**Conversion of Infix to Postfix Expression using Stack**

To convert Infix expression to Postfix expression, we will use the **stack** data structure. By scanning the infix expression from left to right,if we get any operand, simply add it to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

**Infix Expression**: Infix Expression contains operator in-between every pair of operands,Expression of the form a op b.

**Postfix expression**: Postfix Expression contains operator followed for every pair of operands,Expression of the form a b op.

**Why postfix representation of the expression?**

* Infix expressions are readable and solvable by humans because of easily distinguishable order of operators,but compiler doesn't have integrated order of operators.
* Hence to solve the Infix Expression compiler will scan the expression multiple times to solve the sub-expressions in expressions orderly which is very in-efficient.
* To avoid this traversing, Infix expressions are converted to Postfix expression before evaluation.

**Algorithm**

* **Step 1** : Scan the Infix Expression from left to right.
* **Step 2** : If the scanned character is an operand, append it with final Infix to Postfix string.
* **Step 3** : Else,
  + **Step 3.1** : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a ‘(‘ or ‘[‘ or ‘{‘), push it on stack.
* **Step 3.2** : Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
* **Step 4** : If the scanned character is an ‘(‘ or ‘[‘ or ‘{‘, push it to the stack.
* **Step 5** : If the scanned character is an ‘)’or ‘]’ or ‘}’, pop the stack and and output it until a ‘(‘ or ‘[‘ or ‘{‘ respectively is encountered, and discard both the parenthesis.
* **Step 6** : Repeat steps 2-6 until infix expression is scanned.
* **Step 7** : Print the output
* **Step 8** : Pop and output from the stack until it is not empty.

**Example**

**Infix Expression : 3+4\*5/6**

**Step 1** : Initially Stack is Empty ans the very first literal of Infix Expression is '3' which is operand hence push it on output stack.

Stack :

Output : 3

**Step 2** : Next literal of expression is + which is operand, hence needed to be pushed on stack but intially stack is empty hence literal will directly pushed on to stack.

Stack : +

Output : 3

**Step 3** : Further 4 is an operand should be pushed on stack.

Stack : +

Output : 3 4

**Step 4** : Next literal is \* which is an operator, as stack is not empty, priority should be checked of instack operator(top of stack) and of incoming operator i.e \* as priority of instack operator is less than incoming operator, \* will be pushed on to stack.

Stack : + \*

Output : 3 4

**Step 5** : Next literal is 5 which is an operand, hence should be pushed on to output stack.

Stack : + \*

Output : 3 4 5

**Step 6** : Next literal is / which is an operator, as stack is not empty, priority should be checked of instack operator(top of stack) i.e \* and of incoming operator i.e /, as priority of / and \* are equal hence \* will be poped out of stack and will be stored on output stack and operator / will be stored on stack.

Stack : + /

Output : 3 4 5 \*

**Step 7** : Next literal is 6 which is an operand, hence should be pushed on output stack.

Stack : + /

Output : 3 4 5 \* 6

**Step 8** : As now all litrals are traversed, despite stack is not empty, hence pop all litrals from stack and pushed it on to output stack.

**Postfix Expression : 3 4 5 \* 6 / +**

**Implementation**

#include <iostream>

#include <stack>

class InfixToPostfix

{

public:

InfixToPostfix(const std::string &expression) : expression\_(expression) { }

int getPrecedenceOfOperators(char);

std::string convertInfixToPostfix();

private:

std::string expression\_;

};

int InfixToPostfix::getPrecedenceOfOperators(char ch)

{

if(ch == '+' || ch == '-')

return 1;

if(ch == '\*' || ch == '/')

return 2;

if(ch == '^')

return 3;

else

return 0;

}

std::string InfixToPostfix::convertInfixToPostfix()

{

std::stack <char> stack1;

std::string infixToPostfixExp = "";

int i = 0;

while(expression\_[i] != '\0')

{

//if scanned character is open bracket push it on stack

if(expression\_[i] == '(' || expression\_[i] == '[' || expression\_[i] == '{')

stack1.push(expression\_[i]);

//if scanned character is opened bracket pop all literals from stack till matching open bracket gets poped

else if(expression\_[i] == ')' || expression\_[i] == ']' || expression\_[i] == '}')

{

if(expression\_[i] == ')')

{

while(stack1.top() != '(')

{

infixToPostfixExp = infixToPostfixExp + stack1.top();

stack1.pop();

}

}

if(expression\_[i] == ']')

{

while(stack1.top() != '[')

{

infixToPostfixExp = infixToPostfixExp + stack1.top();

stack1.pop();

}

}

if(expression\_[i] == '}')

{

while(stack1.top() != '{')

{

infixToPostfixExp = infixToPostfixExp + stack1.top();

stack1.pop();

}

}

stack1.pop();

}

//if scanned character is operator

else if(expression\_[i] == '+' || expression\_[i] == '-' || expression\_[i] == '\*' || expression\_[i] == '/' || expression\_[i] == '^')

{

//very first operator of expression is to be pushed on stack

if(stack1.empty()) {

stack1.push(expression\_[i]);

} else{

/\*

\* check the precedence order of instack(means the one on top of stack) and incoming operator,

\* if instack operator has higher priority than incoming operator pop it out of stack&put it in

\* final postifix expression, on other side if precedence order of instack operator is less than i

\* coming operator, push incoming operator on stack.

\*/

if(getPrecedenceOfOperators(stack1.top()) >= getPrecedenceOfOperators(expression\_[i]))

{

infixToPostfixExp = infixToPostfixExp + stack1.top();

stack1.pop();

stack1.push(expression\_[i]);

}

else

{

stack1.push(expression\_[i]);

}

}

}

else

{

//if literal is operand, put it on to final postfix expression

infixToPostfixExp = infixToPostfixExp + expression\_[i];

}

i++;

}

//poping out all remainig operator literals & adding to final postfix expression

if(!stack1.empty())

{

while(!stack1.empty())

{

infixToPostfixExp = infixToPostfixExp + stack1.top();

stack1.pop();

}

}

return infixToPostfixExp;

}

int main()

{

InfixToPostfix p("a+b\*c/d-q");

std::cout << "\nPostfix expression : " << p.convertInfixToPostfix();

}

**Complexity**

The time and space complexity of Conversion of Infix expression to Postfix expression algorithm is :

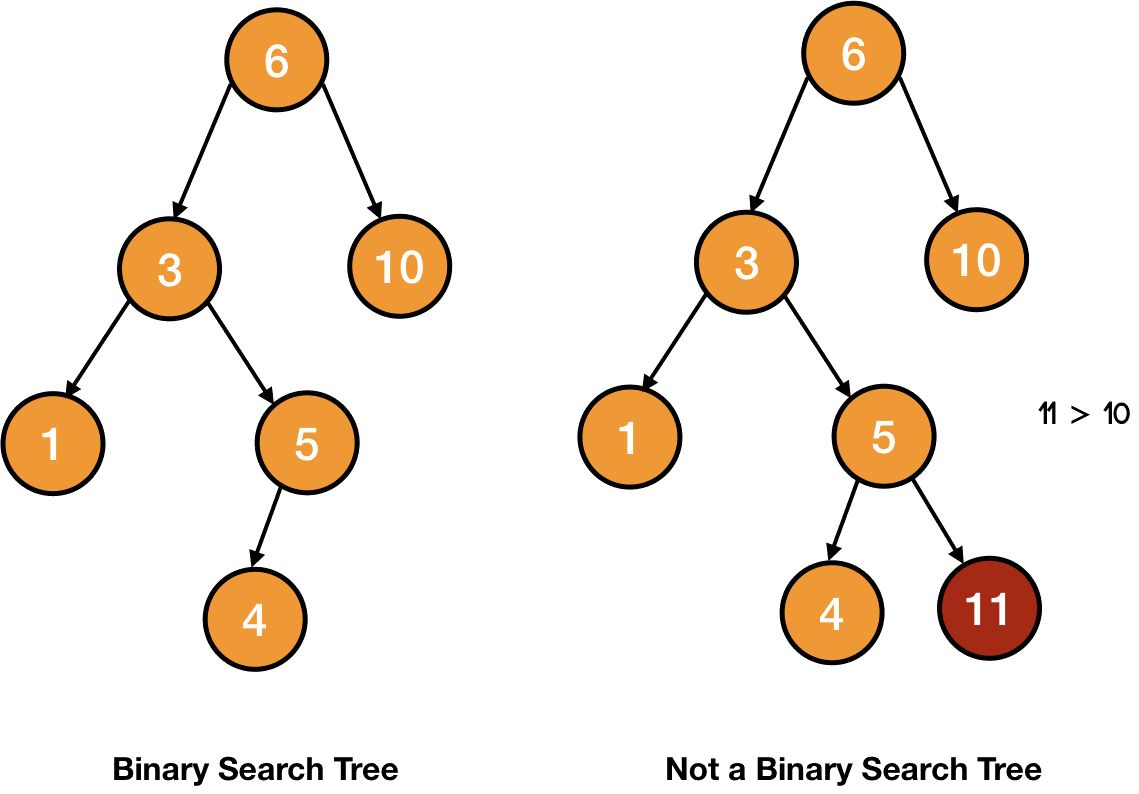
* Worst case time complexity: **Θ(n^2)**
* Average case time complexity: **Θ(n^2)**
* Best case time complexity: **Θ(n^2)**
* Space complexity: **Θ(n)**

where N is the number of literals in Infix Expression

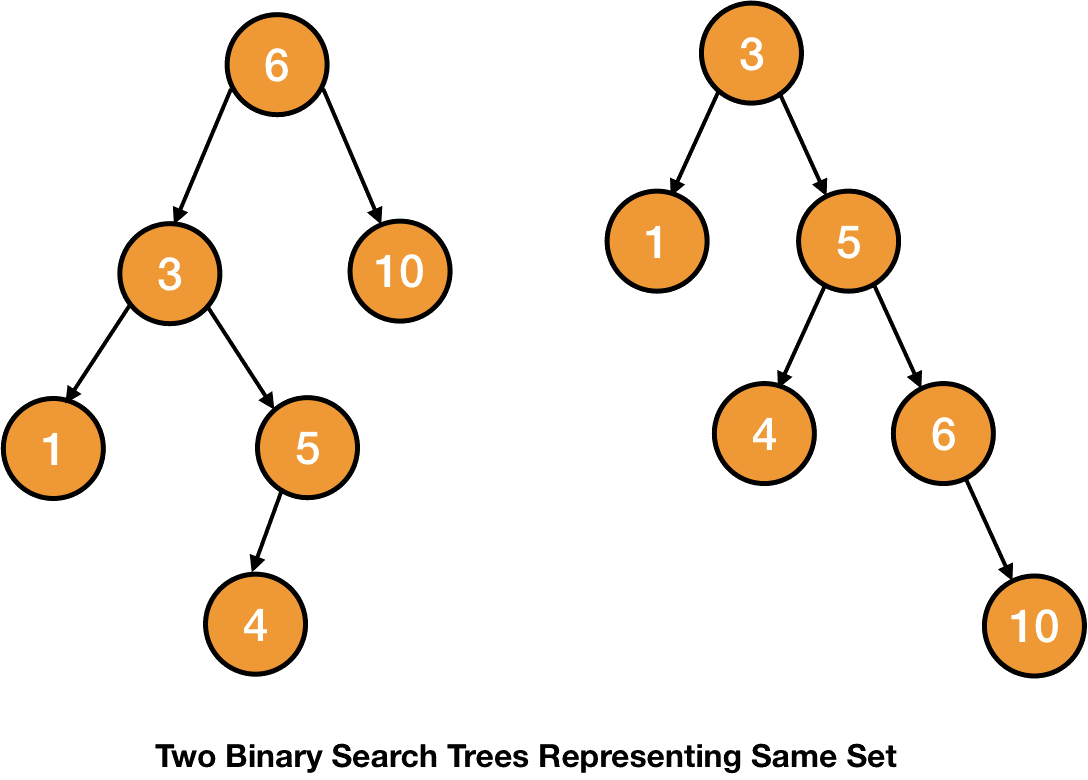
* Complexity of algorithm in both worst and best case is O(n^2), as expression is iterated two times simultaneously, firstly for scanning the infix expression and secondly while poping out of stack.  
  Eg : a + b - d  
  \* As in above Infix expression, O(n) will be the complexity for scanning each literal, while at the same time we pop the literals from stack, hence the complexity of algorithm is O(n\*n) i.e : O(n^2).
* For storing Infix expression of n literals the space complexity is O(n) and for stack to hold atmost n literals the space complexity is O(n), hence  
  \* Total space complexity is O(n+n) = O(2n) i.e : O(n)

# 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.



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.

