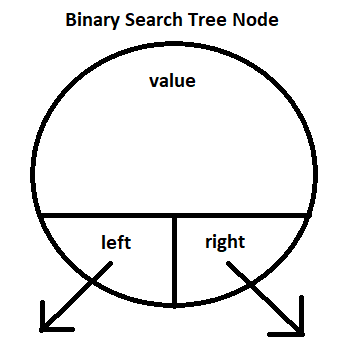
# Binary Search Tree Notes

A Binary Search Tree is a data structure that helps store massive amount of data in an organized fashion such that inserting, removing, and searching for data can be done at an improved speed. Previously, the programmer has been using nodes to create forms of linked lists which allowed the programmer to not be constrained by contiguous memory. However, most of these forms of linked lists only allowed a linear time constraint at best, O(n). A Binary Search Tree, also known as BST, introduces the ability to use nodes in an logarithmic time constraint, O(log n). The Binary Search Tree is created as a sorted data structure that will allow quicker additions, searches, and removals. The worst case scenario for a binary search tree is a linear time constraint, O(n), which makes it a stronger case for large data storage solutions.

## The Binary Search Tree Node

The Binary Search Tree Node is comprised of an Object value and two Binary Search Tree Node pointers, a left pointer and a right pointer.



The programmer should be able to write their own Binary Search Tree Node using this diagram. It is encouraged that the value should be declared as a Comparable, because the value would need to be used with a comparator. If the programmer declares value as a Comparable, then any object that implements Comparable could be used in the Binary Search Tree Node. The programmer would also be encouraged to write a toString method that would return not only the value but also the value of the left node, or null if there is no left node, and the value of the right node, or null if there is no right node. Here is a partial declaration of the Binary Search Tree Node:

public class BinaryNode {

private BinaryNode left, right;

private Comparable myValue;

public BinaryNode(Comparable x)

// implementation not shown

public String toString()

{

String temp = "Value:"+myValue+

", Left:"+(left==null?null:left.myValue)+

", Right:"+(right==null?null:right.myValue);

return temp;

}

//other methods not shown.

}

The code

", Left:"+(left==null?null:left.myValue)

uses the conditional operator ? : which is used in place of if…else. The previous code is similar to writing

if(left==null)

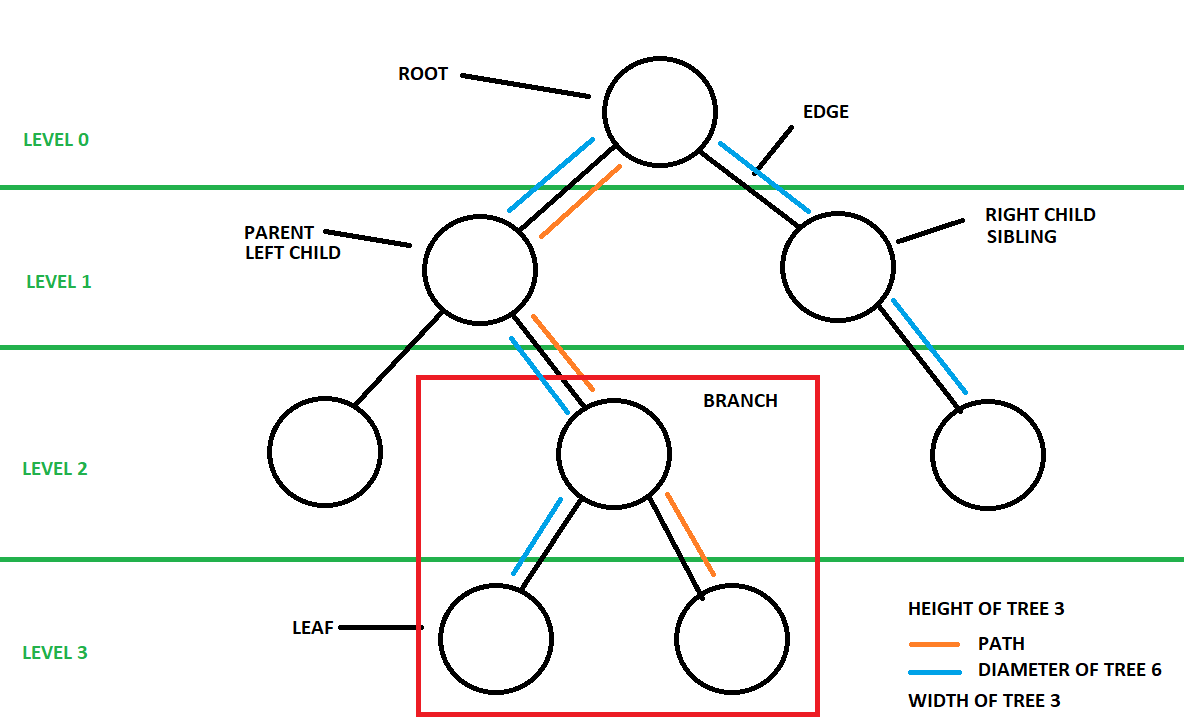
temp+=", Left"+null;

else

temp+=", Left"+left.myValue;

this is sometimes preferable to use when writing a simple line of code.

## Terminology for Binary Trees

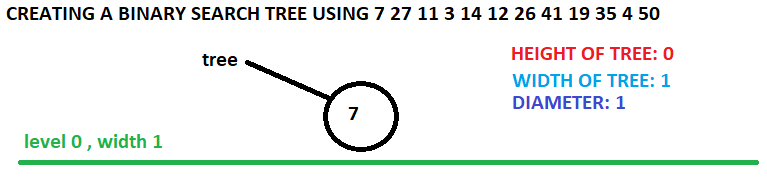
The **root** is the beginning of the binary search tree. It is the only node of a tree that does not have a parent. A **parent** is any node that has at least one child. In a binary search tree, a parent can have a maximum of only two children. A **child** is a node that is being pointed to by a parent. The child is either a left child or a right child. A **left child** is always comparably less than the parent. A **right child** is always comparably greater than the parent. The pointer that connects the parent to the child is called the **edge**. A **sibling** is a child that shares the same parent with another child. A node can be both a parent and a child. A **leaf** is a node that has no children, it is also known as an **outer node** or **terminal node**. An **internal node**, commonly called a parent node, is a node that has a child. An internal node is also known as an **inner node** or a **branch node**. The **degree of a node** is the number of children the node has. In the case of a binary search tree, the nodes could be degree 0, 1, or 2. This means a leaf is a node of degree 0 and a parent is a node of degree 1 or 2. The **level** of a node is based on the distance the node is from the root. The root level is zero. The level, or **depth**, can be found by counting the number of edges between the desired node back to the root. The **height** of the tree is longest path from root to the deepest leaf. It is always equal to the level of that particular leaf. An easy way to find the height is to count the number of edges on the path to the deepest leaf. A **path** is the connection between one node to another, but always moving in a parent to child direction. A **branch**, or **sub-tree**, is any portion of a tree starting from any node in the tree. This node could be a leaf and be a sub-tree of level 0 or it could be internal node and have a level greater than 0 and less than the height of the tree. The **diameter** of a tree is the number of nodes from lowest level of the left branch of the tree to the lowest level of the right branch of the tree which also passes through the root of the tree. This can often be confused with the width of the tree. The **width** of the tree is the largest number of nodes on any of the levels of the tree.

## Inserting into a Binary Search Tree by Hand

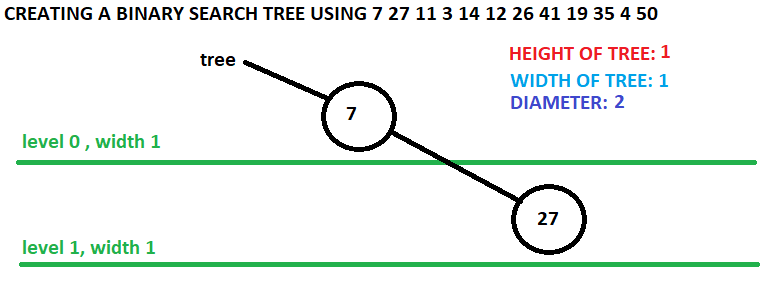
Let us consider the following numbers to input into a binary search tree:

7 27 11 3 14 12 26 41 19 35 4 50

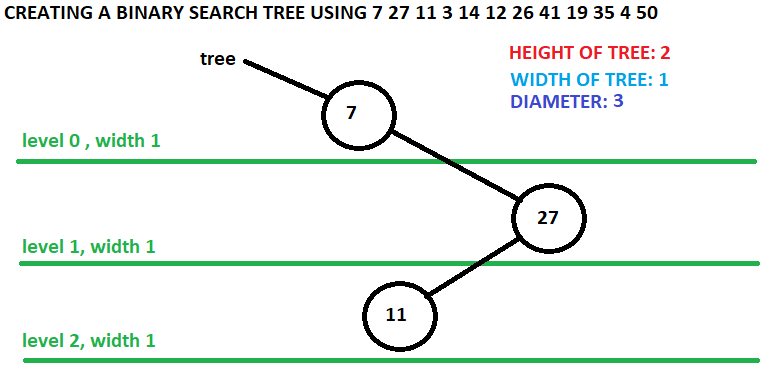
The first value entered will be 7 and it will be placed as the root. This will create the 0th level of the tree. The tree’s height is 0. The tree’s width is 1. The tree’s diameter is 1.



