

DATA STRUCTURE

Program development steps:

A set of programs to solve a problem (or) to perform a task is called software. To develop a software (or) program following steps are followed

1. understand the problem
2. Analyze the problem
3. write an algorithm
4. Draw a flow chart
5. Represent Pseudo code
6. Implement a program
7. Test the program
8. Deploy multiple modules into one entity
9. Support and maintenance

Algorithm:

A simple step by step procedure is called an algorithm. Every algorithm must satisfy the below properties: [characteristics]

- i) Input/output: algorithm must accept the input value and produce the output result
- ii) Finiteness: each step of an algorithm must terminate in finite time
- iii) Definiteness: Each step must be clearly defined
- iv) Correctness/effectiveness: an algorithm must produce correct and accurate result
- v) Generality: an algorithm must be universal which states that for any valid input algorithm must produce correct result

→ The relationship between a problem and algorithm is always 1:many. i.e a single problem can be solved in multiple ways using multiple algorithms. So to decide the efficiency of an algorithm 2 parameters are measured:

1. Time Complexity

2. Space Complexity

→ The Amount of memory space consumed by data and code together in an algorithm is called Space Complexity

→ The total time taken by an algorithm to complete its execution and produce results is called time complexities

→ The efficiency (Or) performance of an algorithm is analyzed in either of two ways.

i) posterior analysis: it is performed once the program is ready. Hence it produce accurate results. But for guarenteed better result before posterior analysis at the begining of program development anlysis is perform

ii) Priory analysis: an analysis performed at the stage of an algorithm is called priory analysis. As it is done with out program the results are estimated results & they will be approximate results

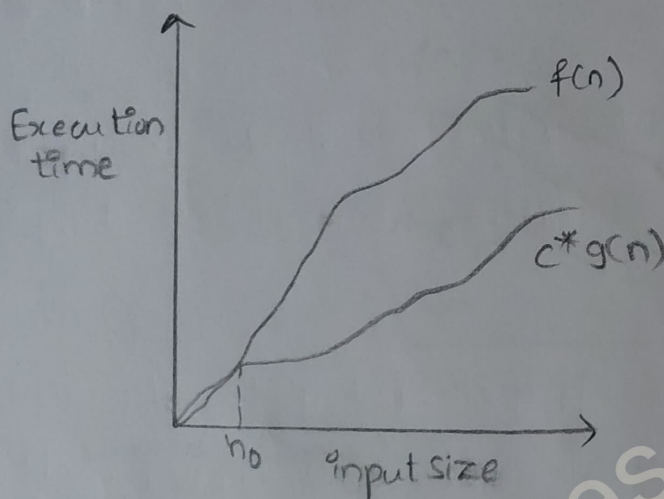
• To represent the results evaluated in priory analysis, the notations used are called ~~Asyntotic~~ Asymptotic

Asymptotic notation

To represent the time and space in mathematical form these notations are used. Time complexity and space complexity are analyzed in 3 different cases Best, Average & worse

Omega notation

It is used to represent best case result. It states that if $f(n) \geq c * g(n) \forall n \geq n_0$ then $f(n) = \Omega(g(n))$



Analyse the time complexity of below code in best case

```
void f() // const type, 1
{
    int i, n; // const type, 1
    scanf("%d", &n); // 1
    for(i=0; i<n; i++) // it run n+1 time
        printf("%d", i); // n
}
```

True False
n times (n+1)th time

$$T(n) = 1 + 1 + 1 + n + 1 + n = 2n + 4$$

$f(n)$ for above code $= 2n+4$

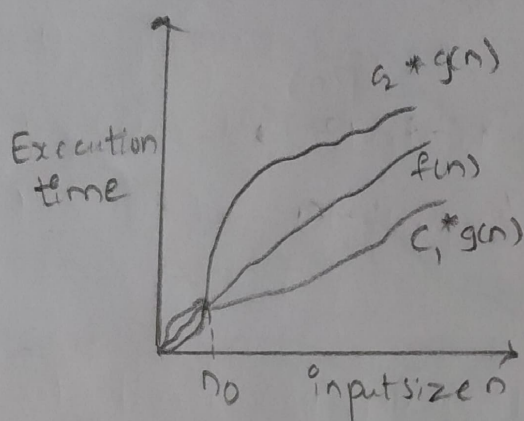
let $c = 2$, $g(n) = n$

$$2n+4 \geq 2 * n \quad \forall n \geq n_0, \text{ where } n_0 = 1$$

Hence best case time complexity of given code is $\Omega(g(n))$ i.e. $\Omega(n)$

Average case

It states that if $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$, where $c_1, c_2 > 0 \quad \forall n \geq n_0$, Then average time complexity of function f on input n is $\Theta(g(n))$



Ex:

let $f(n) = 2n+4$, then we can prove that

$$2 * n \leq 2n+4 \leq 3 * n \quad \text{where } c_1 = 2, c_2 = 3, g(n) = n$$

$$\forall n \geq 4, \text{ where } n_0 = 4$$

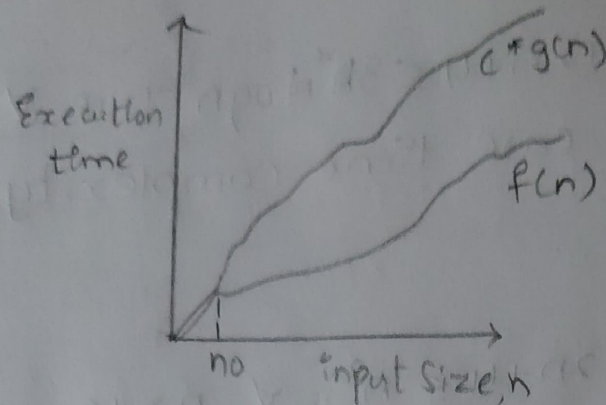
Hence average time of $f(n)$ is $\Theta(g(n))$ i.e. $\Theta(n)$

Bigoh notation

It is used to represent worst case performance.

It states that let $f(n)$ be the function to compute a task on input n .

- If $f(n) \leq c * g(n) \forall n \geq n_0$ then the worst case performance of $f(n)$ can be given as $O(g(n))$.



Ex:

Analyze the time complexity of below code:

```
void f(int n) // constant time, 1
{
    int i, j, sum = 0; // const time 1
    for (i = 0; i < n; i++) // n+1 times
    {
        for (j = 1; j <= n; j = j * 2) // n(log2n + 1) times
        {
            sum = sum + i + j; // n * log2n
        }
    }
    printf("%d", sum); // 1
}
```

$$\text{Total time} = t(n) = 1 + 1 + n + 1 + n(\log_2 n + 1) + n * \log_2 n + 1$$

$$t(n) = 2 + n + 1 + n \log_2 n + n + n \log_2 n + 1$$

$$t(n) = 2 + n + 1 + n \log_2 n + n + n \log_2 n + 1$$

$$= 2n * \log_2 n + 2n + 4 = f(n)$$

If $2n \cdot \log n + 2n + 4 \leq c \cdot g(n)$, Then the worst case time complexity will be $O(g(n))$

• When $g(n) = \log n$

$$2n \log n + 2n + 4 \leq 3n \log n \text{ True } \forall n \geq 8 \quad \text{order of / Big O}$$

hence worst case time complexity is $O(n \cdot \log n)$

• If $g(n) = n^2$

$$2n \log n + 2n + 4 \leq n^2 \text{ True } \forall n \geq 16, \text{ where } n_0 = 16$$

So it is equivalent to $O(n^2)$

• Hence $f(n) \leq c \cdot g(n)$ is True, when $g(n) = n \log n$, $g(n) = n^2$, $g(n) = n^3$ and so on...

• But it can be stated as worst case performance of $f(n)$ will be $O(n \log n)$

***Note:**

Order of time complexities $O(\sqrt{n})$

$$O(1) < O(\log \log n) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) \dots < O(2^n) < O(n!) < O(n^n)$$

Flowchart

Diagrammatic representation of an algorithm

Shape

Description

oval

Start/Stop

Parallelogram

Input/output statements

circle

connector

rhombus

Decision making statements

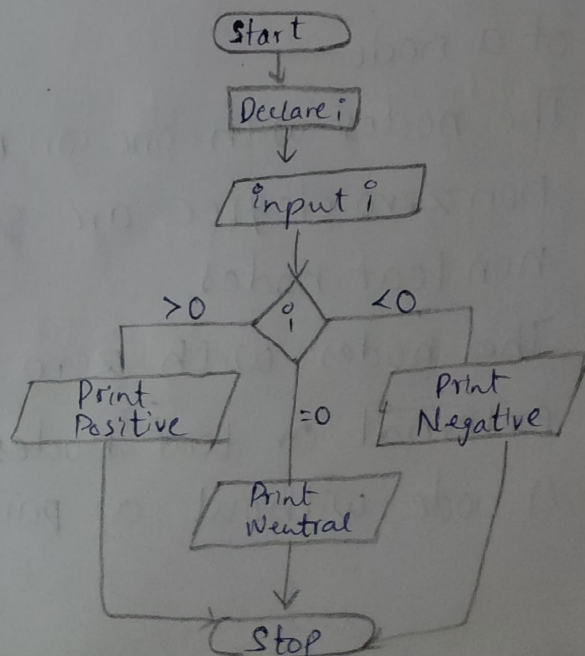
Rectangle

Processing statements

function definition

Flow of control

Ex: void main()
{
 int i;
 scanf("%d", &i);
 if (i < 0)
 printf("Negative")
 else if (i > 0)
 printf("Positive")
 else
 printf("Neutral")
}



Pseudo code:

The Syntax free version of a program is called Pseudo code

TREES

- A non linear data structure in which elements are represented in a hierarchical manner is called a tree.
- It is also known as Acyclic graph because a tree should always be open with no cycles in it.

Tree terminology

- In a tree nodes are called as vertices and links are called as edges. Hence tree $T = \{V, E\}$
- As tree is a hierarchical data structure nodes are connected in hierarchy from top to bottom. Top node is called parent node and bottom node is said to be child node of that parent
- The nodes of a tree are classified into 2 types
 - i) internal
 - ii) external
- The number of children per node is called as degree of a node
- The nodes with one (or) more children / nodes with non zero degree are said to be internal nodes (or) non leaf nodes
- The nodes with zero children / degree is called external or leaf nodes
- A node without a parent is called root node

- Nodes with common parents are said to be siblings
- In any valid path from root node to respective node, the nodes that are falling in the path are called Successors and trailing nodes are called predecessors (or) Ancestors

- level of a node and depth of a node will always be Same

- The number of edges from root node to respective node is called depth of a node.

- The maximum number of edges from leaf node to respective node is called height of the node.

- Degree of a tree = Max of the degrees

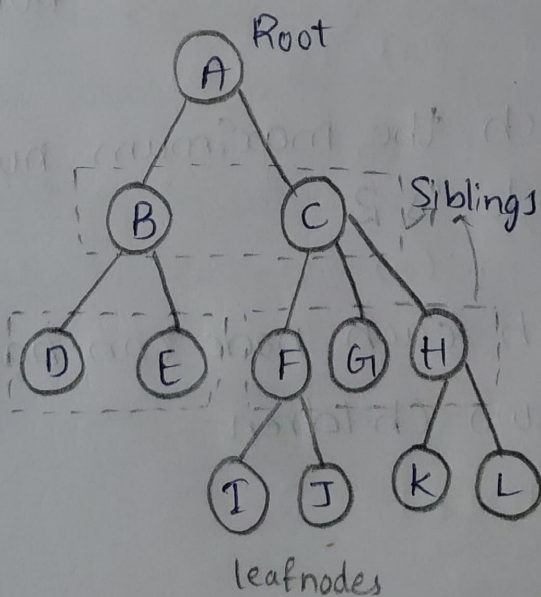
level of a tree = Max of the levels

depth of a tree = max of the depths

height of a tree = Max of heights of all nodes

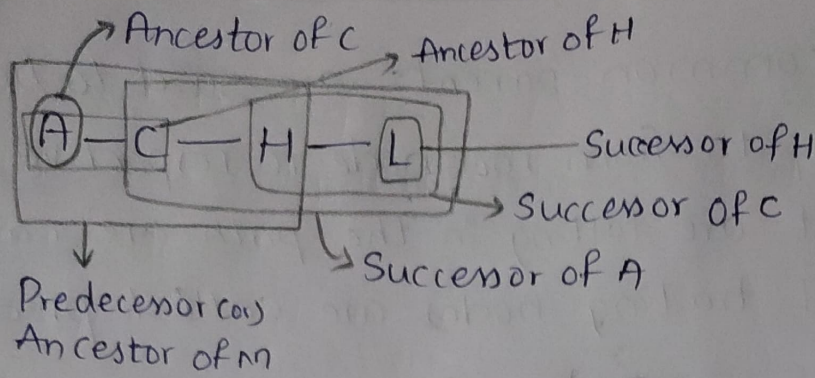
- * level of tree == Depth of a tree == height of a tree

Ex 1:

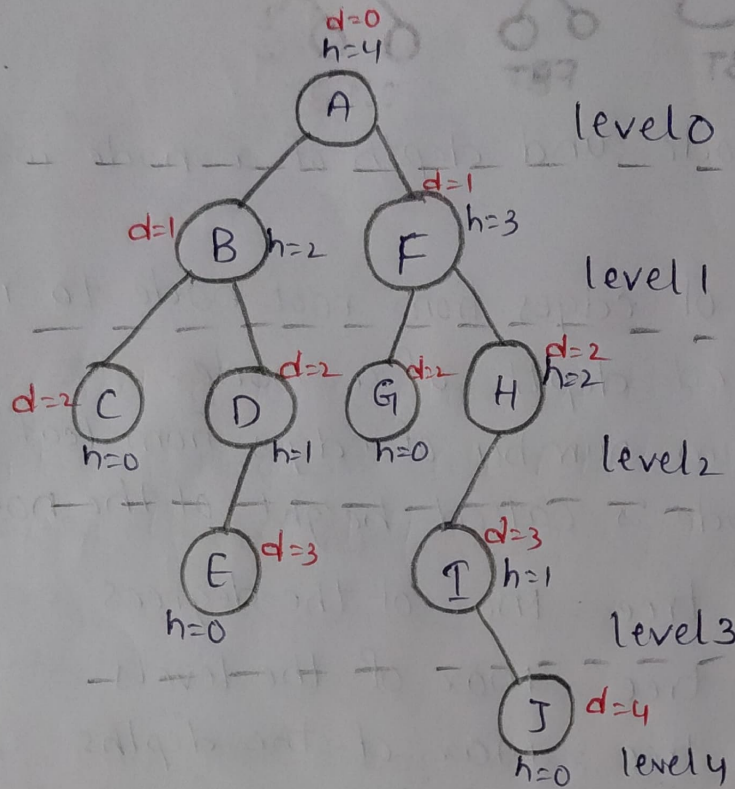


Nodes = {A, B, C, D, E, F, G, H, I, J, K, L}

Degrees = {2, 2, 3, 0, 0, 2, 0, 2, 0, 0, 0, 0}



Ex: 2



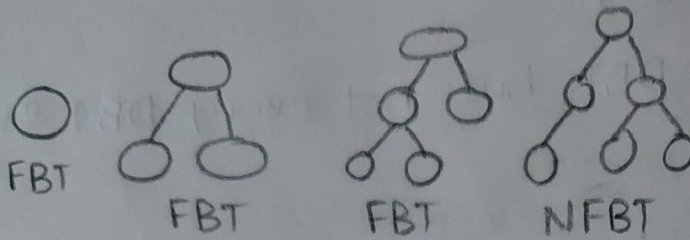
Binary trees

- A tree in which degree of each node is less than or equal to 2 (or)
A tree in which the maximum number of children per node is 2 (or)
A tree in which each node can have either zero (or) one (or) two children

Types of binary trees

i) Full binary tree: A binary tree in which each node can have either 0 (or) 2 children only

Ex:

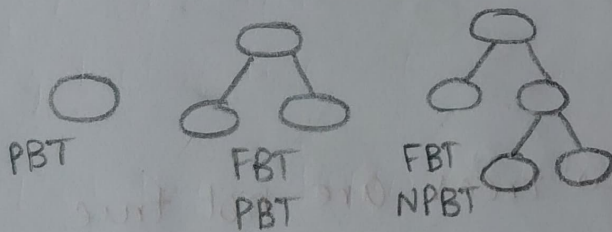


- A FBT is also known as proper binary tree (or) Strict binary tree

ii) Perfect binary tree: A full binary tree with all leaf nodes at the same level is called perfect binary tree

- First check the tree is FBT & after check all leaf nodes are in same level (or) not. If tree is not FBT then no need to check PBT

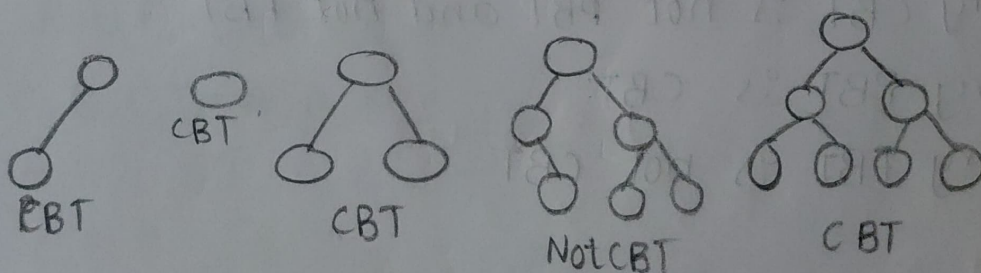
Ex:



iii) Complete binary tree: A binary tree in which nodes are filled from top to bottom and left to right (or)

A binary tree in which all levels are completely filled may be except the last level

Ex:

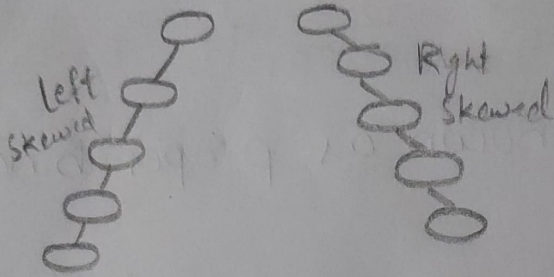


v) Skewed binary tree: A binary tree in which each node has only left child or only right child except leaf node

- Degree must be 1.

