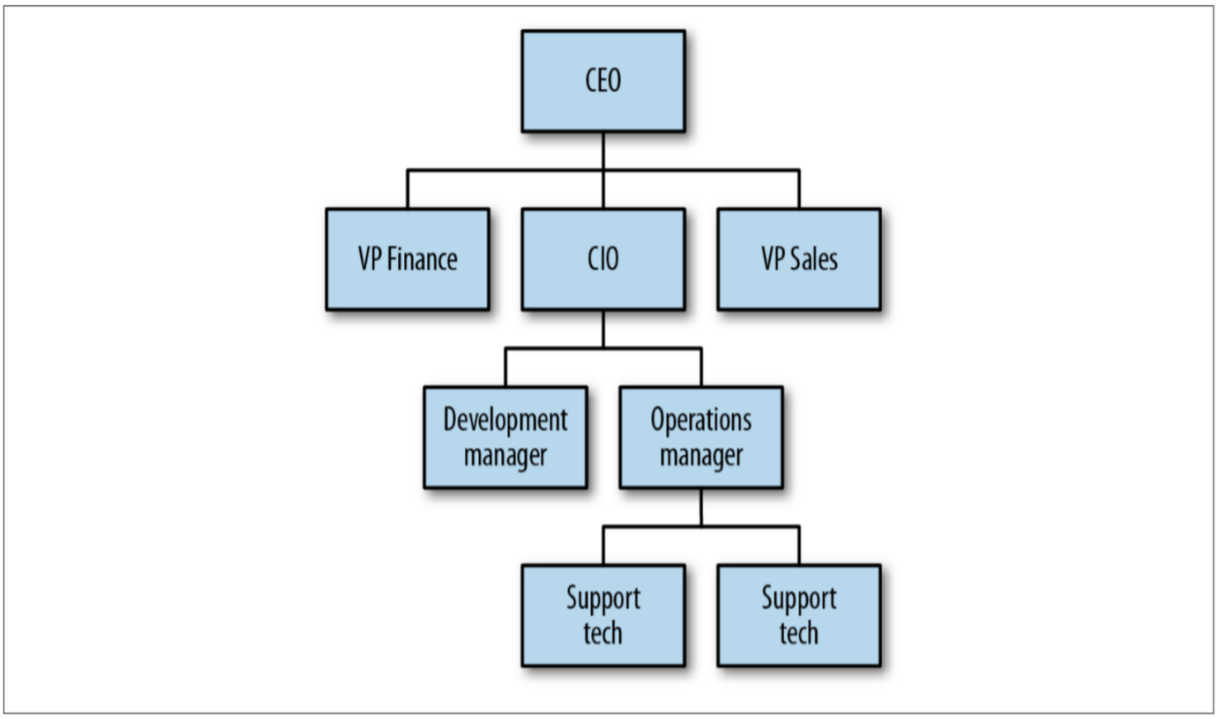
09. DATASTRUCTS Binary Trees.

Trees are a commonly used data structure in computer science. A tree is a nonlinear data structure that is used to store data in a hierarchical manner. Tree data structures are used to store hierarchical data, such as the files in a file system, and for storing sorted lists of data. 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).

9.1. 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 is shown next.

