Data Structures Notes

By

Dr. G. Nagaraju

B.Tech(CSE), M.Tech(AI from HCU), Ph.D (CSE from JNTUH)
Assistant Professor
Department of CSE(AIML&IOT)
VNR Vignana Jyothi Institute of Engineering and Technology
E-mail: nagaraju_g@vnrvjiet.in

Data Structures Syllabus

COURSE OBJECTIVES:

- To introduce various searching and sorting techniques
- To demonstrate operations of linear and non-linear data structure
- To develop an application using suitable data structure.

COURSE OUTCOMES: After completion of the course, the student should be able to

CO-1: Understand basic concepts of data structures and analyse computation complexity.
CO-2: Apply linear data structures to implement various sorting, searching techniques.
CO-3: Solve the given problem using linear data structures.
CO-4: Execute the given problem using non-linear data structures.
CO-5: Analyze appropriate and efficient data structure to implement a given problem.

UNIT-I:

Introduction to Data Structures: Abstract Data Types (ADT), Asymptotic Notations. Time-Space trade off. Searching: Linear Search and Binary Search Techniques and their time complexities.

Linear Data Structures: Stacks - ADT Stack and its operations: Applications of Stacks: Recursion, Expression Conversion, and evaluation.

UNIT-II:

Linear Data Structures: Queues - ADT queue, Types of Queue: Linear Queue, CircularQueue, Double ended queue, operations on each types of Queues

Linked Lists: Singly linked lists: Representation in memory, Operations: Traversing, Searching, insertion, Deletion from linked list; Linked representation of Stack and Queue. Doubly linked List, Circular Linked Lists: All operations

UNIT-III:

Trees: Basic Tree Terminologies, Different types of Trees: Binary Tree, Binary Search Tree, AVL Tree; Tree Operations on each of the trees and their algorithms with time complexities.

B-Trees: Definition, Operations.

UNIT-IV:

Priority Queue: Definition, Operations and their time complexities.

Sorting: Objective and properties of different sorting algorithms: Quick Sort, Heap Sort, Merge Sort; Radix sort

UNIT-V:

Dictionaries: Definition, ADT, Linear List representation, operations- insertion, deletion and searching, Hash Table representation, Hash function-Division Method, Collision Resolution Techniques-Separate Chaining, open addressing-linear probing, quadratic probing, double hashing, Rehashing.

Graphs: Graph terminology –Representation of graphs –Graph Traversal: BFS (breadth first search) –DFS (depth first search) –Minimum Spanning Tree.

TEXT BOOKS:

1. Fundamental of Data Structure, Horowitz and Sahani, Galgotia Publication
2. Data Structure, Lipschutz, Schaum Series

REFERENCES:

1. Algorithms, Data Structures, and Problem Solving with C++, Illustrated Edition byMark Allen Weiss, Addison-Wesley Publishing Company
2. How to Solve it by Computer, 2nd Impression by R.G. Dromey, Pearson Education

UNIT-I

1.1. Abstract data type (ADT):

- Abstract Data type is the way we look at a data structure, focusing on what it does and ignoring how it does its job.
- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- The following figure gives the visual representation of abstract data type.

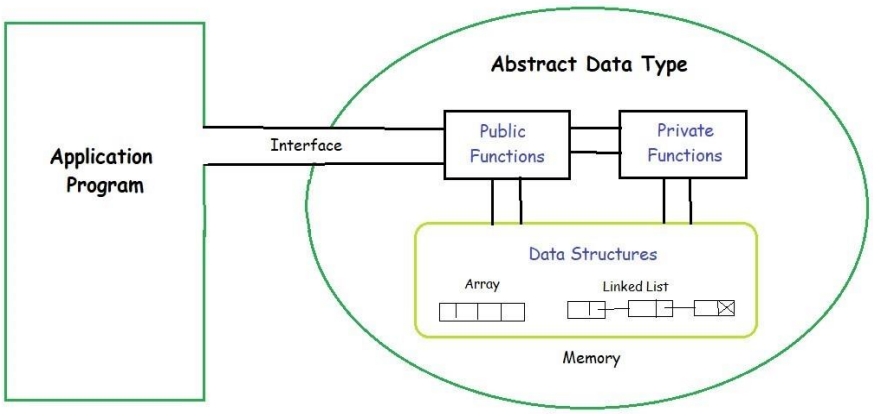


Fig 1: Abstract data type

List ADT:

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

STACK ADT:

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false
- isFull() – Return true if the stack is full, otherwise return false.

QUEUE ADT:

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

1.2. Asymptotic notations:

Asymptotic Notations are the expressions that are used to represent the complexity of an algorithm.

There are three types of analysis that we perform on a particular algorithm. They are

1. Best Case
2. Worst Case
3. Average Case

Best Case: Minimum time or space required for program execution.

Worst Case: Maximum time or space required for program execution.

Average Case: Average time or space required for program execution.

Asymptotic notations are three types. Which are

1. Big- O Notation (O)
2. Omega Notation (Ω)
3. Theta Notation (θ)

Big-O Notation (O) : Big-O notation specifically describes worst-case scenario. It represents the upper bound running time complexity of an algorithm.

Definition: **f(n) and g(n) are two functions and f(n) = O(g(n)) iff there exist two positive constants c and n_0 such that $f(n) \leq c.g(n)$ for all $n \geq n_0$.**

The following example explains how we represent the time and space complexity using **Big O notation**.

O(1) – Constant time

It represents the complexity of an algorithm that always executes at the same time or space regardless of the input data.

Example: The following step will always execute at the same time (or space) regardless of the size of the input data.

```
num = a[5];
```

O(n) – Linear time

It represents the complexity of an algorithm, whose performance will grow linearly (in direct proportion) to the size of the input data.

Example: The execution time will depend on the size of the array. When the size of the array increases, the execution time will also increase in the same proportion (linearly).

Traversing an array

```
for(i=0 ;i<=n; i++){   }
```

$O(n^2)$

It represents the complexity of an algorithm, whose performance is directly proportional to the square of the size of the input data.

Example: Traversing 2D array

```
for(i=1 ;i<=n; i++)
{
        for(j=1; j<=n;j++)
        {
                ..........
        }
}
```

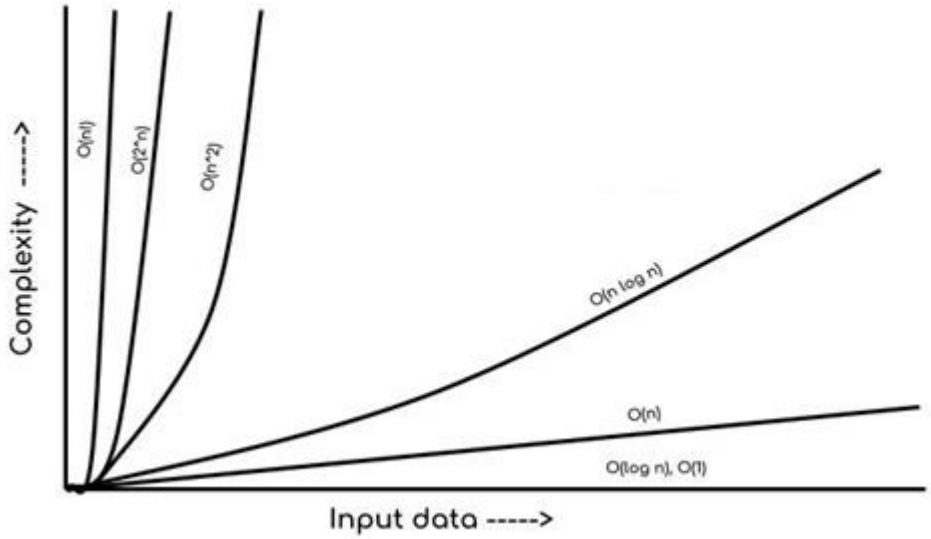
Other examples: Bubble sort, insertion sort, and selection sort algorithms

Similarly, there are other Big O notations such as:

Logarithmic growth O(log n), log-linear growth O(n log n), exponential growth O(2^n), and factorial growth O(n!).

Relationship between the complexities:

O(1) < O(log n) < O (n) < O(n log n) < O(n^2) < O (n^3)< O(2^n) < O(n!)



Omega Notation (Ω)

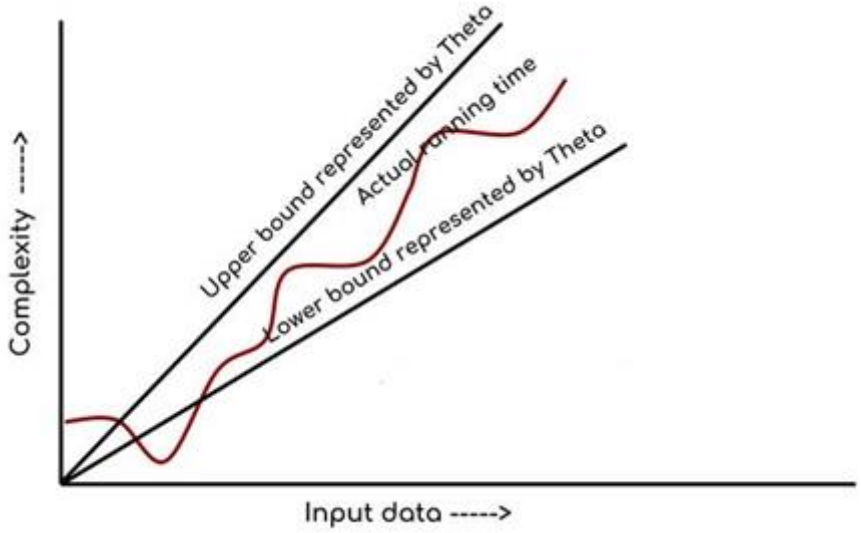
Omega notation specifically describes the best-case scenario. It represents the lower bound running time complexity of an algorithm. So if we represent a complexity of an algorithm in Omega notation, it means that the **algorithm cannot be completed in less time than this**, it would at least take the time represented by Omega notation or it can take more (when not in the best case scenario).

Definition: **f(n) and g(n) are two functions and f(n) = Ω (g(n)) iff there exist two positive constants c and n_0 such that *f*(n) ≥ c.*g*(n) for all n ≥ n_0**

Theta Notation (θ)

This notation describes both the upper bound and lower bound of an algorithm so we can say that it defines exact asymptotic behavior. In the real-case scenario the algorithm does not always run on best and worst cases, the average running time lies between best and worst and can be represented by the theta notation.

Definition: **f(n) and g(n) are two functions and f(n) = Ω (g(n)) iff there exist three positive constants c1, c2 and n_0 such that c1.g(n) ≤ *f*(n) ≤ c2.*g*(n) for all n ≥ n_0**



1.3. Time and Space Tradeoff

A trade-off is a situation where one thing increases, and another thing decreases.

Definition: A time-space or space-time trade-off is a way of solving a problem in:

- less time by using more memory, or
- less space by spending more time.

The best Algorithm is that which helps to solve a problem that requires less space in memory and also takes less time to generate the output. But in general, it is not always possible to achieve both of these conditions at the same time.

There are 4 types of time-Space trade-off

- Compressed or Uncompressed data
- Re Rendering or Stored images
- Smaller code or loop unrolling
- Lookup tables or Recalculation

Compressed vs Uncompressed:

Time-space trade-off plays a role in data storage also. If stored data is uncompressed, it required more space but less access time. If stored data is compressed, it requires less space but more time.

Re Rendering or Stored images

Storing only the source and rendering it as an image every time the page is requested would be trading time for space: more time used, but less space. Storing the image would be trading space for time: more space, but less time.

Smaller code or loop unrolling

Smaller code with loops occupies less space but requires more computation time (for jumping back to the beginning of the loop after each iteration).
Larger codes require more space but less computation time.

Lookup table vs Recalculation

Lookup table: An implementation can include the entire table, which reduces computing time, but increases the amount of memory needed.
Re-calculation: An implementation can compute table entries as needed increasing computing time, but reducing memory requirements.

1.4. Searching

Searching is the processing of finding a particular element is present in the list or not. There are two types of searching techniques.

1. Linear search

2. Binary search.

1.4.1. Linear Search

Linear search technique is also known as sequential search technique. The linear search is a method of searching an element in a list in sequence. In this method, the array is searched for the required element from the beginning of the list/array or from the last element to first element of array and continues until the item is found or the entire list/array has been searched.

Algorithm

Step 1: set-up a flag to indicate “element not found”.

Step 2: Take the first element in the list.

Step 3: If the element in the list is equal to the desired element.

- Set flag to “element found.”
- Display the message “element found in the list.”
- Go to step 6

Step 4: If it is not the end of list,

- Take the next element in the list
- Go to step 3

Step 5: If the flag is “element not found”

Display the message “element not found”

Step 6: End of the Algorithm

Program

```
/*
        Linear search
*/
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10],i,key,flag=0,n;
    clrscr();
    printf("\n Array length(No of elements)\n");
    scanf("%d",&n);
    printf("\n ENter array elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("\n Enter key\n");
    scanf("%d",&key);
    for(i=0;i<n;i++)
    {
        if(key == a[i])
        {
            flag=1;
            break;
        }
    }
    if(flag == 1)
    {
        printf("\n Search is successful\n");
    }
    else
    {
        printf("\n Search is unsuccessful\n");
    }
    getch();
}
```

Advantages

1. It is a simple and conventional method of searching for data. The linear or sequential name implies that the items are stored in a systematic manner.
2. The elements in the list can be in any order. i.e. The linear search can be applied on sorted or unsorted linear data structure.

Disadvantages

1. This method is insufficient when a large number of elements is present in list.
2. It consumes more time and reduces the retrieval rate of the system.

Time complexities:

1. Best case – O(1) – if the key element is at the first position
2. Average case – O(n) – if the key element is in the middle of the array
3. Worst case - O(n) – if the key element is the at the end of the array.

1.4.2. Binary Search

Binary search is quicker than the linear search. However, it cannot be applied on unsorted data structure. The binary search is based on the approach divide-and-conquer. The binary search starts by testing the data in the middle element of the array. This determines target is whether in the first half or second half. If target is in first half, we do not need to check the second half and if it is in second half no need to check in first half. Similarly, we repeat this process until we find target in the list or not found from the list. Here we need 3 variables to identify first, last and middle elements.

To implement binary search method, the elements must be in sorted order. Search is performed as follows:

1. The key is compared with an item in the middle position of an array.
2. If the key matches with the item, return it and stop.
3. If the key is less than mid positioned item, then the item to be found must be in the first half of array, otherwise it must be in second half of array.
4. Repeat the procedure for lower (or upper half) of array until the element is found.

Algorithm:

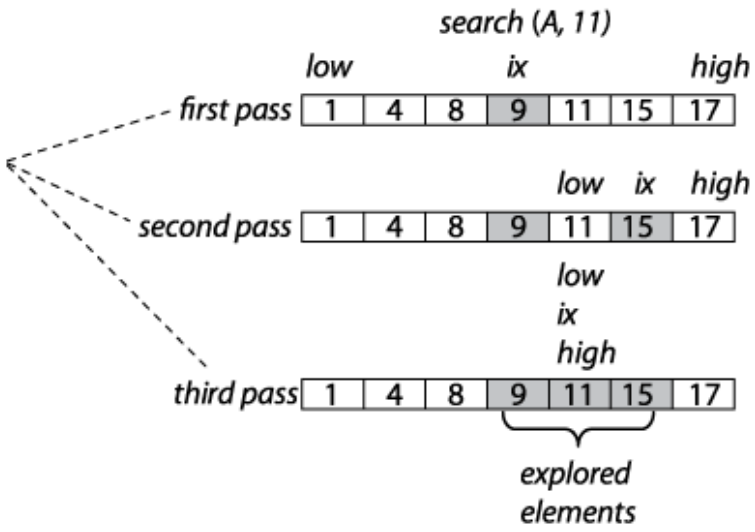
BINARY SEARCH

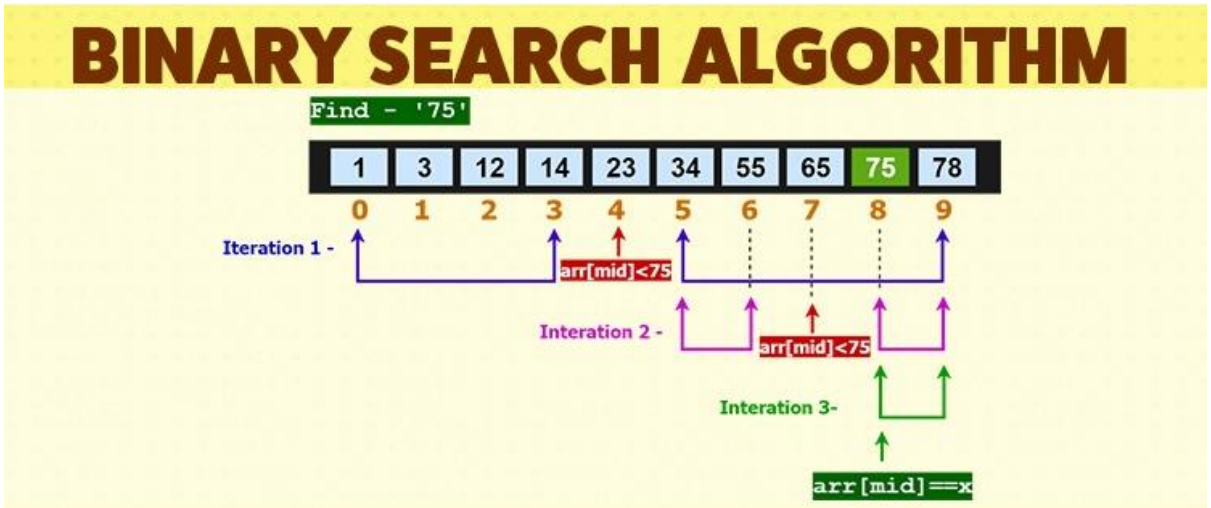
Best	Average	Worst
O (1)	O (log n)	O (log n)

Array

Divide and Conquer

```
search (A, t)
1.    low = 0
2.    high = n –1
3.    while (low ≤ high) do
4.      ix = (low + high)/2
5.      if (t = A[ix]) then
6.          return true
7.      else if (t < A[ix]) then
8.          high = ix – 1
9.      else low = ix + 1
10.   return false
end
```





Program:

```
/*
            Binary search
*/
#include<stdio.h>
#include<conio.h>
void main()
```

```
{
	int a[10],low,i,high,key,flag=0,n,mid;
	clrscr();
	printf("\n Array length(No of elements)\n");
	scanf("%d",&n);
	printf("\n Enter array elements\n");
	for(i=0;i<n;i++)
	{
		scanf("%d",&a[i]);
	}
	printf("\n Enter key\n");
	scanf("%d",&key);
	low=0;
	high=n-1;

	while(low<=high)
	{
		mid = (low+high)/2;
		if(key == a[mid])
		{
			flag=1;
			break;
		}
		else if(key < a[mid])
		{
			high = mid-1;
		}
		else
		{
			low = mid+1;
		}
	}
	if(flag==1)
	{
		printf("\n Search is successful\n");
	}
	else
	{
		printf("\n Search is unsuccessful\n");
	}
	getch();
}
```

Advantages

1. It takes less time complexity (O(logn))

2. it is efficient when the data set is large.

Disadvantages

1. The elements must be in either ascending or descending order.

Time complexities

1. Best case – O(1) – if the key element is in the middle position of an array.
2. Average case – O(logn)
3. Worst case – O(logn)

1.5. Data Structures

- A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.
- **Definition:** it is a particular way of organizing and storing data in a computer so that it can be used efficiently.
- Examples: Arrays, linked lists, queues, stacks, binary trees, and hash tables.
- Data structures are applied in the following areas:

• Compiler Design	• Operating System
• Data base Management Systems	• Statistical Analysis
• Simulation	• Artificial Intelligence
• Graphics	

1.5.1. **Classification of Data Structures**

- Basically, data strictures are divided into two categories, which are Primitive data structures and non-primitive data structures.
- Primitive data structures are also called as simple data types. The primitive data structures are integer, real, character and bool.
- Based on the structure and arrangement of data, non-primitive data structures are further divided into linear and non-linear data structure.
- **Linear data structure:** A data structure is said to be linear if its elements form a linear or sequential list.
- In linear data structures, the data is arranged in a linear fashion although the way they are stored in the memory need not be sequential.

Example: Arrays, stacks, queues and linked lists

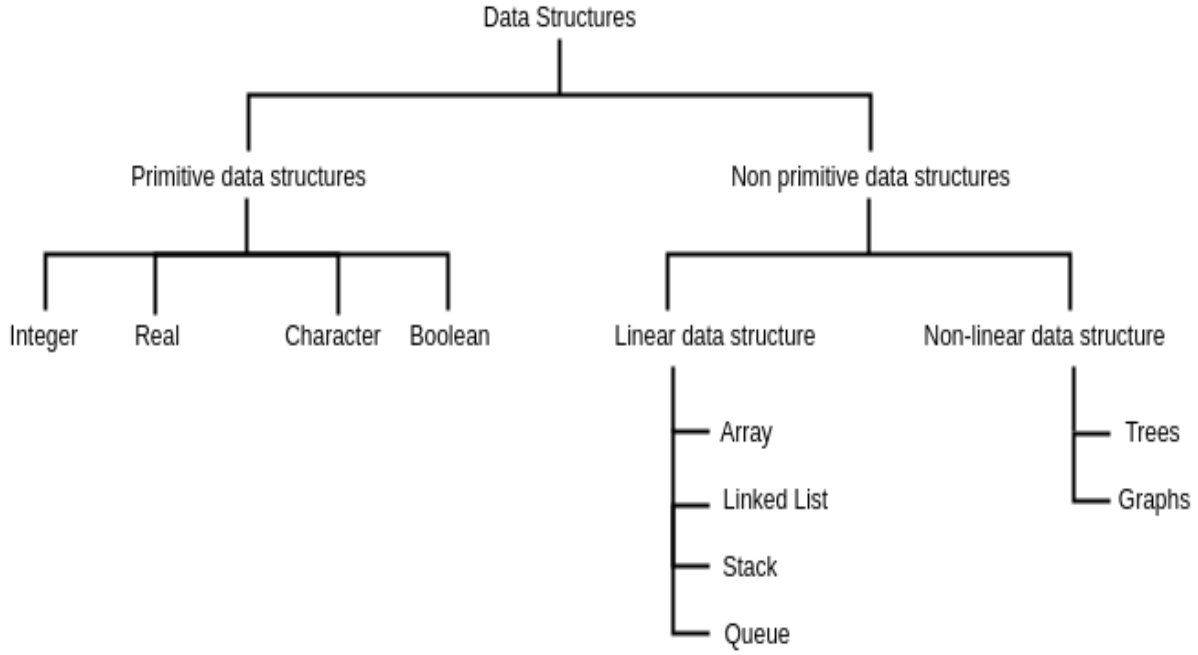


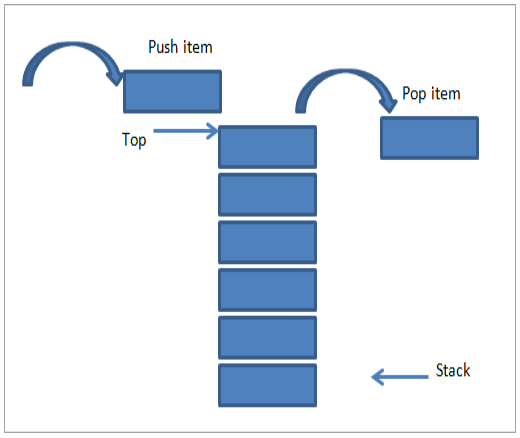
Fig. Classification of data structures.

- **Non-linear data structures:** A data structure is to be non-linear if the data is not arranged in sequential.
- In non-linear data structures, insertion and deletion of data is therefore not possible in a linear fashion.
- Examples: Trees and Graphs.

Array: An array is a group of similar data elements.

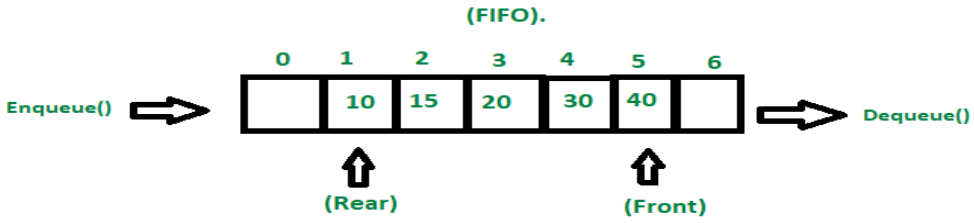
Stack: A stack is a linear data structures, in which insertions and deletions are performed only from one end is called top of the stack.

In stack the elements which is inserted last is to be removed first. Hence it is called Last-in-first-out (LIFO) data structure.

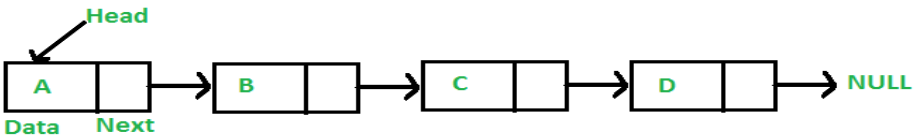


Queue: A queue is a linear data structure in which insertions are performed at rear end and deletions are performed at front end.

In queue, the first inserted element is to be removed first. Hence it is called First-In-First-Out (FIFO) data structure.



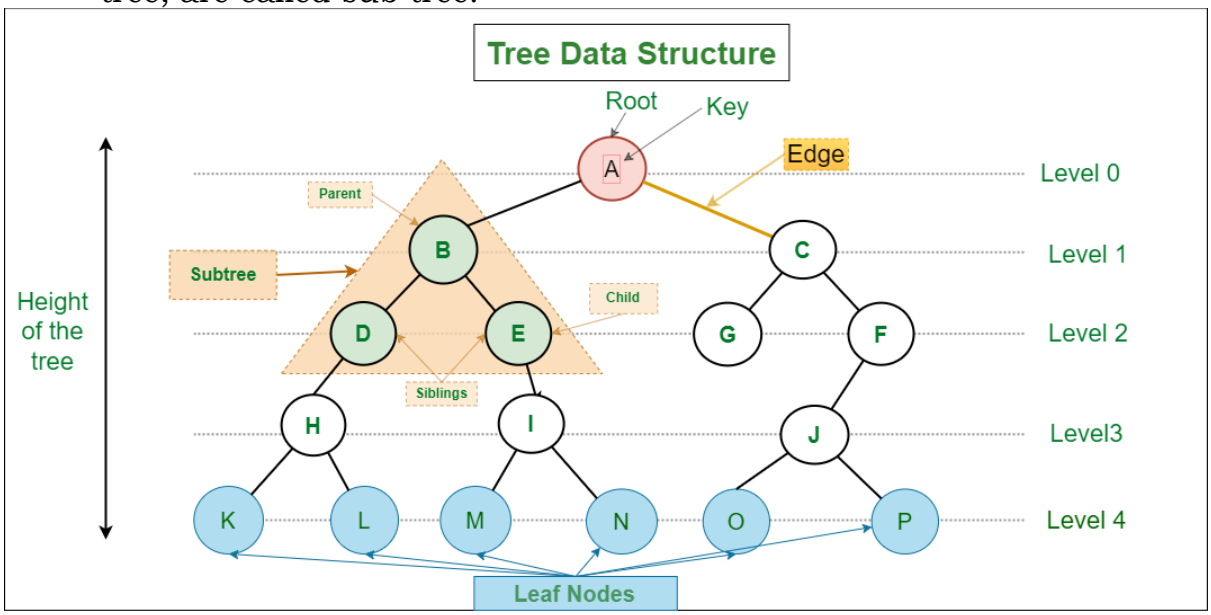
Linked List: A linked list is a linear data structure; it consists of nodes where each node contains a data field and a reference(link) to the next node in the list.



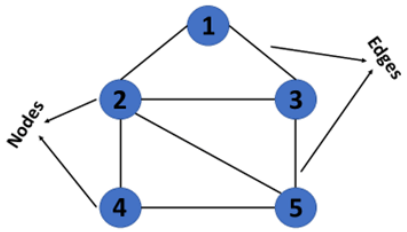
Note: Non-linear data structures follows hierarchy, i.e. the elements are arranged in multiple levels.

Tree: A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that :

1. There is a special node called the root of the tree.
2. Removing nodes (or data item) are partitioned into number of mutually exclusive (i.e., disjoined) subsets each of which is itself a tree, are called sub tree.



Graph: A graph is a structure made of two components: a set of vertices V, and a set of edges E. Therefore, a graph is G = (V, E), where G is a graph.



1.5.2. Data Structure Operations

1. Traversing: It means to access each data item exactly once so that it can be processed
2. Searching: It is used to find the location of one or more data items that satisfy the given constraint.
3. Inserting: It is used to add new data items to the given list of data items.
4. Deleting: It means to remove (delete) a particular data item from the given collection of data items.
5. Sorting: Data items can be arranged in some order like ascending order or descending order depending on the type of application.
6. Merging: Lists of two sorted data items can be combined to form a single list of sorted data items.

1.6. Stack:

Definition: A stack is a linear data structure in which insertions and deletions are performed at only one end called the top of the stack.

The last added element will be first removed from the stack. Hence it is also called Last-in-First-out (LIFO) data structure.

Note:

1. The most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack.

The basic operations on stack are 1. Push 2. Pop

1. Push: Inserting an element into the stack is called push.

2. Pop: Removing an element from the stack is called pop.

PRIMITIVE STACK OPERATIONS: PUSH AND POP

The primitive operations performed on the stack are as follows:

PUSH: The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the

new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

Algorithm:

Step 1: IF TOP = MAX-1 PRINT OVERFLOW [END OF IF]

Step 2: SET TOP = TOP+1

Step 3: SET STACK[TOP] = VALUE

Step 4: END

POP: The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed, then the stack underflow condition occurs.

Algorithm:

Step 1: IF TOP = NULL PRINT UNDERFLOW [END OF IF]

Step 2: SET VAL = STACK[TOP]

Step 3: SET TOP = TOP-1

Step 4: END

1.6.1. Stack ADT

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false
- isFull() - Return true if the stack is full, otherwise return false.

1.6.2. Implementation of Stack

There are two ways to implement a stack.

1. Array representation

2. Linked list representation. (will be discussed in UNIT-2)

C program to implement stack using arrays

```
#include<stdio.h>
#include<ctype.h>
#define SIZE 5
int s[100];
int top=-1;

void push(int x)
{
        if(top==SIZE-1)
        {
                printf("\n Overflow");
        }
        else
        {
                top++;
                s[top]=x;
        }
}

void pop()
{
        if(top==-1)
        {
                printf("\n Underflow");
        }
        else
        {
                printf("\n Deleted item is %d",s[top]);
                top--;
        }
}

void peek()
{
        if(top==-1)
        {
                printf("\n Underflow");
        }
        else
        {
                printf("\n top of the stack is %d",s[top]);
        }
}

void display()
{
        int i;
```

```
        if(top==-1)
        {
                printf("\n Stack is empty");
        }
        else
        {
                for(i=top;i>=0;i--)
                {
                        printf("\n %d",s[i]);
                }
        }
}

void main()
{
        int ch,x;
        while(1)
        {
                printf("\n 1. Push \n 2. Pop \n 3. Peek \n 4. Display \n 5. Exit\n");
                printf("\n Enter your choice:");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: printf("\n Enter the item to be inserted into the stack:");
                                        scanf("%d",&x);
                                        push(x);
                                        break;
                        case 2: pop();
                                        break;
                        case 3: peek();
                                        break;
                        case 4: display();
                                        break;
                        case 5: exit(0);
                        default : printf("\n Enter correct choice");
                }
        }
}
```

1.6.3. Applications of Stack:

Stack has mainly three applications:

1. Converting infix expression into postfix expression.

2. Evaluating postfix expression

3. Recursion.

Expression: An expression is defined as the number of operands or data items combined with several operators.

There are basically three types of notation for an expression (mathematical expression.

1. Infix expression

2. Postfix expression

3. Prefix expression

Infix expression: The operator is placed between the operands is called infix expression.

Example: 2+3, a+b*c, 2+5/2*3-4^5, etc…

Postfix expression: The operator is placed after the operands is called postfix expression. it is also known as suffix notation or reverse polish notation.

Example: 23+, abc*+, 23+5*, etc…

Prefix expression: The operator is placed before the operands is called prefix expression. it is also called polish notation in the honour of the polish mathematician Jan Lukasiewicz who developed this notation.

Example: +23, +a*bc, *+235, etc…

Note: postfix notation is most suitable for a computer to calculate any expression (due to its reverse characteristic) and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor).

Advantages of Postfix notation: Using infix notation, we cannot tell the order in which operators should be applied. But using postfix expression operands appear before the operator, so there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator. Place the result back onto the stack.

Operator precedence

1. Exponential operator ^ Highest precedence
2. Multiplication/Division *, / Next precedence
3. Addition/Subtraction +, - Least precedence

Converting infix expression to postfix expression.

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.

2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Give postfix form for **A + [(B + C) + (D + E) * F] / G**

Solution. Evaluation order is

A + { [(BC +) + (DE +) * F] / G}

A + { [(BC +) + (DE + F *] / G}

A + { [(BC + (DE + F * +] / G}

A + [BC + DE + F *+ G /]

ABC + DE + F * + G / + Postfix Form

Algorithm

Step 1: Add ")"to the end of the infix expression

Step 2: Push "(" on to the stack

Step 3: Repeat until each character in the infix notation is scanned

a. IF a "(" is encountered, push it on the stack
b. IF an operand is encountered, add it postfix expression.
c. IF a ")" is encountered, then
    i. Repeatedly pop from stack and add it to the postfix expression until a "(" is encountered.
    ii. Discard the "(". That is, remove the "(" from stack and do not add it to the postfix expression
d. IF an operator is encountered, then
    i. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than
    ii. Push the operator to the stack

[END OF IF]

Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty.

Step 5: EXIT

Example: **A – (B / C + (D % E * F) / G)* H**

Infix Character	Scanned Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (*	A B C / D E%
F	(– (+ (*	A B C / D E %F
)	(– (+	A B C / D E %F *
/	(– (+ /	A B C / D E %F *
G	(– (+ /	A B C / D E %F * G
)	(–	A B C / D E %F * G / +
*	(– *	A B C / D E %F * G / +
H	(– *	A B C / D E% F * G / + H
)		A B C / D E %F * G / + H * –

C Program to convert infix expression to postfix expression

```
#include<stdio.h>
#include<ctype.h>
#define SIZE 50
char stack[100];
int top=-1;
char infix[100];
char postfix[100];

void push(char elem)
{
        stack[++top]=elem;
}
```

```
char pop()
{
        return stack[top--];
}
int pr(char symbol)
{
        if(symbol=='(')
                return 1;
        else if(symbol=='+'||symbol=='-')
                return 2;
        else if(symbol=='*'||symbol=='/'||symbol=='%')
                return 3;
        else if(symbol=='^')
                return 4;
}
void main()
{
        char ch,elem;
        int i=0,k=0;
        printf("Type infix expression: \n");
        scanf("%s",infix);
        push('(');
        while(infix[i]!='\0')
        {
                ch=infix[i];
                if(ch=='(')
                        push(ch);
                else if(isalnum(ch))
                {
                        postfix[k++]=ch;
```

```
            }
            else if(ch==')')
            {
                    while(stack[top]!='(')
                    {
                            postfix[k++]=pop();
                    }
                    elem=pop();
            }
            else
            {
                    while(pr(stack[top])>=pr(ch))
                    {
                            postfix[k++]=pop();
                    }
                    push(ch);
            }
            i++;
    }
    while(stack[top]!='(')
    {
            postfix[k++]=pop();
    }
    postfix[k]='\0';
    printf("Postfix expression: %s",postfix);
}
```

Evaluating postfix expression

Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped from the

stack and the operator is applied on these values. The result is then pushed on to the stack.

Algorithm

Step 1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]

Step 2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.

Step 3. If an operand is encountered, put it on STACK.

Step 4. If an operator @ is encountered, then:

(a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.

(b) Evaluate B @ A.

(c) Place the result on to the STACK.

Step 5. Result equal to the top element on STACK.

