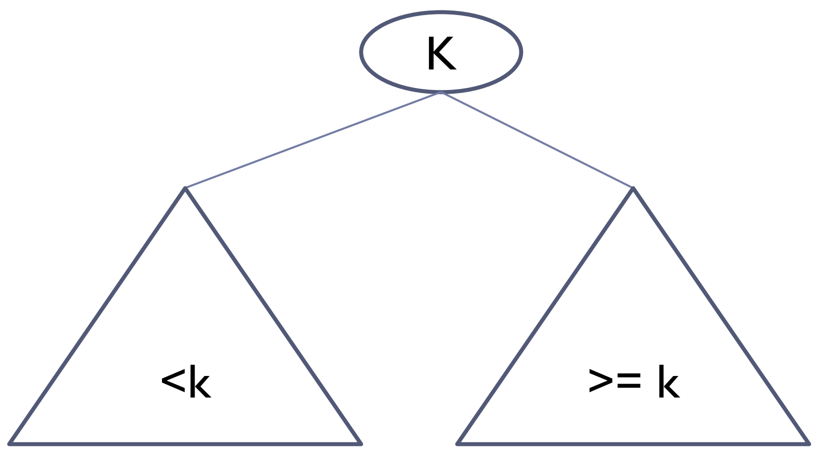
Binary Search Tree

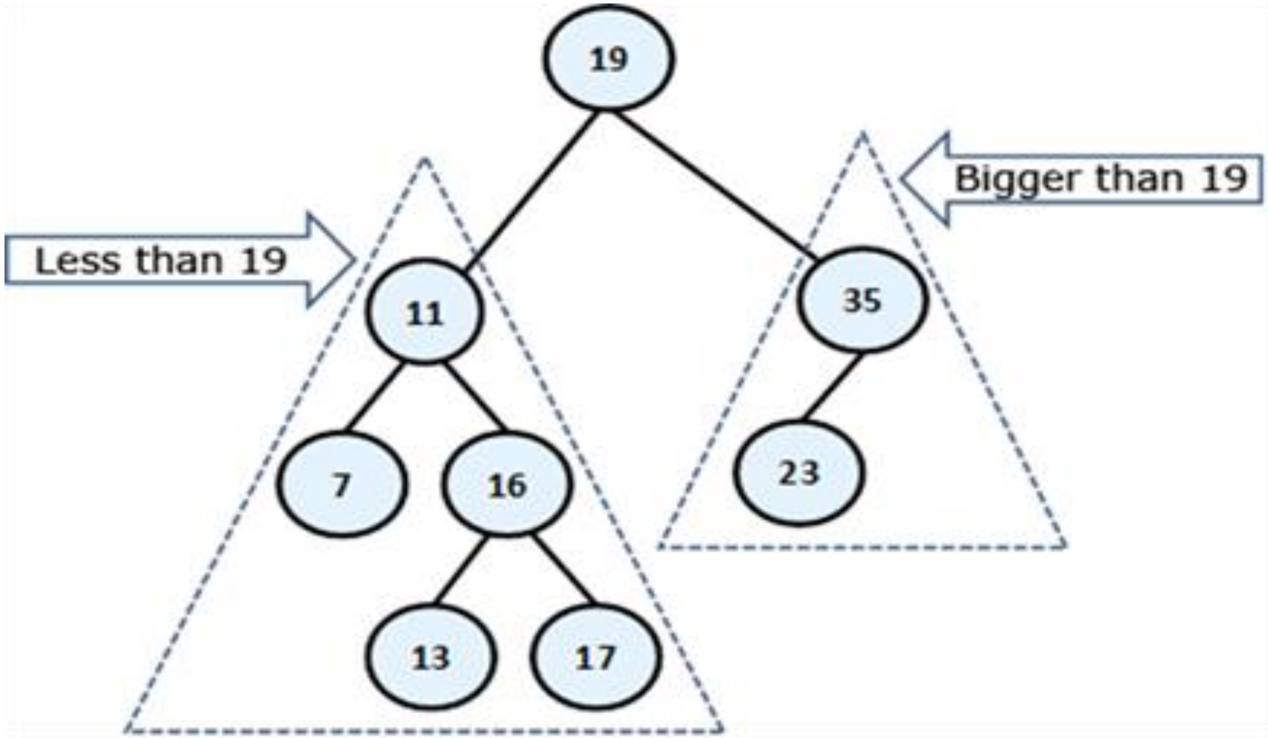
Binary Search Tree

- BST is a binary tree with following properties:
  - All items in the left subtree are less than the root
  - All items in the right subtree are greater than or equal the root
  - Each subtree is itself a binary search tree

i.e. For a node with key *k* , every key in the left subtree is less than *k* and every key in the right subtree is greater than *k*.

