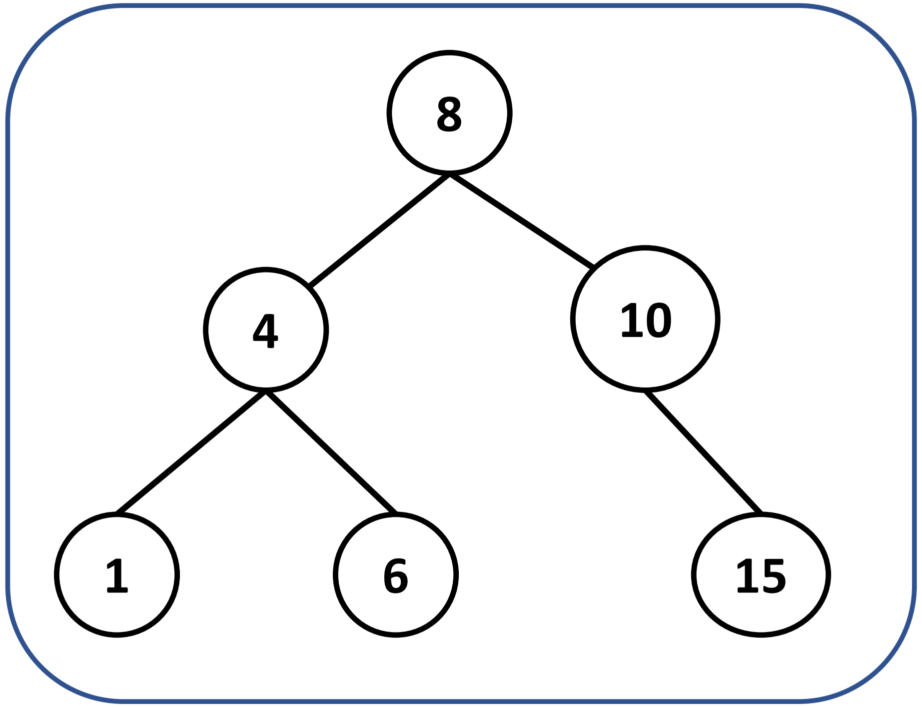
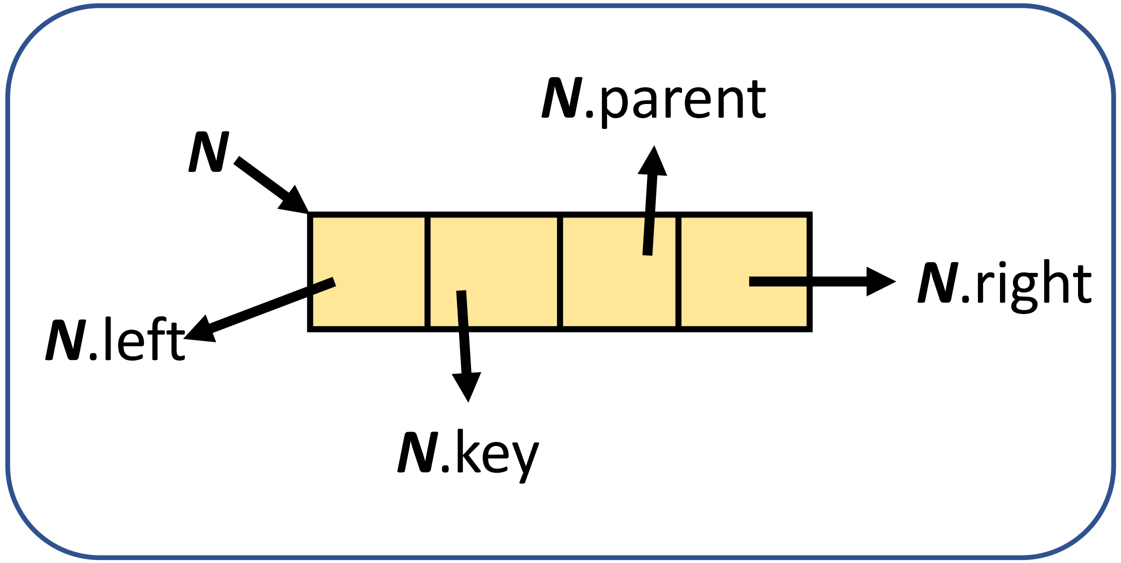
Binary Search Tree (BST)

PROF. NAVRATI SAXENA

Binary Search Trees

- A special tree with the following properties:
  1. Left subtree contains nodes with keys less than the parent's key
  2. Right subtree contains nodes with keys greater than the parent's key
  3. Both the left and right subtrees must also be binary search trees





Binary Search – Recap (1/3)

1. Sort the array/search space
2. Compare the key to be searched with the mid element
3. If the record being searched is less

        Search in the left half

    else

        Search in the right half

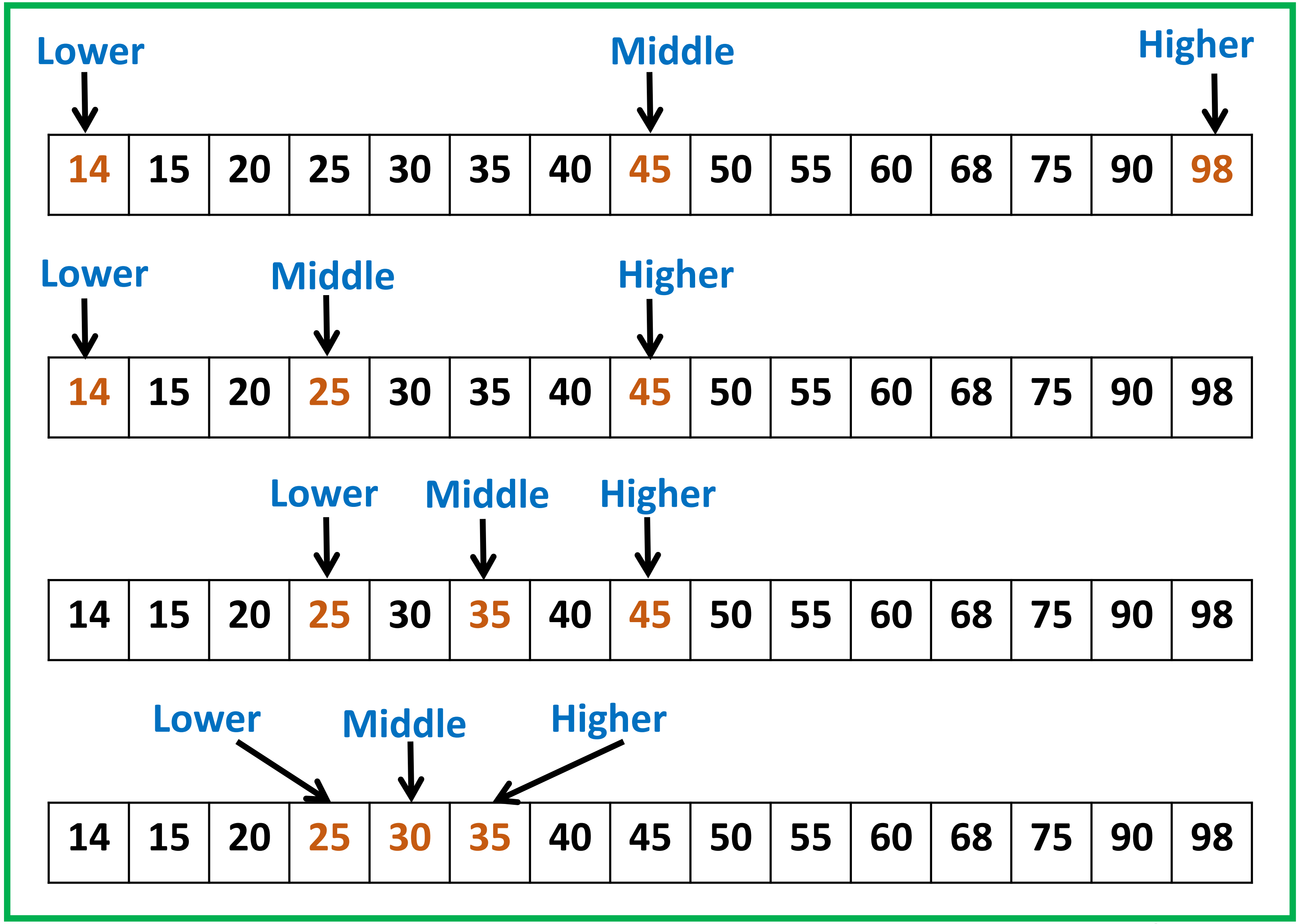
    In case of equality

        We found the element.

- Start with **'n'** elements in search space and then if mid element is not the element,
- Reduce the search space to **'n/2'** and
- Go on reducing the search space till we either find the element or
- We get to only one element in search space.

Binary Search – Recap (2/3)

- Sort the array
- Divides the array into two smaller sub-arrays
- Recursively operate the sub-arrays.
- Reduces the search space to half at each step



Binary Search Algorithm – Recap (3/3)

```
// A iterative binary search procedure. It returns location of item in given array arr[low..high] if present,
otherwise it returns: -1

int binarySearch (int arr[], int lower, int higher, int item)
{
    while (lower <= higher)
            {
            mid = lower + (higher - lower) / 2;
            if (arr[mid] = x)                          //Check if x is present at mid
                    return mid;
          if (arr[mid] < x)
                    lower = mid + 1;                   // If x greater, ignore left half
            else
                     higher = mid - 1;                 // If x is smaller, ignore right half
            }
       return -1;                                      // if we reach here, then element isnot present
}
```

Search in BST

1. **Searching in BST is very similar to Binary search**
2. Searching an element in BST is basically a traversal in which at each step
3. We will go either left or right and hence discard one of the sub-trees
4. The search here is also a binary search and that's why the name BST

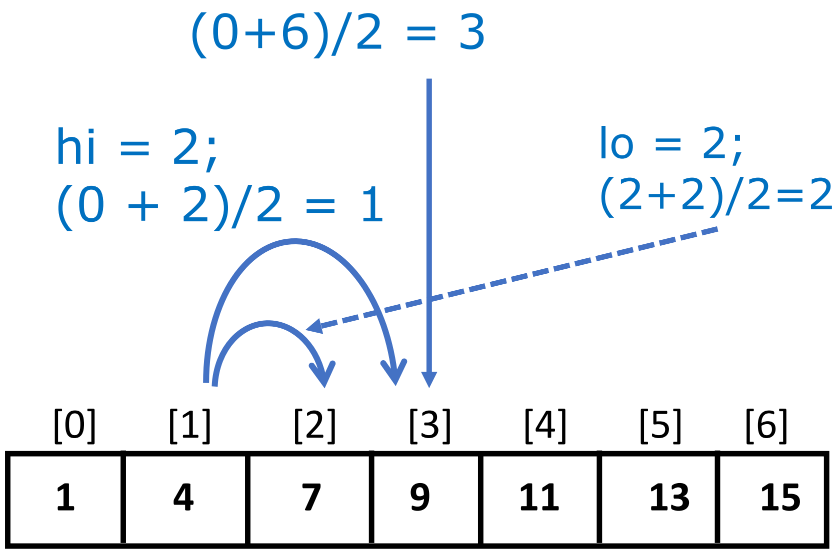
- Assume a search space of ***'n'*** nodes;
- Discarding a subtree means discarding **'n/2'** nodes, reducing search space to **'n/2'**
- Next step, reduce the search space to **'n/4'** and continue until
- We find the element or till our search space is reduced to only one node.

