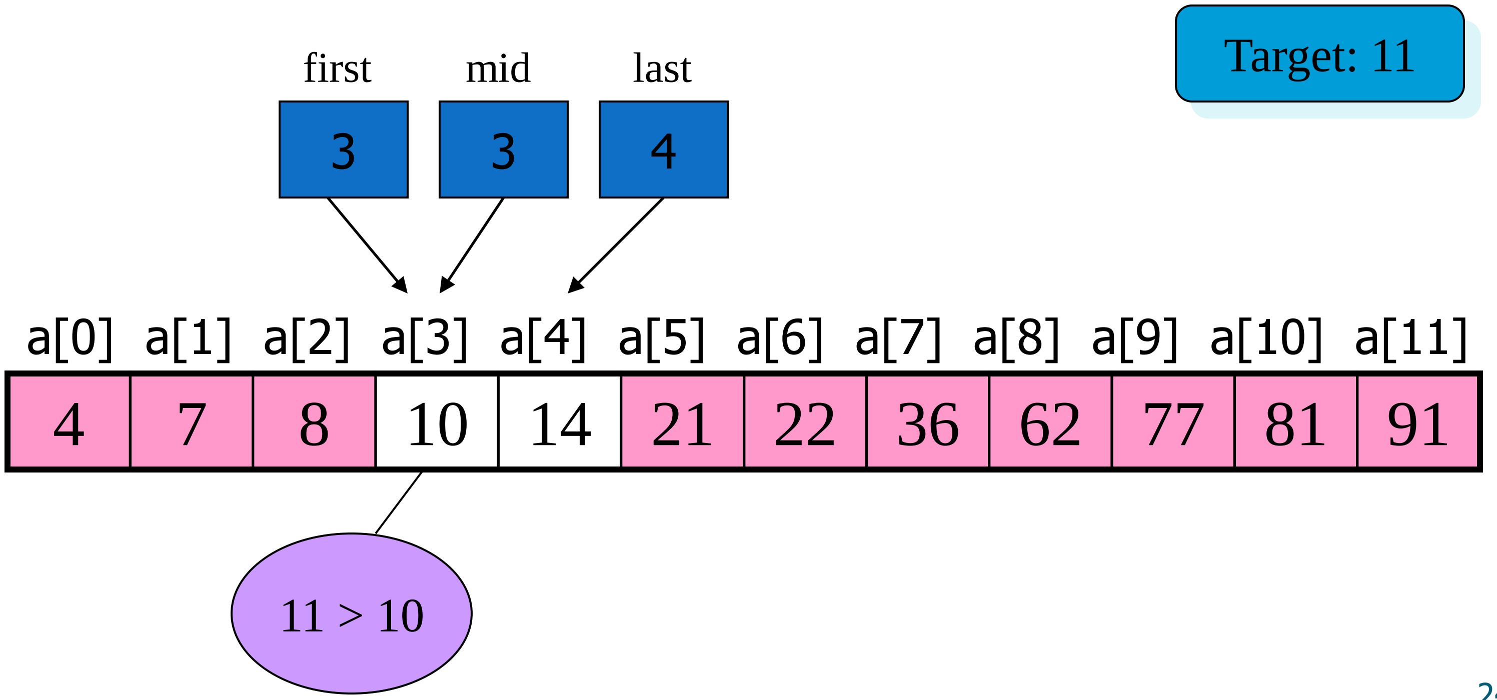
2.2.5 Binary Search Algorithm

- The loop continues to narrow the range as we saw in the successful search until we are examining the data at index locations 3 and 4.



