CS 214 Assignment 2: Sorted List

Kevin Sung and Joyce Wang

October 3, 2013

1 Implementation

We used a singly-linked list to implement the sorted list. Nodes in the list have a count of how many pointers are pointing to it, to ensure that no pointers are left dangling once memory is freed. Therefore, the memory for a node is not necessarily freed when it is removed from the list; if there is an iterator pointing to it, then it will simply be detached and removed when the iterator is destroyed. Similarly, when a list is destroyed, not all the nodes will necessarily be freed.

When an iterator is initialized on a list, it points to the node that is the head of the list at that moment. Inserting nodes to the list does not change which node the iterator points to, and neither does removing items that the iterator is not pointing to. If the item that the iterator is pointing to is removed, then the next time the iterator is called, it returns the item that was after the removed item in the list.

2 Analysis of running time and memory usage

2.1 SLCreate

This function allocates a constant amount of space for a SortedList struct. The input is a pointer to a comparator function, which has no effect on the running time or memory usage of the function. Thus, the running time and memory usage of this function are both $O(1)$.

2.2 SLCreateIterator

This function allocates a constant amount of space for a SortedListIterator struct. The input is a pointer to a list, which has no effect on the running time or memory usage of the function. Thus, the running time and memory usage of this function are both $O(1)$.

2.3 SLInsert

Running time: The inputs to this function are a pointer to a list and a pointer to an object to be inserted. Let n denote the size of the input list. We will count the number of calls to the comparator function of the list. SLInsert works by comparing the object to be inserted to each element of the list, starting at the head, and stopping when it reaches an item that is less than or equal to it. It then inserts the object before this item. In the worst case, the object to be inserted is smaller than all of the items already in the list, and SLInsert makes n comparisons. Therefore, the running time is $O(n)$.
Memory usage: This function uses no memory other than creating a constant number of variables, in addition to allocating space for the new object to be inserted. Thus, the memory usage is $O(1)$.

2.4 SLRemove

Running time: The inputs to this function are a pointer to a list and a pointer to an object to be inserted. Let n denote the size of the input list. We will count the number of calls to the comparator function of the list. SLRemove works by comparing the object to each element of the list, and stopping if it reaches an item that is equal to it, in which case it removes it from the list. If the item is not found, the function stops after it reaches the end of the list. In the worst case, the object is not in the list, and n comparisons are made. Therefore, the running time is $O(n)$.
Memory usage: This function uses no memory other than creating a constant number of variables. Thus, the memory usage is $O(1)$.

2.5 SLNextItem

Running time: The input to this function is a pointer to an iterator, and the function returns the iterator's next item. The only time when the function would not just return the item that the iterator is pointing to is if it is pointing to an item that has been removed from the list. The iterator may be pointing to a chain of items which have been removed, in which case it would have to advance past all of them until it gets to the list. In addition, it may have to free the deleted items that it passes. Thus, if we count the number of times the function frees a node, then the running time is $O(m)$, where m is the size of the chain of deleted nodes that the iterator is pointing to.
Memory usage: This function uses no memory other than creating a constant number of variables. Thus, the memory usage is $O(1)$.

2.6 SLDestroy

Running time: The input to this function is a pointer to a list. Let n denote the size of this list. This function goes through the list, starting at the head, and frees each node if it is not being pointed to by an iterator or chain of deleted

nodes. We will count the number of times the function frees a node. In the worst case, every node must be freed, so n calls are made. Thus, the running time of this function is $O(n)$.
Memory usage: This function uses no memory other than creating a constant number of variables. Thus, the memory usage is $O(1)$.

2.7 SLDestroyIterator

Running time: The input to this function is a pointer to an iterator. The only time when this function would not simply free the iterator is if the iterator is pointing to a chain of deleted nodes. In this case, the function first advances through the chain, possibly freeing each node it encounters, and then frees the iterator. Thus, if we count the number of times the function frees a node, then the running time is $O(m)$, where m is the size of the chain of deleted nodes that the iterator is pointing to.
Memory usage: This function uses no memory other than creating a constant number of variables. Thus, the memory usage is $O(1)$.