Day 1

Data structures & Algorithm

1. Basic
   - Introduction
   - Recursion
   - Algo Run Time Analysis
2. Physical DS
   - Array
   - linked list.
3. Logical DS
   - Tree
   - Hashing
   - Stack
   - Queue
   - Graph.
4. miscellaneous — Sort.
5. Algo Technique
   - magic formula
   - Greedy Algorithm.
   - Divide & conquer
   - Dynamic Programming.

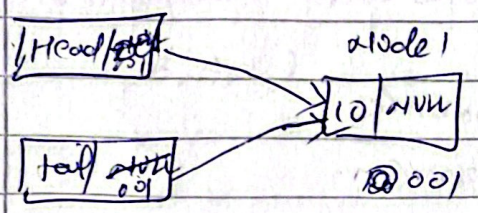
Scanned with CamScanner

Linked list

~~Joy R~~ Toy train

1. Creation of linked list
2. Insertion of linked list
3. Traversal of linked list
4. Searching in a linked list
5. Deletion of a node from a linked list
6. Deletion of linked list

Creation of Single linked list



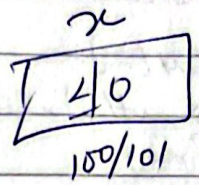
1. Create a node
2. Initialize
3. Make head & tail to node.

```
CreateSingleLinkedList(node value);
[ 1. Create a head, tail pointer & initialize with NULL;
[ 2. Create a blank node
  3. node.value = node value;
  4. node.next = null
[ 5. head = node
  6. tail = node;
```

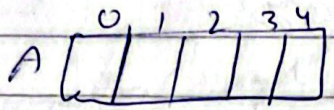
Array

(I) what is an array?
(IV) Declaring and Initilizing
(IV) ~~Accessing~~ Accessing array

global → int x=10

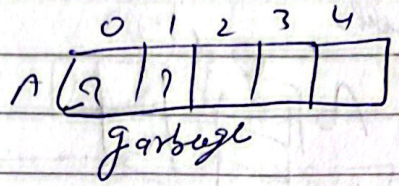


vectors → int A[5].



have dimension that why vector variable

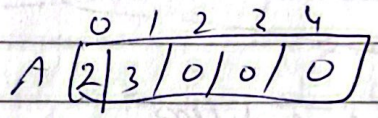
(I) Declaration int A[5];



(II) initialization

int A[5] = { 2,4,5,6 7 }; A[1 1 1 1]

(III) int A[5] = {2, 3}

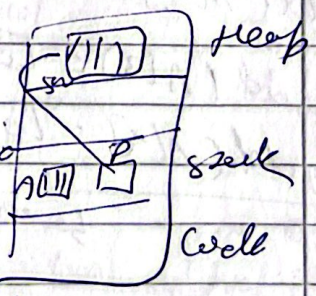


(IV) int A[5] = {0} → A[0 0 0 0 0]

(5) int A[ ] = {2,4,5,6,7}; A[2 3 4 5 6 7]

later can be add another elt then size would be updated in array.

* static VS Dynamic Array



void main() {
int A[5] — in C size would be decl only compile time
int n; but in C++ it can be decl at
cin >> n; even run time in stack.
int B[n];

int *P;
P = new int[5];

in C, P = malloc(5 * sizeof(int));
this row
but cast int then
P = (int*) malloc(5 * sizeof(int));

memory leak problem

System programming → Assembly programming

C++ delete [] p;

C free(p);

{ p[0] = 5; assign

3/

How to increase array size:

int *p = new int[2];

int *q = new int[0];

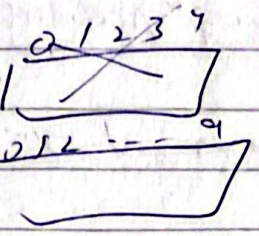
for (int i=0; i<5; i++)

q[i] = p[i] // memcopy;

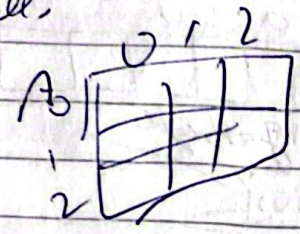
delete [] p;

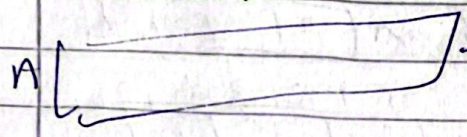
p = q;

q = null;

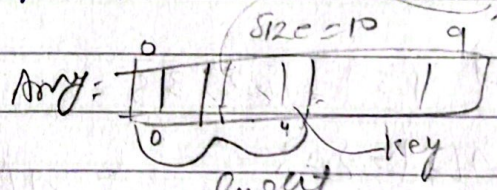


2D Arrays

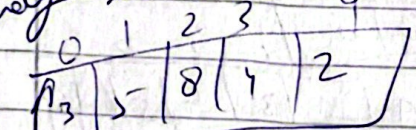




Compiler providing like this



Arrays in Compilers



add(A[3]) = 2001 + 3 × 2 = 207

add(A[3]) = Lo + 3 × 2

add(A[i]) = Lo + i × w — size of data type

base add, index

insert:

if lastIdx < size

Array[lastIdx++] = value

if index < lastIdx

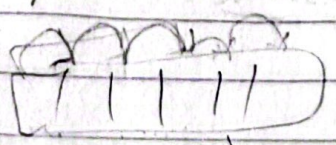
for (lastIdx; i > index; i--)

array[i] = array[i-1]

array[index] = value

lastIdx++;

linear search or sequential search



key = Array[1]

O(1) best case

O(n) worst case

avg: n/2

delete index = 6

if index < lastIdx

for (i = lastIdx; i > index; i--)

Array[index] = Array[i-1]

for (i = index; i < lastIdx - 1; i++)

Array[i] = Array[i+1]

~~Array DAT~~ Linked List.

Why Linked List?

Drawback Array: Array has fixed size.

Pointer of node type

Pointer is some node type

node

data	next

struct node {
int data;
struct node *next;
};

address 2 byte

2 byte

4 byte

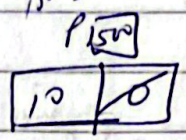
struct node *p;
p = (struct node) malloc(sizeof(struct node));

Self Referential Structure

p = new node;

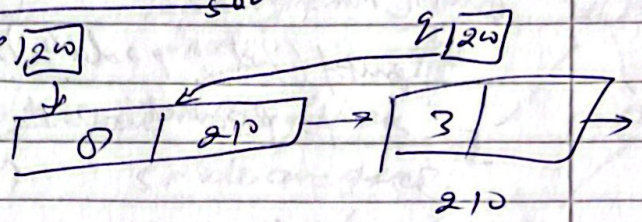
p->data = 10;

p->next = 0;



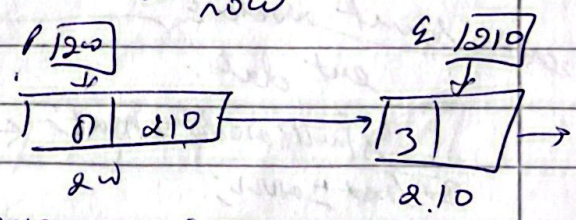
struct node *p, *q.

① ~~q = p~~ q = p

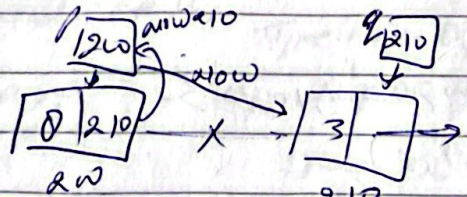


2. q = p->next.

210 want to save in q now



③ p = p->next



struct node *p = NULL

Working or not?

if (p == NULL)

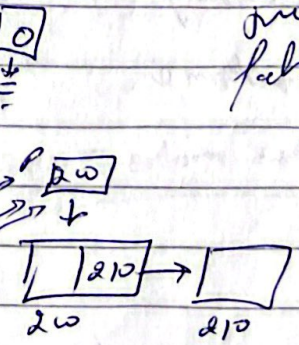
if (p == 0)

if (!p)

if (p != NULL)

if (p != 0)

if (p)



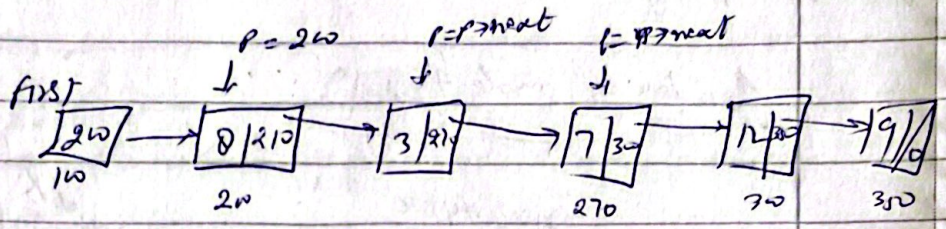
true value

false 0

if (p→next == NULL) — node not pointing anywhere

if (p→next != NULL) — Root node is not NULL.

traversing



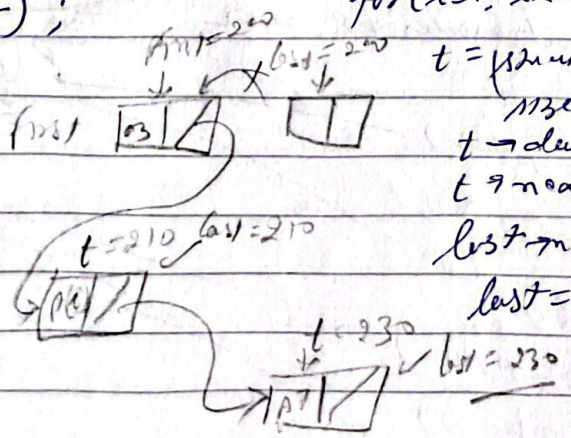

```
struct node *p = first;
while (p!=0) {
    p = p→next
}
```

```
while(p!=0) {
    printf("%d", p→data);
    p = p→next;
}
```

#

```
struct node {
    int data;
    struct node *next;
};
int main() {
    int A[] = {3,5,7,10,15};
    Create(A, 5);
    display(first);
    return 0;
}
```

```
void Create(int A[], int n) {
    int i;
    struct node *p, *last;
    first = (struct node*) malloc(
        sizeof(struct node));
    first→data = A[0];
    first→next = NULL;
    last = first;
    for(i=1; i<n; i++) {
        t = (struct node*) malloc(
            sizeof(struct node));
        t→data = A[i];
        t→next = NULL;
        last→next = t;
        last = t;
    }
```



```
void display (struct Node *p) {
    while (p != NULL) {
        printf("%d", p->data);
        p = p->next;
    }
}
```

Recursive display

```
void Display (struct Node *p) {
    if (p != NULL) {
        printf("%d", p->data);
        Display(p->next);
    }
}
```

or

```
{
Display(p->next)
Display("%d", p->data);
}
```

time → depending of no. of nodes

activation space → space/stack

O(n)

s node are space

(n+1) — best 6 activation code

activation code deleted

Counting nodes

```
int Count (struct Node *p) {
    int c = 0;
    while (Null != p) {
        c++;
        p = p->next;
    }
    return c;
}
```

time take O(n)

space complexity → depend on data used

space O(1)

or

```
if (p == 0)
    return 0;
else
    return count(p->next)+1;   // why return 1+Count(p->next)
}
```

Sum of all elements in a linked list.

```
int add(struct node *p){
    int sum=0;
    while(p!=NULL){
        sum=sum+p->data;
        p=p->next;
    }
}
```

```
int add(struct node *p){
    if(p==0)
        return(0);
    else
        return add(p->next)
            +p->data;
}
```

maximum elements in a linked list:

max = MIN_INT → Initial value

max = -32768

```
int max(node *p){
    int m=-32768; // assign MIN_INT
    while(p){
        if(p->data>m)
            m=p->data;
            p=p->next;
    }
    return m;
}
```

rewrite again recursive function.

```
x=max(p->next)
return x>p->data?x:p->data;
```

recursive function = ?

```
if(p==0)
    return MIN_INT;
else{
    x=max(p->next);
    if(x>p->data)
        return x;
    else
        return p->data;
}}
```

Searching

(I) Linear Search.

(II) Binary Search → Binary cannot use because have to traverse from first node then only use linear search

Linear Search.

```
node* Search(node *p, int key){
    while(p!=NULL){
        if(key==p->data)
            return(p)
        p=p->next;
    }
    return NULL;
}
```

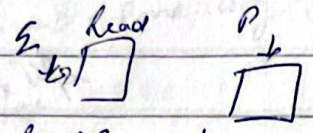
recursive function write?

Improve linear Search:

(i) transposition → Interchange previous value

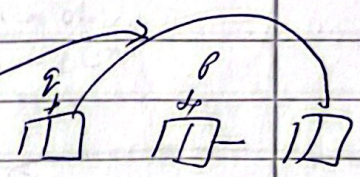
(ii) move to header → move to address nove to head.

(i) move to head

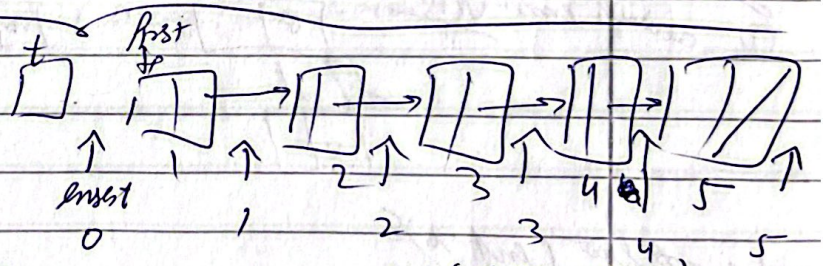


move to p & q node

```
node* search (node *p, int key) {
    node *q = NULL;
    while (p != NULL) {
        if (key == p->data) {
            q->next = p->next
            p->next = first
            first = p;
        }
        q = p;
        p = p->next;
    }
}
```



Inserting in a linked list



(i) Insert before first

(ii) Inserting after given position

(i)

(i) Create a new node

(ii) initialize data

(iii) make a point on first.

```
void Insert (int pos, int x)
{ if (pos == node *t, *p)
    node *t = new node;
    t->data = x;
    t->next = first;
    first = t; else if (pos==0)
```

(ii) after position 4

(i) create a new node

(ii) initialize data.

