29 / August 2023

Algorithm Design + Analysis

Code flexibility.

Test and Debugging : parameter & return types are known before implementation

Offseating: ~~binding~~ to make header file public and hiding implementation

Templates.

→ Function Overloading

- Different data type of parameter
- Different number of parameter.
- Different order of parameters

∴ during compile time (static binding)

compares parameters and figures which overloaded function to call.

→ Templates is called generic programming

- functions and classes where datatype used would be defined later
- datatype is provided as a parameter

```
* template <typename T>
  void print ( T a[], int size)
  {   for (int i=0; i<size; i++)
        cout << a[i] << endl; }

  int main()
  {   int a[10] = {1,2,3,4,5,6,7,8,9,10};
      int s = 10;
      print (a , s);   }
```

```
* template < class X >
  X sum ( X a, X b)
  {  return a+b; }

  main()
  {  string = sum ("xyz", "abc");
     const char *z = sum ("xyz","abc");
  }
```

x doesn't matter if you write class / typename.

x similarly you can write X, T, ... or any other letter

∴ I hope result is xyz abc

sum(7.5, 5)
↳ adds as double.
as double has higher precedence than integer

GAMA

LOL

29/Aug/2023.

```
* template <class M>
  bool isEqual (M a, M b)
  { if (a == b) return 1;
    return 0; }

main()
{ cout << isEqual (5,5);          // 1.
  cout << isEqual ("abc","abc");  // 0
  { cout << isEqual ("XYZ","abc"); // 0
  if (strcmp ("a","b"))
      cout << "WAH";              // lexographic order mein compare.
  else cout << "OH"; }
```

check location in RAM

* Template specialization | User Defined Specialization

```
template <>                 // datatype for which specialization works
                            // → represents template specialization
bool isEqual <char *> (char *a, char *b)
{ return (strcmp (a,b) == 0); }
```

* if there a way that during call time we can identify data type

```
main()
  isEqual <int> (7.5, 5.5);   // takes as integer & ≠ double
```

```
template <typename X>          * function without parameter
X fun ( )
{ X a;
  cin >> a;
  return a; }

main()
{ int a = fun <int> (); }
```

* explicitly mentioning datatype.
* specifying datatype which it is working with.

compares character and their corresponding indexes ASCII values

matches a - a = 97 - 97 : 0

p - p

p - p

i = ?

// method 1.

template <typename T>

class List. {

// can use T type data type in this class

T *an;

// now whole class is generic

}

// method 2

class student

{ char *name;

public:

friend operator == (const student &g)

{ // }

// naming is equal is defined

// can use template specialization.

main()

{ student a, b;

isEqual (a, b);

}

GAMA

31 / Aug / 2023.

→ Class Template.

```
template <class T>.
class Vector {
        private:  int size;
                  T* ptr;
        public:
                  Vector <T> (int = 10);
                  ~Vector<T>();   };
```

Out of line

```
template <class T>
return-type Classname <T> :: function-name { // code }
```

exit(1) = unsuccessful termination of program.   x exception handling.

```
main() {
  Vector <int> i(2);
```

User-Specialized template.

```
template <>
class Vector< char* > {
  private:
           int size; char** ptr;
  public:
           // vector <char*>(int = 10);        // Both works.
              Vector (int = 10);   };
```

5 /Sept/ 2023

Complexity Analysis.

→ Time complexity: amount of time that an algorithm needs to run to completion

→ Space complexity: amount of memory an algorithm needs to run.

The running time depends on input (data provided & its size).

Worst case scenarios allow us to determine the lowest performance.

Analyzing a solution

Works n times ← i++ ⇒ i = i + 1; 2 instructions.

```
int SumArray(int A[], int n) {
    int s=0;                          // 3
    for (int i=0; i<n; i++)           // 2, 3, 4
        s = s + A[i];                 // 5, 6, 7
    return s; }                       // 8
```

Total time: 1+1+1+(n+1)+n+n+n+n = 5n+4.

i<n works till n-1 at everytime and then last check when i?n. So total instructions n+1.

instruction 4, 5, 6, 7 works once per each iteration of for loop n iteration

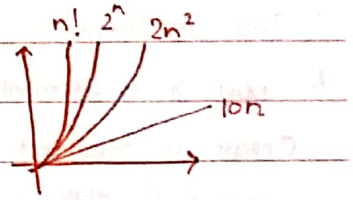
f(n) = 5n+2

f(50) = 52 f(500) = 502

↳ has linear growth rate

f(n) = 100.

f(10) = 100 f(1000) = 100 ↓ has constant growth rate.



* lower order terms and constant have negligible effect when taken to ∞

so in 5n+2 actual growth rate is n.

Arrays ADT and C++ Implementation.

-consecutive memory locations + same data type + direct Access [] + binary search $\log_2(n)$

* Selection Sort (Complexity $O(n^2)$ - time)

```
void sllect (int data [], int size)
{   int temp;
    int max_index;
    for (int rightmost = size-1; rightmost >0; rightmost--)
    {   max_index=0;
        for (int current=1; current <=rightmost; current++)
            if (data[current] > data[max_index]
                max_index = current;
```

GAMA LOL

```
if (data [max_index > data [right most ]{
        temp = data [max_index ];
        data [max_index] = data [rightmost] ;
        data[rightmost ] = temp ; }
}
```

Bubble Sort	$\frac{n(n-1)}{2} = O(n^2)$ comparisons	$\frac{n(n-1)}{4} : O(n^2)$ swa
Selection sort	$\frac{n(n-1)}{2} : O(n^2)$ comparisons	$3(n-1) = O(n)$ swaps.
Insertion Sort	$\frac{n^2}{4} + O(n) = O(n^2)$ Comparison	$\frac{n^2}{4} + O(n) : O(n^2)$ Su

Insertion Sort.

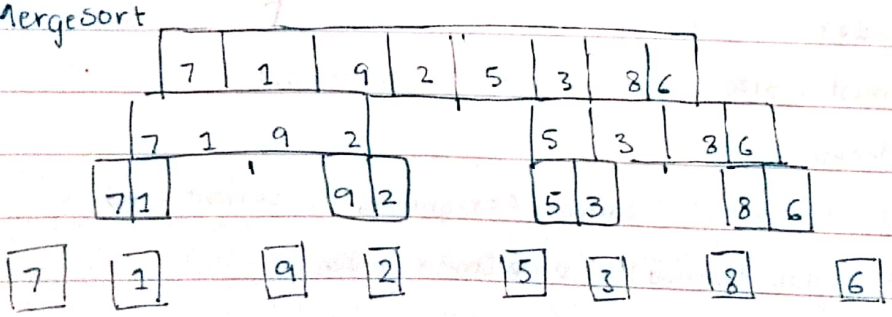
A list of n elements will take at most n-1 passes to sort data.

Create a sub-list

```
template <class Item>.
  void  insertionsort (Item a[], int n){
     int i,j;
      for (~~int~~ i = 1 ; i < n ; i++ )
          Item temp = a[i];
       for (j = i ; j > 0 and temp < a[j-1] ; j--) {
              a[j] = a[j-1]; }
       a[j] = tmp;
  }}
```

Best case : O(n) time.

Mergesort



now swap and sort each block.

levels are decided by $\log_2(n)$ eg: 8 elements then 3 $\log_2(8)=3$.

16 elements the $\log_2(16)=4$

n as final array will be of n size. $\therefore n\log_2(n)$

merge sort uses a lot of space. **Space Complexity**.

Q. How to select the right pivot for Quicksort?

```
int X=0
cin>>X;
const int Y=X ; Runtime allocation . ; runs on other compiler
```

in stack we require compile time allocation for array size [VS studio].

List

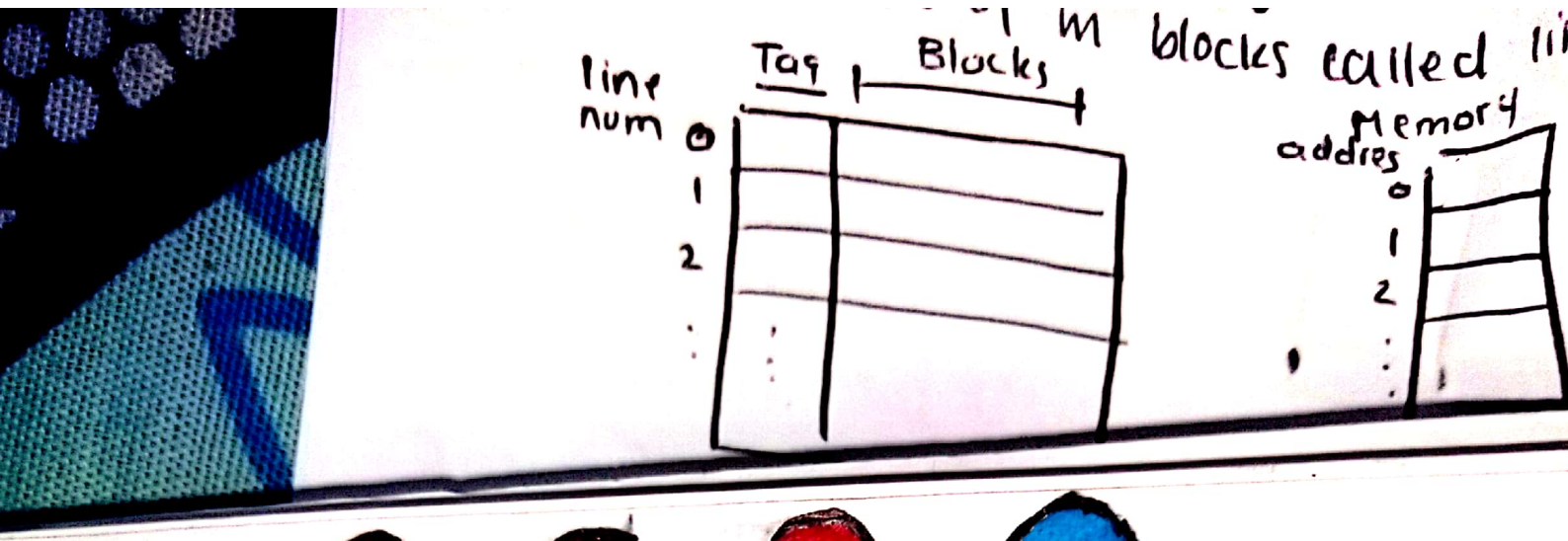
- collection of items of same type. Elements can be inserted, deleted and accessed at any position
- sequence of n elements arranged from i=0 or more elements.
- number of elements = length of list.
- list can have be list of lists; can be concatenated together
- can be split into sub-lists

→ List as an ADT :

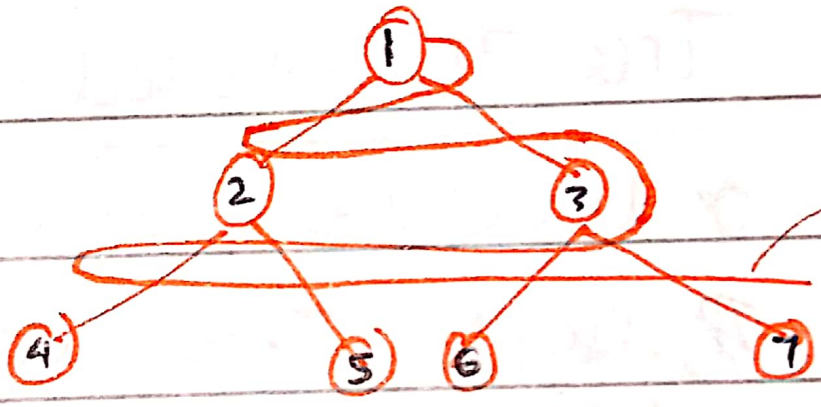
1. Homogeneous.
2. Sequential elements
3. Can be finite length or not.

Basic Operations

1. Create list
2. Determine whether list is empty.
3. Finding current size of list.
4. Destroy, Insert, Delete,



blocks called li

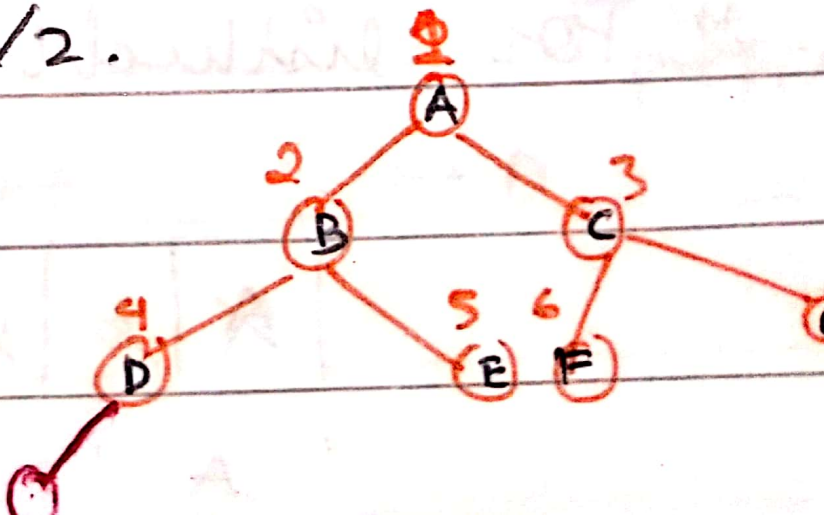


level or
transve

ndex
ta + 1

parent = index data / 2.

