Problem Set 2 – Searching Algorithms

Jordan Stein

Introduction

I have implemented four different algorithms to simulate the runtime of searching/creating a concordance of every word in the complete works of Shakespeare with four different data structures. The four data structures used are a unsorted vector, a sorted vector, a binary search tree, and a hash table. The expected order of growth to create the concordance using the unsorted vector is O(N^2) because we must compare all of the elements against each other to calculate their occurrence. The expected order of growth of the sorted vector is O(N) because we can iterate through the vector and compare the elements with the next one to find the occurrence. The binary search tree has an expected order of growth of O(log n) because we can truncate through half of the tree for each recursive call to find the concordance. Finally, the hash tree has an expected growth of O(1) because we can access the elements directly once they are hashed into the associative array. I have implemented an assignment and comparison count for each algorithm used to compare the efficiency of each data structure against each other.

Problem Definition

The searching problem is defined as:

Input:

A sequence of words containing Shakespear's complete works from a file named "wordlist.txt".

Below is a sample from the beginning and end of the file.

```
A
MIDSUMMER-NIGHT'S
DREAM
Now
,
fair
Hippolyta
,
our
…
state
,
That
like
events
may
ne'er
it
ruinate
```

I have only read in words that are not blank and whose first character is alphabetic.

Output:

A concordance of every unique word found in the file, including an occurrence count for each word found. Each data structure used creates and stores this concordance.

I have analyzed the output of each algorithm, (unsorted vs sorted vs binary search vs hash table), by sorting the words to compare the efficiency of the algorithms against each other.

Part 1 – Pseudo-Code for each Algorithm

Unsorted Vector

Given an unsorted vector containing all of the words in Shakespear's complete works, I have molded the algorithm to compare every word against each other. To obtain the concordance, I created two new vectors. One will store every unique word and the other will store the count of that word. After algorithm is finished, the function returns the size of the uniqueWords vector. I also created a function called countStr which counts the occurrence of a string by iterating through the whole unsorted vector.

```
Int UniqueCount(Vector<String> allWords)
1   Vector<String> uniqueWords
2   Vector<Int> uniqueCount
3   for i to allWords size
4       if uniqueWords doesn't contain allWords[i]
5           uniqueWords.add(allWords[i])
6           uniqueCount.add(countStr(allWords[i], allWords))
7   return uniqueWords size
```

```
Int countStr(String str, Vector<String> allWords)
1   int count = 0;
2   for i to allWords size
3       if allWords[i] == str
4           count++;
5   return count
```

The UniqueCount function returns the count of all unique words in the allWords vector in O(N^2) time.

Sorted Vector

Given a sorted vector containing Shakespear's complete works, I have taken advantage of the fact that the words are sorted in ascending order and only compare the words with the next word in the vector. I created two new vectors to obtain the concordance with this algorithm as well. At the start of the algorithm, I create a count variable that will assign the count of each word. We will compare each word against the next one in the vector. If the word is not equal to the next, we will increment the count and store that word, along with the count in their respective vectors, then reset the count to zero. If the word is equal to the next, we will only increment the count.

```
Int sortedUniqueCount(Vector<String> allWordsSorted)

    Vector<String> uniqueWords
    Vector<Int> uniqueCount
    int count = 0
    for i to allWordsSorted size - 1
        if allWordsSorted[i] != allWordsSorted[i+1]
            count++;
            uniqueWords.add(allWordsSorted[i])
            uniqueCount.add(count)
            count = 0
        else count++;
    return uniqueWords  size
```

The sortedUniqueCount function returns the amount of unique words found in the given sorted vector in O(n) time.