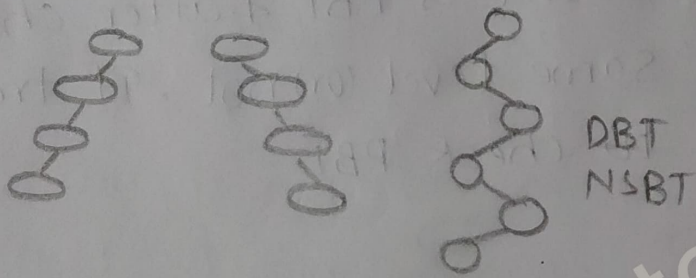
* Every SBT is a DBT but not every DBT is SBT

Ex:



v) Degenerated binary tree: A binary tree in which except leaf node all other nodes must have one children either on left or on right

Ex:



Q. Which of the below trees are not true

1. Every perfect binary tree is Complete
2. Every CBT need not to be FBT
3. Every FBT is perfect binary tree
4. Every SBT is DBT

- Every CBT is not PBT and not FBT
- Every PBT is CBT
- Every FBT is not CBT

✓) Skewed binary tree: A binary tree

Tree Traversal

The order in which nodes of a binary tree are visited or accessed is said to be traversal

There are two ways to traversal

1. **Breadth-First**: In this nodes are visited from root level to leaf level, left to right. Hence it is also known as level-order traversal

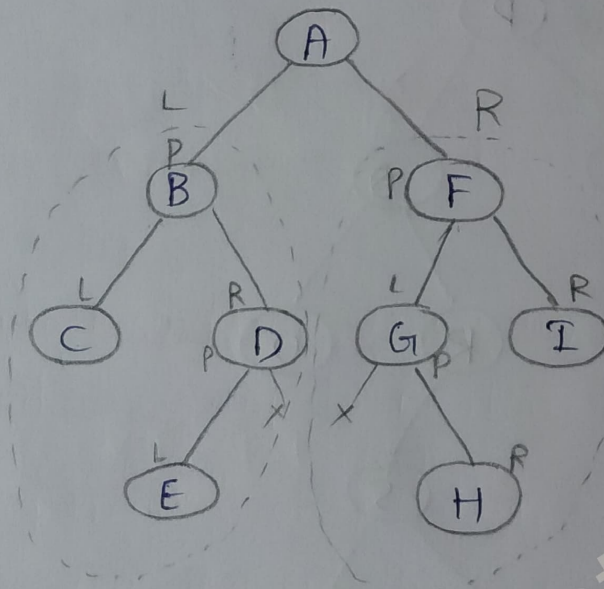
2. **Depth-First**: It is subclassified into 3 types

a) **Inorder traversal**: left subtree, parent node, right subtree (LPR)

(PLR) b) **Pre order traversal**: Parent node, left subtree, right subtree

(LRP) c) **Post order traversal**: left subtree, right subtree, parent node

Ex: 1



BFT: A, B, F, C, D, G, I, E, H.

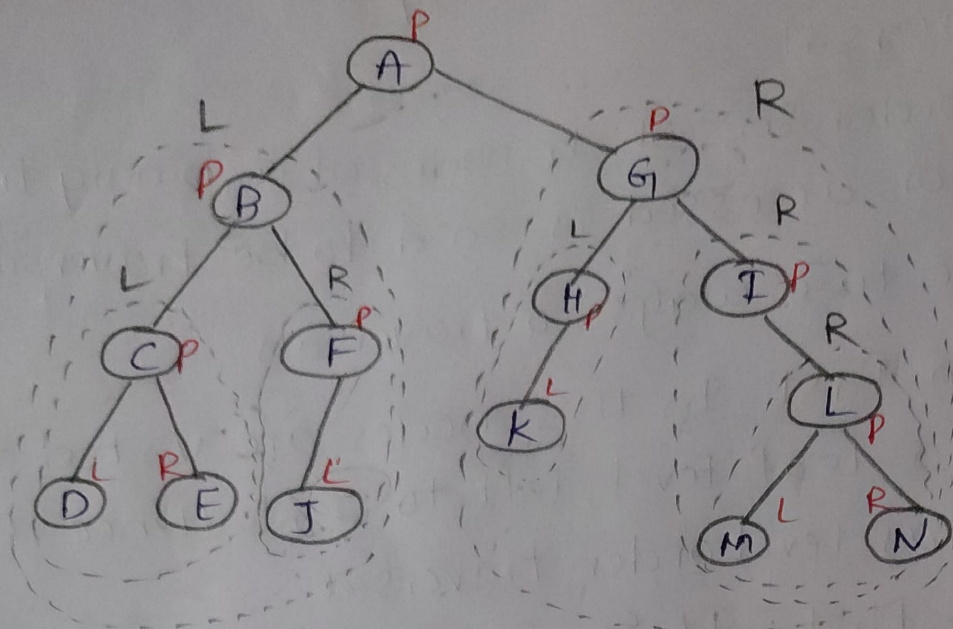
LPR In: C, B, E, D, A, G, H, F, I

PLR Pre: A, B, C, D, E, F, G, H, I

LRP Post: C, E, D, B, G, H, I, F, A

Notes by Parth

Ex: 2

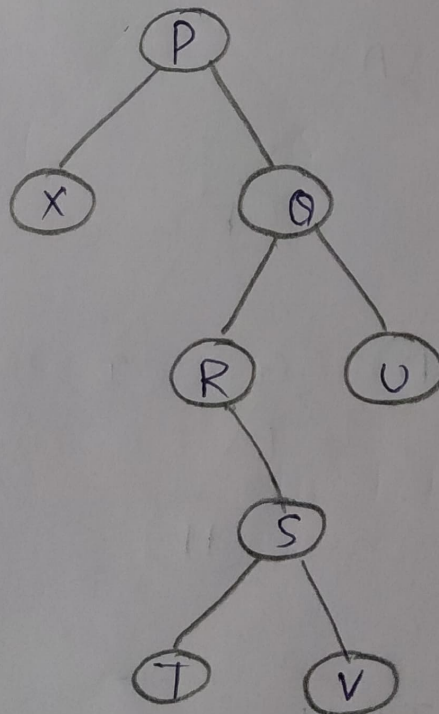


LPR In: D, C, E, B, J, F, A, K, H, G, I, M, L, N

PLR Pre: A, B, C, D, E, F, J, G, H, K, I, L, M, N

LRP Post: D, E, C, J, F, B, K, H, M, N, L, I, G, A

Ex: 3

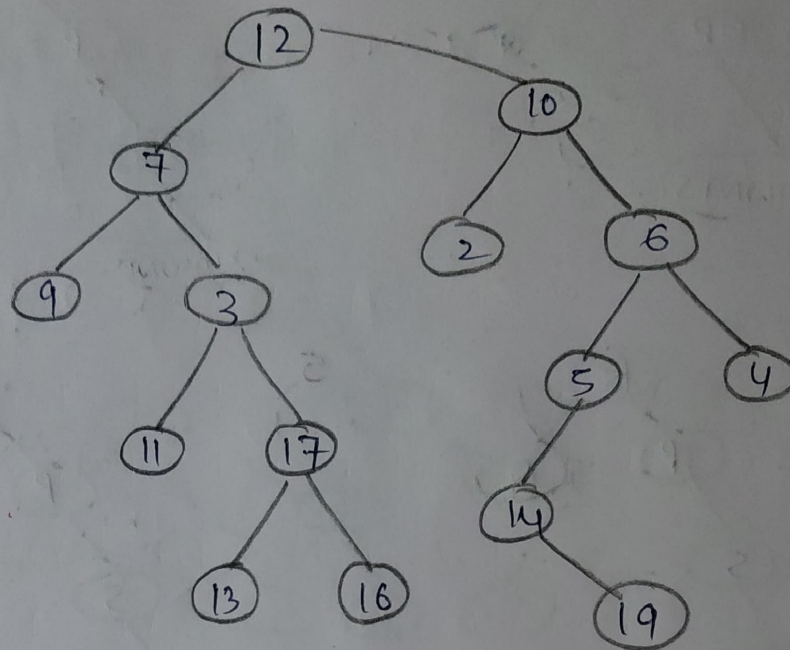


LPR In: X, P, R, T, S, V, Q, U

PLR Pre: P, X, Q, R, S, T, V, U

LRP Post: X, T, V, S, R, U, Q, P

Ex: 4



LPR In: 9, 7, 11, 3, 13, 17, 16, 12, 2, 10, 14, 19, 5, 6, 4

PLR Pre: 12, 7, 9, 3, 11, 17, 13, 16, 10, 2, 6, 5, 14, 19, 4

LRP Post: 9, 11, 13, 16, 17, 3, 7, 12, 2, 19, 14, 5, 4, 6, 10, 12

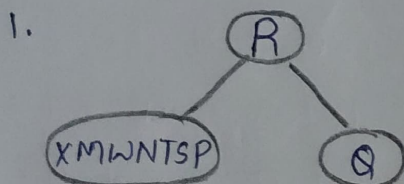
Ecet-2026

Consider a binary tree with Inorder traversal: XMWN
TS PRQ, Post order traversal as: MNWXTSPQR the
Pre order traversal is -

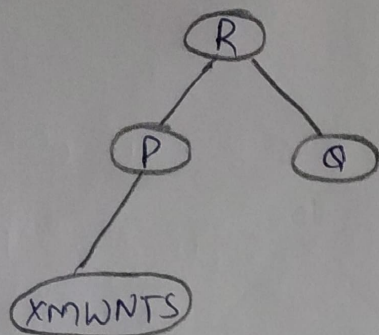
1. RPSTXMWNQ
2. RPS TXWMNQ
3. RPTS XWMNQ
4. RPTS XMWNQ

In: XMWN TS PRQ

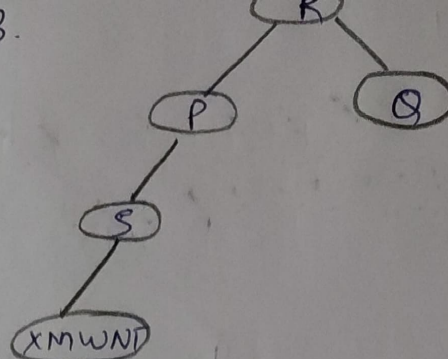
Post: MNWXTSPQR → parent/root node



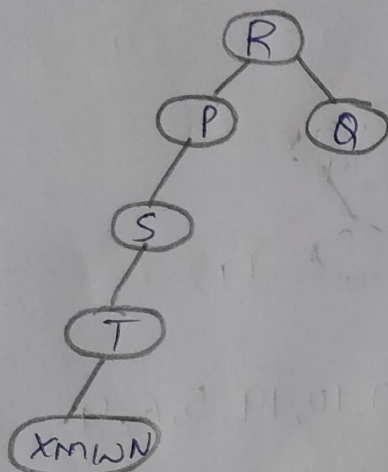
2.



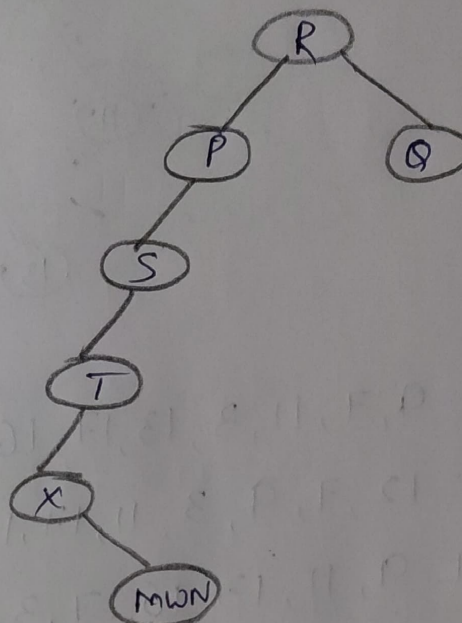
3



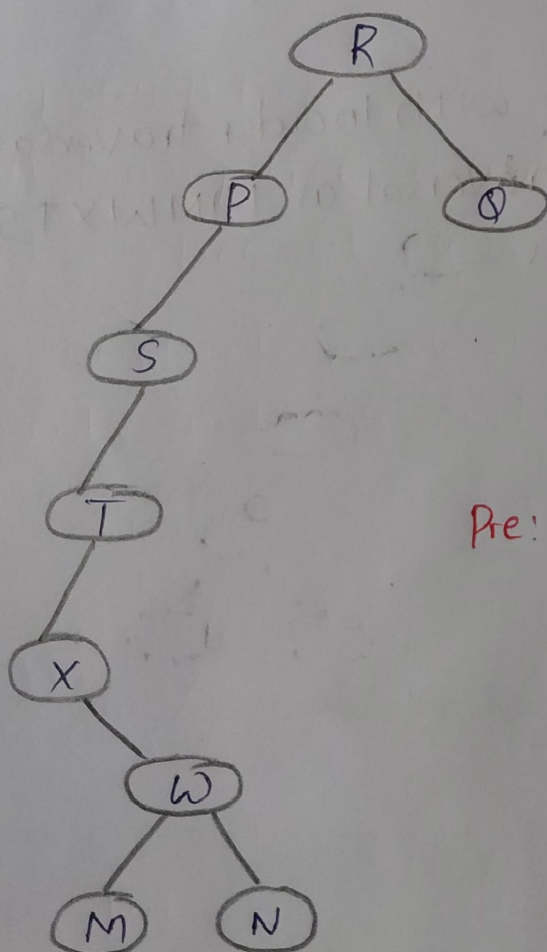
4.



5.



6.



