GUDLAVALLERU ENGINEERING COLLEGE
(An Autonomous Institute with Permanent Affiliation to JNTUK, Kakinada)
Seshadri Rao Knowledge Village, Gudlavalleru – 521 356.

Department of Computer Science and Engineering

HANDOUT

on

DATA STRUCTURES

Vision

To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society.

Mission

- To impart quality education through well-designed curriculum in tune with the growing software needs of the industry.
- To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society.
- To serve our students by inculcating in them problem solving, leadership, teamwork skills and the value of commitment to quality, ethical behavior & respect for others.
- To foster industry-academia relationship for mutual benefit and growth

Program Educational Objectives

- Identify, analyze, formulate and solve Computer Science and Engineering problems both independently and in a team environment by using the appropriate modern tools.
- Manage software projects with significant technical, legal, ethical, social, environmental and economic considerations
- Demonstrate commitment and progress in lifelong learning, professional development, leadership and Communicate effectively with professional clients and the publiC)

HANDOUT ON DATA STRUCTURES

Class & Sem. : II B. Tech – I Semester Year : 2018-19

Branch : CSE Credits: 3

===

1. Brief History and Scope of the Subject

A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers. Data structures are used in almost every program or software system. Data structures provide a means to manage huge amounts of data efficiently, such as large databases and internet indexing services. Usually, efficient data structures are a key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

2. Pre-Requisites

- Knowledge of any programming language that supports pointers for referencing.

3. Course Objectives:

- To gain knowledge of linear and non-linear data structures.
- To familiarize with different sorting and searching techniques

4. Course Outcomes:

Upon successful completion of the course, the students will be able to

- demonstrate the working process of sorting (bubble, insertion, selection and heap) and searching (linear and binary) methods using a programming language.
- design algorithms to create, search, insert, delete and traversal operations on linear and non-linear data structures.
- evaluate the arithmetic expressions using stacks.

- choose appropriate collision resolution techniques to resolve collisions.
- compare array and linked list representation of data structures.

5. Program Outcomes:

Graduates of the Computer Science and Engineering Program will have an ability to

a. apply knowledge of computing, mathematics, science and engineering fundamentals to solve complex engineering problems.

b. formulate and analyze a problem, and define the computing requirements appropriate to its solution using basic principles of mathematics, science and computer engineering.

c. design, implement, and evaluate a computer based system, process, component, or software to meet the desired needs.

d. design and conduct experiments, perform analysis and interpretation of data and provide valid conclusions.

e. use current techniques, skills, and tools necessary for computing practice.

f. understand legal, health, security and social issues in Professional Engineering practice.

g. understand the impact of professional engineering solutions on environmental context and the need for sustainable development.

h. understand the professional and ethical responsibilities of an engineer.

i. function effectively as an individual, and as a team member/ leader in accomplishing a common goal.

j. communicate effectively, make effective presentations and write and comprehend technical reports and publications.

k. learn and adopt new technologies, and use them effectively towards continued professional development throughout the life.

l. understand engineering and management principles and their application to manage projects in the software industry.

6. Mapping of Course Outcomes with Program Outcomes:

	a	b	c	d	e	f	g	h	i	j	k
CO1		3									
CO2		3									
CO3		3									
CO4			3		2						
CO5					2						

7. Prescribed Text Books

a. Debasis samanta, Classic Data Structures, PHI, 2nd edition, 2011.

b. Richard F, Gilberg , Forouzan, Data Structures, 2nd edition, , Cengage

8. Reference Text Books

a. Seymour Lipschutz, Data Structure with C, TMH.

b. G. A. V. Pai, Data Structures and Algorithms, TMH, 2008.

c. Horowitz, Sahni, Anderson Freed, Fundamentals of Data Structure in C, University Press, 2nd edition

9. URLs and Other E-Learning Resources

a. https://www.courserA)org/learn/data-structures

b. http://www.studytonight.com/data-structures/

c. http://www.indiabix.com/technical/data-structures/

d. http://nptel.aC)in/courses/106102064/1

e. http://freevideolectures.com/Course/2279/Data-Structures-And-Algorithms/2#

10. Lecture Schedule / Lesson Plan

Topic	No. of Periods	
	Theory	**Tutorial**
UNIT – 1: Searching and Sorting		
Concepts of data structures, Overview of data structures	1	1
Linear search	1	
Binary search	1	
Internal sorting: Basic concept	1	1
Bubble sort	1	
Insertion sort	1	
Selection sort	1	
	7	**2**
UNIT –2: Linked Lists		
Linked Lists- Basic concepts	1	1
Single linked list-operations	4	
Circular linked list	2	
Double linked list	4	1
	11	**2**
UNIT – 3: Stacks and Queues		
Stack introduction, Array and Linked List representations of stack	2	1
Operations on stacks using array and linked list	4	
Evaluation of arithmetic expression	3	
Queue introduction, Array and Linked List representations of queue	2	1
Operations on queues using array and linked list	3	
Circular queue introduction	1	
	15	**2**
UNIT – 4: Trees		
Basic tree concepts, Properties	2	1
Representation of Binary Trees using Arrays, linked lists	1	
Binary Tree Traversals (recursive)	1	
Binary search trees: Basic concepts, Search, insertion operations	2	1
Deletion Operation (Examples only)	1	
Creation of binary search tree from in-order and pre (post) order	1	
	8	**2**
UNIT – 5: Heap Trees and Graphs		
Heap Trees: Basic Concept, Operations	2	1
Graphs-Basic concepts, Representations of graphs	2	1
Graph traversals Breadth First Search (BFS), Depth	4	

First Search (DFS)		
	8	**2**
UNIT - 6: Hashing		
Hashing: Basic concepts, hashing functions (division method, multiplication method)	3	1
Collision resolution techniques- open hashing	1	1
Closed hashing (Linear Probing, Quadratic Probing, Double Hashing)	3	
	7	**2**
Total Number of Hours	**56**	**12**

11. Seminar Topics: -

UNIT – I

Sorting and Searching

Objective:

- To impart the concepts of searching and sorting.

Syllabus:

Sorting and Searching

Introduction- Concept of data structures, overview of data structures.

Searching: Linear search, Binary search.

Sorting (Internal): Basic concepts, sorting by: insertion (insertion sort), selection (selection sort), exchange (bubble sort).

Introduction to external searching and sorting

Learning Outcomes:

At the end of the unit student will be able to:

- demonstrate the working process of sorting (bubble, insertion, selection and heap) and searching (linear and binary) methods using a programming language.

Learning Material

Searching:

Is a process of verifying whether the given element is available in the given set of elements or not. Types of Searching techniques are:

1. Linear Search
2. Binary Search
3. Fibonacci Search

1. Linear Search

In linear search, search process starts from starting index of array. i.e. **0th** index of array and end's with ending index of array. i.e. **(n-1)th** index. Here searching is done in Linear fashion (Sequential fashion).

Algorithm Linearsearch(a<array>, n, ele)

Input: ***a*** is an array with *n* elements, *ele* is the element to be searcheD)

Output: Position of required element in array, if it is available.

```
1. found = 0
2. i =0
3. while(i < n)
        a) if(ele == a[i])
```

```
                i) found = 1
                ii) print( element found at ith position)
                iii) break
            b) end if
            c) i = i +1
    4. end loop
    5. if( found == 0)
            a) print( required element to be search is not available)
    6. end if
```

End Linearsearch

Algorithm Linearsearch_Recurssion(a<array>, i, n, ele)

Input: *a* is an array with n elements, *ele* is the element to be searched, *i* is starting index of array and *n* is the ending index.

Output: Position of required element in array, if it is available.

```
    1. found = 0
    2. if(i<n)
            a) if(a[i] == ele)
                i) found = 1
                ii) print( element found at ith position)
            b) end if
            c) Linearsearch_Recurssion(a, i+1, n, ele)
    4. end if
    5. if( found == 0)
            a) print( required element to be search is not available)
    6. end if
```

End Linearsearch_Recurssion

2. Binary Search

The input to binary search must be in *ascending order*. i.e. set of elements be in ascending order.

Searching process in Binary search as follows:

- First, the element to be search is compared with middle element of array.

- If the required element to be searched is equal to middle element of array then Successful Search.
- If the required element to be searched is *less than* the middle element of array, then search in LEFT side of the midpoint of the array.
- If the required element to be search is *greater than* middle element of array, then search in RIGHT side of the midpoint of the array.

Algorithm Binarysearch(a<array>, n, ele)

Input: *a* is an array with *n* elements, *ele* is the element to be searcheD)

Output: Position of required element in array, if it is available.

```
1. found = 0
2. low = 0
3. high = n-1
4. while(low <= high)
      a) mid = (low+high )/2.
      b) if(ele == a[mid])
            i) print(required element was found at mid position)
            ii) found = 1
            iii) break
      c) else if(ele < a[mid])
            i) high = mid - 1
      d) else if(ele > a[mid])
            i) low= mid + 1
      e) end if
5. end if
6. if(found == 0)
      a) print(required element is not available)
7. end if
```

End Binarysearch

Algorithm Binarysearch_Recursion(a<array>,ele, low, high)

Input: ***a*** is array with n elements, *ele* is the element to be searched, *low* is starting index, *high* is ending index of array.

Output: Position of required element in array, if it is available.

```
1. found = 0
2. if(low <= high)
        a) mid = (low+high )/2.
        b) if(ele == a[mid])
                i) print(required element was found at mid position)
                ii) found = 1
        c) else if(ele < a[mid])
                i) Binarysearch _Recursion(a,ele,low,mid-1)
        d) else if(ele > a[mid])
                i) Binarysearch_Recursion(a,ele,mid+1,high)
        e) end if
3. end if
4. if(found == 0)
        a) print(required element to be search is not available)
5. end if
```

End Binarysearch_Recursion

Sorting: Sorting means arranging the elements either in ascending or descending order.

There are two types of sorting.

1. Internal Sorting.
2. External Sorting.

1. Internal Sorting: For sorting a set of elements, if we use only primary memory (Main memory), then that sorting process is known as internal sorting. i.e. internal sorting deals with data stored in computer memory.

2. External Sorting: For sorting a set of elements, if we use both primary memory (Main memory) and secondary memory, then that sorting process is known as external sorting. i.e. external sorting deals with data stored in files.

Different types of sorting techniques.

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Merging sort
5. Quick sort
6. Radix sort

1. Bubble sort:

- In bubble sort for sorting n elements, we require (n-1) passes (or) iterations and in each pass compare every element with its successor. i.e. ith index element will compare with (i+1)th index element, if they are not in ascending order, then swap them.
- Here for each pass, the largest element is moved to height index position of array to be sort.

Process:

1. In pass1, a[0] and a[1] are compared, then a[1] is compared with a[2], then a[2] is compared with a[3] and so on. Finally a[n-2] is compared with a[n-1]. Pass1 involves (n-1) comparisons and places the biggest element at the highest index of the array to be sorteD)
2. In pass2, a[0] and a[1] are compared, then a[1] is compared with a[2], then a[2] is compared with a[3] and so on. Finally a[n-3] is compared with a[n-2]. Pass2 involves (n-2) comparisons and places the biggest element at the highest index of the array to be sorteD)
3. In pass (n-1), a[0] and a[1] are compareD) After this step all the elements of the array are arranged in ascending order.

Eg: sort the elements 72, 85, 4, 32 and 16 using Bubble sort.

	0	**1**	**2**	**3**	**4**	
Pass 1	72	85	24	32	16	
	72	85	24	32	16	(0, 1) No Exchange
	72	24	85	32	16	(1, 2) Exchange
	72	24	32	85	16	(2, 3) Exchange
	72	24	32	16	**85**	(3, 4) Exchange
Pass 2	72	24	32	16	85	
	24	72	32	16	85	(0, 1) Exchange
	24	32	72	16	85	(1, 2) Exchange
	24	32	16	**72**	**85**	(2, 3) Exchange

Pass 3	24	32	16	72	85	
	24	32	16	72	85	(0, 1) No Exchange
	24	16	**32**	**72**	**85**	(1, 2) Exchange
Pass 4	24	16	32	72	85	
	16	**24**	**32**	**72**	**85**	(0, 1) Exchange

Sorted List is **16** **24** **32** **72** **85**

Algorithm Bubblesort(a<array>, n)

Input: *a* is an array with n elements to be sort.

Output: array elements in ascending order.

```
1. for( i = 0 to n-1)
        a) for( j = 0 to n-i-1)
                i) if ( a[j] > a[j+1] )
                        A) t = a[j]
                        B) a[j] = a[j+1]
                        C) a[j+1] = t
                ii) end if
        b) end j for loop
2. end i for loop
```

End Bubblesort

2. Insertion sort

- In the insertion sort, initially consider the 0th index element value as only sorted element, and then take remaining elements of the given set one by one.
- For every pass, compare unsorted elements one by one with sorted list.
- If sorted list value is GREATER than the unsorted element value, then **move** sorted list element to **next index.**
- Continue the above moving process up to sorted list element value is LESS than given unsorted element value.
- Continue the above process for all the elements in the given array.

Algorithm insertionsort(a<array>, n)

Input: a is array with n elements to be sorteD)

Output: array elements in ascending order.

```
1. i = 1
2. while( i < n)
        a) x = a[i]                    /* x is unsorted element */
        b) j = i -1
        c) while( j >= 0 && a[j] > x )
                i) a[j+1] = a[j]
                ii) j = j -1
        d) end loop
        e) a[j+1] = x
        f) i = i + 1
3. end loop
```

End insertionsort

Eg: sort the elements 15,10, 8, 46, 32 using Insertion sort. Where **x** is an unsorted element

	0	**1**	**2**	**3**	**4**		**x**
Pass 1	15	10	8	46	32	Sorted *ele* > unsorted *ele*. TRUE. i.e. **15>10**. So move 15 from 0th index to 1st index	10
		15	8	46	32		
	10	15	8	46	32	Last index is 0th index. So place **x** value in 0th index	

	0	**1**	**2**	**3**	**4**		**x**
Pass 2	**10**	**15**	8	46	32	Sorted *ele* > unsorted *ele*. TRUE. i.e. 15 > 8 So move 15 from 1st index to 2nd index Sorted *ele* > unsorted *ele*. TRUE. i.e. 10 > 8 So move 10 from 0thindex to 1stindex	8
	10		15	46	32		
		10	15	46	32		
	8	**10**	**15**	46	32	Last index is 0th index. So place **x** value in 0th index	

	0	**1**	**2**	**3**	**4**		**x**
Pass 3	**8**	**10**	**15**	46	32	Sorted *ele*> unsorted *ele*. FALSE. i.e. 15>46 is FALSE	46

So place x value in next index of sorted element

8	**10**	**15**	**46**	32

Pass 4	**0**	**1**	**2**	**3**	**4**		**x**
	8	**10**	**15**	**46**	32	Sorted *ele*> unsorted *ele*. TRUE. i.e. 46 > 32 So move 46 from 3rdindex to 4thindex	32
	8	**10**	**15**		**46**	Sorted *ele*> unsorted *ele*. FALSE. i.e. 15 >32 is FALSE So place x value in next index of sorted element	
	8	**10**	**15**	**32**	**46**		

Now all the elements are sorted

3. Selection Sort

In selection sort first find the smallest element in the array and place it in to the array to the 0th position. Then find the second smallest element in the array and place it in 1st position. Repeat this procedure until array is sorteD)

Process:

- In Pass1, find the position of the smallest element in the array and then swap a[pos] and a[0]. Now a[0] is sorteD)
- In Pass2, find the position of the smallest element in sub array of n-1 elements, then swap a[pos] and a[1]. Now a[1] is sorteD)
- In Pass n-1, find the position of the smallest element from a[n-2] and a[n-1], then swap a[pos] and a[n-21]. So that a[0], a[1], a[2], a[3], ………, a[n-1] is sorteD)

Eg: sort the elements 15, 10, 11, 41, 3 using selection sort

	0	**1**	**2**	**3**	**4**		pos
Pass 1	15	10	11	41	3	pos is initially assigned at 0th index	0
	15	10	11	41	3	a[1] < a[pos] is TRUE. So pos =1	1
	15	10	11	41	3	a[2] < a[pos] is FALSE. So No change in pos	1
	15	10	11	41	3	a[3] < a[pos] is FALSE. So No change in pos	1
	15	10	11	41	3	a[4] < a[pos] is TRUE. So pos =4	4

Now swap a[pos] and a[0]

0	**1**	**2**	**3**	**4**
3	10	11	41	**15**

Now a[0] is sorted

	0	**1**	**2**	**3**	**4**		**pos**
Pass 2	**3**	10	11	41	15	Now pos is assigned at 1st index	1
	3	10	11	41	15	a[2] < a[pos] is FALSE. So No change in pos	1
	3	10	11	41	15	a[3] < a[pos] is FALSE. So No change in pos	1
	3	10	11	41	15	a[4] < a[pos] is FALSE. So No change in pos	1

Now swap a[pos] and a[1]

0	**1**	**2**	**3**	**4**
3	**10**	11	41	15

Now a[1] is sorted

	0	**1**	**2**	**3**	**4**		**pos**
Pass 3	**3**	**10**	11	41	15	Now pos is assigned at 2ndindex	2
	3	**10**	11	41	15	a[3] < a[pos] is FALSE. So No change in pos	2
	3	**10**	11	41	15	a[4] < a[pos] is FALSE. So No change in pos	2

Now swap a[pos] and a[2]

0	**1**	**2**	**3**	**4**
3	**10**	**11**	41	15

Now a[2] is sorted

	0	**1**	**2**	**3**	**4**		**pos**
Pass 4	**3**	**10**	**11**	41	15	Now pos is assigned at 3rdindex	3
	3	**10**	**11**	41	15	a[4] < a[pos] is TRUE. So pos = 4	4

Now swap a[pos] and a[3]

0	**1**	**2**	**3**	**4**
3	**10**	**11**	**15**	**41**

Now all the elements are sorted

Algorithm selectionsort (a<array>, n)

Input: *a* is an array with *n* elements to be sorteD)

Output: array elements in ascending order.

```
1. i = 0
2. while( i < n)
        a) pos = i
        b) j = i+1
        c) while ( j < n)
```

```
                    i) if ( a[j] < a[pos] )
                              A) pos = j
                    ii) end if
                    iii) j = j +1
          d) end loop
          e) t = a[pos]
          f) a[pos] = a[i]
          g) a[i] =t
          h) i= i+1
     3. end loop
```

End selectionsort

UNIT-I
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. Find the location of the element with a given value is___?[ ]

   A) Traversal B) Searching C) Sorting D) None of above

2. Which of the following is false? [ ]

   A) A linear search begins with the first array element

   B) A linear search continues searching, element by element, either until a match is found or until the end of the array is encountered

   C) A linear search is useful when the amount of data that must be search is small

   D) For a linear search to work, the data in the array must be arranged in either alphabetical or numerical order

3. Which characteristic will be used by binary search but the linear search ignores is _____. [ ]

   A) Order of the elements of the list B) Length of the list

   C) Maximum value in list D) Type of elements of the list

4. Choose the false statement. [ ]

   A) A binary search begins with the middle element in the array

   B) A binary search continues having the array either until a match is found or until there are no more elements to search

   C) If the search argument is greater than the value located in the middle of the binary, the binary search continues in the lower half of the array

   D) For a binary search to work, the data in the array must be arranged in either alphabetical or numerical order

5. Which of the following is *not* a limitation of binary search algorithm? [ ]

   A) Must use a sorted array

   B) Requirement of sorted array is expensive when a lot of insertion and deletions are needed

   C) There must be a mechanism to access middle element directly

   D) Binary search algorithm is not efficient when the data elements more than 1500

6. What is the complexity of searching an element from a set of n elements using Binary search algorithm is [ ]

   A) O(n) B) O(log n) C) O(n^2) D) O(n log n)

7. Label the process of arranging values in an ordered manner is called as ____________.

8. In which sorting technique, consecutive adjacent pairs of elements in the array are compared with each other. [ ]

   A) Bubble sort B) Selection Sort C) Insertion Sort D) None

9. Identify the number of comparisons required to sort a list of *10* numbers in *pass 2* by using *Bubble Sort* is_________. [ ]

   A) 10 B) 9 C) 8 D) 7

11. Consider an array of elements arr[5]= {99,22,55,44,33}, what are the steps done while doing bubble sort in the array. [ ]

   A) 22 55 44 33 99 33 22 44 99 55 22 44 99 33 55 44 22 55 33 99

   B) 22 55 44 33 99 22 44 33 55 99 22 33 44 55 99 22 33 44 55 99

   C) 55 44 33 99 22 44 22 33 99 55 55 33 99 22 44 99 55 44 33 22

   D) None of the above

12. Which sorting technique sorts a list of elements by moving the current data element past the already sorted values with the preceding value until it is in its correct place. [ ]

    A) Insertion sort B) Bubble Sort C) Selection Sort D) None

13. Identify the number of passes required by insertion sort for the list **size 15.** [ ]

    A) 15 B) 16 C) 14 D) 13

14. Which of the following sorting algorithms in its implementation gives best performance when applied on an array which is sorted or almost sorted (maximum 1 or two elements are misplaced). [ ]

    A) Insertion sort B) Bubble Sort C) Selection Sort D) None

15. Consider an array of elements arr[5]= {5,4,3,2,1} , what are the steps of insertions done while doing insertion sort in the array. [ ]

    A) 4 5 3 2 1 3 4 5 2 1 2 3 4 5 1 1 2 3 4 5

    B) 5 4 3 1 2 5 4 1 2 3 5 1 2 3 4 1 2 3 4 5

    C) 4 3 2 1 5 3 2 1 5 4 2 1 5 4 3 1 5 4 3 2

    D) 4 5 3 2 1 2 3 4 5 1 3 4 5 2 1 1 2 3 4 5

16. Consider the array A[]= {6,4,8,1,3} apply the *insertion sort* to sort the array . Consider the cost associated with each sort is 25 rupees, what is the total cost of the insertion sort when element 1 reaches the first position of the array? [ ]

    A) 50 B) 25 C) 75 D) 100

17. Consider a situation where swap operation is very costly. Which of the following sorting algorithms should be preferred so that the numbers of swap operations are minimized in general? [ ]

    A) Bubble Sort B) Selection Sort C) Insertion Sort D) None

18. Which one of the following in-place sorting algorithms needs the minimum number of swaps? [ ]

A) Insertion Sort B) Bubble Sort C) Selection Sort D) All of the above

19. Discover the comparisons needed to sort an array of length 5 if a straight selection sort is used and array is already in the opposite order? [ ]

    A) 1 B) 10 C) 5 D) 20

20. Determine the advantage of bubble sort over other sorting techniques? [ ]

    a) It is faster

    b) Consumes less memory

    c) Detects whether the input is already sorted

    d) All of the mentioned

SECTION-B

SUBJECTIVE QUESTIONS

1. Given a telephone directory and a name of the subscriber, choose search method you would suggest for finding the telephone number of the given subscriber.
2. Apply linear search for an element 18 and 100 in the following list.

   36, 72, 19, 45, 18, 22, 12, 55
3. Apply binary search for an element 54 and 100 in the following list.

   13, 27, 91, 54, 81, 6, 51, 59, 45, 69
4. Make use of bubble sort for the following elements.

   30, 52, 29, 87, 63, 27, 19, 54
5. Make use of insertion sort for the following elements.

   59, 19, 54, 96, 81, 801, 45, 72, 64, 92
6. Make use of selection sort for the following elements.

   36, 12, 81, 45, 90, 27, 72, 18
7. Explain bubble sort algorithm.

8. Explain insertion sort algorithm.
9. Explain selection sort algorithm.
10. Explain non recursive linear search algorithm.
11. Explain recursive binary search algorithm.
12. Develop a C program using for loop to find all the occurrences of a given key in a given list using linear search. The algorithm should display locations of all the occurrences of the given key. Discuss with an example.

SECTION-C

QUESTIONS AT THE LEVEL OF GATE

1. Consider the C function given below. Assume that the array listA contains n (> 0) elements, sorted in ascending order.**(GATE-CS-2014)**
[ ]

```
int ProcessArray(int *listA, int x, int n)
{
        int i, j, k;
       i = 0;
      j = n-1;
     do
     {
            k = (i+j)/2;
            if (x <= listA[k])
                   j = k-1;
             if (listA[k] <= x)
                i = k+1;
     }  while (i <= j);
```

```
            if (listA[k] == x)

                    return(k);
          else
                   return -1;
    }
```

Which one of the following statements about the function ProcessArray is CORRECT?
(A) It will run into an infinite loop when x is not in listA.
(B) It is an implementation of binary search.
(C) It will always find the maximum element in listA.
(D) It will return −1 even when x is present in listA.

2. Consider the following C program that attempts to locate an element x in an array Y[ ] using binary search. The program is erroneous. **(GATE CS 2008) [ ]**

```
1.   f(int Y[10], int x) {
2.     int i, j, k;
3.     i = 0; j = 9;
4.     do {
5.             k =  (i + j) /2;
6.             if( Y[k] < x)  i = k; else j = k;
7.         } while(Y[k] != x && i < j);
8.     if(Y[k] == x) printf ("x is in the array ") ;
9.     else printf (" x is not in the array ") ;
10. }
```

On which of the following contents of Y and x does the program fail?
(A) Y is [1 2 3 4 5 6 7 8 9 10] and x < 10
(B) Y is [1 3 5 7 9 11 13 15 17 19] and x < 1
(C) Y is [2 2 2 2 2 2 2 2 2 2] and x > 2
(D) Y is [2 4 6 8 10 12 14 16 18 20] and 2 < x < 20 and x is even

3. In the above question, the correction needed in the program to make it work properly is **(GATE CS 2008) [ ]**
(A) Change line 6 to: if (Y[k] < x) i = k + 1; else j = k-1;
(B) Change line 6 to: if (Y[k] < x) i = k – 1; else j = k+1;
(C) Change line 6 to: if (Y[k] <= x) i = k; else j = k;
(D) Change line 7 to: } while ((Y[k] == x) && (i < j));

4. The average number of key comparisons done in a successful sequential search in a list of length it is (**GATE CS 1996)[**]

(A) log n **(B)** (n-1)/2 **(C)** n/2 **(D)** (n+1)/2

UNIT –II

Objective:

- To gain knowledge on linked lists.

Syllabus:

Unit-II: Linked lists

Linked Lists- Basic concepts and operations of Single linked list, Circular linked list, Double linked list.

Learning Outcomes:

At the end of the unit student will be able to:

1. Define a self referential structure.
2. Describe about linked lists.
3. Implement the operations on linked lists.
4. Choose an appropriate linked list for a given problem.
5. Distinguish between single, double and circular linked lists.

Learning Material

➢ OPERATIONS ON DATA STRUCTURES:

The basic operations that are performed on data structures are as follows:

1) ***Traversing:***It means to access each data item exactly once so that it can be processed.

   For example, to print the names of all the students in a class.

2) ***Searching:***It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items.

   For example, to find the names of all the students who secured 100 marks in mathematics?

3) ***Inserting: It*** is used to add new data items to the given list of data items.

   For example, to add the details of a new student who has recently joined the course?

4) ***Deleting: It*** means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

5) ***Sorting:*** Dataitems can be arranged in some order like ascending order or descending order depending on the type of application.

For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

6) ***Merging:*** *Lists* of two sorted data items can be combined to form a single list of sorted data items.

➢ ***AbstractDataType:***

- Abstract data type is a Mathematical model or concept that defines a data type logically.
- **ADT** specifies a set of data and collection of operations that can be performed on that data.
- The definition of ADT only mentions ***What Operations are to be Performed but not how it is implemented***.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called "**abstract**" because it gives an implementation independent view (overview).
- The process of providing only the essentials and hiding the details is known as **Abstraction.**

➢ **LINKED LISTS:**

- In arrays once memory is allocated, it can't extended any more. So array is known as static Data Structure.
- Linked List is dynamic Data Structure, where amount of memory required can be vary during it use.

Definition: A Linked List is an ordered collection of finite, homogeneous data elements called nodes.

Where the linear order is maintained by means of links or pointers.

- The representation of node is as follows:

Data	Link

Node: an element in Linked List

- A node consists of two parts. i.e. **Data part** and **Link part**.
- The data part contains actual data to be represented.
- The link part is also referred as address field, which contains address of the next node.

- **Representation of Linked List in memory**

  There are two ways to represent a Linked List in memory.

  1. Static memory allocation using arrays.
  2. Dynamic allocations using pool of storage.

<u>Dynamic allocations using pool of storage</u>

- The efficient way of representation of linked list is using pool of storage.
- In this method memory bank, memory manager and garbage collector is available.

Memory bank: is a collection of free memory spaces.

Memory manager: is a program.

- Whenever a linked list requires a anode, then request is placed to memory manager.
- If the required node is available in the memory bank, then that node is send to caller.
- If the required node is not available in the memory bank, then memory manager send NULL to caller.

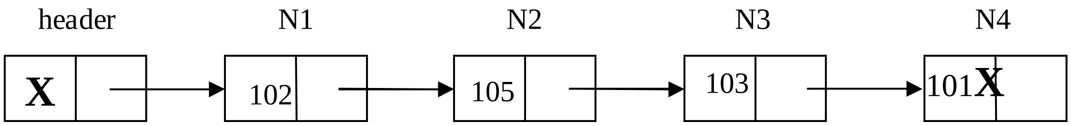
Garbage collector:collect the unused nodes in the linked list and send back to memory bank.

Types of linked lists

1. Single Linked List (SLL)
2. Circular Linked List (CLL)
3. Double Linked List (DLL)

1.Single Linked List (SLL):

- In SLL, each node contains only one link, which points to the next node in the list.
- The pictorial representation of SLL is as follows.



SLL with 4 nodes

- Here header is an empty node, i.e. data part is NULL, represented by X mark.
- The link part of the header node contains address of the first node in the list.
- In SLL, the last node link part contains NULL.
- In SLL we can move from left to right only. So SLL is called as one way list.

- **Operations on Single Linked List**

    1. Traversing a SLL
    2. Insertion of a node in to SLL
    3. Deletion of a node from SLL
    4. Search for a node in SLL
    5. Reversing a SLL

1. Traversing a SLL

Traversing a SLL means, visit every node in the list starting from first node to the last node.

```
AlgoritmSLL_Traverse(header)
Input: header is a header node.
Output: Visiting of every node in SLL.
    1. ptr=header
    2. while(ptr.link != NULL)
        a) ptr=ptr.link go to step(b)
        b) print "ptr.data"
    3. end loop
End SLL_Traverse
```

2. Insertion of a node into SLL

- The Insertion of a node in to SLL can be done in various positions.
    i) Insertion of a node into SLL at beginning.
    ii) Insertion of a node into SLL at ending.
    iii) Insertion of a node into SLL at any position.
- For insertion of a node into SLL, we must get node from memory bank.
- The procedure for getting node from memory bank is as follows:

Procedure for getnewnode( )

```
    1. Check for availability of node in memory bank
    2. if ( AVAIL = NULL)
        a) print "Required node is not available in memory"
        b) return NULL
    3. else
        a) return address of node to the caller
    4. end if
End Procedure for getnewnode
```

i) Insertion of a node into SLL at beginning

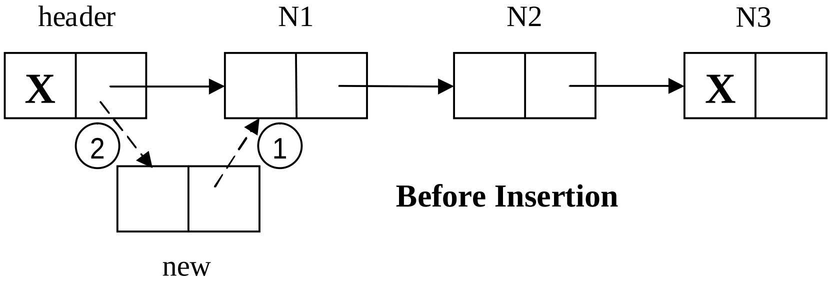
AlgoritmSLL_Insert_Begin(header,x)

Input: header is a pointer to the header node, x is data part of new node to be inserted.

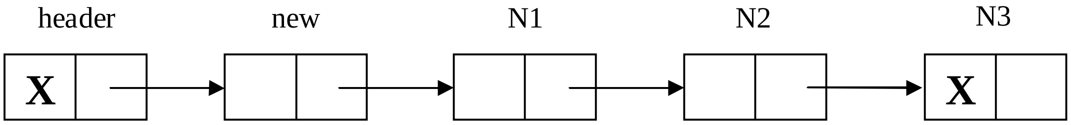
Output: SLL with new node inserted at beginning.

```
    1. new=getnewnode( )
    2. if(new = = NULL)
        a) print "required node was not available in memory, so unable to process"
    3. else
        a) new.link=header.link        /* 1 */
        b) header.link=new             /* 2 */
        c) new.data=x
    4. end if
End SLL_Insert_Begin
```

1. Link part of new node is replaced with address of first node in list, i.e. link part of header node.
2. Link part of header node is replaced with new node address



Before Insertion



After Insertion

.

ii) Insertion of a node into SLL at ending

- To insert a node into SLL at beginning first we need to traverse to last node, then insert as new node as last node.

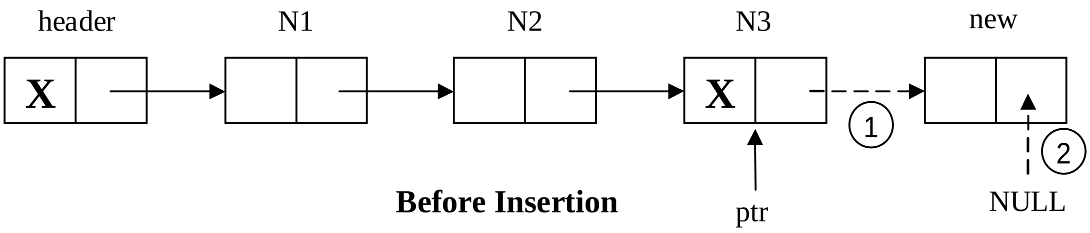
Algorithm SLL_Insert_Ending(header,x)

Input: header is header node, x is data part of new node to be insert.

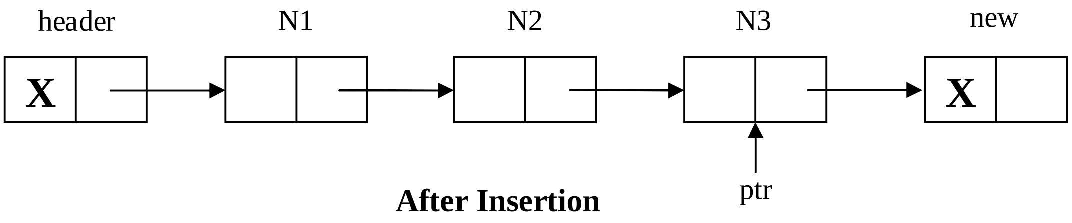
Output: SLL with new node at ending.

```
1. new=getnewnode()
2. if(new = NULL)
        a) print "Required node was not available in memory bank, so unable to process"
3. else
        a) ptr=header
        b) while(ptr.link!=NULL)
                i) ptr=ptr.link go to step(b)
        c) end loop
        d) ptr.link=new                /* 1 */
        e) new.link=NULL               /* 2 */
        f) new.data=x
4. end if
```

End_SLL_Insert_Ending



Before Insertion



After Insertion

1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with NULL, because new node becomes the last node.

iii) Insertion of a node into SLL at any position.

- For insertion of a node at any position in SLL, a key value is specified. Where key being the data part of a node, after this node new node has to be inserting.

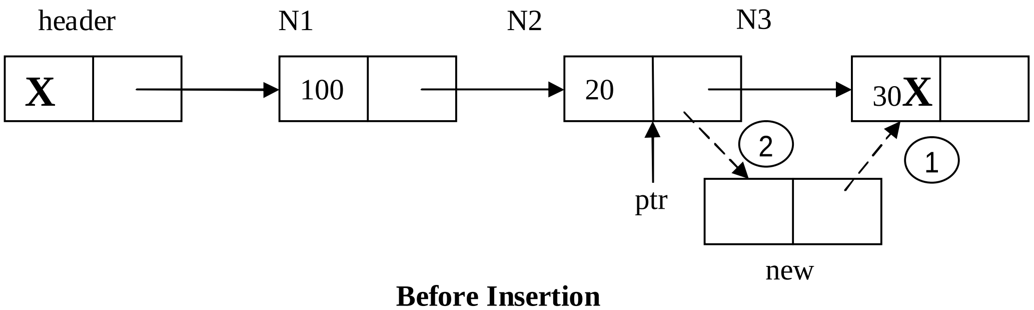
Algorithm SLL_Insert_ANY(header,x,key)

Input: header is header node, x is data pat of new node to be inserting, key is the data part of a node, after this node we want to insert new node.

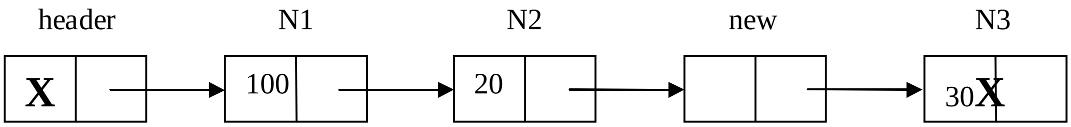
Output: SLL with new node at ANY

```
1. new=Getnewnode( )
2 .if(new = = NULL)
        a. print “required node was not available in memory bank, so unable to process”
3. else
        i. ptr=header
        ii. while(ptr.data!=key and ptr.link!=NULL)
                a) ptr=ptr.link
        iii. end loop
        iv. if(ptr.link=NULL and ptr.data!=key)
        a) print “required node with data part as key value is not available, so unable to process”
        v. else
                a) new.link=ptr.link            /* 1 */
                b) ptr.link=new                 /* 2 */
                c) new.data=x
        vi. end if
4. end if.
```

End SLL_insert_ANY



Before Insertion



After Insertion

1. Link part of new node is replaced by the address of next node. i.e. in the above example N3 becomes next node for newly inserting node.
2. Link part of previous node is replaced by the address of new node. i.e. in the above example N2 becomes previous node for newly inserting node.

3. Deletion of a node from SLL

The deletion of a node in from SLL can be done in various positions.

i) Deletion of a node from SLL at beginning.

ii) Deletion of a node from SLL at ending.

iii) Deletion of a node from SLL at any position.

i) Deletion of a node from SLL at beginning

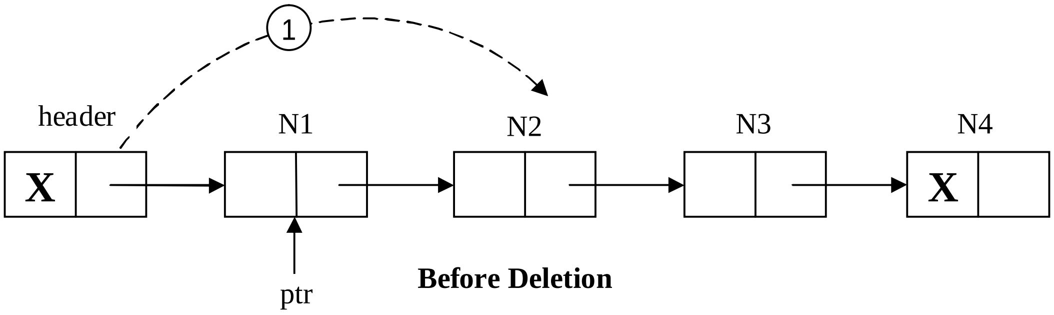
Algorithm SLL_Delete_Begin(header)

Input: Header is a header node.

