~~struct LinkedList {~~
~~int data~~
~~struct~~

- Linear: Array, stack, queue, linked list
- Non linear: Tree, graphs

☆ STACK:

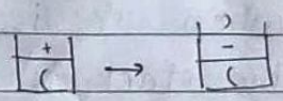
- #define stacksize 100
  struct stack {
  int top;
  int items [stacksize]; };
- infix a+b (in between)
  postfix ab+ prefix +ab
- Infix to postfix:

a b+c - d*e f+-

+
(
* -
* (
* -
)

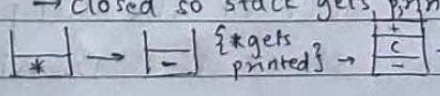
(1) Read expression from left to right
(2) If an operand is read, copy it to the output.
(3) If operator is '(', push it into the stack.
(4) If operator is ')', pop the stack until opening bracket is found '('. When that occurs, both parenthesis are discarded.
(5) If an operator is read and has a higher precedence */-+ than the operator at the top of the stack (TOS), the operator being read is pushed on the stack. If precedence is lower or equal then TOS is popped and copied to the output
(6) When reached end of expression, remaining operators at the end of stack are copied.

Eg:
infixVec (a+b-c)* d - (e+f)



{+ gets printed}

postfixVec ab+c - d*e f + - → closed so stack gets printed →



{* gets printed} → + -

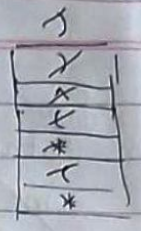
ABC*DEF^/G*-H*+

Eg:

A + (B*C - (D/E ^ F)* G)* H

~~ABC*DE/F^-G*+H*~~

ABC*DEF^/G*-H*+



~~Infix to pre~~

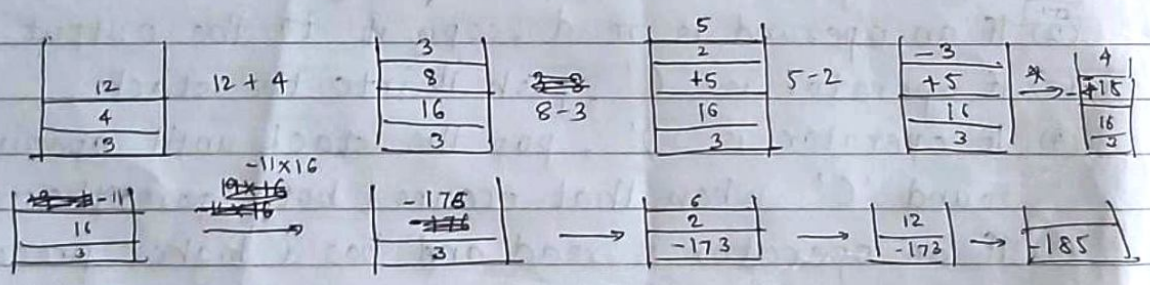
- Evaluation of postfix expression:

(1) Read the expression ~~fro~~ problem from L to R and ~~build~~ read the stack of operands. build

(2) When an operator is read, two operands are popped out of the stack. They are evaluated with operator and result is pushed into stack.

(3) At the end of the expression there must be only one operand left in stack, otherwise error.

Eg: 3 4 6 2 * + 8 3 - 2 5 - * 4 + * + 2 6 * -



=-185

- Infix to prefix:

(1) Read the expression from right to left.

(2) If it is operand, copy it to the left of the output.

(3) If ')' is read, push it into the stack.

(4) when '(' is encountered, the operator at the top of the stack is popped off the stack and copied to the left of the output until the symbol at the top of the stack is ')'. When that occurs, both parenthesis are

++-+KL*MN*//*^OPWUVTQ

discarded.

(5) If an operator is scanned and has a higher or equal precedence than the operator on TOS, the operator being scanned is pushed onto the stack. If the precedence of the operator being scanned is lower than precedence of operator on TOS, the operator at TOS is popped and copied to the left of the output. (Keep scanned one in the stack).

(6) When end of expression is reached, the remaining operators in the stack are popped and copied to the left of the output.

Eg:

K+L-M*N+(O^P)*W/U/V*T+Q

~~+*//**K+L-MN+^OPWUVTQ~~

++-+KL*MN*//*^OPWUVTQ

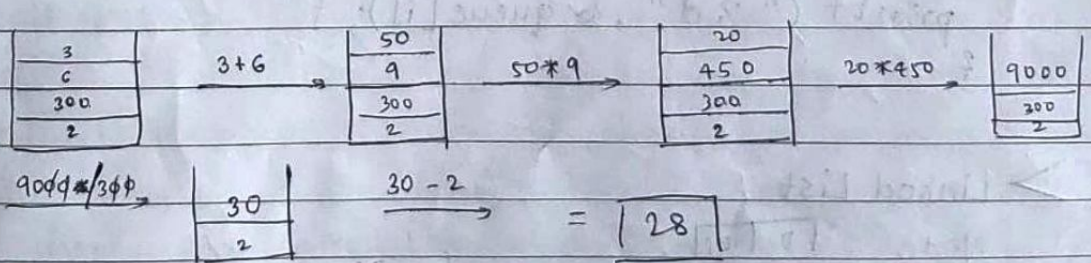
- Evaluation of prefix expression:

(1) Read expression from right to left and build stack of ~~operands~~. operands.

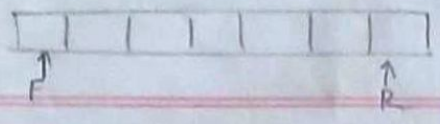
(2) When an operator is read, two operands are popped out of the stack. They are evaluated with the operator and the result is pushed into the stack.

(3) At the end of the expression, there must be only one operand in the stack.

Eg. -/* 20 * 50 + 3 6 300 2



Queue



> Implementation of queue using array:

- Initially there is no queue ∴ F = R = -1 (F = front ; R = rear)

size = 5

- Enqueue ( ) {

```
if (R == size -1)          } check if full
 printf ("overflow");
else
 { scanf ("%d", &element);
   if (F == -1) {          } if there was no queue before.
      {F = 0; }}
    R++;
    Queue [R] = element; }}
```

- Dequeue ( ) {

```
if (R == -1 || F > R) {    } check if empty
  printf ("Underflow"); }
else {
 element = Queue [F];
 printf ("%d is deleted", &element);
  F++; }}
```

- Display ( ) {

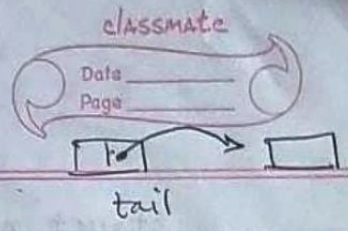
```
if (R == -1 || F > R) {
  printf ("Queue is empty"); }
else {
for (int i = F ; i <= R ; i++) {
  printf ("%d", & queue [i]); }
}
```

> Linked List

Node | D | NEXT |

NEXT ≡ Pointer D = data

Next is a self-referential pointer as one node is pointing to another node only.



- struct Node {
  int data;
  struct Node *next;
  } *new; → (new is ~~another~~ element of structure like an object in java, it is of type pointer as it needs to store both data and pointer)

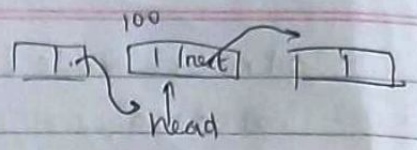
~~main ()~~

new = (struct Node *) malloc (size of (struct Node));
pointer type — datatype

- head and tail pointers are null when there is no linked list.
  Head ~~po~~ is the address of first node.
  Tail is the address of last node.
  Last node always has data and null pointer.

- struct node{
  int data;
  struct node *next;
  } *new, *head, *tail;
  create list () {
  ~~do~~
  new = (struct node *) malloc (size of (struct node));
  scanf ("%d", &value);
  new → data = value;
  new → next = NULL;
  if (head = = NULL) { } no list existed
  head = new;
  tail = new; }
  else {
  tail → next = new;
  tail = new; }

- For displaying linked list:
  temp = head;




```
struct node {
int data;
struct node *next;
}*new, *head, *tail, *temp;
display () {
temp = head;
while (temp != NULL) {
 printf ("%d", temp -> data);
 temp = temp -> next; }
```

- Inserting element at beginning:

```
new = (struct node *) malloc (sizeof (struct node));
scanf ("%d", &value);
new -> data = value;
new -> next = head;
head = new;
```

- Insert at end:

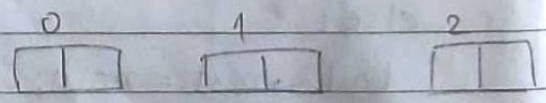
```
scanf ("%d", &value);
~~tail -> next = new;~~ new -> data = value;
new -> next = NULL;
tail -> next = new;
tail = new; return head;
```

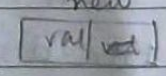
i = 0
temp = temp -> next ⇒ temp 1.
new -> next = 2

- Insert in the middle:

```
~~head -> next~~ address = head -> next;
head -> next = new
new -> next = address;
temp = head;
scanf ("%d", &position);
for (i = 0 ; i < position - 1; i++) {
temp = temp -> next;
new -> data = value;
```

pos = 2





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

- Deletion from start

```
DeleteFirstNode () {
temp = head;
head = head->next;
free (temp); }    //for freeing memory allocated
```

- Deletion from end

```
DeleteLastNode () {
temp = head;
while (temp->next != 0){
prev node = temp;
temp = temp->next }
if (temp == head) {
head = 0; }
else {
prevnode ->next = 0;
free (temp); }
```

- Delete from specified position:

```
DeleteMiddle () {   struct node * next node;
temp = head;
scanf ("%d", &position);
for (i = 1; i < position - 1; i++) {
temp = temp->next; }
nextnode = temp->next;
temp->next = nextnode->next;
free (next node);
}
```

- Rearranging railroad cars (Queue program)

581742963 ↑ input 987654321 ↑ output

Algorithm:

→ If the car is the expected next one in the output track, move it directly to the output track.

→ If not, move it to a holding track which operates in FIFO manner (queue).

→ The new car 'u' is moved to the holding track h that has, at its front, car with smallest label 'v' such that $v < u$.

→ The front of each holding track has a smallest value.

★ Some codes:

```
- void linkedlisttraversal (struct Node *temp){
  while (temp != NULL){
    printf ("%d", temp -> data);
    temp = temp -> next; }}
- //create list
  int main(){
  int value;
  char ch = 'y';
  int position;
  while (ch == 'y'){
  new = (struct Node*) malloc (sizeof (struct Node));
  scanf ("%d", &value);
  new -> data = value;
  if (head == NULL){
  head = new;
  tail = new; }
  fflush (stdin);
  printf ("Y-continue, N-exit");
  scanf ("%c", &ch); }
  printf ("Enter value to be inserted");
  scanf ("%d", &value);
```

Check this

```
head = InsertAtBegin (head, value);
printf ("Linked List");
linkedList Traversal (head);
return 0;
```

- Queue codes:

```
struct queue {
int size, f, r;
int * arr; }
int ~~isFull~~ isEmpty (struct queue *q) {
if (q->r == q->f) {          F=R
return 1; }
return 0; }
int isFull (struct queue * q) {
if (q->r == q->size - 1) {     R = size-1
return 1; }
return 0; }
void enqueue (struct queue *q, int val) {
if (isFull (q)) {
printf ("This queue is full"); }
else {
q->r ++;                       R++
q->arr [q->r] = val;           arr[R] = value
printf ("Enqued element: %d ", val); } }
int dequeue (struct queue * q) {
int a = -1;
if (isEmpty (q)) {
printf ("This queue is empty"); }      a = -1
else {
q->f ++;                       a = arr [F]
a = q->arr [q->f]; }           F++
return a; }
```

```
int main( ){
struct queue q;
q.size = 4;
q.f = q.r = 0;
q.arr = (int*) malloc (q.size* sizeof (int));
enqueue (&q, 12);
enqueue (&q, 15);
enqueue (&q, 1);
printf ("Dequeuing element %d ", dequeue (&q));
  "
  "
enqueue (&q, 15);
  "
  "
if (isEmpty(&q)) {
printf ("Queue is empty"); }
if (isFull (&q)) {
printf ("Queue is full "); }
return 0;
```

- CLL: CIRCULAR LINKED LIST

- void CLLTraversal (Struct Node * head)

```
{ temp= head;
  do {
    printf ("Element is %d ", temp→data);
    temp = temp → next; }
  while (temp != head);
}
```

- Creating CLL (see code for LL)

```
new → data = value;
if (head == NULL) {
 head = new;
```

```
tail = new;
head -> next = head; }
~~Flush(st~~ else {
tail -> next = new;
tail = new;
tail -> next = head; }
```

- Insert at beginning:

```
new -> data = data;
temp = head -> next;
while (temp -> next != head)
{temp = temp -> next; }
temp -> next = new;
new -> next = head;
head = new;
```

- Insert at end:

```
~~tail -> next~~ = new -> data = data;
tail -> next = new;
tail = new;
new -> next = head;
```

- Insert ~~at~~ in middle:

```
temp = head;
scanf ("%d", &position);
~~for~~ i = 1;
new -> data = data;
~~for~~ while (i < pos - 1) {
temp = temp -> next;
i++; }
~~new -> temp -> next = new~~
new -> next = temp -> next;
temp -> next = new;
```

- Delete from start:

```
temp = head;
head = head->next;
free (temp);
tail->next = head;
```

- Delete from end:

```
temp = head;
struct node *p = head->next;
while (p->next != head)
{p = p->next; }
p->next = temp->next;
free (temp);
return p->next.
```

- Delete from end:

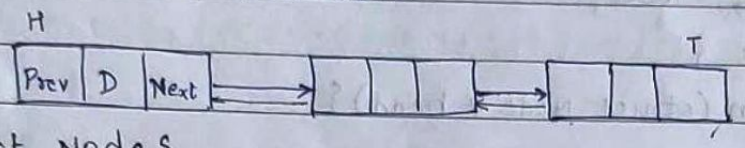
```
temp = head;
while (temp->next != head)
 { prevnode = temp;
   temp = temp->next; }
if (temp == head)
 {head = 0 }
 else
 {prevnode->next = head;
   tail = temp; }
 free (temp);
 return head;
```

- Delete from middle:

```
pos, i = 1;
temp = head;
while (i < pos-1) {
 temp = temp->next;
 i++; }
```

```
nextnode = temp->next;
temp->next = nextnode->next;
free (nextnode);
} return head;
```

- DLL: Double Linked List



- Struct Node {

```
int data;
struct Node * next;
struct Node * prev;
};
```

- Traversal:

```
void traversal (struct Node * head){
temp = head;
do{
printf("Element is %d", temp->data);
temp = temp->next;}
while (temp->next != NULL);
}
```

- Create:

```
new->data = data;
new->next = NULL;
if (head == NULL){
head = new;
tail = new;
head->prev = NULL;}
else {
tail->next = new;
new->prev = tail;
tail = new;}
```

- Insert at beginning:

```
atbegin (struct Node * head, data) {
 new→data = data;
 new→prev = NULL;
 new→next = head;
 head→prev = new;
 head = new; }
```

- Delete at Begin (struct Node * head) {

```
 temp = head;
 head = head→next;
 head→prev = NULL;
 free (temp);
 return head; }
```

- Insert at End (struct Node *head, data) {

```
 new→data = data;
 new→next = NULL;
 new→prev = tail;
 tail→next = new;
 tail = new;
 return head; }
```

- Delete at End (struct Node *head) {

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

- Insert In Mid (struct Node * head, data, pos) {

```
 int i = 1;
 new→data = data;
 temp = head;
 while (i < pos-1) {
 temp = temp→next;
 i++; }
```

```
temp1 = temp->next;
temp->next = new;
new->next = temp1;
temp1->prev = new;
new->prev = temp;
return head; }
```

- Delete At Mid (struct Node *head, pos) {

```
int i=1;
temp = head;
while (i<pos) {
temp = temp->next;
i++; }
prevnode = temp->prev;
nextnode = temp->next;
prevnode->next = nextnode;
nextnode->prev = prevnode;
free(temp);
return head; }
```

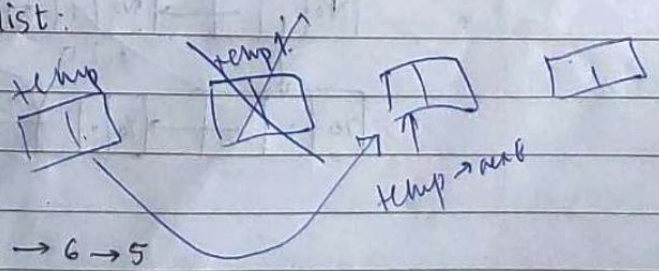
(Q) For lab: Check whether linked list is palindrome or not.