Pre: RPSTXWMNQ-②

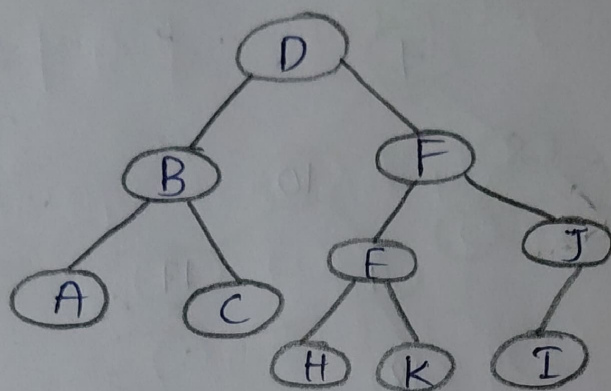
Notes by Parth

Ex: 3

In: A B C D H E K F I J

Pre: ~~D~~ B A C F E H K J I

root
node



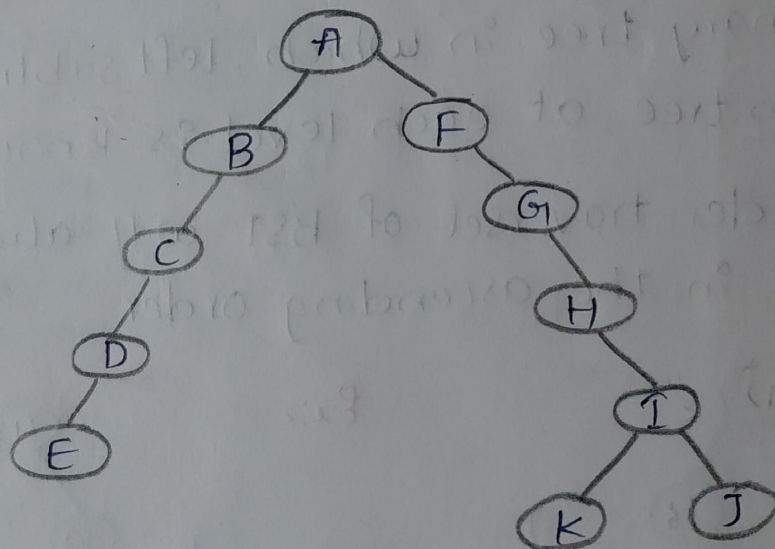
Post: A C B H K E I J F D

Ex: 4

In: E D C B A F G H K I J

Pre: ~~A~~ B C D E F G H I J K

root

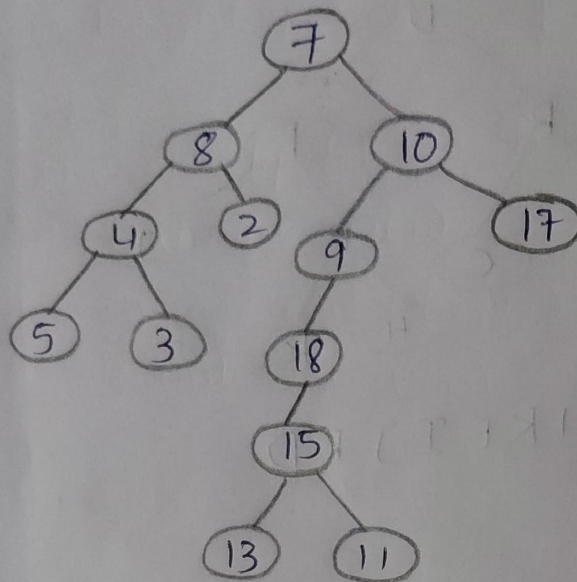


Post: E D C B K J I H G F A

Ex: 5

In: 5, 4, 3, 8, 2, 7, 13, 15, 11, 18, 9, 10, 17

Post: 5, 3, 4, 2, 8, 13, 11, 15, 18, 9, 17, 10, 7 → Parent node



Pre: 7, 8, 4, 5, 3, 2, 10, 9, 18, 15, 13, 11, 17

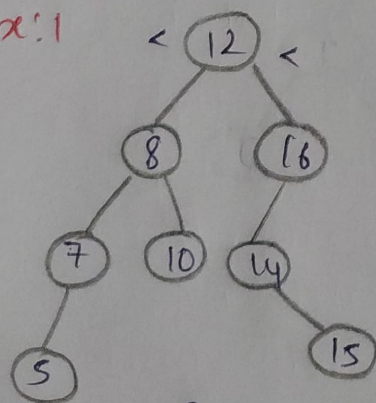
Binary search tree

A binary tree in which, left subtree < parent < right sub tree, at each level is known as BST

Note:

- The inorder traversal of BST will always produce elements in the ascending order

Ex: 1

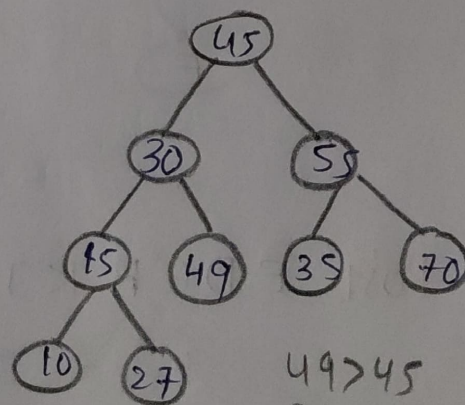


BST

In: 5, 7, 8, 10, 12, 14, 15, 16

Ascending order

Ex: 2

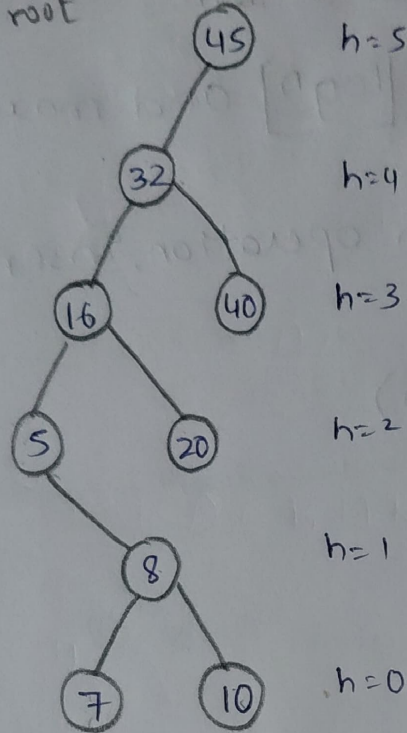


Not BST

BST - Construction / Insertion

Insertion order:

(45) 32, 16, 5, 8, 7, 40, 20, 10

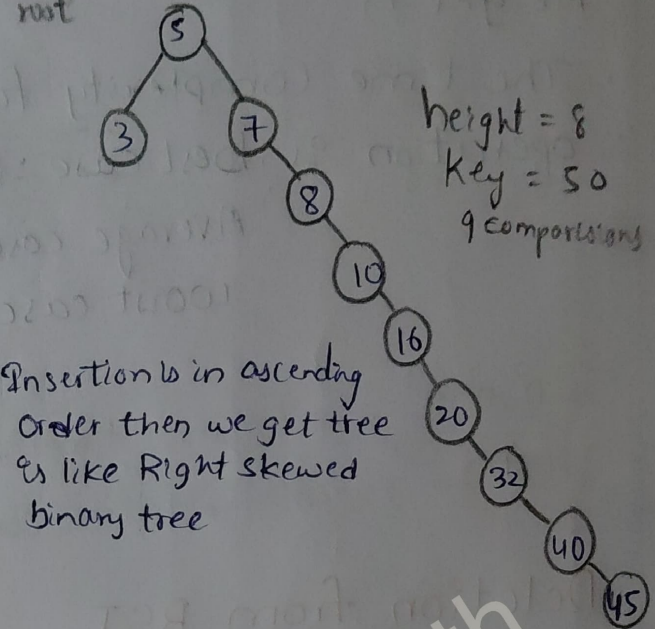


height = 6
Key = 12

In given order first one is root and insertion can be done by comparing each successive element with each root.

key = 12 & it is not in tree then it makes 5 comparisons and says 12 is not in tree
comparisons $\hat{=}$ height

(5) 7, 8, 10, 16, 20, 32, 40, 45, 3



height = 8
Key = 50
9 comparisons

Insertion is in ascending order then we get tree is like Right skewed binary tree

Notes by Parth

The height of BST depends on the order of insertion of elements. The time complexity for insertion operation, search operation and deletion operation is directly proportional to height of BST.

The minimum height possible is $\lfloor \log_2 n \rfloor$ and maximum height n

The time complexity for search operation, insertion operation is

best case: $O(1)$	→ Constant time complexity
Average case: $O(\log_2 n)$	
worst case: $O(n)$	→ linear time complexity

- n^2, n^3, n^4 ... polynomial time complexity
- 2^n → exponential time complexity

Deletion from BST

The node to be deleted from BST may be either leaf node (or) node with one child (or) node with two children

Case 1: Delete a node with zero child

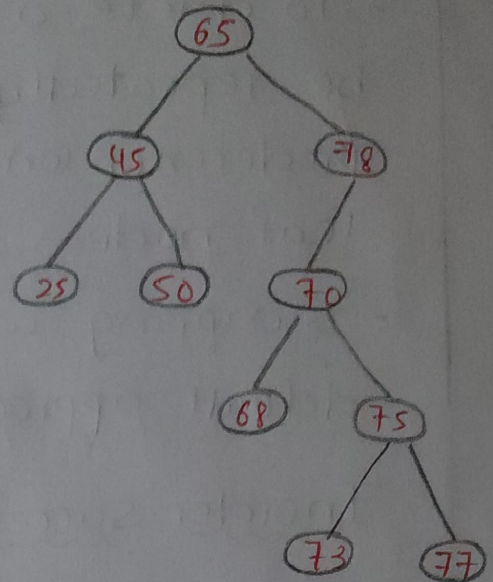
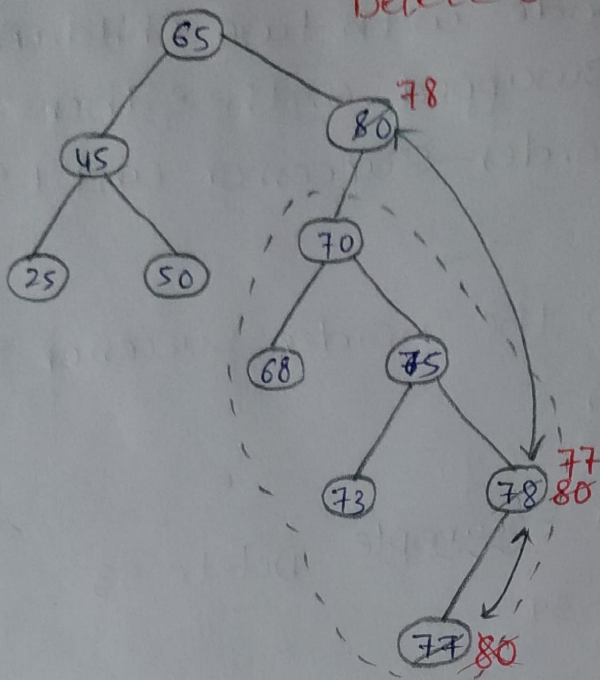
- It is a direct deletion and free up the memory space allotted to it

Case 2: Delete a node with one child

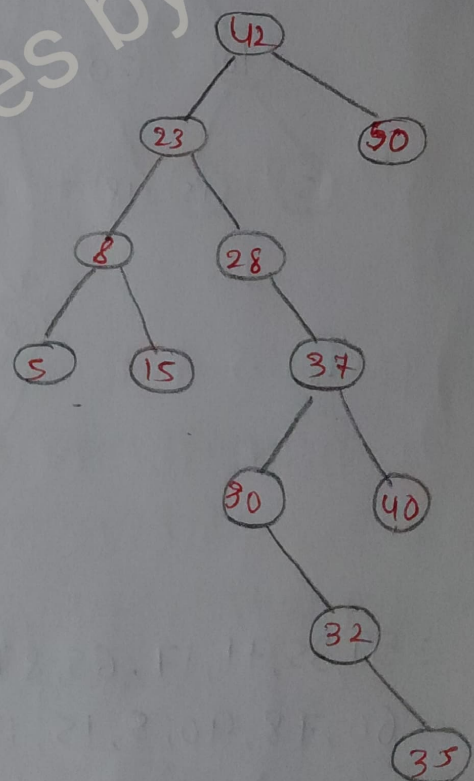
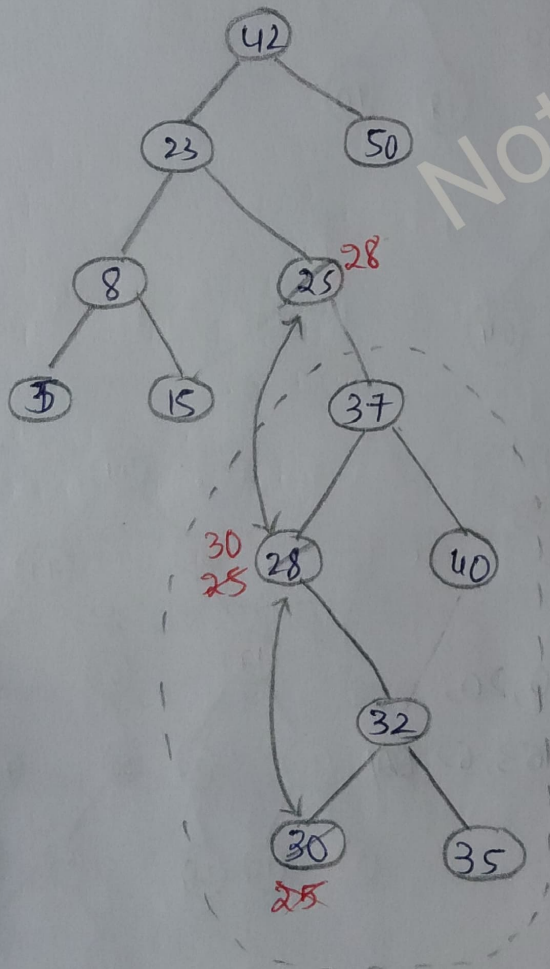
- | | |
|---|--|
| <ul style="list-style-type: none">- If left child
Swap the node with Inorder predecessor (biggest of left sub tree), until the node becomes leaf- Once node becomes leaf then it can directly deleted and free up the memory space | <ul style="list-style-type: none">- If right child swap the node with inorder successor (^{smallest} least of right sub tree) until it becomes leaf- Once the node becomes leaf then it can be directly deleted and free up the memory space |
|---|--|

Ex: Delete Node with left child

Delete 80



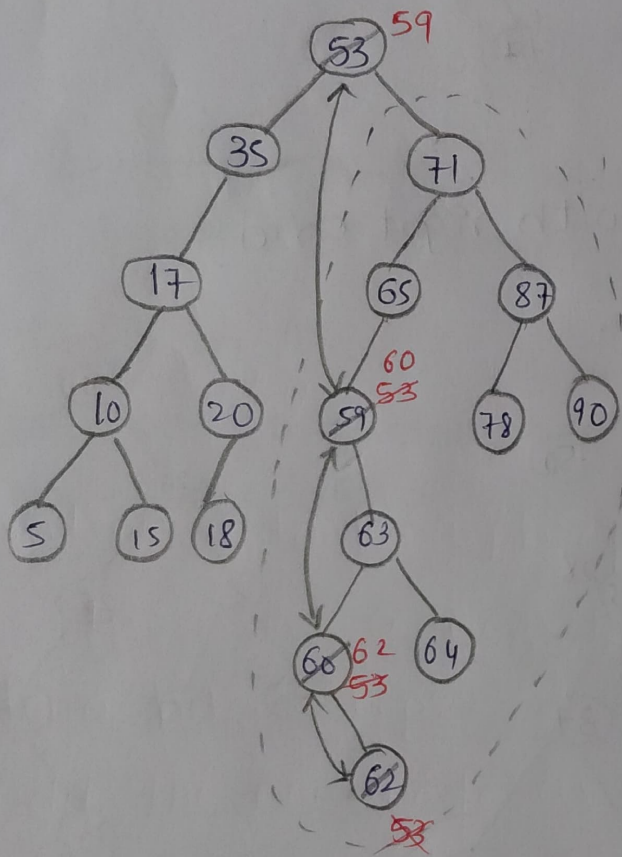
Ex: Delete node with right child



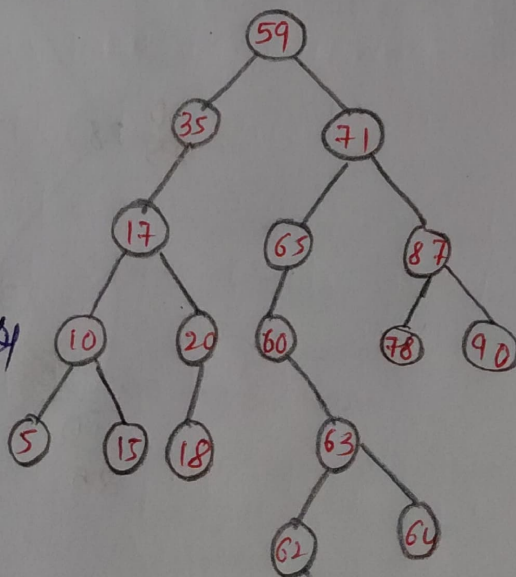
Case 3: Delete a node that has two children

- To delete a node with two children it may be repeatedly swapped with either inorder predecessor (or) inorder Successor until it becomes leaf node
- Swapping with the inorder Successor is the default approach

