Programming Assignment 7: Templates and Trees

1. Introduction

This assignment provides students with an opportunity to work with templates and Binary Search Trees (BST). This opportunity is provided in the context of revisiting and creating and alternate implementation of a previous application: the music data base. It is important for students to realize that while this assignment should be familiar in several respects, it is important that they begin with the starter code provided, and limit their use of previous code to reading as a way of refreshing their memory. This advice is provided because the differences from the previous assignment are considerable, and trying to reuse previous code is likely to be no easier than writing new code, and is certain to be less effective in achieving the educational goals of the assignment. That said, a secondary aspect of this assignment is to demonstrate how the strong encapsulation provided by the Object Oriented programming paradigm enables developers to re-implement specific components of an application while minimizing changes to other parts of the software. In the case of this assignment, the basic framework of the assignment remains the same as before, while the data structures keeping track of music tracks and playlists are modified and extended in comparison to the first implementation.

The main purposes of the assignment include: the implementation of Linked List and Binary Search Tree templates, use of the templates to organize storage and access to data appropriate to supporting specific Music DB operations, and extension of the previous Music DB implementation with some new commands. The sample code provided with this assignment is the common core of the previous and the new implementations, so once again we strongly advise students to avoid trying to reuse previous code.

2. Getting Started

The Music DB command "music_db" assumes two input file arguments. The first file contains commands which build the Music DB, while the second contains commands specifying a set of operations on the DB built. The top level of the command is defined in the source file "main.cpp", which is provided as it appears in the reference solution. Students should thus not need to modify it for the final implementation, although they may wish to do so as part of their particular way of iteratively improving the implementation of other assignment components. In the **main( )** routine of the command, the **process_db_definition_file( )** routine processes the set of commands building the Music DB, and the **process_db_cmd_file( )** routine processes the set of commands specifying operations on the Music DB. These functions are defined in the "Utility.cpp" and “Utility.h" files, which contain implementations of other utility routines supporting the application. The **identify_next_command( )** translates the strings in the input file into enumeration values, and includes support for identifying the “play_added_order” and “play_randomized” commands that will be implemented by the students. Similarly, the **parse_cmd_line( )**, **process_add_track( )**, **process_add_playlist( )**, and **process_db_definitions_file( )** are provided in the same form in which they appear in the reference solution, and should not need to be modified by the students. The **process_db_cmd_file( )** routine as provided contains considerable useful code, but also marks the portions where students should add their own code using the Linked List and BST related code with comments containing the string “IMPLEMENT ME”. We request that students leave these comments in place in their final code, as it makes reading the code to determine work done by students both more efficient and more accurate. In general terms, the modifications to this routines consist of method calls to class instances of the Collection, Play_List, and Track classes.

The commands related to creating the data base are described in more detail later in this document, but are similar to those in the previous assignment, with a couple of additions and some modifications.

The following is an example of how the program can be run from the command line:

```
$ ./music_db db_create1 db_cmd1
```

The Make file provided with the starter code implements three tests which are run as a group by the "all_tests" target. The first two tests, "test1" and "test2", capture the output in a file and look for differences from the reference output. The first is quite similar to the reference test in the previous assignment, but has been adapted to the updated set of commands. The second tests error conditions and error output. The third test, "test3" tests the "play" commands that should be implemented by the students. The output of "test3" is not tested against reference output because the nature of the "play_randomized" command is that it should produce different output each time, although ensuring this requires somewhat subtle use of seeds for pseudo-random number generators as demonstrated in the sample code provided with the Algorithmic Complexity assignment.

2.1. Linked List Template

This ADT is implemented in the files "Linked_List_template.h". Note that both the interface and the implementation are specified in the header file as a convenience, even though the test implements the routines in a separate ".cpp" file and then uses the "#include" command to include the ".cpp" contents at the end of the header file. There is no functional difference between the two approaches, so we opt for the approach with fewer files. This ADT as implemented in the reference solution closely follows the semantics of the example from the book, indexing the elements of the list starting at 1 and maintaining a "size" state variable tracking the size of the list. The methods of the ADT are listed only by name, since the details of the return and parameter type declarations are part of the exercise in implementing the Linked List ADT as a template, and deciding what they should be is a good exercise for the students, even when they are not specifically related to template implementation. As an aid to more efficient and accurate grading, we ask that students put the implementations of these routines in the "Linked_List_template.h" file in the order listed here and in the header file. While it may be possible for students to implement the ADT somewhat differently than the reference solution, we urge students to take implementation using the specified set of routines as a useful exercise, and as an aide to more accurate and efficient assignment grading.

Methods for this class should include:

Linked_list( )
This is the constructor of an empty list.

