Static and Dynamic List

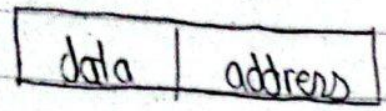


fig: node

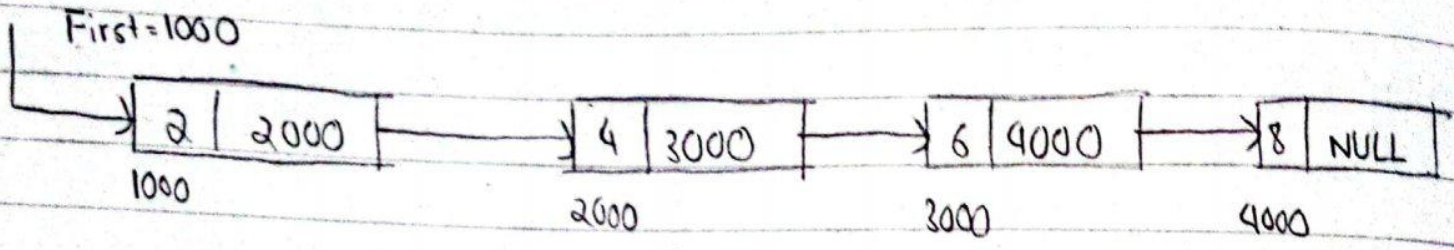


fig: linked list

A linked list is a series of connected nodes. Each node contain at least a piece of data (any type) and pointer to the next node in the list

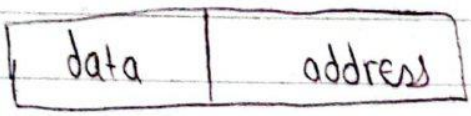


Fig: Node

Implementation of linked list

1) Start Implementation: using array

2) Dynamic Implementation

Static Linked List

Example

List 1: 50, 100, 300, 200
List 2: 5, 10, 15, 20

	Index	Data	Next (index)
	0	100	5
	1		
List 2 =	2	5	4
	3		
	4	10	7
	5	300	14
	6		
	7	15	10
	8		
	9	~~20~~	~~-1~~
	10	250	-1
List 1 =	11	~~50~~	0
	12		
	13		
	14	200	10

Types of linked list

1) Singly linked list
   - 1.1 Singly linear linked list
   - 1.2 Singly circular linked list

2) Doubly linked list
   - 2.1 Doubly linear linked list
   - 2.2 Doubly linear linked list

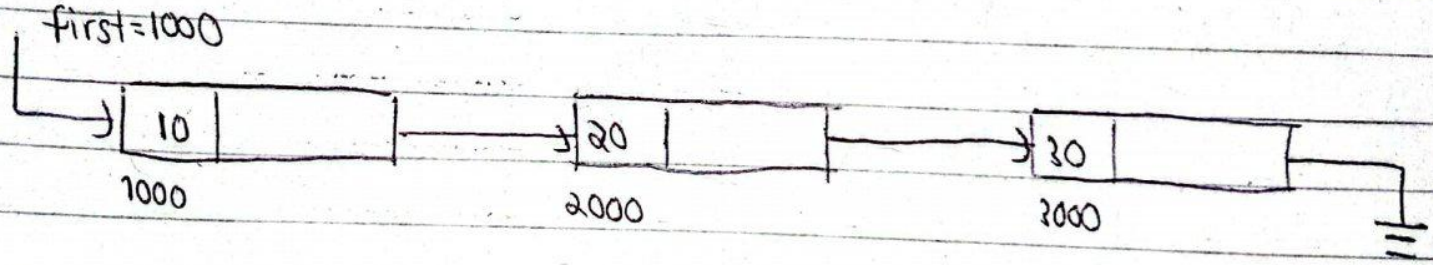


fig: Singly linear linked list

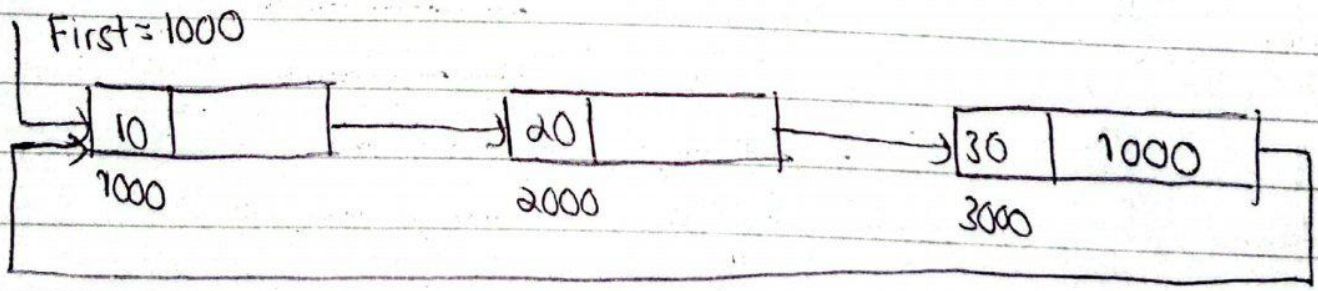


fig: Singly circular linked list

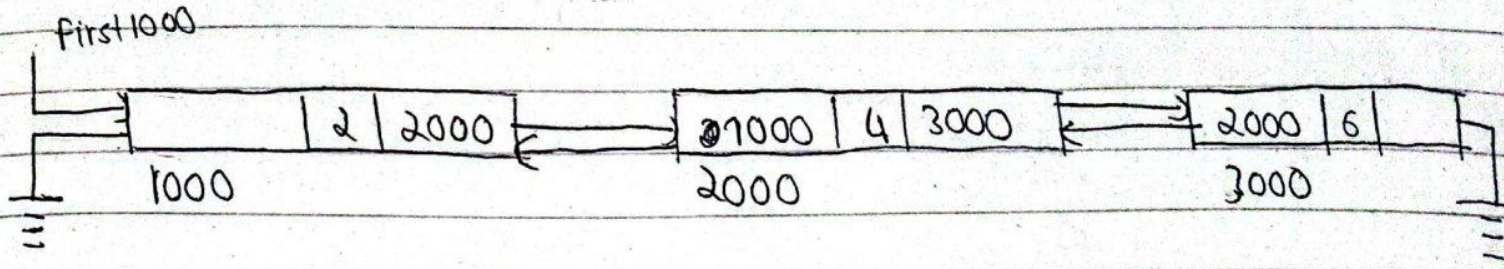


Fig: Doubly linear linked list

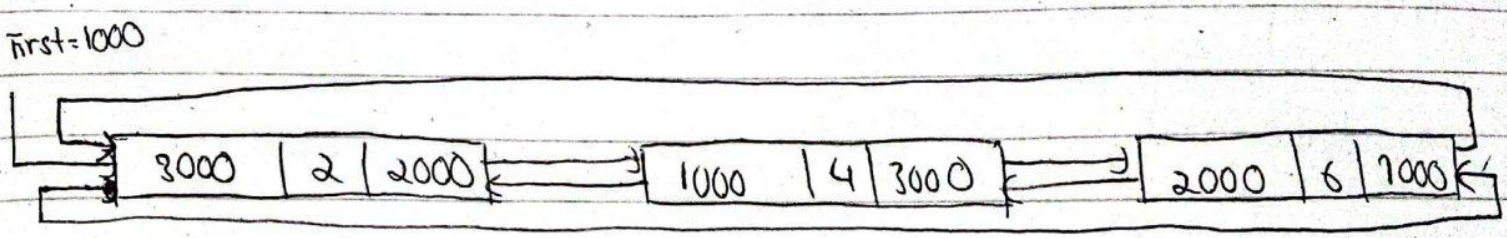


Fig: Doubly circular linked list

Singly linked list

Insertion in a Singly linked list

Node creation

```
struct linked-list {
    int info;
    struct linked-list *next;
};

typedef struct linked-list SLL;
```

1) `SLL *First = NULL, *p, *q;`

2) `p = (SLL *) malloc (sizeof (SLL));`

3) p → info = 12;
4) p → next = NULL;
5) first = p;

Case 1: Insert as first node

1) temp = (SLL *) malloc (sizeof(SLL))

2) temp → info = 12 (data)

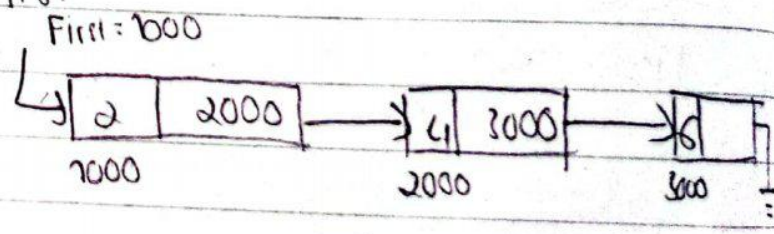
3) temp → next = first

4) first = temp

Singly linear linked list

Insertion in Singly linear linked list

Case 1: Insert as first node



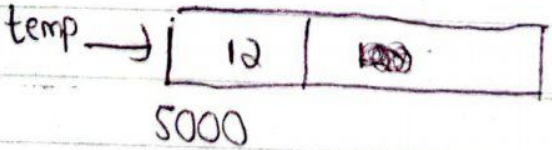


fig: Before insertion

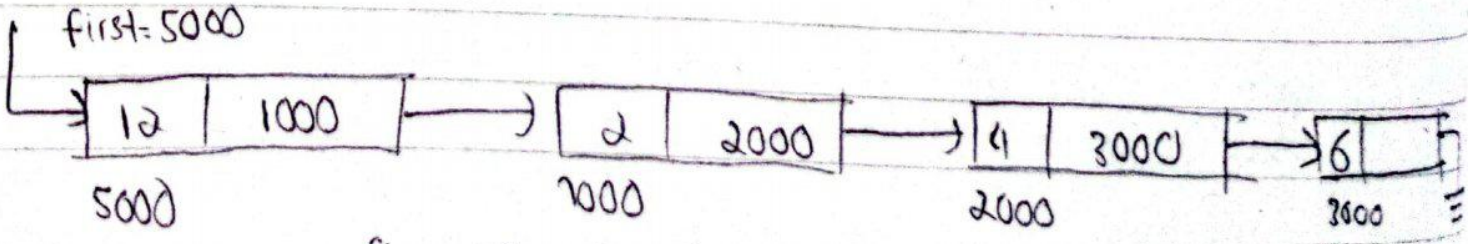


fig: After insertion

Case 2: Insert as last node

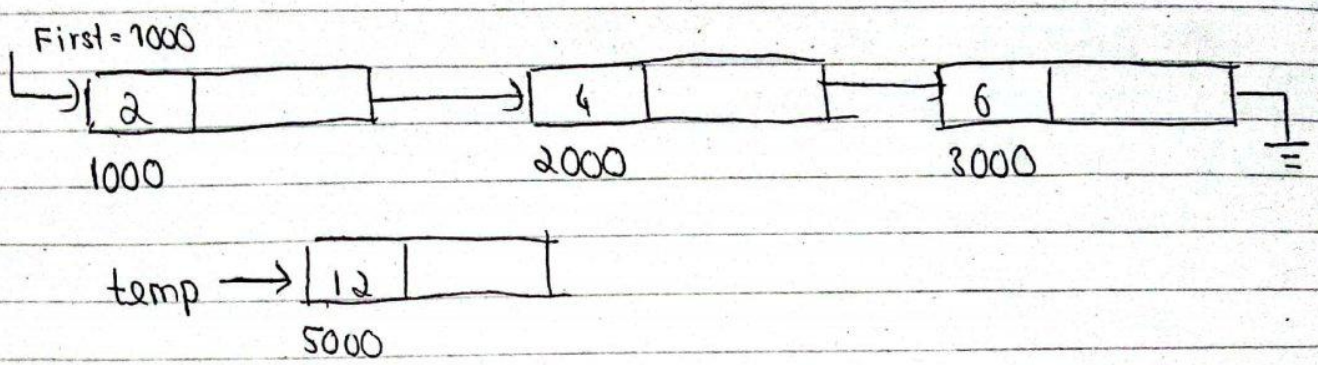


fig: Before insertion

1) temp = (SLL *) malloc (sizeof(SLL));
2) temp → info = data
3) P = first
4) while (p → next ≠ NULL)
5) p = p → next
6) End while
7) P → next = temp
8) temp → next = NULL

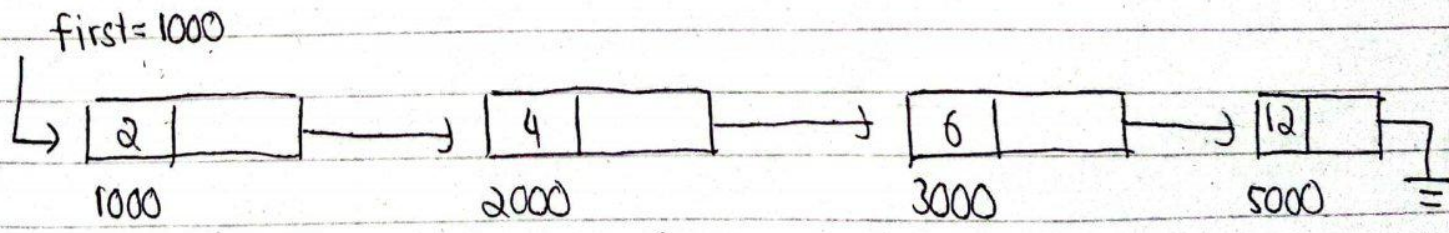


fig: After insertion.

