Section Solutions 6

Grab a pencil and paper, or a whiteboard, or a reed and papyrus when you're reading over these solutions. The pointer juggling here is intricate but beautiful, like a delicate flower. If you work through everything one line at a time, you'll get a much deeper sense for what's going on here. Simply reading over this code and trying to keep track of everything in your head is not likely to work out very well.

Problem One: Linked List Mechanics

First, we need a `Cell` structure! Here's one possibility:

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

Most linked list cells look more or less the same – they have some data and a pointer to the next element in the list. Here's two version of the code to sum up the elements of one of these lists:

```
/* Iterative version */
int sumOfElementsIn(Cell* list) {
    int result = 0;
    for (Cell* curr = list; curr != nullptr; curr = curr->next) {
        result += curr->value;
    }
    return result;
}

/* Recursive version. */
int sumOfElementsIn(Cell* list) {
    /* The sum of the elements in an empty list is zero. */
    if (list == nullptr) return 0;

    /* The sum of the elements in a nonempty list is the sum of the elements in
     * the first cell plus the sum of the remaining elements.
     */
    return list->value + sumOfElementsIn(list->next);
}
```

And two versions of a function to get the last element of a linked list.

```
/* Iterative version */
Cell* lastElementOf(Cell* list) {
    if (list == nullptr) error("Empty lists have no last element.");

    /* Loop forward until the current cell's next pointer is null. That's the
     * point where the list ends.
     */
    Cell* result = list;
    while (result->next != nullptr) {
        result = result->next;
    }
    return result;
}
```

```
/* Recursive version. */
Cell* lastElementOf(Cell* list) {
    /* Base Case 1: The empty list has no last element. */
    if (list == nullptr) error("Nope!");

    /* Base Case 2: The only element of a one-element list is the last element. */
    if (list->next == nullptr) return list;

    /* Recursive Case: There's at least two cells in this list. The last element
     * of the overall list is the last element of the list you get when you drop
     * off the first element.
     */
    return lastElementOf(list->next);
}
```

Problem Two: Cleaning Up Linked Lists

The second version of the code has the issue we saw from lecture: it blows up the cell, then tries to read from its `next` pointer, which causes *undefined behavior*! This is a Very Bad Thing.

The interchanging of the order of the lines is interesting for another reason, though. The recursive logic in both cases essentially has two parts: process the current cell, and process the rest of the list. In the correct case, we first destroy the remainder of the list so that all the memory except the first cell is freed, then we free the first cell. In the incorrect case, we first free the current cell, then try to free the rest of the list… having forgotten where it is!

We can generalize this idea to process linked lists in several different ways recursively. We can either process the first cell, then the rest of the list, or we can process the rest of the list, then the current cell. As a result, if we wanted to print out the linked list in reverse, we would do so as follows: first, print the rest of the list in reverse order, then print the first element. That's shown here, in a remarkably short piece of code:

```
void printReversed(Cell* list) {
    /* Base Case: Printing the empty list requires no action. */
    if (list == nullptr) return;

    /* Recursive Case: Print the rest of the list in reverse, then print the
     * current cell afterwards.
     */
    printReversed(list->next);
    cout << list->value << endl;
}
```

Problem Three: Tail Pointers

Here's one option, which is closely related to our logic to get the queue to work in worst-case O(1) time. I'm assuming the list holds strings, but really this'll work for any type as long as there's a sentinel value:

```
Cell* readList() {
    Cell* head = nullptr;
    Cell* tail = nullptr; // There is no last element... at least, not yet. :-)

    while (true) {
        string line = getLine("Next entry: ");
        if (line == "") break;

        /* Get the cell basics set up. It always goes on the end, so its next
         * pointer is always null.
         */
        Cell* cell  = new Cell;
        cell->value = line;
        cell->next  = nullptr;

        /* If the list is empty, this is now both the head and the tail. */
        if (head == nullptr) {
            head = tail = cell;
        }
        /* Otherwise, splice this element in right after the tail. */
        else {
            tail->next = cell;
            tail = cell;
        }
    }
    return head;
}
```

Problem Four: Pointers by Reference

Let's go through this one step at a time.

