Practice Second Midterm Exam

Based on handouts by Eric Roberts and Jerry Cain

Midterm Locations:

- **Last Name A – F: Go to Hewlett 201**
- **Last Name G – Z: Go to Hewlett 200**

This handout is intended to give you practice solving problems that are comparable in format and difficulty to the problems that will appear on the second midterm examination on Thursday, May 31. A solution set to this practice exam will be released online on Monday and handed out in class on Wednesday (since there is no class on Monday).

Time and place of the exam

The midterm exam is scheduled for a two-hour block at two different locations (note that the exams are not in the regular lecture room), divided by last name. Although the syllabus lists the time as 7:00PM to 10:00PM, the exam is only two hours long.

Coverage

The midterm covers the material presented in class through and including the lecture on Friday, May 24th on graphs. Although all of the material that we have covered so far may appear on the test, the exam will primarily focus on algorithmic efficiency and data structures.

General instructions

Answer each of the questions given below. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit.

Each question is marked with the number of points assigned to that problem. The total number of points on the exam is 120. We intend for the number of points to be roughly comparable to the number of minutes you should spend on that problem.

In all questions, you may include functions or definitions that have been developed in the course. First of all, we will assume that you have included any of the header files that we have covered in the text. Thus, if you want to use a `Vector`, you can simply do so without bothering to spend the time copying out the appropriate `#include` line. If you want to use a function that appears in the book that is not exported by an interface, you should give us the page number on which that function appears. If you want to include code from one of your own assignments, we won't have a copy, and you'll need to copy the code to your exam.

Unless otherwise indicated as part of the instructions for a specific problem, comments are not required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit on a problem if they help us determine what you were trying to do.

The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

Problem One: Rebuilding Binary Search Trees

When implementing a binary search tree, an important efficiency concern is taking care that the tree remains balanced to ensure logarithmic performance for insert and lookup. As it turns out, strategies that continually make minor rearrangements to rebalance the tree often end up spending too much time on those operations. An alternative strategy is to let the client determine whether a problem exists and, if so, rebalance the tree all at once.

One reasonably efficient strategy for rebalancing a tree is to transfer the nodes into a sorted vector and then reconstruct an optimally balanced tree from the vector. If you adopt this approach, you can implement a rebalancing operation for the `BSTNode` type by decomposing the problem into two helper methods as follows:

```
void rebalance(BSTNode*& root) {
   Vector<BSTNode *> v;
   fillVector(root, v);
   root = rebuildTree(v, 0, v.size() - 1);
}
```

The helper method

```
void fillVector(BSTNode *node, Vector<BSTNode *> & v);
```

adds all the nodes in the subtree rooted at `node` to the vector `v`, making sure that the nodes are added in the order they appear in the binary search tree. The helper method

```
BSTNode *rebuildTree(Vector<BSTNode *> & v, int start, int end);
```

recreates a tree by adding all the nodes from the vector `v` between the indices `start` and `end`, inclusive. Here, the goal is to make sure that the subtree returned by this operation is as balanced as possible, which means that the root must be as close as possible to the center of the range.

Write code for `fillVector` and `rebuildTree` to complete the implementation of the `rebalance` method.

In writing this problem, you should keep the following ideas in mind:

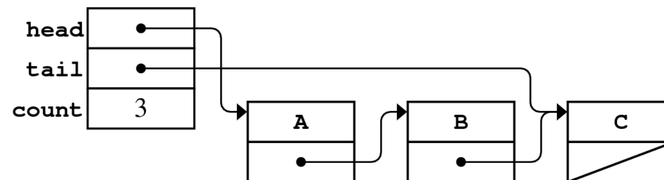
- Your implementation should not create or destroy any existing nodes, but simply rearrange them into a more balanced structure.
- For full credit, your implementation must run in $O(N)$ time, where N is the number of nodes. As always, this calculation does not take into account infrequent operations (such as the various versions of `expandCapacity`) whose amortized cost is constant. Thus, you may assume that adding an element to the end of a vector is a constant-time operation; adding an element at the beginning of a vector is not.

Problem Two: Reversing a Queue

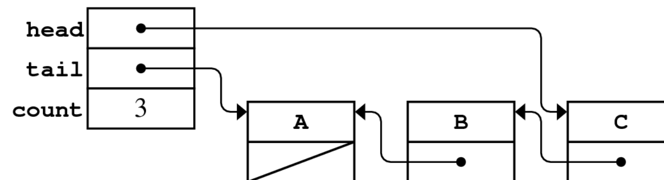
Suppose that you are implementing the **Queue** type using a linked list (as done in lecture) and are interested in adding the following member function:

```
void reverse();
```

This member function should reverse the contents of the queue. For example, if the queue initially contained the elements 1, 2, 3, and 4 in sequence, the reversed queue should then contain 4, 3, 2, and then 1. For example, if the variable **myQueue** is initialized to the structure



calling **myqueue.reverse()** should change that structure to



Your implementation of **reverse** must not allocate any new heap storage but must instead simply change the pointers in the existing cells, as illustrated in the preceding diagram. Since all of the collections types we have seen so far (**Vector**, **Stack**, **Queue**, etc.) allocate heap storage, this precludes the usage of any of those types. You should assume that the **private** section of the queue is defined as follows:

```
private:
      struct Cell {
            string value;
            Cell* next;
      };
      Cell* head;
      Cell* tail;
      int count;
```

Problem Three: Wildcard Searches

One interesting application of tries is *wildcard searching*, in which you are interested in searching the trie not for a specific word, but for all words matching some particular pattern. For example, consider the set of words

cat, chart, cot, car, hat, hot, war, warn, writ

Suppose want want to find all three-letter words that start with 'c' and end with 't'. In that case, we could do a wildcard search for the string “c?t”, where the question mark stands for “any character.” If we were to do this search over this set of words, we would get “cat” and “cot.” However, “c?t” does not match “chart,” since the question mark can match just a single character. Similarly, the search “?a?” would return “cat,” “car,” “hat,” and “war,” but not “warn.” We could also find all three-letter words ending with 'ot' by searching for “?ot,” which matches “cot” and “hot.”