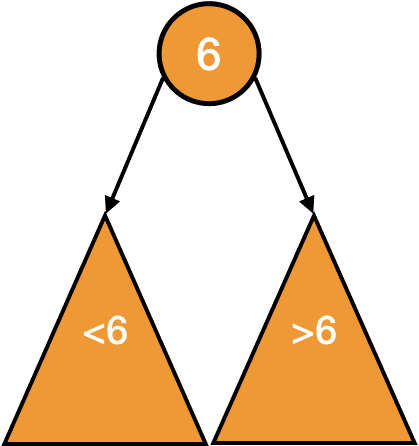


As the name suggests, binary search tree is usually used to perform an optimized search. So, let's look at the searching process of a BST.

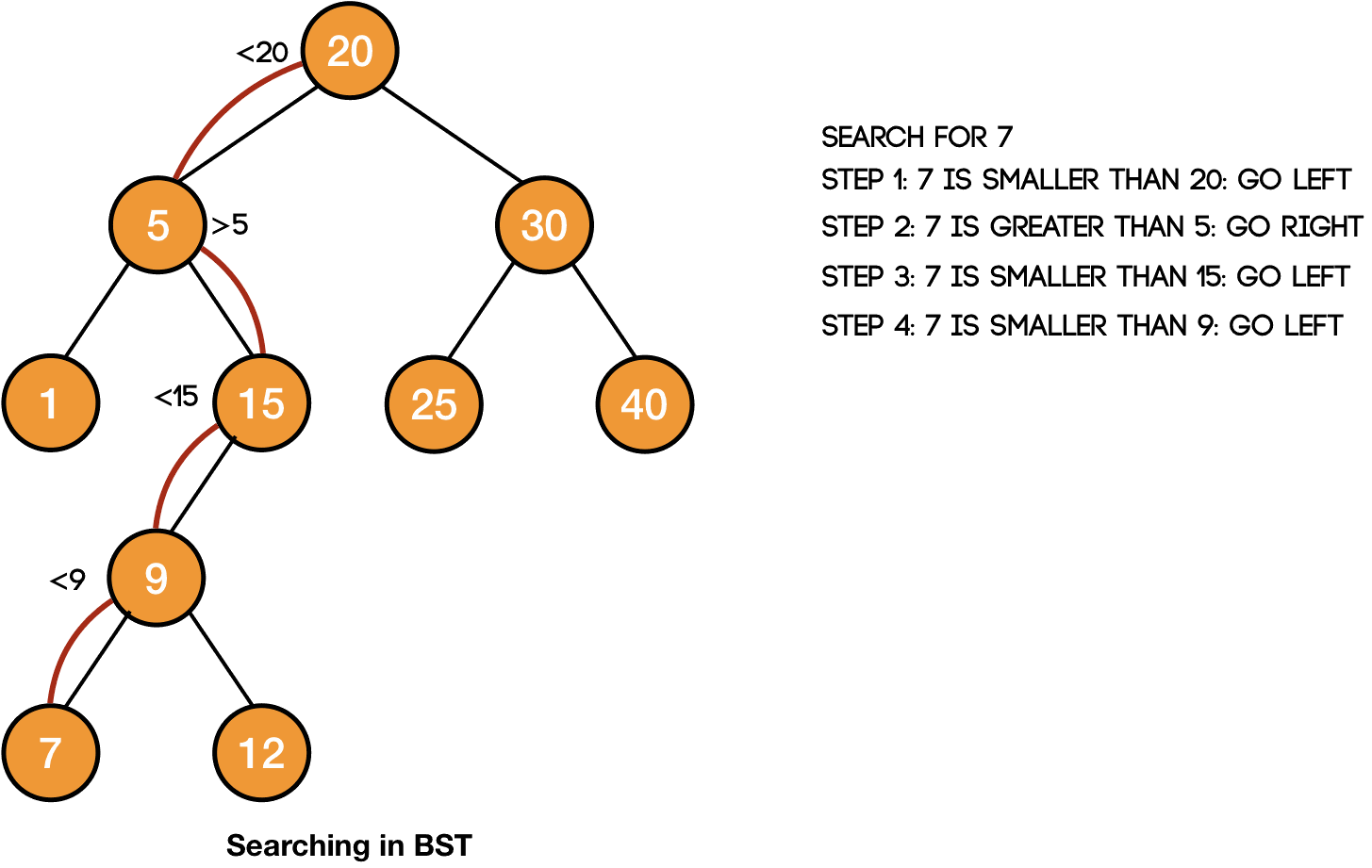
## Searching a BST

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.



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**

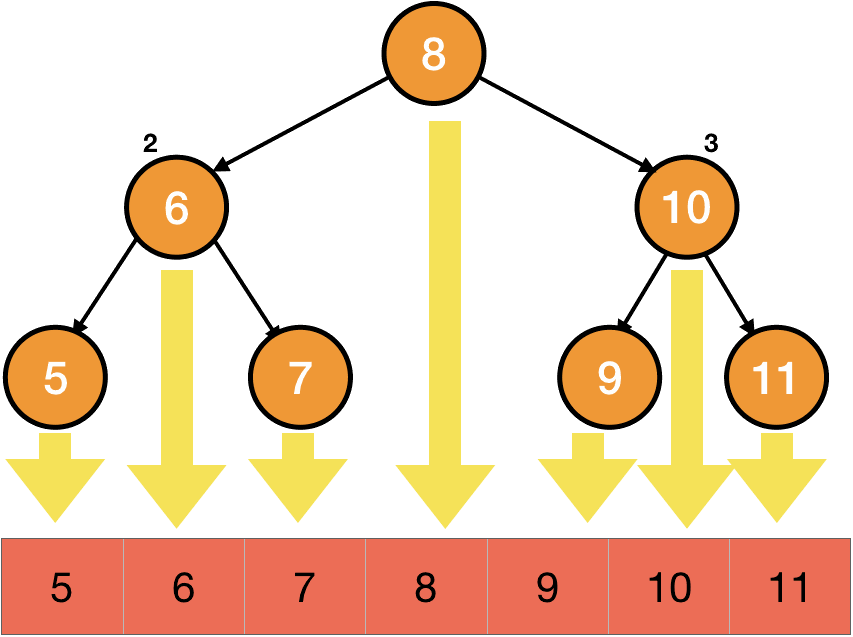
**else if(T.root.data > x)**

**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.



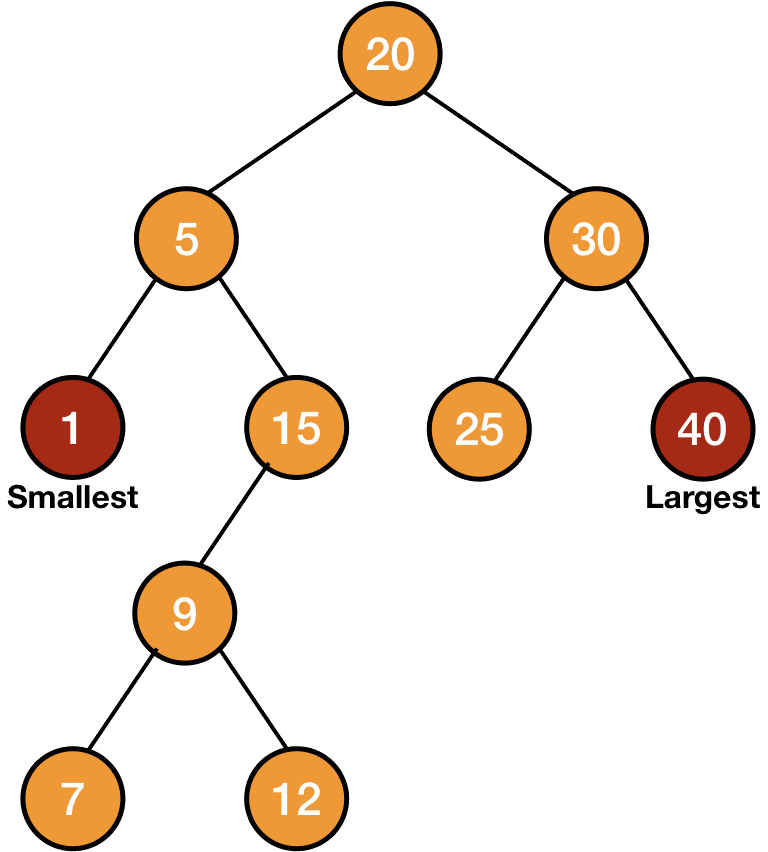
To search an element in the tree, we are taking a simple path from the root to leaf. Thus, searching in a binary search tree is done O(h)

time.

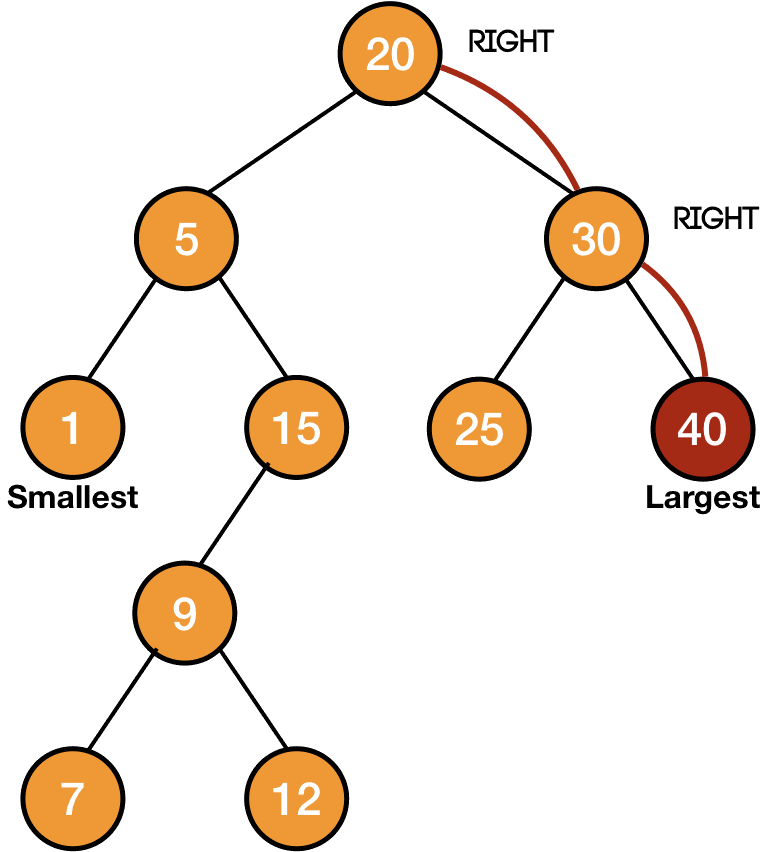
We also get the maximum and the minimum element of a BST using MAXIMUM and MINIMUM operations. Let's have a look at these.

