# Binary Trees and Binary Search Trees

Trees are a very common data structure in computer science. A tree is a nonlinear data structure that is used to store data in a hierarchical manner. We examine one particular tree structure in this chapter – the *binary tree*. Binary trees are chosen over other more primary data structures because you can search a binary tree very quickly (as opposed to a linked list, for example) and you can quickly insert and delete data from a binary tree (as opposed to an array).

## Trees Defined

A tree is made up of a set of *nodes* connected by *edges*. An example of a tree is a company's organizational chart (see Figure x.1).

The purpose of an organizational chart is to communicate the structure of an organization. In Figure x.1, each box is a node and the lines connecting the boxes are the edges. The nodes represent the positions that make up an organization and the relationships between those positions. For example, the CIO reports directly to the CEO, so there is an edge between those two nodes. The Development Manager reports to the CIO, so there is an edge connecting those two positions. The VP of Sales and the Development Manager do not have a direct edge connecting them, so there is not a direct relationship between those two positions.

Figure x.2 displays another tree that defines more of the terms we need when discussing trees. The top node of a tree is called the *root* node. If a node is connected to other nodes below it, the top node is called the *parent*, and the nodes below it are called the parent's *children*. A node can have zero, one, or more child nodes connected to it. A node without any child nodes is called a *leaf*.

Special types of trees, called binary trees, restrict the number of children to no more than two. Binary trees have certain computational properties that make them very efficient for many operations. Binary trees are discussed extensively in the sections to follow.

Continuing to examine Figure x.2, you can see that by following certain edges, you can travel from one node to other nodes that are not directly connected. The series of edges you follow to get from one node to another is called a *path* (depicted in the figure with dashed lines). Visiting all the nodes in a tree in some particular order is known as a tree *traversal*.

A tree can be broken down into *levels*. The root node is at level 0, its children are at level 1, those node's children are at level 2, and so on. A node at any level is considered the root of a *subtree*, which consists of that root node's children, it's children's children, and so on. We can define the depth of a tree as the number of layers in the tree.

Finally, each node in a tree has a value. This value is sometimes referred to as the *key* value.

## Binary Trees and Binary Search Trees

A binary tree is defined as a tree where each node can have no more than two children. By limiting the number of children to two, we can write efficient programs for inserting data, deleting data, and searching for data in a binary tree.

Before we discuss building a binary tree in JavaScript, we need to add two terms to our tree lexicon. The child nodes of a parent node are referred to as the *left* node and the *right* node. For certain binary tree implementations, certain data values can only be stored in left nodes and other data values must be stored in right nodes. An example binary tree is shown in Figure x.3.

Identifying the child nodes is important when we consider a more specific type of binary tree – the *binary search tree*. A binary search tree is a binary tree in which data with lesser values are stored in left nodes and data with greater values are stored in right nodes. This property provides for very efficient searches, as we shall soon see.

## Building a Binary Search Tree

A binary search tree is made up of nodes, so we need a Node object that is similar to the Node object we used for linked lists. Here is the definition of the Node object for binary search trees:

function Node(data, left, right) {

this.data = data;

this.left = left;

this.right = right;

this.show = show;

}

function show() {

return this.data;

}

The Node object stores both data and links to other nodes (left and right). We also provide a method, show(), for displaying the data being stored in a node.

Now we're ready to build a class to represent a binary search tree (BST). The class consists of just one data member – a Node object that represents the root node of the BST. The constructor for the class sets the root node to null, creating an empty node.

The first method we need for the BST class is insert() to add new nodes to the tree. This method is somewhat complex and will require some explanation. The first step in the method is to create a Node object, passing in the data the node is storing.

The next step in insertion is to check whether our BST has a root node. If there is not already a root node, then this is a new BST and the node we are inserting is the root node and the method is finished. Otherwise, the method moves on to the next step.

If the node being inserted is not the root node, then we have to prepare to traverse the BST to find the proper insertion point. This process is similar to traversing a linked list. The method uses a Node object that we can assign as the current node as the method moves from level to level. The method also has to position itself inside the BST at the root node.

