New L J Institute of Engineering and Technology, Bodakdev Branch: CSE (AIML)/IT
Semester: III
Subject: Data Structures (3130703)

1	Write a short note on performance analysis and performance measurement of an algorithm.

Time and Space Complexity

- Analyzing an algorithm means determining the time and memory needed to execute it.
- Algorithms take a random number of inputs, so the efficiency of an algorithm is measured in terms of time and space complexity.
- The time and space complexity can be expressed using a function f(n) where n is the input size for a given instance of the problem being solved.

The time complexity of an algorithm is the amount of time that is required by a program as a function of the input size.
The space complexity of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.

Expressing Time and Space Complexity (Asymptotic Analysis)

- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.
- Using asymptotic analysis, we can conclude the best case, average case, and worst case scenario of an algorithm.

Usually, the time required by an algorithm falls under three **CASES**

- Best Case − Minimum time required for program execution.
- Average Case − Average time required for program execution.
- Worst Case − Maximum time required for program execution.

Asymptotic Notations

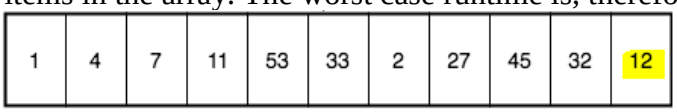
Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm in a simple way by ignoring the constant and lower order terms from the time complexity.

- Big Oh Notation, O
- Big Omega Notation, Ω
- Big Theta Notation, θ

Worst-case

- Worst-case running time denotes the behavior of an algorithm with respect to the worst possible case of the input instance.

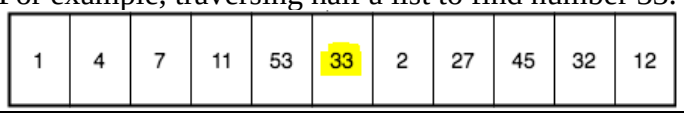
In the linear search, the worst case happens when the item we are searching is in the last position of the array (i.e 12 here) or the item is not in the array. In both the cases, we need to go through all n items in the array. The worst case runtime is, therefore, O(n)



Average-case

- The average-case running time of an algorithm is an estimate of the running time for an 'average' input.

For example, traversing half a list to find number 33.



Best-case

- Best-case running time The term 'best-case performance' is used to analyse an algorithm under optimal conditions.

if we want to search 1 in the array, it is present at the first position of the array, it will return the index immediately. The for loop runs only once.

1	4	7	11	53	33	2	27	45	32	12

Asymptotic Notations

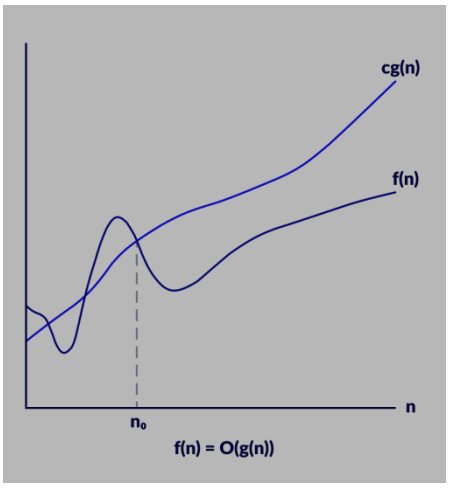
Big Oh Notation (O)

- Big O notation cares about the worst-case scenario. E.g., when you want to sort elements in the array that are in reverse order.

- If f(n) and g(n) are the functions defined on a positive integer number n, then f(n) = O(g(n)) Big O notation is simply written as $f(n) \in O(g(n))$ or as f(n) = O(g(n)).

f(n) = O(g(n)), If there exists positive constants c and n0 such that $f(n) \leq cg(n)$ for all $n \geq n0$ then g(n) is an asymptotically tight upper bound for f(n).

- It means that for large amounts of data, f(n) will **not** grow more than a constant factor than g(n). Hence, g(n) provides the tight upper bound.

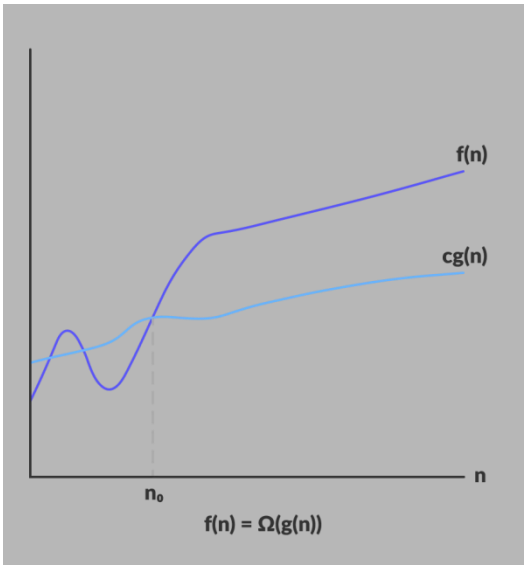


- Drops constants and lower order terms.
  E.g. $O(3n^2 + 10n + 10)$ becomes $O(n^2)$.
- Table shows the relationship between g(n) and f(n).

g(n)	f(n) = O(g(n))
10	O(1)
$2n^3 + 1$	$O(n^3)$
$3n^2 + 5$	$O(n^2)$
$2n^3 + 3n^2 + 5n - 10$	$O(n^3)$

Big Omega Notation (Ω)

- The Omega notation(**Ω**) provides a tight lower bound for f(n).
- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.
- For example, when sorting an array the best case is when the array is already correctly sorted.
- This means that the function can never do better than the specified value.



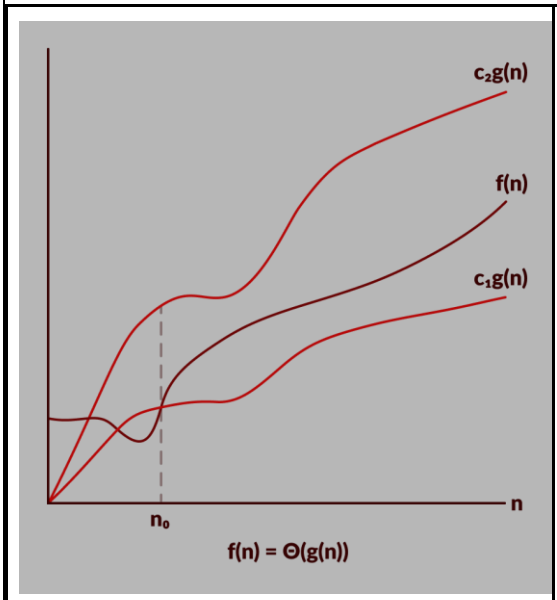
Ω notation is simply written as, f(n) ∈ Ω(g(n)), where n is the problem size and

f(n) =Ω(g(n)), If there exists positive constants c and n0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n0$ and g(n) is an asymptotically tight lower bound for f(n).

Examples of functions in $\Omega(n^2)$ include:
n^2, $n^{2.9}$, $n^3 + n^2$, n^3
Examples of functions not in $\Omega(n^3)$ include:
n, $n^{2.9}$, n^2

Big Theta Notation(θ)

- Theta notation provides an asymptotically tight bound for f(n).
- It defines exact asymptotic behavior.
- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.



- Θ notation is simply written as, f(n) ∈ Θ(g(n)), where n is the problem size and

f(n) = Θ(g(n)), if there exist positive constants c1,c2 and n0 such that $c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n0$

2 Define data structure. Briefly explain linear and non-linear data structures with their applications.

Linear and Non-linear Structures

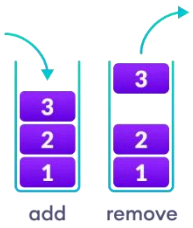
Linear data structures

- If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure.
- Examples: arrays, linked lists, stacks, and queues.
- Linear data structures can be represented in 2 ways:
  - linear relationship between elements by means of sequential memory locations.
  - linear relationship between elements by means of links.

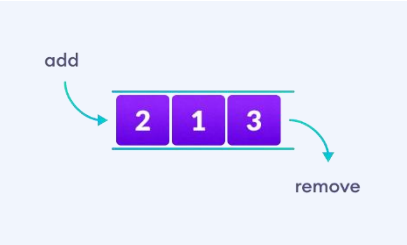
Arrays

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.

Stack Data Structure
In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.
It works just like a pile of plates where the last plate kept on the pile will be removed first.



In a stack, operations can be performed only from one end (top here).
Queue Data Structure
Unlike stack, the queue data structure works in the FIFO principle where the first element stored in the queue will be removed first.
It works just like a queue of people in the ticket counter where the first person on the queue will get the ticket first.

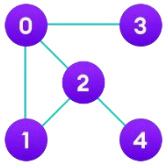


In a queue, addition and removal are performed from separate ends.

Linked List Data Structure
In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node.

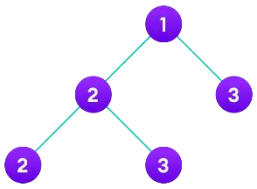
<u>Non-Linear data structures</u>
- if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure.
- The relationship of adjacency is not maintained between elements of a non-linear data structure.
- Examples: trees and graphs.

Graph Data Structure
In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.



Trees Data Structure

Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.



3 What is a sparse matrix? Explain memory representation of a sparse matrix.

Sparse matrix is a matrix that has a large number of elements with a zero value. Sparse data can be easily compressed, which in turn can significantly reduce memory usage.

Sparse Matrix Representations

1. Array Representation

Each element of the array into which the sparse matrix is mapped need to have three fields:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)

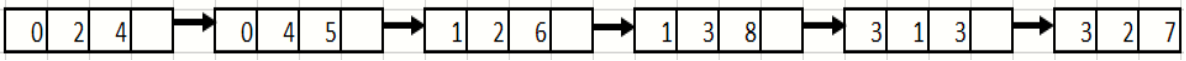
	0	1	2	3	4
0	0	0	4	0	5
1	0	0	6	8	0
2	0	0	0	0	0
3	0	3	7	0	0

row	0	0	1	1	3	3
col	2	4	2	3	1	2
val	4	5	6	8	3	7

2. Linked list representation

In the linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



4 Write algorithms for Push and Pop, Peep and Change operations on a stack.

Push (S, Top, X)

This procedure inserts an element X to the top of a stack which is represented by a vector S containing N elements with a pointer Top denoting the top element in the stack.

1. [Check for Stack overflow]
   If Top ≥ N
   then Write ('Stack Overflow')
   Return

2. [Increment Top]
   Top ← Top + 1
3. [Insert Element]
   S[Top] ← X
4. [Finished]
   Return

Pop (S, Top)

This function removes the top element from a stack which is represented by a vector S and return this element. Top is a pointer to the top element of the stack.

1. [Check for underflow on Stack]
   If Top = 0
   then Write ('Stack Underflow on Pop')
   Exit
2. [Decrement Pointer]
   Top ← Top-1
3. [Return former top element of Stack]
   Return (S[Top+1])

Peep (S, Top, I)

Given a vector S (consisting of N elements) representing a sequentially allocated stack, and a pointer Top denoting the top of the element of the stack, this function returns the value of the I^{th} element from the top of the stack. The element is not deleted by this function.

1. [Check for Stack Underflow]
   If (Top - I + 1) ≤ 0
   then Write ('Stack Underflow on Peep')
   Exit
2. [Return I^{th} element from top of the stack]
   Return (S[Top-I+1])

Change (S, Top, X, I)

As before, a vector S (consisting of N elements) represents sequentially allocated stack and a pointer Top denotes the top element of the stack. This procedure changes the value of I^{th} element from the top of the stack to the value contained in X.

1. [Check for stack underflow]
   If (Top - I + 1) ≤ 0
   then Write ('Stack Underflow on Change')
   Exit
2. [Change the I^{th} element from top of stack]
   S[Top - I + 1] ← X
3. [Finished]
   Return

5	Write a C program to implement a stack with all necessary overflow and underflow checks using array.

```
#include<stdio.h>

 #define MAX 3 // Altering this value changes size of stack created
 int st[MAX], top=-1;
 void push(int st[], int val);
 int pop(int st[]);
 int peek(int st[]);
 void display(int st[]);
 int main(int argc, char *argv[]) {
        int val, option;
```

```
	do
	{
		printf("\n ************* MAIN MENU ************");
		printf("\n 1. PUSH");
		printf("\n 2. POP");
		printf("\n 3. PEEK");
		printf("\n 4. DISPLAY");
		printf("\n 5. EXIT");
		printf("\n Enter your option: ");
		scanf("%d", &option);

		switch(option) {
			case 1:
				printf("\n Enter the number to be pushed on stack: ");
				scanf("%d", &val);
				push(st, val);
				break;
			case 2:
				val = pop(st);
				if(val != -1)
				printf("\n The value deleted from stack is: %d", val);
				break;
			case 3:
				val = peek(st);
				if(val != -1)
				printf("\n The value stored at top of stack is: %d", val);
				break;
			case 4:
				display(st);
				break;
		 }
	} while(option != 5);

	return 0;
}
void push(int st[], int val)
{
	if(top == MAX-1) {
		printf("\n STACK OVERFLOW");
	}
	else {
		top++;
		st[top] = val;
	}
}
int pop(int st[])
{
	int val;


	if(top == -1) {
		printf("\n STACK UNDERFLOW");
		return -1;
	}
	else {
```

```
                val = st[top];
                top--;
                return val;
        }
}
void display(int st[])
{
        int i;

        if(top == -1)
                printf("\n STACK IS EMPTY");
        else {
                for(i=top;i>=0;i--)
                        printf("\n %d",st[i]);
                printf("\n"); // Added for formatting purposes
        }
}
int peek(int st[])
{
        if(top == -1) {
                printf("\n STACK IS EMPTY");
                return -1;
        }
        else
                return (st[top]);
}
```

6	Write an algorithm to convert infix expression to postfix expression.
	1. [Initialize stack] Top ← 1 S[Top] ← '(' 2. [Initialize output string and rank count] POLISH ← '' RANK ← 0 3. [Get first input symbol] NEXT ← NEXTCHAR (INFIX) 4. [Translate the infix expression] Repeat thru step 7 while NEXT ≠ '' 5. [Remove symbols with greater precedence from stack] If Top < 1 then Write ('INVALID') Exit Repeat while f(NEXT) < g(S[TOP]) TEMP ← Pop (S, Top) POLISH ← POLISH O TEMP RANK ← RANK + r(TEMP) If RANK < 1 then Write ('INVALID') Exit 6. [Are there matching parentheses?] If f(NEXT) ≠ g(S[Top]) then Call Push (S, Top, NEXT) else Pop (S, Top) 7. [Get next input symbol] NEXT ← NEXTCHAR (INFIX) 8. [Is the expression valid?]

If Top ≠ 0 or RANK ≠ 1
then Write ('INVALID')
else Write ('VALID')
Exit

7 Convert following infix expressions to the postfix expressions. Shows stack trace.
A/B$C+D*E/F-G+H
(A+B)*D+E/(F+G*D)+C

A/B$C+D*E/F-G+H

Character Scanned	Content of Stack (Rightmost symbol is top of stack)	Reverse Polish Expression	Rank
	#		
A	#A		
/	#/	A	1
B	#/B	A	1
$	#/$	AB	2
C	#/$C	AB	2
+	#+	ABC$/	1
D	#+D	ABC$/	1
*	#+*	ABC$/D	2
E	#+*E	ABC$/D	2
/	#+/	ABC$/DE*	2
F	#+/F	ABC$/DE*	2
-	#-	ABC$/DE*F/+	1
G	#-G	ABC$/DE*F/+	1
+	#+	ABC$/DE*F/+G-	1
H	#+H	ABC$/DE*F/+G-	1
	#	**ABC$/DE*F/+G-H+**	1

(A+B)*D+E/(F+G*D)+C

Character Scanned	Content of Stack (Rightmost symbol is top of stack)	Reverse Polish Expression	Rank
	(		
(	((		
A	((A		
+	((+	A	1
B	((+B	A	1
)	(	AB+	1
*	(*	AB+	1
D	(*D	AB+	1
+	(+	AB+D*	1
E	(+E	AB+D*	1
/	(+/	AB+D*E	2
(	(+/(	AB+D*E	2
F	(+/(F	AB+D*EF	3
+	(+/(+	AB+D*EF	3
G	(+/(+G	AB+D*EF	3
*	(+/(+*	AB+D*EFG	4
D	(+/(+*D	AB+D*EFG	4
)	(+/	AB+D*EFGD*+	3
+	(+/+	AB+D*EFGD*+	3

C	(+/+C	AB+D*EFGD*+	3
		AB+D*EFGD*+C+/+	1

8 Write an algorithm to implement insert and delete operations in a simple queue

QINSERT (Q, F, R, N, Y)
Given F and R, pointers to the front and rear elements of a queue, a queue Q consisting of N elements, and an element Y, this procedure inserts Y at the rear of the queue. Prior to the first invocation of the procedure, F and R have been set to zero.
1. [Overflow?]
If R ≥ N
then Write ('Overflow')
Return
2. [Increment rear pointer]
R ← R + 1
3. [Insert element]
Q[R] ← Y
4. [Is front pointer properly set?]
If F = 0
then F ← 1
Return

QDELETE (Q, F, R)
Given F and R, the pointers to the front and rear elements of a queue, respectively, and the queue Q to which they correspond, this function deletes and returns the last element of the queue. Y is a temporary variable.
1. [Underflow?]
If F = 0
then Write ('Underflow')
Return (0) ('0' denotes an empty queue)
2. [Delete element]
Y ← Q[F]
3. [Queue empty?]
If F = R
then F ← R ← 0
else F ← F + 1 (Increment front pointer)
4. [Return element]
Return (Y)

9 Write a C functions for insertion and deletion operation in simple queue.

```
#include <stdio.h>
#define MAX 10 // Changing this value will change length of array
int queue[MAX];
int front = -1, rear = -1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);

int main()
{
        int option, val;
        do
```

```
	{
		printf("\n\n ***** MAIN MENU *****");
		printf("\n 1. Insert an element");
		printf("\n 2. Delete an element");
		printf("\n 3. Display the queue");
		printf("\n 4. EXIT");
		printf("\n Enter your option : ");

		scanf("%d", &option);

		switch(option)
		{
			case 1:
				insert();
				break;
			case 2:
				val = delete_element();
				if (val != -1)
					printf("\n The number deleted is : %d", val);
				break;
			case 3:
				display();
				break;
		}
	} while(option != 4);
	return 0;
}
void insert()
{
	int num;
	printf("\n Enter the number to be inserted in the queue : ");
	scanf("%d", &num);
	if(rear == MAX-1)
		printf("\n OVERFLOW");
	else if(front == -1 && rear == -1)
		front = rear = 0;
	else
		rear++;
	queue[rear] = num;
}
int delete_element()
{
	int val;
	if(front == -1 || front>rear)
	{
		printf("\n UNDERFLOW");

		return -1;
	}
	else
	{
		val = queue[front];
		front++;
		if(front > rear)
			front = rear = -1;
```

```
                return val;
        }
}
void display()
{
        int i;
        printf("\n");
        if(front == -1 || front > rear)
                printf("\n QUEUE IS EMPTY");
        else
        {
                for(i = front;i <= rear;i++)
                        printf("\t %d", queue[i]);
        }
}
```

10 Write differences between simple queue and circular queue. Write an algorithm for insert and delete operations for circular queue.