Binary Search Tree

Given an unsorted vector containing Shakespear's complete works, I have thrown every element into a binary search tree data structure to create my concordance. Each node of the binary tree holds the word and the occurrence counter. When inserting a word into the three, I check if the word is already in the tree using a find function. If the word is not in the tree, we will insert it. Otherwise, we will increment the occurrence count for the word. I used the pseudocode for a binary search tree on code2learn.com's website to help me construct the tree.

http://www.code2learn.com/2013/02/binary-search-tree-bst-algorithm.html

Searching for a matching node.

```
1    Start at the root node as the current node
2    If the search key's value matches the current node's key then found a match
3    If the search key's value is greater than the current node's value
4        If the current node has a right child, search right
5        Else, no matching node is in the tree
6    If the search key's value is less than the current node's
7        If the current node has a left child, search left
8        Else, no matching node is in the tree
```

Node find(Node n, int value)

```
1    if (n.value == value || n == null)
2        return n;
3    Else if (value < n.value)
4        find(n.left, value)
5    Else find(n.right, value)
```

Inserting a value into the tree

```
1    Always insert new node as leaf node
2    If new node's key < current node's key
3        If current node has a left child, search left
4        Else add new node as current's left child
5    If new node's key > current node's key
6        If current node has a right child, search right
7        Else add new node as current's right child
```

Void insert (Node n, int value)

```
1    if (value < n.value)
2        if (n.left != null)
3            insert(n.left, value);
4        Else n.left = new Node(value)
5    If (value > n.value)
6        If (n.right != null)
7            insert(n.right, value)
8        Else n.right = new Node(value)
```

Searching and inserting a concordance into the binary tree is done in O(log n) time because half of the tree is truncated for each recursive call.

In order to obtain the occurrence count, I had edited my find function to increment the occurrence of the node if the node was already found in the tree prior to insertion. This happens as I am filling the tree to prevent duplicates. If the node is not found into the tree, it will be inserted. Otherwise, the occurrence of that word in the node will be incremented.

Below is the pseudo-code I used to implement this.

Int binarySearchTree(Vector<String> allWords)

```
1       binarySearchTree tree = new BinarySearchTree();
2       for i to allWords size
3               if !tree.find(allWords[i])
4                       tree.insert(allWords[i])
5       return tree.size
```

The binarySearchTree function creates the tree and fills is with all of the unique words and the occurrence of each word in each node. The size of the tree is incremented every time a word is inserted into it. I return that size from this function to receive the total amount of unique words.

Hash Table

Given an unsorted string array containing Shakespear's complete works, I have hashed every word and inserted it into an that associates the hashcode as the index to the word along with its occurrence. I create a function to create a hash key based off of the string passed into it, and another function to insert the word into the array based off of the key. I had used and edited the implementation I found on newthinktank.com to help me implement the hash table. The hashtable object has an array of linked lists named theArray, as well as an amount in table variable to keep track of how many elements have been inserted.

http://www.newthinktank.com/2013/03/java-hash-tables-3/

Pseudo-code for hashing function

We find the charCode for each element of the string and subtract 96 from it to make our letters start at 1 (because in ascii, 'a' == 96). We then calculate the hash key by multiplying it by 27. (26 alphabetical characters plus blank) and mod it by the size of the hash array.

int stringHashFunction(String wordToHash)

```
1       int hashKeyValue = 0
2       for i to wordToHash length
3           int charCode = wordToHash.charAt(i) – 96
4           int HKVTemp = hashKeyValue
5           hashKeyValue = (hashKeyValue * 27 + charCode) % arraySize
6       Return hashKeyValue
```

Pseudo-code for insertion into list.

I used a word class that acts as a linked list to hold the word and occurrence count. The first line in the pseudo code takes the word we wish to hash, generates a hashkey using the stringHashFunction, and inserts it into theArray at the hashkey's location.

Void insert(Word newWord)

```
1    String wordToHash = newWord.theWord
2    Int hashKey = stringHashFunction(wordToHash)
3    theArray[hashKey].insert(newWord, hashKey)
```

Pseudo-code for inserting the unsorted string array of Shakespear's works into the hash table array.

Every time I insert, I increment a local variable (amountInTable) in the hashtable object to store the total amount of unique words. Once all values are inserted, I will have the total count of all unique words. This function generates the hashkey for every word in the string array and checks if the value is in the hash array at the hashkeys index. If that index is equal to null, it is not in the table so we will insert it. Otherwise, we will increment the occurrence count in the node at that location in the hashtable array.

Void addTheArray(String[] elementsToAdd)

```
1    String word = elementsToAdd[i]
2    int hashKey = StringHashFunction(word)
3    for i to elementsToAdd length
4        if (theArray[hashKey.find(hashKey, word) == null
5            Word newWord = new Word(word,1)
6             Insert(newWord)
7            amountInTable++
8        Else theArray(hashKey.find(hashKey, word).increment()
```

Pseudo-code for finding an element in the hash array.

