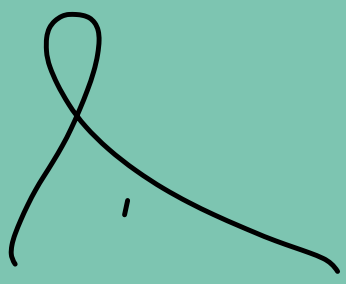
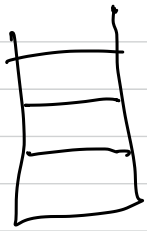
Data Structure

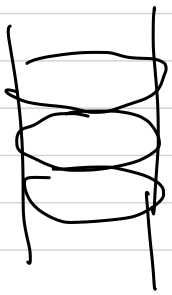


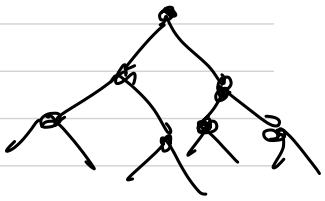
Algorithms

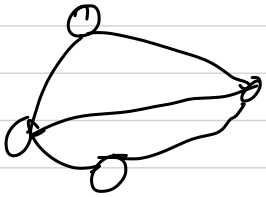
Data Structures - Categories

Linear Data structures	Non-linear DS
stacks, Queues, Linked lists	Trees, Heaps, Graphs









Algorithms —

- plan or blueprint
- Sets of instructions
- Execute finite amount of time

Abstract Data Type (ADT):

- Primitive Data Type
- Non-Primitive Data Type

} Data Types

Abstract Data Type specifies:

- Type of data represented or stored
- Operations supported
- parameters for operations

- ADT specifies WHAT each operation does, NOT HOW operations are done.

- Algorithm analysis -

+ Constant time execution:

Declarations
Assignment
Arithmetic Operations
comaparison statements
Accessing elements
Calling functions, Returning functions.

```
int total = 0;   - 1
int i = 1;       - 1

While (i <= n)   n+1
{   total = total + i;
    i = i + 1;
}
```

primitive operation	freq
Declarations	2
Assignment	2
comparison op	n+1
Arithmatic op	n+n

$2+2+(n+1)+(n+n)$

$= 5+3n$

$= O(n)$

Time complexity & Recursion: Recurrence Relation

```
cal (int n)          → T(n)
{ if n>0             → 1
    k = n^2          → 1
    print (k)        → 1
    cal (n-1)        → T(n-1)
}

main  cal (4);
```

$\Rightarrow T(n) = 1+1+1+(T_{n-1})$

$= 3 + T(n-1) \quad n>0$

$\downarrow$

1 if $n=0$

solve $T(n)$:

$$T(n) = \begin{cases} T(n-1) + 1 & n > 0 \\ 1 & n = 0 \end{cases}$$

methods

- subs
- master theorem

Constant = c → 1

$$T(n) = T(n-1) + 1 \quad (1)$$

$$T(n-1) = T(n-1-1) + 1 = T(n-2) + 1 \quad (2)$$

$$T(n-2) = T(n-2-1) + 1 = T(n-3) + 1 \quad (3)$$

Substitution:

$$T(n) = T(n-1) + 1$$

$$T(n) = T(n-1) + 1$$

$$T(n) = T(n-2) + 1 + 1 = T(n-2) + 2$$

$$T(n) = T(n-3) + 1 + 2 = T(n-3) + 3$$

$$\vdots$$

$$T(n) = T(n-k) + k$$

$$T(n) = T(n-k) + k$$

$n - k = 0 \Rightarrow k = n$

$T(n) = T(n-n) + n$

$= T(0) + n$

T $n=0$

↓

$T + n$

$T(n) = 1 + n$

$T(n) = O(1+n)$

$= O(n)$

- Types of recursions —

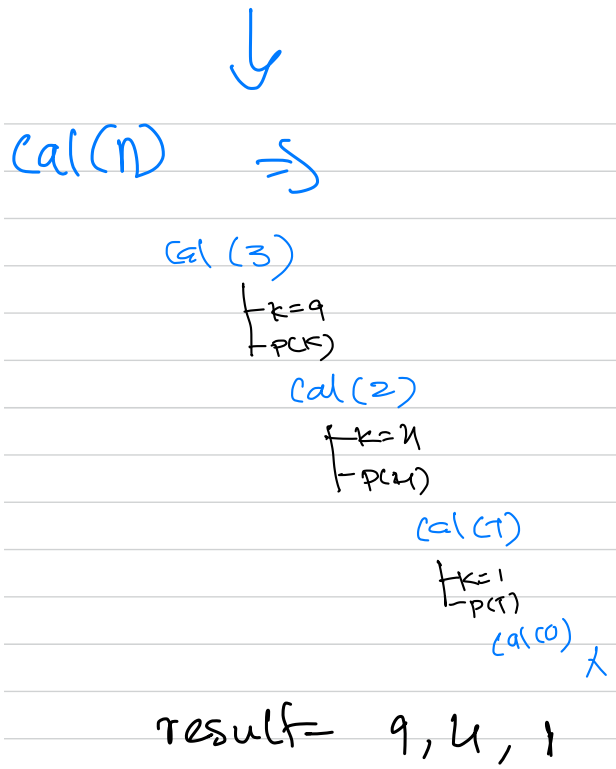
- Tail recursion
- Head recursion
- Tree recursion
- Indirect recursion

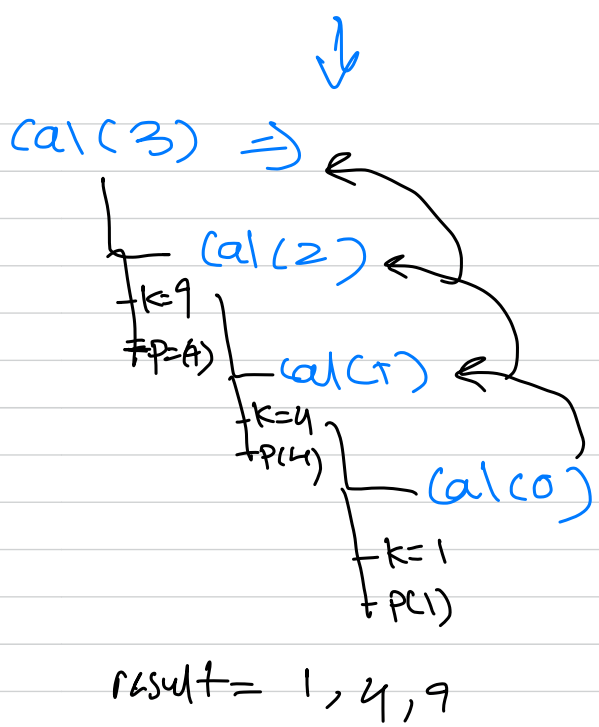
Tail

```
f   cal(n)
    if n>0
    { k<= n^2
      print(k)
      cal(n-1)
    }
```

Head

```
{   cal(n)
    if n>0
      cal(n-1)
      k<= n^2
      print(k)
    }
```

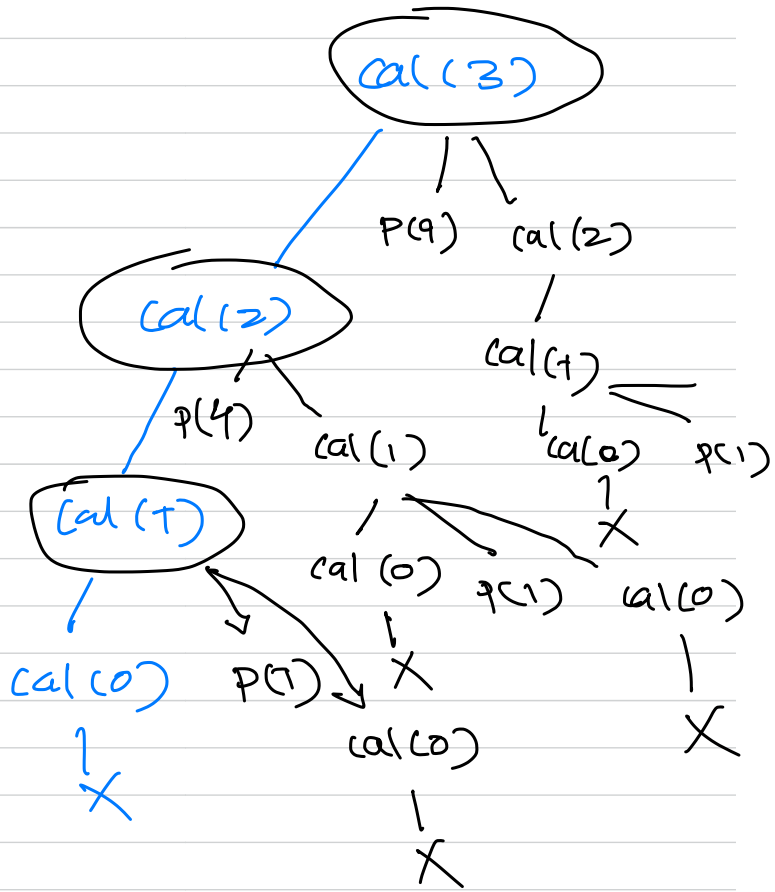


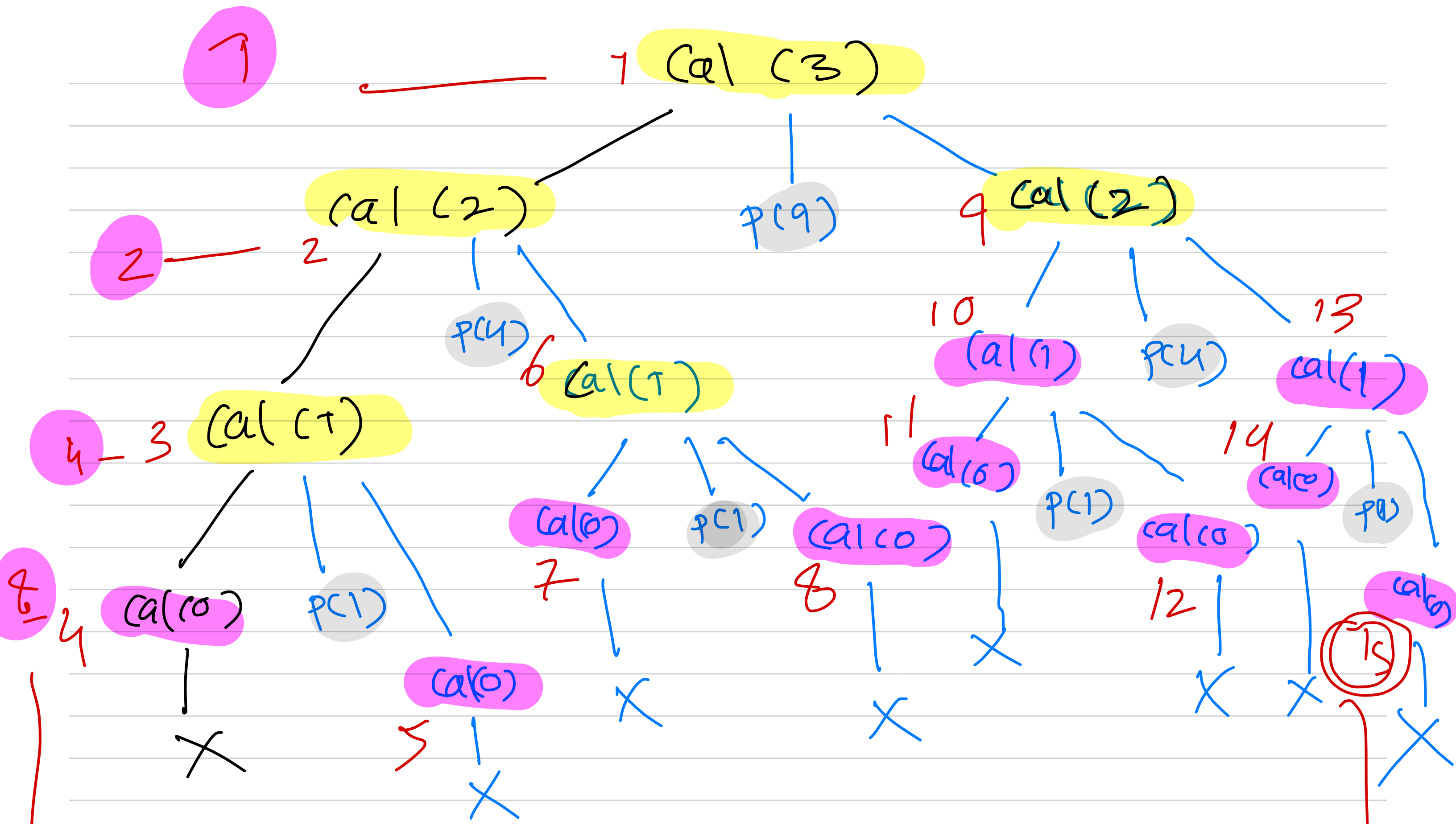


Tree recursion: when a function calls itself more than once.

```
cal(n)
{  if n>0
       cal(n-1)
       k= n^2
       print(k)
       cal(n-1)
}
```

output: 1 4 1 9 1 4 1





1 4 1 9 1 4 1

(15) recursive calls

$n=3$ level = 4

$n=4$ level = 5 ...

$1+2+4+8 = 15$

$= 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + \cdots + 2^n = O(2^n)$

e.g. :) $n = 3$

$2^{3+1} - 1 = 15$

$16 - 1 = 15$

$2^{n+1} - 1$

Indirect Recursion - We will have more than one function and these functions will be calling the other in a loop or a circular pattern.

```
CalculateA(n)
{   if n>0
    ...
    CalculateB(n-1)
    ---
}
```

```
CalculateB(n)
{   if n>0
    ---
    CalculateA(n-1)
    ---
}
```

Linear Search Algorithm –

1) Go through the array till last element
2) return that index if match found
3) if not return -1.

```
int LinearSearch ( Array, n, key )
                          length of array
    index = 0
    while ( index < n ) do
        if A[index] == key then
            return index;
        index = index + 1;
    return -1;
```

Binary Search iterative:

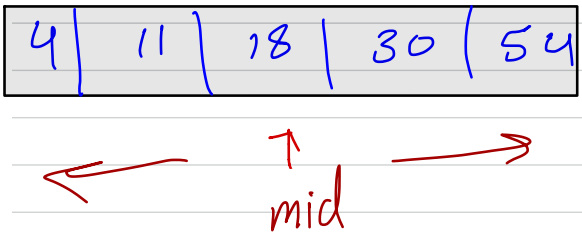
1) Array is in sorted order

2) Examine the middle element

3) if matches found, return index

4) if key < middle element, search lower half

5) if key > middle element, search upper half




```
int BinarySearch (A, n, key)
{
    L = 0
    R = n-1
    while( L ≤ R) do:
        m = floor((L+R)/2)
        if key == A[m] then
            return m;
        else if key < A[m]
            R = m-1
        else if K > A[m]
            L = m+1
    return Not found;
```

- Binary Search Recursive:

```
BinRec ( A , key , L , R )
{   if L > R
        return Notfound;

    else
        m = floor ( (L+R) /2);
        if key == A[m]
            return m
        else if key < A[m]
            return BinRec (A, key, L, m-1);
        else if key > A[m]
            return BinRec (A, key, m+1, R);
```

Sorting Algorithms –

Selection
Insertion
Bubble
merge
Quick
Shell
Heap

} Comparison Based

Count
Bucket
Radix

} Index Based

Stable & Unstable Sorting:

- The concept of stable and unstable.

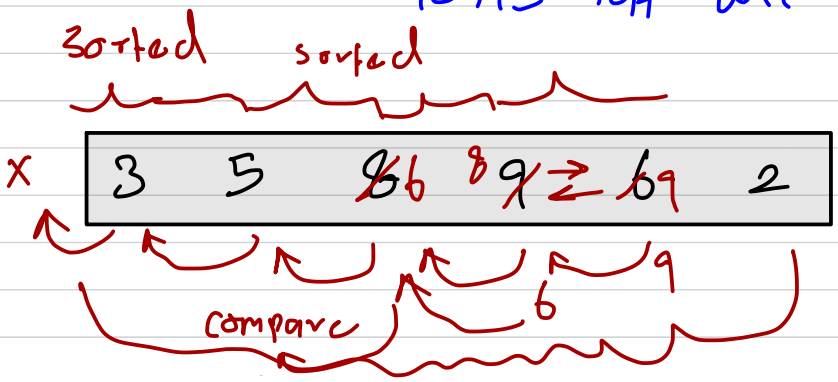
  → when there duplicate elements in a collection.

- Insertion sort:

1) Select one element at a time from the left of the collection.

2) Insert the element at proper position.

3) After insertion, every element to its left will be sorted.



- Analysis -

comp : $1+2+3+\cdots+n-1$

swap : $1+2+3+\cdots+n-1$

$= n(n-1)/2 = n(n-\frac{1}{2})$ $O(n^2)$

$= O(n^2)$

Time: $O(n^2)$

```
insertionS ( Arr, n )
{   for i=1, i<n-1; i++
        value = A[i]
        pos = i
    while pos >0 and A[pos-1] > value
        A[pos] = A[pos-1]
        pos--;
    A[pos] = value.
}
```

Bubble Sort :

1) Compare the consecutive elements

2) if element is greater than the right, swap them.

3) Continue till the end of the collections & perform several passes to sort the elements.

```
BubbleSort ( A )
{
  for pass = n-1, pass >= 0, pass--
    for i=0, i < pass, i++
      { if A[i] > A[i+1]
          temp = A[i]
          A[i] = A[i+1]
          A[i+1] = temp
      }
}
```

Time = $O(n^2)$

Shell Sort :

1) Selects an element and compare element after a gap

2) Similar to Insertion sort

3) Insert selected element from the gap at its proper position.

```
ShellSort ( A )
{
  for gap = n/2 , gap > 0 , gap = gap/2
      for i = gap , i < n , i++
          gvalue = A[i]
          j = i - gap
          while (j >= 0 and A[j] > gvalue)
              A[j+gap] = A[j]
              j = j - gap
          A[j+gap] = gvalue
}
```

Time complexity –

Rounds:

$gap = n/2, gap/2, gap/2 \cdots 0 = \log_2(n)$

$O(n \log(n))$

- **Merge Sort:**

1) Divide the collection of elements into smaller subsets

2) Recursively sort the subsets

3) Combine or merge the result into a solution

4) Divide and Conquer approach

```
mergesort ( A, left, right )
{   if left < right
        mid = ((left + right) / 2
        mergesort ( A, left, mid )
        mergesort ( A, mid+1, right )
        merge ( A, left, mid, right )
}
```

$O(\log_2 n)$

Time complexity:

$O(n \log(n))$

```
merge ( A, left, mid, right )
{   i = left, j = mid+1, k = left
    while i ≤ mid && j ≤ right
        if A[i] < A[j]
            B[k] = A[i]
            i = i+1, k = k+1
        else
            B[k] = A[j]
            j = j+1, k = k+1
    while i ≤ mid
        B[k] = A[i]
        i = i+1, k = k+1
    while j ≤ right
        B[k] = A[j]
        j = j+1, k = k+1
    for m = left, m ≤ right, m++
        A[m] = B[m].
```

$O(n)$

merging:

two-merge sort

$O(n \log n)$

A	B	C
i ← 2	5 → j	2 ← k
8	9 → j	5
i ~~<~~ 15	12 → j	8
i ← 18	17 → j	9
	j	12
		15
		17
		18

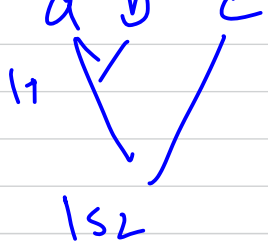
2 < 5 = 2, i++

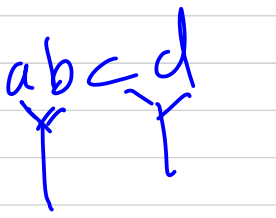
8 < 5 = 5, j++

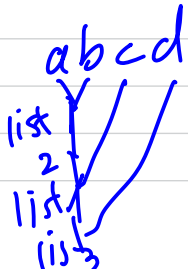
```
if ( i < j )
    add i to c
    i++
else
    add j to c
    j++
```

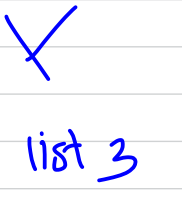
m number of lists →

m-way merge sort.









Quick Sort:

1) Divide the collection of elements into subsets or partitions.

2) Partitions based on Pivot

3) Recursively sort the partitions using quick sort.

4) Divide and Conquer approach

Quick sort:

Partitions at Levels: $1+2+4+8+16+\cdots$

$= 2^0 + 2^1 + 2^2 + \cdots + = \log_2(n)$

```
quickSort ( A, low, high )
{
    if  low < high
        pi =  partition (A, low, high)
        quickSort (A, low, pi-1)
        quickSort (A, pi+1, high)
}
```

comparison at each level: n

Total:

$n \times \log_2(n)$

$= O(n\log(n))$

```
partition (A, low, high)
{
    pivot = A[low]
    i = low, j = high
    do
    {   do
            i = i+1
        while ( A[i] <= pivot)
        do
            j = j-1
        while (A[j] > pivot)
        if  i < j
            swap ( A[i], A[j])
    } while(i<j)
    swap (A[low], A[j])
    return j.
}
```

Partitions at levels: $1+1+1\cdots$ ($=n$)

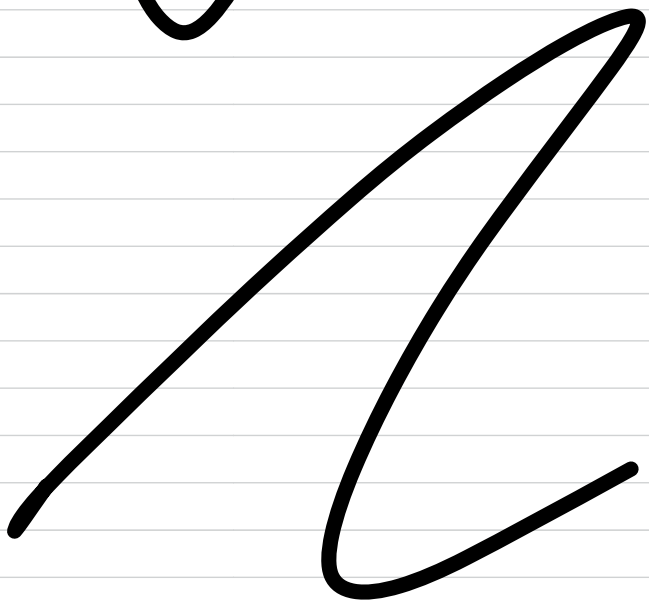
Comparison:

$n + (n-1) + (n-2) \cdots + 3 + 2 + 1$ ($:n$)

$O(n^2)$

Algorithms	Best case	Avg	worst	Space	Stable
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Shell	$O(n \log n)$	//	$O(n^2)$	$O(1)$	No
Merge	$O(n \log n)$	//	$O(n \log n)$	$O(n)$	Yes
Quick	$O(n \log n)$	//	$O(n^2)$	$O(n)$	no
Heap	$O(n \log n)$	//	$O(n \log n)$	$O(1)$	no

Linked List



Linked list :

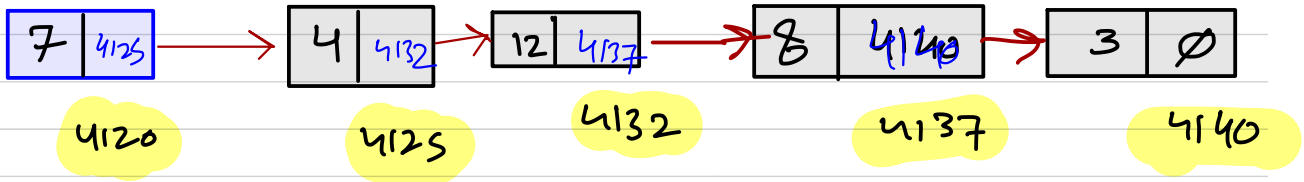
Is a collection of ele ments , where each element is represented as a node

Node : contains

stores the address of the next node

[7, 4, 12, 8, 3]

nodes :

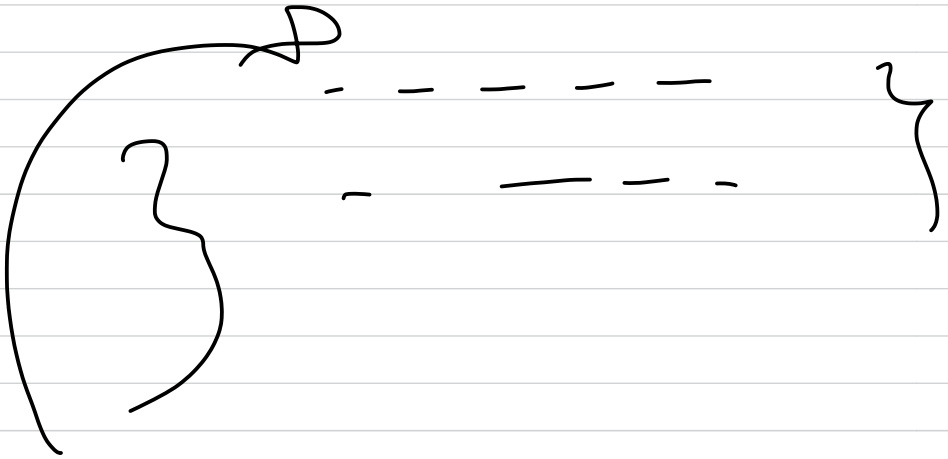


* Creating Node

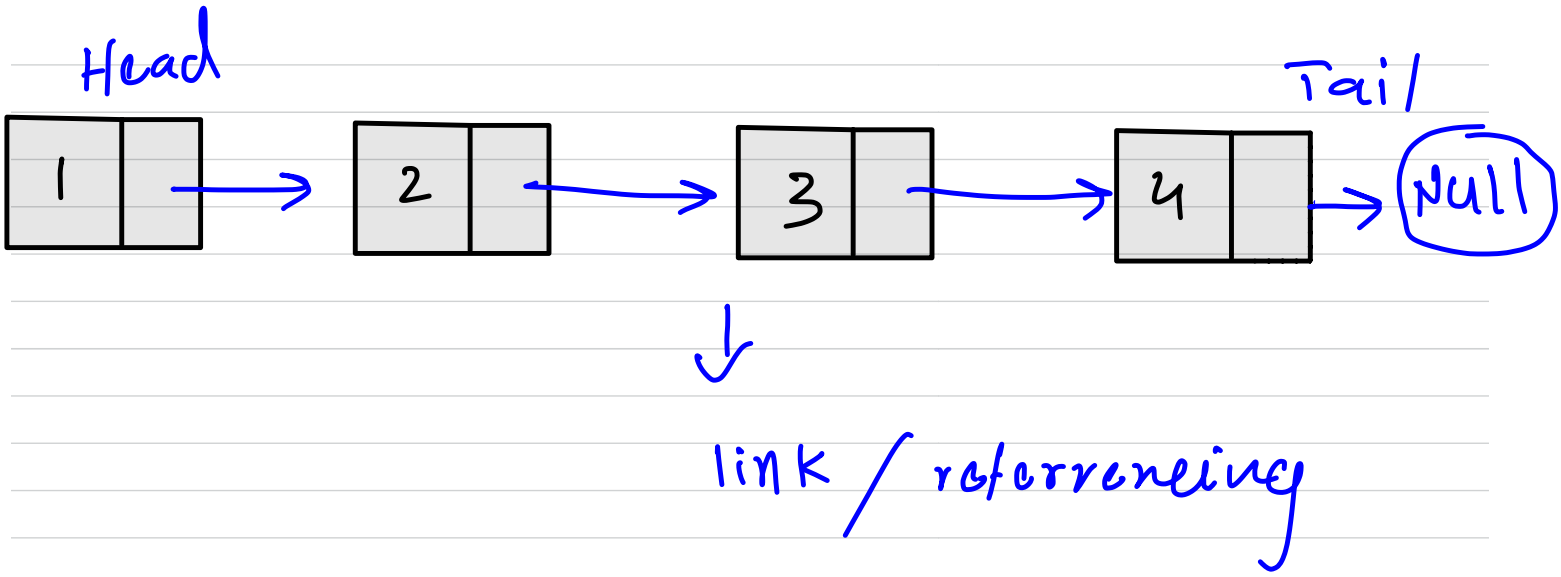
```
class Node
{
    int data;
    Node next;
    public Node (int data)
    {
        data = data
        next = null;
    }
}
```

* Creating LinkedList

```
Class LinkedList
{
    Node head;
    int size;

    public LinkedList()
    {
        head = null
        size = 0
    }

    ----
    ----
}
```




```
function Display()
{
    Node p = head;
    While (p != null)
        { print(p.data + " --> ");
          p = p.next;
        }
    println();
}
```



link / referrencing

INSERT:

1 - Create a new node
2 - Assign data to next field
3 - Assign the head
4 - O(1)

DELET:

1 - Assign a temp variable
2 - Assign new head
3 - Return temp variable
4 - O(1)

ITERATE:

1 - Assign a current node
2 - make a while loop
3 - check current node is null
4

Adding element at the beginning:

```
function addFirst(e)
{   newest: Node(e, null);
    if isEmpty()
        head = newest;
        tail = newest;
    else
        newest.next = head
        head newest;
    count++;
```

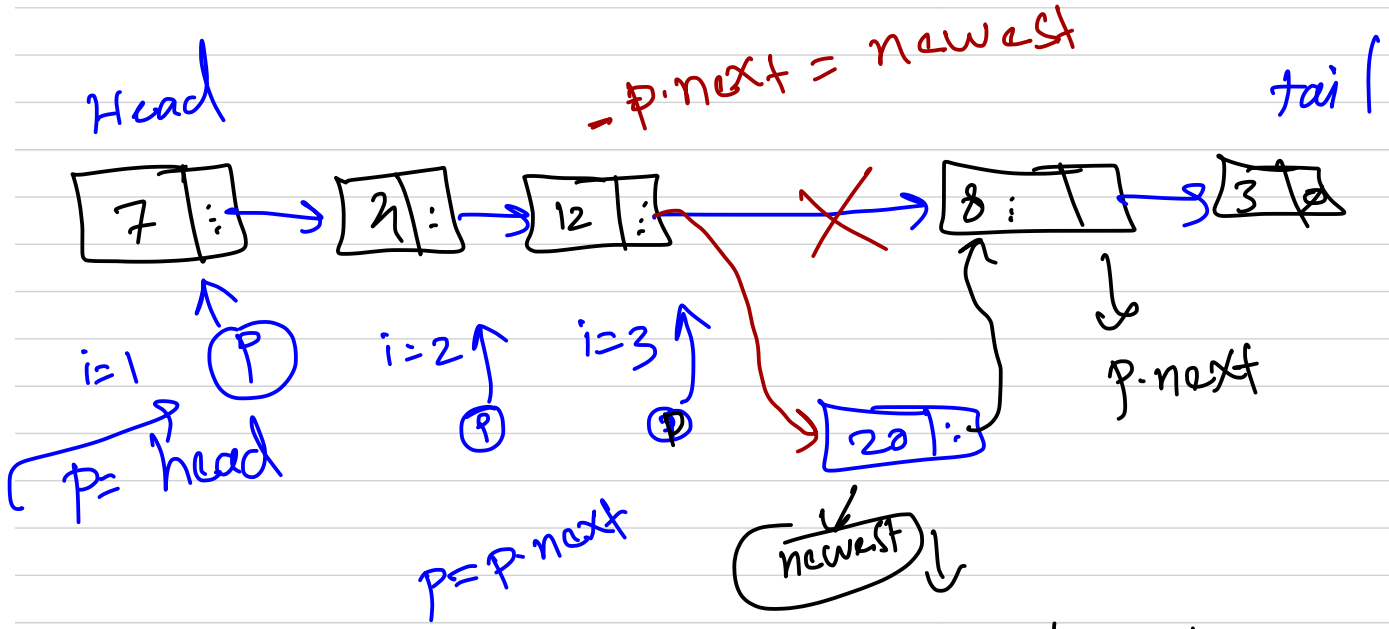
Inserting element anywhere:

```
function addAny(e, position)
{   newest= Node(e, null)
    p= head                    ——→ traverse the list
    i= 1
    while(i < position-1)
    {  p=p.next
    }   i= i+1
```

O(n)

newest = p.next;

p.next = newest; count ++;



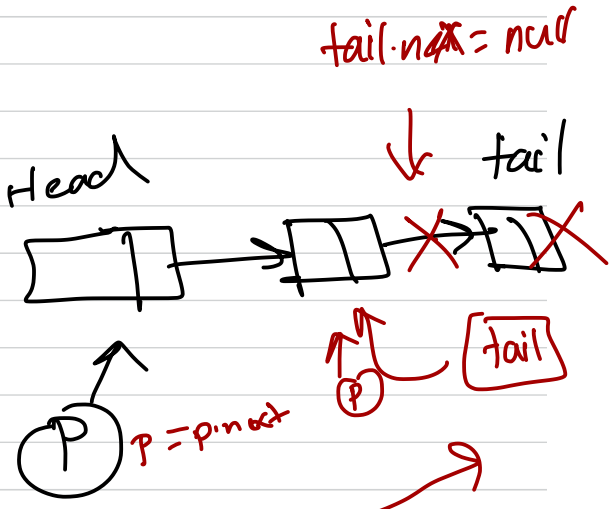
- newest.next = p.next

- Deleting element at the beginning of list:

```
function removeFirst( )
{     if ( isEmpty() )
          print( list is empty);
          return.
      e= head.element
      head= head.next
      size--;
      if isEmpty()
          tail= Null
}     return e
```

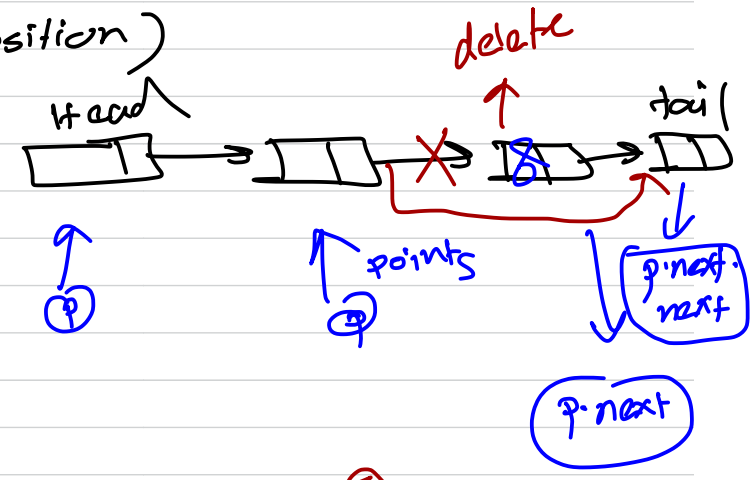
Delete element at End of list:

```
function removeLast ()                    O(n)
    if isempty ()
    { print ("List is empty");
      return;
    }
    p = head;
    i = 1
    while i < length()-1
            p = p.next
            i++;
    tail = p
    p = p.next
    e = p.element
    tail.next = null
      Size --;
    return e;
}
```



Delete element in any position

```
function removeAny ( position)
{
    p= head
    i= 1
    while i<position -1
        p=p.next
        i= i+1
    e= p.next.element
    p.next = p.next.next
    size --;
    return e;
}
```



O(n)

p.next = p.next.next ⇒ creates link to the address of the node after the node we delete.

Search element in list-

* starts at head.

```
function search ( key)
{
    p = head
    index = 0
    while p
    { if p.element == key
          return index;
      p = p.next
      index ++;
    }
    return -1;  ~> if the key doesn't exist
}
```

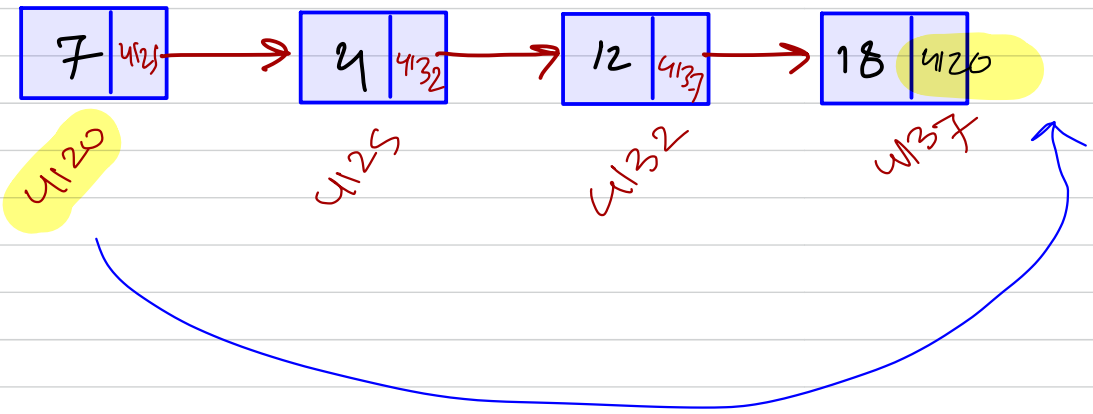
O(n)

Inserting Elements in Sorted order

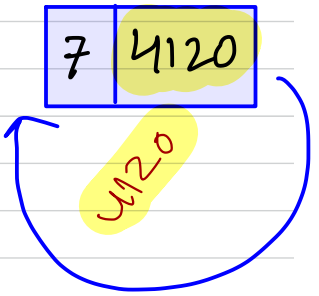
```
function insertSorted ( int e )
{
    Node newest = new Node (e, null)
    if isEmpty()
        head = newest;
    else {
        Node p = head;
        Node q = head;
        While ( p != null && p.element <e)
        {   q=p;
            p = p.next;
        }
        if (p == head)
        {   newest.next = head;
            head = newest;
        }
        else
        {   newest.next = q.next;
            q.next = newest;
        }
    }
    size ++;
}
```

Circular LinkedList

* is a linkedlist where the tail node points back to the head node.

