

Code to do Delete

Last time we outlined how to delete a node from a binary tree. We broke up our work into three cases:

- 1) Deleting a left node
- 2) Deleting a node with one child
- 3) Deleting a node with two children.

Keep these three in mind, as well as how we determined the delete would work in these cases when looking at this code.

First, here are the changes to the `BinTreeNode` class:

```
// Returns a reference to the parent of the node that x
// references. If x does not have a parent in the tree, then null is
// returned.
public BinTreeNode parent(BinTreeNode x) {

    // Case of one node.
    if ((left == null) && (right == null))
        return null;
    // Case that this is the parent.
    if ((left == x) || (right == x))
        return this;

    // Recursively determine parent.
    else if (x.data < this.data)
        return left.parent(x);
    else if (x.data > this.data)
        return right.parent(x);
    else
        return null;
}
```

```
// Returns the minimum value stored in the tree.  
public int minVal() {
```

```
    if (left == null)  
        return data;  
    else  
        return left.minVal();  
}
```

```
// Returns the maximum value stored in the tree.  
public int maxVal() {
```

```
    if (right == null)  
        return data;  
    else  
        return right.maxVal();  
}
```

```
// Returns true if the current node is a leaf node.
```

```
public boolean isLeafNode() {  
    return ((left == null) && (right == null));  
}
```

```
// Returns true if the current node has exactly one child  
// node.
```

```
public boolean hasOneChild() {  
    return (!isLeafNode() && ((left == null) || (right == null)));  
}
```

```

// Delete a node in the tree that stores the value x. If the
// deletion is successful, true is returned, otherwise false is
// returned and the tree is left unchanged.
public boolean delete(int x) {

    // Checks if the node is in the tree.
    BinTreeNode temp = searchNode(x);
    if (temp == null)
        return false;

    // Takes care of deleting a leaf node.
    if (temp.isLeafNode()) {
        BinTreeNode theparent = parent(temp);
        if ((theparent.left != null) && (theparent.left.data == x))
            theparent.left = null;
        else
            theparent.right = null;
    }

    // Takes care of deleting a node with one child node.
    else if (temp.hasOneChild()) {
        BinTreeNode theparent = parent(temp);
        if (theparent.left != null && theparent.left.data == x) {
            if (temp.left == null)
                theparent.left = temp.right;
            else
                theparent.left = temp.left;
        }
        else {
            if (temp.left == null)
                theparent.right = temp.right;
            else
                theparent.right = temp.left;
        }
    }
}

```

```

// Deletes node with two children
else {
    int newval = temp.right.minVal();
    delete(newval);
    temp.data = newval;
}
return true;
}

```

Now, let's look at the changes in the BinTree class:

```

// Deletes a node containing x from the binary tree. The
// method returns true if the deletion was done successfully.
public boolean delete(int x) {

    // Can't delete from an empty tree.
    if (root == null)
        return false;

    // Deleting root node.
    if (root.getData() == x) {

        // Take care of each case individually
        if (root.isLeafNode())
            root = null;
        else if (root.hasOneChild()) {
            if (root.getLeft() != null)
                root = root.getLeft();
            else
                root = root.getRight();
        }
    }
}

```

```
// Two Child BinTreeNode method works fine since the  
// actual reference root is not being moved.  
else  
    return root.delete(x);  
  
    return true;  
}  
// BinTreeNode method works in all other cases.  
else  
    return root.delete(x);  
}
```

Order Statistics w/a Binary Search Tree

Given a binary search tree, let's consider the problem of finding the k th smallest item stored in the tree. Basically, the item can be in one of three places:

- 1) Left side of the tree
- 2) root
- 3) Right side of the tree

If L is the size of the left side of the tree, then how can we determine which of the three places to look?

- 1) if k is less than or equal to L
- 2) if k equals $L+1$
- 3) if $k > L+1$

Using this, we can add some code to the `BinTreeNode` and `BinTree` classes to implement a method that solves this problem:

```
// Finds the kth smallest item in the binary tree and returns
// it. If no such item exists, -1 is returned.
public int kthSmallest(int k) {

    // Deals with the invalid case of k being too big.
    if (k > size() || k < 1)
        return -1;

    // calculate number of nodes on the left side of the tree.
    int leftside = 0;
    if (left != null)
        leftside = left.size();
```

```
// Recursively determine the kth smallest value.  
if (k <= leftside)  
    return left.kthSmallest(k);  
else if (k == (leftside+1))  
    return data;  
else  
    return right.kthSmallest(k-leftside-1);  
  
}
```

```
// Finds the kth smallest element in the binary tree and returns  
// it's value. If there is not a kth smallest item, -1 is returned.  
public int kthSmallest(int k) {  
  
    if (root == null)  
        return -1;  
    else  
        return root.kthSmallest(k);  
}
```

Binary Search Tree Analysis

Here are some of the methods to analyze:

```
size();  
height();  
searchNode(int x);  
minVal();  
insert();  
delete();
```

Although these are different methods, we can see that we can fit each into one of two categories:

- 1) A method that "visits" each node once.
- 2) A method that "visits" each node IN A PATH of the tree, once.

Clearly, all of the methods in the first group will run in $O(n)$ time, where n is the number of nodes in the tree. (This is assuming that the amount of processing for each node is constant.)

It follows that at worst case, all the methods in the second group will run proportional to the depth of the binary tree. Simply put, the depth is the length of the longest path from the root to a leaf node of the tree.

Thus, what we need to determine is the depth of a binary tree.

First, let's consider the worst case:

The elements are inserted in order, just the binary tree looks like a linked list. In this case the depth is $n-1$, for n nodes.

Now the best case:

The elements are inserted in such a way to form a perfectly balanced binary tree. Then the depth will be the floor($\log n$). (To see this, consider that there are 2^k nodes at a depth of k in the tree. Thus, in a perfectly balanced binary tree with in between 2^k nodes and $2^{k+1}-1$ nodes, the depth is k .)

Finally, the average case:

If you look at the structure of a binary search tree, and compare it to quick sort, what sort of similarity do you find?

Each node is essentially a partition element. The root node is the first partition element chosen, and then the nodes under it are the ones chosen as partition elements in the two main recursive calls, and this pattern continues.

In essence, the SUM of the path lengths to each node, defined as the internal path length, in a Binary Search Tree is IDENTICAL to the running time of Quick Sort.

Thus, the AVERAGE internal path length of a Binary Search Tree is $O(n \log n)$. This means that the average depth of a node in a random Binary Search Tree is $O(n \log n)/n = O(\log n)$.

With respect to both inserting and deleting, the average time is proportional to the average depth of a node.