(iii)

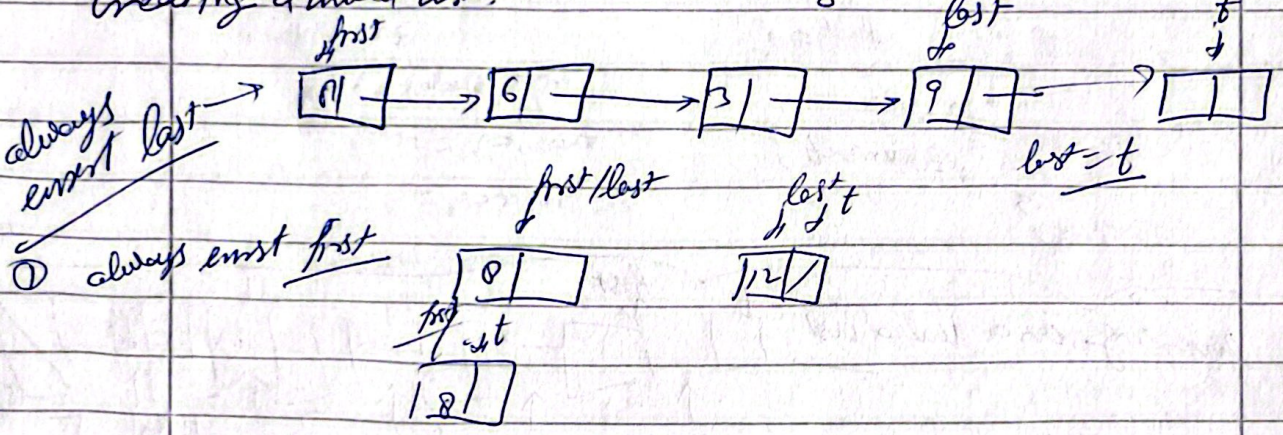
```
else if (pos>0)
    node *t = new node
    t->data = x;
    p = first
    for (i=0; i<pos-1; i++) {
        p = p->next
    }
    t->next = p->next
    p->next = t;
```

} if work on last node also.

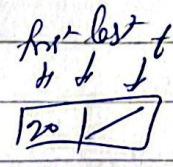
two extra pointers required

```
void Insert(int pos, int x) {
    node *t, *p;
    if (pos==0) {
        t= new node;
        t->data = x;
        t->next = first;
        first = t;
    }
    else if (pos>0) {
        p= first;
        for(i=0; i<pos-1; i++)
            p=p->next;
        if(p) {
            t= new node;
            t->data =x;
            t->next =p->next;
            p->next = t;
        }
    }
}
```

Creating a linked list:

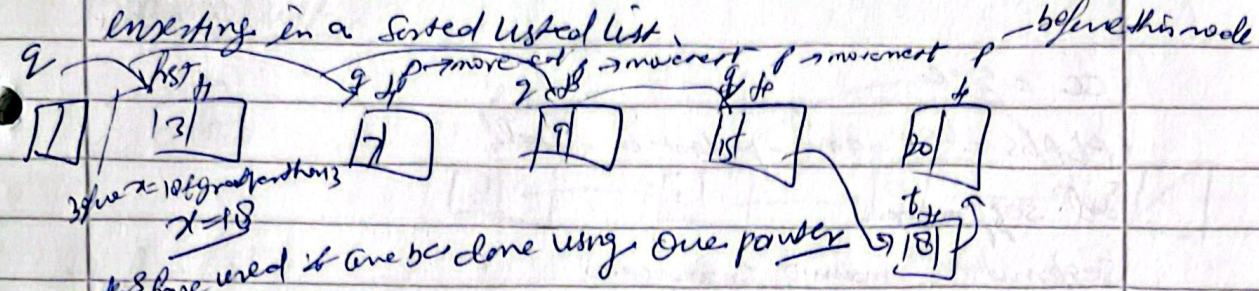



```
void insert last (int x){
    node *t = new node;
    t->data = x;
    t->next =NULL;
    if(first==NULL)
        first =last = t;
    else {
        last->next = t;
        last= t;
    }
}
```



insertlast(3);
(5)
(7)

Inserting in a Sorted Listed list.



two pointers have used it can be done using one pointer

```
p = first
q = NULL
while (p->data < x) {
    q = p;
    p = p->next
}
t = newNode
t->data = x;
t->next = q->next;
q->next = t;
```

for with one pointer only with p

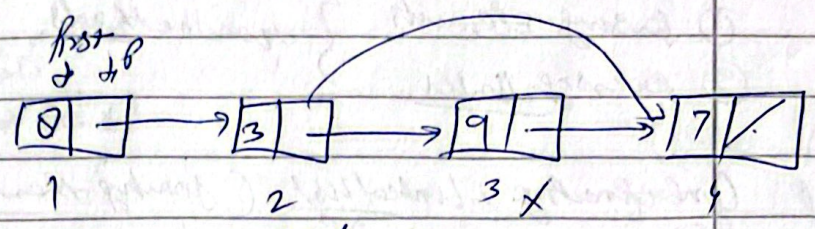
min → O(1)
max – O(n)

```
void SortedInserted (Struct node *p, int x) {
    struct node *t, *q = NULL;
    t = (struct node *) malloc (sizeof (struct node));
    t->data = x
    t->next = NULL;
    if (first == NULL)
        first = t;
    else {
        while (p->data < x) {
            q = p;
            p = p->next;
        }
        if (p == first) {
            t->next = first;
            first = t;
        }
        else {
            t->next = q->next;
            q->next = t;
        }
    }
}
```

```
CC = gcc
CFLAGS = -g3 -ansi -pedantic -Wall
all: seqpmtr
seqpmtr: main.o sequence.o
        $(CC) $(CFLAGS) -o $@ $^
test: seqpmtr
        ./seqpmtr
main.o: main.c sequence.h
sequence.o: sequence.c sequence.h
clean:
        $(RM) -f seqpmtr *.o
```

Deleting list:

new
delete



delete from first

```
node *p = first
first = first->next
x = p->data
delete p;
```

deleting from given position:

p = move(k-1) posing.

```
node *p = first.
node *q = NULL;
for (i=0; i<pos; i++) {
    q = p;
    p = p->next;
}
q->next = p->next
x = p->data
delete p;
```

~~Is#listc~~ list is Sorted?

```
int x = -32768
node *p = first
while(p!=NULL) {
    if (p->data < x)
        return false;
    x = p->data
    p = p->next
}
return true;
```

if greater than continue

remove Duplicates from sorted list.

first p q

3 5 5 8 8 8

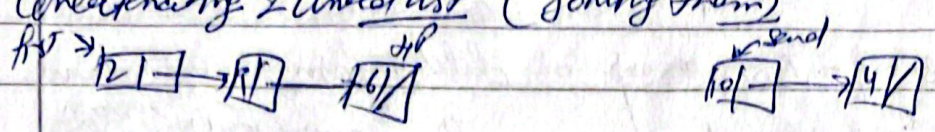
```
node *p = first
node *q = first->next
if (p->data != q->data)
{ p = q
  q = q->next;
} else {
    p->next = q->next
    delete q;
    q = p->next;
```

Reversing a linked list

① forming elements
② Reversing links.

} two methods

#1 Concatenating 2 linked list (joining them)




```
p=first
while(p->next != NULL){
    p = p->next;
}p->next = second
```

#1 merging of two linked list (merge two sorted list)

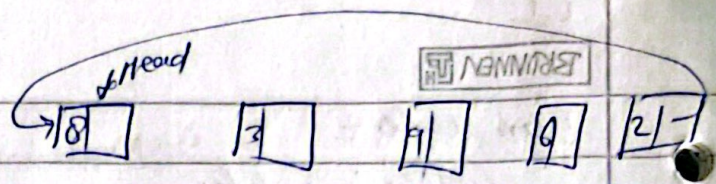
#1 LOOP , linear & loop
↓ NULL at end
— circular

Scanned with CamScanner

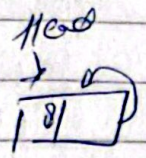
CPP

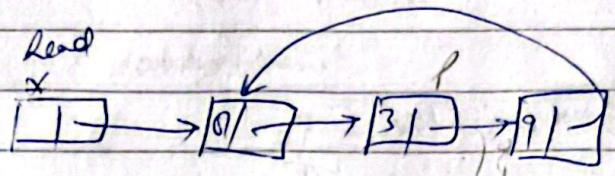
```
class node {
    public: int data
            node *next;
}

class LinkedList {                    Implementation class
    private: node *first;

    public:
        LinkedList() { first=NULL; }
        LinkedList(int A[], int n);
        ~LinkedList();
        void Display()
```

Circular Linked list



if one node





p->t->next = p->next

Display Circular linked list

Here in last node don't have NULL.

```
void Display( node *p )   Head pointer
{
    while( p != Head )
    {
        print("%d", p->data
        p = p->next;
    }
    do{
    }while(p != Head);
}
```

we do while loop

Inserting in a Circular linked list:

```
void Insert(int pos; int x ){
node *t, *p;
int i;
if (pos == 0){
    t = new node;
    t->data = x;
    if( Head == NULL){
        Head = t;
        Head->next = Head
    }else{
        p = Head;
        while(p->next != Head)
            p = p->next.
        p->next = t;
        t->next = Head;
    }
    Head = t;
}
```

before Head node

```
node need to move
Head to before
node *p = Head
Node *t = new node;
t->data = x;
t->next = Head
while(p->next != Head){
    p = p->next
}
p->next = t
Head = t
```

first node marked Head

Position

```
Node *t;
Node *p = Head
for(i=0; i<pos-1; i++)
    p = p->next
t = new node
t->data = x;
t->next = p->next
p->next = t;
```

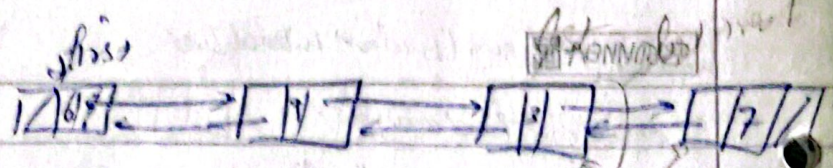
Deleting from Circular linked list.

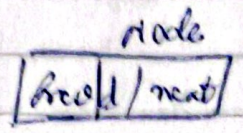
i) deleting Head node.

ii) deleting a node from given position.

Scanned with CamScanner

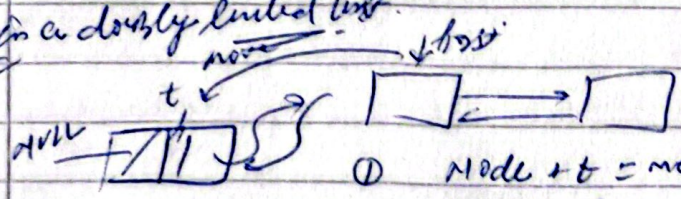
Doubly linked list:

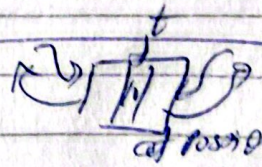





```
struct node {
    struct node * prev
    int data
    struct node * next;
};

struct node *t
t = new node;
t = prev = NULL
t->data = 10;
t->next = NULL;

display( struct node *p ){
while(p) {
    put
    p=p->next;
}

create( ){
    struct node *t,*last;
    int i;
    first = new node;
    first->data = A[0];
    first->prev = first->next = NULL;
    last = first;
    for (i=1; i<n; i++){
    t = new node;
    ~~t->data[i]~~ t->data = A[i];
    t->next = last->next
    t->prev = last
    last->next = t;
    last = t;
```

Insert in a doubly linked list:





(i) Node *t = new node
(ii) t->data = x
(iii) t->prev = NULL
(iv) t->next = first
(v) first->prev = t
(6) first = t;

```
node *t = new node
t->data = x;
for(i=0; i<pos-1; i++)
    p = p->next;
t->next = p->next;
t->prev = p;
if (p->next)
    p->next->prev = t
p->next = t;
```

Deleting doubly linked list.

Iterate doubly list

Circular doubly list

Scanned with CamScanner

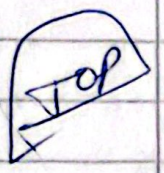
Stack

① Introduction/Application

② ADT stack.