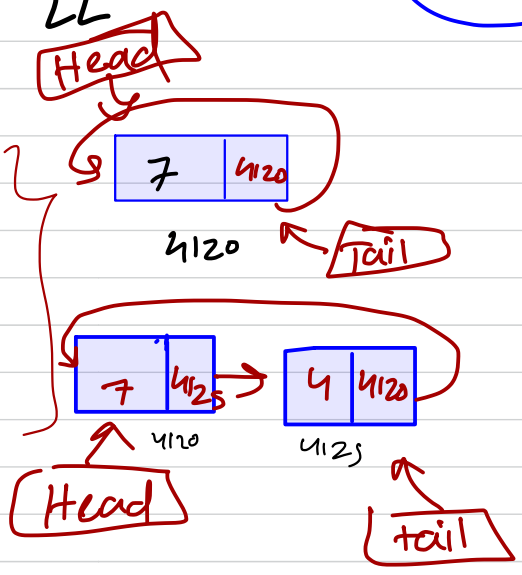


Circular linkedlist: 1 node



Create a Circular LL

```
function addLast (e)
{
    newest = new node (e, null);
    if isEmpt()
        newest.next = newest
        head = newest
    else
        newest.next = tail.next;
        tail.next = newest

    tail = newest
    size ++
}
```



O(1)

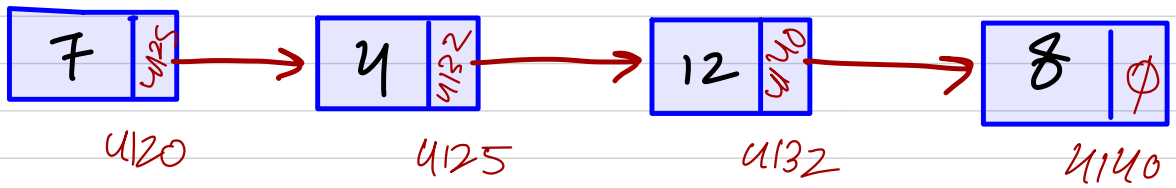
Traversing a Circular LL

```
function Display()
{   p = head          (1)
    i = 0             (1)
    while i < length()   (n)
    {   print(p.element)   (n)
        p = p.next
        i++
    }
}
```

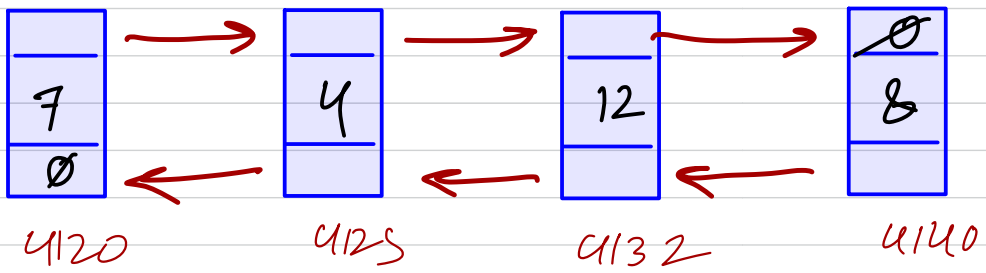
O(n)

Doubly LinkedList

* Single Linked List :



* Doubly Linked List :



* Creating Node of Doubly LL

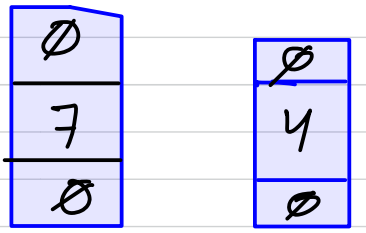
```
Class Node {
    int element; Node next; Node prev;
    public Node (int e, Node n, Node p)
    {
        element = e
        next    = n
        prev    = p
    }
}
```

```
n1 = Node(7, null, null);
n2 = Node(4, null, null);
```

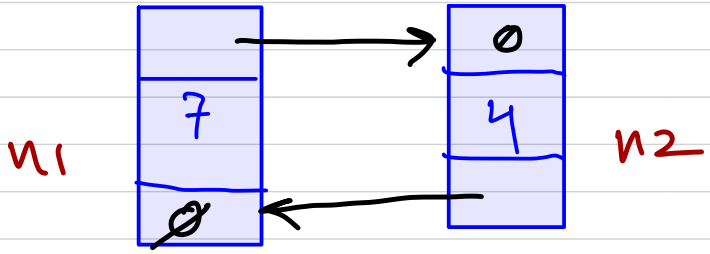
n1. element 7

n1. next ∅

n1. prev ∅



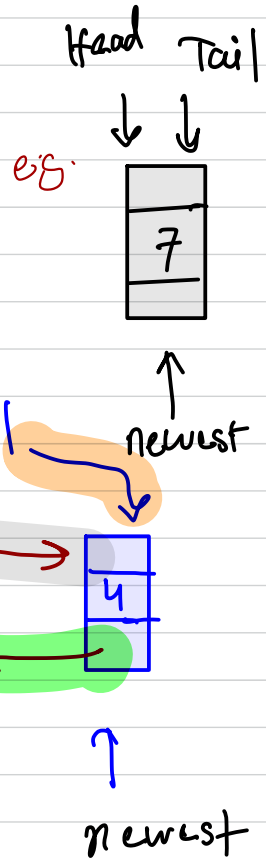

```
n1.next = n2;
n2.prev = n1;
```

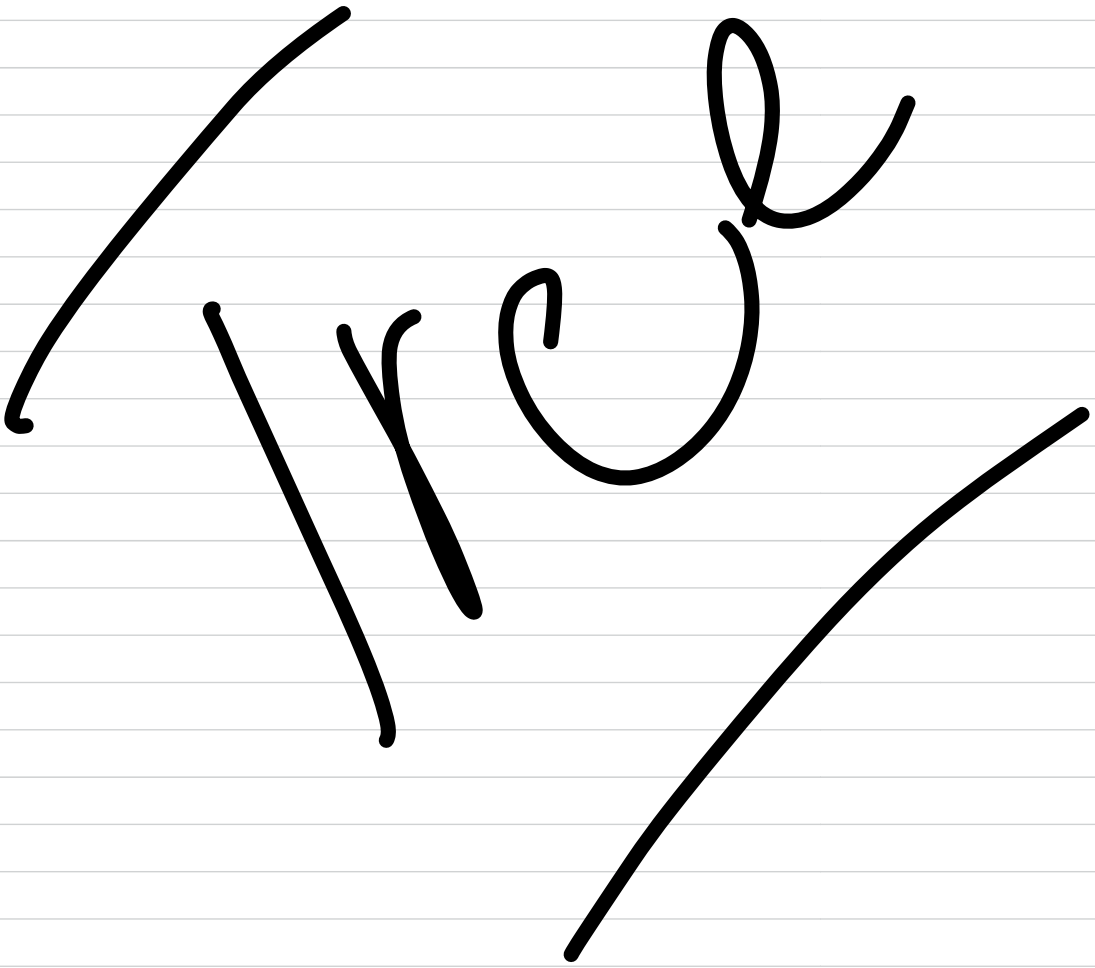


Doubly LinkedList

```
function addLast(e)
{   newest = Node(e, null, null);
    if (isEmpty())
        head = newest;
        tail = newest;
    else
        tail.next = newest;
        newest.prev = tail;
        tail = newest;
    sizet++
}
```

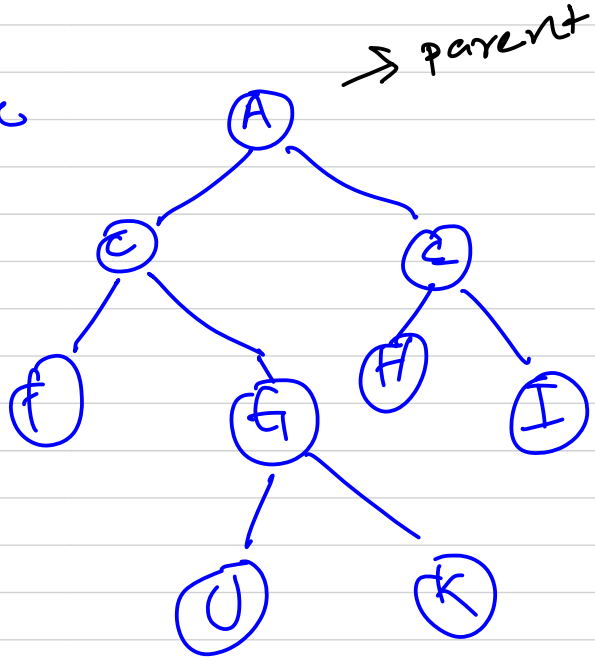
O(1)



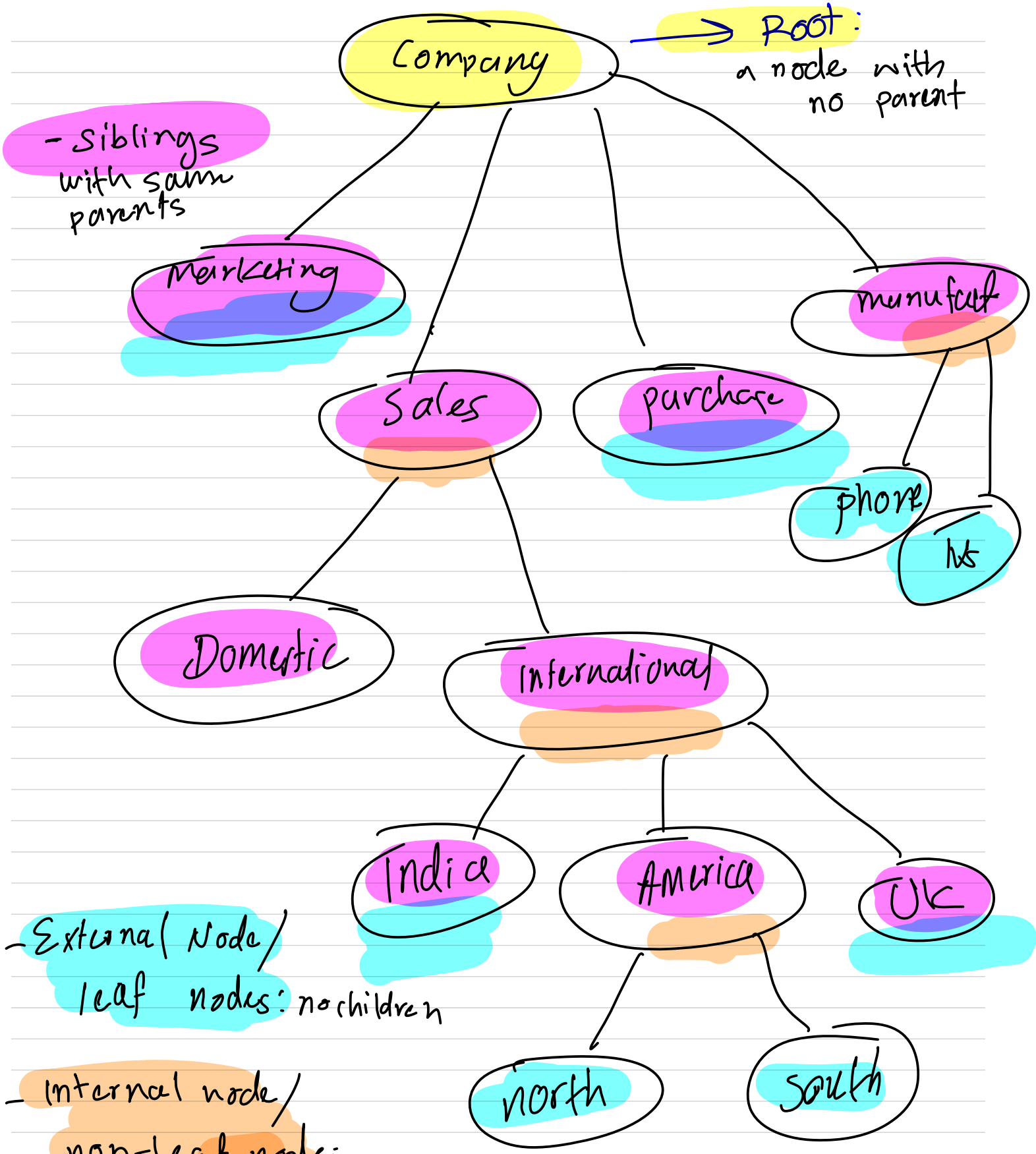


Binary Trees

* is a non-linear data structure.
* is an abstract data type that stores data Hierarchly.

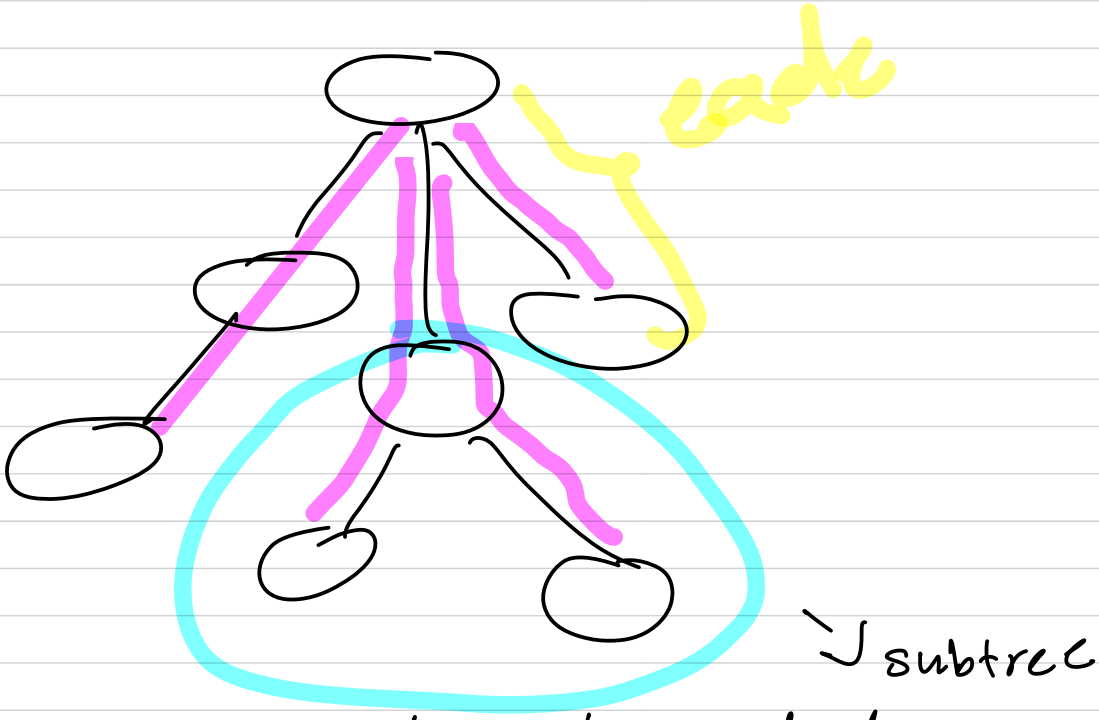


* Set of Node
* Parent-child relationship through edges
  - number of nodes: n
  - number of edges: n-1
* Tree can be empty


Company
→ Root:
a node with no parent
- Siblings with same parents
Marketing
Sales
purchase
manufact
phone
Domestic
International
India
America
UK
north
south
- External Node / leaf nodes: no children
- Internal node / non-leaf node:
with 1 or more child

Edge - an edge of a tree is a pair of nodes (u, v) such that u is parent of v

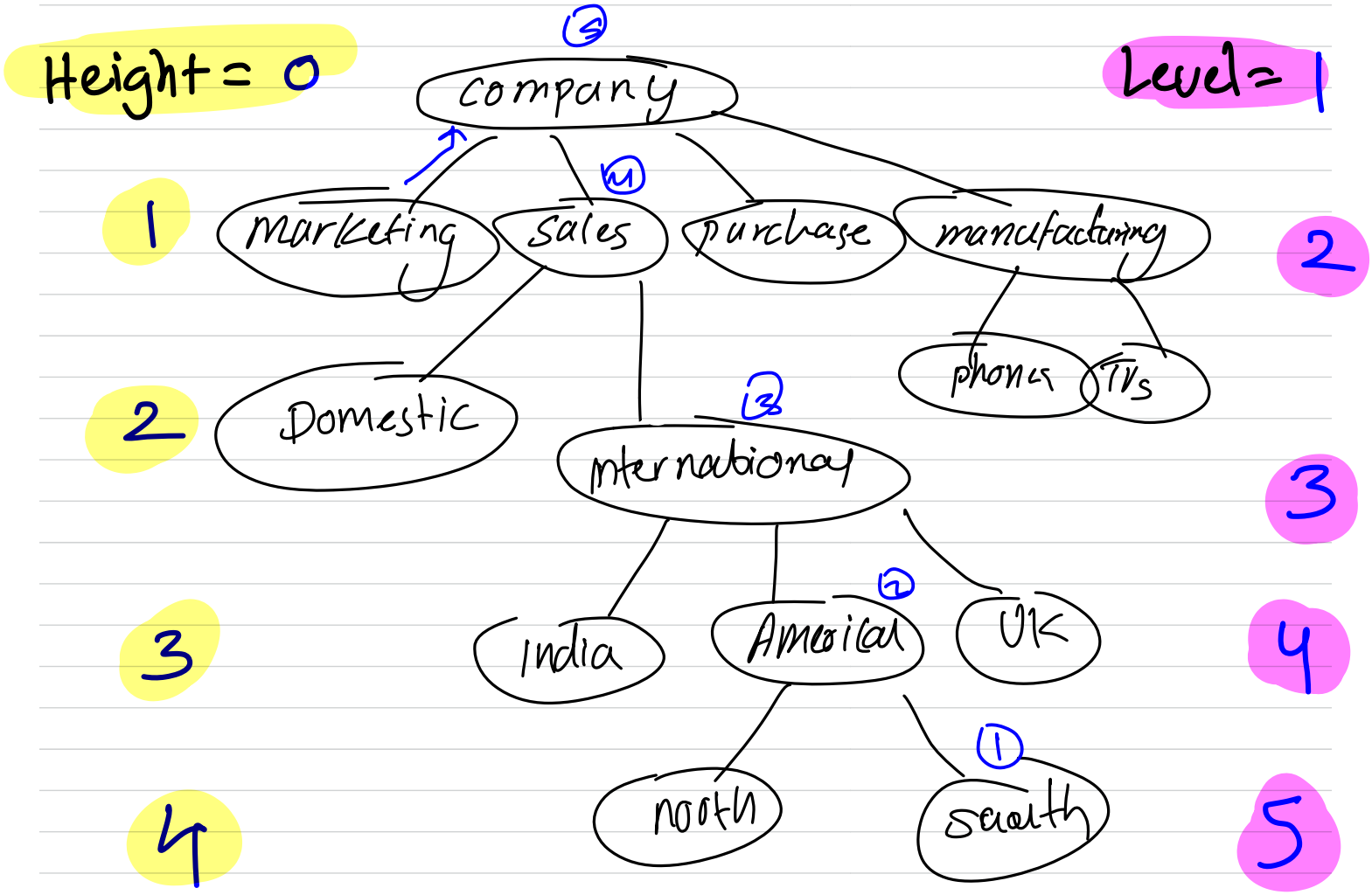
Path - Sequence of nodes such that two consecutive nodes forms an edge.



Subtree - any node with its children

Forest - a collection of trees

Height & levels of tree



no. Height — represents the number of Edges. eg → at height = 1 there is 1 edges from any node to root.

no. Level — the number of nodes

eg → at level 5, there is 5 nodes to root.

Degrees of Node & Tree

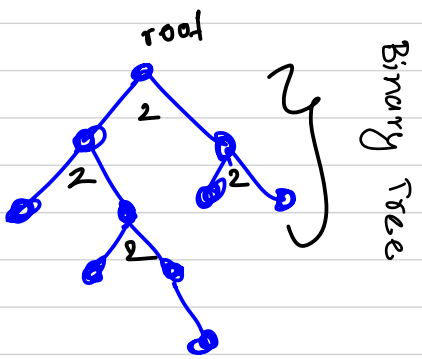
- Degree of node — the number of children of the node. e.g. degree of company = 4
  * degree of the leaf node is zero.

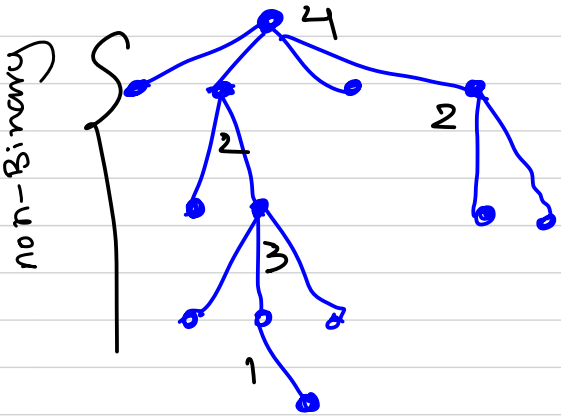
- Degree of Tree — the maximum degree of the node. eg → company — has the max degree

Binary Trees & its properties

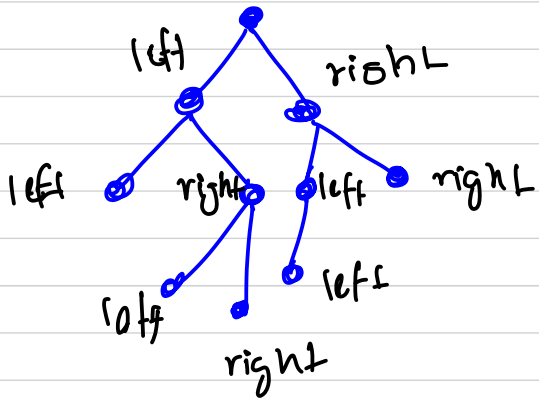
Binary Trees :

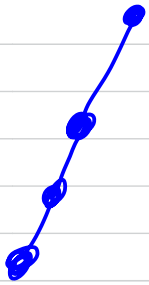
* Every node has at most two children
* Every child node is labelled as left child or right child
* left child precedes right child in order of nodes.



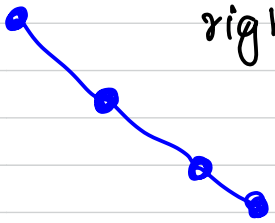


Binary Trees deg max = 2



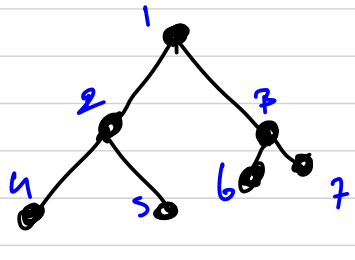


left-skewed binary tree



right-skewed binary tree

— maximum number of nodes in a binary tree of height h:



h = 0, nodes = 1

h = 1, nodes = 2

h = 2, nodes = 7

* Formula for finding number of nodes in binary tree.

$$2^{h+1} - 1 = \text{no. Nodes}$$

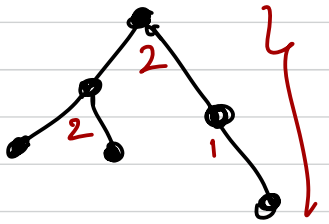
eg → h = 2

$$2^{2+1} - 1 = 2^3 - 1$$

$$= 8 - 1 = 7$$

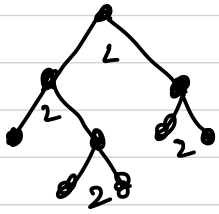
Proper Binary Tree :

- Each node has either 0 or 2 children.



✓ Binary Tree : 0 or 1 or 2 children

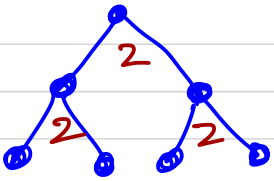
✗ proper Binary Tree: 0 or 2 children



proper Binary tree

Full Binary Tree:

- Every internal node has exactly 2 children and all leaf nodes are at same level.

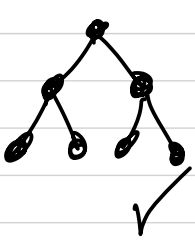


leaf nodes

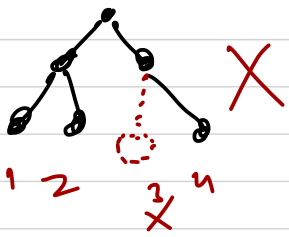
$$H = 2^{h+1} - 1$$

Complete Binary Tree:

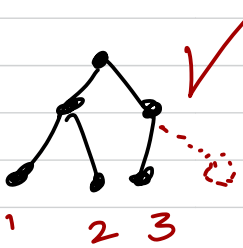
- where nodes at each level are numbered from left to right without any gap.



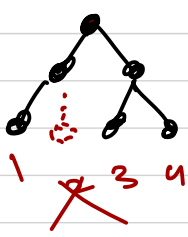
✓



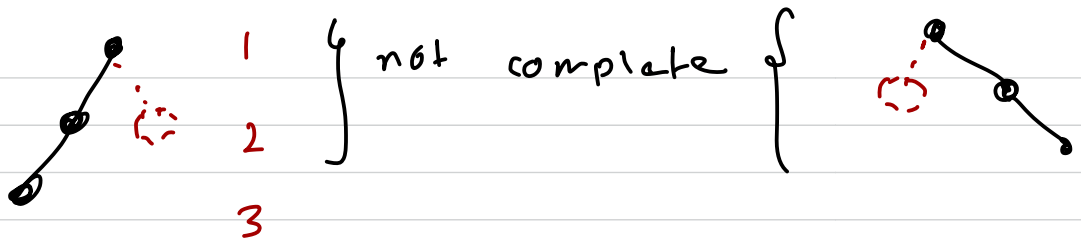
✗



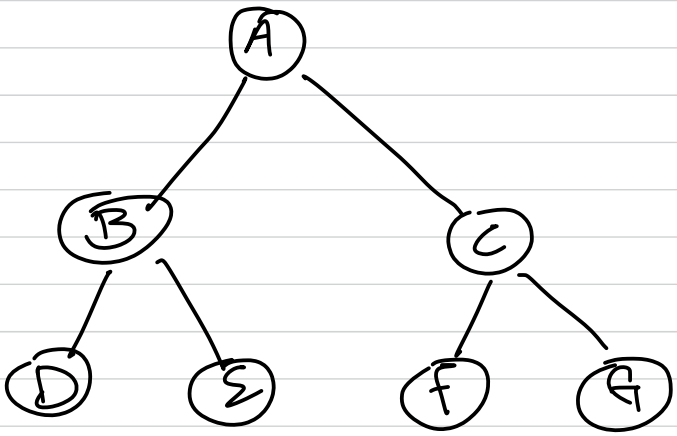
✓



✗



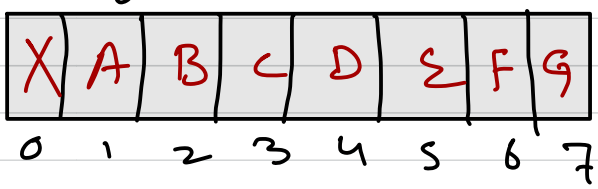
Binary Tree representation - Array Based



Array -> size = $2^{h+1} - 1$

$= 7$

Array =



Element	Index	left child	right child
A	1	2	3
B	2	4	5
C	3	6	7
	i	i x 2	i x 2 + 1

- To get the parent of a node:

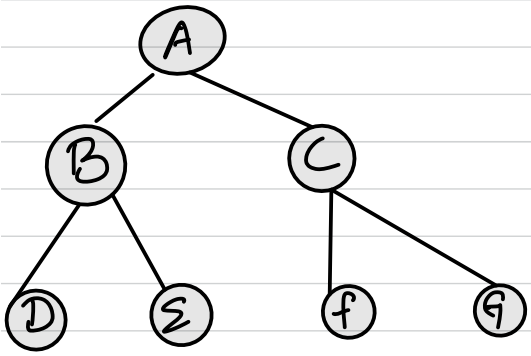
$\lfloor i/2 \rfloor$ → eg: D ⇒ [4]

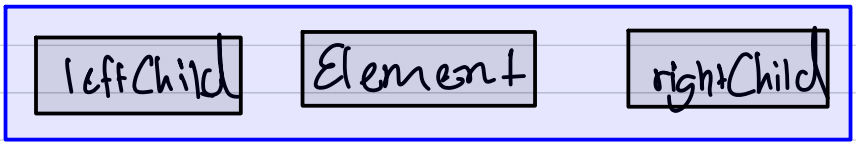
D $\lfloor 4/2 \rfloor = [2]$ → at index 2 is (B)

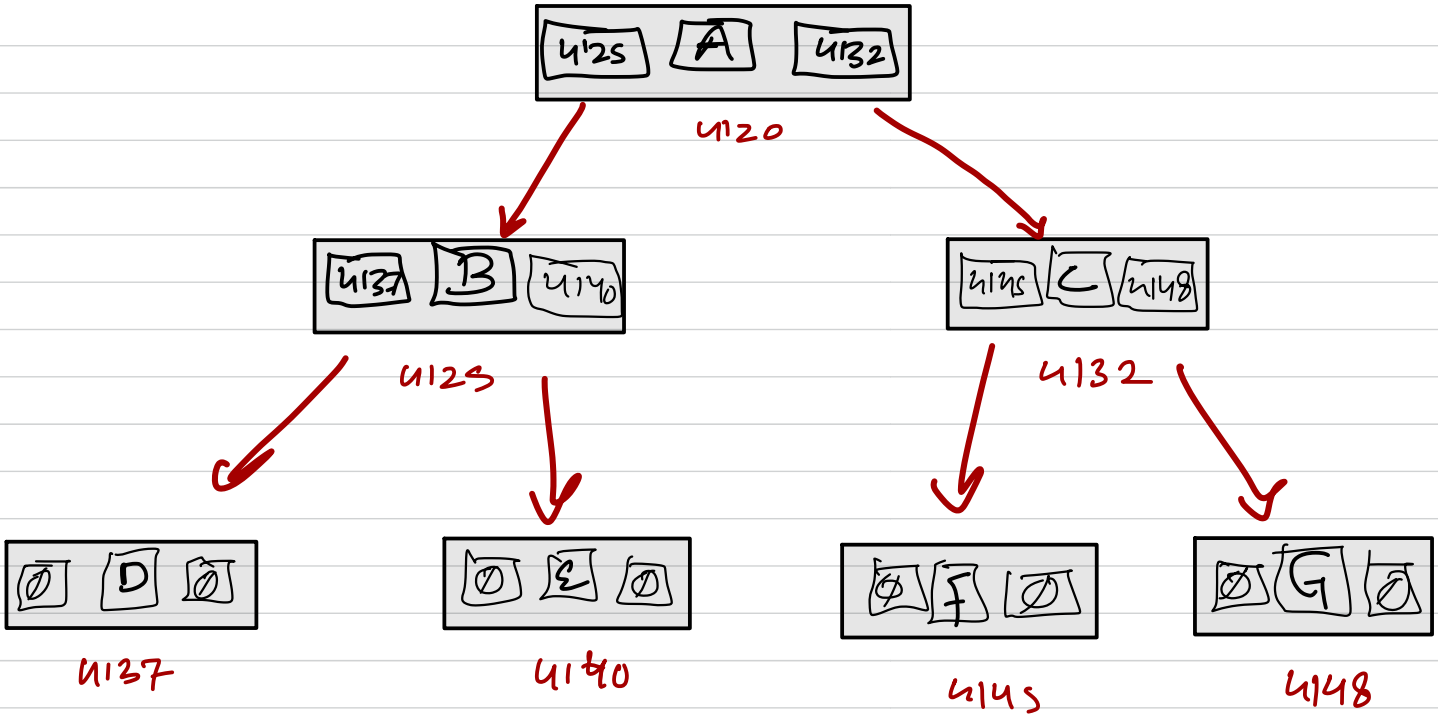
E $\lfloor 5/2 \rfloor = 2.5 \rightarrow 2$ → (B)

to get the children of the node at index i

Binary Tree representation – Linked Based

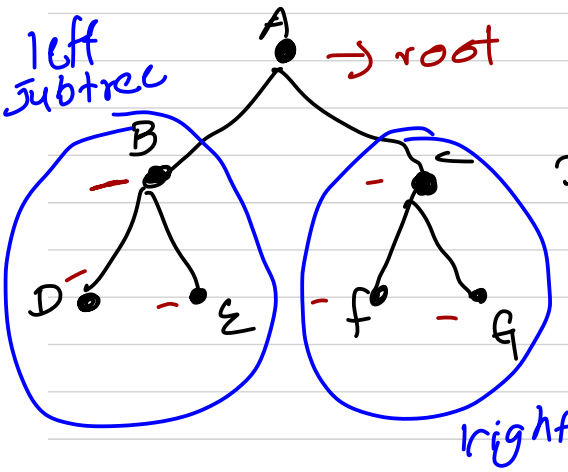






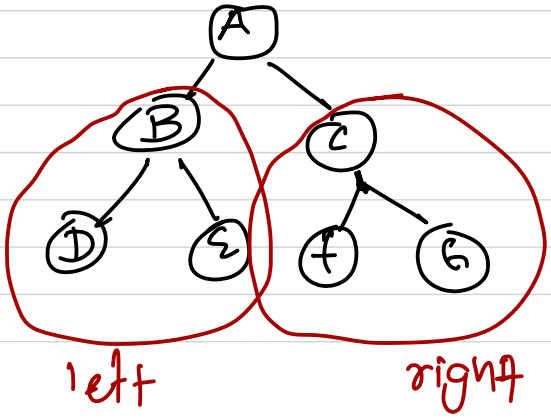
Binary Trees Traversal

- **Pre-order :**
  - * Visit Root
  - * Visit left subtree recursively pre-order
  - * Visit right subtree recursively pre-order



Pre-order Traversal: A, B, D, E, C, F, G

- **In-Order :**
  - * Visit left subtree recursively Inorder
  - * Visit root
  - * Visit right subtree recursively Inorder

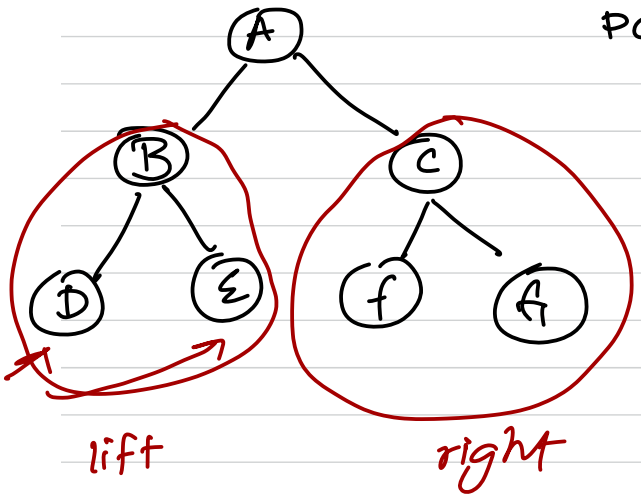


In-order traversal:

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

• Post-order:

* Visit left subtree recursively postorder
* visit right subtree recursively postorder
* visit root

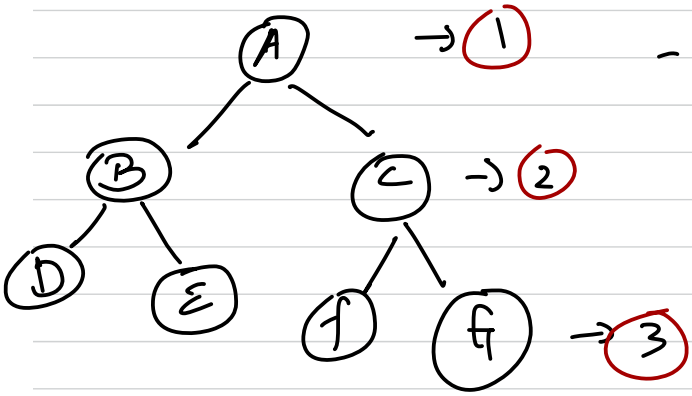


post-order Traversal:

D, E, B, f, G, C, A

• Level-Order:

* visit nodes level by level from Top to Bottom
* within Level, visit nodes from left to right.



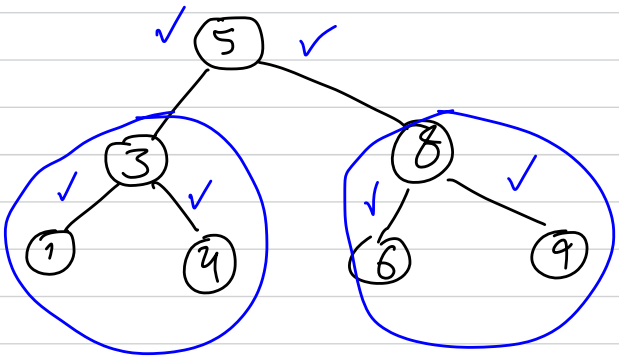
- Level-order traversal:

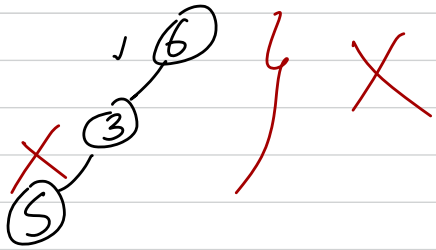
A, B, C, D, E, f, G

Binary Search Tree

* is a binary tree where for each node all the elements in its left side is smaller than the node and all elements in the right hand side are greater than that node.

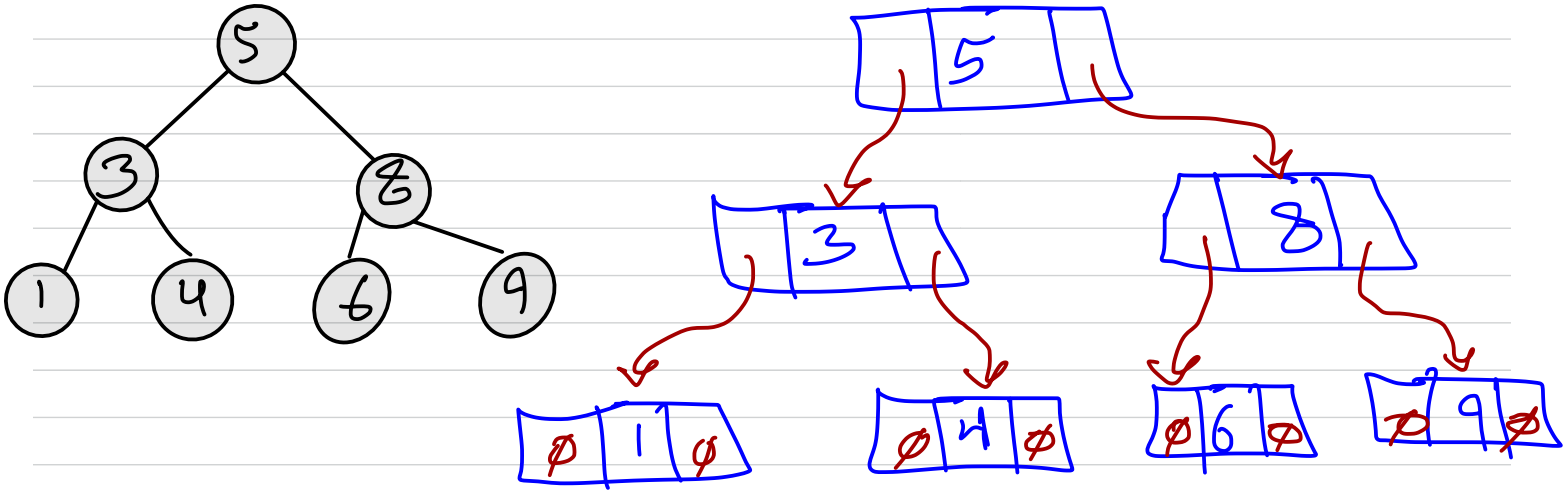
eg-





* Every node has a key
* Keys in left sub-tree of node are smaller than the key in the node.
* Keys in the right sub-tree of node are greater than the key in the node
* Left and right sub-tree are also binary search tree
* Binary search trees will not have duplicate elements. Or nodes.