Case 3: Insert as n^{th} node

1) temp = (SLL *) malloc(sizeof(SLL));
2) temp -> info = data
3) p = first
4) while (n>2)
5) p = p -> next
6) n--
7) End while
8) temp -> next = p -> next
9) p -> next = temp

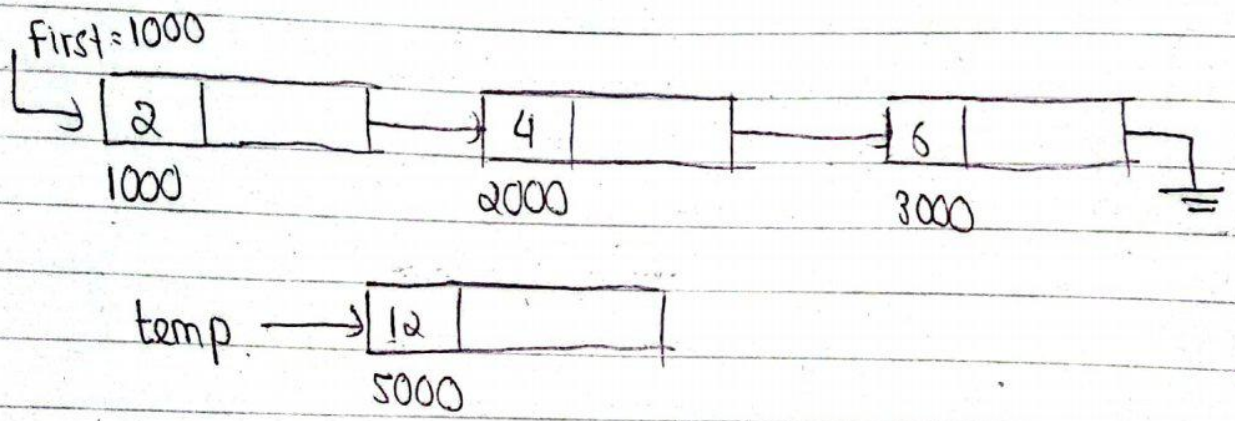


fig: Before insertion

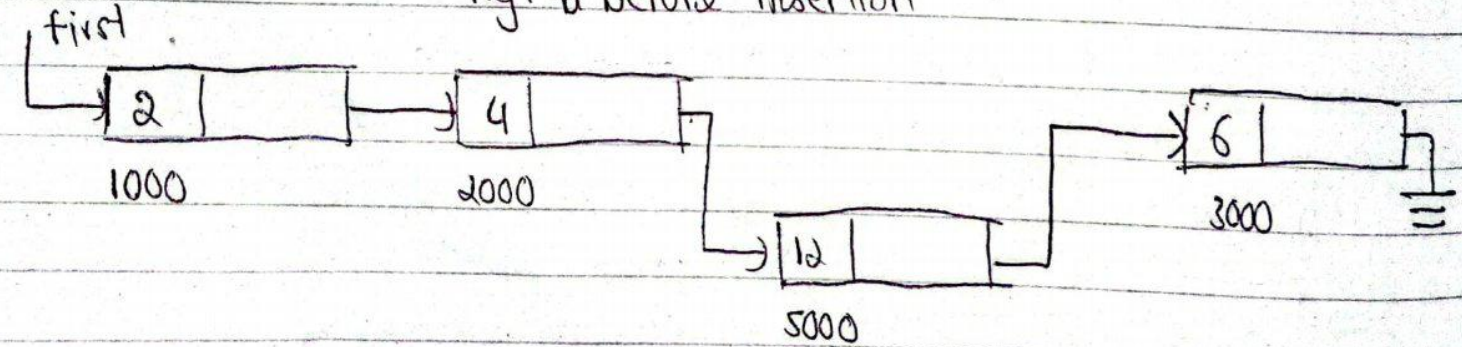
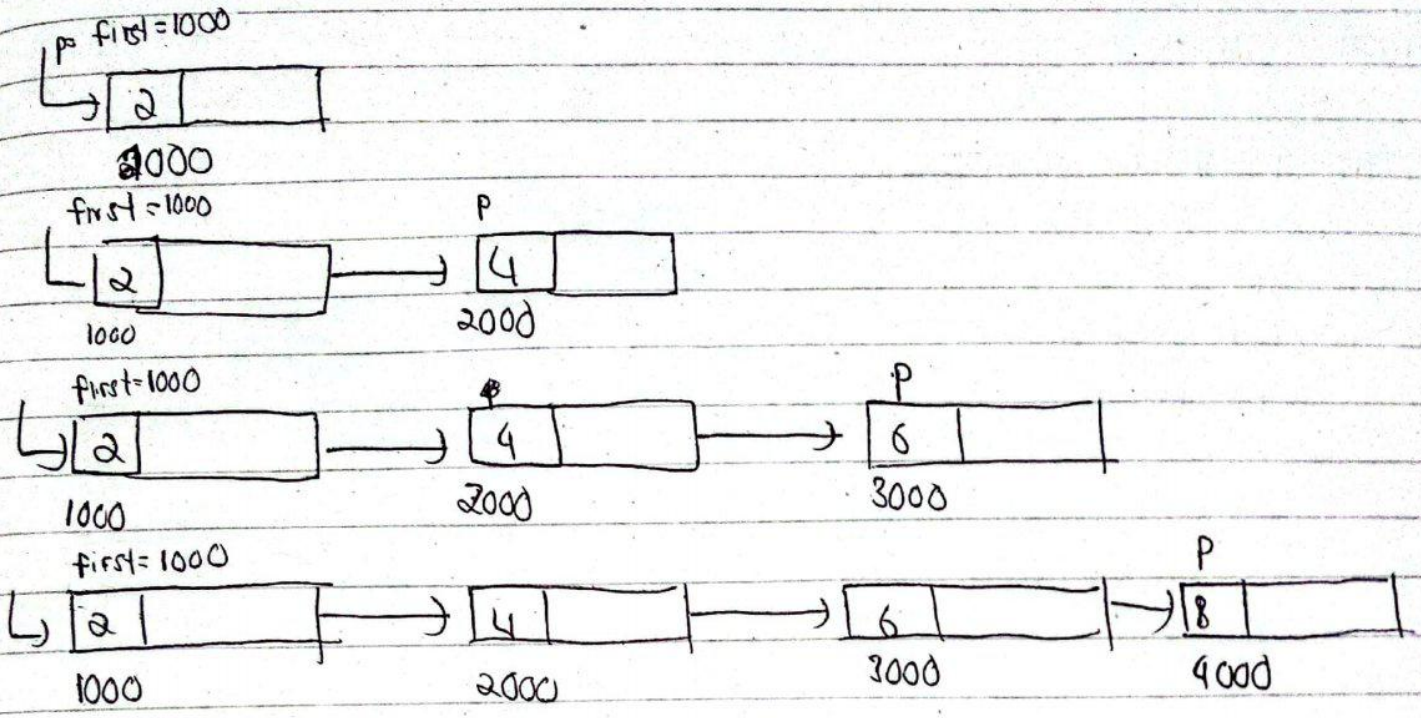


fig: After insertion



WAP to display all the values stored in SLL.

~~#include <iostream>~~

~~using name~~

1) p = first
2) while(p → next ≠ NULL)
3) Display p → info
4) ~~End while~~ p = p → next
5) End while
6) Display p → info

Deletion in SLL

Case 1: Delete first node

1) temp = first
2) first = first → next [or next = first]
3) free(temp)

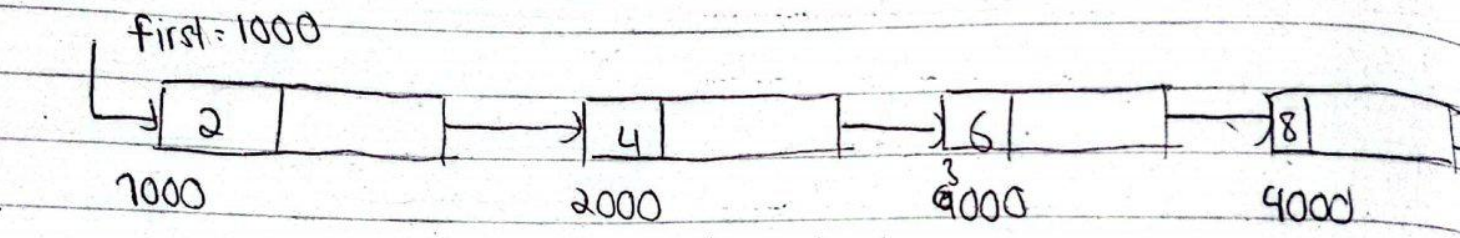


fig: Before ~~delete~~ deletion

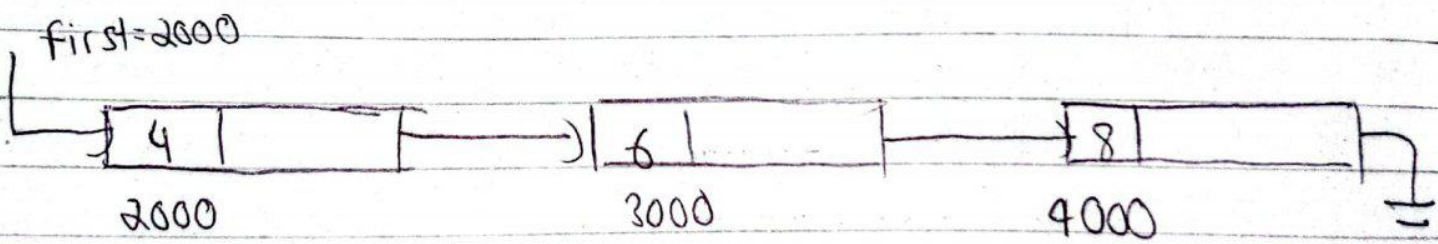


fig: After deletion

Case 2: Delete last node

1) p = first
2) while(p → next → next ≠ NULL)
3) p = p → next
4) End while
5) temp = p → next
6) p → next = NULL
7) free (temp)

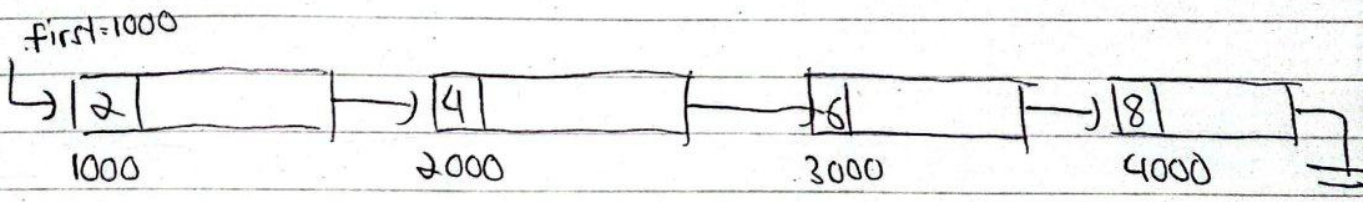


figure: Before Deletion

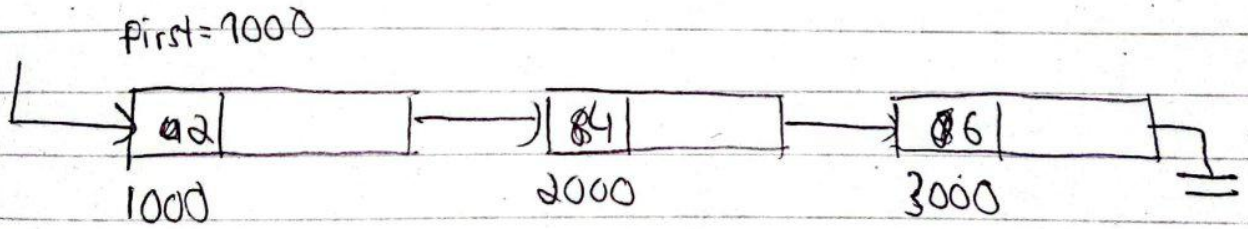
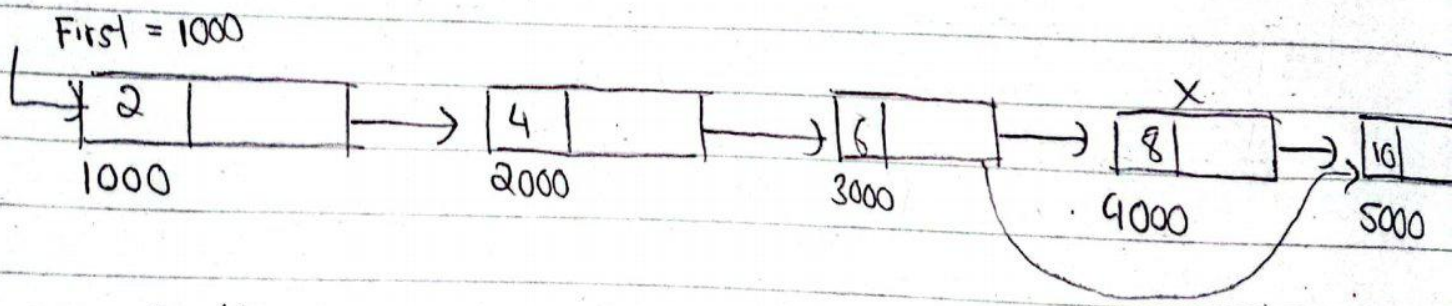


fig: After deletion

Case 3: Delete n^{th} node

1) p = first
2) while (n > 2)
3) p = p → next
4) n--
5) End while
6) temp = p → next
7) p → next = p → next → next
8) free (temp)



n = 4

fig: Before Deletion

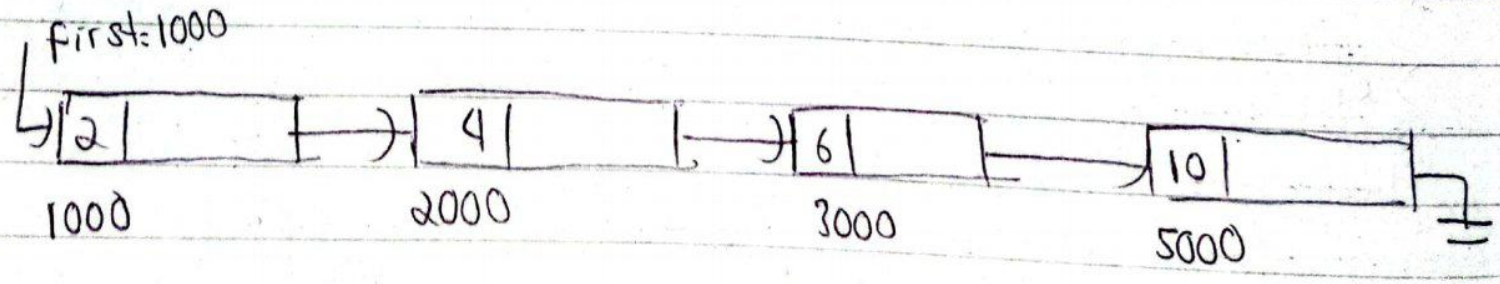


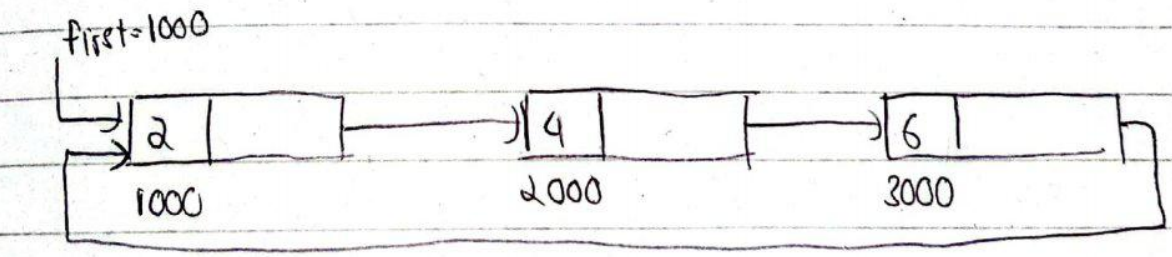
fig: After Deletion.

