2nd Assignment: The Sorted List

By **Thanh Ngo** and **Tiago Costa**

1. **NodePtr nodeDestroy(NodePtr node, DestructFuncT destruct)**
nodeDestroy() destroys the current node and recursively destroys the following linked nodes from this node which have the number of pointers pointing to it: 1. This function also returns the node where it stops the destroying process.
The time complexity of this function is: O(n).
As this function runs recursively, without any loop. The space complexity is O(n).

2. **NodePtr nodeCreate(void* data)**
nodeCreate() creates a new node containing the data.
The time complexity of this function is: O(1).
The space complexity is: O(1).

3. **SortedListPtr SLCreate(CompareFuncT cf, DestructFuncT df)**
SLCreate() creates one new sorted list. There is no loop in this function. The time complexity of this is: O(1) and space complexity is O(1).

4. **void SLDestroy(SortedListPtr list)**
SLDestroy destroys the sorted list by going through the nodes from the head and destroys those nodes and skips the node that has 2 or more pointers point to it. What SLDestroy actually does is to repeatedly call SLRemove(first->data) until the first node is null. Hence, its time complexity is O(n) and its space complexity is O(1).

5. **int SLInsert(SortedListPtr list, void* newObj)**
SLInsert inserts a new node into the list. It goes through the linked list and finds a suitable location by comparing each node's data to the newObj data. Its time complexity is O(n) as it goes through each node. Its space complexity is O(1), since the function uses the same variables as it goes through the nodes.

6. **SLRemove(SortedListPtr list, void* newObj)**
SLRemove traverses through the sorted list and searches for the node containing the newObj data. If the data is found then SLRemove() try to eliminate the node containing that data using nodeDestroy(). However, since we try to destroy the expected node, the next node is going to be linked with the previous node of the deleted one. Hence, the number of pointers point to the next one is greater or equal to 2 (the deleted one and the previous one both link to it). As a result, the recursive nodeDestroy() only runs once in this case. nodeDestroy()'s complexity in space and time is O(1).
SLRemove() time complexity: O(n) + O(1) = O(n).
SLRemove() space complexity O(1) + O(1) = O(1). (The first Big Oh notation O(1) is because all variables created are shared equally among nodes when traversing.)

7. **SortedListIteratorPtr SLCreateIterator(SortedListPtr list)**
SLCreateIterator() creates a new iterator and assigns the tracking node pointer with the head node of the linked list. Its time and space complexity is, thereby, O(1).

8. **void* SLNextItem(SortedListIteratorPtr tier)**

SLNextItem will get to the next node if it finds the current node has more than 1 pointers point to it. Otherwise, this function will try to eliminate the linked nodes following that until it reaches a node has at least 2 pointers point to. In another word, it will traverse through the list of node, until it found a node that has not been yet deleted from the linked list or there is some other pointers point to it. SLNextItem has no loop and it makes a call to nodeDestroy, hence its time and space complexity is O(n).

9. **void* SLGetItem(SortedListIteratorPtr iter)**
SLGetItem returns the data inside the current node located in the Iterator struct. Hence, its time and space complexity is O(1).

10. **void SLDestroyIterator(SortedListIterator tier)**
SLDestroyIterator makes a call to nodeDestroy() whose complexity, both in time and space, is O(n). Hence, this function's time and space complexity is O(n).