Simple queue	**Circular queue**
Arrange the data in a linear pattern.	Arranges the data in a circular order where the rear end is connected with the front end.
The insertion and deletion operations are fixed i.e, done at the rear and front end respectively.	Insertion and deletion are not fixed and it can be done in any position.
Linear queue requires more memory space.	It requires less memory space.
It is inefficient in comparison to a circular queue.	It is more efficient in comparison to a linear queue.
It follows the FIFO principle in order to perform the tasks.	It has no specific order for execution.

CQINSERT (F, R, Q, N, Y)
Given pointers to the front and rear of a circular queue, F and R, a vector Q consisting of N elements,
and an element Y, this procedure inserts Y at the rear of the queue. Initially F and R set to zero.
1. [Reset rear pointer?]
If R = N
then R ← 1
else R ← R + 1
2. [Overflow?]
If F = R
then Write (‘Overflow’)
Return
3. [Insert element]
Q[R] ← Y
4. [Is front pointer properly set?]
If F = 0
then F ← 1
Return

CQDELETE (F, R, Q, N)
Given F and R, pointers to the front and rear of a circular queue, respectively, and a vector Q

consisting of N elements, this function deletes and returns the last element of the queue. Y is a temporary variable.

1. [Underflow?]
If F = 0
then Write ('Underflow')
Return (0)
2. [Delete element]
Y ← Q[F]
3. [Queue empty?]
If F = R
then F ← R ← 0
Return (Y)
4. [Increment front pointer]
If F = N
then F ← 1
else F ← F + 1
Return (Y)

11 Write a C program to implement a circular queue using array with all necessary overflow and underflow checks.

```
#include <stdio.h>
#define MAX 5
int queue[MAX];
int front=-1, rear=-1;
void insert(void);
int delete_element(void);
void display(void);
int main()
{
        int option, val;
        do
        {
                printf("\n ***** MAIN MENU *****");
                printf("\n 1. Insert an element");
                printf("\n 2. Delete an element");
                printf("\n 3. Display the queue");
                printf("\n 4. EXIT");
                printf("\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
                        case 1:
                                insert();
                                break;
                        case 2:
                                val = delete_element();

                                if(val!=-1)
                                        printf("\n The number deleted is : %d", val);
                                break;
                        case 3:
                                display();
                                break;
                }
        } while(option!=4);
        return 0;
```

```
}
void insert()
{
	int num;
	printf("\n Enter the number to be inserted in the queue : ");
	scanf("%d", &num);
	if(front==0 && rear==MAX-1)
		printf("\n OVERFLOW");

	else if(front==-1 && rear==-1)
	{
		front=rear=0;
		queue[rear]=num;
	}
	else if(rear==MAX-1 && front!=0)
	{
		rear=0;
		queue[rear]=num;
	}
	else
	{
		rear++;
		queue[rear]=num;
	}
}
int delete_element()
{
	int val;

	if(front==-1 && rear==-1)
	{
		printf("\n UNDERFLOW");
		return -1;
	}

	val = queue[front];

	if(front==rear)
		front=rear=-1;
	else
	{
		if(front==MAX-1)
			front=0;
		else
			front++;
	}

	return val;
}
void display()
{
	int i;
	printf("\n");

	if (front ==-1 && rear==-1)
```

```
                printf ("\n QUEUE IS EMPTY");
        else
        {
                if(front<rear)
                {
                        for(i=front;i<=rear;i++)
                                printf("\t %d", queue[i]);
                }
                else
                {
                        for(i=front;i<MAX;i++)
                                printf("\t %d", queue[i]);
                        for(i=0;i<=rear;i++)
                                printf("\t %d", queue[i]);
                }
        }
}
```

12	Explain Priority Queue and Double-ended queue.

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

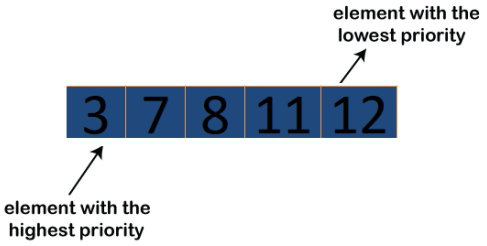
Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

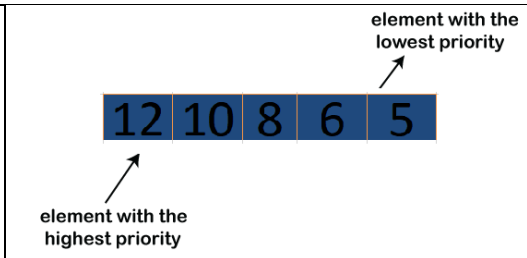
Types of Priority Queue

There are two types of priority queue:



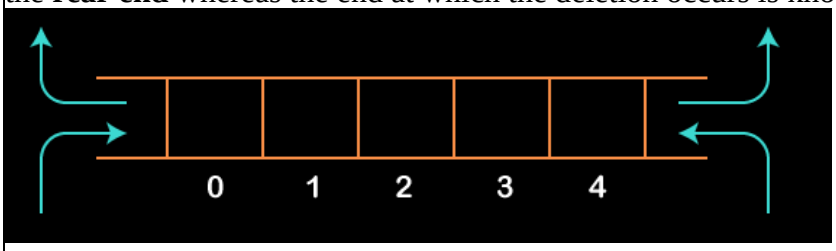
Ascending order priority queue: In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.

Descending order priority queue: In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority



in a priority queue.

The dequeue stands for **Double Ended Queue**. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the **rear end** whereas the end at which the deletion occurs is known as **front end**.

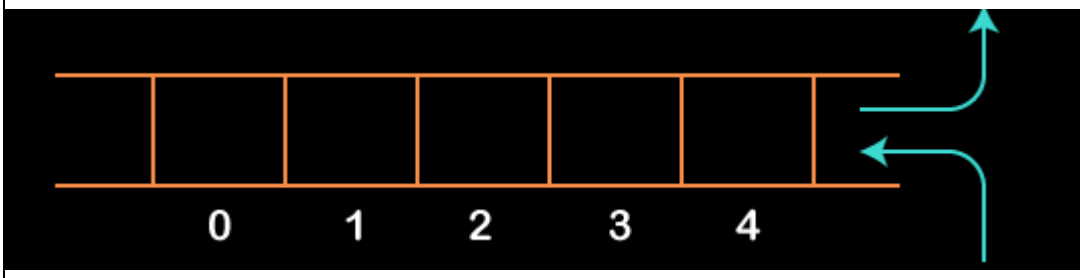


Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

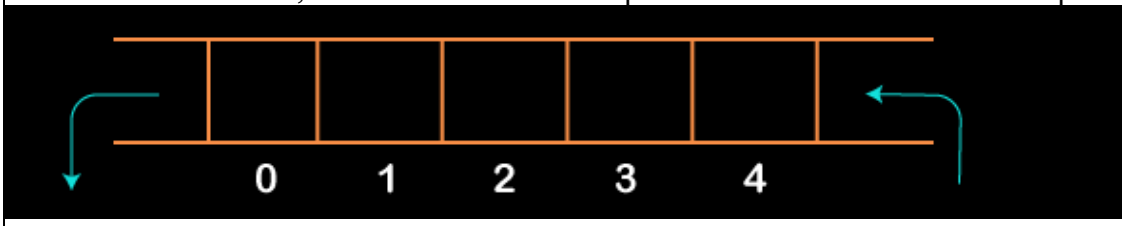
Let's look at some properties of deque.

- Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that deque can be considered as a stack.



In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.



There are two types of Queues, **Input-restricted queue**, and **output-restricted queue**.

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.
2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.

Operations on Deque
The following are the operations applied on deque:

- **Insert at front**
- **Delete from end**
- **insert at rear**
- **delete from rear**

Other than insertion and deletion, we can also perform **peek** operation in deque.
Through **peek** operation, we can get the **front** and the **rear** element of the dequeue.
We can perform two more operations on dequeue:

- **isFull():** This function returns a true value if the stack is full; otherwise, it returns a false value.
- **isEmpty():** This function returns a true value if the stack is empty; otherwise it returns a false value.

13 Write an algorithm/program to implement following operations in the Singly Link List Singly Link List (1) Insert at first position (2) Insert at end of list.

(1) Insert at first position
INSERT (X, FIRST)
Given X, a new element, and FIRST, a pointer to the first element of a linked linear list whose typical node contains INFO and LINK fields as previous described, this function inserts X. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. It is required that X precede the node whose address is given by FIRST.

1. [Underflow?]
If AVAIL = NULL
then Write ('Availability Stack Underflow') Return (FIRST)

2. [Obtain address of next free node]
NEW ← AVAIL

3. [Remove free node from availability stack]
AVAIL ← LINK (AVAIL)

4. [Initialize fields of new node and its link to the list]
INFO (NEW) ← X
LINK (NEW) ← FIRST

5. [Return address of new node]
Return (NEW)

(2)Insert at end of list

INSEND (X, FIRST)
Given X, a new element, and FIRST, a pointer to the first element of a linked linear list whose typical node contains INFO and LINK fields as previously described, this function inserts X. AVAIL is a pointer to the top element of the availability stack; NEW and SAVE are temporary pointer variables. It is required that X be inserted at the end of the list.

1. [Underflow?]
If AVAIL = NULL
then Write ('Availability Stack Underflow') Return (FIRST)

2. [Obtain address of next free node] NEW ← AVAIL

3. [Remove free node from the availability stack] AVAIL ← LINK (AVAIL)

4. [Initialize fields of new node]
INFO (NEW) ← X LINK (NEW) ← NULL

5. [Is the list empty?]
If FIRST = NULL
then Return (NEW)

6. [Initiate search for the last node] SAVE ← FIRST

7. [Search for end of list]
Repeat while LINK (SAVE) ≠ NULL SAVE ← LINK (SAVE)

8. [Set LINK field of last node to NEW] LINK (SAVE) ← NEW

9. [Return first node pointer]
Return (FIRST)

14	Write an algorithm for insertion and deletion of a node in Doubly Linked List

(1) Insert in Doubly Linked List

DOUBINS (L, R, M, X)
Given a doubly linked linear list whose left-most and right-most node addresses are given by the pointer variables L and R, respectively, it is required to insert a node whose address is given by the pointer variable NEW. The left and right links of a node are denoted by LPTR and RPTR, respectively. The information field of a node is denoted by the variable INFO. The name of an element of the list is NODE. The insertion is to be performed to the left of a specified node with its address given by the pointer variable M. The information to be entered in the node is contained in X.

1.[Obtain new node from availability stack] NEW ⇐ NODE

2.[Copy information field]
INFO (NEW) ← X

3.[Insertion into an empty list?]
If R = NULL
then LPTR (NEW) ← RPTR (NEW) ← NULL L ← R ← NEW
Return

4.[Left-most insertion?]
If M = L
then LPTR (NEW) ← NULL RPTR (NEW) ← M LPTR (M) ← NEW
L ← NEW
Return

5.[Insert in middle]
LPTR (NEW) ← LPTR (M) RPTR (NEW) ← M
LPTR (M) ← NEW
RPTR (LPTR (NEW)) ← NEW
Return

(2) Delete in Doubly Linked List

DOUBDEL (L, R, OLD)
Given a doubly linked list with the addresses of the left-most and right-most nodes given by the pointer variables L and R, respectively, it is required to delete the node whose address is contained in the variable OLD. Nodes contain left and right links with names LPTR and RPTR, respectively.

1.[Underflow?]
If R = NULL
then Write ('Underflow') Return

2.[Delete node]
If L = R (Single node in list) then L ← R ← NULL
else If OLD = L (Left-most node being deleted) then L ← RPTR (L)
LPTR (L) ← NULL
else if OLD = R (Right-most node being deleted) then R ← LPTR (R)
RPTR (R) ← NULL
else RPTR (LPTR (OLD)) ← RPTR (OLD) LPTR (RPTR (OLD)) ← LPTR (OLD)

3.[Return deleted node]
Restore (OLD) Return

15	List applications of Stack, Queue and Linked list.

Applications of Stack:
- Reversing a list, Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression, Recursion
- Tower of Hanoi

Applications of Queue:
-In waiting lists for a single shared resource like printer, disk, CPU.
-Used to transfer data asynchronously,-
-Used as buffers on MP3 players
-Used in operating system for handling interrupts

Applications of Linked List:
-Implementation of stacks and queues
-Implementation of graphs: Adjacency list representation of graphs is most popular which uses linked lists to store adjacent vertices.
-Dynamic memory allocation: We use a linked list of free blocks.
-Maintaining directory of names
-Performing arithmetic operations on long integers
-Manipulation of polynomials by storing constants in the node of linked list representing sparse matrices

16	Write down algorithm for sequential search and binary search method

	LINEAR_SEARCH(A, N, VAL) Step 1: [INITIALIZE] SET POS = -1 Step 2: [INITIALIZE] SET I = 1 Step 3: Repeat Step 4 while I<=N Step 4: IF A[I] = VAL SET POS = I PRINT POS Go to Step 6 [END OF IF] [END OF LOOP] Step 6: EXIT SET I = I + 1 Step 5: IF POS = –1 PRINT VALUE IS NOT PRESENT IN THE ARRAY [END OF IF] Step 6: EXIT **BINARY_SEARCH(A, lower_bound, upper_bound, VAL)** Step 1: [INITIALIZE] SET BEG = lower_bound END = upper_bound, POS = - 1 Step 2: Repeat Steps 3 and 4 while BEG <= END Step 3: SET MID = (BEG + END)/2 Step 4: IF A[MID] = VAL SET POS = MID PRINT POS Go to Step 6 ELSE IF A[MID] > VAL SET END = MID - 1 ELSE SET BEG = MID + 1 [END OF IF] [END OF LOOP] Step 5: IF POS = -1 PRINT “VALUE IS NOT PRESENT IN THE ARRAY” [END OF IF] Step 6: EXIT
17	Explain the trace of selection sort, insertion sort and bubble sort on following data. 42,23,74,11,65,58,94,36,99,87

Bubble Sort

Pass: 0

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

Compare two elements and swap if element at right is greater

as 23 < 42, Swap

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

as 74 > 42 already, No Swap

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

as 11 < 74, Swap

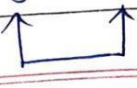
4. 23 42 11 74 65 58 94 36 99 87
   0 1 2 3 4 5 6 7 8 9

as 65 < 74, Swap

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

as 58 < 74, Swap

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



as 94 > 74, No Swap already

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

as 36 < 94, Swap

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

as 99 > 94 already, No swap

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

as 87 < 99, Swap

10. 23 42 11 65 58 74 36 94 87 **99**

final position

Pass-0 end

Pass: 1

1. 23 42 11 65 58 74 36 94 87 99

42 > 23 already, No swap

2. 23 42 11 65 58 74 36 94 87 99

11 < 42, swap

3. 23 11 42 65 58 74 36 94 87 99

65 > 42, No swap

4. 23 11 42 65 58 74 36 94 87 99

58 < 65, swap

5. 23 11 42 58 65 74 36 94 87 99

74 > 65, No swap

6. 23 11 42 58 65 74 36 94 87 99

36 < 74, swap

7. 23 11 42 58 65 36 74 94 87 99

94 > 74, No swap

8. 23 11 42 58 65 36 74 94 87 99

87 < 94, Swap

9. 23 11 42 58 65 36 74 87 94 99

99 > 94, No swap

23 11 42 58 65 36 74 87 [94] [99]

final position

Pass: 2

1. 23 11 42 58 65 36 74 87 94 99

11 < 23, Swap

2. 11 23 42 58 65 36 74 87 94 99

42 > 23, No Swap

3. 11 23 42 58 65 36 74 87 94 99

58 > 42, No swap

4. 11 23 42 58 65 36 74 87 94 99

65 > 58, No swap

5. 11 23 42 58 65 36 74 87 94 99

36 < 65 swap

6. 11 23 42 58 36 65 74 87 94 99

74 > 65, No swap

7. 11 23 42 58 36 65 74 87 94 99

87 > 74, No swap

11 23 42 58 36 65 74 87 94 99

final position

Pass: 3

1. 11 23 42 58 36 65 74 87 94 99

23 > 11, No Swap

2. 11 23 42 58 36 65 74 87 94 99

42 > 23, No Swap

3. 11 23 42 58 36 65 74 87 94 99

58 > 42, No Swap

4. 11 23 42 58 36 65 74 87 94 99

36 < 58, Swap

5. 11 23 42 36 58 65 74 87 94 99

74 > 65, No Swap

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

Pass: 4

1. 11 23 42 36 58 65 74 87 94 99

23 > 11, No swap

2. 11 23 42 36 58 65 74 87 94 99

42 > 23, No Swap

3. 11 23 42 36 58 65 74 87 94 99

36 < 42; Swap

4. 11 23 36 42 58 65 74 87 94 99
58 > 42, NO swap

5. 11 23 36 42 58 65 74 87 94 99
65 > 58, NO Swap

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

Pass: 4

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

Pass: 5

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

Pass: 6

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

Pass: 7

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

Pass: 8

11 23 36 42 58 65 74 87 94 99

Selection Sort

Steps
- Set 1st element as minimum
- Compare minimum with all elements.
- If the element is smaller than minimum, assign it as minimum
- After each pass, the smallest element is placed at the start of unsorted list

Pass: 01

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

1. 42 23 74 11 65 58 94 36 99 87
min --> min
- as 23 is smaller than 42, min is updated and will point to 23.

2. 42 23 74 11 65 58 94 36 99 87
min
Now 11 < 23, update min

3. 42 23 74 11 65 58 94 36 99 87
i=0 min
Scan j until whole list is Completed. but NO smaller element is found., so final minimum is 11.

