Middle East Technical University Department of Computer Engineering

CENG 213

Data Structures

Fall '2016-2017

Programming Assignment 1

Due date: 27 November 2016, Sunday, 23:55

1 Objectives

This assignment aims to make you familiar with linked list and stack implementations. You will implement template classes for these containers then you will use them in a problem to further improve your skills.

Keywords: *Linked List, Stack*

2 Stack Template (20pts)

You will implement a Stack template that can be used with different data types. The stack will be represented by a dynamic array that you will resize if necessary. You cannot use *vector* class or any other container in STL. Throw exceptions for out-of-bonds operations such as pop from empty stack.

2.1 Private Data

The class has a pointer to represent the array and two integers (size_t) to show the capacity of the stack and the number of items currently on the stack.

```
/* array keeping the items of the stack */
T* items;
/* number of items in the stack */
size_t  size;
/* capacity of the stack */
size_t  capacity;
```

You can add other variables and functions to the private part of the class.

2.2 Interface

Public interface of the class shows its capabilities. Do not modify the public part of the class.

2.2.1 `Stack(int capacity=8);`

The constructor of the class allocates an array of given capacity. The number of items is 0 and the default capacity in 8.

2.2.2 `Stack(const Stack<T>& stack);`
`Stack<T>& operator=(const Stack<T>& stack);`
`~Stack();`

The copy constructor, assignment operator and the destructor should work as expected.

2.2.3 `void push(const T& item);`

This method is used to push an item onto the stack. The new item must be placed at the top of the stack as expected.
If the stack is full when you need to push a new item, allocate a new array that has twice the capacity of the old one and transfer all of the items to the new array. Do not resize after pushing an item but do it before the push operation (Do not check whether the stack is full or not after a successful push operation).

2.2.4 `T pop();`

This method is used to remove the top item of the stack and return a copy of it.
If the stack contain items less than one third of the capacity you need to allocate a new array that has half of the capacity of the old one. Do not make the capacity less than 8 items. Do not resize after popping an item but do it before the pop operation (Do not compare the size and the capacity after pop operation).

2.2.5 `const T& top() const;`

This method returns the top item of the stack without modifying the stack. Consider this as a peek at the stack.

2.2.6 `void clear();`

This methods clears the contents of the stack. It deallocates the items, then creates a new empty stack of capacity 8.

2.2.7 `bool isEmpty() const;`

This method returns *true* if the stack has no items and *false* otherwise.

2.2.8 `size_t getSize() const;`
`size_t getCapacity() const;`

These methods return the number of items in the stack and the capacity of the stack.

2.2.9 `void print() const;`
`void printReversed() const;`

These methods print the contents of the stack, from top to bottom one item per line for the print and from bottom to top for the reversed print. It assumes *operator* << is overloaded for any type T that is used to create a stack.

3 Linked List Template (30pts)

You will implement a doubly linked list template. For that, you will implement two classes with the given interfaces below. The first one is the node container and the second one is the actual linked list. Throw exceptions for out-of-bonds operations such as accessing an invalid position in the list.

3.1 Node Private Data

A node contains a data member of a given type T. Additionally it has pointers to the previous and the next nodes of the same type.

```
/* pointer to the previous node */
Node<T> *prev;
/* pointer to the next node */
Node<T> *next;
/* data of type T */
T data;
```

Do not modify the node class.

3.2 Node Interface

3.2.1 Node(); Node(const T& d);

The node class has a default constructor and constructor that sets its data.

3.2.2 void setNext(Node<T> *newNext); void setPrev(Node<T> *newPrev); void setData(const T& data);

Setter functions of the Node class assigns new values to the respective members.

3.2.3 Node<T>* getNext() const; Node<T>* getPrev() const; T getData() const;

Getter functions of the node class returns the values of its members.

3.2.4 T* getDataPtr();

This method returns the pointer of the data in the node so that you can access and modify the data.

