SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – I-Data Structures – SCSA1203

UNIT I

INTRODUCTION TO ALGORITHMS **9 Hrs.**

Introduction Data Structures - Need - classification - operations –Abstract data types (ADT) - Array - characteristics - types - storage representations. Array Order Reversal-Array Counting or Histogram-Finding the maximum Number in a Set, Recursion- Towers of Hanoi-Fibonacci series-Factorial.

I. ALGORITHM

An algorithm is a well-defined computational procedure having well defined steps for solving a particular problem. Algorithm is finite set oflogic or instructions, written in order for accomplishing the certain predefined task.

Each algorithm must have:

- **Specification:** Description of the computational procedure.
- **Pre-conditions:** The condition(s) on input.
- **Body of the Algorithm:** A sequence of clear and unambiguous instructions.
- **Post-conditions:** The condition(s) on output.

Characteristics of an Algorithm

An algorithm must follow the mentioned below characteristics:

- **Input:** An algorithm must have 0 or well-defined inputs.
- **Output:** An algorithm must have 1 or well-defined outputs, and should match with the desired output.
- **Feasibility:** An algorithm must be terminated after the finite number of steps.
- **Independent:** An algorithm must have step-by-step directions which is independent of any programming code.
- **Unambiguous:** An algorithm must be unambiguous and clear. Each of their steps and input/outputs must be clear and lead to only one meaning.

  The performance of algorithm is measured on the basis of following properties:
- **Time complexity:** It is a way of representing the amount of time needed by a program to run to the completion.
- **Space complexity:** It is the amount of memory space required byan algorithm, during a course of its execution. Space complexity is required in situations when

limited memory is available and for the multi user system.

2. INTRODUCTION TO DATA STRUCTURES

2.1 Data

In the modern context, data stands for both singular and plural. Data means a value or set of values. Data can be defined as an elementaryvalue or the collection of values, for example, student's name and its idare the data about the student.

2.2 Structure

A building is a structure. A bridge is structure. In general, a structureis made up of components. It has a form or shape. It is made up of parts.A structure is an arrangement of and relations between parts or elements.

2.3 Data Structures

A data structure is an arrangement of data elements. Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspectof Computer Science i.e. Operating System, Compiler Design, Artificial Intelligence, Graphics and many more.

2.3.1 Needs

As applications are getting complex and amount of data is increasing day by day, the following issues might be araised:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day tothe billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store;if our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously in a web server, it fails to process the requests.

In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items arenot required to be searched and required data can be searched instantly.

Data structures are important for the following reasons

1. Data structures are used in almost every program or software system.
2. Specific data structures are essential ingredients of many efficient algorithms, and make possible the management of huge amounts of data, such as large integrated collection of databases.
3. Some programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

Advantages of Data Structures

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. element. Hence, using array may not be very efficient here.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place.

Abstraction: Data structure is specified by the ADT which provides alevel of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

3.Classification of Data Structure

The data structure is classified into two different types, primitive and non-primitive data structures is shown in Fig.1.

Primitive Data Structures

Simple data structure can be constructed with the help of primitivedata structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, pointers, structures, unions, etc. are examples of primitive data structures.

Non Primitive Data Structures

Non Primitive Data Structures are classified as linear or non-linear. Arrays, linked lists, queues and stacks are linear data structures. Trees and Graphs are non-linear data structures. Except arrays, all other data structures have many variations. Non Primitive data structure can be constructed with the help of any one of the primitive data structure andit is having a specific functionality. It can be designed by user.

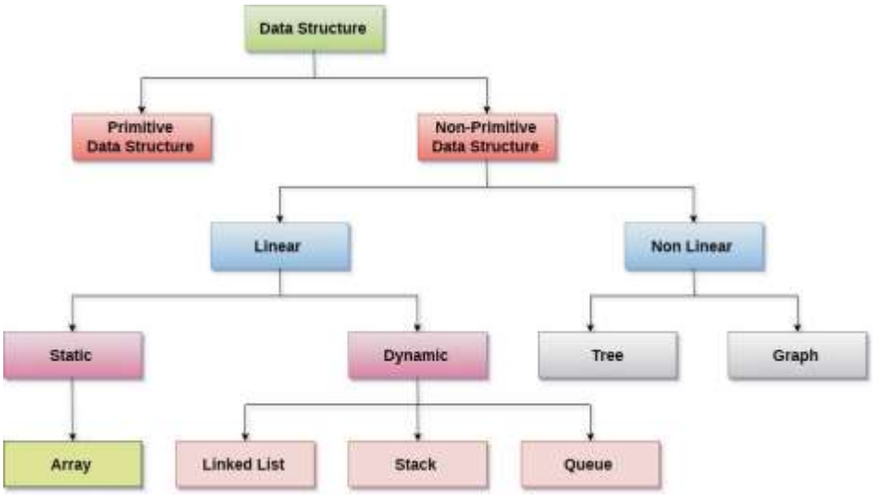


Fig. 1 Classification of Data Structure

Linear Data Structures

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element. In linear data structure the elements are stored in sequential order.

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items stored in consecutive memory location and is referred by common name; each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double. The elements of array share the same variable name but each onecarries a different index number known as subscript.

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node. It is a collection of data of same data type but the data items need not be stored in consecutive.

Stack: Stack is a linear list in which insertion and deletions are allowedonly at one end, called top. It is a Last-In-First-Out linear data structure.

A stack is an abstract data type (ADT), can be implemented in mostof the programming languages. It is named as stack because it behaveslike a real-world stack, for example, piles

of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only atone end called rear and deleted only at the other end called front. It isa First-In-First-Out Linear data structure.

It is an abstract data structure, similar to stack. Queue is opened atboth end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Operations applied on Linear Data Structure

The following list of operations applied on linear data structures

- Add an element
- Delete an element
- Traverse
- Sort the list of elements
- Search for a data element

Non-linear Data Structures

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement.The data elements are not arranged in sequential structure. Elements are stored based on the hierarchical relationship among the data. The following are some of the Non-Linear data structures.

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottom most nodes are called leaf node while the top most node is called root node. Each node contains pointers to point adjacent nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

Operations applied on non-linear data structures

The following list of operations applied on non-linear data structures.

1. Add elements
2. Delete elements

3. Display the elements

4. Sort the list of elements

5. Search for a data element

4. ABSTRACT DATA TYPES

Abstract Data Type (ADT) is a type or class for objects whose behaviour is defined by a set of value and a set of operations.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. The process of providing only the essentials and hiding the details is known as abstraction is shown in fig,2.

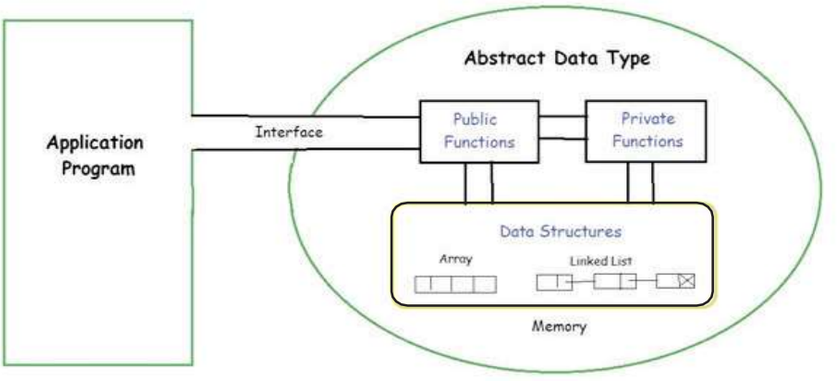


Fig. 2 Abstract Data Type

4.1 List ADT

* The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list is shown in fig.3.

* The data node contains the pointer to a data structure and a self-referential pointer which points to the next node in the list.

```
//List ADT Type Definitions
     typedef struct node{
 void *DataPtr;
struct node *link;
    } Node;
typedef struct{
   int count;
  Node *pos;
 Node *head;
  Node *rear;
      int (*compare) (void *argument1, void *argument2);
                                } LIST;
```

Fig.3 List ADT

✱ A list contains elements of the same type arranged in sequential order and following operations can be performed on the list.

- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from anon-empty list.
- removeAt() – Remove the element at a specified location from anon-empty list.
- replace() – Replace an element at any position by anotherelement.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise returnfalse.
- isFull() – Return true if the list is full, otherwise return false.

The List ADT Functions is shown in Fig.4.

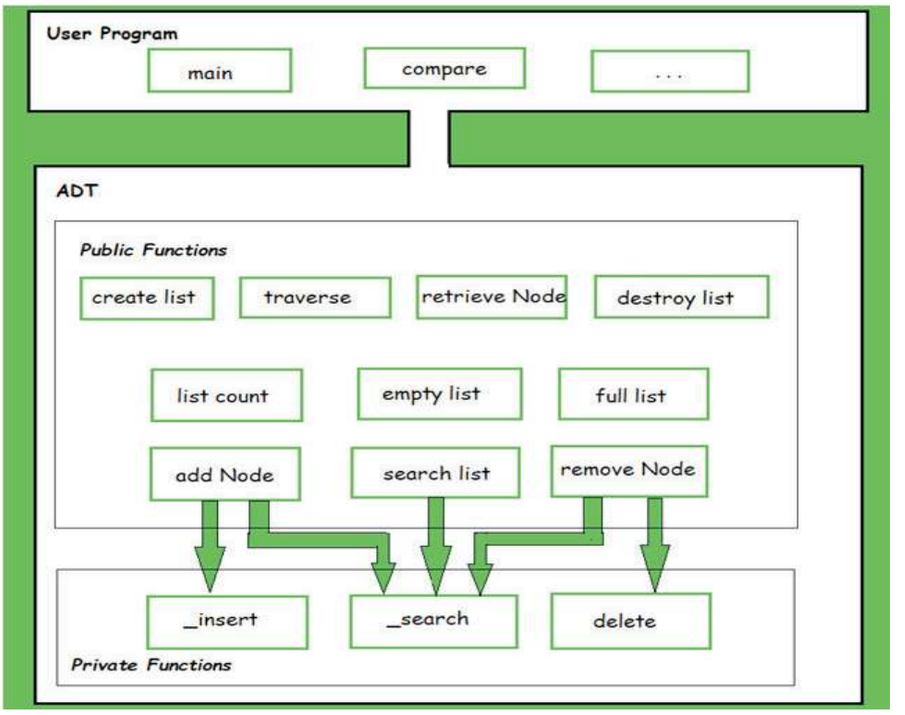


Fig. 4 List ADT Functions

4.2 Stack ADT

- In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the data and address is passed to the stack ADT is shown in Fig.5.
- The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to top and countof number of entries currently in stack.
- A Stack contains elements of the same type arranged in sequential order.

```
//Stack ADT Type Definitions
typedefstruct node{
void *DataPtr;
struct node *link;
} StackNode;
typedefstruct{
int count;
StackNode *top;
} STACK;
```

Fig. 5 Stack ADT

All operations take place at a single end that is top of the stackand following operations can be performed:

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack,if it is not empty.
- peek() – Return the element at the top of the stack withoutremoving it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise returnfalse.
- isFull() – Return true if the stack is full, otherwise return false.

4.3 Queue ADT

✱ The queue Abstract Data Type (ADT) follows the basic design of the stack abstract data type is shown in fig.6.

```
//Queue ADT Type Definitions
typedefstructnode {
void*DataPtr;
 structnode *next;
} QueueNode;
typedefstruct {
QueueNode *front;
QueueNode *rear;
intcount;
} QUEUE;
```

Fig 6. Queue ADT

✱ A Queue contains elements of the same type arranged in sequential order. Operations take place at both ends, insertion is done at theend and deletion is done at the front. Following operations can be performed:

- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue,if the queue is not empty.
- peek() – Return the element of the queue without removing it,if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise returnfalse.
- isFull() – Return true if the queue is full, otherwise return false.

6.ARRAYS

The number of data items chunked together into a unit is known as data structure. When the data structure consists simply a sequence of data items, the data structure of this simple variety is referred as an array.

Definition: Array is a collection of homogenous (same data type) dataelements that are stored in contiguous memory locations.

Array Syntax

Syntax to declare an array:

✱ dataType [ ] arrayName;

arrayName= new dataType[n]; //keyword new performs dynamicmemory location

(or)

✱ dataType [ ] arrayName = new dataType[n];

Example:

int [ ] x; x=new int [10];(or)

int [] x=new int [10];

Array Initialization

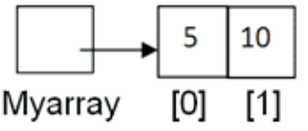
The values of an array can be initialized as follows,

Example 1:

int [] j=new int [1]; j[0] =10;**(or)**int [] j= {25};

Example 2:

int [] myarray= {5, 10};



5.1 Characteristics of an Array

The following are the characteristics of an array data structure.

(i) Array stores elements of same data type.

(ii) The elements of an array are stored in subsequent memorylocations.

(iii) The name of array represents the starting address of the elements.

(iv) When data objects are stored in array, individual objects areselected by an index.

(v) By default an array index starts from 0 and ends with (n-1). Indexis also referred

as subscripts.

(vi) The size of the array is mentioned at the time of declaring array.

(vii) While declaring 2D array, number of columns should be specifiedwhereas for number of rows there is no such rule.

(viii) Size of array can't be changed at run time.

5.2 Array Types

1.One-Dimensional Array or Linear arrays

2. Multi-Dimensional Array

3.Two dimensional (2D) Arrays or Matrix Arrays

5.2.1 One-Dimensional Array

In one dimensional array each element is represented by a single subscript. The elements are stored in consecutive memory locations. E.g.A [1], A [2], ..., A [N].

5.2.2 Two dimensional (2-D) arrays or Matrix Arrays

In two dimensional arrays each element is represented by two subscripts. Thus a two dimensional m x n array has m rows and n columns and contains m * n elements. It is also called matrix array because in this case, the elements form a matrix. For example A [4] [3]has 4 rows and 3 columns and 4*3 = 12 elements.

```
int [] [] A = new int [4] [3];
```

5.2.3 Multi dimensional arrays:

In it each element is represented by three subscripts. Thus a three dimensional m x n x l array contains m * n * l elements. For exampleA [2] [4] [3] has 2 * 4 * 3 = 24 elements.

6.STORAGE REPRESENTATION

An array is a set of homogeneous elements. Every element is referredby an index. Memory storage locations of the elements are not arrangedas a rectangular array with rows and columns. Instead, they are arranged in a linear sequence beginning with location 1, 2, 3 and so on. The elements are stored either column-by-column or row-by-row. The first one is called column-major order and later is referred as row-major order.

6.1 Row Major Order

The table 1 shows the linear arrangement of data in row major order.

Example

- Rows : 3
- Columns : 4

Data (A):

1	2	3	4
5	6	7	8
9	10	11	12

Table 1 Linear Arrangement of Array A in Row Major Order

The formula is:

Location (A [j,k]) = Base Address of (A) + w [(N * (j-1)) + (k-1)]

Location (A [j, k]): Location of j^{th} row and k^{th} column

Base (A) : Base Address of the Array A

Linear Arrangement of Array A

Row	**1**				**2**				**3**			
Index	**(1,1)**	**(1,2)**	**(1,3)**	**(1,4)**	**(2,1)**	**(2,2)**	**(2,3)**	**(2,4)**	**(3,1)**	**(3,2)**	**(3,3)**	**(3,4)**
Memory	**100**	**102**	**104**	**106**	**108**	**110**	**112**	**114**	**116**	**118**	**120**	**122**
Data	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

w : Bytes required to represent single element of the Array A

N : Number of columns in the Array

j : Row position of the element

k : Column position of the element

Example

Suppose to find the address of (3,2) then

Base (A) = 100

w = 2 Bytes (integer type)

N = 4

j = 3

k = 2

Location (A [3, 2]) = 100 + 2 [(4 * (3-1) + (2-1)]

= 118

6.2 Column Major Order

The table 2 shows the linear arrangement of data in column major order.

- Rows : 3
- Columns : 4

Linear Arrangement of Array A												
Column	**1**			**2**			**3**			**4**		
Index	**(1,1)**	**(2,1)**	**(3,1)**	**(1,2)**	**(2,2)**	**(3,2)**	**(1,3)**	**(2,3)**	**(3,3)**	**(1,4)**	**(2,4)**	**(3,4)**
Memory	**100**	**102**	**104**	**106**	**108**	**110**	**112**	**114**	**116**	**118**	**120**	**122**
Data	**1**	**5**	**9**	**2**	**6**	**10**	**3**	**7**	**11**	**4**	**8**	**12**

Table 2 Linear Arrangement of Array A in Column Major Order

The formula for column major order is:

Location (A [j, k]) = Base Address of (A) + w [(M * (k-1)) + (j-1)]

Location (A [j, k]): Location of j^{th} row and k^{th} column

Base (A) : Base Address of the Array A

w : Bytes required to represent single element of the Array A

M : Number of rows in the Array

j : Row position of the element

k : Column position of the element

Example

Base (A) = 100
w = 2 Bytes (integer type)
M = 3
j = 3
k = 2
Location (A [3, 2]) = 100 + 2 [(3 * (2-1) + (3-1)]
= 110

7. Array Order Reversal

Given an array (or string), the task is to reverse the array/string.

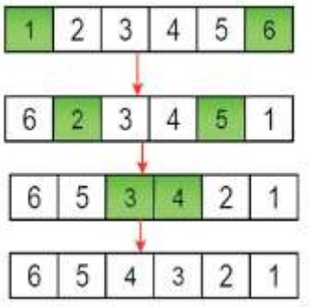
Examples:

Input : arr[] = {1, 2, 3} Output : arr[] = {3, 2, 1}

Input : arr[] = {4, 5, 1, 2} Output : arr[] = {2, 1, 5, 4}

Algorithm

1) Initialize start and end indexes as start = 0, end = n-1.
2) In a loop, swap arr[start] with arr[end] and change start and end as follows : start = start +1, end = end – 1



8.Array Counting

1. Create a function arraycounting(array, size)

2. Find largest element in array and store it in max

3. Initialize count array with all zeros

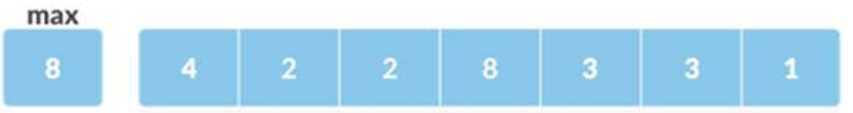
4. for j = 0 to size

5. Find the total count of each unique element and

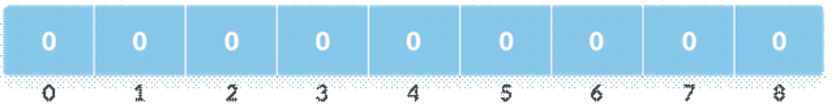
6. Store the count at jth index in count array

Example A={4,2,2,8,3,3,1}

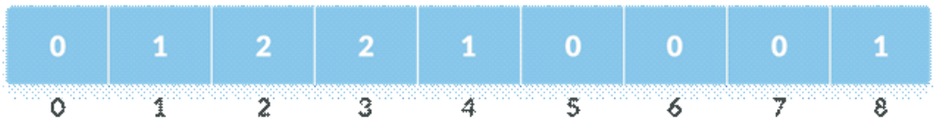
1. Find out the maximum element (let it be max) from the given array.



2. Initialize an array of length max+1 with all elements 0. This arrayis used for storing the count of the elements in the array.



3. Store the count of each element at their respective index in count array. For example: If the count of element "4" occurs 2 times then2 is stored in the 4^{th} position in the count array. If element "5" isnot present in the array, then 0 is stored in 5^{th}



8. Finding the maximum Number in a Set

Algorithm

✱ Read the array elements

- ✱ Initialize first element of the array as max.
- ✱ Traverse array elements from second and compare every elementwith current max
- ✱ Find the largest element in the array and assign it as max
- ✱ Print the largest element.

Example: A={56,78,34,23,70}

Step 1: Initialize max=0 , n=len(A)
Step 2: Repeat step 3 until n
Step 3: 56>0 yes, Assign max=56
78>56, yes Assign max=78
34>78, No
23>78, No
70>78, No
Step 4: print Max Output:78

9. RECURSION

Recursion is a programming technique using function or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one calledto the first.

A simple recursive algorithm:

- ✱ Solves the base cases directly.
- ✱ Recurs with a simpler sub problem or sub problems.
- ✱ May do some extra work to convert the solution to the simpler subproblem into a solution to the given problem.

Some Example Algorithms

1. Factorial
2. All permutations
3. Tree traversal
4. Binary Search
5. Quick Sort
6. Towers of Hanoi

9.1 Design Methodology and Implementation of Recursion

- The recursive solution for a problem involves a two-way journey:
- First we decompose the problem from the top to the bottom
- Then we solve the problem from the bottom to the top.

Steps for Designing Recursive Algorithms

- Each call of a recursive algorithm either solves one part of theproblem or it reduces the size of the problem.
- The general part of the solution is the recursive call. At eachrecursive call, the size of the problem is reduced.
- The statement that ôsolvesö the problem is known as the base case.
- Every recursive algorithm must have a base case.
- The rest of the algorithm is known as the general case. The generalcase contains the logic needed to reduce the size of the problem.
- Once the base case has been reached, the solution begins.We now know one part of the answer and can return that part tothe next, more general statement.
- This allows us to solve the next general case.
- As we solve each general case in turn, we are able to solve the next-higher general case until we finally solve the most general case, the original problem.

Rules for Designing a Recursive Algorithm

1. First, determine the base case.
2. Then determine the general case.
3. Combine the base case and the general cases into an algorithm.
4. Each recursive call must reduce the size of the problem and moveit toward the base case.
5. The base case, when reached, must terminate without a call to therecursive algorithm; that is, it must execute a return.

9.2 Broad Categories of Recursion

Recursion is a technique that is useful for defining relationships, andfor designing algorithms that implement those relationships. It is a natural way to express many algorithms

in an optimized way. Recursive function is defined in terms of itself.

- Linear Recursion
- Binary Recursion

Linear Recursion:

Linear recursion is by far the most common form of recursion. In this style of recursion, the function calls itself repeatedly until it hits the termination condition (Base condition).

Binary Recursion

Binary recursion is another popular and powerful method. This formof recursion has the potential for calling itself twice instead of once asbefore. This is pretty useful in scenarios such as binary tree traversal,generating a Fibonacci sequence, etc.

Tail Recursion

A function call is said to be tail recursive if there is nothing to doafter the function returns except return its value. Since the current recursive instance is done executing at that point, saving its stack frameis a waste

For example the following C function print () is tail recursive.

```
// An example of tail recursive functionPrint (n) {
If (n < 0) return;

Display n;

Print (n-1);

}
```

10. FIBONACCI SERIES

Fibonacci Series generates subsequent number by adding two previous numbers. Fibonacci series starts from two numbers F0 & F1. The initial values of F0 & F1 can be

- 0 and 1

✱ 1 and 1 respectively.

The Fibonacci series looks like
F8 = 0 1 1 2 3 5 8 13

The algorithm for generating Fibonacci series can be drafted in 2 ways

1. Fibonacci Iterative Algorithm.

2. Fibonacci Recursive Algorithm.

Fibonacci RecursiveAlgorithm

Algorithm Fibo (n)
If n = 0
Return 0Else If n = 1
Return 1
Else
Fibo (n) = Fibo (n-1) + Fibo (n-2)Return Fibo (n)

11.FACTORIAL

The factorial of a positive number is the product of the integral values from 1 to the number: Factorial of the given number can be calculated as:

Algorithm

RecursiveFactorial (n)if (N equals 0)
Return 1
else
Return (n*recursiveFactorial (n-1))
end if
end recursiveFactorial

Calling a Recursive Factorial Algorithm with n=3

Fig.6 shows the steps for calculating the factorial using Recursion for n=3.

