



University of
Pittsburgh

Algorithms and Data Structures 1

CS 0445



Fall 2022

Sherif Khattab

ksm73@pitt.edu

Announcements

- Upcoming Deadlines:
 - Homework 9 and Lab 9
 - reopened until this Friday 12/2 @ 11:59 pm
 - Homework 10 (to be posted soon) and Lab 11
 - next Monday 12/5 @ 11:59 pm
 - Assignment 3 and 4: both due on Friday 12/9 @ 11:59 pm
 - very small amount of work!

This Lecture ...

- Hashing!

Muddiest Points

- **Q:** I could not quite follow the last step of the average runtime for QuickSort. Could you explain the math a bit more?
- **Q:** Will we go through the code of Radix method?
- **Q:** Lab 10 was difficult. I understood the logic of how to do it but had trouble implementing it

Wouldn't it be wonderful if...

- Search through a collection could be accomplished in $\Theta(1)$ with relatively small memory needs?
- Lets try this:
 - Assume we have an array of length m (call it HT)
 - Assume we have a function $h(x)$ that maps from our key space to $\{0, 1, 2, \dots, m-1\}$
 - E.g., $\mathbb{Z} \rightarrow \{0, 1, 2, \dots, m-1\}$ for integer keys
 - Let's also assume $h(x)$ is efficient to compute
- This is the basic premise of *hash tables*

How do we search/insert with a hash map?

- Insert:

```
i = h(x)
```

```
HT[i] = x
```

- Search:

```
i = h(x)
```

```
if (HT[i] == x) return true;
```

```
else return false;
```

- This is a very general, simple approach to a hash table implementation
 - Where will it run into problems?

What do we do if $h(x) == h(y)$ where $x \neq y$?

- Called a *collision*



Consider an example

- Company has 500 employees
- Stores records using a hashmap with 1000 entries
- Employee SSNs are hashed to store records in the hashmap
 - Keys are SSNs, so $|\text{keyspace}| = 10^9$
- Specifically what keys are needed can't be known in advance
 - Due to employee turnover
- What if one employee (with SSN x) is fired and replacement has an SSN of y ?
 - Can we design a hash function that guarantees $h(y)$ does not collide with the 499 other employees' hashed SSNs?

Can we ever guarantee collisions will not occur?

- Yes, if our keyspace is smaller than our hashmap
 - If $|\text{kpacespace}| \leq m$, *perfect hashing* can be used
 - i.e., a hash function that maps every key to a distinct integer $< m$
 - Note it can also be used if $n < m$ and the keys to be inserted are known in advance
 - E.g., hashing the keywords of a programming language during compilation
- If $|\text{kpacespace}| > m$, collisions cannot be avoided

Handling collisions

- Can we reduce the number of collisions?
 - Using a good hash function is a start
 - What makes a good hash function?
 1. Utilize the entire key
 2. Exploit differences between keys
 3. Uniform distribution of hash values should be produced

Examples

- Hash list of classmates by phone number
 - Bad?
 - Use first 3 digits
 - Better?
 - Consider it a single int
 - Take that value modulo m
- Hash words
 - Bad?
 - Add up the ASCII values
 - Better?
 - Use Horner's method to do modular hashing again

The madness behind Horner's method

- Base 10
 - 12345
 - $= 1 * 10^4 + 2 * 10^3 + 3 * 10^2 + 4 * 10^1 + 5 * 10^0$
- Base 2
 - 10100
 - $= 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$
- Base 16
 - BEEF3
 - $= 11 * 16^4 + 14 * 16^3 + 14 * 16^2 + 15 * 16^1 + 3 * 16^0$
- ASCII Strings
 - HELLO
 - $= 'H' * 256^4 + 'E' * 256^3 + 'L' * 256^2 + 'L' * 256^1 + 'O' * 256^0$
 - $= 72 * 256^4 + 69 * 256^3 + 76 * 256^2 + 76 * 256^1 + 79 * 256^0$

Modular hashing

- Overall a good simple, general approach to implement a hash map
- Basic formula:
 - $h(x) = c(x) \bmod m$
 - Where $c(x)$ converts x into a (possibly) large integer
- Generally want m to be a prime number
 - Consider $m = 100$
 - Only the least significant digits matter
 - $h(1) = h(401) = h(4372901)$

Back to collisions

- We've done what we can to cut down the number of collisions, but we still need to deal with them
- Collision resolution: two main approaches
 - Open Addressing
 - Closed Addressing

Open Addressing

- I.e., if a pigeon's hole is taken, it has to find another
- If $h(x) == h(y) == i$
 - And x is stored at index i in an example hash table
 - If we want to insert y , we must try alternative indices
 - This means y will not be stored at $HT[h(y)]$
 - We must select alternatives in a consistent and predictable way so that they can be located later

Linear probing

- Insert:
 - If we cannot store a key at index i due to collision
 - Attempt to insert the key at index $i+1$
 - Then $i+2$...
 - And so on ...
 - $\text{mod } m$
 - Until an open space is found
- Search:
 - If another key is stored at index i
 - Check $i+1, i+2, i+3$... until
 - Key is found
 - Empty location is found
 - We circle through the buffer back to i

Linear probing example

- $h(x) = x \bmod 11$
- Insert 14, 17, 25, 37, 34, 16, 26

0	1	2	3	4	5	6	7	8	9	10
	34		14	25	37	17	16	26		

- How would deletes be handled?
 - What happens if key 17 is removed?

Alright! We solved collisions!

- Well, not quite...
- Consider the *load factor* $\alpha = n/m$
- As α increases, what happens to hash table performance?
- Consider an empty table using a good hash function
 - What is the probability that a key x will be inserted into any one of the indices in the hash table?
- Consider a table that has a cluster of c consecutive indices occupied
 - What is the probability that a key x will be inserted into the index directly after the cluster?

Avoiding clustering

- We must make sure that even *after* a collision, all of the indices of the hash table are possible for a key
 - Probability of filled locations need to be distributed throughout the table

Double hashing

- After a collision, instead of attempting to place the key x in $i+1 \bmod m$, look at $i+h_2(x) \bmod m$
 - $h_2()$ is a second, different hash function
 - Should still follow the same general rules as $h()$ to be considered good, but needs to be different from $h()$
 - $h(x) == h(y)$ AND $h_2(x) == h_2(y)$ should be very unlikely
 - Hence, it should be unlikely for two keys to use the same increment