Programming Assignment #3: Linked List Strings

COP 3502, Summer 2013

Due: Sunday, July 7, 11:59 PM on Webcourses2@UCF

Abstract

In this programming assignment, you will use linked lists to represent strings. You will implement functions that manipulate these linked lists to transmute the strings they represent. In doing so, you will master the craft of linked list manipulation!

By completing this assignment, you will also gain experience with file I/O in C and reinforce your understanding of how to process command line arguments.

Attachments

ListString.h, input{01-03}.txt, output{01-03}.txt

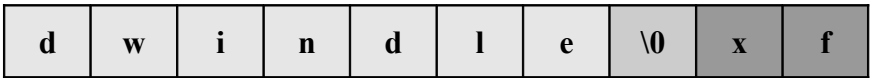
Deliverables

ListString.c

1. Overview

1.1. Array Representation of Strings in C

We have seen that strings in C are simply char arrays that use the null terminator (the character '`\0`') to mark the end of a string. For example, the word "dwindle" is represented as follows:

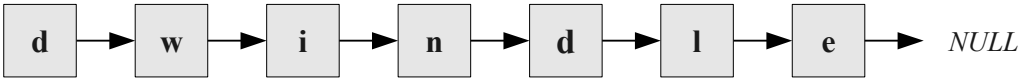


Notice the unused portion of the array may contain garbage data.

1.2. A Linked List Representation of Strings

In this assignment, we will use linked lists to represent strings. Each node will contain a single character of the string. The null terminator ('`\0`') will not be afforded its own node in the linked list. Instead, we will know that we have reached the end of a string when we encounter a `NULL` pointer.

For example, the word "dwindle" is represented as follows:



1.3. String Transformations

You will implement two key string transformations: a *replace character* function that replaces all instances of a particular character with a specified string, and a *reverse string* function that reverses a linked list. These are described in detail in Section 3, "Function Requirements," on page 4.

1.4. Linked List Node Struct (ListString.h)

You must use the linked list node struct we have specified in `ListString.h` without any modifications. You should `#include` the header file from `ListString.c` like so:

```
#include "ListString.h"
```

The node struct is defined in `ListString.h` as follows:

```
typedef struct node
{
   char data;
   struct node *next;
} node;
```

2. Input Files

2.1. Command Line Arguments

When we run your program, we will use a command line argument to specify the name of the input file your program should read. For example:

```
./a.out input1.txt
```

For more information on processing command line arguments with `argc` and `argv`, see the instructions from the Program #2 PDF. You can assume that we will always specify a valid input file when we run your program.

2.2. Input File Format

The input file will begin with a single string that will contain at least 1 character and no more than 1023 characters. Read it in and convert it to a linked list. That will become your working string, and you will manipulate it according to the remaining commands in the input file.

Each of the remaining lines in the file will correspond to one of the following string manipulation commands:

Command	Description
`@ key str`	`key` is a single character. `str` is a string. In your working string, replace all instances of `key` with `str`. **Note:** `key` and `str` are guaranteed to contain alphanumeric characters only (A-Z, a-z, and 0-9). `str` can range from 1 to 1023 characters (inclusively).
`- key`	`key` is a single character. Delete all instances of `key` (if any) from your working string.
`~`	Reverse the working string.
`!`	Print the working string.

For more concrete examples of how these commands work, see the attached input/output files and check out the function descriptions below in Section 3, “Function Requirements” (page 4).

2.3. Sample Input/Output Files

We have attached a few sample input and output files so you can check that your program is working as intended. Be sure to use `diff` on Eustis to make sure your output matches ours exactly. We also encourage you to develop your own test cases; ours are by no means comprehensive.

3. Function Requirements

You have a lot of leeway with how to approach this assignment. There are only four required functions, and there are lots of different ways to implement them. How you structure the rest of your program is up to you.

```
node *stringToList(char *str);
```

> **Description:** Convert the string `str` to a linked list. If `str` is `NULL` or an empty string (“”), simply return `NULL`.
>
> **Returns:** The head of the new linked list.

```
node *replaceChar(node *head, char key, char *str);
```

> **Description:** Takes a linked list (`head`) and replaces all instances of a certain character (`key`) with the specified string (`str`). If `str` is `NULL` or the empty string (“”), this function simply acts to delete all instances of `key` from the linked list. If `key` does not occur anywhere in the linked list, the list remains unchanged.
>
> **Returns:** The head of the modified linked list.

```
node *reverseList(node *head);
```

> **Description:** Reverse the linked list.
>
> **Returns:** The head of the reversed list.

```
void printList(node *head);
```

> **Description:** Print the string stored in the linked list, followed by a newline character, ‘`\n`’. If `head` is `NULL`, simply print “`(empty string)`” (without the quotes), follow by a newline character, ‘`\n`’.
>
> **Returns:** Nothing. It’s a `void` function.

4. Compilation and Testing (Linux/Mac Command Line)

To compile at the command line:

```
gcc ListString.c
```

By default, this will produce an executable file called `a.out` that you can run by typing, e.g.:

```
./a.out input01.txt
```

If you want to name the executable something else, use:

```
gcc ListString.c -o ListString.exe
```

...and then run the program using:

```
./ListString.exe input01.txt
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following:

```
./ListString.exe input01.txt > whatever.txt
```

This will create a file called `whatever.txt` that contains the output from your program.

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt output01.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt output01.txt
3c3
< riddlee
---
> riddle
seansz@eustis:~$ _
```

5. Grading Criteria and Submission

5.1. Deliverables

Submit a single source file, named `ListString.c`, through Webcourses2@UCF. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work. Don't forget to `#include "ListString.h"` in your source code. We will compile your program using:

```
gcc ListString.c
```

Be sure to include your name and PID as a comment at the top of your source file.

5.2. Additional Restrictions: Use Linked Lists; Do Not Use Global Variables

You must use linked lists to receive credit for this assignment. Also, please do not use global variables in this program. Doing so may result in a huge loss of points.

5.3. Grading

The expected scoring breakdown for this programming assignment is:

45%	Correct Output for Test Cases (Autograder)
45%	Unit Testing (Autograder)
10%	Comments and Whitespace

Your program must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based primarily on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization, punctuation, or whitespace errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program thoroughly.

For this program, we will also be unit testing your code. That means we will devise tests that determine not only whether your program's overall output is correct, but also whether each function does exactly what it is required to do. So, for example, if your program produces correct output but your `stringToList()` function is simply a skeleton that returns `NULL` no matter what parameters you pass to it, your program will fail the unit tests.

5.4. Closing Remarks

Start early. Work hard. Ask questions. Good luck!