1. Quantitative Analysis

A. After finishing your implementation of the HashMap and TrieMap, click run on the TestHarness.java. Note: this will take a few minutes to run on your computer. We recommend that you take a brief walk outdoors in the meantime, or sword-fight a friend. Now please copy and paste the output of TestHarness.java as an answer to this question, and place it inside of a verbatim environment.

B. For dictionary.txt, which implementation had a better running time? Which implementation had better space usage? What about for phonenumbers.txt? Was this what you were expecting? Why or why not?

C. It was mentioned in lecture that tries are more space-efficient than hash tables due to the fact that they compress common prefixes. Did your implementation reflect this for dictionary.txt? What about for phonenumbers.txt? If so, why? If not, what could you potentially do to improve the memory consumption of your TrieMap?

D. Based on your answer to 3, does Big-Oh notation tell us anything about the actual running time or space usage of an algorithm on a data set? Why or why not? What might an implication of this be for software development?

E. Add a call to initChildren() to the end of the Node constructor in TrieMap.java, then re-run the test harness, and consider the results for TrieMap and dictionary.txt. The difference here is that now we initialize the children array as soon as the node is constructed, instead of waiting until we actually add a child (the "lazy" way). How much memory, both absolutely and relatively, does the lazy initialization save us? (Give actual numbers.) Would you say the lazy initialization is a worthwhile optimization?

Solution.

A. Provide output.

B. Give accurate answers about runtime and space usage.

C. Explanation

D. No. Big-Oh notation provides the upper bound. In practical cases the average running time may be faster.

E. Provide absolute and relative memory calculations.
Suggested rubric. Two points for each subquestion. For B-E, one point for correct answer and one for explanation.

2. Hacking a Hash Table  The insert and search operations on a hash table run in constant time on average. However, the worst case runtime for these operations is $O(N)$. Describe a way that you could maliciously construct keys in order to force the hash table into $O(N)$ runtime for your keys. This could cause a denial of service attack if the hash table were being used in a web server like Apache's Tomcat. In your description please include the following details (more details are fine):

- What assumption does the constant time runtime rely on? What do your maliciously constructed keys do to that assumption?
- What information would you need to know in order to effectively hack the symbol table?
- Would your attack hurt the performance of a separate chaining implementation of a hash table, or a linear probing implementation, or both? Why?
- What alternate symbol table data structure could the designer replace the hash table with in order to thwart your attack? Why would that work? Would it cause different average runtime behavior for non-malicious calls to the new symbol tables insert and search operations?

Solution  In order to hack a hash table, we can maliciously construct distinct keys that all have the same hash value. If we insert $N$ such keys into the hash table they will all map onto the same place. This will result in a chain of length $N$ in a separate chaining implementation, and will require $N$ probes for each insertion into a linear probing implementation. This causes the worse-case $O(N)$ performance for Hash Tables.

- Hash tables rely on the uniform hashing assumption for the constant time operation of their insert, search and delete operations. Under the uniform hashing assumption we assume that each key is equally likely to hash to an integer between 0 and $M - 1$, where $M$ is the size of the array used for separate chaining or for linear probing. Our malicious keys break the uniform hashing assumption, since they are constructed to all go to the same location.
- In order to construct the malicious keys, all we need to know the how your hash function is implemented. For standard Java classes this can be done straightforwardly by reading Java API.
- The attack hurts the performance of both the separate chaining implementation and the linear probing implementation. It will result in a chain of length $N$ in a separate chaining implementation, and will require $N$ probes for each insertion into a linear probing implementation.
To prevent the attack, the designer could replace the Hash Table with a Red Black Tree. It has guaranteed worse-case performance of $O(\log N)$. Its average runtime performance is also $\log N$, versus the Hash Table’s constant time, so it would be slower for non-malicious use.

**Suggested rubric.** 2 points for each subquestion - one point for correct answer, one for justification.

Points for the following:

- 2 points:
  - Stating that we rely on a uniform hashing assumption for good performance in Hash Tables.
  - Saying that maliciously constructed keys will output identical hashCodes and thus hash to the same location.

- 1 point:
  - Saying that we must know the hash function implementation

- 2 points:
  - Saying that this affects both the separate chaining implementation and the linear probing implementation of Hash Tables.
    - Explanation

- 2 points:
  - Saying that the attack could be prevented with a Red Black Tree (or BST)
  - Correctly remembering that the average runtime of a RBT is $\log N$ so it will be worse than a Hash Table in non-malicious settings.

3. **Hash tables versus TSTs** When might we want to use a hash table instead of a ternary search trie?

**Solution**

- Hash Tables support keys of various types as opposed to TST’s, who’s keys have to implement Comparable.

- If you expect to do many searches for keys that are likely to be already stored (e.g. looking up names in a directory), and $N/M < w$.

**Suggested rubric.** 4 points for either correct answer (2 points for answer, 2 for justification).

4. **R-way trie with a linked list** What would happen if we had an R-way trie where we keep our next nodes in a linked list? How would this affect memory performance? Timing? What if we used an LLRB? A hash table?
Solution

- **Linked list.**
  
  Memory performance would improve in most cases, as we would need only store links for each character that is actually part of a key. Cases where a linked list would *not* improve memory are (a) if $R$ is very small (as the overhead of storing pointers would outweigh the benefit of storing only present key characters), or (b) if the trie is saturated.

  Time for any operations that involve search (i.e. get, put, etc.) would increase by a factor of up to $R$ (where $R$ is the length of the alphabet) to account for the time to traverse each linked list.

- **LLRB.**

  As with the linked list, memory performance would improve in most cases as each LLRB must be only as large as the number of characters that are part of a key.

  Time for any operations involving search would increase by a factor of up to $\log R$, to account for the time to search for the next character of a key in the LLRB.

- **Hash table.**

  Memory performance could improve if the height of the hash tables, $M$, is less than the size of the alphabet, $R$.

  If we maintain a cache of the hash value for each character, then the time for each trie operation involving search would increase by $w$ (one hash look-up for each character).

**Suggested rubric.** 6 points total; 2 per data structure (one for memory, one for timing).

5. **Making a hash of strings**  Consider modular hashing for string keys with $R = 256$ and $M = 255$. Show that this is a bad choice because any permutation of letters within a string hashes to the same value.

  In class we used $R = 31$:

  ```java
  public final class String {
      private final char[] s;
      ...
      ...
      public int hashCode() {
          int hash = 0;
          int R = 31;
          for (int i = 0; i < s.length(); i++)
              hash = s[i] + R * hash;
  ```
```java
    return hash;
  }

  public int hash()
  {
    return Math.abs(hashCode()) % M;
  }
}
```

**Solution**  Assume we have some length-\(l\) string \(s = [s_0, s_1, \ldots s_{l-1}]\). Then \(s\).hash() resolves to:

\[
s.hashCode() = [s_0 \cdot 256^l + s_1 \cdot 256^{l-1} + \cdots + s_{l-1} \cdot 256^0] \mod 255 \\
= [s_0 \cdot 256^l \mod 255 + s_1 \cdot 256^{l-1} \mod 255 + \cdots + s_{l-1} \cdot 256^0 \mod 255] \mod 255 \\
= [s_0 + s_1 + \cdots + s_{l-1}] \mod 255
\]

since 256 mod 255 = 1 and 256\(^k\) mod 255 = 1\(^k\).

Because the addition is commutative, the original string can have the characters in any order and produce the same resulting hash code.

**Suggested rubric.**  4 points.