Data structure

- linear
  - arrays, L.L, queue, stack
- non-linear
  - tree, Graph.

⇒ Trees can be defined as a collection of entities (nodes) linked together to simulate a hierarchy.

root : The first node of a tree. (or) node which not having any predecessor

parent : An immediate predecessor of any node is known is parent.

child : An immediate successor of any node.

Leaf node : (external nodes) don't have any child node

non-leaf node : having atleast one child node (internal node)

path : sequence of consecutive edges from source node to destination node. (path from B to J is B to E and E to J) edge = link b/w 2 nodes.

ancestors : any predecessor node on the path from root to that node.

descendent : any successor node on the path from the node to leaf node.

sub tree : a node with all its descendent nodes

Sibling : child nodes having same parent.

degree : no. of children. of that node.

⇒ degree of tree = max degree of tree

⇒ degree of leaf node = 0

Depth of node : length of the path from root to that node

⇒ Depth of root = 0 ⇒ Depth of tree = max depth of tree

height of node : no. of edges in the longest path from that node to a leaf

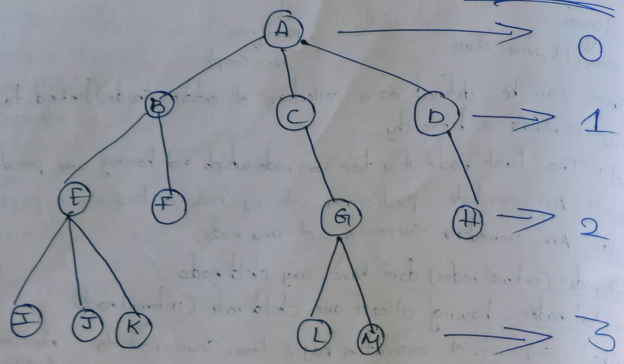
height of tree = height of root node.

level node : level of node = Depth of node.

level of tree = height of tree

height:

depth = level



root = A

nodes = A, B, C, D, E, F, G, H, I, J, K, L, M

Parent node = G is the parent of L & M.

Child node = L & M are children of G.

leaf node = J, I, K, F, L, M, H

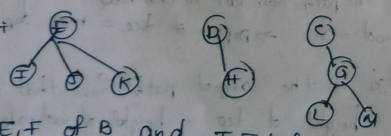
non leaf = A, B, C, D, E, G.

ancestors = L is A, C, G E, M is A, C, G

descendent = C is G, L, M ~~are C is G, M~~

B is E, I, J, K.

sub tree =



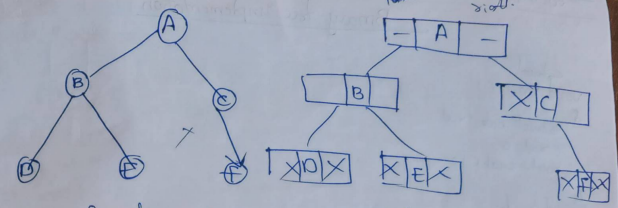
Sibling: E, F of B and I, J, K of E

Degree of: E is 3, G is 2 D, is 1

Depth of node: J is 3, D is 1, A is 0 count J to top, E is 2

height of node: B is 2 edges count B to down, E is 1

⇒ If n nodes = (n-1) edges.



⇒ max no. of node in a level is $2^n = 2^0, 2^1, 2^2, 2^3, 2^4, 2^5$ (i)

⇒ Max no. of nodes ~~of a binary tree~~ = $2^{n+1} - 1$ = $2^{1+1} - 1 = 2^2 - 1 = 4 - 1 = 3$

height h

$= 2^{3+1} - 1 = 2^4 - 1 = 15$

~~max~~

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

$= 3^{rd}$ level $= 0 + 1 + 2 + 4 + 8 = 15$

$= 2^0 + 2^1 + 2^2 + 2^3 + 2^4 \dots + 2^h$

⇒ Min no. of node of height h = h+1

→ each node can have atmost 2 children.

Binary tree implementation

```
Void main()
{
  struct node *root;
    root=0;
  root=create();
}
struct node.
 int data;
struct node *left,*right;
};
struct node*create()
  { int x;
    struct node* newnode;
newnode =(struct node*) malloc (size of (struct node));
   printf ("Enter data (-1 for no node ):");
   Scanf ("%d", &x);
    if (x==-1)
    { return 0;
    }
newnode->data=x;
Pf("Enter Lc of %d ",x)
newnode->left=create();
 printf ("Enter right child of %d",x);
   newnode->right=create();
   return newnode;
}
```

Array representation of Binary tree

case I

if a node is at ith index:

left child be at i: (2*i)+1

Right child be at: (2*i)+2

Parent at = (i-1)/2

for

A	B	C	D	E	F	G	H	I	
0	1	2	3	4	5	6	7	8	case I

case II

if a node at ith index

let child be at: (2*i)

Right child be at: (2*i)+1

parent = i/2

for

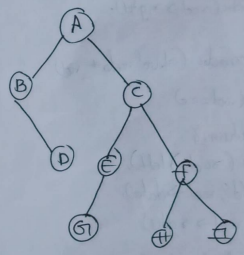
A	B	C	D	E	F	G	H	I	
1	2	3	4	5	6	7	8	9	case II

Inorder, preorder, postorder

Inorder: left Root Right

preorder: Root left Right

Postorder: left Right Root



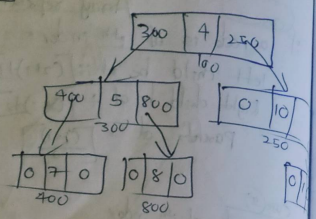
Inorder: BDAGECHFI

Preorder: ABDCEGFHI

postorder: DBGEHIFCA

Binary tree traversals:

```
Void main()
{ struct node *root;
  pf ("preorder is:");
  preorder (root);
  Inorder (root);
  Postorder (root);
Void preorder (struct node* root)
  { if (root)==0)
    { return; }
    Pf ("%d", root->data);
  preorder (root->left)
  preorder (root-> right);
}
Void Inorder (struct node* root)
  { if (root==0)
    { return; }
  inorder (root->left);
  Pf ("%d", root->data);
inorder (root-> right);
}
void postorder (struct node* root)
{ if (root==0)
  { return; }
 postorder (root->left)
postorder (root->right)
pf ("%d", root-> data);
}
```



tree with array:

```
# include <stdio.h>
# include <stdlib.h>
# define size 20
 int tree [size];
int inorder (int);
int preorder (int);
int postorder (int);
 int main()
{ int x,t,i;
 pf ("enter the tree size index:\n");
sf ("%d", &t);
for (int i=0; i<t; i++)
 { pf ("enter the tree data :\n")
  sf ("%d", &tree [i]);
 }
 pf ("entered tree data :\n");
  for (int i=0; i<t; i++)
  { pf ("tree [%d]: %d \n", i, tree [i]);
 }
 int c;
pf ("enter the choice to check tree data
     1 inorder 2 preorder 3 postorder 4 exit\n");
  sf ("%d", &c);
```

binary tree with linked list

```
#include <stdlib.h>
#include <stdio.h>
struct node
{ int data;
  struct Node * left;
  struct Node* right;
};
struct Node* createNode()
{int d;
  struct Node* newnode = (struct Node*)malloc (sizeof(struct node));
   NewNode->left = NULL;
   NewNode->right= NULL;
   Pf ("enter the data");
   sf ("%d",&d);
   if (d==-1)
   return NULL;
   NewNode->data = d;
   Pf ("enter the left child of %d",d);
   NewNode -> left = createNode();
   Pf ("enter the right child of %d",d);
   newNode -> right = creatr Node();
   return NewNode;
}
Void preorder (struct Node* root
{ if (root == NULL)
      return;
   pf ("%d ", root -> data);
  preorder (root -> left);
  preorder (root -> right);
}
Void inorder (struct Node* root)
{  if (root== NULL)
      return;
   inorder (root -> left);
   Pf ("%d", root->data);
   inorder (root -> right);
}
void postorder (struct Node* root)
{ if (root == NULL)
      return;
   postorder (root -> left);
   postorder (root -> right);
    Pf ("%d", root -> data);
}
int main()
{ struct Node* root = NULL;
   root = createNode();
   if (root == NULL)
      return 0;
  Pf ("preorder traversal:");
  preorder (root);
  Pf ("\n");
Pf ("Inorder traversal:");
Inorder (root);
  Pf ("\n");
  Pf ("postorder traversal:");
  postorder (root);
   Pf("\n");
   return 0;
```

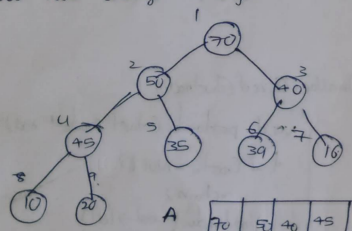
heap

heap is also a tree data structure, it should be complete binary tree. it is divided into two Max heap & min heap

Max heap: for every node i, the value value of node i is less than or equal to its parent value. (except root node because root node not has parent)