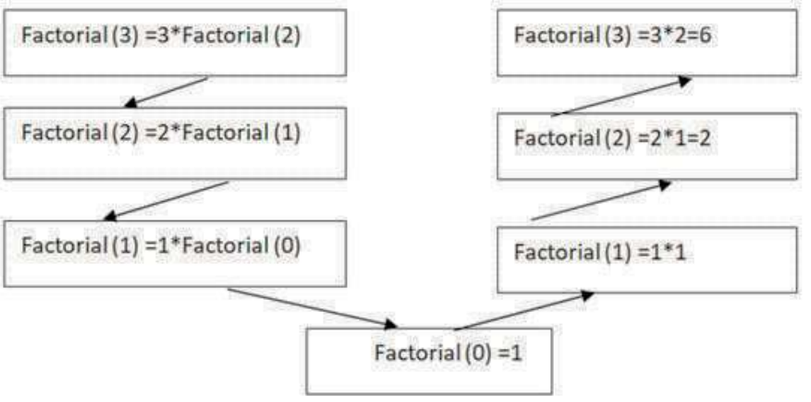


Fig 6. Factorial using Recursion Steps

Output: 6

12. TOWERS OF HANOI

Tower of Hanoi is a mathematical puzzle which consists of three tower (pegs) and more than one ring; as depicted in Fig.7.

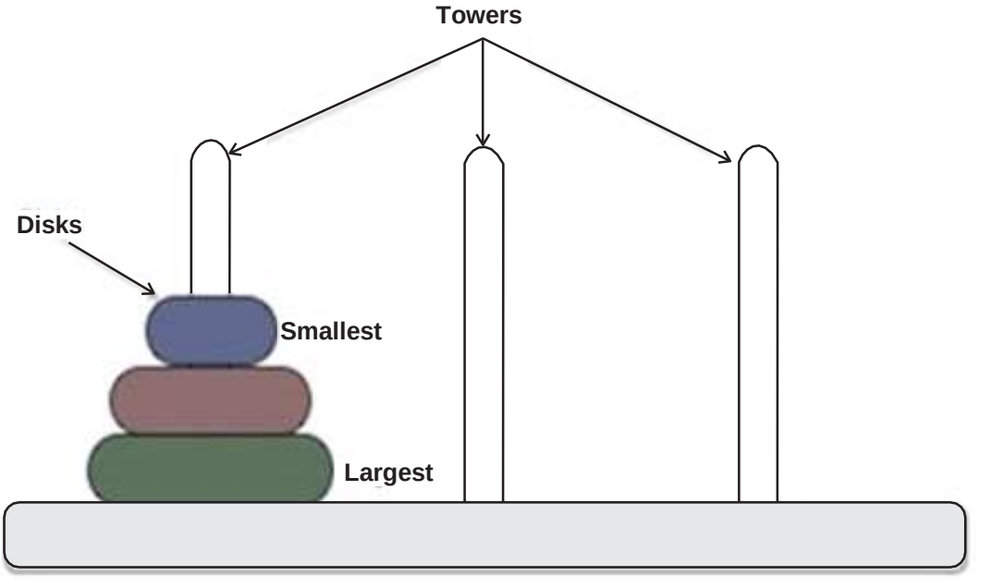


Fig.7 Tower of Hanoi

These rings are of different sizes and stacked upon in ascending order i.e. the smaller one sits over the larger one. There are other variationsof puzzle where the number of disks increases, but the tower count remains the same.

7.1 Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. The below mentioned are few rules which are to be followed for Tower of Hanoi

- Only one disk can be moved among the towers at any given time.
- Only the “top” disk can be removed.
- No large disk can sit over a small disk.

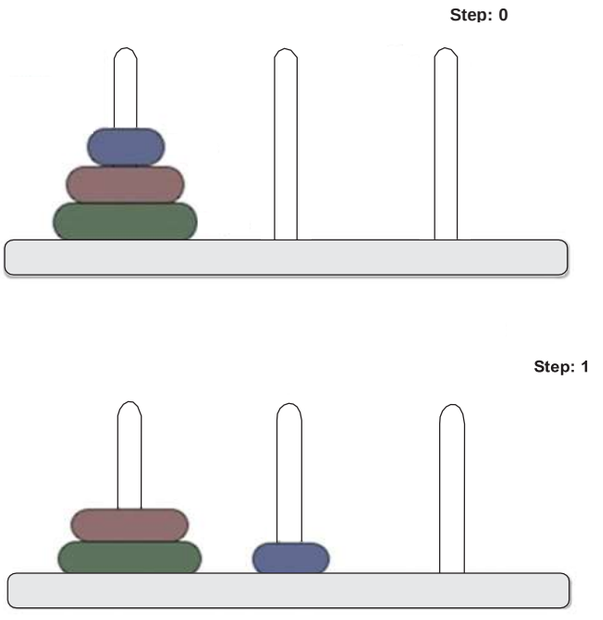
Steps for solving the Towers of Hanoi problem

The following steps are to be followed.

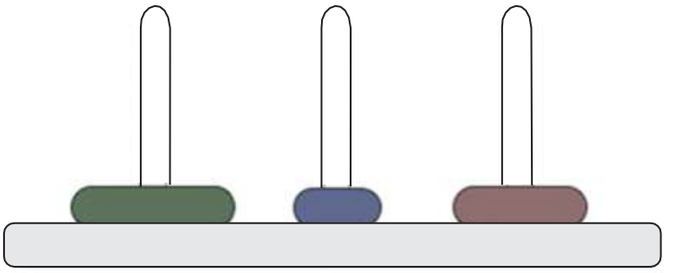
Step 1: Move n-1 disks from source to aux.

Step 2: Move nth disk from source to destinationStep 3: Move n-1 disks from aux to destination.

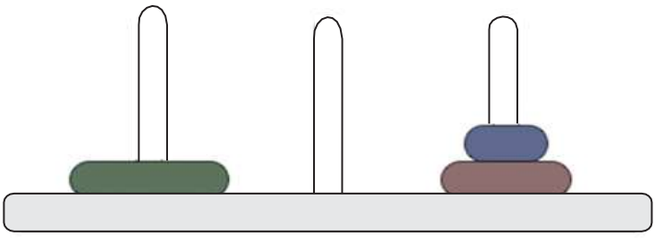
Following Fig.8 illustrates the step by step movement of the disks to implement Tower of Hanoi.



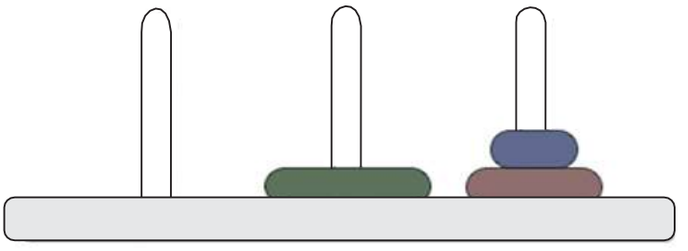
Step: 2



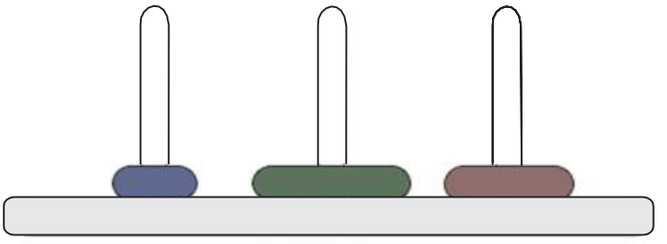
Step: 3



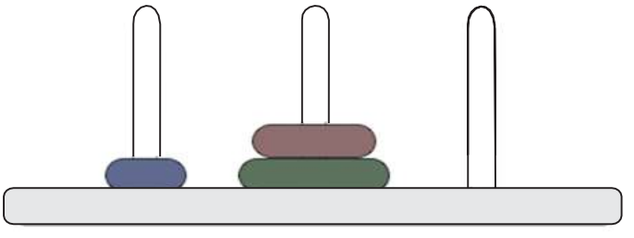
Step: 4



Step: 5



Step: 6



Done!

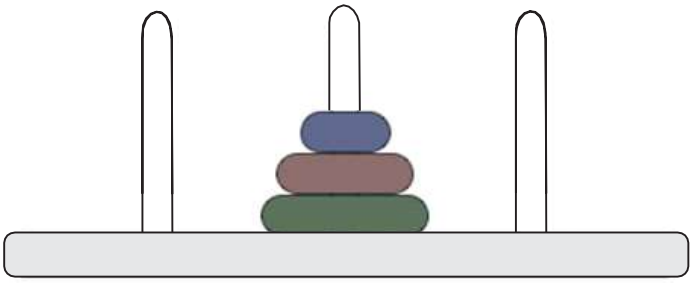


Fig.8 Tower of Hanoi

A recursive algorithm for Tower of Hanoi can be driven as followsSTART

Procedure **Hanoi** (disk, source, dest, aux)

IF disk = 0, THEN

Move disk from source to destELSE

Hanoi (disk-1, source, aux, dest) //Step1Move disk from source to dest //Step2 Hanoi (disk-1, aux, dest, source) //Step3 ENDIF

END

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – II-Data Structures – SCSA1203

UNIT –II

LINKED LISTS

Introduction - Singly linked list - Representation of a linked list in memory - Operations on a singly linked list - Merging two singly linked lists into one list - Reversing a singly linked list - Applications of singly linked list to represent polynomial - Advantages and disadvantages of singly linked list - Circular linked list - Doubly linked list - Circular Doubly Linked List.

INTRODUCTION

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collection of similar elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast. But arrays are suffer from the following limitations:

- Arrays have a fixed dimension. Once the size of an array is decided it cannot be increased or decreased during execution. For example, if we construct an array of 100 elements and then try to stuff more than 100 elements in it, our program may crash. On the other hand, if we use only 10 elements then the space for balance 90 elements goes waste.
- Array elements are always stored in contiguous memory locations. At times it might so happen that enough contiguous locations might not be available for the array that we are trying to create. Even though the total space requirement of the array can be met through a combination of non-contiguous blocks of memory, we would still not be allowed to create the array.
- Operations like insertion of a new element in an array or deletion of an existing element from the array are pretty tedious. This is because during insertion or deletion each element after the specified position has to be shifted one position to the right (in case of insertion) or one position to the left (in case of deletion).

Linked list overcomes all these disadvantages. A linked list can grow and shrink in size during its lifetime. In other words, there is no maximum size of a linked list. The second advantage of linked lists is that, as node (elements) are stored at different memory locations it hardly happens that we fall short of memory when required. The third advantage is that, unlike arrays, while inserting or deleting the nodes of the linked list, shifting of nodes is not required.

What is a Linked list?

Linked list is a very common data structure often used to store similar data in memory. While the elements of an array occupy contiguous memory locations, those of a linked list are not constrained to be stored in adjacent

locations. The individual elements are stored “somewhere” in memory, rather like a family dispersed, but still bound together. The order of the elements is maintained by explicit links between them. **-

- Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.
- A node contains two fields i.e. data and pointer which contains the address of the next node in the memory.
- Linked list requires more memory compared to array because along with value it stores pointer to next node.
- Linked list are among the simplest and most common data structures. They can be used to implement other data structures like stacks,queues and symbolic expressions,etc…
- The last node of the list contains pointer to the null.
- Typically, a linked list, in its simplest form looks like the following

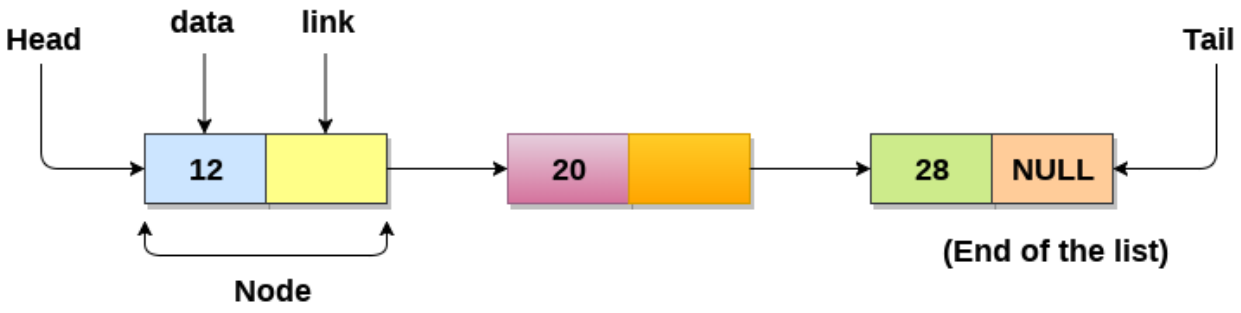


Fig:1 Linked List

Few salient features

- There is a pointer (called header) points the first element (also called node)
- Successive nodes are connected by pointers.
- Last element points to NULL.
- Linked lists have efficient memory utilization. Here, memory is not pre allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
- Linked lists are dynamic data structures. i.e., It can grow or shrink in size during execution of a program.
- Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
- Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node when compared to array.

2. Searching a particular element in list is difficult and also time consuming.

Defining a Node of a Linked List

Each structure of the list is called a node, and consists of two fields:

- Item (or) data
- Address or pointer to the next node in the list

How to define a node of a linked list?

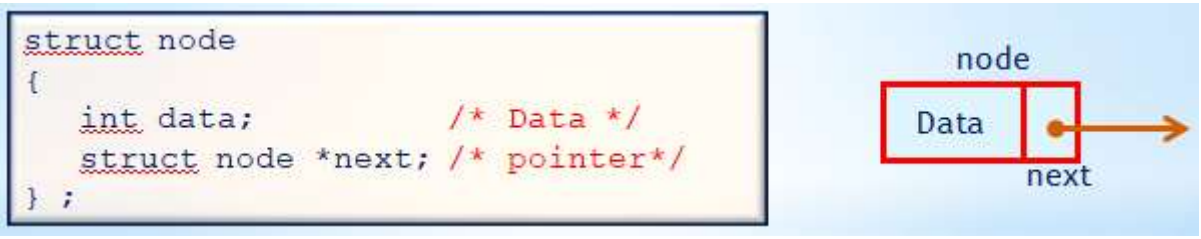


Fig: 2 Defining a node

Note:

Such structures which contain a member field pointing to the same structure type are called self-referential structures.

Initialize the head node

Struct node *head=NULL;

Memory allocation for a node

Struct node *newnode=(struct node *)malloc(sizeof(struct node));

Creation of node

Steps to create node:

1. Defining a structure of a node
2. Allocate memory dynamically.
3. Read the data into data field
4. Assign null value to its next

struct node

{

```
int data;
struct node *next;      //defining node structure
 };
struct node *head=null,*newnode; //initialize the head node
newnode=(struct node *)malloc(sizeof(struct node)) // memory allocation for a newnode
newnode->data =data // Read data
newnode-> next =null;
```

There are 3 different implementations of Linked List available, they are:

- Singly Linked List
- Doubly Linked List
- Circular Linked List

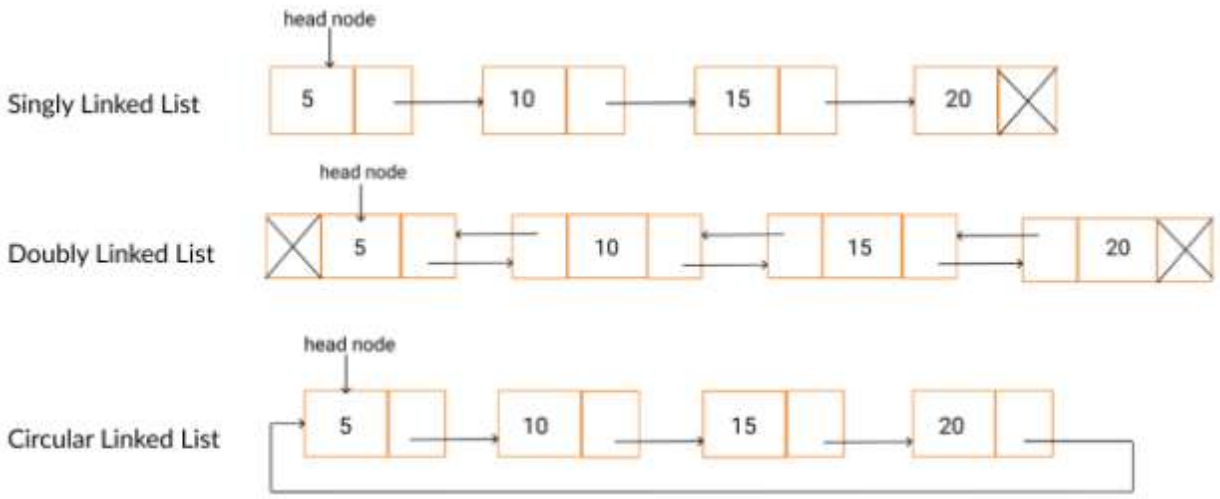


Fig: 3 Types of linked list

Representation of Linked List in memory

Linked lists can be represented in memory by using two arrays respectively known as data and next contains information of element and address of next node repectively.

The list also requires a variable name or start which contains address of first node. Pointer field of last node denoted by NULL which indicates the end of list. Eg. Consider a linked list given below. The linked list can be represented in memory as

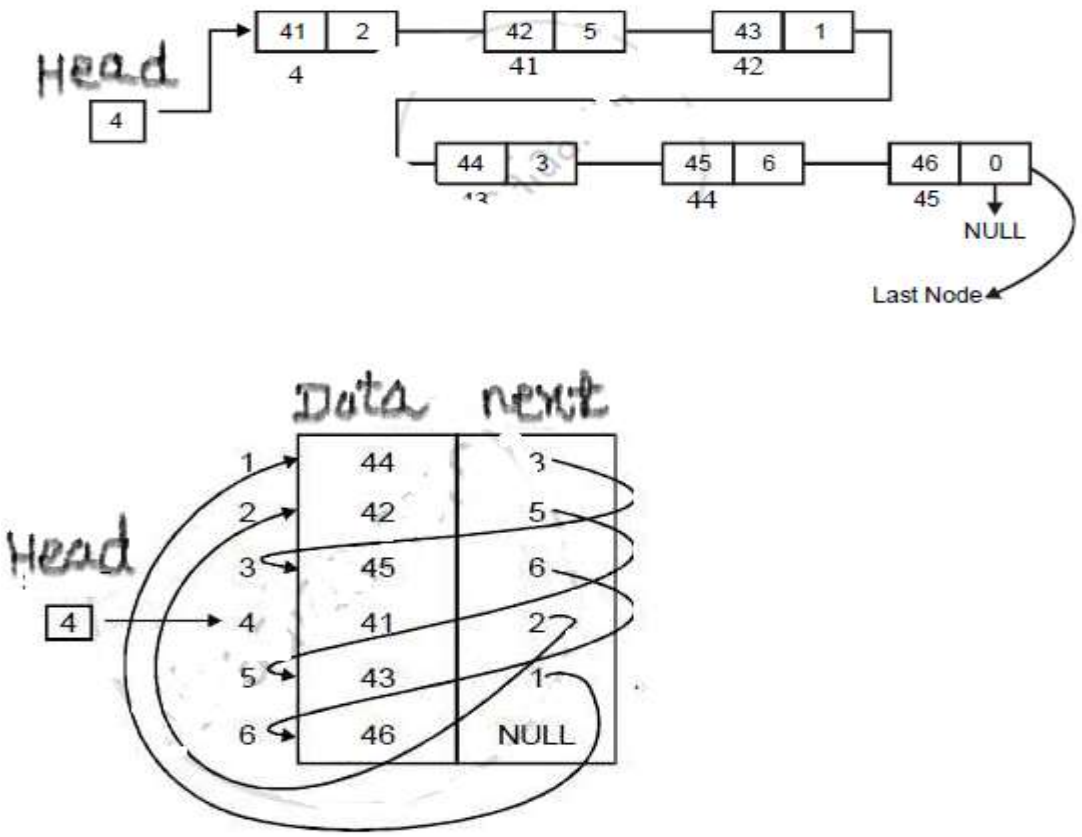


Fig 4: Representation of Linked List

Singly linked list

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we cannot traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.

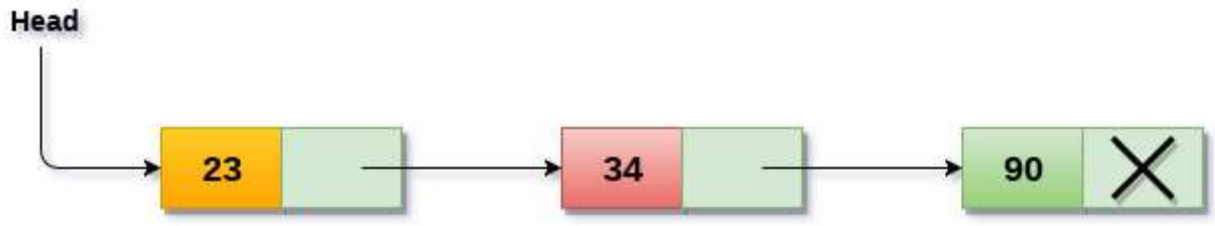


Fig:5 Singly Linked List

In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

- Traversing the list
- Inserting a node into the list
- Deleting a node from the list
- Merging the linked list with another one to make a larger list
- Searching for an element in the list.

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Insertion operation in singly linked list

Inserting a node into a single linked list There are various positions where a node can be inserted:

(i) Inserting at the front (as a first element)

(ii) Inserting at the end (as a last element)

(iii) Inserting at any other position.

Inserting a node at the front of a single linked list

Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links adjustments to make the new node as the first node of the list. There are the following steps which need to be followed in order to insert a new node in the list at beginning.

- Allocate the space for the new node and store data into the data part of the node.
- Make the link part of the new node pointing to the existing first node of the list.
- At the last, we need to make the new node as the first node of the list.

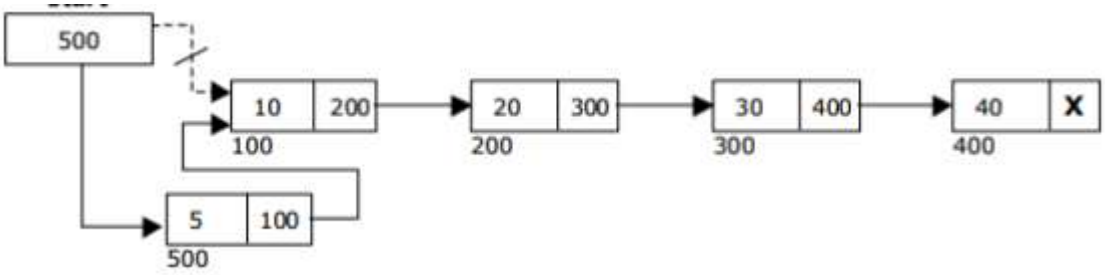


Fig:5 Inserting a node at the front of a single linked list

The algorithm lnsertatfront: is used to insert a node at the front of a single linked list.

Algorithm INSERT_FRONT(head,X)

Input: Head is the pointer to the first node and X is the data of the node to be inserted.

Output: A singly linked list with newly inserted node in the front of the list.

1. newnode = create newnode // Create a newnode and store its pointer in newnode
   newnode->data =data // Read data
   newnode-> next =null;
2. if(head!= NULL)
   newnode->next=head
   head=newnode
3. endif
4. stop

Inserting a node at the end of a single linked list

- We need to declare a temporary pointer temp in order to traverse through the list. temp is made to point the first node of the list. At the end of the loop, the temp will be pointing to the last node of the list. Now, allocate the space for the new node, and assign the item to its data part. Since, the new node is going to be the last node of the list hence, the next part of this node needs to be pointing to the null. We need to make the next part of the temp node (which is currently the last node of the list) to null .

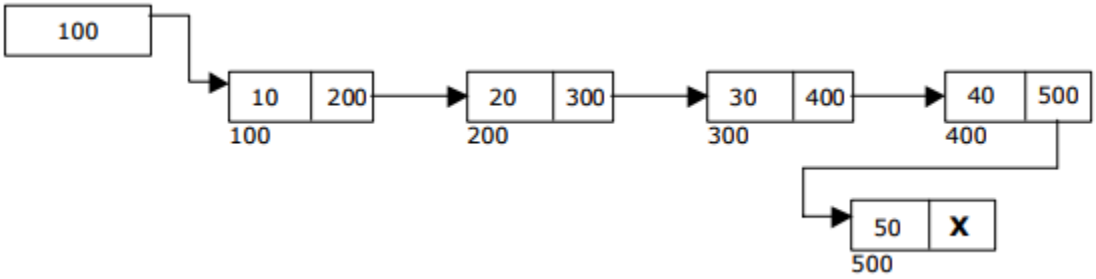


Fig:6 Inserting a node at the end of a single linked list

The algorithm Insert_End is used to insert a node at the end of a single linked list

Algorithm INSERT_END(head,X)

Input: Head is the pointer to the first node and X is the data of the node to be inserted.

