Vitaliy Shydlonok

CSC365: Programming Assignment 1

Project Report

## Introduction:

The objective of this project is to implement the first part of a sentiment analysis program: feature extraction. This assignment implements a function that takes a dictionary of words and a text document and counts the number of occurrences of dictionary words within the text document. In the process of completing this project, I learned how to store large amounts of data in a way that is easily searchable. I ended up going with a custom hash table implementation where I hash words based on the first character and put them in a doubly-linked list located at the hashed index in an array.

## Implementation:

The core of this assignment was my implementation of a FeatureVector class. I decided to store both the dictionary and the review words into separate instances of this class. The FeatureVector class contains an array of linked lists; this array is of size 26, enough to fit all 26 letter of the alphabet. I give the FeatureVector a word and an optional value. Then it adds the word to a linked list at the array location associated with the first letter of the word or increments its value if the word is already in the FeatureVector.

I implemented the extractWordFeatures() function by first loading the entire dictionary file into a FeatureVector, keeping each word phrase and the semantic value of the phrase. Then I read the review line by line, checking to see if the next three, two, or one word phrase exists in the dictionary FeatureVector. If it does exist, I add it to another FeatureVector, incrementing its count if it already exists in the review FeatureVector. Finally, the resulting review FeatureVector is returned.

## Theoretical Analysis:

Resizing an array by doubling:

Resizing an array by doubling requires creating a new array that's twice the size and then copying N elements from the previous array to the new one. The doubling only happens when the array is full, with the time between doubling increasing by a factor of two every time the array is doubled. This means that the big-O complexity of doubling the size of an array is O(log(N)).

Resizing and array by incremental increase:

Resizing an array by an incremental increase means creating a new array that is only one element larger than the previous array. All elements from the old array must be copied over, and this happens every time an element is added once a certain N is reached. This means that the big-O complexity of incrementally increasing an array size is O(N).

Pushing back elements on a list:

Pushing back elements of a list is a big-O of O(1) because the time to do that is constant. No matter the number of elements within the list, pushing back the elements takes constant time.

Inserting/erasing elements in a list and vector:

Inserting and erasing elements in a list requires finding the position of the insertion, then modifying the links at that position. This takes at most N operations, giving it a big-O of O(N). Inserting and erasing elements in a vector also means going through at most N elements to find the position and then N operations to shift the array. This means 2N operations, which is a big-O of O(N).

Implementation of a stack (list vs vector):

A stack implemented as a list or a vector has the same big-O when pushing and popping, if the vector never has to be resized. If the vector does need to be resized, it can have a big-O complexity of O(log(N)) by doubling or O(N) by incremental increase. The list will always have a big-O of O(1) because to push or pop to the list always only modifies the reference to the head of the list.

Implementation of a queue (list vs vector):

A queue implemented as a list has a big-O complexity of O(1) if the references to the head and tail are saved, otherwise to dequeue from a list queue would take a big-O of O(N) to reach the end of the list. A vector implementation of a queue has big-O complexity of O(N) on a dequeue because the whole array must be shifted to the front. On an enqueue, a vector implementation has a big-O of O(log(N)) or O(N) because the access to the end of the vector is constant but it has to double or increment its size.

## Experimental Setup:

My machine is running 64bit Windows 10, Octa-core AMD Ryzen 7 2700X processor, 16GB of 3000MHz DDR4 ram, reading from a 500GB Samsung 860EVO SSD.

## Results & Discussion:

A push back on a vector can be done by simply adding an element to the end, which means this operation can be either O(log(n)) or O(N) depending on whether it is doubled or incremented on a resize. A push back on a list is always constant time so a big-O of O(1). The plots start out with the vectors having lowest times and the list having the highest time, but after a certain threshold, the list has the lowest time, the doubling vector in the middle, and the incremental vector the highest.

A pop from a stack implemented using a list or vector both take constant time, a big-O of O(1). But the actual time that it takes for a list to pop an element can be higher than the time it takes for a vector because deleting the last element of a vector is one operation but deleting the head of a list can be two operations: one to remove the previous link of the second element and one to switch the head pointer to the second element.

With a queue, using a list to en-queue is O(1) and to de-queue is either O(1) or O(N) depending on whether the tail is saved. Using a vector to en-queue is O(log(N)) or O(N) depending on doubling or increment, and to de-queue is O(1). On the plot, the times to de-queue are the same for the list and vector, but the vector has lower times to en-queue.

**Conclusion:**

The most time efficient method of storing a lot of unordered data is by using a list, because pushing back the list is constant time. The drawback of using a list instead of a vector is space efficiency. Lists have a lot more overhead than vectors because they have to store one or two pointers for every element in the list.