$A[parent(i)] \geq A[i]$

→ root node always be largest



i = 3

70 ≥ 40

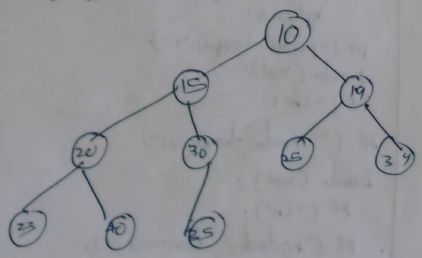
A

70	50	40	45	35	39	16	10	20
1	2	3	4	5	6	7	8	9

Min heap: for every i, the value of node is greater than or equal to its parent value (except root node)

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

→ root node always be smallest

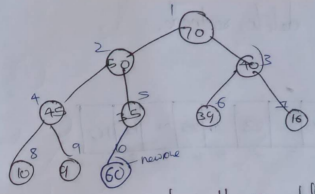


A

10	15	19	20	30	25	39	23	40	35
1	2	3	4	5	6	7	8	9	10

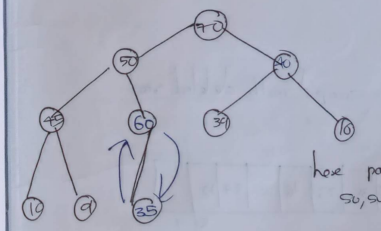
Insertion in Max heap

insertion is always take from last node



70	50	40	45	35	39	16	10	9	60
1	2	3	4	5	6	7	8	9	10

compare check the Inserted value with parent value



70	50	40	45	60	39	16	10	9	35
1									10

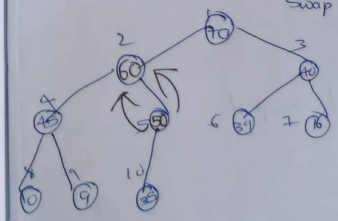
here parent is less than inserted node so, swap it

i = 10

$\frac{i}{2} = 5$

5 is parent node

check 60 with parent (50) if not swap it



70	60	40	45	50	39	16	10	9	35
1	2	3	4	5					

i = 5 $\frac{5}{2} = 2.5$

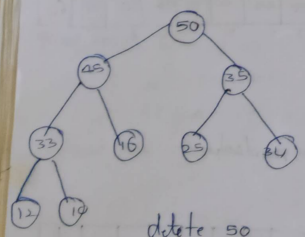
= 2

2 is parent

check 60 with parent

Delete in Max heap

Deletion can be done can only delete rootnode.

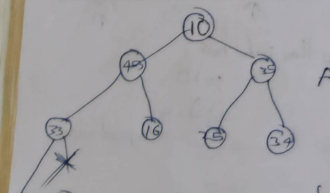


A

1	2	3	4	5	6	7	8	9
50	45	35	33	16	25	34	12	10

delete 50

⇒ and last element in the array/tree swap it with deleted root



A

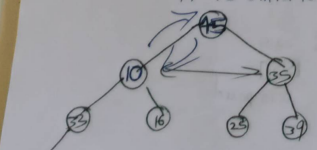
1	2	3	4	5	6	7	8	9
10	45	35	33	16	25	34	12	~~10~~

heap = n-1

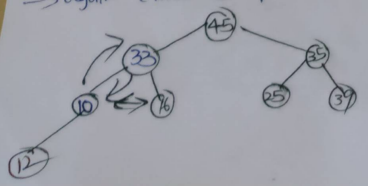
→ and check the Max heap property

→ compare the root with children ...

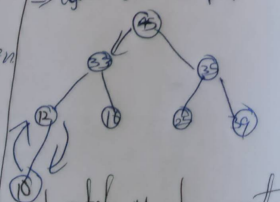
if the child is bigger than root swap it



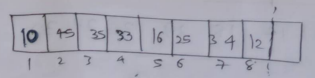
→ again check ~~co~~ compare with children



→ again check compare



It satisfy Max heap property

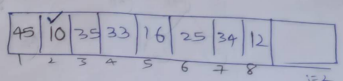


i = 1

left child = 2×i = 2×1 = 2 left

right child = (2×i)+1 = 3 right

2,3 are children of 1 or 10



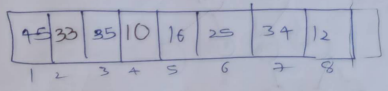
i = 2

45 is larger compare it with 10 if it large swap

left child = 2×2 = 4 ~~33~~

right child = (2×2)+1 = 5 (5 are children of 10)

16

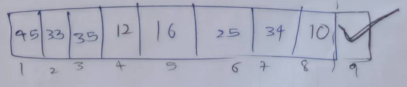


33 is larger compare it with 10 if it large swap

i = 4

l.c = 2×4 = 8

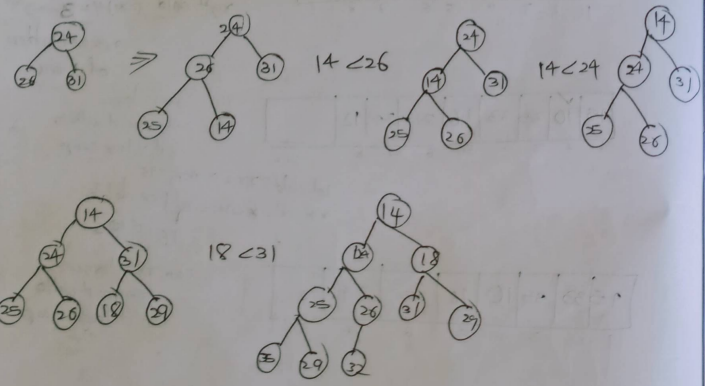
r.c = (2×4)+1 = 9



the i no 4 compare it with 8 (is) swap it

MIN-heap

Ex: 24 26 31 25 14 18 29 35 27 32



Max heap

```
#include <stdio.h>
int maxheap[20], size=0;
void build_max_heap();
void insert();
void display();
void delete()
int main()
{  build max_heap();
   int choice;
   while(1)
  {pf("MENU: \n1.insert \n2. delete \n3. display\n4.exit\n");
   pf("Enter your choice:");
   sf("%d",&choice);
   switch(choice)
   { case 1: insert();
             break;
     case 2: delete();
             break;
     case 3: display();
             break;
     case 4: exit
             return(0);
     default: pf("enter correct choice \n");
   }}
}
void build_max_heap()
        int n;
      pf("Enter the no. of nodes:");
      sf("%d",&n)
      for(int i=0;i<n;i++)
        {
          insert();
        }
        return;
}
```

```
Void insert()
{
  pf("Enter node Data.");
  sf("%d", &maxheap[size]);
    int i= size
     size++,
    while(i>0)
    {
        int parent = (i-1)/2;
        if (maxheap[parent]<maxheap[i])
          {
              int temp= maxheap[parent];
              maxheap[parent]=maxheap[i];
              maxheap[i]=temp;
              i= parent;
          }
        else
        {return;
        }
    }
}
```

```
Void display()
{
  for(int i=0;i<size;i++)
  {
    pf("%d",maxheap[i]);
  }
}
```

```
Void delete()
{
   maxheap[0]=maxheap[size-1];
      size--;
      int i=0;
   while(i<size)
   {
     int max=i
     int left=(2*i)+1;
     int right=(2*i)+2;
   if(left<size && maxheap[left]>maxheap[max])
     {
       max=left;
     }
   if(right<size && maxheap[right]>maxheap[max])
     {
       max=right;
     }
   if(max!=i)
   {
     int temp= maxheap[i];
      maxheap[i]=maxheap[max];
      maxheap[max]=temp;
      i= max;
   }}}
   else
      return;
```

Min heap:

```
#include<stdio.h>
int minheap[20], size=0;
void build_min_heap();
void insert();
void display();
void delete();
int main()
{
build_min_heap();
int choice;
while(1)
{
pf("Menu: 1.ins 2. delete 3. display 4. exit\n");
pf("Enter your choice:");
sf("%d", choice);
switch(choice)
{
case1: insert();
       break;
case2: delete();
       break;
case 3: display();
       break;
case4: exit;
       return 0;
default: pf("Enter correct choice\n");
}
}
void build_min_heap()
{
int size;
pf("Enter no. of nodes:");
sf("%d", &size);
for(int i=0; i<size; i++)
{
insert();
}
return 0;
}
```

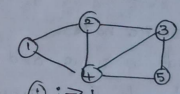
```
void insert()
{
pf("Enter node data:");
sf("%d", &minheap[size]);
    size++;
int i=size-1;
while(i>0)
{
int parent=(i-1)/2;
if(minheap[parent]>minheap[i])
{
int temp=minheap[parent];
minheap[parent]=minheap[i];
minheap[i]=temp;
i=parent;
}
else
{
return;
}
}
}
void delete()
{
minheap[0]=minheap[size-1];
  size--;
  int i=0;
while(i<size)
{
int min=i;
int left=(2*i)+1;
int right=(2*i)+2;
if(left<size && minheap[left]<maxheap[min])
{
min=left;
}
if(right<size && minheap[right]<maxheap[min])
{
min=right;
}
if(min!=i)
{
int temp=minheap[i];
minheap[i]=minheap[max];
minheap[max]=temp;
i=min;
}
else
return;
}
return;
}
void display()
{
for(int i=0; i<size; i++)
{
pf("%d", minheap[i]);
}
}
```

Graphs:

1) Adjacency Matrix: [n][n]
2) Adjacency list

1) Adjacency Matrix:

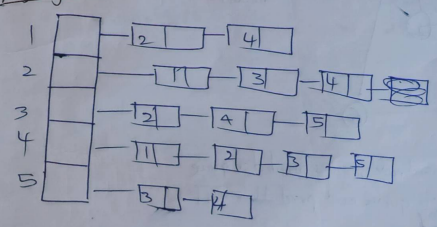
~~if~~ as a Matrix A[n][n]
when n is no. of vertices
& a[i][j]=1 if i & j are adjacent
=0 otherwise



mxn	j=1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

5×5

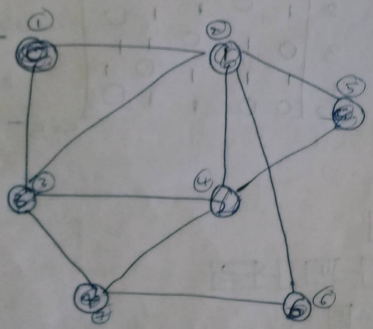
2) Adjacency list:



BFS (Breadth first search or level order):

- follows queue (FIFO)
- first select a source node
- insert the first node in queue and mark the node as visited
- remove the element from queue and find the adjacent vertices
- repeat the steps until all the node are visited

ex:



Queue = 0 1 3 2 4 5 6

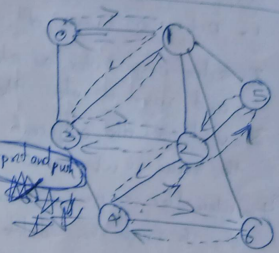
Result=

- starting from zero, as root node, zero is inserted in queue and 0 is deleted in queue and print the result so, the 0 is visited
- adjacent vertices of 0 as 1 and 3
- then delete 1 and print 1 in result so, 1 is visited
- then insert adjacent unvisited vertices in queue (6, 2, 5)
- the delete 3 and print in result, and check adjacent unvisited vertices (0, 1, 4) and mark 3 is visited
- next 2 is delete and make print in result, check adjacent unvisited vertex and mark 2 is visited (4, 5)
- same check for 5, 6, 4

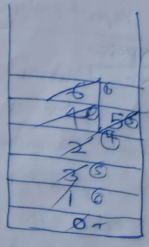
DFS (Depth first search)

- follows stack (LIFO)
- select any node as source node and insert into stack and display it
- Mark it as visited implicitly print the element and mark as visited
- Find the adjacent unvisited vertices of any following node

ex:



- take 0 as root node as starting node
- push 0 into the stack and print in result
- check any one of the vertex of 0, take one and push into the stack (1) and print
- take, unvisited adjacent vertex of 1, only 0 and push into the stack (3) and print
- take, unvisited vertex 3 only as push into stack 2 and print
- then take 4 push into stack and print
- then 6 push into stack
- no unvisited vertices for 6 and print
- then do backtracking 6 to 4 (while 6 is popped out from stack)
- the check for 4 has any unvisited adjacent vertices if not (4 is popped out from stack)



- then check for 2 has any unvisited adjacent vertex as 5 the push into stack and print

Result= 0 1 3 2 4 5 6

- check adjacent vertex for 5, no, perform backtracking by popout of 5
- and check for 2, no adjacent vertex, popout of 2
- and 3, 1, 0
- and stack is empty

BFS:

⇒ BFS Traversal of a graph produces a spanning tree (n vertices, n-1 edges) as final result. Spanning tree is a graph without any loops. we use queue data structure with max size of total number of vertices in the graph to implement BFS

Steps:

1. Define a queue of size total number of vertices in the graph
2. select any vertex as starting point for traversal. visit that vertex and insert it into the queue.
3. visit all the adjacent vertices of the vertex which is at front of the queue which is not visited and insert them into the queue.
4. when there is no new vertex to be visited from the vertex at front of the queue then delete that vertex from the queue.
5. Repeat steps 3 & 4 until queue become empty

DFS:

⇒ DFS traversal of a graph, produces a spanning tree as final result. Spanning tree is a graph without any loops. we use stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph

Steps:

1) select the vertex A as starting point. push A onto the stack.
2) visit any adjacent vertex of A which is not visited (B) push newly visited vertex B on to the stack. and immediately print
3) visit any adjacent vertex of B which is not visited c) Push C onto stack and print
4) If there is no unvisited nodes, perform backtracking by removing current (top most element) from stack.

⇒ Repeat until stack is empty.

adjacency matrix of BFS

```
#include <stdio.h>
#define MAX 6
int adj[MAX][MAX]={0}, visited[MAX]={0};
void breadth_first_search(int);
void addEdge(int,int);
void main()
{
  addEdge(0,1);
  addEdge(0,2);
  addEdge(0,3);
  addEdge(1,3);
  addEdge(2,4);
  Pf("Entered matrix is:\n");
  for(int i=1;i<=MAX;i++)
  {for(int j=1;j<=MAX;j++)
   {Pf("%d",adj[i][j]);
   }
   Pf("\n");
  }
  Pf("\n BFS traversal:");
  Breadth_first_search(0);
}
void addEdge(int src, int dest)
{ adj[src][dest]=1;
  adj[dest][src]=1;
}
void breadth_first_search(int vertex)
{ int queue[MAX],rear=0,front=0,i;
  queue[rear++]=vertex;
  Visited[vertex]=1;
  while(front<rear)
  { int visit=queue[front++];
    Pf("%d",visit);
```

```
for(i=0;i<MAX;i++)
  {
    if(adj[visit][i]==1 &&
                visited[i]==
      {queue[rear++]=i;
       visited[i]=1;
      }
  }
}
```

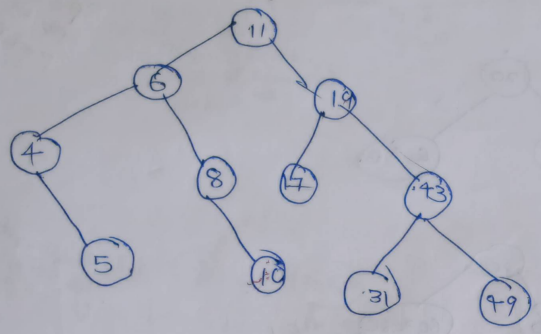
Binary Search tree

→ BST is a non-linear data structure

→ In this left side element must be less than its and parent and right side element must be greater than its parent recursively this condition should be satisfied through out the sub tree

→ searching starts from root and inserting

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31



Deletion → (3) cases

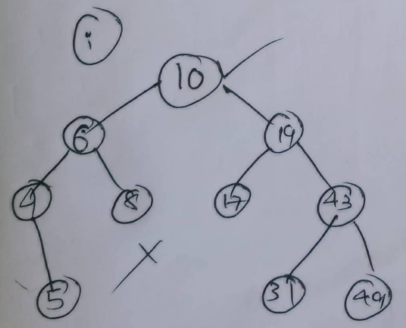
① 0 child → directly deleted
② 1 child —
③ 2 children.

4, 5, 6, 8, 10, 11, 17, 19, 31, 43, 45

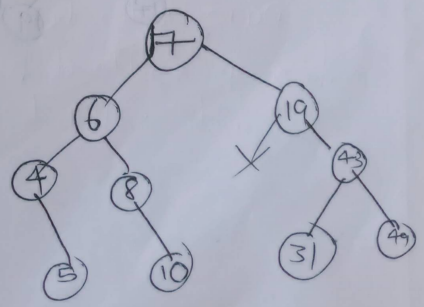
② if ④ is deleted, 4 is replaced ⑤

③ can deleted one with i) inorder predecessor (largest element in left subtree)
ii) inorder successor (smallest element in right subtree)

i)



ii)



Construct of BST when preorder

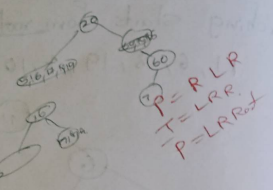
Preorder :- 20, 16, 5, 18, 17, 19, 60, 85, 70 (root, left, Right)

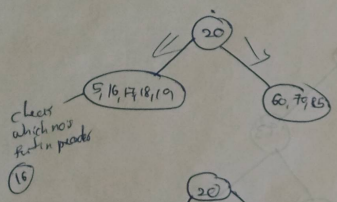
Inorder :- 5, 16, 17, 18, 19, 20, 60, 70, 85 (left, root, right)

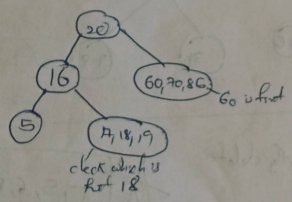
ascending order

left — root — Right

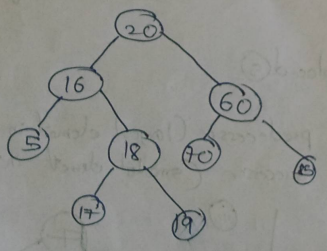
20 is the root from preorder







Root left right
left root right
left is root



postorder = (left, right, root)

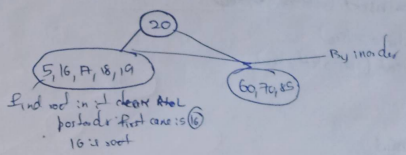
⇒ 5, 17, 19, 18, 16, 70, 85, 60, 20

postorder :- 5, 17, 19, 18, 16, 70, 85, 60, 20 (L R root)

