Lecture 11

Hashing: Mission Possible

Last time:

Recursive methods have the following characteristics:

- It calls itself
- When it calls itself, it does so to solve a smaller version of the same problem. (Recursive case)
- There is a base case (or base cases) that does not call itself anymore.

In the last lecture note, we have used recursion to solve a few problems such as Fibonacci numbers and Binary Search.

Recursion is usually used because it simplifies a problem conceptually, not because it is more efficient.

For a recursive method, there is a lot of bookkeeping information that needs to be stored as you can see from the example below.

```
recursiveSum(5)                                          15
   5+recursiveSum(4)                                10+5=15
      4+recursiveSum(3)                        6+4 = 10
        3+recursiveSum(2)                 3+3 = 6
           2+recursiveSum(1) = 1    1+2 = 3
```

We will have a chance to look at recursion in action again.

Today, let's get back to our original discussion, data structures.

Our mission today is ***to improve the speed of searching***.

We know that searching for a key value in an array that is not sorted requires us to examine all the elements of the array: O(n)

In case the array is sorted, we can then use the binary search search and its worst-case running time complexity is O(log n), which is significantly better than O(n).

However, we do not want to stop here, being satisfied with this improvement. We've got a mission to accomplish.

Now, what if, in advance, we know the index at which the value we are looking for in the array is located?

That would be great because that will give us a constant time, O(1), to search the value.

Hmm… how do we apply this for our mission?

We need to hire someone who is good at it.

Where is our Tom Cruise who can make our "mission impossible" to be "mission possible"?

There is a **hash function**, our Tom Cruise.

unordered array	linear search	O(n)
ordered array	binary search	O(log n)
array (index)	hashing	O(1)

Step 1: Conceptual View

Problem at hand: I want to create my own dictionary to store 50,000 words that I use most often into an array to find them quickly later.

And, I would like every word to occupy its own cell in the array whose length is 50,000 so that I can access (search) the words using an index value.

What is the relationship between the words and index numbers?

Converting words to numbers, more specifically integers

A simplest way: Adding the digits.

a is 1, b is 2, c is 3…. z is 26. (For the sake of simplicity, let's assume that there are only lowercases.)

cats
c = 3
a = 1
t = 20
s = 19
3 + 1 + 20 + 19 = 43

Thus, the word, cats, would be stored in the array cell whose index is 43.

Because I am not that smart, I do not like any word longer than 10 letters. So, I will only store words that have maximum of 10 letters.

Now, what is the total range of numbers with this methodology?

The possible first word is "a"

0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 = 1

And possible last word is "zzzzzzzzzz" assuming that it is a word.
26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 = 260
The range of word codes is from 1 to 260, which is way small.

What does this mean?
Each array cell of 260 cells needs to hold about 192 words (50000/260). *This is what we call "Collision." More on this later!*

That's too many words at the same index that makes not feasible to have a fast search for a word.

Our mission has not yet been accomplished!

We need to figure out a way to **spread out the range of possible indices**.

Getting little more complicated: Multiplying by Powers

Is there a way to guarantee to have a word occupy its own unique array cell?

An ordinary multi-digit number, such as 7,546, really means:
7*1,000 + 5*100 + 4*10 + 6*1 or
7*10^3 + 5*10^2 + 4*10^1 + 6*10^0 *(showing multipliers as powers of 10)*

How can we decompose a word in this way?

For English lower case alphabets and blank, we have 27 possible characters. Let's use powers of 27.

cats
3*27^3 + 1*27^2 + 20*27^1 + 19*27^0 = 60,337

This process does generate a unique number for every word but there is another issue.

How about "zzzzzzzzzz"?
26*27^9 + 26*27^8 + 26*27^7 + 26*27^6 + 26*27*5 + 26*27^4 + 26*27^3 + 26*27^2 + 26*27^1 + 26*27^0

How big is it?
27^9 itself is bigger than 7,000,000,000,000.
An array cannot have that many elements at all.

Suppose that it is possible to have this kind of array, then the other problem is that we create an array that is too big and most array cells are empty.

Our mission has not yet been accomplished!

These are two extremes at both ends.

Trying to find something in the middle: Hashing

How can we reduce or SQUEEZE the length of the array into a reasonably sized one and at the same time spread out the range of possible indices as much as we can?