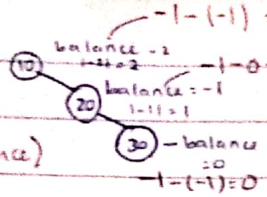
5	6	7
E	F	G



Avl

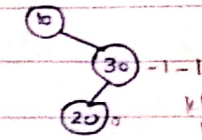
14 Nov 2023

(a) Insert 10, 20, 30

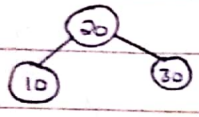


(left rotate to balance)

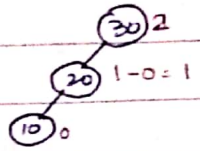
(b) Insert 30, 30, 20



(c) Insert 20, 30, 10

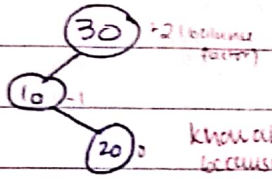


(d) Insert 30, 20, 10

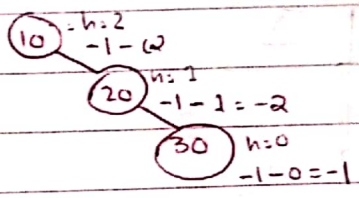


Right Rotate

(e) Insert 30, 10, 20



(f)



How to calculate balance factor?

balancing allows searching, insertion, deletion in O(log n)

balanced tree - each node has balance of -1 and 1 (in case of no mod)

balance difference can't > 1

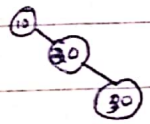
|left - right| height ≤ 1.

- LEFT ROTATE (Single Rotate)

+ if balanced out because of node → right → right

- Right - Left (b)

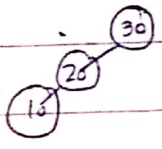
+ right rotation on node → right



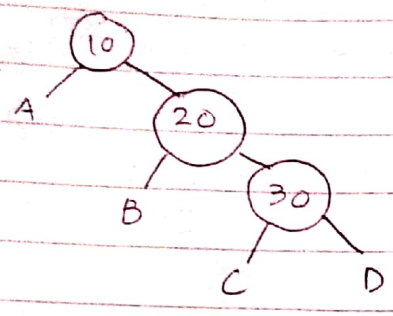
+ now call left rotation

- Left - Right (z)

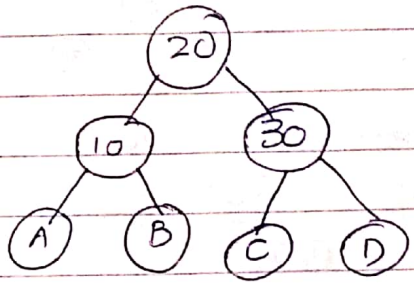
+ left rotation on node → left

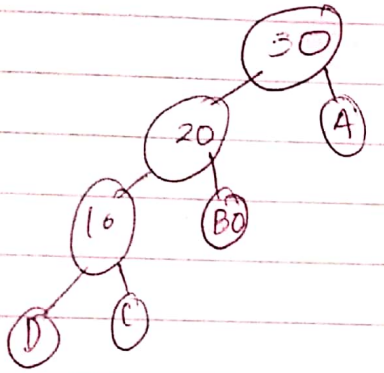


+ now call right rotation

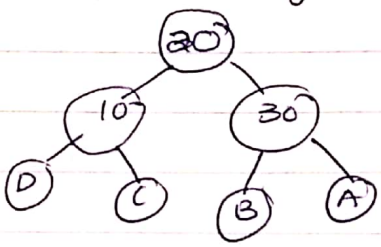


+ assign B to 10 → right
+ attach 10 to 20 - left-

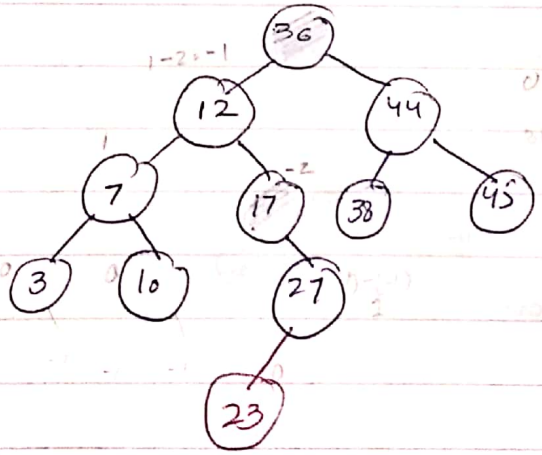




+ assign B to 30 → left
+ assign 30 to right of 20 :

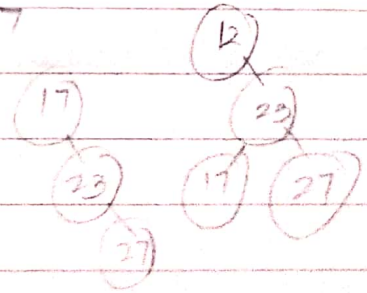


Insert 23



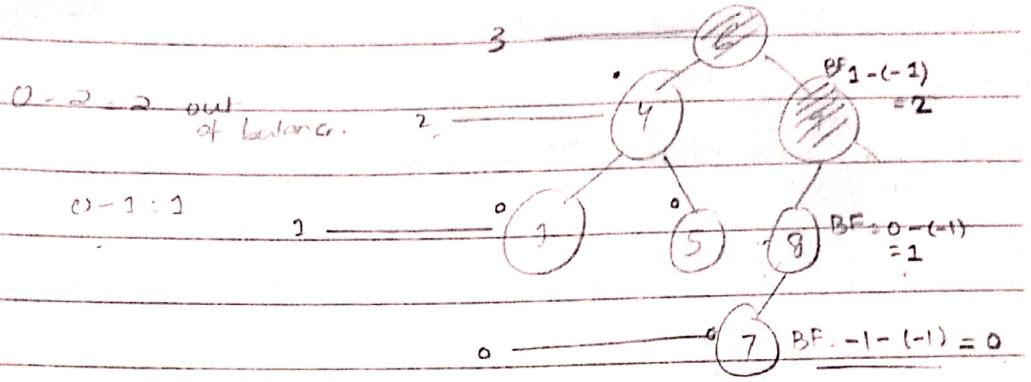
fix the unbalanced at 17

- 17 → right rotate



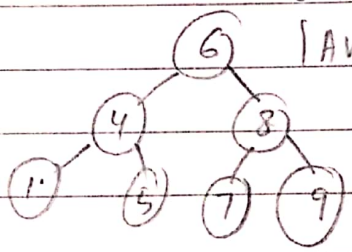
- left rotate

(NOT AVL) BF 0-2=2.



9 and 6 are out of balance.

so rotate right



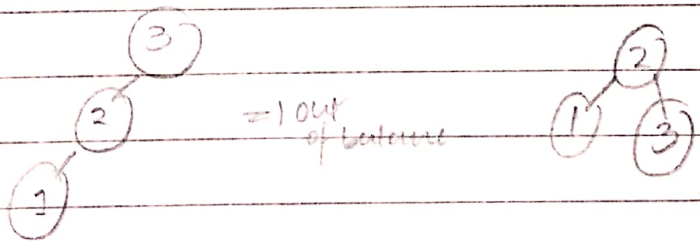
[AVL]

another way check height

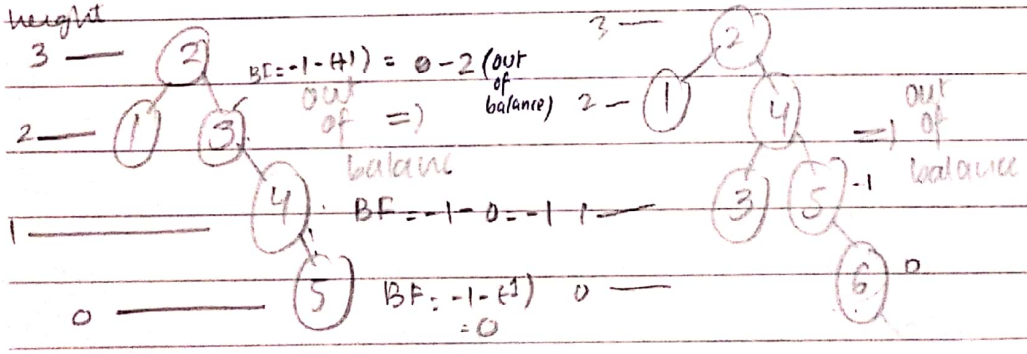
Insert

3, 2, 1, 4, 5, 6, 7

* create function
+ balance factor
+ height



height



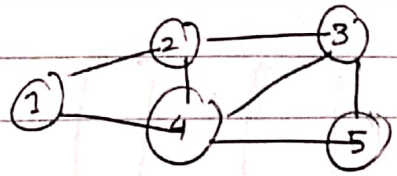
Representation of Graph

1 — Adjacency Matrix — n × n matrix (n = vertices)

2 — Adjacency List.

(1)

	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

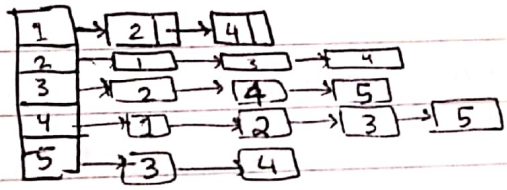


space complexity $O(n^2)$

a[i][j] = 1 if i and j are adjacent otherwise 0.

(2) Adjacency List [Linked List]

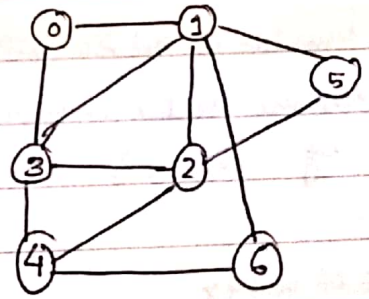
each vertex has a linked list. space complexity $O(n+2e)$ e is edge.



better to use adjacency matrix for dense graph

better to use adjacency list for sparse graph.

Graph Transversal



— BFS (start at any node as root)

- use queue [FIFO]

considering (0) as root.

0	1	3	2	5	6	4

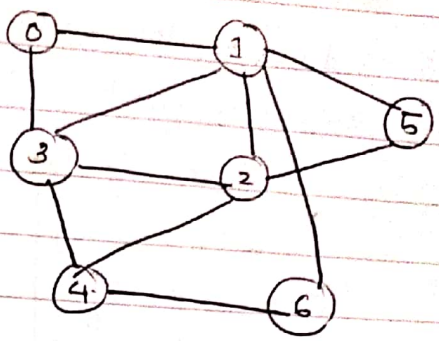
result

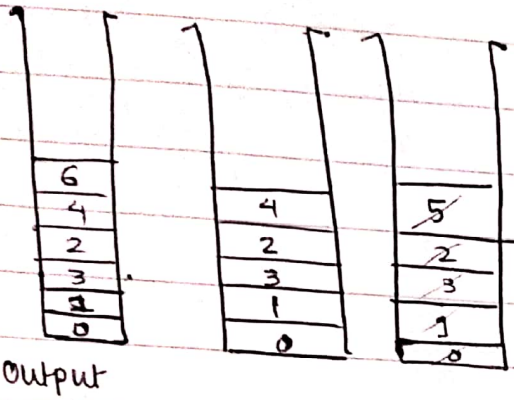
0 1 3 2 5 6 4

```
void BFS (Graph g, int s)
{
    bool visited [max_vertices];
    for (int i=0; i<g->vertex; i++)
        visited [i] = 0;
```

DFS transversal.

use Stack (take any node as root)





Output

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

push one adjacent node of node.
at 6 no unvisited nodes — now backtrack.
pop 6
now check if 4 has any unvisited vertex if no then pop 4
pop 4
check if 2 has any unvisited vertex yes ie 5 push 5, print 5, ~~pop~~
check if any nodes of 5 is unvisted no then pop 5
" " " nodes of 2 is " no pop 2
pop 3
pop 1
pop 0.

Undirected Graph G(V, E) (E: edge, V: vertex)

max number of edges in undirected graph is $O(|V^2|)$

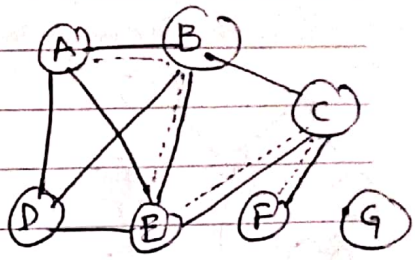
Assume: vertex is never adjacent to itself.
eg $\{v_0, v_1\}$ will not define an edge.

= Degree of vertex is number of adjacent vertices.
using above eg deg(0) = 2

Vertex adjacant to a given vertex are called Neighbours

= Subgraph
1. Subset of vertices.
2. Subset of the edges that connected that subset vertices in the original graph.

Path: it is an undirected graph is an ordered sequence of vertices
- consecutives vertices connected through edges.
- length of path = number of edges.



eg A to F

Path (A, B, E, C, F)

length = 4

Simple Path
- no repetitions except first and last vertex.

