Section Handout 5

Based on handouts by Jerry Cain, Julie Zelenski, and Eric Roberts

Problem One: Double-Ended Queues

An important data structure that we do not provide as part of the CS106B collections classes is the *double-ended queue*, often shortened to *deque* (pronounced "deck," as in a deck of cards). A deque is similar to a queue, except that elements can be added or removed from both ends of the deque.

Here is one possible interface for a `Deque` class:

```
class Deque {
public:
    Deque();
    ~Deque();

    /* Adds a value to the front or the back of the deque. */
    void pushFront(int value);
    void pushBack(int value);

    /* Returns and removes the first or last element of the deque. */
    int popFront();
    int popBack();
};
```

One efficient way of implementing a deque is as a doubly-linked list. The deque stores pointers to the head and the tail of the list to support fast access to both ends.

Design the `private` section of the `Deque` class, then implement the above member functions using a doubly-linked list.

Problem Two: Merge Sort Revisited (Again)

Although merge sort is typically slower than other sorting algorithms like heap sort and quicksort when applied to arrays, it is one of the fastest algorithms for sorting linked lists. Recall that merge sort works as follows: if the list has zero or one elements, it is already sorted. Otherwise:

- Split the list in half.
- Recursively sort each half.
- Merge the two halves together.

Suppose that you have a linked list cell defined as

```
struct Cell {
    int value;
    Cell* next;
};
```

Write a function

```
void mergeSort(Cell*& list);
```

That accepts as input a pointer to the first cell in a linked list, then uses merge sort to sort the linked list into ascending order. The function should change the pointer it receives as an argument so that it points to the first cell of the new linked list.

Problem Three: Scrambled Hashes

Below are three descriptions of hash functions that can be used to produce hash codes for English words. Each of these functions has a problem with it that would make it a poor choice for a hash function. For each of the hash functions, describe what the weakness is.

- Hash function 1: Always return 0.
- Hash function 2: Return a random `int` value.
- Hash function 3: Return the sum of the ASCII values of the letters in the word.

For problems 4 and 5, assume that `BSTNode` is defined as follows:

```
struct BSTNode {
   string key;
   BSTNode *left, *right;
};
```

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

In the first example of binary search trees, the text uses the names of the dwarves from Walt Disney's 1937 classic animated film, *Snow White and the Seven Dwarves.* Dwarves, of course, occur in other stories. In J. R. R. Tolkien's *The Hobbit*, for example, 13 dwarves arrive at the house of Bilbo Baggins in the following order:

Dwalin, Balin, Kili, Fili, Dori, Nori, Ori, Oin, Gloin, Bifur, Bofur, Bombur, Thorin

Diagram the binary search tree you get from inserting these names into an empty tree in this order. Once you have finished, answer the following questions about your diagram:

4a. What is the *height* (defined on page 681 as the number of nodes in the longest path from the root to a leaf) of the resulting tree?

4b. Which nodes are leaves?

4c. Which nodes, if any, are out of balance (in the sense that the subtree rooted at that node fails to meet the definition of balanced trees on page 696)?

4d. Which key comparisons are required to find the string `"Gloin"` in the tree?

Problem Five: Calculating the Height of a Binary Tree (Chapter 16, exercise 6, page 716)

Write a function

```
int height(BSTNode *tree);
```

that takes a binary search tree and returns its height.