Recursion → Stack, Iteration

LIFO

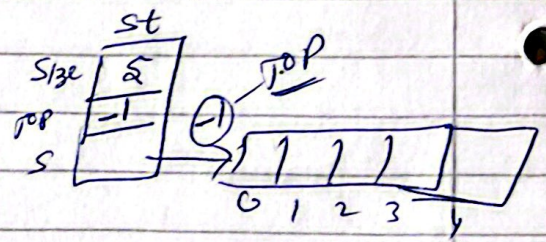


Stack, space for storing elements
+ TOP ~~pointer~~ pointer

operation
push
pop
peek(index)
stack(TOP)
isEmpty()
isFull()

Array Stack

size, TOP, Array




```
struct stack {
    int size;
    int TOP;
    int *s;  ---> // decide size at run time
}
int main() {
    struct stack st;

    st.size = 5;
    st.s = new int[st.size];
    st.TOP = -1;
void push(stack *st, int x) {
    if (st->TOP == st->size - 1)
        overflow
    else {
        st->TOP++;
        st->s[st->top] = x;
    }
int pop(stack *st) {
    int x = -1
    if (st->TOP == -1) {
        "stack underflow"
    else {
        x = st->s[st->top];
        st->top--;
    }
    return x;
```

if TOP = -1 ⇒ means stack is empty

stack full = [size - 1]

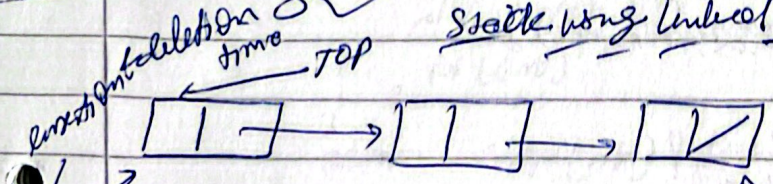
Stack underflow if delete stack after [-1] = top

Constant time for push, & popping

```
peek (stack st, int pos) {
    int x=-1;
    if (TOP-pos+1<0) {
        "Invalid position"
    else
        x = st.s[st.TOP-pos+1];

    return x;
}

int stackTOP (stack st) {
    if (st.TOP==-1)
        return -1;
    else
        return st.s[st.TOP];

int isEmpty (stack st) {
    if (st.TOP==-1)
        return 1;
    else
        return 0;
}

int isfull (stack st)
    if (st.TOP==st.size-1)
        return 1;
    else return 0;
}
```

Pos	Index = TOP-pos+1	
1	3	TOP-pos+1
2	2	3-1+1=3
3	1	3-2+1=2
4	0	3-3+1=1
		3-4+1=0

Stack using Linked list



Node

data	next

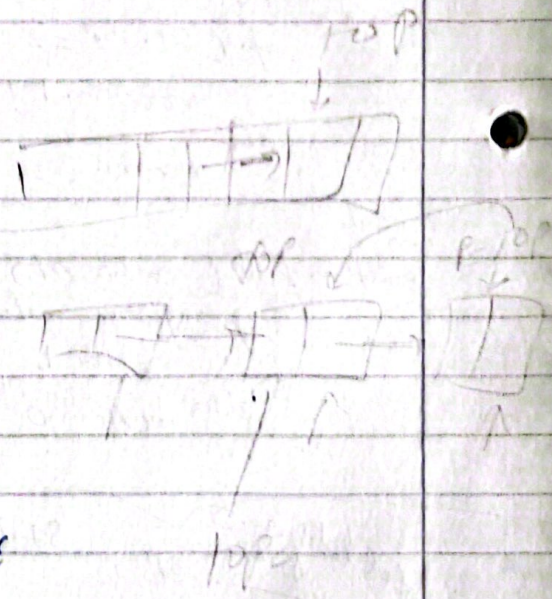
```
struct Node {
    int data;
    struct Node * next;
}
```

insert at first & make TOP as first

Empty if (TOP == NULL)

full node *t = new Node
    if (t==NULL)

```
void push (int x) {
    Node *t = new Node
    if(t == NULL)
        "Stack overflow"
    else
        t->data = x
        t->next = TOP
    }   TOP = t;

int POP ( ) {
    node *p;
    int x=-1;
    if (TOP == NULL)
        Stack is empty
    else {
        p = TOP
        TOP=TOP->next;
        x = p->data;
    }   delete p;
    return x;
int peek( int pos) {
    int x=-1
    Node *p=TOP;
    for (i=0; p!=NULL && i<pos-1; i++) {
    }   p=p->next;
    if (p!=NULL)
        return p->data;
    else
        return -1;
    }
int stackTOP() {
    if (TOP) {
        return TOP->data;
        return -1
    }
}
int isEmpty() {
    return TOP ? 0 : 1;

isFull() {
    Node *t = new Node
    int r=t ? 1:0;
        delete t;
    }   return r;
```



) (→ called paranthesis

BOMPS

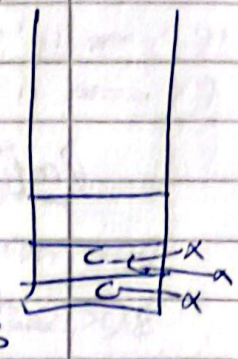
. Parenthesis Matching

((a+b) * (c-d))

using stack to check paranthesis in balance or not

respond if have open or close bracket otherwise ignore.

operand like a+b should be parenthesis



don't check parenthesis are proper or not only check matching

exp | (| a | + | b |) | * | (| c | - | d |) |) | \0 |

int isBalance (char *exp) {

stack st;

st.size = strlen(exp);

st.top = -1;

st.s = new char [st.size];

for (i=0; exp[i] != '\0'; i++) {

if (exp[i] == '(') push (&st, exp[i]);

else if (exp[i] == ')') {

if (isempty(st)) return false;

pop (&st);

}

return isempty(st) ? true : false;

}

Infix to postfix Conversion

(i) infix → a+b — operand operator operand

(ii) prefix → +ab — operator operand operand

(iii) postfix → ab+ → operand operand operator

operator precedence in order so that is can stack in order

8 + 3 * (9 - 6) / 2^2 + 6/2

6 5 1 3 2 4

8 3 9 6 - 2 2 ^ / * + 6 2 / +

1 2 3

Symbol	Precedence
+ -	1
* /	2
( )	3

Precedence table in programming

make first fully parenthesis other compiler make it

⇒ Put higher Precedence in the brackets

→ a+b*c

(a+(b*c)) → Compiler makes this parenthesis

a+b+c*d

↑ first put in brackets

Prefix

(a+*bc)

+a*bc

Postfix (a+bc*)

abc*+

Associativity & unary

a+b+c-d - Left to Right

same

(((a+b)+c)-d)

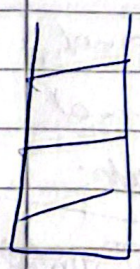
Same Precedence work Right to left

Symbol	Pre	Asso
+,-	1	L-R
*,/	2	L-R
^	3	R-L
- (Unary minus)	4	R-L
( )	5	L-R

infix to postfix using stack method

a+b*c-d/e

Sym	Pre	Asso
+-	1	L-R
*,/	2	L-R



Postfix abc*+de/-

term in stack greater than in stack of higher don't put ← of operator

+-	1	L-R
*,/	2	L-R
a,b,c	3	L-R

going to operand to push into stack full

FIFO

~~Array~~ Queue (of logical data structure)

→ insertion from one end deletion is from another end.
Front Rear

Queue ADT

Data.
1. space for deletion
2. front - for deletion
3. Rear for insertion.

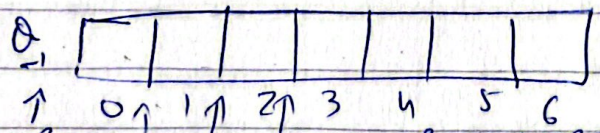
Operation.
1. enqueue(x)
2. dequeue()
3. isEmpty()
4. isFull()
5. first()
6. last()

① Implementation by — 1. Array
2. Linked List.

Queue using Array.

① Queue using single pointer
② Queue " front & Rear
(iii) Drawbacks of Queue using Array

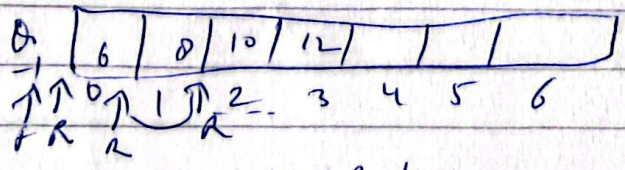
size=7



Rear R R R-always move Rear to next

Insertion - O(1)
deletion - O(n) — drawback

(ii)



insertion by R pointer
deletion move R & take out the element.

Counter move to next position then in queue
Implement ~~queue~~ queue.

~~For~~ Initially

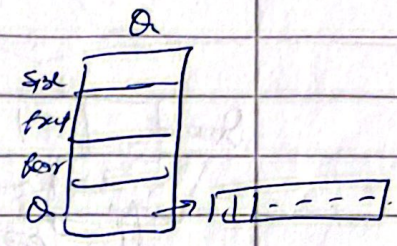
front = Rear = -1

enque — O(1)
deque — O(1)

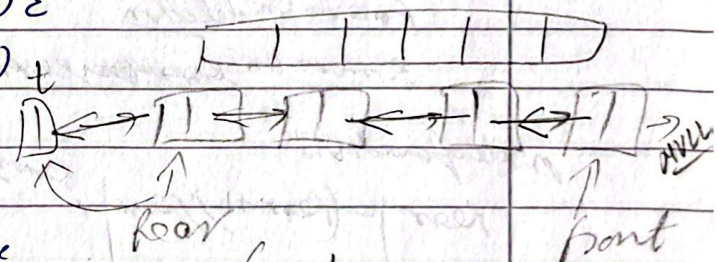
Empty
if (front == Rear)

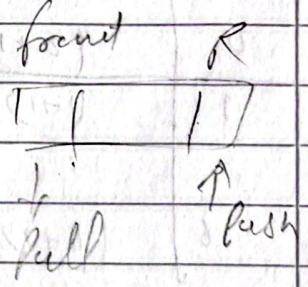
full
if (Rear == size -1)

```
struct Queue {
    int size;
    int front;
    int Rear;
    int *Q
};
```

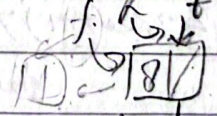



```
int main() {
    struct Queue q;
    q.size = 7;
q.Q = new ~~int[2]~~ int[size]
void enque (Queue *q, int x) {
    if (q->Rear == q->size-1)
        Queue is full
    else {
        q->Rear++;
        q->Q[q->Rear] = x
}
void deque (Queue *q) {
    int x = -1;
    if (q->front == q->Rear)
        Queue is Empty
    else {
        q->front++
        x = q->Q[q->front];
    }
    return x;
}
```





empty = front & Rear both null



full → t = new
if (t == null)
// memory full then full

~~Drawback of~~ Drawback of Queue

Solution ① Resetting pointer
② circular Queue.

Q | | | | 8 | 10 | 6 | 9 |

0 1 2 3 4 5 6

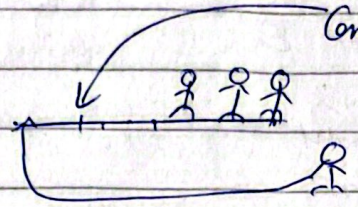
F R

> Cannot reuse the space of deleted elements.

forcing Buffer whenever $\boxed{F=R}$ does not give guarantee

(11) Circular Queue.

Confining Queue from Back.



Q | | | | | | | |

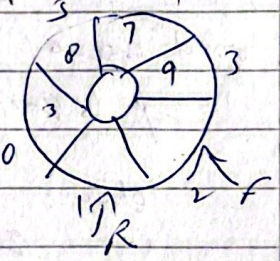
0 1 2 3 4 5 6 7

F R enqu R

Start from 0 location

F move on deletion

R move on ~~insertion~~ insertion.



node function.

Rear	= (Rear+1)%size	Remainder
0	(0+1)%7	1
1	(1+1)%7	2
⋮	⋮	⋮
6	(6+1)%7	0

R = (rear+1)%size = 1

Only 1 queue start at 0, 1, 2

if (q→rear+1)%q→size == q→front

full

else

q→rear = (q→rear+1)%q→size

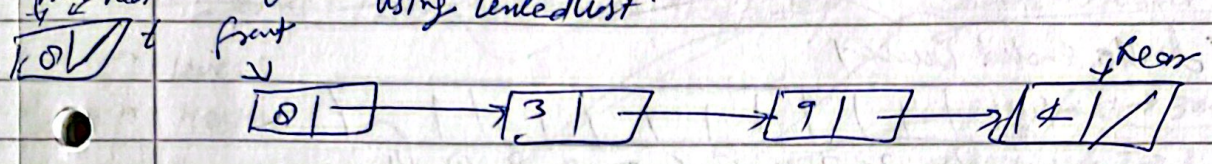
q→arr[q→rear] = x;

```
void enqueue (struct Queue *q, int x)
{
    if (q->Rear+1)%q->size == q->front)
        Queue is full
    else {
        q->Rear = (q->Rear+1)%q->size;
        q->Q[q->Rear] = x;
    }
}
```

```
int dequeue (struct Queue *q) {
    int x=-1;
    if ( q->front == q->Rear)
        Queue is Empty
    else {
        q->front =(q->front +1) % q->size;
        x= q->Q[q->front];
    }
    return x;
}
```

Priority Queue

Queue using linkedlist:



Empty
if (front == NULL)

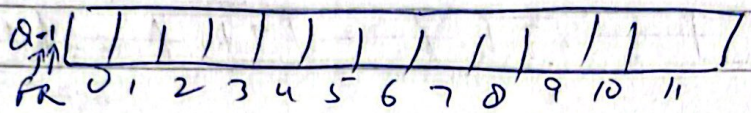
full
node *t = new node
if (t == NULL)

```
void enque (int x) {
    Node *t = new node;
    if (t== NULL)
        Queue is full
    else {
        t->data = x;
        t->next = NULL;
        if (front == NULL) front=Rear=t    // if it first node
        else {
            Rear->next = t;
            Rear = t;
        }
    }
}
```

```
int dequeue() {
    int x=-1;
    Node *p;
    if (front==NULL) {
        Empty;
    else {  p= front;
            front = front->next;
            x=p->data;
            delete p;
        }
    return x;
}
```

/double Ended Queue/

Deq



in Deq Can we both pointer for deletion as well as insertion.

