James Hinson

Design 5 Template

Part 1: Recursion

Understanding the problem:

We are given a staircase of *n* number of steps. Using 3 predefined ways of climbing the stairs (taking one small step, taking one medium step – skipping 1 small step, and taking one large step – skipping 2 small steps) we are tasked with recursively determining the number of ways that we can get to exactly the top of the staircase of *n* steps.

To make sure that you understand the problem correctly, answer the following guiding questions:

1. **(6 pts) What would `ways_to_top (1)` return? `ways_to_top (2)`? `ways_to_top (6)`? List them out, like the examples provided in the assignment.**

ways_to_top(1) would return a statement saying that there is one way to get to the top.
ways_to_top(2) would return a statement saying that there are 2 ways to get to the top.
ways_to_top(6) would return a statement saying that there are 24 ways to get to the top.

(1 pt) [List assumptions that you are planning to make for this program]

I am assuming that this program will only process integer values and won't be processing floats or other datatypes.

Program Design:

To help you break the problem down into smaller subtasks, answer the following guiding questions:

1. **(1 pt) What is the general pattern of this recursion problem? i.e., how can you break the problem down into smaller versions of the same problem?**
   The general pattern of this recursion problem is to divide the total number of steps into subgroups that you can better work with. You can divide it into subgroups that are easily divisible by 1, 2, or 3, for example.

2. **(2 pts) For a general function call of `ways_to_top(N)`, how many recursive calls should it have? And what are they?**
   For a general function call of ways_to_top(N), the function should generally have 3 recursive calls, for calls where N is greater than 2. This grows exponentially with larger numbers. The calls are ways_to_top(N-1).

3. **(1 pt) What is / are the base case(s) of `ways_to_top()`?**
   The base cases of ways_to_top(N) are any case where N is less than or equal to one.

4. **(1 pt) How many helper functions are you going to implement? What is the purpose of each of them?**
   I will need to implement two helper functions; a helper function to store the results, and a helper function to get the initial input and display the results.

(4 pts) [Create pseudocode (or flowchart) for `ways_to_top (N)` and each of your helper function]

ways_to_top:

check base case

calculate initial ways to top

store results (helper function)

recursively calculate ways to top (in the ways_to_top function), storing results each time (using the helper function)

display (helper function)

Program Testing:

To help you consider the possible test cases, answer the following guiding questions:

1. (1 pt) What is an example of good input?
   Good input is an integer value.

2. (1 pt) Are there any inputs that would cause your program to crash?
   Non-integer input – specifically strings and chars.

3. (1 pt) What if the inputs are too large for your recursive problem to run?
   If the inputs are too large for the recursive problem to run, then the program should quit or re-prompt. Otherwise, it will run until the host machine decides it's using too much memory and kills it.

(4 pts) [Create a testing table that has representative good, bad, and edge cases, and their expected outputs]

Conditions	Good	Bad	Edge
ways_to_top	An acceptable integer is entered; program will calculate results	A non-integer is entered; program rejects it and re-prompts	An integer that is too large is entered; program re-prompts for a smaller integer

storage_helper_function	An acceptable integer is entered; function will store the integer value and return	Function is called incorrectly; should do nothing if given a non-integer input	Function is called incorrectly; should do nothing if given a non-integer input
display_helper_function	The function completes successfully; the results are displayed	Function is called prematurely; should display what it has available, or nothing	Function is called prematurely; should display what it has available, or nothing

Part 2: Linked List Implementation

Understanding the problem:

We are tasked with implementing a linked list class using pointers, as well as analyzing the complexity of the algorithms that we use when implementing the linked list.

To make sure that you understand the problem correctly, answer the following guiding questions:

1. **(2 pts) What would each `Node` object store? What does the `next` pointer point to?**

   Each *Node* object would store an integer value and the pointer to the location of the next node in line in the list (the *next* pointer).

2. **(2 pts) In `Linked_List` class, what's the purpose of the `head` pointer? Why is it a pointer instead of a `Node` object?**

   The purpose of the *head* pointer is to give the programmer easy access to the first node in the list of nodes. It is a pointer so that you can directly access the head node instantly, allowing for much faster access to the top of the linked list, and so that more nodes can be added quickly through dynamic access to the linked list.

