Assignment #2

Due date: Friday, 29 January 2016, 5:59 pm

- *For all programming questions below, write your solutions in the dialect of C++ used in class* e.g., *don't use extra libraries without checking with us first.*
- *Store each function in its own file named* `a2p1.cc`*,* `a2p2q1.cc`*,* `a2p2q2.cc`*, ...*
- *You may define your own* `main` *functions, but we will use our own* `main` *to test your code.*
- *Note that the some questions are harder than others, and the harder ones will be worth more marks.*

Part I

We're going to re-organize the logic of Assignment #1 to make it more like a real command-line Unix program. Write a program that reads in a positive integer `N`, plus a character string `textFileName`, followed by a sequence of commands using token-oriented input from `cin`. `N` is the line length, as before. `textFileName` should be the name of a file in the current directory in which you are working that contains the text you are to process.

After you read in `N` and `textFileName` from `cin`, you should then process the text as in Q3–5 of Assignment #1 and store it in an appropriate data structure. Note that you will need to use *file-based* token-oriented input (as discussed in class) to read in the text.

Then you will read in a sequence of commands (which are string tokens) from `cin`. You should continue to process commands until either EOF is detected in `cin` or the `q` (for "quit") command is encountered. The commands are as follows:

`rr`	Switch justification mode to ragged right (the default)
`rl`	Switch justification mode to ragged left
`c`	Switch justification mode to centred
`j`	Switch justification mode to right and left justified
`f`	Change the print direction mode to forward (the default)
`r`	Change the print direction mode to reverse
`p`	Print all of the lines (using the current justification mode and print direction)
`k <k>`	Print the k^{th} line of text you have built up (if there is one; if not, print nothing)
`s <s>`	Print only lines that contain the specified string `<s>` anywhere
`q`	Quit gracefully

Justification mode `j` should cause *all* lines to printed as right and left justified, *including the final line.* Commands `p`, `k`, and `s` should print their line(s) according to the current justification scheme (which is ragged right by default) and print direction (forward, by default). In the above, `<k>` represents an actual integer and `<s>` represents an arbitrary character string; so if you notice that the command is `k` (or `s`) then read the next token into an integer (or string) variable. Valid values of `<k>` are between 0 and $M - 1$, where M is the total number of lines you end up with. If the print direction mode is reverse, then `k 0` means print the *last* line.

Note that in a single session, a user may print the lines, then change the print direction and justification mode and print them again, then print only the k^{th} line, then change direction and justification mode again, and print the results one more time. This means that you have to put some thought into just when you perform the justification, and what you store where. If the specified file name can't be found, print this error message to `cerr` and quit:

```
Error, can't open specified text file.
```

If `command` is any other string, print this message to `cerr` and quit:

```
Error, command is illegal.
```

Part II

For each of the following questions, use this definition for `Node` (as presented in class):

```
struct Node{
    string val;
    Node* next;
};
```

1. Write a C++ function called `makeList` that reads in a list of strings from `cin` (stopping when there's no more input) and returns a pointer to a list of those strings in the order in which they were read in. Make sure the `next` pointer of the last element is `NULL`. The signature of the function should look like this:
   ```
   Node* makeList () { /* implement me! */ }
   ```

2. Write a C++ function called `printList` that takes an existing list (i.e., a pointer to `Node`), and prints the elements in order, one string per line. You may assume that the last element's `next` pointer is `NULL`. The signature of the function should look like this:
   ```
   void printList (Node* p) { /* implement me! */ }
   ```

3. Write a C++ function called `printPairInOrder` that takes two `Node` pointers and prints the `val` fields in alphabetical order (use the normal "<" operator to compare string values), with each string on its own line. (The values of the `next` pointers aren't relevant for this function.) You should `assert` at the beginning of the function that neither pointer is `NULL`. The signature of the function should look like this:
   ```
   void printPairInOrder (Node* p1, Node* p2) { /* implement me! */ }
   ```

4. Write a C++ function called `sortPair` that takes two `Node` pointers and returns a pointer to a list of the two elements sorted alphabetically. You should `assert` at the beginning of the function that neither pointer is `NULL`. Make sure the `next` pointer of the last element is `NULL`. (The *incoming* values of the `next` pointers aren't relevant for this function.) The signature of the function should look like this:
   ```
   Node* sortPair (Node* p1, Node* p2) { /* implement me! */ }
   ```

5. Write a C++ function called `makePairList` that takes two strings and returns a pointer to an alphabetically sorted list of the two strings. Make sure the `next` pointer of the last element is `NULL`. The signature of the function should look like this:
   ```
   Node* makePairList (string s1, string s2) { /* implement me! */ }
   ```

6. Write a C++ function called `append` that takes two pointers that each point to the first element of a list of `string`s, call them `p1` and `p2`, and returns a pointer to the list that results from appending `p2`'s list onto the end of `p1`'s list. For example, if the first list consists (in order) of the strings `alpha baker charlie` and the second list consists of `delta echo`, then the function should return a pointer to the list whose elements are (in order) `alpha baker charlie delta echo`. You should not create any new `Node`s, just reset pointers appropriately. The signature of the function should look like this:
   ```
   Node* append (Node* p1, Node* p2) { /* implement me! */ }
   ```

7. Write a C++ function called `printReverseRecursive` that reads in a list of strings from `cin` (using token-oriented input, stopping when there's no more input), and prints the list in the reverse order of reading, one token per line. Here's the catch: you must use only recursion to do this. You may *not* use the `Node` class here, nor may you use a `vector`, nor any other cool container you or someone else wrote. Just simple, plain old recursion, OK?
   ```
   void printReverseRecursive () { /* implement me! */ }
   ```