Inorder Successor Example: Delete 53

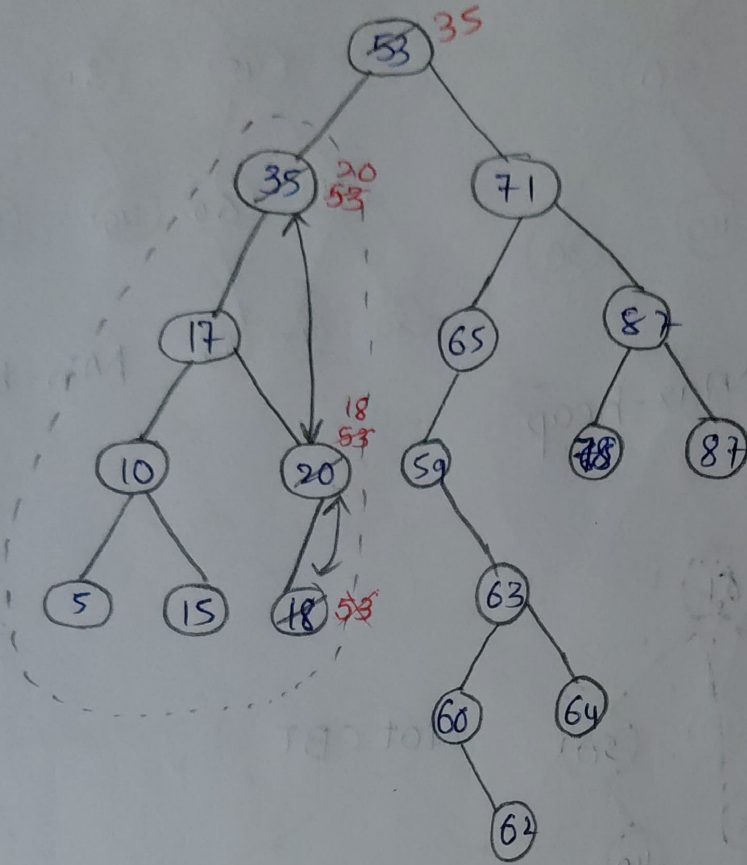


∴ 59, 35, 71, 17, 65, 87, 10, 20,
60, 78, 90, 5, 15, 18, 63, 62, ~~61~~



In order predecessor Example:

Delete 53



Binary heap

A binary tree is said to be binary heap if and only if it satisfy the below conditions:

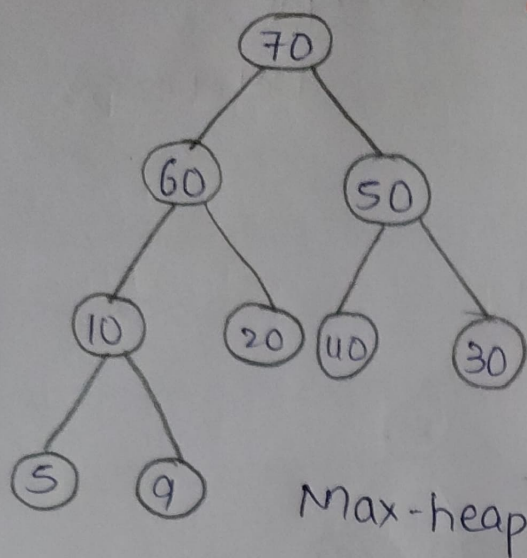
1. It must be a complete binary tree
2. ^{Every} Parent value must be either greater than all the its children (or) less than all its children

There are two types of heap max heap & min heap

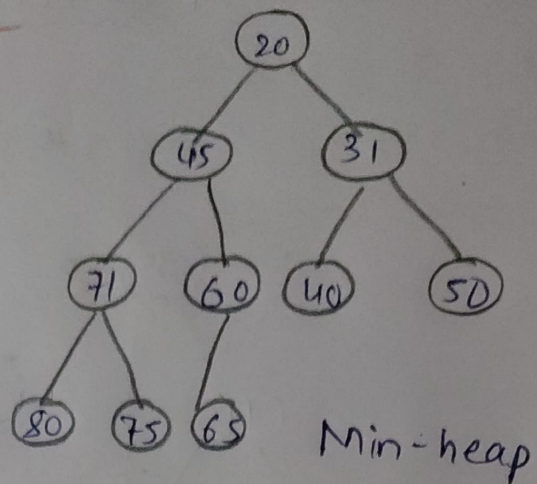
max heap = CBT + Parent $>$ all children

min heap = CBT + Parent $\leq$ all children

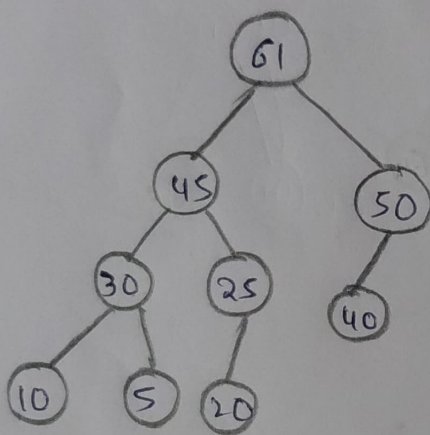
Ex. 1



Ex. 2



Ex. 3



Insertion into Binary heap

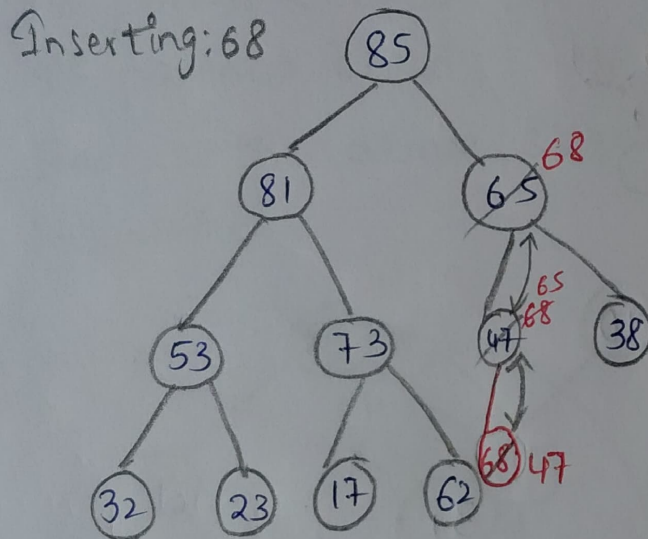
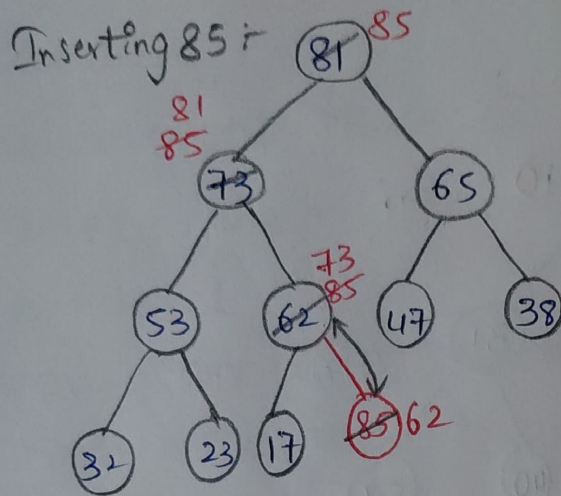
To insert a new node into binary heap perform below steps:-

1. Add the new node as last leaf node
2. compare new node with its parent value
3. For max heap: if $\text{parent} < \text{new node}$ swap
for min heap: if $\text{parent} > \text{new node}$ swap

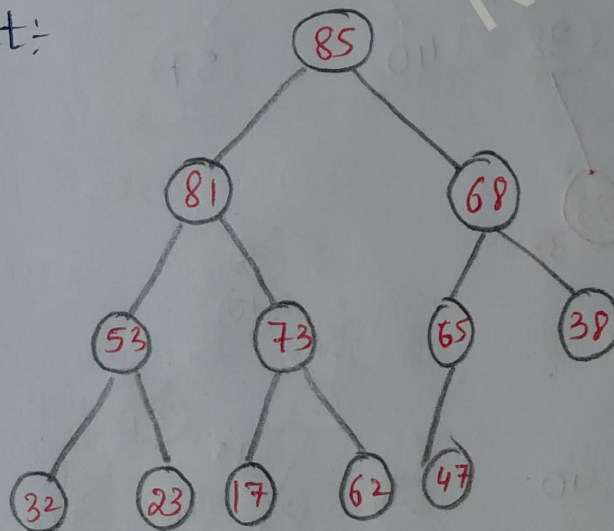
This procedure is called heapification

4. Repeat Step 2 & 3 until complete tree is heapified

Q. Consider a binary heap with elements 81, 73, 65, 56, 47, 38, 32, 23, 17. The resultant heap after inserting elements 85 and 68 is —



Result:



85, 81, 68, 53, 73, 65, 38, 32, 23, 17, 62, 47

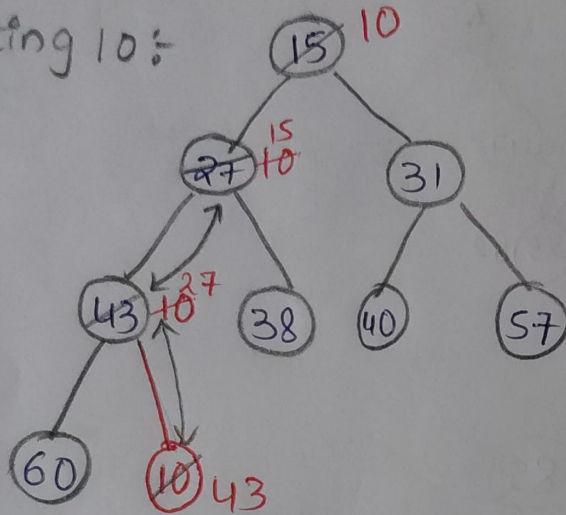
Ex: 2

15, 27, 31, 43, 38, 40, 57, 60

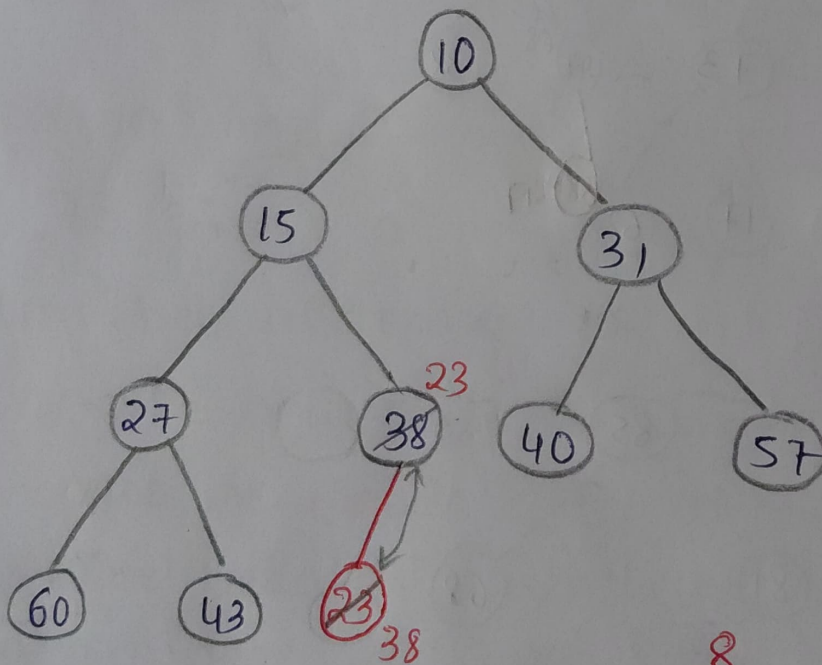
Insert 10, 23, 8

Resultant min heap:—

Inserting 10:

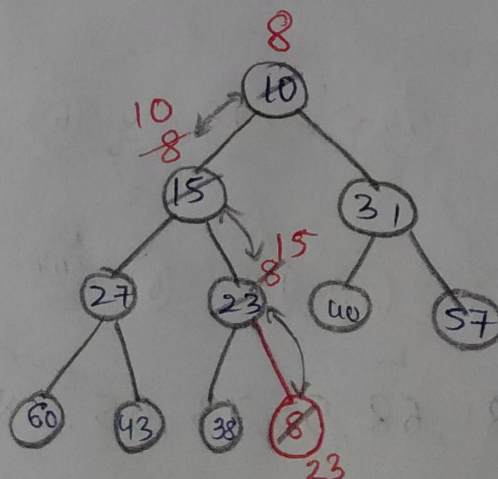


Inserting 23:



Inserting 8:

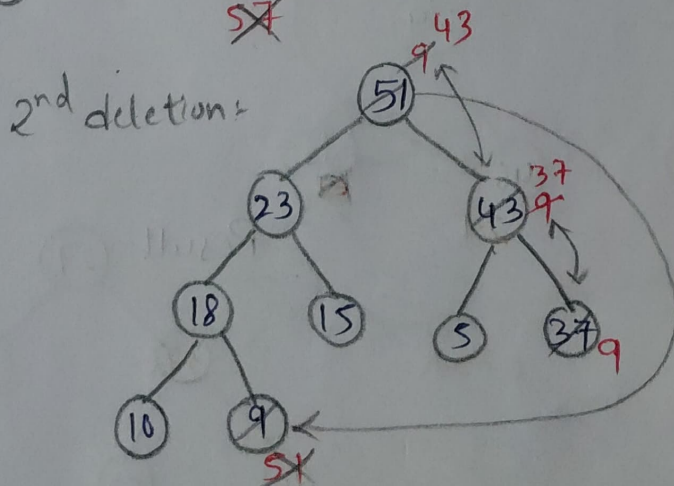
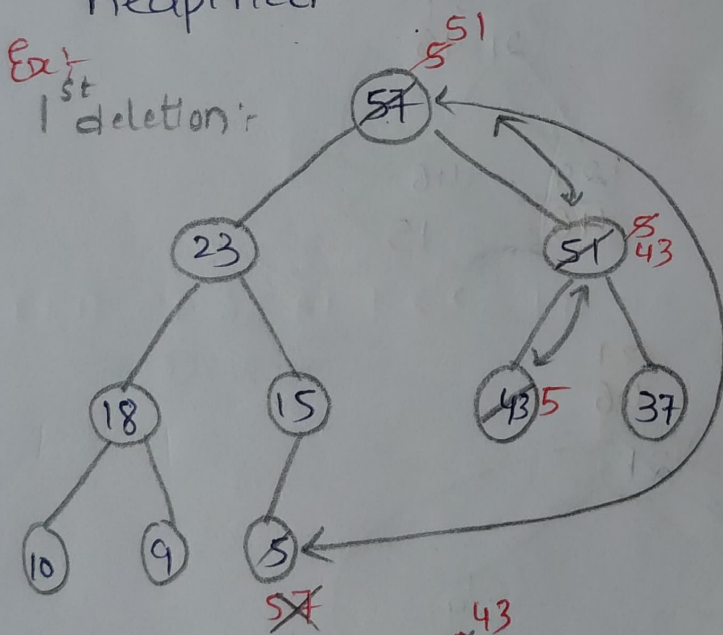
- 8, 10, 31, 27, 15, 40
57, 60, 43, 38, 23



Deletion from binary heap

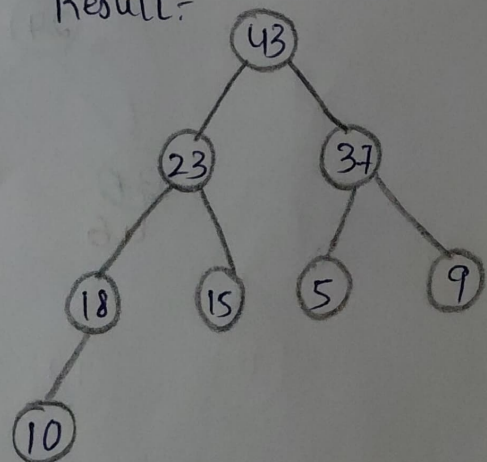
In a binary heap deletion operation can be performed only at the root node by the below procedure:

1. Swap root node with its last leaf node
2. Delete the root node that is swapped at the leaf level
3. Now compare parent value with max of its children in max heap, with min of its children for min heap.
4. If $\text{parent} < \text{child}$: swap in max heap
If $\text{parent} > \text{child}$: swap in min heap
Repeat the above steps until complete tree is heapified



