[**Computer science course notes**](http://www.microveggies.com/csci)

# Home

## [Lecture15 - Trees](http://www.microveggies.com/csci/index.php/csci-2270-lecture-notes/3-lecture15-trees)

**Details**

Written by Rhonda Hoenigman

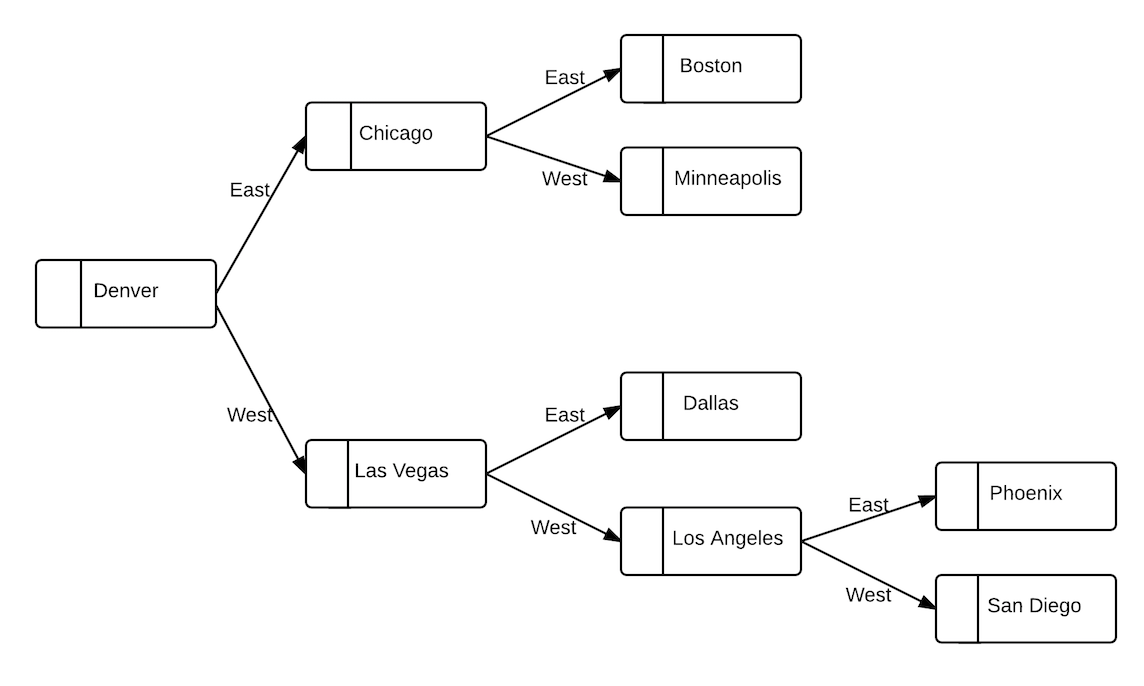
 Published: 15 February 2015

 Hits: 943

When we were looking at linked lists, we had a pointer to the next and previous nodes in a list.



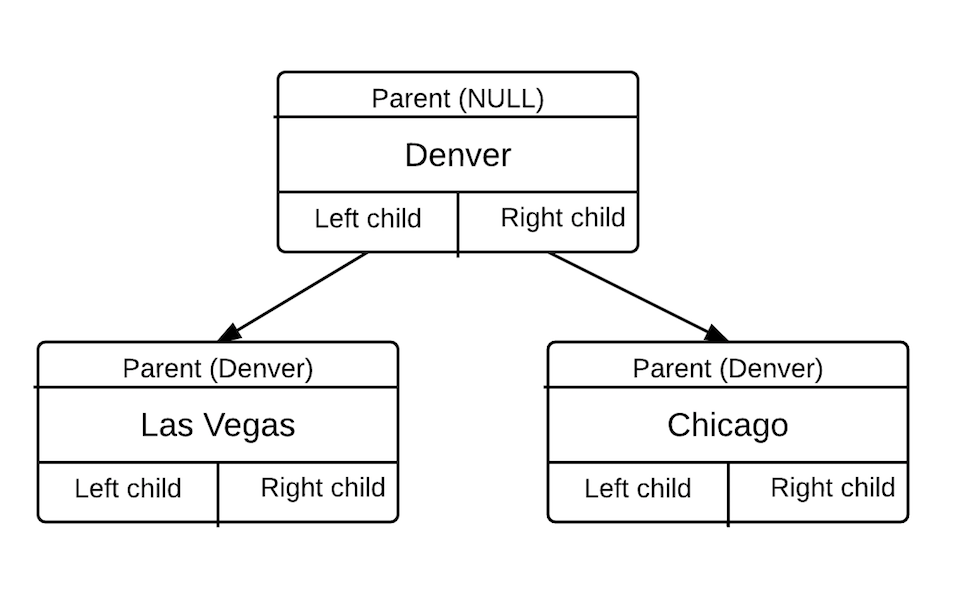
A simple linked list doesn't handle the situation where you need to have multiple next pointers from any given node. For example, imagine you are in Denver and you want to deliver a message to another city somewhere in the U.S. From Denver, you can go east to Chicago or west to Las Vegas. From Chicago, you can go east to Boston or west to Minneapolis. You could represent the network of cities by including multiple next locations from your current city:



Traveling between Denver and any city in the network involves making a right or left turn at each node until you arrive at your desired location.

## Binary Trees

Having two options (at most) for going to a next node from the current node is a feature of a structure called a binary tree. We can re-frame the city network we've been looking at and view it as a tree by putting the starting city at the top, also known as the root, and putting the two possible destinations from the root as the children. We get something that looks like this:



In this example, each node has a name that identifies it, a parent node, and a left and a right child node. The root is Denver, and since it is the root, it doesn't have a parent. The left child of the root is Last Vegas, and the right child of the root is Chicago. If we compare these tree nodes to linked list nodes we see that we've added an extra pointer. Otherwise, the concept is very similar.

Pointers in a doubly linked list:

* next
* previous

Pointers in a binary tree:

* parent
* left child
* right child

### Some properties of Binary Trees

1. Each node in the tree is an object, which we'll identify as x.
2. Each node has a key that identifies it. In the city example, the key is the city name.
3. Each node has a parent, which we'll identify as P.
4. Each parent can have at most two children.
5. If P==NULL, then the node is the root of the tree, it doesn't have a parent.
6. If node x has no left child, then x.leftChild == NULL.
7. If node x has no right child, then x.rightChild == NULL.

The pointers to the parent and the children establish the root and bottom of the tree. If we are searching the tree and want to know if we are at the root, we can check if P==NULL. This is the only way to know if we've reached the root because there's nothing necessarily intrinsic in the data to identify the root. The same is true for the left and right child pointers. Checking that these are NULL is the only way for us to know that there are no additional children in the tree. There's nothing else necessarily in the tree to identify the bottom for any given branch.

