Programming Assignment 2
Searching Algorithms

CSCI 3412 – Algorithms
Fall 2018
Lucas Fulmer

Introduction

For this assignment, I have implemented three different search algorithms in order to search through three data structures. A single word is read-in and then searched for. If the word is found to already be in the data structure, it is not added to the data structure (there are no duplicates in the data structures). The three data structures are an unsorted array, sorted array, and hash table. In order to search the unsorted array, I have implemented a linear search. The sorted array utilizes a binary search. The hash table hashes using a key in order to search. This document analyzes the performance of the three algorithms, both in their expected and actual asymptotic growth. NOTE: The search algorithms are conducted for every single entry (n).

From these analyses, we show that the expected order of growth is as follows:

- Linear Search – O(n)
- Binary Search – O(log n)
- Hashing – O(1)

Problem Definition

The searching problem is defined as:

Input: A sequence of n entries $\langle a_1, a_2, ..., a_n \rangle$ and a single search entry.

Output: Boolean True or False based on whether the term being searched for is found within the sequence.

Discussion

The purpose of the assignment is to create three versions of a concordance for the complete works of William Shakespeare. Generally, a concordance will list unique words in alphabetical order and the number of times that the word occurs. As stated above, I have arranged the data in three different ways: unsorted, sorted, and a hash table. I read in words one at a time and checked to see if the word was already contained in the data structure. If the word was not found, then it was added to the data structure. If the word was found to already be in the data structure, then I increment the number of entries. For the case of the sorted array, I re-sorted the array each time a word was added.

Part 1 – Pseudo-Code

Unsorted Array – Linear Search

The linear search simply checks each index of the array. If an index matches the search term, it is returned, otherwise the function returns -1.

LINEARSEARCH(A, item)

```
for  i = 1 to A.length
        if A[i] = item
                Return i
Return -1
```

Expected search time is O(m) where m is the number of unique words, however, because we conduct the search n times (n = total number of words), the actual growth is O(m*n).

Sorted Array – Binary Search

The binary search works with sorted arrays by constantly checking the middle index for a value. If the value is less than the middle point, then we check left. If the value is greater than the middle, then we check right. We constantly halve the size of the array until the value is found. While the binary search can be recursive, I chose an iterative implementation.

BINARYSEARCH(A, start, end, value)

```
while start ≤ end
    middle = (start+end)/2
    if A[middle] = value
            return middle
    else if A[middle] < value
            start = middle + 1
    else
            end = middle - 1
```

Expected search time is O(log m) where m is the number of unique words. We conduct this search n times giving an actual expected growth of O(n log m)

Hash Table - Hashing

The hash table uses a hashing function to find an integer representation of a given word. We then take the modulo of that value to find the index or “key” for the word. In the event of collisions, each index contains a linked-list for chaining. When searching, we use the key to search if a word is already in our list. I used Python’s built in hash() function in order to create key values for my algorithm

HASHSEARCH(A, word)

```
key = hash(word)
found = false
while A[key] not = Null and not found
        go to next node
```

```
        if A[key] = word
            found = True
            increment A[key]
            return found
    return False
```

Expected search time of O(1) for a hash.