NOT FIFO in Conventional But Can we FIFO.

① input restricted dequeue.

Input Restrict	Insert	Delete
front	✗	✓
Rear	✓	✓

O/P	restrict.	
front	✓	✓
Rear	✓	✗

Priority Queues

Priority-Based Scheduling.

① limited set of Priorities

② Element Priority.

A 1, B 1, C 2, D 3, E 2, F 1, G 2, H 3, I 2, J 2

Q1 | A | B | F |

Q2 | C | E | G | I | J |

Q3 | D | H |

starvation — otherwise

(ii) Element steeff Priority

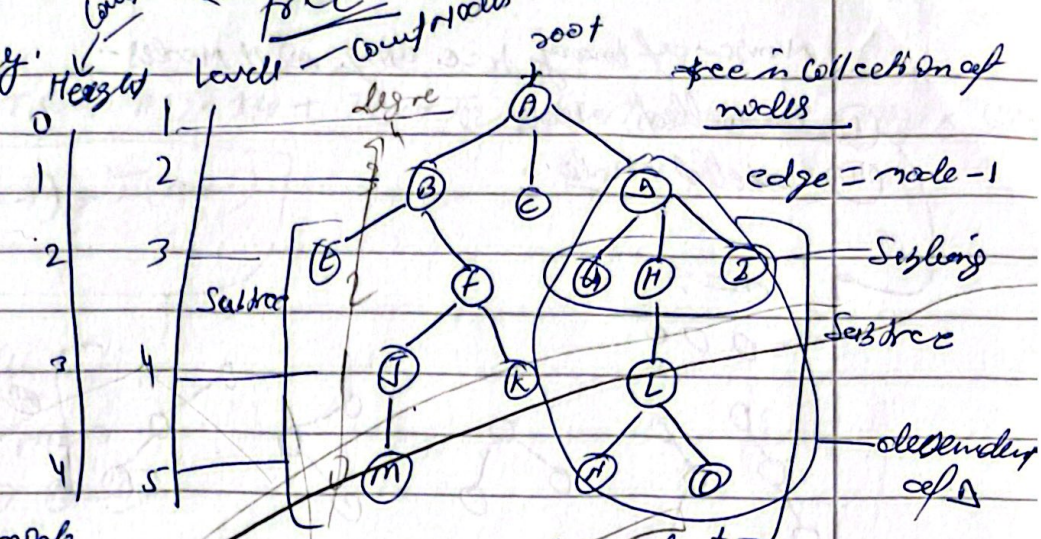
6, 8, 3, 10

Queue using 2 stacks.

Terminology

Height	Level
0	1
1	2
2	3
3	4
4	5

- Root
- Parent
- Child
- Siblings
- Descendents
- Ancestors
- Degree of a node
- Internal / External nodes · leaf / non leaf
- Levels
- Height
- Forest



tree is collection of nodes

edge = node − 1

Siblings

Subtree

descendents of D

descendents
diagonal subset

degree of A → zero
degree of J → 1
degree of D → 3
degree of O → Internal nodes

Collection of tree is forest.

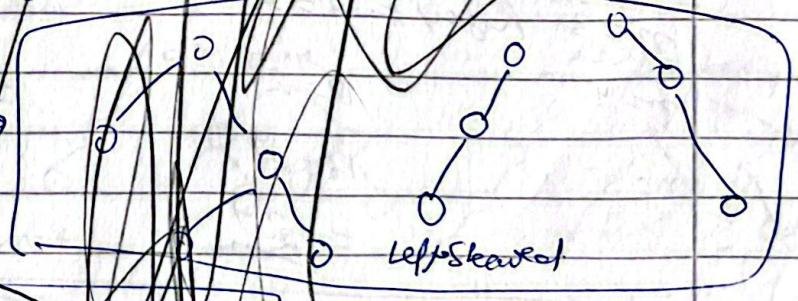
Binary tree

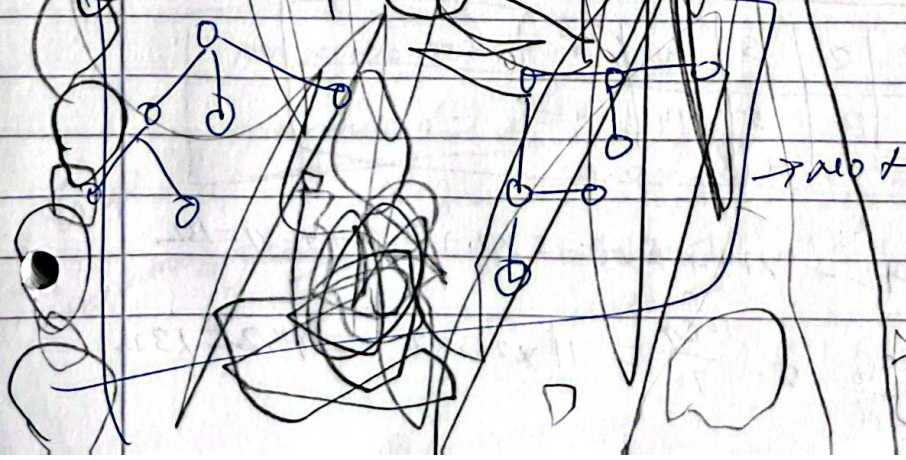
→ degree of two

degree must be two → degree = 2

Checker Condition {0, 1, 2} not more than 2.

Binary tree →





→ not binary tree

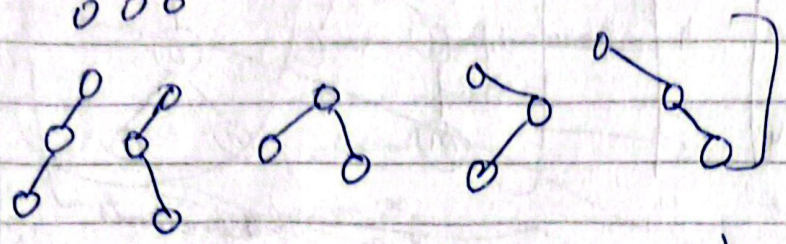
Number of binary tree using n nodes..

(i) Unlabelled node

(ii) Labelled node

$n=3$

o o o



max of tree with max height $= 4 = 2^2$

$T(3) = 5$ (5 different shape tree)

node $= 4$

o o o o

left skewed

$T(4) = 14$

$T(5) =$

$T(n) = \frac{2nC_n}{n+1}$ formula - Catalan Number

$T(5) = \frac{2 \times 5C_5}{5+1} = \frac{10C_5}{6} = \frac{\frac{10 \times 9 \times 8 \times 7 \times 6}{5 \times 4 \times 3 \times 2 \times 1}}{6} = 42$

$n=3$ height $= 4 = 2^2$

$n=4$ $8 = 2^3$

$n=5$ $16 = 2^4$

$= 2^{n-1}$

$T(3) = \frac{2 \times 3C_3}{3+1} = \frac{6C_3}{4}$

$= \frac{\frac{6 \times 5 \times 4}{3 \times 2 \times 1}}{4} = 5$

n	0	1	2	3	4	5	6
$T(n)$ $\frac{2nC_n}{n+1}$	1	1	2	5	14	42	

$T(6) = 1 \times 42 + 1 \times 14 + 2 \times 5 + 5 \times 2 + 14 \times 1 + 42 \times 1 = 132$

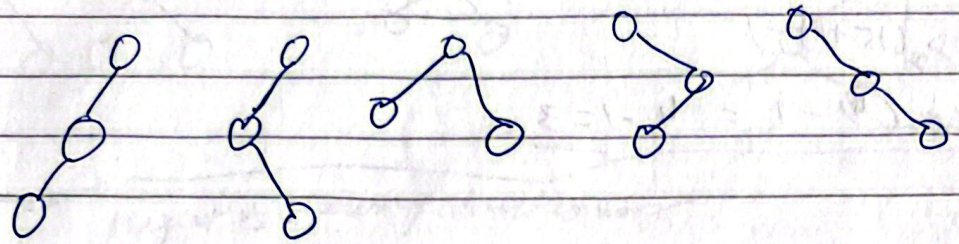
$T(6) = \frac{2 \times 6C_6}{6+1} = \frac{12C_7}{7} = 11 \times 2 \times 3 \times 2 = 11 \times 12 = 132$

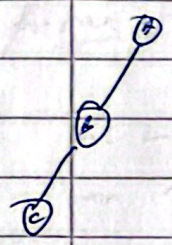
$T(6) = T(0) * T(5) + T(1) \times T(4) + T(2) \times T(3) + T(3) \times T(2) + T(4) \times T(1) + T(5) \times T(0)$

$$\boxed{T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i)}$$

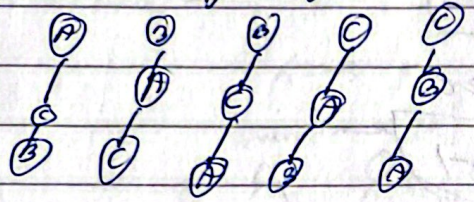
Levelled Nodes

$n=3$





levelled hierarchy way



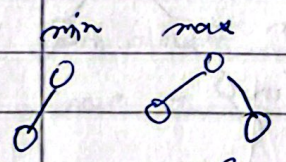
$$T(n) = \frac{{}^{2n}C_n}{n+1} \times n!$$

↑ finance ↑ filling on level

$6 = 3! = n!$

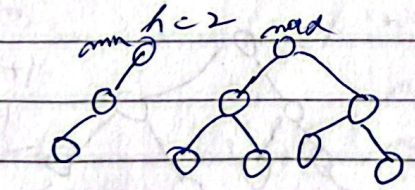
Height Vs Nodes

Height $h=1$



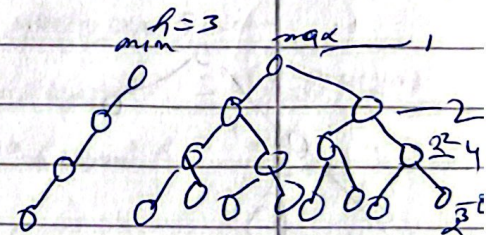
$n=2$ $n=3$

Height $h=2$



$n=3$ $n=7$

Height $h=3$



$n=4$ $n=15$

$\text{max} = 1+2+2^2+2^3 = 15$

$2^{3+1}-1 = 16-1 = 15$

$$\boxed{\begin{array}{l} \text{min nodes } n = h+1 \\ \text{max node } n = 2^{h+1}-1 \end{array}}$$

G.P. Series $a + ar + ar^2 + ar^3 + \cdots + ar^k = \frac{a(r^{k+1}-1)}{r-1}$

geometric progression

$1+2+2^2+2^3+\cdots+2^h$

$a=1 \quad r=2$

$= \frac{1 \cdot (2^{h+1}-1)}{2-1}$

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

if we have given

max height $= h = n-1$

min " $= \cancel{h = n-1}$ $\Rightarrow n = 2^{h+1} - 1$

$\Rightarrow n+1 = 2^{h+1}$

$\Rightarrow h+1 = \log_2(n+1)$

$h = \log_2(n+1) - 1$

$h = \log_2(15+1) - 1$

$= \log_2(4) - 1 = 4 - 1 = \underline{3}$

No. of nodes in a Binary Tree

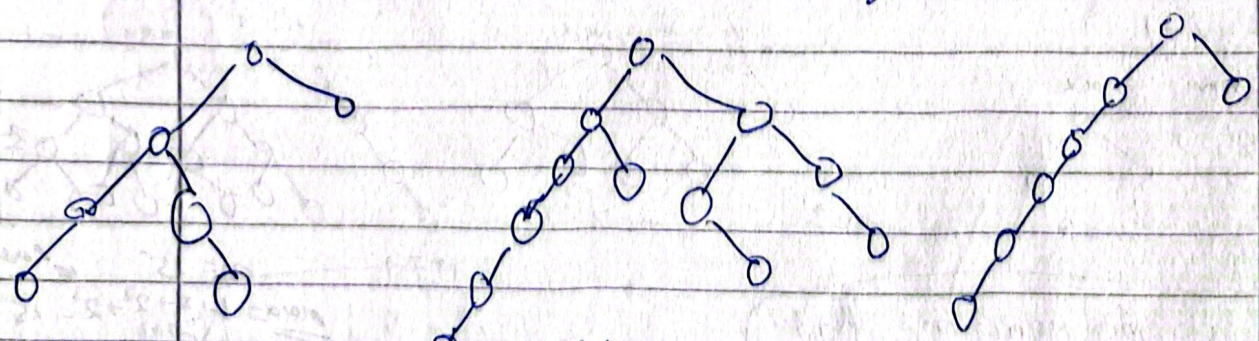
$$\boxed{h+1 \le n \le 2^{h+1} - 1}$$

Height of Binary Tree.

$$\boxed{\log_2(n+1) - 1 \le h \le n-1}$$

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

Internal nodes vs External nodes in Binary Tree.



