HackerRank For Work

Fall_2017_INFO6205_Midterm 90 minutes

Question - 1
Data Structure

SCORE: **4 points**

You've been asked to program a bag in the knowledge that the number of elements in the bag will always be less than 10,000 and you have whatever memory you need. But the time to add an element must be constant. Also, the total time to iterate forwards or backward must be no worse than O(n). With which data structure would you choose to implement the bag?

- Hash table
- Array
- Doubly-linked list
- Linked list

Question - 2
Question 2

SCORE: **4 points**

You have been assigned to design a stack. It is essential that there is never a delay in pushing or popping the stack, that's to say these operations should be performed in O(1) time. Again, there are no memory constraints. Which data structure would be most suitable?

- Doubly-linked list
- B-tree
- Array
- Linked List

Question - 3
Question 3

SCORE: **4 points**

You have developed an algorithm that takes $O(n^2)$ time to solve a problem with n elements. The algorithm currently runs on a cluster of 10 nodes. Thus far, n has never exceeded 10,000. However, a new customer has requested a quote for running n = 20,000. You can spend $20,000 on building the cluster out to 40 nodes or you can pay the same amount to a programmer who claims that he can refactor your algorithm to run in $O(n \log n)$ time. You should start ordering the new hardware, right?

- True
- False

Question - 4

SCORE: **4 points**

Question 4

Why did you choose your answer to Q3?

Question - 5

SCORE: **9 points**

Question 5

An experiment which involves successively doubling the number of elements which must be processed yields the following mean times:

ExperimentTimes

n	**t**
8	12.4
16	32.2
32	79.4
64	192.2
128	443.7

Postulate a model which explains this behavior (that's to say, derive an expression for *t* in terms of *n*)

Question - 6

SCORE: **4 points**

Question 6

```
Question 5:

An experiment which involves successively
doubling the number of elements which must be
processed yields the following mean times:

ExperimentTims
n        t
8       12.4
16      32.2
32      79.4
64      192.2
128     443.7

Postulate a model which explains this behavior
(that's to say, derive an expression for t in
terms of n)
```

Predict the amount of time that the algorithm will take for 256 elements.

Question - 7

SCORE: **4 points**

Question 7

A list has n elements in order. A new random element is to be added to the list. Without yet knowing the value of the new element, In how many different ways can it be added to the list while maintaining the proper order?

Question - 8

SCORE: **2 points**

Question 8

Express the answer to Q7 in terms of entropy.

Question - 9

SCORE: **2 points**

Question 9

What is the minimum number of comparisons that the algorithm in Q7 must perform to complete the task?

Question - 10

SCORE: **6 points**

Question 10

Prove that sum of the numbers 1 through n is equal to $n(n+1)/2$

Question - 11

SCORE: **2 points**

Question 11

Represent the expression in Q10 in "tilde", i.e. "~" notation.

Question - 12

SCORE: **5 points**

Question 12

Stirling's approximation for $n!$, the number of permutations of a list of n different objects, expressed in entropy, as bits, is: $n \lg n/e + \ln(2 \pi n)/2$.
Express this in ~ ("tilde") notation.

Question - 13

SCORE: **4 points**

Question 13

Selection sort and Insertion sort are both $O(n^2)$ algorithms so they are only suitable for relatively small n. Which of the following are true?
a. Insertion sort does half as many comparisons, generally speaking, as Selection sort;
b. Insertion sort can reduce the time for exchanging elements by moving blocks of elements;
c. Insertion sort is linear when the list is already sorted--unfortunately, this is not true for Selection sort.

- a only
- b only

- c only
- a and b
- b and c
- a and c
- all of the above

Question - 14

SCORE: **3 points**

Question 14

You are required to implement a method for the storage of up to 1 million elements. Each element has a key which represents a total ordering. It is desired to be able to select any element according to its key. However, you expect that the total number of different possible keys is somewhat greater than 4 billion (the number of possible *int* values). You don't want to search for these elements by traversing the list and comparing keys. But obviously you don't want to assign array storage to have one open slot for every possible key value! The way to do this is to implement a hash table. When you add an element to the table, you compute its *hashCode* and map that (typically by shifting bits to the right) into an index value which points to an element of the table. When it's possible (as it usually is) for many values to map to the same element (called a *collision*) there are several schemes to deal with that. The simplest is to use the next higher empty slot.

However, for lookup (in the table) and comparison purposes, you will use the *hashCode* as a surrogate for the key itself. The *hashCode* is monotonically increasing with the key. That's to say that if *hashCode(x) > hashCode(y)*, then *x>y*. However, since the *hashCode* is essentially a 32-bit digest of the key, the *hashCode* itself does not qualify as a total order for the elements. That's because it is possible that *x > y* or *x < y* while, at the same time, *hashCode(x) = hashCode(y).*

If there are 100 million possible different values in the domain of the values (that's to say there are 100 million possible values of the key), what is the (approximate) probability that two different elements will have the same hash key (assume that the *hashCode* is uniformly distributed over the domain)?

Question - 15

SCORE: **16 points**

Question 15

Implement the missing code segment for the following implementation of Insertion Sort.

This implementation operates on a *List* and returns a (different) *List*. The underlying type of the lists is *Comparable*. However, the internal work is done with two arrays: one for the indices and one for the hash codes. The idea is to do all of the comparisons (remember, this is an O(N^2) operation) using *hashCode* rather than using the full, natural key. The hashCodes are, therefore, pre-computed and placed into the *hashes* array.

The *indices* array is simply the index into the original list.

As we discussed in part (a), there is a possibility that some *hashCodes* will match, even though the actual values don't match. Therefore, there is a verification pass which is run after the preliminary sorting. If any pair of hashes are equal, we adjust their positions as appropriate, according to the true key (defined by the *compareTo* method of the underlying type).

Once the verification is done, we simply load up a list according to the array of indices we now have.
There is a simple unit test for this which involves date-time objects (the class is called Date).