Date: _____ Page no: ___

DATA STRUCTURES AND ALGORITHMS

⇒ **Data Structures :**
Arrangement of data so that they can be used efficiently in memory. Array, Linked List, Stack, Queue, Graph, BST...

⇒ **Algorithms :**
Sequence of steps on data using efficient data structures to solve a given problem.
Eg. Sorting an array.

OTHER TERMINOLOGY

⇒ **Database :** Collection of information in permanent storage for faster retrival and updation. (HDD)

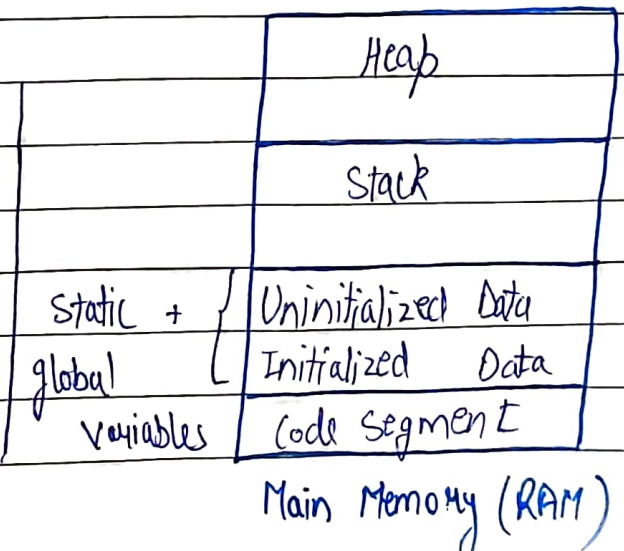
⇒ **Data Warehousing :** Management of huge amount of legacy data for better analysis.

⇒ **Big Data :** Analysis of ~~two~~ too large or complex data which cannot be deal with traditional data processing application.

⇒ **Memory Layout**

① Stack holds the memory occupied by the functions.

② Heap contains the data which is requested by the program as dynamic memory.

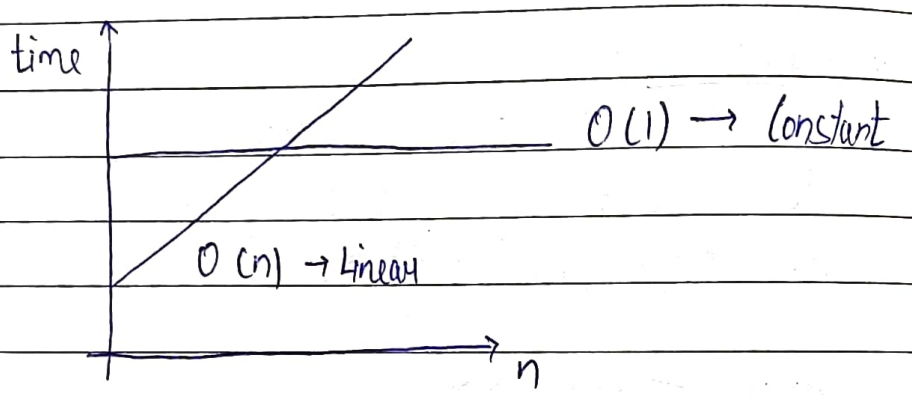


Date: _____ Page no: _____

① Time Complexity & Big O Notations

Time Complexity: Is the study of efficiency of algorithms.

★ How time taken to execute an algorithm grows with the size of the input.



