

PONDER 06 : INSERTION SORT

Due Saturday at 11:59 PM MST

The sixth programming assignment will be to implement a linked-list and use it to implement the Insertion Sort.

Linked-List

Create a class encapsulating the notion of a linked-list Node. You will only be required to make a singly-linked-list for this assignment so you will only need to point to the next Node in the list. Of course, your linked-list will need to be able to store any data-type.

The Node class will consist of data, a next pointer, and constructors. Your class should only have two methods: constructors:

* **Constructors**: You will want to have exactly two methods in your Node class: a default constructor setting pNext to NULL, and a non-default constructor which takes a T as a parameter.

Create the following stand-alone (non-member) functions to be used for building a linked list. These non-member functions should be in your "node.h" file along with the Node class definition.

* **copy()**: Copy a linked-list. Takes a pointer to a Node as a parameter and returns a newly created linked-list containing a copy of all the nodes below the list represented by the parameter. This should be a non-member function.
* **insert()**: Insert a new Node into the current linked-list. The first parameter is the value to be placed in the new Node. The second parameter is the Node preceding the new Node in the list. An optional third parameter is set to true if the new item is to be at the head of the list. This should be a non-member function.
* **find()**: Find the Node corresponding to a given passed value from a given linked-list. The first parameter is a pointer to the front of the list, the second is the value to be found. The return value is a pointer to the found node if one exists. This should be a non-member function.
* **operator<<()**: Display the contents of a given linked-list.
* **freeData()**: Release all the memory contained in a given linked-list. The one parameter is a pass-by-reference pointer to the head of the list. Once the memory is freed this function should set the parameter to NULL. This should be a non-member function.

Unlike all the previous assignments, this will not be a self-contained abstract data-type. We will do that next lesson. Instead, this will just be a linked-list. The client will become aware of all the inner workings of this linked-list.

A few hints that may come in handy when implementing this data-structure:

* A single Node is not a linked-list. It should therefore only be aware of itself and the node following. Don't try to make your Node class or structure do too much. We will build a Listclass next week to encapsulate the notation of an entire linked-list.
* It is easiest to create the display and free function recursively. Other functions can be done recursively as well.
* Please do not create more methods for your Node class than the constructors. All the other functions should be non-member functions.
* Do not forget to remove all the nodes in the linked-list when the object is destroyed.
* You may need more functions or methods for your linked-list than described at the beginning of the assignment.

Insertion Sort

In addition to the linked-list, you will need to implement an insertion sort. As you may recall from previous semesters, a sort is an algorithm that converts an un-ordered collection of values into an ordered collection of values. By convention, sorts order the collection from least to greatest. In order for this to happen, sort algorithms typically need the collection data-type to support the less-than operator and the assignment operator.

The insertion sort is a sort similar to the method most humans use when ordering a deck of cards or a collection of papers. There are two parts to an insertion sort: the unsorted list and the sorted list. Initially the unsorted list consists of the entire collection and the sorted list is empty. With each iteration, items are removed from the unsorted list and inserted into the sorted list. This operation is continued until the unsorted list is empty and the sorted list consists of all the items in the collection.

You are to implement the insertion sort using your linked-list. This will be done because inserting items into the middle of a linked-list can be done in constant time, unlike array implementations where shifting of the array is required. The input will be an unsorted array of values. Build a sorted linked-list from this array and, when finished, copy the elements from the linked-list back into the input array. The stub-function for the insertion sort function is the following:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\* INSERTION SORT  
\* Sort the items in the array  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
template <class T>  
void insertionSort(T array[], int num)  
{  
// your code here  
}

Your insertion sort algorithm must be performed in O(n2) to get full credit.

Common Mistakes

The most common mistakes students make with this assignment include the following:

* **Insert at head**. Inserting a node at the head of the linked-list needs to be a special case somehow. This can be done many ways of course.
* **Dereferencing a NULL pointer**. When you attempt to dereference a NULL pointer, the program will crash. This can be very frustrating; you are not given a clue as to where the crash happened. You can use couts and cerrs to isolate the problem. Another solution is to use the GDB debugger. This can be accomplished in four steps:
  1. Compile your programs with the -g switch:

g++ file.cpp -g

Note that if you are using separate compilation, you will need to use the -g switch both when you compile the object file (g++ -c week06.cpp -g) and when you do the final link (g++ week06.o).

* 1. Load the program into GDB with:

gdb a.out

* 1. In GDB, execute the program itself. This can be done with:

(gdb) run

* 1. When the crash happens, execution will stop. Then you can ask GDB where the crash occured. This can be done with:

(gdb) bt

Test Bed

The testBed for this assignment is:

testBed cs235/week06 week06.tar

You can also run testBed on the executable:

testBed cs235/week06 a.out

Of course, you will need to pass testBed to get full credit on the assignment.

Submitting

You will submit this assignment as a pair using the Linux submit command. Please:

1. Create a TAR file built from the makefile, which will contain at least four files:
   * makefile: Directly from /home/cs235/week06/makefile except with your edits on the comment block.
   * node.h: Your definition for Node.
   * sortInsertionSort.h: Containing the code for sortInsertion() and any other functions or classes you may need.
   * week06.cpp: Unmodified from /home/cs235/week06/week06.cpp.
2. Run the program by hand a few times through all four test cases as well as the insertion sort program.
3. Verify your solution with testBed.
4. Submit your file using the submit command. The submit command will prompt you for your instructor, the class (cs235), and the assignment (week06). You submit your file with:

submit week06.tar

Your program will be graded according to the following rubric:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Exceptional 100% | Good 90% | Acceptable 70% | Developing 50% | Missing 0% |
| Node interface  20% | The Nodeclass/structure and all associated actions are defined in the best possible way | week06.cppcompiles without modification | All the actions forNode as well as the Nodeclass/structure definition are implemented | Node has many of the same interfaces as the problem definition | The Nodeclass/structure and the associated actions does not resemble the problem definition |
| Node Implementation  40% | Passes all fourNode testBed tests | Passes three testBed tests | Passes two testBed tests | Passes one testBed test | Program fails to compile or does not pass any testBed tests |
| Insertion Sort  20% | The insertion sort part of the test-bed passes all the tests | A minor bug exists in the insertion sort algorithm | A major bug exists in the insertion sort algorithm | Elements of the solution exist | No attempt was made to implement the insertion sort |
| Code Quality  10% | There is no obvious room for improvement | All the principles of encapsulation and modularization are honored | One function is written in a "backwards" way or could be improved | Two or more functions appears "thrown together" | The code appears to be written without any obvious forethought |
| Style  10% | Great variable names, no errors, great comments | No obvious style errors | A few minor style errors: non-standard spacing, poor variable names, missing comments, etc. | Overly generic variable names, misleading comments, or other gross style errors | No knowledge of the BYU-I code style guidelines were demonstrated |

Please make sure to fill out the program header in the makefile with the following information: the amount of coding time required to complete the assignment and what was the most difficult part.