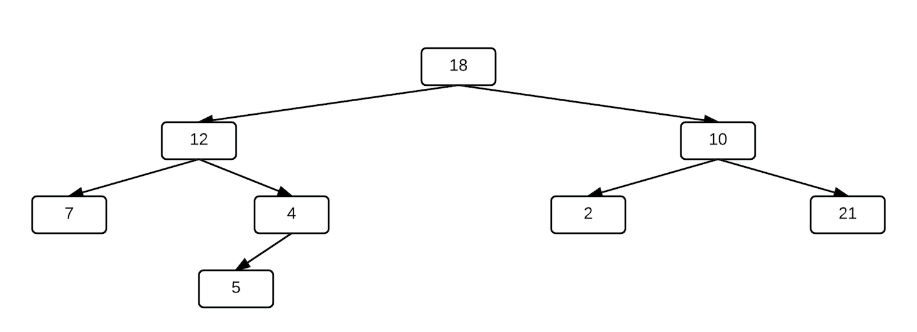
**Exercise:**

Build a binary tree from the following input data:

|  |  |  |  |
| --- | --- | --- | --- |
| Parent Key | Key | Left Child Key | Right Child Key |
| 18 | 12 | 7 | 4 |
| NULL | 18 | 12 | 10 |
| 12 | 7 | NULL | NULL |
| 12 | 4 | 5 | NULL |
| 18 | 10 | 2 | 21 |
| 10 | 2 | NULL | NULL |
| 10 | 21 | NULL | NULL |

 The root node in the tree is the one where Parent Key = NULL, which is the node with Key = 18. From there, we assign Keys 12 and 10 as the left and right children, respectively. For the node with Key=12, the left child is 7 and the right child is 4. Checking for the node with Key = 7, this node has no children as evidenced by the left and right child keys being NULL.

**Solution:**



**Binary tree node in C/C++**

A binary tree node in C/C++ can be built with struct, where the members of the struct include the key, a pointer to the parent node, and pointers to the left and right children nodes.

struct node{

  int data;

  node \*parent;

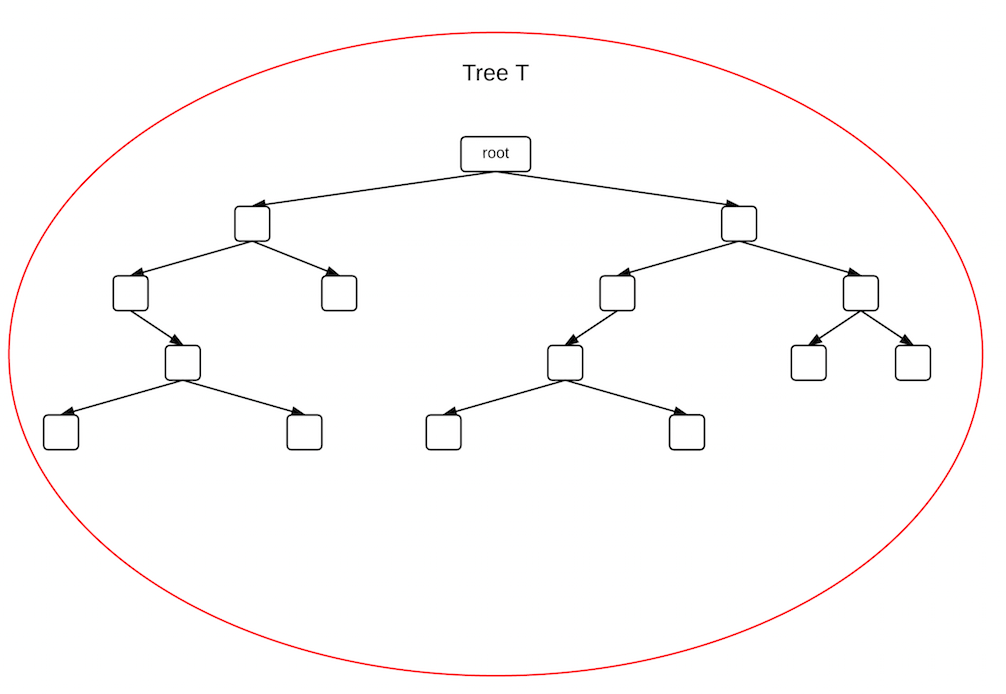
  node \*leftChild;

  node \*rightChild;