Step 6. Exit

Example: 9 3 4 * 8 + 4 / –

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
–	4

Program to evaluate postfix expression.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
int s[100];
int top=-1;
```

```
void push(int x)
{
        s[++top]=x;
}
int pop()
{
        return s[top--];
}
int main(int argc, char *argv[]) {
        char p[20];
        char ch;
        int i,num,a,b,res;
        printf("\n Enter postfix expression");
        scanf("%s",p);
        i=0;
        while(p[i]!='\0')
        {
                ch=p[i];
                if(isdigit(p[i]))
                {
                        num = p[i]-48;
                        push(num);
                }
                else
                {
                        a=pop();
                        b=pop();
                        switch(ch)
                        {
                                case '+': res = b+a;
```

```
                                        push(res);
                                        break;
                        case '-': res = b-a;
                                        push(res);
                                        break;
                        case '*': res = b*a;
                                        push(res);
                                        break;
                        case '/': res = b/a;
                                        push(res);
                                        break;
                        case '^': res = pow(b,a);
                                        push(res);
                                        break;
                    }

                }
                i++;
        }
        printf("\n Result is %d",s[top]);
        return 0;
}
```

Recursion

Definition: A recursion is defined as a function that calls itself.

Every recursive solution has two major cases. They are

1. Base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.
2. Recursive case, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Types of Recursions

1. Direct recursion
2. Indirect recursion
3. Tail recursion

1. **Direct recursion:** A function is said to be directly recursive if it explicitly calls itself.
   Example:

```
int Func (int n)
{       if (n == 0)
                return n;
        else
                return (Func (n–1));
}
```

2. **Indirect Recursion:** A function is said to be indirectly recursive if it contains a call to another function which ultimately calls it.
   Example:

```
int Funcl (int n)
{
        if (n == 0)
                return n;
        else
                return Func2(n);
}
int Func2(int x)
{
        return Func1(x–1);
}
```

3. **Tail Recursion:** A recursive function is said to be tail recursive if no operations are pending to be performed when the recursive function returns to its caller.
   Example:

```
int Fact(n)
{
        return Fact1(n, 1);
}
int Fact1(int n, int res)
{
        if (n == 1)
                return res;
        else
```

```
        return Fact1(n–1, n*res);
    }
```

Disadvantages of Recursion

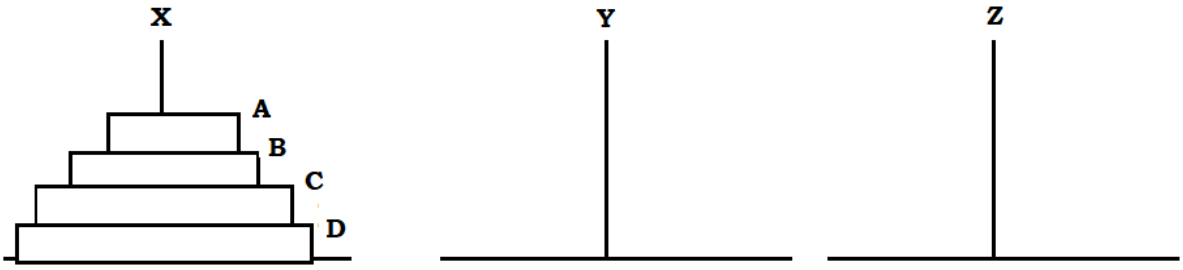
1. It consumes more storage space because the recursive calls along with automatic variables are stored on the stack.
2. The computer may run out of memory if the recursive calls are not checked.
3. It is not more efficient in terms of speed and execution time.
4. If proper precautions are not taken, recursion may result in non-terminating iterations.
5. Recursion is not advocated when the problem can be through iteration. Recursion may be treated as a software tool to be applied carefully and selectively.

One of the recursive problems is towers of Hanoi

Towers of Hanoi

Now let us look at the Tower of Hanoi problem and see how we can use recursive technique to produce a logical and elegant solution.

The initial setup of the problem is shown below. Here three pegs (or towers) X, Y and Z exists. There will be four different sized disks, say A, B, C and D. Each disk has a hole in the center so that it can be stacked on any of the pegs. At the beginning, the disks are stacked on the X peg, that is the largest sized disk on the bottom and the smallest sized disk on top as shown in Fig. below.



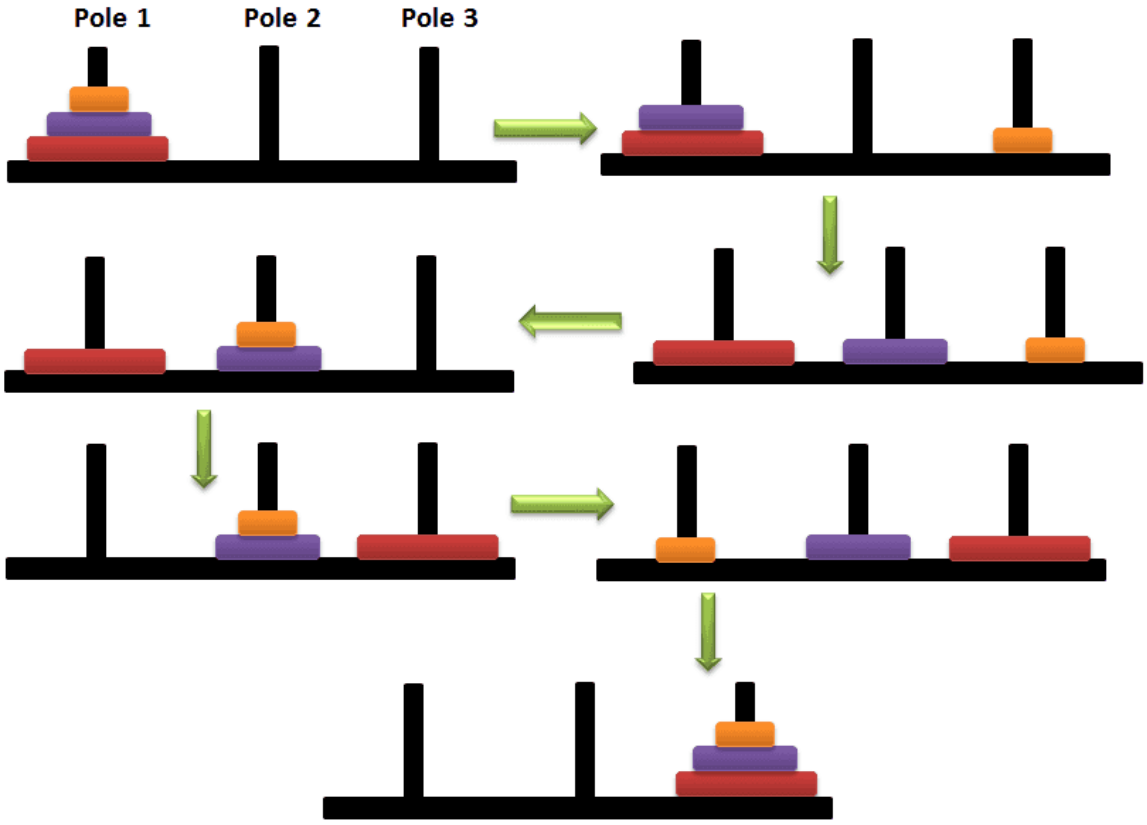
Initial configuration of Towers of Hanoi

Here we have to transfer all the disks from source peg X to the destination peg Z by using an intermediate peg Y. Following are the rules to be followed during transfer:

1. Transferring the disks from the source peg to the destination peg such that at any point of transformation no large size disk is placed on the smaller one.
2. Only one disk may be moved at a time.

3. Each disk must be stacked on any one of the pegs.

Example: Towers of Hanoi Solution



Algorithm

- Base case: if n=1
    - Move the ring from A to C using B as spare
- Recursive case:
    - Move n – 1 rings from A to B using C as spare
    - Move the one ring left on A to C using B as spare
    - Move n – 1 rings from B to C using A as spare

Towers of Hanoi Program

```
#include <stdio.h>
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
          if (n == 1)
           {
              printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
           }
        else
        {
           towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
           printf("\n Move disk %d from rod %c to rod %c", n, from_rod,
        to_rod);
           towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
        }
}
int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
    return 0;
}
```

Important Question:

Short Answer questions

1. Define time and space complexity.
2. What are the asymptotic notations.
3. Define BigO, Omega and Theta notations.
4. What is time-space trade-off?
5. Explain types of time-space trade-off.
6. What are the advantages and disadvantages of linear and Binary search.
7. What are the time complexities of linear and binary search.
8. Define stack.
9. Define ADT
10. Give stack ADT.
11. What are the applications of stack.
12. What is recursion? Explain its disadvantages.
13. explain recursion types.
14. What is infix, prefix and postfix expression.

Long answer questions

1. Define data structure. Explain different types of data structure.
1. Explain Asymptotic notations.
2. Explain types of time space trade-off.
3. Explain linear search with an example.
4. Explain binary search with an example.
5. Write a c program to implement stack using arrays.
6. Convert the given infix expression A+B^C+(D*E/F)*G into its postfix expression, and evaluate the same using stack. Here A=3, B=5, C=2, D=7, E=4, F=1, G=8.
7. Explain the procedure to evaluate postfix expression. Evaluate the following postfix expression 7 3 4 + - 2 4 5 / + * 6 / 7 + ?.
8. Write a c program to implement towers of Hanoi.

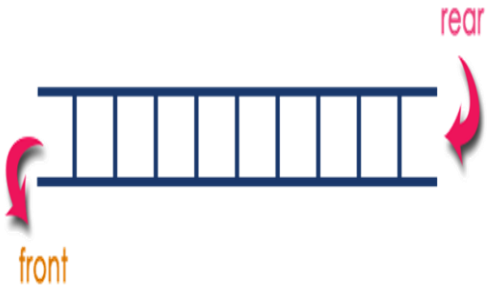
9. Write an algorithm / program to convert infix expression to postfix expression.
10. Write an algorithm/program to evaluate postfix expression.
11. Convert following expression X+(Y * Z) – ((N * M +O) /P) in to post form.
12. Explain the evaluation of prefix expression. Find the equivalent prefix of 8 6 3 + * 1 2 3 -/-
13. Convert following infix expression into postfix expression: A+B^(C+D)-E*F+G.
14. Transform the expression into its equivalent post fix expression: A*(B+D)/E-F*(G+H/K).

UNIT - II

QUEUE

Definition: A Queue is a linear data structure in which insertion operations are performed at rear end and deletion operations are performed front end.

In queue data structure, the insertion and deletion operations are performed based on FIFO (First In First Out) principle. Which means that the element inserted first is to be removed first. Hence it is called First in First Out(FIFO) data structure.



Uses of Queue:

1. Railway reservation
2. Cinema ticket booking
3. Operating systems

QUEUE ADT

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

Types of Queues

1. Simple queue
2. Circular queue
3. Double ended queue

Simple Queue

Basically, queue has two operations. 1. Enqueue 2. Dequeue

1. **Enqueue:** Inserting an element into the queue is called enqueue.

2. **Dequeue:** Deleting / removing an element from the queue is called dequeue.

Enqueue

- Step 1 - Check whether queue is FULL. (rear == SIZE-1)
- Step 2 - If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
- Step 3 - If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

Dequeue

- Step 1 - Check whether queue is EMPTY. (front = rear=-1)
- Step 2 - If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
- Step 3 - If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

Implementing queue using arrays

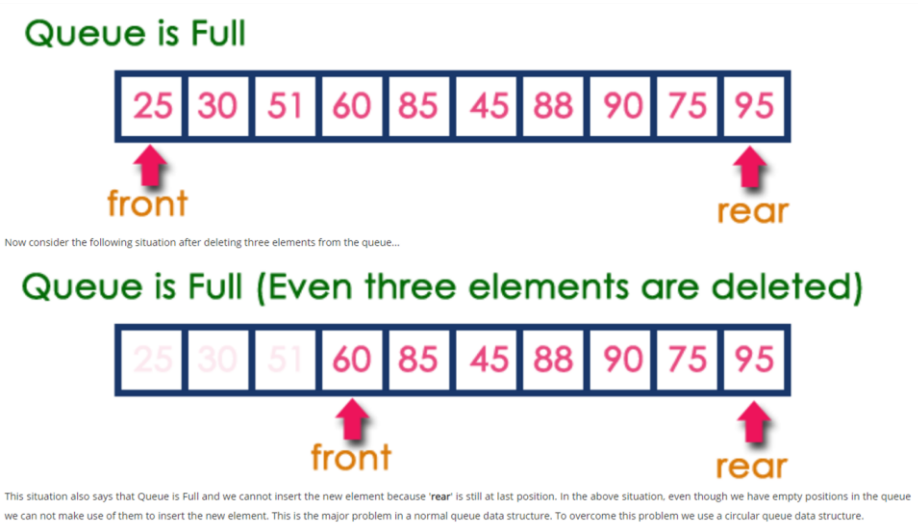
```
#include <stdio.h>
#include <stdlib.h>
#define size 5
int Q[size];
int f=-1,r=-1;

void enqueue(int x)
{
        if(r==size-1)
        {
                printf("\n overflow");
        }
        else if(r==-1&&f==-1)
        {
                r = r+1;
                f=f+1;
                Q[r]=x;
        }
        else
        {
```

```
                r=r+1;
                Q[r]=x;
        }

}

void dequeue()
{

        if(f==-1)
        {
                printf("underflow\n");
        }
        else if(f==r)
        {
                printf("\n The deleted element is %d",Q[f]);
                r=-1;
                f=-1;
        }
        else
        {
                printf("\n The deleted element is %d",Q[f]);
                f=f+1;
        }

}

void peek()
{

        if(f==-1)
        {
                printf("underflow\n");
                return 0;
        }
        else
        {
                printf("\n Peek element is %d",Q[f]);

        }

}

void display()
{
        int i;
        if(r==-1)
        {
                printf("\n Queue is empty\n");
```

```
        }
        else
        {
                for(i=f;i<=r;i++)
                {
                        printf("%d\n",Q[i]);
                }
        }
}


int main(int argc, char *argv[]) {

        int ch,x;

        while(1)
        {
                printf("\n 1. Insert");
                printf("\n 2. Delete");
                printf("\n 3. Display ");
                printf("\n 4. Peek ");
                printf("\n 5. exit");
                printf("\n Enter your choice\n");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: printf("\n insert a value\n");
                                        scanf("%d",&x);
                                        enqueue(x);
                                        break;
                        case 2: dequeue();
                                        break;
                        case 3: display();
                                        break;
                        case 4: peek();
                                        break;
                        case 5: exit(0);
                        default: printf("\n Enter correct choice\n");
                }
        }

        return 0;
}
```

Drawback of Simple queue

Let us consider the following scenario.

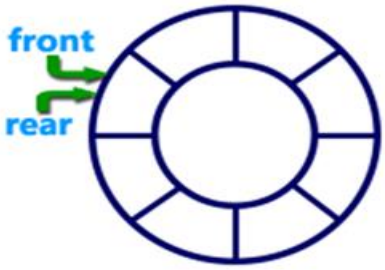


The above situation says that queue is full, and we cannot insert an element into the queue because rear is pointing to last position. In the above situation even though we have empty position in the queue, but we cannot use of them to insert the new element.
To overcome the above the problem we use circular queue.

Circular Queue

Circular queue is a linear data structure in which operations are performed based on FIFO (FIRST IN FIRST OUT) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows.



Circular Queue Insertion

Step 1: IF (REAR+1)%MAX = FRONT Write " OVERFLOW " Goto step 4

Step 2: IF FRONT = -1 and REAR = -1 then SET FRONT = REAR = 0

ELSE IF REAR = MAX - 1 and FRONT ! = 0 then SET REAR = 0

ELSE SET REAR = (REAR + 1) % MAX

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

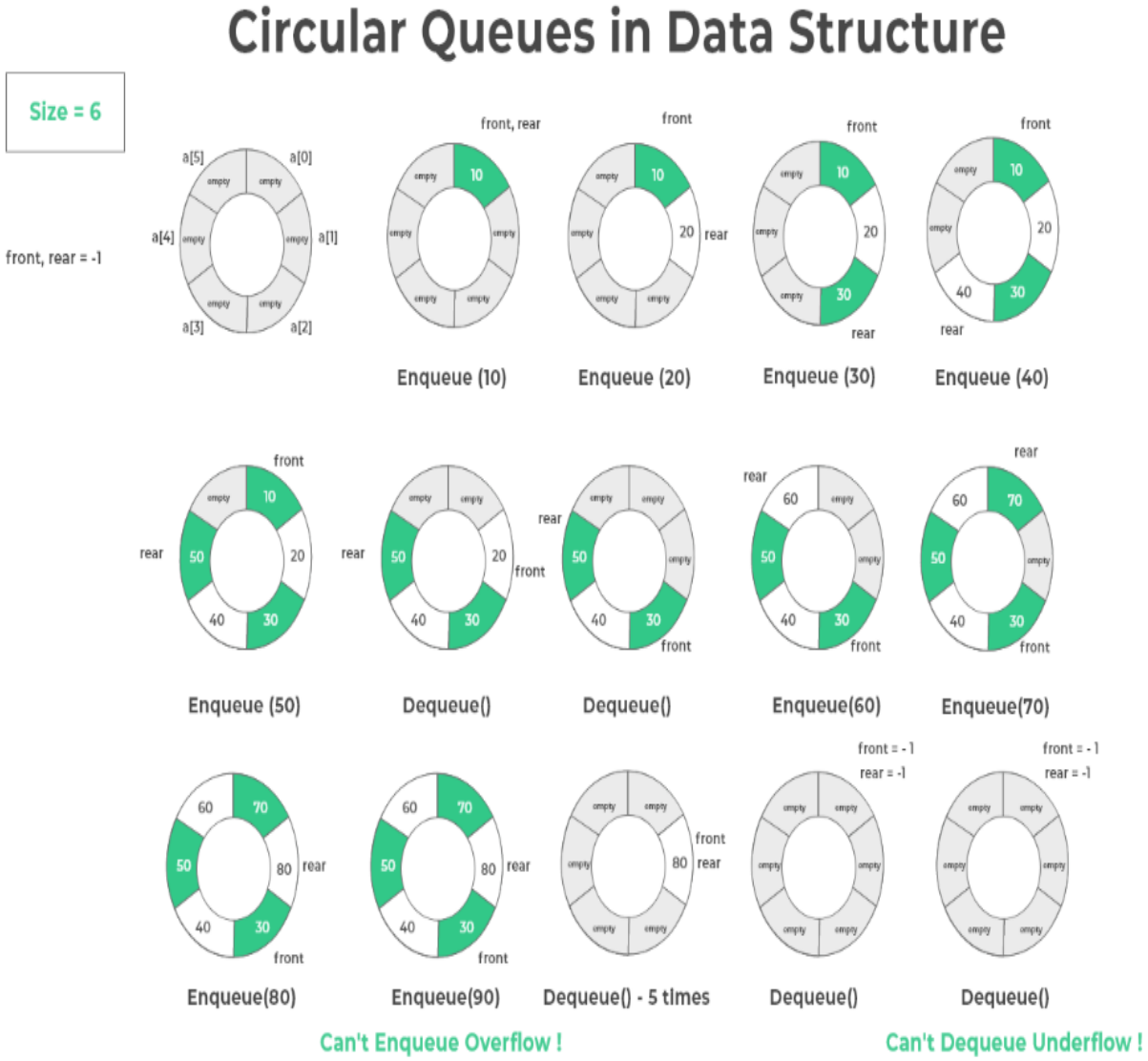
Circular Queue Deletion

- Step 1: IF FRONT = -1 Write " UNDERFLOW " Goto Step 4
- Step 2: SET VAL = QUEUE[FRONT]
- Step 3: IF FRONT = REAR then SET FRONT = REAR = -1

  ELSE IF FRONT = MAX -1 then SET FRONT = 0

  ELSE SET FRONT = FRONT + 1
- Step 4: EXIT

Circular queue Operations



Implementing Circular queue using arrays

```
#include <stdio.h>
#include <stdlib.h>
#define size 5
int CQ[size];
int front=-1;
int rear=-1;

void enqueue(int x)
{
        if((rear+1)%size== front)
        {
                printf("\n Overflow");
        }
        else if(front==-1&& rear==-1)
        {
                front =front+1;
                rear = rear +1;
                CQ[rear]= x;
        }
        else
        {
                rear = (rear + 1)%size;
                CQ[rear] = x;
        }

}

int dequeue( )
{
        int x;
        if(front==-1 && rear == -1)
        {
                printf("\n Underflow \n");
                return 0;
        }
        else if(front == rear)
        {
                x = CQ[front];
                front =-1;
                rear = -1;
        }
        else
        {
                x = CQ[front];
                front = (front+1)%size;

        }
```

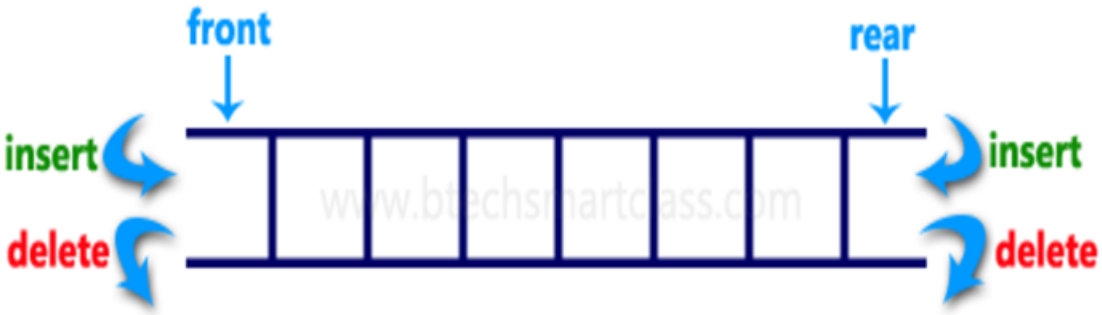
```
        return x;
}
void display()
{
        int i;

        if(front ==-1 && rear == -1)
        {
                printf("\n Queue is empty\n");
        }
        else
        {

                        for(i=front; i!=rear;i=(i+1)%size)
                        {
                                printf("%d\n",CQ[i]);
                        }
                        printf("%d\n",CQ[i]);
        }

}

int main(int argc, char *argv[]) {
        int ch,x;
        while(1)
        {

                printf("\n 1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter
your choice\n");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: printf("\n Enter the number\n");
                                        scanf("%d",&x);
                                        enqueue(x);
                                        break;
                        case 2: x = dequeue();
                                        printf("\n Deleted element is %d\n",x);
                                        break;
                        case 3: display();
                                        break;
                        case 4: exit(0);
                        default: printf("\n Enter correct choice\n");
                }
        }
        return 0;
}
```

The drawback of circular queue is we cannot insert and delete the elements from both the ends. It is overcome with the deque (double ended queue).

Double ended queue (Deque)

Definition: Double ended queue is also a queue data structure, in which deletion and insertions are performed at the both the ends (front and rear).

Graphical representation of Deque is shown below.



Double ended queue is represented in two ways. They are.

1. input restricted deque.
2. output restricted deque

Input restricted deque: In input restricted dequeue insertion is performed at rear end but deletion is performed at both the ends (front and rear).

Output restricted deque: In output restricted deque, deletion is performed at front end, but insertion is performed at both the ends (front and rear).

Operations on deque:

1. insertFront(): Insert an element at the front of Deque.
2. insertRear(): Insert an element at the rear of Deque.
3. deleteFront(): Deletes an element from the front of Deque.
4. deleteRear(): Deletes an element from the rear of Deque.

Implementing deque (double ended queue) using arrays

```
#include <stdio.h>
#include <stdlib.h>
#define size 10
int Q[size];
int f=-1,r=-1;

void insert_rear(int x)
{
        if((r+1)%size==f)
```

```
        {
                printf("\n Overflow");
        }
        else if(f==-1)
        {
                f=0;
                r=0;
                Q[r]=x;
        }
        else
        {
                r=(r+1)%size;
                Q[r]=x;
        }
}

void insert_front(int x)
{
        if((r+1)%size==f)
        {
                printf("\n Overflow");
        }
        else if(f==0)
        {
                f=size-1;
                Q[f] = x;
        }
        else
        {
                f=f-1;
                Q[f]=x;
        }
}

int delete_front()
{
        int x;

        if(f==-1)
        {
                printf("\n Underflow\n");
                return 0;
        }
        else if(f==r)
        {
                printf("\n Deleted element is %d",Q[f]);
                f=-1;
                r=-1;
        }
```

```
        else
        {
                printf("\n Deleted element is %d",Q[f]);
                f = (f+1)%size;

        }
}

int delete_rear()
{
        int x;
        if(f==-1)
        {
                printf("\n Underflow\n");
                return 0;
        }
        else if(f==r)
        {
                Printf("\n Deleted element is %d",Q[r]);
                f=-1;
                r=-1;
        }
        else
        {
                Printf("\n Deleted element is %d",Q[r]);
                if(r==0)
                        r=size-1;
                else
                        r = r-1;
        }
}

void display()
{
        int i;
        if(f==-1)
        {
                printf("\n Queue is empty\n");
        }
        else
        {
                for(i=f;i!=r;i=(i+1)%size)
                {
                        printf("%d\n",Q[i]);
                }
                printf("%d\n",Q[i]);
        }
}
```

```
int main(int argc, char *argv[]) {
        int ch,x;
        while(1)
        {
                printf("\n 1. Insert rear\n2. Insert front \n 3. delete front \n4.
delete rear\n5. display\n6. exit\n Enter your choice\n");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: printf("\n Enter your choice\n");
                                scanf("%d",&x);
                                insert_rear(x);
                                break;
                        case 2: printf("\n Enter your choice\n");
                                scanf("%d",&x);
                                insert_front(x);
                                break;\
                        case 3:  delete_front();
                                break;
                        case 4: delete_rear();
                                break;
                        case 5: display();
                                break;
                        case 6: exit(0);
                        default: printf("\n Enter correct choice\n");
                }
        }
        return 0;
}
```

Applications of Queue

1. **Task Scheduling**: Queues can be used to schedule tasks based on priority or the order in which they were received.
2. **Resource Allocation:** Queues can be used to manage and allocate resources, such as printers or CPU processing time.
3. **Batch Processing**: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.
4. **Message Buffering**: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.
5. **Event Handling**: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.
6. **Traffic Management**: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.
7. **Operating systems:** Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.

Important Questions:

Short Answer questions

1. Define queue, circular queue, deque, input restricted deque, and output restricted deque.
2. What is the drawback of a simple queue?
3. All Queues – Full and empty conditions.

Long Answer questions

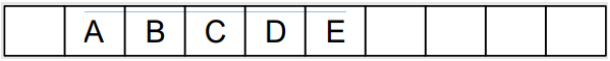
1. Write an algorithm for enqueue and dequeue.
2. Implement queue using arrays.
3. Implement circular queue using arrays.
4. Implement deque using arrays.

Practice questions

1. Draw the queue structure in each case when the following operations are performed on an empty queue.

(a) Add A, B, C, D, E, F
(b) Delete two letters
(c) Add G
(d) Add H
(e) Delete four letters
(f) Add I

2. Consider the queue given below which has FRONT = 1 and REAR = 5.



Now perform the following operations on the queue:
(a) Add F
(b) Delete two letters
(c) Add G
(d) Add H
(e) Delete four letters
(f) Add I

Multiple choice questions

1. A line in a grocery store represents a
   (a) Stack (b) Queue (c) Linked List (d) Array

2. In a queue, insertion is done at
   (a) Rear (b) Front (c) Back (d) Top

3. The function that deletes values from a queue is called
   (a) enqueue (b) dequeue (c) pop (d) peek

4. Typical time requirement for operations on queues is
   (a) O(1) (b) O(n) (c) O(log n) (d) O(n2)

5. The circular queue will be full only when
   (a) FRONT = MAX –1 and REAR = Max –1
   (b) FRONT = 0 and REAR = Max –1
   (c) FRONT = MAX –1 and REAR = 0
   (d) FRONT = 0 and REAR = 0

Linked Lists

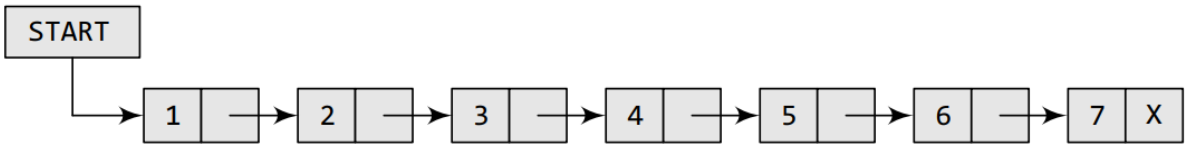
An array is a linear collection of data elements in which the elements are stored in consecutive memory locations. While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store. But what if we are not sure of the number of elements in advance? Moreover, to make efficient use of memory, the elements must be stored randomly at any location rather than in consecutive locations. So, there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs. So, there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs.

Linked list is a data structure that is free from the aforementioned restrictions. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.

Drawback of Linked list

Unlike an array, a linked list does not allow random access of data. Elements in a linked list can be accessed only in a sequential manner.

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.

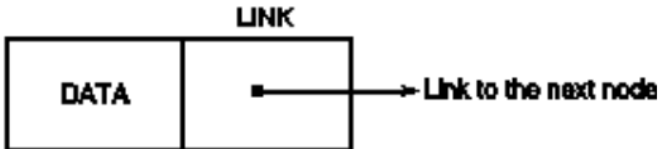


In a linked list, every node contains a pointer to another node which is of the same type, hence it is also called a self-referential data type.

Node structure

```
struct node
{
Int data;
struct node *next;
}
```

Visual representation of a Node



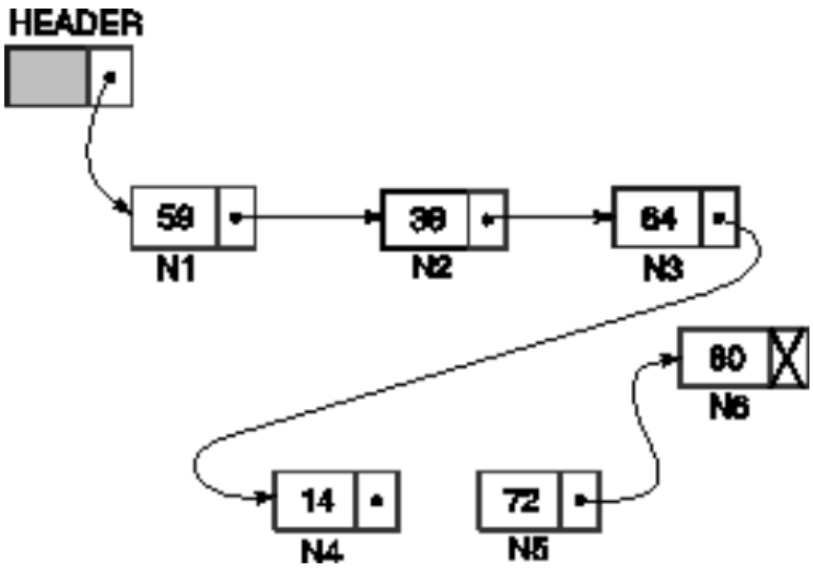
Representation of a Linked list in memory

There are two ways to represent a linked list in memory:

1. Static representation using array
2. Dynamic representation using free pool of storage

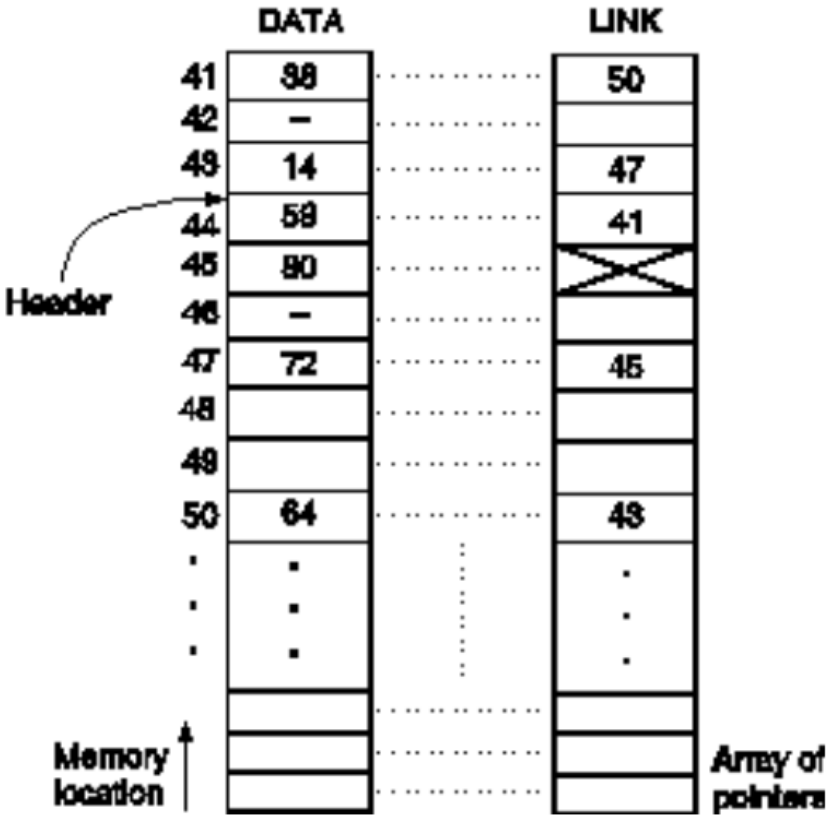
Static representation

In static representation of a single linked list, two arrays are maintained: one array for data and the other for links. The linked list and its static representation using arrays are shown in figures.



Linked list with six nodes.

Two parallel arrays of equal size are allocated which should be sufficient to store the entire linked list. Nevertheless, this contradicts the idea of the linked list (that is non-contagious location of elements). Hence it is not an efficient method.



Linked list representation using arrays

Dynamic representation

The efficient way of representing a linked list is using the free pool of storage. In this method, there is a *memory bank* (which is nothing but a collection of free memory spaces) and a *memory manager* (a program, in fact). During the creation of a linked list, whenever a node is required, the request is placed to the memory manager; the memory manager will then search the memory bank for the block requested and, if found, grants the desired block to the caller. Again, there is also another program called the *garbage collector*; it plays whenever a node is no more in use; it returns the unused node to the memory bank.

Operations on a single linked list

1. Inserting a node at the beginning of a single linked list
2. Inserting a node at the end of the single linked list
3. Inserting a node at any position
4. Deleting a node from the beginning of the single linked list
5. Deleting a node from the end of the single linked list
6. Deleting a node from any position in the linked list.
7. Traversing linked list (display linked list elements).
8. Searching a node in the linked list
9. Sorting the list elements

10. Reverse linked list elements.

Global declarations

```
typedef struct linkedlist
{
        int data;
        struct linkedlist *next;
}node;
node *head=NULL;
```

1. Inserting a mode at the beginning of the single linked list

Code:

```
void insert_begin()
{
        int x;
        node *temp;
        temp=(node*)malloc(sizeof(node));
        if(temp==NULL)
        {
                printf("\n Insufficient Memory\n");
        }
        else
        {
                printf("\n Enter a value\n");
                scanf("%d",&x);
                temp->data = x;
                temp->next = NULL;
                if(head==NULL)
                {
                        head=temp;
                }
                else
```

```
                {
                        temp->next = head;
                        head=temp;
                }
        }
}
```

2. Inserting a node at the end of the single linked list

```
void insert_end()
{
        int x;
        node *temp,*temp1;
        temp = (node*)malloc(sizeof(node));
        if(temp==NULL)
        {
                printf("\n Insufficient Memory\n");
        }
        else
        {
                printf("\n Enter a value\n");
                scanf("%d",&x);
                temp->data=x;
                temp->next=NULL;
                if(head==NULL)
                {
                        head=temp;
                }
                else
                {
                        for(temp1=head;temp1->next!=NULL;temp1=temp1-
>next);
                        temp1->next=temp;
                }
```

```
        }
}
```

3. **Inserting a node at any position in the single linked list**

```
void insert_pos()
{
        int x,p,n;    //p represents the position and n represents no of nodes in the linked list
        node *temp,*temp1;
        printf("\n Enter position:");
        scanf("%d",&p);
        for(temp1=head,n=1;temp1->next!=NULL;temp1=temp1->next,n++);
        if(p==1)
        {
                insert_begin();
        }
        else if(p==n+1)
        {
                insert_end();
        }
        else if(p > n+1||p<1)
        {
                printf("\n Insertion is not possible\n");
        }
        else
        {
                temp=(node*)malloc(sizeof(node));

                if(temp==NULL)
                {
                        printf("\n Insufficient Memory\n");

                }
```

```
            else
            {
                for(temp1=head,n=1;n<p-1;n++,temp1=temp1->next);
                printf("\n Enter a value\n");
                scanf("%d",&x);
                temp->data = x;
                //temp->next = NULL;
                temp->next = temp1->next;
                temp1->next = temp;
            }
        }
}
```

4. Deleting a node from the beginning of the single linked list

```
void delete_begin()
{
        node *temp;
        if(head==NULL)
        {
            printf("\n Linked List is empty\n");
        }
        else
        {
            temp=head;
            printf("\n Deleted node is %d \n",temp->data);
            head=head->next;
            free(temp);
        }
}
```

5. Deleting a node from the end of the single linked list

```
void delete_end()
{
        node *temp,*temp1;

        if(head==NULL)
        {
                printf("\n List is empty\n");
        }
        else
        {
                if(head->next==NULL)
                {
                        temp=head;
                        printf("\n Deleted node is %d\n",temp->data);
                        head=NULL;
                        free(temp);
                }
                else
                {
                        for(temp1=head;temp1->next->next!=NULL;temp1=temp1->next);
                        temp=temp1->next;
                        temp1->next=NULL;
                        printf("\n Deleted node is %d\n",temp->data);
                        free(temp);
                }
        }
}
```

6. Deleting a node from any position of the linked list

```
void delete_pos()
{
        node *temp,*temp1;
        int i, pos, count=0;
        printf("\n Enter position:");
        scanf("%d",&pos);
        for(temp1=head;temp1!=NULL;temp1=temp1->next, count++);
        if(pos==1)
        {
                delete_begin();
        }
        else if(pos==count)
        {
                delete_end();
        }
        else if(pos<1||pos>count)
        {
                printf("\n Deletion is not possible");
        }
        else
        {
                for(i=2,temp1=head; i<pos && i<count;i++,temp1=temp1->next);
                temp=temp1->next;
                temp1->next = temp->next;
                printf("\n Deleted node is %d \n", temp->data);
                free(temp);
        }
}
```

7. Traversing linked list (display linked list elements)

```
void display()
{
        node *temp;
        if(head==NULL)
        {
                printf("\n List is empty\n");
        }
        else
        {
                temp = head;
                for(temp=head;temp!=NULL;temp=temp->next)
                {
                        printf("%d\n",temp->data);
                }
        }
}
```

8. Searching a node in the linked list

```
void search()
{
        int key,found=0;
        node *temp;
        printf("\n Enter the key elements:");
        scanf("%d",&key);
        temp=head;
        if(temp==NULL)
        {
                printf("\n List is empty");
        }
        else
```

```
    {
        for(;temp!=NULL; temp=temp->next)
        {
            if(key==temp->data)
            {
                found=1;
                break;
            }
        }
        if(found==1)
        {
            printf("\n %d is found in the list",key);
        }
        else
        {
            printf("\n %d is not found in the list",key);
        }
    }
}
```

9. Sorting the list elements

```
    void sort()
    {
       int x;
       node *i,*j;
       i=head;
       while(i!=NULL)
       {
          j=i->next;
          while(j!=NULL)
          {
```

```
            if(i->data>j->data)
            {
                x=i->data;
                i->data=j->data;
                j->data=x;
            }
            j=j->next;
        }
        i=i->next;
    }
}
```

10. Reverse linked list elements

```
void reverse()
{
    node *curr,*prev,*next1;
    curr=head;
    prev=next1=NULL;
    while(curr!=NULL)
    {
        next1=curr->next;
        curr->next=prev;
        prev=curr;
        curr=next1;
    }
    head=prev;
}
```

The corresponding main program for the above functions is as follows.

Main program

```
typedef struct list
{
        int data;
        struct list *next;
}node;
node *head=NULL;
void main()
{
   int x;
   while(1)
   {
        printf("1.inser begin\n2.Insert End \n 3. Insert position\n 4. Delete begin \n 5. Insert end\n 6. Insert position \n 7. Search\n 8. Sort\n9.Reverse\n10.Display\n11.Exit\n");
       printf("Enter your choice\n");
       scanf("%d",&x);
       switch(x)
       {
          case 1:insert_begin(); break;
                case 2: insert_begin();break;
                case 3: insert_pos(); break;
                case 4: delete_begin(); break;
                case 5: delete_end(); break;
                case 6: delete_pos(); break;
                case 7: search(); break;
                case 8: sort();break;
                case 9: reverse();break;
                case 10: display();break;
                case 11: exit(0);
```

```
                default: printf("Enter correct choice\n");
        }
    }
}
```

C program to create a linked list.

