Middle East Technical University

Department of Computer Engineering

CENG 213

Data Structures
Fall 2022
Programming Assignment 1 - A Browser Tab Mechanism
Implementation via Linked Lists

Due: 6. November 2022 23:55
Submission: **via ODTUClass**

1 Objectives

In this programming assignment, you will first implement a ***circular, doubly linked list***. Then, using the linked list implementation, you will build a simple tabbed browser that supports multiple tabs and windows. The details of the assignment are explained in the following sections.

Keywords: C++, Data Structures, Linked List, Circular Doubly Linked List, Browser, Tab

2 Linked List Implementation

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 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 data stored in nodes. `Node` class is the basic building block of the `LinkedList` class. `LinkedList` class has a `Node` pointer in its private data field (namely `head`) which points to the first node of the linked list. It also has an `int` that keeps the number of nodes in the list, called `size`.

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

2.1 `Node` class

`Node` class represents nodes that constitute linked lists. A `Node` instance keeps two pointers (namely `prev` and `next`) to its previous and next nodes in the list, and a data

variable of type T (namely `data`) to hold the data. The class has two constructors and the overloaded output operator (`operator<<`). They are already implemented for you. You should not change anything in file *Node.h*.

2.2 `LinkedList` class

2.2.1 `LinkedList()`

This is the default constructor. You should make necessary initialization(s) in this function. Don't forget that our list will be **circular, doubly linked list**.

2.2.2 `LinkedList(const LinkedList &rhs)`

This is the copy constructor. You should make necessary initializations, create new nodes by copying the nodes in the `rhs`, and insert those new nodes into the linked list.

2.2.3 `~LinkedList()`

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

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

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

2.2.5 `int getSize() const`

This function should return an integer that is the number of nodes in the linked list (i.e., the size of the linked list).

2.2.6 `bool isEmpty() const`

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

2.2.7 `bool containsNode(Node<T> *node) const`

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

2.2.8 `int getIndex(Node<T> *node) const`

This function should return the index of the given node in the linked list. If the node is not in the list, it should return `-1` instead.

2.2.9 `Node<T> *getFirstNode() const`

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

2.2.10 `Node<T> *getLastNode() const`

This function should return a pointer to the last 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 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 multiple such nodes in the linked list, return a pointer to the first occurrence. If there exists no such 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 node at the given zero-based `index` (i.e., `index=0` means the first node, `index=1` means the second node, ..., `index=size-1` means the last node) and return a pointer to that node. If there exists no such node in the linked list (i.e., `index` is out of bounds), you should return `NULL`.

2.2.13 `void append(const T &data)`

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

2.2.14 `void prepend(const T &data)`

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

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

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

2.2.16 `void insertAtIndex(const T &data, int index)`

You should create a new node with the given data and insert it at the given `index` (i.e., `index=0` means the first node, `index=1` means the second node, ..., `index=size-1` means the last node). Don't forget to make necessary pointer, head and size modifications. If there exists no such index in the linked list (i.e., index is out of bounds), you should not insert the element.

2.2.17 `void moveToIndex(int currentIndex, int newIndex)`

This function should move the node at the `currentIndex` to `newIndex`. For this function, you are not allowed to just change the data in the given nodes. Also, you are not allowed to create new nodes. Do the moving by changing the pointers in the nodes of the linked list. If the `newIndex` is greater than the number of nodes in the list, then the node should be moved to the end of the list. If the `currentIndex` is greater than the number of nodes in the list, then do nothing.

2.2.18 `void removeNode(Node<T> *node)`

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

2.2.19 `void removeNode(const T &data)`

You should remove the node that has the same data with the given `data` from the linked list. Don't forget to make necessary pointer, head and size modifications. If there exists multiple such nodes in the linked list, remove all occurrences. If there exists no such node in the linked list, do nothing.

2.2.20 `void removeNodeAtIndex(int index)`

You should remove the node at the given `index` from the list. Don't forget to make necessary pointer, head and size modifications. If there exists no such `index` in the linked list (i.e., `index` is out of bounds), or if the list is empty, you should not do anything.

