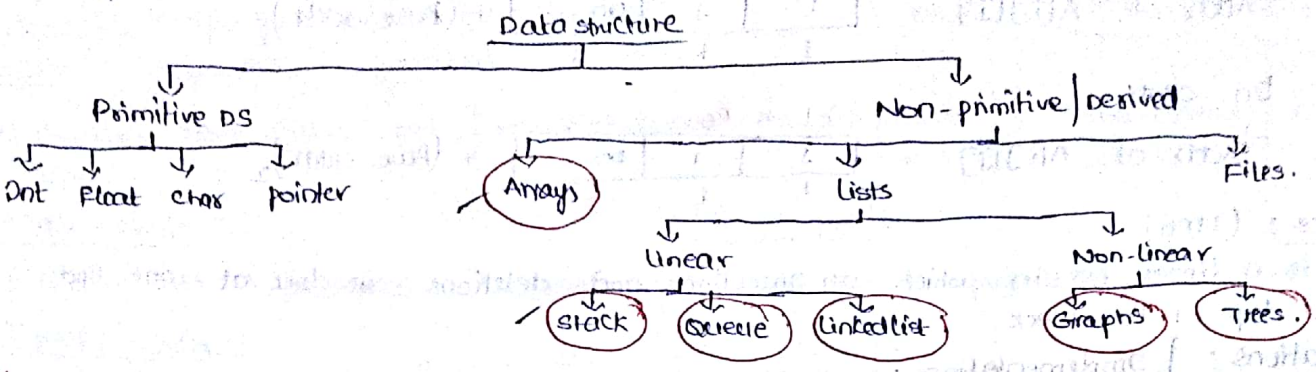
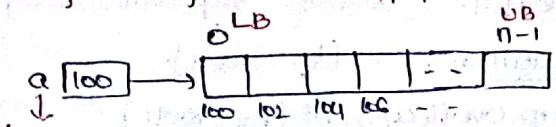
Data structures:

Def: logical or mathematical model of organizing data is called DS.



Arrays:

1D array: only one subscript | index is used to refer all the elements.



*) Size | length of array = UB − LB + 1

*) LOC of A[i] = Base Addr + C * (i − LB)

C ↓ size of datatype

2D Array:

→ Arrays are used for Random Access because of Contiguity

→ 2D arrays are the homogenous collection where the elements are ordered in num of rows & cols.

Memory rep of 2D array:

Row major order: Amxn

Case (i): If index start with '0'

$$Loc(A[i][j]) = Baseaddr + C * (i*n + j)$$

Case (ii): If index start with '1'

$$Loc(A[i][j]) = Baseaddr + C * ((i-1)*n + (j-1))$$

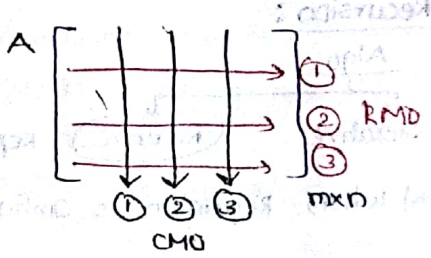
Column major order: Amxn

Case (i): If index start with '0'

$$Loc(A[i][j]) = Baseaddr + C * (j*m + i)$$

Case (ii): If index start with '1'

$$Loc(A[i][j]) = Baseaddr + C * ((j-1)*m + (i-1))$$



∵ RMO, CMO

→ Decrease page faults if proper ordering is used.

Binary Addressing of 2D-Array:

Row major order:

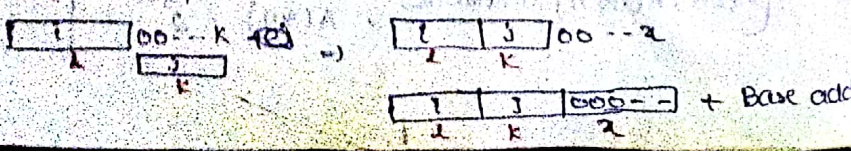
$$Loc(A[i][j]) = Base\ addr + (i*n + j) * C$$

$n \downarrow 2^k$, $C \downarrow 2^x$

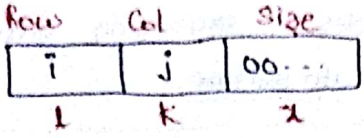
$$A[i][j] = Base\ addr + (i*2^k + j) * C$$

$0 \le i \le m$ | $0 \le j \le n$

$0 \le i \le 2^l - 1$ | $0 \le j \le 2^k - 1$

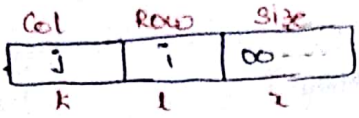
Base addr + 

∴ In RMO

Addr of A[i][j] = 

+ (Base addr)$_2$

In CMO

Addr of A[i][j] = 

+ (Base addr)$_2$

Stacks : (LIFO)

It is a linear DS in which all insertions and deletions are done at one end called top of the stack.

Operations :

1) push - O(1)
2) pop - O(1)

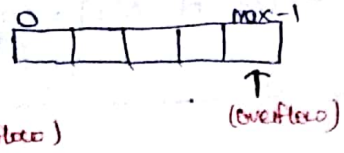
Array — Linked list

Implementation :

① Array :

Overflow : If (top == max-1)

Underflow : If (top == -1)



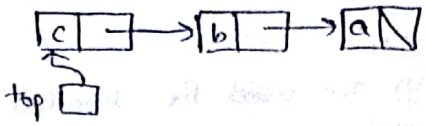
② Linked list :

Overflow : X (but Heap overflow may occur)

Underflow : If (top == NULL)

Heap overflow : If (P == NULL)

Heap running out



Applications :

① Recursion
② Infix, prefix, postfix Conversions.
③ parsing
- matching parenthesis
- matching tags

④ Tree traversals
⑤ Graph traversals.

