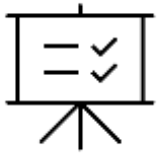
Advanced Algorithms & Data Structures

AIM OF THE SESSION

To familiarize students with the basics of data structures, and algorithmic analysis to evaluate time complexities.

INSTRUCTIONAL OBJECTIVES



This Session is designed to:

- Understand the concept of Data Structures.
- Understand the concept of arrays.
- Analyse the how the access array elements..

.

LEARNING OUTCOMES

At the end of this session, you should be able to:

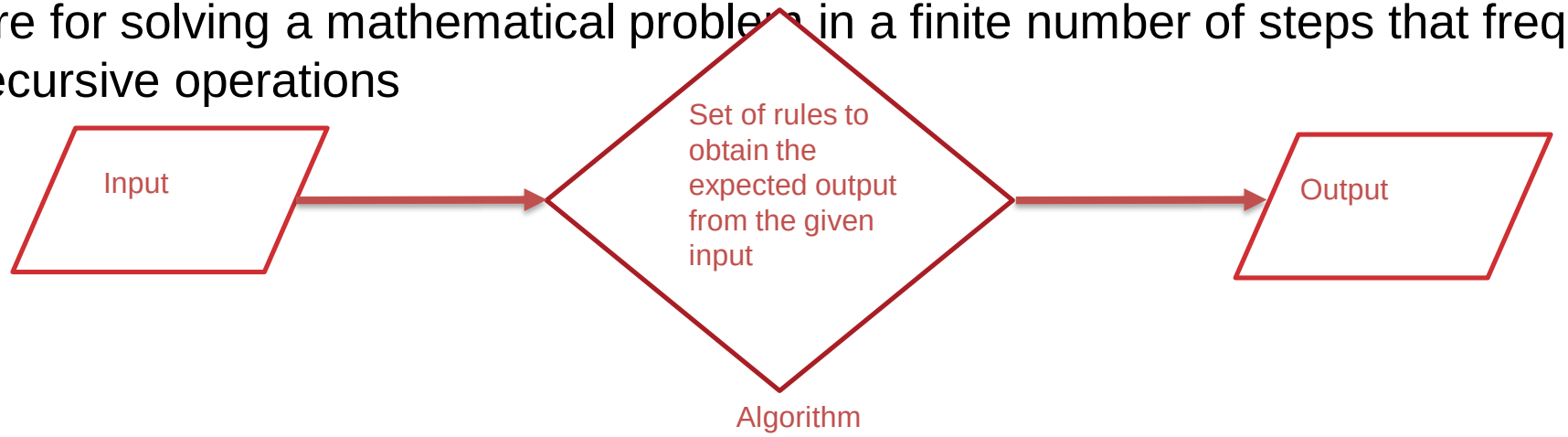
- Evaluate the time complexities of the different Algorithmic solutions for various types of Problems.

ALGORITHM

- A set of finite rules or instructions to be followed in calculations or other problem-solving operations

(or)

- A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations



DATA STRUCTURE

- Data Structures are used to store and organize the data in memory in an efficient manner
- It is a way of arranging data on a computer to be accessed and updated efficiently.
- Data Structure is used not only for organizing the data but also for processing, retrieving, and storing data.

COMPLEXITY OF AN ALGORITHM

- Complexity in algorithms refers to the amount of resources (such as time or memory/Space) required to solve a problem or perform a task.
- The most common measure of complexity is time complexity, which refers to the amount of time an algorithm takes to produce a result as a function of the size of the input.
- Memory/Space complexity refers to the amount of memory used by an algorithm.

TIME COMPLEXITY

- Time taken by the algorithm to solve the problem.
- It is measured by calculating the iteration of loops, the number of comparisons etc.
- Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.

Array in Data Structure

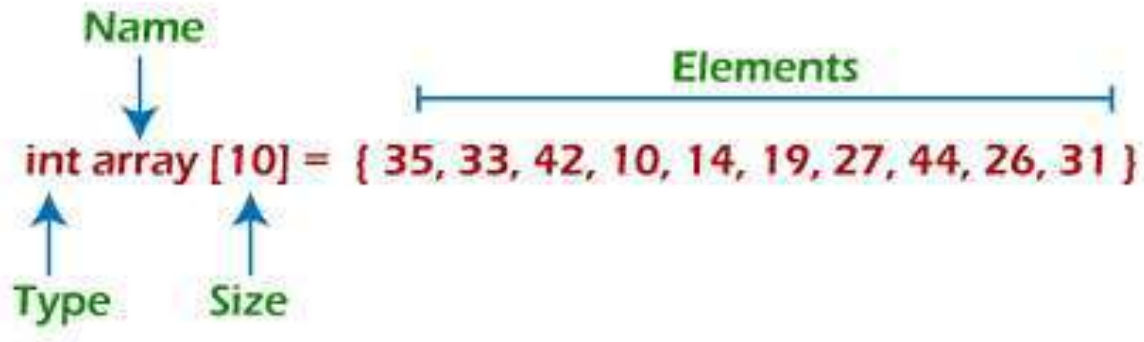
Arrays are defined as the collection of similar types of data items stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