$\deg(R)$

$\deg(P)$

$\deg(Y) = 2$

$\deg(Q) = 3$

Internal node $\begin{cases} \deg(P) = 3 \\ \deg(Y) = 5 \end{cases}$

$\deg(Q) = 4$

$\deg(Q) = 1$

$\deg(Y) = 4$

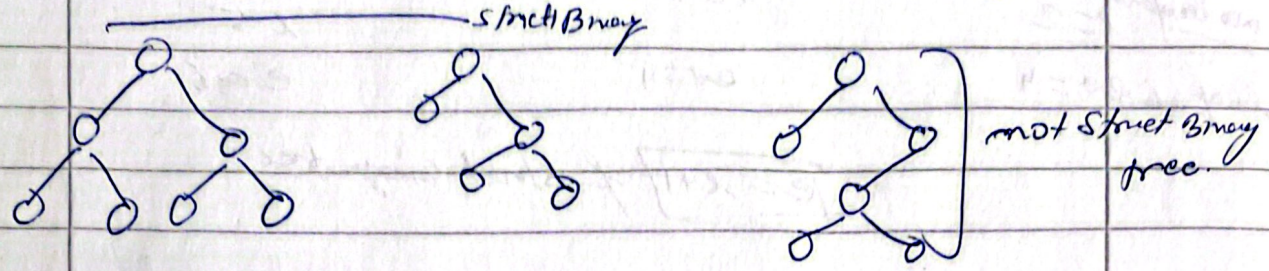
$\deg(Q) = 2$

$$\boxed{\deg(v) = \deg(2) + 1}$$

Strict Binary Tree

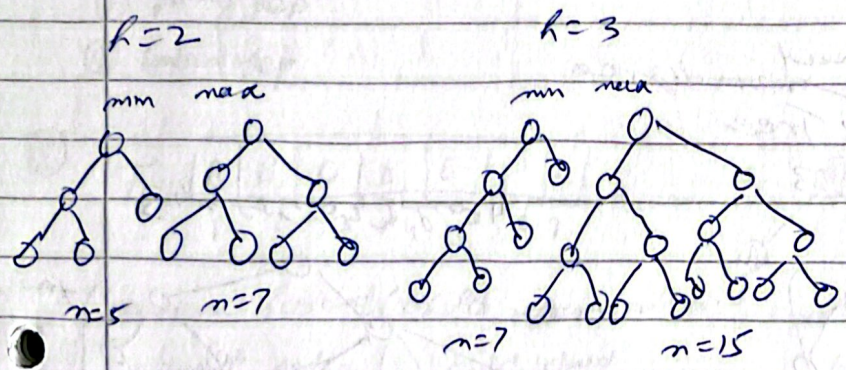
1. Strict / Proper / Complete
2. Ht Height vs Node
3. Internal vs External Nodes

Strict → mean Node can not have ~~a node~~ degree 1 only have 0 or 2 degree.



Height vs Nodes of Strict Binary

for max it perfect is have strict



if Height 'h' is given

min Node $n = 2h+1$

max " $n = 2^{h+1} - 1$

if 'n' Node are given

min Height $h = \log_2(n+1) - 1$

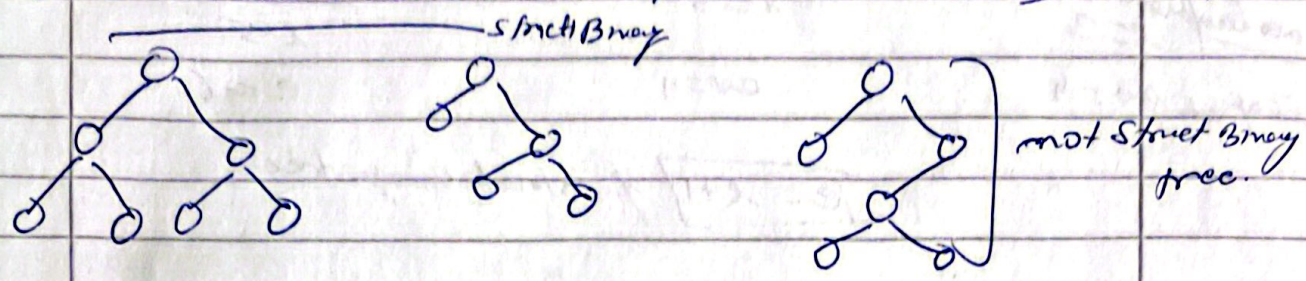
max Height $h = \frac{n-1}{2}$

$$\log_2(n+1) \le h \le \frac{n-1}{2}$$

Strict Binary Tree

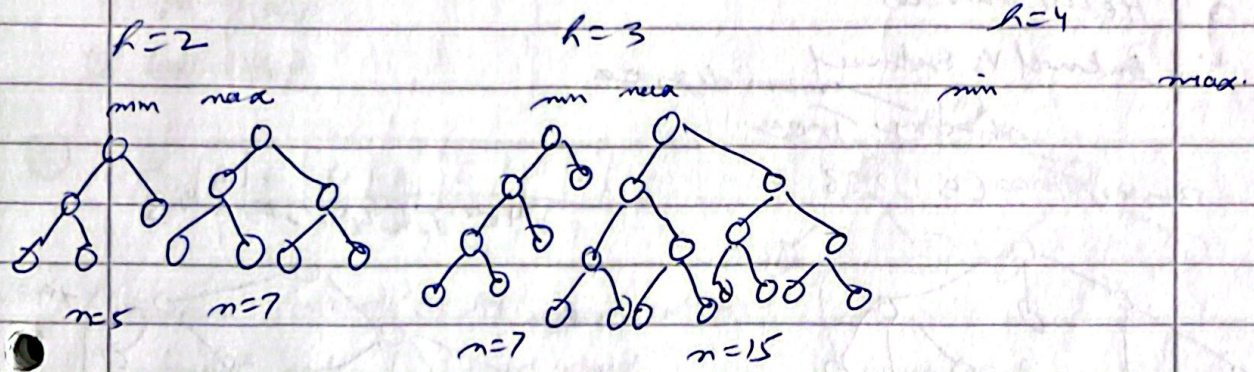
1. Strict / Proper / Complete
2. Height vs Node
3. Internal vs External Nodes

Strict → mean Node can not have ~~A~~ degree 1 ~~node~~ only have 0 or 2 degree.



Height vs Nodes of Strict Binary

for max it perfect is have strict



if height 'h' is given

min node $n = 2h+1$

max " $n = 2^{h+1} - 1$

$\log_2(n+1) \le h \le \frac{n-1}{2}$

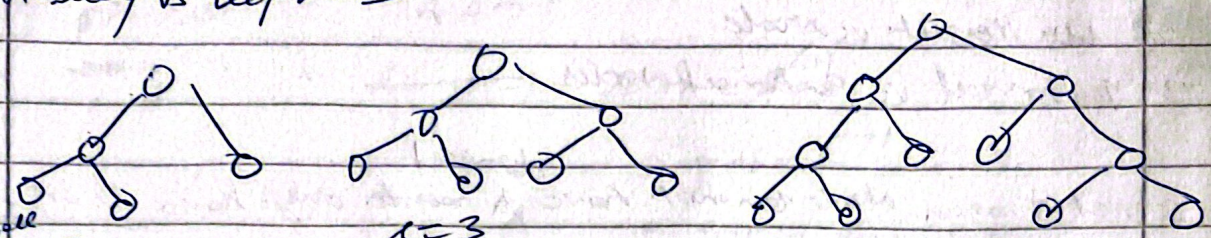
if 'n' node are given

min Height $h = \log_2(n+1) - 1$

max Height $h = \frac{n-1}{2}$

Internal Vs External nodes of Strict Binary Trees.

non leaf Vs leaf nodes



non leaf nodes $i = 2$

leaf nodes $e = 4$

$i = 3$

$e = 4$

$i = 5$

$e = 6$

$\boxed{e = i + 1}$ for strict binary tree.

---

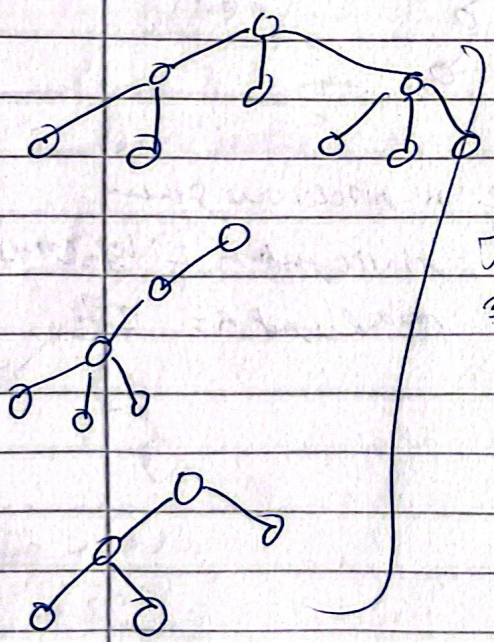
n-ary Trees

n-ary Tree

1. what are nary trees
2. Strict n-ary Tree
3. Perfect nary
4. Internal Vs External mean degree

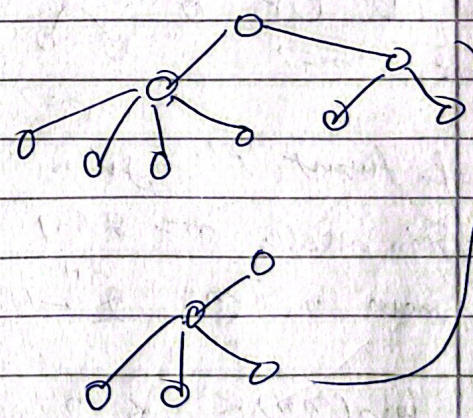
n-ary Tree

3-ary Tree {0, 1, 2, 3}

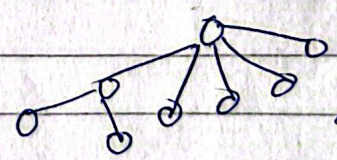


Then 3ary tree

4-ary {{0, 1, 2, 3, 4}}

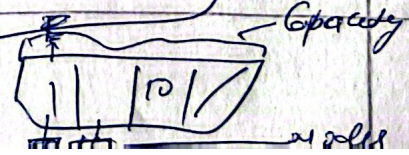


Then 4ary tree



X Then not 4ary tree

Seve the capacity for ary



Strict 3 ary Tree → have 1 or 3 degree (degree)

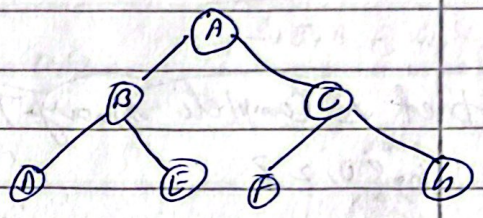
Analysis of n-ary Trees

Representation of Binary Tree

① Array Rep.
② Linked Rep

(I)

T	A	B	C	D	E	F	G
	1	2	3	4	5	6	7



maintain parent & child representation:

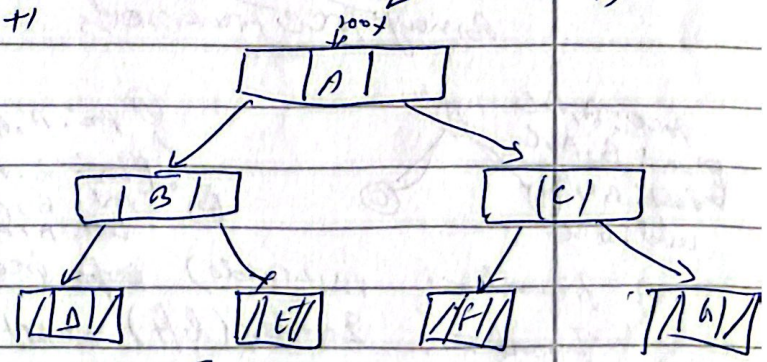
element	index	L.child	R.child
A	1	2	3
B	2	4	5
C	3	6	7
	i	$2*i$	$2*i+1$

element – i
L.child – $2*i$
R.child – $2*i+1$
parent $= \frac{i}{2} = \text{floor}(\frac{i}{2})$

(II) Node

lchild	data	rchild

struct node E
struct node * lchild
int data
struct node * rchild
}



$n = 7$
null pointer $= n+1$

$\boxed{c = i+1}$ → from strict binary tree

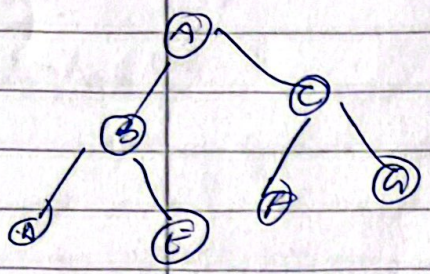
Scanned with CamScanner

Full vs Complete Binary Tree:

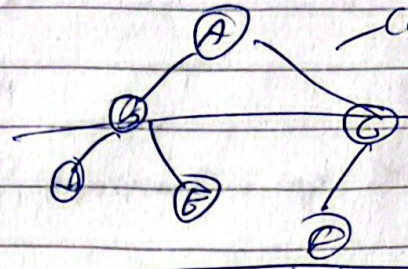
Height of Binary tree having maximum no. of nodes.

~~$h = 2$~~

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



T | A | B | C | D | E | F | G |
1 2 3 4 5 6 7

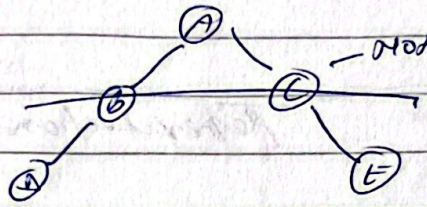


Complete

Height
0
1
2

up to here full

T | A | B | C | D | E | F |
1 2 3 4 5 6



Not Complete 1/2 space in array
↓ full upto here but below
L-R missing

T | D | B | C | A | - | - | E |

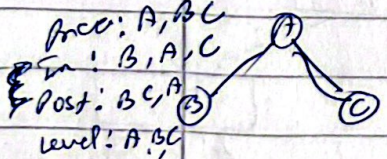
Strict vs Complete Binary Tree

Strict deg {0, 2}

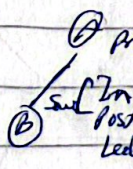
Complete → should have any blank space

Strict	Complete
↓ Sometime Complete	↓ almost Complete

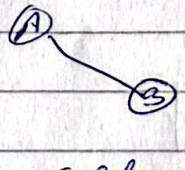
Binary Tree Traversals



Pre: A, B, C
In: B, A, C
Post: B, C, A
Level: A, B, C

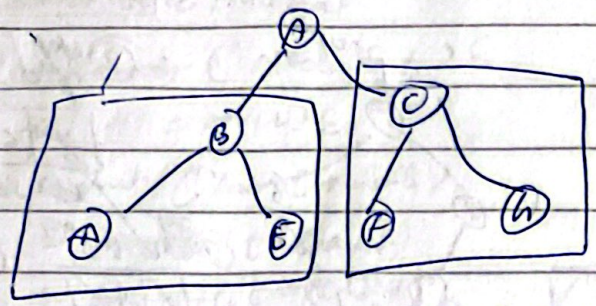


Pre: A, B
Sub In: B, A
Post: B, A
Level: A, B



Pre: A, B
In: A, B
Post: B, A
Level: A, B

Preorder: visit(node) perform preorder(left subtree), preorder(right subtree)
Inorder: Inorder(left), visit(node), Inorder(right)
Postorder: Postorder(left), Postorder(right), visit(node)
Level order: Level by Level.



