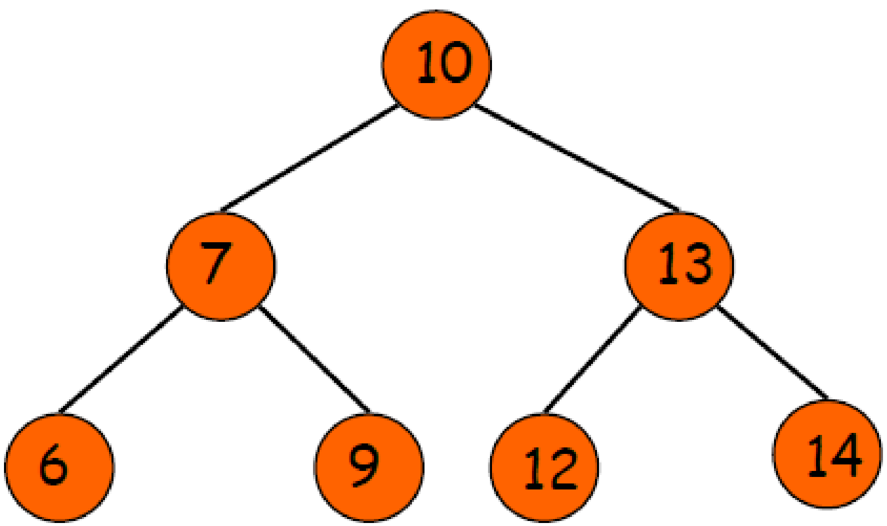
# Lecture 11: Binary Search Tree

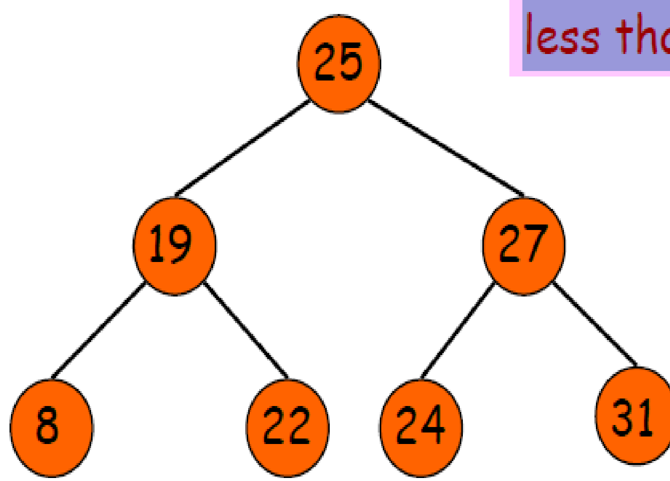
# Binary Search Tree

- It’s a binary tree !
- For each node in a BST
  - left subtree is smaller than it
  - right subtree is greater than it




- **Is this a BST ?**

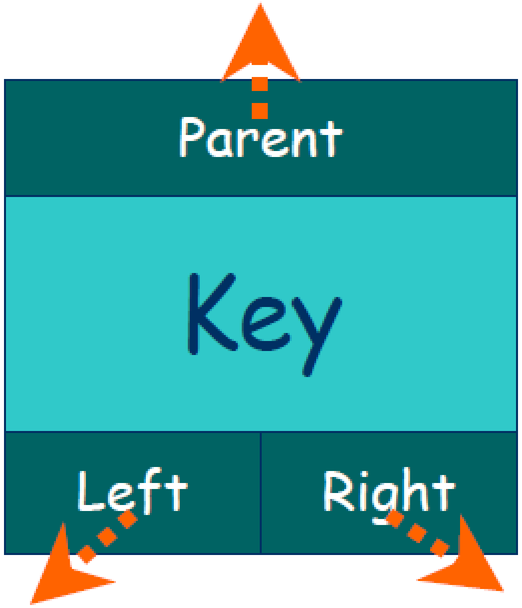
NO! Because 24 is less than 25



# Applications of Binary Search Tree

- Used in *many* search applications where data is constantly entering/leaving
- Used to represent arithmetic expressions
- Used in Unix kernels for managing a set of virtual memory areas (VMAs). Each VMA represents a section of memory in a Unix process. VMAs vary in size from 4KB to 1GB.

