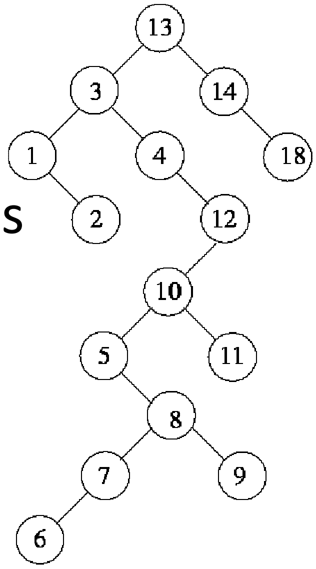
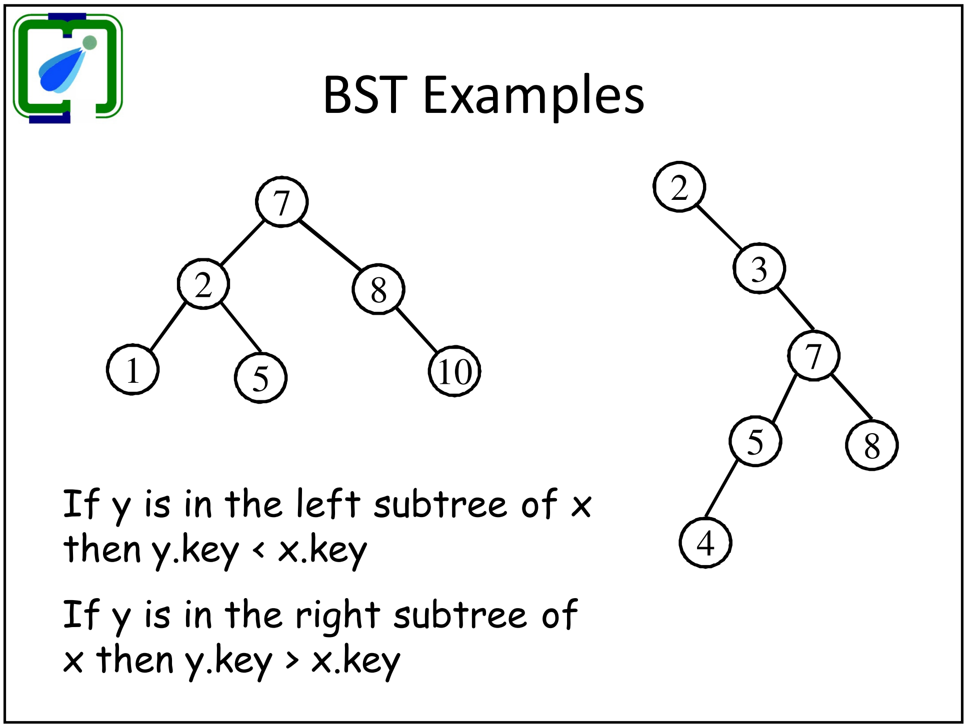
Binary Search Trees (BSTs)



Binary search trees

A binary tree data structure has the following properties:

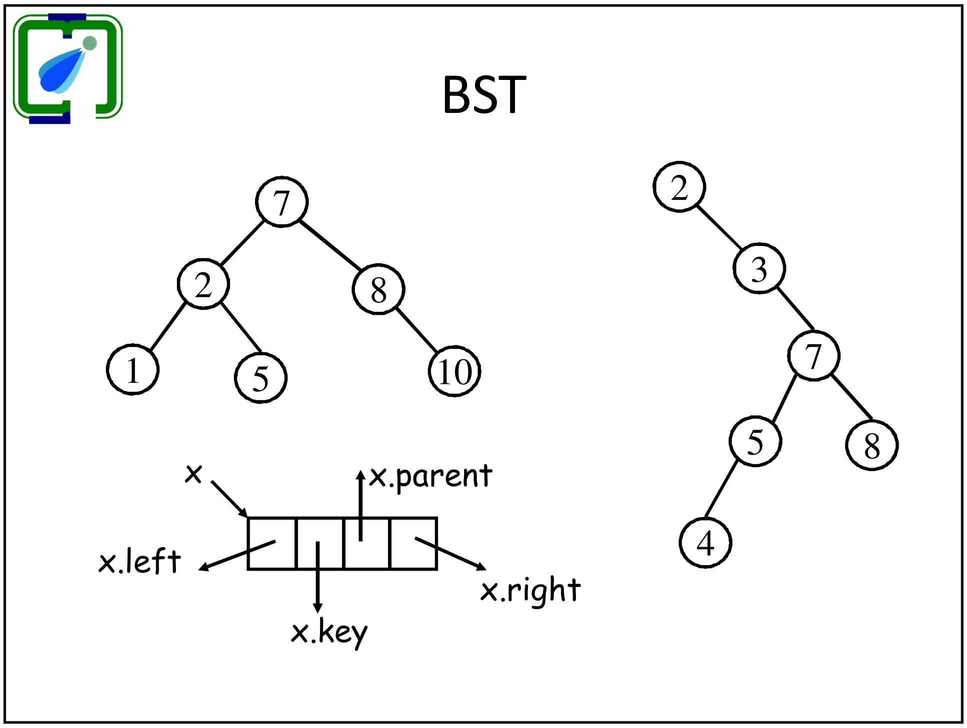
- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.



