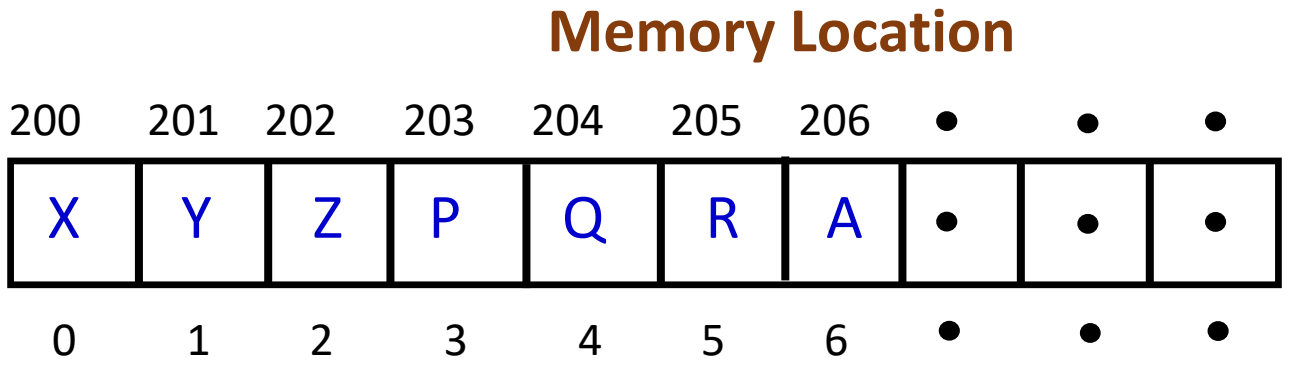
ARRAYS

PROF. NAVRATI SAXENA

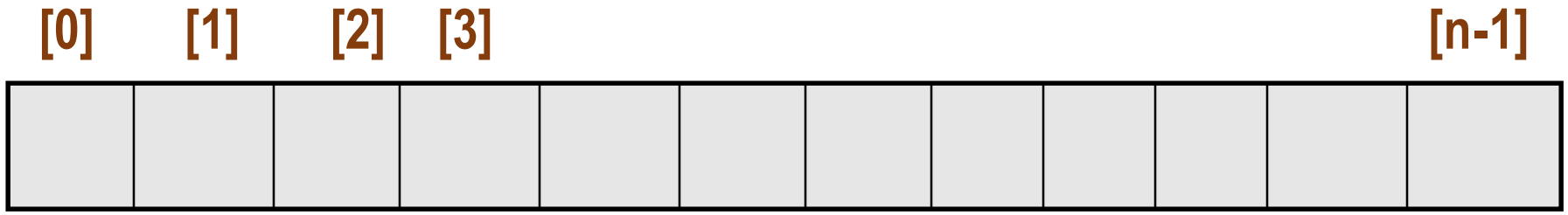
ARRAYS

- Set of finite homogeneous elements
- It means an array can contain one type of data only
  - Either all integer, or all character
- The elements of array will always be stored in the consecutive (continues) memory location



Array Format

- Elements of an array can be integers, floats, characters etc.
- All the elements share a common name with an index called subscript.
- In an array of n elements:



Arrays in C

- **Declaration**: int arr[10];
- **int**: data type or type of elements the array will store
- **arr**: Name of the array
- **[10]**: Size or length of array; Number of elements an array can store
- **Read an array** in C: read / write through loops – Defining the array

```
for(i=0; i<=9; i++)
        {
        scanf("%d", &arr[i]);
        printf("%d", arr[i]);
        }
```