Properties of array:

- Each element in an array is of the same data type and carries the same size that is 4 bytes.
- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Representation of an array

We can represent an array in various ways in different programming languages. As an illustration, let's see the declaration of array in C language –

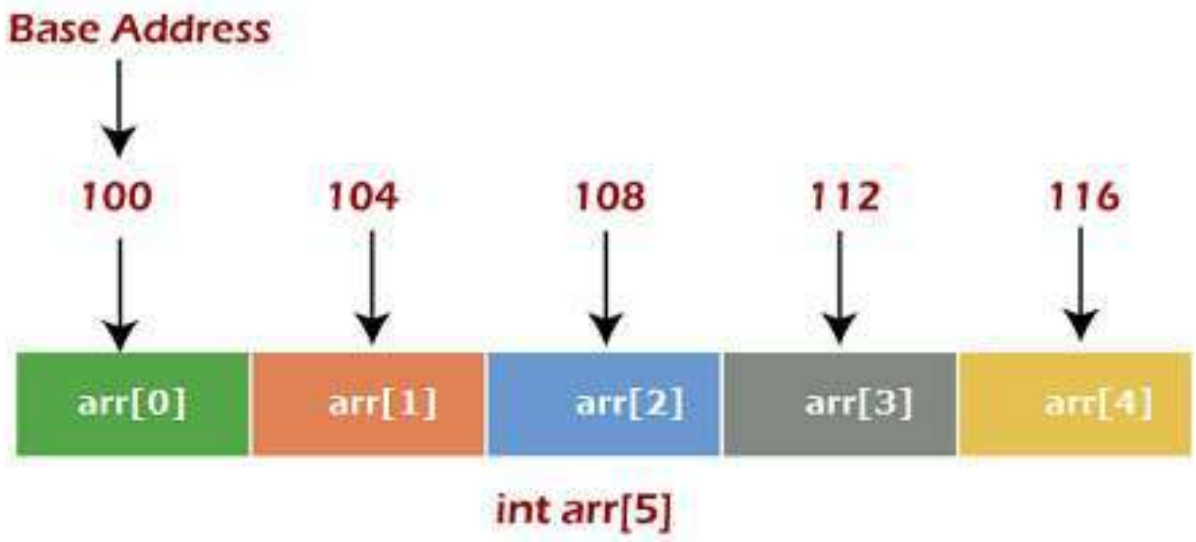


As per the above illustration, there are some of the following important points -

- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

- Why are arrays required?
- Arrays are useful because -
- Sorting and searching a value in an array is easier.
- Arrays are best to process multiple values quickly and easily.