1 2 3 4 5 6 7
r a c e c a r

```
create list ( ) {
int x1 = deleteatBegin();
int x2 = deleteatEnd();
if (x1 != x2) { not a palindrome;
  break; }
else { i++ }; }
```

(Q) Remove duplicates from sorted list:

Eg: 1 → 1 → 2 → 2 → 3
H T



(Q) Given: 1 → 2 → 3 → 4 → 5 → 6

Swap output 2 → 1 → 4 → 3 → 6 → 5

(Q) $5x^2 + 4x + 2$ Columnwise addition using linked list:

$\quad\quad\quad 5x - 5$ node → data : 5 node → next = 4

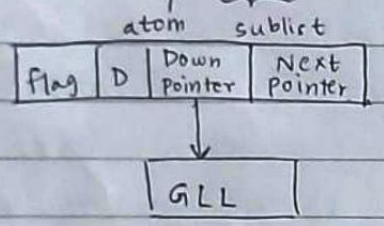
$5x^2 + 9x - 3$ node → power = 2

```
if ( node1 → power == node2 → power) {
  new → data = (node1 → data) + (node2 → data); }
```

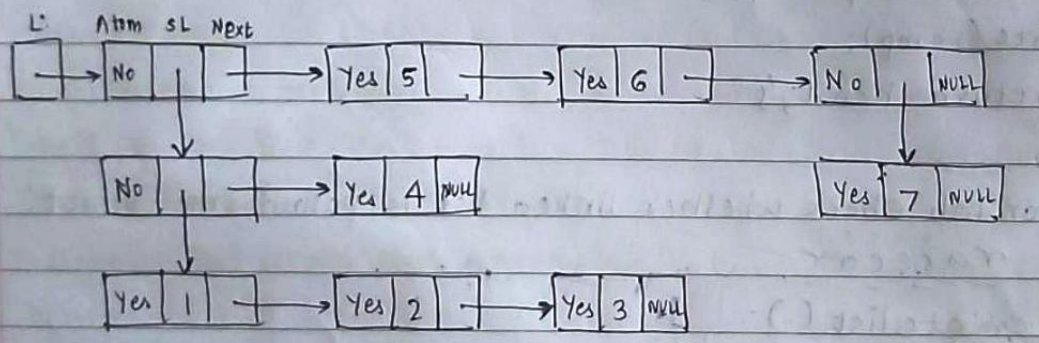
- Generalized Linked List: (GLL)

A finite sequence of $n \geq 0$ elements $a_0, a_1 \ldots$ where a_n is either an atom or a list. The elements a, $0 \leq i \leq n-1$ that are not atoms are called as sublists.

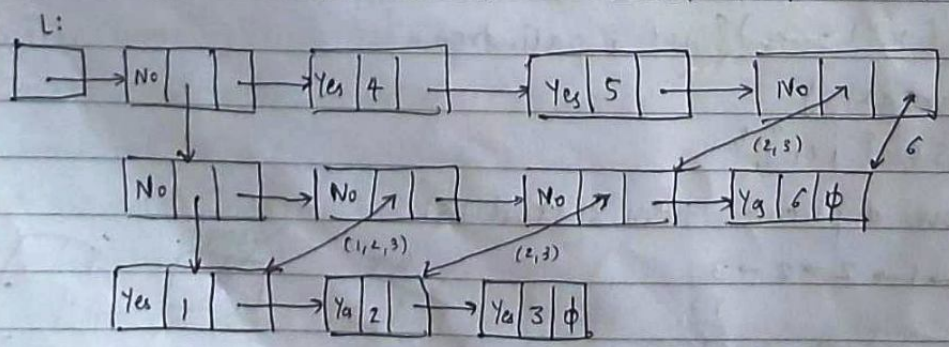
- Eg: $A = \{a, (b, c)\}$



- Eg: $L = (((1,2,3), 4), 5, 6, (7))$

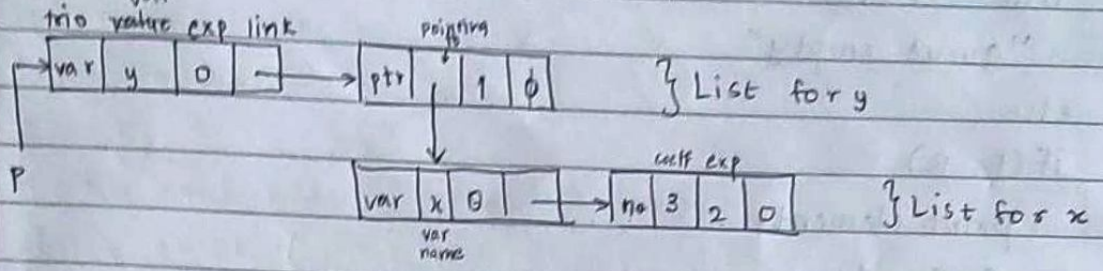


- Eg: $L = (((1,2,3), (1,2,3), (2,3), 6), 4, 5, ((2,3), 6))$



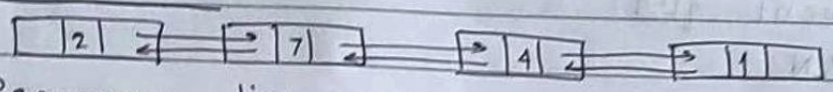
trio, ~~value~~ var, exp, link

- trio can be variable, pointer or number.
- Eg: $(3x^2)y$



- Eg: $P(x,y,z) = ((x^{10}+2x^8)y^3 + 3x^8y^2)z^2 + ((x^4+6x^3)y^4 + 2y)z)$

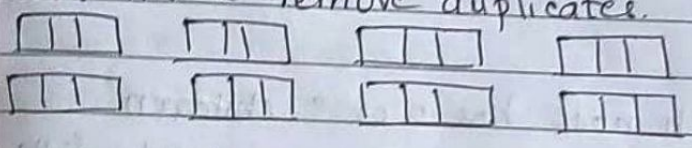
LAB: DLL

(Q) 

Rearrange list such that all even nos are displayed first.

(Q) Reverse linked list

(Q) Sort list & remove duplicates.

(Q) 
} merge 2 sorted lists.

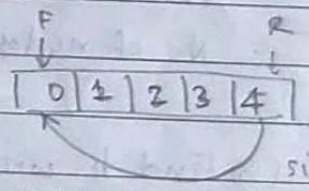
Single Linked List

(Q) Equivalence class Problem

$0 \equiv 4,\ 3 \equiv 1,\ 6 \equiv 10,\ 8 \equiv 9,\ 7 \equiv 4,\ 6 \equiv 8,\ 3 \equiv 5,\ 2 \equiv 11,\ 11 \equiv 0$

output: $\underbrace{\{0, 2, 4, 7, 11\}}_{\text{all equal}}$ $\{1, 3, 5\}$ $\{6, 8, 9, 10\}$

- CIRCULAR QUEUE:



size = 5.

$R+1 = 5 \% 5 = 0 = F$

& let $R = 2$

$R+1 = 3 \% 5 = 3$

$q[3] = x.$

- void enqueue (int x){

  if (F==-1 && R==-1){ → empty

    F=R=0;

    q[R] = x;}

  if ((R+1) % N) == F)

    "Queue overflow"

  else{

    R = (R+1) % N;

    q[R] = x;}

$(r+1)\% N$

- void dequeue ( ){
  if (F == -1 && R == -1)
    "queue empty"
  else
  if (F == R)
    print element } one element
    F = R = -1;
  else
  print element q[F]
    F = (F+1) % N

- ~~Jos~~ Josephus Problem: Using circular queue.

TREE:

- Full Binary tree: Each node has 0 or 2 children.
- Complete Binary tree: If all levels are completely filled except possibly the last level and last level has all keys as left as possible.
- Perfect binary tree: All internal nodes have 2 children and all leaves are at same level.
- No of nodes in perfect tree = $2^{h+1} - 1$ where h is height.
- No of nodes in full tree = $2^{h+1} - 1$

(Q) Find h with 50 nodes.

→ 49 nodes. (if skewed)

$2^{h+1} = 51$

- Tree structure | left | D | Right |

- struct node * create () {
  struct node * temp, * root;
  int data, choice;
  temp = malloc . . . . . };
- 0-exit
  1 - for new node
  choice (0 OR 1)
  ~~c~~ if (choice == 0)
  return;
  else {
  data
  temp → data = data;
  printf ("Enter left child of %d", data);
  temp → left = create ();
  printf ("Enter right child of %d", data);
  temp → right = create ();
  return temp; }
- Inorder traversal
  void Inorder (struct .... * node) {
  if (node == NULL)
  return;
  ~~Print~~ Inorder (node → left);
  printf ("%d", node → data);
  Inorder (node → right);

Inorder: Left, root (data), right.
LDR

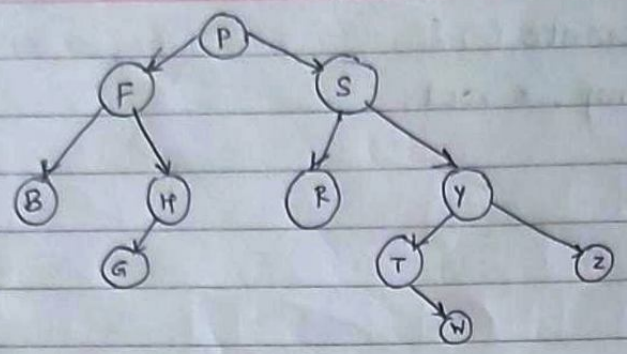
Preorder: DLR

Postorder: LRD

- DFS: Depth First Search
  BFS: Breadth First Search

- ~~void Postorder (stru~~
  Mam has shared ppts.

-



Inorder: B F G H P R S T W Y Z

Preorder: P F B H G S R Y T W Z

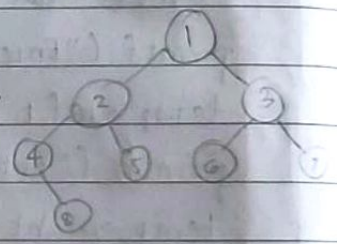
Postorder: B G H F R W T Z Y S P

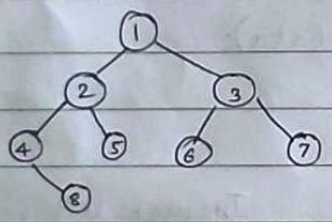
- Tree from two traversals:

Eg1: In : 4 8 2 5 1 6 3 7 LDR

Post : 8 4 5 5 2 6 7 3 1

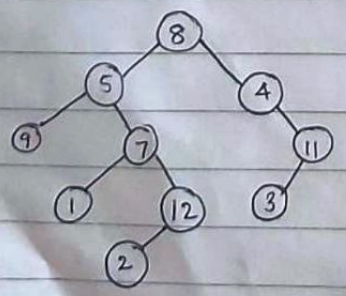
← Read postorder from right to left.





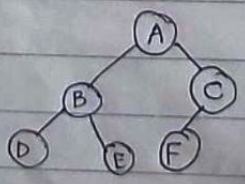
Eg2: Post: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

In: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11



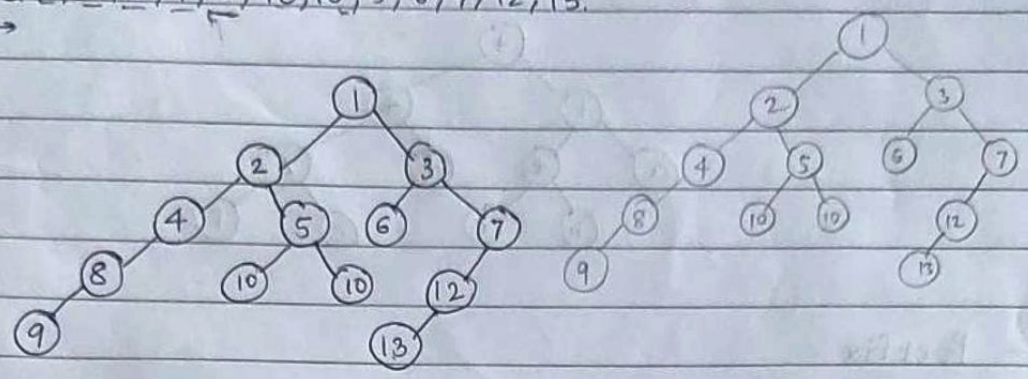
Eg3: In: D B E A F C

Pre: A B D E C F



Eg1: In: 9, 8, 4, 2, 10, 5, 10, 1, 6, 3, 13, 12, 7

Pre: 1, 2, 4, 8, 9, 5, 10, 10, 3, 6, 7, 12, 13.



- Read tree from Pre and Post from Ronak's notes.

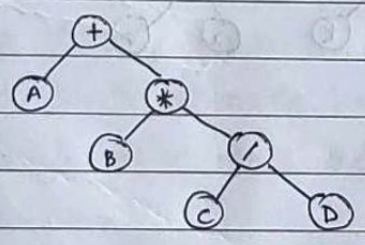
> Expression Tree:

- Infix expression

Eg1: A + (B * (C / D))

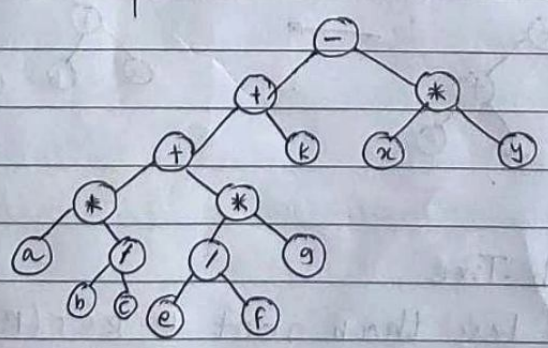
The operator with lowest precedence is generally the root.

Expression which will get evaluated at the end will be at a lower depth.



Eg2: a *(b / c) + (e / f)* g + k - x * y

Last operation performed will be -.