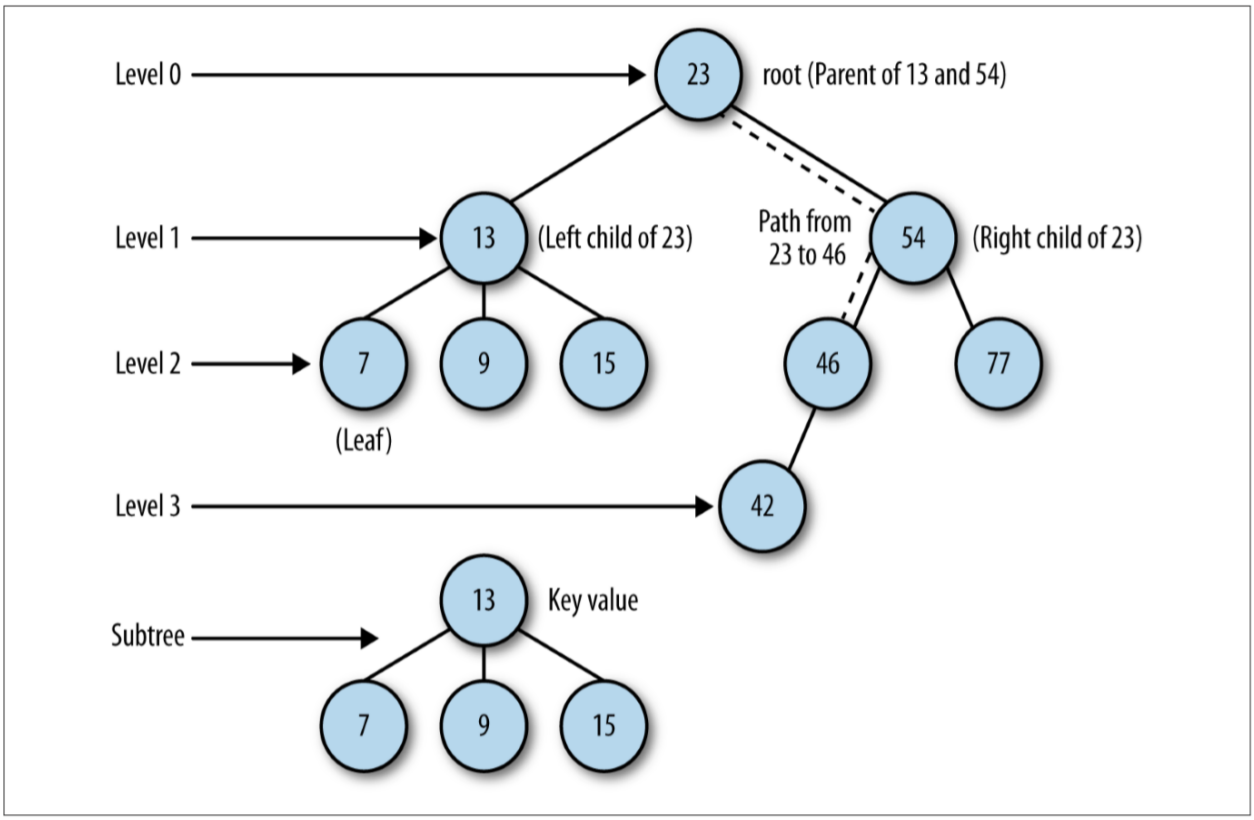


An organizational chart is a tree structure

The previous figure 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 preceding node is called the parent node, and the nodes following it are called child nodes. A node can have zero, one, or more child nodes connected to it. A node without any child nodes is called a leaf node.

Special types of trees, called **binary trees, restrict the number of child nodes to no more than two.** Binary trees have certain computational properties that make them very efficient for many operations.

Continuing to examine the next figure, 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 node is called a path. Paths are depicted in the figure with dashed lines. Visiting all the nodes in a tree in some particular order is known as a tree traversal.



The parts of a tree

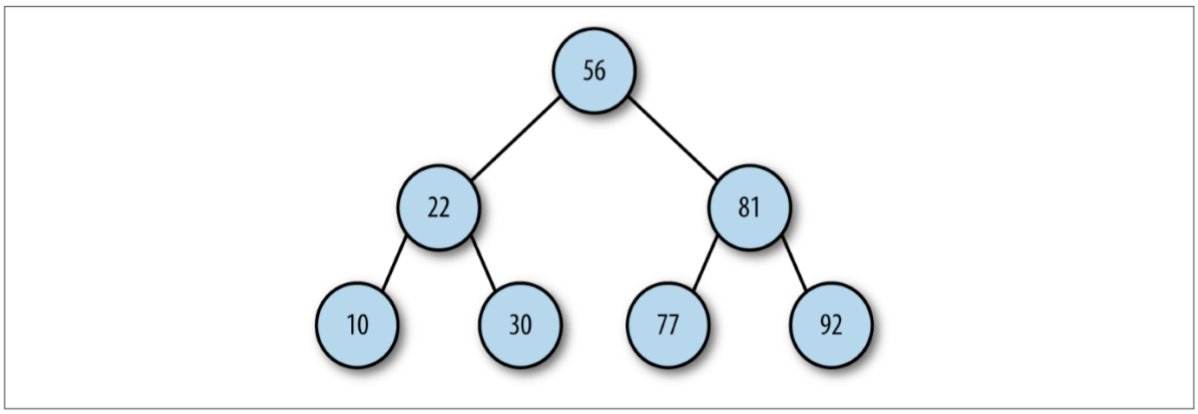
A tree can be broken down into levels. The root node is at level 0, its children are at level 1, those nodes' children are at level 2, and so on. A node at any level is considered the root of a subtree, which consists of that root node's children, its children's children, and so on. We can define the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

This concept of the root node being at the top of a tree, while in real life a tree's root is at the bottom of the tree, is counterintuitive, but it is a time-honored convention in computer science to draw trees with the root at the top. The computer scientist Donald Knuth actually tried to change the convention but gave up after a few months when he discovered that most computer scientists refused to adapt to the natural way of drawing trees.

