Project 3: DictionaryADT

Writeup by **masc0264**, Vincent Chan. RedID **815909699**.

Preform emperical timing tests to verify that the data strctures created do complete their functionality. Methods to be tested include:

- Insert
- Delete
- search

All these tests will be done with an already populated dictionary.

Ordered Array

An ordered array is a constant sized array that will find and re organize the array on insertion and deletion. It's search time is really quick but is counteracted by it's need for shifting every time it insrts or deletes elements.

Pros:

- O(log n) search

Cons:

- Limited memory without a scaling function
- O(n) insert, deletion

Insert O(n)

Although the Ordered array can find the insertion location fairly quickly, it suffers from the need to shift the array to make room for the newly added element as shwon:

```
    for(int i=size; i>insertLoc; i--) nodeArray[i] = nodeArray[i-1];
```

At the worst case, the insertion will be at the first element of the array, requiring the whole array be shifted. This give it an O(n) complexity.

Deletion O(n)

Deletion is exactly the inverse of insert in this case, so it also has a complexity of O(n).

Search O(log n)

Search is quick because an ordered array allows for a binary search. This gives the search algorithm a similar complexity to a BST data structure of **O(log n)**.

Hash Table

Hash tables are powerful for dictionaries, giving insertion, deletion, and search all O(1) complexities. However, one major downside is that to reach this complexity, memory space much larger than the actual used memory space must be allocated.

Pros:

- O(1) search, insert, deletion

Cons:

- Takes more memory than it should

Insert O(1)

The hash table can quickly insert in two steps: hash the key for the index, then insert at that index. That way, it knows exactly where it is if we ever need to search for it. In cases of collisions, the linked list located at that index will chain the elements together. An overfilled hash table may exhibit O(n) behavior, but it generally keeps an O(1) behavior.

Deletion O(1)

The hash table simply checks to see if the element is in the dictionary (it knows the exact indexes of this element because

it uses hashes to assign elements pre-determined indexes). Again, in this implementation, which uses linked lists to chain collisions, it might have to make a further search to find the element. But, in general, it has an O(1) behavior.

Search O(1)

The hash table will simply hash the object to search, and search at that index for that object. This is an O(1) behavior because no iterations are needed (unless there is a collision, in which case very few iterations happen, which are generally negligible).

Binary Search Tree and Red Black Tree

Binary search trees and red black trees are very similar, so they are easily explained in one section. The difference between a BST and a RBT is that the RBT self balances on the insertion. An unlucky insertion sequence can create an O(n) search time for a BST, while a RBT will self balance, eliminating the O(n) worst complexity.

Pros:

- Quick search, deletion, insertion
- scaling memory

Cons:

- Can't contain duplicates (Although not a con for a dictionary...)
- Huge trees may cause a stack overflow because of the tendancy of BSTs to be implemented with a lot of recursive functions.

Deletion O(log n)

Deleting a node is O(1), because all it has to do is pop out the node, and re-refrence the parent of that node to the next node in line. This function is only limited by it's search time, which is O(log n).

Insertion O(log n)

Same as the deletion, the insertion is also limited by it's searching, which is O(log n).

Search O(log n)

One of a tree's best benefit comes from it's ability to quickly search for elements using a binary search. This benefit is due to it's structure (less on left, more on right, two children each). Its structure lets us eliminate half the possibilities each comparison, letting us drill down to the sought element quick, O(log n) quick.