- Swap Min with i^{th} index
11 23 74 42 65 58 94 36 99 87
final position

Pass: 2

i=1 j=2

11	23	74	42	65	58	94	36	99	87

Min

j=2, as 23 < 74, Min Not updated &
j=3 23 < 42, Min not updated
j=4 23 < 65, Min not updated
j=5 23 < 58, Min not updated
j=6 23 < 94, Min not updated
j=7 23 < 36, Min not updated
j=8 23 < 99, Min not updated
j=9 23 < 87, Min not updated

- as Minimum is already pointing to the smallest element of unsorted list, 23 is fixed at it's current position

11	23	74	42	65	58	94	36	99	87

Pass: 3

i=2 j=3

11	23	74	42	65	58	94	36	99	87

min ↑min min

j=3 42 < 74, update Min Now Min = 42
j=4 42 < 65, Min not updated
j=5 42 < 58, Min not updated
j=6 42 < 94, Min not updated
j=7 42 > 36, update Min, Now Min = 36
j=8 36 < 99 Min not updated
j=9 36 < 87 Min not updated

→ final Min points to 36, swap it with A[i] = 74

11 23 [36] 42 65 58 94 74 99 87

min

Pass: 4 i=3 j=4

11 23 36 42 65 58 94 74 99 87

min ↑ ↑

Current min is 42, Comparing it with the rest of the elements, doesn't update Min

- 42 will be fixed at current position

Pass-5 i=4 j=5

11 23 36 [42] 65 58 94 74 99 87

min ↑ ↑

j=5

5 58 < 65, update Min, Now Min = [58]

No other elements are smaller than 58 for j=6 to j=9.

- Swap A[i] = 65 with final Min = 58

11 23 36 42 [58] 65 94 74 99 87

Pass: 6 i=5 j=6

11 23 36 42 58 65 94 74 99 87

min ↑ ↑

94 > 65, No update

74 > 65, No update

99 > 65, No update

87 > 65, No update

65 is already at Min

Pass: 7

i=6 j=7

11 23 36 42 58 65 94 74 99 87

min ↑ ↑ min

- 74 < 94, update Min
- No elements are smaller than 74.
- Swap A[i] = 94 ⟷ 74

11 23 36 42 58 65 74 94 99 87

Pass: 8

i=7 j=8

11 23 36 42 58 65 74 94 99 87

min ↑ ↑ ↑ min

87 < 94 update Min

Swap A[i] = 94 ⟷ 87

11 23 36 42 58 65 74 87 99 94

Pass: 9

i=8 j=9

11 23 36 42 58 65 74 87 99 94

min ↑ ↑

94 < 99 update Min

Swap A[i] = 99 ⟷ 94

11 23 36 42 58 65 74 87 94 99

The list is sorted now.

Insertion Sort

42, 23, 74, 11, 65, 58, 94, 36, 99, 87 — Assume 42 is a Sorted list

i=0, j=1; Sorted | unsorted

→ 23, 42, 74, 11, 65, 58, 94, 36, 99, 87 | 42 > 23, Shift it to the right

→ 23, 42, 74, 11, 65, 58, 94, 36, 99, 87 | 74 is at its correct place

→ 23, 42, 74, 11, 65, 58, 94, 36, 99, 87 | 11 < 74, 11 < 42, 11 < 23

→ Shift 74, 42 and 23 at right respectively

11, 23, 42, 74, 65, 58, 94, 36, 99, 87 | Compare 65 now

as 65 < 74 ⇒ Shift 74 at right

→ 11, 23, 42, 65, 74, 58, 94, 36, 99, 87 | 58 < 74, 58 < 65

Shift 74 and 65 at right

→ 11, 23, 42, 58, 65, 74, 94, 36, 99, 87 | 94 is already at correct place

→ 11, 23, 42, 58, 65, 74, 94, 36, 99, 87

36 < 94, 74, 65, 58 and 42 respectively, Shift all these elements at right

→ 11, 23, 36, 42, 58, 65, 74, 94, 99, 87

11, 23, 36, 42, 58, 65, 74, 94, 99, 87 | 99 at Correct position

Compare 87 NOW,

87 < 99, 94 respectively, shift both at right

11, 23, 36, 42, 58, 65, 74, 87, 94, 99 [list Sorted]

18 Sort the following numbers in ascending order by applying quick sort.
29 15 11 82 22 17 53 57 4 8
Sort the following numbers in ascending order by applying merge sort.
2 4 1 6 8 5 3 7

Quick Sort

29 15 11 82 17 53 57 48
pivot i i i j j j

Quick Sort

1. 29 15 11 82 22 17 53 57 48
pivot i i i j j j

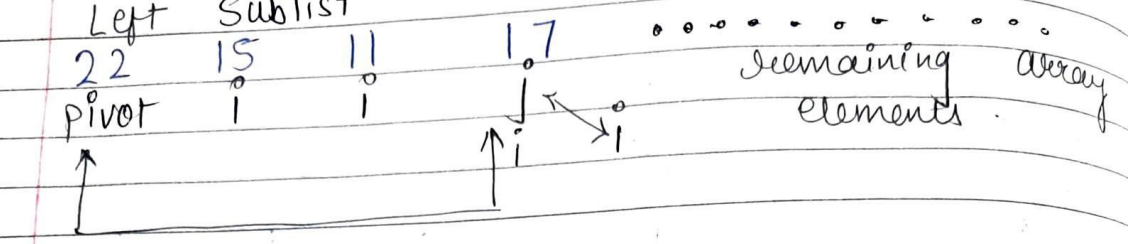
Steps : increment i, till the elements are greater than pivot
- decrement j, till the elements are smaller than pivot
- swap A[i] and A[j]
- When index i and j crosses, each other, swap A[j] <-> pivot

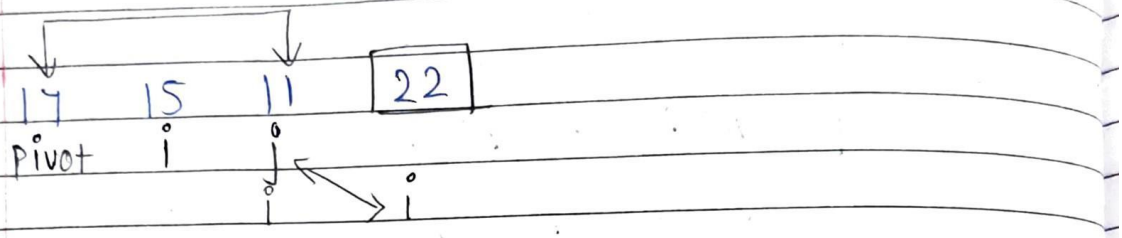
2. 29 15 11 17 22 82 53 57 48
i and j crossed
A[j] <-> pivot

3. 22 15 11 17 [29] 82 53 57 48
Left Sublist
final position of first pivot
Right Sublist

→ Apply Same procedure to left and right Sublist.

Left Sublist

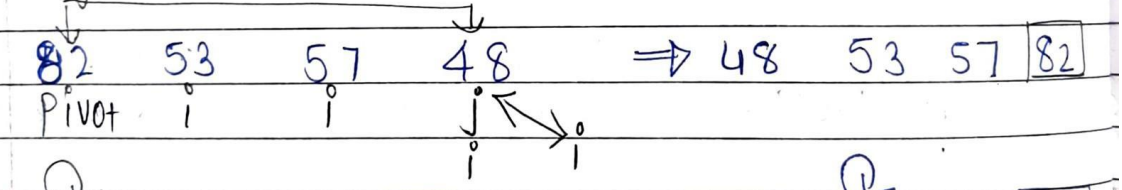


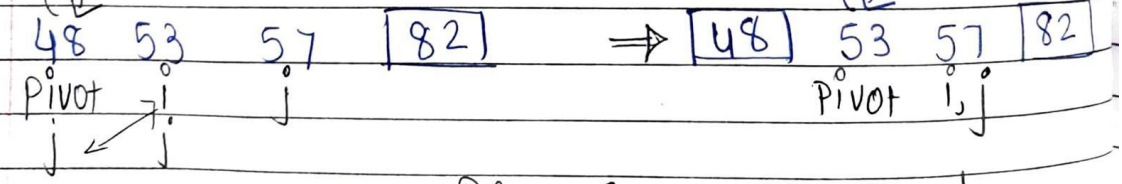


11 15 17 22 left list Now Sorted

Pivot i, j

→ Right sublist



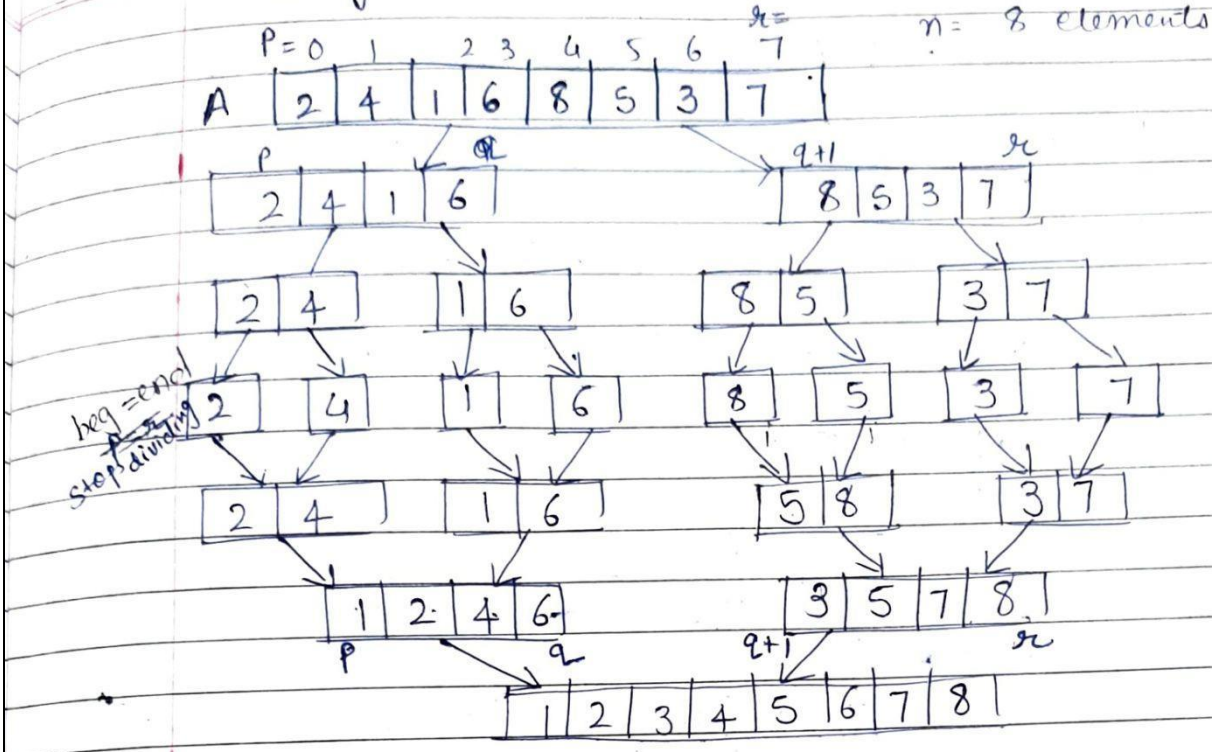


Right list Now sorted

→ Combine all the elements together we get sorted array as follows.

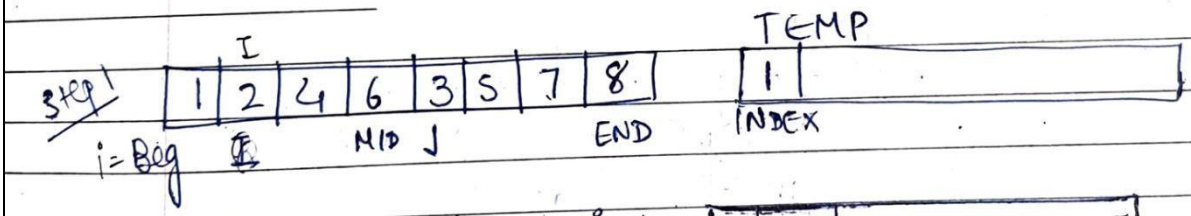
11, 15, 17, 22, 29, 48, 53, 57, 82

Merge Sort

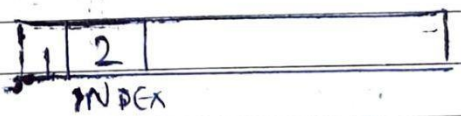


L 1 2 4 6 R 3 5 7 8

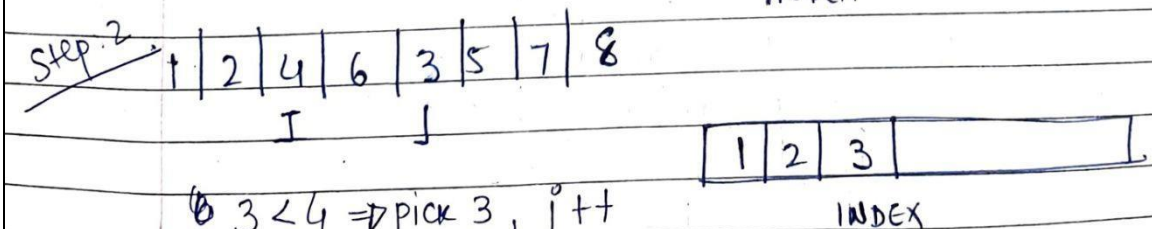
i = beg



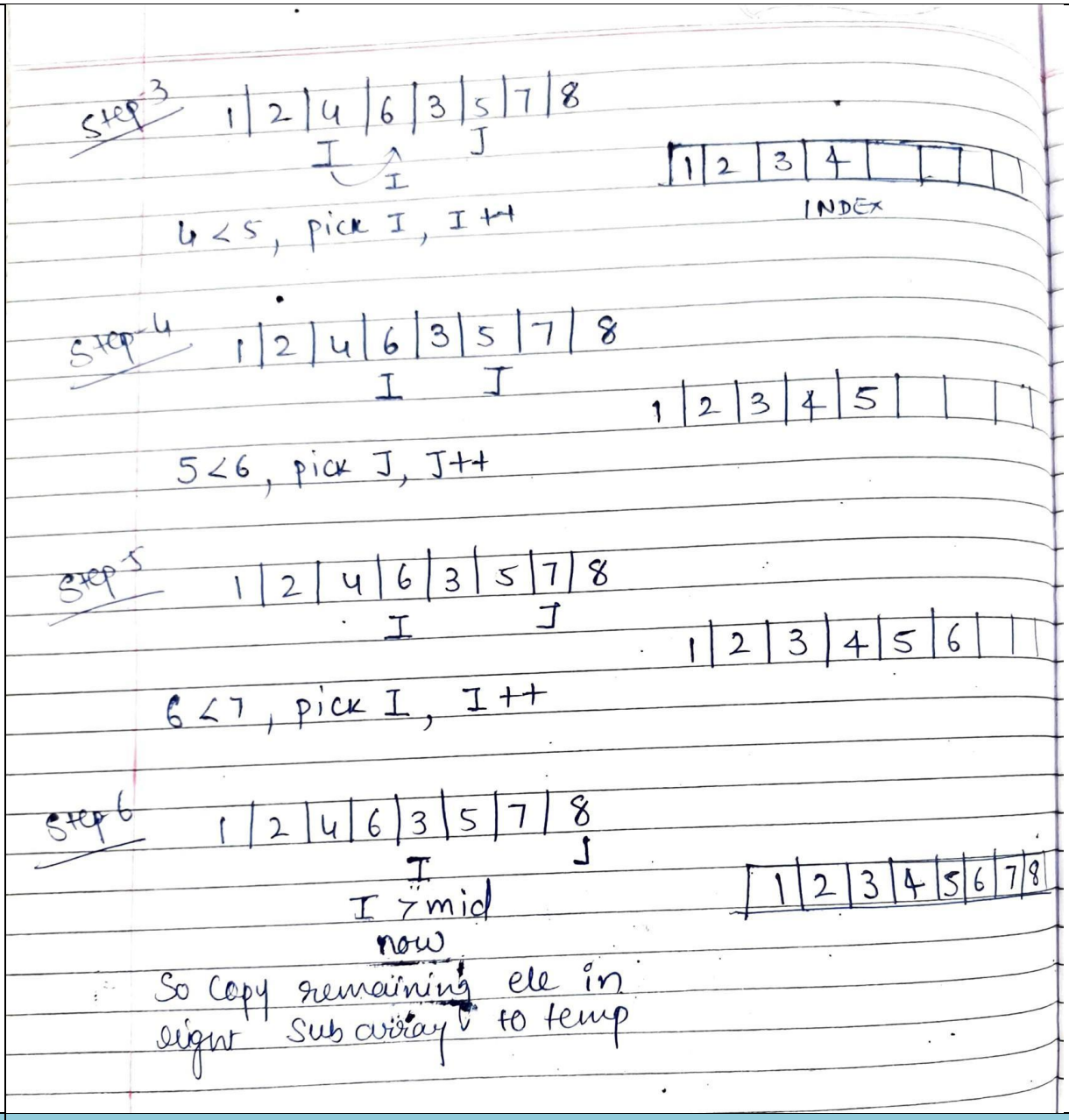
2 < 3 ⇒ pick 2, i++



step 2



3 < 4 ⇒ pick 3, j++



19	Write an algorithm for evaluation of postfix expression and evaluate the following expression showing every status of stack in tabular form. 5 6 2 - * 4 9 3 / + *

Algorithm for evaluation of postfix expression

1. Read the postfix expression from left to right
2. If the input symbol read is an operand then push it onto the stack
3. If the operator is read, pop two operands and perform arithmetic operation
4. Push the result onto the stack
5. Repeat steps 1-24 till the postfix expression is over.

Evaluate: 5 6 2 - * 4 9 3 / +

562-*493/+*

Current Symbol	Action Taken	Stack	Output
5	Push 5	5	-
6	Push 6	5|6	-
2	Push 2	5|6|2	-
-	[Pop 2 and 6 Perform 2-6]	5|8	-
*	Pop 8 and 5 Perform 5*8	40	-
4	Push 4	40|4	-
9	Push 9	40|4|9	-
3	Push 3	40|4|9|3	-
/	Pop 3 and 9 Perform 9/3	4 40|4|3	-
+	Pop 3 and 4 Perform 3+4	40|7	-
*	Pop 7 and 40 Perform 40*7	280	280

20	Define recursion. Give a recursive solution for the problem of "Towers of Hanoi" and trace with n=3 disks.

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH).

