UNIVERSITY OF VIRGINIA, DEPARTMENT OF COMPUTER SCIENCE

Lab 08 - Hash Tables

Dr. Mark R. Floryan

October 18, 2018

1 PRE-LAB

This week, you will be building a custom hash table class, and using it to perform a word search. The overview of this prelab is as follows:

1. Download the provided starter code.
2. Implement the HashTable.java class.
3. Test the correctness of your hash table using the provided tester.
4. Implement the word search solver by using your hash table implementation.
5. Optimize your hash table to run as quickly as possible on the large word search grids provided
6. **FILES TO DOWNLOAD:** hashing.zip
7. **FILES TO SUBMIT:** lab08.zip

1.1 HashTable.java

First, you will implement a hash table class. This class has two generic arguments K (the key) and V (the value). To handle these, we will have our table internally store HashNode objects. This code is provided to you and a summary is shown below:

```java
public class HashNode<K,V> {
    private K key; private V value;

    //Constructor
    public HashNode(K key, V value);

    //HashNodes are "equal" if both keys are equal
    public boolean equals(Object other) {
        return this.key.equals(((HashNode<K,V>)other).key);
    }

    public K getKey() {return this.key;}
    public V getValue() {return this.value;}
    public void setValue(V newValue) { this.value = newValue; }
}
```

Notice that two HashNode objects are considered to be "equal" to each other if the key in each is equal. Thus, you can use the provided HashNode.equals() method to examine equality between HashNode objects directly.

Using this object, you will implement the HashTable.java class which implements the Map interface shown below. You may use *any collision resolution strategy* you would like when implementing this class.

```java
public interface Map<K,V> {

    public void insert(K key, V value);
    public V retrieve(K key);
    public boolean contains(K key);
    public void remove(K key);
}
```

Once you have implemented your HashTable class, there is a provided tester that will test some of the operations of the table to ensure they work. **This tester IS NOT a thorough test, but rather a sanity check that the table seems to be working**. You should write your own tests as well to ensure your table is working correctly.

1.2 Word Search Solver

Next, you will be writing some code within the WordPuzzleSolver.java class. This class reads in a two-dimensional array of characters, and looks for valid English words within the puzzle. (see Wikipedia for more details). You will be provided with two input files, a **dictionary text file** and a **grid text file**. The dictionary file enumerates a list of valid English words. Your goal is thus to find each of these words in the word puzzle. The grid text file specifies the number of rows and cols in the grid and then enumerates the characters of the grid all on a single line.

Some other notes / requirements about this part of the lab:

1. You should instantiate your hash table and insert each of the words from the dictionary. In order to this, you will need to read input from a file. See the course java cheat sheet for an example of how to read from a file.

2. You need to check words that start at ANY starting location in the grid, moving in any of the eight directions (North, NorthWest, etc.), and check any words of length $3 \leq n \leq 22$. Make sure you do not move off the end of the grid when looking up words.

3. Your output should match the format shown in the provided starter project under *input/3x3.out.txt*. The order the words are printed is not important, but would be nice to make grading easier.

4. the *input/* folder provides several grids and dictionaries to test your code with. Your code should be correct for all combinations and also be reasonably fast (less than 10 seconds on the largest grids). How fast can you make it?

2 In-Lab

The goal of this in-lab is to continue practicing with hash tables in a laid back environment. As usual, you will:

1. Get into small groups of two
2. The TAs will present you with some programming challenges regarding trees. These will not be graded, but you are required to attend and to participate.
3. You may think about the challenges before lab below if you'd like, but we highly recommend that you not solve them ahead of time.
4. The TAs will give you time to solve each problem and lead you in sharing solutions with one another.

2.1 Hash Tables

The TAs will lead you in going through the following challenges. It is ok if you do not get through each of these.

1. Write a method that takes two String parameters $s1$ and $s2$. The method returns true iff $s2$ and $s1$ are anagrams of one another. Your method should use a hash table, must be $\Theta(n)$, and can only loop over each String at most once.
2. Write a method that takes in an array of ints as a parameter. Your method should return true iff there exists four distinct elements in the array such that a, b, c, and d such that $a \neq b \neq c \neq d$ and $a + b = c + d$. Your solution must be $O(n^2)$.
3. Consider a list of n integers (passed as a parameter) and an integer k (also passed by parameter). Write a method that returns a list of all contiguous windows in the array of exactly k adjacent elements AND also prints out the number of unique integers in each window. Your solution should be $O(nk)$.

If you have coded up all of these and they work, then show the TA and you may leave early.

3 Post-Lab

The goal of this post-lab is to produce a report analyzing optimizations to your hash table implementation.

You will perform some experiment(s) by doing the following:

1. Experiment 1: Implement more than one of the collision resolution strategies shown in class. For each, count the number of collisions that occur and also record the time on various input to your word puzzle solver. Report which strategies had the most/fewest collisions, and how slow/fast the strategies were.

2. Experiment 2: Brainstorm and implement some kind of optimization to your hash table / word puzzle solver. Describe your optimization and how it works. Report how much faster it makes your word puzzle solver run compared to the best approach from experiment 1 above. How fast were able to make your code run?

3. **FILES TO DOWNLOAD:** None

4. **FILES TO SUBMIT:** PostLabEight.pdf

3.1 Report

Summarize your experiment and your findings in a report. Make sure to adhere to these general guidelines:

- Your submission MUST BE a pdf document. You will receive a zero if it is not.

- Your document MUST be presented as if submitted to a professional publication outlet. You can use the template posted in the course repository or follow Springer's guidelines for conference proceedings.

- You should write your report as if it is original novel research.

- The grammar / spelling / professionalism of this document should be sound.

- When possible, do not use the first person. Instead of "I ran the code 60 times", use "The code was executed 60 times...".

In addition to the general guidelines above, please follow the following rough outline for your paper:

- **Abstract**: Summarize the entire document in a single paragraph
- **Introduction**: Present the problem, and provide details regarding the algorithms you implemented (especially the "hybrid" algorithm).
- **Methods**: Describe your methodology for collecting data. How many executions, which inputs, when did the timer start/stop, etc.
- **Results**: Describe your results from your execution runs.
- **Conclusion**: Interpret your results. Which algorithm/approach was fastest in each situation? Did the fastest algorithm change? If so, why? Does the performance you see match the theoretical runtimes of the algorithms? Why or why not?

Lastly, your paper MUST contain the following things:

- A table (methods section) summarizing the experiments and how many execution runs were done in each group.
- At least one table (results section) summarizing the results of all of your experiments.
- Some kind of graph visualizing the results of the table from the previous bullet.