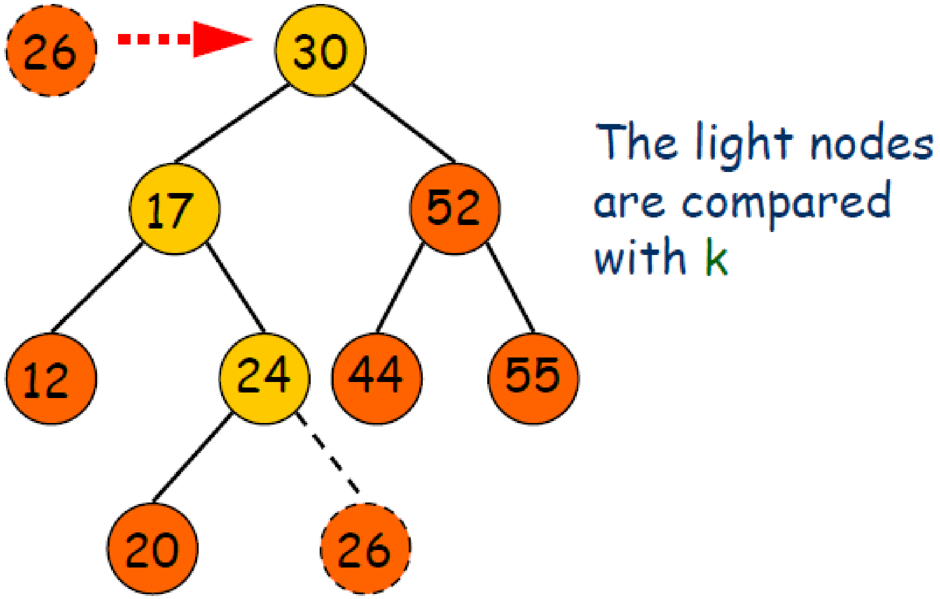
# Node Structure & Operations




- 3 common operations are:
  - INSERT
  - QUERY
  - DELETE

# Operation - Insert

- Insert(T,z)
  - Insert a node with KEY=z into BST T
  - Time complexity: O(h)




  - **Step1:** if the tree is empty, then Root(T)=z
  - **Step2:** Search for z in BST T, until we meet a null node
  - **Step3:** Insert z

# Insert - Algorithm

NEWNODE is a pointer variable to hold the address of the newly created node. DATA is the information to be pushed.

1. Input the DATA to be pushed and ROOT node of tree.
2. NEWNODE = Create a New Node.
3. If (ROOT == NULL)
   (*a*) ROOT=NEW NODE
4. Else If (DATA < ROOT → Info)
   (*a*) ROOT = ROOT → Lchild
   (*b*) GoTo Step 4
5. Else If (DATA > ROOT → Info)
   (*a*) ROOT = ROOT → Rchild
   (*b*) GoTo Step 4
6. If (DATA < ROOT → Info)
   (*a*) ROOT → LChild = NEWNODE
7. Else If (DATA > ROOT → Info)
   (*a*) ROOT → RChild = NEWNODE
8. Else
   (*a*) Display ("DUPLICATE NODE")
   (*b*) EXIT
9. NEW NODE → Info = DATA
10. NEW NODE → LChild = NULL
11. NEW NODE → RChild = NULL
12. EXIT

