INDEXER - Christopher Diehl and Sandeep C Mattappali

For this project I have decided to use a singly linked list to start out with. The reason I am using a singly linked list is because the grab time from the list will be O(1) although worst case insert is O(n). We have a node struct which is used to manage the linked list.

For our SLCreate we create a Node with the new object if the firstNode is already created. We then insert the newly created Node in one of three positions. The front, the end, or some undetermined place in the middle. We do not allow duplicate entries in the list. In the case of a duplicate entry, we return a 0 and free the created Node. We also check if the list and the data we are supposed to add is not equal to 0.

SLDestroy destroys all nodes that have no iterators pointing to them. If there is one or more iterators pointing to the node, SLDestroy decrements the variable in the node which determines the number of iterators pointing to said node. When a node is created in SLInsert, the number of iterators pointing to the node == 0. When we destroy an iterator, or move to the next node from the iterator, we check if the number of iterators pointing to the node == 0, if so then the node is freed from the iterator.

Iterators : We keep track of number of iterators pointing to each node with number of iterators. If you delete a node that the iterator is on, the number of iterators gets decremented by 1. So the iterator is tasked with free the node it is on if the number of iterators == 0. The next node then has it's number of iterators ++.

SLRemove posed a challenge because you had to make sure any nodes iterators are on don't get deleted. So if the number of iterators pointing to a node > 0, then we decrement the number of iterators value, so the iterators know if they need to destroy the node. Then we increment the number of iterators value in the next node, so that will not get removed, ie our iterator won't run off the list. When the user calls SLRemove we do not destroy the value in the node, because the .h file says to not alter the data item pointed to by newObj, so we do not free it. If you would like to edit this, or free the object using SLRemove, you can uncomment the line (180 ish) that destroys the newObj.

ASSIGNMENT 1 Conceptual overview
-Main, used to test things
SORTED-LIST.H
-Need to implement SortedList struct

SORTED-LIST.C
-create and destroy sortedlist
-remove and insert into list
-iterate through list

SORTED LIST IMPLEMENTATION
-store void * pointer to some value, put into list
-Basically make sorted list into node
-Malloc new sortedList, then point to it
-Basically when inserting, create new sorted list, malloc, then point next to it
-Iterator basically a pointer to the next sorted list item.

VOID *
void *ptr = &insertObj

ITERATOR
-Get item, get next item
-SLIterator is a struct in of itself
-create and destroy…..
-Check for null then return value
-The iterator should continue to be correct, even in the face of list modification
-Could copy the sortedList in the iterator
-Or check if node is being referenced
-Can check if Node is being pointed to by adding numOfReferences field to node
-if numOfReferences > 1(ie there is an iterator)
    -delete from list by making node.prevNode.nextNode = node.nextNode
    -then iterator is in charge of free node
    -could make node.prevNode ==0,
Num of references ++ only when iterator

Tests:
We tested it with ints to make sure everything worked then progressed to a struct with a char * value which was used to compare. Then we tested with a char *. We used a linux script to just run through everything, all of our tests will be included with our submission.

Algorithmic analysis :
Worst Case Analsysis
SLCreate O(1)
SLDestroy O(N)
    We have to go through each element in the linked list and remove the item.
SLInsert O(N)
    We have to find the correct position to place the object and then insert it there
SLRemove O(N)
    We have to first find the object to be deleted and we might have to go through the entire list to do this
SLCreateInterator O(1)
SLDestroyInterator O(1)

SLNextItem O(1)
SLGetItem O(1)

Space Requirements :

The space required will be proportional to the size of the list.

Tradeoff :

We debated whether to go with a tree structure as this will ensure faster finding of the element, we decided not to do this because of the additional complexity and space over head.

Ambiguities :

SLRemove :

We do not delete the newObj as it was stated not to modify the newObj.

If an iterator is pointing to the element to be removed , then the element is not removed, but it is delinked from the sorted list such that the current iterator will have access to the element , but a new iterator pointing to the beginning of the sorted list will not have access to it.

SLGetItem:

On calling this function we do not advance the iterator to the next element . So calling getitem repreatedly will return the same element.

Even if the iterator is pointing at a orphaned node , SLGetItem will not delete this , in order to delete it , SLNextItem has to be called.

Even if no other “ thing “ points to the node , the element will not be deleted.

SLNextItem:

This function advances the iterator to the next item. If it is the case that no other “ thing “ points to the element we are moving forward from , then the nextItem function will delete this node , including the void * data pointed to by the element. The reason we made this decision was to take of the scenario where we destroy a Sorted List called SLDestroy , but since the iterator is point to an element , SLDestory will not delete that node , and if SLNextItem does not delete this node , then this will lead to a memory leak. In order to prevent the delete of orphan nodes SLNextItem exhibits this behavior.