$1 \times (1) + (1) \times (1) + 1 - (1)$

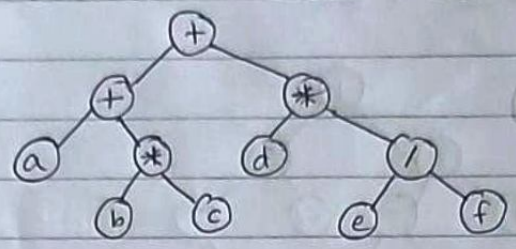
$1 + 1 + 0$

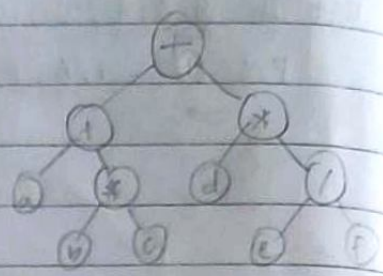
$= 2$

$2 \times (1) + (1) \times 2) + 2 - (4)$

$2 + 2 + 2 - 4 \quad = 2$

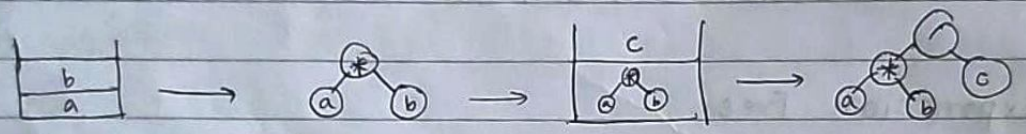
Eg3: a + (b * c) + d * (e / f)



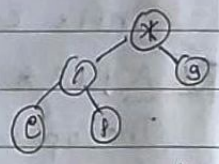


- Postfix

Eg1: ab * c / ef / g * + k + xy * -



Keep doing this.

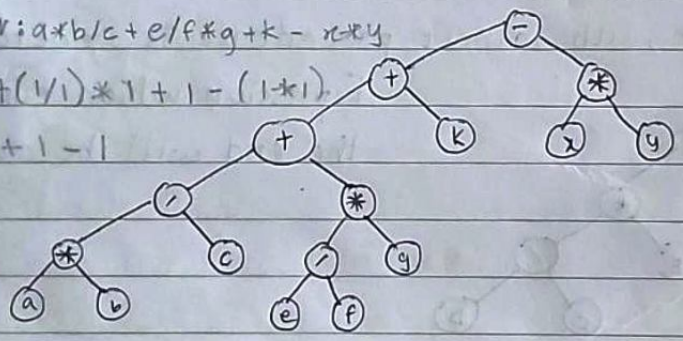


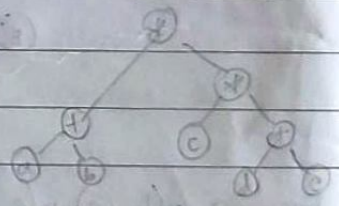
Inorder: a*b/c+e/f*g+k-x*y

1*(1/1)+(1/1)*1+1-(1*1)

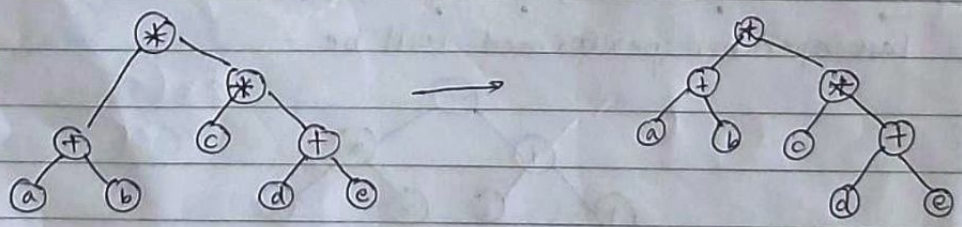
= 1+1+1-1

= 2





Eg2: ab + cde + **



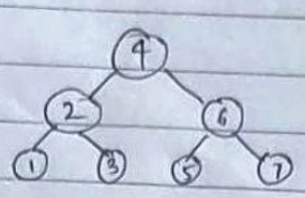
> BST : Binary Search Tree

LST (left subtree) is less than root & RST (Right) is greater than root.

NOTE: For BST, Inorder = Sorted List.

Eg1: 4 2 3 6 5 7 1 : Pre.

In: 1 2 3 4 5 6 7



> Delete:

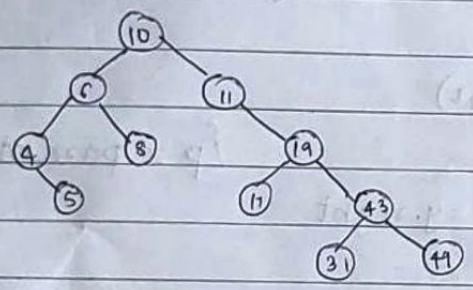
- Cases:

(1) Node has 0 child.

(2) 1 child.

(3) 2 child.

Eg1:

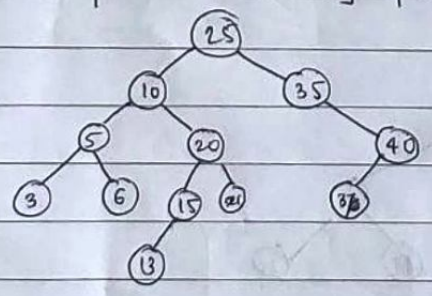


0 child : Direct delete

1 child : Replace node by child.

2 child : Replace node by predecessor or successor.

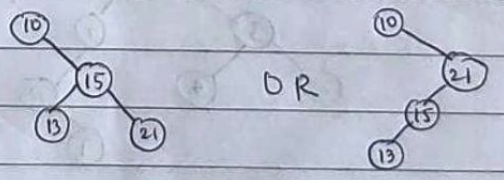
Eg2:



37 can be deleted directly.

35 replaced by 40.

If you want to delete 20, 2 cases:



> Searching : Recursive Algorithm:

Tree (x, k)

(1) If x == NIL or k == x : key

(2) Return x

(3) If k < x.key

(4) return Tree (x.left, k)

(5) else return Tree (x.right, k)

- tree Min (x)

  while x.left ≠ NIL

  x = x.left

  return x.

- tree Max (x)

  while x.right ≠ NIL

  x = x.right

  return x

- Finding successor: two cases:-

  Right subtree of node x is non empty.

  successor of x is tree Min (x.right)

  If x.right ≠ NIL

  return tree Min (x.right)

  y = x.p {p ≡ parent}

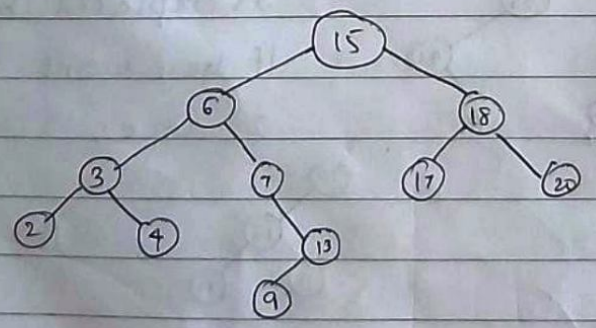
  while y ≠ NIL and x == y.right

  {x = y

  y = y.p }

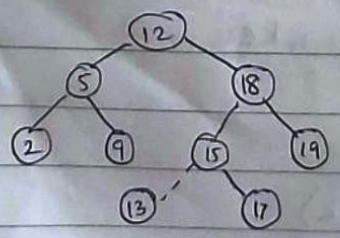
  return y.

- If the current node is the left child of its parent, parent is the successor node.



- Successor of 4 is 6, lowest ancestor of 4 whose left child is also an ancestor of 4.

> Insertion

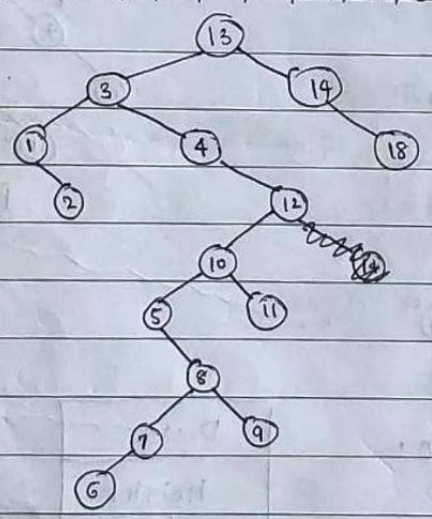


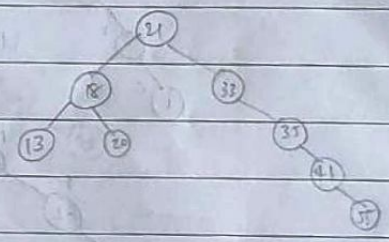
- TreeInsert (T, z)

```
y = NIL
x = T.root
while x ≠ NIL
   { y = x
     if z.key < x.key
        x = x.left
     else x = x.right }
z.p = y
if y == NIL                 } if tree is empty, z will become root
     T.root = z
else z.key < y.key
     y.left = z
else y.right = z
```

- Tree from keys;

13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18





Inorder: 13, 18, 20, 21, 33, 35, 41, 55

Preorder: 21, 18, 13, 20, 33, 35, 41, 55

Postorder: 13, 20, 18, 55, 41, 35, 33, 21

✱ AVL TREE:

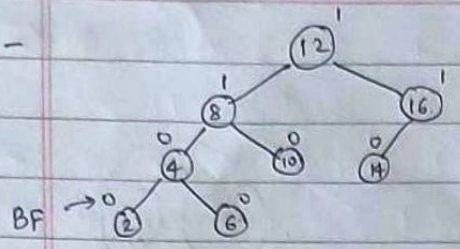
Balancing factor (BF) = (left child - Right child) height of

BF should be 0, 1 or -1. If it is anything else, balance the tree.

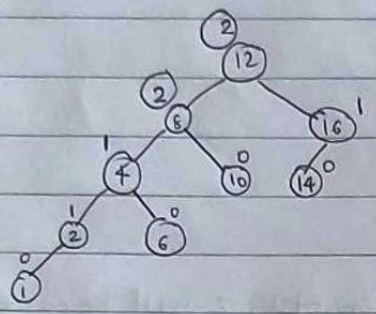
Also called height balance tree.

AVL Trees are Binary search trees first.

0, 1, -1

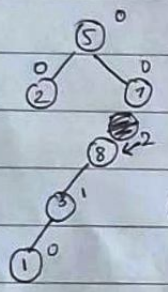
- 

∴ AVL Tree

- 

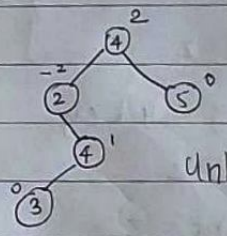
∴ Not balanced.

∴ Not AVL.

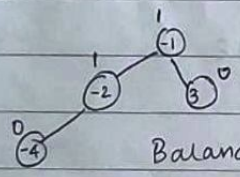
- 

Balanced

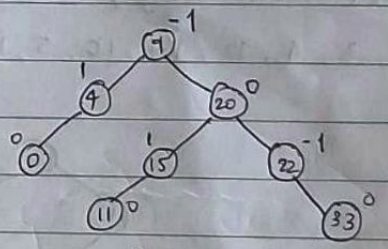
Unbalanced.



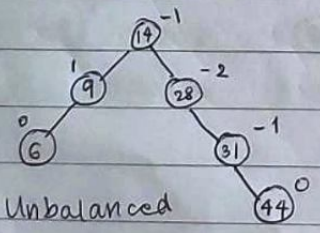
Unbalanced



Balanced



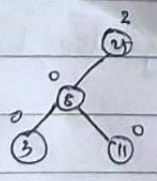
Balanced



Unbalanced

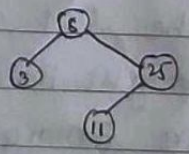
- For AVL, structure:

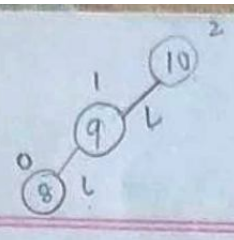
Data	
Height	
Left	Right

- 

Unbalanced

$\xrightarrow[\text{right}(2)]{\text{rotate}}$

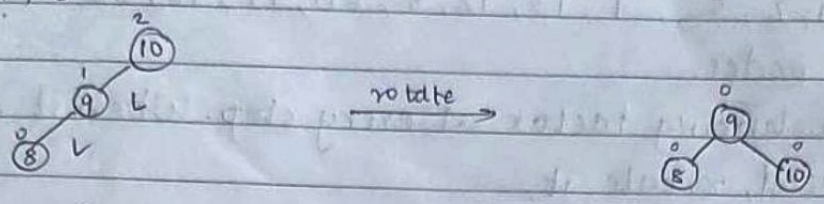




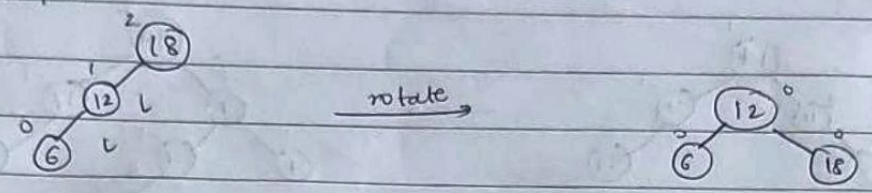
R

R

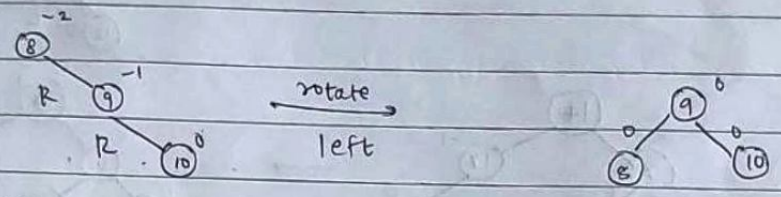
(Q) 10, 9, 8



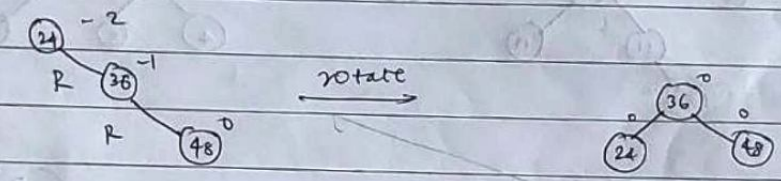
(Q) 18, 12, 6



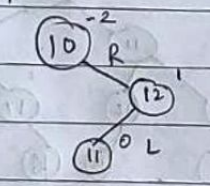
(Q) 8, 9, 10



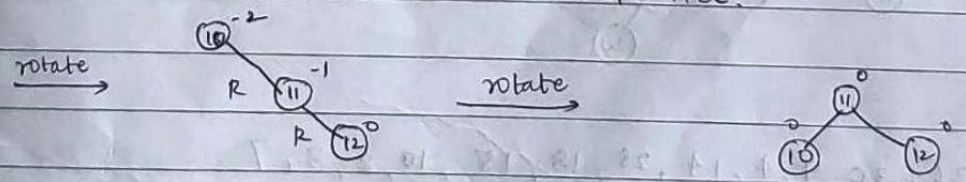
(Q) 24, 36, 48



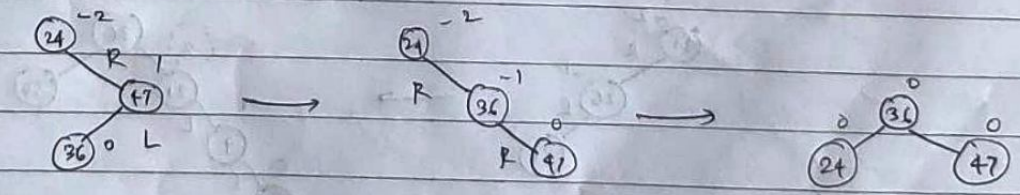
(Q) 10, 12, 11



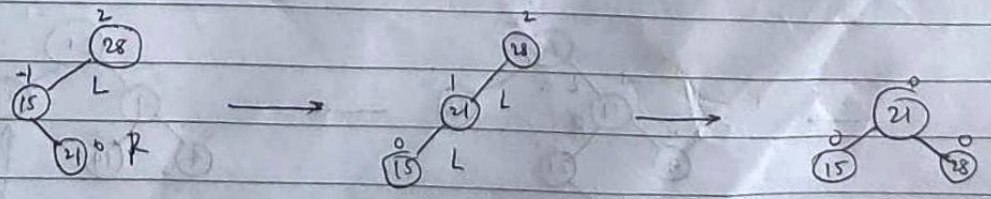
for RL and LR apply 2 rotations
First convert RL to RR then balance.



