Harry Stern

CS 214 Assignment 2

Big O analysis

Analyses below are assuming malloc are O(1), and n is the number of nodes in the list
SLCreate: We just malloc and assign some variables, so it's $O(1)$
SLDestroy: We dealloc each item in the list, and NodeDestroy is $O(1)$, so it's $O(n)$
SLInsert: In the worse case we have to insert a value smaller than all the other values in the list. If our comparison function is $O(f(n))$ then the entire function is $O(nf(n))$
SLRemove: Again in the worst case we may have to remove a value smaller than all the other values, so as in SLInsert it's $O(nf(n))$
SLCreateIterator: We do a few mallocs and move some pointers, all of which are $O(1)$, so it's $O(1)$.
SLDestroyIterator: This may actually be $O(n)$ because of the NodeDestroy; see NodeDestroy analysis.
SLGetItem: We just dereference two pointers. $O(1)$
SLNextItem: Again we dereference some pointers and move one. $O(1)$
NodeCreate: We are doing some allocations. $O(1)$
NodeLink: We move a pointer and do an increment. $O(1)$
NodeUnlink and NodeDestroy: This is where it gets slightly complicated. When we unlink a node, we decrement the former next node's refcount. If it reaches zero, we destroy that node. In NodeDestroy, if that node has a next, we do an Unlink on node-¿next, which may cause a destroy on that node. Therefore in the worst case if we remove from the list in reverse order, starting at the smallest until we hit the head, the last NodeUnlink will cause all the reference counts to go to 0 in succession. Therefore, the worst-case runtime is $O(n)$ for both NodeUnlink and NodeDestroy. This also technically applies to NodeUnsetHead, though it never gets called without calling NodeSetHead on something else first.
NodeSetHead: This is just an increment so it's $O(1)$.

Correction in sample main.c

In the sample main.c provided, the compareInt function does not adhere to the CompareFuncT specification in the pdf and source code. It returns (int1-int2) which may not be in the set $\{-1, 0, 1\}$. I fixed this in my own main.c.

Temporary nodes with no data

I had some problems with node insertion. I wrote functions NodeLink and NodeUnlink to set the node->next and do reference counting, but in NodeUnlink I would also check whether the refcount dropped to zero. In that case I destroy the node. However, in an insert, you have to unlink one thing and store it temporarily and then link it back up. This caused the deletion of the last node because its refcount temporarily dropped to zero. I solved this by creating a temporary node and linking its next to the one I was about to unlink. I then free the temporary node. Incidentally, this is also how I implemented iterators. I created a temporary node whose next points to the current node of the iteration. This prevents nodes from being destroyed if they are removed from the list while an iterator is pointing to it.

Node deletion

In Professor Russell's clarification on sakai, he says "Any nodes whose reference counts goes to zero gets deleted." I took this to mean that the node gets deallocated at this point, which includes calling the DestructFuncT df on the void *data as well. This causes problems if you try to, eg, change the value of a pointer, so I assume that it never happens in the test code and all pointers are essentially const. Otherwise, you could just do the following: You have a pointer to an int x with value 3, which gets inserted into the middle of the list somewhere. You then do *x = 1000, and that changes the value of the object inside the node since it's the same pointer. Then your list is out of order.

Freeing everything

My code appears to work because when I run my testcases (main.c) in valgrind, it claims everything is freed.