Simple Cycle
- simple path of at least two vertices with first & last vertices equal

Loop
- edge from a vertex to itself

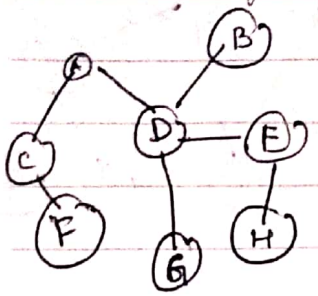
Connectedness
- a graph is connected when there exists a path from every vertex to every other vertex

TREES
- a graph is a tree if
  1. Graph is connected
  2. unique path b/w any two vertices

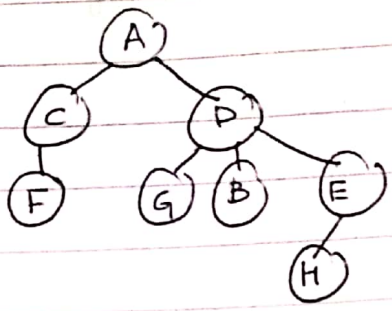
This results in
1. number of edges $|E| = |V| - 1$
2. graph is acyclic - contains no cycles.
3. adding one more edge creates a cycle.
4. Removing one edge creates disjoint non-empty sub-graphs

Any Tree can be converted into rooted tree by

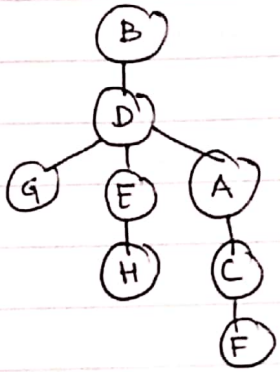
1. Choosing vertex to be the root
2. Defining its neighbour vertices as children.



root A



Choosing Root B



Forest

– graph has no cycles.

Results:

1. edges $|E| < |V|$
2. number of trees is $|V|-|E|$
3. Removing any one edge adds one more tree to the forest

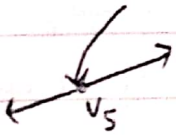
Directed Graphs

ordered pairs of edges (v_j, v_k) denotes $v_j \longrightarrow v_k$

In and Out Degree:

Out : Outgoing edges

In : incoming edges.



out : 2
in : 1

Path : in directed graph it is ordered sequence of vertices.

Connectedness:

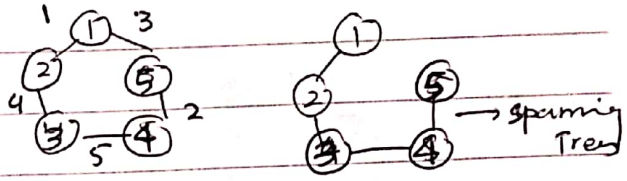
Two vertices v_j, v_k are said to be connected if there exists a path from v_j to v_k.

- graph is strongly connected if there exists a directed path b/w any two vertices
- " weakly " " " exists a path b/w any two vertices that ignores the direction.

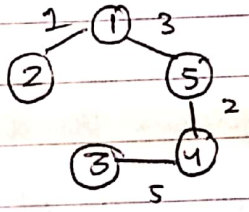
Adjacency Matrix of undirected graph is symmetric
$a[i, j] = a[j, i]$

Minimum Spanning Tree:

⇒ Spanning Tree
has same number of vertices
Edges = |V| − 1.

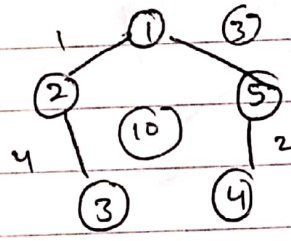


⇒ Minimum
- should not have any cycle.
- should not be disconnected.



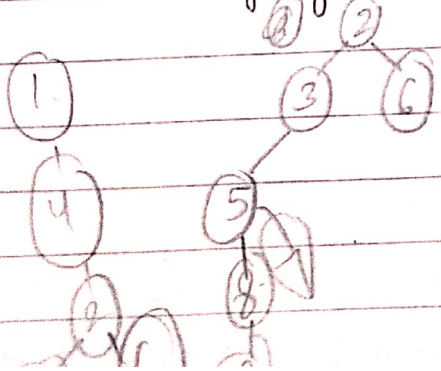
1 + 3 + 2 + 5 = 11.
this is not minimum spanning tree

Algorithms for obtaining MST
- Kruskal's
- Prim's
- Boruvka's



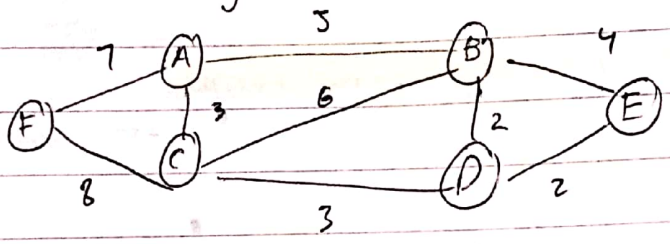
This is minimum.

Prim's [graph must be connected] faster for dense graph → where no. edges > no. ver
- remove all loops and parallel edges.
- choose any vertex as root
- check all outgoing/incident edges from node (choose min)

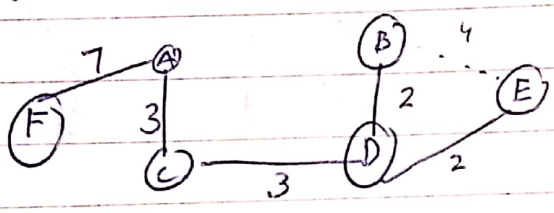


Kruskal's Algorithm.

1. remove all loops + parallel edges
2. arrange all edges according to increasing order of their weight
3. choose edge having minimum & connect those edges ensuring no cycles.

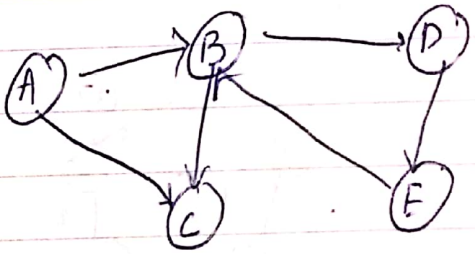


BD 2
DE 2
AC 3
CD 3
BE 4 X forms cycle
AB 5 X
BC 6 X
AF = 7
CF = 8

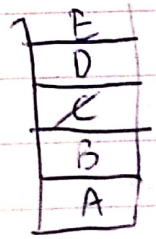


← This is MST which has |V| - 1 edges.

How to detect cycle in Directed Graph using DFS.



flag -1 unvisited 1 visited & popped
0 visited in stack



A B C D E

at E you will check B which has flag 0 thus indicated cycle.

B-Tree :

balanced m-way tree.

each node can have more than 1 key.

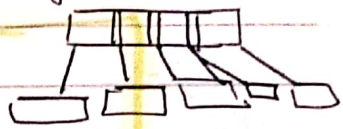
sorted data

all leaves are at same level.

→ B-tree of m order.

- each node has max m children.
- min children : leaf → 0
  root 2.
  internal node $\lceil \frac{m}{2} \rceil = 5/2 = \lceil 2.5 \rceil = 3$.

eg 5



- every node has max (m−1) keys
- min keys : root → 1
  all other nodes $\lceil \frac{m}{2} \rceil - 1$.
- Insertion happens in leaf nodes

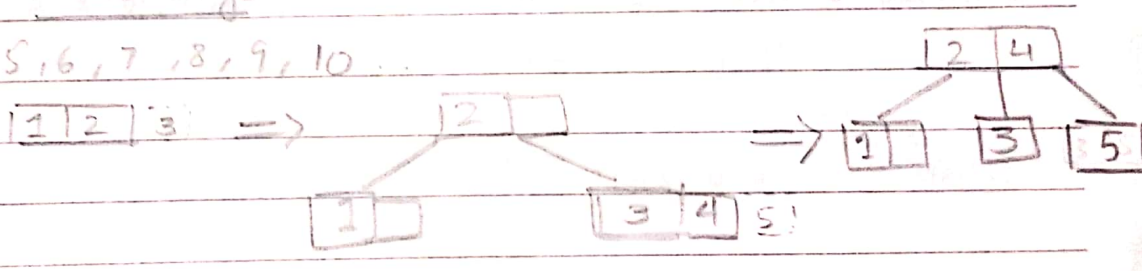
Insert :

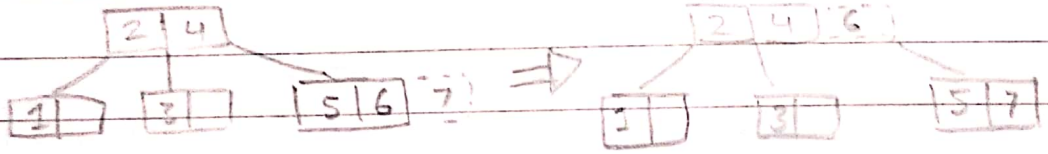
Eg · M=3 keys min $\lceil \frac{3}{2} \rceil - 1 = 1$.

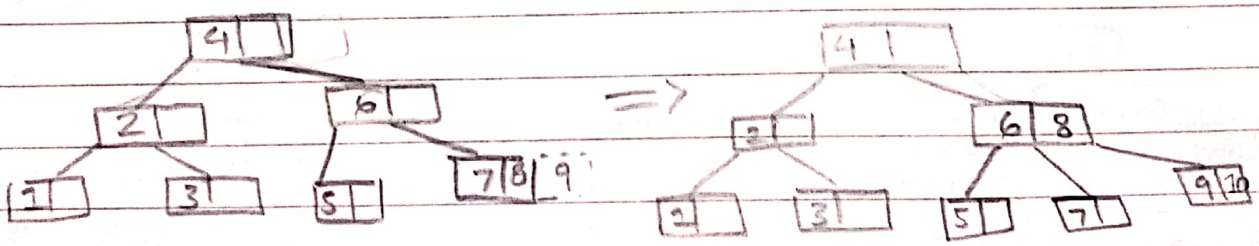
max children 3.

max keys m−1 = 2

1, 2, 3, 4, 5, 6, 7, 8, 9, 10.





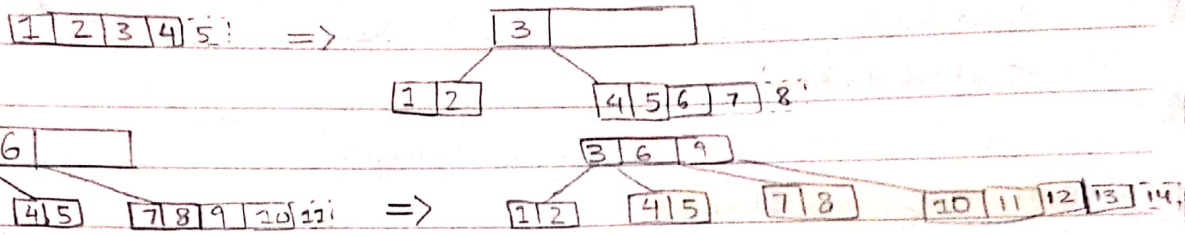


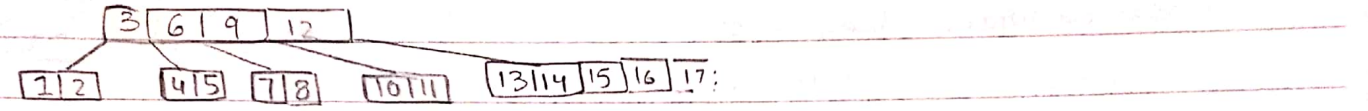
GAMA

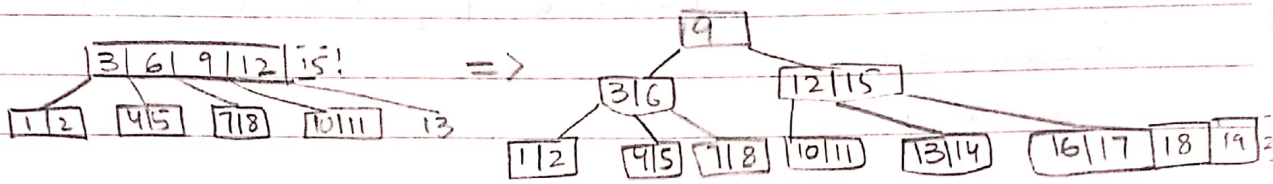
LOL

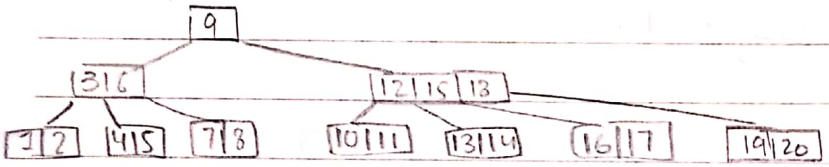
M = 5 1 — 20 values.

max child = 5 max keys = 5-1 = 4

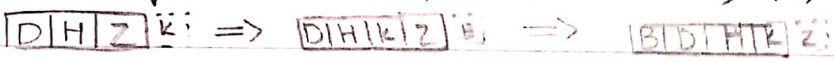


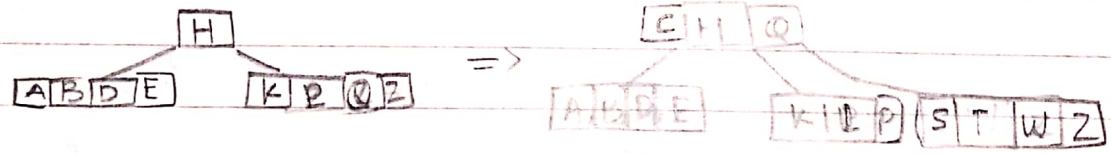


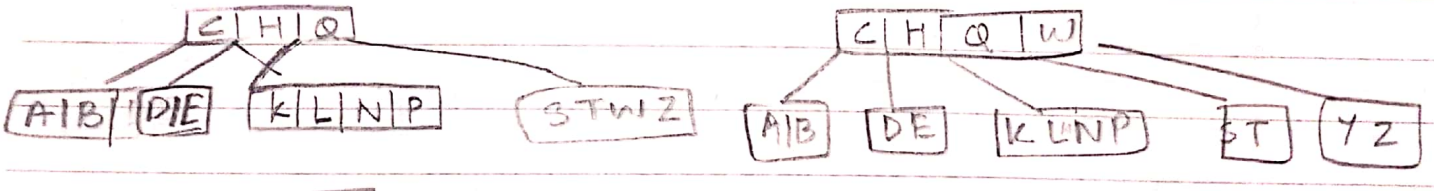


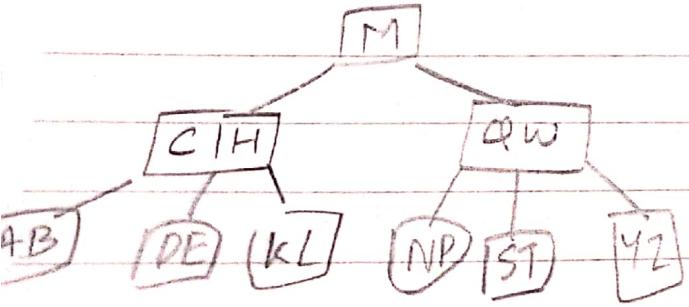


Construct B-tree of order 5 [D, H, Z, K, B, P, Q, E, A, S, W, T, C, L, N, Y, M]