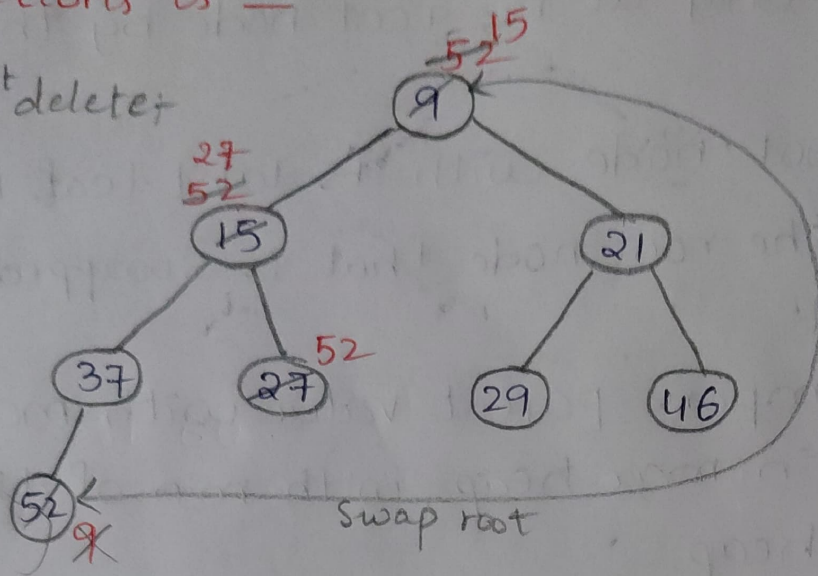
Notes by Parth

Result:-

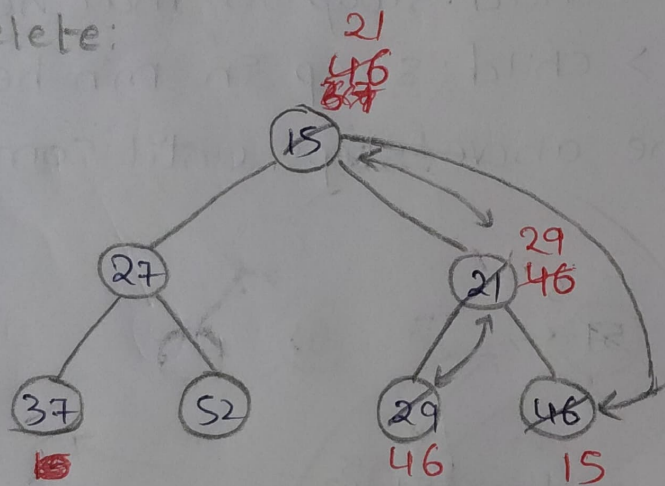


Consider a min heap with elements 9, 15, 21, 37, 27, 29, 46, 52, the resultant min heap after three deletions is —

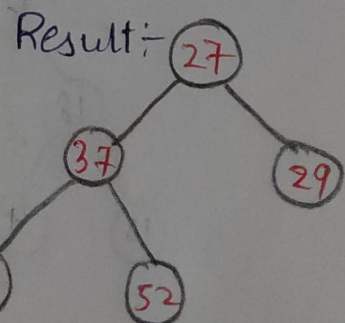
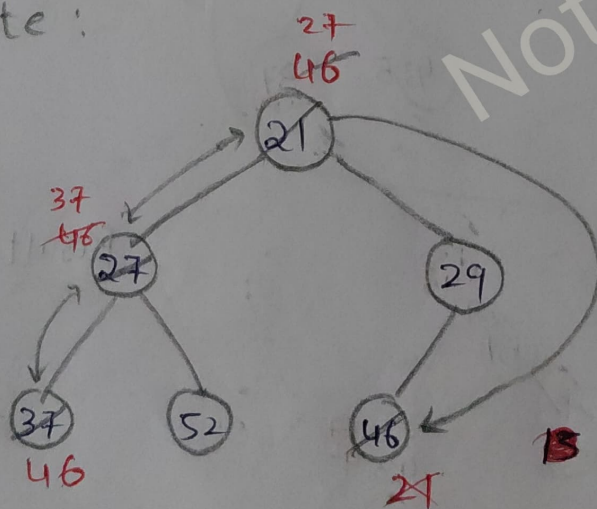
1st delete:



2nd delete:



3rd delete:

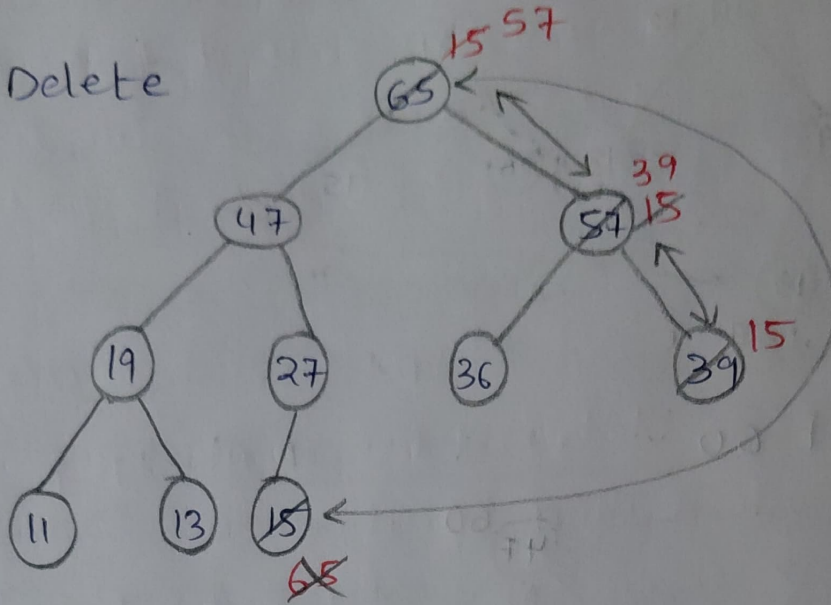


Notes by Parth

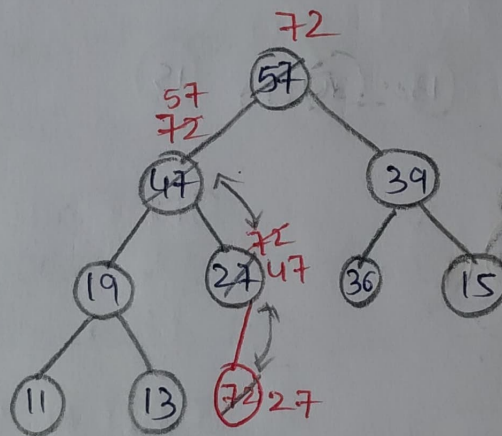
Ex: 65, 47, 57, 19, 27, 36, 39, 11, 13, 15

1. Delete 2. Insert 72 3. Delete 4. Delete 5. Insert 60

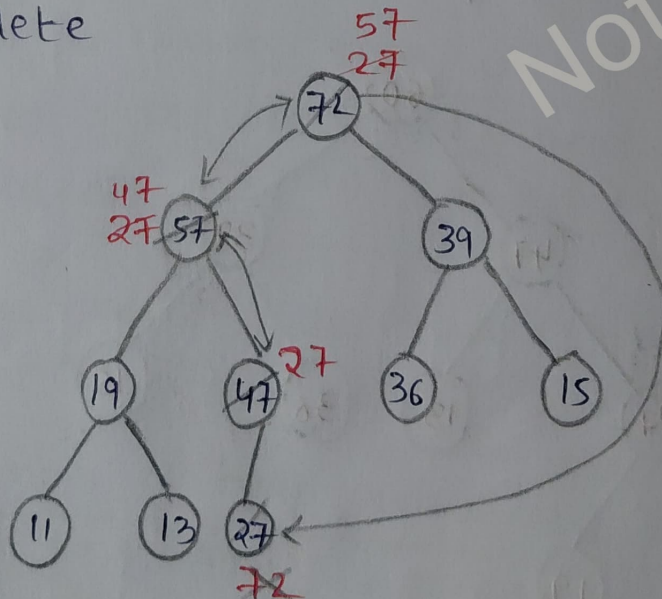
1. Delete



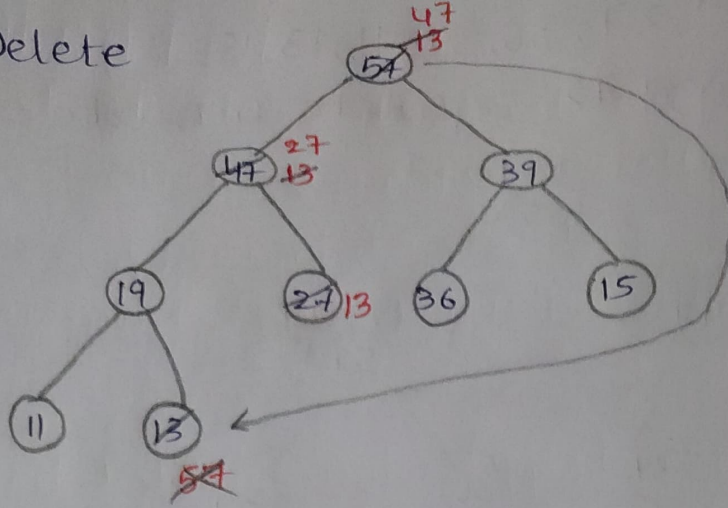
2. Insert 72



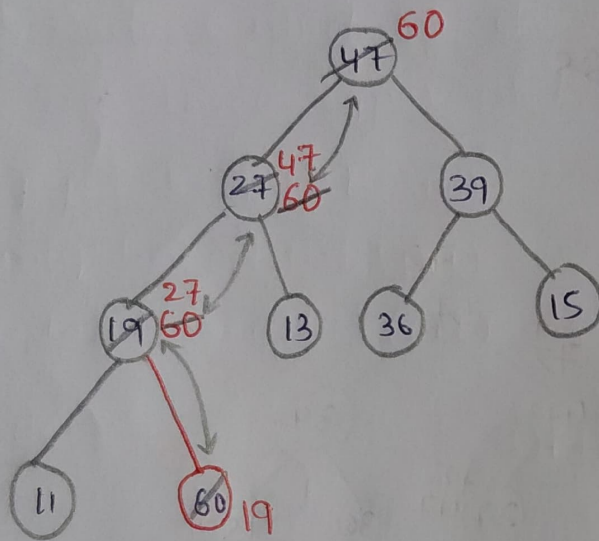
3. Delete



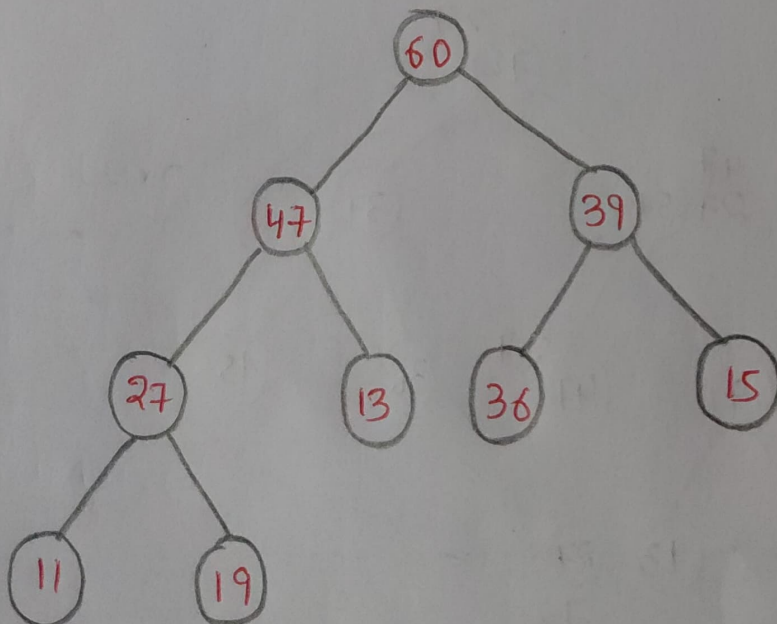
4. Delete



5. Insert 60



Result: max heap: 60, 47, 39, 27, 13, 36, 15, 11, 19



Theorems (or) Formulae on Binary trees:

1. In a binary tree $T = \{V, E\}$

$$V = E + 1$$

2. In a perfect binary tree the number of nodes at level 'L' equal to 2^L

3. A ~~full~~ binary tree ^{in any} ~~non perfect~~ binary tree with total 'N' nodes such that no. of nodes with exactly two children = no. of leaf nodes - 1

4. The total number of binary trees that can be formed using n unlabelled nodes = $\frac{2n C_n}{n+1}$

$${}^n C_r = \frac{n!}{(n-r)! r!}$$

5. The number of ~~root~~ binary trees that can be formed using n labelled nodes = $\frac{2n C_n}{n+1} * n!$

6. Given N/nodes the minimum height of the binary tree possible with N/nodes = $\lceil \log_2(N+1) \rceil - 1$

$$\text{Max height} = N - 1$$

$$N=6 \quad \lceil \log_2(7) \rceil - 1 = 3 - 1 = 2$$

7. To form a binary tree with height H the minimum no. of nodes required = $H+1$ and maximum no. of nodes required = $2^{(H+1)} - 1$

$$H=3 \quad \text{min} = 4 \quad \text{max} = 2^4 - 1 = 15$$

8. The total no. of nodes in a perfect binary tree of height (or) level (or) depth $i = 2^{(i+1)} - 1$

9. The number of heap orderings possible with 'N' nodes = $T(N) = (N-1)_{\text{left}} * T(L) * T(R)$

$$N=5 \Rightarrow T(5) = {}^4 C_3 * T(3) * T(1) = \frac{4!}{1!3!} * [{}^2 C_1 * T(1) * T(1)] * 1$$



$$= \frac{4 \times 3!}{1!3!} [2 * 1 * 1] * 1 = 8 \rightarrow \text{total No. of heap orderings}$$

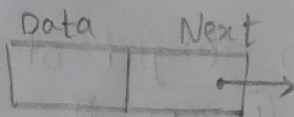
Linked List

A list of elements that are connected to one (Or) two other elements is called linked list.
• It is a linear data structure in which the memory for elements (nodes) will be allocated at random locations

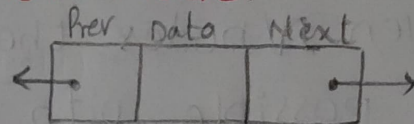
* Linked list does not support random access to elements.
• The first node of linked list is always pointed by a special node called head
• There are three types of linked list

1. Singly linked list (SLL)
2. Doubly linked list (DLL)
3. Circular linked list (CLL)

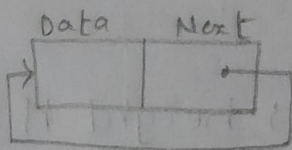
SLL Node:



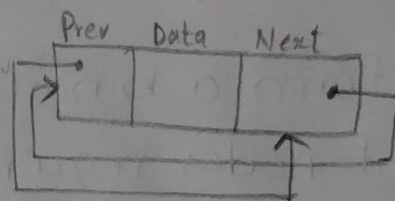
DLL Node:



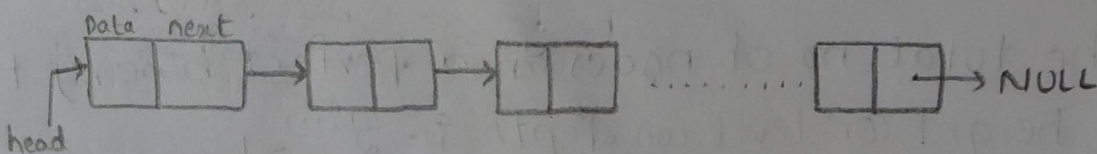
CSLL Node:



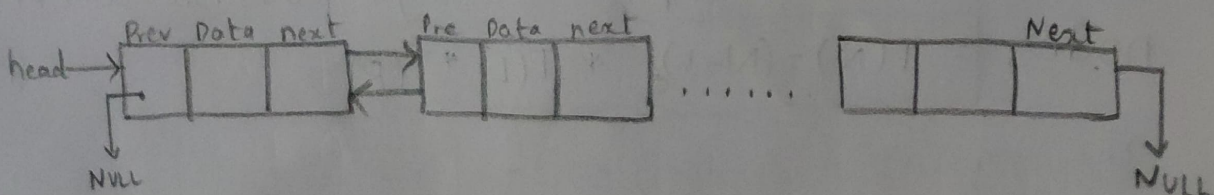
CDLL Node:



