DSA

Array → An array is the collection of homogenous data.

Algorithm for Inserting any element at the beginning of the array →

show A as A[ ] array

```
Ins Beg (A, item, n, maxsize)
1) If n = maxsize then write "overflow" & exit.
2) For i = n to 1
    a) A(i+1) = A(i)
    b) i = i-1
3) A[1] = item ; n = n+1
4) Exit
```

Algo for Inserting any element at the end of an array →

```
Ins End(A, item, n, maxsize)
1) If n = maxsize then write "overflow" & exit.
2) A(n+1) = item
3) n = n+1
4) exit
```

CPU Scheduling ↔ Multitasking by CPU ↔ Multiprogramming by CPU

Algo for deleting an element from end →

```
Del End (A, n, max size, item)
1) If n = 0 then write "underflow" & exit.
2) item = A[n]
3) n = n-1
4) exit.
```

Algo for deleting an element from beginning →

```
Del Beg (A, n, maxsize, item)
1) If n = 0 then write "underflow" & exit
2) item = A[1]
3) for i = 2 to n
       a) A[i-1] = A[i]
       b  i = i+1
4) n = n-1
5) exit
```

Algo for inserting an element at a specific location →

```
Ins Spec (A, i, n, K, ~~s~~ item} maxsize)
1) If n = max size, then write "overflow" & exit;
2) For (i = n ; i => K , i--)          → If K > n then
       a) A[i+1] = A[i];                    a) A[n+1] = item
                                            b) return
3) A[K] = item
4) n = n+1
5) exit
```

Algo for deleting an element from a specific location ->

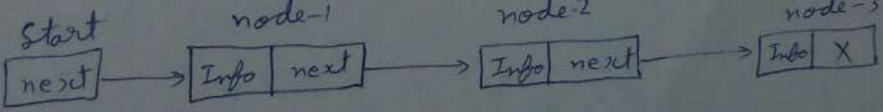
```
Ins Del (A, i, n, K, item)

1) If n=0, then write "underflow" & exit
2) item = A[K]
3) For (i = K+1 to n)
      a) A[i-1] = A[i]
      b) i = i+1
4) n = n-1
5) exit
```

To print each and every element of an array ->

```
1) For (i=1 to n; i++)
      a) Print A[i]
2) Exit
```

Linked list (LL) -> When there is no continuous location in the memory to save the elements then we use linked list, in it elements are saved at different locations but their location is stored in the previous element's node.



Info = element / used to store values

next = location of next node.

Start = location of ~~next~~ first node.

Algo for Inserting an element at the Beginning of (LL) →

```
Insbeg (start, item)
1) If Avail = Null(x) then write "overflow" & exit
2) Newnode = Avail
3) Avail = Avail → next
4) ~~Create~~ Newnode → Info = item
5) Newnode → next = start
6) Start = newnode
7) Exit
```

→ Create Newnode

Avail → It is a linked list of all the empty nodes available in the memory.

Algo for Inserting an element at the end of (LL) →

```
Ins End (start, item)
1) create Newnode.
2) Newnode → info = item
3) PTR = start
4) While (PTR → next ! = null)
      a) PTR = PTR → next
5) PTR → next = Newnode
6) Newnode → next = NULL
7) Exit
```

PTR → Pointer for storing the location

! = → not equal

& A

```
1) If start = NULL then start = Newnode
2) newnode → Info = item
3) newnode → next = NULL
4) Exit
```

Algo for inserting an element at a specific location in LL ->

```
Ins spec (start, item, K)                       K = ~~item~~ no. where
                                                we want to add
                                                item
1) If Avail = Null then write "overflow" & exit.
2) Newnode = Avail
3) Avail = Avail -> next
4) Newnode -> Info = item
5) If K = 1 then Newnode -> next = start (address of next/node)
                                                          /Ist
      a) Start = newnode
      b) return.
6) PTR = start, counter = 1                     (Counter = for storing no of
                                                 the element)
7) while (counter < K && PTR ! = NULL)
      a) Prev = PTR                             (Prev = Pointer for storing
      b) PTR = PTR -> next                       the address of prev element
      c) Counter = Counter + 1                   /node
8) If PTR = NULL then
      a) Prev -> Next = Newnode
      b) Newnode -> next = NULL
      c) exit
9) Newnode -> Next = PTR -> Next
10) PTR -> Next = newnode
11) Exit
```

Advantages of LL ->

1) It is dynamic means ~~so~~ it does not have a fixed size.