Complexity = O(log (n))

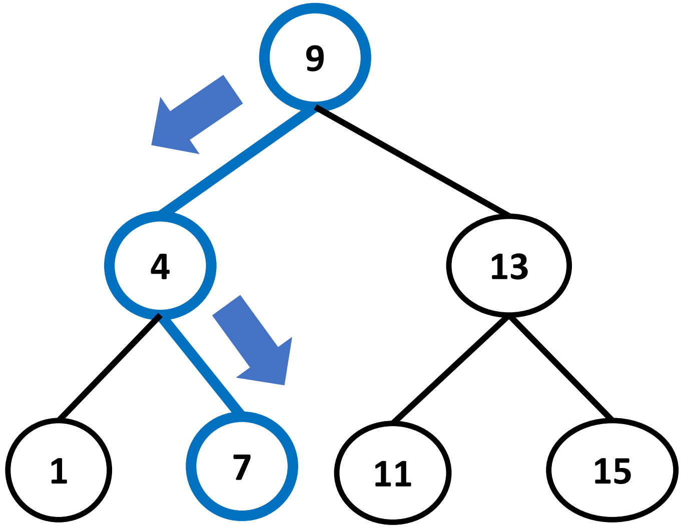
Binary Search in an Array

Look at array location (lo + hi)/2

Searching for 7:

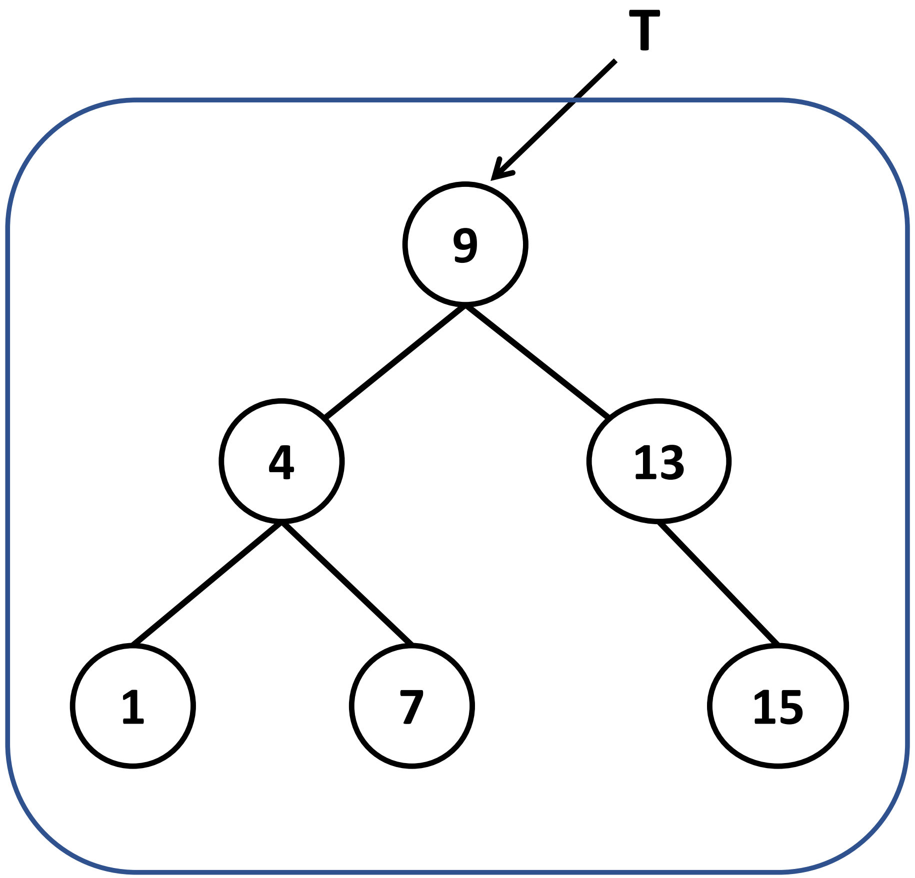


Using a BST



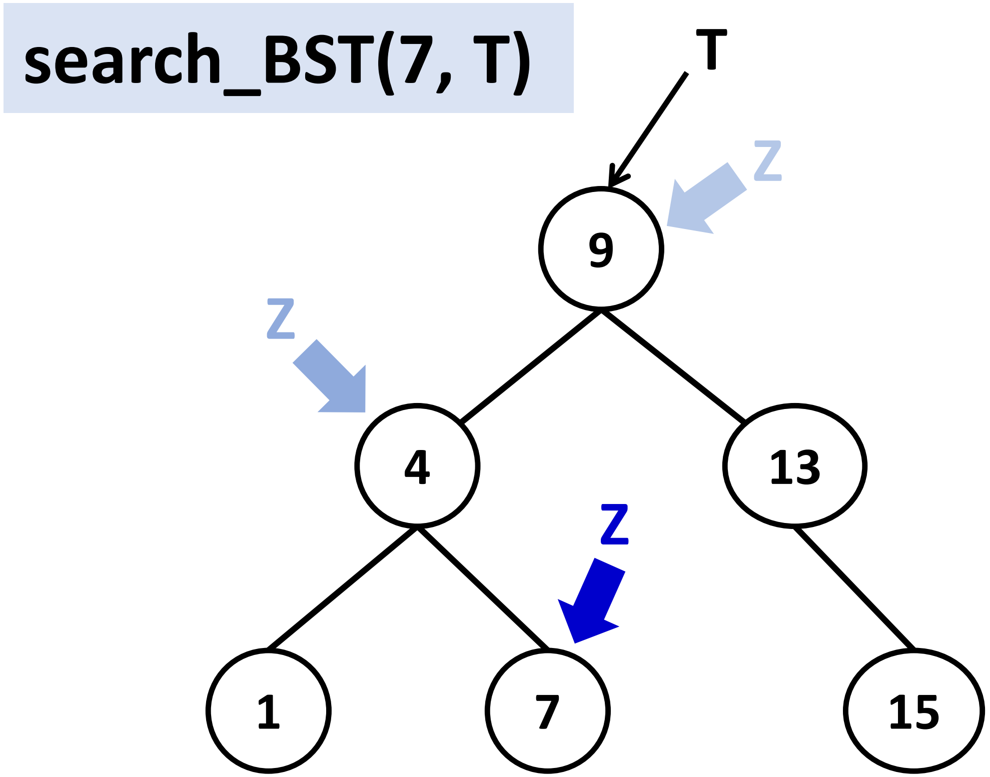
Searching in BST

```
search_BST(x,T) //x = item to be searched
{
 z ← T
 while z ≠ null do
   {
     if x == z.key
          return z
     else if x < z.key
          z ← z.left
     else
          z ← z.right
     }
return z
}
```



Example: Search an Element

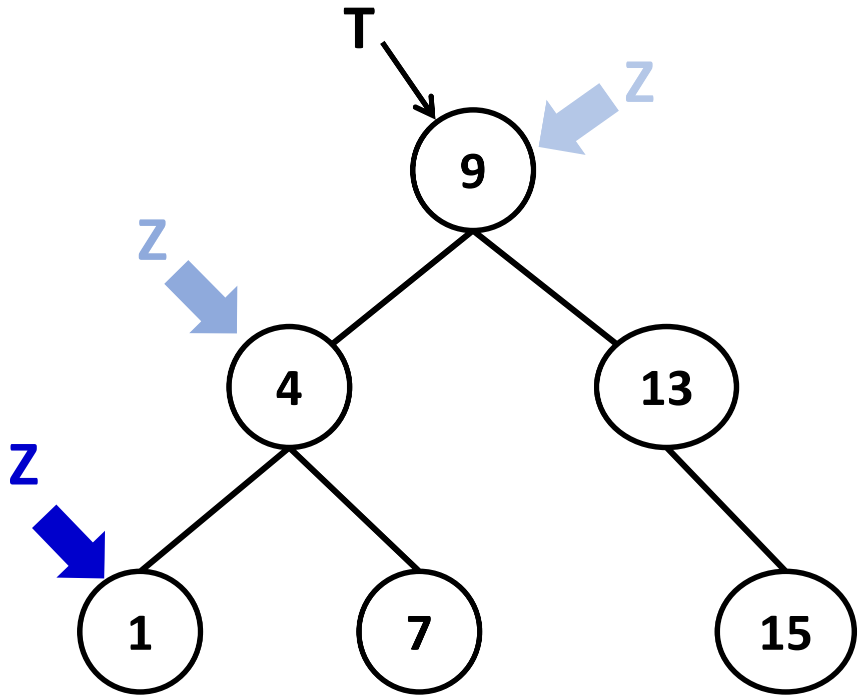
```
search_BST(7,T)
{
 z ← T
 while z ≠ null do
   {
     if 7 == z.key
          return z
     else if 7 < z.key
          z ← z.left
     else
          z ← z.right
     }
return z
}
```



Finding Minimum

Question: Where is the minimum element located?

