Problem Set 9 - Solution

Hash table

1. You are given the following keys to be hashed into a hash table of size 11:

   ```
   96,  43,  72,  68,  63,  28
   ```

   Assume the following hash function is used

   ```
   H(key) = key mod 11
   ```

   and chaining (array of linked lists) is used to resolve collisions.

   1. Show the hash table that results after all the keys are inserted.
   2. Compute the average number of comparisons for successful search.

   SOLUTION

   1.
      ```
      0 []->/

      1 []->/

      2 []->68->/

      3 []->/

      4 []->/

      5 []->/

      6 []->28->72->/

      7 []->/

      8 []->63->96->/

      9 []->/

      10[]->43->/
      ```

   2. The average number of comparisons for successful search is:

      ```
      (1+1+2+1+2+1)/6 = 4/3
      ```

2. Using chaining to resolve collisions, give the worst-case running time (big O) for inserting *n* keys into an initially empty hash table table for each of the following kinds of chains:
   - Chain is an unordered list
   - Chain is an ordered list (entries stored in ascending order of keys)
   - Chain is an AVL tree (ordered by keys)

   SOLUTION

   In the worst case, ALL *n* entries are in the same chain.

   - Chain is an unordered list
     Every new entry is inserted at the front of the list, in *O(1)* time. For *n*, the total time is *O(n)*
   - Chain is an ordered list (entries stored in ascending order of keys)
     In the worst case, every new entry is to be added to the end of the chain. The first entry is added in 1 step, the second in 1 step, third in 2 steps, etc. The total time is:

     ```
     1 + 1 + 2 + ... n-1
     ```

     which sums to *O(n^2)*
   - Chain is an AVL tree (ordered by keys)
     Each insert takes logarithmic time in the size of the AVL tree (size includes the inserted entry). The first entry takes unit time to insert, the subsequent entries sum up to the following:

     ```
     log(2)+log(3)+...+log(n)
     ```

     This simplies to *O(nlog n)*

3. Using the following class definitions:

   ```
   class Node {
       int key;
       String value;
       Node next;
   }

   class Hashtable {
       Node[] entries;
       int numvalues;
       float max_load_factor;
       public Hashtable(float max_load_factor) { ... } // constructor
   }
   ```

   Complete the following methods of the Hashtable class, using the hash function h(key) = key **mod** table_size.

   ```
   public void insert(int key, String value) {
      // COMPLETE THIS METHOD
   }
   ```

```
private void rehash() {
   // COMPLETE THIS METHOD
}
```

Note: When expanding the hash table, double its size.

SOLUTION

```
public void insert(int key, String value) {
    int index = key % entries.length();
    Node e = new Node();
    e.key = key;
    e.value = value;
    e.next = entries[index];
    entries[index] = e;
    numvalues++;
    float load_factor = (double)numvalues / entries.length;
    if (load_factor > max_load_factor) {
       rehash();
    }
}

private void rehash() {
    Node oldEntries[] = entries;
    int oldCapacity = oldEntries.length();
    int newCapacity = 2*oldCapacity;
    entries = new Node[newCapacity];
    numvalues=0;
    for (int i = 0 ; i < oldCapacity ; i++) {
        for (Node e = oldEntries[i] ; e != null ; e = e.next) {
            insert(e.key, e.value);
        }
    }
}
```

---

4. Suppose you are asked to write a program to count the frequency of occurrence of each word in a document. Desrcibe how you would implement your program using:
   1. A hash table to store words and their frequencies.
   2. An AVL tree to store words and their frequencies.

   For each of these implementations:
   1. What would be the worst case time to populate the data structure with all the words and their frequencies?
   2. What would be the worst case time to look up the frequency of a word?
   3. What would be the worst case time to print all words and their frequencies, in alphabetical order of the words?

   Assume there are *n* distinct words in the document, and a total of *m* words, and *m* is much greater than *n*.

   SOLUTION

   Implementation Process:

   1. Hash table: For each word, hash and search in the chain at that location. If word already exists, then increment frequency by 1, otherwise add it with frequency 1. (So key is word, and value is frequency.)
   2. AVL tree: Tree ordering is alphabetical on words. (In other words, search compares words, and each node in the tree has a word and its frequency.) For each word, search for it in the tree. If it exists, increment its frequency by 1, otherwise add it with frequency 1.

   Running times:

   1. Populating the data structure

      1. Hash table:

         In the worst case, all words hash to the same location, giving a chain of length *n*. In order to push the running time to the worst possible, the scenario to consider is when the first *n* words in the document are the distinct words, and the following *(m-n)* words are duplicates of the first *n*

         To add the first *n* words would take time:

         ```
         1 + 2 + 3 + ... + n = O(n^2)
         ```

         Since a word is first searched to see if it exists, and a worst case search will be all the way down the chain, the first and add takes 1 unit of time, the second takes 2 units, etc.

         The subsequent *m-n* words would each be searched on a chain of length *n*, with the worst case search running all the way through the chain, for *n* units of time. So this would amount of a time of *(m-n)*n*. Since *m* is much greater than *n*, this amounts to *O(mn)* time.

         So the total time is:

         ```
         O(n^2) + O(mn)
         ```

         and again, since *m* much greater than *n*, this simplifies to *O(mn)*.

      2. AVL Tree:

         With the same scenario for the worst case as above (distinct words all up front):

         The first *n* inserts would take time:

         ```
         log 1 + log 2 + log 3 ... + log n = log (n!)
         ```

         There's a math quantity called Stirling's formula that says *n*! is approximately equal to *(n/2)^(n/2)*:

         ```
         log (n!) is approximately = log ((n/2)^(n/2)) = O(n log n)
         ```

         The subsequent *(m-n)* searches would in the worst case take *O(log n)* time each, and since *m* is much greater than *n*, this results in *O(m log n)* time

   2. Looking up the frequency of a word
      1. Hash table: O(*n*), since the longest chain is of length *n*
      2. AVL Tree: O(log *n*), since the height of the tree is O(log *n*)

3. Printing words,frequencies in alphabetical order of words
   1. Hash table: Since there is no relative ordering of the words in the hash table, all the words need to be retrieved (with their frequencies), then sorted. The retrieval will take O(*n*) time, and the sorting will take O(*n*log *n*) time (using mergesort, for example). So the total time is O(*n*log *n*).
   2. AVL tree: Since the tree is already ordered alphabetically by words, an inorder traversal will do the trick, in only O(*n*) time.