CSC 2430 Spring 2019 LAB 4 – Inventory Manager (Linked List)

Due: See Canvas

Goal: Your task is to create a linked list data structure for an inventory manager application.

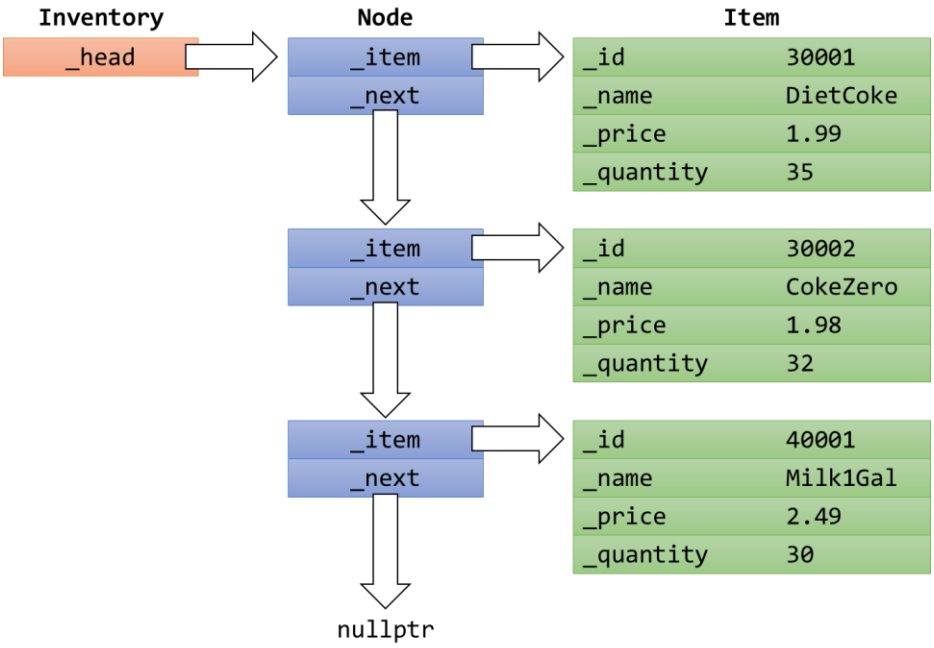
What you learn: This lab assignment will give you further opportunities to practice writing classes and using dynamic memory allocation. This lab will give you the opportunity to implement the Linked List ADT.

Implementation: You are supposed to write three classes, called `Item`, `Node` and `Inventory` respectively.

- `Item` is a plain data class with item id, name, price and quantity information accompanied by getters and setters.
- `Node` is a plain linked list node class with `Item` pointer and next pointer (with getters/setters)
- Inventory is an inventory database class that provides basic linked list operations, as well as find / add / delete / load from file / formatted print functionalities. The majority of implementation will be done in this class.

During or after your writing these classes, you can test your codes with `lab4unittest.cpp`, for the first milestone. Then you can move forward to the actual inventory manager application implementation in `stockmgmt.cpp` for the second and last milestone.

The following diagram illustrates the data structure maintained by your `Inventory` class:



The top left corner of the above figure shows the only private data member of the `Inventory` class:

- `_head`, a pointer to the first node in the linked list. The value is `nullptr` if the list is empty.

Just to the right of these two data members is a column (in blue) of list nodes linked. Each node of the (singly) linked list has a `_next` field, and an `_item` field storing a pointer to an `Item` object. The `Item` object that the `_item` field of each node points to are displayed (in green) to the right. The types of the four fields are int, string, double, and int, respectively.

- For all nodes in the list except the last node, the `next` field of the node at position `pos` will point to the next node. The `_next` field of the last node will point to nullptr.

Note that the order of the items is last-come-first-out. That is, the most recently added node is at the head. We do not manage a tail pointer this time.

Files you will submit:

- `stockmgmt.cpp` contains your **main** function
- `item.h` contains the **Item** class declaration
- `item.cpp` contains the **Item** class member function definitions
- `inventory.h` contains the **Node/Inventory** class declaration
- `inventory.cpp` contains the **Node/Inventory** class member function definitions

The following table explains each of the members that you will need to implement.