## 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)

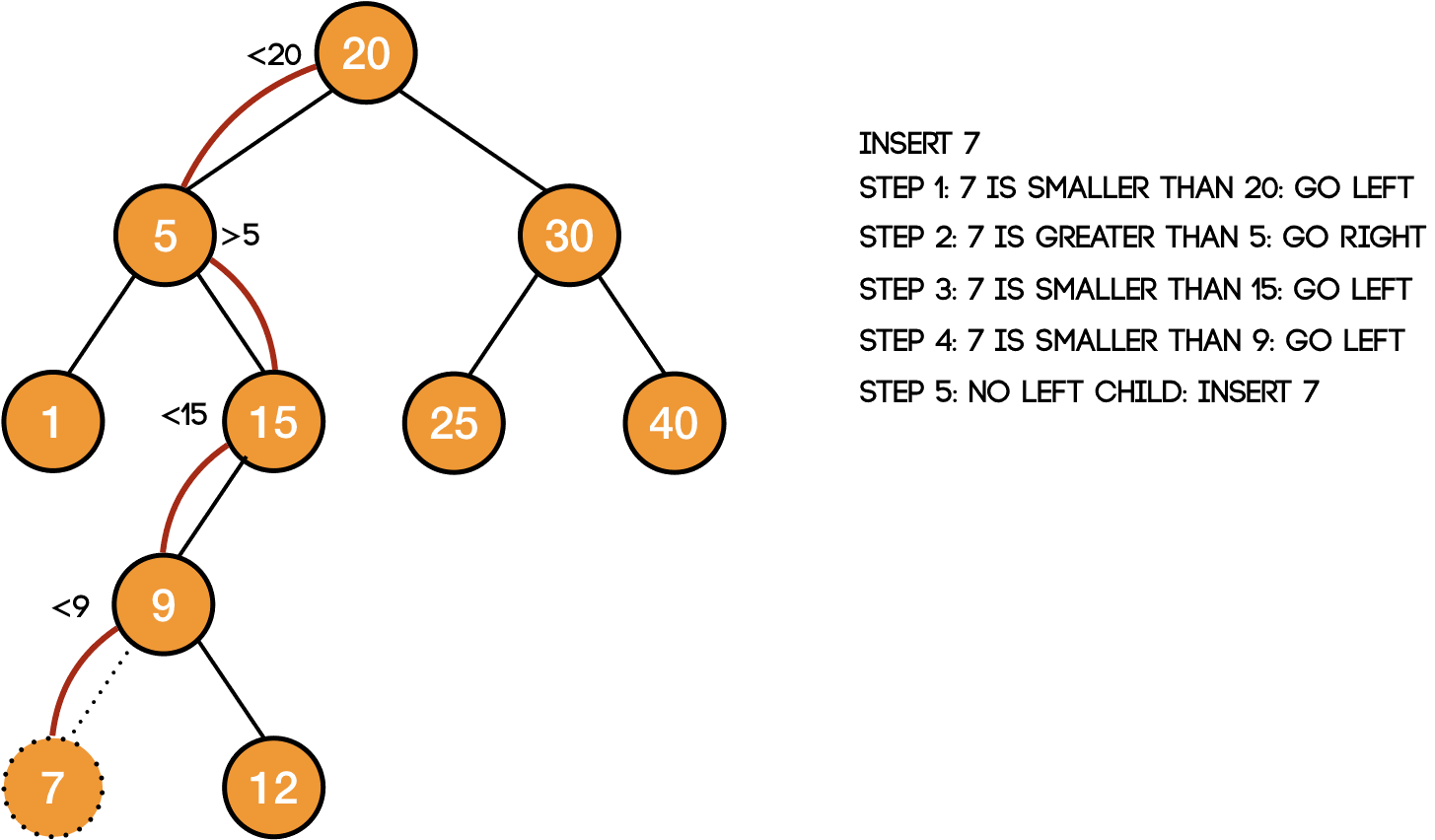
operations.

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.

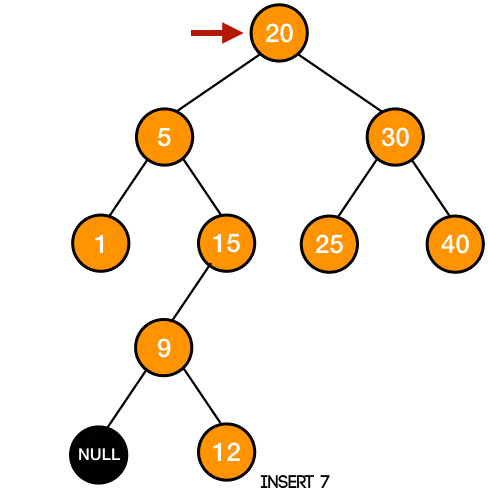
## Insertion in BST

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.

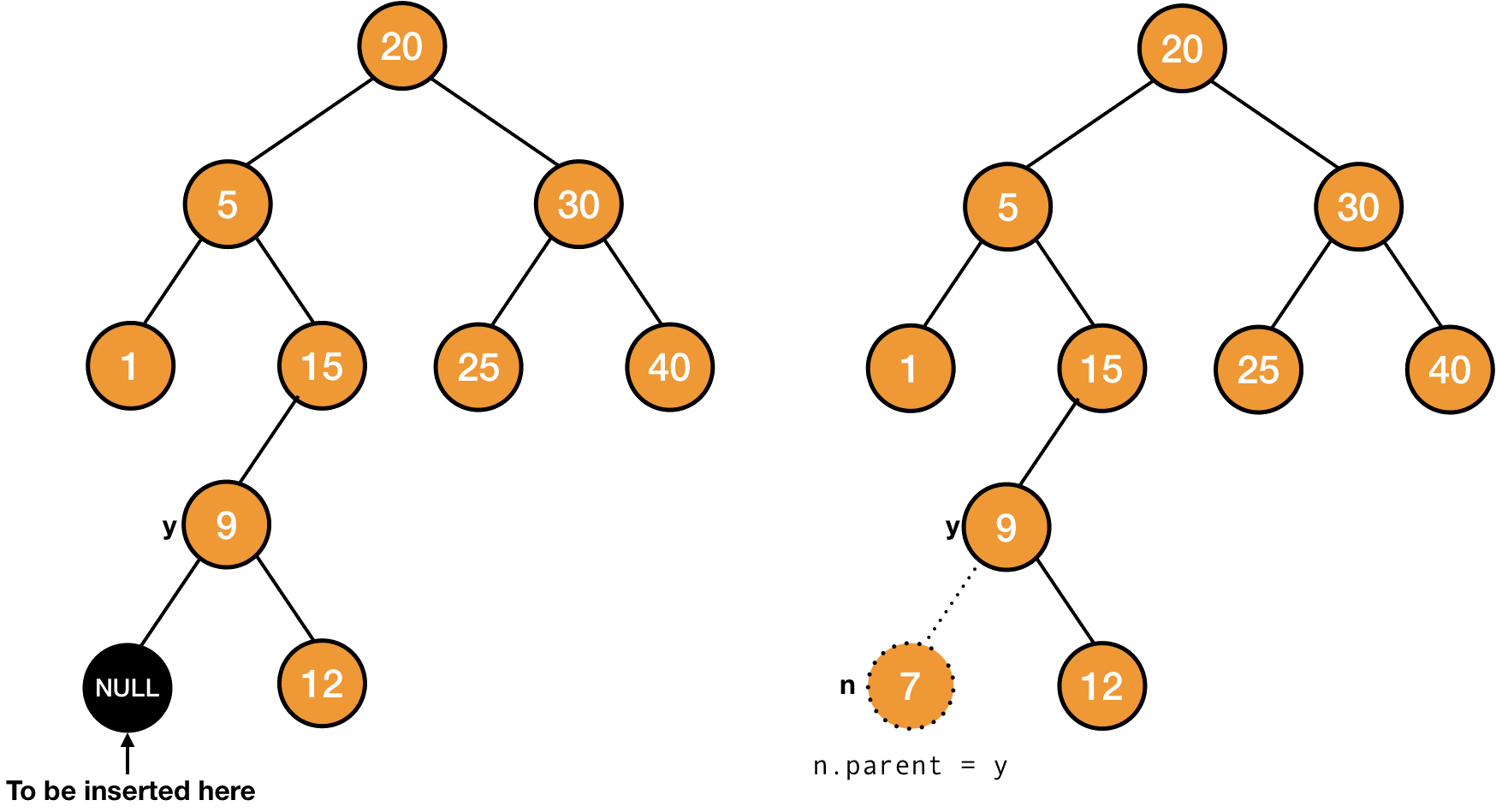


INSERT(T, n)  
    temp = T.root  
    while temp != NULL  
        if n.data < temp.data  
            temp = temp.left  
        else  
            temp = temp.right

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.

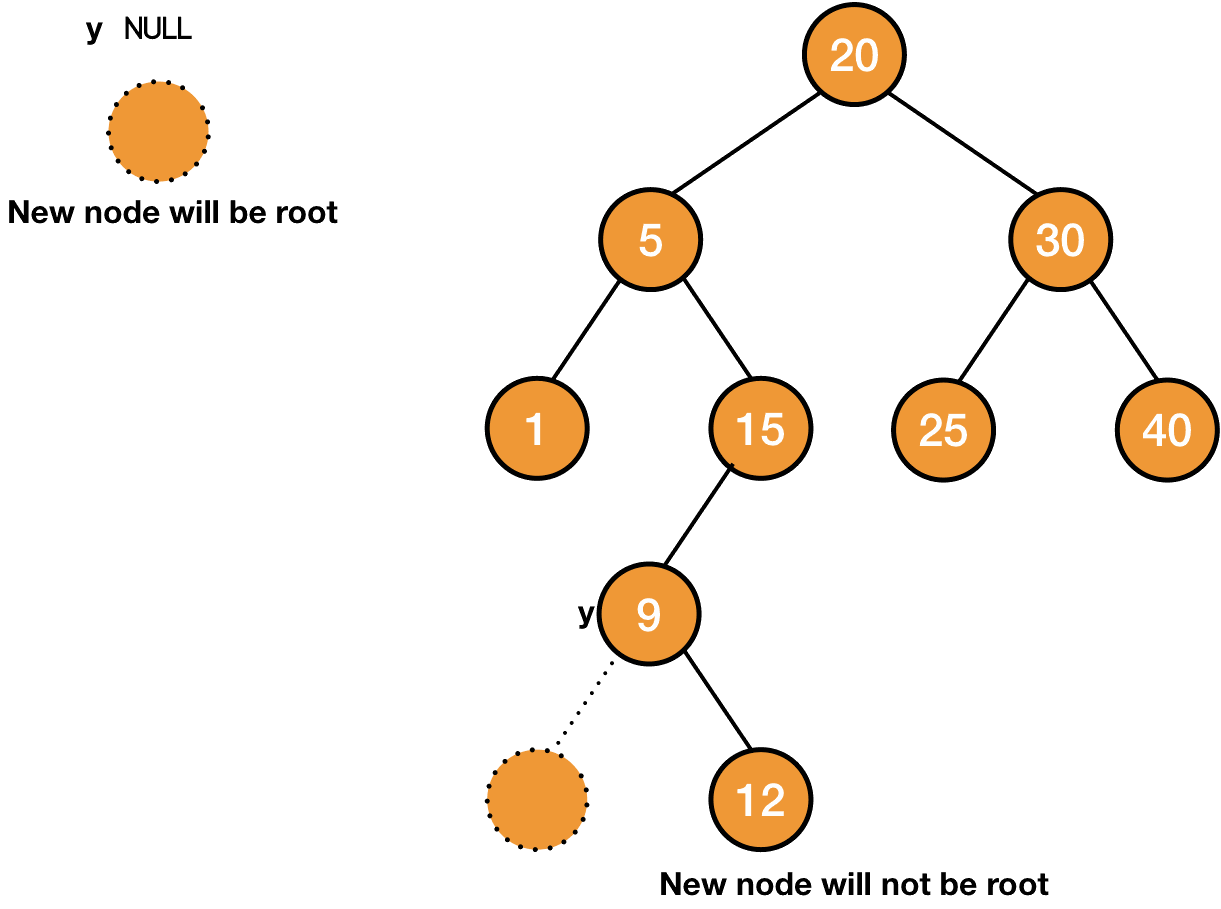
We need to make the last node in the above iteration the parent of the new node. So, let's use a variable for this.