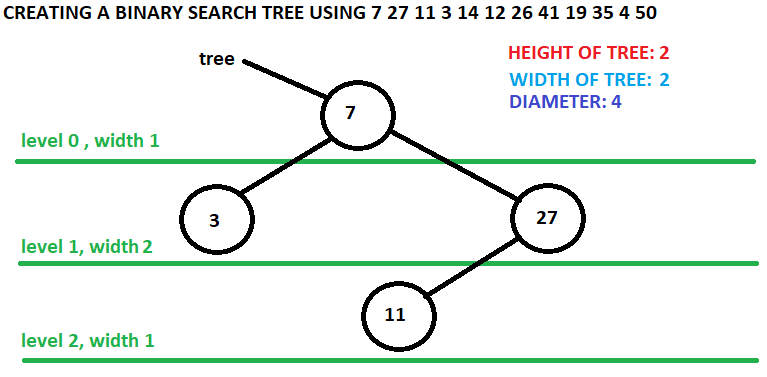
The next value to be entered will be 27. The 27 node will start by being compared to the root node. Since the 27 node is greater than the root 7, the 27 node will be attached as a right child. This will create a new level, increase the height of the tree and the diameter of the tree.



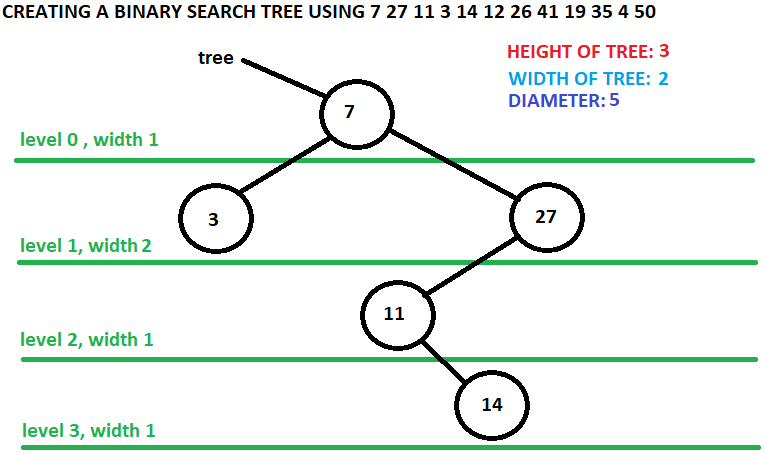
The next value to be entered will be 11. The 11 node will start by being compared to the root node. Since the 11 node is greater than the root 7, the 11 node will be traverse to the 27 right child node. The 11 node would then compare to the 27 node. The 11 node will then attach as the left child of the 27 node as it is comparatively smaller than 27. Every node that is added to a binary search tree will continue down the path in this way until it gets to a null child, at which point it would attach to that parent as that particular child. When 11 is added, a new level is created, the height is increase as well as the diameter.



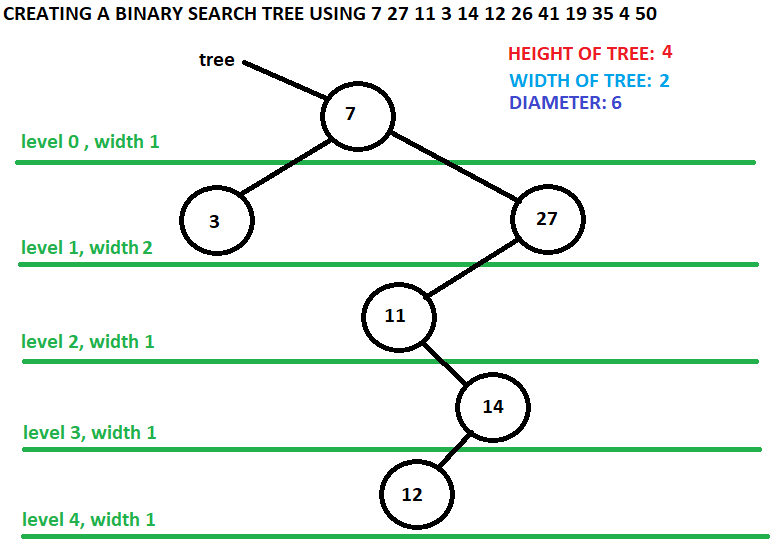
The next node, 3, will be compared to the root and placed as the left child of the root. Level 1’s width will increase, along with an increase to the width of the tree and the diameter of the tree.



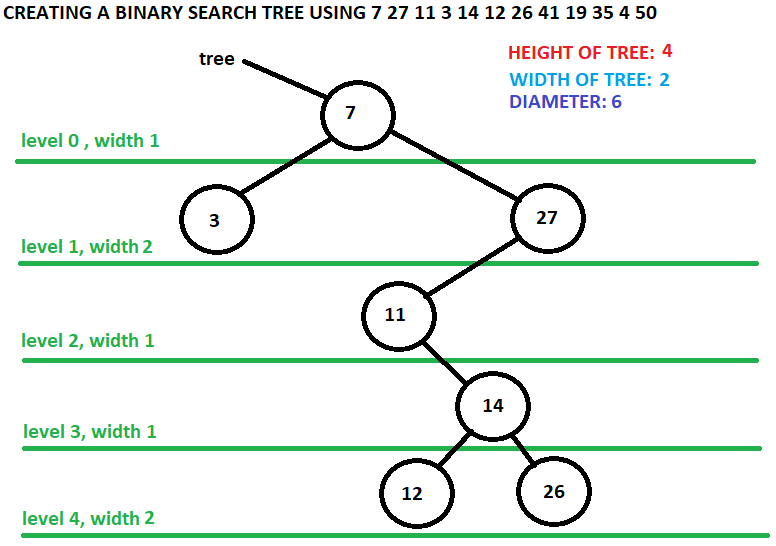
The 14 node will follow the path of the 7 node to the 27 node to the 11 node. The 14 node will attach to the 11 node as its right child. A new level is created, the height of the tree increases, and the diameter increases.



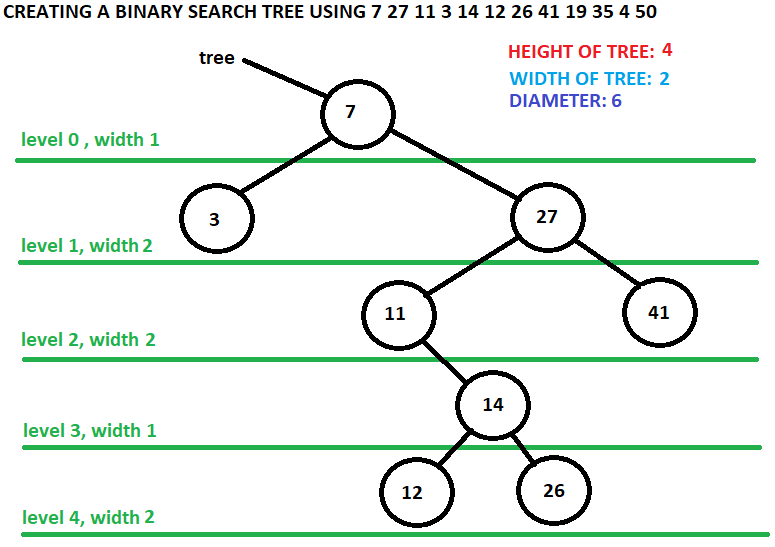
The 12 node will follow the path of the 7 node, 27 node, 11 node, and 14 node where it will attach as the left child to the 14 node. At this point in the binary search tree, it looks more like a linked list than a tree, causing it to be inefficient. The worst case scenario would occur if this keeps up, but with the addition of more values, it is likely that the tree will start to fill out. One of the reasons the binary search tree looks bad in this case is that the root is an extremely low value of the list causing the tree not to be more full or balanced. A **balanced tree** is a tree whose left and right subtrees differ in height by no more than 1 level. A **full tree** is a tree which only have leaves or parents with two children. A **complete tree** is a tree where every level, except the bottom level, is maxed out and the nodes on the bottom level are as far left as possible. When node 12 is added, a new level is created, the height is increased, and the diameter is increased.



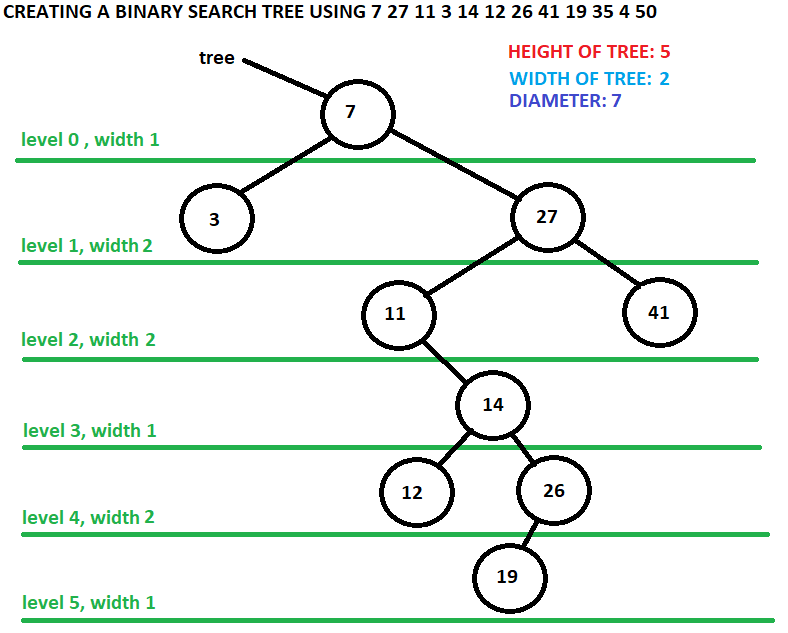
The 26 node will follow the path of the root, 27 node, 11 node, and 14 node where it will attach itself as the right child. The level 4 width will increase.