- The call to `confuse` updates the first element of the list to store 137, so the call to `printList` will print out 137, 3, 5.
- The call to `befuddle` takes its argument by value. That means it's working with a *copy* of the pointer to the first element of the list, so when we set `list` to be a new cell, it doesn't change where the `list` variable back in `main` is pointing. The cell created here is leaked, and the next call to `printList` will print out 137, 3, 5.
- The call to `confound` takes its argument by value. However, when it writes to `list->next`, it's following the pointer to the first element of the linked list and changing the actual linked list cell it finds there. This means that the list is modified by dropping off the 3 and the 5 (that memory gets leaked) and replacing it with a cell containing 2718. Therefore, the next call to `printList` will print out 137, 2718.
- The call to `bamboozle` takes its argument by reference, but notice that it never actually reassigns the `next` pointer. However, it *does* change the memory in the cell at the front of the list to hold 42, so the next call to `printList` will print 42, 2718.
- The call to `mystify` takes its argument by reference and therefore when it reassigns `list` it really is changing where `list` back in `main` is pointing. This leaks the memory for the cells containing 42 and 2718. The variable `list` back in `main` is changed to point at a new cell containing 161, so the final call to `printList` prints 161.
- Finally, we free that one-element list. Overall, we've leaked a lot of memory!

Problem Five: Concatenating Linked Lists

Here's one possible solution using that nice and handy function we wrote in Problem One of this section handout! It's a great exercise to do this from first principles as well.

```
Cell* concat(Cell* one, Cell* two) {
    /* If the first list is empty, then the result is just the second list. */
    if (one == nullptr) return two;

    /* Otherwise, take the last element of the first list and have it point to
     * the first element of the second list. The first element overall is
     * unchanged.
     */
    lastElementOf(one)->next = two;
    return one;
}
```

If we opt to use reference parameters, the core logic is the same, but the execution is a bit different:

```
void concat(Cell*& one, Cell* two) {
    if (one == nullptr) {
        one = two;
    } else {
        lastElementOf(one)->next = two;
    }
}
```

Problem Six: The Classic Interview Question

Here's an iterative version of the function. It's remarkably similar to the code we came up with to read in the contents of a linked list from the user, except that the elements that we read get pulled off of the main linked list.

```
void reverse(Cell*& list) {
    Cell* head = nullptr; // New pointer to the head of the linked list.

    while (list != nullptr) {
        /* Store the next element of the list in an auxiliary pointer so we don't
         * lose it in the next step.
         */
        Cell* next = list->next;

        /* Move this cell onto the front of the new list we're building. */
        list->next = head;
        head = list;

        /* Move to the new front of the old linked list. */
        list = next;
    }

    list = head;
}
```

Recursively, the idea is to pull the first element off the list, recursively reverse the rest of the list, then tack the new element onto the end of the list. To make things run quickly, we'll have the recursive logic return a pointer to the very last element of the reversed list so that we can put the cell that used to be at the front of the list into its proper place.

```
/* Reverses a linked list and returns a pointer to its last element. */
Cell* reverseRec(Cell* list) {
    /* Base Case 1: An empty or single-element list is its own reverse. */
    if (list == nullptr || list->next == nullptr) return list;

    /* Recursive Case: Pull this element off the list and reverse what's left. */
    Cell* last = reverseRec(list->next);

    /* We know that last is non-null, because this list has at least two elements.
     * So tack this element onto the back.
     */
    last->next = list;

    /* Finally, remember that we're now at the back. */
    list->next = nullptr;
    return list;
}

void reverse(Cell*& list) {
    /* Store the last element for later use - we'll need to update the list head
     * pointer at the end.
     */
    Cell* result = lastElementOf(list);
    reverseRec(list);
    list = result;
}
```

Problem Seven: Doubly-Linked Lists

As before, we're going to need to have a `Cell` type we can use for our doubly-linked list. Here's one way to do this:

```
struct Cell {
    string data; // Or whatever type you'd like
    Cell* next;
    Cell* prev;
};
```

Here's how we might read one of these lists. This is basically the same code as before with some extra logic to maintain the previous pointers.

```
Cell* readList() {
    Cell* head = nullptr;
    Cell* tail = nullptr; // There is no last element... at least, not yet. :-)

    while (true) {
        string line = getLine("Next entry: ");
        if (line == "") break;

        /* Get the cell basics set up. It always goes on the end, so its next
         * pointer is always null.
         */
        Cell* cell  = new Cell;
        cell->value = line;
        cell->next  = nullptr;

        /* The preceding cell is whatever was the tail. */
        cell->prev = tail;

        /* If the list is empty, this is now both the head and the tail. */
        if (head == nullptr) {
            head = tail = cell;
        }
        /* Otherwise, splice this element in right after the tail. */
        else {
            tail->next = cell;
            tail = cell;
        }
    }
    return head;
}
```

