Practice Second Midterm Exam

Based on handouts by Eric Roberts and Jerry Cain

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 Tuesday, May 28. A solution set to this practice exam will be released online on Friday.

Time and place of the exam

The midterm exam is scheduled for 7PM – 10PM at several different locations (note that the exams are not in the regular lecture room), divided by last name:

- Last name starts with **Aar – Mai**: Go to **Hewlett 200**.
- Last name starts with **Mar – Sch**: Go to **Hewlett 201**.
- Last name starts with **Sco – Swe**: Go to **Hewlett 101**.
- Last name starts with **Sza – Wal**: Go to **Hewlett 102**.
- Last name starts with **Wan – Xue**: Go to **Hewlett 103**.
- Last name starts with **Yag – Zuo**: Go to **380-380Y**.

Coverage

The midterm covers the material presented in class through and including the lecture on Wednesday, May 22nd on binary search trees. Although all of the material that we have covered so far may appear on the test, the exam will primarily focus on class implementation, data structures, and algorithmic efficiency.

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 180. 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.

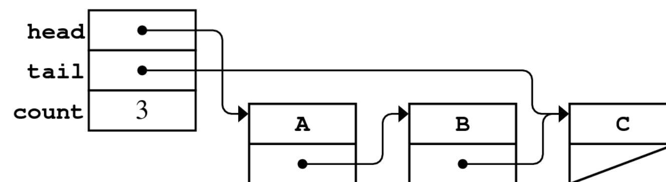
Normally, we would leave lots of space for you to write your answers, but in the interest of saving trees this handout is printed without much whitespace.

Problem One: Queue Reversal (40 Points)

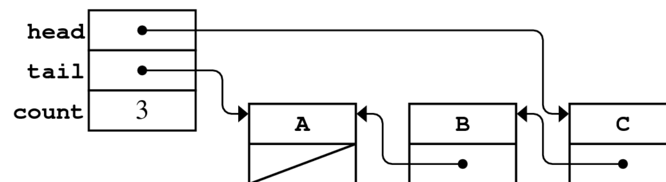
Suppose that you are implementing the **Queue** type using a linked list with head and tail pointers (as done in lecture) and are interested in adding the following member function:

```
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 Two: Rebuilding Binary Search Trees (40 Points)

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.

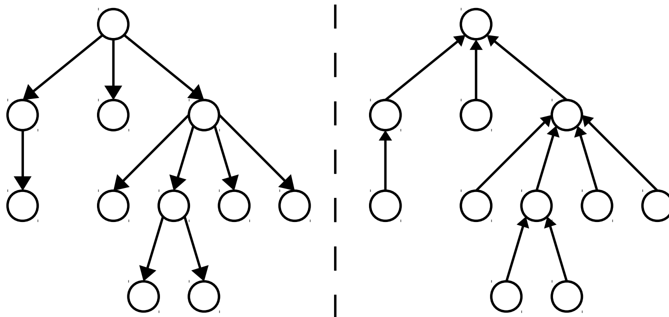
Write code for `fillVector` and `rebuildTree` to complete the implementation of the `rebalance` method.

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 `grow`) 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 Three: Spaghetti Stacks **(40 Points)**

When representing trees and tries in lecture, we had each node store a pointer to each of its children. This makes it easy to walk down from the root to any leaf node. However, we did not have nodes store pointers back to their parents, making it hard to walk up the tree from any node to the root.



Another representation for a tree adopts the opposite approach – each node stores a pointer to its parent node, but no parent stores a pointer back to its child. This representation is sometimes called a *spaghetti stack*. To the left of this paragraph is a tree represented using our standard representation and using the new spaghetti stack representation.

Spaghetti stacks differ from normal tree representations in two major ways. First, in our normal top-down tree representation, we can store the tree as a single pointer to the root node. This is sufficient to reach any node in the tree, because every node ultimately descends from the root. In a spaghetti stack, we store one pointer for each *leaf* in the tree. This enables us to find any node in the tree by beginning at the appropriate leaf and walking upward to the root.

Second, because each node in a spaghetti stack stores just a single pointer to its parent, it is possible to think of a spaghetti stack as a bundle of linked lists whose nodes overlap with one another. Given any specific node in the tree, the set of nodes we can reach by following that node's parent pointer upward forms a linked list connecting that node to the root of the tree. You can try this out with the above tree.

Suppose that you are given the following `struct` representing a node in a tree using the standard top-down representation. Each node can have any number of children:

```
struct TreeNode {
      string value;
      Vector<TreeNode*> children;
};
```

You are also given the following `struct`, which represents a node in a spaghetti stack:

```
struct SpaghettiNode {
      string value;
      SpaghettiNode* parent;
};
```

(i) Making Spaghetti **(20 Points)**

Write a function

```
Set<SpaghettiNode*> spaghettify(TreeNode* root);
```

that accepts as input a pointer to the root of a normal, top-down representation of tree, then constructs a copy of that tree as a spaghetti stack. Your function should return a `Set` containing pointers to all of the leaf nodes of that tree, and should not make any changes to the input tree.

(ii) Cleaning up Spaghetti **(20 Points)**

Write a function