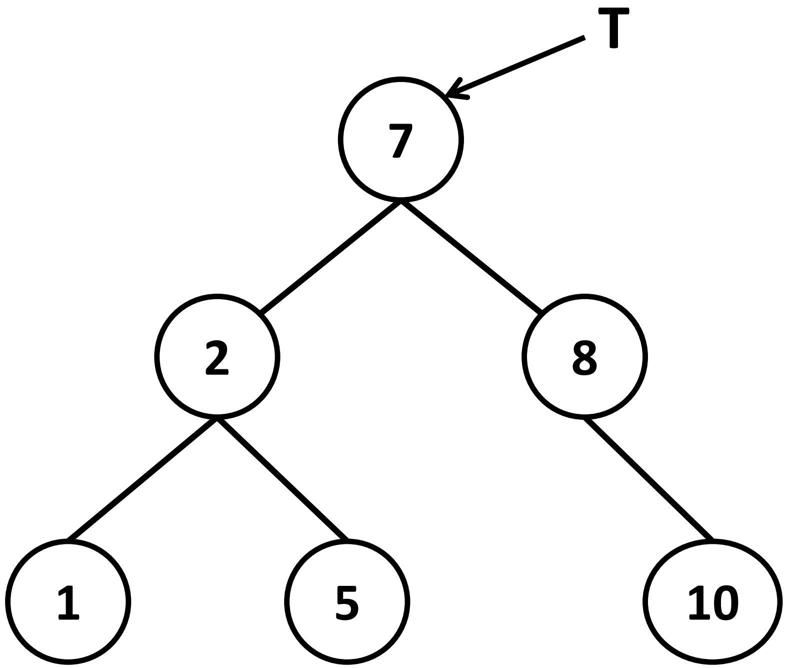
```
Min_BST(T)
{
z ← T
while (z.left ≠ null)
        z ← z.left
return (z)
}
```



Insertion in BST

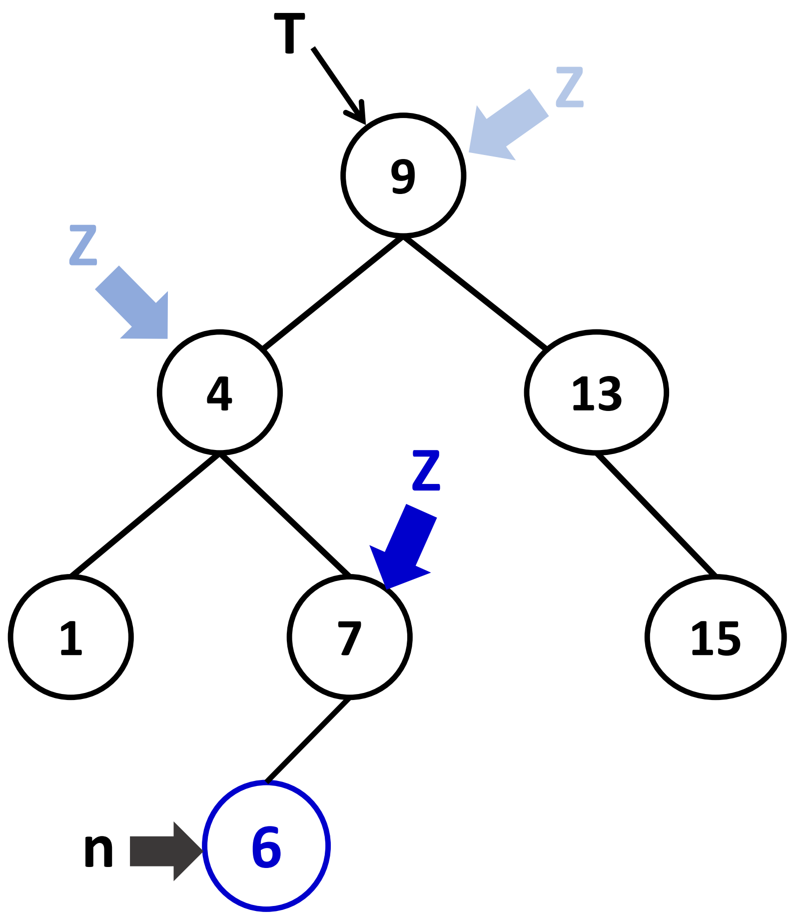
```
Insert_BST(x,T) {
 create new node (n)
 n.key←x
 n.left ← null
 n.right ← null
 if (T == null) then
     T ← n
else
     z ← search_BST(x,T)
     n.parent ← z
     if x < z.key then
          z.left ← n
     else
          z.right ← n
}
```

Discussed before



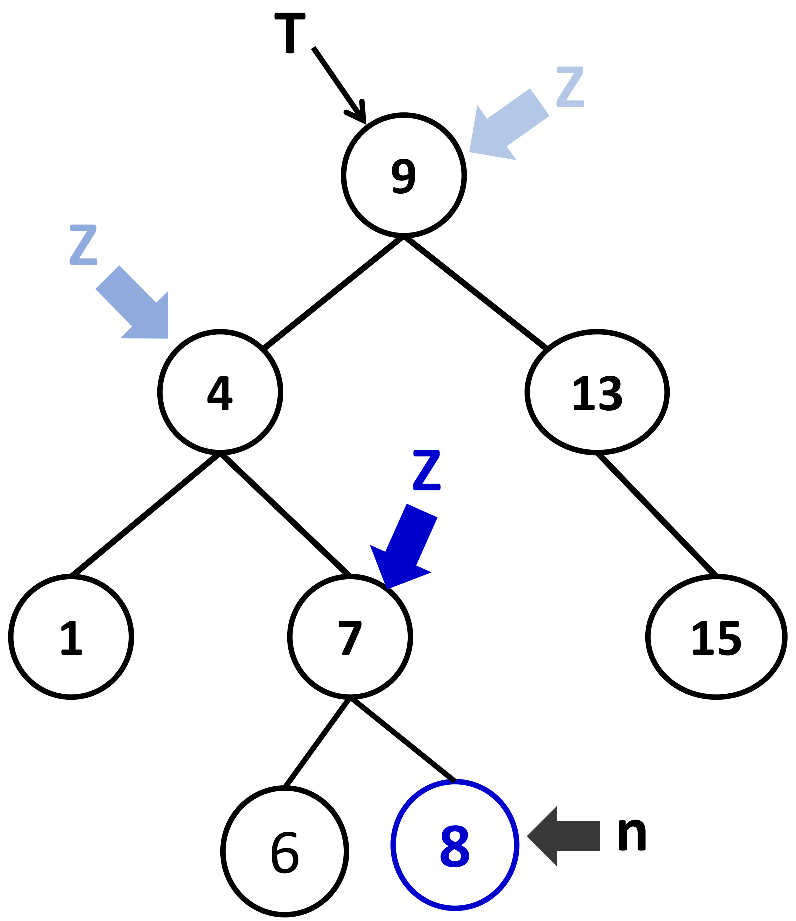
Example: Insertion in BST (1/2)

```
Insert_BST(6,T) {
 create new node (n)
 n.key←6
 n.left ← null
 n.right ← null
 if (T == null) then
     T ← n
else
     z ← search_BST(6,T)
     n.parent ← z
     if 6 < z.key then
          z.left ← n
     else
          z.right ← n
}
```



Example: Insertion in BST (2/2)

```
Insert_BST(8,T) {
 create new node (n)
 n.key←8
 n.left ← null
 n.right ← null
 if (T == null) then
     T ← n
else
     z ← search_BST(8,T)
     n.parent ← z
     if 8 < z.key then
          z.left ← n
     else
          z.right ← n
}
```



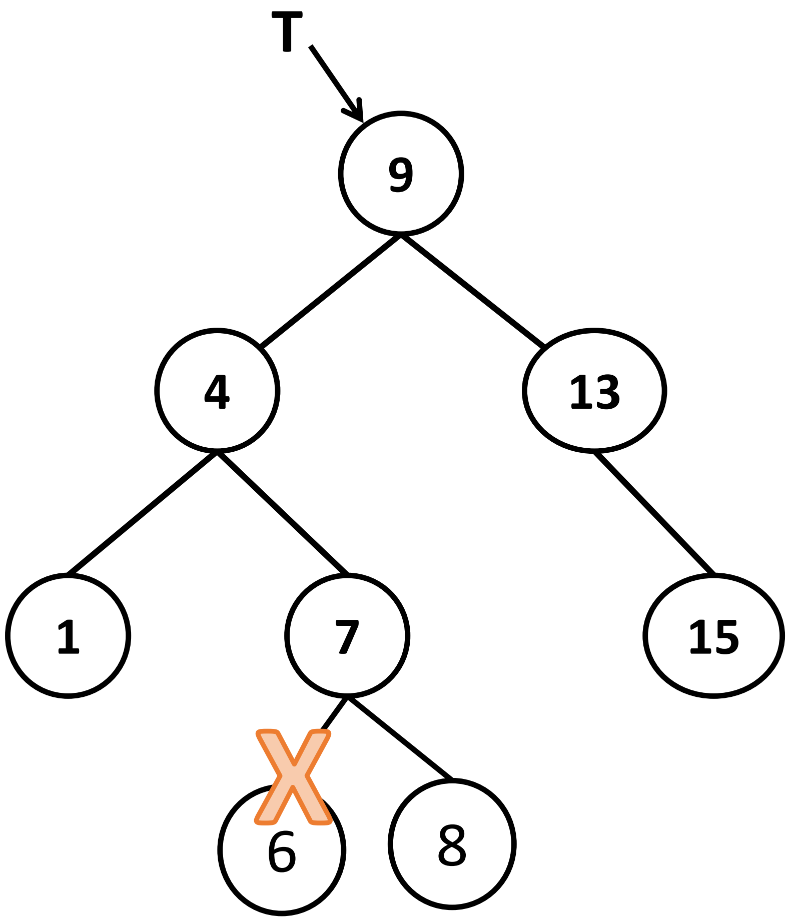
Example: Deletion in BST (1/3)

CASE 1

Delete(6,T)

- Simply delete the node, free memory
- Easy, but Why?