Recursion :

Algo

Iterative / Recursive : Repetetive process in which Algo calls itself

*) when Recursion is initiated, the info like
- value of parameter
- value of local addr
- return addr
- return value.

is pushed to stack and the child recursion is over the control comes back to parent and the info is popped out from stack.

① Euclidian Algorithm :

$$GCD(x,y) = \begin{cases} GCD(y,x) & \text{if } x<y \\ x & \text{if } y=0 \\ GCD(y, x \bmod y) & \text{otherwise.} \end{cases}$$

② Combination of objects :

$$c(n,k) = \begin{cases} 1 & \text{if } k=0 \mid k=n \\ c(n-1,k) + c(n-1,k-1) & \text{if } n>k>0 \end{cases}$$

③ Ackermann function :

$$A(x,y) = \begin{cases} y+1 & \text{if } x=0 \\ A(x-1,1) & \text{if } y=0 \\ A(x-1, A(x,y-1)) & \text{otherwise} \end{cases}$$

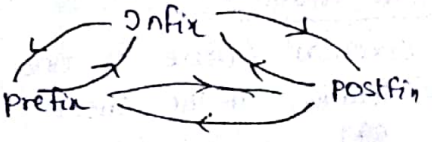
Formulae :
- A(0,y) = y+1
- A(1,y) = y+2
- A(2,y) = 2y+3
- A(3,y) = $2^{y+3} - 3$

④ Towers of Hanoi :

$$M(n) \begin{cases} 2M(n-1)+1, & n>1 \\ 1 & n=1 \end{cases}$$

Sol → $2^n - 1$

② Infix, prefix, postfix Conversions



Infix : <operand> <operator> <operand>
Prefix : <operator> <operand> <operand>
Postfix : <operand> <operand> <operator>

*) Compiler uses either prefix/postfix because the expression is evaluated in one scan.

a) Infix – prefix :
→ Follow rules of precedence & Associativity

b) Infix – postfix :
→ Follow rules of precedence & Associativity.

c) prefix – Infix :
Right to left
→ Read the prefix strn from reverse order
→ If the symbol is operand push to stack.
→ If the symbol is operator pop two elements from stack and place operator in b/w two operands
→ Repeat the same until the end of the Prefix Expression.

d) Postfix – Infix :
→ Read the prefix from left to right
→ If the symbol is operand push to stack
→ If the symbol is operator pop two values from stack and place operator in between operands and push the resulted string back to stack
→ Repeat the same until the end of the postfix Exprenion

e) prefix – postfix Conversion :
→ Read the prefix strn from Right to left
→ If the symbol is operand push to stack.
→ If the symbol is operator pop two elements and place like (op1, op2, operator) and push the result back to stack.
→ Repeat until the end of prefix Expr

f) Postfix – prefix :
→ Read the postfix Expr from left to right
→ If the symbol is operand push to stack.
→ If the symbol is operator pop two elements and place like (operator, op1, op2) and push the result back to stack
→ Repeat until the end of postfix Expr.

Queues : (FIFO)

→ queue is a linear DS in which deletions can take place only in one end called front and insertions can take place at other end called rear.

Implementation :

1) Array :
Initially front = -1
rear = -1
At insertion we increment rear pointer
At deletion we increment front pointer
overflow : (rear == n-1)
underflow : (front == rear)
∴ space wasted if deleted
↓ sol
using circular queue.

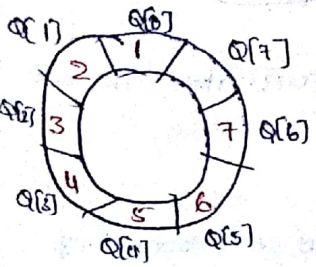
2) Linkedlist :
Initially front = NULL
rear = NULL
At insertion we increment rear pointer
At deletion we increment rear pointer
overflow : (rear == n-1)
underflow : (front == rear)

Infix – post/pre → Operator stack
post/pre – Infix → Operand stack

Operations :
1) Enqueue : O(1)
2) Dequeue : O(1)

Circular Queue :

A circular queue is one in which insertion of a new element is done at very first location of the queue if the last location of the queue is full.



Overflow : (front == (rear + 1) mod n)

Underflow : (front == rear)

Operations :
- Cenqueue : O(1)
- Cdequeue : O(1)

Priority Queue :

To Implement a priority queue 2 linear arrays are read.
- data
- Prority

*) The lower priority info will be proccened first and then higher priority.

Types :

Ascending priority Queue

25	23	12	17	8	32	9
1	1	1	2	2	3	3

Descending priority Queue

7	91	32	64	12	25	16
3	3	2	2	2	1	1

*) priority Queues may be
- strictly Increasing order
- strictly decreasing order
- Non-decreasing order
- Non-Increasing order

Linked List : D | L

→ It consists of nodes which are created dynamically.

∵ size can grow & shrink
Insertion and deletion are easier & efficient

∴ Access of data takes linear time
Need more space than array bcoz of links

*)

operation	Array	LinkedList
Insertion	O(n)	O(1)
deletion	O(n)	O(1)
Access	O(1)	O(n)
search (unsorted)	O(n)	O(n)
search (sorted)	O(logn)	
Contiguous	✓	X

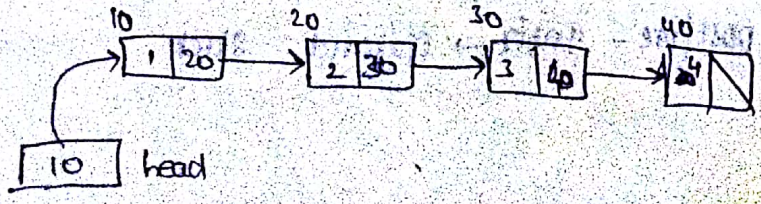
Types :
1) singly LL
2) doubly LL
3) circular LL

Operations
- Insertion : O(1) → Front, middle, End
- deletion : O(1) → Front, middle, end.
- Traversal O(n)

*) creation of a node in a LL uses self referential structures.

