CSCI 2270 - Data Structures and Algorithms

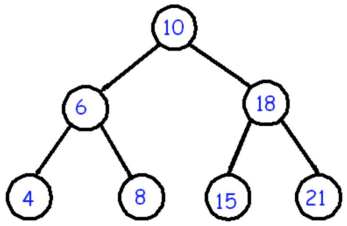
Instructor: Hoenigman / Jacobson
Recitation 8[1]

Binary Search Trees

A binary search tree (BST) is a special case of a binary tree where the data in the tree is ordered. For any node in the tree, the nodes in the left sub-tree of that node all have a value less than the node value, and the nodes in the right sub-tree of that node all have a value greater than or equal to the node value.

Let x and y be nodes in a binary search tree. If y is in the left sub-tree of x, then y.key < x.key. If y is in the right sub-tree of x, then y.key >= x.key.

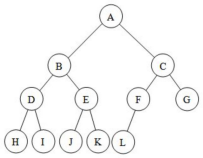
Visualization of a Perfect Binary Tree:[2]



Types of Binary Trees:

There are several types of binary search trees. A few common ones are listed.

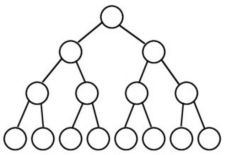
Complete Tree:[3]
Every level, except possibly the last, is completely filled, and all nodes are as far left as possible



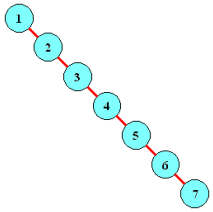
Full Tree:[4]
Every node has either 0 or 2 children

---

[1] Excerpts From: Rhonda Hoenigman. “Visualizing Data Structures.” iBooks.
[2] Visual from: www.cs.cmu.edu
[3] http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/FullvsComplete.html
[4] http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/FullvsComplete.html



Degenerate Tree:[5]
Every parent node has only one child



C++ implementation of a binary tree node

A binary tree node in C/C++ can be built with a struct, just like a node in a linked list, where the members of the struct include the key, a pointer to the parent node, pointers to the leftChild and rightChild nodes, and any additional data that the program needs to store to operate successfully.

```
struct node{
        int key
        node *parent
        node *leftChild
        node *rightChild
}
```

Searching a BST

The BST ordering generates a structure, whereby, from any given node, a search operation can identify a section of the tree that might contain the search value and eliminate the rest of the tree from consideration. For example, if the search value is less than the value of a given node, then all nodes to the right of that node don't need to be evaluated. This ordering prunes the search space by removing branches that won't contain the search value.

The BST search can be performed recursively or iteratively. We'll focus on the iterative version today. The search completes when the value is found, or the bottom of the tree is reached, which indicates that the value does not exist in the tree.

Algorithm search(value)
Returns a pointer to the node where the key matches the search value.

---

[5] condor.depaul.edu

Pre-conditions:
value is a valid search value that is the same type as the node key.

Post-conditions:
Returns a pointer to the node where the search value matches the key or NULL if the key does not exist in the tree.

```
searchIterative(value)
1.      node = root
2.      while(node != NULL)
3.              if (value < node.key)
4.                      node = node.left
5.              else if( value > node.key)
6.                      node = node.right
7.              else
8.                      return node
9.      return NULL
```

Inserting in a BST

Inserting a node into a BST involves searching for the correct placement of the node, and then, modifying the tree to add the node.

Algorithm insert(value)
Inserts a node with the specified value into a BST at the appropriate position.

Pre-conditions
value is a valid node value.

Post-conditions:
Memory for the node is allocated and the BST has been modified correctly to include the new node.

```
insert(value)
1.      tmp = root                                      //set pointer to root node
2.      node.key = value
3.      node.parent = NULL
4.      node.leftChild = NULL
5.      node.rightChild = NULL
6.      while(tmp != NULL)                      //find out where to put it
7.              parent = tmp
8.              if(node.key < tmp.key)                  //compare input value to root key
```

```
9.                              tmp = tmp.leftChild
10.                         else
11.                          tmp = tmp.rightChild
                                                        // actual placement of node
12.      if (parent == NULL)                            //check if tree is empty
13.              root = node
14.      else if(node.key < parent.key)                         // if left < parent, add to left
15.              parent.leftChild = node
16.              node.parent = parent
17.      else                                           //otherwise, add to right
18.              parent.rightChild = node
19.              node.parent = parent
```