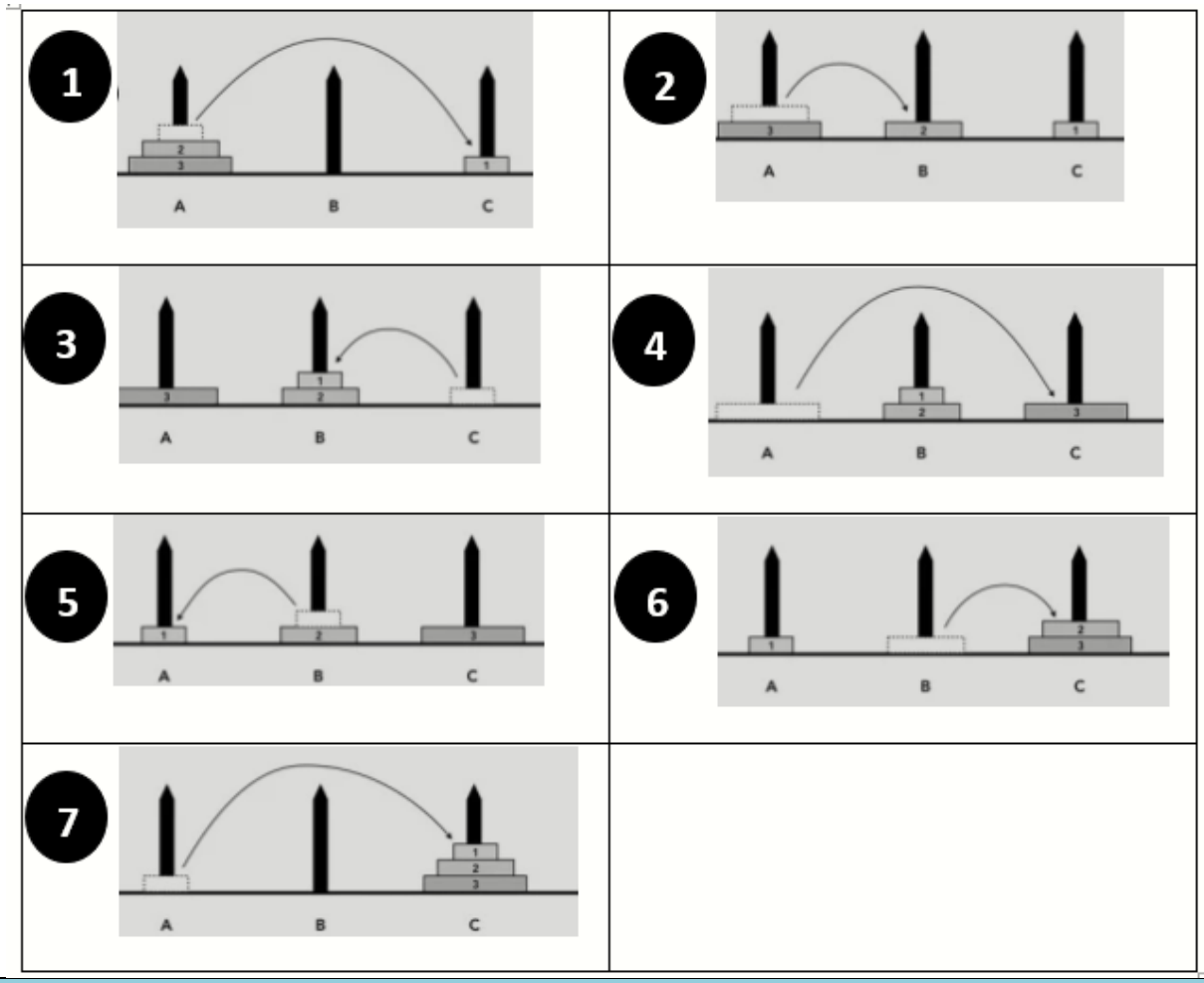
Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

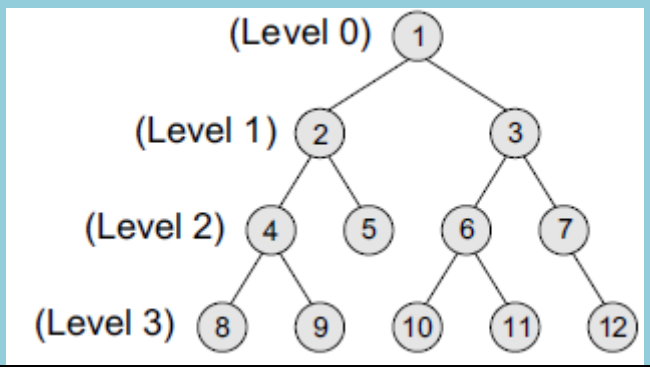
For a given N number of disks, the way to accomplish the task in a minimum number of steps is:
Move the top (N-1) disks to an intermediate peg.
Move Nth disk to the destination peg.
Finally, move the (N-1) disks from the intermediate peg to the destination peg.

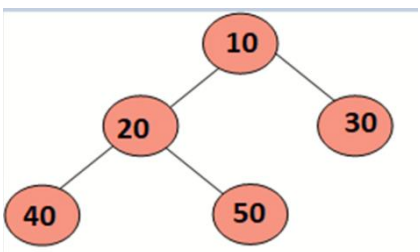
Steps for 3 disks :
Disk 1 moved from A to C

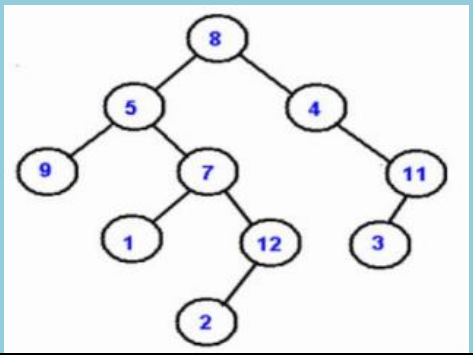
Disk 2 moved from A to B
Disk 1 moved from C to B
Disk 3 moved from A to C
Disk 1 moved from B to A
Disk 2 moved from B to C
Disk 1 moved from A to C

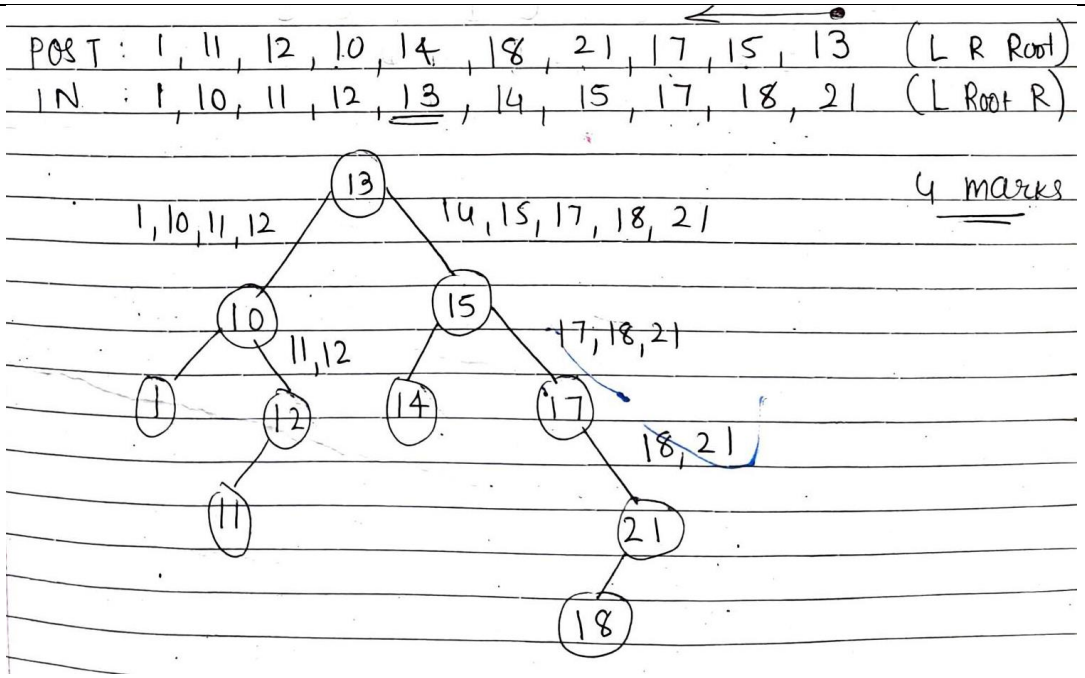


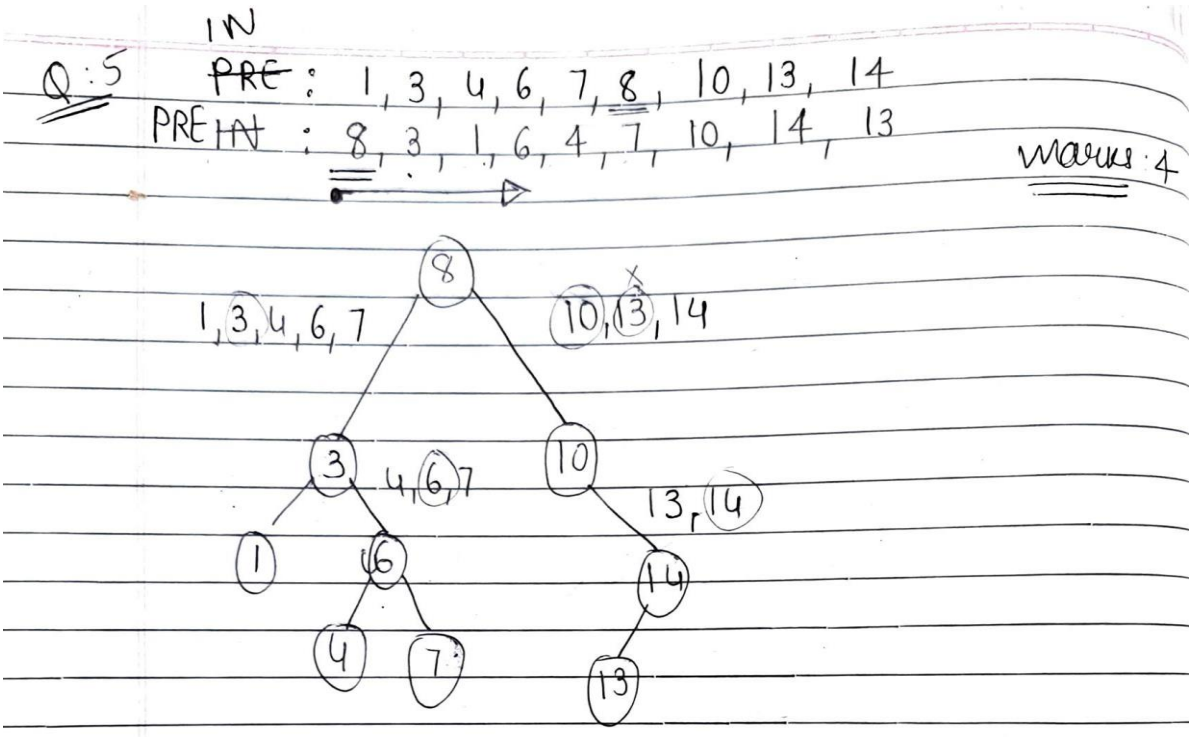
21 Discuss following with reference to trees.
(1) Height of the tree (2) Binary tree (3)Strictly binary tree (4) Sibling (5) Complete binary tree (6) Depth of tree (7) Perfect binary tree (8) Path (9) Degree of vertex



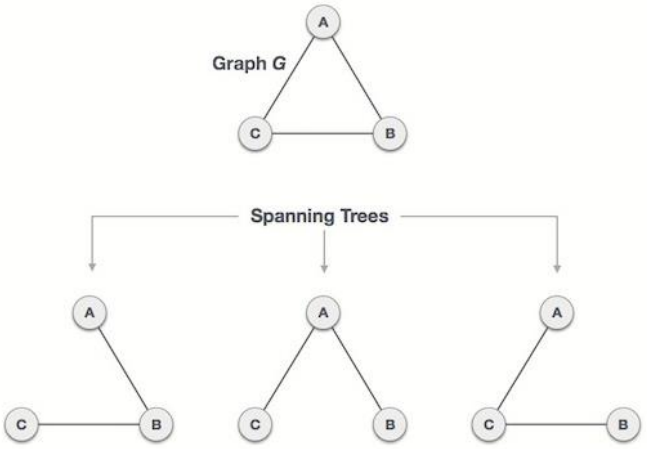
	(1) Height of the tree: It is the total number of nodes on the path from the root node to the deepest node in the tree. A tree with only a root node has a height of 1. A binary tree of height h has at least h nodes and at most $2^h - 1$ nodes. (2) Binary tree: A binary tree is a tree data structure composed of nodes, each of which has at most, two children, referred to as left and right nodes. In a binary tree, the topmost element is called the root node (3)Strictly binary tree: A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree (4) Sibling: All nodes that are at the same level and share the same parent are called siblings (brothers). (5) Complete binary tree: A complete binary tree is a binary tree that satisfies two properties. First, in a complete binary tree, every level, except possibly the last, is completely filled. Second, all nodes appear as far left as possible. (6) 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. (7) Perfect binary tree: A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level. (8) Path: A sequence of consecutive edges. For example, the path from the root node to the node 8 is given as: 1, 2, 4, and 8. (9) Degree of vertex: 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.
22	What is binary tree traversal? What are the various traversal methods? Explain all two with suitable example.
	Traversal is a process to visit all the nodes of a tree and may print their values too. Because all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree − ● In-order Traversal ● Pre-order Traversal ● Post-order Traversal Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.  **Inorder Traversal :** Algorithm Inorder(tree) 1. Traverse the left subtree, i.e., call Inorder(left-subtree) 2. Visit the root. 3. Traverse the right subtree, i.e., call Inorder(right-subtree) Example: In order traversal for the above-given figure is

	Inorder (Left, Root, Right) : 40 20 50 10 30 **Preorder Traversal:** Algorithm Preorder(tree) 1. Visit the root. 2. Traverse the left subtree call Preorder(left-subtree) 3. Traverse the right subtree call Preorder(right-subtree) Example: Preorder traversal for the above-given figure is Preorder (Root, Left, Right) : 10 20 40 50 30 **Postorder Traversal:** Algorithm Postorder(tree) 1. Traverse the left subtree call Postorder(left-subtree) 2. Traverse the right subtree call Postorder(right-subtree) 3. Visit the root. Example: Preorder traversal for the above-given figure is Postorder (Left, Right, Root) : 40 50 20 30 10
23	Perform inorder, postorder and preorder traversals for the following binary tree. 
	INORDER: 9 5 1 7 2 12 8 4 3 11 POSTORDER: 9 1 2 12 7 5 3 11 4 8 PREORDER: 8 5 9 7 1 12 2 4 11 3
24	a) Construct a binary tree from the traversal b) ls given below: Inorder: 1, 10, 11, 12, 13, 14, 15, 17, 18, 21 Postorder: 1, 11, 12, 10, 14, 18, 21, 17, 15, 13 c) Construct a binary tree from the traversals given below: Preorder traversal: 8 3 1 6 4 7 10 14 13 Inorder traversal: 1 3 4 6 7 8 10 13 14





25	Write a short note on: spanning tree , threaded binary tree
	A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.



A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

Properties of Spanning Tree

Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.

THREADED BINARY TREE

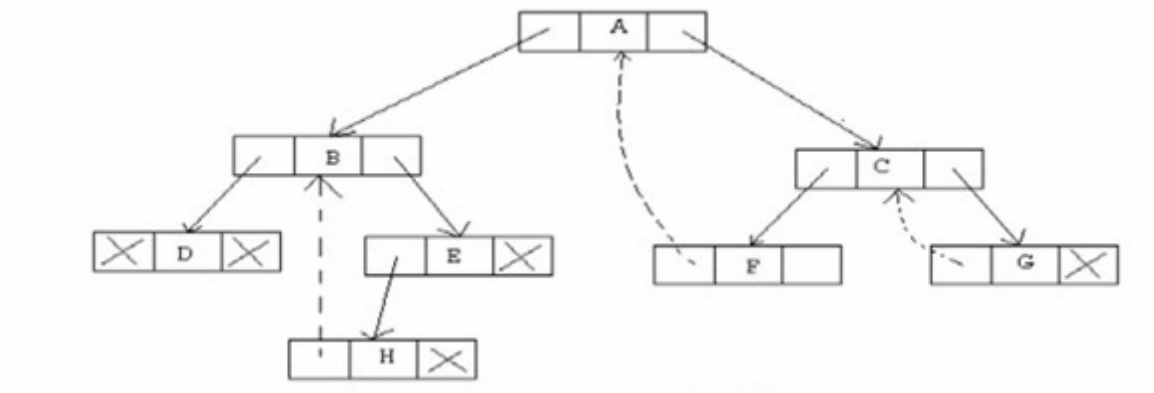
Consider the linked representation of a binary tree T, approximately half of the entries in left pointer field and right pointer field contains NULL elements. This space occupied by NULL entries can be efficiently utilized to store some kind of valuable information. These special pointers are called threads, and the binary tree having such pointers is called a threaded binary tree.

Threads in a binary tree are represented by a dotted line. There are many ways to thread a binary tree these are—

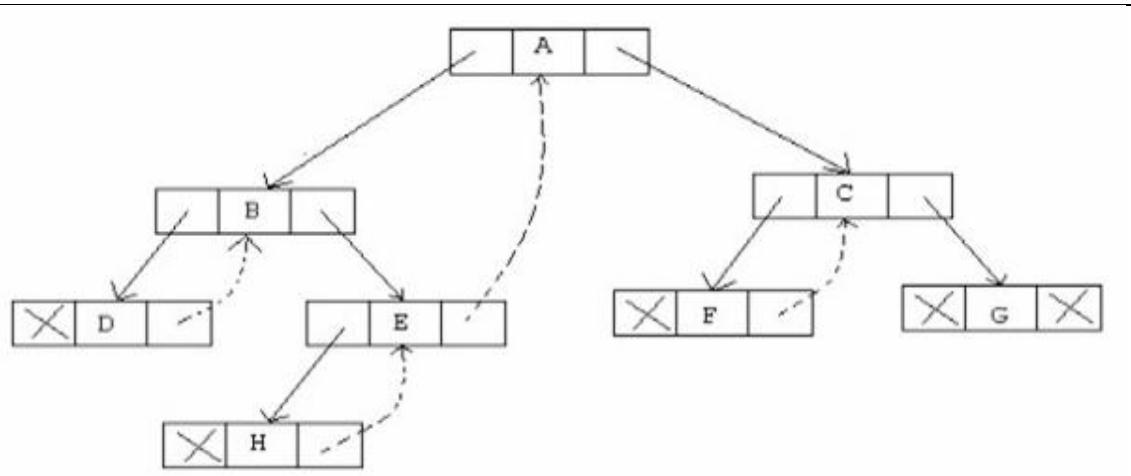
1. The right NULL pointer of each leaf node can be replaced by a thread to the successor of that node under in order traversal called a right thread, and the tree will called a right threaded tree or right threaded binary tree.
2. The left NULL pointer of each node can be replaced by a thread to the predecessor of that node under in order traversal called left thread, and the tree will called a left threaded tree.
3. Both left and right NULL pointers can be used to point to predecessor and successor of that node respectively, under in order traversal. Such a tree is called a fully threaded tree.

