Section Solutions 5

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

Problem One: Double-Ended Queues

```
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();

private:
    struct Cell {
        int value;
        Cell* next;
        Cell* prev;
    };
    Cell* head;
    Cell* tail;
};

/* Initially, there are no elements at all. */
Deque::Deque() {
    head = tail = NULL;
}

/* Standard linked-list deletion code. */
Deque::~Deque() {
    while (head != NULL) {
        Cell* next = head->next;
        delete head;
        head = next;
    }
}

void Deque::pushFront(int value) {
    /* Create the new cell to add. */
    Cell* cell = new Cell;
    cell->value = value;

    /* This cell is at the front of the list. */
    cell->next = head;
    cell->prev = NULL;

    /* If the list is empty, the new cell is now the sole element. */
    if (head == NULL) {
        head = tail = cell;
    }
    /* Otherwise, rewire the first element to point back at the new cell,
     * then update the head pointer.
     */
    else {
        head->prev = cell;
        head = cell;
    }
}
```

```
void Deque::pushBack(int value) {
    /* Create the new cell to add. */
    Cell* cell = new Cell;
    cell->value = value;

    /* This cell is at the back of the list. */
    cell->prev = tail;
    cell->next = NULL;

    /* If the list is empty, the new cell is now the sole element. */
    if (tail == NULL) {
        head = tail = cell;
    }
    /* Otherwise, rewire the last element to point into the new cell,
     * then update the tail pointer.
     */
    else {
        tail->next = cell;
        tail = cell;
    }
}

int Deque::popFront() {
    if (head == NULL) error("That which does not exist cannot be popped.");

    /* Cache the value to be removed, since we're going to free memory. */
    int result = head->value;
    Cell* toRemove = head;

    /* Advance the head to the next location. */
    head = head->next;

    /* There are two cases to consider.  First, if the deque is nonempty,
     * then we need to break the backward-pointing link to the cell we're
     * about to remove.
     */
    if (head != NULL) {
        head->prev = NULL;
    }
    /* Otherwise, we have to also update the tail pointer to be NULL. */
    else {
        tail = NULL;
    }

    /* Reclaim memory. */
    delete toRemove;

    return result;
}

/* … continued … */
```

```
int Deque::popBack() {
    if (tail == NULL) error("That which does not exist cannot be popped.");

    /* Cache the value to be removed, since we're going to free memory. */
    int result = tail->value;
    Cell* toRemove = tail;

    /* Retreat the tail to the previous location. */
    tail = tail->prev;

    /* There are two cases to consider.  First, if the deque is nonempty,
     * then we need to break the forward-pointing link to the cell we're
     * about to remove.
     */
    if (tail != NULL) {
        tail->next = NULL;
    }
    /* Otherwise, we have to also update the head pointer to be NULL. */
    else {
        head = NULL;
    }

    /* Reclaim memory. */
    delete toRemove;

    return result;
}
```

Problem Two: Merge Sort Revisited (Again)

Here is one possible implementation:

```
/* Given two pointers representing the first and last cells of a linked list,
 * along with a pointer to a linked list cell, appends that cell to the first list.
 */
void appendList(Cell*& head, Cell*& tail, Cell* toAppend) {
    if (tail == NULL) {
        head = tail = toAppend;
    } else {
        tail->next = toAppend;
        tail = tail->next;
    }
}

/* Given two sorted linked lists, merges those linked lists together into a single
 * sorted list and returns a pointer to the beginning of that list.
 */
Cell* merge(Cell* first, Cell* second) {
    /* Track two pointers - one to the head and tail of the result list. */
    Cell* head = NULL;
    Cell* tail = NULL;

    /* As long as both of the lists aren't empty, continuously compare the first
     * elements of each list and take the smaller one out.
     */
    while (first != NULL && second != NULL) {
        /* Determine which cell is smaller, then remove it from its list. */
        Cell* smaller;
        if (first->value < second->value) {
            smaller = first;
            first = first->next;
        } else {
            smaller = second;
            second = second->next;
        }

        /* Break the removed cell's link. */
        smaller->next = NULL;

        /* Insert this element into the sorted linked list. */
        appendList(head, tail, smaller);
    }

    /* Now, at least one list is empty.  Whichever list that is needs to get
     * appended to the result list.
     */
    appendList(head, tail, first != NULL? first : second);

    /* Return a pointer to the head of the linked list. */
    return head;
}

/* … continued … */
```

```
/* Uses merge sort to sort a linked list. */
void mergeSort(Cell*& list) {
   /* Check for base cases - is the list empty or a singleton? */
   if (list == NULL || list->next == NULL) return;

   /* Split the list into two smaller lists.  We'll split the list by moving the
    * even-index cells into one list and the odds into another.
    */
   Cell* evenHead = NULL;
   Cell* evenTail = NULL;
   Cell* oddHead = NULL;
   Cell* oddTail = NULL;

   int counter = 0;
   while (list != NULL) {
      /* Break off the first cell from the rest of the list. */
       Cell* curr = list;
       list = list->next;
       curr->next = NULL;

       /* Insert into the appropriate list. */
       if (counter % 2 == 0) {
           appendList(evenHead, evenTail, curr);
       } else {
           appendList(oddHead, oddTail, curr);
       }

       counter++;
   }

   /* Recursively sort each half. */
   mergeSort(evenHead);
   mergeSort(oddHead);

   /* Merge them together. */
   list = merge(evenHead, oddHead);
}
```