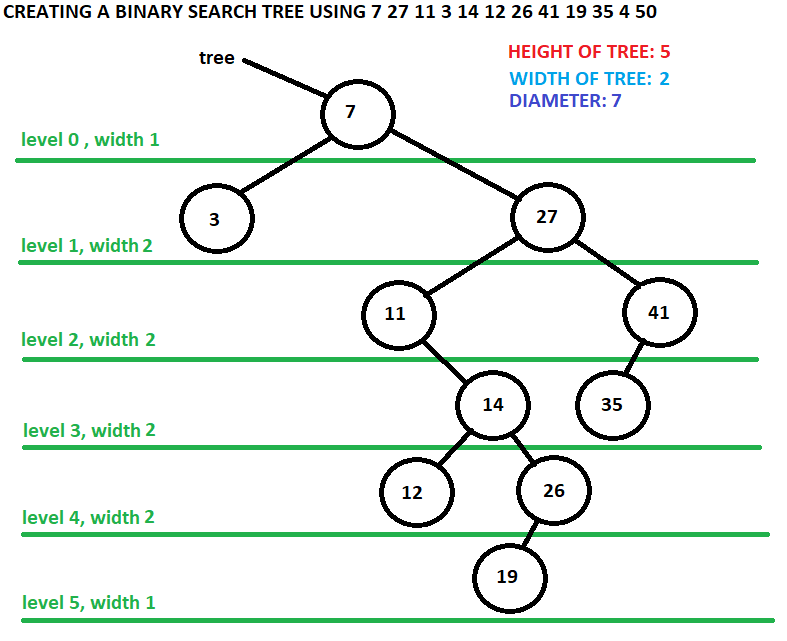
The 41 node will follow the path of the root and 27 node where it will attach itself as the right child. This will increase level 2’s width by 1.



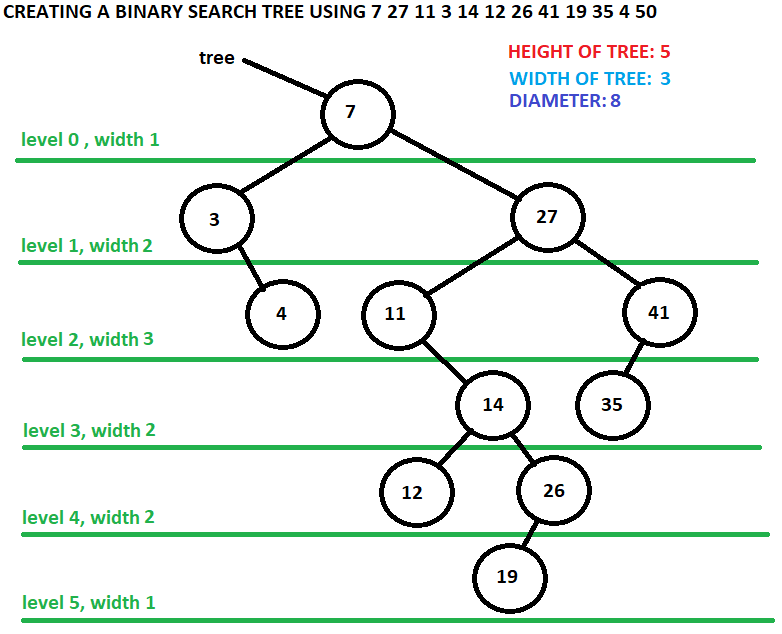
The 19 node will follow the path of the root, 27 node, 11 node, 14 node, and 26 node where it attach itself as the left child. This will create a new level, increase the height of the tree, and increase the diameter.



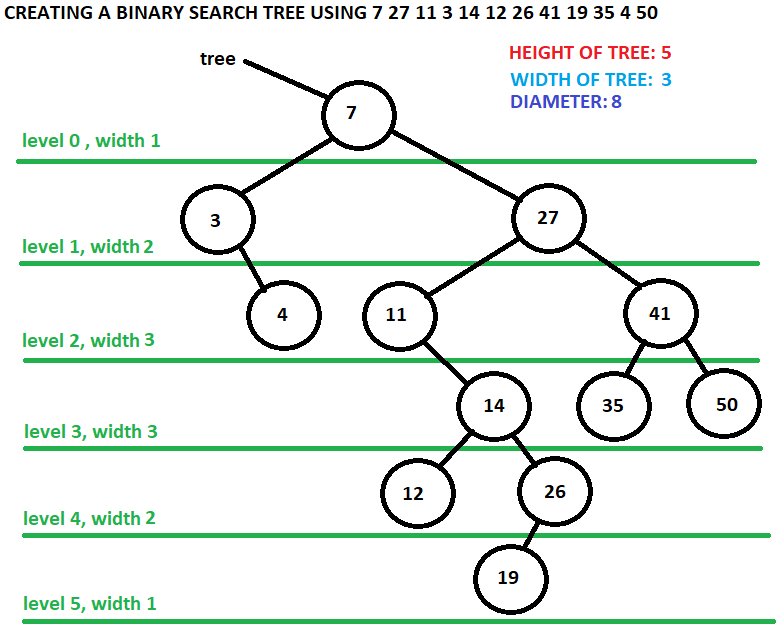
The 35 node will follow the path of the root, 27 node, and 41 node where it will attach as the left child. The level 3 width will increase by 1.



The 4 node will follow the path of the root and 3 node where it will attach as the right child. The level 2 width will increase which will cause the width of the tree to increase as well as increasing the diameter.



Finally, the 50 node will follow the path of the root, 27 node, and 41 node where it will attach itself as the right child. This will increase level 3’s width. At the end of this attachment, the height of the tree is 5, the width of the tree is 3, and the diameter of the tree is 8.



## The Binary Search Tree Class

A binary search tree class is built by the programmer. The class should create a root attribute and have a default constructor that sets the root to null. The class should have many methods, among them should be an add method, a remove method, and a traversal method. The programmer will start by building the add method. The programmer will start with a private Binary node called root and a constructor that will create an empty binary search tree where root points at null.

public class BinarySearchTree {

private BinaryNode root;

public BinarySearchTree()

{

root = null;

}

}

## The add method

The add method is meant for client to use to add a binary node to the tree. The programmer will be adding the node by way of recursion which means that an additional overloaded private add method will be created. The public add method will check to see if the binary tree is empty and if so will add the binary node into the tree as a root. If the tree is not empty, then the method will call on the private recursive add method.

public void add(BinaryNode x)

{

if(root == null)

{

root = x;

return;

}

add(root, x);

}

The recursive add method will receive two binary nodes, the parent node and the desired node to be added to the tree. In order to protect from a memory error, the method will first make sure that the parent node is not null, this should never be the case but it is useful to have that check in place in case that does happen. Once the parent node is confirmed to exist, the desired node is checked against the parent node. If the desired node is comparatively less than the parent node, the method will attempt to place the desired node in as the left child of the parent. In order to do this, the method confirms that the left child of the parent is null. If it is null, then the parent assigns the left node to the desired node and the method is complete. If it is not null, then the method would recursively call the add method again, sending up the left child along with the desired node. Conversely, if the desired node is comparatively greater than the parent node, then the same logic is applied but using the right child rather than the left child. The following is the described code:

private void add(BinaryNode parent, BinaryNode x)

{

if(parent == null) return;

if(x.getValue().compareTo(parent.getValue()) < 0)

if(parent.left()==null)

parent.setLeft(x);

else

add(parent.left(),x);

else

if(parent.right()==null)

parent.setRight(x);

else

add(parent.right(),x);

}

The programmer could check this code by creating a binary search tree driver that would create a binary search tree and fill the tree with sample data.

public class BinarySearchTreeDriver

{

public static void main(String args[])

{

BinarySearchTree tree = new BinarySearchTree();

String str = "7 27 11 3 14 12 26 41 19 35 4 50";

String[] list = str.split(" ");

for(String k:list)

tree.add(new BinaryNode(Integer.*parseInt*(k)));