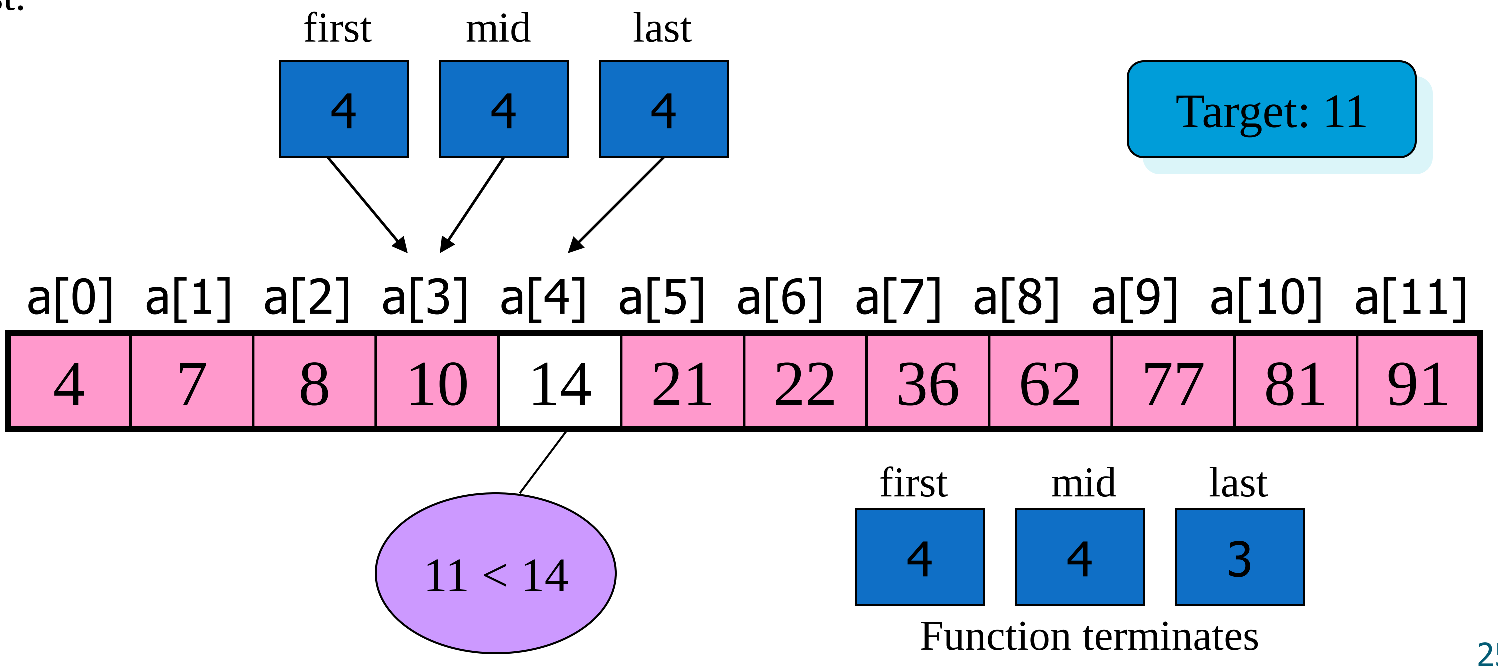
2.2.5 Binary Search Algorithm

- These settings of first and last set the mid index to 3 as follows:
  mid = (3 + 4) / 2 = 7 / 2 = 3



2.2.5 Binary Search Algorithm

- The test at index 3indicates that the target is greater than the list value, so we set first to mid + 1, or 4. We now test the data at location 4 and discover that 11 < 14. The mid is as calculated as follows:
- At this point, we have discovered that the target should be between two adjacent values; in other words, it is not in the list. We see this algorithmically because last is set to mid – 1, which makes first greater than last, the signal that the value we are looking for is not in the list.



2.2.5 Binary Search Algorithm

- Example algorithm:

  DATA – sorted array

  ITEM – Info

  LB – lower bound

  UB – upper bound

  ST – start Location

  MID – middle Location

  LAST – last Location

2.2.5 Binary Search Algorithm

```
1.  [Define variables]
        ST = LB, LAST= UB;
        MID = (ST+LAST)/2;
2.  Repeat 3 and 4 DO ST <= LAST & DATA[MID] != ITEM
3.  If ITEM < DATA[MID] then
        LAST = MID-1
    else
        ST = MID+1
4.  Set MID = INT((ST + LAST)/2)
    [LAST repeat to 2]
5.  If DATA[MID] = ITEM then
        LOC = MID
    Else
        LOC = NULL
6.  Stop
```

2.2.6 Merging Algorithm

- Suppose A is a sorted list with r elements and B is a sorted list with s elements. The operation that combines the element of A and B into a single sorted list C with n=r + s elements is called merging.

2.2.6 Merging Algorithm

```
• Algorithm: Merging (A, R,B,S,C)
        Here A and B be sorted arrays with R and S elements
        respectively. This algorithm merges A and B into an array
        C with N=R+ S elements
• Step 1:   Set NA=1, NB=1 and NC=1
• Step 2: Repeat while NA ≤ R and NB ≤ S:
            if A[NA] ≤ B[NB], then:
                    Set C[NC] = A[NA]
                    Set NA = NA +1
            else
                    Set C[NC] = B[NB]
                    Set NB = NB +1
            [End of if structure]
                    Set NC= NC +1
      [End of Loop]
```

2.2.6 Merging Algorithm

```
• Step 3: If NA >R, then:
                Repeat while NB ≤ S:
                 Set C[NC] = B[NB]
                 Set NB = NB+1
                 Set NC = NC +1
                 [End of Loop]
        else
                Repeat while NA  ≤ R:
                Set C[NC] = A[NA]
                Set NC = NC + 1
                Set NA = NA +1
                [End of loop]
        [End of if structure]
• Step 4: Return C[NC]
```

CODERINDEED

2.2.6 Merging Algorithm

- Complexity of merging: The input consists of the total number n=r+s elements in A and B. Each comparison assigns an element to the array C, which eventually has n elements. Accordingly, the number f(n) of comparisons cannot exceed n:

$$f(n) \leq n = O(n)$$

Exercises

- Find where the indicated elements of an array LA are stored, if the base address of LA is 200* and LB = 0

  a) double LA[10];L A[3]?

  b) int LA[26]; LA[2]?

*(assume that int(s) are stored in 4 bytes and double(s) in 8 bytes).