Singly circular linked list (SCLL)

Insertion in SCLL

Case 1: Insert as first node

1) temp = (SCLL *) malloc (sizeof (SCLL));
2) temp → info = value;
3) p = first
4) while (p → next ≠ first)
5) p = p → next
6) End while
7) p → next = temp
8) temp → next = first
9) first = temp



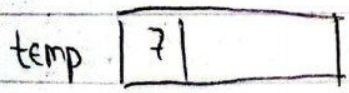


fig: Before insertion

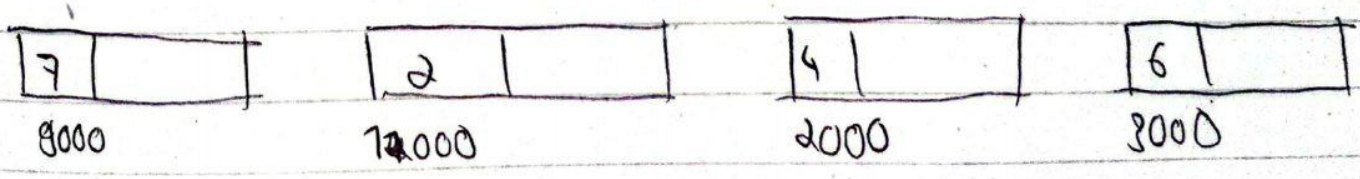


fig: After insertion

Case 2: Insert as last node

1) temp = (SCLL*) malloc (sizeof(SCLL));
2) temp → info = value;
3) p = first
4) while(p → next ≠ first)
5) p = p → next
6) End while
7) p → next = temp
8) temp → next = first

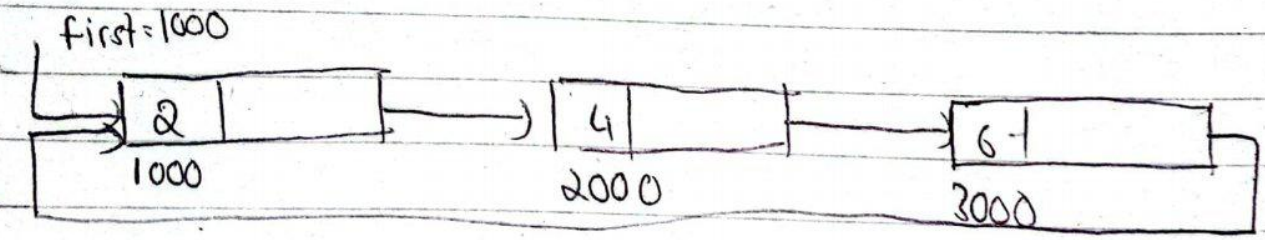


fig: Before insertion

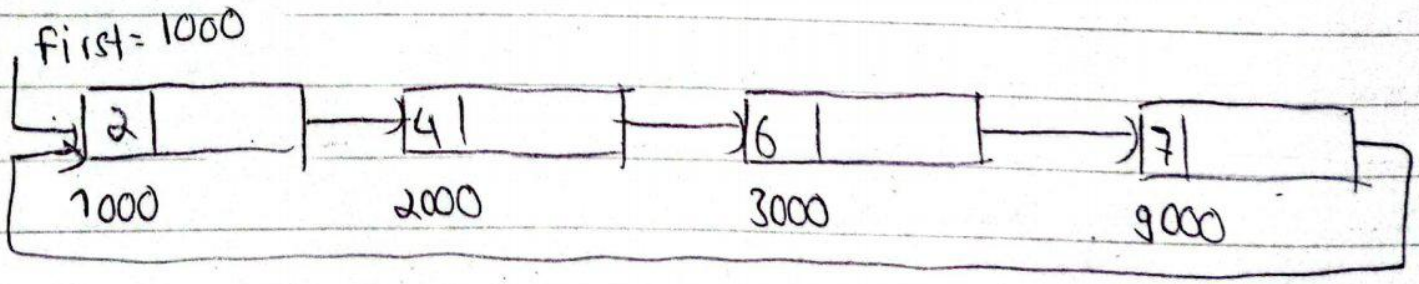
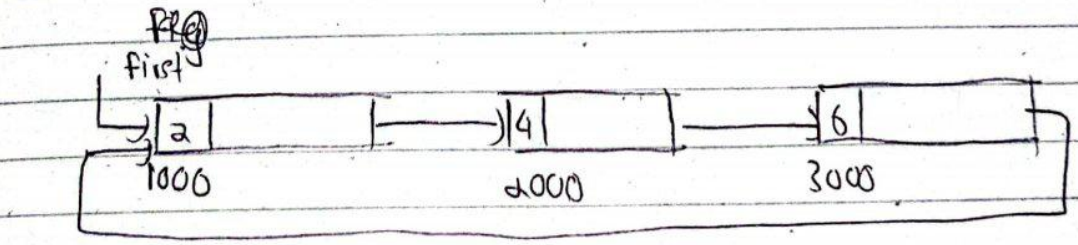
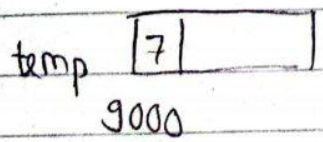


fig: After insertion

Case 3: Insert as n^{th} node

1) temp = (SCLL *) malloc (sizeof(SCLL))
2) temp → info = value
3) p = first
4) while (n > 2)
5) p = p → next
6) n--
7) End while
8) temp → next = p → next
9) p → next = temp





n=3

fig: Before insertion

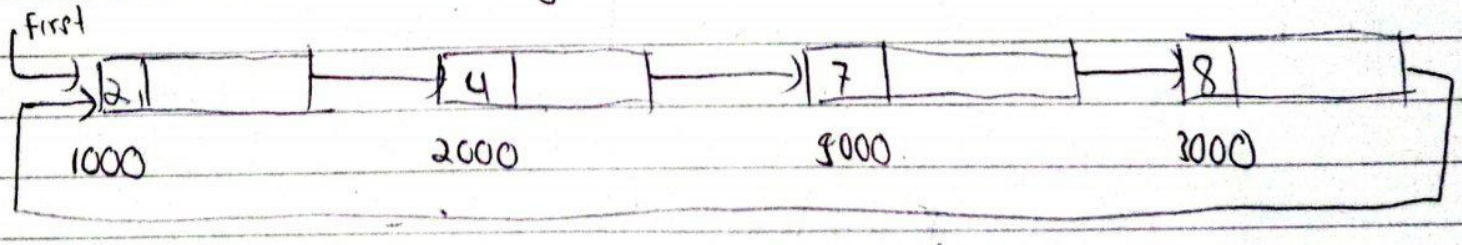
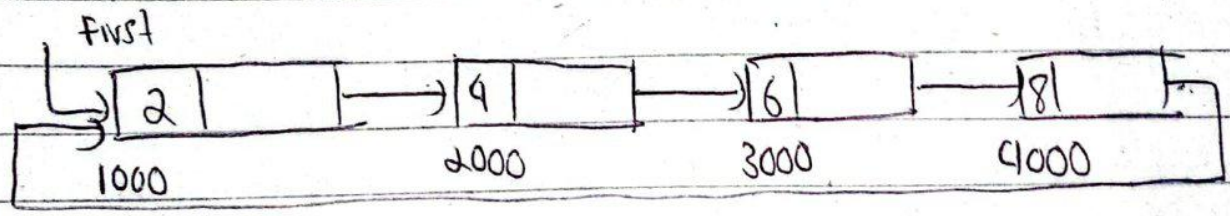


fig: After insertion

Deletion in SCLL

Case 1: Delete first node

1) p = first
2) while (p → next ≠ first)
3) p = p → next
4) End while
5) temp = first
6) first = first → next
7) p → next = first
8) free (temp)



Before Deletion

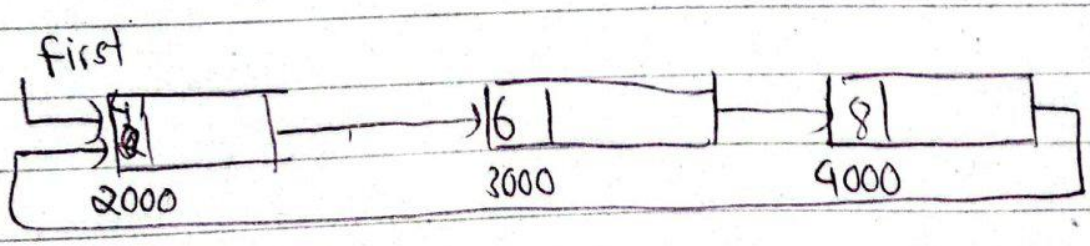
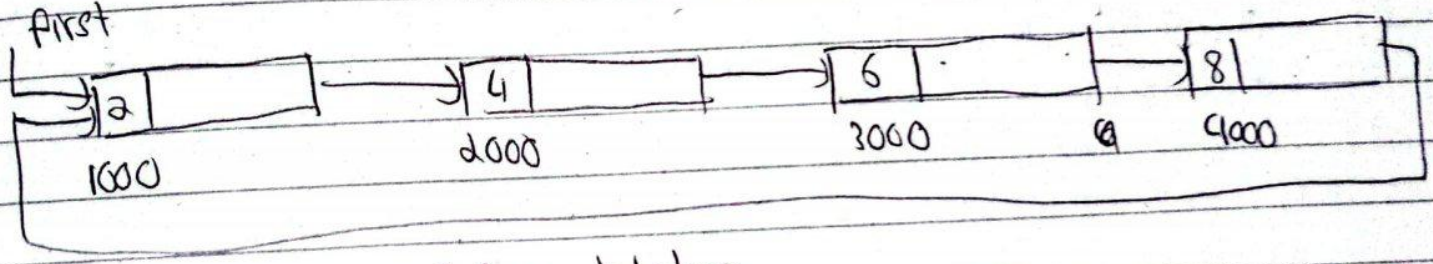


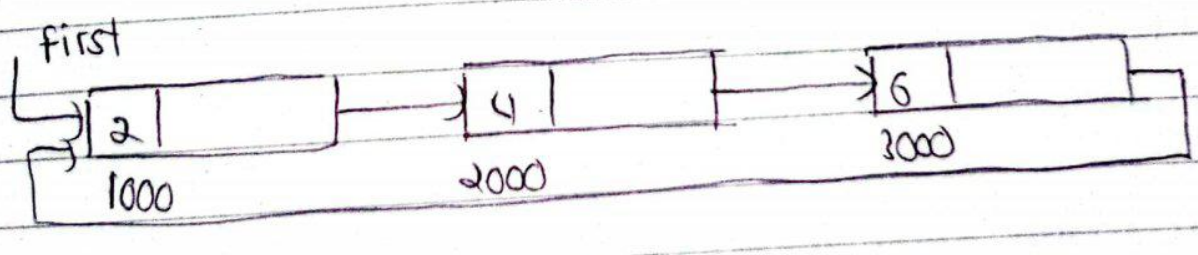
fig: After Deletion

Case 2: Delete last node

1) p = first
2) while(p → next → next ≠ first)
3) p = p → next
4) End while
5) temp = p → next
6) p → next = first
7) free (temp)



Before deletion



After deletion

Case 3: Delete n^{th} node

→ Same as SLL

Insert at any position in SCLL

```
1)  temp = (SCLL**) malloc(sizeof(SCLL))
2)  temp → next = NULL
3)  temp → data = value
4)  If first = NULL Then
5)  first = temp
6)  first → next = first
7)  End If
8)  IF pos ≤ 1 Then
9)  p = first
10  while (p → next ≠ first)
11) p = p → next
12) End While
13) temp → next = first
14) p → next = temp
15) first = temp
16) else if pos ≥ N
17) p = first
18) while( p → next ≠ first)
19) p = p → next
20) End While
21) p → next = temp
```

22) temp → next = first
23) else
24) while pos > 2 Then
25) p = p → next
26) pos--
27) End while
29) temp → next = p → next
29 p → next = temp

Doubly linear Linked list (DLL)

Creating a node

```
Struct doubly_linear_list
{
    int info;

    struct doubly_linear_list *next;
    struct doubly_Linear_list * prev;

};

typedef struct doubly_linear_list DLL;
DLL *
temp = (DLL *) malloc (sizeof (DLL));
temp → info = value;
temp → next = first
```

Single node DLL

1) temp = (DLL *) malloc(sizeof(DLL))
2) temp → info = value
3) If first == NULL
4) temp → prev = NULL
5) temp → next = NULL
6) first = temp

Insertion in ~~first~~ node DLL

Insert in 1st node

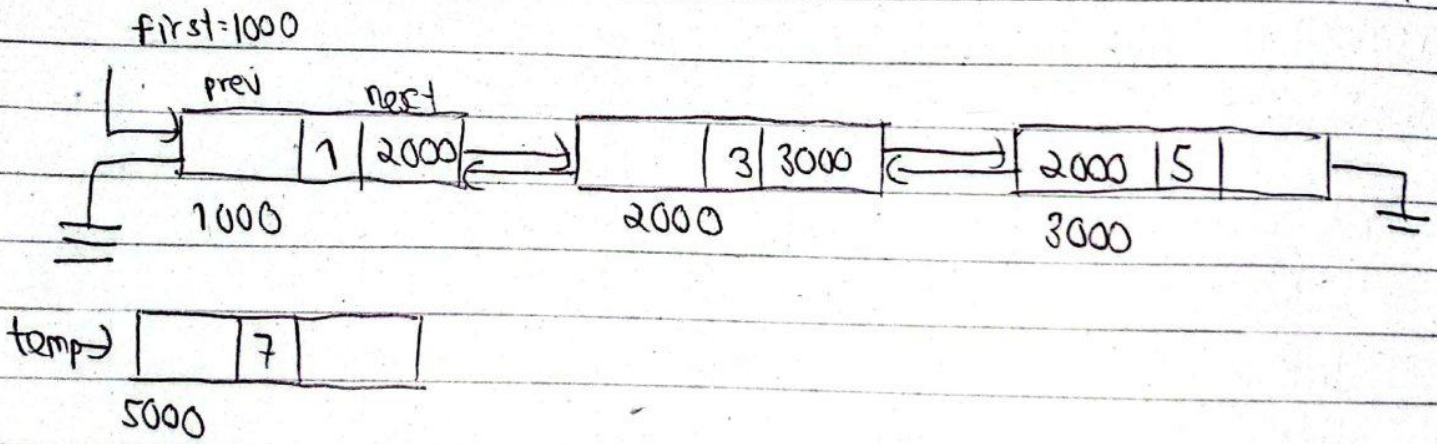


fig: Before insertion

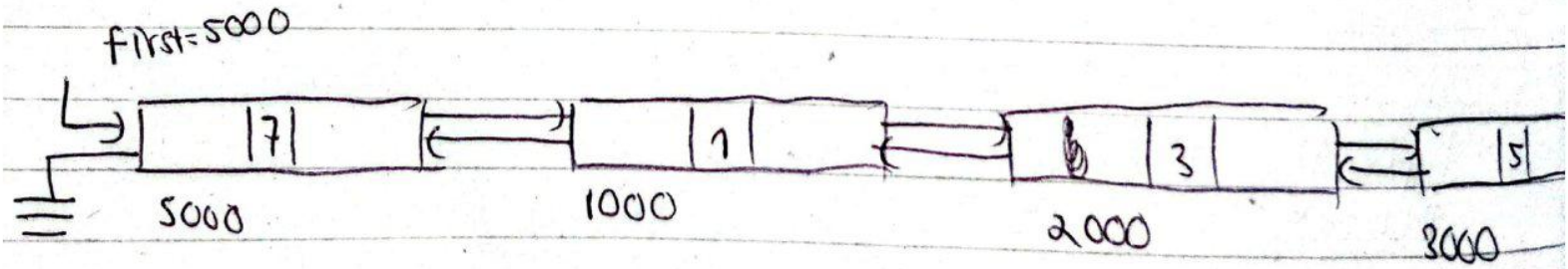


fig: After insertion

1) temp = (DLL*) malloc (sizeof(DLL))
2) temp → info = value
3) temp → next = first
4) first → prev = temp
5) first = temp
6) first → prev = NULL

