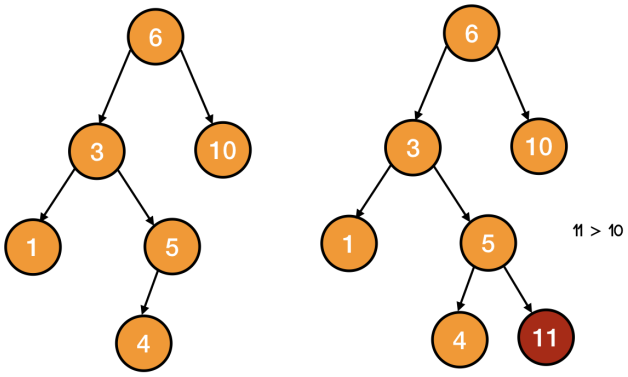
[redacted] (/)

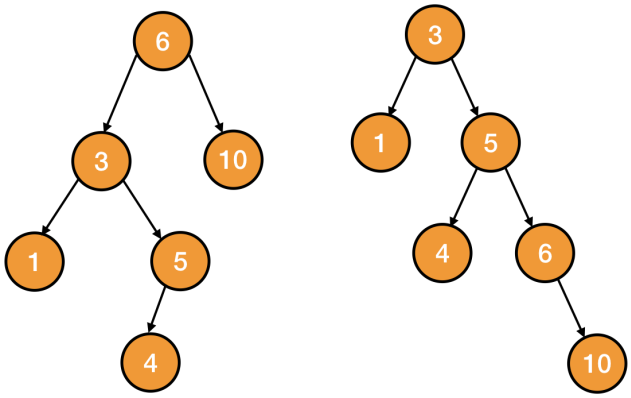
Binary Search Trees

Binary Search Tree (or BST) is a special kind of binary tree in which the values of all the nodes of the left subtree of any node of the tree are smaller than the value of the node. Also, the values of all the nodes of the right subtree of any node are greater than the value of the node.



Binary Search Tree Not a Binary Search Tree

In the above picture, the second tree is not a binary search tree because all the values of all the nodes of the left subtree are not smaller than all the nodes of the right subtree.

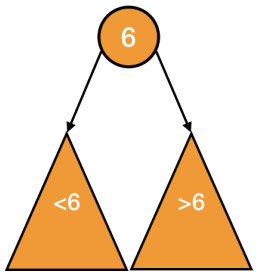


Two Binary Search Trees Representing Same Set

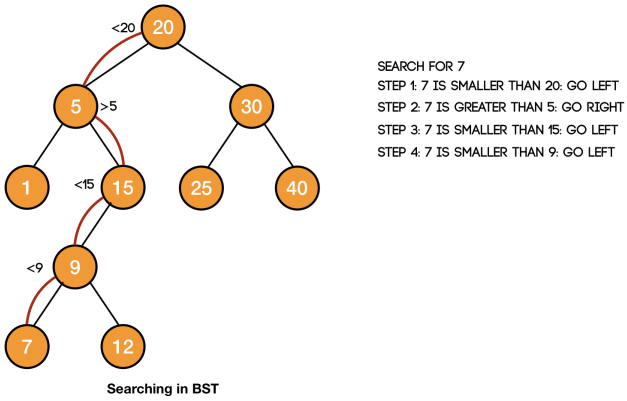
As the name suggests, binary search tree is usually used to perform an optimized search. So, let's look at the searching process of a BST.

Searching a BST

The property that all the values lesser than the value of a node lies on the left subtree and all the values greater than the value of a node lies on the right subtree helps to perform the searching in $O(h)$ time (where h is the height of the tree).



Suppose we are on a node and the value to be searched is smaller than the value of the node. In that case, we will search for the value in the left subtree. Otherwise, if the value to be searched is larger, we will just search the right subtree.



Searching in BST

So, our function will take the element to be searched (x) and the tree (T) i.e., `SEARCH(x, T)`.

We will perform the search operation if the root of the tree is not null - `if(T.root != null)`.

We will first check if the data to be searched is at the root or not. If it is at the root, we will return it.

```
if(T.root.data == x)
    return r
```

Otherwise, we will search the left subtree if the value to be searched is smaller.

