CS106B
Spring 2013

Handout #20S
May 20, 2013

Section Solutions 7

Based on handouts by Eric Roberts

Problem One: Scrambled Hashes

- Hash function 1: Always return 0.

This is a valid hash function, but because every string will get hashed into the same bucket the cost of looking up values in the hash table will be O(n).

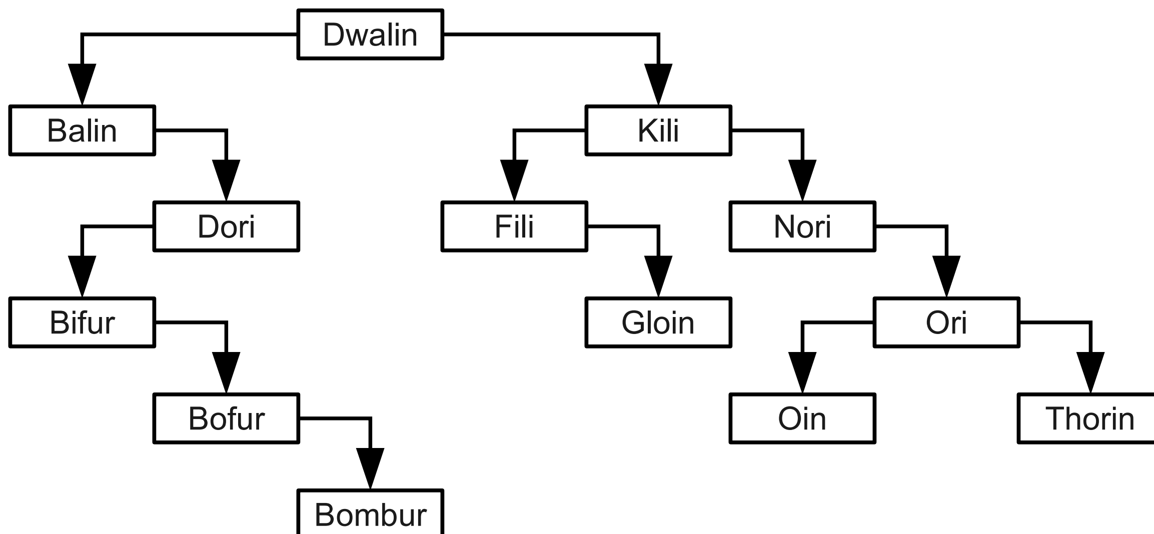
- Hash function 2: Return a random `int` value.

This is not a valid hash function because hashing the same string multiple times will produce different hash codes. Consequently, you won't be able to look up any strings in the hash table!

- Hash function 3: Return the sum of the ASCII values of the letters in the word.

This hash function is better than the first, but will not distribute the words evenly. Note that any two words with the same letter frequencies will have the same hash code (for example, "table" and "bleat"). Some words with the same length might also collide if they have two letters that have shifted from one another by the same amount; for example, "mass" and "last."

Problem Two: Tracing Binary Tree Insertion (Chapter 16, review question 9, page 711)



2a. What is the height of the resulting tree? **6**

2b. Which nodes are leaves? **`Bombur`**, **`Gloin`**, **`Oin`**, and **`Thorin`**

2c. Which key comparisons are required to find the string `"Gloin"` in the tree?

`"Gloin"` > `"Dwalin"`, `"Gloin"` < `"Kili"` , `"Gloin"` > `"Fili"`, **and** `"Gloin"` == `"Gloin"`

Problem Three: Checking BST Validity

There are many ways to solve this problem. One option is to do an inorder walk of the tree, keeping track of the last node visited. The value of the current node should always be at least the value that came before it:

```
bool isBST(Node* root) {
    Node* lastNode = NULL;
    return recIsBST(root, lastNode);
}

/* Checks if the current node represents a valid BST, given that the last
 * node visited was prev.
 */
bool recIsBST(Node* root, Node*& prev) {
    /* Base case: If the tree is empty, it's a valid BST. */
    if (root == NULL) return true;

    /* Recursive step: Check if the left subtree is valid.  If so, then check
     * if our value comes next in sequence, then check the right subtree.
     */
    if (!recIsBST(root->left, prev)) return false;

    /* Check our value against the previous one, if one exists. */
    if (prev != NULL && root->value <= prev->value) return false;

    /* Mark that we visited this node, then search the right subtree. */
    prev = root;
    return recIsBST(root->right, prev);
}
```

Problem Four: Order Statistic Trees

This function has a beautiful recursive structure to it. Essentially, to look up a value, we compare the current index to how many nodes are in the left subtree. If it's equal, then we have the value we want. If it's less, we descend into the left. Otherwise, we descend into the right.

```
Node* nthNode(Node* root, int n) {
    /* Base case: If the root is NULL, nothing exists. */
    if (root == NULL) return NULL;

    /* Recursive step: If this is the value we're looking for, we're done. */
    if (n == root->leftSubtreeSize) return root;

    /* Otherwise, if this node is to the left, look there. */
    if (n < root->leftSubtreeSize) return nthNode(root->left, n);

    /* Otherwise, look to the right.  Take into account the number of nodes
     * that we skipped when doing so.
     */
    return nthNode(root->right, n – 1 – root->leftSubtreeSize);
}
```