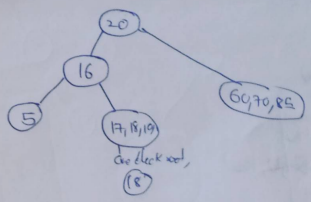
Inorder :- 5, 16, 17, 18, 19, 20, 60, 70, 85 (L Root right)

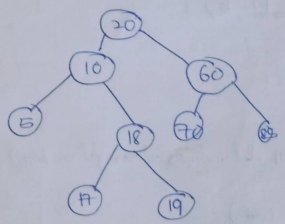
Find root clear post order

20 is root (20)



find root in left check AtoL
postorder first one is 16
16 is root





algorithm:

1) Insert (node, key)

2) Insert (root == NULL)
   return node;

3) if (root->data < key)
   root->right = insert (root->right, key)

4) else if (root->data > key)
   root->left = insert (root->left, key)

5) return node

Searching:

search (root, key)

1) if (root->data == key)
   return root;

2) else if (root->data < key)
   return search (root->right, key)

3) else
   return search (root->left, key)

4) return root;

Deletion:

i) no child
   if (root->left == NULL && root->right == NULL)
   {
   free (root);
   return (NULL);
   }

ii) 1 child.
   replace node with its child, delete the child
   else if (root->left == NULL)
   {temp = root->right;
   root->data = temp->data;
   free (temp);
   }

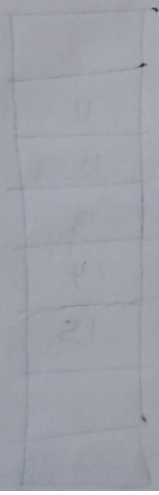
   else if (root->right == NULL)
   {temp = root->left
   root->data = temp->data;
   free (temp);
   }

(3) 2 children

→ find inorder successor or predessor

→ replace node with inorder successor or predessor

→ delete the ~~inorder~~ successor



Hashing:

⇒ Hashing is a searching technique a process which is use of to map key with the values

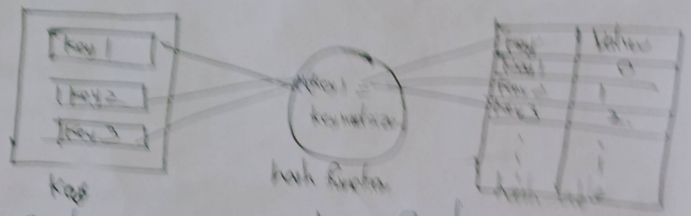
⇒ with the help of hashing we can directly access the elements from the hash table

⇒ hash efficiency depends upon type of hash function applied on.

Key: key is a values a string which will be given as input to the hash function

hash table: hash table which is used to map keys and value into a type data structure

hash function:



key

⇒ hash function is mathematical function where key will be taken as input and it will generate hash value and will store the key based on generated hash value in a hash table.

List: 11, 12, 13, 14, 15

M=10

H(K) = K mod M

key size.

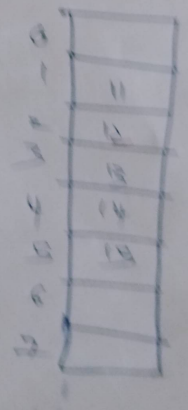
H(11) = 11 mod 10 = 1

H(12) = 12 mod 10 = 2

H(13) = 13 mod 10 = 3

H(14) = 14 mod 10 = 4

H(15) = 15 mod 10 = 5



Types of hash function :-

① Division ② midsquare ③ digit folding ④ multiplication.

① Division method :-

$H(K) = K \bmod M$

(K = key, M = size)

$M = 10$

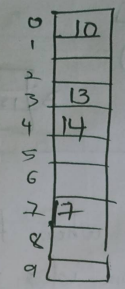
list = {10, 13, 17, 14}

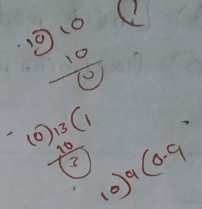
$H(10) = 10 \bmod 10 = 0$

$H(13) = 13 \bmod 10 = 3$

$H(17) = 17 \bmod 10 = 7$

$H(14) = 14 \bmod 4 = 4$





② Midsquare method :-

⇒ find square of key

⇒ Middle digit is hash value (index)

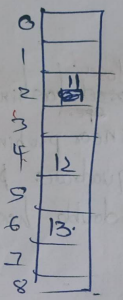
⇒ In that hash value store key

list = {11, 12, 13}

$K = 11, 12, 13$

$K^2 = 121, 144, 169$

(2, 4, 6) or hash value.



③ Digit folding Method :-

⇒ ~~First, we have to divide the key into equal no. of parts except last digit. no. of digits should be less than remaining parts~~

- Keys are partioned.
- parts are added together.

$K = 12345$

$H(K) = (K_1, K_2 \cdots K_n)$

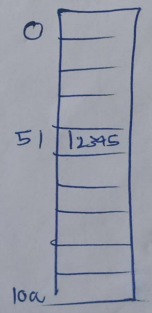
$S = K_1 + K_2 + \cdots K_n$

$H(K) = 5$

$S \Rightarrow (1, 2, 3, 4, 5)$

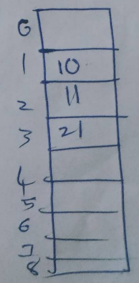
$S = 1 + 2 + 3 + 4 + 5$

$S = 15$



$K = 10, 21, 11$

$= 1, 3, 2$



④ multiplication method:

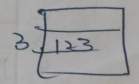
* Multiply the key by k by a constant A in the range $0 < A < 1$ and extract the fractional part of [kA]
* Multiply this fractional part by m and take the floor

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

$$h(k) = \text{floor}(m(kA \bmod 1))$$

M=10 k=123

floor (10 (123 * 0.61803 mod 1))



10 % 10 = 0

Collission:

collission is a problem it will occur when two or more keys are mapped with same location or same value in a hash table.

E. resolving techiques:

- separate chaining (open hashing)
- ~~closed~~ open hashing
  1) linear probing $(u+i) \% m$
  2) quadratic probing $(u+i^2) \% m$
  3) double hashing $(u+v*i) \% m$

$h(k) = \lfloor m(kA \bmod 1) \rfloor$

$h(k) = A[n\lceil$

$(u+0) \% m$

$(u+1) \% m$

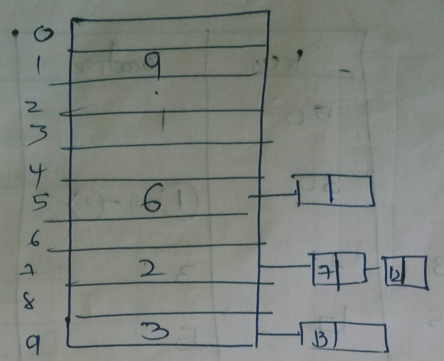
$(u+v*i) \% m$

→ use division method By separate chaining or closed hashing is closed addressing again.

A = 3, 2, 9, 6, 11, 13, 7, 12.

$h(k) = 2k + 3$

M = 10



Key	Location	Probe / remainder
3	$[(2\times3)+3]\%10 = 9\%10 = 9$	1
2	$[(2\times2)+3]\%10 = 7\%10 = 7$	1
9	$[(2\times9)+3]\%10 = 1\%10 = 1$	1
6	$15\%10 = 5$	1
11	$25\%10 = 5$	2
13	$29\%10 = 9$	2
7	$17\%10 = 7$	1
12	$27\%10 = 7$	2

---

use Division method & ~~open~~ closed addressing

linear probing: $(u+i)\%m$

⇒ $h(k) = k \bmod m$.

→ if the location is empty within in hash table

hashtable[value] = key

else apply ⇒ $(u+i)\%m$

u = location of collision

i = 0 to m − 1

→ repeat these 2, 3 steps till to all keys checked

50, 30, 21, 17, 11, 23

$h(k) = 2k+1$

$M = 10$

$h(50) = [(2\times50)+1] \% 10 \Rightarrow 1$

$h(30) = [(2\times30)+1] \% 10 \Rightarrow 1$

$a+i \% m$

$(1+0) \% 10 = 1$

$(1+1) \% 10 = 2$

$h(21) = (2(21)+1) \% 10 \Rightarrow 43 \% 10 = 3$

$h(17) = (2(17)+1) \% 10 \Rightarrow 35 \% 10 = 5$

$h(11) = (2(11)+1) \% 10 \Rightarrow 23 \% 10 = 3$

$a+i \% m$

$(3+0) \% 10 = 3$

$(3+1) \% 10 = 4$ ✓

$(3+2) \% 10 = 7$

$h(23) = (2\times23+1) \% 10 \Rightarrow 7$

key	location	probe
50	1	1
30	(1) coll – (2)	2
21	3	1
17	5	1
11	(3) coll – (4)	2
23	7	1

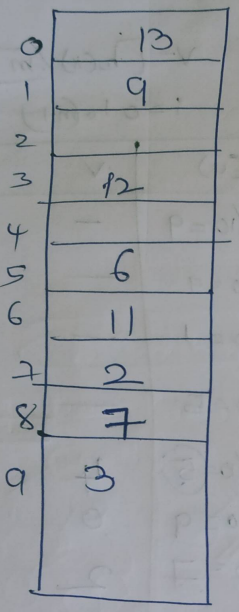
0	
1	50
2	30
3	21
4	11
5	17
6	
7	23
8	
9	