[Answer]: 6 is leaf-node



Example: Deletion in BST (2/3)

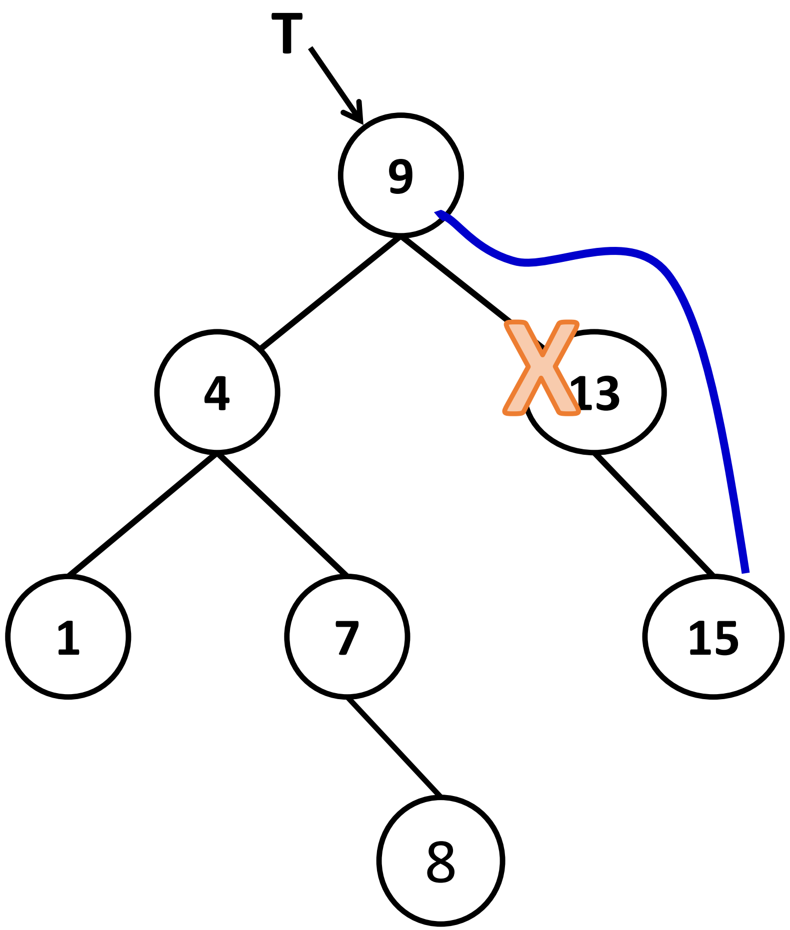
CASE 2

Delete(13, T)

- No more a leaf node

- However, has only one child
- Need to adjust the single **child below 13**

[Tip]: Make the child of target as:
child of (parent of (target))



Example: Deletion in BST (3/3)

CASE 3

Delete(4, T)

- No more a leaf node

- Has both the leaves

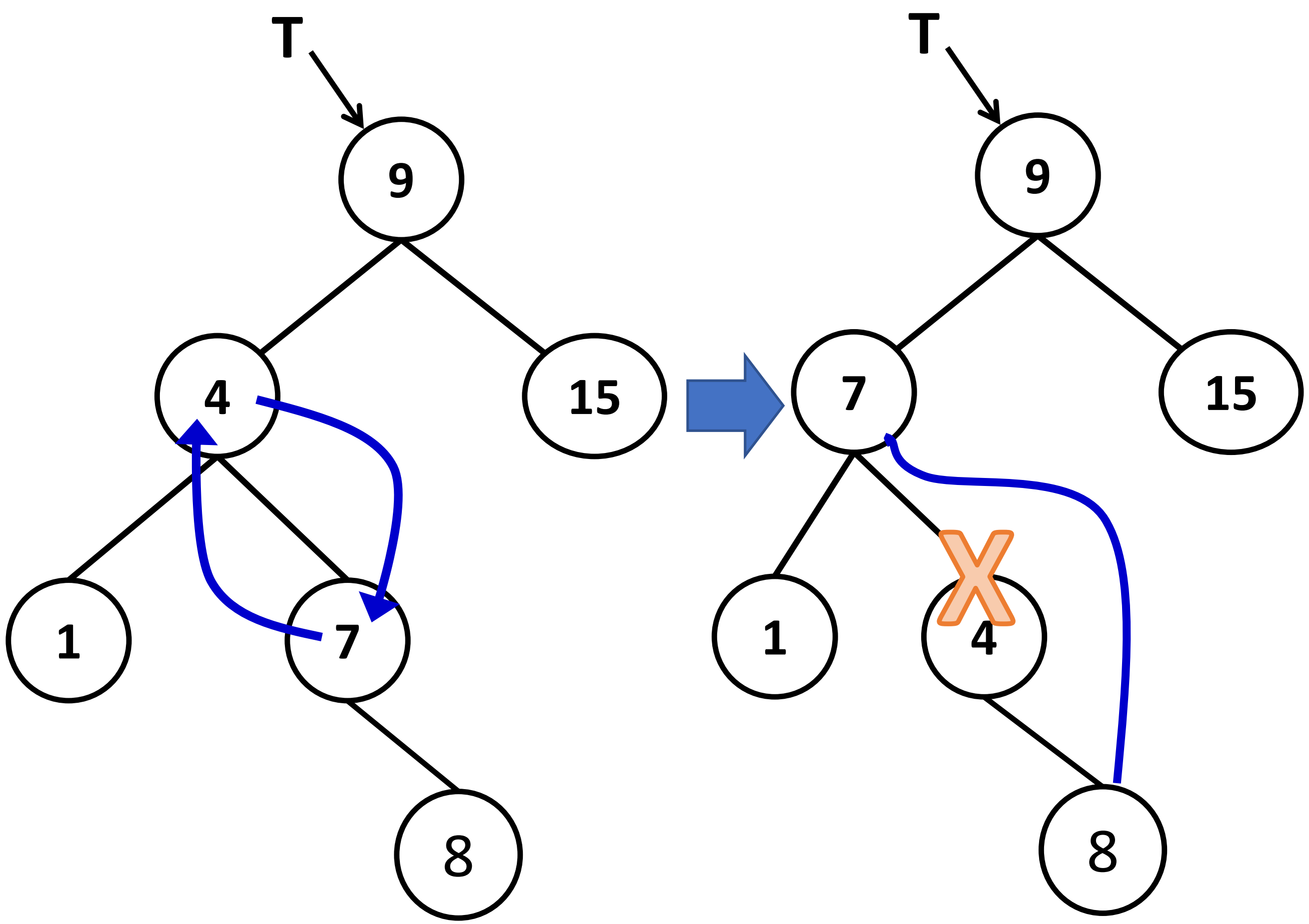
- Need to adjust the both **leaves below 4**

[Tip]:

- Switch the target with its right child,

- Delete node, now containing the target

- Example: switch 4 and 7 and delete the node containing 4



Deletion in BST (1/4)

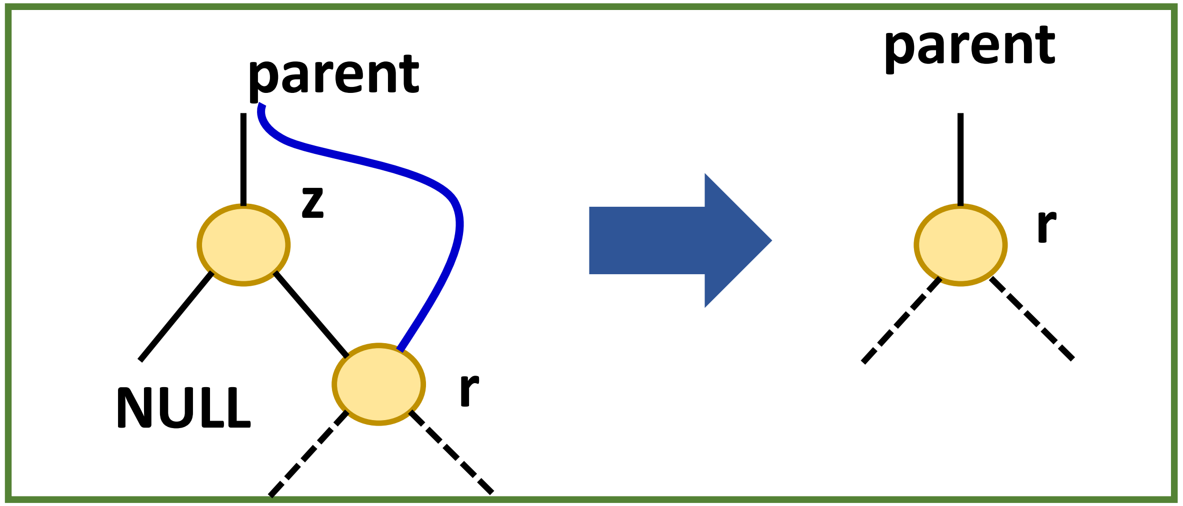
Cases

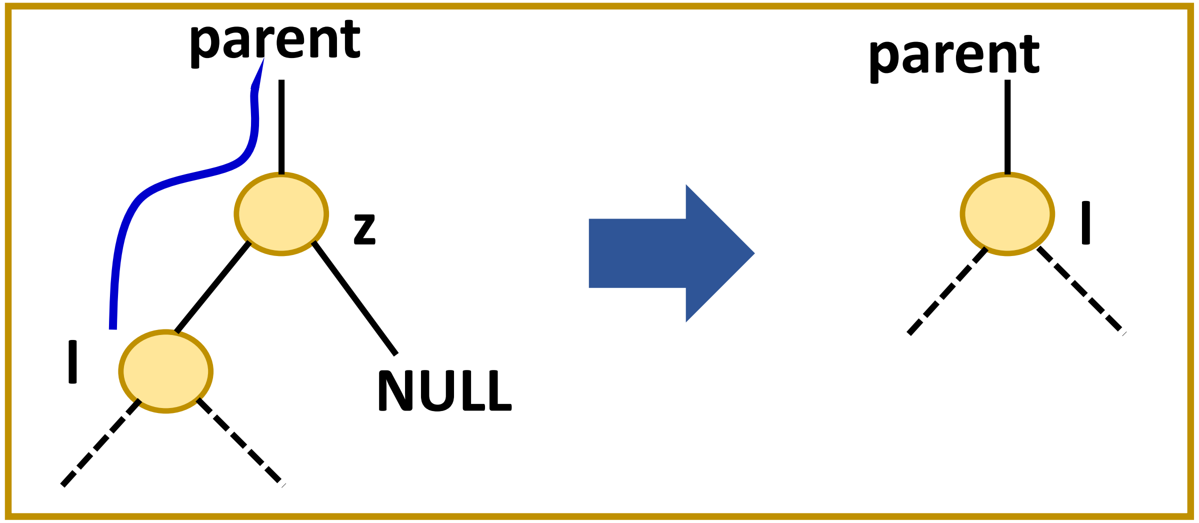
1. Deleting a leaf node
2. Deleting a node having a single child
3. Deleting a node having both left and right child
   - Two sub-cases