Binary Search Tree – Linked based



Searching

Iterative –

```
function Search (key)
{   Node troot = root;
    while (troot)
    {   if key == troot.element
            return True;
        else if (key < troot.element)
            troot = troot.left;
        else if (key > troot.element)
            troot = troot.right;
        else
            return False
    }
}
```

Recursive –

```
function rSearch ( troot, key )
{
    if troot
    {
        if (key == troot.element)
            return True
        else if ( key < troot.element)
            return rSearch(troot.left, key);
        else if ( key > troot.element )
            return rSearch (troot.right, key);
    }
    else
        return false;
}
```

Binary Search Trees – Insertion

Iterative:

```
function insert ( troot , key )
{
    temp = null
    while (troot != null)
    {
        temp = troot
        if (e == troot.element) return
        else if (e < troot.element) troot = troot.left
        else if (e > troot.element) troot = troot.right
    }
    n = new Node(e)
    if (troot != null)
        { if (e < temp.element) temp.left = n
        } else                  temp.right = n
    else
        root = n
}
```

Recursive:

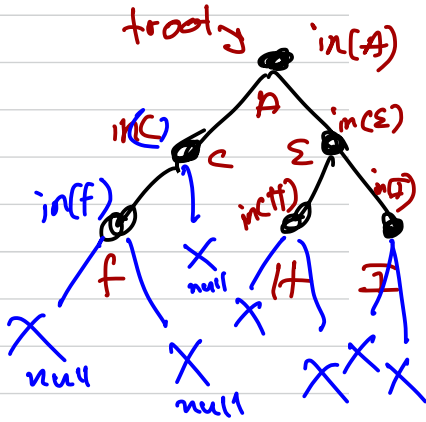
```
function rInsert ( troot, e )
{
    if (troot != null)
    {
        if (e < troot.element)
            troot.left = rinsert(troot.left, e)
        else if (e > troot.element)
        }
            troot.right = rinsert (troot.right, e)
        else
        {
            n = node(e)
            troot = n
        }
}
return troot;
```

Binary Tree Traversal

Inorder

1)

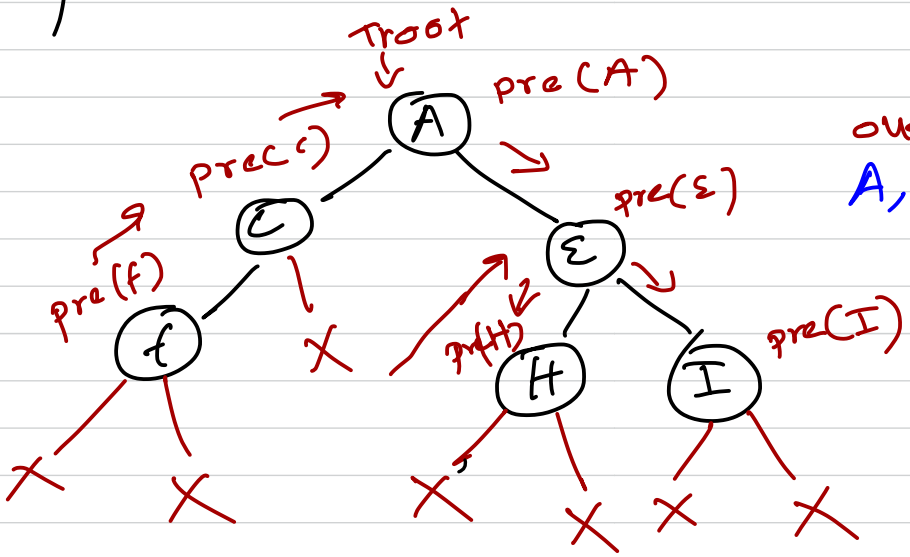
```
inOrder ( troot )
{   if ( troot != null )
        inOrder ( troot.left )
        print ( troot.element )
    }   inOrder ( troot.right )
```



output = f, C, A, H, Σ, I

2) Pre-order

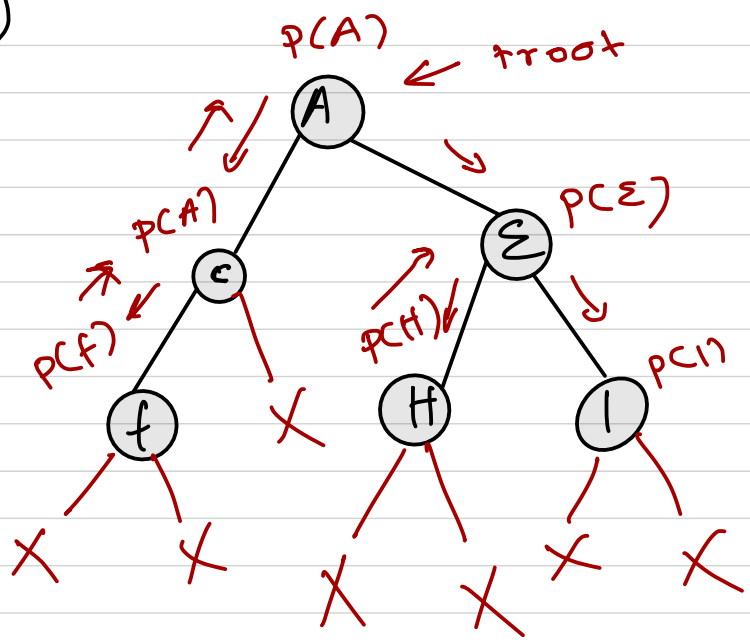
```
preOrder ( troot )
{   if (troot != null )
    {   print ( troot.element )
        preOrder ( troot.left )
        preOrder ( troot.right )
    }
}
```



output =
A, C, f, Σ, H, I

3) post-order

```
postorder (troot)
{   if (troot != null)
    {
        postorder (troot.left)
        postorder (troot.right)
    }
    print (troot.element)
}
```



output: f, C, H, I, E, A

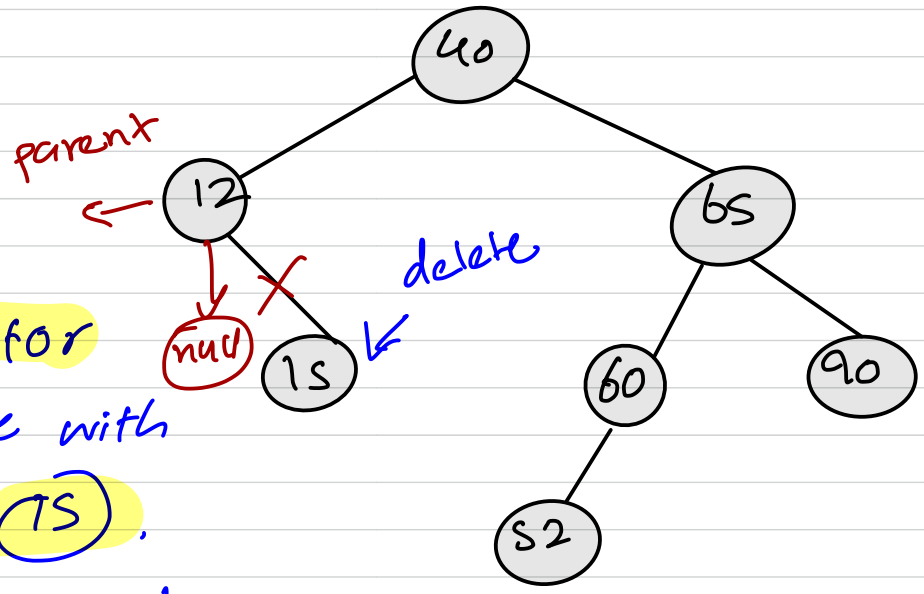
4) Level Order

```
levelorder ()
{   Q = Queue()
    t = root
    print(t.element)
    Q.Enqueue(t)
    while !(Q.isEmpty())
        t = Q.dequeue()
        if (t.left)   print(t.left.element), Q.Enqueue(t.left);
    }   if (t.right)  print(t.right.element), Q.Enqueue(t.right);
```

Binary Search Tree Deletion

- possibilities of Deletion:
  - Leaf node
  - Node with one subtree
  - Node with both subtree

1) Leaf Node

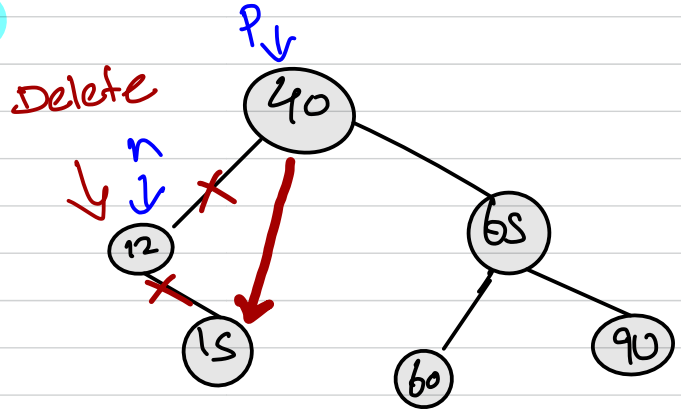


- Delete 15

* Search for the Node with element (15).

then, set the parent node right / left child to null

2) One subtree (L/R)

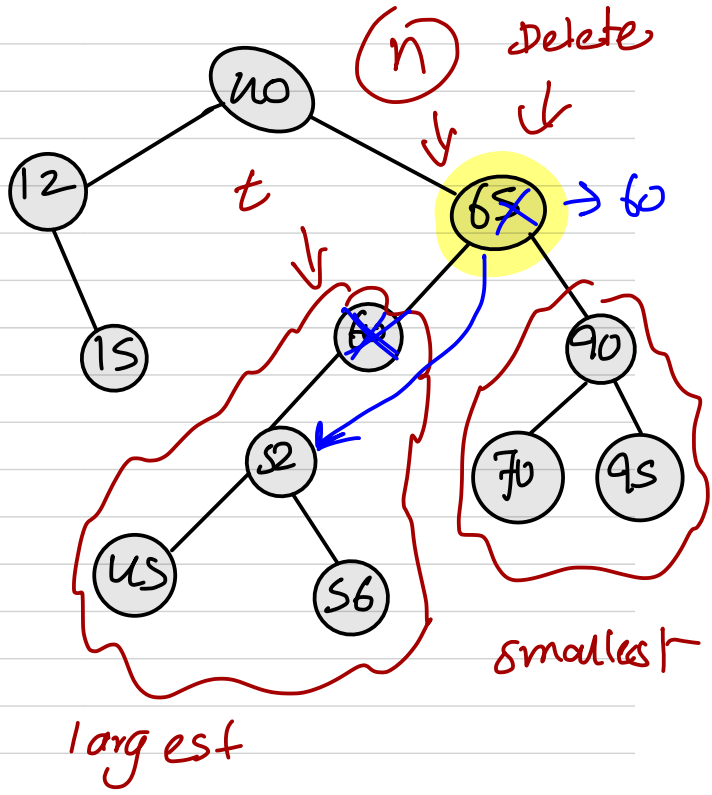


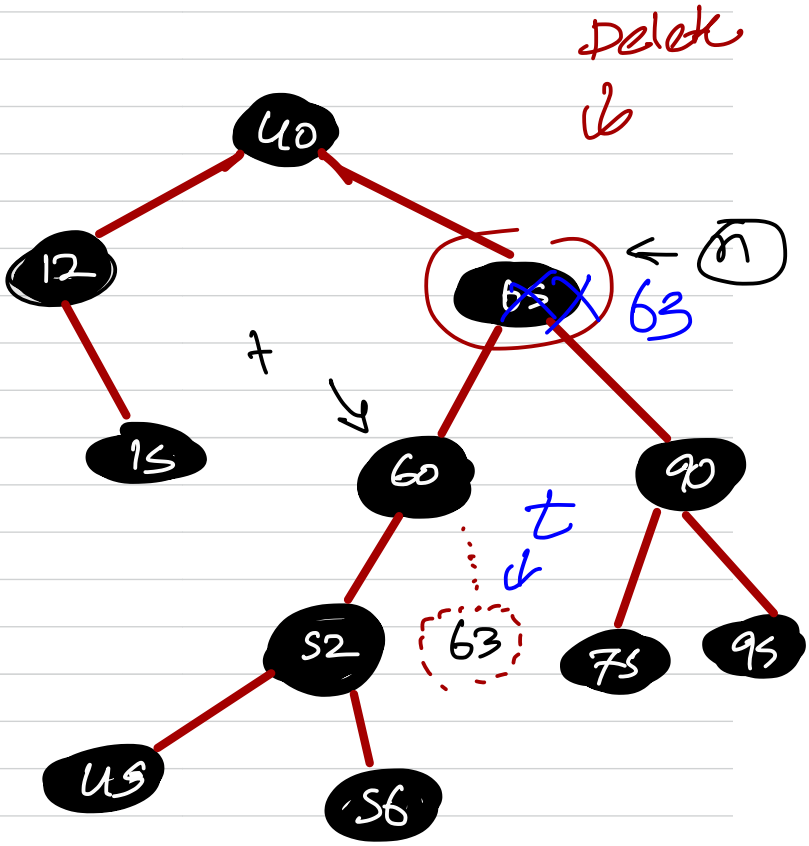
- first, create a link from the root to the (12) right child.

p.left = n.right

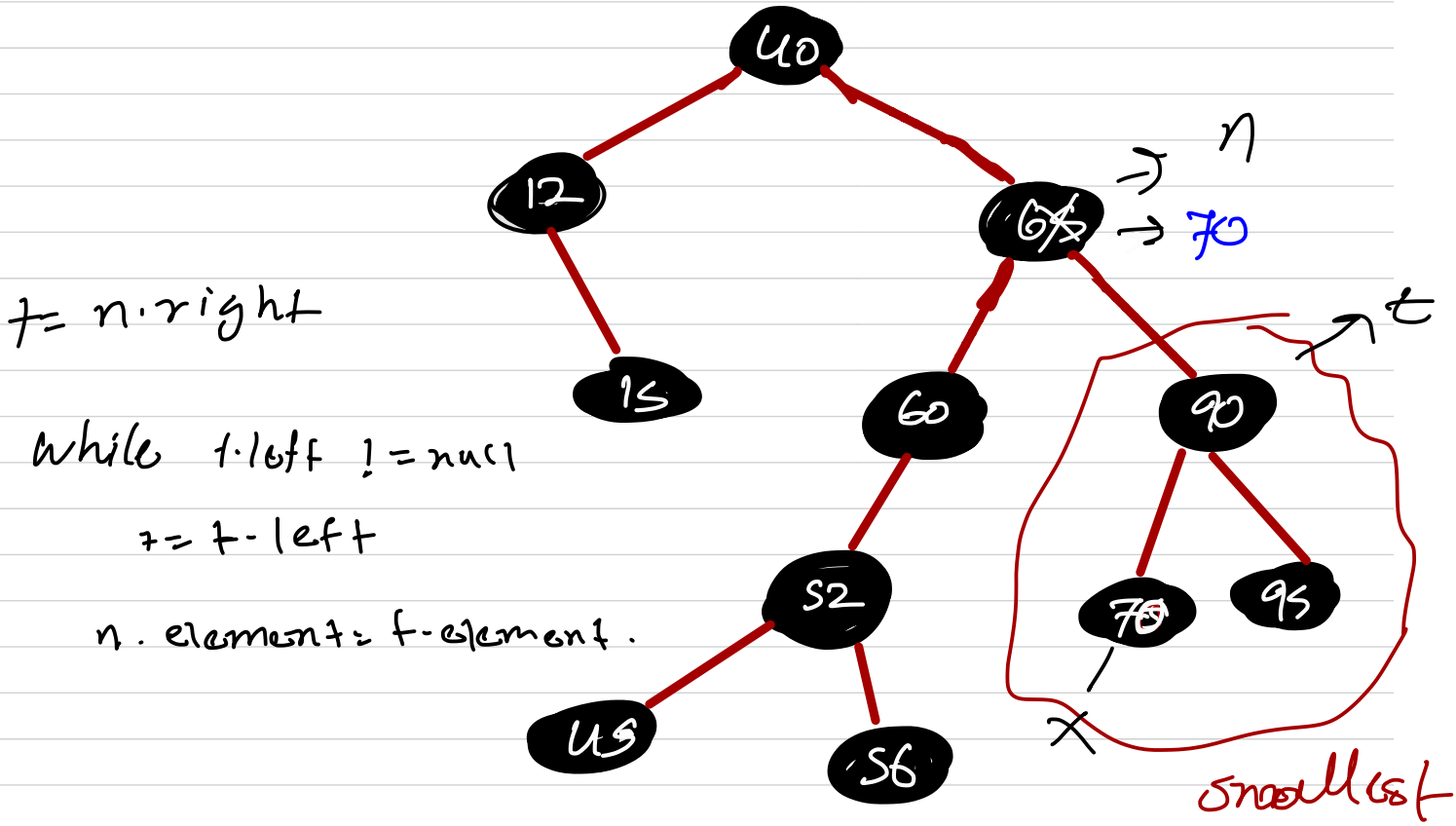
3) Two subtrees (L && R)

```
t = n.left

while (t.right != null)
    { t = t.right
    }

n.element = t.element

n.left = t.left
```






```
t = n.left

while(t.righ != null)
    t = t.right
n.element = t.element
```



- Count the number of nodes

```
Count ( troot )
{   if (troot != nuu)
        x = count(troot.left)
        y = count(troot.right)
    }
    return x+y+1;
    return 0;
```

Finding the Height of Binary Tree

```
height (troot)
{   if (troot != null)
    {   x= height (troot.left);
        y= height (troot.right);
        if (x>y)    return x+1;
        else        return y+1;
    }
    return 0;
}
```

Balance Search Tree

Binary Search Trees:

* Searching_ Insertion, Deletion: Avg case: $O(\log n)$
* " " " : Worst case: $O(n)$ → height of tree
* Worst case: Height is proportional to Number of Nodes → $O(n)$

— Balanced Search Trees —

* Reduce the Height of the BST
* Rotations or restructuring
* modifies the relationship between parent-child
* Balanced Search Tree – AVL Tree
  : Red-black Tree
  : Splay Tree

AVL Tree

Balanced Search Tree –

provides better performance

Binary search tree – Reshape, reduce Hieght

Height-balance property or Balance factor

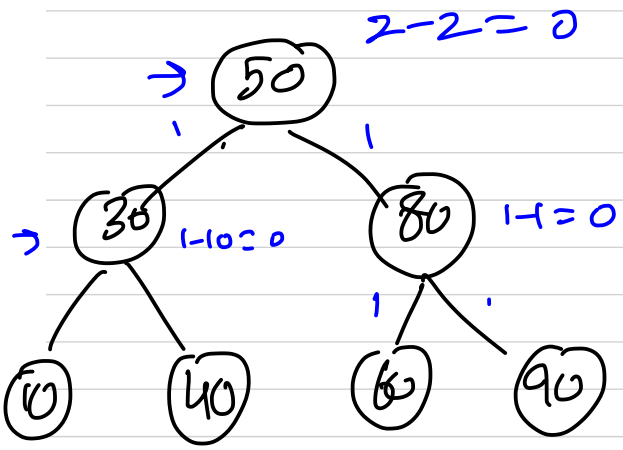
- Height-balance property or balance factor:
  - for every node, height of children differ by at most 1.

- AVL Trees – any binary search tree, that satisfies height balance property.

AVL tree balance factor

Balance factor: 1 0 -1

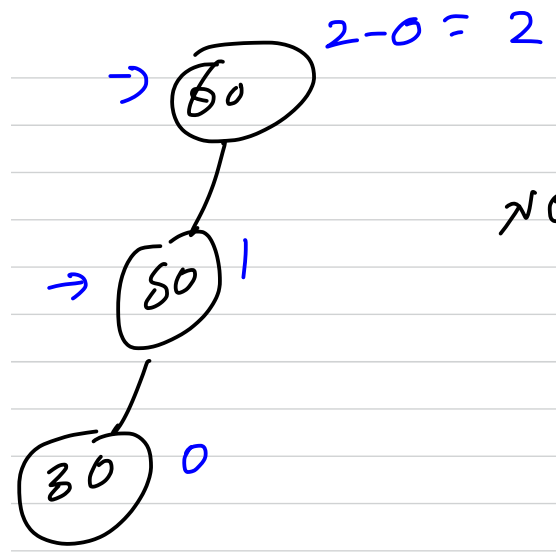
Example:



Balance factor = 0

So, this is an AVL tree.

balanced



Balance f : (2) } unbalanced

not AVL tree

AVL Tree - Rotations

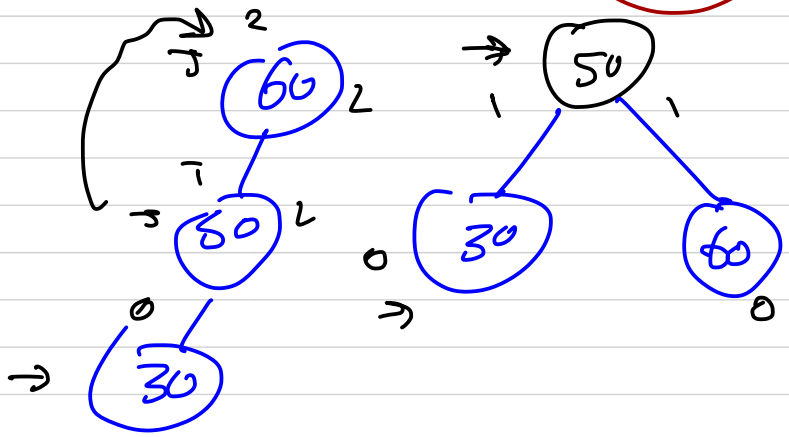
* LL
* RR
* RL
* LR

} Rotations

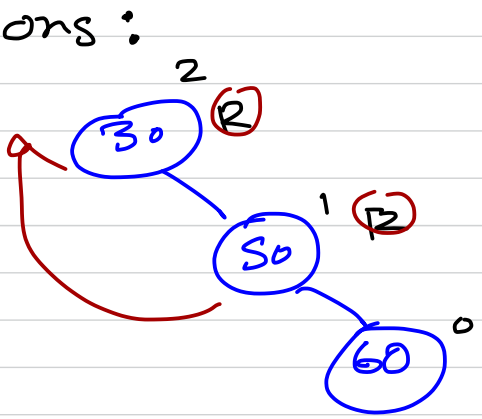
AVL tree balanced

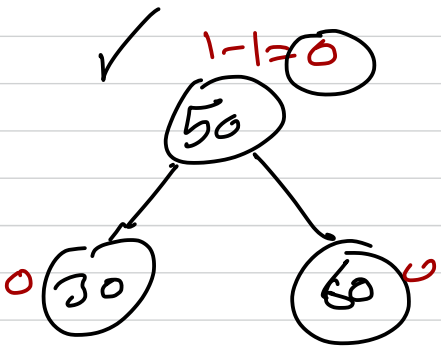
1-1=0

* LL rotation :

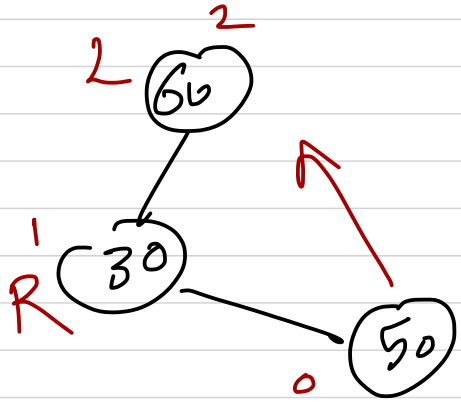


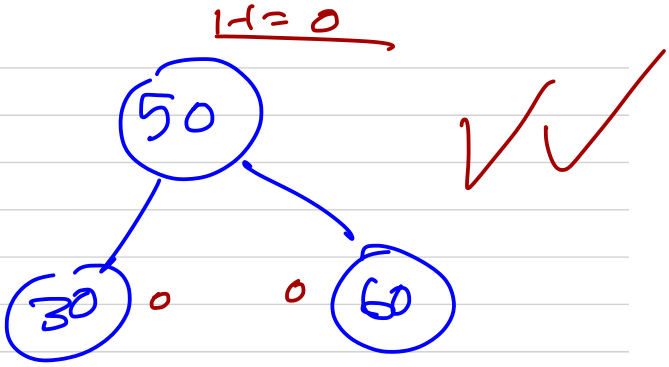
* RR rotations :



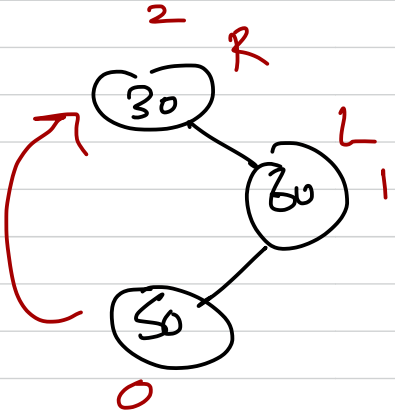


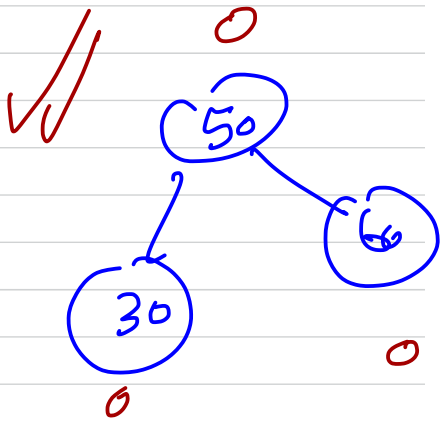
* LR rotation :



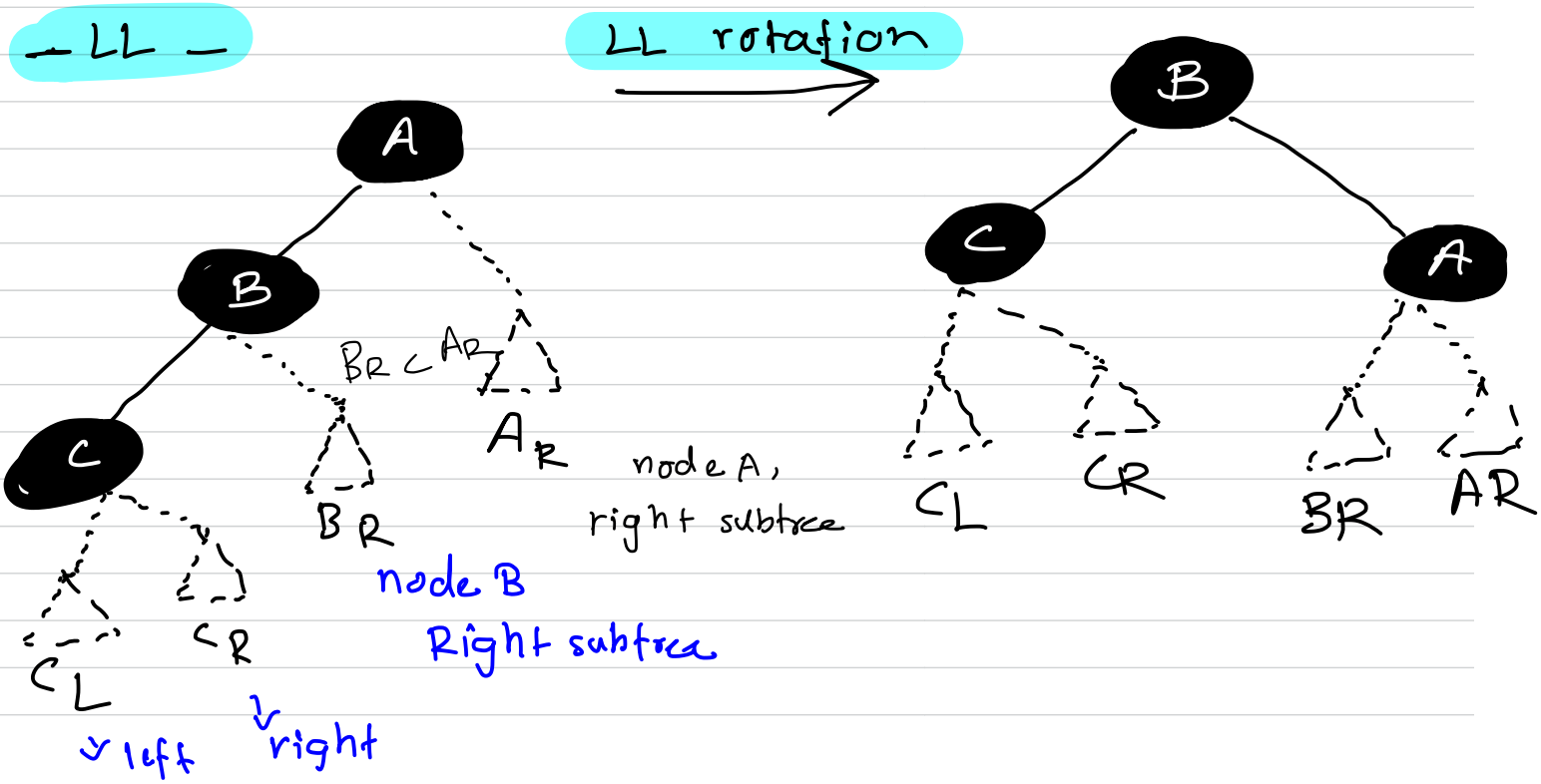


* RL rotation :

