Amar Bakir
CS 214
Project Assignment 2: Sorted-list
Friday, October 3rd, 2014

SLCreate:
Running time: O(1)
Memory usage: O(1)
Allocates memory for the new list and initializes the values of the sorted-list struct with the proper values. This is all done in constant time and with constant memory, independent of any input.

SLDestroy:
Running time: O(n)
Memory usage: O(1)
This function must run through the list it receives and free all memory. This means that the function must access every node and call the destruct function on the object data as well as free the node. Since we are not concerned with coefficients for Big-O, this running time analysis shows that the running time is linearly related to the input, and so O(n). The memory usage is simply that of the temporary nodes needed to perform the operation, and so it is constant at O(1).

SLInsert:
Running time: O(n)
Memory usage: O(1)
SLInsert requires that, in the worst case, you run through the entire list looking for a spot to insert an item. That item would have to be smaller than any other item in the list, so that at the very end of the sorted-list, sorted in descending order, you insert the item. This implies a linear correlation of time with input, and so the running time is O(n). The only memory used it that dynamically allocated for the new Node. This remains constant regardless of the input, and so is constant.

SLRemove:
Running time: O(n)
Memory usage: O(1)
Similarly to SLInsert, in the worst case, SLRemove must traverse the entire list structure, which happens if the last item is the item to be removed or if the item to be removed doesn't exist. Regardless, the function must travel the length of the number of nodes. Technically it only has to hit the non-removed nodes, but since this is the worst case we can assume all the nodes are in the list and will be accessed. This is a linear correlation regarding running time and input. O(n) is the running time. Nothing is being dynamically allocated, and the temporary nodes needed for the traverse and remove operation require constant memory (and will be removed at the end of the function call anyways).

SLCreateIterator:
Running time: O(1)
Memory usage: O(1)
SLCreateIterator creates the iterator and sets its values based on the list passed to it. This takes constant time, and will either pass or fail. This function also allocates memory for the iterator, but this is also constant as there is only one thing to allocate. In the worst case the function succeeds and memory is created for the iterator, and its values set, regardless of how large the list is. In conclusion, this function runs in constant time and with constant memory usage.

SLDestroyIterator:
Running time: O(n)
Memory usage: O(1)
SLDestroyIterator destroys the current iterator, but also accounts for the cascading effect that might occur if the iterator pointed to a long line of removed (but not deleted) nodes. The memory usage is constant, as we always need two nodes to perform the delete operations and keep track of where we are in the list. In the worst case, all the nodes have been removed but not deleted, and so destroying the iterator ends up deleting all the nodes in the list. Each deletion and movement along the nodes takes constant time, and since this is Big-O, I will be ignoring the constants. In a lost of n nodes, in which all n have been removed (worst case), then destroying the iterator will end up performing n deletions. This takes O(n)

SLGetItem:
Running time: O(1)
Memory usage: O(1)
SLGetItem only returns the object to which the current node, pointed to by the iterator, points to. If the iterator points to null, then it returns 0. In both these cases the running time is constant and independent of the size of the input to the program. It also doesn't use any extra memory as nothing is being allocated and no new variables are being declared.

SLNextItem:
Running time: O(1)
Memory usage: O(1)
Similarly to SLGetItem, SLNextItem return the content of a node, but of the next node. It also moves the iterator up to this node before it returns its value. This is obviously done in constant time. In the worst case, moving up the iterator causes the pervious node to be destroyed, which takes constant time. If the iterator points to NULL or goes past the list to NULL, NULL is returned. Again nothing is being allocated except for temporary variables to perform the moving of the iterator and possibly the deleting of the previous node. This uses up constant space as well.