```
void freeSpaghettiStack(Set<SpaghettiNode*> leaves);
```

that accepts as input a set of the leaf nodes in a spaghetti stack, then deallocates all nodes in that spaghetti stack. Be careful when implementing this function not to free the same node twice and to make sure that all nodes of the tree, not just the leaf nodes, are deallocated.

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

When implementing a hash table, we have to handle *hash collisions* where multiple elements hash to the same bucket. In lecture, we resolved collisions by storing all the elements with the same hash code in the same bucket (specifically, in a linked list). In this problem, you'll see another approach.

A *perfect hash function* is a hash function that, for a given set of objects and buckets, puts every object into its own bucket. In other words, a perfect hash function is one that causes no hash collisions. A *perfect hash table* is a hash table that distributes elements using a perfect hash function. This guarantees that each bucket has at most one element in it, which gives worst-case O(1) lookups. Nifty!

Note that there is no one “perfect hash function” that works for every set of elements; every hash function will cause collisions for some inputs. For example, given the numbers { 0, 1, 2, 3, 4 } and five buckets, the hash function $f(x) = x$ `% 5` puts each number into its own bucket. However, the same hash function applied to { 0, 5, 10, 15, 20 } would not be a perfect hash function, since each number would be dropped into bucket 0. Consequently, when building a perfect hash table, we have to try out several candidate hash functions to find one that produces no collisions.

Typically, perfect hash tables are used only when the elements to be stored in the table are known in advance. That way, we can construct the hash table as follows:

1. Choose a random hash function for the n elements.
2. Distribute the elements according to that hash function.
3. If there was a hash collision, go back to (1) and try again.

How exactly do we choose a random hash function? One way of choosing to do this is to build a hash function with the following signature:

```
int hashCode(string key, int whichFunction);
```

(Don't worry... we won't ask you to write this function. You can assume that it's provided for you.)

You can think of this not as a single hash function, but rather as a huge collection of hash functions, one for every **`int`**. The first parameter represents the key to hash, and the second parameter is an integer that says which of those hash functions should be used to compute the hash code. For example, calling **`hashCode(key, 0)`** computes the hash code for **`key`** using hash function 0, **`hashCode(key, 1)`** will compute the hash code for **`key`** using hash function 1, etc. To choose a random hash function, you can choose a random integer by calling **`randomInteger(INT_MIN, INT_MAX)`**, then can pass the result as the second parameter to **`hashCode`** whenever you want to find the hash code of some value. Like the hash function we saw in lecture, this function returns a positive integer, which you can then convert into a bucket index by modding it by the number of buckets.

One last detail: when storing n elements in a perfect hash table, you should create n^2 buckets for your hash table. This greatly improves the chance that a randomly-chosen hash function ends up being perfect.

Your task is to implement the constructor, destructor, and **`containsKey`** method for a perfect hash table. The next page contained a specification of a **`PerfectHashTable`** class. You should implement the three listed functions, plus any helper functions that you use in the course of your implementation.

```
class PerfectHashTable {
public:
    /* Constructor: PerfectHashTable(Set<string>& values);
     * Usage: PerfectHashTable(values);
     * ---------------------------------------------------------------------------
     * Constructs a perfect hash table storing the specified set of values.  The
     * number of buckets will be n², where n is the number of values.
     */
    PerfectHashTable(Set<string>& values);

    /* Destructor: ~PerfectHashTable();
     * Usage: (implicit)
     * ---------------------------------------------------------------------------
     * Frees all resources allocated by this PerfectHashTable.
     */
    ~PerfectHashTable();

    /* Member function: containsKey(string key);
     * Usage: if (pht.containsKey("dikdik")) { … }
     * ---------------------------------------------------------------------------
     * Returns whether the specified key is contained in the hash table.
     */
    bool containsKey(string key);

private:
    /* Type: Bucket
     * ---------------------------------------------------------------------------
     * A type representing a bucket in a perfect hash table.  Each bucket stores up
     * to one value.  The "isUsed" field determines whether or not the bucket has
     * an element in it.  If so, then the "value" field stores that value.  If not,
     * the contents of "value" are ignored.
     */
    struct Bucket {
        bool isUsed;
        string value;
    };

    Bucket* elems;      // Pointer to the dynamically-allocated array of buckets.
    int numBuckets;     // How many buckets exist in the hash table.
    int whichFunction; // Which hash function is used to hash entries.

    /* Add any other fields or private member functions that you wish. */









};
```

```
/* Function: hashCode(string key, int whichFunction);
 * Usage: int bucket = hashCode(key, whichFunction) % numBuckets;
 * ---------------------------------------------------------------------------------
 * Computes the hash code for the given key using the hash function specified by
 * whichFunction.  The return value is a nonnegative integer hash code given by
 * the hash function with the given index.
 */
int hashCode(string key, int whichFunction);


PerfectHashTable::PerfectHashTable(Set<string>& values) {
      // Implement this function
}

PerfectHashTable::~PerfectHashTable() {
      // Implement this function
}

bool PerfectHashTable::containsKey(string value) {
      // Implement this function
}
```

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?

(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.