Systems Programing: Project 2: Sorted-List Library

Sangini Shah
Robert Williams, III

The Library

Our library, sorted-list.h, implements a linked list that sort its input upon insertion. The list requires that the user provide a comparator and destroyer function for the data type that the user intends to store, because of this our list can store any data type via void pointers. Each data node, called a SortedListNode, holds the

- Data stored by the user
- A variable ref_count to keep track of how many pointers reference the data
- A deletion flag
- And a reference to the next node in the list

An iterator class is also provided with our library to make accessing data within the list simple. The iterators keep a reference to the next to be accessed node and their original list.

What this code can do

- Store any data type in a linked list, sorted based off the provided comparator function
- Store duplicate values, i.e. separate objects that are considered equal by the comparator function
- Dynamically update the iterators to deal with added and removed values from the initial list.

What this code cannot do

- Store any item that will be freed, twice. That is, you cannot use the same item reference more than once in the list. The result is undefined, but will probably result in a double free error.

The Functions Available

SLCreate(CompareFuncT cf, DestructFuncT df)

This function runs in constant time, O(1), it sets the compare and destroy functions for the list and mallocs the space needed for the list. If nothing is malloced, or we have invalid input, our method returns a null list.

SLDestroy(SortedLIstPtr)

This function frees all space used by the sorted list. The algorithm runs in O(n) time, because each node must be freed.

SLInsert(SortedLIstPtr, void*)

The insertion function takes an object (We use object in terms of, a thing, not an OOP object) of unknown type along with the list one desires to input the object into and performs a compare between the items until the correct position in the list if found. On multiple inputs of the same value, the object is inserted before the original. The algorithm runs in worst case, O(n) time, because one might be inserting something at the end of the list. However, it runs in best

case, O(1). This method also takes time to properly set up a SortedListNodePtr for object storage.

SLRemove(SortedListPtr, void*)

This is the removal function that runs opposite to the insert function. We iterate through the list until we find the first instance of an object in the list. We then, link the previous node to the next node and perform the delete. Now, in the event that there is an iterator present and multiple things point to the node, the node will not be immediately deleted and will instead be flagged for deletion with the flag field. This runs in O(n) time, worst case.

SLCreateIterator(SortedListPtr)

This function runs in O(1) time and creates an Iterator for the input sorted list. The next value the iterator will return will be the head of the sorted list.

SLDestroyIterator(SortedLIstIteratorPtr)

This first calls an update to make sure that the iterator Is on a valid item, then we have some code to remove the current item from existence, in the event that it is the last reference, or the item has already been removed from the list. Finally, we free the Iterator. This runs in worst case O(n) time. This is because of the function call to SLNextItem, which performs the previously mentioned update.

SLGetItem(SortedListIteratorPtr)

This function runs in O(n) time, however in the average case, this runs in O(1) time. O(n) only occurs when the SLUpdateIterator function is called to update the iterator in the event that an item has been removed from the list, so we only return valid entries. This function returns the data at the current position of the iterator. If the end of the iterator has been reached, it returns 0.

SLNextItem(SortedListIteratorPtr)

This function runs in O(n) time, however in the average case, this runs in O(1) time. O(n) only occurs when the SLUpdateIterator function is called to update the iterator in the event that an item has been removed from the list, so we only return valid entries. This function increments the iterator first, then returns the data. If the end of the iterator has been reached, it returns NULL.

SLSearch(SortedLIstPrt, void*)

This function runs in O(n) time. It searches the list for the next smallest data object to set as the next item to be returned from the iterator.

SLUpdateIterator(SortedLIstIteratorPtr)

This function calls SLSearch, making it run in O(n) time. This however, also will free data items that are flagged fro deletion with no more pointers to them.