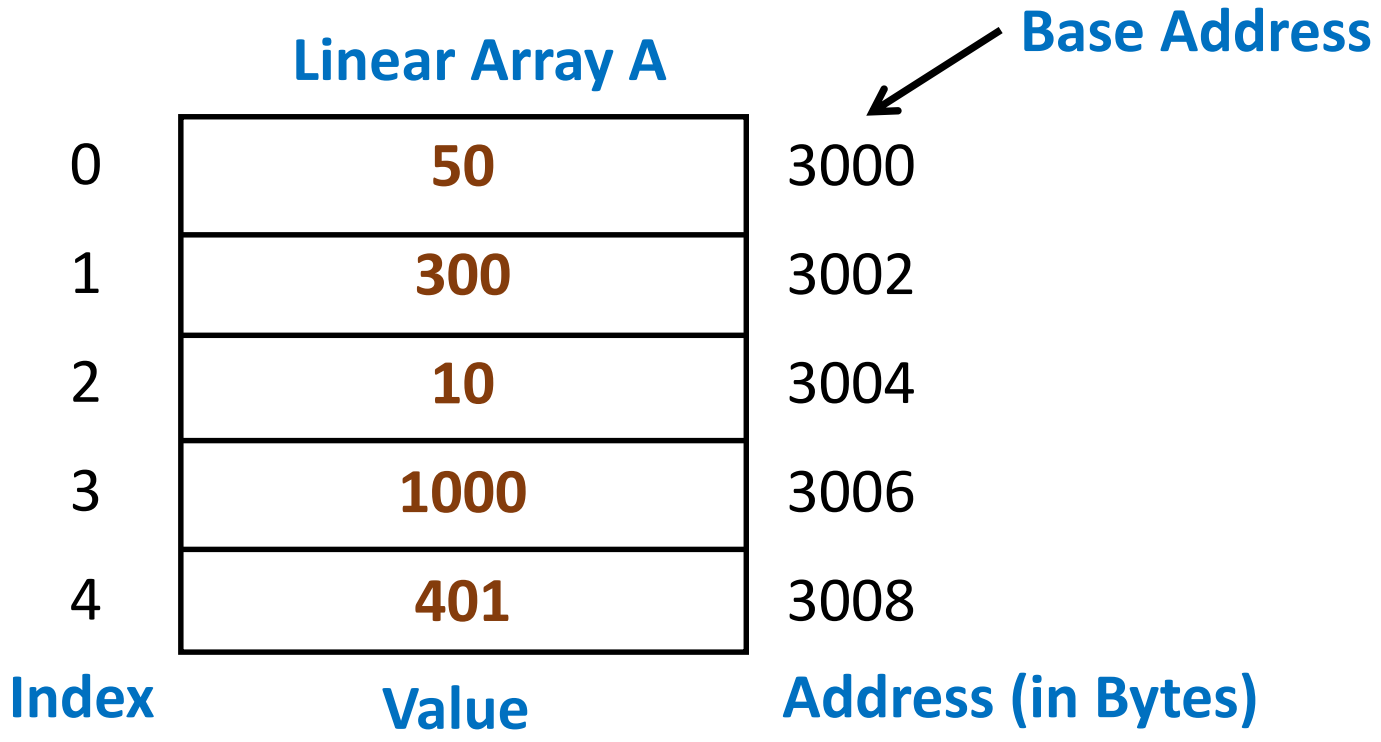
Advantages of Array

- Represents multiple data items using single name
- Implement other data structures like stacks, queues, tree, etc.
- Two-dimensional arrays are used to represent matrices.
- Many databases include one-dimensional arrays whose elements are records.

Disadvantages of Array

- Must know in advance the number elements to be stored
- Static structure
  - Fixed size
  - Allocated memory cannot be increased or decreased
  - If we allocate more memory than required; memory space will be wasted
  - If we allocate less memory then our elements can’t be stored
- Elements stored in consecutive memory locations
  - Insertion and deletion: difficult and time consuming

Represent a Linear Array in Memory



Consecutive memory locations

Declaration

- When declaring arrays, specify
  - Data type of array (integers, floats , characters…..)
  - Name of the array.
  - Size: number of elements

    array_type array_name[size];
  - Example:
    - int student[10];
    - float my_array [300];