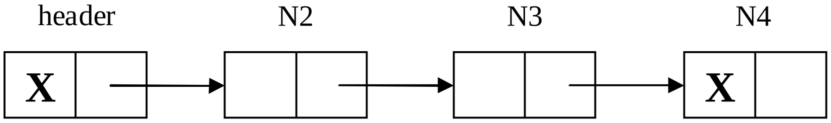
Output: SLL with node deleted at Beginning.

```
1. if(header.link = = NULL)
        a) print "SLL is empty, so unable to delete node from list"
2. else                      /*SLL is not empty*/
        i. ptr=header.link              /* ptr points to first node into list*/
        ii. header.link=ptr.link        /* 1 */
        iii. return(ptr)           /*send back deleted node to memory bank*/
3. end if
```

End SLL_Delete_Begin



Before Deletion



After Deletion

1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

ii) Deletion of a node from SLL at ending

- To delete a node from SLL at ending, first we need to traverse to last node in the list.
- After reach the last node in the list, last but one node link part is replaced with NULL.

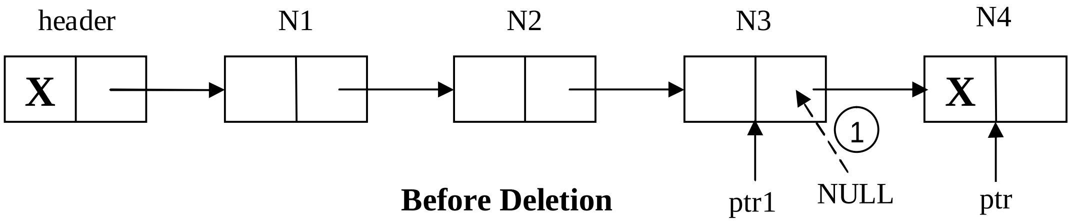
Algorithm SLL_ Delete_End (header)

Input: header is a header node

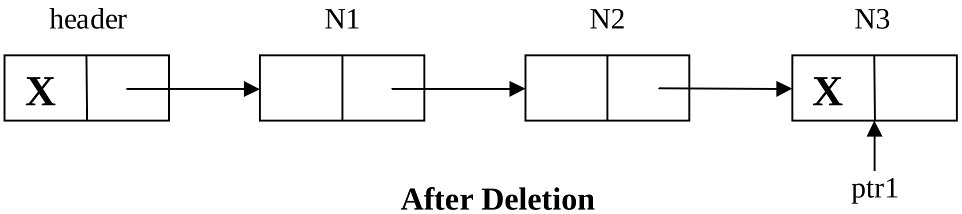
Output: SLL with node deleted at ending.

```
1. if(header.link = = NULL)
        a)print "SLL is empty, so unable to delete the node from list"
2. else                                 /*SLL is not empty*/
        a) ptr=header                    /*ptr  initially points to header node*/
        b) while(ptr.link!=NULL)
                i) ptr1=ptr
                ii) ptr=ptr.link         /*go to step b*/
        c) end loop
        d) ptr1.link=NULL        /* 1 */
        e) return(ptr)
3. end if
```

End SLL_ Delete_ End



Before Deletion



After Deletion

1. Link part of last but one node is replaced with NULL. Because after deletion of last node in the list, last but one node become the last node.

iii) Deletion of a node from SLL at any position

- For deletion of a node from SLL at any position, a key value is specified.
- Where key being the data part of a node to be deleting.

Algorithm SLL_ Delete_ ANY (header,key)

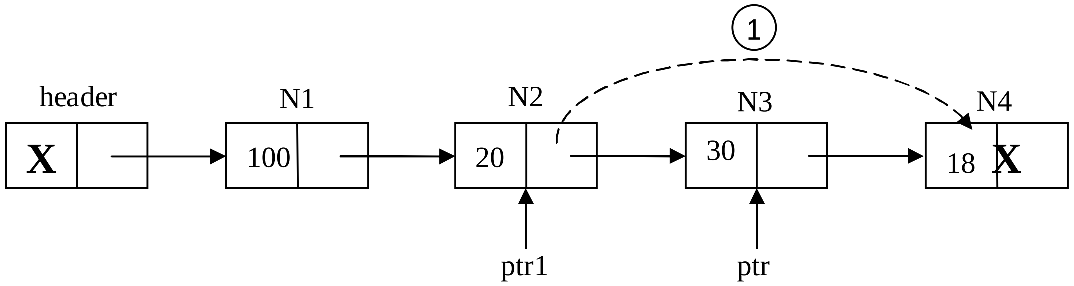
Input: header is a header node, key is the data part of the node to be delete.

Output: SLL with node deleted at Any position. i.e. Required element.

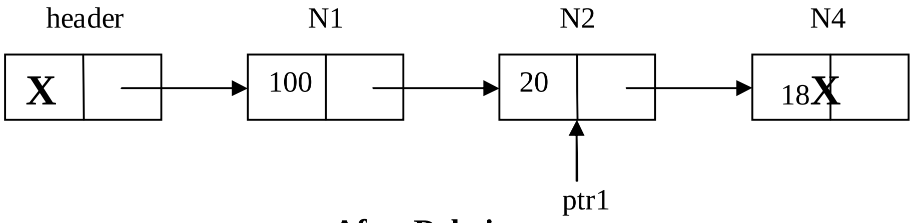
```
1. if(header.link = = NULL)
        a)print “SLL is empty, so unable to delete the node from list”
2. else                                    /*SLL is not empty*/
        a) ptr=header                              /*ptr  initially points to header node*/
        b) while(ptr.link!=NULL and ptr.data!=key)
                i) ptr1 = ptr
                ii) ptr=ptr.link   go to step b
        c) end loop
        d) if(ptr.link = = NULL and ptr.data!=key)
                i) print “Required node with data part as key value is not available”
        e) else                            /* node with data part as key value available */
                i) ptr1.link = ptr.link    /* 1 */
                ii) return(ptr)
        f) end if
3. end if
```

End SLL_ Delete_ ANY

1. Previous node link part is replaced with address of next node in the list. i.e. in the above example N2 becomes the previous node and N4 becomes the next node for the node to be delete.



Before Deletion



After Deletion

4. Search for a node in SLL:

- For searching a node in SLL, a key value is specified by the user.
- If any node's data part is equal to key value, then search is successful.
- Otherwise the required node is not available or unsuccessful search.

Algorithm SLL_Search(header,key)

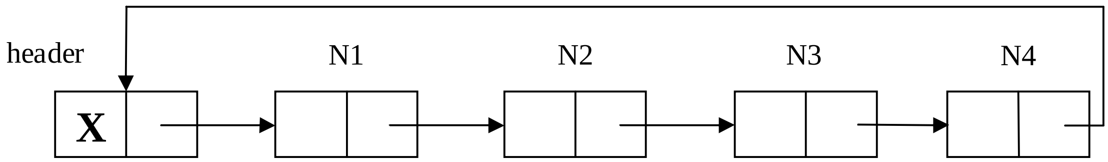
Input: header is a header node
Output: Location is pointer to a node with data part as key value.

```
1. ptr = header
2. flag = 0
3. Location = NULL
4. while(ptr.link != NULL)
    a) ptr = ptr.link
    b)if(ptr.data==key)
      i) flag=1
      ii) Location=ptr
      iii) break
c)end if
5. end loop
6. if (ptr.data = = key)
    a) flag = 1
```

```
            b) Location = ptr
            c) return (Location)
    7. else
            a) print "required node was not available"
    8. end if
End SLL_Search
```

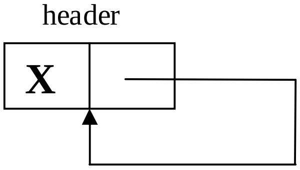
2. Circular Linked List :

- Circular linked list is a special type of linked list.
- In a Circular Linked list, the field of the last node points to the first node of the list.
- It is mainly used in lists that allow to access to nodes in the middle of the list without starting at the beginning.



Circular Linked List

- If a CLL is empty, then the link part of the header node points to itself.



Empty Circular Linked List

➢ **Operations on Circular Linked List:**

1. Insertion of a node in to CLL
2. Deletion of a node from CLL

1. Insertion of a node in to CLL

The Insertion of a node in to CLL can be done in various positions.

i) Insertion of a node into CLL at beginning.

ii) Insertion of a node into CLL at ending.

iii) Insertion of a node into CLL at any position.

i) Insertion of a node into CLL at beginning.

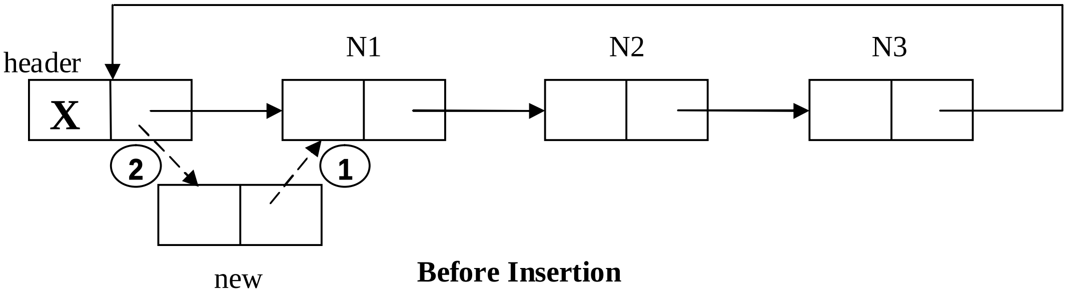
AlgorithmCLL_Insert_Begin(header,x)

Input: header is a header node, x is data part of new node to be insert.

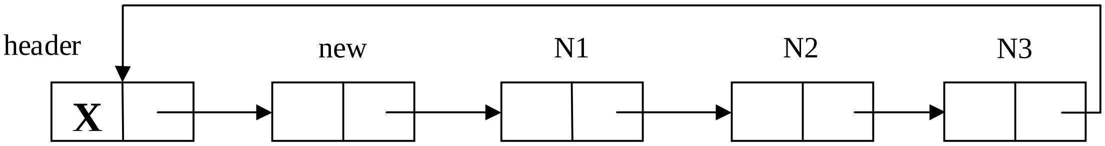
Output: CLL with new node inserted at beginning.

```
1. new=getnewnode( )
2. if(new = = NULL)
      a) print "required node was not available in memory, so unable to process"
3. else
      a) new.link = header.link                    /* 1 */
      b) header.link = new                         /* 2 */
c )new.data = x
4. end if
```

End CLL_Inset_Begin



Before Insertion



After Insertion

1. Link part of new node is replaced with address of first node in list, i.e. link part of header node.
2. Link part of header node is replaced with new node address.

ii) Insertion of a node into CLL at ending.

AlgorithmCLL_Insert_END(header,x)

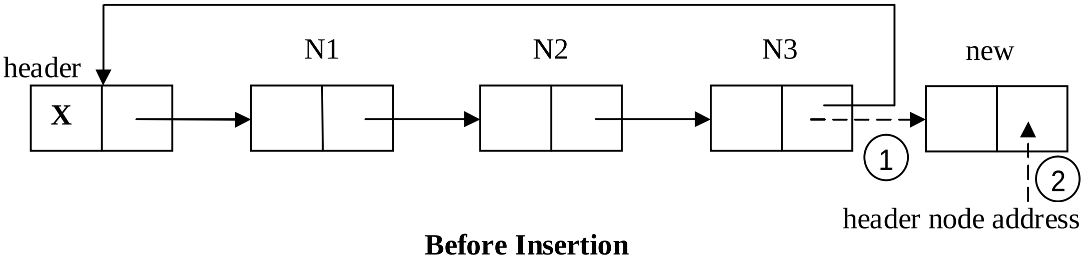
Input: header is a header node, x is data part of new node to be insert.

Output: CLL with new node inserted at ending.

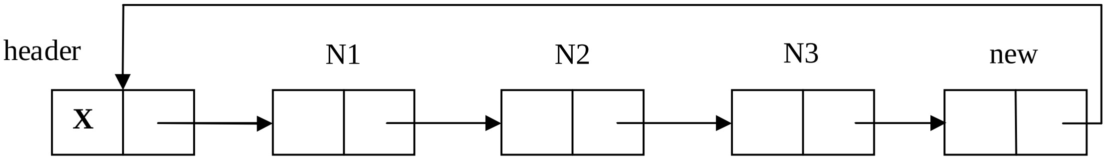
```
1. new=getnewnode()
```

```
2 .if(new = = NULL)
        a) print “required node was not available at memory bank, so unable to process”
3. else
        a) ptr=header
        b) while(ptr.link != header)
                i) ptr=ptr.link      //goto step b
        c) end loop
        d) ptr.link=new
        e) new.link=header
        f) new.data=x
4. end if
```

End CLL_Insertion_END



Before Insertion



After Insertion

1. Previous last node link part is replaced with address of new node.
2. **2.** Link part of new node is replaced with address of header node, because new node becomes the last node.

iii) Insertion of a node into CLL at any position.

Algorithm CLL_Insert_ANY(header,x,key)

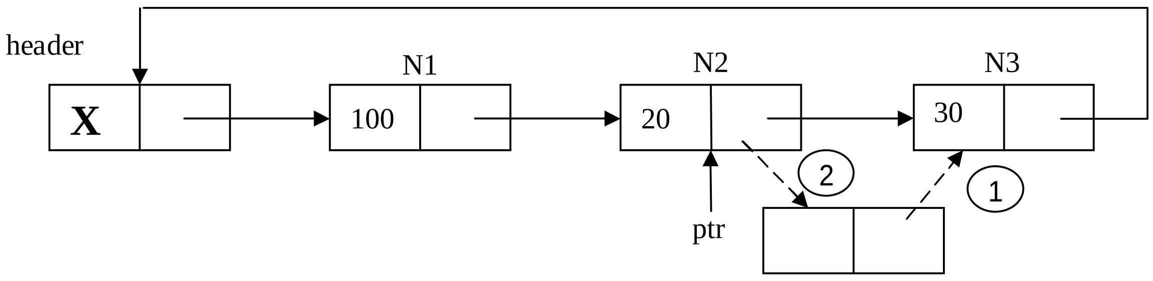
Input: header is header node, x is data pat of new node to be insert, key is the data part of a node, after this node we want to insert new node.

Output: CLL with new node at ANY

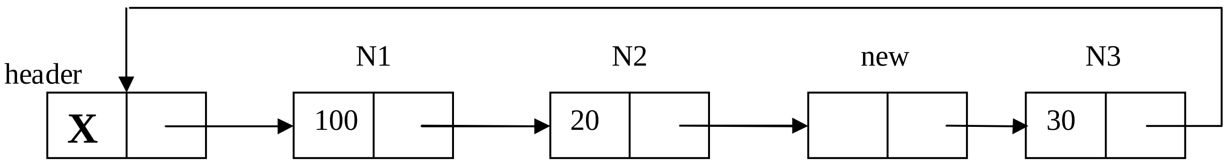
```
1. new=Getnewnode()
2 .if(new=NULL)
```

```
        a. print "required node was not available in memory bank, so unable to process"
3. else
        i. ptr=header
        ii. while(ptr.data! =key and ptr.link!=header)
                a) ptr=ptr.link
        iii. end loop
        iv. if(ptr.link=header and ptr.data!=key)
                a) print "required node with data part as key value is not available, so unable to
process"
        v. else
                a) new.link=ptr.link
                b) ptr.link=new
                c) new.data=x
        vi. end if
4. end if.
```

End CLL_insert_ANY



Before Insertion



After Insertion

1. Link part of new node is replaced by the address of next node. i.e. in the above example N3 becomes next node for newly inserting node.
2. Link part of previous node is replaced by the address of new node. i.e. in the above example N2 becomes previous node for newly inserting node.

2. Deletion of a node from CLL:

The Deletion of a node from CLL can be done in various positions.

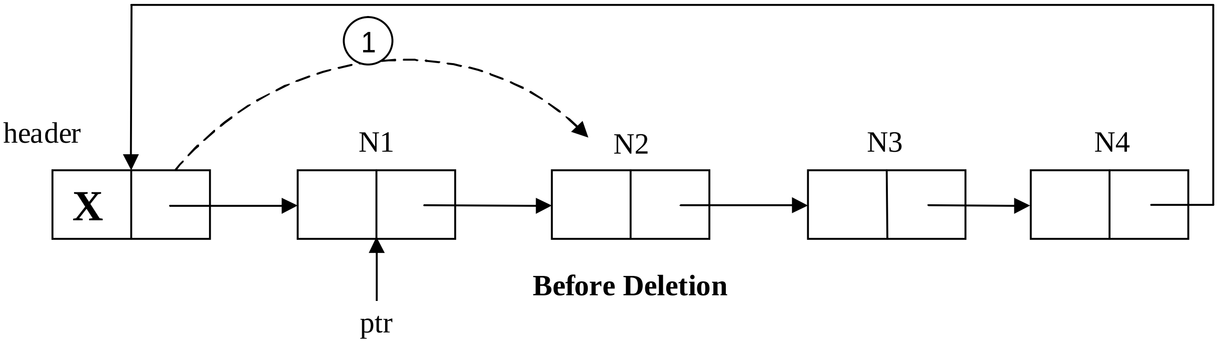
i) Deletion of a node from CLL at beginning.

ii) Deletion of a node from CLL at ending.

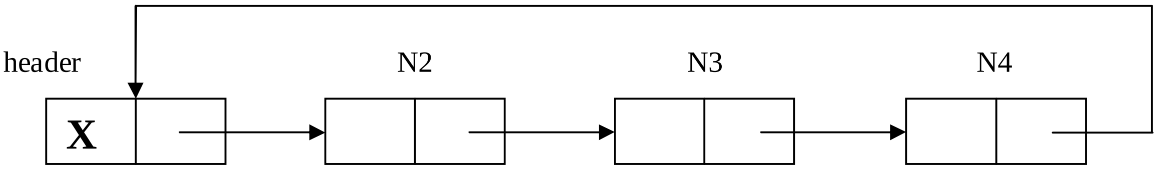
iii) Deletion of a node from CLL at any position

i) Deletion of a node from CLL at beginning

```
Algorithm CLL_Delete_Begin(header)
Input: Header is a header node.
Output: CLL with node deleted at Beginning.
    1. if(header.link = = header)
        a) print "CLL is empty, so unable to delete node from list"
    2. else                     /*DLL is not empty*/
        i. ptr=header.link                  /* ptr points to first node into list*/
        ii. header.link=ptr.link   /* 1 */
        iii. return(ptr)           /*send back deleted node to memory bank*/
    3. end if
End CLL_Delete_Begin
```



Before Deletion



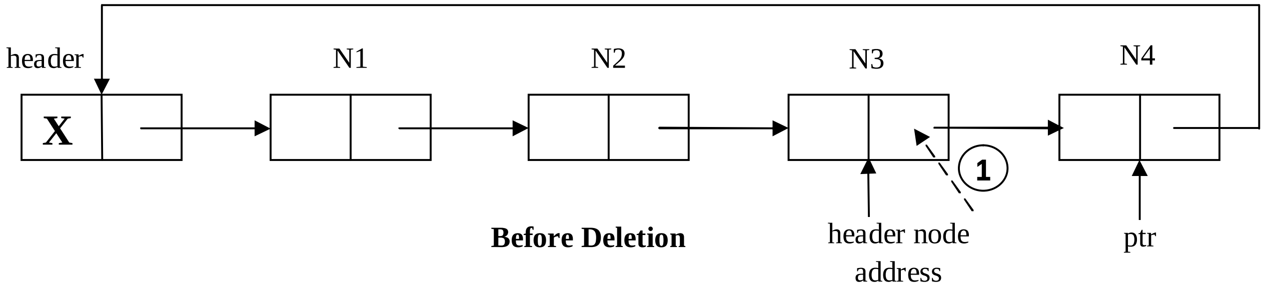
After Deletion

1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

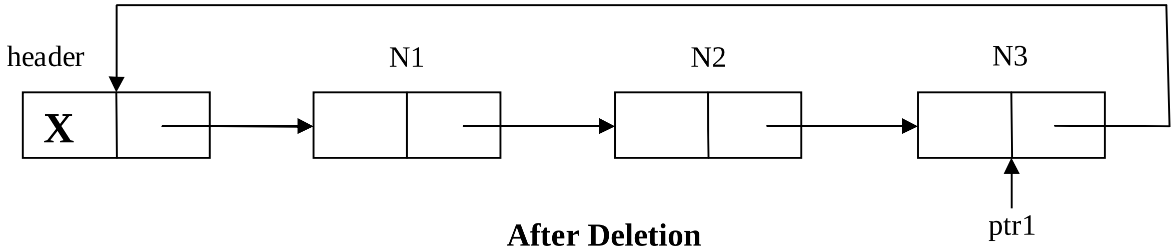
ii) Deletion of a node from CLL at ending

```
Algorithm   CLL_ Delete_ End (header)
Input:  header is a header node
Output: CLL with node deleted at ending.
    1. if(header.link = = header)
        a)print “CLL is empty, so unable to delete the node from list”
    2. else                                   /*CLL is not empty*/
        a) ptr=header  /*ptr  initially points to header node*/
        b) while(ptr.link!=header)
            i) ptr1=ptr
            ii) ptr=ptr.link   /*go to step b*/
        c) end loop
        d) ptr1.link=header                   /* 1 */
        e) return(ptr)
    3. end if
End   CLL_ Delete_ End
```

1. Link part of last but one node is replaced with address of header node. Because after deletion of last node in the list, last but one node become the last node.



Before Deletion



After Deletion

iii) Deletion of a node from CLL at any position

- For deletion of a node from CLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

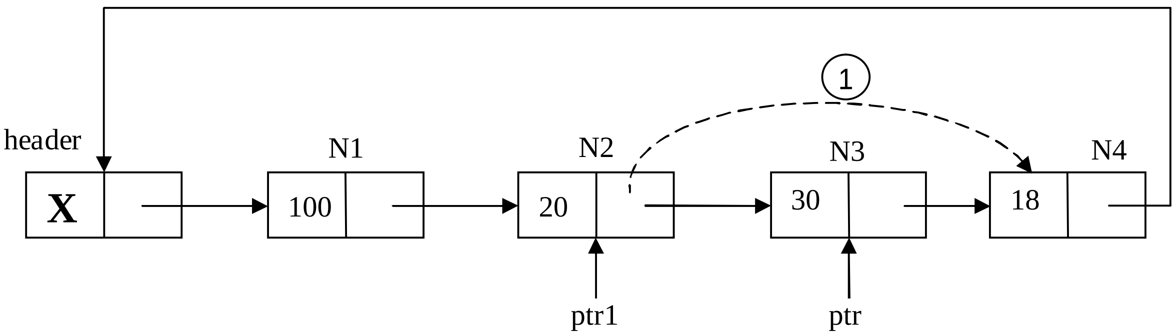
Algorithm CLL_ Delete_ ANY (header,key)

Input: header is a header node, key is the data part of the node to be delete.

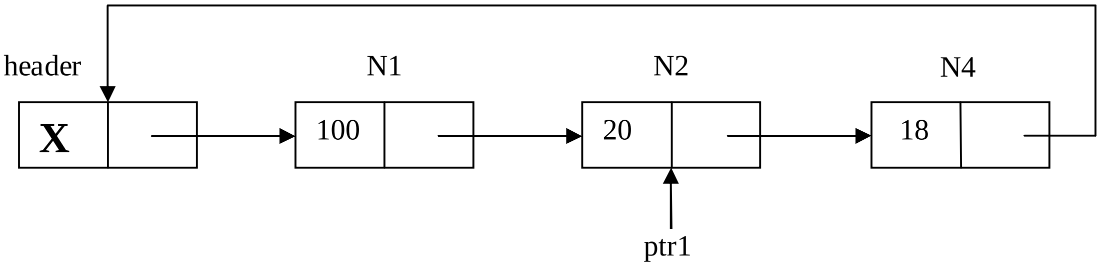
Output: CLL with node deleted at Any position. i.e. Required element.

```
1. if(header.link = = header)
        a)print “SLL is empty, so unable to delete the node from list”
2. else                                 /*CLL is not empty*/
        a) ptr=header  /*ptr  initially points to header node*/
        b) while(ptr.link!=header and ptr.data!=key)
                i) ptr1 = ptr
                ii) ptr=ptr.link   go to step b
        c) end loop
        d) if(ptr.link = = header and ptr.data!=key)
                i) print “Required node with data part as key value is not available”
        e) else                         /*node with data part as key value available
                i) ptr1.link = ptr.link          /* 1 */
                ii) return(ptr)
        f) end if
3. end if
```

End CLL_ Delete_ ANY



Before Deletion

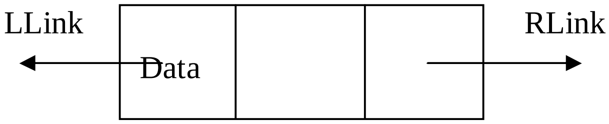


After Deletion

1. Previous node link part is replaced with address of next node in the list. i.e. in the above example N2 becomes the previous node and N4 becomes the next node for the node to be delete.

3. Double Linked List:

- In a SLL one can move from the header node t o any node in one direction only. i.e. from left to right.
- A DLL is a two way list. Because one can move either from left to right or right to left.
- In DLL, each node maintains two links.



Structure of a node in DLL

- Here**LLink** refers Left Link and **RLink** refers Right Link.
- The LLink part of a node in DLL always points to the previous node. i.e. LLink part of a node Consists address of previous node.

- The RLink part of a node in DLL always points to the next node. i.e. RLink part of a node Consists address of next node.

➢ Operations on Double Linked List:

1. Insertion of a node in to DLL

2. Deletion of a node from DLL

1. Insertion of a node in to DLL:

The Insertion of a node in to DLL can be done in various positions.

i) Insertion of a node into DLL at beginning.

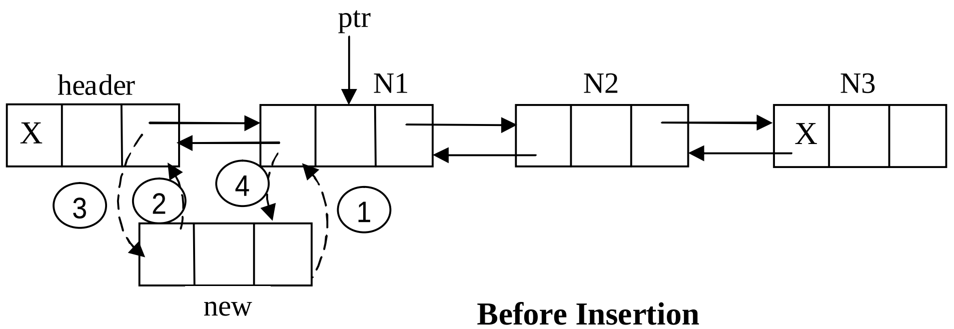
ii) Insertion of a node into DLL at ending.

iii) Insertion of a node into DLL at any position.

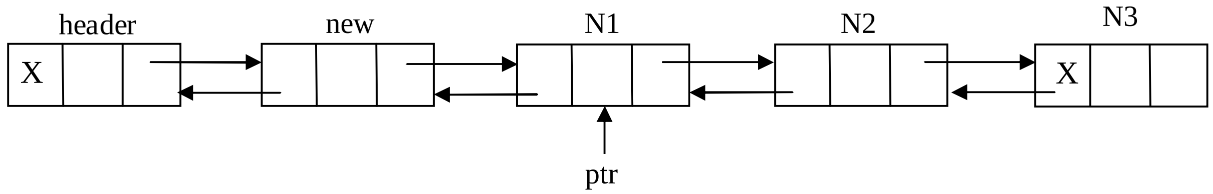
- For insertion of a node into DLL, we must get node from memory bank. The procedure for getting node from memory bank is same as getting node for SLL from memory bank.

i) Insertion of a node into DLL at beginning

```
Algorithm DLL_insertion_Begin(header,X)
Input: header is a header node.
Output: DLL with new node at begin.
        1. new=getnewnode()
        2. if(new = = NULL)
                a) print “required node is not available in memory”
        3. else
                a) ptr=header.rlink
                b) new.rlink=ptr                /* 1 */
                c) new.llink=header             /* 2 */
                d) header.rlink=new             /* 3 */
                e) ptr.llink=new                /* 4 */
        4. end if
End DLL_insertion_Begin
```



Before Insertion



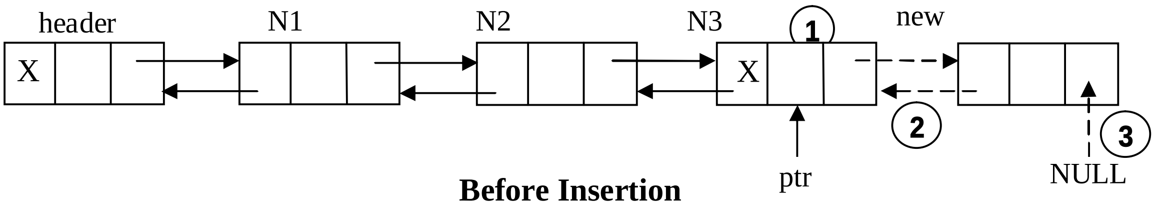
After Insertion

1. Rlink part of new node is replaced with the address of first node in the DLL. i.e. address of first node is available in Rlink part of header node.
2. Llink part of new node is replaced with the address of header.
3. Rlink part of header node is replaced with the address of new node.
4. Llink part of previous first node is replaced with the address of new node.

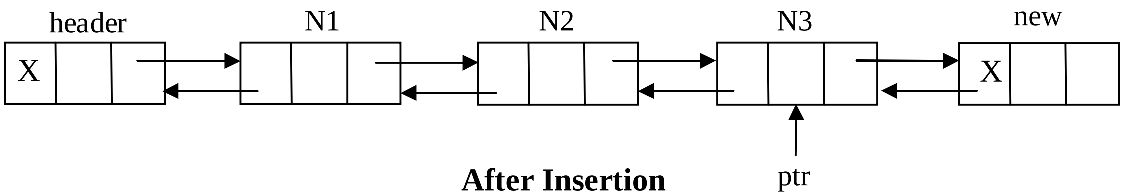
ii) Insertion of a node into DLL at ending.

```
Algorithm DLL_Insert_Ending(Header,x)
Input: Header is the header node, x is the data part of new node to be inserted.
Output: DLL with new node inserted at the ending.
1. new=getnewnode()
2. if(new = = NULL)
        a)print "Required node was not available"
3. else
        a) ptr=header
        b) while(ptr.rlink != NULL)
                i) ptr=ptr.rlink            goto step(b)
        c) end while loop
        d) ptr.rlink=new                /* 1 */
        e) new.llink=ptr                /* 2 */
        f) new.rlink=NULL               /* 3 */
        g) new.data=x
    4. end if
```

End DLL_Insertion_Ending



Before Insertion



After Insertion

1. RLink part of last node in the DLL is replaced with address of new node.
2. LLink part of new node is replaced with address of previous last node.
3. RLink part of new node is replaced with NULL. Because newly inserted node becomes the last node in the list.

iii) Insertion of a node into DLL at any position

- For insertion of a node at any position in DLL, a key value is specified.
- Where key being the data part of a node, after this node new node has to be inserting.

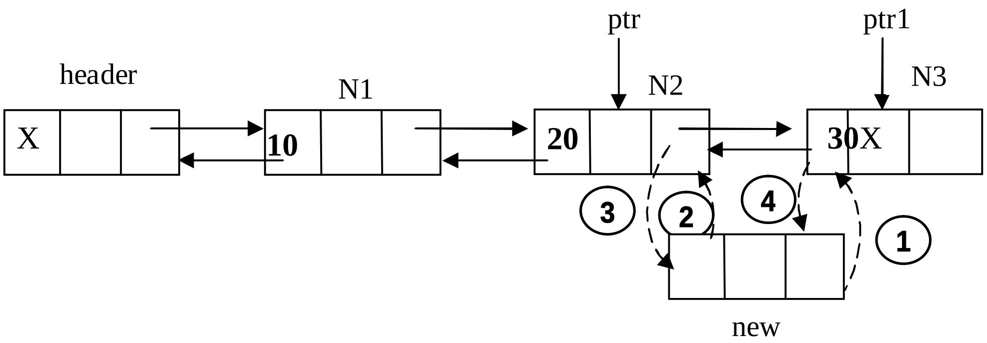
Algorithm DLL_Insertion_ANY(header,x,key)

Input: Header is a header node, key is the data part of a node, after that node new node is inserted, x is data part of new node to be insert.

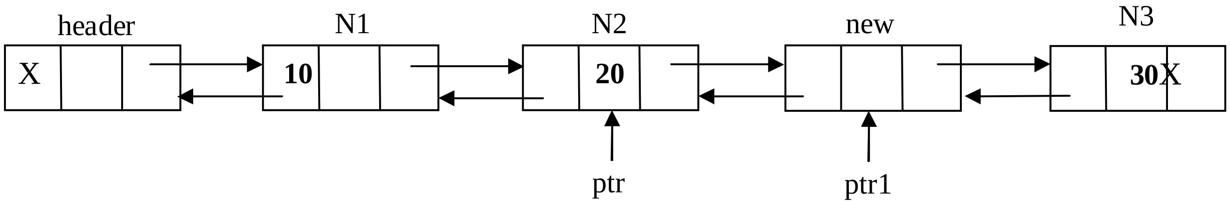
Output: DLL with new node inserted after the node with data part as key value

```
    1. new=getnewnode()
    2. if(new = = NULL)
        a) print"required node is not available in memory"
    3. else
        a) ptr=header
        b) while(ptr.data!=key and ptr.rlink!=NULL)
            i) ptr=ptr.rlink     go to step(b)
        c) end loop
        d) if(ptr.rlink = = NULL and ptr.data != key)
            i) print "required node with key value was not available"
        e) else
            i) ptr1=ptr.rlink
            ii) new.rlink=ptr1                /* 1 */
            iii) new.llink=ptr                /* 2 */
            iv) ptr.rlink=new                 /* 3 */
            v) ptr1.llink=new                 /* 4 */
            vi)new.data=x
        f) end if
    4. end if
```

EndDLL_Insertion_ANY



Before Insertion



After Insertion

1. RLink part of new node is replaced with the address of next node. i.e. in the above example N3 becomes the next node for newly inserting node.
2. LLink part of new node is replaced with the address of previous node. i.e. in the above example N2 becomes the previous node for newly inserting node.
3. RLink part of previous node is replaced with address of new node.
4. LLink part of next node is replaced with address of new node

2. Deletion of a node from DLL

The deletion of a node in from DLL can be done in various positions.

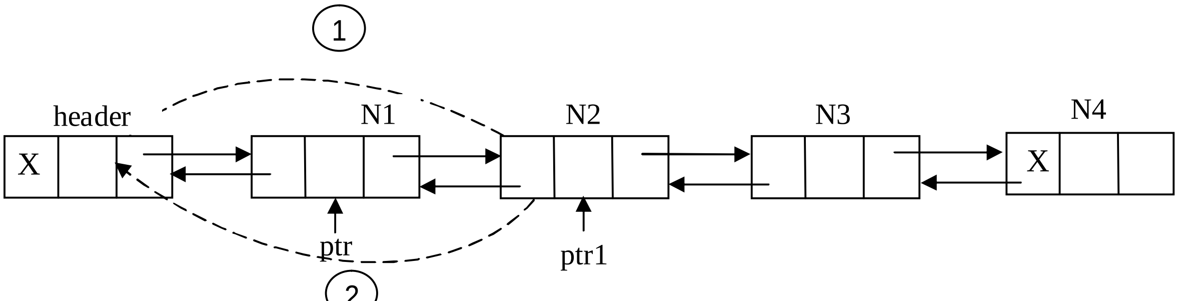
i) Deletion of a node from DLL at beginning.

ii) Deletion of a node from DLL at ending.

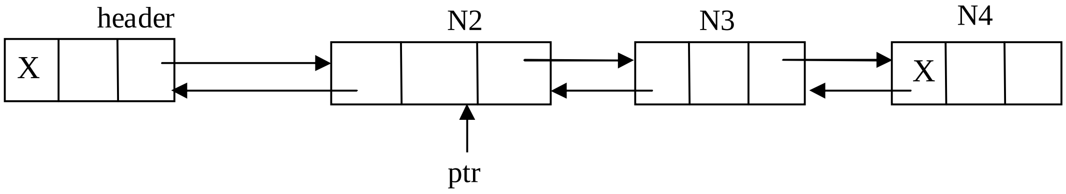
iii) Deletion of a node from DLL at any position.

i) Deletion of a node from DLL at beginning

```
Algorithm DLL_Deletion_Begin(header)
Input: header is a header node
Output:  DLL with node deleted at begin
    1. if(header.rlink = = NULL)
        a) Print "DLL is empty, not possible to perform deletion operation"
    2. else
        a) ptr=header.rlink
        b) ptr1=ptr.rlink
        c) header.rlink=ptr1          /* 1 */
        d) ptr1.llink=header          /* 2 */
        e) return(ptr)
    3. End if
        End DLL_Deletion_Begin
```



Before Deletion



After Deletion

1. RLink part of header node is replaced with the address of second node. i.e. address of second node is available RLink part of first node.
2. LLink part of second node is replaced with the address of header node.

ii) Deletion of a node from DLL at ending.

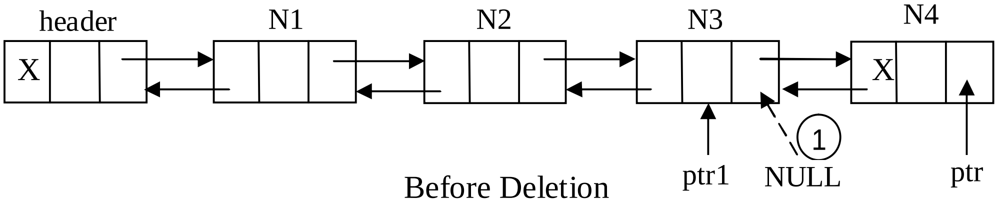
Algorithm DLL_Deletion_End(header)

Input: header is a header node.