}

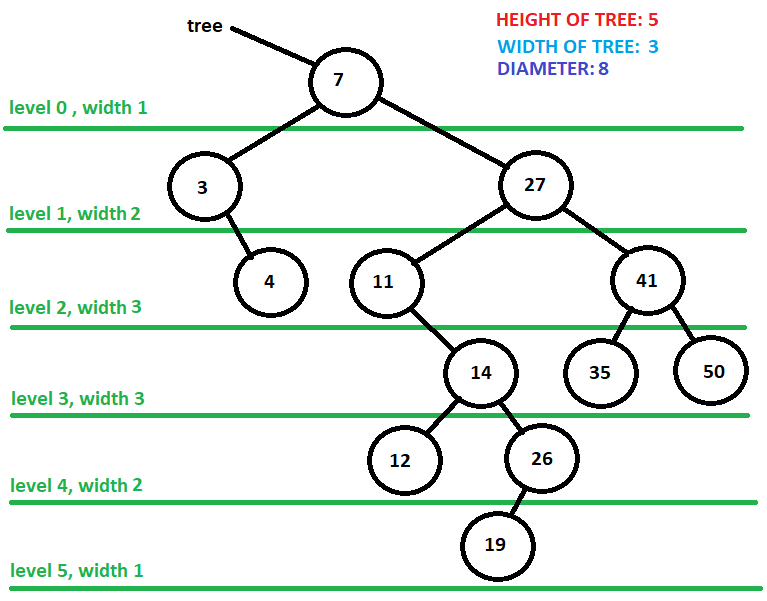
}

## Traversing a Tree

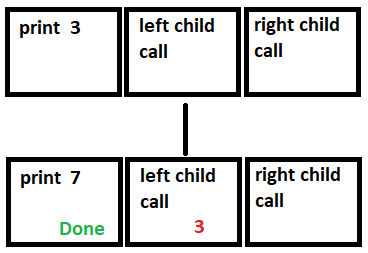
When the programmer tries to run the main method of the Binary Search Tree Driver program, there is no output. The programmer needs a way to show the binary tree. A traversal is a way to go to every single element of a tree. There are three basic traversals, using a similar from of recursion, that will accomplish this: the preorder traversal, the inorder traversal, and the postorder traversal.

## The Preorder Traversal

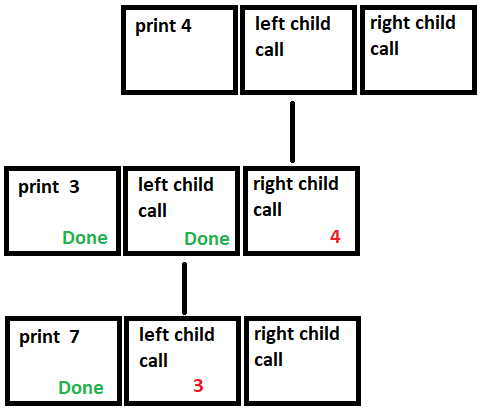
The preorder traversal follows the algorithm of: get the value of the node, go to the left child of the node, and go to the right child of the node. Every time the traversal goes to a new node, that node’s algorithm must be completed before it returns to the previous node. It is creating a stack to follow. Look at the previously created binary search tree from earlier.



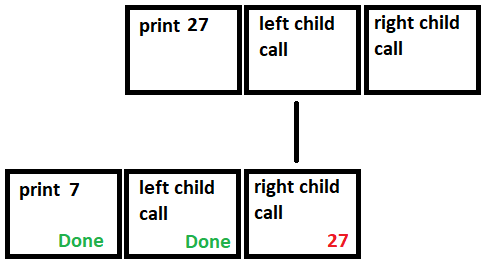
Starting with the root, the traversal command would be to print 7 and then go to the left child. The minute it goes to the left child, a new recursive method would start up and the remaining command for the 7 node traversal would not occur until the left child’s traversal is completed. Here is a visualization of the current orders



The current command is sitting at print 3, and the 7 node right child call is waiting for the 3 node to complete is traversal before continuing. The value 7 would have already been printed at this time. The 3 node would print 3 and then call the left child which is a null. Since it is a null the traversal algorithm would do nothing for the left child and move on to the right child call of the algorithm, which is to do the traversal for the 4 node as seen below

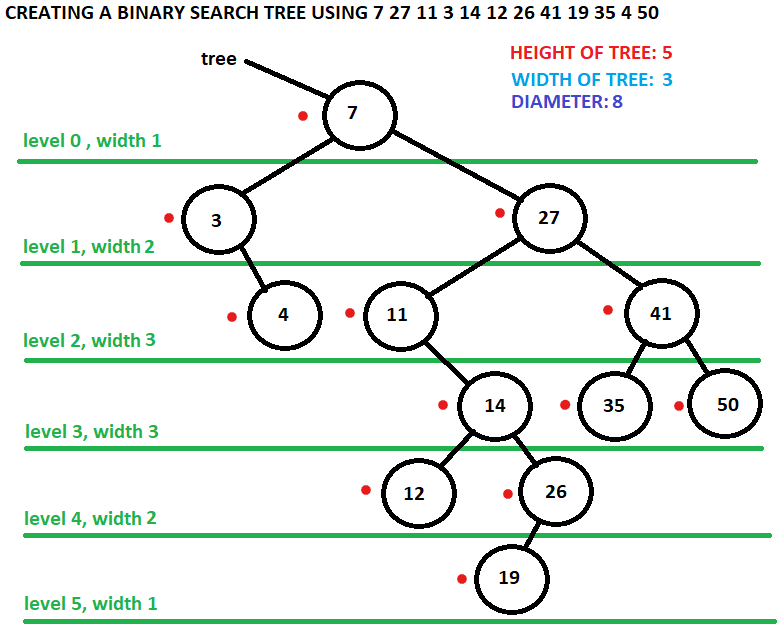


The 7 and the 3 have been printed out and the call to the 4 node is occurring. When the traversal goes to the 4 node, the 4 is printed out and the left child is called, which is a null. This will complete the call to the 4 node’s left child. The call to the 4 node’s right child would occur which is also a null and would result in the completion of the the 4 node’s right child call. This ends the traversal of the 4 node. The control would return to the 3 node and complete the 3 node’s right child call which ends the the 3 node’s traversal, returning the control back to the 7 node. The 7 node can now make the 7 node’s right child call. Here is what the stack would look like at that moment.

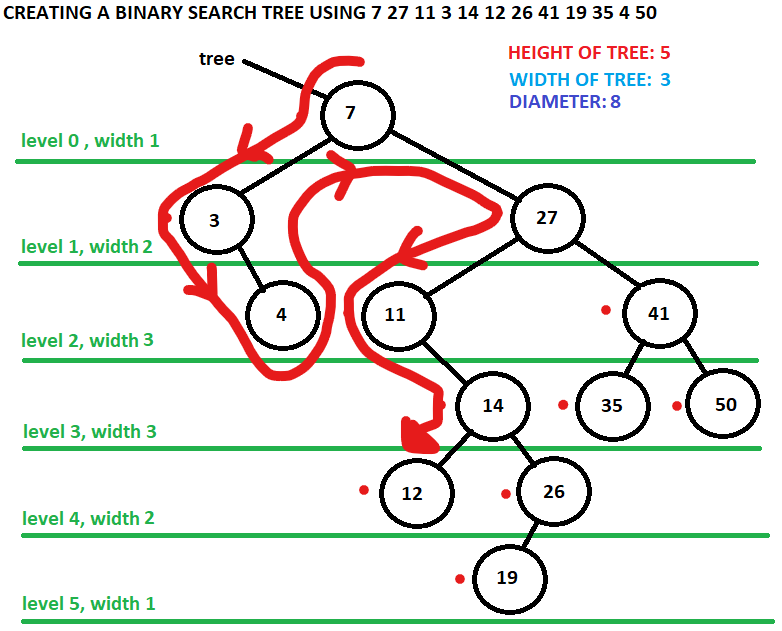


The current print would be 7 3 4 and a call to the traversal of 27 would be occurring. This traversal would continue until the final printout would be 7 3 4 27 11 14 12 26 19 41 35 50. Each traversal would follow the same stacking order, at one time the stack would be the 7 right child call, waiting on the 27 left child call, waiting on the 11 right child call, waiting on the 14 right child call, waiting on the 26 left child call, waiting on the 19 traversal to complete. That is the nature of the of recursive calls, it can be very fast but it can also stack up memory usage as well.

There is a faster way to physically create the preorder traversal from a handwritten binary search tree. Once the tree is created, take a writing utensil and place a dot on the left of every node like the picture below



Starting from the top of the root node, trace a path counter clockwise around the edges of the tree, making sure to go through each dot. Every time the path goes through a dot, write the nodes value down. When the path is completed a preorder traversal has been created. The example below is partway through the preorder traversal, having covered 7, 3, 4, 27, 11, and 14.



