3013 - Algorithms **Name:** _____
Fall 2021
Exam 1
02/09/2020

READ THE INSTRUCTIONS

- Use pencil only
- Write your name at the top of all pages turned in.
- Do not remove the staple from your test.
- Handwriting that is illegible (messy, small, not straight) will lose points.
- Indentation matters. Keep code aligned correctly.
- Answer all questions in the provided space directly on the test.
- **Failure to comply will result in loss of letter grade.**

This exam is 6 pages (without cover page) and 3 questions. Total of points is 143.

Grade Table (don't write on it)

Question	Points	Score
1	58	
2	36	
3	49	
Total:	143	

1. **Hashing Questions**
   Use the following values for the next 3 questions:

8, 2, 7, 18, 15, 19, 23, 15, 20, 16

(a) (10 points) Draw a hash table of size 13 using open addressing with linear probing for collision resolution. $(h(K))\ mod\ m$

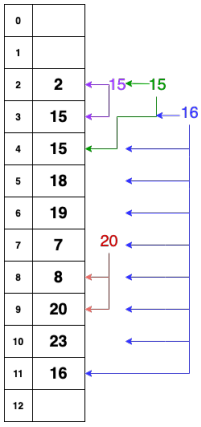

Figure 1: Linear Probing Collisions: 15, 15, 20, 16

(b) (10 points) Draw a hash table of size 13 using open addressing with quadratic probing for collision resolution. $(h(K)\ +\ i^2)\ mod\ m$

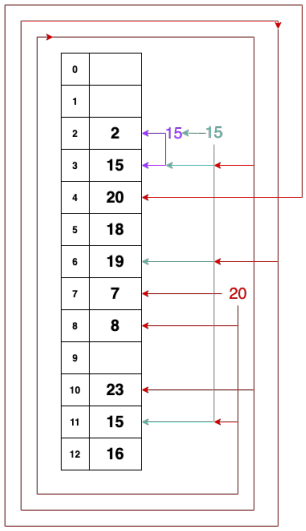

Figure 2: Quadratic Probing Collisions: 15, 15, 20

(c) (10 points) Draw a hash table of size 13 using chaining for collision resolution. $(h(K))\ mod\ m$

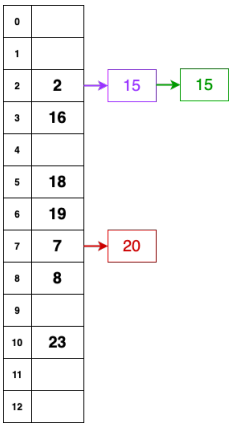

Figure 3: Chaining Collisions: 15, 15, 20

(d) (7 points) What are Trie's good for? Can they be used in place of a binary tree?

Answer:

Tries are good for (to name a few). (Your answer should include more than just a single example):
- Storing words in an organized fashion.
- Finding words that have the same prefix.
- Fast lookups for for whole or partial words.
- Auto suggestions.
- Backend (dictionary lookup) for a spell checker.

Can they replace a BST?
The short answer: **no**. They are two different beasts. A BST can store many different values, where a Trie cannot (or should not). A Trie is good at storing words in an organized "prefix" type fashion so you can find similar words **fast**. Thats it. BST's are good at storing any type of data that has an associated "key", thus allowing items to be found in O(lg n) time (with a balanced tree). Tries ... are not robust in this way.

(e) (7 points) What makes a good hash function?

Answer:

1. The hash value is fully determined by the data being hashed.
2. The hash function uses all the input data.
3. The hash function "uniformly" distributes the data across the entire set of possible hash values.
4. The hash function generates very different hash values for similar strings.

(f) (7 points) Hashing integer values is easy, but what if I want to hash strings? How do I convert a string to an integer?

Answer:

You would need to convert the string to an integer. There are a multiple methods for doing this, but the simplest is to take each individual letter in a string and sum the Ascii values of each letter creating a single integer value. The accepted method is known as a **Polynomial Rolling Hash** which can be read about HERE.

(g) (7 points) What is the load factor, and give a general overview of how certain load factors are chosen.

Answer:

What is the load factor?
Hash table size: n
Items in the hash table: m
Load factor : $\lambda = m/n$

In other words the load factor is a ratio of items in the hash table vs number of total slots available.
Choosing a load factor:
Choosing a load factor is determined by your collision resolution technique AND your data. If I have intimate knowledge of my data and develop a "perfect" hash, then my load factor can be 1 (100%), because I am guaranteed NO collisions. Mostly however, we do not have a perfect hash and we need to deal with collisions.

Open Addressing:
In open addressing we usually shoot for 75% to 80% of the hash table size. Why? Because we need to ensure we have open slots to "kill" a search for an item that does not exist. Remember, a full hash table (no empty slots) degrades to ***O(n)*** search because there is no way to terminate a search if a key does not exist.
Chaining:
In chaining, the worst case lookup degrades to the length of your longest chain. Searching a has table that uses chaining cannot degrade to ***O(n)*** unless the hash function had a stroke and returns the same value for all hash keys. However, following linked addresses is slower than array indexes. OK, then what is a good load factor when using chaining as a collision resolution technique? In this case a load factor can easily be 1 or more! As long as the "longest" chain doesn't get too long (that's lots longs in one sentence) then performance will not degrade too much.

Your answer should define load factor, and distinguish between open addressing and chaining. My answer is more than necessary.

2. Trie Questions

(a) (15 points) Add the following words to a Trie data structure. Draw your Trie to clearly show you understand how they are built.