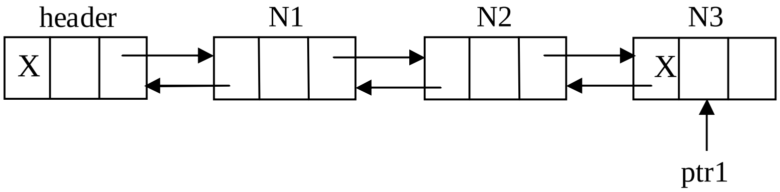
Output: DLL with deleted node at ending.

```
1. if(header.rlink=NULL)
        a) Print “DLL is empty, not possible to perform deletion operation”
2. else
        a) ptr=header
        b) while(ptr.rlink != NULL)
                i) ptr1=ptr
                ii) ptr=ptr.rlink
        c) end loop
        d) ptr1.rlink=NULL                    /* 1 */
        e) return(ptr)
3. end if
```

End DLL_Deletioon_Ending



Before Deletion



AfterDeletion

1. RLink part of last but one node in DLL is replaced with NULL. Because last but one node becomes last node.

iii) Deletion of a node from DLL at any position.

- For deletion of a node from DLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

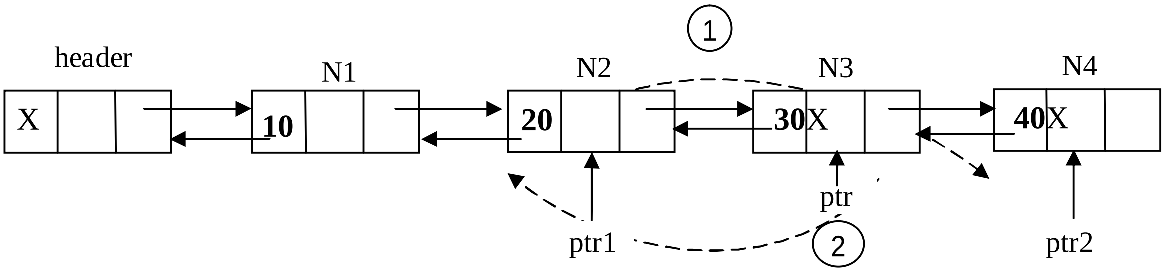
Algorithm DLL_Deletion_Any(header,key)

Input: header is header node, key is the data part of a node to be delete.

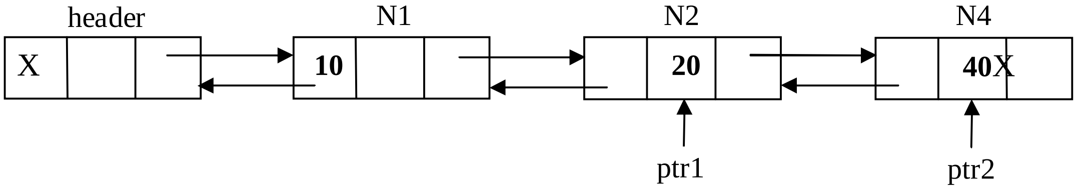
Output: DLL without node as data part is key value.

```
    1. if(header.rlink = = NULL)
            a) print "DLL is empty, not possible for deletion operation"
    2. else
            i) ptr=header
            ii) while(ptr.data!=key and ptr.rlink!=NULL)
                    a) ptr=ptr.rlink
            iii) end loop
            iv) if(ptr.rlink=NULL and ptr.data != key)
                    a) print "required node was not available in list"
            v) else
                    a) ptr1 = ptr.llink
                    b) ptr2 = ptr.rlink
                    c) ptr1.rlink = ptr2            /* 1 */
                    d) ptr2.rlink = ptr1            /* 2 */
            vi) end if
    3. end if
```

End DLL_Deletion_Any



Before Deletion



After Deletion

1. RLink part of previous node is replaced with the address of next node. i.e. in the above example N2 become the previous node to node to be delete, N4 becomes the next node to node to be delete.
2. LLink part of next node is replaced with the address of previous node. i.e. in the above example N2 become the previous node to node to be delete, N4 becomes the next node to node to be delete.

UNIT-II
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. The logical or mathematical model of a particular organization of data is defined as ______________.

2. An ordered collection of finite, homogeneous data elements where the linear order is maintained by means of links or pointers is defined as ___________________.

3. In single linked list each node contain minimum of two fields. One field is data field to store the data and select for what purpose the second field is used to store _____________? [ ]

   a) Pointer to character b) Pointer to integer c) Pointer to next node d) None

4. Identify the memory allocation process in Linked list [ ]

   a)Dynamic b)Compile Time c)Static d)None of these

5. A variant of linked list, identify in which last node of the list points to the first node of the list is? [ ]

   a)Singly linked list b) Doubly linked list c)Circular linked list d) Multiply linked list

6. In doubly linked lists, identify which type of traversal can be performed? [ ]

   a)Only in forward direction b) Only in reverse direction c)In both directions d) None

7. A variant of the linked list, identify in which none of the node contains NULL pointer is? [ ]

   a)Singly linked list b) Doubly linked list c)Circular linked list d) None

8. Identify non-linear Data Structure from the following [ ]

   a. Array b. Stack c. Graph d. Linked list

9.A node in single linked list can reference the previous node. [True/False]

10. Choose, Which type of structure is used to create a linked list? [ ]

   a) Nested structure b) Self referential structure c) Array of structure d)pointers to structure

11. Predict, Which type of linked list occupies more memory? [ ]

   a)SLL b) DLL c)CLL d)None

12. Compute how many pointers need to modify in inserting a node at the beginning of the single linked list

   a) 1 b) 2 c) 3 d) 0 [ ]

13. What does the following function do for a given Linked List with first node as head? [ ]

```
void fun1(struct node* head)
{
  if(head == NULL)
    return;


  fun1(head->next);
  printf("%d  ", head->data);
}
```

a) Prints all nodes of linked lists

b) Prints all nodes of linked list in reverse order

c) Prints alternate nodes of Linked List

d) Prints alternate nodes in reverse order

14. Deleting a node at any position (middle) of the single linked list needs to modify __________pointers.

a) 1 b) 2 c) 3 d) 0 [ ]

15. A double linked list is declared as follows: [ ]

```
struct dllist
{
struct dllist *fwd, *bwd;
int data;
}
```

Where fwd and bwd represents forward and backward links to adjacent elements of the list. Which among the following segments of code deletes the element pointed to by X from the double linked list, if it is assumed that X points to neither the first nor last element of the list?

```
a. X -> bwd -> fwd = X -> fwd;
   X -> fwd -> bwd = X -> bwd
b. X -> bwd -> fwd = X -> bwd;
   X -> fwd -> bwd = X -> fwd
c. X -> bwd -> bwd = X -> fwd;
   X -> fwd -> fwd = X -> bwd
d. X -> bwd -> bwd = X -> bwd;
   X -> fwd -> fwd = X -> fwd
```

15. Which among the following segment of code inserts a new node pointed by X to be inserted at the beginning of the double linked list. The start pointer points to beginning of the list? [ ]

```
a. X -> bwd = X -> fwd;
   X -> fwd = X -> bwd;
b. X -> fwd = start;
   start -> bwd = X;
   start = X;
```

```
c. X -> bwd = X -> fwd;
    X -> fwd = X -> bwd;
   start = X;
d. X -> bwd -> bwd = X -> bwd;
    X -> fwd -> fwd = X -> fwd
```

16. Does C perform array out of bound checking? What is the output of the following program? [ ]

```
int main()
{
   int i;
   int arr[5] = {0};
   for (i = 0; i <= 5; i++)
      printf("%d ", arr[i]);
   return 0;
}
```

a) Compiler Error: Array index out of bound.
b) The always prints 0 five times followed by garbage value
c) The program always crashes.
d) The program may print 0 five times followed by garbage value, or may crash if address (arr+5) is invalid.

17. Which boolean expression indicates whether the numbers in two nodes (p and q) are the same. Assume that neither p nor q is null. [ ]

a) p == q
b) p.data == q.data
c) p.link == q.link
d) None of the above.

18. Which of the following statement is true [ ]

I. Using single linked list it is not possible to traverse the list in backward direction.

II. To find the predecessor it is required to traverse the list from the first node in case of single linked list.

a) I only b) II only c) Both I and II d) None of the above

19. Suppose each set is represented as a linked list with elements in arbitrary order. Which of the operations among union, intersection, membership, cardinality will be the slowest? [ ]

a) union only

b) intersection, membership

c) membership, cardinality

d) union, intersection

20. The following C function takes a singly linked list as input argument. It modifies the list by moving the last element to the front of the list and returns the modified list. Some part of the code is left blank.

```
Typedefstruct node
    {
     Intvalue;
     Structnode *next;
}Node;
Node *move_to_front(Node *head)
{
       Node *p, *q;
       If ((head == NULL: || (head->next == NULL))
                  Return head;
        Q = NULL;
         p = head;
       While (p->next !=NULL)
      {
          Q = p;
          P = p->next;
         }
            ---------------------------------------
     Return head;
}
```

Choose the correct alternative to replace the blank line. [ ]

a) q = NULL; p->next = head; head = p;

b) q->next = NULL; head = p; p->next = head;

c) head = p; p->next = q; q->next = NULL;

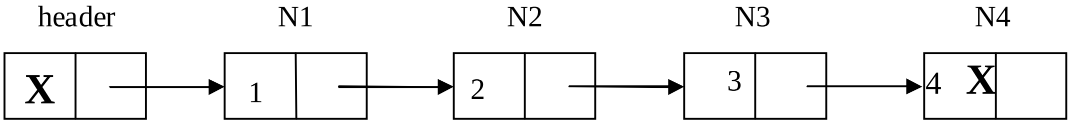
d) q->next = NULL; p->next = head; head = p;

SECTION-B

SUBJECTIVE QUESTIONS

1. Explain about delete operation in singly linked list.
2. Compare single linked list and circular single linked list.
3. Write an algorithm to perform deletion operation on circular linked list.
4. Write an algorithm to perform insertion operation on a double linked list.
5. Write an algorithm to perform deletion operation on a double linked list.
6. Write short notes on data structures.

7. **Consider the following single linked list.**

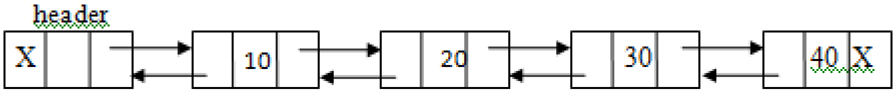


Demonstrate the following operations on this list and draw the updated single linked listafter each operation.

1. Insert 5 at end   2. Insert 6 at begin   3.Insert 9 after 2

4. Delete 6   5. Delete 5   6. Delete 3

8. Consider the following double linked list.

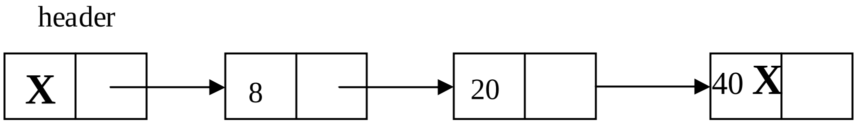


Illustrate the following operations on this list and draw the updated single linked list after each operation.

1. Insert 50 at end   2. Insert 60 at begin   3.Insert 90 after 20

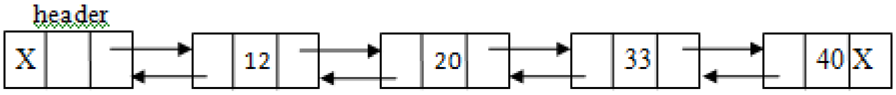
4. Delete 60   5. Delete 50   6. Delete 30

9. Consider the following single linked list.



Insert the following elements into the list **2,15,30,50**. Such that the list will be in **ascending**order and draw the updated single linked list after each insertion operation.

10. Consider the following double linked list.



Insert the following elements into the list **2,15,30,50**. Such that the list will be in **ascending**order and draw the updated single linked list after each insertion operation.

11. Write a program to implement insert operation in a doubly linked List.

12. Write a program to perform deletion operation in the middle of a doubly linked list.

13. Develop a program to delete an element of a single linked list.

14. Develop a program to merge two single linked lists into one list so that the resultant list will be inascending order.

SECTION-C

QUESTIONS AT THE LEVEL OF GATE

1. Consider the function f defined below. **(GATE 2003)**

```
struct item
{
  int data;
  struct item * next;
};
int f(struct item *p)
{
return( (p == NULL) || (p->next == NULL) || (( P->data <= p->next->data) && f(p->next)) );
}
```

For a given linked list p, the function f returns 1 if and only if

a) the list is empty or has exactly one element

b) the elements in the list are sorted in non-decreasing order of data value

c) the elements in the list are sorted in non-increasing order of data value

d) not all elements in the list have the same data value.

2. A circularly linked list is used to represent a Queue. A single variable p is used to access the Queue. To which node should p point such that both the operations enQueue and deQueue can be performed in constant time? **(GATE 2004)**

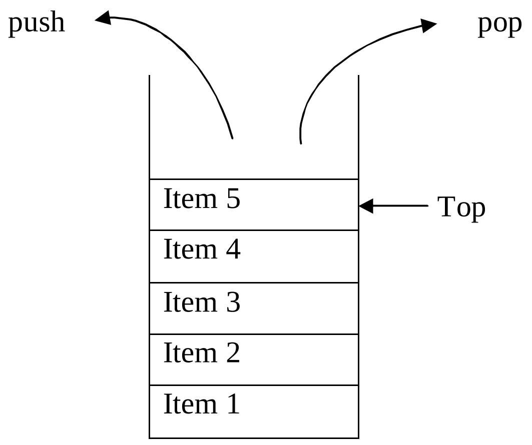
a) rear node b) front node c)not possible with a single pointer d) node next to front

3. In the worst case, the number of comparisons needed to search a singly linked list of length n for a given element is (GATE 2002)

a) log 2 n b) n/2 c) log 2 n – 1 d) n

STACKS

- Stack is a linear data structure.
- Stack is an ordered collection of homogeneous data elements, where insertion and deletion operations take place at only end.
- The insertion operation is termed as PUSH and deletion operation is termed as POP operation.
- The PUSH and POP operations are performed at TOP of the stack.
- An element in a stack is termed as ITEM.
- The maximum number of elements that stack can accommodate is termed as SIZE of the stack.
- Stack follows LIFO principle. i.e. Last In First Out.



Schematic diagram of a stack

Representation of stack

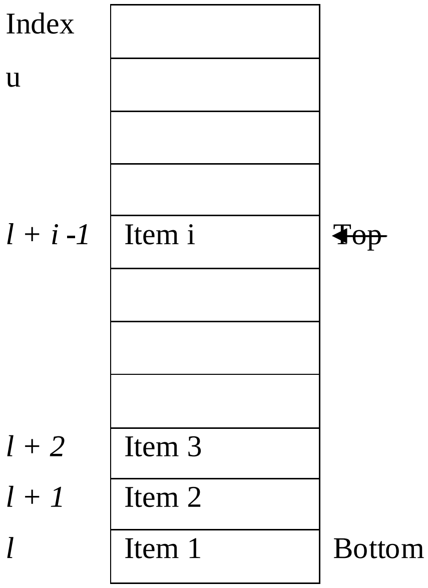
There are two ways of representation of a stack.

1. Array representation of a stack.
2. Linked List representation of a stack.

<u>1. Array representation of a stack.</u>

First we have to allocate memory for array.

Starting from the first location of the memory block, items of the stack can be stored in sequential fashion.



Array representation of stack

In the above figure item i denotes the ith item in stack.

l and u denotes the index ranges.

Usually *l* value is 1 and u value is size.

From the above representation tthe following two status can be stated.

Empty Stack: top < *l* i.e. top < 1

Stack is full: top > = *u*+*l*-1

i.e. top >= size + 1 -1

top >= size

Stack overflow

Trying to PUSH an item into full stack is known as stack overflow.

Stack overflow condition is top >= size

Stack underflow

Trying to POP an item from empty stack is known as Stack underflow.

Stack underflow condition is top < 1 or top = 0

Operations on Stack

PUSH	:	To insert element in to stack
POP	:	To delete element from stack

Status : To know present status of the stack

Algorithm Stack_PUSH(item)

Input: item is new item to push into stack

Output: pushing new item into stack at top whenever stack is not full.

```
1. if(top >= size)
    a) print(stack is full, not possible to perform push operation)
2. else
    a) top=top+1
    b) s[top]=item
3.End if
```

End Stack_PUSH

Algorithm Stack_POP( )

Input: Stack with some elements.

Output: item deleted at top most end.

```
1. if(top < 1)
    a) print(stack is empty not possible to pop)
2. else
    a) item=s[top]
    b) top=top-1
    c) print(deleted item)
3.End if
```

End Stack_POP

Algorithm Stack_Status( )

Input: Stack with some elements.

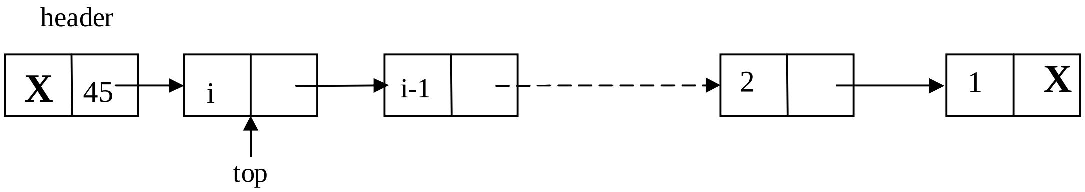
Output: Status of stack. i.e. Stack is empty or not, full or not, top most element in Stack.

```
1. if(top > = size)
    a) print(stack is full)
2. else if(top < 1)
    a. print(stack is empty)
3. else
    a) print(top most item in stack is s[top])
4. end if
```

End Stack_Status

2. Linked List representation of a stack

- The array representation of stack allows only fixed size of stack. i.e. static memory allocation only.
- To overcome the static memory allocation problem, linked list representation of stack is preferred.
- In linked list representation of stack, each node has two parts. One is data field is for the item and link field points to next node.



Linked List representation of stack

- Empty stack condition is

  top = NULL or header.link=NULL
- Full condition is not applicable for Linked List representation of stack. Because here memory is dynamically allocated**.**
- In linked List representation of stack, top pointer always points to top most node only. i.e. first node in the list.

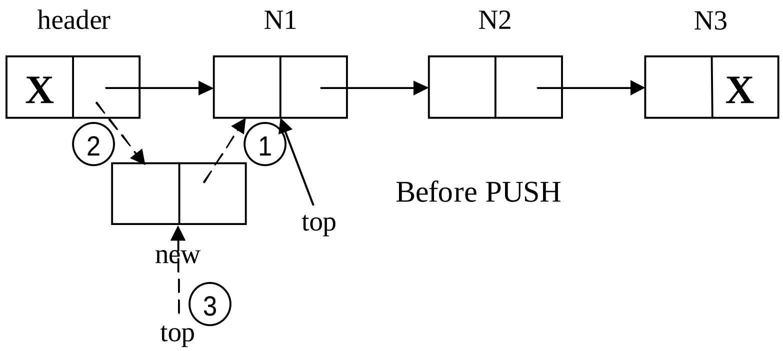
Operations on Stack with linked list representation

PUSH	:	To insert element in to stack
POP	:	To delete element from stack
Status	:	To know present status of the stack

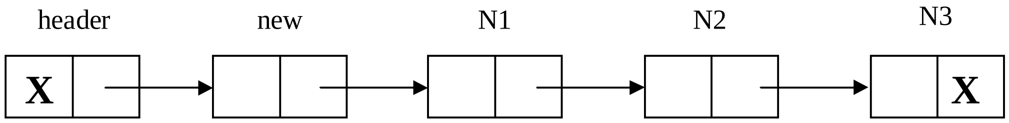
Algorithm Stack_PUSH_LL(item)

```
1. new = getnewnode( )
2. if( new = = NULL)
        a) print(Required node is not available in memeory)
3. else
        a)  new.link=header.link
        b)  header.link=new
        c) top=new
        d) new.data=item
```

End Stack_PUSH_LL



Before PUSH



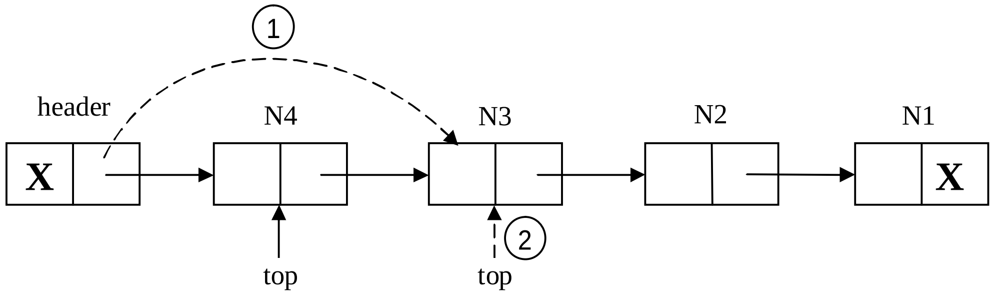
After PUSH

1. The link part of the new node is re placed with address of the previous top most node.
2. The link part of the header node is replaced with address of the new node.
3. Now the new node becomes top most node. So top is points to new node.

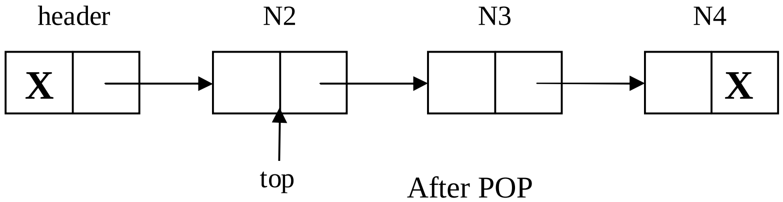
Algorithm Stack_POP_LL( )

```
    1. if( header.link = = NULL)
            a) print(Stack is empty, unable to perform POP operartion)
    2. else
            a) header.link=top.link
            b) item=top.data
            c) top=header.link
```

End Stack_POP_LL



Before POP



After POP

1. Link part of the header node is replaced with the address of the second node in the list.
2. After deletion of top most node from list, the second node becomes the top most node in the list. So top points to the second node.

Algorithm Stack_Status_LL( )

Input: Stack with some elements.

Output: **S**tatus of stack. i.e. Stack is empty or not, top most element in Stack.

```
1. if header.link = = NULL or top = = NULL)
        a) print(Stack is empty)
2. else
        a) print( Element present at top of stack is top.data)
3,end if
```

End Stack_Status_LL

Applications of stack

1. Factorial calculation
2. Infix to postfix conversion
3. Evaluation of postfix expression
4. Reversing list of elements

1. Factorial Calculation

- To calculate the factorial of a given number using stack, we require two stacks. One for storing the parameter n and another stack is hold the return address.
- Let us assume the two stacks, one PARAM for parameter and ADDR for return address.
- Assume PUSH(X,Y) operation for pushing the items X and Y into the stack PARAM and ADDR respectively.

Algorithm Factorial_Stack(n <integer>)

Input: An integer n

Output: Factorial of n

```
1.  Top = 0
2.  Addr = step10
3.  push(n,addr)
4.  n = n - 1 , addr = step8
5.  if ( n = = 0 OR n = = 1)
        a)  fact = 1
        b)  goto step9
6.  else
        a)  push(n,addr)
        b)  goto step4
7.  end if
8.  fact = fact * n
9.  n = pop_PARAM( ),  addr = pop_ADDR( )
10. goto addr
11. return(fact)
```

End Factorial_Stack

2. Infix to postfix conversion

An expression is a combination of operands and operators.

Eg. c= a + b

In the above expression a, b, c are operands and +, = are called as operators.

We have 3 notations for the expressions.

i. Infix notation
ii. Prefix notation
iii. Postfix notation

Infix notation: Here operator is present between two operands.

eg. a + b

The format for Infix notation as follows

<operand> <operator> <operand>

Prefix notation: Here operator is present before two operands.

eg. + a b

The format for Prefix notation as follows

<operator> <operand> <operand>

Postfix notation: Here operator is present after two operands.

eg. a b +

The format for Prefix notation as follows

<operand> <operand> <operator>

While conversion of infix expression to postfix expression, we must follow the precedence and associativity of the operators.

Operator	**Precedence**	**Associativity**
^ or $ (exponential)	3	Right to Left
* / %	2	Left to Right
+ -	1	Left to Right

In the above table * and / have same precedence. So then go for associativity rule, i.e. from Left to Right.

Similarly + and - same precedence. So then go for associativity rule, i.e. from Left to Right.

Eg. 1

(A + B) * (C – D)

A B + * (C – D)

A B + * C D –

A B + C D - *

Eg. 2

([(A - { B + C }) * D] $ E + F)

([(A - BC+) * D] $ E + F)

([ABC+- * D] $ E + F)

(ABC+-D* $ E + F)

(ABC+-D*E$ + F)

ABC=-D*E$F+

- To convert an infix expression to postfix expression, we can use one stack.
- Within the stack, we place only operators and left parenthesis only. So stack used in conversion of infix expression to postfix expression is called as operator stack.

Algorithm Conversion of infix to postfix

Input: Infix expression.

Output: Postfix expression.

```
1. Perform the following steps while reading of infix expression is not over
    a) if symbol is left parenthesis then push symbol into stack.
    b) if symbol is operand then add symbol to post fix expression.
    c) if symbol is operator then check stack is empty or not.
        i) if stack is empty then push the operator into stack.
        ii) if stack is not empty then check priority of the operators.
            (I) if priority of current operator > priority of operator present at top of stack then push operator into stack.
            (II) else if priority of operator present at top of stack >= priority of current operator then pop the operator present at top of stack and add popped operator to postfix expression (go to step I)
    d) if symbol is right parenthesis then pop every element form stack up corresponding left parenthesis and add the poped elements to postfix expression.
2. After completion of reading infix expression, if stack not empty then pop all the items from stack and then add to post fix expression.
```

End conversion of infix to postfix

3. Evaluation of postfix expression

- To evaluate a postfix expression we use one stack.
- For Evaluation of postfix expression, in the stack we can store only operand. So stack used in Evaluation of postfix expression is called as operand stack.

Algorithm PostfixExpressionEvaluation

Input: Postfix expression

Output: Result of Expression

```
    1. Repeat the following steps while reading the postfix expression.
        a) if the read symbol is operand, then push the symbol into stack.
        b) if the read symbol is operator then pop the top most two items of the stack
        and apply the operator on them, and then push back the result to the stack.
    2. Finally stack has only one item, after completion of reading the postfix expression.
    That item is the result of expression.
```

End PostfixExpressionEvaluation

4. Reversing List of elements

- A list of numbers can be reversed by reading each number from an array starting from 1st index and pushing into stack.
- Once all the numbers have been push into stack, the numbers can be poped one by one from stack and store ito array from the 1st index.

Algorithm Reverse_List_Stack(a <array>, n <intetger>)

Input : Array a with n elements

Output: Reversed List of elements

```
    1. i=1 , top =0
    2. while ( I <=n)
        a) top = top + 1
        b) s[top] = a[i]
        c) i = I + 1
    3. end while loop
    4. i = 1
    5. while( i <= n)
        a) a[i] = s[top]
        b) top = top – 1
        c) i = i + 1
    6. end while loop
```

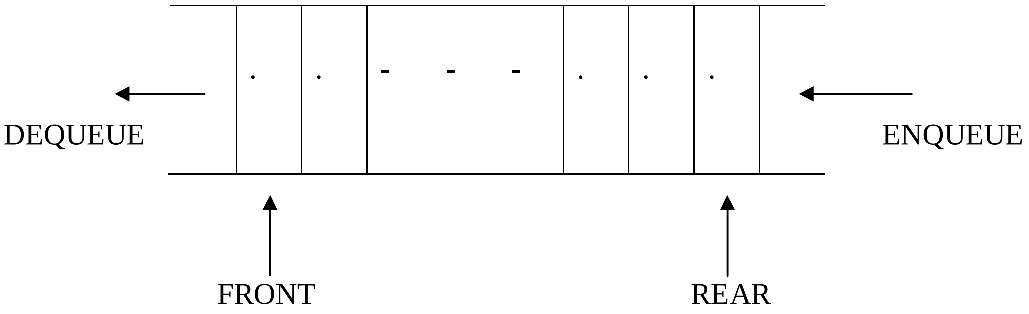
End Reverse_List_Stac

QUEUES

Queue is a linear Data structure.

Definition: Queue is a collection of homogeneous data elements, where insertion and deletion operations are performed at two extreme ends.

- The insertion operation in Queue is termed as ENQUEUE.
- The deletion operation in Queue is termed as DEQUEUE.
- An element present in queue are termed as ITEEM.
- The number of elements that a queue can accommodate is termed as LENGTH of the Queue.
- In the Queue the ENQUEUE (insertion) operation is performed at REAR end and DEQUEUE (deletion) operation is performed at FRONT end.
- Queue follows FIFO principle. i.e. First In First Out principle. i.e. a item First inserted into Queue, that item only First deleted from Queue, so queue follows FIFO principle.



Schematic Representation of Queue

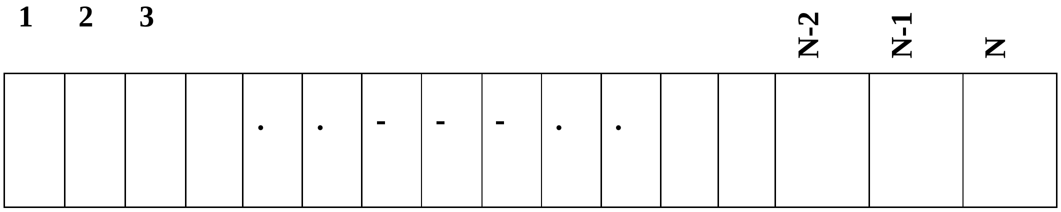
Representation of Queue

A Queue can be represented in two ways

1. Using arrays
2. Using Linked List

1. Representation of Queue using arrays

A one dimensional array Q[1-N] can be used to represent a queue.



Array representation of Queue

In array representation of Queue, two pointers are used to indicate two ends of Queue. The above representation states as follows.

1. Queue Empty condition

    Front = 0 and Rear = 0

2. Queue Full condition

    Rear = N where N is the size of the array we are taken

3. Queue conta[illegible]e element

    Front = = Rear

4. Number of items in Queue is

    Rear – Front + 1

Queue overflow: Trying to perform ENQUEUE (insertion) operation in full Queue is known as Queue overflow.

Queue overflow condition is Rear > = N

Queue Underflow: Trying to perform DEQUEUE (deletion) operation on empty Queue is known as Queue Underflow.

Queue Underflow condition is Front = 0

Operation on Queue

1. ENQUEUE	:	To insert element in to Queue
2. DEQUEUE	:	To delete element from Queue
3. Status	:	To know present status of the Queue

Algorithm Enqueue(item)

Input: item is new item insert in to queue at rear end.

Output: Insertion of new item queue at rear end if queue is not full.

```
1. if(rear = = N)
        a)print(queue is full, not possible for enqueue operation)
2. else
```

```
        i) if(front = = 0 and rear = = 0)  /* Q is Empty */
                a) rear=rear+1
                b) Q[rear]=item
                c) front=1
        ii) else
                a) rear=rear+1
                b) Q[rear]=item
        iii) end if
    3.end if
End Enqueue
```

While performing ENQUEUE operation two situations are occur.

1. if queue is empty, then newly inserting element becomes first element and last element in the queue. So Front and Rear points to first element in the list.
2. If Queue is not empty, then newly inserting element is inserted at Rear end.

Algorithm Dequeue( )

Input: Queue with some elements.

Output: Element is deleted from queue at front end if queue is not empty.

```
    1. if(front = = 0 and rear = = 0)
        a) print(Q is empty, not possible for dequeue operation)
    2. else
        i) if(front = = rear)            /* Q has only one element */
                a) item=Q[front]
                b) front=0
                c) rear=0
        ii) else
                a)item=Q[front]
                b)front=front+1
        iii) end if
        iv) print(deleted item is item)
    3. end if
End Dequeue
```

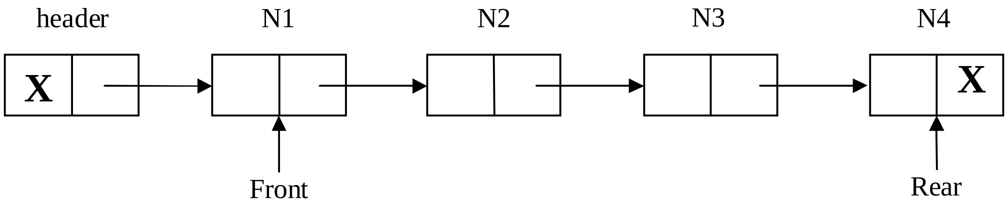
While performing DEQUEUE operation two situations are occur.

1. if queue has only one element, then after deletion of that element Queue becomes empty. So Front and Rear becomes 0.
2. If Queue has more than one element, then first element is deleted at Front end.

```
Algorithm Queue_Status( )
Input: Queue with some elements.
Output: Status of the queue. i.e. Q is empty or not, Q is full or not, Element at front end and rear end.
    1. if(front = = 0 and rear = = 0)
        a) print(Q is empty)
    2. else if (rear = = size )
        a) print(Q is full)
    3. else
        i) if(front = = rear)
            a) print(Q has only one item)
        ii) else
            a) print(element at front end is Q[front])
            b) print(element at rear end is Q[rear])
        iii) end if
    4. end if
End Queue_Status
```

2. Representation of Queue using Linked List

- Array representation of Queue has static memory allocation only.
- To overcome the static memory allocation problem, Queue can be represented using Linked List.



Linked List Representation of Queue

- In Linked List Representation of Queue, **Front** always points to **First** node in the Linked List and **Rear** always points to **Last** node in the Linked List.

The Linked List representation of Queue stated as follows.

1. Empty Queue condition is

    Front = NULL and Rear = NULL or header.link == NULL

2. Queue full condition is not available in Linked List representation of Queue, because in Linked List representation memory is allocated dynamically.

3. Queue has only one element

    Front = = Rear

Operation on Linked List Representation of Queue

1. ENQUEUE	:	To insert element in to Queue
2. DEQUEUE	:	To delete element from Queue
3. Status	:	To know present status of the Queue

Algorithm Enqueue _LL(item)

Input: item is new item to be insert.

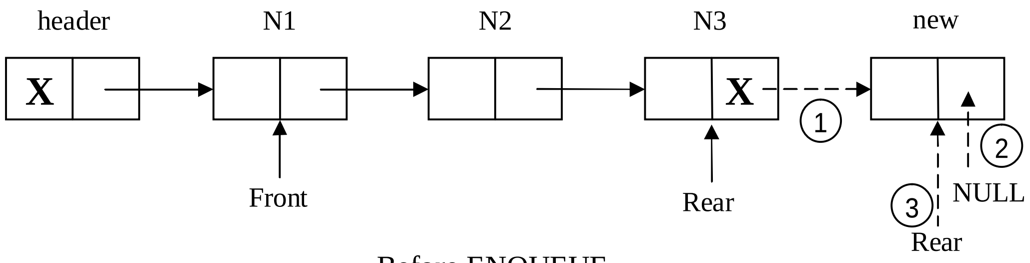
Output : new item i.e new node is inserted at rear end.

```
1. new=getnewnode()
2. if(new = = NULL)
        a) print(required node is not available in memory)
3. else
        i) if(front = = NULL and rear = = NULL)    /* Q is EMPTY */
                a) header.link=new
                b) new.link=NULL
```

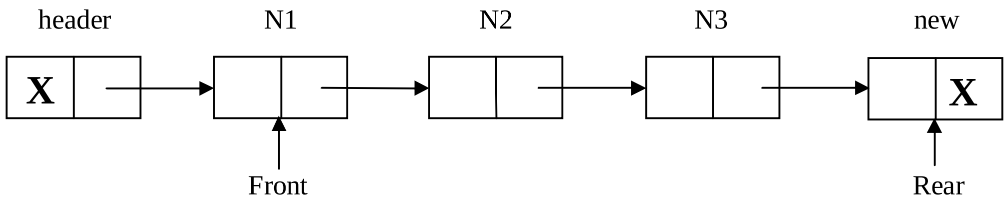
```
                        c) front=new
                        d) rear=new
                        e) new.data=item
                ii) else                                /* Q is not EMPTY */
                        a) rear.link=new            /* 1 */
                        b) new.link=NULL            /* 2 */
                        c) rear=new                 /* 3 */
                        d) new.data=item
                iii) end if
        4. end if
End_Enqueue_LL
```

While performing ENQUEUE operation two situations are occur.

1. if queue is empty, then newly inserting element becomes first node and last node in the queue. So Front and Rear points to first node in the list.
2. If Queue is not empty, then newly inserting node is inserted at last.



Before ENQUEUE



After ENQUEUE

1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with NULL, because new nodes becomes the last node.
3. Rear is points to last node in the list. i.e. newly inserted node in the list.

Algorithm Dequeue_LL( )

Input: Queue with some elements

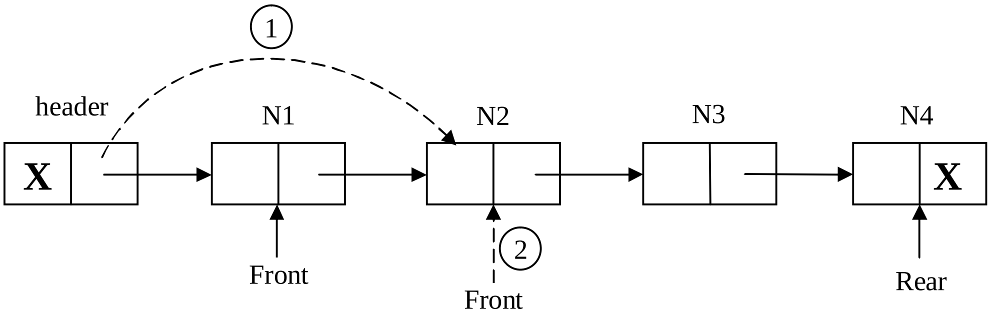
Output: Element is deleted at front end if queue is not empty.

```
    1.if(front= =NULL and rear = =NULL)
        a) print(queue is empty, not possible to perform dequeue operation)
    2. else
        i) if(front = = rear)               /* Q has only one element */
            a) header.link = NULL
            b) item=front.data
            c) front=NULL
            d) rear=NULL
        b) else                             /* Q has more than one element */
            a) header.link = front.link         /* 1 */
            b) item=front.data
            c) free(front)
            d)front=header.link                 /* 2 */
        c) end if
        d) print(deleted element is item)
    3. end if
```

End_Dequeue_LL

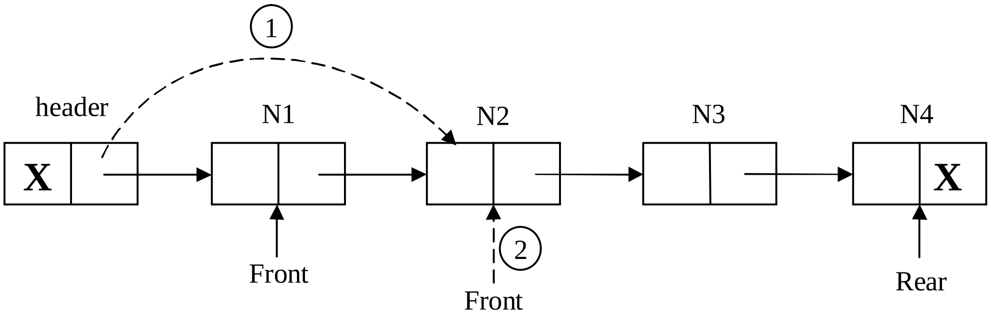
While performing DEQUEUE operation two situations are occur.