Output: A singly linked list with newly inserted node at the end of a linked list

1. newnode = create newnode // Create a newnode and store its pointer in newnode
   newnode->data =data // Read data
   newnode-> next =null
2. temp = head

   while (temp -> next != NULL) do

   temp = temp -> next

   Endwhile

   temp->next = newnode

   newnode->next=null

   stop

Inserting a node into a single linked list at any position in the list

In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted.

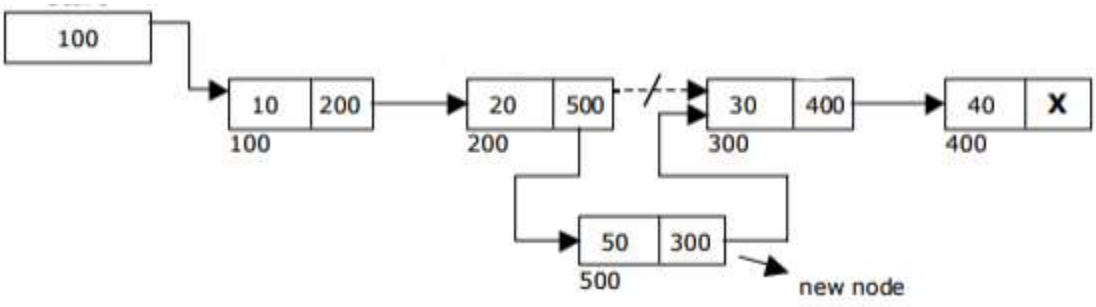


Fig: 7 Inserting a node into a single linked list at any position in the list

The algorithm InsertAnyPosition is used to insert a node into a single linked list at any position in the list.

Algorithm INSERT_ANYPOSITION(Head,X,pos)

1. temp=head;

   while(i<pos)

   temp=temp->next;

   i++

   Endwhile

2. newnode = create newnode // Create a newnode and store its pointer in newnode

   newnode->data =data // Read data

   newnode->next=temp->next

   temp->next=newnode

Deletion Operation in Singly linked list

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description

1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.

Deletion in singly linked list at beginning

Deleting a node from the beginning of the list is the simplest operation of all. It just need a few adjustments in the node pointers. Since the first node of the list is to be deleted, therefore, we just need to make the head, point to the next of the head. This will be done by using the following statements.

temp=head;

head=head->next;

Now, free the pointer ptr which was pointing to the head node of the list. This will be done by using the following statement.

Free(temp)

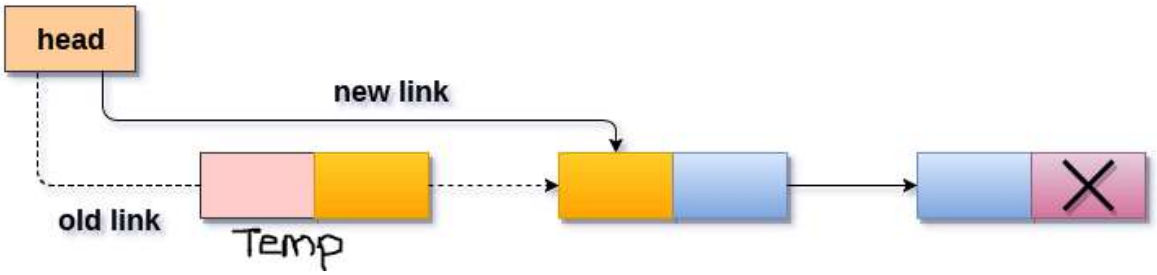


Fig 8: Deleting a node from the beginning

Algorithm DELETE_FRONT(Head,temp)

- **Step 1:** IF HEAD = NULL

  Write UNDERFLOW

  Go to Step 5

  [END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** SET HEAD = HEAD -> NEXT
- **Step 4:** FREE TEMP
- **Step 5:** EXIT

Deletion in singly linked list at the end

There are two scenarios in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.

In the first scenario,

the condition head → next = NULL will survive and therefore, the only node head of the list will be assigned to null. This will be done by using the following statements.

temp = head

head = NULL

free(ptr)

In the second scenario,

The condition head → next = NULL would fail and therefore, we have to traverse the node in order to reach the last node of the list.

For this purpose, just declare a temporary pointer temp and assign it to head of the list. We also need to keep track of the second last node of the list. For this purpose, two pointers temp and ptr1 will be used where temp will point to the last node and ptr1 will point to the second last node of the list.

this all will be done by using the following statements.

```
temp = head;
while(temp->next != NULL)
{
   ptr1 = temp;
   temp = temp ->next;
}
```

Now, we just need to make the pointer ptr1 point to the NULL and the last node of the list that is pointed by temp will become free. It will be done by using the following statements.

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

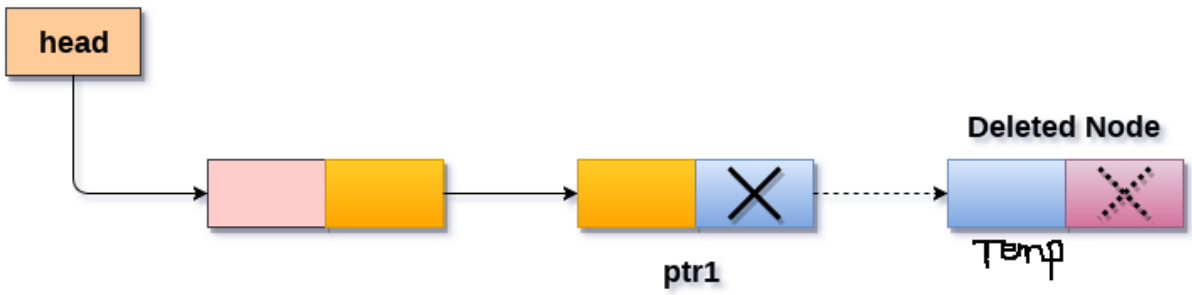


Fig 9 Deletion in singly linked list at the end

Algorithm DELETE_END(Head,Temp,ptr1)

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Steps 4 and 5 while TEMP -> NEXT!= NULL

Step 4: SET PRETEMP = TEMP

Step 5: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 6: SET PRETEMP -> NEXT = NULL

Step 7: FREE TEMP

Step 8: EXIT

Deletion in singly linked list after the specified node :

In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used: temp and ptr1.

Use the following statements to do so.

temp=head;

```
    for(i=0;i<loc;i++)
    {
       ptr1 = temp;
       temp = temp->next;

       if(temp == NULL)
       {
          printf("\nThere are less than %d elements in the list..",loc);
          return;
       }
    }
```

Now, our task is almost done, we just need to make a few pointer adjustments. Make the next of ptr1 (points to the specified node) point to the next of ptr (the node which is to be deleted).

This will be done by using the following statements.

```
ptr1 ->next = temp ->next;
    free(temp);
```

Algorithm DELETE_POS(Head,Temp,Ptr1)

STEP 1: IF HEAD = NULL

WRITE UNDERFLOW

GOTO STEP 10

END OF IF

STEP 2: SET TEMP = HEAD

STEP 3: SET I = 0

STEP 4: REPEAT STEP 5 TO 8 UNTIL I

STEP 5: PTR1 = TEMP

STEP 6: TEMP = TEMP → NEXT

STEP 7: IF TEMP = NULL

WRITE "DESIRED NODE NOT PRESENT"

GOTO STEP 12

END OF IF

STEP 8: I = I+1

END OF LOOP

STEP 9: PTR1 → NEXT = TEMP → NEXT

STEP 10: FREE TEMP

Traversing in singly linked list

Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following statements.

```
temp = head;
    while (temp!=NULL)
      {
        temp = temp -> next;
```

}

Algorithm TRAVERSING()

STEP 1: SET TEMP = HEAD

STEP 2: IF TEMP = NULL

WRITE "EMPTY LIST"

GOTO STEP 7

END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL TEMP != NULL

STEP 5: PRINT TEMP→ DATA

STEP 6: TEMP = TEMP → NEXT

[END OF LOOP]

STEP 7: EXITMP

STEP 11: EXIT

Consider two singly linked list are sorted in increasing order, and merges the two together into one list which is in increasing order. SortedMerge() should return the new list. The new list should be made by splicing together the nodes of the first two lists.

For example: if L1 = 1 -> 3 -> 10 and L2 = 5 -> 6 -> 9 then your program should output the linked list

1 -> 3 -> 5 -> 6 -> 9 -> 10.

Algorithm

The algorithm for this question is quite simple since the two linked lists are already sorted. We create a new linked list and loop through both lists appending the smaller nodes. We'll be using some code that we used in a previous linked list interview question.

Algorithm:

(1) Create a new head pointer to an empty linked list.

(2) Check the first value of both linked lists.

(3) Whichever node from L1 or L2 is smaller, append it to the new list and move the pointer to the next node.

(4) Continue this process until you reach the end of a linked list.

Example

L1 = 1 -> 3 -> 10

L2 = 5 -> 6 -> 9

L3 = null

Compare the first two nodes in both linked lists: (1, 5), 1 is smaller so add it to the new linked list and move the pointer in L1.

L1 = 3 -> 10

L2 = 5 -> 6 -> 9

L3 = 1

Compare the first two nodes in both linked lists: (3, 5), 3 is smaller so add it to the new linked list and move the pointer in L1.

L1 = 10

L2 = 5 -> 6 -> 9

L3 = 1 -> 3

Compare the first two nodes in both linked lists: (10, 5), 5 is smaller so add it to the new linked list and move the pointer in L2.

L1 = 10

L2 = 6 -> 9

L3 = 1 -> 3 -> 5

Compare the first two nodes in both linked lists: (10, 6), 6 is smaller so add it to the new linked list and move the pointer in L2.

L1 = 10

L2 = 9

L3 = 1 -> 3 -> 5 -> 6

Compare the first two nodes in both linked lists: (10, 9), 9 is smaller so add it to the new linked list and move the pointer in L2.

L1 = 10

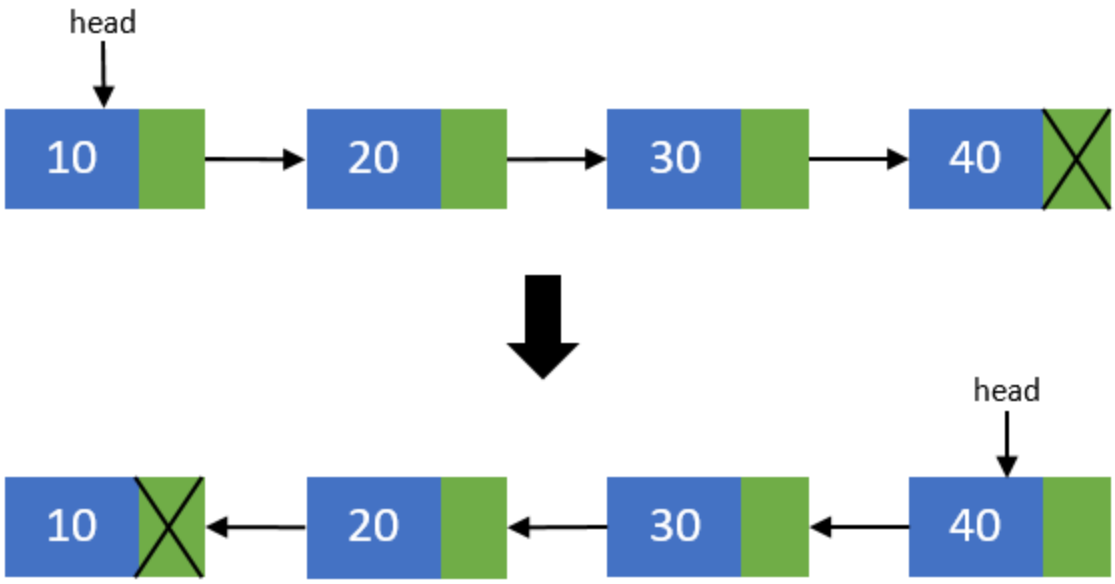
L2 = null

L3 = 1 -> 3 -> 5 -> 6 -> 9

Because L2 points to null, simply append the rest of the nodes from L1 and we have our merged linked list.

L3 = 1 -> 3 -> 5 -> 6 -> 9 -> 10

Reverse a Singly Linked List



Algorithm to reverse a Singly Linked List

Begin:

```
If (head != NULL) then
    prevNode ← head
    head ← head.next
    curNode ← head
    prevNode.next ← NULL
    While (head != NULL) do
        head ← head.next
        curNode.next ← prevNode
        prevNode ← curNode
        curNode ← head
```

End while

head ← prevNode

End if

End

Application of singly linked list to represent polynomial

What is Polynomial?

A polynomial p(x) is the expression in variable x which is in the form (axn + bxn-1 + …. + jx+ k), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

one is the coefficient

other is the exponent

Example

10x2 + 26x, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

1. The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
2. Additional terms having equal exponent is possible one
3. The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent

Polynomial can be represented in the various ways. These are:

1. By the use of arrays
2. By the use of Linked List

Polynomial can be represented in the various ways. These are:

- By the use of arrays

- By the use of Linked List

  A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two-tuple which holds two pieces of information:

  The exponent part

  The coefficient part

  Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

  Example:

  Input:

  1st number = 5x2 + 4x1 + 2x0

  2nd number = -5x1 - 5x0

  Output:

  5x2-1x1-3x0

  Input:

  1st number = 5x3 + 4x2 + 2x0

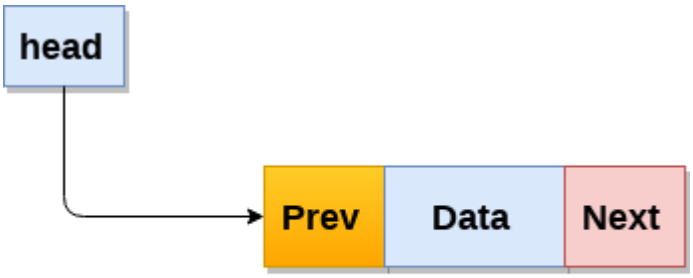
  2nd number = 5x^1 - 5x^0

  Output:

  5x3 + 4x2 + 5x1 - 3x0

Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.

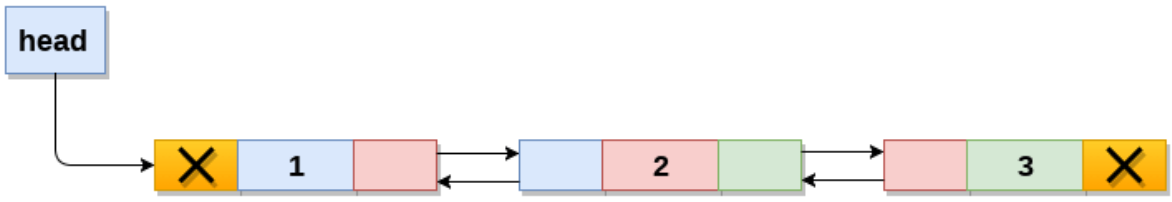


Fig: Doubly linked list

In C, structure of a node in doubly linked list can be given as :

```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
}
```

The prev part of the first node and the next part of the last node will always contain null indicating end in each direction.

In a singly linked list, we could traverse only in one direction, because each node contains address of the next

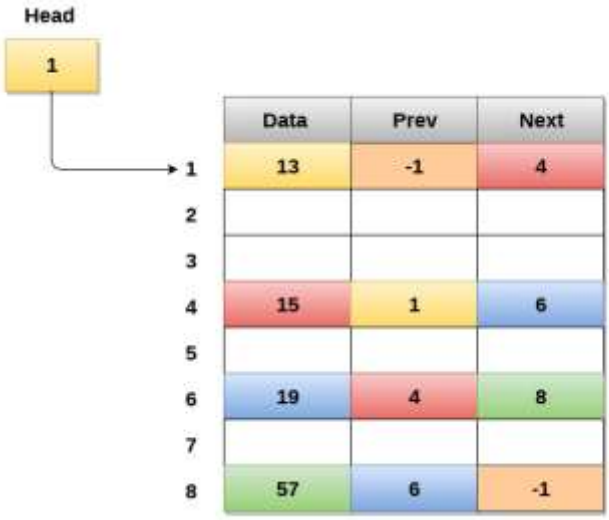
node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the prev of the list contains null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



Operations on doubly linked list

Node Creation

struct node

{

```
    struct node *prev;
    int data;
    struct node *next;
};
struct node *head;
```

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Insertion in doubly linked list at beginning

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.

There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at

least one element. Perform the following steps to insert a node in doubly linked list at beginning.

1. Allocate the space for the new node in the memory.
2. Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

```
newnode->next = NULL;

newnode>prev=NULL;

newnode->data=item;

head=newnode;
```

In the second scenario, the condition head == NULL become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.

This will be done by using the following statements.

```
newnode->next = head;

 head→prev=newnode;
```

Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

```
newnode→prev =NULL

head = newnode
```

Algorithm :

Step 1: IF NEWNODE = NULL

Write OVERFLOW

Go to Step 9

[END OF IF]

Step 2: SET NEW_NODE = TEMP

Step 3: SET TEMP = TEMP -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> PREV = NULL

Step 6: SET NEW_NODE -> NEXT = START

Step 7: SET head -> PREV = NEW_NODE

Step 8: SET head = NEW_NODE

Step 9: EXIT

Insertion in doubly linked list at the end

In order to insert a node in doubly linked list at the end, we must make sure whether the list is empty or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.

1. Allocate the memory for the new node. Make the pointer ptr point to the new node being inserted.
2. Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

```
newnode->next = NULL;

newnode ->prev=NULL;

newnode ->data=item;

head= newnode;
```

In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer temp to head and traverse the list by using this pointer.

```
Temp = head;

while (temp != NULL)

{

temp = temp → next;

   }
```

the pointer temp point to the last node at the end of this while loop. Now, we just need to make a few pointer adjustments to insert the new node ptr to the list. First, make the next pointer of temp point to the new node being inserted i.e. newnode.

```
temp→next =newnode;
```

make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.

```
newnode → prev = temp;
```

make the next pointer of the node ptr point to the null as it will be the new last node of the list.

```
newnode → next = NULL
```

```
Algorithm INSERT_END(Head,Temp,Newnode)

Step 1: IF PTR = NULL

 Write OVERFLOW

  Go to Step 11

 [END OF IF]
```

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = NULL

Step 6: SET TEMP = START

Step 7: Repeat Step 8 while TEMP -> NEXT != NULL

Step 8: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW_NODE

Step 10C: SET NEW_NODE -> PREV = TEMP

Step 11: EXIT

Insertion in doubly linked list after Specified node

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Use the following steps for this purpose.

1. Allocate the memory for the new node.
2. Traverse the list by using the pointer temp to skip the required number of nodes in order to reach the specified node.

```
temp=head;
for(i=0;i<loc;i++)
{
  temp = temp->next;
```

```
    if(temp == NULL)  //up to mentioned location
    {
        return;
    }
  }
```

The temp would point to the specified node at the end of the for loop. The new node needs to be inserted after this node therefore we need to make a few pointer adjustments here. Make the next pointer of ptr point to the next node of temp.

ptr → next = temp → next;

make the prev of the new node ptr point to temp.

ptr → prev = temp;

make the next pointer of temp point to the new node ptr.

temp → next = ptr;

make the previous pointer of the next node of temp point to the new node.

temp → next → prev = ptr;

Algorithm INSERT_POS()

Step 1: IF PTR = NULL

  Write OVERFLOW

  Go to Step 15

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET TEMP = START

Step 6: SET I = 0

Step 7: REPEAT 8 to 10 until I

Step 8: SET TEMP = TEMP -> NEXT

STEP 9: IF TEMP = NULL

STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"

GOTO STEP 15

[END OF IF]

[END OF LOOP]

Step 11: SET NEW_NODE -> NEXT = TEMP -> NEXT

Step 12: SET NEW_NODE -> PREV = TEMP

Step 13 : SET TEMP -> NEXT = NEW_NODE

Step 14: SET TEMP -> NEXT -> PREV = NEW_NODE

Step 15: EXIT

Deletion at beginning

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

Ptr = head;

head = head → next;

now make the prev of this new head node point to NULL. This will be done by using the following statements.

head → prev = NULL

Now free the pointer ptr by using the free function.

free(ptr)

Algorithm

STEP 1: IF HEAD = NULL

WRITE UNDERFLOW

GOTO STEP 6

STEP 2: SET PTR = HEAD

STEP 3: SET HEAD = HEAD → NEXT

STEP 4: SET HEAD → PREV = NULL

STEP 5: FREE PTR

STEP 6: EXIT

Deletion in doubly linked list at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.

If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.

Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;

    if(ptr->next != NULL)

    {

       ptr = ptr -> next;

    }
```

The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of ptr to NULL.

ptr → prev → next = NULL

free the pointer as this the node which is to be deleted.

free(ptr)

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 7

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL

Step 4: SET TEMP = TEMP->NEXT

[END OF LOOP]

Step 5: SET TEMP ->PREV-> NEXT = NULL

Step 6: FREE TEMP

Step 7: EXIT

Deletion in doubly linked list after the specified node

In order to delete the node after the specified data, we need to perform the following steps.

Copy the head pointer into a temporary pointer temp.

temp = head

Traverse the list until we find the desired data value.

while(temp -> data != val)

temp = temp -> next;

Check if this is the last node of the list. If it is so then we can't perform deletion.

if(temp -> next == NULL)

{

return;

}

Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.

```
if(temp -> next -> next == NULL)
   {
      temp ->next = NULL;
   }
```

Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

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

Algorithm

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 9

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Step 4 while TEMP -> DATA != ITEM

Step 4: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 5: SET PTR = TEMP -> NEXT

Step 6: SET TEMP -> NEXT = PTR -> NEXT

Step 7: SET PTR -> NEXT -> PREV = TEMP

Step 8: FREE PTR

Step 9: EXIT Traversing in doubly linked list

Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr.

Ptr = head

then, traverse through the list by using while loop. Keep shifting value of pointer variable ptr until we find the last node. The last node contains null in its next part.