Additionally, we might be in interested in finding every word in the set that ends with a 't', regardless of how many letters are in the word. In that case, we could do a search for “*t,” which would match “cat,” “chart,” “cot,” “hat,” “hot,” and “writ.” The search “*” will match every single word, while the search “cat*” will only match “cat.”

Using a trie, this sort of search can be implemented very efficiently. Suppose that you have the following **`struct`** representing a node in a trie:

```
struct Node {
    bool isWord;
    Node* children[26];
};
```

Write a function

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

that accepts as input a pointer to the root of the trie and string containing a pattern to search for, then fills in the specified **`Vector`** with every word in the trie that matches the given pattern. You can assume that every letter in the pattern string is either a lower-case letter, a star, or a question mark. As a hint, you might want to try implementing * in terms of ?.

Problem Four: Open Addressing

Many hash table implementations—including the hash table that backs our **Map** class—use external probing, so each bucket is actually a data structure storing all of the keys that hash to the same number.

For this version of the **Map**, you're going to use *quadratic internal probing*, which means that each bucket stores at most one key-value pair. If the **Map** wants to store a second pair in a particular bucket, we just won't let it. We'll force it to look elsewhere.

When entering a new key, the key is hashed and reduced to a number h. If that particular bucket is empty, then the key-value pair is assigned and that's that. If the bucket is occupied, but the new key matches the key already residing there, the old value is replaced with the new one. If the bucket is occupied, but occupied by a key different from the one being inserted, the search for an unoccupied bucket advances to examine slot h + 1, and if that fails, h + 3, and if that fails, h + 6, then h + 10, then h + 15, and so forth. (Of course, all of these numbers—h, h + 1, h + 3, etc—are modulo the number of buckets.)

You might ask why we're going with the triangular numbers—0, 1, 3, 6, 10, 15, 21, etc— for offsets. (A number is triangular if can be expressed as the sum of the form 1 + 2 + 3 + … + k. For instance, 10 is triangular because it can be written as 1 + 2 + 3 + 4.) We could try h, then h + 1, then h + 2, then h + 3, etc. But quadratic probing, using the triangular number offsets, does a better job of distributing the elements across the full range of buckets. And as long as the number of buckets is a power of 2, this quadratic probing method will explore every bucket if necessary.

Of course, there's the danger that we'll run out of buckets! After all, there are a limited number of them, and each one can accommodate at most one key-value pair. We're going to adopt the strategy that, at the beginning of each insertion request, we'll check to see if strictly more than 75% of the buckets are full, and if so, we'll double the number of buckets and rehash all previously inserted elements to their new home as if they're being inserted for the very first time. You're going to implement a new version of the **Map** using the ideas outlined above. We'll map strings to doubles to make things a little simpler. Here is the class definition you'll be working with (in practice, there would be **containsKey**, **getValue**, **iterator**, and **mapAll** methods as well, but you're not going to implement or even use them, so they're being omitted):

```
class Map {
public:
      Map();
      ~Map();
      bool enter(string key, double value);

private:
      struct bucket {
            bool occupied;
            string key;
            double value;
      };
      bucket *buckets; // addresses the array where all data gets stored
      int numBuckets;  // allocated length of the array
      int count;       // number of meaningful key-value pairs in the map

      int hash(string key, int numBuckets);
      void rehash();
};
```

The **buckets** field points to an array of **numBuckets** bucket records. The **occupied** field keeps track of whether or not that particular bucket is occupied by real data. If true, then some meaningful key-value pair occupies the rest of the record; if false, the **key** and **value** fields are irrelevant and ignored. (Note that all key-value pairs reside directly within the array addressed by buckets.)

i. Implement the **Map** constructor, which constructs a raw **Map** to be logically empty but with enough space for 64 key-value pairs. Then implement the destructor, which disposes of all resources maintained by the **Map** being destroyed.

```
/**
 * Constructor: Map
 * ----------------
 * Initializes the raw space addressed by this so that it represents an
 * empty Map otherwise capable of storing up to 64 key-value pairs.
 */
const int kInitNumBuckets = 64;
Map::Map();

/**
 * Destructor: ~Map
 * ----------------
 */
Map::~Map();
```

ii. Using the supplied hash method, implement the **enter** method, which ensures that the specified key is part of the **Map** and that it's associated with the specified value. It uses the quadratic internal probing technique, as described above, to find a home for the new key-value pair. **enter** returns true if the new key-value pairs gets inserted into a previously unoccupied bucket, and false if the new value replaces a previously inserted one. (Don't worry about rehashing the **Map** if more then three quarters of the buckets are occupied. You'll worry about that in part c.)

```
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;
}
/**
 * Enters the key-value pair into the Map.
 * If the specific key matches some previously inserted one,
 * then the old value is overwritten by the old one.
 * Otherwise, the new element is dropped into a previously
 * unoccupied bucket, and the logical size of Map increases by
 * one.
 */
bool Map::enter(string key, double value) {
      if (count > 3 * numBuckets / 4) {
      rehash(); // you'll implement this helper method in part c

      // TODO: Fill this in!
}
```

iii. Finally, implement the **rehash** method, which updates the **Map** so that it has twice as many buckets and all of its key-value pairs are rehashed to take up residence in a bucket where they could have resided had the new number of buckets been the number of buckets all along. (You'll benefit by figuring out how to call enter to help with the redistribution.)

```
/**
 * 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();
```