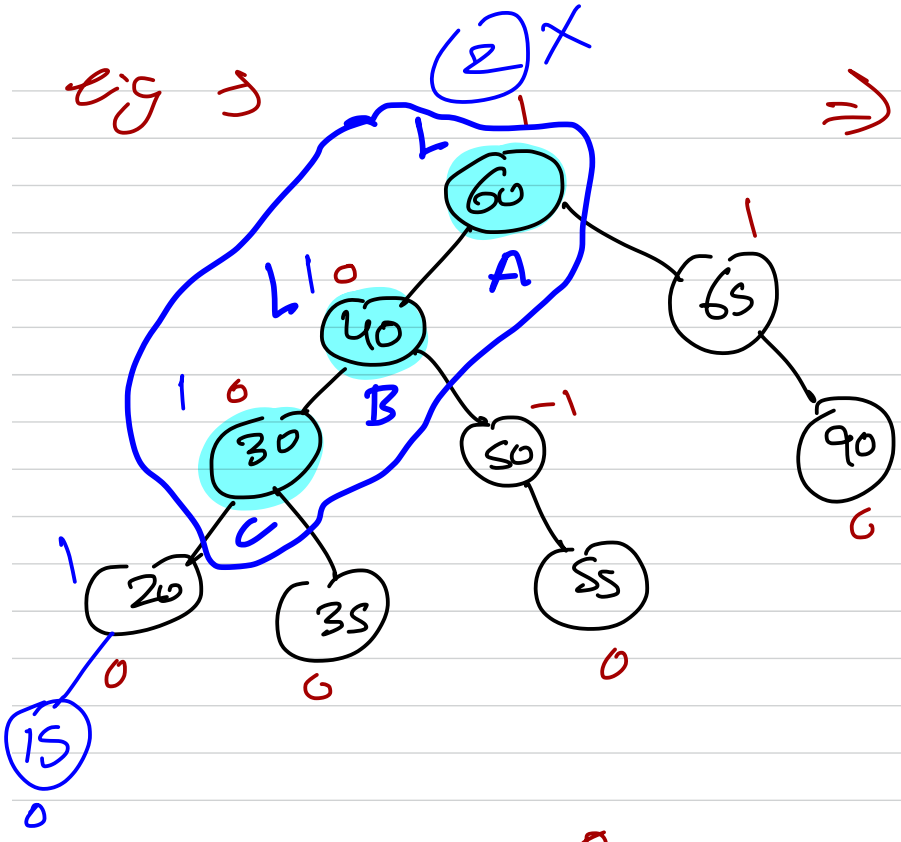




AVL Tree - Rotation

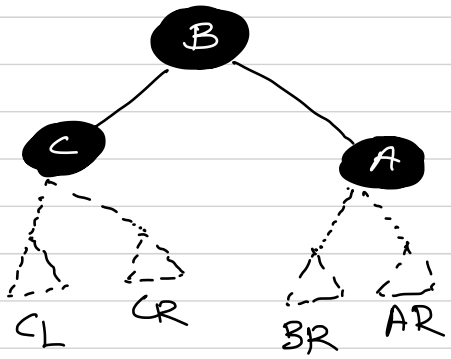


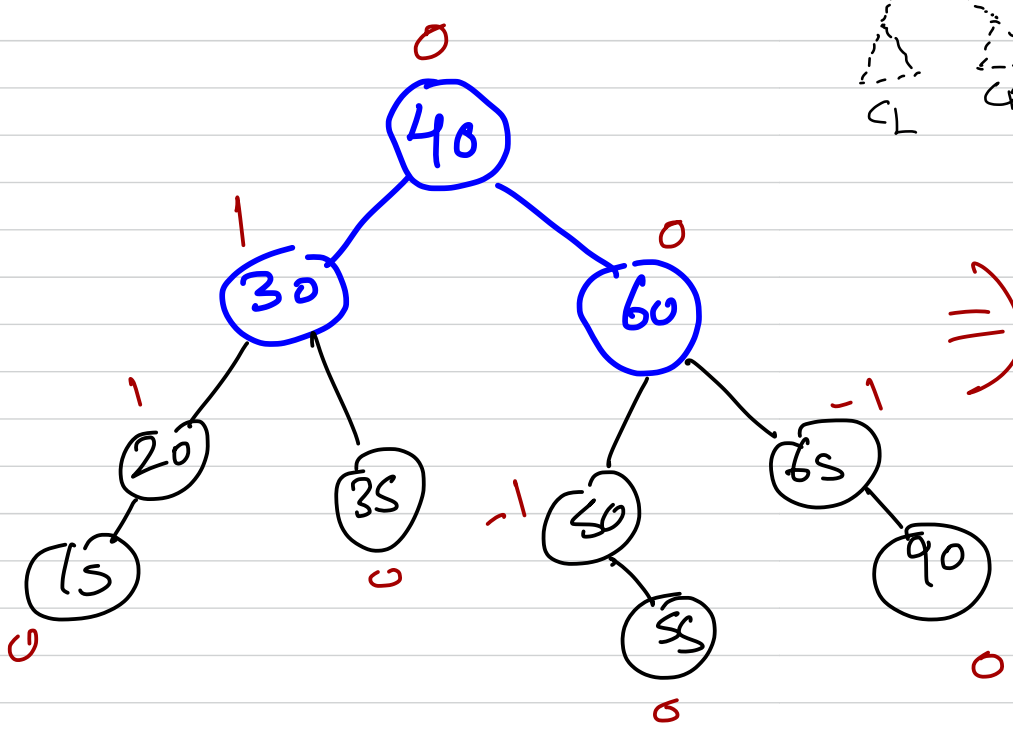
eg →



⇒ Balanced

↳ Add a node 15

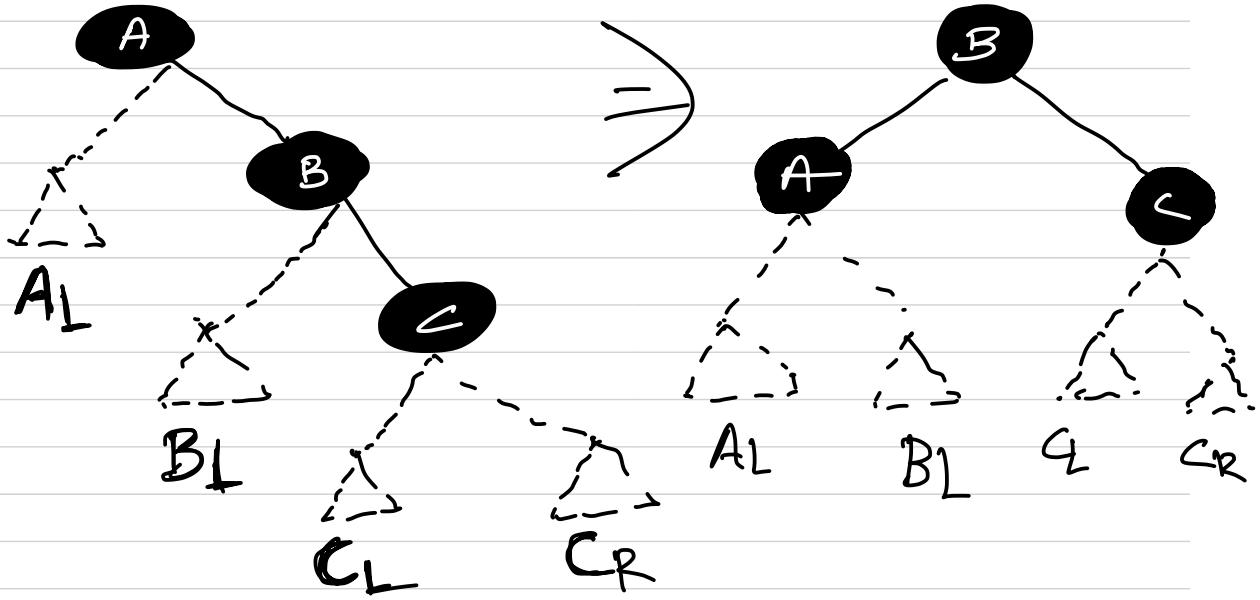




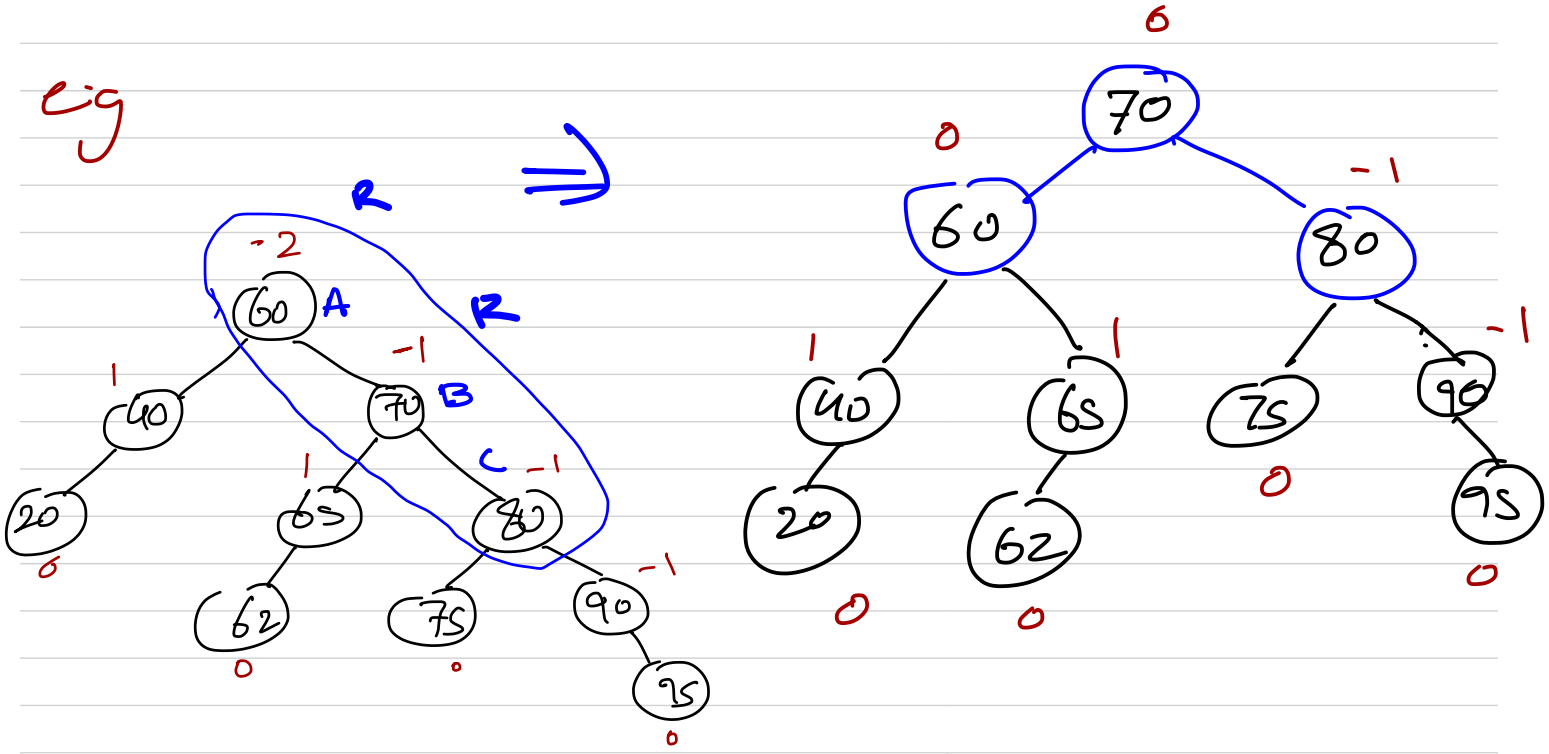
⇒ Balanced AVL tree

BF = 0

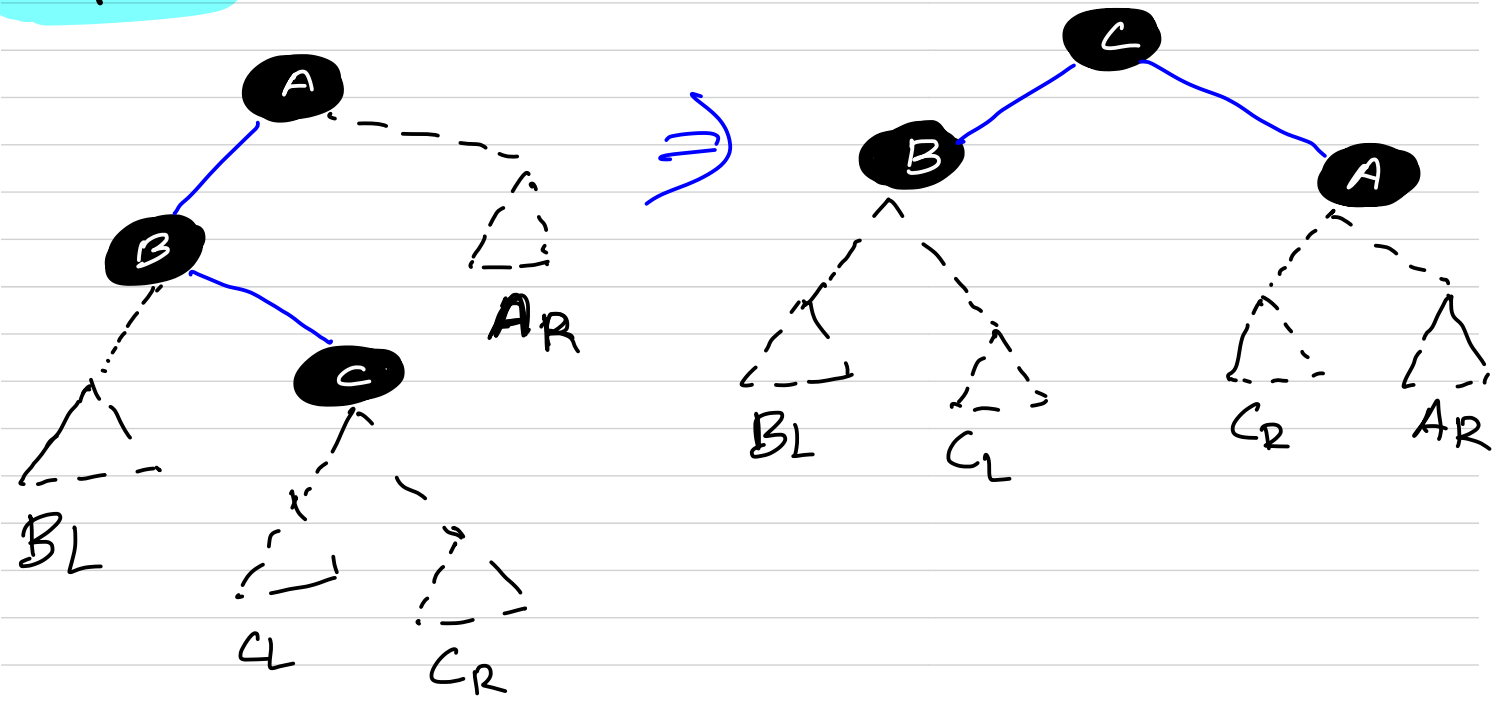
- RR -


A
B
C
AL
BL
CL
CR
B
A
C
AL
BL
CL
CR

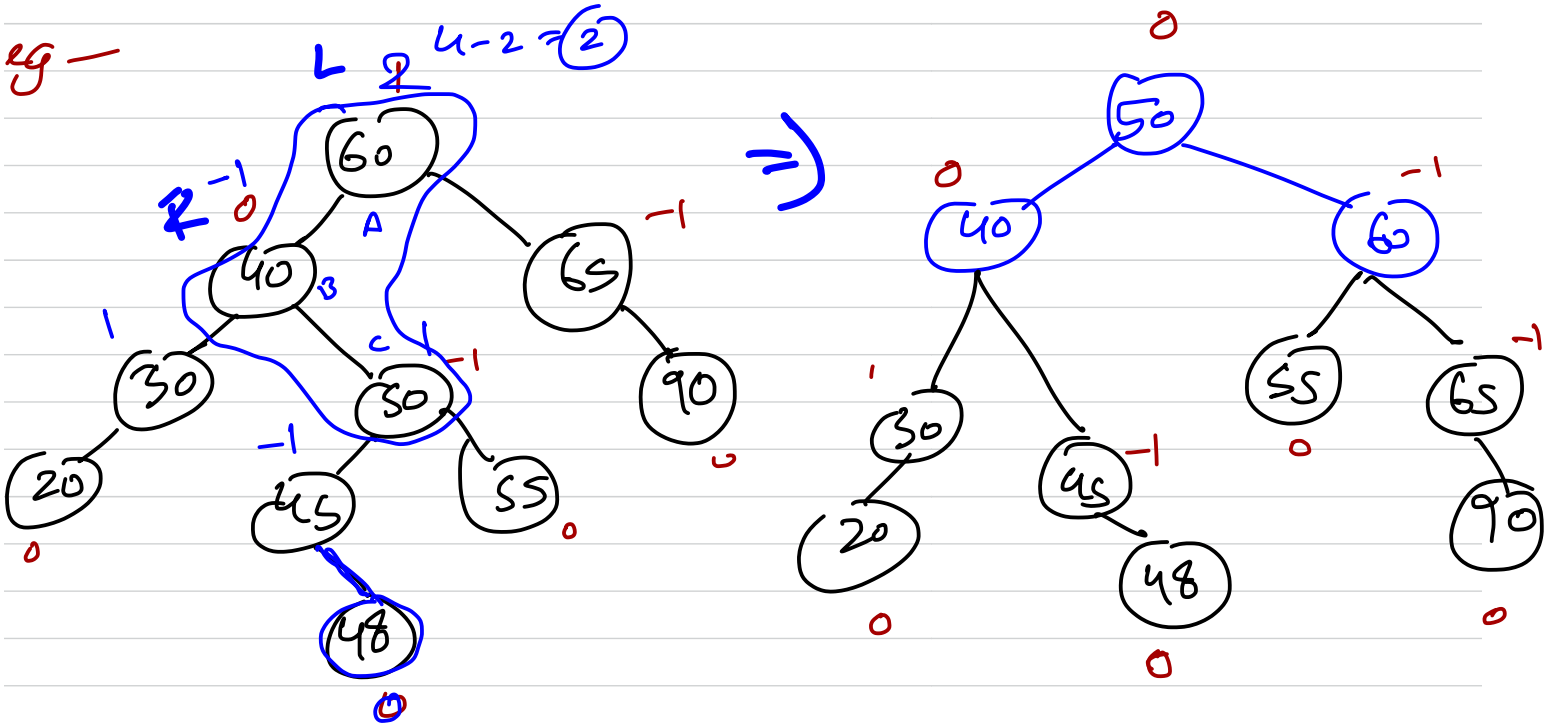
eg


R
-2
60
A
R
-1
40
70
B
C
-1
20
65
80
62
75
90
95
0
70
60
80
40
65
75
90
20
62
95

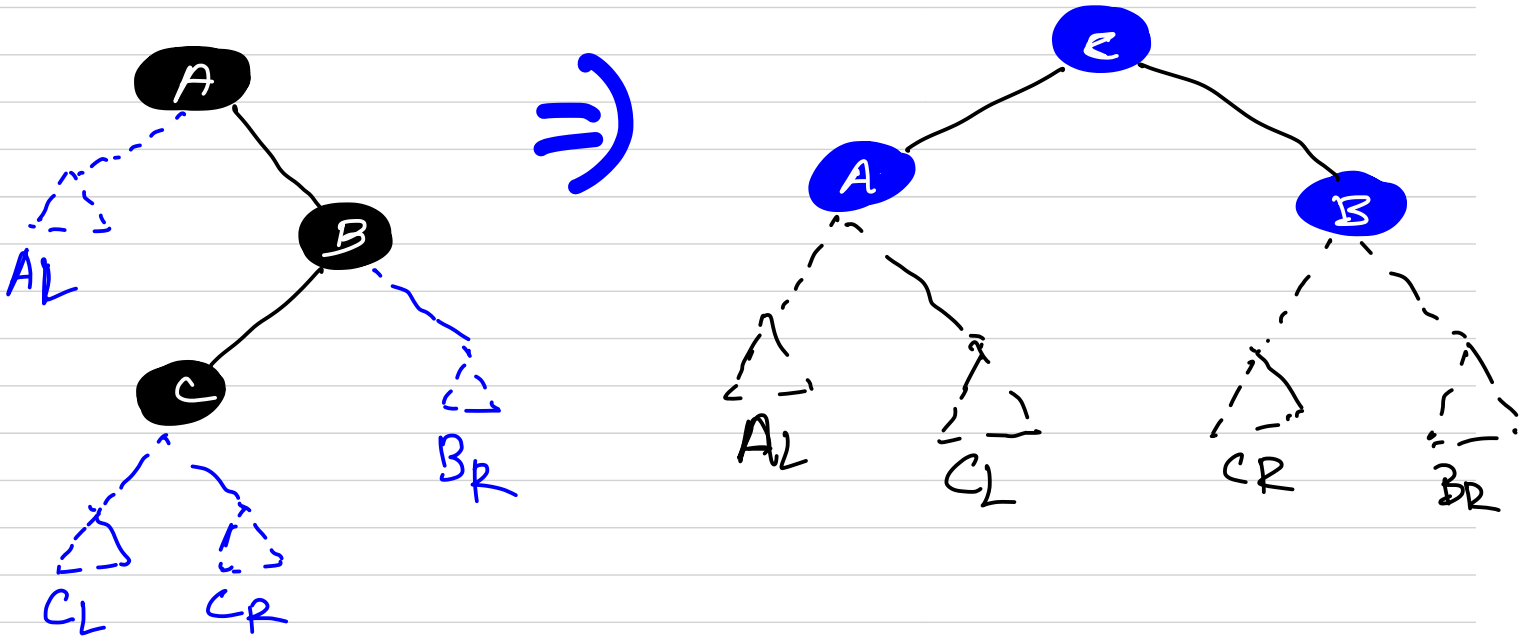
– LR –


A
B
C
AR
BL
CL
CR
C
B
A
BL
CL
CR
AR

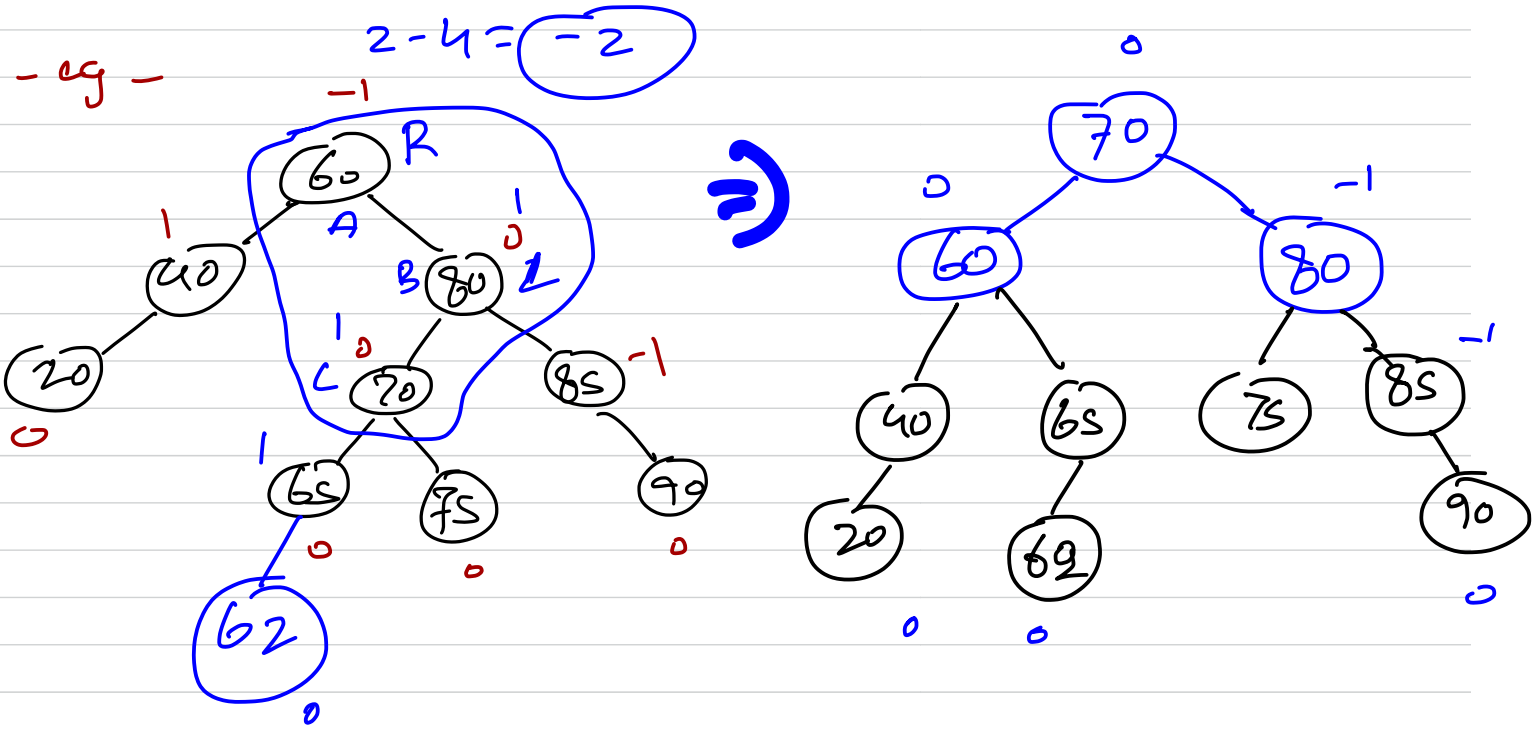
eg –


60
40
65
30
50
90
20
45
55
48
50
40
60
30
45
55
65
20
48
90

– RL –

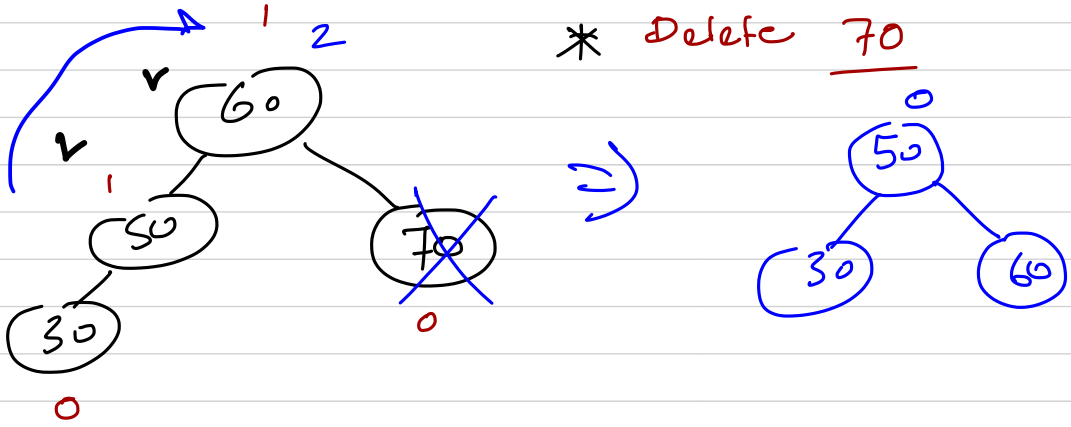

A
B
C
AL
BR
CL
CR
C
A
B
AL
CL
CR
BR

– eg –

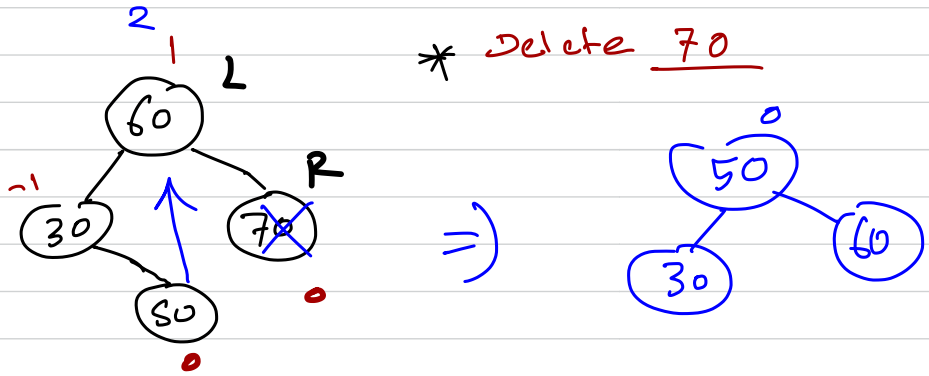

2 - 4 = -2
-1
R
60
A
40
B
80
L
20
C
70
85
-1
65
75
90
62
0
70
60
80
40
65
75
85
20
62
90

- AVL Trees Rotation After Deletion -

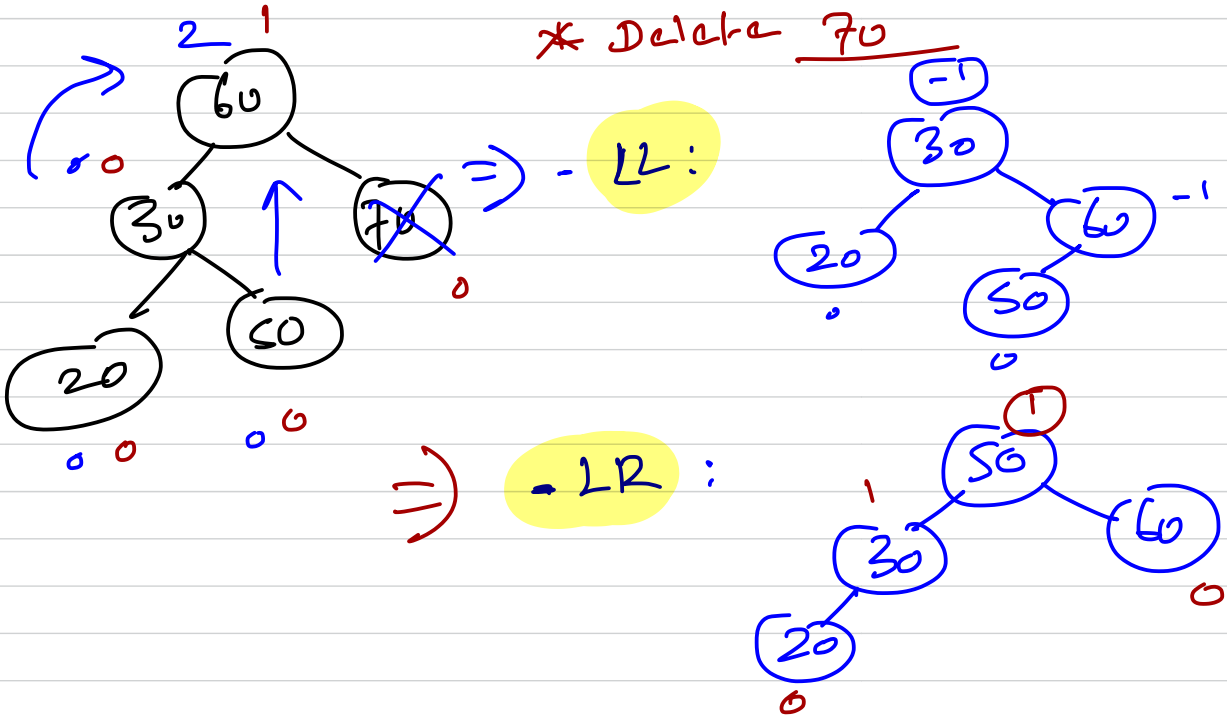
- LL -



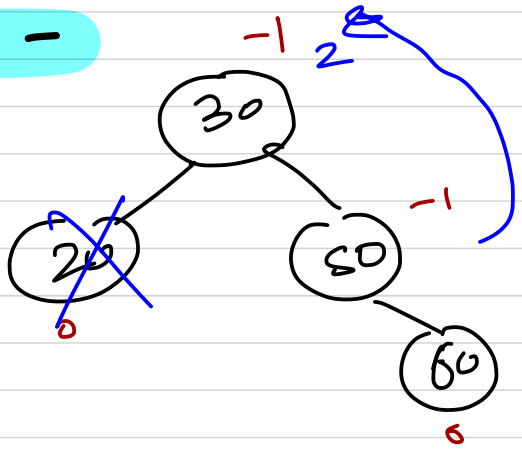
- LR -



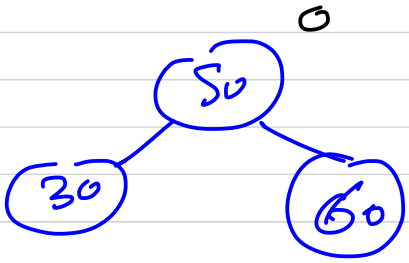
- LL || LR -



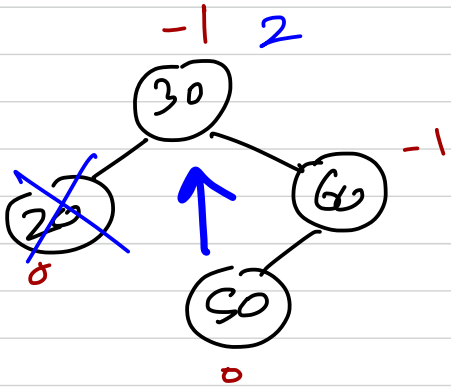
- RR -



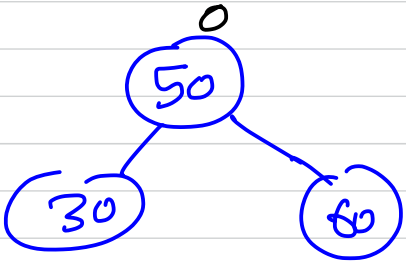
* Delete 20



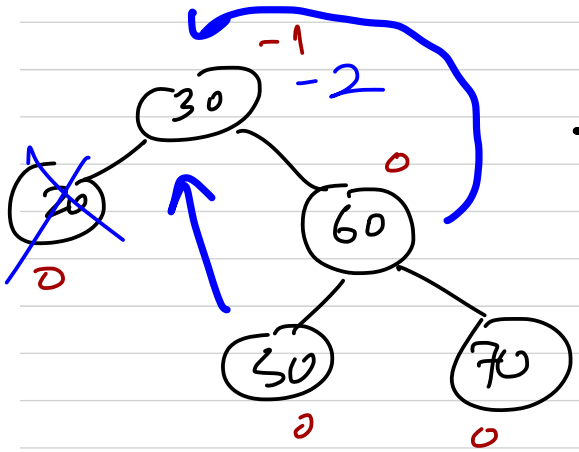
- RL -



* Delete 20

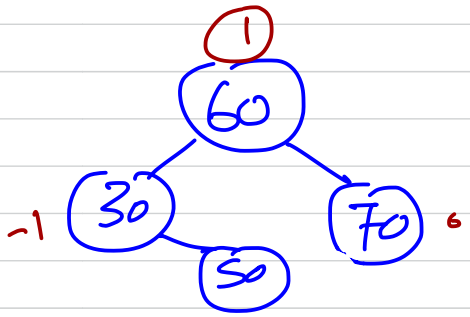


- RR II RL -

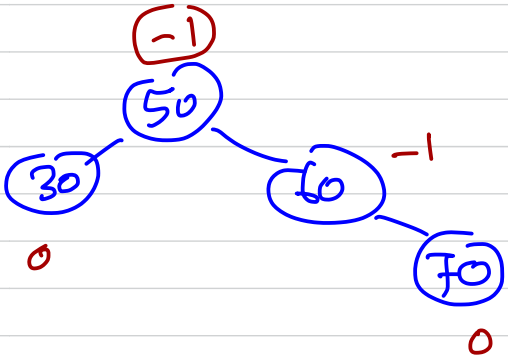


* Delete 20

⇒ RR:



⇒ RL:



— Performance Analysis of AVL —

- AVL Trees rotations:
  - * LL
  - * LR
  - * RR
  - * RL

} O(1)

- Height :
- Searching :
- Insertion :
- Deletion :

} O(log n)

— Red - Black Tree —

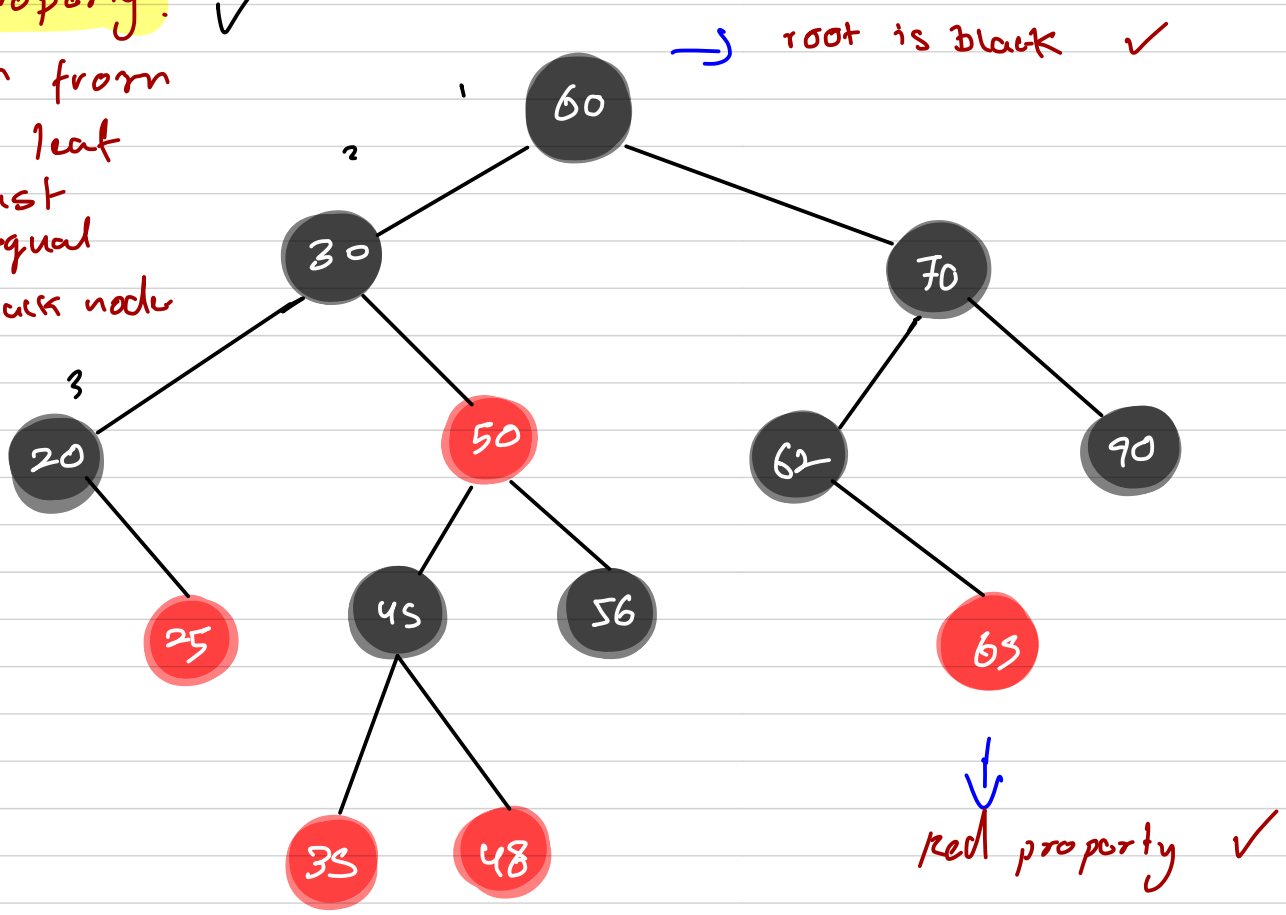
* Balanced Binary Tree
* AVL Trees require additional restructuring operations.

— Properties — Every node colored Red or Black.

* Root property — Root is Black
* Red property — children of Red node are Black.
* Depth property — All nodes have same Black depth

Depth property: ✓

the path from root to leaf node, must contain equal no. of black node

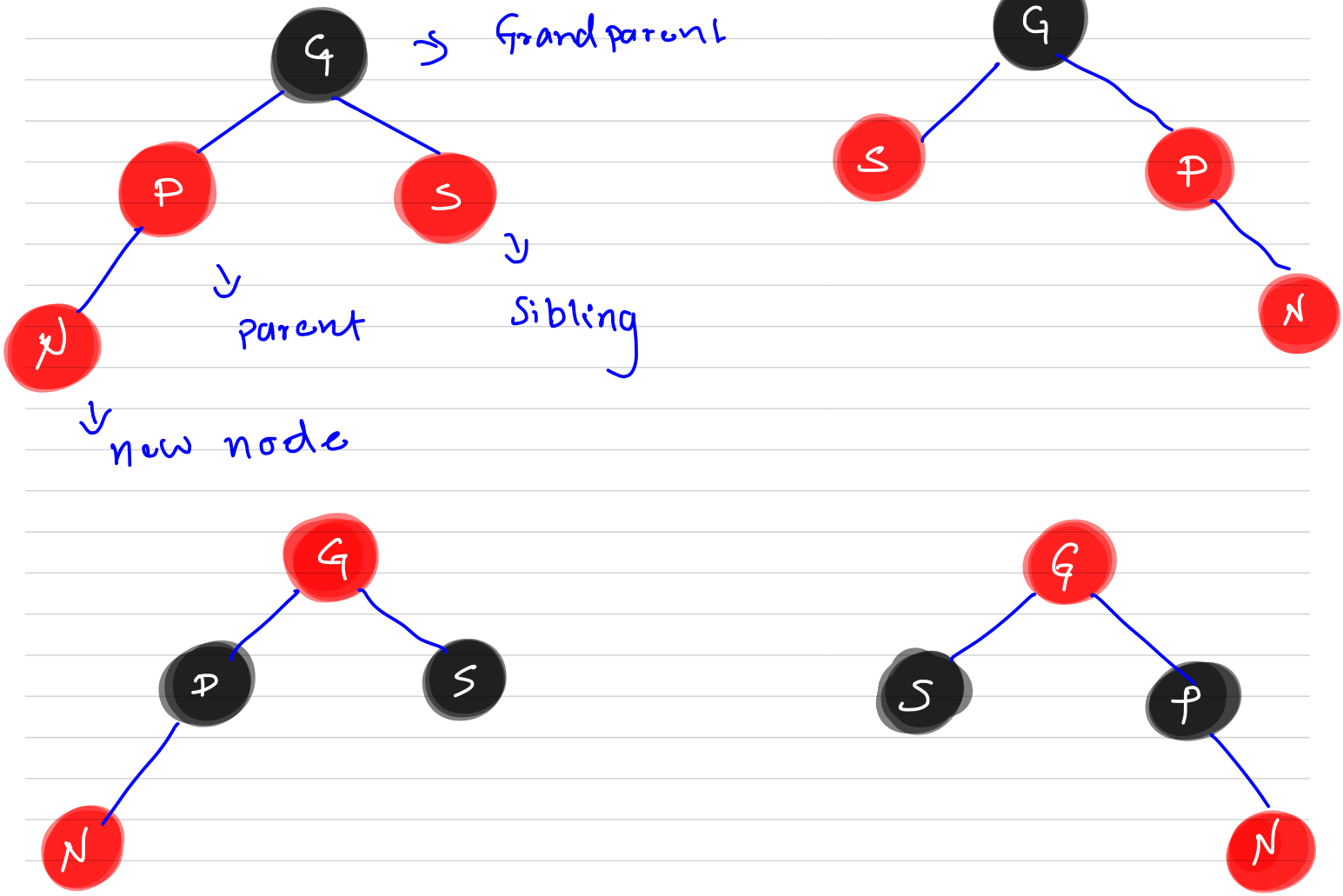


Red - Black Tree Restructuring

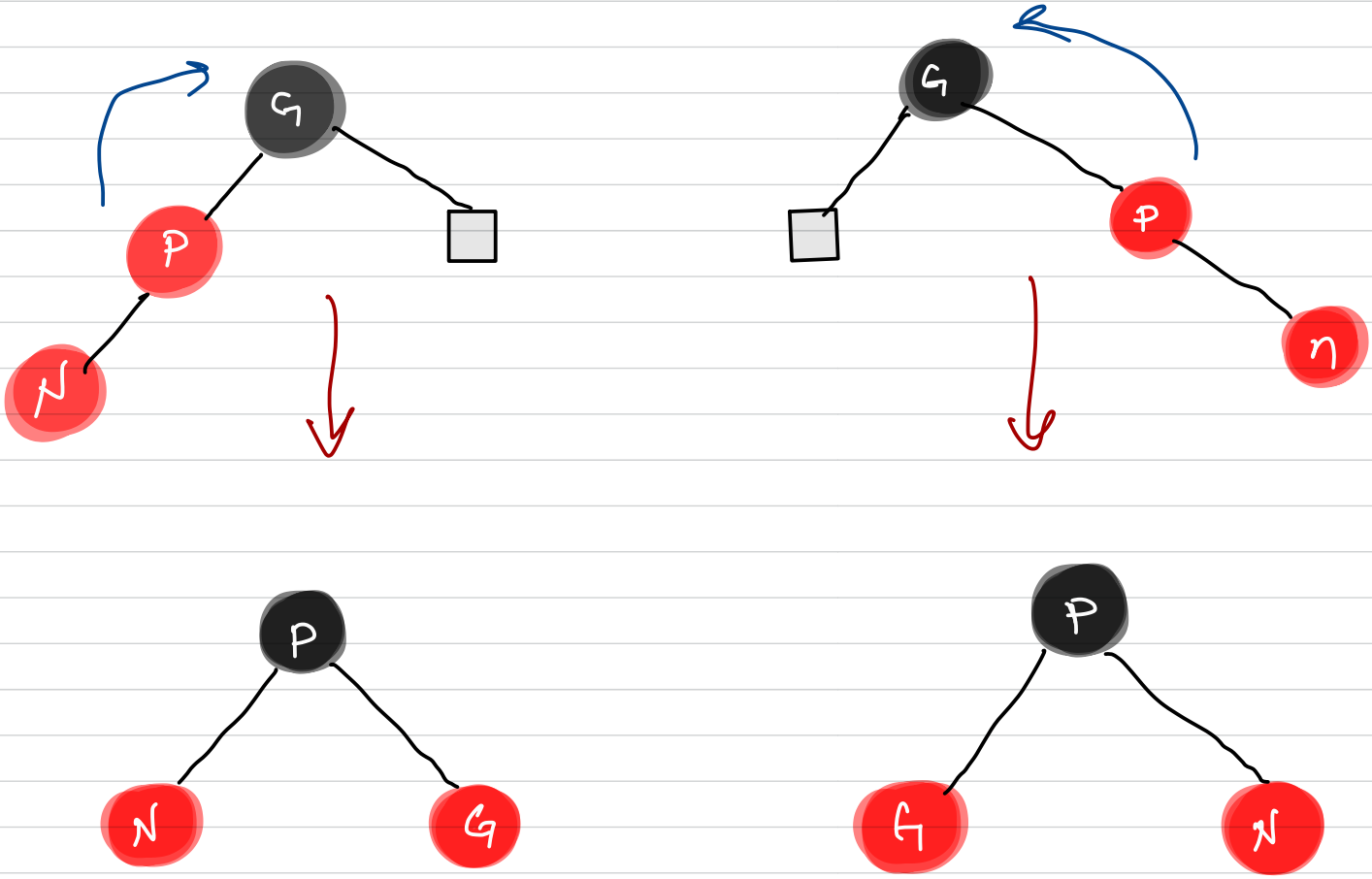
_Restructuring:

* Restores the balance
* Done after the update (insertion or Del)
* Involves coloring & may be rotations

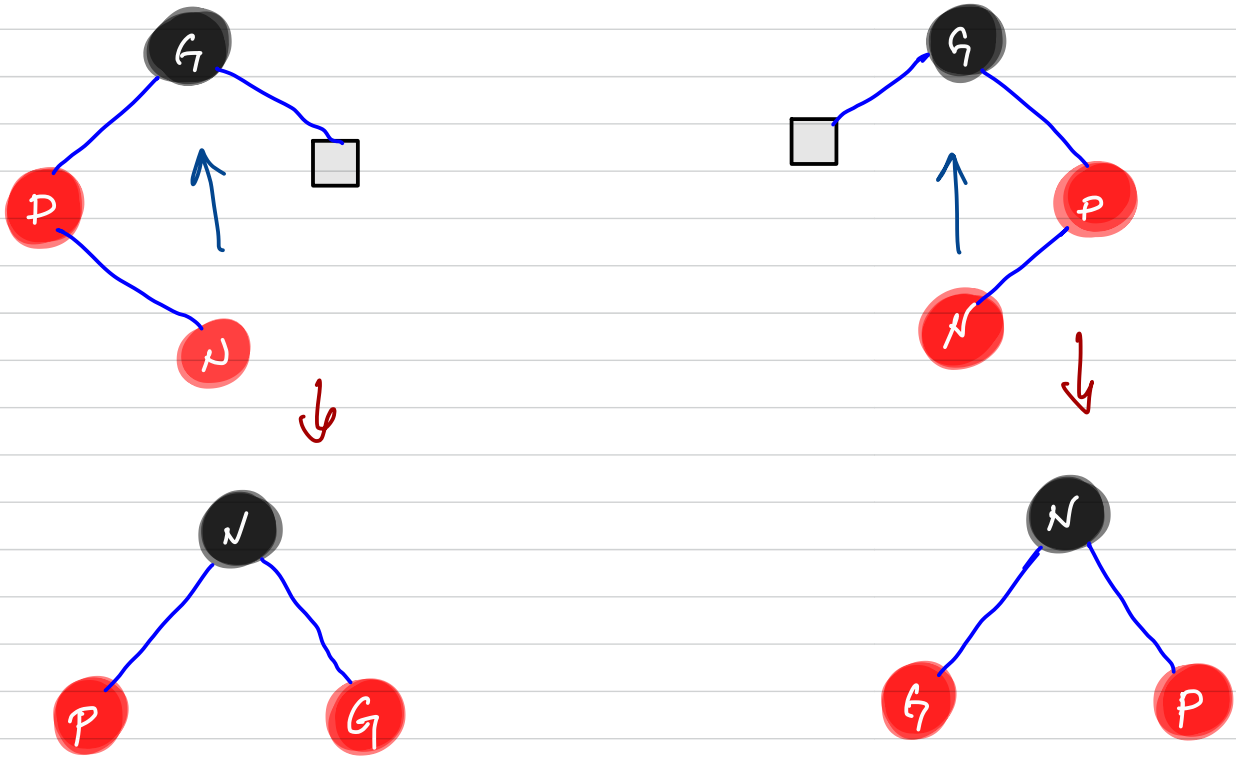
- Scenario-1 - Recoloring



• Scenario-2 – Rotation



• Scenario-3 – Rotation



Red-Black Tree Insertion

properties:

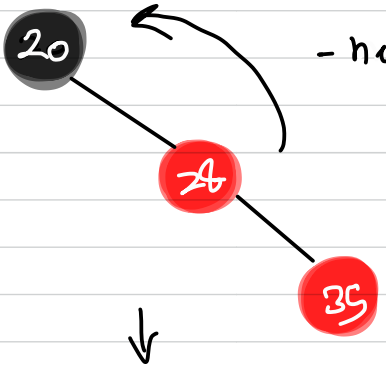
- Root
- Red
- Depth

Elements: 20, 28, 35, 52, 45, 68

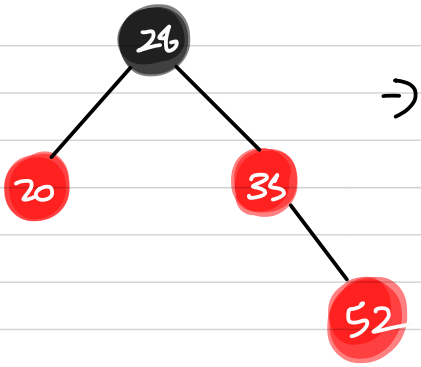
1)

→ bcuz of root property change to black.