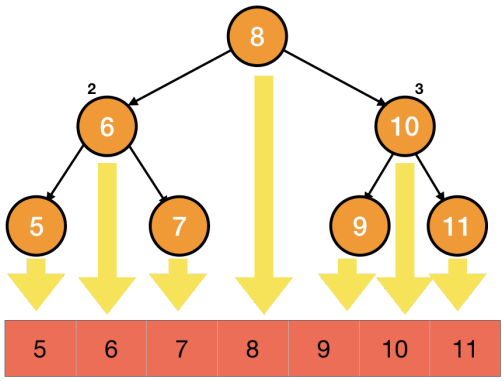
```
else if(T.root.data > x)
    return SEARCH(x, T.root.left)
```

And if the value to be searched is larger, we will search the right subtree.

```
else
    return SEARCH(x, T.root.right)
```

```
SEARCH(x, T)
    if(T.root != null)
        if(T.root.data == x)
            return r
        else if(T.root.data > x)
            return SEARCH(x, T.root.left)
        else
            return SEARCH(x, T.root.right)
```

</>
Inorder traversal prints all the data of a binary search tree in a sorted order.

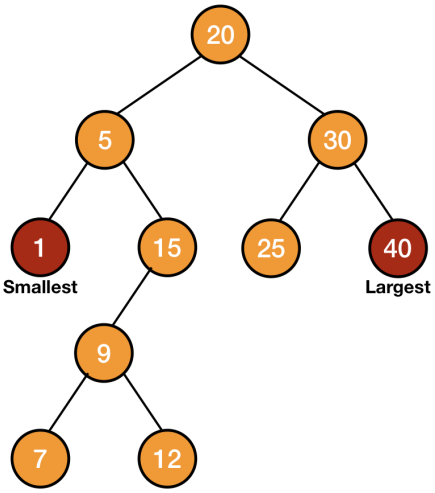


To search an element in the tree, we are taking a simple path from the root to leaf. Thus, searching in a binary search tree is done $O(h)$ time.

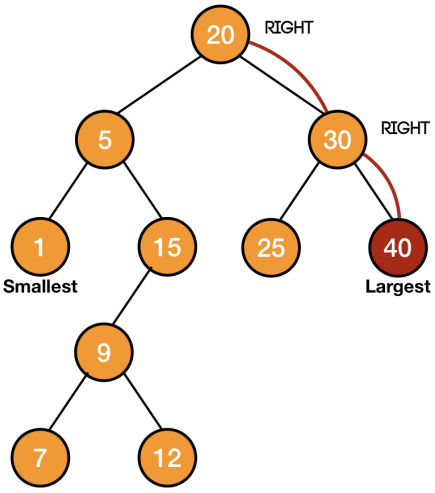
We also get the maximum and the minimum element of a BST using `MAXIMUM` and `MINIMUM` operations. Let's have a look at these.

Maximum/Minimum element of a BST

The smallest element of a binary search tree is the leftmost element of the tree and the largest element is the rightmost one.



So, to find the maximum/minimum element, we have to find the rightmost/leftmost element respectively. Thus to find the maximum element, we will go to the right subtree every time until the rightmost element is found i.e., the right child is null.



So, we will start by passing a node (n) to our function - `MAXIMUM(n)`.

Then, we will move to the right subtree every time until the right child is not null.

```
if(n.right == null)
     return n
else
     return MAXIMUM(n.right)
```

```
MAXIMUM(T)
     if(n.right == null)
          return n
     else
          return MAXIMUM(n.right)
```

Similarly, we can write the MINIMUM function.

```
MINIMUM(n)
     if(n.left == null)
          return n
     else
          return MAXIMUM(n.left)
```

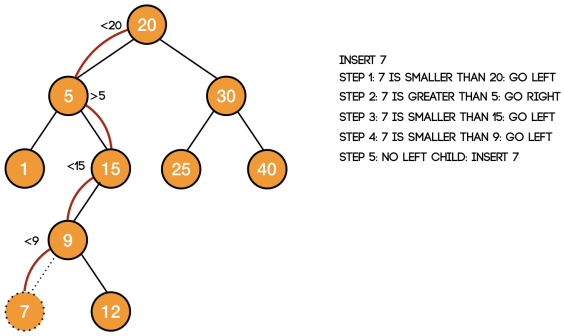
In these two operations also, we are starting from the root and moving to leaf, thus these are also $O(h)$ operations.