Note: elements(i) will be stored at first free location from $(a+i) \% m$ in a hash table

quadratic probing : $(h(k)+i^2)\%m$

A = 3, 2, 9, 6, 11, 13, 7, 12

$h(k) = 2k+3$ $\boxed{M=10}$



key	location	probes
3	$(2\times3+3)\%10 = 9$	1 search
2	$(2\times2+3)\%10 = 7$	1
9	$(2\times9+3)\%10 = 21$	1
6	$[2\times6+3]\%10 = 5$	1
11	$[(2\times11+5)\%10 = 5]$ (1)	2 no. of searches
13	$[(2\times13+3)\%10 = 9]$ (1)	2
7	$[(2\times7+3)\%10 = 7]$ (1)	2
12	$[(2\times12+3)\%10 = 7$	5

(11) $(5+0)\%10 = 5$

$(5+1)\%10 = 6$ ✓

(13) $(9+0)\%10 = 9$

$(9+1)\%10 = 0$ ✓

(7) $(7+0)\%10 = 7$

$(7+1)\%10 = 8$ ✓

(12) $(7+0)\%10 = 7$ ×

$(7+1)\%10 = 8$ ×

$(7+2^2)\%10 = 1$ ×

$(7+3^2)\%10 = 6$ ✓

$(7+4^2)\%10 = 3$ ✓

13, 9, —, 12, —, 6, 11, 2, 7, 3

Double hashing C

A = 3, 2, 9, 6, 11, 13, 7, 12

$h_1(k) = 2k+3$ $\quad m = 10$

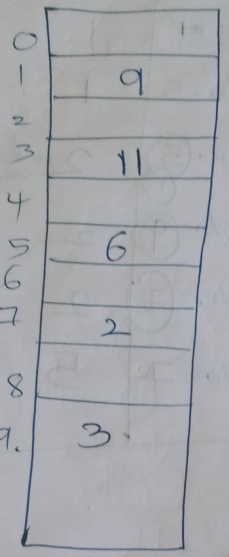
$h_2(k) = 3k+1$

h_2x is used when collision comes

→ insert k at first place from $(u + v*i) \% m$

$v = [h_2(k) \% m]$

$i = 0$ to $(m-1)$



key	location (u)	v	Probe
3	$(2\times3+3) \% 10 = 9$	—	1
2	$(2\times2+3) \% 10 = 7$	—	1
9	$(2\times9+3) \% 10 = 1$	—	1
6	$(2\times6+3) \% 10 = 5$	—	1
11	$(2\times11+3) \% 10 = 5$	4	3
13	$(2\times13+3) \% 10 = 9$	0	✗
7	$(2\times7+3) \% 10 = 7$	2	✗
12	$(2\times12+3) \% 10 = 7$	7	2

11) u = 5

$v = h_2(k) \% m$

$= (3\times11+1) \% 10 = 34 \% 10 = 4$

$\Rightarrow (5 + 4*0) \% 10 = 5 \% 10 = 5$ ✗

$(5+4*1) \% 10 = 9 \% 10 = 9$

$(5+4*2) \% 10 = 13 \% 10 = 3$ ✓

$(7+2*5) \% 10 = 23 \% 10 = 3$ ✗ $\quad (7+2*9) \% 10 = 5$ ✗

$(7+2*7) \% 10 = 21 \% 10 = 1$ ✗

$(7+2*4) \% 10 = 15 \% 10 = 5$ ✗

$(7+2*3) \% 10 = 13 \% 10 = 7$ ✗

$(7+2*6) \% 10 = 19 \% 10 = 9$ ✗

7 is not inserted

13) u = 9

$v = h_2(k) \% m$

$(3\times13+1) \% 10 = 0$

$\Rightarrow (9+0*0) \% 10 = 9$

$(9+0*1) \% 10 = 9$

So, 13 is not inserted

7) u = 7

$v = (3\times7+1) \% 10 = 2$

$(7+2*0) \% 10 = 7 \% 10 = 7$

$(7+2*1) \% 10 = 9 \% 10 = 9$ ✗

$(7+2*2) \% 10 = 11 \% 10 = 1$ ✗

$(7+2*3) \% 10 = 13 \% 10 = 3$ ✗

12) $u = 7$

$v = [(3 \times 12) + 1] \% 10$

$= 7$

$(u=7, v=7)$

$(7+7+0) \% 10 = 3$

$(7+7+1) \% 10 = 4$ ✓

Probe
1
1
1
1
3 ✗
✗
2

AVL tree

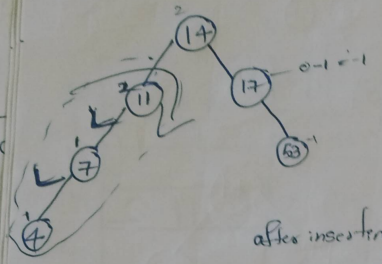
① It is a BST

② |height of left subtree − height of right subtree| ∈ {−1, 0, 1}

is called balance factor.

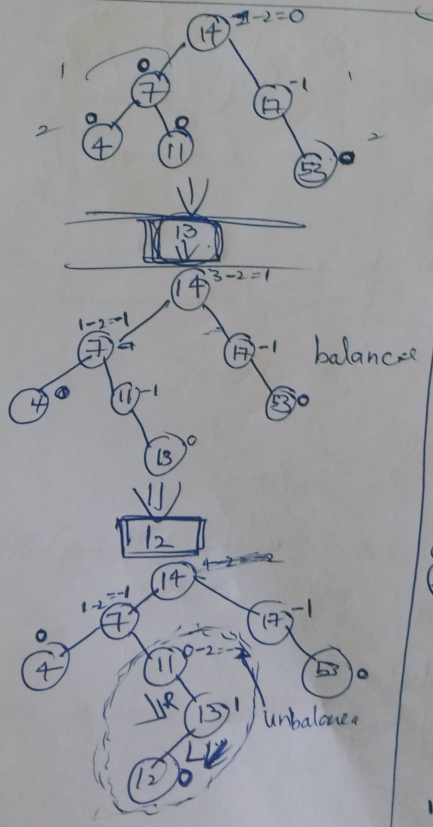
construct AVL tree by inserting the follow data

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

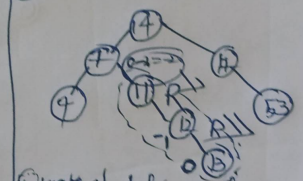


after inserting 4 at 11 b.f is 2, so it should be balanced

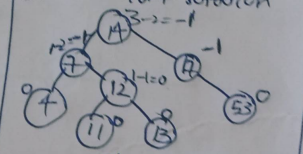
so make middle be parent & make right rotation



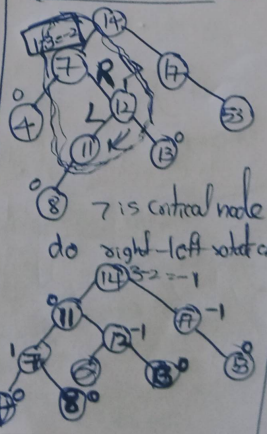
① make it right rotation.