Here is the code for a preorder traversal

public String preOrder()

{

return preOrder(root).trim();

}

private String preOrder(BinaryNode k)

{

String temp = "";

if(k != null)

{

// use value

temp += k.getValue()+ " ";

// go left

temp += preOrder(k.left());

// go right

temp += preOrder(k.right());

}

return temp;

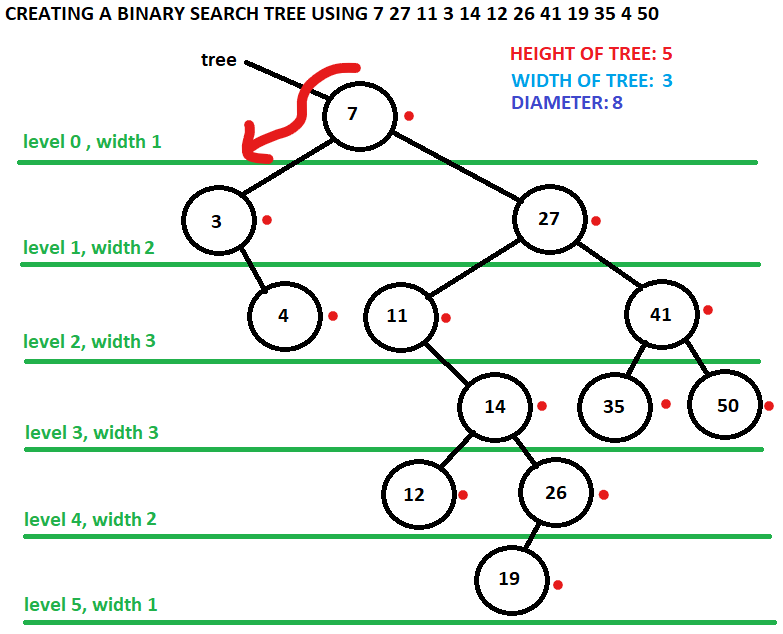
}

The first method is what a user outside of the binary search tree class would call. Then the method would call a private helper recursive method that is seeded with the root. When the helper method is called it will create a String to send back once the method call is done that will store the values. The base case of this recursive method is if the binary node k is null. If it is null, then temp sends back an empty String. If it is not null, the value is added to temp along with whatever is returned by the recursive call to preorder using the left child and whatever is returned by the recursive call to the right child. At the conclusion of the original preorder method call a String will be sent back composed of the preorder traversal of the binary search tree. The String is trimmed before it is sent back to get rid of the extra space that is included from the last occurrence of the concatenation of the getValue() method in the recursive call.

A preorder traversal has a couple of useful characteristics. A preorder traversal will always start with the root. A preorder traversal will always print the parent before it prints any of the parent’s children.

## The Postorder Traversal

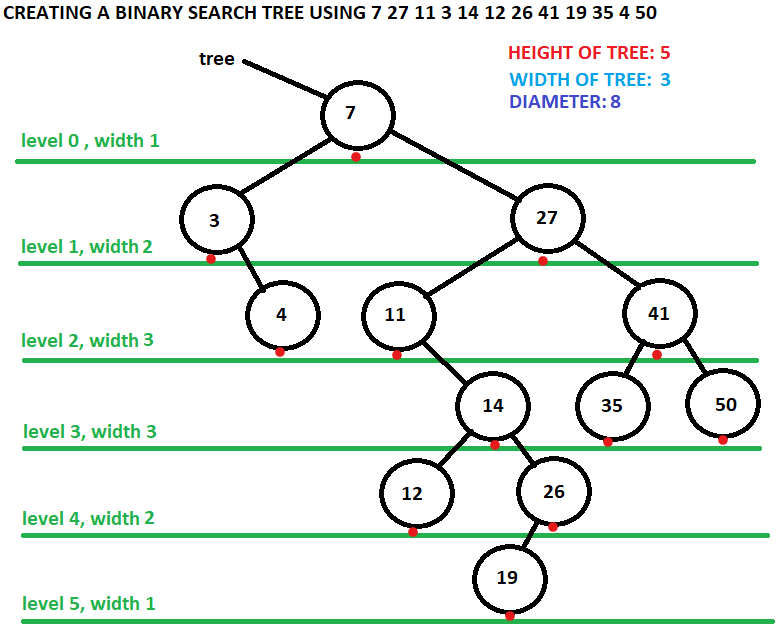
The postorder traversal works in the same manner as the preorder traversal, except that instead of printing the value first, it is printed after both children has been explored. When doing a preorder traversal by hand, place a dot to the right, and then trace the binary search tree counter clockwise and write the value down when the path passes through the dot. Consider the example below:



When the path is traced, the postorder traversal will print 4 3 12 19 26 14 11 35 50 41 27 7. A postorder traversal has a couple of useful characteristics. A postorder traversal will always end with the root node. A postorder traversal will always print both children before it prints out the children’s particular parent.

## The Inorder traversal

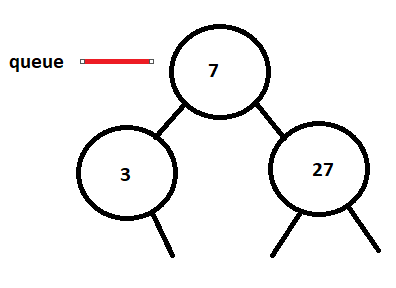
The inorder traversal works the same way as both the preorder and postorder traversal. However, in the case of the inorder traversal, the print occurs between the call for the inorder traversal of the left child and the call for the inorder traversal for the right child. This is one of the most useful traversals because when it is applied to a binary search tree that is correctly created it will generate a list of node values increasing based on its Comparative operation. In other words, the values will be in increasing order. When predicting the binary search tree inorder traversal, place the dot below the node and do a counter clockwise path around the binary tree. The inorder traversal of the example we have been doing would be 3 4 7 11 12 14 19 26 27 35 41 50.



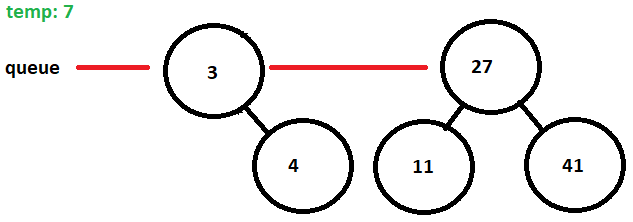
## The Level Order Traversal

The level order traversal is written to print out the binary search tree by level from root to deepest node, from left of the level to right of the level. The level order for the previous example would be 7 3 27 4 11 41 14 35 50 12 26 19. This traversal does not need to be written as a recursion although it could be if the programmer chooses to do so. The traversal does, however, use a queue to produce its result.

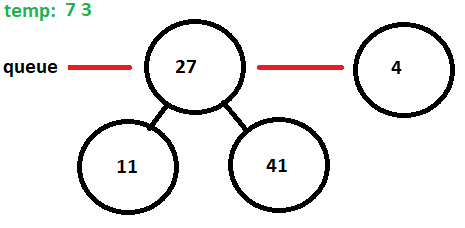
The algorithm of the level order traversal is to create a queue of the binary search tree and place the root into the queue. The programmer would also create an empty String variable called temp.



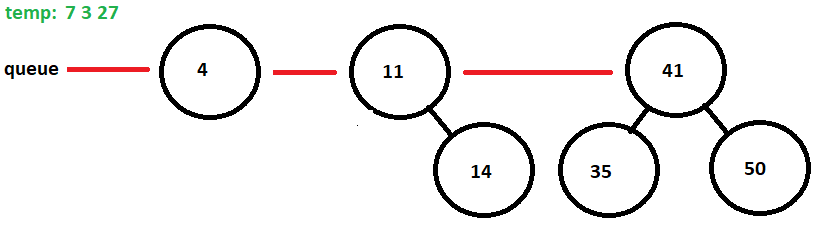
Once the queue is started, the programmer would write a while loop that would continue to go until there is nothing in the queue. The body of the loop would remove the front of the queue, add the value to the temp String variable and then place the left child, as long as it wasn’t a null, followed by the right child, as long as it wasn’t a null, back into the queue.



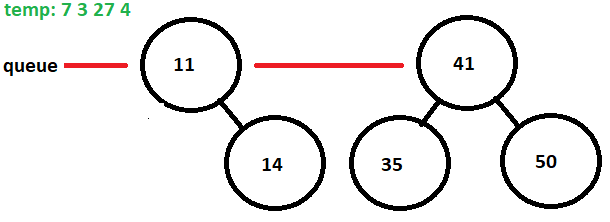
The node 7 was removed from the queue and the value was placed into the String temp. Then the left child 3 was placed into the queue followed by the right child.



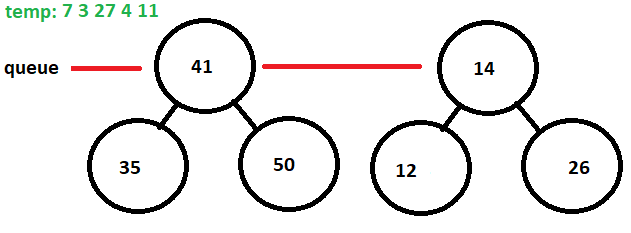
The 3 node was removed from the queue, the value was added to temp and then the right child was added to the queue. Since the left child was null, the left child was not added.



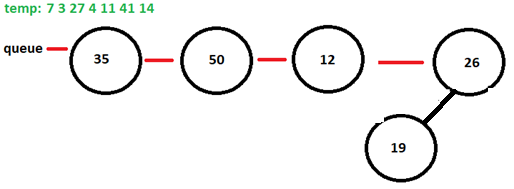
The 27 node was removed from the queue, the value was added to temp and then the left and right child were added to the queue.



The 4 node was removed from the queue and the value was placed in the temp. Nothing was added to the queue because the 4 node did not have a left nor a right child.



The 11 node is removed from the queue and the value was placed in the temp. There is no left child to add to the queue but the right child is added to the queue. This will continue until there is nothing left in the queue. The queue will remove the 41 node and add the 35 and 50 node at the end of the queue, causing the queue to be three nodes long. Then the 14 node will be removed and the 12 and 26 node will be add to the queue, causing the queue to be four nodes long.



At this point, the queue is down to mostly leaves. The 35, 50, and 12 node are removed and the values are placed into temp. The 26 node is removed from the queue and the value is placed into temp while its only child is placed into the queue. Finally 19 is removed from the queue and placed its value is placed into temp. Since the 19 node does not have a child, nothing is placed into the queue, and the queue is now empty. This ends the loop and the temp is returned.