3. Implementation – Source Code

All algorithms were developed using Python programming language and are contained in the file fulmerPA2.py.

```
#PA2 - Algorithms

#Lucas Fulmer


import time


#creating a class for the words we read in

class WordEntry:


    word = ' '

    numEntries = 0


    def __init__(self):

        self.word = ''

        self.numEntries = 1


    def __init__(self, addWord, num):

        self.word = addWord

        self.numEntries = num


    def incrEntry(self):

        self.numEntries += 1


    def __lt__(self, other):
```

```
        return self.word < other.word


    def __eq__(self, other):
        return self.word == other.word


#creating a hash table data structure
#taken from
http://interactivepython.org/runestone/static/pythonds/SortSearch/Hashing.htm
l
class HashTable:


    def __init__(self):
        self.size = 27886
        self.slots = [None] * self.size
        self.data = [None] * self.size


    def put(self,key,data):
        hashvalue = self.hashfunction(key,len(self.slots))


        if self.slots[hashvalue] == None:
            self.slots[hashvalue] = key
            self.data[hashvalue] = data
        else:
            if self.slots[hashvalue] == key:
                self.data[hashvalue] = data  #replace
            else:
                nextslot = self.rehash(hashvalue,len(self.slots))
                while self.slots[nextslot] != None and self.slots[nextslot]
!= key:
                    nextslot = self.rehash(nextslot,len(self.slots))


            if self.slots[nextslot] == None:
                self.slots[nextslot]=key
```

```
            self.data[nextslot]=data
        else:
            self.data[nextslot] = data #replace

def hashfunction(self,key,size):
    return key % size

def rehash(self,oldhash,size):
    return (oldhash+1)%size

def get(self,key):
    startslot = self.hashfunction(key,len(self.slots))

    data = None
    stop = False
    found = False
    position = startslot
    while self.slots[position] != None and not found and not stop:
        if self.slots[position] == key:
            found = True
            data = self.data[position]
        else:
            position=self.rehash(position,len(self.slots))
            if position == startslot:
                stop = True
    return data

def __getitem__(self,key):
    return self.get(key)

def __setitem__(self,key,data):
    self.put(key,data)
```

```
#iterative binary search for our sortedArray function
def binSearch(arr, start, end, value):
    global bincompare

    if end == 0:
        if arr[0].word == value:
            return 0
        else:
            return -1
    while start <= end:
        mid = int((start + end) / 2)
        bincompare += 1
        if arr[mid].word == value:
                return mid

        elif arr[mid].word < value:
            start = mid + 1

        else:
            end = mid - 1

    return -1

#Function to read in shakespear.txt into unsorted array
def unsortedArray(textfile):
    infile = open(textfile, 'r')
    entryArray = []
    compare = 0
    assign = 1
```

```
    first = infile.readline()
    first = first.replace('\n', '')
    entryArray.append(WordEntry(first.lower(), 1))
    for line in infile:
        tempWord = line
        if tempWord[0].isalpha():
            tempWord = tempWord.replace('\n', '')
            entry = WordEntry(tempWord.lower(), 1)
            for x in range(len(entryArray)):
                compare += 1
                if entry.word == entryArray[x].word:
                    entryArray[x].numEntries += 1
                    break
                elif x == len(entryArray)-1:
                    assign += 1
                    entryArray.append(entry)
    infile.close()
    print('There are {} comparisons and {} assignments in the unsorted 
array'.format(compare, assign))
    return entryArray


#reads in the text file, uses binary search and sorts each entry
def sortedArray(textfile):
    global bincompare
    bincompare = 0
    assign = 1
    infile = open(textfile, 'r')
    entryArray = []
    first = infile.readline()
    first = first.replace('\n', '')
    entryArray.append(WordEntry(first.lower(), 1))#adding the first word
```