```
while(ptr != NULL)
    {
       printf("%d\n",ptr->data);
       ptr=ptr->next;
    }
```

Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

Algorithm

Step 1: IF HEAD == NULL

WRITE "UNDERFLOW"

GOTO STEP 6

[END OF IF]

Step 2: Set PTR = HEAD

Step 3: Repeat step 4 and 5 while PTR != NULL

Step 4: Write PTR → data

Step 5: PTR = PTR → next

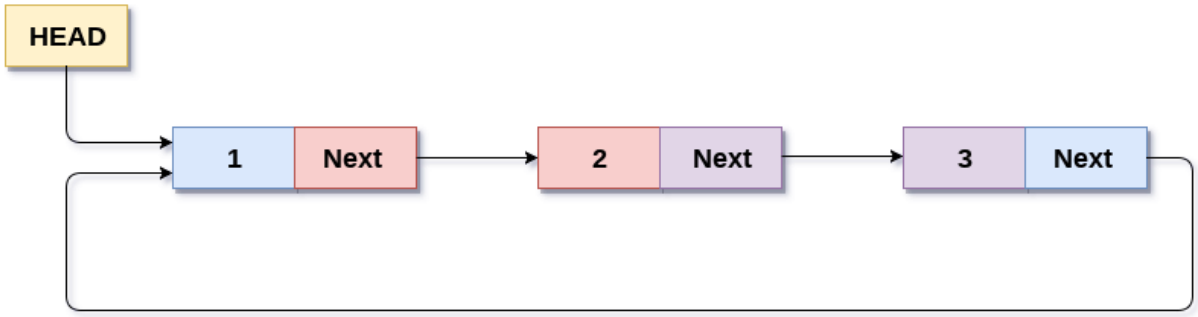
Step 6: Exit

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



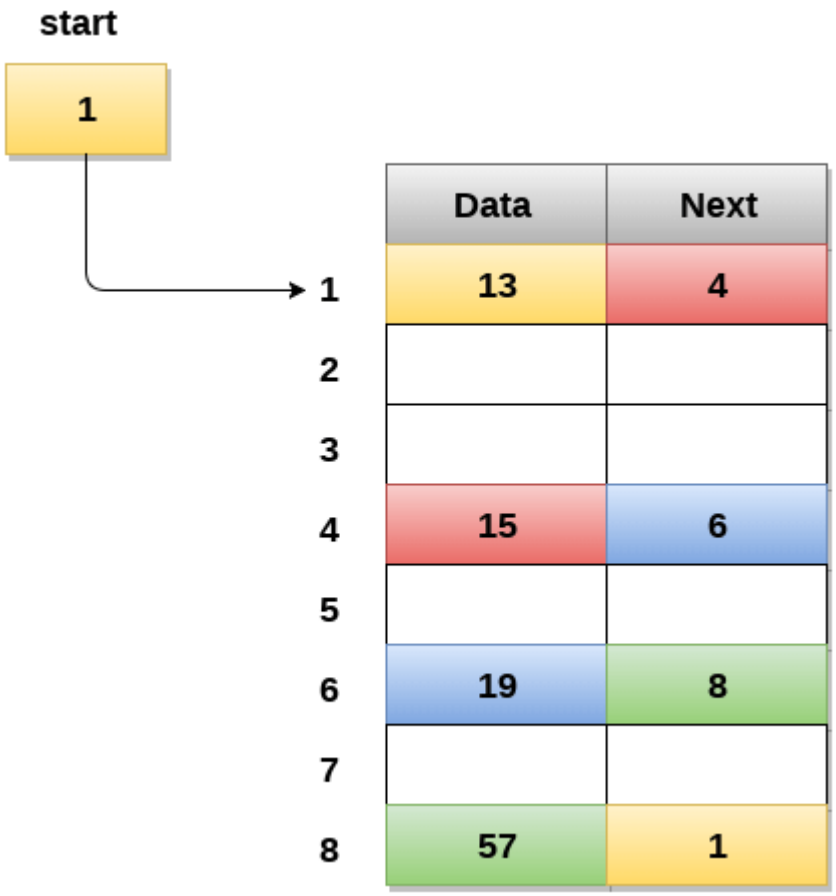
Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects.

However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.



However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.

Circular Singly Linked List

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

Operations on Circular Singly linked list:

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Insertion into circular singly linked list at beginning

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

Firstly, allocate the memory space for the new node

In the first scenario, the condition head == NULL will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node. This will be done by using the following statements.

```
if(head == NULL)
    {
       head = ptr;
       ptr -> next = head;
    }
```

In the second scenario, the condition head == NULL will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

```
temp = head;
        while(temp->next != head)
           temp = temp->next;
```

At the end of the loop, the pointer temp would point to the last node of the list. Since, in a circular singly linked list, the last node of the list contains a pointer to the first node of the list. Therefore, we need to make the next pointer of the last node point to the head node of the list and the new node which is being inserted into the list will be the new head node of the list therefore the next pointer of temp will point to the new node ptr.

This will be done by using the following statements.

```
temp -> next = ptr;
```

the next pointer of temp will point to the existing head node of the list.

```
ptr->next = head;
```

Now, make the new node ptr, the new head node of the circular singly linked list.

```
head = ptr;
```

in this way, the node ptr has been inserted into the circular singly linked list at beginning.

Algorithm

Step 1: IF PTR = NULL

  Write OVERFLOW

Go to Step 11

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET TEMP = HEAD

Step 6: Repeat Step 8 while TEMP -> NEXT != HEAD

Step 7: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 8: SET NEW_NODE -> NEXT = HEAD

Step 9: SET TEMP → NEXT = NEW_NODE

Step 10: SET HEAD = NEW_NODE

Step 11: EXIT

Insertion into circular singly linked list at the end

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

Algorithm

Step 1: IF PTR = NULL

Write OVERFLOW

Go to Step 1

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = HEAD

Step 6: SET TEMP = HEAD

Step 7: Repeat Step 8 while TEMP -> NEXT != HEAD

Step 8: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW_NODE

Step 10: EXIT

eletion & Traversing

	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.

4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Deletion in circular singly linked list at beginning

In order to delete a node in circular singly linked list, we need to make a few pointer adjustments.

There are three scenarios of deleting a node from circular singly linked list at beginning.

Scenario 1: (The list is Empty)

If the list is empty then the condition head == NULL will become true, in this case, we just need to print underflow on the screen and make exit.

```
if(head == NULL)
   {
      printf("\nUNDERFLOW");
      return;
   }
```

Scenario 2: (The list contains single node)

If the list contains single node then, the condition head → next == head will become true. In this case, we need to delete the entire list and make the head pointer free. This will be done by using the following statements.

```
if(head->next == head)
{
   head = NULL;
   free(head);
```

}

Scenario 3: (The list contains more than one node)

If the list contains more than one node then, in that case, we need to traverse the list by using the pointer ptr to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;
 while(ptr -> next != head)
    ptr = ptr -> next;
```

At the end of the loop, the pointer ptr point to the last node of the list. Since, the last node of the list points to the head node of the list. Therefore this will be changed as now, the last node of the list will point to the next of the head node.

```
ptr->next = head->next;
```

Now, free the head pointer by using the free() method in C language.

```
free(head);
```

Make the node pointed by the next of the last node, the new head of the list.

```
head = ptr->next;
```

In this way, the node will be deleted from the circular singly linked list from the beginning.

Algorithm

Step 1: IF HEAD = NULL

Write UNDERFLOW

  Go to Step 8

 [END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Step 4 while PTR → NEXT != HEAD

Step 4: SET PTR = PTR → next

[END OF LOOP]

Step 5: SET PTR → NEXT = HEAD → NEXT

Step 6: FREE HEAD

Step 7: SET HEAD = PTR → NEXT

Step 8: EXIT

Deletion in Circular singly linked list at the end

There are three scenarios of deleting a node in circular singly linked list at the end.

Scenario 1 (the list is empty)

If the list is empty then the condition head == NULL will become true, in this case, we just need to print underflow on the screen and make exit.

```
if(head == NULL)
   {
      printf("\nUNDERFLOW");
      return;
   }
```

Scenario 2(the list contains single element)

If the list contains single node then, the condition head → next == head will become true. In this case, we need to delete the entire list and make the head pointer free. This will be done by using the following statements.

```
if(head->next == head)
{
   head = NULL;
   free(head);
}
```

Scenario 3(the list contains more than one element)

If the list contains more than one element, then in order to delete the last element, we need to reach the last node. We also need to keep track of the second last node of the list. For this purpose, the two pointers ptr and preptr are defined. The following sequence of code is used for this purpose.

```
ptr = head;
      while(ptr ->next != head)
      {
         preptr=ptr;
         ptr = ptr->next;
      }
```

now, we need to make just one more pointer adjustment. We need to make the next pointer of preptr point to the next of ptr (i.e. head) and then make pointer ptr free.

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

Algorithm

Step 1: IF HEAD = NULL

  Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != HEAD

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 6: SET PREPTR -> NEXT = HEAD

Step 7: FREE PTR

Step 8: EXIT

Traversing in Circular Singly linked list

Traversing in circular singly linked list can be done through a loop. Initialize the temporary pointer variable temp to head pointer and run the while loop until the next pointer of temp becomes head. The algorithm and the c function implementing the algorithm is described as follows.

Algorithm

STEP 1: SET PTR = HEAD

STEP 2: IF PTR = NULL

WRITE "EMPTY LIST"

GOTO STEP 8

END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR → NEXT != HEAD

STEP 5: PRINT PTR → DATA

STEP 6: PTR = PTR → NEXT

[END OF LOOP]

STEP 7: PRINT PTR→ DATA

STEP 8: EXIT

SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – III -Data Structures – SCSA1203

UNIT -III

UNIT 3 STACKS **9 Hrs.**

Basic Stack Operations - Representation of a Stack using Arrays - Algorithm for Stack Operations - Stack Applications: Reversing list - Factorial Calculation - Infix to postfix Transformation - Evaluating Arithmetic Expressions

3. STACK

3.1 INTRODUCTION

Stack is a linear data structure which follows a particular order in which the operations are performed. In stack, insertion and deletion of elements happen only at one end, i.e., the most recently inserted element is the one deleted first from the set.

This could be explained using a simple analogy, a pile of plates, where one plate is placed on the top of another. Now, when a plate is to be removed, the topmost one is removed. Hence insertion/removal of a plate is done at the topmost position. Few real world examples are given as:

- Stack of plates in a buffet table. The plate inserted at last will be the first one to be removed out of stack.

- Stack of Compact Discs

- Stack of Moulded chairs

- Bangles on Women's Hand

- Books piled on top of each other

These above examples state, stack as a linear list where all insertion and deletion are done only at one end of the list called Top. i.e., Stack implements **Last-in, First-out (LIFO)**policy. Stack can be implemented as an Array or Linked list.

3.2 STACK

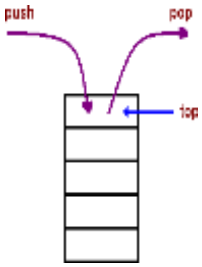


Fig. 3.2.1 Representation of stack as Array

Where does the stack concept implemented in computers? Though there are various applications, simple answer is in function calls. Consider the below example:

main ():	def funA():	def funB():	def funC():
----	----	----	----
funA()	funB()	funC()	----
----	----	----	----

In order to keep track of returning point of each active function, a special stack named System stack is used.

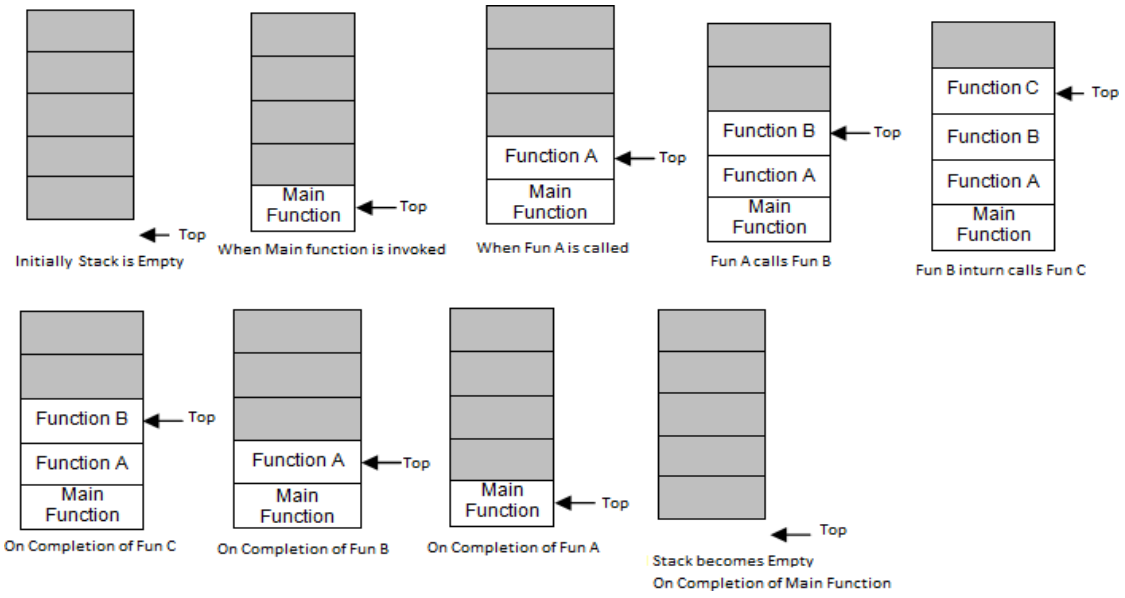


Fig. 3.2.2 Function calls with Stack

when A calls function B, A is pushed onto system stack. Similarly when B calls C,B is pushed onto stack and so on. Once the execution of function C is complete, the control will remove C from the stack. Thus system stack ensures proper execution order of functions.

The below diagram (Figure 3.3) depicts expansion and shrinking of a stack, initially stack is empty

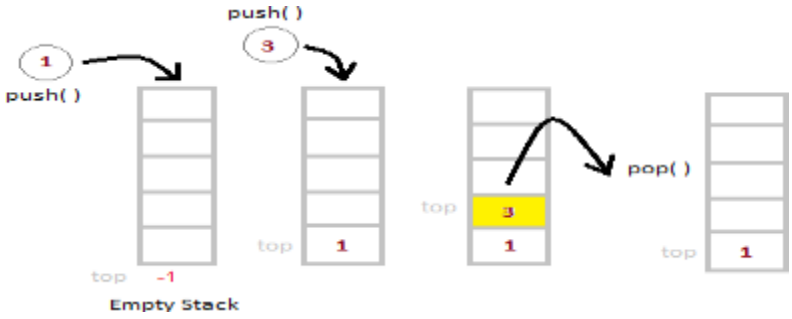


Fig. 3.2.3 Operations on Stack

In Stack, all operations take place at the "top" .Push(x) operation adds an item x at the top of stack and the Pop() operation removes an item from the top of stack.

3.2.1 Stack as an ADT (Abstract Data Type):

Stack can be represented using an array. A one dimensional array is used to hold the elements of the stack and a variable "top" is used for representing the index of the top most element. Formally defined as:

```
type def struct stack
{
int data[size];          // size is constant, represents maximum
int top;                 // number of elements that can be stored.
} S;
```

3.2.2 Basic Stack Operations

- **isEmpty:** Returns true if stack is empty, else false.
- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

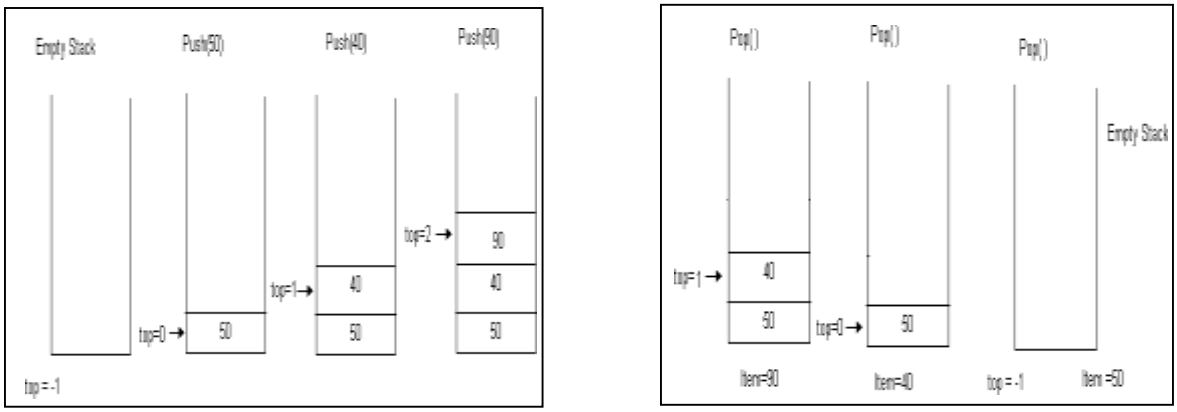


Fig.3.2.4.1 Push and Pop Operation of Stack

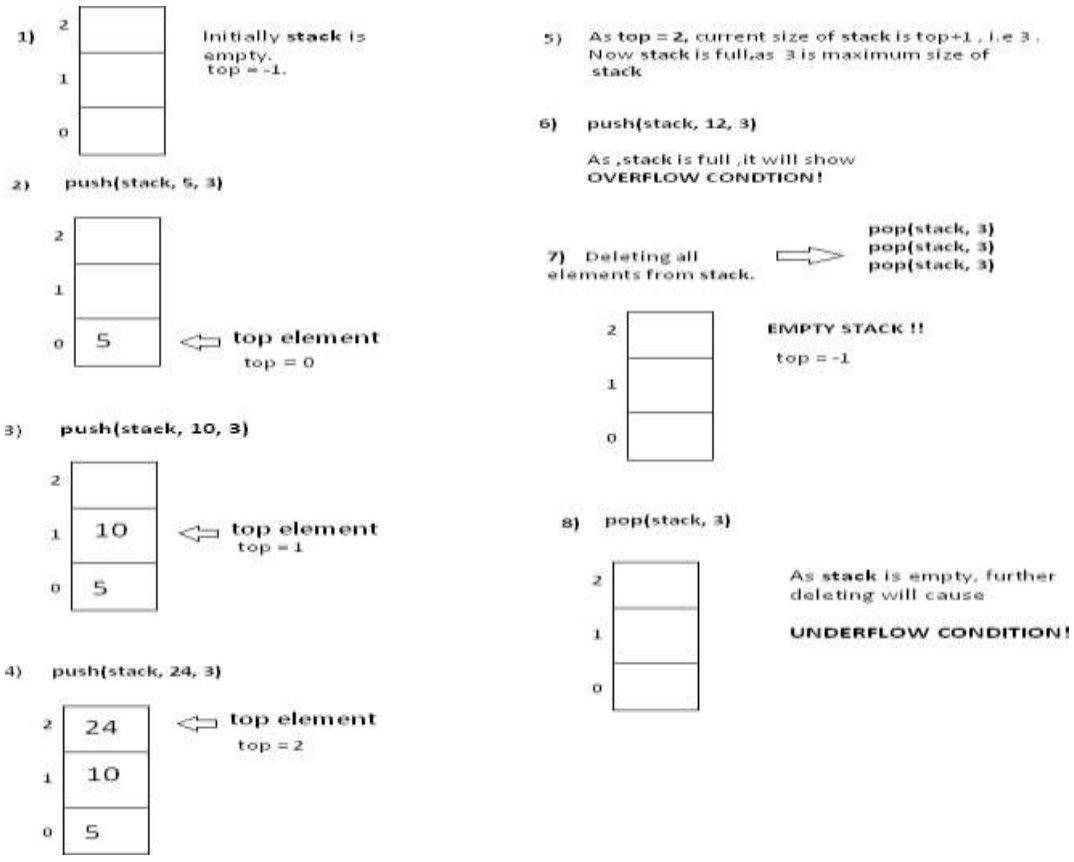


Fig.3.2.4.2 Push and Pop Operation of Stack

The fundamental operations in stack are given as follows:

STACK_EMPTY(S)

```
 if  S.top  ==  0
    return TRUE
 else
    return FALSE

PUSH ( S.x )
   S.top = S.top+1
   S[S.top] = x

POP(S)
  if STACK_EMPTY(S)
      error   "underflow"
  else
      x   =   S[S.top]
      S.top = S.top-1
      return x
```

When S.top = 0, the stack contains no elements and is empty. In that case an attempt to pop an element from empty stack is named stack **underflow**, which is normally reported as error. If S.top exceeds size, then stack **overflows**. In pseudo code implementation stack overflow is not represented. Each of the above three stack operations takes **O(1)** time.

3.2.3 ARRAY REPRESENTATION OF STACK

Stack is represented as a linear array. In an array-based implementation the following fields are maintained: an array S of a default Size (≥ 1), the variable *top* that refers to the top element in the stack and the size that refers to the array size. The variable *top* changes from -1 to Size-1. Stack is called empty when top = -1, and the stack is full when top = Size-1.Consider an example stack with size=10.

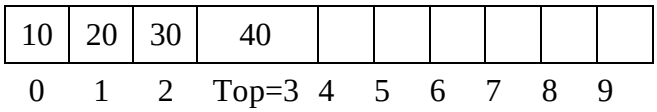


Fig.3.2.3.1 Array Representation of Stack

The variable Top represents the topmost element of the stack. In the above example six more elements could be stored.

OPERATIONS ON STACK

The two basic operations of stack are push and pop. Let's first discuss about array representation of these operations.

Push Operation

The push operation is used to insert an element into the stack. The element is added always at the topmost position of stack. Since the size of array is fixed at the time of declaration, before inserting the value, check if top=Size-1, if so stack is full and no more insertion is possible. In case of an attempt to insert a value in full stack, an OVERFLOW message gets displayed.

Consider the below example: Size=10

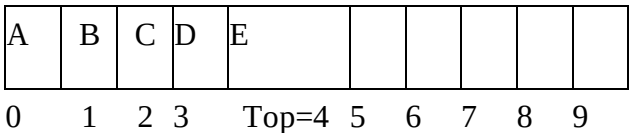


Fig.3.2.3.2 Array Representation of Stack, Push(E)

To insert a new element F, first check if Top==Size-1. If the condition fails, then increment the Top and store the value. Thus the updated stack is:

Fig.3.2.3.3 Array Representation of Stack, Push(F)

Algorithm:

Push(X)

```
if Top == Size-1
    Write "Stack Overflow"
```

```
                    return
else
                    Top = Top + 1       // St represents an Array with maximum limit as Size
                    St[Top] = X         // X element to be inserted
End
```

Pop Operation

The pop operation is used to remove the topmost element from the stack. In this case, first check the presence of element, if top == -1 (indicates no array elements), then it indicates empty stack and thereby deletion not possible. In case of an attempt to delete a value in an empty stack, an UNDERFLOW message gets displayed.

Consider the below example: Size=10

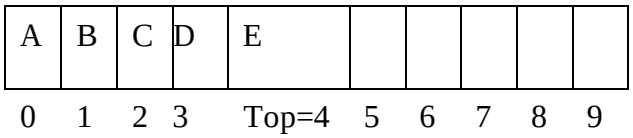


Fig.3.2.3.4 Array Representation of Stack, Pop()

To delete the topmost element, first check if Top== -1. If the condition fails, then decrement the Top.Thus the updated stack is:

Fig.3.2.3.5 Array Representation of Stack, Pop()

Algorithm:

```
Pop()
  if Top == -1
     Write   "Stack   Underflow"
     return
  else
    X = St[Top]          // St represents an Array with maximum limit as Size
    Top = Top-1          // X represents element removed
     return X
```

End

Peek Operation

The peek operation returns the topmost element of the stack. Here if stack is not empty, the top element is displayed.

Consider the below example: Size=10

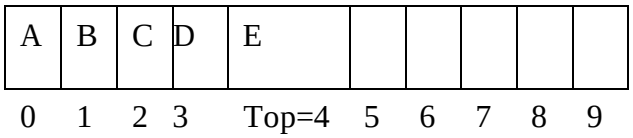


Fig.3.2.3.6 Array Representation of Stack, Peek()

Here, the peek operation will return E, as it's the topmost element.

Algorithm:

```
Peek()
    if Top == -1
        Write  "Stack  Underflow"
    else
        return St[Top]          // St represents an Array with maximum limit as Size
End
```

Implementation of stack operations using arrays in Python

```
class Stack:
    #   Constructor
    def _init_(self):
      self.stack   =   list()
      self.maxSize = 5
      self.top = 0
```

```
    # Add element to the Stack def
    push(self,data):
        if    self.top>=self.maxSize:
            return ("Stack Full!")
        self.stack.append(data)
        self.top += 1
        return "element inserted"

    # Remove element from the stack
    def pop(self):
        if self.top<=0:
            return  ("Stack  Empty!")
        item    =    self.stack.pop()
        self.top -= 1
        return item

    # Size of the stack
    def size(self):
        return self.top

s = Stack()
print("Push : ",s.push(1))
print("Push : ",s.push(2))
print("Push : ",s.push(3))
print("Push       :       ",s.push(4))
print("Size  of  Array  :",s.size())
print("Element Popped :",s.pop())
print("Element Popped :",s.pop())
print("Element Popped :",s.pop())
print("Element Popped :",s.pop())
print("Element Popped :",s.pop())
```

```
Push :  element inserted
Push :  element inserted
Push :  element inserted
Push :  element inserted
Size of Array : 4
Element Popped : 4
Element Popped : 3
Element Popped : 2
Element Popped : 1
Element Popped : Stack Empty!
>>>
```

Pros:-

- Easier to use.Array elements could be accessed randomly using the array index.
- Less memory allocation, no need to track the next node.
- Data elements are stored in contiguous locations in memory.

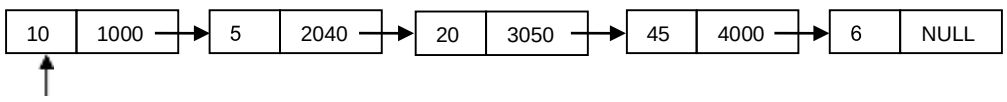
Cons:-

- Fixed size – cannot increase or decrease the total number of elements.