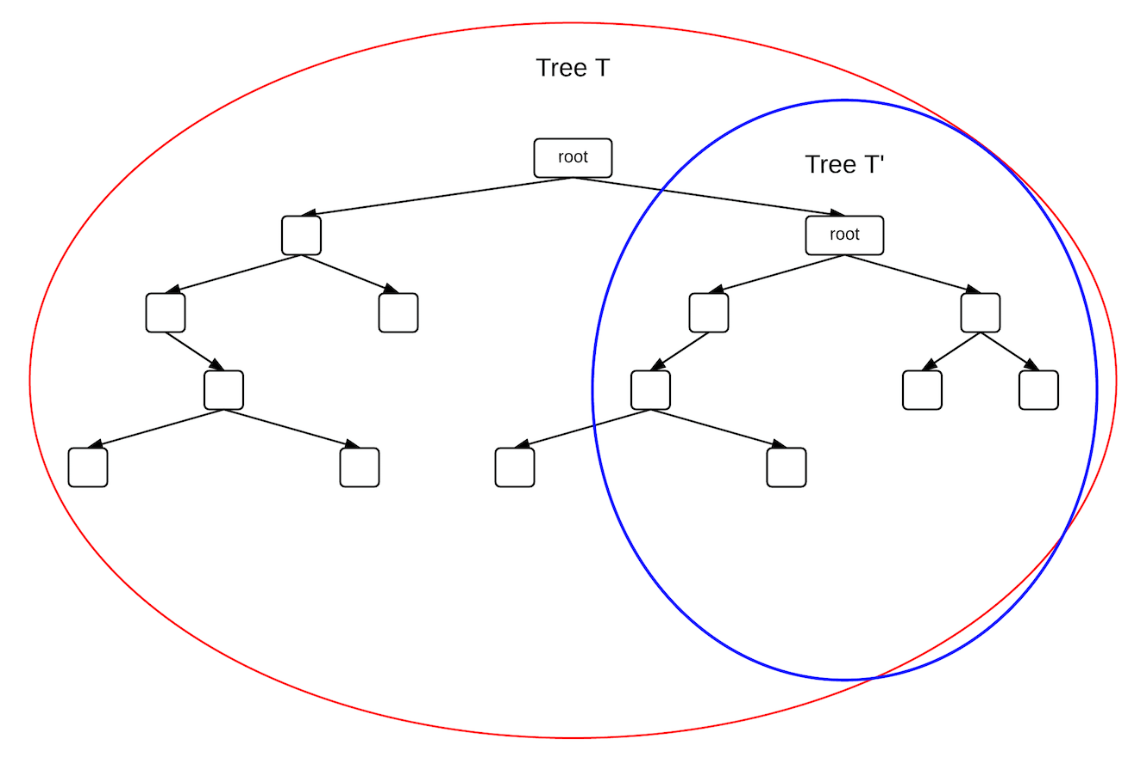
}

## ****Trees and Subtrees****

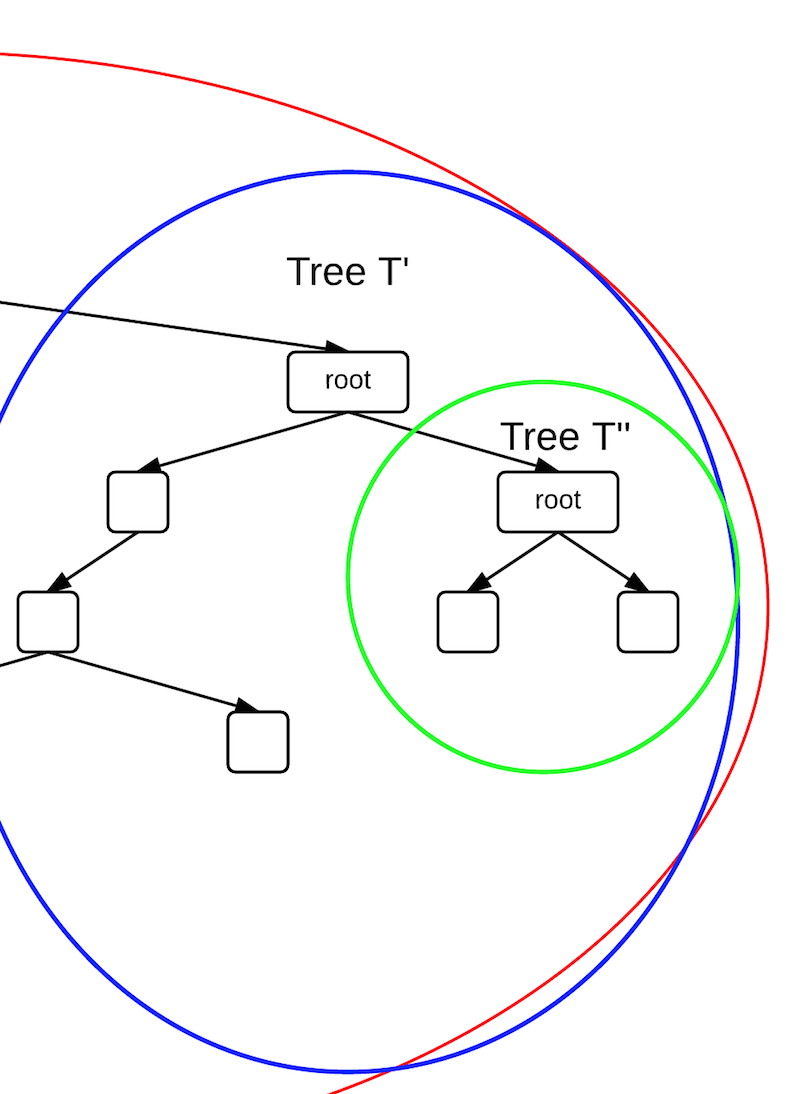
In any tree, there can many subtrees, each with the same basic structure as the larger tree. For example, consider the following tree T, which has a root node, and off of the root there is a left and right child, and those children also have children.



We can redraw the tree using the left or right child as the new root, and for that new subtree T', there is a root and a left and right child.



The pattern continues down to the smallest subtree, which just contains a root and two children, but no additional nodes from the children.



Binary trees are defined in terms of the smaller subtrees within it. This is called self-similarity and it's computationally significant because it means there are elegant ways we can search the tree. Consider a function that takes the root of a tree as an argument and prints the value of the node.

void printNode(Node \*root){  
 cout<<" city: "<<root->name<<endl;  
 if(root->leftChild!=NULL)  
 printNode(root->leftChild);  
 if(root->rightChild!=NULL)  
 printNode(root->rightChild);  
}

We call printNode and pass it a pointer to a Node. If the Node has a left child, then we call printNode again, passing it a pointer to the left child. If the Node doesn't have a left child, we check to see if the Node has a right child, and call printNode passing it a pointer to the right child. This process of a function calling itself is called recursion and it is frequently used on tree structures.

## Recursion

A recursive call to a function solves a smaller and smaller instance of the problem until we've reached the smallest case that we're interested in. Recursion is typically used on problems where the structure of the data is also recursive, such as a tree. For example, consider the file system on your computer. At the top level, there are directories and files. Within the directories, there are other directories and files, and within those directories there can be other directories, and so on. If you search down through the file system, you will see a repeating pattern until you eventually reach a level where there are only files. The directory structure is defined recursively in terms of smaller and smaller directory structures. The smallest unit of the structure is called the base case.

### Algorithms for recursively defined structures

Any recursive algorithm needs to exhibit the following properties.

* **A base case:** The smallest unit of the problem that can be defined. For the printNode algorithm shown above, the base case if when the child of a node is NULL. When the algorithm reaches this condition, the function not call printNode again and instead exit the function.
* **A set of rules that can reduce all cases down to the base case:** For the printNode function the rule is to pass the child of the root node into the next call to printNode. In doing so, we step down in the tree another level until reaching the base case, which is the node in the tree with no children.

[**Computer science course notes**](http://www.microveggies.com/csci)

**Home**

[**Lecture16 - Binary Search Trees**](http://www.microveggies.com/csci/index.php/csci-2270-lecture-notes/4-lecture16-binary-search-trees)

**Details**

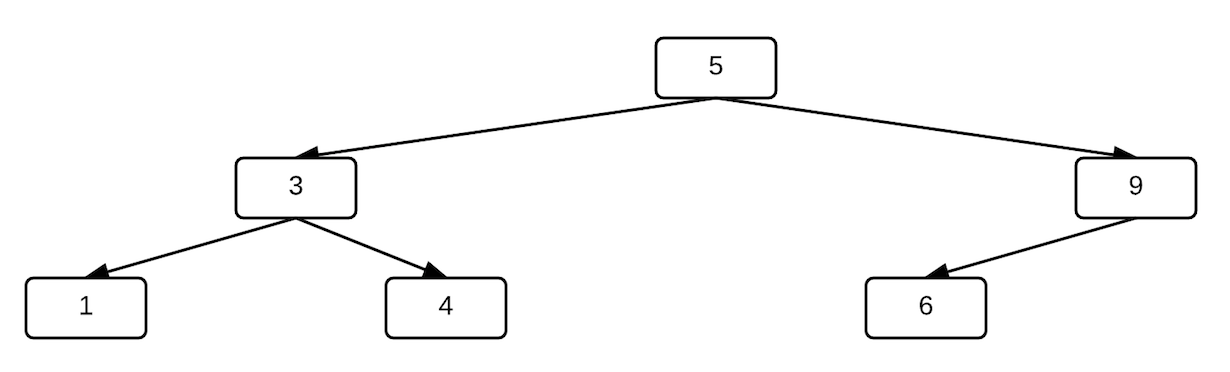
Written by Rhonda Hoenigman

 Published: 17 February 2015

 Hits: 1799

*Pages 286 - 294 in your textbook.*

A "binary search tree" (BST) is an ordered binary tree where the nodes in the left subtree all have a value less than the subtree root value, and the nodes in the right subtree all have a value greater than the value of the subtree root. For example, consider the following BST where the value of each node is an integer.



