Ceng213 - Data Structures
Programming Assignment 1 : String Implementation via Linked Lists

Spring 2021

1 Objectives

In this programming assignment, you are first expected to implement a *doubly linked list with dummy nodes* data structure, in which each node will contain data and two pointers to the previous and the next nodes. The linked list data structure will include two pointers that points to the *dummy head* and the *dummy tail* nodes of the linked list. The details of the structure are explained further in the following sections. Then, you will use this specialized linked list structure to implement *strings*.

Keywords: *C++, Data Structures, Linked List, Doubly Linked List, Dummy Nodes, String Implementation*

2 Linked List Implementation (50 pts)

The linked list data structure used in this assignment is implemented as the class template `LinkedList` with the template argument `T`, which is used as the type of the data stored in the nodes. The node of the linked list is implemented as the class template `Node` with the template argument `T`, which is the type of the data stored in nodes. `Node` class is the basic building block of the `LinkedList` class. `LinkedList` class has two `Node` pointers in its private data field (namely `dummyHead` and `dummyTail`) which point to the dummy head and the dummy tail nodes of the linked list.

The `LinkedList` class has its definition and implementation in *LinkedList.h* file and the `Node` class has its in *Node.h* file.

2.1 `Node`

`Node` class represents nodes that constitute linked lists. A `Node` keeps two pointers (namely `prev` and `next`) to its previous and next nodes in the list, and the data variable of type T (namely `data`) to hold the data. The class has three constructors, and the overloaded output operator. They are already implemented for you. You should not change anything in file *Node.h*.

2.2 LinkedList

`LinkedList` class implements a doubly linked list with dummy nodes data structure with the `dummyHead` and the `dummyTail` pointers. Previously, data members of `LinkedList` class have been briefly described. Their use will be elaborated in the context of utility functions discussed in the following subsections. You must provide implementations for the following public interface methods that have been declared under indicated portions of *LinkedList.h* file.

There are two types of nodes in this linked list implementation: *dummy nodes* and *data nodes*. *Dummy nodes* are the first and last nodes in the list that do not contain any useful data but always specify the location of the front and tail of the list. *Data nodes* are the other nodes (i.e., the middle nodes) in the list that contain useful data.

2.2.1 `LinkedList();`

This is the default constructor. You should initialize the linked list by creating the dummy head and tail nodes and setting their pointers to make the dummy tail node the next node of the dummy head node, and the dummy head node the previous node of the dummy tail node.

2.2.2 `LinkedList(const LinkedList<T> &obj);`

This is the copy constructor. You should initialize the linked list as described in the previous chapter, create new nodes by copying the data nodes in given `obj`, and insert new nodes into the linked list.

2.2.3 `~LinkedList();`

This is the destructor. You should deallocate all the memory that you allocated before.

2.2.4 `int getSize() const;`

This function should return the number of data nodes in the linked list.

2.2.5 `bool isEmpty() const;`

This function should return `true` if the linked list is empty (i.e., there exists no data nodes in the linked list). If the linked list is not empty, it should return `false`.

2.2.6 `bool contains(Node<T> *node) const;`

This function should return `true` if the linked list contains the given `node` (i.e., any `next`/`prev` in the linked list matches with `node`). Otherwise, it should return `false`.

2.2.7 `Node<T> *getDummyHead() const;`

This function should return a pointer to the dummy head node in the linked list.

2.2.8 `Node<T> *getDummyTail() const;`

This function should return a pointer to the dummy tail node in the linked list.

2.2.9 `Node<T> *getActualHead() const;`

This function should return a pointer to the first data node in the linked list. If the linked list is empty, it should return `NULL`.

2.2.10 `Node<T> *getActualTail() const;`

This function should return a pointer to the last data node in the linked list. If the linked list is empty, it should return `NULL`.

2.2.11 `Node<T> *getNode(const T &data) const;`

You should search the linked list for the data node that has the same data with the given `data` and return a pointer to that node. You can use `operator==` to compare two `T` objects. If there exists no such data node in the linked list, you should return `NULL`.

2.2.12 `Node<T> *getNodeAtIndex(int index) const;`

You should search the linked list for the data node at given zero-based `index` (i.e., `index=0` means first data node, `index=1` means second data node, ...) and return a pointer to that node. If there exists no such data node in the linked list (i.e., `index` is out of bounds), you should return `NULL`.

2.2.13 `void insertAtTheBeginning(const T &data);`

You should create a new node with given `data` and insert it at the beginning of the linked list as the first data node. Don't forget to make necessary pointer, and dummy head/tail modifications.

2.2.14 `void insertAtTheEnd(const T &data);`

You should create a new node with given `data` and insert it at the end of the linked list as the last data node. Don't forget to make necessary pointer, and dummy head/tail modifications.

2.2.15 `void insertBeforeGivenNode(const T &data, Node<T> *node);`

You should create a new node with given `data` and insert it before the given data node `node` as its previous node. Don't forget to make necessary pointer, and dummy head/tail modifications. If the given `node` is not in the linked list, do nothing.

2.2.16 `void deleteNode(Node<T> *node);`

You should delete the given data node `node` from the linked list. Don't forget to make necessary pointer, and dummy head/tail modifications. If the given `node` is not in the linked list (i.e., the linked list does not contain the given `node`), do nothing.

2.2.17 `void deleteNode(const T &data);`

You should delete the data node that has the same data with the given `data` from the linked list. Don't forget to make necessary pointer, and dummy head/tail modifications. If there exists no such node in the linked list, do nothing.

