Questions

1. Consider each of the following data structures to store a set of entries of a map. What would be time complexity (worst case) for each of the operations: put, get, and remove?

   Note that the arguments are: put(key,value), get(key), and remove(key).

   (a) arraylist or linked list

   (b) binary search tree

   (c) heap

2. Two different Java strings can have the same `hashcode()` value. Prove this statement, by considering strings of length 2. (You might think this is just obvious. Fine, but how would you convince *someone else* who wasn't as sure as you?)

   Assume the characters in a string are coded with 16 bits each.

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

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

   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).

4. 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?

5. 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 $O(n^2)$ steps.

   Give a $O(n)$ method for finding the duplicates. Hint: use a data structure based on hashing.

6. 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?

7. Suppose you have approximately 1000 images that you would like to store. Each pixel can have intensity values from 0 to 255 (ignore color here.) Rather than labelling the images 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.

   Further suppose we want to use the thumbnail images as keys in a hash function, that is, (key, value) is (thumbnail image, original image). Suggest a suitable hash function, namely one that tends to avoid collisions.

8. 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 a company and you wish to keep track of customers who have that postal code, so the map might be (postal code, list of customers). 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 (array) need to be?

Answers

1. (a) arraylist or linked list – For get(key) and remove(key), you need to scan through the list in the worst case, so it is $O(n)$ for a list of size n.

   For put(key,value), you might think that you could just add on an entry to the end of the list and so it would be $O(1)$. However, that's not correct: you need to scan the list first to make sure that the key is not already there as part of another entry. So in fact it would also be $O(n)$.

   (b) binary search tree – in the worst case, a binary search tree is essentially just a linked list, namely it has height $n-1$. So the same answers as above hold.

   (c) heap – Here we think of each node of the heap as having a map entry, and all the usual heap operations apply. So put(key,value) is just like "add" and takes time $O(\log n)$. remove(key) and get(key) cannot take advantage of the heap structure; instead one needs to scan through the entire underlying array to find the element; so it takes time $O(n)$ in the worst case.

2. 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 $2^{16} * 2^5$, 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.

3. The largest hash code is $(2^{16}-1)(x^0 + x^1 + \cdots + x^{n-1}) = 2^{16}(\frac{x^n-1}{x-1})$. One can show that if $x \geq 2$, then $\frac{x^n-1}{x-1} < x^n$. (One can prove this by induction – but I'll omit that proof here.)

   Thus, the largest hash code is a number strictly less than $2^{16}x^n$. To know the number of bits that we need to represent this largest hash code, recall that the number of bits needed to represent an integer m in binary is $\lfloor \log_2 m \rfloor + 1$ where the brackets denote the "floor" operator. So an upper bound on the number of bits of the hashCode is the floor of $\log_2(2^{16}x^n) + 1$, or $16 + n\log_2 x + 1$ or $17 + n\log_2 x$.

   For example, if $x = 31$ (as in Java's String class'es hashCode method), the hashcode would be at most $17 + 5n$ bits. Here is how you can think of this for $n = 4$. Each of the * symbols is a bit. Note that multiplying a number by 31 (which is approximately 2^5) will roughly increase the number of bits by 5. So in each row below, I've increased the number of bits by 5. Then I'm adding up the rows.

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

4. The Java API for `String.hashCode()` says that it computes the given polynomial "*using* `int` *arithmetic*". Thus, the hashcode is a number between 0 and $2^{32}-1$ rather than -2^{31} to $2^{31}-1$. So you can think of taking the number mod 2^{32} and then mapping this to the range -2^{31} to $2^{31}-1$ as was illustrated back in lecture 3 when we discussed Java primitive types.

5. 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 $O(n)$ because there are n elements in the list and hashing each of them takes $O(1)$ time.

6. First, a BST typically gives slower access than a hash table. Why? With a BST, we 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. (You might have argued that hashing is more expensive because you need to compute hash values. However, note that the BST requires a comparison to be made at each node, and these comparisons take time.)

   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! A 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. 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 represent the key orderings at all. So even if elements are comparable, there is no way to read off an ordered list of keys.

[1] keep in mind that it takes extra work to keep it balanced – you will learn about balanced binary search trees in COMP 251

7. 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 intensity values 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 $m = 2000$. Then, to get the hash value, we could take the *sum* of the intensity values and compute "$sum \mod m$" where the m was mentioned above.

8. (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.