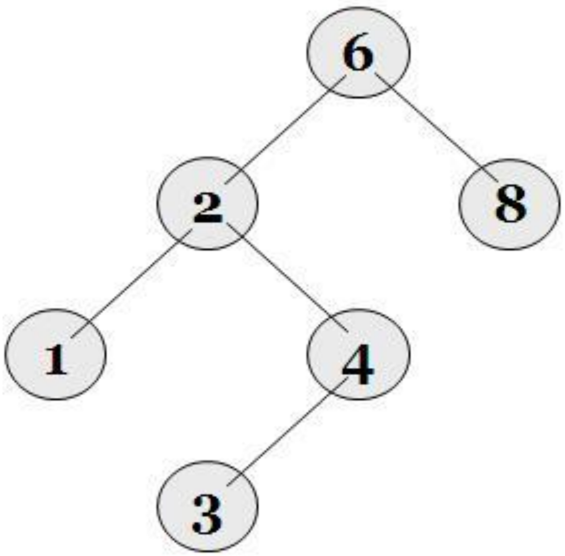


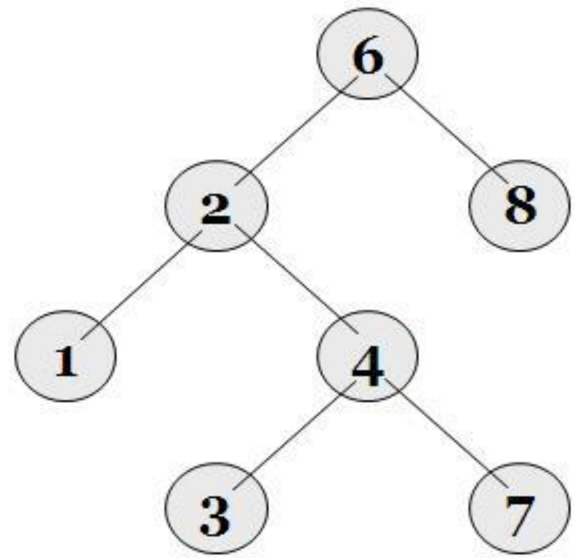
Example of BST



BST vs Binary Tree

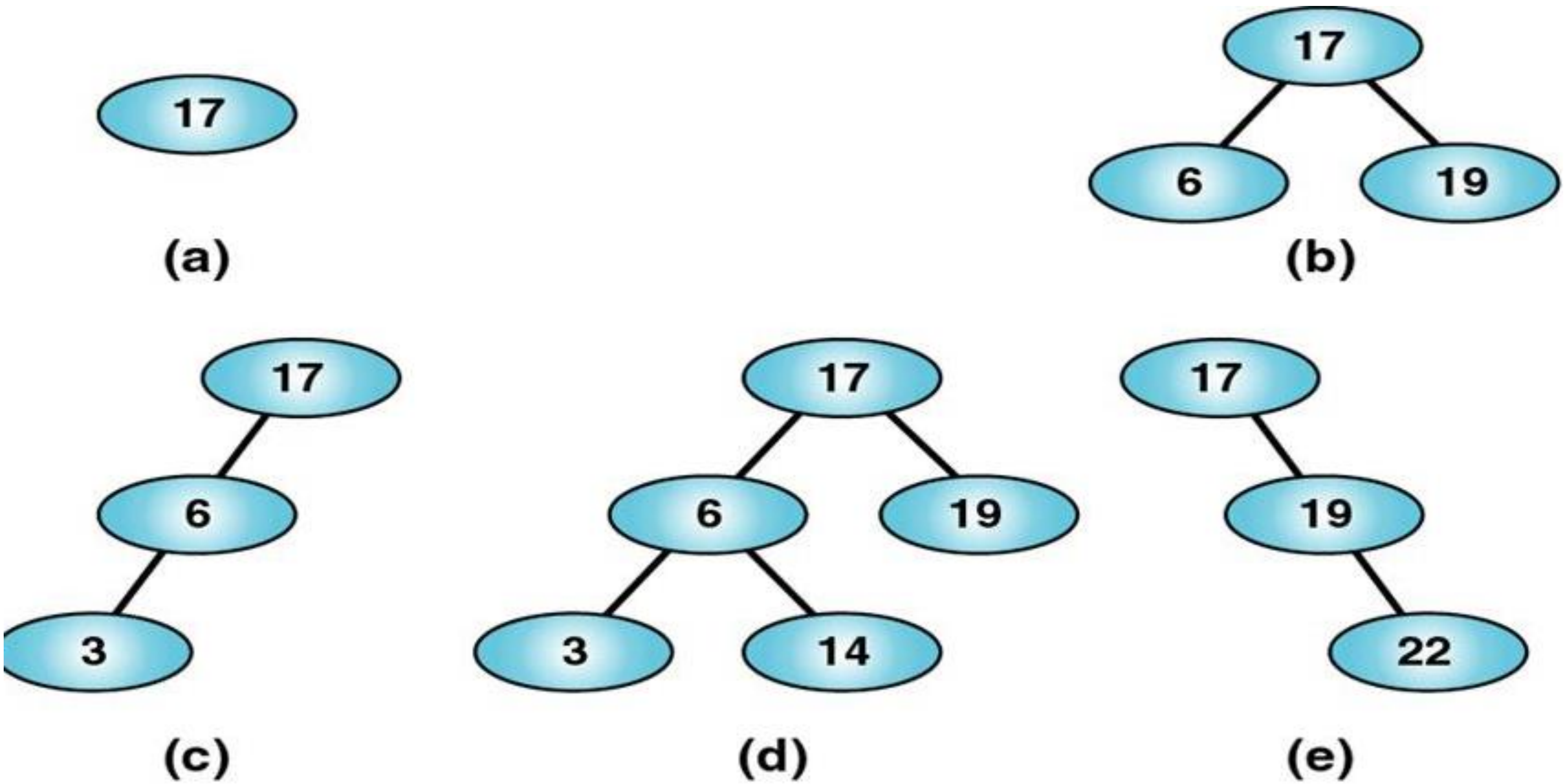
A *binary search tree*



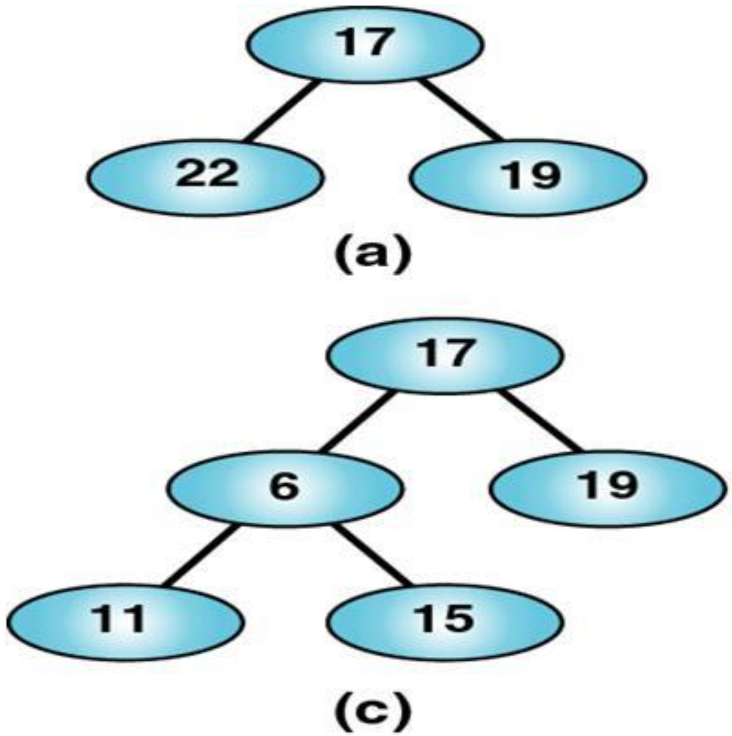


Not a *binary search tree*,
but a *binary tree*

Valid BST



Invalid BST



Here we see examples of binary trees that are *not* binary search trees ... why?

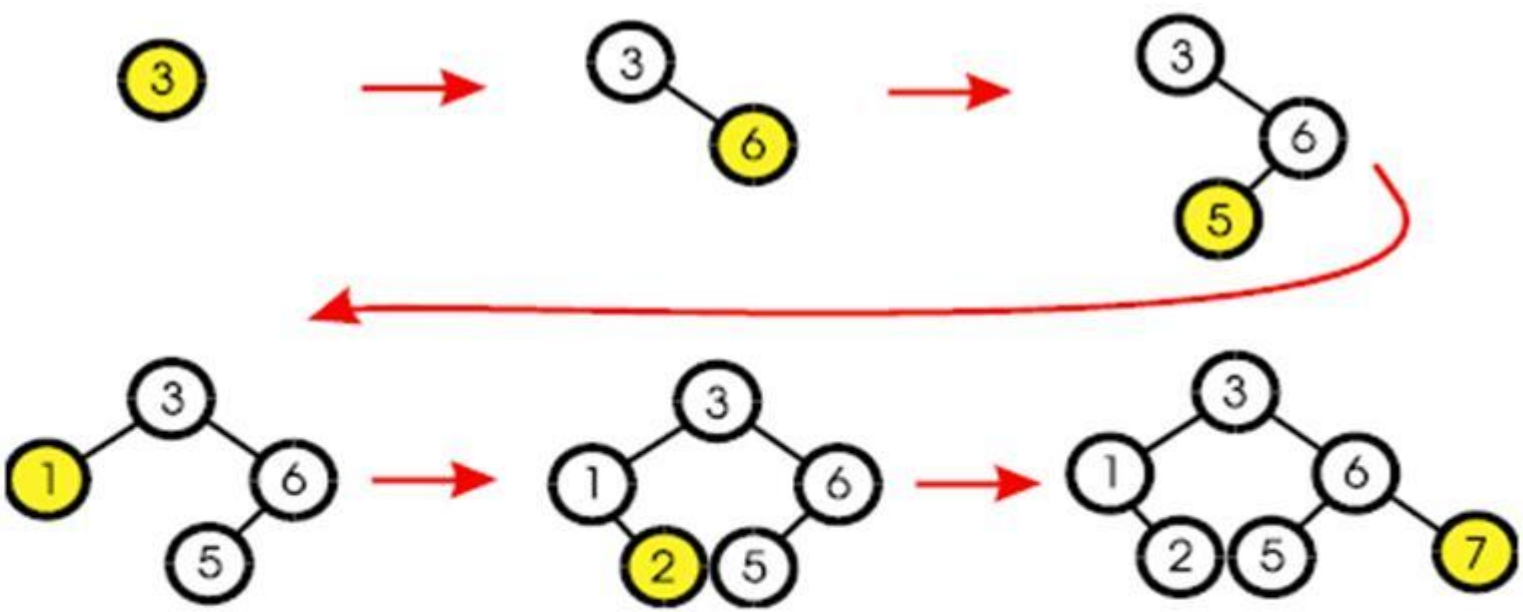
a) 22 > 17
b) 11 < 17
c) 11 > 6
d) 22 > 17

Operations on Binary Search Trees

- Binary trees offer short paths from root. A node has up to two children. Data is organized by value:
  - Insertion
  - Search
  - Traversal
  - Deletion
  - Find Minimum: Find the item that has the minimum value in the tree
  - Find Maximum: Find the item that has the maximum value in the tree
  - Print: Print the values of all items in the tree, using a traversal strategy that is appropriate for the application
  - Successor
  - Predecessor