BST Examples

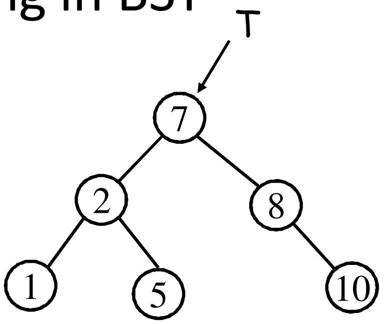
If y is in the left subtree of x then y.key < x.key

If y is in the right subtree of x then y.key > x.key



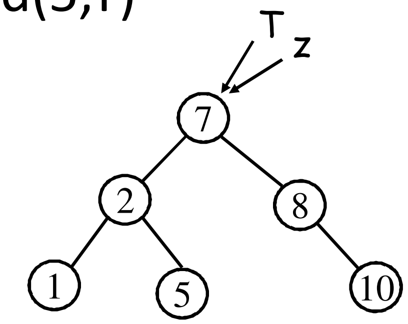
BST

Searching in BST



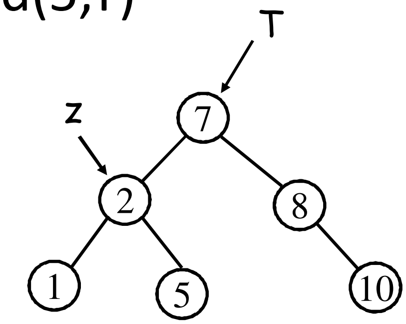
```
find(x,T)
z ← T.root
while z ≠ null do
        if  x = z.key return z
        if  x < z.key then z ← z.left
                      else z ← z.right
return z
```

Find(5,T)



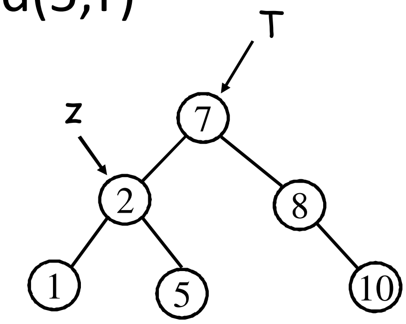
```
find(5,T)
z ← T.root
while z ≠ null do
        if  5 = z.key return z
        if  5 < z.key then z ← z.left
                      else z ← z.right
return z
```

Find(5,T)



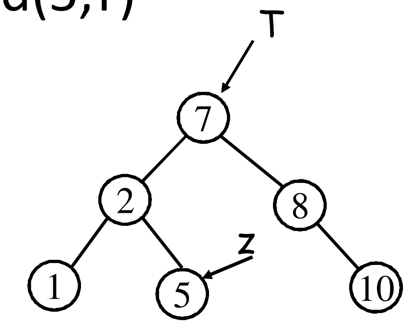
```
find(5,T)
z ← T.root
while z ≠ null do
        if  5 = z.key return z
        if  5 < z.key then z ← z.left
                      else z ← z.right
return z
```

Find(5,T)

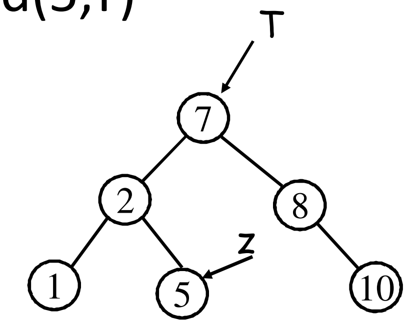


find(5,T)
z ← T.root
while z ≠ null do
        if 5 = z.key return z
        if 5 < z.key then z ← z.left
                      else z ← z.right
return z