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

9.2. Binary Trees and Binary Search Trees .

As mentioned earlier, a binary tree is one 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, searching for data, and deleting data in a 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 be stored only in left nodes, and other data values must be stored in right nodes. An example binary tree is shown in the next figure.



A binary tree

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 and holds for both numeric data and non-numeric data, such as words and strings.

9.3. Building a Binary Search Tree Implementation.

A binary search tree is made up of nodes, so the first object we need to create is a Node object, which is similar to the Node object we used with linked lists. The definition for the Node class is:

Example: *Node* class constructor:

```
class Node {

   constructor(data, left, right){
      this.data = data;
      this.left = left;
      this.right = right;
   }

   show(){
      return this.data;
   }
}
```

The *Node* object stores both data and links to other nodes (left and right). There is also a show() function for displaying the data stored in a node. Now we can 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 function we need for the BST is insert(), to add new nodes to the tree. This function is complex and requires explanation. The first step in the function is to create a Node object, passing in the data the node will store. The second step in insertion is to check the BST for a root node. If a root node doesn't exist, then the BST is new and this node is the root node, which completes the function definition. Otherwise, the function moves 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 function uses a Node object that is assigned as the current node as the function moves from level to level in the BST. The function also has to position itself inside the

BST at the root node. Once inside the BST, the next step is to determine where to put the node. This is performed inside a loop that breaks once the correct insertion point is determined.

The algorithm for determining the current insertion point for a node is as follows:

1. Set the root node to be the current node.
2. If the data value in the inserted node is less than the data value in the current node, set the new current node to be the left child of the current node. If the data value in the inserted 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 be the right child of the current node.
5. If the value of the right 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.

With this algorithm complete, we're ready to implement this part of the BST class. Example 10-1 has the code for the class, including the code for the Node object.

Example: The *BST* class:

```
class BST {

   constructor(){
      this.root = null;
   }

   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;
                }
            }
        }
    }

}
```

9.4. Traversing a Binary Search Tree.

We now have the beginnings of the BST class, but all we can do is insert nodes into the tree. We need to be able to traverse the BST so that we can display the data in different orders, such as numeric or alphabetic order.

There are three traversal functions used with BSTs: inorder, preorder, and postorder. An inorder traversal visits all of the nodes of 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 node, followed by the nodes in the subtrees under the right child of the root node. A postorder traversal visits all of the child nodes of the left subtree up to the root node, and then visits all of the child nodes of the right subtree up to the root node.

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 traversal functions now and explain their uses in a later section.

The inorder traversal is best written using recursion. Since the function visits each node in ascending order, the function 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.

Example: Here is the code for the inorder traversal function:

```
inOrder(node) {
   if (!(node == null)) {
      inOrder(node.left);
      console.log(node.show() + " ");
      inOrder(node.right);
   }
}
```

The next code provides a short program to test the function.

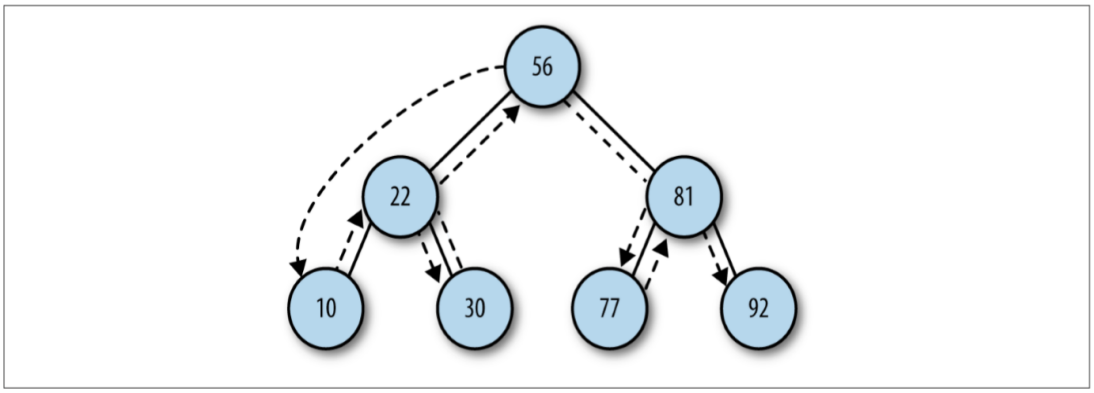
Example: Inorder traversal of a BST

```
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);

console.log("Inorder traversal: ");
inOrder(nums.root);
```