~Linked_list( )
This is the destructor for the ADT, which should include deallocation of any dynamically allocated memory associated with the ADT.

print( )
Print the elements of the list. Note that this must be implemented in a way that makes no assumptions about the specific type used to create an instance of the template, but which prints the elements of the list in the order in which they were added to the list.

is_empty( )
This routine indicates whether the list is empty or not.

get_size( )
Provides the calling context with the size of the list.

insert( )
This routine inserts an element into the list at a specific index.

append( )
This routine appends an element to the end of the list

find ( )
Find a node in the list at a specific index.

remove( )
Remove an element of the list at a specified index.

retrieve( )
Find the node within the list at the specified index and return the instance of the data elements managed by the list held by that node.

2.2. Binary Search Tree Template

This ADT implements a simple Binary Search Tree (BST). This is not a balanced tree, which involves non-trivial additional complications. For the purposes of this assignment, we accept the danger that the tree will be unbalanced due to an unlucky sequence of input values.

Methods for this class should include:

Bst( )
The constructor for the class creating an empty tree.

~Bst( )
Destructor for the class , which calls **deleteNodeItem( ).**

isEmpty( )
This method indicates to the calling context whether the list is empty or not.

insert( )
This method inserts a data element into the tree using a specified key.

search( )
This method searches the tree for a data item whose key matches the specified search key. Note that for purposes of simplifying the assignment, we assume that titles of tracks and playlists will be unique among the items in the data base.

searchNode( )
This method searches the tree for a node whose data element matches the search key. This routine is called by **search( )**.

deleteNodeItem( )

This method deletes a node from the tree which is specified by a node pointer. There are four cases. First, when the node has no children, so the node can be de-allocated and the pointer referring to it set to NULL. The second case is if the node has only a right sub-tree, so the node can be deleted, and the input pointer can be set to point to the right sub-tree. The third case is the complement of the second, where the node has only a left sub-tree, so that the node can be deleted and the input pointer pointer set to the left sub-tree. The fourth case is where the node has both left and right sub-trees. In this case, the data element of the node being deleted can be replaced with the data held by the left-most node of the right sub-tree. This can be done by calling the **processLeftMost( )** routine.

processLeftMost( )

This routine takes reference to a node pointer and a reference to a pointer to a data element as arguments. It recursively finds the leftmost element of the node argument, setting the data element argument to the data of the node, and deleting the node. This routine is called by **deleteNodeITem( )**.

remove( )

This routine removes the node from the tree holding the data element matching the search key. This is a subtle routine which calls **deleteNodeItem( )** to delete the node that matches the search key. The subtle part is that the argument to **deleteNodeItem( )** is the pointer from the parent node to the node being deleted, passed by reference. This made the deletion of the root node of the tree a special case in the reference solution, and meant that we used an iterative loop to step through the tree looking for the node with the right key, but also keeping track of the pointer to the parent node.

print_inorder( )

This routine takes an output stream as an argument and prints the tree according to inorder semantics.

inorder( )

This routine takes a tree root node pointer as an argument and recursively prints the tree according to inorder semantics.

print_preorder( )

This routine takes an output stream as an argument and prints the tree according to preorder semantics.

preorder( )

This routine takes a tree root node pointer as an argument and recursively prints the tree according to preorder semantics.

print_postorder( )

This routine takes an output stream as an argument and prints the tree according to postorder semantics.

postorder( )

This routine takes a tree root node pointer as an argument and recursively prints the tree according to inorder semantics.

2.3. DB Creation and Operation Commands

The commands related to creating the data base are: **add_track** and **add_playlist**. These are unchanged from the previous assignment addressing a music data base. The first file provided as an argument to the command contains a set of these commands, which builds a specific music data base. The commands in the second file, which use the data base built by the commands in the first file, include: **delete_track**, **play_added_order, play_randomized, print_track, print_tracks_title, print_tracks_artist, print playlist, print_playlists,** and **print_collection**. Some of these commands appeared in the previous assignment creating a music data base and some did not. Some that appeared in the previous assignment should be implemented differently in this assignment than in the previous one.

Command Discussions

The semantics of the commands should, mostly, be evident from the names of the commands. Remaining ambiguities should be resolvable by looking at the reference output. A modest amount of thought about and investigation of the semantics of the application should be sufficient for the student to deduce how to implement each command, with a look at the reference output to clear up a few details. However, several students have expressed confusion about how the commands should be implemented using linked lists and binary search trees. In some cases there was a little more ambiguity in the first version of the instructions than the authors thought. Resolving such ambiguities is a big part of the software engineering process.

Working software engineers are expected to deduce the answers to considerably harder questions, to apply known principles to considerably more subtle situations, and to extrapolate basic semantics of an application to deduce features of the application far more subtle than these. Everyone must learn to work up to operating at a professional level but students should be able to start now in at least the limited way required by this assignment.