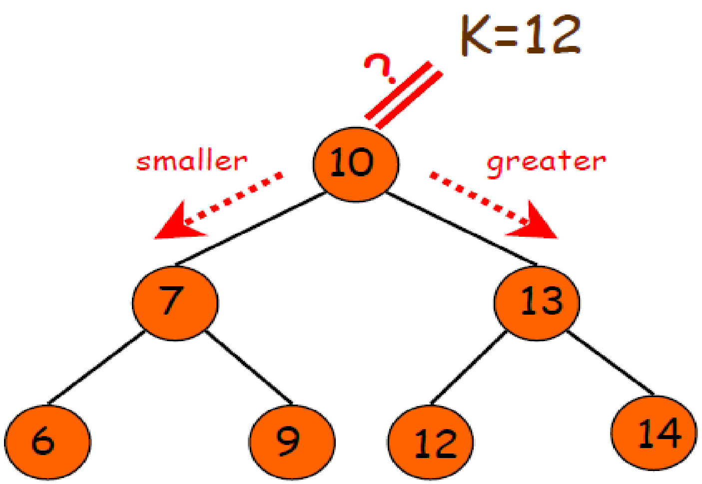
# Insert()

```
struct node* insert(struct node* node, int data) {
                                   // 1. If the tree is empty, return a new, single node
  if (node == NULL) {
    return(newNode(data));
  }
  else {
                                   // 2. Otherwise, recur down the tree
      if (data <= node->data)
        node->left = insert(node->left, data);
     else
     node->right = insert(node->right, data);
  return(node);                    // return the (unchanged) node pointer
  }
}
```

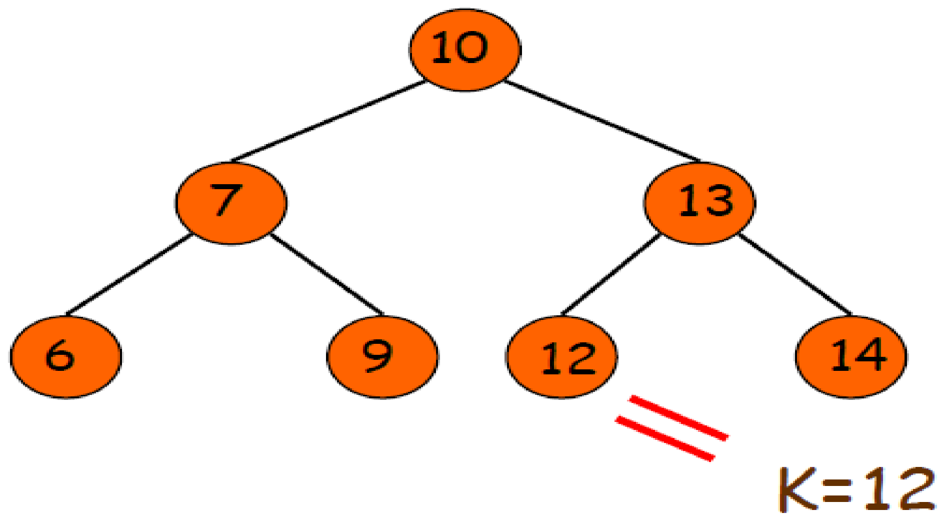
# Operation - Query

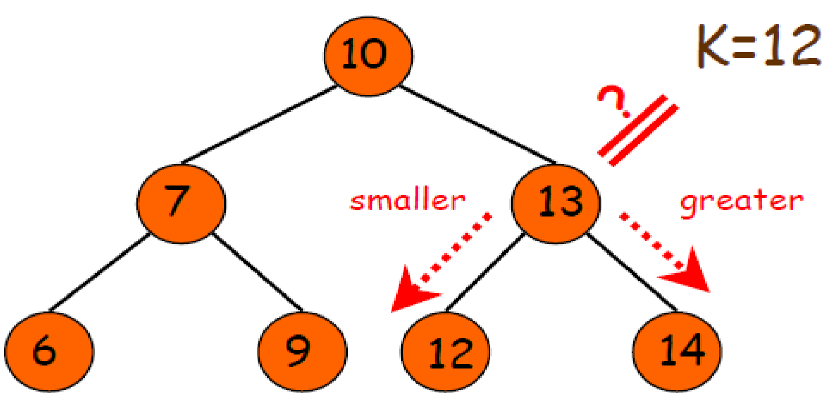
- The QUERY operation can be further split into:
  - Search
  - Max/Min
  - Successor/Predecessor

# Operation - Search




- Search(T,k)
  - search the BST T for a value k




- Search operation takes time O(h), where h is the height of a BST

# Search - Algorithm

1. Input the DATA to be searched and assign the address of the root node to ROOT.
2. If (DATA == ROOT → Info)

    (*a*) Display “The DATA exist in the tree”

    (*b*) GoTo Step 6
3. If (ROOT == NULL)