```
#include<stdio.h>
#include<stdlib.h>
typedef struct linkedlist
{
    int data;
    struct linkedlist *next;
}node;
node *head=NULL;
void create()
{
    node *temp;
    char ch='y';
    int x;
    while(ch=='y')
    {
        temp=(node *)malloc(sizeof(node));
        if(temp!=NULL)
        {
            printf("Enter a value\n");
            scanf("%d",&x);
            temp->data=x;
            temp->next=NULL;
            if(head==NULL)
            {
                head=temp;
```

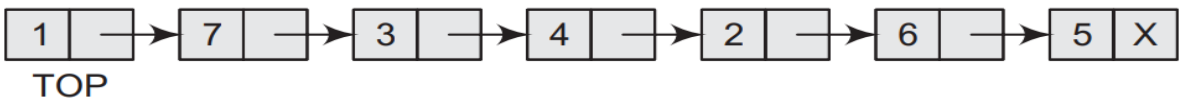
```
            }
            else
            {
                temp->next=head;
                head=temp;
            }
        }
        printf("Do you want to continue (y/n)\n");
        fflush(stdin);
        scanf("%c",&ch);
    }
}
void display()
{
    node *temp;
    if(head==NULL)
    {
        printf("List is empty\n");
    }
    else
    {
        for(temp=head;temp!=NULL;temp=temp->next)
        {
            printf("%d\n",temp->data);
        }
    }
}
void main()
{
    int x;
```

```
    while(1)
    {
        printf("1.create linked list \n2.Display\n3.Exit\n");

        printf("Enter your choice\n");
        scanf("%d",&x);
        switch(x)
        {
            case 1: create();break;
            case 2: display();break;
            case 3: exit(0);
            default:printf("Enter correct choice\n");
        }
    }
}
```

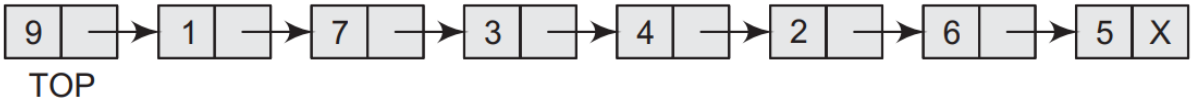
C Program for merging two linked lists.

C Program to implement stack using linked list

Push operation: The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown in Figure.



To insert an element with value 9, we first check if TOP=NULL. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP!=NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP. Thus, the updated stack becomes as shown in figure.



Algorithm

Step 1: Allocate memory for the new node and name it as NEW_NODE

Step 2: SET NEW_NODE DATA = VAL

Step 3: IF TOP = NULL SET TOP = ELSE SET TOP = [END OF IF]

Step 4: END

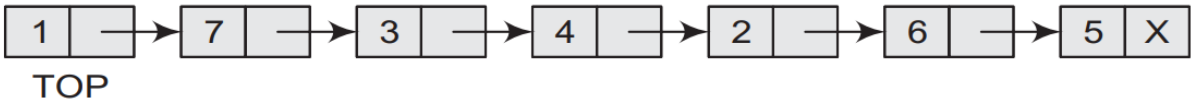
C Code: (which is similar to inserting a node at the beginning of the single linked list).

```
void insert_begin()
{
        int x;
        node *temp;
        temp=(node*)malloc(sizeof(node));
        if(temp==NULL)
        {
                printf("\n Insufficient Memory\n");
        }
        else
        {
                printf("\n Enter a value\n");
                scanf("%d",&x);
                temp->data = x;
                temp->next = NULL;
                if(top==NULL)
                {
                        top=temp;
                }
                else
                {
                        temp->next = top;
                        top=temp;
                }
```

```
    }
}
```

POP Operation: The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack shown in Figure.

In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown in Figure.



Algorithm:

Step 1: IF TOP = NULL PRINT UNDERFLOW [END OF IF]

Step 2: SET PTR = TOP

Step 3: SET TOP = TOP NEXT

Step 4: FREE PTR

Step 5: END

C Code: (which is similar to deleting a node from the beginning of the single linked list).

```
void delete_begin()
{
        node *temp;
        if(top ==NULL)
        {
                printf("\n Linked List is empty\n");
        }
        else
        {
                temp= top;
```

```
            printf("\n Deleted node is %d \n",temp->data);
            top=head->next;
            free(temp);
        }
}
```

C Program to implement stack using linked list

```
#include<stdio.h>
#include<stdlib.h>
typedef struct linkedlist
{
   int data;
   struct linkedlist *next;
}node;
node *top=NULL;
void push()
{
        int x;
        node *temp;
        temp=(node*)malloc(sizeof(node));
        if(temp==NULL)
        {
            printf("\n Insufficient Memory\n");
        }
        else
        {
            printf("\n Enter a value\n");
            scanf("%d",&x);
            temp->data = x;
            temp->next = NULL;
            if(top==NULL)
            {
```

```
                top=temp;
            }
            else
            {
                temp->next = top;
                top=temp;
            }
        }
}
void pop()
{
        node *temp;
        if(top ==NULL)
        {
            printf("\n Linked List is empty\n");
        }
        else
        {
            temp= top;
            printf("\n Deleted node is %d \n",temp->data);
            top=top->next;
            free(temp);
        }
}
void display()
{
    node *temp;
    if(top ==NULL)
    {
        printf("List is empty\n");
```

```
    }
    else
    {
        for(temp= top;temp!=NULL;temp=temp->next)
        {
            printf("%d\n",temp->data);
        }
    }
}
void main()
{
         int x;
         while(1)
         {
                printf("1.push\n2.pop\n3.Display\n4.Exit\n");
                printf("Enter your choice\n");
                scanf("%d",&x);
                switch(x)
                {
                    case 1:push();break;
                    case 2:pop();break;
                    case 3:display();break;
                    case 4:exit(0);
                    default:printf("Enter correct choice\n");
        }
    }
}
```

Queue implementation using linked list

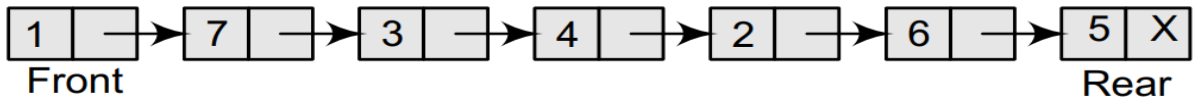
A queue has basically two operations 1. Enqueue 2. Dequeue

Enqueue: inserting an element into the queue

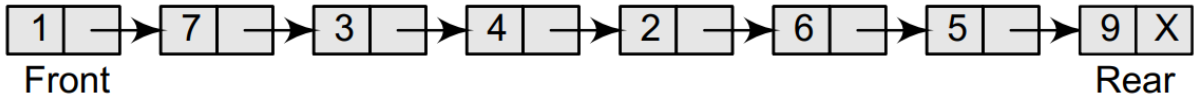
Dequeue: Deleting an element from the queue.

Enqueue:

The enqueue operation is used to insert an element into a queue. The new element is added as the last element of the queue. Consider the linked queue shown in Figure.



To insert an element with value 9, we first check if FRONT=NULL. If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its data part and NULL in its next part. The new node will then be called both FRONT and rear. However, if FRONT != NULL, then we will insert the new node at the rear end of the linked queue and name this new node as rear. Thus, the updated queue becomes as shown in Figure.



Algorithm

Step 1: Allocate memory for the new node and name it as PTR

Step 2: SET PTR DATA = VAL

Step 3: IF FRONT = NULL

SET FRONT = REAR = PTR

SET FRONT->NEXT = REAR->NEXT = NULL

ELSE

SET REAR->NEXT = PTR

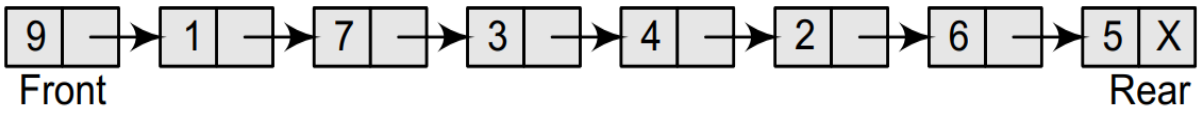
SET REAR = PTR

SET REAR->NEXT = NULL

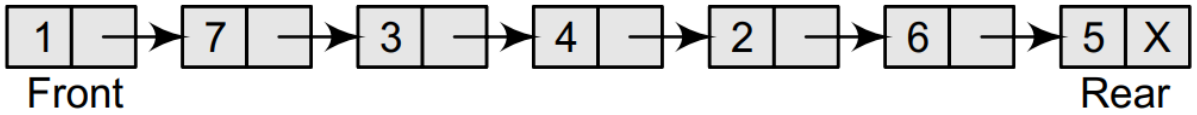
[END OF IF]

Step 4: END

Dequeue: The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT. However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed. Consider the queue shown in Figure.



To delete an element, we first check if FRONT=NULL. If the condition is false, then we delete the first node pointed by FRONT. The FRONT will now point to the second element of the linked queue. Thus, the updated queue becomes as shown in Figure.



Algorithm

Step 1: IF FRONT = NULL

Write “Underflow”

Go to Step 5

[END OF IF]

Step 2: SET PTR = FRONT

Step 3: SET FRONT = FRONT NEXT

Step 4: FREE PTR

Step 5: END

C Program to implement queue using linked list

```
//Queue using linked list
#include<stdio.h>
#include<stdlib.h>

typedef struct linkedlist
{
   int data;
   struct linkedlist *next;
```

```
} node;

node *front = NULL, *rear = NULL;

void enqueue()
{
    node *temp;
    int x;
    temp = (node *)malloc(sizeof(node));
    if(temp==NULL)
    {
        printf("Insufficient Memory\n");
    }
    else
    {
        printf("Enter a number : ");
        scanf("%d", &x);
        temp->data = x;
        temp->next = NULL;
        if(rear==NULL)
        {
            front = temp;
            rear = temp;
        }
        else
        {
            rear->next = temp;
            rear = temp;
        }
    }
```

```
}
void dequeue()
{
    node *temp;
    if(front==NULL)
    {
        printf("Queue Underflow\n");
    }
    else
    {
        temp = front;
        if(front->next==NULL) //Checking for one node
        {
            rear = NULL;
        }
        front = temp->next;
        printf("Deleted %d\n", temp->data);
        free(temp);
    }
}

void display()
{
    node *temp;
    if(rear==NULL)
    {
        printf("Queue is empty\n");
    }
    else
    {
```

```
        printf("Queue elements are");
        for(temp = front; temp!=rear; temp = temp->next)
        {
            printf("\n%d", temp->data);
        }
        printf("\n%d\n", rear->data);
    }
}

void main()
{
    int ch;
    while(1)
    {
        printf("\nOperations\n1. Enqueue\n2. Dequeue\n3. Display\n4.
Exit\n");
        printf("\nEnter your choice : ");
        scanf("%d", &ch);
        switch(ch)
        {
            case 1: enqueue();
                    break;
            case 2: dequeue();
                    break;
            case 3: display();
                    break;
            case 4: printf("\n");
                    exit(0);
            default: printf("\nInvalid Choice\n");
        }
```

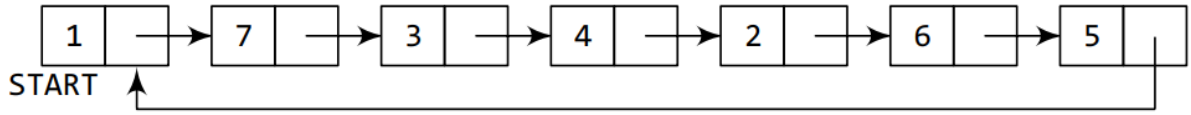
```
    }
}
```

Drawback of single linked list

Using single linked list, we cannot traverse and access the previous nodes. This can be overcome through circular linked list.

Circular linked list:

Definition: In circular linked list, every node points to the next node and the last node of the linked list points to the first node.



Note: This feature makes it circular in nature. It is essential to know that the circular linked lists have no end, and we need to be careful while traversing the linked list.

Operations performed on a circular linked list are

1. Inserting a node at the beginning of a circular linked list
2. Inserting a node at the end of the circular linked list
3. Inserting a node at any position in the circular linked list
4. Deleting a node from the beginning of the circular linked list
5. Deleting a node from the end of the circular linked list
6. Deleting a node from any position in the circular linked list.
7. Traversing linked list (display linked list elements).

C Program to implement circular linked list

```
//Circular Linked List
#include<stdio.h>
#include<stdlib.h>

typedef struct linkedlist
{
   int data;
   struct linkedlist *next;
```

```
} node;

node *head = NULL;

void insert_begin()
{
    node *temp, *temp1;
    int x;
    temp = (node *)malloc(sizeof(node));
    if(temp==NULL)
    {
        printf("Insufficient Memory\n");
    }
    else
    {
        printf("Enter a number : ");
        scanf("%d", &x);
        temp->data = x;
        temp->next = NULL;
        if(head==NULL)
        {
            head = temp;
            head->next = head;
        }
        else
        {
            temp->next = head;
            for(temp1=head;temp1->next!=head;temp1=temp1->next);
            temp1->next = temp;
            head = temp;
```

```
        }
    }
}

void insert_end()
{
    node *temp, *temp1;
    int x;
    temp = (node *)malloc(sizeof(node));
    if(temp==NULL)
    {
        printf("Insufficient Memory\n");
    }
    else
    {
        printf("Enter a number : ");
        scanf("%d", &x);
        temp->data = x;
        temp->next = NULL;
        if(head==NULL)
        {
            head = temp;
            head->next = head;
        }
        else
        {
            temp->next = head;
            for(temp1=head;temp1->next!=head;temp1=temp1->next);
            temp1->next = temp;
        }
```

```
    }
}

void insert_pos()
{
    node *temp, *temp1;
    int x, pos, count, i;
    printf("Enter a position : ");
    scanf("%d", &pos);
    for(count=1, temp1 = head; temp1->next!=head; temp1=temp1-
>next,count++);
    if(head==NULL)
    {
        count--;
    }
    if(pos<1||pos>count+1)
    {
        printf("Insertion not possible\n");
    }
    else if(pos==1)
    {
        insert_begin();
    }
    else if(pos==count+1)
    {
        insert_end();
    }
    else
    {
        temp = (node *)malloc(sizeof(node));
```

```
        if(temp==NULL)
        {
            printf("Insufficient Memory\n");
        }
        else
        {
            printf("Enter a number : ");
            scanf("%d", &x);
            temp->data = x;
            temp->next = NULL;
            temp1 = head;
            i = 2;
            while(i<pos)
            {
                temp1 = temp1->next;
                i++;
            }
            temp->next = temp1->next;
            temp1->next = temp;
        }
    }
}

void delete_begin()
{
    node *temp, *temp1;
    if(head==NULL)
    {
        printf("Underflow\n");
    }
```

```
    else if(head->next==head)
    {
        temp = head;
        printf("Deleted %d\n", temp->data);
        head = NULL;
        free(temp);
    }
    else
    {
        for(temp1=head;temp1->next!=head;temp1=temp1->next);
        temp = head;
        head = head->next;
        temp1->next = head;
        printf("Deleted %d\n", temp->data);
        free(temp);
    }
}

void delete_end()
{
    node *temp, *temp1;
    if(head==NULL)
    {
        printf("Underflow\n");
    }
    else if(head->next==head)
    {
        temp = head;
        printf("Deleted %d\n", temp->data);
        head = NULL;
```

```
        free(temp);
    }
    else
    {
        for(temp1=head;(temp1->next)->next!=head;temp1=temp1->next);
        temp = temp1->next;
        temp1->next = head;
        printf("Deleted %d\n", temp->data);
        free(temp);
    }
}

void delete_pos()
{
    node *temp, *temp1;
    int count, pos, i;
    if(head==NULL)
    {
        printf("Underflow\n");
    }
    else
    {
        for(temp=head,count=1;temp->next!=head;temp=temp->next,count++);
        printf("Enter a position : ");
        scanf("%d", &pos);
        if(pos<1||pos>count)
        {
            printf("Deletion is not possible\n");
        }
        else if(pos==1)
```

```
        {
            delete_begin();
        }
        else if(pos==count)
        {
            delete_end();
        }
        else
        {
            temp = head;
            i = 2;
            while(i<pos)
            {
                temp = temp->next;
                i++;
            }
            temp1 = temp->next;
            temp->next = temp1->next;
            printf("Deleted %d\n", temp1->data);
            free(temp1);
        }
    }
}

void display()
{
    node *temp;
    if(head==NULL)
    {
        printf("The List is empty\n");
```

```
    }
    else
    {
        printf("The list elements are\n");
        for(temp=head;temp->next!=head;temp=temp->next)
        {
            printf("%d\n", temp->data);
        }
        printf("%d\n", temp->data);
    }
}

void search()
{
    node *temp;
    int key, f=0;
    if(head==NULL)
    {
        printf("List is empty\n");
    }
    else
    {
        printf("Enter key : ");
        scanf("%d", &key);
        temp = head;
        while(temp->next!=head)
        {
            if(temp->data==key)
            {
                f=1;
```

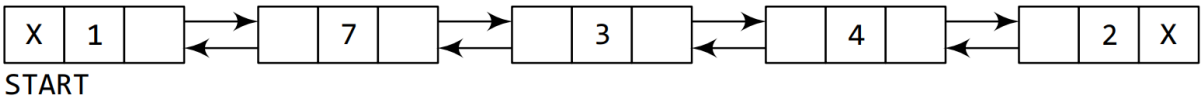
```
            break;
        }
        temp = temp->next;
    }
    if(temp->data==key)
    {
        f=1;
    }
    if(f==1)
    {
        printf("Search is successful\n");
    }
    else
    {
        printf("Search is unsuccessful\n");
    }
  }
}

void main()
{
  int ch;
  while(1)
  {
    printf("\nOperations\n");
    printf("1. Insert Begin\n2. Insert End\n3. Insert Position\n");
    printf("4. Delete Begin\n5. Delete End\n6. Delete Position\n");
    printf("7. Display\n8. Search\n9. Exit\n");
    printf("\nEnter your choice : ");
    scanf("%d", &ch);
```

```
        switch(ch)
        {
            case 1: insert_begin();
                    break;
            case 2: insert_end();
                    break;
            case 3: insert_pos();
                    break;
            case 4: delete_begin();
                    break;
            case 5: delete_end();
                    break;
            case 6: delete_pos();
                    break;
            case 7: display();
                    break;
            case 8: search();
                    break;
            case 9: printf("\n");
                    exit(0);
            default: printf("Invalid Choice\n");
        }
    }
}
```

Drawback of circular linked list: The main drawback of circular linked list is it cannot traverse in a reverse direction. To overcome this drawback through the double linked list.

Double linked list

A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node as shown in Figure.



In C, the structure of a doubly linked list can be given as,

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

The PREV field of the first node and the NEXT field of the last node will contain NULL. The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.

Double linked list operations

1. Inserting a node at the beginning of a double linked list
2. Inserting a node at the end of the double linked list
3. Inserting a node at any position in the double linked list
4. Deleting a node from the beginning of the double r linked list
5. Deleting a node from the end of the double linked list
6. Deleting a node from any position in the double linked list.
7. Traversing linked list (display linked list elements).

C Program to implement double linked list

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node
{
        int data;
        struct node *next;
        struct node *prev;
}dnode;
```

```
dnode *head=NULL;

void insert_begin()
{
        int x;
        dnode *temp;
        temp=(dnode*)malloc(sizeof(dnode));
        if(temp==NULL)
        {
                printf("\n Insufficient memory");
        }
        else
        {
                printf("\n Enter a value:");
                scanf("%d",&x);
                temp->data=x;
                temp->next=NULL;
                temp->prev=NULL;
                if(head==NULL)
                {
                        head=temp;
                }
                else
                {
                        temp->next=head;
                        head->prev=temp;
                        head = temp;
                }
        }
}
```

```
void insert_end()
{
        int x;
        dnode *temp,*last;
        temp=(dnode*)malloc(sizeof(dnode));

        if(temp==NULL)
        {
                printf("\n Insufficient memory");
        }
        else
        {
                printf("\n Enter a value:");
                scanf("%d",&x);
                temp->data=x;
                temp->next=NULL;
                temp->prev=NULL;

                if(head==NULL)
                {
                        head=temp;
                }
                else
                {
                        for(last=head;last->next!=NULL;last=last->next);
                        last->next=temp;
                        temp->prev=last;
                }
        }
```

```
}

void insert_pos()
{
        dnode *temp,*temp1;
        int count,pos,x,i;
        for(count=0,temp=head;temp!=NULL;temp=temp->next,count++);
        printf("\n Enter position:");
        scanf("%d",&pos);
        if(pos==1)
        {
                insert_begin();
        }
        else if(pos==count+1)
        {
                insert_end();
        }
        else
        {
                temp=(dnode*)malloc(sizeof(dnode));
                if(temp!=NULL)
                {
                        printf("\n Enter a value:");
                        scanf("%d",&x);
                        temp->data=x;
                        temp->next=NULL;
                        temp->prev=NULL;
                        i=2;
                        temp1=head;
                        while(i<pos)
```

```
                {
                        temp1=temp1->next;
                        i++;
                }
                temp->next=temp1->next;
                temp1->next->prev = temp;
                temp1->next = temp;
                temp->prev = temp1;
        }
    }
}

void delete_begin()
{
    dnode *temp;
    if(head==NULL)
    {
        printf("\n List is empty");
    }
    else
    {
        if(head->next==NULL)
        {
            temp=head;
            printf("\n deleted node is %d",temp->data);
            head=NULL;
            free(temp);
        }
        else
        {
```

```
                temp=head;
                printf("\n deleted node is %d",temp->data);
                head=head->next;
                head->prev=NULL;
                free(temp);
            }
        }
}

void delete_end()
{
        dnode *temp,*temp1;
        if(head==NULL)
        {
            printf("\n List is empty");
        }
        else
        {
            if(head->next==NULL)
            {
                temp=head;
                printf("\n deleted node is %d",temp->data);
                head=NULL;
                free(temp);
            }
            else
            {
                for(temp1=head;temp1->next->next!=NULL;temp1=temp1-
>next);
                temp=temp1->next;
```

```
                printf("\n deleted node is %d",temp->data);
                temp1->next=NULL;
                free(temp);
            }
        }
}

void delete_pos()
{
        dnode *temp,*temp1;
        int pos,count,i;
        for(count=0,temp=head;temp!=NULL;temp=temp->next,count++);
        printf("\n Enter position:");
        scanf("%d",&pos);
        if(pos==1)
        {
            delete_begin();
        }
        else if(pos==count)
        {
            delete_end();
        }
        else
        {
            i=2;
            temp1=head;
            while(i<pos)
            {
                temp1=temp1->next;
                i++;
```

```
            }
            temp=temp1->next;
            printf("\n Deleted element is %d",temp->data);
            temp1->next=temp->next;
            temp1->next->prev = temp->prev;
            free(temp);
        }
}
void display_front()
{
        dnode *temp;
        if(head==NULL)
        {
            printf("\n List is Empty");
        }
        else
        {
            printf("\n");
            for(temp=head;temp!=NULL;temp=temp->next)
            {
                printf(" %d-->",temp->data);
            }
            printf("\n");
        }
}

void display_last()
{
        dnode *last;
        if(head==NULL)
```

```
	{
		printf("\n List is empty");
	}
	else
	{
		for(last=head;last->next!=NULL;last=last->next);
		printf("\n");
		do
		{
			printf(" %d-->",last->data);
			last=last->prev;
		}while(last!=NULL);
		printf("\n");
	}
}

void main()
{
	int ch;
	while(1)
	{
		printf("\n 1. Insert begin \n 2. Insert End \n 3. Insert any position \n 4. Delete Begin \n 5. Delete End\n 6. Delete Position \n 7. Display from front \n 8. Display from last \n 9. exit\n");
		printf("\n Enter your choice\n");
		scanf("%d",&ch);
		switch(ch)
		{
			case 1: insert_begin();
				break;
			case 2: insert_end();
```

```
                        break;
                case 3: insert_pos();
                        break;
                case 4: delete_begin();
                        break;
                case 5: delete_end();
                        break;
                case 6: delete_pos();
                        break;
                case 7: display_front();
                        break;
                case 8: display_last();
                        break;
                case 9: exit(0);
                default: printf("\n Enter correct choice\n");
            }
        }
}
```

C Program for merging two linked list elements

```
#include <stdio.h>
#include <stdlib.h>

typedef struct list
{
        int data;
        struct list *next;
}node;

node *create(node *first)
```

```
{
        node *temp;
        char ch='y';
        int x;
        while(ch=='y')
        {
                temp=(node*)malloc(sizeof(node));
                if(temp==NULL)
                {
                        printf("\n Insufficient memory");
                        return NULL;
                }
                else
                {
                        printf("\n Enter a value:");
                        scanf("%d",&x);
                        temp->data=x;
                        temp->next=NULL;
                        if(first==NULL)
                        {
                                first = temp;
                                printf("\n %d",first->data);
                        }
                        else
                        {
                                temp->next=first;
                                first = temp;
                                printf("\n %d",first->data);
                        }
                }
```

```
            printf("\n Do you want to continue: (y/n):");
            fflush(stdin);
            scanf("%c",&ch);
        }
        return first;
}

node *sort(node *head)
{
        node *i,*j;
        int x;
        i=head;
        while(i!=NULL)
        {
            j=i->next;
            while(j!=NULL)
            {
                if(i->data > j->data)
                {
                    x=i->data;
                    i->data=j->data;
                    j->data=x;
                }
                j=j->next;
            }
            i=i->next;
        }
        return head;
}
```

```
void display(node *first)
{
        node *temp;
        if(first==NULL)
        {
                printf("\n List is empty");
        }
        else
        {
                printf("\n");
                for(temp=first;temp!=NULL;temp=temp->next)
                {
                        printf("%d-->",temp->data);
                }
                printf("\n");
        }
}
node *merge(node *first,node *second)
{
        node *temp1,*temp2,*temp3,*head;
        temp1=first;
        temp2=second;
        head=temp3=NULL;
        if(temp1->data > temp2->data)
        {
                head = temp3 = temp2;
                temp2 = temp2->next;
                temp3->next=NULL;
        }
        else
```

```
{
        head = temp3 = temp1;
        temp1 = temp1->next;
        temp3->next=NULL;
}
while(temp1!=NULL&&temp2!=NULL)
{
        if(temp1->data > temp2->data)
        {
                temp3->next = temp2;
                temp2 = temp2->next;
                temp3=temp3->next;
        }
        else
        {
                temp3->next = temp1;
                temp1 = temp1->next;
                temp3=temp3->next;
        }
}
while(temp1!=NULL)
{
        temp3->next = temp1;
        temp1=temp1->next;
        temp3=temp3->next;
}
while(temp2!=NULL)
{
        temp3->next = temp2;
        temp2=temp2->next;
```

```
            temp3=temp3->next;
        }
        temp3->next=NULL;
        return head;
}

int main(int argc, char *argv[]) {
        node *first=NULL,*second=NULL,*third=NULL;
        printf("\n Enter first list values:");
        first = create(first);
        printf("\n Enter Second list values:");
        second = create(second);
        printf("\n First list values are:");
        display(first);
        printf("\n Second list values are:");
        display(second);
        first = sort(first);
        printf("\n After sorting first list values are:");
        display(first);
        second = sort(second);
        printf("\n After sorting second list values are:");
        display(second);
        third = merge(first,second);
        printf("\n After merging list values are:");
        display(third);
        return 0;
}
```

C program to implement polynomial addition

```
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct polynomial
{
        int coef;
        int expo;
        struct polynomial *next;
}poly;

poly *create(poly *first)
{
        poly *temp,*temp1;
        char ch='y';
        int x,y;
        while(ch=='y')
        {
                temp=(poly*)malloc(sizeof(poly));
                if(temp==NULL)
                {
                        printf("\n Insufficient memory");
                        return NULL;
                }
                else
                {
                        printf("\n Enter coefficient value:");
                        scanf("%d",&x);
                        printf("\n Enter exponent value:");
                        scanf("%d",&y);
                        temp->coef=x;
                        temp->expo=y;
                        temp->next=NULL;
```

```
                        if(first==NULL)
                        {
                                first = temp;
                        }
                        else
                        {
                                for(temp1=first;temp1->next!=NULL;temp1=temp1-
>next);
                                temp1->next =temp;
                        }
                }
                printf("\n Do you want to continue: (y/n):");
                fflush(stdin);
                scanf("%c",&ch);
        }
        return first;
}


void display(poly *first)
{
        poly *temp;
        if(first==NULL)
        {
                printf("\n List is empty");
        }
        else
        {
                printf("\n");
                for(temp=first;temp!=NULL;temp=temp->next)
```

```
                {
                        printf(" %dX%d +",temp->coef,temp->expo);
                }
                printf("\n");
        }
}

poly *addition(poly *first,poly *second)
{
        poly *f,*s,*head=NULL,*temp=NULL;
        f=first;
        s=second;
        head=temp;

        while(s!=NULL&&f!=NULL)
        {
                if(head==NULL)
                {
                        head=(poly*)malloc(sizeof(poly));
                        temp=head;

                }
                else
                {
                        temp->next = (poly*)malloc(sizeof(poly));
                        temp=temp->next;
                }
                if(s->expo > f->expo)
                {
                        temp->coef=s->coef;
```

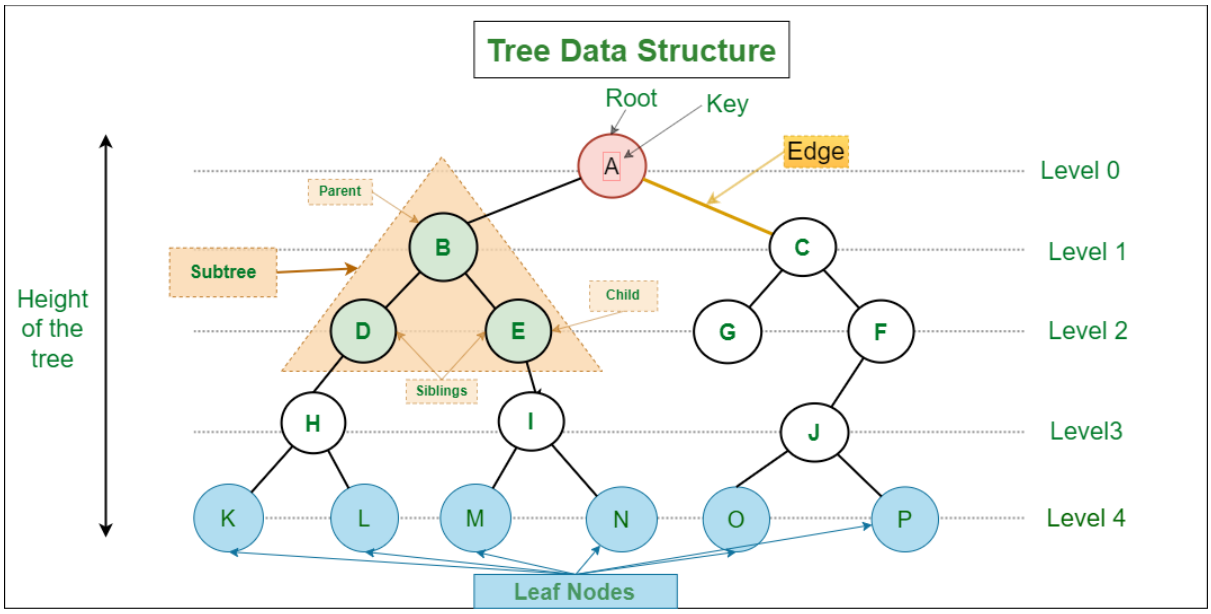
```
                temp->expo = s->expo;
                s = s->next;
        }
        else if(s->expo < f->expo)
        {
                temp->coef=f->coef;
                temp->expo = f->expo;
                f=f->next;
        }
        else
        {
                temp->coef=f->coef+s->coef;
                temp->expo = f->expo;
                s=s->next;
                f=f->next;
        }
}
while(f!=NULL)
{
        temp->next = (poly*)malloc(sizeof(poly));
        temp=temp->next;
        temp->coef= f->coef;
        temp->expo =f->expo;
        f=f->next;
}
while(s!=NULL)
{
        temp->next = (poly*)malloc(sizeof(poly));
        temp=temp->next;
        temp->coef= s->coef;
```

```
            temp->expo =s->expo;
            s=s->next;
        }
        temp->next=NULL;
        return head;
}

int main(int argc, char *argv[]) {
        poly *first=NULL,*second=NULL,*third=NULL;
        printf("\n Enter first polynomial values(Give correct order):");
        first = create(first);
        printf("\n Enter second polynomial values(Give correct order):");
        second = create(second);
        printf("\n first polynomial values are:");
        display(first);
        printf("\n Second polynomial values are:");
        display(second);
        third=addition(first,second);
        printf("\n Addition of two polynomials are:");
        display(third);
        return 0;
}
```

UNIT-3

Trees

Trees are very flexible, versatile and powerful non-liner data structure that can be used to represent data items possessing hierarchical relationship between the grand father and his children and grand children as so on.