We have learned the basic operations to be performed on a binary search tree. Let's learn to insert and delete nodes from a binary search tree so that we can make a binary search tree.

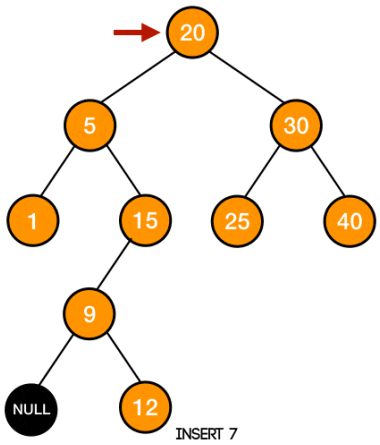
Insertion in BST

We can't insert any new node anywhere in a binary search tree because the tree after the insertion of the new node must follow the binary search tree property.

To insert an element, we first search for that element and if the element is not found, then we insert it.



Thus, we will use a temporary pointer and go to the place where the node is going to be inserted.

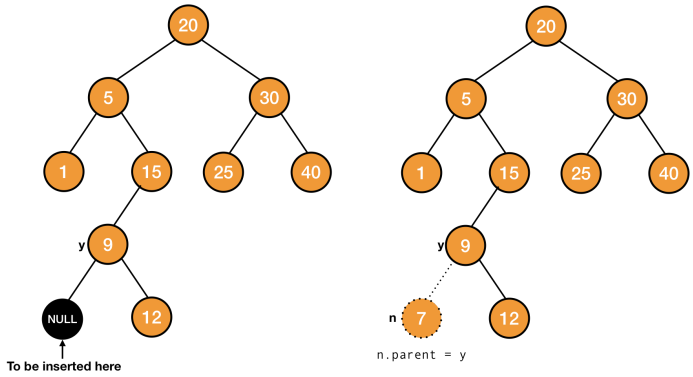



```
INSERT(T, n)
     temp = T.root
     while temp != NULL
          if n.data < temp.data
               temp = temp.left
          else
               temp = temp.right
```

Here, we are starting from the root of the tree - `temp = T.root` and then moving to the left subtree if the data of the node to be inserted is less than the current node - `if n.data < temp.data → temp = temp.left`.

Otherwise, we are moving right.

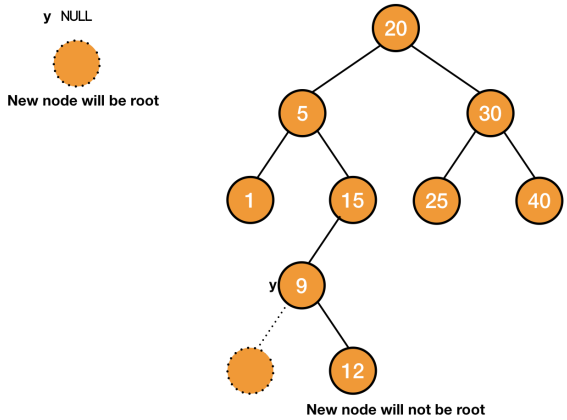
We need to make the last node in the above iteration the parent of the new node. So, let's use a variable for this.



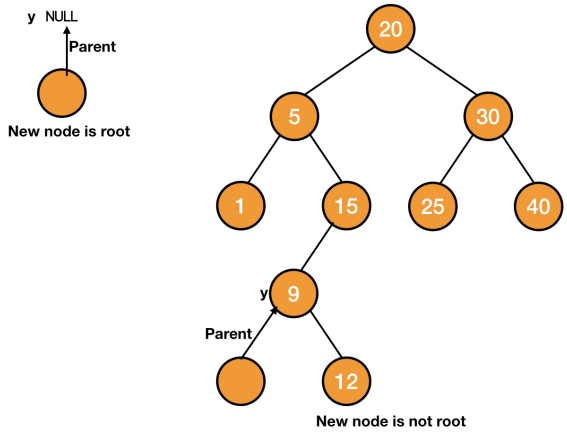

```
INSERT(T, n)
    temp = T.root
    y = NULL
    while temp != NULL
        y = temp
        if n.data < temp.data
            temp = temp.left
        else
            temp = temp.right
```

We have used a variable *y*. When the tree won't have any node, the new node will be the root of the tree and its parent will be `NULL`. So, initially the value of *y* is `NULL`. In this case, the loop will also not run.