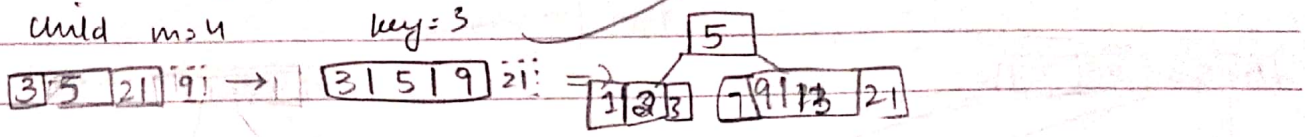




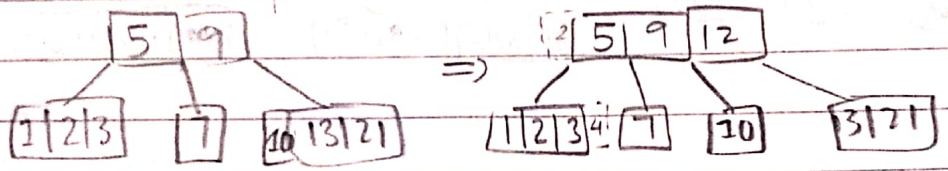


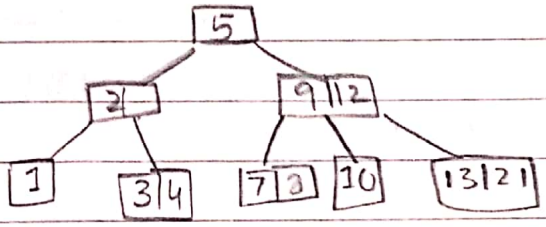
Order 4

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



child m=4 key=3





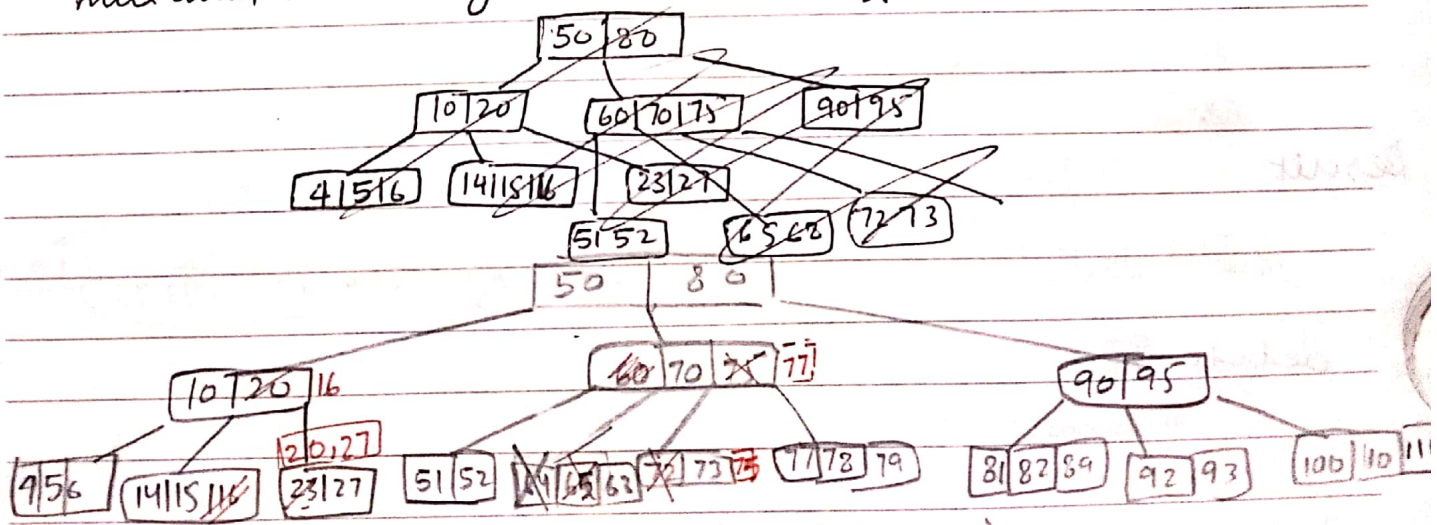
Deletion in B-Tree.

(1) Leaf nodes. (2) Internal Node.

- Leaf has more than min no. of keys (just delete)
- The leaf node contains min no of keys.
  - borrow key from immediate left/right iff sibling has min no of keys
  - neither left nor right has > min num of keys.

Order (m) = 5

max child 5 max key 4 min child = $\lceil \frac{m}{2} \rceil = 3$. min keys = $\lceil \frac{5}{2} \rceil - 1 = 2$



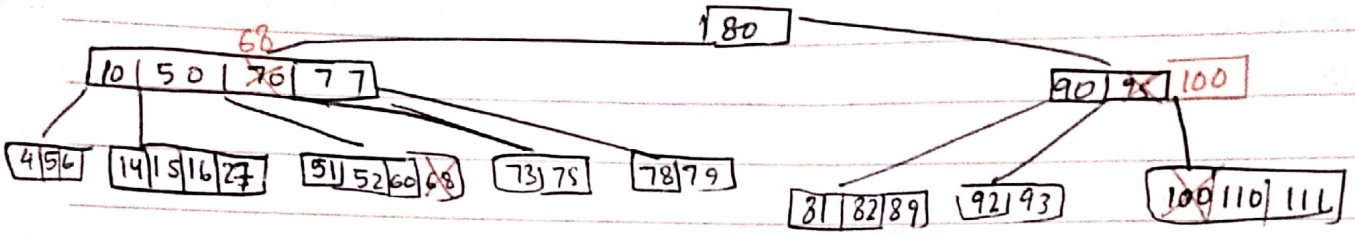
delete 64. (leaf node; node has > min keys so just delete)

delete 23 (has min keys i.e 2)

delete 72

delete 65 (merge with [51, 52, 60, 68] ↳ new node.) would also merge with right side LOL

Target key is in internal Node.



delete 70.

choose inorder predecessor. 68

OR choose inorder successor 73.

→ check if has more than min keys if YES Replace.

delete 95

Predecessor 93

↳ cant replace as has min keys.

Successor 100

↳ has > min num keys so REPLACE

delete 77

Predecessor 75

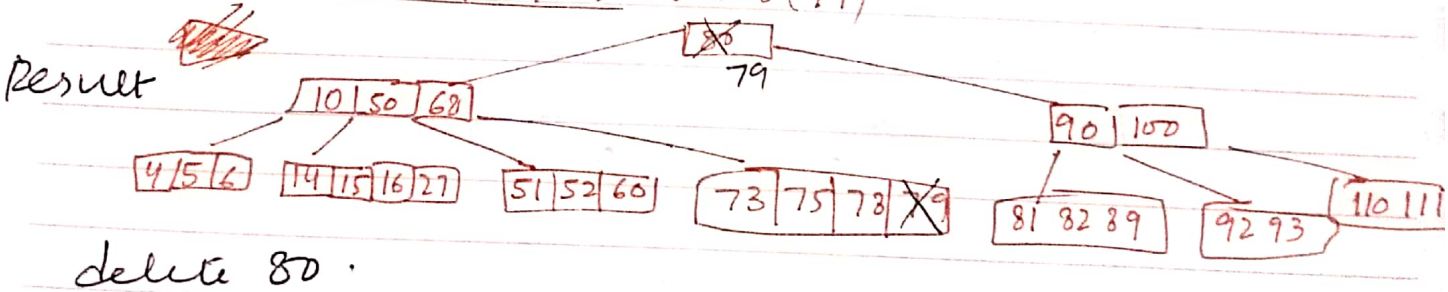
↳ can't as only has Min keys

Successor. 78

↳ can't ~~delete~~ replace as has only Min keys.

Merge 73 75 77 78 79 delete (77)

Result

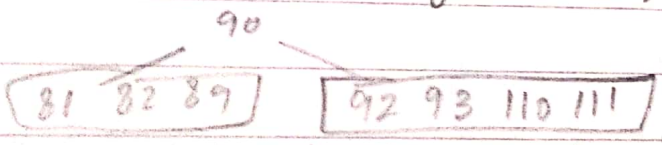


delete 80.

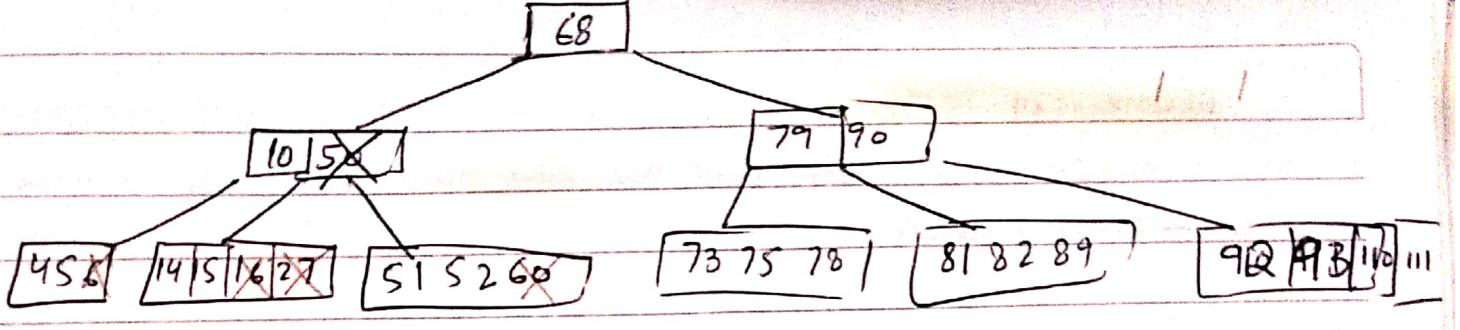
Predecessor 79

↳ can replace

delete 100. (merge delete)



GAMA Ask siblings as 90 violates



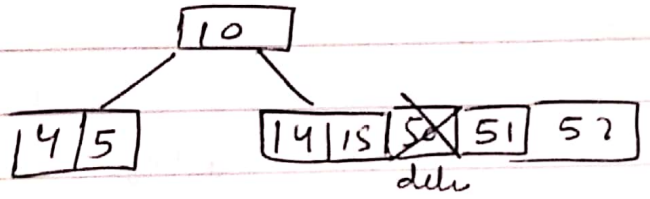
delete 6, 27, 60, 16

delete 50

↳ Predecessor 15 can't replace

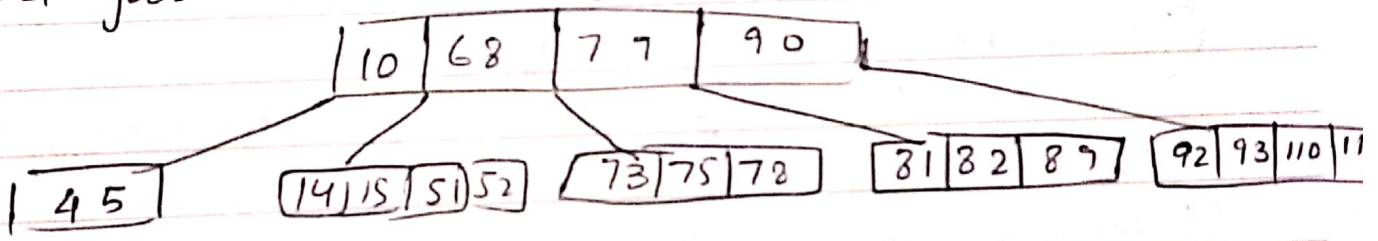
Successor 51. Can't replace.

MERGE



problem is violates has only 1 key.

can't borrow from sibling (has min keys)
parent goes down and merges.



case of shrinking height ↑

Deletion of B-Tree.

(1) key is deleted from leaf and has more than minimum number of ke
— simply delete

(2) key is deleted from leaf and has minimum number of keys

Two Cases.

Siblings donate

(a) • The leaf has a right/left siblings with more than minimum number of keys — Here borrow key from parent and donate a key from left/right sibling to parent.

No Siblings Donation. (MERGE)

(b) • Leaf doesn't have a sibling w/8 more than minimum number of keys in this case merge with left/right sibling and delete node.
Remember when merging bring down the parent and then delete node.
Then check if Parents Violate (Recursively)

(1) Parents have minimum number of keys — do Nothing

(2) Parent has fewer keys — but a sibling can lend a key apply 2a

(3) Parent has fewer keys — sibling can't donate apply 2b

= Done recursively till root reached

(3) Internal Node has to be deleted:

↳ find successor Replace if Node of Successor has > min key
↳ find predecessor Replace if Node of predecessor has > min key
↳ bring down parent merge with children delete node
↳ make sure from when parent came that node doesn't violate any B-tree properties

Insertion Sort: = O(n²) even in best case

```
for(int i=0 ; i< size ;i++)
{  int temp=arr[i];
   for (int j=i ; j>0 && temp<arr[j-1] ;j--).
   {
      arr[j] = arr[j-1]
   }
   arr[j] = temp;
}
```

O(n) best case scenario

more efficient

Slow + inefficient for larger data.