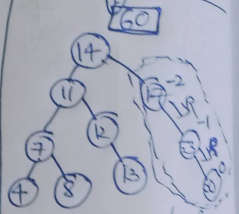
② make it left rotation



insert 8

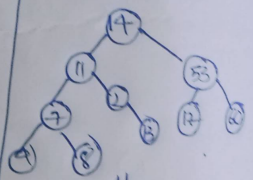


60

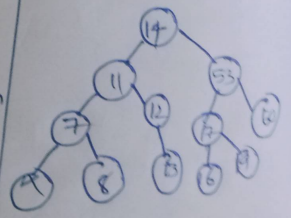


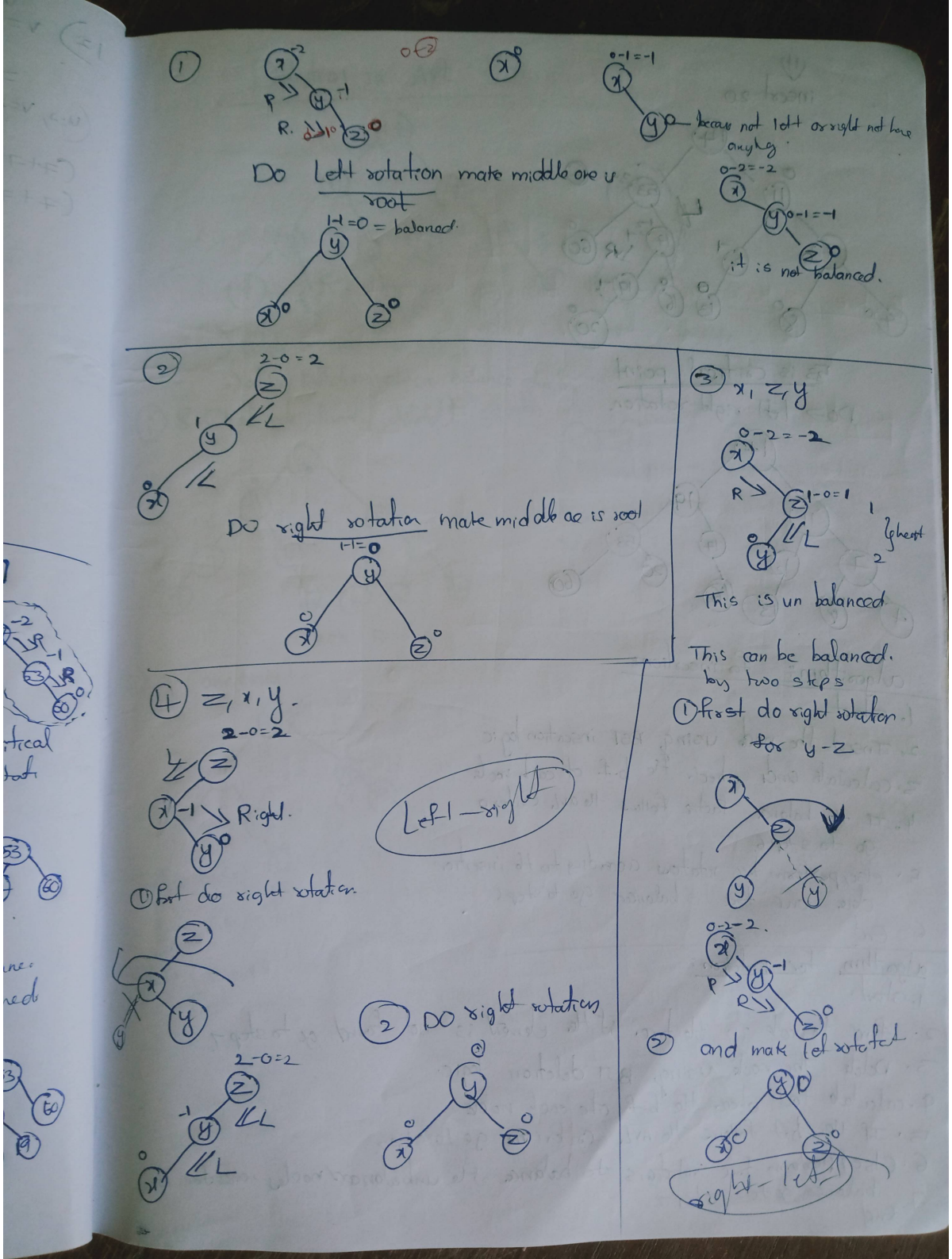
P is critical

make left rotation



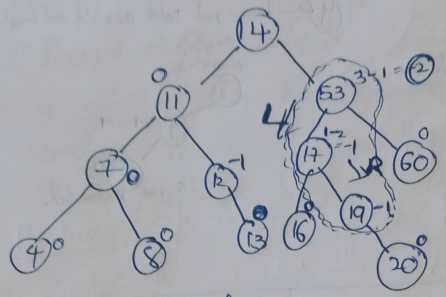
19 balanced and 16 balanced




1
x
y
z
P
R
Do Left rotation make middle one is root
1-1=0 = balanced.
0-1=-1
because not left or right not have anything.
0-2=-2
0-1=-1
it is not balanced.
2
2-0=2
1-1=0
Do right rotation make middle as is root
3
x, z, y
0-2=-2
1-0=1
height
This is un balanced
This can be balanced by two steps
1 first do right rotation for y-z
0-2=-2
and make left rotated
right-left
4
z, x, y.
2-0=2
-1
Right.
Left-right
1 first do right rotation.
2 DO right rotation
2-0=2

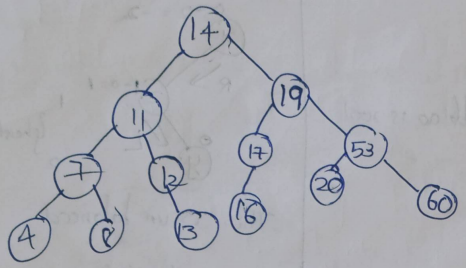
(ii)

insert 20



53 is critical point

DO → left right rotation.



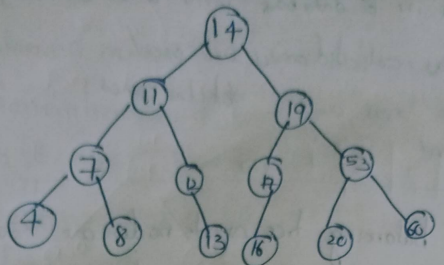
algorithm: for insertion:

1. start
2. Insert the node using BST insertion logic
3. calculate and check the b.f of each node
4. If the balance factor follows the AVL criteria go to step 6.
5. else, perform the rotations according to the insertion done. once tree is balanced go to step 6
6. End

algorithm for deletion:

1. start.
2. find the node in the tree. if the element is not found go to step 7
3. Delete the node using BST deletion logic.
4. calculate and check the b.f of the each node
5. If the b.f follows the AVL criterian go to step 7.
6. Else, perform the rotations to balance the unbalanced nodes. once tree is balanced go to step 7
7. End

Deletion in AVL tree

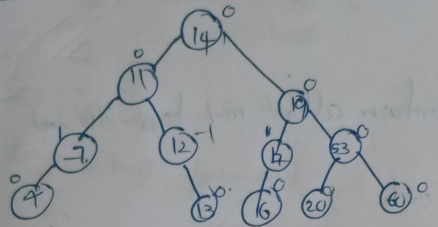


i) Delete is same as BST

ii) after Deletion check Balance factor

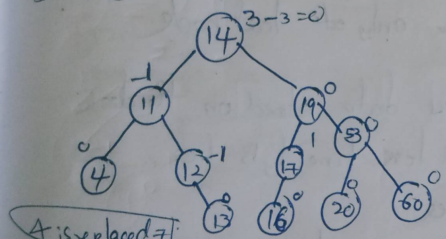
8,7,11,14,17

(i) 8 (leaf node, directly deleted)



check B.f

(ii) 7 (one child)



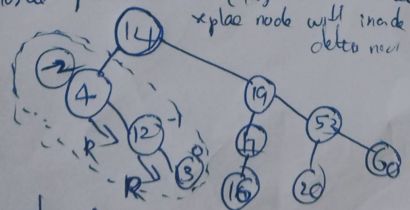
check B.f

(iii) 11 (Two children)

Find inorder predecessor (largest in left subtree)

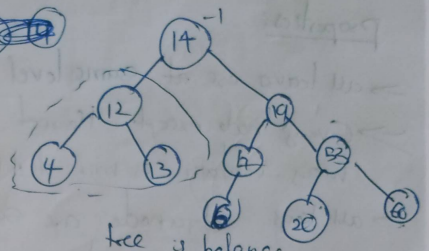
xplace node with inorder predecessor

delete node



4 is critical. Check B.f

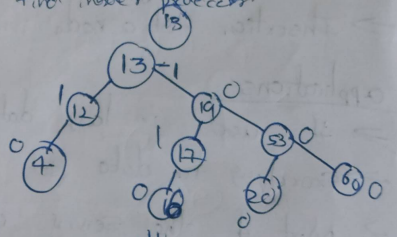
Do left rotation.



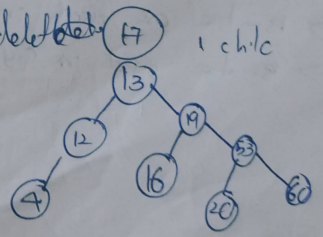
tree is balance

(iv) 14 (Two children)

Find inorder predecessor



(v) delete 17 (1 child)



balance

Application of AVL:

1. It is used store huge records in a database and also to efficiently search.
2. For all types of memory collection sets, dictionaries avl used
3. It is applied in corporate areas and stock line games
4. In software that needs optimized search

B-tree: is known as a self-balancing tree as its nodes are sorted. in the inorder traversal. Unlike the binary trees, in B-tree a node can have more than 2 children.

Properties:

- all leaves are at same level
- Every node except the root must contain atleast m-1 keys. The root may contain a min of 1 keys
- all keys of a node are sorted in order
- B-tree grows & shrinks from the root
- time complexity search, insert & delete is O(log n).
- Insertion of a node in B-tree happen only at leaf node

applications:

- it is used in large data base to access data stored on the disk
- Searching for data set can be achieved less time by B-tree
- Most of the servers also use B-tree
- B-tree are used in CAD systems to organize data.
- B-tree used in cryptography, computer networks

B-tree

→ balanced m-way tree (m = order 1, 2, 3, 4, 5)

→ Generalization of BST in which a node can have more than one key & more than 2 children

→ maintains sorted data.

