4/10/17

Priority Queues

Operations

1. Make NULL 2. Insert 3. Delete min

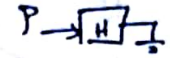
Linked list : p → [H] → [ ] → [ ] → [ ] → ⊥

Header Node

Insert (p, q) ⇒ O(1)

```
nodeptr p,q;
{
  if(p==NULL){
    printf("void insertion");
  }
  q->next = p->next;
  p->next = q;
}
```

Make Null



```
typedef struct node{
    elementtype thing;
    nodeptr nextnode;
}nodetype
    *nodeptr;
```

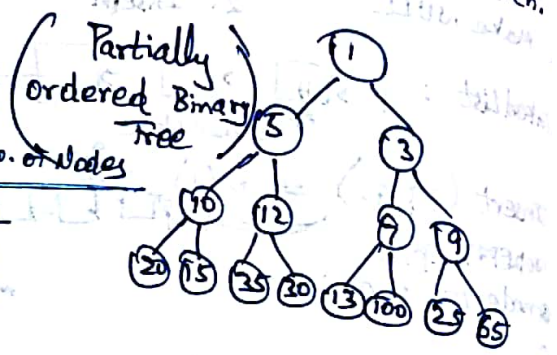
(int, char)

```
deletemin(p, x)                    (* Use nextnode instead of next)
nodeptr p; int *x;
{
  if (p==NULL || p->next==NULL)
    printf("void deletion");
  nodeptr q = p->next, r = q; s;
  int min = q->thing;
  while (q->next != NULL){
    if (q->next->thing < min){
      min = q->next->thing;
      r = q;
    }
    q = q->next;
  }
  x = r->next->thing; s = r->next;
  r->next = r->next->next;
  return Freenode(s);
}
```

Priority Queue using Binary Trees

① Parent should have lower value (higher priority) than its children.

② Balance as much as possible



Height of Tree	Min no. of Nodes	Max no. of Nodes
1	1	1
2	2	3
3	4	7
4	8	15
⋮	⋮	⋮
l	2^{l-1}	$2^l - 1$

$2^{l-1} \le n \le 2^l - 1$ (Heap)

$l \le 1 + \log n$

$O(l) = O(\log n)$

Slide 17

Array Implementation

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

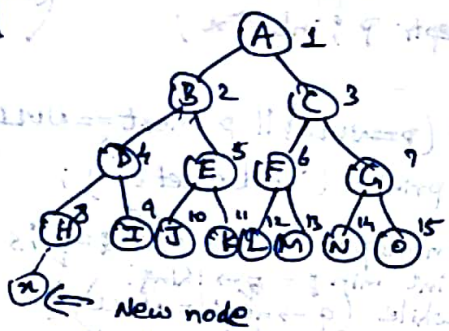
Root

PQ[1] = root;

For every node $i > 1$

Parent of i is at PQ[i/2];

children of i, if they exist, will be at PQ[i*2] and PQ[i*2+1];




```
typedef struct PQueue {
        element type thing[maxPQ]
        int last;
      } *PQptr PQ-type;
```

```
MakeNull (PQ)
PQ ptr PQ;
{
 PQ → last = 0;
}
```

```
Insert (PQ, x)
PQptr PQ ; int x ;
{
  if (PQ→last == maxPQ)
     printf ("overflow! ");
  else
     PQ → thing [PQ→ last] = x ;
```

```
int a, b , a = PQ→last ;
while (a != 1) {
b = (a/2) - (a%2);
 if (PQ→thing [a] < PQ→thing[b])
     swap (PQ→thing[a], PQ→thing[b]);
     a = b ; }
else
  exit();
}
```

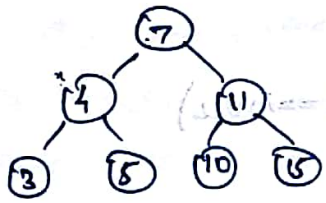
```
DeleteMin (PQ, x)
 PQptr PQ; int *x;
{
  if (PQ->last==0){
     printf ("No elements");
     exit(1);
   }
  *x = PQ->thing[1];
  PQ->thing[1] = PQ->thing[PQ->last--]
  int a = 1;
  while (a < PQ->last) {
     if (PQ->thing[2*a] > PQ->thing[2*a+1])
     b = PQ->thing[2*a] , c = PQ->thing[2*a+1];
     if (PQ->thing[a] > min(b,c)) {
        swap (PQ->thing[a], min(b,c));
        a = index (min(b,c));
     }
     else
        exit();
   }
}
```

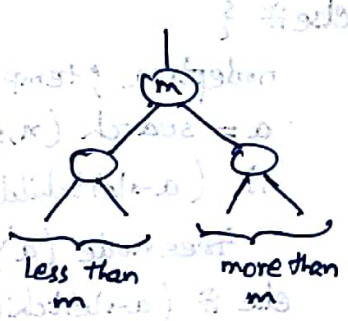
(Swap with smaller child)

6/10/17

Binary Search Tree (BST)

Eg:






```
bool IsMember (x,s)
  elementtype x ; nodeptr s;
{
  if (s==NULL) {
     printf("Nullsearch");
     exit();
  }
else if (x==s->element)
     return true;
  else if (x <= s->element)
    return IsMember (x, s->leftchild);
  else
    return IsMember (x, s->rightchild);
}
```

```
typedef struct node{
    elementtype element;
    nodeptr leftchild, rightchild
  } nodetype *nodeptr;
```

```
MakeNull (s)
nodeptr s;
{
  s = makenewnode();
  s->leftchild = NULL;
  s->rightchild = NULL;
  s->element = NULL;
  return(s);
}
```

```
Insert (x,s)
elementtype x; nodeptr s;
{                                   if (s==NULL){      if (IsMember(x,s))
                                      MakeNull(s);       exit();
                                      s->element = x;
                                    }
  nodeptr a = s; b; MakeNull
  MakeNull(b); b->element = x;
  while (a->leftchild != NULL && a->rightchild != NULL){
     if (x > a->element)
       Insert(x, a->rightchild);
     else if (x < a->element)
       Insert(x, a->leftchild);
     }
   if (x > a->element)
     a->rightchild = b;
   else
     a->leftchild = b;
}
```

```
Delete (x,s)        [Replace with Inorder Successor]
elementtype x; nodeptr s;
{                   if (s==NULL)
                      exit();
  if (IsMember(x,s))
     Printf ("Element does not exist!");
  else if {
     nodeptr a; temp;
     a = search (x,s);
     if (a->leftchild==NULL && a->rightchild==NULL)
        freenode (a);
     else if (a->leftchild ==NULL) {
        temp = a->right;
        a = a->rightchild;
        freenode (temp);
        }
     else if (a->rightchild ==NULL) {
        temp = a;
        a = a->leftchild;
        freenode (temp);
     }
     else {
       a->element = Deletemin (a->rightchild);
     }
}
```

```
nodeptr search (x,s)
element type x; nodeptr s;
{ if (x==s->element) {
    return s;
  else if (x < s->element)
      search (x, s->leftchild);
    else
      search (x, s->rightchild);
  }

element type Deletemin (A)
 node ptr A;
{
 element type temp;
 if (A->leftchild == NULL) {
    temp = A->element;
    A = A->rightchild;
    return temp;
  }
 else
   return (Deletemin (A->leftchild));
}
```

eg: Make BST, insert fn., delete fn.