Once we're inside the BST, the next step is to determine where to put the new node. This performed inside a loop that we break once we've determined the correct insertion point. The algorithm for determining the proper insertion position for a node is as follows:

1. Set the parent node to be the current node, which is the root node.
2. If the data value in the new node is less than the data value in the current node, set the current node to be the left child of the current node. If the data value in the new node is greater than the data value in the current node, skip to step 4.
3. If the value of the left child of the current node is null, insert the new node here and exit the loop. Otherwise, skip to the next iteration of the loop.
4. Set the current node to the right child node of the current node.
5. If the value of the right child of current node is null, insert the new node here and exit the loop. Otherwise, skip to the next iteration of the loop.

With this algorithm complete, we're ready to implement this part of the BST class. Here is the code, including the code for the Node object:

function Node(data, left, right) {

this.data = data;

this.left = left;

this.right = right;

this.show = show;

}

function show() {

return this.data;

}

function BST() {

this.root = null;

this.insert = insert;

this.inOrder = inOrder;

}

function insert(data) {

var n = new Node(data, null, null);

if (this.root == null) {

this.root = n;

}

else {

var current = this.root;

var parent;

while (true) {

parent = current;

if (data < current.data) {

current = current.left;

if (current == null) {

parent.left = n;

break;

}

}

else {

current = current.right;

if (current == null) {

parent.right = n;

break;

}

}

}

}

}

### Traversing a Binary Search Tree

We now have the beginnings of a BST class, but all we can do so far is insert nodes into the tree. We need to able to traverse the BST so that we can visit the nodes in different orders for the purpose of displaying the data stored in the tree.

There are three traversal methods used with BSTs: *inorder*, *preorder*, and *postorder*. An inorder traversal visits all the nodes in a BST in ascending order of the node key values. A preorder traversal visits the root node first, followed by the nodes in the subtrees under the left child of the root, followed by the nodes in the subtrees under the right child of the root node. A postorder traversal visits all the child nodes of the left subtree up to the root node and then all the child nodes of the right subtree up to the root node. The paths for the different traversal methods are shown in Figure x.4.

Although it's easy to understand why we would want to perform an inorder traversal, it is less obvious why we need preorder and postorder traversals. We'll implement all three traversals now and then explain their uses in a later section.

The inorder traversal is best written as a recursive procedure. Since the method visits each node in ascending order, the method must visit both the left node and the right node of each subtree, following the subtrees under the left child of the root node before following the subtrees under the right child of the root node.

Here is the code for the inorder traversal method:

function inOrder(node) {

if (!(node == null)) {

inOrder(node.left);

putstr(node.show() + " ");

inOrder(node.right);

}

}

Here is a short BST program to test our inorder() method:

var nums = new BST();

nums.insert(23);

nums.insert(45);

nums.insert(16);

nums.insert(37);

nums.insert(3);

nums.insert(99);

nums.insert(22);

print("Inorder traversal: ");

inOrder(nums.root);

The output from this program is:

Inorder traversal:

3 16 22 23 37 45 99

Figure x.5 shows the path the inorder() method followed.

Now let's define the method for preorder traversal:

function preOrder(node) {

if (!(node == null)) {

putstr(node.show() + " ");

preOrder(node.left);

preOrder(node.right);

}

}

You'll notice that the only difference between the inOrder() and preOrder() methods is how the three lines of code inside the if statement are placed. The call to the show() method is sandwiched between the two recursive calls in the inOrder() method and the call to show() is the first line in the preOrder() method.

If we add a call to preOrder() to the program above and run the code, we get the following output:

Inorder traversal:

3 16 22 23 37 45 99

Preorder traversal:

23 16 3 22 45 37 99

Figure x.6 illustrates the preorder traversal path.

Finally, here is the implementation of the postOrder traversal method:

function postOrder(node) {

if (!(node == null)) {

postOrder(node.left);

postOrder(node.right);

putstr(node.show() + " ");

}

}

And here is the output when adding the method to our program:

Inorder traversal:

3 16 22 23 37 45 99

Preorder traversal:

23 16 3 22 45 37 99

Postorder traversal:

3 22 16 37 99 45 23

The postorder traversal path is illustrated in Figure x.7.

We’ll look at some practical programming examples using BSTs that make use of these traversal methods later in this chapter.

## BST Searches

There are three types of searches typically performed with a BST: 1) searching for a specific value; 2) searching for the minimum value; and 3) searching for the maximum value. We explore these three searches in the sections below.

### Searching for the Minimum and Maximum Value

Searching a BST for the minimum and maximum values stored in a BST are relatively simple procedures. Since lower values are always stored in left child nodes, to find the minimum value in the BST, continually travel down the left edge of the BST and when you get to the last node, you will be at the minimum value. Figure x.8 illustrates how to find the minimum value in a BST.