1. if queue has only one element, then after deletion of that element Queue becomes empty. So Front and Rear points to NULL.
2. If Queue has more than one element, then first node is deleted at Front end.

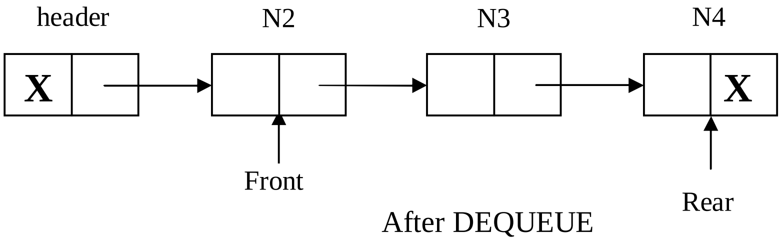


Before DEQUEUE

After DEQUEUE



Before DEQUEUE



After DEQUEUE

1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

2. Front is set to first node in the list.

Algorithm Queue_Status_LL

Input: Queue with some elements

Output: Status of the queue. i.e. Q is empty or not, Q is full or not, Element at front end and rear end.

```
    1. if(front = = NULL and rear = = NULL)
            a) print(Q is empty)
    2. else if(front = = rear)
            a) print(Q has only one item)
    3. else
            a) print(element at front end is front.data)
            b) print(element at rear end is rear.data)
    4. end if
```

End Queue_Status_LL

Queue operations using stack

To perform Queue operations using stack, we require two stacks named as stack1 and stack2. The Queue operations using stack can perform in two ways.

1. **Enqueue operation is cost effective**
2. **Dequeue operation is cost effective**

1. Enqueue operation is cost effective

Algorithm Enqueue_stack(item)

```
1. While stack1 is not empty, PUSH every element from stack1 to stack2.
2. PUSH item in to stack1.
3. While stack2 is not empty, PUSH every element from stack2 to stack1.
```

End Enqueue_stack

Algorithm Dequeue_stack( )

```
1. If stack1 is empty then error occurs. i.e. queue is empty.
2. Else POP an item from stack1.
```

End Dequeue_stack

2. Dequeue operation is cost effective

```
Algorithm Enqueue_stack(item)
    1. PUSH item into stack1
End Enqueue_stack
```

```
Algorithm Dequeue_stack( )
    1. If stack1 and stack2 are empty then error occur. i.e. Queue is empty.
    2. Else if stack2 is empty
            a) While stack1 is not empty, PUSH ever element from stack1 to stack2.
            b) POP element from stack2.
            c) While stack2 is not empty, PUSH ever element from stack2 to stack1
    3. End if
End Dequeue_stack
```

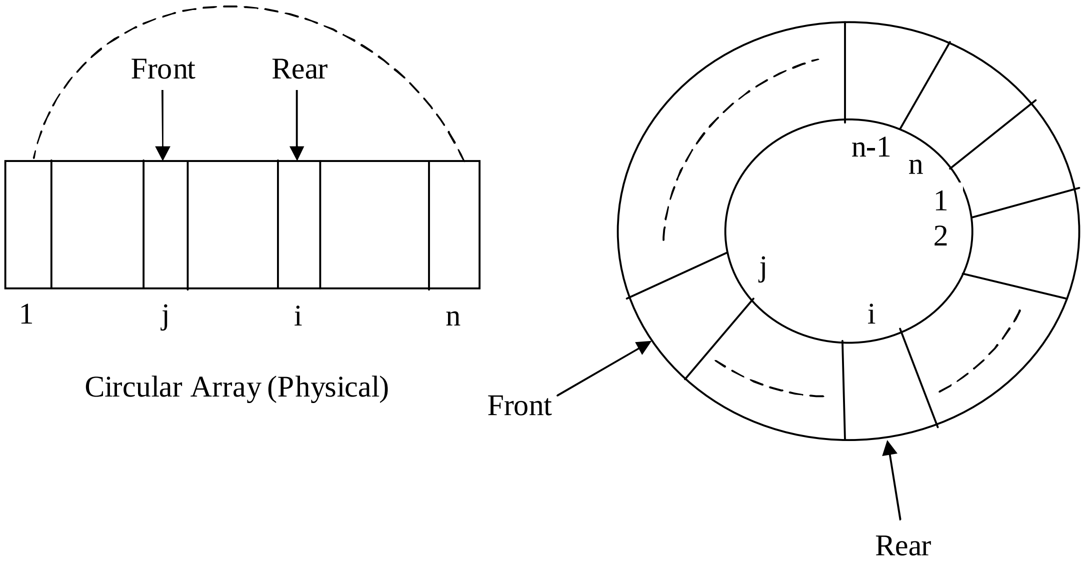
Various Queue Structures

1. **Circular Queues**
2. **Deque**
3. **Priority Queue**

1. Circular Queues

Physically a circular array is same as ordinary arra, say a[i-N], but logically it implements that a[1] comes after a[N] or a[N] comes after a[1].

The following figure shows the physical and logical representation for circular array



Circular Array (Physical)

Circular Queue (Logical)

Logical and physical view of a Circular Queue

- Here both Front and Rear pointers are move in clockwise direction. This is controlled by the MOD operation.
- For e.g. if the current pointer is at i, then shift next location will be (i mod LENTH) +1, 1<= i <= Length

Circular Queue empty condition is

Front = 0 and Rear = 0

Circular Queue is full

Front = = (Rear Mod Length) + 1

Algorithm CQ_Enqueue(item)

Input: item is new item insert in to Circular queue at rear end.

Output: Insertion of new item Circular queue at rear end if vqueue is not full.

```
1. next = (rear % N) = 1
2. if( front == next)
        a)print(Circular queue is full, not possible for enqueue operation)
```

```
    3. else
        i) if(front = = 0 and rear = = 0)  /* CQ is Empty */
            a) rear=( rear % N) = 1
            b) CQ[rear]=item
            c) front=1
        ii) else
            a) rear=( rear % N) = 1
            b) CQ[rear]=item
        iii) end if
    4.end if
```

End CQ_Enqueue

Algorithm CQ_Dequeue( )

Input: Circular Queue with some elements.

Output: Element is deleted from circular queue at front end if circular queue is not empty.

```
    1. if(front = = 0 and rear = = 0)
        a) print(CQ is empty, not possible for dequeue operation)
    2. else
        i) if(front = = rear)            /* Q has only one element */
            a) item=CQ[front]
            b) front=0
            c) rear=0
        ii) else
            a)item=CQ[front]
            b)front=(front % N)+1
        iii) end if
        iv) print(deleted item is item)
    3. end if
```

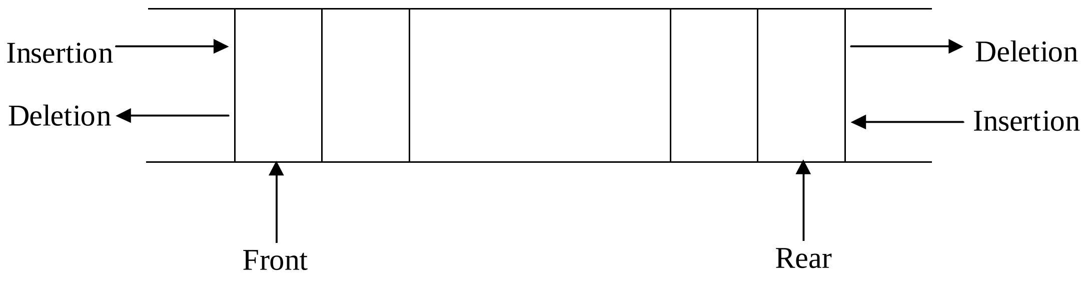
End CQ_Dequeue

2. Deque

Another variation of queue is known as DEQue.

In DEQue, both ENQUEUE (insertion) and DEQUEUE (deletion) operations can be made either of the ends.

DEQue is organized from Double Ended Queue.



A DEQue structure

Here DEQue structure is general representation of stack and Queue. In other words, a DEQue can be used as stack and Queue.

DeQue can be represented in two ways.

1. Using Double Linked List
2. Using a Circular Queue

Here Circular array is popular representation of DEQue.

On DEQue, the following four operations can be performed.

1.	PUSHDQ(item)	:	To insert item at FRONT end of DEQue.
2.	POPDQ( )	:	To delete the FRONT end item from DEQue.
3.	INJECT(item)	:	To insert item at REAR end of DEQue.
4.	EJECT( )	:	To delete the REAR end item from DEQue.

PUSHDQ(item)

If FRONT = 1, then next position of FRONT = Length.

(Here FRONT = 1 means FRONT points to extreme Left)

If FRONT = Length, then next position of FRONT = 1.

(Here FRONT = Length means FRONT points to extreme Right)

Otherwise, i.e. FRONT is at intermediate position. Then next position of FRONT = FRONT – 1.

Algorithm PUSHDQ(item)

Input: item is new item is inserted in to Queue at FRONT end.

Output: New item is inserted in to Queue at FRONT end.

```
        If (front = = 1)
        (i) next = Length
        else if ( front = = Length    OR   front = = 0)
        (i) next = 1
        else
        (i) next = front – 1
        end if
        if( next  = rear)
        (i) Print( Queue is full)
        else
        (i) Front = next
        (ii) DQ[front] = item
        end if
```

End PUSHDQ

<u>POPDQ( )</u> : This algorithm is same as CQDeque( ) algorithm.

<u>INJECT(item)</u> : This algorithm is same as CQEnque( ) algorithm.

<u>EJECT( )</u>

If REAR = 1, then next position of REAR = Length.

(Here REAR = 1 means REAR points to extreme Left)

If REAR = Length, then next position of REAR = 1.

(Here REAR = Length means REAR points to extreme Right)

Otherwise, i.e. REAR is at intermediate position. Then next position of REAR = REAR – 1.

Algorithm EJECT( )

Input : A DEQUE with item.

Output: An item is deleted from REAR end.

```
    if( front == 0   and   rear = = 0)
    (i) print( Queue is empty, not possible to delete)
    else
    (i) if( front = = rear)
```

```
            (a) item = DQ[item]
                  Front = 0
            (c) Rear = 0
          else                                  /* DEQue has more than one element */
            (a) if(rear = 1)
                        (A) item = DQ[rear]
                        (B) rear = Length
            (b) else if( rear = = Length)
                         (A) item = DQ[rear]
                         (B) rear = 1
             (c) else
                         (A) item = DQ[rear]
                         (B) rear = rear -1
             (d) End if
        (iii) End if
      End if
End EJECT
```

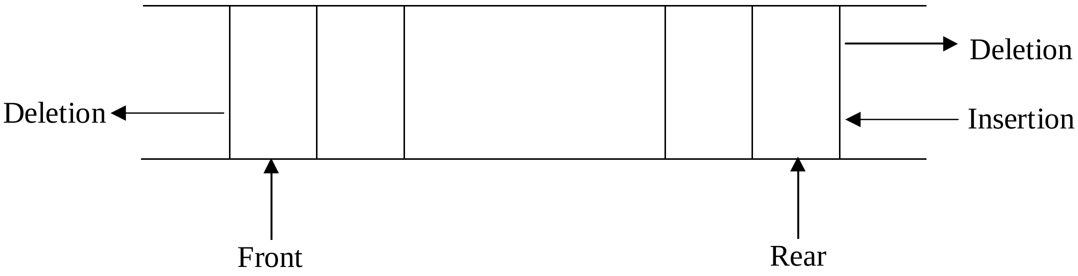
There are two variations of DEQue known as

Input restricted DEQue

Output restricted DEQue

Input restricted DEQue

Here DEQue allows insertion at one end (say REAR end) only, but allows deletion at both ends.

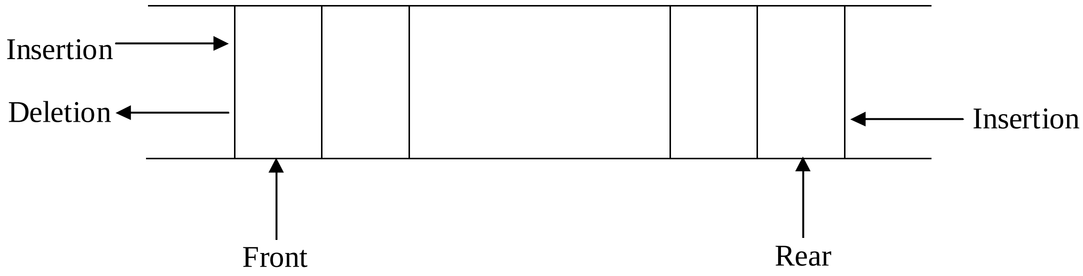


Input restricted DEQue

Output restricted DEQue

Here DEQue allows deletion at one end (say FRONT end) only, but allows insertion at both ends.

DEQue is organized from Double Ended Queue.



Output restricted DEQue

\

UNIT-III
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1) To add and remove nodes from a queue __________ access is used. [ ]

a.) LIFO, Last In First Out b). FIFO, First In First Out
c). Both a and b d) . None

2.)Which one of the following is an application of Queue Data Structure?
a) When a resource is shared among multiple consumers. [ ]
b) When data is transferred asynchronously
c) Load Balancing
d) All of the above

3.)Which of the following is not the type of queue? [ ]
a) Ordinary queue b) Single ended queue c) Circular queue d) Priority queue

4.) suppose a circular queue of capacity (n – 1) elements is implemented with an array of n elements. Assume that the insertion and deletion operation are carried out using REAR and FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect queue full and queue empty are
a) Full: (REAR+1) mod n == FRONT, empty: REAR == FRONT [ ]
b) Full: (REAR+1) mod n == FRONT, empty: (FRONT+1) mod n == REAR
C) Full: REAR == FRONT, empty: (REAR+1) mod n == FRONT
d) Full: (FRONT+1) mod n == REAR, empty: REAR == FRONT

5.)What is the need for a circular queue? [ ]
a) effective usage of memory b) easier computations
c) all of the mentioned d.)none

6.What is the space complexity of a linear queue having n elements? [ ]
a) O(n) b) O(nlogn) c) O(logn) d) O(1)

7).In linked list implementation of a queue, where does a new element be deleted? [ ]

a) At the head of linked list b) At the tail of the linked list

c) At the centre position in the linked list d) None of the above

8.)In a circular queue, how do you increment the rear end of the queue?

a) rear++ b) (rear+1) % CAPACITY [ ]

c) (rear % CAPACITY)+1 d) rear–

9. In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into a NONEMPTY queue? [ ]

a) Only front pointer b) Only rear pointer c) Both front and rear pointer
d) None of the mentioned

10.The value of REAR is increased by 1 when ……. [ ]

a. An element is deleted in a queue b. An element is traversed in a queue

c. An element is added in a queue d.None

11.)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)

12.)Which of the following is true about linked list implementation of stack?

a) In push operation, if new nodes are inserted at the beginning of linked list, then in pop operation, nodes must be removed from end. [ ]
b) In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from the beginning.
c) Both of the above
d) None of the above

13.)Suppose a stack is to be implemented with a linked list instead of an array. What would be the effect on the time complexity of the push and pop operations of the stack implemented using linked list (Assuming stack is implemented efficiently)?

a) O(1) for insertion and O(n) for deletion [ ]
b) O(1) for insertion and O(1) for deletion
c) O(n) for insertion and O(1) for deletion
d) O(n) for insertion and O(n) for deletion

14.)Which of the following permutation can be obtained in the same order using a stack assuming that input is the sequence 5, 6, 7, 8, 9 in that order? [ ]
a) 7, 8, 9, 5, 6 b) 5, 9, 6, 7, 8 c) 7, 8, 9, 6, 5 d) 9, 8, 7, 5, 6

15.)If the sequence of operations – push (1), push (2), pop, push (1), push (2), pop, pop, pop, push (2), pop are performed on a stack, the sequence of popped out values [ ]
a) 2,2,1,1,2 b) 2,2,1,2,2 c) 2,1,2,2,1 d) 2,1,2,2,2

16.)The postfix form of the expression (A+ B)*(C*D- E)*F / G is? [ ]
a) AB+ CD*E – FG /** b) AB + CD* E – F **G /
c) AB + CD* E – *F *G / d) AB + CDE * – * F *G /

17.)The postfix form of A*B+C/D is? [ ]
a) *AB/CD+ b) AB*CD/+ c) A*BC+/D d) ABCD+/*

18.The prefix form of A-B/ (C * D ^ E) is? [ ]
a) -/*^ACBDE b) -ABCD*^DE c) -A/B*C^DE d) -A/BC*^DE

19.) The result of evaluating the postfix expression 5, 4, 6, +, *, 4, 9, 3, /, +, * is? [ ]
a) 600 b) 350 c) 650 d) 588

20.)Which of the following data structures can be used for parentheses matching? [ ]
a) n-ary tree b) queue c) priority queue d) stack

SECTION-B

SUBJECTIVE QUESTIONS

1 .Explain the prefix and post fix notation of (a + b) * (c + d) ?

2 . Define what is stack? Why do we use stack ? And what are the operations performed on stacks?

3. Convert the expression **(a+b)/d-((e-f)%g)** into reverse polish notation using stack and show the contents of stack for every operation.

4. Evaluate the expression **12/3*6+6-6+8%2** using stack.

5. Convert the expression **a+b*c/d%e-f** into postfix expression using stack.

6. Implement queue using arrays?

7 .Implement queue using Linked List?

8. What is Queue? discuss the types of Queues ?And explain why we we are going for circular queue?

9.Lsit out Applications of Stacks?

10.List out applications of queues?

SECTION-C

QUESTIONS AT THE LEVEL OF GATE

1.Consider the following pseudocode that uses a stack

```
declare a stack of characters
while ( there are more characters in the word to read )
{
  read a character
  push the character on the stack
}
while ( the stack is not empty )
{
  pop a character off the stack
  write the character to the screen
}
```

What is output for input "geeksquiz"?

What is output for input “geeksquiz”? [ ]

(A) geeksquizgeeksquiz
(B) ziuqskeeg
(C) geeksquiz
(D) ziuqskeegziuqskeeg

2. Assume that the operators +, -, × are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, x , +, -. The postfix expression corresponding to the infix expression a + b × c - d ^ e ^ f is

abc × + def ^ ^ - [ ]

abc × + de ^ f ^ -

ab + c × d - e ^ f ^

- + a × bc ^ ^ def

3.The following postfix expression with single digit operands is evaluated using a stack: [ ]

```
8 2 3 ^ / 2 3 * + 5 1 * -
```

Note that ^ is the exponentiation operator. The top two elements of the stack after the first * is evaluated are:

(A) 6, 1 (B) 5, 7 (C) 3, 2 (D) 1, 5

Unit –IV

TREES

Objective:

- To impart knowledge of linear and non-linear data structures.

Syllabus:

Binary Trees: Basic tree concepts, properties, representation of binary trees using arrays and linked list, binary tree traversals.

Binary Search Trees: Basic concepts, BST operations: search, insertion, deletion and traversals, creation of binary search tree from in-order and pre (post) order traversals.

Learning Outcomes:

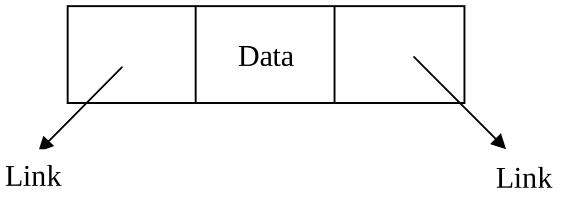
At the end of the unit student will be able to:

1. represent Binary Tees using Arrays and Linked Lists.
2. implement operations on Binary Search Trees.
3. construct Binary Search Trees from its Traversals

Learning Material

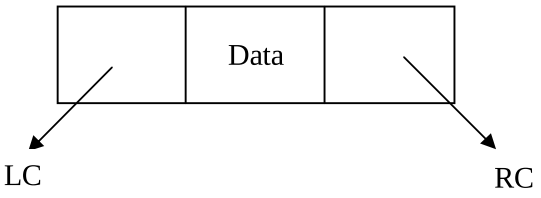
Basic Terminology:

1. Node: It is a main component of tree. It stores the actual data and links to other nodes.



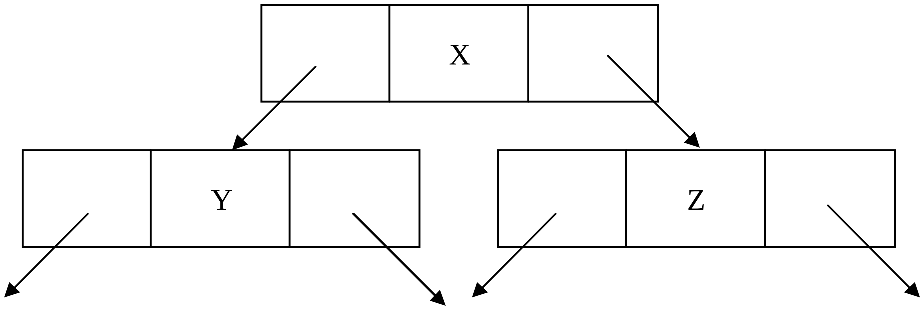
Structure of a node in Tree

2. Link / Edge / Branch: Link is point to other nodes in a tree.



Here *LC* Points To **Left Child** and *RC* Points To **Right Child**.

3. Parent Node: The Immediate Predecessor of a Node is called as Parent Node.

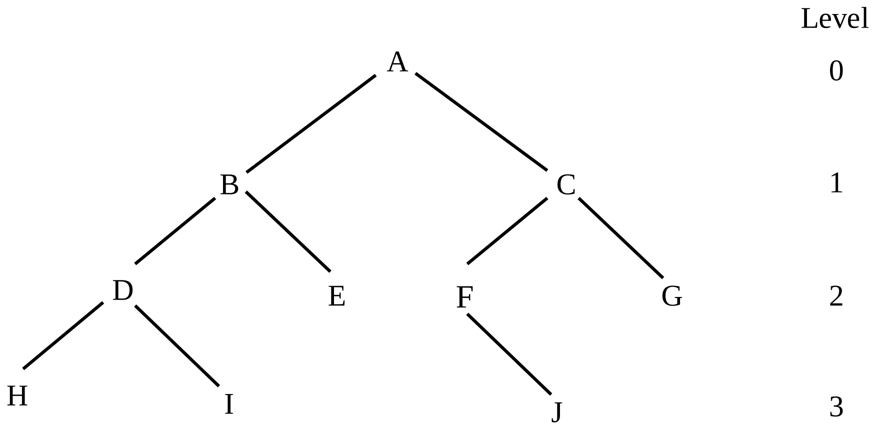


Here X is Parent Node to Node Y and Z.

4. Child Node: The Immediate Successor of a Node is called as Child Node.

In the above diagram Node Y and Z are child nodes to node X.

5. Root Node: Which is a specially designated node, and does not have any parent node.



In the above diagram node A is a Root Node.

6. Leaf node: The node which does not have any child nodes is called leaf node.

- In the above diagram node H, I, E, J, G are Leaf nodes.

7. Level: It is the rank of the hierarchy and the Root node is present at level **0**. If a node is present at level ***l*** then its parent node will be at the level ***l*-1**and child nodes will present at level ***l*+1**.

8. Height / Depth: The number of nodes in the longest path from Root node to the Leaf node is called the height of a tree.

- Height of above tree is 4.
- Height of a tree can be easily obtained as l_{max} + 1. Where l_{max} is the maximum level of a tree.
- In the above example l_{max}= 3. So height = 3 + 1 = 4

9. Siblings: The nodes which have same parent node are called as siblings.

- In the above example nodes B and C are siblings, nodes D and E are siblings , nodes F and G are siblings , nodes H and I are siblings.

10. Degree / Arity: Maximum number of child nodes possible for anode is called as degree of a node.

TREE:

Tree is a nonlinear data structure.

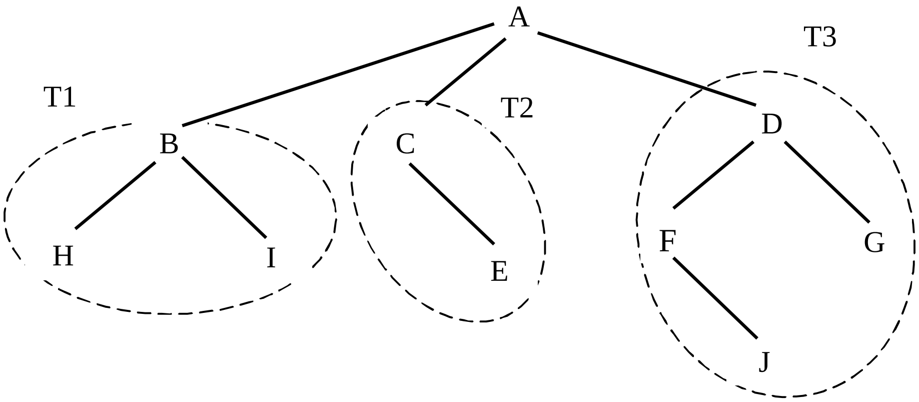
Definition

A tree T is a finite set of one or more nodes such that:

(i) There is a special node called as *root* node.

(ii) The remaining nodes are partitioned into ***n*** disjoint sets T_1, T_2, T_3. . . $T_{n.}$ where n>0.Where each disjoint set is a tree.

T_1, T_2, T_3. . .T_n are called as *sub trees*.



A sample Tree

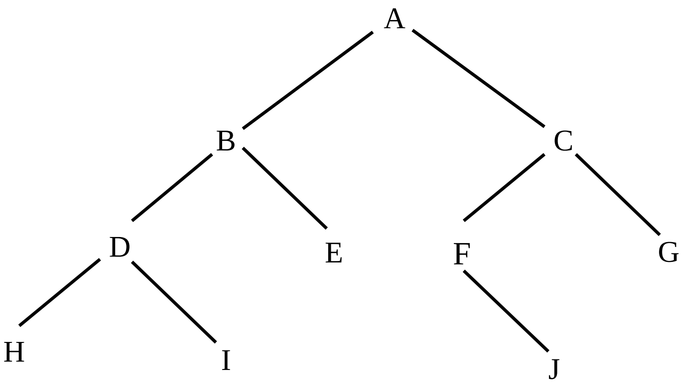
BINARY TREE: Is a special form of a tree.

Definition:

A binary tree is T is a finite set of nodes such that,

(i) T is empty called as empty binary tree.

(ii) T has specially designed node called as Root Node and remaining node of binary tree are partitioned into 2 disjoint sets. One is Left sub tree and another one is Right sub tree.



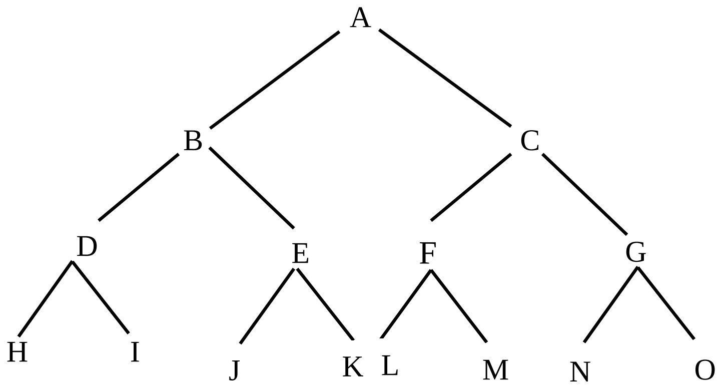
A sample Binary Tree

TWO SPECIAL CASES OF BINAERY TREE

1. Full binary tree
2. Complete binary tree

1. Full Binary Tree: a binary tree is said to be full binary tree, if each level has maximum number of possible nodes.

Eg:

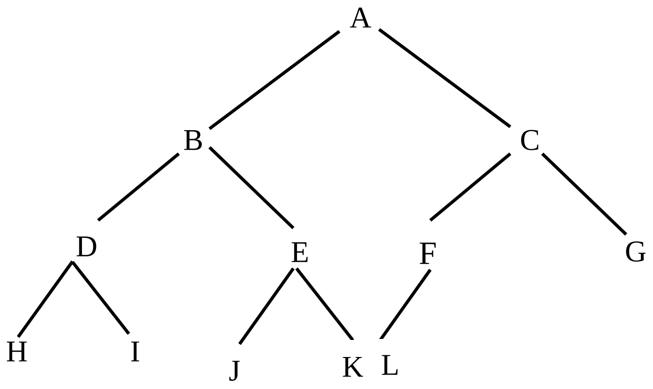


Level	Node’s
0	$2^0= 1$
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$

A Full Binary Tree of height 4

2. Complete Binary Tree: A binary tree is said to be complete binary tree, if all levels except the last level has maximum number of possible nodes, last level nodes are appeared as far left as possible.

Eg:

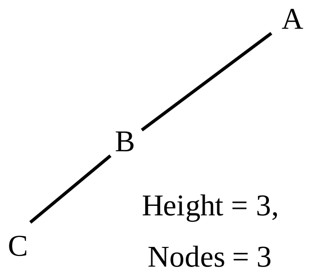


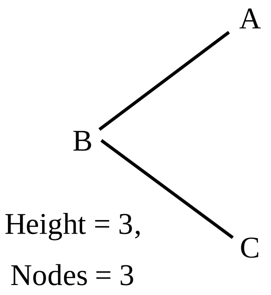
Level	Node’s
0	$2^0= 1$
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$

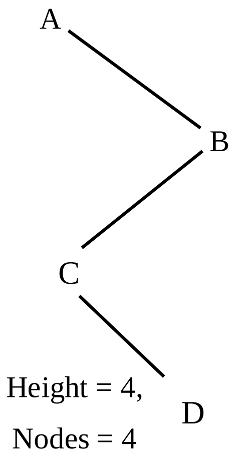
A Complete Binary Tree of height 4

Properties of Binary Trees

1. In any binary tree, maximum number of nodes on level *l* is 2^l (where *l*>= 0).
2. Maximum number of nodes possible in a binary tree of height *h* is 2^h-1.
3. Minimum number of nodes possible in a binary tree of height *h* is *h*.







- Whenever every parent node has only one child, such kind of binary trees are called as ***Skew binary trees***.

4. For any non-empty binary tree, if ***n*** is the number of nodes and ***e*** is the number of edges, then ***n = e + 1***.

   i.e. number of nodes = number of edges +1.
5. For any non-empty binary tree T, if $\boldsymbol{n_0}$ is the number of leaf nodes (degree = 0) and $\mathbf{n_2}$ is the number of intermediate nodes (degree = 2) then $\mathbf{n_0 = n_2 + 1}$.

   i.e. number of leaf nodes = number of non-leaf nodes + 1.
6. The height of a complete binary tree with **n** nodes is $\lceil \log_2(n+1) \rceil$
7. Total number of binary trees possible with **n** number of nodes is $\frac{1}{n+1} 2n_{c_n}$
8. The maximum and minimum size that an array may require to store a binary tree with ***n*** number of nodes are:

   Maximum size =$2^n - 1$

   Minimum size =$2^{\lceil \log_2(n+1) \rceil} - 1$
9. In a linked list representation of binary tree, if there are ***n*** number of nodes, then the number of *NULL* link are ***λ = n +1.***

Representation of Binary Tree

Binary tree can be represented in two ways.

1. Linear (or) Sequential representation using arrays.
2. Linked List representation using pointers.

1. Linear (or) Sequential representation using arrays

- In this representation, a block of memory for an array is to be allocated before going to store the actual tree in it.
- Once the memory is allocated, the size of the tree will be restricted to memory allocated.
- In this representation, the nodes are stored level by level starting from zero level, where only *ROOT* node is present.
- The *ROOT* node is stored in the first memory location. i.e. first element in the array.

The following rules are used to decide the location on any node of tree in the array. (Assume the array index start from 1)

1. The ROOT node is at index**1**.
2. For any node with index i, $1 \leq i \leq n$.

(a) **Parent (i)** = $\left\lfloor \frac{i}{2} \right\rfloor$

For the node when i = 1, there is ***no parent node***.

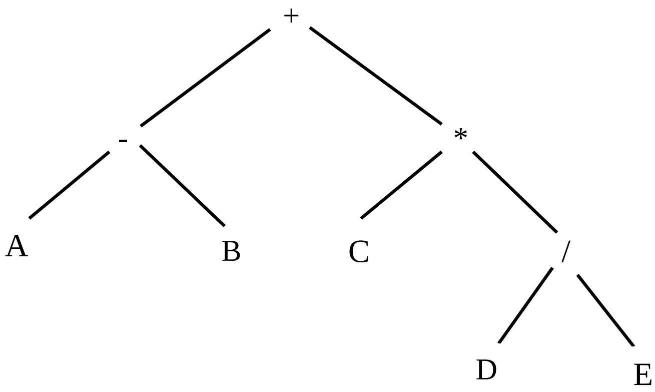
(b) **LCHILD (i)** = 2 * i

If 2 * i > n then **i** has ***no left child***.

(c) **RCHILD (i)** = 2*i +1

If 2*i + 1 > n then **i** has ***no right child***.

Eg. (A-B) + C * (D / E)



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
+	-	*	A	B	C	/							D	E

Array representation of above binary tree.

Advantages of Linear representation of Binary Tree

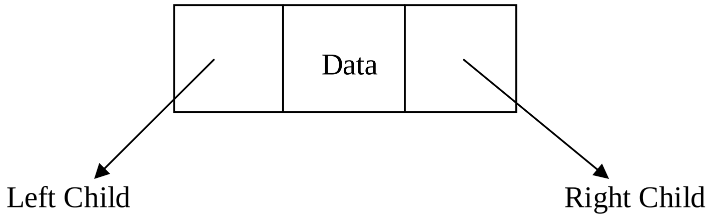
1. Any node can be accessed from any other node by calculating the index and this is efficient from execution point of view.
2. Here only data is stored without any pointers to their successor (or) predecessor.

Disadvantages of Linear representation of Binary Tree

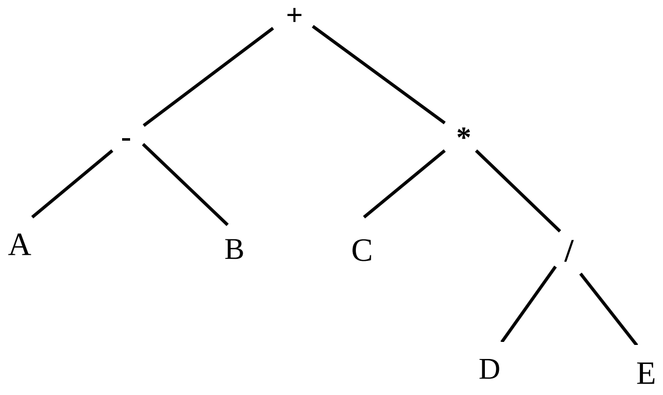
1. Other than full binary tree, majority of entries may be empty.
2. It allows only static memory allocation.

2. Linked List representation of Binary Tree using pointers

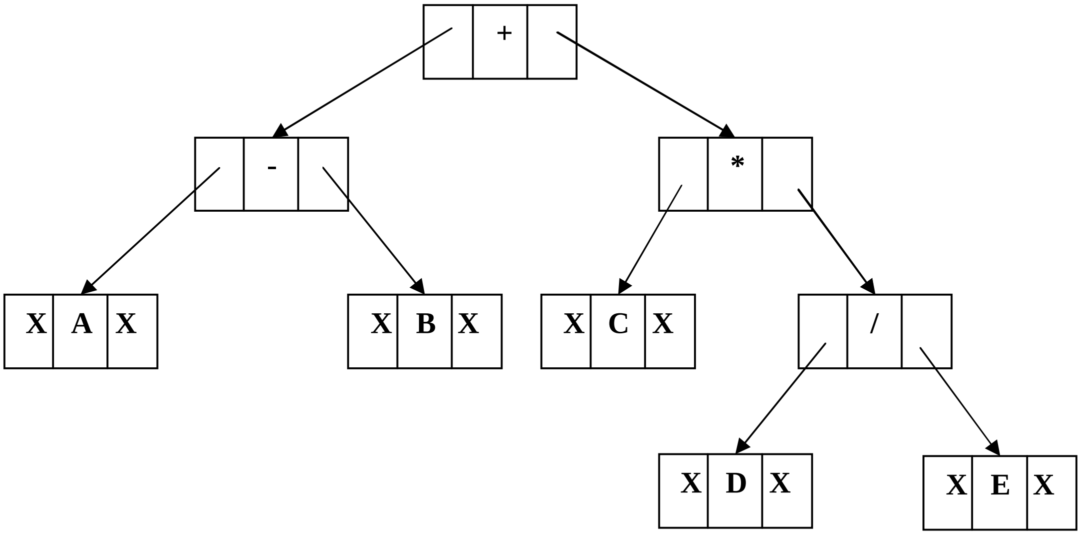
- Linked list representation of assumes structure of a node as shown in the following figure.



- With linked list representation, if one knows the address of *ROOT* node, then any other node can be accessed.



Binary Tree



Linked List representation of Binary Tree

Advantages of Linked List representation of Binary Tree

1. It allows dynamic memory allocation.
2. We can overcome the drawbacks of linear representation.

Disadvantages of Linked List representation of Binary Tree

1. It requires more memory than linear representation. i.e. Linked list representation requires extra memory to maintain pointers.

Binary Tree Traversals

Traversal operation is used to visit each node present in binary tree exactly once.

A Binary tree can be traversed in 3 ways.

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

1. Preorder traversal

Here first **ROOT** node is visited, then **LEFT** sub tree is visited in *Preorder* fashion and then **RIGHT** sub tree is visited in *Preorder* fashion.

i.e. *ROOT*, LEFT, RIGHT (or) ROOT, RIGHT, LEFT

2. Inorder traversal

Here first **LEFT** subtree is visited in *inorder* fashion, then **ROOT** node is visited and then **RIGHT** sub tree is visited in *inorder* fashion.

