**Assignment 12 – Search Trees and Hashing**

*Write pseudo-code not Java for problems requiring code. You are responsible for the appropriate level of detail.*

**1. Show that every n-node binary search tree is not equally likely (assuming items are inserted in random order), and that balanced trees are more probable than straight-line trees. (**This question is asking you to look at the shape of the trees and show that some shapes are more probable than others**.)**

In order to have a straight line tree, every element has to be greater or less than the previous element. This is the only way to get a straight line without any elements in the other child. In a random list, we can make the assumption that on average, each element has a 50% chance to be greater, and 50% chance to be less. This means that a random list of 4 elements would have a 2\*.5^4, or 12.5%, chance of being in a straight line on either side. As the list of elements increases, the chances of it being a straight-line tree decreases more and more. However, having a perfectly balanced tree is also unlikely, as it requires a node that is in the middle of the list of numbers to be selected as a root. However, a nearly balanced is much more likely than either of those possibilities, as there are more combinations that can result in a nearly balanced tree than either a perfectly balanced or straight line tree.

**2. Write a method delete(key1, key2) to delete all records with keys between key1 and key2 (inclusive) from a binary search tree whose nodes look like this:**

left key*i*  right

Class Node(){

Int key;

Node leftNode;

Node rightNode;

}

Class BST{

Node root;

BSTInsert(tree, node) { // putting insert function here to show that root will be initialized

if (BST.root is null)

BST.root = node;

node.leftNode = null;

node.rightNode = null;

else

cur = BST.root;

while (cur is not null)

if (node.key < cur.key)

if (cur.leftNode is null)

cur.leftNode = node;

cur = null;

else

cur = cur.leftNode;

else

if (cur.rightNode is null)

cur.rightNode = node;

cur = null

else

cur = cur.rightNode

node.leftNode = null

node.rightNode = null

}

Void delete(node, key1,key2){

If(node == null){

Return null;

} //Checks to see if tree is empty. Also acts as a base case for recursion.

node.leftNode = delete(node.leftNode, key1, key2);

node.rightNode = delete(node.rightNode,key1,key2);

if (node.key >= min & node.key <= max &node.leftNode !=null and node.rightNode !=null){

//This means that this was the only node caught in the range, as the recursive function would already have removed any less than or greater than this node

Suc = node.rightNode;

While (suc.leftNode != null)

Suc = suc.leftNode;

Node = copy suc;

Delete(node.right, suc.key, suc.key)//this will recursively delete on only the suc.key

else if (node.key >= min & node.key <= max &node.leftNode.key ==null){

return root.rightNode;

**} //**first, removes all left nodes that are within the range. If the node does not have a left child, then return the right Node. Will return null if there is no right node.

Else if (node.key >= min & node.key <= max &node.leftNode.key !=null){

//Only time that this would happen is when the node is in range, but the left node was not

Return root.leftNode;

}

} else { //node is not in range

Return node;

}

}

}

Main(){

BST tree = new BST;

//do insertions here

Delete(tree.root, key1, key2);

}

**3. Write a method to delete a record from a B-tree of order n.**

p0  r1 p1 r2 p2 r3 ……. pn-1 rn-1 pn rn

When we write a method to delete a record from a B-tree of order n, we have to make sure that a node doesn’t get too small during deletion.

Note: I didn’t get very far with this one. I was trying to follow the code in ZyBooks, but I can’t make heads or tails out of how to do it with more than a 2-3-4 tree. I was thinking that Pn-1 is the left child, and Pn is the right child. So to search, I would iterate through Rn until I found the value, or to go to the left pointer once I found a larger value, or the rightmost pointer if no larger value is found. Then I would repeat the process until I found the node. From there, I would delete the node, and check the size of the resulting node. If the node was smaller, then I would do fusion by pulling down the parent and a node from a neighbor. But when I was trying to write code, I got a little confused about how to translate it, so I apologize for that.

