Lab 10: Doubly Linked Lists

CSE/IT 113

Introduction

This lab is a continuation of last week's lab on linked lists, but this time you will implement a doubly linked list using a sentinel. You will test your doubly linked list by creating hailstone sequences and performing various manipulations of the hailstone sequence.

Requirements

Implement a doubly linked list of integers. Your program must have the ability to store an arbitrary number of integers—limited only by how much free memory is available for your computer.

The basic structure of the sentinel and node you will use is:

```c
/* struct info_t is a "sentinel" which keeps track of information about the list */
struct info_t {

        unsigned int count; /* keep track of the number of nodes in the list */

        struct node_t *head; /* a pointer to the head of the list */
        struct node_t *tail; /* a pointer to the tail of the list */
};

/* define the node */
struct node_t {

        unsigned int n;                  /* data */

        struct node_t *next;    /* point to the next node in the list */
        struct node_t *prev;    /* point to the previous node in the list  */
};
```

In implementing your doubly linked list, you must meet (at a minimum) the following requirements:

- When your program ends, you must remember to correctly free all dynamic memory.
- You must implement the doubly linked list correctly: it cannot have a builtin limit on the number of elements it can hold. System limits, such as available memory, will of course cap the length of your linked list.
- You must use a sentinel node to keep track of the head and tail pointer locations and keeps track of the number of nodes in the list.
- All lists created in the lab are doubly linked lists that keep track of both the head and tail of the list.
- You must implement the following **functions**, which means passing your *head* and/or *tail* pointer around or your *info* structure as parameters:

- Create and initialize an info structure (`create_sentinel`)
- Create and initialize a new node (`create_node`)
- Add an element to a list at the head of the list (`insert_head`)
- Add an element to the tail of the list. Use a tail pointer (not traversal) to keep track of the tail. (`insert_tail`)
- Add an element to the "middle" of the list based on a comparison. You will search for a node in the list whose data is greater or equal to the data you want to insert. Once you find it, you will insert the node *before* the node in the list.
- Write a helper function that `insert_middle` uses to perform the comparison (`compare`). The comparison function compares two integers a and b. The function returns -1 if $a < b$; returns 0 if $a == b$; and returns 1 if $a > b$.
- Create a sorted list of the hailstone sequence. (`create_sorted_list`).
- Create a list containing only the even numbers of the hailstone sequence (`create_even_list`).
- Create a list containing only the odd numbers of the hailstone sequence (`create_odd_list`).
- Print a node's data, an integer in this case. (`print_node`)
- Print the list as a space separated list of numbers. The last element in the list is terminated with a new line. (`print_list`). `print_list` calls `print_node`
- Print the list in reverse as a space separated list of numbers. The last element in the list is terminated with a new line. (`print_reverse_list`). `print_reverse_list` calls `print_node`
- Delete all the nodes in the list and correctly free the memory occupied by those nodes. (`delete_list`)

Collatz conjecture and Hailstone sequences

The Hailstone sequence is generated in the following way:

- take any natural number (positive integer) n
- if n is even, you divide by 2 to get $n/2$.
- if n is odd, multiply n by 3 and add 1 to obtain 3n + 1.
- repeat until you reach the number 1

The Collatz Conjecture states that no matter what number you start with, you always end up at 1 eventually. This is an open problem in mathematics. See `http://en.wikipedia.org/wiki/Collatz_conjecture` for more info.

For example, if $n = 6$ the sequence generated is: 6, 3, 10, 5, 16, 8, 4, 2, 1. The stopping time is the number of steps it took the hailstone sequence to get to 1. For 6, the stopping time is 9.

For $n = 3$, you generate the sequence 3, 10, 5, 16, 8, 4, 2, 1. The stopping time is 8.

Menu

Your menu interface must have the following options:

1. Enter a number, the hailstone sequence is calculated
2. Print the hailstone sequence
3. Print the hailstone sequence in reverse order

4. Print the hailstone sequence in sorted order (ascending)
5. Print the hailstone sequence in reverse sorted order (descending).
6. Print the stopping time (how many items in the list)
7. Find the max element of the hailstone sequence
8. Create a sequence of the even numbers in the hailstone sequence
9. Create a sequence of odd numbers in the hailstone sequence.
10. Print all even elements of the hailstone sequence
11. Print all odd elements of the hailstone sequence
12. Find the ratio of even to odd numbers in the hailstone sequence.
13. End program

Enter a number, the hailstone sequence is calculated

The user enters a number. The program first checks to see if a hailstone sequence and any of its associated lists (sorted, even, odd) already exist. If they exist and the number is not the same as the already generated hailstone sequence (i.e the user entered 100 twice in a row), then it deletes all existing lists before it generates the new hailstone sequence. If the user entered the same number twice, don't do anything as the hailstone sequence already exists. Otherwise as no hailstone sequence exists generate a hailstone sequence. All hailstone sequences are doubly linked lists.