3.2.4 LINKED REPRESENTATION OF STACK

Stack using arrays was discussed in the previous section, the drawback of the above concept is that the array must be pre-declared i.e., size of array is fixed. If array size cannot be determined in advance, i.e., in case of dynamic storage,Linked List representation could be implemented.

In a linked list, every node has two parts, one that stores data and the other represents address of next node. The head pointer is given as Top, all insertions and deletions occur at the node pointed by Top.



Top

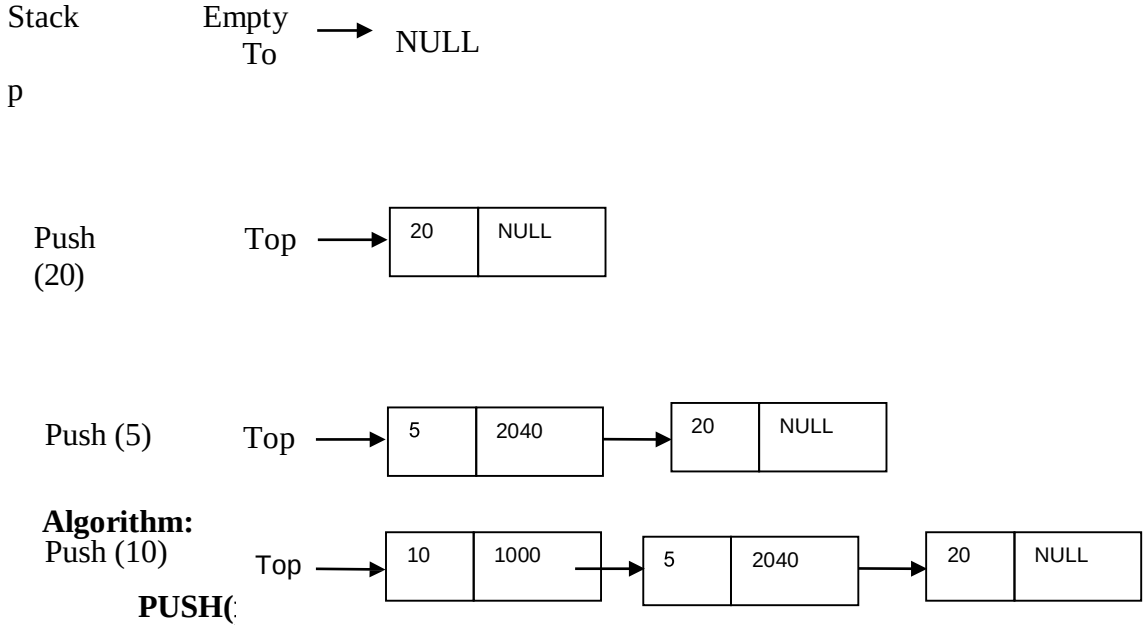
Fig.3.2.4.1 Array Representation of Stack

OPERATIONS ON STACK

Linked List representation supports the two basic operations of stack, push and pop.

Push operation

Initially, when the stack is empty the pointer top points NULL. When an element is added using the push operation, top is made to point to the latest element whichever is added.



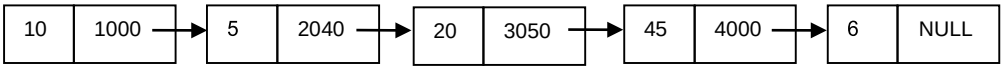
Algorithm:

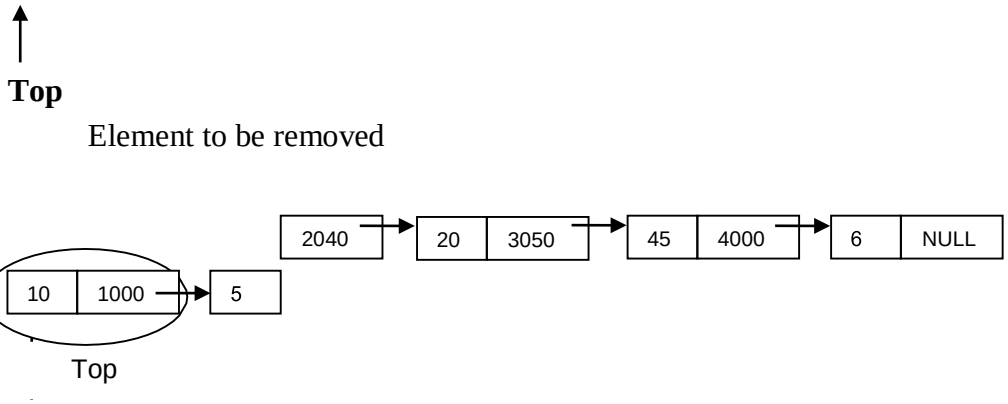
PUSH(

Allocate memory for a new node and name it as Temp
Temp->data = x
Temp->link = Top
Top = Temp
End

Pop operation

The data in the topmost node of the stack, which is to be removed, is stored in a variable called item. Then a temporary pointer is created to point to top. The top is now safely moved to the next node below it in the stack. Temp node is then deleted and the item is returned.




Top
Element to be removed
2040
20
3050
45
4000
6
NULL
10
1000
5
Top
Temp

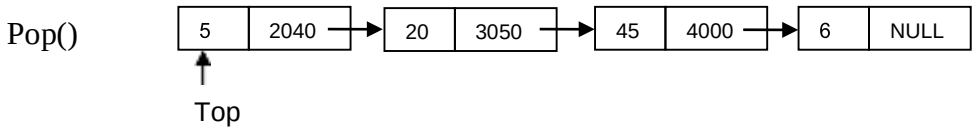


Fig.3.2.4.1 Linked List Representation of Stack

Algorithm:

POP()

```
if Top == NULL print
    "Underflow"
    return
else
    Temp = Top
    Item  =  Temp->data
    Top   =   Top->link
    delete Temp
    return Item
End
```

Implementation of stack operations using linked lists in Python

```
class Node:
    def__init__(self,data):
        self.data    =    data
        self.next = None
class Stack:                    # default value of head is NULL
    def init (self):
        self.head = None
    def isempty(self):          # Checks if stack is empty
```

```
        if self.head == None:
            return True
        else:
            return False

    # Method to add data to the stack
    def push(self,data):
        if self.head == None:
            self.head=Node(data)
        else:
            newnode = Node(data)
            newnode.next = self.head
            self.head = newnode

# Remove element that is the current head (start of the stack) def
    pop(self):
        if self.isempty():
            return None
        else:
            temp = self.head
            self.head = self.head.next
            temp.next = None
            return temp.data
# Returns the head node data
    def peek(self):
        if self.isempty():
            return None
        else:
            return self.head.data
# Prints out the stack
    def display(self):
        temp = self.head
        if self.isempty():
            print("Stack Underflow")
        else:
            while(temp != None):
                print(temp.data,"->",end = " ")
                temp = temp.next
            print("NULL")
            return
```

```
# Driver code
St = Stack()
St.push(11)
St.push(22)
St.push(33)
St.push(44)

# Display stack elements
print("\nElements in Stack : ")
St.display()

# Print top elementof stack
print("\nTop element is ",St.peek())

# Delete top elements of stack
St.pop()
St.pop()

# Display stack elements
print("\nElements in Stack : ")
St.display()

# Print top elementof stack
print("\nTop element is ",St.peek())
```

Output:

```
Elements in Stack :
44 -> 33 -> 22 -> 11 -> NULL

Top element is  44

Elements in Stack :
22 -> 11 -> NULL

Top element is  22
>>>
```

Pros:-

- No fixed size declaration.
- New elements can be stored anywhere and a reference is created for the new element using pointers.

Cons:-

- Requires extra memory to keep details about next node.
- Random accessing is not possible in linked lists. The elements will have to be accessed sequentially.

Difference between an Array and Stack ADT

Stack:

- Size of the stack keeps on changing with insertion/deletionoperation.
- Stack can store elements of different data types [Heterogeneous].

Array:

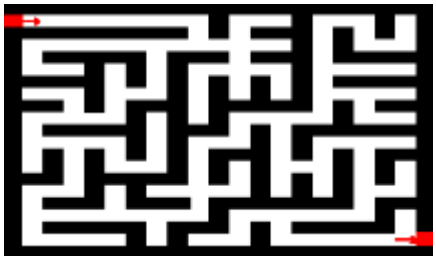
- Size of the array is fixed at the time of declaration itself
- Array stores elements of similar data type [Homogeneous].

3.3 APPLICATIONS OF STACK

In this section, simple applications of stack that are extensively used in computer applications are discussed. Few applications of Stack are listed as below:

- Stack - undo\redo operation in word processors.
- Expression evaluation and syntax parsing.
- Many virtual machines like JVM are stack oriented**.**
- DFS Algorithm - Depth First Search.
- Graph Connectivity.
- LIFO scheduling policy of CPU.
- When a processor receives an interrupt, it completes its execution of the current instruction, then pushes the process's PCB to the stack and then perform the ISR (Interrupt Service Routine)

- When a process calls a function inside a function - like recursion, it pushes the current data like local variables onto the stack to be retrieved once the control is returned.
- Reverse polish AKA postfix notations.
- Used in IDEs to check for proper parenthesis matching.
- Browser back-forth button.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack
- Backtracking. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?



Once you reach a dead end, you must backtrack. But backtrack to where?,the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

3.3.1 Reversing a list

A list of numbers are reversed with a stack by just pushing all elements one by one into the stack and then elements are popped one by one and stored back starting from the first index of the list.

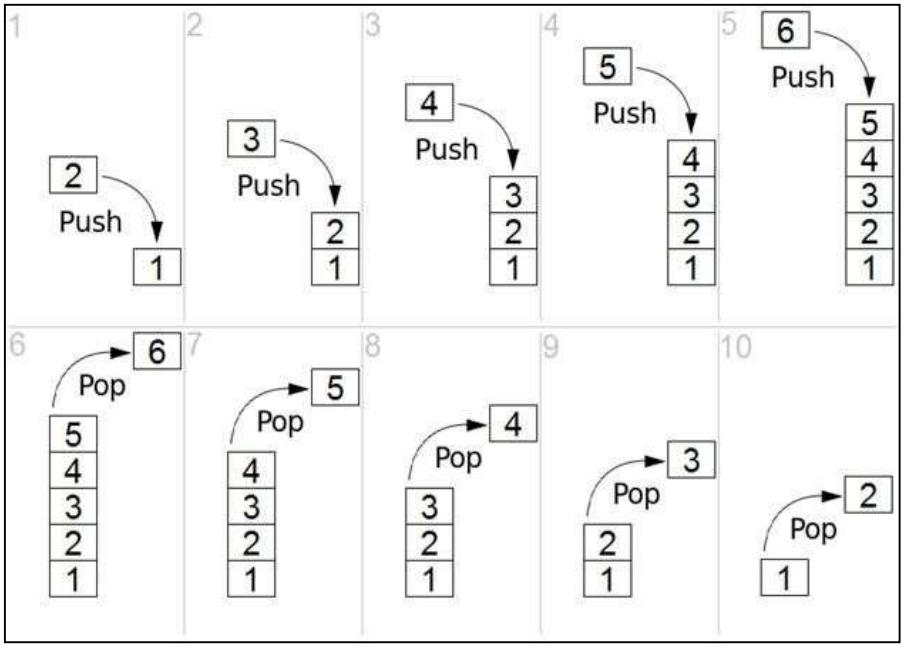


Fig.3.3.1 Reversing a list using Stack

The logic behind for implementing reverse of a string:

- Read a string.
- Push all characters until NULL is not found - Characters will be stored in stack variable.
- Pop all characters until NULL is not found - Stack is a LIFO technique, so last character will be pushed first and finally will get reversed string in a variable in which input string is stored.

Implementation in Python

```
class Stack:
                    def_init_(self):
                    self.items = []
    def is_empty(self):
```

```
        return self.items == []
    def push(self, data):
        self.items.append(data)
    def pop(self):
        return self.items.pop()
    def display(self):
        for data in reversed(self.items):
            print(data,end=" ")
    def insert_at_bottom(s, data):
        if s.is_empty():
            s.push(data)
        else:
            popped = s.pop()
            insert_at_bottom(s, data)
            s.push(popped)
    def reverse_stack(s):
        if not s.is_empty():
            popped = s.pop()
            reverse_stack(s)
            insert_at_bottom(s, popped)
s = Stack()
data_list = input('Enter the elements to push : \n').split()
for data in data_list:
    s.push(int(data))
print('The elements in stack :')
s.display()
reverse_stack(s)
print('\nAfter reversing : ')
s.display()
```

Output :

```
Enter the elements to push :
10 15 20 25 30 35
```

The elements in stack :

35 30 25 20 15 10

After reversing :

10 15 20 25 30 35

3.3.2 Recursion

All recursive functions are examples of implicit application of Stack ADT.A recursive function is defined as a function that calls itself again to complete a smaller version of task until a final call, which does not require a call to itself is met. Therefore recursion is implemented for solving complex problems in terms of smaller and easily solvable problems.

To understand recursive functions, let's consider the simplest example of calculating factorial of a number.

Factorial Calculation using Recursive Approach

To calculate n!, multiply the given number with factorial of that number less than one. i.e., n! = n x (n-1) !

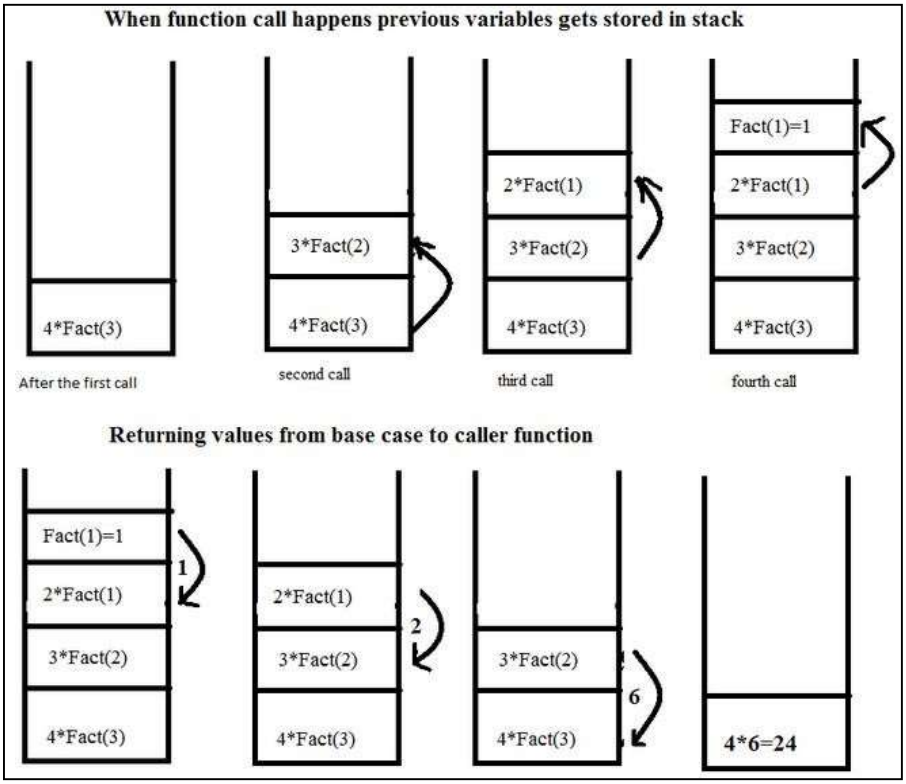


Fig.3.3.2 Factorial Calculation using Stack

Consider an example 5!

5! = 5 x (4)! , where 4! = 4 x (3)! , where 3! = 3 x (2)! ,where 2! = 2 x(1)!,where 1!=1;

i.e., 5! = 5 x 4x 3x 2x 1

= 120

Python Code:

```
def factorial(n):
    if(n <= 1):
        return 1
    else:
        return(n*factorial(n-1))
n = int(input("Enter number :"))
print("Factorial :",factorial(n))
```

Output:

```
Enter number : 5
5
Factorial : 120
>>>
```

3.3.3 Conversion of Infix Expression to Postfix Expression

The stacks are frequently used in evaluation of arithmetic expressions. An arithmetic expression consists of operands and operators. The operands can be numeric values or numeric variables. The operators used in an arithmetic expression represent the operations like addition, subtraction, multiplication, division and exponentiation.

The arithmetic expression expressed in its normal form is said to be Infix notation, as shown: A * B

The above expression in prefix form would be represented as follows:* AB

The same expression in postfix form would be represented as follows:AB *

Hence the given expression in infix form is first converted to postfix form and then evaluated to obtain the results. Postfix expressions are represented as Reverse Polish notation. To understand the conversion of Infix to Postfix expression, knowledge about operator precedence is important.

Operator Precedence:

The Precedence of the operators takes a crucial place while evaluating expressions. The top operator in the table has the highest precedence. As per the precedence, the operators will be pushed to the stack.

Table.3.3.3 Operator Precedence

S.No	Operator	Associativity	S.No	Operator	Associativity	S.No	Operator	Associativity
1	( )	Left to Right	11	&	Right to Left	21	|	Left to Right
2	[ ]	Left to Right	12	sizeof	Right to Left	22	&&	Left to Right
3	.	Left to Right	13	* / %	Left to Right	23	| |	Left to Right
4	->	Left to Right	14	+ -	Left to Right	24	? :	Right to Left
5	++ --	Left to Right	15	<< >>	Left to Right	25	=	Right to Left
6	++ --	Right to Left	16	< <=	Left to Right	26	+= -=	Right to Left
7	+ -	Right to Left	17	> >=	Left to Right	27	*= /=	Right to Left
8	! ~	Right to Left	18	== !=	Left to Right	28	%= &=	Right to Left
9	(type)	Right to Left	19	&	Left to Right	29	^= |=	Right to Left
10	*	Right to Left	20	^	Left to Right	30	<<= >>=	Right to Left

The function to convert an expression from infix to postfix consists following steps:

1. Every character of the expression string is scanned in a while loop until the end of the expression is reached.

2. Following steps are performed depending on the type of character scanned.

    (a) If the character scanned happens to be a space then that character is skipped.

    (b) If the character scanned is a digit or an alphabet, it is added to the target string pointed to by t.

    (c) If the character scanned is a closing parenthesis then it is added to the stack by calling push( ) function.

    (d) If the character scanned happens to be an operator, then firstly, the topmost element from the stack is retrieved. Through a while loop, the priorities of the character scanned and the character popped 'opr' are compared. Then following steps are performed as per the precedence rule.

        i. If 'opr' has higher or same priority as the character scanned, then opr is added to the target string.

        ii. If opr has lower precedence than the character scanned, then the loop is terminated. Opr is pushed back to the stack. Then, the character scanned is also added to the stack.

    (e) If the character scanned happens to be an opening parenthesis, then the operators present in the stack are retrieved through a loop. The loop continues till it does not encounter a closing parenthesis. The operators popped, are added to the target string pointed to by t.

3. Now the string pointed by t is the required postfix expression.

Example: Convert A * (B + C) * D to postfix notation.

Step No	Input	Stack	Output
1	A	Empty	A
2	*	*	A
3	(	(*	A
4	B	(*	A B
5	+	+(*	A B
6	C	+(*	A B C
7	)	*	A B C +
8	*	*	A B C + *
9	D	*	A B C + * D
10	END	Empty	A B C + * D *

Notes:

- In this table, the stack grows toward the left. Thus, the top of the stack is the leftmost symbol.
- Step No.3, the left parenthesis was pushed without popping the * because * had a lower priority then "(".
- Step No.5, "+" was pushed without popping "(" because you never pop a left parenthesis until you get an incoming right parenthesis. In other words, there is a distinction between incoming priority, which is very high for a "(", and instack priority, which is lower than any operator.
- Step No.7, the incoming right parenthesis caused the "+" and "(" to be popped but only the "+" as written out.
- Step.No.8, the current "*" had equal priority to the "*" on the stack. So the "*" on the stack was popped, and the incoming "*" was pushed onto the stack.

Pseudo-code:

```
CONVERSION(INFIX)
Begin
Read infix
```

```
L=Length(infix)
J=1
For i=1 to L
Do
        C=infix(i)
        If C is a number OR Alphabet Then
                Postfix[j]= c
                j=j+1
        if C= ‘( ‘ Then
                Push ( C )
        if C= ‘*’ ||’ /’|| ‘+ ‘||’ –‘ ||’ % ‘
                If top = 0 then
                        Push( C )
                Else If Priority ( C )< Priority (Stack[top]) then
                        Postfix[j]= pop()
                        J=j+1
                        Push( C )
Else
Push ( C )

if C= ‘) ‘
                Repeat
postfix[j]=pop()
                  j=j+1
                Until (stk[top]<>'(')
pop()
        End if
End For

Repeat
        postfix[j]=pop()
j=j+1

Until (top > = 0)
Write postfix        # Print the resultant postfix string
END CONVERSION(INFIX)

FUNCTION PRIORITY(CHAR CH)
Begin
if (ch=='*'||ch=='/'||ch=='%') then
return 2
else if(ch=='+'||ch=='-') then
return 1
else
return 0
END FUNCTION PRIORITY
```

EXAMPLE:

A+(B*C-(D/E – F)*G)*H

Step No	Input	Stack	Output
1	A+(B*C-(D/E-F)*G)*H	Empty	-
2	+(B*C-(D/E-F)*G)*H	Empty	A
3	(B*C-(D/E-F)*G)*H	+	A
4	B*C-(D/E-F)*G)*H	+(	A
5	*C-(D/E-F)*G)*H	+(	AB
6	C-(D/E-F)*G)*H	+(*	AB
7	-(D/E-F)*G)*H	+(*	ABC
8	(D/E-F)*G)*H	+(-	ABC*
9	D/E-F)*G)*H	+(-(	ABC*
10	/E-F)*G)*H	+(-(	ABC*D
11	E-F)*G)*H	+(-(/	ABC*D
12	-F)*G)*H	+(-(/	ABC*DE
13	F)*G)*H	+(-(-	ABC*DE/
14	F)*G)*H	+(-(-	ABC*DE/
15	)*G)*H	+(-(-	ABC*DE/F
16	*G)*H	+(-	ABC*DE/F-
17	G)*H	+(-*	ABC*DE/F-
18	)*H	+(-*	ABC*DE/F-G
19	*H	+	ABC*DE/F-G*-
20	H	+*	ABC*DE/F-G*-
21	End	+*	ABC*DE/F-G*-H
22	End	Empty	ABC*DE/F-G*-H*+

Python code for converting a given expression from Infix to Postfix notation:

```
class Conversion:
```

```
 # Constructor to initialize the class variables
def __init__(self, capacity):
    self.top        =        -1
    self.capacity = capacity
    self.array      =        []
    self.output = []
    self.precedence = {'+':1, '-':1, '*':2, '/':2, '^':3}

# check if the stack is empty
def isEmpty(self):
    if  self.top  ==  -1:
        return True
  else :
      return False

# Return the value of the top of the stack
def peek(self):
    return self.array[-1]

# Pop the element from the stack
def pop(self):
    if      not      self.isEmpty():
        self.top -= 1
        return      self.array.pop()
    else:
        return "$"

# Push the element to the stack
def push(self, op):
    self.top       +=       1
    self.array.append(op)

# A utility function to check is the given character is operand
def isOperand(self, ch):
    return ch.isalpha()

# Check if the precedence of operator is strictly less than top of stack or not
def notGreater(self, i):
    try:
        a = self.precedence[i]
        b  =  self.precedence[self.peek()]
        if a <= b :
          return True
```

```
            else :
                return    False
        except KeyError:
            return False


    # Function that converts given infix expression to postfix expression def
    infixToPostfix(self, exp):
        for i in exp:
            # If the character is an operand,
            # add it to output
            if        self.isOperand(i):
                self.output.append(i)

            # If the character is an '(', push it to stack elif
            i == '(':
                self.push(i)

            # If the scanned character is an ')', pop and #
            output from the stack until and '(' is found
            elif i == ')':
                while( (not self.isEmpty()) and self.peek() != '('):
                    a = self.pop()
                    self.output.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()

            # An operator is encountered
            else:
                while(not          self.isEmpty()          and          self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)

        # pop all the operator from the stack
        print("\nPostfix Notation : ")
        while      not      self.isEmpty():
            self.output.append(self.pop())
        print("".join(self.output))

# Program to test above function
exp    =    "a+b*(c^d-e)^(f+g*h)-i"
print("\n\nInfix    Notation    :",exp)
obj    =    Conversion(len(exp))
obj.infixToPostfix(exp)
```

Output :

Infix Notation : a+b*(c^d-e)^(f+g*h)-i
Postfix Notation :
abcd^e-fgh*+^*+i-

3.3.4 Evaluation of Expression given in Postfix form:

The program takes the input expression in postfix form. This expression is scanned character by character. If the character scanned is an operand, then first it is converted to a digit form and then it is pushed onto the stack. If the character scanned is a blank space, then it is skipped. If the character scanned is an operator, then the top two elements from the stack are retrieved. An arithmetic operation is performed between the two operands. The type of arithmetic operation depends on the operator scanned from the string s. The result is then pushed back onto the stack. These steps are repeated as long as the string s is not exhausted. Finally the value in the stack is the required result.

