COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

* Some slides by Henry Kautz

Hashing

Data Structures so far

	unsorted list	sorted array	Trees BST – average AVL – worst case
insert	θ(n)	θ(n)	θ(log n)
find	θ(n)	θ(log n)	θ(log n)
remove	θ(n)	θ(n)	θ(log n)

Faster ADT

What if $\theta(\log n)$ is still to big?

Internet has grown to millions of users generating terabytes of content every day

With such large data sets, how do we find anything?

Hash-Tables

- Suppose our intent is to find an item in O(1)
  - That is, constant time or time does not depend on data size n
- In most cases, we only care about
  - Finding and retrieving things quickly
  - Updating and inserting things quickly
- We do not care about
  - Order statistics of the data

Hash-Tables

- Strategy: Hashing
- Data structure: Hash-Tables

Hash-Tables: Basic Idea

- Use a key (arbitrary string or number) to index directly into an array – O(1) time to access records
  - A[“kreplach”] = “tasty stuffed dough”
  - Need a hash function to convert the key to an integer: h(“kiwi”) = 2

	Key	Data
0	kim chi	spicy cabbage
1	kreplach	tasty stuffed dough
2	kiwi	Australian fruit

Hash Functions

- A hash function maps a key to a value
- Simplest form:
  - A[i] - key is an integer
- Keys can be anything
  - strings, objects, …

Properties of Good Hash Functions

- Must return number 0, …, tablesize
- Equal keys should be mapped to the same index:
  - $x = y \quad \Rightarrow \quad h(x) = h(y)$
- Should be efficiently computable – O(1) time
- Should not waste space unnecessarily
  - For every index, there is at least one key that hashes to it
  - Load factor lambda λ = (number of keys / TableSize)
- Should minimize collisions

  = different keys hashing to same index in the hash-table

Examples

- Idealistic goal: distribute the keys uniformly.
  - Efficiently computable.
  - Each table position equally likely for each key.
- Practical challenge: need different approach for each type of key
  - Ex: Social Security numbers.
  - Ex: Phone numbers
  - Ex: date of birth

Integer Keys

- Hash(x) = x % TableSize
  - Too many collisions
  - Not applicable to many types

Strings as Keys

- If keys are strings, can get an integer by adding up ASCII values of characters in *key*
  - A string is simply an array of bytes:
  - Each byte stores a value from 0 to 255

```
for (i=0;i<key.length();i++)
   hashVal += key.charAt(i);
```