2.2.21 `void removeAllNodes()`

You should remove all nodes in the linked list so that the linked list becomes empty. Don't forget to make necessary pointer, head and size modifications.

2.2.22 `void print()`

This function print the linked list to the standard output. It is provided for you.

3 Browser Implementation

3.1 `Tab`

The `Tab` class represents a browser tab in our homework. It has its `operator<<` and `operator==` overloaded for you. You should not change anything in the file *Tab.h*.

3.2 Window

The `Window` class represents an open window of the browser. This class has two private member variables: `tabs`, a `LinkedList<Tab>` that keeps the tabs that are in the window; and `activeTab`, an `int` that holds the index of the currently active tab of the window. Public methods of the `Window` class are explained in the following sections.

3.2.1 `Tab getActiveTab()`

This function should return the active tab of the window. If there are no tabs in the window, this function should return an empty `Tab`.

3.2.2 `bool isEmpty() const`

This function should return `true` if the `Window` is empty (i.e., there are no tabs left in the window). If the `Window` is not empty, it should return `false`.

3.2.3 `void newTab(const Tab &tab)`

This function should add a new tab to the `Window`. The new tab should be added after the active tab of the window (i.e. if the active tab is the fourth tab, then the new tab should be added as the fifth tab), and should be made the active tab of the window.

3.2.4 `void closeTab()`

This function should destroy the active tab of the `Window`. The new active tab must be the tab that was following the previous active tab, if there were any other tabs in the window other than the active tab. If there are no tabs left in the window, `activeTab` must be set to `-1`.

3.2.5 `void moveActiveTabTo(int index)`

This function should change the position of the active tab of the window. Active tab should be moved to the position of the `index`. If the `index` is greater than the number of nodes in the list, then the active tab must be moved to the end of the list.

3.2.6 `void changeActiveTabTo(int index)`

This function should change which tab is the active tab of the window. The `index`'th tab of the window should be made the active tab. If the `index` is greater than the number of tabs in the window, then do nothing.

3.2.7 `void closeTab(Node<Tab> &tab)`

This function should destroy the given tab in the window. Do not forget to make the necessary pointer and head modifications. If there are no tabs left in the window, `activeTab` must be set to `-1`.

3.2.8 `void addTab(Node<Tab> &tab)`

This function should add the given tab to the `Window`. We will use this function to add tabs that were removed from other windows. The position of the tab should be right after the active tab. If the `Window` is empty, then the newly added tab should be made the active tab.

3.3 `Browser`

The Browser represents the browser itself. This class has one member variable: `windows`, a `LinkedList<Window>` that holds the windows of the browser.

3.3.1 `void newWindow()`

This function should create a new empty window in the browser. The window that is created should be made the first element of the list.

3.3.2 `void closeWindow()`

This function should destroy the first window of the browser. If there are no windows in the browser, do nothing.

3.3.3 `void switchToWindow(int index)`

This function should make the given window the first window of the browser.

3.3.4 `void moveTab(Window &from, Window &to)`

This function closes the active tab of the `from` window, and create a new tab with the same data in the `to` window. If `from` becomes empty after its active tab is closed, then it should be closed as well.

3.3.5 `void mergeWindows(Window *window1, Window *window2)`

This function merges two windows together by adding all the tabs of one window to the other. In this function, you should add the tabs of `window2` to `window1`. You shouldn't change the order of the tabs in `window1`, and the tabs from `window2` must come after the tabs of `window1` in its list.

3.3.6 `void mergeAllWindows()`

This function merges all the windows of the browser into a single window. Tabs of all the windows in the browser must be put into the first window. You shouldn't change the order of the tabs of the first window, and the windows should be merged in the order that they appear in `windows`.

3.3.7 `void closeAllWindows()`

This function closes all windows of the browser.

3.3.8 `void closeEmptyWindows()`

This function should look for empty windows in the browser, and close any empty windows that is finds. If there aren't any empty windows in the browser, then do nothing.

4 Driver Programs

To enable you to test your `LinkedList` and `Browser` implementations, two driver programs, *main_linkedlist.cpp* and *main_browser.cpp* are provided.

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, remove, 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.