10/17

Balanced Binary search Tree

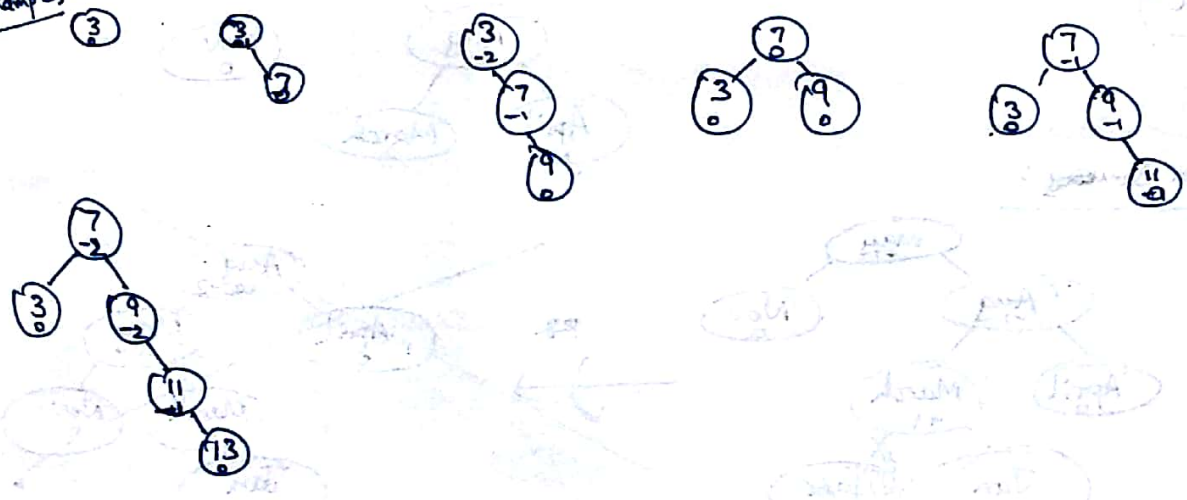
* Height balanced

We call a Binary Tree 'T', as height balanced if,

(i) The left and right subtrees 'T_L' and 'T_R' are height balanced.

(ii) The difference of height of T_L and T_R, i.e., $|h_L - h_R| \leq 1$

examples



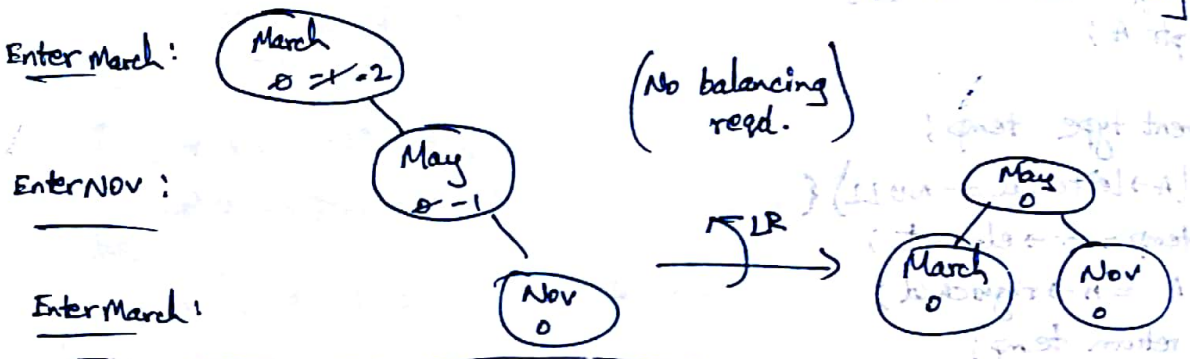
AVL Type of Tree (Addelson, Velskii, Landis) (1962)

A binary tree is an AVL Tree if every non-leaf vertex except the root has a sibling and for every pair of siblings, their height differs by atmost 1.

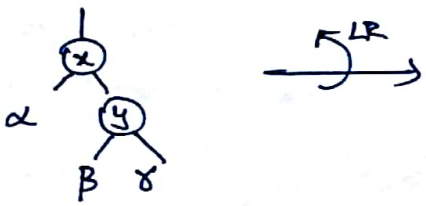
eg: Suppose we enter the following in the order:

March, May, Nov, Aug, April, Jan, Dec, July, Feb, June, Oct, Sep

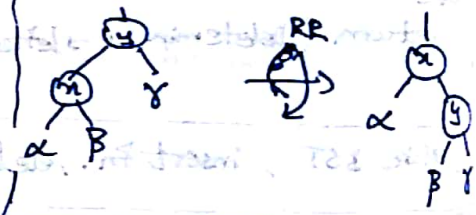
[Alphabetical order]

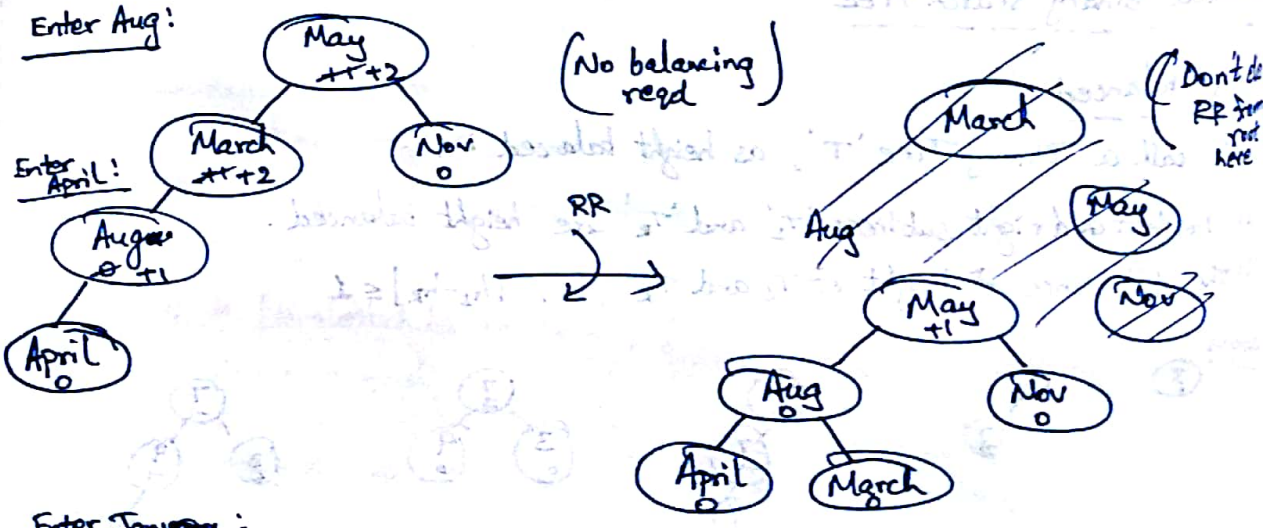


* The idea of ~~Left~~ Rotation (LR)

