Practice Second Midterm Exam Solutions

Based on handouts by Eric Roberts and Jerry Cain

Problem One: 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 Two: 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 Three: Wildcard Searches

Here is one possible implementation. The key insights are

- Using a helper function to track the prefix down through the recursion, and
- Recognizing how to implement ? and * correctly.

This implementation implements ? recursively and * in terms of ?.

```
void wildcardSearch(Node* root, string pattern, Vector<string>& result) {
    recWildcardSearch(root, pattern, result, "");
}

/* This recursive helper function tracks the prefix we've seen so far. */
void recWildcardSearch(Node* root, string pattern,
                       Vector<string>& result, string prefix) {
    /* Base case: If we walked off the trie, we're done. */
    if (root == NULL) return;

    /* Base case: If we are searching for the empty string, then we add the
     * current word to the result list if the current node happens to hold a word.
     */
    if (pattern == "") {
         if (root->isWord) result += prefix;
         return;
    }

    /* Recursive step: See what letter to look for.  If we're looking for a real
     * letter, then search that subtree.
     */
    if (isalpha(pattern[0])) {
        recWildcardSearch(root->children[pattern[0] - 'a'], // Correct child
                          pattern.substr(1),                // Tail of pattern
                          result,
                          prefix + pattern[0]);             // Factoring in letter
    }
    /* Otherwise, if this is a ?, then try searching all possible children.  One
     * way to do this (and the approach we adopt here) is just to swap out that ?
     * for all possible letters and retry the search here.
     */
    else if (pattern[0] == '?') {
        for (char ch = 'a'; ch <= 'z'; ch++) {
            pattern[0] = ch;
            recWildcardSearch(root, pattern, result, prefix);
        }
    }
    /* Finally, if this is a *, then try two options - either expand it out to
     * zero characters by removing it, or expand it out to any character by
     * replacing * with ?*, which means "some character, then more characters."
     */
    else if (pattern[0] == '*') {
        /* Option one: Expand to nothing. */
        recWildcardSearch(root, pattern.substr(1), result, prefix);

        /* Option two: Expand to ?*. */
        recWildcardSearch(root, '?' + pattern, result, prefix);
    }
}
```

Problem Four: Open Addressing

```
const int kInitNumBuckets = 64;

Map::Map() {
   numBuckets = kInitNumBuckets;
   count = 0;

   /* Set up the buckets by setting them all to empty. */
   buckets = new bucket[numBuckets];
   for (int i = 0; i < numBuckets; i++) {
      buckets[i].occupied = false;
   }

}

Map::~Map() {
   delete[] buckets;
}

const int kHashMultiplier = 716911;
int Map::hash(string key, int numBuckets) {
      int hashcode = 0;
      for (int i = 0; i < key.size(); i++) {
          hashcode = hashcode * kHashMultiplier + key[i];
      }
      return hashcode % numBuckets;
}

bool Map::enter(string key, double value) {
      if (count > 3 * numBuckets / 4) {
          rehash(); // you'll implement this helper method in part c
      }

      int basehash = hash(key, numBuckets);
      int offset = 0;
      for (int n = 0; n < numBuckets; n++) {
            offset += n;
            int bucket = (basehash + offset) % numBuckets;
            if (!buckets[bucket].occupied) {
                  buckets[bucket].occupied = true;
                  buckets[bucket].key = key;
                  buckets[bucket].value = value;
                  count++;
                  return true;
            } else if (buckets[bucket].key == key) {
                  buckets[bucket].value = value;
                  return false;
            }
      }
      error("Not supposed to get here.");
      return false; // can't get here, but compiler can't always tell
}
```

```
/**
 * Doubles the number of buckets held by the Map addressed by this,
 * and redistributes all of its key-value pairs.  Your implementation
 * should not orphan any memory whatsoever.
 */
void Map::rehash() {
      count = 0;
      numBuckets *= 2;
      bucket* oldBuckets = buckets;
      buckets = new buckets[numBuckets];
      // we've reset the Map to empty with space for twice as many elements
      for (int bucket = 0; bucket < numBuckets/2; bucket++) {
            if (oldBuckets[bucket].occupied) {
                  enter(oldBuckets[bucket].key, oldBuckets[bucket].value);
            }
      }
      delete[] oldBuckets;
}
```