CSE 373 -- Spring 1999 -- Assignment #2

(*due Friday, April 16*)

1. **Assume that we want to use an integer List ADT to store a *sorted* list of integers -- one in which the values are stored in numerical order from lowest to highest (recall that by nature, the List ADT does not maintain its elements in any particular order other than that which was specified by the user). Thus, we will keep the list sorted ourselves by performing all inserts with a special-purpose function called `InsertSorted(List L,ElementType newval);`. This function inserts `newval` into the appropriate place in the list `L` (Note that unlike the normal `Insert()` routine, no Position parameter is passed in. This is because the routine itself will determine the correct position for `newval` so that the list remains sorted).**

   (a) Assume that you are given an implementation of an integer List ADT, but not its source code. Implement `InsertSorted()` for this ADT using only the standard List operations (`Insert()`, `Delete()`, `First()`, `Next()`, `IsEmpty`, `IsLast()`, `Retrieve()`, etc). Assume that `Insert()` is implemented so that the new value is inserted *before* the position that was passed in as its argument.

   (b) Next, assume that you've been given the source code for the integer List ADT in part (a) and find that it is an array-based implementation. The representation of the list is a pointer to a structure like we used in lecture (or a class with the equivalent structure, for you C++ users):

```
              C                                        C++
   --------------------------                --------------------------
   typedef struct _ListInfo {                class List {
     int val[MAX_ARR_LEN];                     private:
     int length;                                 int val[MAX_ARR_LEN];
   } ListInfo;                                   int length;
                                               ...
   typedef ListInfo *List;                   }
```

   Extend the ADT by implementing the `InsertSorted()` operation, referring directly to the List's structure (*i.e.*, **do not** use the standard list operations).

   (c) Assume once again that you've been given the ADT's source code as in part (b). However, rather than an array-based implementation, you find it to be a linked list implementation in which the nodes are singly linked **and a dummy header node is used.**

```
              C                                        C++
   --------------------------                --------------------------
   typedef struct _Node {                    /* same Node def as C */
     int val;                                class List {
     struct _Node *next;                       private:
   } Node;                                       Node *header;
                                               ...
   typedef Node *List;                       }
```

   Once again, extend the ADT by implementing the `InsertSorted()` operation, referring directly to the List's structure.

(d) Same as parts (b) and (c), but this time it's a doubly-linked list, **with a dummy head and tail node**:

C

```
 --------------------------
typedef struct _Node {
  int val;
  struct _Node *prev;
  struct _Node *next;
} Node;

typedef Node *List;
```

C++

```
-------------------------
/* same Node def as C */

class List {
  private:
    Node *header;
  ...
}
```

(e) Give the asymptotic running times of your algorithms for part (a), (b), (c), and (d).

2. **You are the chief marketing officer at *MicroList*, a company that implements List ADTs for use by consumers. A lowly CSE 373 instructor comes to you, explaining that he's thinking about purchasing some implementations of a List ADT to help him with his class organization. For each application, give a sales pitch for a List implementation that would be well-suited for his requirements, and give your reasons for the suggestion:**

   (a) "I want to maintain my class list as a List. Once the first week of lectures is over, I expect that the class will be fairly fixed -- not many drops or adds. I'd like to store a bunch of information for each student, including their grades, their answers to my first-day survey, and their e-mail address. I expect that once the quarter is underway, I'll use the list to look up people that I have met and received e-mail from to check their hobbies, favorite album, or grades. What do you recommend?"

   (b) "I find that this class is taking up all my time, and I'm finding it difficult to keep track of all the things that I have to do from day to day (prepare lectures, make slides, make handouts, put the handouts on the web, answer questions, attend office hourse, etc). Generally, I scribble notes to myself in my notebook, but it sure would be nice to have a computerized 'todo list' of all the things I have to do in the next day or so. That way, I could add something to the list to remind myself to do it, and delete it once I've completed the task. Generally there would only be a few tasks on the list at any given time, but they would tend to be deleted from the list quickly and replaced by new tasks. What do you recommend?"

   (c) "I appreciate your recommendations. Here's one last question: I've heard about a 'free' implementation of lists that's available on the web called 'C arrays' (apparently it's put out by the same group that makes the gcc and g++ compilers for UNIX). Why should I spend all this money to buy a List ADT from *Microlist* when I could just use a C array for free?"

3. **Integer variables in C or C++ are of fixed size. Therefore, they can only store values of a certain size before they overflow and wrap around to the negative numbers (for example, a `"short"` may only be able to store values from -32767 to 32768).**

   (a) Propose a List-based implementation of integers that can store values of arbitrarily large size.

   (b) Would you use an array-based or linked list-based List implementation for your integers? Why?

(c) Show how the integers 98765432123456789 and 32768 would look like in your implementation (using pictures similar to the ones I use in lecture).

(d) Implement an `Add()` operation for your integers using the standard List operations. (`Add()` should take in two Lists representing integers and return a new List representing the integer sum).

4. **Programming Assignment (due Friday, April 23)**

   This program is designed to give you experience using lists, stacks, and queues by implementing a realistic program. Note that the book provides most of the code implementing the List, Stack, and Queue ADTs, and that this code is available on the web. Thus, this assignment does not test your ability to implement the ADTs, so much as your ability to instantiate them for a specific element type and use them effectively in a realistic application.