To splice something into the list right before some point, we need to juggle the pointers so that whatever comes before us knows to route into us and whatever comes after us knows to step backwards into us. Here's what that looks like:

```
void insertBefore(Cell*& head, Cell* beforeMe, Cell* newCell) {
    /* If we're inserting before the head, there is no preceding cell, so we need
     * to special-case the logic.
     */
    if (beforeMe == head) {
        newCell->next = head;
        newCell->prev = nullptr;
        head->prev = newCell;
        head = newCell;
    }
    /* Otherwise, four pointers need to change: our own next/previous pointers so
     * so that this cell fits nicely into the list, plus the next field of the
     * cell before us and the prev field of the cell after us. We have to be
     * careful with the order in which we do this, though!
     */
    else {
        newCell->prev = beforeMe->prev;
        newCell->next = beforeMe;
        newCell->prev->next = newCell;
        newCell->next->prev = newCell;
    }
}
```

Problem Eight: Dummy Nodes

The `makeEmptyList` function is rather easy to write, since we just make two nodes and wire them together.

```
Cell* makeEmptyList() {
    Cell* head = new Cell;
    Cell* tail = new Cell;
    head->next = tail;
    tail->prev = head;
    head->prev = tail->next = nullptr;
    return head;
}
```

To print out the list, we start one step past the head and stop as soon as we get to the tail. Here's both an iterative and a recursive version of this code, just for funzies:

```
void printList(Cell* head, Cell* tail) {
    for (Cell* curr = head->next; curr != tail; curr = curr->next) {
        cout << curr->value << endl;
    }
}

void printRec(Cell* curr, Cell* tail) {
    if (curr == tail) return;

    cout << curr->value << endl;
    printRec(curr->next, tail);
}

void printList(Cell* head, Cell* tail) {
    printRec(head->next, tail);
}
```

The `insertBefore` and `insertAfter` functions are dramatically easier to write than what we saw in the previous problem simply because we can eliminate the edge case of having to change the head pointer (it *always* points to the dummy node regardless of what the *logically* first element of the list is) or tail pointer. Notice that we don't even need to know what the head or tail are! Maybe we're inserting right before or right after them – we don't know, and we don't need to care!

```
void insertBefore(Cell* newCell, Cell* beforeMe) {
    newCell->next = beforeMe;
    newCell->prev = beforeMe->prev;
    newCell->next->prev = newCell;
    newCell->prev->next = newCell;
}

void insertAfter(Cell* newCell, Cell* afterMe) {
    newCell->prev = afterMe;
    newCell->next = afterMe->next;
    newCell->next->prev = newCell;
    newCell->prev->next = newCell;
}
```

The nifty part is that we can easily use the above code to implement append and prepend. Look at this!

```
void append(Cell* tail, Cell* newCell) {
    insertBefore(tail, newCell);
}

void prepend(Cell* head, Cell* newCell) {
    insertAfter(head, newCell);
}
```

And how do we remove something? Well, we know that there will *always* be a node right before or right after us, so we can just splice around the node to remove!

```
void remove(Cell* toRemove) {
    toRemove->next->prev = toRemove->prev;
    toRemove->prev->next = toRemove->next;
    delete toRemove;
}
```

The best part is that deleting one of these linked lists is *exactly the same* as deleting a regular linked list – the dummy nodes are nodes just like any other node, so we just iterate across them and free them. In fact, since it's *literally* the same code as in the regular case, I'm not going to include any code here for it, since that would just be duplicating the code from lecture. ☺

Linked lists with dummy nodes are used extensively in systems programming. If you take CS140, for example, you'll use them to represent different processes running in the operating system moving between different states.

Problem Nine: Double-Ended Queues

If you think about it, the logic from the previous problem really nicely lets us build a deque – we have the ability to splice things in anywhere we want and splice things out anywhere we want, which is precisely what we'd need to do here.

For completeness' sake, I've included all the code necessary to get the deque working in this solution, even though we could easily have recycled some code. First, the header:

```
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();

private:
    struct Cell {
        int value;
        Cell* next;
        Cell* prev;
    };
    Cell* head;
    Cell* tail;
    int   numElems; // Cache for efficiency; makes size() run in time O(1).

    /* Creates a new cell initialized to a given value, but whose next and prev
     * pointers are uninitialized. This cell is intended to be used inside the
     * linked list, and therefore the size field is adjusted appropriately.
     */
    Cell* makeCell(int value);

    /* Destroys the given cell, which is presumed to be in the linked list. */
    void destroy(Cell* toRemove);
};
```

