COMS 3136 Essential Data Structures in C/C++, Lab #2 (150 points total)

Please read this assignment carefully and follow the instructions EXACTLY.

Submission

Please refer to the lab retrieval and submission instruction, which outlines the only way to submit your lab assignments. Please do not email me your code.

If a lab assignment consists of multiple parts, your code for each part must be in a subdirectory (named "part1", "part2", etc.)

Please make sure that your submission satisfies the requirements for the following items:

- README.txt
- Makefile
- Valgrind

The requirements remain the same as lab 1.

Part 1: Storing argv strings in a linked list (50 points)

What I gave you:

    list1.h - DO NOT MODIFY IT

What you need to write:

    list1.c - implements the functions declared in list1.h
    test1.c - your main() goes here
    Makefile - the executable is named test1

Example input & output:

```
$ ./test1 one two three "" four1234
->[5:four1]->[0:]->[5:three]->[3:two]->[3:one]
```

The test1 program will go through the command line arguments (i.e., the argv array), prepend each string and its length (but only up to 5 characters) to a linked list, print the list in the format shown in the example above, and finally deallocate all nodes so that it will run valgrind-clean.

Note the use of double-quotes to pass an empty string as a command line argument. You can also use double-quotes to pass a string containing spaces.

The list1.h header file declares struct Node, prepend(), strcpy_upto(), print(), deallocate_list(); and it specifies the details of the functions in the comments. You CANNOT modify list1.h. Your job is to implement the 4 functions in list1.c, and main() in test1.c.

Don’t forget to include the valgrind output in your README.txt. You are striving for the following summary line from valgrind:

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Part 2: Sorting and reversing a linked list (40 points)

Write test2, which constructs the same list as test1, and then goes on to sort the list and reverse the list.

Example input & output:

```
$ ./test2 one two three "" four1234
prepended: ->[5:four1]->[0:]->[5:three]->[3:two]->[3:one]
   sorted: ->[0:]->[5:four1]->[3:one]->[5:three]->[3:two]
 reversed: ->[3:two]->[5:three]->[3:one]->[5:four1]->[0:]
```

The part2 directory contains the same list1.h from part1 (again, do NOT modify it.) You should copy over list1.c from part1. You will write list2.h, list2.c, and test2.c.

list2.h/c should contain the sorting function and the reversing function. Put #include "list1.h" in list2.h.

For sorting, implement selection sort on our linked list. You can use either recursive or iterative implementation.

Your selection sort function MUST take the following two parameters (in addition to any other parameter that you need to pass):

```
int (*compareData)(struct Node *, struct Node *)
void (*swapData)(struct Node *, struct Node *))
```

Your selection sort will simply call those function pointers to perform comparison and swap. They will point to the comparison and swap functions you write in test2.c.

The comparison will be a lexicographical comparison (i.e., what strcmp() does, which you can call). The swap function will swap the len and the data array of the two given Node structs, but not the next pointers.

You can also use either recursive or iterative algorithm for reversing a linked list. I give you an outline for a recursive algorithm:

- Remember the 2nd node.
- Recursively call reverse for the sub-list starting from the 2nd node.
- Append what used to be the head node to what used to be the second node. (That old head node will be the new last node.)
- Return the new head node (i.e., the list you got back from your recursive call.)

Don’t forget to check the three rules of writing a recursive function.

Part 3: Permuting a linked list (30 points)

Write test3. Here is an example run:

```
$ ./test3 abc def abc xyz
2 1 0 3 *
original: ->[3:abc]->[3:def]->[3:abc]->[3:xyz]
creating a new list of 4 nodes: 2 1 0 3
permuted: ->[3:abc]->[3:def]->[3:abc]->[3:xyz]
SAME
```

First, test3 reads a series of integers separated by whitespace until it encounters a non-digit character (I typed in '*' here.) Then it constructs two linked list: one that has the command line arguments in the order passed in (you can prepend and then reverse), and another list that is a permutation of the first list in the order specified as the series of integers (which we call a permutation recipe.)

In the example above, the permuted list has the string from the 2nd node in the original list (the head being 0th), followed by the strings contained in the 1st, 0th, 3rd nodes in the original list. Since the 0th and 2nd nodes contain the same string, the permuted list is the same as the original one, so test3 prints "SAME" at the end.