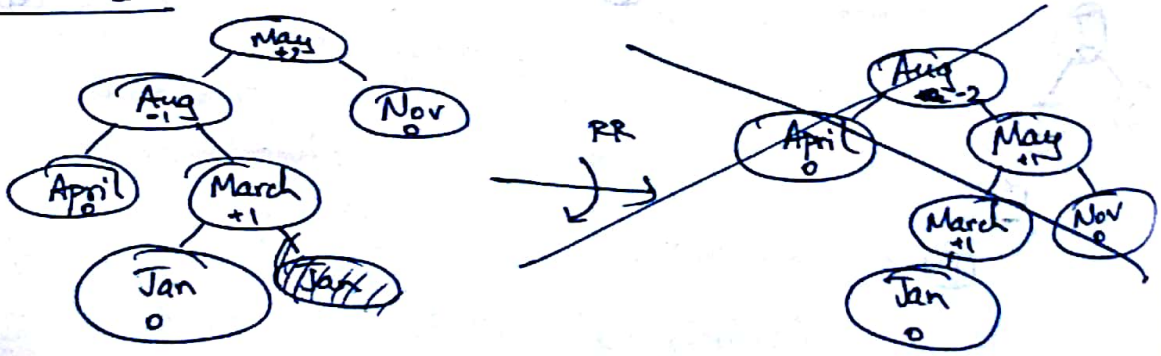


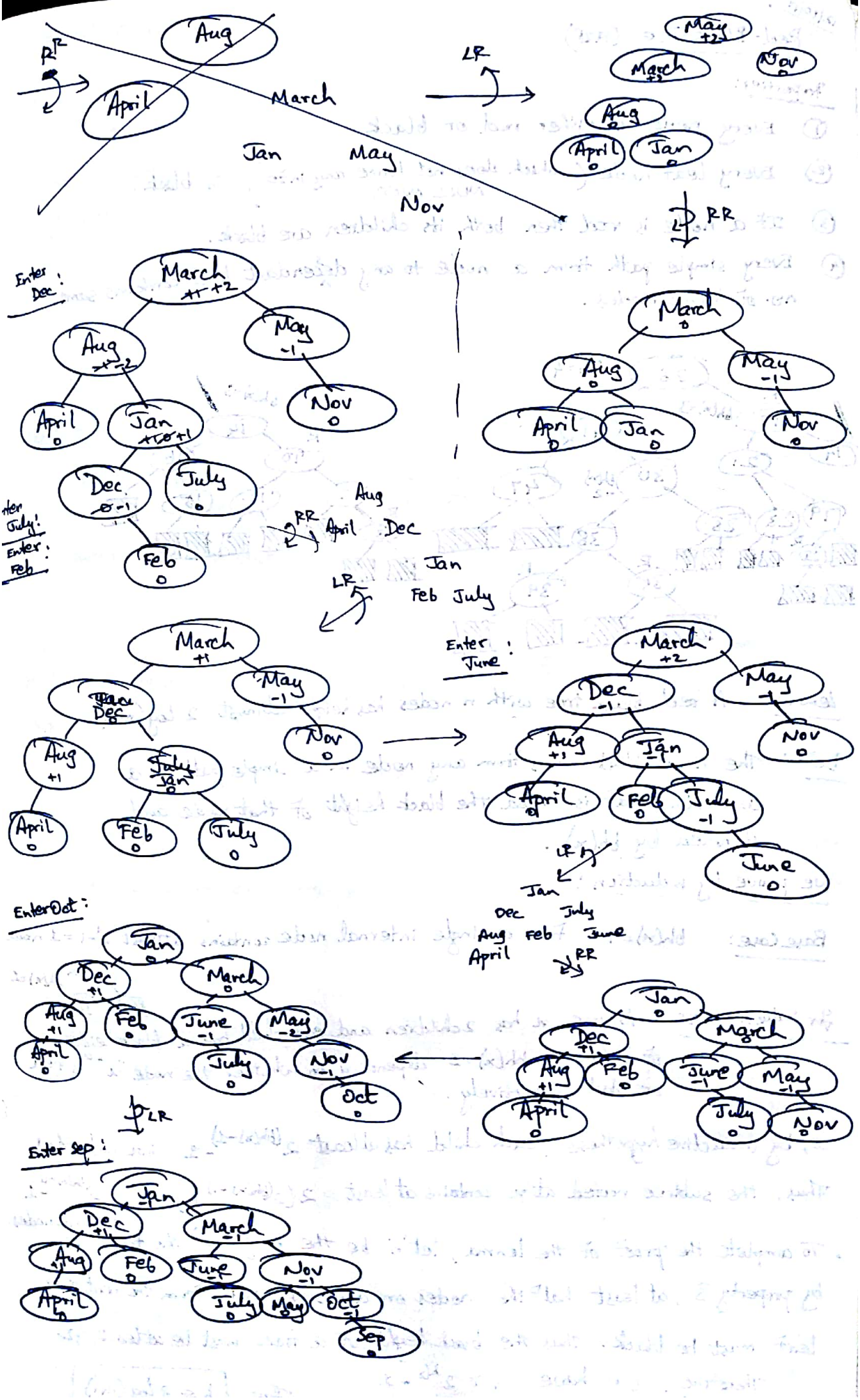
* The idea of Right Rotation





Enter Jan:

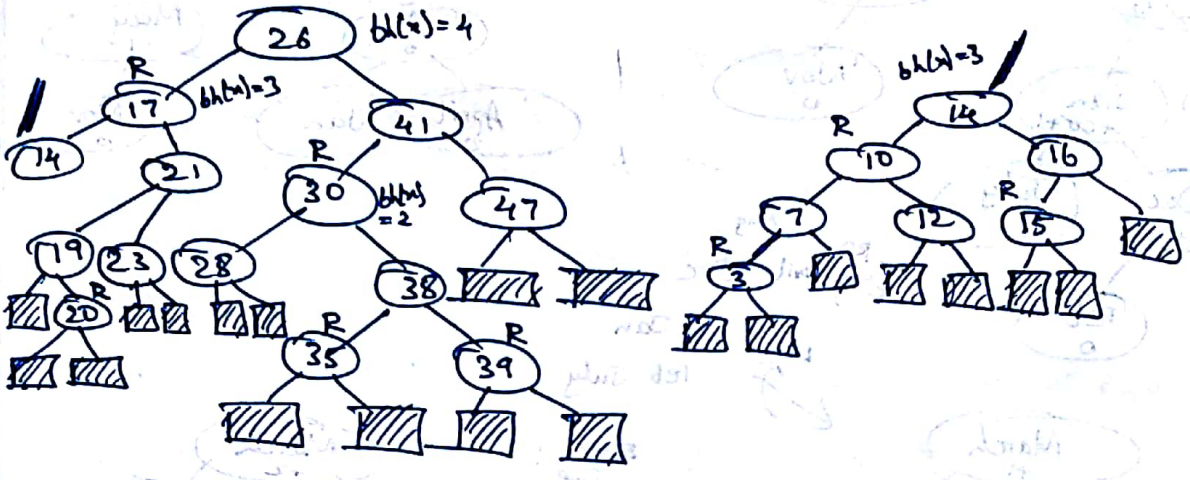




Red-Black Tree (1978)

Properties:

1. Every node is either red or black.
2. Every leaf node (which does not have any into NULL node) is black.
3. If a node is red then both its children are black.
4. Every simple path from a node to any descendant leaf contains same no. of black nodes.



Lemma: A red-black tree with n nodes has height atmost $2\log(n+1)$.

Defn: The no. of black nodes from any node in a simple path to a descendant leaf is called the black height of that node and denoted by bh(x).

we prove by induction:

Base Case: bh(x)=1 for a single internal node contains atleast $2^1-1=1$ n

for inductive case: Assume x has 2 children and each child has a black height of bh(x) or bh(x)-1 depending on whether the node "x" is red or black respectively.

So, by inductive hypothesis, each child has atleast $2^{(bh(x)-1)}-1$ internal nodes. Thus, the subtree rooted at x contains at least $2(2^{(bh(x)-1)}-1)+1 = 2^{(bh(x)-1)}-1$ internal nod

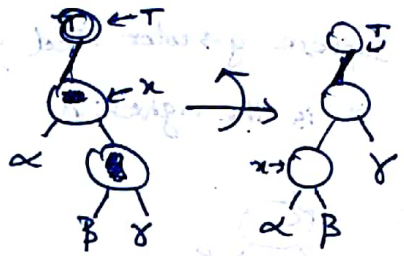
To complete the proof of the lemma, let 'h' be the height of the tree. Now, by property 3, at least half the nodes on any simple path from the root to a leaf must be black. Thus, the black height of a tree must be atleast h/2.

Therefore, we have $n \geq 2^{h/2}-1$

$\log(n+1) \geq h/2$

Thus, $h \leq 2\log(n+1)$

```
Left rotate (T, x)
nodeptr T, x;
{
 T→leftchild = x→rightchild;
 x→rightchild = (x→rightchild)→leftchild;
 (T→leftchild)→leftchild = x;
}
```



Or

```
Left_Rotate (T, x)
Tree T; nodeptr x;
{
 y = x→rightchild;
 x→rightchild = y→leftchild;
 if (y→leftchild == NULL)
    y→leftchild→parent = x;
 if (x→parent == NULL)
    T→root = y;
 else {
    y→parent = x→parent;
 if (x == x→parent→leftchild)
    x→parent→leftchild = y;
 else
    x→parent→rightchild = y;
 }
 y→leftchild = x;
 x→parent = y;
}
```

13/8/17

Insertion:

```
RB_Insert (T, x)
    BST_Insert (T, x)
    x→color = Red;
```

(we insert the new node as red so that black height is not disturbed)

- Property 3 maybe violated.
  i.e. $x \rightarrow parent \rightarrow color = Red$

There may be several cases:
Each case depends on the new node's parent's sibling or x's uncle.