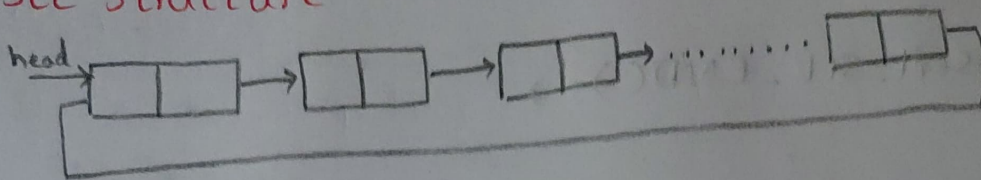
SLL structure



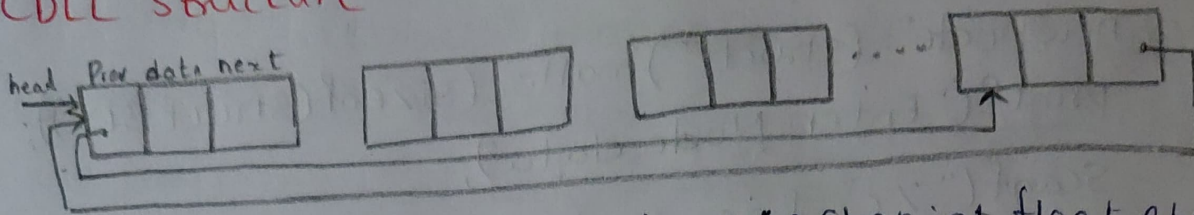
DLL structure



CSLL Structure



CDLL Structure



- Data may be of different types i.e char, int, float etc.
- ↳ Data may contain many variables

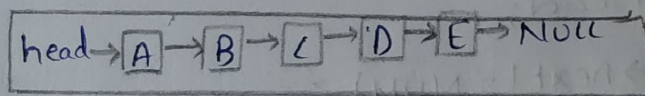
Singly linked list

Construction of SSL:

SSL may be constructed by inserting the elements in different sequences as given below

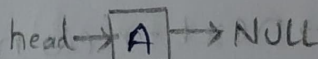
1. Insertion at the end of the list

2. Let the list to be constructed is

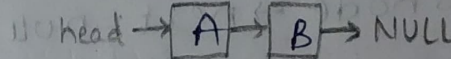


Insertion sequence: list

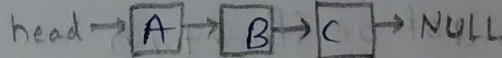
i) Insert A



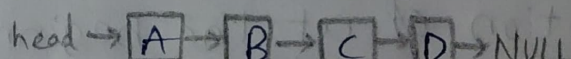
ii) Insert B



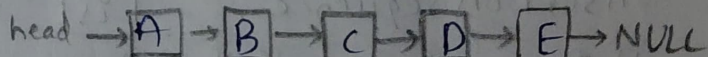
iii) Insert C



iv) Insert D



v) Insert E



SSL Node is created using self referential structure as given below

```
Struct SSL
```

```
{
```

```
// declare data members
```

```
struct SSL *next; // self referential pointer
```

```
};
```


Code:

```
Void Insert_SLL_END()
```

```
{
```

```
SLL *newnode;
```

```
newnode = (struct *) malloc(sizeof(struct SLL));
```

```
printf("Enter Node data");
```

```
scanf("%c", &newnode->data);
```

```
if(head == NULL) // empty list
```

```
{
```

```
head = newnode;
```

```
newnode->next = NULL;
```

```
}
```

```
else
```

```
{
```

```
SLL *temp = head;
```

```
while(temp->next != NULL)
```

```
temp = temp->next;
```

```
temp->next = new node;
```

```
new node->next = NULL;
```

```
}
```

```
} // end of an function
```

** Time Complexity is $O(n)$ *

2. Insertion at the beginning of the list

Insertion Sequence: List

i) Insert E

head → [E] → NULL

ii) Insert D

head → [D] → [E] → NULL

iii) Insert C

head → [C] → [D] → [E] → NULL

iv) Insert B

head → [B] → [C] → [D] → [E] → NULL

v) Insert A

head → [A] → [B] → [C] → [D] → [E] → NULL

code:

```
void Insert_SLL_begin()
```

```
{
```

```
    SLL *newnode;
```

```
    newnode = (struct *) malloc(sizeof(struct SLL));
```

```
    printf("Enter Node data");
```

```
    scanf("%c", &newnode->data);
```

```
    if (head == NULL)
```

```
    {
```

```
        head = newnode;
```

```
        newnode->next = NULL;
```

```
    }
```

```
else
```

```
{
```

```
    newnode->next = head;
```

```
    head = newnode;
```

```
}
```

```
}
```

* Time complexity is $O(1)$ *

3. Insert at any random position

i) Insert C

head → [C] → NULL

ii) Insert E

head → [C] → [E] → NULL

iii) Insert A

head → [A] → [C] → [E] → NULL

iv) Insert D

head → [A] → [C] → [D] → [E] → NULL

v) Insert B

head → [A] → [B] → [C] → [D] → [E] → NULL

Code:

```
Void Insert_SLL-Position()
```

```
{  
    SLL *newnode;  
    int pos;  
    newnode = (struct *) malloc(sizeof(struct SLL));  
    printf("Enter Node data");  
    scanf("%c", &newnode->data);  
    printf("Enter position at which insertion to be done");  
    scanf("%d", &pos);  
    if (head == NULL)  
    {  
        head = newnode;  
        newnode->next = NULL;  
    }  
    else {  
        SLL * temp = head;  
        int count = 1;  
        while (count < pos - 1)  
        {  
            temp = temp->next;  
            count++;  
        }  
        newnode->next = temp->next;  
        temp->next = newnode;  
    }  
}
```

** Time complexity is $O(n)$ *

Deletion from SLL;

The node to be deleted from SLL may be either first node (or) last node (or) middle node

Deletion of first node:

```
void Delete_SLL_first()
{
    if (head == NULL)
        printf("Empty list, cannot be deleted");
    else
    {
        SLL *temp = head;
        head = head -> next;
        free(temp);
    }
}
```

Deletion of last node:

```
void Delete_SLL_last()
{
    if (head == NULL)
        printf("Empty list, cannot be deleted");
    else
    {
        SLL *temp = head;
        while (temp -> next -> next != NULL)
            temp = temp -> next;
        temp1 = temp -> next;
        temp -> next = NULL;
        free(temp1);
    }
}
```


Deletion of middle node

```
void Delete_SLL_Middle()
```

```
{
```

```
    int pos;
```

```
    scanf("%d", &pos);
```

```
    if (head == NULL,
```

```
        printf("Empty list, cannot be deleted");
```

```
    else if (pos == 1)
```

```
        Delete_SLL_first();
```

```
    else if (pos == n)
```

```
        Delete_SLL_last();
```

```
    else if (pos > n)
```

```
        printf("Position need to be between 1 and %d", n);
```

```
    else {
```

```
        SLL *temp = head, *temp1
```

```
        int Count = 1;
```

```
        while (Count < pos - 1)
```

```
        {
```

```
            temp = temp → next;
```

```
            Count ++;
```

```
        }
```

```
        temp1 = temp → next
```

```
        temp → next = temp1 → next
```

```
        temp → next = NULL; // imp to delete the link bcz if not  
                             // it will create dangling pointer
```

```
        free(temp1);
```

```
    }
```

```
}
```

Notes by Parth

****ECEET****

Time Complexity cheat sheet of linked list

linked list	Insertion			Deletion		
	first	last	middle	first	last	middle
Singly	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Doubly	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Circular Singly	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Circular doubly	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

DLL Insertion

At beginning:

```

new → next = head;
new → next → prev = new;
head = new;
new → prev = NULL;

```

At End:

```

while (temp → next != NULL)
temp = temp → next;
temp → next = new;
new → prev = temp;
new → next = NULL;

```

At given position

```

int count = 1;
while (count < pos)
{
temp = temp → next;
count ++
}

```

```

new → next = temp → next;
new → next → prev = new;
temp → next = new;
new → prev = temp;

```


First Node:

```
DLL *temp = head;  
head = head → next;  
temp → next → prev = NULL;  
temp → next = NULL;  
free(temp);
```

Last Node:

```
while (temp → next != NULL)  
temp = temp → next;  
temp → prev → next = NULL;  
temp → prev = NULL;  
free(temp);
```

Middle Node (Node at given position)

```
scanf("%d", &pos);  
int count = 1;  
while (count < pos)  
{ temp = temp → next; count++; }
```

```
temp → next → prev = temp → prev;  
temp → prev → next = temp → next;  
temp → next = NULL;  
temp → prev = NULL;  
free(temp);
```

Note:

• In Circular linked list:

If $temp \rightarrow next = head$ // that indicates it is the last node

Notes by Parth

**Time Complexity with Tail (at last Node)

linked list	Insertion with tail			Deletion with tail		
	First	Last	Middle	First	Last	Middle
Singly	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Doubly	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Circular singly	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Circular doubly	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

Stack

It is a linear data structure in which the elements are inserted, deleted and accessed from only one end, called top of the stack (Tos).

- It is also known as last in first out (LIFO) list.
- In stack data structure insertion, deletion, access operation are termed as push, pop and peek respectively.
- Let Max size indicates total capacity of the stack. The
 if $top == maxsize - 1 \rightarrow$ Stack is full (or) overflow
 if $top == -1 \rightarrow$ stack is empty (or) underflow

Stack using arrays

```
int max-size = 5, Top = -1, Stack[max-size]
```

```
void push(int value)
```

```
{
    if (Top == MAX-SIZE - 1)
        Printf("Stack Overflow");
    else
    {
        Top++;
        Stack[Top] = value;
    }
}
```



```
int pop()
```

```
{
```

```
    if (Top == -1)
```

```
    {
```

```
        printf("stack under overflow")
```

```
    }
```

```
    else
```

```
    {
```

```
        int pop-item = stack[Top];
```

```
        Top--;
```

```
        return pop-item;
```

```
    }
```

```
}
```

```
int peek()
```

```
{
```

```
    if (Top == -1)
```

```
        printf("stack underflow");
```

```
    else
```

```
        return stack[Top];
```

```
}
```

Ex: Let S_1 and S_2 be the stacks and perform below operations & find $a = \text{peek}(S_2)$; $b = \text{Pop}(S_1)$; $a+b = ?$

1. Push($S_1, 10$)
2. Push($S_2, 20$)
3. Push($S_2, \text{Peek}(S_1)$)
4. Pop(S_1)
5. Push($S_2, \text{Pop}(S_1)$)

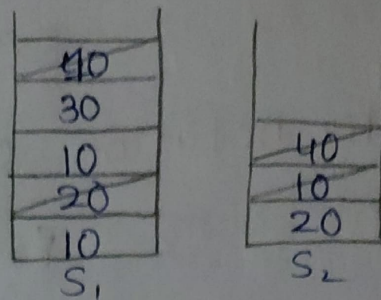
6. Push(S_1 , Peek(S_2))

7. Push(S_1 , 30)

8. Push(S_2 , 40)

9. Pop(S_2)

10. Push(S_1 , Pop(S_2))



$$a=20, b=10 \quad a+b=30$$

Applications of stack

Stack data structure is implemented in various real time applications that include:

- i) undo operation
- ii) function execution
- iii) Recursion implementation
- iv) Priority interrupts servicing
- v) Backtracking applications
- vi) Depth-first traversal
- vii) Depth-first search
- viii) Expression evaluation
- ix) Expression conversion
- x) Bracket balancing...etc

In a Arithmetic expression to limit no. of tokens and to reduce compilation time, expressions are represented in three different notations

1. **Infix notation**: Operand 1 Operator Operand 2

$$\text{Ex: } a + b$$

2. Prefix notation: operator operand 1 operand 2
(Polish) Ex: +ab

3. Postfix notation: operand 1 operand 2 operator
(Reverse polish) Ex: ab+

Infix to postfix conversion: ~~Algo~~

Algorithm

Let 'X' be given infix expression, 'Y' be resultant Postfix expression and 'S' be an empty stack

Step 1: Scan X from left to right one token at a time

Step 2: a) If token == operand, ^{append} add to Y

b) If token == '(' push on to stack

c) If token == ^{Paranthesis} operator, let it be OP_s . Now compare with top of the stack

i) If stack is empty push on to 'S'

ii) If Top == '(' Push on to stack

iii) If Top == operator, let it be OP_t

A) $OP_t \geq OP_s$ Pop OP_t , add it to 'Y', Push OP_s on to stack

B) $OP_t < OP_s$ Push OP_s onto stack

d) If token == ')' until first left '(' is encountered repeatedly pop from stack and add to 'Y'. Then ignore the brackets

Step 3: repeat step 2 until complete expression is scanned

Step 4: Once the expression is over make the stack empty by popping and adding to 'y'

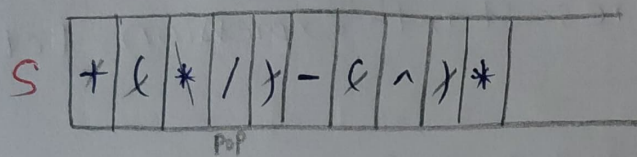
Steps: The resultant postfix expression is in 'y'

Note:

The precedence of operators is $^ > (*, /) > (+, -)$ with left to right associativity

Ex: 1

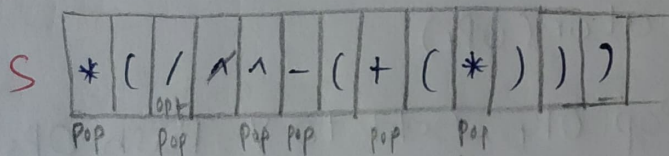
$$X = A + (B * C) / D - (E \wedge F) * G$$



$$Y: ABCD * D / + EF \wedge G * -$$

Ex: 2

$$X = a * (b / c \wedge d \wedge e - (f + (g * h)))$$

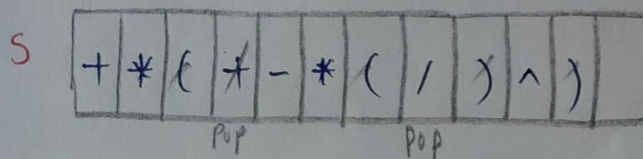


$$Y: abcd \wedge e \wedge / fgh * + - *$$

Ex: 3

The equivalent postfix expression for an Infix expression

$$X: 7 + 3 * (2 + 6 - 5 * (3 / 4) \wedge 7)$$



$$Y: 7326 + 534 / 7 \wedge * - * +$$

Ex: 4

$X: (((((a+b)*(c-d)/e)+f)^g*h))$
left to right

S:

(	(	(	(	+	)	*	(	-	)	/	)	+	)	^	*	)
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

pop pop pop pop pop

Y: $ab+cd-*e/f+g^h*$

Infix to Prefix conversion:

Algorithm

Let x be the given infix expression y is intermediate expression, z be the resultant ^{Prefix} expression, let 'S' be empty stack.