Otherwise, *y* will point to the last node.

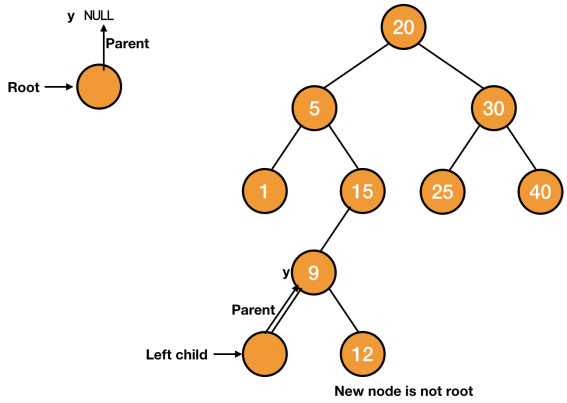


After this, we will make *y* the parent of the new node.




```
n.parent = y
```

Lastly, we need to make the new node the child of *y*. If *y* is null, the new node will be the root of the tree, otherwise we will check if the data of the new node is larger or smaller than the data of *y*, and accordingly we will make it either the left or the right child.



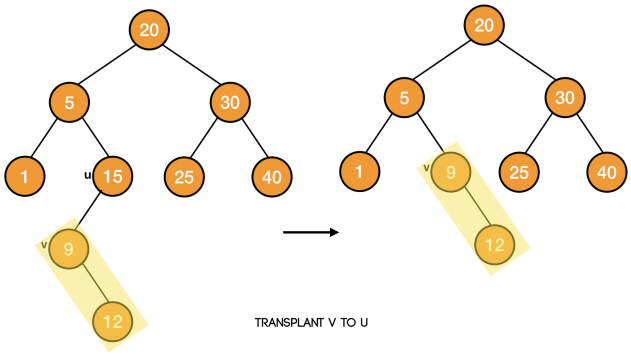

```
if y==NULL
     T.root = n
else if n.data < y.data
     y.left = n
else
     y.right = n
```

```
INSERT(T, n)
     temp = T.root
     y = NULL
     while temp != NULL
          y = temp
          if n.data < temp.data
               temp = temp.left
          else
               temp = temp.right
     n.parent = y
     if y==NULL
          T.root = n
     else if n.data < y.data
          y.left = n
     else
          y.right = n
```

Deletion in BST

The last operation we need to do on a binary search tree to make it a full-fledged working data structure is to delete a node.

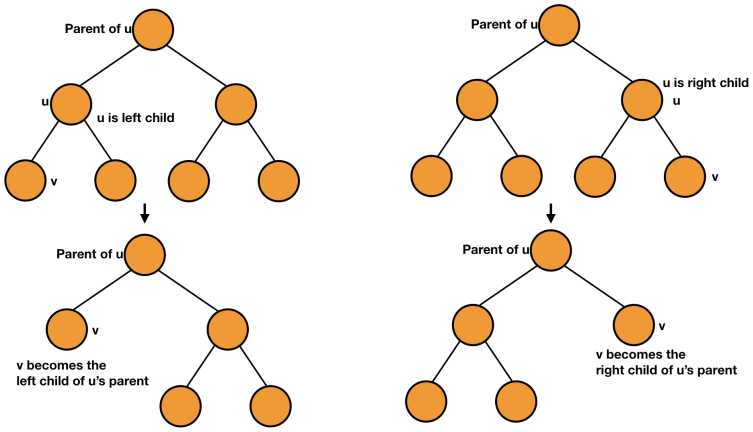
To delete a node from a BST, we will replace a subtree with another one i.e., we transplant one subtree in place of another.



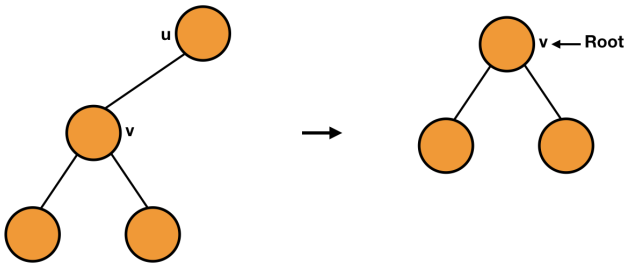
As we are going to use this technique in our delete procedure, so let's first write the code to transplant a subtree rooted at node *v* in place of the subtree rooted at node *u*.