This code generates the hashkey of the string passed into it, accesses that index in the hasharray, and returns the word node that contains the word and the occurrence count in O(1) time.

Word find(String wordToFind)

```
1    int hashkey = StringHashFunction(wordToFind)
2    Word theWord = theArray[hashkey.find(hashkey, wordToFind);
3    Return theWord
```

Part 2 – Implementation

I have implemented all four algorithms in Java. You are able to choose which algorithm you wish to run at the start of the program. The unsorted and sorted vector algorithms were short, so I included all of their work on the SearchingAlgorithms.Java file. I used a different file to create a class for the binary search tree and the hash table. The binary search tree is available in BinarySearchTree.java and the hash table is in HashFunction.java I had used and edited a binary search tree and hash table implementation I had found online to work with my data.

The binary search tree implementation I used is available at: http://algorithms.tutorialhorizon.com/binary-search-tree-complete-implementation/

The hash table implementation I used and edited is available at: http://www.newthinktank.com/2013/03/java-hash-tables-3/

SearchingAlgorithms.java

```
/* Jordan Stein 101390302

 * CSCI 3412 - Algorithms - FALL 2015

 * Program 2 - Searching Algorithms

*/


import java.io.File;

import java.io.FileNotFoundException;

import java.util.Arrays;

import java.util.Collections;

import java.util.Hashtable;

import java.util.Scanner;

import java.util.Vector;


public class SearchingAlgorithms {


        static long assignments = 0, comparisons = 0;


        public static void main(String args[])

        {

                String fileName = "wordlist.txt";
```

```
		Vector<String> readWords = new Vector<String>(); // vector that will store all words from file
		readWords = readFileToVector(fileName);  // reads all words into words vector (without punctuation lines)

		Scanner sc = new Scanner(System.in);
		System.out.println("Which algorithm would you like to use?");
		System.out.println("1. unsorted vector\n2. sorted vector\n3. binary search tree\n4. hash table");
		int choice = sc.nextInt();
		sc.close();

		int uniqueWords = 0;

		switch(choice)
		{
			case(1): uniqueWords = unsortedUniqueCount(readWords);
					break;
			case(2): uniqueWords = sortedUniqueCount(readWords);
					break;
			case(3): uniqueWords = binarySearchTree(readWords);
					break;
			case(4): uniqueWords = hashTable(readWords);
					break;
		}

		System.out.println("Number of unique words = " + uniqueWords);
		System.out.println("Number of assignments = " + assignments);
		System.out.println("Number of comparisons = " + comparisons);

	}

	public static int countStr(String str, Vector<String> readWords)
	{
		int count = 0;
		for (int i=0; i < readWords.size(); i++)
```

```
        {
                if (readWords.get(i).equals(str))
                        count ++; // counts how many times the word appears
                comparisons++;
        }
        return count;
}


public static int hashTable(Vector<String> readWords)
{
        String[] wordArr = new String[readWords.size()];
        for (int i=0; i < readWords.size(); i++)
                wordArr[i] = readWords.get(i);

        HashFunction wordHashTable = new HashFunction(wordArr);

        assignments = wordHashTable.getAssignments();
        comparisons = wordHashTable.getComparisons();
        return wordHashTable.countUniqueWords();
}


public static int binarySearchTree(Vector<String> readWords)
{
        BinarySearchTree tree = new BinarySearchTree();

        for (int i=0; i < readWords.size(); i++)
        {
                if (!tree.find(readWords.get(i))) // if we didn't find the word in the tree, insert it
                        tree.insert(readWords.get(i));
        }
        assignments = tree.getAssignments();

        comparisons = tree.getComparisons();


        return tree.size;

}




public static int unsortedUniqueCount(Vector<String> readWords)

{

        System.out.println("Running unsortedUniqueCount. Please wait.");

        Vector<String> uniqueWords = new Vector<String>(); // stores all unique words

        Vector<Integer> uniqueCount = new Vector<Integer>();


        for (int i=0; i < readWords.size(); i++)

        {

                if (!uniqueWords.contains(readWords.get(i)))

                {

                        uniqueWords.add(readWords.get(i));

                        uniqueCount.add(countStr(readWords.get(i), readWords));

                        assignments++;

                }

                comparisons += uniqueWords.size(); //.contains compares all words against eachother



                if (i % 10000 == 1 ) // only used to tell me how far along this algorithm is running during execution

                {

                        double d = (double) i;

                        double r = (double) readWords.size();

                        System.out.printf("%.2f",(d/r)*100);

                        System.out.println("%");

                }


        }
        return uniqueWords.size();
}

public static int sortedUniqueCount(Vector<String> readWords)
{
        Vector<String> uniqueWords = new Vector<String>();
        Vector<Integer> uniqueCount = new Vector<Integer>();

        Collections.sort(readWords);

        int count = 0; // index for counting same words.
        for (int i=0; i < readWords.size()-1; i++)
        {
                if (!readWords.get(i).equals(readWords.get(i+1)))
                {
                        count++; // increment count
                        uniqueWords.add(readWords.get(i)); // add that unique word to the vector
                        uniqueCount.add(count); //add the count of that word to the count vector
                        count = 0; // reset count to zero for next unique word
                        assignments++;
                }
                else count++; // add to count if the word is the same.
                comparisons++;
        }

        if (!uniqueWords.contains(readWords.get(readWords.size()-1))) //adds the last element if it is unique (*it is*)
        {
                uniqueWords.add(readWords.get(readWords.size()-1));
                uniqueCount.add(Collections.frequency(readWords,  readWords.get(readWords.size()-1)));
                assignments++;
        }

        return uniqueWords.size();
```

```
	}


	public static Vector<String> readFileToVector(String fileName)
	{
		Vector<String> words = new Vector<String>();
		File file = new File(fileName);


		try{
			Scanner sc = new Scanner(file);
			while (sc.hasNextLine())
			{
				String data = sc.nextLine().toLowerCase();
				char ch=' ';
				if (!data.equals(""))
				ch = data.charAt(0); // get first character of string
				if (ch >= 'a' && ch <= 'z')
					words.add(data); // adds values to words vector if character is alphabetical.
			}
			sc.close();
		}
		catch (FileNotFoundException e)
		{
			e.printStackTrace();
		}


		return words;
	}
}
```

