Assignment 6 - Due: December 10 at 6:45 PM

Please hand it in to me before class as a hard copy.

You can e-mail me a soft copy if you miss class.

Handwritten submissions are absolutely fine.

Full Name:____

1- We plan to use this hash function to hash ~ half million words into an array of size ~ a million and we want to keep collisions of unequal words to a minimum. Words are considered equal if they match exactly while ignoring the letter case (e.g. AbCd and abcd are equal).

```
int hash(String str, int arraySize) {
        int h=0;
        for (int i=0; i<str.length(); i++)
                h+=orderInAlphabet(str.charAt(i));
        return h % arraySize;
}
```

orderInAlphabet is an O(1) time method that returns an int equal to the order in the English alphabet of the given letter. E.g.:

orderInAlphabet('a') → 1
orderInAlphabet('A') → 1
orderInAlphabet('b') → 2
orderInAlphabet('c') → 3

a- Is this hash function functionally correct? Yes_____ No ______
Specifically, does it guarantee that equal words have equal hash values? Explain your answer succinctly below:

b- Is this a good hash function for what we plan to use it for? Yes_____ No ______
Specifically, is it uniformly distributing? If yes, briefly and clearly explain why. If no, briefly and clearly explain which part(s) of the array it has a preference for (i.e. it will hash more words there).

c- If we use a hash table implemented using separate chaining, give a rough estimate of the typical chain lengths this hash function would generate if words are 1-10 letters, the average letter has an alphabet order of 10, and we are hashing 500,000 words. State your assumptions (if you make any) and perform some simple math; no complicated probabilistic analysis. Briefly show your work:

2- We are performing the following operations on a hash set implemented using open addressing which follows the linear probing collision resolution scheme. The initial capacity is 8. The table automatically doubles and elements are rehashed when an insert would cause alpha to be greater than 0.5 (equal to

0.5 does not trigger a rehash). When a rehash is triggered, all the existing elements are rehashed into the new array in the order they are encountered when iterating over the array. Then the element that triggered the rehashing is inserted.
Show the state of the array right before resizing and the final resized state after rehashing. Elements that are in the array but are deleted should be shown on the array as crossed out (legibly so).
The hash function is h=val % arrSize.   + == insert, - == delete
22, 12, 20, 13, -22, -20, 32, 12, 14, 13, 16, 29, -32

0	1	2	3	4	5	6	7

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

3- We are performing the following operations on a hash set implemented using separate chaining. The initial capacity is 5. The table automatically doubles and elements are rehashed when an insert would cause alpha to be greater than 2 (equal to 2 does not trigger a rehash). When a rehash is triggered, all the existing elements are rehashed into the new array in the order they are encountered when iterating over the hash set. Then the element that triggered the rehashing is inserted.
Show the state of the array and the chains right before resizing and the final resized state after rehashing. When we add an element to a list, add policy is to add to the end of the list.
The hash function is h=val % arrSize.   + == insert, - == delete
22, 20, 12, 13, 32, 12, 14, 16, 29, 20, -12, 15, 11, 19, 33, 17, 28, 10, 1, -13, 9, 2, 35

(please grow the chains downward)

0	1	2	3	4

0	1	2	3	4	5	6	7	8	9