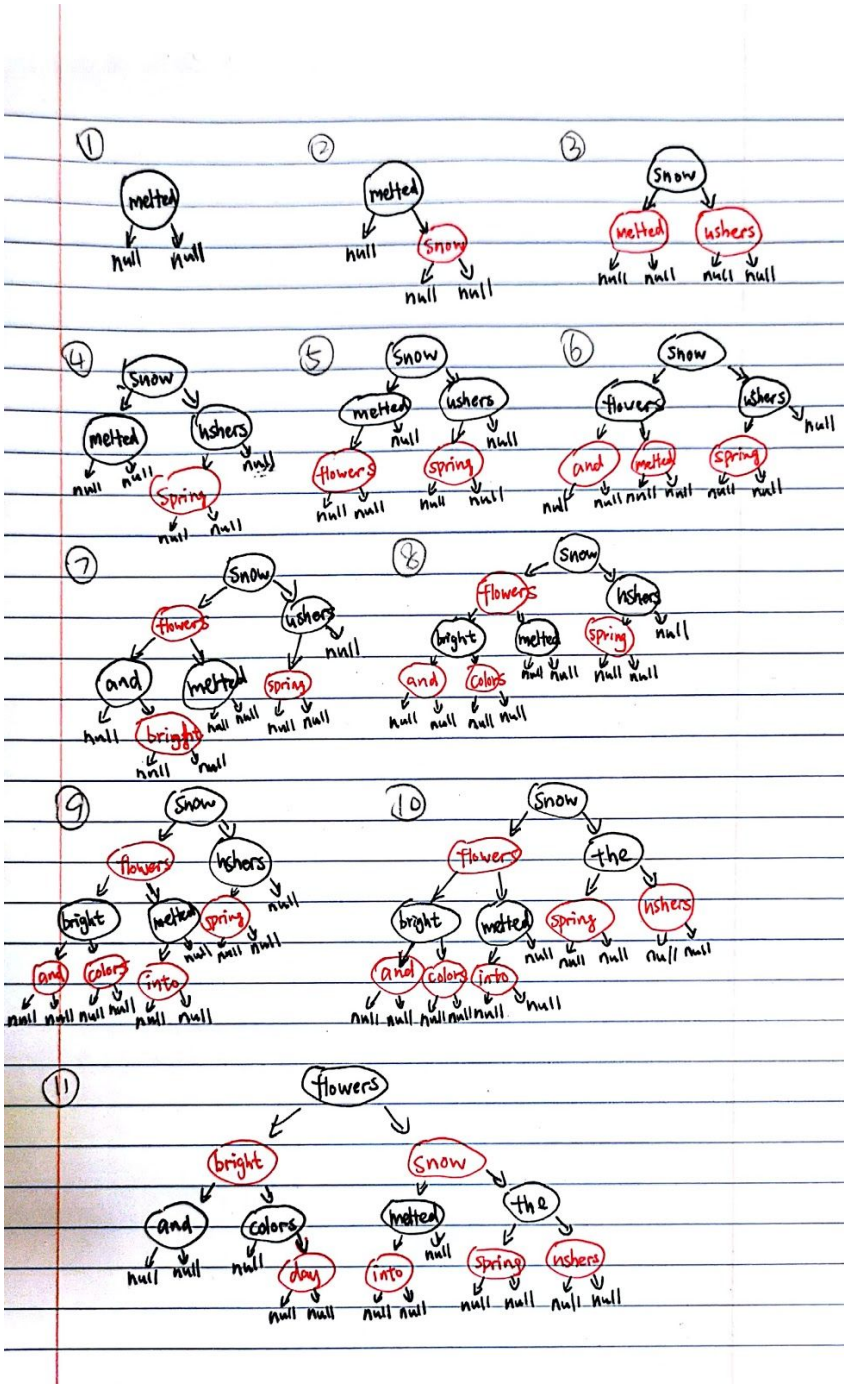
Name: Katrina Li

Red-black trees

1



2

The contains() method works by binary searching, so the efficiency depends on the height of the tree. The regular binary tree is not guaranteed to be bushy. In the worst cases, the regular binary tree would look like linked lists. On the other hand, the colors of nodes are constraints for red black trees. The maximum range of difference in the length from root to leaf does not surpass twice of the shortest one. The asymptotic efficiency for a regular binary tree is O(log n) in average cases and O(n) in worst cases but for red black tree it is O(log n) in average cases and O(2*log n) = O(log n) in worst cases.