In the first example, the conversion of infix to postfix notation is discussed; nextherethe evaluationof postfix notation to obtain the resultant value is discussed.

Example:

Infix Notation :A * (B + C) * DPostfix Notation :A B C + * D *

Let's assume the values of A,B,C,D as 2,3,4,5 respectively.

Evaluate the postfix expression 2 3 4 + * 5 * to obtain the resultant value.

Step No	Input	Stack (grows toward left)
1	2	2
2	3	3 2
3	4	4 3 2

4	+	7 2
5	*	14
6	5	5 14
7	*	70

Notes:

- Step No 4: an operator is encountered, so 4 and 3 are popped, summed, then pushed back onto stack.
- Step No 5: operator * is current token, so 7 and 2 are popped, multiplied, pushed back onto stack.
- Step No 7: stack top holds correct value.
- Notice that the postfix notation has been created to properly reflect operator precedence. Thus, postfix expressions never need parentheses.

Pseudo-code:

```
EVALUATION ( POSTFIX )
Begin
Read          postfix
L=Length(postfix)
For i=1 to L
Do
        C=infix(i)
        If C is a number or Alphabet then
                # Get the value of C and store it in the stack Read
                n
                Push(n)
        Else if C= '*' ||' /'|| '+ '||' –' ||' % '
        Then
                Call EVAL(c)
        End if
End for
Result = pop()    # Print the result
Write Result
End
```

```
EVAL(CHAR C)
Begin
        x=pop()
        y=pop()
Switch(C)
Begin
case  '+':  z=x+y
case   '-':   z=x-y
case  '*':  z=x*y
case   '/':   z=x/y
case '%': z=x%y
End       Switch
push(z)
End EVAL
```

Python code for evaluating Postfix notation:

```
class Evaluate:
   def__init__(self,   capacity):
      self.top          =          -1
      self.capacity = capacity
      self.array = []

   def   isEmpty(self):
      if self.top == -1:
         return True
      else:
         return False

   def peek(self):
      return self.array[-1]

   def pop(self):
      if   not   self.isEmpty():
         self.top -= 1
         return      self.array.pop()
      else:
         return      "$"

   def push(self, op):
```

```
        self.top += 1
        self.array.append(op)

    def evaluatePostfix(self, exp):
        for i in exp:
            if i.isdigit():
                self.push(i)
            else:
                val1 = self.pop()
                val2 = self.pop()
                self.push(str(eval(val2 + i + val1)))
            return int(self.pop())

# Program to test above function
exp = "231*+9-"
print ("\nPostfix Notation : ",exp)
obj = Evaluate(len(exp))
print ("Postfix Evaluation : ",obj.evaluatePostfix(exp))
```

Output:

```
Postfix Notation : 231*+9-
Postfix Evaluation : 2
```

Examples

Infix Expression	**Prefix Expression**	**Postfix Expression**
(A + B) * C	* + A B C	A B + C *
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

Points to Remember

- *A stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end. This end is often known as top of stack. When an item is added to a stack, the operation is called push, and when an item is removed from the stack the operation is called pop. Stack is also called as Last-In-First- Out (LIFO) list.*
- *Stacks are implemented using Arrays or Linked Lists.*
- *The storage requirements of linked representation of stack with n elements is O(n) and the time required for all stack operations is given as O(1).*
- *All Recursive function calls are implemented using System Stack.*

Exercises

1. Check if given expression is balanced expression or not. For example, Input : ((()))Output: 1Input : ()((Output : -1
2. Find duplicate parenthesis in an expression : ((a+b)+((c+d)))
3. Evaluate given postfix expression : A B C * + D +
4. Decode the given sequence to construct minimum number without repeated digits.
5. Explain how to implement two stacks in one array A[1…n] in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in O(1) time.
   Stack Implementation using Linked List.
6. Design a stack which returns minimum element without using auxiliary stack.
7. Reverse a string without using recursion.
8. Reverse a string using stack data structure.
9. In order Tree Traversal | Iterative & Recursive.
10. Preorder Tree Traversal | Iterative & Recursive.
11. Post order Tree Traversal | Iterative & Recursive.
12. Check if two given binary trees are identical or not | Iterative & Recursive.
13. Reverse Level Order Traversal of Binary Tree.
14. Reverse given text without reversing the individual words.
15. Find all binary strings that can be formed from given wildcard pattern.

16. Depth First Search (DFS) | Iterative & Recursive Implementation.
17. Invert given Binary Tree | Recursive and Iterative solution.
18. Longest Increasing Subsequence.
19. Implement Queue using Stacks.
20. Design a stack with operations on middle element.
21. The Stock Span Problem.
22. Check for balanced parentheses in an expression.
23. Next Greater Frequency Element.
24. Number of NGEs to the right.
25. Maximum product of indexes of next greater on left and right.
26. The Celebrity Problem.
27. Iterative Tower of Hanoi.
28. Sorting array using Stacks.

Check your Understanding:

1. What is the best case time complexity of deleting a node in Singly Linked list?
   a) O (n)
   b) O (n^2)
   c) O (nlogn)
   d) O (1)

2. Recursion follows divide and conquer technique to solve problems. State True or False.

3. Consider you have a stack whose elements in it are as follows.
   5 4 3 2 << top
   Where the top element is 2.
   You need to get the following stack
   6 5 4 3 2 << top
   The operations that needed to be performed are (You can perform only push and pop):
   a) Push(pop()), push(6), push(pop())
   b) Push(pop()), push(6)

c) Push(pop()), push(pop()), push(6)

d) Push(6)

4. What does ‘stack overflow’ refer to?

   a) accessing item from an undefined stack

   b) adding items to a full stack

   c) removing items from an empty stack

   d) index out of bounds exception

5. Which of the following statement is incorrect with respect to evaluation of infix expression algorithm?

   a) Operand is pushed on to the stack

   b) If the precedence of operator is higher, pop two operands and evaluate

   c) If the precedence of operator is lower, pop two operands and evaluate

   d) The result is pushed on to the operand stack

6. The process of accessing data stored in a serial access memory is similar to manipulating data on a :

   a) n-ary tree

   b) queue

   c) Stack

   d) Array

7. Representation of data structure in memory is known as :

   a) Storage Structure

   b) File Structure

   c) Record Structure d)

   Abstract Data Type

8. The data structure required to check whether an expression contains balanced parenthesis is :

a) n-ary tree

b) queue

c) Stack

d) Array

9. Which of the following data structures can be used for parentheses matching?

a) n-ary tree

b) queue

c) priority queue

d) stack

10. What will be the output after performing these sequence of operations : push(20); push(4);

pop();pop();push(5);top();

a) 20

b) 4

c) stack underflow

d) 5

11. Which of the following real world scenarios would you associate with a stack data structure?

a) piling up of chairs one above the other

b) people standing in a line to be serviced at a counter

c) offer services based on the priority of the customer

d) tatkal Ticket Booking in IRCTC

12. What is the time complexity of pop() operation when the stack is implemented using an array?

a) O(1)

b) O(n)

c) O(logn)

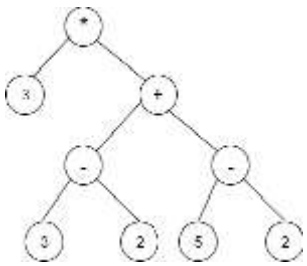
d) O(nlogn)

13. Array implementation of Stack is not dynamic, which of the following statements supports this argument?
    a) space allocation for array is fixed and cannot be changed during run-time
    b) user unable to give the input for stack operations
    c) a runtime exception halts execution d)
    improper program compilation

14. Evaluation of infix expression is done based on precedence of operators.
    a) True
    b) False

15. Of the following choices, which operator has the lowest precedence?
    a) ^
    b) +
    c) /
    d) #

16. From the given expression tree, identify the infix expression, evaluate it and choose the correct result.



    a) 5
    b) 10
    c) 12
    d) 16

17. What is the result of the following operation : Top (Push (S, X))

a. X
b. Null
c. S
d. None of the above

18. Evaluate the following statement using infix evaluation algorithm and choose the correct answer. 1+2*3-2
    a) 3
    b) 6
    c) 5
    d) 4

19. Convert the Expression ((a + B) * C - (d - E) ^ (f + G)) To Equivalent Prefix and Postfix Notations?

20. Evaluate the following infix expression using algorithm and choose the correct answer. a+b*c-d/e^f where a=1, b=2, c=3, d=4, e=2, f=2.
    a) 6 b) 8 c) 9 d)

21. Convert the infix to postfix for A-(B+C)*(D/E)
    a.ABC+DE/*-
    b.ABC-DE/*-
    c.ABC-DE*/-
    d.None of the above

Answers:

1. D
2. True
3. A
4. B

5. B
6. C
7. D
8. C
9. D
10. D
11. A
12. A
13. A
14. A
15. B
16. C
17. A
18. C
19. Prefix Notation: ^ – * +ABC – DE + FG

    Postfix Notation: AB + C * DE – – FG + ^

20. A
21. A

Case Study:

I. Design a stack that supports push, pop, top, and retrieving the minimum element in constant time. push(x) -- Push element x onto stack.• pop() -- Removes the element on top of the stack. top() -- Get the top element. getMin() -- Retrieve the minimum element in the stack.

II. Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

Note that an empty string is also considered valid.

1. Input: "()"Output: true
2. Input: "()[]{}" Output: true
3. Input: "(]" Output: false
4. Input: "([)]" Output: false
5. Input: "{[]}" Output: true

III. Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:

a. For any array, rightmost element always has next greater element as -1.
b. For an array which is sorted in decreasing order, all elements have next greater element as -1.
c. For the input array [4, 5, 2, 25], the next greater elements for each element are as follows :

Element	Next Greater
4	5
5	25
2	25
25	-1

IV. Find the largest rectangular area possible in a given histogram where the largest rectangle can be made of a number of contiguous bars. For simplicity, assume that all bars have same width and the width is 1 unit.

For example, consider the following histogram with 7 bars of heights {6, 2, 5, 4, 5, 1, 6}. The largest possible rectangle possible is 12 (see the below figure, the max area rectangle is highlighted in red).

V. Given a set of time intervals in any order, merge all overlapping intervals into one and output the result which should have only mutually exclusive intervals. Let the intervals be represented as pairs of integers for simplicity.

For example, let the given set of intervals be {{1,3}, {2,4}, {5,7}, {6,8} }. The intervals {1,3} and {2,4} overlap with each other, so they should be merged and become {1, 4}. Similarly {5, 7} and {6, 8} should be merged and become {5, 8}.

VI. Length of the longest valid substring

Given a string consisting of opening and closing parenthesis, find length of the longest valid parenthesis substring.

Examples:

Input : (((
Output : 2
Explanation : ()

Input:)()())
Output : 4
Explanation: ()()

Input: ()(()))))
Output: 6
Explanation: ()(())

VII. Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Example: Input: [0,1,0,2,1,0,1,3,2,1,2,1]

Output: 6.

VIII. The Stock Span Problem

The stock span problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days. The

span Si of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day. For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7
days are {1, 1, 1, 2, 1, 4, 6}

IX. The Celebrity Problem

In a party of N people, only one person is known to everyone. Such a person may be present in the party, if yes, (s)he doesn't know anyone in the party. We can only ask questions like "does A know B? ". Find the stranger (celebrity) in minimum number of questions. We can describe the problem input as an array of numbers/characters representing persons in the party. We also have a hypothetical function HaveAcquaintance(A, B) which returns true if A knows B, false otherwise. How can we solve the problem?
MATRIX = [[0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 0, 0], [0, 0, 1, 0]]

+

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

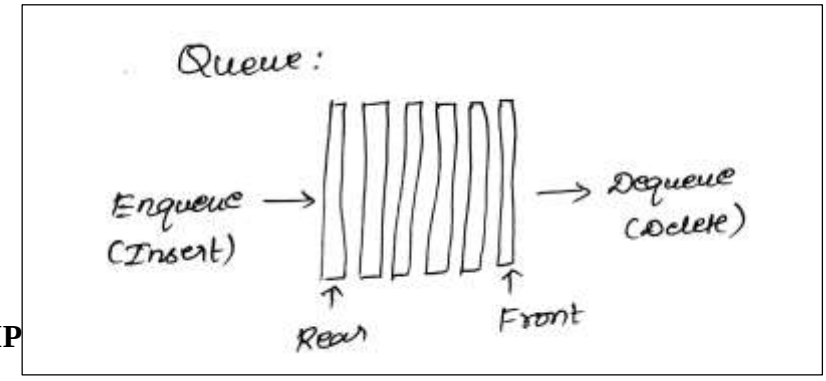
UNIT – IV -Data Structures – SCSA1203

UNIT-IV

Basic Queue Operations - Representation of a Queue using array - Applications of Queues - Round robin Algorithm - Enqueue - Dequeue - Circular Queues - Priority Queues

4.0 QUEUE

A Queue is a **linear structure** which follows a particular order in which the operations are performed. It is an **abstract data structure** similar to stack.
Queue is consisting of two ends such as,
i)**Enqueue** – Using this we can insert the data
ii)**Dequeue** – Using this we can delete the data
It follows two methodology consists of **FIFO** – First In First Out and **LILO** – Last In Last Out
It maintains two pointers consists of **FRONT** and **REAR**



4.1 EXAMP

- A queue of people at ticket-window: The person who comes first gets the ticket first. The person who is coming last is getting the tickets in last. Therefore, it follows first-in-first-out (FIFO) strategy of queue.
- Luggage checking machine: Luggage checking machine checks the luggage first that comes first. Therefore, it follows FIFO principle of queue.
- Patients waiting outside the doctor's clinic: The patient who comes first visits the doctor first, and the patient who comes last visits the doctor last. Therefore, it follows the first-in-first-out (FIFO) strategy of queue.

Fig 4.2 Schematic representation of a queue

4.2 IMPLEMENTATION OF QUEUE

The queue will be implemented in two ways,

i) Array
ii) Linked list
iii) Stack

BASIC OPERATIONS IN QUEUE

i) peek() – Front
ii) isfull() – Check queue is full or not
iii) isempty() – Check queue is empty or not
iv) Enqueue – Add at rear position
v) Dequeue – Add at front position

1. Peek()

It helps to access or view front element of the queue.

```
Algorithm peek (front, queue[])
    1. If front = -1
    2.     Return 0
    3. Else
    4.     Return queue[front]
```

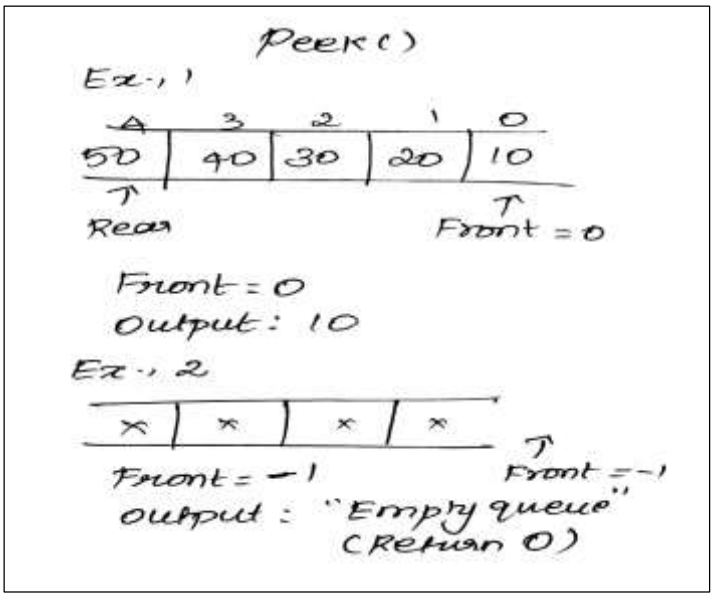


Fig. 4.3 Example for Peek() operation

2. Isfull()

It is used to check whether the queue is full or not.

```
Algorithm isfull (rear, queue[])
    1. If rear = MAX_SIZE
    2.     Return TRUE
    3. Else
    4.     Return FALSE
```

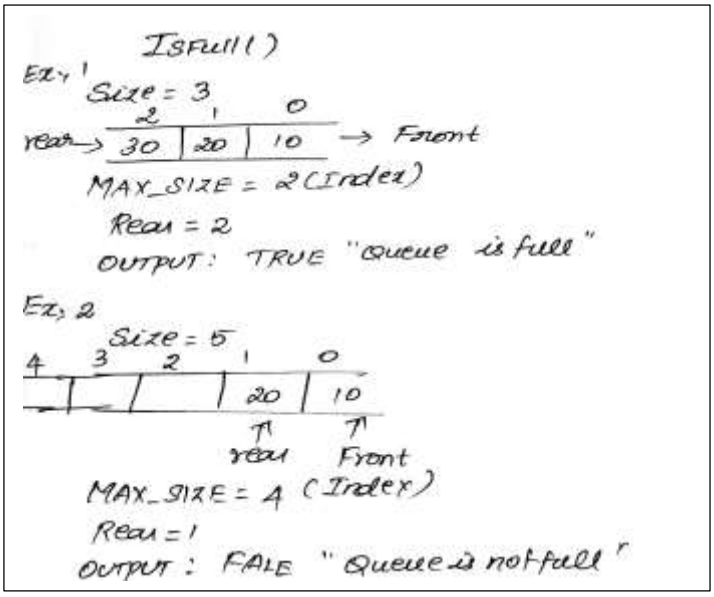


Fig. 4.4 Example for IsFull() operation

3. Isempty()

It helps to check whether the queue is empty or not.

```
Algorithm isempty (queue[], front)
    1. If front <=-1
    2.     Return TRUE
    3. Else
    4.     Return FALSE
```

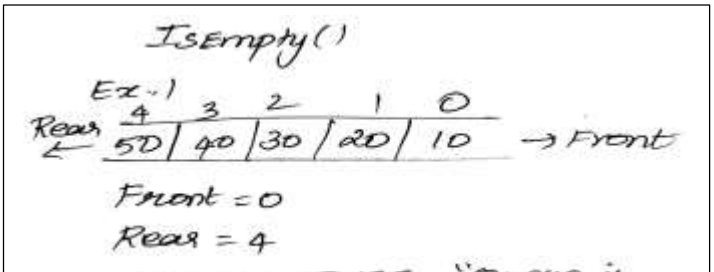


Fig. 4.5 Example for IsEmpty() operation

4. Enqueue:

The operation of inserting an element inside the queue at rear position called enqueue.

```
Algorithm enqueue (data, front, rear, queue[])
    1. If isfull (queue[])
    2.     Return OVERFLOW
    3. Else
    4.     Rear = Rear + 1
    5. queue[Rear] = data
```

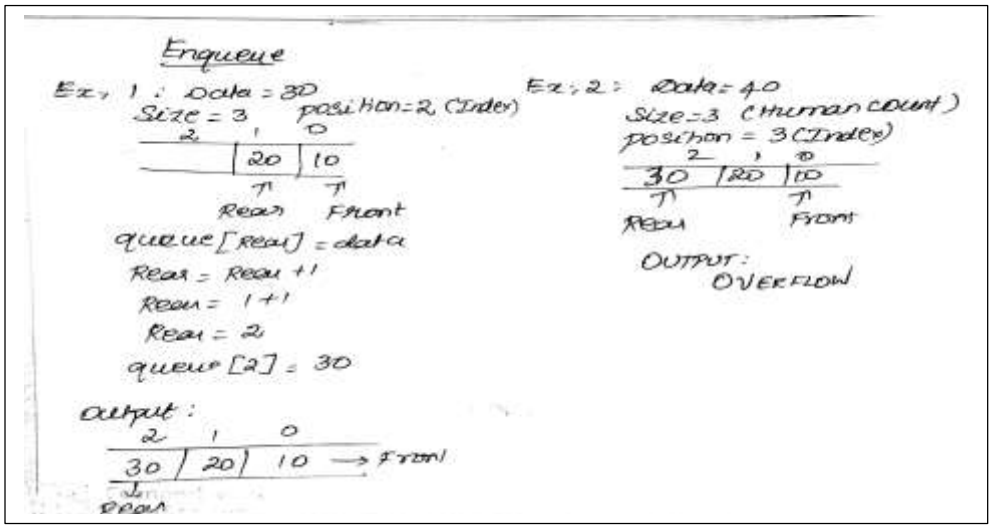


Fig. 4.6 Example for Enqueue() operation

5. Dequeue:

The operation of deleting an element from the queue at front position called dequeue.

```
Algorithm dequeue (data, front, rear, queue[])
    1. If isempty (queue[])
    2.     Return UNDERFLOW
    3. Else
    4.     Data = queue[front]
    5.     Front = Front + 1
```

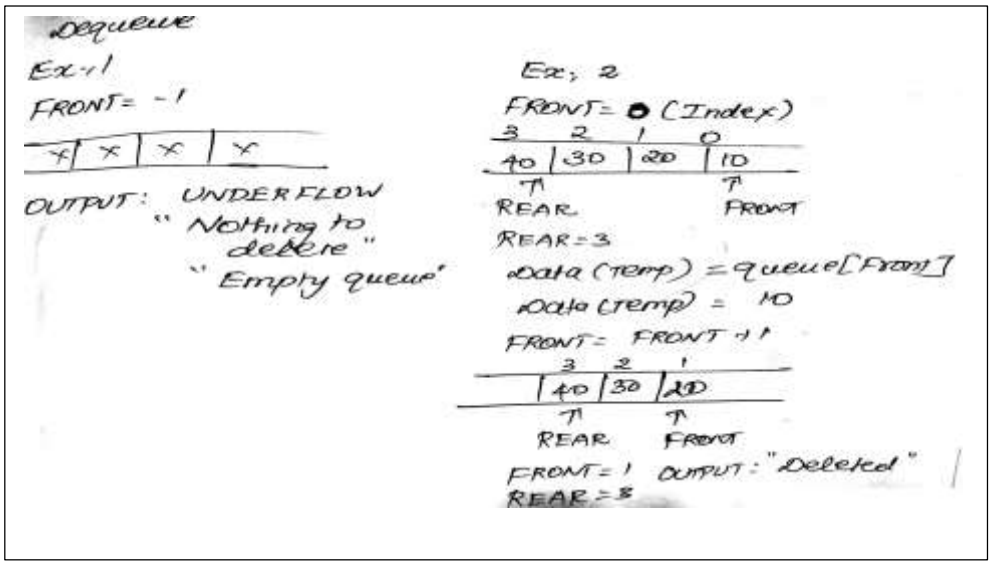


Fig. 4.7 Example for Dequeue() operation

4.3 REPRESENTATION OF QUEUE USING ARRAY

Queue can be implemented using array by restricting the array to follow First In First Out while inserting, deleting and displaying an element.
(For enqueue and dequeue refer previous topic)

4.3.1 Display

The operation is used to display all the element present in the queue.

```
Algorithm display (queue[], front, rear)
    1. If rear == -1
    2.     Return 0
    3. Else
    4.     For i = front to rear
```

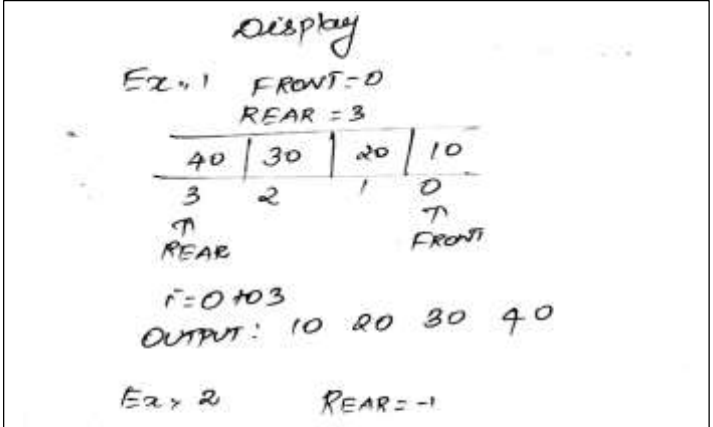


Fig. 4.8 Example for Display() operation

4.4 APPLICATION OF QUEUES

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.
4. In recognizing palindrome.
5. In shared resources management.
6. Keyboard buffer.