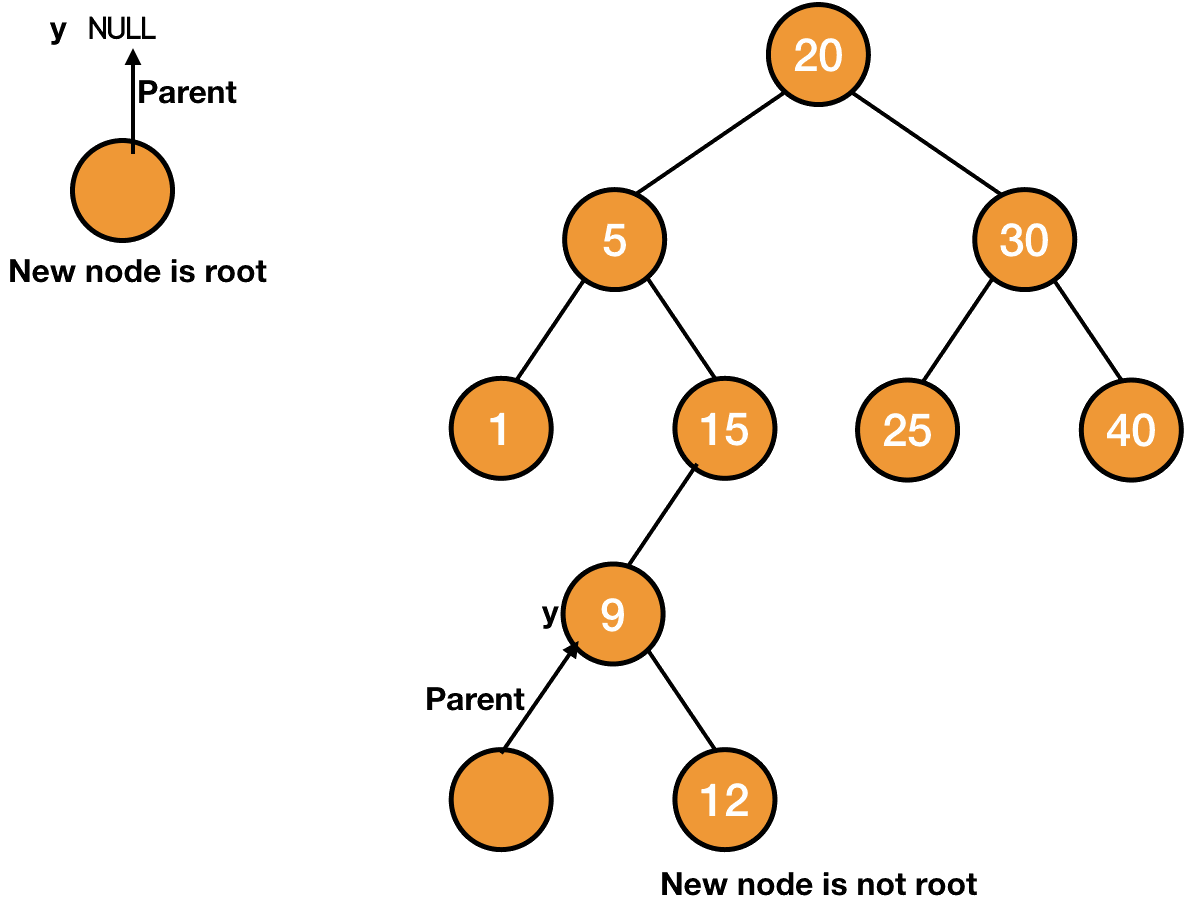
INSERT(T, n)  
    temp = T.root  
    y = NULL  
    while temp != NULL  
        y = temp  
        if n.data < temp.data  
            temp = temp.left  
        else  
            temp = temp.right

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.

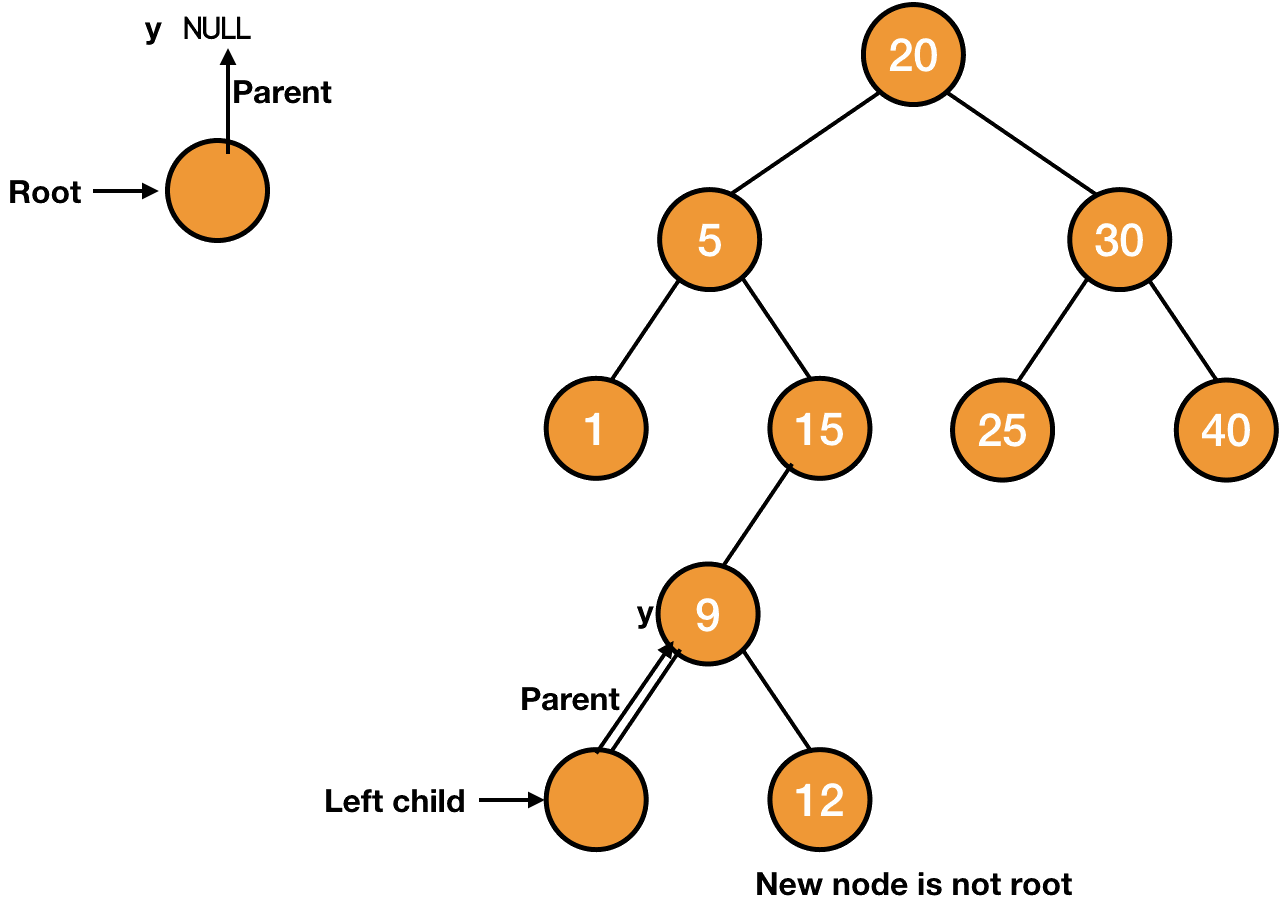


After this, we will make *y* the parent of the new node.