Memory allocation of an array

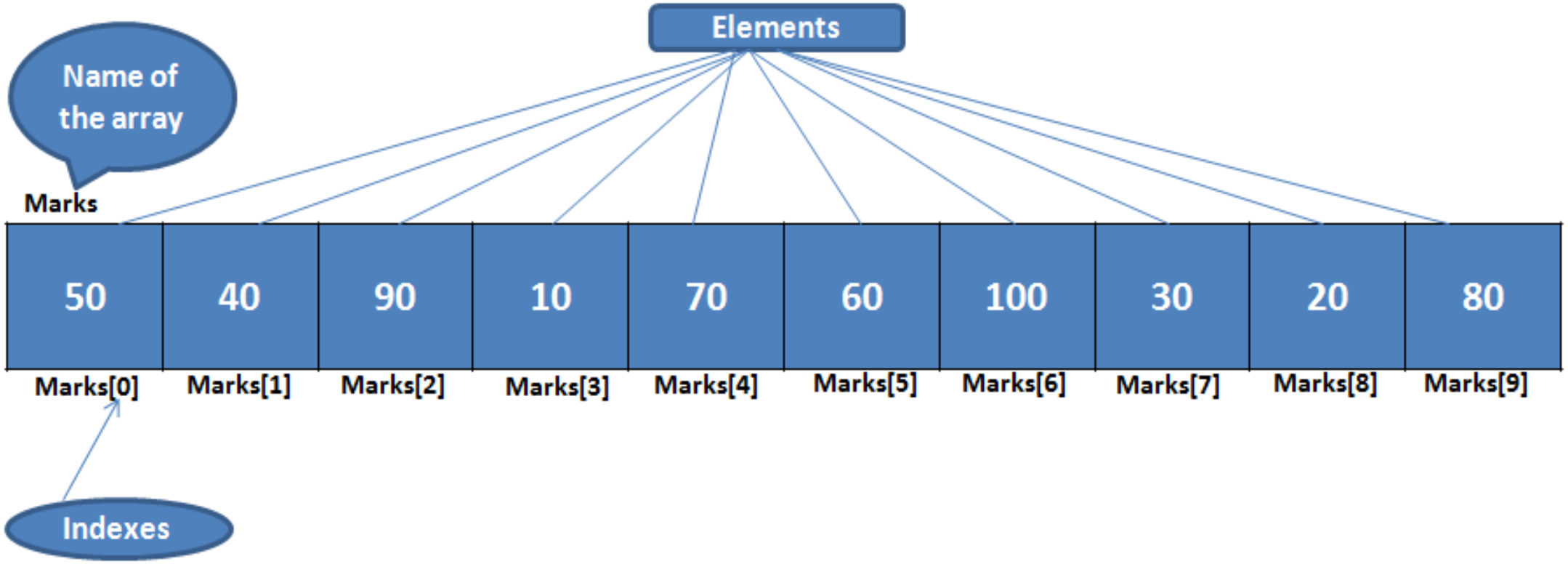


Arrays

- An array is a fundamental data structure that enables the storing and manipulation of potentially huge quantities of data.
- An array can be defined as a data structure consisting of an ordered set of data values of the homogeneous type.
- An array is a collection of individual data elements that is
  a. Ordered
  b. Fixed in size
  c. Homogeneous

- The elements of the array are stored in consecutive memory locations and are referenced by an integer index also known as subscript. The index begins at a zero and is always written inside square brackets.
- Following are the two types of arrays:
  1. Single dimensional arrays
  2. Multi dimensional arrays
- Single dimensional arrays are represented as vectors containing a row of values.
- Multi dimensional arrays can be viewed as tables containing several rows and columns of data.

- **Example:** ***int marks[10];***



Declaration:

- ***Syntax:*** ***data-type ArrayName[SIZE];***
- Ex: 1. int ages[10];
  2. float heights[20];
  3. char name[20];
- The ***data-type*** can be any basic data type or an user-defined data type.
- ***Array Name*** must be a valid identifier.
- The ***SIZE*** of an array represents the maximum number of elements in the array.

Initialization:

- ***Syntax:*** ***Data-type array-name[size] = { value list } ;***
- The ***value list*** is a comma-separated list of constants.
- Ex:

```
void main()
{
        int age[5] = {30,25,45,35,60};
        printf("%d" , age[0]);
}
```

Accessing Elements:

- ***Syntax:*** ***ArrayName[index]***
- To access array elements in an array, specify the array name, followed by square braces enclosing an integer, which is called the array index.
- The array index indicates the particular element of the array which we want to access.
- For example if we want to access the fifth element of an array "A" then write:

printf("%d", A[4]);

Operations on Arrays:

- There are number of operations that can be performed on arrays. These operations include:
  - ❑Traversing an array
  - ❑Inserting an element in an array
  - ❑Deleting an element form an array
  - ❑Merging two arrays
  - ❑Rotation elements of an array
  - ❑Reversing array elements
  - ❑Searching an element in an array
  - ❑Sorting an array in ascending order

Traversing an Array:

- Traversing means to access all the elements of the array, starting from first element up to the last element in the array on-by-one.
- Algorithm:
  1. Let LB be the lower bound and the UP be the upper bound of linear array A.
  2. Set i = Lower Bound LB
  3. Repeat for i = LB to UB

     Display A[i]

     [End of the loop]
  4. Exit

Inserting an element in an array:

- For inserting the element at required position, elements must be moved one position right to new locations.
- Algorithm:
  1. Set i = n-1
  2. Repeat steps while i>=pos

     Set A[i+1] = A[i] [shift the elements by 1 position right]

     Set i = i – 1

     [End of loop]
  3. Set A[pos]= ele
  4. Set n = n + 1
  5. End

Deleting an element from an array:

- For deleting the element at required position, elements must be moved one position left from the next position of the deleting element.
- Algorithm:
  1. Set i = pos
  2. Repeat steps while i<=n-1

     Set A[i] = A[i+1] [shift the elements by 1 position left]

     Set i = i + 1

     [End of loop]
  3. Set n = n - 1
  4. End

Merging Two Sorted Arrays:

Given two sorted arrays of size m and n respectively. Also given an empty third array of size (m+n). write code to merge the two arrays and store the result in the third array.

Input:

a = {1, 4, 5, 9}

b = {0, 2, 3, 7, 15, 29}

Output:

c = {0, 1, 2, 3, 4, 5, 7, 9, 15, 29}

Algorithm:

- Algorithm to merge two arrays A[0..m-1] and B[0..n-1] into an array C[0..m+n-1] is as following:

1. Introduce read-indices **i**, **j** to traverse arrays A and B, accordingly. Introduce write-index **k** to store position of the first free cell in the resulting array. By default **i** = **j** = **k** = 0.
2. At each step: if both indices are in range (**i** < m and **j** < n), choose minimum of (A[**i**], B[**j**]) and write it to C[**k**]. Otherwise go to step 4.

3. Increase **k** and index of the array, algorithm located minimal value at, by one. Repeat step 2.

4. Copy the rest values from the array, which index is still in range, to the resulting array.

Rotate an array by d elements left:

- Write a C program to left rotate an array by d position.