(2 pts) [List assumptions that you made]

I am assuming that this program will only process integer values and won't be processing floats or other datatypes.

Program Design:

To help you break the problem down into smaller subtasks, answer the following guiding questions:

1. **(2 pts) How would you iterate through the entire linked list? How could you tell if it reaches the end of the list?**
   I would iterate through the entire linked list by using a variable to point to the head node, assigning another variable to the next node in the list, finally reassigning the initial variable to the current node in the list. I could tell if it reaches the end of the list by checking that the next node in line doesn't exist.

2. **(1 pt) How would you swap two nodes in a given linked list? Write down the steps in pseudocode.**

   swap_data:
   set a temporary node variable to node1
   set node1 equal to node2
   set node2 equal to the temporary node variable
   set temporary node variable equal to previous node1
   set previous node1 equal to previous node2
   set previous node2 equal to temporary node variable

3. **(1 pt) Can any of the listed functions serve as a helper function to others? If so, how?**
   Yes, some of the listed functions can serve as helper functions to others. The helper functions that would be useful are *push/pop_front* and *push/pop_back*, along with *insert*, *remove*, and *sort_ascending/sort_descending*. They are useful in that they make moving and working with a linked list much easier to do.

4. **(1 pt) How will you implement the merge sort algorithm to work on a linked list in `sort_ascending()`?**
   I would implement the merge sort algorithm to work on a linked list in sort_ascending() by creating a recursive function that performs the merge sort algorithm on the linked list nodes. I would need to split the linked list into two halves before merging the two sorted linked lists, combining them back into one list.

5. **(1 pt) What algorithm are you going to use for `sort_descending()`? How will you implement it to work on a linked list?**
   I would also implement the merge sort algorithm to work on a linked list in sort_descending() by creating a recursive function that performs the merge sort algorithm like previously described. Instead of comparing for ascending order, I would compare for descending order.

6. **(1 pt) What other private member variables are you planning to add? What is the purpose of each?**
   Another private member variable I am planning to add is a tail pointer. This makes it easier to access the end of the linked list, which can be useful when sorting or just working with the list in general.

7. **(1 pt) How many helper functions are you going to implement? What is the purpose of each of them?**
   I am going to implement 11 helper functions. The purpose of each of them is to either manipulate the list (insert/remove, push/pop front/back/node_n), to aid in sorting (clear, sort_ascending/descending), or to display (print)

(4 pts) [Create pseudocode (or flowchart) for every function listed in the document, as well as your helper function(s)]

get_length:

    iterate through/get the length of the list

print

    iterate through and print all integers in the list

clear

    delete the contents of the list and all nodes

push_front

    add an int to the front of the list

push_back

    add an int to the back of the list

insert

    add an int to a given location in the list

pop_back

    remove an int from the back of the list

pop_front

    remove an int from the front of the list

remove

    remove an int at a given location in the list

sort_ascending

sort the list ascending

sort_descending

sort the list descending

Program Testing:

To help you consider the possible test cases, answer the following guiding questions:

1. (1 pt) What is an example of good input?
   Good input is an integer value.

2. (1 pt) Are there any inputs that would cause your program to crash?
   Non-integer input – specifically strings and chars.

3. (1 pt) What is your understanding of the providing test cases?
   My understanding of the provided test cases is that they test the various functions written in the program, including the ability to push/pop, insert/remove, and sort ascending/descending.

(4 pts) [Create a testing table that has representative good, bad, and edge cases, and their expected outputs]

Conditions	Good	Bad	Edge
Push/pop	An acceptable node is entered; program will push/pop node as needed	A non-node is entered; program rejects it and does nothing further to the list	A node/non-node is entered before the list is made; program rejects it and does nothing further to the list
Insert/remove	An acceptable integer is entered; function will store/remove the integer value and return	Function is called incorrectly; should do nothing to the list if given a non-integer input or invalid location	Function is called incorrectly; should do nothing to the list if given a non-integer input or invalid location
Sort ascending/descending	The function completes successfully; the results are sorted and displayed	Function is called prematurely; should display what it has available, or nothing	Function is called prematurely; should display what it has available, or nothing