Practice Second Midterm Exam Solutions

Based on handouts by Eric Roberts and Jerry Cain

Problem One: Reversing a Queue

One way to reverse the queue is to keep moving nodes out of the list one at a time to the front of the list. We walk across the list one node at a time, at each point rewiring the list so that this new cell now points to the front of the list and updating the head of of the list accordingly:

```
void Queue::reverse() {
    /* The head element becomes the tail. */
    tail = head;

    /* Continuously pull the element just after the head in front of the
     * head.
     */
    Cell* curr = head;
    head = NULL;
    while (curr != NULL) {
        Cell* next = curr->link;
        curr->link = head;
        head = curr;
        curr = next;
    }
}
```

Problem Two: Rebuilding Binary Search Trees

Here is one possible implementation of **`fillVector`**, which does a standard inorder walk of the tree to build up the collection of nodes in sorted order. Since each node is visited exactly once, this takes time O(n). Note that the ordering of the recursive calls and the insertion of the current node is critical to this code working correctly!

```
void fillVector(BSTNode* node, Vector<BSTNode*>& v) {
    if (node == NULL) return;

    fillVector(node->left, v);
    v += node;
    fillVector(node->right, v);
}
```

The **`rebuildTree`** function works recursively – we build up the solution tree by recursively building up the left and right subtrees. A key detail is how we handle the case where we try to build up a tree out of an empty range. This needs to return **`NULL`**, and is responsible for ensuring that the left and right subtrees of all leaf nodes are set to **`NULL`**.

This function runs in time O(n) because we make exactly $2n + 1$ calls – one for each of the n nodes, plus an extra $n + 1$ calls for all of the **`NULL`** nodes that we need to fill in.

```
BSTNode* rebuildTree(Vector<BSTNode*>& v, int start, int end) {
    /* Base case: Building a tree from no nodes yields the empty tree. */
    if (start > end) return NULL;

    /* Find the middle node. */
    int mid = (start + end) / 2;
    BSTNode* newRoot = v[mid];

    /* Now, rebuild the left and right subtrees. */
    newRoot->left = rebuildTree(v, start, mid - 1);
    newRoot->right = rebuildTree(v, mid + 1, end);

    return newRoot;
}
```

Interestingly, this rebalancing strategy is a key component in *scapegoat trees*, a type of balanced binary search tree that ensures balance by aggressively rebuilding subtrees when the tree becomes unbalanced.

Problem Three: Spaghetti Stacks (40 Points)

(i) Making Spaghetti (20 Points)

There are two main approaches that can be used to solve this problem. The first approach is to recursively convert each tree to a spaghetti stack by constructing each tree with knowledge of its parent. This is shown here:

```
Set<SpaghettiNode*> spaghettify(TreeNode* root) {
    Set<SpaghettiNode*> result;
    spaghettifyRec(root, NULL, result);
    return result;
}

/* Builds a spaghetti stack from root whose parent in the spaghetti stack is the
 * node parent.
 */
void spaghettifyRec(TreeNode* root, SpaghettiNode* parent,
                    Set<SpaghettiNode>& result) {
    /* If there is nothing to build, we're done. */
    if (root == NULL) return;

    /* Construct a new spaghetti node wired into the parent. */
    SpaghettiNode* sNode = new SpaghettiNode;
    sNode->value = root->value;
    sNode->parent = parent;

    /* If this is a leaf node, add it to the result set. */
    if (root->children.isEmpty()) {
        result += sNode;
    }
   /* Otherwise, build up all the children of this node as spaghetti stacks
    * that use the current node as a parent.
    */
   else {
       foreach (TreeNode* child in root->children) {
           spaghettifyRec(child, sNode, result);
       }
   }
}
```

The other major approach is to spaghettify each tree recursively and have the recursive function hand back a pointer to the root node of the new tree. Given this pointer, we can then change its parent to the new node that we constructed.

```
/* Builds a spaghetti stack from root whose parent in the spaghetti stack is the
 * node parent.
 */
SpaghettiNode* spaghettifyRec(TreeNode* root, Set<SpaghettiNode>& result) {
    /* If there is nothing to build, we're done. */
    if (root == NULL) return NULL;

    /* Construct a new spaghetti node with no parent. */
    SpaghettiNode* sNode = new SpaghettiNode;
    sNode->value = root->value;
    sNode->parent = NULL;

    /* If this is a leaf node, add it to the result set. */
    if (root->children.isEmpty()) {
        result += sNode;
    }
    /* Otherwise, build up all the children of this node as spaghetti stacks
     * that use the current node as a parent.
     */
    else {
        foreach (TreeNode* child in root->children) {
            spaghettifyRec(child, result)->parent = sNode;
        }
    }

    /* Return this node as the root of the new spaghetti stack. */
    return sNode;
}
```

(ii) Cleaning up Spaghetti

The tricky part of this problem was finding a way to clean up the tree without deleting the same node twice. Simply walking up from each leaf node to the root cleaning as you go will cause problems if any node has two children.

