CS 202 - Computer Science II

Project 6

Due date (FIXED): Monday, 4/17/2017, 11:59 pm

Objectives: The main objectives of this project are to test your ability to create and use list-based dynamic data structures. A review of your knowledge to manipulate dynamic memory, classes, pointers and iostream to all extents, is also included.

Description:
For this project you will create List classes. Similarly to the previous Project, **Pointer-based** and **Bracket-notation** ([ ]) in order to perform **array manipulation** are both **allowed**. There will have to be two separate List-based implementations, one Array-based, and the other Node-based.

Array-based List:
The following header file extract is used to explain the required specifications for the class (the actual header ArrayList.h file is provided and accompanies the Project description):

```
…
class ArrayList{
  public:
    ArrayList();                                                //(1)
    ArrayList(int size, const DataType& value);                 //(2)
    ArrayList(const ArrayList& other);                          //(3)
    ~ArrayList();                                               //(4)

    ArrayList& operator= (const ArrayList& other_arrayList);    //(5)
    friend std::ostream& operator<<(std::ostream& os,           //(6)
                                    const ArrayList& arrayList);

    DataType* first();                                         //(7a)
    const DataType* first() const;                             //(7b)

    DataType* last();                                          //(8a)
    const DataType* last() const;                              //(8b)

    DataType* find(const DataType& target,                     //(9a)
                   DataType* &previous,
                   const DataType* const start = NULL);
    const DataType* find(const DataType& target,               //(9b)
                         const DataType* &previous,
                         const DataType* const start = NULL) const;

    DataType* insertAfter(const DataType& target,              //(10)
                          const DataType& value, int count=1);
    DataType* insertBefore(const DataType& target,             //(11)
                          const DataType& value, int count=1);
    DataType* removeForward(const DataType& target,            //(12)
                            int count=1);
    DataType* removeBackward(const DataType& target,           //(13)
                             int count=1);

    DataType& operator[] (int position);                      //(14a)
    const DataType& operator[] (int position) const;          //(14b)
    int size() const;                                           //(15)
    bool empty() const;                                         //(16)
    void clear();                                               //(17)

  private:
    void grow(int addsize);                                     //(18)
    void trim();                                                //(19)

    DataType *m_array;
    int m_size;
    int m_maxsize;
};
...
```

The **ArrayList** Class will contain the following **private** data members:

- **m_array,** a DataType class type Pointer, pointing to the Dynamically Allocated Array data. will need to be reallocated whenever it should grow to accommodate more data than it can fit, and possibly whenever it should trim down when it takes up too much space. Note: This implies that it might be larger in total size than the actually stored (& considered valid) data.
- **m_size**, an int, keeps track of how many DataType elements are currently stored & considered valid inside m_array. Note that this has to be properly initialized and updated each time the dynamically allocated memory is changed.
- **m_maxsize**, an int, denoting how many DataType type objects can fit in total in the currently allocated memory of m_array. Note that this has to be properly initialized and updated each time the dynamically allocated memory is changed, and that generally m_maxsize ≥ m_size.

, will have the following **private** helper methods**:**

- **(18) grow** – will deallocate the dynamic memory pointed to by m_array and then allocate enough total memory to fit the extra int addsize. Note: After successful call, the new m_maxsize should be the sum of the previous m_maxsize and addsize, while m_size should be unaltered. Also, the original m_array data should be copied over to the newly allocated one.
- **(19) trim** – will deallocate the dynamic memory pointed to by m_array and allocate enough total memory to just fit the current amount of m_size. Note: The m_size will be left unaltered. Also, the original m_array data should be copied over to the newly allocated one.

,and will have the following **public** member functions:

- **(1) Default Constructor** – will instantiate a new list object with no valid data. Note: What needs to be initialized in this case?
- **(2) Parametrized Constructor** – will instantiate a new list object, which will hold int size number of elements in total, all of them initialized to have the same value as the DataType value parameter. Note: Has to properly handle allocation.
- **(3) Copy Constructor** – will instantiate a new list object which will be a separate copy of the data of the **other** list object which is getting copied. Note: Remember deep and shallow object copies.
- **(4) Destructor** – will destroy the instance of the list object. Note: Any allocated memory pointed-to by m_array has to be deallocated in here.
- **(5) operator=** will assign a new value to the calling list object, which will be an exact copy of the other_arrayList object passed as a parameter. Returns a reference to the calling object to be used for cascading operator= as per standard practice. Note: Think what needs to happen before allocating new memory for the new data to be held by the calling object.

