Ganga Kalesboard. G
Roll : No : 19
MCA [1st semester]

Advanced Data Structures

1) Data structures are classified into 2 types. Primitive and non-primitive data structures. The linear and non-linear data structures are classifications of non-primitive data structures.

Linear data structure

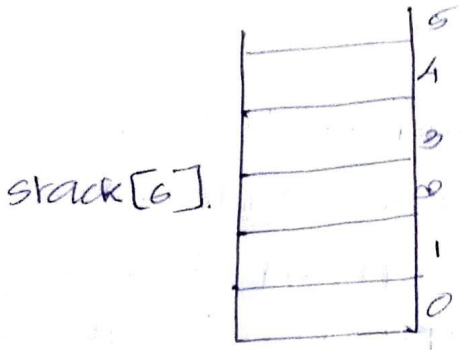
The data structure in which elements are arranged in a sequential order. These data-structures are easy to implement because computer memory is arranged in a linear way. It involves a single level.
Eg:- array, stack, queue, linkedlist.

Non-linear data structure

The data structure in which the elements are arranged in a non-sequential order is the non-linear data structure. These data-structures utilise the memory more efficiently than linear data structures. It involves different levels
Eg:- tree, graph

e) Stack is a linear data structure. It works in a LIFO fashion. The elements are deleted & inserted into and deleted from the same end. The process of inserting elements into the stack is known as push operation & the process of deleting elements from the stack is known as pop operation. 'top' is a pointer that points to the last inserted / top most element of the stack.

Stack implementation using array



There are mainly 4 operations in a stack.

(i) top()

(ii) push()

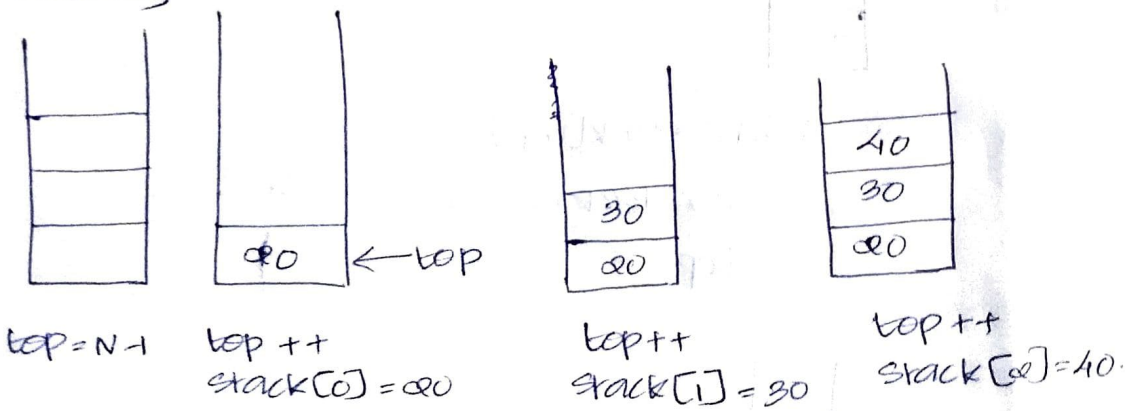
(iii) pop()

(iv) display().

→
```
void push()
{
int x;
printf("enter data");
scanf("%d", &x)
```

```
if(top==N-1)
{
Printf("stack overflow");
}
else
{
top++;
stack[top]=x;
}}
```

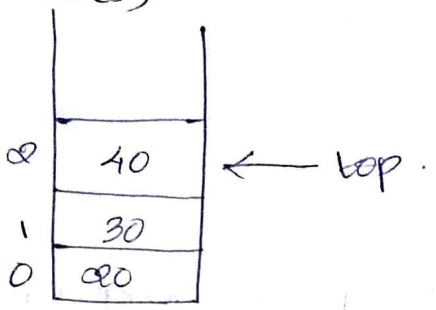
If we want to insert an element into a stack first check, the stack is empty or not. If top = N-1, which means stack is empty. Otherwise, the top is incremented to the next position & the element is inserted.



→
```
void pop()
{
for i=0;
if(top==-1)
{
printf("underflow");
}
else
{
```

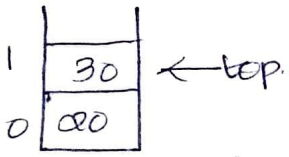
```
Prem = stack[top];
top--;
Printf("%d", Prem)
}
```

stack[3]



The element to be delered is 40.