   HTML documents are text files that contain formatting *tags* which tell web browsers how the file should be drawn on the screen. For example, this assignment was written in HTML to make certain words bold, to number and indent the list of questions, to request a fixed-size font for pieces of code, etc. (To see what the HTML looks like, go to this assignment on the course web and select "View...Source" from your web browser).

   In this assignment, you will implement a program that checks a set of HTML files for simple errors using a list, a stack, and a queue.

   All formatting in HTML is done by surrounding the text with "tags" that indicate things like **bold text**, *italics text*, `fixed-sized text`, etc. These tags are indicated by the less-than (`<`) and greater-than (`>`) symbols. For example, to start using bold text, you would use the tag: `<b>`. Most tags come in "open" and "close" varieties to indicate when that property should start and stop. The "close" version of a tag is similar to the open version, but begins with a slash symbol (`/`). For example, to stop using a bold font, you would use `</b>`.

   Most HTML files also contain "anchor" tags. These indicate that the text that falls between the open and close anchor tags is to be used as a hyperlink to another file. For example, the following HTML specifies that when a user clicks on the word "here", the web browser should display the file named "page2.html":

   ```
   click <a href = "page2.html">here</a> to go to the next page
   ```

   This example also demonstrates that an opening tag sometimes has extra information in it like the "href" and filename used for an anchor. The closing tags for these more complex tags are usually just the closing tag with no additional information (*e.g.*, note that the tag does not contain "href" and a filename).

   A common problem that occurs when writing HTML is failing to nest your tags perfectly. For example, if you open a bold tag and then open an italics tag, it is good style to close the italics tag before closing the bold tag. Here's some HTML in good style:

   ```
   <b>text<i>more text</i>even more</b>
   ```

   A poorly nested version would look like the following:

```
<b>text<i>more text</b>even more</i>
```

Because this sort of mistake is quite easy to make, web browsers are very forgiving. Rather than generating an error or warning, they do their best to draw the text as they think you would want it to look. The problem with this, however, is that if you use **Netscape** to view your document while you’re creating it, it may interpret your poorly nested tags one way, whereas **Internet Explorer** (or even a different version of **Netscape**) may interpret them in a completely different way. Thus, even though you *think* you know what your document looks like, other people might be seeing something completely different. For this reason, a program that scans your web documents, checking that your HTML tags are well-nested might be a useful program to have. Note that this is a very similar problem to the "Balancing Symbols" stack application in the textbook.

For this assignment, you will write such a program. In a sentence, the program will check to see whether a document and all the local HTML documents that it links to (and that they link to, etc.) are perfectly nested. **Note: If you are the type of person who enjoys the challenge of solving this type of assignment for yourself, you may want to stop reading here and think about how you would do this without all my hints.**

At a high level, your program should work as follows:

- It should start by asking the user for the filename of the root HTML document to check.
- It will open this file and begin reading it one character at a time. All characters will be ignored until a ’<’ character is encountered -- at this point, all of the characters that follow should be read into a string or array of characters until the matching ’>’ character is encountered.
- The program should then process this HTML tag according to its type:
  - *singleton* -- a tag which does not have open/close varieties. Examples are `<br>` and `<li>`. These tags should be ignored.
  - *opener* -- a tag that opens a new scope. Examples include `<b>` and `<i>`. These tags should be pushed onto a *Stack*.
  - *closer* -- a tag that closes a scope. Examples include `</b>` and `</i>`. When these tags are encountered, the top of the stack should be popped to see if it matches the last opener that was encountered. If it does, the HTML is well-nested and the program continues. Otherwise, it prints a helpful error message and exits.
- In addition, the HTML file might link to other files using anchor tags. The program should check these files as well, so whenever an anchor tag is encountered, the filename should be enqueued onto a worklist *Queue* so that the program can come back to it when it has finished with the current file.
- When the end of the file is reached, the program should check to see whether the stack is empty. If it is, then the file was perfectly nested and the program should report this. Otherwise, some tags were opened but never closed and you should print out a helpful error message reporting this.
- Once the program has finished processing a file, it should dequeue a filename from the worklist queue and process it in the same way as the original file. If opening the file results in a failure, you should assume that the file is non-local (for example a link to a file on another web site), skip it, and go on to the next filename in the worklist queue.
- One problem that happens commonly on the web is that one document will link to another which will link to another which will eventually link back to the first. This could cause your

program to go into an infinite loop, checking the same documents over and over again. Therefore, you'll want to keep a *List* of all the files that you've checked to be sure that you never check the same one twice.

- When your worklist queue is empty, your program has processed all the files to which your root file led, and the program should exit.

The above description gives you the basic algorithm that your program should follow. Details that you will need to figure out on your own include: (i) what element types you will use in your list, stack, and queue; (ii) how you will categorize a tag as being a singleton or an opener/closer tag; (iii) what types of error messages you will provide and how you will implement them (e.g., A good error message might say something like "ERROR: <b> opened on line 35, but closed by </i> on line 42 of myfile.html.")

Check the course web for a complete list of HTML tags that your program will be expected to recognize, example "webs" that you can use to develop your program, and further hints on how to approach this program in a modular, straightforward manner.