Pre: A, (B, D, E), (C, F, G)

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

Inorder: (D, B, E), A, (F, C, G)

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

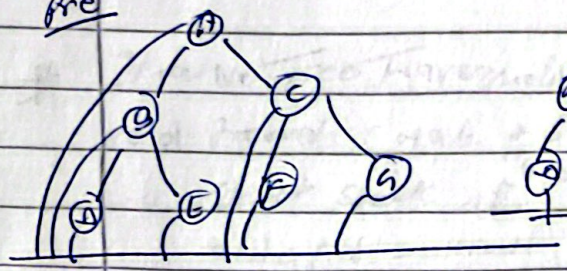
Post: (D, E, B), (F, G, C), A

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

Level: A, B, C, D, E, G, G

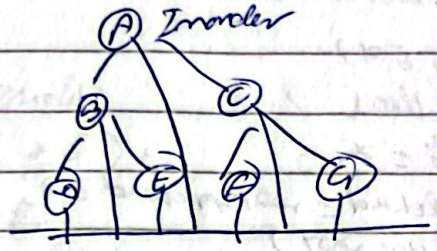
method ①

Pre



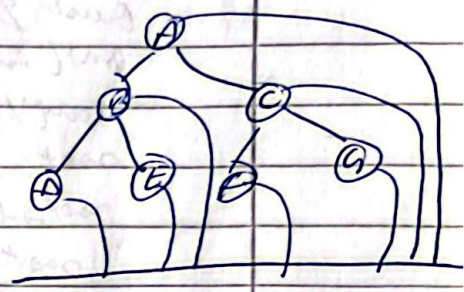
A, B, D, E, C, F, G

Inorder



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

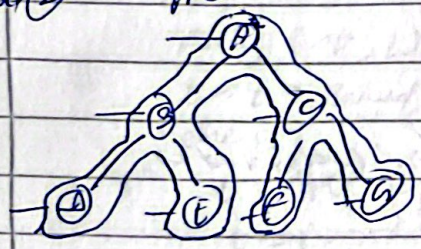
Post



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

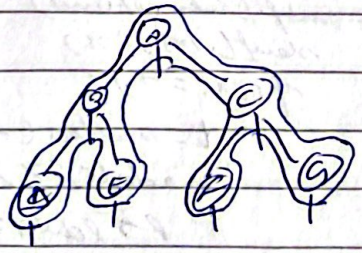
method ②

Pre



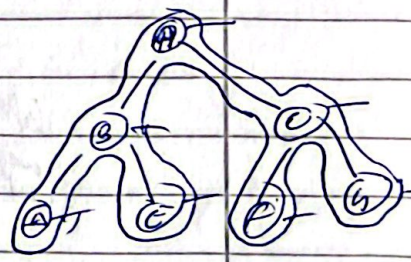
A, B, D, E, C, F, G

In



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

Post



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

method (III) take tree from previous

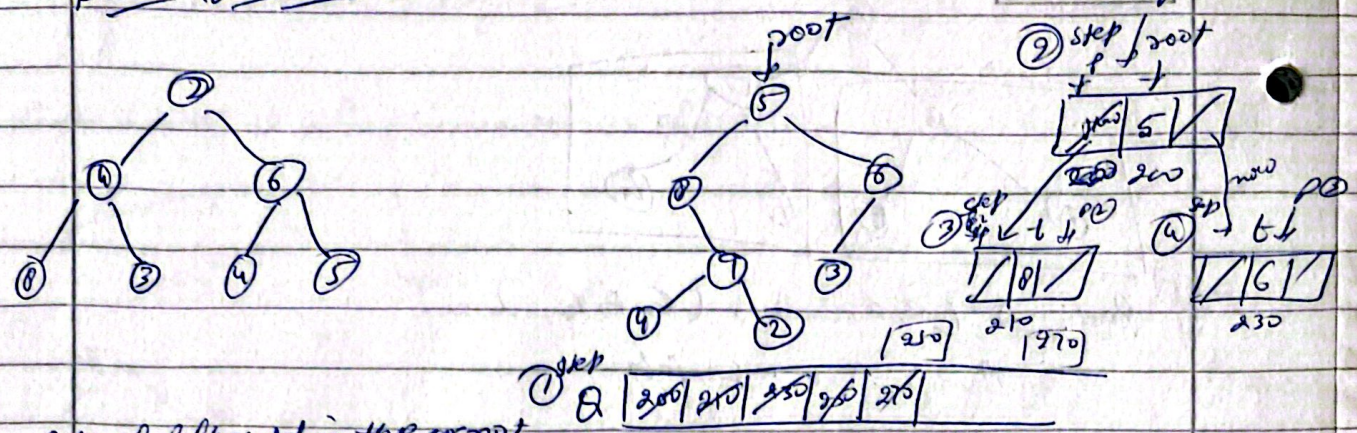
unchanged

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

Inorde: D, B, E, A, F, G, G

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

Creating Binary Tree



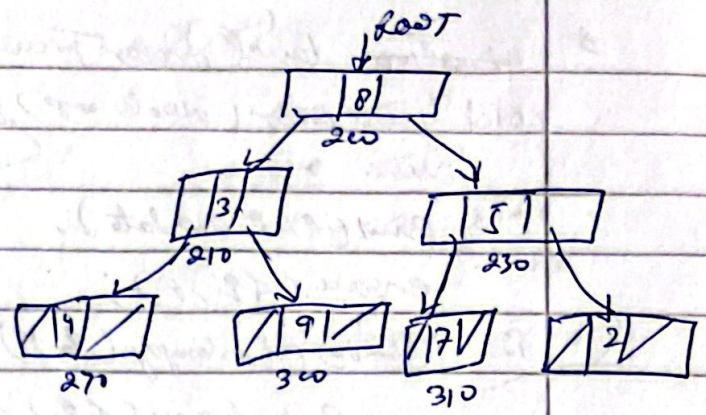
Ask if left child is there or not
if yes then create likewise right also

```
void Create() {
    Node *p, *t;
    int x;
    Queue Q;
    printf(Enter root value)
    scanf("%d", &x)
    root = malloc( );
    root->data = x;
    root->lchild = root->rchild = 0;
    enqueue(root)
    while(!isEmpty(Q)) {
        p = dequeue(&Q)
        printf(Enter left child)
        scanf("%d", &x)
        if(x != -1) {
            t = malloc(--);
            t->data = x; t->lchild->rchild = 0;
            p->lchild = t;
            enqueue(t);
        }
    }
}
```

left
Same for
right

① Preorder Tree Traversal

```
void preorder ( Node * t) {
    if ( t != NULL) {
    1. printf("%d", t->data)
    2. preorder( t->lchild)
    3. preorder( t->rchild)
    }
}
```



② Inorder Tree Traversal

```
void Inorder( Node *t) {
    if ( t != NULL) {
        Inorder ( t->lchild)
        printf("%d", t->data)
        Inorder ( t->rchild);
    }
}
```

Iterative Tree Traversals.

```
void preorder ( Node * t) {
    struct stack st;
    while ( t != NULL || !isEmpty(st) {
    if (t != NULL) {
        printf("%d", t->data)
        push( &st, t);
        t = t->lchild;
    } else {
        t = pop(&st);
        t = t->rchild;
    }
}
```

Iterative Tree Traversals

```
void Inorder ( Node *t) {
    while
    {
        push( &st, t)
        t = t->lchild;
    } else
        t = pop(&st);
        t = t->rchild  → printf("%d", t->data);
    }
}
```

~~Iterative~~ level order Traversal

```
void levelorder (node *p) {
    Queue q;
    print("%d", p->data);
    enqueue (&q, p);
    while (!isEmpty(q)) {
        p = dequeue(&q);
        if (p->lchild) {
            print("%d", p->lchild->data)
            enqueue (&q, p->lchild);
        }
        if (p->rchild) {
            print("%d", p->rchild->data);
            enqueue (&q, p->rchild);
        }
    }
}
```

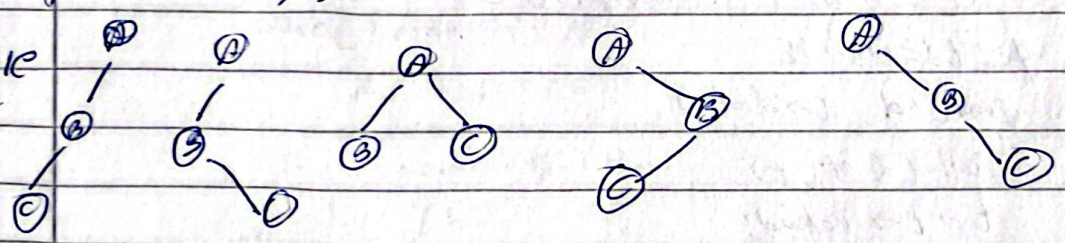
Can generate Tree from Traversal.

n=3

(A) (B) (C)

preorder — A, B, C

possible tree



what if are two traversal preorder & postorder → none from one tree can have same order

preorder: A, B, C postorder: C, B, A

There two can give ~~tree~~ tree by

① preorder + Inorder

② postorder + Inorder

Generating Tree from Traversal

Preorder — 4, 7, 9, 6, 3, 2, 5, 8, 1

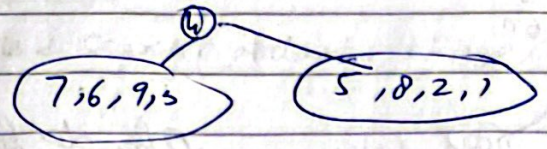
Inorder — 7, 6, 9, 3, 4, 5, 8, 2, 1

take Inorder as a set (7, 6, 9, 3, 4, 5, 8, 2, 1)

(through slant preorder)

Next

$O(n^2)$



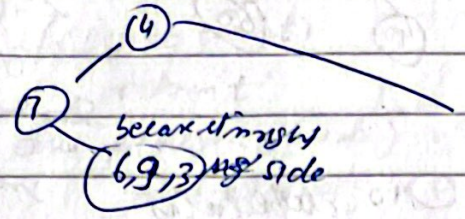
Pre — n

In — n

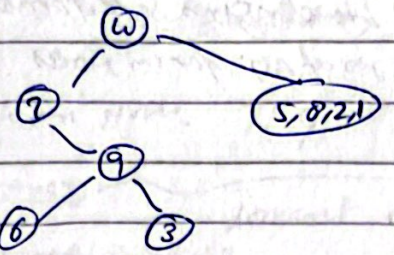
$n \times n = n^2$ search

finally $O(n^2)$

Next search for 7



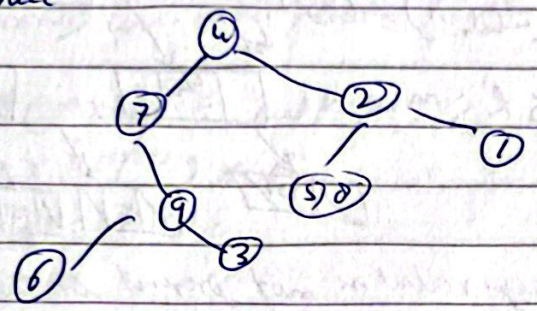
Next 7



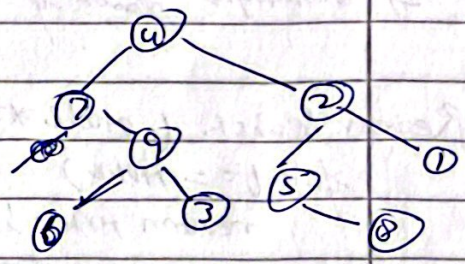
Next 6 found here

Next 3, found here

2



Search 5



Next 8 found

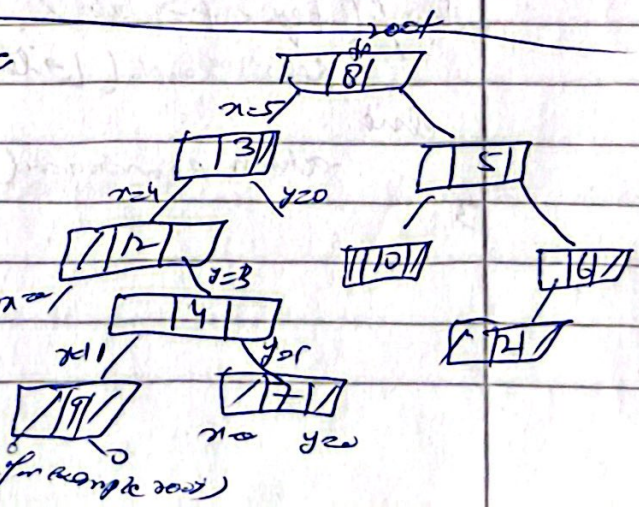
Next 1 found

Height & Count of Binary Tree

```
int Count (Node *p) {
    int x, y
    if (p != NULL) {
        x = Count (p->leftchild)
        y = Count (p->rightchild)
        // return x+y+1;
    }
    else {
        return 0;
    }
}
```

Code modified

```
if (p->leftchild)
    return x+y+1;
else
    return x+y;
```



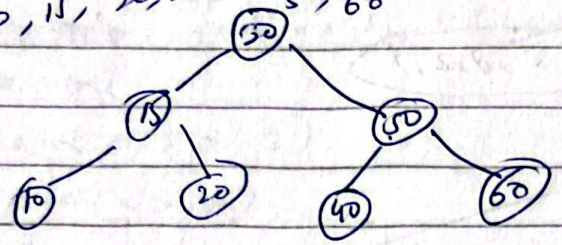
// return x+y+1; — this one

// return x+y+1; — right side

(for example root)

Binary Search Trees (BST)

Inorder - 10, 15, 20, 30, 40, 50, 60



Left side lesser than parent & 15 & right side is greater than Search time height of tree.