Insertion in last node

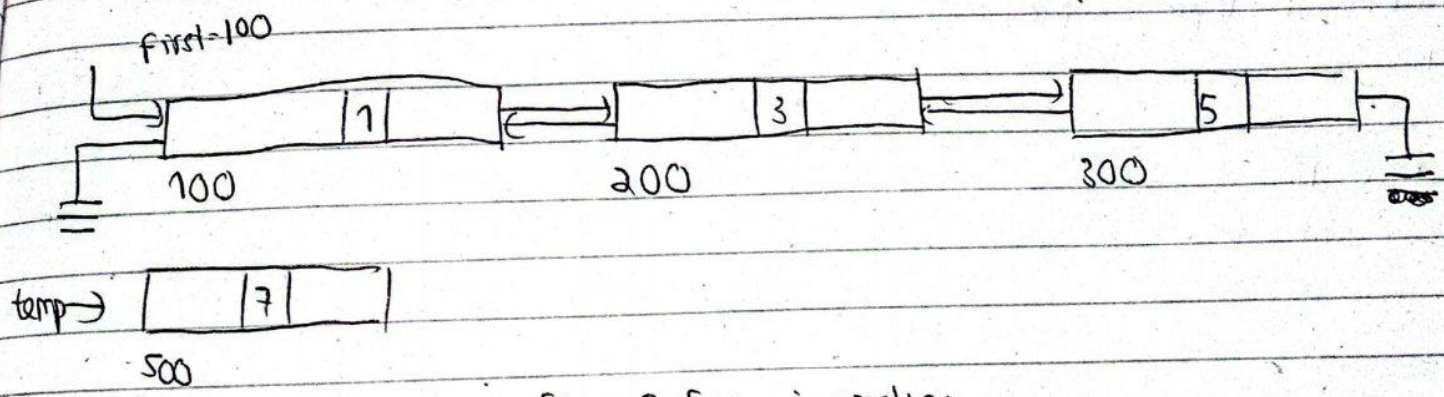


fig: Before insertion

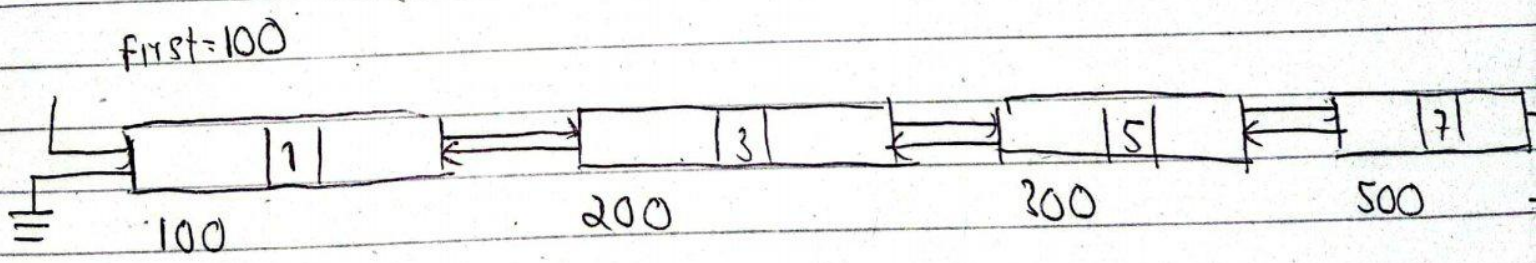


fig: After insertion

1) temp = (DLL *) malloc(sizeof(DLL));
2) temp → info = value
3) temp → next = NULL
4) p = first
5) while (p → next ≠ NULL)
6) ~~p → next = temp~~ p = p → next
7) ~~temp → prev = p~~ End while
8) p → next = temp
~~case III~~ 9) temp → prev = p

case III
Insert in nth node

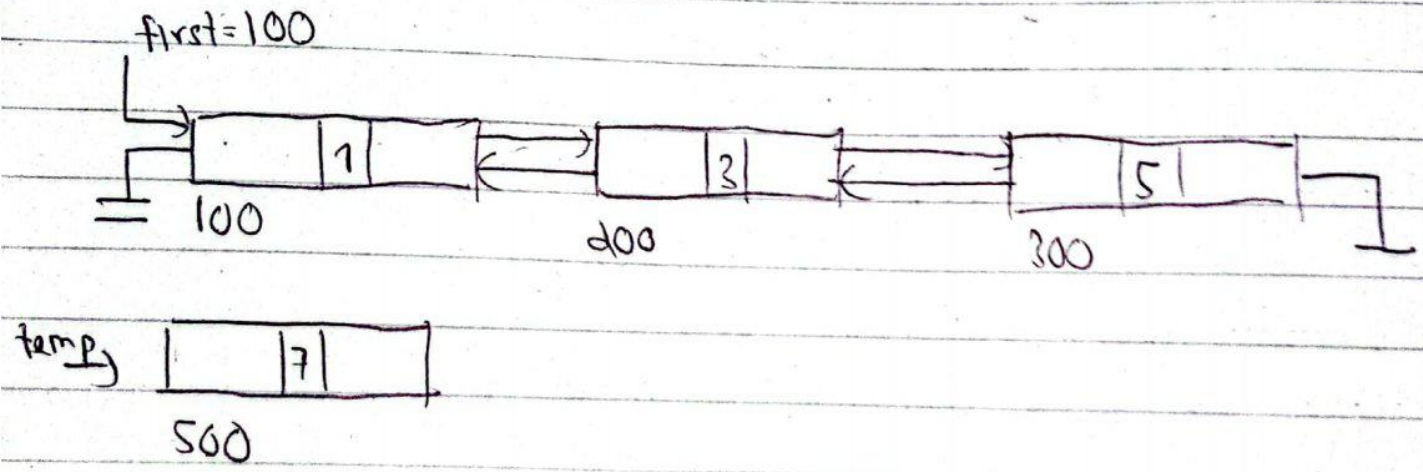


fig: Before insertion

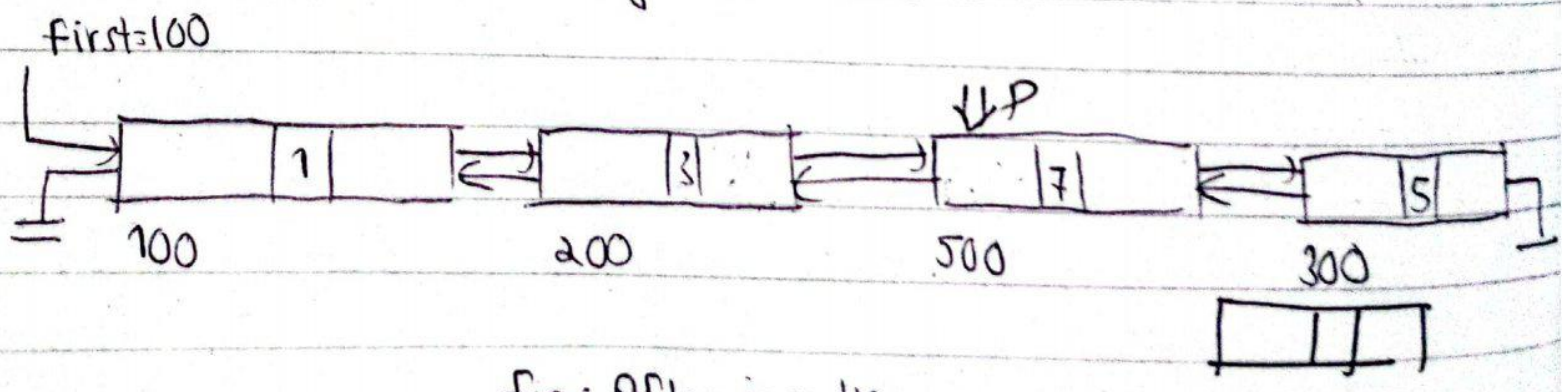


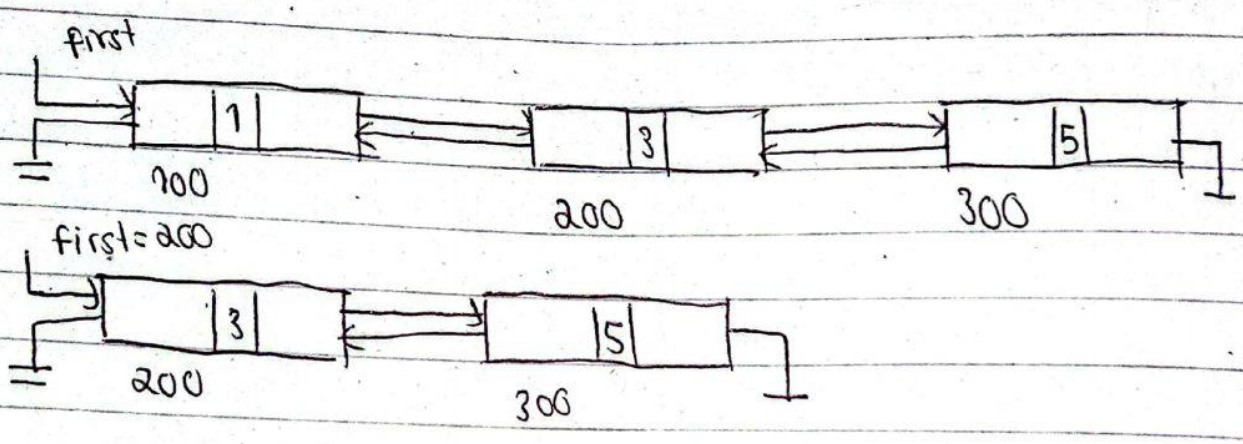
fig: After insertion

1) temp = (DLL *) malloc (sizeof (DLL))
2) temp → info = value
3) p = first
4) while n ≥ 2
5) p = p → next
6) n--
7) End while
8) temp → next = p → next
9) temp → next → prev = temp
8) p → next = temp
9) temp → prev = p
10) temp = p → next → next
11) p → next → next → prev = temp

Deletion in DLL

Case I:
Delete first node

1) temp = first
2) first = first → ~~node~~ next
3) first → prev = NULL
4) Free (temp)



Case 2:
Delete last node

1) p = first
2) while (p→next→next ≠ NULL)
3) p = p→next
4) End while
5) temp = p→next
6) p→next = NULL
7) free (temp)

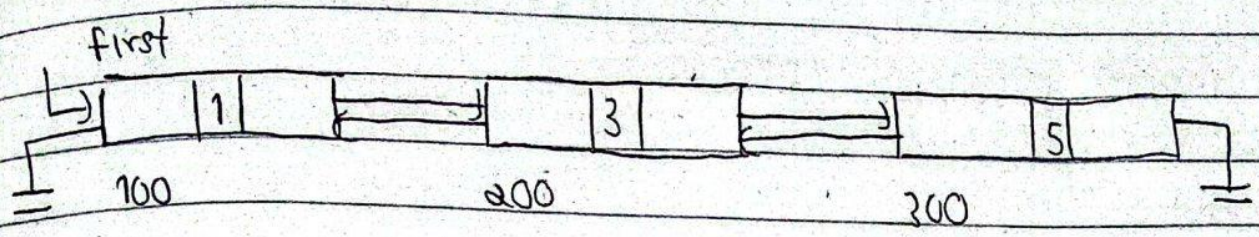
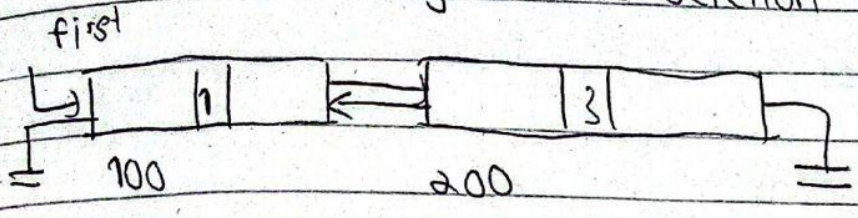


fig: Before Deletion



Deletion of n^{th} node

1) p = first
2) while (n > 2)
3) p = p → next
4) n--
5) End while
6) temp = p → next
7) p → next = p → next → next
8) p → next → prev = p
9) free (temp)

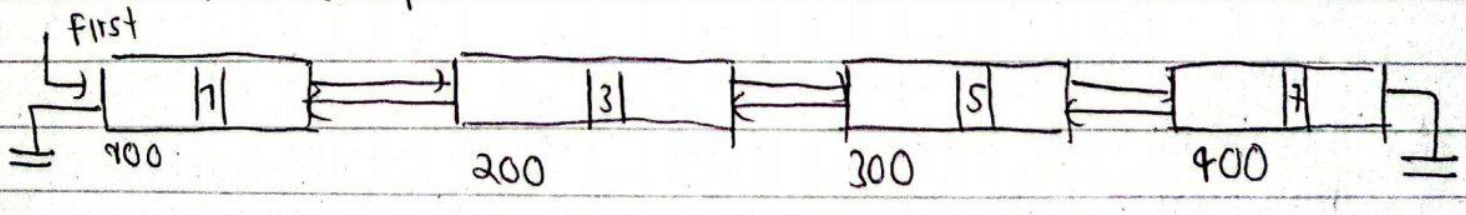


fig: Before Deletion

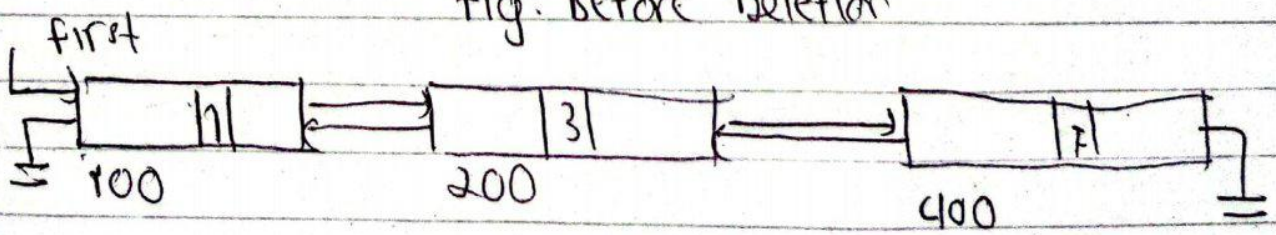


fig: After Deletion

Doubly circular linked list (DCLL)

Node Creation

Same as DLL

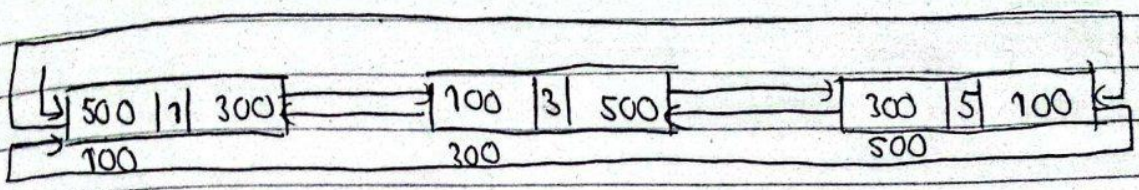
```
typedef struct doubly_circular_list DCLL;

DCLL *first = NULL, *temp, *p, *q
```

Insertion in DCLL

Case I: Insert in last node

1) p = first
2) while (p->next ≠ first)
3) p = p->next
4) End while
5) temp = (DCLL*) malloc(sizeof(DCLL))
6) temp->info = value
7) temp->next = first
8) p->next = temp
9) temp->prev = p
10) first->prev = temp



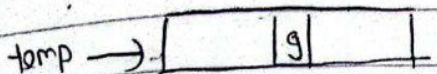


fig: Before insertion

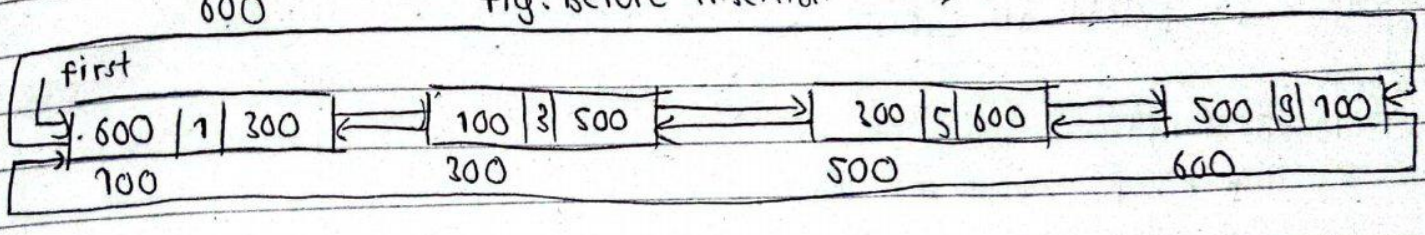


fig: After insertion

Case II: Insert as first node

1) p = first
2) while (p→next ≠ first)
3) p = p→next
4) End while
5) temp = (DCLL*) malloc (sizeof(DCLL))
6) temp→info = value
7) temp→next = first
8) p→next = temp
9) temp→prev = p
10) first→prev = temp
11) first = temp

