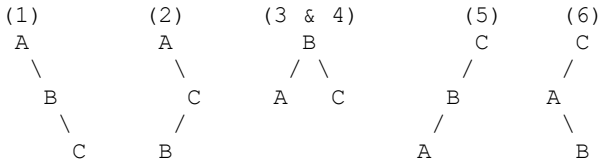
Adam Rich
EN.605.202.87.SP18 Data Structures
Homework Assignment 12
May 1, 2018

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**.)**

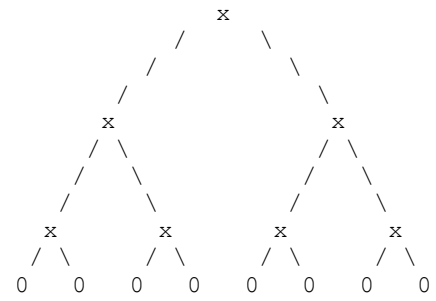
First assume that you have three numbers, a < b < c. When inserting in a random order from a NULL initial search tree, there are six possible outcomes.



In terms of efficiency, there are basically two shapes here:

1. inefficient = order 2, straight-line
2. more efficient = order < 2, with branching

Assume that n = 4, a < b < c < d. The total number of possible randomizations is 4! = 24. The maximum order of the search tree built off these is 3.



There are eight ways to get a "straight-line" tree with four nodes, and exactly eight selection orders of the 24 that will produce them, that is only 1/3. As n went from 3 to 4 the probability of getting a straight-line tree went from 2/3 to 1/3.

To generalize for any n (just assume for now there are all unique in 1..n), there are n! total selection orders (although not n! trees as some of those will be repeats). And there will be 2^(n-1) straight-line trees. As n increases, the probability of a straight-line tree reduces dramatically.

n	n!	2^(n-1)	Prob(straight-line) = 2^(n-1)/n!
3	6	4	.6667
4	24	8	.3333
5	120	16	.1333
6	720	32	.0444
7	5040	64	.0127

The probability that there is branching at the root is 2/n. This is because for there to be no right subtree, the root has to be 1 (prob = 1/n). For there to be no left

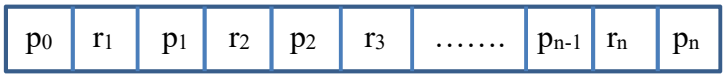
sub-tree, root has to be n (prob = 1/n). So, the higher n is, the more likely it is that there will be branching at the root. And, that recurses to each of the subtrees. So, the larger n is the more short and bushy the tree will be.

Beyond that, I think I will have to bust out my probability generation functions which I have not used in a long time!

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:

```
method rangeDelete(node, parent, key1, key2)
  // Base Cases
  if node = null return;
  if key1 > key2 return;

  if parent != null AND (node != parent.left and node != parent.right)
    throw error "bad call"

  k = node.key
  kleft = k - 1
  kright = k + 1
  rangeDelete(node.left, node, key1, min(kleft, key2))
  rangeDelete(node.right, node, max(kright, k1), k2)

  if (k >= key1 and k <= key2) {

    // no children
    if (node.right is null and node.left is null) {
      if (parent != null AND node == parent.left) parent.left = null
      if (parent != null AND node == parent.right) parent.right = null
      delete node
      return;
    }

    // one left child
    if (node.right is null and node.left is not null) {
      replacement = node.left
      if (parent != null AND node == parent.left) parent.left = null
      if (parent != null AND node == parent.right) parent.right = null
      delete node
      return;
    }

    // one right child
    if (node.right is null and node.left is not null) {
      replacement = node.right
      if (parent != null AND node == parent.left) parent.left = null
      if (parent != null AND node == parent.right) parent.right = null
      delete node
      return;
    }

    // inner node
    if (node.right is not null and node.left is not null) {

      // loop until finding in-order successor
      cur = nod
      next = node.right
      while (next.left != null) {
        cur = next
        next = next.left
      }

     cur.left = next.right
     replacement = next
      if (parent != null AND node == parent.left) parent.left = null
      if (parent != null AND node == parent.right) parent.right = null
      delete node
      return;
    }

  }
end-method
```

