CSE221 Data Structures (Fall 2018)
Instructor: Prof. Myeongjae Jeon
Due date: Sep 18, 2018, 11:59 pm.

Assignment 1: Linked List

In this assignment, you will implement 12 functions intended to manipulate linked lists in different ways. In the lecture, we learnt about the importance of maintaining correct linked list semantics (i.e., correct form of a chain of elements) while applying manipulation functions (e.g., `insert`, `delete`, `reverse`, etc) at runtime on top of the linked list that was already built in a correct form. Also, we talked about the importance of long-term impact of memory leakage that will crash the program in case you dynamically allocate memory over and over but yet do not free it out properly. In this homework, we hope you learn about how to write the linked list functions considering those aspects. Our test cases will evaluate their correctness.

`LinkedList.h`

In this file, you should define a template class `LinkedList` that has the following member functions. Please note that you will, by default, have to write default `constructor` and `destructor`:

- Type LinkedList<Type>::Get(const int index)

  Retrieve the value located at `index`-th element in the linked list. If the `index` is invalid (e.g., -1) or out of range (i.e., passing the last element), return -1. Note that `Get()` does not delete any element from the linked list.

- void LinkedList<Type>::AddAtHead(const Type& val)

  Insert an element containing `val` at the head of the linked list. The element thus must be the first element in the linked list.

- void LinkedList<Type>::AddAtIndex(const int index, const Type& val)

  Insert an element containing `val` to a specific location such that after the insertion, the `val` is located at `index`. If `index` is zero, the function works identical to `AddAtHead()`. If `index` equals to the length of the linked list, the element will be placed at the very end of the linked list after insertion. If `index` is invalid or out of range, `val` will not be inserted.

- void LinkedList<Type>::DeleteAtIndex(const int index)

  Delete an element located at `index`. If `index` is invalid or out of range, no element will be deleted.

- void LinkedList<Type>::DeleteValue(const Type& val)

  Delete an element that contains `val`. If you have multiple elements in the linked list that have the same `val`, delete the first encountered one only, not all. If none has `val`, do not delete any element.

- void LinkedList<Type>::MoveToHead(const Type& val)

  Move the first element that contains `val` to the head of the list. If there is no element containing `val`, no change to the linked list will be made.

- void LinkedList<Type>::Rotate(const int steps)

  Rotate the linked list right by `steps` times. By the nature of rotating, for each rotation step, you will need to move the last element to the head of the linked list. Assume that `steps` is a positive value: you can simply return the function if you encounter a negative `steps` value.

- void LinkedList<Type>::Reduce()

  Reduce the linked list such that you do not contain any duplicates in the linked list. In other words, when a certain value repeats multiple times in the linked list, you will need to maintain only one element (that you encounter first) and remove all subsequent elements.

- void LinkedList<Type>::Swap()

  Swap every two neighboring elements. For example, you swap the first two elements (index 0 and 1), next two elements (index 2 and 3), another next two elements (index 4 and 5), and so on. When you have an odd number of elements, you will end up having the last element without its neighboring element to swap. Then that element just stays there without being swapped with anyone.

- int LinkedList<Type>::Size()

  Return the number of elements in the linked list.

- void LinkedList<Type>::CleanUp()

Delete all elements from the linked list.

- void LinkedList<Type>::Print()

  Print out all current elements in the linked list in order. The printout format will be explained.

`main.cpp` exemplifies how we will assume a series of work on creating a linked list and calling its member functions. `LinkedList.h` is provided including skeleton functions you will need to implement. Please DO NOT change the function formats currently defined in `LinkedList.h`. In addition to the functions listed above, you are allowed to add other functions if needed.

For simplicity, we will use only **non-zero integer** values in evaluating your homework. That being said, in any detailed implementation, it would be safe to assume that you are only processing non-zero integer data. However, there are some fundamental tasks you have to do irrelevant to data types. For example, for each newly inserted value, you have to create an element including an integer value and its link to point out the next element. Note that dealing with dynamic insertion and deletion of values are an important part of this homework.

For another simplicity, we are NOT going to test the performance of your implementation. That being said, we do not expect that you maintain an available list of free elements in order to minimize expensive function calls such as `new` and `delete`. In this homework, please be focused on implementing the above functions in a correct way.

Output format in Print()

In `LinkedList.h` file, there is a function named `Print()` that you write to print out current elements in the linked list in order. All students are required to print out elements' values in a consistent way, as follows:

(1,3,5,7,3,1,9,3)

, where the first element (1) is referred by the header pointer of the list and the last element (3) is what you place at the end of the list. Notice that there is **no whitespace** in printout message.

Advanced functions: Rotate(), Reduce(), Swap()

While it is clear from the function descriptions that what each of these advanced functions does, we want to spend some time to explain about how they behave by taking examples. Given a linked list: (1,3,5,7,3,1,9,3),

Rotate(1) will return: (3,1,3,5,7,3,1,9)

Rotate(2) will return: (9,3,1,3,5,7,3,1)

Rotate(3) will return: (1,9,3,1,3,5,7,3)

Reduce() will return: (1,3,5,7,9)

- ⇨ 1 and 3 appear at multiple locations. We should remove such duplicates, and keep only one unique location for each. Note again that we will keep the first encountered value only for each.

Swap() will return: (3,1,7,5,1,3,3,9)

Note

- You are not allowed to use STL or other libraries.
- You are not allowed to use more than 1 token for extending the deadline this time.
- Grading: compilation (5pt) and 19 test cases (5pt for each test case)
- Make sure you are not violating academic honesty. If you are confused, don not make any guess and immediately talk to either TAs or the instructor.
- While we use only non-zero integers for testing, your template class must be able to handle any data type taking non-zero integers, e.g., `int`, `long`, `short`, etc.
- In case you find out bugs, typos or problems in the given main.cpp, please report to Prof. Myeongjae Jeon at mjjeon@unist.ac.kr. We will look into them and fix as soon as we can. Once fixed, we will push the new version in your repo by naming proper version number for the code, and announce the change by email. Please pay attention to it and make sure you are using the latest version before submission.