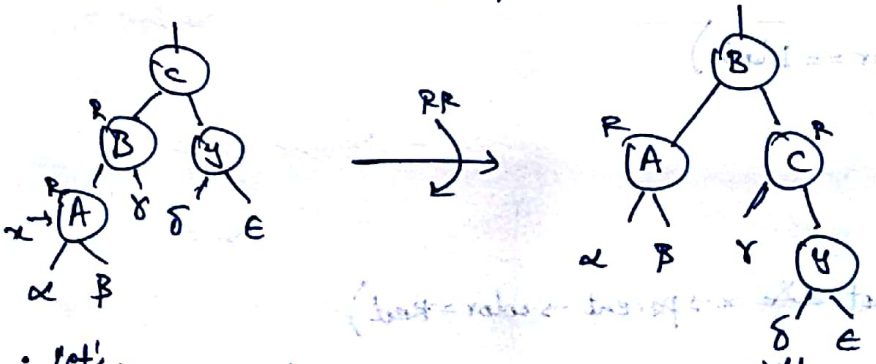
Assuming x's parent is the leftchild of its grandparent.

x's uncle y is —

```
y = x→parent→parent→rightchild;
```

Suppose $y \rightarrow color = black$;

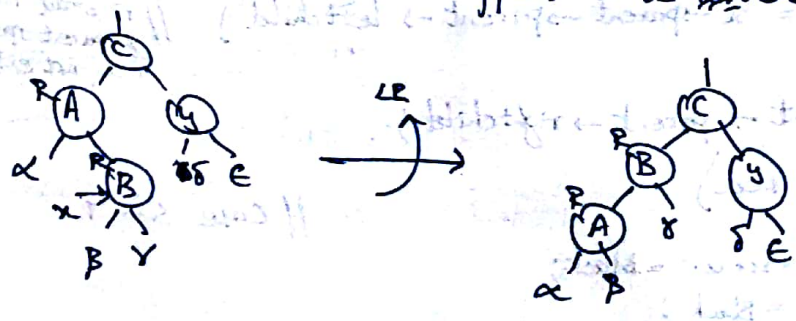
(Only problems arise when you have the parent of new node is red node. Black node is fine.)



Case 1

Color changes:
B→color = Black;
C→color = Red;

- Let's now consider if 'x' happens to be ~~left~~ right child —

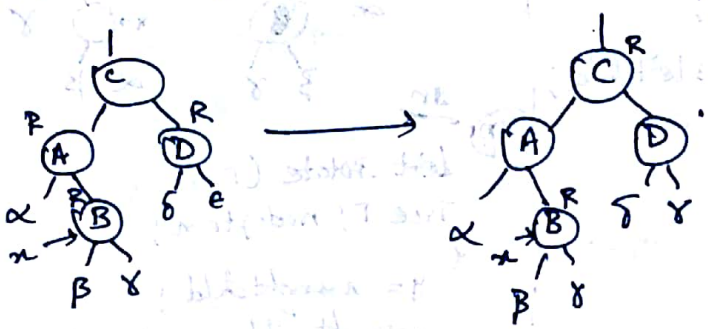


Case 2

Case 2 is now transferred to Case 1.

Suppose $y \rightarrow color = Red$;

x is the rightchild



Case 3

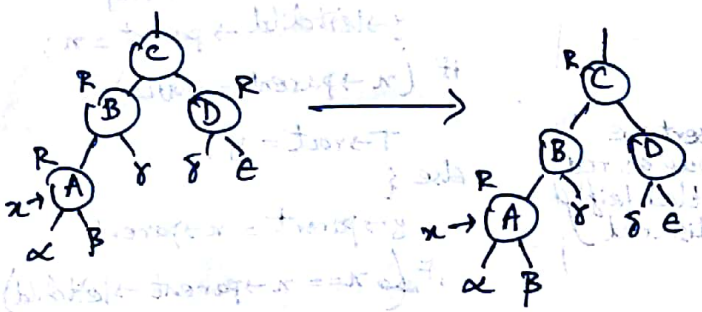
changes:

$x \rightarrow parent \rightarrow color = Black$;

$x \rightarrow parent \rightarrow parent \rightarrow color = Red$;

$y \rightarrow color = Black$;

Case 4



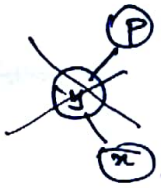
If the parent of C is black, then there should be no issue. But if C's parent is red ~~and black~~ then case 3 & case 4 would pose a proble Hence, we shift the whole problem up. If we reach the root in th way, we make root → color = black to solve the problem.

```
RB_Insert (T, x);
{
  BST_Insert (T, x);
  x → color = Red;
  if (x → parent → color == Black)
     return;
  else {
    y ← x → parent
    while ( x != T → root && x → parent → color == Red)
    {
      if ( x → parent == x → parent → parent → leftchild)   // if small tree x→parent→parent may not exist
      {
        y = x → parent → parent → rightchild;
        if (y → color == Red)
        {
          x → parent → color = Black;
          y → color = Black;                                // case 3 & 4
```

```
            x->parent->parent->color = Red;
            x = x->parent->parent;
        }
    else {                                          // if (y->color == Black)
        if (x == x->parent->rightchild)             // Case 2
        {
            x = x->parent;
            left_rotate (T, x);
        }
        x->parent->color = Black;                   // Case 1
        x->parent->parent->color = Red;
        right_Rotate (T, x->parent->parent);
    }
}
else {                                              // x->parent == x->parent->parent->rightchild
    * same as above replacing 'leftchild' by 'rightchild' and vice versa
    }
}
T->root->color = Black;
}
```

18/01/17

RB_Delete



BST_Delete :

Case 1: If y does not have any child, then just delete y.

Case 2: If y has only one left or right child, make that child as the child of p.

Case 3: Otherwise, replace y with in order successor.

- If y->color == Black, property 4 may be violated.

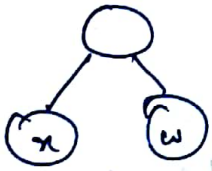
To satisfy property 4, we pass on the color of 'y' to its replacement 'x'. So, if x was previously 'red', it now becomes black. If x was previously black, it now becomes 'twice black'. [Doubly black]

There are two sets of symmetric cases depending on whether 'x' is a left or right child.

So, without loss of generality, we consider x to be the left child of its parent.

$x \Rightarrow$ doubly black

Case I:



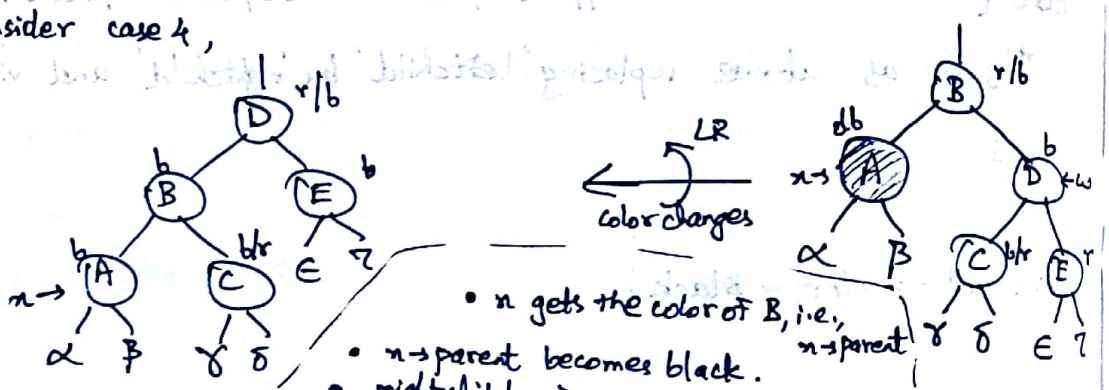
w is "red".

