CS214 Programming Assignment 2
Vincent Xie, Jacob Rizer

Sorted list

Our implementation of the sorted list uses a linked list. When we add to the list, we create new nodes, and when we delete from the list, we remove nodes. I will begin to explain our process using the structs that we used.

struct Node
- data (holds the data that the node holds)
- prev (holds reference to previous node)
- next (holds reference to next node)
- references (holds number of references to the node)
- removed (holds 1 if node is slated for deletion, else 0)

struct SortedList
- compare (holds the compare function that allows for two items to be compared)
- destroy (holds the destroy function that deallocates the memory used for data)
- head (holds the reference to the first node in the list)

struct SortedListIterator
- current (holds the reference to the node that the iterator is pointing to)
- destroy (holds the destroy function that deallocates memory used for the data)
- list (holds a reference to the list, needed for function correctIterator())

Functions

These functions are what the caller uses to interface with the sorted lists.

Node *createNewNode()

Analysis:
    Time: O(1)
    Space: O(1)
Params: None
Return: Pointer to new node

Creates a new node by allocating memory and setting its fields.

SortedListPtr SLCreate(CompareFuncT cf, DestructFuncT df)

Analysis:

Time: O(1)
Space: O(1)
Params: Compare function, Destruct function
Return: Pointer to new list

Creates a new empty list. The head is allocated (not see to null) and the compare and destroy function pointer fields are set accordingly.

void SLDestroy(SortedListPtr list)

Analysis:
Time: O(n), where n = number of elements in the list
Space: O(1)
Params: Pointer to list

Frees up all nodes in the list. Traverses to the end of the list and frees nodes on the way. Then frees the list object. (Note: Does not free nodes that are being pointed to by iterators, this is handled by SLDestroyIterator().)

int SLInsert(SortedListPtr list, void *newObj)

Analysis:
Time: O(n), where n = number of elements in the list
Space: O(1)
Params: Pointer to list, New object to add
Return: 1(successful), 0(unsuccessful)

Adds new node to the list by traversing the list and adding a new node where it would belong in the list in descending order. Does not take in duplicates, empty lists, or empty items (return 0).

int SLRemove(SortedListPtr list, void *newObj)

Analysis:
Time: O(n), where n = number of elements in the list
Space: O(1)
Params: Pointer to list, object to remove
Return: 1(successful), 0(unsuccessful)

Checks to see if the new object to remove is in the list. If it is in the list, we check if there are any references pointing to it. If there are none, we can just free the node by rerouting the previous node to the next one and then freeing the node. However, if there is an iterator pointing to it, or a node slated for deletion pointing

to it, then we can only set the “delete” flag for later deletion. Returns 0 if not valid list, or object not in list.

SortedListIteratorPtr SLCreateIterator(SortedListPtr list)

Analysis:
    Time: O(1)
    Space: O(n), where n = number of elements in the list
Params: Pointer to list
Return: Pointer to new SortedListIterator

Creates a new SortedListIterator that points to the head of the list and sets appropriate fields. Adds a reference to the head.

void SLDestroyIterator(SortedListIteratorPtr iter)
Analysis:
    Time: O(n), where n = number of nodes slated for deletion
    Space: O(1),
Params: Pointer to iterator

If the iterator is pointing to nothing, then we just free it. Otherwise, we would have to check if it is pointing to anything that we can free after getting rid of the iterator. There can also be a chain of nodes after that can also be freed at the same time. This function frees all of those nodes that come after also. Then, after it isn’t pointing to anything anymore, the iterator is freed. However, if the node that the iterator is pointing to cannot be freed yet, we check if we can set the “destroy” flag, decrement the references, and free the iterator.

void * SLGetItem(SortedListIteratorPtr iter)

Analysis:
    Time: O(1)
    Space: O(1),
Params: Pointer to iterator
Return: Pointer to item in node that the iterator is pointing to

If the iterator is valid and is pointing to something, then the pointer to the data is returned.

Node *correctIterator(SortedListIteratorPtr iter, void *item)

Analysis:
    Time: O(n), where n = number of items in the list

Space: O(1)
Params: Pointer to iterator, Pointer to item that the iterator is pointing to
Return: Pointer to correct node for iterator to move to

This allows us to go back to the beginning of the list and check to see where the Iterator should be going. This is a corner case for where a node is added in between a node slated for deletion and the next node in the list. For example, if a node were added in between the node where we started at and where we ended up, our iterator would end up skipping the new node. Thus, we have to go back to the beginning and search from there.

void * SLNextItem(SortedListIteratorPtr iter)

Analysis:
Time: O(n), where n = number of items in the list
Space: O(1)
Params: Pointer to iterator
Return: Pointer to next item

Given that the iterator is valid (not null), and doesn't point to null, then we can advance the iterator, given that it is not at the end of the list. It returns null in that case. However, given that it isn't at the end of the list, we check if it is on a removed node. We move the iterator forward (also incrementing and decrementing references accordingly) and free the previous node if needed. We do this until we are back in the undeleted list. After this, we call correctIterator(). correctIterator() will allow us to put the iterator back to where it SHOULD go next and then returns the data in that node. See the function description above for more details. If the iterator was not on a node slated for deletion, then it just moves on to the next item and returns the data in that node.