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**

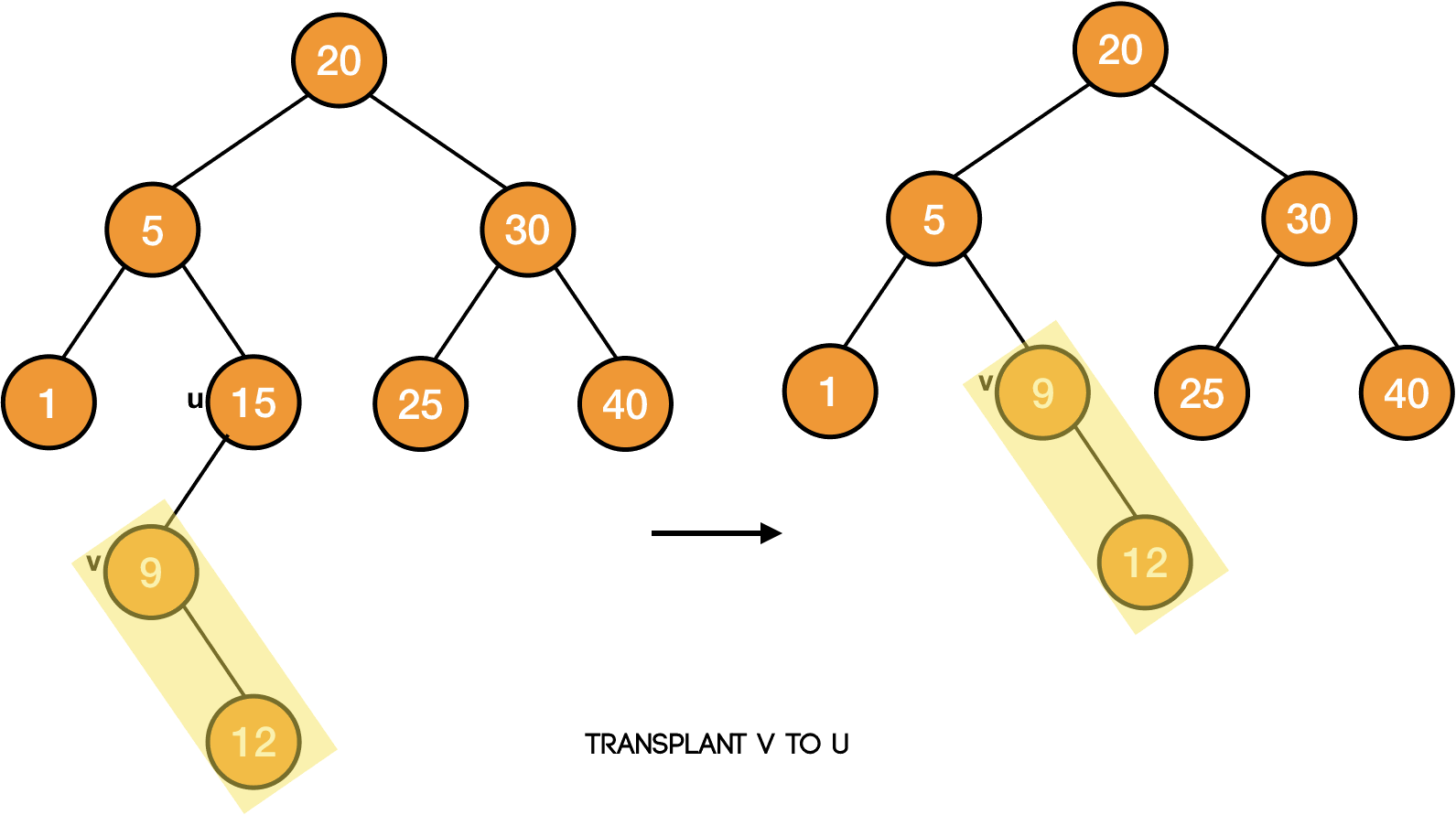
**else**

**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.

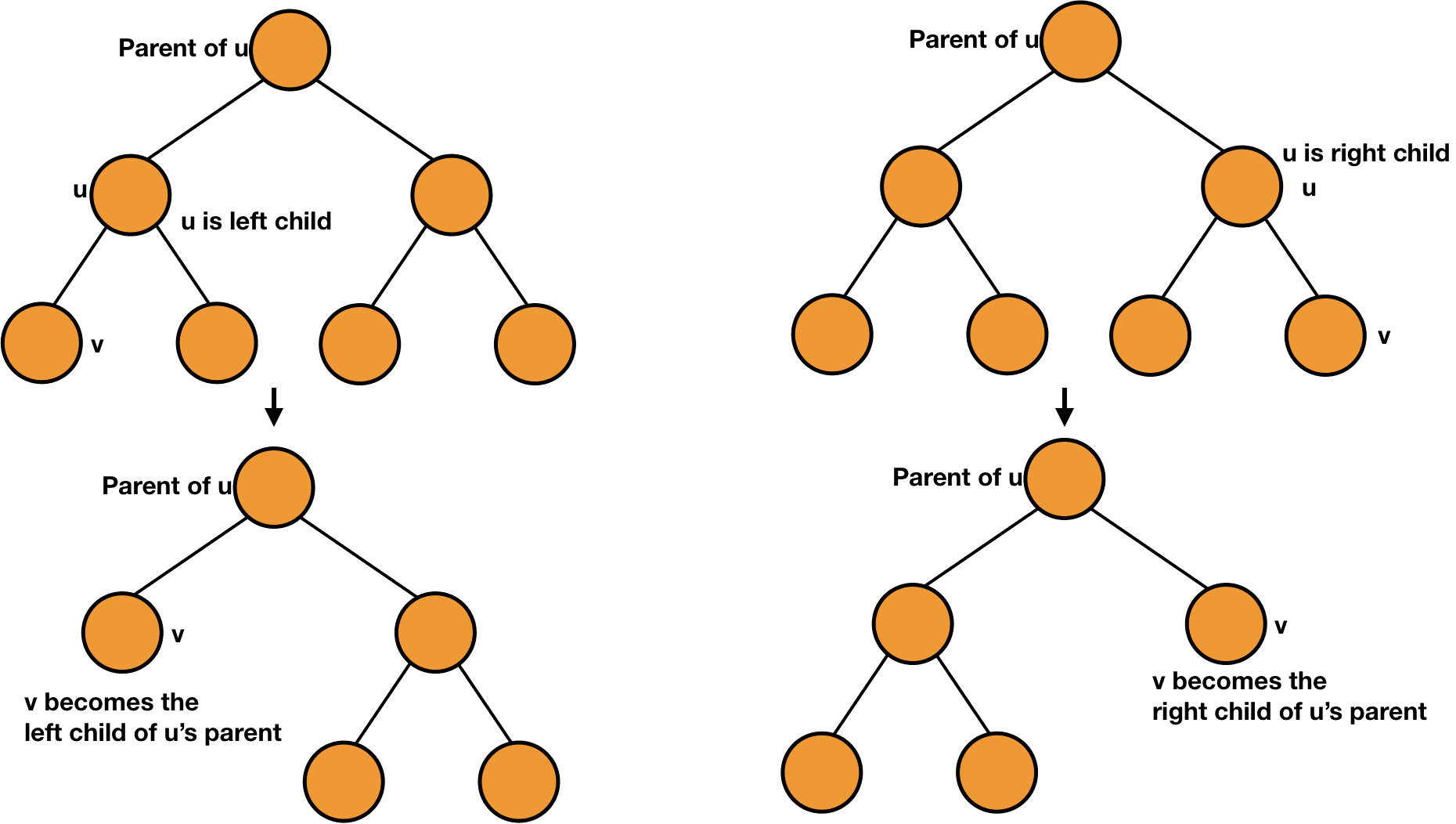
To delete a node from a BST, we will replace a subtree with another one i.e., we transplant one subtree in place of another.



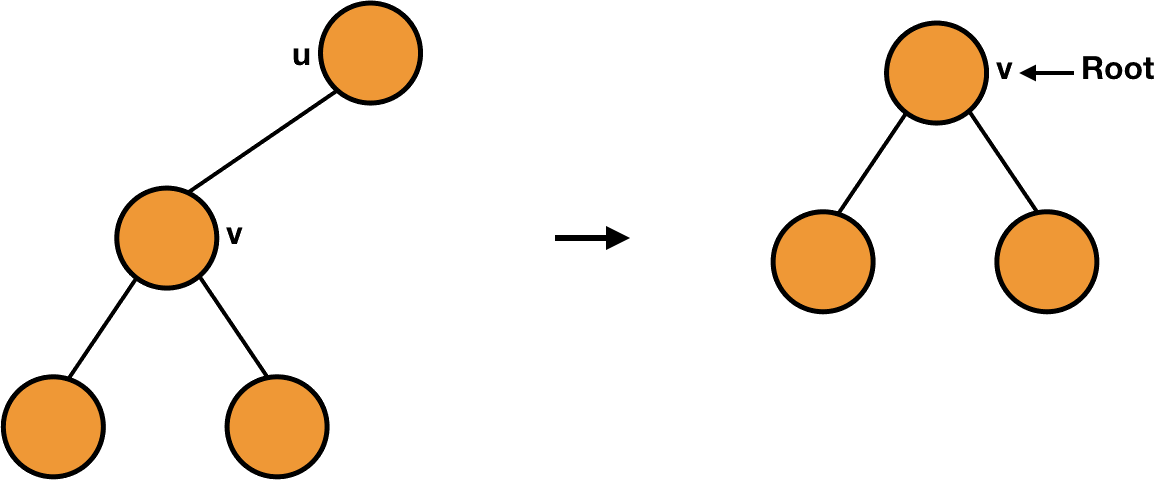
As we are going to use this technique in our delete procedure, so let's first write the code to transplant a subtree rooted at node *v* in place of the subtree rooted at node *u*.

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**

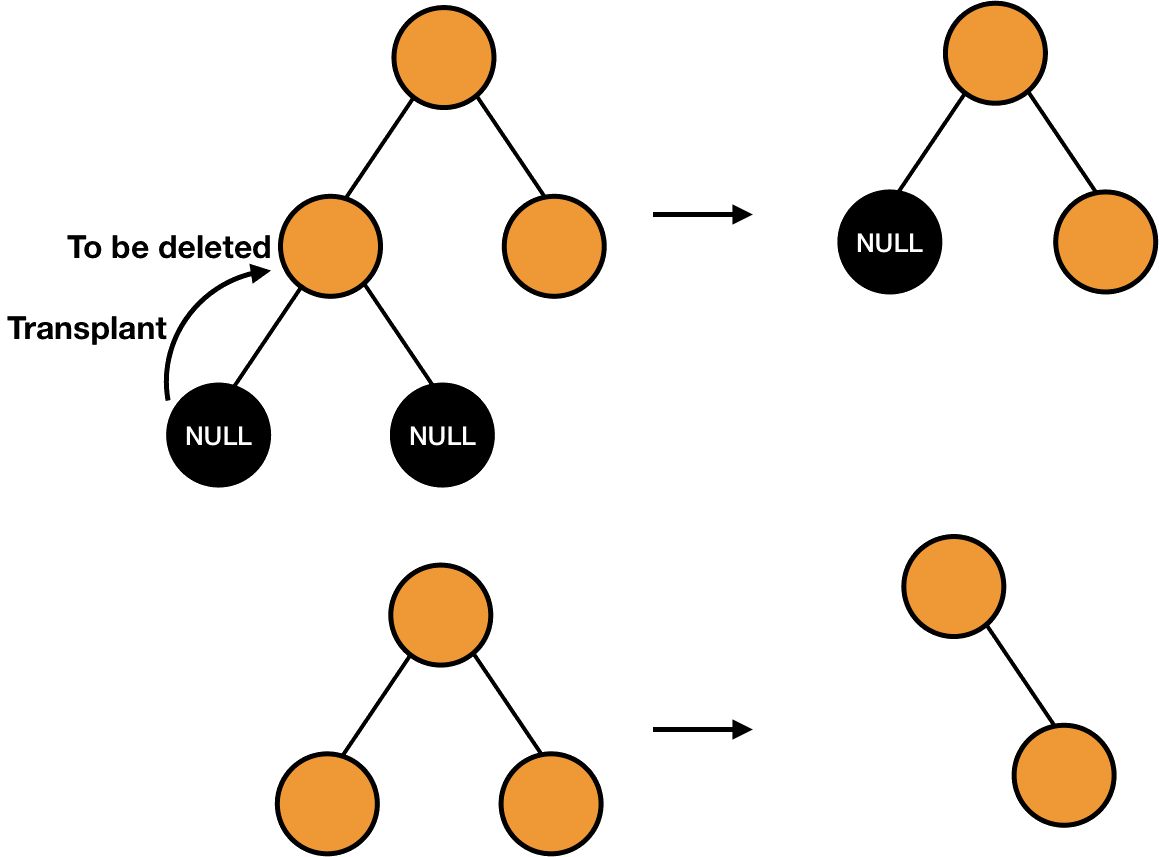
**u.parent.right = v**

**if v != NULL**

**v.parent = u.parent**

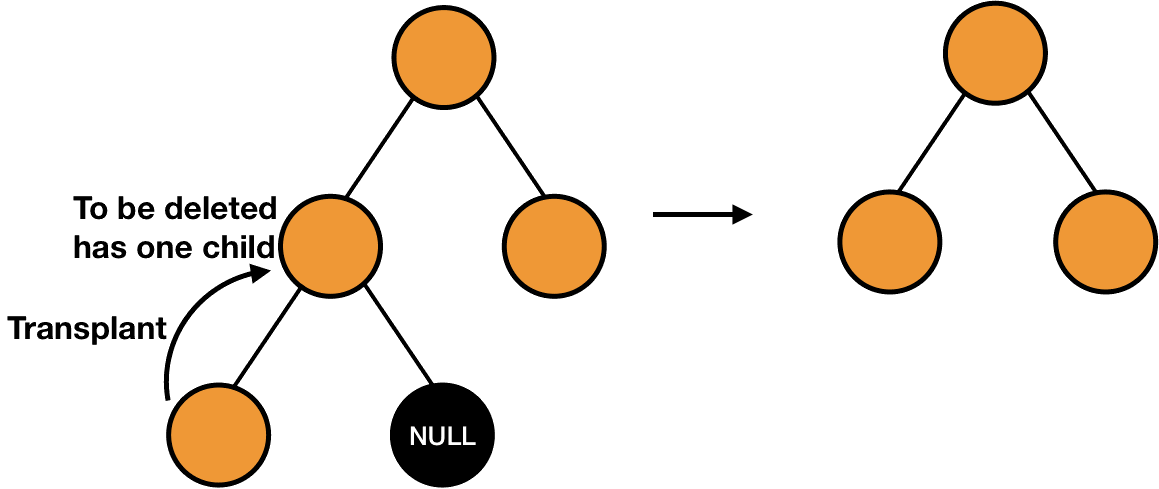
Let's focus on the deletion of a node from a binary search tree.