Here is the code for a level order traversal:

public String levelOrder()

{

String temp = "";

Queue<BinaryNode> queue = new LinkedList<BinaryNode>();

queue.offer(root);

while(!queue.isEmpty())

{

BinaryNode k = queue.poll();

temp += k.getValue()+" ";

if(k.left()!=null)

queue.offer(k.left());

if(k.right()!=null)

queue.offer(k.right());

}

return temp.trim();

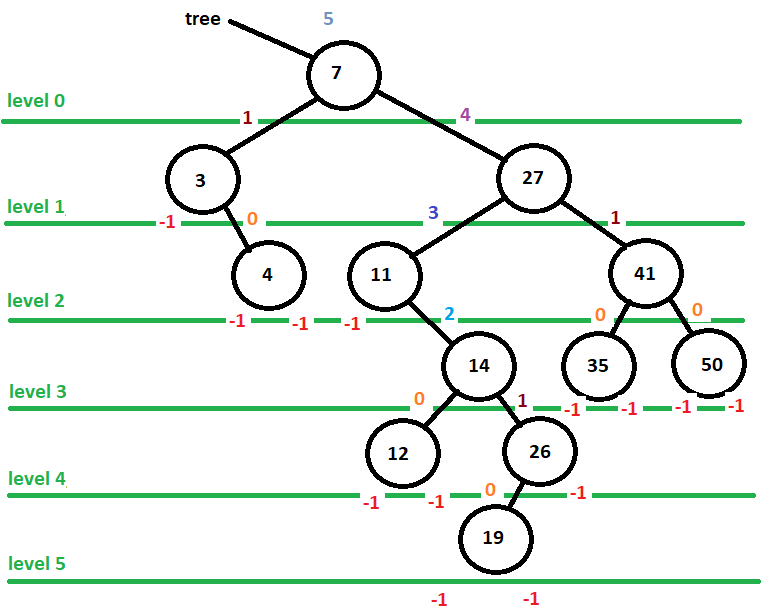
}

## The toString method

The toString method could choose to call any of the four traversal methods: inorder, preorder, postorder, or level order. It is most common for inorder to be called by the toString method, but it is not unheard of to have the level order method to be used.

## The height Method

The height method is a method that will return the height of the tree. Many methods will use recursion to get its solution. When writing a base case for the recursion, the programmer should consider two things: an empty subtree and a subtree that is a leaf only. The empty subtree has a height of -1 because there is no height. The leaf subtree has a height of zero because there are no edges but there is a node. Once the empty subtree and leaf subtree is defined, then the programmer can consider a subtree that has leaves as either of their children. If a subtree has two children who are leaves, then the subtree is of size 1. The programmer then extrapolates that any subtree has a height that is one greater than its highest child subtree. Here is a visualization of this concept, where each level of the subtree is counted and the height of the subtree is listed just above the root of the subtree branch. The null branches are represented with a -1.



The code for the height method using this algorithm is shown below. The programmer wrote a method that is available to the client to call. Inside that method, the programmer called a helper method that would run the algorithm recursively. The null base case is the key to the recursive call, every other node builds from the base case.

public int getHeight()

{

return getHeight(root);

}

private int getHeight(BinaryNode k)

{

if(k == null) return -1;

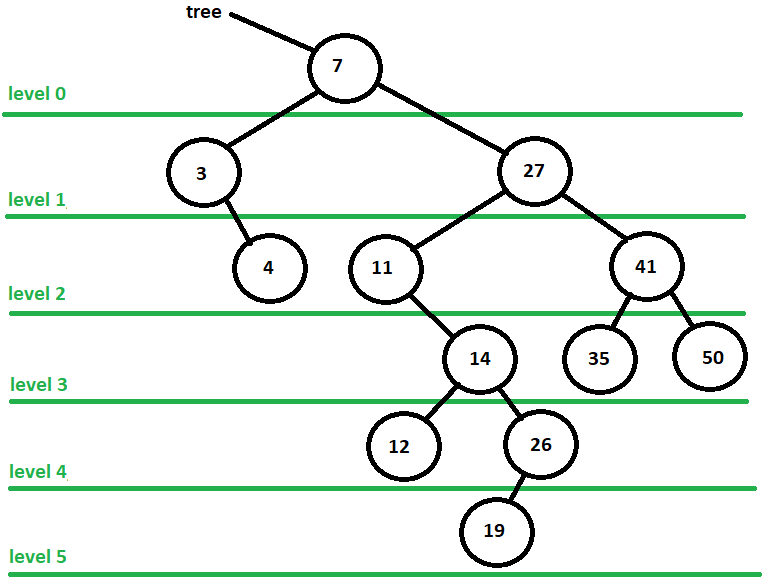
return 1 + Math.*max*(getHeight(k.left()),

getHeight(k.right()));

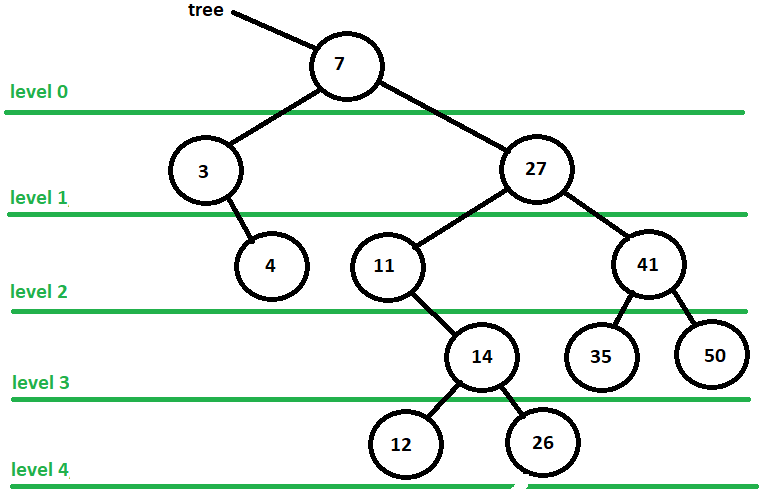
}

## Deletion -- Removing a Node from the Binary Search Tree by Hand

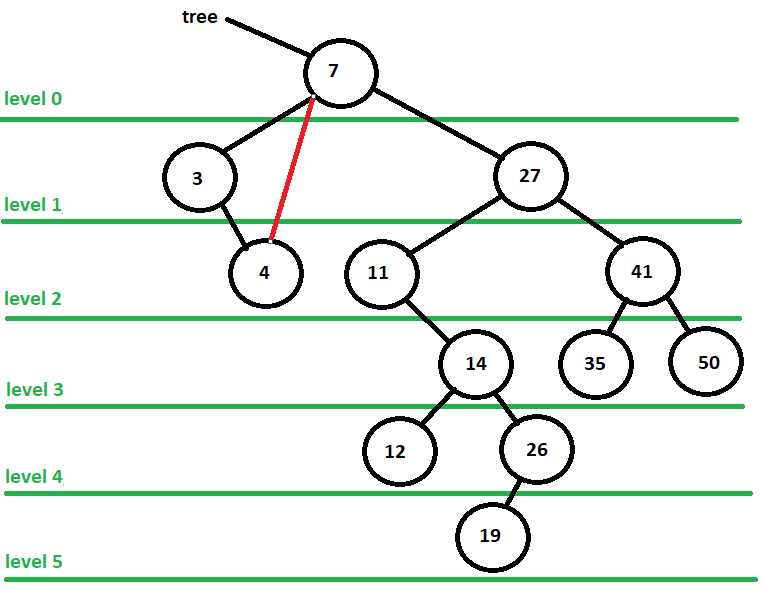
The binary search tree must be able to remove a binary node from itself as well as add a binary node to itself. When removing from a node from the binary search tree, the programmer needs to consider four possibilities: removing a node of degree zero, removing a node of degree one, removing a node of degree two, and the edge case of removing the root. When doing a removal by hand, the edge case is easily addressed, because the programmer simply assigns the root to whatever is replacing the root. However, the edge case will be more important when the programmer starts to write the remove method. Consider the binary search tree the programmer created earlier.