Inserting an item in BST

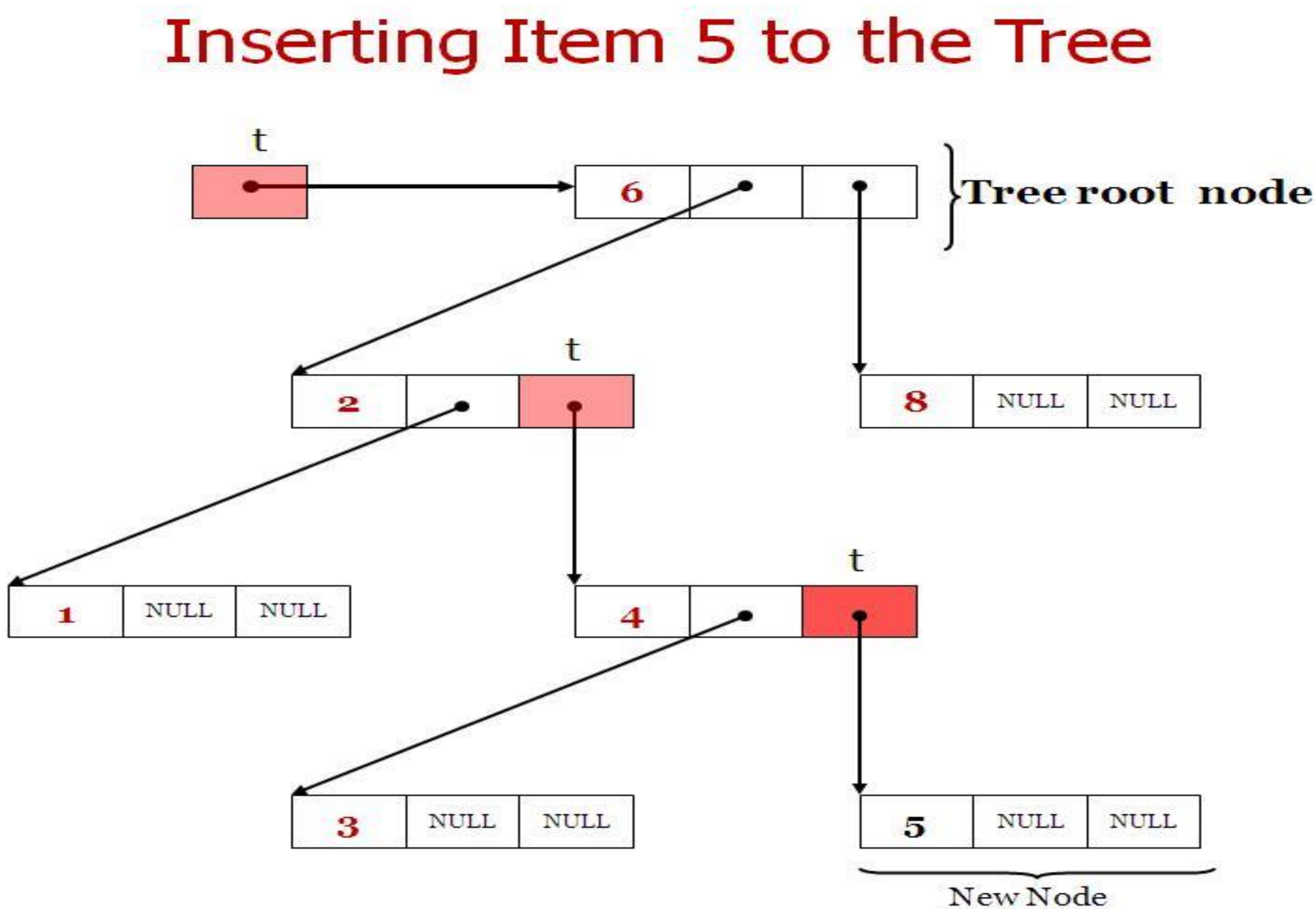
- The first value inserted goes at the root.
- Every node inserted becomes a leaf. □
- Insert left or right depending upon value.



Insertion

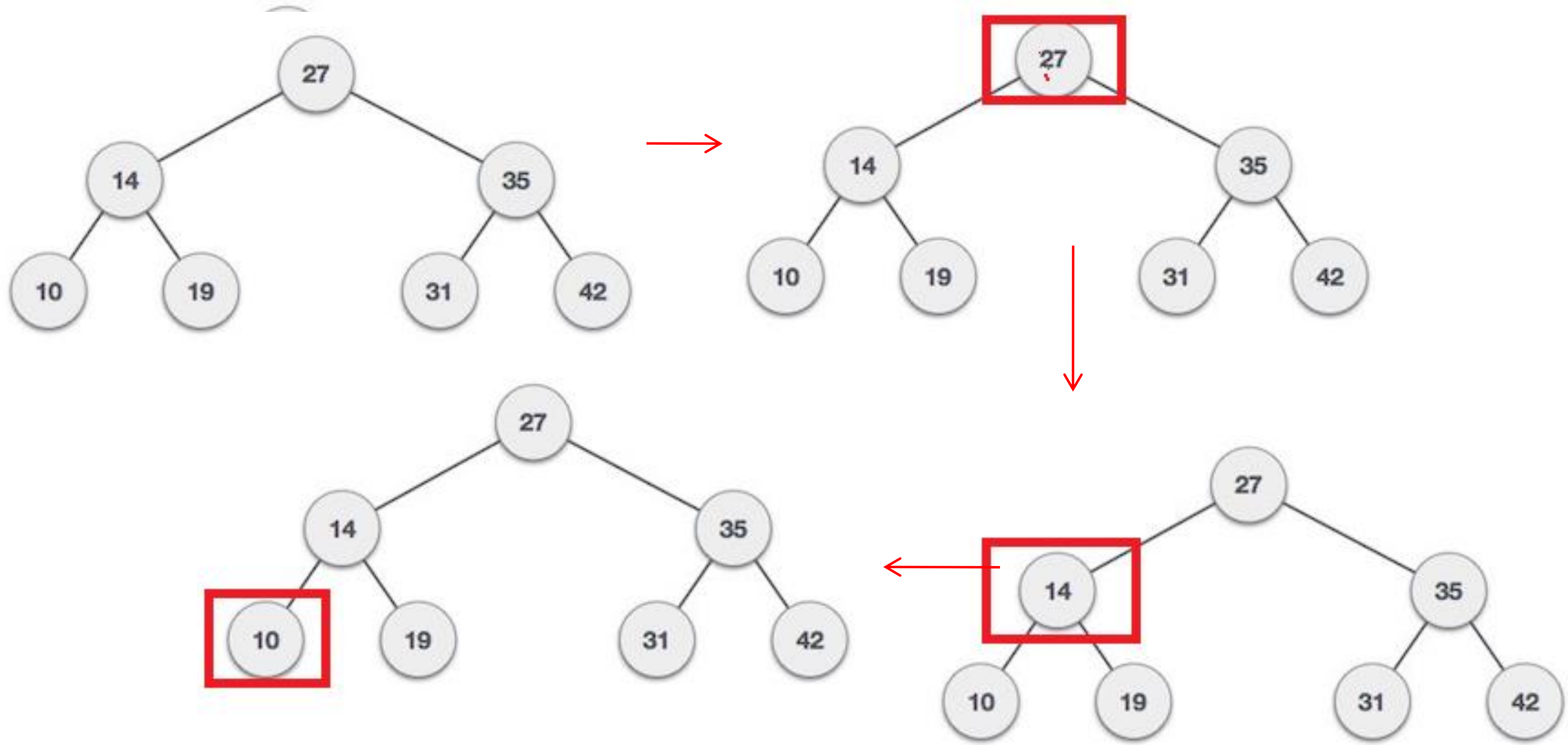
```
struct node* insert(struct node *root, int item)
{
   if(root == NULL) {
     root = new node;
     root->right = NULL;
     root->left = NULL;
     root->data = item;
     return *root;
   }
  else if(item < root->data)
     root->left=insert(root->left, item);
   else
     root->right=insert(root->right, item);
   return *root;
}
```