→ all leaf node must be at same level

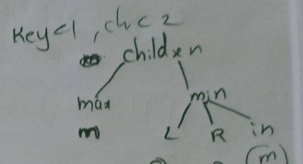
→ B-tree of order m has following properties

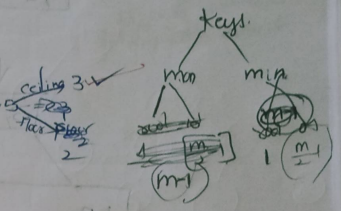
→ Every node has max m children

→ min children : leaf → 0
root → 2.
internal node → $\lceil \frac{m}{2} \rceil$

→ Every node has max (m-1) keys (max key)

→ min keys; root node → 1
all other node $\lceil \frac{m}{2} \rceil - 1$



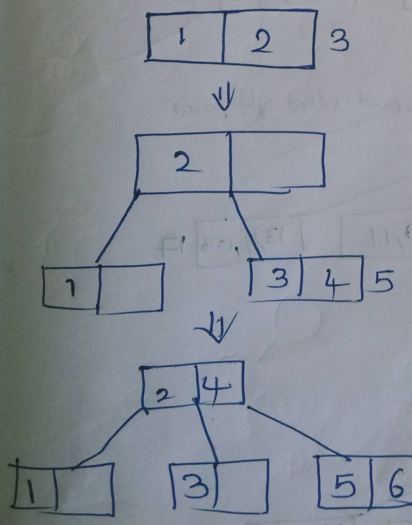


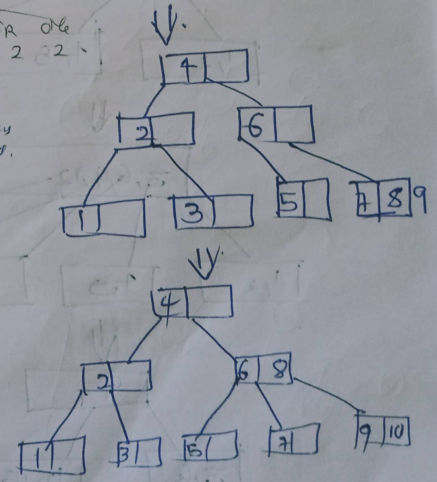
1) Create a B-tree of order 3 by inserting value from 1 to 10

m = 3

max keys = (m-1) = 2

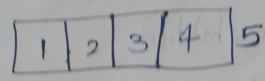
m=3
children → max min
m=3. L R one
0 2 2.
keys = max 2
Have or 3 keys
choose only 2 key
overflowed.



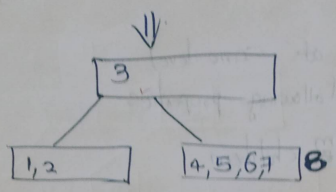


Create a B tree of order 5 1 to 20

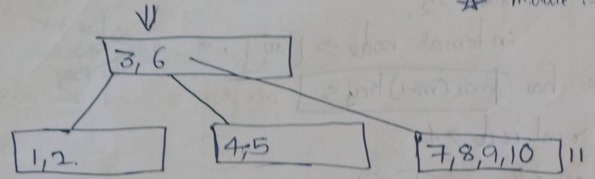
m = 5
max children = 5
max keys = m-1 = 4

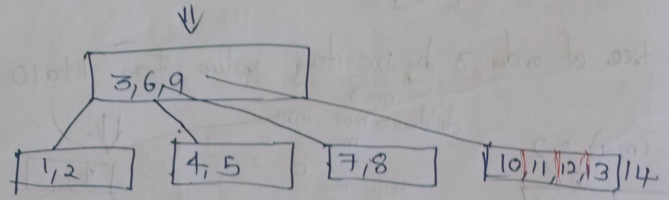


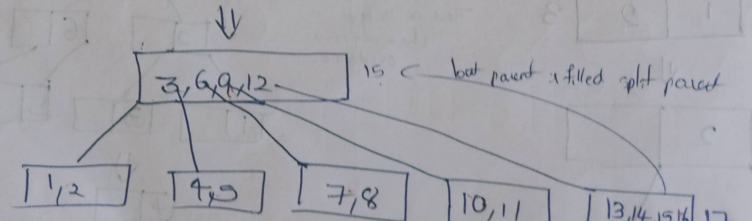
5 is not inserted to key as only 4 split the node.
By considering median middle go up

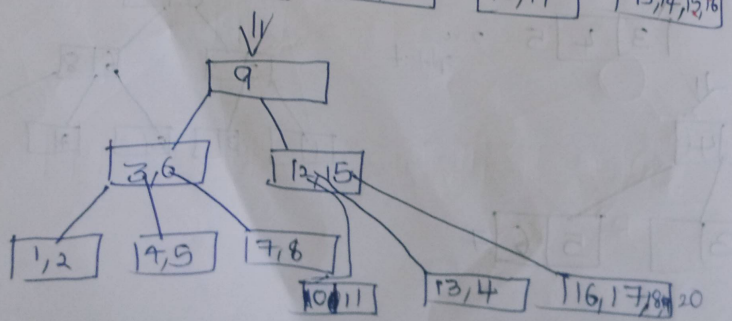


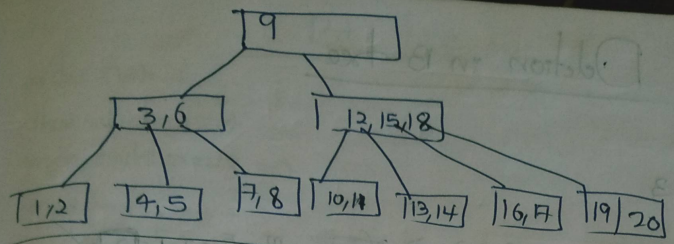
new keys are inserted in leaf nodes only
middle is 6 one level or parent node.









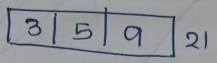


construct a B tree of order 4

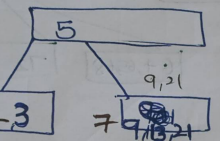
5,3,21,9,1,13,2,7,10,12,4,8

m = 4

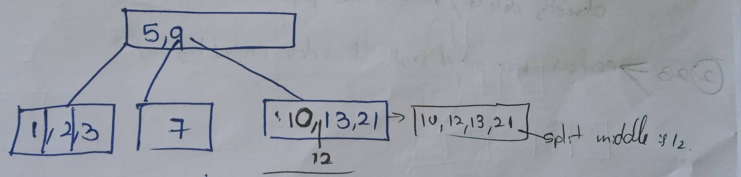
keys = m-1 = 3

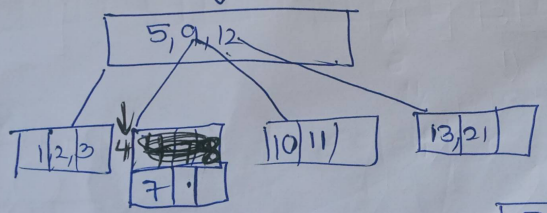


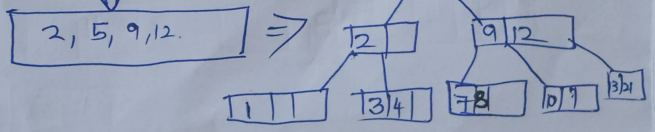
4 in 4 orders: 5, or 9 will take as middle as with.



9 is middle







Deletion in B-tree

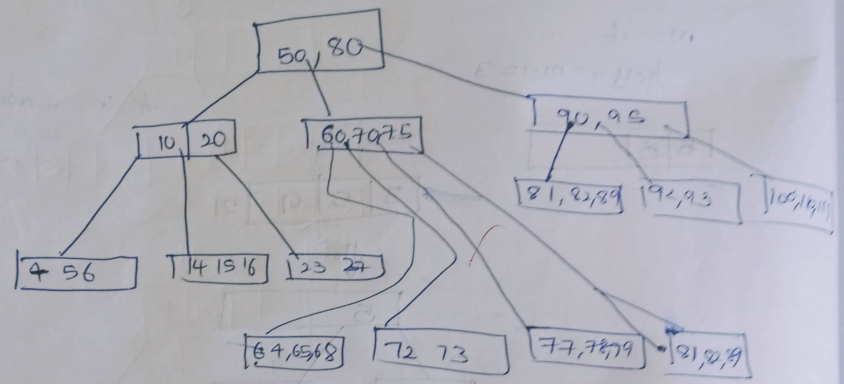
order (m) = 5

min children = $\lceil \frac{m}{2} \rceil = 3$

max " = 5

min keys = $2(\lceil \frac{m}{2} \rceil - 1)$ ~~$\Rightarrow 2(\frac{5}{2}) \Rightarrow \frac{m}{2} = \frac{5}{2} = 3-1 = 2$~~ ✓