(Q) 24, 47, 36



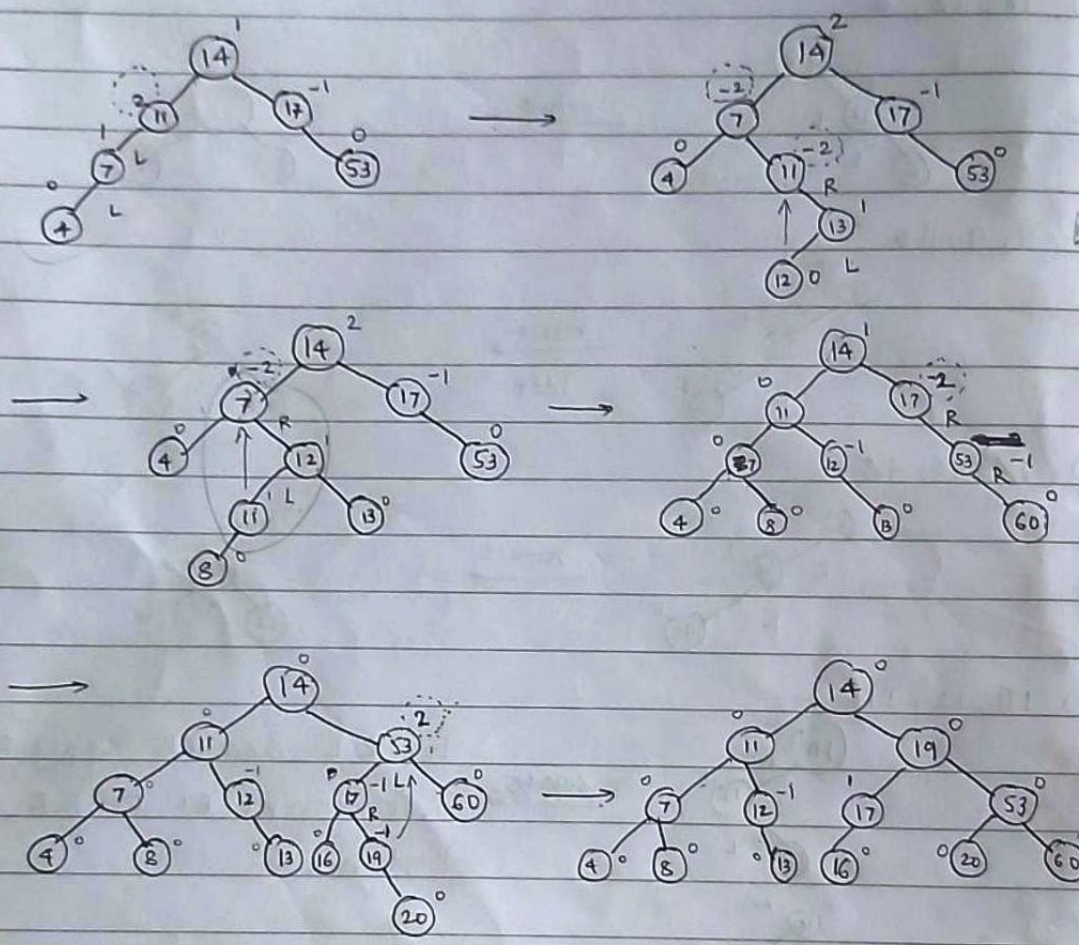
(Q) 28, 15, 21



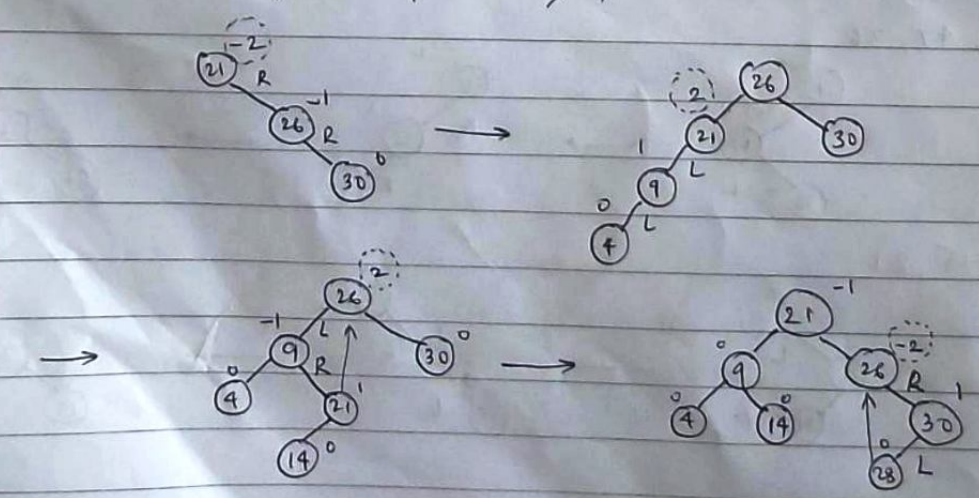
(Q) 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

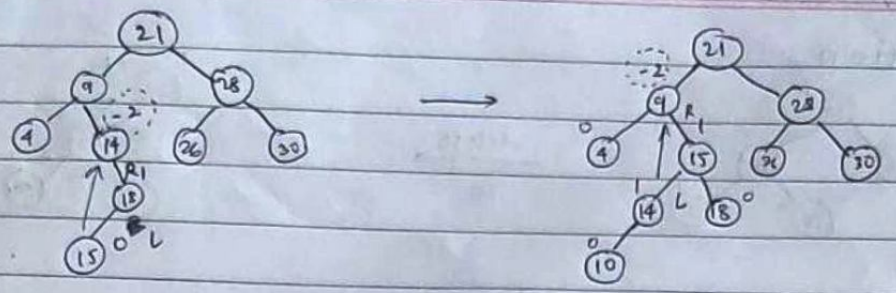
Insert nodes.

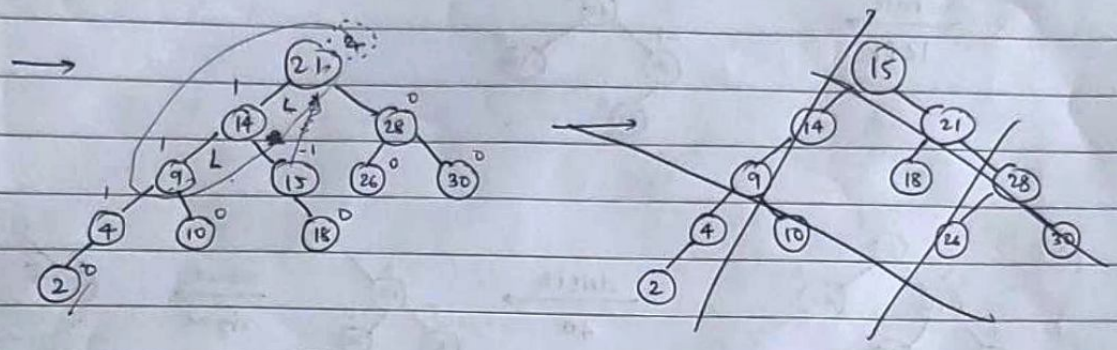
→ Check balancing factor at every step. When it gets unbalanced, rotate it.

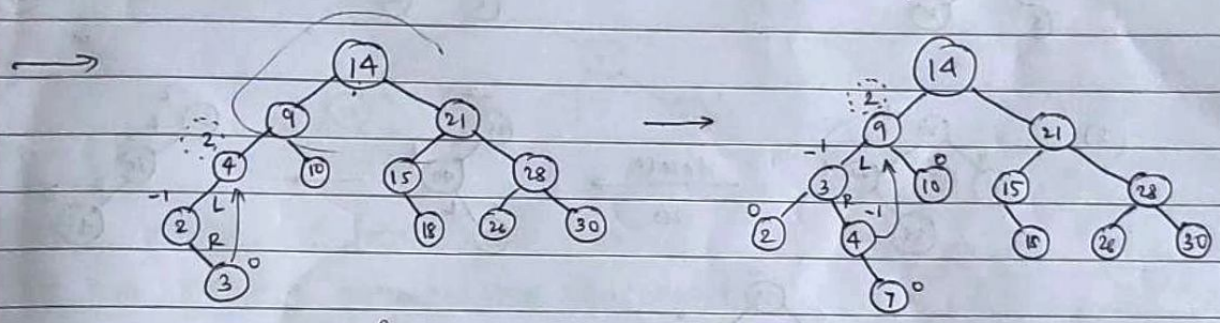


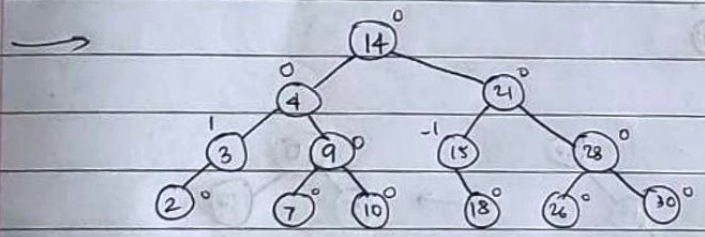
(Q) 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7



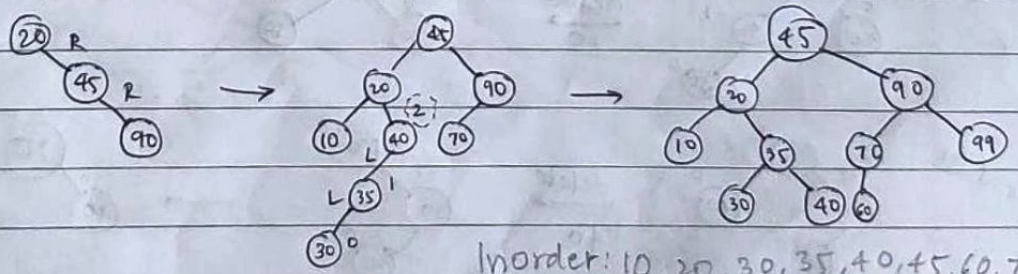








(Q) 20, 45, 90, 70, 10, 40, 35, 30, 99, 60



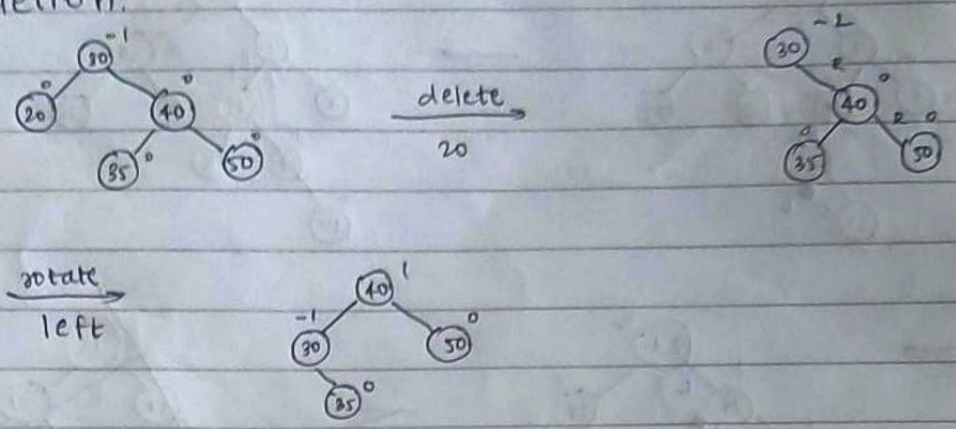
Inorder: 10, 20, 30, 35, 40, 45, 60, 70, 90, 99

~~Preorder: 10, 30, 40, 35~~

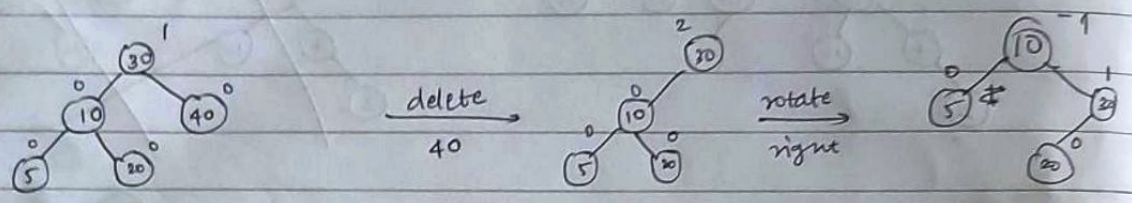
Preorder: 45, 20, 10, 35, 30, 40, 90, 70, 60, 99

Deletion:

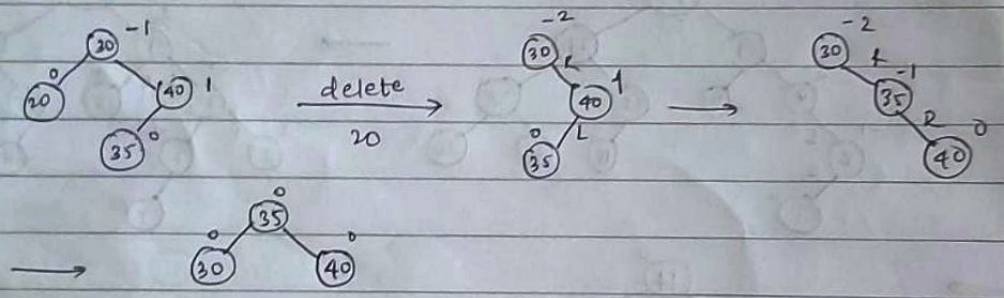
(1)



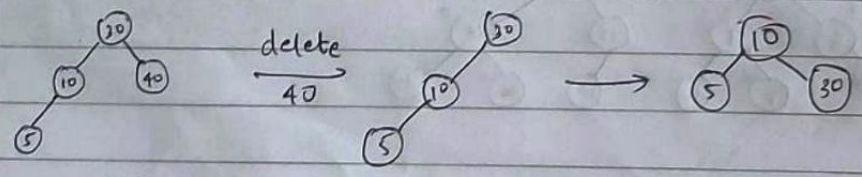
(2)



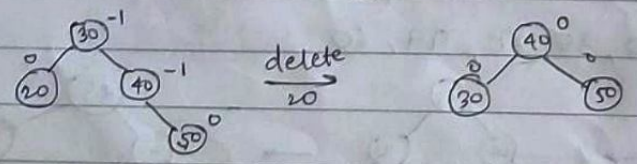
(3)