Problem Three: Scrambled Hashes

- Hash function 1: Always return 0.

This is a valid hash function, but because every string will get hashed into the same bucket the cost of looking up values in the hash table will be O(*n*).

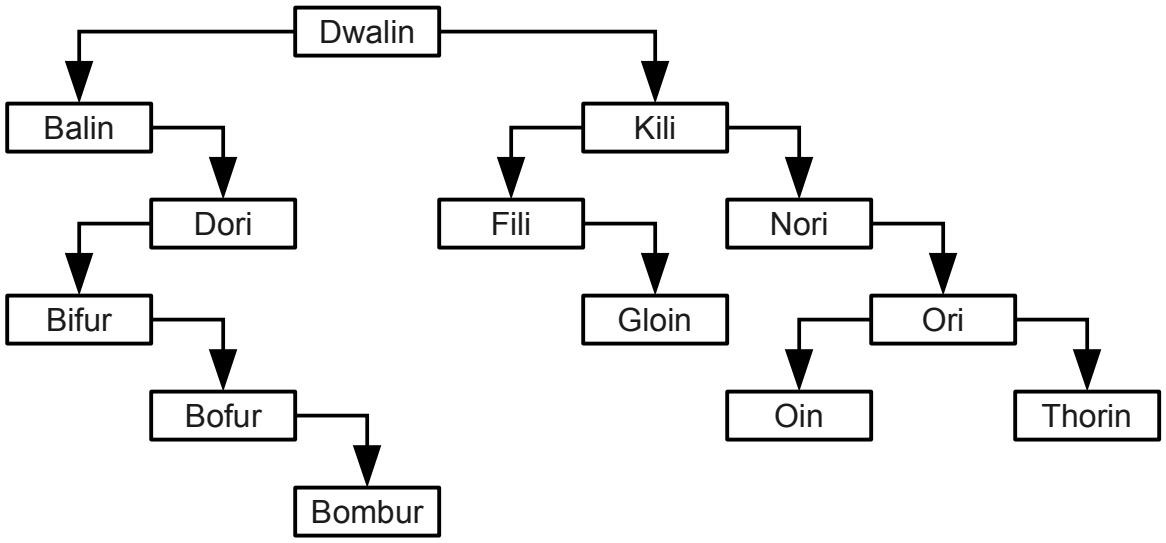
- Hash function 2: Return a random `int` value.

This is not a valid hash function because hashing the same string multiple times will produce different hash codes. Consequently, you won't be able to look up any strings in the hash table!

- Hash function 3: Return the sum of the ASCII values of the letters in the word.

This hash function is better than the first, but will not distribute the words evenly. Note that any two words with the same letter frequencies will have the same hash code (for example, “table” and “bleat”). Some words with the same length might also collide if they have two letters that have shifted from one another by the same amount; for example, “mass” and “last.”

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



1a. What is the height of the resulting tree? **6**

1b. Which nodes are leaves? **`Bombur`**, **`Gloin`**, **`Oin`**, and **`Thorin`**

1c. Which nodes are out of balance? **`Balin`**, **`Bifur`**, **`Dori`**, **`Dwalin`**, **`Kili`**, and **`Nori`**

1d. Which key comparisons are required to find the string **`"Gloin"`** in the tree?

`"Gloin" > "Dwalin"`, **`"Gloin" < "Kili"`** , **`"Gloin" > "Fili"`**, and **`"Gloin" == "Gloin"`**

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

```
int height(BSTNode* root) {
    if (root == NULL) return 0;

    return 1 + max(height(root->left), height(root->right));
}
```