Our function to transplant will take the tree *T*, nodes *u* and *v* - `TRANSPLANT(T, u, v)`.

Now, we want to place the subtree rooted at node *v* in place of the subtree rooted at node *u*. It means that we need to make *v* the child of the parent of *u* i.e., if *u* is the left child, then *v* will become the left child of *u*'s parent. Similarly, if *u* is the right child, then *v* will become the right child of *u*'s parent.



It is also possible that *u* doesn't have any parent i.e., *u* is the root of the tree *T*. In that case, we will simply make *v* as the root of the tree.

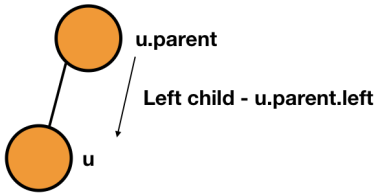


So, we will first check if *u* is root or not i.e., if the parent of *u* is `NULL` or not.

```
if u.parent == NULL //u is root
   T.root = v
```

Now, we will check if *u* is the left child or the right child. Accordingly, we will place *v*.

If *u* is the left child, then the *left* of *u*'s parent will be *u* i.e., `u == u.parent.left` will be true and we will make *v* as its left child i.e., `u.parent.left = v`.



```
if u.parent == NULL
  ...
elseif u == u.parent.left //u is left child
  u.parent.left = v
else //u is right child
  u.parent.right = v
```

Lastly, we also need to point the parent of *v* to the parent of *u*.

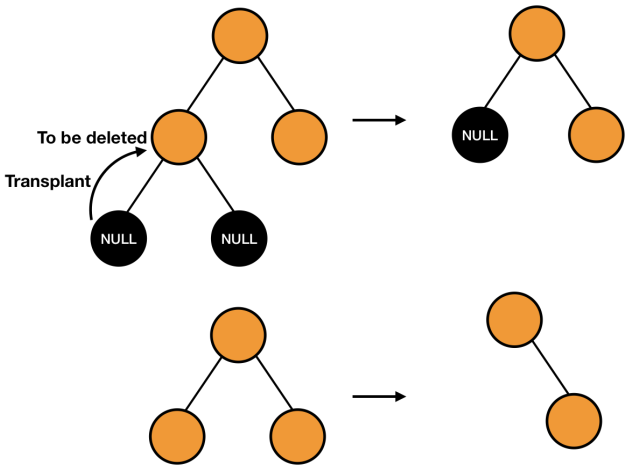
```
if v != NULL
  v.parent = u.parent
```

So, the overall code would be:

```
TRANSPLANT(T, u, v)
    if u.parent == NULL //u is root
        T.root = v
    elseif u == u.parent.left //u is left child
        u.parent.left = v
    else //u is right child
        u.parent.right = v
    if v != NULL
        v.parent = u.parent
```

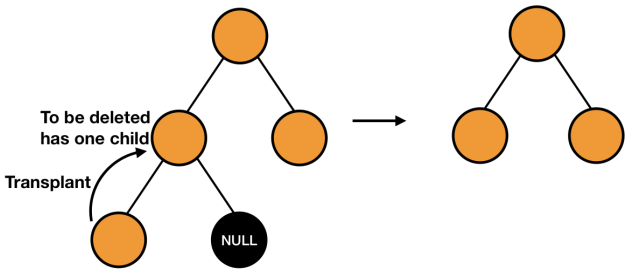
Let's focus on the deletion of a node from a binary search tree.

Suppose the node to be deleted is a leaf, we can easily delete that node by pointing the parent of that node to `NULL`.

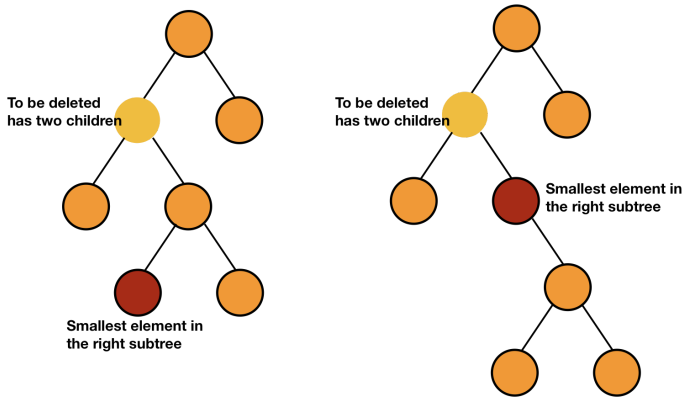


We can also say that we are transplanting the right or the left child (both are `NULL`) to the node to be deleted.