The root of the tree is 5, and therefore, all values to the left of the root should be less than 5 and all values to the right of the root should be greater than 5. This same pattern continues for the 3 node. The value to the left of 3 is less than 3 and the value to the right of 3 is greater than 3. In this example the keys are integers, But they could also be strings and our comparison would be, is "a" < "b", or "A" < "a"?

**We formally define a binary search tree as follows:**

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key < x.key. If y is a node in the right subtree of x, then y.key >= x.key.

**Example 1:**

Build a binary search tree from the following integers: 4, 2, 6, 9, 1, 3.

First, add 4 as the root.

Evaluate the 2. Since 2 < 4, go left and add 2 to tree as left child of 4.

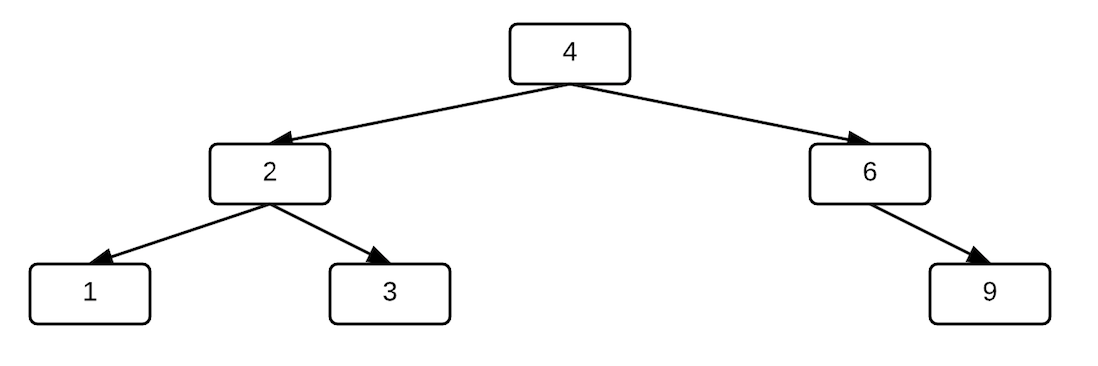
Evaluate the 6. Since 6 > 4, go right and add 6 to tree as right child of 4.

Evaluate the 9. Since 9 > 4, go right. Compare 9 to 6. Since 9 > 6, go right and add 9 as right child of 6.

Evaluate the 1. Since 1 < 4, go left. Compare 1 to 2. Since 1 < 2, go left and add 1 as left child of 2.

Evaluate the 3. Since 3 < 4, go left. Compare 3 to 2. Since 3 > 2, go right and add 3 as right child of 2.

The final tree looks like:



 Example 2:

Build a binary search tree using inputs: DEN, LA, CHI, VEGAS, SD, DET, NY.

First, add DEN as the root.

Evaluate LA by comparing ascii value of "D" to ascii value of "L". "D" is less than "L", so add LA as the right child of DEN.

Evaluate CHI by comparing ascii value of "D" to ascii value of "C". "D" is greater than "C", so add CHI as the left child of DEN.

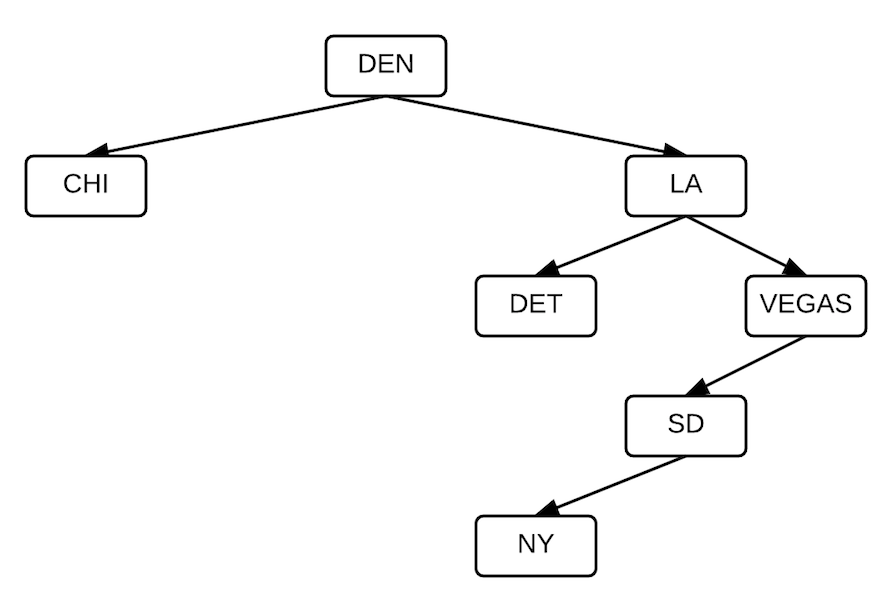
Evaluate VEGAS by comparing ascii value of "D" to ascii value of "V". "D" is less than "V", so go right. Compare "V" to "L" in LA, and add VEGAS as right child of LA.

Evaluate SD by comparing ascii value of "D" to ascii value of "S". "D" is less than "S", so go right. Compare "S" to "L" in LA and go right. Compare "S" to "V" in VEGAS and add SD as left child of VEGAS.

Evaluate DET by comparing "D" to "D", and since they are equal, evaluate the second letter in DEN and DET. They are also equal, so move to the third letter. Since "T" > "N", go right at DEN. Compare DET to LA and add DET as left child of LA.

Evaluate NY by comparing "D" to "N". "D" is less than "N", so go right. Compare "N" to "L" in LA and go right. Compare "N" to "V" in VEGAS and go left. Compare "N" to "S" in SD, and add NY as left child of SD.

The final tree looks like this:



**Searching a binary tree**

The BST ordering generates a structure that is fast to search. From any given node, we can do a targeted search of a portion of the tree that might contain the value we're searching for and eliminate the rest of the tree. If the value we're seeking is less than the value of the node we're evaluating, we know that we don't need to search the right branch of the tree, for example. This removes the unnecessary computation of searching branches that won't contain what we're looking for.

The tree search algorithm pseudocode

Tree\_Search(x, k) //x is the node to evaluate and k is the value being searched for.

  if x==NIL   
 print("node not found")  
 else  
 if x.key == k

    return x

  if x.key > k

    return Tree\_Search(x.left, k) //call Tree\_Search again evaluating left branch only

  else

    return Tree\_Search(x.right, k) //call Tree\_Search again evaluating right branch only

In the tree search algorithm, we only search left when x.key < k, and we only search right when x.key >= k. We can write a non-recursive search algorithm as well that uses a while loop to check for when we have reached the bottom of the tree.

**Adding a node to a binary search tree**

To add a new node to a BST, we need to first search to the location where the node should be added and then insert the node into the tree in the correct place. Adding a node modifies the tree structure.

