CS3021
Data Structures and Intermediate Programming
Program Assignment 3

Preparation

Review the materials up to and including *OOP, template,* and *binary search trees*. The primary focus of this assignment is the creation of an abstract data type OrderedList using the C++ template feature and the binary search tree. The implementation is based on the sample class BST.

Problem Statement

An ordered list is a linear sequence of ordered (sorted) items. When you insert a new item into the ordered list, the item is inserted into a position to maintain the order. Here's an example using integer elements. Suppose the ordered list is (in abstract notation; not the actual C++ statement)

```
L = ( 2, 34, 55, 89, 100, 230, 334, 989 )
```

then L.insert(94) (or L->insert(94) depending on how you declared L) will result in

```
L = ( 2, 34, 55, 89, 94, 100, 230, 334, 989 )
```

Your task for this assignment is to implement the OrderedList class that supports update, query, and scanning operations.

Specification

Here's the declaration of the OrderedList class.

```
template <class T>
class OrderedList {

public:

	OrderedList();
	~OrderedList();
```

```
        //--------- Update Operations ---------------------------//
        void insert(T* item);

        void remove(T* item);


        //--------- Access Operations ---------------------------//
        T* first( );
        T* last( );

        T* next( );
        T* prev( );

        T* at(int idx);
        T* search(T* item);

        vector<T*> scan( );

        //----------------- Queries ----------------------------//
        int size( );
        bool isEmpty( );

private:
        //include private data members and functions of your design
};
```

Operations

Please read the specification for each function very, very carefully.

Function	Specification
`void insert(T* item)`	Insert an item into the ordered list in the correct position so the ascending order is maintained after the insertion. For simplicity, assume the item is not a duplicate of any existing items in the ordered list. The current node becomes undefined after a successful insertion; see the section titled Notion of Current.
`void remove(T* item)`	Delete the designated item from the ordered list. If the designated item is not found, do nothing. If the item is an object, then you only need to pass an item with the key field value specified. The current node becomes undefined after a successful deletion; see the section titled Notion of Current.
`T* search(T* item)`	Search for and return the designated item in the ordered list. If the item is not found, return NULL. If the item to search is an object, then you only need to pass an item with the key field value specified. See the provided sample main on how the search is done with a Pet object.
`T* first( )`	Return the first item in ascending order. Return NULL if the ordered list is empty.
`T* last( )`	Return the last item in ascending order. Return NULL if the ordered list is empty.

Function	Specification
`T* next( )`	Return the next item in ascending order. When there is no next item, then return NULL. To retrieve the first three items in the tree, for example, you can call first(), next(), and next(). The the section Notion of Current provides the detail of the next and prev operations.
`T* prev( )`	The inverse operation of next. The section Notion of Current provides the detail of the next and prev operations.
`T* at(int idx)`	Return the item at the idx's position in ascending order. The first item is at position 0, and the last item at position size()-1. Return NULL for any invalid index position.
`vector<T*> scan( )`	Return the items in the ordered list as a vector. The items are returned in ascending order. If the tree is empty, then return an empty vector, i.e., a vector with no elements, not a NULL.
`int size( )`	Return the number of items in the ordered list.
`bool isEmpty( )`	Return true if the ordered list is empty; otherwise, return false.

Notion of Current

In order to implement the access operations first, last, next, and prev, we need to support the notion of the current node. Include a data member in the class to keep track of the current node. Initially, the current node is undefined (NULL). When you call the first or the next function, the first node (in ascending order) becomes the current node. Similarly, when the current node is undefined, and you call the last or the prev function, the last node becomes the current node. After the current node is set, calling the next and prev function updates the current node to the next or previous node (in ascending order). If the current node is the last node, then calling the next function returns NULL, and the current node is not changed. That is, it points to the last node. It does not become NULL. If the current node is the first node, then calling the prev function returns NULL, and the current node is not changed.

The current node gets undefined (NULL) after a successful insert or delete operation. So, for example, if you execute first(), next(), next(), insert(x), next(), you will get the first node back after the last next() call. Notice the current node changes to NULL only after a successful update operation. If the update operation is not successful, then there is no structural change in the ordered list. Thus, the current node needs to be the same as before the (unsuccessful) update operation.