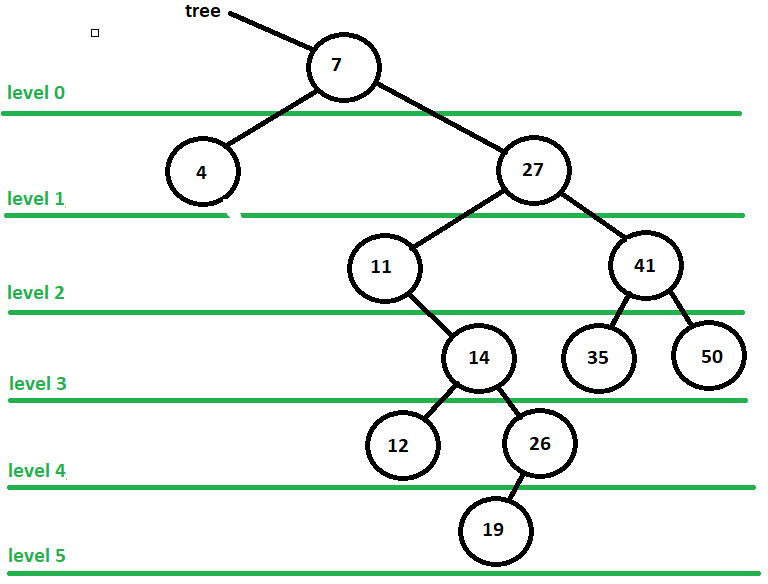
The easiest removal is to remove a leaf. This is because all the programmer needs to do is to point the parent to null. Removing the 19 node from the tree would result in the following binary search tree



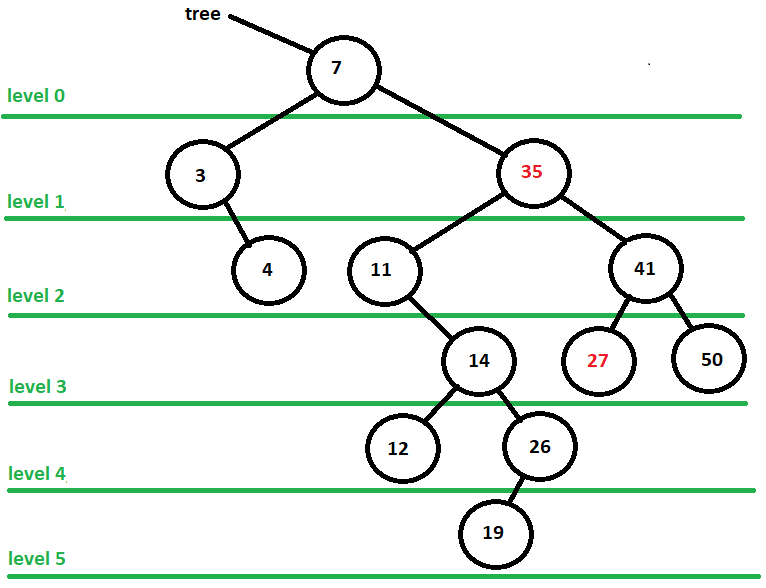
Removing a node of degree one is also straight forward because all the programmer needs to do is to have the parent point to the child of the node being removed. When looking at the original binary search tree, the programmer chooses to remove the 3 node. This will cause the parent to attach itself to the 4 node as it was the child of the 3 node. A red line has been drawn to indicate this change



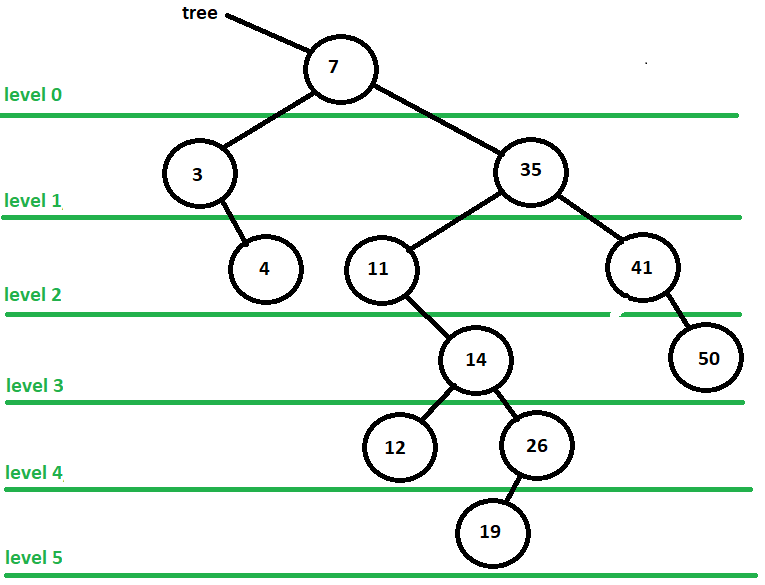
This would yield the following results.



Removing a node of degree 2 requires more thought. The node that is being removed would find either its inorder predecessor or its inorder successor and swap it. If the programmer wanted to remove the 27 node, then the programmer would either choose to swap it with its inorder predecessor, the node furthest right of the left subtree which is 26, or its inorder successor, the node furthest left of the right subtree which is 35. Traditionally, the choice would be the inorder successor. Since this is an introduction to binary search trees, the programmer always chooses the inorder successor. The programmer swaps the 35 node value with the 26 node value. Once the swap is made, the programmer needs to remove the node that was swapped. The programmer knows that the node is either a leaf or a parent with a right child and can remove the node with no troubles. The example below shows the swap using a red number to indicate which two node values were swapped.



Now all the programmer needs to do is remove the new 27 node. If there was a right child to the new 27 node, then the parent would have pointed to the right child, but the new 27 node was a leaf and there was no need. Here is the resulting binary search tree



## Deletion in a Binary Search Tree -- The remove Method

Removing from a binary search tree maybe very straight forward but implementing that into code can be very complex. The best way to show this method is to outline it and build it slowly. The first thing the programmer should do is address the root edge case because the binary search tree starts out as a root. The programmer will write a public remove method that will be available the client user and produce an outline inside of the code to work with. The programmer will also need to be aware of having to write several helper methods that will help remove function. The remove method will receive a Comparable that will be the value to remove and the remove method will return a binary node that will be node that is being removed. The node will be unattached to the binary search tree when it is returned. Here is the public methods basic outline

public BinaryNode remove(Comparable target)

{

//check to see if root is to be removed

//remove root degree 0

//remove root degree 1 – right child

//remove root degree 1 – left child

//remove root degree 2

//if root is not removed call remove helper method

}

At the beginning of the program the programmer would check to make sure the remove method wasn’t working with an empty tree and the check to see if the target was the root. The programmer would also assign a temporary binary node called temp to root so that if root was changed, then there would still be a pointer to the removed node.

public BinaryNode remove(Comparable target)

{

if(root == null) return null;

BinaryNode temp = root;

//check to see if root is to be removed

if(root.getValue().equals(target))

{

//remove root degree 0

//remove root degree 1 – right child

//remove root degree 1 – left child

//remove root degree 2

}

//if root is not removed call remove helper method

}

The programmer would then check to see if the root is degree 0 and if so set the root to null and return the node being removed. This is the easiest of the removals to do.

public BinaryNode remove(Comparable target)

{

if(root == null) return null;

BinaryNode temp = root;

//check to see if root is to be removed

if(root.getValue().equals(target))

{

//remove root degree 0

if(root.left() == null && root.right() == null)

{

root = null;

return temp;

}

//remove root degree 1 – right child

//remove root degree 1 – left child

//remove root degree 2

}

//if root is not removed call remove helper method

}

Once the leaf is eliminated, the programmer would check for degree 1 nodes. The programmer would first check to see if the root has a left child. If it does not, then, by process of elimination, the root must have a right child and it is assigned to be the new root. The old root is still being pointed to by temp. The root sets its right child to null and returns as the removed node. Then the programmer would check to see if the root has a right child. If it does not, then by process of elimination, the root must have a left child and it is assigned to be the new root. The old root sets its left child to null and returns as the removed node.

public BinaryNode remove(Comparable target)

{

if(root == null) return null;

BinaryNode temp = root;

//check to see if root is to be removed

if(root.getValue().equals(target))

{

//remove root degree 0

if(root.left() == null && root.right() == null)

{

root = null;

return temp;

}

//remove root degree 1 – right child

else if(root.left() == null)

{

root = root.right();

temp.setRight(null);;

return temp;

}

//remove root degree 1 – left child

else if(root.right() == null)

{

root = root.left();

temp.setLeft(null);

return temp;

}

//remove root degree 2

}

//if root is not removed call remove helper method

}

The programmer now writes what should occur if the root is a degree 2 node. This is where remove gets complex. Several helper methods need to be written in order to remove root as a degree 2 node. One method is written to find and return the inorder successor.

The successor method will receive the node whose value needs to be swapped. It will go to right child of this node and then proceed down the left branch of that node until it reaches a node that has no left child. That node will be the inorder successor. Here is the code to that method

private BinaryNode successor(BinaryNode k)

{

BinaryNode temp = k;

temp = temp.right();

while(temp.left() != null)

temp = temp.left();

return temp;

}

Another useful method to have would be a swap method that would allow two nodes to swap their values. Here is the code for the swap method

private void swap(BinaryNode x, BinaryNode y)

{

Comparable k = x.getValue();

x.setValue(y.getValue());

y.setValue(k);

}

Finally, there will need to be a recursive, private helper method called remove that will continually look for the target to remove until it finds it or proves it does not exist in the binary search tree.