*) struct node *new = (struct node *) malloc (size of (struct node))
↳ Allocating memory to the node dynamically

Single linked list :



* while(t) ≅ while(t != NULL)

* Null → link : Segmentation fault

Traversal :

```
struct node *t
t = head
while (t)
{
  pf("%d", t->data);
  t = t->next;
}
```

O(n)

Insertion at beginning

```
struct node *new
new->next = head
head = new
```

O(1)

Insertion at middle

```
struct node *new
struct node *t = head
while (t->i != 2)
  t = t->link
new->link = t->link
t->link = new
```

θ(n)

Insertion at end :

```
Struct node * new, *t = head
while(t)
  t = t->link;
t->link = new
new->link = NULL
```

O(n)

Deletion at front :

```
If (head == NULL)
return ;
Else if (head->link == NULL)
{
free(head)
head = NULL;
}
Else
{
struct node *t = head.
head = head->link.
free(t);
}
```

O(1)

Deletion at middle :

```
If (head == NULL)
return
Else if (head->link == NULL)
{ free(head);
  head = NULL;
}
Else
{
struct node *t1, *t2 = head;
while (t2->i != 2)
  t2 = t2->link.
t1 = t2->link.
t2->link = t1->link.
free(t1);
}
```

θ(n)

Deletion at end :

```
If (head == NULL)
  return
Else if (head->link == NULL)
{ free(head);
  head = NULL;
}
Else
{
struct node *t = head
while (t->link->link != NULL)
  t = t->link;
free(t->link);
t->link = NULL;
```

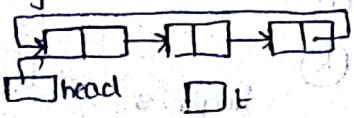
O(n)

Circular linked list :

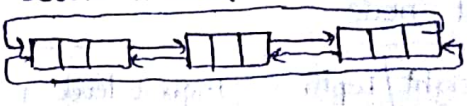
It is a type of linear list in which next pointer field of the last node contains the address of the first node rather than NULL pointer.

Types

- single circular linked list



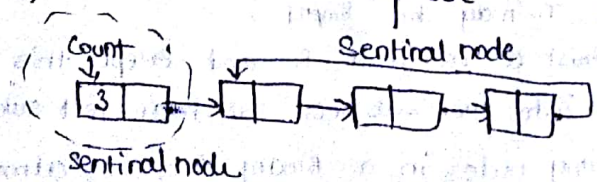
- Double circular linked list



☺ We can traverse back which is not possible in SLL

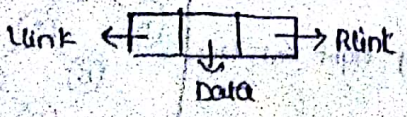
☹ It is difficult to find which node is the head node in CLL

↓ sol



Doubly linked list :

It is a two way list one can move from right to left or from left to right



Creation

```
Struct node
{
  Struct node *Llink;
  int data
  Struct node *Rlink;
};
```

Insertion at beginning :

```
struct node *new;
new → Rlink = head;
head → Llink = new
new → Llink = NULL
head = new;
```

Updated links = (3)

Insertion at middle :

```
struct node * Mnode;
struct node * new = head;
while (new → data != 2)
    new = new → Rlink.
new → Rlink = Mnode
Mnode → Rlink = new → Rlink;
new → Rlink → Llink = Mnode;
Mnode → Llink = new;
```

Updated links = (4)

Insertion at end

```
struct node *new;
struct node *t = head;
while (t → Rlink)
    t = t → Rlink.
t → Rlink = new
new → Llink = t
new → Rlink = NULL
```

Updated links = 3

Deletion at beginning:

```
struct node *t = head;
head = head → Rlink;
head → Llink = NULL
free(t);
```

Updated links = (1)

Deletion at middle :

```
struct node *t = head
while (t → data != 2)
    t = t → Rlink;
t → Llink → Rlink = t → Rlink.
t → Rlink → Llink = t → Llink
free(t);
```

updated links = (2)

Deletion at End :

```
struct node *t = head;
while (t → Rlink)
    t = t → Rlink
t → Llink → Rlink = NULL
free(t);
```

Updated links = (1)

Trees : (Acyclic)

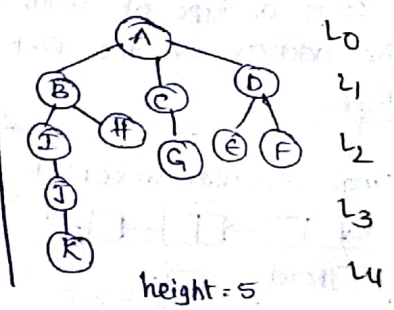
A Tree is a non-linear datastructure where data items are arranged in hierarchial manner

Degree of a node : Num of child nodes for a given node

Degree of tree : max (Degree of all nodes).

*) Root node

*) Height / Depth = Highest level + 1



Binary Trees :

Def : Nodes are arranged in such a way that

(i) The tree T may be Empty

(ii) There Exist a root node R and Except this root node 'R' all other nodes are divided into two subtrees called as left subtree and right subtree.