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

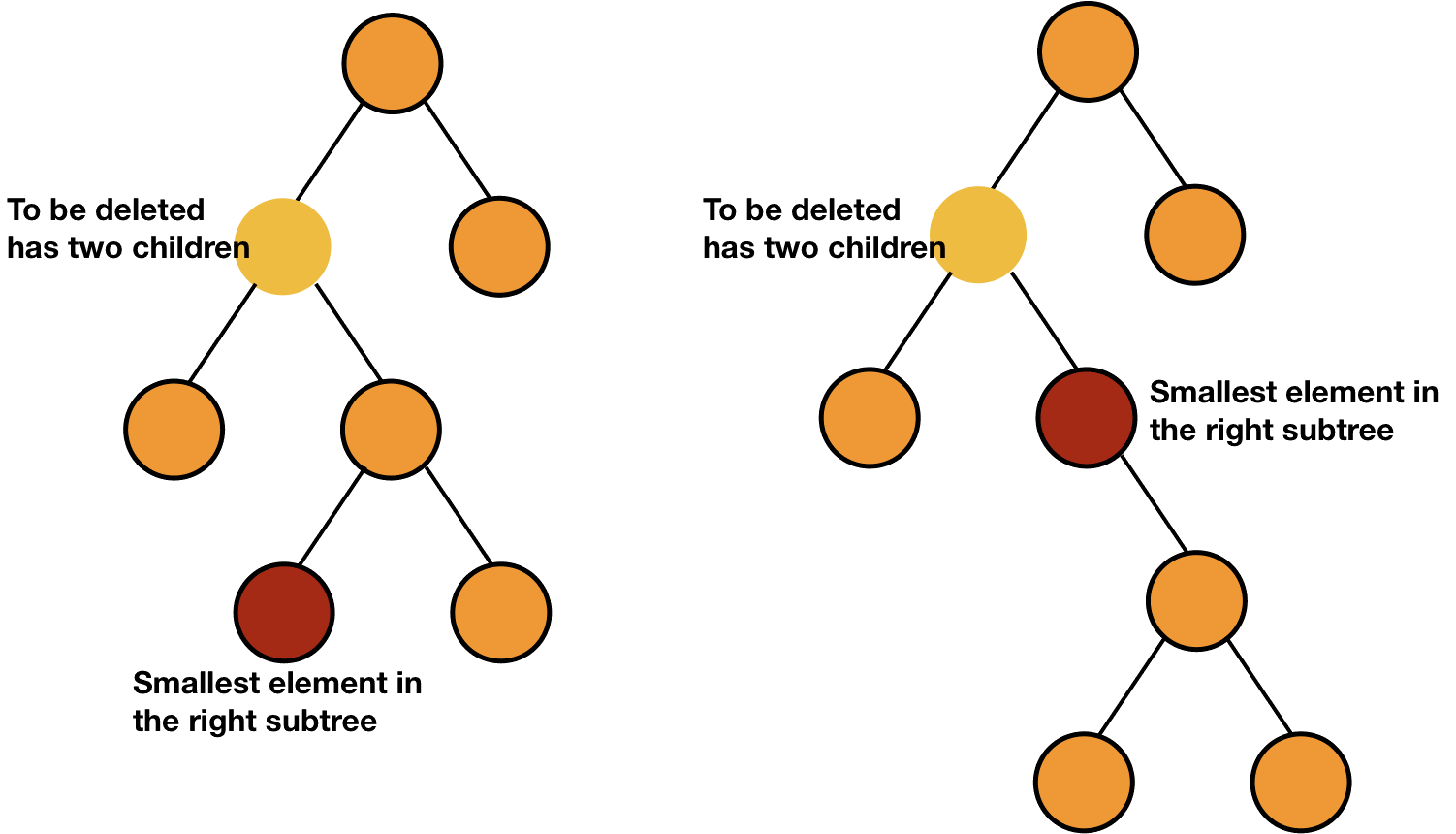


We can also say that we are transplanting the right or the left child (both are NULL) to the node to be deleted.

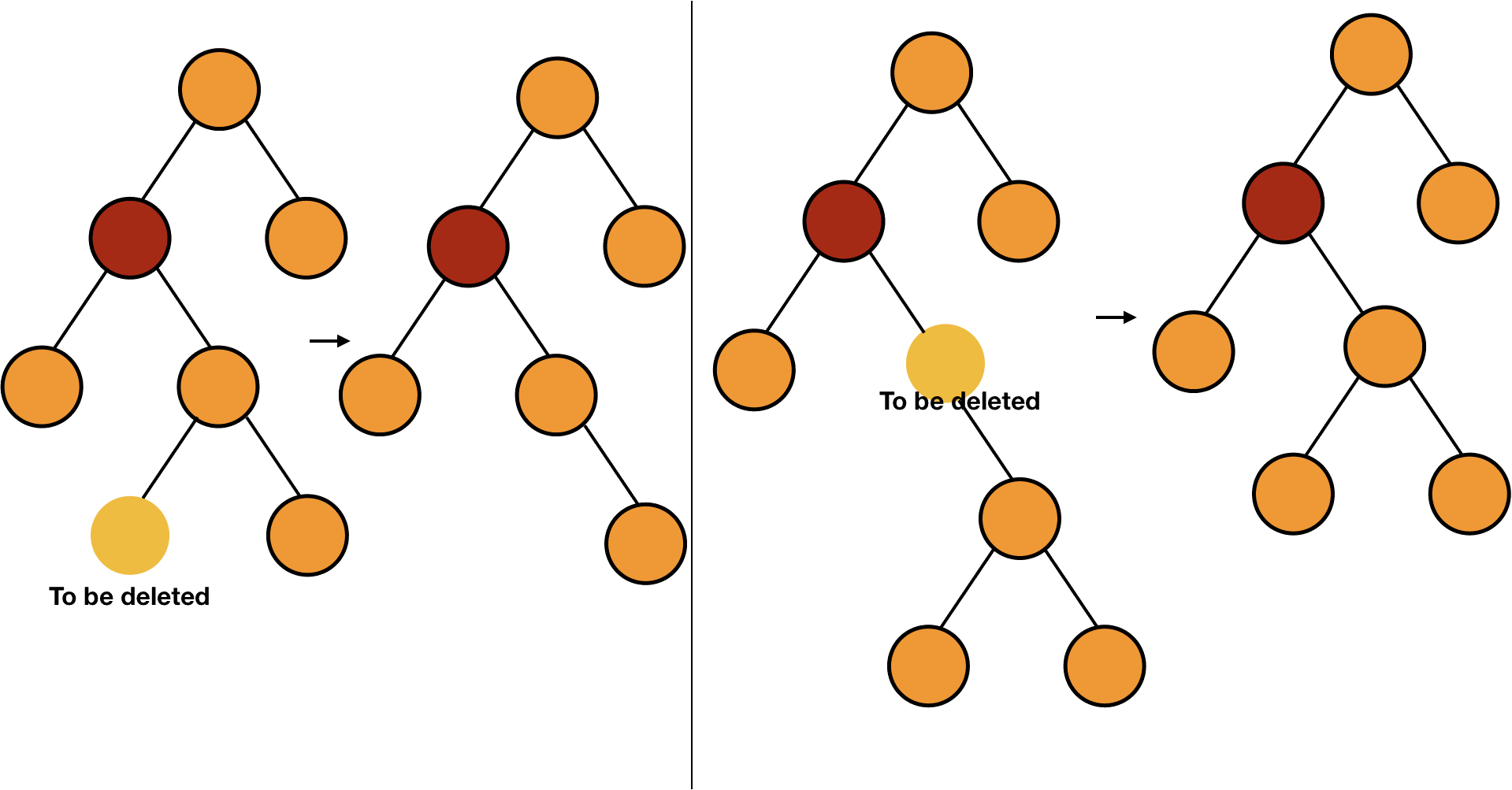
We can also delete a node with only one child by transplanting its child to the node and it will not affect the property of the binary search tree.



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

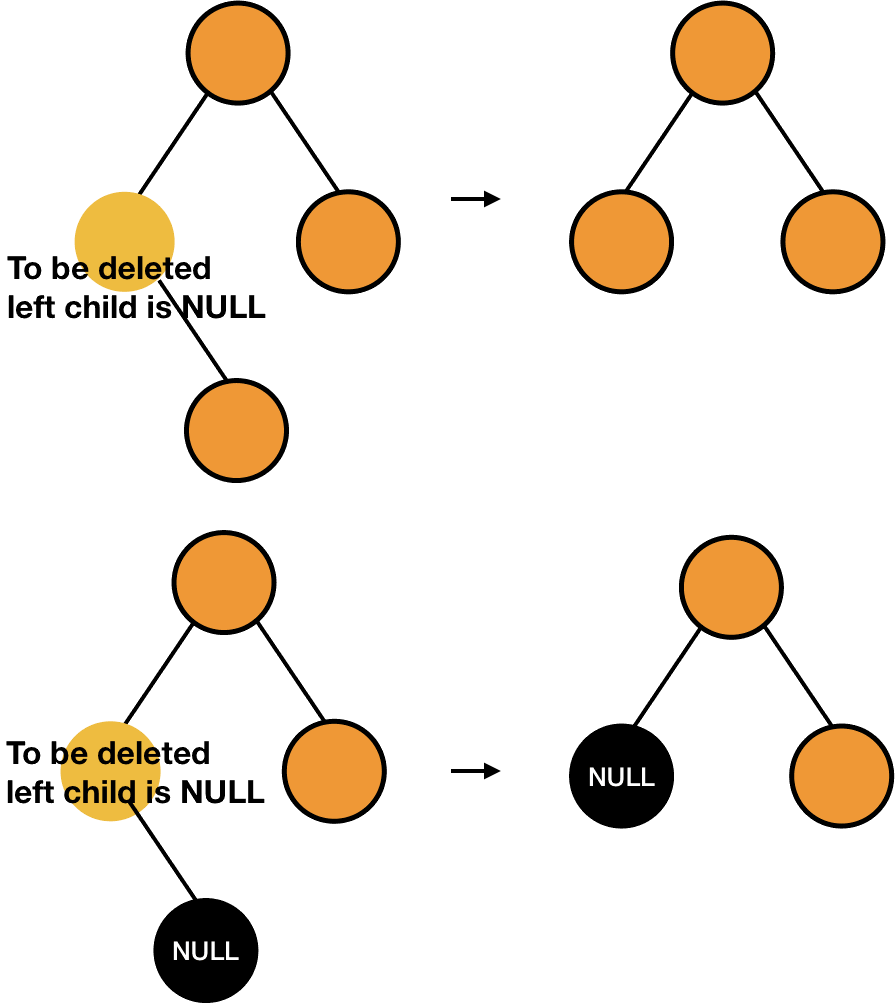


In this case, we can find the smallest element of the right subtree of the node to be deleted (element with no left child in the right subtree) and replace its content with the node to be deleted.