- **(7a,7b) first** returns a pointer to the first (valid) element of m_array, or NULL if it fails.

Note: A reason for failing can be that the list is empty.

- **(8a,8b) last** returns a pointer to the last (valid) element of m_array, or NULL if it fails. Note: A reason for failing can be that the list is empty.
- **(9a,9b) find** returns a pointer to the first (valid) element of m_array, that has the same value as passed parameter DataType target (the equality operator== as overloaded in class DataType should be used to check that). If it fails (it does not find the value it searched for), it returns NULL. Also, it takes in By-Reference a DataType Pointer parameter, and sets it to the Address of the target's predecessor element. If the search fails, or if the target element is found to be the first and has no predecessor, previous should be set to NULL.
  (Extra Functionality (not required for 100pt grade): The method also takes in a constant DataType Pointer named start, which indicates where it should start searching in the list. If this is passed as NULL, it denotes to start searching from the first element. Otherwise, this can be used to start a recursive search in case an element exists twice (otherwise find() only returns the first element's address).
- **(10) insertAfter** first finds the DataType element target, and then inserts after it a new element of the value DataType value. Returns DataType Pointer to the element it inserted (or NULL if it failed). Note: Try to think through what you are doing, and sketch out how it's going to work. Think of all possible cases, e.g. inserting in the middle, at the end, in the start, what happens if m_array already has a size that fits the element, or if it should grow, etc.
  (Extra Functionality (not required for 100pt grade): The method also takes in an int count, signifying how many elements of value DataType value should be inserted. The default behavior is to insert only one. Try to implement inserting multiple at once.
- **(11) insertBefore** first finds the DataType element target, and then inserts before it a new element of the value DataType value. Returns DataType Pointer to the element it inserted (or NULL if it failed). Note: Try to think through what you are doing, and sketch out how it's going to work. Think of all possible cases, e.g. inserting in the middle, at the end, in the start, what happens if m_array already has a size that fits the element, or if it should grow, etc.
  (Extra Functionality (not required for 100pt grade): The method also takes in an int count, signifying how many elements of value DataType value should be inserted. The default behavior is to insert only one. Try to implement inserting multiple at once.
- **(12) removeForward** first finds the DataType element target, and then removes it from the list. Returns a DataType Pointer to the element after the (last) one it removed (if the last it removed was the last in the list, it should return NULL). Note: Try to think through what you are doing, and sketch out how it's going to work. Think of all possible cases, e.g. removing in the middle, the first element, the last element.
  (Extra Functionality (not required for 100pt grade): The method also takes in an int count, signifying how many elements in total after target (with target included) should be removed. The default behavior is to remove only target. Try to implement removing multiple at once. If the number extends more than the list size, it should remove up to the last element.
- **(13) removeBackward** first finds the DataType element target, and then removes it from the list. Returns a DataType Pointer to the element before the (last) one it removed (if the last it removed was the first in the list, it should return m_array). Note: Try to think through what you are doing, and sketch out how it's going to work. Think of all possible cases, e.g. removing in the middle, the first element, the last element.
  (Extra Functionality (not required for 100pt grade): The method also takes in an int count, signifying how many elements in total before target (with target included) should be removed. The default behavior is to remove only target. Try to implement removing

multiple at once. If the number extends more than the list start, it should remove up to the first element.

- **(14a,14b) operator[]** will allow by-reference accessing of a specific DataType at index int position within the allocated m_array Note: Should not care if the position requested is more than the m_array size.
- **(15) size** will return the size of the current list. Note: This is the m_size of m_array and not its m_maxsize, i.e. it is the number of valid DataType entries inside it.
- **(16) empty** will return a bool, true if the list is empty, and false otherwise.
- **(17) clear** will clear the contents of the list, so after its call it will be an empty list object. Note: Does this need to perform memory deallocation?

as well as a friend function:

- **(6) operator<<** will output (to terminal or file depending on the type of ostream& os object passed as a parameter to it) the content of the calling list object. Note: it will do so by traversing the list and calling the insertion operator<< on the valid DataType elements contained within it.