Class	Access	Member	Description
Item	Private	`_id : int`	ID
	Private	`_name: string`	Item name
	Private	`_price: double`	Item price
	Private	`_quantity: int`	Quantity of item in stock
	Public	`Item( int, const string&, double, int )`	Creates an item using the values given by the parameters.
	Public	`int getID() const`	Accessor, returns the ID
	Public	`string getName() const`	Accessor, returns the name
	Public	`double getPrice() const`	Accessor, returns the price
	Public	`int getQuantity() const`	Accessor, returns the quantity
	Public	`void setID(int)`	Mutator, sets a new ID
	Public	`void setName(string)`	Mutator, sets a new name
	Public	`void setPrice(double)`	Mutator, sets a new price
	Public	`void setQuantity(int)`	Mutator, sets a new quantity
Node	Private	`_item: Item*`	Item pointer
	Private	`_next: Node*`	Next pointer
	Public	`Node(Item* item)`	Constructor with an item instance's pointer. It assigns the parameter item to _item, then set _next as nullptr.
	Public	`~Node()`	Destructor. It should delete _item.
	Public	`Item* getItem() const`	Accessor, returns the item pointer.
	Public	`Node* getNext() const`	Accessor, returns the next node pointer.
	Public	`void setItem(Item*)`	Mutator, simply sets a new item instance pointer.
	Public	`void setNext(Node*)`	Mutator, simply sets a new next node pointer.
Inventory	Private	`_head: Node*`	Points to the first node in the list. Will be `nullptr` if the list is empty.
	Public	`Inventory()`	Initializes `_head` to `nullptr`.
	Public	`~Inventory()`	Free up (i.e., delete) all nodes in the list.
	Public	`void push_front(Item*)`	Allocate a new Node with the parameter item, then insert it at the front of the list. Update _head to the new Node pointer.
	Public	`Item* front() const`	Returns the first item of the list (i.e., _head's item). If there is no Node, return nullptr.
	Public	`bool pop_front()`	Deletes the first node. Return true if the delete was successful, false otherwise (i.e., no node).

Class	Access	Member	Description
	Public	`Node* findNodeByName( const string &) const`	Traverse the nodes in the list and return if there is a node containing an item having the same name to the string parameter. Return a node pointer if found, nullptr otherwise.
	Public	`Node* findNodeByID( const int) const`	Traverse the nodes in the list and return if there is a node containing an item having the same ID to the integer parameter. Return a node pointer if found, nullptr otherwise.
	Public	~~`Node* findItemByName(`~~ `Item* findItemByName( const string &) const`	Traverse the nodes in the list and return if there is a node containing an item having the same name to the string parameter. Return an item pointer if found, nullptr otherwise. (Hint: you may want to take advantage of findNodeByName function)
	Public	~~`Node* findItemByID(`~~ `Item* findItemByID( const int) const`	Traverse the nodes in the list and return if there is a node containing an item having the same ID to the integer parameter. Return an item pointer if found, nullptr otherwise. (Hint: you may want to take advantage of findNodeByID function)
	Public	`int stockIn(const int, const int)`	Find an item with the given id (first parameter), then increase its quantity by the given value (second parameter). Return the updated quantity. If there is no item with the given id, return -1.
	Public	`int stockOut(const int, const int)`	Find an item with the given id (first parameter), then decrease its quantity by the given value (second parameter). If the given value is greater than the current quantity in stock, update the quantity as zero. Return how many items are withdrawn. (For example, if there was 20 in stock and 30 was asked, the updated quantity is 0 and 20 is returned.)
	Public	`bool addNewItem(Item*)`	Add a new item to the inventory, if there is no item with the same ID. Return true if the addition was successful. False otherwise, that is, there is already an item with the same ID.
	Public	`bool deleteItem( const int)`	Find an item with the given ID, then delete it. **[IMPORTANT] How to delete an item:** (1) Find the node with the item with the given ID (say A) (2) Switch A's item and head's item (3) Delete head (using pop_head)
		`bool isEmpty() const`	Returns `true` if there are no nodes in the list, `false` otherwise. (**Hint**: check if _head is nullptr)
		`int load(istream&)`	Reads records from an istream, and insert them to the inventory. Each line has the four fields for one item record. Per record, an Item object is allocated and added to the inventory. Returns the number of records read. **Note**: There is no corrupt data. No validation needed. **Note** 2: still, you need to check if there is a duplicate id. **Note** 3: Use addNewItem function.

		`void formattedPrint( ostream& out)`	From the head to the tail, print all records in a formatted way to ostream. It should format the output in the following way per record. (make sure to include <iomanip>)
			`out << left << fixed << setprecision(2);`
			`out << setw(6) << [ID];`
			`out << setw(12) << [Name];`
			`out << setw(8) << [Price];`
			`out << setw(4) << [Quantity] << endl;`

Main Program (stockmgmt.cpp).
Your main program will have a menu. The following is a sample execution. (Inputs are in the text boxes)

```
1. Load an Inventory File
2. List all items
3. Search by Name
4. Search by ID
5. Add a New Item
6. Delete Item
7. Exit
1
Input filename:
inven1.txt
5 items loaded.
1. Load an Inventory File
2. List all items
3. Search by Name
4. Search by ID
5. Add a New Item
6. Delete Item
7. Exit
2
ID    Name        Price   Quantity
40002 OrgMilk1G   3.49    20
40001 Milk1Gal    2.49    30
30003 Pepsi       1.49    40
30002 CokeZero    1.98    32
30001 DietCoke    1.99    35
1. Load an Inventory File
2. List all items
3. Search by Name
4. Search by ID
5. Add a New Item
6. Delete Item
7. Exit
3
Input item name:
CokeZero
ID: 30002, Quantity: 32
1. Load an Inventory File
2. List all items
3. Search by Name
4. Search by ID
5. Add a New Item
6. Delete Item
7. Exit
4
Input item ID:
40001
Name: Milk1Gal, Quantity: 30
1. Load an Inventory File
2. List all items
3. Search by Name
4. Search by ID
5. Add a New Item
6. Delete Item
7. Exit
4
```