Case III: Insert nth node

Same as DLL

Deletion in DCLL

Case I: Delete first node

1) p = first
2) while (p→next ≠ first)
3) p = p→next
4) End while
5) temp = first
6) first = first→next
7) p→next = first
8) first→prev = p
9) free (temp

Case II: Delete last node

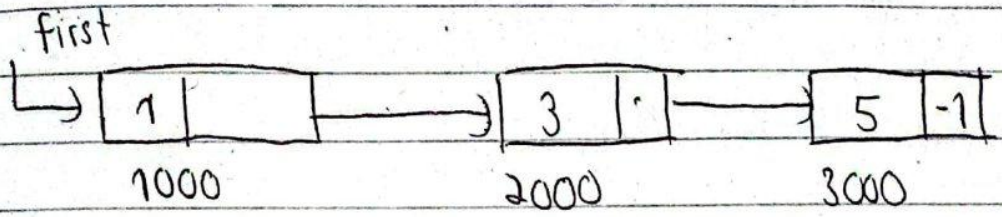
1) p = first
2) while (p→next→next ≠ first)
3) p = p→next
4) End while
5) ~~p→next~~ = temp = p→next
6) p→next = first
7) first→prev = p
8) ~~temp (free)~~ free (temp)

Case III: Delete nth node

Same as DLL

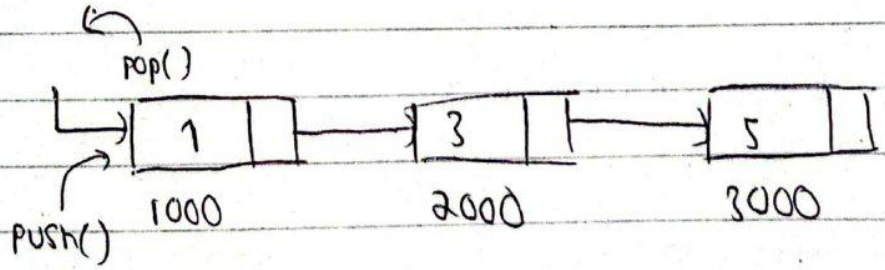
Static dynamic implementation (FILO)

1) Singly linear linked list



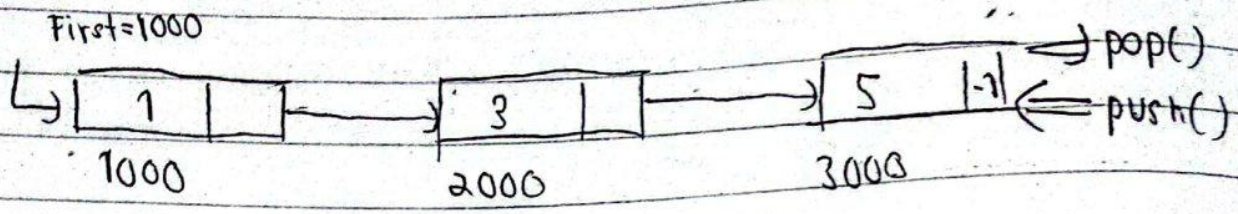
So as to realize the stack we perform both the operations push() and pop() at the same ~~time~~ end

1) Insert node at First and delete first node.



push(): add a node at first
pop(): remove a node

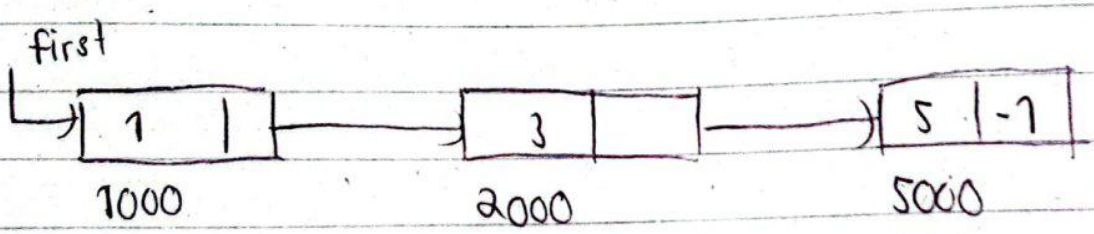
2) Insert node at last and delete last node



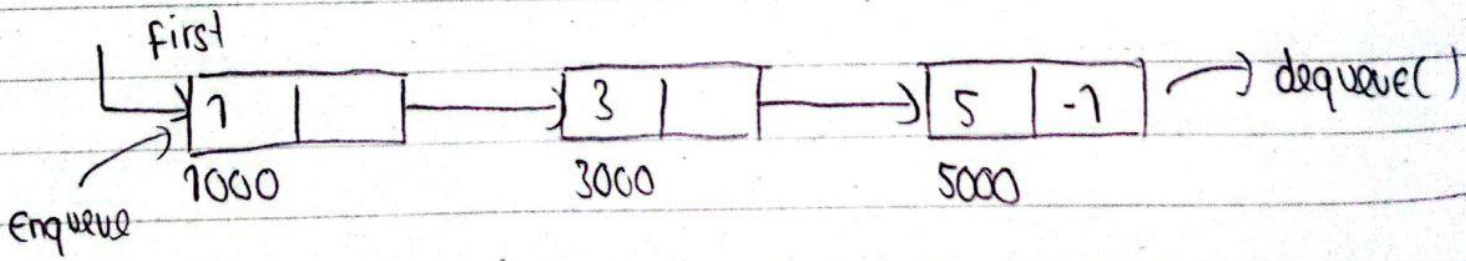
push(): add node at last
pop(): delete last node

Dynamic implementation of ~~queu~~ Queue

Using Singly linear Linked List

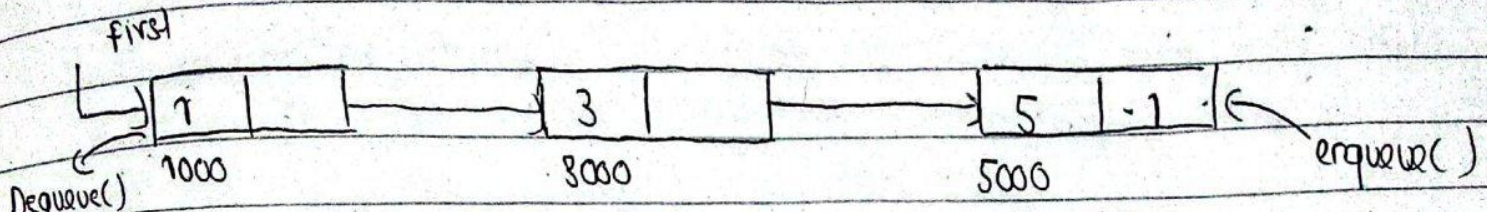


1) Enqueue at first and dequeue at end



Enqueue(): Insert node at first
Dequeue(): delete last node

2) Enqueue at end and dequeue at first



Enqueue(): Insert node at last
Dequeue(): delete first node

Advantages of linked list

1) Linked list are dynamic data structure

2) Size is not fixed

3) Data are store in non-continuous memory block

4) Insertion and deletion of nodes are easier and efficient

5) Real life complex problems can be easily modeled using linked list.

Disadvantages

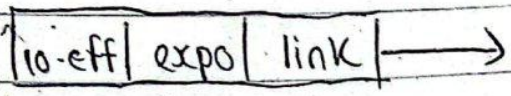
1) More memory is required to store the memory address

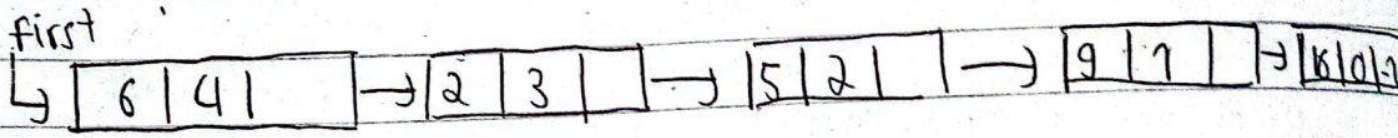
2) If head pointer is overwritten the whole list is lost

3) Accessing arbitrary node is time consuming.

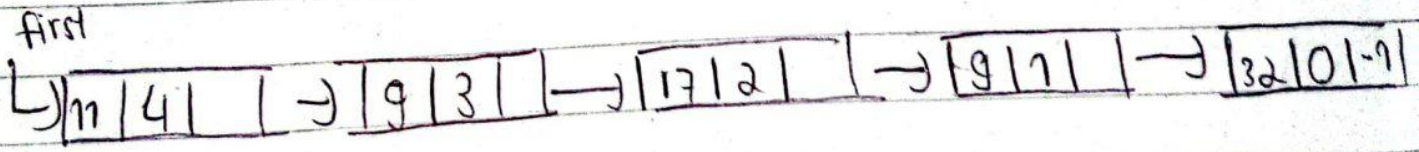
Linked list and polynomials

We can store polynomials like $6x^4 + 2x^3 + 9x + 16$ using linked list where each of polynomial will be represented by a node.





$5x^4 + 7x^3 + 12x^2 + 16$



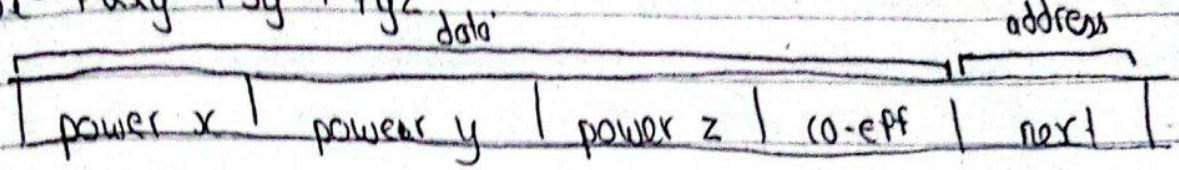
Algorithm

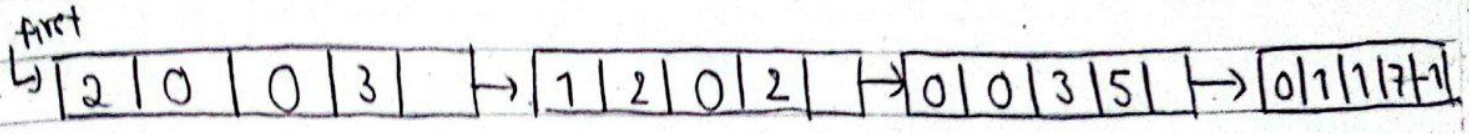
Add 2 polynomial

1) Start
2) ~~Add the to~~ ~~while(~~ p = first
3) while (p → next ≠ NULL)
4)

Multivariable Polynomial

$3x^2 + 2xy^2 + 5y^3 + 7yz$





Tree: Non Linear data structure

Tree is a non linear (hierarchical) data structure consisting of data nodes connected to each other in a same way as a linked list. Each node in a tree maybe connected to two or more nodes and order of tree is determined by a node connected to maximum number of nodes.

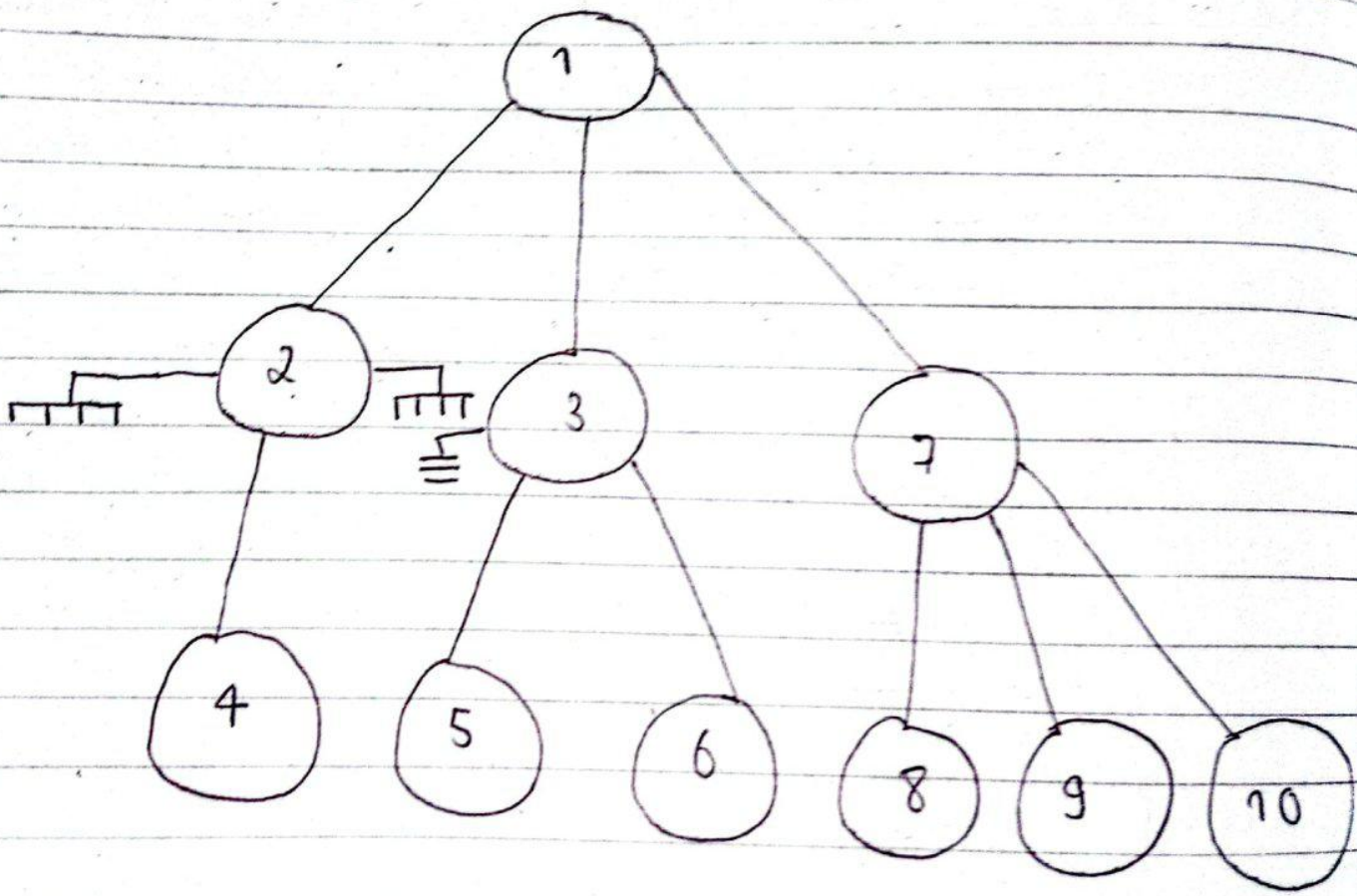
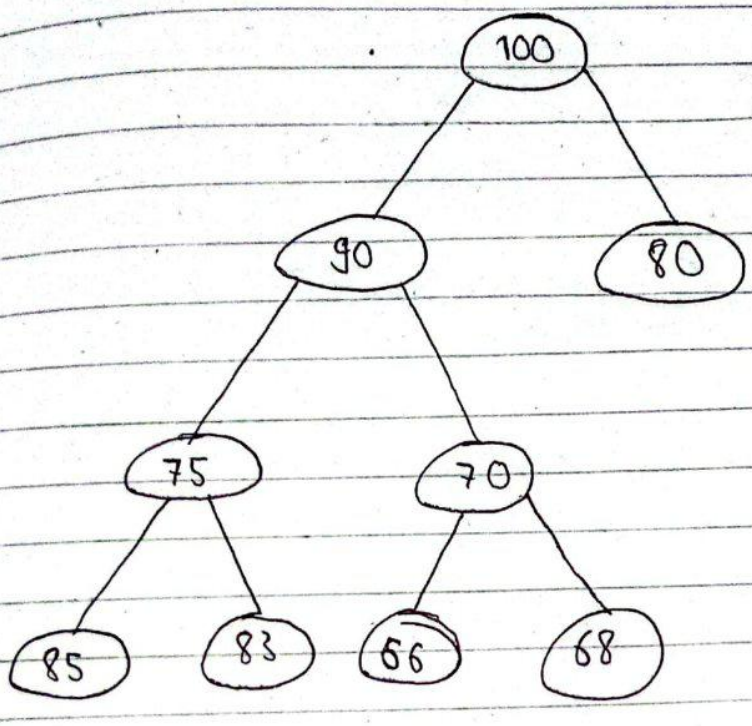