Now, the .cpp file:

```
#include "error.h" // Because bad things happen.

Deque::Deque() {
    /* Set up the empty dummied, doubly-linked list. */
    head = new Cell;
    tail = new Cell;
    head->next = tail;
    tail->prev = head;
    head->prev = tail->next = nullptr;

    numElems = 0;
}

Deque::~Deque() {
    /* Delete all cells in the list, including the head and tail. */
    while (head != nullptr) {
        Cell* next = head->next;
        delete head;
        head = next;
    }
}

/* Because we’ve cached the size, we don’t need to scan through the list when
 * we want to determine how many elements there are.
 */
int Deque::size() const {
    return numElems;
}

/* Good programming exercise: suppose that we didn’t cache the number of elements
 * in the list and that size() had to scan over the entire list in time O(n). How
 * would you implement isEmpty() given that we have a head and tail pointer?
 */
bool Deque::isEmpty() const {
    return size() == 0;
}

/* Our helper function that makes a cell. We could have alternatively defined a
 * constructor on the Cell type (yes, you can do that!), but I chose to do things
 * this way to show off Yet Another Memorable Piece of C++ Syntax. Since the
 * Cell type is nested inside Deque, the return type of this function has to be
 * Deque::Cell*, indicating that Cell is defined inside of Deque.
 */
Deque::Cell* Deque::makeCell(int value) const {
    Cell* result = new Cell;
    result->value = value;

    numElems++;
    return result;
}
```

```
/* pushFront is essentially insertAfter with the head pointer. */
void Deque::pushFront(int value) {
    Cell* cell = makeCell(value);
    cell->prev = head;
    cell->next = head->next;
    cell->prev->next = cell;
    cell->next->prev = cell;
}

/* pushBack is essentially insertBefore with the tail pointer. */
void Deque::pushBack(int value) {
    Cell* cell = makeCell(value);
    cell->next = tail;
    cell->prev = tail->prev;
    cell->prev->next = cell;
    cell->next->prev = cell;
}

/* To look at the front or back, we have to skip over the head or tail nodes,
 * since they're dummies and don't actually have any data in them.
 */
int Deque::peekFront() const {
    if (isEmpty()) error("This is why we can't have nice things.");
    return head->next->value;
}

int Deque::peekBack() const {
    if (isEmpty()) error("Thanks, Obama.");
    return tail->prev->value;
}

/* The destroy operation is essentially our remove function from earlier.
 * Notice that we do NOT need to use the full name Deque::Cell here because Cell
 * is an argument to a member function that's part of the Deque type.
 */
void Deque::destroy(Cell* toRemove) {
    toRemove->next->prev = toRemove->prev;
    toRemove->prev->next = toRemove->next;

    /* At this point it's spliced out of the list. */
    delete toRemove;
    numElems--;
}

/* popFront and popBack are essentially just wrapped splice-outs. */
int Deque::popFront() {
    int result = peekFront();
    destroy(head->next);
    return result;
}

int Deque::popBack() {
    int result = peekBack();
    destroy(tail->prev);
    return result;
}
```

Problem Ten: Quicksort

```
void quicksort(Cell*& list) {
    /* Determine the length of the list.  If it's length 0 or 1, we're done. */
    if (list == nullptr || list->next == nullptr) return;

    /* Remove the first element as the pivot element. */
    Cell* pivot = list;
    Cell* rest = pivot->next;

    /* Remove the pivot element from the list. */
    pivot->next = nullptr;

    /* Distribute the elements into the three lists based on  whether they are
     * smaller than, equal to, or greater than the pivot.
     */
    Cell* smaller = nullptr;
    Cell* bigger = nullptr;
    partitionList(rest, smaller, pivot, bigger);

    /* Recursively sort the two smaller regions. */
    quicksort(smaller);
    quicksort(bigger);

    /* Concatenate everything together using our concatenation functions from
     * before! For efficiency’s sake, we concatenate the “equal” and “greater”
     * lists, then tack those onto the end of the “smaller” list. (Why?)
     */
    concatLists(pivot, bigger);
    concatLists(smaller, pivot);
    list = smaller;
}

/* Prepends the given single cell to the given list, updating the pointer to
 * the first element of that linked list.
 */
void prependCell(Cell* toPrepend, Cell*& list) {
    toPrepend->next = list;
    list = toPrepend;
}

void partitionList(Cell* list, Cell*& smaller, Cell*& pivot, Cell*& bigger) {
    /* Distribute cells in the list into the three groups. */
    while (list != NULL) {
        /* Remember the next pointer, because we're going to remove this
         * element from the list it is currently in.
         */
        Cell* next = list->next;

        /* Determine which list this element belongs to. */
        if (list->value == pivot->value) {
            prependCell(list, pivot);
        } else if (list->value < pivot->value) {
            prependCell(list, smaller);
        } else {
            prependCell(list, bigger);
        }
        list = next;
    }
}
```