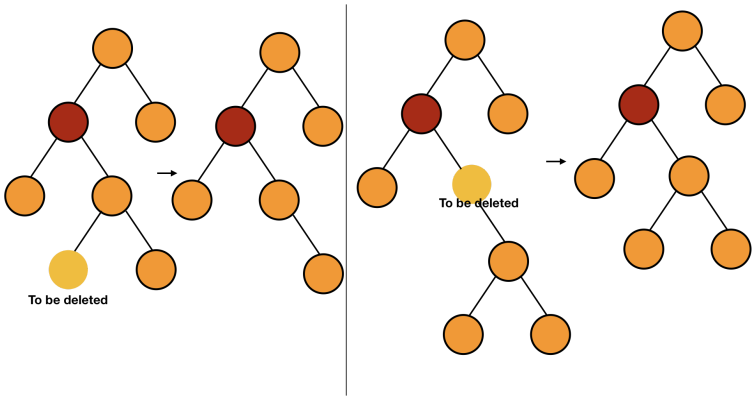
We can also delete a node with only one child by transplanting its child to the node and it will not affect the property of the binary search tree.



But things will become a bit little complicated when the node to be deleted has both the children.

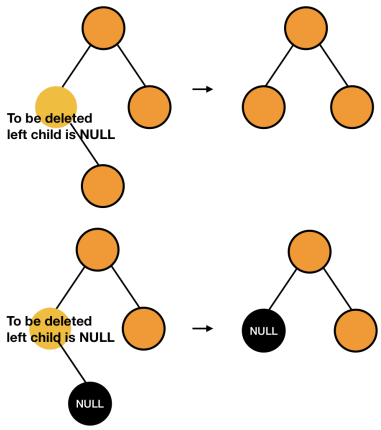


In this case, we can find the smallest element of the right subtree of the node to be deleted (element with no left child in the right subtree) and replace its content with the node to be deleted.



Doing so is not going to affect the property of binary search tree because it is the smallest element of the right subtree, so all the elements in the right subtree are still greater than it. Also, all the elements in the left subtree were smaller than it because it was in the right subtree, so they are still smaller.

The smallest element of the right subtree will have either have no child or one child because if it has left child, then it will not be the smallest element. So, we can delete this node easily as discussed in the first two cases.



We have understood the concepts of deleting a node, we can now write the code to do so.

We will start by passing the tree *T* and the node to be deleted *z* to the function - `DELETE(T, z)`.

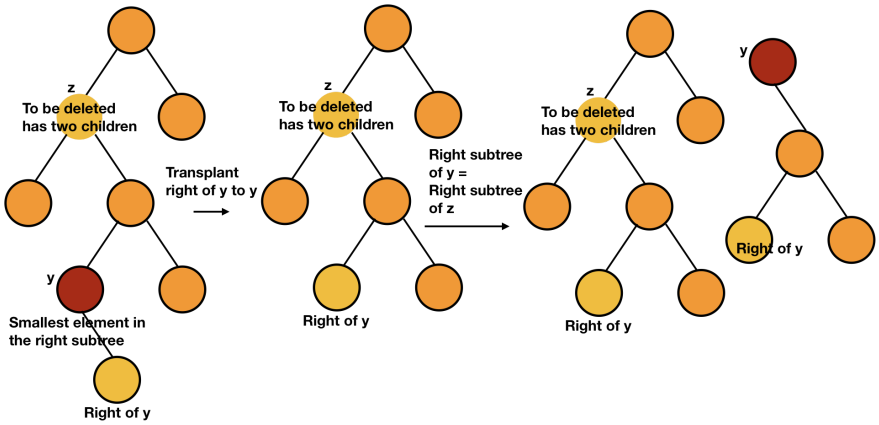
Now, we have to check the number of children of the node *z*. We will first check if the left child of the node *z* is `NULL` or not. If the left child is `NULL`, then either it has only one child (right one) or none. In both the cases, we can transplant its right child to it.

```
DELETE(T, z)
  if z.left == NULL
    TRANSPLANT(T, z, z.right)
```