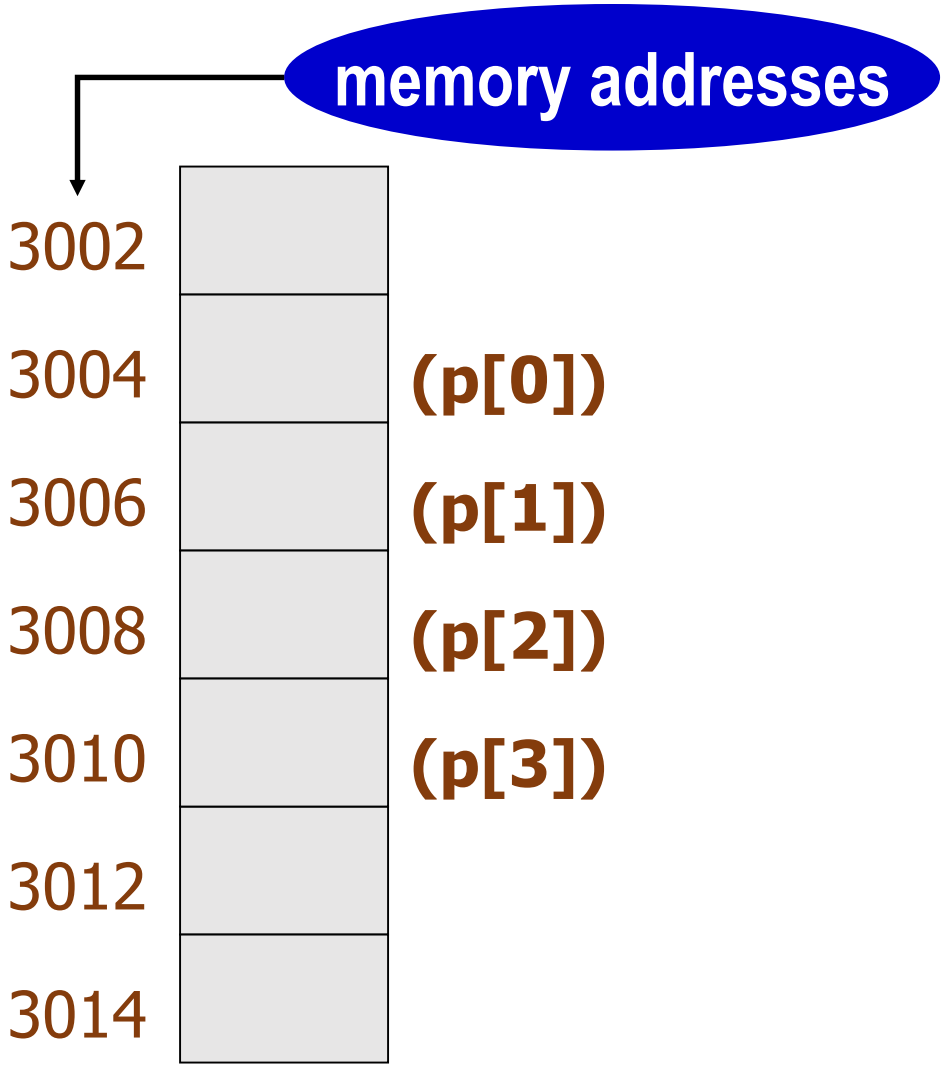
Example

- For example,

```
int   p[ 4 ];
```

Square bracket

- An array of integers of 4 elements.
- Starting memory address is determined by the operating system (just like simple variable).
- Contiguous memory locations are allocated.



Examples

- Integer array of 10 elements:

```
int  arr_1  [10];
```

- Character array of 500 elements:

```
char arr_2 [500];
```

- Float array of 1000 elements:

```
float arr_3 [1000];
```

Size of an Array

- Amount of storage required to hold an array is directly related to its data type and the size.

- **Example**

  Total size in bytes for a 1 D array:

  total bytes = *sizeof(data type) * size of array*

- **int a[7]** ;

  total_bytes = 2 * 7 (one integer needs 2 bytes of storage)

  = 14 bytes in memory

Sample Exercise

```
int main( ) {
  float fp_number[20] ;
  char character[20] ;
  printf(“%d”, sizeof(fp_number));
  printf(“%d”, sizeof(character));
  return(0);
}
```

Output:

- 80
- 20

Initializing arrays

- Array elements must be initialized at time of declaration
  - Otherwise they may contain garbage values

- Initialization can be done either at compile time or run time

Compile Time Initialization (1/2)

- Initialize elements of array in same way as any other ordinary variables at time of declaration
- Example: int num[3] = {1,1,1};
  - All three elements of integer array num is initialized to the same value 1
- float fl_num[5] = {1.0, 12.50, 100.35};
  - First three elements of array total will be initialized;
  - Rest two will be initialized to 0.
- If initializer list exceeds size of array, it will produce compiler error

Compile Time Initialization (2/2)

- float fl_num[5] = = {1.0, 12.50, 100.35};

- All individual array elements that are not assigned explicit initial values will automatically be set to zero.

- fl_num[0] = 1.0 fl_num[1] = 12.50 fl_num[2] = 100.35
  fl_num[3] = 0.0 fl_num[4] = 0.0

Initializing Integer arrays

- Another way of initializing the elements of the array is:

```
int num[   ] = {5, 10, -20, 0, 1000} ;
```

- The array size need not be specified explicitly when initial values are included as a part of array declaration.