    (*a*) Display “The DATA does not exist”

    (*b*) GoTo Step 6
4. If(DATA > ROOT→Info)

    (*a*) ROOT = ROOT→RChild

    (*b*) GoTo Step 2
5. If(DATA < ROOT→Info)

    (*a*) ROOT = ROOT→Lchild

    (*b*) GoTo Step 2
6. Exit

# Search()

```
static int lookup(struct node* node, int target) {
                                                    // 1. Base case == empty tree
                                                    // in that case, target is not found so return false
 if (node == NULL) {
  return(false);
 }
 else {
                                                    // 2. see if found here
  if (target == node->data)
         return(true);
    else {
                                                    // 3. otherwise recur down the correct subtree
     if (target < node->data)
       return(lookup(node->left, target));
   else return(lookup(node->right, target));
  }
 }
}
```

# Operation –Min/Max

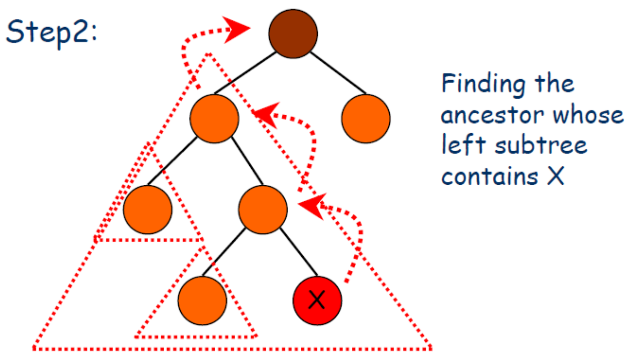
- For Min, we simply follow the left pointer until we find a null node
- Why? Because if it’s not the minimum node, then the real min node must reside at some node’s right subtree
- By the property of BST, it’s a contradiction
- Similar for Max
- Time complexity: O(h)

# Operation –Min/Max

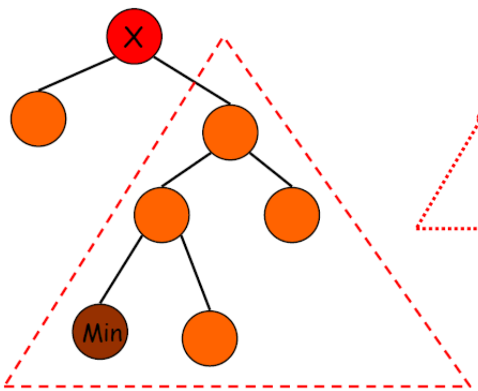
```
findMin( Node* t )
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left);
}

findMax( Node* t )
{
    if( t != NULL )
        while( t->right != NULL )
            t = t->right;
    return t; }
```

# Operation Predecessor/Successor

- Successor(x)
  - If we sort all elements in a BST to a sequence,
  - return the element just after x
    - Time complexity: O(h)