fig: Tree of order 3

Binary tree
Tree having order 2



Terminologies

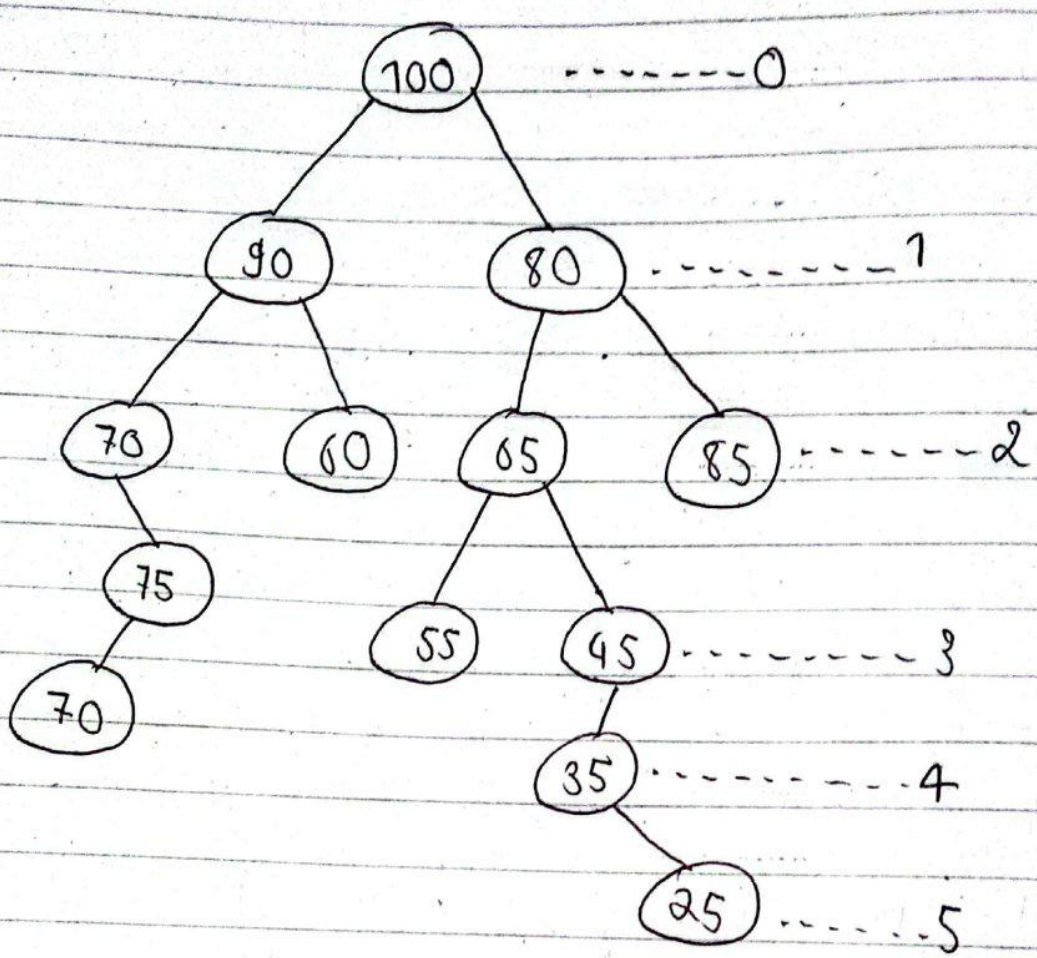
1) Root: First node of a tree i.e node 100

2) leaf node (external node): Node which doesnot have any left or right child node eg: 85, 83, 66, 68, 65, 60.

3) Non-leaf node (internal node): Node having at least on left or right child eg: 75, 70, 90, 80, 100

4) left sub tree: A sub tree which belongs to the left child of any node

5) Right sub tree: A sub tree which belongs to the right child of any node



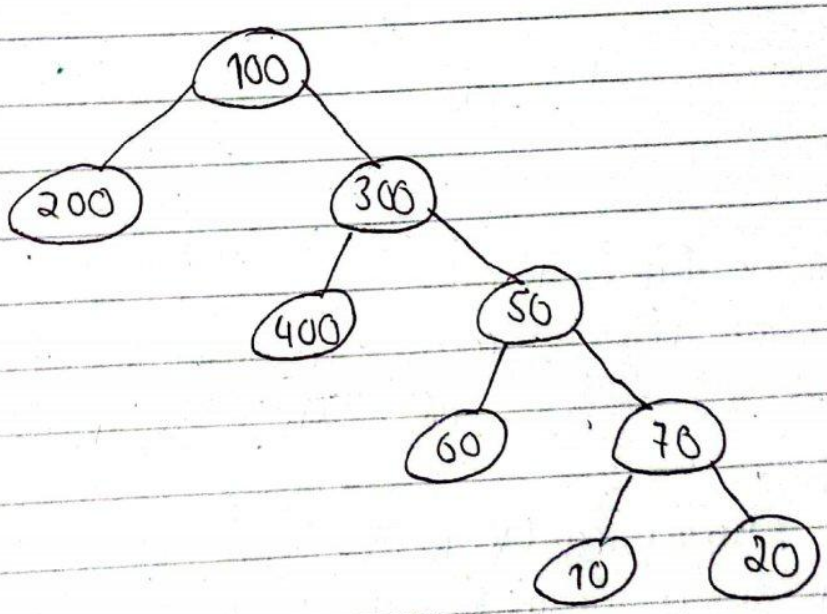
a) Depth:
Depth of a tree is the maximum number of edges/nodes from root to leaf here the depth is 5

b) Height:
Height of a node in a binary tree is the number of edges in the longest path from the node of interest to leaf
Height of 65 is 3.

c) level
level of root node is zero and other ~~nother~~ nodes have 1 level more than parent node or predecessor node.

* Type of Binary Tree

1) Strictly Binary Tree(SBT)
If every non leaf node in a binary tree has non empty left and right sub tree then its a SBT



2) Complete Binary tree (CBT) (if and only if)
A BT is known as CBT IFF all leaf nodes are at same level

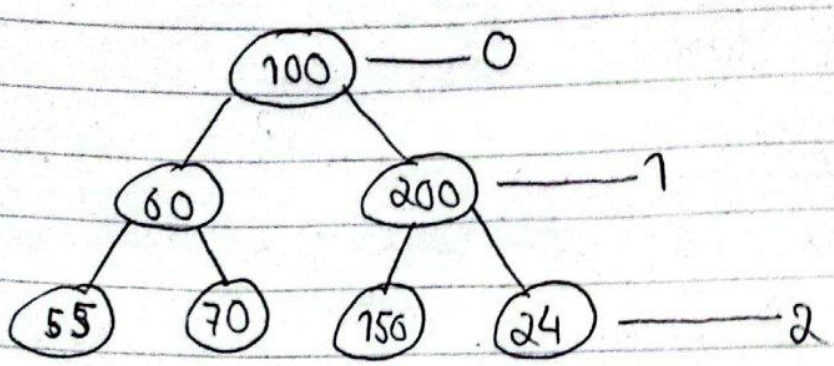


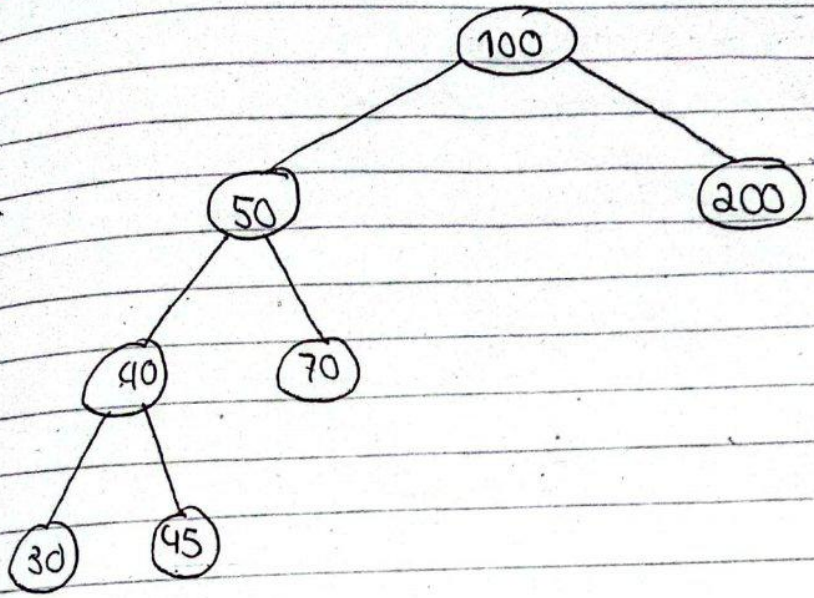
Fig: CBT

* Traversing in Binary Tree
Visiting all nodes present in a binary tree is called traversing where no data item should be left nor repeated.

1) Pre-order Traversing (VLR)
Root node is visited first then visit the left and right child respectively

2) Post-order Traversing (LRV)
Visit left child, right child then visit root node at last.

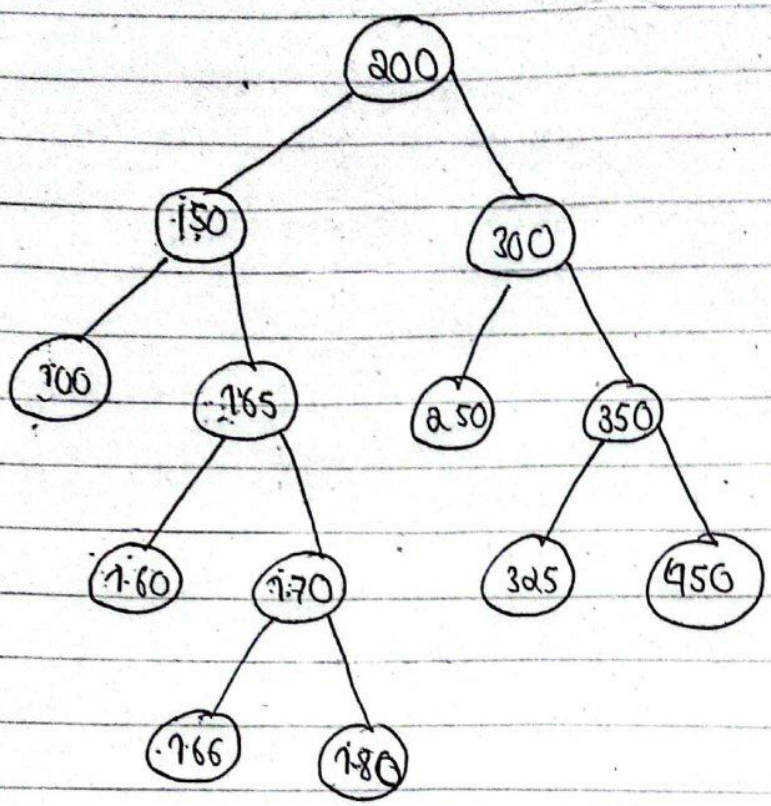
3) In-order Traversing (LVR)
Root node is visited in between left and right child.



Pre-order (V L R)
100, 50, 40, 30, 45, 70, 200

Post-order (L R V)
~~50, 40, 30, 45, 70, 200, 100~~ 30, 45, 40, 70, 50, 200, 100

In-order (L V R)
30, 40, 45, 50, 70, 100, 200

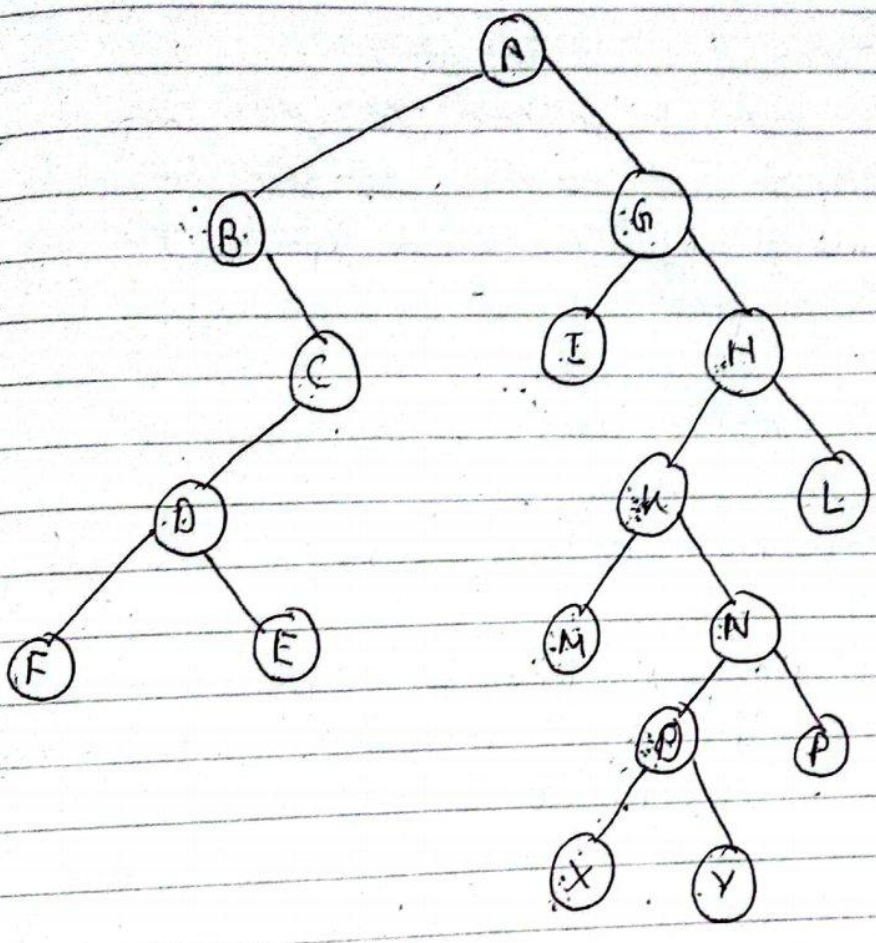


Pre-order (V-L-R): 200, 150, 100, 165, 160, 170, 166, 180, 300, 250, 350, 325, 450

~~In-order (L-V-R): 100, 150, 160, 165, 166, 170, 180, 200, 300, 250, 350, 325, 450~~

Post-order (L-R-V): ~~100, 160, 166, 166, 180, 170,~~
100, 160, 166, 180, 170, 165, 150, 250, 325, 450, 350, 300, 200

In order (L-V-R): 100, 150, 160, 165, 166, 170, 180, 200, 250, 300, 325, 350, 450