Case II: w is black, both the children of 'w' are black.

Case III: 'w' is black, left child of w is red and right child of w is black.

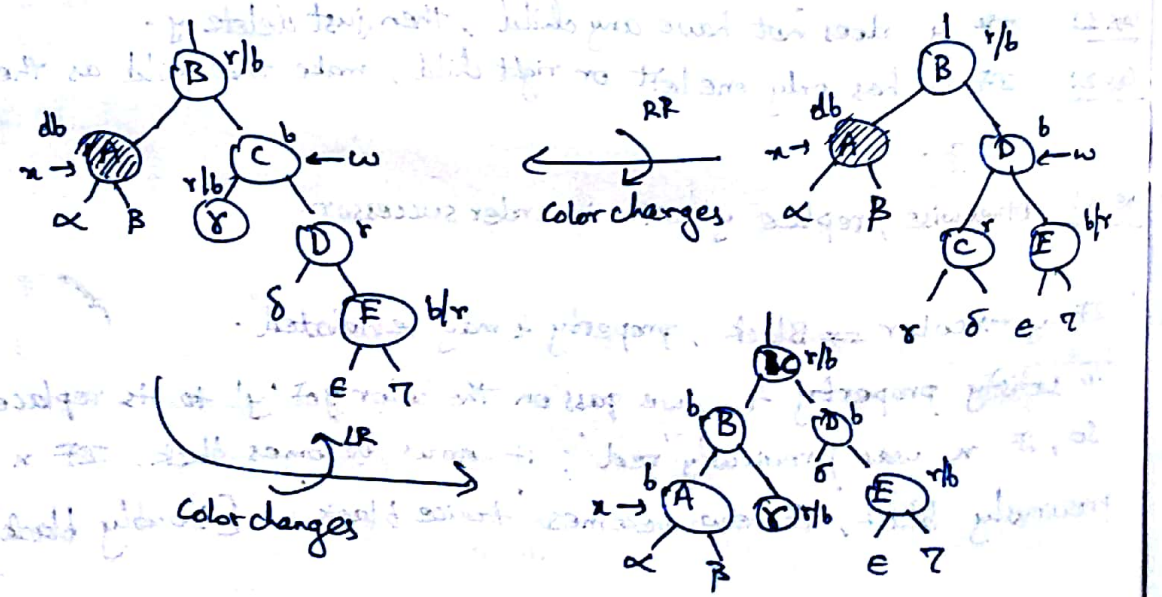
Case IV: 'w' is black, right child of w is red and left child of w is black.

- First consider case 4,

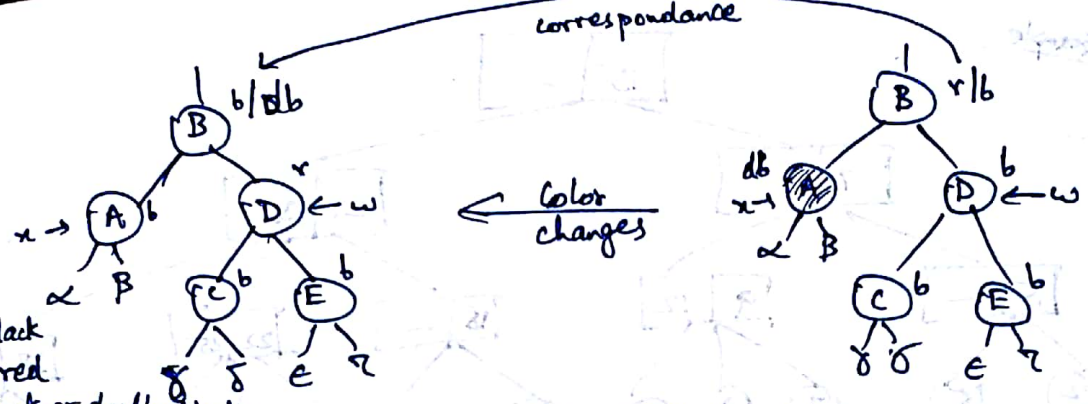


- x gets the color of B, i.e., x→parent
- x→parent becomes black.
- right child of w becomes black.
- left rotation
- x becomes black.

- Case 3,



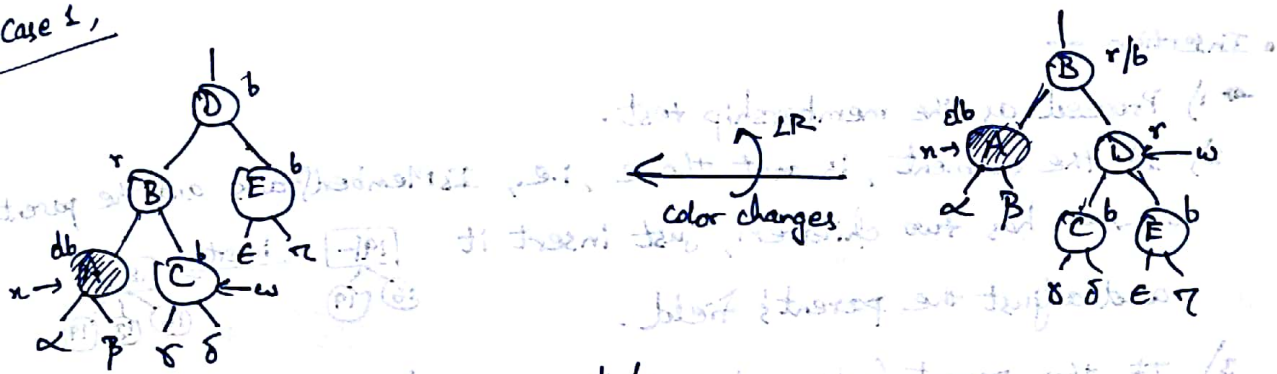
Case 2:



x → color = black
w → color = red
B becomes black or doubly black.

• Push up to the root, if db still persists in the tree.

Case 1,



Now Case 1 is shifted to Case 2/3/4 depending on colors of γ and δ.

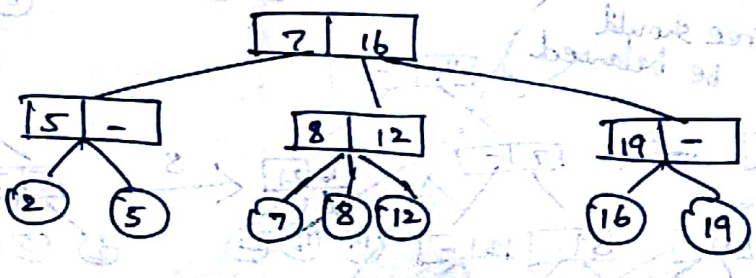
Search Trees (contd.)

2 Classes ↑

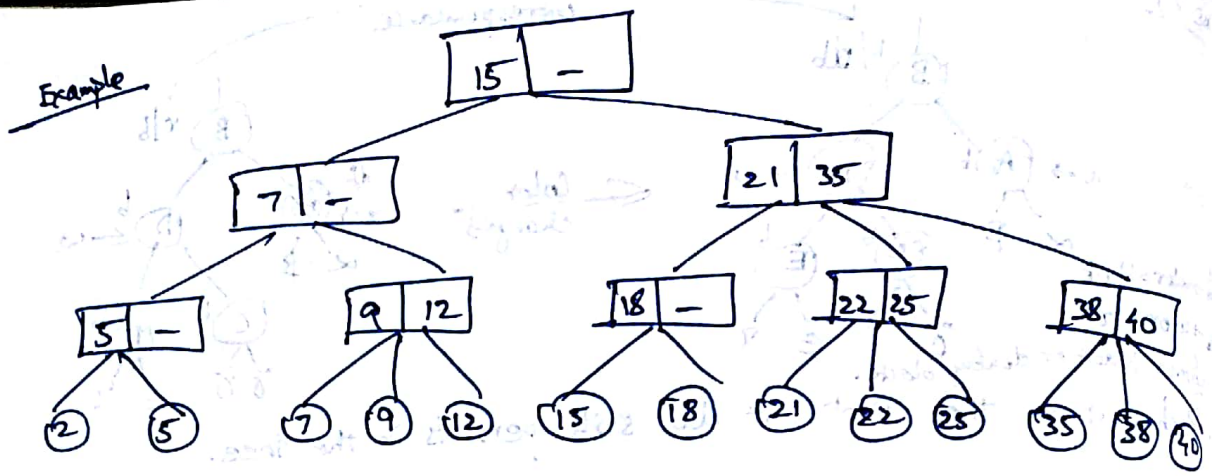
2-3 Tree (Hopcroft, 1966)

Characteristics:

1) The element occurs only at the leaves.
2) Each interior node (also root) has either 2 or 3 children.
3) Each path from the root to a leaf has the same length.
4) Each interior node (including the root) has 2 field
   Field 1: Smallest element of the tree rooted at the 2nd child
   Field 2: " " " " " " " " 3rd child, if any else blank
5) As a special case, a interior node may have one child or no child.