Case 1: Deleting a leaf node is simple – just delete it

Deletion in BST (2/4)

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

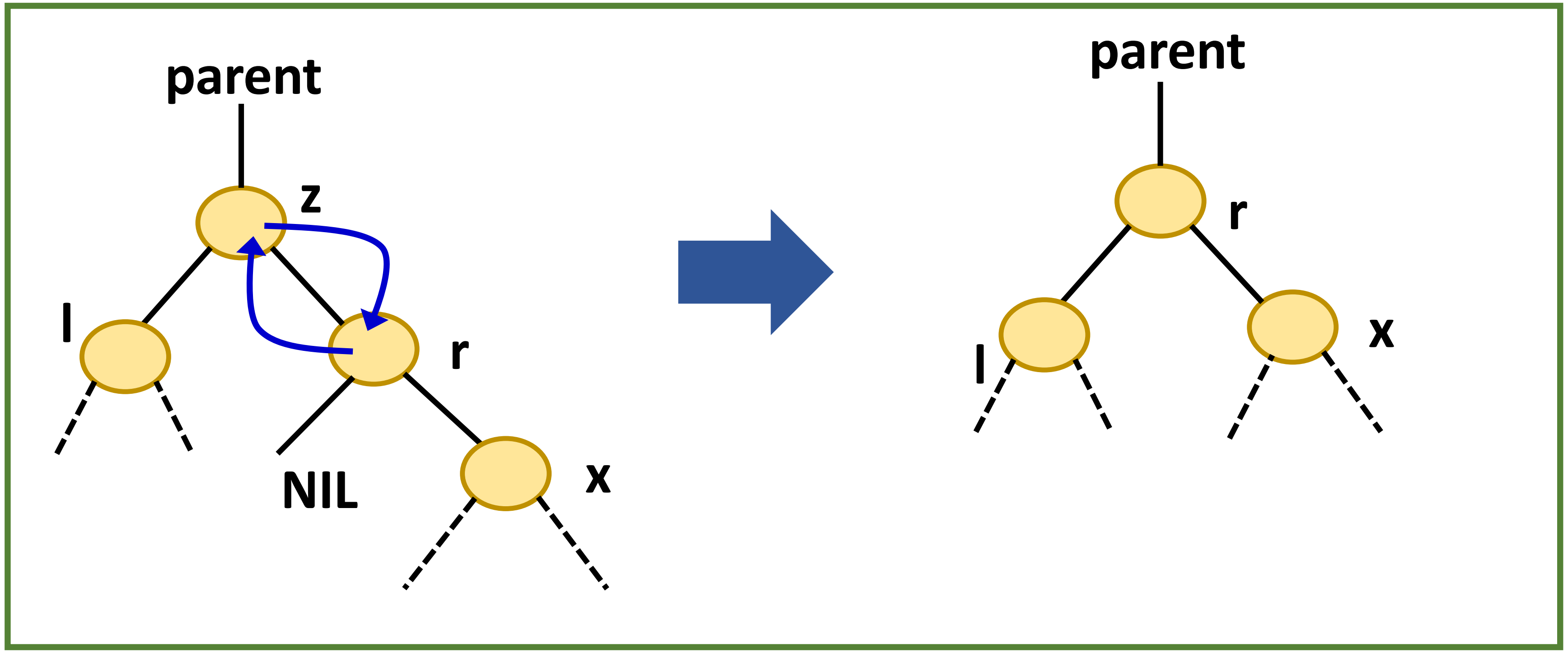




Target node for deletion: z

Deletion in BST (3/4)

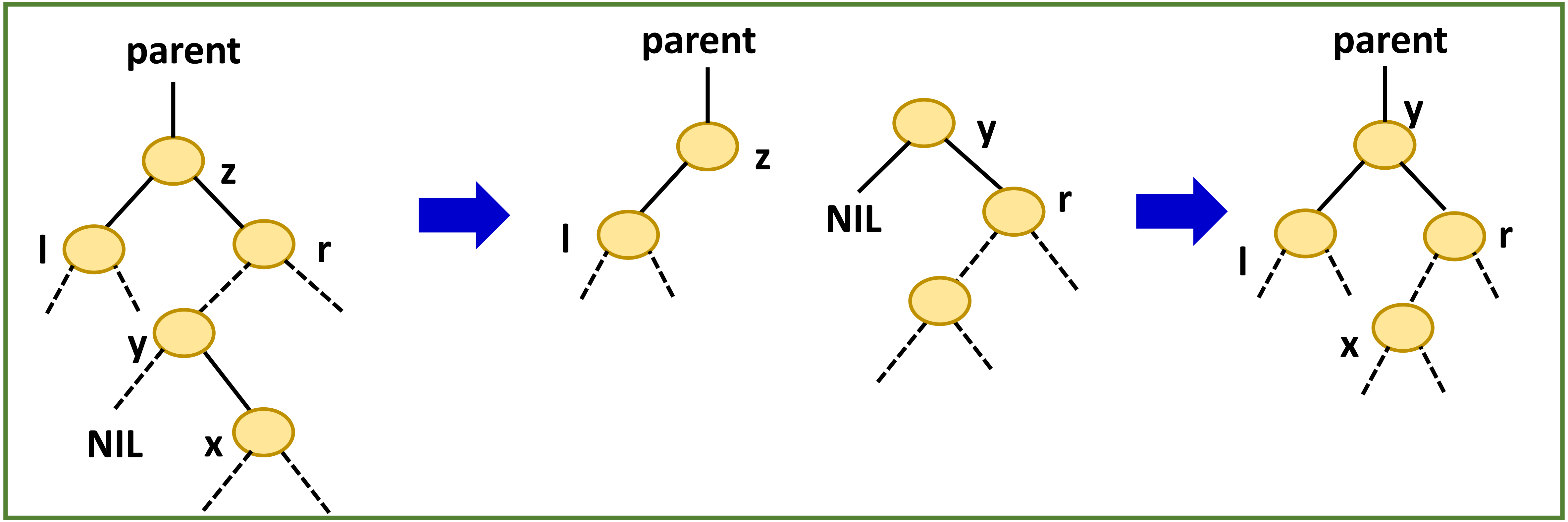
Case – III(a): Deleting a node having both left and right child present



Target node for deletion: z

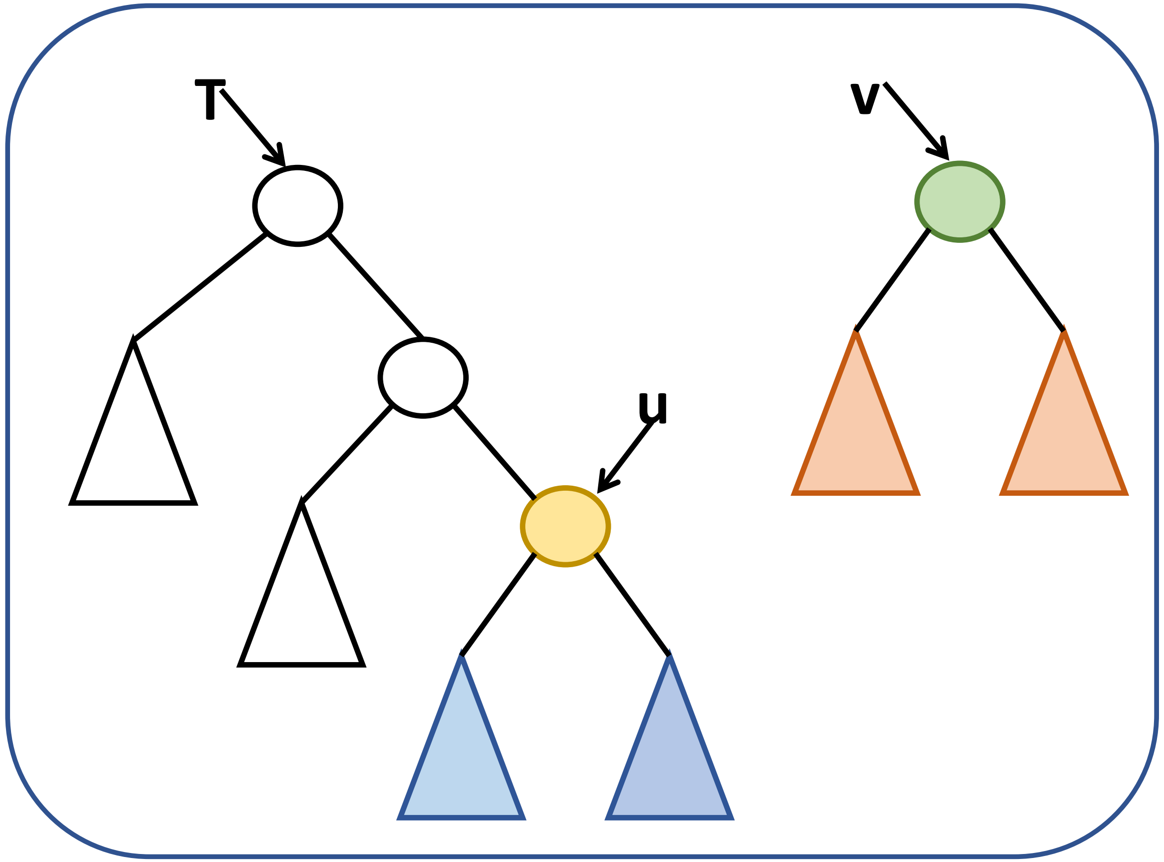
Deletion in BST (4/4)

Case – III(b): Deleting a node having both left and right child present



Transplant

Transplant (T, u, v)
{
if u.parent == null then //u is the root node
T ← v
else if u == u.parent.left
u.parentarent.left ← v
else
u.p.right ← v
if v ≠ null
v.p ← u.parent
}