- Array size will automatically be set equal to the number of initial values specified with in definition.

Initializing Arrays

The second way is to Initialize each array element separately

```
num[0] = 5;

num[1] = 10;

num[2] = -20;

num[3] = 0;

num[4] = 1000;
```

Run time Initialization

- Explicitly initializing an array at run time
- Normally used for large array size
- **Example**:

```
for(i=0; i<100; i++)
        {
        if(i < 50)
                num[i] = 0;
        else
                num[i] = 1;
        }
```

Basic operations of Arrays

- Traversing
- Insertion
- Deletion
- Searching
- Sorting
- Merging

Traversing Arrays

Reading / accessing the elements of the array

[0]	[1]	[2]	[3]	[4]
100	200	300	400	500

num [0] = 100
num [1] = 200
num [2] = 300
num [3] = 400
num [4] = 500

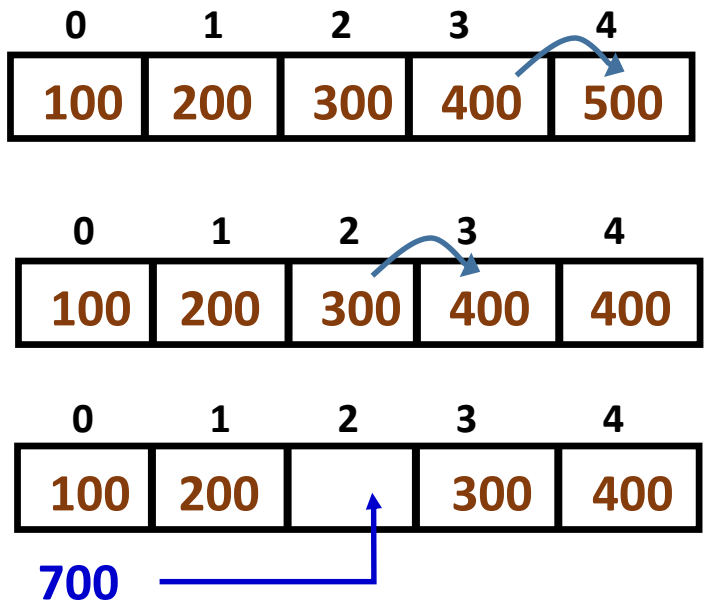
```
ALGORITHM: Traversal(A, LB, UB)
A is the array with Lower Bound LB and Upper Bound UB
Step 1: for i = LB to UB do
Step 2:   process/read/access A[i]
Step 3: end for loop
```

Insertion into Array

Add a new data item in the given collection of data items.

[0]	[1]	[2]	[3]	[4]
100	200	300	400	500

Assume: New element to be inserted is 700 at location/index 2. Shift all the elements from 2nd (last) location to the 4th location upwards by 1 place. Then insert 700 at the 2nd location.



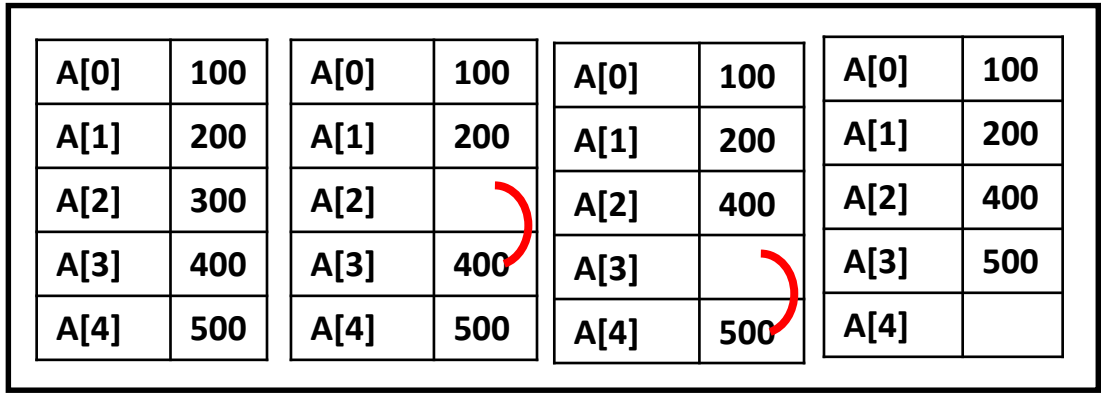

```
ALGORITHM: Insert(Arr, N, Item, index)
A: array with N elements; Item: item
to be inserted at position=index

Step 1: for i = N-1 down to index
Step 2:    Arr[i+1] ← Arr[i]
Step 3: Arr[index] ← Item
Step 4: N ← N+1
```

Deletion from Array

To delete an existing data item from the given collection of data items.

