Ethan Berroa

Rushit Sanghrajka

CS 214: Programming Assignment 2

Sorted List

In this programming assignment, we implement a sorted list. This sorted list stores objects in descending order, and can be used for any type of objects. We provide the objects as void pointers to the sorted list, along with a comparator and destructor function, and the implementation will allow us to traverse and access all elements in descending order.

Hence, our implementation contains of the SortedList itself, along with the SortedListIterator.

Dealing with modifications in the list

If a node in the sorted list is modified while there is an iterator pointing on the list, our implementation allows modifying the contents of the node. However, if that element is removed from the sorted list, or the sorted list itself is destroyed while an iterator still exists and points to a node, we unlink the particular node and isolate it from the sorted list. In this manner, the user can still access the contents of the data in the node via the iterator, however, the node has been disconnected from the sorted list itself. This helps in preventing the user from losing out any important data until they themselves destroy the iterator.

Run time analysis of the implementation

1. SortedListPtr SLCreate (CompareFuncT cf, DestructFuncT df);

   In this function, we simply create an empty sorted list structure. This has a constant run time complexity of O(1).

2. void SLDestroy(SortedListPtr list);

   Destroying the SortedList will have the complexity of O(n), where n is the number of elements in the list. Since this is a dynamic storage, it uses linked lists and has linear complexity.

3. int SLInsert(SortedListPtr list, void *newObj);

   SLInsert inserts elements in descending order. The complexity for this function would be O(n) in worst case, and O(1) in best case.

4. int SLRemove(SortedListPtr list, void *newObj);

   SLRemove first looks for an element, and then removes it. The time complexity for this is linear once again as it travels till the element is found or if it reaches a node with a compare value lesser than the element to be deleted. Time complexity = O(n).

5. SortedListIteratorPtr SLCreateIterator (SortedListPtr list);

   The time complexity for creating an iterator is simply O(1). It just points to the head of the linked list.

6. void SLDestroyIterator(SortedListIteratorPtr iter);

   SLDestroyIterator has a time complexity of O(1). It simply deletes the iterator.

7. void * SLGetItem(SortedListIteratorPtr iter);

SLGetItem gets the data for the current node that the iterator points to. Complexity is O(1).

8. void * SLNextItem(SortedListIteratorPtr iter);

SLNextItem traverses to the next element in the sorted list and returns the data for that node. The complexity for this function will be O(1).

Memory Analysis of the Implementation:

1. SortedListPtr SLCreate (CompareFuncT cf, DestructFuncT df);

We assign 3 pointers: list, comparef, and destructf. Head is assigned to NULL. Additonally, we must factor in our two parameters: the function pointers cf and df. Together, they take up 4 bytes * 5, which is 20 bytes of data. As far as running time goes, we are simply performing 5 "basic computer steps" that take constant time. Thus, the function is O(5) = O(1)

2. void SLDestroy(SortedListPtr list);

In this function, we use two Node pointers: tmp and p. Using them both, we traverse through the SortedList, freeing up all it's Nodes and associated data. Also, our parameter, list, is a 4 byte SortedListPtr. Thus, we use no more the 4 bytes * 3, which is 12 bytes of data to accomplish this function. The running time of this function is dependent on the length of the SortedList, due to the while(p!=NULL) statement. Thus our running time is linearly dependent on our input, or O(n)

3. int SLInsert(SortedListPtr list, void *newObj);

Analysis of Memory Usage: Our parameters, list and newObj, take up 8 bytes of data since they are both pointers. Since this is our insert function, we have to create a Node object and associated pointers. The Node pointer takes up 4 bytes, while the struct itself takes up another 12 bytes with: data, refctr, and next. If we're adding the Node to the end of the list, next will point to NULL and the struct will only take 8 bytes. In total, we're managing between 24 to 20 bytes of data. With regards to running time, our main point of interest is the while loop: while(p!=NULL). We are going through the list, one by one, and comparing our newObj to the nodes already inserted. At worst, we add our new Object to the very end of our n long list. Thus, the running time is O(n)

4. int SLRemove(SortedListPtr list, void *NewObj);

Analysis of Memory Usage: First off, our two parameters, list and newObj, take up 8 bytes of data. We use two Node pointers, tmp and prev, to traverse the list while comparing each Node to the target to be deleted, that's another 8 bytes. We also utilize pointers such as: tmp->data, list->comparef, list->destructf, tmp->next, and prev->next. That's another 20 bytes. In total, we manipulate 36 bytes of data. When finding the Big O of this function, our main point of interest is again a while loop set to traverse the whole list at worst, while(tmp!=NULL). Thus, the running time is O(n) once again.

5. SortedListIteratorPtr SLCreateIterator (SortedListPtr list);

Analysis of Memory Usage: Our SortedListPtr parameter is a 4 byte pointer. We create a SortedListerIterator struct, and a pointer to it, "helper". We then initialize all the struct values using pointers in the list struct. After all is said and done, we use 32 bytes of data. Running time is O(1) because all we're doing in this function is the constant time steps of initializing some data.

6. void SLDestroyIterator(SortedListIteratorPtr iter)

Analysis of Memory Usage: The initial paramter, iter, is a 4 byte pointer to an Iterator struct. We utilize said structure's “current” node pointer, and said node pointers refctr value. Both are an additional 4 bytes. We also access the function pointer to the destruct function, for another 4 bytes. In total, 16 bytes are utilized.

7. void *SLGetItem(SortedListIteratorPtr iter)

Analysis of Memory Usage: The parameter, iter, is a 4 byte pointer to an Iterator struct. Using this, we access the Ierator structure's 4 byte “current” node pointer, and it's associated data. Thus, we access 12 bytes of data. Or 8 byes, if we find out our current node pointer points to NULL.

8. void *SLNextItem(SortedListIteratorPtr iter)

Once again, our paramater is a 4 bye Iterator structure pointer. Using this, we access the Iterator structure's “current” pointer, which accesses the whole of the associated Node structure (the data, refctr, and next pointer; each 4 bytes.). All in all, we use 20 bytes of data.