There were two main approaches we saw for solving this problem. The first option was to just build up a set of all the nodes in the spaghetti stack, then to free each of them.

```
void freeSpaghettiStack(Set<SpaghettiNode*> leaves) {
    Set<SpaghettiNode*> nodes;
    foreach (SpaghettiNode* leaf in leaves) {
        for (SpaghettiNode* curr = leaf; curr != NULL; curr = curr->parent) {
            nodes += curr;
        }
    }
    foreach (SpaghettiNode* node in nodes) {
        delete node;
    }
}
```

Another option is to walk the tree deleting nodes and keeping track of what was deleted so that we don't delete anything twice.

```
void freeSpaghettiStack(Set<SpaghettiNode*> leaves) {
    Set<SpaghettiNode*> deletedNodes;
    foreach (SpaghettiNode* leaf in leaves) {
       while (leaf != NULL) {
           if (deletedNodes.contains(leaf)) break;
           deletedNodes += leaf;

           SpaghettiNode* next = leaf->parent;
           delete leaf;
           leaf = next;
       }
    }
}
```

Problem Four: Perfect Hash Tables **(40 Points)**

Here is one possible implementation:

```
PerfectHashTable::PerfectHashTable(Set<string>& values) {
    /* Set up the buckets array and number of elements. */
    numBuckets = values.size() * values.size();
    elems = new Bucket[numBuckets];

    /* Keep picking hash functions until we find one that works. */
    while (true) {
        whichFunction = randomInteger(INT_MIN, INT_MAX);

        /* Mark all buckets as empty. */
        for (int i = 0; i < numBuckets; i++) {
            elems[i].isUsed = false;
        }

        /* Hash each element into a bucket and see if we can do so
         * without collisions.
         */
        bool success = true;
        foreach (string value in values) {
           int bucket = hashCode(value, whichFunction) % numBuckets;
           if (elems[bucket].isUsed) {
               success = false;
               break;
           }
           elems[bucket].value = value;
           elems[bucket].isUsed = true;
        }

        if (success) break;
    }
}

PerfectHashTable::~PerfectHashTable() {
     delete[] elems;
}

bool PerfectHashTable::containsKey(string value) {
    int bucket = hashCode(value, whichFunction) % numBuckets;
    return elems[bucket].isUsed && elems[bucket].value == value;
}
```

Problem Five: Reasonably Short Answer (20 Points)

(i) Hash Table Performance (10 Points)

Suppose that you have a hash table with *n* elements and *b* buckets. Assuming that the hash table uses a good hash function, the *average* amount of work required to do a lookup is O(1 + *n* / *b*).

What are the *best-case* and *worst-case* runtimes of doing a lookup in a hash table of *n* elements and *b* buckets?

The best-case runtime for a lookup in a hash table is O(1). You might try to look up an element that hashes to an empty bucket, in which case only O(1) work is required to compute the hash code and to scan the empty list.

The worst-case runtime for a lookup in a hash table is O(n). It is possible that all elements in the hash table end up in the same bucket, in which case a lookup might have to look at every single element in the table when scanning over that bucket.

(ii) Bad Binary Search Trees (10 Points)

As mentioned in lecture, the order in which the elements of a binary search tree are inserted can influence the shape of that binary search tree and, accordingly, the amount of time required to do a lookup. In the worst-case, a binary tree with *n* nodes will have height *n* – 1.

How many ways are there to insert five elements into a binary search tree that cause the resulting tree to have height 4? Justify your answer.

A worst-case binary search tree with n elements will be one in which each node only has one child. This means that the root must have only one child, so it must be either the biggest or the smallest element in the tree (or otherwise there would be elements on both sides). Its child must have only one child as well, meaning that it must be either the biggest or the smallest of the remaining elements after the root is inserted. More generally, each node must be either the biggest or smallest of the remaining elements when it is inserted into the tree. This means that the number of ways to get a worst-case binary search tree for n elements is given by the number of ways to pick either the biggest or smallest of the remaining elements at each step.

For n = 5 elements, there are 16 ways to do this, because for the first four insertions you can pick either the biggest or smallest element, and for the very last element there is only one choice left of what to insert. Since there are four independent binary choices, there are $2^4 = 16$ possible ways to get a worst-case binary search tree.