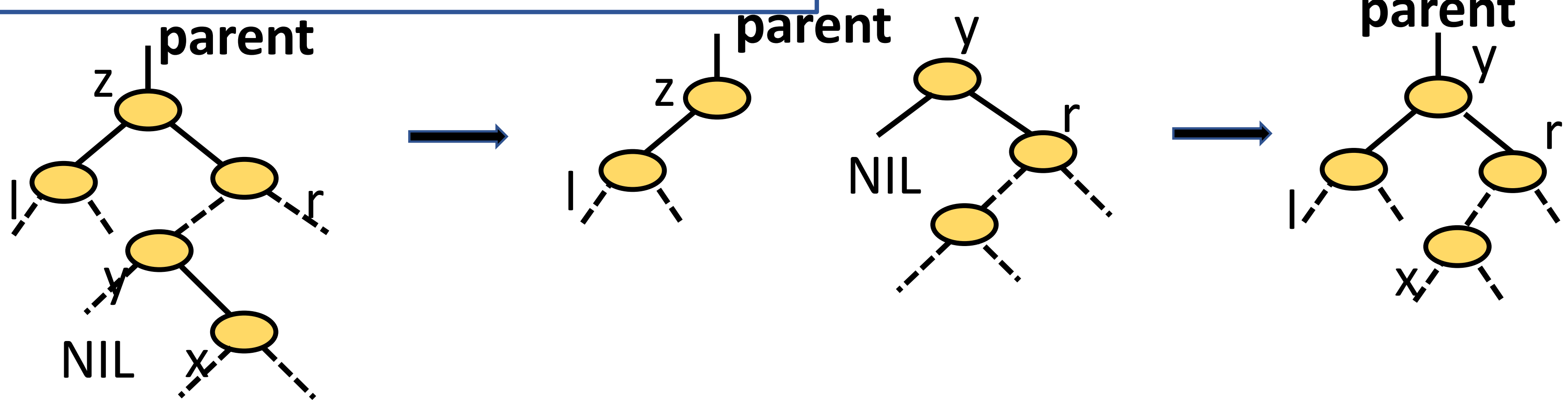
Deletion Pseudo-code (1/3)

```
delete (T, z)

if (z.left == null) and (z.right == null) then          //Case 1
        if z == z.parent.left then
            z.parent.left ← null
        else
            z.parent.right ← null
        else if (z.left == null)  then                  //Case 2
                Transplant (T,z,z.right)
        else if (z.right == null)  then
                Transplant (T,z,z.left)
        else                                            //Case 3
…
```

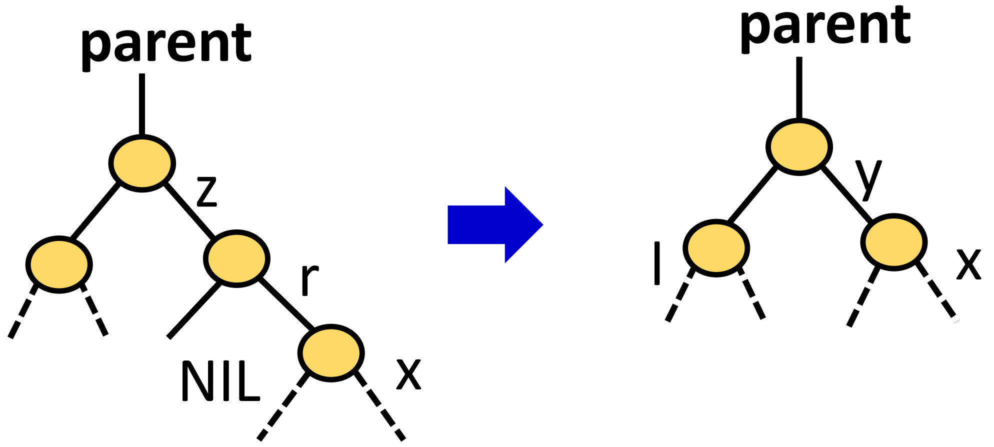
Deletion Pseudo-code (2/3)

```
…
else                                        //Case 3
   y = Tree-Minimum(z.right)
   if (y.parent <> z) then
      Transplant (T, y, y.right)            //Case 3 (b)
      y.right ← z.right
      y.left ← z.left
      z.right.parent ← y
      z.left.parent ← y
```

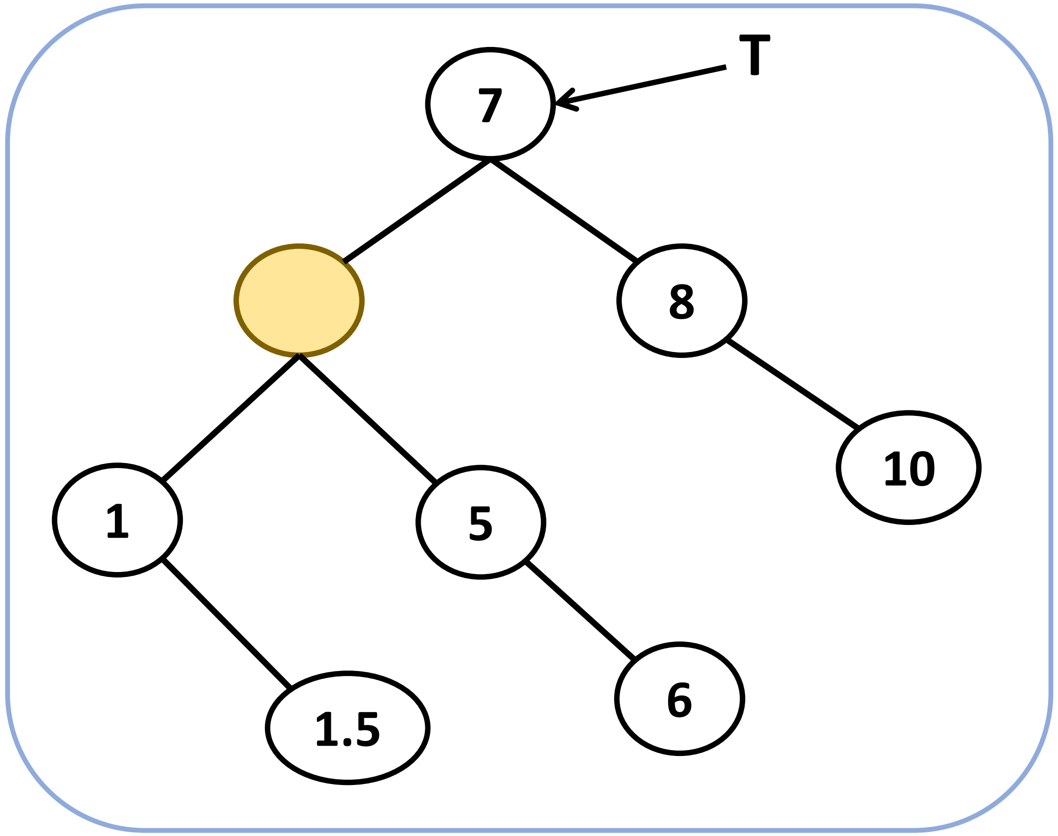


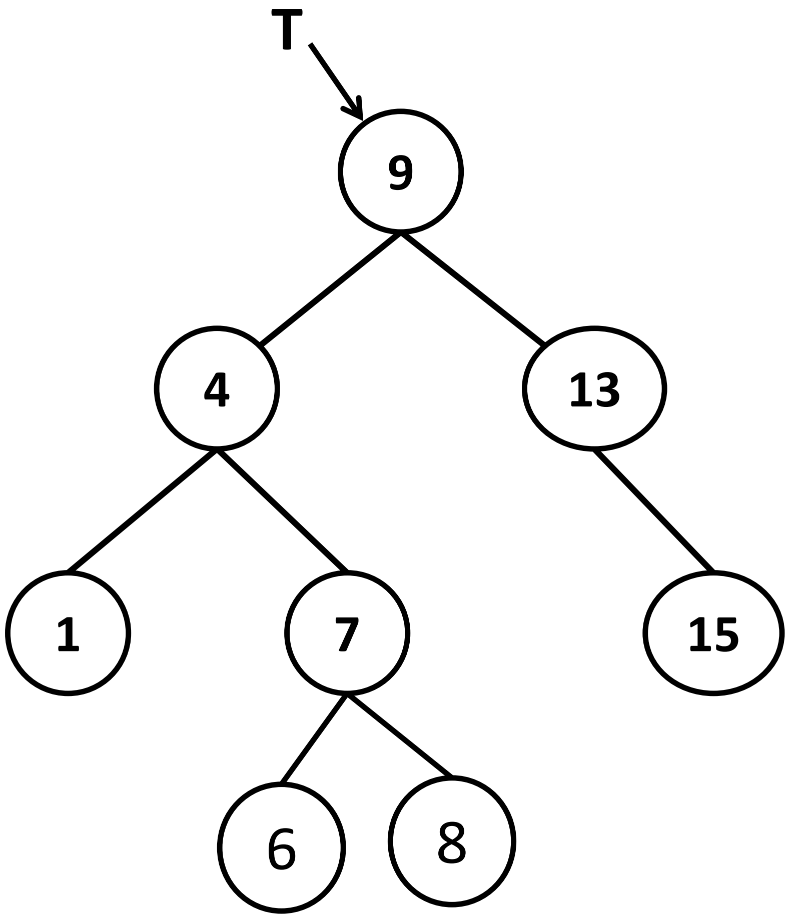
Deletion Pseudo-code (3/3)

```
…
else
        y = Tree-Minimum(z.right)       //Case 3
        if (y.p <> z) then              //Case 3 (b)
                Transplant (T,y,y.right)
                y.right ← z.right
                y.left ← z.left
                z.right.p ← y
                z.left.p ← y
else                                    //Case 3 (a)
        Transplant(T,z,y)
        y.left ← z.left
        y.left.p ← y
```