2) It do~~e~~ not require continuous memory location.

3) If we need to insert or delete an element from a specific location then we do not have to shift all the elements, we just need to add a newnode at that ~~pos~~ location.

Algo for deleting an element from beginning of(LL) ->

```
Del Beg (start, item)
1) If start = NULL, Then write "underflow" & exit
2) item = start -> info ( info of Ist node)
3) Temp = start                (Temp -> Temporary variable)
4) start = start -> next
5) Free Temp
6) Exit
```

Algo for deleting an element from end of(LL) ->

```
Del End (start, item)
1) If start = NULL then write "underflow" & exit
2) If start -> next = NULL
      a) temp = start
      b) start = NULL
      c) free temp
      d exit
3) PTR = Start
4) While (PTR -> next ! = null)
      a) Prev. = PTR
      b) PTR = PTR -> next
5) Prev. -> next = null
6) Free PTR
7) Exit
```

Algo for deleting an element from a specific location in(LL)

```
Del Spec (start, item, k)

1) If Start = null then write "underflow" and exit
2) If K = 1 then item = start → info
      a) start = start → next
      b) exit
3) PTR = start, counter = 1
4) while (counter < K && PTR ! = NULL)
         a) Prev = PTR
         b) PTR = PTR → next
         c) Counter = Counter + 1
5) If PTR = NULL then write not enough elements & exit.
6) Prev → next = PTR → next
7) Item = PTR → info
8) Free PTR
9) Exit
```

To print each element of the linked list →

```
1) PTR = Start
2) while (PTR ! = Null)
         a) Print PTR → info
         b PTR = PTR → next
3) Exit
```

TRAVERSAL → Printing each & every element of a data structure

Traversal

Applications of stacks →

Infix expression → when operator comes in between operants

eg → A+B, A-B

Postfix expression → when operator comes after operants.

eg → AB+, AB-

Prefix expression → when operator comes before operants.

eg → +AB, -AB

Infix to postfix Conversion →

1) Add a left parenthesis to the stack and a right parenthesis at the end of infix expression.
2) Scan the infix expression from left to right.
3) If the scanned character is an operant then add it at the end of postfix expression.
4) If the scanned character is an operator then check the top element (operator) of the stack.
5) If the precedence (priority) of the current operator is lesser than the top operator of the stack then pop out all the operators which ~~are~~ have equal or greater precedence.
6) If the precedence of the current operator is greater than the top operator of the stack then add it to the stacks.
7) If a left parenthesis [( ] is found then add it to stacks.
8) If a right parenthesis [ )] is found then pop out all the elements from the stack and add to the postfix until a left parenthesis is encountered.

Priority of (+) and (-) is equal and (x) and (/) is equal

Priority of (^) is highest

Let (P) an infix expression and (Q) be its postfix expression

$$P = A+B-(c \times D)/E \wedge F \times G - H + I)$$

$$\boxed{Q = AB+CD\times EF\wedge/G\times-H-I+}$$

Symbol scanned	Stack
	(
A	(
+	(+
B	(+
–	(–
(	(–(
C	(–(
×	(–(×
D	(–(×
)	(–
/	(–/
E	(–/
^	(–/^
F	(–/^
×	(–×
G	(–×
–	(–
H	(–
+	(+
I	(+
)	()
	Empty

Evaluation of postfix expression →

1) Scan the expression from left to right.

2) If an operant is encountered push it on the stack.

3) If an operator ⊗ is encountered pop the top 2 elements (let A & B) and evaluate $(A \otimes B)$.

4) Push the result back on the stack.