A threaded binary tree where only one thread is used is also known as one way threaded tree and where both threads are used is also known as two way threaded tree.

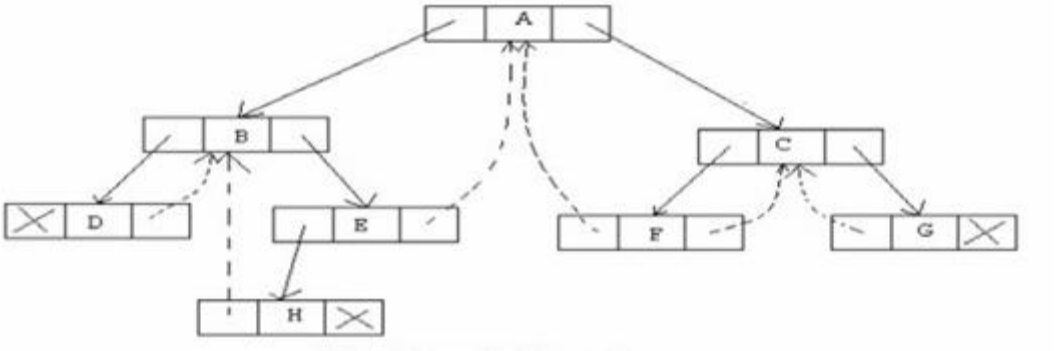
LEFT THREADED BINARY TREE



RIGHT THREADED BINARY TREE



FULL THREADED BINARY TREE



Memory Representatin Of Treaded Binayr Tree

typedef struct node
{
struct node *left
int info;
boolean lthread;
Boolean rthread
struct node *right
)
LTHREAD is true if left null pointer present and RTHREAD is true if right null pointer is present.

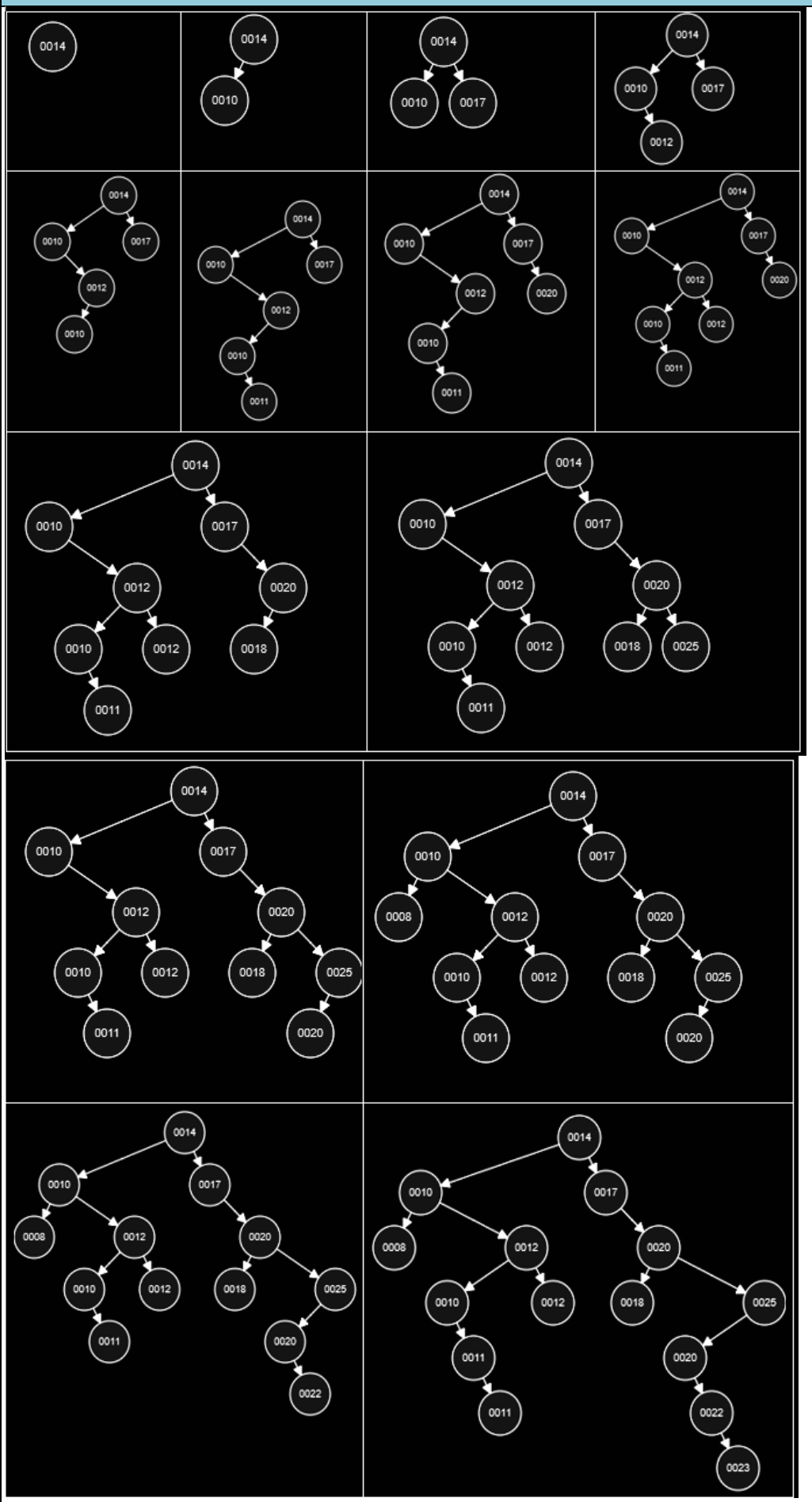
26	What is Binary Search Tree? Write recursive algorithm/program to implement in-order and pre-order traversal of the Binary Search Tree

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

struct node
{
        int data;
        struct node *left;
        struct node *right;
};

struct node *tree;
void create_tree(struct node *);
struct node *insertElement(struct node *, int);
void preorderTraversal(struct node *);
void inorderTraversal(struct node *);
void postorderTraversal(struct node *);
  int main()
```

```
{
	int option, val;
	struct node *ptr;
	create_tree(tree);

	do
	{
		printf("\n ******MAIN MENU******* \n");
		printf("\n 1. Insert Element");
		printf("\n 2. Preorder Traversal");
		printf("\n 3. Inorder Traversal");
		printf("\n 4. Postorder Traversal");
printf("\n 5. Exit");
		printf("\n\n Enter your option : ");
		scanf("%d", &option);
		switch(option)
		{
		case 1:
			printf("\n Enter the value of the new node : ");
			scanf("%d", &val);
			tree = insertElement(tree, val);
			break;
		case 2:
			printf("\n The elements of the tree are : \n");
			preorderTraversal(tree);
			break;
		case 3:
			printf("\n The elements of the tree are : \n");
			inorderTraversal(tree);
			break;
		case 4:
			printf("\n The elements of the tree are : \n");
			postorderTraversal(tree);
			break;
}
	} while(option!=5);

	return 0;

}
void create_tree(struct node *tree)
{
	tree = NULL;
}

struct node *insertElement(struct node *tree, int val)
{
	struct node *ptr, *nodeptr, *parentptr;
	ptr = (struct node*)malloc(sizeof(struct node));
	ptr->data = val;
	ptr->left = NULL;
	ptr->right = NULL;

	if(tree==NULL)
	{
```

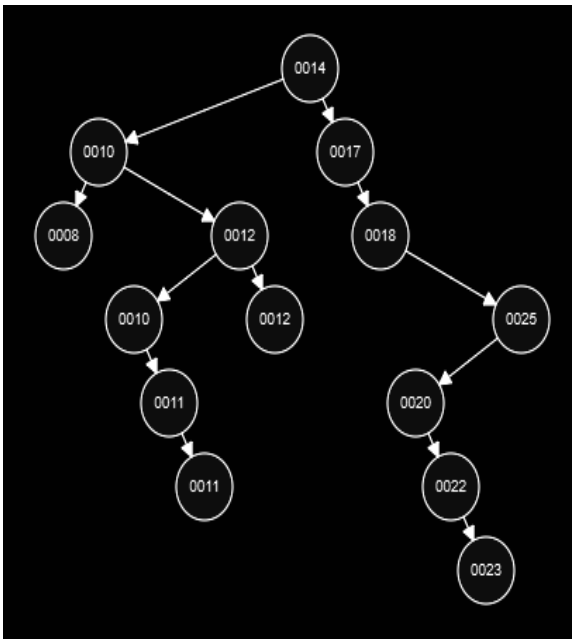
```
                tree=ptr;
                tree->left=NULL;
                tree->right=NULL;
        }
        else
        {
                parentptr=NULL;
                nodeptr=tree;
                while(nodeptr!=NULL)
                {
                parentptr=nodeptr;
                if(val<nodeptr->data)
                        nodeptr=nodeptr->left;
                else
                        nodeptr = nodeptr->right;
                 }
                if(val<parentptr->data)
                parentptr->left = ptr;
                else
                parentptr->right = ptr;
        }
        return tree;
}

void preorderTraversal(struct node *tree)
{
        if(tree != NULL)
        {
                printf("%d\t", tree->data);
                preorderTraversal(tree->left);
                preorderTraversal(tree->right);
        }
}

void inorderTraversal(struct node *tree)
{
        if(tree != NULL)
        {
                inorderTraversal(tree->left);
                printf("%d\t", tree->data);
                inorderTraversal(tree->right);
        }
}