4.5 ROUND ROBIN ALGORITHM: (For algorithm circular queue)

Round robin algorithm is a CPU scheduling algorithm that is effectively used to schedule or order the processes inside the queue. The process residing in CPU may be in one of the five states given below,

1. **Stack** – Process creation
2. **Ready** – Process in ready queue
3. **Running** – Process running currently
4. **Waiting** – Process in waiting queue
5. **Terminate** – Process completed

In Round robin algorithm each process can reside in the CPU for only particular amount of time called **time quantum**.

Each process after completing its **burst time** fully will come out of the queue, until that it will be lined inside the

queue again and again in a circular manner.

Round Robin algorithm is an application for circular queue.

Example: **Given are the processes with arrival time and burst time. Find the waiting time and turnaround time.**

Time quantum = 2 seconds

PROCESS	**ARRIVAL TIME**	**BURST TIME**
P1	0	4
P2	1	5
P3	2	2
P4	3	3
P5	4	1

4.5.1 SOLUTION:

PROCESS NO.	**ARRIVAL TIME(AT)**	**BURST TIME(BT)**	**REMAINING BURST TIME(RBT)**			**COMPLETION TIME(CT)**	**WAITING TIME(WT)** **(CT-BT)**	**TURN ARROUND TIME (TAT)** **(CT-AT)**
P1	0	4	2	2	0	11	7	11
P2	1	5	2	2	1	15	10	14
P3	2	2	2	0	0	6	4	4

P4	3	3	2	1	0	14	11	11
P5	4	1	1	0	0	9	8	5

4.5.2 Grant Chart:

P1	**P2**	**P3**	**P4**	**P5**	**P1**	**P2**	**P4**	**P2**

0 2 4 6 8 9 11 13 14

Where 0,2,4,6,8,9,11,13,14,15 represents the time taking for each process under the condition of time quantum = 2 sec.

4.6 CIRCULAR QUEUE

It is a linear data structure in which the operation are performed based on **FIFO** principle. In circular queue the **last position is connected back to the first position** to make circle. It is also called as **ring buffer**.

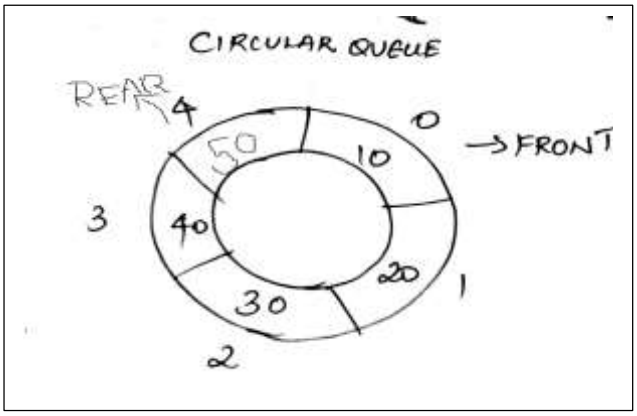


Fig. 4.9 Representation of a circular queue

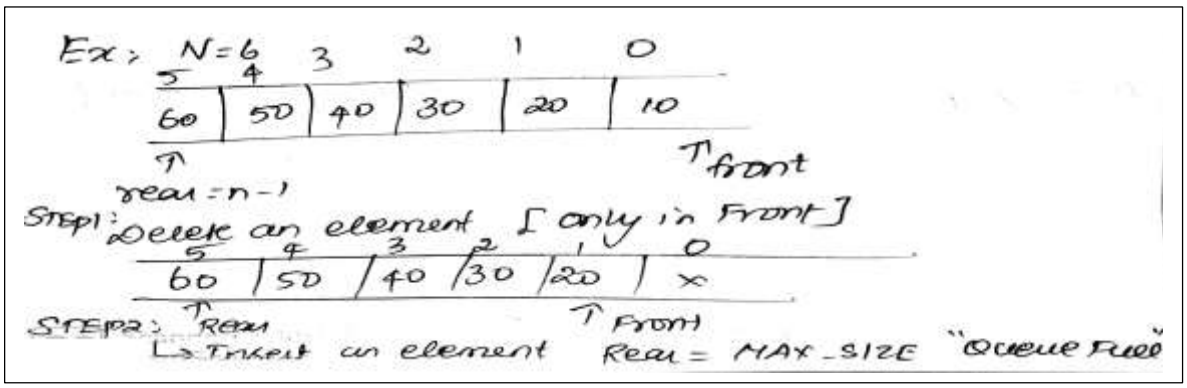


Fig. 4.10 Inserting an element into a circular queue

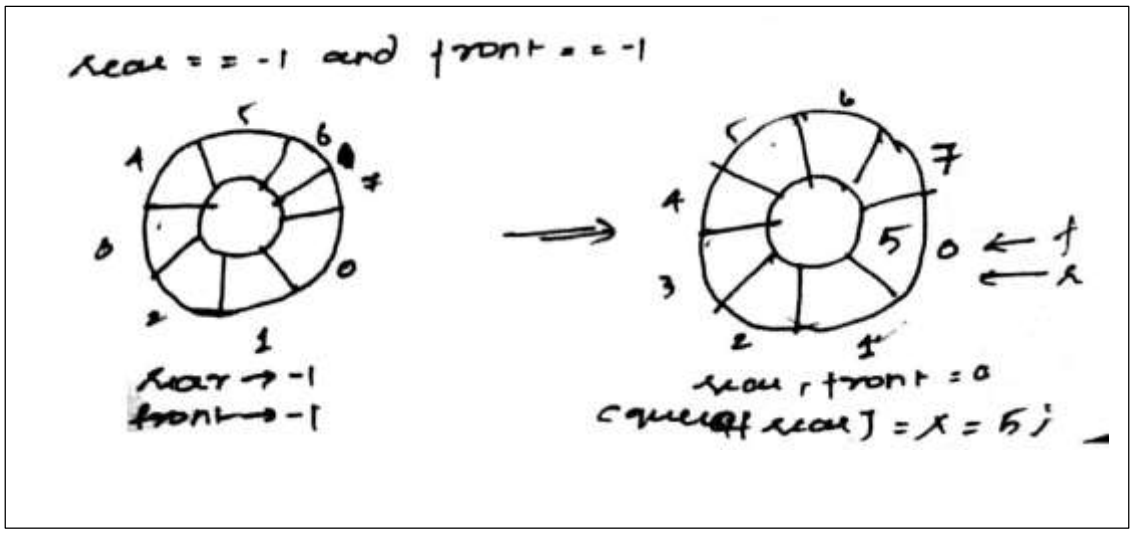
In normal queue we can insert element at rear position until queue becomes full. But cannot able to insert in front if the queue is full but front position having space. To overcome this shown in above diagram circular queue is implemented.

4.6.1 Operation on circular queue

1. Front: Get the Front item from queue.
2. Rear: Get the Rear item form queue.
3. Enqueue: (value)

   It is used to insert an element into the circular queue at rear position.

```
Algorithm Enqueue_circularqueue (enqueue [], rear, front)
1.  If rear >=  size – 1 and fron = 0
2.      Return OVERFLOW
3.  Else if rear == -1 and front == -1
4.      Front = 0
5.      Rear = 0
6.  Else if rear == sixe -1 and front !=0
7.      Rear = 0
8.  Else
9.      Rear = Rear + 1
10. Enqueue [rear] = x
```



3.1

Fig. 4.11 Operations into a circular queue

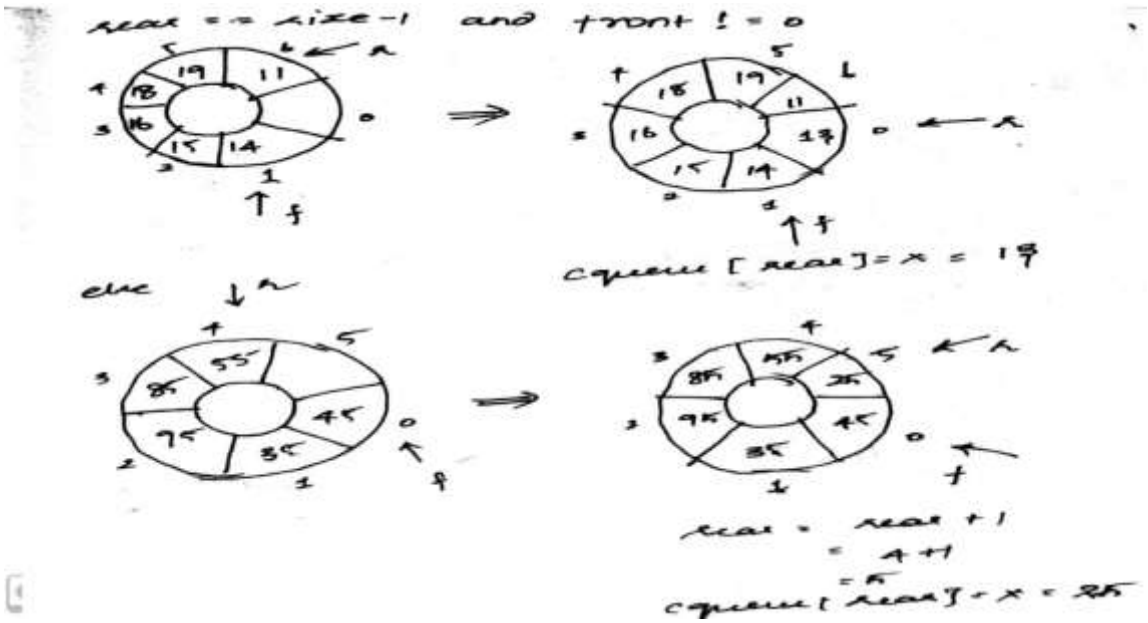


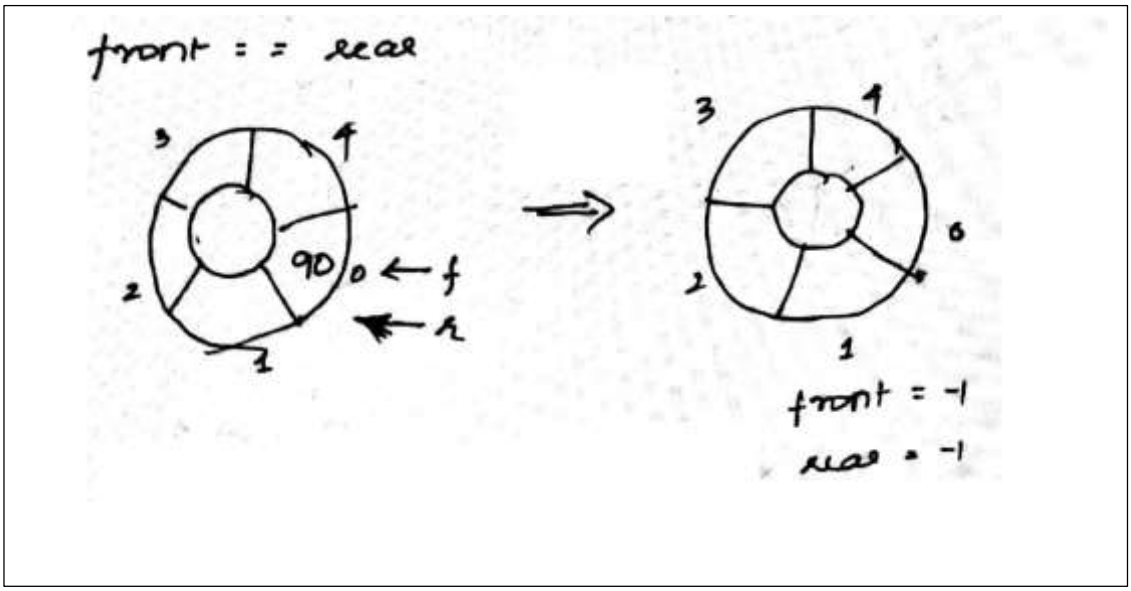
Fig. 4.12 Insertion in a circular queue

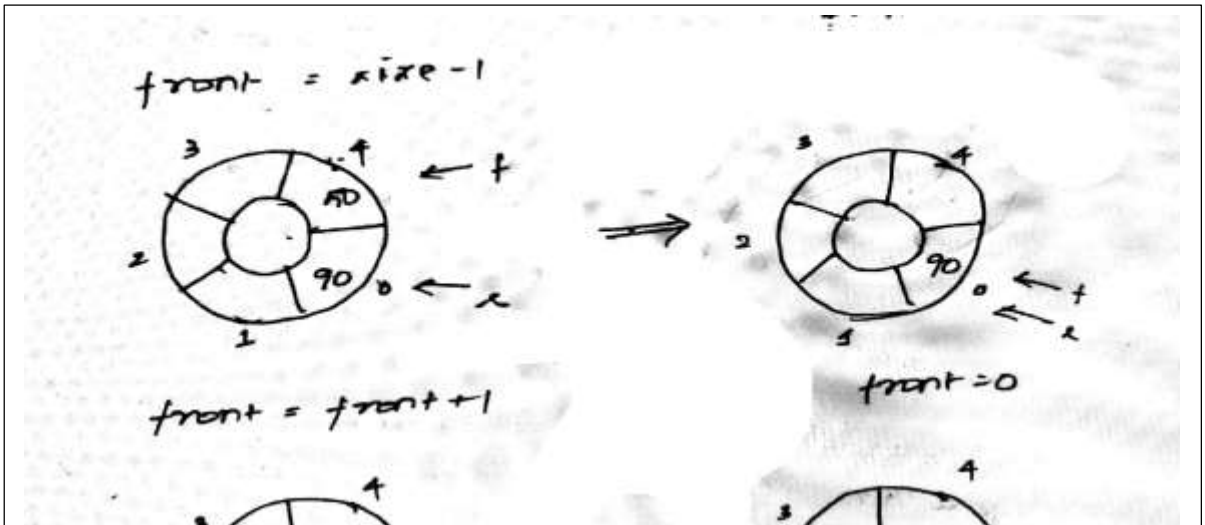
4. Dequeue:

This function is used to delete an element from the circular queue at front position.

```
Algorithm Dequeue_circularqueue (dequeue [], rear, front)
        1.  If front == -1
        2.      Return UNDERFLOW
        3.  Else
        4.      Return dequeue[front]
        5.      If front == rear
        6.          Front = -1
        7.          Rear = -1
```

3.1


front = = rear
3
4
90
0
f
r
2
1
front = -1
rear = -1


front = size - 1
3
4
90
90
0
f
r
2
1
front = front + 1
front = 0

3.1

Fig. 4.13 Deletion in a circular queue

4.7 PRIORITY QUEUE

Priority queue is an extension of queue data structure where each element has a priority associated with it. An element with high priority is dequeued before an element with low priority. If two elements have same priority, they are dequeued according to their order in the queue.

```
Algorithm enqueue (pqueue[], front, rear, X)
1. If rear >= SIZE -1
2.     Return OVERFLOW
3. If front == -1 and rear ==-1
4.     Front ++
5.     Rear ++
6. Pqueue [rear]=x
7. Else
8.     Call check (x)
9. Rear ++
```

```
Algorithm check (x, pqueue [], front, rear)
1. For I =0 to rear
2.     If data >= pqueue [i]
3.         For j=rear + 1 to 0
4. Pqueue [j] = pqueue [j-1]
5. Pqueue[i]=x
6.         Break
7. Pqueue[i] = x
```

Algorithm pdqueue (pqueue[], front, rear, X)

1. If front == -1 and rear == -1
2. Return UNDERFLOW
3. For I = 0 to rear
4. If data == pqueue [i]
5. For j=I to rear
6. Pqueue[j]=pqueue[j+1]
7. Rear – –

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – V -Data Structures – SCSA1203

UNIT-V

UNIT 5 SEARCHING AND SORTING TECHNIQUES

Basic concepts - List Searches using Linear Search - Binary Search - Fibonacci Search - Sorting Techniques - Insertion sort - Heap sort - Bubble sort - Quick sort - Merge sort - Analysis of sorting techniques.

SEARCHING

Searching and sorting are fundamental operations in computer science. Searching refers to the operation of finding the location of a given item in a collection of items. Sorting refers to the operations of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabetically, with character data.

Searching : *Searching is an operation which finds the location of a given element in a list. The search is said to be successful or unsuccessful depending on whether the element that is to be searched is found or not.*

The two standard searching techniques are:

- *Linear search*
- *Binary search.*

LINEAR SEARCH

Linear search is the simplest method of searching. In this method, the element to be found is sequentially searched in the list (Hence also called sequential search). This method can be applied to a sorted or an unsorted list. Hence, it is used when the records are not stored in order.

Principle : *The algorithm starts its search with the first available record and proceeds to the next available record repeatedly until the required data item to be searched for is found or the end of the list is reached.*

Algorithm :

```
Procedure LINEARSEARCH( A, N, K, P )

// A is the array containing the list of data items
// N is the number of data items in the list
// K is the data item to be searched
// P is the position where the data item is found

P € -1
Repeat For I = 0 to N -1
        Step 1 If A( I ) = K
        Then
                P € I
                Exit
                Loop
        End
```

In the above algorithm, A is the list of data items containing N data items. K is the data item, which is to be searched in A. P is a variable used to indicate, at what position the data item is found. If the data item to be searched is found then the position where it is found is stored in P. If the data item to be searched is not found then -1 is stored in P, which will indicate the user, that the data item is not found.

Initially P is assumed -1. The data item K is compared with each and every element in the list A. During this comparison, if K matches with a data item in A, then the position where the data item was found is stored in P and the control comes out of the loop and the procedure comes to an end. If K does not match with any of the data items in A, then P value is not changed at all. Hence at the end of the loop, if P is -1, then the number is not found.

Example:

K €Number to be searched : **40**

i = 0	i =1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7	i = 8	i = 9
45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[0]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[1]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[2]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[3]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[4]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[5]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[6]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K = A[7] € P = 7 : Number found at position 7

Advantages:

1. Simple and straight forward method.
2. Can be applied on both sorted and unsorted list.

Disadvantages:

1. Inefficient when the number of data items in the list increases.

BINARY SEARCH

Binary search method is very fast and efficient. This method requires that the list of elements be in sorted order. Binary search cannot be applied on an unsorted list.

Principle: *The data item to be searched is compared with the approximate middle entry of the list. If it matches with the middle entry, then the position is returned. If the data item to be searched is lesser than the middle entry, then it is compared with the middle entry of the first half of the list and procedure is repeated on the first half until the required item is found. If the data item is greater than the middle entry, then it is compared with the middle entry of the second half of the list and procedure is repeated on the second half until the required item is found. This process continues until the desired*

Algorithm:

```
Procedure BINARYSEARCH( A, N, K, P )

// A is the array containing the list of data items
// N is the number of data items in the list
// K is the data item to be searched
// P is the position where the data item

is found Lower € 0, Upper € N – 1, P

€ -1

While Lower ≤ Upper

        Mid € ( Lower + Upper )
        / 2 If K = A[Mid]
        Then
                P € Mid
                Exit Loop
        Else
                If K <
                A[Mid] Then
                        Upper €Upper - 1
                Else
                        Lower €Lower + 1
                End If
        End If
```

In Binary Search algorithm given above, A is the list of data items containing N data items. K is the data item, which is to be searched in A. P is a variable used to indicate, at what position the data item is found. If the data item to be searched is found then the position where it is found is stored in P. If the data item to be searched is not found then -1 is stored in P, which will indicate the user, that the data item is not found.

Initially P is assumed -1, lower is assumed 0 to point the first element in the list and upper is assumed as upper -1 to point the last element in the list. The mid position of the list is calculated and K is compared with A[mid]. If K is found equal to A[mid] then the value mid is assigned to P, the control comes out of the loop and the procedure comes to an end. If K is found lesser than A[mid], then upper is assigned mid – 1, to search only in the first half of the list. If K is found greater than A[mid], then lower is assigned mid + 1, to search only in the second half of the list. This process is continued until the element searched is found or the search interval becomes empty.

Example:

K €Number to be searched : **40**
U € Upper
L € Lower
M€ Mid

i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

1	22	35	40	43	56	75	83	90	98

L = 0 M = 4 U = 9
K < A[4] € U = 4 – 1 = 3

1	22	35	40	43	56	75	83	90	98

L = 0 M= 1 U = 3
K > A[1] €L = 1 + 1 = 2

1	22	35	40	43	56	75	83	90	98

L, M = 2 U = 3
K > A [2] € L = 2 + 1 = 3

1	22	35	40	43	56	75	83	90	98

L, M, U = 3
K = A[3] € P = 3 : Number found at position 3

Advantages:

1. Searches several times faster than the linear search.
2. In each iteration, it reduces the number of elements to be searched from n to n/2.

Disadvantages:

1. Binary search can be applied only on a sorted list.

Fibonacci Search

Given a sorted array arr[] of size n and an element x to be searched in it. Return index of x if it is present in array else return -1.

Fibonacci Search is a comparison-based technique that uses Fibonacci numbers to search an element in a sorted array.

Similarities with Binary Search:

- Works for sorted arrays
- A Divide and Conquer Algorithm.
- Has Log n time complexity.

Differences with Binary Search:

- Fibonacci Search divides given array into unequal parts
- Binary Search uses a division operator to divide range. Fibonacci Search doesn't use /, but uses + and -. The division operator may be costly on some CPUs.
- Fibonacci Search examines relatively closer elements in subsequent steps. So when the input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.

Algorithm:

Let the searched element be x. The idea is to first find the smallest Fibonacci number that is greater than or equal to the length of the given array. Let the found Fibonacci number be fib (m'th Fibonacci number). We use (m-2)'th Fibonacci number as the index (If it is a valid index). Let (m-2)'th Fibonacci Number be i, we compare arr[i] with x, if x is same, we return i. Else if x is greater, we recur for sub array after i, else we recur for sub array before i. Below is the complete algorithm. Let arr[0..n-1] be the input array and the element to be searched be x.

1. Find the smallest Fibonacci Number greater than or equal to n. Let this number be fibM [m'th Fibonacci Number]. Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)'th Fibonacci Number] and fibMm2 [(m-2)'th Fibonacci Number].
2. While the array has elements to be inspected:
    1. Compare x with the last element of the range covered by fibMm2
    2. If x matches, return index
    3. Else If x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two- third of the remaining array.
    4. Else x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate the elimination of approximately front one-third of the remaining array.
3. Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If match, return index.

SORTING

Practically, all data processing activities require data to be in some order. Ordering or sorting data in an increasing or decreasing fashion according to some linear relationship among data items is of fundamental importance.

Sorting : *Sorting is an operation of arranging data, in some given order, such as increasing or decreasing with numerical data, or alphabetically* 3.

Let A be a list of n elements A_1, A_2,…A_n in memory. Sorting A refers to the operation of rearranging the contents of A so that they are increasing in order (numerically or lexicographically), that is,

$$A_1 \leq A_2 \leq A_3 \leq \ldots \leq A_n$$

Sorting methods can be characterized into two broad categories:

- *Internal Sorting*
- *External Sorting*

Internal Sorting : *Internal sorting methods are the methods that can be used when the list to be sorted is small enough so that the entire sort can be carried out in main memory.*

The key principle of internal sorting is that all the data items to be sorted are retained in the main memory and random access in this memory space can be effectively used to sort the data items.