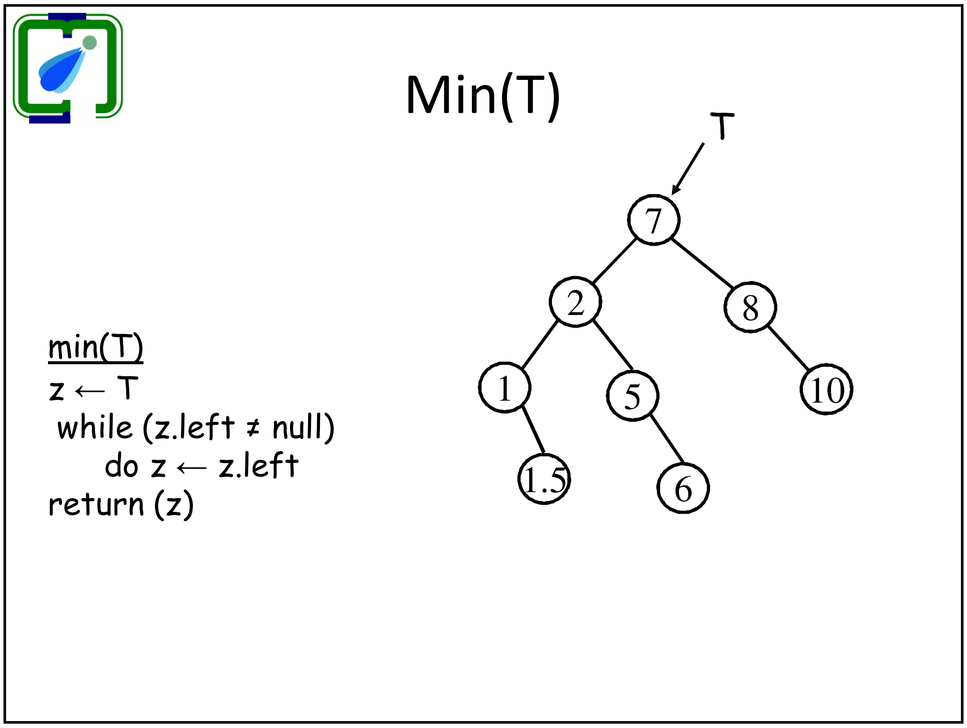
Find(5,T)



```
find(5,T)
z ← T.root
while z ≠ null do
        if  5 = z.key return z
        if  5 < z.key then z ← z.left
                      else z ← z.right
return z
```

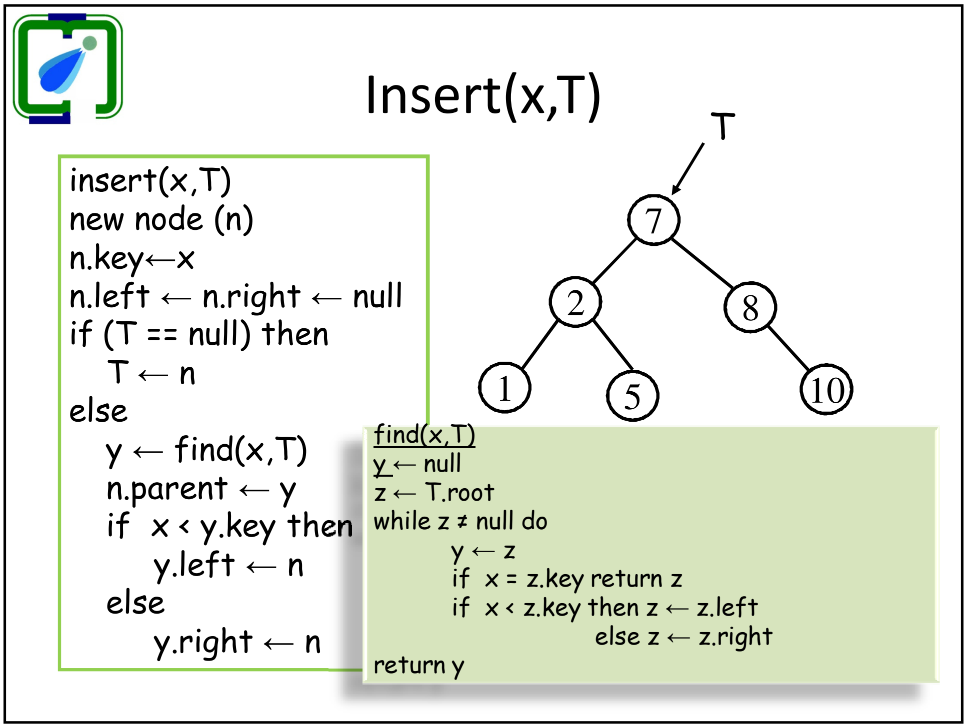

Find(5,T)

find(5,T)
z ← T.root
while z ≠ null do
        if 5 = z.key return z
        if 5 < z.key then z ← z.left
                      else z ← z.right
return z