BinarySearchTree.java

```
public class BinarySearchTree {
        public static  Node root;
        public static int size = 0; // size of tree. (increments if insert function is called)
        int comparisons = 0;
        int assignments = 0;

        int getComparisons()
        {
                return comparisons;
        }
        int getAssignments()
        {
                return assignments;
        }

        public BinarySearchTree(){
                this.root = null;
        }

        public boolean find(String w){
                Node current = root;
                comparisons++;
                while(current!=null){
                        if(current.word.compareTo(w) == 0){
                                current.occurrence++; // if we found the word, increment the
occurrence
                                return true;
                        }else if(current.word.compareTo(w) > 0){
                                current = current.left;
                        }else{
                                current = current.right;
                        }
                }
                return false;
        }

        public Node get(String w){ // return's the node if word is found, otherwise returns null
                Node current = root;
                while(current!=null){
                        if(current.word.compareTo(w) == 0){
                                return current;
                        }else if(current.word.compareTo(w) > 0){
                                current = current.left;
                        }else{
                                current = current.right;
                        }
                        comparisons++;
                }
                return null;
        }



        public boolean delete(String w){
                Node parent = root;
                Node current = root;

                boolean isLeftChild = false;
                while(current.word.compareTo(w) != 0){
                        parent = current;
                        if(current.word.compareTo(w)>0){
                                isLeftChild = true;
                                current = current.left;
                        }else{
                                isLeftChild = false;
                                current = current.right;
                        }
                        if(current == null){
                                return false;
                        }
                }
                //if i am here that means we have found the node
                //Case 1: if node to be deleted has no children
                if(current.left==null && current.right==null){
                        if(current==root){
                                root = null;
                        }
                        if(isLeftChild ==true){
                                parent.left = null;
                        }else{
                                parent.right = null;
                        }
                }
                //Case 2 : if node to be deleted has only one child
                else if(current.right==null){
                        if(current==root){
                                root = current.left;
                        }else if(isLeftChild){
                                parent.left = current.left;
                        }else{
                                parent.right = current.left;
                        }
                }
                else if(current.left==null){
                        if(current==root){
                                root = current.right;
                        }else if(isLeftChild){
                                parent.left = current.right;
                        }else{
                                parent.right = current.right;
                        }
                }else if(current.left!=null && current.right!=null){

                        //now we have found the minimum element in the right sub tree
                        Node successor = getSuccessor(current);
                        if(current==root){
                                root = successor;
                        }else if(isLeftChild){
                                parent.left = successor;
                        }else{
                                parent.right = successor;
                        }
                        successor.left = current.left;

                }
                comparisons++;
                return true;
        }

        public Node getSuccessor(Node deleleNode){
                Node successsor =null;
                Node successsorParent =null;
                Node current = deleleNode.right;
                while(current!=null){
                        successsorParent = successsor;
                        successsor = current;
                        current = current.left;
                }
                //check if successor has the right child, it cannot have left child for sure
                // if it does have the right child, add it to the left of successorParent.
//              successsorParent
                if(successsor!=deleleNode.right){
                        successsorParent.left = successsor.right;
                        successsor.right = deleleNode.right;
                }
                comparisons++;
                return successsor;
        }
        public void insert(String w){
                this.size++; // increment size of tree
                assignments++;
                comparisons++;
                Node newNode = new Node(w);
                if(root==null){
                        root = newNode;
                        return;
                }

                Node current = root;
                Node parent = null;
                while(true){
                        parent = current;
                        if(0<current.word.compareTo(w)){
                                current = current.left;
                                if(current==null){
                                        parent.left = newNode;
                                        return;
                                }
                        }else{
                                current = current.right;
                                if(current==null){
                                        parent.right = newNode;
                                        return;
                                }
                        }
                        comparisons++;
                }
        }
        public void display(Node root){
                if(root!=null){
                        display(root.left);
                        System.out.print(" " + root.word);
                        display(root.right);
                }
        }
        public static void main(String arg[]){

        }
}

class Node{

        String word;
        int occurrence;

        Node left;
        Node right;
        public Node(String w){
                this.occurrence = 0;
                this.word = w;
                left = null;
                right = null;
        }
}
```