Here is a definition for the getmin() method of the BST class:

function getMin() {

var current = this.root;

while (!(current.left == null)) {

current = current.left;

}

return current.data;

}

The method travels down the left link of each node in the BST until it reaches the left end of the BST, which is defined as current.left = null. When this point is reached, the data stored in the current node must be the minimum value.

To find the maximum value, we simply travel down the path of the right links of nodes until we reach the right end of the BST. The value stored in this node must be the maximum value. Figure x.8 illustrates searching for the maximum value.

Here is the definition for the getmax() method:

function getmax() {

var current = this.root;

while (!(current.right == null)) {

current = current.right;

}

return current.data;

}

Let's test these two methods with the BST data we used earlier:

var nums = new BST();

nums.insert(23);

nums.insert(45);

nums.insert(16);

nums.insert(37);

nums.insert(3);

nums.insert(99);

nums.insert(22);

var min = nums.getmin();

print("The minimum value of the BST is: " + min);

print("\n");

var max = nums.getmax();

print("The maximum value of the BST is: " + max);

Here is the output from this program:

The minimum value of the BST is: 3

The maximum value of the BST is: 99

Both of these methods return the data stored in the minimum and maximum positions, respectively. We may, instead, want the methods to just return the nodes where the minimum and maximum values are stored. To make that change, just have the methods return the current node, rather than the value stored in the current node.

### Searching for a Specific Value

Searching for a specific value in a BST requires that a comparison be made between the data stored in the current node and the value we are searching for. The comparison will determine if we travel to the left child node or the right child node if the current node doesn't store the specified value. Figure x.10 demonstrates how the search progresses for the number 45 in the BST.

Here is our definition of the find() method:

function find(data) {

var current = this.root;

while (current.data != data) {

if (data < current.data) {

current = current.left;

}

else {

current = current.right;

}

if (current == null) {

return null;

}

}

return current;

}

This method returns the node if the value is found in the BST and returns null if the value was not found.

Here is a program to test the find() method:

var nums = new BST();

nums.insert(23);

nums.insert(45);

nums.insert(16);

nums.insert(37);

nums.insert(3);

nums.insert(99);

nums.insert(22);

inOrder(nums.root);

print("\n");

putstr("Enter a value to search for: ");

var value = parseInt(readline());

var found = nums.find(value);

if (found != null) {

print("Found " + value + " in the BST.");

}

else {

print(value + " was not found in the BST.");

}

Here are the results of running this program:

3 16 22 23 37 45 99

Enter a value to search for: 23

Found 23 in the BST.

## Removing Nodes

The most complex operation on BSTs is deleting a node. The complexity of the deletion depends on what node you are trying to delete. If you are trying to delete a node with no children, the deletion is actually fairly simple. If the node has just one child node, either left or right, the deletion is a little more complex to accomplish. If, on the other hand, the node has two children, then its deletion is the most complex operation to perform.

To aid in managing the complexity of the operation, we perform deletion recursively. Before we begin defining the method, we have to mention that we can't name the method delete because delete is a reserved word in JavaScript, used to delete an object's property. So we will use the names remove and removeNode for the methods used to delete nodes from a BST.

The basic outline we follow when deleting BST nodes is to first check to see if the current node holds the data we are trying to remove. If so, delete that node. If not, then we compare the data in the current node to the data we are trying to remove. If the data to be removed is less than the data in the current node, move to the left child of the current node. If the data to be removed is greater than the data in the current node, move to the right child of the current node.

The first case to consider is when the node to be removed is a leaf (a node with no children). Then all we have to do is set the appropriate link that is pointing to the node of the parent node to null. This scenario is illustrated in Figure x.11.

When the node to be removed has one child, then the link that is pointing the node to be removed needs to be adjusted to point to the removed node's child node. This scenario is illustrated in Figure x.12.

Finally, when the node to be removed has two children, the correct solution is to either find the largest value in the subtree to the left of the removed node, or to find the smallest value in the subtree to the right of the removed node. We will choose to go to the right.

We need a method that finds the smallest value of a subtree and then we use it to create a temporary node containing that smallest value. We copy that value into the position of the node that we are replacing and we delete the temporary node to complete the operation. Figure x.13 illustrates this scenario.