The non-recursive tree insert algorithm pseudocode

Tree\_Insert(T, z)

  x = T.root

  P = NIL  //P is a tree node

  while(x != NIL){

    P = x  //store current root of the tree

    if z.key < x.key

      x = x.left

    else

      x = x.right

 }     
 z.p = P //set the parent of the new node

 if P == NIL  //tree is empty

   T.root = z

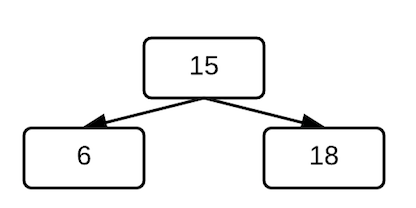
 else if z.key < P.key

   P.left = z

else

   P.right = z

 Example: Given the following BST, add the value 3 to the tree.

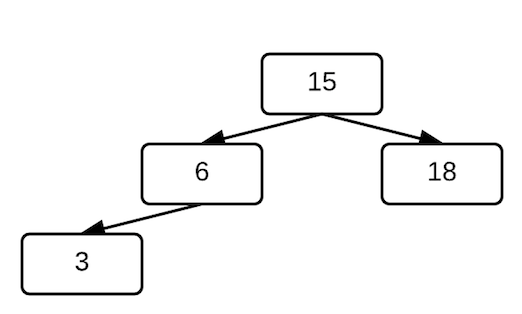


We set x to the 15 node at the line x = T.root. In the while loop, P is set to the 15 node. 3 < 15, so we go left with x = x.left.

The next time through the while loop, P = x sets P to the left child of x, which is the 6 node. We evaluate 3 < 6, so we go left again with x = x.left. Since the 6 node has no left child, this sets x to NULL.

We evaluate the while loop condition again, and since x is NULL, we exit the loop. At this point, P is the 6 node, so z.p = P sets the parent of the new 3 node to the 6 node.

We evaluate the conditional. It's false that P is NULL, and it's true that z.key < P.key, since z.key is 3 and P.key is 6. We set the left child of P to the new 3 node. The new tree looks like:



[**Computer science course notes**](http://www.microveggies.com/csci)

**Home**

[**Lecture17 - Binary Search Trees II**](http://www.microveggies.com/csci/index.php/csci-2270-lecture-notes/6-lecture17-binary-search-trees-ii)

**Details**

Written by Rhonda Hoenigman

 Published: 23 February 2015

 Hits: 1794

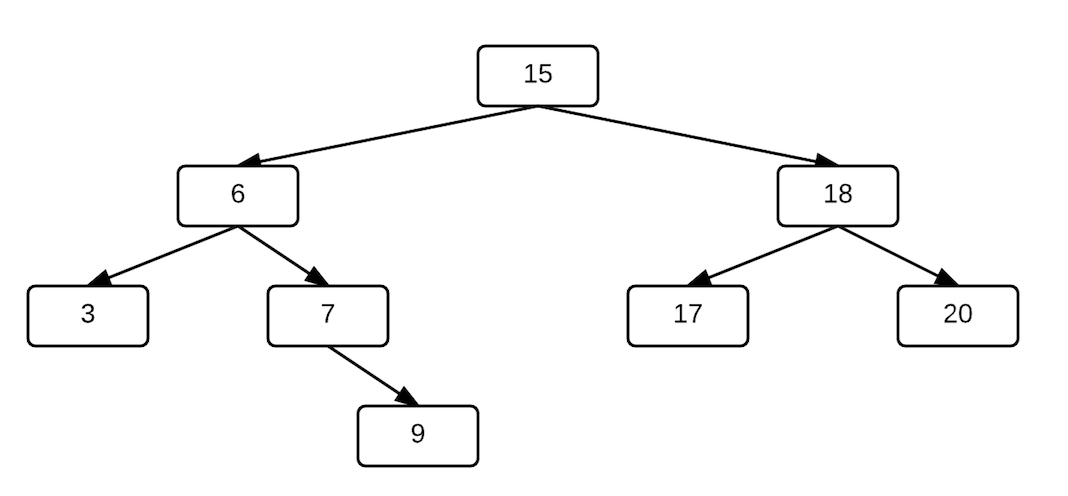
**Deleting a node**

Just as we want to add nodes to a tree, we also want to delete nodes from a tree. Once we do, we may need to realign the tree. The node to delete falls into one of three categories:

1. Node has no children.

2. Node has one child.

3. Node has two children.



**Node has no children**

In the tree shown here, nodes 3, 9, 17, 20 have no children. To delete one of these nodes, reset the child pointer for the parent to NULL, and delete the node. For example, to delete the 3 node, set the left child pointer for the 6 node to NULL and delete the 3 node. To delete the 9 node, reset the right child pointer for the 7 node to NULL and delete the 9 node.

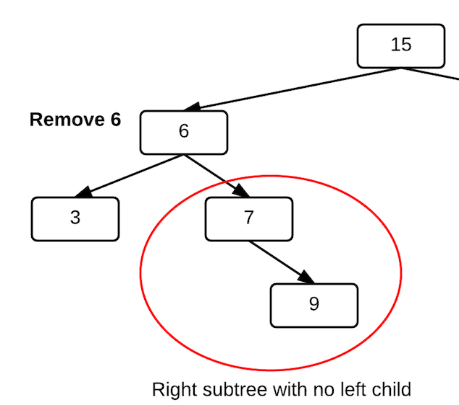
**Node has one child**

In the tree shown here, the 7 node has one child. To delete this node, replace the child pointer of the parent node and delete the node. For example, to delete the 7 node, set the right child pointer of the 6 node to the 9 node.

**Node has two children**

In the tree shown here, the 6, 15, and 18 nodes all have two children.

**Example 1:** Remove 6 node. The deleted node should be replaced with a node from the right subtree that has no left child.



Replace 6 node with 7 node.

Reset pointers for nodes 3, 7 and 15.