A tree is an ideal data structure for representing hierarchical data

Definition: A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that:

1. There is a special node called the root of the tree.

2. Removing nodes (or data item) are partitioned into number of mutually exclusive (i.e., disjoined) subsets each of which is itself a tree, are called sub tree.

Tree Terminology

Root node: The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.

Sub-trees: If the root node R is not NULL, then the trees T1 , T2 , and T3 are called the sub-trees of R.

Leaf node: A node that has no children is called the leaf node or the terminal node.

Path: A sequence of consecutive edges is called a path.

For example, in Figure, the path from the root node A to node M is given as: A, B,E,I and M.

Ancestor node An ancestor of a node is any predecessor node on the path from root to that node.

Note: The root node does not have any ancestors.

In the tree given in Figure, nodes A, C, and F are the ancestors of node J.

Descendant node A descendant node is any successor node on any path from the node to a leaf node.

Note: Leaf nodes do not have any descendants.

In the tree given in Figure, nodes C, F, J, and O are the descendants of node A.

Level number Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1 and so on.

Thus, every node is at one level higher than its parent.

So, all child nodes have a level number given by parent's level number + 1.

Degree Degree of a node is equal to the number of children that a node has.

Note: The degree of a leaf node is zero.

In-degree In-degree of a node is the number of edges arriving at that node.

Out-degree Out-degree of a node is the number of edges leaving that node

Types of Trees

1. General trees
2. Forests
3. Binary trees
4. Binary search trees
5. Expression trees
6. Tournament tree

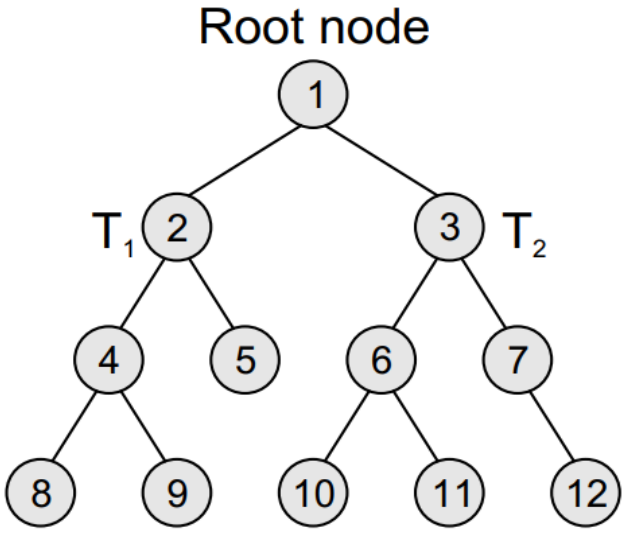
We are focusing only on Binary tress and Binary search trees.

Binary Tree:

A binary tree is a tree in which no node can have more than two children. Typically, these children are described as "left child" and "right child" of the parent node.

Definition: A binary tree T is defined as a finite set of elements called nodes, such that

1. T is empty (i.e. if T has no nodes called the NULL tree or empty tree).
2. T contains a special node R, called the root node of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T1 and T2 each of which represents a binary tree.



Parent If N is any node in T that has left successor S1 and right successor S2, then N is called the parent of S1 and S2.

Correspondingly, S1 and S2 are called the left child and the right child of N. Every node other than the root node has a parent.

Level number Every node in the binary tree is assigned a level number (refer Figure). The root node is defined to be at level 0. The left and the right child of the root node have a level number 1.

Similarly, every node is at one level higher than its parents. So all child nodes are defined to have level number as parent's level number + 1.

Degree of a node It is equal to the number of children that a node has.

The degree of a leaf node is zero.

For example, in the tree, degree of node 4 is 2, degree of node 5 is zero and degree of node 7 is 1.

Sibling All nodes that are at the same level and share the same parent are called siblings (brothers).

For example, nodes 2 and 3; nodes 4 and 5; nodes 6 and 7; nodes 8 and 9; and nodes 10 and 11 are siblings.

Leaf node A node that has no children is called a leaf node or a terminal node.

The leaf nodes in the tree are: 8, 9, 5, 10, 11, and 12.

Similar binary trees Two binary trees T and T' are said to be similar if both these trees have the same structure.

Edge It is the line connecting a node N to any of its successors. A binary tree of n nodes has exactly n – 1 edges because every node except the root node is connected to its parent via an edge.

Path A sequence of consecutive edges.

For example, in Figure, the path from the root node to the node 8 is given as: 1, 2, 4, and 8.

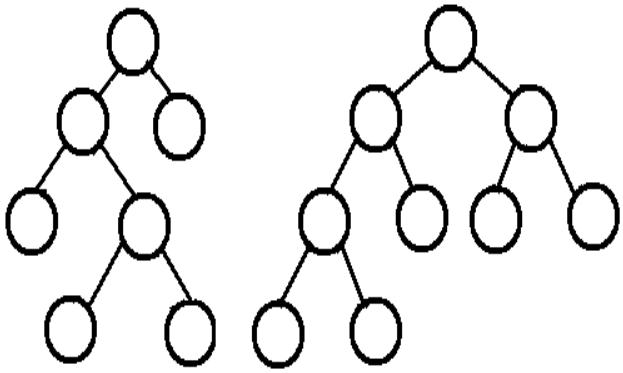
Depth The depth of a node N is given as the length of the path from the root R to the node N. The depth of the root node is zero.

Height of a tree It is the total number of nodes on the path from the root node to the deepest node in the tree. A tree with only a root node has a height of 1.

Note: A binary tree of height h has at least h nodes and at most 2^h-1 nodes.

Full binary tree: A full binary tree is a binary tree in which every node has **0 or 2 children**. All the nodes have two children except the leaf nodes.

Complete Binary Tree: It is nothing but a type of binary tree in which **all levels are filled** except possibly the last level. The nodes in the last level of a complete binary tree should be **as left as** possible. This indicates that a **particular order i**s followed for the filling in nodes in a complete binary tree from left to right.



full tree complete tree

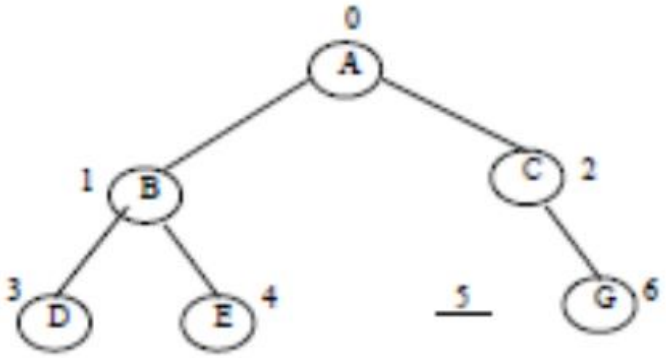
Representation of Binary trees:

There are two ways to represent a binary tree in memory:

1. Arrays 2. Linked lists.

Array Representation: An array can be used to store the nodes of a binary tree. The nodes stored in an array of memory can be accessed sequentially. Suppose a binary tree T of depth d. Then at most $2^d - 1$ nodes can be there in T (i.e., SIZE = 2^d–1). So the array of size "SIZE" to represent the binary tree. Consider a binary tree of depth 3. Then SIZE = $2^3 - 1 = 7$. Then the array A[7] is declared to hold the nodes.

The array representation of the binary tree is shown. To perform any operation often we have to identify the father, the left child and right child of an arbitrary node.



1. The parent of a node having index n can be obtained by (n – 1)/2.
   For example to find the father of D, where array index n = 3. Then the father nodes index can be obtained = (n – 1)/2 = (3-1)/2=1. i.e., A[1] is the father D, which is B.

2. The left child of a node having index n can be obtained by (2n+1).

For example, to find the left child of C, where array index n = 2. Then it can be obtained by = (2n +1) = 2*2 + 1 = 4 + 1 = 5 i.e., A[5] is the left child of C, which is NULL. So no left child for C.

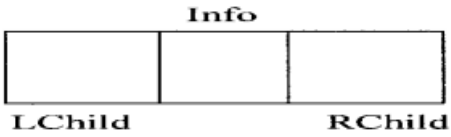
3. The right child of a node having array index n can be obtained by the formula (2n + 2).

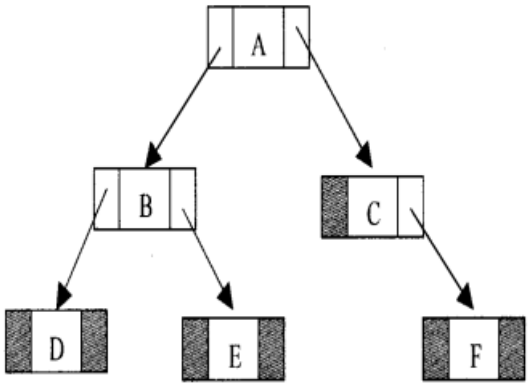
For example, to find the right child of B, where the array index n = 1. Then = (2n + 2) = 2*1 + 2 = 4. i.e., A[4] is the right child of B, which is E.

Note: Complete binary trees are better as compared to the full binary trees when a tree is represented using an array.

Linked List Representation

The most popular and practical way of representing a binary tree is using linked list (or pointers). In linked list, every element is represented as nodes. A node consists of three fields such as : (a) Left Child (LChild) (b) Information of the Node (Info) (c) Right Child (RChild)



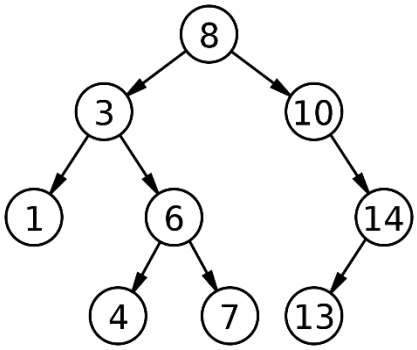


If a node does not have left/right child, corresponding left/right child is assigned to NULL.

Binary Search Tree (BST)

A binary search tree (BST) is a binary tree where the value of every node in the left subtree is less than the root, and every node in the right subtree value is greater than the root.

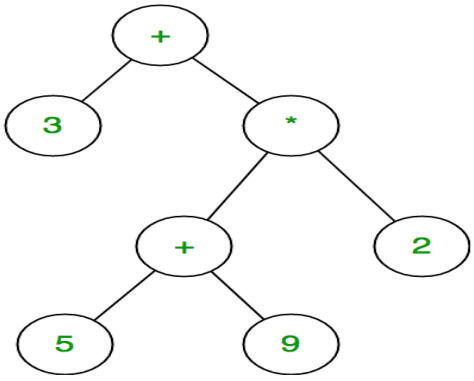
The following tree represents the binary search tree.



Expression Tree

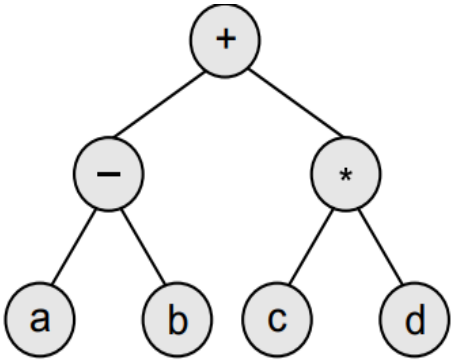
The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand.

Expression tree for 3 + ((5+9)*2) would be:

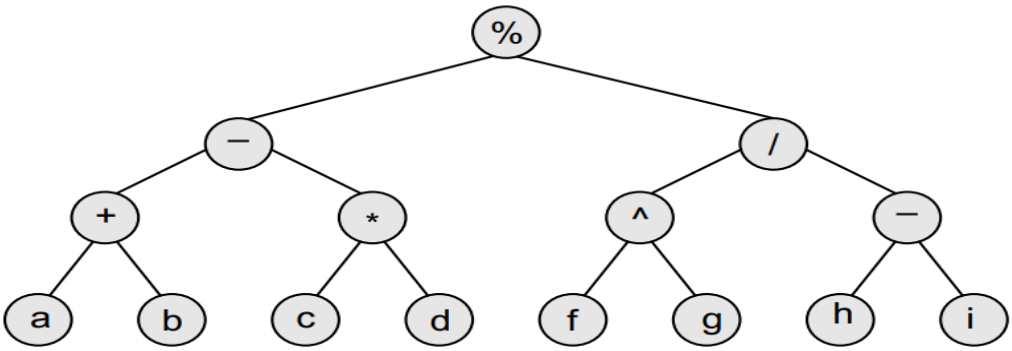


Example:

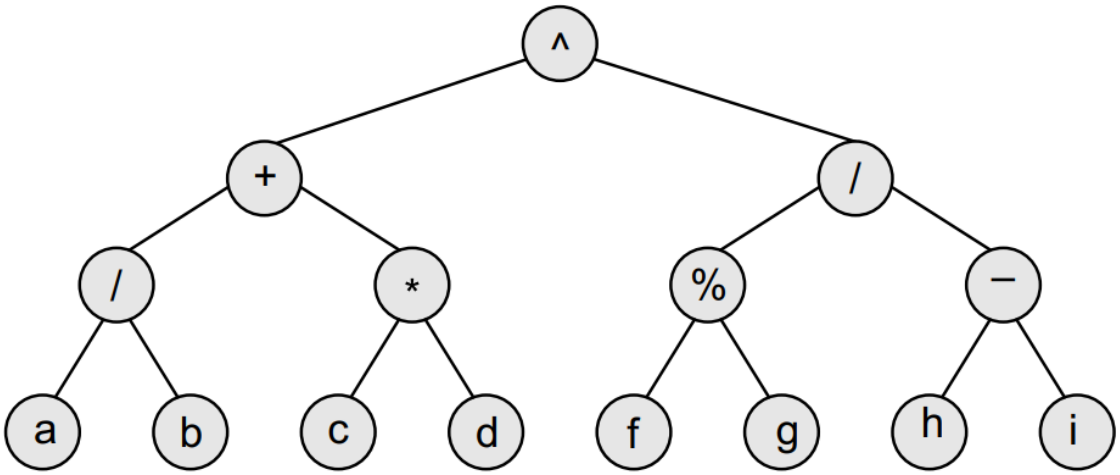
1. Construct expression tree for (a – b) + (c * d)



2. Construct expression tree for ((a + b) – (c * d)) % ((e ^f) / (g – h))



3. Given the binary tree, write down the expression that it represents



Answer: [{(a/b) + (c*d)} ^ {(f % g)/(h – i)}]

Traversing a Binary Tree

Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way.

There are three different algorithms for traversing a binary tree. Which are

1. Pre-order traversal

2. In-order traversal

3. Post-order traversal

Pre-order Traversal

Steps:

Step 1: Visiting the root node.

Step 2. Traversing the left sub-tree, and finally

Step 3. Traversing the right sub-tree.

Algorithm

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: Write TREE→ DATA

Step 3: PREORDER(TREE →LEFT)

Step 4: PREORDER(TREE →RIGHT)

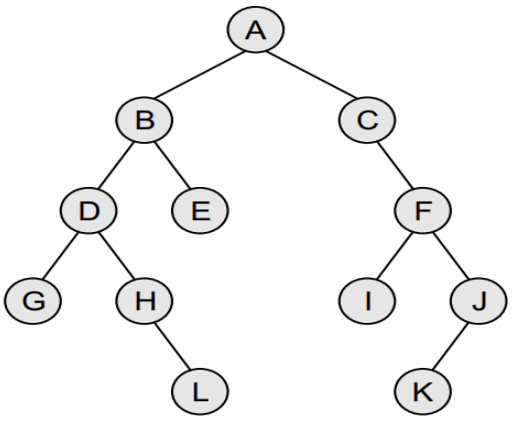
[END OF LOOP]

Step 5: END

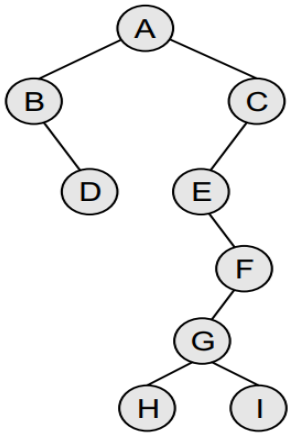
Pre-order traversal is also called as depth-first traversal. In this algorithm, the left sub-tree is always traversed before the right sub-tree. The word 'pre' in the pre-order specifies that the root node is accessed prior to any other nodes in the left and right sub-trees.

Note: Pre-order traversal algorithms are used to extract a prefix notation from an expression tree.

Q: In Figs (a) and (b), find the sequence of nodes that will be visited using pre-order traversal algorithm.



(a)



(b)

Solution: For tree(a): A, B, D, G, H, L, E, C, F, I, J, K

For tree(b): A, B, D, C, D, E, F, G, H, I

In-order Traversal

Steps:

1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree

Algorithm:

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: INORDER(TREE →LEFT)

Step 3: Write TREE →DATA

Step 4: INORDER(TREE→ RIGHT)

[END OF LOOP]

Step 5: END

In this algorithm, the left sub-tree is always traversed before the root node and the right sub-tree. The word 'in' in the in-order specifies that the root node is accessed in between the left and the right sub-trees. In-order algorithm is also known as the LNR traversal algorithm (Left-Node-Right).

The in-order traversal for the above trees is

For tree(a) - G, D, H, L, B, E, A, C, I, F, K, J

For tree(b) - B, D, A, E, H, G, I, F, C

Post-order Traversal

Steps:

1. Traversing the left sub-tree,

2. Traversing the right sub-tree, and finally

3. Visiting the root node.

Algorithm:

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: POSTORDER(TREE →LEFT)

Step 3: POSTORDER(TREE →RIGHT)

Step 4: Write TREE →DATA

[END OF LOOP]

Step 5: END

In this algorithm, the left sub-tree is always traversed before the right sub-tree and the root node. The word 'post' in the post-order specifies that the root node is accessed after the left and the right sub-trees. Post-order algorithm is also known as the LRN traversal algorithm (Left-Right-Node).

The post-order traversal for the above trees is

For tree(a): G, L, H, D, E, B, I, K, J, F, C, A

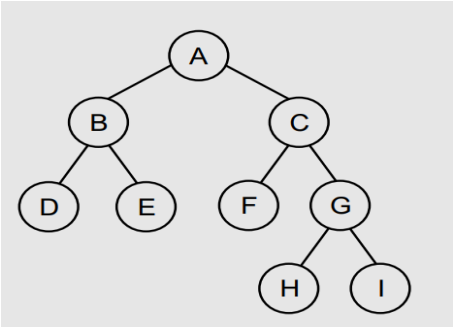
For tree(b): D, B, H, I, G, F, E, C, A

Q: What is a binary tree? Construct a binary tree given the pre-order traversal and inorder traversals as follows:

Pre-Order Traversal: G B Q A C K F P D E R H

In-Order Traversal: Q B K C F A G P E D H R

Q: Find the pre-order, in-order and post-order traversal for the given binary tree.



Q: What is a binary tree? Construct a binary tree given the pre-order traversal and in-order traversals as follows:

- Pre-Order Traversal: G B Q A C K F P D E R H
- In-Order Traversal: Q B K C F A G P E D H R

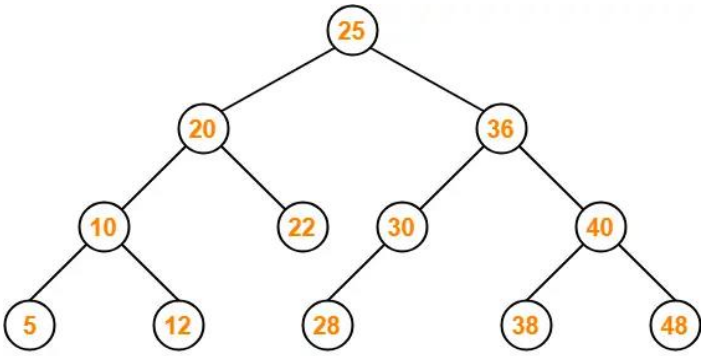
Q: A binary tree has seven nodes. The Preorder and Post-order traversal of the tree are given below. Can you draw the tree? Justify.

- Preorder : GFDABEC
- Post-order : ABDCEFG

Binary Search Tree (BST)

A binary search tree is also known as an ordered binary tree, in which the value of any node in its left subtree is less than the value of any node and the value of any node in its right subtree is greater than the value of the node.

Example:



Binary Search Tree

Q: Develop a binary search tree after inserting the following keys 49, 27, 12, 11, 33, 77, 26, 56, 23, and 6.

1) check whether the tree is almost complete or not.

2) Determine the height of the tree.

3) write post-order and pre-order traversals.

C structure to represent a tree node

```
struct node
{
        Int data;
        struct node *left_child;
        struct node *right_node;
};
```

C Program to create a BST and tree traversals

```
#include <stdio.h>
#include <stdlib.h>


/* run this program using the console pauser or add your own getch, system("pause") or input loop */
```

```
typedef struct node
{
        int data;
        struct node *lchild;
        struct node *rchild;
}tree;

tree *createtree(tree *root,int value)
{
        if(root==NULL)
        {
                root = (tree*)malloc(sizeof(tree));
                root->data=value;
                root->lchild=NULL;
                root->rchild=NULL;
        }
        else
        {
                if(value < root->data)
                {
                        root->lchild=createtree(root->lchild,value);
                }
                else if(value > root->data)
                {
                        root->rchild=createtree(root->rchild,value);
                }
                else
                {
                        printf("\n Duplicate Element:");
                }
```

```
            return root;
        }

}

void inorder(tree *root)
{
        if(root!=NULL)
        {
                inorder(root->lchild);
                printf("\n %d",root->data);
                inorder(root->rchild);
        }
}

void preorder(tree *root)
{
        if(root!=NULL)
        {
                printf("\n %d",root->data);
                preorder(root->lchild);
                preorder(root->rchild);
        }
}

void postorder(tree *root)
{
        if(root!=NULL)
        {
                postorder(root->lchild);
```

```
                postorder(root->rchild);
                printf("\n %d",root->data);
        }
}

int main(int argc, char *argv[]) {
        tree *root=NULL;
        int ch,x,i,n;
        do
        {
                printf("\n 1. create\n 2. In order\n 3. Pre order\n 4. Post
order\n 5. exit\n");
                printf("\n Enter your choice:");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: printf("\n Enter no of nodes:");
                                scanf("%d",&n);
                                for(i=1;i<=n;i++)
                                {
                                        printf("\n Enter value:");
                                        scanf("%d",&x);
                                        root=createtree(root,x);
                                }
                                break;
                        case 2: inorder(root);
                                break;
                        case 3: preorder(root);
                                break;
                        case 4: postorder(root);
```

```
                    break;
                case 5: exit(0);
                default: printf("\n Enter correct choice:");
            }
        }while(1);
        return 0;
}
```

Deletion

We should take care when we delete node from the BST, because after removing the node the tree must satisfy binary search tree property. It has 3 cases.

1. Deleting a node that has no children.

2. deleting a node with one child

3. deleting a node with two children.

1. Deleting a node that has no children

We can simply remove this node without any issue. This is the simplest case of deletion.

Note: Refer class notes for example

2. Deleting a node with one child

To handle this case, the nodes child is set as the child of the nodes parent. In other words replace the node with its child.

Refer class notes for example

3. deleting a node with two children

To handle this case, replace the nodes value with its in-order predecessor (largest value in the left sub tree) or in order successor (smallest value in the right sub tree). The in order predecessor or the successor can then be deleted using any of the above cases.

Refer class notes for example

Binary search tree program for creating BST, tree traversals and deleting a node from BST.

```
#include <stdio.h>
#include <stdlib.h>

/* run this program using the console pauser or add your own getch, system("pause") or input loop */
typedef struct node
{
	int data;
	struct node *lc;
	struct node *rc;
}tree;
/*
tree *create_tree(tree*);
void inorder(tree*);
void preorder(tree*);
void postorder(tree*);
*/
tree *create_tree(tree *root,int value)
{
	if(root==NULL)
	{
		root = (tree*)malloc(sizeof(tree));
		root->data = value;
		root->lc = NULL;
		root->rc = NULL;
	}
	else
	{
		if(value < root->data )
```

```
		{
			root->lc = create_tree(root->lc,value);
		}
		else if(value > root->data)
		{
			root->rc = create_tree(root->rc,value);
		}
		else
		{
			printf("\n Duplicate element !!!!!");
		}
		return root;
	}
}

void inorder(tree *root)
{
	if(root!=NULL)
	{
		inorder(root->lc);
		printf("\t%d",root->data);
		inorder(root->rc);
	}
}

void preorder(tree *root)
{
	if(root!=NULL)
	{
		printf("\t%d",root->data);
```

```
                preorder(root->lc);
                preorder(root->rc);
        }
}

void postorder(tree *root)
{
        if(root!=NULL)
        {
                postorder(root->lc);
                postorder(root->rc);
                printf("\t%d",root->data);
        }
}

tree *insert(tree *root, int value)
{
        if(root==NULL)
        {
                root=(tree*)malloc(sizeof(tree));
                root->data=value;
                root->lc=root->rc=NULL;
                return root;
        }
        else
        {
                if(value < root->data )
                {
                        root->lc = create_tree(root->lc,value);
                }
```

```
        else if(value > root->data)
        {
                root->rc = create_tree(root->rc,value);
        }
        else
        {
                printf("\n Duplicate element !!!!!");
        }
        return root;
    }
}

tree *findmin(tree *root)
{
    tree *temp=root;
    while(temp&&temp->lc!=NULL)
    {
        temp=temp->lc;
    }
    return temp;
}
tree *deletenode(tree *root, int value)
{
    tree *temp;
    if(root==NULL)
    {
        printf("\n Tree is Empty\n");
        return root;
    }
    if(value < root->data)
```

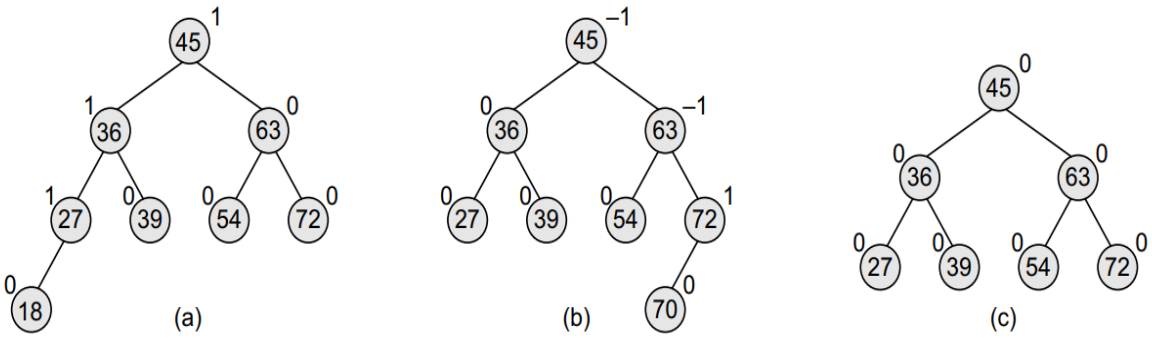
```
{
        root->lc = deletenode(root->lc,value);
}
else if(value > root->data)
{
        root->rc = deletenode(root->rc,value);
}
else
{

if(root->lc==NULL && root->rc==NULL) //case 1: leaf node
{
        free(root);
        root=NULL;
}
else if(root->lc==NULL) //case 2: one child
{
        tree *temp=root;
        root=root->rc;
        free(temp);
}
else if(root->rc==NULL) //case 2: one child
{
        tree *temp=root;
        root=root->lc;
        free(temp);
}
else  //case 3: Two children
{
        tree *temp = findmin(root->rc);
```

```
            root->data = temp->data;
            root->rc = deletenode(root->rc,temp->data);
        }

        }
        return root;
}

int main(int argc, char *argv[]) {
        tree *root=NULL;
        int ch, value,i,n;
        do
        {
            printf("\n 1. creating a binary search tree\n");
            printf("\n 2. Inorder traversal\n");
            printf("\n 3. Pre order traversal\n");
            printf("\n 4. Post order traversal\n");
            printf("\n 5. Inserting a node to the binary search tree\n");
            printf("\n 6. Deleting a node from the binary search tree\n");
            printf("\n 7. Exit\n");
            printf("\n Enter the correct choice\n");
            scanf("%d",&ch);
            switch(ch)
            {
                case 1: printf("\n Enter number of nodes\n");
                        scanf("%d",&n);
                        for(i=1;i<=n;i++)
                        {
                            printf("\n Enter the value to be inserted
into the tree\n");
```

```
                        scanf("%d",&value);
                        root=create_tree(root,value);
                    }
                    break;
            case 2: inorder(root);
                    break;
            case 3: preorder(root);
                    break;
            case 4: postorder(root);
                    break;
            case 5: printf("\n Enter the node to be inserted into the
tree");
                    scanf("%d",&value);
                    root = insert(root,value);
                    break;
            case 6: printf("\n Enter the node to be deleted from the
tree");
                    scanf("%d",&value);
                    root = deletenode(root,value);
                    break;
            case 7: exit(0);
            default: printf("\n Enter the correct choice\n");
        }
    }while(1);
    return 0;
}
```

AVL (Addson-Velsky-Landis) Tree

- It is a height-balanced binary search tree.
- The height of two subtrees of a node may differ by at most one.
- Advantage: it takes O(logn) time to perform a search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to O(logn).
- Height-balanced tree: A binary tree in which every node has a balance factor of -1, 0, or 1 is said to be a height-balanced tree.
- Balance factor = Height of left subtree – Height of right subtree
- If the balance factor of a node is 1, then it means that the left subtree of the tree is one level higher than that of the right subtree. Such a tree is called a left-heavy tree.
- If the balance factor of a node is 0, then it means that the height of the left subtree is equal to the height of the right subtree.
- If the balance factor of a node is -1, then it means that the left subtree of the tree is one level lower than that of the right subtree. Such a tree is called a right heavy tree.



(a) Left-heavy AVL tree (b) right-heavy tree (c) balanced tree

Operations on AVL Trees

There are 3 operations on AVL trees, which are

1. Search

2. Insertion

3. Deletion.

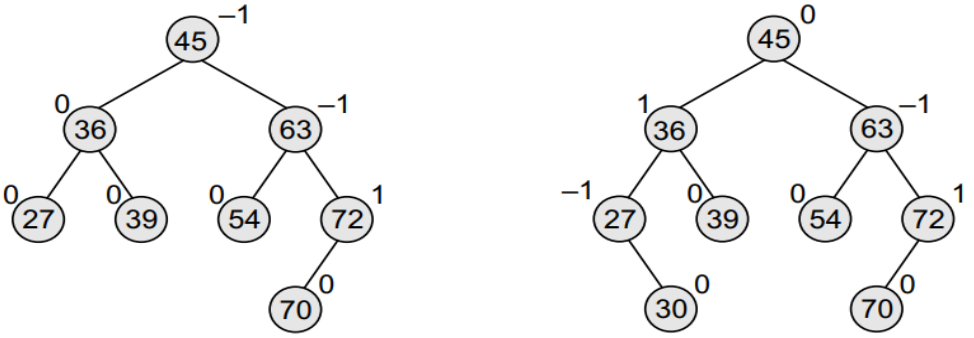
Searching for a node in an AVL tree

Searching is similar to the search of a node in the binary search tree. Due to the height balancing of the tree, the search operation takes O(logn) time to compute. Since this operation doesn't modify the structure of the tree.

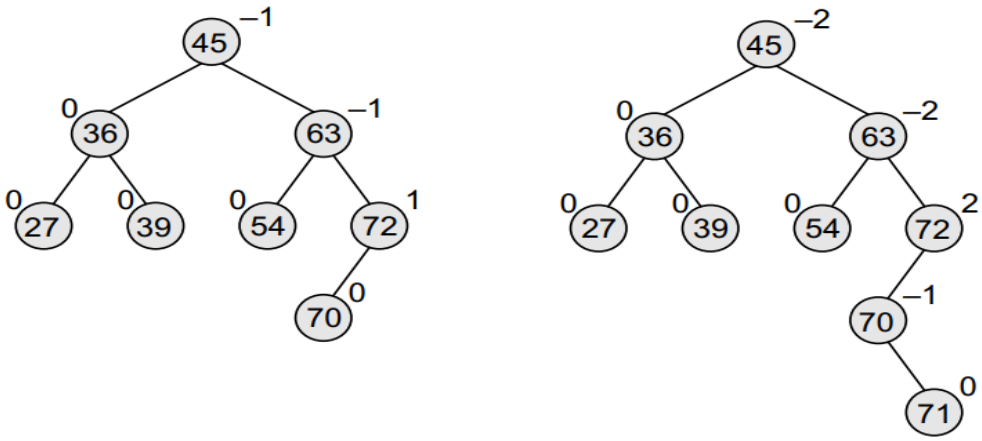
Insertion:

- The new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation.
- Rotation is done to restore the balance of the tree. However, if the insertion of the new node doesn't disturb the balance factor, that is if the balance factor of every node is still -1, 0, or 1 then rotations are not required.
- During the insertion of a new node, some node's balance factors will change are those nodes which lie in the path between the root of the tree and the newly inserted node. The possible changes which may take place in any node on the path are as follows.

  1. Initially, the node was either left or right heavily, and after insertion it becomes balanced.
  2. Initially, the node was balanced and after insertion, it becomes left or right-heavy.
  3. Initially, the node was heavy (left or right) and the new node has been inserted in the heavy subtree, thereby creating an unbalanced subtree. Such a node is said to be a critical node.



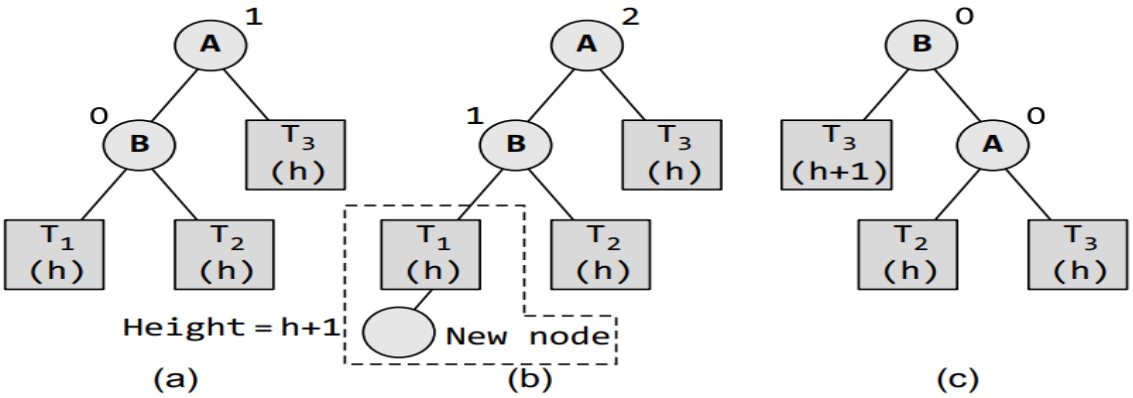
Inserting a new node 71 into the tree makes the tree as unbalance. To make it as balanced we can apply rotations.



Rotations:

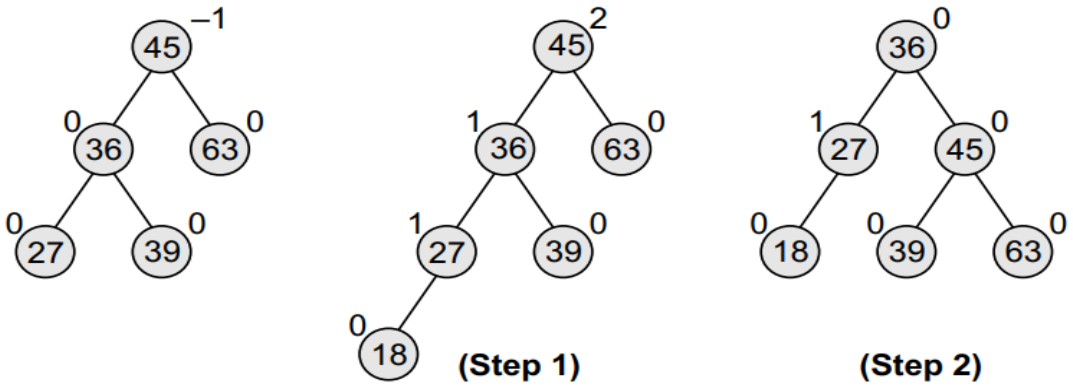
- Rotations are used to make the tree balanced.
- To perform the rotation, first, identify the critical node.
- **Critical node:** Critical node is the nearest ancestor node on the path from the inserted node to the root whose balance factor is neither –1, 0, nor 1.
- **We have 4 types of rotations.** They are LL rotation, RR rotation, LR rotation, and RL rotation.
- **LL Rotation:** The new node is inserted in the left subtree of the left subtree of the critical node.
- **RR Rotation:** The new node is inserted in the right subtree of the right subtree of the critical node.
- **LR Rotation:** The new node is inserted in the right subtree of the left subtree of the critical node.
- **RL Rotation:** The new node is inserted in the left subtree of the right subtree of the critical node.