*) Degree of any node in a Binary tree is atmost 2

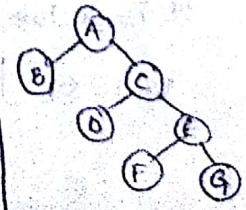
Types of Binary trees :

① **strict Binary tree :**

→ Each Internal node has non Empty left subtree and non-Empty right sub tree

*) If 'n' number of terminal nodes are present then

Num of nodes = $2n - 1$

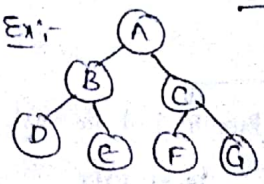


Full Binary tree :

→ Each level 'l' has 2^l num of nodes.

*) If highest level is 'l' then num of nodes = $2^0 + 2^1 + \cdots + 2^l$

Ex:-



0	$2^0 = 1$
1	$2^1 = 2$
2	$2^2 = 4$

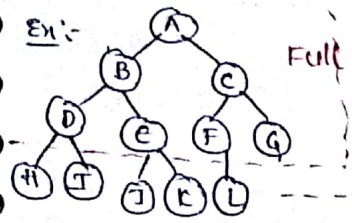
Num of nodes = $2^{l+1} - 1$

= $2^h - 1$

Complete Binary tree :

→ A Binary tree where (highest level − 1) are fully filled and highest level may or may not be full but it must be left most completed is called complete binary tree.

Ex:-



*)	Full	Complete
Height	$\log_2(N+1)$	$\lfloor \log_2 N \rfloor + 1$
Num of nodes (N)	$2^{l+1} - 1$ (or) $2^h - 1$	

*) Max/Min num of nodes in a binary tree of height 'h'

$N_{min} = h = l+1$

$N_{max} = 2^h - 1 = 2^{l+1} - 1$

*) Num of leaf nodes in a Full binary tree is 2^l

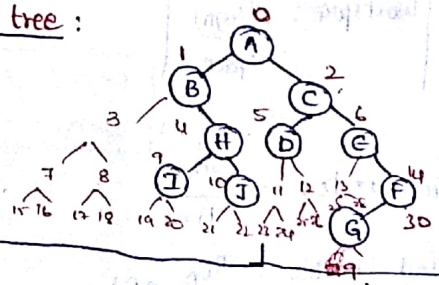
*) Num of non-leaf nodes in a Full binary tree is $2^l - 1$

* An n-ary tree, with x internal nodes will have $x(n-1)+1$ leaf nodes

* Infix - post/pre → Operator Stack

post/pre - Infix → Operand Stack.

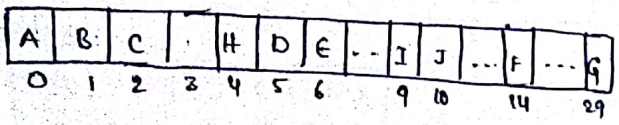
Representation of a tree :



Array representation

→ static rep

Array size = $2^4 - 1 = 2^5 - 1 = 31$



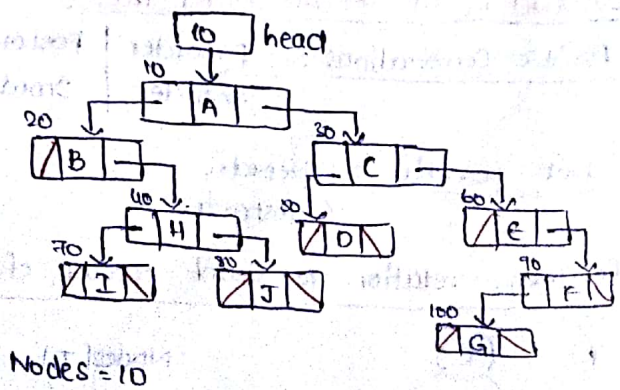
∵ Direct Access

∴ Space Inefficient

Linked list representation

→ Dynamic rep

→ Doubly linked list is used



Nodes = 10

Nulls = 11

∵ Space Effectiveness

∴ Sequential Access —worst case→ O(h)

Applications of trees :

- → Used to implement file system in os
- → Used to Evaluate Arithmetic Expressions
- → making search efficient in O(logn)
- → Tree traversals.

Tree traversals

Depth First traversal

Preorder :
- → visit root
- → Traverse left subtree
- → Traverse Right subtree

Inorder
- → Traverse left subtree
- → visit root
- → Traverse Right subtree

Postorder
- → Traverse left subtree
- → Traverse right subtree
- → visit root

Breadth First Traversal

levelorder
- → Traverse level by level from level 0 to max level.

*)

Preorder	:	Root	L	R
Inorder	:	L	Root	R
Postorder	:	L	R	Root

I st time
II nd time
III rd time

Analysis of traversals :

$$T(n) = 2T(n/2) + C \quad ; n > 1$$
$$= 1 \quad ; n = 1$$

Common for Inorder, Preorder, postorder

Preorder :
- Time : O(n)
- Space : O(n)
- Workspace : O(logn) stack

Inorder
- Time : O(n)
- Space : O(n)
- Workspace : O(logn) stack

Postorder
- Time : O(n)
- Space : O(n)
- Workspace : O(logn) stack.

Max num of Binary trees :

Max num of binary tree with unlabelled nodes = $\frac{2nCn}{n+1}$

Max num of binary trees with labelled nodes = $\frac{2nCn}{n+1} * n!$

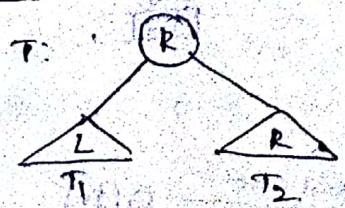
Max num of Binary tree with preorder of particular sequence = $\frac{2nCn}{n+1}$.