void postorderTraversal(struct node *tree)
{
        if(tree != NULL)
        {
                postorderTraversal(tree->left);
                postorderTraversal(tree->right);
                printf("%d\t", tree->data);
        }
}
```

27	What is a binary search tree? Create a binary search tree for the following data. 14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23 Explain deleting node 20 in the resultant binary search tree.



When we delete a node, three possibilities arise.
1) ***Node to be deleted is the leaf:*** Simply remove from the tree.
2) Node to be deleted has only one child: Copy the child to the node and delete the child
3) ***Node to be deleted has two children:*** Find its inorder successor(largest element from left) of the node and replace the node to be deleted with it.

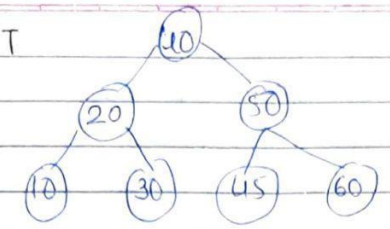
Delete node 20: Its Inorder successor (largest value from left subtree) is 18



28	What is an AVL tree? Explain the different types of rotations used to create an AVL tree with suitable examples

AVL Trees

BST

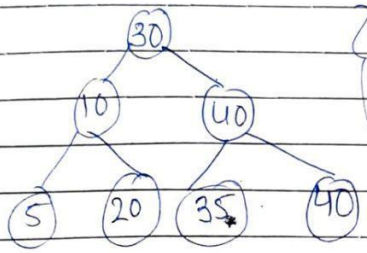


Key = 30
Key = 60
Key = 32 ✗

min: log n max : n (Time to search element)

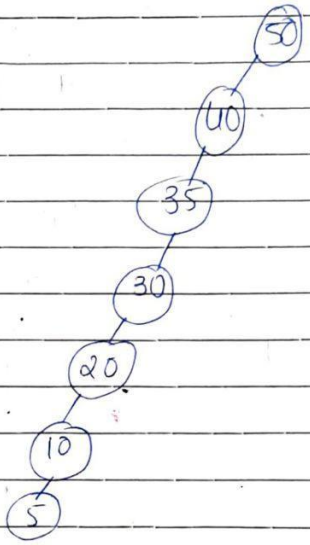
insert elements in BST

30, 40, 10, 50, 20, 5, 35



log n

50, 40, 35, 30, 20, 10, 5

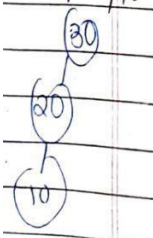


height = n

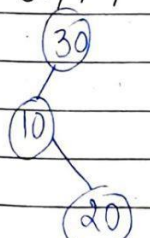
problem in BST: height is not balanced

AVL: self balancing binary search tree

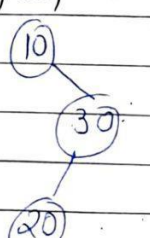
30, 20, 10



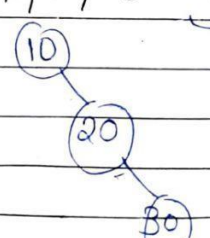
30, 10, 20



10, 30, 20



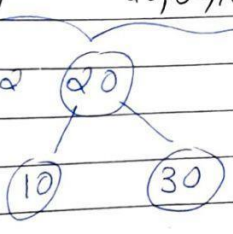
10, 20, 30



height: 3

20, 10, 30

20, 30, 10



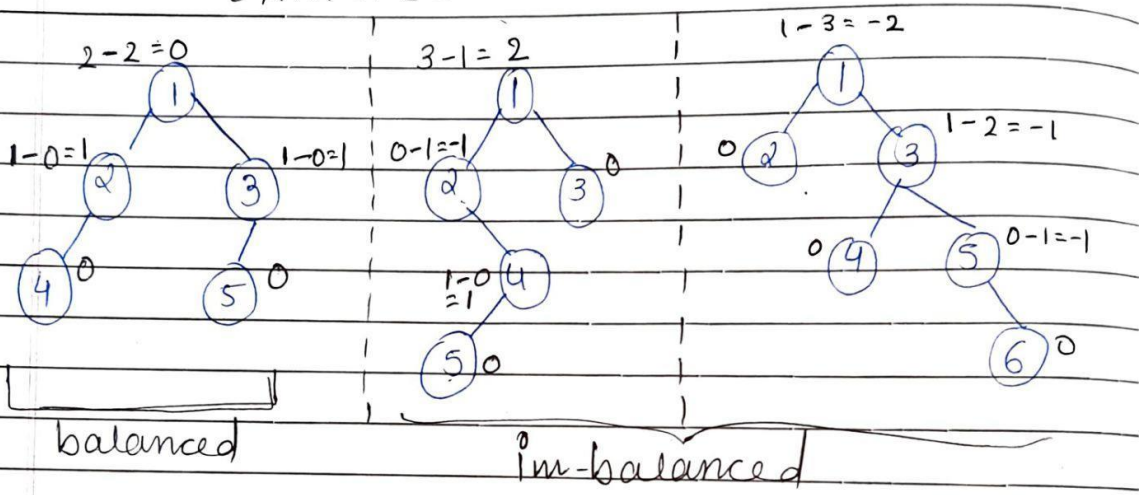
height: 2 (preffered)

Note: Rotations are done only on 3 nodes

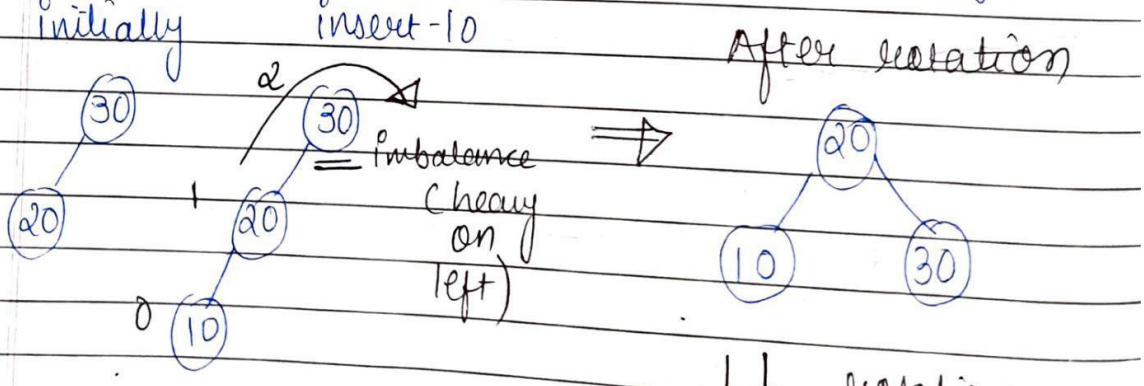
Balance Factor = height of left subtree − height of right subtree

↳ if a node is [balanced]
BF = $h_l - h_r = \{-1, 0, 1\}$

EXAMPLES

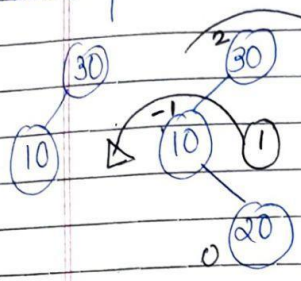


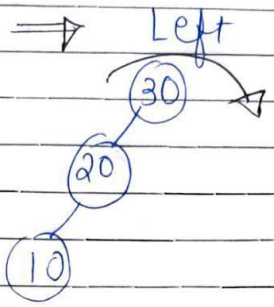
i) LL- imbalance [left of left]: Single rotation at right.



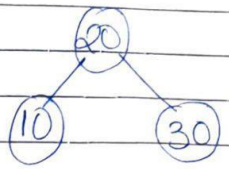
ii) LR - imbalance : double rotation, first Left & then Right

Initially. insert-20



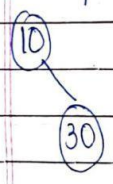


After LR rotation

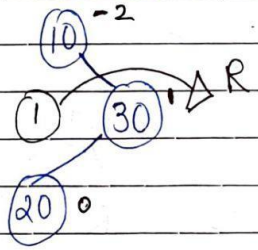


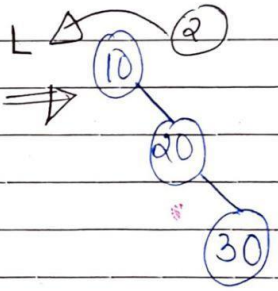
iii) RL - imbalance : double rotation : (1) Right (2) Left

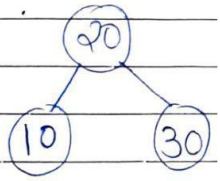
Initially



insert-20



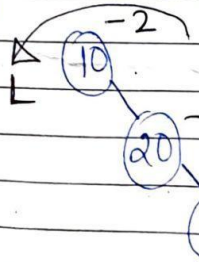




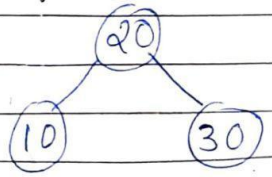
iv) RR - imbalance : single rotation at left

Initially

insert-30



heavy on right

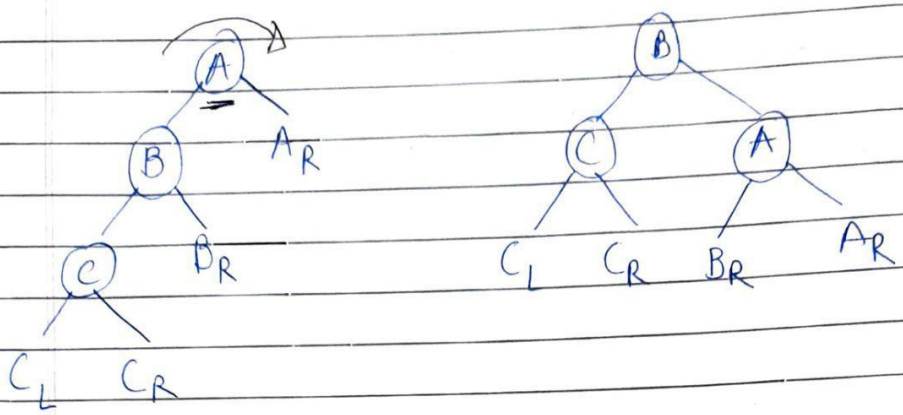


After rotation

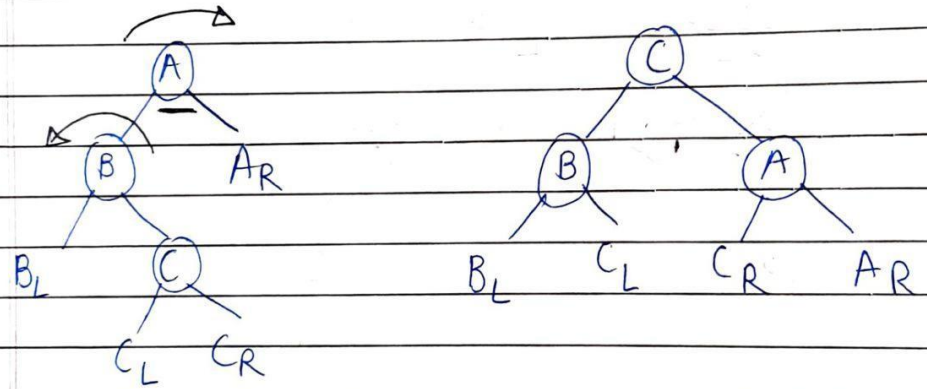
RR rotation

Special Cases

LL - Rotation

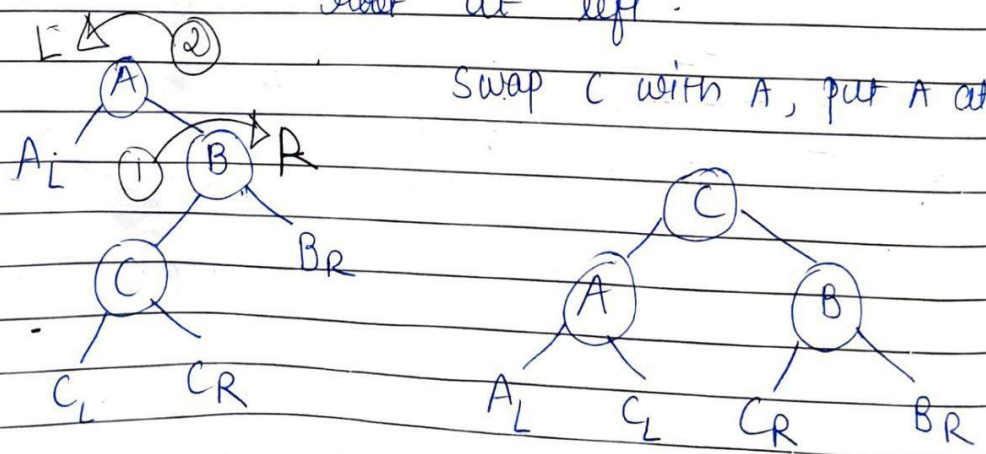


LR - Rotation : Swap with 3rd node Root & move Root at Right

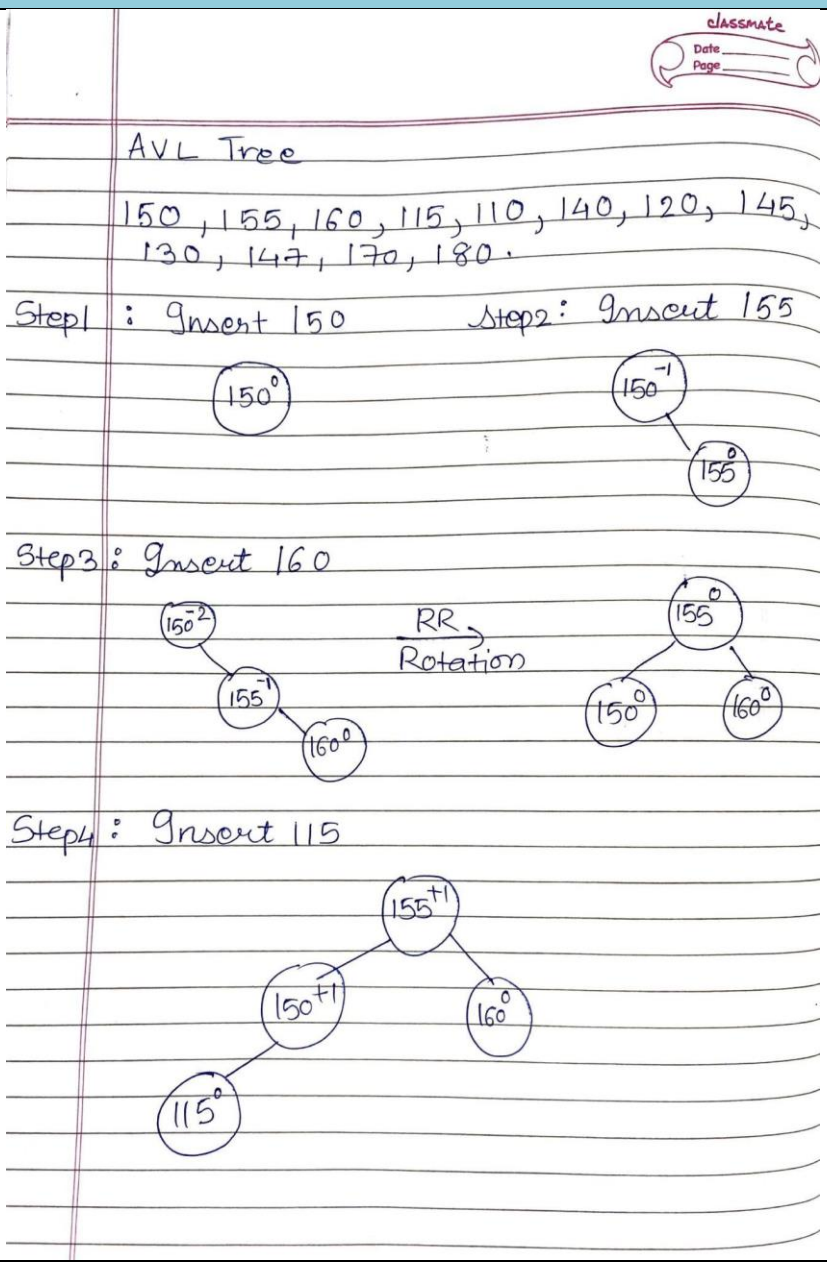


R-L Rotation : Swap 3rd node (last node in imbalance) with Root & move root at left.

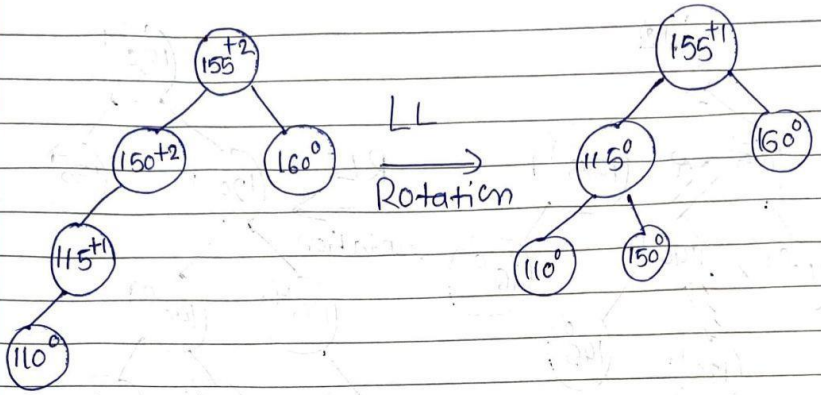
Swap C with A, put A at left



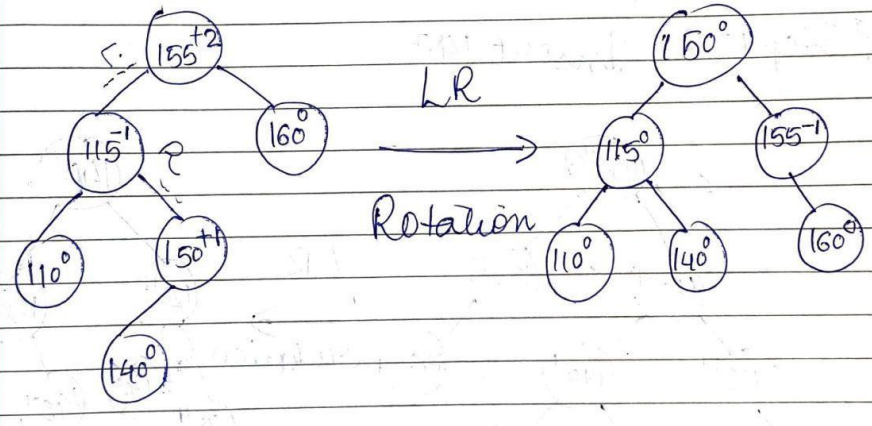
29	Obtain an AVL tree by inserting one integer at a time in the following sequence. 150, 155, 160, 115, 110, 140, 120, 145, 130, 147, 170, 180. Show all the steps.



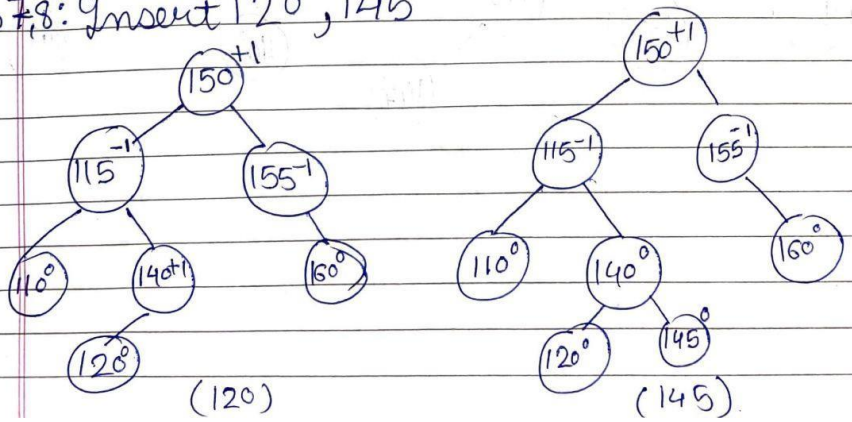
Step 5 Insert 110



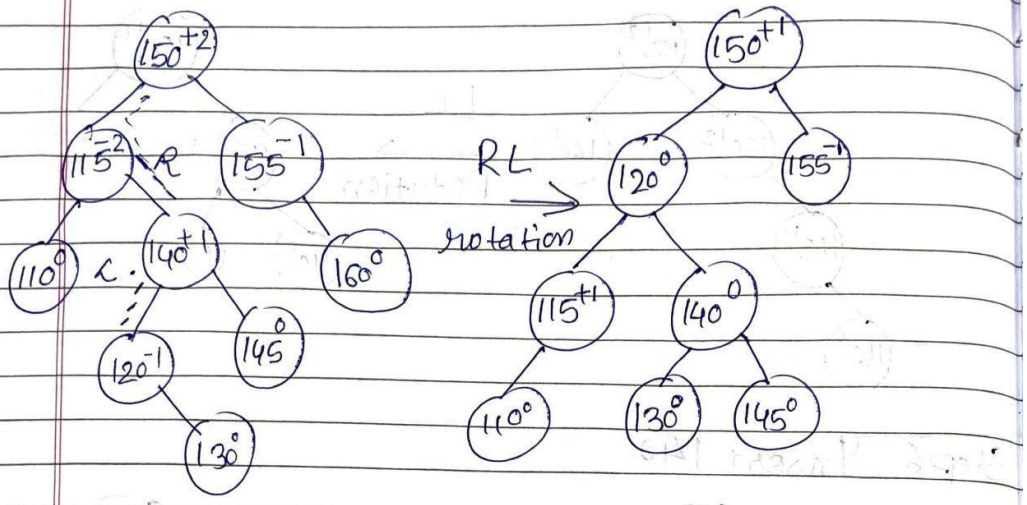
Step 6 Insert 140



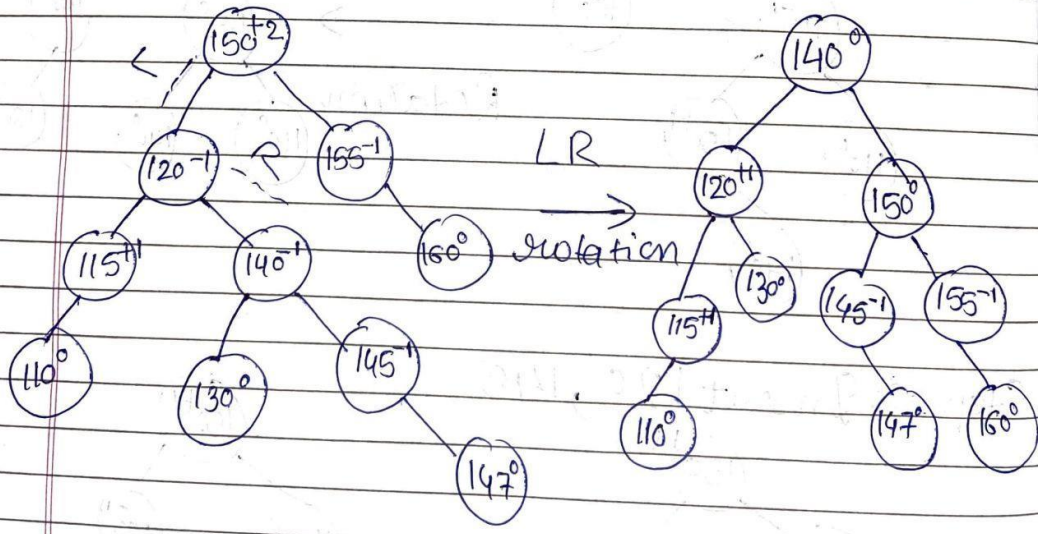
Step 7,8: Insert 120, 145

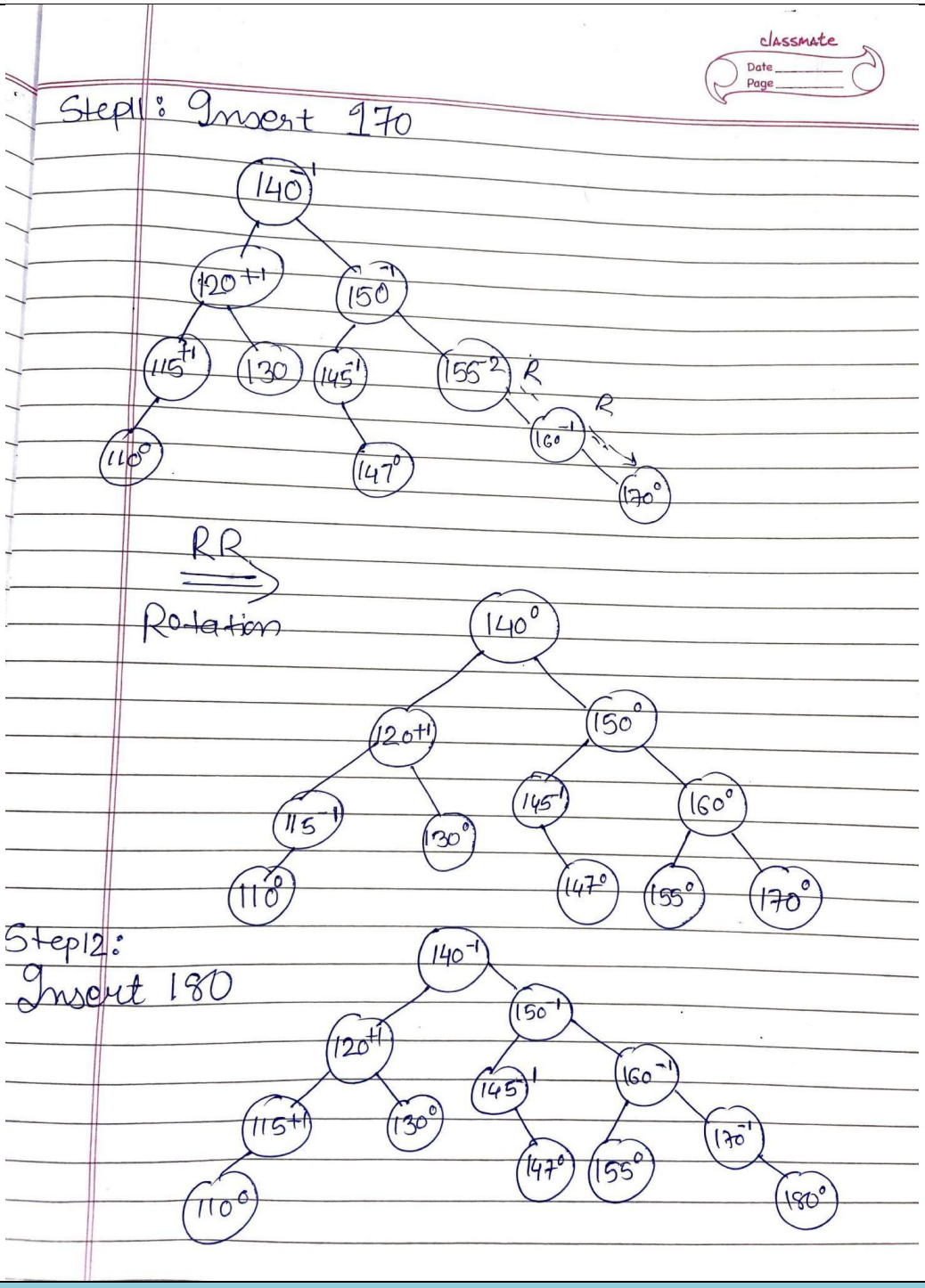


Step 9 Insert 130



Step 10 : Insert 147



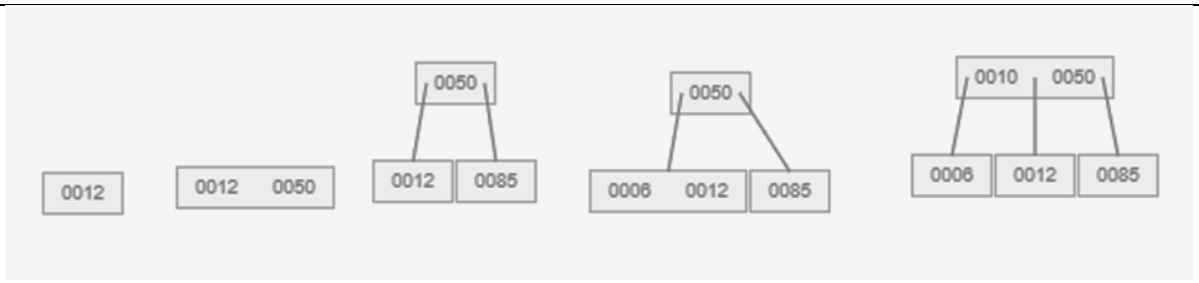


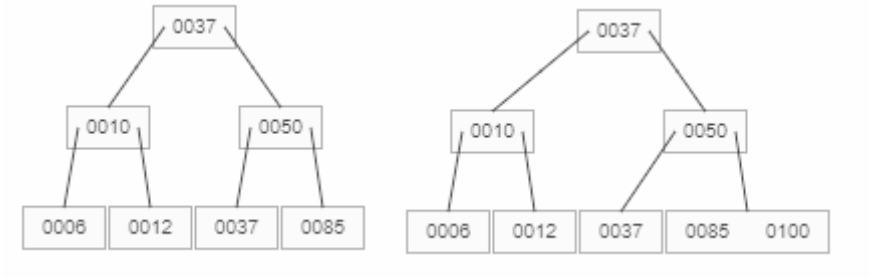
30	What are the advantages of a Multi way search tree over a binary search tree? Construct 2-3 tree for the following data 12, 50, 85, 6, 10, 37, 100, 120, 25, 70

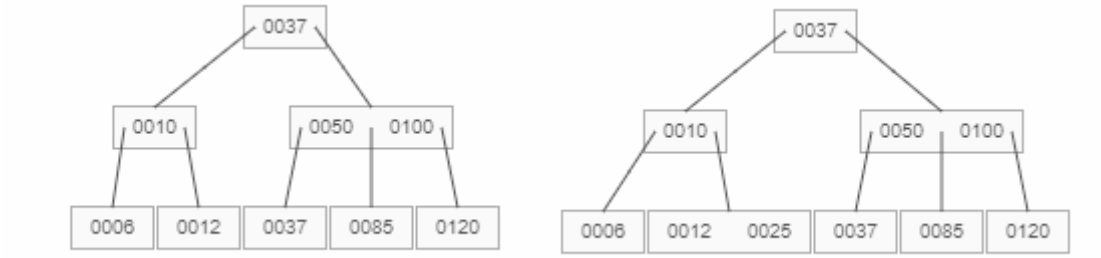
What are Multi-way search trees?

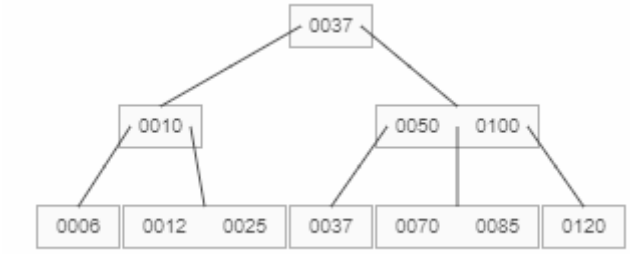
-Multi-way search trees generalize binary search trees into m-ary search trees
-They allow more than one key to be stored in a node
-There will always be one more child pointer than key
-So if a node has two keys, it would have three pointers, with possibility three children
-Increasing the number of children decreases the height of a tree, given the same number of nodes
-The keys in a node are maintained in order
-2 keys, 3 child pointers max for each node

One of the advantages of using these multi-way trees is that they often require fewer internal nodes than binary search trees to store items.









31	Discuss following with reference to graphs. (1) Directed graph (2) Undirected graph (3) Cycle (4)Null graph (5) fringe vertex (6) Hash table (7) Hash function (8) Rehashing (9) Adjacent node (10) Adjacency matrix

(1) Directed graph: A directed graph G, also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G.

(2) Undirected graph: An undirected graph is a set of nodes and a set of links between the nodes. Each node is called a vertex, each link is called an edge, and each edge connects two vertices. The order of the two connected vertices is unimportant. An undirected graph is a finite set of vertices together with a finite set of edges.

(3) Cycle: A path in which the first and the last vertices are the same. A simple cycle has no repeated edges or vertices (except the first and last vertices).

(4) Null graph: A null graph is defined as a graph which consists only the isolated vertices.

(5) Fringe vertex: Vertices adjacent to visited vertices but not yet visited

(6) Hash table: a hash table (hash map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values.

(7) Hash function: A hash function is any function that can be used to map data of arbitrary size to

fixed-size values.

(8) Rehashing: Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table.

(9) Adjacent node: Any two nodes connected by an edge or any two edges connected by a node are said to be adjacent.

(10) Adjacency matrix: An adjacency matrix is used to represent which nodes are adjacent to one another. Two nodes are said to be adjacent if there is an edge connecting them.

32	a) Write an algorithm for Breadth First Search Traversal of a Graph and explain it with an example. b) Write an algorithm for Depth First Search Traversal of a Graph and explain it with an example.

a)

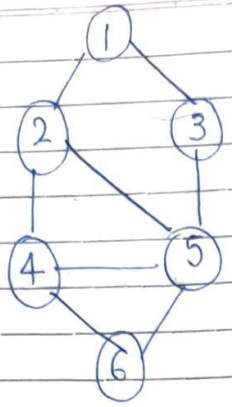
Breadth First Search (BFS)

- BFS is a level-order traversal