Inorder traversal

(i) What is BST?

(ii) Properties of BST — (i) No duplicates
(ii) Inorder gives sorted order
(iii) No of BST for 'n' nodes

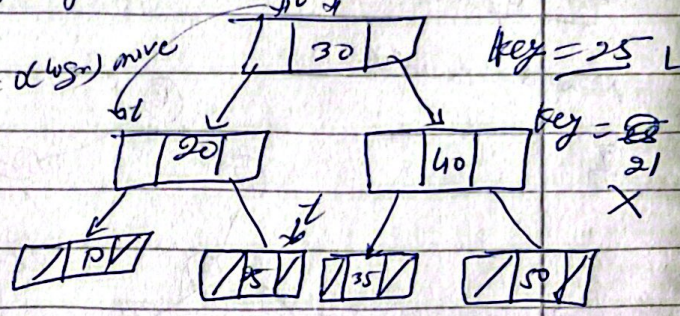
Q3 BST representing by linked.

Searching in a Binary Search tree:

Successfull Search in height of tree.

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

Height $= \log_2 n \le h \le n$

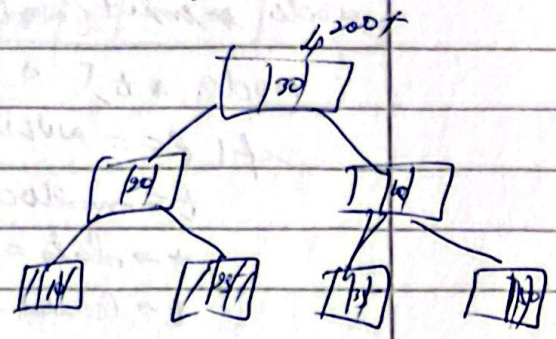


→ unsuccessfull search if key value is not present in tree.

```
node* RecursiveSearch (node *t, int key) {
    if (t == NULL)
        return NULL;
    if (key == t->data)
        return t;
    else if (key < t->data)
        RecursiveSearch (t->leftchild, key);
    else
        return RecursiveSearch (t->rightchild, key);
}
```

Iterative function

key = 35




```
Node * search (Node*t, int key) {

    while (t != NULL) {
        if (key == t->data)
            return t;
        else if (key < t->data)
            t = t->lchild;
        else
            t = t->rchild;
    }
    return NULL;  // if not key found
}
```

Insert in a Binary Search Tree

① first find this key is there or not in tree.
where have to inserted there only will be inserted.

key = 38

create New Node
to store the
link.

→ fail searches are two pointers frequently used

p, t
Help to insert node
when t becomes NULL

```
void insert (Node* t, int key) {
    Node *r = NULL, *p;
    while (t != NULL) {
        r = t;
        if (key == t->data)
            return;
        else if (key < t->data)
            t = t->lchild;
        else
            t = t->rchild;
    }
    p = malloc (--) p->data = key;
    p->lchild = p->rchild = NULL;
    if (p->data < r->data) r->lchild = p;
    else r->rchild = p;
```

```
int main() {
    node *root=NULL;
    root=insert(root,30);
    insert(root,20);
    insert(root,25);
```

Recursive insert in binary search tree.

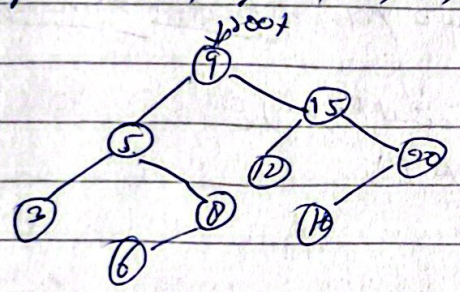
```
node *insert(node * p; int key) {
    node *t;
    if(p==NULL) {
        t=malloc()()
        t->data=key
        t->lchild=t->rchild=NULL;
        return t;
    }
    if(key<p->data)
        p->lchild=insert(p->lchild,key);
    else if(key>p->data)
        p->rchild=insert(p->rchild,key);
    return p;
}
```

if it is root.

Creating a Binary Search Tree.

key: 9, 15, 5, 20, 16, 8, 12, 3, 6

start from root



insert → n element
search – log n

$O(n \times \log n)$

Deleting from binary search tree.

if find make parental link NULL.

if have one child then no problem if have two child then who will take place of its next smaller predecessor or successor.

Generating BST from preorder

pre	30	20	10	15	25	40	50	45
	0	1	2	3	4	5	6	7

(i) preorder + inorder

(ii) postorder + inorder

```
void create( int pre[], int n) {
    Stack stk;
    node *t;
    int i=0;
    root = new Node;
    root->data = pre[i+1];
    root->lchild = root->rchild = NULL;
    p = root;
    while (i<n) {
        if (pre[i]<p->data) {
            t = new Node;
            t->data = pre[i+1];
            t->lchild = t->rchild = NULL;
            p->lchild = t;
            push(&stk, p);
            p = t;
        }
        else {
            if( pre[i] > p->data && pre[i] < stack(&stk.)->data
                t = new Node; t->data = pre[i+1];
                t->rchild = t->rchild = NULL;
                p->rchild = t; p = t;
            }
            else p = pop(&stk);
        }
    }
}
```

key = 40, 20, 30, 60, 50, 70, 70

key = 10, 20, 30, 40, 50, 60, 70

Drawbacks of BST

Same set of key could be different height

& as minimum as $\log n$
" maximum of n

} in dependent order & ~~Com~~ order
Can not be do

Height balanced tree in AVL tree

av

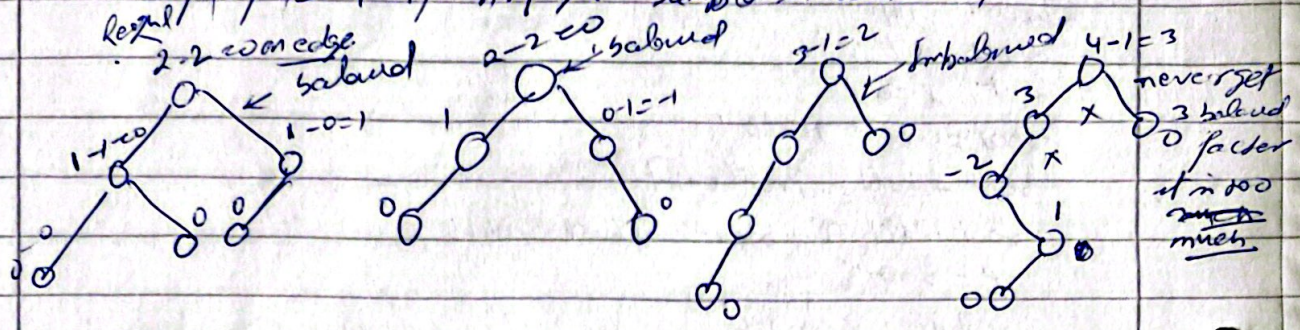
AVL trees

Balance factor = height of left subtree − height of right subtree.

$bf = h_l - h_r = \{-1, 0, 1\}$

If $|bf| = |h_l - h_r| \le 1$ balanced

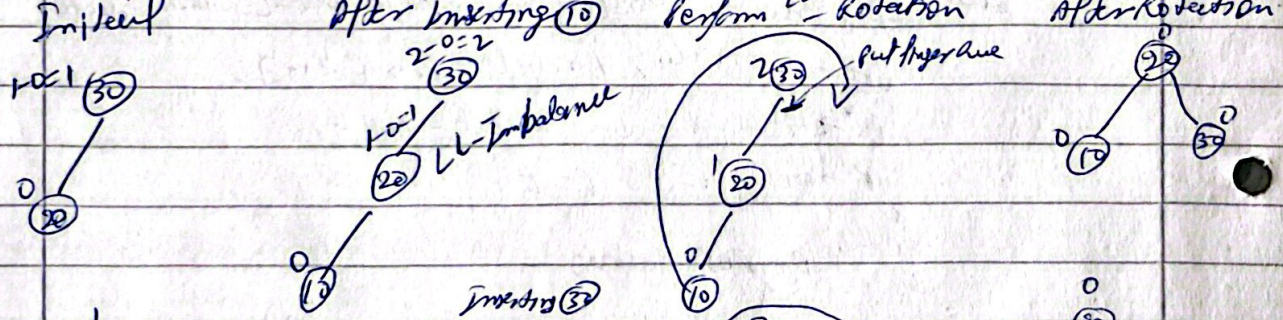
$|bf| = |h_l - h_r| > 1$ imbalanced.



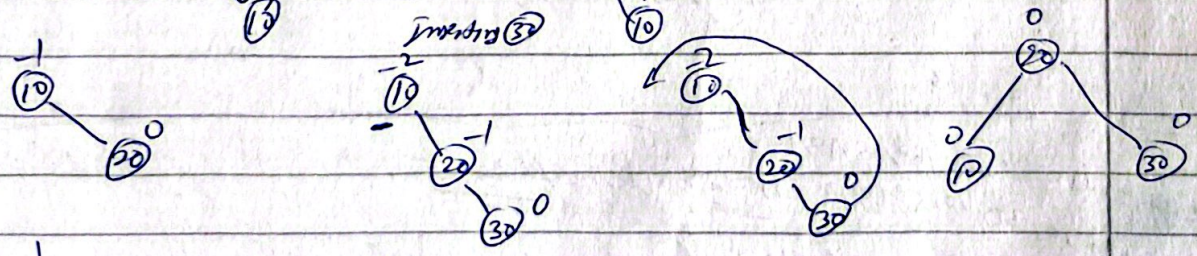
Inserting in AVL with rotations

1. LL - rotation
2. RR - rotation
3. LR - rotation
4. RL - rotation

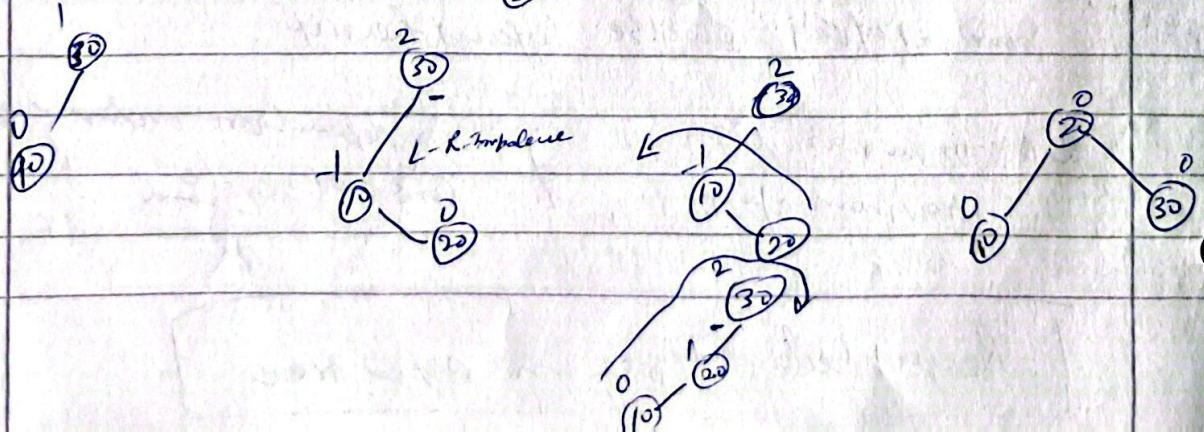
(I) Initial — After Inserting (10) — Perform LL - Rotation — After Rotation



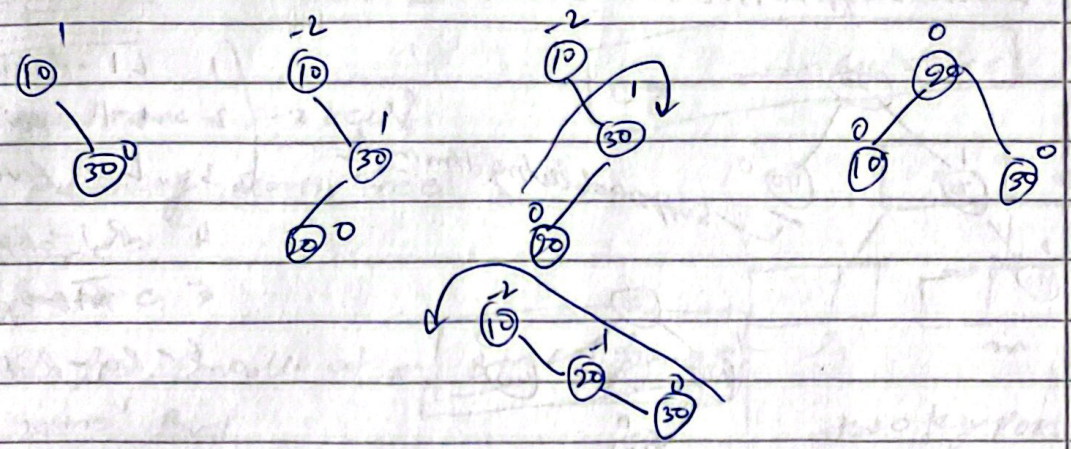
(II)



(III)



(N)



LL-Rotation
RR-Rotation] - Single Rotation

LR-Rotation
RL-Rotation] Double Rotation

Generalized form of AVL Rotations.