That said, there were a few low-level ambiguities in the original write-up, most of which could be resolved by looking at the example output. This section provides a brief discussion of each command that must be implemented or completed, and a hint or two of how to use linked lists and binary trees. We hope that this is obvious to most readers and at least seems obvious to all, after reading it.

play_added_order

Play the elements of the named playlist in the order they were added to the playlist. Look in the file “db_cmd3” for an example use of the command. Since the semantics are those of a list, a total order on the elements of the playlist, using a Linked List to keep track of the order in which playlist items are added is obviously reasonable.

play_randomized

Play the elements of the named playlist in random order. Look in the file “db_cmd3” for an example use of the command. Since the Linked List is an ordered set, playing elements randomly can be done by choosing random numbers in the range 1 to playlist-size. From the Complexity assignment students know how to choose random numbers in a given range. Note that the command takes a “repetitions” argument, which specifies the number of times to play a track randomly selected from the list.

print_track

Print the track whose title is specified. Obviously, this requires that tracks be stored in a data structure keyed by title, which permits searching for a track by title. Since the binary search tree has search in the name, it seems like an excellent candidate. This command would thus be supported by a BST keyed by track title which holds a record for all tracks in the collection. Look in the file “db_cmd1” for an example use of the command. For purposes of this assignment, we can assume titles are unique, but if you wish to be more precise, you could use the artist name to distinguish among songs with the same title. Note that the “db_create1” file contains tracks with unique titles.

print_tracks_title

Print all tracks in the collection by title, meaning in alphabetical order by title. Since BSTs can be visited in-order to produce a list of items in sorted order, a BST storing tracks keyed by title seems an excellent approach. Look in the file “db_cmd1” for an example use of the command.

print_tracks_artist

Print all tracks in the collection grouped by artist. Artist names are strings. BSTs visited in-order produce sorted output. A BST holding track records and using the artist name as a key seems an excellent candidate. Look in the file “db_cmd1” for an example use of the command.

print playlist

Print the elements of a playlist. This is probably the one command where a student would have had to look at the example command file “db_cmd1” to deduce what is required. Look in the file “db_cmd1” for example uses of the command. This command has been modified from the previous Music DB assignment to take two arguments: the title of the playlist and the order in which to print it. The three order choices are: title, artist, and added_order. Since BSTs are useful for visiting things in sorted order, BSTs are excellent candidates for organizing the elements of a playlist by title and by artist. Since a linked list inherently stores its elements in an order, a linked list is an excellent candidate to store the tracks in a playlist in the order in which they were stored.

print_playlists

This command should print all the playlists in the collection, in the order of playlist title. As with other commands, a BST keyed by playlist title seems like an excellent way to implement this. We have added specifications to some of the “IMPLEMENT ME” tags where order in which the reference solution did things was not obvious.

print_collection

Print the contents of the collection. This uses the overloaded “<<” operator for the Collection class. It prints all tracks by title and then all tracks by artist. Then it prints the playlists in the order of the playlist title. Each playlist is printed by the **print_inorder()** method of the BST. The data element of each node int he tree is printed by using the “<<” operator, which is overloaded for each class for which the BST is used. The “<<” operator for playlists print the elements of the playlist by title.

delete_track

This method deletes the track, specified by title, from the BST organizing the set of tracks in the collection by title. Note that for purposes of the assignment, you should limit **delete_track** to *only* deleting the entry for the specified title in the BST which organizes the tracks by title. You need not look through the BST organizing the tracks by artist, and you need not find it in any playlist to eliminate it. The **delete_track** command comes last in the “db_cmd1” file and we print the tracks by title right after that to verify the deletion from the one BST. That is enough for this exercise though it would not be enough for a real application.

3. Input & Output

Your program will be expected to handle the data base construction and operation commands as described in Section 2.3. Note that in the Make file provided with the sample code. The “db_create1” file defines a a small music data base. The “db_cmd1” file prints individual tracks, individual play lists, all play lists, all tracks sorted by title, all tracks by artist, and finally the whole collection. The file “output1.correct” provides the output for test1 produced by the reference solution. The assumption is that a correct solution for these capabilities will produce an output file identical to “output1.correct”. If any students believe this assumption is incorrect, then they should contact the instructor to discuss the issue. The file “db_cmd2” tests four simple error conditions, and the file “output2.correct” provides the corresponding output from the reference solution. As with “output1.correct”, the assumption is that a correct implementation should produce matching output. The file “db_cmd3” uses the play_added_order and play_randomized commands. The semantics of randomized play order precludes providing reference output, and the added order is known from the file constructing the data base.

4. Grading Criteria

20%	12 pts	Templates	
	Linked List	3 pts	
	BST	9 pts	
35%	21 pts	BST Implementation	
10%	6 pts	New Commands	
25%	15 pts	Output Correctness	
	BST	6 pts	output1
	LL	3 pts	output1
	Error	3 pts	output2
	New	3 pts	output3
10%	6 pts	Documentation and coding style	

100%	60 pts	Total	