2)

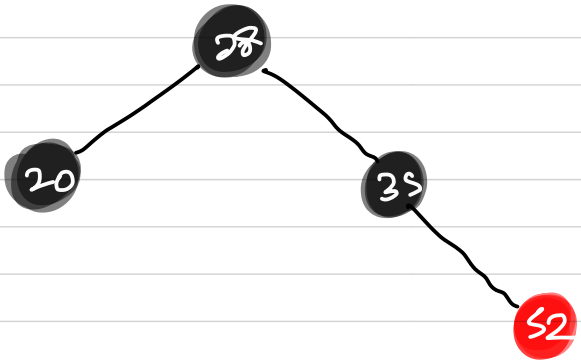


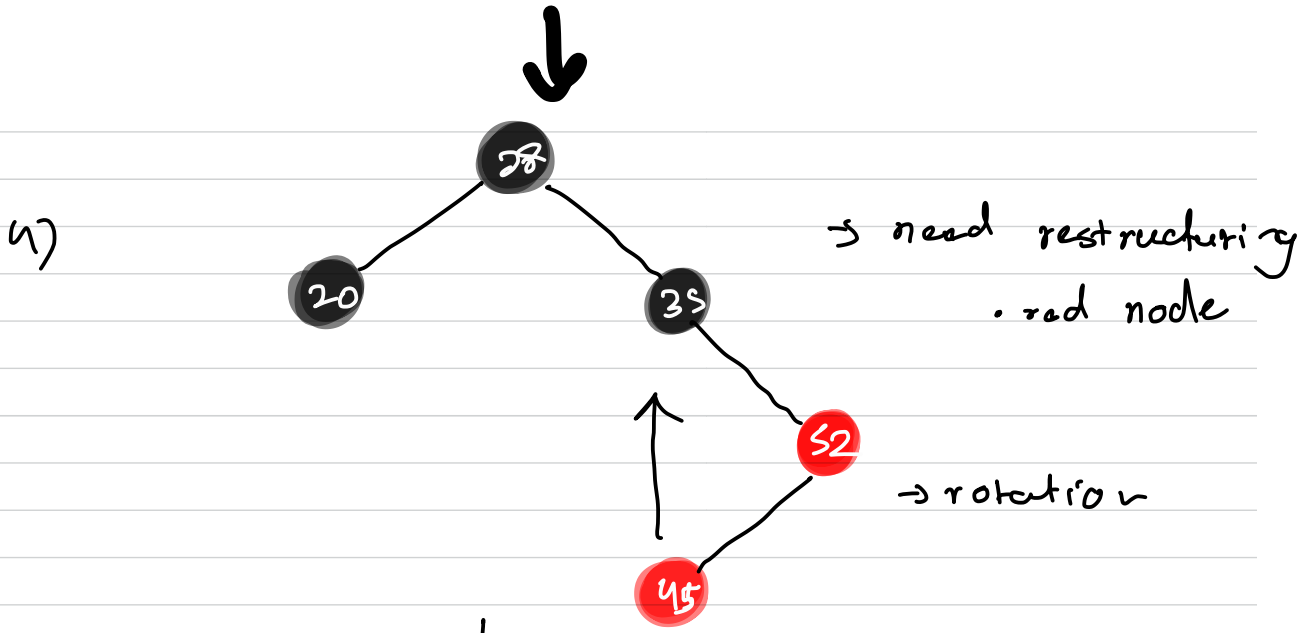
- need restructuring
- red node property

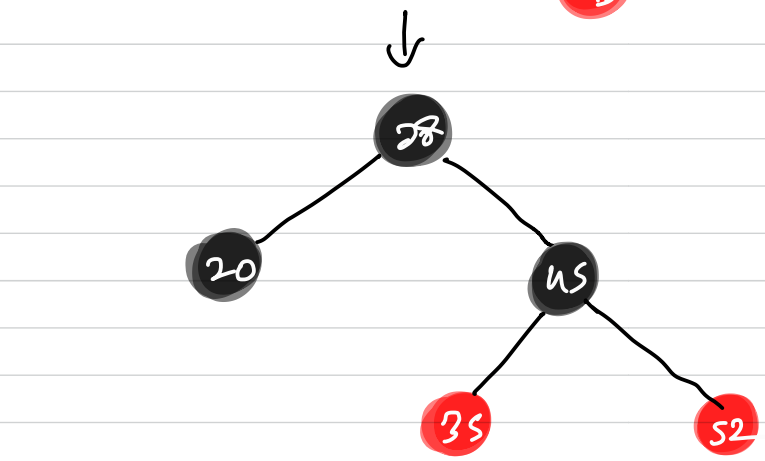


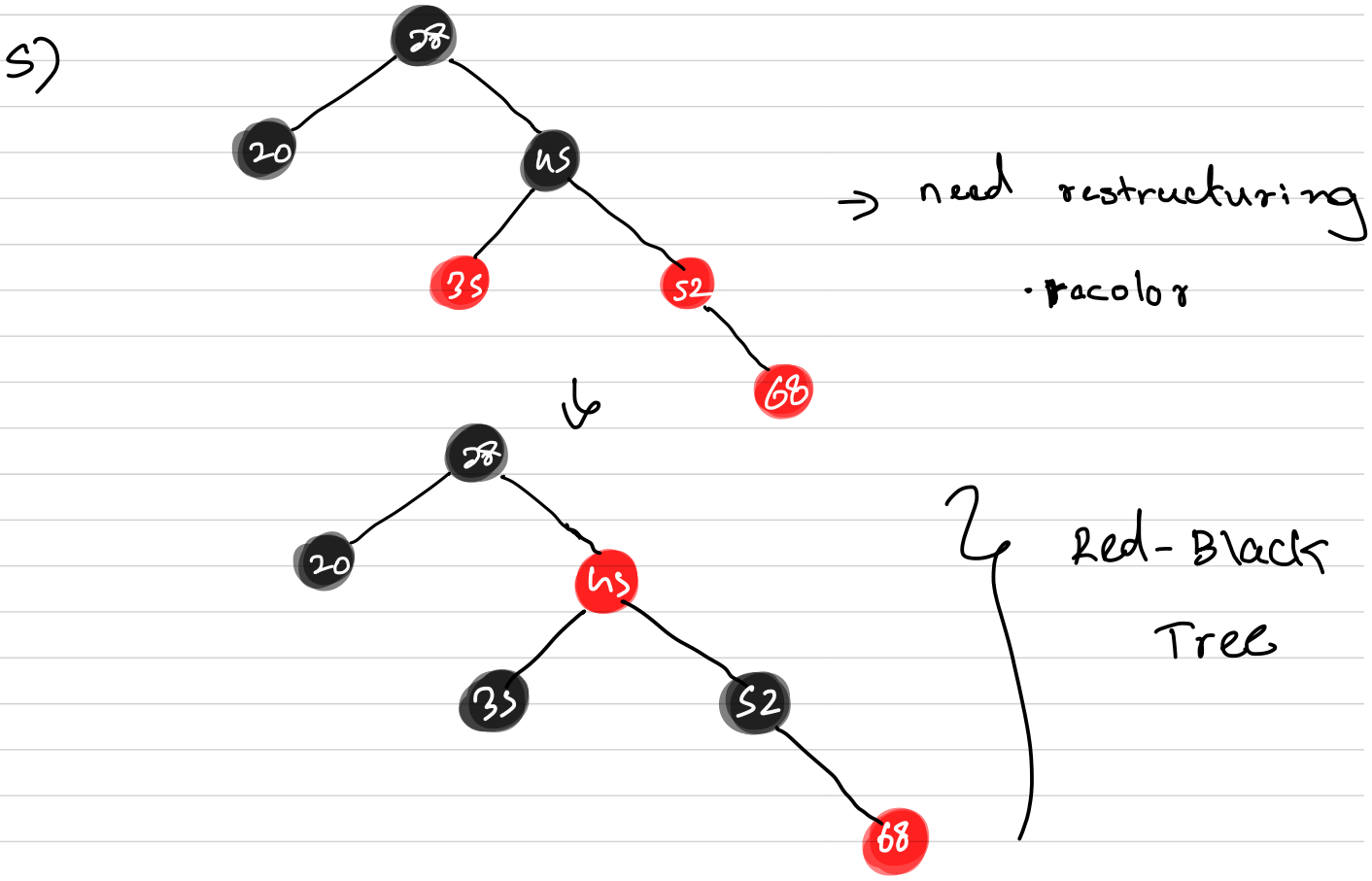
→ need restructuring
- red property

3)




4)
78
20
35
52
45
→ need restructuring
· red node
→ rotation


78
20
45
35
52


5)
78
20
45
35
52
68
→ need restructuring
· recolor
78
20
45
35
52
68
Red-Black Tree

Red-Black Tree Deletion

- Deleting node is same as Binary Search Tree.

→ Scenarios :

1) removed node is leaf node and is red.

2) removed node is leaf node and is black.

3) " " is RED non-leaf with one subtree

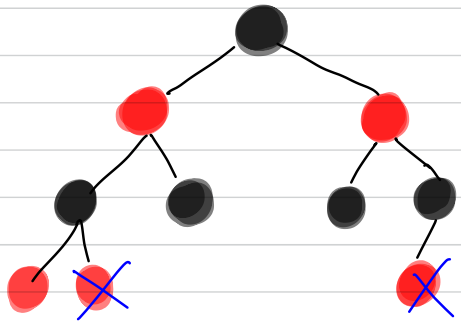
4) " " is BLACK non-leaf with one subtree

5) " " is RED non-leaf with two subtree

6) " " is BLACK non-leaf with two subtree

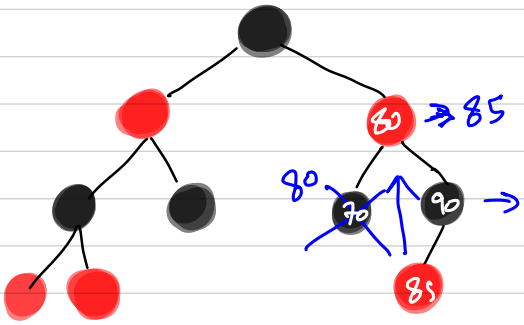
Senarios –

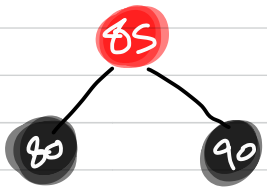
1)



Removing node:
is leaf node
and is red
doesn't chang
the structure.
we don't need
to change.

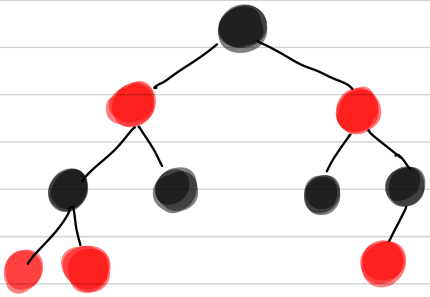
2)





2. leaf node
and is
black.

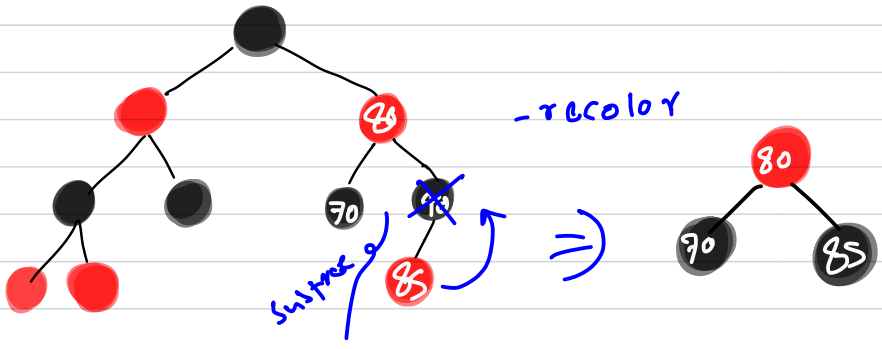
3)



non-leaf node
which is red with
one subtree.

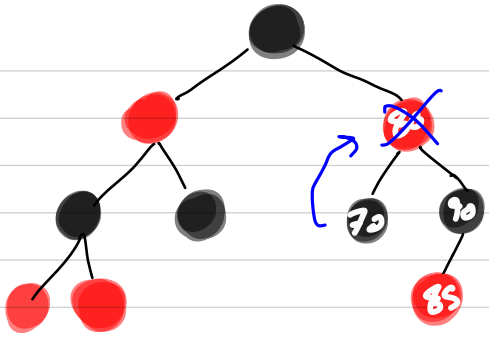
can't be restructured.

4)



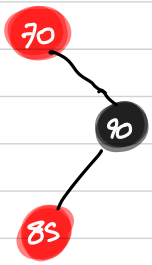
non-leaf Black
with one
subtree.

5)



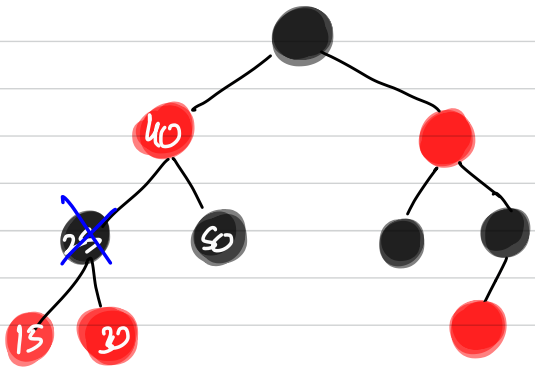
→ this is BST,

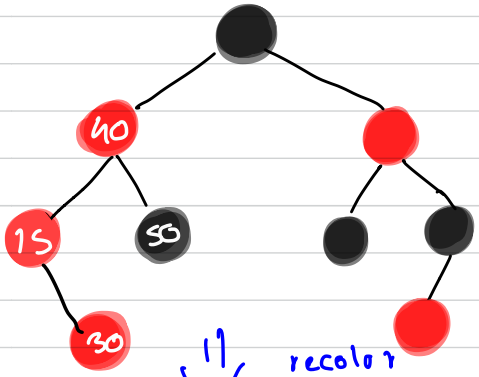
- 70 → 80
- recolor



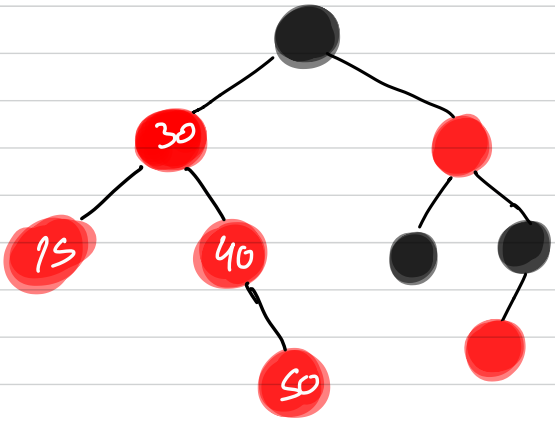
- RED - non-leaf, with two subtree.

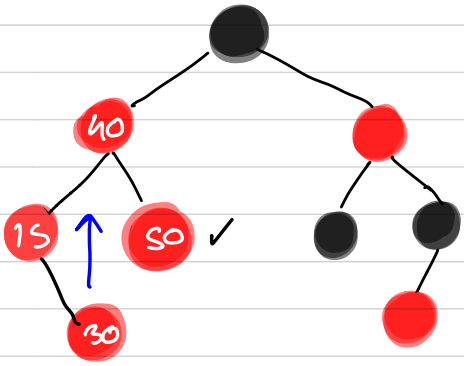
6)

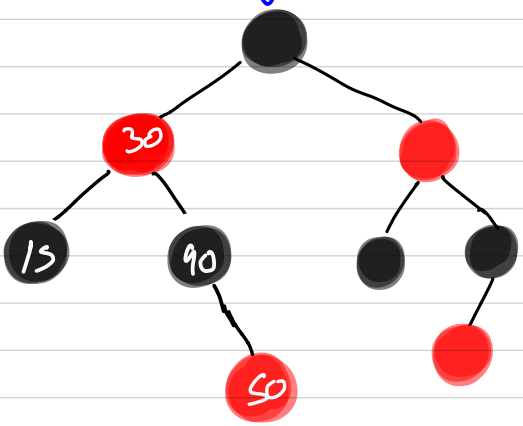




recolor







– Performance Analysis of Red-Black T –

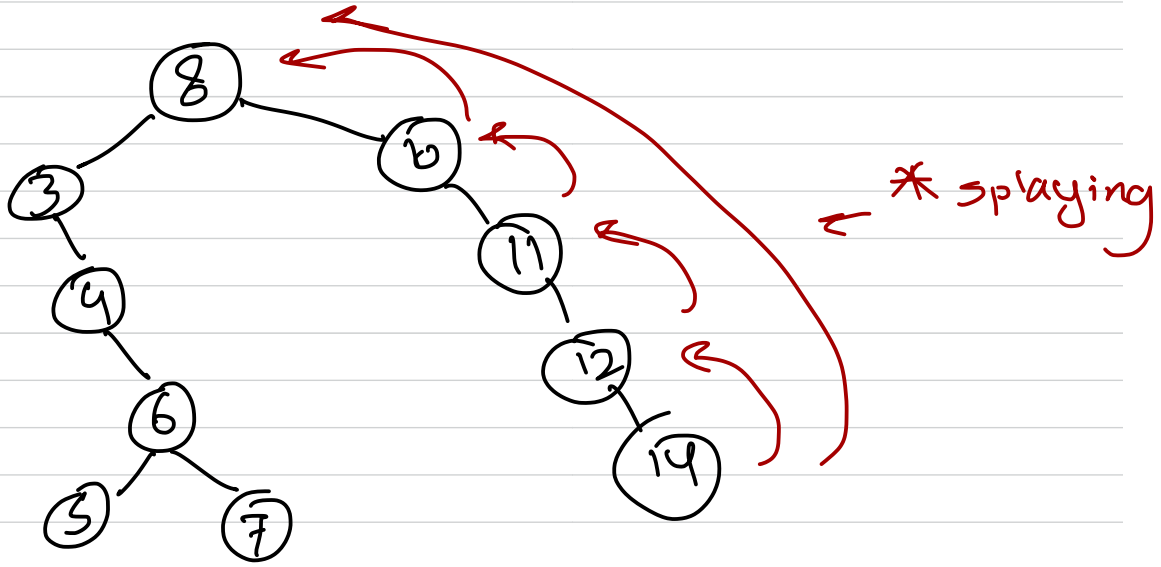
* Searching
* Insertion
* Deletion

} $O(\log n)$

- Splay Trees -

* Splay trees are balanced search trees
* Doesn't enforce bound on the height
* Performance achieved through Splaying * operation
* frequently accessed element is near the root

- Splaying: move the node to the root.
- Depends on the position of the node.
- Also depends on its parent and grandparent.



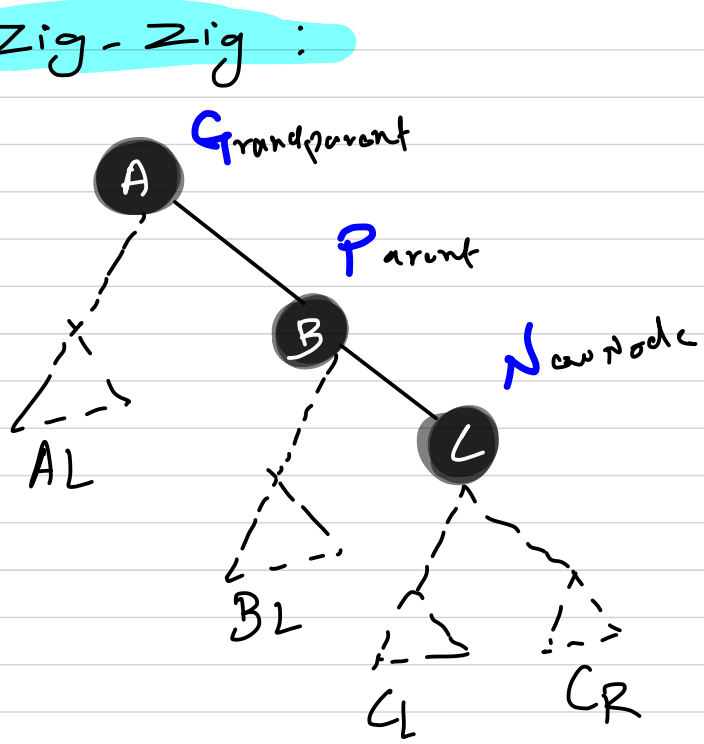
- Rules to perform Splaying -

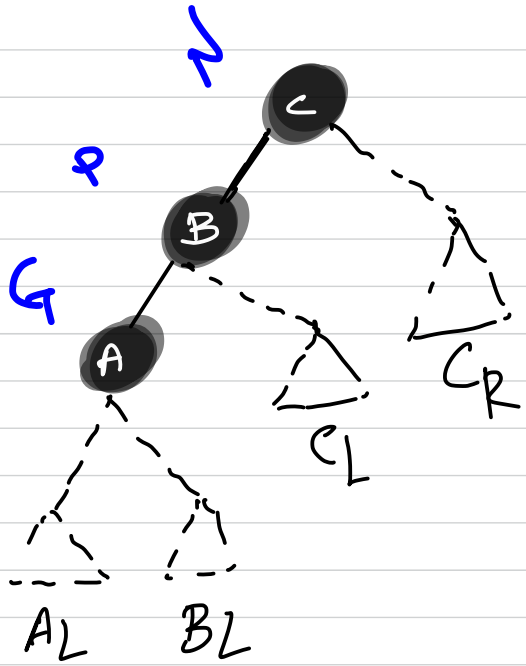
* Searching : When key is found, then splay the node at that position.
* Insertion : Splay the newly created node
* Deletion : Splay the parent of the removed node.

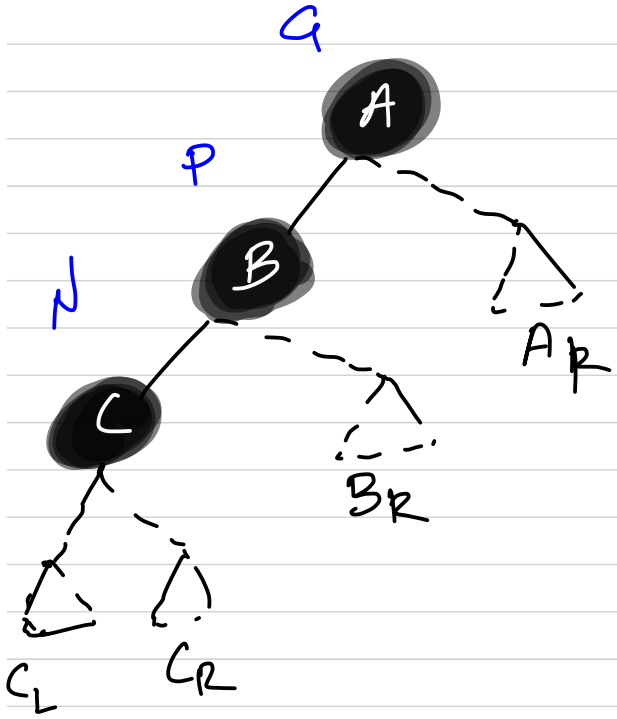
- Splay Trees - Restructuring -

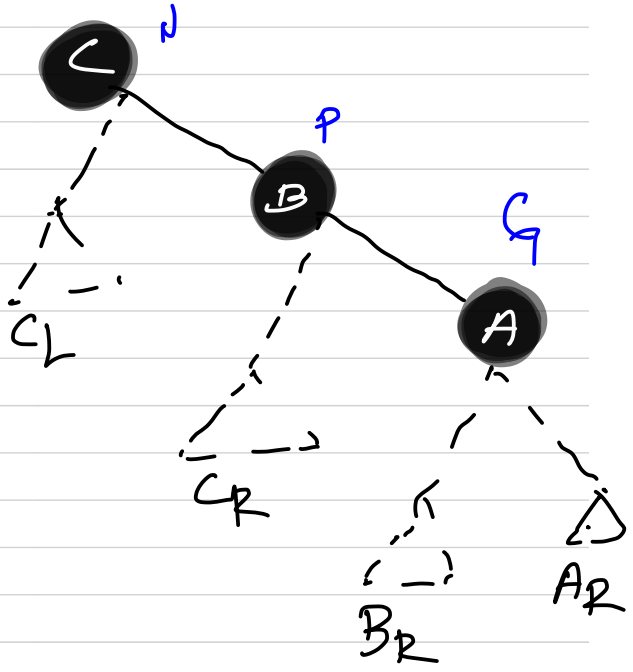
* Zig-Zig : Node & parent both left or right children.
* Zig-Zag : Node as left & parent as right child or vice verse
* Zig : Node dosn't have grandparent.

Zig-Zig :

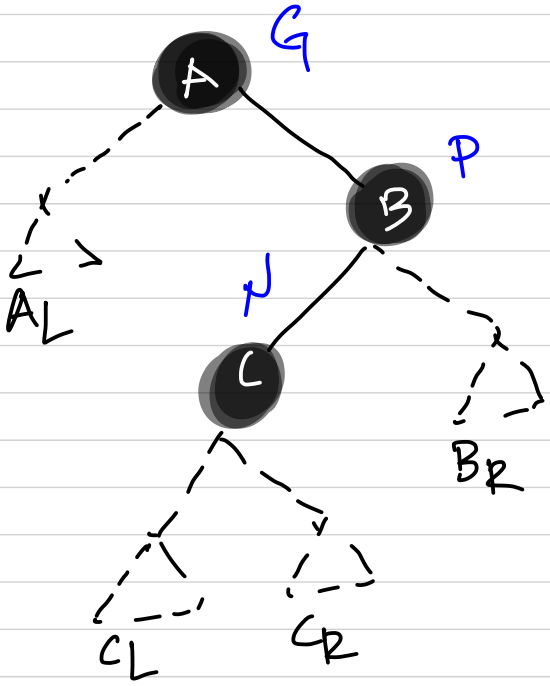

G Grandparent
P Parent
N New node
A
B
C
AL
BL
CL
CR

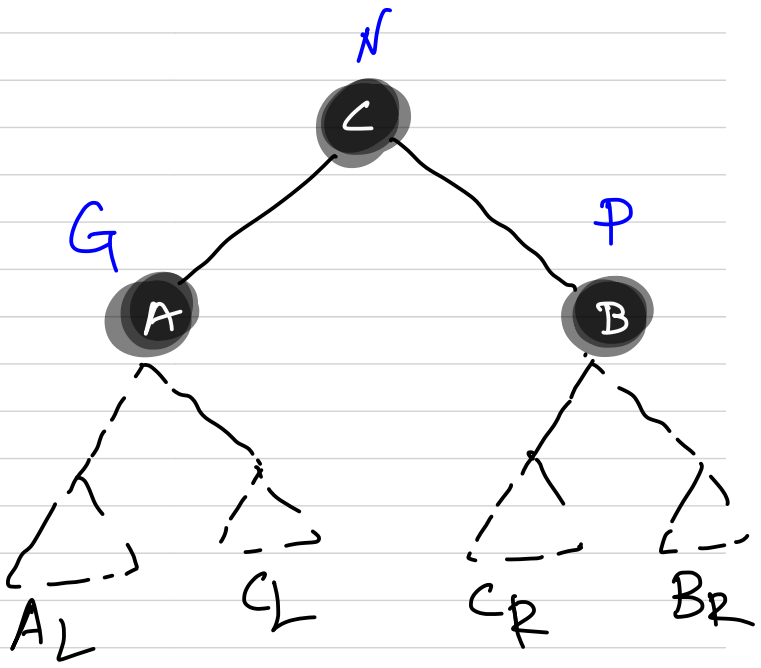

N
P
G
C
B
A
CR
CL
AL
BL

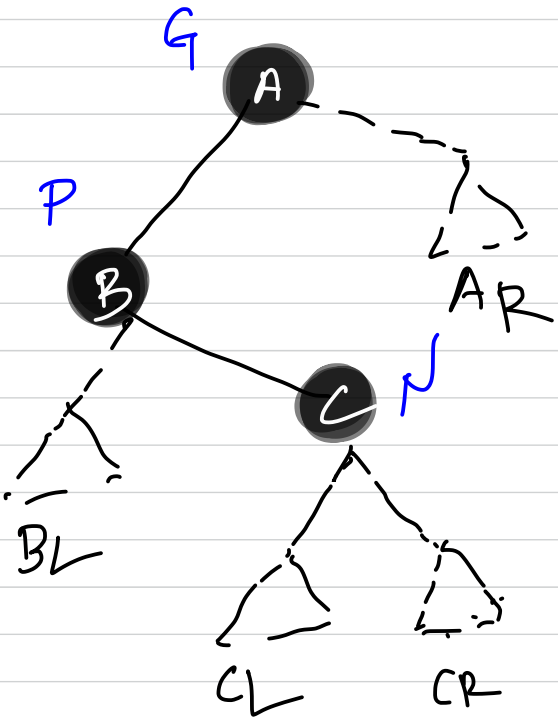

G
P
N
A
B
C
AR
BR
CL
CR

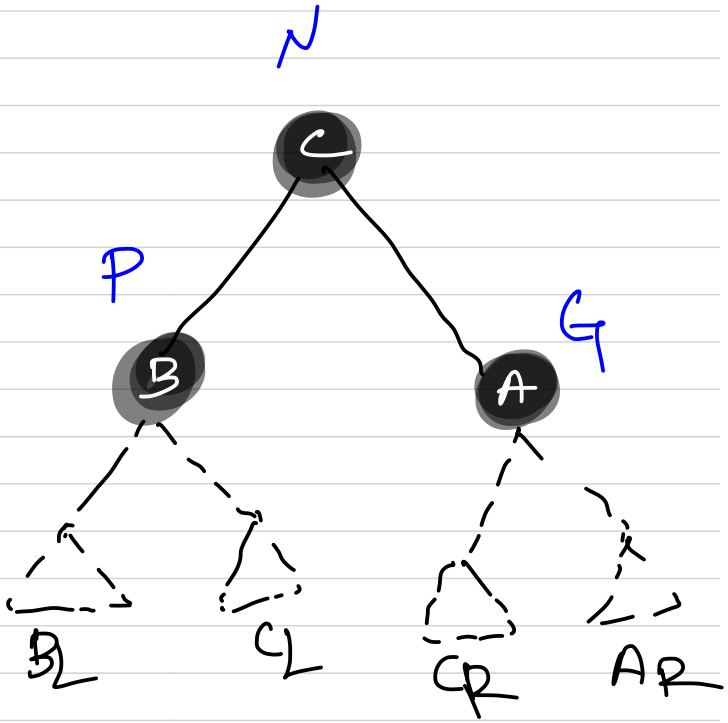

N
P
G
C
B
A
CL
CR
BR
AR

Zig - Zag -

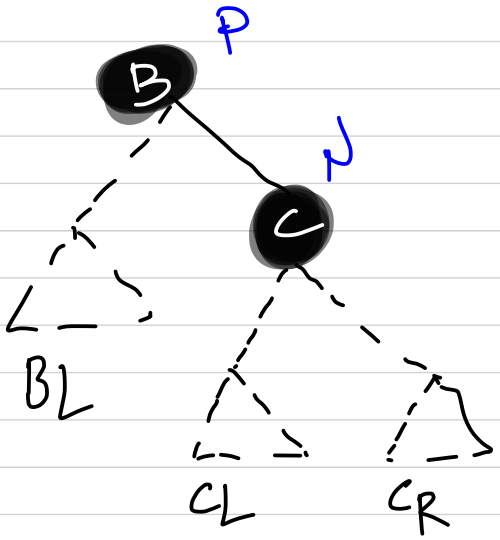

G
A
P
B
N
C
A_L
B_R
C_L
C_R

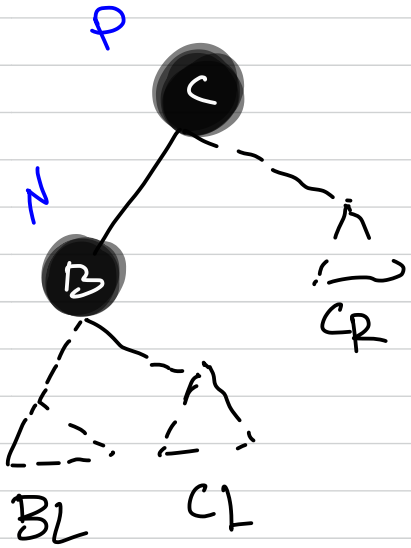

N
C
G
A
P
B
A_L
C_L
C_R
B_R

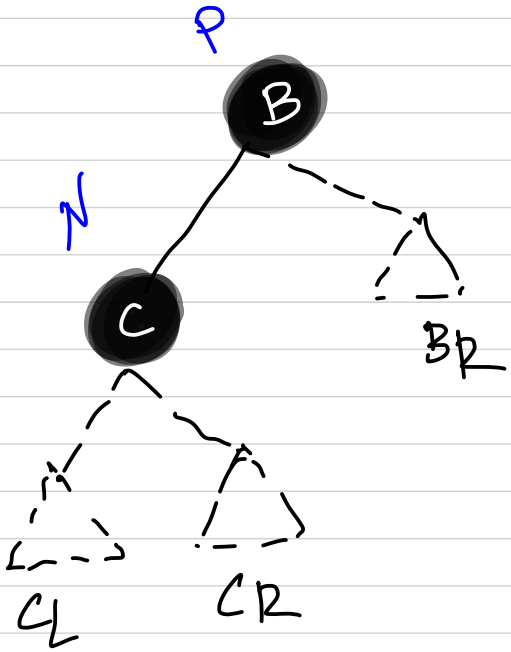

G
A
P
B
A_R
N
C
B_L
C_L
C_R

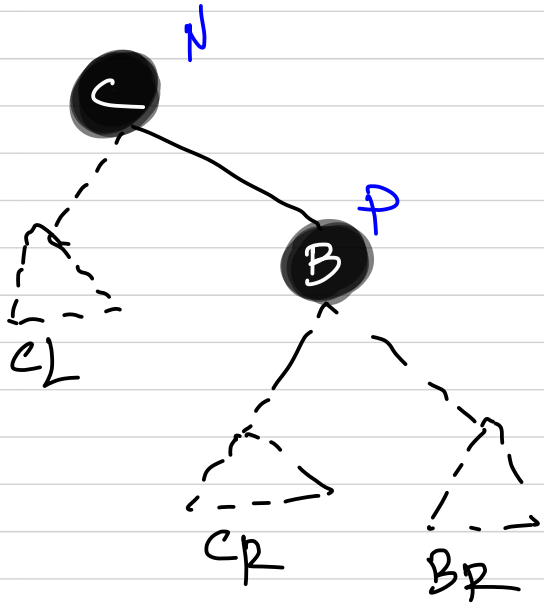

N
C
P
B
G
A
B_L
C_L
C_R
A_R

Zig –





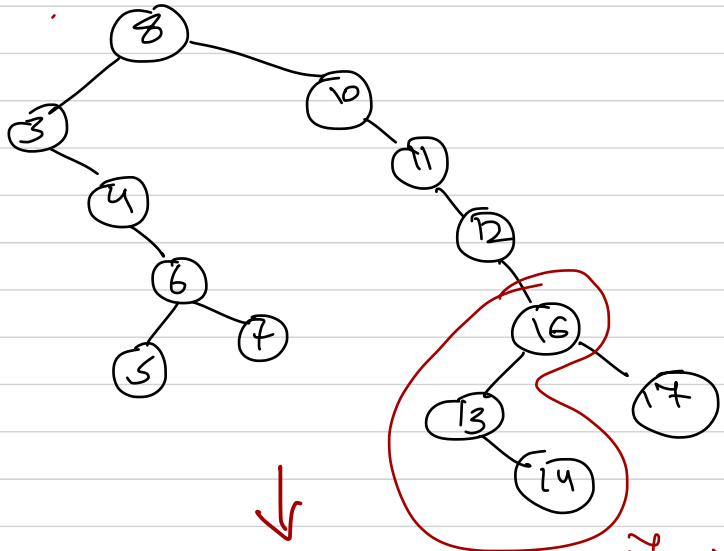




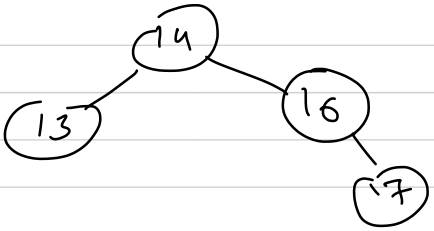
- Splaying -

- move the node to the root.
- Depends on the position of the node.
- Also depends on its parent and grandparent.

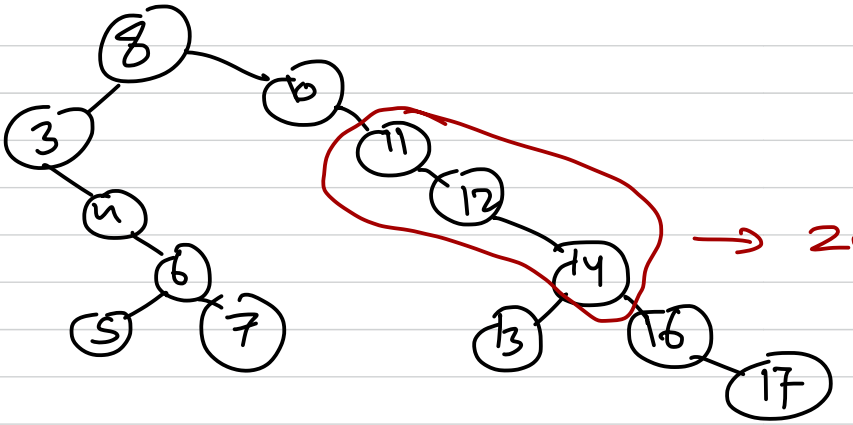
→ splaying.



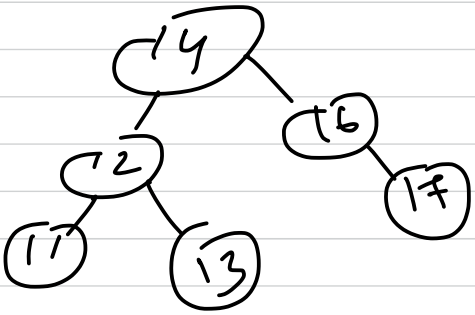
→ zig-zag



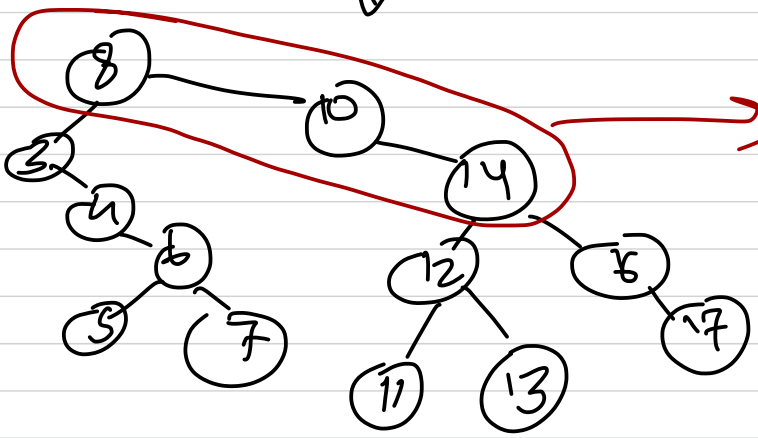
→ after zigzag



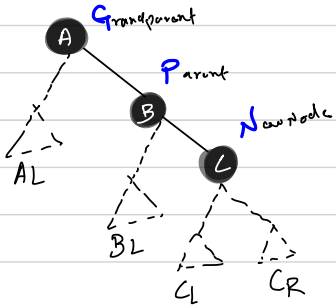
→ zig zig

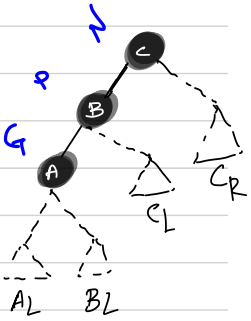


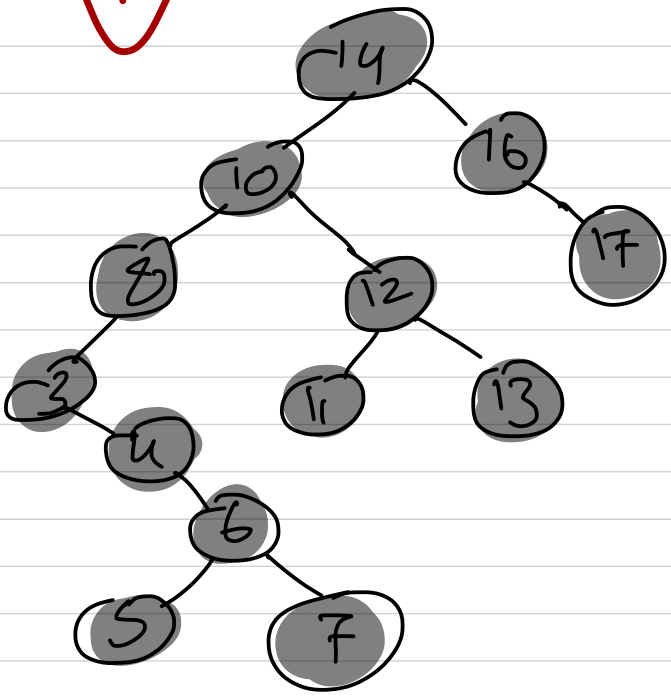
→ after zig zig



→ zig-zig







→ Splaying ✓✓✓

- Performance Analysis of Splay Trees -

* Searching :
* Inserting :
* Deleting :

} $O(h)$ → height of the tree.


HEAP

- Heaps -

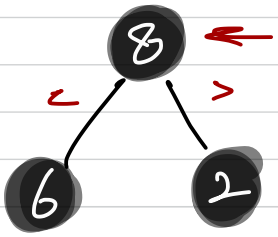
- priority Queues -

* Collection of prioritized objects
* Insertion: According to the first come basis.
* Removal: based on priority of obj
* Key is associated when element is inserted in prio. queues.
* Element with min key will be next next to be removed.

