Motivation (Fast Lookup)

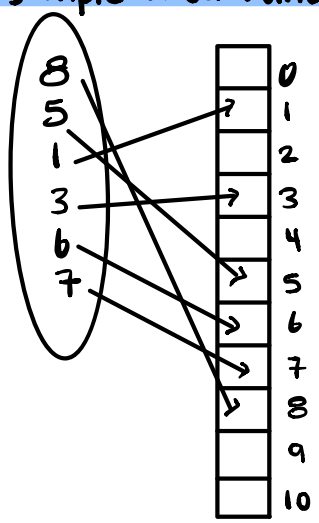
- Linear search O(n)
  - must loop from start to end (n times) until value is found.
- Binary search O(logn)
  - list must be sorted since we split in half, check if value to the left or right, and discard the half that doesn't contain the value.
- Binary search Tree. O(n) when unbalanced, O(logn) when balanced

The above methods are ok, but we can lookup values much faster on average by using hashing

Hash Table

- Hash Table: helps organize data
- Hash Function: applied to keys and gives them addresses h(x)
- O(1) on average
- O(n) worst time
  - when we don't have a good hash function and many collisions.

A simple hash function h(x) = x

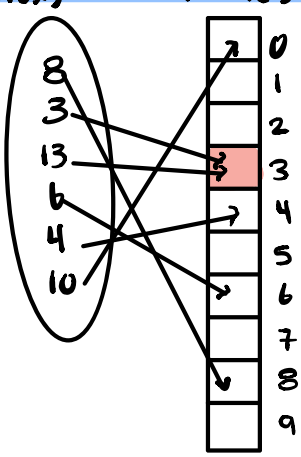


notice how this hash function maps values to their corresponding index.

Although this is simple, we should modify this hash function to efficiently utilize space.

Imagine if we had the values 8, 5, 1, 3, 200. We will need at minimum a table size 200 and a lot of space will go to waste.

h(x) = x % tableSize



h(8) = 8 % 10 = 8
h(3) = 3 % 10 = 3
h(13) = 13 % 10 = 3
h(6) = 6 % 10 = 6
h(4) = 4 % 10 = 4
h(10) = 10 % 10 = 0

This hash function will utilize the space more efficiently than the previous since we are taking the remainders.

However, we've now run into an issue where some values collide. How can we fix this?

Collision Resolution Methods

1. Direct hashing
2. Closed hashing (open addressing)
   - Linear probing method
   - Quadratic probing method
   - Double hashing method
3. Open hashing (Separate Chaining method)

Direct hashing

Insert: 19, 50, 89, 39 $h(x) = x \% \text{ table size}$.

↳ use 5, one more than the given values

0	50
1	
2	
3	
4	~~19~~ ~~89~~ 39

↓ index

$h(19) = 19 \% 5 = 4$
$h(50) = 50 \% 5 = 0$
$h(89) = 89 \% 5 = 4$
$h(39) = 39 \% 5 = 4$

How does this method resolve collisions?
It simply overwrites the previous value.

issues: we lose data. not a good choice for resolving collisions if every bit of data is important.

Advantages: Simple to implement and doesn't require much calculations

Linear Probing $h'(x) = (h(x) + i) \% \text{ tablesize}$, where $i = 0, 1, 2 \ldots$

↳ $h(x) = x \% \text{ tablesize}$

↳ use 5

Insert: 19, 50, 89, 39

0	50
1	89
2	39
3	
4	19

↓ index

Cluster

$h(19) = 19 \% 5 = 4$
$h(50) = 50 \% 5 = 0$
$h(89) = 89 \% 5 = 4$
$h(39) = 39 \% 5 = 4$

$h'(19) = (4 + 0) \% \text{ tablesize} = 4 \% 5 = 4$
$h'(50) = (0 + 0) \% \text{ tablesize} = 0 \% 5 = 0$
$h'(89) = (4 + 0) \% \text{ tablesize} = 4 \% 5 = 4$ * occupied, i++
$h'(89) = (4 + 1) \% \text{ tablesize} = 5 \% 5 = 0$ * occupied, i++
$h'(89) = (4 + 2) \% \text{ tablesize} = 6 \% 5 = 1$
$h'(39) = (4 + 0) \% \text{ tablesize} = 4 \% 5 = 4$ * occupied, i++
$h'(39) = (4 + 1) \% \text{ tablesize} = 5 \% 5 = 0$ * occupied, i++
$h'(39) = (4 + 2) \% \text{ tablesize} = 6 \% 5 = 1$ * occupied, i++
$h'(39) = (4 + 3) \% \text{ tablesize} = 7 \% 5 = 2$

How does this method resolve collisions?
increment i whenever a collision occurs and try again. If another collision occurrs, repeat.

Shortcut: just calculate h(x) and if a collision occurs, continue down until the next available slot. Assume circular space.

issues: values cluster near each other, which will make our search time O(n) in the worst case.

keep in mind that when searching for a value, x, we use the same formula and check whether x is at that location. If not there, we continue to check down until we find it or hit an empty space. If we hit an empty space then it's safe to assume that the value does not exist because if it did then it would have occupied that empty slot when we were mapping.