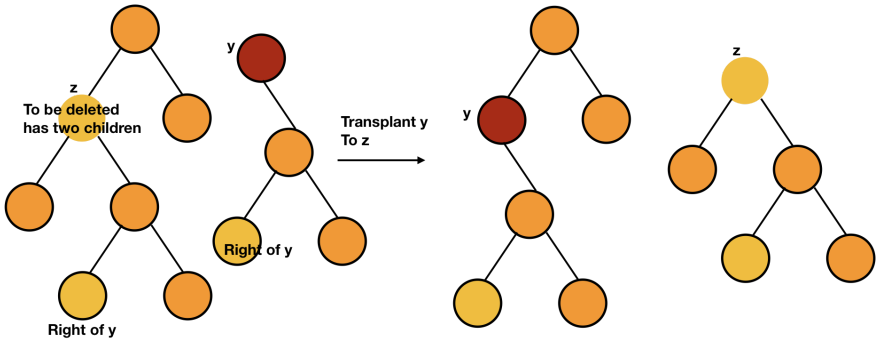
Similarly, we will next check if the right child is `NULL` or not.

```
DELETE(T, z)
   ...
   elseif z.right == NULL
      TRANSPLANT(T, z, z.left)
```

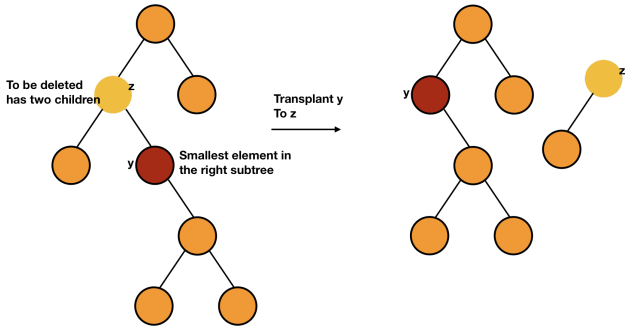
If none of the above cases are true, the node *z* has both children and we will find the minimum in the right subtree (*y*). Now, we have to put this minimum node (*y*) in the place of *z*. Firstly, we will transplant the right of *y* to *y* and then take the right subtree of *z* and make it the right subtree of *y*.



After this, we will transplant *y* to *z*.

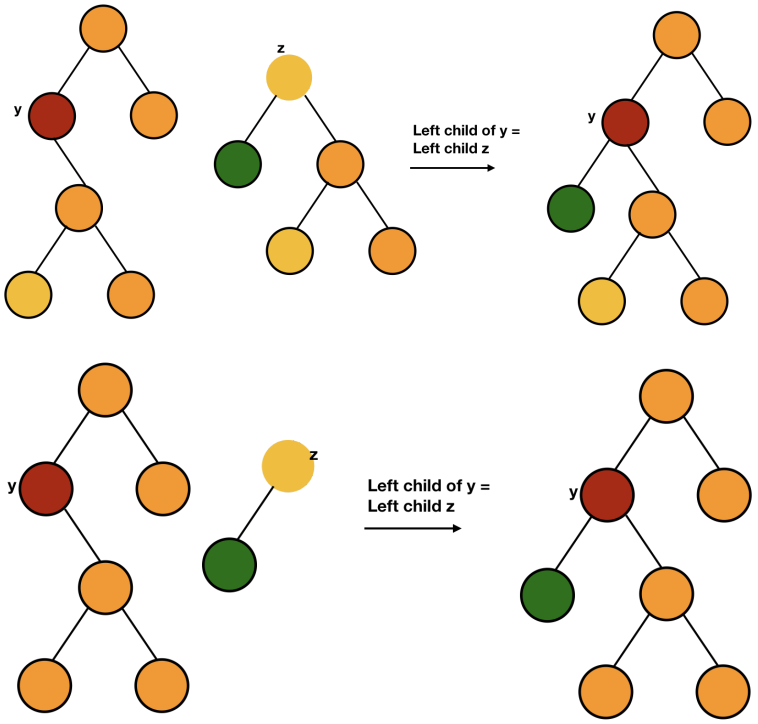


However, it can also be possible that the minimum node is the direct child of the node *z*. In that case, we will just transplant *y* to *z*.




```
DELETE(T, z)
   ...
   else
      y = MINIMUM(z.right) //minimum element in right subtree
      if y.parent != z //z is not direct child
         TRANSPLANT(T, y, y.right)
         y.right = z.right
         y.right.parent = y
      TRANSPLANT(T, z, y)
```