- it uses Queue to maintain the visited nodes and the list of adjacent nodes.

Algorithm Steps

1. Take a queue with size equal to the total number of vertices in graph.
2. select any arbitrary vertex, visit the vertex and insert into queue.
3. Visited all unvisited adjacent vertex of a node at front of the queue and insert all of them into queue.
4. if No new adjacent vertex to be visited for vertex at front, then delete the vertex.
5. Repeat step 3 & 4 - until queue becomes empty
6. Final answer will be a spanning tree.

Ex:



Step: 1

Nodes	1	2	3	4	5	6
visited	1	1	1	0	0	0
steps	1	2	3			

Q: | 1 | 2 | 3 | | | |

F R

delete 1

Step: 2

	1	2	3	4	5	6
	~~1~~	1	1	1	1	0
Steps	1	2	3	4	5	

delete - 2

Q: ~~1~~ | ~~2~~ | 3 | 4 | 5 |

F R

Step: 3

1	2	3	4	5	6
1	1	1	1	1	0

delete - 3

~~1~~ | ~~2~~ | ~~3~~ | 4 | 5

F R

Step: 4

1	2	3	4	5	6
1	1	1	1	1	1

~~1~~ | ~~2~~ | ~~3~~ | ~~4~~ | 5 | 6

delete 4

Step: 5 delete 5 and 6 respectively

PRINT : 1, 2, 3, 4, 5, 6

b)

Traversal in graph

- it is a process to visit each node in graph
- it is also used to decide the order of vertices visited in search process.

- Techniques: a) depth first search (DFS)
  b) breath first search (BFS)

→ Depth first Search (DFS)

- The algo starts with some arbitrary node as the root node.
- Here, we need to explore the nodes in depth of each branch - as down as possible.

- DFS uses stack for traversal
- As a final result, it generates spanning tree.

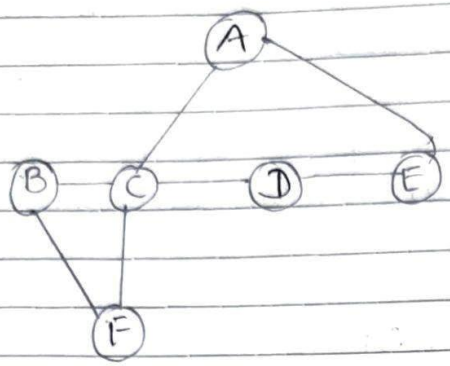
⇒ Algorithm

1. Select an arbitrary vertex.
2. Visit the unvisited vertex and push it into stack.
3. Visit any unvisited adjacent vertex of a node at top of the stack & push it into stack
4. Repeat step-3 until all adjacent vertices are visited.
5. When all adjacent vertices are visited for node at top of the stack, then pop the value from stack (Back-tracking)
6. Repeat steps 3 to 6 until, stack becomes empty.

Example

initial vertex : A

Step: 1



Nodes	A	B	C	D	E	F
	1					

Stack

A

Step: 2

Nodes	A	B	C	D	E	F
	1		(1)			

C
A

Ans: A - C

Step 3

Nodes	A	B	C	D	E	F
	1	(1)	1			

B
C
A

Stack

Ans: A - C - B

Step-4

Nodes	A	B	C	D	E	F
	1	1	1			(1)

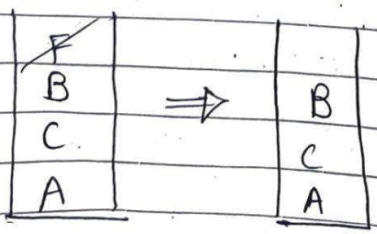
F
B
C
A

Stack

Ans: A - C - B - F

Now from F there is NO unvisited adjacent, SO Backtrack to B and POP F

Step: 5



Step 6 From B, no unvisited adjacent nodes, so backtrack to C. Pop B from stack

B
C
A

⇒

C
A

Step 7 Nodes

A	B	C	D	E	F
1	1	1	(1)		1

D
C
A

Stack

→ unvisited adjacent of C is D

Ans: A-C-B-F-D

Step 8 Nodes

A	B	C	D	E	F
1	1	1	1	(1)	1

E
D
C
A

Stack

Ans: A-C-B-F-D-E

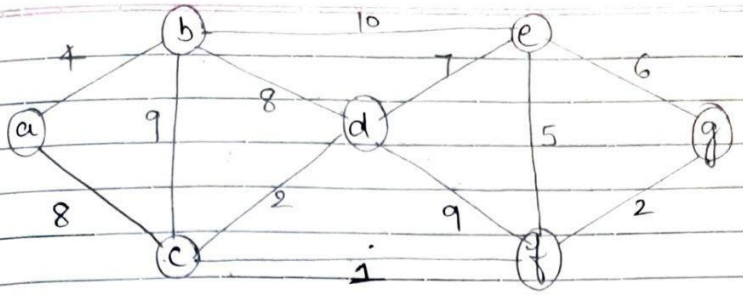
→ Now, All the vertices are visited

33 a) Write an algorithm for Prim's for finding MST and explain it with an example for given graph.
b) Write an algorithm for Kruskal's for finding MST and explain it with an example for given graph.

a)

Prim's Algorithm

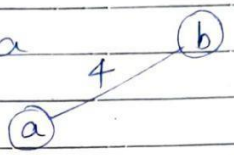
1. Select the pair with minimum weight
2. Select the adjacent vertex and select the minimum weighted edge using the adjacent vertex
   - The selected adjacent vertex should not form the circuit.
3. Repeat step 1 and 2 until all vertices are getting covered



Step: 1 Start with vertex a

a-b: 4 (min)

a-c: 8



weight : 4

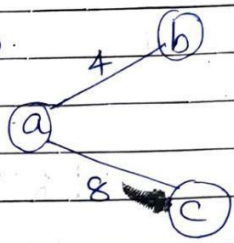
Step: 2 Consider Vertex a & b.

a-c : 8 (min) ✓

b-c : 9

b-d : 8 (min)

b-e : 10



total weight = 8+4 = 12

Step: 3 Consider Vertex a, b and c

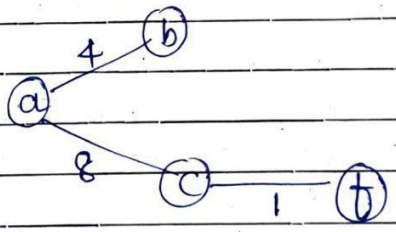
b-d : 8

b-c : 9

b-e : 10

c-d : 2

c-f : 1 (min)



Total weight = 12+1=13

Step: 4 Consider vertex a, b, c and f

b-d : 8

b-c : 9

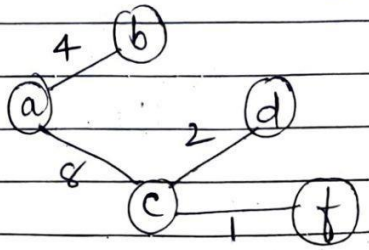
b-e : 10

c-d : 2 (min) ✓

f-d : 9

f-e : 5

f-g : 2 (min)

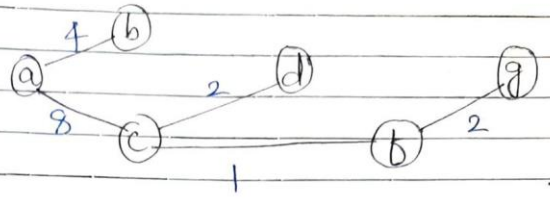


Total weight = 13+2 = 15

Step: 5

Consider vertex a, b, c, f and d

b-d : 8
b-c : 9
b-e : 10
f-d : 9
f-e : 5
f-g : 2 (min)
d-e : 7

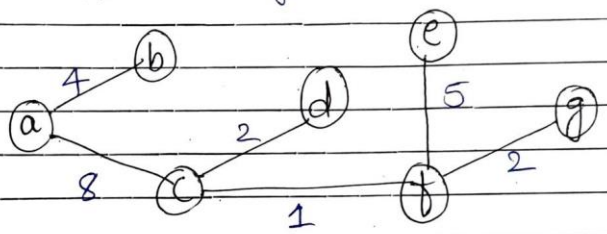


weight : 15 + 2 = 17

Step: 6

Consider vertex a, b, c, f, d and g

b-d : 8
b-c : 9
b-e : 10
f-d : 9
f-e : 5 (min)
d-e : 7
g-e : 6



weight : 17 + 5 = 22

b)

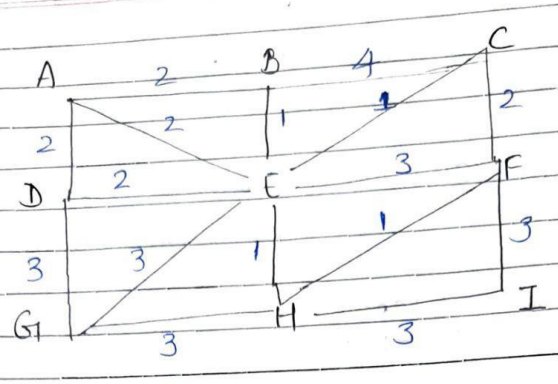
Kruskal's Algorithm

- it is another method for finding the minimum cost spanning tree of the given graph.
- in Kruskal's algorithm, edges are added to the spanning tree in increasing order of cost

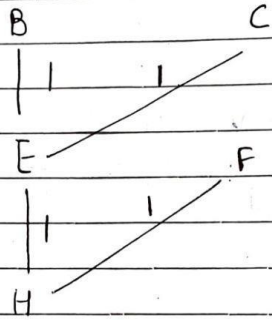
Algorithm Steps :

1. Set $A = \phi$
2. Sort edges E of G_1 into non-decreasing order by their weight w.
3. pick the edge $(u, v) \in G_1$, taken into consideration with minimum weight, if it is not forming the cycle
4. Add edge (u, v) in set A
5. Repeat step 3 and 4 until all the vertices are covered

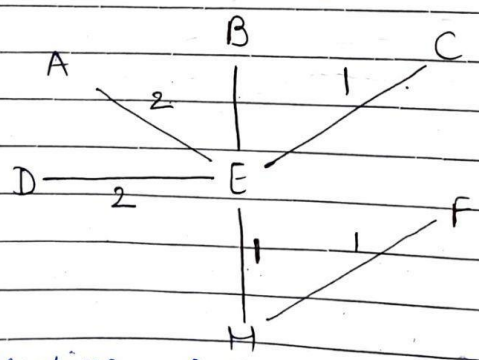
EX:



Step:1 pick edges with minimum weight 1 till the cycle is not formed.



Step:2 Now take edges with weight 2, if they dont form a cycle.



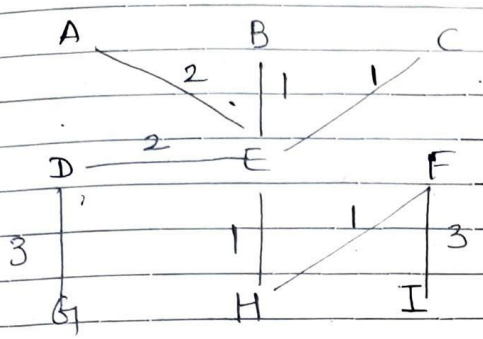
A-B
A-E ✓
D-E ✓
A-D
C-F

After selecting A-E and D-E, None of the edges with weight 2 can be selected as they form a cycle.

Step: 3 Select edges with weight 3 and add them in tree, if they dont form cycle

Select F - I, D - G1 that will complete visiting all vertices

Total weight: 14



34	What do you mean by hashing? What are various hash functions? Explain all methods in brief.

→ Hashing is a technique or process of mapping keys, values in the hash table by using a hash function

→ It is done for faster access to elements

→ Efficiency of mapping depends on the efficiency of hash function used

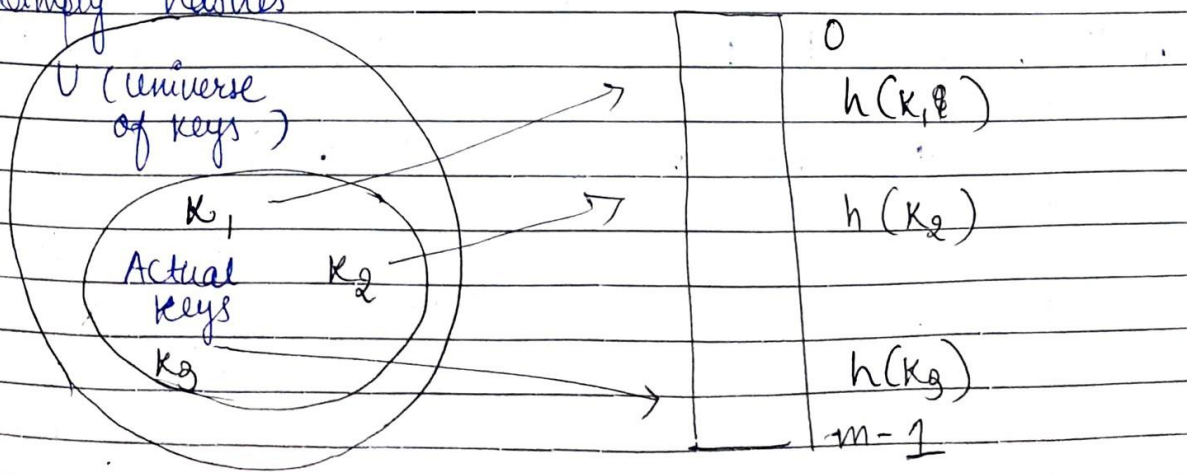
→ Applying Hash function on key (the data to be searched & inserted in hash table), gives the index where the key is stored i.e 24 mod 10 = 4 [24 stored at 4]

Hash Tables: Fixed size array

Hash Function: Apply on keys to get hash values (index of hash tables)

- A hash function is any function that can be used to map a data set of an arbitrary size to a dataset of fixed size, which falls into the hash table.

- The values returned by a hash function are called hash values, hash codes, hash sums or simply hashes



Ex: Hash Table

hash function (Ex $h(key) = value \% n$

Data: 10, 35, 48

N = 10 = Hash table size

$h(10) = 10 \% 10 = 0$ (index)

$h(35) = 35 \% 10 = 5$

$h(48) = 48 \% 10 = 8$

10	0
35	
48	9

Hash functions

1. Division Remainder Method

given Values key = K and slots = N

$h(K) = K \bmod N$

- remainder becomes the index of hash table.

Ex: N = 101, key = 1123456

$h(K) \Rightarrow$ key mod N $\Rightarrow$ 1123456 % 101 = 33

So the key K is stored at index 33