Pre order (V-L-R) = A, B, C, D, F, E, G, I, H, K, M, N, O, X, Y, P, L

~~Post-order (L-R-V) = B, F, E, D, C, I, M, X, Y, O, P, N, M, K, L, H, G, A~~

In-order (L-V-R) = B, F, D, E, C, A, I, G, M, K, X, O, Y, N, P, H, L

Post order (L-R-V) : F, E, D, C, B, I, M, X, Y, O, P, N, K, L, H, G, A

Binary Expression Tree

Expression tree are used to represent arithmetic expression specially the combination of operators, operands and order of evaluation. Binary expression tree is a special kind of binary tree with following properties

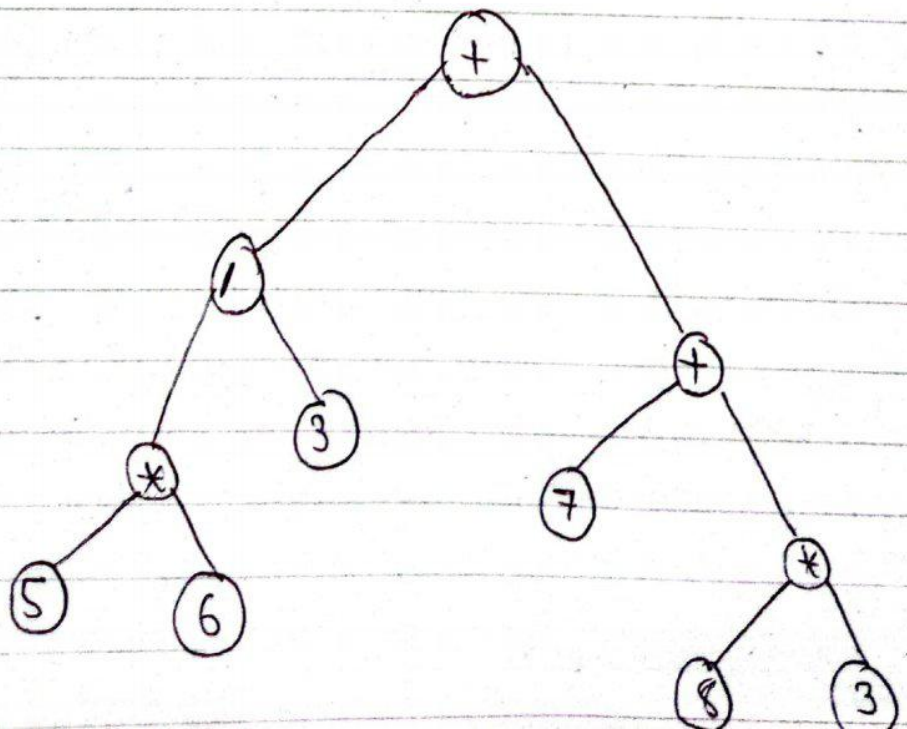
1) Each leap node contains single operand.

2) Each non-leaf node contains a single binary operator

3) The left and right sub trees of an operator node represents sub-expression that must be evaluated before appointing the operator at the root of sub-tree.

Example

(5 * 6 / 3 + (7 + 8 * 3))



Pre-order (V-L-R) = + / * 5 6 3 + 7 * 8 3

Post-order (L-R-V) = 5 6 * 3 / 7 8 3 * + +

Binary Search Tree (BST)

BST is a type of Binary tree with a special organization of data BST is either empty or contains a data value that satisfies following properties

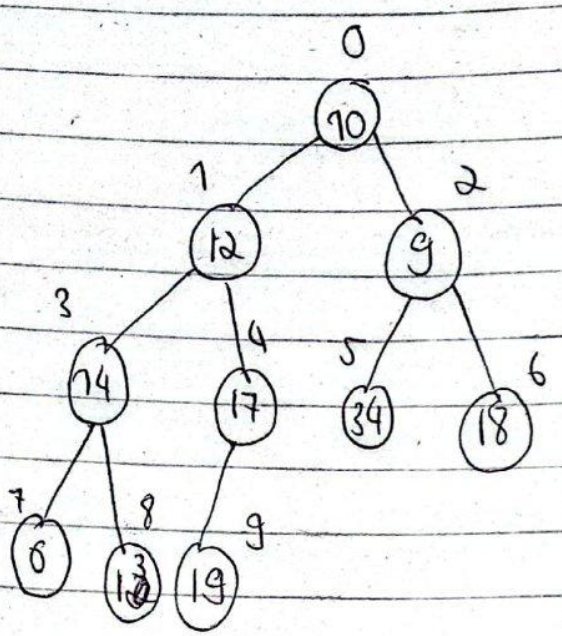
1) Every node has a unique keys

2) All the data value in the left sub tree are smaller than its root

3) All the data value in the right subtree are greater that its root

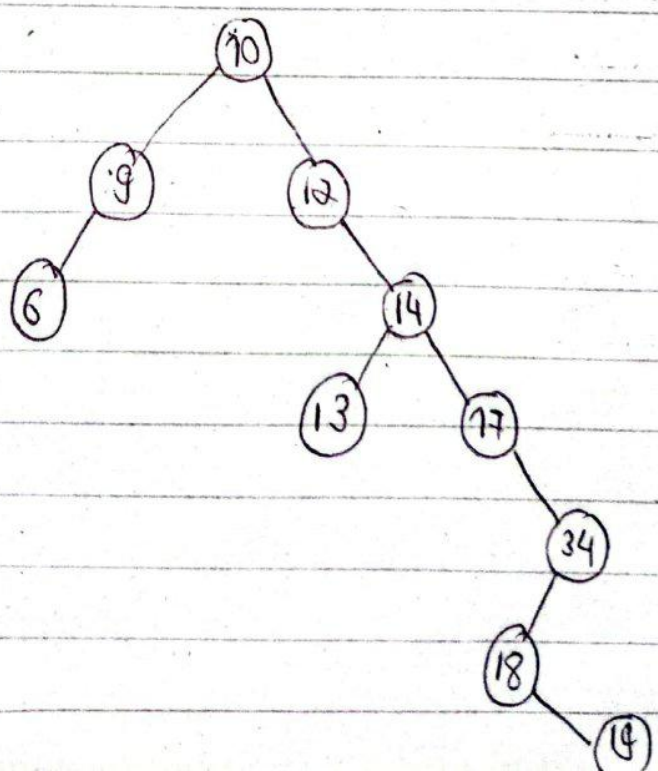
4) The left and right sub-trees are also binary search tree.

Draw a binary tree and binary search tree
10, 12, 9, 14, 17, 34, 18, 6, 13, 19

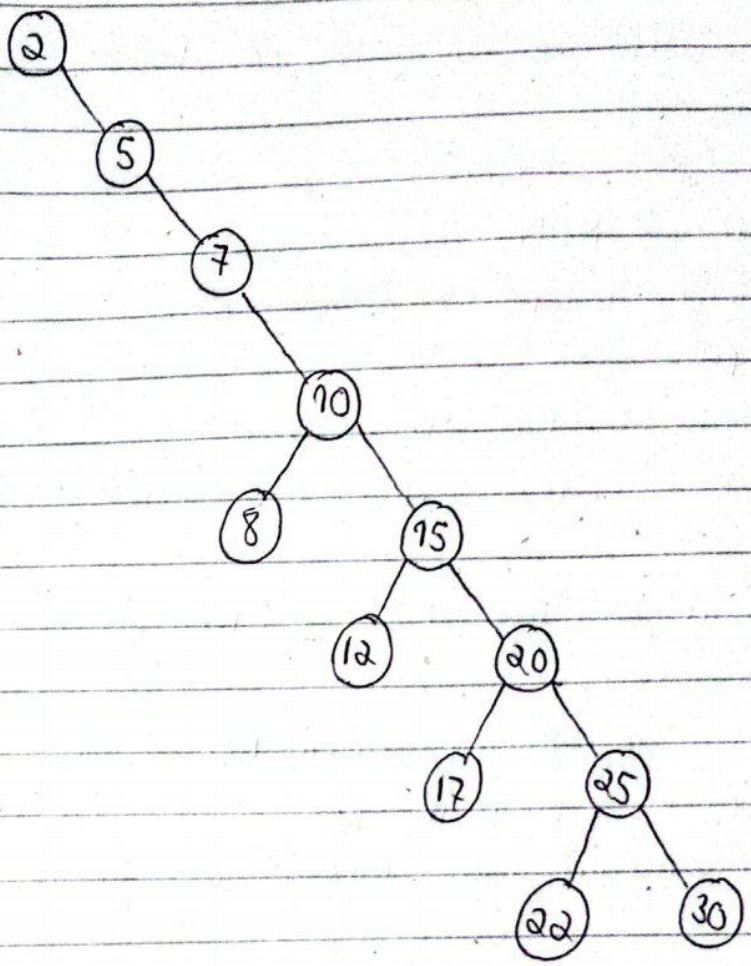


0	1	2	3	4	5	6	7	8	9
10	12	9	14	17	34	18	6	13	19

Fig: Array Representation of Binary Tree



BST: 2, 5, 7, 10, 8, 15, 20, 25, 12, 30, 17, 22



BST_SEARCH (Root, item)

Input: Root is the starting node and item is the key to be searched

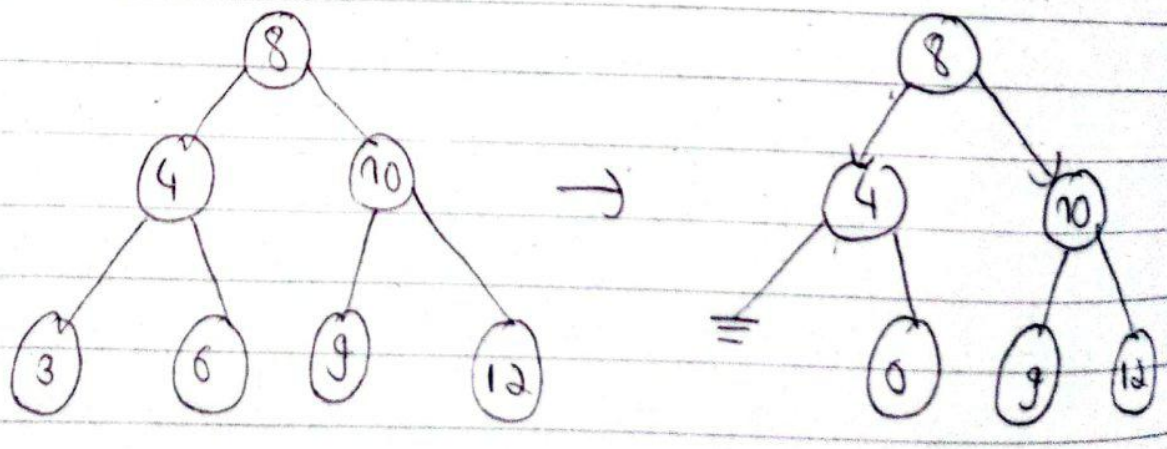
Output: item is found or not found.

```
1) If (root == NULL)
2) return NULL
3) else if (root -> data == item)
4) return item
5) else if (root -> data > item)
6) return BST_SEARCH (root -> left, item)
7) else
8) return BST_SEARCH (root -> right, item)
```

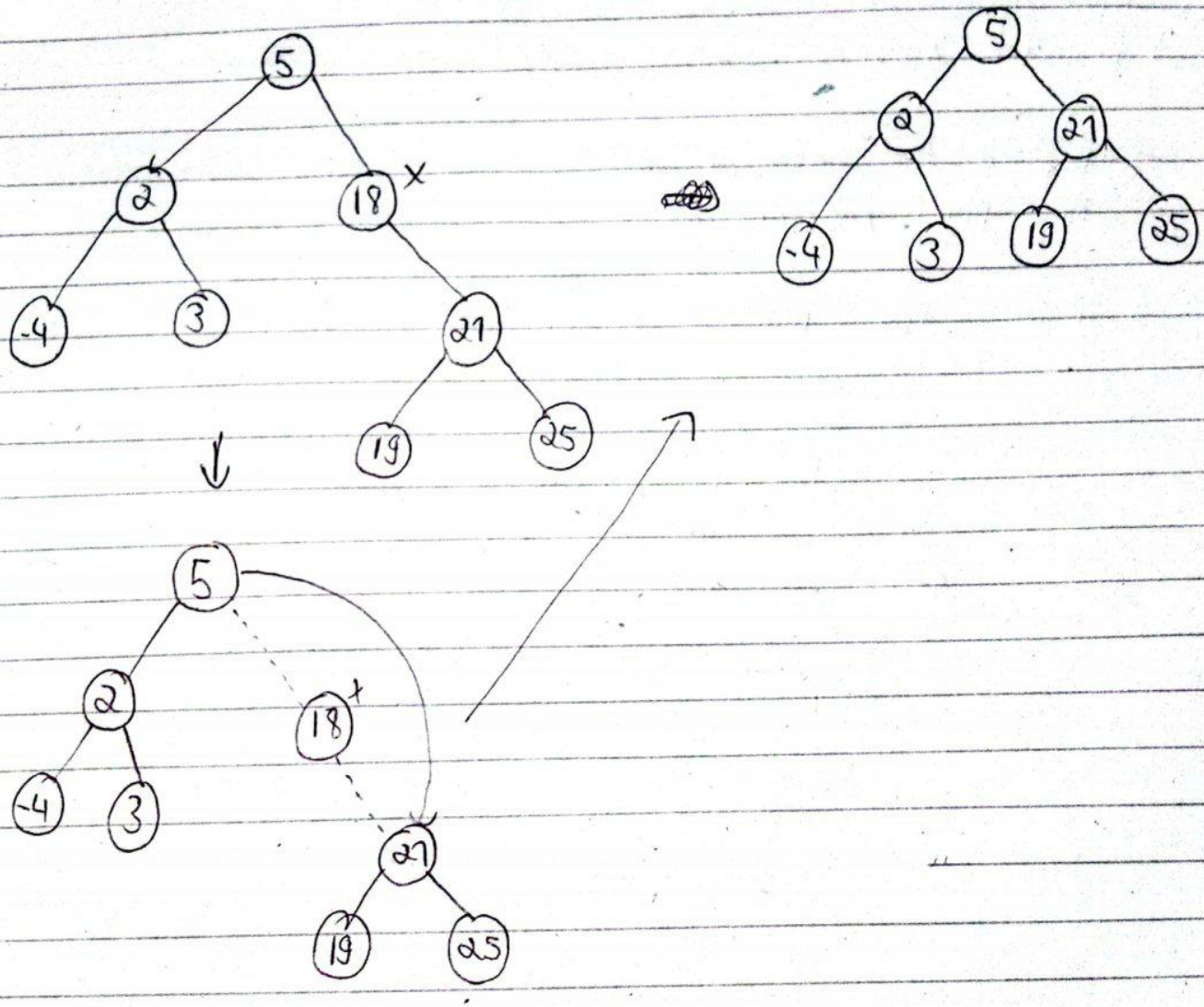
Deletion in Binary Search Tree

1) Deletion of leaf node

Set corresponding link of the parent node to NULL



2) Node to be removed has one child
Cut the node to be removed from the tree, and than redirect the pointer from the parent of the deleted node to its sub trees.