Inserting Specific Item to the tree



Searching Specific Item to BST

- Start search from root node
- If data is less than key value, search element in left subtree
- Otherwise search element in right subtree.
- For Example : we need to search element 10 from the BST



Pseudo code using recursion & without recursion

<table>
<tr>
<td>

```
 struct node* search(struct node* root, int key)
{
   // Base Cases: root is null or key is present at
root
   if (root == NULL || root->key == key)
      return root;
   // Key is greater than root's key
   if (root->key < key)
      return search(root->right, key);
   // Key is smaller than root's key
   return search(root->left, key);
}
```

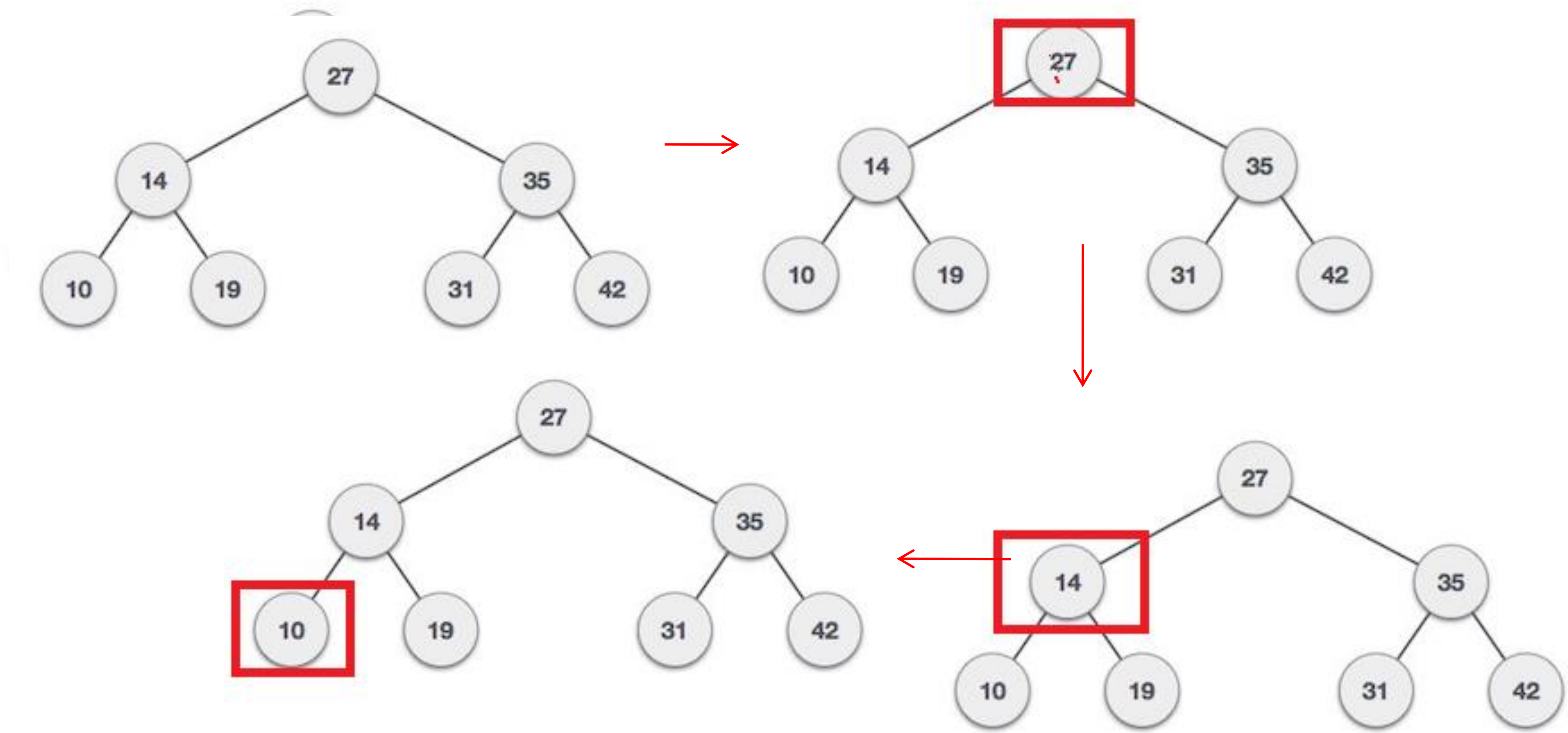
</td>
<td>

```
struct node* search(struct node* root, int
data)
{
struct node *current = root;
while(current->data != data)
  {
     if(current != NULL)
      {
        if(current->data > data)
             current = current->leftChild;
        else
            current = current->rightChild; }
      }
 return current;
}
```