Example:

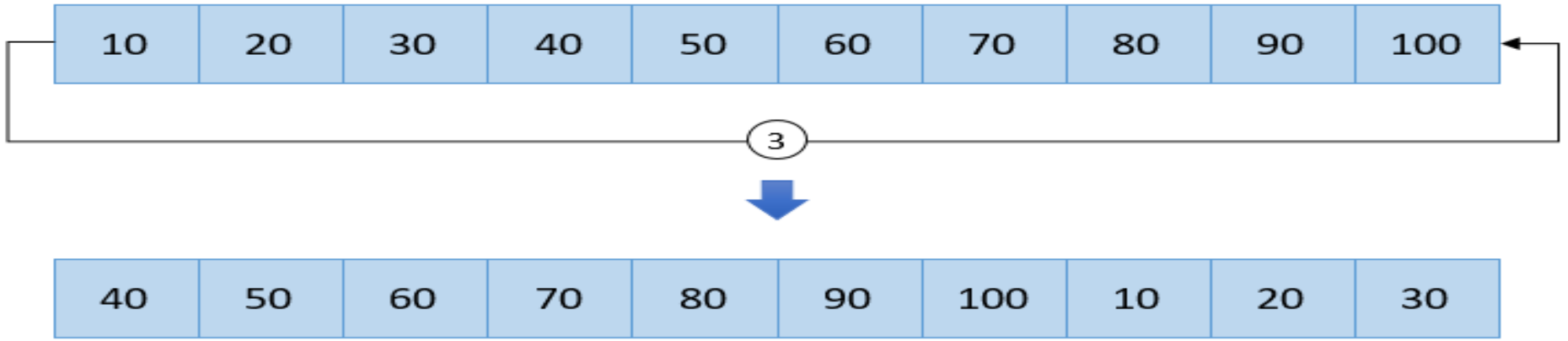
Input

Input 10 elements in array: 1 2 3 4 5 6 7 8 9 10

Input number of times to rotate: 3

Output

Array after left rotation 3 times: 4 5 6 7 8 9 10 1 2 3



Method:1

1. Let **A** be an array to be rotated left by **d** positions.
2. Print **A** before rotation.
3. Define one temporary variable **temp** which will hold the first element of array **A**.
4. First copy the first element of array **A** into **temp** and then shift all elements one position left. After shifting, copy back **temp** into *Array* **A** as last element. Repeat this step **d** times.
5. Print the rotated array **A**.

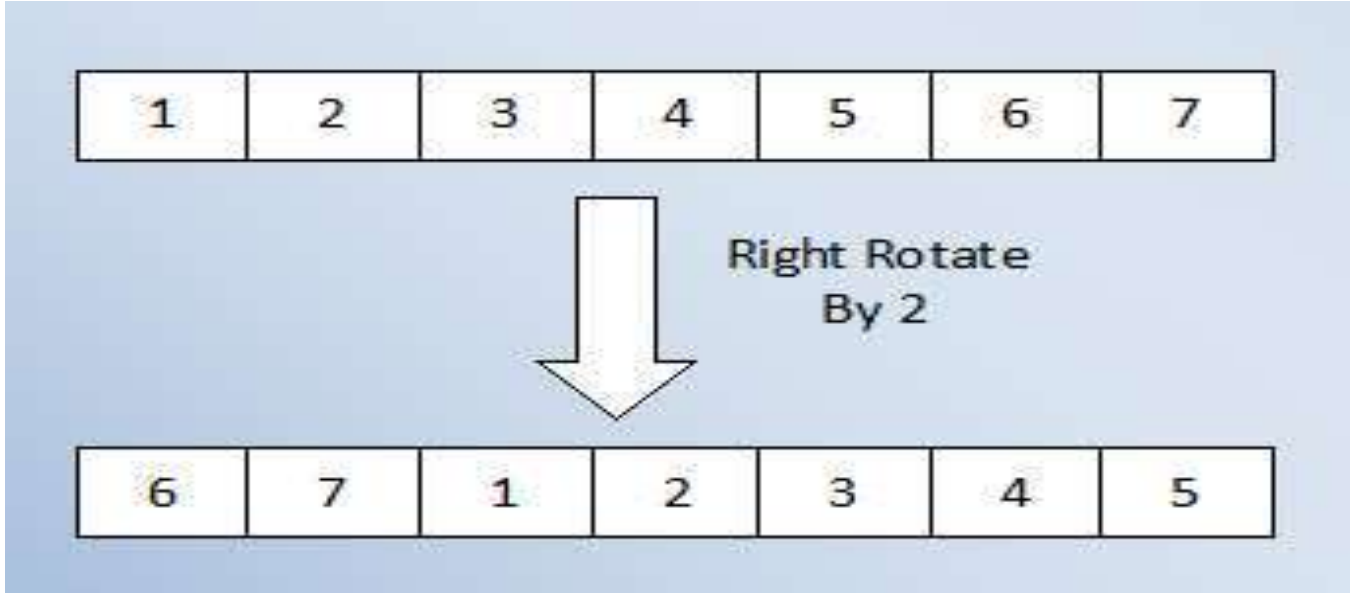
Method: 2 . Using Temporary Array [Assignment]