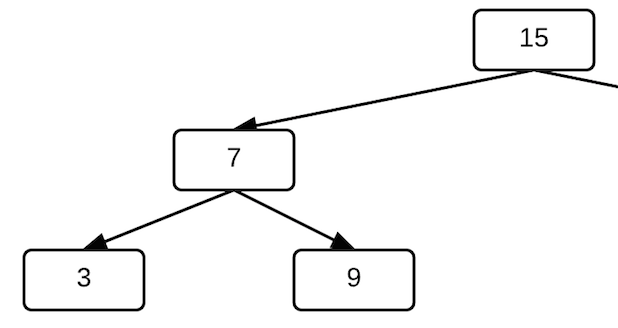
3.parent = 7

15.left = 7

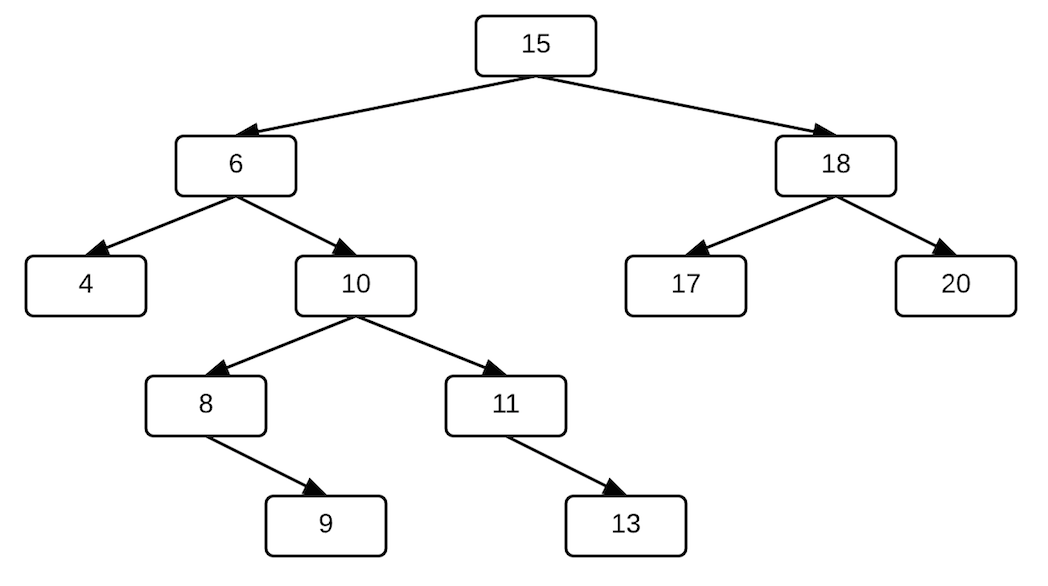
7.parent = 15

The rest of the right subtree is unmodified.

The new tree looks like:



**Example 2:** Remove the 6 node from this tree:



The replacement is in the right subtree, but it not the right child. It needs to be a node that doesn't have a left child and is **the minimum value in the right subtree**.

Evaluate the 10 node, which has a left child, so the 10 can't be the replacement.

Evaluate each of 10's children.

The 11 node doesn't have a left child, so it is a candidate for replacing the 6 node. But, we can't move 11 into the 6 position because 11 > 10, and the BST requirement of the right child > the parent will be violated.

The 8 node doesn't have a left child, and 8 < 10. Move 8 into the 6 position and update the pointers. The 9 node replaces the 8 node as the left child of the 10 node.

10.left = 9

15.left = 8

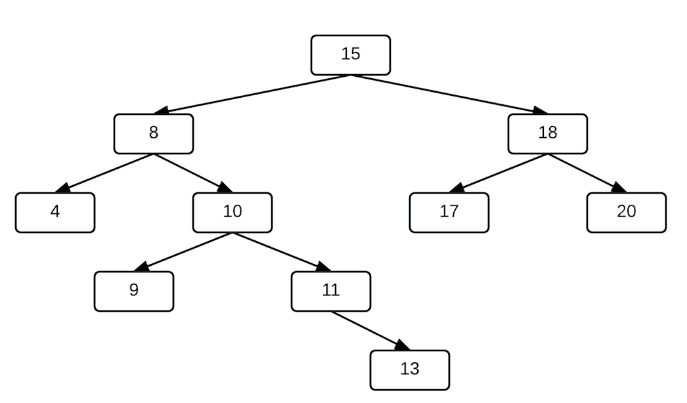
8.left = 4

8.right = 10

10.parent = 8

4.parent = 8

The final tree looks like:



**Other properties of binary search trees**

**Finding the min, max of a tree, subtree**

The minimum value in a tree is the left-most node in the tree.

Tree\_minimum(x)

  while x.left != NIL

    x = x.left

  return x

The maximum value in a tree is the right-most node in the tree.

Tree\_maximum(x)

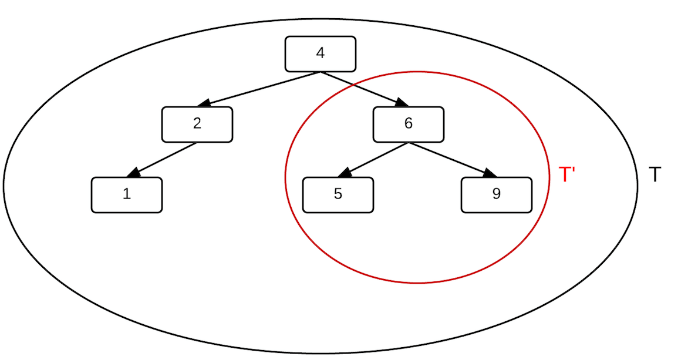
  while x.right != NIL

    x = x.right

  return x

**Find the minimum of a subtree**

 The Tree\_minimum and Tree\_maximum algorithms can take any node in the tree as the starting node. For example, consider the tree T and subtree T' in this image:



Tree\_minimum(T.root) = 1

Tree\_minimum(T'.root) = 5

Tree\_maximum(T.root) = 9

Tree\_maximum(T'.root) = 9

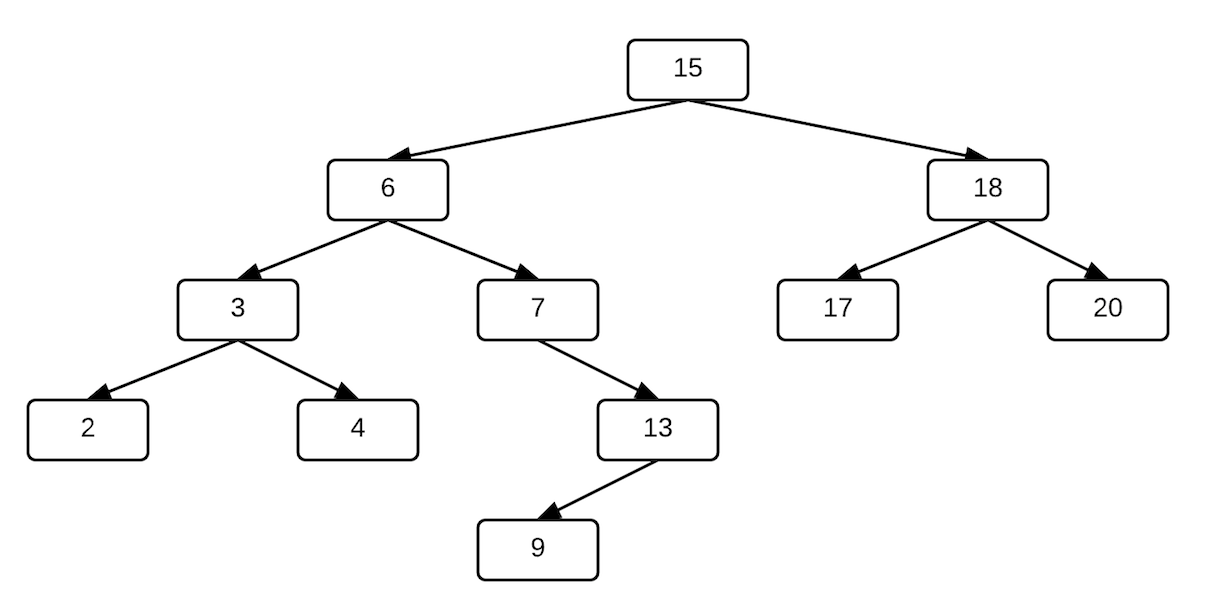
**Finding the nodes in a tree that satisfy a given criteria**

**Example 1:** Find the nodes in the tree T that have a value less than 3?

Start at the root, 3 < 4, so go left.