</td>
</tr>
</table>

Find Smallest Node in BST

- Start search from root node
- Search element in left subtree



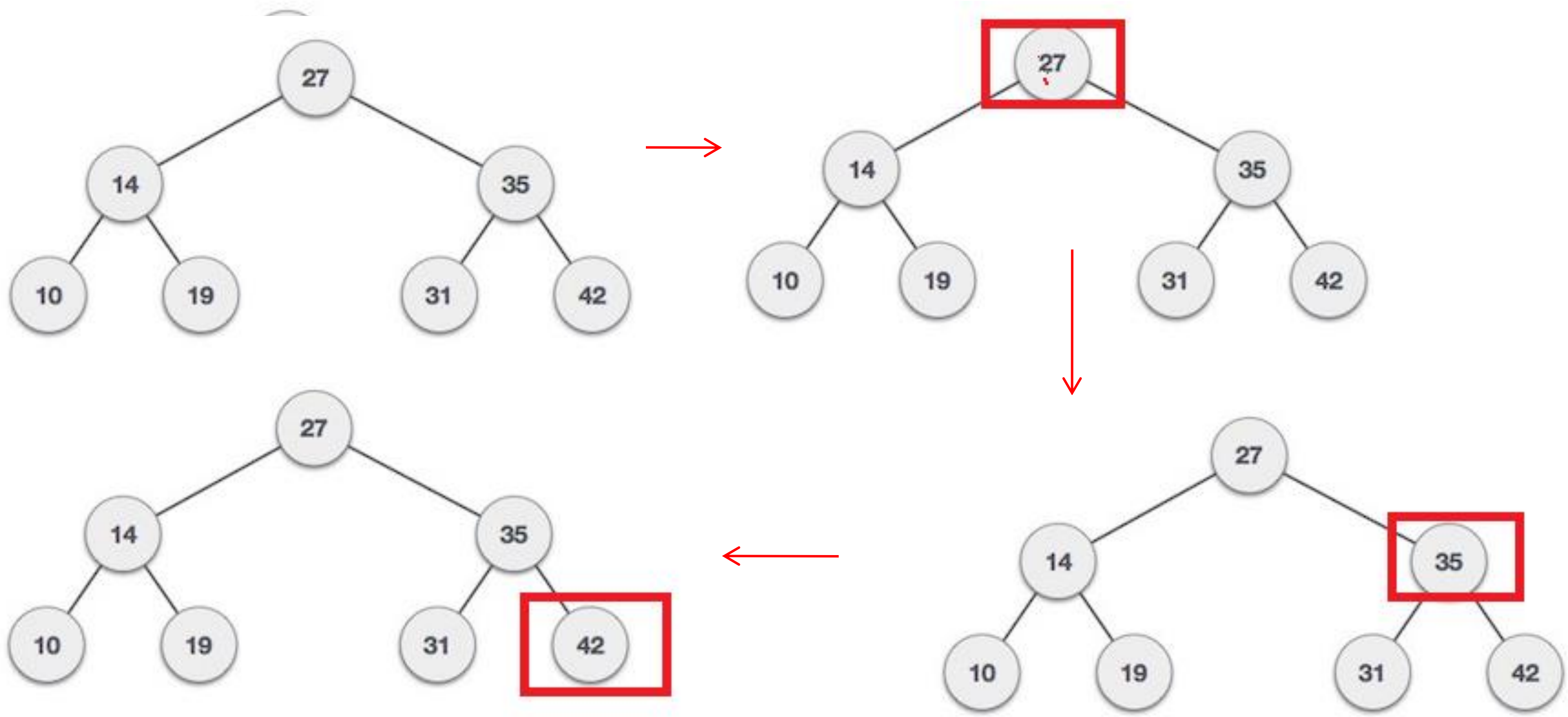
Pseudo code using recursion and without recursion

```
searchmin(struct node* root)
{
    // Base Cases: left subtree is null
    if (root->left == NULL)
       return root;
    return searchmin(root->left);

}
```

```
 minValue(struct node* node) {
  struct node* current = node;

  /* loop down to find the leftmost
leaf */
  while (current->left != NULL) {
    current = current->left;
  }
  return(current->data);
}
```

Find Maximum Node in BST

- Start search from root node
- Search element in right subtree

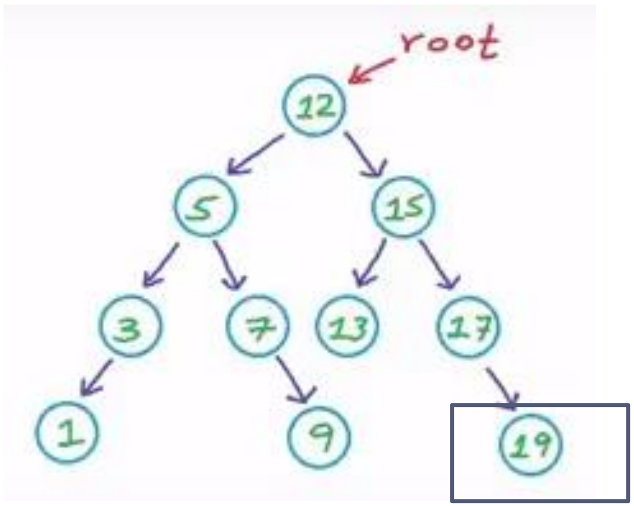


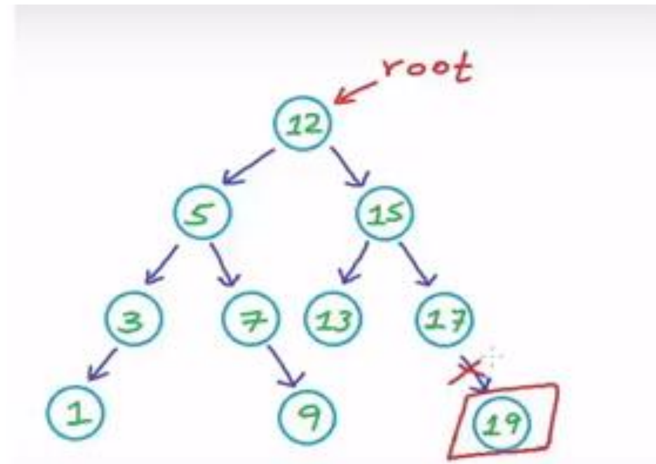
Pseudo code using recursion & without recursion