Eg →

Let

AB+CD×EF^/G×-H-I+

Value of A ⟶ I = 2

Symbol encountered	Stack	Evaluation
A	A	
B	A, B	
+	4	A + B = 2+2 = 4
C	4, C	
D	4, C, D	
×	4, 4	C × D = 2×2 = 4
E	4, 4, E	
F	4, 4, E, F	
^	4, 4, 4	$E^{\wedge F} = 2^{\wedge 2} = 4$
/	4, 1	4/4 = 1
G	4, 1, G	
×	4, 2	1 × G = 1×2 = 2
-	2	4 - 2 = 2
H	2, H	
-	0	2 - H = 2-2 = 0
I	0, I	
+	2	0 + I = 0+2 = 2

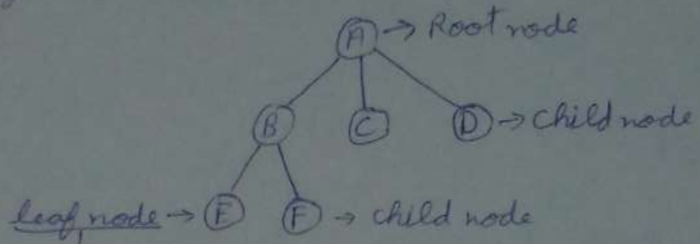
Now check the result by putting the values of operants in the infix expression. If both results are same it means the evaluation is correct.

TREES → A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search.

→ It is a Non-linear data structure.

→ The topmost node is called the root and the nodes below it are called the child nodes. Each node can have multiple child nodes.

eg →

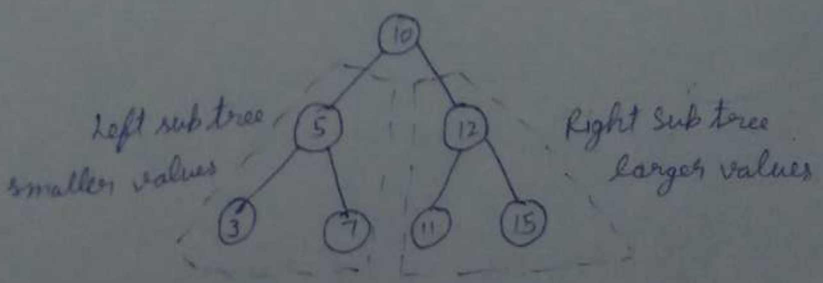


↳ The last node / which do not have a child is a leaf node.

Binary trees → It is a special type of tree in which a parent can have only a maximum of 2 child nodes.

Binary Search tree → The subtree in the left of the root will have values smaller than the root value.

→ The subtree in the right of the root will have values greater than the root value.



Always take first element of the list as root node.

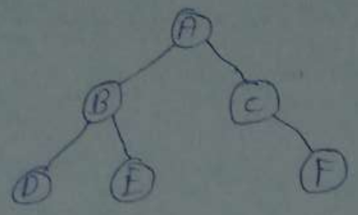
Binary Search tree ->

Let location of a node = K^{th}

Then its left value will be at = $(2 \times K)^{th}$ position

& " Right " " " " = $(2 \times K+1)^{th}$ "

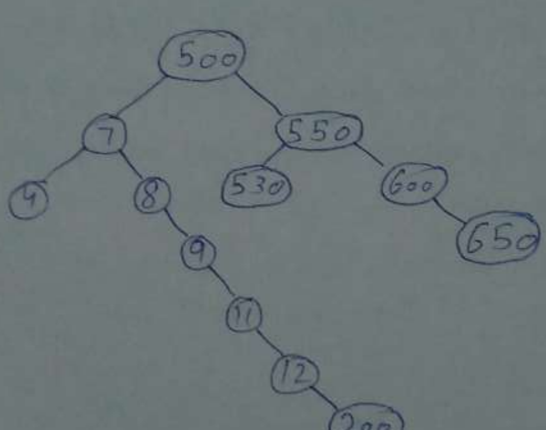
eg ->



A, B, C, D, E, , F

Q -> Make a binary search tree out of ->

500, 7, 8, 550, 4, 9, 11, 12, 600, 650, 530, 200



Traversal -> In Binary search tree we have 3 type of traversals

Inorder -> Left, Root, Right

Preorder -> Root, Left, Right

Postorder -> Left, Right, Root

For making a tree back from traversal we need at least 2 types one is Inorder & other one from remaining 2