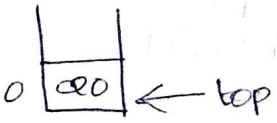
If delere stack[2]. Then top is decremented.



delere stack[top]

ie, stack[1] = 30

top--



delere stack[top]

ie, stack[0] = 20

top--

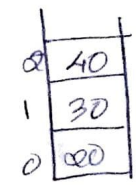
Stack is empty

→ void top()

```
{
if(top==-1)
{
printf("stack empty");
}
else
{
printf("%d", stack[top]);
}
```

Inorder to get the top most item first print stack[top]. Because top points to the topmost element

Eg:-



Stack[3]

∴ stack[top] = 40

→ void display()

```
{
int i;
for(i=top; i>=0; i--)
{
printf("%d", stack[i]);
}}
```

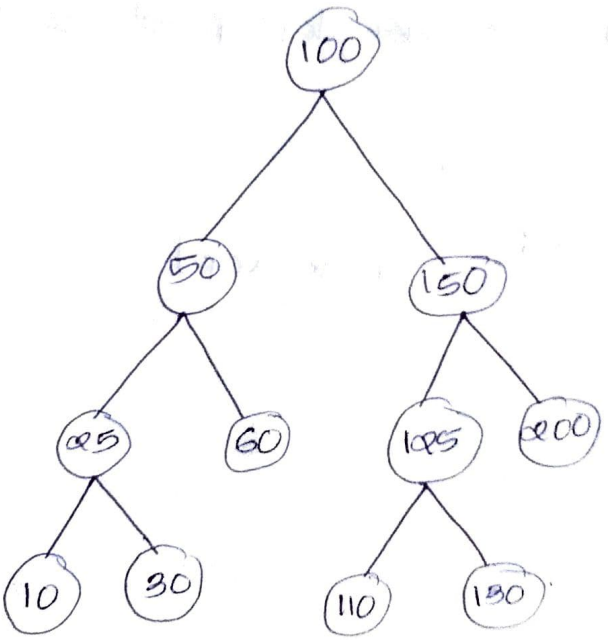
Inorder to display the elements of a stack, the value of i start from to 0. The last inserted element is displayed first and

the least inserted element is displayed last

Eg:- from above Eg:-

40
30
20.

3) The binary search tree is a binary tree and are specially designed for the purpose of searching.



If N is the number of nodes, then an element in the BST can be searched in an average of ~~order of~~ O(log N) time.

Properties

A binary search tree can be empty and if it is not empty. It must satisfy the following properties.

(i) All keys in the left subtree of the root are less than the key in the root.

(ii) All the keys in the right subtree of the root are always greater than the key in the root.

(iii) Left and right subtree of root are ~~greater than the key in the root~~ always binary search trees

In the above figure, the key in the root is 100, All the keys in the left subtrees such as 50, 25, 60, 10, 30 are less than 100 and all the keys in the right subtrees such as 150, 125, 200, 110 and 130 are greater than 100 and Left subtree and right subtree are always BST. So it satisfies all the properties of binary search tree.

5) A disjoint set data structure is a collection of disjoint sets and is represented as $S = \{S_1, S_2, \ldots S_k\}$ where $S_1, S_2, \ldots S_k$ are disjoint dynamic sets.

~~A disjoint~~ Disjoint sets are sets in

which no two sets have any common elements.

The operations in a disjoint set are

(i) MAKE-SET(x) — This operation creates a new set whose only element is pointed to by x.

Eg:- MAKE-SET(x) → from 0 to 7

{0} {1} {2} {3} {4} {5} {6} {7}.

(ii) UNION(x,y) → This operation unites or merges dynamic sets that contain x and y into a new set. The union of Sx and Sy is denoted by Sx ∪ Sy.

Eg:- Sx = {1,2,3} From above example,

Sy = {4,6,3} UNION(1,2)

Sx {0} {1,2} {3} {4} {5} {6} {7}

UNION(5,6)

{0} {1,2} {3} {4} {5,6} {7}.

(iii) FindSet(x) → returns a pointer to the unique set containing x.

Eg:- Sx = {1,2,3} Sy = {6,4,5}.

find(1) → belongs to Sx

find(5) → belongs to Sy.

→ Eg: Make-set(x) .. from 1 to 5

{1} {2} {3} {4} {5}

union(2, 3)

{1} {2, 3} {4} {5}.

Here 2 & 3 are connected because both are in the same set.

ie, find(2) == find(3).

4) We saved some amount over a period of time and after a long time, we can buy some expensive things using these saved rupees. These saved rupees are called credit. This is also done in ~~amortise~~ accounting method.

Stack

Operation	Push	Pop	Multipop
Actual cost	1	1	min(n, k)
Amortized cost	2	0	0

The credit of push operation is

Amortized cost − Actual cost

= 2 − 1 = 1.

pop and multipop operations cannot be done

worst push operation.

So the amortized cost of push operation is 2.

The amortized cost of pop operation is 0

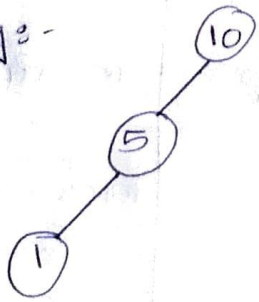
The amortized cost of multipop operation is 0

c) The various rotations in height balanced tree are

→ Left - Left rotation .
→ Right - Right rotation
→ Left - right rotation
→ Right - left rotation.

