Midterm Review Problems

The midterm will be on Friday, April 28th from 2:15 to 4:15 pm in Kresge Auditorium. The exam will be open notes, open books, etc. and will cover material up to chapter 11.1.

Recursion and big-O - Odd Even Sort

Your job in this problem is to figure out the Big-O performance, in the worst case, for the following sort algorithm:

```
void OddEvenSort(int array[], int n)
{
    int i, swaps;

    while (TRUE)
    {
        swaps = 0;
        for (i = 0; i < n - 1 ; i += 2)
            if (array[i] > array[i + 1])
            {
                Swap(array, i, i + 1);
                swaps++;
            }
        for (i = 1; i < n - 1 ; i += 2)
            if (array[i] > array[i + 1])
            {
                Swap(array, i, i + 1);
                swaps++;
            }
        if (swaps == 0) return;
    }
}

void Swap(int array[], int i, int j)
{
    int temp;

    temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```

The algorithm makes multiple passes through the array. On each pass it first is compares the even-numbered elements of the array with the elements that follow them, swapping any pairs that are out of order. Then it compares the odd-numbered elements with the elements that follow them, again swapping any pairs that are out of order. When it makes it through both these compare loops without swapping any elements, the array is sorted.

(continued on the next page)

Part (a) The best case for this algorithm is an array that is already sorted. What is the Big-O performance of the algorithm (as a function of N, the size of the array) in the best case, and why? (Just writing the Big-O expression is not sufficient to earn points on this question--you have to explain how you arrived at that answer.)

Part (b) The worst case for this algorithm is an array that is exactly backwards. What is the Big-O performance of the algorithm (as a function of N, the size of the array) in the worst case, and why? (Just writing the Big-O expression is not sufficient to earn points on this question--you have to explain how you arrived at that answer.)

Recursion - Your Friends and Boggle!

After you've shown your Boggle assignment to your friends, they all wanted to try it out. So one night, you all got together and decided to play a bit of competitive Boggle. Problem is, you can't decide how to split up the teams. When you try to do it evenly, people complain that Julie has a better vocabulary (its because of her screen saver), or that Yves doesn't know anything (for example). In the end, you come up with a solution. You decide to split everyone into teams so that the combined IQ of each group is equal. Can this be done? You decide to write a function to find that out.

Your job now is to write a function that will see if this is possible. This is the prototype of your function:

```
bool CanEvenlySplit(int IQ[], int nPeople, int nTeams)
```

`IQ` is an array of the people's IQ's, `nPeople` is the number of people there, and `nTeams` is the number of teams you want to split into. Remember, you must use recursion to solve this problem.

ADT's Implementation Side- Arrow Keys Galore

Your task in this problem is to extend the `buffer.h` interface to help support navigation in a multi-line document. In all modern text editors, text may fall on different lines of the display. If you enter a new-line character, '**`\n`**', you create a new line and can continue entering text there. If you press the *down-arrow* key on the keyboard, and there is a line of text below the current line, the **`cursor`** moves in between the characters immediately below its current position.

Assume we begin with the **singly-linked-list** implementation of `buffer.h` as seen on p. 405 of the text. In order to facilitate line by line navigation, let's augment the data structure with an additional linked-list of **`lineT`**'s, defined as follows:

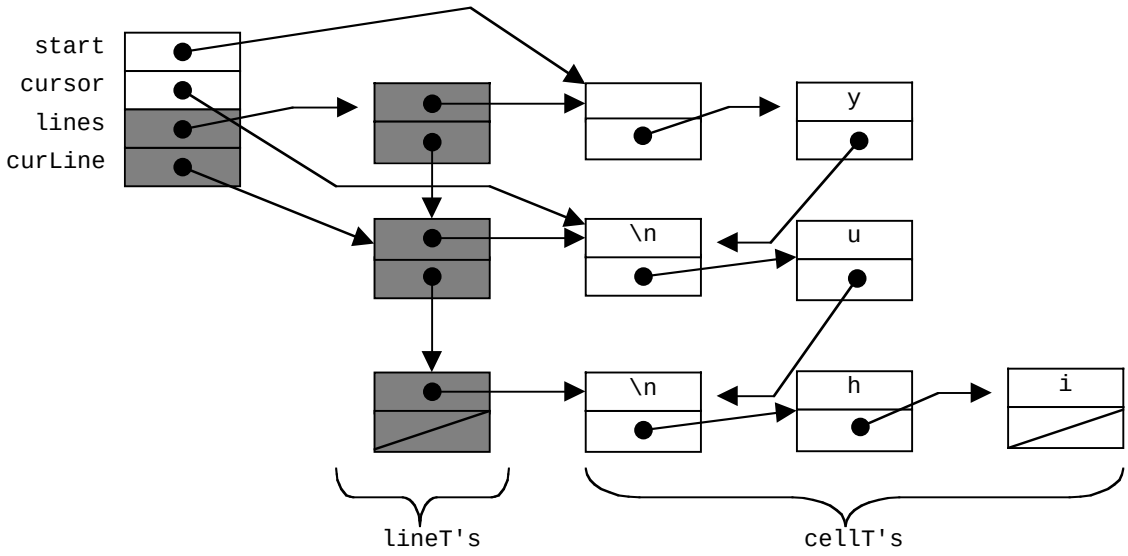
```
typedef struct lineT{
      cellT *ln;
      lineT *link;
} lineT;
```

Here each **`lineT`** has a pointer to the first **`cellT`** on its line of characters, as well as a pointer to the next line down. Now we'll modify our buffer itself to hold this line information:

```
struct bufferCDT {
      cellT *start;
      cellT *cursor;
      lineT *lines;
      lineT *curLine;
};
```

Here, **`lines`** points to the start of the list of **`lineT`**'s and **`curLine`** points to the line the **`cursor`** can currently be found on, as shown in the diagram below.

Many of the original `buffer.h` functions will behave as they always have since our modifications to the data structure don't destructively change the older version. In the diagram below, the new portions of the data structure are highlighted in gray:



Notice that all of the characters are still linked together as a single list, and that `start` points to the usual dummy cell at the beginning of that list. The last character in the first line links down to the newline at the beginning of the second line. By putting the newlines at the beginning of the second and following lines, they can serve as a kind of dummy cell for those lines.

Part (a) Modify the **`InsertCharacter`** function found on p. 407 of the text to support lines and maintain a data structure like the one seen above. For your convenience, here is the code from the text:

```
void InsertCharacter(bufferADT buffer, char ch)
{
    cellT *cp;

    cp = New(cellT *);
    cp->ch = ch;
    cp->link = buffer->cursor->link;
    buffer->cursor->link = cp;
    buffer->cursor = cp;
}
```

When implementing your new version of the function, keep in mind the following things:

- For normal characters, the function should behave as before.
- When the function receives a '**`\n`**' character, you need to create the appropriate **`lineT`** and **`cellT`**; then you need to update related pointers accordingly. The cursor should be placed so that it logically follows the '**`\n`**' character. The diagram above shows how to represent the cursor following the **`'\n'`** on the second line.

Part (b) Write the function **`MoveCursorDown`**:

```
void MoveCursorDown(bufferADT buffer);
```

This function should have the following behavior:

- If there is no line below the current line, nothing happens, (it should **not** raise an error message).
- If the **`cursor`** currently follows the **nth** character of the **`curLine`** and **there is** an nth character on the next line, it should end up following that nth character on the next line.
- If the **`cursor`** currently follows the **nth** character of the **`curLine`** but **there is no** nth character on the next line, it should end up following the last character on the next line.
- All related pointers should be updated accordingly.

For both (a) and (b) you are extending the editor buffer interface and are thus writing **implementation-side** code. In your implementation, you may, of course, make use of any of the functions defined in the standard **`buffer.h`** interface as printed in the text on pp. 377-378.

Write your answer here:

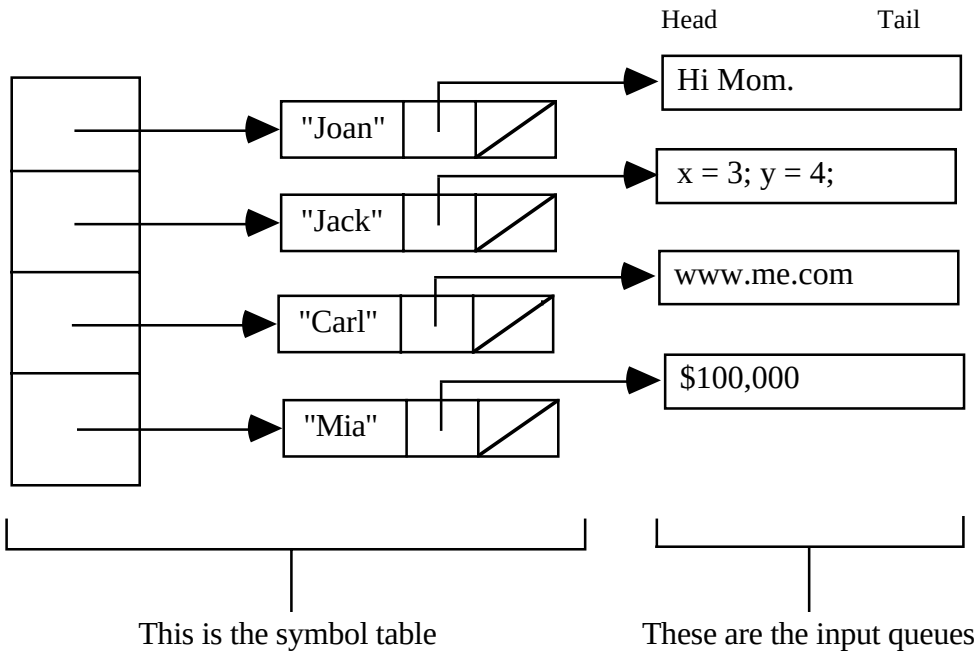
```
#include "buffer.h"

void InsertCharacter(bufferADT buffer, char ch)
{









void MoveCursorDown(bufferADT buffer)
{
```

ADT's Client Side - Character Buffering

This problem will explore some of the issues involved in dealing with data flow in a multi-user computer system. To keep up with their input from all the users, the system might maintain a queue of characters for each user. Characters typed are placed in the appropriate queue, and when the system allocates processing time to a given user, it retrieves the characters from that user's queue.

To simulate some aspects of such a system, we must first decide on a data structure. We know we will be using queues, with one queue for every user. But how can we associate users and queues? That is a perfect job for a symbol table. Each user will have a user name that is a string, and these names will be the keys for the symbol table entries. The values associated with the keys will be the queues. That will work fine since our symbol tables use `void *` values, and a `queueADT` is a pointer type. You may assume that the queues stores characters, i.e., that **`queueElementT`** is typedef'ed to be a **`char`**.

The following diagram shows what we have in mind. For simplicity we show each user name hashing to a different bucket, but that is not a requirement.



As you can see, each entry in the table has a name as its key and a queue as its value.

What you are going to do for this problem is write two functions that could be used in such a system. These are the functions that place data in the queues and extract data when needed. Here are the details:

Part (a) Write a function with the following prototype:

```
void AddData(symtabADT table, string name, string data);
```

name is a user name and **data** is a string that is to be added to the queue for the user. If the specified user name is not already in the table, the function adds a new entry to the table with **name** as the key and **data** as the value. If the name already exists in the table, the function adds the data to the appropriate queue. For example, if the structure currently held the data shown in the diagram above, then the call

```
AddData(theTable, "Jack", "z=9;");
```

would change Jack's data queue to be `"x=3;y=4;z=9;"`. You may assume that their first argument is a symbol table set up as shown above, and that **data** is a non-empty string. As indicated, you may not assume the name **name** is already in the table.

Part (b) Write a function with the following prototype:

```
string ExtractData(symtabADT table, string name, int n);
```

This function returns a string that holds the first **n** characters from the queue for user **name**, and removes those characters from that queue. The function should allocate memory for the string it returns and copy the characters into that memory. If fewer than **n** characters are present in the queue, the function extracts all the characters present. If the queue is actually empty, then the return value is an empty string. In the diagram above, the call

```
userData = ExtractData(theTable, "Joan", 2);
```

would set **userData** to "`Hi`" and change Joan's queue to be "` Mom`".

Be sure you answer both parts of the question.