- Heaps -

* collections of obj or elements stored as a binary T.
* Binary Heap
* Relational property: Key in each node of binary tree is greater than or equal to its children.
* Structured property: Binary Tree should complete Binary T.
* Max Heap && Min Heap

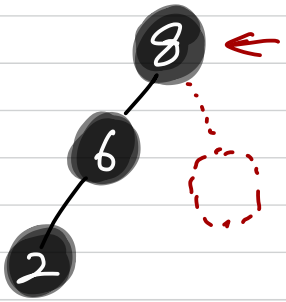
Example:



✓ Heap

→ relational ✓

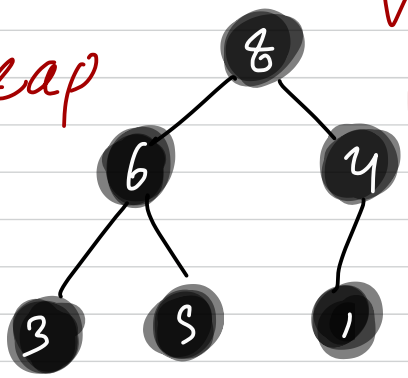
→ structural - complete BT ✓



relational ✓

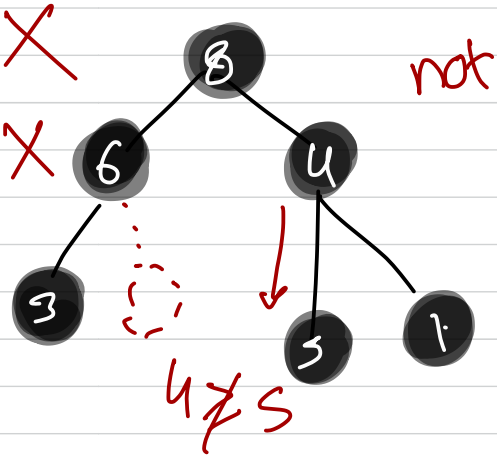
structural ✗

Heap



✓ : relational: ✗

: structural: ✗



not Heap

4 ≠ 5

- Heap Abstract Data Type -

- Members:
  - max size
  - current size
- Operations:
  - Insert (object) : Insert element in Heap
  - DeleteMax() : Delete max & return it
  - max() : return max

- Heap - Insertion: O(log n)

↳ O(1)

* Element is inserted as a new node in tree.

* Structural: New node is inserted after the last node.

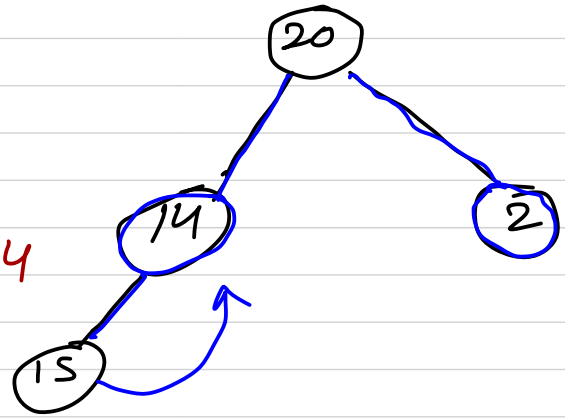
* Relational: perform up-heap bubbling → O(log n)

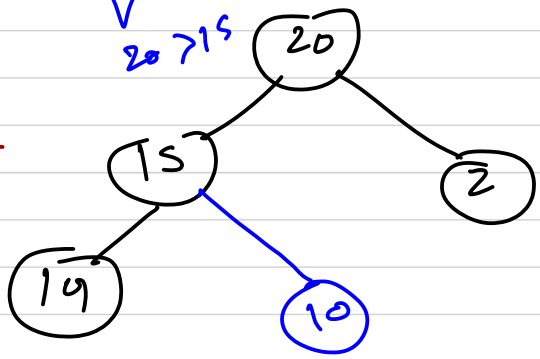
Example —

insert(20)
// (14)
// (2)
// (15)
// (10)

* 15 > 14

*perform up-heap bubbling

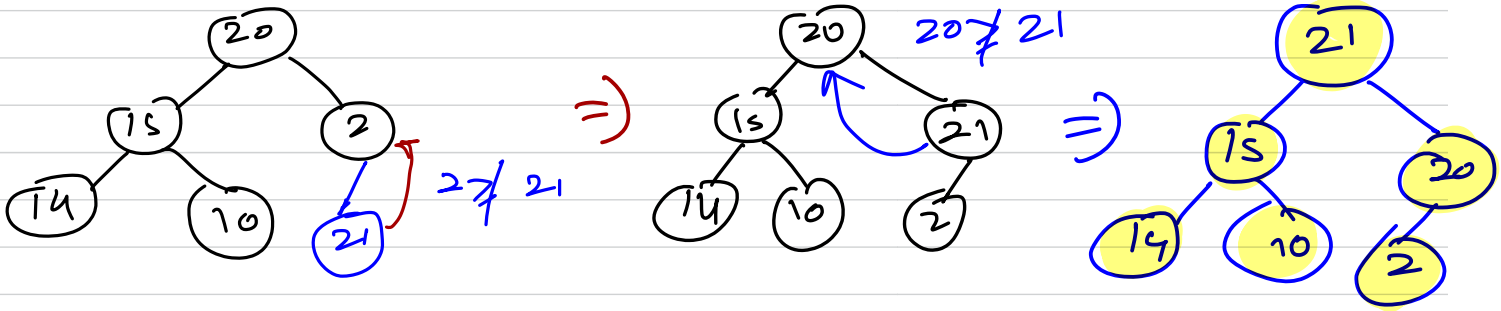




✓ 20 > 15 ✓✓

* check it with root as well

insert(21)



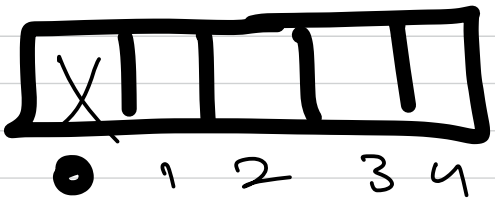
heap_insert(e)

current size

```
{  if csize == maxsize
       print("no space");
       return
   csize++;
   hi = csize;
   while hi > 1 && e > data[⌊hi/2⌋]
   {  data[hi] = data[⌊hi/2⌋]   → swap
   }  hi = ⌊hi/2⌋
   data[hi] = e   → swap
}
```

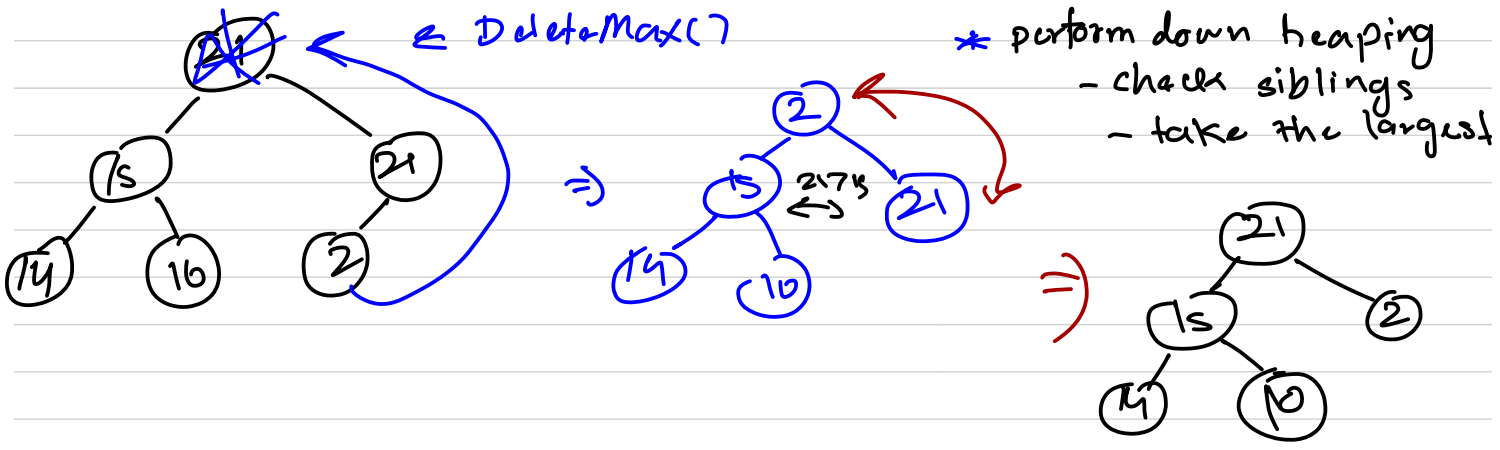
heap index

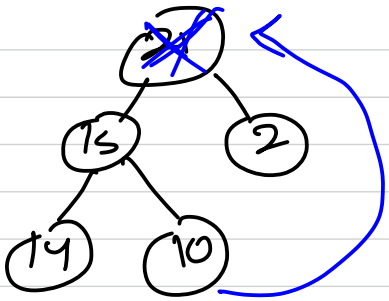
up-heap bubbling



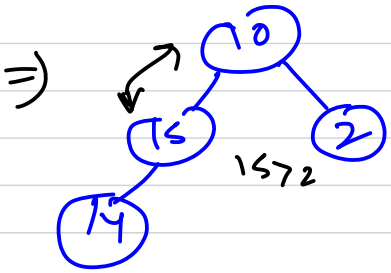
Heap - Deletion — O(log n)

* Element is removed from the root
* Structural: Root is replaced by the last node
* Relational: perform down-heaping



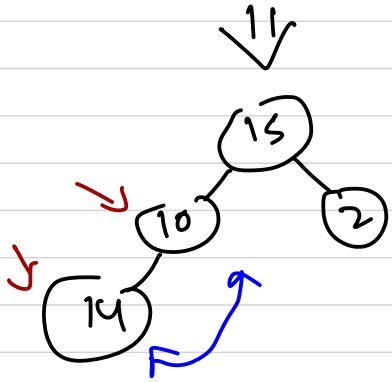


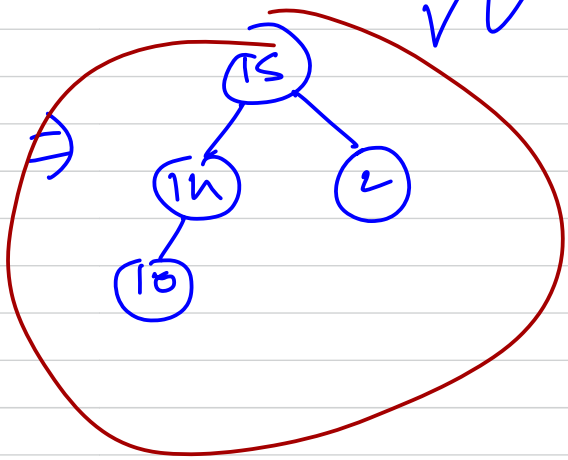
deleteMax() :



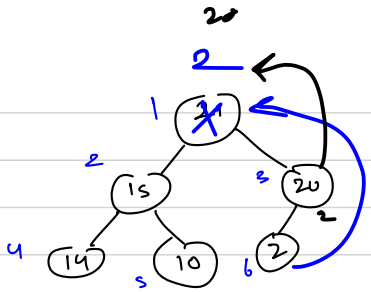
* perform down-heap to satisfy the relational property.

* Still the relational property isn't satisfied.





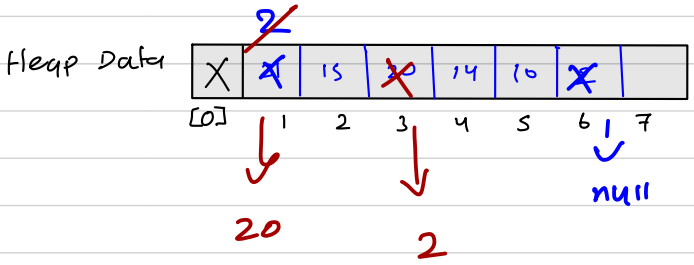
O(logn)



size = ~~5~~

i = ~~3~~

j = i*2 = ~~3~~ 6



deleteMax()

{

e = data[1] 21

data[1] = data[csize] 2

data[csize] = null

csize --

i = 1, j = i * 2

while j ≤ csize

{

if data[j] 15 < 20 data[j+1]

j++

if data[i] 2 < data[j]

temp = data[i] = 2

data[i] = data[j]

data[j] = temp

i = j

j = i * 2

else

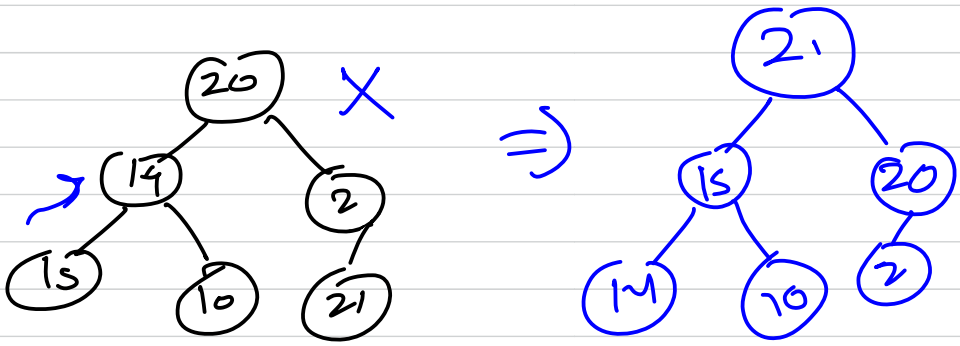
break

}

— HeapSort —

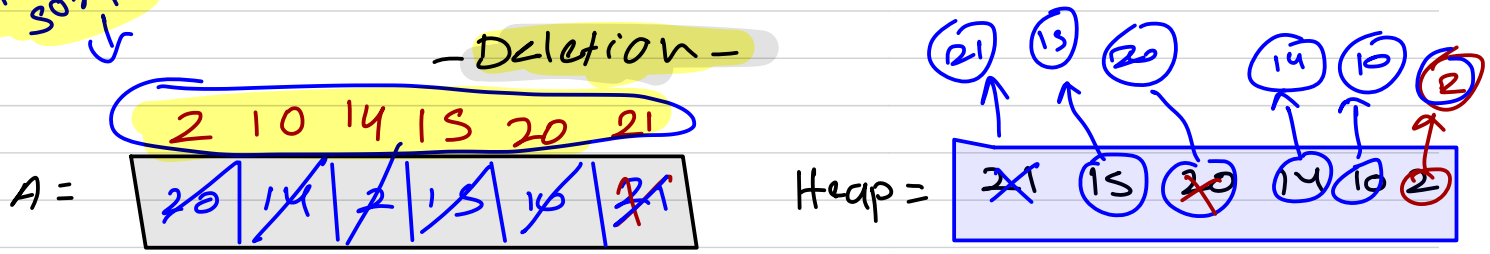
* Uses Heap Data Structure
* Insert element in the Heap
* Perform deletion until the heap is empty
* Store deleted elements from heap back into the array.

— Insertion —

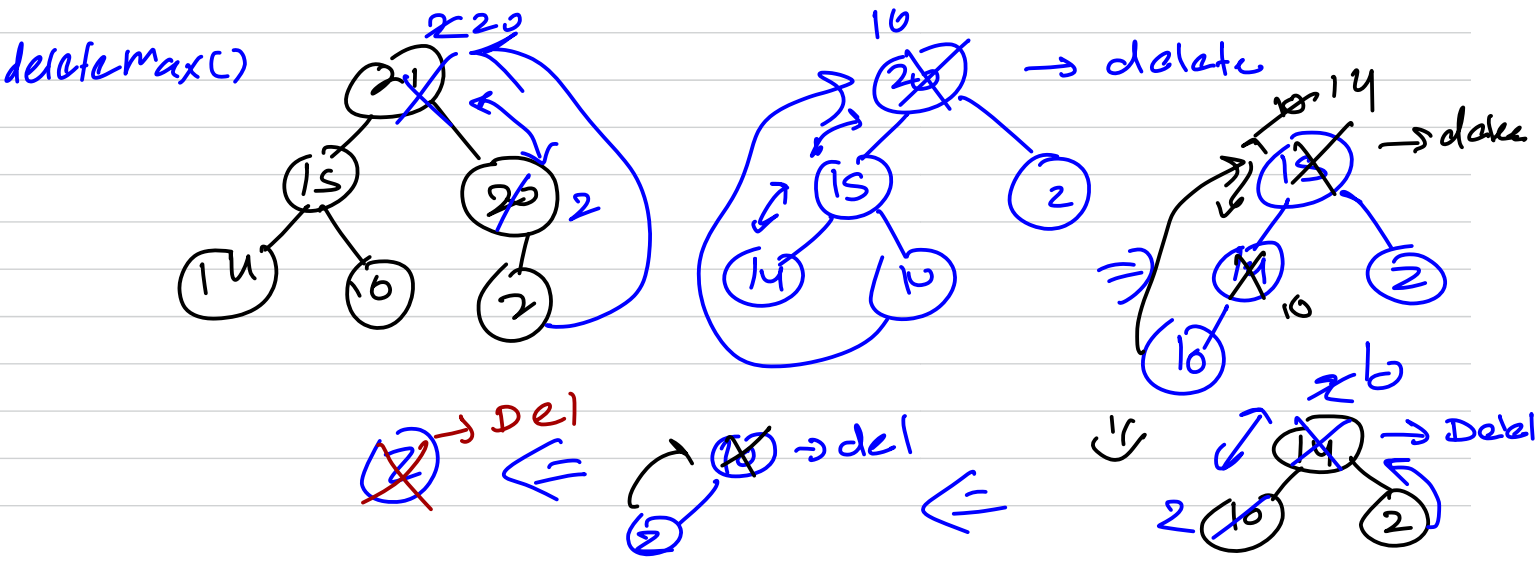


Heap sort

— Deletion —



deleteMax()



```
heapsort ( A, n )
{
    H = heap()
    for i=0, i<n, i++
        H.insert(A[i])
    k= n-1
    for i=0, i < H.current_size, i++
        A[k] = H.deletemax()
        k--
}
```

heapsort (A, n)

{

H = heap()

for i=0, i<n, i++ → O(n)

H.insert(A[i]) → O(log n)

O(n log n)

k= n-1

for i=0, i < H.current_size, i++ → O(n)

A[k] = H.deletemax() → O(log n)

O(n log n)

k--

}

= O(n log n)

Hashing

Hash Tables

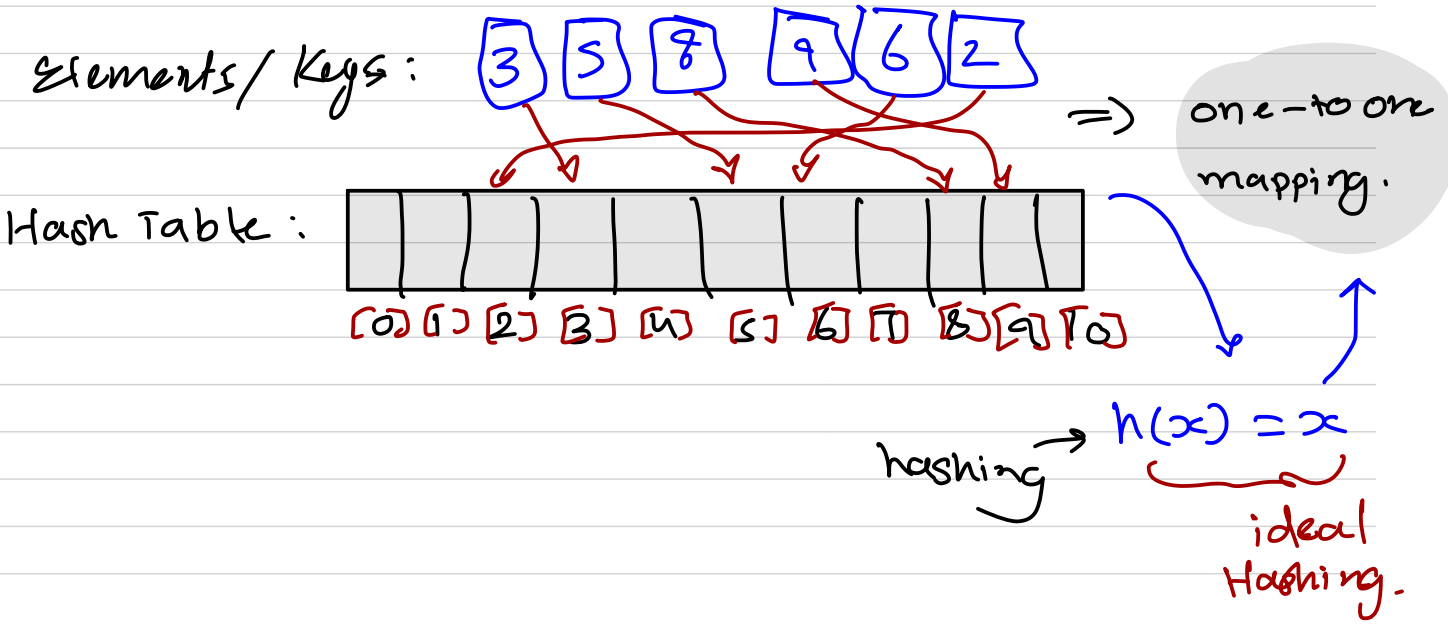
— Hashing —

- is a technique used for searching, inserting, deleting elements from a collection.

* Linear search: $O(n)$
* Binary Searh: $O(\log n)$
* BinarySearchTree: $O(h) \rightarrow O(\log n)$
* Hashing : $O(1)$

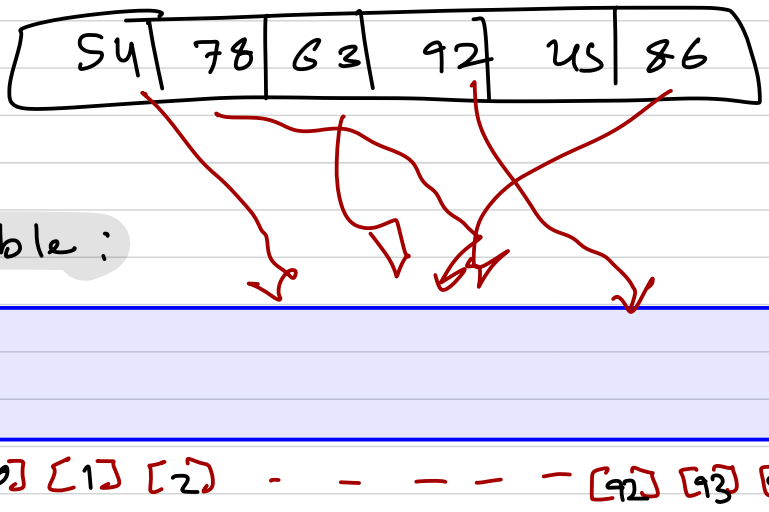
— Ideal Hashing —

* HashTable - is used to store the elements/data
* Hash functions - maps elements to corresponding indices.



eg.

Elements/Keys:



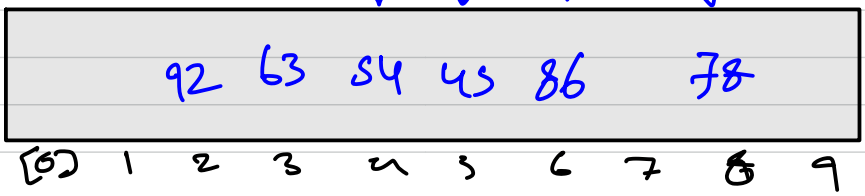
* uses a lot of space;

⇒ SOLUTION: Compression Hashing.

Elements/keys: 54 78 63 92 45 86 → %10

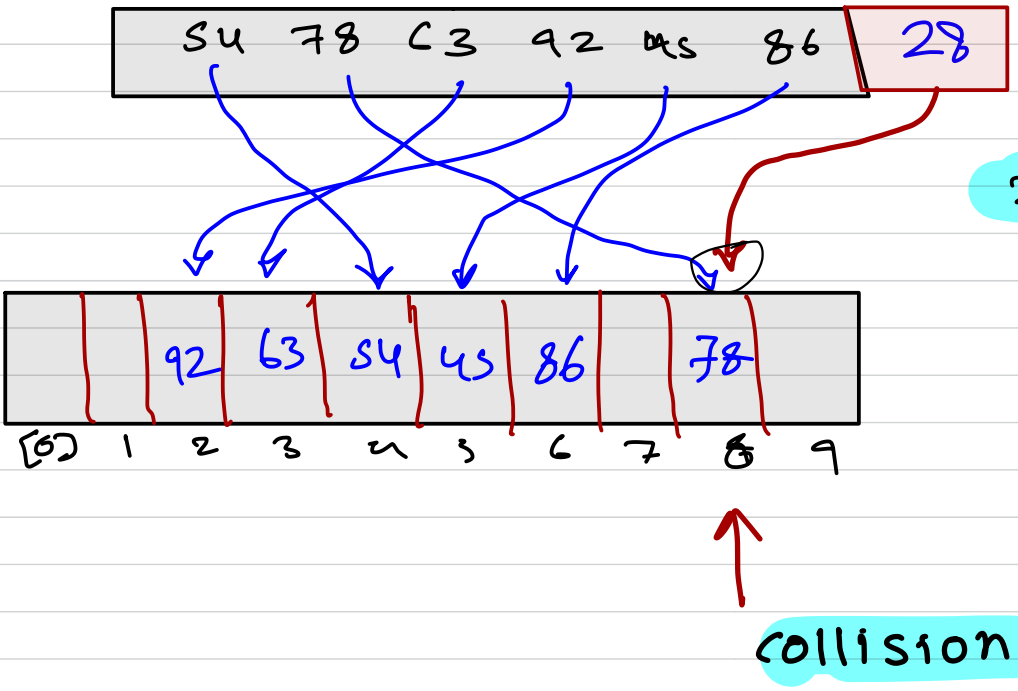
54%10 = 4

Hash Table:



* mapping → $h(x) = x \% 10$

* Search(54) ⇒ 54%10 = [4] ⇒ check index 4.



-Collision Handling Scheme-

Collision- when one or more than one key maps to the same index in hash table.

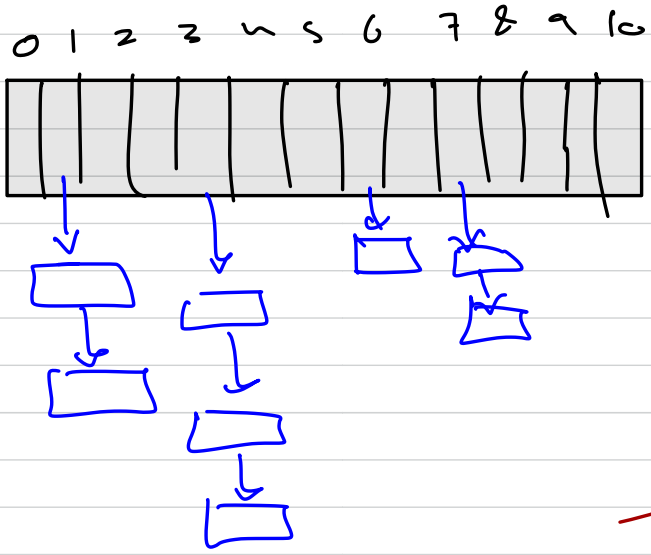
* Chaining - using auxilary list at each index to store collided keys.

* Open addressing: uses existing spaces available in Hash Table to store colliding keys.
  - a) Linear Probing
  - b) Quadratic probing
  - c) Double Hashing

Chaining - Collision Handling Scheme

* Chain uses auxiliary lists
* It's the simplest and efficient way of handling collisions.

- Hash Table =



- Example -

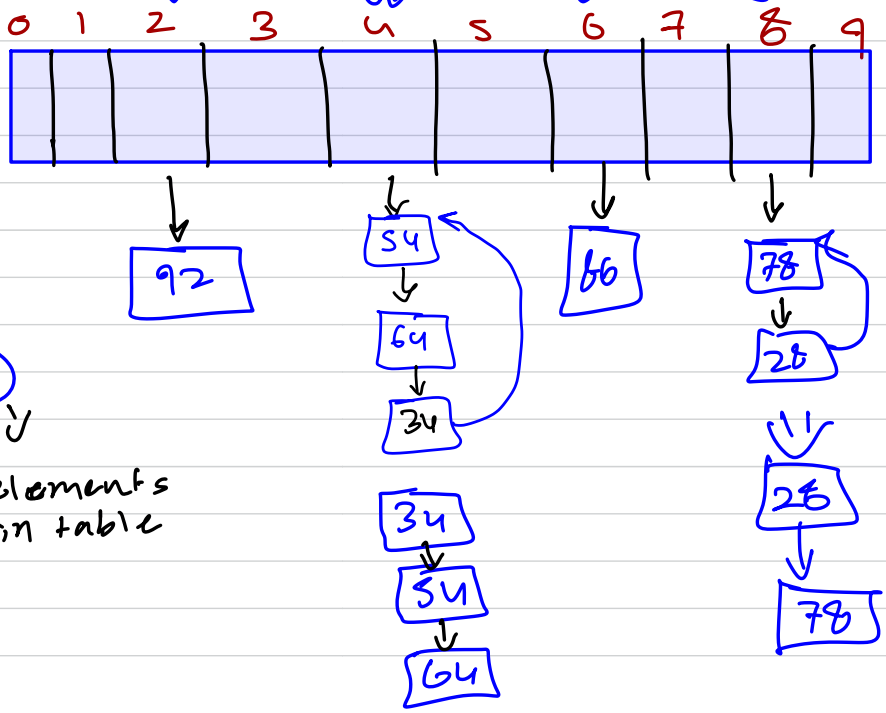
n → length of element

Elements = 54 78 64 92 34 86 28

h(x) = x % 10

s → length of table

Hash Table =



* Load factor : n/s

e.g. → n = 50, s = 10 } 50/10 = 5

5 elements in table

for time complexity

O(n/s)

* Load factor changes:

e.g. $h(x) = x \% 15$ → limits the size of Hash Tables.

then,

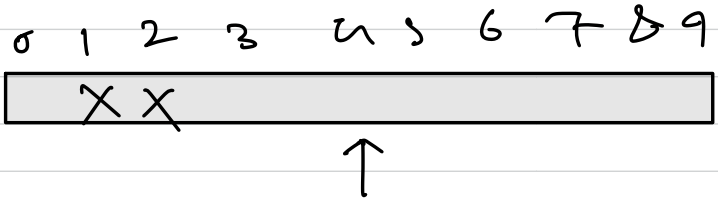
load factor = $n/15$

– Linear Probing –

* open addressing scheme

* Insert element in the next available index, if cell is already occupied.

HashTable =



$h(x) = x \% 10$

$h'(x) = (h(x) + i) \% 10$, for $i = 0, 1, 2 \ldots$

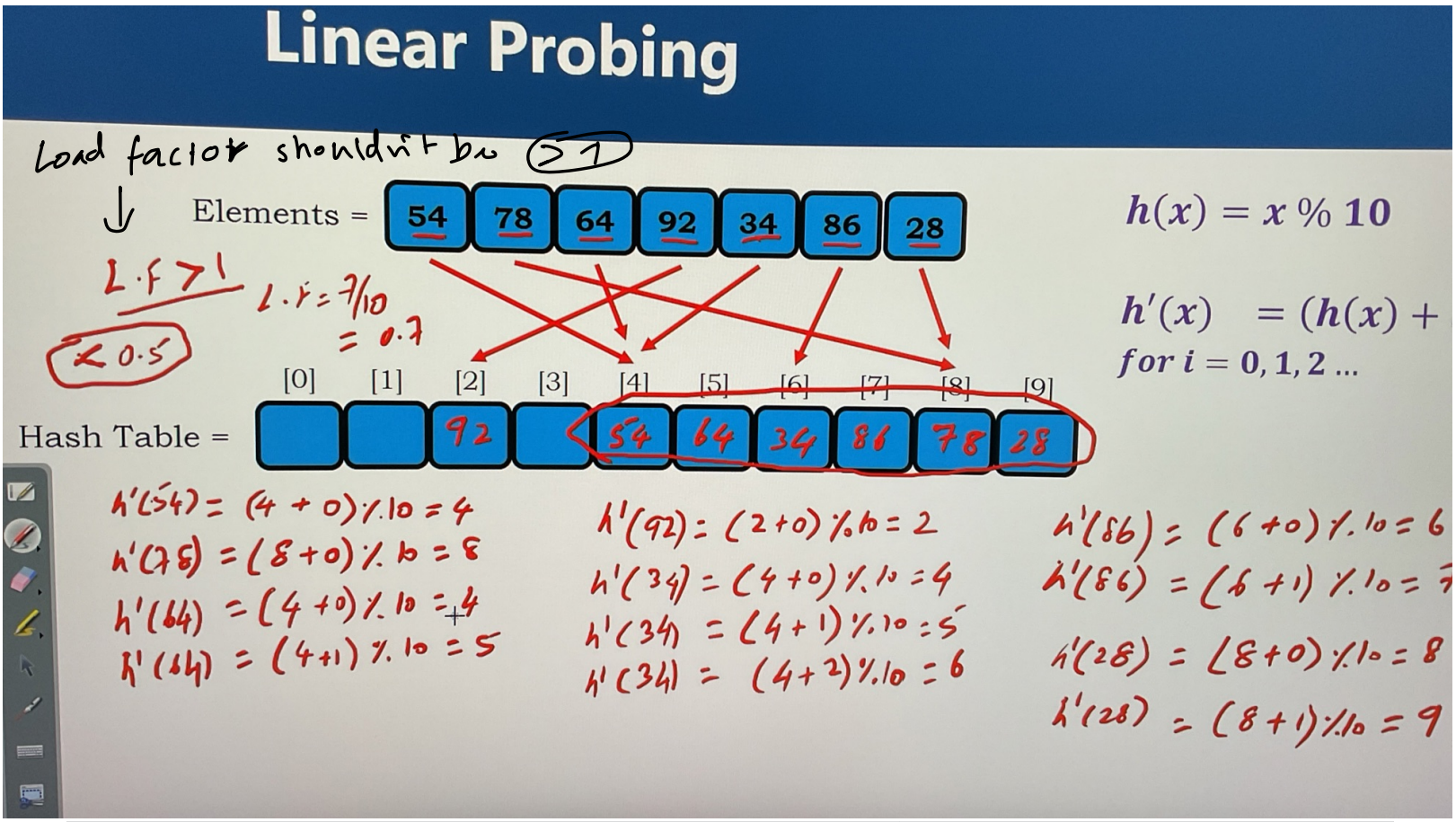
$h'(x) = (h(x)) \% 10$

$h'(x) = (h(x)+1) \% 10$

$h'(x) = (h(x)+2) \% 10$

$h'(x) = \ldots \vdots \ldots$

Linear Probing



- Quadratic Probing -

* open addressing scheme
* Insert element at the next Quadratically available index, if cell is already occupied.

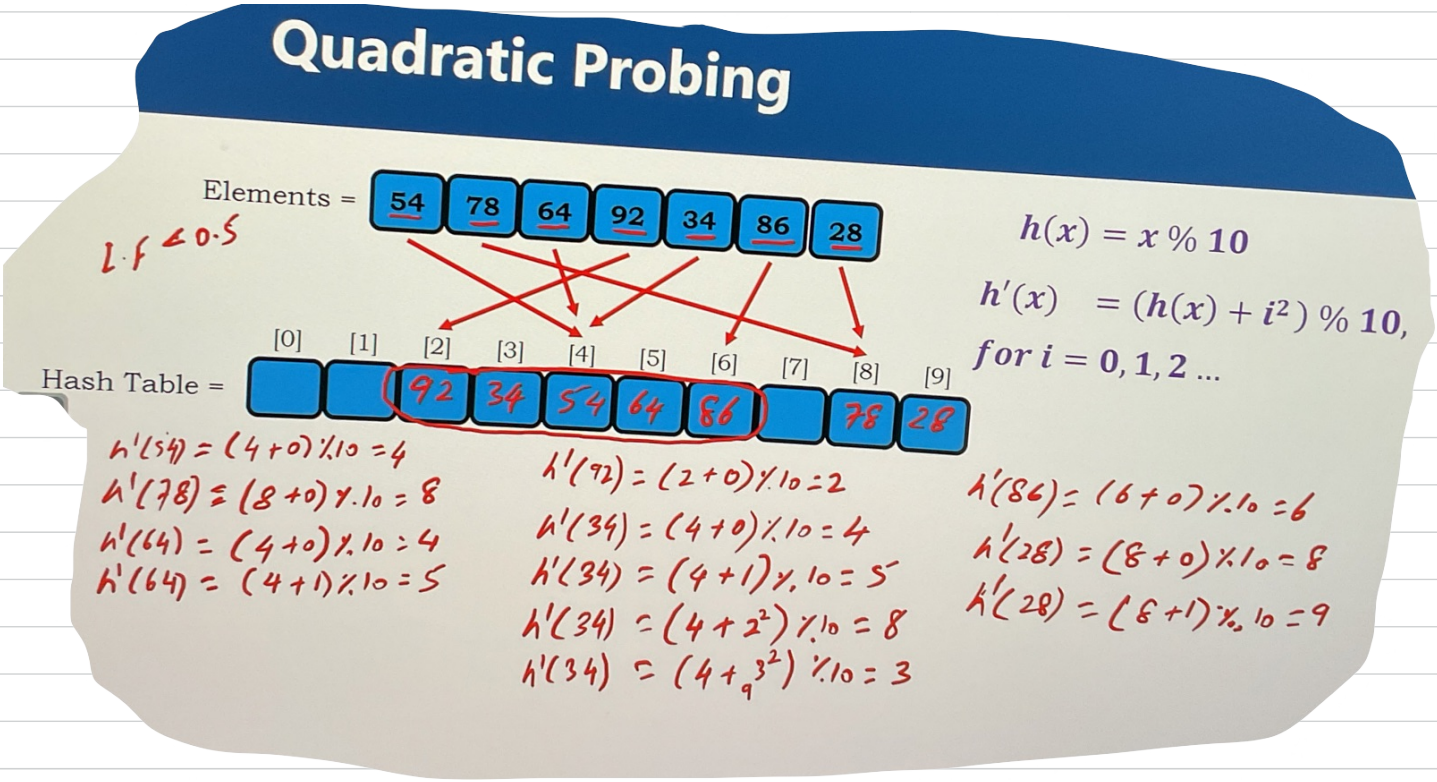
Hash Table =

0	1	2	3	4	5	6	7	8	9
		x	x						

$h(x) = x \% 10$

$h'(x) = (h(x) + i^2) \% 10$

for $i = 0, 1, 2$



- Double Hashing -

* open addressing scheme
* insert element using another hash function, if cell is already occupied.