Asymptotic Notations (Big O, Ω, Θ)

(1) Big Oh → O

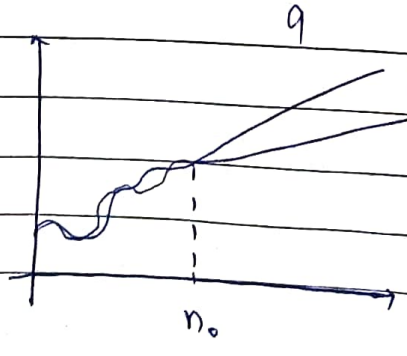
A function $f(n)$ is said to be $O(g_n)$ if there exists constant c and a constant n_0 such that

$$0 \leq f(n) \leq c\, g(n) \quad \forall \quad n \geq n_0$$

Eg. $f(n) = n^3 + 1$

$g(n) = n^3, n^4 \ldots n^{100}$

$0 \leq n^3+1 \leq 2n^3$

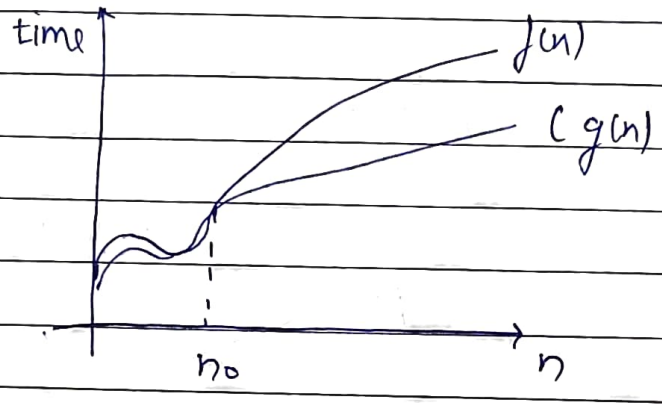


Date:_____ Page no:___

(2) Big Omega → Ω

A function $f(n)$ is said to be $\Omega\, g(n)$ if there exists a positive constant C and n_0 such that

$$0 \leq C g(n) \leq f(n) \qquad \forall\ n \geq n_0$$



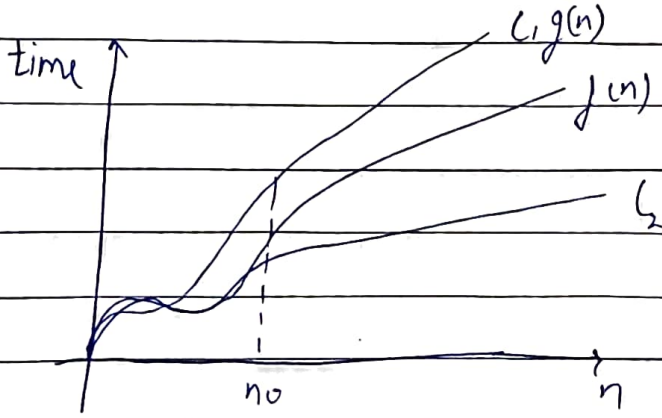
★(3) Big theta → Θ

A function $f(n)$ is said to be $\Theta(gn)$ iff $f(n)$ is $O(gn)$ and $\Omega(gn)$.

Mathematically,

$$0 \leq f(n) \leq C_1 g(n) \qquad \forall\ n \geq n_0$$
$$0 \leq f_2 g(n) \leq f(n)$$

$$0 \leq C_2 g(n) \leq f(n) \leq C_1 g(n) \qquad \forall\ n \geq n_0$$



Date:_____ Page no:___

$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n^n$

↓ Best

Worst ↓

(2) Abstract Data Types

ADTs are the way of classifying data structures by providing a minimal expected interface and set of methods

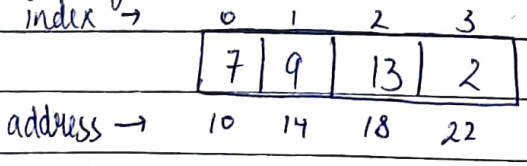
ADT → Minimal Required Functionality
ADT → Operations

⊛ Arrays

An array is a collection of items stored at contigous memory locations. (indices)

Operations :- Max(), Min(), search(num), Insert(i, num) Append(n).

Static Arrays : Size cannot be changed (Fixed Size)
Dynamic Arrays : Size can be changed.



⇒ Arr[4]

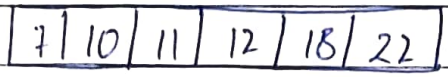
* Elements in an array can be accessed using the base address in constant time O(1).