Evaluate 2 node, 2 < 3, so entire left branch of 2 will be less than 3. The 2 node doesn't have a right branch to evaluate.

**Example 2:**Find the nodes that have a value less than 10 in the following tree?



Evaluate the root, 10 < 15, so go left.

Evaluate the 6 node, 6 < 10, so the 6 and all nodes left of the 6 node are less than 10. The nodes in the right branch of 6 are between 6 and 14, we need to evaluate that branch to determine how many of the nodes are less than 10.

Evaluate 6.right, which is 7 and less than 10.

Evaluate 7.right, which is 13. The right children of 13 are all greater than 10, and the left children of 13 have to be between 7 and 12.

Evaluate 13.left, which is 9. The 9 node has no children to evaluate.

[**Computer science course notes**](http://www.microveggies.com/csci)

**Home**

[**Lecture18 - Binary Search Trees III**](http://www.microveggies.com/csci/index.php/csci-2270-lecture-notes/7-lecture18-binary-search-trees-iii)

**Details**

Written by Rhonda Hoenigman

 Published: 25 February 2015

 Hits: 1646

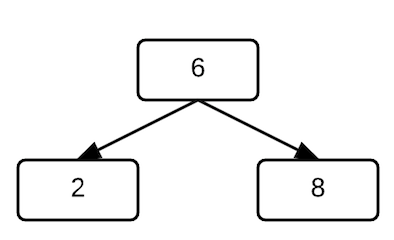
**Tree Traversal Algorithms**

For any tree, whether it's a binary tree, a binary search tree, or an n-ary tree (which we haven't talked about yet, but we will), we want to be able to traverse the tree such that we visit each node in the tree exactly once. The algorithm we use for tree traversal determines the order in which the nodes are visited. There are three orderings to consider:

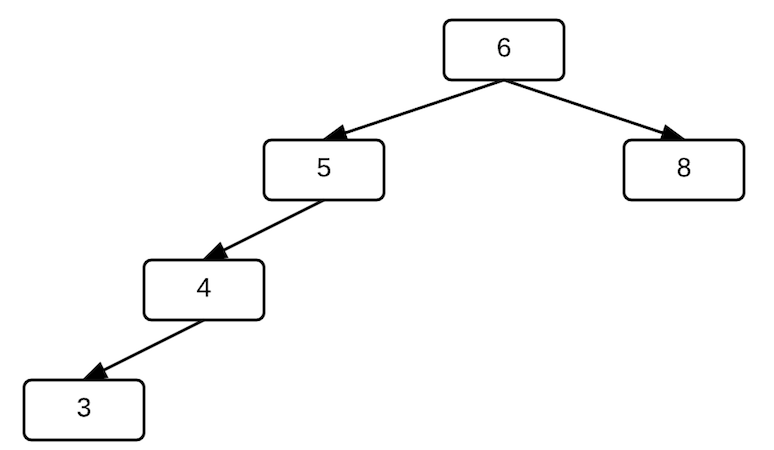
* Inorder - print the values in the tree in sorted order. Nodes are visited in the order: left-parent-right
* Preorder - print the root value before printing the left and right child values
* Postorder - print the left and right child values before the parent value.

**Inorder tree traversal**

Consider the three-node tree shown here:



The sorted values of the nodes, lowest to highest, is 2, 6, 8. This can also be expressed as left-parent-right. If the 2 or the 8 had children, we would need to apply left-parent-right to each of those nodes as well to get all of the nodes sorted correctly. For example, in the following tree, there are three left children, and we can find the sorted order by traversing all the way down the left branch and printing the values from the bottom of the tree to the top before we print the parent node and then the right child node.



The code to print the tree inorder looks something like this:

printNode(x){

  if x.left != NULL

    printNode(x.left);

  cout<<x.key<<endl;

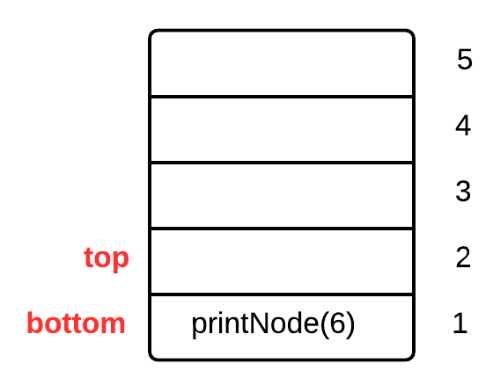
  if x.right != NULL

    printNode(x.right);

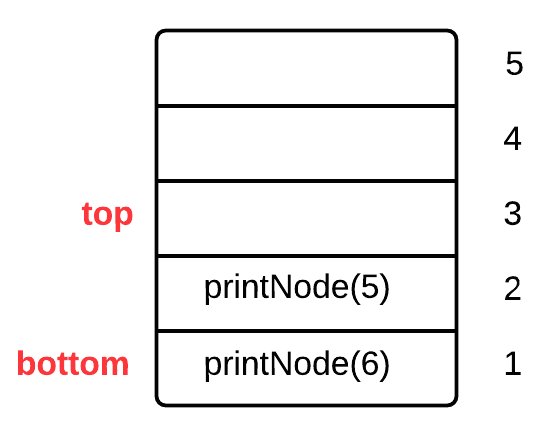
}

The code calls printNode recursively, each time passing in the left or the right child of a parent node. When the function is called, it is pushed onto the call stack, and then popped off the stack when the function completes. We can step through these pushes and pops to see why the values are ordered when they are displayed.

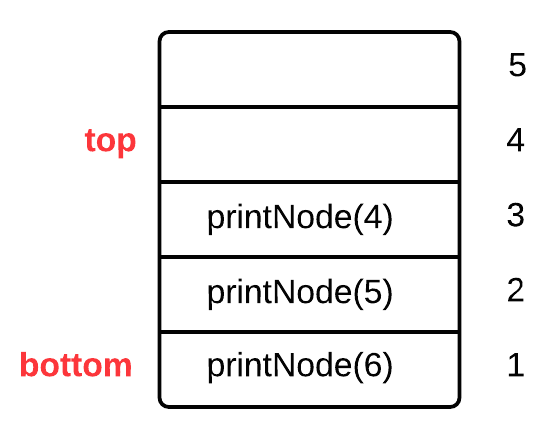
First, we call printNode from the root, which we'll represent as printNode(6), since 6 is the value of the root node. At this point, there's one call to printNode on the call stack, which looks something like:



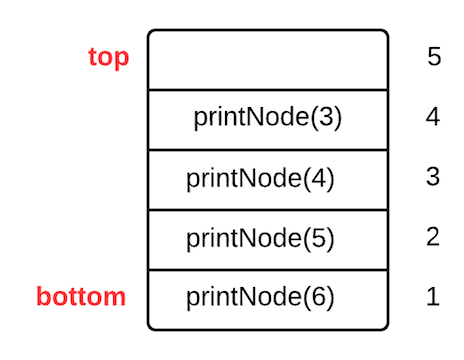
We evaluate the conditional x.left != NULL and it's true, so we call printNode again, this time passing it the left child of the root, printNode(5). The call stack looks like:



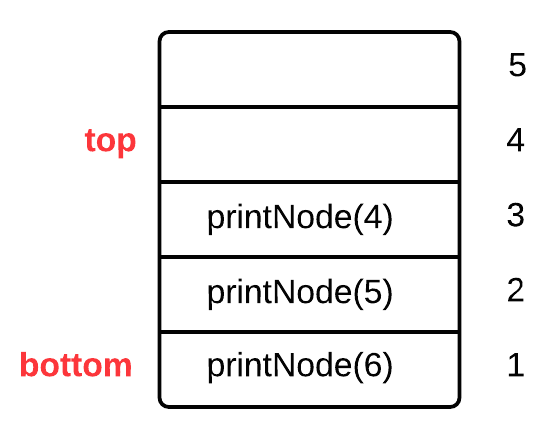
On this call to printNode, we again evaluate the x.left != NULL conditional and it's still true, so we call printNode(4). This pushes another call to printNode onto the call stack. After the call to printNode(4), the call stack looks like:



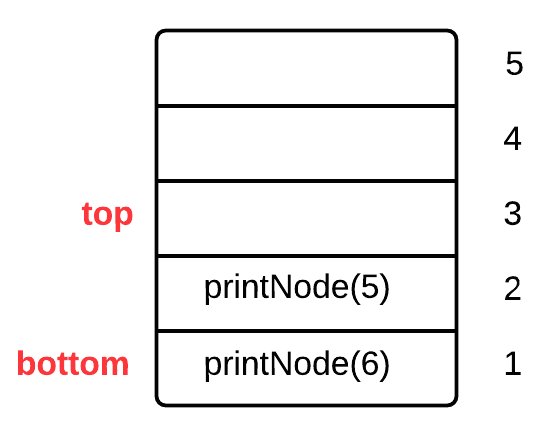
We call printNode again, this time for the left child of the 4 node, which looks like printNode(3). This pushes the call to printNode onto the call stack, and the stack looks like:



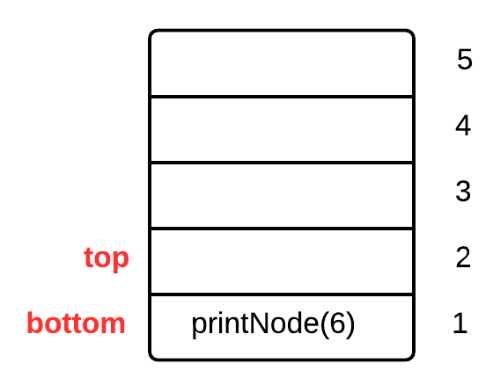
On this call to printNode, the x.left != NULL conditional is false because the 3 node doesn't have a left child, so we skip over the call to printNode, and we advance to the line that prints the value of x.key, which is 3. Our program prints a 3 at this point. Next, we go to the line that evaluates if x.right != NULL, and since the 3 node doesn't have a right child, the conditional is false, so we skip over the call to printNode and exit the function. Once we do, this call to printNode is popped off the stack, and the call stack looks like:



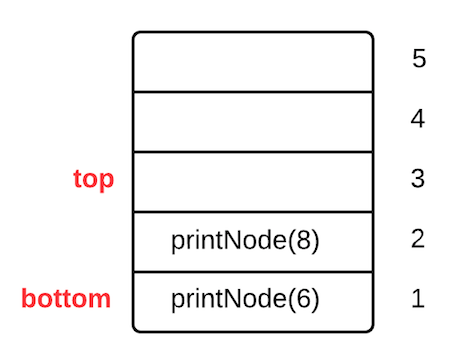
We return to the point where printNode(3) was called, which is just after the call to printNode(4). The next line to execute is the cout, which prints a 4. At this point, our program has output 3, 4. The next code line that runs is the x.right != NULL, and this evaluates to false since the 4 node has no right child. We skip the call to printNode for a right child, and exit the function. Exiting pops printNode(4) off the stack, and our call stack looks like:



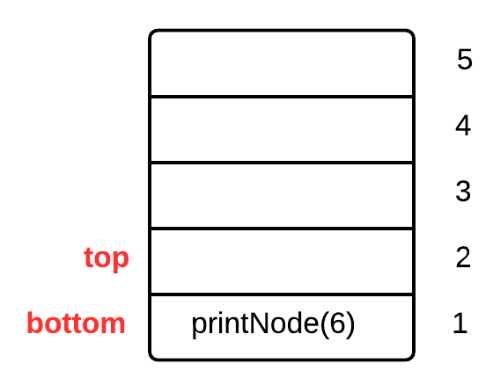
 The code executing returns to the point in printNode(5) where printNode(4) was called. The next line to execute is the cout statement, which prints 5. At this point in the program, our output is 3, 4, 5. The next line checks for a right child, which the 5 node doesn't have. The printNode(5) call completes and is popped off the stack, which makes the stack look like:



The code executing returns to the point in printNode(6) where printNode(5) was called, and the next line to execute is the cout statement, which prints a 6. At this point in the program, our output is 3, 4, 5, 6. Next, we evaluate the x.right != NULL, which is true because the 6 node has a right child. printNode is called again as printNode(8) and the call stack looks like:



In the call to printNode(8), the first line to execute is the conditional x.left != NULL. This is false because the 8 node doesn't have a left child, so we skip over the call to printNode inside the conditional block. The next line to execute is the cout statement, which prints an 8. The output from the program up to this point is 3, 4, 5, 6, 8. The next line checks if the 8 node has a right child, which it doesn't, and the printNode(8) call completes and is popped off the stack:



The code returns to the point in printNode(6) where printNode(8) was called. The next line in printNode(6) is the end of the function. The printNode(6) function completes and is popped off the stack. The stack is empty, and we're done traversing the tree.

**Preorder tree traversal**

In a preorder tree traversal, the root value is printed before the children.

In a recursively defined function called printNodePreorder (shown below), the cout statement to output the value of the node precedes additional calls to printNodePreorder for each of the node's children. If you step through this code and draw the output and the call stack each time printNodePreorder is called, the output you will see for the tree used in the inorder traversal is 6, 5, 4, 3, 8.

printNodePreorder(x){

  cout<<x.key<<endl;

  if x.left != NULL

    printNodePreorder(x.left);

  if x.right != NULL

    printNodePreorder(x.right);

}

**Postorder tree traversal**

In a postorder tree traversal, the children are printed before the root value.

Using the postorder printing routine and the tree from the previous examples, the output would be 3, 4, 5, 8, 6.

printNodePostorder(x){

  if x.left != NULL

    printNodePostorder(x.left);

  if x.right != NULL

    printNodePostorder(x.right);

cout<<x.key<<endl;

}

[Back to Top](http://www.microveggies.com/csci/index.php/8-csci-2270-lecture-notes/7-lecture18-binary-search-trees-iii#top)

© 2015 Computer science course notes