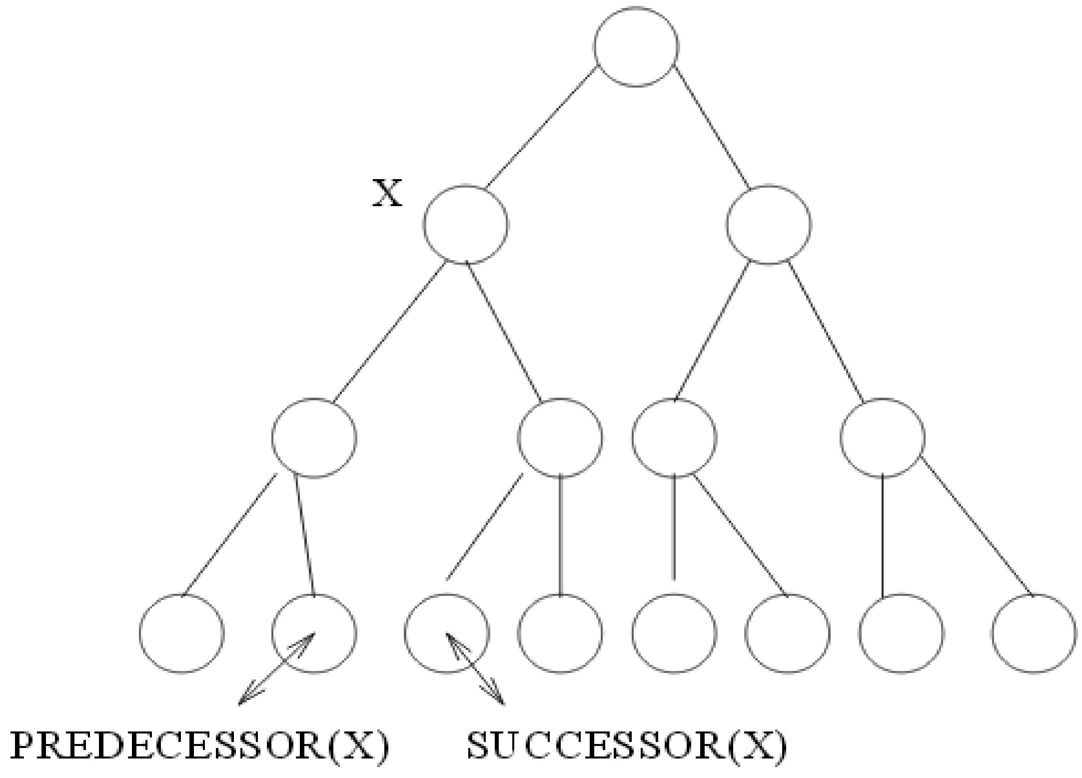



**Step 1**



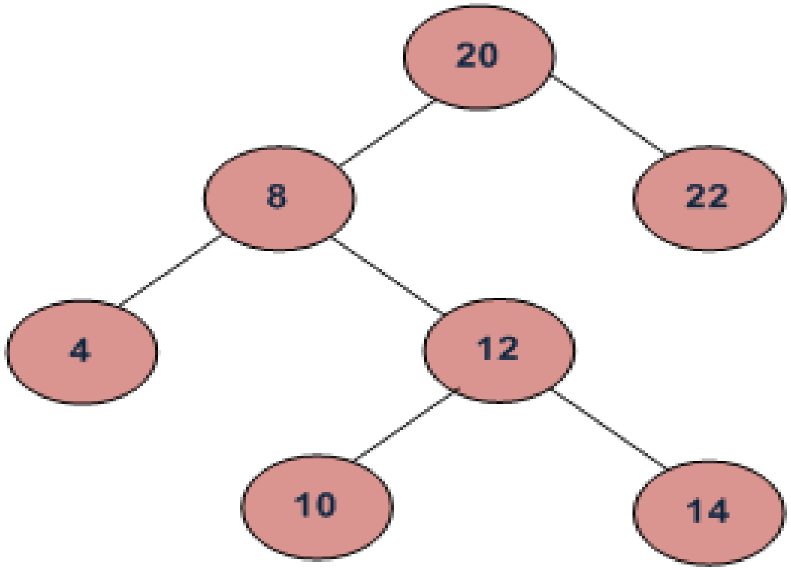

- Find Successor
  - if Right(x) exists,                    // **Step 1**
  - then return **Min( Right(x)** ) ;
  - else                    // **Step 2**
  - Find the **first ancestor** of x whose left subtree contains x ;

# Operation Predecessor/Successor




If $X$ has two children, its predecessor is the maximum value in its left subtree and its successor the minimum value in its right subtree.