2.3 MULTIDIMENSIONAL ARRAY

- Two or more subscripts.

2-D ARRAY

- A 2-D array, A with *m X n* elements.
- In math application it is called *matrix*.
- In business application – table.
- <u>Example:</u>

  Assume 25 students had taken 4 tests.

  The marks are stored in 25 X 4 array locations:

	U0	U1	U2	U3
Stud 0	88	78	66	89
Stud 1	60	70	88	90
Stud 2	62	45	78	88
..	..	..	..	..
..	..	..	..	..
Stud 24	78	88	98	67

m

n

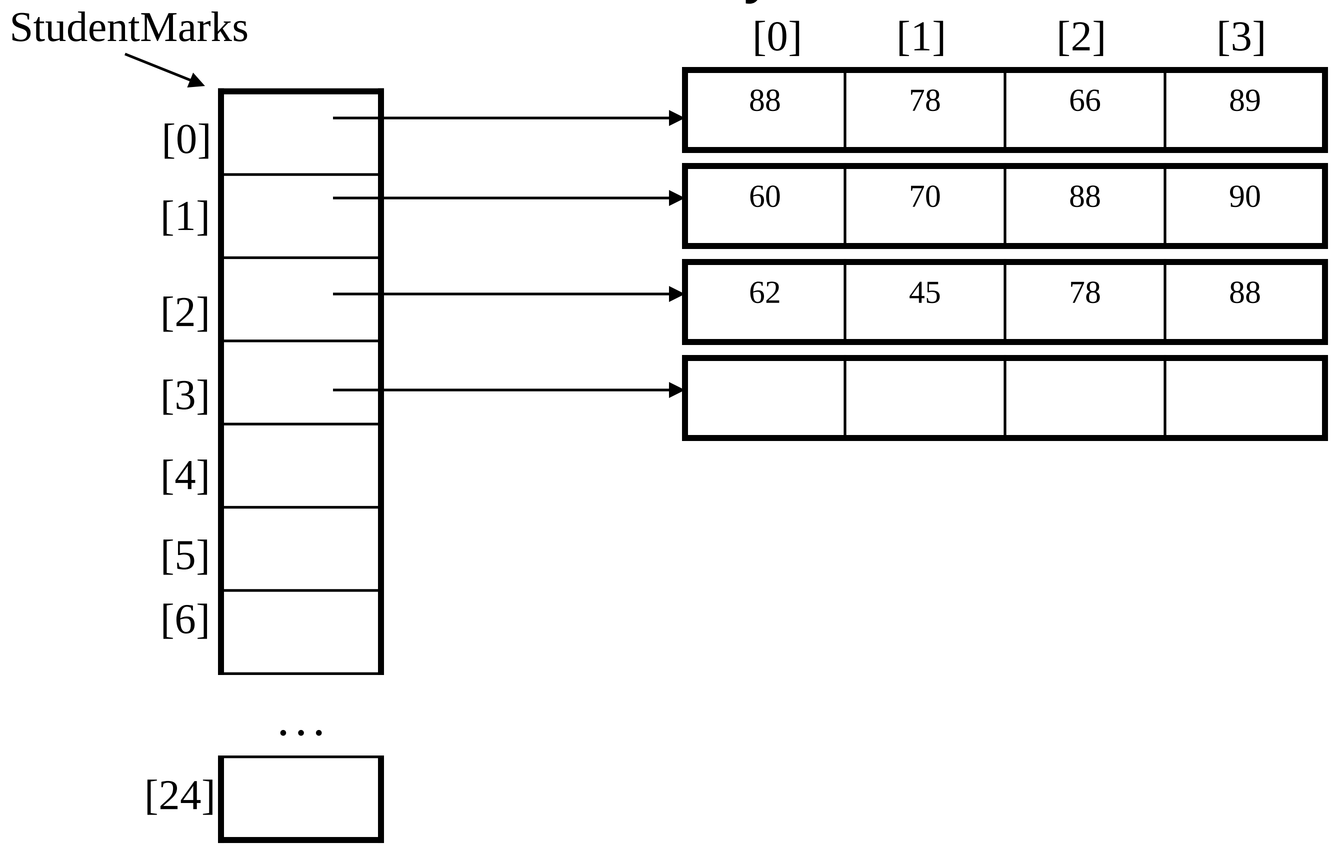
2-D ARRAY

- Multidimensional array declaration in C++:-

```
int StudentMarks [25][4];
StudentMarks[0][0] = 88;
StudentMarks[0][1] = 78;…..
OR
int StudentMarks [25][4] = {{88, 78, 66, 89},
                            {60, 70, 88, 90},…}
```

2.3.1 2-D ARRAY

- In C++ the 2-D array is visualized as follows:



2.3.2 Representation of 2D arrays in Memory

Column Major Order:

LOC(A[j, k])=Base(A)+w[m*k + j]

Row Major order:

LOC(A[j, k])=Base(A)+w[n*j + k]

Given: A 2-D array, A with *m X n* elements.

CODERINDEED

Thank You