Date: _____ Page no: ____

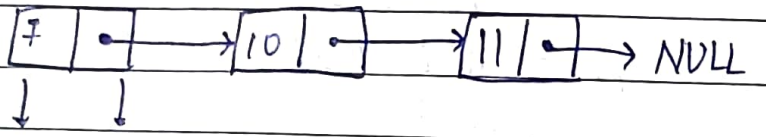
(3) LINKED LISTS

(SINGLY L.L)

Linked List are similar to arrays (Linear Data structures)



⇒ In arrays elements are stored in contigous memory locations.



Data Pointer to next element

⇒ In linked list, elements are stored in non contigous memory locations.

⇒ Why Linked list ?

Memory and the capacity of an array remains fixed. In case of linked list, we can keep adding and removing elements without any capacity constraints.

⇒ Drawbacks of Linked Lists

→ Extra memory space for pointers is required (for every node 1 pointer is added).

→ Random Access not allowed as elements are not stored in contigous memory locations.

```
class Node { public:
    int data;
    Node* next;
}
```

→ Self referencing structure

It can be declared by structs, class, vectors..

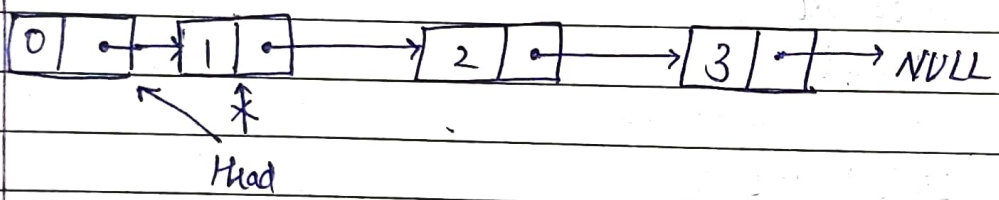
3 types :- Singly, Doubly, circular.

Date: _____ Page no: ___

Insertion of a node

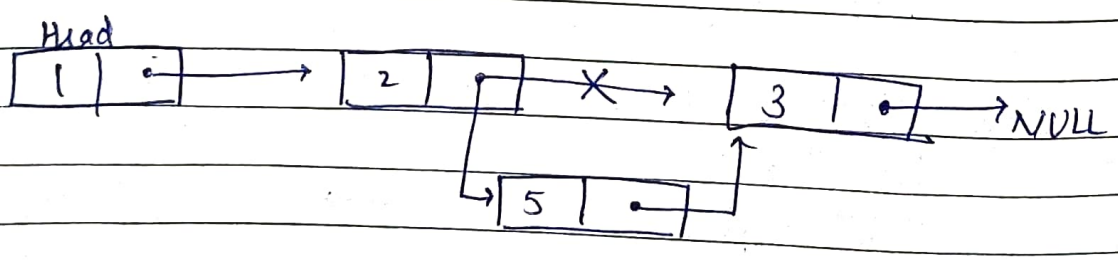
Case 1: Insert at the beginning

Make a new node :
```
Node* ptr = new Node();
ptr → next = head;
head = ptr;
```
} In this sequence only



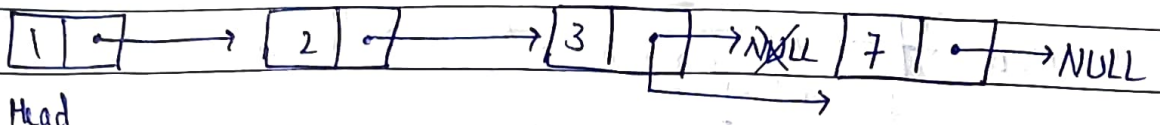
Case 2: Insert at a given node

```
void in_ (Node* prev_node, int data) {
    Node* ptr = new Node();
    ptr → data = data;
    ptr → next = prev_node → next;
    prev_node → next = ptr;
}
```