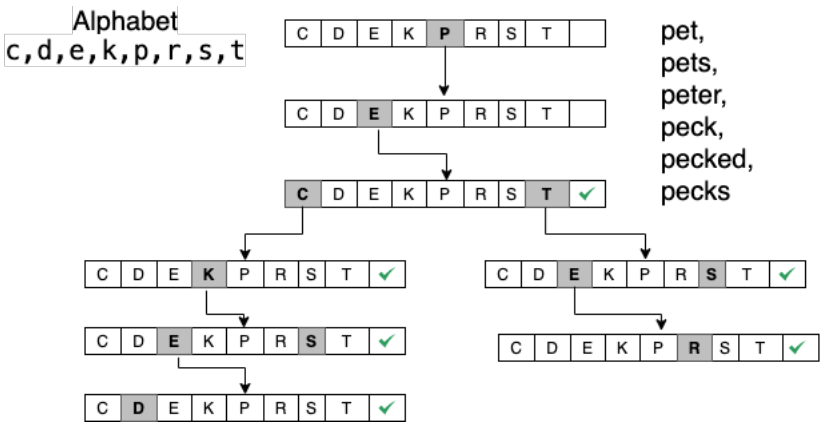

Figure 4: Trie visualization

(b) (7 points) What is the complexity of creating a Trie?

Answer:

The complexity of creating a Trie is ***O(W*L)***
where ***W*** is the number of words,
and ***L*** is average length of the word.

(c) (7 points) What is the complexity of searching for a specific word in a Trie?

Answer:

The complexity of searching a Trie is ***O(L)***
where ***L*** is the length of the word (or prefix).

(d) (7 points) What is the complexity of searching for all words with a given prefix in a Trie?

Answer:

Since we don't know what words are in the Trie (they may all start with "pre"), we must assume that all nodes must be looked at! So, this means that a search for all words beginning with some "prefix" could result in looking at every single node, or ***O(W*L)***!

3. Sorting Questions

(a) (7 points) What are the 2 major categories of sorting algorithms.

Answer:

Crappy / Slow = $O(n^2)$
Good / Fast = $O(n\ lg\ n)$

(b) (7 points) Which sorting algorithms (discussed in class) fit into each of the two categories.

Answer:

Crappy / Slow
- Bubble Sort
- Insertion Sort
- Selection Sort

Good / Fast
- Heap Sort
- Merge Sort
- Quick Sort

(c) (7 points) What is a stable sorting algorithm?

Answer:

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

For example. The goal is to sort them on face value, keep cards with the same face value in the same order they were in the original set. The fives in this example should stay in the same order as they are encountered. See below:

Before	**5♦**	**10♠**	**3♣**	**7♦**	**5♠**	**8♠**	**K♣**	**5♥**	**A♥**	**5♣**
After	**3♣**	**5♦**	**5♠**	**5♥**	**5♣**	**7♦**	**8♠**	**10♠**	**K♣**	**A♥**

Table 1: Before and after stable sort.

(d) (7 points) Which sorting algorithms use divide and conquer techniques? Why or why is this not a good strategy?

Answer:

Which sorting algorithms use divide and conquer techniques?

Both **Merge sort** and **Quicksort** employ a divide and conquer approach.

Why or why is this not a good strategy?

The short answer is that even though the algorithms for for Quicksort and Mergesort are recursive which tend to be resource hogs, having a running time of $O(n\ lg\ n)$ far outweighs the $O(n^2)$ algorithms that do not use divide and conquer. Remember that recursion also lends itself to solving tough problems but allowing them to be broken down into manageable chunks. Both Quicksort and Mergesort take advantage of this.

Advantages of Divide and Conquer
- Allows us to solve difficult problems since it divides the problem into manageable sub problems.
- Tends to give algorithms that are faster. Meaning it can solve a problem in less steps.

Disadvantages of Divide and Conquer
- Recursion is slow.
- It uses more resources.
- It has bigger space requirements

(e) (7 points) How do counting sort and radix sort relate, and what makes them a great pair?

Answer:

How do counting sort and radix sort relate and what makes them a great pair?

Keeping it simple (since I did not ask you to get in depth with these two sorts) **Counting Sort** and **Radix Sort** are an integer sorting algorithms, not comparison based algorithms. Our comparisons based sorting algorithms peak out at $O(n\ lg\ n)$ performance. But given the right circumstances when Radix Sort uses Counting Sort as a subroutine, we can achieve linear time sorting $O(n)$! That is what makes them a great pair.

(f) (7 points) Quicksort and Mergesort have the same complexity. Briefly explain why, despite similar complexity, Quicksort is the more popular sorting algorithm?

Answer:

Mergesort uses extra space and Quicksort requires little space. Quick sort is an in-place sorting algorithm. In-place sorting means no additional storage space is needed to perform sorting. Merge sort requires a temporary array to merge the sorted arrays and hence it is not in-place giving Quick sort the advantage of space.

Even though Quicksort's worst case complexity is $O(n^2)$ where Mergesort's is $O(n\ lg\ n)$, there are steps one can take to avoid Quicksort worst case scenario (choosing good randomized pivots).
In addition quick sort takes advantage of "locality of reference". This means the next required set of instructions are most likely located near other instructions already loaded into cache. And if the arrays being worked on remain in cache, we can bet a huge performance boost. Since merge sort generates many new arrays, this typically is not the case (also depends on how much data is being sorted).

(g) (7 points) Given the following array [43,63,35,75,81,21,10,90,?] and assuming that Quicksort will be used to sort this array in ascending order, select a value for the last element of the array (indicated by "?") such that the partitioning performed by Quicksort is most balanced.

Answer:

This isn't the best way to choose a pivot, but that's not what the question is asking. So a value that splits the list evenly on either side will work. So any value **between** 43 and 63. The issue is we have to calculate that answer which slows down quicksort.