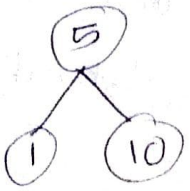
→ Left - Left rotation, perform right rotation.

Eg:-



(2-0=2) // unbalanced.

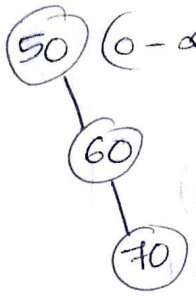
This is a left - skewed tree. This is an unbalanced tree. To make it balanced rotate right.



balanced.

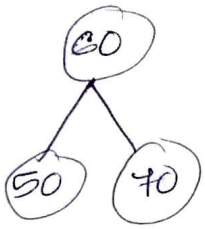
5 becomes the root & previous root is rotated right & it becomes the right child.

(ii) Right - right rotation, left rotation



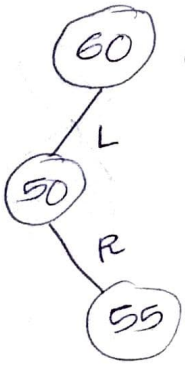
(0 - 2) = -2 // unbalanced.

To make it balanced perform left rotation, the root 50 is rotated left & it becomes the left child of 60



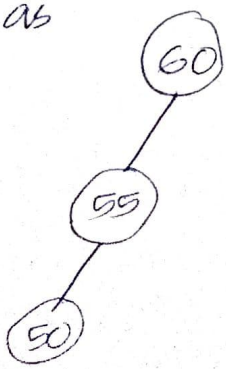
// balanced.

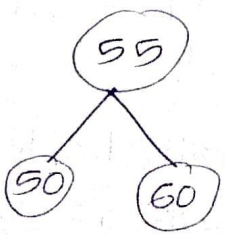
(iii) Left - right rotation



(2 - 0) = 2 // unbalanced.

To make it balanced, make the tree as





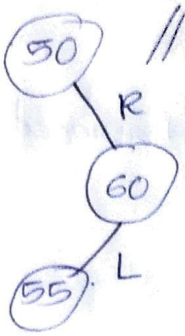
// balanced

Here 2 rotations are performed

R → Rotate to left

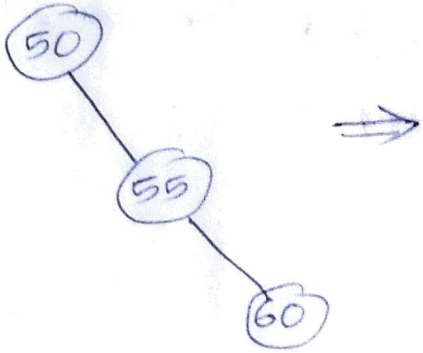
L → rotate to right

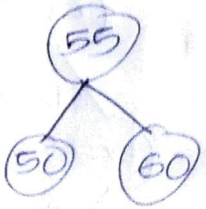
(iv) Right - left rotation



// 0 - 2 - -2 // unbalanced

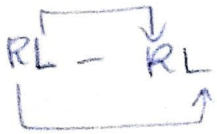
To make it balanced, make RL rotation





// balanced

Two rotations are performed here

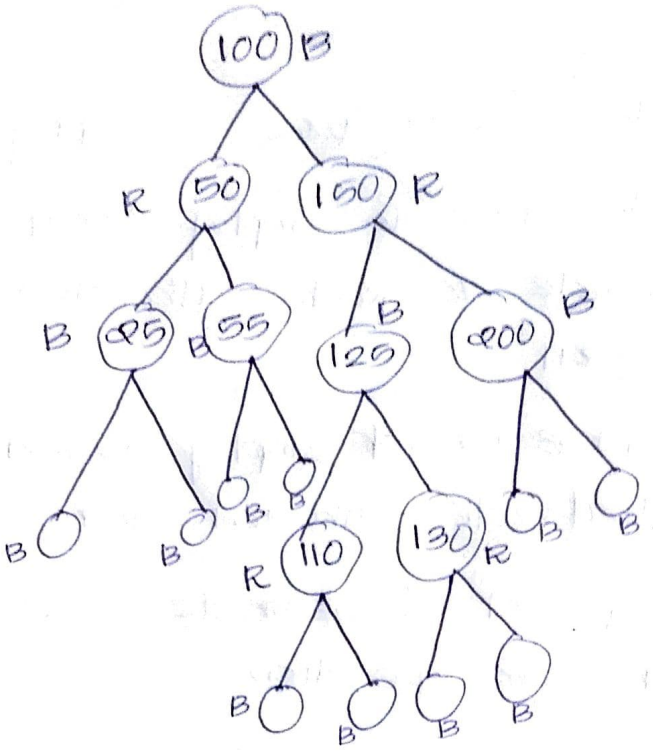


~~R → rotate to left~~

L → rotate to right

R → rotate left

7)



The properties of RB tree are.

(i) Red black tree must be a binary search tree

(ii) Root must be colored black.

(iii) The children of red node must be black.
(There should not be a consecutive red nodes)

(iv) In all path of the tree, there should be same number of black colored nodes.

