Department of Electrical and Computer Engineering
University of Puerto Rico
Mayagüez Campus

ICOM 4035 – Data Structures
Spring 2003

Project #2: Sorted Singly-Linked List Container Class
Due Date: 11:59 PM-March 13, 2003

Objectives

1. Understand the design, implementation and use of a sorted singly- linked lists class container.
2. Gain experience implementing applications using layers of increasing complexity and fairly complex data structures.
3. Gain further experience with object-oriented programming concepts, specially inheritance and virtual methods.

Overview

You will design, implement and test a container class the sorted singly-linked list, to be named here on as SortedSLL. The sorted will be in increasing order. This class will inherit all the public methods from an abstract class called the list container class. Basically, class list indicates the operations that can be performed on a any type of list container class. Meanwhile, SortedSLL implements the operations specified in list for the case of a singly-linked list that is kept sorted at all times in increasing order. You need to assume that the value type stored in the SortedSLL supports the relational operators: >, <, >=, <=, ==, !=.

In addition, you need to implement an iterator class called the SortedSLLIterator. This iterator class will inherit from the abstract class Iterator, which declares the kind of operations to be supported by an iterator.

SortedSLL Specification

The following is the specification of the class SortedSLL.

Private members

- `header` – a pointer to a list node (class list_node). When the list is empty, header is NULL. Otherwise, header points to the first node in the list that is holding data.

Public members

- `value_type` – typedef for the value type of the SortedSLL. Must be equal to the value_type declared in class list_node. Represents the data type for the data in a node.
- `size_type` – typedef for the size type indicating the size of the SortedSLL. Must be equal to the std:size_type value.
- Default constructor – creates a new SortedSLL empty list (i.e. `header` is NULL).
- Copy constructor – creates a new list that is a copy of another SortedSLL, which is passed by constant reference.
- Destructor – removes all elements in the SortedSLL, and make the header equal to NULL. This method cannot create memory leaks.
- Copy Assignment operator – removes all current elements in the SortedSLL, and then makes this SortedSLL equal to a SortedSLL passed by constant reference. This method must check for self-assignment.
- size() – returns the current number of elements in the SortedSLL. This is a virtual function.
- insert() – adds a new element to the SortedSLL, and keeps the increasing sorted order. This function must do an ***in-place*** insert operation. This is a virtual function.
- is_empty() – returns true if this SortedSLL is empty (i.e. header == NULL) or false otherwise. This is a virtual function.
- make_empty() – erases all the elements in the SortedSLL and makes the header equal to NULL. This is a virtual function.
- erase() – removes the first copy it finds from an element obj from the SortedSLL, and keeps the sorted order. NOTE: THIS FUNCTION MUST BE AN IN-PLACE OPERATIONS, MEANING THAT YOU MUST REMOVE THE ELEMENT DIRECTLY FROM THE LIST. SOLUTIONS THAT COPY THE LIST TO ANOTHER, AND SKIP THE ELEMENT TO BE DELETED WILL BE CONSIDERED AS NOT RUNNING. This is a virtual function.
- find() – finds a element obj in the SortedSLL and returns a pointer to an iterator that is positioned on the node where obj is stored. The function will find the first occurrence of the object. The pointer must be a valid pointer to a SortedSLLITerator object. This is a virtual function. The pointer cannot be NULL. If the object is not found in the list, the function must return an empty iterator, which is an iterator whose anchor node points to NULL (see specification of SortedSLLIterator below).

SortedSLLIterator Specification

The following is the specification of the SortedSLLIterator.

Private Members

- `anchor` – pointer to the first node where the iterator is originally positioned when it is first created.
- `current` – pointer to the node where the iterator is currently positioned.
- Constructor – build an iterator from a pointer to a node in the SortedSLL. This node can be NULL, and this represents an empty iterator (one that is associated with no data).

Public Members

- get_data() – returns a reference to the data element in the node where the iterator is currently positioned. This is a virtual function.
- next() – moves the iterator to the next node in the SortedSLL. This operation is only executed if `current` is not NULL. This is a virtual function.
- has_data() – returns true if the pointer `current` different from NULL, meaning that the iterator is positioned on a node with data. This is a virtual function.
- reset() – sets the value of the `current` pointer to be equal to `anchor`. This is a virtual function.

Distribution Files

You can go to the class web page and download a tar file containing all the files related with this project. Just access the link named Projects, and download the sources files associated with the link: *Project #2– Sorted Singly-Linked List*.

You implementation will consist of adding C++ code to implement two modules: SortedSLL.h, and Sorted.cpp.You will receive all the .h files with declaration of the abstract list class and the abstract iterator class.. In addition, you will be provided with a main program that uses the SortedSLL class, and interacts with the user to ask his/her input on the operations to be performed. Finally, you will be given a Makefile with all the commands needed to compile and submit your project.

1. list.h – interface for the singly linked list.
2. iterator.h – interface for the iterator class.
3. SortedSLL.h – interface for the sorted singly linked list container class. YOU MUST WRITE THE DECLARATION OF THE METHODS TO APPEAR IN THIS FILE.
4. SortedSLL.cpp – implementation of the interface for the sorted singly linked list container class. YOU MUST IMPLEMENT THE METHODS TO APPEAR IN THIS FILE.
5. list_test.cpp –test program for the sorted singly-linked list container.
6. Makefile – file with the commands to compile and submit you project.
7. test1.in – test input file 1.
   NOTE: YOU PROGRAM MUST PASS THIS FILE WITHOUT ERRORS IN ORDER TO BE CONSIDERED A RUNNING PROGRAM.
8. test1.out – expected output from test input file 1.
9. test2.in – test input file 2.
10. test2.out – expected output from test input file 2.
11. test3.in – test input file 3.
12. test3.out – expected output file from test input file 3.
13. prof_list_test – professor's version of the list_test program. NOTE: Known to be working correctly.

PROJECT DUE DATE: **11:59 PM – March 13, 2003**.