3.2.5 friend ostream &operator<< <> (ostream &out, const Node<T>& n);

This helper method is provided for your convenience. It prints the values of the pointers and data in the node.
You will see an output similar to this:

```
  .........0    <-| 0x2126010 |-> .0x2126030 : 1101
.0x2126250 <-| 0x2126270 |-> .........0  :  1120
```

You can play with the implementation and use it for debugging, this method will not be used in the evaluation of your assignment.

```
/* forward declerations for overloading operator<< in template*/
template<class T>
class Node;
template<class T>
ostream &operator<<(ostream &out, const Node<T>& n);

/* method implementation */
template<class T>
ostream &operator<<(ostream &out, const Node<T>& n){
    out << setfill('.') << setw(10) << (void*)n.prev
        <<" <-| "<< (void*)&n <<" |-> "
        << setfill('.') << setw(10) << (void*)n.next << " : "
        << n.data ;
    return out;
};
```

3.3 LinkedList Private Data

The linked list does not have a dummy head and keeps pointers to both ends:

```
/* pointer to the first node */
Node<T>* front;
/* pointer to the last node */
Node<T>* back;
```

You can add other variables and functions to the private part of the class.

3.4 LinkedList Interface

Public interface of the class shows its capabilities. Do not modify the public part of the class.

3.4.1 LinkedList();

Default constructor should set both ends to NULL. After this, the list contains no elements and the size of the list is zero.

3.4.2 LinkedList(const LinkedList<T>& ll);
LinkedList<T>& operator=(const LinkedList<T>& ll);
~LinkedList();

The copy constructor, assignment operator and the destructor should work as expected.

3.4.3 Node<T>& getFront() const;
Node<T>& getBack() const;
Node<T>& getNodeAt(int pos) const;

Getter methods that return the nodes of the linked list. *getFront()* returns the first node, *getBack()* returns the last node and *getNodeAt(index)* returns the node at given position with zero based indices.

3.4.4 `Node<T>* getNodePtrAt(int pos) const;`

This method returns the pointer of the node at given position (with 0 for the first node).

3.4.5 `void insert(Node<T>* prev, const T& data);`

This method creates a new node with the given data and inserts the new node after the node given as its previous node. Note that this is a doubly linked list and set the connections appropriately. Consider the first node, front, back and middle positions of the linked list while inserting a new node.

3.4.6
```
void insertAt(int pos, const T& data);
void pushFront(const T& data);
void pushBack(const T& data);
```

These three functions are the other methods to insert new nodes to different positions in the linked list. *insertAt(index, data)* inserts the new node at the given position. *pushFront(data)* makes the new node the first node of the linked list and *pushBack(data)* makes it the last node. It is better to implement the *insert* function above and use it in these methods.

3.4.7 `void erase(Node<T>* node);`

This function removes the given node from the linked list. Make sure to connect the remaining parts of the list and deallocate any resources allocated for the given node.

3.4.8
```
void eraseFrom(int pos);
void popFront();
void popBack();
```

These are other methods to remove nodes from the list. *eraseFrom(index)* removes the node at given index (0 for the first node), *popFront()* and *popBack()* removes the nodes at the front and the back ends of the list.

3.4.9 `void clear();`

This methods clears the contents of the list. It deallocates the nodes but does not destroy the linked list.

3.4.10 `bool isEmpty() const;`

This method returns *true* is there are no items in the linked list, *false* otherwise.

3.4.11 `size_t getSize() const;`

This method returns the number of nodes in the linked list.

3.4.12 `void print() const;`

This method prints the contents of the linked list. You should only print the data in the nodes. Print each node in a single line from the front end to the back end of the list.

4 Browser (50pts)

You will implement a basic tab handling system in a web browser. The browser will have a single window with multiple tabs. Each tab will represent single web page with history of pages opened at that tab. The browser will have the ability to open, close and reopen web pages in different tabs, select and move tabs to change their order and move forward and backward in the history of a tab. The browser will have the ability to display its current state (currently viewed/selected tab, opened and closed tabs in the window) and the details of a single tab (previously visited urls in the tab). You will implement two classes, namely Tab and Browser to provide the expected functionalities. However, there are no given interfaces and templates for this classes except a single entry point. You have to decide and design the classes. You are expected to use *Stack* and *LinkedList* classes described in the previous two sections.

4.1 Tab Class

Tab class will hold the url of the current web page, previously visited pages in this tab for the back operation and forward history for the forward operation. You can implement the tab history using two stacks. There are no functions given and you are free to modify this class however you choose.