Date:_____ Page no:___

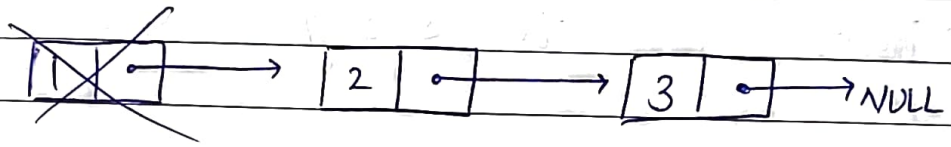
Case 3: Insert at last




```
last_node -> next = new_node;
new_node -> next = NULL;
```

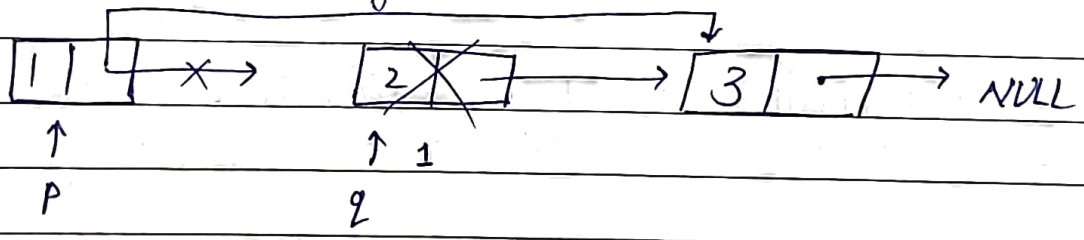
⊛ Deletion of a node (Same cases as linked list (insertion)).

Case 1:

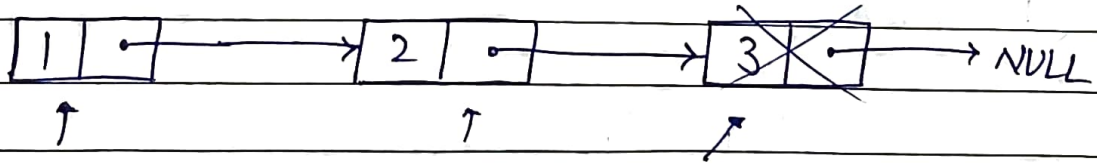



```
Node * ptr = head;
head = head -> next;
free(ptr);
```

Case 2: Deletion at a given index.

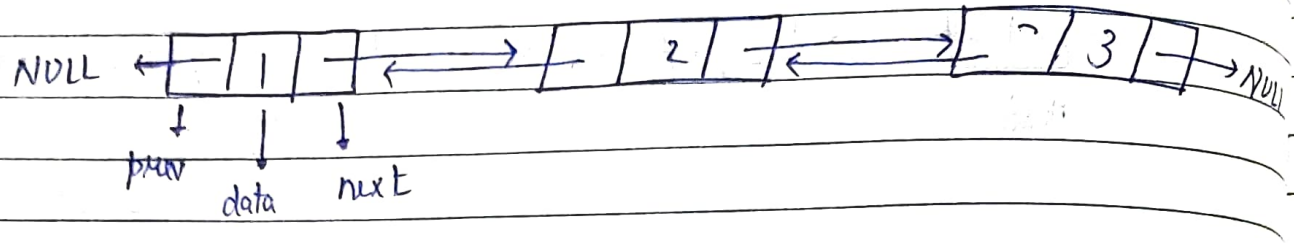


Case 3: Deletion at last




```
Node *p = head;   Node* q = head -> next;   if (q == NULL)   p -> next = NULL;
```

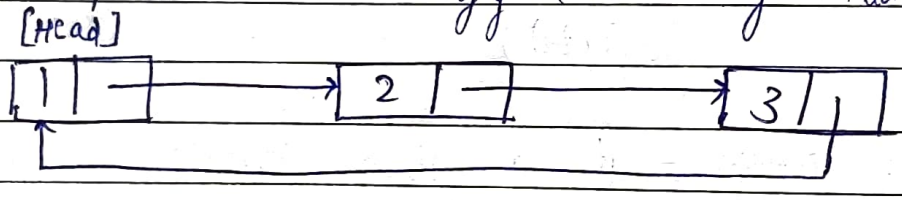
ii. DOUBLY LINKED LIST



→ Can be traversed both in forward as well as backward dirⁿ.
→ Insertion & Deletion is easy than singly Linked List.

iii. CIRCULAR LINKED LIST

No NULL, can be a singly or a doubly linked list.