Let A[5] = 100, 200, 300, 400, 500
Delete item 300 from index 2
Let Item = 300
Shift 400 to index 2; 500 to index 3




```
ALGORITHM: Delete (Arr, N, Item, index)
Arr is the array with  N elements; Item is
to be deleted from position = index

Step 1: Item ← Arr[index]
Step 2: for i = index down to N-1
Step 3:     Arr[i] ← Arr[i+1]
Step 4: N ← N-1
```

Searching in Arrays

To find out the location of an data item if it exists in the array

[0]	[1]	[2]	[3]	[4]
150	500	350	200	250

To search item **200**

1. Start from the beginning
2. Compare 200 with each item of the array until
    1. 200 is found or end of array is met

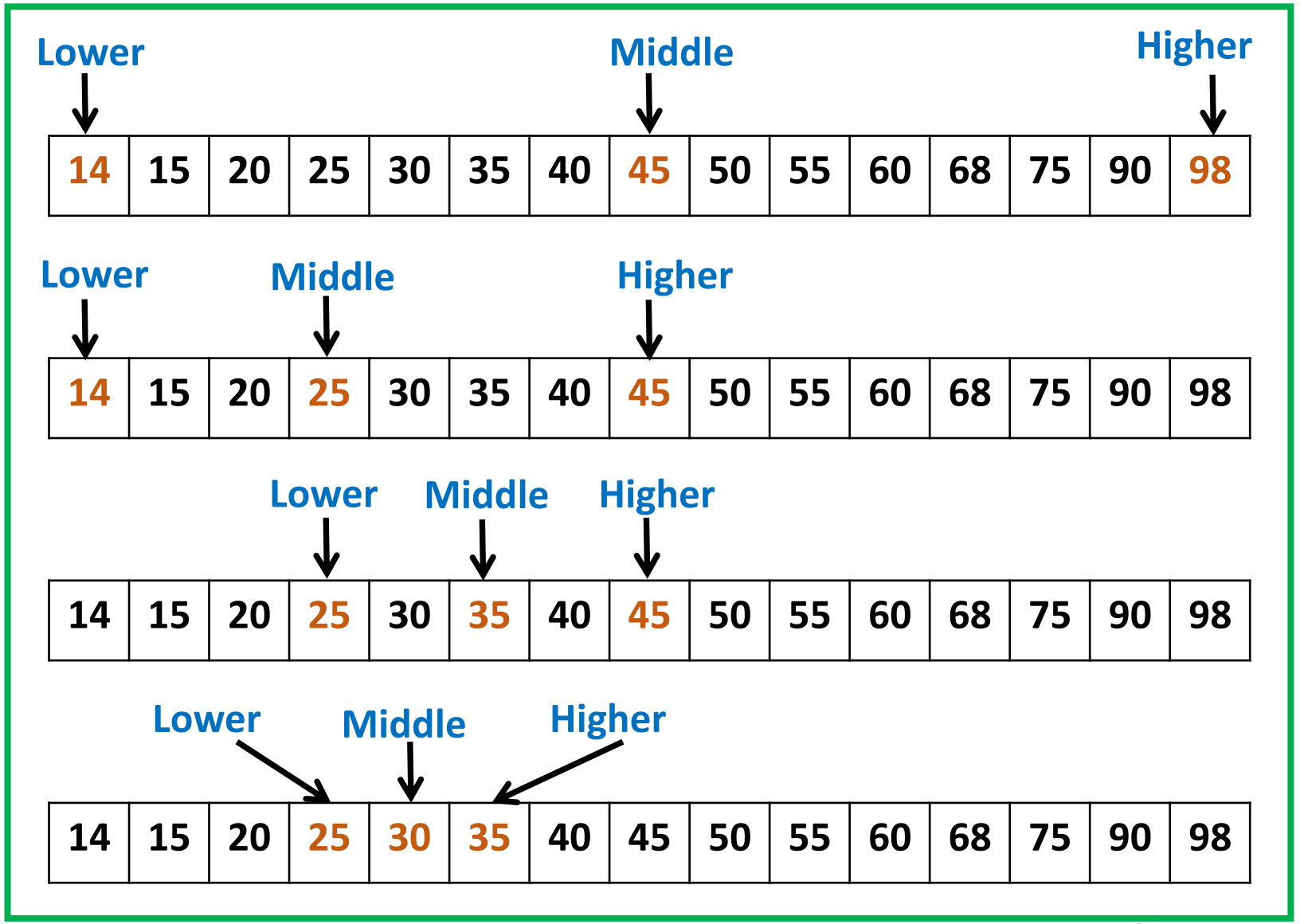
Linear/Sequential Search

INPUT: array, item

1. *Begin*
2. *Set i ← 0*
3. *Repeat steps 4 and 5 while i < N*
4. *if (array[i] = item)*
5. *print i, item*
6. *i ← i+1*
7. *End*

Binary Search

- Sort the array
- Divides the array into two smaller sub-arrays
- Recursively operate the sub-arrays.
- Reduces the search space to half at each step



Binary Search Algorithm

```
// A iterative binary search procedure. It returns location of item in given array arr[low..high] if present,
otherwise it returns: -1

int binarySearch (int arr[], int lower, int higher, int item)
{
   while (lower <= higher)
            {
            mid = lower + (higher - lower) / 2;
            if (arr[mid] = x)                       //Check if x is present at mid
                        return mid;
          if (arr[mid] < x)
                        lower = mid + 1;            // If x greater, ignore left half
            else
                         higher = mid - 1;          // If x is smaller, ignore right half
            }
       return -1;                                   // if we reach here, then element isnot present
}
```

	Linear Search	Binary Search
1	This can be used in sorted and unsorted array	This can be used in only in sorted array
2	Array elements are accessed sequentially	One must have direct access to the middle element in the sub list
3	Slow	Faster
4	This can be used in single and multi dimensional array	Used only in single dimensional array
5	This technique is easy and simple in implementing	Complex in operation

Sorting

- Arrangement of the elements of the array in some order
  - Numeric array – ascending or descending order