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).

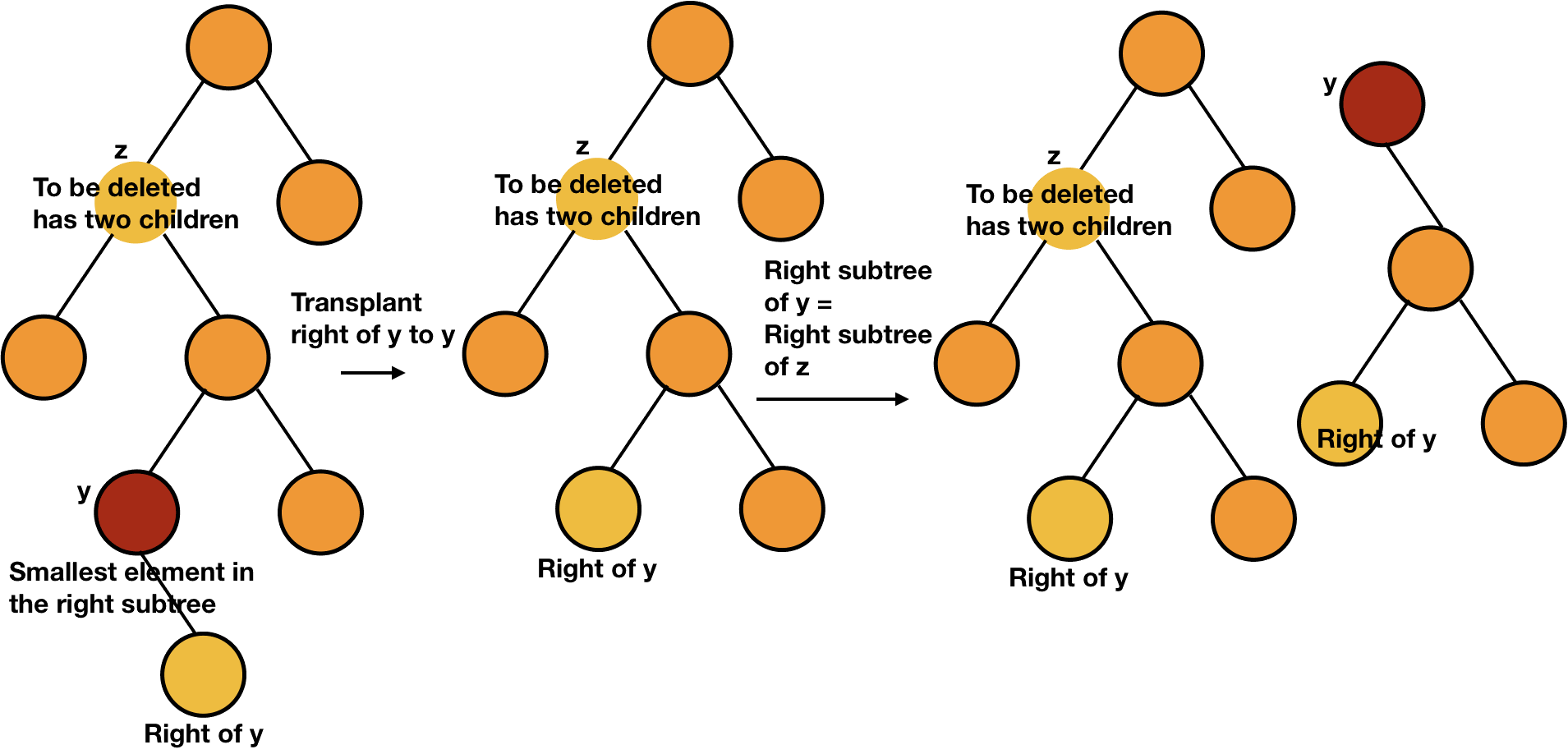
Now, we have to check the number of children of the node *z*. We will first check if the left child of the node *z* is NULL or not. If the left child is NULL, then either it has only one child (right one) or none. In both the cases, we can transplant its right child to it.

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

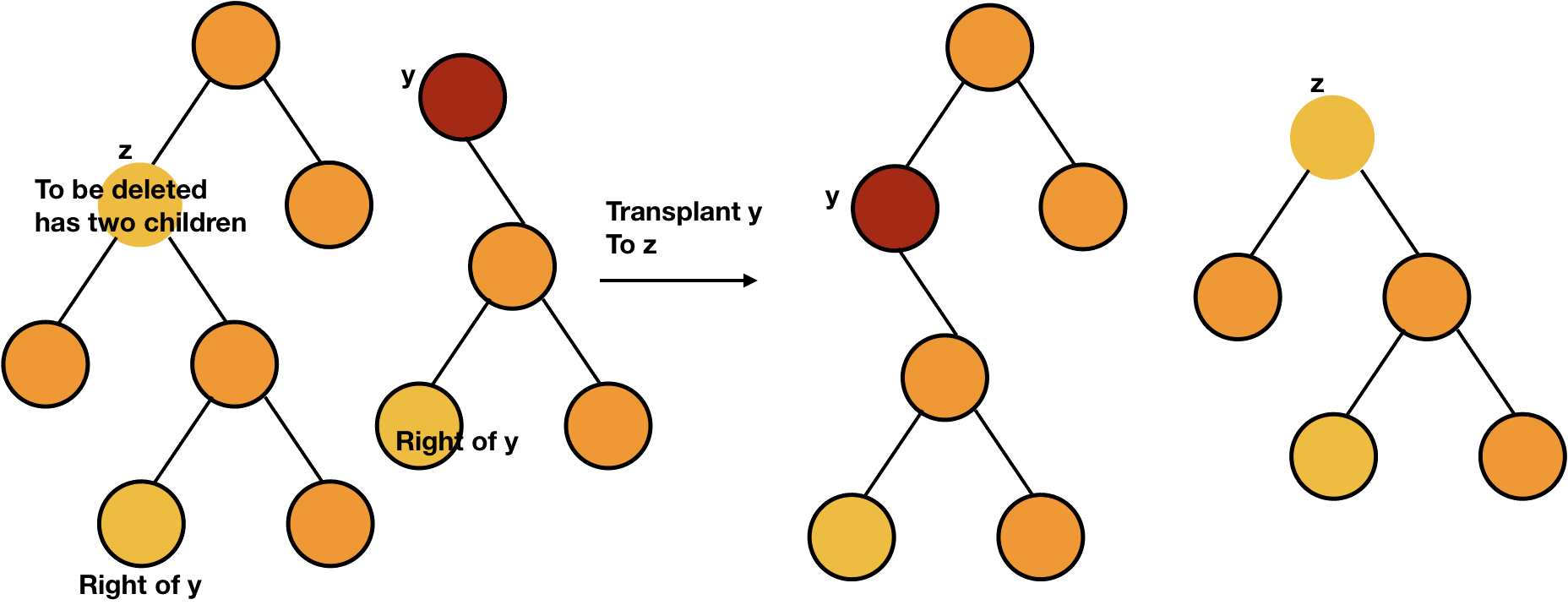
Similarly, we will next check if the right child is NULL or not.

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

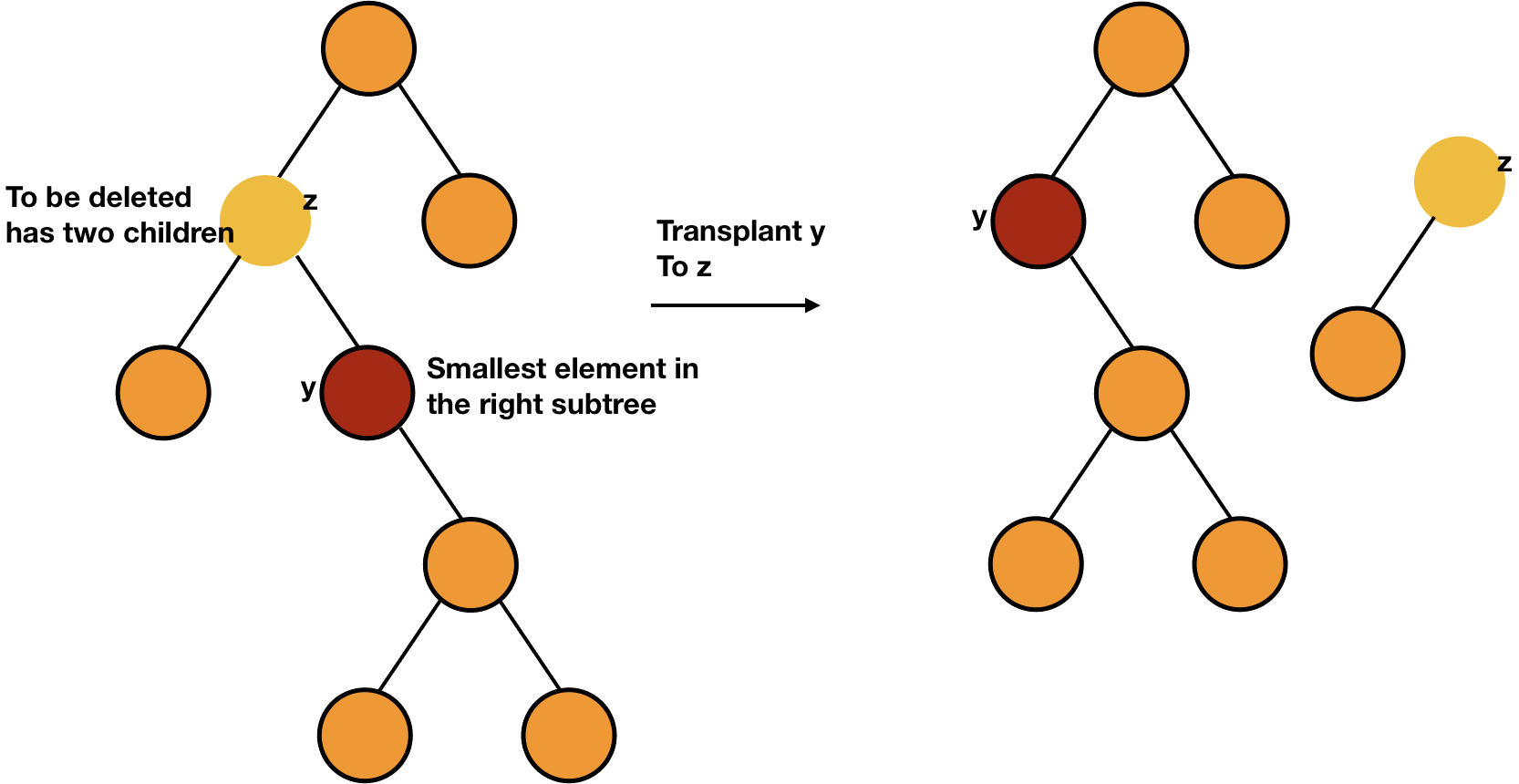
If none of the above cases are true, the node *z* has both children and we will find the minimum in the right subtree (*y*). Now, we have to put this minimum node (*y*) in the place of *z*. Firstly, we will transplant the right of *y* to *y* and then take the right subtree of *z* and make it the right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