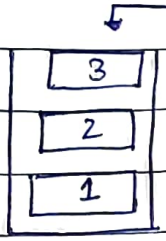
→ Any node can be a starting point (head).
→ Useful for many data structures implementations.

(4) STACK

It is a linear data structure which follows particular order in which the operations are performed.
The order may be LIFO (Last in First out) or FILO (First in last out).

push(), pop(), isEmpty(), isFull(), peek() all take O(1) time.

Date: _____ Page no: ___



Operations (Insertion / deletion) can only performed on this end.

LIFO or FILO

Applications :

1. Used in function calls
2. Infix to postfix conversion (and other similar conversions)
3. Paranthesis matching and more.

In order to create a stack we need a pointer to the topmost elements which are stored inside the stack.

Some of the operations :-

push(), pop(), peek(index), isEmpty(), isFull().

Implemented using an array or linked List.

STACK IMPLEMENTED USING ARRAYS

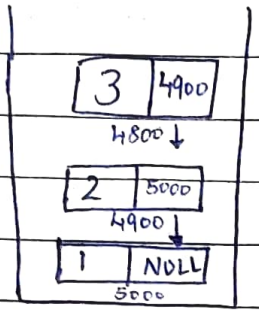
Pros → Easy to implement. Memory saved as pointers are not involved.

Cons → It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

STACK IMPLEMENTED USING LINKED LISTS

Pros → The LL implemented of stack can grow and shrink according to the needs at runtime.

Cons → Requires extra memory due to involvement of pointers.



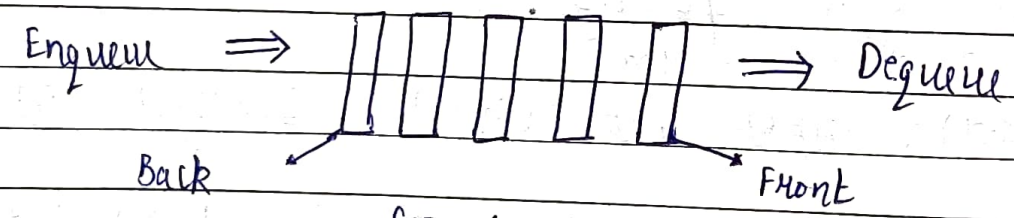
push (1)
push (2)
push (3)