LL Rotation:

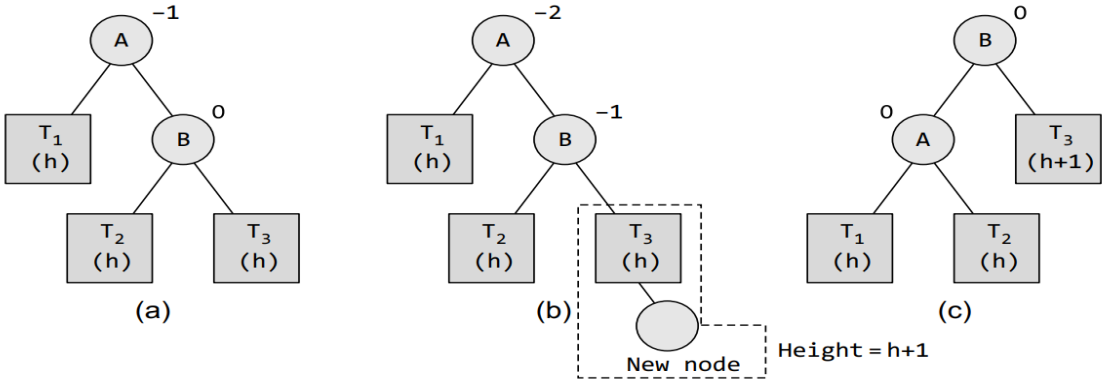


Tree (a) is an AVL tree. In tree (b), a new node is inserted in the left sub-tree of the left sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1), so we apply LL rotation as shown in tree (c). While rotation, node B becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.

Example: inserting node 18 into the tree

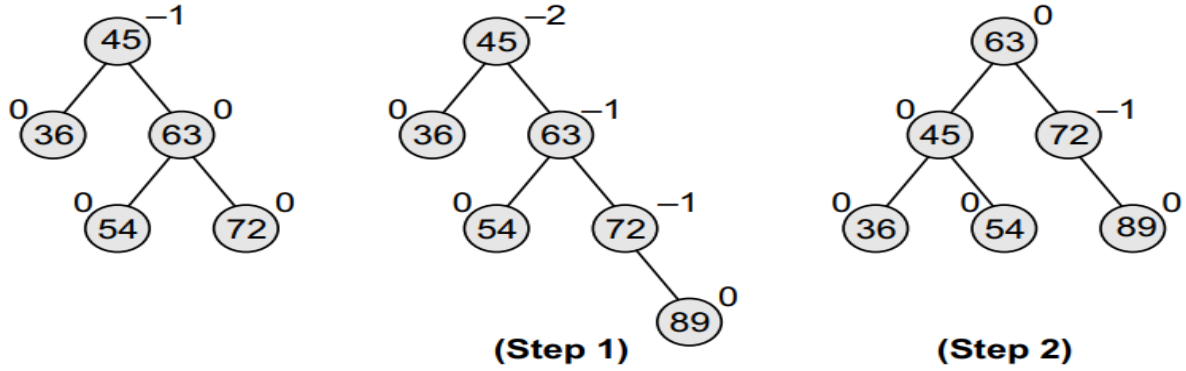


RR Rotation:

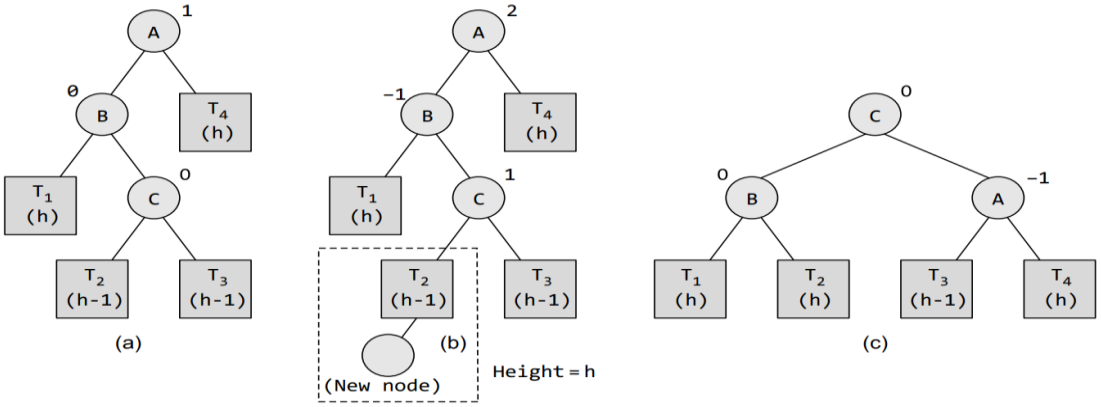


Tree (a) is an AVL tree. In tree (b), a new node is inserted in the right sub-tree of the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1), so we apply RR rotation as shown in tree (c). Note that the new node has now become a part of tree T3. While rotation, node B becomes the root, with A and T3 as its left and right child. T1 and T2 become the left and right sub-trees of A.

Example: Inserting node 89 into the tree

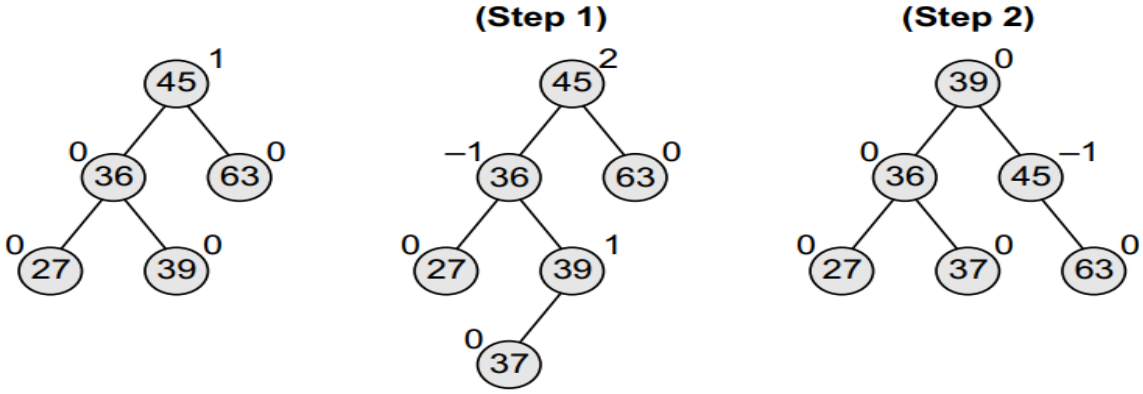


LR Rotation:

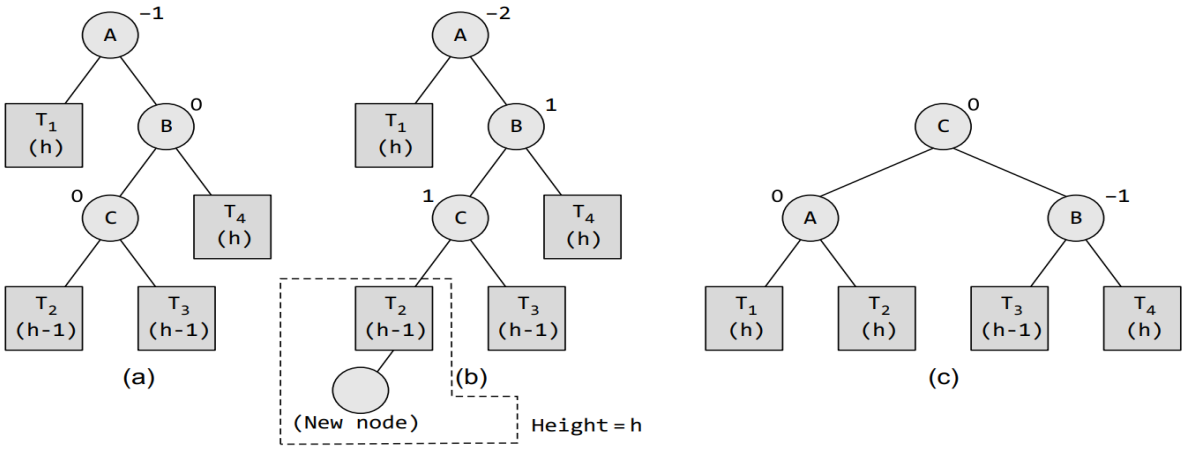


Tree (a) is an AVL tree. In tree (b), a new node is inserted in the right sub-tree of the left sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0 or 1), so we apply LR rotation as shown in tree (c). Note that the new node has now become a part of tree T2. While rotation, node C becomes the root, with B and A as its left and right children. Node B has T1 and T2 as its left and right sub-trees and T3 and T4 become the left and right sub-trees of node A.

Example: inserting node 37 into the tree



RL Rotation:



Tree (a) is an AVL tree. In tree (b), a new node is inserted in the left sub-tree of the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1), so we apply RL rotation as shown in tree (c). Note that the new node has now become a part of tree T2. While rotation, node C becomes the root, with A and B as its left and right children. Node A has T1 and T2 as its left and right sub-trees and T3 and T4 become the left and right sub-trees of node B.

Exercises

- 1. Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81
- Construct an AVL tree having the following elements: **H, I, J, B, A, E, C, F, D, G, K, L**
- Construct an AVL tree for the following sequence of numbers: 20, 11, 5, 32, 40, 2, 4, 27, 23, 28, 50.

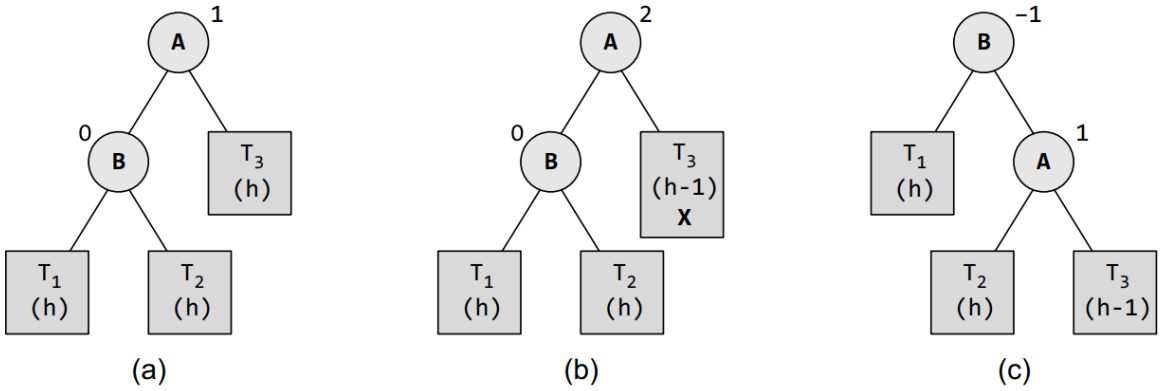
Deletion

- If the node to be deleted is a leaf node, it is simply removed from the tree.
- If the node to be deleted has one child node, the child node is replaced with the node to be deleted simply.
- If the node to be deleted has two child nodes, then,
    - Either replace the node with its in-order predecessor, i.e., the largest element of the left sub-tree.
    - Or replace the node with its inorder successor, i.e, the smallest element of the right subtree
- Deletion may disturb the AVLness of the tree, so to rebalance the AVL tree we need to perform rotations (R and L rotation).
- If the node to be deleted is present in the left subtree, then L rotation is applied.
- If the node is to be deleted in present in the right subtree, then R rotation is applied.
- There are 3 types of L and R rotations.
- The variations of L rotations are L-1, L0, and L1, correspondingly R rotations are R-1, R0, and R1.

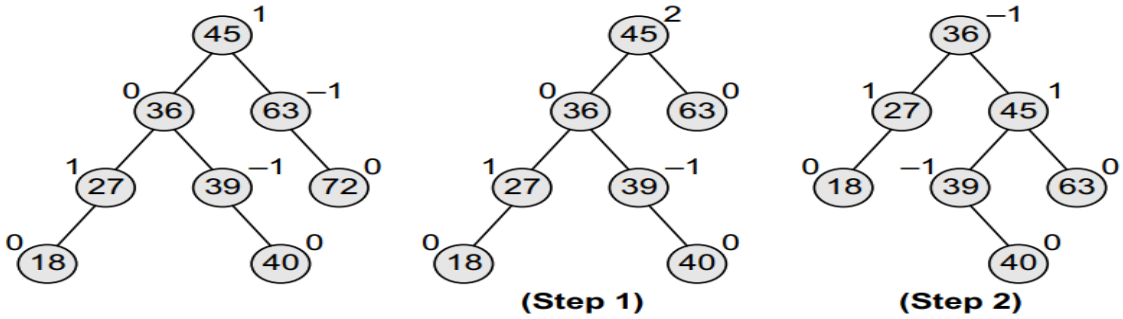
R0 Rotation

Tree (a) is an AVL tree. In tree (b), node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1). Since the balance factor of

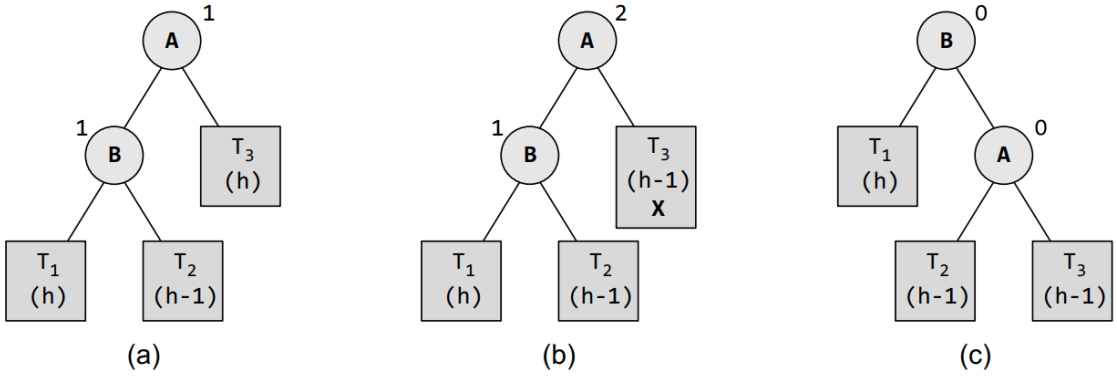
node B is 0, we apply R0 rotation as shown in tree (c). During the process of rotation, node B becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A



Example: Delete node 72 from the tree

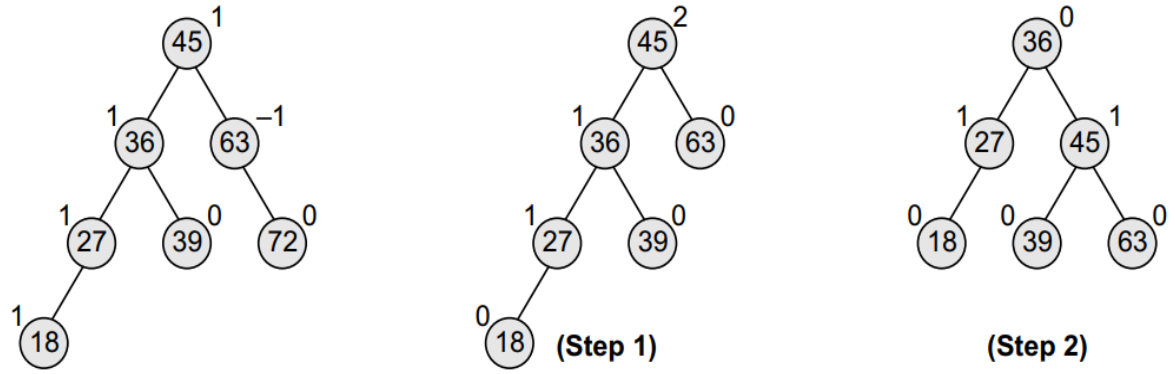


R1 Rotation

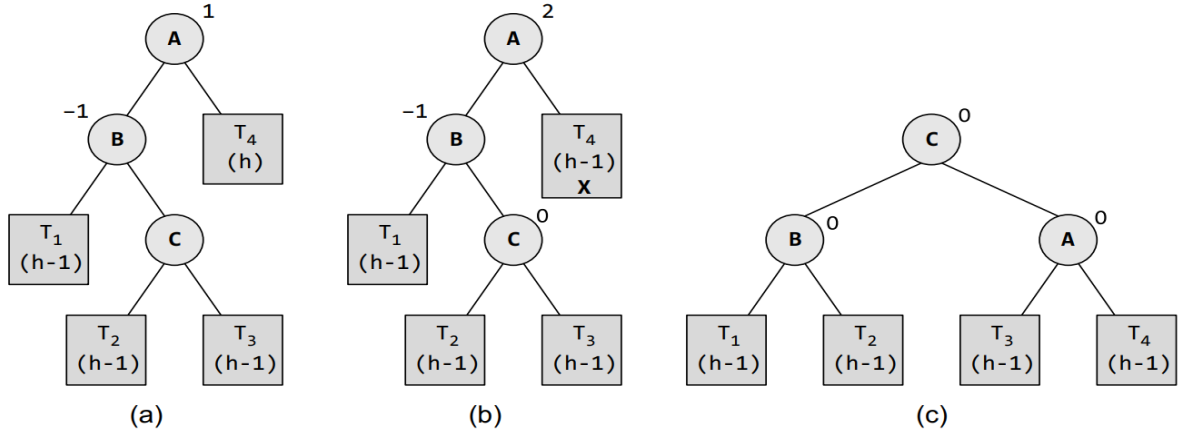


Tree (a) is an AVL tree. In tree (b), node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1). Since the balance factor of node B is 1, we apply R1 rotation as shown in tree (c). During the process of rotation, node B becomes the root, with T1 and A as its left and right children. T2 and T3 become the left and right sub-trees of A

Example: Delete node 72 from the tree.

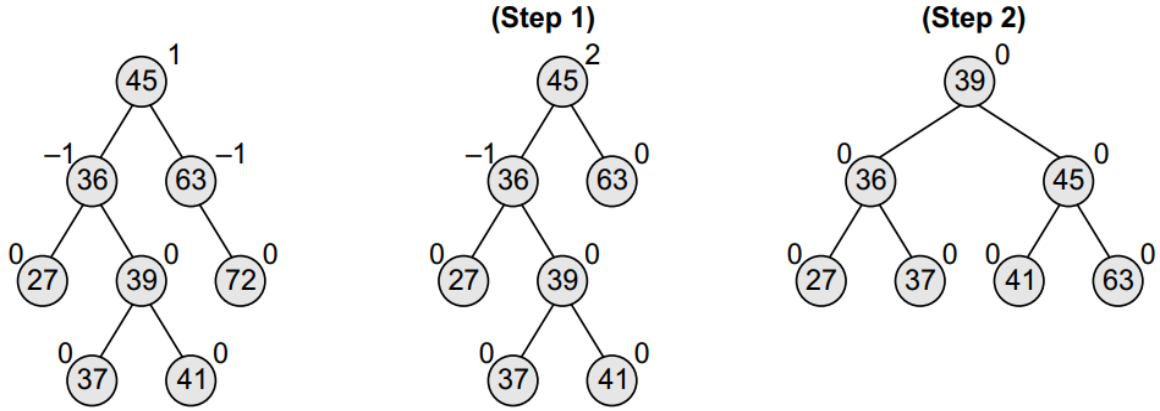


R-1 Rotation:



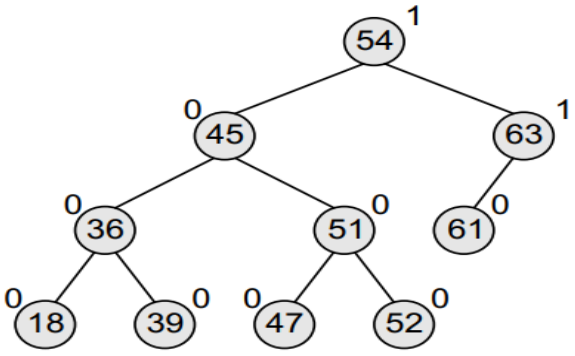
Tree (a) is an AVL tree. In tree (b), node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0 or 1). Since the balance factor of node B is –1, we apply R–1 rotation as shown in tree (c). While rotation, node C becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.

Example: Delete a node 72 from the tree

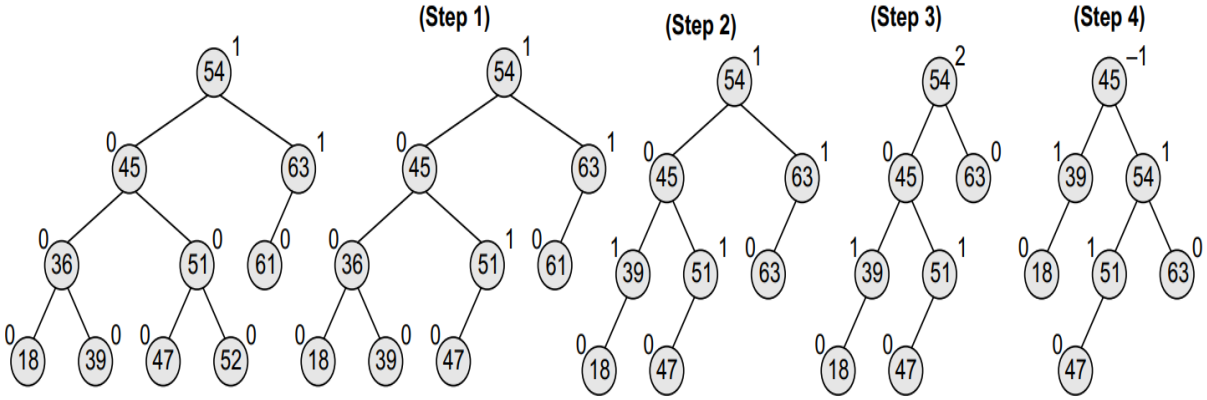


Exercises:

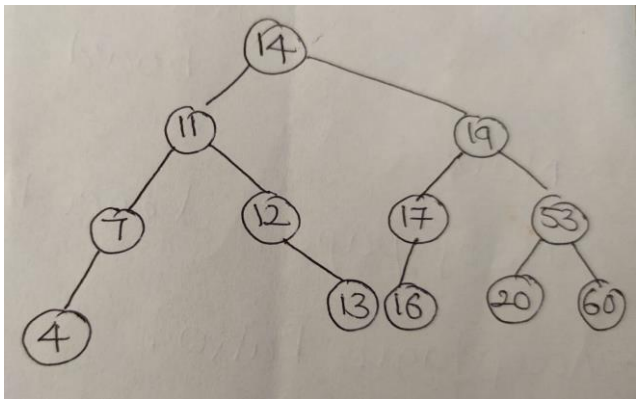
1. Delete the nodes 52, 36, and 61 from the AVL tree given in Figure.



Ans:

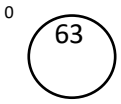


2. Delete the nodes 7, 11, 14 and 17 from the AVL tree

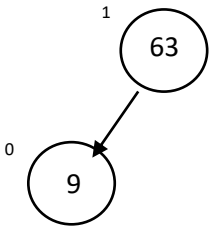


1. Construct an AVL tree by inserting the following elements in the given order –
   63, 9, 19, 27, 18, 108, 99, 81

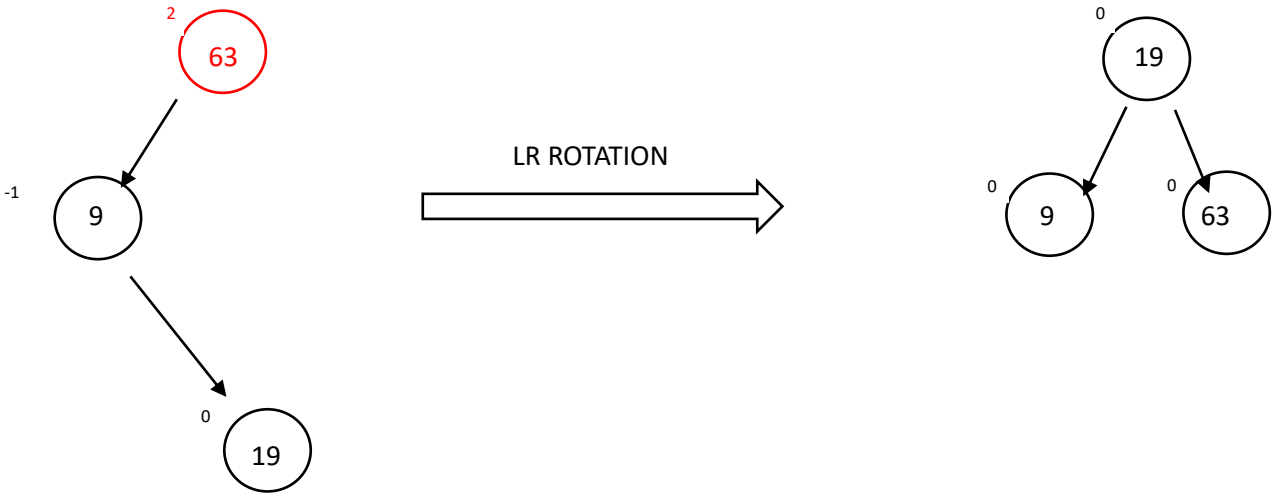
Step 1: insert 63



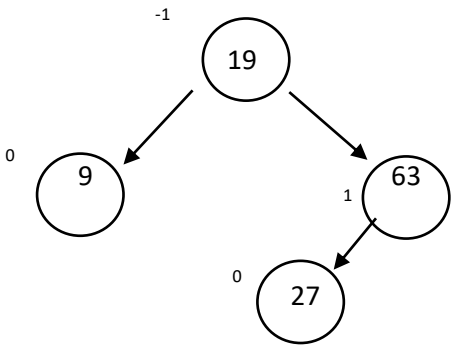
Step 2: insert 9



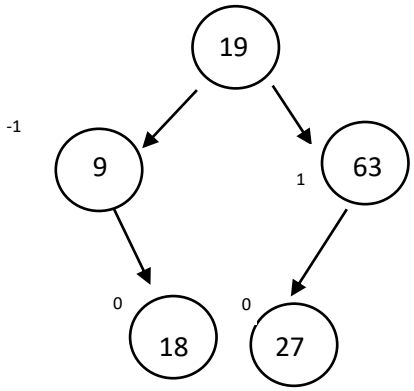
Step 3: insert 19



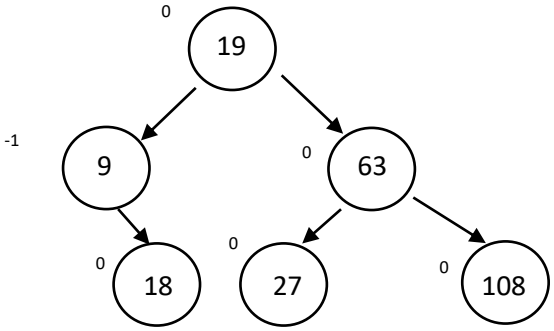
Step 4: insert 27



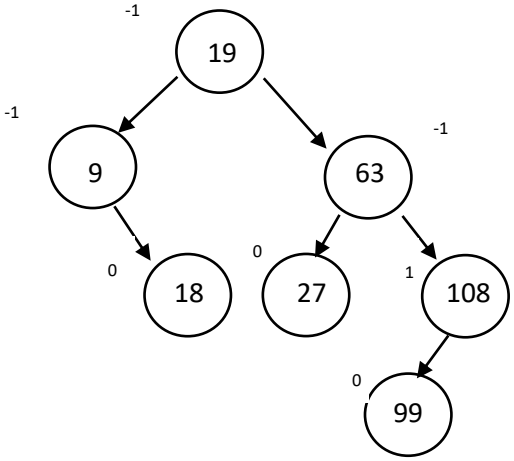
Step 5: insert 18



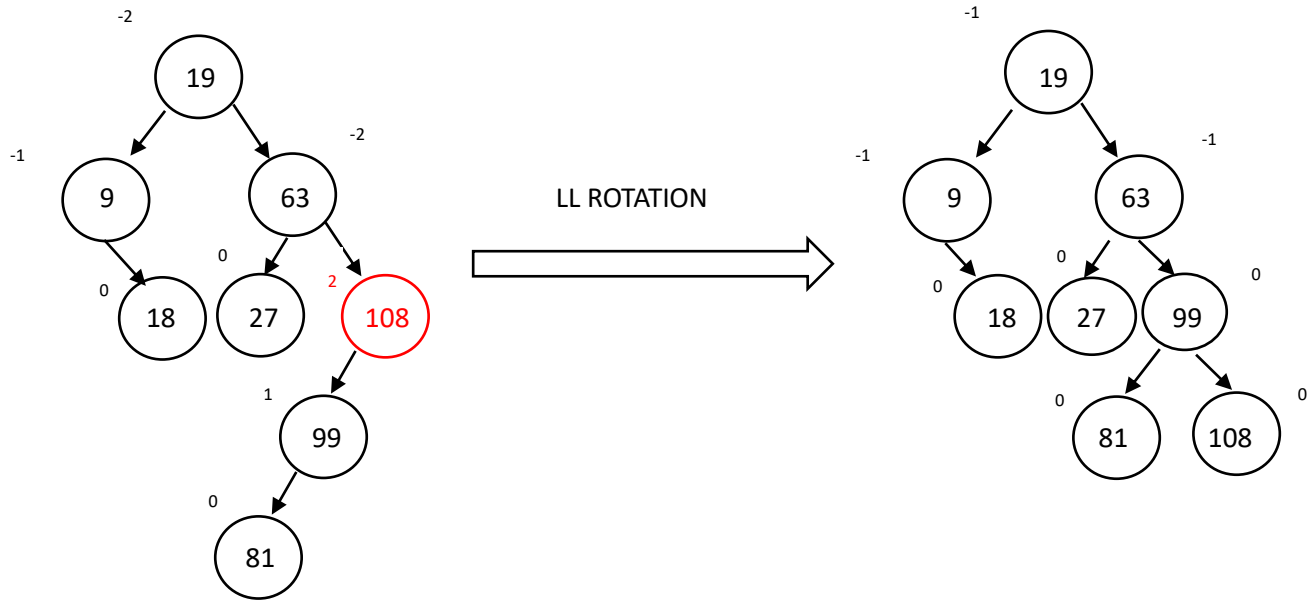
Step 6: insert 108



Step 7: insert 99

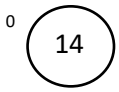


Step 8: insert 81

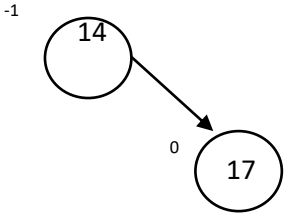


2. Construct an AVL tree by inserting the following elements in the given order –
   14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