Inodr → 4, 7, 8, 9, 11, 12, 200, 500, 530, 550, 600, 650

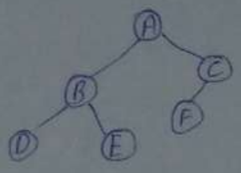
Pre.odr → 500, 7, 4, 8, 9, 11, 12, 200, 550, 530, 600, 650

Postodr → 4, 200, 12, 11, 9, 8, 7, 530, 650, 600, 550, 500

Q → Print/make tree from

Inodr → $D\underset{L}{B}E\,A\,\underset{R}{F}C$

Preodr → $\underset{①}{A}BDECF$



Deletion from binary search tree →

There are 3 cases from deletion in a binary search tree →

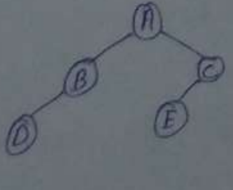
Case 1 - Value present at leaf node

Case 2 = Value present at which it has 1 child node

Case 3 = Value present at a node which has 2 child nodes.

Traversal of trees in terms of array → If Ist element is at 1 element at = K and
its left child at = 2 × K
its right child at = 2 × K + 1

If Ist element is at 0 then
Left child = 2 × K + 1
Right child = 2 × K + 2

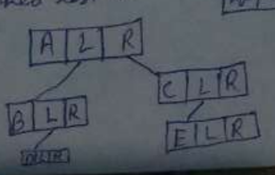


A	B	C	D	-	E	-
1	2	3	4	5	6	7

Left of A = 2×1 = 2
Right " " = 2+1 = 3

Left of B = 2×2 = 4
Left of C = 2×3 = 6

In Linked list →



info	Left	Right

Left & Right are address of left & right child.

empty left or right will be NULL

AVL tree → It is a special type of binary search tree. It is a self balancing tree where the difference b/w the heights of left and right subtrees cannot be more than 1.

AVL → Adelson-Velskii and Landis

Balance factor = Height of Left Subtree - Height of Right subtree

For a balanced tree balance factor should be → +1, -1, 0

Height = no. of elements in the tree

During making a tree →

→ check balance factor after every insertion.
→ Always count balance factor from bottom node.
→ Don't go upward from critical node.
↳ which has a misbalanced factor.

If the balance factor is misbalanced
then it can have 4 possibilities →

Misbalanced due to Left node of Left node → LL
" " " Left node of Right node → RL
" " " Right " " Left " → LR
" " " Right " " Right " = RR

For balancing the balance factor we have to rotate the tree i.e
for
LL → 1 Rotation ⇒ ↷
RR → 1 " " ⇒ ↶
RL → 2 Rotations ⇒ ↷↶
LR ⇒ 2 " " ⇒ ↶↷