Successor/"next value" to a Node



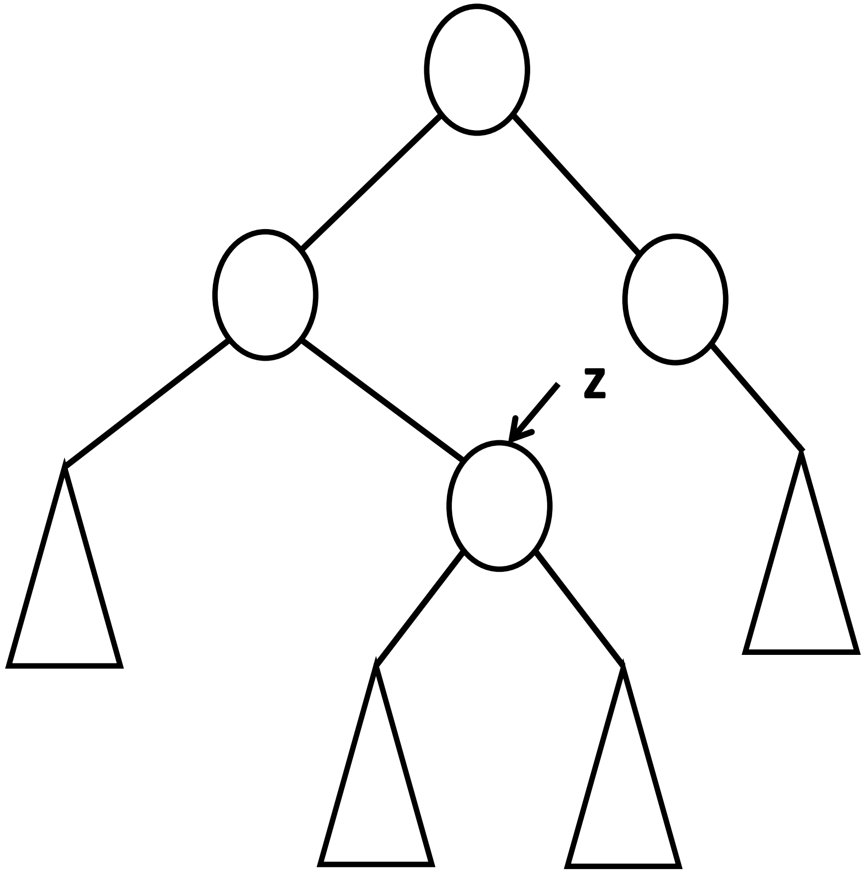


Question: How to find the successor ?

Successor of a node is the left-most leaf of its immediate right-sub-tree

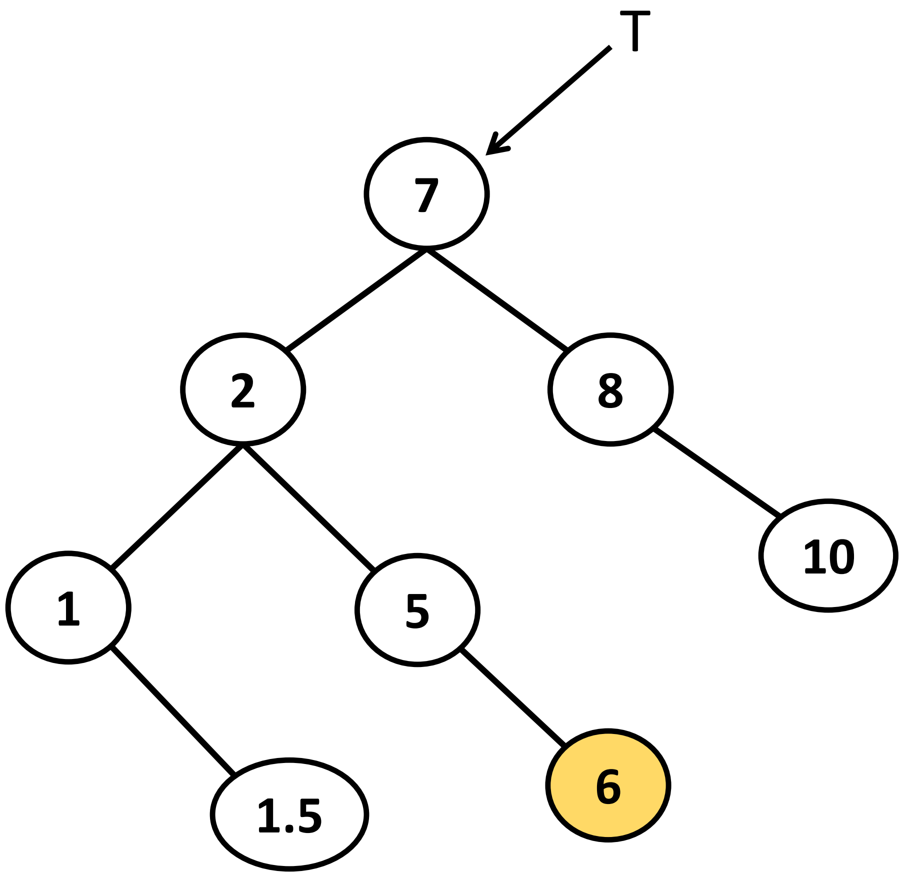
Finding Successor: Case I

1. If **z** has a right child;
2. The successor is the minimum in the subtree of **z.right** => Successor of a node is the left-most leaf of its immediate right-sub-tree



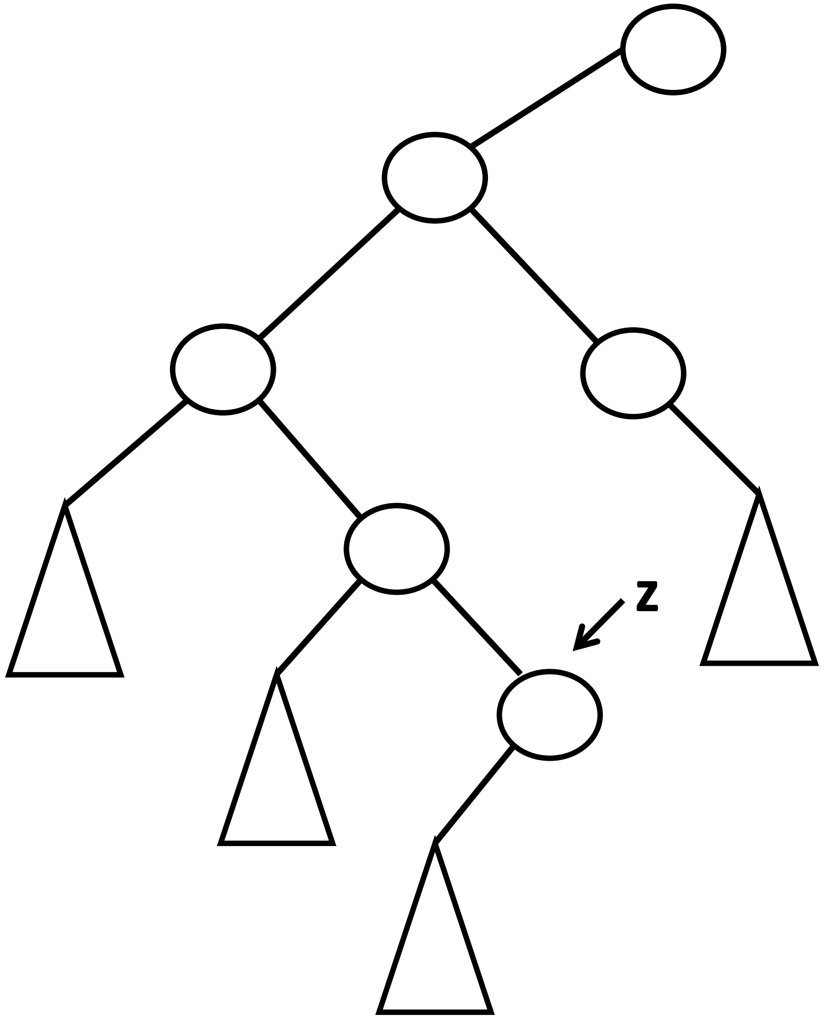
Finding Successor: Case I – Example

Successor(5, T)



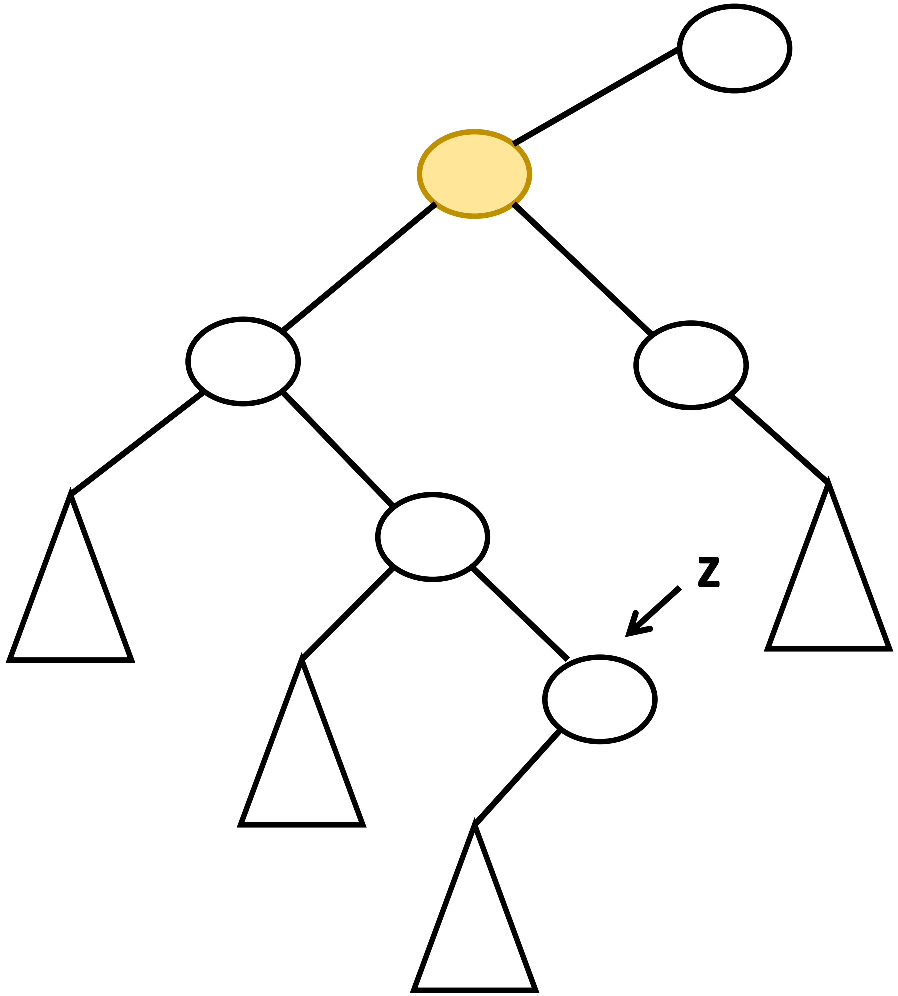
Finding Successor: Case II

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



Finding Successor: Algorithm (1/4)