2.2.18 `void deleteAllNodes();`

You should delete all data nodes in the linked list so that the linked list becomes empty.

2.2.19 `void swapNodes(Node<T> *node1, Node<T> *node2);`

You should swap the two given data nodes `node1` and `node2`. You are not allowed to just swap the data in the nodes. Also, you are not allowed to create new nodes in this function. Don't forget to make necessary pointer, and dummy head/tail modifications. If either of the given nodes is not in the linked list, do nothing.

2.2.20 `LinkedList<T> &operator=(const LinkedList<T> &rhs);`

This is the overloaded assignment operator. You should delete all data nodes in the linked list and then create new nodes by copying the data nodes in given `rhs` and insert new nodes into the linked list.

3 String Implementation (50 pts)

The strings in this assignment are implemented as the class `String`. `String` class has a `LinkedList` object in its private data field (namely `characters`) with the type `char`. This `LinkedList` object keeps characters of the string.

The `String` class has its definition in *String.h* file and its implementation in *String.cpp* file.

3.1 `String`

In `String` class, all member functions should utilize `characters` member variable to operate as described in the following subsections. In *String.cpp* file, you need to provide implementations for following functions declared under *String.h* header to complete the assignment.

3.1.1 `int getLength() const;`

This function should return the length of the string (i.e., number of characters in the string).

3.1.2 `bool isEmpty() const;`

This function should return `true` if the string is empty (i.e., there exists no characters in the string). If the string is not empty, it should return `false`.

3.1.3 `void appendCharacter(char character);`

This function appends a new character to the string. It takes the character to append as parameter (`character`) and inserts a new character to the `characters` linked list.

3.1.4 `void insertCharacter(char character, int pos);`

This function inserts a new character to the string at the given zero-based position (i.e., `pos=0` means insert as the first character, `pos=1` means insert as the second character, ...). It takes the character to insert and the position to insert as parameters (`character` and `pos`) and inserts a

new character to the `characters` linked list at the correct position. If `pos` is out of bounds, you should do nothing.

3.1.5 `void eraseCharacters(int pos, int len);`

This function erases the portion of the string that begins at the zero-based position `pos` and spans `len` characters. It takes the position of the first character to be erased and the number of characters to erase as parameters (`pos` and `len`) and erases corresponding characters from the `characters` linked list. If given portion is out of bounds, you should do nothing.

3.1.6 `void eraseAllCharacters();`

You should erase all characters from the string so that the string becomes empty.

3.1.7 `String substring(int pos, int len);`

This function returns a newly constructed `String` object with its value initialized to a copy of a substring of this string. The substring is the portion of the object that starts at zero-based position `pos` and spans `len` characters. The function takes the position of the first character to be copied as a substring and the number of characters to include in the substring as parameters (`pos` and `len`). If given portion is out of bounds, you should do nothing.

3.1.8 `LinkedList<String> split(char separator);`

This function breaks up this string at the specified separators and returns a linked list of strings. It takes the separator to use when splitting the string as parameter (`separator`).

3.1.9 `bool isPalindrome() const;`

This function should return `true` if the string is a palindrome (i.e., a word, sentence, or longer written work that reads the same backwards ignoring whitespaces). If the string is not a palindrome, it should return `false`. For this function, you may assume that all characters in the string are either lowercase letters (a-z) or space.

3.1.10 `bool operator<(const String &rhs) const;`

This is the overloaded less than comparison operator. If this string is lexicographically less than the given string `rhs`, return `true`. Otherwise, return `false`.

4 Driver Programs

To enable you to test your `LinkedList` and `String` implementations, two driver programs, *main_linkedlist.cpp* and *main_string.cpp* are provided. Their expected outputs are also provided in *output_linkedlist.txt* and *output_string.txt* files, respectively.

5 Regulations

1. **Programming Language:** You will use C++.
2. **S**tandard Template Library is **not** allowed.

3. **E**xternal libraries other than those already included are **not** allowed.

4. **T**hose who do the operations (insert, delete, get) without utilizing the linked list will receive **0 grade**.

5. **T**hose who modify already implemented functions and those who insert other data variables or public functions and those who change the prototype of given functions will receive **0 grade**.

6. **T**hose who use STL vector or compile-time arrays or variable-size arrays (not existing in ANSI C++) will receive **0 grade**. Options used for g++ are "-ansi -Wall -pedantic-errors -O0". They are already included in the provided Makefile.

7. **Y**ou can add private member functions whenever it is explicitly allowed.

8. **Late Submission Policy:** Each student receives 5 late days for the entire semester. You may use late days on programming assignments, and each allows you to submit up to 24 hours late without penalty. For example, if an assignment is due on Thursday at 11:30pm, you could use 2 late days to submit on Saturday by 11:30pm with no penalty. Once a student has used up all their late days, each successive day that an assignment is late will result in a loss of 5% on that assignment.

   No assignment may be submitted more than 3 days (72 hours) late without permission from the course instructor. In other words, this means there is a practical upper limit of 3 late days usable per assignment. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.

9. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations. Remember that students of this course are bounded to code of honor and its violation is subject to severe punishment.

10. **Newsgroup:** You must follow the Forum (`odtuclass.metu.edu.tr`) for discussions and possible updates on a daily basis.

6 Submission

- Submission will be done via CengClass (`cengclass.ceng.metu.edu.tr`).
- Don't write a main function in any of your source files.
- A test environment will be ready in CengClass.
  - You can submit your source files to CengClass and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for evaluation of your final grade. So, your actual grades may be different than the ones you get in CengClass.
  - Only the last submission before the deadline will be graded.