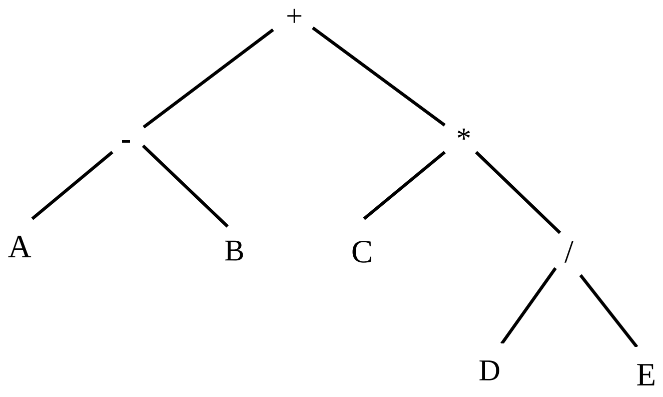
i.e. LEFT, *ROOT*, RIGHT (or) RIGHT, *ROOT*, LEFT

3. Postorder traversal

Here first **LEFT** subtree is visited in *postorder* fashion, then **RIGHT** sub tree is visited in *postorder* fashion and then **ROOT** node is visited.

i.e. LEFT, RIGHT, *ROOT* (or) RIGHT, LEFT, *ROOT*

Eg:



Preorder traversal	:	+ **-** A B * C / D E
Inorder traversal	:	A – B + C * D / E
Postorder traversal	:	A B – C D E / * +

Recursive Binary Tree Traversals

Algorithm preorder(ptr)

Input: Binary Tree with some nodes.

Output: *preorder* traversal of given Binary Tree.

```
1. if(ptr != NULL)
        a) print(ptr.data)
        b) preorder(ptr.lchild)
        c) preorder(ptr.rchild)
2. end if
```

End preorder

Algorithm inorder(ptr)

Input: Binary Tree with some nodes.

Output: *inorder* traversal of given Binary Tree.

```
1. if(ptr != NULL)
```

```
        a) inorder(ptr.lchild)
        b) print(ptr.data)
        c) inorder(ptr.rchild)
    2. end if
End inorder
```

```
Algorithm postorder(ptr)
Input: Binary Tree with some nodes.
Output: postorder traversal of given Binary Tree.
    1. if(ptr != NULL)
        a) postorder(ptr.lchild)
        b) postorder(ptr.rchild)
        c) print(ptr.data)
    2. end if
End postorder
```

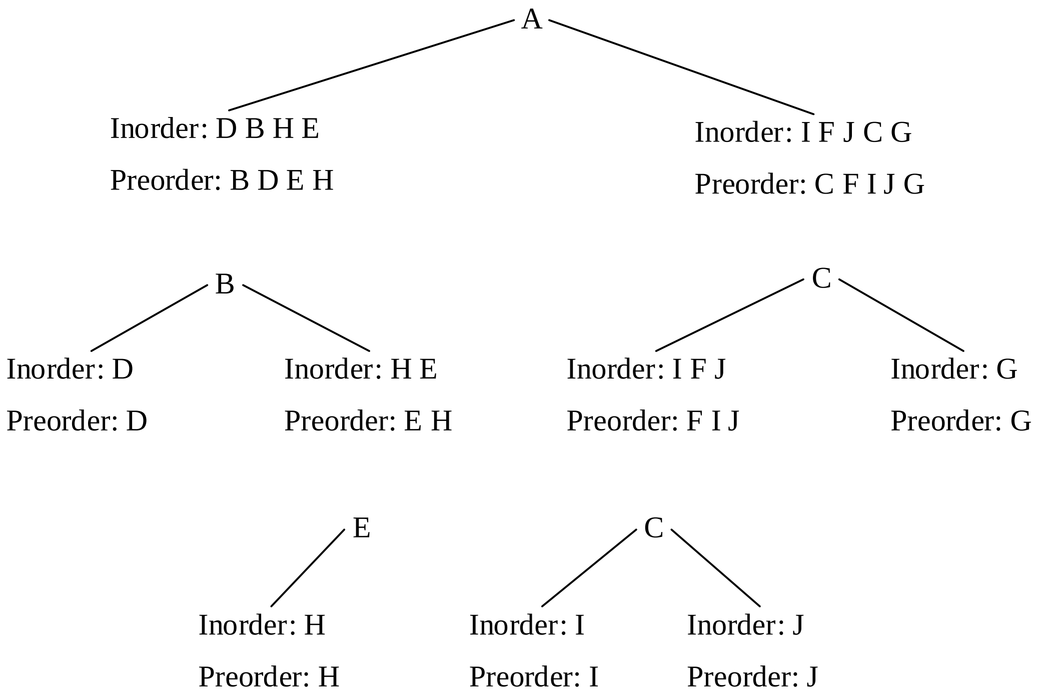
Creation of Binary Tree from its Tree traversals

- A binary tree can be constructed from its traversals.
- If the *Preorder* traversals is given, then the ***first node*** is *ROOT* node and *Postorder* traversal is given then ***last node*** is the *ROOT* node.
- For construction of a binary tree from its traversals, two traversals are essentials. Out of which one should be *inorder* traversal and another one is either *preorder* (or) *postorder* traversal.
- If preorder and post order is given to construct a binary tree, then binary tree can't be obtain *uniquely*.

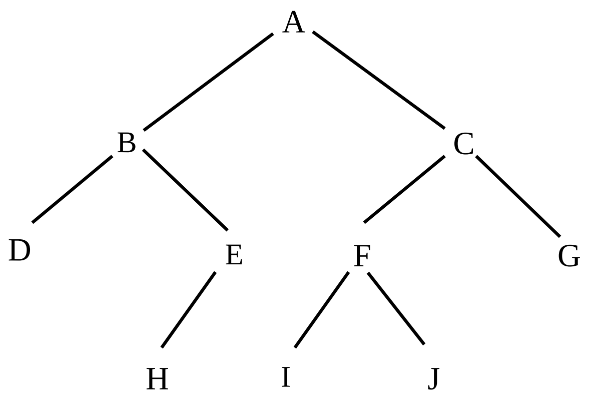
Eg.1.

Inorder:	D	B	H	E	A	I	F	J	C	G
Preorder:	A	B	D	E	H	C	F	I	J	G

1. From the preorder traversal, *A* is the *ROOT* node.
2. In the inorder traversal, all the nodes which are LEFT side of **A** belongs to LEFT sub tree and those node which are RIGHT side of **A** belongs to RIGHT sub tree.
3. Now the problem is reduced to two sub trees and same procedure can be applied repeatedly.

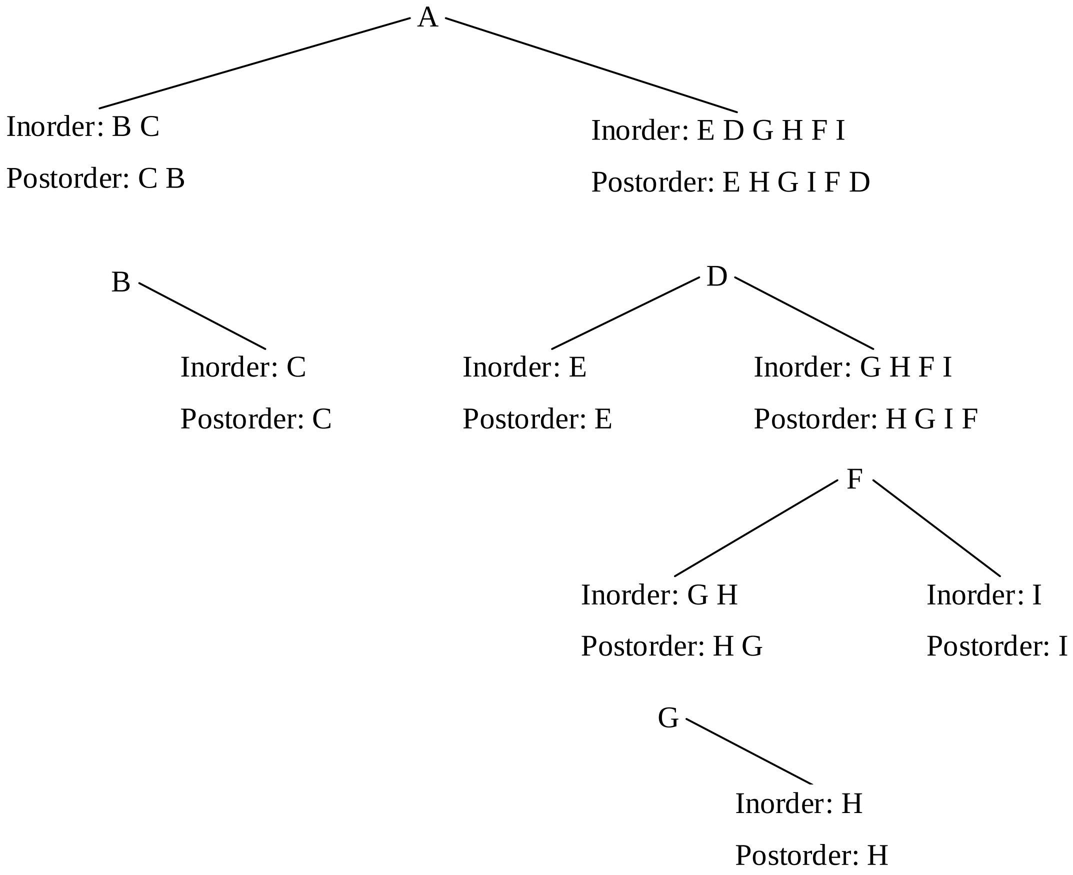


Final Binary tree from the Inorder and Preorder as follows:

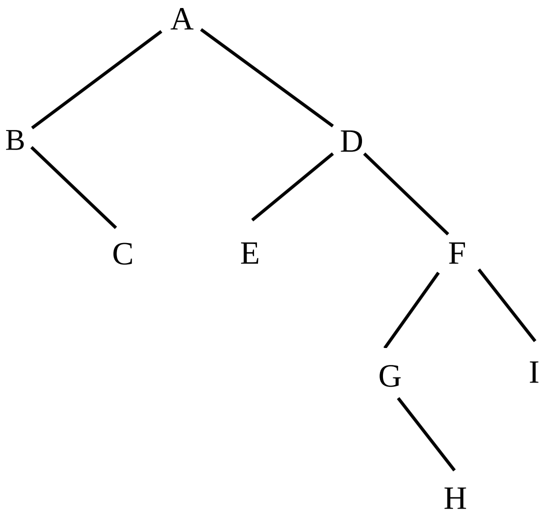


Eg. 2.

Inorder:	B	C	A	E	D	G	H	F	I
Postorder:	C	B	E	H	G	I	F	D	A



Final Binary tree from the Inorder and Postorder as follows:



Types of Binary Tress

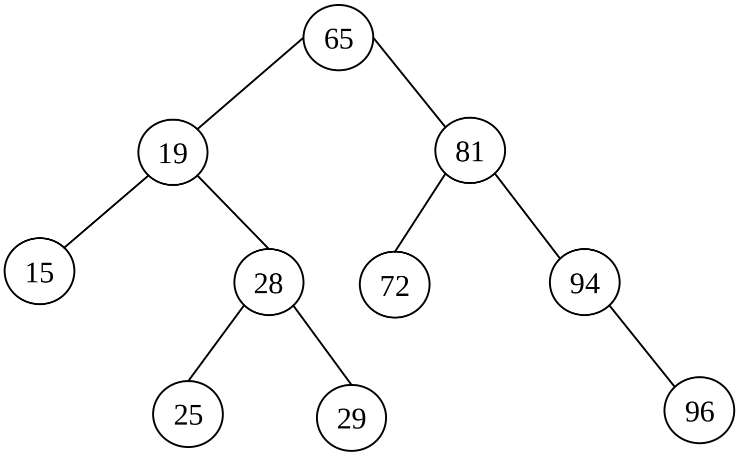
1. Expression Trees
2. Binary Search Trees
3. Threaded Binary Trees
4. Heap Tree
5. Height Balanced Binary Trees
6. Decision Trees
7. Huffman Tree

Binary Search Tree

Definition:

A binary tree T is termed binary search tree (or binary sorted tree) if each node N of T satisfies the following property:

The value at N is greater than every value in the left sub-tree of N and is less than every value in the right sub-tree of N.



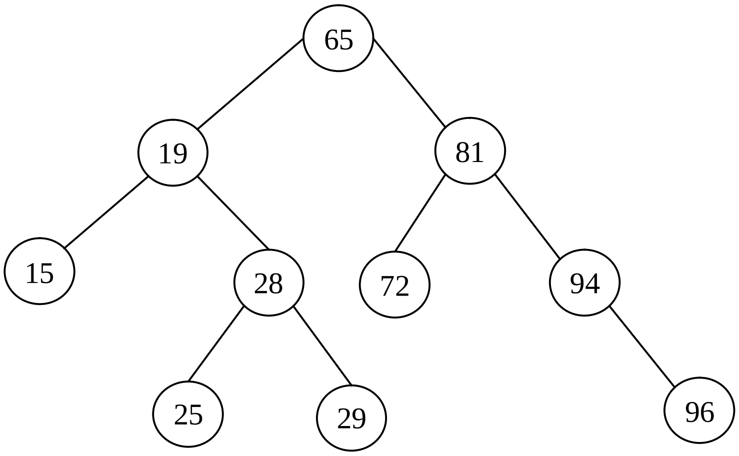
Binary Search Tree with numeric data

Operations on Binary Search Trees:

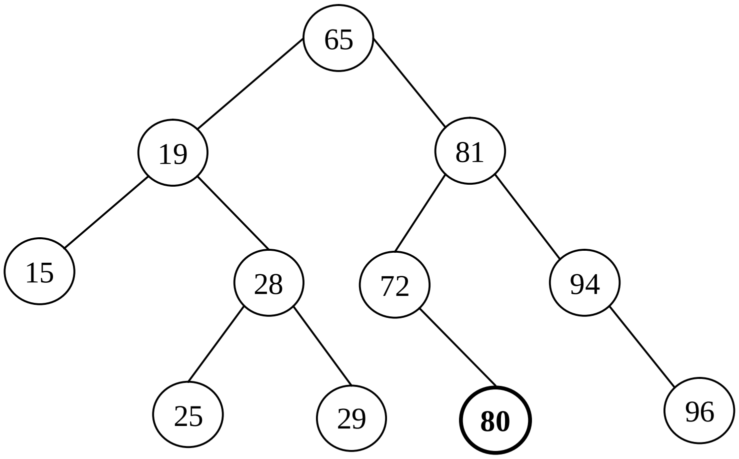
1. Inserting data
2. Deleting Data
3. Traversing the Tree

1. Inserting data into a binary search tree

To insert a node with data say item into a binary search tree, first binary search tree is searched starting from ROOT node for the item. If the item is found, do nothing. Otherwise item is to be inserted as LEAF node where search is halt.



Inserting 80 into the above figure



After insertion of new node 80

Algorithm BST_Insert(item)

Input: *item* is data part of new node to be insert into BST.

Output: BST with new node has data part *item*.

```
1. ptr = Root
2. flag = 0
3. while(ptr != NULL && flag == 0 )
        a) if( item == ptr.data)
                i) flag = 1
                ii) print “item already exist”
        b) else if( item < ptr.data)
                i) ptr1 = ptr
                ii) ptr = ptr.LCHILD
        c) else if( item > ptr.data)
            i) ptr1 = ptr
            ii) ptr = ptr.RCHILD
        d) end if
    4. end loop
    5. if(ptr == NULL)
        a) new = getnewnode()
        b) new.data = item
        c) new.lchild = NULL
        d) new.rchild = NULL
        e) if(root.data == NULL)
            i) root = new
                A) print "New node inserted successfully  as ROOT Node"
        f) else if( item < ptr1.data)        /* inserting new node as left child to its parent*/
            i) ptr1.lchild = new
            ii) print "New Node is inserted successfully as LEFT child"
        g) else                              /* inserting new node as right child to its parent*/
            i) ptr1.rchild = new;
            ii) print "New Node is inserted successfully as Right Child"
        h) end if
    6. end if
End BST_Insert
```

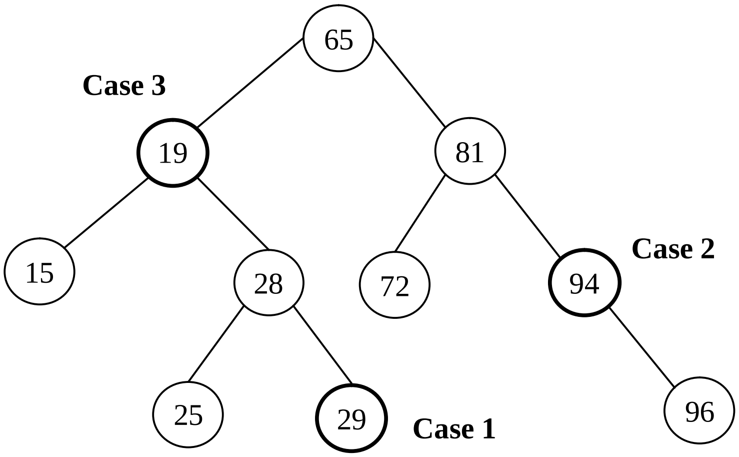
2. Deleting data from a Binary Search Tree

- If ITEM is the information given which is to be deleted from a BST. Let N be the node which contains the information ITEM. Assume PARENT(N) denotes the parent node of N and SUCC(N) denotes the inorder successor of N.
- Then the deletion of the node N depends on the number of its children. Hence, 3 cases may arise and they are:

    Case 1: N is the leaf node. i.e. no child nodes.

    Case 2: N has exactly one child.

    Case 3: N has two children.

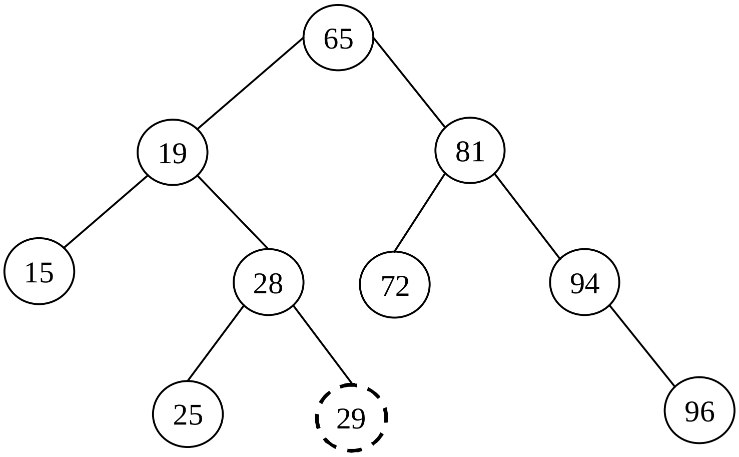


Three cases from deleting a node from BST

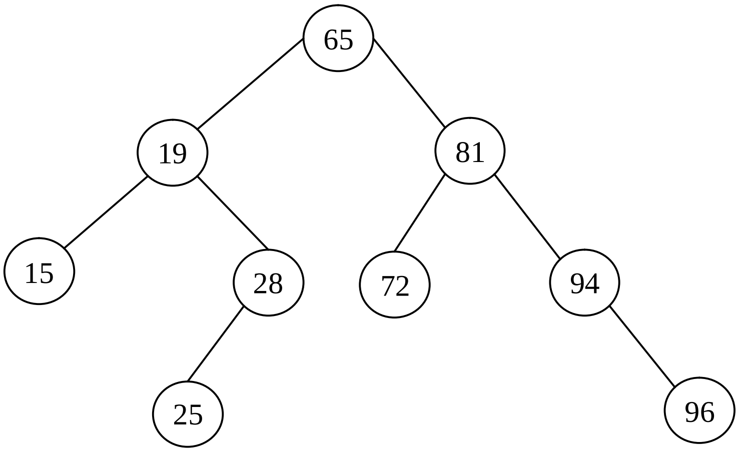
In the above Binary Search Tree (BST) deletion of a node *29* leads to *Case 1*. Here node *29* is a leaf node i.e. which does not have any child nodes. Deletion of node *94* leads to *Case 2*. Here node *94* has only one child node. Deletion of a node *19* leads to *Case 3*. Here node 19 has two child nodes.

Case 1:

N is a leaf node, this node is to be delete. N is deleted from T by simply setting the pointer of N in the parent node PARENT(N) by **NULL** value.



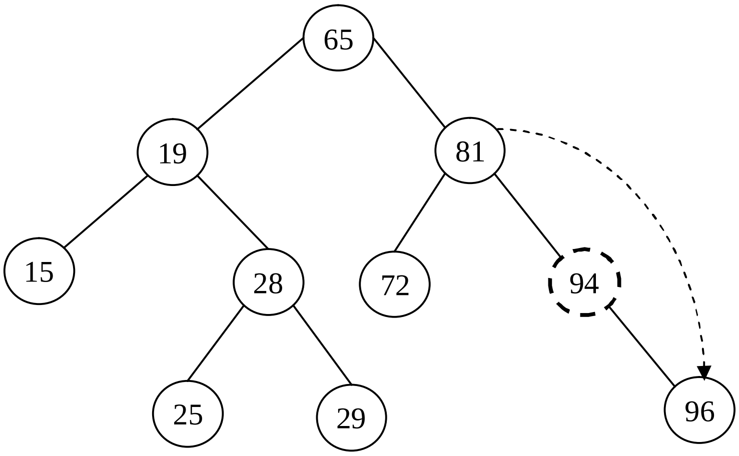
Deletion of node 29



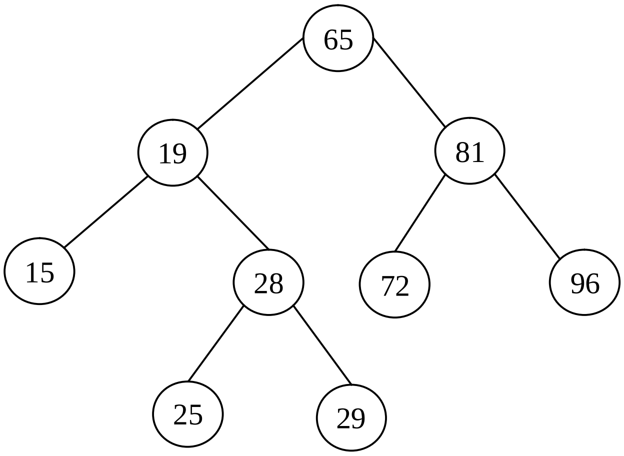
After deletion of node 29 form given BST

Case 2: N has exactly one child.

N is deleted from T by simply replacing the pointer of N in PARENT(N) by the pointer of the only child of N.



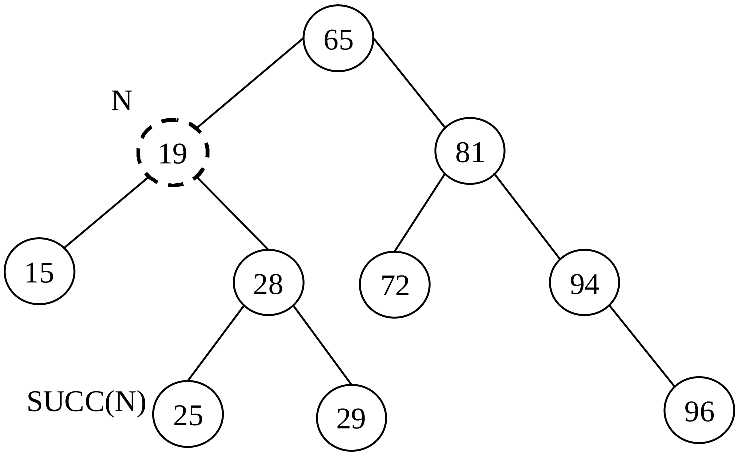
Deletion of node 94



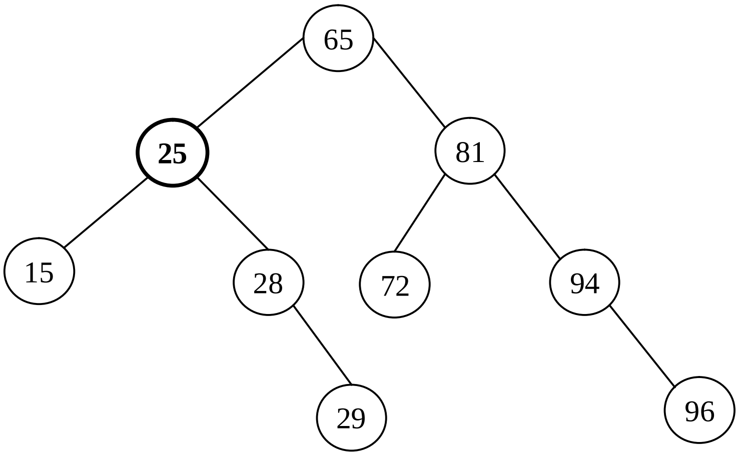
After deletion of node 94 form given BST

Case 3: N has two child nodes.

N is deleted from T by first deleting SUCC(N) from T(by using Case 1 or Case 2 it can be verified that SUCC(N) never has a left child) and then replacing the data content in node N by the data content in node SUCC(N).



Deletion of node 19



After deletion of node 19 form given BST

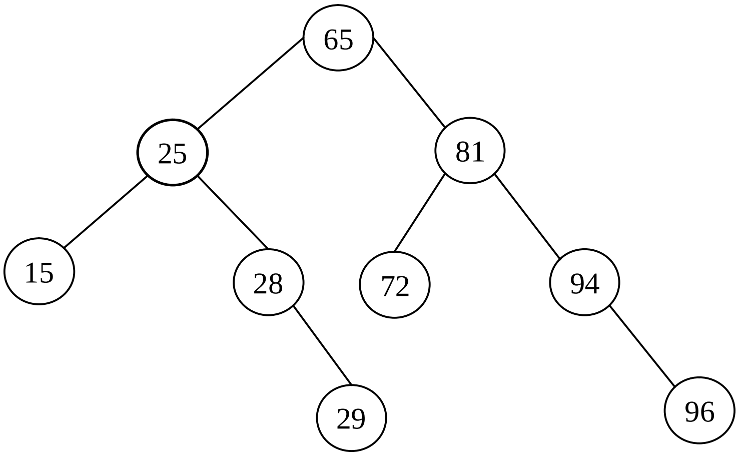
3. Binary Search Tree Traversals

Traversal operation is used to visit each node present in binary search tree exactly once.

A Binary search tree can be traversed in 3 ways.

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

Example:



Inorder traversal for above BST is: 15, 25, 28, 29, 65, 72, 81, 94, 96

(Inorder traversal of BST always gives Ascending order of elements)

Preorder traversal for above BST is: 65, 25, 15, 28, 29, 81, 72, 94, 96

Postorder traversal for above BST is: 15, 29, 28, 25, 72, 96, 94, 81, 65

Recursive Binary Search Tree Traversals

Same as Recursive implementation of Binary Tree Traversals

Algorithm preorder(ptr)

Input: Binary Tree with some nodes.

Output: *preorder* traversal of given Binary Tree.

```
    1. if(ptr != NULL)
        a) print(ptr.data)
        b) preorder(ptr.lchild)
        c) preorder(ptr.rchild)
    2. end if
End preorder
```

Algorithm inorder(ptr)

Input: Binary Tree with some nodes.

Output: *inorder* traversal of given Binary Tree.

```
    1. if(ptr != NULL)
        a) inorder(ptr.lchild)
        b) print(ptr.data)
        c) inorder(ptr.rchild)
    2. end if
End inorder
```

Algorithm postorder(ptr)

Input: Binary Tree with some nodes.

Output: *postorder* traversal of given Binary Tree.

```
    1. if(ptr != NULL)
        a) postorder(ptr.lchild)
        b) postorder(ptr.rchild)
        c) print(ptr.data)
    2. end if
End postorder
```

Creation of Binary Search Tree from its Tree traversals

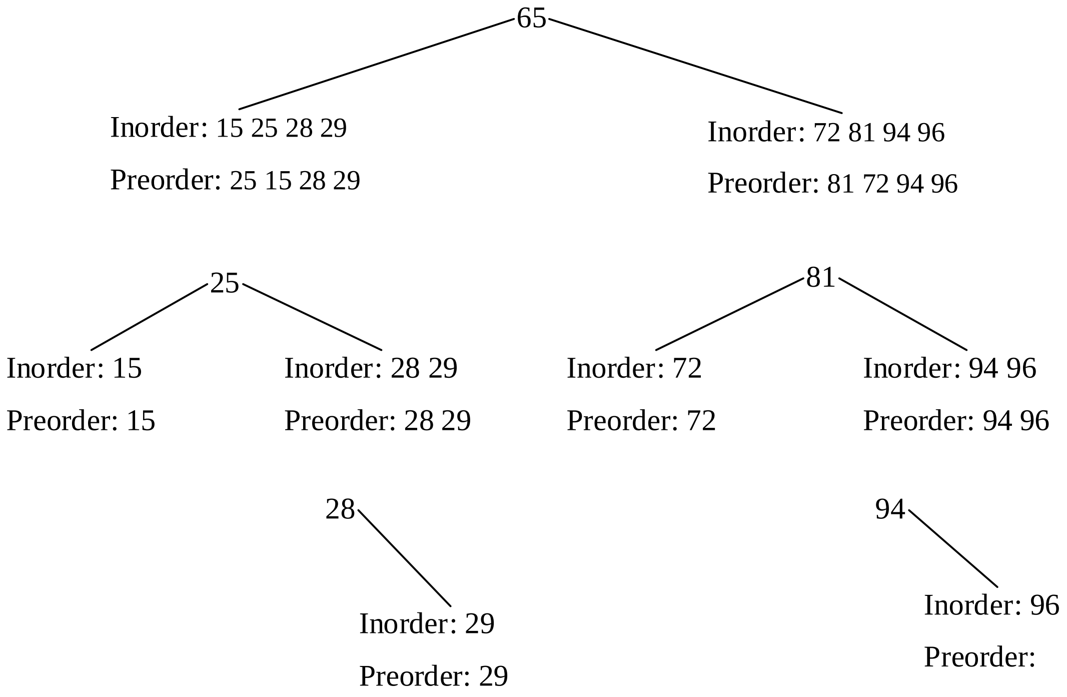
- A binary search tree can be constructed from its traversals.
- If the *Preorder* traversals is given, then the ***first node*** is *ROOT* node and *Postorder* traversal is given then ***last node*** is the *ROOT* node.

- For construction of a binary search tree from its traversals, two traversals are essentials. Out of which one should be *inorder* traversal and another one is either *preorder* (or) *postorder* traversal.

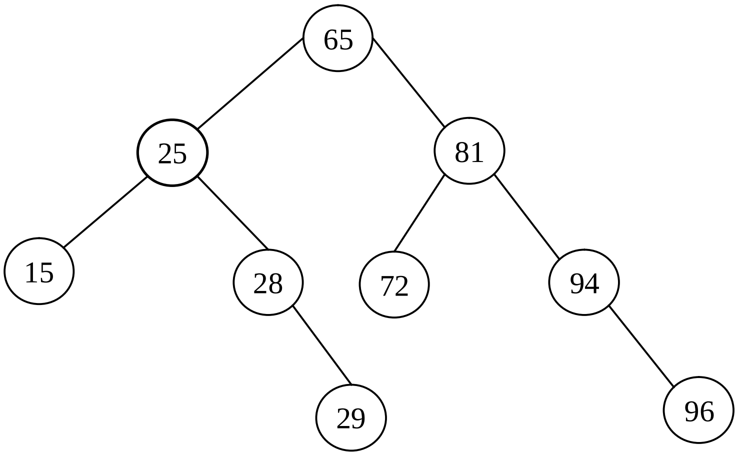
Eg.1:

Inorder:	15	25	28	29	65	72	81	94	96
Preorder:	65	25	15	28	29	81	72	94	96

1. From the preorder traversal, *65* is the *ROOT* node.
2. In the inorder traversal, all the nodes which are LEFT side of **65** belongs to LEFT sub tree and those node which are RIGHT side of **65** belongs to RIGHT sub tree.
3. Now the problem is reduced to two sub trees and same procedure can be applied repeatedly.



Final Binary Search Tree from the Inorder and Preorder as follows:


65
25
81
15
28
72
94
29
96

UNIT-IV
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. How many nodes in a tree have ***no*** ancestors? [ ]

   (A) 0 (B) 1 (C) 2 (D) n

2. What is the maximum possible number of nodes in a binary tree at level ***6***? [ ]

   (A) 6 (B) 12 (C) 64 (D) 32

3. A full binary tree with *2n+1* nodes contain_____________? [ ]

   (A) n leaf nodes (B) n non-leaf nodes

   (C) n-1 leaf nodes (D) n-1 non-leaf nodes

4. A full binary tree with *n* leaves contains___________? [ ]

   (A) n nodes (B) log_2^n nodes (C) 2n –1 nodes (D) 2^n nodes

5. The number of leaf nodes in a complete binary tree of depth d is _______? [ ]

   (A) 2^d (B) 2^{d-1}+1 (C) 2^{d+1}+1 (D) 2^d+1

6. The pre-order and post order traversal of a Binary Tree generates the same output. The tree can have maximum___________. [ ]

   (A) Three nodes (B) Two nodes

   (C) One node (D) Any number of nodes

7. The height of a tree is the length of the longest root-to-leaf path in it. The maximum and minimum number of nodes in a binary tree of height 5 are: ______ [ ]

   (A) 63 and 6, respectively (B) 64 and 5, respectively

   (C) 32 and 6, respectively (D) 31 and 5, respectively

8. If a node in a BST has two children, then its *inorder predecessor* has ______. [ ]

   (A) No left child (B) No right child

   (C) Two children (D) No child

9. In order to get the contents of a Binary search tree in ascending order, one has to traverse it in _______ fashion? [ ]

   (A) pre-order (B) in-order (C) post order (D) Not possible

10. A BST is traversed in the following order recursively: **right, root, left.** The output sequence will be in____________. [ ]

    (A) Ascending order (B) Descending order

    (C) Bitomic sequence (D) No specific order

11. In order to get the information stored in a Binary Search Tree in the descending order, one should traverse it in which of the following order? [ ]

    (A) left, root, right (B) root, left, right

(C) right, root, left (D) right, left, root

12. What is common in three different types of traversals (Inorder, Preorder and Postorder)?

    (A) Root is visited before right subtree

    (B) Left subtree is always visited before right subtree

    (C) Root is visited after left subtree

    (D) All of the above

13. A binary search tree contains the numbers 1, 2, 3, 4, 5, 6, 7, 8. When the tree is traversed in pre-order and the values in each node printed out, the sequence of values obtained is 5, 3, 1, 2, 4, 6, 8, 7. If tree is traversed in post-order, the sequence obtained would be____ [ ]

    (A) 8, 7, 6, 5, 4, 3, 2, 1 (B) 1, 2, 3, 4, 8, 7, 6, 5

    (C) 2, 1, 4, 3, 6, 7, 8, 5 (D) 2, 1, 4, 3, 7, 8, 6, 5

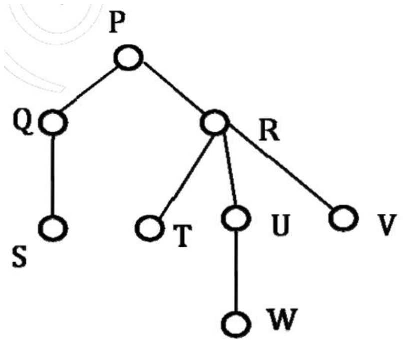
14. Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 55. Which of the following sequences CANNOT be the sequence of nodes examined? [ ]

    (A) {10, 75, 64, 43, 60, 57, 55} (B) {90, 12, 68, 34, 62, 45, 55}

    (C) {9, 85, 47, 68, 43, 57, 55} (D) {79, 14, 72, 56, 16, 53, 55}

15. Consider the following rooted tree with the vertex P labeled as root. The order in which the nodes are visited during in-order traversal is________? [ ]

    (A) SQPTRWUV (B) SQPTURWV (C) SQPTWUVR (D) SQPTRUWV



Explanation:

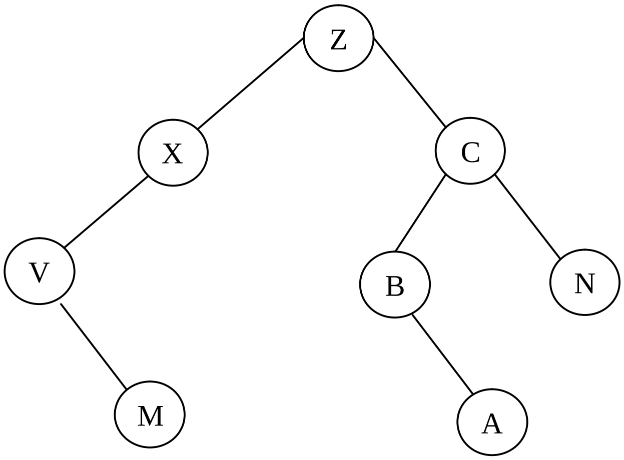
The only confusion in this question is, there are 3 children of R. So when should R appear – after U or after T? There are two possibilities: SQPTRWUV and SQPTWURV. Only 1st possibility is present as an option A, the 2nd possibility is not there. Therefore option A is the right answer.

SECTION-B

Descriptive Questions

1.Write recursive algorithms for Binary Search Tree Traversals.

2.What is the inorder, preorder and postorder for the following binary tree?



3. Construct Binary Tree for the following tree traversals.

   Inorder: W U R O P I T Y E

   Preorder: P O U W R I Y T E

   What is the **Post order** traversal for the above constructed binary tree?

Ans: W R U O T E Y I P

4. Construct Binary Tree for the following tree traversals.

   Inorder: N Z V A M C B S X D

   Postorder: Z A V N C S D X B M

   What is the **Preorder** traversal for the above constructed binary tree?

Ans: M N V Z A B C X S D

5. Create Binary Search with the following elements.

20 30 15 25 42 61 72 18 10 8

What is the **Inorder** traversal for the above constructed Binary Search tree?

Ans: 8 10 15 18 20 25 30 42 61 72

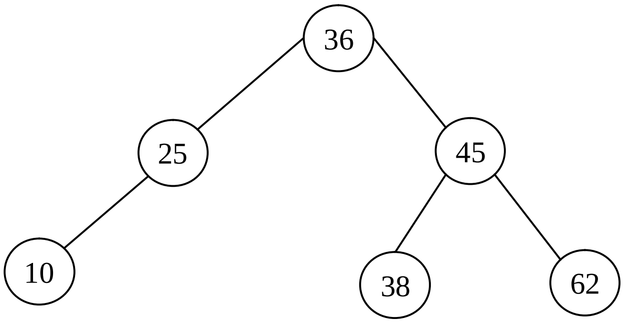
6. Create Binary Search with the following elements.

100 90 110 80 95 125 115 108 104 76 49 62

What is the **Inorder** traversal for the above constructed Binary Search tree?

Ans: 49 62 76 80 90 95 100 104 108 110 115 125

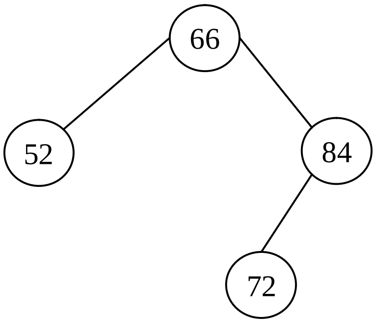
7. Consider the following Binary Search Tree and perform the following sequence of operations.



Insert the elements 55, 68, 49, 18, 28, 27, 30. Now **delete** the elements 55, 45, 36, 10 and 18. Finally what is the root node?