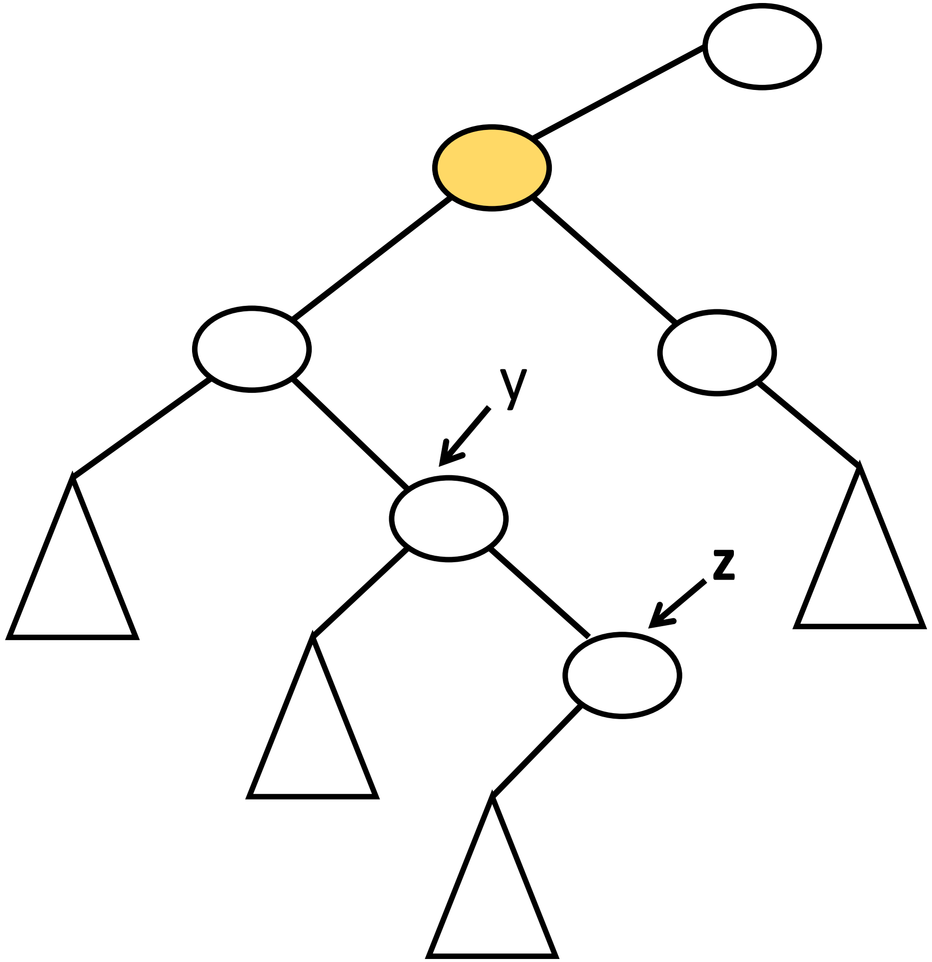
If z.right ≠ null then min(z.right)
y ← z.parent
While y≠null and z = y.right
    z ← y
    y ← y.parent
return(y)



Finding Successor: Algorithm (2/4)

Successor(z, T)

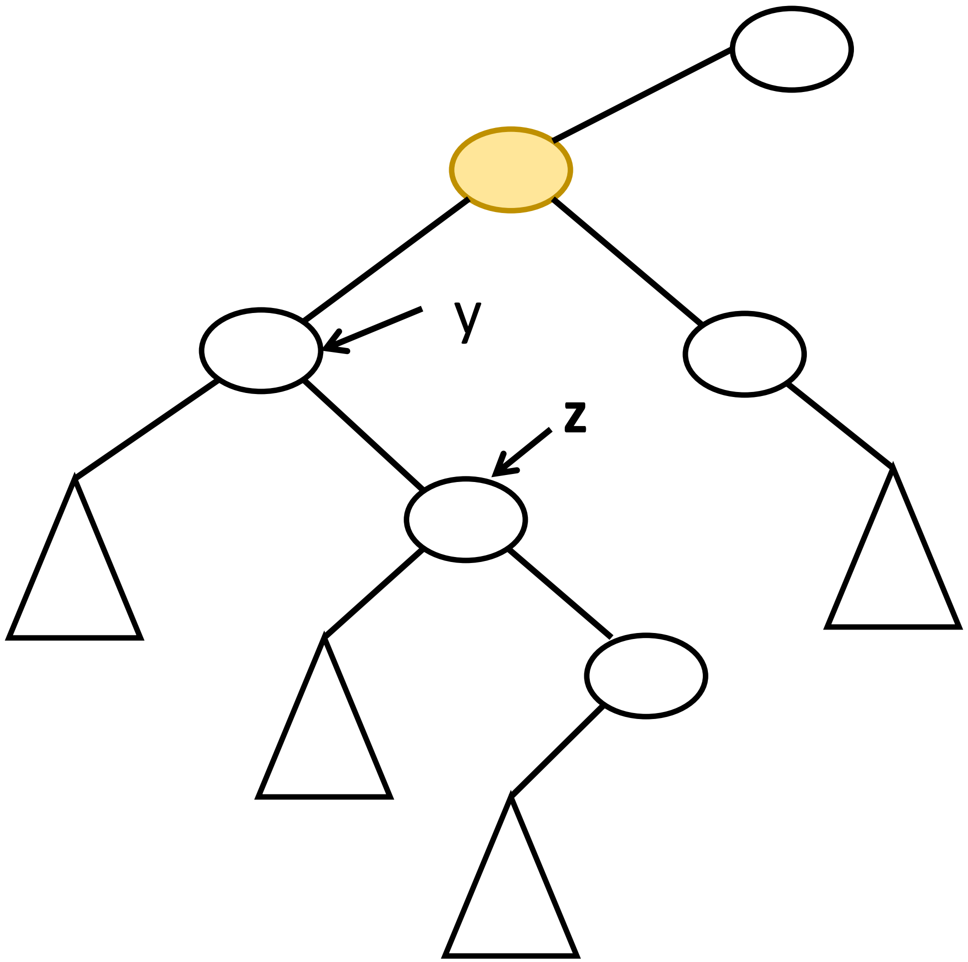
If z.right ≠ null then min(z.right)
y ← z.parent
While y≠null and z = y.right
    z ← y
    y ← y.parent
return(y)



Finding Successor: Algorithm (3/4)

Successor(z, T)

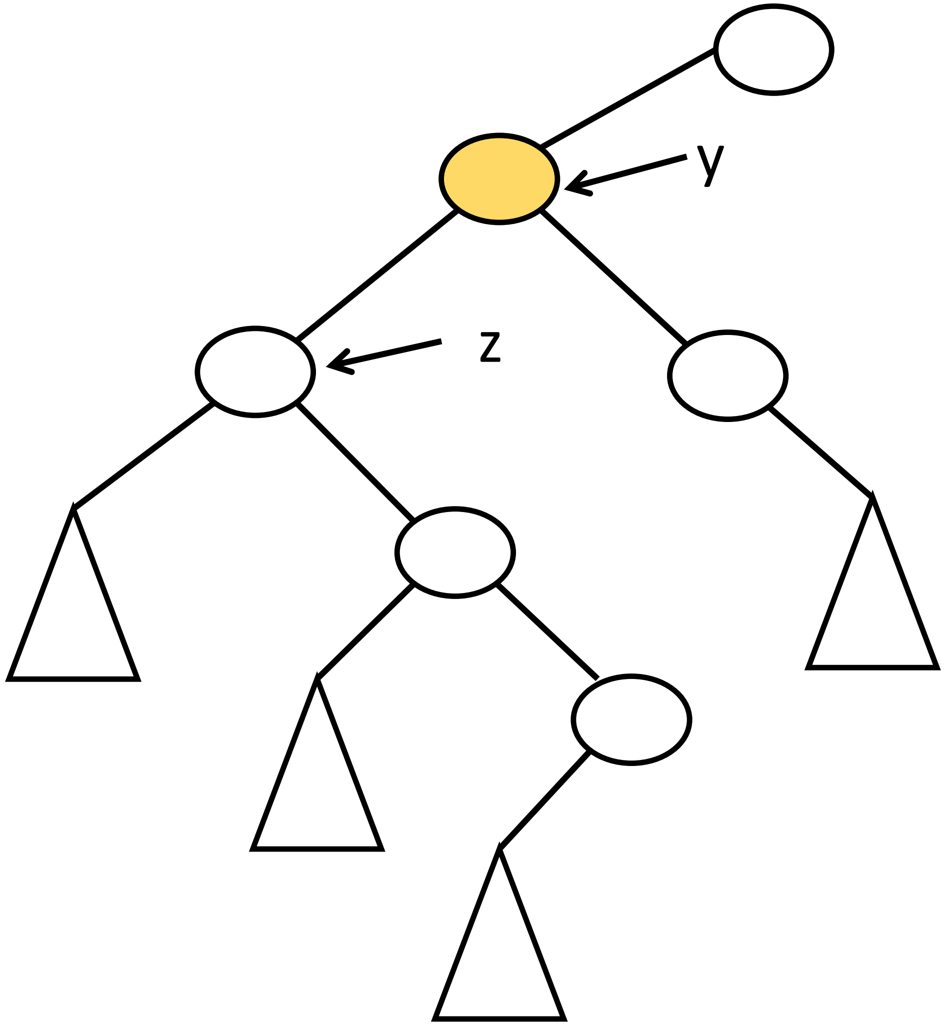
If z.right ≠ null then min(z.right)
y ← z.parent
While y≠null and z = y.right
    z ← y
    y ← y.parent
return(y)



Finding Successor: Algorithm (3/4)

Successor(z, T)

If z.right ≠ null then min(z.right)
y ← z.parent
While y≠null and z = y.right
z ← 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)**

Thank you!