(v) Every new node is inserted with red color.

(vi) Every leaf must be colored black.

10)

Step 1 : check whether tree is empty

Step 2 : If the tree is empty insert new node as root with color black and exit.

Step 3 : If tree is not empty insert new node as leaf node with red color.

Step 4 : If parent of new node is black then exit from operation.

Step 5 : If parent of new node is red, then check color of parent's sibling of new node.

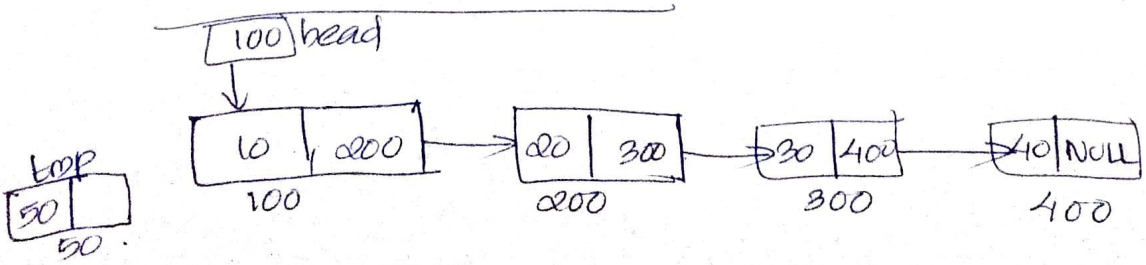
Step 6 : If it is colored black or null, then make suitable rotation and recolor it.

Step 7 : If it is colored red, then perform recolor (black) in root until tree becomes RB tree.

11) (a) The linked lists were introduced to overcome the drawback of arrays.

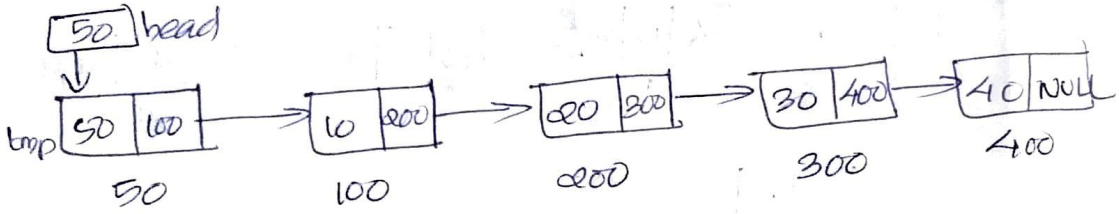
Singly linked list

Insertion at beginning

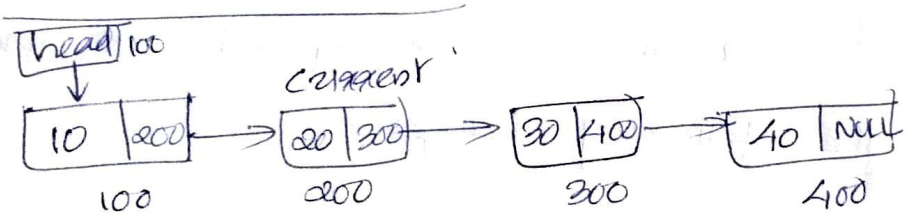


Here tmp is the pointer that points to the newnode to be inserted. To insert the newnode at the beginning, do the following steps

```
tmp -> data = x;
tmp -> link = head;
tmp -> head = tmp;
```

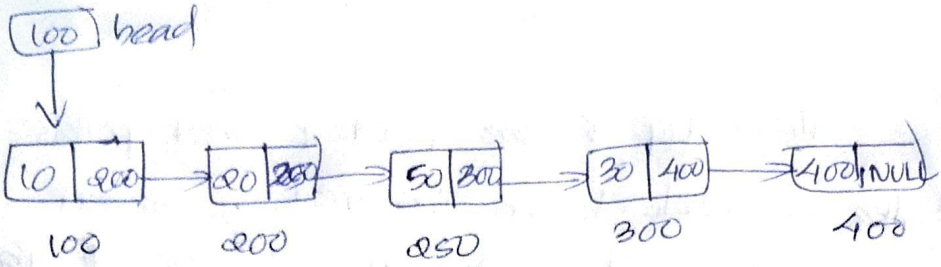


Insertion at middle



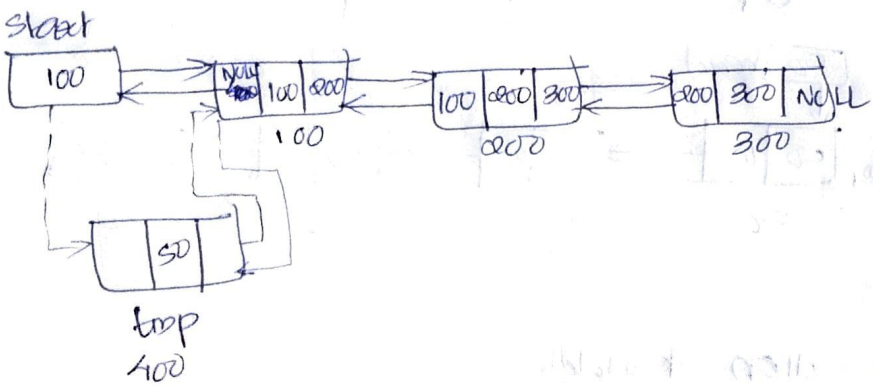
tmp is node to be inserted at middle. First we have to traverse to obtain the node after which the newnode is to be inserted. Then make the following changes.

```
tmp -> data = x;
tmp -> link = current -> link;
current -> link = tmp;
```