- Different sorting methods
  - Bubble sort
  - Selection sort
  - Shell sort
  - Quick sort
  - Heap sort
  - Insertion sort

Merging from Array (1/2)

Combine the data items of two sorted arrays

Array X

[1]	[2]	[3]	[4]	[5]	[6]
20	50	60	90	105	110

Array Y

[1]	[2]	[3]	[4]
30	45	55	100

Merged Array

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
20	**30**	**45**	**50**	**55**	**60**	**90**	**100**	**105**	**110**

Merging from Array (2/2)

Algorithm

1. Create an array Z[] of size = size of (X) + size of (y)
2. Simultaneously traverse X[] and Y[]
    - Pick smaller of current elements in X[] and Y[]
    - Copy this element to next position in Z[]
    - Move ahead in Z[] and the array whose element is picked
3. If there are remaining elements in X[] or Y[], copy them also in Z[]

Types of Arrays

- Single Dimension Array
  - Array with one subscript
- Two Dimension Array
  - Array with two subscripts (Rows and Column)
- Multi Dimension Array
  - Array with Multiple subscripts

Two dimensional array

- Each element is identified by a pair of subscripts (num[m] [n])
  - m = row; n = column
- The elements are stored in continuous memory locations
- Order of the matrix
  - Number of rows and columns in the matrix
  - Denoted as *m x n*
- Number of elements in the array = number of rows X number of columns.

	num[0]	num[1]	num[2]
num[0]	10	20	30
num[1]	40	50	60
num[2]	70	80	90

```
num[2][3]
for(i=0; i<2; i++)
      for(j=0;j<3;j++)
            print num[i][j]
```

Two-dimensional Array: Row-major Method

- All the first-row elements are stored in sequential memory locations
- Then all the second-row elements are stored and so on. Ex: A[Row][Col]

10000	**100**	num[0][0]
10002	**200**	num[0][1]
10004	**300**	num[0][2]
10006	**400**	num[1][0]
10008	**500**	num[1][1]
10010	**600**	num[1][2]
10012	**700**	num[2][0]
10014	**800**	num[2][1]
10016	**900**	num[2][2]

Two-dimensional Array: Column-major Method

- All the first column elements are stored in sequential memory locations
- Then all the second- column elements are stored and so on. Ex: A [Col][Row]

10000	**100**	num[0][0]
10002	**200**	num[1][0]
10004	**300**	num[2][0]
10006	**400**	num[0][1]
10008	**500**	num[1][1]
10010	**600**	num[2][1]
10012	**700**	num[0][2]
10014	**800**	num[1][2]
10016	**900**	num[2][2]

How will you store a table or matrix?