Construction of unique Binary trees :

Possible Combinations : preorder → Inorder | Postorder ← Inorder

Not possible : preorder ✗ Postorder

Recurrence relation to Count number of nodes :



$$Nodes(T) = 1 + Nodes(T_1) + Nodes(T_2) \quad ; T \text{ is not null}$$
$$= 0 \quad ; T \text{ is null}$$

Analysis :

$$T(n) = 2T\left(\frac{n}{2}\right) + C \quad ; n>1$$
$$= 1 \quad ; n=1$$

Time : $O(n)$

Space : $O(n)$

Workspace : $O(\log n)$ => Stack

Recursive program to Count num of leaves and non-leaves :

$$\text{leaves}(T) = 1 \quad ; T \text{ is leaf}$$
$$= \text{leaves}(T_1) + \text{leaves}(T_2) \quad : \text{otherwise}$$

Analysis :

$$T(n) = 2T\left(\frac{n}{2}\right) + C \quad ; n>1$$
$$= 1 \quad ; n=1$$

Time : $O(n)$

Space : $O(n)$

Workspace : $O(\log n)$

$$\text{Nonleaves}(T) = 0 \quad ; T \text{ is leaf}$$
$$= 1 + \text{Nonleaves}(T_1) + \text{Nonleaves}(T_2) \quad ; \text{otherwise}$$

Analysis :

$$T(n) = 2T\left(\frac{n}{2}\right) + C \quad ; n>1$$
$$= 0 \quad ; n=1$$

Time : $O(n)$

Space : $O(n)$

Workspace : $O(\log n)$

Recursive program to find full node :

$$FN(T) = 0 \quad ; n=1$$
$$= 1 + FN(T_1) + FN(T_2) \quad ; n>1$$

Analysis :

$$T(n) = 2T\left(\frac{n}{2}\right) + C \quad ; n>1$$
$$= 0 \quad ; n=1$$

Time : $O(n)$

Space : $O(n)$

Workspace : $O(\log n)$

Recursive program to find height of a tree :

$$H(n) = 1 \quad ; n=1$$
$$= 1 + \text{Max}\left(H(T_1), H(T_2)\right) \quad ; n>1$$

Analysis :

$$T(n) = 2T\left(\frac{n}{2}\right) + C \quad ; n>1$$
$$= 1 \quad ; n=1$$

Time : $O(n)$

Space : $O(n)$

Workspace : $O(\log n)$

*) Convene preorder : Root R L

Convene inorder : R Root L

Convene postorder : R L Root

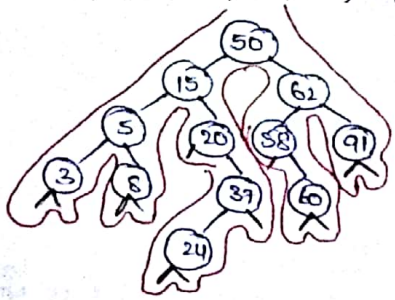
Indirect recursion :

A
B
A
B

Binary Search Trees :

A Binary Tree T is called BST if it satisfies

Ex:- 50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24



*) Creating BST : O(nlogn)
Inorder : O(n)
Sorting : O(nlogn)

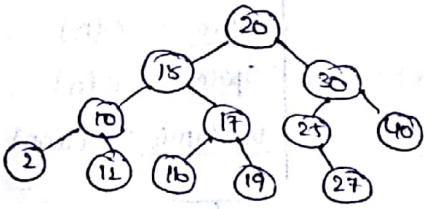
Inorder : 3 5 8 15 20 24 37 50 58 60 62 91
↓
Sorted order :

*) If we apply Inorder on a BST then the resultant will always be in sorted order.

*) Num of BST with n-distinct keys - $\frac{2nC_n}{n+1}$

Operations on BST :

1. Insertion : O(logn)
2. Deletion : O(logn)
3. Searching : O(logn)

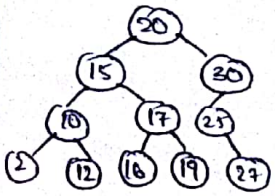


Deletion in BST :

Deletion

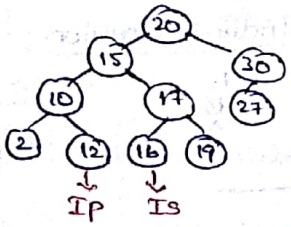
Leaf deletion

Delete : 40



Node with one child

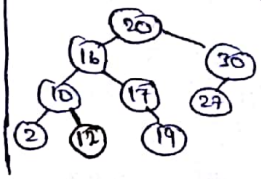
Delete 25 :



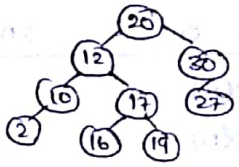
Node with two children

Delete 15 :

Inorder successor (Is)



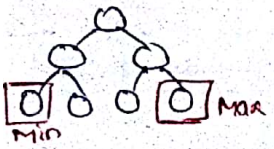
Inorder predecessor (Ip)



*) Deletion may decreare the height of the tree

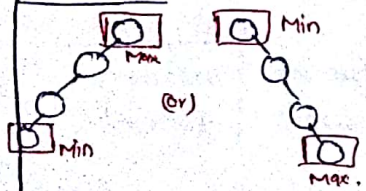
Finding min or max in a BST :

Best Case :



Time : O(logn)

Worst case :



Time : O(n)

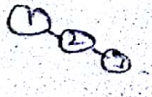
AVL Trees :

A Binary Tree is called AVL tree if left subtree and right subtree are balanced with balance factor -1, 0, 1

BF(N) = $h_L - h_r$

∴ BST

order : 1 2 3 4 ...