Doubly linked list

Inserting at beginning:

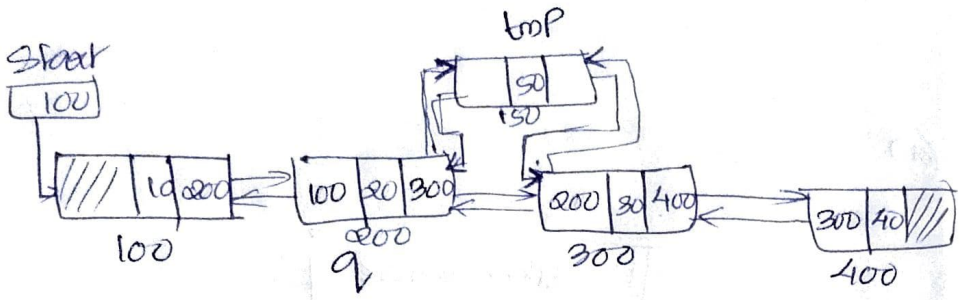


To insert the new node temp into the linked list make the following changes:

```
temp -> prev = NULL;
temp -> next = start;
start -> prev = temp;
start = temp
```

start points to the 1st node of the linked list.

Insertion at middle



To insert tmp into the LinkedList, make the following changes.

temp = 1st we have to traverse to reach the node after which tmp is to be inserted. Traversal starts from ~~start -> next.~~ ~~which points to~~ the 1st node.

~~tmp = start~~

~~start = start -> next.~~

~~start ->~~

tmp = q -> next;

q -> next = tmp -> next;