Ans: 38

8. Consider the following Binary Search Tree and perform the following sequence of operations.



Insert the elements 89, 46, 48, 26, 76, 98, 100. Now **delete** the elements 84, 48, 52 and 66. Finally what is the root node?

Ans: 72

Section - C

1. Consider a binary tree T that has 200 leaf nodes. Then, the number of nodes in T that have exactly two children are? **(GATE 2016)** **[ ]**

(A) 201 (B) 100 (C) 199 (D) 50

2. The maximum number of binary trees that can be formed with three unlabelled nodes is: ________ **(GATE 2007)** [ ]

(A) 1 (B) 5 (C) 4 (D) 3

3. The height of a binary tree is the maximum number of edges in any root to leaf path. The maximum number of nodes in a binary tree of height h is: **(GATE 2007)** [ ]

(A) 2^h -1 (B) 2^(h-1) – 1 (C) 2^(h+1) -1 (D) 2*(h+1)

4. The inorder and preorder traversal of a binary tree are ***d b e a f c g*** and ***a b d e c f g***, respectively. The postorder traversal of the binary tree is: **(GATE 2007)** [ ]

(A) d e b f g c a (B) e d b g f c a (C) e d b f g c a (D) d e f g b c a

5. Consider the label sequences obtained by the following pairs of traversals on a labelled binary tree. Which of these pairs identify a tree uniquely? **(GATE CS 2004)** [ ]

   i) preorder and postorder

   ii) inorder and postorder

   iii) preorder and inorder

   iv) level order and postorder

   (A) (i) only (B) (ii), (iii) only (C) (iii) only (D) (iv) only

6. Let **LASTPOST**, **LASTIN** and **LASTPRE** denote the last vertex visited in a postorder, inorder and preorder traversal. Respectively, of a complete binary tree. Which of the following is always true? **(GATE CS 2000)** [ ]

   (A) LASTIN = LASTPOST (B) LASTIN = LASTPRE

   (C) LASTPRE = LASTPOST (D) None of the above

7. While inserting the elements 71, 65, 84, 69, 67, 83 in an empty binary search tree (BST) in the sequence shown, the element in the lowest level is? **(GATE 2015)** **[ ]**

   (A) 65 (B) 67 (C) 69 (D) 83

8. Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty *binary search tree*. The binary search tree uses the usual ordering on natural numbers. What is the in-order traversal sequence of the resultant tree? **(GATE CS 2003)** [ ]

   (A) 7 5 1 0 3 2 4 6 8 9 (B) 0 2 4 3 1 6 5 9 8 7

   (C) 0 1 2 3 4 5 6 7 8 9 (D) 9 8 6 4 2 3 0 1 5 7

9. Which of the following is/are *correct* inorder traversal sequence(s) of binary search tree(s)?

   (GATE 2016) **[ ]**

   I. 3, 5, 7, 8, 15, 19, 25

   II. 5, 8, 9, 12, 10, 15, 25

   III. 2, 7, 10, 8, 14, 16, 20

   IV. 4, 6, 7, 9 18, 20, 25

   (A) I and IV only (B) II and III only (C) II and IV only (D) II only

10. Postorder traversal of a given binary search tree, T produces the following sequence of keys ***10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29.*** Which one of the following sequences of keys can be the result of an in-order traversal of the tree T? **(GATE CS 2004)** [ ]

    (A) 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95

    (B) 9, 10, 15, 22, 40, 50, 60, 95, 23, 25, 27, 29

    (C) 29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95

    (D) 95, 50, 60, 40, 27, 23, 22, 25, 10, 9, 15, 29

11. The following numbers are inserted into an *empty binary search tree* in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)? **(GATE CS 2004)** [ ]

    (A) 2 (B) 3 (C) 4 (D) 6

12. The *preorder* traversal sequence of a *binary search tree* is 30, 20, 10, 15, 25, 23, 39, 35, and 42. Which one of the following is the postorder traversal sequence of the same tree? **(GATE 2013)** [ ]

    (A) 10, 20, 15, 23, 25, 35, 42, 39, 30 (B) 15, 10, 25, 23, 20, 42, 35, 39, 30

    (C) 15, 20, 10, 23, 25, 42, 35, 39, 30 (D) 15, 10, 23, 25, 20, 35, 42, 39, 30

13. Let T be a binary search tree with 15 nodes. The minimum and maximum possible heights of T are: (**GATE-CS-2017 -Set 1) [ ]**

    Note: The height of a tree with a single node is 0.

    (A) 4 and 15 respectively (B) 3 and 14 respectively

    (B) 4 and 14 respectively (D) 3 and 15 respectively

14. Let T be a tree with 10 vertices. The sum of the degrees of all the vertices in T is _____. **(GATE-CS-2017 - Set 1)** [ ]

    (A) 18 (B) 19 (C) 20 (D) 21

Explanation:

Given, v= Total vertices = 10 e = v - 1 =9 Degree = 2 * e = 18 Therefore, option A is correct.

15. The pre-order traversal of a binary search tree is given by 12, 8, 6, 2, 7, 9, 10, 16, 15, 19, 17, 20. Then the post-order traversal of this tree is: **(GATE-CS-2017 -Set 2)** [ ]

    (A) 2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 19, 20

    (B) 2, 7, 6, 10, 9, 8, 15, 17, 20, 19, 16, 12

    (C) 7, 2, 6, 8, 9, 10, 20, 17, 19, 15, 16, 12

    (D) 7, 6, 2, 10, 9, 8, 15, 16, 17, 20, 19, 12

UNIT – V

GRAPHS

Objective:

- To gain knowledge of heap trees and graph data structure.

Syllabus:

Heap Trees and Graphs

Heap Trees: Basic concept, operations, application-heap sort.

Graphs- Basic concept, representations of graphs, graph traversals-breadth first search and depth first search techniques.

Learning Outcomes:

At the end of the unit student will be able to

- differentiate between Min-heap and Max-heap
- construct Binary heap for the given set of keys
- apply Insertion and Deletion operations on Binary Heap
- Sort the given set of keys using Heap sort
- know various graph representation techniques.
- implement graph traversal techniques.

Learning Material

Binary Heap / Heap Tree

- ***Binary Heap/Heap Tree:*** A Heap is a Complete Binary tree with elements from partially ordered set which satisfies Heap Ordering property. The ordering can be of 2 types:

  1. ***Min Heap:*** For each node N in a complete binary tree, the value of N is less than or equal to the value of its children's, such a heap tree is called a Min Heap.

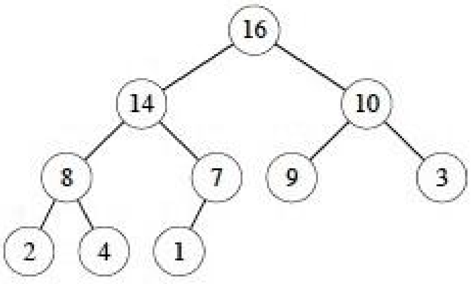


Fig: Min heap

2. ***Max Heap:*** For each node N in a complete binary tree, the value of N is greater than or equal to the value of its children's, such a heap tree is called a Max Heap.

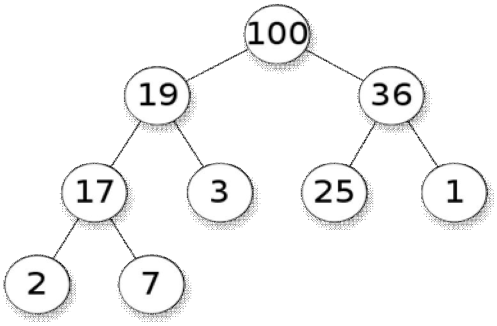
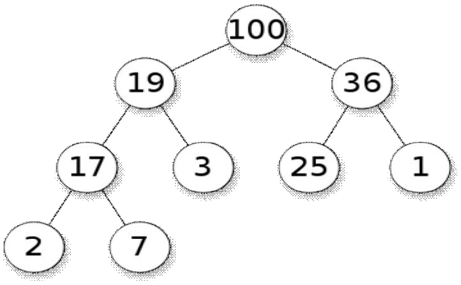


Fig: Max Heap

- ***Properties of Binary Heap:***
    1. *Structure Property:* A Heap is a binary tree that is also a Complete Binary tree.
    2. *Heap Ordering property:*
        - Min Heap property
        - Max Heap property

- **Array Representation of Heap Tree/Binary Heap:**
    - A heap tree can be represented using array and linked list.
    - For an element in array position i, the left child is in the position 2i, the right child is at position (2i+1) i.e in the cell after the left child.
    - The parent is in position floor(i/2).



Array -	**100**	**19**	**36**	**17**	**3**	**25**	**1**	**2**	**7**	**-**	**-**	**-**	**-**	**-**	**-**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**8**	**10**	**11**	**12**	**13**	**14**	**15**

Heap ADT:

1. Inserting a new element
2. Deleting an element

INSERTION OPERATION ON BINARY HEAP:

- To insert an element say x, into the heap with n elements, we first create a hole in position (n+1) and see if the heap property is violated by putting x into the hole.

- If the heap property is not violated, then we have found the correct position for x. Otherwise, we “Reheap –up”' or “percolate-up”' x until the heap property is restored.
- *Percolate-up / Reheap-up:* we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. We continue this process until x can be placed in the whole.
- *Algorithm:*

```
Algorithm Insert_Maxheap( A[1…N], N, X )
{
        N=N+1;
        A[N]=X;
        Reheap_up(N);  /* rebuild heap tree if the heap ordering property is violated */
}
Algorithm Reheap_up(node M)
{
        while(M>1)
        {
                parent=M/2;
                if(A[parent] > A[M])        /*  if(A[parent < A[M]) in case of Min-heap */
                {
                        temp=A[M];
                        A[M]=A[parent];
                        A[parent]=temp;
                        M=parent;
                }
        }
}
```

Example:

- In the below example, 4 is inserted at the last position, which satisfies complete binary tree property.
- Now assume newly inserted node as current node and x is the parent of the current node.
- Now compare current node value with its parent(x). If current node is less than parent(x) node value, then interchange the current node value and x value.
- Continue the re-heap up to the current node is less than the child node value.

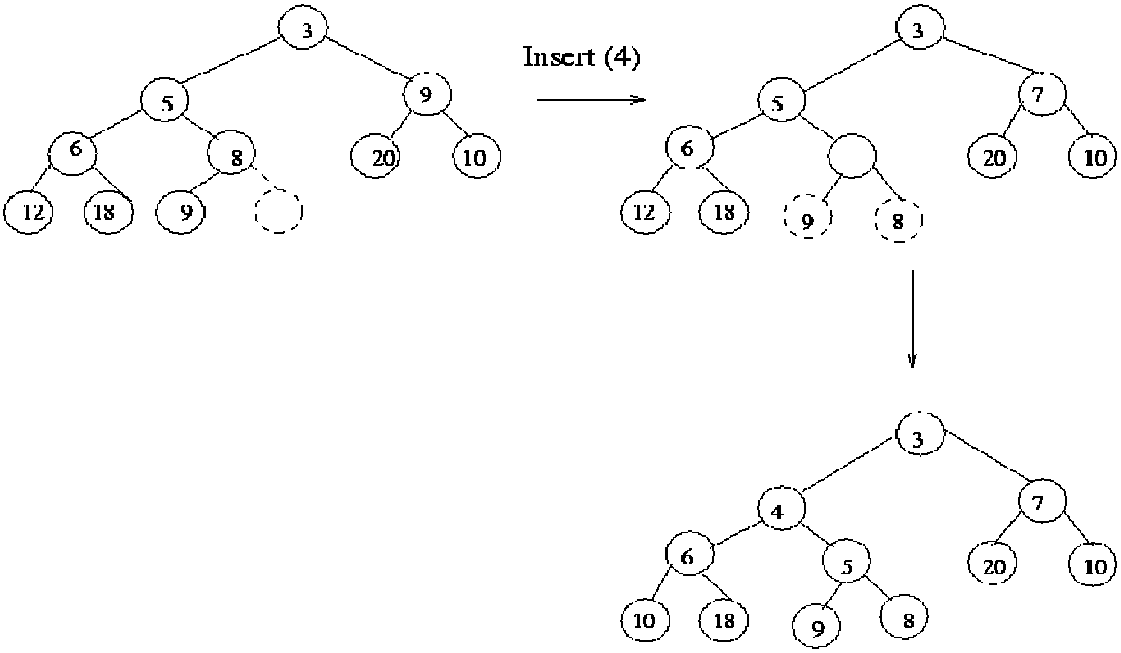


Fig: Insertion of node **4** into Heap tree

- Worst-case complexity of insert is O (h) where h is the height of the heap.
- Thus insertions are O (log n) where n is the number of elements in the heap.

DELETION OPERATION ON BINARY HEAP:

- When the minimum is deleted, a hole is created at the root level. Since the heap now has one less element and the heap is a complete binary tree, the element in the least position is to be relocated.
- First place the last element in the hole created at the root.
- This will leave the heap property possibly violated at the root level. We now “Reheap-down” or ``percolate-down'' the hole at the root until the violation of heap property is stopped.
- *Percolate-down/Reheap-down:* we slide the smaller of the hole’s children into the hole, thus pushing the hole down one level.

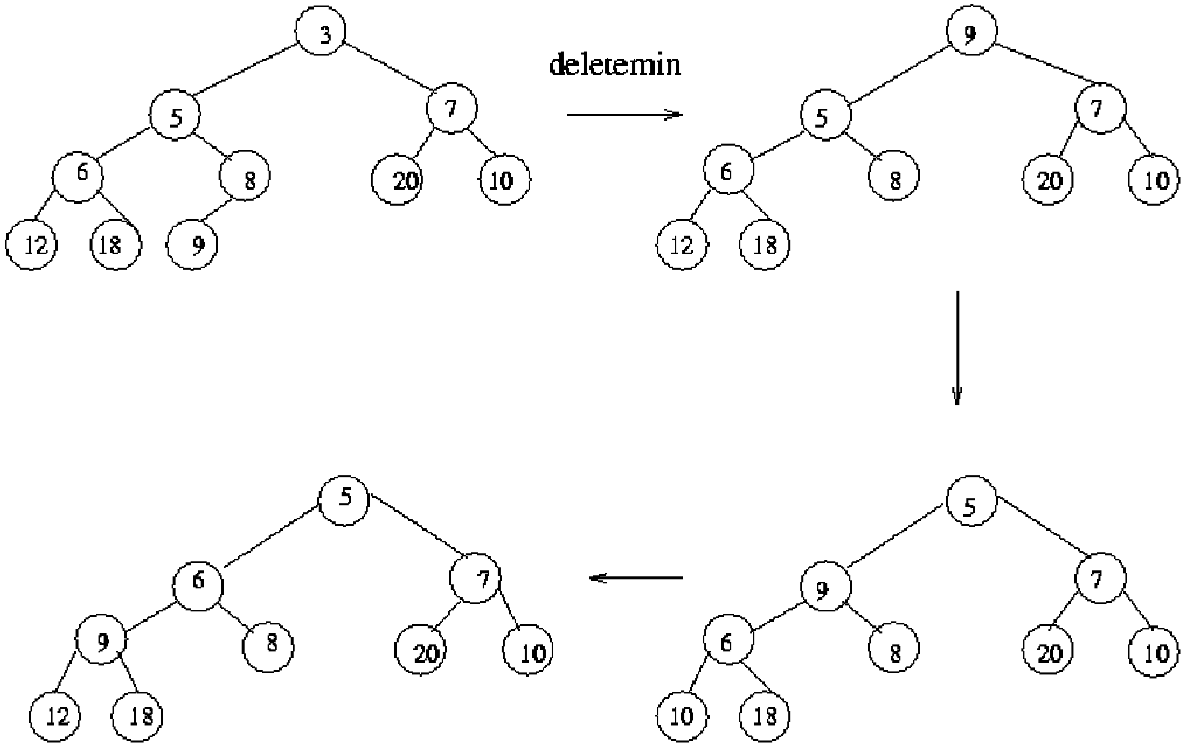
Algorithm:

```
Algorithm DeleteMin( A[1…N], N, X )
{

        if ( N == 0)
                print(“Heap tree is empty, deletion not possible”)
        else
        {
                A[1]=A[N];
                A[N]=0;
                N--;
```

```
            Reheap_down(1);        /* rebuild heap tree if the heap ordering property is violated */
            }
    }
    Algorithm Reheap_down(node M)
    {
            while(2*M <= N)
            {
                    left=2*M;
                    right=2*M+1;
                    if(right <=N and A[right] < A[left])
                            target=right;
                    else
                            target=left;
                    if(A[target] < A[M])      /* if (A[target] > A[M]) in case of  Max-heap */
                    {
                            temp=A[M];
                            A[M]=A[target];
                            A[target]=temp;
                            M=target;
                    }
            }
    }
```

Example:



- ➢ The worst-case time complexity of deletemin/deletemax is O (log n) where n is the number of elements in the heap

Heap Using Linked List

It doesn't make any sense at all to implement a heap as a linked list. Heaps are inherently binary trees. You can store a heap in an array because it's easy to compute the array index of a node's children: the children of the node at position K live at positions 2K and 2K+1. It's massively more efficient to find the Kth element of an array than the Kth element of a linked list.

Advantages of storing a heap as an array rather than a pointer-based binary tree include the following.

- Lower memory usage (no need to store three pointers for every element of the heap).
- Easier memory management (just one object allocated, rather than N).
- Better locality of reference (the items in the heap are relatively close together in memory rather than scattered wherever the allocator put them).

HEAP SORT:

- ➢ It is a Comparison based sorting algorithm.
- ➢ It is an In place sorting method, because it does not require any extra storage space other than the input storage list.
- ➢ Basic steps in the Heap sort are:
  1. *Create max-heap:* Create a max-heap with n elements stored in the array A

  for i = n down to 2 do

  2. *Remove max:* select the value in the root node and swap it with the value at the ith location in A
  3. *Rebuild heap:* rebuild the heap tree for elements A[1,2,……., i-1]

Algorithm:

Input: An array A[1,2,…..n] where n is the number of elements to be sorted
Output: n elements are arranged in A in ascending order

```
Algorithm Heapsort(A, n)
{
    CreateMaxHeap(A, n)
    for i=n down to 2
    {
        temp=A[1]            //swap(A[1], A[n])
        A[1]=A[n]
        A[n]=temp
        Reheap_down(1,i-1)
    }
}
Algorithm CreateMaxHeap(A, n)
{
    for i=n/2 down to 1
```

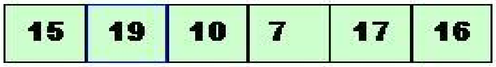
```
                    Reheap_down(i, n)
}
Algorithm Reheap_down(M, N)
{
        while(2*M <= N)
        {
                left=2*M;
                right=2*M+1;
                if(right <=N and A[right] < A[left])
                        target=right;
                else
                        target=left;
                if(A[target] < A[M])
                {
                        temp=A[M];
                        A[M]=A[target];
                        A[target]=temp;
                        M=target;
                }
        }
}
```

Example:

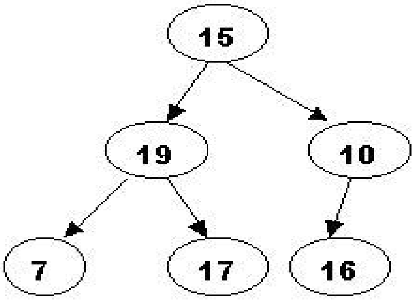
Given an array of 6 elements: 15, 19, 10, 7, 17, 16, sort it in ascending order using heap sort.

A. Building the heap tree

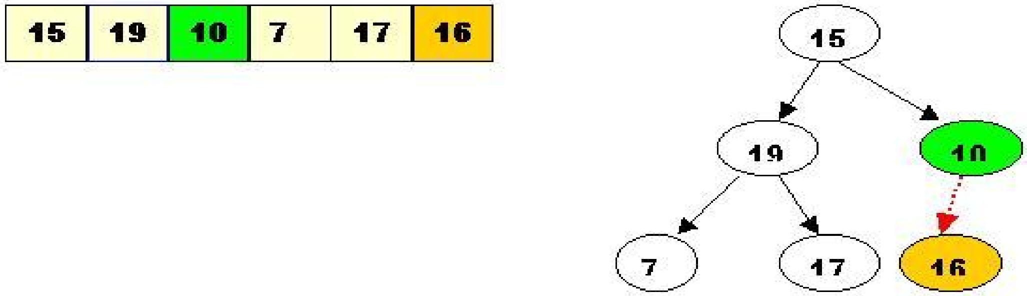
The array represented as a tree, complete but not ordered:



15	19	10	7	17	16

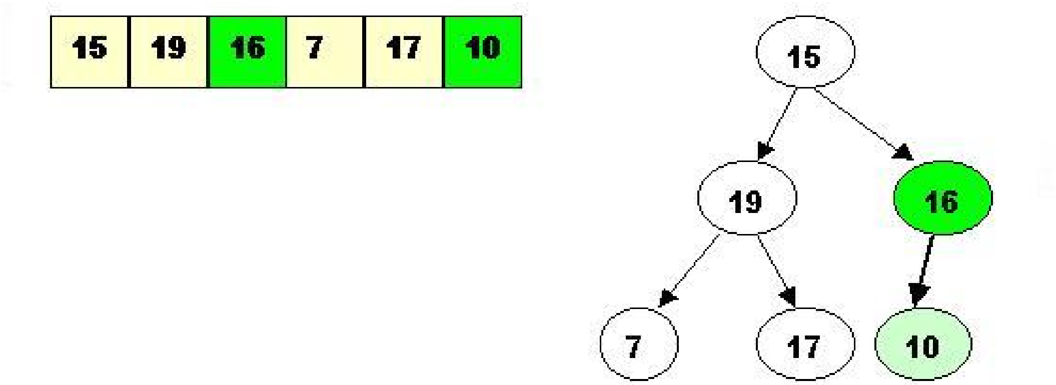


Start with the rightmost node at height 1 - the node at position 3 = Size/2.
It has one greater child and has to be percolated down:



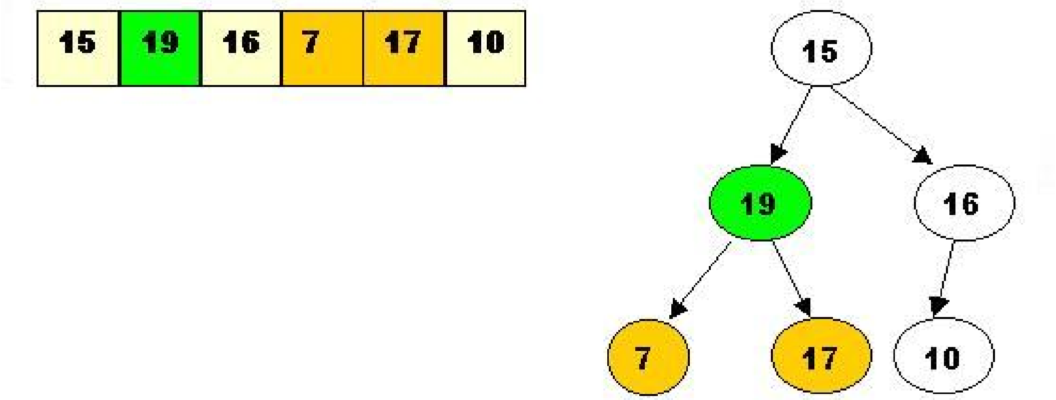
15	19	10	7	17	16

After processing array[3] the situation is:



15	19	16	7	17	10

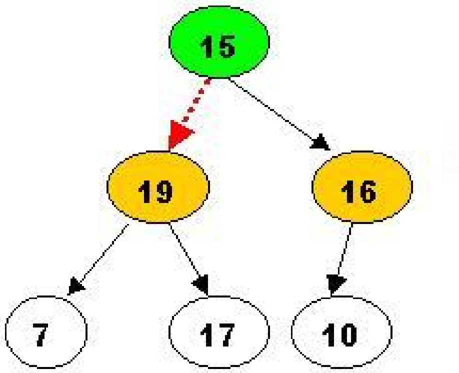
Next comes array[2]. Its children are smaller, so no percolation is needed.



15	19	16	7	17	10

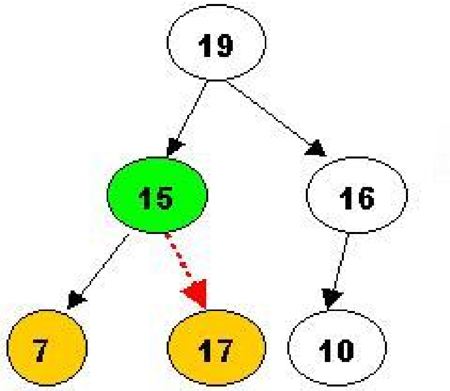
The last node to be processed is array[1]. Its left child is the greater of the children. The item at array[1] has to be percolated down to the left, swapped with array[2].

15	19	16	7	17	10



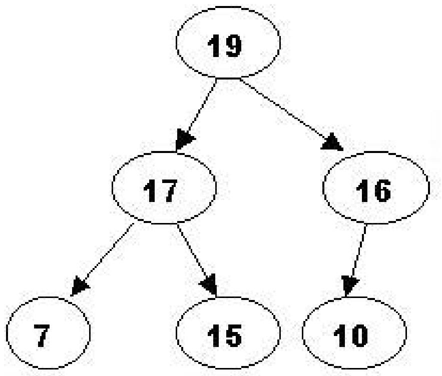
As a result the situation is:

19	15	16	7	17	10



The children of array[2] are greater, and item 15 has to be moved down further, swapped with array[5].

19	17	16	7	15	10

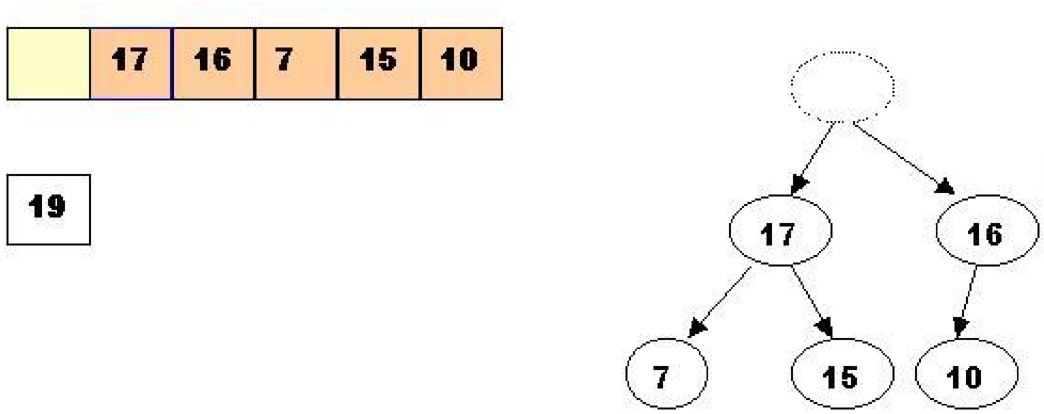


Now the tree is ordered, and the binary heap is built.

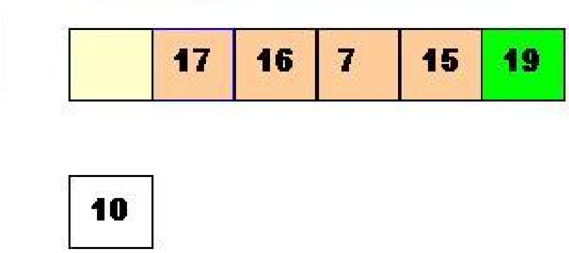
B. Sorting - performing deleteMax operations:

1. Delete the top element 19.

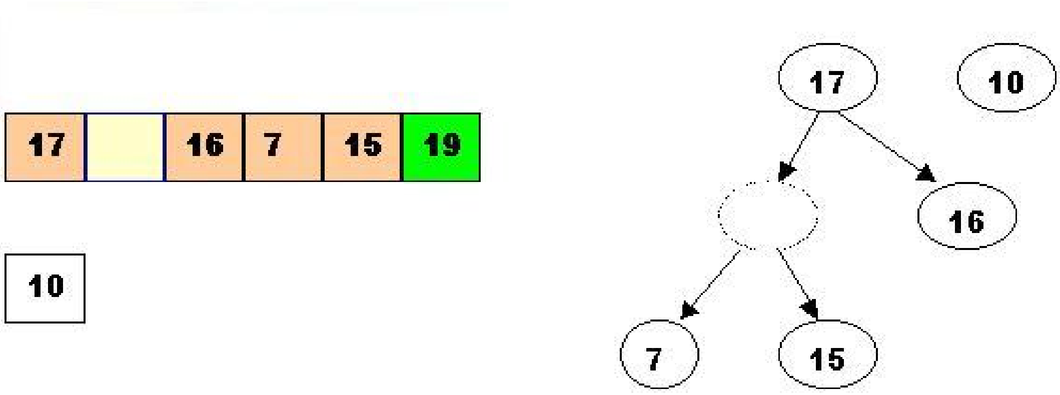
1.1. Store 19 in a temporary place. A hole is created at the top



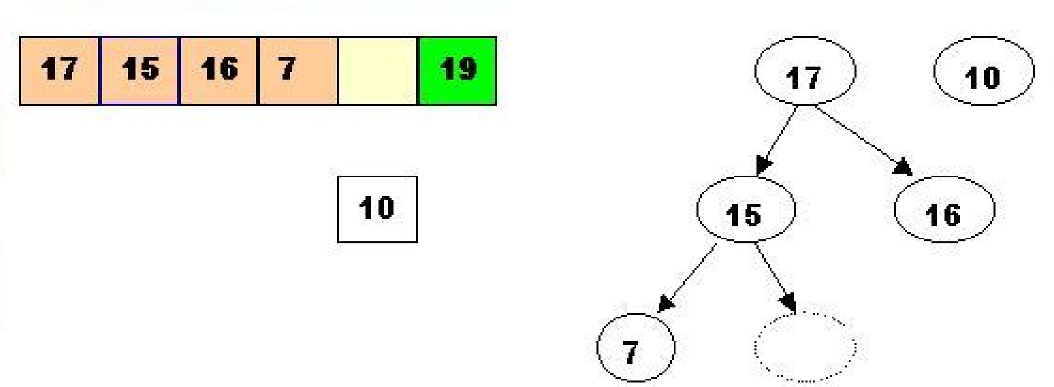
1.2. Swap 19 with the last element of the heap.
As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.
Instead it becomes a cell from the sorted array



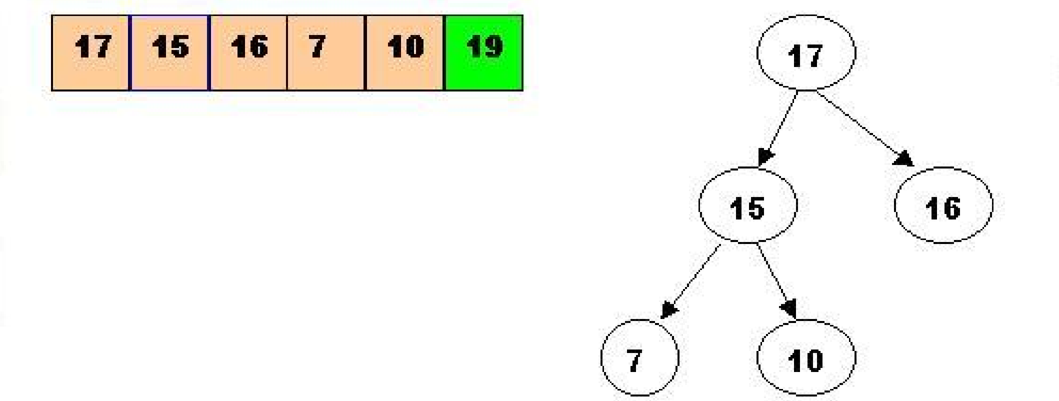
1.3. Percolate down the hole



1.4. Percolate once more (10 is less that 15, so it cannot be inserted in the previous hole)

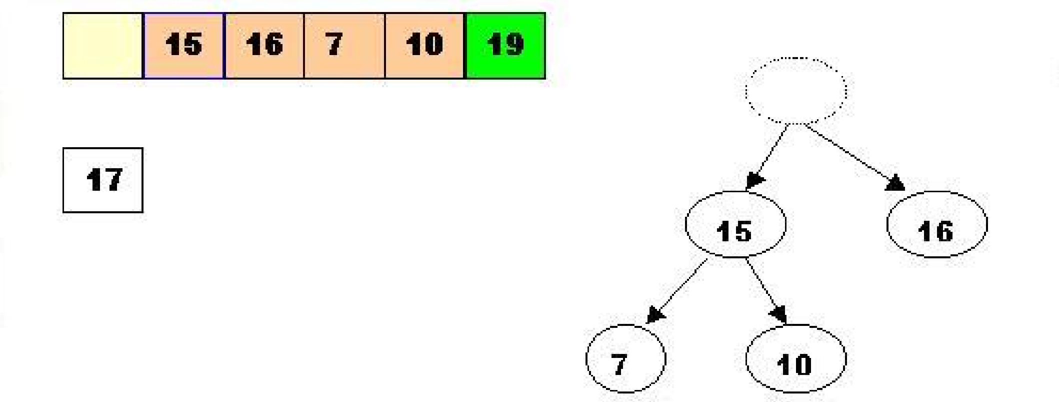


Now 10 can be inserted in the hole



2. DeleteMax the top element 17

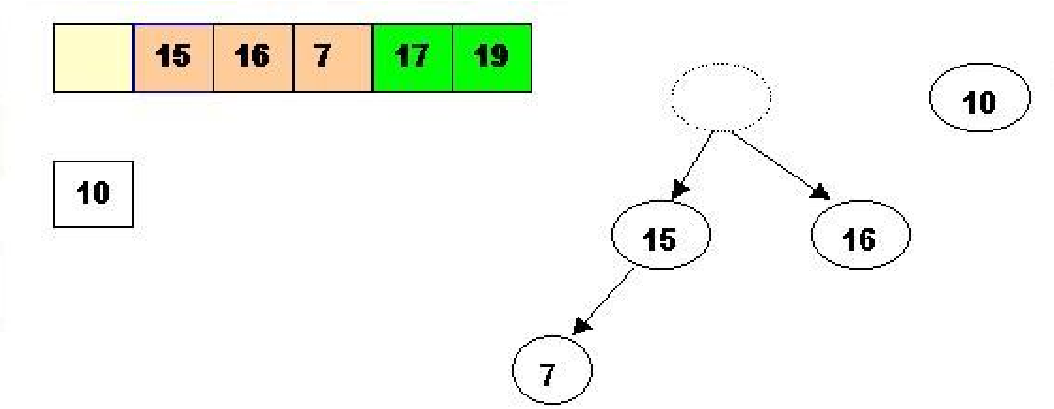
2.1. Store 17 in a temporary place. A hole is created at the top



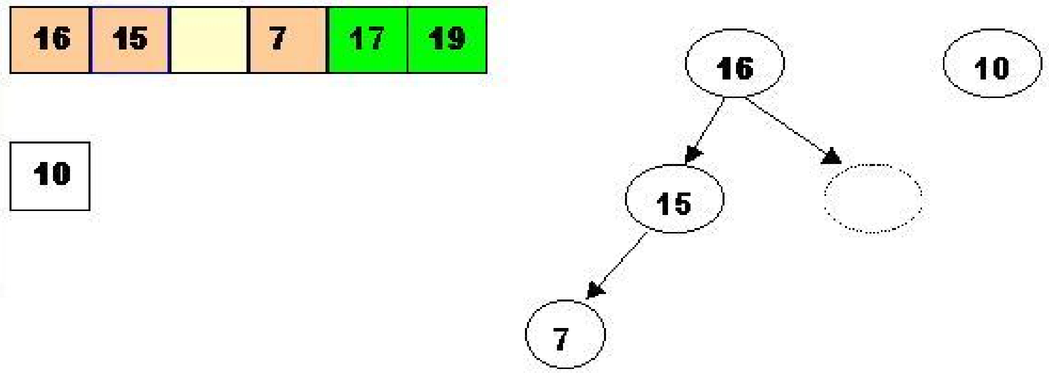
2.2. Swap 17 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.