Example

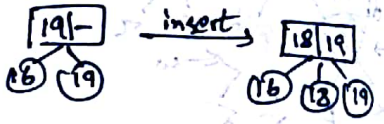


Implementation:

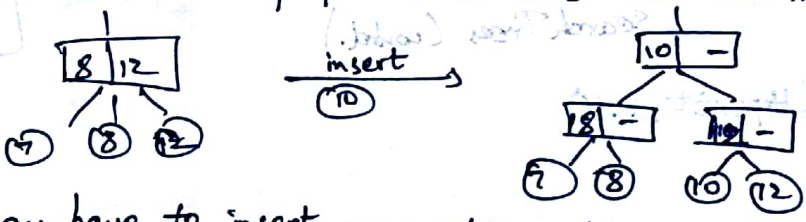
- Insertion —

1) Proceed as the membership test.

2) If the element, is not there, i.e., IsMember() fails and the parent of the has two children, just insert it and adjust the parent's field.



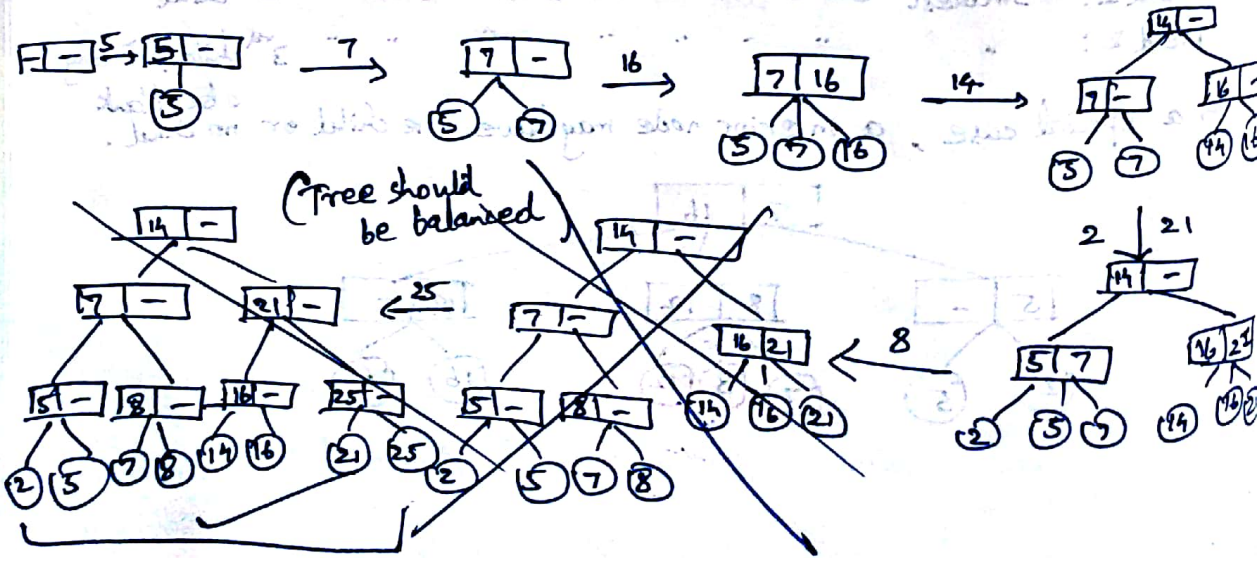
3) If the parent (where it is to be inserted) has 3 children already split it into two nodes one having the smaller two children and the other having the greater two children and proceed recursively to adjust the parents so that all the properties of 2-3 Tree are intact.

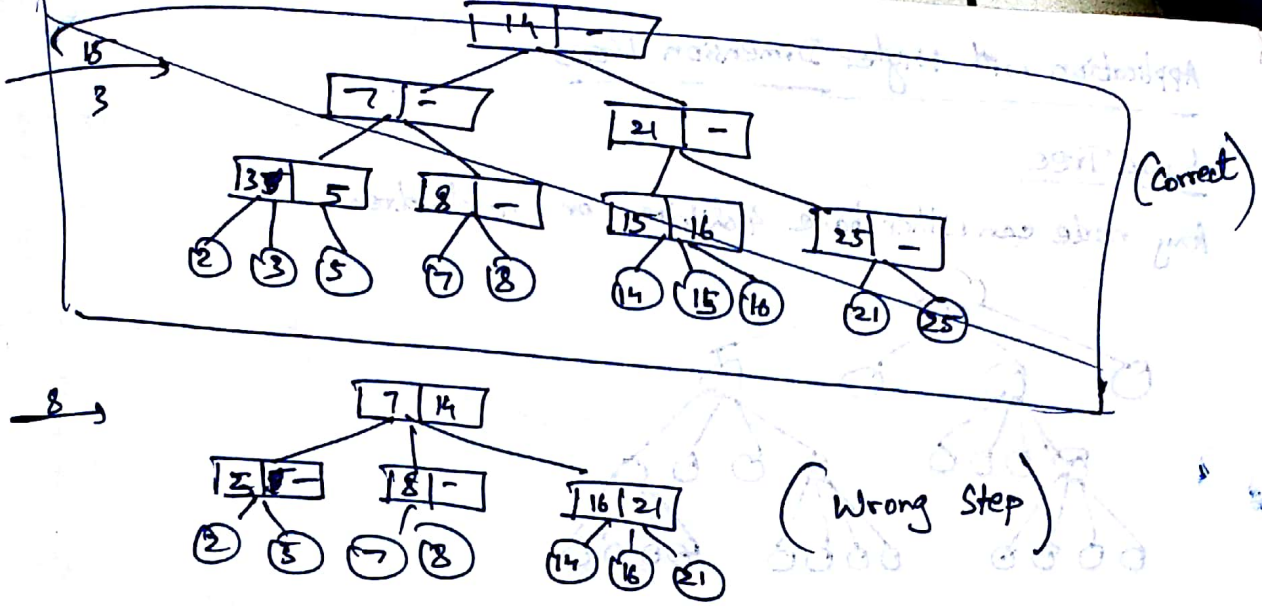


. You may have to insert new nodes and proceed recursively, also you may have a new root.