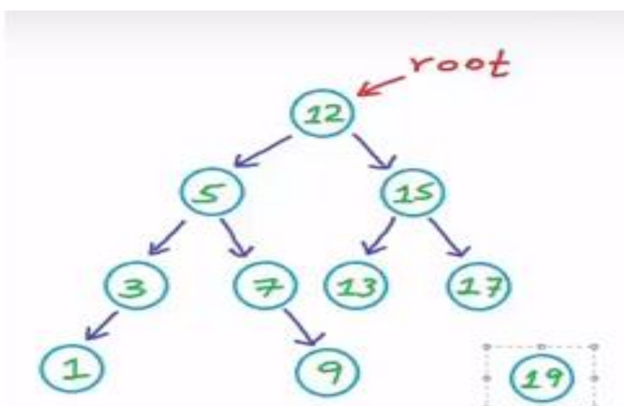
searchmax(struct node* root)
{
    // Base Cases: right subtree is null
    if (root->right == NULL)
        return root->data;
        return searchmax(root->right);
}
	maxValue(struct node* node) {
	  struct node* current = node;
	
	  /* loop down to find the leftmost leaf */
	  while (current->right!= NULL)
	{
	    current = current->right
	  }
	  return(current->data);
	}

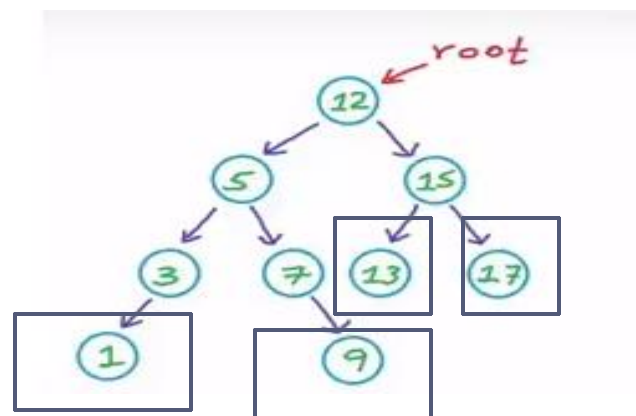
Delete a node from BST

- **Case 1**: leaf node has no child, so this node can easily wiped out from memory
- Property of BST must hold



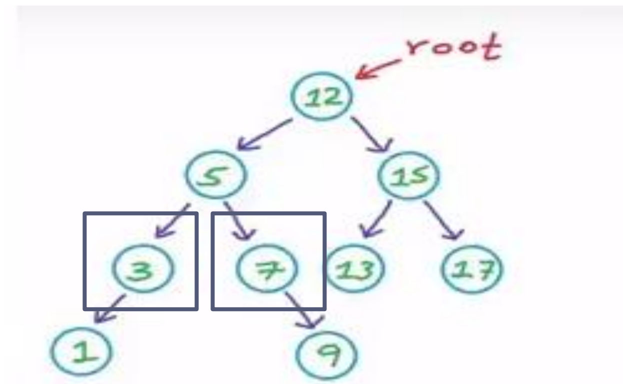


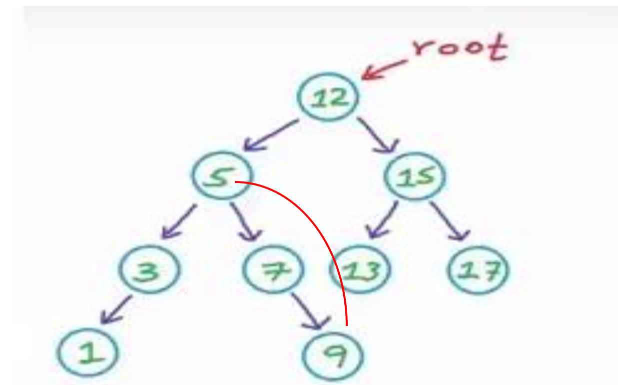


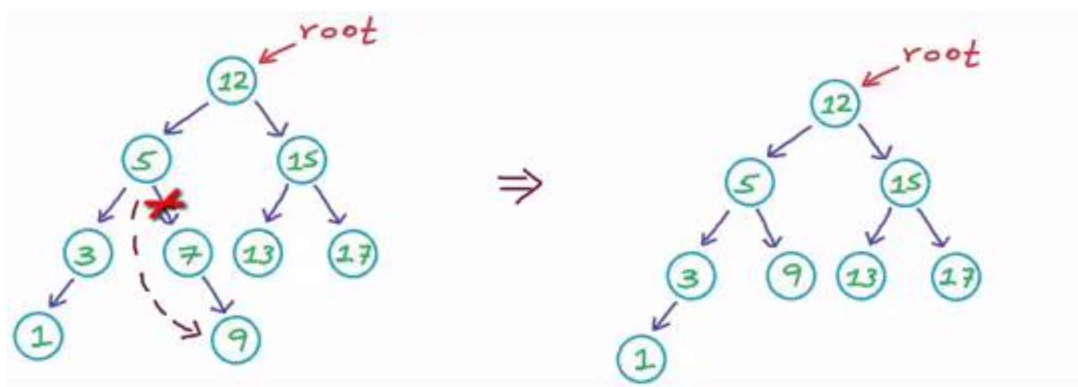


Delete a node from BST

- **Case 2**: if node has one child (left or right), then link the parent node with the child node and wipe out the node from memory
- Property of BST must hold