The various internal sorting methods are:

1. *Bubble Sort*
2. *Selection Sort*
3. *Insertion Sort*
4. *Quick Sort*
5. *Merge Sort*
6. *Heap Sort*

External Sorting : *External sorting methods are the methods to be used when the list to be sorted is large and cannot be accommodated entirely in the main memory. In this case some of the data is present in the main memory and some is kept in auxiliary memory such as hard disk, floppy disk, tape, etc.*

The key principle of external sorting is to move data from secondary storage to main memory in large blocks for ordering the data.

√ Criteria for the selection of a sorting method.

The important criteria for the selection of a sorting method for the given set of data items are as follows:

1. Programming time of the sorting algorithm
2. Execution time of the program
3. Memory space needed for the programming environment

√ Objectives involved in design of sorting algorithms.

The main objectives involved in the design of sorting algorithm are:

1. Minimum number of exchanges
2. Large volume of data block movement

This implies that the designed and desired sorting algorithm must employ minimum number of exchanges and the data should be moved in large blocks, which in turn increase the efficiency of the sorting algorithm.

INTERNAL SORTING

Internal Sorting: These are the methods which are applied on the list of data items, which are small enough to be accommodated in the main memory.

There are different types of internal sorting methods. The methods discussed here sort the data in ascending order. With a minor change we can sort the data in descending order.

BUBBLE SORT

This is the most commonly used sorting method. The bubble sort derives its name from the fact that the smallest data item bubbles up to the top of the sorted array.

Principle : The bubble sort method compares the two adjacent elements starting from the start of the list and swaps the two if they are out of order. This is continued up to the last element in the list and after each pass, a check is made to determine whether any interchanges were made during the pass. If no interchanges occurred, then the list must be sorted and no further passes

Algorithm:

```
Procedure BUBBLESORT( A, N )

// A is the array containing the list of data items
// N is the number of data items in

the list Last ∈ N– 1

While Last > 0
        Exch ∈ 0

        Repeat For I = 0 to Last
                Step 1 If A[I] >
                A[I+1]
                Then
                        A[I] ↔ A[I+1]
                        Exch ∈ 1
                End
        If End
        Repeat

        If Exch = 1
        Then
                Exit Loop
        Else
                Last ∈ Last – 1
        End
```

In Bubble sort algorithm, initially *Last* is made to point the last element of the list and *Exch* flag is assumed 0. Starting from the first element, the adjacent elements in the list are compared. If they are found out of order then they are swapped immediately and *Exch* flag is set to 1. This comparison is continued until the last two elements are compared. After this pass, the *Exch* flag is checked to determine whether any exchange has taken place. If no exchange has taken place then the control comes out of the loop and the procedure comes to an end as the list is sorted. If any exchange has taken place during the pass, the *last* pointer is decremented by 1 and the next pass is continued. This process continues until list is sorted.

Example:

N = 10 € Number of elements in the list
L €Points to last element (Last)

Pass 1

i = 0	**i =1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**	**i = 8**	**i = 9**
42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

Out of order €Swap L=9

23	**42**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

Out of order €Swap L=9

23	**42**	**11**	**74**	**65**	**58**	**94**	**36**	**99**	**87**

Out of order € Swap L=9

23	**42**	**11**	**65**	**74**	**58**	**94**	**36**	**99**	**87**

Out of order €Swap L=9

23	**42**	**11**	**65**	**58**	**74**	**94**	**36**	**99**	**87**

Out of order €Swap L=9

23	**42**	**11**	**65**	**58**	**74**	**36**	**94**	**99**	**87**

Out of order €Swap L=9

Pass 2

23	**42**	**11**	**65**	**58**	**74**	**36**	**94**	**87**	**99**

Out of order €Swap L=8

23	**11**	**42**	**65**	**58**	**74**	**36**	**94**	**87**	**99**

Out of order €Swap L=8

23	11	42	58	65	74	36	94	87	99

Out of order €Swap L=8

23	11	42	58	65	36	74	94	87	99

Out of order €Swap L=8

Pass 3

23	11	42	58	65	36	74	87	94	99

Out of order €Swap L=7

23	11	42	58	65	36	74	87	94	99

Out of order €Swap L=7

Pass 4

23	11	42	58	36	65	74	87	94	99

Out of order €Swap L=6

11	23	42	58	36	65	74	87	94	99

Out of order €Swap L=6

Pass 5

11	23	42	36	58	65	74	87	94	99

Out of order €Swap L=5

Pass 6

Adjacent numbers are compared up to L=4. But no swapping takes place. As there was no swapping taken place in this pass, the procedure comes to an end and we get a sorted list:

11	23	36	42	58	65	74	87	94	99

Advantages:

1. Simple and works well for list with less number of elements.

Disadvantages:

1. Inefficient when the list has large number of elements.
2. Requires more number of exchanges for every pass.

INSERTION SORT

The main idea behind the insertion sort is to insert the i^{th} element in its correct place in the i^{th} pass. Suppose an array A with n elements A[1], A[2],…A[N] is in memory. The insertion sort algorithm scans A from A[1] to A[N], inserting each element A[K] into its proper position in the previously sorted subarray A[1], A[2],..A[K-1].

Principle: *In Insertion Sort algorithm, each element A[K] in the list is compared with all the elements before it (A[1] to A[K-1]). If any element A[I] is found to be greater than A[K] then A[K] is inserted in the place of A[I}. This process is repeated till all the elements are sorted.*

Algorithm:

```
Procedure INSERTIONSORT(A, N)

// A is the array containing the list of data items
// N is the number of data items in

the list Last € N– 1

Repeat For Pass = 1 to Last Step 1
        Repeat For I = 0 to Pass – 1
                Step 1 If A[Pass] <
                A[I]
                Then
                        Temp € A[Pass]
                        Repeat For J = Pass -1 to I
                                Step -1 A[J +1] € A[J]
                        End Repeat
                        A[I] €Temp
                End
        If End
        Repeat
End Repeat
```

In Insertion Sort algorithm, *Last* is made to point to the last element in the list and *Pass* is made to point to the second element in the list. In every pass the *Pass* is

incremented to point to the next element and is continued till it reaches the last element. During each pass A[Pass] is compared all elements before it. If A[Pass] is lesser than A[I] in the list, then A[Pass] is inserted in position I. Finally, a sorted list is obtained.

For performing the insertion operation, a variable temp is used to safely store A[Pass] in it and then shift right elements starting from A[I] to A[Pass-1].

Example:

N = 10 € Number of elements in the list
L € Last
P €Pass

i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

P=1 *A[P] < A[0] €Insert A[P] at 0* L=9

23	**42**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

P=2 L=9

A[P] is greater than all elements before it. Hence No Change

23	**42**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

P=3 *A[P] < A[0] € Insert A[P] at 0* L=9

11	**23**	**42**	**74**	**65**	**58**	**94**	**36**	**99**	**87**

P=4 L=9

A[P] < A[3] € Insert A[P] at 3

11	**23**	**42**	**65**	**74**	**58**	**94**	**36**	**99**	**87**

P=5 L=9

A[P] < A[3] € Insert A[P] at 3

11	**23**	**42**	**58**	**65**	**74**	**94**	**36**	**99**	**87**

P=6 L=9

A[P] is greater than all elements before it. Hence No Change

11	**23**	**42**	**58**	**65**	**74**	**94**	**36**	**99**	**87**

P=7 L=9

A[P] < A[2] € Insert A[P] at 2

11	23	36	42	58	65	74	94	99	87

P=8 L=9

A[P] is greater than all elements before it. Hence No Change

11	23	36	42	58	65	74	94	99	87

P, L=9

A[P] < A[7] € Insert A[P] at 7

Sorted List:

11	23	36	42	58	65	74	87	94	99

Advantages:

1. Sorts the list faster when the list has less number of elements.
2. Efficient in cases where a new element has to be inserted into a sorted list.

Disadvantages:

1. Very slow for large values of n.
2. Poor performance if the list is in almost reverse order.

QUICK SORT

Quick sort is a very popular sorting method. The name comes from the fact that, in general, quick sort can sort a list of data elements significantly faster than any of the common sorting algorithms. This algorithm is based on the fact that it is faster and easier to sort two small lists than one larger one. The basic strategy of quick sort is to divide and conquer. Quick sort is also known as ***partition exchange sort.***

The purpose of the quick sort is to move a data item in the correct direction just enough for it to reach its final place in the array. The method, therefore, reduces unnecessary swaps, and moves an item a great distance in one move.

Principle: *A pivotal item near the middle of the list is chosen, and then items on either side are moved so that the data items on one side of the pivot element are smaller than the pivot element, whereas those on the other side are larger. The middle or the pivot element is now in its correct position. This procedure is then applied recursively to the 2 parts of the list, on either side of the pivot*

Algorithm:

```
Procedure QUICKSORT(A, Lower, Upper)

// A is the array containing the list of data items
// Lower is the lower bound of the  array
// Upper is the upper bound of the  array

If Lower ≥
Upper Then
        Return
End If

I = Lower +
1 J = Upper
+ 1

While I < J
        While A[I] <
                A[Lower] I €
                I + 1
        End While

        While A[J] >
                A[Lower] J €
                J – 1
        End While

        If I < J
        Then
```

QUICKSORT(A, Lower, J
– 1) QUICKSORT(A, J +
1, Upper)

In Quick sort algorithm, *Lower* points to the first element in the list and the *Upper* points to the last element in the list. Now I is made to point to the next location of *Lower* and J is made to point to the *Upper*. A[Lower] is considered as the *pivot* element and at the end of the pass, the correct position of the *pivot* element is fixed. Keep incrementing I and stop when A[I] > A[Lower]. When I stops, start decrementing J and stop when A[J] < A[Lower]. Now check if I < J. If so, swap A[I] and A[J] and continue moving I and J in the same way. When I meets J the control comes out of the loop and A[J] and A[Lower] are swapped. Now the element at position J is at correct position and hence split the list into two partitions: (A{Lower] to A[J-1] and A[J+1] to A[Upper]). Apply the Quick sort algorithm recursively on these individual lists. Finally, a sorted list is obtained.

Example:

N = 10 € Number of elements in the list
U € Upper
L€Lower

i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

L=0 I=0 U, J=9
Initially I=L+1 and J=U, A[L]=42 is the pivot element.

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

L=0 I=2 J=7 U=9
A[2] > A[L] hence I stops at 2. A[7] < A[L] hence J stops at
7 I < J € Swap A[I] and A[J]

42	**23**	**36**	**11**	**65**	**58**	**94**	**74**	**99**	**87**

L=0 J=3 I=4 U=9
A[4] > A[L] hence I stops at 4. A[3] < A[L] hence J stops at
3 I > J € Swap A[J] and A[L]. Thus 42 go to correct position.

The list is partitioned into two lists as shown. The same process is applied to these lists individually as shown.

€ List 1 € € List 2 €

11	**23**	**36**	**42**	**65**	**58**	**94**	**74**	**99**	**87**

L, J=0 I=1 U=2

(applying quicksort to list 1)

11	23	36	42	65	58	94	74	99	87

L, J=1 U, I=2

11	23	36	42	65	58	94	74	99	87

L=4 J=5 I=6 U=9

(applying quicksort to list 2)

11	23	36	42	58	65	94	74	99	87

L=6 I=8 U, J=9

11	23	36	42	58	65	94	74	87	99

L=6 J=8 U, I=9

11	23	36	42	58	65	87	74	94	99

L=6 U, I, J=7

Sorted List:

11	23	36	42	58	65	74	87	94	99

Advantages:

1. Faster than any other commonly used sorting algorithms.
2. It has a best average case behavior.

Disadvantages:

1. As it uses recursion, stack space consumption is high.

MERGE SORT

Principle: *The given list is divided into two roughly equal parts called the left and the right subfiles. These subfiles are sorted using the algorithm recursively and then the two subfiles are merged together to obtain the sorted file.*

Given a sequence of n elements A[1], ….A[N], the general idea is to imagine them split into two sets A[1],…A[N/2] and A[(N/2) + 1],…A[N]. Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of N elements. Thus this sorting method follows Divide and Conquer strategy.

Algorithm:

```
Procedure MERGE(A, low, mid, high)

// A is the array containing the list of

data items I € low, J € mid+1, K € low
While I Y mid and J Y high
        If A[I] < A[J]
        Then
                Temp[K]
                €A[I] I €I + 1
                K € K+1
        Else
                Temp[K]
                €A[J] J €J +
                1
                K €K + 1
        End
```

```
If I > mid
Then
        While J Y high
                Temp[K]
                €A[J] K € K
                + 1
                J €J + 1
        End While
Else
        While I Y mid
                Temp[K]
                €A[I] K € K +
                1
                I €I + 1
        End While
End If

Repeat for K = low to high step
```

```
Procedure MERGESORT(A, low, high)

// A is the array containing the list of

data items If low < high
Then
        mid € (low + high)/2
        MERGESORT(low, high)
        MERGESORT(mid + 1, high)
        MERGE(low, mid, high)
End If
End MERGESORT
```

The first algorithm MERGE can be applied on two sorted lists to merge them. Initially, the index variable I points to low and J points to mid + 1. A[I] is compared with A[J] and if A[I] found to be lesser than A[J] then A[I] is stored in a temporary array and I is incremented otherwise A[J] is stored in the temporary array and J is incremented. This comparison is continued until either I crosses mid or J crosses high. If I crosses the mid first then that implies that all the elements in first list is accommodated in the temporary array and hence the remaining elements in the second list can be put into the temporary array as it is. If J crosses the high first then the remaining elements of first list is put as it is in the temporary array. After this process we get a single sorted list. Since this method merges 2 lists at a time, this is called 2-way merge sort.

In the MERGESORT algorithm, the given unsorted list is first split into N number of lists, each list consisting of only 1 element. Then the MERGE algorithm is applied for first 2 lists to get a single sorted list. Then the same thing is done on the next two lists and so on. This process is continued till a single sorted list is obtained.

Example:

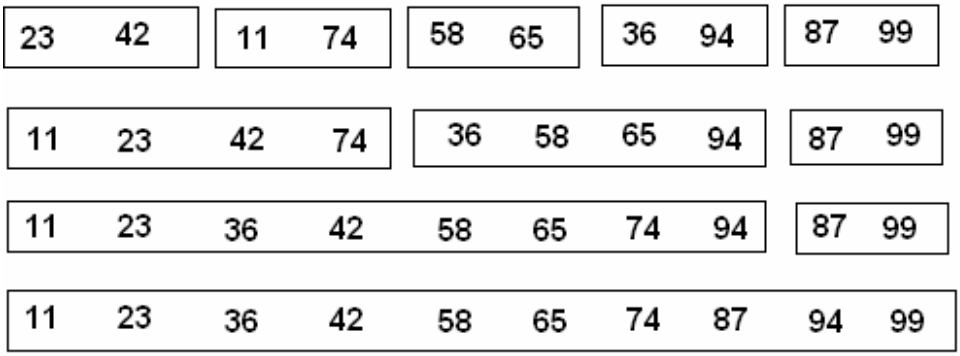
Let L € low, M€ mid, H € high

i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

42	23	74	11	65	58	94	36	99	87
U				M					H

In each pass the mid value is calculated and based on that the list is split into two. This is done recursively and at last N number of lists each having only one element is produced as shown.

Now merging operation is called on first two lists to produce a single sorted list, then the same thing is done on the next two lists and so on. Finally a single sorted list is obtained.



Advantages:

1. Very useful for sorting bigger lists.
2. Applicable for external sorting also.

Disadvantages:

1. Needs a temporary array every time, for storing the new sorted list.

HEAP SORT

Heap: *A Heap is a compete binary tree with the property that the value at each node is at least as large as (or as small as) the values at its children (if they exist). If the value at the parent node is larger than the values on its children then it is called a* ***Max heap*** *and if the value at the parent node is smaller than the values on its children then it is called the* ***Min heap****.*

- If a given node is in position I then the position of the left child and the right child can be calculated using **Left (L) = 2I** and **Right (R) = 2I + 1.**
- To check whether the right child exists or not, use the condition **R = N.** If true, Right child exists otherwise not.
- The last node of the tree is **N/2.** After this position tree has only leaves.

Principle: The Max heap has the greatest element in the root. Hence the element in the root node is pushed to the last position in the array and the remaining elements are converted into a max heap. The root node of this new max heap will be the second largest element and hence pushed to the last but one position in the array. This process is repeated till all the elements get sorted.

Algorithm:

```
Procedure WALKDOWN(A, I, N)

// A is the list of unsorted elements
// N is the number of elements in the array
// I is the position of the node where the walkdown procedure is to
be applied.

While I Y N/2
        L € 2I, R € 2I +
1 If A[L] > A[I]
Then
        M € L
Else
        M € I
End If
If A[R] > A[M] and R Y N
Then
        M € R
End If
If M [
I
Then
        A[I] ↔ A[M]
        I € M
Else
        Return
End If
```

```
Procedure HEAPSORT(A, N)

// A is the list of unsorted elements
// N is the number of elements in the array

Repeat For I = N/2 to 2 step -1
        WALKDOWN(A, I, N)
End Repeat
Repeat For J = N to 2
        Step -1
        WALKDOWN(A, 1,
        J) A[1] ↔ A[J]
```

The WALKDOWN procedure is used to convert a subtree into a heap. If this algorithm is applied on a node of the tree, then the subtree starting from that node will be converted into a max heap. In the above given WALKDOWN algorithm, the element at the given node is compared with its left and right child and is swapped with the maximum of that. During this process the element at the given node may walkdown to its correct position in the subtree. The procedure stops if the element at I reaches a leaf or it reaches its correct position.

In the HEAPSORT algorithm there are two phases. In the first phase the walkdown procedure is applied on each node starting from the last node at N/2 to node at position 2. The root node is not disturbed. During this first phase, the subtrees below the root satisfy the max heap property.

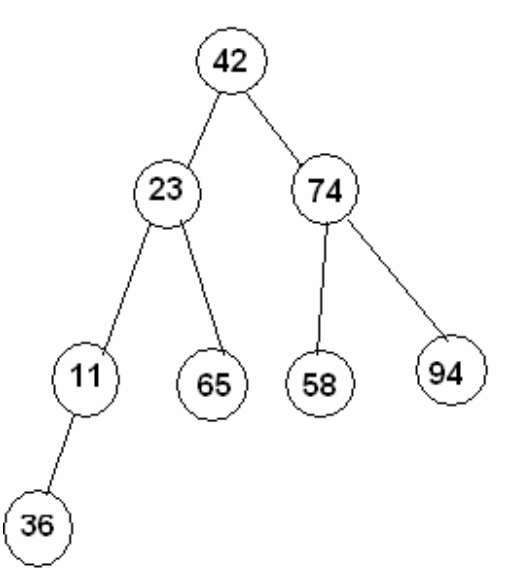
In the second phase of the sorting algorithm, the walkdown procedure is applied on the root node. After this pass the entire tree becomes a heap. The root node element and the last element are swapped and the last element is now not considered for the next pass. Thus the tree size reduces by one in the next pass. This process is repeated till we obtain a sorted list.

Example:

Given a list A with 8 elements:

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**

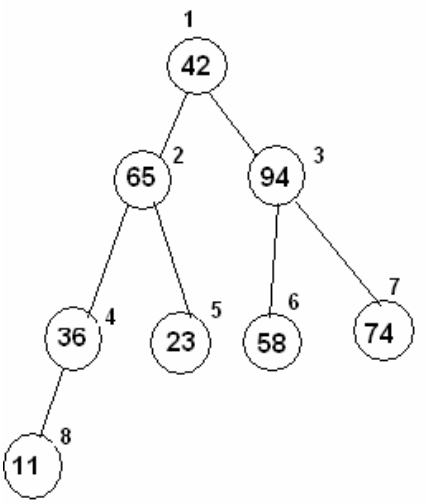
The given list is first converted into a binary tree as shown.



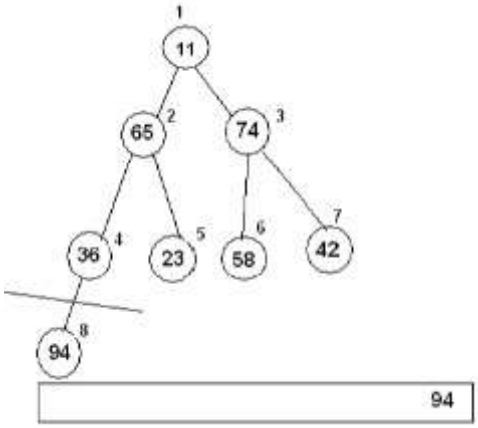
Binary tree

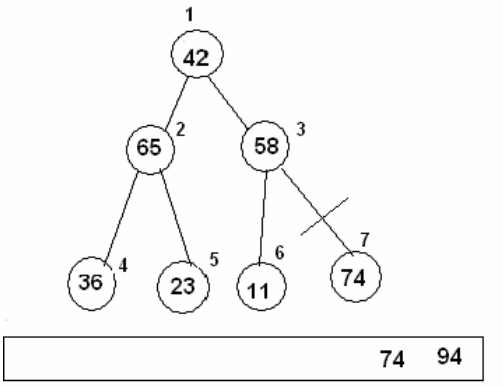
Phase 1:

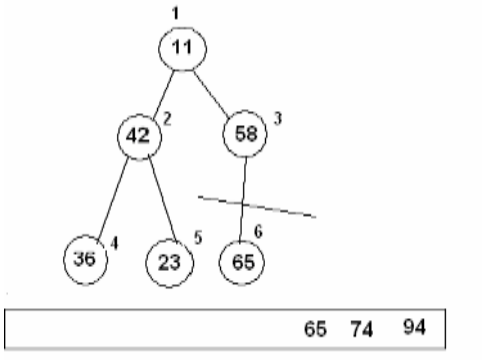
The rearranged tree elements after the first phase is

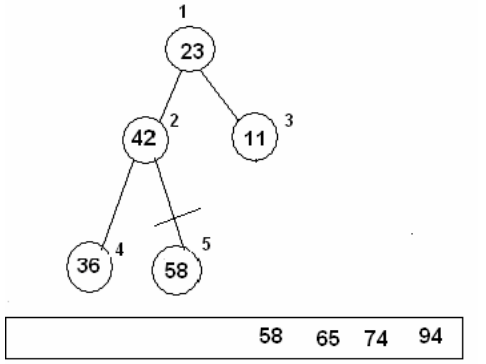


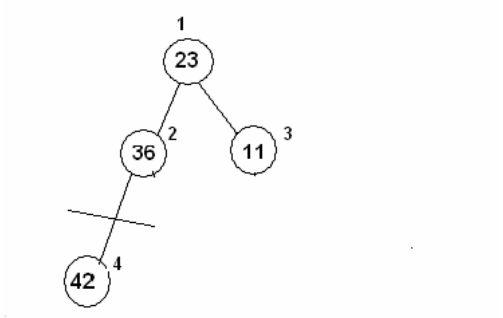
Phase 2:

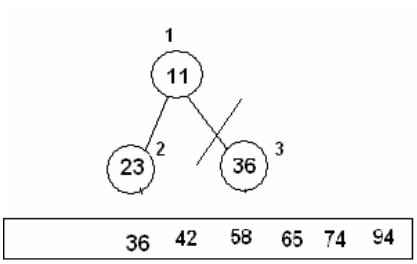


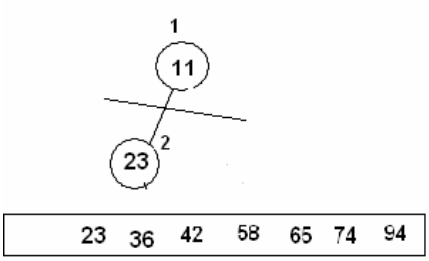


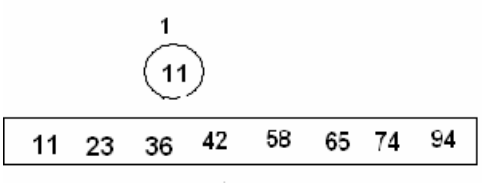












ANALYSIS OF SORTING TECHNIQUES

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)