HashFunction.java

```
public class HashFunction {

        WordList[] theArray;
        int arraySize;
        int itemsInArray = 0;
        int amountInTable = 0;

        int assignments = 0;
        int comparisons = 0;


        public int countUniqueWords()
        {
                return amountInTable;
        }
        public int getAssignments()
        {
                return assignments;
        }
        public int getComparisons()
        {
                return comparisons;
        }

        public int stringHashFunction(String wordToHash)
        {
                int hashKeyValue = 0;

                for (int i = 0; i < wordToHash.length(); i++) {
                        // 'a' has the character code of 97 so subtract
                        // to make our letters start at 1
                        int charCode = wordToHash.charAt(i) - 96;
                        int hKVTemp = hashKeyValue;

                        // Calculate the hash key using the 26 letters
                        // plus blank
                        hashKeyValue = (hashKeyValue * 27 + charCode) % arraySize;
                }
                return hashKeyValue;
        }

        HashFunction(String[] elementsToAdd)
        {
                arraySize = elementsToAdd.length*2;
                theArray = new WordList[arraySize];

                for (int i = 0; i < arraySize; i++)
                {
                        theArray[i] = new WordList();
                }
                addTheArray(elementsToAdd);
        }

        public void insert(Word newWord)
        {
                String wordToHash = newWord.theWord;

                int hashKey = stringHashFunction(wordToHash);
                // Add the new word to the array and set
                // the key for the word
                try{
                theArray[hashKey].insert(newWord, hashKey);
                }
                catch (ArrayIndexOutOfBoundsException e){}
                assignments++;
        }
```

```
	public Word find(String wordToFind)
	{
		Word theWord = 
theArray[stringHashFunction(wordToFind)].find(stringHashFunction(wordToFind), wordToFind);
		return theWord;
	}

	public void addTheArray(String[] elementsToAdd) {
		for (int i = 0; i < elementsToAdd.length; i++) {

			String word = elementsToAdd[i];

			try{
			if (theArray[stringHashFunction(word)].find(stringHashFunction(word), 
word) == null)
			{
				Word newWord = new Word(word, 1);
				// Add the Word to theArray
				insert(newWord);
				amountInTable++;

			} else theArray[stringHashFunction(word)].find(stringHashFunction(word), 
word).increment(); // increment occurrence if word is in array
			}catch (ArrayIndexOutOfBoundsException e){}
			comparisons++;
		}

	}

	public void displayTheArray()
	{
		for (int i = 0; i < arraySize; i++)
		{
			System.out.println("theArray Index " + i);
			theArray[i].displayWordList();
		}
	}

}

class Word {

	public String theWord;
	public int occurrence;

	public int key;

	public Word next;

	public Word(String theWord, int occurrence)
	{
		this.theWord = theWord;
		this.occurrence = 1;
	}

	public String toString()
	{
		return theWord + " : " + occurrence;
	}

	public void increment()
	{
		this.occurrence++;
	}

}

class WordList {

	public Word firstWord = null;
```