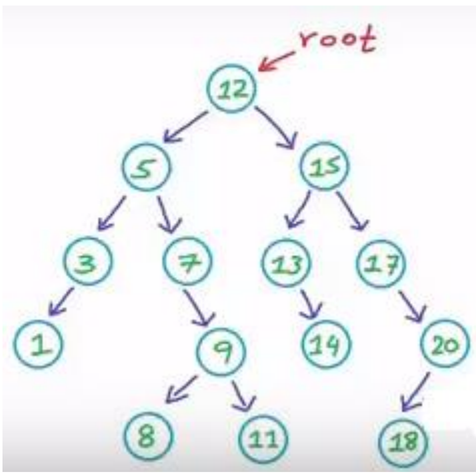


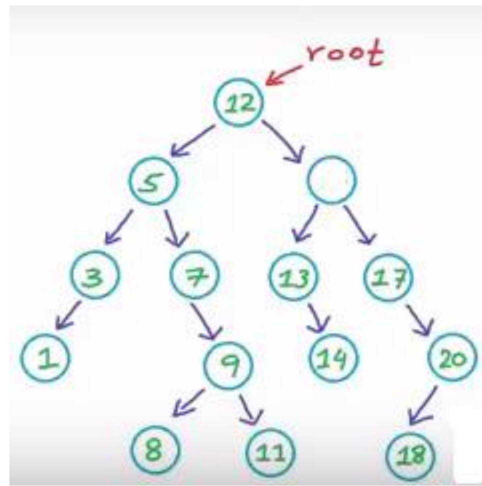


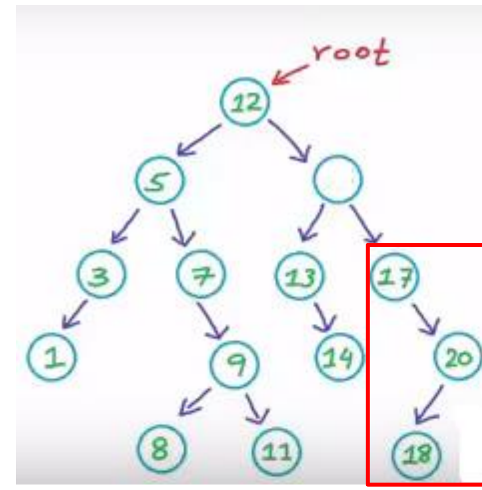


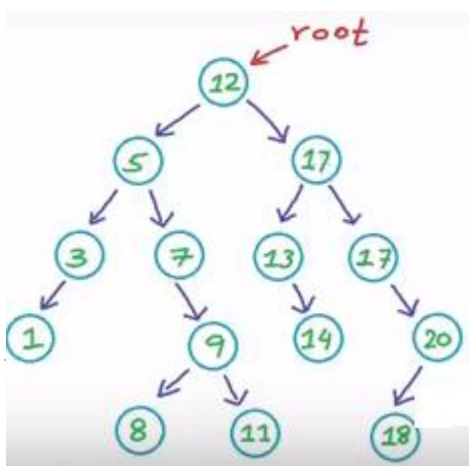
Delete a node from BST

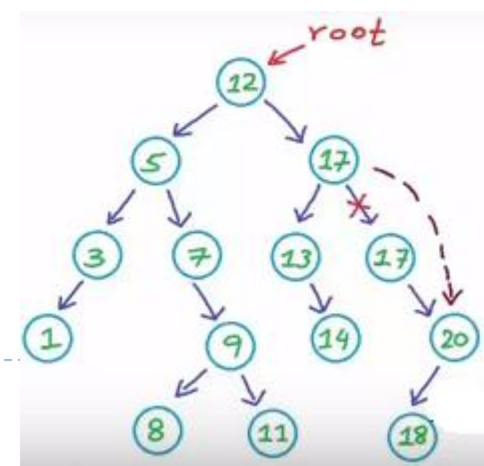
- **Case 3**: if node has two child then two case can be considered:
  - Find min node from right subtree or max node from left subtree
  - save the min value in the place of the node deleted
  - delete the dupicate value
- Property of BST must hold

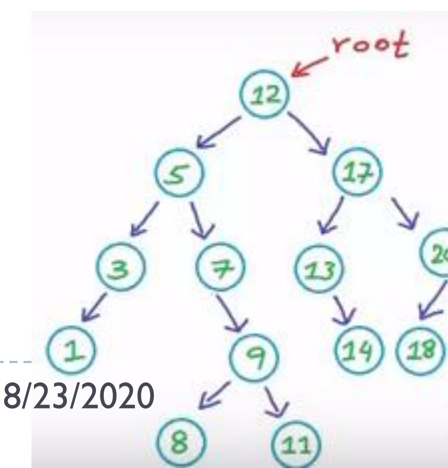












Pseudo code to delete the Node

```
Delete (struct node * root, int data)
{
    if(root == NULL) return  root;
    if (data < root->data) root->left =  Delete (root->left , data);
    if (data > root->data) root->right=  Delete (root->right , data);
    else
    {
        // case 1: no child
          if (root ->left == NULL && root->right == NULL)
              {delete root; root = NULL; return root;}
          // case 2: one child
        else   if (root ->left == NULL )
              {  struct node *temp = root;
                 root = root->right;
                 delete temp;  return root;
              }
        else   if (root ->right == NULL )
              { struct node *temp = root;
                 root = root->left;
                 delete temp;  return root;
              }
    }
}
```

Pseudo code to delete the Node

```
Delete (struct node * root, int data)
{
    if(root == NULL) return  root;
    if (data < root->data) root->left =  Delete (root->left , data);
    if (data > root->data) root->right=  Delete (root->right , data);
    else
    { // case 3
        struct node *temp = findmin( root->right);
        root->data = temp->data;
         root->right = delete (root->right, temp->data);
          return root;
    }

}
```