For smaller data it is the fastest

simulation

i=0 7 8 6 10 5

i=1 7 8 6 10 5

i=2 7 8 8 10 5 — 6 7 8 10 5

j: 2 6 < 8 True.
j: 1 6 < 7 True.

j=3 6 7 8 10 5

j=4 6 7 8 10 10 — 6 7 8 8 10 — 6 7 7 8 10 — 5 6 7 8 10

5 < 10 True
5 < 8 True.
5 < 7 True
5 < 6 True.

Bubblesort
- slow for larger data.
- Best case O(n)
- worst + Avg O(n²)

Sorts subarrays

Selection Sort: O(n²) worst case scenario + best + avg ~~O(n)~~

```
for (int i=size-1; i>=0 ;i--)
{   max_index=0;
    for (int j=1 ; j<i ;j++)
    {   if (arr[max] < arr[j])
            max=j;
        if (arr[i] < arr[max].
        {  swap (arr[i], arr[max]);}
    }
}
```

Simulation

i: max=1
5 < 8.
7 5 6 10 8

j: 2 max = 2

j=3 max=10
7 5 6 8 10

i=3
j=1 max=0 — 6 5 7 8 10
j=2 max=

i=2
j=1 max=
j=

5 6 7 8 10

Complete Tree

each internal node has exactly 2 children
all leaf nodes are on same level

has N nodes and h height

$\lceil N/2 \rceil$ leaves nodes.
$\lfloor N/2 \rfloor$ internal nodes
height $= \log_2(N)$
levels $\log_2 N + 1$.
$N = 2^{h+1} - 1$.

Full binary Tree

every node has either 0 or 2 children

Nearly Complete Tree

every level except last are completely filled all nodes are as far left as possible

height $\lfloor \log N \rfloor$

```
int partition (int a[], int s, int e)
{
  int pindex = s-1;
  int pivot = arr[e];
  for (int i=s ; i<=e ; i++)
  {
    if (arr[i] < pivot)
    {
      pindex++
      swap (arr[pindex], arr[i])
    }
  }
  swap (arr[pindex+1], arr[e]);
}
```

0	1	2	3	4
9	8	6	4	5

call 1 s:0 e=4.
pivot 5 pindex = -1.
j:0 if (9 < 5) false
swap (arr[0], arr[0])

j:1 if (8 < 5) false
j:2 if (6 < 5) false
j=3 if (4 < 5) True
pindex: 0
swap (arr[0], arr[3])

4	8	6	9	5

swap (arr[pindex+1], arr[e])

4	5	6	9	8

return pindex = 1
call 2 QuickSort (arr, 0, p-1)
call 3. arr (p+1, high) 2, 4

6	9	8

Partition
pivot 8 pindex = -1

j=0 if (6 < 8) True.
pindex = 0+2
swap (arr[0], arr[0])

j:3 if (9 < 8) false.
j:4
j: break loop
swap (arr[3], arr[4])

4	5	6	8	9

pi 3
QuickSort (2,2
4,4

3
4
1
0

Binary Tree

(a) number of leaves (max) : 2^h.

(b) max number of nodes : $2^{h+1} - 1$.

Pre Order
Root-left-right

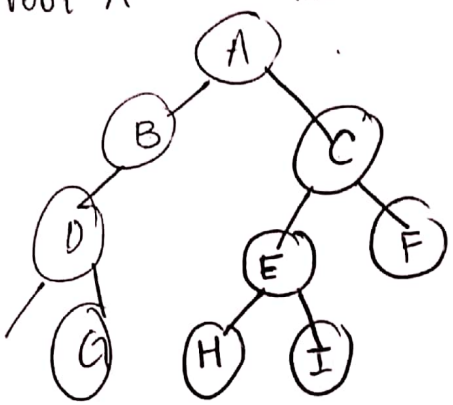
Post Order
left-right-root

In order
left → Root → right

= Inorder preorder.
Inorder postorder.
Inorder levelord.

Eg A B D G C E H I F Pre
D G B A H E I C F Inorder.

root A



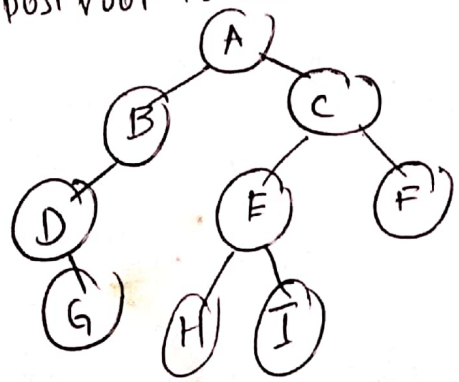
C E H I F

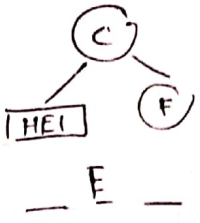
H E I C F

left right

Eg G D B H I E F C A Post
D G B A H E I C F Inorder.

in post root is last

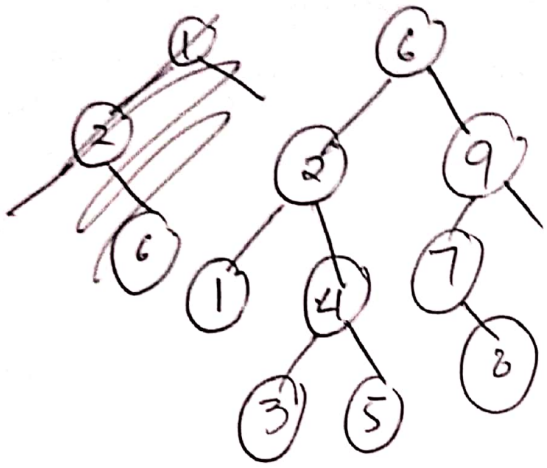




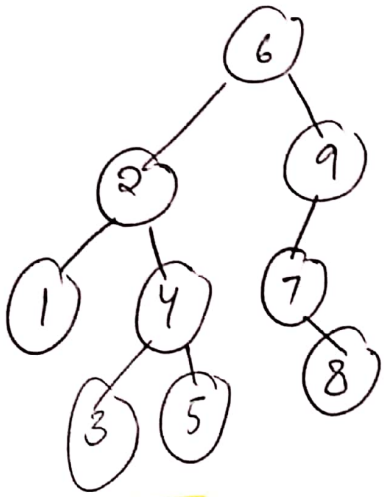
— E —

Inorder
Postorder: 1 2 3 4 5 6 7 8 9
Preorder :
Inorder: 6 2 1 4 3 5 9 7 8.

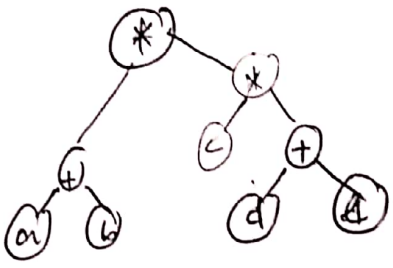
↑ root

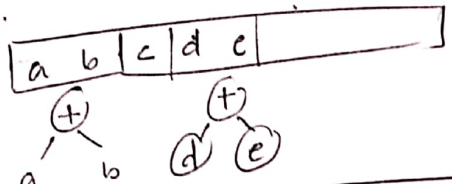


Inorder : 1 2 3 4 5 6 7 8 9
Postorder: 1 3 5 4 2 8 7 9 6.

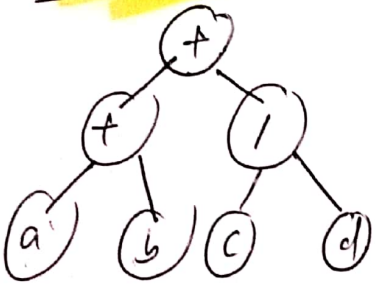


Construct Tree
ab + cd e + * *.





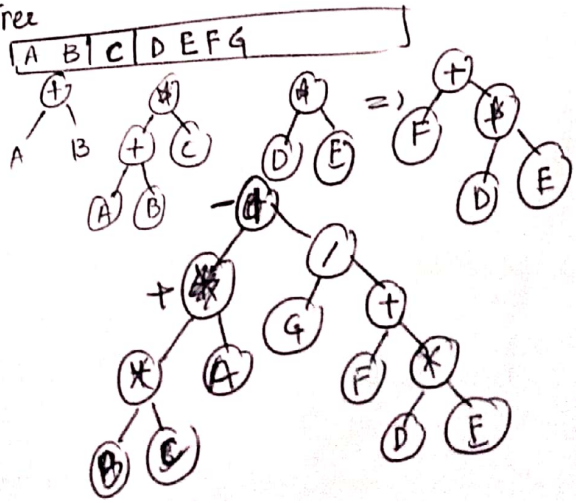
Expression Tree



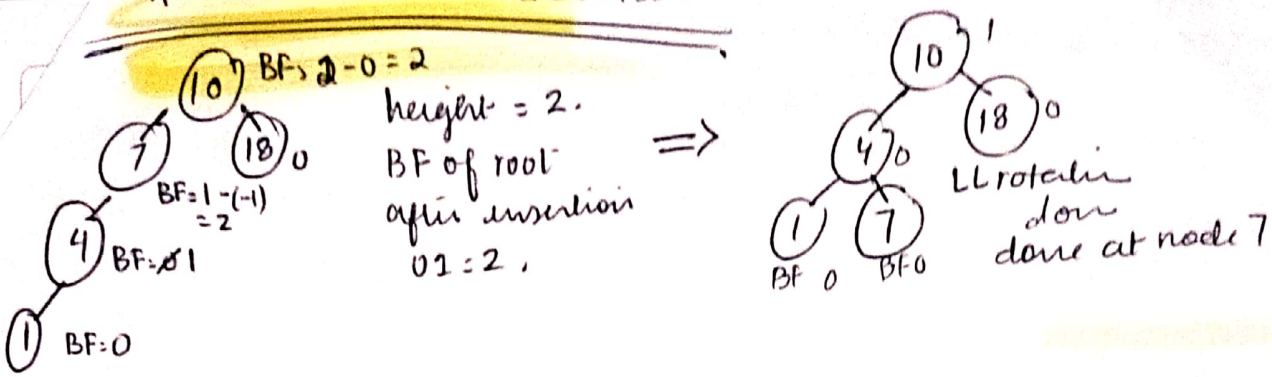
Inorder a + b + c / d
Preorder + + ab / cd
Postorder ab + cd / +.