After this, we will change the left child of *y* to the left child of *z*.




```
DELETE(T, z)
  ...
  else
    y = MINIMUM(z.right) //minimum element in right subtree
    ...       TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.parent = y
```

```
DELETE(T, z)
    if z.left == NULL
        TRANSPLANT(T, z, z.right)
    elseif z.right == NULL
        TRANSPLANT(T, z, z.left)
    else
        y = MINIMUM(z.right) //minimum element in right subtree
        if y.parent != z //z is not direct child
            TRANSPLANT(T, y, y.right)
            y.right = z.right
            y.right.parent = y
        TRANSPLANT(T, z, y)
        y.left = z.left
        y.left.parent = y
```

C **Python** **Java**

```
class Node:
  def __init__(self, data):
    self.data = data
    self.right = None
    self.left = None
    self.parent = None

class BinarySearchTree:
  def __init__(self):
    self.root = None

  def minimum(self, x):
    while x.left != None:
      x = x.left
    return x

  def insert(self, n):
    y = None
    temp = self.root
    while temp != None:
      y = temp
      if n.data < temp.data:
        temp = temp.left
      else:
        temp = temp.right

    n.parent = y

    if y == None: #newly added node is root
      self.root = n
    elif n.data < y.data:
      y.left = n
    else:
      y.right = n

  def transplant(self, u, v):
    if u.parent == None:
      self.root = v
    elif u == u.parent.left:
      u.parent.left = v
    else:
      u.parent.right = v

    if v != None:
      v.parent = u.parent

  def delete(self, z):
    if z.left == None:
      self.transplant(z, z.right)

    elif z.right == None:
      self.transplant(z, z.left)

    else:
      y = self.minimum(z.right) #minimum element in right subtree
      if y.parent != z:
        self.transplant(y, y.right)
        y.right = z.right
        y.right.parent = y

      self.transplant(z, y)
      y.left = z.left
      y.left.parent = y

  def inorder(self, n):
    if n != None:
      self.inorder(n.left)
      print(n.data)
      self.inorder(n.right)

if __name__ == '__main__':
  t = BinarySearchTree()

  a = Node(10)
  b = Node(20)
  c = Node(30)
  d = Node(100)
```

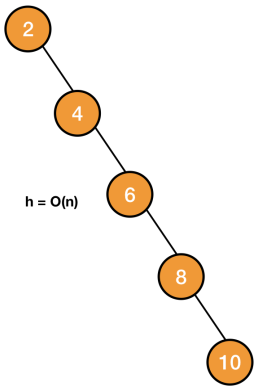
```
e = Node(90)
f = Node(40)
g = Node(50)
h = Node(60)
i = Node(70)
j = Node(80)
k = Node(150)
l = Node(110)
m = Node(120)

t.insert(a)
t.insert(b)
t.insert(c)
t.insert(d)
t.insert(e)
t.insert(f)
t.insert(g)
t.insert(h)
t.insert(i)
t.insert(j)
t.insert(k)
t.insert(l)
t.insert(m)

t.delete(a)
t.delete(m)

t.inorder(t.root)
```

In this chapter, we saw that we can insert, search and delete any item in a binary search tree in $O(h)$ time, where h is the height of the tree. But the problem is that for an unbalanced binary tree, h can be pretty large and can go up to n, the number of nodes in the tree.



In those cases, making a binary search tree won't be of much help rather than using a simple singly linked list. There are some techniques to get a balanced binary search tree after every operation which we are going to study in the next few chapters.

> “ Shall I refuse my dinner because I do not fully understand the process of digestion? ”
>
> - Oliver Heaviside

PREV (/course/data-structures-binary-trees/) (/course/data-structures-red-black-trees/) NEXT

Further Readings

- Binary Search Tree (/blog/article/binary-search-tree/)