Node-based List:
The following header file extract is used to explain the required specifications for the class (the actual header NodeList.h file is provided and accompanies the Project description):

```
…
class NodeList{
  public:
    NodeList();                                                   //(1)
    NodeList(int size, const DataType& value);                    //(2)
    NodeList(const NodeList& other);                              //(3)
    ~NodeList();                                                  //(4)

    NodeList& operator= (const NodeList& other_nodeList);         //(5)
    friend std::ostream& operator<<(std::ostream& os,             //(6)
                                    const NodeList& nodeList);

    Node* first();                                                //(7a)
    const Node* first() const;                                    //(7b)

    Node* last();                                                 //(8a)
    const Node* last() const;                                     //(8b)

    Node* find(const DataType& target,                            //(9a)
               Node* &previous,
               const Node* const start = NULL);
    const Node* find(const DataType& target,                      //(9b)
                     const Node* &previous,
                     const Node* const start = NULL) const;

    Node* insertAfter(const DataType& target,                     //(10)
                      const DataType& value, int count=1);
    Node* insertBefore(const DataType& target,                    //(11)
                       const DataType& value, int count=1);

    Node* removeForward(const DataType& target,                   //(12)
                        int count=1);
    Node* removeBackward(const DataType& target,                  //(13)
                         int count=1);

    DataType& operator[] (int position);                          //(14a)
    const DataType& operator[] (int position) const;              //(14b)
    int size() const;                                                 //(15)
    bool empty() const;                                               //(16)
    void clear();                                                     //(17)

  private:

    Node *m_head;
};
...
```

The **NodeList** Class will contain the following **private** data members:

- **m_head,** a Node class type Pointer, <u>pointing</u> to the <u>Dynamically Allocated Node</u> object considered as the first element of the list. Note: If the list is empty, m_head should be NULL.

,and will have the following **public** member functions:

- **(1) Default Constructor** – will instantiate a new list object with no data (Nodes). Note: What needs to be initialized in this case?
- **(2) Parametrized Constructor** – will instantiate a new list object, which will hold int size number of elements (Nodes) in total, all of them initialized to hold the same value as the DataType value parameter. Note: Has to properly handle allocation.
- **(3) Copy Constructor** – will instantiate a new list object which will be a separate copy of the data of the **other** list object which is getting copied. Note: Remember deep and shallow object copies.
- **(4) Destructor** – will destroy the instance of the list object. Note: Any allocated memory taken up by elements (Nodes) belonging to the list has to be deallocated in here.
- **(5) operator=** will assign a new value to the calling list object, which will be an exact copy of the other_nodeList object passed as a parameter. Returns a reference to the calling object to be used for cascading operator= as per standard practice. Note: Think what needs to happen before allocating new memory for the new data to be held by the calling object.

- **(7a,7b) first** returns a pointer to the first element (Node) of the list, or NULL if the list is empty
- **(8a,8b) last** returns a pointer to the last element (Node) of the list, or NULL if the list is empty.
- **(9a,9b) find** returns a pointer to the first element (Node) of the list, that holds the same value as passed parameter DataType target (the equality operator== as overloaded in class DataType should be used to check that). If it fails (it does not find the value it searched for inside a Node), it returns NULL. Also, it takes in By-Reference a Node Pointer parameter named previous, and sets it to the Address of the target Node's predecessor element (Node). If the search fails, or if the target element is found within the first Node of the list and has no predecessor, previous should be set to NULL.
  (Extra Functionality (not required for 100pt grade): The method also takes in a constant Node Pointer named start, which indicates where it should start searching in the list. If this is passed as NULL, it denotes to start searching from the first element (Node). Otherwise, this can be used to start a recursive search in case an element exists twice (otherwise find() only returns the first element's (Node's) address).
- **(10) insertAfter** first finds the element (Node) that contains DataType target, and then inserts after it a new element (Node) that holds the value DataType value. Returns a Node Pointer to the element (Node) it inserted (or NULL if it failed). Note: Try to think through what you are doing, and sketch out how it's going to work. Think of all possible cases, e.g. inserting in the middle, at the end, in the start, etc.

(Extra Functionality (not required for 100pt grade): The method also takes in an int count, signifying how many elements (Nodes) that hold the value DataType value should be inserted. The default behavior is to insert only one. Try to implement inserting multiple.