$h_1(x) = x \% 10$

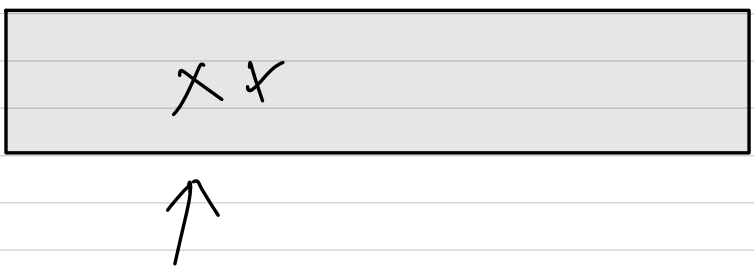
$h_2(x) = q - (x \% q)$

$h'(x) = (h_1(x) + i * h_2(x)) \% 10$

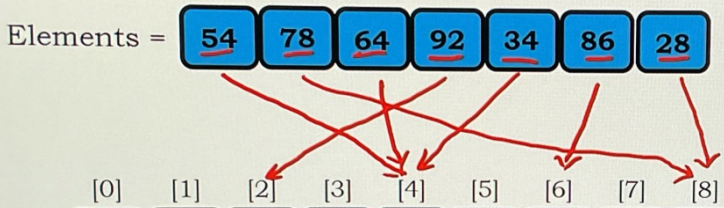
for $i = 0, 1, 2$

Hash table:-

0 1 2 3 4 5 6 7 8 9



Double Hashing



$h_1(x) = x \% 10$

$q = 7$

$h_2(x) = q - (x \% q) = 7 - (x \% 7)$

$h'(x) = (h_1(x) + i * h_2(x)) \% 10,$

$for\ i = 0, 1, 2 \ldots$

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Hash Table =	64		92		54	34	86		78	28

h'(54) = (4+0) % 10 = 4

h'(78) = (8+0) % 10 = 8

h'(64) = (4+0) % 10 = 4

h'(64) = (4 + 1*(7 - 64%7)) % 10

= (4 + 7 - 1) % 10 = 0

h'(28) = (8 + 2×7) % 10 = 9

h'(92) = (2+0) % 10 = 2

h'(34) = (4+0) % 10 = 4

h'(34) = (4 + 1*(7 - 34%7)) % 10

= (4 + (7 - 6)) % 10

= 5

h'(86) = (6 + 0) % 10 = 6

h'(28) = (8 + 0) % 10 = 8

h'(28) = (8 + 1×(7 - 28%7)

= (8 + 7 - 0) % 10 = 5

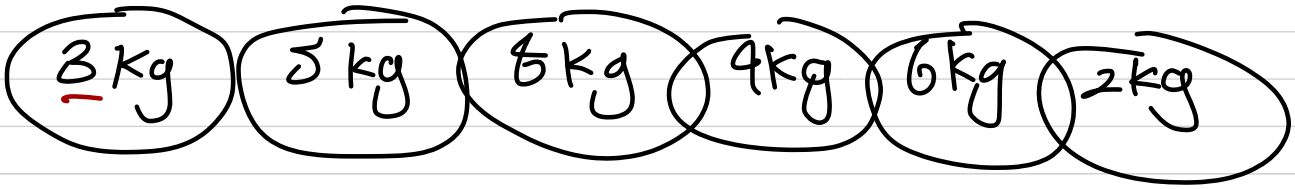
h'(28) = (8 + 2(7 - 28%7)) % 10

= 8 + 2×(7 - 0) = (8 + 14) % 10 = 2

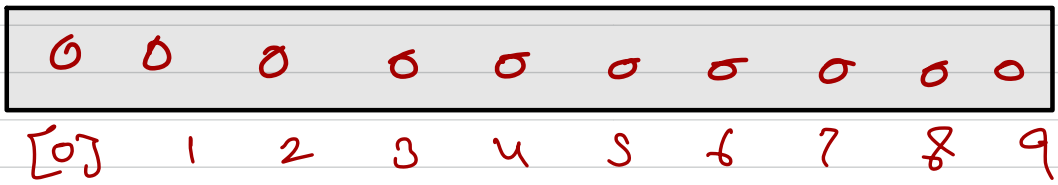
4 comparisons for 28

Index Based Sorting Algorithm

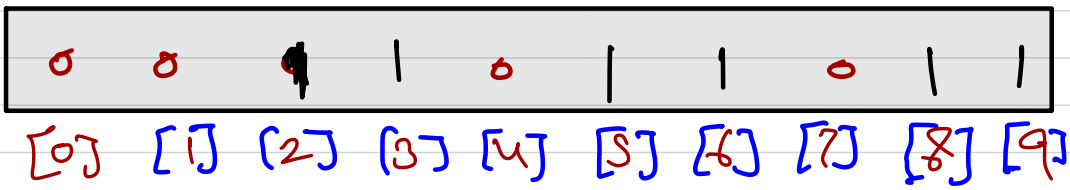
– Count Sort –



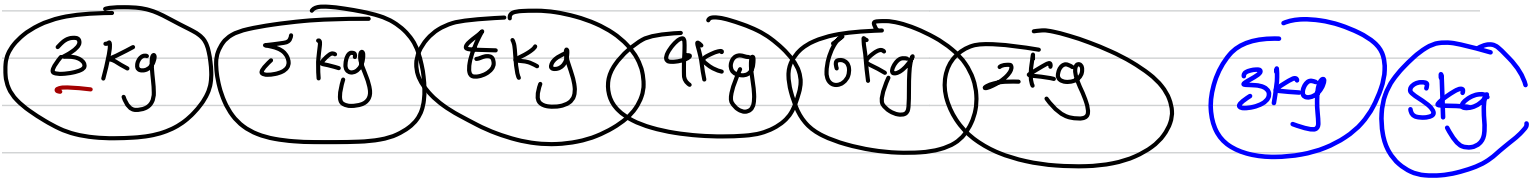
Count =



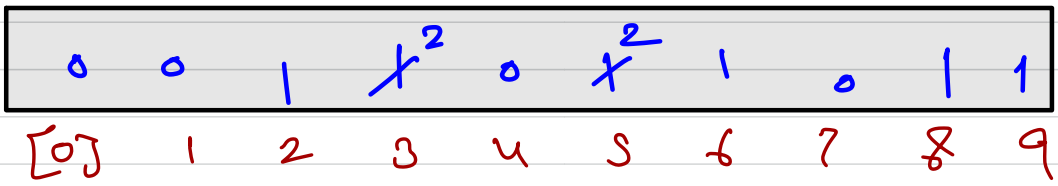
Count =



2 3 5 6 8 9 ⇒ sorted.



Count =



2 3 3 5 5 6 8 9

```
countSort(A, n)
{
    maxsize = max(A)

    carray = [0, 0 ..... maxsize + 1]

    for i=0, i<n, i++                          } n times
        carray[A[i]] = carray[A[i]] + 1

    i, j = 0

    while i < maxsize + 1                      } m times
        if carray[i] > 0
            A[j++] = i
            carray[i] = carray[i] - 1
        else
            i++
}
```

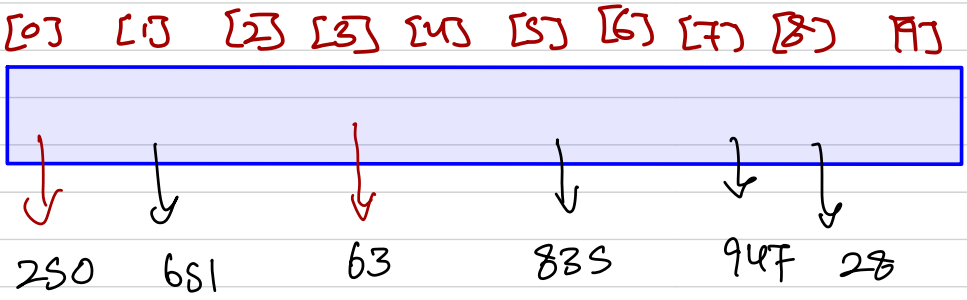
$O(n+m) \Rightarrow$

$O(n)$

- Radix Sort -

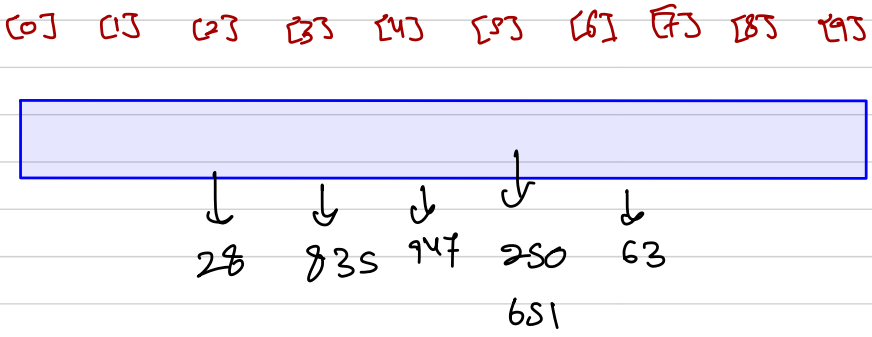
1) 63 250 833 947 651 28

* start from least significant digit.

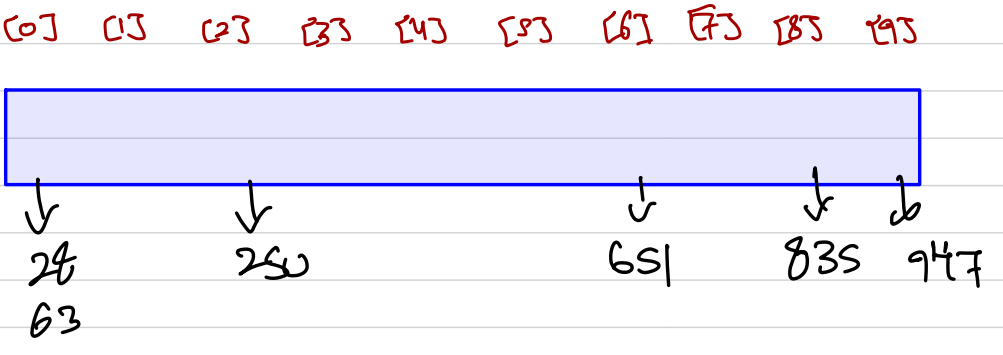


2) 250 651 63 833 947 28

* select the second digit



3) 028 833 947 250 651 063



* 3rd digit

28 63 250 650 833 947

```
radixsort ( A , n )
{
    maxelement = max (A)
    digits = finddigits (maxelement)
    bins = [] // array of size 10

    for i=0, i<digits, i++          } n times
        for j=0, j<n, j++
            e = (A[j] / pow(10,i)) % 10
            bins[e].append(A[j])

    k=0
    for x=0, x<10, x++              } 10 times
        A[k] = bins[x].remove();
        k++
}
```

n times

d x n times

= O(d*n)

⇒ O(n)

_ Bucket Sort _

* uses an array of buckets to perform sorting
* insert elements in the buckets according to index computed.
* Sort non-empty bucket using insertion sort
* Traverse buckets in sequence and store elements back into array.

* formula

$$index = \lfloor n * e / (max + 1) \rfloor$$

$$index = \lfloor 6 * e / (92+1) \rfloor$$

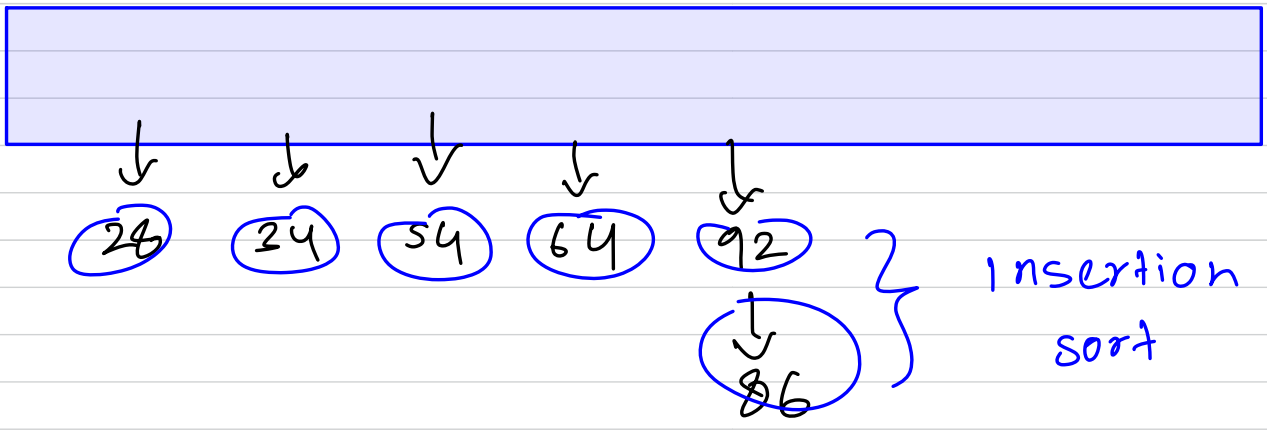
n = 6

max = 92

34 54 92 66 64 28

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

Buckets =



index =

$\lfloor 6 \times 34 / (92+1) \rfloor = 2$

= 28 34 54 64 86 92

```
bucketSort(A, n)
{
    max = maximum(A)

    buckets = []

    for i=0, i<n, j++
        j= n * A[i] / (max +1)
        buckets[j] = A[i]

    for i=0, i<10, i++
        insertionsort(buckets[i])

    k=0
    for i=0, i<10, i++
        A[k] = buckets[i].remove()
        k++;
}
```

O(n)

$\rightarrow O(n^2)$

$= O(n^2)$

Graphs

- Graphs Intro -

* Graphs represents relationship between objs.
* collection of objects along with pairwise connection between objects.

- Definition -

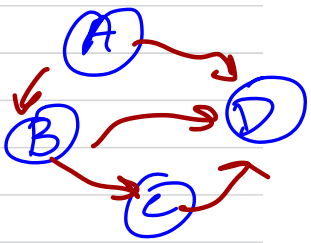
* is a collection of object called as Vertices and together with relationship between them called as Edges.

Graph (G) = { V, E }

V → vertices

E → Edges

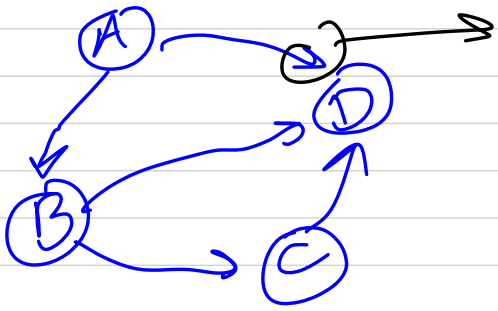
* Each edges in the Graph joins two vertices



* Vertices (V) = { A, B, C, D }
* Edges (E) = { AB, AD, CB, BD, CD }

* **Directed Edge:** An edge (u, v) is directed if pair (u, v) is ordered, with u preceding v.

Edge is oriented or Direction.



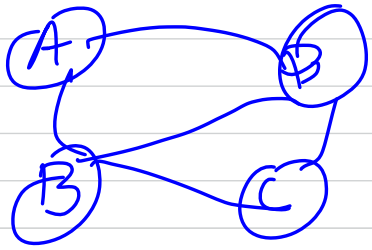
Directed Graph

or

Digraph

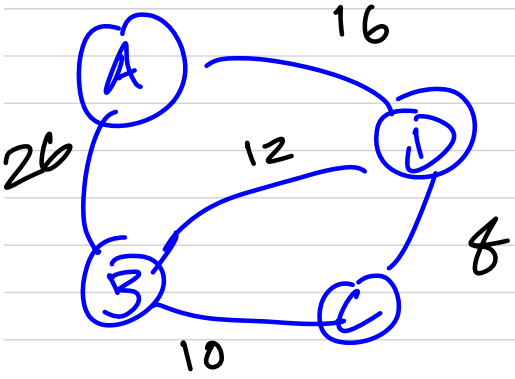
* **Undirected Edge:** (u, v) is undirected if pair (u, v) is not ordered.

Edge has no orientation.

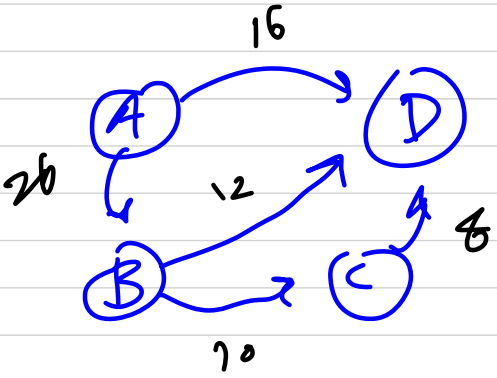


undirected

* **weighted Edge:** cost or weight is assigned to each edge (u, v)

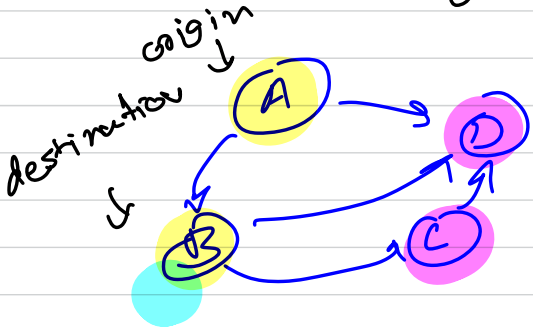


weighted undirected Graph



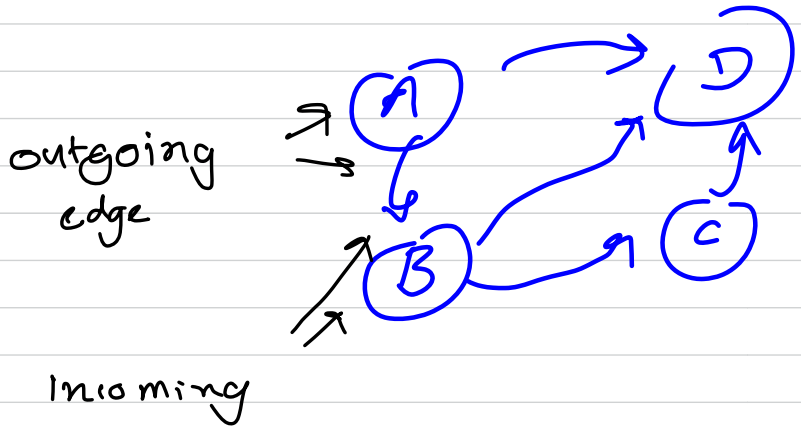
weighted Directed Graph

* **End vertices:** Two vertices joined by an Edge
* **Adjacent vertices:** Two vertices are adjacent if there is an edge between them.
* **Incident Edge:** if vertex is one of the end points.

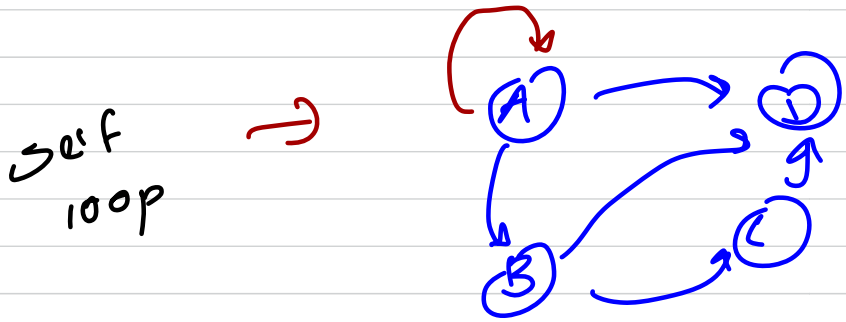


* Outgoing Edge : origin is the vertex

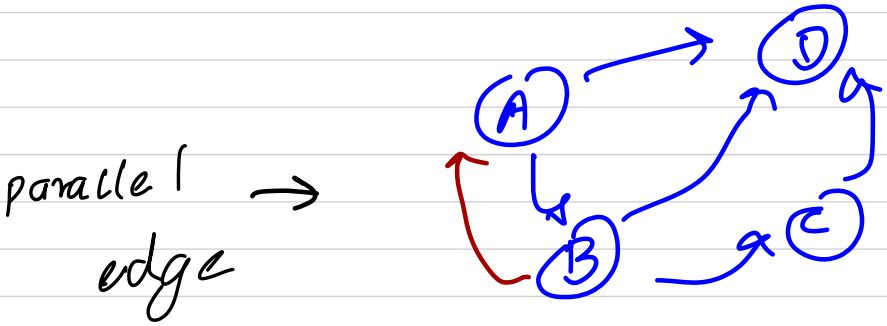
* Incoming edge : destination is the vertex



* Self-loop : if the two end points are same

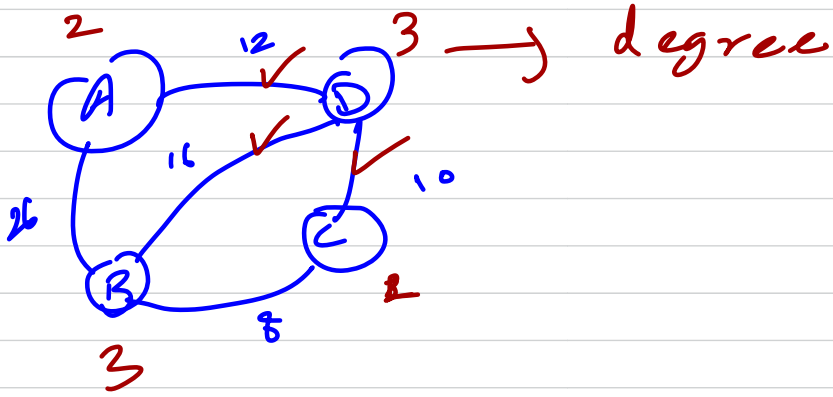


* parallel edge - Edge from u to v (u, v) as well as an edge from v to u (v, u)



– Degree of a vertex –

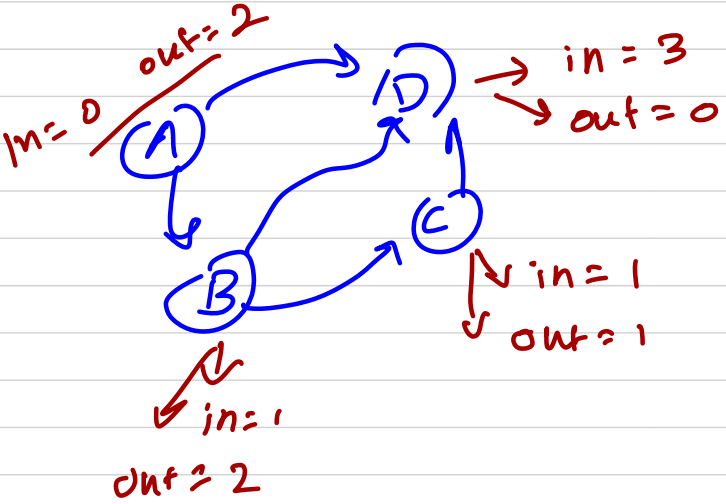
– deg (v) : number of Edges



→ degree

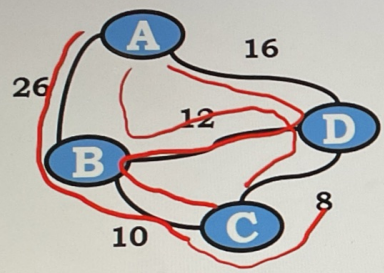
* In-degree – indeg (v) : number of incoming edges.

* out-degree – outdeg (v) : number of outgoing edges.



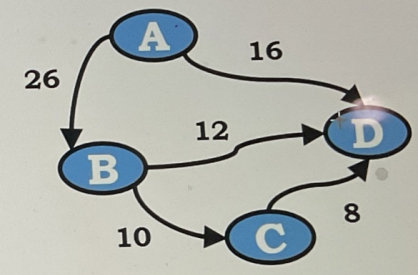
Path & Cycles

Path: sequence of edges starting at one vertex and ending at another vertex.



Weighted Undirected Graph

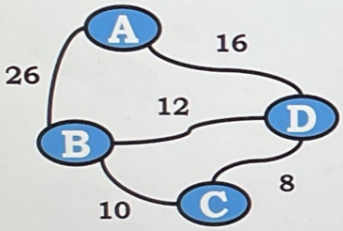
A-B-C , A-B-D-C
A-B-C-D, A-D-B-C



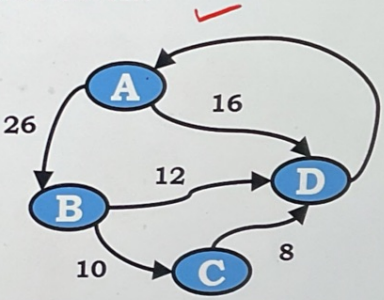
Weighted Directed Graph

A-B-C , A-B-D
A-B-C-D, A-D

Cycle: path that starts and ends at the same vertex

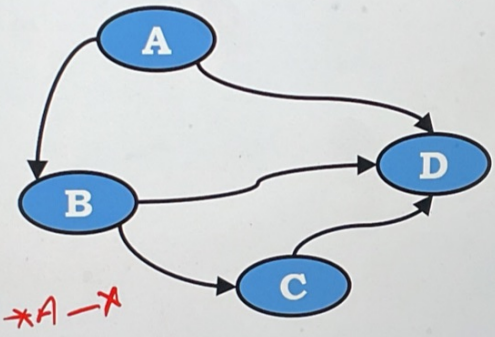


→ A-B-C-D-A
→ A-B-D-A



→ A-B-D-A
→ A-B-C-D-A
→ B-C-D-A-B

Directed Acyclic Graph: when there are no cycles in a directed graph

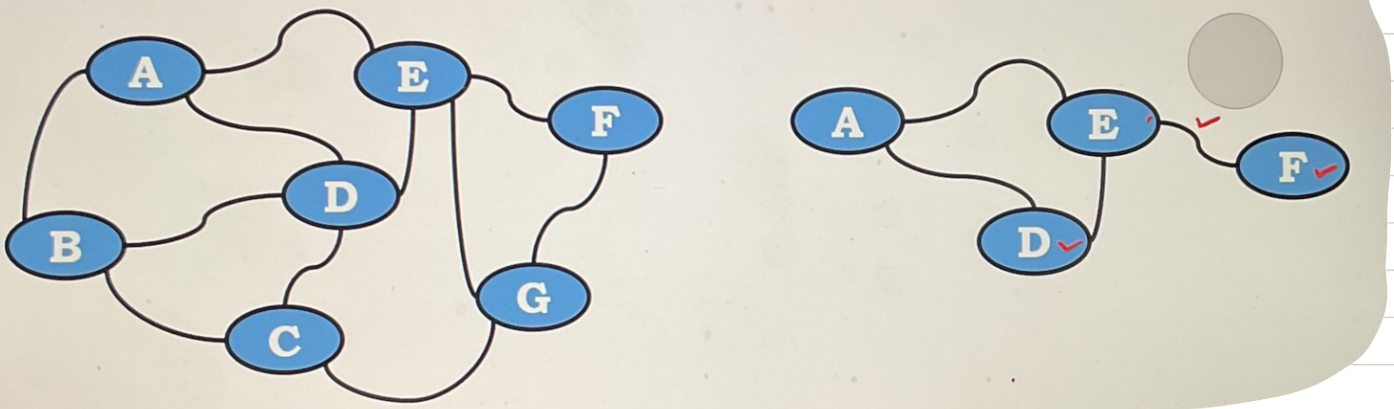


A-B-D ✗ A ✗

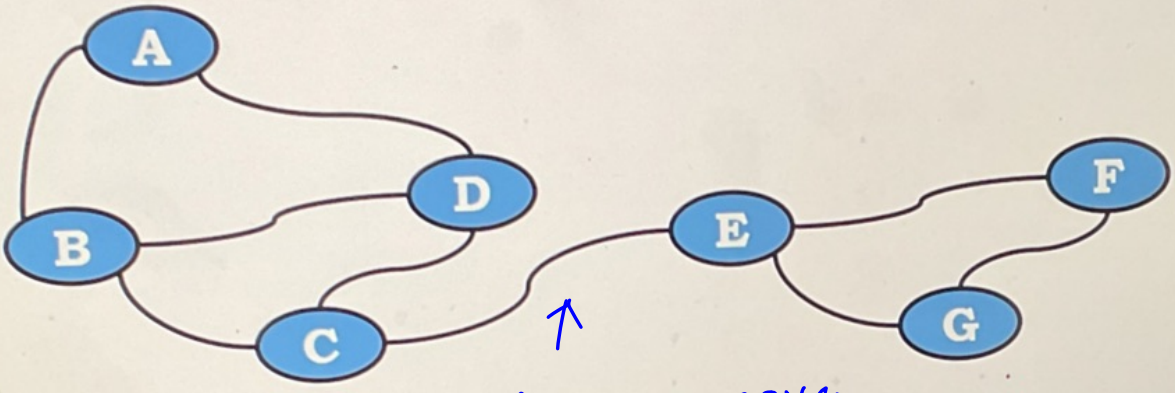
B-D ✗ B ✗
B-C-D ✗ B ✗

– Subgraphs & connected components –

Subgraph: whose vertices and edges are subsets of vertices and edges of another graph

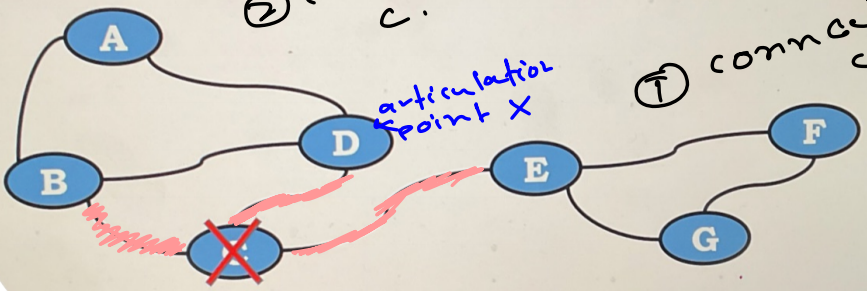


Connected components: connected subgraphs are known as connected components



if we remove that edge, then, it's not connected

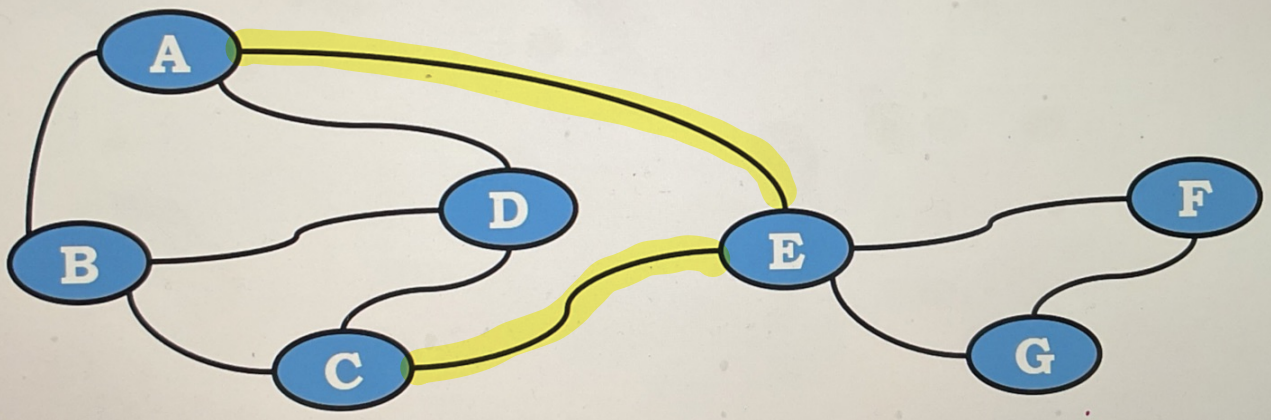
Articulation Point: vertex whose removal results in connected components



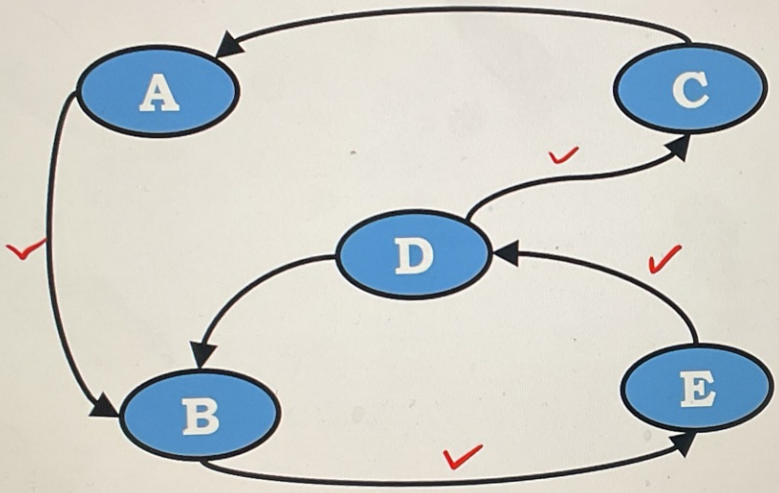
② connected c.

① connected c.

A **Bi-connected components:** components connected by two edges



A **Strongly Connected Graph:** all the vertices are reachable from any vertex



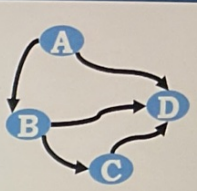
A → B → E → D → C

- Graphs - Abstract Data Type -

* Create(n): Creates graph with (n) vertices & no edges.
* Insert_edge(u,v, w=1): Edges from (u to v) storing weight=1 (create)
* remove_edge(u,v): deletes edge from u to v
* exist_edge(u,v): returns true if edge exists between u & v, else false.
* vertex_count(): returns no. of vertices
* edge_count(): // // of edges
* vertices(): returns all the vertices
* edges(): // // the edges
* degree(u): returns the degree of (u) vertex
* indegree(u): // // indegree of (u)
* outdegree(u): // // out degree of (u)

Graphs – Representation

A Graph can be represented using different data structures



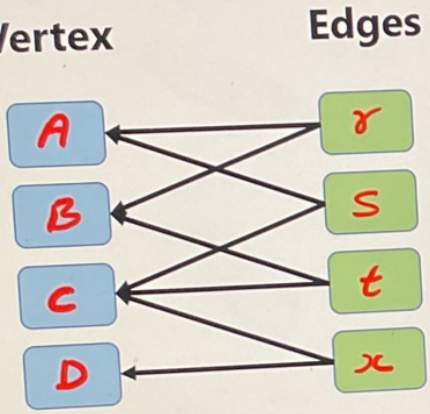
A **Edge List :** Maintains list of all edges

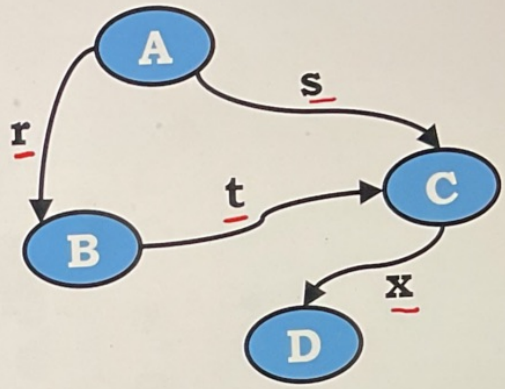
B **Adjacency List :** For each vertex, separate list of edges is maintained

C **Adjacency Matrix :** Maintains a matrix of vertices, where each cell stores the reference to the edge.

- Edge List -

Edge List : Maintains list of all edges

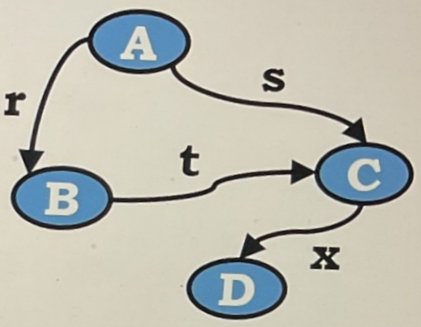




Weighted Directed Graph

Vertices: n **Edges: m**

Operation	Time Complexity
insert_edge(u, v, w=1)	*O(1)*
remove_edge(u, v)	*O(1)*
exist_edge(u, v)	*O(m)*
vertex_count()	*O(1)*
edge_count()	*O(1)*
vertices()	*O(n)*
edges()	*O(m)*
degree(u)	*O(m)*

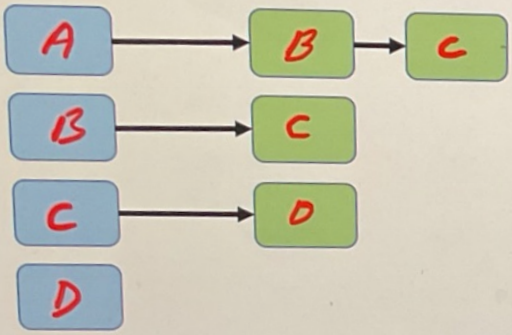


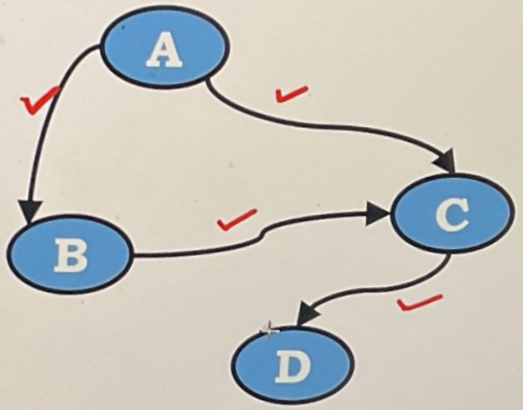
Space Complexity: ***O(n+m)***

- Adjacency List Rep -

B **Adjacency List :** For each vertex, separate list of edges is maintained

Vertices

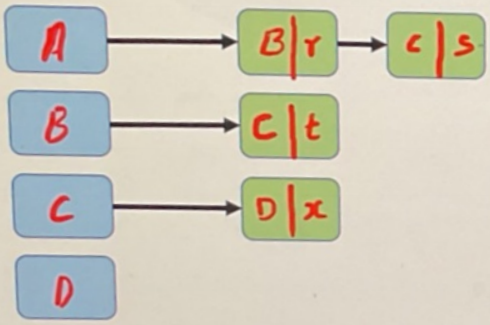


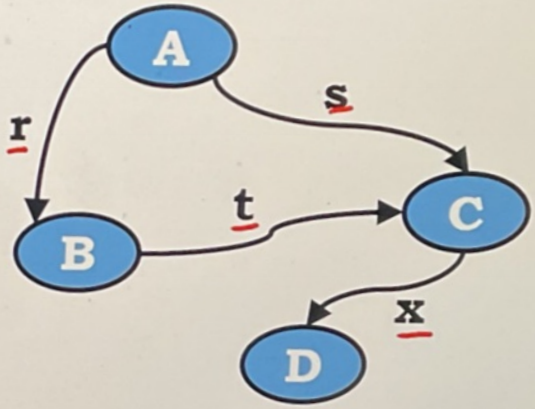


Directed Graph

B **Adjacency List :** For each vertex, separate list of edges is maintained

Vertices

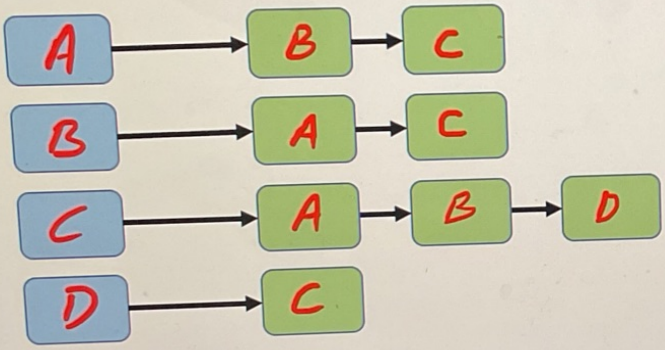


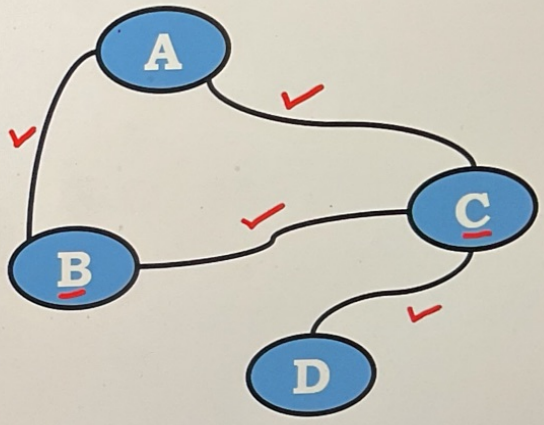


Weighted Directed Graph

B **Adjacency List :** For each vertex, separate list of edges is maintained

Vertices

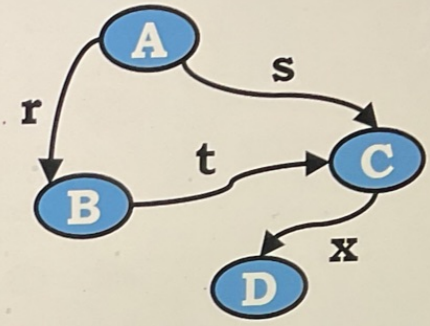




Undirected Graph

Vertices: n Edges: m

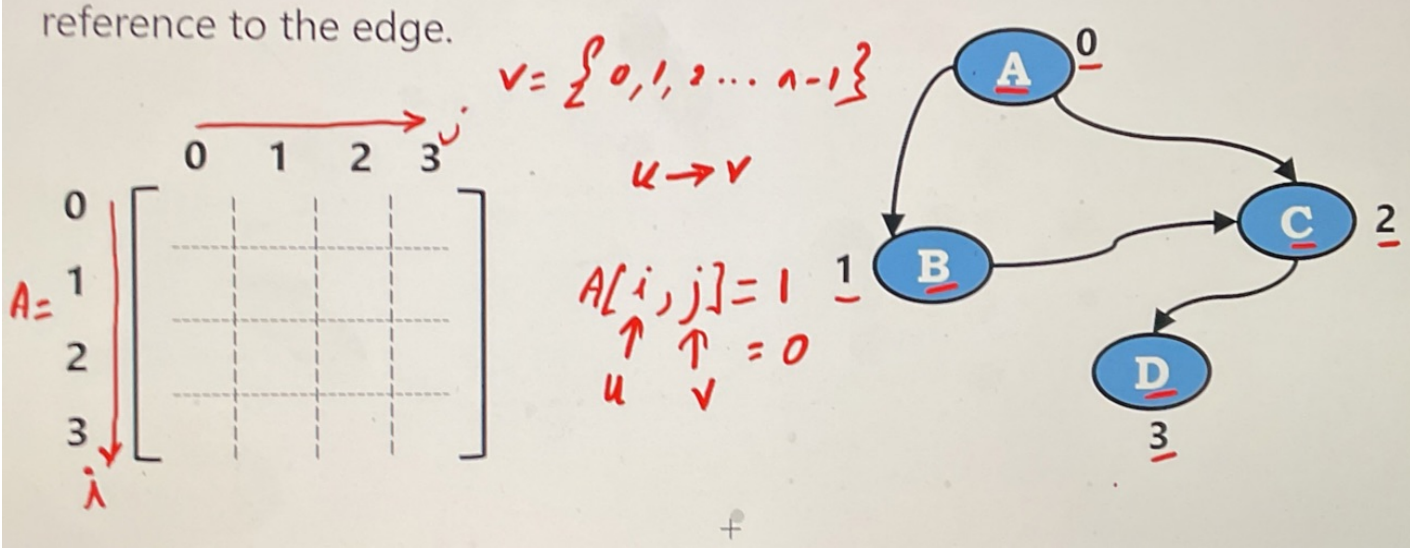
Operation	Adjacency List
insert_edge(u, v, w=1)	$O(1)$
remove_edge(u, v)	$O(1)$
exist_edge(u, v)	$O(min(d_u, d_v))$
vertex_count()	$O(1)$
edge_count()	$O(1)$
vertices()	$O(n)$
edges()	$O(m)$
degree(u)	$O(1)$



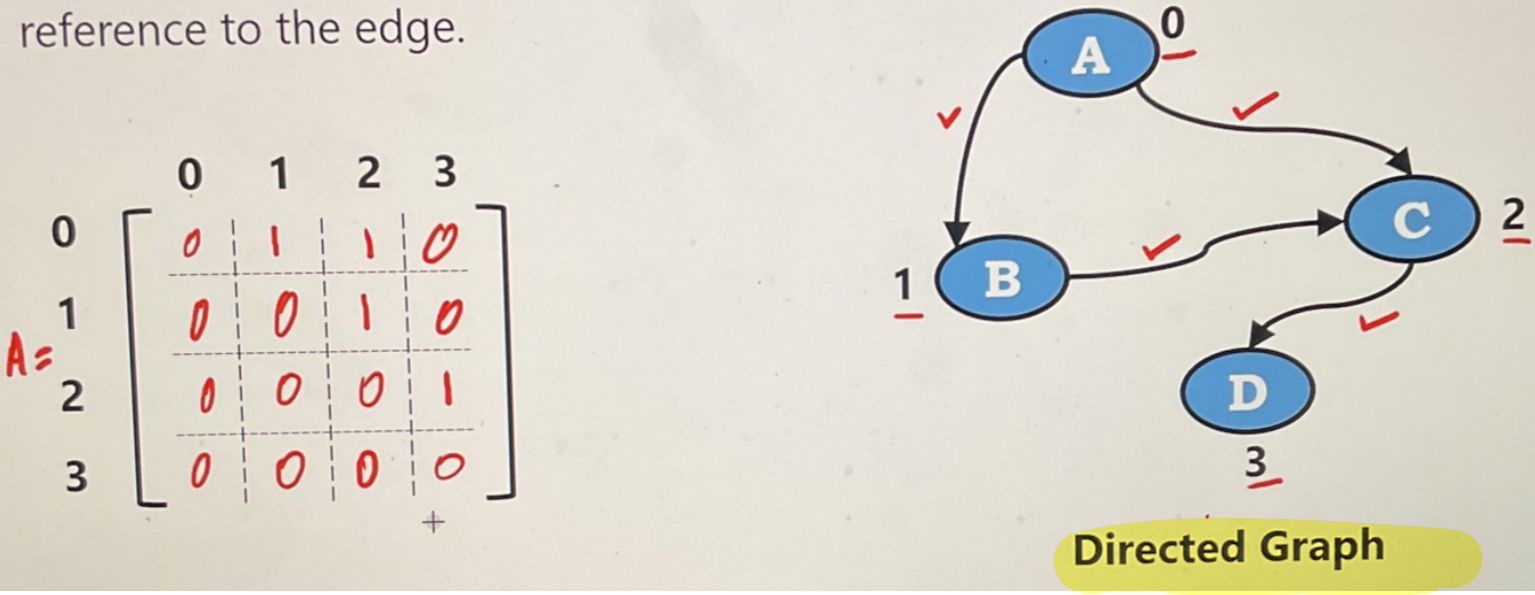
Space Complexity: $O(n+m)$

- Adjacency Matrix Representation -

Adjacency Matrix : Maintains a matrix of vertices, where each cell stores the reference to the edge.