Here is another example:

```
$ ./test3 abc def abc xyz
3 3 3 0 0 0 *
original: ->[3:abc]->[3:def]->[3:abc]->[3:xyz]
creating a new list of 6 nodes: 3 3 3 0 0 0
permuted: ->[3:xyz]->[3:xyz]->[3:xyz]->[3:abc]->[3:abc]->[3:abc]
DIFFERENT
```

Note that we use the term permutation loosely, i.e., we allow repetitions and omissions of any node.

You should copy over list1.h, list1.c, list2.h and list2.c from part1 and part2 directories as needed. You will write list3.c, list3.h, and test3.c.

You must implement the following two functions (plus any other helper function you need) in list3.c:

```
/*
 * Create a new linked list which is a rearrangement of the data
 * contained in the original list.  The recipe parameter specifies
 * the rearrangement.  The size parameter contains the size of the
 * recipe array.
 *
 * For example:
 *      list:   ->[a]->[b]->[c]
 *      recipe: { 2, 2, 0, 1, 0, 1 }
 *      size:   6
 *
 * will return:
 *              ->[c]->[c]->[a]->[b]->[a]->[b]
```

```
     *
     * The behavior when the recipe contains an invalid position is
     * undefined.
     *
     * The original list is not modified.  The recipe array is assumed to
     * point to an integer array of 'size' elements.
     */
    struct Node *createPermutedLinkedList(
            struct Node *list,
            int *recipe,
            int size);

    /*
     * Returns 1 if list1 and list2 contain the same number of nodes
     * and each pair of nodes at the same position contain the same
     * string.  Returns 0 otherwise.
     */
    int sameContents(struct Node *list1, struct Node *list2);
```

The test3 program will use those functions to implement the behavior shown in the examples above. Please produce your output in the same exact format as the examples.

You can limit the size of the permutation recipe to 100 integers (which means you can use "int a[100]".) And you can use scanf() in a loop to read your numbers. The return value of scanf() is helpful here. See man 3 scanf for detail.

As always, don't forget to make sure test3 runs valgrind-clean.

Part 4: Sorting by length (30 points)

In part 4, we modify struct Node as follows:

```
    struct Node {
        struct Node *next;
        char *data;
    };
```

The part4 directory contains a new list1.h which has the new definition for struct Node. The behavior of prepend() function changes in part 4 as follows:

```
    /*
     * Given a string s, create a node that stores a new string:
     *
     *    min(12,strlen(s)):s
     *
     * For example,
     *
     *    for s == "hello", the node will store "5:hello".
     *
     *    for s == "aReallyLongStringThatsLongerThan12Chars",
     *          "12:aReallyLongStringThatsLongerThan12Chars" will be stored.
     *
     * The node is added to the front of the given list, and the resulting
```

```
     * list is returned.  NULL is returned if malloc fails.
     *
     * Behavior is undefined if s == NULL.
     */
    struct Node *prepend(struct Node *list, char *s);
```

You must implement the new prepend() in list1.c. The print() and deallocate_list() functions must also change slightly for the new struct Node.

Write test4 that sorts a linked list of command line arguments. The test4 program is similar to test2 that you wrote in part 2, except that test4 sorts the list according to the number prefixed to each string, not the string itself. Here is an example run:

```
$ ./test4 cccc zz abcdef0123456789 aaabbbcccddd
prepended: ->[12:aaabbbcccddd]->[12:abcdef0123456789]->[2:zz]->[4:cccc]
   sorted: ->[2:zz]->[4:cccc]->[12:aaabbbcccddd]->[12:abcdef0123456789]
```

test4 first builds a linked list by calling prepend() with the command line arguments, and then it sorts the list according to the prefixed length, ignoring the actual strings.

The main() function in test4.c can be identical to the one in test2.c (except that you don't have to reverse the list after sorting in test4.)

You should be able to use the same selection sort function from part 2. In fact, if you did part 2 correctly, you should be able to copy list2.h and list2.c from part 2 and use it with no modification. All you have to do is to rewrite the comparison and the swap functions that you pass to the selection sort function.

The comparison function will look at the two prefixed numbers of the two nodes' strings, and return -1, 0, or 1, according as the first number is less than, equal to, or greater than the second number.

A few library functions may come in handy for implementing this part: sprintf() and atoi().

Don't forget that test4 must run valgrind-clean.

--

Good luck!