4.2 Browser Class

Browser class will hold the list of open tabs, the closed tabs -in a stack- so that they can be reopened and pointer to currently selected tab. You are free to add other variables. There is a single function void handleTask(string task); in the public part. The users of the class will only use this function. They will call the function with the commands described below. You are free to modify the class and add as many functions as you like. The browser will be used as shown below:

```
Browser browser;
browser.handleTask("open_new_page https://ceng.metu.edu.tr");
browser.handleTask("open_link_in_new_tab https://ceng.metu.edu.tr/assistants");
browser.handleTask("display");
```

4.3 Operations

The section describes the commands that can be used in the Browser.

4.3.1 open_new_page *url*

This will insert a new tab with the given address at the end of currently open tabs.

```
open_new_page metu.edu.tr
```

4.3.2 open_link *url*

This method will open the link in the current tab. Previous web pages should be saved in history of the tab.

```
open_link metu.edu.tr/general-information
```

4.3.3 open_link_in_new_tab *url*

This will open a new tab with the given link. The new tab will be positioned after the selected tab and then it will become the new selected tab.

```
open_link_in_new_tab ceng.metu.edu.tr/assistants
```

4.3.4 close_tab *index*

This will close the tab in the given position. 0 shows the first tab in the browser. Close tab operation will not change the selected tab, unless the index matches the selected tab. Do not forget to save closed tabs.

```
close_tab 1
```

4.3.5 reopen_closed_tab

This will reopen the closed tabs in the reverse order they are closed. You need keep every tab after a *close* command is used. Then you need to reopen the last closed tab when a *reopen* command is sent and you need to insert the tab at its previous position (the one where they are when the close command is called). The reopened tab becomes the selected tab.

```
reopen_closed_tab
```

4.3.6 move_tab *from_index to_index*

This method will move a tab from the first position to the other and make it the selected tab.

```
move_tab 4 1
```

4.3.7 display

This command will display the current browser state. The first line contains the position of the selected tab. The the next line contains the url of the tab. The list of open tabs follows after. Finally the list of closed tabs is shown in order so that they can be reopened if necessary. The number of open tabs and closed tabs are also shown before the urls.

```
display
```

An example output is shown below:

```
CURRENT TAB: 3
ii .metu.edu.tr
OPEN TABS: 4
ceng.metu.edu.tr
ceng.metu.edu.tr/staff
metu.edu.tr/faculties-institutes-schools
ii .metu.edu.tr
CLOSED TABS: 2
ceng.metu.edu.tr/announcement/fall-2016-2017-course-schedule
ceng.metu.edu.tr/assistants
```

4.3.8 select_tab *index*

This will change the selected tab and make the tab in the given position the current tab of the browser.

```
select_tab 0
```

4.3.9 display_tab_details

This command will display the history of the tab. It will first display the tabs available for a forward operation, then the current web page, then the page available for a back operation. The current page address is prepended with > and a single space character.

```
display_tab_details
```

An example output is shown below:

```
map.metu.edu.tr
metu.edu.tr/graduate-programs-and-degrees-offered-metu
metu.edu.tr/university-administration
> metu.edu.tr/general-information
metu.edu.tr
```

4.3.10 close_selected_tab

This command will close the selected tab and remove it from the list of open tabs. After the operation, the new tab at the same position becomes the selected tab. If the last tab is closed, the new tab at the end becomes selected tab.

```
close_selected_tab
```

4.3.11 back

This command will move backwards in the history of the current tab. Do not forget to save the addresses for *forward* operation.

```
back
```

4.3.12 forward

This command will move forwards in the history of the current tab. Do not forget to save the addresses for *back* operation.

```
forward
```

5 Regulations

1. **Programming Language:** You will use C++.
2. **S**tandard Template Library is not allowed, you cannot use *vector*, *stack* etc. that are available in STL.
3. **E**xternal libraries are not allowed.
4. **Late Submission:** You have 3 days for late submission.
5. **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.
6. **R**emember that students of this course are bounded to code of honor and its violation is subject to severe punishment.
7. **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.

6 Submission

- Submission will be done via Moodle.
- Do not write a *main* function in any of your source files.
- A test environment will be ready in Moodle.
  - You can submit your source files to Moodle and test your work with a subset of evaluation inputs and outputs.
  - This will not be the actual grade you get for this assignment, additional test cases will be used for evaluation.
  - Only the last submission before the deadline will be graded.