# Operation Predecessor/Successor

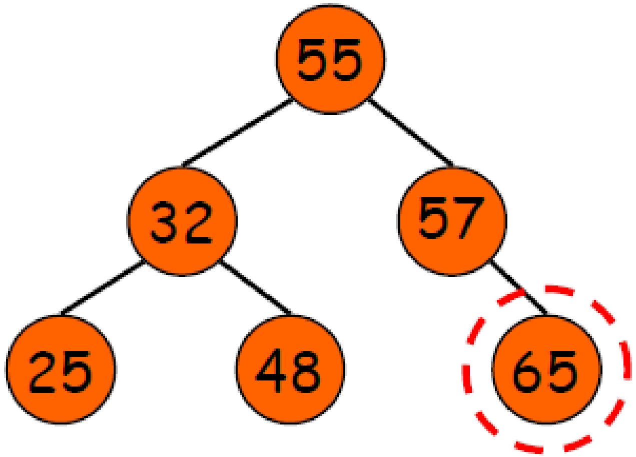


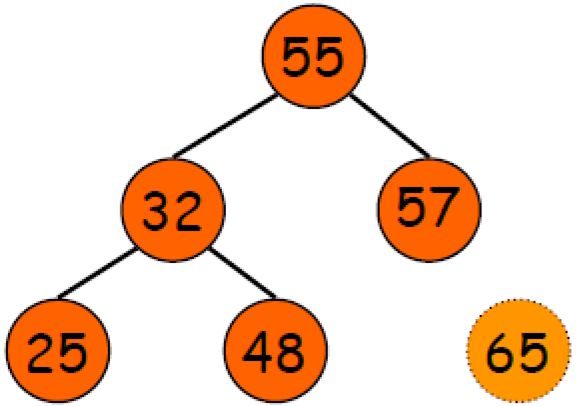

In the above diagram, inorder successor of **8** is **10**, inorder successor of **10** is **12** and inorder successor of **14** is **20**.

# Operation - Delete

- Delete (T,z)
  - Delete a node with key=z from BST T
  - Time complexity: O(h)