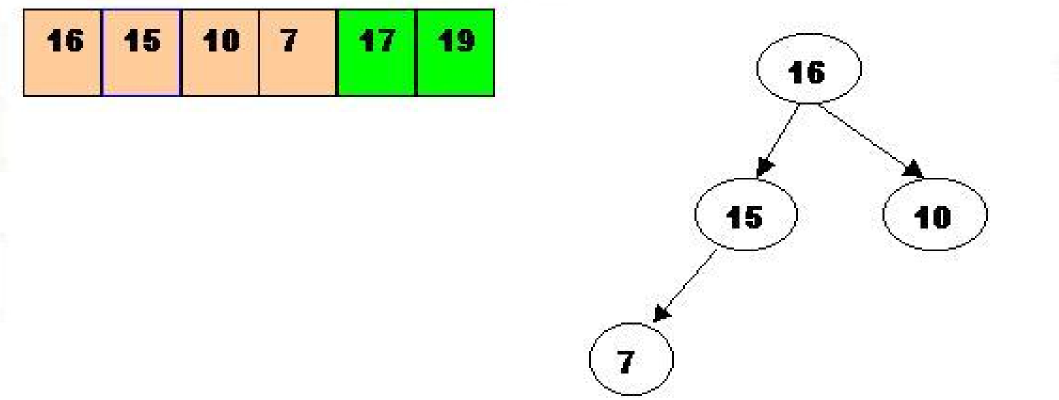
Instead it becomes a cell from the sorted array



2.3. The element 10 is less than the children of the hole, and we percolate the hole down:

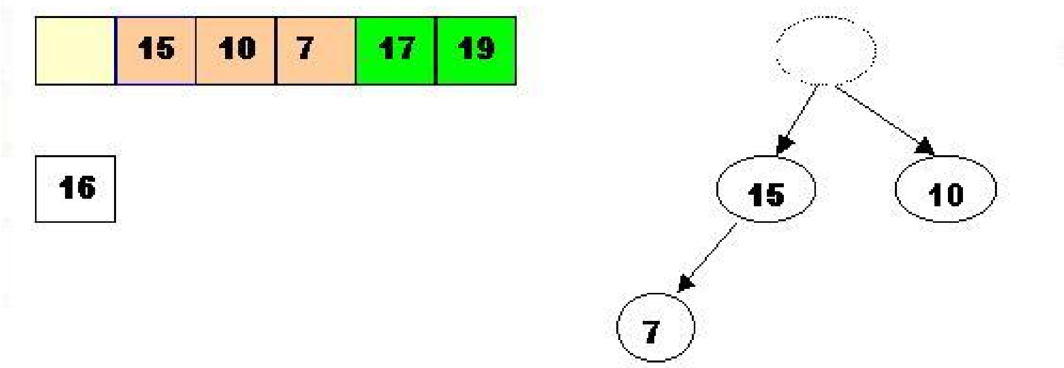


2.4. Insert 10 in the hole

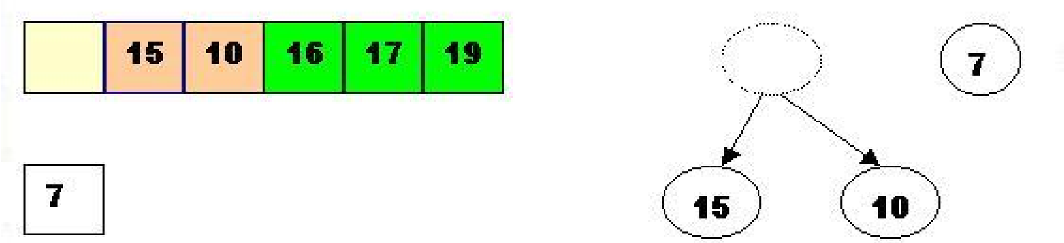


3. DeleteMax 16

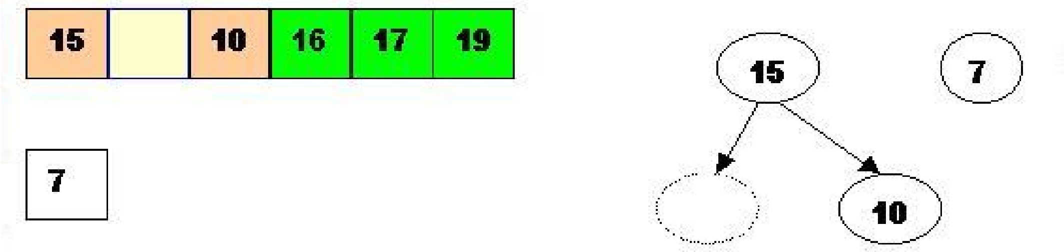
3.1. Store 16 in a temporary place. A hole is created at the top



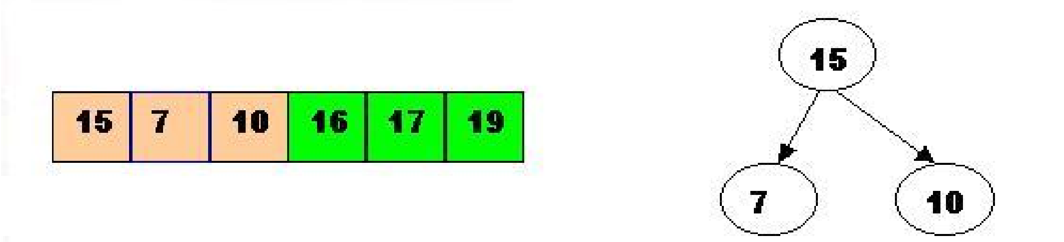
3.2. Swap 16 with the last element of the heap.
As 7 will be adjusted in the heap, its cell will no longer be a part of the heap.
Instead it becomes a cell from the sorted array



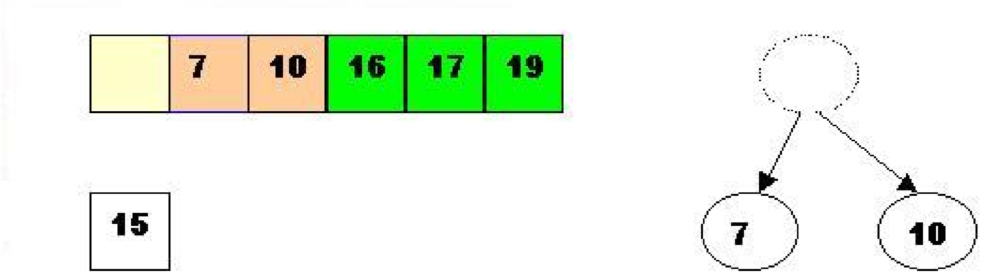
3.3. Percolate the hole down (7 cannot be inserted there - it is less than the children of the hole)



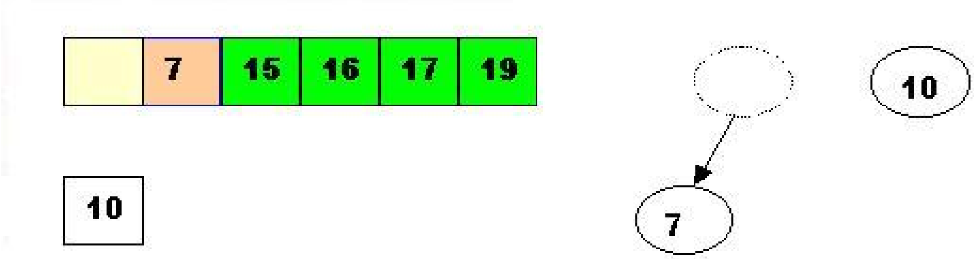
3.4. Insert 7 in the hole



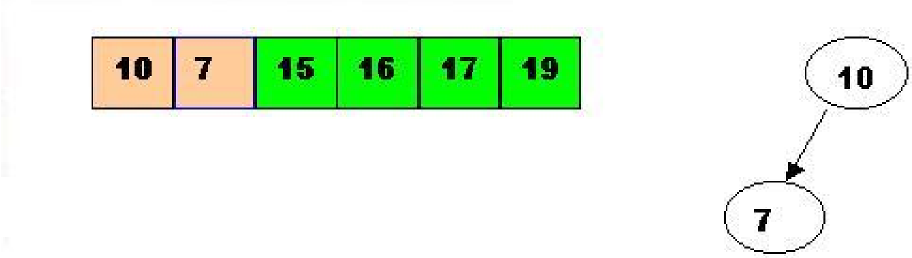
4. DeleteMax the top element 15
4.1. Store 15 in a temporary location. A hole is created.



4.2. Swap 15 with the last element of the heap.
As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.
Instead it becomes a position from the sorted array

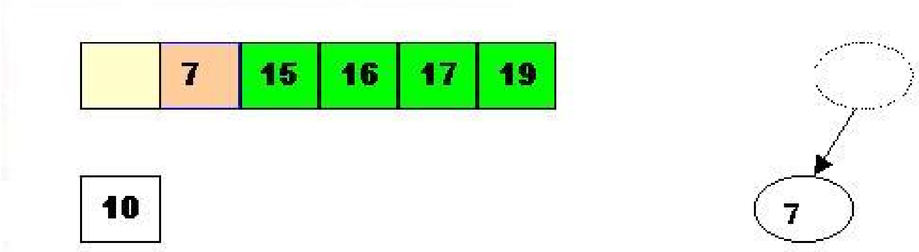


4.3. Store 10 in the hole (10 is greater than the children of the hole)

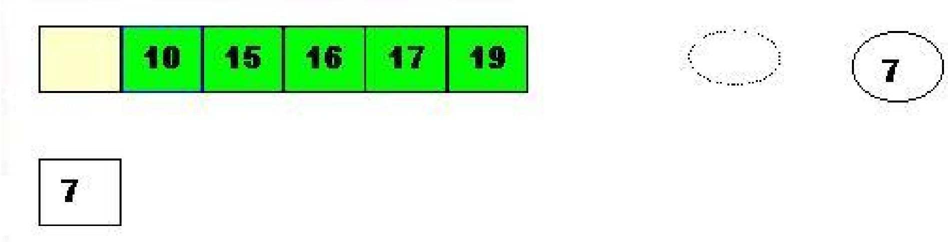


5. DeleteMax the top element 10.

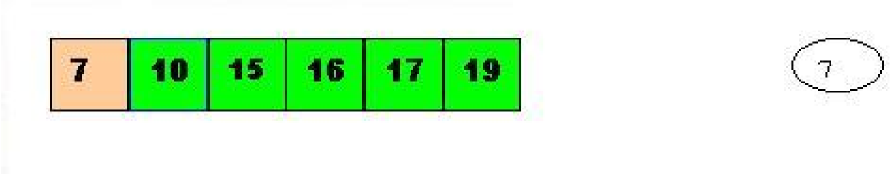
5.1. Remove 10 from the heap and store it into a temporary location.



5.2. Swap 10 with the last element of the heap.
As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



5.3. Store 7 in the hole (as the only remaining element in the heap



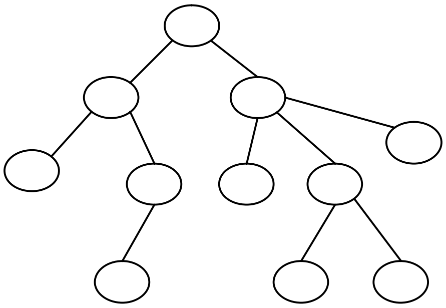
7 is the last element from the heap, so now the array is sorted

- The Time complexity of Heap sort is O(nlogn), where n is the number of elements in the heap

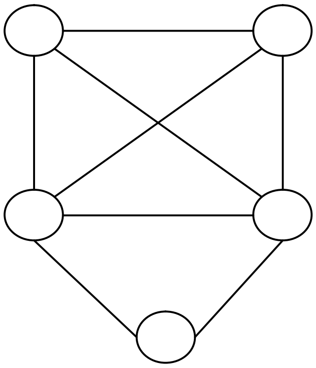
Graphs

Basic Concept:

- **Graph** is another important non-linear data structure.
- The tree structure is a special kind of graph structure.
- In tree structure there is a hierarchical relationship between parent and children, i.e. one parent and many children.
- In the graph relationship is from many parents to many children.



TREE



GRAPH

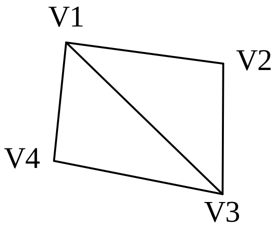
- **Graph Terminology**

  Graph: A graph consists of two sets.

  (i) A set V(G) called as set of all vertices.

  (ii) A set E(G) called as set of all edges (arcs). This set of edges is pair of elements from V(G).

  Eg:

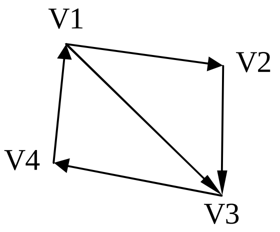


For the above graph V(G)={V1, V2, V3, V4}

E(G) = {(V1,V2), (V2,V3), (V1,V3), (V1,V4), (V3,V4)}

- **Digraph:** A digraph is also called as directed graph. It is a graph G, such that G=<V,E>, where V is set of all the vertices and E is set of ordered pair of elements from V.

  Eg:

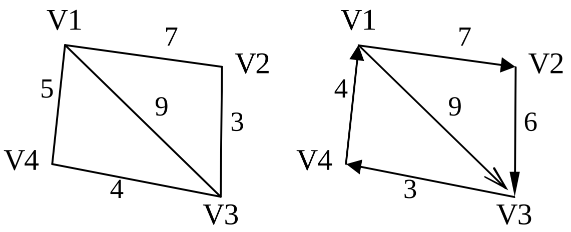


V(G)={V1, V2, V3, V4}

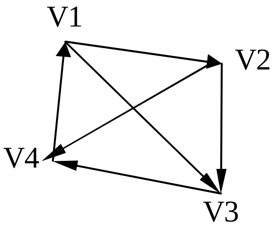
E(G) = {(V1,V2), (V2,V3), (V3,V4), (V4,V1), (V1,V3)}

- **Weighted Graph:** A graph (or digraph) is termed as weighted graph, if all the edges in the graph are labeled with some weights.

  Eg:



- **Adjacent Vertex:** A vertex V_i is adjacent (neighbor of) of another vertex say V_j, if there is an edge from V_i to V_j.



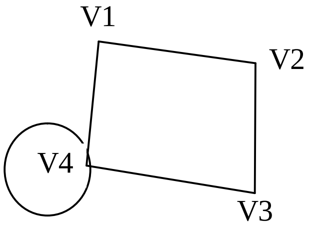
  Eg:

  V2 is adjacent to V3 and V4.

  V1 is not adjacent to V4.

- **Self-loop:** if there is an edge whose starting and ending vertices are same, i.e. (Vi, Vi), then that edge is called as self-loop (loop).

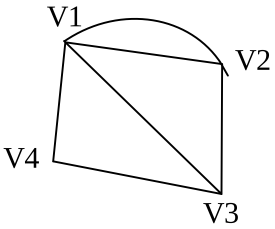
  Eg:



In the above graph Vertex V4 has self-loop.

- **Parallel edges:** if there is more than one edge between the same pair of vertices, then they are known as parallel edges.

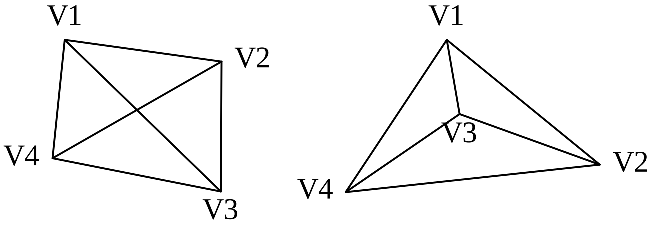
  Eg:



In the above graph two parallel edges between vertices V1 and V2.

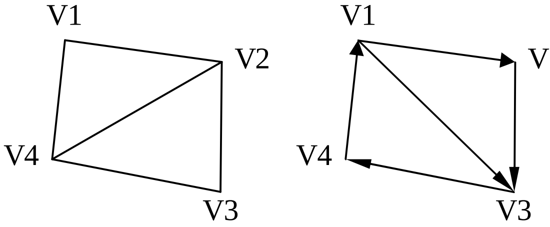
- **Multi graph:** A graph which has either self-looped (or) parallel edges (or) both, that type of graph is called as multi graph.
- **Simple Graph (digraph):** A graph (digraph), if it doesn't have any parallel edges or self-loops, such types of graphs are called as Simple Graph (digraph).
- **Complete Graph:** A graph G is said to be complete graph, if there are edges from any vertex to all other vertices present in Graph.

  Eg:

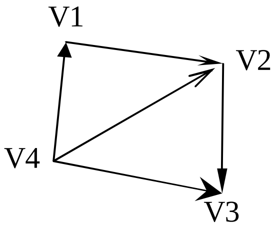


For **n** number of vertices present in complete graph, the total no.of edges are $\frac{n(n-1)}{2}$.

- **Cycle:** If there is a path containing one or more edges, which start from vertex V_i and terminates with same vertex V_i, then that path is known as Cyclic path (or) cycle.

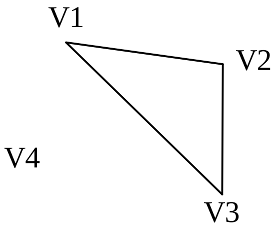


- **Cyclic Graph**: A graph (digraph) that have cycle(s) is called Cyclic Graph (digraph).
- **Acyclic Graph:** A graph (digraph) that does not have cycle(s) is called Acyclic Graph (digraph).



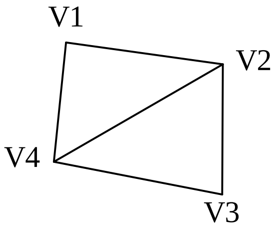
- **Isolated Vertex:** In a graph, a vertex is isolated, if there is no edge is connected from any vertex to the other vertex.

  Eg:



In the above graph V4 is a isolated vertex.

- **Degree of Vertex:** The no.of edges are connected to a vertex is called as degree of a vertex and is denoted by degree(V_i).

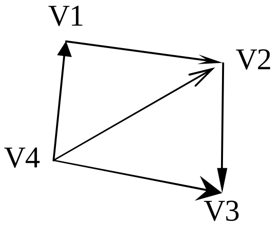


degree(V1) = 2

degree(V2) = 3

- For **digraph**, there are two edges. i.e. indegree and outdegree.
- Indegree of V_i denoted as indegree(V_i) = no.of edges coming towards vertex V_i.
- Outdegree of Vi denoted as outdegree(V_i) = no.of edges coming out from vertex V_i.

Eg:

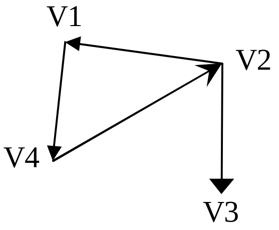


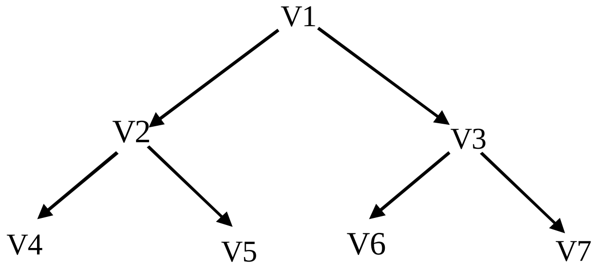
Indegree(V1) = 1 Indegree(V2) = 2

Indegree(V4) = 0 Outdegree(V4) = 3

- **Pendent Vertex:** A vertex V_i is pendent, if its indegree(V_i) = 1and Outdegree(V_i) = 0.

Eg:



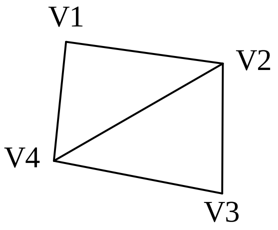


Here V3 is Pendent Vertex. Here V4, V5, V6 and V7 are Pendent Vertices.

- **Connected Graph:** In a graph (not digraph) G, two vertices V_i and V_j are said to be connected, if there is a path in G from V_i to V_j (or) V_j to V_i.

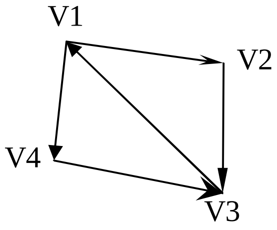
A Graph G is said to be the connected, if for every pair of distinct vertices V_i, V_j in graph, there is a Path.

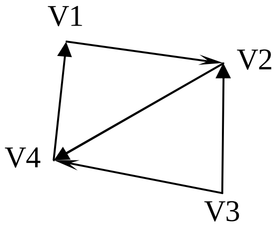
Eg:



A **digraph** with the above property is called as **Strongly Connected graph**.i.e. a digraph G is said to be Strongly Connected, if for every pair of distinct vertices V_i, V_j in G, there is a Direct path from V_i to V_j and also fromV_j to V_i.

Eg:



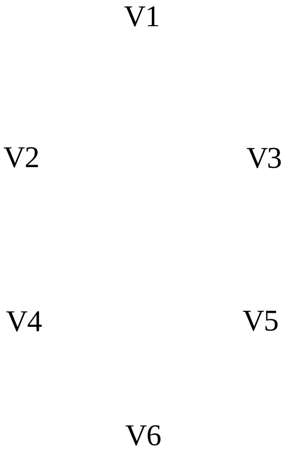


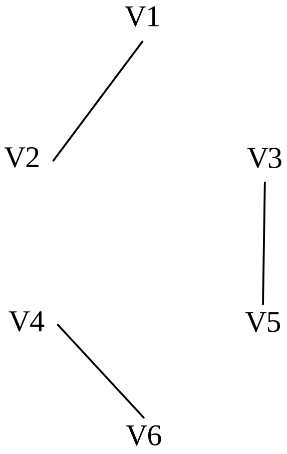
Strongly Connected Graph **Not Strongly Connected Graph**

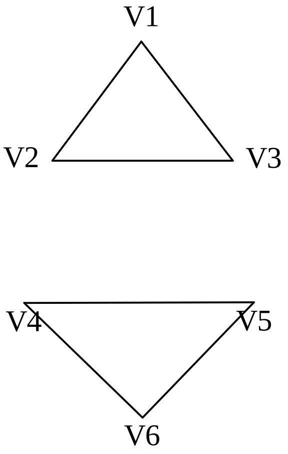
- **Regular Graph:** In a graph, where every vertex has same degree, such type of graph is called as Regular Graph.

A Regular Graph with vertices of degree K is called as **K-Regular Graph**.

Eg:







0- Regular Graph **1- Regular Graph** **2- Regular Graph**

Representation of a Graph

A graph can be represented in the following ways.

1. Set Representation
2. Linked List Representation (or) Adjacency List Representation
3. Matrix (or) Adjacency Matrix Representation

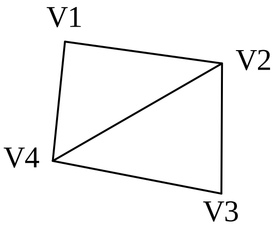
1. Set Representation:

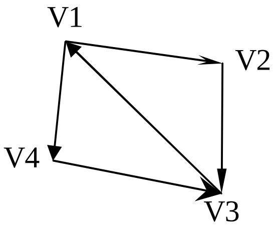
- This is straight forward method of representing graph.
- In this method, 2 sets are maintained V(G) and E(G).

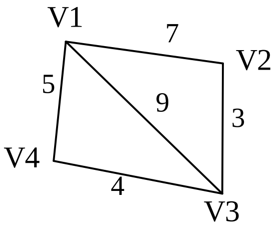
V(G) is set of Vertices.

E(G) is set of Edges.

Eg: G1 G2 G3







V(G1)={V1, V2, V3, V4}

E(G1) = {(V1,V2), (V1,V4), (V2,V4), (V2,V3), (V3,V4)}

V(G2)={V1, V2, V3, V4}

E(G2)={(V1,V2), (V1,V4), (V2,V3), (V3,V1), (V4,V3)}

For representation of weighted graphs, the edge set consist of 3 tuples. i.e. E= W × V × V, where W is the set of edge weights.

V(G3)={V1, V2, V3, V4}

E(G3)={(7,V1,V2), (5,V1,V4), (9,V1,V3), (3,V2,V3), (4,V4,V3)}

* Multi graphs (undirected) can’t be able to represent with the help of Set representation.

2. Linked List Representation:

Linked List Representation is another space saving way of graph representation.

In this graph representation, two types of node structures are assumed.

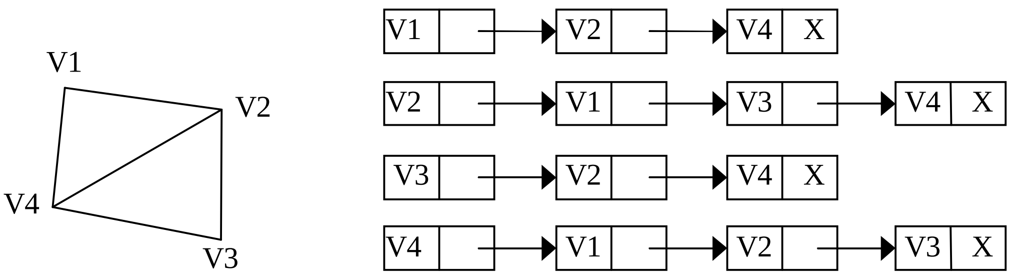
Node_Label	Adj_List

Weight	Node_Label	Adj_List

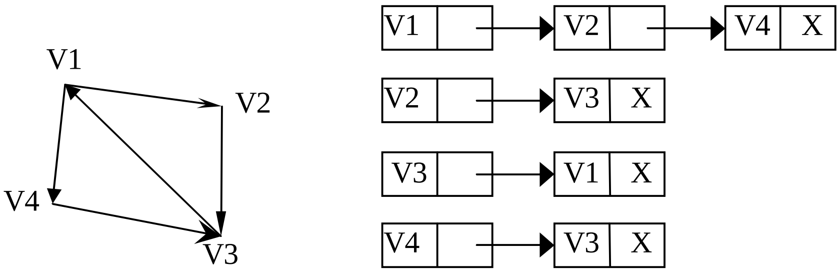
Node Structure for Un-weighted Graph

Node Structure for weighted Graph

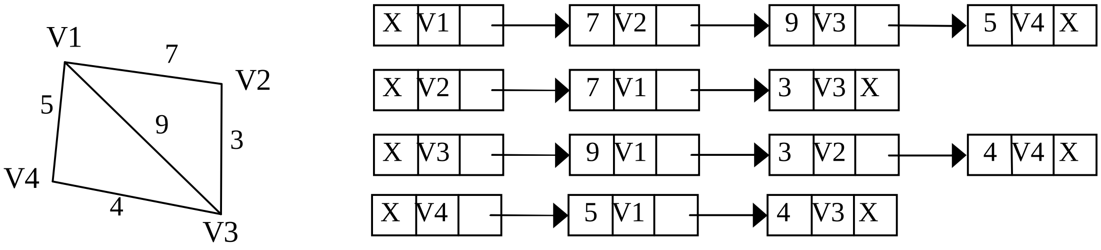
For Graph G1:



For Graph G2:



For Graph G3:

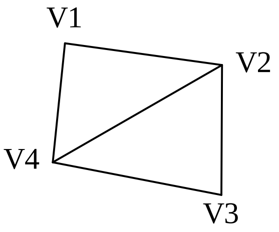


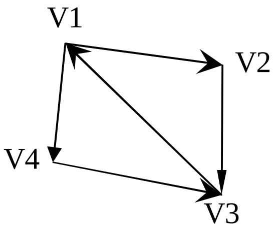
3. Adjacency Matrix Representation:

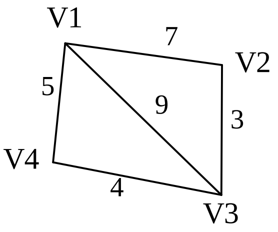
This representation uses a square matrix of order n × n, where n is no.of vertices in graph.

Adjacency Matrix is also termed as Bit matrix (or) Boolean Matrix as the entries are 0 (or) 1.

Eg: G1 G2 G3







$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 1 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 7 & 9 & 5 \\ 2 & 7 & 0 & 3 & 0 \\ 3 & 9 & 3 & 0 & 4 \\ 4 & 5 & 0 & 4 & 0 \end{array}$$

ADT / Operations on Graphs:

1. Insertion
    a. Inserting a new vertex
    b. Inserting a new edge
2. Deletion
    a. Deleting a existing vertex
    b. Deleting a existing edge
3. Traversal: To visit all the vertices present in the graph.

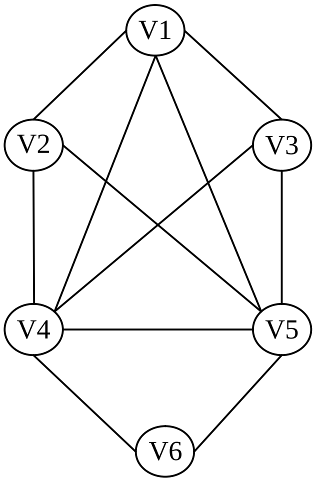
Graph Traversals:

Two techniques are available.

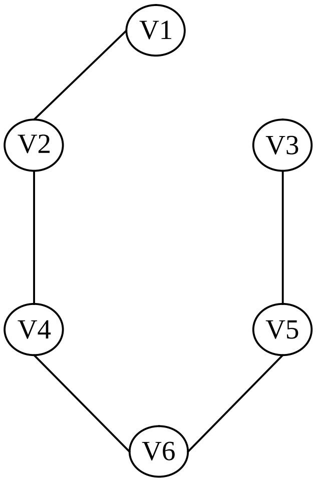
1. Depth First Search (DFS)
2. Breadth First Search (BFS)

Depth First Search (DFS):

- It is similar to the inorder traversal of a binary tree.
- Here, starting from the given node, DFS traversal visit all the nodes upto the deepest level and so on.
- While traversing the vertices, Cyclic path (Closed path) can't be occur. i.e. we can visit a vertex only once.



Graph



DFS of Graph

The sequence of visiting of vertices for the above as follows.

V1-V2-V4-V6-V5-V3

- To traverse a graph in DFS, a stack and one single linked kist is required.
- A stack can be used to maintain the track of all paths from a vertex. Let the name of the stack is OPEN.
- A Single Linked List, VISIT can be maintained to store the vertices already visited.
- Here OPEN is the name of the stack and VISIT is the name of the Single Linked List.

Process:

- Initially the starting vertex will be pushed on to the stack OPEN.
- To visit a vertex, POP a vertex from stack OPEN, and then PUSH all the adjacent vertices into stack OPEN.
- Whenever a vertex s popped, check whether it is already visited or not by searching in Single Linked List VISIT.

- If the vertex is already visited, then we will simply ignore it and we will POP the stack OPEN for the next vertex to be visited. This procedure is continued till the stack is not empty.

Informal description for DFS Traversal of a graph using *array representation* as follows:

Algorithm DFS( )

Input: Adjacent matrix representation of a graph.

Output: DFS traversal of graph.

```
1. PUSH the starting vertex into stack
2. While stack is not empty
   a) POP a vertex v from stack
   b) if vertex v is not visited
      (i) visit the vertex v
      (ii) PUSH all the adjacent vertices of v into stack
   c) end if
3. end loop
```

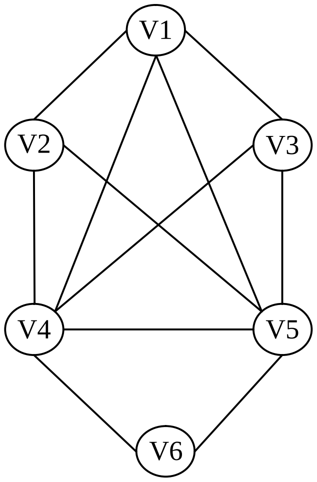
End DFS

Breadth First Search (BFS)

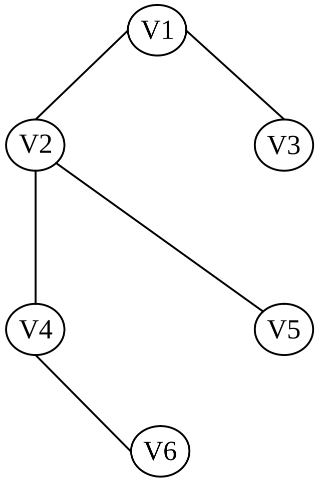
Here any vertex in level i will be visited only after the visiting of all the vertices present in the preceding level. i.e. level i-1.

Simply BFS can be call as Level-by-Level traversal.

Eg:



Graph



BFS of Graph

The order of visiting of vertices in the above BFS traversal of a graph as follows.

V1-V2-V3-V4-V5-V6

- The implement idea of BFS traversal is almost same as DFS traversal except that, BFS uses Queue Data Structure instead of Stack Data Structure in DFS**.**
- Let us assume the name of the Queue is OPENQ to use it in BFS and Single Linked List is VISIT, to store the order of vertices visited during the BFS traversal.

Informal description for BFS Traversal of a graph using array representation as follows:

Algorithm BFS( )

Input: Adjacent matrix representation of a graph.

Output: BFS traversal of graph.

```
1. ENQUEUE the starting vertex into queue
2. While queue is not empty
   a) DEQUEUE a vertex v from queue
   b) if vertex v is not visited
      (i) visit the vertex v
      (ii) ENQUEUE all the adjacent vertices of v into queue
   c) end if
3. end loop
```

End BFS

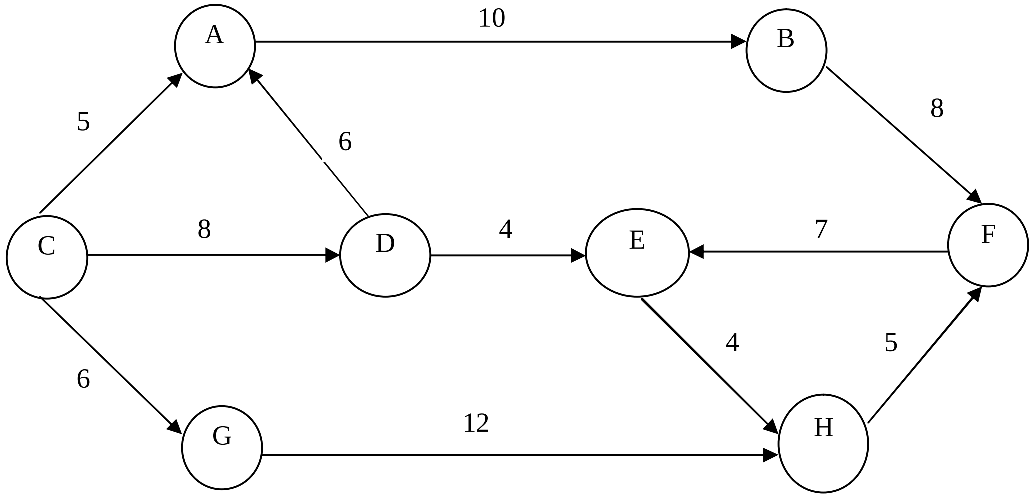
UNIT-V
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. A _____________ is a heap where the value of each parent is less than or equal to the values of its children.
2. Consider any array representation of an n element binary heap where the elements are stored from index 1 to index n of the array. For the element stored at index i of the array (i <= n), the index of the left child and right child are _________. [ ]

   A) 2i+1, 2i B) 2i+1, floor(i/2) C) 2i, floor(i/2) D) 2i, 2i+1

Consider the following graph and answer to the questions 3 to 9



3. The above graph is ____________ [ ]

   A) Complete Graph B) Weighted Graph

   C) Multi Graph D) None of the above
4. In the above graph which of the following is a pendant vertex? [ ]

   A) vertex B B) vertex D C) vertex E D) None of the above
5. In the above graph indegree and outdegree of vertex H is ___ [ ]

   A) indegree - 2 outdegree – 0 B) indegree - 3 outdegree - 0

   C) indegree - 3 outdegree – 1 D) indegree - 2 outdegree - 1
6. The above graph is a __________ [ ]

   A) Connected Graph B) Simple Graph

   C) Cyclic graph D) None of the above

7. The node A is adjacent to __________ node. [ ]

   A) B B) C C) D D) None

8. In the above graph there is a self loop with vertex _________ [ ]

   A) E B) G C) H D)None

9. In a graph if e=(u,v) means .... [ ]

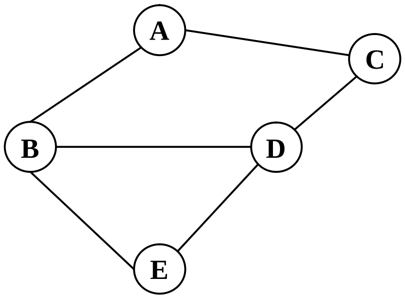
   A) u is adjacent to v but v is not adjacent to u.

   B) e begins at u and ends at v

   C) u is node and v is an edge.

   D) both u and v are edges.

Consider the following graph to answer the questions 10 to 10



10. The above graph is a _____________ [ ]

   A) Weighted graph B) Simple graph C) Acyclic Graph D) None

11. The adjacent vertices of node A are _________ [ ]

   A) B, D, E B) B, D, C C) E, D D) None

12. The above graph is a ______________ [ ]

   A) Connected graph B) Complete graph C) Both A&B D) None

13. For an undirected graph with n vertices and e edges, the sum of the degree of each vertex is equal to [ ]

   A) 2n B) (2n-1)/2 C) 2e D) $e^2/2$

14. A graph with n vertices will definitely have a parallel edge or self loop, if the total number of edges are [ ]

   (A) more than n (B) more than n+1

   (C) more than (n+1)/2 (D) more than n(n-1)/2

15. The maximum degree of any vertex in a simple graph with n vertices is_____ [ ]

16. An adjacency matrix representation of a graph cannot contain information of ______

   (A) Nodes (B) edges [ ]

   (C) Direction of edges (D) parallel edges

17. How many undirected graphs (not necessarily connected) can be constructed out of a given set V= {V 1, V 2,…V n} of n vertices ? [ ]

(A) n(n-l)/2 (B) 2^n (C) n! (D) 2^(n(n-1)/2)

18. The data structure required for Breadth First Traversal on a graph is _________

(A) Queue (B) Stack (C) Array (D) Tree

19. Which of the following statements is/are TRUE for an undirected graph? [ ]

P: Number of odd degree vertices is even

Q: Sum of degrees of all vertices is even

A) P Only B) Q Only

C) Both P and Q D) Neither P nor Q.

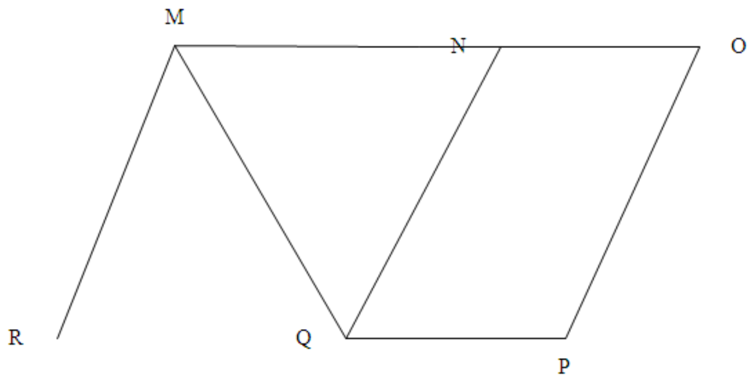
20. The Minimum no.of edges in a connected cyclic graph on n vertices is___? [ ]

(A) n-1 (B) n (C) n+1 (D) None of the above

21. The no.of simple graphs on **n** labled vertices is___ [ ]

(A) n (B) n(n-1)/2 (C) $2^{n(n-1)/2}$ (D) n(n+1)/2

22. The BFS Algorithm has been implemented using Queue Data Structure. One possible order of visiting nodes in the following graph is [ ]



(A) MNOPQR (B) NQMPOR (C) QMNPRO (D) QMNPOR

SECTION-B

Descriptive Questions

1. Show the result of inserting the keys: 14, 5, 12, 6, 4, 8, 9, 13, 11, 2, 18, 30 one at a time into an initially empty Max heap with neat diagrams.
2. Show the result of inserting the keys: 10, 12, 8, 14, 6, 5, 1, 3 one at a time into an initially empty Min heap. Apply deleteMin operation on the resulting min heap
3. Construct a Max heap for the following keys: 4, 67, 23, 89, 12, 8, 7, 44, 78, 64, 70. Apply deleteMax operation on the resulting max heap

4. Sort the following keys using Heap sort: 5, 8, 11, 3, 9, 2, 10, 1, 45, 32.
5. Write adjacency matrix representation of graph with an example.
6. Write linked representation of graph with an example.
7. Write set representation of graph with an example.
8. Write DFS Algorithm& Write BFS Algorithm.
9. Consider the graph given below
    a) Write the adjacency matrix of G1.
    b) Give Linked list representation of G1.
    c) Give Set representation of G1.
    d) Is the graph complete?
    e) Is the graph strongly connected?
    f) Find out the degree of each node.
    g) Is the graph regular?