```
      11,123
101 | 1123456
    | 11,23,423
      33
```

K / N = Quotient

N * Quotient = ANS

K − ANS = DIFF ← Remainder

2 Multiplication Method

1 → Key K is multiplied by Constant C,
Where $0 < C < 1$

2 → Fractional part of KC is extracted

3 → Floor of value is multiplied by N and the floor of result is taken as the hash value.

$\lfloor x \rfloor \leq x$

floor of val = largest int.

Function $h(K) = \lfloor N (KC \bmod 1) \rfloor$

(KC mod 1: Real Number mod 1 is fractional part of the whole Number)

EX: K = 132437 N = 101
C = 0.6180339887488

h(132437) = floor (101 * ((132437 * 0.6180339) mod 1))
= floor (101 * (81850.567365098 mod 1))
= floor (101 * 0.567368098)
= floor (57.304175098)
= 57

3. Folded key Method

- its a 2 step process
- 1st step: key is divided into several groups from the left most digit, where each group contains N number of digits, except the last one which may contain lesser number of digits.

[divide the key in parts whose size matches the size of required address]

2nd step:
groups are added together and the hash value is obtained by ignoring the last carry

EX:
if N = 100, then each group is divided in 2 digits and the sum of the groups after ignoring the last carry will also be a 2-digit number between 0 to 99.

- so the hash value for the key = 132437
- first divide the key in 2 digit groups => 13, 24, 37
- Then added the groups together => 13 + 24 + 37 = 74

so the hash value for the K = 132437 is 74

Ex: 6578459

$65+78+45+9 = 197$ (Table index ⇒ 99 max) Max 2 digits can be there

Ans: 97 (ignoring carry 1)

4. Mid Square Method

1. Square the key ⇒ K^2
2. Some of the digits from left and right end of K^2 are removed
3. Whatever the number you get after removing the digit becomes the hash value.

Ex: N = 1000 Key = 132437

$(132437)^2 = 17539558969$

- remove some digits from left and right, the address is 955.

NOTE: Extract the middle digits where they match the length of Max index.
i.e Address = 3 digits then Extract 3 middle digits.

35	What is a collision? Explain two broad classes of collision resolution techniques.

COLLISION RESOLUTION TECHNIQUES

- Collision is handled by collision resolution techniques
  - Chaining
  - Open Addressing
    - Quadratic probing
    - Double Hashing

(1) Chaining

- additional field with data i.e chain
- a seperate chain table is maintained for colliding data
- when collision occurs then a linked list (chain) is maintained at the home bucket

eg Consider keys to be placed in their home buckets
131, 3, 4, 21, 61, 24, 7, 97, 8, 9

Hash function
H(key) = key % D

Index	Key	
0	NULL	29 ← move back to 0
1	131	
2	NULL	21
3	NULL	31
4	4	
5	NULL	5
6	NULL	61
7	7	
8	8	
9	NULL	9
		29

Collision at 21 as $21 \% 10 = 1$
So next empty location is selected

Problem: Primary clustering — process in which a block of data is formed in the hash table when collision is resolved.

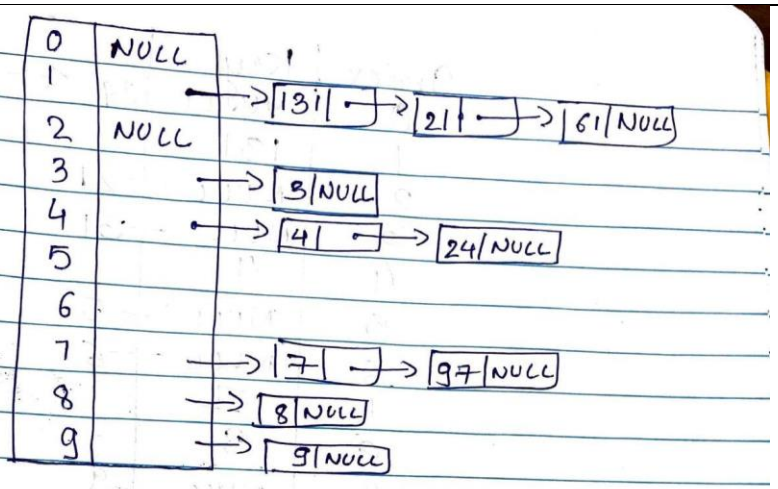
For example

$19 \% 10 = 9$
$18 \% 10 = 8$
$39 \% 10 = 9$
$29 \% 10 = 9$
$8 \% 10 = 8$

0	39
1	29
2	8
3	
4	
5	
6	
7	
8	18
9	19

cluster formed

rest of table is empty



(2) Open addressing linear probing.

- Here collision is solved by placing the second record linearly down whenever the empty bucket is found.
- In this method, hash table is represented as a 1D array with indices that range from 0 to size - 1.
- Initialize to empty slots to detect overflows and collisions

Ex 131, 4, 8, 7, 21, 5, 31, 61, 9, 29

Using division hash function placing values

Double Hashing.

→ Here second hash function is applied to the key when a collision occurs. By applying second hash function we will get the number of positions from the point of collision to insert.

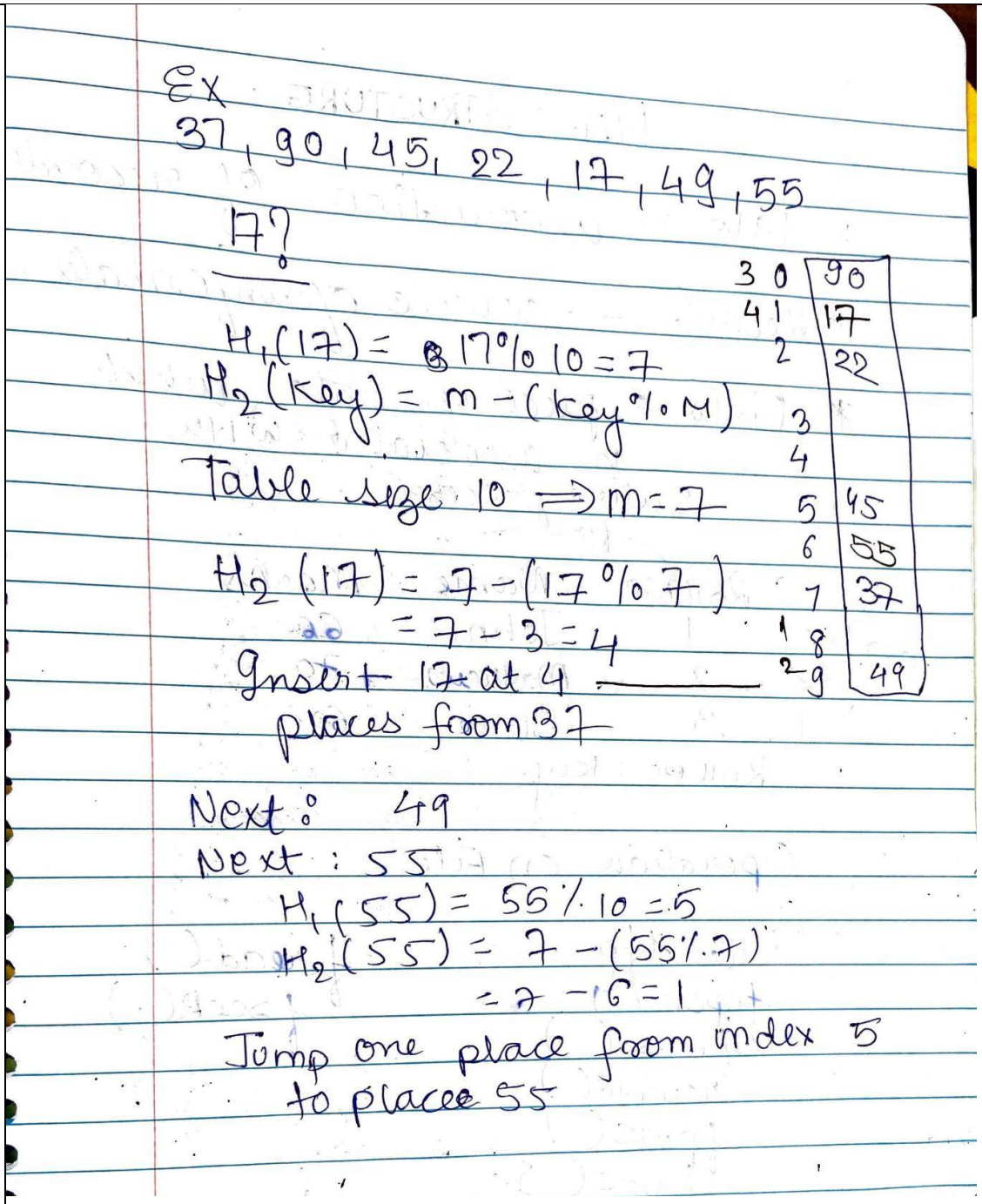
2 rules
- never evaluate to zero.
- all cells can be probed.

The formula to be used

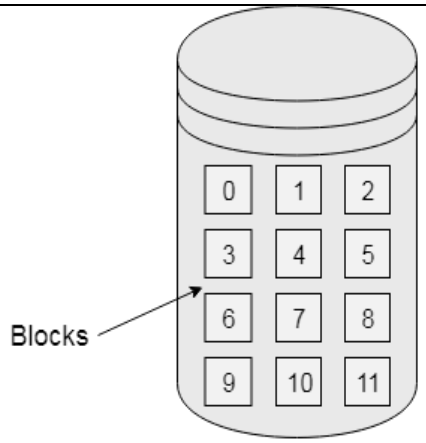
$H_1(key) = key \bmod tablesize$

$H_2(key) = M - (key \bmod M)$

M is a prime no. smaller than the size of the table



36	State different File Organizations and discuss the advantages and disadvantages of each of them.
	The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods. • Contiguous Allocation • Linked Allocation • Indexed Allocation The main idea behind these methods is to provide: • Efficient disk space utilization. • Fast access to the file blocks. If the blocks are allocated to the file in such a way that all the logical blocks of the file get the contiguous physical block in the hard disk then such allocation scheme is known as contiguous allocation. In the image shown below, there are three files in the directory. The starting block and the length of each file are mentioned in the table. We can check in the table that the contiguous blocks are assigned to each file as per its need.



Hard Disk

File Name	Start	Length	Allocated Blocks
abc.text	0	3	0,1,2
video.mp4	4	2	4,5
jtp.docx	9	3	9,10,11

Directory

Contiguous Allocation

Advantages

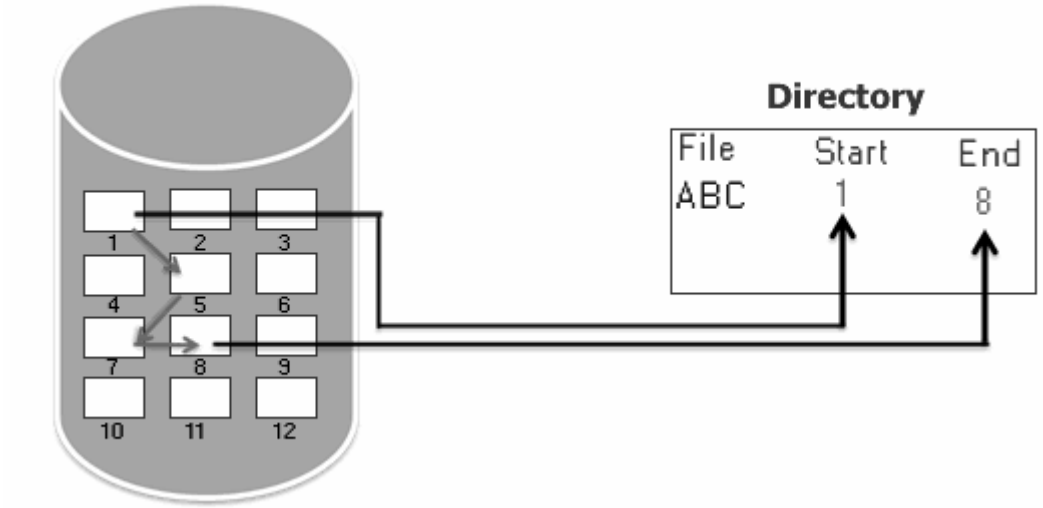
1. It is simple to implement.
2. We will get Excellent read performance.
3. Supports Random Access into files.

Disadvantages

1. The disk will become fragmented.
2. It may be difficult to have a file grow.

Linked List Allocation

Linked List allocation solves all problems of contiguous allocation. In linked list allocation, each file is considered as the linked list of disk blocks. However, the disks blocks allocated to a particular file need not be contiguous on the disk. Each disk block allocated to a file contains a pointer which points to the next disk block allocated to the same file.



Advantages

1. There is no external fragmentation with linked allocation.
2. Any free block can be utilized in order to satisfy the file block requests.

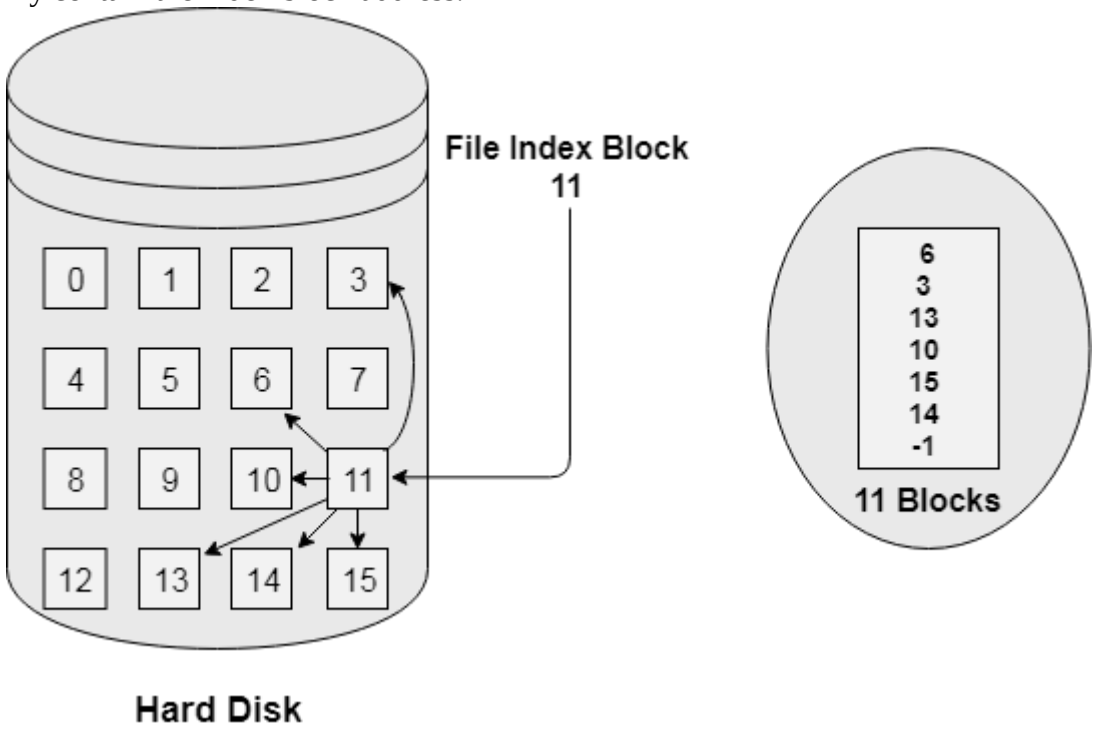
3. File can continue to grow as long as the free blocks are available.
4. Directory entry will only contain the starting block address.

Disadvantages

1. Random Access is not provided.
2. Pointers require some space in the disk blocks.
3. Any of the pointers in the linked list must not be broken otherwise the file will get corrupted.
4. Need to traverse each block.

Indexed Allocation Scheme

Instead of maintaining a file allocation table of all the disk pointers, Indexed allocation scheme stores all the disk pointers in one of the blocks called as indexed block. Indexed block doesn't hold the file data, but it holds the pointers to all the disk blocks allocated to that particular file. Directory entry will only contain the index block address.



Advantages

1. Supports direct access
2. A bad data block causes the lost of only that block.

Disadvantages

1. A bad index block could cause the lost of entire file.
2. Size of a file depends upon the number of pointers, a index block can hold.
3. Having an index block for a small file is totally wastage.
4. More pointer overhead