Min(T)

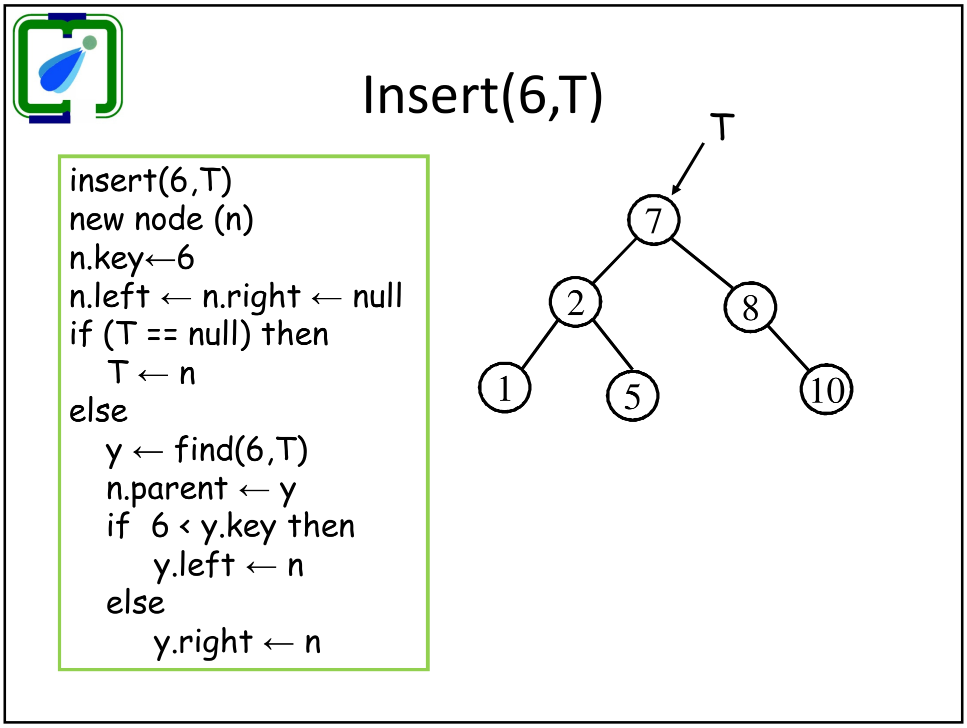
```
min(T)
z ← T
 while (z.left ≠ null)
      do z ← z.left
return (z)
```



Insert(x,T)

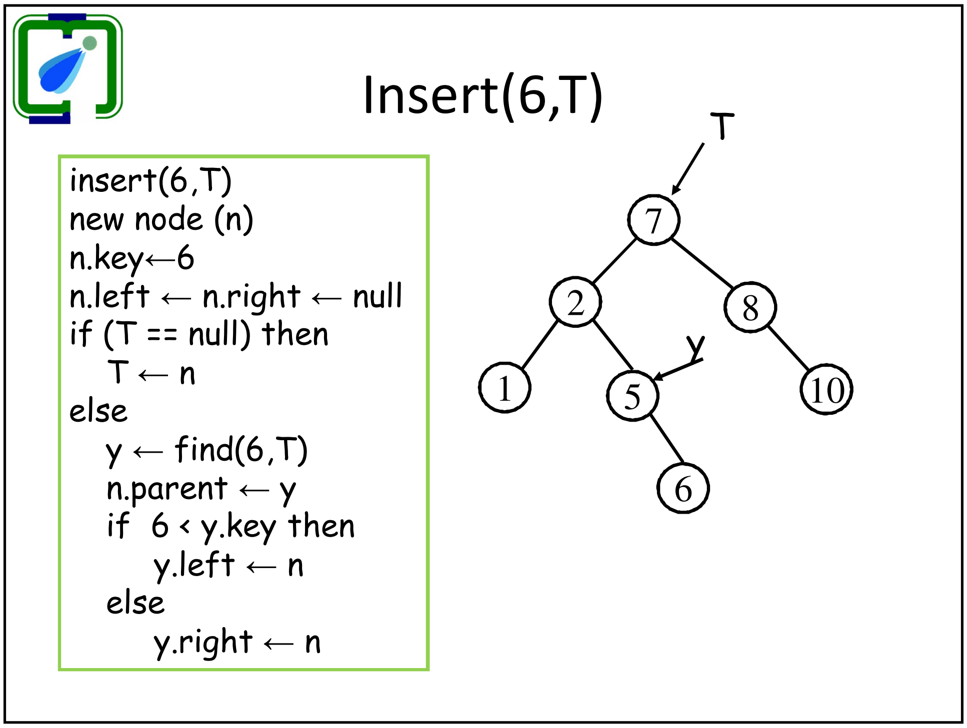
```
insert(x,T)
new node (n)
n.key←x
n.left ← n.right ← null
if (T == null) then
    T ← n
else
    y ← find(x,T)
    n.parent ← y
    if  x < y.key then
        y.left ← n
    else
        y.right ← n
```

```
find(x,T)
y ← null
z ← T.root
while z ≠ null do
        y ← z
        if  x = z.key return z
        if  x < z.key then z ← z.left
                       else z ← z.right
return y
```



