Questions

1. Suppose you are given a list of n elements. A "brute force" method to find duplicates could use two (nested) loops. The outer loop iterates over position i the list, and the inner loop iterates over the positions j such that $j > i$ and checks if the elements at i and j are the same. The brute force method takes $\Theta(n^2)$ steps.

   Give a $\Theta(n)$ method for finding the duplicates.

2. Suppose you are given a singly linked list but you don't know how many elements are in the list. When traversing the list you find that it takes a very long time and you suspect that there is an error in the code and that in fact there is a loop in the list. How can determine if there is indeed a loop in the list in $O(n)$ time, where n is the actual number of elements in the list?

   One way to do this would be to mark the nodes as you visit them. However, if you cannot change the code then you cannot do that.

3. Both hashing and binary search trees allow you to efficiently store and access map entries. What are the advantages/disadvantages of each, which would make you choose one over the other?

4. Consider a hash table with capacity m and with n entries, i.e. the load factor is n/m. Give O() and Ω() bounds for a method (called an iterator) that visits all entries.

5. Two different Java strings can have the same `hashcode()` value. Prove it, by considering strings of length 2.

   Note: so far in this course, we have only considered ASCII coding for characters in strings (namely 8 bit codes). In Java, characters in a string in fact are coded with 16 bits each. THe code is called UNICODE. It allows for characters in many different languages. In answering this question, you should assume that each character is coded with 16 bits and so a string with two characters has 32 bits.

6. Suppose we were to define a hash code on strings `s` by:

$$h(\mathtt{s}) = \sum_{i=0}^{n-1} \mathtt{s}[i]\ x^i$$

   where $s[i]$ is the 16-bit unicode value of the character at position i in the string, n is the length of s, and x is some positive integer.

   Give an upper bound on the number of bits needed for the hash code as a function of x and n (the length of the string).

7. The return type of the Java `hashCode()` method is `int`. But the definition of the hashcode of a string is a polynomial whose value can easily exceed the 32 bit limit of the *int* type. How is this possible?

8. Suppose you have approximately 1000 images that you would like to store. Rather than labelling them using a string (i.e. filename) and indexing them based on filename, you would like to label them using small images called "thumbnails". Let's say each thumbnail image is 64×64 pixels, and each pixel can have intensity values from 0 to 255 (ignore color here.) We want to use the thumbnail images as keys in a hash function.

   Suggest a suitable hash function, namely one that avoids collisions and that doesn't take too much space.

9. Canadian postal codes are of the form $L_1D_1L_2D_2L_3D_3$ where L is always a letter `(A-Z)` and D is always a digit `(0-9)`. Suppose you have your own company and you wish to index your customer addresses using the postal code as the key. Let the letters `A-Z` be coded with numbers 1 to 26, for example, $code(B) = 2$ and let the digits be coded by their numerical value.

   (a) Define a hash function:

   $$h(L_1D_1L_2D_2L_3D_3) = (\sum_{i=1}^{3} code(L_i) + \sum_{i=1}^{3} D_i) \bmod 10.$$

   Give an example of a postal code that begins with `H3A` and that collides with `H3A2A7`.

   (b) Give an example of a hash function that would *never* result in a collision. How large would the hash table need to be?

Answers

1. This question is in the hashing exercises, so naturally the solution involves hashing. The method is simple: start with an empty hash set. Then, iterate through the list and add each element to a hash set. If the element is already in the hash set, then you have found an element that is duplicated. If you wish to store the number of duplicates, you can use a hash map where the value stored for each element is the number of copies of the element. This method is $\Theta(n)$ because there are n elements in the list and hashing each of them takes $O(1)$ time.

2. Use a hash set. As you visit each node in the list, add the element to a hash set. If there is a loop, then you will you will discover a duplicate node after n elements have been visited.

3. First, a BST generally gives slower access than a hash table. Why?

   With a BST, we would search for a key by following a path from the root towards the leaves. For each node in the BST we encounter, we do a key comparison $(<, =, >)$. If the BST is balanced (best case), then it will take us $O(\log n)$ steps to find a key, or determine that there is no matching key, where n is the number of keys in the BST. So, for example, if $n = 2000$ and the tree is balanced, then it will take us on average roughly 10 comparisons to find an item (i.e. $2000 \approx 2^{11}$, and about that half the nodes in a complete binary tree are leaves.) If the BST is not balanced[1], then it will take longer to find the key. This is faster than using a linked list, but still slow relative to a hash table. Why? With hashing, $h(key)$ provides a hash value which is a number from 0 to $m - 1$ where $m \approx n$. Given a key, k, we compute $h(k)$ using some formula or algorithm, and then we go directly to the entry $h(k)$ in the hash table and search through a (typically very short) list for key k and so we have $O(1)$ access. (Note that the BST requires a comparison to be made at each node, and these comparisons take time. So you can't just argue that hashing is more expensive because you need to compute hash values.)