BTreeRotateLeft(node) {

leftSibling = BTreeGetLeftSibling(node)

keyForLeftSibling = BTreeGetParentKeyLeftOfChild(node->parent, node)

BTreeAddKeyAndChild(leftSibling, keyForLeftSibling, node->left)

BTreeSetParentKeyLeftOfChild(node->parent, node, node->A)

BTreeRemoveKey(node, 0)

}

BTreeFuse(leftNode, rightNode) {

parent = leftNode->parent

if (parent is root and has 1 key)

return BTreeFuseRoot(parent)

middleKey = BTreeGetParentKeyLeftOfChild(parent, rightNode)

fusedNode = new Node(leftNode->A, middleKey, rightNode->A)

fusedNode->left = leftNode->left

fusedNode->middle1 = leftNode->middle1

fusedNode->middle2 = rightNode->left

fusedNode->right = rightNode->middle1

keyIndex = BTreeGetKeyIndex(parent, middleKey)

BTreeRemoveKey(parent, keyIndex)

BTreeSetChild(parent, fusedNode, keyIndex)

return fusedNode

}

BTreeMerge(node){

leftSib = BTreeGetLeftSibling(node);

rightSib = BTreeGetRightSibling(node);

if (leftSib != null && leftSib.numKeys >= 2){

BTreeRotateRight(leftSib)

}

else if (rightSib != null && rightSib.numKeys >= 2){

BTreeRotateLeft(rightSib)

} else {

if (leftSib == null)

BTreeFuse(node, rightSib)

else

BTreeFuse(leftSib, node)

}

BTreeRemove(tree,key){

If (tree.root has 1 key & tree.leftNode == null & tree.rightNode == null & tree.root.A == key){

Tree.root = null;

Return true;

}

Current = tree.root;

While (current != null){

If (current has 1 key and current != tree.root){

BTreeMerge(cur);

}

**4. If a hash table contains *tablesize* positions and *n* records currently occupy the table, the load factor *lf* is defined as *n/tablesize.* Suppose a hash function uniformly distributes *n* keys over the *tablesize* positions of the table and the table has load factor *lf*. Show that of new keys inserted into the table, (n-1)\**lf*/2 of them will collide with a previously entered key. Think about the accumulated collisions over a series of collisions.**

Lf = n/m. Thus, the equation can be rewritten as (n-1)\*n/2m

If we have n=1, then the equation will give us f(n) = 0.

If we have n=2, then the equation is 1\*2/2m, or 1/m.

If we have n=3, then the equation is 2\*3/2m, or 3/m.

If we have n=4, then the equation is 3\*4/2m, or 6/m.

This is a very simple series. For each n, there are n-1 places for there to be a collision. This means that for each, there is a (n-1)/m expected value of a collision. For the 1st n, that would be 0, for the second, that would be 1, so and and so forth. For the 4th n, there would be a 3/m chance of a collision for that particular node to be inserted. This, written out, is (0 + 1 + 2 + … + (n-1))/m. The top part of this equation is a series that is expressed by n(n+1)/2. Divide that by m from equation, and we get n(n+1)/2m, which is the equation shown.

**5. Assume that *n* random positions of a *tablesize*-element hash table are occupied, using hash and rehash functions that are equally likely to produce any index in the table. The hash and rehash functions themselves are not important. The only thing that is important is they will produce any index in the table with equal probability.**

**Start by counting the number of insertions for each item as you go along. Use that to show that the average number of comparisons needed to insert a new element is *(tablesize* + 1*)*/*(tablesize*-n+1).**

**Explain why linear probing does not satisfy this condition.**

Let tablesize =m. We can write the equation as (m+1)/(m-n+1), n being the number of filled elements.

If we let n=0, then the number of comparisons to insert a new object is (m+1)/(m+1), or 1.

N=1, (m+1)/(m).

N=2, (m+1)/(m-1).

N=3, (m+1)/(m-2).

If m = 10, then

N=1, 11/10 = 1.1 comparisons

N =2, 11/9 comparisons = 1.22 comparisons.

N=3, 11/8 = 1.375 comparisons

N=4, 11/7 = 1.57 comparisons

N=5, 11/6 = 1.833 comparisons

N=6, 11/5 = 2.2 comparisons

N=7, 11/4 = 2.75

N=8, 11/3 = 3.66

N=9, 11/2 = 5.5

N=10, 11/1 = 11

Assuming m = 10 again.

For n = 1, we have 1+1/10 comparisons, which is 1.1 comparisons.

For n =2, we have 1+2/10\*1/9\*2 + 2/10\*(8)/(9)\*1 chance of collision. This is 1+2/45 + 8/45, which is 1+2/9, which is 1.22 comparisons.

For n=3, we have 1+ 3/10\*2/(9)\*1/8\*3 + 3/10\*2/9\*7/8\*2 + 3/10\*(7)/(9)\*1 chance of collision. 1+3/120 + 14/120 + 7/30 = 1.375.

This means that for each n, there is a summation series that converges towards (m+1)/(m-n+1). This can be shown to be true for all values of m and n.

Linear probing does not satisfy this condition because the “rehash” function will not produce any index in the table with equal probability. This means that there will be a bias to the rehash function, and that the rehash function will not select independently from the hash function.