To better understand why clustering increases search time to O(n) in the worst case, take a look at the following example:

0	a	
1	b	
2	c	
3	d	
4	e	
5	f	→ cluster
6	g	
7	h	
8	i	
9	j	
10	k	
11	L	
12	m	
13		
14	n	
↑		
Index		

Let's say we want to find the value z, and $h'(z) = 0$. notice how z does not exist, but the program will have to search from 0 down until it hits an empty slot. Since this method tends to cluster data, it could take a while until we find that empty slot. Approximately O(n)

Advantages: Fixed our data loss issue

Quadratic probing $h'(x) = (h(x) + i^2) \% \text{ tablesize}$, where $i = 0, 1, 2 \ldots$

↳ $h(x) = x \% \text{ tablesize}$

Insert: 19, 50, 89, 39 ↳ use 5

0	50
1	
2	
3	89
4	19
↓	
Index	

$h(19) = 19 \% 5 = 4$
$h(50) = 50 \% 5 = 0$
$h(89) = 89 \% 5 = 4$
$h(39) = 39 \% 5 = 4$

$h'(19) = (4 + 0^2) \% 5 = 4$
$h'(50) = (0 + 0^2) \% 5 = 0$
$h'(89) = (4 + 0^2) \% 5 = 4$ * occupied
$h'(89) = (4 + 1^2) \% 5 = 0$ * occupied
$h'(89) = (4 + 2^2) \% 5 = 3$
$h'(39) = (4 + 0^2) \% 5 = 4$ * occupied
$h'(39) = (4 + 1^2) \% 5 = 0$ * occupied
$h'(39) = (4 + 2^2) \% 5 = 3$ * occupied
$h'(39) = (4 + 3^2) \% 5 = 3$ * occupied

⋮

* occupied

How does this method resolve collisions?
Like linear but i^2 now.

Issues: we can run into an infinite loop. So you'll have to be careful what type of hash function is selected. There are methods to selecting hash functions that will not get ∞ loops, but these proofs are beyond the scope of this course.

Advantages: We fix the clustering from linear probing by using i^2, but we get something known as secondary clustering. After a while, certain sequences will arise.

Double hashing $h'(x) = (h_1(x) + i\,h_2(x)) \,\%\, \text{tablesize}$, where $i = 0, 1, 2 \ldots$

↳ use $h_1(x) = x \,\%\, \text{table size}$, $h_2(x) = 3 - (x \,\%\, 3)$

Insert: 19, 50, 89, 39 ↳ uses

index	
0	50
1	89
2	39
3	
4	19

$h_1(x)$

$h_1(19) = 19 \,\%\, 5 = 4$

$h_1(50) = 50 \,\%\, 5 = 0$

$h_1(89) = 89 \,\%\, 5 = 4$

$h_1(39) = 39 \,\%\, 5 = 4$

$h_2(x)$

$h_2(19) = 3 - (19 \,\%\, 3) = 2$

$h_2(50) = 3 - (50 \,\%\, 3) = 1$

$h_2(89) = 3 - (89 \,\%\, 3) = 1$

$h_2(39) = 3 - (39 \,\%\, 3) = 3$

$h'(19) = (4 + 0(2)) \,\%\, 5 = 4$ (i = 0)

$h'(50) = (0 + 0(1)) \,\%\, 5 = 0$ (i = 0)

$h'(89) = (4 + 0(1)) \,\%\, 5 = 4$ * occupied (i = 0)

$= (4 + 1(1)) \,\%\, 5 = 0$ * occupied (i = 1)

$= (4 + 2(1)) \,\%\, 5 = 1$ (i = 2)

$h'(39) = (4 + 0(3)) \,\%\, 5 = 4$ * occupied (i = 0)

$= (4 + 1(3)) \,\%\, 5 = 2$ (i = 1)

How does this method resolve collisions?

we now calculate 2 functions and increment "i" if collisions occur. If h_1 is not given, assume $h_1(x) = x \,\%\, \text{tablesize}$.

Advantages: Allows for smaller hash table since it effectively finds a free slot.

Issues: Computationally expensive. Still possible to run into ∞ loop issue. However, can avoid ∞ loop by:

use p, q prime numbers such that $2 < q < p$

$h(x) = x \bmod p$

$g(x) = q - (x \bmod p)$

Open hashing (separate chaining)

Insert: 19, 50, 89, 39

0	50
1	
2	
3	
4	19 → 89 → 39

↓
index

$h(19) = 19 \% 5 = 4$
$h(50) = 50 \% 5 = 0$
$h(89) = 89 \% 5 = 4$
$h(39) = 39 \% 5 = 4$

How does this method resolve collisions?
Chain together when collision occurs. Typically implemented using linked list. One of the more popular choices.

Advantages: Simple to implement, usually an array of linked list. Therefore hash table never fills up since we can keep adding to the linked list.

Issues: If we get a long chain, search time can become $O(n)$ in the worst case. Think of a case where all values map to a single index thus forming one long linked list. Now when searching for this value, we essentially have a linear search along that single linked list.
This method also consumes extra space.

Final notes for your exam

- If table size not given, use one more than the values needed to insert. For example, if told to insert: 19, 50, 89, 39, use tablesize 4+1 = 5
- If h_1 missing when double hashing, assume $h(x) = x \%$ tablesize.
- never skip a free response if something similar missing. make an assumption and explain how the method works.
- get used to calculating multiple collisions. Very unlikely that you'll get ∞ loop on exam.

Please inform me if any mistakes are found in these notes!