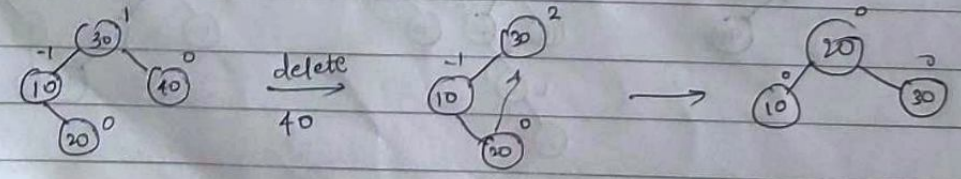
(4)



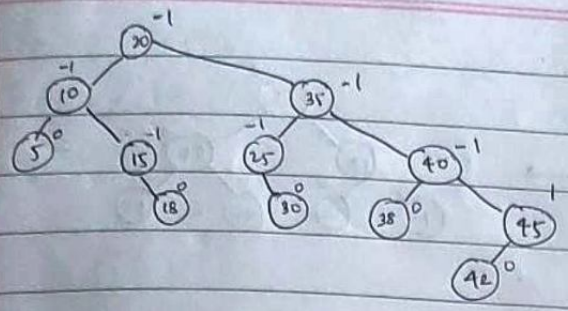
(5)



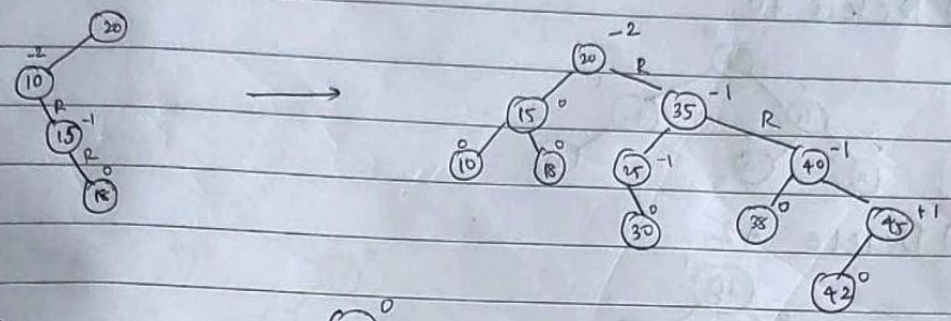
(6)



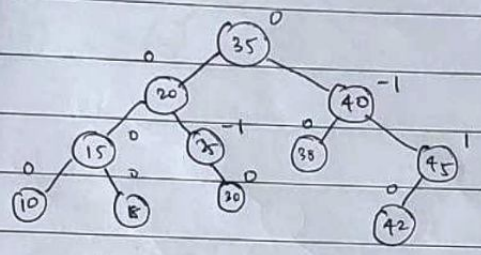
(Q)



(i) Delete 5

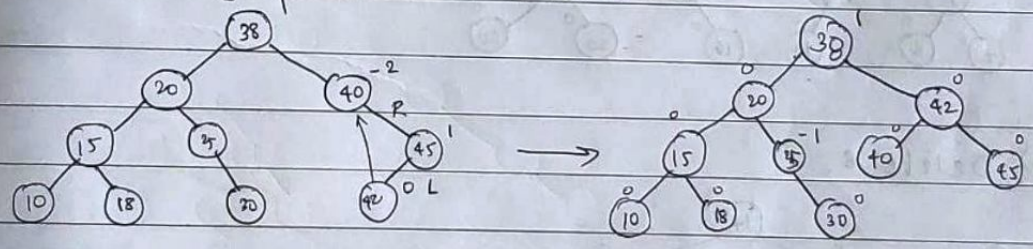


rotate left

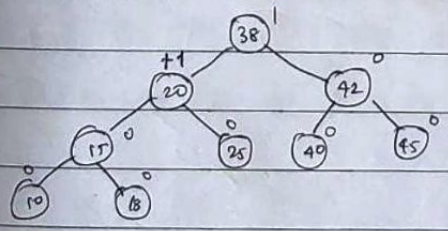


(ii) Now delete 35 from new tree.

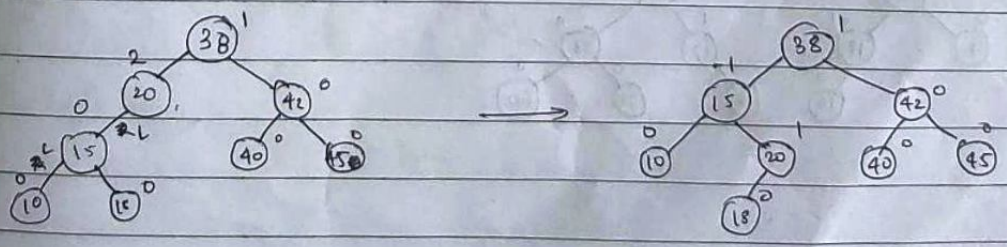
We are taking ~~predecessor~~ successor.



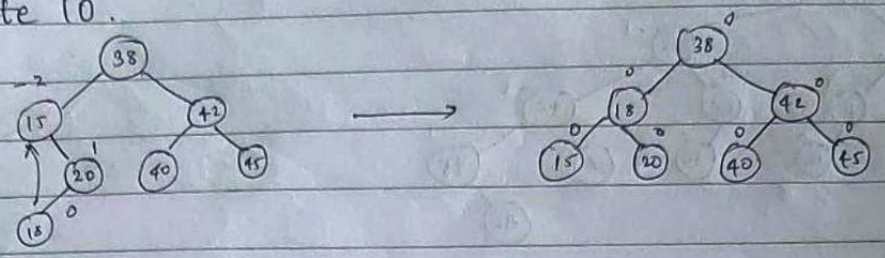
(iii) Delete 30 now.



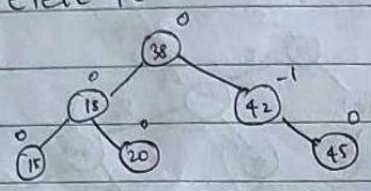
(iv) Delete 25 now.



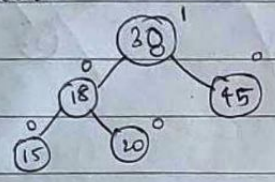
(v) Delete 10.



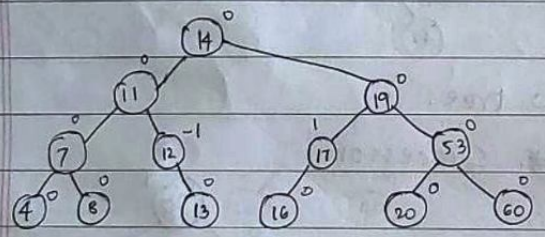
(vi) Delete 40



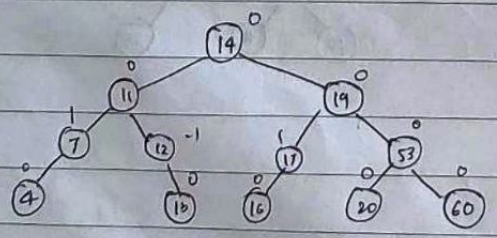
(vii) Delete 42



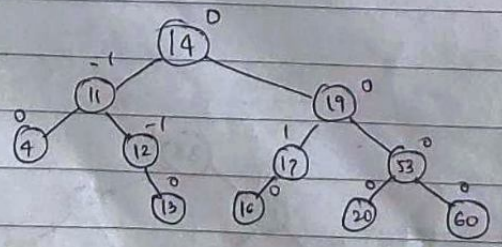
(Q)



(i) Delete 8.

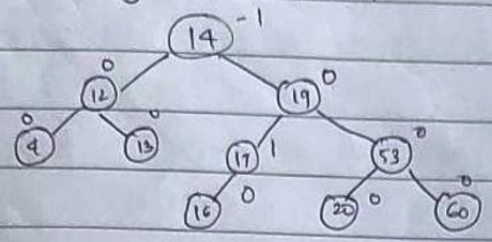


(ii) Delete 7.



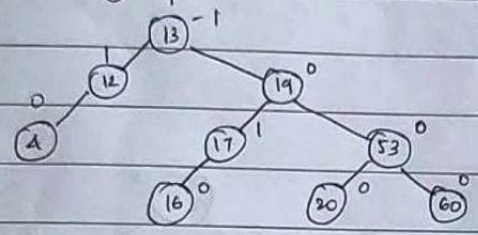
(iii) Delete 11

Replace by successor.

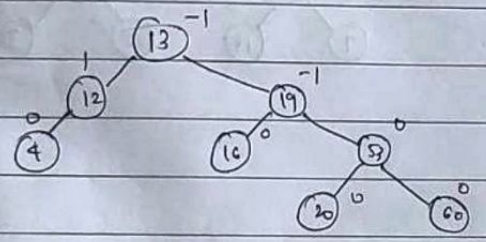


(iv) Delete 14.

Replace by predecessor.

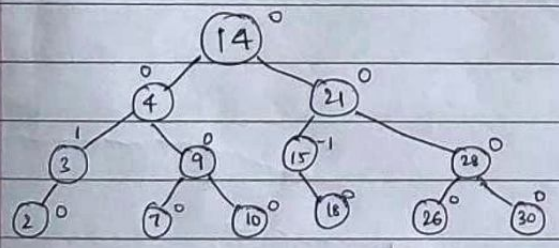


(v) Delete 17

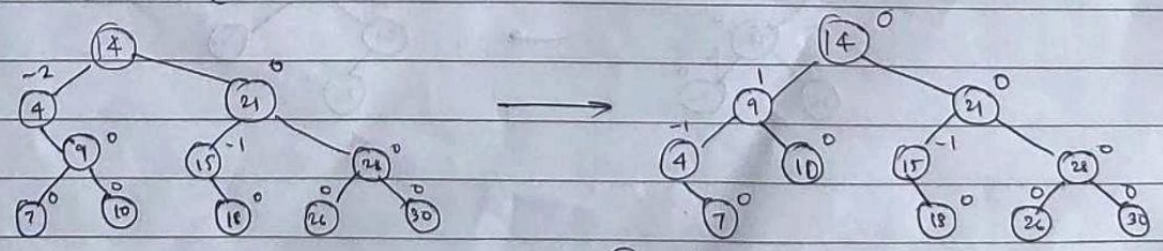


~~(vi) Delete~~

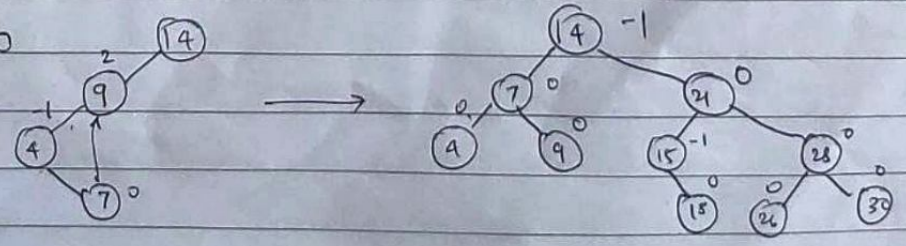
(Q)



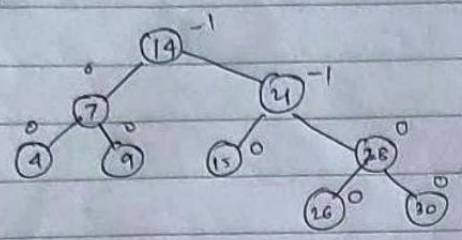
(i) Delete ~~4~~ 2 & 3.



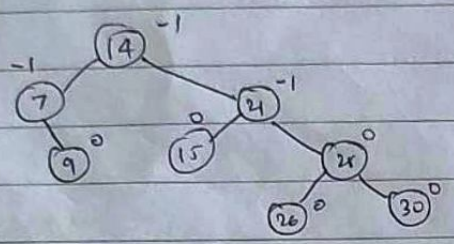
(ii) Delete 10



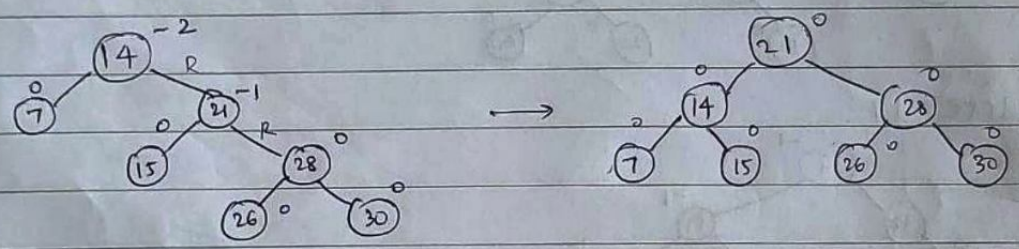
(iii) 18.



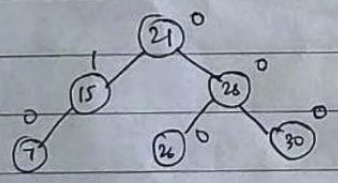
(iv) 4



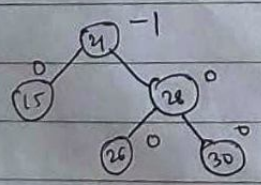
(v) 9



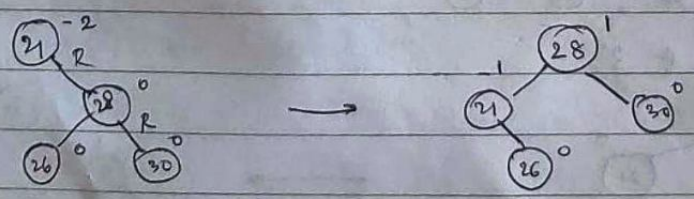
(vi) 14



(vii) 7



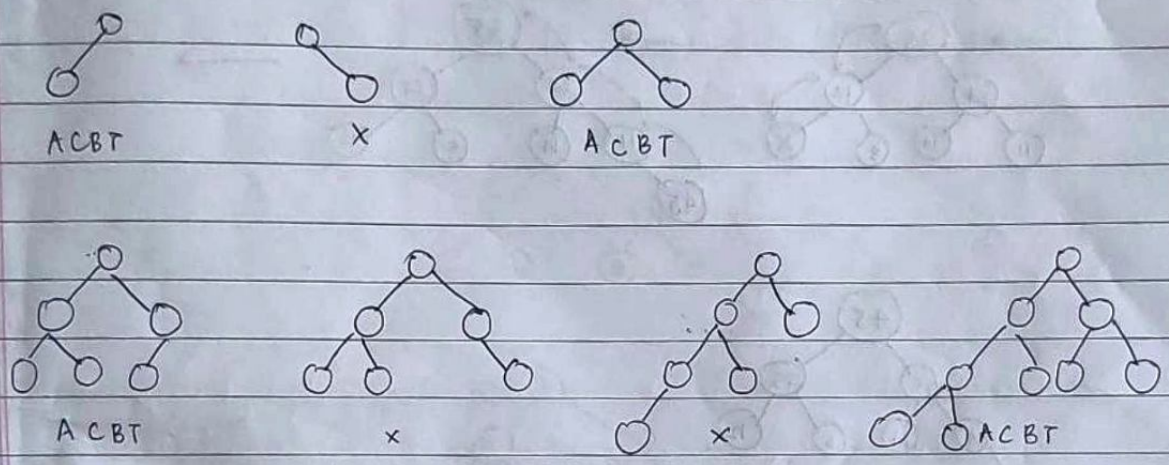
(viii) 15



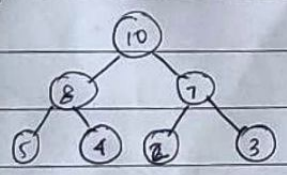
> HEAP TREE:

- ACBT (Almost complete binary tree): You cannot insert right child unless left is inserted. Also insertion has to be done level by level.
- Ordering : Min P < Child Root is smallest
  Max P > Child Root is largest

(Q)

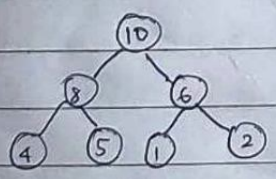


- Egs:

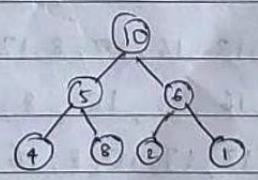


Max

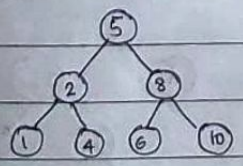
(Q)



Max ACBT



ACBT but not MAX or MIN



ACBT but not ordered

- Heaptree Construction Methods:

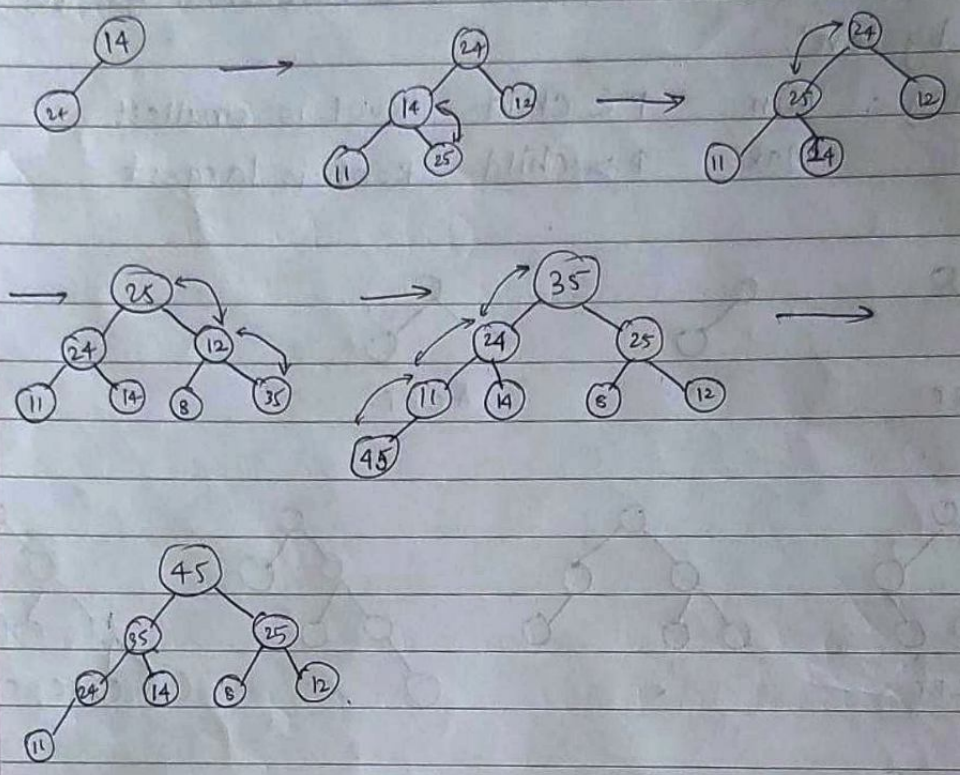
(1) 1 by 1

(2) Heapify

- Egs:

(Q) 14, 24, 12, 11, 25, 8, 35, 45

Max



- Height of tree = $\log_2 N$ → no of nodes.

∴ No of swaps / comparisons = $\log_2 N$

For n elements → $n \log_2 N$

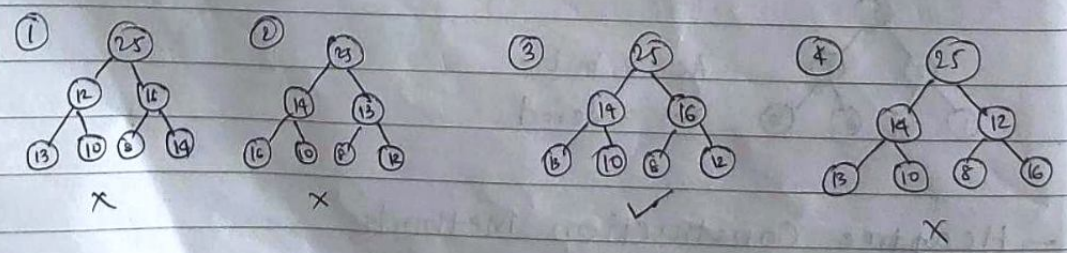
(Q) Which of the following array represents a max heap tree?

(1) 25, 12, 16, 13, 10, 8, 14

(2) 25, 14, 13, 16, 10, 8, 12

✓ (3) 25, 14, 16, 13, 10, 8, 12

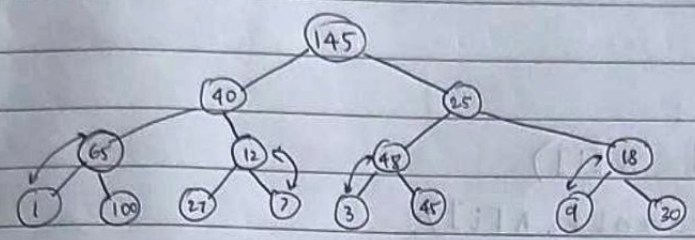
(4) 25, 14, 12, 13, 10, 8, 16



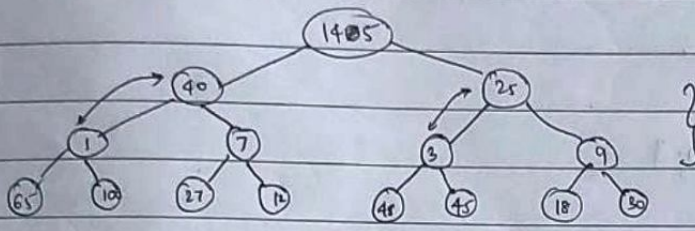
- Heapify Method:

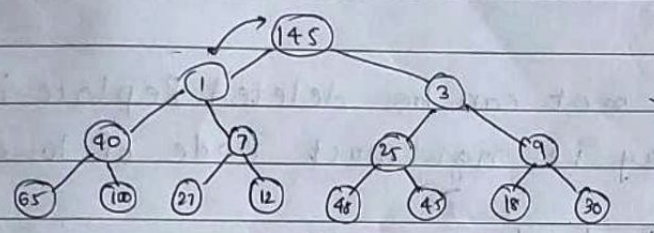
Eg: 145, 40, 25, 65, 12, 48, 18, 1, 100, 27, 7, 3, 45, 9, 30

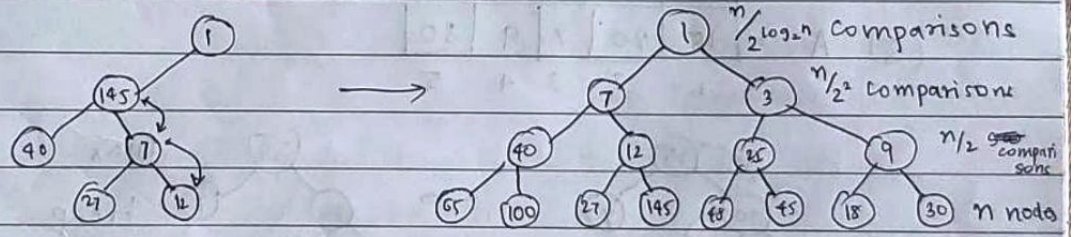
Min.



First check these 2 levels.







> Max Heap: For every node i the value of the node is less than or equals to ~~par~~ its parent value.

$\therefore$ A [Parent(i)] $\geq$ A [i]

> Min Heap:

A [Parent (i)] $\leq$ A [i]

- Parent is always at position $i/2$ if child is at i. Take the ~~higher~~ lower value in case of fraction. NOTE: Counting indices starts from 1 here.

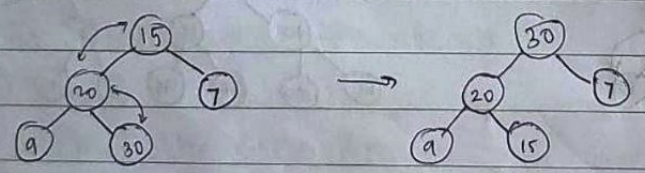
Array no of elements in array

- InsertHeap(A, n, value) ← For max.

```
{ n = n+1;
  A[n] = value;
  i = n;
  while (i > 1)
  { parent = i/2;
    if (A[Parent] < A[i])
    { swap (A[parent], A[i]);
      i = parent;
    }
    else { return; }
  }}
```

> Deletion: Only root can be deleted. Replace it with last element of array i.e, rightmost node of lowest level.

(Q) A = | 15 | 20 | 7 | 9 | 30 |
1 2 3 4 5



Max Heap.

Now keep deleting root to get sorted ~~element~~ array.
Left child = 2i , Right child = 2i+1
This method is called HeapSort Algorithm.

> For heapify, we need to find which ones are leaf nodes.
leaf node : $n/2 + 1$ to n
largest index of non leaf : $n/2$

- Heapify (A, n, i) → i will be n/2 i.e largest nonleaf index

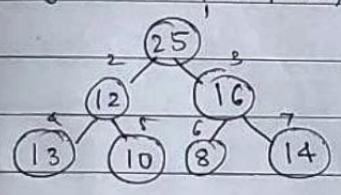
```
{ int largest = i;
  int l = 2i, r = 2i+1;
```

that chapter is CLOSED!

```
while (l≤n && A[l] > A[largest])
 { largest = l ; }
while (r≤n && A[r] > A[largest])
 { largest = r; }
if (largest != i)
{ swap (A[largest] , A[i]);
   Heapify (A, n, largest);
 }
}

Buildheap (A, n)
{ for (i = n/2 ; i ≥ 1 ; i--)
   Heapify (A, n, i); }
```

(Q) Array : 25, 12, 16, 13, 10, 8, 14 Max.

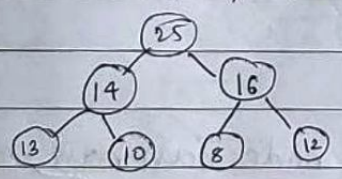


Array size = 7

Heap size = 1 ∵ 12, 13, 10 not max heap

∴ only one element (25) in heap. Remember left is first.

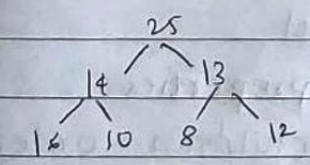
(Q) 25, 14, 16, 13, 10, 8, 12



Array size = 7

Heap size = 7

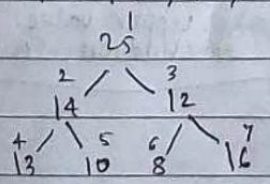
(Q) 25, 14, 13, 16, 10, 8, 12



Heapsize = 1

Array size = 7

(Q) 25, 14, 12, 13, 10, 8, 16



Arraysize = 7

Heapsize = 2

index before heap fails. Here it is failing at element 12 at index 3.

- Updating an element: In max, element can be increased. In min, element can be reduced.

• HeapExtractMax (A) {
  if A.heapsize <1
    error "heap underflow"
  max = A[1]
  A[1] = A[A.heapsize]
  A.heapsize = A.heapsize - 1
  Max Heapify (A,1)
  return max.

• HeapIncrease (A, i, key)
  { if (key < A[i])
      error
  A[i] = key
  while (i ≥ 1 & A[i/2] < A[i])
  exchange (A[i] & A[i/2]);
  i = i/2 ;

> B-TREE: Self balanced binary search tree. It can have any no of children.

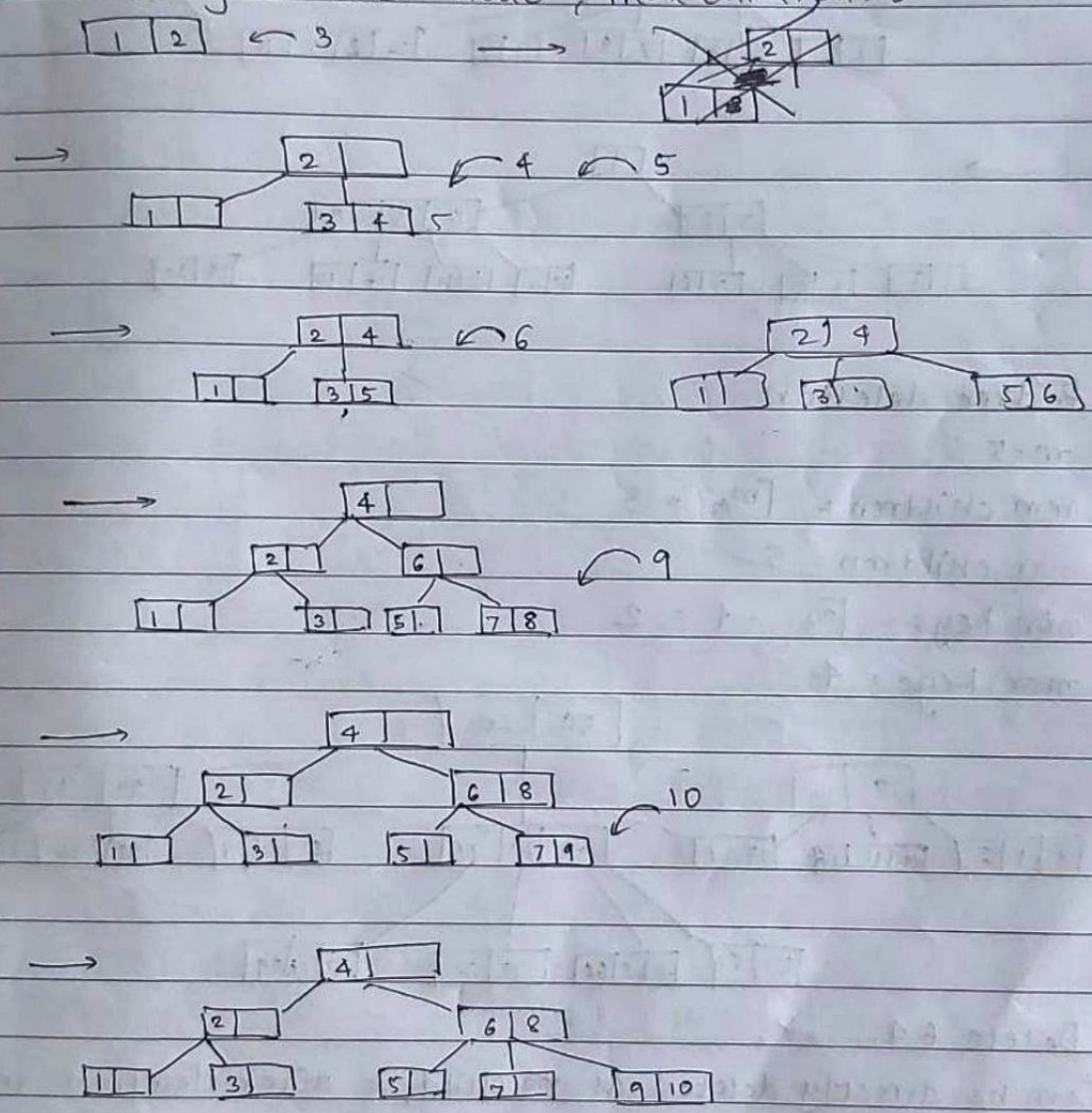
- m-way / m-ordered
- Generalization of BST in which node can have more than one key & more than 2 children.
- Maintains sorted data
- All leaf nodes must be at same level.
- B tree of order m has following properties:

• min children : leaf = 0 max children = order
  root = 2
  intermediate = m/2 ← take upper value. m=5 ⇒ children = 5/2 = 3.