LIFO

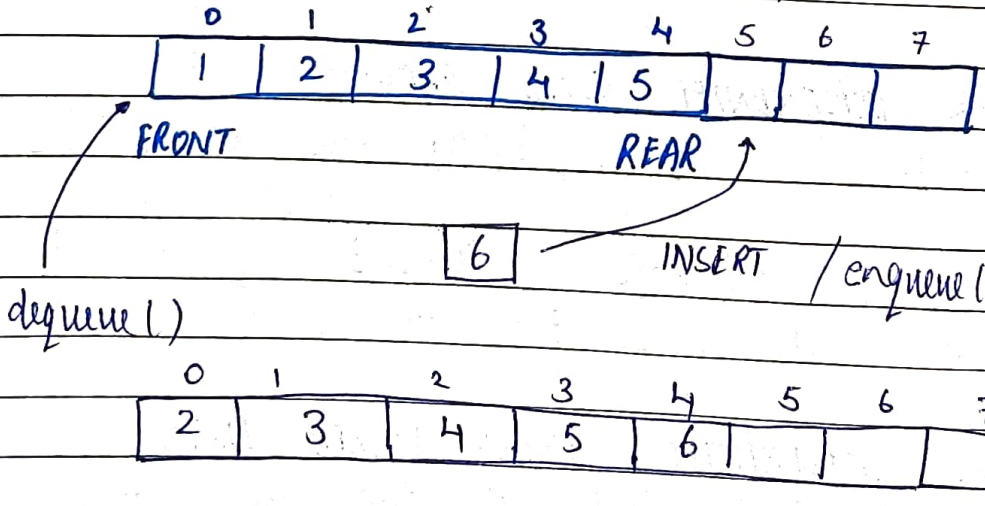
(5) Queue

It is a linear structure like stack,
Order is FIFO (First In First Out)
the pop() / dequeue() is the only difference b/w stack & Queue.

Operations:
Enqueue (+), Dequeue (-), Front (Front item), Back (Back item)



FIFO (First In First Out)



Queue Implementation Using Array.

Date:_____ Page no:___

Drawbacks of queue using array.

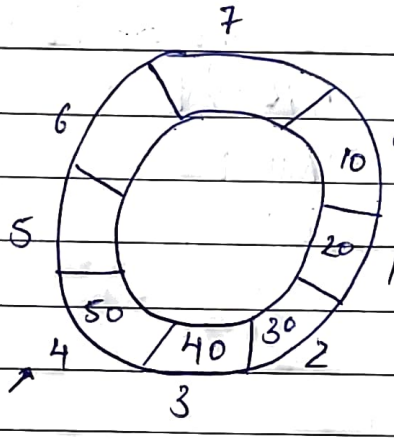
→ Space is not used efficiently.

i. CIRCULAR QUEUE

FIFO

→ Last position is connected back to the first position to make a circle.

'Ring Buffer'.



Front = Rear = -1

if front = 7 Rear = 0

Date:_____ Page no:___

ALGORITHMS

(1) SEARCHING

Brute Force Approach : Linear Search
Divide & Conquer Approach : Binary search

```
int linearSearch ( int arr[], int n, int key) {
    for (int i=0 ; i<n ;i++) {
        if (arr[i] == key) {
            return i;
        }
    else {
        return -1;
        }
}
```

Time Complexity : O(n)
Space Complexity : O(1)

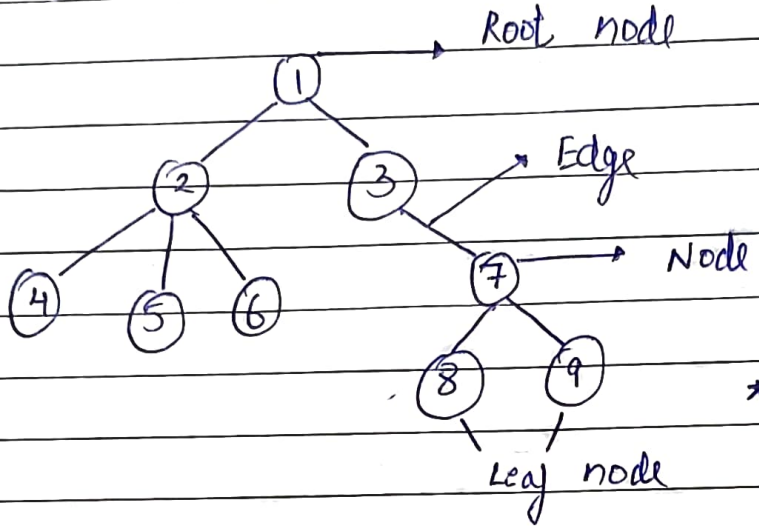
```
int binarySearch ( int arr[], int n, int key) {
        int s = 0;
        int e = n-1;
        while ( s <= e) {
            int mid = (s+e) /2 ;
            if (arr[mid] == key ) {
                return mid;
                }
            else if ( key > arr[mid] ) {
                    s = mid +1 ;
                    }
            else {
                e = mid -1 ;
                }
}
```

Time Complexity: O(log n)
Space Complexity: O(1)

