Name: Gregory Grzymski
Assignment: Sorted List Assignment

The primary data structure for this assignment was a Linked List, comprised of struct Nodes. Each Node contained a void *data field, an integer reference count to how many Nodes are pointing at that specific Node, and a pointer to the next Node in the linked list. The Linked List was sorted in descending order.

There were several reference field counts tracked by this program:

ref_count=0;   // Item needs to be freed and removed
ref_count=1; // Item is being pointed by another node, which also includes head initialization.
ref_count=2 and above; // Item is also being pointed by iterator/iterators

My struct SortetListIteratorPtr comprised of a struct Node that initially points to the head of the Sorted List.

## Main.c

I have provided 14 test cases that are run by an interface command prompt. The user should input 1-14 in order to see the different test case outputs. Inputs are provided within the testplan.txt file.

### SLCreate

This creates the memory for an empty Linked List. Memory is allocated for the Sorted List head field, and the Sorted Lists's CompareFunction is specified. This is a direct operation with minimal comparison work.

Big(O):  O(1)

### SLDestroy

This frees up all the memory that was allocated during the run of the program. Each Node was individually freed, and then the sorted list was also freed. This was also a direct operation.

Big(O): O(1)

### SLInsert

This function handled the insertion of a new element into the already created sorted list. Duplicate items were returned with 0 as they weren't allowed in the list. There were several cases writen for:

1. An empty head field.
2. When one element exists in the list, how is the greater/lesser element handled.
3. When newObj is less than everything on the list
4. When newObj is greater than everything on the list.
5. When newObj is greater than some elements, but also less than one.

The worst-case situation for this function is when the whole list needs to be traversed in order to add the element at the end of the list.

Big O: O(n)

**SLRemove**

This function handled the removal of a specific Node, depending on certain factors. There were several cases created:

1. Check if the object exists in the list.
2. Check if the list is empty.
3. Check if the head node needs to be removed
4. Check if the Node is the middle or tail element.

Another condition to check for is whether the item to be removed is pointed by the iterator or not. If it is, then a count is decremented by 1 and the free operation is handled by SLNextItem. If it isn't, then the Node is freed in this function.

Big O: O(n). Worst case is when the whole list needs to be traversed to find the item to be deleted.

**SLCreateIterator**

This creates the Iterator pointer to the already created Sorted Linked List. Memory is allocated by the pointer, while also incrementing the reference count of the Node that is being pointed to by the iterator to 2. The Iterator was initialized to point at the head node of the sorted list.

Big O: O(1). This was a direct operation.

**SLDestroyIterator**

This destroys the Iterator pointer of the sorted list. Memory is freed, while also Decrementing the reference count of the Node that was currently pointed by the Iterator.

Big O: O(1). This was a direct operation.

**<u>SLNextItem</u>**

This function was responsible for the returning the next data object that was currently pointed by the iterator. Before the function finishes its call, the iterator is already pointing to the next object in the sorted list.
The reference count was also decremented to indicate the repositioning of the iterator. If the reference count was equal to 0, this meant that SLRemove was called on that specific Node and that this function was responsible for freeing the Node. If the count was equal to 1, then the iterator just moves to the next item in the linked list. And if the next node was not equal to NULL, its reference count was set equal to 2.

Big O: O(1). This was a direct operation in returning a data object, while iterating to the next element on the list.