CSE 332: Data Structures and Parallelism

Exercise 5 - Spec

Exercise 5 Spec (25sp)

The objectives of this exercise are

- **Implement the mechanics of an efficient chaining hash table**
- **See how to write a good hash function for a new data structure**
- **Apply the advantages of a hash table by using it as part of an algorithm that involves a large number of inserts and finds of a large dictionary.**

Overview

This exercise consists of the following parts:

1. Complete an implementation of a separate chaining hash table data structure.
2. Write a good hash function for the Word class provided.
3. Use your implementation of the hash table to complete an implementation of a word search algorithm, which searches through a grid of items for sequences that appear in a given dictionary.

Parts 2 and 3 rely on each other, but part 1 can be done independently if you temporarily refactor some code to use Java's HashMap instead of your own separate chaining hash table implementation. If you want to go this route for debugging, I'd recommend writing a "wrapper" class the implements the DeletelessDictionary interface, but simply uses the java HashMap class as the underlying data structure (or if you implement a contains method, you can use your AVL implementation from exercise 4!).

Motivating Application: Word Search

For a newspaper word search puzzle, you're given a grid of letters and a list of words. Your goal is to find all of the given words within the grid. The challenge is that the words do not necessarily appear in left-to-right order. Instead, the words can go in any of 8 directions: left-to-right, right-to-left, vertically down, vertically up, or any of 4 directions diagonally (up-left, up-right, down-left, down-right).

For this assignment, in addition to implementing a separate chaining hash table, you will be completing an implementation of a word search solver. For this word search solver we use two text files. One containing our grid of letters, the other containing a list of potential words. This solver will use these slightly differently from the puzzles you might find in the newspaper. In the newspaper it is guaranteed that all words in the list appear somewhere in the grid. For this application our task will be to determine *which* of the words appear. Additionally, we're implementing the algorithm using generics, so rather than a grid of letters we could instead do a wordsearch on a grid of integers, or doubles, or booleans, or pixels, or any other object we could dream of!

You'll see in the starter code and in the list of provided files below that we give you two pairs of puzzles. One that is a small grid paired with a relatively short list of words (this is actually the list of words used by xkcd author Randall Munroe for his Thing Explainer book, but with the swear words removed). The other is a larger grid paired with a large list of words (this list of words is the official scrabble dictionary).

Implementation Guidelines

Your implementations in this assignment must follow these guidelines

- You may not add any import statements beyond those that are already present (except for where expressly permitted in this spec). This course is about learning the mechanics of various data structures so that we know how to use them effectively, choose among them, and modify them as needed. Java has built-in implementations of many data structures that we will discuss, in general you should not use them.
- Do not have any package declarations in the code that you submit. The Gradescope autograder may not be able to run your code if it does.
- Remove all print statements from your code before submitting. These could interfere with the Gradescope autograder (mostly because printing consumes a lot of computing time).
- Your code will be evaluated by the gradescope autograder to assess correctness. It will be evaluated by a human to verify running time. Please write your code to be readable. This means adding comments to your methods and removing unused code (including “commented out” code).

Provided Code

Several java classes have been provided for you in this zip file. Here is a description of each.

- `DeletelessDictionary`
  - A dictionary interface. This differs from the one discussed in class in the following ways:
    - It does not have a delete operation (hence being called a "deleteless" dictionary). This means that once a key-value pair has been added to the dictionary there is no way of removing that key later.
    - It requires the operations `getKeys` and `getValues` which return a list of the keys or values in the dictionary. These lists must be index-aligned with one another. By this we mean that index 0 of `getKeys` is associated with the value at index 0 of `getValues`. More generally, `getKeys().get(i).equals(getValues().get(i))` should always be true.
    - It requires the operation `contains` which returns true or false to indicate whether the given key is present in the dictionary.
  - *Do not submit this file*
- `ChainingHashTable`
  - When you download the zip, this will contain the beginnings of an implementation of a separate chaining hash table. We have implemented a constructor as well as the isEmpty and size methods. You're welcome to change any of those if you'd like. You may also add fields to this class if you wish.
  - To help you with debugging, we have provided a toString method. We will not use this method in the autograder, so you're welcome to change that if you wish as well.
  - We've also provided an array of primes that will be useful for rehashing. Your implementation must be able to rehash to a size beyond these pre-compute primes. There's more guidance on how to do this in the specification for `insert` below.
  - *You will submit this file to Gradescope*
- `Item`
  - This class just encapsulates a key-value pair for adding to the chaining hash table (think of it as playing a similar role as the nodes did in AVL trees). Overall, the chaining hash table will be an array of linked lists of these items.
  - *Do not submit this file*
- `Word`
  - This class is intended to serve sort of like a generic form of a string. Essentially it is just an array of some generic type T that's encapsulated in an object (you can think of it like a fixed-sized array list).
  - We've added some methods, though, to make things a bit easier for the word search application. Some important but routine operations have been provided, including a toString and equals method (neither of which you should change in your final submission).