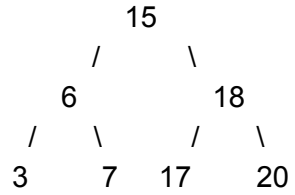
In the insert() algorithm, the node is created with value as the key value and the parent, leftChild, and rightChild pointers initialized to NULL. The while loop on Lines 6-11 identifies the correct placement for the new node by searching for a node with a NULL pointer its left or right child. At the end of the while loop, the value of tmp will be NULL because it will be pointing to the child. Lines 12-19 of the algorithm add the node to the tree as either the root if the tree is empty, or the left or right child. The parent value of the new node is also set.

Deleting a Node in Binary Trees

When a node is deleted from the tree, the node may need to be replaced with another node in the tree. The replacement node needs to be selected such that the BST properties are preserved.

There are three cases to consider when deleting a node. Exactly one of the following conditions is true about the deleted node:
1. The node has no children.
2. The node has one child.
3. The node has two children.

Figure 11 shows a BST with examples of nodes with 0, 1, or 2 children. The nodes with values of 3, 9, 17, and 20 have no children. The node with a value of 7 has one child, and the nodes with values of 6, 15, and 18 have two children. The delete() algorithm that handles all three cases is shown in Algorithm 9.5. For brevity, only the case where the deleted node is its parent's left child is shown.



Algorithm delete(value)

Deletes the node where the value matches the node key value.

Pre-conditions

value is a valid search value whose type matches the node key type.
search() algorithm exists to identify the node to delete.
treeMinimum() algorithm exists to identify the minimum value in a sub-tree, which will be the replacement node for a deleted node with two children.

Post-conditions

Node with specified key value is deleted from the tree.
parent, left child, and right child pointers for the deleted node and neighboring nodes are reset accordingly.

Algorithm

(Note: this is not the complete delete() algorithm. For the one- and two-children cases, only the case where the deleted node is the left child of its parent is shown. Additional cases are needed to handle when the deleted node is the right child.)

```
delete(value)
1.    node = search(value)
2.    if(node != root)
3.          if(node.leftChild == NULL and node.rightChild == NULL)          //no children
4.                node.parent.leftChild = NULL
5.          else if(node.leftChild != NULL and node.rightChild != NULL)     //two children
6.                min = treeMinimum(node.rightChild)
7.                if (min == node.rightChild)
8.                          node.parent.leftChild = min
9.                          min.parent = node.parent
10.               else
11.                         min.parent.leftChild = min.rightChild
12.                         min.parent = node.parent
13.                         min.right.parent = min.parent
14.                         node.parent.leftChild = min
15.                         min.leftChild = node.leftChild
16.                         min.rightChild = node.rightChild
17.                         node.rightChild.parent = min
18.                         node.leftChild.parent = min
19.         else          //one equals null and the other child doesn't                    //one
child
20.               x = node.leftChild
```

```
21.                 node.parent.leftChild = x
22.                 x.parent = node.parent
23.   else
24.                     //repeat cases of 0, 1, or 2 children
25.           //replacement node is the new root
26.            //parent of replacement is NULL
27.   delete node
```

Examples to cover:

Node has no children

delete(3)

Steps:

1. Set the left child pointer for the 6 to NULL.
2. Delete the 3 node.

Node has one child

delete(7)

Steps:

1. Set the right child pointer of the 6 to point to the 9.
2. Delete the 7 node.

Node has two children

delete(6)

Steps:

1. The parent property of the 3 is updated to point to the 7.
2. The leftChild property of the 15 is updated to point to the 7.
3. The parent property of the 7 is updated to point to the 15.
4. Delete the 6 node.

In class exercise

What is the pseudocode for a function that will return True if two trees are structurally identical (nodes with same values, but can be in different arrangement) or otherwise, returns False. The function will take two arguments: treenode of each tree. This means it will pass in both trees. You will have to traverse both of them and then fill up an array to see if they are equal by comparing the arrays.

Recitation 8 Programming Assignment

There is a link on Moodle to a Recitation 8 Programming exercise. In the quiz, you are asked to take the psuedocode of the in class exercise and write the function for it. You will have the directions on the code-runner exercise for specific clarification. You have until Sunday at 5pm to complete the exercise.