Hash functions and hash tables

1 // I used lowercase() and got rid of the characters just like the “word count” in homework 11.

		#empty bucket	avg(items / non-empty bucket)	#empty bucket	avg(items / non-empty bucket)
#Bucket	HashFunction	HoundOfTheBaskervilles		Words	
	-1	193324	1.796	59967	1.715
	0	196612	5906	196612	234369
	1	196588	236.2	196587	9014
	2	195695	6.434	194608	116.9
196613	3	190810	1.018	59651	1.711
	-1	196710	1.795	61956	1.698
	0	199999	5906	199999	234369
	1	199975	236.2	199974	9014
	2	199082	6.434	197995	116.9
200000	3	194192	1.017	61799	1.696

Based on the testing I would like to use the original hashCode() method or the hashCode3() because they almost have no collision. The advantage of this reflects in the following ways: Firstly, they achieve O(1) for looking for a node in a map with a huge number of buckets. Secondly, this greatly helps with collision resolution by taking less space in separate chaining and storing objects faster in open addressing.

An explanation for the other hashCode() functions: hashCode0() simply returned the same number for any string, so all words went into the same bucket, which is not really generating. The hashCode1() generated hash values based only on the initial letter so there are about 50 possible values, and will fill in at most about 50 buckets. The hashCode2() summed up the (integer) values of each letter/character in the string so it will fill in at most about 2000 buckets.

Make # of buckets a prime number would slightly help. For hashFunction-1 (the original one) in words.txt, there are respectively 196613-59967 = 136646 buckets and 200000-61956 = 138044 in use. This is significantly different than the small numbers that we discussed in class. My explanation is that it is unlikely to have lots of collisions, given that the hashCode() function is perfect, when about 30% of the buckets were still empty. This is the same with both a huge prime number of buckets and a huge non-prime number of buckets.

2 Assume that an averaging English word has at most 20 letters so the range of resulting hash value would range from 39(the character is ‘’’) or 65 (the word is ‘A’) to 122*20 = 2440 (with 20 ‘z’s in the

word). Therefore this function fails to create essential variance in hash values, and the method will not use more than 2500 buckets out of about 200000 buckets.

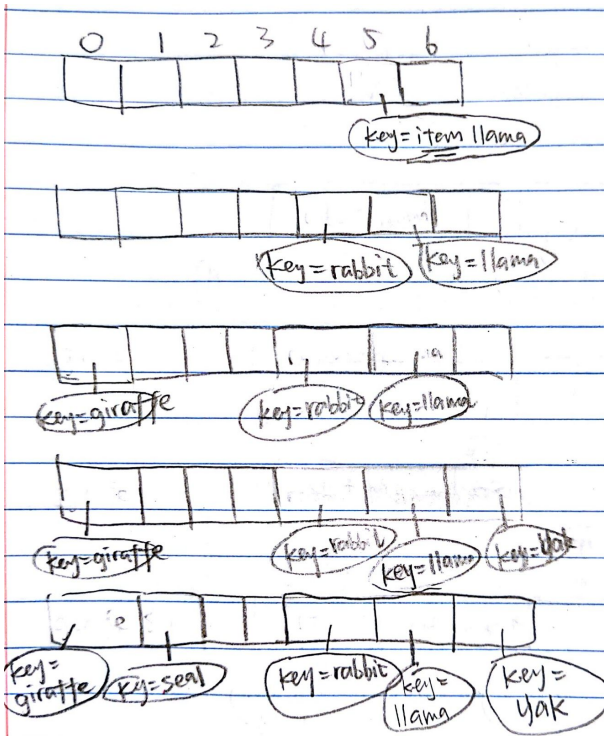
3 This function is ineffective because there are no words that could begin with most of the four letters in combination. For example, no English words could begin in “abcd” but words such as “computer”, “compassion”, “compost” all begin in “comp”. Therefore this creates unnecessarily large number of empty brackets and collisions at the same time.