Step 1: Scan x from right to left one token at a time such that treat ')' = '(' and vice versa '(' = ')'

Step 2: Apply infix to postfix conversion procedure except

When Token = Operator (OP_s)

Top = Operator (OP_t)

if $OP_t > OP_s$ Pop OP_t , Add to y , Push OP_s

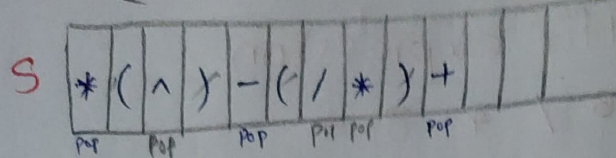
if $OP_t \leq OP_s$ Push OP_s onto stack

Step 3: the intermediate result is obtained in ' y '

Step 4: Resultant prefix expression, ' z ' equal to reverse of ' y '

Ex: 1

$$X = A + (B * C / D) - (E \wedge F) * G$$

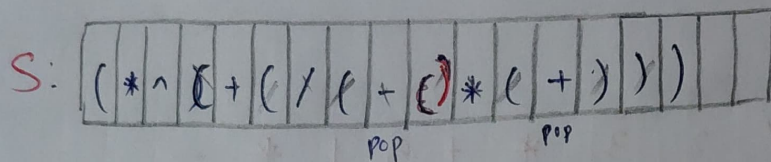


Y: G F E ^ * D C B * / A + -

Z: - + A / * B C D * ^ E F G

Ex: 2

$$X = (((c + b) * (c - d) / e) + f) \wedge g * h$$



Y: hg fedc - ba + * / + ^ *

Z: * ^ + / * + ab - cdefgh

Notes by Parth

Postfix expression evaluation

Algorithm

Let 'x' be given postfix expression and 'S' be an empty stack

Step 1: Scan left to right one token at a time

Step 2: a) If token == operand push on to stack

b) If token == operator

i) Pop from 'S', let it be 'a'

ii) Pop again from 'S', let it be 'b'

iii) Perform operation as "b operator a"

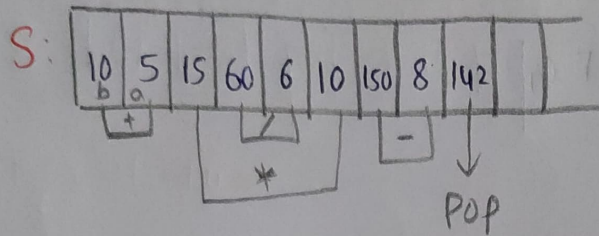
iv) Push back obtained result on stacks

Step 3: Repeat Step 2 until complete 'x' is scanned

Step 4: The ^{final} result will be top of the stack, so Pop the result

Ex: 1 Evaluate the Postfix expression

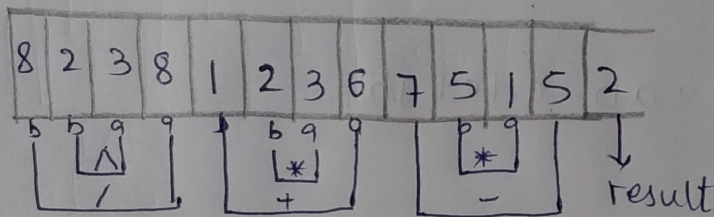
x: 10 5 + 60 6 / * 8 -



Result: 142

Ex: 2

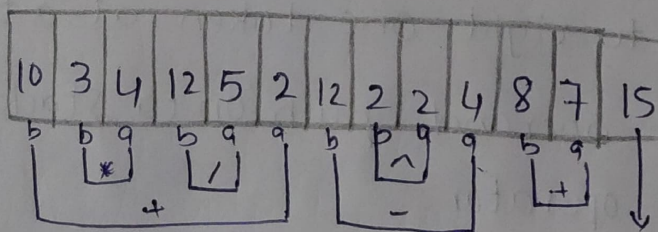
8 2 3 ^ / 2 3 * + 5 1 * -



Result: 2

Ex: 3

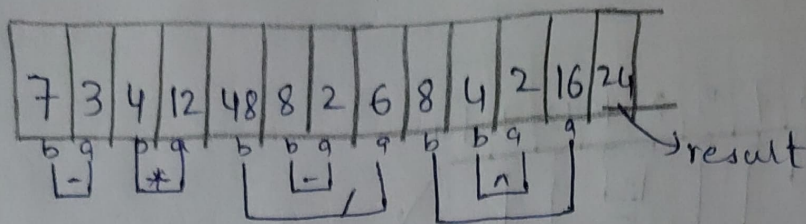
10 3 4 * 5 / + 2 2 ^ - 7 +



Result: 15

Ex: 4

7 3 - 12 * 8 2 - / 4 2 ^ +



Prefix expression evaluation

Algorithm

Let 'x' equal to prefix expression and 's' equal to an empty stack

Step 1: Scan 'x' from right to left one token at a time

Step 2: a) If token == operand push on to stack

b) If token == operator

i) pop from s, let it be 'a'

ii) pop again from s, let it be 'b'

iii) perform operation as "a operator b"

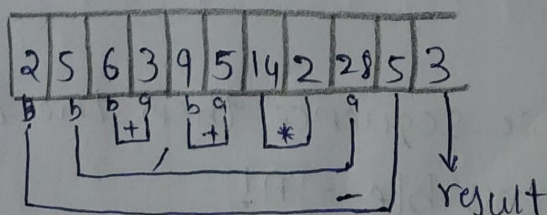
iv) Push back obtained result on stack

Step 3: Repeat step 2 until complete 'x' is scanned

Step 4: The final result will be top of the stack, so pop the result

Ex: 1

- / * 2 + 5 + 3 6 5 2

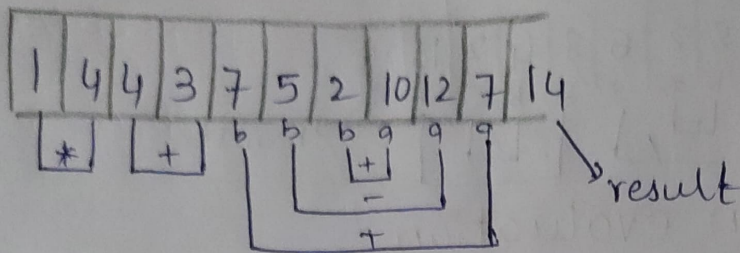


Result = 3

Notes by Parth

Ex: 2

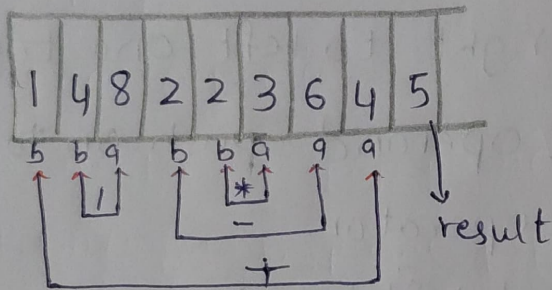
+ - + 10 2 5 + 3 * 4 1



Result = 14

Ex: 3

+ - * 3 2 / 8 4 1



Result = 5

Stack Permutations:

- The elements pushed into the stack may follow either strict (fixed) / static sequence (or) variable (dynamic) sequence
- If elements pushed in strict sequence the total number of output permutations possible equal to

$$\frac{2^n C_n}{n+1}$$

- If ^{n elements} pushed in dynamic sequence then the total output permutations possible = $n!$

Consider we need sequence $A \rightarrow B \rightarrow C$ to do this we have to insert only A after B next C but pop as we wish

Different^{olp} permutations

1. Push A, B, C, Pop, Pop, Pop results C B A
2. Push A, Pop, Push B, Pop, Push C Pop results A B C
3. Push A, Push B, Pop, Push C, Pop, Pop results B C A
4. Push A, Push B, Pop, Pop, Push C, Pop results B A C
5. Push A, Pop, Push B, Push C, Pop, Pop results A C B
6. But C, A B is invalid olp combination bcz to get first C we insert all A $\rightarrow$ B $\rightarrow$ C bcz C is the last next pop gives C next without popping B, popping A is not possible so

With formula

$$\frac{2(3)3}{4}$$

$$= \frac{6 \cdot 3}{4} = \frac{20}{4} = 5$$

Invalid.

Dynamic Sequence: no specific sequence to insert
it is variable sequence total olp combinations = $n!$

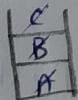
with 3 nodes - $3! = 6$ olp combinations

1. Push A, B, C

C
B
A

Pop, Pop, Pop = C, B, A

2. Push A, Pop
Push B, Pop
Push C, Pop
= A, B, C



3. Push A, Push B, Pop,
Push C, Pop, Pop
= B, C, A

4. Push A, Push B, Pop, Pop,
Push C, Pop
= B, A, C

Notes by Parth

5. Push A, Pop, push B, push C, pop, pop
= A, C, B

6. Push C pop, push A pop
push B pop

= C, A, B

→ this is not valid using static

Ex 1:-

A, B, C, D in a static sequence which one of the following sequence is invalid

1. B, C, D, A 2. C, D, B, A 3. C, B, A, D 4. C, D, A, B

Sequence A, B, C, D

Op 1 > Push A Push B pop push C pop push D pop pop valid

Op 2 > Push A push B push C pop push D pop pop pop valid

Op 3 > Push A push B push C pop pop pop push D pop valid

Op 4 > Push A push B push C pop push D pop = upto C, D it comes after in stack top is B but in sequence we need A. without popping B we can't A so invalid

Ex 2:-

P, q, r, s, t is static sequence then what is valid

Op in following

1. r, s, p, q, t ⇒ Push P, q, r, Pop, Push s, Pop next we get q but we need s so invalid

2. s, t, q, r, p ⇒ Push P, q, r, s Pop push t pop we have r in top but we have to pop q so invalid

3. q, r, p, t, s ⇒ Push P, q pop push r pop pop push s, t, pop pop - valid

4. S, q, r, t, p ⇒ Push P, q, r, s pop next in sequence we have q but in stack top is r without popping r we can't q so invalid

Stack using linked list

Push operation: Time complexity = $O(1)$

It is equivalent to Insertion at the beginning of the list

Pop operation: Time complexity = $O(1)$

It is equivalent to Deletion of first node from the list

Peak operation:

Printing the first element of the list

Ex:

Push A head $\rightarrow$ [A] $\rightarrow$ NULL

Push B head $\rightarrow$ [B] $\rightarrow$ [A] $\rightarrow$ NULL

Pop O/p: B head $\rightarrow$ [A] $\rightarrow$ NULL

Push C head $\rightarrow$ [C] $\rightarrow$ [A] $\rightarrow$ NULL

Pop O/p: C head $\rightarrow$ [A] $\rightarrow$ NULL

Pop O/p: A

Stack using queues

To implement stack using queues, two queues are needed. It can be implemented in either of two ways

i) By making Push operation costly :-

push operation - $O(n)$

Pop operation - $O(1)$

ii) By making Pop operation costly :-

Push operation - $O(1)$

Pop operation - $O(n)$

QUEUE

- It is one of the linear data structure
- There are 4 types of Queues Simple Queue, Circular Queue, double^{ended} Queue, Priority Queue
- In any form of Queue insertion operation is called as "Enqueue" & Deletion operation as "Dequeue" operation
- In simple and circular queues only insertion end is called "rear end" & deletion end as "front end"
Hence only Simple, Circular queue said as FIFO list
- In double ended queue insertion, deletion can be performed from either ends front, rear.
If both operations happen from same end then QUEUE function as LIFO list
If insertion and deletion are performed from different end then QUEUE work as FIFO
- Priority Queue is a Special form of Queue in which elements are deleted not based on the arrival order, but based on their priorities
Hence priority Queue may function as either FIFO (or) LIFO (or) Random list

Simple Queue using Arrays

```
#define MAX_SIZE 5
```

```
int f = r = -1, Q[MAX_SIZE];
```

```
void Enqueue (int value)
```

```
{
```

```
    if (r == MAX_SIZE - 1)
```

```
    {
```

```
        printf("Q is full");
```

```
    }
```

```
    else
```

```
    {
```

```
        rear++;
```

```
        Q[rear] = value;
```

```
        if (f == -1)
```

```
            f = 0;
```

```
    }
```

```
}
```

```
int dequeue ( )
```

```
{
```

```
    if (f == -1)
```

```
    {
```

```
        printf("Q is Empty");
```

```
    }
```

```
    else
```

```
    {
```

```
        int deleted element = Q[f];
```

```
        f++;
```

```
        if (f > r)
```

```
            f = -1, r = -1
```

```
        return deleted_Element;
```

```
    }
```

```
}
```

Notes by Parth

- In simple queue implementation once the queue becomes full until all elements of queue are deleted no element is allowed to insert. Hence memory gets wasted.
- To utilize the memory space effectively simple queue is modified into circular queue form

Circular queue

```
#define MAX_SIZE 5
int f = -1, Q[MAX_SIZE];
void Enqueue(int value)
{
    if (f == (r+1) % MAX_SIZE)
    {
        printf("Q is full");
    }
    else
    {
        r = (r+1) % MAX_SIZE;
        Q[r] = value;
        if (f == -1)
            f = 0;
    }
}
```

** Time complexity is $O(1)$

```
int dequeue()
{
    if (f == -1)
    {
        printf("Q is Empty");
    }
}
```

else

{

int deleted_Element = Q[f];

f = (f+1) % MAX_SIZE;

if (f == r)

f = -1, r = -1

return deleted_Element;

}

}

Ex:1 Consider two Simple queues Q_1 & Q_2 with element

$Q_1 = \{1, 2, 3, 4, 5\}$ and $Q_2 = \{5, 4, 3, 2, 1\}$ respectively,

Now perform below operations in the given

Sequence Q_1 $f_1=0$ $r_1=4$ Q_2 $f_2=0$ $r_2=4$

- Enqueue (Q_2 , dequeue(Q_1)) Q_1 $f_1=1$ $r_1=4$ Q_2 $f_2=0$ $r_2=5$
- Enqueue (Q_1 , dequeue(Q_1)) Q_1 $f_1=2$ $r_1=4$ Q_2 $f_2=0$ $r_2=5$
- Enqueue (Q_2 , dequeue(Q_2)) Q_1 $f_1=2$ $r_1=5$ Q_2 $f_2=1$ $r_2=6$

• dequeue (Q_1)

• dequeue (Q_2)

• Enqueue (Q_1 , dequeue(Q_2))

• dequeue (Q_2)

a = dequeue(Q_1)

b = dequeue(Q_2)

a+b = _____

Q_1 = _____

Q_2 = _____

f_1 = r_1 =

f_2 = r_2 =

Notes by Parth

Consider a stack 's' with elements 15, 20, 20, 12 in bottom to top order. Let queue be a simple queue with elements 30, 12, 15, 20 inserted in the same order. Now perform below operations on S & Q in the same sequence

Push(s, dequeue(Q))

Enqueue(Q, dequeue(Q))

Enqueue(Q, Pop(s))

Push(s, Dequeue(Q))

Pop(s)

Dequeue(Q)

Enqueue(Q, peek(s))

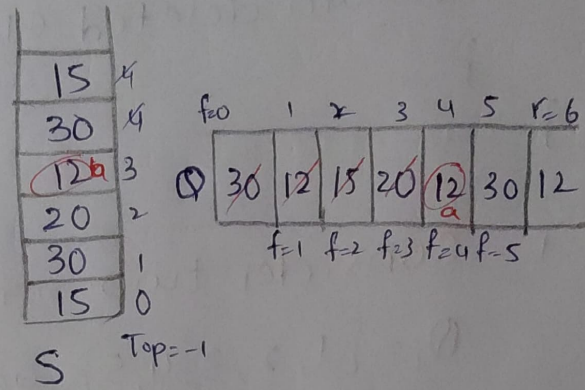
Push(s, pop(s))

a = dequeue(Q)

b = Peek(s)

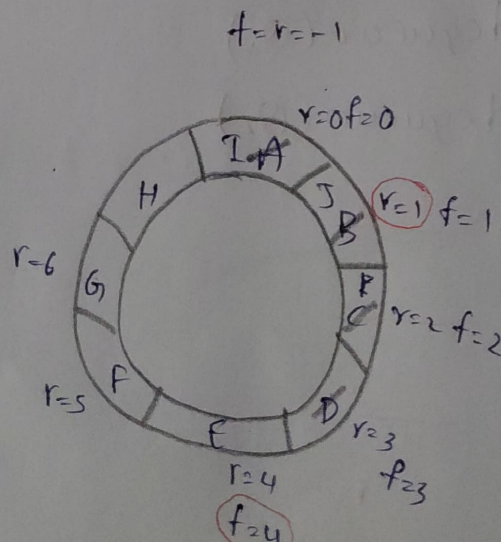
$$a + b = 12 + 12 = 24$$

The resultant of S & Q = ?



Consider a circular queue of size 8 elements, initially empty. Now 7 elements inserted, 4 elements deleted and 3 elements inserted again. The front and rear index values respectively are -

$$f = 4, r = 1$$



Double ended queue (DEQUEUE)

A queue in which Enqueue & Dequeue operations can be performed from either ends: Front, Rear is called DEQUEUE

- It can be implemented either of 2 ways

1. Input restricted DEQUEUE:

Enqueue() $\rightarrow$ rear end

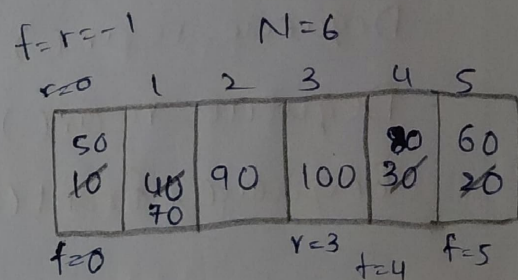
Dequeue() $\rightarrow$ front, rear

2. Output restricted DEQUEUE

Enqueue() $\rightarrow$ rear, front

Dequeue() $\rightarrow$ front

Ex:



- Enqueue_rear(10) $f=0, r=0$
- Enqueue_front(20) if $f=0: f=N-1$, $Q[f]=20, f=5$
- Enqueue_front(30) $f--$; $f=4, Q[4]=30$
- Enqueue_rear(40) $r=(r+1)\%N$ $r=1$ $Q[r]=40$;
- dequeue_front() delete 30, $f=(f+1)\%N$ $f=5$
- dequeue_front() delete 20, $f=0$
- dequeue_rear() delete 40, $r--$; $r=0$
- dequeue_rear() / front() delete 10
- Enqueue_front(50) if $f=-1, f=0, r=0$ $Q[0]=50$
- Enqueue_front(60) $f=5, Q[5]=60$
- Enqueue_rear(70) $r=1$ $Q[1]=70$
- Enqueue_front(80) $f=4$ $Q[4]=80$

- Enqueue-rear(90) $r=2$ $Q[2]=90$

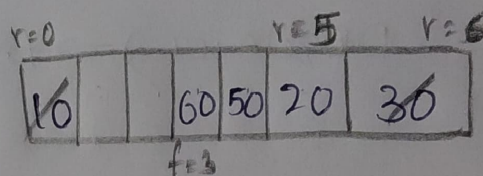
- Enqueue-rear(100) $r=3$ $Q[3]=100$, $f=4$

(or)

Enqueue-front(100) $f=3$, $r=2$

Ex: 2

Consider a DEQUEUE of size 7 with elements 10, ..., 20, 30. The current index values are rear & equal to 0, front = 5. After below operations performed the final front & Rear index values are respectively —



- Enqueue-front(50)

- Dequeue-rear()

- Dequeue-rear()

- Enqueue-front(60)

Note: ECET

* The conditions to check DEQUEUE overflow and underflow

Overflow $\rightarrow f = (r+1) \% \text{MAX-SIZE}$

underflow $\rightarrow f = -1$ (or) $f = r$

Priority Queue

It is a special form of queue in which elements are deleted based on their priorities

Eg:

Consider a priority queue with elements and priorities inserted in the order {value, priority} = {2, 3} {3, 1} {4, 2} {1, 5} {5, 4}. Let 1 = high priority

Insertion sequence of values = 2, 3, 4, 1, 5

Deletion sequence would be = 3, 4, 2, 5, 1

** Priority Queue is used in binary heap implementation and heap sort technique

Searching & Sorting

Searching

The process of finding an element in the list of element is called searching

• There are 2 popular algorithm for Searching

i) linear search

ii) Binary search

Linear

In this procedure key is compared with each element until it is found (or) all elements are compared. Hence for n elements n comparisons would be made in the worst case. So worst case time complexity = $O(n)$

Best case if the key is present at the beginning of the list

* Time complexity : $O(1)$

- Average case: If the key is present in the middle of the list. $\frac{n}{2} + 1$ comparisons will be made.