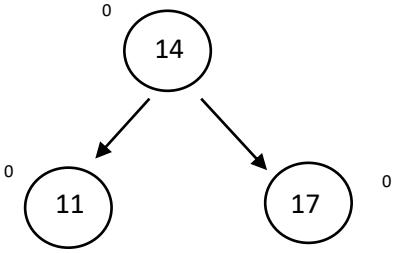
Step 1: insert 14



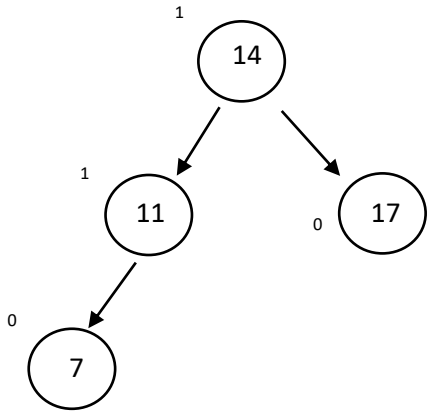
Step 2: insert 17



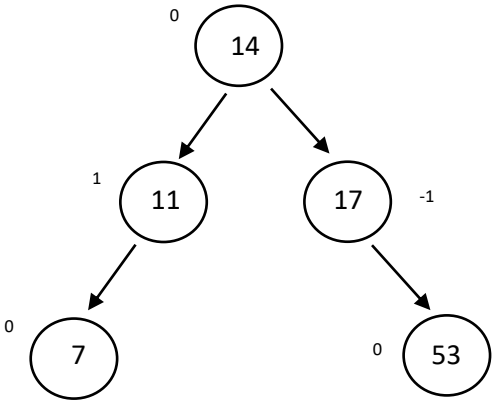
Step 3: insert 11



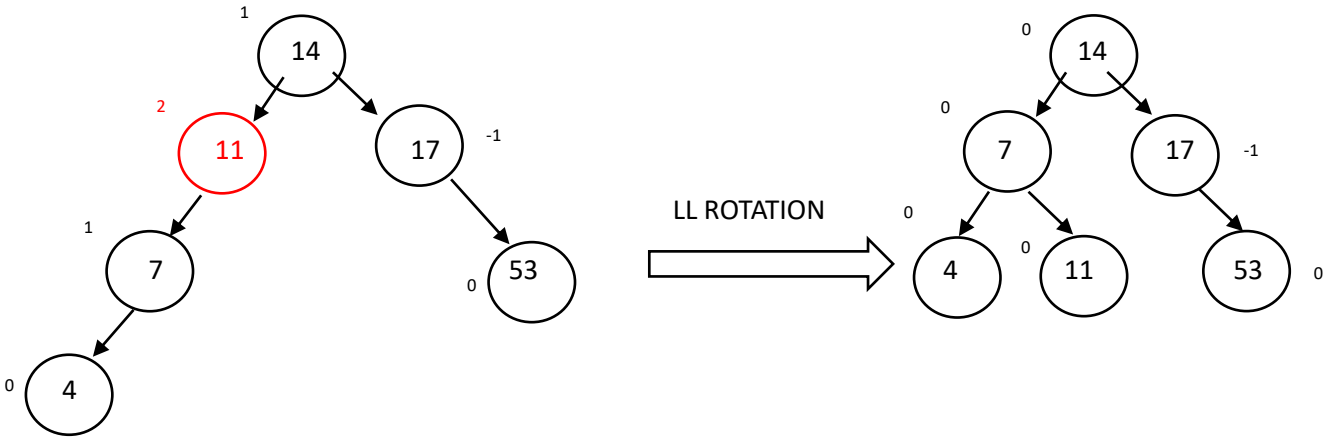
Step 4: insert 7



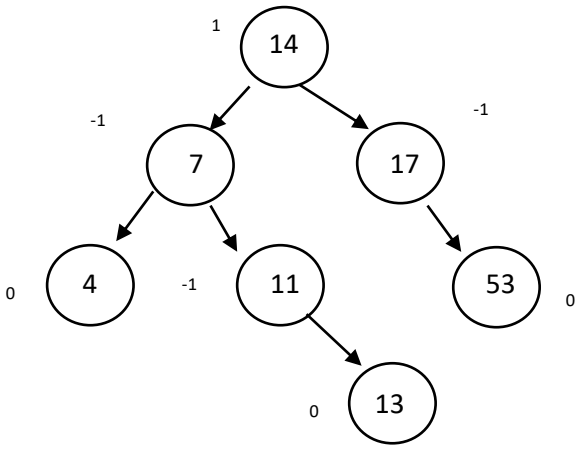
Step 5: insert 53



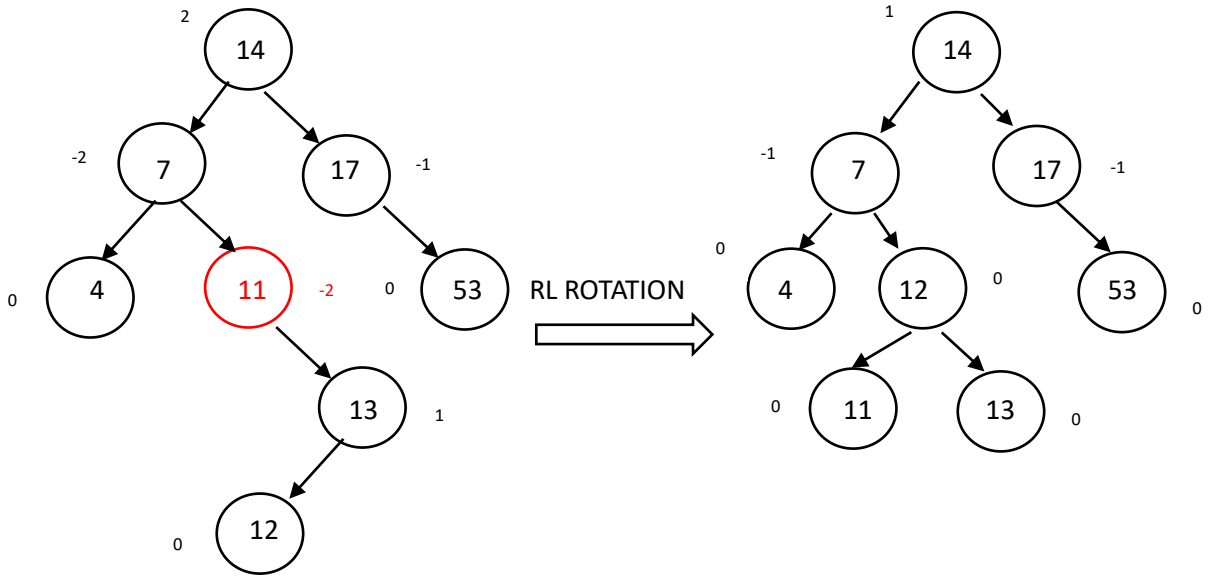
Step 6: insert 4



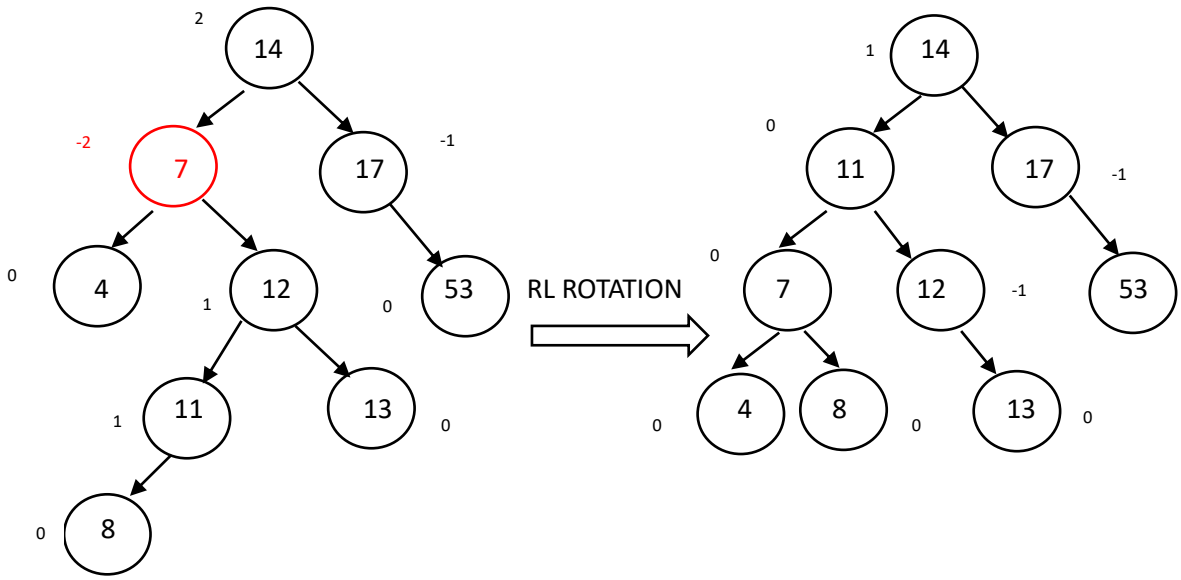
Step 7: insert 13



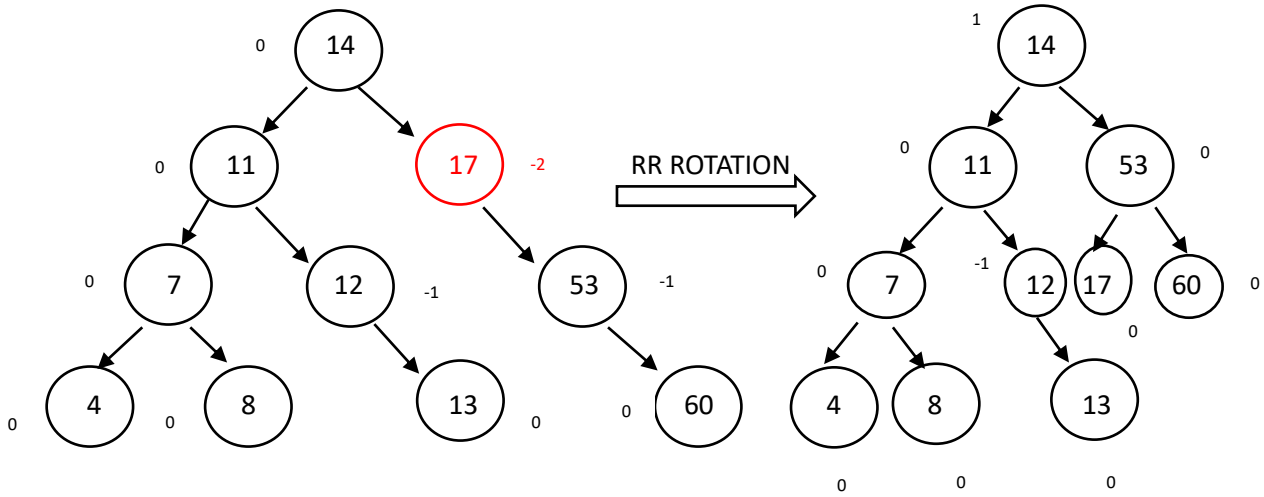
Step 8: insert 12



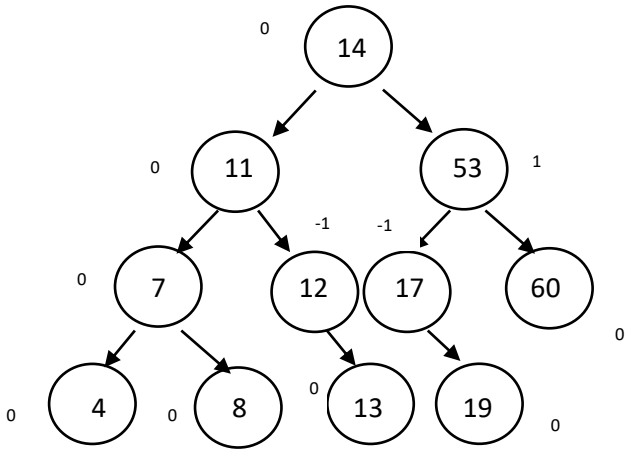
Step 9: insert 8



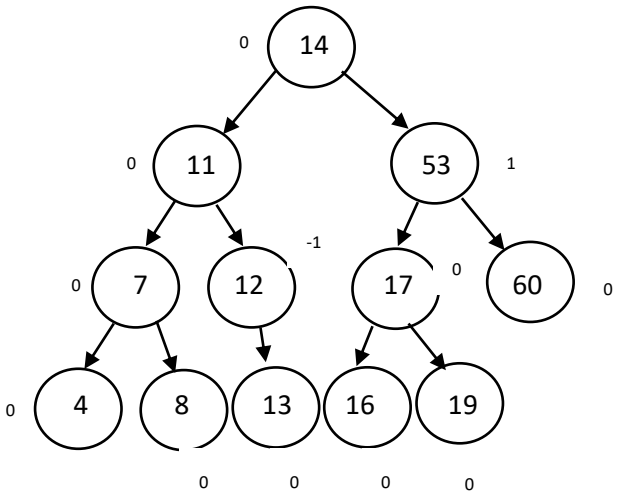
Step 10: insert 60



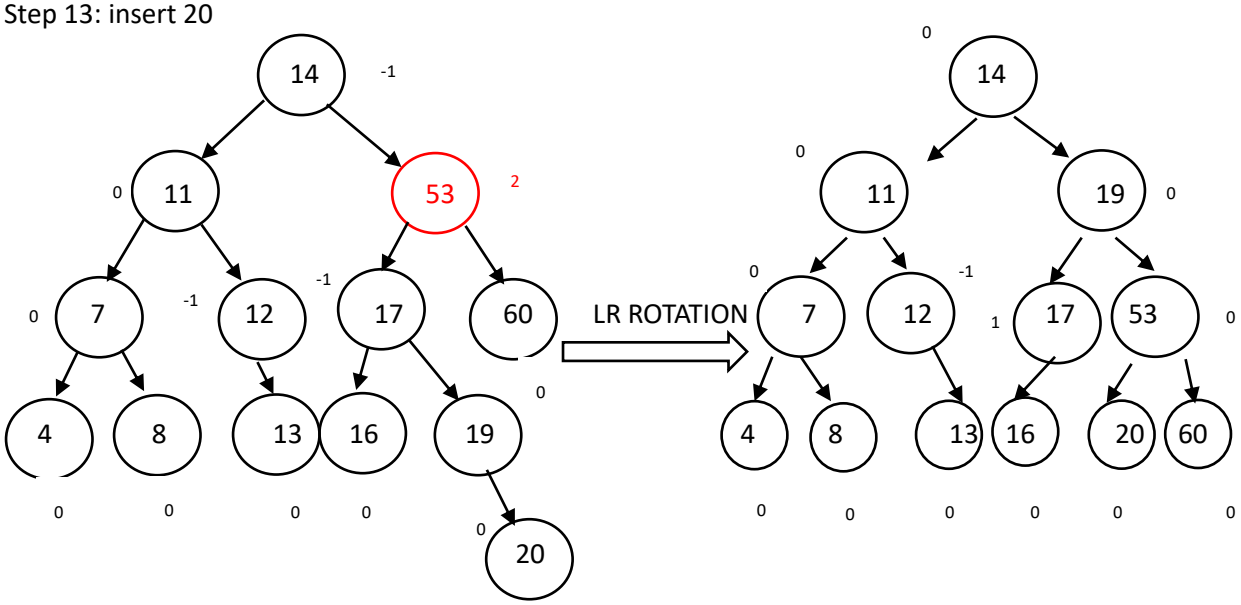
Step 11: insert 19



Step12: insert 16

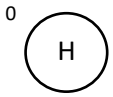


Step 13: insert 20

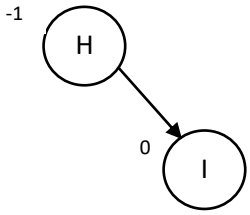


3. Construct an AVL tree by inserting the following elements in the given order – H, I, J, B, A, E, C, F, D, G, K, L.

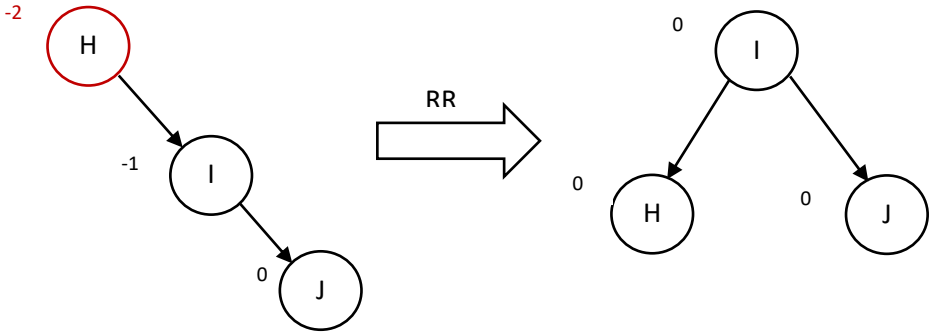
A) Step 1: Insert H



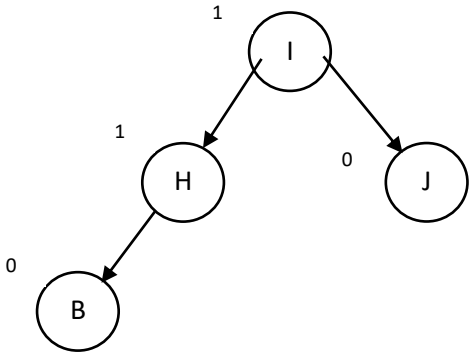
Step 2: Insert I



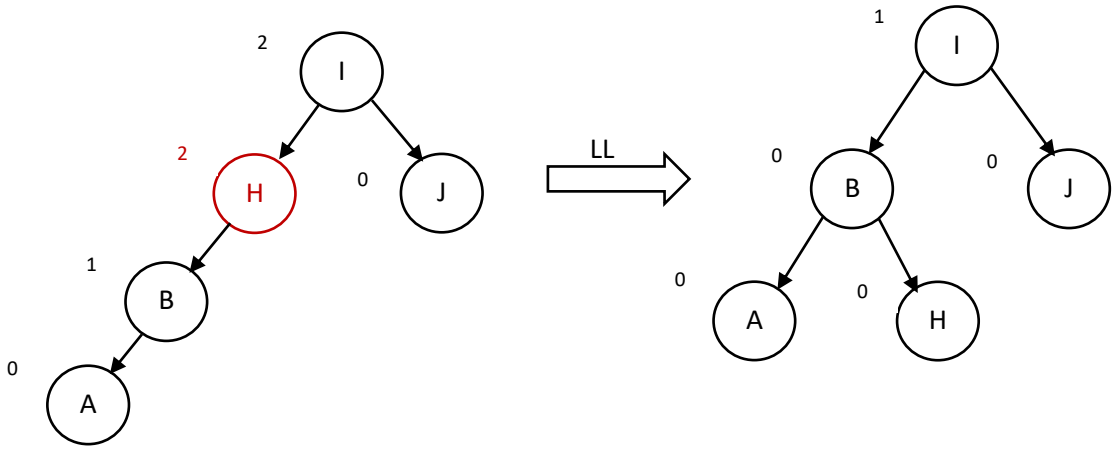
Step 3: Insert J



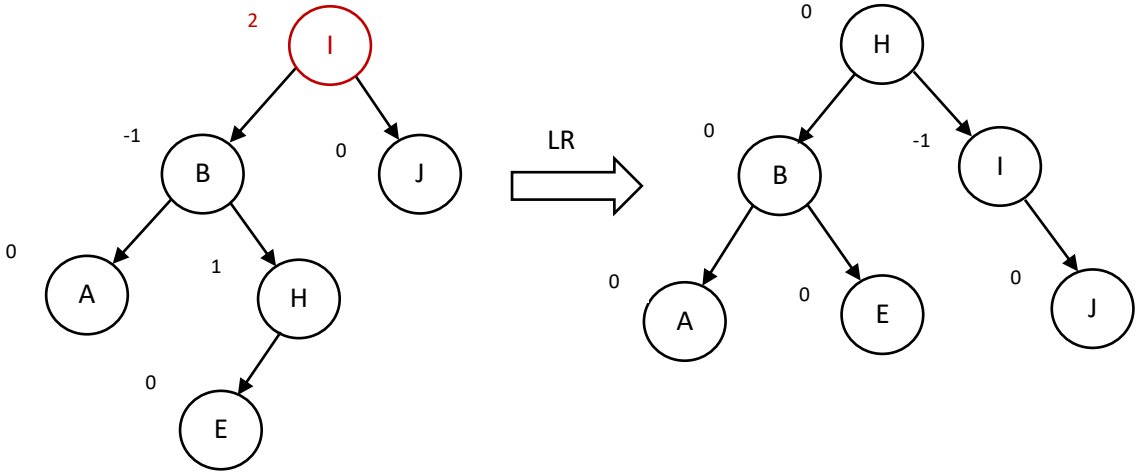
Step 4: Insert B



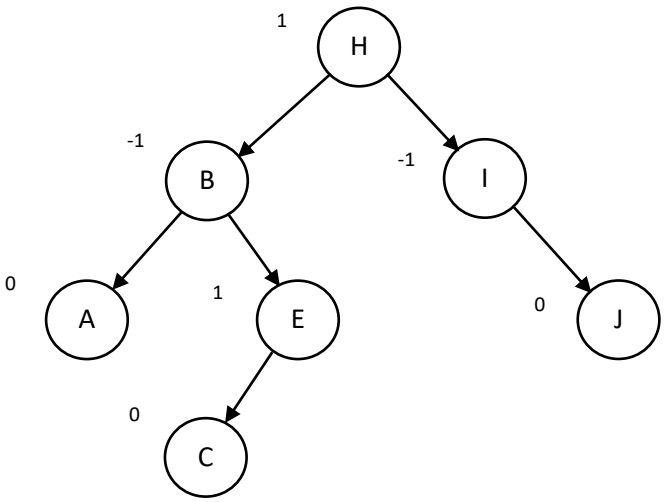
Step 5: Insert A



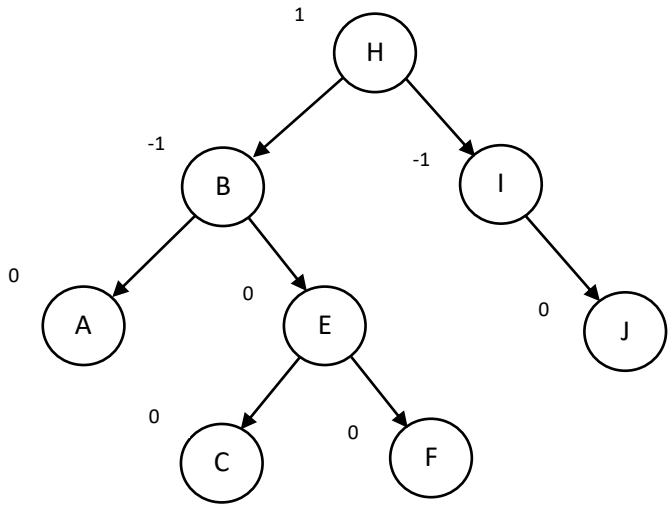
Step 6: Insert E



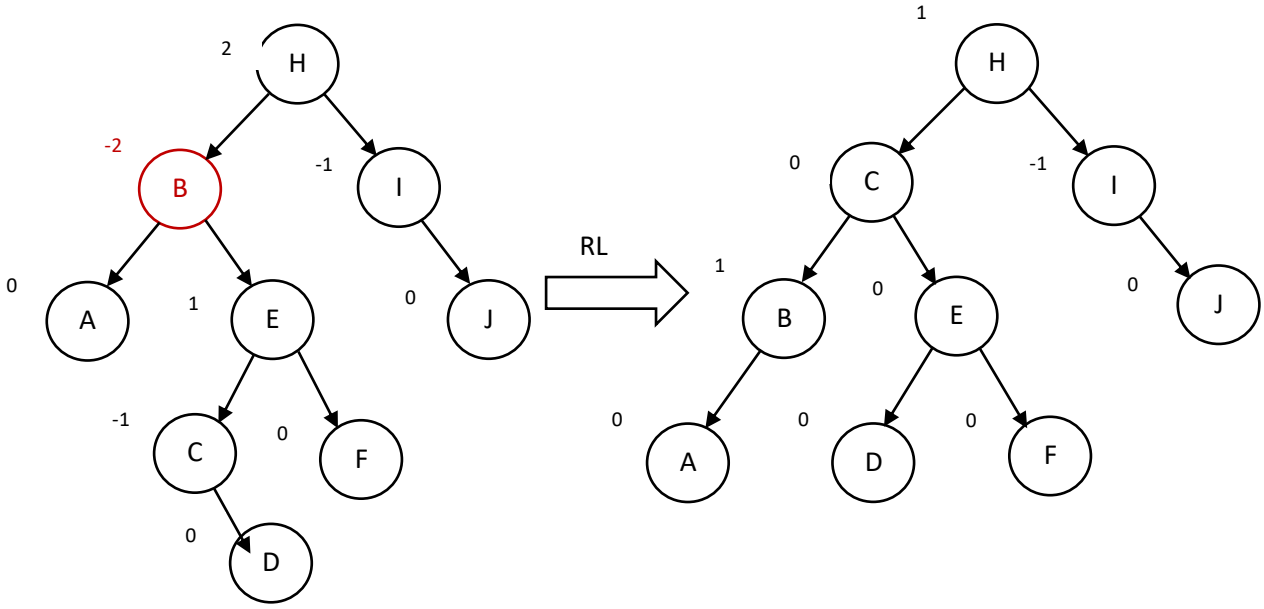
Step 7: Insert C



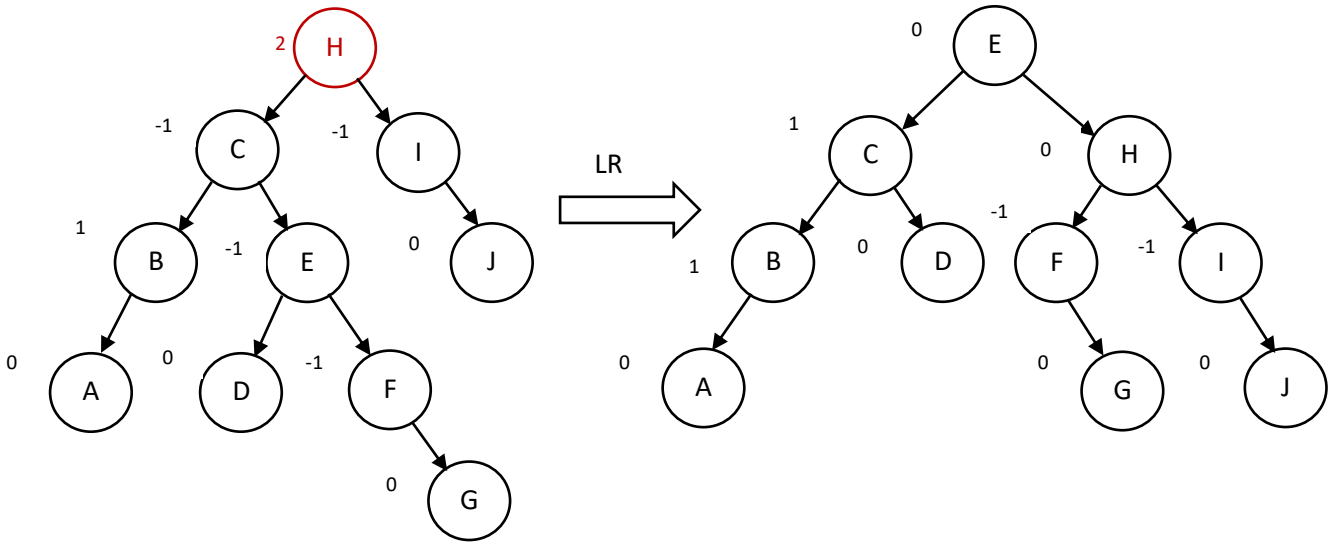
Step 8: Insert F



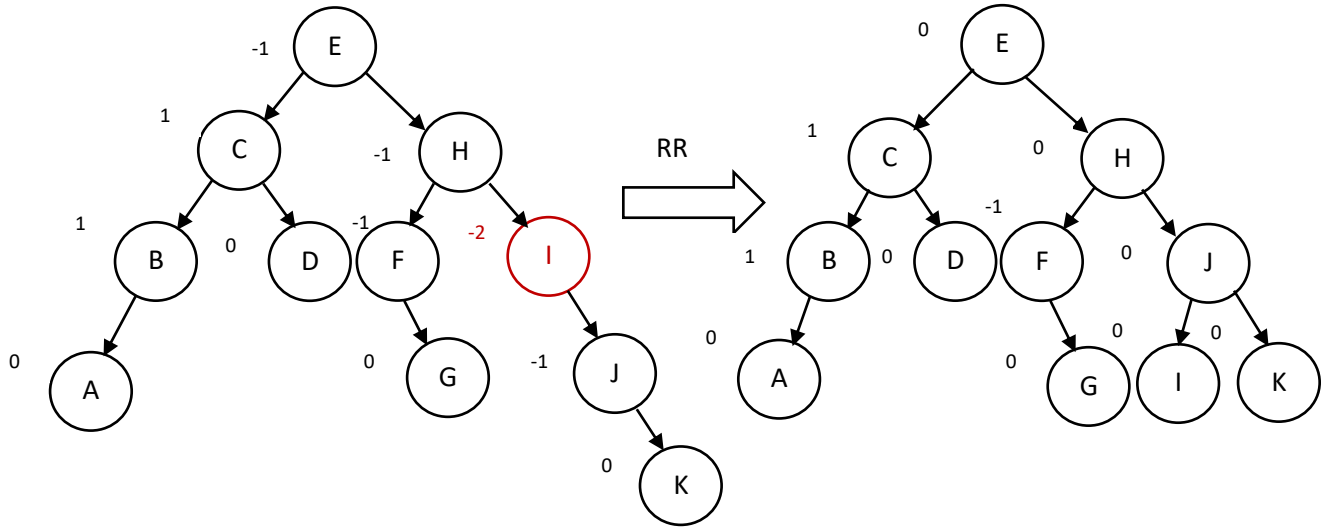
Step 9: Insert D



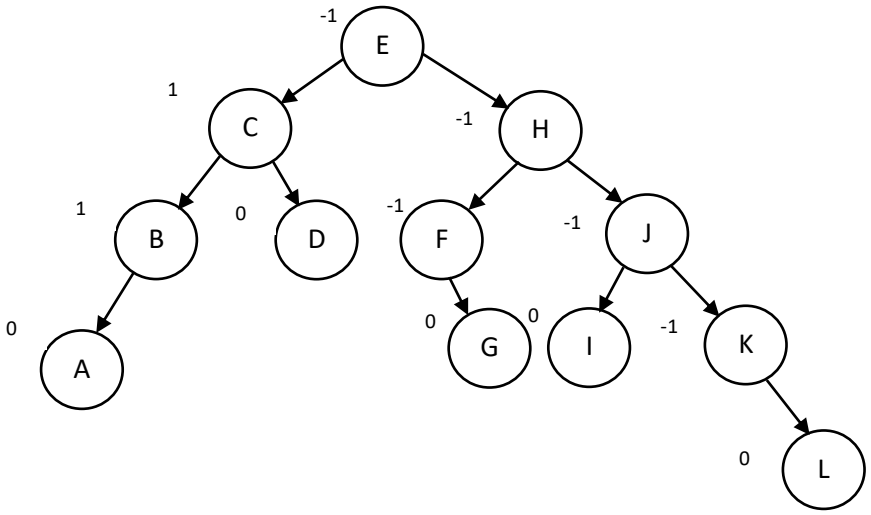
Step 10: Insert G



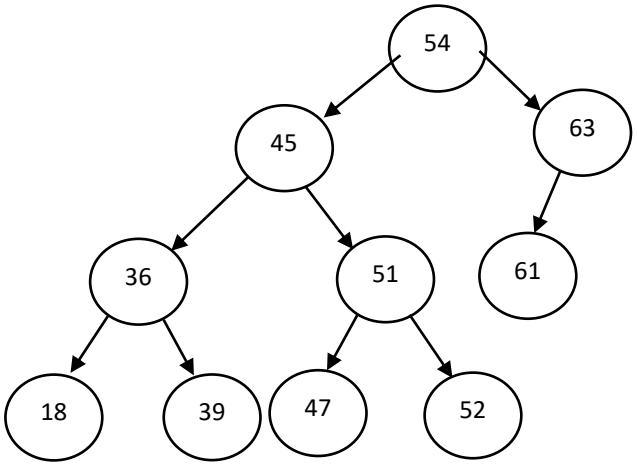
Step 11: Insert K



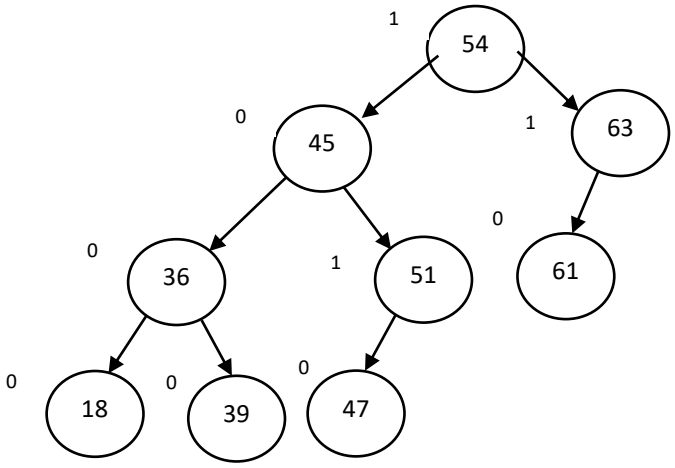
Step 12: Insert L



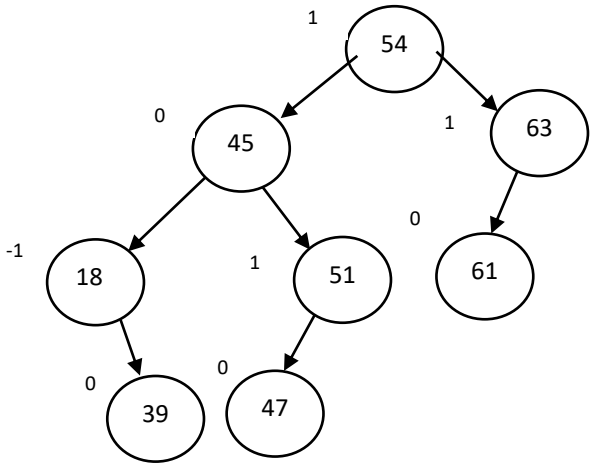
4. Delete the nodes 52, 36, 61 from the given AVL tree.



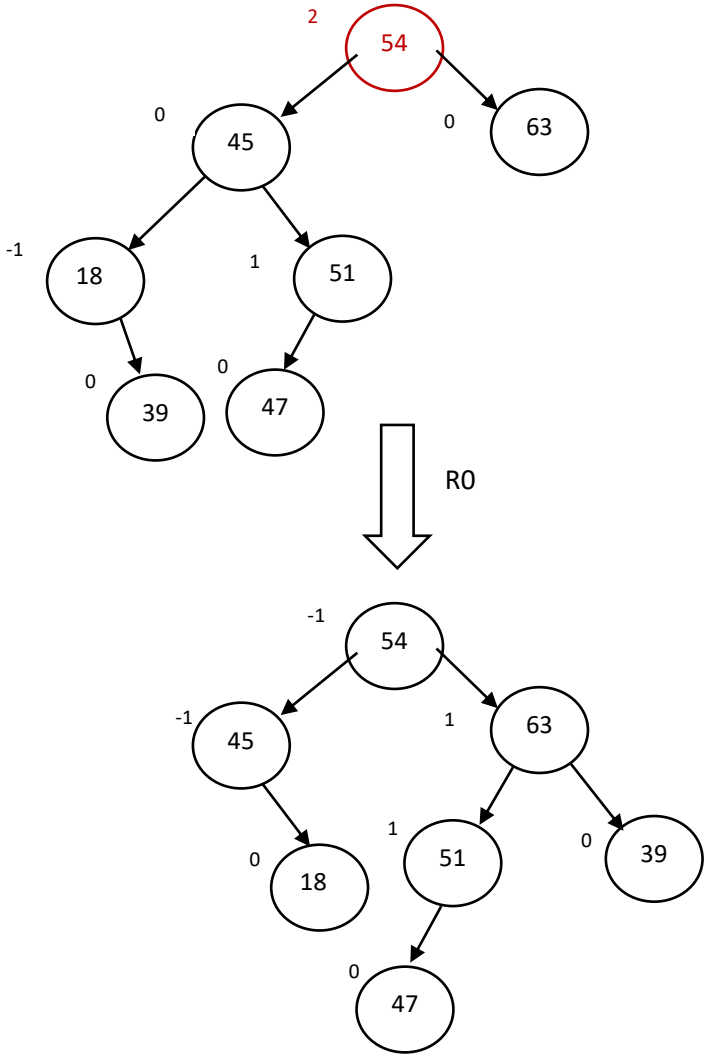
A) Step 1: Delete 52



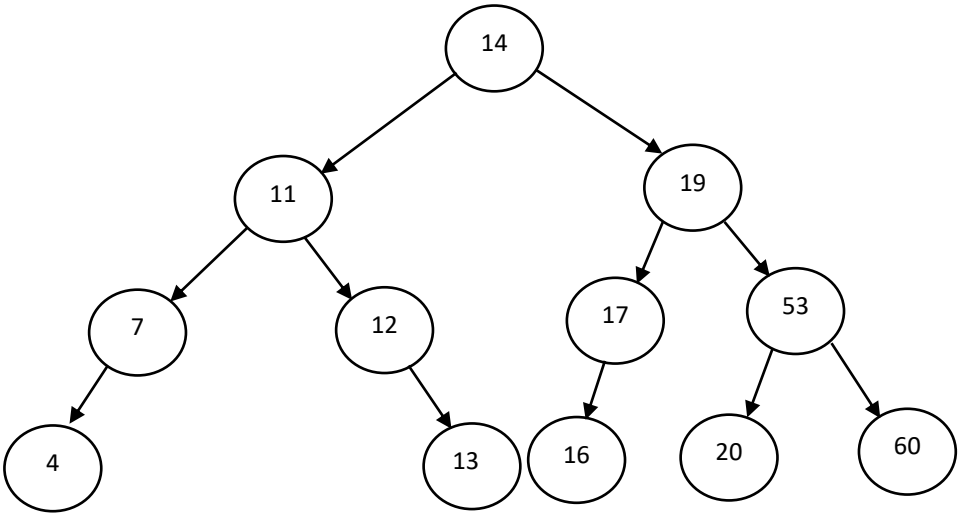
Step 2: Delete 36



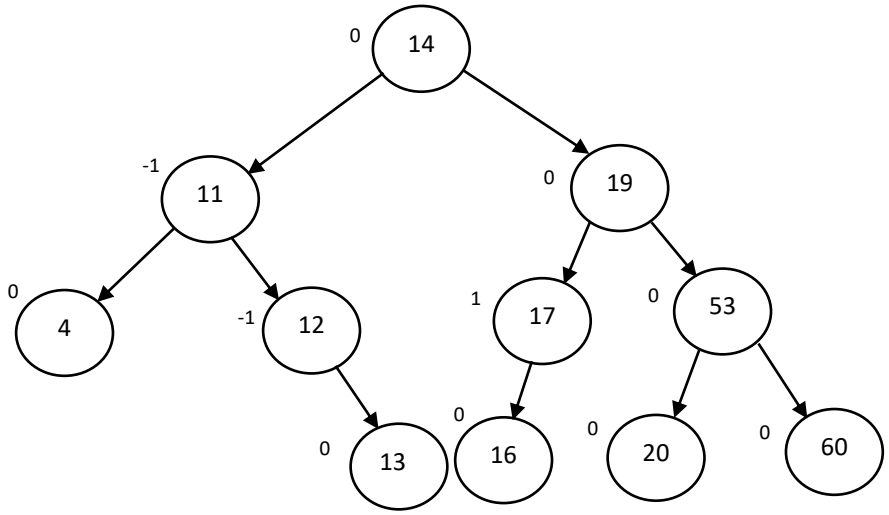
Step 3: Delete 61



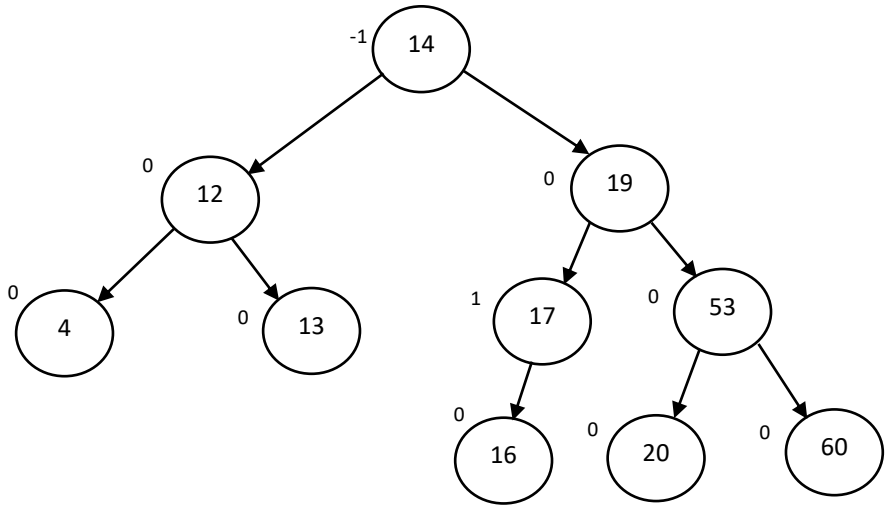
5. Delete the nodes 7, 11,14, 17 from the given AVL tree.