- **(11) insertBefore** first finds the element (Node) that contains DataType target, and then inserts before it a new element (Node) that holds the value DataType value. Returns a Node Pointer to the element (Node) it inserted (or NULL if it failed). Note: Try to think through what you are doing, and sketch out how it's going to work. Think of all possible cases, e.g. inserting in the middle, at the end, in the start, etc.
  (Extra Functionality (not required for 100pt grade): The method also takes in an int count, signifying how many elements (Nodes) that hold the value DataType value should be inserted. The default behavior is to insert only one. Try to implement inserting multiple.
- **(12) removeForward** first finds the element (Node) that contains DataType target, and then removes it from the list. Returns a Node Pointer to the element (Node) after the (last) one it removed (if the last it removed was the last Node in the list, it should return NULL). Note: Try to think through what you are doing, and sketch out how it's going to work. Think of all possible cases, e.g. removing in the middle, the first element, the last element.
  (Extra Functionality (not required for 100pt grade): The method also takes in an int count, signifying how many elements (Nodes) in total after target (with target Node included) should be removed. The default behavior is to remove only the target Node. Try to implement removing multiple at once. If the number extends more than the list final element (Node), it should remove up to that.
- **(13) removeBackward** first finds the element (Node) that contains DataType target, and then removes it from the list. Returns a Node Pointer to the element (Node) after the (last) one it removed (if the last it removed was the first Node in the list, it should return m_head). Note: Try to think through what you are doing, and sketch out how it's going to work. Think of all possible cases, e.g. removing in the middle, the first element, the last element.
  (Extra Functionality (not required for 100pt grade): The method also takes in an int count, signifying how many elements (Nodes) in total before target (with target Node included) should be removed. The default behavior is to remove only the target Node. Try to implement removing multiple at once. If the number extends more than the list first element (Node), it should remove up to that.
- **(14a,14b) operator[]** will allow by-reference accessing of a specific DataType within a Node at an index int position within the list. Note: Since this is not an array-based implementation, the int pos index is a "fake index", just an incremental value such that pos=0 corresponds to the first element (Node) in the list and each subsequent element corresponds to ++pos.
- **(15) size** will return the size of the current list. Note: Since this is not an array-based implementation, the function has to traverse the list to find how many elements (Nodes) lie within it.
- **(16) empty** will return a bool, true if the list is empty (m_head is NULL), and false otherwise.
- **(17) clear** will clear the contents of the list, so after its call it will be an empty list object. Note: Does this need to perform memory deallocation?

as well as a friend function:

- **(6) operator<<** will output (to terminal or file depending on the type of ostream& os object passed as a parameter to it) the content of the calling list object. Note: it will do so by traversing the list and calling the insertion operator<< on the DataType data held within the list's elements (Nodes).

The DataType.h and DataType.cpp files are provided fully implemented. Also, the ArrayList.h and

NodeList.h header files are provided, and NodeList.h provides a class Node implementation in it as well. You will create the necessary ArrayList.cpp and NodeList.cpp source files to implement the range of required functionalities You should also create a source file proj6.cpp which will be a test driver for your classes.

Do not forget to initialize pointers and/or set them to NULL appropriately where needed. Do not forget to perform allocation, dellocation, dellocation-&-reallocation of dynamic memory when needed! Memory accesssing without proper allocation will cause Segmentation Faults. Forgetting to deallocate memory will cause Memory Leaks!

The completed project should have the following properties:

- Written, compiled and tested using Linux.
- It must compile successfully on the department machines using Makefile(s), which will be invoking the g++ compiler. Instructions how to remotely connect to department machines are included in the Projects folder in WebCampus.
- The code must be commented and indented properly.
  Header comments are required on all files and recommended for the rest of the program.
  Descriptions of functions commented properly.
- A one page (minimum) typed sheet documenting your code. This should include the overall purpose of the program, your design, problems (if any), and any changes you would make given more time.

Turn in: Compressed Header & Source files, Makefile(s), and project documentation.

Submission Instructions:

- You will submit your work via WebCampus
- Name your code file proj6.cpp
- If you have header file, name it proj6.h
- If you have class header and source files, name them as the respective class (ArrayList.h ArrayList.cpp NodeList.h NodeList.cpp) This source code structure is not mandatory, but advised.
- Compress your:
  1. Source code
  2. Makefile(s)
  3. Documentation

  Do not include executable
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  Ex: PA6_Smith_John.zip

Late Submission:
A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects will be accepted up to 24 hours late, with 20% penalty.