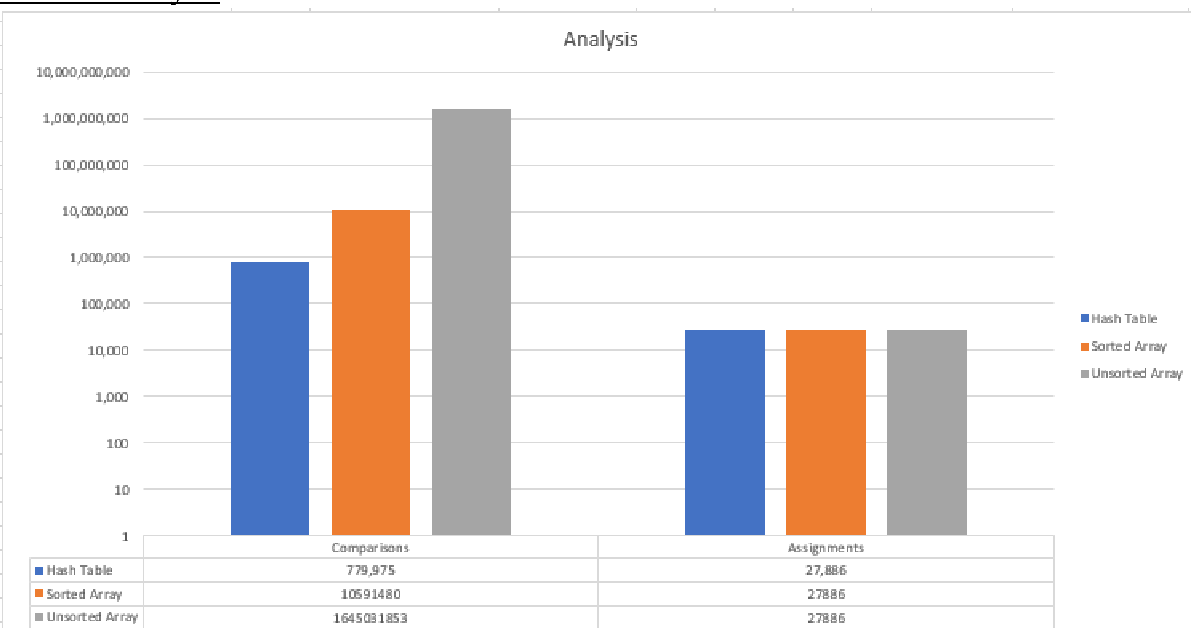
```
    for line in infile:#going through the file line-by-line/word-by-word

        tempWord = line

        if tempWord[0].isalpha():#making sure that first character is
alphabetic

            tempWord = tempWord.replace('\n', '')
            entry = WordEntry(tempWord.lower(), 1)
            index = binSearch(entryArray, 0, len(entryArray)-1, entry.word)

            if index != -1:
                entryArray[index].numEntries += 1

            else:
                entryArray.append(entry)
                entryArray.sort()
                assign += 1

    infile.close()
    print('There were {} comparisons and {} assignments in the sorted
search'.format(bincompare, assign))
    return entryArray

#using our hashing function
def readIntoHash(textfile):
    infile = open(textfile, 'r')
    hashWords = HashTable()
    first = infile.readline()
    first = first.replace('\n', '')
    hashWords.put(hash(first.lower()), WordEntry(first.lower(), 1))
    compare = 0
    for line in infile:
```

```
        tempWord = line
        if tempWord[0].isalpha():
            tempWord = tempWord.replace('\n','')
            entry = WordEntry(tempWord.lower(), 1)
            if hashWords.get(hash(entry.word)):
                compare += 1
                hashWords[hash(entry.word)].numEntries += 1
            else:
                hashWords.put(hash(entry.word), entry)

    infile.close()
    return compare

def builtInHash(textfile):
    infile = open(textfile, 'r')
    hashWords = dict()
    first = infile.readline()
    first = first.replace('\n', '')
    hashWords[first.lower()] = WordEntry(first.lower(), 1)
    for line in infile:
        tempWord = line
        if tempWord[0].isalpha():
            tempWord = tempWord.replace('\n','')
            entry = WordEntry(tempWord.lower(), 1)
            if entry.word in hashWords:
                hashWords[entry.word].numEntries += 1
            else:
                hashWords[entry.word] = entry

    infile.close()
    return hashWords
```

```
##start = time.perf_counter()
unsorted = unsortedArray('wordlist.txt')
unsorted.sort()
print("Here are the first 10 and last 10 entries of the unsorted array")
for x in range(10):
    print("{} : {}".format(unsorted[x].word, unsorted[x].numEntries))

print('------------------------------')

for x in range(len(unsorted)-10, len(unsorted)):
    print("{} : {}".format(unsorted[x].word, unsorted[x].numEntries))

sortArray = sortedArray('wordlist.txt')
myhash = readIntoHash('wordlist.txt')
hashing = builtInHash('wordlist.txt')
##elapsed = (time.perf_counter() - start)
numWords = len(hashing)

print("Now printing the results of the sorted array.")
for x in range(10):
    print("{} : {}".format(sortArray[x].word, sortArray[x].numEntries))

print('------------------------------')

for x in range(len(sortArray)-10, len(sortArray)):
    print("{} : {}".format(sortArray[x].word, sortArray[x].numEntries))

print('There were {} comparisons in the hashing function.'.format(myhash))

print("There are {} unique words.".format(numWords))
flag = 0
for i, x in enumerate (sorted(hashing)):
```

```
if i < 10:

    print('{} : {}'.format(x, hashing[x].numEntries))


elif i == 11:

    print('--------------------------')


elif i < len(hashing) and i > len(hashing)-11:

    print('{} : {}'.format(x, hashing[x].numEntries))
```

<u>Part 4 – Analysis</u>



Analysis: The above graph shows the number of comparisons and assignments. Notice that assignments are equal across each algorithm, because only the initial assignment is recorded. For the sorted array data structure, the additional assignments conducted during the re-sorting were not measured.

Analysis Table:

Data Structure	Comparisons	Assignments	Time
Unsorted Array	1,645,031,853	27,886	446sec
Sorted Array	10,591,480	27,886	142sec
Hash Table	779,975	27,886	5.4sec

Conclusions:

Because assignments only correspond to unique words, we see that the preponderance of computational time and the cause of asymptotic growth is due to the comparisons. As a result, the search algorithms which prioritize searching speed are far more efficient. In the case of the three algorithms tested, we see that the linear search is by far the slowest (as expected) and the hash function is by far the fastest (as expected). Although the sorted algorithm requires an additional sorting step for every assignment, we can see that it is quickly overpowered by the sheer number of comparisons in the unsorted array. In other words, we can sort 27,886 words much quicker than we can compare approximately 800,000.

Overall conclusion is that if our algorithm prioritizes searching and assigning, hash tables are by far the most efficient choice.