Section Handout 7

Problem One: Linked List Mechanics

This section handout is almost exclusively about linked lists, so before we jump into some of their applications, let's start off by reviewing some of the basic mechanics about how they work!

To begin with, let's imagine we have a linked list of integers. Go and define a `Cell` `struct` representing a single cell in the linked list. Then, write a function

```
int sumOfElementsIn(Cell* list);
```

that adds up the values of all the elements in the linked list. Write this function two ways – first, do it iteratively; then, do it recursively. Which one did you think was easier to write? Why?

Next, write a function

```
Cell* lastElementOf(Cell* list);
```

that returns a pointer to the last element of a linked list (and reports an error if the list is empty). Again, write this function two ways, iteratively and recursively. Which one did you think was easier to write?

Problem Two: Tail Pointers

In lecture, we wrote a function to read a list of values from the user and return a linked list containing those values (and we wrote it two different ways, too!) The iterative version of that function had the odd property that it returned the elements that were read in reverse order, which was a consequence of the fact that we kept adding elements at the front of the list that we'd made.

Write an *iterative* function

```
Cell* readList();
```

that reads a list of values from the user. It should return a linked list containing those values in the order in which they were entered. To make it run in time O(n), where n is the number of elements read, maintain a tail pointer keep track of the very last element in the list.

Problem Three: Pointers by Reference

One of the trickier nuances of linked lists comes up when we start passing around pointers as parameters by reference. To better understand exactly what that's all about, trace through the following code and show what it prints out. Also, identify any memory leaks that occur in the program.

```
void confuse(Cell* list) {
    list->value = 137;
}

void befuddle(Cell* list) {
   list = new Cell;
   list->value = 42;
   list->next = nullptr;
}

void confound(Cell* list) {
    list->next = new Cell;
    list->next->value = 2718;
    list->next->next = nullptr;
}

void bamboozle(Cell*& list) {
    list->value = 42;
}

void mystify(Cell*& list) {
    list = new Cell;
    list->value = 161;
    list->next = nullptr;
}

int main() {
    Cell* list = /* some logic to make the list 1 → 3 → 5 → null */

    confuse(list);
    printList(list); // from lecture

    befuddle(list);
    printList(list);

    confound(list);
    printList(list);

    bamboozle(list);
    printList(list);

    mystify(list);
    printList(list);

    freeList(list); // from lecture
    return 0;
}
```

Problem Four: Concatenating Linked Lists

Write a function

```
Cell* concat(Cell* one, Cell* two);
```

that takes as input two linked lists, then concatenates the second list onto the back of the first linked list. Your function should return a pointer to the first element in the resulting list.

Then, update the function so that it has this signature:

```
void concat(Cell*& one, Cell* two);
```

This function should proceed as before, except that instead of returning the new head of the linked list, it changes the pointer given as a first parameter so that it holds the new head of the combined list. Then, answer this question: why didn't we make `two` a reference parameter as well?

Problem Five: The Classic Interview Question

Here's a classic interview question that's so overused that it's almost an in-joke among software engineers: write a function that reverses a linked list. Do it both iteratively and recursively, and aim to get your code to run in time O(n).

Problem Six: Doubly-Linked Lists

The linked lists we talked about in lecture are called ***singly-linked lists*** because each cell just stores a single link pointer, namely, one to the next element in the list. A common variant on linked lists is the ***doubly-linked list***, where each cell stores two pointers – a pointer to the next element in the list (as before) and a pointer to the previous element in the list. Here's what a cell in a doubly-linked list might look like:

```
struct Cell {
    string value; // Or whatever type of data goes here
    Cell* next;
    Cell* prev;
};
```

Doubly-linked lists have one really nice property: it is *really* easy to splice a new element into or out of a doubly-linked list. Write a function