    - Most importantly here we've added a really handy constructor. You can construct a Word by either giving it an array, which it then just makes a shallow copy of, or by giving it a grid along with the start cell, length and direction. This latter constructor will shallow-copy all indices of the grid starting from the given cell and then preceding the indicated direction (any of the 8 valid directions for a word search).
    - The only thing you need to do in this class is implement the hashCode method (see details below). You should not change anything else.
    - *You will submit this file to Gradescope*
- `WordSearch`
    - This is the class that actually does the word searching. We've provided the trickiest part of the code for you (the part that actually navigates the grid, that method is called `wordSearch`). We've just left out the parts that actually use the hash table.
    - To assist with debugging we added a method called `printFoundWords` that prints out all of the words found within the grid.
    - *You will submit this file to Gradescope*
- Client
    - This class contains the main, which is what will make the word search happen! The main method invokes methods which will read the grid and dictionary (i.e. word list) files, then creates a WordSearch object with them, then does the word search! It currently does this for both of the small and large puzzles. To check correctness it only checks if the number of words found is correct, but does not check that the actual words were the correct ones.
    - Currently there are no tests provided specifically for the ChainingHashTable. The expectation is that you obtained experience on how to verify dictionary behavior from Exercise 4.
    - *Do not submit this file*
- Various text files
    - These text files contain the information associated with each example puzzle. The ones whose names begin with the prefix "small" are associated with the small test. The rest with the big test. Each test has a "puzzle" file with the grid, a "words" file with the list of valid words, and a "solution" file which contains all of the words that your algorithm should find.

Part 1: ChainingHashTable

To obtain good performance of our `DeletelessDictionary` we will implement a separate chaining hash table data structure according to the comments provided for each method in the interface. You should finish all non-yet implemented methods in the `DeletelessDictionary` interface.

It will be helpful to know, for this assignment, that all objects in Java have a `hashCode` method. This method returns an integer that can then be used in various ways, such as selecting an index in a hash table! When implementing the `ChainingHashTable` class, you may assume that this `hashCode` method has been implemented to be "good" for the type that the data structure will contain. The integer given, though, is not guaranteed to be in the range of your underlying array.

Here is the list of methods you must implement, along with any guidelines for implementation:

- `find`
    - This behavior should pretty much just match how we described it in class. You give it a key, it returns the associated value (if it exists)
- Contains
    - This one is new, but the algorithm is almost exactly the same as find, but can be convenient in some circumstances. You give it a key, it returns true or false to indicate whether it appears in the dictionary at all.
- Insert
    - This behaves just like we discussed, but we'll review here. You give it a key and a value, it then pairs together that key and value in the dictionary. If the key already had some associated value, it should return the old value. If the key was not already present, it should return `null`.
    - Most importantly, here, you will need to resize your array and rehash the items when the load factor gets too high. The size should be chosen to be a prime number from the provided list of pre-computed primes as long as possible. Once you run out of primes, you should use a different method for selecting the next size of the array (keeping sure that the running time is still constant amortized!). As a tip, numbers that are 1 more than a power of 2 tend to behave a lot like primes (and have a higher probability of being primes themselves)!
- getKeys
    - Returns a list of all keys in the dictionary
- getValues
    - Returns a list of all values in the dictionary. The order of this list should parallel the list returned by `getKeys`. That is, the value at index `i` should be associated with the key at index `i` of `getKeys`.

Part 2: Word.hashCode()

The only way to obtain good performance from hash tables is to use a good hash function. In Java, `Object` has a method called `hashCode` that serves as the default hash function. As with the default behavior of other `Object` methods like `equals` and `toString`, the default `hashCode` is almost certainly not going to do what we want, so we'll need to override it.

Write your own implementation of the hashCode method for the Word class to be a "good" hash function. Keep in mind the properties of a "good" hash function that we discussed in class. You may assume that the objects that the Word contains themselves have a "good" hash function implementation.

Beware! When Java integers exceed a certain size they may become negative! This is called integer overflow. You might find Java's `Math.floorMod()` helpful for dealing with negative hashes.

Part 3: WordSearch

Now for the main event – the `WordSearch` algorithm! For this part you will finish our implementation. Rest assured, though, the hardest parts are provided for you. The only things you need to implement are the parts that actually use the dictionary data structure!

The most important fields for your pieces are a dictionary and a 2-d array. The dictionary (which will be your `ChainingHashTable` implementation) maps `Word` objects to `booleans`. Initially, it will contain all words from the list of words as keys, with each key associated with the value `false`. This dictionary is used to keep track of which words from the dictionary have been found in the grid (which is represented by the 2-d array).

The `WordSearch` constructor does the following:

- Its parameters are a two dimensional array to serve as the grid to search in and a list of `Word` objects to indicate the dictionary of words. In this context we mean "dictionary" as in a list of words in some language, e.g. Webster's Dictionary.
- It constructs a `ChainingHashTable`. In other words, it creates an instance of the class you implemented in Part 1.
- It adds each `Word` in the input list of `Word` objects as a key in the `ChainingHashTable`, with its value being `false`. The value indicates whether the `Word` was found in the grid, so before searching we haven't seen it yet.
- It calls the `wordSearch` method, which looks for each `Word` within the grid and updates the value associated with each `Word` in the `ChainingHashTable` to be `true` if it appears.

You need to implement the following methods:

- `addIfWord`
    - The argument to the method will be a `Word` object. This `Word` object will be a sequence found by the `wordSearch` method. This method should update the

dictionary so that, if the given word is present, it is associated with the value true to indicate it has been found in the grid. If the given `Word` is not already a key in the dictionary then this method should not modify the dictionary.

- `countWords`
    - This method should return the number of words from the list of valid words that were found in the grid.
- `getWords`
    - This method returns a list of all of the words that were found within the grid. There is no particular requirement about the order that the words should appear.