Insert(6,T)

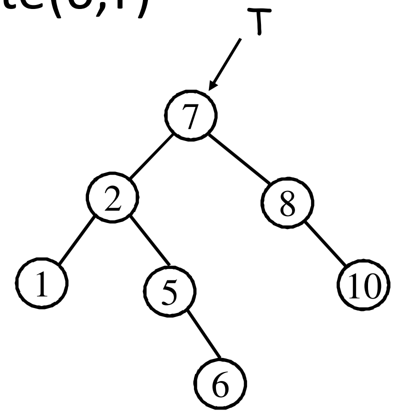
```
insert(6,T)
new node (n)
n.key←6
n.left ← n.right ← null
if (T == null) then
    T ← n
else
    y ← find(6,T)
    n.parent ← y
    if  6 < y.key then
        y.left ← n
    else
        y.right ← n
```



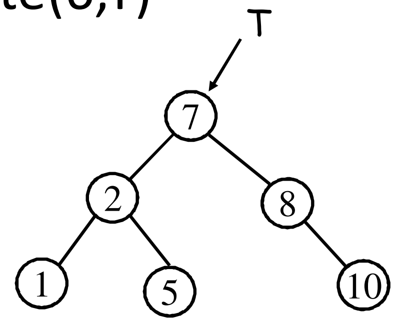
Insert(6,T)

```
insert(6,T)
new node (n)
n.key←6
n.left ← n.right ← null
if (T == null) then
    T ← n
else
    y ← find(6,T)
    n.parent ← y
    if  6 < y.key then
        y.left ← n
    else
        y.right ← n
```

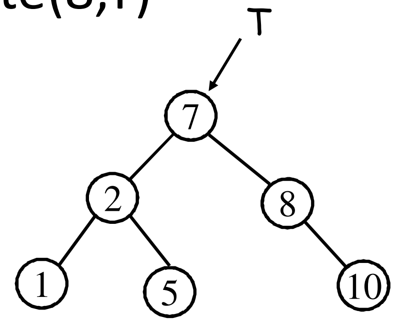
Delete(6,T)



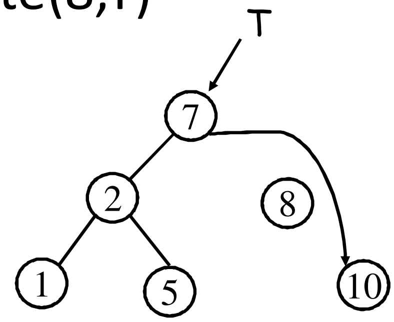
Delete(6,T)



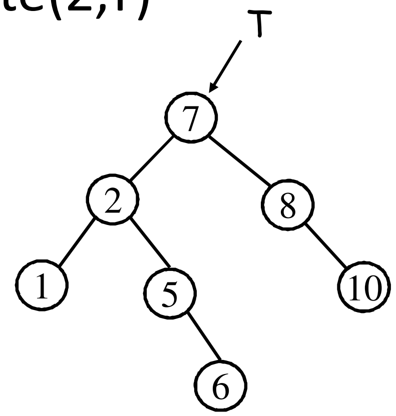
Delete(8,T)



Delete(8,T)

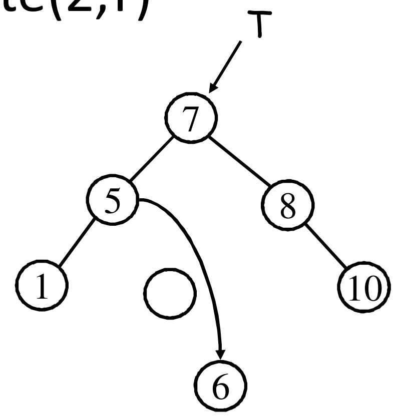


Delete(2,T)



Switch 5 and 2 and delete the node containing 5

Delete(2,T)