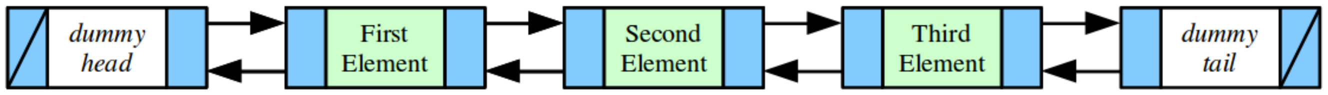
```
void insertBefore(Cell*& head, Cell* beforeMe, Cell* newCell);
```

that takes as input a pointer to the first element in a doubly-linked list, a pointer to a cell somewhere in the linked list (`beforeMe`), and a newly-allocated `Cell` object, then splices the new cell into the doubly-linked list right before the cell `beforeMe`. Your function should update head so that when the function returns, it still points at the first cell in the linked list. (Why is it necessary to pass in the head of the list?) You can assume that `beforeMe` is not null.

Problem Seven: Dummy Nodes

When working with linked lists, it's common to encounter some weird edge cases when growing a zero-element list into a one-element list or shrinking a one-element list to a zero-element list, since in those cases you typically need to modify some external head and tail pointers. One technique that simplifies the logic in these cases is to add ***dummy nodes*** to the linked list. A dummy node is a node that's *technically* a part of a linked list, but is specifically intended to signify "the start of the list" or "the end of the list."

For example, let's imagine we have a doubly-linked list. We might insert dummy nodes before the first element of the list and after the last element of the list, as shown here:



First, write a function

```
Cell* makeEmptyList();
```

that creates a new doubly-linked list with a dummy head and tail, then returns a pointer to the head.

Now, write a function

```
void printList(Cell* head, Cell* tail);
```

that prints out the contents of a doubly-linked list whose dummy first and last elements are pointed at by the head and tail parameters. Next, write a pair of functions

```
void insertBefore(Cell* newCell, Cell* beforeMe);
void insertAfter(Cell* newCell, Cell* afterMe);
```

that take as inputs a newly-allocated cell to insert into the list and a cell that comes right after or right before the cell to insert, then splices the new cell into the list. You can assume that no one will try to insert anything before the head or after the tail, though they could do something like insert before the tail or after the head to append or prepend a new cell to the list.

Next, write a pair of functions

```
void append(Cell* tail, Cell* newCell);
void prepend(Cell* head, Cell* newCell);
```

that insert the specified element at the beginning or end of the linked list. Then, write a function

```
void remove(Cell* toRemove);
```

that removes an element from a doubly-linked list with a dummy head and tail and deallocates the cell. Finally, write a function

```
void freeList(Cell* head);
```

that frees the list whose first element is `head`.

How did that compare with the regular doubly-linked list from before?

Problem Eight: Double-Ended Queues

This problem concerns a data structure called a ***double-ended queue***, or ***deque*** for short (it's pronounced "deck," as in a deck of cards). A deque is similar to a stack or queue in that it represents a sequence of elements, 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();

    /* Seems like all containers have the next two functions. :-) */
    bool isEmpty() const;
    int  size() const;

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

    /* Looks at, but does not remove, the first/last element of the deque. */
    int peekFront() const;
    int peekBack() const;

    /* 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. As a hint, this is way easier to do using dummy nodes!

Problem Nine: Quicksort

The ***quicksort*** algorithm is a sorting algorithm that, in practice, tends to be one of the fastest sorts available. It's a recursive algorithm that works as follows. If the list of values has at most one element in it, it's already sorted and there's nothing to do. Otherwise:

- Choose a ***pivot element***, typically the first element of the list.
- Split the elements in the list into three groups – elements less than the pivot, elements equal to the pivot, and elements greater than the pivot.
- Recursively sort the first and last of these groups.
- Concatenate the three lists together.

Write a function

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

That accepts as input a pointer to the first cell in a singly-linked list of integers, then uses quicksort 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. ***This is a fairly large piece of code***, so make sure that you pick a good decomposition. With the right decomp, this code is quite easy to read.