tmp -> next -> prevPoints = q

~~}~~

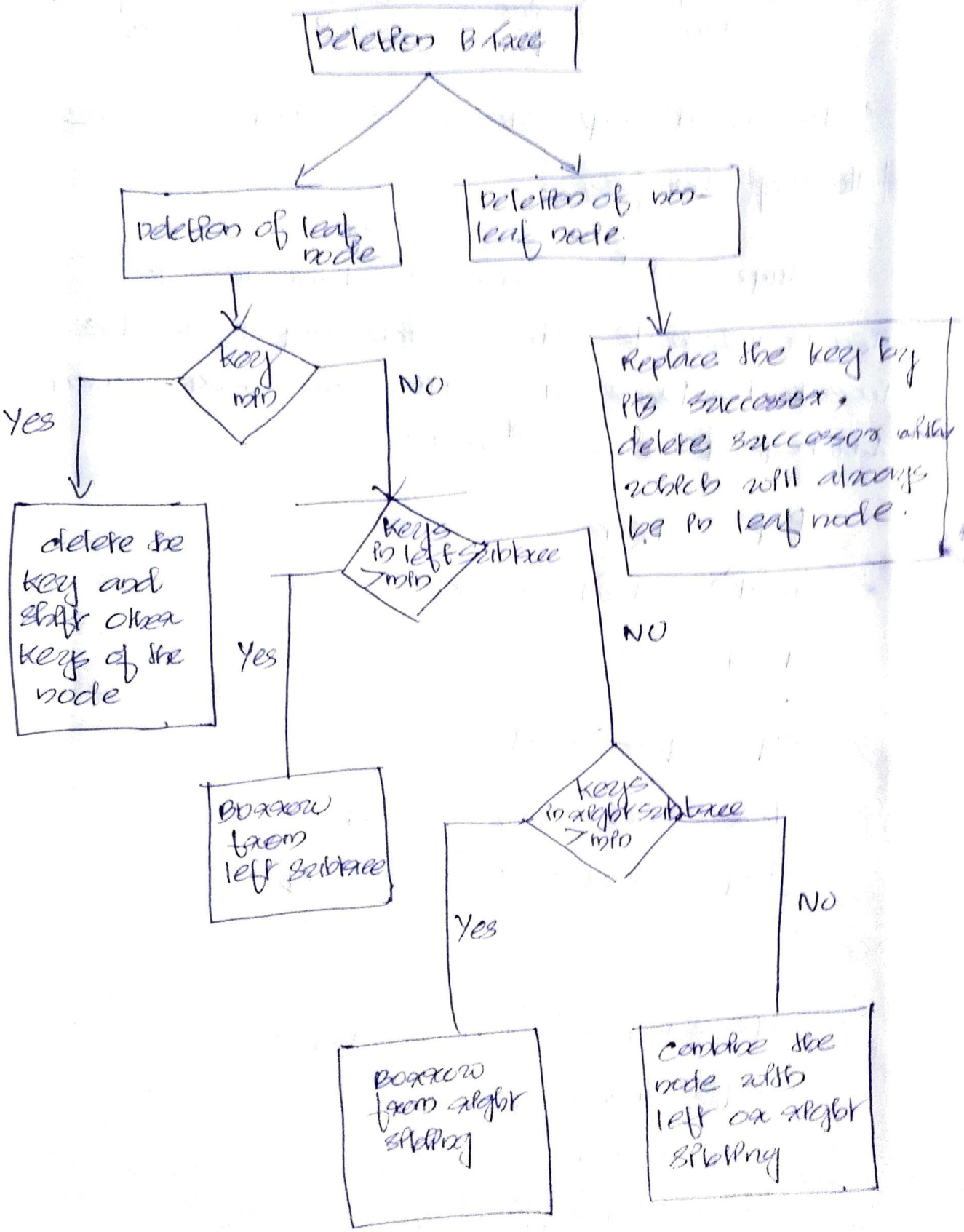
q -> link -> prevPoints = tmp

tmp -> link = q -> link

tmp -> prevPoints = q

q -> link = tmp

14) (a)


Deletion B tree
Deletion of leaf node
Deletion of non-leaf node
key min
Yes
NO
delete the key and shift other keys of the node
Keys in left subtree > min
Yes
NO
Borrow from left subtree
Keys in right subtree > min
Yes
No
Borrow from right sibling
combine the node with left or right sibling
Replace the key by its successor, delete successor which will always be in leaf node.

9) If a node is full, then the insertion of a new key will cause overflow. Therefore to overcome this situation splitting is done

First find the middle / median value of the node. Then create a new leaf node and copy into it all the keys that appear after the median. Move up the median at appropriate position in the parent of this node. Add an additional child pointer from the parent to the new node. Then Add the new key at the right location in the child nodes of the parent.

13) (a)

To delete a node in the RB tree, we perform normal BST deletion.

If the node to be deleted is red, just delete the node

But if the node to be deleted is black. there is an issue called double black.

(i) If root is DB, then remove root

(ii) If DB's sibling is black & both of its childrens are black

Process :-

1. Delete one black from DB & S (black to red)

2. Add black to the parent

(2.a) If parent is red it becomes black

(2.b) If parent is black, it becomes DB.

(2.c) If DB's sibling is black & one of its (DB) near child is red and the other is black

Processes

1. exchange the color 's' (sibling) & its red child, then 's' becomes red & its child becomes black

2. perform rotation w.r.t respect to 's' in opposite direction of DB.

Or

(2.d) If DB's sibling 's' is black and one of its far child is red or both its children are red

process

1. Swap the color of parent of DB & S

2. perform rotation w.r.t respect to parent of DB in direction of DB.

3. Remove one black from DB.

4. color the red child of s as black

(2.e) If DB's sibling is red

1. Swap the color of parent of DB & sibling

2. Rotate w.r.t respect to parent in direction of DB.

3. reapply the case (2.e).

12(a)

The process of converting keys into addresses is known as hashing. The hash function generates address from a given key. The hash function takes a key as input and returns the hash value of that key and this hash value is used as the address for storing the key. An array in which the insertion and searching is performed through hashing is the hash table.

key → Hash function → Address

There are different methods for generating the hash value

→ Truncation method
→ Folding method
→ division method
→ midsquare method.

Truncation method

A part of the key is taken as the address ie, takes the rightmost or leftmost digits from the key

For eg:- keys are 1251359, 1351390, 1539950

The addresses are 59, 90 and 50. This is an easy method to compute addresses. But the addresses generated through this method is same for many different keys.

Folding method

In this method, the keys are broken into parts. Then these parts are shifted and added.

Eg:- The key is 139857359438

```
139  857  359  438
1 3
139
857
359
438
-----
1793
```

Here we ignore the carry 1 and 793 is taken as the address for this key

Another type of folding is boundary folding. Here, even parts are reversed before addition.

```
1 3
139
758 (reversed)
359
834 (reversed)
-----
2090
```

If we take the carry 2 & the address is 090

Midsquare method

Here we find the square of the given key and some digits from the middle of the square is taken as the address of the key.