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

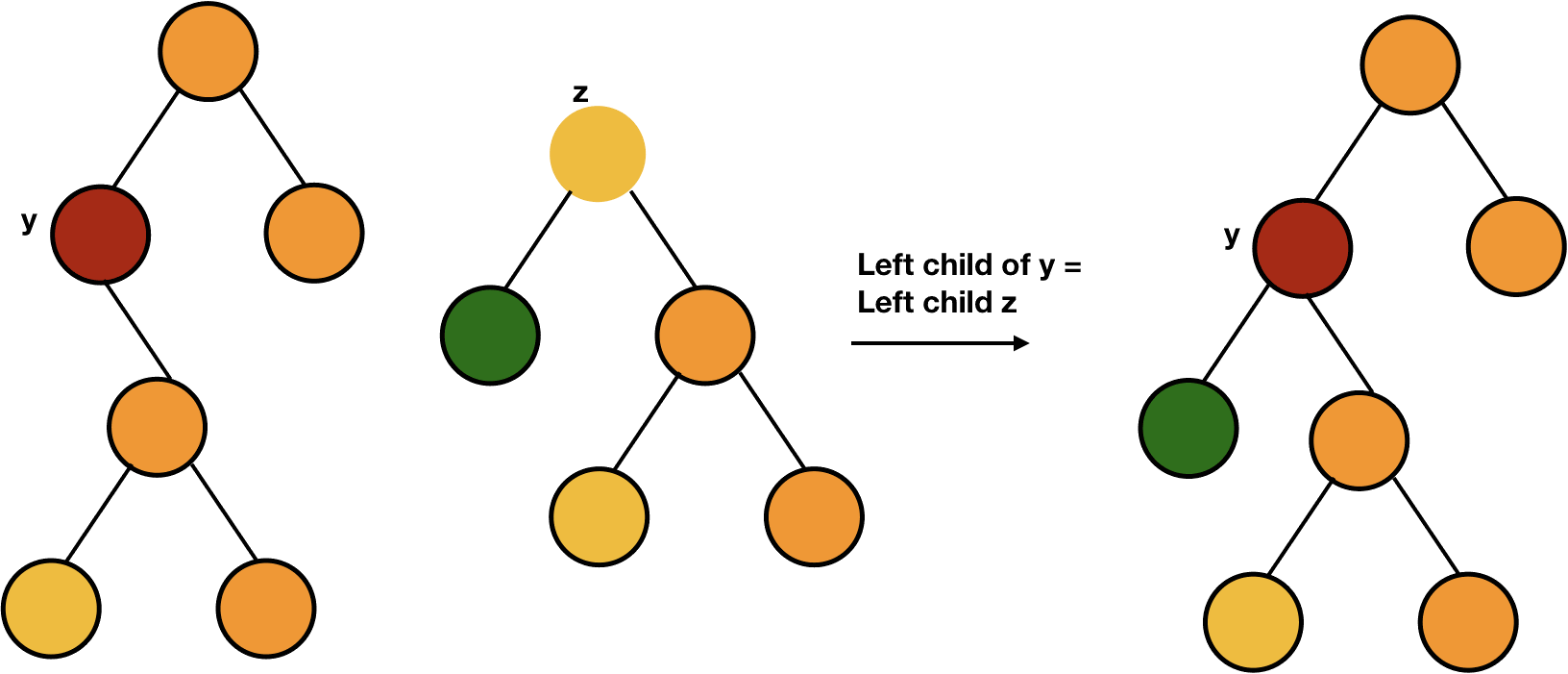


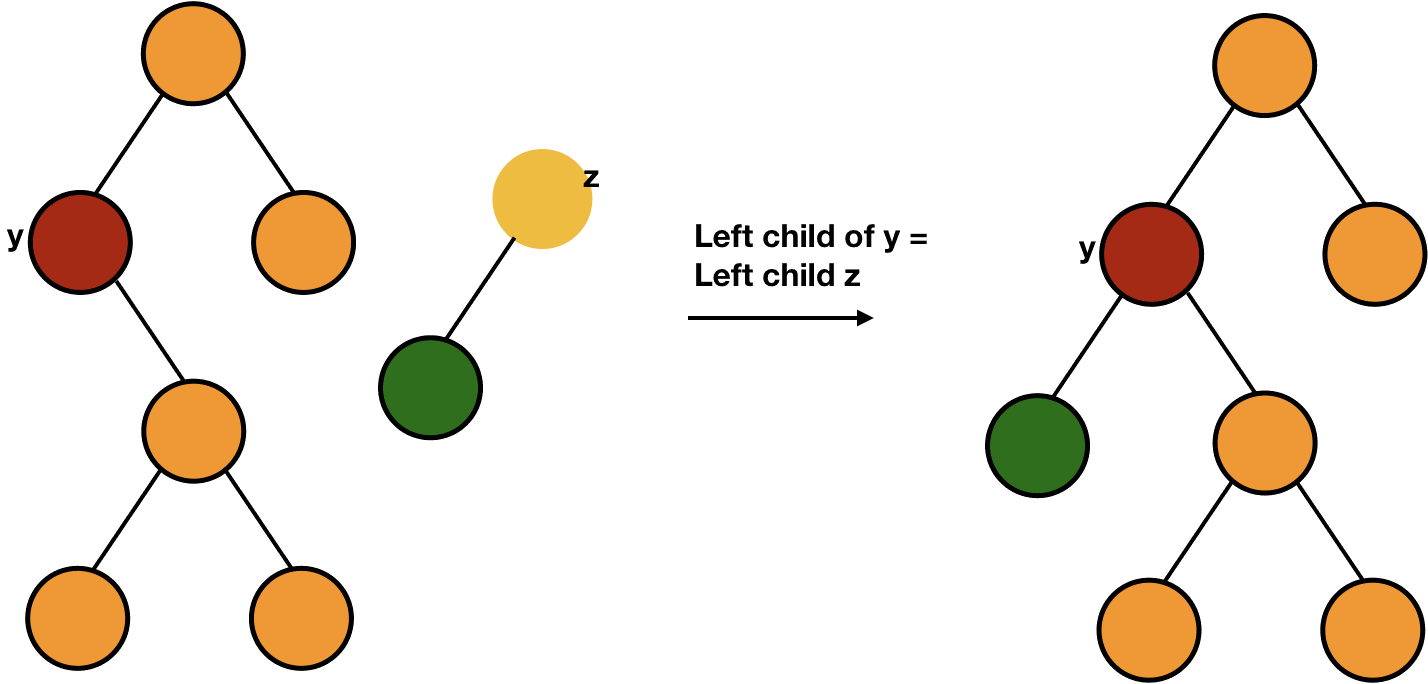
However, it can also be possible that the minimum node is the direct child of the node *z*. In that case, we will just transplant *y* to *z*.



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

#include <stdio.h>

#include <stdlib.h>

typedef struct node {

int data;

struct node \*left;

struct node \*right;

struct node \*parent;

}node;

typedef struct binary\_search\_tree {

node \*root;

}binary\_search\_tree;

node\* new\_node(int data) {

node \*n = malloc(sizeof(node));

n->data = data;

n->left = NULL;

n->right = NULL;

n->parent = NULL;

return n;

}

binary\_search\_tree\* new\_binary\_search\_tree() {

binary\_search\_tree \*t = malloc(sizeof(binary\_search\_tree));

t->root = NULL;

return t;

}

node\* minimum(binary\_search\_tree \*t, node \*x) {

while(x->left != NULL)

x = x->left;

return x;

}

void insert(binary\_search\_tree \*t, node \*n) {

node \*y = NULL;

node \*temp = t->root;

while(temp != NULL) {

y = temp;

if(n->data < temp->data)

temp = temp->left;

else

temp = temp->right;

}

n->parent = y;

if(y == NULL) //newly added node is root

t->root = n;

else if(n->data < y->data)

y->left = n;

else

y->right = n;

}

void transplant(binary\_search\_tree \*t, node \*u, node \*v) {

if(u->parent == NULL) //u is root

t->root = v;

else if(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;

}

void delete(binary\_search\_tree \*t, node \*z) {

if(z->left == NULL) {

transplant(t, z, z->right);

free(z);

}

else if(z->right == NULL) {

transplant(t, z, z->left);

free(z);

}

else {

node \*y = minimum(t, z->right); //minimum element in right subtree

if(y->parent != z) {

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;

free(z);

}

}

void inorder(binary\_search\_tree \*t, node \*n) {

if(n != NULL) {

inorder(t, n->left);

printf("%d\n", n->data);

inorder(t, n->right);

}

}

int main() {

binary\_search\_tree \*t = new\_binary\_search\_tree();

node \*a, \*b, \*c, \*d, \*e, \*f, \*g, \*h, \*i, \*j, \*k, \*l, \*m;

a = new\_node(10);

b = new\_node(20);

c = new\_node(30);

d = new\_node(100);

e = new\_node(90);

f = new\_node(40);

g = new\_node(50);

h = new\_node(60);

i = new\_node(70);

j = new\_node(80);

k = new\_node(150);

l = new\_node(110);

m = new\_node(120);

insert(t, a);

insert(t, b);

insert(t, c);

insert(t, d);

insert(t, e);

insert(t, f);

insert(t, g);

insert(t, h);

insert(t, i);

insert(t, j);

insert(t, k);

insert(t, l);

insert(t, m);

delete(t, a);

delete(t, m);

inorder(t, t->root);

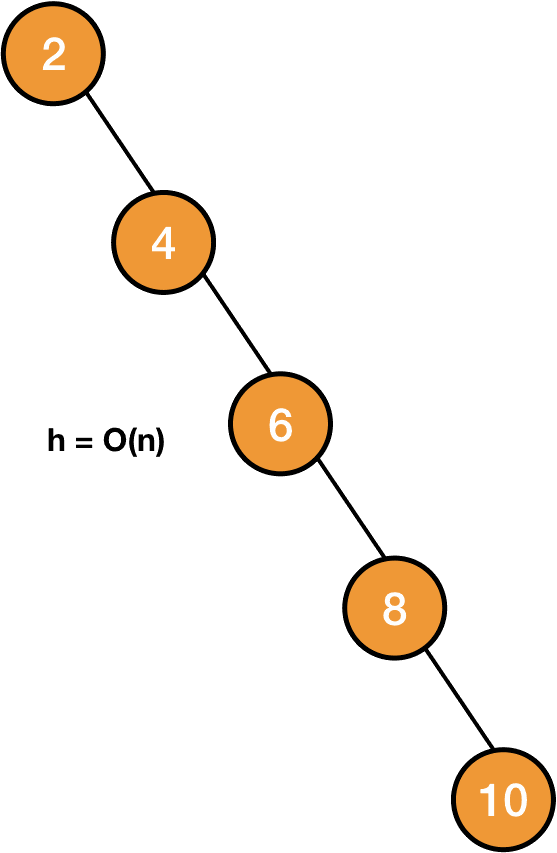
return 0;

}

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.