* Time complexity: $O(\frac{n}{2}) = O(n)$

```
void Linear-search (int *a, int size, int key)
{
    int i, status = 0;
    for (i = 0; i < size; i++)
    {
        if (key == *(a+i))
        {
            status = 1;
            if (status == 1)
                break;
        }
        if (status == 1)
            printf("key found at %dth Pos", i);
        else
            printf("key is not found");
    }
}
```

Binary search

To apply binary search procedure the input list must be in the sorted order.

- In this procedure,

Step 1: Compare key with middle element, if key is found search successful.

Step 2: Otherwise if key is less than middle element

repeat the procedure in the left sublist (0 to mid-1)

Step 3: Otherwise repeat the procedure in the right
sublist (mid+1 to high)

```
int binary-search(int *list, int low, int high, int key)
{
    mid = low +  $\left(\frac{\text{high} - \text{low}}{2}\right)$ ;
    if (key == *(list + mid))
        return 1;
    if (key > *(list + mid))
        binary-search(list, mid + 1, high, key);
    else
        binary-search(list, low, mid - 1, key);
    return 0;
}
```

* Time complexity

Best case: key is the middle element $O(1)$

Average case: $O(\log n)$

Worst case: $O(\log n)$

Sorting algorithm

The process of arranging the elements either in ascending or in descending order is called sorting technique. Some of the popular sorting algorithms include Selection sort, Insertion, Bubble, Merge, Quick Sort.

Sorting algorithms are classified based on 3 factors as
given below

1. Based on stability: When there are duplicate elements in the list to be sorted their relative positions ordering if preserved then it is called Stable sorting algorithm otherwise unstable

Stable - IBM Insertion, Bubble, Merge

Unstable - SQ Selection, Quick

2. Based on the adaptability: Based on the input sequence of elements if an algorithm can run faster and improve the performance it is called adaptive

Adaptive - I Insertion

Non adaptive - BMSQ

3. Based on the memory consumption: While sorting the elements at any given time if there is a possibility to consume ^{space} memory/ in the secondary memory then it is called external sorting algorithm

External/out-place - Merge sort

Internal/ⁱⁿplace - SBIQ

Sorting Algorithm	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

1 Selection sort:

In this technique select smallest element in unsorted list and place in 0^{th} index

Ex: 53 27 65 15 37 72 30

Pass 1:- In first Pick smallest element in above series i.e 15 and compare with all remaining elements ($n-1$) comparisons and place at first

15 53 27 65 37 72 30

Pass 2:- next take second smallest element i.e 27 and compare with all other elements except first one 15 and itself 27 and place in second

Place ($n-2$) comparisons 15 27 53 65 37 72 30

Pass 3:- next smallest one is 30 place it 3^{rd} place without comparing with 15, 27 and itself (30). ($n-3$) comparisons

15 27 30 ~~37~~ 53 65 37 72

Pass 4:- again repeat smallest in 37 place in 4^{th} place of array after 30 without comparing (15, 27, 30, 37) ($n-4$) comparisons

15 27 30 37 53 65 72

Pass 5:- Above after pass 4 the sequence is in sorted but not stops again compares 53 with other elements (65, 72) ($n-5$) comparisons

15 27 30 37 53 65 72

Pass 6:- Smallest one is 65 Compare with only 72 and place it correct place

15 27 30 37 53 65 72

Pass 7:- place 72 in correct place 0 comparisons

15 27 30 37 53 65 72

In this technique smallest element placed in first place no. of comparisons is $(n-1) + (n-2) + \dots$
 $= \frac{n(n-1)}{2}$

* Time complexity is $O\left(\frac{n(n-1)}{2}\right) = O\left(\frac{n^2}{2} - \frac{n}{2}\right)$
 $O\left(\frac{1}{2} \cdot n^2\right)$ takes longest power i.e. n^2
 $\rightarrow$ const coeff can be deleted
 $= O(n^2) *$

Notes by Parth

2. Bubble Sort:-

In this technique each element is compared with adjacent element if element is adjacent smaller than element then swap but for all comparisons can be done

Ex:

~~1111~~ 53 27 65 15 37 72 30

Pass 1: 27 53 65 15 37 72 30 $(n-1)$ Comparisons = 6 comp
 but in this technique after 1st pass we found that largest element at last in next pass comparisons with that element is skip
 27 53 15 65 37 72 30
 27 53 15 37 65 72 30
 27 53 15 37 65 30 72

Pass 2: 27 53 15 37 65 30 72 $(n-2)$ Comparisons = 5 comp
 27 15 53 37 65 30 72
 27 15 37 53 65 30 72
 27 15 37 53 30 65 72

Pass 3: 27 15 37 53 30 65 72 $(n-3)$ Comparisons = 4 comp
 15 27 37 53 30 65 72
 15 27 37 30 53 65 72

Pass 4: 15 27 37 30 53 65 72 3 comparisons
 15 27 30 37 53 65 72
 No swap

Pass 5: above in Pass 4 we get sorted sequence but still it compares remaining elements

15 27 30 37 53 65 72

Pass 6: 15 27 30 37 53 65 72
 15 27 30 37 53 65 72

No of comparisons = $(n-1) + (n-2) + \dots + 1 + 0 = \frac{n(n-1)}{2}$

* Time complexity = $O\left(\frac{n^2}{2} - \frac{n}{2}\right)$
 $= O\left(\frac{1}{2} \times n^2\right)$
 $= O(n^2)$ *

3. Insertion Sort (Adaptive)

In this technique the sequence is divided into sorted and unsorted and next one by one each element is taken from unsorted array and placed in the correct place of sorted array

Ex:

Pass 1: 53 27 65 15 37 72 30
 Sorted unsorted list

First element is considered as sorted

27 53 65 15 37 72 30
 unsorted list

65 > 53 so placed directly right

Pass 2: 27 53 65 15 37 72 30
 unsorted list

15 < 65, 15 < 53, 15 < 27
 so place leftmost

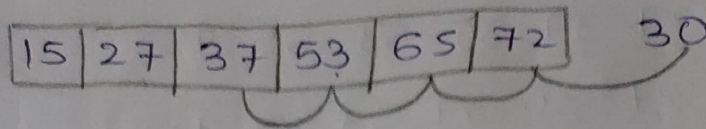
Pass 3: 15 27 53 65 37 72 30
 unsorted

37 < 65, 37 < 53 but
 37 > 27 so after 27 it is placed

Pass 4: 15 27 37 53 65 72 30
 unsorted

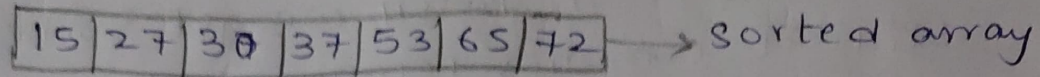
72 > 65 so place right

Pass 5:-



as 30 is greater than 27 so place right of 27

Pass 6:-



In this no. of comparisons depends on first element present in unsorted array.

* Time complexity is $O(n^2)$ *

Notes by Parth

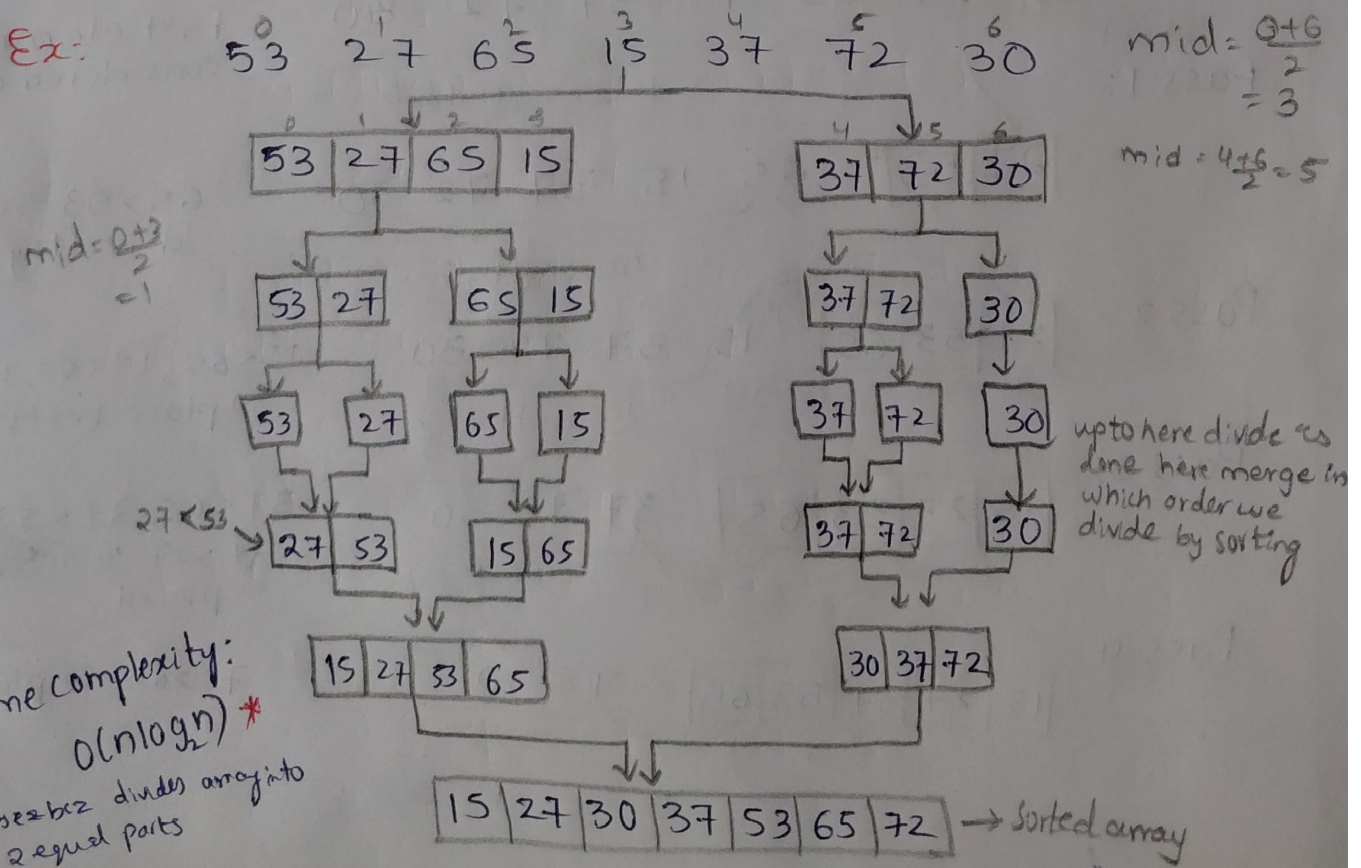
4. Merge Sort

This technique is also called as divide and conquer approach

- In this we divide sequence into two equal parts until each element is separated after dividing we have to merge conquer the elements in same order how we are dividing by placing elements in correct position

To calculate mid Formula = $\text{low} + \frac{(\text{high} - \text{low})}{2}$ for big numbers

For small elements that is in range of signed int formulate $\frac{\text{low} + \text{high}}{2}$ is also possible (-32766 to 32767)



Quick Sort

In this we have to select one element called as PIVOT element (the element that is to be placed in its right position) pivot element can be any element

Ex: $\textcircled{53}$ ^{$i=1$} 27 65 15 37 72 ^{$j=6$} 30
Pivot

consider 53 as pivot element then $i=1$ (starting) and $j=6$ (ending element)

Conditions:

if $A[i] \text{ not } > \text{pivot}$ then $i++$;

if $A[j] \leq \text{pivot}$ not satisfied then $j--$;

compare $i \ \& \ j$ ($i \leq j$) swap $a[i]$ and $a[j]$

if ($i > j$) swap $a[j]$ & pivot

Pass 1:- First $A[i]$

$A[1] = 27$ not $>$ pivot so, $i++ \Rightarrow i=2$

$A[j] = A[6] = 30 < \text{pivot}$ so don't perform

$j--$. So, j will be 6 only

Compare i and j :-

$i=2$ & $j=6$

$2 < 6$ so swap $a[2]$ and $a[6]$

$\textcircled{53}$ ^{$i=1$} 27 ^{$i=6$} 30 15 37 72 65

Now also pivot element is 53

now $i=2$ now again check conditions

$A[2] = 30$ not $>$ 53 $i++ \Rightarrow i=3$

$A[3] = 15$ not $>$ 53 $i++ \Rightarrow i=4$

$A[4] = 37$ not $>$ 53 $i++ \Rightarrow i=5$

$A[5] = 72 > 53$ now stops $i=5$

$j \Rightarrow A[6] = 65 \text{ not } < 53 \text{ so } j-- \Rightarrow j=5$

$A[5] = 72 \text{ not } < 53 \text{ so } j-- \Rightarrow j=4$

$A[4] = 37 < 53 \text{ now stops at } j=4$

now compare $i \& j$ ($5 > 4$) ($i > j$) swap $a[i] = 37$ & pivot 53

37 27 30 15 53 72 65
pivot

now pivot element position changes now 2nd pass

Pass 2: $i=1$ $j=3$
37 27 30 15 53 72 65
Pivot Pivot

take 53 pivot left side as separate list and perform same operations by taking one pivot element (here 37 taken) and also take $i \& j$ values accordingly

37 27 30 15
Pivot $i=1$ $j=3$

$A[1] = 27 \text{ not } > 37 \rightarrow i++ \Rightarrow i=2$

$A[2] = 30 \text{ not } > 37 \rightarrow i++ \Rightarrow i=3$

$A[3] = 15 \text{ not } > 37 \rightarrow i++ \Rightarrow i=4$

$j \Rightarrow A[3] = 15 < 37 \text{ so dont decrement } j$

$i=4$ $j=3$

$i > j$ ($4 > 3$) swap $a[i] \rightarrow a[3] = 15$ with pivot 37

15 27 30 37

53 pivot left side is completed then right elements are considered

72 65
Pivot

$i \Rightarrow A[1] = 65 \text{ not } > 72 \Rightarrow i++ \Rightarrow i=2$

$j \Rightarrow A[1] = 65 < 72 \text{ so } j \text{ not decremented}$

$i=2$ $j=1$

$i > j$ ($2 > 1$) swap $a[i] \Rightarrow a[1] = 65$ with pivot 72

Now list be comes

15 27 30 37 53 65 72

Pass 3: Right side elements of 53 pivot is completed started now we have left list of 37

$i=1$ $j=2$
15 27 30
↓
pivot

if we perform same operations as above i not changes but j decrements to '0' and pivot also at '0' so no swapping

Pass 4:

15 27 30 37 53 65 72

now is right side list 27 $i=1$ $j=1$ 30 is taken
it is also like pass 3 so no swapping are done
As all elements are considered it is completed
final list

15 27 30 37 53 65 72

THIS NOTES BELONGS TO N.PARTH