CSCI 2720: Data Structures

Spring 2021

Project 2
Sorted List ADT

Due: Friday, Feb 26, 2021 @11:59:59 PM

In the previous project you have implemented the array representation of the ADT list. In this project you will implement the linked list representation of the Sorted List ADT.

Objectives:

- Practice linked list implementations of the **Sorted List ADT.**
- Practice implementing generic data types.
- Practice using generic data types at the application level, with both primitive data types and user defined data types.

Project Requirements:

1) Use C++ template mechanism to implement a generic **SortedList** ADT represented by a **singly linear linked list**. Your implementation should satisfy the following specification:

   ADT Sorted List Functional Specification (see the attached SortedList.h file):

   1. **SortedList()**
      Function: Initializes the list.
      Precondition: None.
      Postcondition: List is initialized.
   2. **~SortedList();** // class destructor
      Function: Deallocates all dynamically allocated data members.
      Precondition: List is initialized.
   3. **bool isEmpty() const;**
      Function: Determines if the list is empty.
      Precondition: List is initialized.
      Postcondition: The function returns true if the list is empty and false otherwise.
   4. **int getLength() const;**
      Function: Returns number of elements in the list.
      Precondition: List is initialized.
      Postcondition: Returns the number of list elements.
      This function should have O(1) running Time complexity.
   5. **void makeEmpty();**
      Function: Deallocates all dynamically allocated data members. Reinitializes the list to empty state.
      Precondition: None.
      Postcondition: List is empty.

6. **ItemType getAt (int index);**
   Function: Returns the element at specified position (index) in this list.
   Precondition: List is initialized.
   Postcondition: The function returns the element at the specified position in this list, or throws OutOfBound exception if the index is out of range (index < 1 || index > length).
   The index here indicates the order; index 1 should return the 1st item.
7. **void putItem (ItemType newItem);**
   Function: Adds a new element to list. This function should **not allow duplicate keys**.
   Precondition: List is initialized.
   Postconditions: newItem is in the list if newItem's key was not in the list. The DuplicateItem exception is thrown if newItem's keys was found in the list. List order property is conserved.
   This function should call the helper function findItem. (see the private members below)
8. **void deleteItem (ItemType item);**
   Function: Removes an element from the list.
   Precondition: List is initialized.
   Postconditions: item is in not in the list. The DeletingMissingItem exception is thrown if item was not found in the list. List order property is conserved.
   This function should call the helper function findItem. (see the private members below)
9. **void printList (ofstream& outFile);**
   Function: Prints list elements. For example, if the list consists of elements 1, 2 and 3, then the function should output "List Items : 1->2->3->NULL". Also see the provided outFile.txt and soutFile.txt for details.
   Precondition: list is initialized.
   Postcondition: List elements are written to the output file outFile.
10. **void merge (SortedList& otherList) [20 points]**
    Function: Merges the current list and **otherList** into a single sorted list.
    Preconditions: **List** is initialized. **otherList** is initialized and otherList is sorted.
    Postcondition: The elements of **otherList** are merged into the current list (the SortedList object that is invoking this function). Current list becomes the merged list and with **no duplicate keys;** otherList becomes **empty**.

    To test the **merge** command, we will insert elements into two lists then merge them.
    For example, if list1 and list2 are lists of integers, where list1 contains the integers 10->20->30->NULL and list2 contains the integers 1->15->20->NULL
    We will execute these statements in the driver program:
    list1.merge(list2);
    list1.printList(outFile); // The output of this print should be 1->10->15->20->30->NULL

    Your merge function should have a linear Time Complexity O(N+M). Where N is the number of items in the first list, and M is the number of items in the second list.
    To receive full credit, you must merge in place without allocating new nodes.
    *Note: If you call putItem to insert items of otherList into the original List then the complexity of your function is O(N*M), which is more like a quadratic algorithm.*

Private Members:

1. **head:** A pointer to Node that points to the first node in the list (i.e. the head of the list). For this project, you are free to decide whether you want use a dummy head node or not.
2. **length:** integer; stores the number of items in the list.
3. **pair<bool, Node<ItemType>*> findItem(ItemType item);** The function findItem searches for item in the list and returns a pair containing 1) a boolean value indicating whether the item is in the list or not, and 2) a pointer to the predecessor node, which is the node containing the largest value < item's key. If no predecessor exists, then return NULL (Note: if you are using a dummy head node, then there will always be a predecessor).
   This function should be used by your putItem and deleteItem functions.

   If needed, you can add additional private helper functions.

2) Implement a simple Student class

Implement a simple Student class based on the provided Student.h file. You will need to overload some comparison operators including ==, <, and >. It's possible that you may need to overload additional comparison operators depending on how you implemented SortedList. For example, if you have the statement: if (a->info != b->info) in your SortedList.cpp, then you will need to overload the != operator in the Student class. Note that for the 2 stream operators << and >>, there is nothing you need to do. The implementations are given in the driver file.

3) Attached Files:

The following files are attached to this project:

- The driver program **SortedListDr.cpp**, this program will read the data type of elements then will continue reading commands and data from the input file inFile until the command "Quit" is read. We will use our driver program **SortedListDr.cpp** to test your implementation.
  The students list will be sorted by students' IDs.
- **SortedList.h:** contains the specification of the sortedList ADT. You are allowed to add any additional private helper functions, but be sure to not remove or modify any existing functions or data members.
- **Student.h:** contains the specification of the Student class. You are allowed to add any additional helper functions or overload additional operators, but be sure to not remove or modify any existing functions or data members.
- **intcommands.txt** contains sample test cases to test the list of integers.
- **outFile.txt** contains the expected output when executing the driver program with the commands in intcommands.txt.
- **studcommands.txt** contains sample test cases to test the list of student objects.
- **soutFile.txt** contains the expected output when executing the driver program for type student, with the commands in the **studcommands.txt**.

DO NOT CHANGE the driver program when you are ready to submit, we will test with our driver. You can modify the driver program (e.g., add cout statements, comment certain lines out, etc.) for your own debugging/testing purposes, but just be sure to revert back to the ORIGINAL driver program when you submit.

What to Submit?

Name your project directory **project2**. Submit your project2 directory to **csci-2720c** on *odin* (use the following command: *submit project2 csci-2720c*).

If you are off campus, you must use UGA VPN (remote.uga.edu) before you can ssh into *odin.cs.uga.edu* using your MyID and password.

Your project2 directory should contain the following files:

1. SortedListDr.cpp (DO NOT MODIFY)
2. SortedList.h and SortedList.cpp
3. Student.h and Student.cpp
4. README file to tell us how to compile and execute your program. **Include your name and UGA ID# on the top of this file.**
5. The makefile containing at least the *compile*, *run*, and *clean* directives (read the syllabus for details)

Basic grading criteria:

1.	If the project passed all test cases defined in our commands files	100%
2.	If the project did not compile on *odin*	0%
3.	If the project compiled but did not run 30% is given if all required files were submitted and program code completely satisfies all functional requirements.	0% - 30%
4.	If the project runs with wrong output for some test cases, test case rubrics will apply	

See course syllabus for late submission evaluation