is a

the

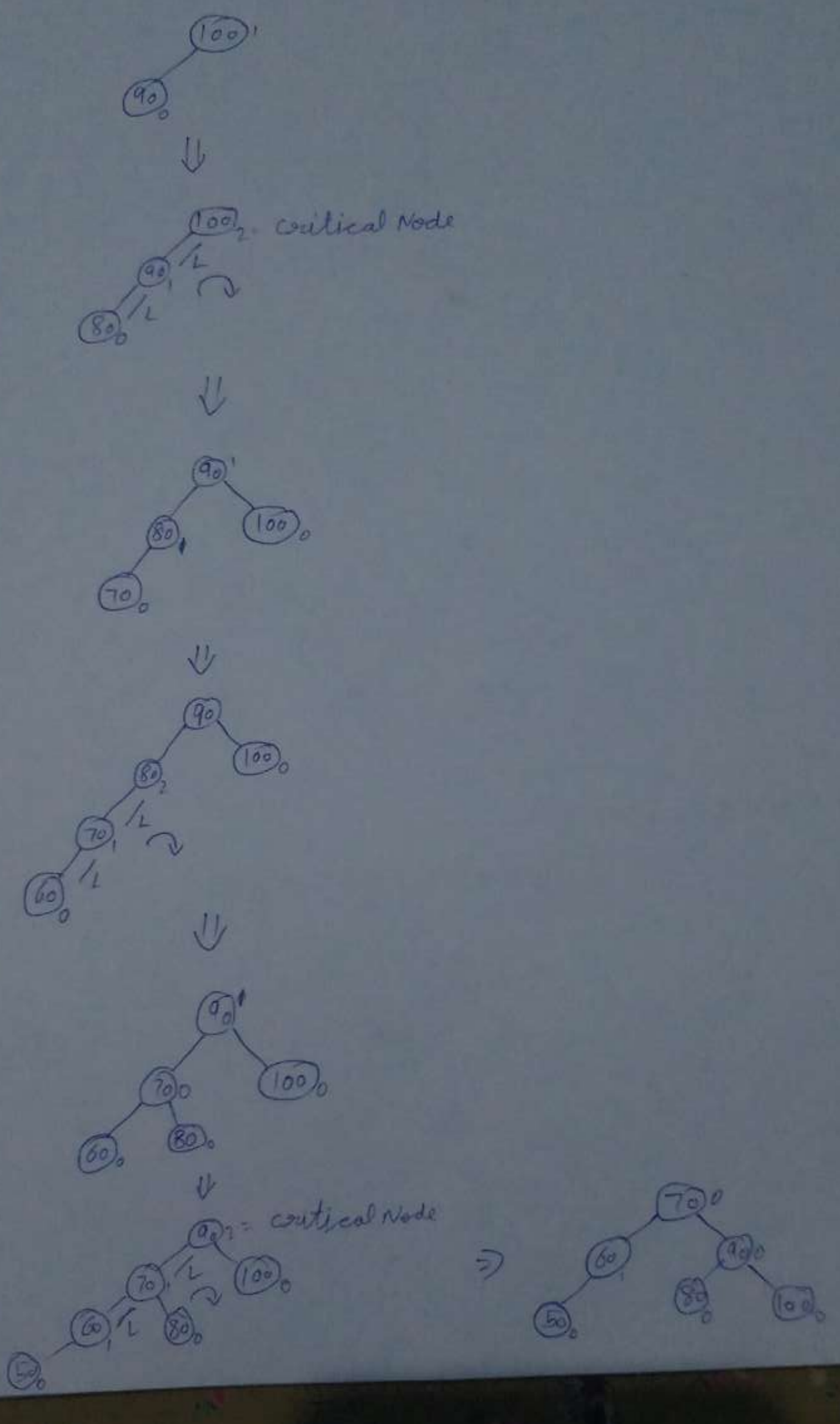
than

tree

Balanced

Q -> Make AVL tree for ->

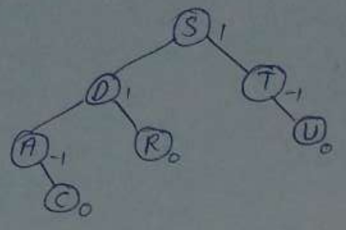
100, 90, 80, 70, 60, 50



Deletion →

Case 1 → Value present at leaf node.

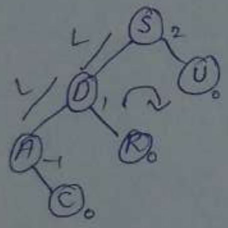
Delete E →



All nodes are balanced.

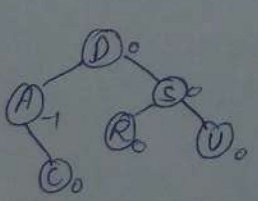
Case 2 ⇒ Value present in the node having 1 child node.

Delete T →



Its position is given to its child.

Rotation →



Case 3 ⇒ Value in the node having both child nodes.

Delete R →

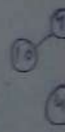
Step 1 → Write the inorder traversal of the tree.