- Max (m-1) keys in every node.
- min keys at root = 1 is also allowed
  at other nodes = $(m/2) - 1$ → greater value
- Insertion always from leaf.

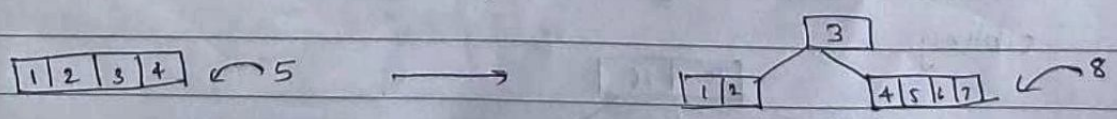
(Q) Create a Btree of order 3 by inserting values from 1-10.

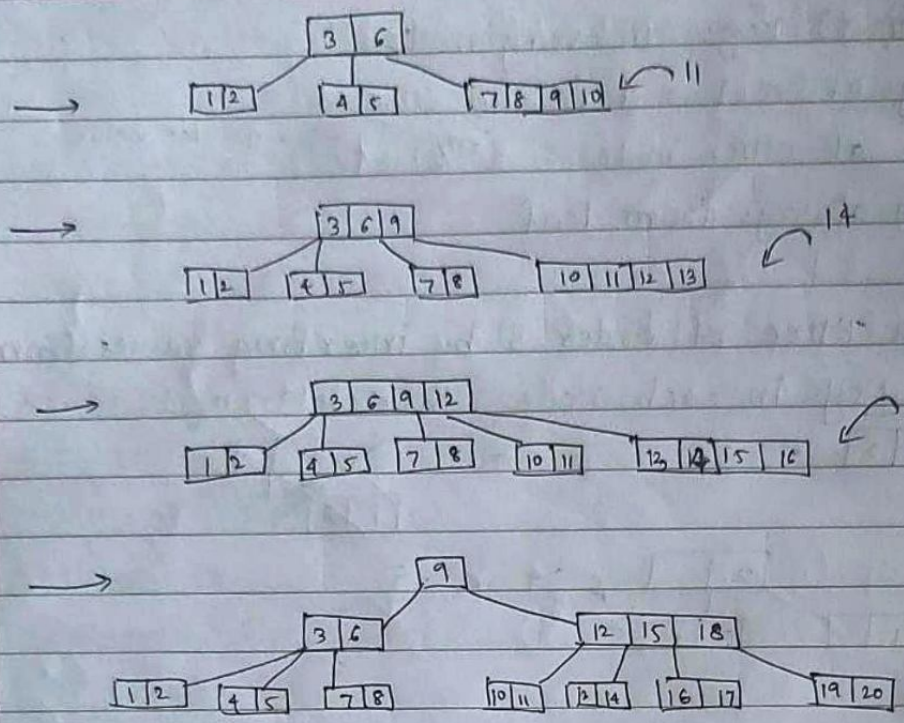
→ Max 2 keys in each node, max children = 3



(Q) 1-20 order 5

→ max children = 5, max keys = 4





- B tree deletion

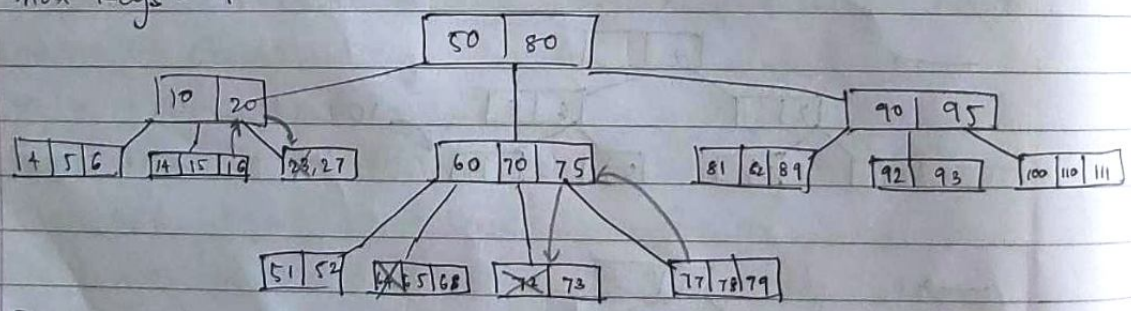
(Q) $m = 5$

min children = $\lceil m/2 \rceil = 3$

max children = 5

min keys = $\lceil m/2 \rceil - 1 = 2$

max keys = 4

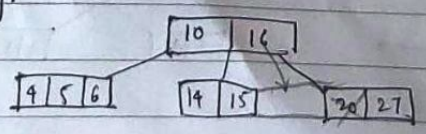


Delete 64.

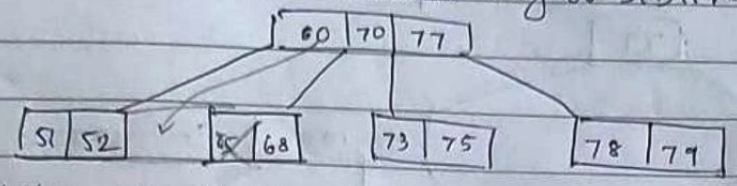
Can be directly deleted as no child, & after deletion, min no of keys are present.

Delete 23.

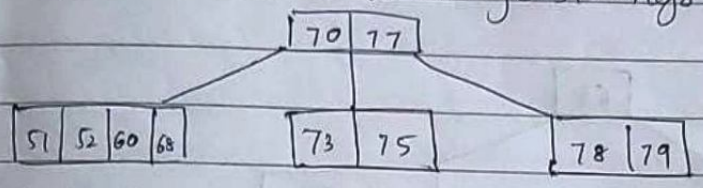
After deletion, only 1 key remains so not allowed. Borrow from sibling.



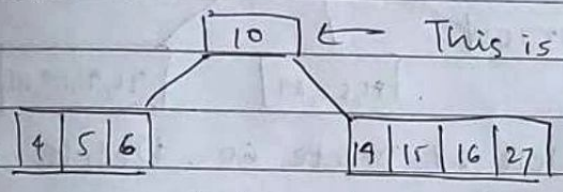
Delete 72. Borrow from right sibling.



Delete 65. Both R & L sibling don't have extra keys to borrow. So we need to merge siblings and then delete.

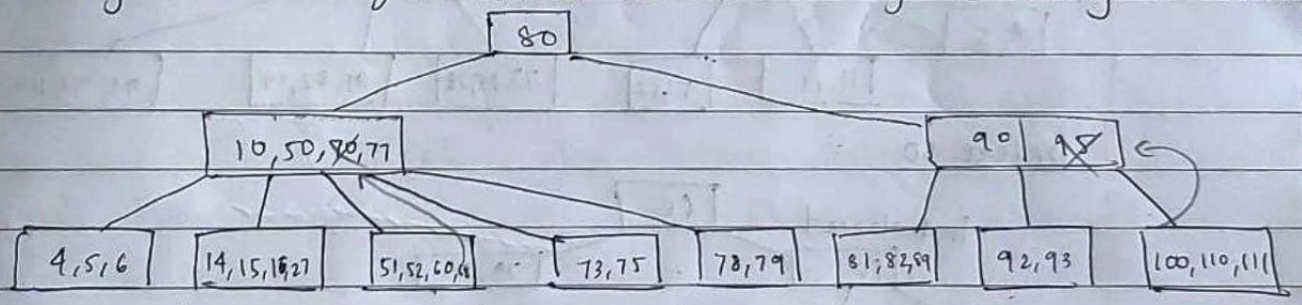


Delete 20



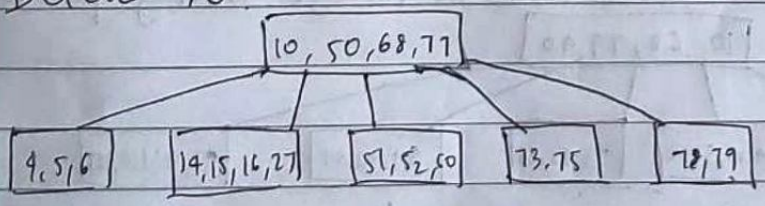
← This is imbalanced. Min 2 keys required

So we need to ~~ch~~ check with 10's siblings. 70, 77 only 2 keys there so you cannot borrow. Merge & bring 50 down.

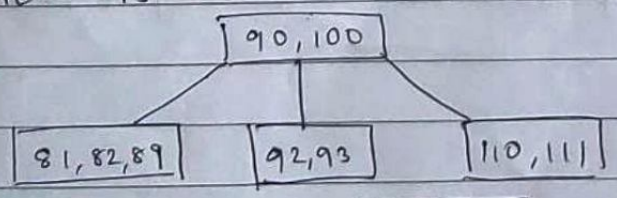


- For non-leaf nodes, replace with inorder successor or predecessor.

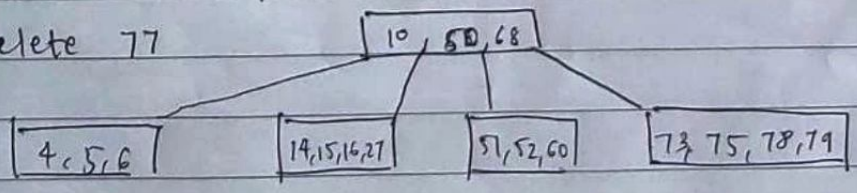
Delete 70



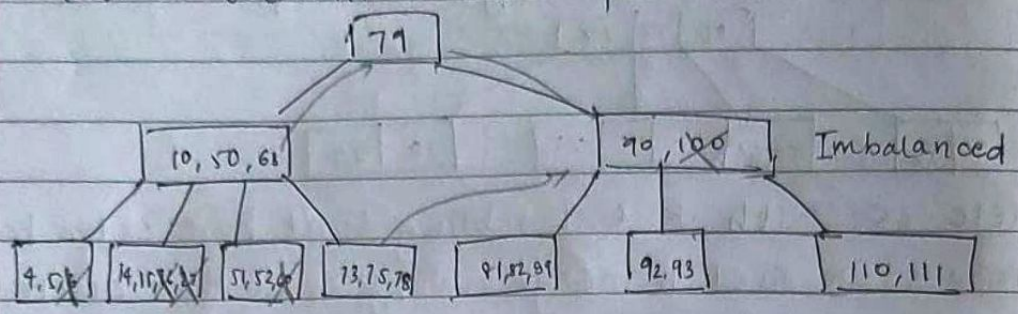
Delete 95



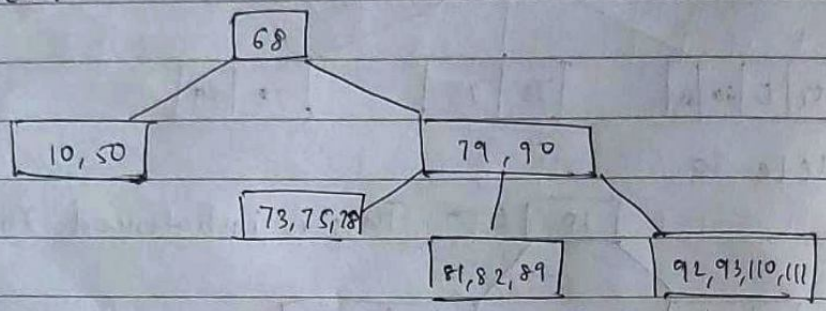
Delete 77



Delete 80. We will have to replace it with 79 or 81.

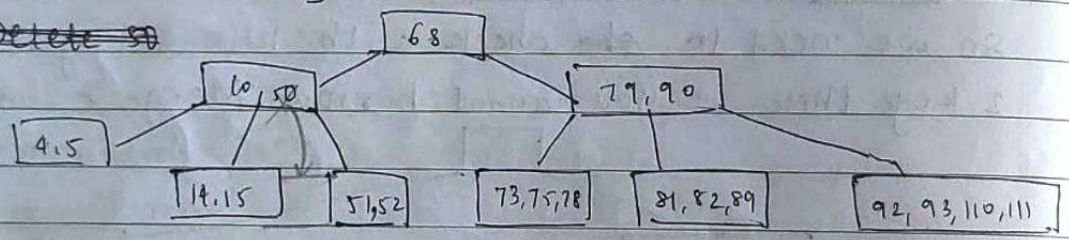


Delete 100.

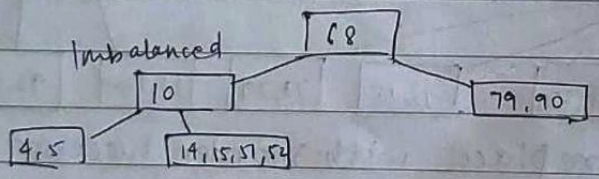


Delete 6. Directly. Delete 27. Delete 60. Delete 16

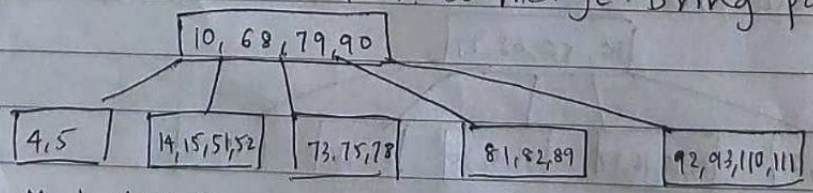
~~Delete 50~~



Delete 50



Sibling doesn't hv extra (79, 90). So merge. Bring parent (68) down

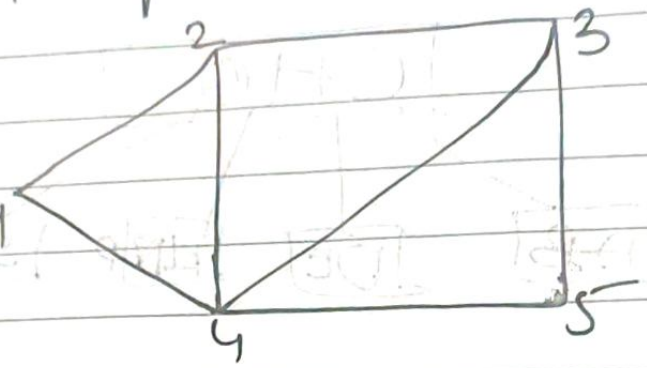


This is called shrinking of B tree as height is reduced.