Input arr[ ] = [1, 2, 3, 4, 5, 6, 7], d = 2, n =7

1. Store d elements in a temp array temp[ ] = [1, 2]
2. Shift rest of the arr[ ] arr[ ] = [3, 4, 5, 6, 7, 6, 7]
3. Store back the d elements arr[ ] = [3, 4, 5, 6, 7, 1, 2]

Rotate an array by d elements right:

Program: Write a C program to right rotate an array by d position. [Assignment]

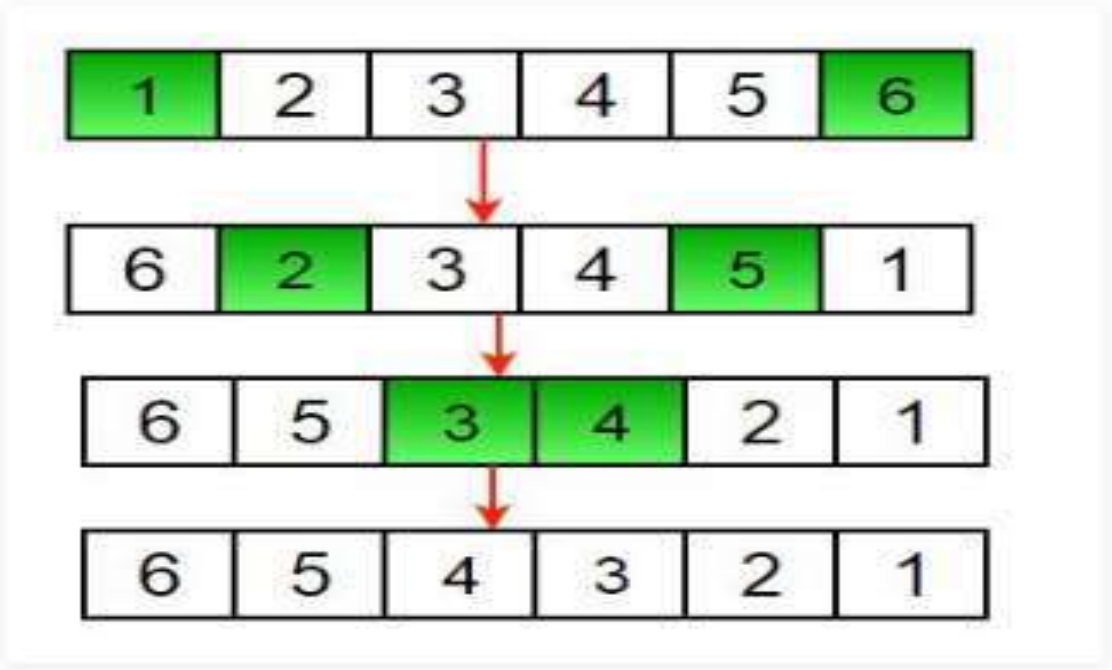


Reverse elements of an array:

Method

1. Let **A** be an array to be reversed with n number of elements.
2. Define two indexes i and j pointing first and last element of an array. [i=0 , j=n-1]
3. Repeat the loop for **n/2** number of times.
   a. In each iteration swap the array elements pointed by pointers i and j
   b. After swapping modify indexes as i=i+1 and j=j-1

Example:



Linked List

Linked List

- The term ***list*** refers to a linear collection of data items.
- One way to store such data is by means of ***Arrays and Linked List***.
- The ***Array*** implementation uses static structures where as ***Linked lists*** implementation uses dynamically allocated structures.

Array Implementation:

The array implementation of a list has certain drawbacks. These are:

1. Operations like insertion and deletion at a specified location in a List requires a lot of movement of data, therefore, leading to inefficient and time consuming algorithm.
2. Memory storage space is wasted; very often the List is much shorter than the array size declared.
3. List cannot grow in its size beyond the size of the declared array if required during program execution.

Linked List Implementation:

Following are the advantages of Linked Lists over Arrays:

1. Linked list is a dynamic structure where as array is a static structure.
2. Insertion and deletion of nodes requires no data movement.

Types of Linked List :

Following are the 4 types of linked lists:

1. Singly Linked List
2. Circular Linked List
3. Doubly Linked List
4. Circular Doubly Linked List

Singly Linked List:

- A **Singly Linked List** is a linear collection of data elements, called **nodes.**
- Here each node is divided into two fields. The first field contains the information of the element, and the second field contains the address of the next node in the list.
- The first field is called **Data field** and the second field is called **Next field.**
- Following figure shows the structure of a node in a linked list. The nodes in a linked list are called **self-referential** structures.