Eg: 1527 → key

Square → 2331729

Address → 317

```
    2 1 1 1 3
      3 1 4
      1527
      1527
  1 1 1
  1 0 6 8 9
    1 3 0 5 4
  1 7 6 3 5
  1 5 2 7
-----------
  2 3 3 1 7 2 9
```

Division method

Here, the given key is divided by the table size & the remainder obtained is taken as the address

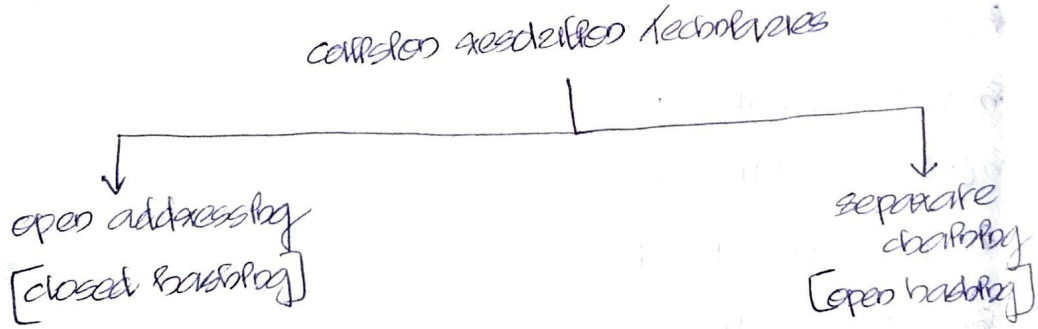
$H(k) = k \bmod m$

Eg:- table size = 11

key → 100

Address = 100 % 11 = 1

If the hash function generates same address for different keys, a situation called collision To resolve collision, there are several collision resolution techniques



Here, a given key is compared with many different keys & each comparison is known as a Probe

(i) open addressing

Here, if a key causes collision, that key is placed in a location other than its hash address in the hash table

3 methods

→ linear probing
→ quadratic probing
→ double hashing

→ Linear probing

If the address generated for a key is 'm' and that location is not empty then that key is inserted in the next location ie, 'm+1'. If that location is also not empty, then insert that key in 'm+2' location & so on.

Eg: keys → 10, 16, 13, 42, 38, 28, table size = 11

h(key) = key % 11 // division method

h(10) = 10 % 11 = 10

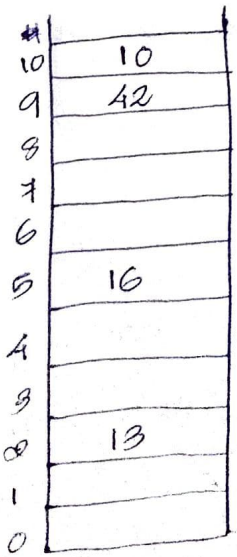
h(16) = 16 % 11 = 5

h(13) = 13 % 11 = 2

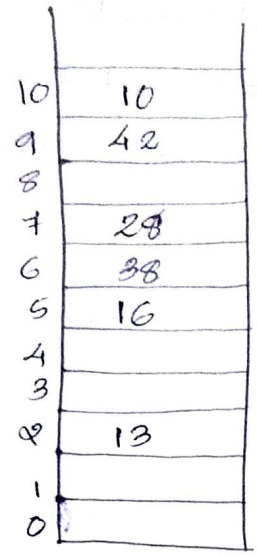
h(42) = 42 % 11 = 9

h(38) = 38 % 11 = 5

h(28) = 28 % 11 = 6



Here key 10, 42, 16 & 13 are inserted. The next key to be inserted is 38. But ~~that location~~ access it causes collision. Location 5 is not empty. So next location is 5+1=6. 6 is empty so 38 inserted there.



The next key to be inserted is 28. But the location of this key is not empty. So it is inserted in next location. ie, 6+1 = 7

→ Quadratic probing

In quadratic probing, the colliding keys are placed away from the initial colliding point

$H(k,i) = h(k) + i^2 \mod \text{TSize}$

Eg:- 56, 49, 32, 40, 16, 39 → keys

Table size = 11

h(56) = 56 % 11 = 1

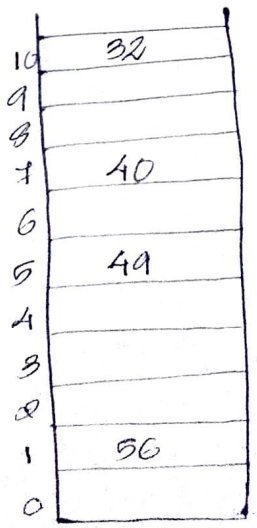
h(49) = 49 % 11 = 5

h(32) = 32 % 11 = 10

h(40) = 40 % 11 = 7

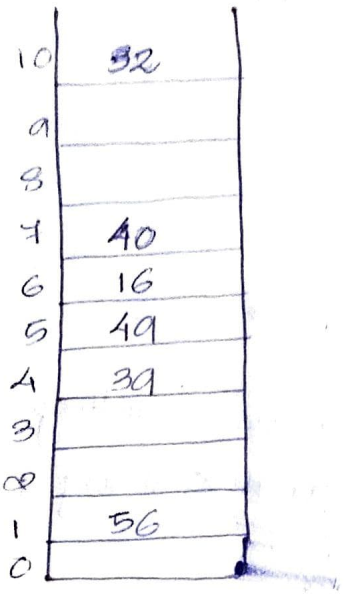
h(16) = 16 % 11 = 5

h(39) = 39 % 11 = 6



keys 32, 40, 49 & 56 are inserted. Now, key 16 is to be inserted. But location 5 is not empty. So next location is $(5+1^2)\%11 = 6$. 6 is empty. So 16 is inserted there. Next 39 is to be inserted. But location 6 is not empty. So next location is $(6+1^2)\%11 = 7$. 7 is also not empty. So next location

is $(6+2^0) \% 11 = 10 \% 11 = 10$ and location 10 is also not empty. So next location is $(6+3^0) \% 11$ $(6+9) \% 11 = 15 \% 11 = 4$. Location 4 is empty so 39 is inserted there.