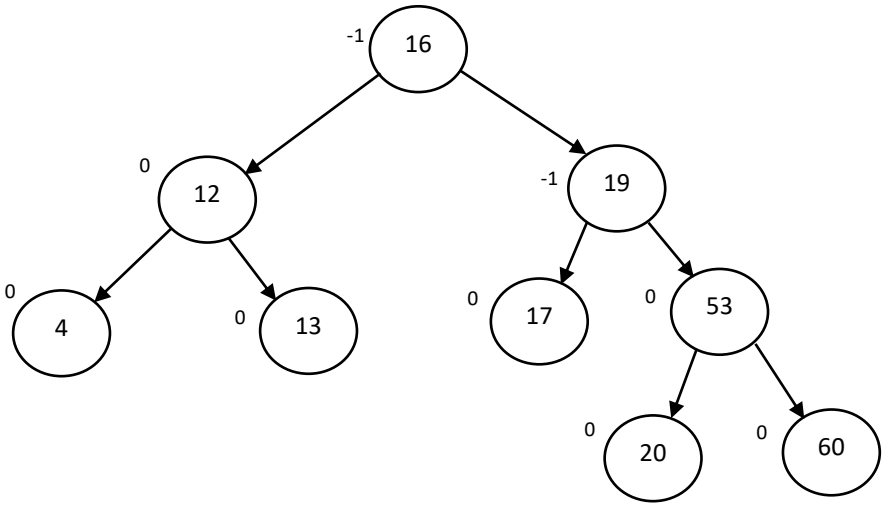
A) Step 1: Delete 7



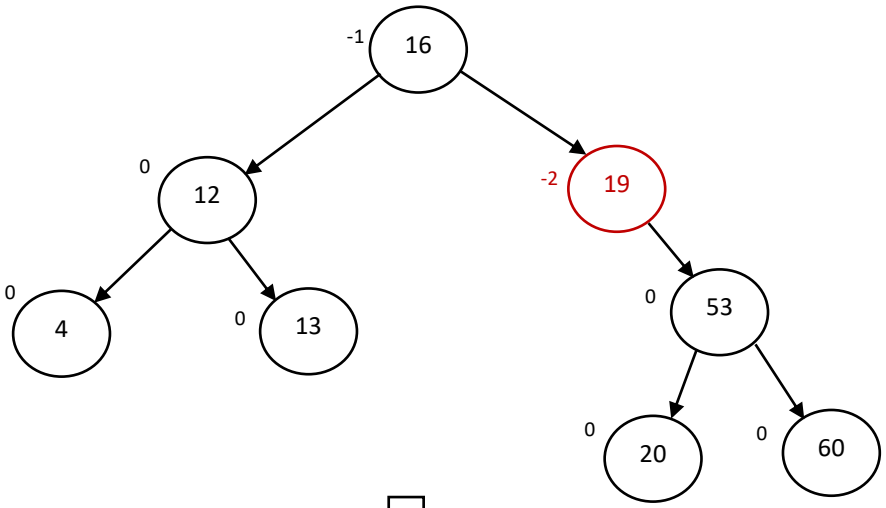
Step 2: Delete 11

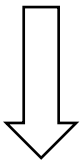


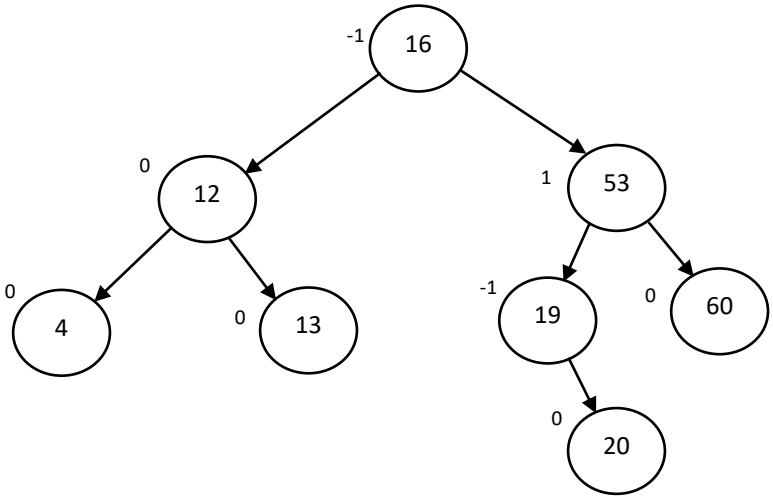
Step 3: Delete 14



Step 3: Delete 17

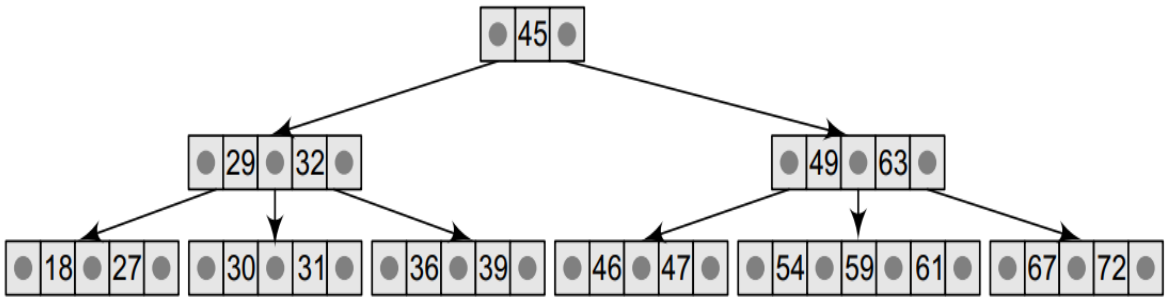






B-Trees

- A B-Tree is a specialized M-way tree, developed by Rudolf Bayer and Ed Mc Creight in 1970.
- It is widely used for disk access.
- A-B-tree of order M can have a maximum of m-1 keys and M pointers to subtrees. Storing many keys in a single node keeps the height of the tree relatively small.
- A B-tree is designed to store sorted data and allows search, insertion, and deletion operations to be performed in logarithmic time.
- A B-tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree. It has the following properties:
    a. Every node in the B-tree has at most m children.
    b. Every node in the B-tree except root and leaf nodes has at least m/2 children. (This condition helps to keep the tree bushy so that the path from the root node to the leaf is very short, even in a tree that stores a lot of data).
    c. The root node has at least two children if it is not a terminal node.
    d. All leaf nodes are at the same level.



Searching for an element in a B-Tree

Searching for an element in a B-Tree is like searching a node in the binary search tree.

Example: To search for 59, we begin at the root node. The root node has a value of 45 which is less than 59. So we traverse in the right subtree. The right subtree of the root node has two key values 49 and 63. Since 49<= 59 <= 63, we traverse the subtree of 49, that is the left subtree of 63. This left subtree has 3 values 54, 59, and 61. On finding the value 59, the search is successful.

The time complexity of the search operation is $O(\log_t n)$

Insertion

In a B-tree, all insertions are done at the leaf node level. A new value is inserted in the B-tree using the algorithm given below.

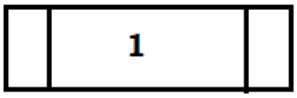
Algorithm:

1. Search the B-Tree to find the leaf node where the new key value should be inserted.
2. If the leaf node is not full, that is it contains less than m-1 values, then insert the new element in the node keeping the elements of the node ordered.
3. If the leaf node is full, that is, the leaf node already contains m-1 key values, then
    a. Insert the new value in order into the existing set of keys.
    b. Split the node as its median into the existing set of keys.
    c. Push the median element up to its parent node. If the parent's node is already full, then split the parent node by following the same steps.

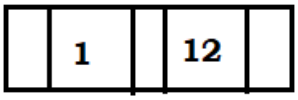
Example: Create a B-Tree of order 5 with the following elements: 1, 12, 8, 2, 25, 6, 14, 28, 17, 7, 52, 16, 48, 68, 3, 26, 29, 53, 55, 45, 67

Sol: Each node has 4 keys and 5 pointers.

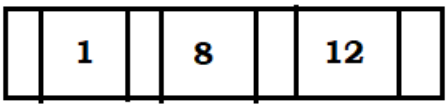
Step 1: Insert 1.



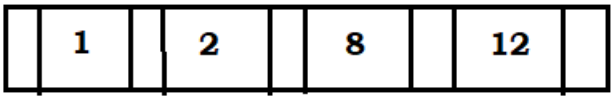
Step 2: Insert 12



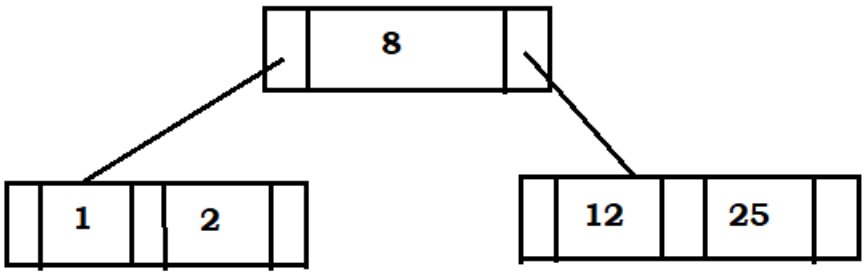
Step 3: Insert 8



Step 4: Insert 2



Step 5: Insert 25, Already node is full, now take the elements in order, it is 1, 2, 8, 12, and 25. Among this median is 8, hence 8 is included in the parent node and this node is split into two nodes. The key value 8 left subtree node contains the values 1and 2, and the right subtree node contains the values 12 and 25.



Example: Create a B-Tree of order 5 with the following elements: 3,14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19

Example: Create a B-Tree of order 5 with the following elements: 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

Deleting an element from a B-Tree

Like insertion, deletion is also done from the leaf node. There are two cases of deletion. In the first case, a leaf node has to be deleted. In the second case an internal node has to be deleted.

The following steps involved in a deletion of a leaf node.

1. locate the leaf node which has to be deleted.
2. If the leaf node contain more than minimum number of key values(more than m/2 elements), then delete the value.
3. Else if leaf node does not contain m/2 elements then fill the node by taking an element either from the left or from the right sibling.
    a. If the left sibling has more then the minimum number of key values, push its largest key into its parents node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
    b. Else, if the right sibling has more than minimum number of key values, push its smallest key values into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
4. Else if both the left and right siblings contain only the minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node. (ensuring that the number of elements a node can have i.e. m). if pulling the intervening element from the parent node leaves it with less than the minimum number of keys in the node. Then propagates the process upwards, there by reducing the height of the B-tree.

To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. This predecessor or

successor will always be in the leaf node. So the processing will be done as if a value from the leaf node has been deleted.

UNIT-IV

Topics:

1. Priority Queue.
2. Quick Sort
3. HeapSort
4. Merge Sort
5. Radix sort

1. Priority Queue

Definition: Apriority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed.

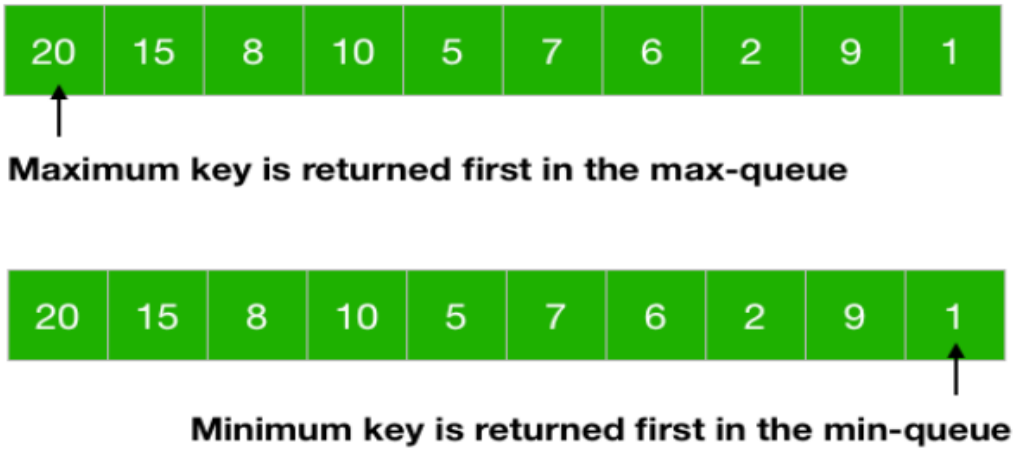
The general rules of processing the elements of a priority queue are:

1. An element with higher priority is processed before an element with a lower priority.
2. Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

Priority queues are widely used in operating systems to execute the highest priority process first. The priority of the process may be set based on the CPU time it requires to get executed completely. For example, if there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed.

Priority queue is a type of queue in which every element has a key associated to it and the queue returns the element according to these keys, unlike the traditional queue which works on first come first serve basis.

Thus, a **max-priority queue** returns the **element with maximum key first** whereas, a **min-priority queue** returns the **element with the smallest key first.**



Priority queues are used in many algorithms like Huffman Codes, Prim's algorithm, etc. It is also used in scheduling processes for a computer, etc.

There are two ways to implement the priority queue 1. List 2. Heap.

Heaps are great for implementing a priority queue because of the largest and smallest element at the root of the tree for a max-heap and a min-heap respectively. We use a max-heap for a max-priority queue and a min-heap for a min-priority queue.

There are mainly 4 operations we want from a priority queue:

1. **Insert** → To insert a new element in the queue.

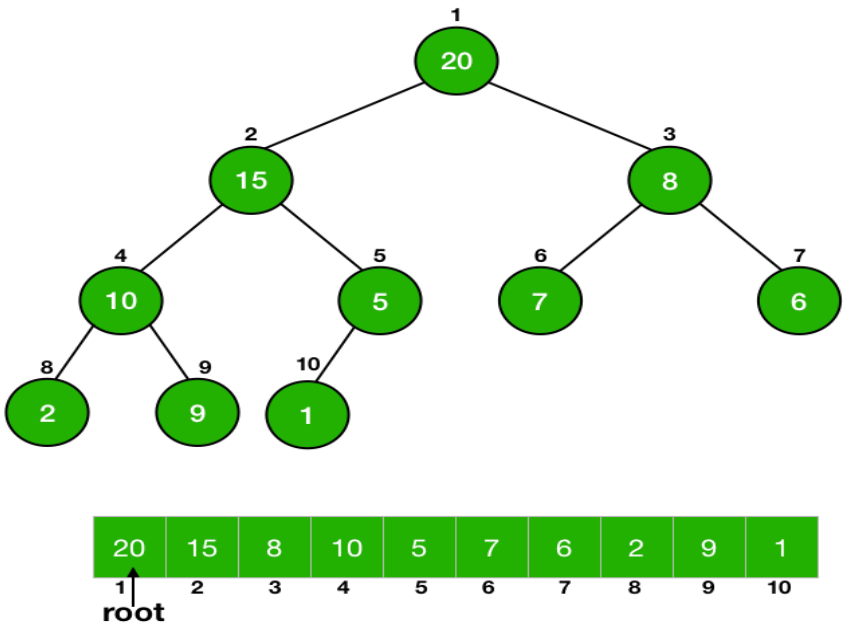
2. **Maximum/Minimum** → To get the maximum and the minimum element from the max-priority queue and min-priority queue respectively.

3. **Extract Maximum/Minimum** → To remove and return the maximum and the minimum element from the max-priority queue and min-priority queue respectively.

4. **Increase/Decrease key** → To increase or decrease key of any element in the queue.

Maximum/Minimum

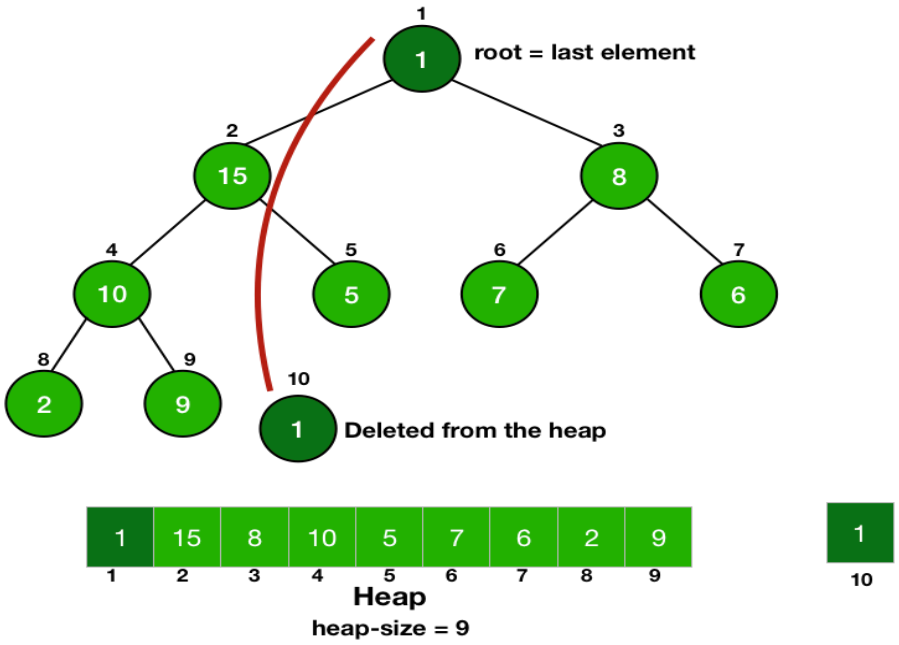
We know that the maximum (or minimum) element of a priority queue is at the root of the max-heap (or min-heap). So, we just need to return the element at the root of the heap.



Returning an element from an array is a constant time taking process, so it is a Θ(1) process.

Extract Maximum/Minimum

This is like the pop of a queue, we return the element as well as **delete** it from the heap. So, we have to return and delete the root of a heap. Firstly, we store the value of the root in a variable to return it later from the function and then we just make the root equal to the last element of the heap. Now the root is equal to the last element of the heap, we delete the last element easily by reducing the size of the heap by 1.



Doing this, we have disturbed the heap property of the root but we have not touched any of its children, so they are still heaps. So, we can call *Heapify* on the root to make the tree a heap again.

All the steps are constant time taking process except the *Heapify* operation, it will take O(logn) time and thus the Extract Maximum/Minimum is going to take O(logn) time.

2. Quick Sort

- Quick sort is also known as partition exchange sort
- Quick sort, sorts the element using divide and conquer strategy. Quick sort first divides the large array into two smaller subarrays: The lowest elements and high elements. Quick sort can then recursively sort the sub arrays.

The quick sort algorithm works as follows:

1. Select an element pivot from the array elements.
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the partition operation.
3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

Algorithm:

Quicksort Algorithm

Step 1: Make the left most index value pivot.

Step 2: Partition the array using pivot value.

Step 3: Apply quick sort on left paratitioned array.

Step 4: Apply quick sort on right partitioned array.

Quick Sort Partiton Algorithm

Step 1: Choose the lowest index value has pivot

Step 2: Take two variables to point to left and right of the list excluding pivot.

Step 3: Left points to the low index

Step 4: Right points to the high index

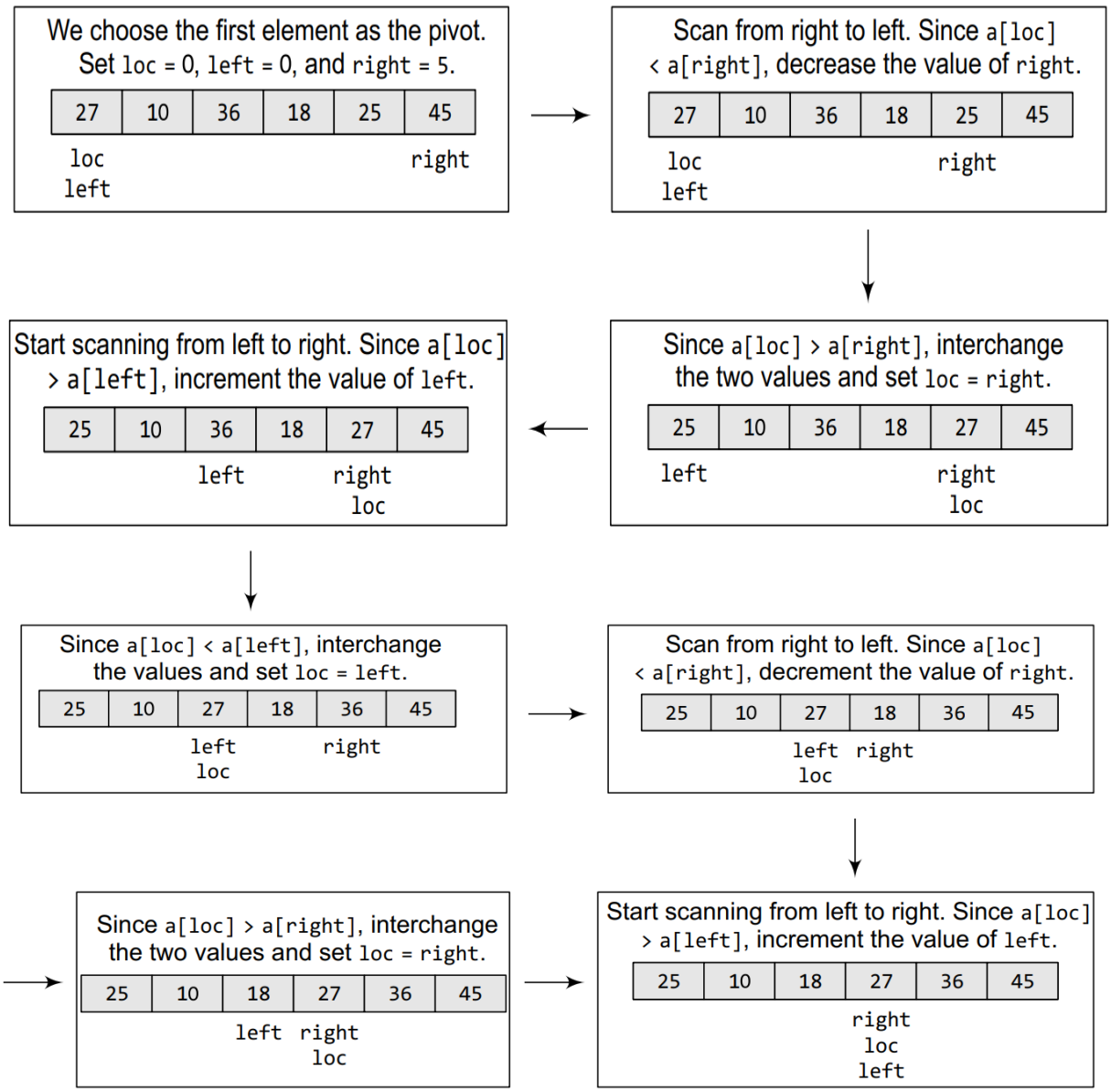
Step 5: While value at left is less than pivot, move right.

Step 6: While value at right is greater than pivot, move left.

Step 7: If left <= right then swap left and right

Step 8: If left >= right then swap point where they met is new pivot.

Example: Sort the elements given in the following array using quick sort algorithm 27, 10, 36, 18, 25, 45



C Program to implement the quick sort

```
#include <stdio.h>

#include <stdlib.h>

#include<math.h>

void quicksort(int[],int,int);


int main(int argc, char *argv[]) {
```

```
        int A[10],i,n;
        printf("\n Enter array size\n");
        scanf("%d",&n);
        printf("\n Enter array elements\n");
        for(i=0;i<n;i++)
        {
                scanf("%d",&A[i]);
        }
        quicksort(A,0,n-1);
        printf("\n After sorting array elements are\n");
        for(i=0;i<n;i++)
        {
                printf("%d\n",A[i]);
        }
        return 0;
}

void quicksort(int A[],int low,int high)
{
        int i,j,pivot,temp;

        if(low<=high)
        {
                i=low;
                j=high;
                pivot = low;

                while(i<=j)
                {
                        while(A[i]<=A[pivot] && i<=j)    i++;
```

```
                while(A[j] >A[pivot] && j >=i)  j--;
                if(i<j)
                {
                        temp=A[j];
                        A[j] = A[i];
                        A[i] = temp;
                }
        }

        temp = A[j];
        A[j] = A[pivot];
        A[pivot] = temp;

        quicksort(A,low,j-1);
        quicksort(A,j+1,high);
    }
}
```

Advantage and Drawback

- Advantage: It does not require any extra space for sorting array elements.
- Drawback: It takes more time for sorting array elements.

Time Complexities

1. Best and average case time complexity is O(nlogn)

2. Worst case time complexity is **$O(n^2)$**.

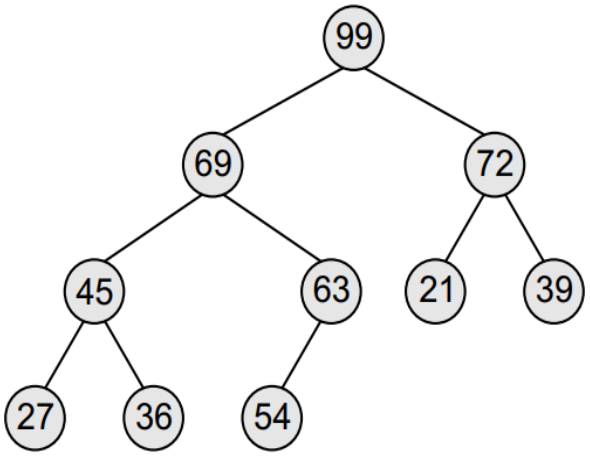
In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. At the most, only logn nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is O(log n). And because at each level, there can only be O(n), the resultant time is given as O(nlog n) time.

Practically, the efficiency of quick sort depends on the element which is chosen as the pivot. Its worst-case efficiency is given as $O(n^2)$. The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot.

3. Heap Sort

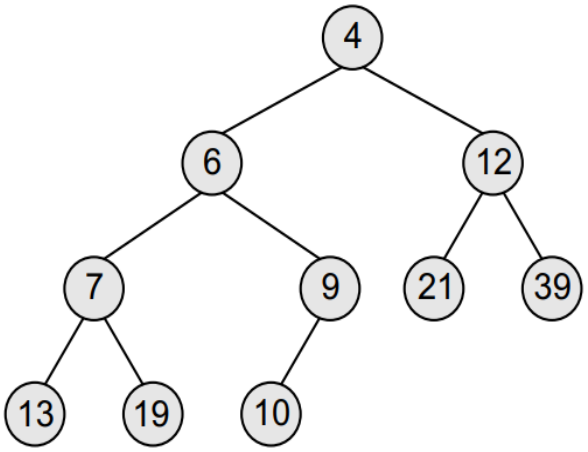
Heap is a complete binary tree.

Max heap: It is a complete binary tree in which every parent of the node greater than its descendants. Such a heap is called Max heap.



Max heap

Min Heap: It is a complete binary tree in which every parent of the node is less than its descendants. Such a heap is called Min heap.



Min heap

A heap is defined as a complete binary tree, all its elements can be stored sequentially in an array. It follows the same rules as that of a complete binary tree. That is, if an element is at position i in the array, then its left child is stored at position 2i and its right child at position 2i+1. Conversely, an element at position i has its parent stored at position i/2.

1. Being a complete binary tree, all the levels of the tree except the last level are completely filled.
2. The height of a binary tree is given as log2 n, where n is the number of elements.
3. Heaps (also known as partially ordered trees) are a very popular data

structure for implementing priority queues.

A binary heap is a useful data structure in which elements can be added randomly but only the element with the highest value is removed in case of max heap and lowest value in case of min heap.

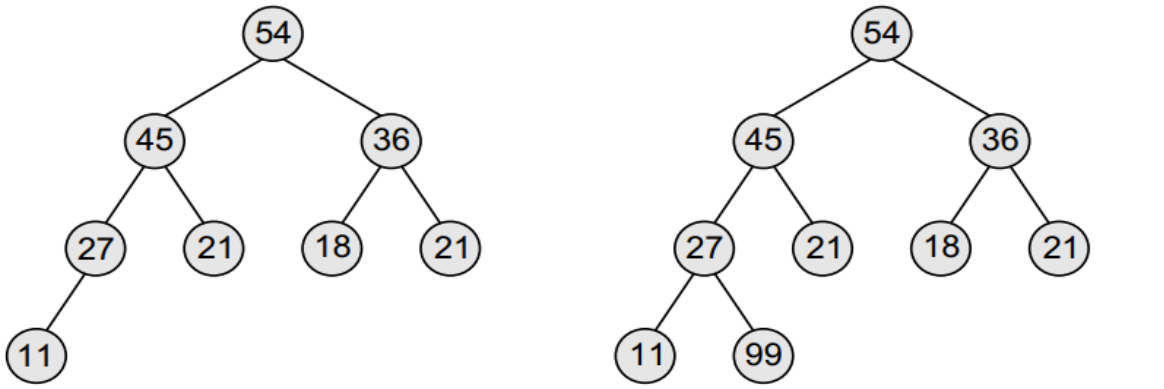
Inserting a New Element in a Binary Heap

Consider a max heap H with n elements. Inserting a new value into the heap is done in the following two steps:

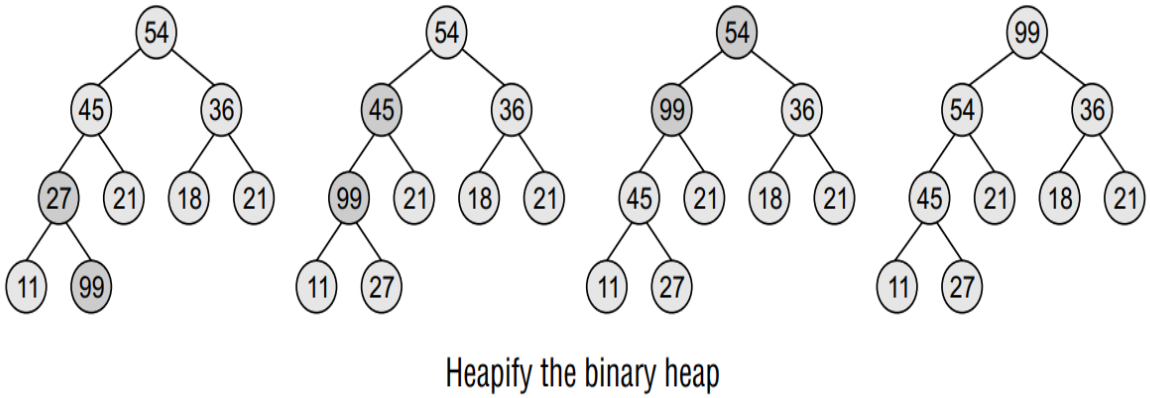
1. Add the new value at the bottom of H in such a way that H is still a complete binary tree but not necessarily a heap.
2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well.

To do this, compare the new value with its parent to check if they are in the correct order. If they are, then the procedure halts, else the new value and its parent's value are swapped and Step 2 is repeated.

Consider the max heap given in Figure and insert 99 in it

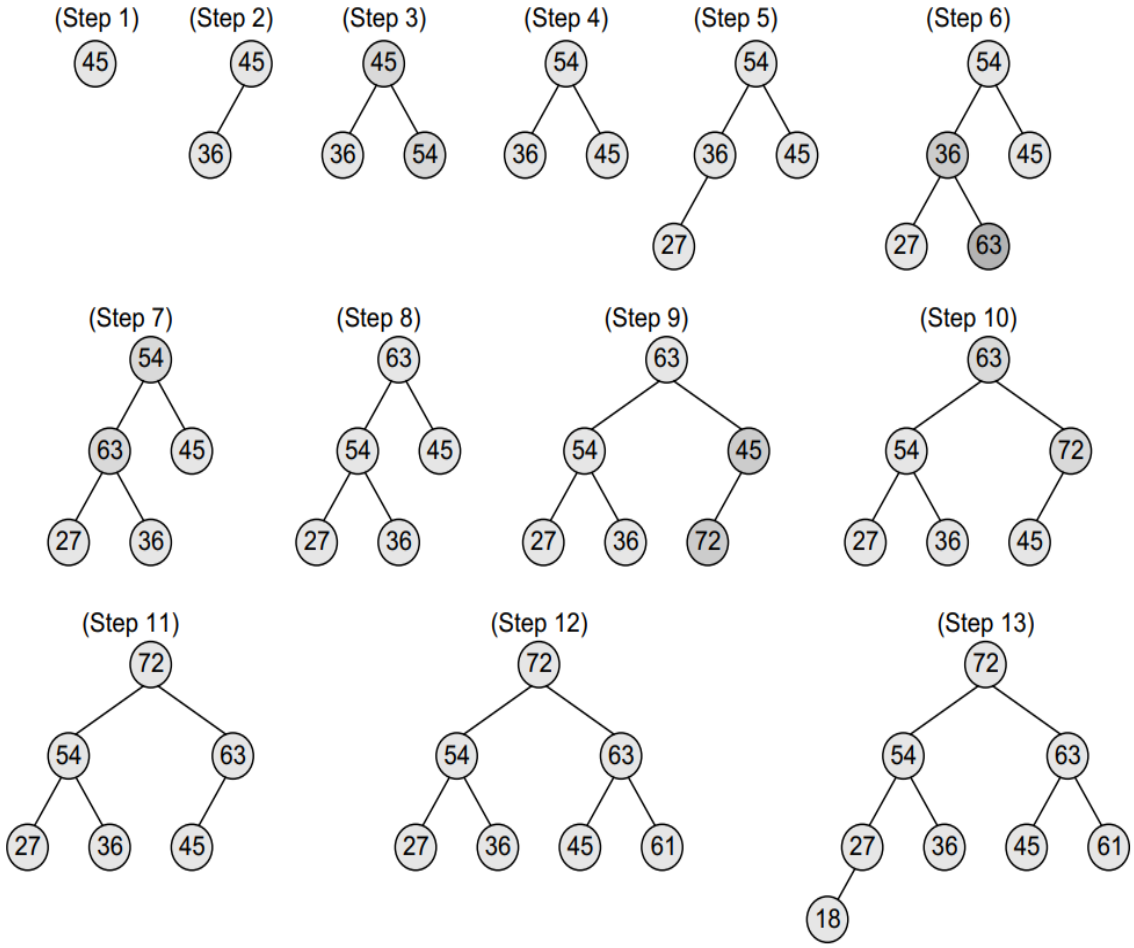


The first step says that insert the element in the heap so that the heap is a complete binary tree. So, insert the new value as the right child of node 27 in the heap.



Now, as per the second step, let the new value rise to its appropriate place in H so that H becomes a heap as well. Compare 99 with its parent node value. If it is less than its parent's value, then the new node is in its appropriate place and H is a heap. If the new value is greater than that of its parent's node, then swap the two values. Repeat the whole process until H becomes a heap. This is illustrated in Figure.

Exercise: Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18. Also draw the memory representation of the heap.



The memory representation of H can be given as shown in below

HEAP[1]	HEAP[2]	HEAP[3]	HEAP[4]	HEAP[5]	HEAP[6]	HEAP[7]	HEAP[8]	HEAP[9]	HEAP[10]
72	54	63	27	36	45	61	18		

Algorithm

Step 1: [Add the new value and set its POS]

SETN=N+1, POS=N

Step 2: SET HEAP[N] = VAL

Step 3: [Find appropriate location of VAL]

Repeat Steps 4 and 5 while POS>1

Step 4: SET PAR = POS/2

Step 5: IF HEAP[POS] <= HEAP[PAR],

then Goto Step 6.

ELSE

SWAP HEAP[POS], HEAP[PAR]

POS = PAR

[END OF IF]

[END OF LOOP]

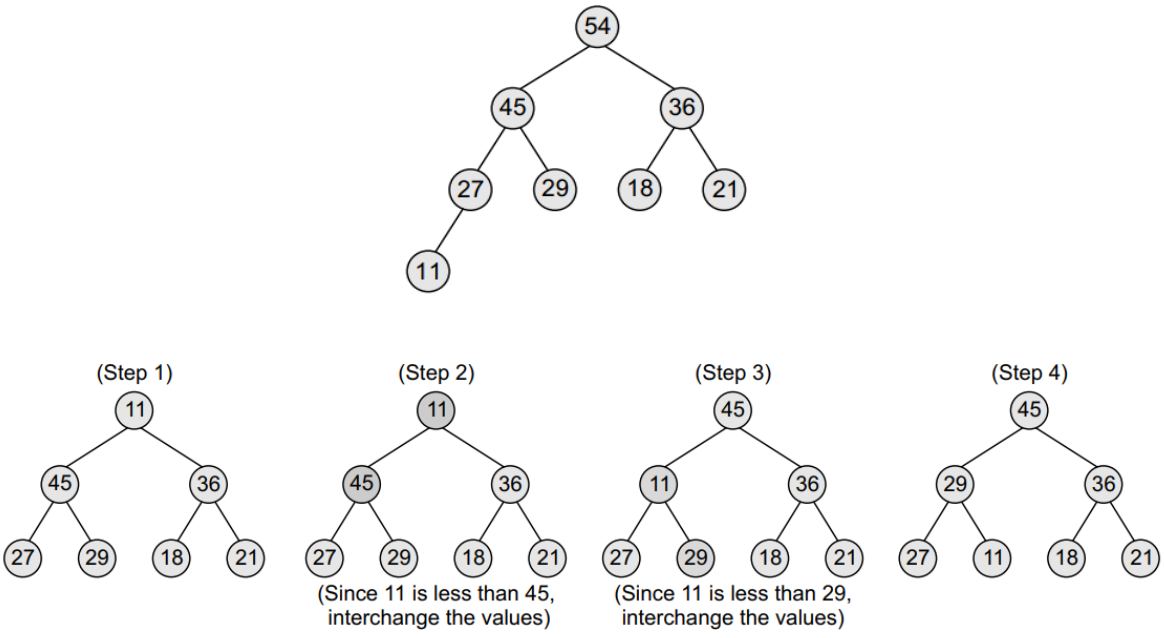
Step 6: RETURN

Deleting an Element from a Binary Heap

Consider a max heap H having n elements. An element is always deleted from the root of the heap. So, deleting an element from the heap is done in the following three steps:

1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.
2. Delete the last node.
3. Sink down the new root node's value so that H satisfies the heap property. In this step, interchange the root node's value with its child node's value (whichever is largest among its children).