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




```
method delete(key)
  // Tree has order n, so
  int maxKids = n
  int maxKeys = n-1
  int minKids = ceiling(n/2)
  int minKeys = minKids - 1

  node <- find the node with key in it
  sib <- find nearest sibling to node
  mNode = key count in node
  parent <- parent of node
  mParent = key count in parent
  iNode = index of key in node, e.g. 1 to n
  iParent = index of key in parent that is between node and sibling

  // Case 0: node is in root, root has only one key
  consolidate children of root
  split consolidated node if bigger than maxKeys

  // Case 1: node is a leaf and m > minKeys
  delete the key in node and shift values, all pointers are null

  // Case 2: node is a leaf and m == minKeys
  //         and nearest sibling has number of keys > minKeys
  rotate value from sibling in to parent and parent to node

  // Case 3: node is a leaf and m == minKeys
  //         and nearest sibling has nothing to spare
  //         but mParent > minKeys
  consolidate node, sibling and parent.r[iParent]

  // Case 4: node is a leaf and m == minKeys
  //         nearest sibling has nothing to spare
  //         parent has nothing to spare
  consolidate node, sibling and parent.r[iParent]
  parent will not have enough, so will need to bubble up the consolidation

  // Case 5: internal node, mNode > minKeys, left and right child nodes
  //         can be consolidated without going over maxKeys
  delete node, consolidate children

  // Case 6: internal node
  //         a carefully picked descendant has m > minKeys
  successor = find left-most descendant leaf of right subtree
  predecessor = find right-most descendant leaf of left subtree

  if successor.node.m > minKeys
    replace key with successor, remove successor from its node

  if predecessor.node.m > minKeys
    replace key with predecessor, remove predecessor from its node


  // Case 7: internal node
  //         Case 6 fails
  delete
  consolidate with sibling
  check nodes going up to see if they need to be split
end-method
```

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.**

I think I got it now, I am going to insert n records in to an empty hash table. To make it easier, let S = *tablesize*.

For the first record, the probability of collision is 0. There are zero records in there so every slot is clear.

For the second, the probability of collision is 1/S. Assume that we keep going, maybe linearly?, until we find an empty spot (so each record gets entered somewhere) and that we are only counting records that collide, no matter how many times they collide.

So, for the m^{th} record, the probability of having a collision is (m-1)/S.

To get an expected value of collisions, add up all these probabilities.

```
E = [0 + 1 + ... + (n-1)] / S
  = [(n-1)*n/2]/S
  = (n-1)/2 * n/S
  = (n-1)/2 * LF
```

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.

Start with a table of size *tablesize* = S with n records already inserted. Assume that if we get a collision with a record, we can keep rehashing forever until we find an empty slot. The sequence of hashing will be called H1, H2, etc.

For the m^{th} insert, the probability of collision on any attempt will be (m-1)/S. The probability of no collision on an attempt is (S-m+1)/S.

For the m^{th} insert attempts, call p = (m-1)/S and q = 1 - p.

Since we are going to attempt the m^{th} insert until we get it, we can get the expected number of comparisons (collisions+1) to be represented by this infinite sum (K is the key of the $(n+1)^{th}$ record to insert):

```
  1*prob(H1(K) is not occupied)
+ 2*prob(H1(K) is occupied)*prob(H2(K) is not)
+ 3*prob(H1(K) and H2(K) are occupied)*prob(H3(K) is not)
+ ...

= q + 2p*q + 3p^2*q + 4p^3*q + ...
= q*[1 + 2p + 3p^2 + ...]
= q*1/(1-p)^2   (Taylor Series Differentiation)
= (1-p)/(1-p)^2
= 1/(1-p)
= 1/q
= S/(S-m+1)
```

Using Taylor series differentiation,
$1 + 2r + 3r^2 + ... = 1/(1-r)^2$

The average from 1 to n is going to be

$= 1/n * [S/S + S/(S-1) + S/(S-2) + ... + S/(S-n+1)]$
$= S/n * [1/S + 1/(S-1) + 1/(S-2) + ... + 1/(S-n+1)]$

This is a partial harmonic sum which has no closed-form solution, so I don't think I have interpreted the formula correctly.

Linear probing will not get the results shown in the prompt because not all slots are equally likely, namely because you move in one direction to find an empty slot, and there will likely be empty slots behind the first one tried.