Graph

↳ traversal

↳ representation



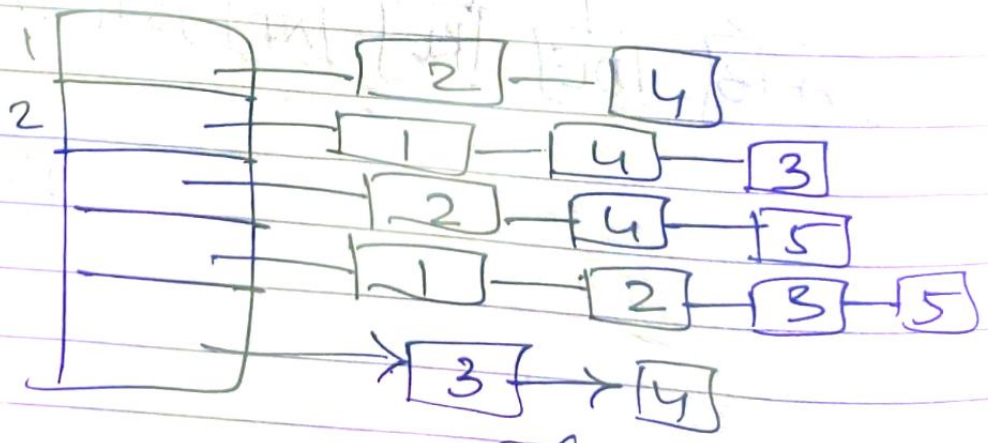
$n \times n$

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Adjacency Matrix

Space complexity $n^2 \Rightarrow \Theta(n^2)$

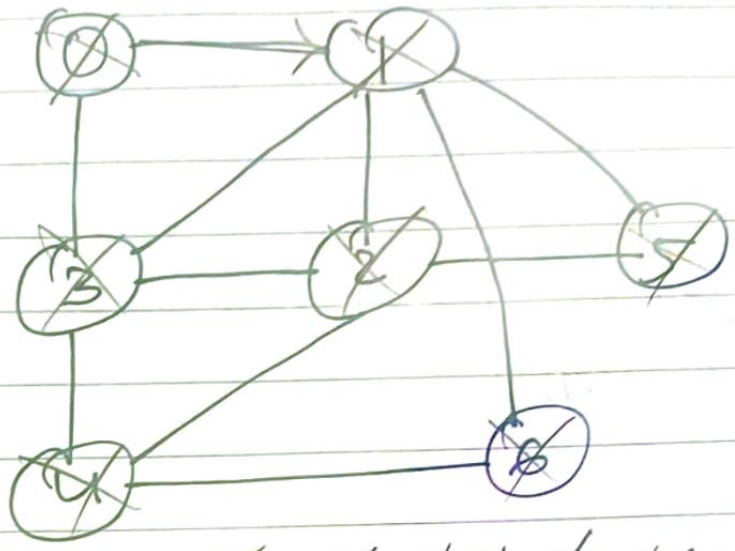
List



$\Theta(n+2e)$

BFS & DFS

BFS



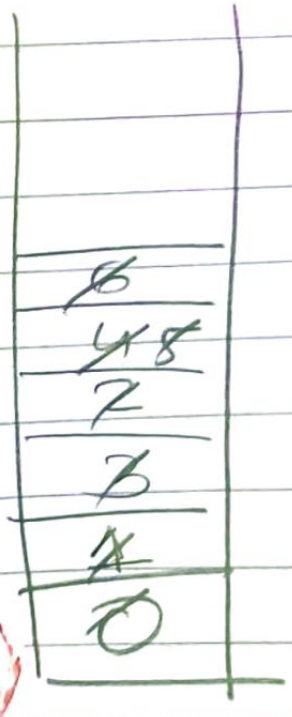
Queue: 0, 1, 3, 2, 5, 6, 4

Result:- 0132564

DFS

Stack

0132465



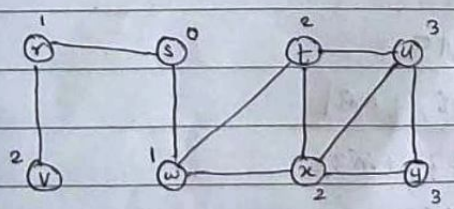
- Adjacent list requires memory = $\Theta(V+E)$
- BFS : by Queue

It discovers all vertices at distance k from s before discovering vertices at distance k+1.

- BFS (G, s) → starting node (G → graph)

```
for each vertex u ∈ V[G] - {s}
do colour [u] ← WHITE          //colour of u
   d[u] ← ∞                    // distance of s to u
   π[u] ← NIL                  // predecessor of u.
color [s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ϕ
ENQUEUE (Q, s)
while Q ≠ ϕ                    // Q is FIFO-queue
  do u ← DEQUEUE(Q)            // ~~iter~~ loops as long as gray vertices
  for each v ∈ Adj[u]          // find undiscovered adjacent vertices.
  do if color[v] = WHITE       // enqueued whenever painted gray
    then color [v] ← GRAY
         d[v] ← d[u] + 1
         π[v] ← u
         ENQUEUE (Q, v)
color [u] ← BLACK              // painted black when dequeued
```

Eg:

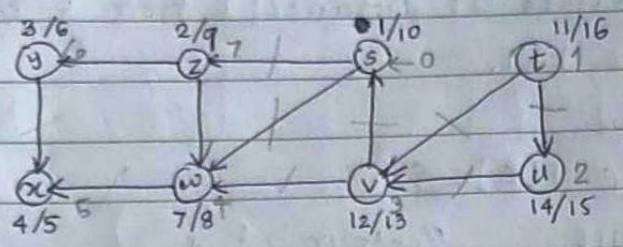


srwvtxuy

BFS will not be unique until distance is unique.

03471256

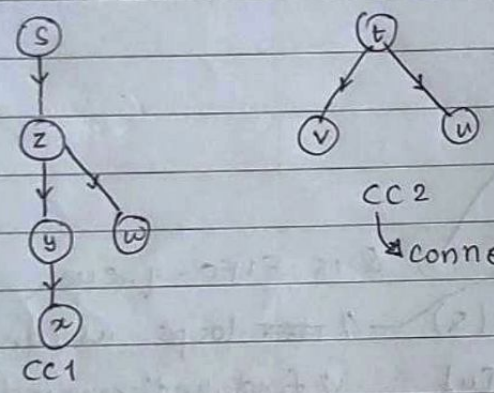
(Q) d = discovery time } For DFS d/f
f = finish time



Start with s.

Traversal: ~~szyxw zsvut~~ DFS szyxwtvu

Tree:

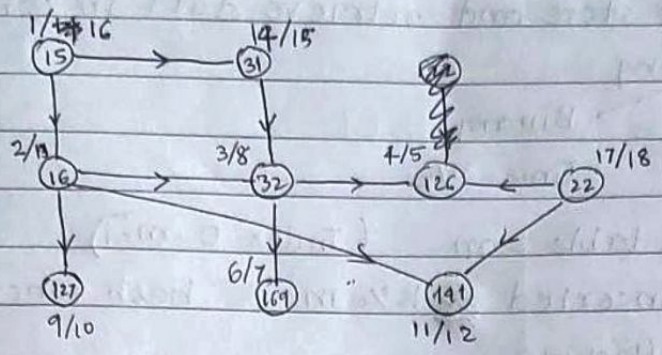


Application of DFS: To find subgraphs (CCs)

Application of BFS: To find shortest path.

(Q) Courses LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, LA169

Course	Prerequisite
LA15	–
LA16	LA15
LA22	–
LA31	LA15
LA32	LA16, LA31
LA126	LA22, LA32
LA127	LA16
LA141	LA22, LA16
LA169	LA32

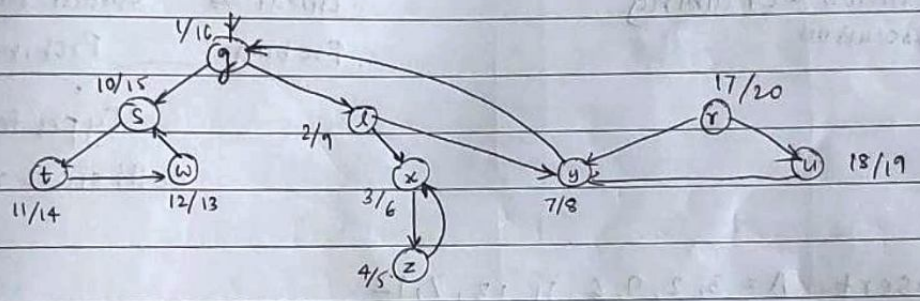


DFS : 15, 16, 32, 126, 169, 127, 141, 31, 22

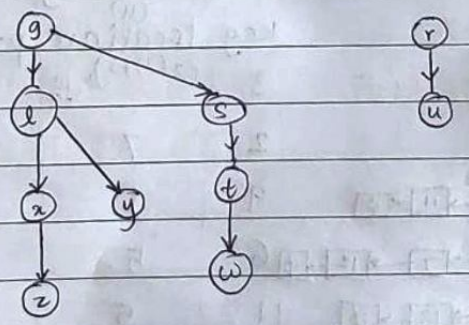
Restriction: Go in increasing order.

Traversal: 15, 16, 32, 126, 169, 127, 141, 31, 22

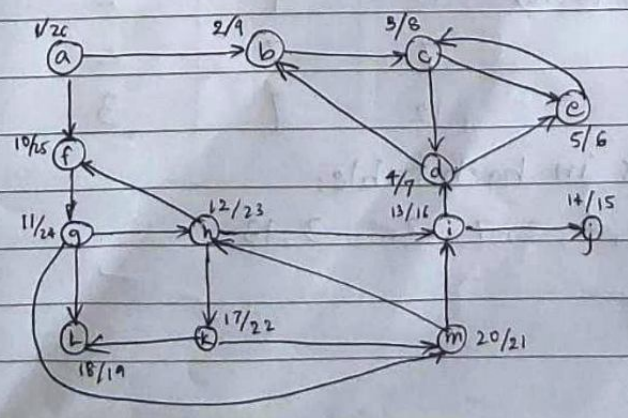
(8)



Increasing order: glxzystwru



(9)



abcdefghijklm

> HASHING: To store and retrieve data in constant time.

Searching
- linear — Time = n
- Binary — Time = n/2

Size of hash table = m (Index 0 - m-1)

∴ key to be inserted k % m ← hash function

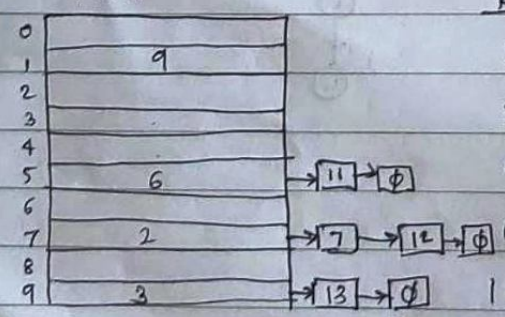
Hashing
- Open Hashing (closed addressing) → collision resolution → chaining
- Closed Hashing (open addressing) → Linear Probing, Quadratic Probing, Double Hashing — Types for collision resolution

(Q) Insert A = 3, 2, 9, 6, 11, 13, 7, 12

$h(k) = 2k+3$ $m = 10$

① Division method and closed addressing.

k % m



(Using LL within hash table)

Key	Location (u)	probes
3	$(2(3)+3)\%10 = 9$	1 (no of index needed to be s
2	7	1
9	1	1
6	5	1
11	5	2
13	9	2
7	7	2
12	7	3

} Chaining (Just like GLL)

Order of elements in hash table:

-, 9, -, -, -, 6, 11, -, 2, 7, 12, -, 3, 13

② Division method & closed hashing (Linear Probing)
(only hash table used no other data structure)

0	13
1	9
2	12
3	
4	
5	6
6	11
7	2
8	7
9	3

When there is collision, insert ki at 1st free location from $(u+i)\%m$. Refer table in ①.

Eg: For k=11, u=5, i from 0 to 9.
$(5+0)\%m = 5$ Not free
$(5+1)\%10 = 6$ Free

For 13, $(9+0)\%10 = 9$ ✗
$(9+1)\%10 = 0$ ✓ } Probe = 2

For 7, $(7+0)\%10 = 7$ ✗
$(7+1)\%10 = 8$ ✓ } Probe = 2

For 12, $(7+0)\%10 = 7$ ✗
$(7+1)\%10 = 8$ ✗
$(7+2)\%10 = 9$ ✗
$(7+3)\%10 = 0$ ✗
$(7+4)\%10 = 1$ ✗
$(7+5)\%10 = 2$ ✓ } Probe = 6

Order: 13, 9, 12, -, -, 6, 11, 2, 7, 3

③ Division method & closed hashing (Quadratic Probing)
$(u+i^2)\%m$ $i = 0$ to 9

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

For 11, $(5+0)\%10 = 5$ ✗
$(5+1)\%10 = 6$ ✓

For 13, $(9+0)\%10 = 9$ ✗
$(9+1)\%10 = 0$ ✓

For 7, $(7+0)\%10 = 7$ ✗
$(7+1)\%10 = 8$ ✓

For 12, $(7+0)\%10 = 7$ ✗
$(7+1)\%10 = 8$ ✗
$(7+4)\%10 = 1$ ✗
$(7+9)\%10 = 6$ ✗
$(7+16)\%10 = 3$ ✓ } Probe = 5

Order: 13, 9, -, 12, -, 6, 11, 2, 7, 3

④ Division method and closed hashing (Double hashing)

8, 2, 9, 6, 11, 13, 7, 12 $\quad m = 10$

$h_1(k) = 2k + 3 \qquad h_2(k) = 3k + 1$

0	13
1	9
2	
3	
4	11
5	6
6	
7	2
8	
9	3

key	location (u)	Probes	v
3	9	1	
2	7	1	
9	1	1	
6	5	1	
11	~~(2(11)+3)%10=5~~ ~~(3(11)+1)%10=4~~	~~1~~	~~4~~
13	~~(2(13)+3)%10=9~~ ~~(3(13)+1)%10=0~~	~~1~~	~~0~~
7	~~(2(7)+3)%10=~~		

For 11,

11	5	4

Formula : $(u + v*i)\%m$

For 11, $(5 + 4(0))\%10 = 5$ ✗
$(5 + 4(1))\%10 = 9$ ✗
$(5 + 4(2))\%10 = 3$ ✓ } Probes = 3

0	
1	9
2	
3	11
4	12
5	6
6	
7	2
8	
9	3

For 13, $v = 0$ ← You will get 9 no matter what so you cannot insert this value.

no probes

For 7, $u = 7$, $v = 2$

$(7 + 2(0))\%10 = 7$ ✗
$(7 + 2(1))\%10 = 9$ ✗
~~$(7 + 2(2))\%10 = 9$~~ ✗

$(7 + 2(2))\%10 = 1$ ✗
$(7 + 2(3))\%10 = 3$ ✗
$(7 + 2(4))\%10 = 5$ ✗
$(7 + 2(5))\%10 = 7$ ✗
$(7 + 2(6))\%10 = 9$

Not possible
Cannot be inserted.

For 12, $u = 7$, $v = 7$

$(7 + 7(0))\%10 = 7$ ✗
$(7 + 7(1))\%10 = 4$ ✓ } Probes = 2

(Q) 18, 41, 22, 44, 59, 32, 31, 73 $m = 13$

$h_1(k) = k \bmod 13$

$h_2(k) = 8 - (k \bmod 13)$

0	
1	
2	41
3	
4	
5	18
6	32
7	59
8	44
9	22
10	
11	31
12	

key	location (u)	v	Probes
18	5	-	1
41	2	-	1
22	9	-	1
44	5	3	2
59	7	-	1
32	6	-	1
31	5	3	3
73	8	0	Insertion not possible.

- Note: An element with key k is stored in slot h(k) and not in slot k (array's index) for open hashing.
- Linear probing ~~is has~~ takes less memory than chaining as it does not have to store all links. It is slower than chaining.