max " = 4



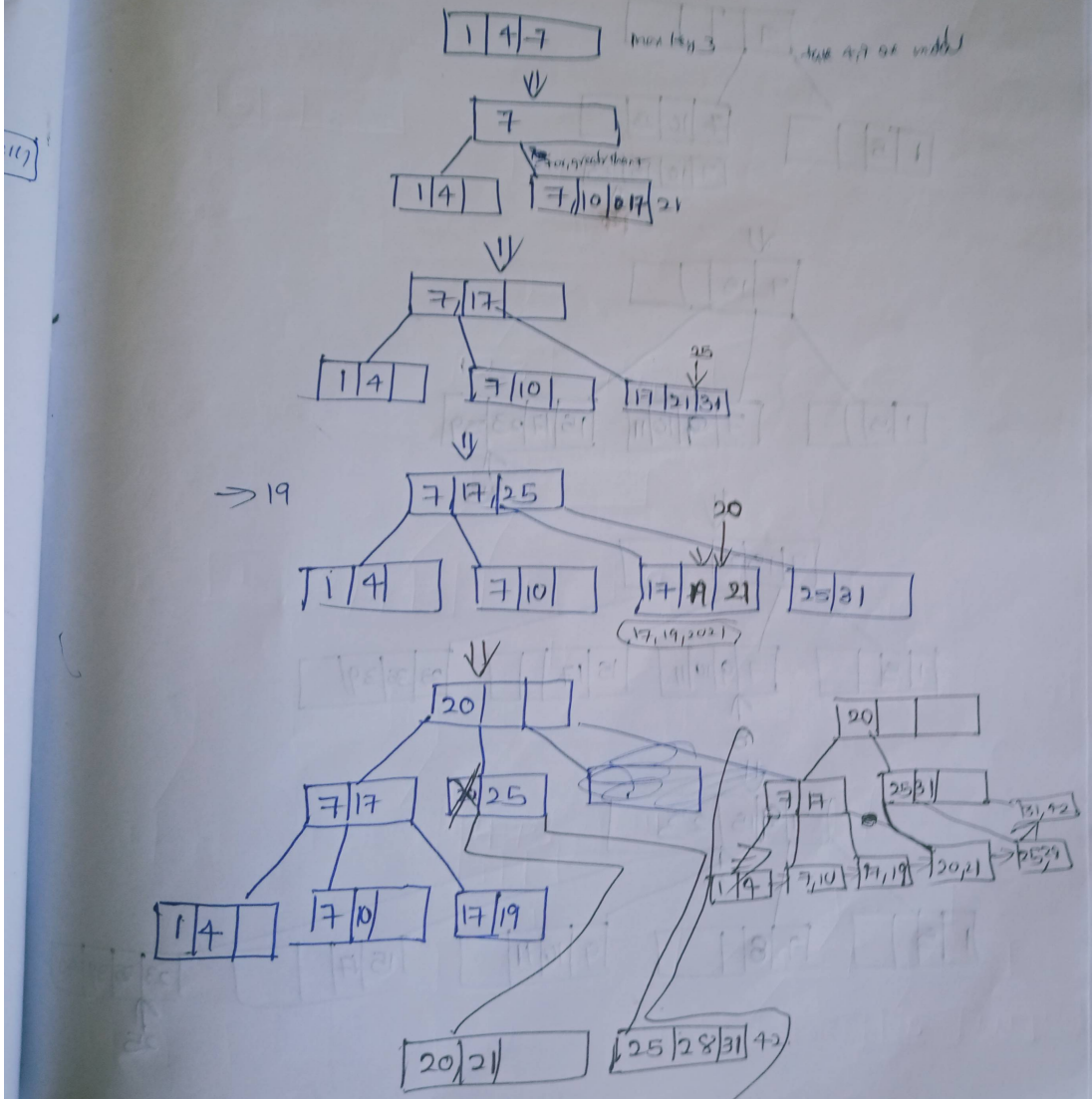
① 64 ⇒ contains more than min no. of keys
directly delete, having B tree property

② 23 ⇒ contains min no. of keys, if we delete 1 only 1 key exist

B+ tree

order (m) = 4
max children = 4
min children = $\frac{m}{2} = 2$
max keys = (m-1) = 3
min keys = $\lceil \frac{m}{2} \rceil - 1 = 1$

1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42

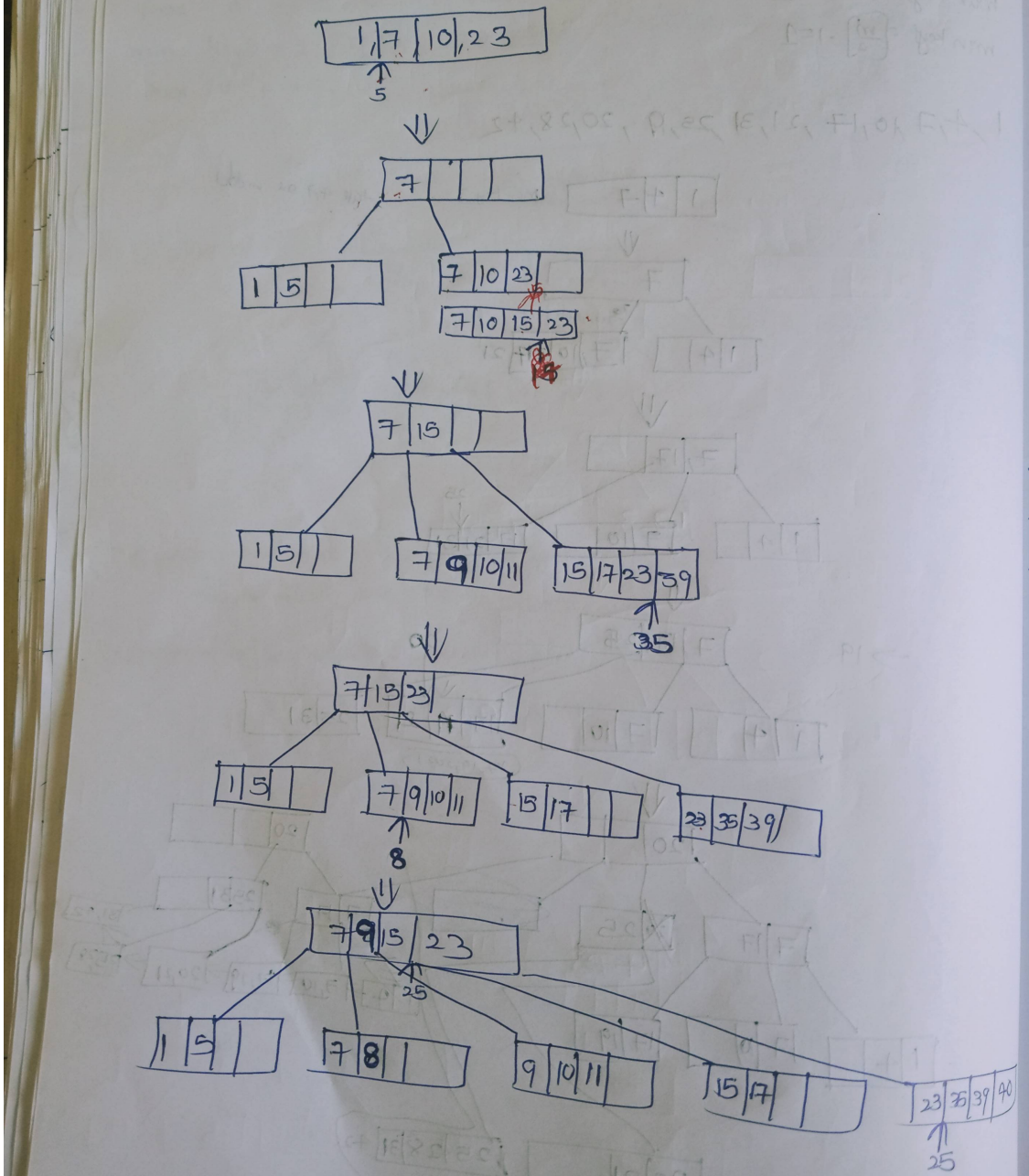


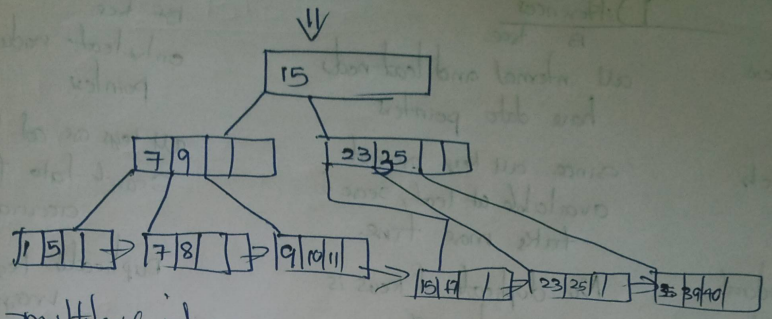
B+ tree

order 5

7, 10, 1, 23, 5, 15, 17, 9, 11, 39, 35, 8, 40, 25

max children = 5
min " = $\lceil \frac{5}{2} \rceil = 3$
max keys = 4
min key = 2





applications: ⇒ multilevel indexing
⇒ Database
⇒ faster operations (insert, delete, search)

B+ tree: A B+ tree is an advanced form of a self-balancing tree in which all the value are present in the leaf leaves

properties:
→ all leaves are at the same level
→ the root has at least 2 children
→ Each node except root can have a max of 'm' children & at least $\frac{m}{2}$ children
→ Each node can contain a max of m-1 keys and min $\frac{m}{2}-1$ keys

Differences

	B tree	B+ tree
Pointers	all internal and leaf node have data pointers	only leaf nodes have data pointers
Search	since all keys are not available at leaf, search takes more time.	all keys are at leaf node, search data faster & accurate
Redundant keys	No duplicate of keys is maintained.	Duplicate keys are maintained
Insertion	Insertion take more time	Insertion is easier.
Deletion	Deletion of internal node is complex, go have more transformation	Deletion of any node is easy because all node found at leaf
leaf node.	not stored as structural linked list	stored as structural linked list.
access	sequential access to node is not possible	possible
height	for a particular no. node height is larger	hgt is less than B tree for same no. of node
application	used in Database search engine	multi level indexing Database indexing