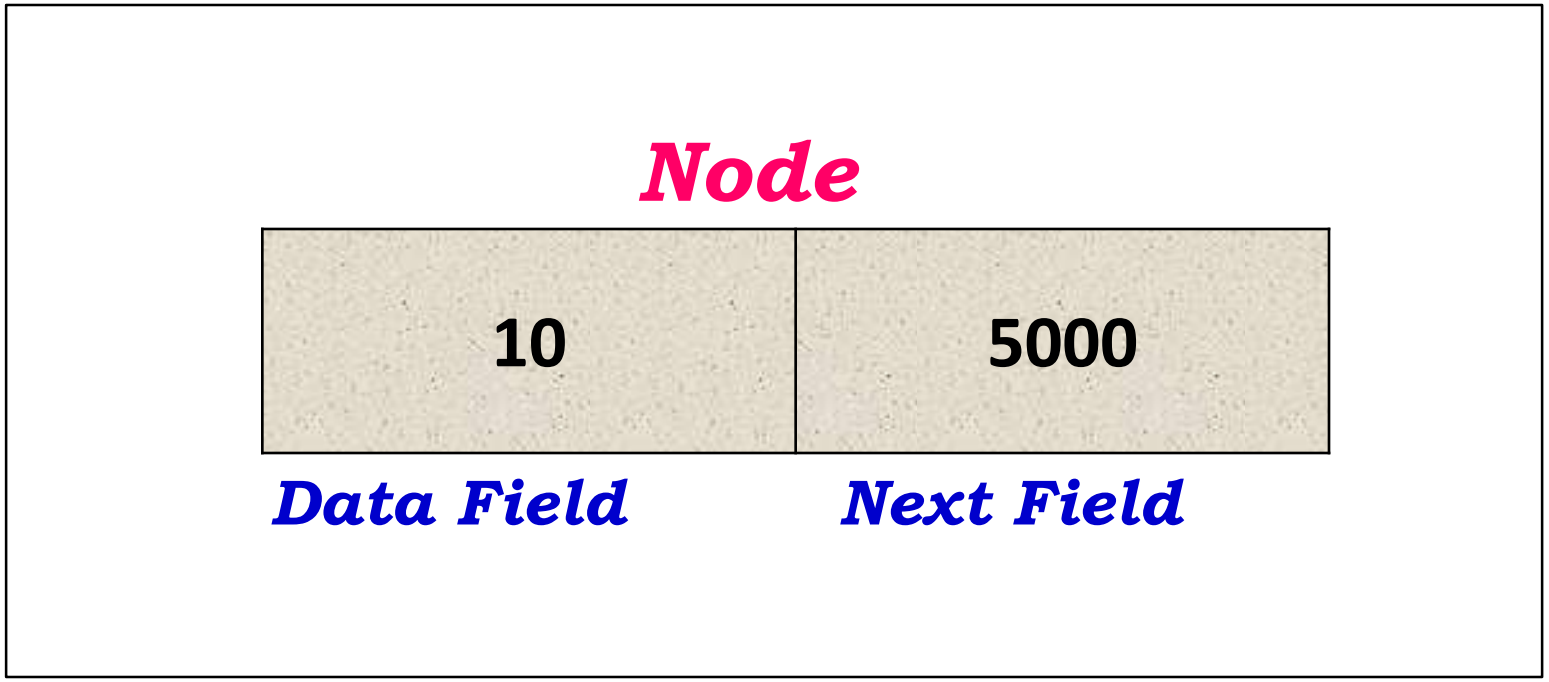
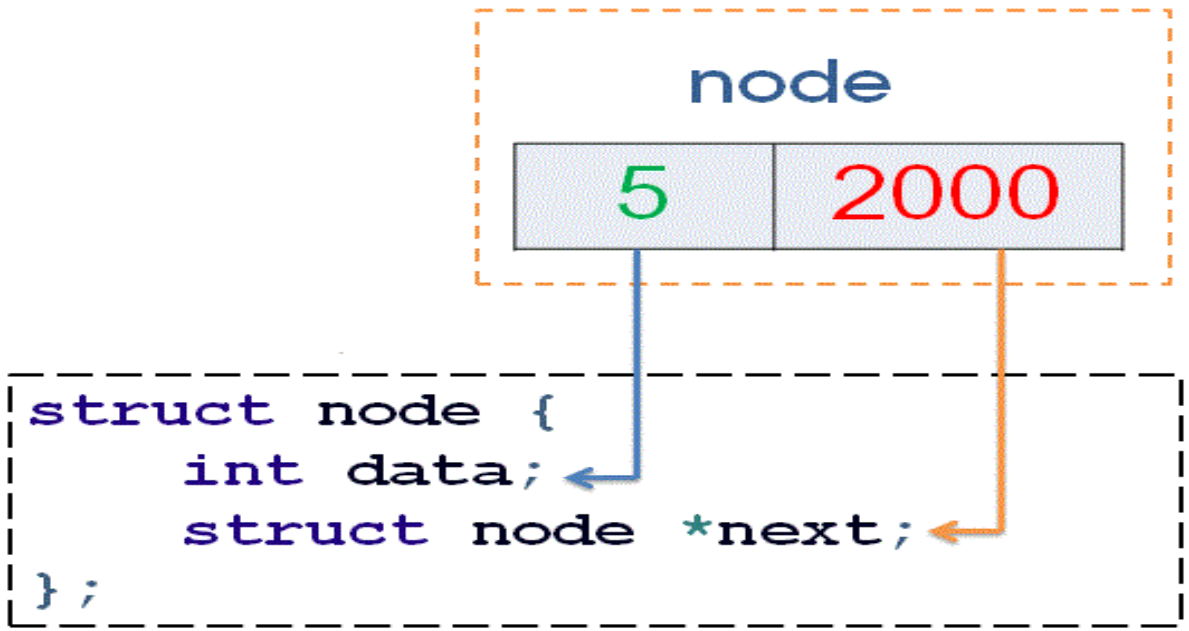