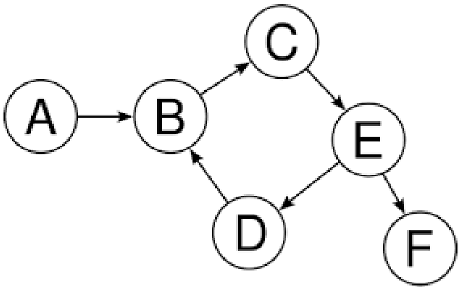
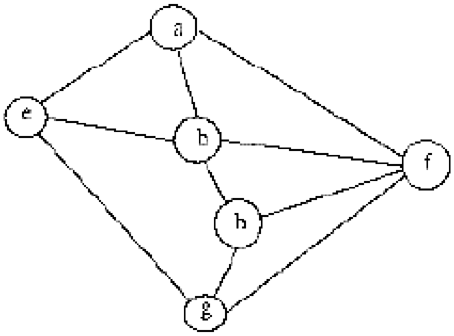


Fig. Graph G1

10. Consider the following adjacency matrix, draw the weighted graph.

$$\begin{pmatrix} 0 & 4 & 0 & 2 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

11. Consider the following graph

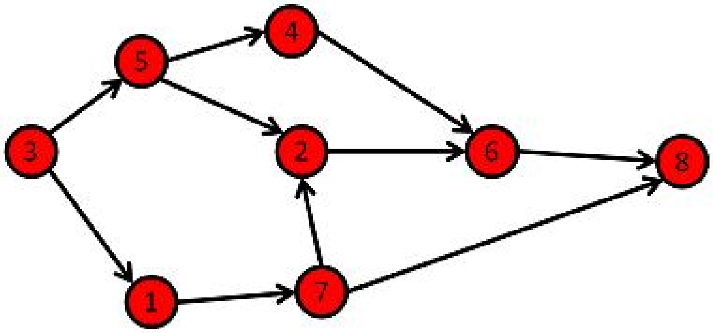


Among the following sequences

i) a b e g h f ii) a b f e h g iii) a b f h g e iv) a f g h b e

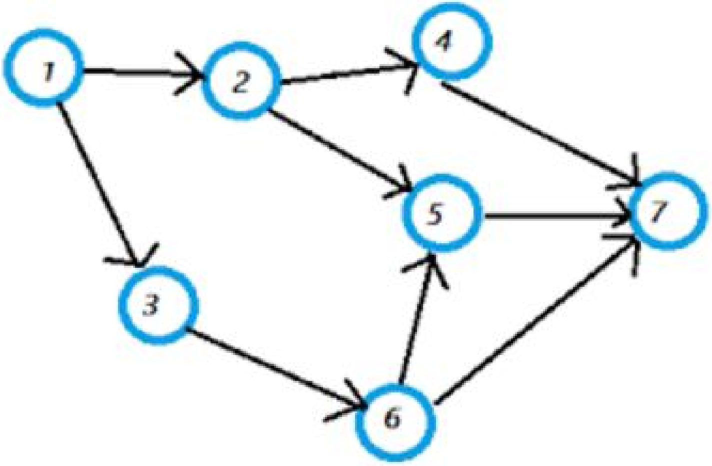
Which are depth first traversals of the above graph?

12. Consider the following graph



What is breadth first traversal of the above graph if starting vertex is 3?

13. Consider the following graph



What is the depth first traversal of the above graph if starting vertex is 1?

Section C

Questions asked in GATE

1. Consider any array representation of an n element binary heap where the elements are stored from index 1 to index n of the array. For the element stored at index i of the array (i <= n), the index of the parent is_______ **(GATE-CS-2001)** [ ]

    A) i-1 B) floor(i/2) C) ceiling(i/2) D)(i+1)/2

2. In a Binary max heap containing n numbers, the smallest element can be found in time **(GATE 2006)** **[ ]**

    A) O(n) B) O(logn) C) O(loglogn) D) O(1)

3. Which of the following sequences of array elements forms a heap? [ ]

    A) {23, 17, 14, 6, 13, 10, 1, 12, 7, 5} **(GATE IT 2006)**

    B) {23, 17, 14, 6, 13, 10, 1, 5, 7, 12}

    C) {23, 17, 14, 7, 13, 10, 1, 12, 5, 7}

    D) {23, 17, 14, 7, 13, 10, 1, 5, 6, 12}

4. Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap? **(GATE CS 2009)** **[ ]**

    A) 25,12,16,13,10,8,14 B) 25,14,16,13,10,8,12

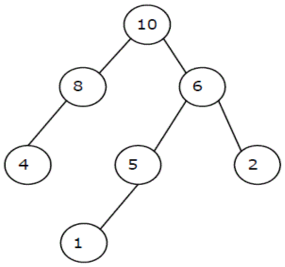
    C) 25,14,12,13,10,8,16 D) 25,16,12,13,10,8,12

5. What is the content of the array after two delete operations on the correct answer to the previous question? **(GATE CS 2009**) [ ]

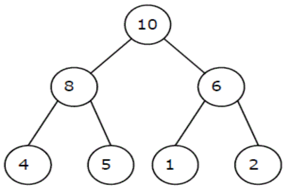
    A) 14,13,12,10,8 B) 14,12,13,8,10 C) 14,13,8,12,10 D) 14,13,12,8,10

6. A max-heap is a heap where the value of each parent is greater than or equal to the values of its children. Which of the following is a max-heap? **(GATE CS 2011)** **[ ]**

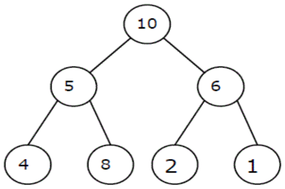
(A)



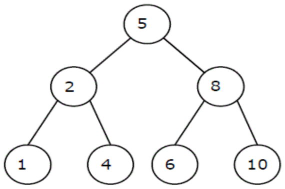
(B)



(C)



(D)



7. A priority queue is implemented as a Max-Heap. Initially, it has 5 elements. The level-order traversal of the heap is: 10, 8, 5, 3, 2. Two new elements 1 and 7 are inserted into the heap in that order. The level-order traversal of the heap after the insertion of the elements is: **(GATE-CS-2014)** [ ]

   (A) 10, 8, 7, 3, 2, 1, 5 (B) 10, 8, 7, 2, 3, 1, 5

   (C) 10, 8, 7, 1, 2, 3, 5 (D) 10, 8, 7, 5, 3, 2, 1

8. Consider a max heap, represented by the array: 40, 30, 20, 10, 15, 16, 17, 8, 4. Now consider that a value 35 is inserted into this heap. After insertion, the new heap is **(GATE-CS-2015) [ ]**

   A) 40, 30, 20, 10, 15, 16, 17, 8, 4, 35 B) 40, 35, 20, 10, 30, 16, 17, 8, 4, 15

   C) 40, 30, 20, 10, 35, 16, 17, 8, 4, 15 D) 40, 35, 20, 10, 15, 16, 17, 8, 4, 30

9. A 3-ary max heap is like a binary max heap, but instead of 2 children, nodes have 3 children. A 3-ary heap can be represented by an array as follows: The root is stored in the first location, a[0], nodes in the next level, from left to right, is stored from a[1] to a[3]. The nodes from the second level of the tree from left to right are stored from a[4] location onward. An item x can be inserted into a 3-ary heap containing n items by placing x in the location a[n] and pushing it up the tree to satisfy the heap property. Which one of the following is a valid sequence of elements in an array representing 3-ary max heap? **(GATE 2006)** [ ]

   A) 1, 3, 5, 6, 8, 9 B) 9, 6, 3, 1, 8, 5 C) 9, 3, 6, 8, 5, 1 D) 9, 5, 6, 8, 3, 1

10. Suppose the elements 7, 2, 10 and 4 are inserted, in that order, into the valid 3- ary max heap found in the above question, which one of the following is the sequence of items in the array representing the resultant heap? **(GATE CS 2006)** [ ]

    A) 10, 7, 9, 8, 3, 1, 5, 2, 6, 4 B) 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

    C) 10, 9, 4, 5, 7, 6, 8, 2, 1, 3 D) 10, 8, 6, 9, 7, 2, 3, 4, 1, 5

11. Consider the following array of elements. ⟨89, 19, 50, 17, 12, 15, 2, 5, 7, 11, 6, 9, 100⟩. The minimum number of interchanges needed to convert it into a max-heap is **(GATE-CS-2015)** [ ]

    A) 4 B) 5 C) 2 D)3

12. An operator delete(i) for a binary heap data structure is to be designed to delete the item in the i-th node. Assume that the heap is implemented in an array and i refers to the i-th index of the array. If the heap tree has depth d (number of edges on the path from the root

to the farthest leaf), then what is the time complexity to re-fix the heap efficiently after the removal of the element? **(GATE 2016)** [ ]

A) O(1) B) O(d) but not O(1) C) O(2^d) but not O(d) D) O(d 2^d) but not O(2^d)

13. A complete binary min-heap is made by including each integer in [1, 1023] exactly once. The depth of a node in the heap is the length of the path from the root of the heap to that node. Thus, the root is at depth 0. **(GATE 2016)** **[ ]**

A) 6 B) 7 C) 8 D) 9

14. Which of the following statements is/are TRUE for undirected graphs?

P: Number of odd degree vertices is even.

Q: Sum of degrees of all vertices is even. **(GATE 2013)** **[ ]**

A) P only B) Q only C) Both P and Q D) Neither P nor Q

15. Let G be a simple undirected planar graph on 10 vertices with 15 edges. If G is a connected graph, then the number of bounded faces in any embedding of G on the plane is equal to_________ **(GATE 2012)** **[ ]**

A) 3 B) 4 C) 5 D) 6

16. Which one of the following is TRUE for any simple connected undirected graph with more than 2 vertices? **(GATE 2009)** **[ ]**

A) No two vertices have the same degree. B) At least two vertices have the same degree.

C At least three vertices have the same degree. D All vertices have the same degree.

UNIT –6

Objective:

- To explore dictionaries.

Syllabus:

Hashing:Basic concepts

Hashing Functions: (Division Method, Multiplication Method),

Collision resolution techniques:-

Open Hashing-Examples,

Closed Hashing:-Linear Probing,QuadraticProbing,Double Hashing Examples

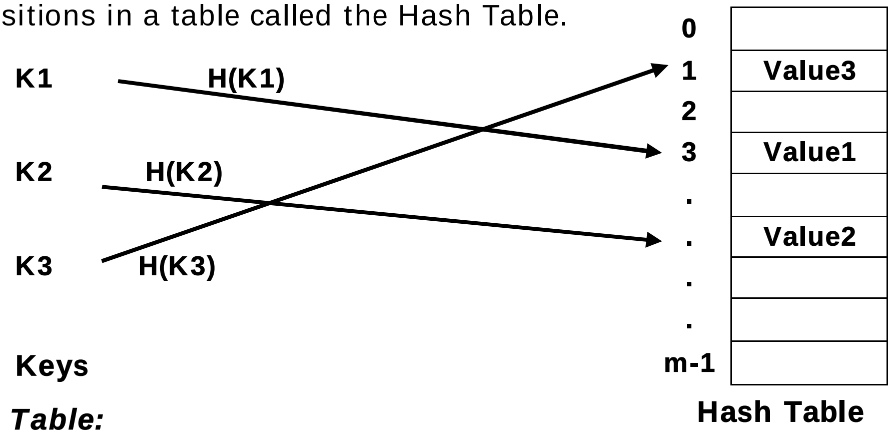
Learning Outcomes:

Student will be able to:

1. Describe Hashing and Various Hash Functions.
2. Insert elements into the Hash Table using various Hash Functions
3. Apply various Collision Resolution Techniques to resolve collisions in Hashing.
4. Implement Hash Table Restructuring when the Hash Table is nearly full.

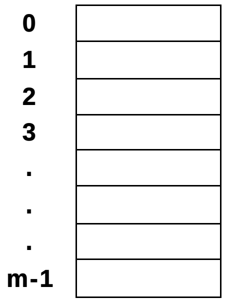
LEARNING MATERIAL

- ***Hashing:***
  - The main goal of Hashing/Hashed search is to find the data with only one test.
  - Hashing is a process of computing the address where an element is to be inserted or found in a hash table by applying a hash function to a given key.
  - Hashing is a Key-to-index transformation in which the keys map to index in an array called as Hash Table.
  - This method used a Hash function (Hashing algorithm) to map keys into positions in a table called the Hash Table.



- ***Hash Table:***
  - *Definition:*A Hash table is a data structure that stores elements and allows insertions, lookups, and deletions to be performed in O(1) time.
  - A hash table consists of an array in which data is accessed via a special index called a Key.
  - Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associated with some key.

- For example, for storing an employee record in the hash table the employee ID will work as a key.

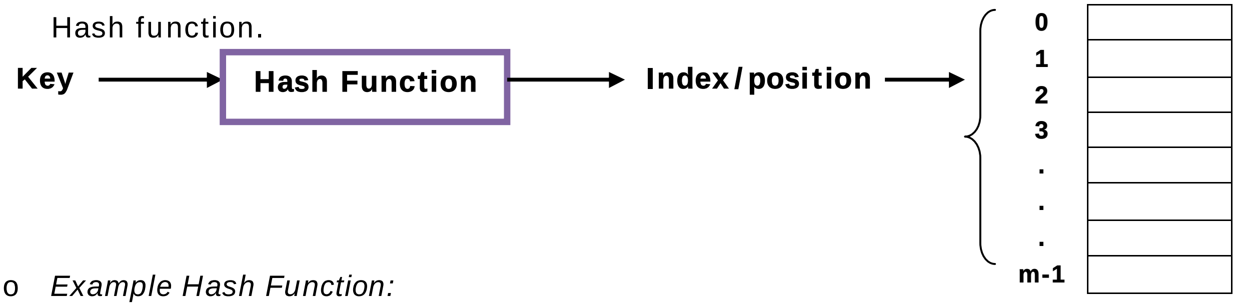


The positions in the hash table are indexed 0 through m-1

Hash table of Size m

- Load factor (α) = (no. of elements) / (no. of table slots)

- ***Hash Function:***
  - Hash functions are primarily used in hash tables, to quickly locate a data record given its search key
  - *Definition:* Hash function is used to map the search key to an index; the index gives the place in the hash table where the corresponding record should be stored.
  - A Key is mapped into one of the m slots of the Hash table using Hash function.



Hash Table

  - *Example Hash Function:*

    H(K)= K%m

    Here K is the Key, H is the Hash Function & m is the size of the Hash Table
  - A Hash function may map several keys into the same position (index) of the hash table.
  - Each position of the hash table is often called as a **'Bucket'**. Bucket consists of number of slots for holding elements.

- Position h(K) is the **‘Home Bucket’** for the element whose key is K.

- ***Example:***

  Consider inserting the keys 80, 40, 65, 24, 44, 58 into a hash table of size m=11 using hash function h(k)=k mod m

  1. K=80

     h(80) = 80 mod 11 = 3
  2. K=40

     h(40) = 40 mod 11 = 7
  3. K=65

     h(65) = 65 mod 11 = 10
  4. K=24

     h(24) = 24 mod 11 = 2
  5. K=44

     h(44) = 44 mod 11 = 0
  6. K= 58

     h(58) = 58 mod 11 = 3 (already occupied by 80 -> Collision has occurred)

0	**44**
1	
2	**24**
3	**80 ***
4	
5	
6	
7	**40**
8	
9	
10	**65**

Hash Table

- A **Collision** occurs whenever two or more different keys have the same Home Bucket. (in our example keys 80 & 58 have same Home bucket 3)
- *“The situation in which the hash function returns the same index for more than one key is called as collision”*.
- An **Overflow**occurs when there is no room in the home bucket for the new element.
- ***Applications of Hash Tables:***
  - Used to implement Associative Arrays
  - Database Management Systems – Telephone book database, Library books Catalogue, Employee details
  - Compiler - Symbol Table
  - Operating System - Page Mapping Tables, Cache Memory
- ***Analysis of Hash tables:***Time complexity

Operation	Average case	Worst case
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

➢ HASH FUNCTIONS/ HASHING METHODS:

- Hash function is used to map the search key to an index; the index gives the place in the hash table where the corresponding record should be stored.
- A Key is mapped into one of the m slots of the Hash table using Hash function.
- Some Commonly used Hash functions are:
  1. Division Method or Modulo Division
  2. Multiplication Method

1. *Division Method:*

- This method divides the key by the hash table size and uses the remainder as the index of the hash table.
- In this method, we map a key K into one of the m slots of the hash table by taking remainder of K divided by m.
- The hash function is

  H(K)= K % m

  Here K is the Key, H is the Hash Function & m is the size of the Hash Table
- *Example:*

  Consider inserting keys 26,33,76,86 into a hash table of size m=11 using division method

  h(26) = 26 % 11 = 4

  h(33) = 33 % 11 = 0

  h(76) = 76 % 11 = 10

  h(86) = 86 % 11 = 9

0	**33**
1	
2	
3	
4	**26**
5	
6	
7	
8	
9	**86**
10	**76**

Hash Table

- Efficiency of this method depends on the value of 'm'
- When using a division method, we usually avoid certain values of m.
  a. m should not be a power of 2

b. A prime number not too close to an exact power of 2 is often a good choice for m. Because it results in fewer collisions than other values for m.

2. *Multiplication Method:*

- The multiplication method for generating hash functions operates in two steps:
  1) Multiply the key **k** by a constant **A** in the range 0 < A < 1 and extract the fractional part of **kA**
  2) Multiply this fractional part by **m** and take the floor.

- The hash function is

$$\mathbf{h(k) = \lfloor m(kA \bmod 1) \rfloor}$$

Where $\mathbf{kA \bmod 1 = kA - \lfloor kA \rfloor}$ i.e the fractional part of KA

A = 0.61804 (Golden value suggested by Knuth)

- *Example:*

Consider m=32 and key k=100

(i). kA = 100 * 0.61804

= 61.804

Taking the fractional part of the result

kA mod 1 = 61.804 mod 1

= 0.804

(ii). m* (kA mod 1) = 32 * 0.804

= 25.728

By taking the floor of the result the index we got is 25

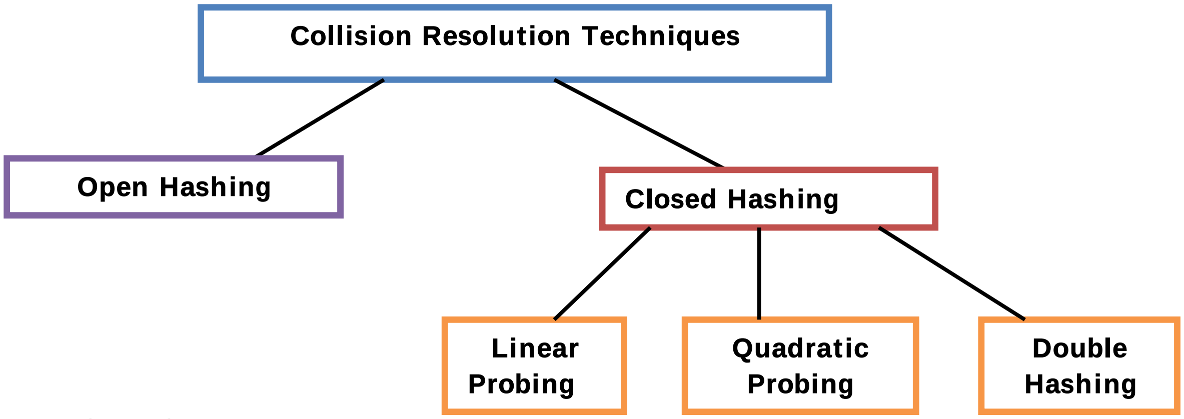
h(32) = 25

- The advantage of this method is that the value of m is not critical. Typically it is chosen to be a power of 2.

➢ <u>COLLISION RESOLUTION TECHNIQUES:</u>

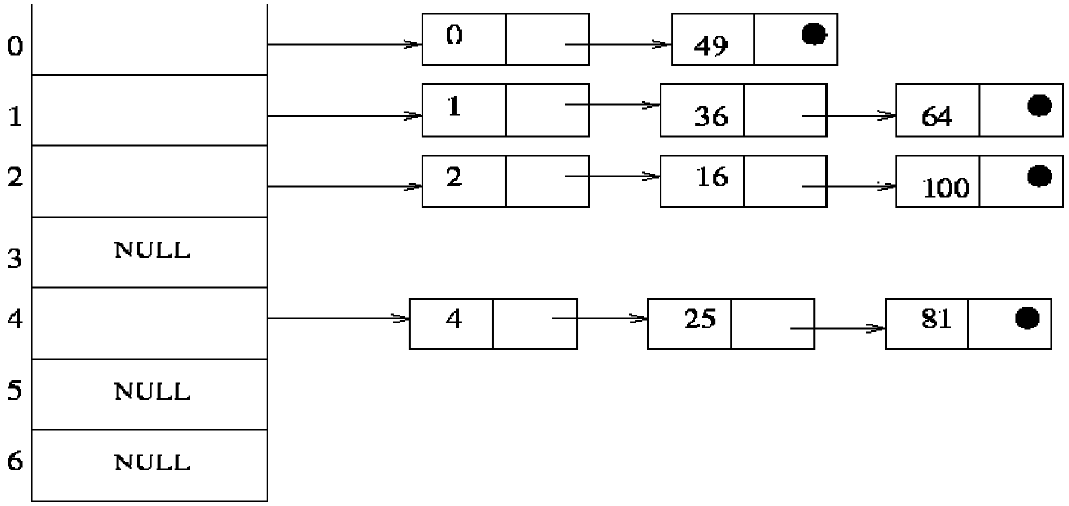
- ***Definition:*** The situation in which a hash function maps two or more keys to same index (Home bucket) in hash table is called as Collision.
- When two keys hash to the same position in a hash table they collide.

- If x1 and x2 are two different keys, it is possible that **h(x1) = h(x2)**, then collision is said to occur between keys x1 and x2.
- Whatever may be the hash function used in hashing, complete removal of collisions is almost impossible. So collisions in hashing cannot be ignored.
- Two techniques used to resolve collisions are
  1. Open hashing or Separate chaining
  2. Closed hashing or Open addressing



1) OPEN HASHING:

- It is also known as Separate chaining
- In this technique, hash table is implemented as an array of Linked lists.
- In this technique, *"hash table is an array of pointers and each pointer will point to one linked list".*
- Whenever a collision occurs then a linked list (chain) is maintained at that home bucket.
- All the elements that hash to the same value are placed in the same home bucket in a Linked list.
- Hash table that used open hashing is called as Open Hash table(or chained hash table).
- **Example:**
  Consider the keys 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100. Let the hash function be:h(x) = x % 7



- ***Advantages:***

1. Simple and effective approach to collision resolution.
2. Hash table size need not be a prime number

- ***Disadvantages:***

1. Additional Data structure (linked list) needs to be used to accommodate collision data.
2. Wastage of memory space for storing pointers (linked lists).
3. More number of collisions leads to unevenly distributed keys resulting in
    a. Long length lists
    b. Increased search time
    c. Many empty spaces in the hash table

2) CLOSED HASHING:

- It is also known as Open Addressing.
- In Open Addressing, all the elements are stored in the hash table itself. That is, each table entry contains either an element or NULL.
- It avoids pointers
- Instead of following pointers as in case of Open hashing, we compute the sequence of slots to be examined.
- The process of examining the locations in the hash table is called ***Probing***.
- To perform insertion using open addressing, we successively examine, or probe, the hash table until we find an empty slot in which to put the key.
- Collisions are handled by generating a sequence of rehash values

h : U × {0,1,2,....... } -->{0,1,2,.......,m-1}

Universe of Keys Probe number Hash Table

- Given a key x, it has a hash value h(x,0) and a set of rehash values h(x,1), h(x,2), h(x,3),.........,h(x,m-1)
- Closed Hashing Techniques:
  1. Linear Probing
  2. Quadratic Probing
  3. Double Hashing

1. ***Linear Probing:***

- Linear probing is a scheme for resolving collisions of values by *sequentially searching the hash table for a free location*.
- Suppose the hash table size is m, and key value is mapped to index i, with a hash function. If collision has occurred at i^{th} index. then follow the following sequence of locations in hash table and do sequential / linear search.

  i+1, i+2,........,m,0,1,2,........i
- The search will continue until any one of the following cases occur
  1. The key value is already present in the table
  2. The unoccupied location(empty slot) is encountered.
  3. It reaches to the location where search was started.
- Here hash table is considered as circular, so that when the last location is reached, the search proceeds to the starting location of the table.
- Hash function in linear probing is defined as

  $\boxed{\mathbf{h^1(k,i) = (h(k) + i) \text{ mod } m} \text{ fori=}}$0,1,2,3,....,m-1 and h(k) is the primary hash function
- **Example:**Insert the following keys 15,11,25,16,9,8,12,8,23,47into hash table of size m=11 using linear probing, consider the primary hash function as h(k)=k mod m

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Initially hashtable is empty

0	
1	
2	
3	
4	15
5	
6	
7	
8	
9	
10	

Inserting 15

0	11
1	
2	
3	
4	15
5	
6	
7	
8	
9	
10	

Inserting 11

0	11
1	
2	
3	25
4	15
5	
6	
7	
8	
9	
10	

Inserting 25

0	11
1	
2	
3	25
4	15
5	16
6	
7	
8	
9	
10	

Inserting 16

0	11
1	
2	
3	25
4	15
5	16
6	
7	
8	
9	9
10	

Inserting 9

0	11
1	
2	
3	25
4	15
5	16
6	
7	
8	8
9	9
10	

Inserting 8

0	11
1	12
2	
3	25
4	15
5	16
6	
7	
8	8
9	9
10	

Inserting 12

0	11
1	12
2	
3	25
4	15
5	16
6	
7	
8	8
9	9
10	

Inserting 8

Here already key 8 is presented in hash table. So does not insert once again.

0	11
1	12
2	23
3	25
4	15
5	16
6	
7	
8	8
9	9
10	

Inserting 23

Here collision occurs at 1st index. So next index is 2

0	11
1	12
2	23
3	25
4	15
5	16
6	47
7	
8	8
9	9
10	

Inserting 47

Here collision occurs at 3, 4, 5 indexes. So next index is 6

- **rawbacks of linear probing:**
  a. The major drawback isthat, as half of the hash table is filled, there is a tendency towards clustering. The key values are clustered in large groups and as a result sequential search becomes slower. This kind of clustering known as primary clustering.
  b. Performance degrades as the hash table gets nearly full.

2. ***Quadratic Probing:***

- Quadratic probing is a collision resolution method that eliminates the primary clustering problem in linear probing.
- If there is a collision at index i

  In linear probing , the next indexes to be probe are i+1, i+2, i+3,-------------

  But in quadratic probe, the next indexes to be probe are $i+1^2$, $i+2^2$, $i+3^2$,------
- If m is the size of the hash table and h(k) is the hash function , then quadratic probing search the indexes as

$\boxed{\mathbf{h^1(k,i) = (h(k)+ i^2) \bmod m}}$ for i=0,1,2,3,----------

- ***Example:*** Insert the following keys 89, 18, 49, 58, 69 into hash table of size m=10 using quadratic probing, consider the Hash function h(k) = k % m

  Hash function used in quadratic probing is $\mathbf{h^1(k,i) = (h(k)+ i^2) \bmod m}$

  1. $h^1(89,0)$= (h(89) + 0)%10

  h(89) = 89%10 = 9

  $h^1(89,0)$ = 9 (empty slot)

  2. $h^1(18,0)$ = (h(18) + 0)%10

  h(18) = 18%10 = 8

  $h^1(18,0)$ = 8 (empty slot)

  3. **a)**$h^1(49,0)$ = (h(49) + 0)%10

  h(49) = 49%10 =9

  $h^1(49,0)$ = 9 (Collision)

  b)$h^1(49,1)$ = (h(49) + 1^2)%10

  = (9+1) %10

  $h^1(49,1)$ = 0 (empty slot)

  4. **a)**$h^1(58,0)$ = (h(58) + 0)%10

  h(58) = 58%10 = 8

  $h^1(58,0)$ = 8 (Collision)

0		0		0		0	49	0	49	0	49
1		1		1		1		1		1	

b) $h^1(58,1) = (h(58) + 1^2)\%10$

$=(8+1)\ \%10$

$h^1(58,1)\ =\ \ 9$ (Collision)

c)$h^1(58,2) = (h(58) + 2^2)\%10$

$= (8+4)\ \%10$

$= 2$ (empty slot)

5. **a)**$h^1(69,0) = (h(69) + 0)\%10$

$h(69) = 69\%10= 9$

$h^1(69,0) = 9$ (Collision)

b)$h^1(69,1) = (h(69) + 1^2)\%\ 10$

$= (9+1)\%10$

$h^1(69,1) = 0$ (Collision)

c)$h^1(69,2) = (h(69) + 2^2)\%\ 103$

$=(9+4)\%10$

$h^1(69,2) = 3$ (empty slot)

	Empty hash table	Insert 89	Insert 18	Insert 49	Insert 58	Insert 69
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89
10						

- **Drawbacks of Quadratic probing:**

  a) There is no guarantee of finding an empty cell once more than half of the table gets full or even before that, if the table size is not prime.

  b) It suffers from Secondary Clustering. If two keys have the same initial probe position, then their probe sequences are the same.

  If $h^1(k1,0) = h^1(k2,0)$ then $h^1(k1,i) = h^1(k2,i)$ for i=1,2,3.......

3. ***Double Hashing*:**

- Double Hashing overcomes the problem of Secondary Clustering.
- This approach uses 2 hash functions. This second hash function results the value of m for a key which is different from the first hash function value.
- Double hashing uses hash function of the form

  h(k,i) = (h1(k) + i h2(k)) mod m for i=0,1,2,3.......
- Here h1 and h2 are the auxiliary hash functions
- There are two important rules to be followed for the second function:

  1) It must never evaluate to Zero

  2) Must make sure that all the slots can be examined
- Generally used hash functions are

  First hash function=>**h1(k)=k mod m**

  Second hash function =>**h2(k)=R-(k mod R)**
- In second hash function, R is a prime number, which is smaller than m.

- ***Disadvantage*** of double hashing is that Complex computation is required. We have to use the second hash function when there is a collision.
- ***Example:***Insert the following keys89, 18, 49, 58, 69, 59into a hash table of size 10 using Double hashing

  h(k,i) = (h1(k) + i h2(k)) mod m for i=0,1,2,3.......

  h1(k) = k%m

  h2(k) = R – k mod r

  m= 10 and consider R = 7 which is prime and less than m

1. h(89,0) = (h1(89)+ 0*h2(89)) mod 10

h1(89) = 89%10 =9

h(89,0) = 9 (empty slot)

2. h(18,0) = (h1(18)+ 0*h2(18)) mod 10

h1(18) = 18%10 =8

h(18,0) = 8 (empty slot)

3. **a)**h(49,0) = (h1(49)+ 0*h2(49)) mod 10

h1(49) = 49%10 = 9

h(49,0) = 9 (Collision)

b)h(49,1) = (h1(49)+ 1*h2(49)) mod 10

h1(49) = 49%10 = 9

h2(49) = 7-(49%7) = 7

h1(49,1) = (9+7) mod 10

= 6 (empty slot)

4. **a)** h(58,0) = (h1(58)+ 0*h2(58)) mod 10

h1(58) = 58%10 = 8

h(58,0) = 8 (Collision)

b)h(58,1) = (h1(58)+ 1*h2(58)) mod 10

h1(58) = 58%10 = 8

h2(58) = 7-(58%7) = 5

h(58,1) = (8+5) mod 10

= 3 (empty slot)

5. **a)** h(69,0) = (h1(69)+ 0*h2(69)) mod 10

h1(69) = 69%10 = 9

h(69,0) = 9 (Collision)

b)h(69,1) = (h1(69)+ 1*h2(69)) mod 10

h1(69) = 69%10 = 9

h2(69) = 7-(69%7) = 1

h(69,1) = (9+1) mod 10

= 0 (empty slot)

6. a) h(59,0) = (h1(59)+ 0*h2(59)) mod 10

h1(59) = 59%10 = 9

h(59,0) = 9 (Collision)

b)h(59,1) = (h1(59)+ 1*h2(59)) mod 10

h1(59) = 59%10 = 9

h2(59) = 7-(59%7) = 4

h(59,1) = (9+4) mod 10

h(59,1) = 3 (Collision)

c)h(59,2) = (h1(59)+ 2*h2(59)) mod 10

h1(59) = 59%10 = 9

h2(59) = 7-(59%7) = 4

h(59,2) = (9+2*4) mod 10

= (9+8) mod 10

h(59,2) = 7 (empty slot)

	Empty Hash table		**Insert 89**		**Insert 18**		**Insert 49**		**Insert 58**		**Insert 69**		**Insert 59**
0		0		0		0		0		0	69	0	69
1		1		1		1		1		1		1	
2		2		2		2		2		2		2	
3		3		3		3		3	58	3	58	3	58
4		4		4		4		4		4		4	
5		5		5		5		5		5		5	
6		6		6		6	49	6	49	6	49	6	49
7		7		7		7		7		7		7	59
8		8		8	18	8	18	8	18	8	18	8	18
9		9	89	9	89	9	89	9	89	9	89	9	89

UNIT-VI
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. The mapping of keys to indices of a hash table is done using _______________
2. ____________________ is the formula used for Multiplication hash function
3. Define Bucket in a hash table
4. Define Home Bucket in a hash table
5. Given a Hash table of size m=17 then its range of indices are ____________
6. Load factor (α) = ____________________________________
7. In a hash table of length 11, the key value 80 can be placed using division hash function at index __________

   A). 3 B). 4 C). 5 D). 6 **[ ]**
8. ____________ occurs when hashing produces same index for two different keys.
9. List different collision resolution techniques
10. The disadvantage of Separate chaining is [ ]

    A). Need to maintain a Separate Data Structure

    B). Need more Memory Space

    C). Both A & B

    D). None of these
11. Closed hashing is also called as _____________________
12. Primary Clustering occurs in **[ ]**

    A). Quadratic Probing B). Linear Probing

    C). Double hashing D). All of the above
13. In a hash table of size 13, the elements to be inserted are 18, 31, and 44 using Division hash function. With Quadratic probing 44 can be placed in _____ cell.

    A). 6 B). 7 C). 8 D). 9 **[ ]**
14. Primary clustering and secondary clustering are solved by _______________
15. __________ is a collision resolution technique that uses linked lists to handle collisions [ ]

    A). Linear probing B). Quadratic probing

    C). Double hashing D). Open Hashing

SECTION-B

Descriptive Questions

1. Calculate the hash table indexes using Division and Multiplication hash functions for the keys: 25, 4, 16, 100, 32, 58 with the size of the hash table as m=11
2. Construct the open hash table using separate chaining for the input: 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 140 using the hash function h(k)= k mod 11
3. Show the result of inserting the keys: 12, 44, 13, 88, 23, 94, 11, 39, 16 into a hash table of size m =13 with the primary hash function as h(k) = k % m using Linear Probing
4. Show the result of inserting the keys: 12, 44, 13, 88, 23, 94, 11, 39, 20 into a hash table of size m =11 with the primary hash function as h(k) = k % m using Quadratic Probing
5. Show the result of inserting the keys: 15, 11, 25, 16, 36, 47, 22 into a hash table of size m =11 using Double hashing with h1(k) = k % m and h2(k)=R- (k mod R)where R < m and is prime
6. Consider inserting the keys: 20, 29, 45, 49, 52, 59, 65 into a hash table of size m=10 using the primary hash function as h(k) = k % m. Illustrate the result of inserting these keys using quadratic probing with $h^1(k)=(h(k) + i + 3i^2) \bmod m$
7. Consider inserting the keys 10, 22, 31, 4, 15, 28, 17 into a hash table of length m = 11 using the primary hashing function $h(k) = k \bmod m$. Illustrate the result of inserting these keys usingDouble hashing with rehashing function $h^1(k) = (h(k) + i(1 + k \bmod(m - 1)) \bmod m$.
8. Consider inserting the keys 7, 18, 48, 10, 36, 25, 47 into a hash table of size m=10 using linear probing. Apply hash table restructuring and show the resulting new Hash table.

Section C

Questions asked in GATE

1.Which one of the following hash functions on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for i ranging from 0 to 2020?**[gate 2002]** **[ ]**

(A) $h(i)=i^2$ mod10 **(B)** $h(i)=i^3$ mod10

(C) $h(i)=(11*i^2)$mod10 **(D)** $h(i)=(12*i)$mod

2.Given a hash table T with 25 slots that stores 2000 elements, the load factor α for T is ___________ **[ ][gate 2005]**

A) 80 B) 0.0125 C) 8000 D) 1.25

3.A hash function h defined h(key)=key mod 7, with linear probing, is used to insert the keys 44, 45, 79, 55, 91, 18, 63 into a table indexed from 0 to 6. What will be the location of key 18?**[gate 2007]** **[ ]**

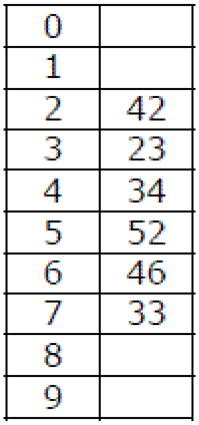
(A) 3 **(B)** 4 **(C)** 5 **(D)** 6

4.Which of the following statement(s) is TRUE?**[gate 2008]** **[ ]**

1. A hash function takes a message of arbitrary length and generates a fixed length code.
2. A hash function takes a message of fixed length and generates a code of variable length.
3. A hash function may give the same hash value for distinct messages.

(A) I only **(B)** II and III only
(C) I and III only **(D)** II only

5.A hash table of length 10 uses open addressing with hash function h(k)=k mod 10, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.



0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Which one of the following choices gives a possible order in which the key values could have been inserted in the table? [ ]

(A) 46, 42, 34, 52, 23, 33 **(B)** 34, 42, 23, 52, 33, 46

(C) 46, 34, 42, 23, 52, 33 **(D)** 42, 46, 33, 23, 34, 52

6.Consider a hash function that distributes keys uniformly. The hash table size is 20. After hashing of how many keys will the probability that any new key hashed collides with an existing one exceed 0.5.**[gate 2009]** **[ ]**

(A) 5 **(B)** 6 **(C)** 7 **(D)** 10

7.A hash table has space for 100 records. What is the propability of collision before the table is 10% full?**[gate 2015]**

A). 0.45 B). 0.5 C). 0.3 D). 0.34 **[ ]**

8.Consider a 13 element hash table for which f(key)=key mod 13 is used with integer keys. Assuming linear probing is used for collision resolution, at which location would the key 103 be inserted, if the keys 661, 182, 24 and 103 are inserted in that order?**[gate 2016]** **[ ]**

(A) 0 **(B)** 1 **(C)** 11 **(D)** 12

9.Consider a hash table with 9 slots. The hash function is $h(k) = k$ mod 9. The collisions are resolved by chaining. The following 9 keys are inserted in the order: 5, 28, 19, 15, 20, 33, 12, 17, 10. The maximum, minimum, and average chain lengths in the hash table, respectively, are

[Gate 2017] **[ ]**

(A) 3, 0, and 1 (B) 3, 3, and 3 (C) 4, 0, and 1 D) 3, 0, and 2

10.Consider a hash table of size seven, with starting index zero, and a hash function $(3x+4)$mod7. Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing? Note that - denotes an empty location in the table.**[gate 2007]** **[ ]**

.A) 8, - , - , - , - , - , 10 B) 1, 8, 10, - , - , - , 3 (C) 1, - , - , - , - , - , 3 (D) 1, 10, 8, - , - , - , 3