Q Construct a 2-3 Tree when elements are arriving in the order —

5, 7, 16, 14, 2, 21, 8, 25, 15, 3





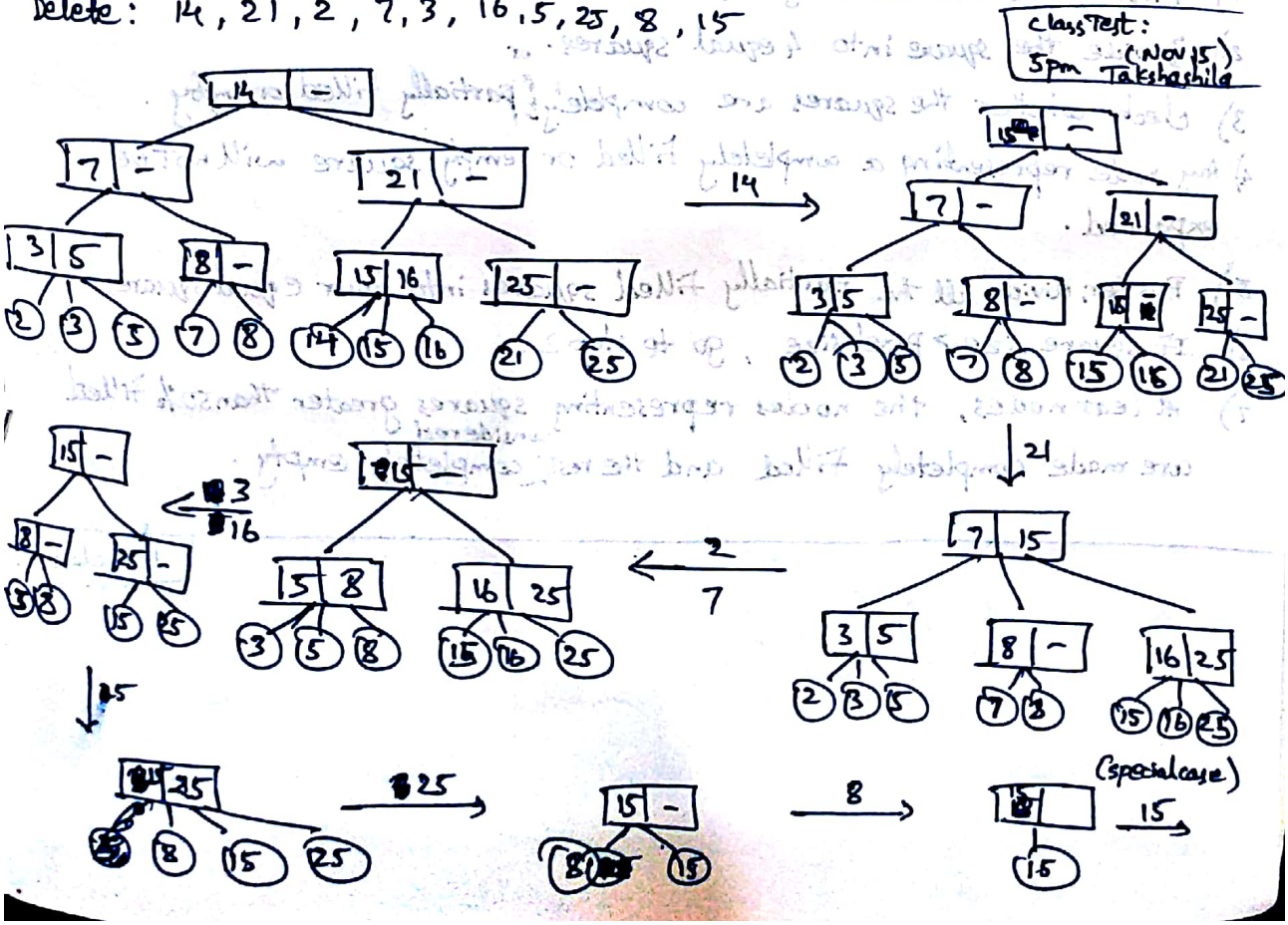
Delete —

1) If the ~~field~~ element to be deducted has got two siblings, just delete and adjust parent fields.

2) Else, suppose that the element has got one sibling, then its deletion results in its parent node 'p' having only one child.
If p has a sibling with 3 children, then we can transfer one of its child else, tx. the lone child of p to one of its sibling and delete p.
Now proceed recursively.

3) If the new pointer reaches the root and the root happens to have a single child, then delete root and make its new child as the root.

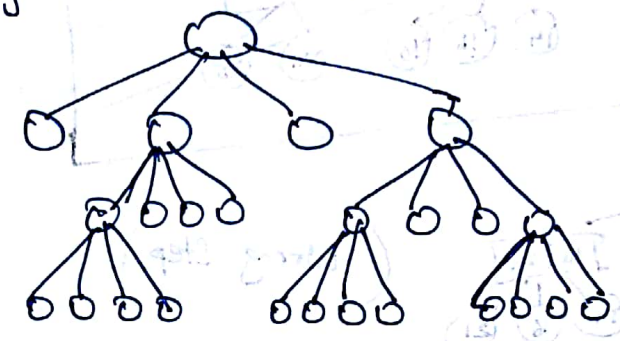
Delete: 14, 21, 2, 7, 3, 16, 5, 25, 8, 15



Application with Higher Dimension Trees

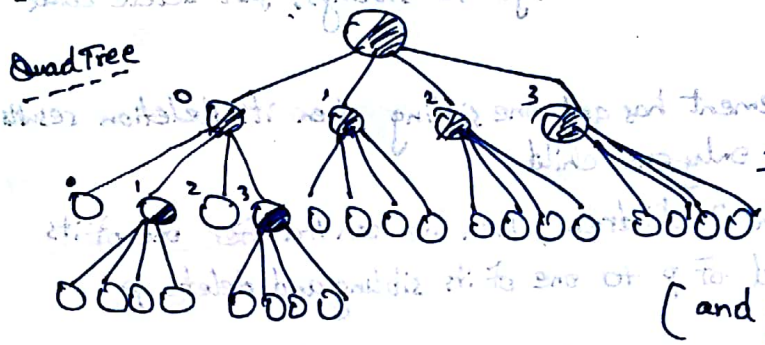
Quad Tree

Any node can either have 4 children or no children.



Oct Tree

Any node can either have 8 children or no children. ⇒ could be used for 3D image



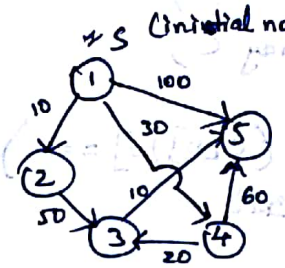
3D pixel – voxel

1) First find the bounding square.
2) Divide the square into 4 equal squares. or
3) check whether the squares are completely & partially filled or empty.
4) Any node representing a completely filled or empty square will not be expanded.
5) Further, divide all the partially filled squares into four equal squares.
6) If square size > pixel size, go to step 2.
7) At leaf nodes, the nodes representing squares greater than 50% filled considered are made completely filled and the rest completely empty.

1-2 classes

Graph Algorithms (contd.)

Dijkstra's shortest path Algorithm



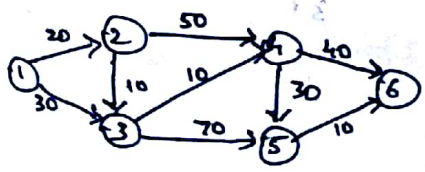
	1	2	3	4	5
1	0	10	∞	30	100
2	∞	0	50	∞	∞
3	∞	∞	0	∞	10
4	∞	∞	20	0	60
5	∞	∞	∞	∞	0

* V set of all vertices
* s=1 (without loss of generality)

```
begin
  S := {1};
  For i:=2 to n do
      D[i] := c[1,i]; // initialize D
  For i:=1 to n-1 do begin
  → choose a vertex w in V-S such that D[w] is minimum;
  → add w to S;
  → For each vertex v in V-S do
        D[v] := min[D[v], D[w] + c[w,v]];
    end;
  end;
```

Iteration	S	w	D[2]	D[3]	D[4]	D[5]
Initial	{1}	–	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,3,4,3,5}	5	10	50	30	60

Eg



S	D[2]	D[3]	D[4]	D[5]	D[6]
{1}	20	30	∞	∞	∞
{1,2}	20	30	70	∞	∞
{1,2,3}	20	30	40	100	∞
{1,2,3,4}	20	30	40	70	80
{1,2,3,4,5}	20	30	40	70	80
{1,2,3,4,5,6}	20	30	40	70	80

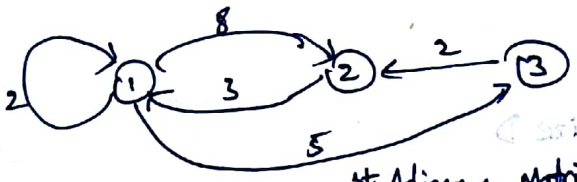
```
Dijkstra (c)
2D array c)
{
  if (c[0][0] == 0)
    exit;
```