Figure: Structure of a node in a Linked List



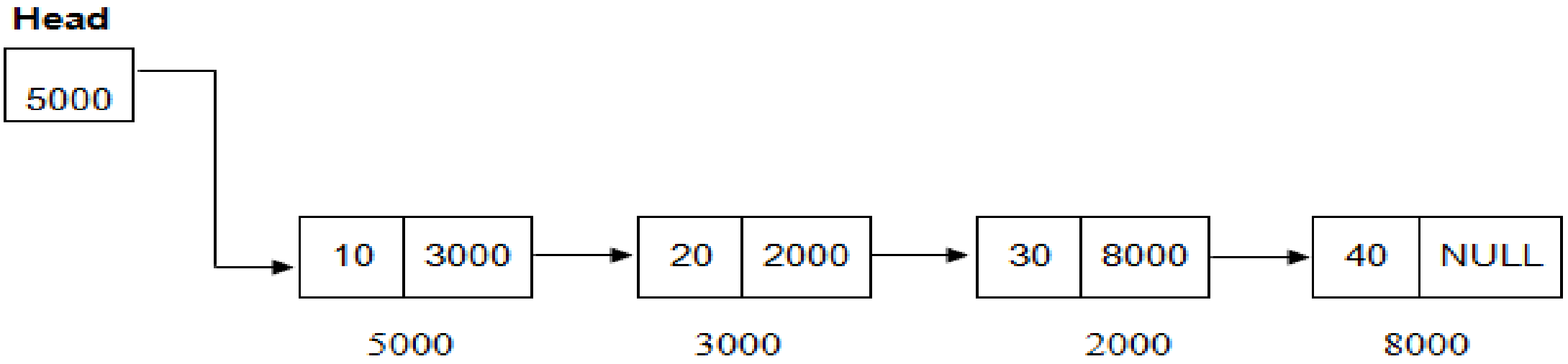
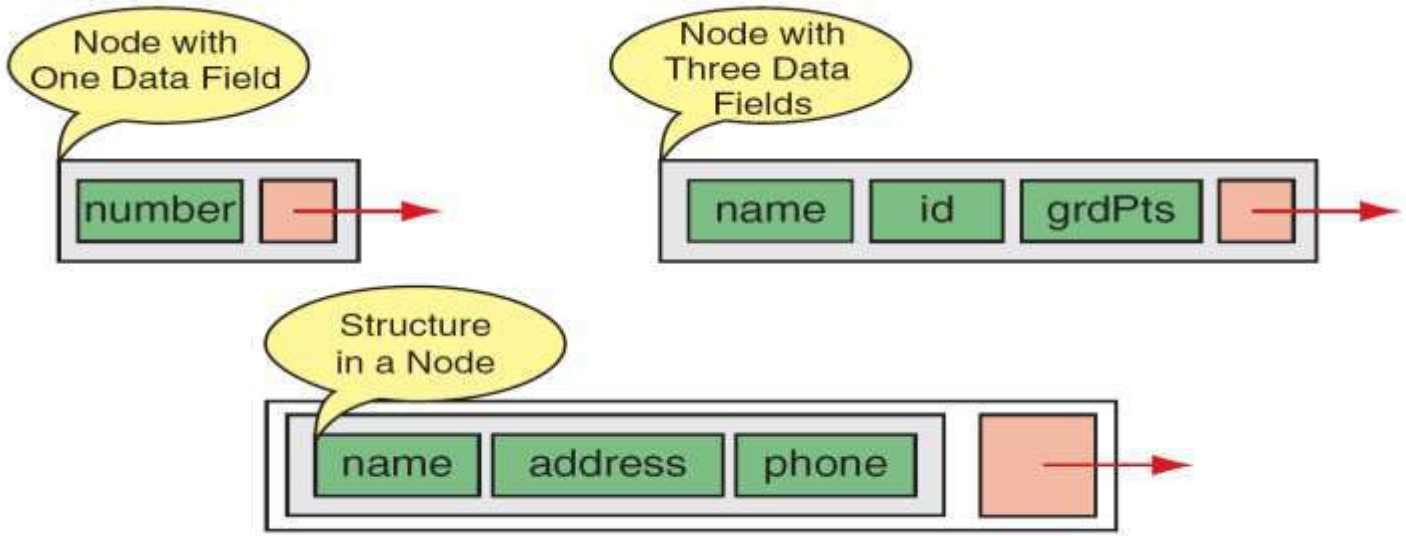