Range of numbers (200)
0 1 2 3 4 5 6 7 8 9 10 11 12 ………………………….194 195 196 197 198 199

0 1 2 3 4 5 6 7 8 9
Available space (10)

Here comes our organizer: The modulus (%) operator
Value in larger range of numbers % available space → small range
0 % 10 = 0
13 % 10 = 3
55 % 10 = 5
157 % 10 = 7
199 % 10 = 9

Do you see **hashing happening** here?
199 is divided, or **hashed**, into 19 and 9 and we throw away 19 and keep 9.

Closer look at collisions
The price for squeezing a large range into a small one is that we no longer have a guarantee that two or more words won't hash into the same array index. But, luckily, the situation is much better than our first approach, adding the digits.

Our hope or goal to have one word per one index turns out to be not possible.

Let’s take a look at an example, shall we?
Adding *melioration* into the following array using a hashing function.
→ Collides with demystify.

parchment
demystify
slander
quixotic

Does the possibility of collisions make hashing impractical?

Luckily, there are workarounds. We will take a look at a few of them.

Workaround 1: Open Addressing (mainly linear probing)
From the example that is shown before (assuming that the indices are as follows):

5418	
5419	parchment
5420	
5421	demystify
5422	slander
5423	
5424	quixotic
5425	

Where do we put *melioration*?
We check for vacant cells sequentially. We go to 5422 and check whether it is empty or not. If it is not empty which is the case here, then go to 5423. If it is empty, put the word, *melioration*, into 5423.

Step size:
In linear probing, the step size is always 1 that means the probe goes to x, x+1, x+2, x+3 and so on.

Clustering:
A sequence of filled cells in a hash table that is long.

4	144
5	
6	
7	87
8	208
9	989
10	329
11	869
12	867
13	27
14	
15	
16	336

step size can be

$x, x+1^2, x+2^2, x+3^2$

Quadratic

As a hash table gets more and more full, the clusters grow larger and larger.

- When the hash table is half full, the performance is still not bad.
- It is proven that, when it is beyond two-thirds full, the performance degrades seriously.

It is critical to ensure that it never becomes more than two-thirds full.

Load factor
It is the ratio of number of data items in a hash table to the length of the array.
In linear probing, search time goes to infinity as the load factor approaches 1.

How to solve this issue?

Rehashing
First, it is necessary to create a new array that is mostly twice bigger than the old array length but it depends on the load factor you provided.

The hash method then calculates the location of a given data item based on the new array length.

Second, we need to go through the old array, cell by cell, and insert them by calling hash function over and over.

It's a time-consuming process.

Now, is there any other way?

In Open Addressing, there are two other major collision resolution mechanisms: **Quadratic Probing and Double Hashing**.

step size
compare w. linear

$$\begin{tikzcd}
 & \text{hash1} \arrow[r] & \text{index value} \\
\text{double hashing (best choice)} \arrow[ur, dash] \arrow[r, dash] & \text{hash2} \arrow[r] \arrow[d] & \text{step size} \\
 & H(key \% (m-2)) &
\end{tikzcd}$$

m: table length

Workaround 2: Separate Chaining (closed addressing)

In open addressing, collisions are resolved by looking for an open cell in the hash table.

Another approach is to put a linked list at each index in the hash table.

Array Linked Lists

5
6
7
8
9
10
11
12
13

add $\longrightarrow$ addFirst
(O(1))

search $\frac{n}{m} = \alpha$ (O(n))

worst case

Load factor in separate chaining

In separate chaining, it is normal to put N or more items in an array of length N.

Finding the initial cell takes O(__1___) whereas searching through a linked list takes O(__1___) when there are *k* number of elements in the list.

Thus, we do not want the linked lists become too full either. Especially, if your hash function is not good.

However, the load factor in separate chaining can rise above 1 without hurting performance too much.

Which one is better? Well, it depends.

When in doubt, you may consider to use separate chaining, especially, if the number of items that will be inserted into a hash table is unknown. In other words, separate chaining would be better when you would expect to have a high load factor.

Also, think about whether there will be many deletions or not.

In the next lecture, we will look at the implementation view of hash table.