→ Double hashing

In double hashing, increment factor is not constant. It is another hash function. So named double hashing.

$H(k,i) = (h(k) + ih'(k)) \bmod Tsize$

Eg:- 56 49 32 39 12 34

$h(56) = 56 \% 11 = 1$

$h(49) = 49 \% 11 = 5$

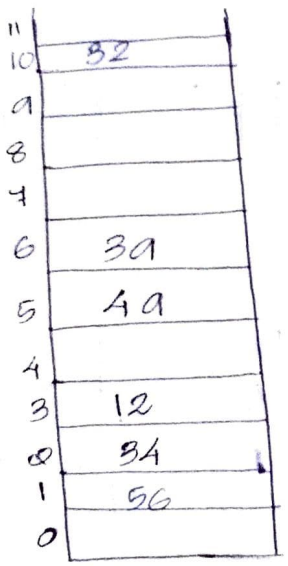
$h(32) = 32 \% 11 = 10$

$h(39) = 39 \% 11 = 6$

$h(12) = 12 \% 11 = 1$

$h(34) = 34 \% 11 = 1$

We can insert 56, 49, 32, 39 without any collision But when we insert 12 causes collision
Because location 1 is not empty
So next location is $1 + 7 - (12\%7) = 1+7-5 = 1+2 = 3$
Location 3 is empty so 12 is inserted there
insertion of 34 causes collision ~~so next location is 3 which is also not empty~~ So next location is $1 + 7 - (34\%7) = 1+7-6 = 1+1 = 2$ · 2 is ~~not~~ empty ~~so next location is 1+1=~~ 34 is inserted there.



Separate chaining

In this method, a linked list is maintained for elements with same hash address These linked lists are referred to as chains & hence this method is known as separate chaining.

Eg:- 1235, 2148, 3249, 1368, 2348, 1945, 4263 → keys

Table size = 5

h(1235) = 1235 % 5 = 0

h(2148) = 2148 % 5 = 3

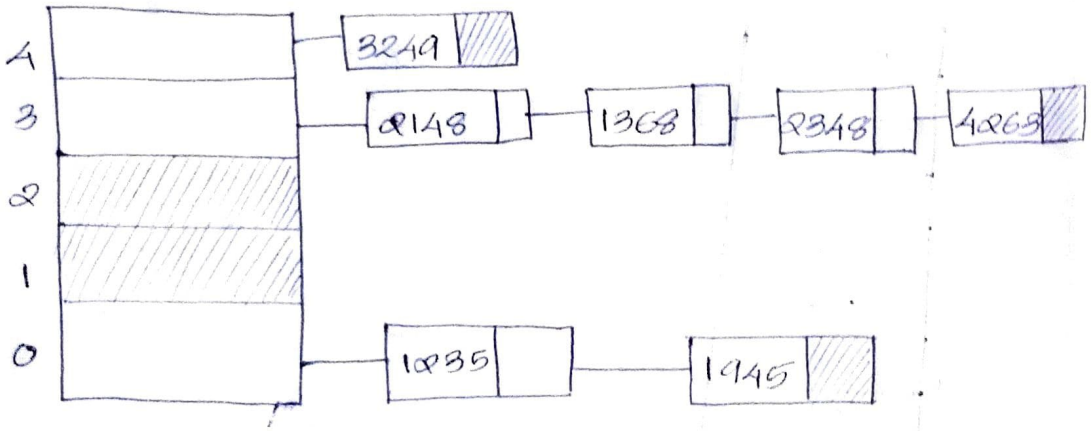
h(3249) = 3249 % 5 = 4

h(1368) = 1368 % 5 = 3

h(2348) = 2348 % 5 = 3

h(1945) = 1945 % 5 = 0

h(4263) = 4263 % 5 = 3



In separate chaining, load factor (α) > 1

Bucket hashing

In this method, the hash table is made of buckets. Each bucket holds a number of records. This method doesn't resolve the collision completely.

Eg:- 1235, 2145, 3249, 1368, 2348, 1945, 4263

table size = 5

$h(1035) = 1035 \% 5 = 0$

$h(0148) = 0148 \% 5 = 3$

$h(3049) = 3049 \% 5 = 4$

$h(1368) = 1368 \% 5 = 3$

$h(0348) = 0348 \% 05 = 3$

$h(1945) = 1945 \% 5 = 0$

$h(4063) = 4063 \% 5 = 3$