Figure: Singly Linked List with 4 Nodes

- There is an arrow from a node to the next node in the list.
- The pointer to the last node contains a special value called the ***null pointer*****.**
- Linked list also contains a pointer variable, called ***HEAD*****,** which points to the first node of the list.
- As nodes are added to a list, memory for a node is dynamically allocated. Thus the number of items that may be added to a list is limited only by the amount of memory available.
- If HEAD is NULL then list is said to be ***empty list*** or ***null list.***

- The data part in a node can be single field or multiple fields but the next part should be single field.



Operations on Linked List

- There are number of operations that can be performed on Linked list.
  - ❑Traversal
  - ❑Insertion
  - ❑Deletion
  - ❑Reversing list
  - ❑Searching
  - ❑Sorting

1. Insert a node:

- Insertion logic varies depending on where in the list the element is going to be inserted, first, middle or at the end. In all situations it is as simple as making the pointer point to different nodes and hence insertion at any place will be much faster.

a. Insert at Beginning

1. Create a new node
2. Make the ***next*** field of new node point to the node pointed by ***head***
3. Make the ***head*** pointer point to new node

```
newnode -> next = head;
head = newnode;
```

b. Insert in Middle

1. Create a new node.
2. We must point the new node to its successor by making the ***next*** field of new node point to the ***next*** of predecessor node.
3. Make the predecessor node point to new node.

```
newnode -> next = temp -> next
temp -> next = newnode;
```

c. Insert at End

1. Create a new node.
2. Make the ***next*** field of new node point to NULL.
3. Make the predecessor node point to new node.

```
temp -> next = newnode;
newnode -> next = NULL;
```

2. Delete a Node:

- Again deletion logic varies depending on from where the node is getting deleted, beginning, middle or at the end.

a. Delete First Node

1. Store the node to be deleted in a temporary pointer
2. Make ***head*** point to the first nodes successor in the list
3. Delete the node pointed by temporary pointer.

```
temp = head;
head = head -> next;
free(temp);
```

b. Delete Middle Node

1. Store the node to be deleted in a temporary pointer.
2. Make the predecessor node point to the successor of the node being deleted.
3. Delete the node pointed by temporary pointer.

```
ptr  -> next = temp -> next;
free (temp);
```

c. Delete Last Node

1. Store the node to be deleted in a temporary pointer
2. Move the null pointer to the predecessor ***next*** filed of last node.
3. Delete the node pointed by temporary pointer

```
temp = ptr -> next
ptr -> next = NULL;
free(temp);
```

SUMMARY

- Array is a fundamental data structure and many other data structure are implemented using this. Implementing data structures such as stacks and queues
- Representing data in tables and matrices
- Creating dynamic data structures such as Hash Tables and Graph.
- When compared to other data structures, arrays have the advantages like random access (we can quickly access i-th item) and cache friendliness (all items are stored at contiguous location).

1. Identify the two types of efficiencies that are important for computer algorithms?

a) Time efficiency and Higher power efficiency
b) Higher power efficiency and Computational efficiency
c) Computational efficiency and Space efficiency
d) Space efficiency and Time Efficiency

2.Which one of the following asymptotic notation is used to provide lower bound constraint?

a) X
b) Ω
c) Θ
d) O

TERMINAL QUESTIONS

1. How is time complexity measured?

2. Which of the following best describes the useful criterion for comparing the efficiency of algorithms?

3. What does it mean to say that an algorithm X is asymptotically more efficient than Y?

4. Describe the various types of Asymptotic Notations with examples?

REFERENCES FOR FURTHER LEARNING OF THE SESSION

Reference Books :

1 Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein., 3rd, 2009, The MIT Press.
2 Algorithm Design Manual, Steven S. Skiena., 2nd, 2008, Springer.
3 Data Structures and Algorithms in Python, Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser., 2nd, 2013, Wiley.
4 The Art of Computer Programming, Donald E. Knuth, 3rd, 1997, Addison-Wesley Professiona.

Sites and Web links:

1. https://nptel.ac.in/courses/106102064
2. https://nptel.ac.in/courses/106101060/4
3. https://www.edx.org/course/algorithms-and-data-structures-1
4. https://in.udacity.com/course/intro-to-algorithms--cs215
5. https://www.coursera.org/learn/data-structures?action=enroll
6. https://www.geeksforgeeks.org/types-of-asymptotic-notations-in-complexity-analysis-of-algorithms/

THANK YOU