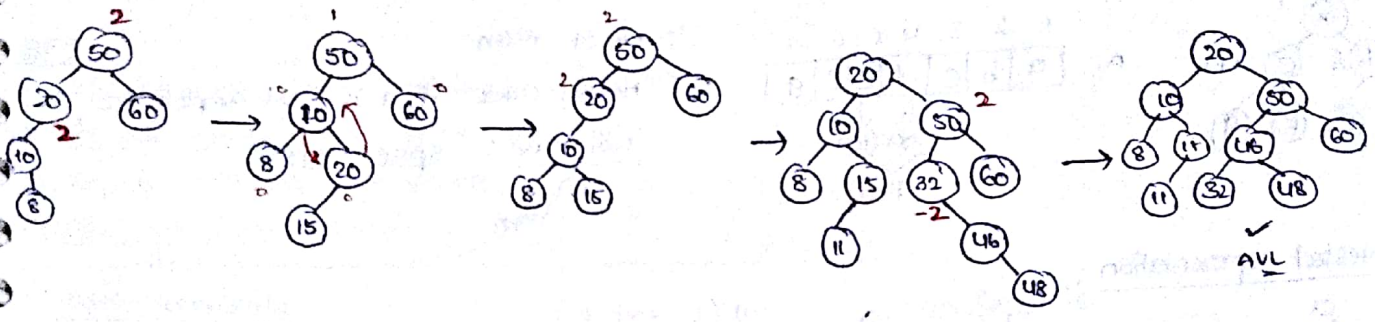
∴ AVL : Height is always balanced
→ Height is always O(logn)
→ For balancing O(logn)

*) In worst case

	BST	Balanced BST AVL
Search	O(n)	O(logn)
height	O(n)	O(logn)
Insertion	O(n)	O(logn) + O(logn) + C

O(logn) + O(logn) + C : Insert; Checking balanced / not; If not Perform rotations.

Ex:- 50 20 60 10 8 15 32 46 11 48



*) Maximum and minimum num of nodes in AVL tree

Min: $N(h) = 1 + N(h-1) + N(h-2)$ where $N(0) = 1$, $N(1) = 2$

Min:

Max: $2^{h+1} - 1$

Ex:- h = 4.

Min: $N(4) = 1 + N(3) + N(2)$
$= 1 + 7 + 4 = 12$

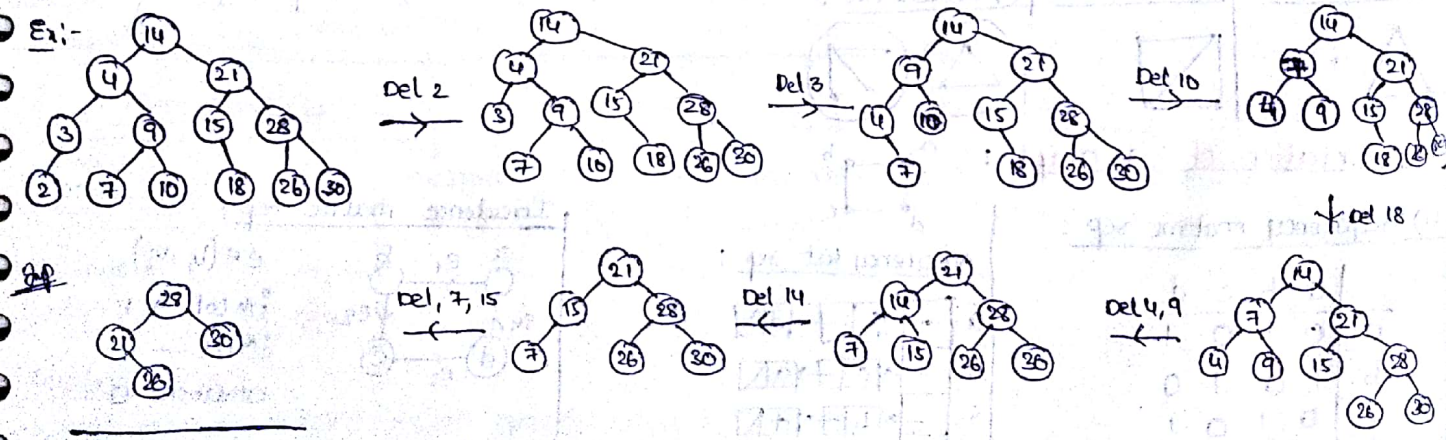
Max: $2^4 - 1 = 31$

Operations on AVL:

AVL Insertion : O(logn)
1) LL Rotation
2) LR Rotation
3) RR Rotation
4) RL Rotation

AVL Deletion : O(logn)
① L_0 ⑤ R_0
L_{+1} R_{+1}
L_{-1} R_{-1}

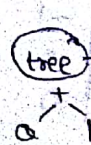
Ex:-



*) A binary tree T has n-leaf nodes. The num of nodes of degree 2 in T is n-1

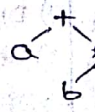
Expression trees:

Expression a+b → Convert → tree → Expression tree.



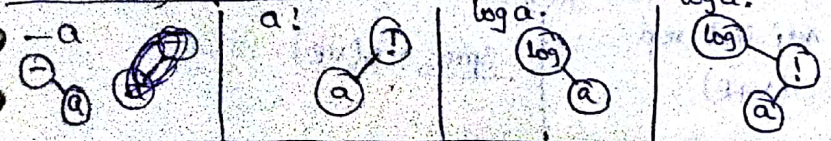
Pre : +ab
In : a+b
Post : ab+

Ex:- a+b*c



Pre : +a*bc
In : a+b*c
Post : abc*+

For unary operators:

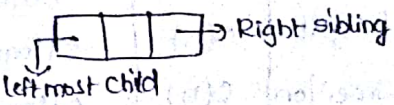


Various tree representations:

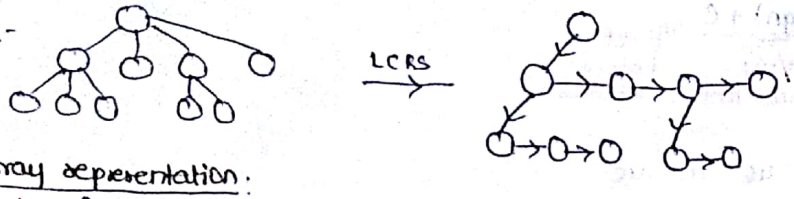
∵ For n-ary tree we should maintain n pointers (space inefficient)

↓sol

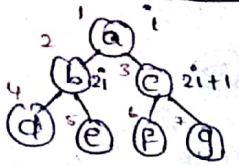
(i) Left child right sibling rep. (2 pointers)

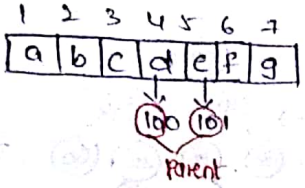


Ex:-



(ii) Array representation:



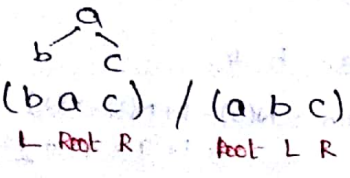


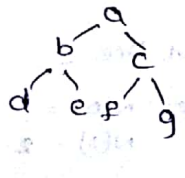
∵ Direct Access

∴ Memory waste if it is not complete tree.

Worst case: space: $O(2^n)$

(iii) Nested representation:





(i) (L Root R)

(d b e) a (f c g))

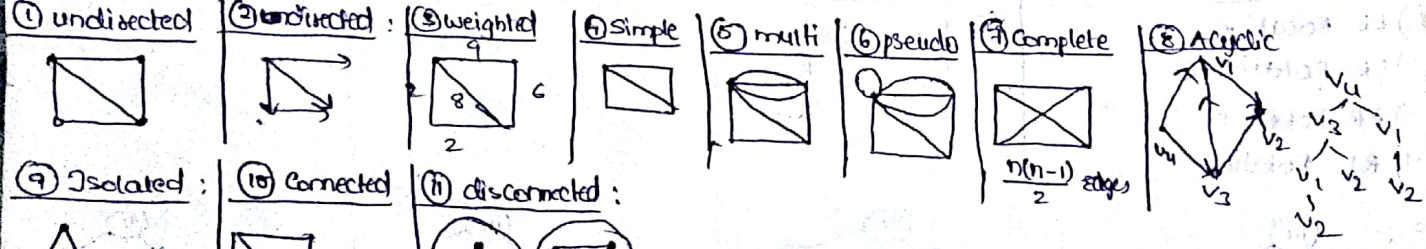
(ii) (Root L R)

(a (b d e) (c f g))

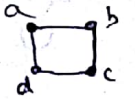
Graphs:

→ Graph is a non-linear DS which contains vertices and edges G(V,E)

Types:



Representation of a graph:



(i) Adjacency matrix rep:

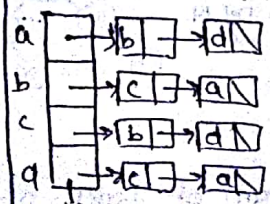
	a	b	c	d
a	0	1	0	1
b	1	0	1	0
c	0	1	0	1
d	1	0	1	0

→ weight can also be placed in the Entries of the matrix

→ If Graph is DENSE (Almost complete graph) ie $E = O(V^2)$ then use Adj matrix rep.

→ Space: $O(V^2)$

Adjacency list rep:



Array of pointers.

Num of lists

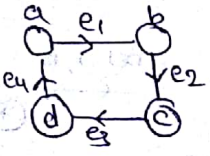
1) Directed → e

2) Undirected → 2e

→ If Graph is sparse ie $E = O(V)$ then use Adj list rep.

→ Space: $O(V+E)$

Incidence matrix rep:



$e \in (V_i, V_j)$

i^{th} col → 1

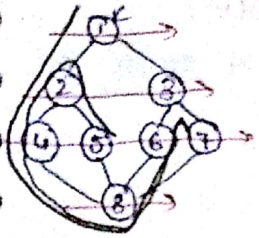
j^{th} col → -1

Otherwise 0

	e_1	e_2	e_3	e_4
a	1	0	0	-1
b	-1	1	0	0
c	0	-1	1	0
d	0	0	-1	1

Space: $O(VE)$

Graph Traversals (BFS and DFS)



BFS : 1 2 3 4 5 6 7 8
DFS : 1 2 4 8 6 3 7 5

Nodes can be
- Visited (refered)
  - 1 2 3 4 5 6 7 8
  - 1 0 0 0 0 0 0 0
  - O(n)
- Explored
  - BFS (queue) — O(n)
  - DFS (stack) — O(n)

→ Space Complexity of traversal is atleast 'n'

BFS:
→ Visit the vertex and update 'visited array'.
→ Add the children of vertex to queue
→ Repeat until all are visited.

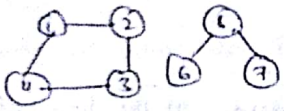
Analysis of BFS:- (Same for BFT)

- Space complexity
  - Input space
    - Adj matrix — $O(V^2)$
    - Adj list — $O(V+E)$
  - Workspace — $O(V)$
- Time Complexity
  - Adj matrix — $O(V^2)$
  - Adj. list — $O(V+E)$

Difference b/w search and Traversal:

Search : visiting only reachable nodes.
Traversal : visiting all nodes.

Ex:-



BFS : 1 2 4 3 | 5 6 7
BFT : 1 2 4 3 5 6 7

Imp!
Also used to find a graph is connected or not.