Switch 5 and 2 and delete the node containing 5

Deleting in BST

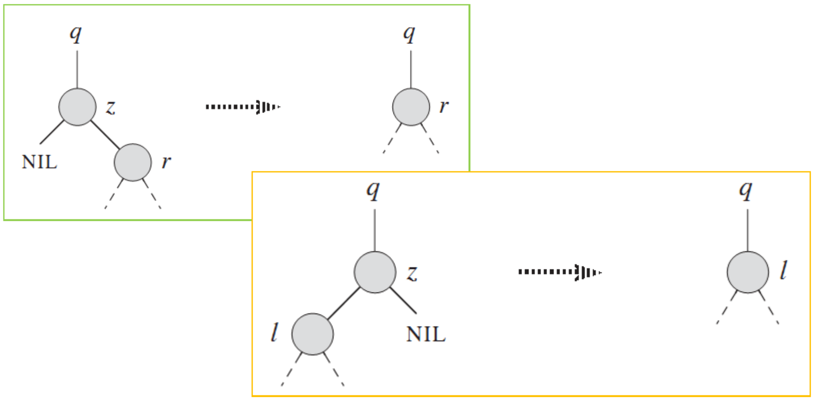
- Four cases

Deleting in BST

- Case – I: Deleting a leaf node

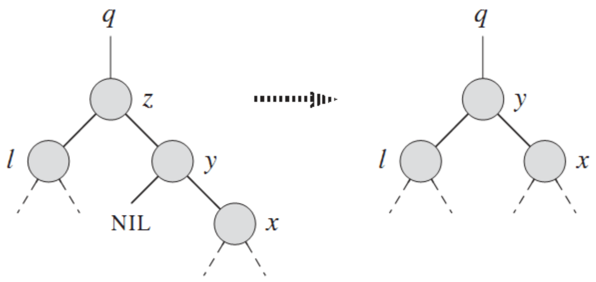
Deleting in BST

- Case – II: Deleting a node having either left or right child as empty (NIL)



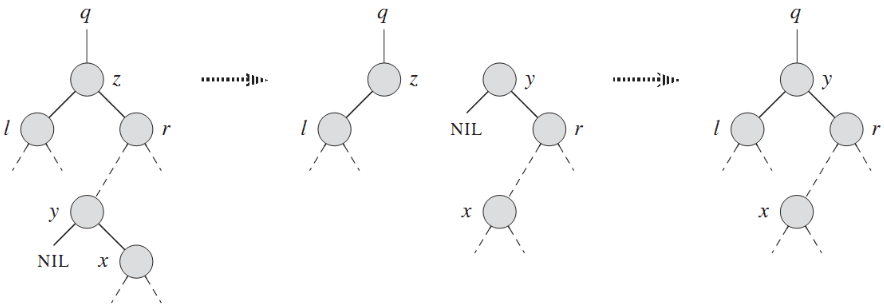
Deleting in BST

- Case – III(a): Deleting a node having neither left nor right child as empty (NIL)



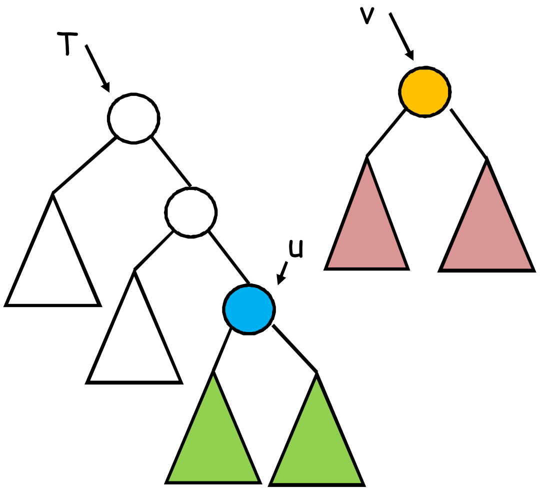
Deleting in BST

- Case – III(b): Deleting a node having neither left nor right child as empty (NIL)