Inorder ⇒ A, C, D, R, S, V

Step 2 → Give
succe

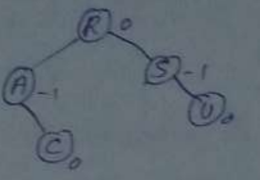
eg → if in the

Illy →

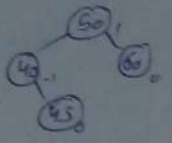


Dele

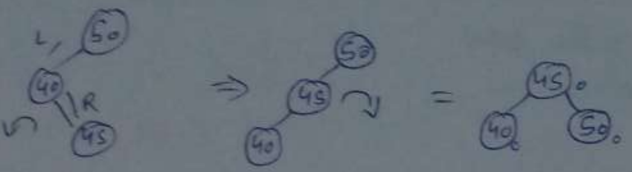
Step2 → Give the position of deleted element to its inorder successor. (i.e R)



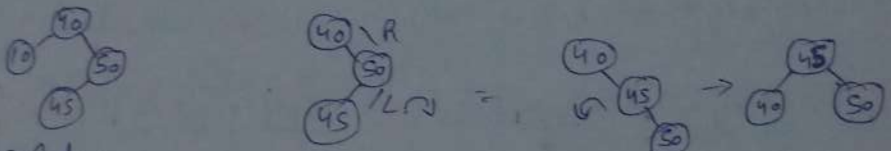
eg → if in the given example



we have to delete [60] then →



IIly →



Delete 10 →

B-Trees ->

m-way search trees -> It has m. no of ptr. and m-1 key values.

Conditions for B. tree ->

1) All leaf nodes should be at same level.
2) It should be a binary search tree.
3) It should have a minimum of

$m/2$ = child nodes/ptr. where $\lceil \frac{m}{2} \rceil$ eg $\lceil \frac{5}{2} \rceil = 3$

$\lceil m/2 \rceil - 1$ = Key values

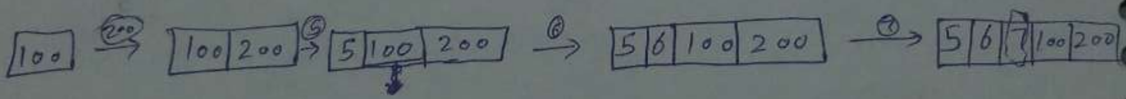
4) It can have a maximum of ->

m = child nodes /ptr.

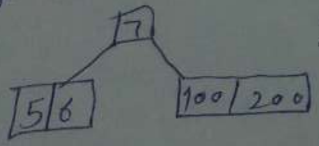
m-1 = Key values.

eg -> Make a 5 way tree ->

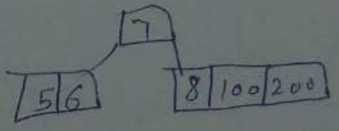
100, 200, 5, 6, 7, 8, 9, 10, 300, 310, 320, 330, 340, 350



split ->



Now check the conditions and add upcoming elements in same way



Deletion in B-

Case 1 (leaf

Part 1 =>

eg ->

Delete 4

Part 2 => el

eg ->

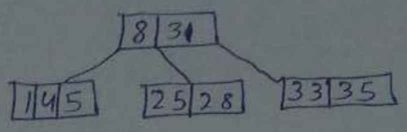
Delete 2

Deletion in B-trees →

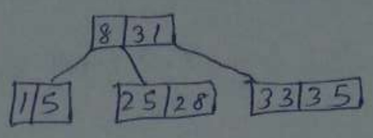
Case 1 (Leaf node)

Part 1 ⇒ elements in the node are more than the min. no. of values.

eg →

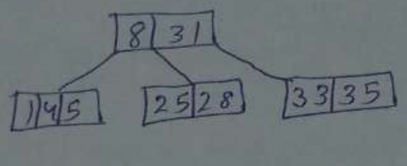


Delete 4 →

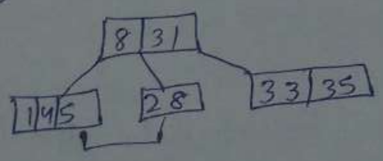


Part 2 ⇒ elements in the node are equal to the min. no. of values.

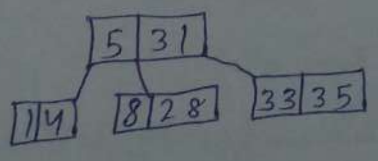
eg →



Delete 25 →



- Now check that its sibling have extra element or not
- sibling = having same parent
- take largest value from left
- take smallest value from right


5 | 31
1 | 4
8 | 28
33 | 35