- **Case 1: z has no child**

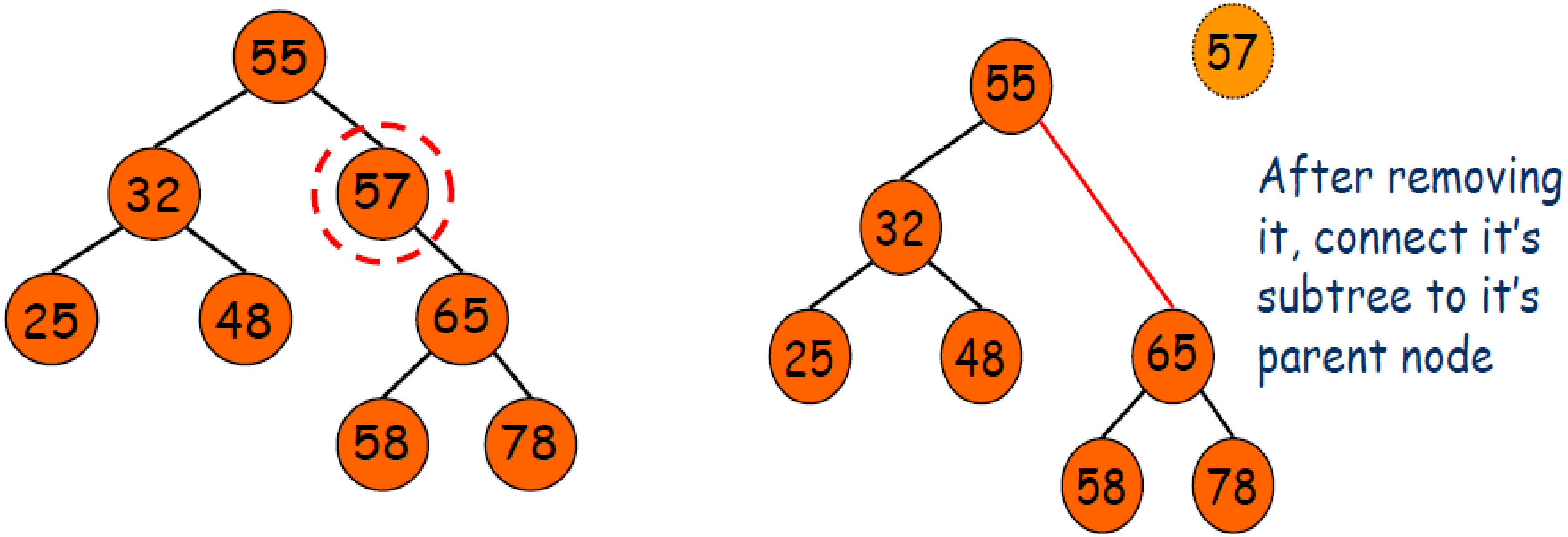



We can simply remove it from the tree

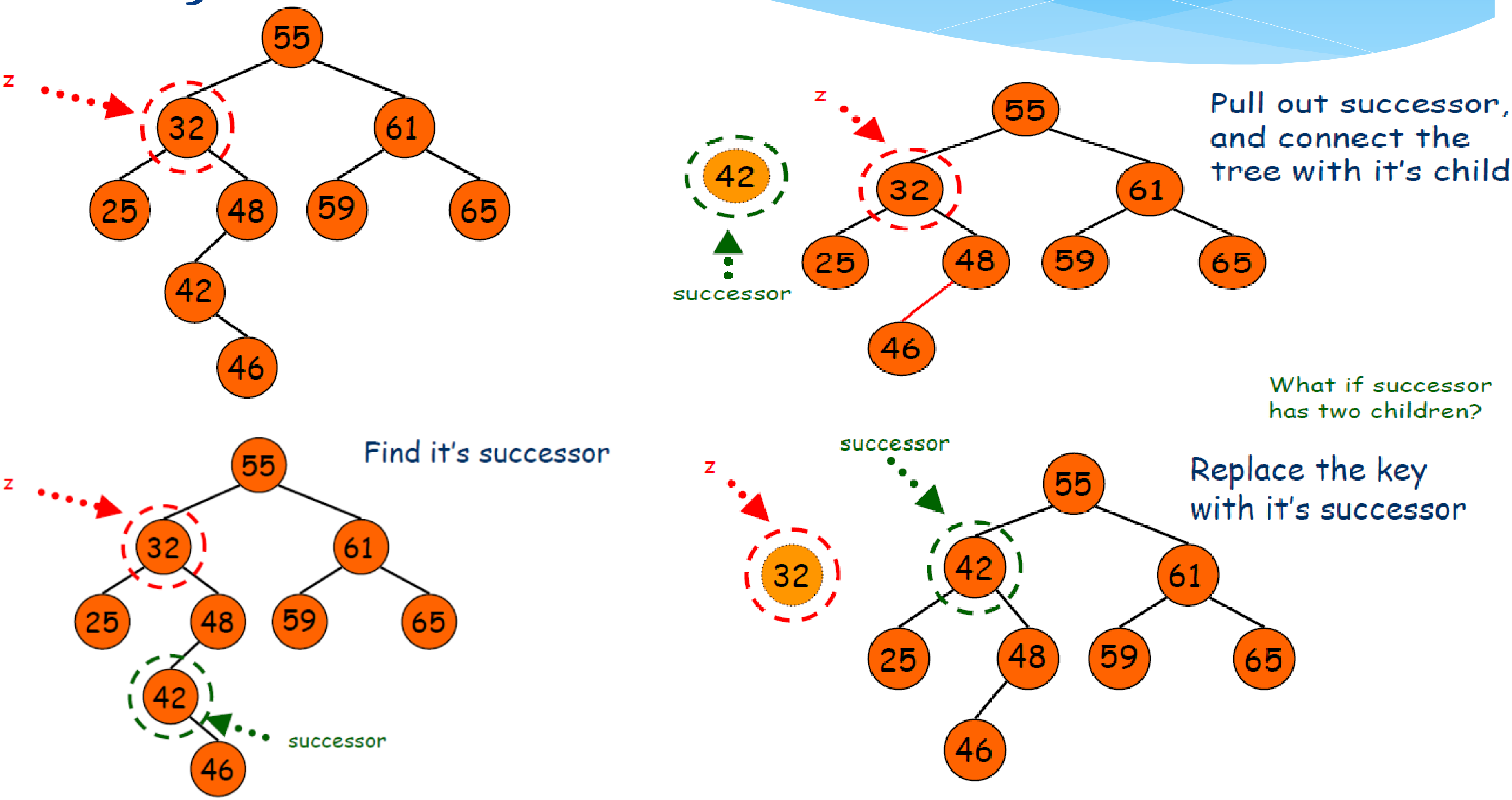
# Operation - Delete

- **Case 2: z has one child**



# Operation - Delete

➢ **Case 3: z has two child**



# Operation - Delete