The next figure illustrates the path the *inOrder*() function followed.



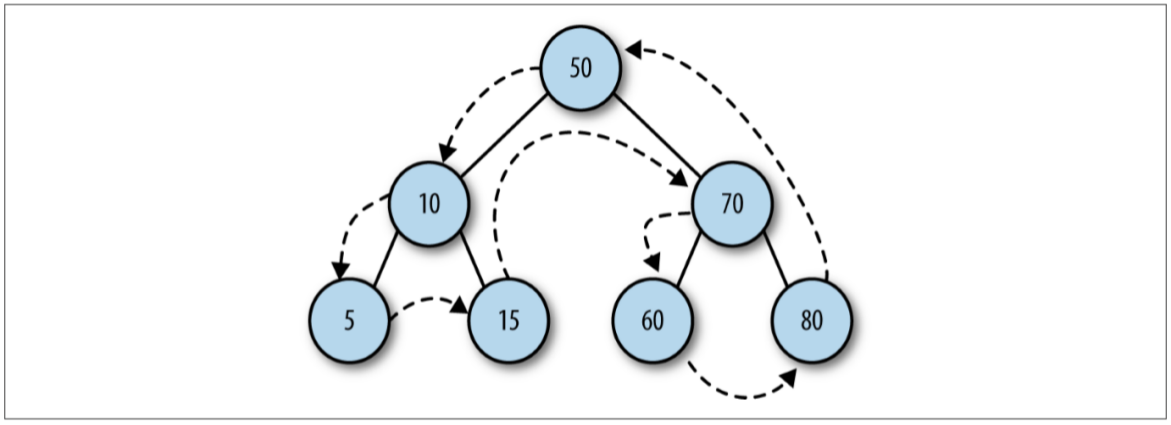
Path of inorder traversal

Example: Here is the code for the preorder traversal function:

```
preOrder(node) {
   if (!(node == null)) {
      console.log(node.show() + " ");
      preOrder(node.left);
      preOrder(node.right);
   }
}
```

You'll notice that the only difference between the *inOrder*() and *preOrder*() functions is how the three lines of code inside the if statement are ordered. The call to the show() function is sandwiched between the two recursive calls in the *inOrder*() function, and the call to show() is before the two recursive calls in the *preOrder*() function.

The next figure illustrates the path the *preOrder*() function followed.



Path of preorder traversal

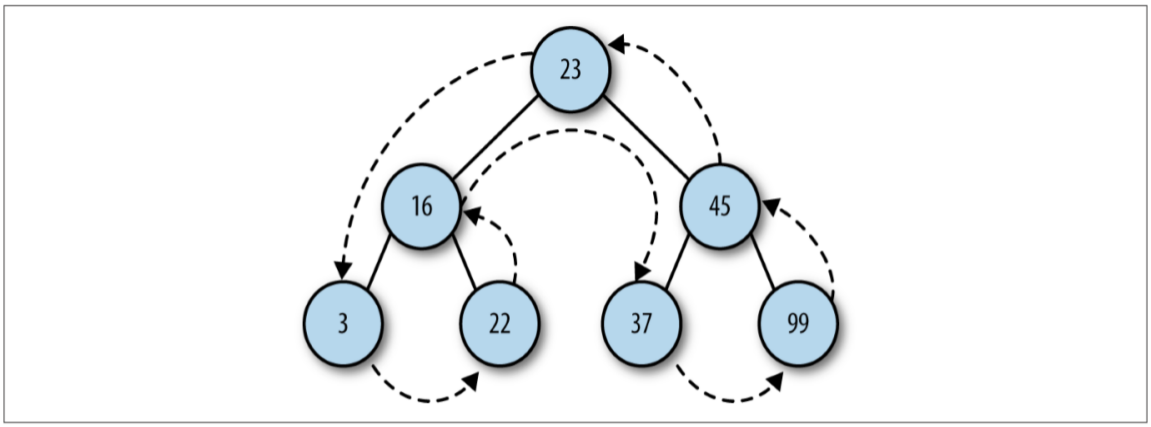
If we add a call to *preOrder*() to the preceding program, we get the following results:

Output:

```
Inorder traversal: 3 16 22 23 37 45 99

Preorder traversal: 23 16 3 22 45 37 99
```

The path of a **postorder** traversal is shown in the next figure:



Path of preorder traversal

Example: Here is the implementation of the ***postOrder***() function:

```
postOrder(node) {
   if (!(node == null)) {
      postOrder(node.left);
      postOrder(node.right);
      console.log(node.show() + " ");
   }
}
```

Output: And here is the output when we add the function 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
```

We will demonstrate some practical programming examples using BSTs that make use of these traversal functions later in the chapter.

9.5. 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.
3. Searching for the maximum value.

We explore these three searches in the following sections.

9.5.1. Searching for the Minimum and Maximum Value

Searches in a BST for the minimum and maximum values stored are relatively simple procedures. Since lower values are always stored in left child nodes, to find the minimum value in a BST, you only have to traverse the left edge of the BST until you get to the last node. Here is the definition of a function, getMin(), that finds the minimum value of a BST:

Example:

```
getMin() {
  var current = this.root;
  while (!(current.left == null)) {
     current = current.left;
  }
  return current.data;
}
```

The function travels down the left link of each node in the BST until it reaches the left end of the BST, which is defined as:

Example:

```
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 stored in a BST, the function must simply traverse the right links of nodes until the function reaches the right end of the BST. The value stored in this node must be the maximum value.

Example: The definition for the getMax() function is below:

```
getMax() {
  var current = this.root;
  while (!(current.right == null)) {

     current = current.right;
  }
  return current.data;
}
```

Let's tests the getMin() and getMax() functions with the BST data we used earlier.

Example: Testing getMin() and getMax() .

```
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();
console.log("The minimum value of the BST is: " + min);

var max = nums.getMax();
console.log("The maximum value of the BST is: " + max);
```

Output: The output from this program is:

```
The minimum value of the BST is: 3

The maximum value of the BST is: 99
```

These functions return the data stored in the minimum and maximum positions, respectively. Instead, we may want the functions to return the nodes where the minimum and maximum values are stored. To make that change, just have the functions return the current node rather than the value stored in the current node.

9.5.2. 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 being searched for. The comparison will determine if the search travels to the left child node, or to the right child node if the current node doesn't store the searched-for value.

We can implement searching in a BST with the find() function, which is defined here:

Example: BST *find*() method:

```
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 function returns the current node if the value is found in the BST and returns null if the value is not found.

Example: Using *find*() to search for a value

```
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);

var input = prompt("Enter a value to search for: ");
var value = parseInt(input);

var found = nums.find(value);
if (found != null) {
   console.log("Found " + value + " in the BST.");
 } else {
   console.log(value + " was not found in the BST.");
 }
```

The output from this program is:

Output:

```
3 16 22 23 37 45 99

Enter a value to search for: 23

Found 23 in the BST.
```

9.6. Removing Nodes from a BST

The most complex operation on a BST is removing a node. The complexity of node removal depends on which node you want to delete. If you want to remove a node with no children, the removal is fairly simple. If the node has just one child node, either left or right, the removal is a little more complex to accomplish. The removal of a node with two children is the most complex removal operation to perform.

To aid in managing the complexity of removal, **we remove nodes from a BST recursively**. The two functions we will define are **remove**() and **removeNode**().

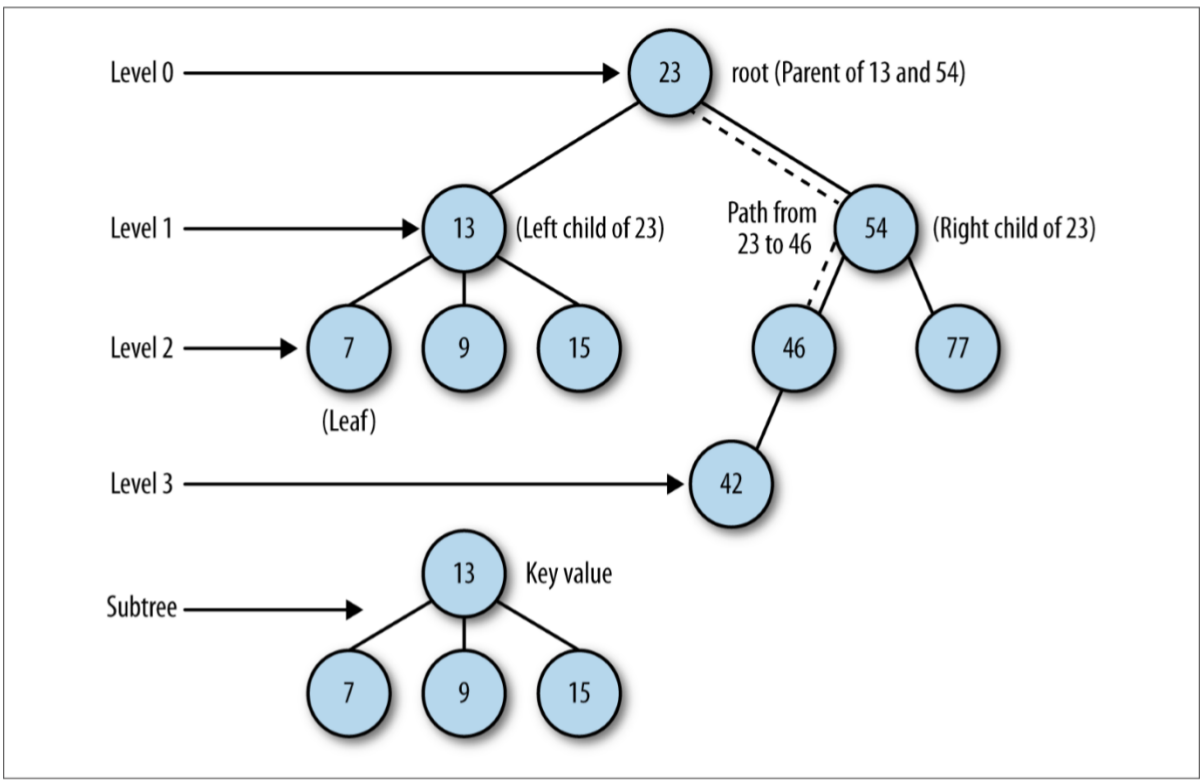
The first step to take when removing a node from a BST is to check to see if the current node holds the data we are trying to remove. If so, remove that node. If not, then we compare the data in the current node to the data we are trying to remove. If the data we want to remove is less than the data in the current node, move to the left child of the current node and compare data. If the data we want to remove is greater than the data in the current node, move to the right child of the current node and compare data.

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 link that is pointing to the node of the parent node to null.

When the node we want to remove has one child, then the link that is pointing to the node to be removed has to be adjusted to point to the removed node's child node.

Finally, when the node we want to remove 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 function that finds the smallest value of a subtree, which we will then use to create a temp re repla iis scen



Removing a node with two children

The node removal process consists of two functions. The *remove*() function simply receives the value to be removed and calls the second function, *removeNode*(), which does all the work. The definitions of the two functions are shown here:

Example:

```
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;
   }
 }
```

9.7. 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 grades, we can write a program that checks to see if the grade is in the BST, adding the grade to the BST if it is not found, and incrementing the number of occurrrences of it if the grade is found in the BST.

To solve this problem, we need to modify the Node object to include a field for keeping track of the number of occurrences of a grade in the BST, and we need a function for updating a node so that if we find a grade in the BST, we can increment the occurrences field.

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

Example:

```
class Node {

   constructor(data, left, right){
      this.data = data;
      this.left = left;
      this.right = right;
      this.count = 1;
   }

}
```

When a grade (a Node object) is inserted into a BST, its count is set to 1. The BST in sert() function will work fine as is, but we need to add a function to update the BST when the count field needs to be incremented. We'll call this function update():

Example:

```
update(data) {
   var grade = this.find(data);
   grade.count++;
   return grade;
}
```

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

Example:

```
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;
}
```

The next code presents a program for testing out this new code for counting occurrences of grades.

Example: Counting occurrences of grades in a data set.

```
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:

Output:

```
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
```