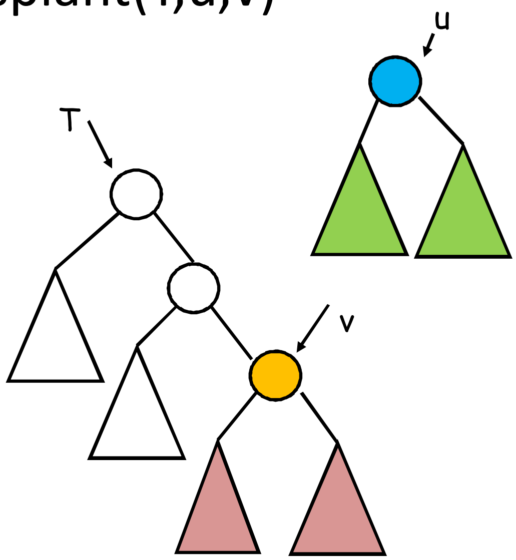
Transplant(T,u,v)

```
Transplant (T,u,v)
if u.p = null then
    T ← v
else if u == u.p.left
    u.p.left ← v
else
    u.p.right ← v
if v <> null
    v.p ← u.p
```



Transplant(T,u,v)

```
Transplant (T,u,v)
if u.p = null then
    T ← v
else if u == u.p.left
    u.p.left ← v
else
    u.p.right ← v
if v <> null
    v.p ← u.p
```

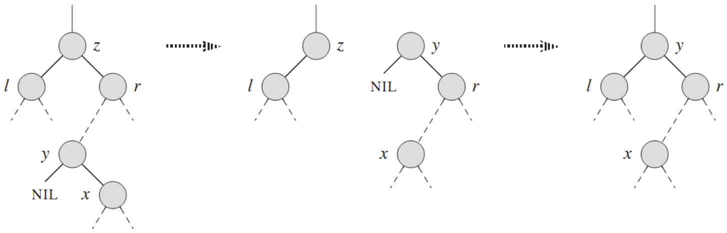


delete(T, z)

```
delete (T, z)
if (z.left == null) and (z.right == null) then        //Case I
    if z == z.p.left then
        z.p.left ← null
    else
        z.p.right ← null
else if (z.left == null)  then                         //Case II
    Transplant (T,z,z.right)
else if (z.right == null)  then
    Transplant (T,z,z.left)
else                                                    //Case III
…
```

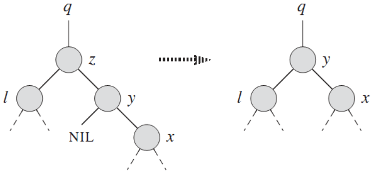
delete(T, z)

```
…
else                                    //Case III
     y = Tree-Minimum(z.right)
     if (y.p <> z) then                 //Case III (b)
         Transplant (T,y,y.right)
         y.right ← z.right
         y.left ← z.left
         z.right.p ← y
         z.left.p ← y
```

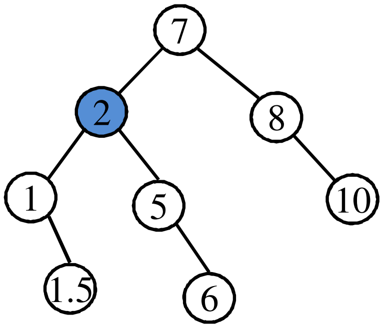


delete(T, z)

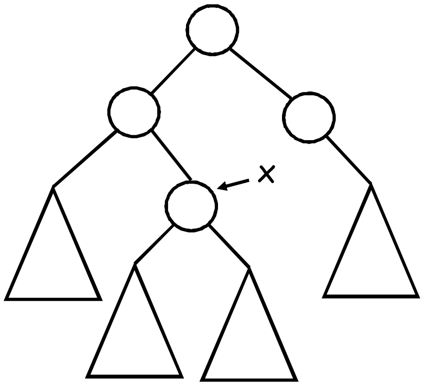
```
…
else                                    //Case III
     y = Tree-Minimum(z.right)
     if (y.p <> z) then                 //Case III (b)
         Transplant (T,y,y.right)
         y.right ← z.right
         y.left ← z.left
         z.right.p ← y
         z.left.p ← y
else                                //Case III (a)
         Transplant(T,z,y)
         y.left ← z.left
         y.left.p ← y
```



successor(x,T)

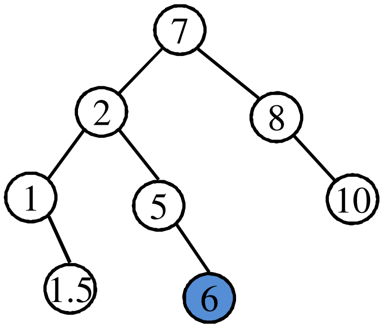


successor(x,T)