The programmer will first find the inorder successor by calling the successor method and assigning it to a created binary node called inorder successor. The programmer then swaps the root value with the inorder successor value. The programmer then calls the remove helper method and sends the roots right child node along with the target value so that the program can find and remove the node that now holds the value to be remove. The programmer sends the right node of the root because the root’s value is no longer following the requirements of the binary search tree, and will not follow that requirement until the node that holds the target is found and removed. Here what the code will look like once these changes are applied. The programmer made sure to declare the inorderSuccessor at the top of the method.

public BinaryNode remove(Comparable target)

{

if(root == null) return null;

BinaryNode temp = root;

BinaryNode inorderSuccessor;

//check to see if root is to be removed

if(root.getValue().equals(target))

{

//remove root degree 0

if(root.left() == null && root.right() == null)

{

root = null;

return temp;

}

//remove root degree 1 – right child

else if(root.left() == null)

{

root = root.right();

temp.setRight(null);;

return temp;

}

//remove root degree 1 – left child

else if(root.right() == null)

{

root = root.left();

temp.setLeft(null);

return temp;

}

else

//remove root degree 2

{

inorderSuccessor = successor(root);

swap(root,inorderSuccessor);

return remove(root.right(),target);

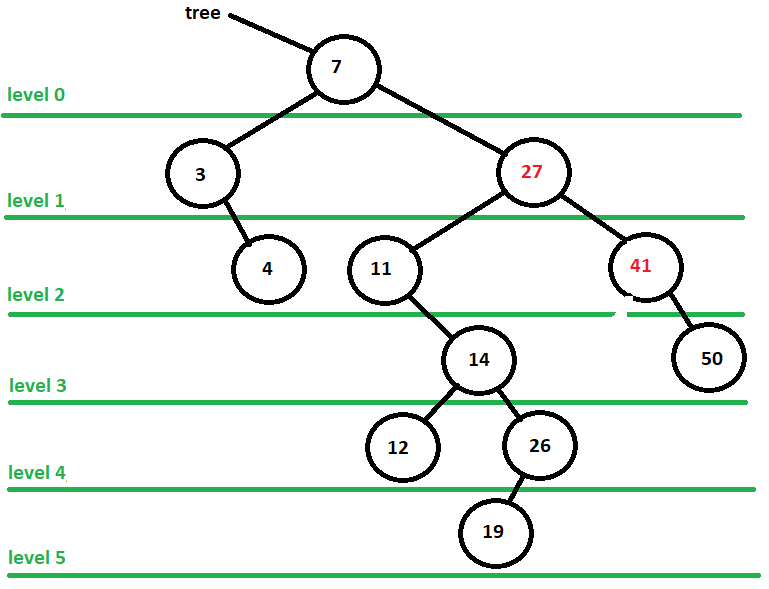
}

}

//if root is not removed call remove helper method

}

Removing from the a degree 2 root is not complete yet, however, because there is an edge case that will occur that will not allow the remove helper method to work in the expected way. The helper method is going to assume the incoming node is not the node to be removed, but it is possible that inorderSuccessor is root.right which means that it needs to be removed instead of calling the helper method. An example of the edge case, which is not a root in this situation, follows where 27 is being removed but the immediate successor is the right child of 27.



The 27 node value will swap with 41 node value, but if the helper remove method is called, then the right child node will be sent up and the remove helper method will not consider it a possible node to be removed and the node will stay in the tree.

The programmer can solve this problem by writing an edge case check to see if the inorderSuccessor is the right child of the node being removed. If it is the right child, then the programmer knows that the right child is node of degree 1 and that right child has a right child as well. The programmer can have the root’s right pointer set to the inorderSuccessor’s right child, disconnect the inorderSuccessor from its right child and return the inorderSuccessor instead of calling the helper return method. This is shown in the code below

public BinaryNode remove(Comparable target)

{

if(root == null) return null;

BinaryNode temp = root;

BinaryNode inorderSuccessor;

//check to see if root is to be removed

if(root.getValue().equals(target))

{

//remove root degree 0

if(root.left() == null && root.right() == null)

{

root = null;

return temp;

}

//remove root degree 1 – right child

else if(root.left() == null)

{

root = root.right();

temp.setRight(null);;

return temp;

}

//remove root degree 1 – left child

else if(root.right() == null)

{

root = root.left();

temp.setLeft(null);

return temp;

}

//remove root degree 2

{

inorderSuccessor = successor(root);

swap(root,inorderSuccessor);

if(root.right()==inorderSuccessor)

{

root.setRight(inorderSuccessor.right());

inorderSuccessor.setRight(null);

return inorderSuccessor;

}

return remove(root.right(),target);

}

}

//if root is not removed call remove helper method

}

The last piece of code the programmer needs to do is call the remove helper method in case the root is not being removed from the tree. When calling this method, the programmer sends the root and the target to the remove helper method because the programmer now knows the root will not be removed from the tree. The final look of the remove method is below.

public BinaryNode remove(Comparable target)

{

if(root == null) return null;

BinaryNode temp = root;

BinaryNode inorderSuccessor;

//check to see if root is to be removed

if(root.getValue().equals(target))

{

//remove root degree 0

if(root.left() == null && root.right() == null)

{

root = null;

return temp;

}

//remove root degree 1 – right child

else if(root.left() == null)

{

root = root.right();

temp.setRight(null);;

return temp;

}

//remove root degree 1 – left child

else if(root.right() == null)

{

root = root.left();

temp.setLeft(null);

return temp;

}

//remove root degree 2

{

inorderSuccessor = successor(root);

swap(root,inorderSuccessor);

if(root.right()==inorderSuccessor)

{

root.setRight(inorderSuccessor.right());

inorderSuccessor.setRight(null);

return inorderSuccessor;

}

return remove(root.right(),target);

}

}

//if root is not removed call remove helper method

return remove(root,target);

}

The remove helper method is then written. The remove helper method receives a binary node that is called startNode and a target value to remove. This method is similar to the remove method except that in order to remove a node from the tree, the programmer needs to know the parent of the node in order to connect the parent to the removed nodes child. A helper search method is written to find this parent. The search method will receive a binary node that it will consider the parent and the target value that is being searches. The search method will then check to see if parent is null. If it is, then that means the target is not found in the tree and the search should return null, this is the base case of the recursion. The search then checks to see if the parent node has a left child or a right child that if the target value. When checking these two children, the method must check to see if the child exists first before checking to see if it has the target value. If the parent has a child that matches the target, the parent is returned because the remove will need to know parent of the node to be removed. If it does not have a child that matches, then the method makes a recursive call by checking to see if the target could be on the left child’s branch or on the right child’s branch. Here is the code for the search

private BinaryNode search(BinaryNode parent, Comparable target)

{

if(parent == null) return null;

if(parent.left()!=null && parent.left().getValue().equals(target)||

parent.right()!=null && parent.right().getValue().equals(target))

return parent;

else if(target.compareTo(parent.getValue()) < 0)

return search(parent.left(),target);

else

return search(parent.right(),target);

}

The programmer can now write the remove helper method. The remove helper method will create a binary node called nodeToRemove which will indicate which node the method wants to remove and a binary node called inorderSuccessor. The method will then create a node that will hold the parent that is returned by the search method. Before going any further, the method will check to see if the parent is null. If it is, then the method ends because the target is not in the binary search tree. This is the base case for the recursive method.

The method needs to assign the nodeToRemove to the target node. The method also needs to know if the nodeToRemove is a left or right child. A Boolean variable isLeft is created. If the nodeToRemove is a left child, then isLeft will be true. If isLeft is false, then the nodeToRemove is a right child. When checking to see if the nodeToRemove is left child, the method should check to see if the left child exists before it checks to see if the target matches the value. Once the child is determined, the method can assign nodeToRemove to the proper child. In the example below, the programmer used the comparative operator ?: to assign the node.

private BinaryNode remove(BinaryNode startNode, Comparable target)

{

BinaryNode nodeToRemove, inorderSuccessor;

BinaryNode parent = search(startNode,target);

if(parent == null) return null;

//decide if it is a left or right child

boolean isLeft = parent.left()!=null &&

parent.left().getValue().equals(target);

nodeToRemove = isLeft ? parent.left() : parent.right();

//degree 0

//degree 1

//degree 2

}

The rest of the method is written like the original remove method, except that the programmer must always check to see if the parent is dealing with a left child or a right child. The programmer cannot use the comparative operator ?: because that can only be used with assignment statements, so they will have to use the if…else structure. The code for the degree method will use the if…else structure in a straight forward manner, choosing which child will be set to null.

//degree 0

if(nodeToRemove.left() == null && nodeToRemove.right() == null)

{

if(isLeft)

parent.setLeft(null);

else

parent.setRight(null);

return nodeToRemove;

}

When checking the degree 1 nodes, the programmer must be very careful. The programmer checks to see if the left node is null, indicating that there is only a right node. Then the programmer checks to see if the parent is removing a left child or a right child. When that child is set to the deleted node’s child, it will always be the deleted node’s right child.

When the deleting node only has a left child, then no matter what child is being removed from the parent, it is always replaced by the deleted node’s left child. The deletion of degree 1 nodes are shown below

//degree 1

else if(nodeToRemove.left() == null)

{

if(isLeft)

parent.setLeft(nodeToRemove.right());

else

parent.setRight(nodeToRemove.right());

nodeToRemove.setRight(null);

return nodeToRemove;

}

else if(nodeToRemove.right() == null)

{

if(isLeft)

parent.setLeft(nodeToRemove.left());

else

parent.setRight(nodeToRemove.left());

nodeToRemove.setLeft(null);

return nodeToRemove;

}

When the node is removing a degree 2 node, the parent is not necessary because there will be a swapping occurring between the nodeToRemove and the inorderSuccessor. It is the same as the remove method, except that root.right is replaced with nodeToRemove.right. Here is the code that completes the helper remove method.

//degree 2

else

{

inorderSuccessor = successor(nodeToRemove);

swap(inorderSuccessor, nodeToRemove);

if(nodeToRemove.right()==inorderSuccessor)

{

nodeToRemove.setRight(inorderSuccessor.right());

inorderSuccessor.setRight(null);

return inorderSuccessor;

}

return remove(nodeToRemove.right(), target);

}

With the completion of the binary search tree’s deletion method, remove, the programmer now has a robust structure to store large quantities of data.