Date:_____ Page no:___

(6) TREE

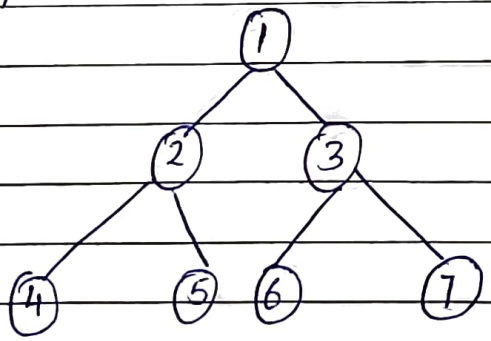
Collection of nodes (starting at a root node) where each node is a data structure consisting of a value



* Data is stored in hierarchical way (levels)

i. BINARY TREE

A tree whose elements have at most 2 children (left & right child).



* PROPERTIES

① Max. nodes at level $L = 2^L$

② Max. nodes in a tree $2^H - 1$ H = Height (3).

③ For N nodes, min. possible height $H = \log_2(N+1)$

④ Tree with L leaves has at least $\log_2(N+1)+1$ no. of levels.

Date: _____ Page no: ___

```
struct Node {
    int data;
    struct Node* left;
    struct Node *Hight;
};
```

(1) Preorder Traversal : 1, 2, 4, 5, 3, 6, 7 — NLR

- Root
- Left subtree
- Right Subtree

(2) Inorder Traversal : 4, 2, 5, 1, 6, 3, 7 — LNR

- Left Subtree
- Root
- Right Subtree

(3) Postorder Traversal : 4, 5, 2, 6, 7, 3, 1 — LRN

- Left subtree
- Right Subtree
- Root

Preorder:

```
if (root == NULL) return;
cout << root -> data << " ";
preorder (root -> left);
preorder (root -> Right);
```

TC : O(h)

h → height of tree.

Inorder:

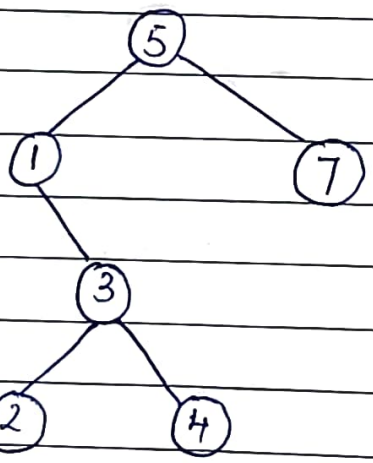
```
if (root == NULL) return;
preorder (root -> left);
cout << root -> data << " ";
preorder (root -> left);
```

Date:_____ Page no:___

BINARY SEARCH TREES

1. Left subtree of a node with keys lesser than the node's key.
2. Right subtree of a node contains only nodes with keys greater than the node's key.
3. There must be no duplicate nodes.

For ex: arr[] = {5, 1, 3, 4, 2, 7}



* Inorder traversal of BST will give sorted array.
{1, 2, 3, 4, 5, 7}

Delete in a BST

Deleting a leaf node (4)

" " node with 1 child (1)

" " " 2 children (3)

TREE — BT, BST, AVL, Red Black, Splay, B, B+.

Date:_____ Page no:___

(7) Graph

Components :-

→ Nodes : Vertex

→ Edges : Links. b/w vertex

- Undirected - Two way edges
- Directed - One way edge.

REPRESENTATION ⇒ ① Adjacency Matrix ② Adjancy List

* 1. Build
* 2. Traverse (BFS & DFS)
* 3. Cyclic or acyclic (BFS & DFS)
* 4. Shortest Path (BFS & DIJKSTRA, FLOYD)
* 5. Topological sorting
* 6. MST (Kruskal, Prim's)
* 7. Backtracking (n-Queen, Rat n Maze, Knight tour)
* 8. ALGOs
  * i) Bellman Ford
  * ii) Floyd warshall
  * iii) TSP
  * iv) Flood fill
  * v) Graph colouring
  * vi) Snake & Ladder