4

We could use a similar method as hashCode3(). We could concatenate the Strings for name, species, image path, and year of birth into a huge String and calculate the hashCode as hashCode3() did. Since two equal pets do not necessarily have the same preferred toys, the String for preferred toys should not be involved in the calculation of hash value, as pet1.equals(pet2) is equivalent to hashCode(pet1) = hashCode(pet2).

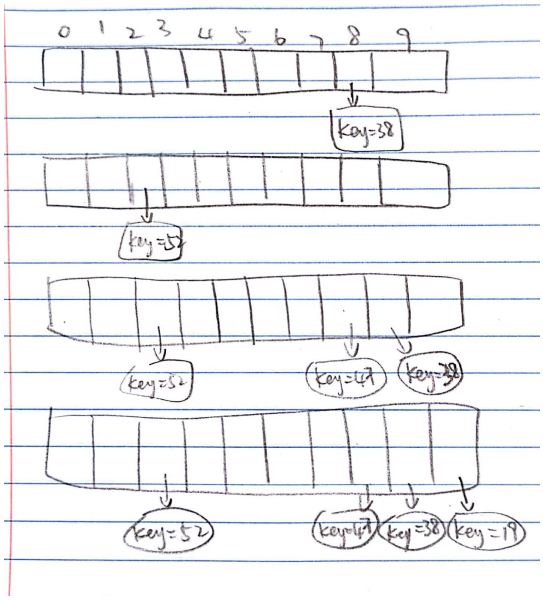
5

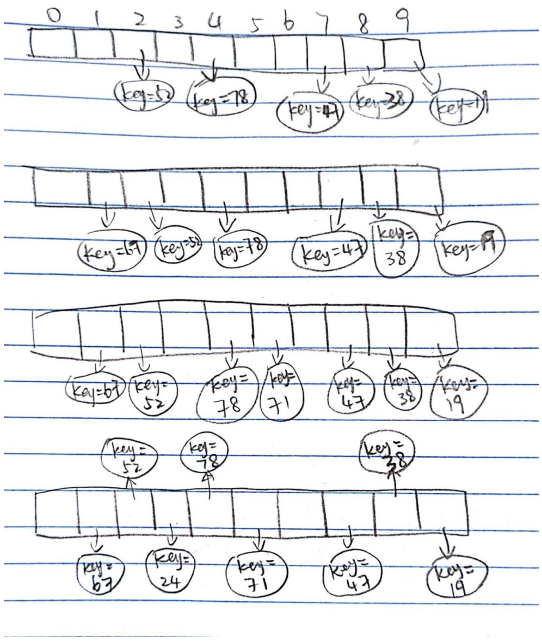
a.



b. I should examine array[6], array[0], array[1], array[2] and stop because I found out that array[2] is a null bucket.

6





7

The compression method(s) takes in the concern of both the number of buckets and the collision resolution. Hash code is more related to the Object itself (kind of like an instance variable) so it should be arranged and designed as one of the methods (hashCode()) of the Object class. However, different map implementations can import this class and use the objects in the original class in a number of ways, and the objects that need to be created in those classes could have different characteristics such that the compression method and the ways of collision resolution would be altered in order to achieve higher efficiency. Furthermore, it is not guaranteed that the keys stored in the maps are of the same kind, so if the hashCode() is defined in maps it has to exist in every one of the map implementations which leads to inconvenience and error.