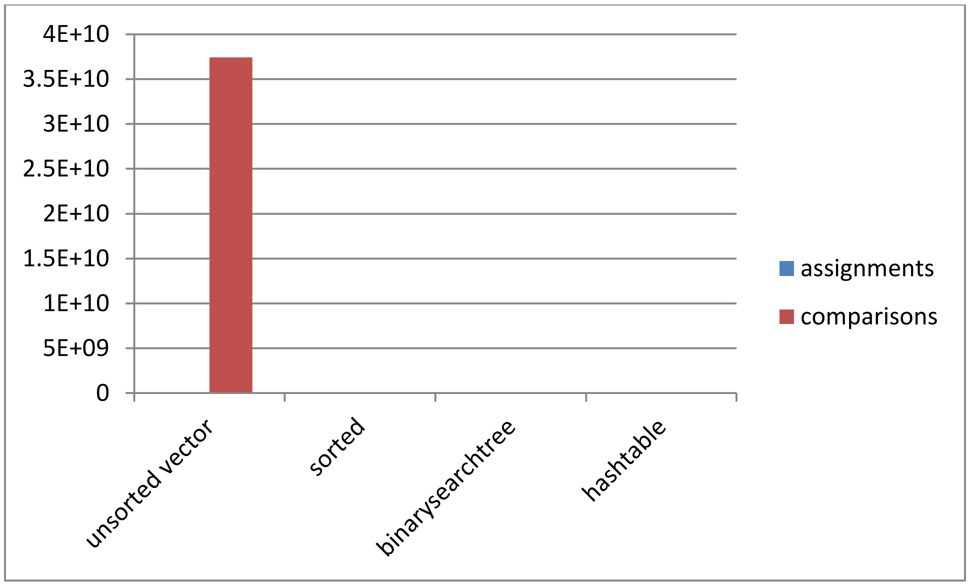
```
        public void insert(Word newWord, int hashKey)
        {
                Word previous = null;
                Word current = firstWord;
                newWord.key = hashKey;

                while (current != null && newWord.key > current.key)
                {
                        previous = current;
                        current = current.next;
                }

                if (previous == null)
                        firstWord = newWord;
                else previous.next = newWord;

                newWord.next = current;
        }

        public void displayWordList()
        {
                Word current = firstWord;

                while (current != null)
                {
                        System.out.println(current);
                        current = current.next;
                }
        }

        public Word find(int hashKey, String wordToFind)
        {
                Word current = firstWord;

                // Search for key, but stop searching if
                // the hashKey < what we are searching for
                // Because the list is sorted this allows
                // us to avoid searching the whole list

                while (current != null && current.key <= hashKey)
                {
                        if (current.theWord.equals(wordToFind))
                                return current;

                        current = current.next;
                }
                return null;
        }
}
```

Part 3 – Analysis

After implementing the data structures and algorithms to obtain the concordance of Shakespear's work, I am able to provide a concrete analysis for each implementation (unsorted, sorted, binary search tree, and hash table) and I can compare the results against each other. The following pages include charts and explanations of the growth of each algorithm.