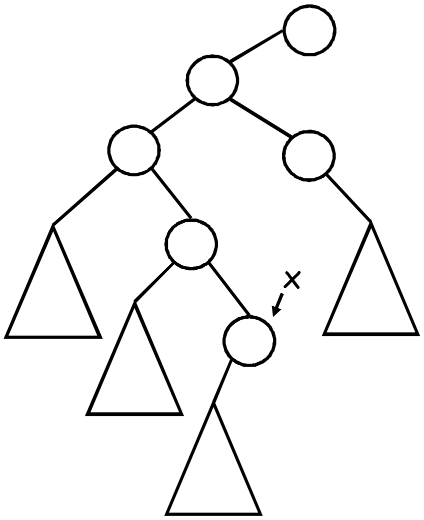
1. If x has a right child it's the minimum in the subtree of x.right

successor(x,T)



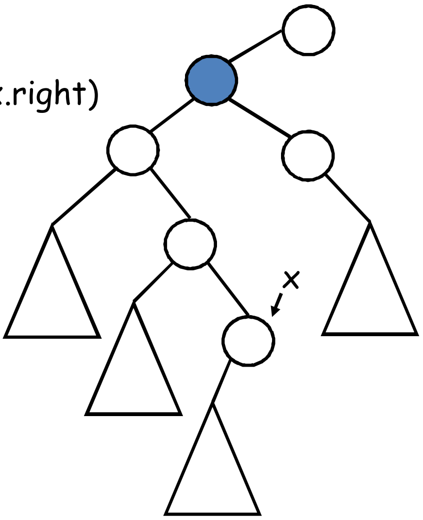
successor(x,T)

2. If x.right is null, go up until the lowest ancestor such that x is at its left subtree



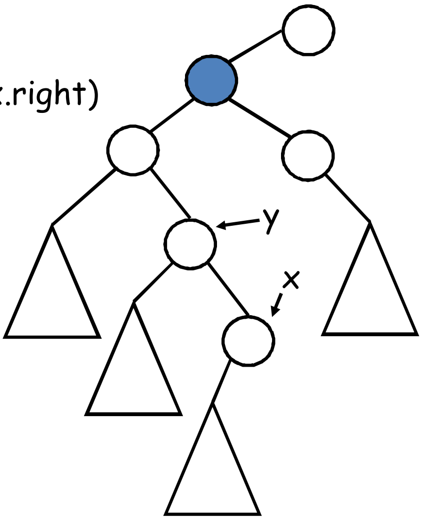
successor(x,T)

```
If x.right ≠ null then min(x.right)
y ← x.parent
While y≠null and x=y.right
    do  x ← y
        y ← y.parent
return(y)
```

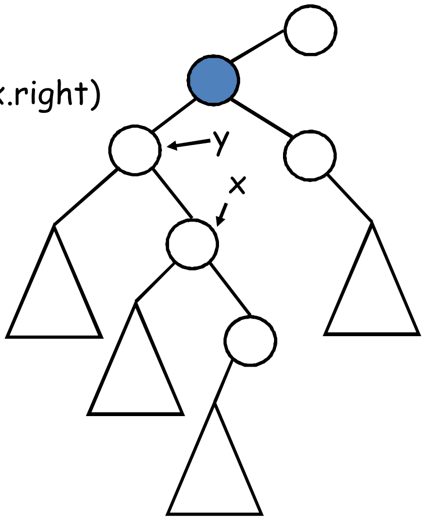


successor(x,T)

```
If x.right ≠ null then min(x.right)
y ← x.parent
While y≠null and x=y.right
    do  x ← y
        y ← y.parent
return(y)
```

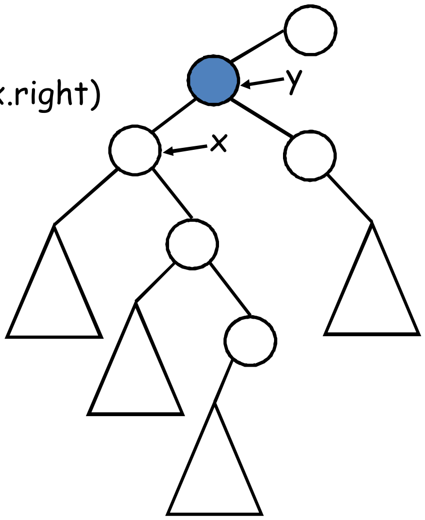


successor(x,T)



```
If x.right ≠ null then min(x.right)
y ← x.parent
While y≠null and x=y.right
    do  x ← y
        y ← y.parent
return(y)
```

successor(x,T)



```
If x.right ≠ null then min(x.right)
y ← x.parent
While y≠null and x=y.right
    do  x ← y
        y ← y.parent
return(y)
```

summary

- BST is an efficient search data structure if it is fairly balanced
- Complexity (if balanced)
  - Insertion O(log n)
  - Deletion O(log n)
  - Search O(log n)