Deleting a root node from the max heap as follows



Algorithm

Step 1: [Remove the last node from the heap]

SET LAST = HEAP[N], SETN=N-1

Step 2: [Initialization]

SET PTR=1, LEFT=2, RIGHT=3

Step 3: SET HEAP[PTR] = LAST

Step 4: Repeat Steps 5 to 7 while LEFT <= N

Step 5: IF HEAP[PTR] >= HEAP[LEFT] AND HEAP[PTR] >= HEAP[RIGHT]

Go to Step 8

[END OF IF]

Step 6: IF HEAP[RIGHT] <= HEAP[LEFT]

SWAP HEAP[PTR], HEAP[LEFT]

SET PTR = LEFT

ELSE

SWAP HEAP[PTR], HEAP[RIGHT]

SET PTR = RIGHT

[END OF IF]

Step 7: SET LEFT=2* PTR and RIGHT = LEFT+1

[END OF LOOP]

Step 8: RETURN

Applications of Binary Heaps

1. Sorting an array using heapsort algorithm.
2. Implementing priority queues

C Program to implement heap sort

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
```

```
void RestoreHeapUp(int *heap,int index)
{
	int val;
	val = heap[index];
	while((index>1)&&(heap[index/2]<val)) //check parents value
	{
		heap[index]=heap[index/2];
		index = index/2;
	}
	heap[index]=val;
}

void RestoreHeapDown(int *heap,int index, int n)
{
	int val,j;
	val = heap[index];
	j=index*2;
	while(j<=n)
	{
		if((j<n)&&(heap[j]<heap[j+1]))
			j++;
		if(heap[j]<val)
			break;
		heap[j/2]=heap[j];
		j=j*2;
	}
	heap[j/2]=val;
}
```

```
int main(int argc, char *argv[]) {
        int Heap[MAX],n,i,j,temp;
        printf("\n Enter array size");
        scanf("%d",&n);
        printf("\n Enter array elements:\n");
        for(i=1;i<=n;i++)
        {
                scanf("%d",&Heap[i]);
                RestoreHeapUp(Heap,i);    //Heapify
        }
        // Delete the root element and heapify the heap
        printf("\n After building heap array elements are\n");
        for(i=1;i<=n;i++)
        {
                printf("\n%d",Heap[i]);
        }
        j=n;
        for(i=1;i<=j;i++)
        {
                temp=Heap[1];
                Heap[1]=Heap[n];
                Heap[n]=temp;
                n=n-1;
                RestoreHeapDown(Heap,1,n); //Heapify
        }
        n=j;
        printf("\n After sorting array elements are\n");
        for(i=1;i<=n;i++)
        {
                printf("\n%d",Heap[i]);
```

```
    }
    return 0;
}
```

Time Complexities:

1. Best case time complexity is O(n).

The best case for heapsort would happen when all elements in the list to be sorted are identical. In such a case, for 'n' number of nodes-

- Removing each node from the heap would take only a constant runtime, O(1). There would be no need to bring any node down or bring max valued node up, as all items are identical.
- Since we do this for every node, the total number of moves would be n * O(1).

Therefore, the runtime in the best case would be O(n)

2. Worst case and average time complexity is O(nlogn)

5. Radix Sort

- Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order.
- When we have a list of sorted names, the radix is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort.
- It works by sorting the elements based on each digit or letter from the least significant digit to the most significant digit. Radix sort is often used for sorting strings, integers, or other data types where the order of individual digits or letters is significant.
- Words are first sorted according to the first letter of the name, During the second pass, names are grouped according to the second letter, After the second pass, names are sorted on the first two letters. This process is continued till the n^{th} pass, where n is the length of the name with maximum number of letters.
- When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, …, 9) and the number of passes will depend on the length of the number having maximum number of digts.

Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE

Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE

Step 3: SET PASS = 0

Step 4: Repeat Step 5 while PASS <= NOP-1

Step 5: SET I = 0 and INITIALIZE buckets

Step 6: Repeat Steps 7 to 9 while I < N-1

Step 7: SET DIGIT = digit at PASSth place in A[I]

Step 8: Add A[I] to the bucket numbered DIGIT

Step 9: INCEREMENT bucket count for bucket numbered DIGIT

[END OF LOOP]

Step 1 : Collect the numbers in the bucket

[END OF LOOP]

Step 11: END

Example: Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

Sol: In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The numbers are 911, 472, 123, 654, 924, 345, 555, 567, 808. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

The collected numbers after pass-2 are 808, 911, 123, 924, 345, 654, 555, 567, 472. In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

C Program to implement Radix sort

```
#include <stdio.h>

#include <stdlib.h>

#define size 10
```

```
int largest(int A[],int n)
{
	int large,i;
	large=A[0];
	for(i=1;i<=n;i++)
	{
		if(A[i]>large)
		{
			large=A[i];
		}
	}
	return large;
}

void Radixsort(int A[],int n)
{
	int bucket[size][size],bucket_count[size];
	int i,j,k,rem,nop=0,div=1,large,pass;
	large=largest(A,n);
	while(large>0)
	{
		nop++;
		large=large/size;
	}
	for(pass=0;pass<nop;pass++)
	{
		for(i=0;i<size;i++)
			bucket_count[i]=0;
		for(i=0;i<=n;i++)
		{
```

```
                rem = (A[i]/div)%size;
                bucket[rem][bucket_count[rem]]=A[i];
                bucket_count[rem] +=1;
            }
            //collect the number after PASS pass
            i=0;
            for(k=0;k<size;k++)
            {
                for(j=0;j<bucket_count[k];j++)
                {
                    A[i]=bucket[k][j];
                    i++;
                }
            }
            div = div*size;
        }
}
int main(int argc, char *argv[]) {
        int i,n,A[10];
        printf("\n Enter array size");
        scanf("%d",&n);
        printf("\n Enter array elements:\n");
        for(i=0;i<n;i++)
        {
            scanf("%d",&A[i]);
        }
        Radixsort(A,n-1);
        printf("\n After sorting array elements are\n");
        for(i=0;i<n;i++)
        {
```

```
        printf("\n%d",A[i]);
    }
    return 0;
}
```

Complexity of Radix sort

- The time complexity of radix sort is given by the formula,**T(n) = O(d*(n+b))**, where d is the number of digits, n is the number of elements in the list, and b is the base or bucket size used, which is normally base 10 for decimal representation.
- In practical implementations, radix sort is often faster than other comparison-based sorting algorithms, such as quicksort or merge sort, for large datasets, especially when the keys have many digits. However, its time complexity grows linearly with the number of digits, and so it is not as efficient for small datasets.

5. Merge Sort

- Merge sort uses divide and conquer strategy.
- It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.
- **Divide** means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A and A2, each containing about half of the elements of A.
- **Conquer** means sorting the two sub-arrays recursively using merge sort.
- **Combine** means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

Merge sort Algorithm

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

   middle m = (l+r)/2

2. Call mergeSort for first half:

   Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

Merge Algorithm

MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SETI= BEG,J= MID+1, INDEX = 0

Step 2: Repeat while (I <= MID) AND (J<=END)

IF ARR[I] < ARR[J]

SET TEMP[INDEX] = ARR[I]

SETI=I+1

ELSE

SET TEMP[INDEX] = ARR[J]

SETJ=J+1

[END OF IF]

SET INDEX = INDEX+1

[END OF LOOP]

Step 3: [Copy the remaining elements of right sub-array, if any]

IF I > MID

Repeat while J <= END

SET TEMP[INDEX] = ARR[J]

SET INDEX = INDEX+1, SETJ=J+1

[END OF LOOP]

[Copy the remaining elements of left sub-array, if any]

ELSE

Repeat whileI<= MID

SET TEMP[INDEX] = ARR[I]

SET INDEX = INDEX+1, SETI=I+1

[END OF LOOP]

[END OF IF]

Step 4: [Copy the contents of TEMP back to ARR] SET K= 0

Step 5: Repeat whileK< INDEX

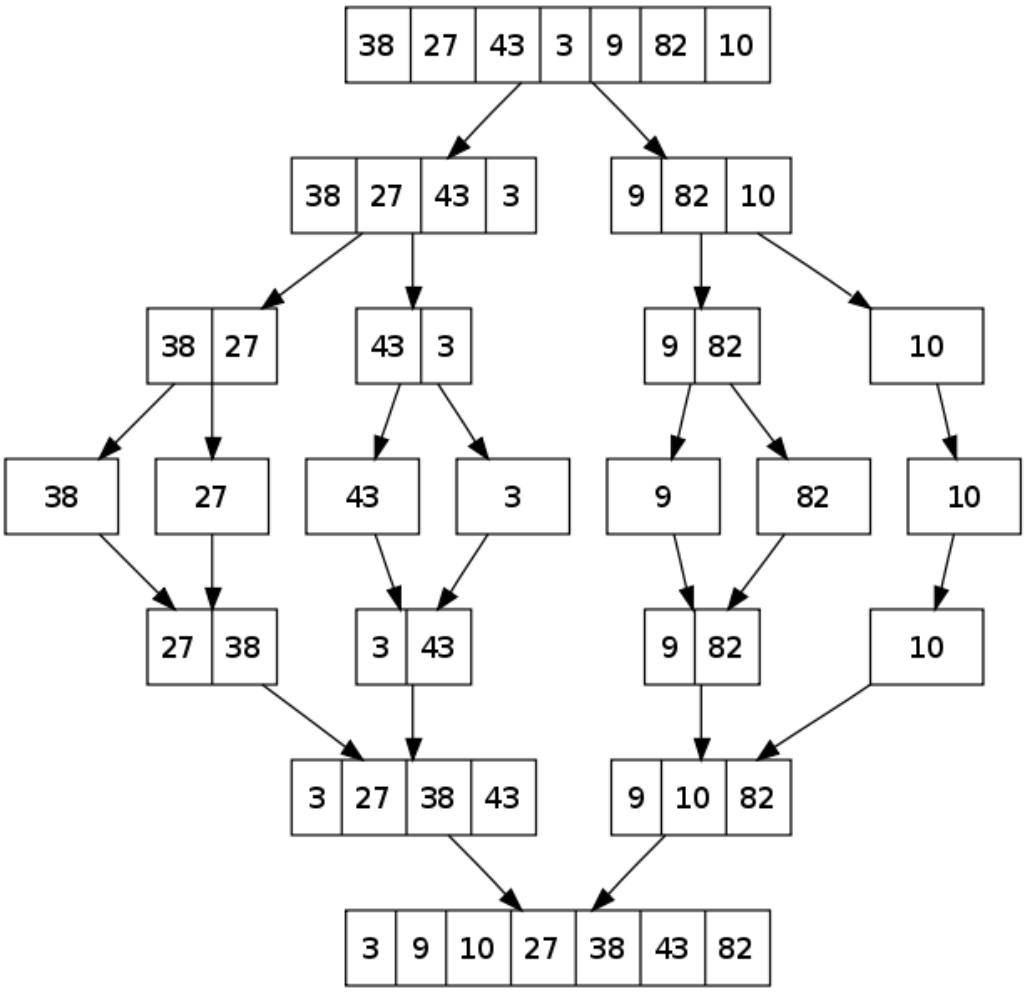
SET ARR[K] = TEMP[K]

SETK=K+1

[END OF LOOP]

Step 6: END

Example:



Advantage and Drawback:

Advantage: It takes less for sorting the elements.

Drawback: It require extra space (extra memory) for sorting the elements.

Time Complexities

1. Best, average and worst case time complexity is O(nlogn)

Summary of all sorting techniques

Algorithm	Time Complexity		
	Best case	**Average case**	**Worst case**
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick sort	O(nlogn)	O(nlogn)	$O(n^2)$
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)
Radix sort	O(n+k)	O(n+k)	O(n.k)

UNIT-V

Syllabus:

Dictionaries: Definition, ADT, Linear List representation, operations- insertion, deletion and searching, Hash Table representation, Hash function-Division Method, Collision Resolution Techniques-Separate Chaining, open addressing-linear probing, quadratic probing, double hashing, Rehashing.

Graphs: Graph terminology –Representation of graphs –Graph Traversal: BFS (breadth first search) –DFS (depth first search) –Minimum Spanning Tree.

Dictionary:

A dictionary is an ordered or unordered list of key-value pairs, where keys are used to locate values in the list.

A dictionary is a container of elements, each element is a pair of key and value. Where each value is associated with a corresponding key.

Example:

1. Bank account: It can be viewed as a dictionary where account number serves as a key for identification of account objects.
2. Students marks: In a class roll number serves as a key and marks serves as values

Visual representation of a dictionary

Key	**values**
460	90
470	95
480	70
490	50
4A0	75
4B0	80
4C0	60

Basic operations of Dictionary or Dictionary ADT

A dictionary D is a dynamic set Abstract data type (ADT), it has the following operations

1. Insert(key k, value v) - inserting (key and value) into the dictionary D
2. Delete(key k) – deleting (key and value) from the dictionary D with the help of key corresponding to k
3. Search(key k) – search a value x in the dictionary D with a key of an element x
4. Member(x, D) – it return true it x is in D otherwise return false
5. Size(D) – it returns the total number of elements in D.
6. Max(D) – it returns the maximum value in the dictionary D.
7. Min(D) – it returns minimum value in the dictionary D

Examples:

Empty Dictionary D - { }

Insert((1,A),D) - **{(1,A)}**

Key	**value**
1	**A**

Insert((3,B),D) - **{(1,A),(3,B)}**

Key	**value**
1	**A**
3	**B**

Insert ((10, C),D) - **{(1,A),(3,B), (10,C)}**

Key	**value**
1	**A**
3	**B**
10	**C**

Delete(3, D) - **{(1,A), (10,C)}**

Key	value
1	A
10	C

Search(10,D) - **returns true**

Max(D) - **returns key 10**

Min(D) - **returns 1**

Size(D) - **returns 2 (total number of elements in the dictionary)**

Implementation of dictionary

dictionary is implemented using

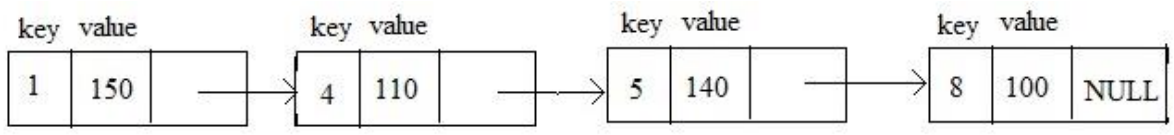
1. Fixed length array
2. Linked list (linear list)
3. Hashing
4. Trees

Linear list representation

The dictionary can be represented as a linear list, it is a collection of pairs (key and value).

There are two methods in representation of a dictionary using linked list. They are

1. Sorted array
2. Sorted chain



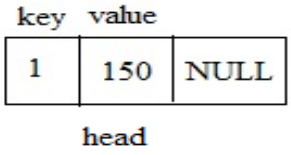
The contents of dictionary are always is in sorted form.

Linked list node

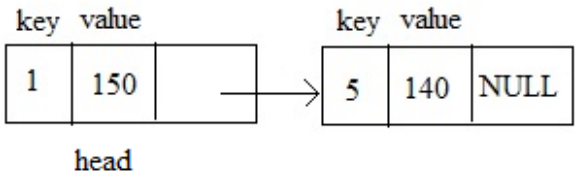
```
Struct node
{
int key;
int value;
struct node *next;
};
```

Operations

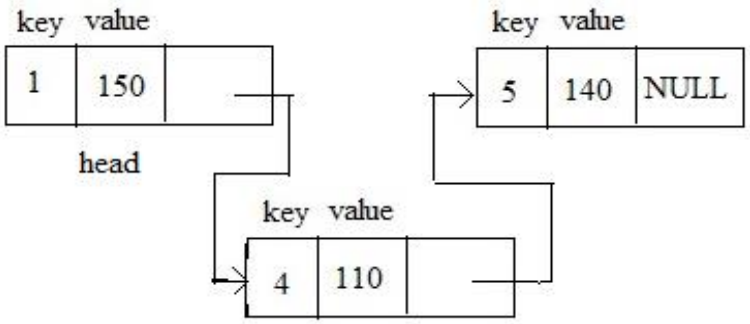
Insertion: initially dictionary is empty, it means head is NULL. Then we will create a new node with some key and value. Now newly created node becomes a head node.



Now we want to insert a key and value pair(key=5, value=140), then it will create a new node with key 5, value 140 and its next field NULL. This newly created node will be inserted at the end of the list. Because its key value 5 is greater than the list node key value 1.



Now we want to insert a key and value pair(key =4, value = 110), then it will create a new node with key 4, value 140 and next field NULL. This new node will be inserted between first and second nodes, because its key 4 between the first and second node keys 1 and 5 respectively.

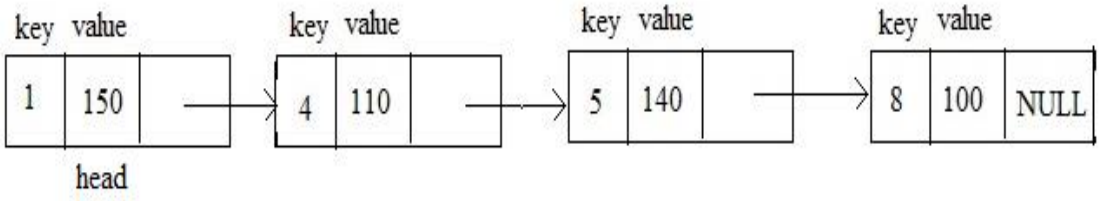


Note:

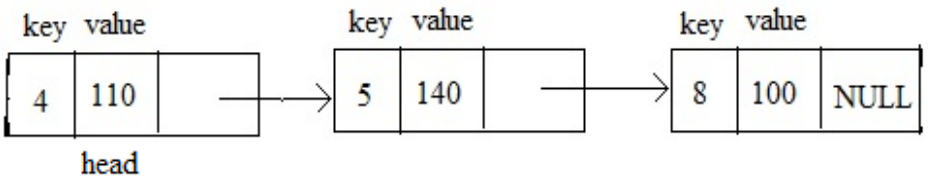
1. After insertion operation, all nodes key values must be in ascending order.

Deletion

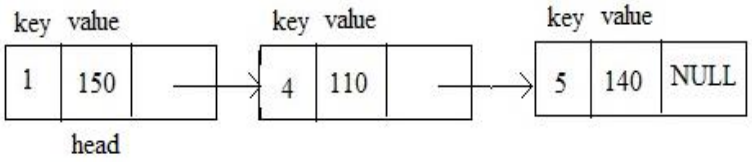
We have three cases in deletion operation, they are deleting a node at the beginning, ending and in between nodes.



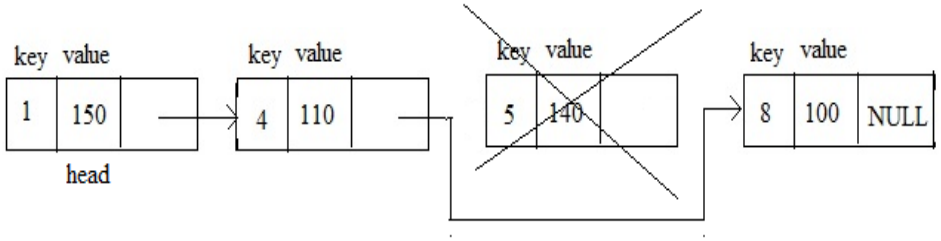
Case 1 - deleting a head node: In this case after deleting the first node, head is pointing to the second node.



Case 2 – deleting a last node: after removing the last node, last but one node becomes the last node and its next field will be set to NULL.



Case 3- deleting in between nodes: we want to delete a node with key 5.



Search

Using search operation, we will find the key value is available in the list or not. It uses linear search, so key value is compared with the first node's key value, if match occurs it returns true, otherwise compare with the second node's key value, this process is repeated until key is found in the list or we

reached to the last node. If last node's key value is not equal to the key value then we return false, it means key is not found in the list.

Why hashing

1. The sequential search algorithm takes time proportional to the data size i.e. **O(n)**
2. Binary search improves on linear search reducing the search time to **O(logn)**
3. With a BST an O(logn) search efficiency can be obtained, but the worst case complexity is **O(n).**
4. To guarantee the O(logn) search time, BST height balancing is required (i.e. AVL trees).
5. Suppose we want to store 10,000 students records (each with a 5-digit ID) in a given container
    1. A linked list implementation would take **O(n) time**.
    2. A height balanced tree would give **O(logn)** access time.
    3. Using an array of size 1,00,000 would give **O(1)** access time but will **lead to a lot of space wastage**.

Note: using hashing we can access records in **O(1)**. So we use hashing

Hashing is a common method of accessing data records using the hash table. Hashing can be used to build, search or delete from a table.

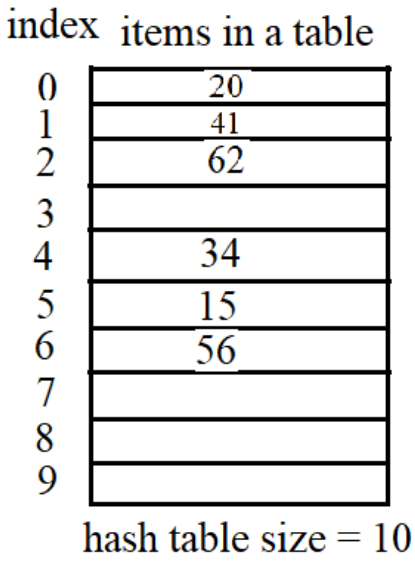
Hashing is the process of mapping large amount of data item to a smaller table with the help of hashing function.

1. Hash table is an extremely effective and practical way of implementing dictionaries.
2. It takes O(1) time for search, insert and delete operations in the average case and O(n) in the worst case.

Hash Table: Hash table is a data structure in which keys are mapped to array positions by a hash function.

Load factor: the ratio of the number of items in a table to the table's size is called the load factor.

$$\text{load factor} = \frac{\text{no of items in a table}}{\text{hash table size}} = \frac{6}{10} = 0.6$$



A **hash table** is a data structure that stores records in an array called a hash table. Hash table can be used for quick insertion and searching

A **hash table** is a data structure for storing key/value pairs. This table can be search for an item in O(1) time using a hash function to form an address from the key.

Hash function: Hash function maps key values to array indices.

(or)

Hash Function is a function which, when applied to the key, produces an integer which can be used as an address in a hash table.

Example: Given key values: 56, 41, 20, 34, 15, 62

We will use h(k) for representing the hashing function,

h(key) = key % table_size

h(56) = 56 %10 = 6

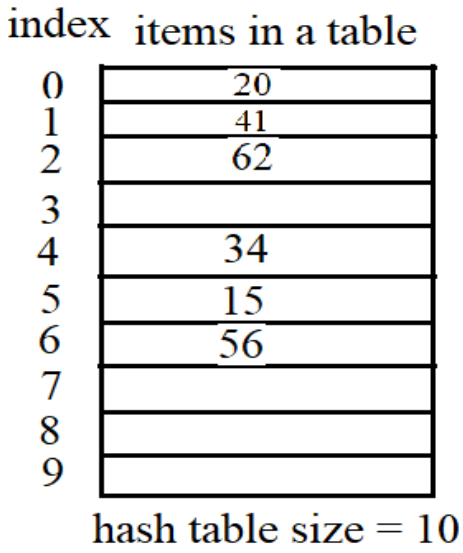
h(41) = 41 % 10 = 1

h(20) = 20 % 10 = 0

h(34) = 34 % 10 = 4

h(15) = 15 % 10 = 5

h(62) = 62 % 10 = 2



Hash Values: The values returned by a hash function are called hash values or hash codes or hash sums or simply hashes.

Insertion of data in the hash table is based on the key value. Every entry in the hash table is associated with some key.

Types of hash functions:

hash function types are

1. Division method
2. Mid square method
3. Folding method
4. Radix hashing

Division method: Mapping a key K into one of m slots by taking the remainder of K divided by m.

h(k) = k mod m

Example: Assume a table has 10 slots (m=10). Using division method, insert the following elements into hash table. 56, 41, 20, 34, 15 and 62 are inserted in the order.

h(56) = 56 %10 = 6

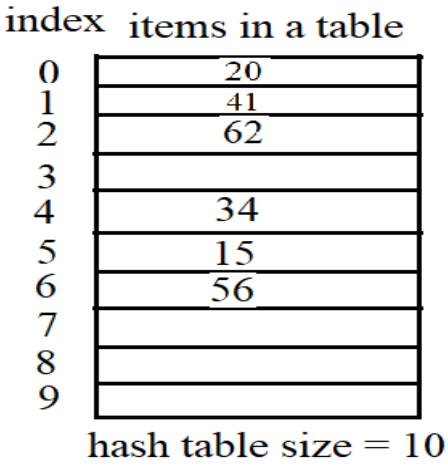
h(41) = 41 % 10 = 1

h(20) = 20 % 10 = 0

h(34) = 34 % 10 = 4

h(15) = 15 % 10 = 5

h(62) = 62 % 10 = 2



Mid-Square Method: mapping a key K into one of m slots, by getting the some middle digits from values K^2 .

$H(K^2) = K^2$ and get middle (logm base 10) digits.

Example: 15 is a key and square of 15 is 225. Middle part is 2 (with a table size of 10).

Folding Method: divide the K into some sections, besides the last section, have same length. Then, add these sections together.

Example: Shift folding (123456 is folded as 12 +34+56)

Folding at the boundaries (123456 is folded as 12 + 43 + 56)

Radix Sorting: transform the number into another base and then divide by the maximum address.

Example: Addresses from 0 to 99, key = 453 in base 11 = 382. Hash address = 382 mod 99 = 85.

Problems with hashing:

Collision: No matter what the hash function, there is the possibility that two different keys could resolve to the same hash address. This situation is called collision.

Handing collisions: the following techniques can be used to handle the collisions.

1. Separate chaining
2. Open address (Linear probing, Quadratic probing)
3. Double hashing (Re-hashing).

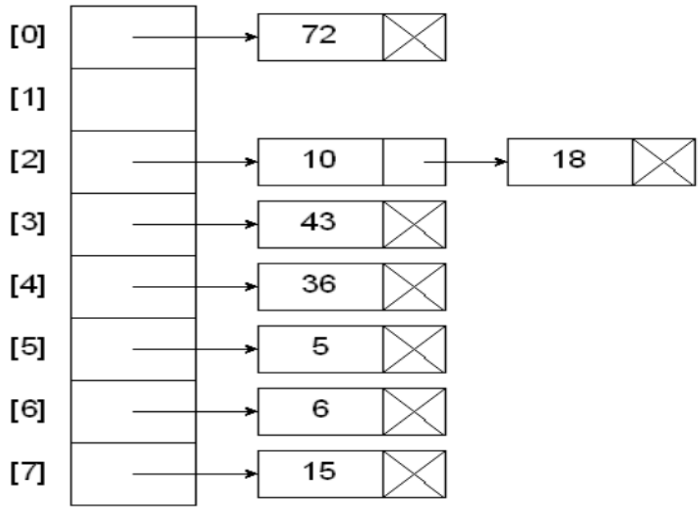
Separate chaining:

- A chain is simply a linked list of all the elements with the same hash key.
- A linked list is created at each index in the hash table.
- Hash function: h(k) = k mod n

Example: Assume a table has 8 slots (m = 8) using the chaining, insert the following elements into the hash table, 36, 18, 72, 43, 6, 10, 5 and 15 are inserted in the order.

Hash key = key % table size

36 % 8 = 4
18 % 8 = 2
72 % 8 = 0
43 % 8 = 3
6 % 8 = 6
10 % 8 = 2
5 % 8 = 5
15 % 8 = 7



- A data item's key is hashed to the index in simple hashing, and the item is inserted into the linked list at that index.
- Other items that hash to the same index are simply added to the linked list.
- Average length = N/M where N is size of the array and M is the number of linked lists.

Analysis of chaining:

1. If M is too large then too many empty chains.
2. If M is too small, then chains are too long.

Advantages of chaining: unbounded elements can be stored (No bound to the size of table).

Disadvantages of chaining: too many linked lists (overhead of linked lists).

Open addressing:

In open addressing, when a data item can't be placed at the hashed indexed value, another location in the array is sought. We will explore three methods of open addressing, which vary in the method used to find the next empty location. These methods are linear probing, quadratic probing and double hashing.

Linear probing:

When using a linear probing method, the item will be stored in the next available slot in the table, assuming that the table is not already full.

This is implemented via linear searching for an empty slot, from the point of collision.

If the end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there.

Example: Assume a table has 8 slots (m = 8) using the chaining, insert the following elements into the hash table, 36, 18, 72, 43, 6, 10, 5 and 15 are inserted in the order.

Hash key = key % table size

36 % 8 = 4
18 % 8 = 2
72 % 8 = 0
43 % 8 = 3
6 % 8 = 6
10 % 8 = 2
5 % 8 = 5
15 % 8 = 7

[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

Insert 36, 18, 72, 43, 6, 10, 5 and 15

insert 36 h(36) = 36%8 = 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
				36			

insert 18 h(18) = 18 % 8 = 2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
		18		36			

insert 72 h(72) = 72%8 =0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
72		18		36			

insert 43 h(43) = 43%8 =3

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
72		18	43	36			

insert 6 h(6) = 6%8 =6

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
72		18	43	36		6	

insert 10 h(10) = 10%8 =2
here index 2 is already filled so we have to find the next empty position from index 2, i.e index 5

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
72		18	43	36	10	6	

insert 5 h(5) = 5 % 8 = 5
index 5 is alreay filled so next empty position from index 5 is 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
72		18	43	36	10	6	5

insert 15 h(15) = 15 % 8 = 7
index 7 is already filled so next empty slot from index 7 is 1

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
72	15	18	43	36	10	6	5

Relationship between probe length (P) and load factor(L) for linear probing:

1. For successful search: $P = (1 + 1 / (1 – L)^2) / 2$
2. For unsuccessful search $P = (1 + 1 / (1 – L) / 2$

Analysis of Linear Probing: if load factor is too small then too many empty cells.

Quadratic Probing:

If the collision occurs instead of moving one cell, i^2 cells from the point of collision, where I is the number of attempt to resolve the collision.

Example: assume a hash table has 10 slots. Using quadratic probing, insert the following elements in the given order.

89, 18, 49, 58 and 69 are inserted into a hash table.

89 % 10 = 9

18 % 10 = 8

49 % 10 = 9

49 = 9 is occupied, so 1 attempt needed. Insert at $9 + 1^2 = 10 = 0$ location.

58 % 10 = 8 3 attempts – 3^2 = 9 spots

58 = 8 is occupied, and next 3 locations are occupied.

So $8 + 3^2 = 17 = 7$ location.

69 % 10 = 9 2 attempts – 2^2 = 4 spots

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Advantages of quadratic probing: compared to linear probing access becomes inefficient at a higher load factor.

Disadvantages of Quadratic Probing: insertion sometimes fails although the table still has free fields.

Double Hashing:

- Double hashing requires that the size of the hash table is a prime number.
- Double hashing uses the idea of applying a second hash function to the key when a collision occurs.
- The primary hash function determines the home address. If the home address is occupied, apply a second hash function to get a number c. (c relatively prime to N). this c is added to the home address to produce an overview addresses: If occupied, proceed by adding c to the overflow address, until an empty slot is found.
- Primary hash function is similar to direct hashing.

Hash1(key) = key % table_size

A popular secondary hash function is

Hash2(key) = R – (key % R)

where R is a prime number that is smaller than the size of the table.

- Example: assume a table has 10 slots. Primarly hash function is

H1(key) = key mod 10 and second hash function is H2(key) = 7 – (key mod 7)

With double hashing, insert the following elements in the given order.

89, 18, 49, 58 and 69 are inserted into a hash table

$H_1(89) = 89 \% 10 = 9$ (primary hashing)

$H_1(18) = 18 \% 10 = 8$

$H_1(49) = 49 \% 10 = 9$ a collision (primary hashing)

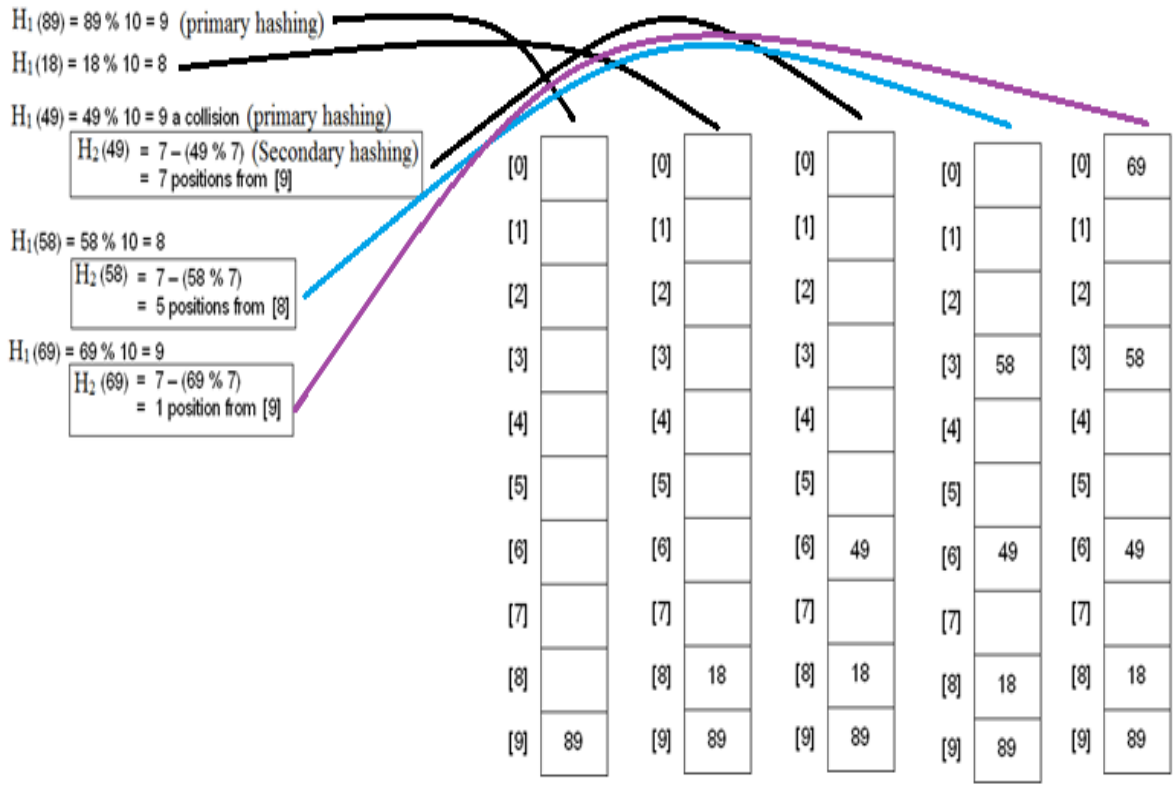
$H_2(49) = 7 - (49 \% 7)$ (Secondary hashing)
$= 7$ positions from [9]

$H_1(58) = 58 \% 10 = 8$

$H_2(58) = 7 - (58 \% 7)$
$= 5$ positions from [8]

$H_1(69) = 69 \% 10 = 9$

$H_2(69) = 7 - (69 \% 7)$
$= 1$ position from [9]



Advantages of double hashing:

1. Compared to linear probing access becomes inefficient at a higher load factor
2. Resolution sequences for different elements are different even if the first hash function hashes the elements to the same field.
3. If the hash functions are chosen appropriately, insertion never fails if the table has at least one free field.

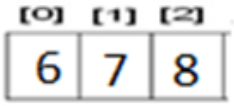
Rehashing:

As the name suggests, rehashing means hashing again. Basically, when the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity.

Example: Store key values 6, 7, 8 into the hash table size 3.

H(k) = k mod table_size

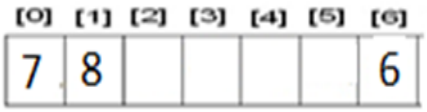
- H(6) = 0, h(7) = 1, h(8) = 2



- Now load factor = 1, suppose we want to insert some more keys into the hash table then we should increase hash table size.
- New hash table size must be the nearer value of 2*previous table size.
- New table size = 3*2 = 6 prime number nearer 6 is 7. so we should take the new table size is 7.
- Now new hash function

H(key) = key mod 7

- Now all the existing values are stored in the new hash table using the updated hash function i.e



Graphs