	Item1	Item2	Item3
Shop1	**1310**	**1275**	**1365**
Shop2	**1210**	**1190**	**1325**
Shop3	**1405**	**1235**	**1240**
Shop4	**1260**	**1300**	**1380**

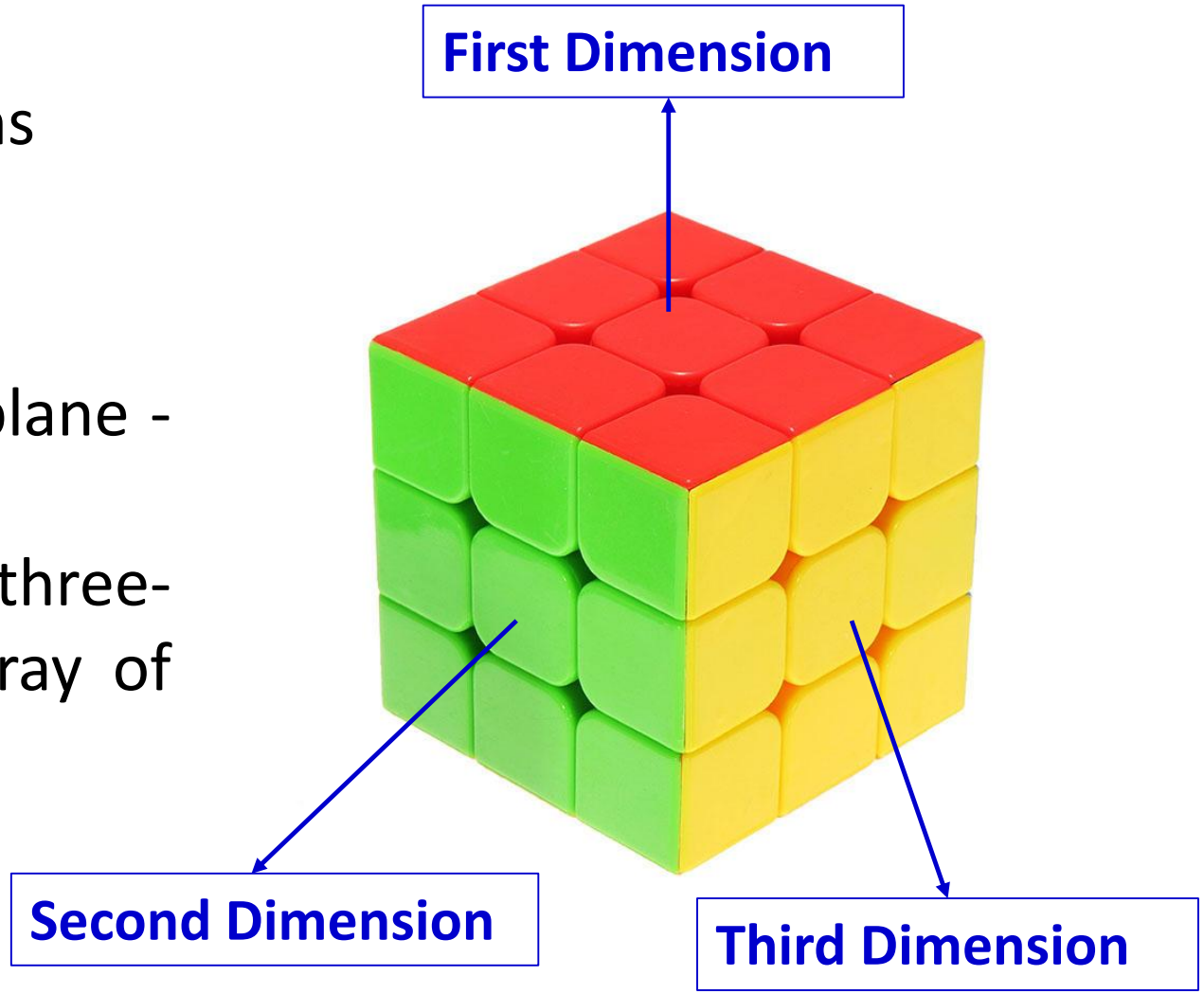
Store this data, create 2-D array : int arr[4][3];

(data type) array_name[# of rows] [# of columns];