(continued)

```
Input item ID:
50001
No such item
1. Load an Inventory File
2. List all items
3. Search by Name
4. Search by ID
5. Add a New Item
6. Delete Item
7. Exit
5
Input item ID:
50001
Input item name:
Coffee
Input price:
5.99
Input quantity:
30
Item has been successfully added
1. Load an Inventory File
2. List all items
3. Search by Name
4. Search by ID
5. Add a New Item
6. Delete Item
7. Exit
6
Input item ID:
40002
Item has been successfully deleted
1. Load an Inventory File
2. List all items
3. Search by Name
4. Search by ID
5. Add a New Item
6. Delete Item
7. Exit
2
ID    Name        Price   Quantity
50001 Coffee      5.99    30
40001 Milk1Gal    2.49    30
30003 Pepsi       1.49    40
30002 CokeZero    1.98    32
30001 DietCoke    1.99    35
1. Load an Inventory File
2. List all items
3. Search by Name
4. Search by ID
5. Add a New Item
6. Delete Item
7. Exit
7
```

For a brief description of each choice read the following table:

Menu Option	**Description**
`Load an Inventory File`	Asks the user for the file name containing the employee's information. Along with this Lab definition there are some TXT files containing sample company's information.
`List all items`	Lists all the items stored in the inventory.
`Search by Name`	The user inputs a name, then, using the method `FindItemByName,` the program displays an item information (ID and Quantity only). '**`No such item'`** if not found.
`Search by ID`	The user inputs an ID, then, using the method `FindItemById,` the program displays the item information (Name and Quantity only). '**`No such item'`** if not found.
`Add a New Item`	Request the user for the new item information: ID, name, price and quantity, then add it to the inventory. Check duplicate (by ID). Possible outputs are: • **`Item has been successfully added`** • **`Same ID exists, failed to add`**
`Delete Item`	The user inputs an ID, then, find an item with the ID. Delete if found. Possible outputs are: • **`Item has been successfully deleted`** • **`No such item`**
`Exit`	Exits the program

Program Style

Please refer to all previous Lab documents for appropriate style. Here are some additional style rule:

- Use the `const` modifiers appropriately
- For the data member of the classes remember the prefix _ (underscore)

Submitting your code

Your solution should be contained in the five files: item.h, item.cpp, inventory.h, inventory.cpp, and stockmgmt.cpp. Recommended task flow is as follows:

1) Write `item.h, item.cpp, inventory.h,` and `inventory.cpp`.
    a. First, write all declarations (headers) and function stubs.
    b. Implement Item class → Node → then Inventory class. Follow the order of the table above.
    c. You may want to check your progress and implementation correctness with `lab4unittest.cpp`. This time, unit test has a lot of progress-related outputs, so take advantage of them actively.
2) **[Milestone 1]** Do 1) until you see the congratulation message from the unit test.
3) Submit your `item.h, item.cpp, inventory.h,` and `inventory.cpp` to the autograder part 1.
4) Then write your stockmgmt.cpp main function. Refer to the execution example above.
5) **[Milestone 2]** Once your program generates similar outputs to the execution example, make a fine-grained adjustments of your output format with the zyBooks autograder part 2.
6) Make sure your last submission to the Autograder Part 2 follow the appropriate style.

Academic Integrity

This programming assignment is to be done on **an individual basis**. At the same time, it is understood that learning from your peers is valid and you are encouraged to talk among yourselves about programming in general and current assignments in particular. Keep in mind, however, that each individual student must do the work in order to learn. Hence, the following guidelines are established:

- Feel free to discuss any and all programming assignments but do not allow other students to look at or copy your code. Do not give any student an electronic or printed copy of any program you write for this class.
- Gaining the ability to properly analyze common programming errors is an important experience. Do not deprive a fellow student of his/her opportunity to practice problem solving: control the urge to show them what to do by writing the code for them.
- If you've given the assignment a fair effort and still need help, see the instructor or a lab assistant.
- **If there is any evidence that a program or other written assignment was copied from another student, neither student will receive any credit for it. This rule will be enforced.**
- Protect yourself: Handle throw-away program listings carefully.

Grading

Correctness is essential. Make sure your solution builds as described above and correctly handles the input files provided and the Unit Tests. We will test on other input file(s) as well.

Even if your solution operates correctly, points will be taken off for:

- Not following the design described above
- Not adhering to style guidelines described above
- Using techniques not presented in class
- Programming error not caught by other testing