Print the hailstone sequence

Prints the hailstone sequence from the head to the tail of the list.

Print the hailstone sequence in reverse order

Prints the hailstone sequence from the tail to the head of the list.

Print the numbers in sorted order

It first checks to see if a sorted list already exists. If the sorted list exists, print the sorted list. If the sorted list does not exist then it takes the hailstone sequence and creates a second list that is a sorted version of the hailstone sequence, using the function `insert_middle` and then prints the list.

Print the hailstone sequence in reverse sorted order (descending)

It first checks to see if a sorted list already exists. If the sorted list exists, print the sorted list in reverse. If the sorted list does not exist then it takes the hailstone sequence and creates a second list that is a sorted version of the hailstone sequence, using the function `insert_middle` and then prints the list in reverse.

Print the stopping time (how many items in the list)

Prints the stopping time of the hailstone sequence using the count in the info structure. Include a label in your output. Write a function to print the stopping time.

Find the max element of the hailstone sequence

Finds and prints the max element in the hailstone sequence. Include a label in your output. This is similar to what you did to find the max elements in an array, except this time you have to walk through a doubly linked list. Write a function that finds the max element in the list.

Create a sequence of even numbers in the hailstone sequence

Creates a new doubly linked list that just contains the even numbers. The even numbers are in the order they were generated in the hailstone sequence.

Create a sequence of odd numbers in the hailstone sequence

Creates a new doubly linked list that just contains the odd numbers. The odd numbers are in the order they were generated in the hailstone sequence.

Print all even elements of the hailstone sequence

First checks to see if the even sequence exists. If it does then prints the even list. If the even list doesn't exist then walks through the hailstone sequence printing out the even elements. Write a function that prints the evens (`print_evens`). The function needs to handle both cases.

Print all odd elements of the hailstone sequence

First checks to see if the odd sequence exists. If it does then prints the odd list. If the odd list doesn't exist then walks through the hailstone sequence printing out the even elements. Write a function that prints the odds `print_odds`. The function needs to handle both cases.

Find the ratio of even to odd numbers in the hailstone sequence

Finds and prints the ratio of even to odd numbers in the hailstone sequence. Include a label in your output. Write a function (`find_ratio`) to do this. You will have to test if the even or odd sequence exist or not. If they do, you have the count already. If both or one of the even or odd lists do not exist than you have to find the count by walking through the list. In that case, `find_ratio` should call a helper function `find_count` that returns the count of the number of odds or the number of evens in the hailstone sequence.

End program

When the user ends the program, it cleans up all lists (hailstone, even, odd, sorted) currently in memory and exits.

Testing

Initially as you develop/debug your program you should test the program with some small numbers. Some sample hailstone sequences are:

For $n = 1$, the hailstone sequence is 1

For $n = 2$, the hailstone sequence is 2 1

For $n = 3$, the hailstone sequence is 3 10 5 16 8 4 2 1

For $n = 7$, the hailstone sequence is 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

For $n = 8$, the hailstone sequence is 8 4 2 1

For $n = 9$, the hailstone sequence is 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

Once you have tested your doubly linked list functions, write the menu system.

Script File Testing

Once the menu system is in place, test with the following values of n and capture these tests in your script file.

For $n = 100,000$ perform all menu options in order, except exiting the program.

For $n = 1,000,000$ print the sequence in sorted reverse order, find the stopping time, find the max, and print the odds of the sequence. Do not exit the program.

For $n = 10,000,000$ print the hailstone in reverse order, find the stopping time, find the max, create a sequence of even numbers, print the sequence of even numbers, and exit the program.

Run and capture the output via a script file. Make sure to run the tests and capture the output with and without Valgrind. Run the tests first without Valgrind.

README

Write a standard README as in other labs, but make sure to add a section that compares and contrasts singly and doubly linked lists.

Pseudo code is required for this README. Please write pseudo code for each doubly linked list function you will write (all functions listed under the Requirements section). You do not need to write pseudo code for the menu. Make sure your pseudo code is checked off before your lab period ends.

Submission Guidelines

Tar the source code, script and README into a tarball named `cse113_firstname_lastname_lab10.tar.gz` and upload to Moodle before the start of your next lab section.