Laboratory 10: Cover Sheet

Name Terence Henriod Date 11/2/2013

Section 1001

Place a check mark in the *Assigned* column next to the exercises your instructor has assigned to you. Attach this cover sheet to the front of the packet of materials you submit following the laboratory.

Activities	**Assigned:** Check or list exercise numbers	**Completed**
Implementation Testing	✓	
Programming Exercise 1		
Programming Exercise 2		
Programming Exercise 3		
Analysis Exercise 1		
Analysis Exercise 2		
	Total	

Laboratory 10: Analysis Exercise 1

Name Terence Henriod Date 11/2/2013

Section 1001

Given a hash table of size *T*, containing *N* data items, develop worst-case, order-of-magnitude estimates of the execution time of the following Hash Table ADT operations, assuming they are implemented using singly-linked lists for the chained data items and a reasonably uniform distribution of data item keys. Briefly explain your reasoning behind each estimate.

insert O(1)

Explanation:
Assuming we are inserting items into the linked lists without regard for order, the insert operation takes a constant amount of time. The constant time that is required is just that of the hashing function and the time it takes to insert an item at any place in a linked list (that is already indicated by a head, cursor, or tail pointer). The number of data items will not affect this operation due to the constant access nature of the hash table array and the ease of insertion into a linked list.

However, if we assume that we are inserting items into the linked lists in order, the complexity of the operation increases to O(N). While the hash table referencing time is constant, we are still limited by the data structure utilized for chaining. The hash table certainly reduces the access time by allowing us to directly select a list at the cost of the constant time required for the hashing function, but still, once a list is selected, it takes linear time to search a linked list for the appropriate insertion point. Thus, the time required is (1/T) * N, but (1/T) is a constant, so it gets eliminated to produce O(N).

retrieve O(N)

Explanation:
Again, even though the hash table allows us to select a list in constant time, it will still take linearly increasing time to search a list for the sought item.

What if the chaining is implemented using a binary search tree instead of a singly-linked list? Using the same assumptions as before, develop worst-case, order-of-magnitude estimates of the execution time of the following Hash Table ADT operations. Briefly explain your reasoning behind each estimate.

insert O(N)

Explanation:
There is no random insertion in a binary tree, so a binary search must take place in order to find the appropriate insertion point for the new data item. Again, the hash table reduces this time by the constant (1/T), but Big O notation ignores such constants. Thus, a constant time selection of a tree and a log(N) binary search come together to make an overall complexity of
O($\log_2$(N)), for a balanced tree. However, the worst case for searching a binary search tree is the case where the tree is more akin to a list because each node (except the leaf) has exactly one child.

retrieve O(N)

Explanation:
This estimate uses the same reasoning as above. Constant time selection of a tree followed by a binary search for the sought item results in a O($\log_2$(N)) complexity for a balanced tree. However, in the worst case tree, the tree is more akin to a list, so this would take O(N) time to search.

It should be noted that while the complexity estimates don't seem to reflect an advantage, the constant time reduction factor provided by the hash table (1/T) is probably very much noticeable in practical application, therefore the hash table should not be disregarded based on these complexity estimates.

Laboratory 10: Analysis Exercise 2

Name Terence Henriod Date 11/2/2013

Section 1001

Part A

For some large number of data items—e.g., *N*=1,000,000—would you rather use a binary search tree or a hash table for performing data retrieval? Explain your reasoning.

A hash table. A hash table with a moderate table size of 1000 would result in trees of size 1000 (assuming a perfect hash and balanced trees). Even though binary search trees are relatively quick to search, scaling their size down by 1000 will likely result in noticeable performance increases by search time reductions. For example, a worst case search of a (balanced) tree of size one million would take ~14 comparisons to search, while the worst case (balanced) tree of 1000 would take ~7 comparisons to search, worst case. The search time was roughly halved!

Part B

Assuming the same number of data items given in Part A, would the binary search tree or the hash table be most memory efficient? Explain your assumptions and your reasoning.

The binary search tree is more memory efficient because there is not the overhead of instantiating multiple trees and the overhead required by the hash table object.

Part C

If you needed to select either the binary search tree or the hash table as the general purpose best data structure, which would you choose? Under what circumstances would you choose the other data structure as preferable? Explain your reasoning.

For a general purpose best structure, I would choose the binary search tree because it does not require the same kind of support the hash table does. It still has good performance, and only a getKey() member is required. The hash table must also have a carefully designed hashing function supported. The hash table's performance benefits are also likely only noticeable with very large data sets, much larger than anything that I have personally worked with.