(A + B * C) - ((D * E + F) / G).
(AB + C *) - (DE * F + G /)
AB C * * DE * F + G / ←

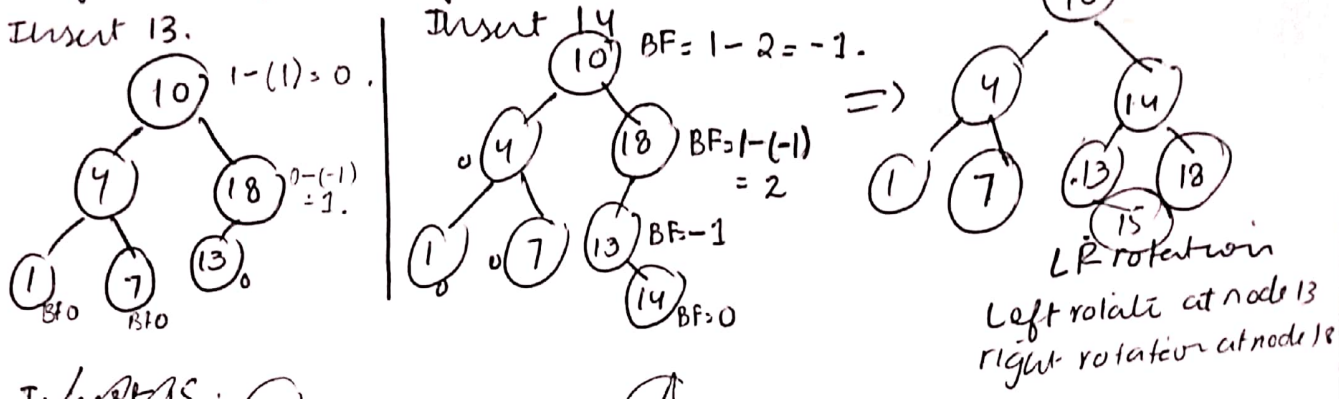
Tree

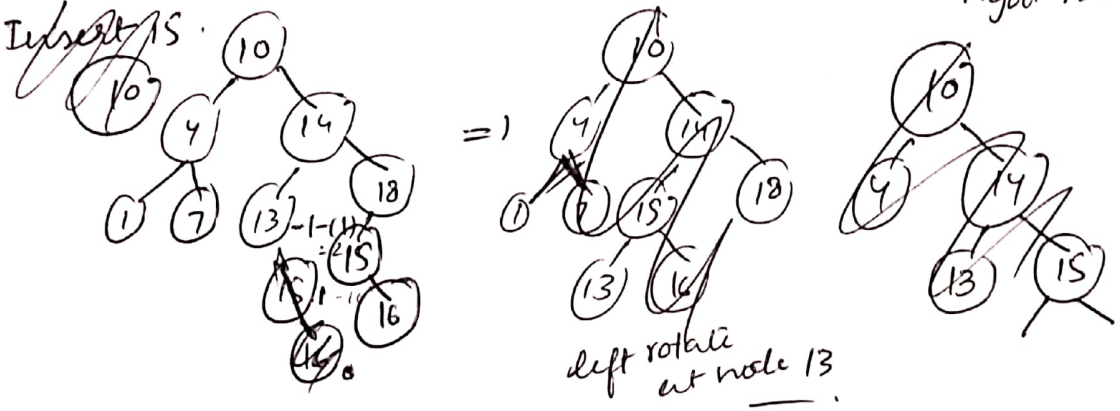


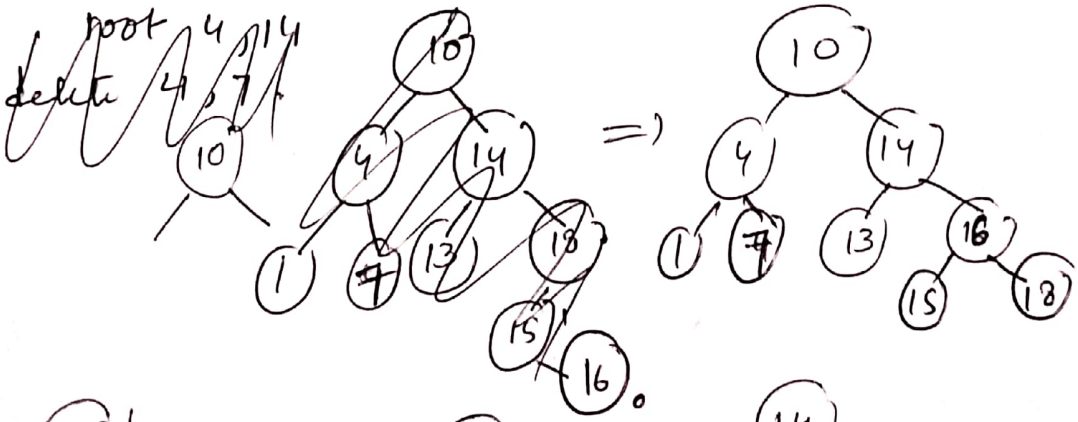
AVL insertion + deletion

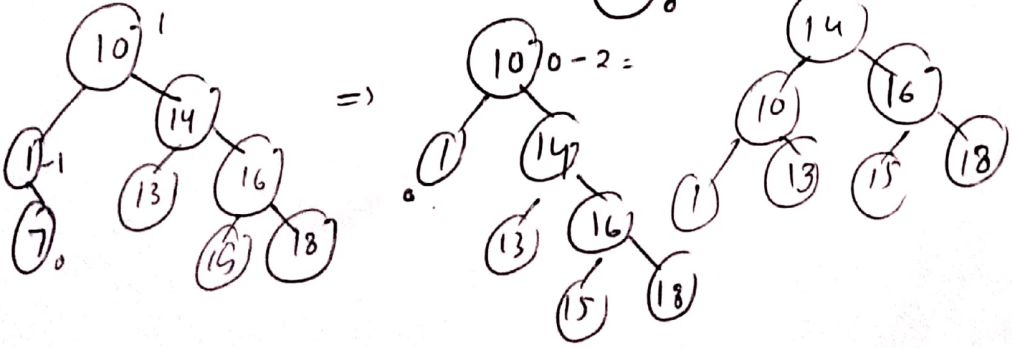


Left and Right child of 4 are 1, 7.









✓ Expression Tree.

✓ Create tree using inorder + preorder
inorder + postorder
inder + level order.

convert infix exp → postfix code.
prefix.

✓ Equal brackets.
Heapify code.

Infix To Postfix.

(A+B) - C

result = AB+C-

case

```
if (a-z || A-Z || 0-9)
    result += character.
else if '('
    push to stack.
else if ')'
    while (stack.top() != '(')
    {   result += st.top()
        st.pop()}
    st.pop();

else if (operator).
    while(st.empty()!=) && pre[character] ≤ st.top(precede).
    {   result += st.top()
        st.pop()
    }
st.push(c)
```

Infix to prefix,

~~)A+B(-C~~

reverse.

C-)B+A(

reverse) (

C-(B+A).

get infixPost

reverse ↑.

RED BLACK TREES

1. follows property of BST
2. each node is black/red
3. Root is always black.
4. Every leaf which is Nil is black.
5. If a node is red then its children are black
6. Every path from node to any of its descendants NIL node has same no. of black nodes.

AVL $\subseteq$ Red black tree

So every AVL tree after coloring == red black tree

but not every red black tree is avl tree

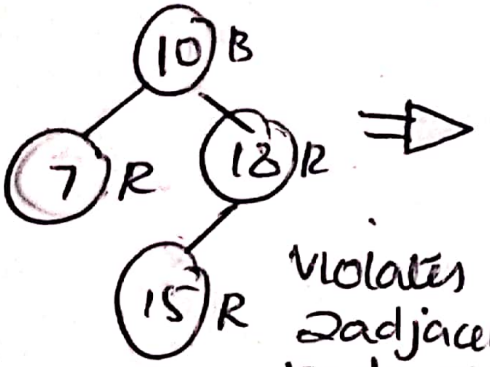
as Red black tree is roughly height balance

AVL tree is strictly height balanced.

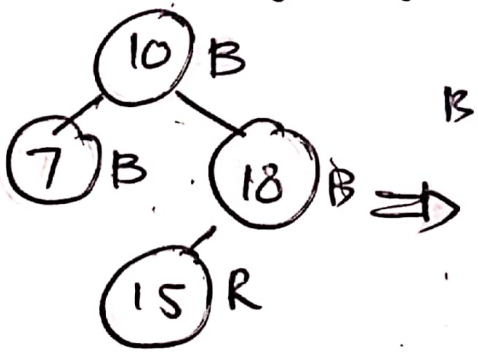
Insertion

1. If tree is empty, create new node as root with color black
2. if tree is not empty, create newnode as leaf node w/8 color red.
3. If parent of newnode is black then exit.
4. if parent of newnode is red, check the color of parent's sibling of newnode.
   (a) If color is black/null the do suitable rotation + color.
   (b) if color is red then recolor and also check if parent's parent of newnode is not root node then recolor it and recheck

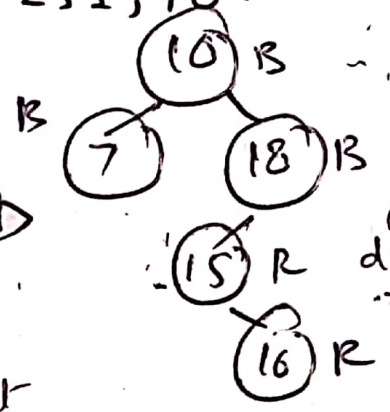
10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70.



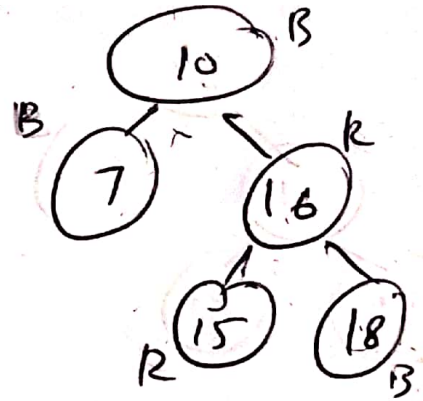
Violates 2 adjacent red nodes property



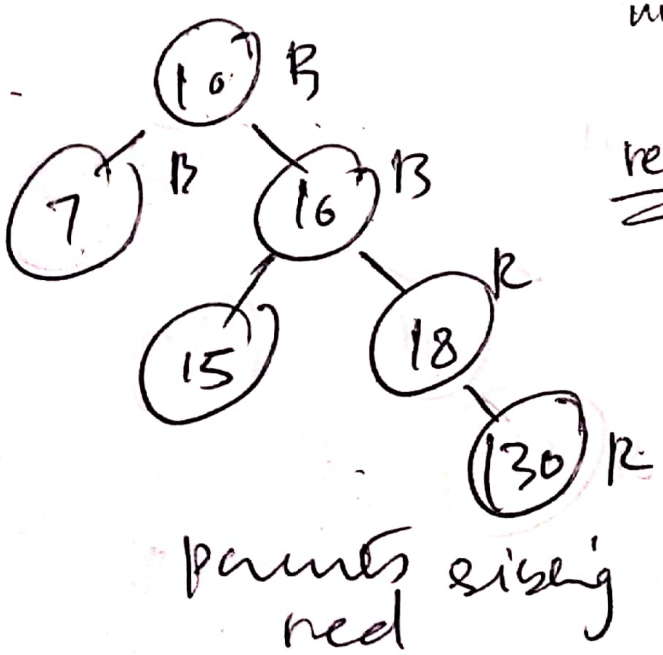
as parent's parent is root node so nothing done to it