   Another way to think about the difference is to note that a binary search tree does a sequence of comparisons, i.e. each comparison returns one of three values $(<, =, >)$, which is relatively little information. (We only need 2 bits to specify one of three values.) By contrast, a hash function gives you $\log m$ bits of information, namely it specifies one of m values. You still need to scan the entries within the bucket at index $h(k)$. But if the hash function does a good job, then most of the buckets will have very few entries.

   So is there any advantage of a binary search tree over the hash table? Yes! The BST is only used if the elements are comparable. In this case, you might sometimes want to list the elements in order. The BST is great for that. You can do an in order traversal and list the elements in order. You can also list a range of elements (by doing a traversal, and only listing the elements in the range). With a hash table, one doesn't reprsent the key orderings at all, so reading off an ordered list of keys cannot be done.

[1]keep in mind that it takes extra work to keep it balanced – wait for COMP 251

4. The simplest way to define the iterator would be to step through each bucket in the hash table. There are m buckets and each needs to be examined, both in the best case and worst case. For each non-empty bucket, the (key,value) pairs ("entries") in the bucket must each be visited i.e. the list in each bucket needs to be traversed. These traversal require a total of n visits, regardless of how the entries are distributed over the buckets. Thus, the best case is the same as the worst case, namely $\Theta(m+n)$.

5. There are $2^{16} * 2^{16} = 2^{32}$ possible strings of length 2. The hashcode for a string $s[0]s[1]$ is $s[0] * 31 + s[1]$ which is less than $2^{16} * (31+1)$, or $2^{16} * 2^5$, that is, 2^{21}. Since there are 2^{32} strings of length 2, and there can be at most 2^{21} hashcodes for these strings, it must be the case that two strings have the same hashcode.

6. The largest hash code is $2^{16}(x^0 + x^1 + \cdots + x^{n-1}) = 2^{16}(\frac{x^n-1}{x-1}) < 2^{16}x^n$. Taking the log gives the number of bits of the hashCode is at most $16 + n\log_2 x$.

   For example, if $x = 31$ (as in Java's String class'es hashCode method), the hashcode would be at most $16 + 5n$ bits. Here is an example with $n = 4$.

```
                  ****************     s[0] x 31^0
             *********************     s[1] x 31^1
        **************************     s[2] x 31^2
+  *******************************     s[3] x 31^3
   -------------------------------
  ********************************     hashCode( s )
```

7. The `String.hashCode()` computes the value of the polynomial <u>`mod` 2^{32}</u>, i.e. it uses the last 32 bits of the number defined above as the return value of `hashCode()`. This is implicitly stated in the Java API where it says that this method computes the above polynomial "*using* `int` *arithmetic*".

8. Let's make a hash table with $m = 2000$ entries so that we are sure there will be plenty of buckets with no elements (and hence collisions are relatively rare).

   For the hash function, we need to map the 64×64 pixel colors to a number larger than $m = 2000$. To do so, we could define a hash code to be the sum of the intensity values at the pixels. Assume the average intensity value is 128, we would get a number on average of $64 * 64 * 128$ which is much bigger than m. Then, to get the hash value, we could take the *sum* of the color values and compute "*sum* mod m".

9. (a) h(`H3A2A7`) = (8 + 3 + 1 + 2 + 1 + 7) mod 10 = 2
   So, you need to come up with a postal code $D_2L_3D_3$ such that $D_2 + code(L_3) + D_3$ mod 10 is the same as 2 + 1 + 7 mod 10. The latter is 0. So, for example, if you take "3A6", then $D_2 + code(L_3) + D_3$ mod 10 is 0, and h(`H3A3A6`) = (8 + 3 + 1 + 3 + 1 + 6) mod 10 = 2

   (b) One simple solution is to use base $b = 26$ and define:

$$h(L_1D_1L_2D_2L_3D_3) \equiv code(L_1) + D_1b + code(L_2)b^2 + D_2b^3 + code(L_3)b^4 + D_3b^5$$

In these cases, the postal code $Z9Z9Z9$ would give the largest hash value, and you can plug in the numbers and letters to get the largest value. That is how big the hash table would need to be for this hash code.

You can be more clever and use a smaller hash table, by noticing that there are $10^3 * 26^3$ possible postal codes. You could come up with a hash function that maps each postal code to one of the numbers from 0 to $10^3 * 26^3 - 1$. For example,

$$h(L_1D_1L_2D_2L_3D_3) \equiv (D_1+D_2*10+D_3*10^2)+10^3*(code(L_1)+code(L_2)*26+code(L_3)*26^2).$$

The idea here is that the three letters have at most 26^3 triplets, which we can code with the numbers 1 to 26^3 (and this code is in base 10). We can then multiply each of these numbers by 10^3, freeing up the three digits. These digits can be filled with the code for the D_i's.