**Code:**

```
import java.io.*;

public class Analyzer {

    public FeatureVector extractWordFeatures(String reviewPath, String dictPath)
{

        FeatureVector features = new FeatureVector();
        FeatureVector dictionary = new FeatureVector();

        try {
            File reviewFile = new File(reviewPath);
            File dictFile = new File(dictPath);

            // Read the whole dictionary file
            BufferedReader dictReader = new BufferedReader(new
FileReader(dictFile));
            String dictLine = dictReader.readLine();
            while(dictLine != null) {

                // Split each line into tokens
                String tokens[] = dictLine.split("[ \t\n]");
                String word = tokens[0];
                int value = 0;

                // Add all words in the dictionary phrase
                for(int i=1; i<tokens.length; i++) {
                    // Make sure that the token isn't a semantic value
                    try {value = Integer.parseInt(tokens[i]); }
                    catch (NumberFormatException ex) {
                        word += " "+tokens[i];
                    }
                }

                dictionary.add(word, value);
                dictLine = dictReader.readLine();
            }

            dictReader.close();

            //System.out.print("Dictionary ");
            //dictionary.print();
            // Read the review
            BufferedReader reviewReader = new BufferedReader(new
FileReader(reviewFile));
            String reviewLine = reviewReader.readLine();
            while (reviewLine != null) {
                String[] tokens = cleanString(reviewLine).split("[ \t\n]");

                for (int i=0; i<tokens.length; i++) {
                    // Make sure the token isn't empty
                    if (tokens[i].length() > 0) {
                        // Check if three word, two word, or one word phrase
exists in the dictionary
                        if (dictionary.get(tokens[i]) != null) {
                            if (i < tokens.length-1 &&
dictionary.get(tokens[i]+" "+tokens[i+1]) != null) {
                                if (i < tokens.length-2 &&
dictionary.get(tokens[i]+" "+tokens[i+1]+" "+tokens[i+2]) != null) {
                                    features.add(tokens[i]+" "+tokens[i+1]+"
"+tokens[i+2]);
                                    i+=2;
                                } else {
                                    features.add(tokens[i]+" "+tokens[i+1]);
                                    i++;
                                }
                            } else {
                                features.add(tokens[i]);
                            }
                        }
                    }
                }
                reviewLine = reviewReader.readLine();
            }

            reviewReader.close();

            System.out.print("\nReview ");
            features.print();

        } catch (Exception ex) {
            ex.printStackTrace();
            System.exit(-1);
        }

        return features;
    }

    private String cleanString(String str) {
        String result = str.toLowerCase();
        result = result.replaceAll("[^a-z\\d\\s]+", "");
        result = result.replaceAll("  ", " ");
        return result;
    }
```

```java
private class FeatureVector {

    private Feature[] alphabet = new Feature[26];

    public void add(String word) {
        add(word, 1);
    }

    public void add(String word, int value) {
        int pos = word.toLowerCase().charAt(0)-'a';

        // Look an existing feature with that name
        boolean found = false;
        Feature node = alphabet[pos];
        while(node != null && !found) {
            if (node.word.equals(word)) {
                node.count++;
                found = true;
            } else {
                node = node.next;
            }
        }

        if (!found) {
            // Check if the list is empty
            if (alphabet[pos] == null) {
                alphabet[pos] = new Feature(word, value);
            } else {
                // Link a new feature to the front of the list
                Feature addition = new Feature(word, value);
                addition.next = alphabet[pos];
                alphabet[pos].prev = addition;
                alphabet[pos] = addition;
            }
        }
    }

    public void remove(String word) {
        int pos = word.toLowerCase().charAt(0)-'a';

        // Look for the word
        Feature node = alphabet[pos];
        while(node != null && !node.word.equals(word)) {
            node = node.next;
        }

        // Remove the word from the list of features
        if (node != null) {
            if (node.prev != null)
                node.prev.next = node.next;
            if (node.next != null)
```

```
                node.next.prev = node.prev;
            if (node == alphabet[pos])
                alphabet[pos] = node.next;
        }
    }

    public Feature get(String word) {
        int pos = word.toLowerCase().charAt(0)-'a';
        Feature node = null;

        // Only search for words starting with an alphabetical character
        if (pos >= 0 && pos < 26) {
            // Look for the word
            node = alphabet[pos];
            while (node != null && !node.word.equals(word)) {
                node = node.next;
            }
        }

        return node;
    }

    public void print() {
        System.out.println("Feature Vector:");

        for (int i=0; i<alphabet.length; i++) {

            System.out.print((char)(i+'a')+"{");

            Feature node = alphabet[i];
            while(node != null) {
                System.out.print("("+node.word+":"+node.count+"),");
                node = node.next;
            }

            System.out.print("}\n");

        }
    }
}

private class Feature {
    public Feature next;
    public Feature prev;

    public String word;
    public int count;

    public Feature(String word, int value) {
        this.word = word;
        this.count = value;
    }
```

```java
    }

    public static void main(String[] args) {
        (new Analyzer()).extractWordFeatures("review.txt", "AFINN-en-165.txt");
    }
}
```