(a) do rotation + recolor

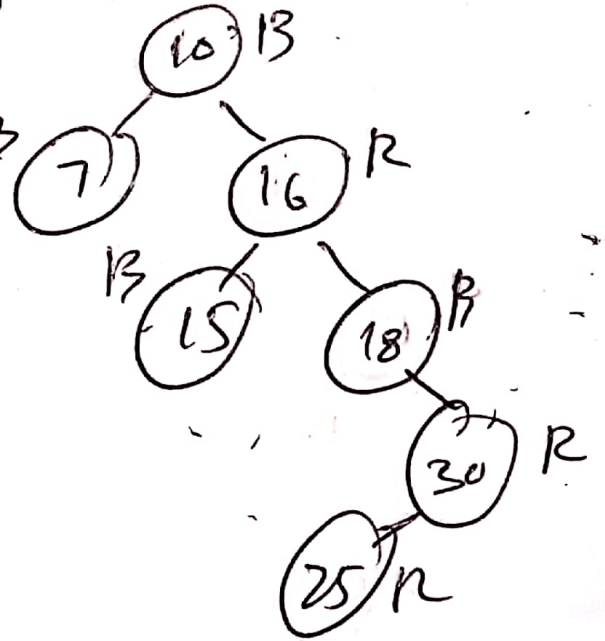


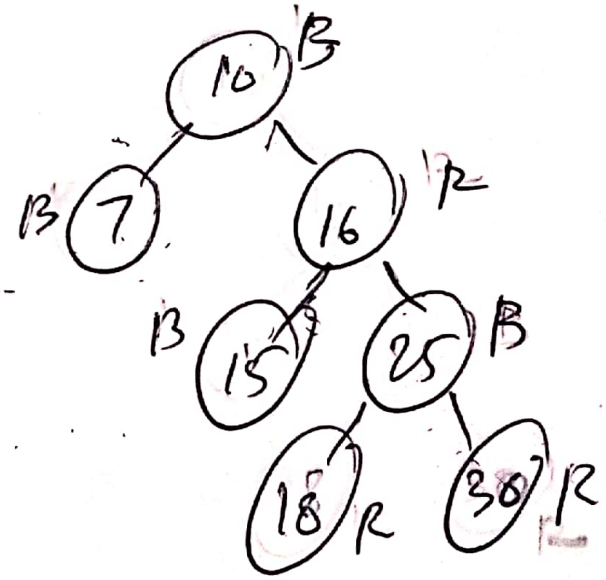
now recolor



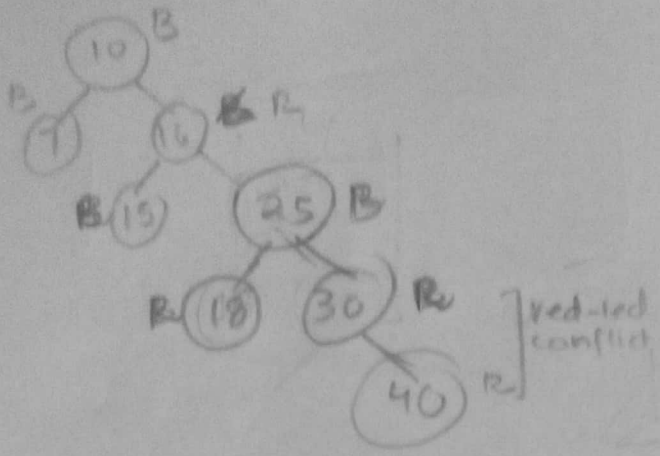
parents red siblng

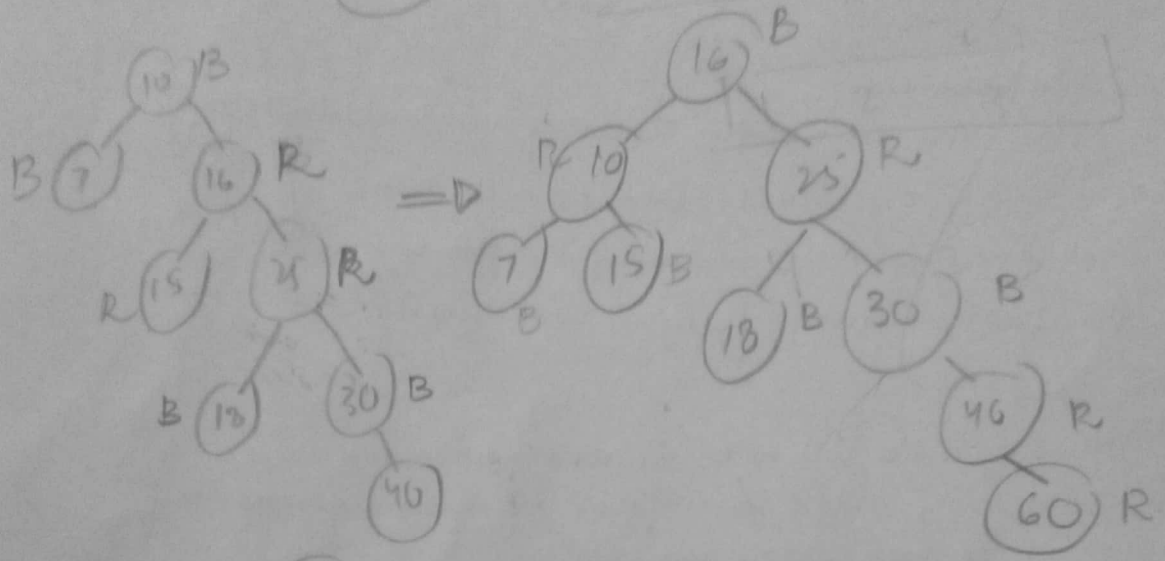
recolor

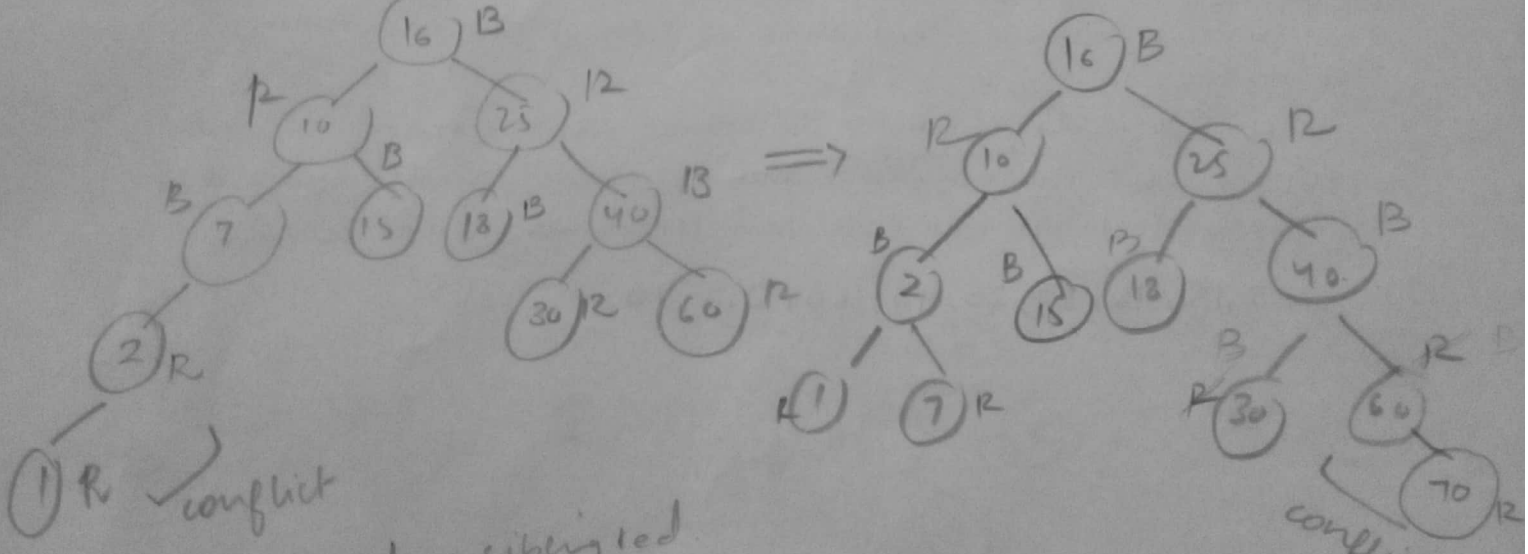




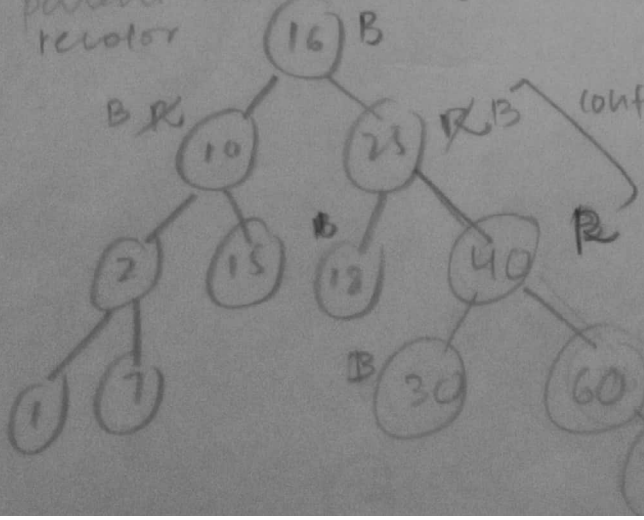
If parent of newnode is red
check siblings color
if black rotate + recolor
if red. recolor





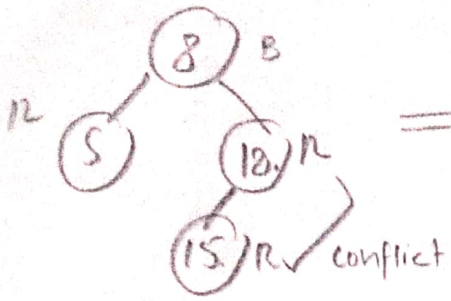


parent red + sibling red
recolor

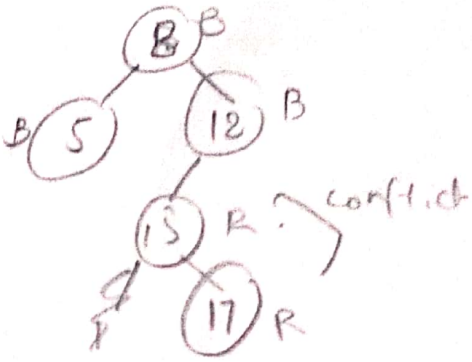


2.

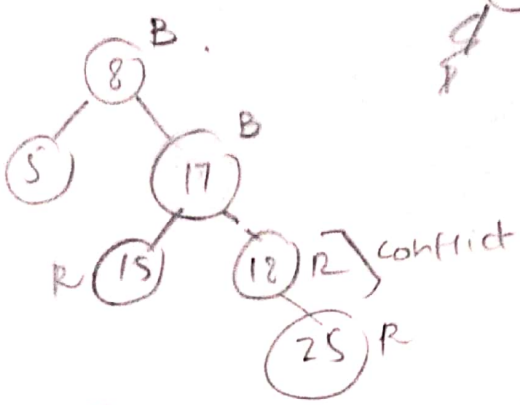
8 18 5 15 17 25 40 80.



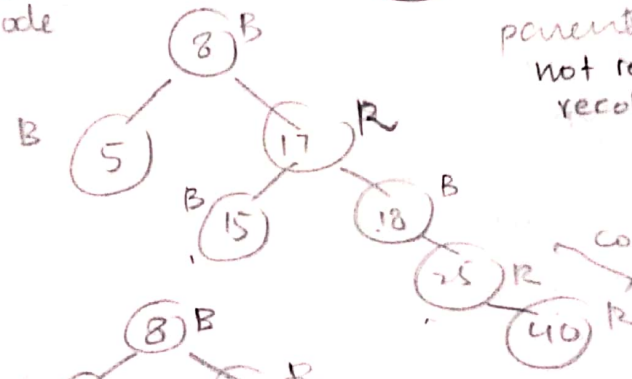
⟹ parent red + uncle red. recolor.



parent red uncle black so rotate + recolor.

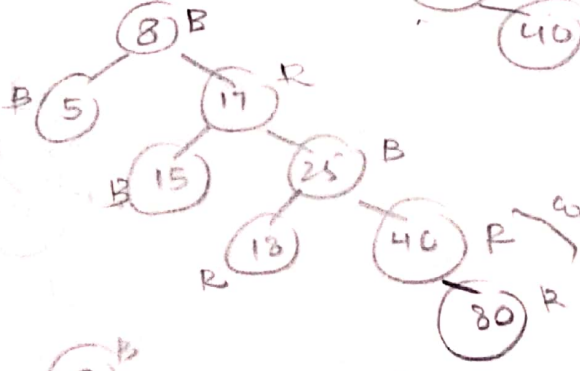


parent red + uncle red ⟹ recolor + check if parent's parent is not node the recolor and recheck.

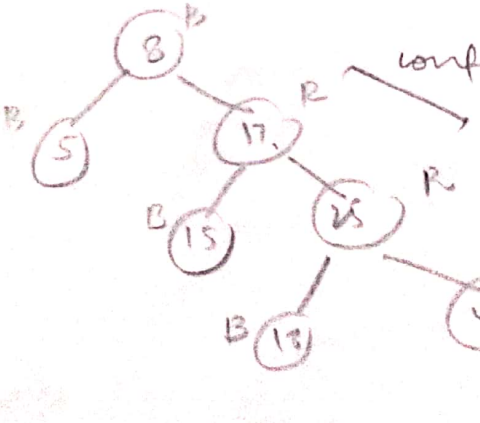


parents parent is not root so recolor;

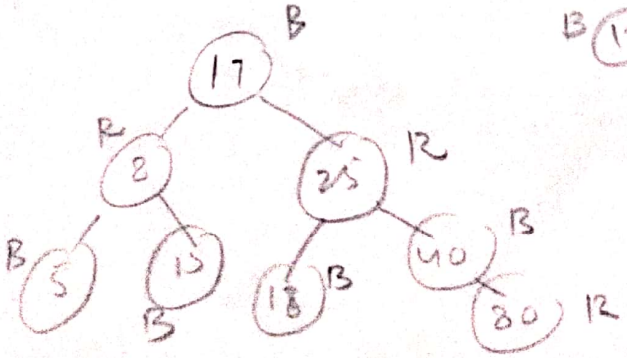
parent is red uncle is black rotate + recolor.



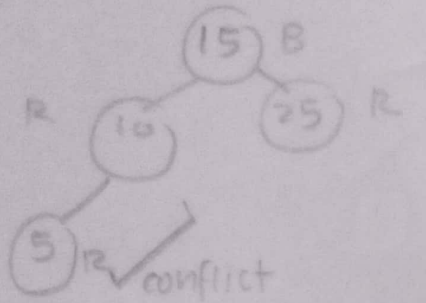
parent red + uncle red. recolor —



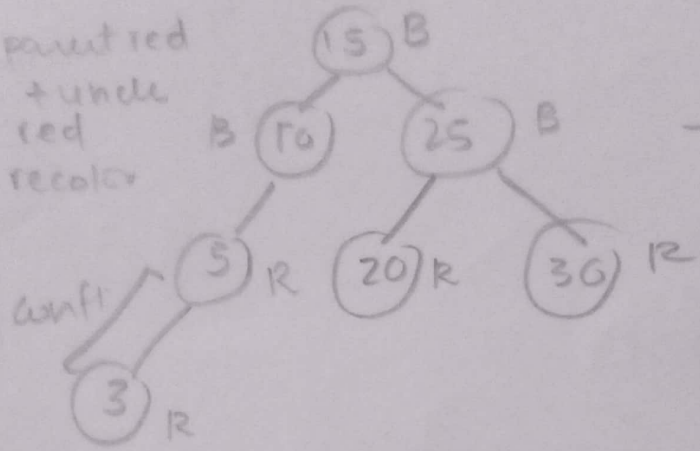
parent sibling is black rotate + recolor.



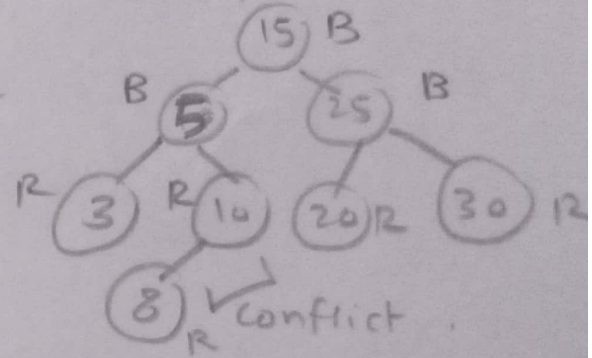
15 25 10 5 20 30 3 8 18 16.



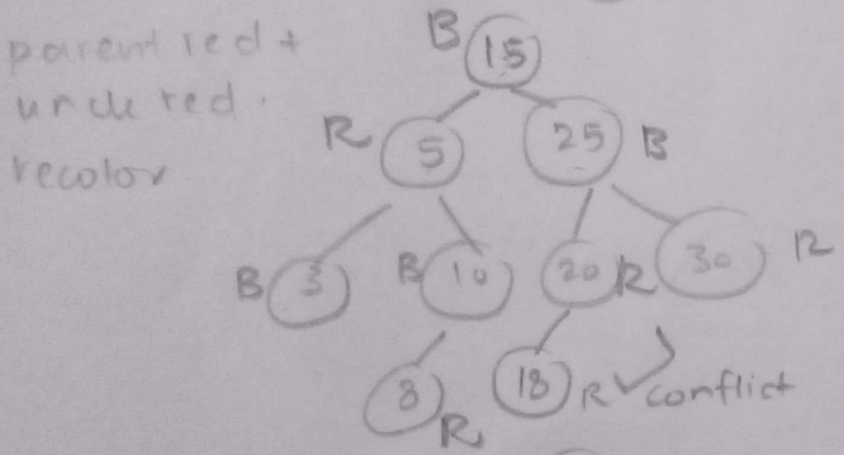
parent red + uncle red recolor



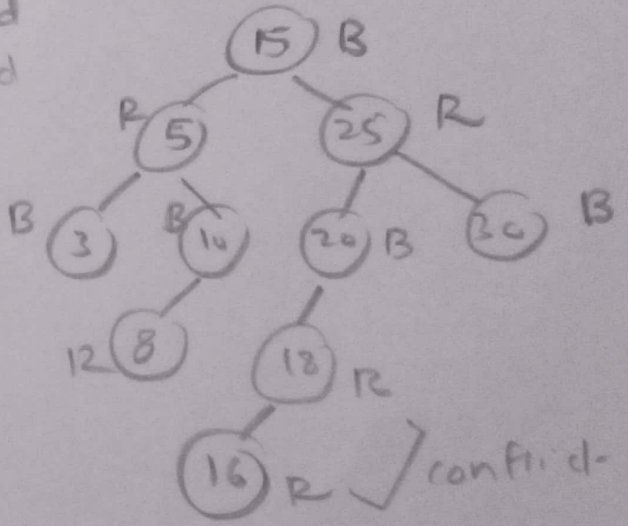
parent red uncle is black rotate + recolor.