Member Object

To be able to organize and manage objects in the structure, the concrete type (primitives like int and classes like Pet) that replaces the template type T of the OrderedList class must support the comparison operations < and == and the stream output operation <<. The OrderedList class implementation should not assume any more than these three operations.

Requirements

1. All functions must have good performance. For example, you should not implement the at function by first calling the scan function and returning the i'th data in the vector. You should implement the at function without scanning the complete tree. Likewise, make sure your other access functions first, last, next, and prev perform well.
2. There should be absolutely NO input and output statements in the functions. You may place temporary I/O statements in a function for testing/debugging purposes during the development, but you must remove them before the submission.
3. DO NOT CHANGE the public member functions defined in the file OrderedList.h. The instructor's test program will import functions from your module so it is critical that the function signatures are not changed.
4. DO NOT ADD any new public member functions to the OrderedList class. But, you may add as many private data members and private helper functions as you wish.
5. Functions you define in the program should not access any global variables, those define outside of the class. Use of the global symbolic constants is fine.
6. Because you are submitting only a single source file, be sure to include a header comment in the file to record your name, the course number, the assignment number, and any detailed explanation of your functions as appropriate.
7. Format the program in a clean manner. Group statements into logical groups and include a descriptive comment for each group. Group comments do not have to be verbose. One sentence would suffice if the statements in a group are written correctly and precisely. Include enough blank lines between logical groups.
8. Insert one or more blank lines between statements as necessary to make the code more readable and easier to follow. Do not pack statements too densely (e.g., 20 statements with no blank lines between them).
9. Use comments judiciously. Do not include redundant and meaningless comments.

Starter Code

The following source files are provided to you.

Filename	Description
orderedlist.h	The source file for the template class OrderedList. *Do not make any changes to the public segment of the class declaration. The private segment, however, is under your complete control.*
pet.h and pet.cpp	The Pet class header and implementation files. See the example in the main program on how to create and manipulate an ordered list of Pet objects.
prog3tester.cpp	This sample main program illustrates a basic interaction with an ordered list. Review the code here to confirm your understanding of the expected behavior of an ordered list is correct.

Starter code is provided in two format.

1. Eclipse project named Prog3. You can import this project using the menu option Import... and select the option General->Existing Projects into Workspace. The project is provided in the zip format Prog3.zip.
2. Plain source files orderedlist.h, pet.h, pet.cpp, and progetester.cpp. You can create an empty C++ project and then import these files into the project.

There's also a single text file sampleoutput.txt that contains an output from running the provided prog3tester.cpp. This is the output you expect to receive when your OrderedList implementation is correct and complete.

Development Ideas

Review the BST class implementation. This class is the model you follow to complete this assignment. Many of the functions in the OrderedList class are identical to those in the BST class (disregarding the obvious naming differences, e.g., find vs. search, BST vs. OrderedList, etc.). You can reuse the BSTNode class in this assignment.

First, implement the functions that have counterparts in the BST class. Many of these can be implemented with copy-and-paste.

Next, implement the access functions. Implement the first and the last functions. Then, implement the prev and the next functions. Finally, implement the at function. The at function is similar to the inorder traversal of the binary search tree, but in the at function, you must keep track of the order of the node visits. You can implement the at function with a recursive or non-recursive helper function. A non-recursive version requires a stack.

Incremental Development

Here's the suggested order of development (assuming you're starting with the provided OrderedList.h).

1. Implement insert
2. Implement scan
3. Implement search
4. Implement first and last
5. Implement next and prev
6. Implement remove
7. Implement at
8. Implement the destructor

Testing the Functions

As was the case with the previous assignments, create an adequate number of test data and carry out exhaustive tests. Make sure you carry out adequate number of tests after each step and integral tests at several points in the development (e.g., after Step 3, test three functions collectively with a large data set). Coming up with a good set of tests is the best way to construct bug-free code. You need to think very carefully in formulating test routines.

Submission

Submit the source file orderedlist.h, that fully implements the OrderedList class, via the Sakai course website's Program Assignment 3 page. You are only required to submit this source file. On the submission due date, you will be asked to make modifications to the orderedlist.h file and submit the modified version also.