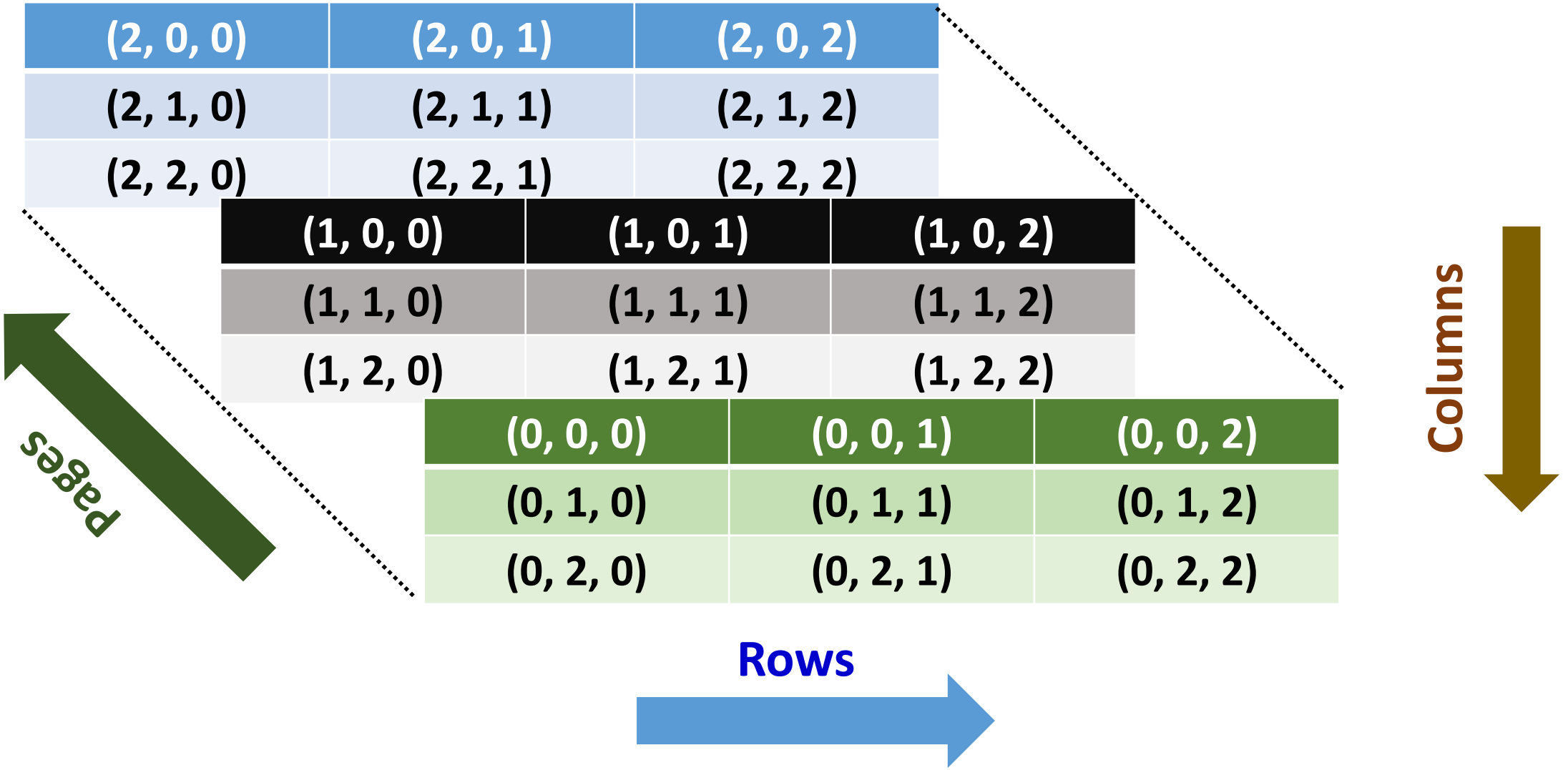
Multidimensional Arrays

- Arrays with two or more dimensions

- In a three dimensional array:
  - The first dimension is called a plane - consists of rows and columns.
  - C language considers the three-dimensional array to be an array of two-dimensional arrays.



C View of Three-dimensional Array



DISADVANTAGE OF ARRAYS

- Static memory allocation
- Maximum size reserved in advance
  - Problems of Less Resource Utilization
- Different data types could not be stored in an array

Thank you!