The deletion process actually consists of two methods. The remove() method simply takes the value to be removed and then calls removeNode(), which does all the work. Here are the definitions of the two methods, with comments noting the code for the three possible deletion scenarios:

function remove(data) {

root = removeNode(this.root, data);

}

function removeNode(node, data) {

if (node == null) {

return null;

}

if (data == node.data) {

// node has no children

if (node.left == null && node.right == null) {

return null;

}

// node has no left child

if (node.left == null) {

return node.right;

}

// node has no right child

if (node.right == null) {

return node.left;

}

// node has two children

var tempNode = getSmallest(node.right);

node.data = tempNode.data;

node.right = removeNode(node.right, tempNode.data);

return node;

}

else if (data < node.data) {

node.left = removeNode(node.left, data);

return node;

}

else {

node.right = removeNode(node.right, data);

return node;

}

}

## Counting Occurrences

One use of a BST is to keep track of the occurrences of data in a data set. For example, we can use a BST to record the distribution of grades on an exam. Given a set of exam grade, we can write a program that checks to see if the grade is in the BST, adding it to the BST if it is not, and just incrementing the number of occurrences of the grade if the grade is found in the BST.

It becomes immediately apparent that we need to modify the Node object to include a field to keep track of the number of occurrences of a grade in the BST, and we also need a method for updating a node so that if we find a grade in the BST, we can increment the number of occurrences.

Let's start by modifying our definition of the Node object to include a field for keeping track of the number of occurrences:

function Node(data, left, right) {

this.data = data;

this.count = 1;

this.left = left;

this.right = right;

this.show = show;

}

When a new grade (a Node object) is inserted, its count is set at 1. The insert() method will work fine as is, but we need to add a method to update the BST when we need to increment the count field of a grade node. Here is the definition of update():

function update(data) {

var grade = this.find(data);

grade.count++;

return grade;

}

All the other methods of the BST class are fine as is. We just need a couple of functions to generate a set of grades and display the grades:

function prArray(arr) {

putstr(arr[0].toString() + ' ');

for (var i = 1; i < arr.length; ++i) {

putstr(arr[i].toString() + ' ');

if (i % 10 == 0) {

putstr("\n");

}

}

}

function genArray(length) {

var arr = [];

for (var i = 0; i < length; ++i) {

arr[i] = Math.floor(Math.random() \* 101);

}

return arr;

}

Here is a program to test everything out:

// main program

var grades = genArray(100);

prArray(grades);

var gradedistro = new BST();

for (var i = 0; i < grades.length; ++i) {

var g = grades[i];

var grade = gradedistro.find(g);

if (grade == null) {

gradedistro.insert(g);

}

else {

gradedistro.update(g);

}

}

var cont = "y";

while (cont == "y") {

putstr("\n\nEnter a grade: ");

var g = parseInt(readline());

var aGrade = gradedistro.find(g);

if (aGrade == null) {

print("No occurrences of " + g);

}

else {

print("Occurrences of " + g + ": " + aGrade.count);

}

putstr("Look at another grade (y/n)? ");

cont = readline();

}

Here is the output from one run of the program:

25 32 24 92 80 46 21 85 23 22 3

24 43 4 100 34 82 76 69 51 44

92 54 1 88 4 66 62 74 49 18

15 81 95 80 4 64 13 30 51 21

12 64 82 81 38 100 17 76 62 32

3 24 47 86 49 100 49 81 100 49

80 0 28 79 34 64 40 81 35 23

95 90 92 13 28 88 31 82 16 93

12 92 52 41 27 53 31 35 90 21

22 66 87 80 83 66 3 6 18

Enter a grade: 78

No occurrences of 78

Look at another grade (y/n)? y

Enter a grade: 65

No occurrences of 65

Look at another grade (y/n)? y

Enter a grade: 23

Occurrences of 23: 2

Look at another grade (y/n)? y

Enter a grade: 89

No occurrences of 89

Look at another grade (y/n)? y

Enter a grade: 100

Occurrences of 100: 4

Look at another grade (y/n)? n

## Exercises

1. Add a method to the BST class that counts the number of nodes in a BST.
2. Add a method to the BST class that counts the number of edges in a BST.
3. An arithmetic expression such as 1+2\*3 can be stored in a BST so that the expression can be correctly evaluated. Write a program that uses a BST to correctly evaluate arithmetic expressions.
4. Write a program that stores the words from a large text file in a BST and displays the number of times each word occurs in the text.