Unsorted Vector

```
Number of unique words = 27886
Number of assignments = 27886
Number of comparisons = 37413174228
```



The assignments vs comparisons output for the unsorted vector did not surprise me. An assignment was made only when the word was unique, which correlates to it being equal to the amount of unique words. The comparisons count is very large because we had to compare every element in the original vector against each other to calculate the concordance of the words. For each iteration, I also had to check if we had placed the current word in the uniqueWords vector to avoid placing the word in twice. I have included this on its own bar chat because it's data is very large compared to the other three algorithms.The runtime of this algorithm was O(N^2) as expected.

Sorted Vector

```
Number of unique words = 27886
Number of assignments = 27886
Number of comparisons = 807860
```

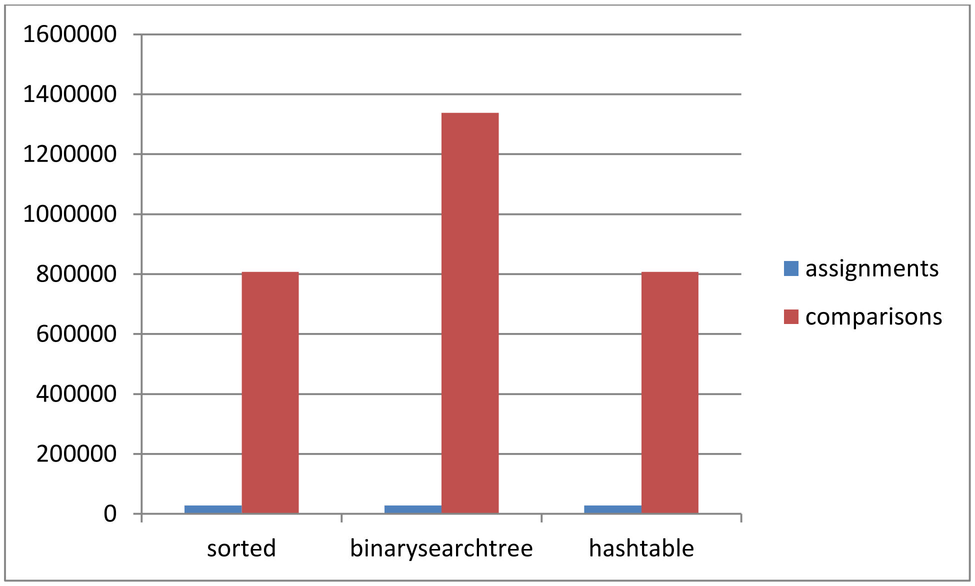
	sorted	binarysearchtree	hashtable
assignments	27886	27886	27886
comparisons	807860	1338243	807861

Binary search tree:

```
Number of unique words = 27886
Number of assignments = 27886
Number of comparisons = 1338243
```

Hash table:

```
Number of unique words = 27886
Number of assignments = 27886
Number of comparisons = 807861
```



The sorted vector algorithm output the comparison and assignment count was as expected. Based off the implementation, I was comparing every element index in the sorted vector against the next index to determine if it was unique. Thus, the amount of comparisons done was equal to the size of the vector. An assignment was made every time a unique word was found, which correlates to the amount of assignments being equal to the amount of unique words. This function successfully output in O(N) time as expected.

The binary search tree surprised me. Although the amount of comparisons made during the formation of the tree was higher, the tree algorithm ran just as fast compared to the sorted vector and hash table algorithms. The amount of unique words found was on par with the amount expected. The comparison is larger because inserting a node into the tree requires us to iterate through half of the nodes that are currently in the tree, requiring many comparisons while truncating through the tree. The runtime was on par with O(log n) as expected because we truncated half of the tree for each recursive call while building it. I only counted assignments that were made when placing words into the tree, thus the assignment count is on par with the unique words found. The binary search tree was my favorite algorithm used in this program.

The hash table was the most confusing to implement but fortunately I was able to create the correct concordance. I did not assume that the comparison count would be equal to the sorted vector's comparison count however it does make sense because I only make one comparison whenever I am inserting a value into the table by checking if it already exists at the hashed index. Again, I only counted assignments that assigned words into the table itself, which makes it be the same as the unique word count.

Conclusion:

I specified, implemented, and analyzed the runtime of creating a concordance of Shakespear's works using an unsorted vector, sorted vector, binary search tree, and a hash table. The outputs of my program had successfully matched the expected order of growth based off the inputs given. I learned a lot about the implementation of each algorithm used and the importance of choosing the right data structure for a given problem.