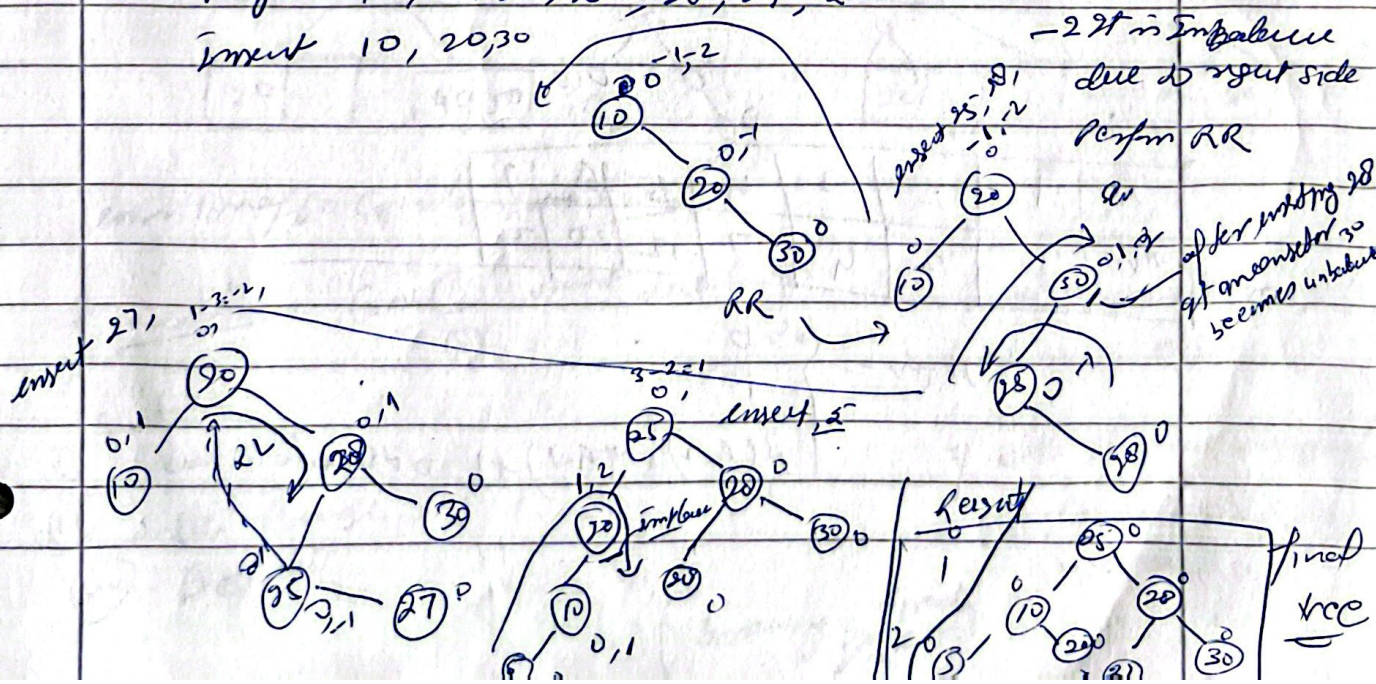
Formulae of Rotations for Insertion
LL-Rotation

LL-LR code

Generating AVL Trees.

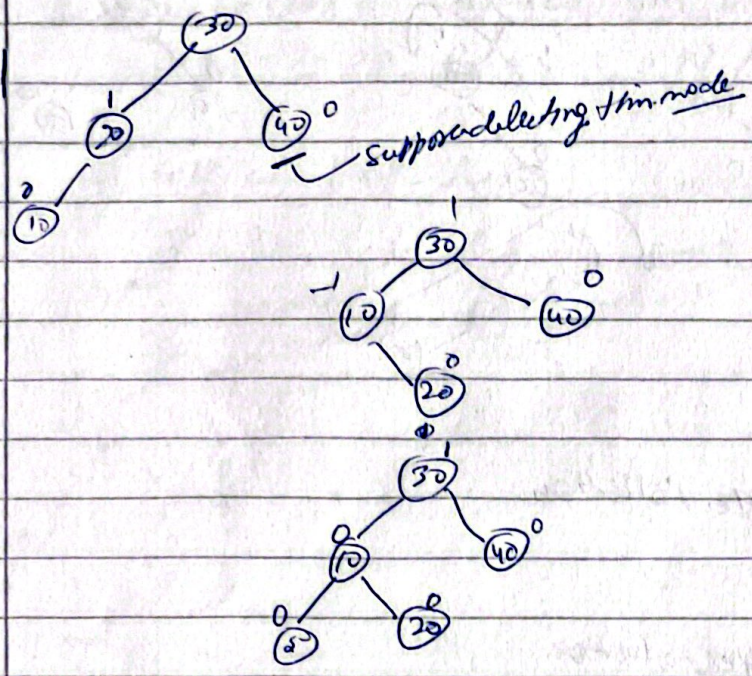
Keys: 10, 20, 30, 25, 28, 27, 5

Insert 10, 20, 30



Deletion from AVL Tree with Rotations

L1



1 L1 - Rotation
2 L-1 - Rotation
3 L0 - Rotation
4 R1 - Rotation
5 ~~R-1~~ R-1 - Rotation
6 - R0 - Rotation

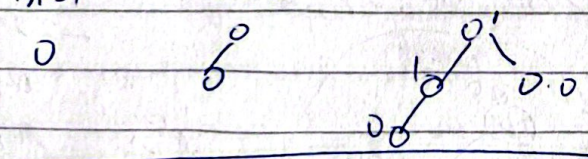
Height Analysis

if Height is given find

min node $n =$

max node $n = 2^{h+1} - 1$ ← ~~this coming from binary tree~~.

$h=1$	$h=2$	$h=3$
$n=1$	$n=2$	$n=4$



minimum means it should be balanced.

h	1	2	3	4	5	6	7
$N(h)$	1	2	4	7	12	20	33

$$N(h) = \begin{cases} 0 \\ 1 \\ N(h-2) + N(h-1) + 1 & \text{otherwise} \end{cases}$$

Search Trees

2-3 Trees

Wheels are 2-3 trees

→ multiway Search Tree.

→ Degree 3

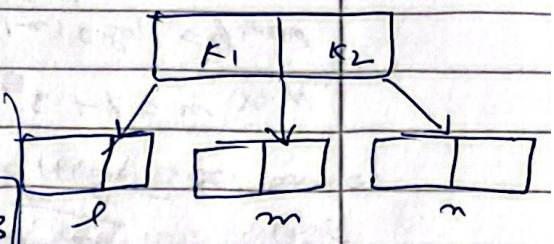
→ B-Tree

→ All leaf nodes at same level

→ Every node must have $\lceil \frac{m}{2} \rceil = \lceil \frac{3}{2} \rceil = 2$ children

$\boxed{\text{way} - \text{means the degree}}$

$\boxed{\text{B tree of degree 3}}$



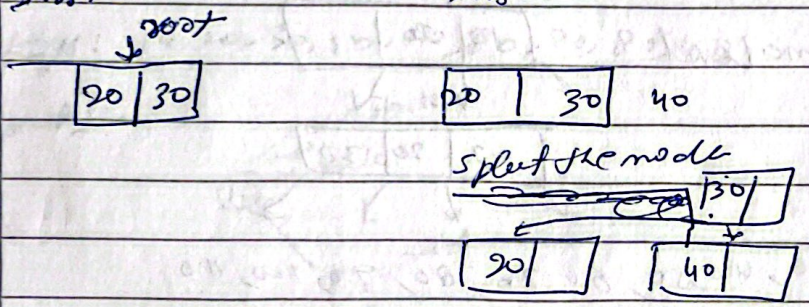
two key possible $K_1 < K_2$

$l < K_1$ $\quad K_2 > n$

$K_1 < m < K_2$

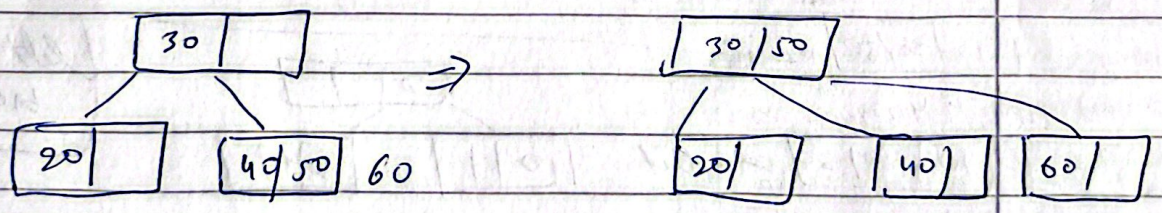
Keys → 20, 30, 40, 50, 60, 10, 15, 70, 80, 90

Insert 20, 30 $\quad$ Insert 40



2-3 tree grow upward

insert 50, 60



insertion of 60 creates split here

Delete. Case (i) want delete 90, 80 (ii)

if node empty merge at with sibling

Simple delete

(i) delete

(ii) delete & merge

(iii) Borrow

if deleting right sibling then shift left sibling to right side

(Borrowing from root)

7th

B-tree
B+-tree] mostly used in DBMS

Ceil value $\left\lceil \frac{3}{2} \right\rceil = 2$

$\boxed{\text{2-3 is type of B tree}}$

min $n = 1 + 2 + 2^2 + \dots + 2^h = 2^{h+1} - 1$

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

max $h = \log_2(n-1) - 1 = O(\log_2 n)$

Max $n = 1 + 3^1 + 3^2 + \dots + 3^h = \frac{3^{h+1} - 1}{3 - 1}$ (GP Series)

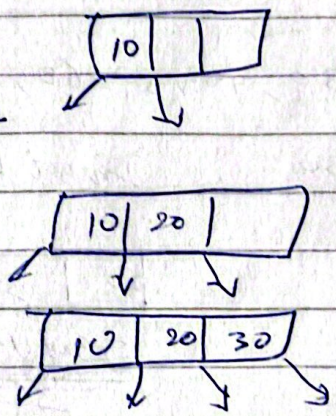
$n = \frac{3^{h+1} - 1}{3 - 1}$

min $h = \log_3[n(3-1)+1] - 1 = O(\log_3 n)$ called big O

2-3-4 trees

$d = 4$

- B-tree of degree 4
- Every node must have $\left\lceil \frac{d}{2} \right\rceil = 2$ children
- All leaf at same level

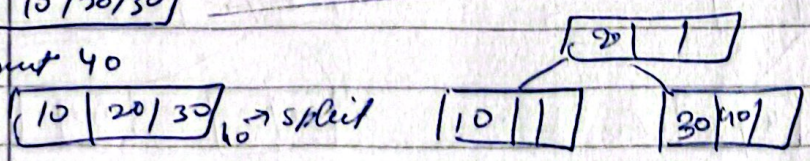


keys: 10, 10, 30, 40, 50, 60, 70, 80, 90, 100, 110

Insert 10, 20, 30

$\boxed{10 / 20 / 30}$

insert 40

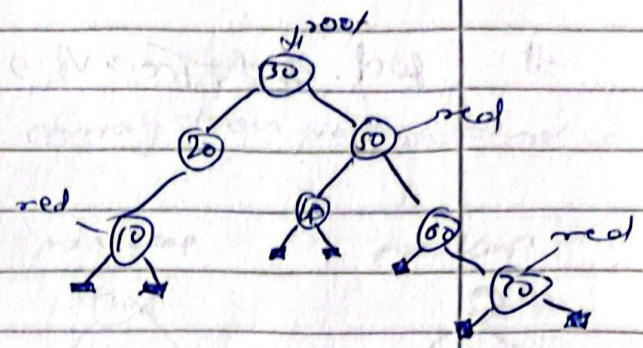


left-bias or Right bias splitting

1

Red-Black Trees Introduction

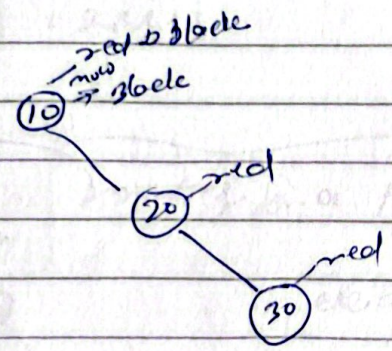
1. Its a Regular Balanced Binary Search Tree, similar to 2-3-4 tree
2. Every node is either Red or Black
3. Root of the Tree Black
4. NULL is also Black
5. Number of Blacks on paths from Root to leaf are same.
6. No. 2 Consecutive Red, parent & child of Red are Black.
7. Newly inserted node is Red
8. Height is $\log n \le h \le 2 \log n$



Red-Black Tree creation

Keys: 10, 20, 30, 50, 40, 60, 70, 80, 4, 8

insert 10



insert 20

insert 30

(insertion just like binary search tree)

(newly inserted node will be Red but root must be black.)

(recoloring or rotating)

→ (newly inserted node if its parents red then check uncle
Uncle may be red or black
if uncle is red the recolor
if uncle is black then perform rotation)

Red-Black Trees vs 2-3-4 Trees