Adjacency Matrix : Maintains a matrix of vertices, where each cell stores the reference to the edge.

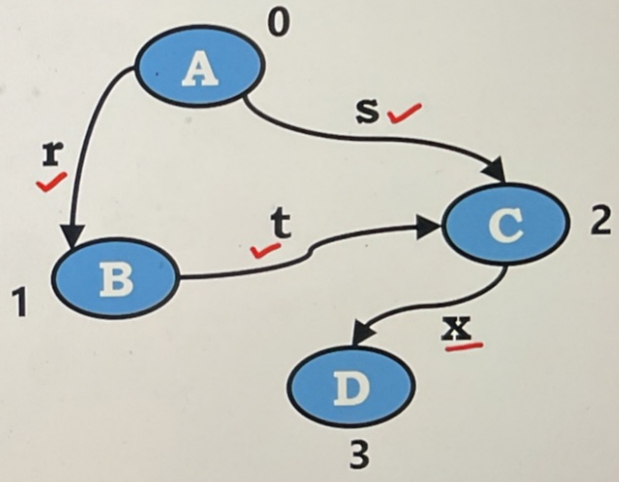


Directed Graph

Adjacency Matrix : Maintains a matrix of vertices, where each cell stores the reference to the edge.

	0	1	2	3
0	0	r	s	0
1	0	0	t	0
2	0	0	0	x
3	0	0	0	0

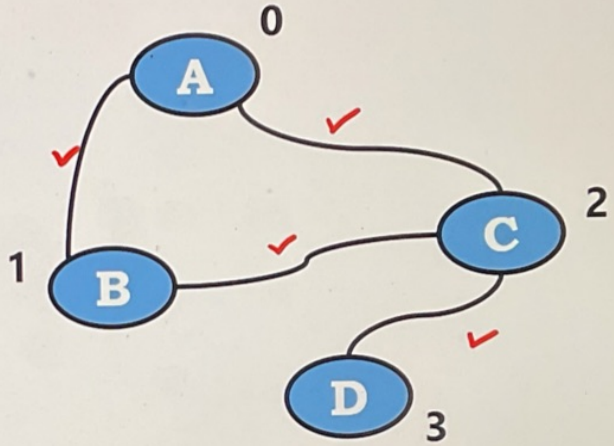
A=



Weighted Directed Graph

Adjacency Matrix : Maintains a matrix of vertices, where each cell stores the reference to the edge.

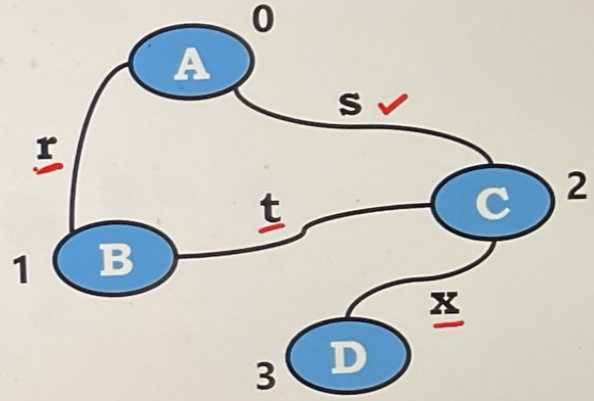
	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0



Undirected Graph

Adjacency Matrix : Maintains a matrix of vertices, where each cell stores the reference to the edge.

	0	1	2	3
0	0	r	s	0
1	r	0	t	0
2	s	t	0	x
3	0	0	x	0

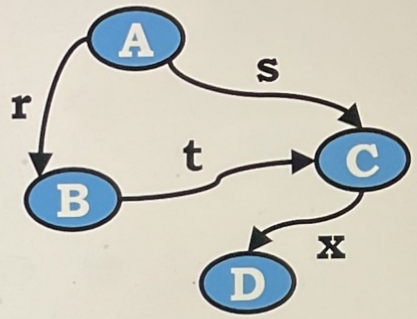


Weighted Undirected Graph

_ Performance

Vertices: n Edges: m

Operation	Adjacency Matrix
insert_edge(u, v, w=1)	O(1)
remove_edge(u, v)	O(1)
exist_edge(u, v)	O(1)
vertex_count()	O(1)
edge_count()	O(1)
vertices()	O(n)
edges()	O(m)
degree(u)	O(n)



→ Space Complexity: $O(n \times n)$

$= O(n^2)$

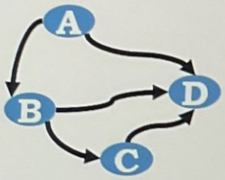
Graphs – Summary of Performance

	Edge List	Adjacency List	Adjacency Matrix
Space Complexity	O(n + m)	O(n + m)	$O(n^2)$

Operation	Edge List	Adjacency List	Adjacency Matrix
insert_edge(u, v, w=1)	O(1)	O(1)	O(1)
remove_edge(u, v)	O(1)	O(1)	O(1)
exist_edge(u, v)	O(m)	$O(\min(d_u, d_v))$	O(1)
vertex_count()	O(1)	O(1)	O(1)
edge_count()	O(1)	O(1)	O(1)
vertices()	O(n)	O(n)	O(n)
edges()	O(m)	O(m)	O(m)
degree(u)	O(m)	O(1)	O(n)

– Breadth first Search –

A. Breadth-First Search subdivides the vertices into levels and proceeds in rounds

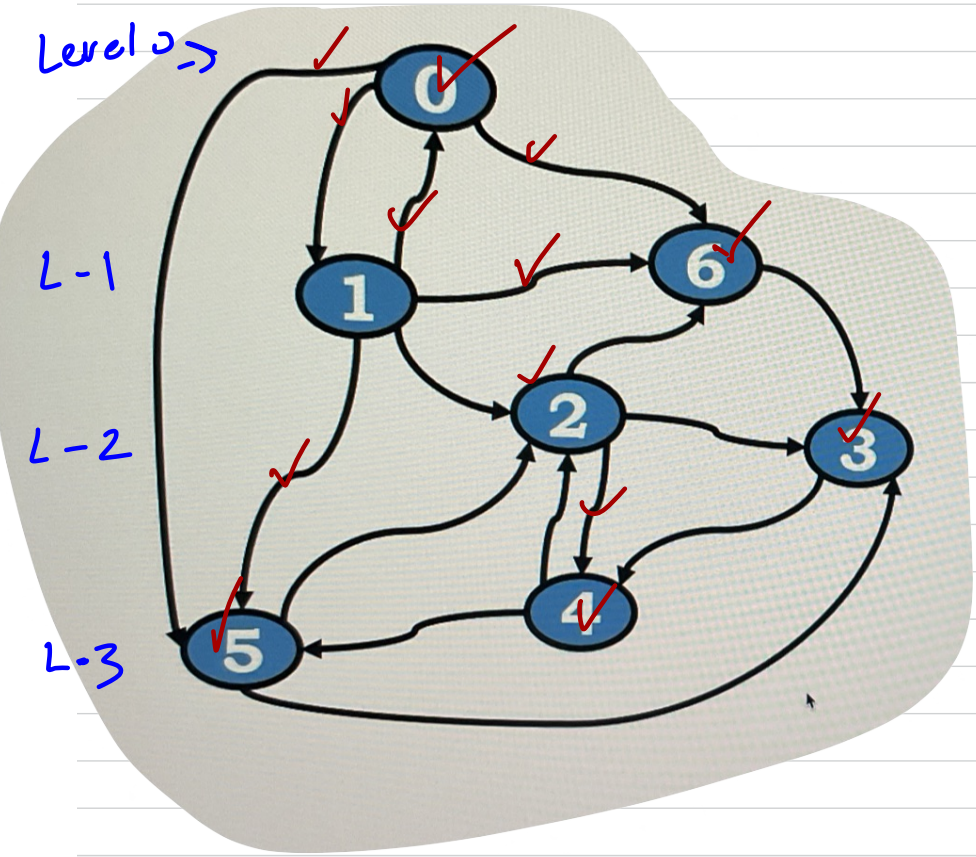


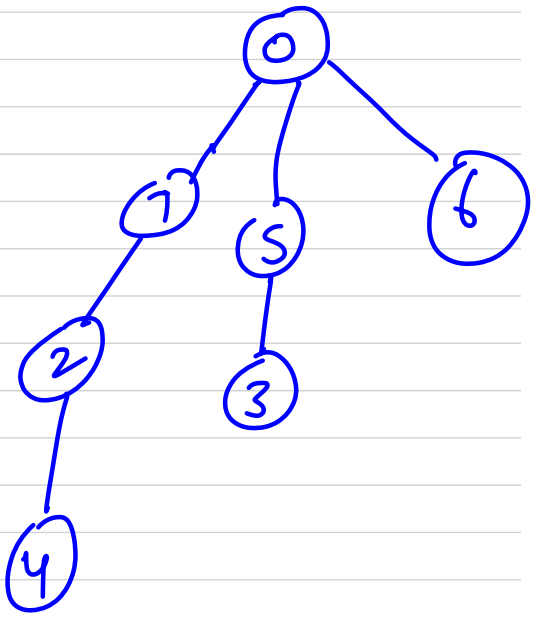
B. Starts at a vertex, which is considered at level 0

C. Identifies all the vertices reachable from starts vertex at level 1, marks them visited

D. In next round, identifies new vertices reachable from level 1 vertices which are not yet visited, marks them visited

E. This process continues until no vertices are found



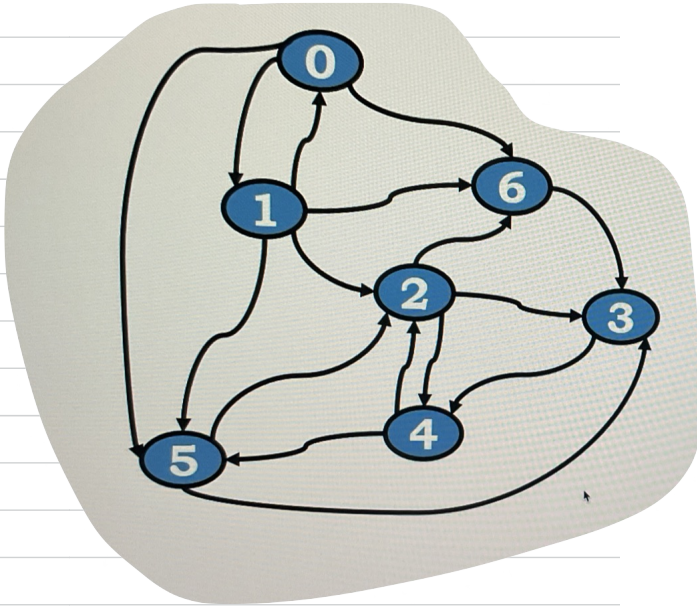


Bfs: 0 1 5 6 2 3 4

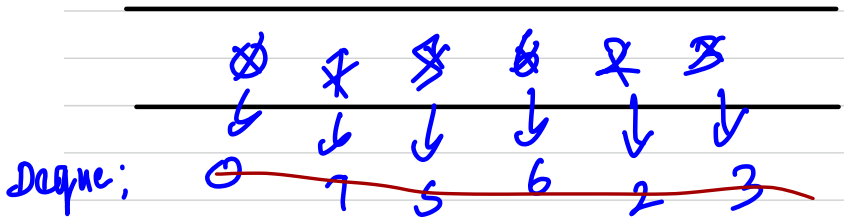
Breadth-First Search

```
function BFS (s)
    i = s , q  = Queue() , visited [] = [0,0,0.....n-1]
    print(i)
    visited[i] = 1
    q.enqueue(i)
    while  ! q.isempty() then
        i = q.dequeue()
        for ( j = 0, j < n, j++)
            if adjmat[i][j]==1 && visited[j] == 0 then
                print(j)
                visited[j] = 1
                q.enqueue(j)
```

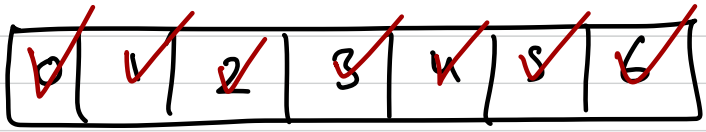
Start vertex = 0



Queue:



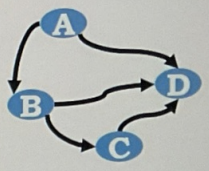
Visited:



BFS: 0 1 5 6 2 3 4

- Depth - First Search -

Depth-First Search

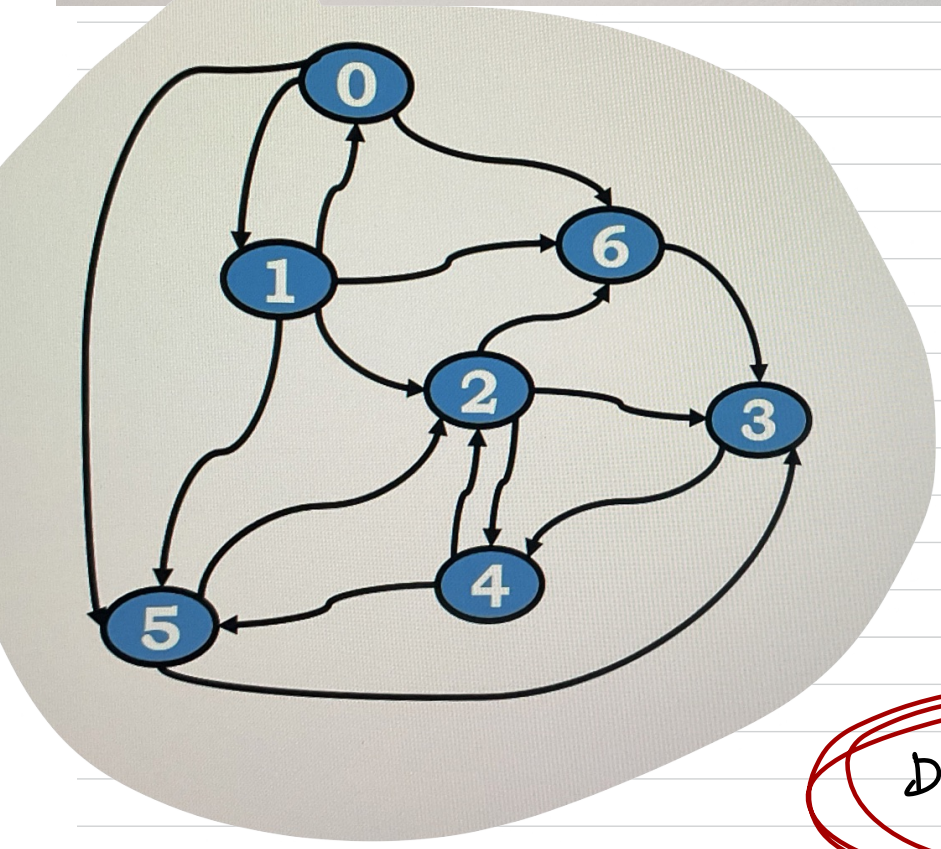


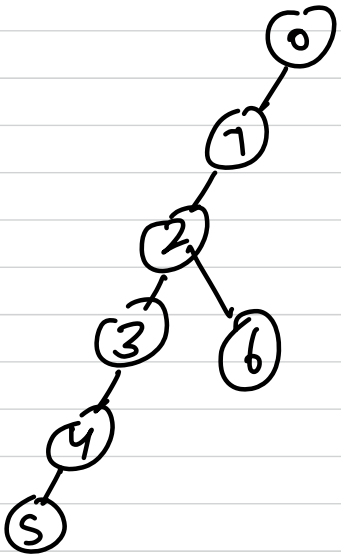
A Depth-First Search starts at a vertex

B Selects the adjacent vertex from the start vertex

C Visit the adjacent vertex, mark as visited

D Continue the above procedure, until there are no more unexplored edges, then terminate





Dfs: 0 1 2 3 4 5 6

```
function DFS (s)
    if visited[s] == 0 then
        print(s)
        visited[s] = 1
        for ( j = 0, j < n, j++)
            if adjmat[s][j]==1 && visited[j] == 0 then
                DFS(j)
```


Backtracking

TT, 2024

Uni staffs

SIT 221, Data Structure & Algorithm

- Algorithm Analysis — As the size of a problem grows for an algorithm:

  . Time : How much longer does it run?

  . Space : How much memory does it use?

- Running Time T(n) :

Running Time T(*n*)

The **running time** of a given algorithm is **the number of elementary operations** to reach a solution, for example

- Sorting – number of array elements
- Arithmetic operation – number of bits
- Graph search – number of vertices and edges

We can evaluate running time by

- Operation Counting
- Asymptotic Notations
- Substitution Method
- Recurrence Tree → recursive fn
- Master Method

- Asymptotic Notations -

* Notations that are used:

- O big-oh — upper bound
- Ω big-Omega — lower bound
- Θ theta — average

- Time Complexities -

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < n^n$

- Big-Oh -

. The function $f(n) = O(g(n))$ iff

$\exists$ + constants c and n_0.

such that $f(n) \leq c * g(n)$ $\forall$ $n \geq n_0$

eg. $f(n) = 2n+3$

$2n+3 \leq 10n$ $\quad n \geq 1$ $\quad \therefore f(n) = O(n)$

$f(n)$ $\quad$ c $g(n)$

Week 4 -

Recursion,

Linked list

1- Recursion - How to think Recursively?

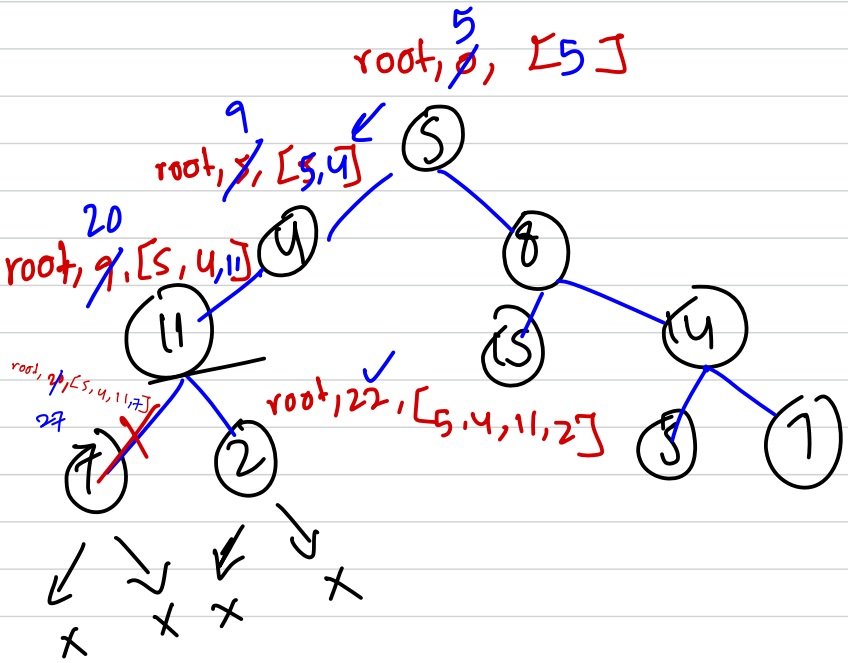
1) Binary Tree - recursion structure

↳ example - Finding path sum II :

targetSum = 22

current sum ↓ → path

⇒ root, 0, [ ]



Example - COMBINATION SUM

target = 8 [2, 3, 5]

0 1 2

- Base condition -

- current sum = target
- " " > target

[2, 3, 5]

Thinking recursively

Finding the recursive structure of the problem is the hard part.

- **Common patterns:**
  - divide in half, solve one half
  - divide in sub-problems, solve each sub-problem recursively, "merge"
  - solve one or several problems of size n-1
  - process first element, recurse on the remaining problem
- **Recursion**
  - functional: function computes and returns result
  - procedural: no return result (function returns void), i.e. the task is accomplished during the recursive calls.
- **Recursion**
  - exhaustive
  - non-exhaustive: stops early

Recursive Template

To solve a problem recursively :

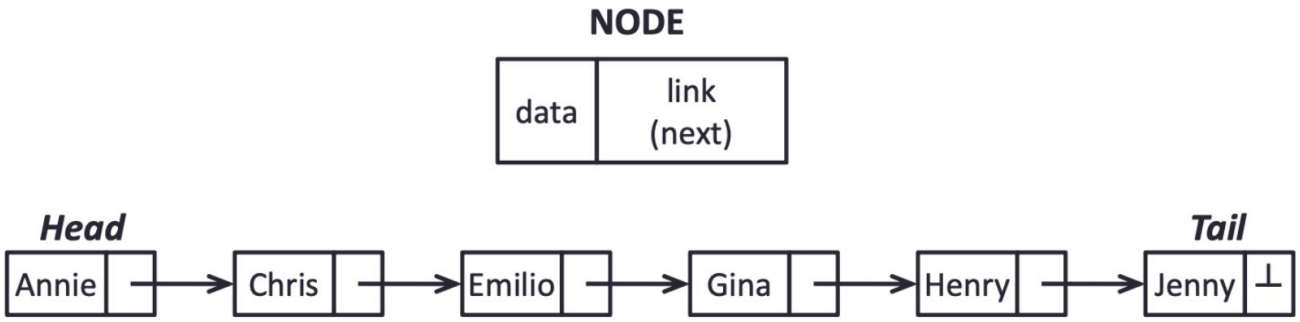
- break into smaller problems
- solve sub-problems recursively
- assemble sub-solutions.

```
recursive-algorithm (input)
{
        // base case
    if (isSmallEnough (input))
        ------

    else
        // recursive case
        break input into simpler instance, 1, 2
        solution1 = recursive-algorithm (input1)
        solution2 = -- ---- -- (input2)

        return solution1 + solution2.
}
```

Singely Linked list -

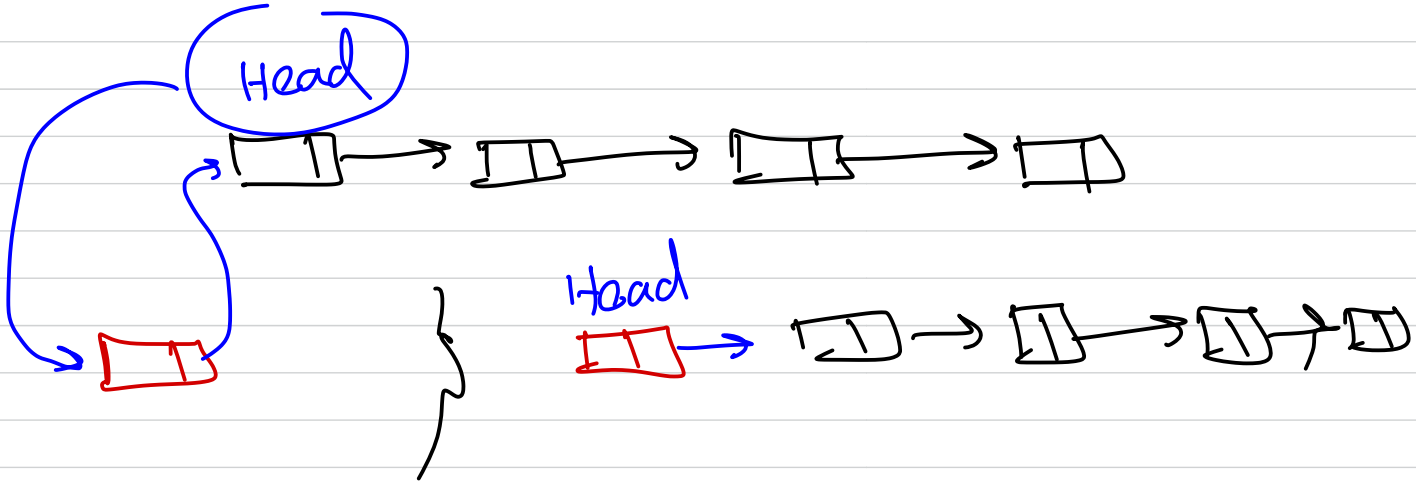
Singly Linked List: The Idea

- Store elements discretely in separate objects called nodes.
- Let each of them know the next element (via references).

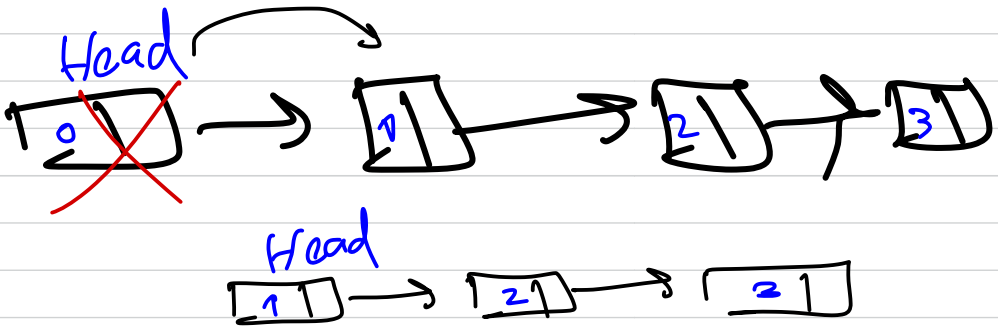


- Keeping track of our linked list requires that we know where it starts. Then we can follow the trail of links to traverse the whole list.
- So we are going to use the Head (First) node to keep track of it.
- We may also record the Tail to introduce some extra operations.

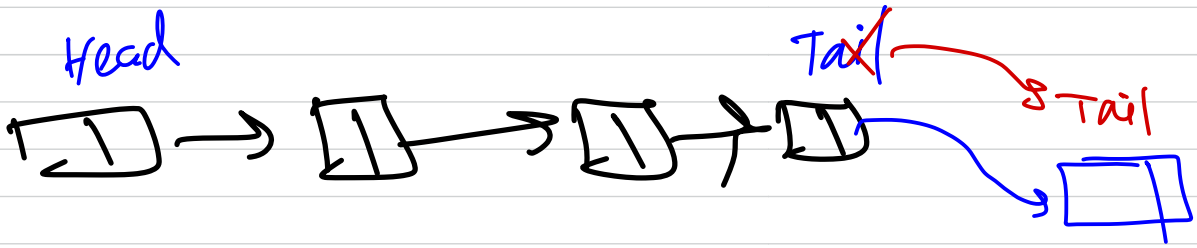
- Inserting at the Head: $\Theta(1)$



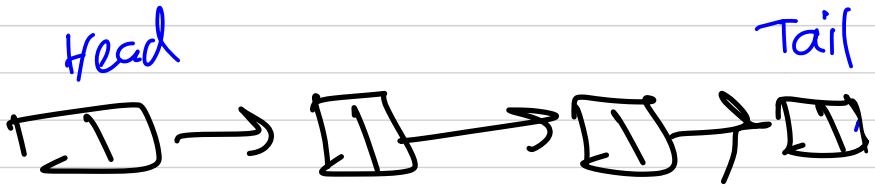
- Removing at Head: O(1)



- Inserting at the Tail: O(1)



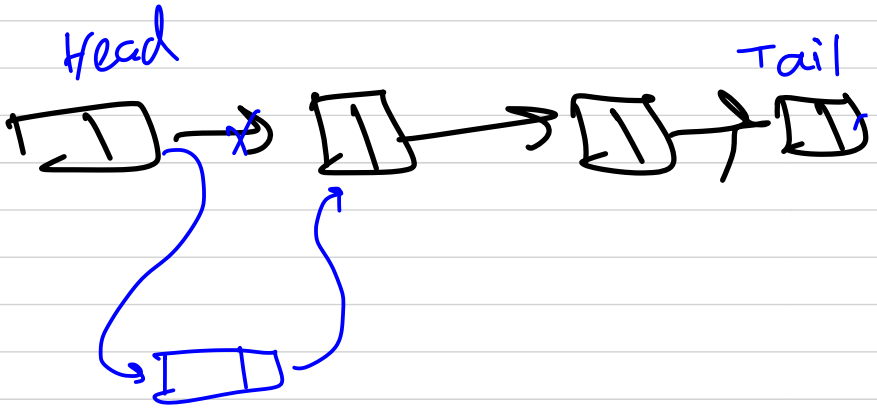
- Removing at the Tail : O(n)



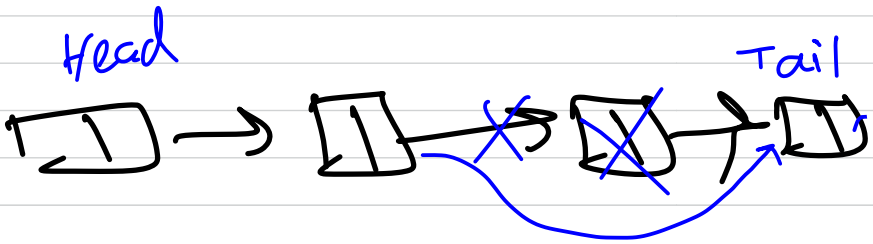
* no reference to the previous node

- loop thro list,
- and keep track of each node position
- find the node before Tail
- update the Tail to that.

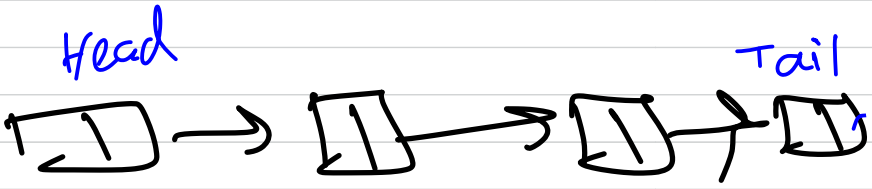
- Inserting after a node * : $\Theta(1)$



- Removing after a node * : $\Theta(1)$



- Removing a node: $\Theta(n)$



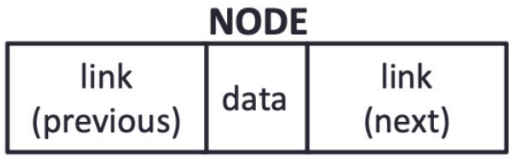
- loop thro to find the node before the node you want delete, update the ref to node the the current node points.

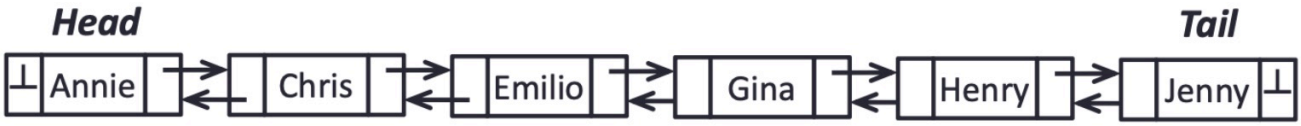
Doubly LinkedList -

to make adding & removing nodes easy

Doubly Linked List

Let each of elements know the next and the previous element.





Now we can insert or delete a node in $\theta(1)$ given only that node's memory address (reference).

Linked Lists vs. Array: Complexity Summary

Operation	Singly Linked List	Doubly Linked List	Dynamic Array
Access by index	$\theta(n)$	$\theta(n)$	$\theta(1)$
Insert	$\theta(1)^{*}$	$\theta(1)$	$\theta(n)$
Remove	$\theta(1)^{*}$	$\theta(1)$	$\theta(n)$
First	$\theta(1)$	$\theta(1)$	$\theta(n)$
Last	$\theta(n)^{**}$	$\theta(1)$	$\theta(1)$
Concatenation	$\theta(n)^{**}$	$\theta(1)$	$\theta(n)$
Count	$\theta(1)^{***}$	$\theta(1)^{***}$	$\theta(1)$

* only as Insert After and Remove After

** if the Last element is not tracked

*** only if an additional counter for the number of elements is used

Auxiliary Memory requirements:

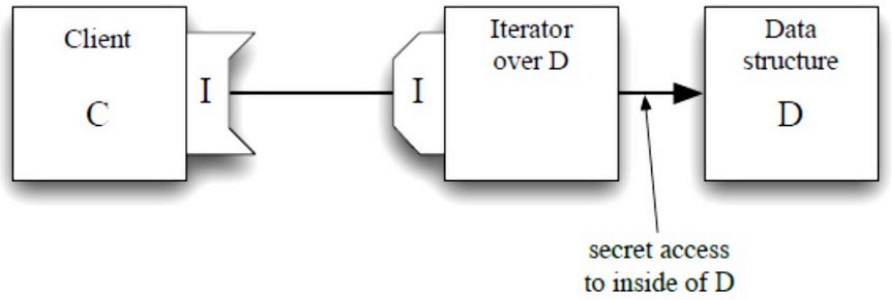
- Array stores only elements, i.e. $\theta(1)$
- Singly linked list stores the successor of each element, i.e. $\theta(n)$
- Doubly linked list stores the predecessor and successor of each element, i.e. $\theta(2n)$

Enumeration of Elements via Iterator

Enumeration interface to traverse data structures

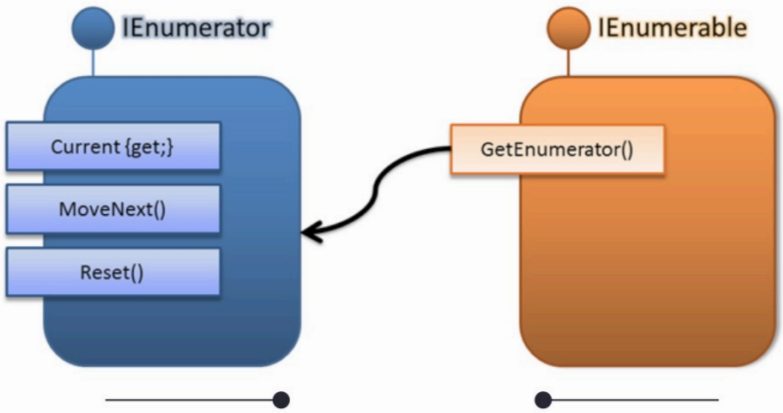
Enumeration of elements in a generic data structure is possible through implementation of so-called **Iterator** object-oriented programming design pattern.

Iterator provides a simple way for a program to access all of the components of a data structure **without knowing the representation of that data structure**.



Enumeration interface to traverse data structures

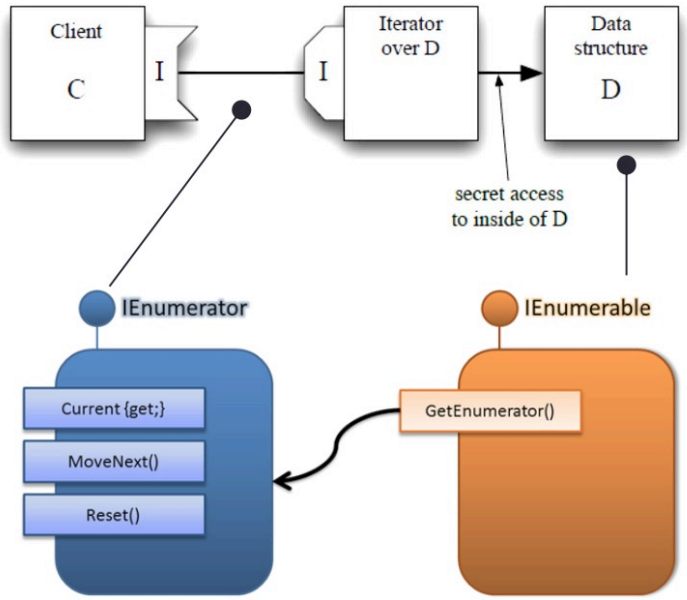
IEnumerator and IEnumerable are two interfaces to support enumeration of elements in a generic data structure in C#.



Implemented by a special Iterator class providing interface-specified access to the elements of the data structure

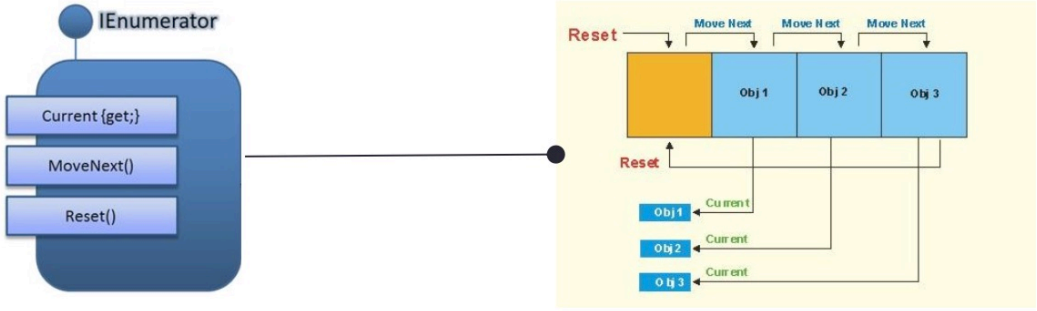
Implemented by a data structure

Enumeration interface to traverse data structures



- The data structure creates and returns an instance of Iterator via GetEnumerator() method inherited from the IEnumerable interface.
- This happens every time when the client needs to traverse the elements of the data structure.

Enumeration interface to traverse data structures



Current	Property. Gets the element in the collection at the current position of the enumerator.
bool MoveNext()	Advances the enumerator to the next element of the collection.
void Reset()	Sets the enumerator to its initial position, which is before the first element in the collection.
void Dispose()	Performs application-defined tasks associated with freeing, releasing, or resetting unmanaged resources.