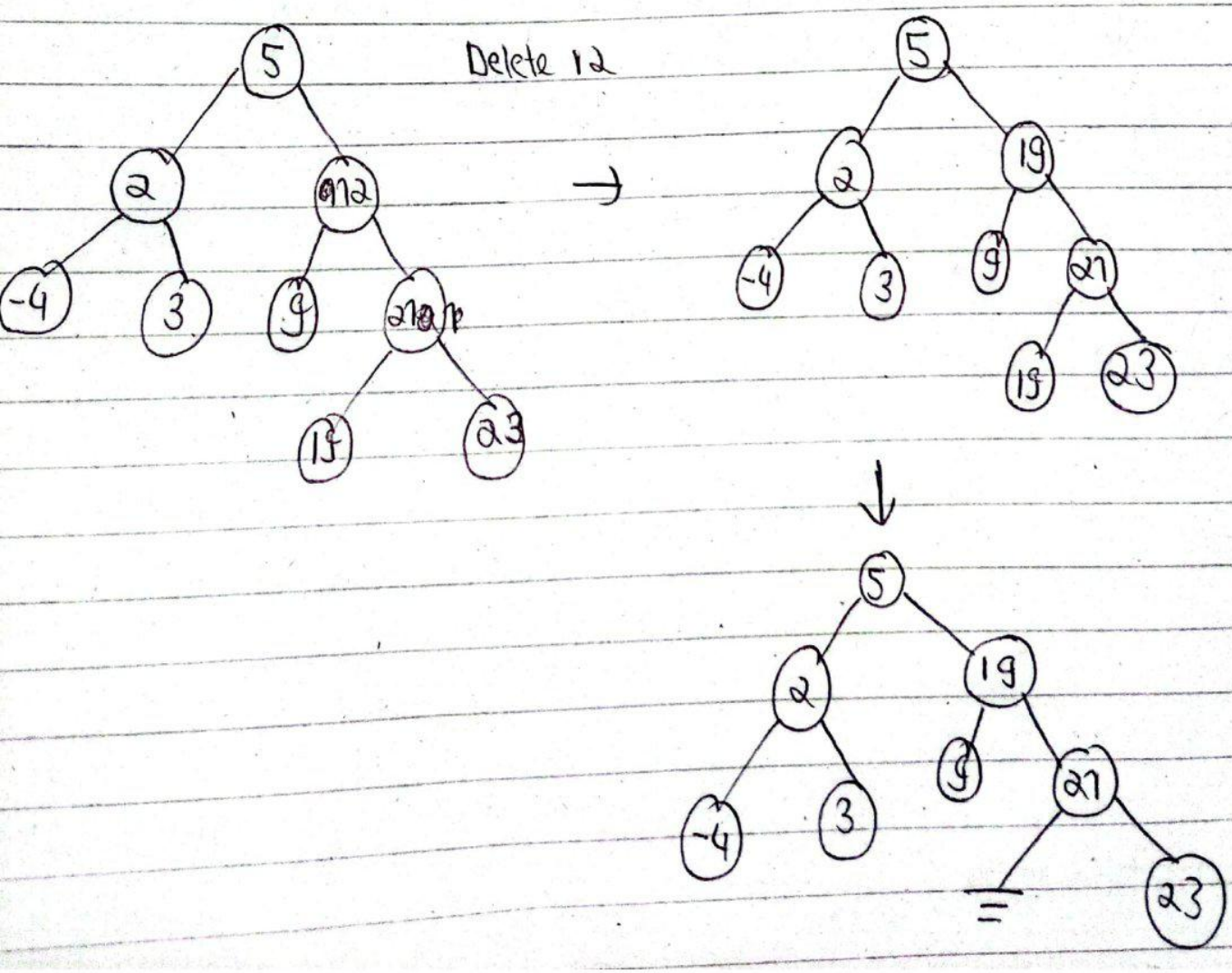


3) Node to be deleted has two children
Replace the node to be deleted with smallest key of the right sub tree.

· Basic approach
→ find the smallest value in its right subtree

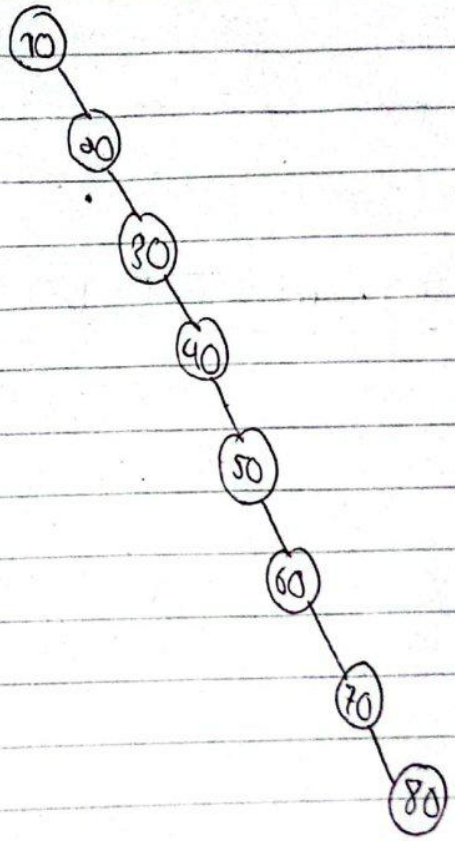
→ Replace the value of node to be removed with minimum. Now right subtree contains duplicate key

→ Delete the minimum element which is leaf node.



Problem in a BST : Degenerate tree (Unbalanced)

10, 20, 30, 40, 50, 60, 70, 80

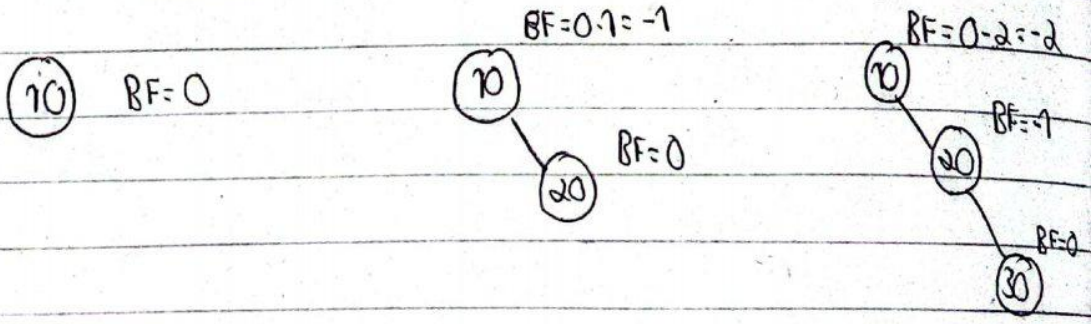


AVL Tree

AVL tree is a height balanced binary search tree. AVL tree was introduced by Adelson-Velsky and Landis : In an AVL tree every node maintains height, the height of the left sub-tree and right sub tree can differ by at most one i.e 0, -1, 1

Balance factor (BF) = height of left sub tree - height of right sub tree

10, 20, 30, 12



Balance binary tree

The disadvantage of BST is that its height can be as large as N-1 where N is number of nodes. This means that the time need to perform insertion, deletion and other operation can be (O(N) in worst case.

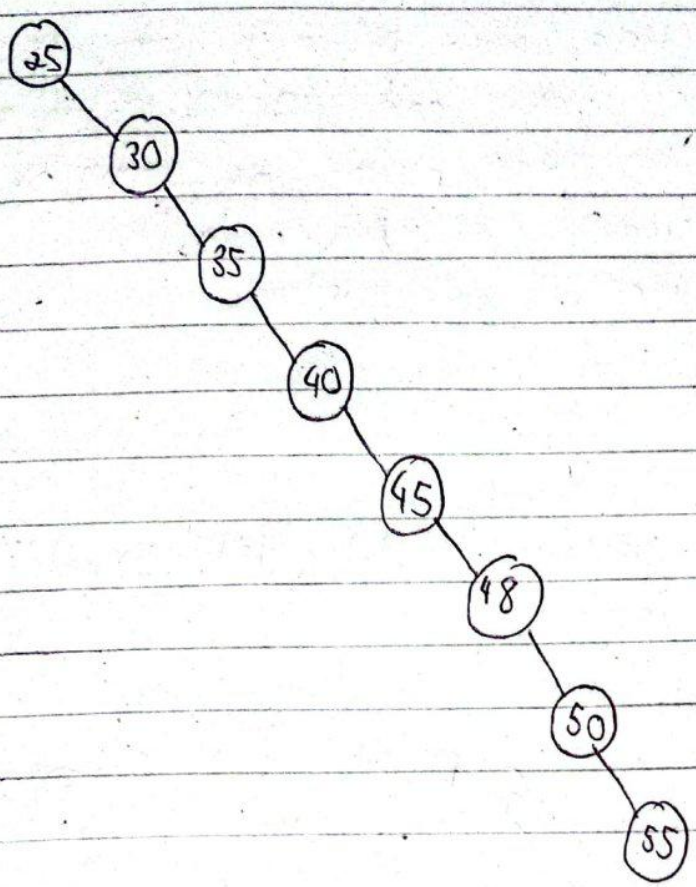


fig: Height of the tree is 7 for 8 nodes

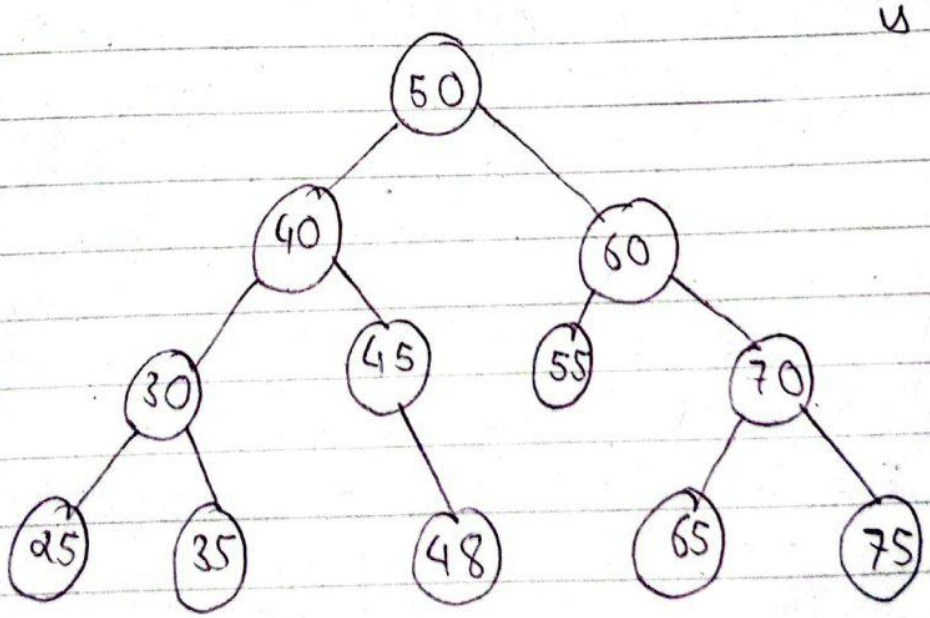


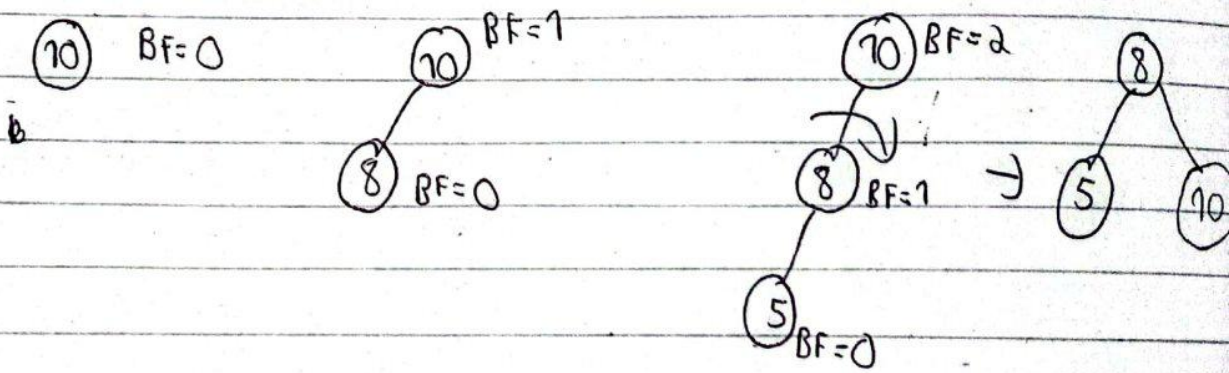
fig: Height of tree is 3 for 12 nodes

A binary search tree with N nodes has at least O ($\log_2 N$) in the best case and (OCN) in the worst case Thus, our objective is to make height as small as possible i.e O ($\log_2 N$) to increase the search efficiency and other operations such as insertions and deletion.

Balancing BST to create AVL

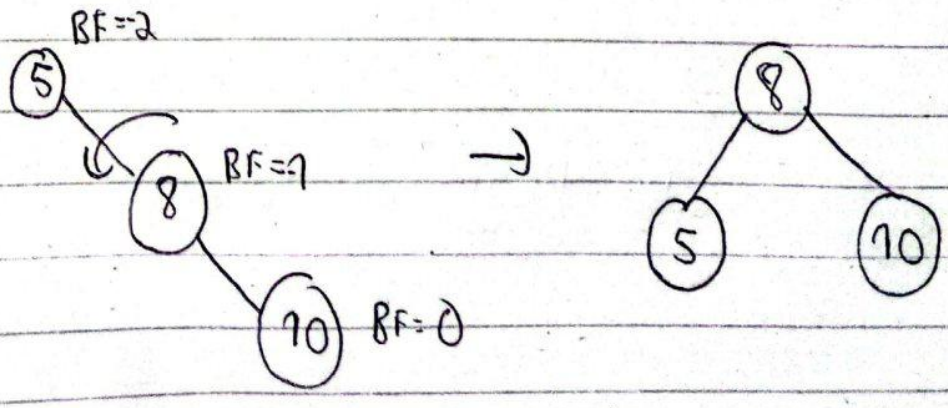
Case 1: If a node is inserted at the left subtree of left child of root node

Eg 10, 8, 5



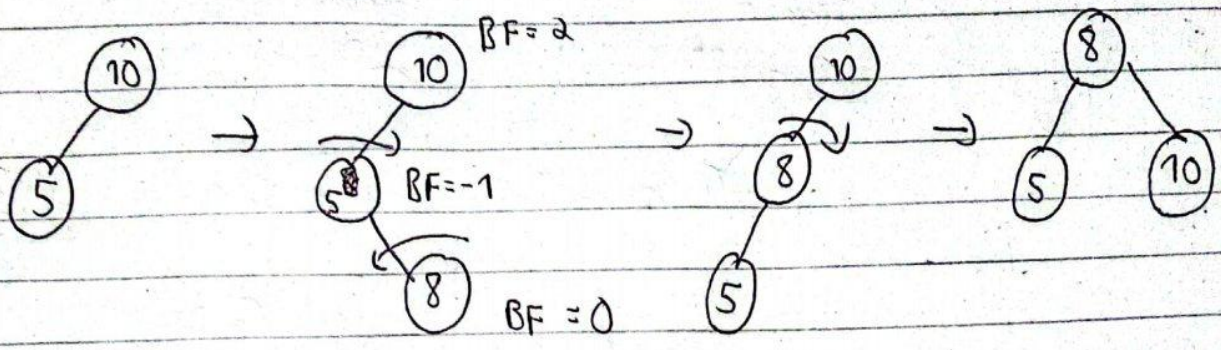
Case 2: If a node is inserted at the right subtree of right child of root node

Eg 5, 8, 10



Case III : If a node is inserted at the right sub tree of left child of root node

Eg: 10, 5, 8



Dog leg pattern

Case IV : If a node is inserted at the left sub tree of right child of root node

Eg: 5, 10, 8