(complete on your own)

9/11/17

- A graph is minimally connected or not (can be tested by Warshall's algorithm)

Warshall's Algorithm



Cost Matrix

	1	2	3
1	2	8	5
2	3	0	0
3	0	2	0

1st Adjacency Matrix

	1	2	3
1	1	1	1
2	1	0	0
3	0	1	0

(We convert the cost matrix to an adjacency matrix

Procedure Warshall (Var A : array [1....n, 1....n] of boolean ; c : array[of b

```
var
i,j,k : integer;
begin
for i:=1 to n do
   for j:=1 to n do
      A[i,j] := c[i,j];
For k:=1 to n do
   for i:=1 to n do;
      for j:=1 to n do;
         if [A[i,j]=0] else then
            A[i,j] := A[i,k] and A[k,j];
end; {warshall}
```

2nd Adjacency Matrix

	1	2	3
1	1	1	1
2	1	1	1
3	0	1	0

3rd Adjacency Matrix

	1	2	3
1			
2			
3			

All pair shortest path problem :

Floyd's algorithm

```
procedure 'Floyd' (Var A, c)
var
i,j,K: integer;
begin
for i:= 1 to n do
```

```
        A[i,j] := c[i,j] ;
for i := 1 to n do
    A[i,i] := 0 ;
for k := 1 to n do
    for i := 1 to n do
        for j := 1 to n do
            if A[i,k] + A[k,j] < A[i,j] then
                A[i,j] := A[i,k] + A[k,j] ;
end ;
```

(1)

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

(2)

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

(Can make 3 matrices on iterations for each row).

```
Warshall (c)
int array c[n][n] ; int n ;
{
    int
    for(i=0 ; i<n ; i++)
        For(j=0 ; j<n ; j++)
```

(Complete later)

Transitive closure [: : :]

→ If every node is connected or not!

* We try to show transitive closure using warshall's algorithm

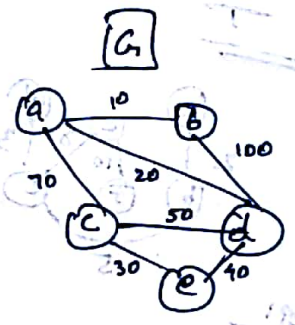
---

Undirected graphs

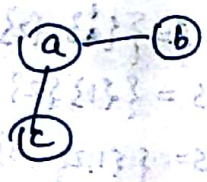
Spanning Tree

Subgraph of $G'(V', E')$ when 1. $V' \subseteq V$
2. $E' \subseteq E$

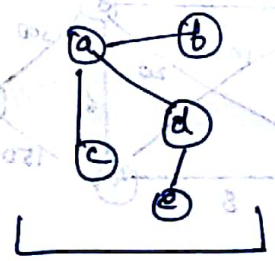
$G(V, E)$ such that $(v, w) \in E'$ $v', w' \in V'$



G' can be

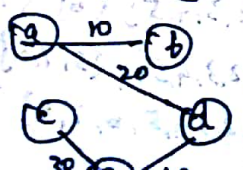


or



Spanning Tree ⇒ Since it has all nodes

Min weight Spanning Tree



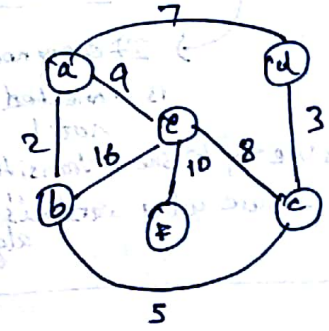
(Use the min weights & don't make loops)

Kruskal's Algorithm

We maintain two sets T & S such that initially

```
T ← ϕ ;
S ← ϕ ;
For each v ∈ V, do S ∪ {{v}} ;   ( {{u}, {w}} )
while |s| > 1 do begin
    choose (v,w) ∈ E of lowest cost ;
    delete (v,w) from E ;
    if v and w are in different sets w1 & w2 in S
    then
        begin
            replace w1 and w2 by w1 ∪ w2 in S
            add (v,w) to T ;
        end ;
end ;
end (Kruskal) ;
```

E = (a,b,c,d,e,f)



S = {{a},{b},{c},{d},{e},{f}}

Find the lowest cost —

S = {{a,b},{c},{d},{e},{f}} del path a-b from E

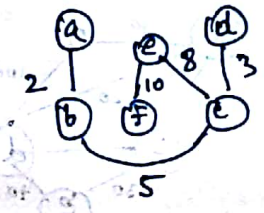
S = {{a,b},{c,d},{e},{f}} del path c-d from E

S = {{a,b,c,d},{e},{f}}

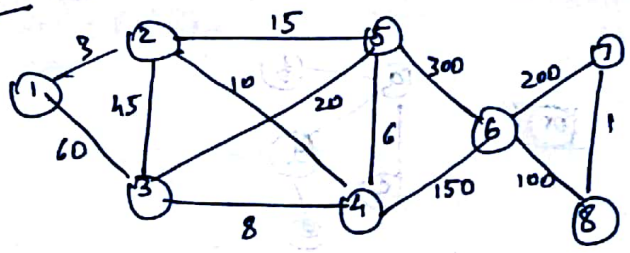
S = {{a,b,c,d,e},{f}}

S = {{a,b,c,d,e,f}}

T



eg:



S = {{1},{2},{3},{4},{5},{6},{7},{8}}

S = {{1}{2}{3}{4}{5}{6}{7}{8}}

S = {{1,2}{3}{4}{5}{6}{7,8}}

S = {{1,2}{3}{4,5}{6}{7,8}}

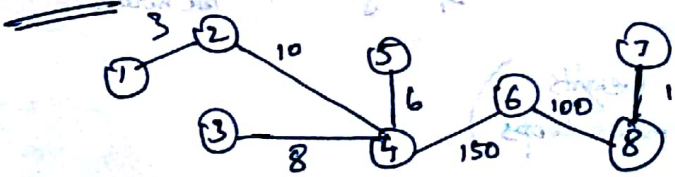
S = {{1,2}{3,4,5}{6}{7,8}}

S = {{1,2,3,4,5}{6}{7,8}}

S = {{1,2,3,4,5},{6,7,8}}

S = {{1,2,3,4,5,6,7,8}}

T



```
Kruskal (E)
int array * E [n];
```

n ⇒ no. of nodes in the graph

```
{ NODEPTR p; head;
p=create getnode (E[0]) ; head = p;
  For (i=1 ; i<n ; i++) {
     p→next = getnode (E[i]);
       p = p→next ;
    }
  while (head→next != NULL) {
   NODEPTR q ;
(x,y) = Findmincost (head);C );      // C ⇒ Cost Matrix
    if x
    p=head;
    For(i=1;i<n;i++) {
       if(p→ value == x)
          if(search (p,y) != 1)
             break ;
    }
  q= search (head, y) ;
   del (q);
  p→ value = append (y);
  append cost (T, (x,y));
}
```

Edge	(v,w)	Cost

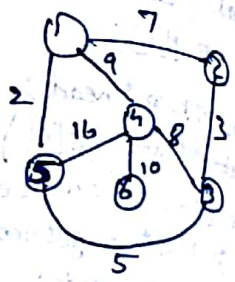
(can be written in a better way!)

Prim's algorithm

```
T ← ϕ
S ← ϕ
Initialize S as S ← {a} ; where a ∈ V
while S ≠ V do begin
    choose (v,w) ∈ E which are not -
    considered before with lowest cost
    such that v in S and w is in V-S.
      T ← T U {(v,w)} ;
      S ← S U {w} ;
  end ;
end;
```

$S = \{1\}$
$S = \{1, 5\}$
$S = \{1, 5, 3\}$
$S = \{1, 5, 3, 2\}$

$S = \{1, 5, 3, 2, 4\}$
$S = \{1, 5, 3, 2, 4, 6\}$



T