DFS:

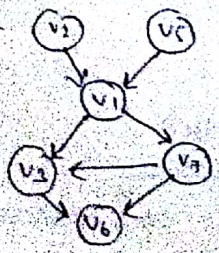
Analysis of DFS (Same for DFT)

- space Complexity
  - Input space
    - Adj matrix — $O(V^2)$
    - Adj list — $O(V+E)$
  - Workspace — $O(V)$
- Time Complexity
  - Adj matrix — $O(V^2)$
  - Adj list — $O(V+E)$

Topological Sort:

→ Applicable to directed Acyclic graph.
→ If there is a edge b/w v_i to v_j then v_i should come before v_j in the sorted order is called Topological sort.

Ex:-



Topological sort:

$V_2 - V_5 - V_1 - V_7 - V_3 - V_6$

$V_5 - V_2 - V_1 - V_7 - V_3 - V_6$.

Algorithm:

Step(1): Identify vertices that has no incoming edges : $O(v) * V \Rightarrow O(v^2)$

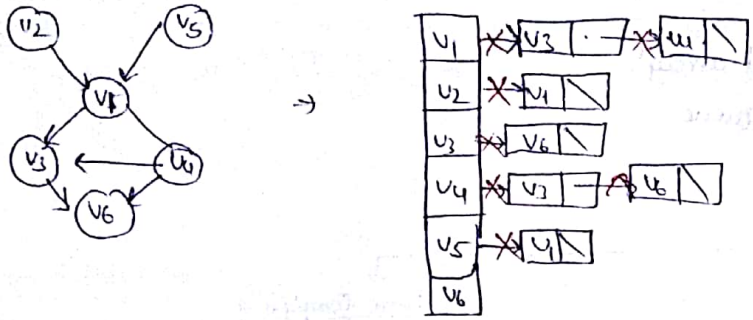
Step(2): Delete this vertex of indegree 0 and all its outgoing edges from the graph and place it in the output : $O(E)$

Repeat step (1) & step (2) until the graph is empty

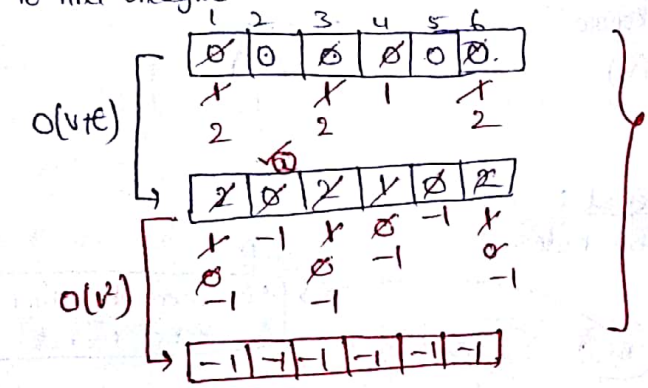
$\Downarrow$

TC : $O(v^2) + O(E) \rightarrow O(v^2)$

Optimizing topological sort :



To find indegree



Topological order

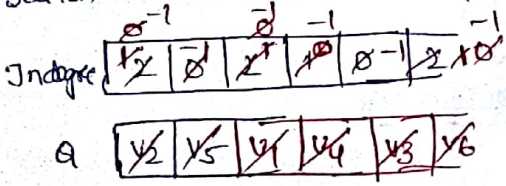
$v_2\ v_5\ v_1\ v_4\ v_3\ v_6$

Time comp $\Rightarrow O(V+E) + O(V^2)$

$\Rightarrow O(V^2)$

Alternate approach :

Instead of finding indegree 0 every time maintain a queue at the time of initialization which contains vertices of indegree 0



$v_2\ v_5\ v_1\ v_4\ v_3\ v_6$.

TC: $O(V+E) + O(V) \Rightarrow \boxed{O(V+E)}$ Imp

Hashing :

Search → Binary search (Ologn) → Efficient → Direct Addr table → ∵ Range should be known & small space wasted.

O(1) ↑

→ Hashing

↓ problem

Collisions

↓ minimize.

Good Hash fn

Collision resolution tech

Chaining ☺ (Ins, Sea, del)

Open Addressing ☺ (Ins, Sea)

Load factor : $\alpha = \frac{n}{m}$ — n = num of Elements, m = Hash Entries

Avg search = $O(1+\alpha)$

$= O\left(1+\frac{n}{m}\right)$ if $n = c*m$

$= O\left(1+\frac{c \cdot m}{m}\right)$

Avg search	$O(1+c) \Rightarrow O(1)$
Avg deletion	$O(1)$

Chaining : (open hashing)

→ $\alpha \geq 1$

→

Insertion	$O(1)$
Search	$O(n)$
Deletion	$O(n)$

worst case.

∵ store out side of Hash table more space is reqd for pointers.

Open Addressing : (closed hashing)

→ max num of keys is atmost 'm'

→ $0 \leq \alpha \leq 1$

→

	Avg	Worst
Insertion	$\theta(1)$	$O(n)$
search	$\theta(1)$	$O(n)$

→ Extra space not reqd

∵

*) It gives wrong results beoz of deletion Nulls will accomodate will not give scope for further searching

↓ solution

Keeping marker at deleted Element

↓ Problem with search

↓ usually $\theta(1)$ — ↓ Due to markers $O(m)$

*) probability of filling an Element in primary clustering is $\frac{i+1}{m}$

Points to remember

Name	Mechanism	Primary clustering	Secondary clustering	Probe sequences
Linear probing	$h'(k) = (h(k)+i) \bmod m$	✓	✓	m
Quadratic	$h'(k) = (h(k) + c_1 i + c_2 i^2) \bmod m$	×	✓	m
Double	$h'(k) = (h_1(k) + i\, h_2(k)) \bmod m$	×	×	m^2

Should be relatively prime ie GCD = 1

*

$h_2(k)$	m
odd	2^k
$<m$	prime

x——The End——x