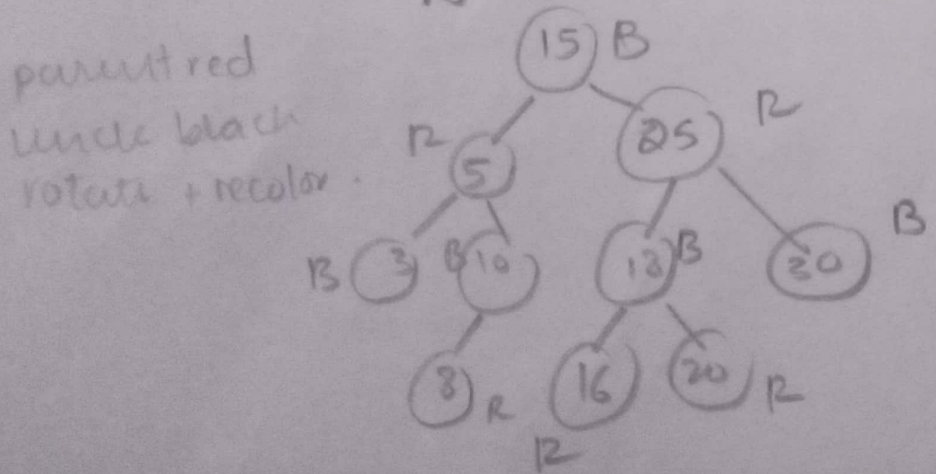
parent red + uncle red. recolor

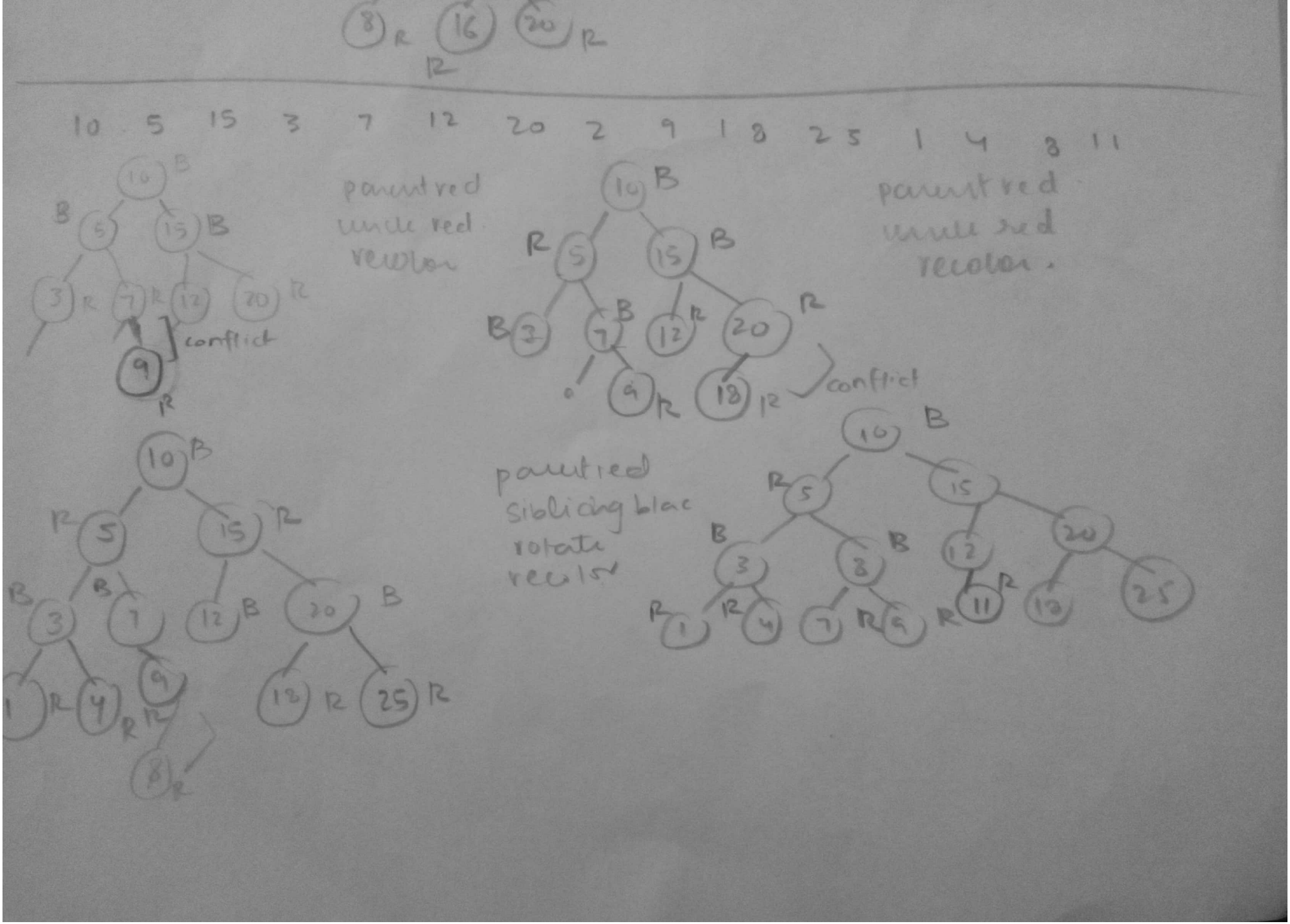


parent red uncle red recolor

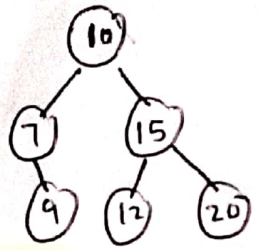


parent red uncle black rotate + recolor.




8 R
16
R
20 R
10 5 15 3 7 12 20 2 9 1 8 2 5 1 4 8 11
parent red
uncle red
recolor
conflict
parent red
sibling blac
rotate
recolor
parent red
uncle red
recolor.
conflict

Deletion in red black trees

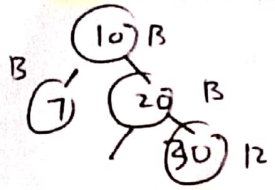


deleting a red node no problem

deleting a black node - violates property no. of black nodes in a path is less

- first perform BST deletion

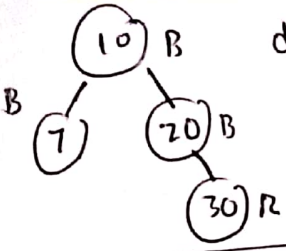
Case I : node is Red.



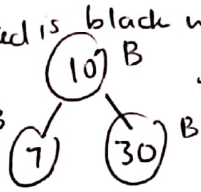
delete 30 it is red so delete and has no children.

=>

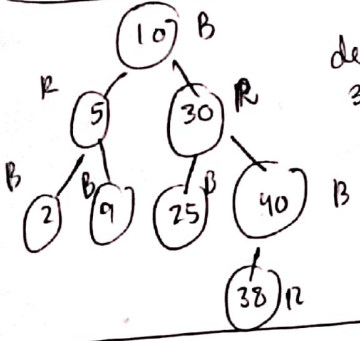
Case II : node to be deleted is black with one child of color Red.



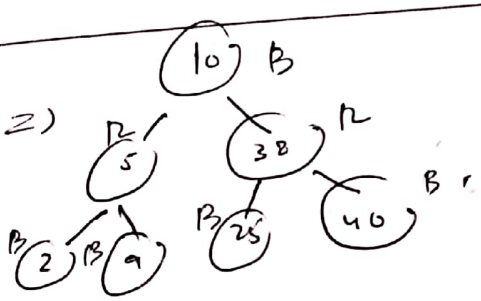
delete 20 =>



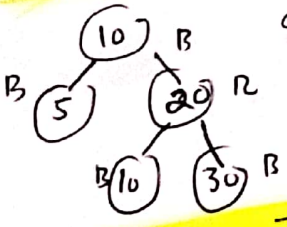
internal node preserves color as node is replaced.



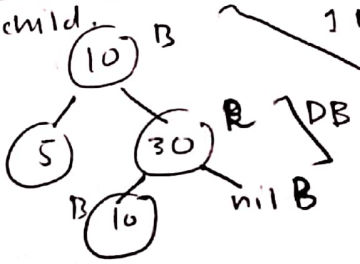
delete 30 =>



Case III black node with black child.



delete 20 replace with 30



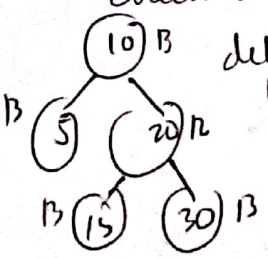
1 less black node here.

— **Sub Case I**

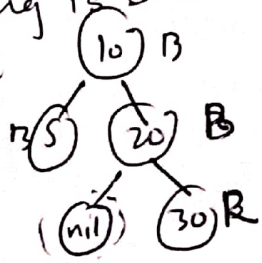
root is DB remove DB

— **Sub Case II**

check sibling is Black & children is also black.



delete 15

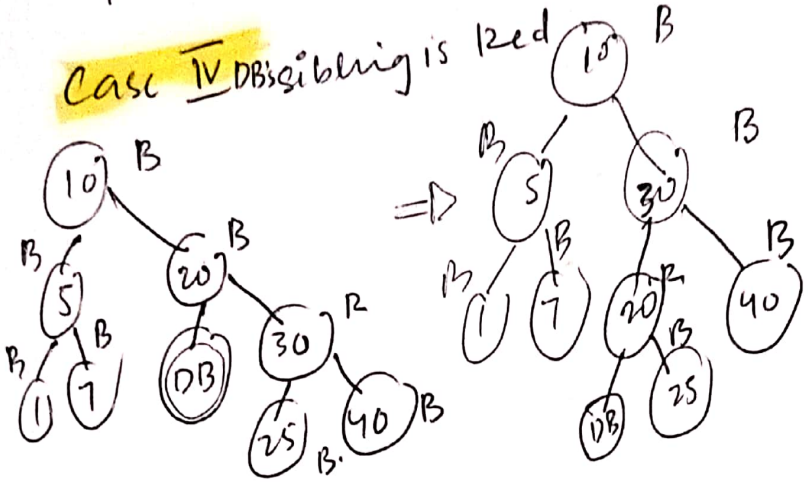


- remove DB
- add black to parent
  - if P is red it becomes black
  - if P is black is becomes DB.
- make sibling red.

+ if still DB exist ... apply other cases.

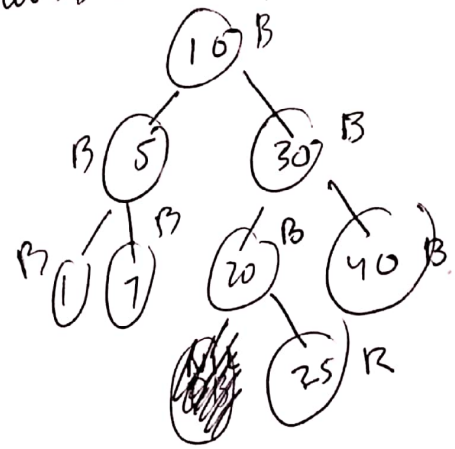
if DB still exist check if sibling and its children are
if yes give DB to parent & make sibling red
if root is DB then simply remove it.

Case IV DB's sibling is red



- swap color of DB parent and sibling
- rotate towards direction of DB
- reapply cases.

still DB exist apply case 3 (in this case).



Case V

DB sibling is black, sibling child whose is far from DB is black but sibling children who is near ~~child~~ DB is red.

(1) swap color of DB sibling with its child near to DB
rotate in opposite direction to DB.
apply case 6.

Case VI

DB's sibling is black and far child is red.

- swap color of parent and sibling.
- rotate parent in DB's direction.
- remove dB.
- change color of red child to black.

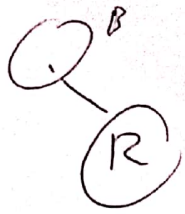
Case I delete node — color Red.

simply delete.

Case II deleting internal node [any color]

replace with successor / predecessor and

don't the node to delete ka color.



Case III

deleting a leaf node with is black. ⇒ cause DB (manhoos)

Case 1° (i) if its root just remove DB

Case 2° (ii) if its sibling is black + siblings kids also black.

give black to parent

(i) if parent is red then after this becomes black.

(ii) if parent is black, make it DB

change color of sibling to red

[if DB still exist apply other cases].

Case IV sibling is RED.

- swap color of DB parent & sibling
- rotate toward direction of DB
- reapply cases.

Case V

DB sibling is black, ~~unrelated~~ sibling child ~~from~~ far to DB is black and sibling child ~~from~~ near from DB is red

- swap color DB sibling with its child near DB
- rotate in opposite direction of DB
- apply case 6.

Case VI

DB sibling is black, sibling far child from DB is ~~black~~ red.

- swap color of sibling with parent
- rotate in direction of DB.
- remove DB
- make sibling red child black