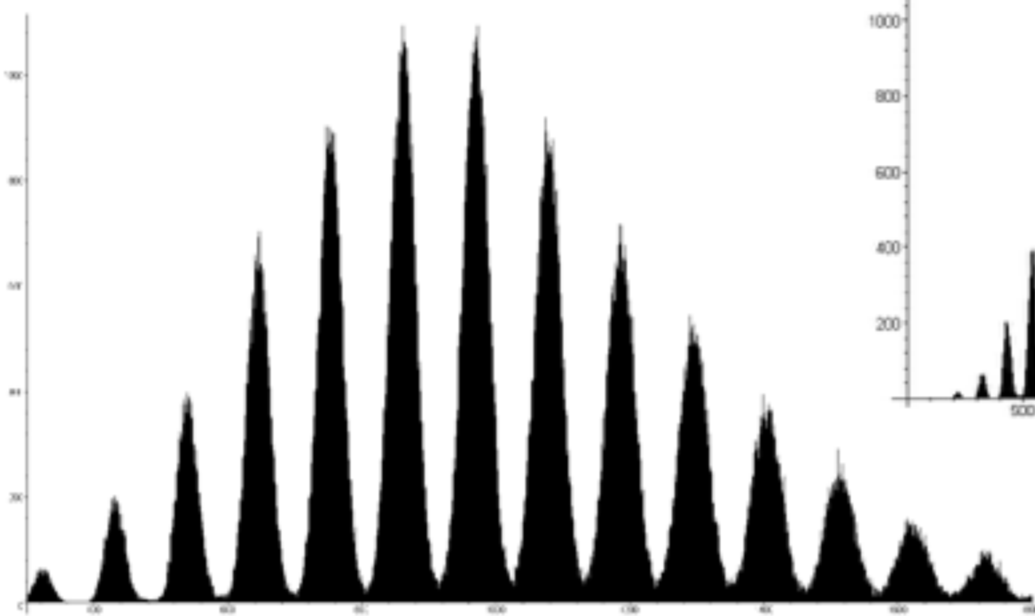
Strings as Keys

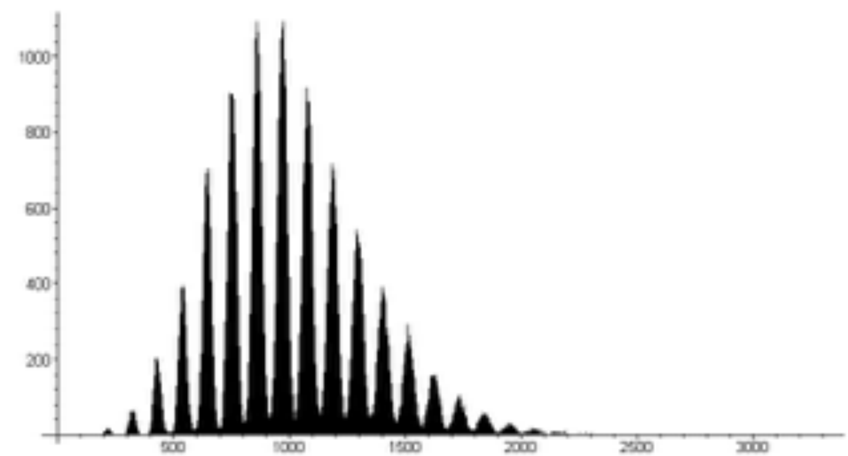
- Problem1: What if $TableSize$ is 10,000 and all keys are 8 or less characters long?
- Problem2: What if keys often contain the same characters ("abc", "bca", etc.)?

Strings as Keys

Not very good:

- A poor distribution
- Words with the same characters hash to the same code:
  - "form" and "from"
- Slow run time: $\Theta(n)$





diagrams by Douglas W. Harder

Hashing Strings

Let the individual characters represent the coefficients of a polynomial in x:

$$c_0 x^{n-1} + c_1 x^{n-2} + \cdots + c_{n-3} x^2 + c_{n-2} x + c_{n-1}$$

Then apply integer keys:

$$(c_0 x^{n-1} + c_1 x^{n-2} + \cdots + c_{n-3} x^2 + c_{n-2} x + c_{n-1})\%\text{TableSize}$$

E.*g*., $x = 128$,

$$h(\text{“abc”}) = (\text{“}a\text{”}\, 128^2 + \text{“}b\text{”}\, 128^1 + \text{“}c\text{”})\%\text{TableSize}$$

Hashing Strings

Problem: although a char can hold 128 values (8 bits), only a subset of these values are commonly used (26 letters plus some special characters)

- So just use a smaller "base"

$$h(\text{“abc”}) = (\text{‘}a\text{’}\ 32^2 + \text{‘}b\text{’}\ 32^1 + \text{‘}c\text{’})\%\text{TableSize}$$

Making the String Hash Easy to Compute

```
int hash(String s) {
  h = 0;
  for (i = s.length() - 1; i >= 0; i--) {
    h = (s.keyAt(i) + h<<5) % tableSize;
  }
  return h;
}
```

What is happening here???

- Advantages:

How Can You Hash…

- A set of values – (name, birthdate) ?

- An arbitrary pointer in C?

- An arbitrary reference to an object in Java?

How Can You Hash…

- A set of values – (name, birthdate) ?

  (Hash(name) ^ Hash(birthdate))% tablesize

  What's this?

- An arbitrary pointer in C?

  ((int)p) % tablesize

- An arbitrary reference to an object in Java?

  Hash(obj.toString())

  or just obj.hashCode() % tablesize

Collisions and their Resolution

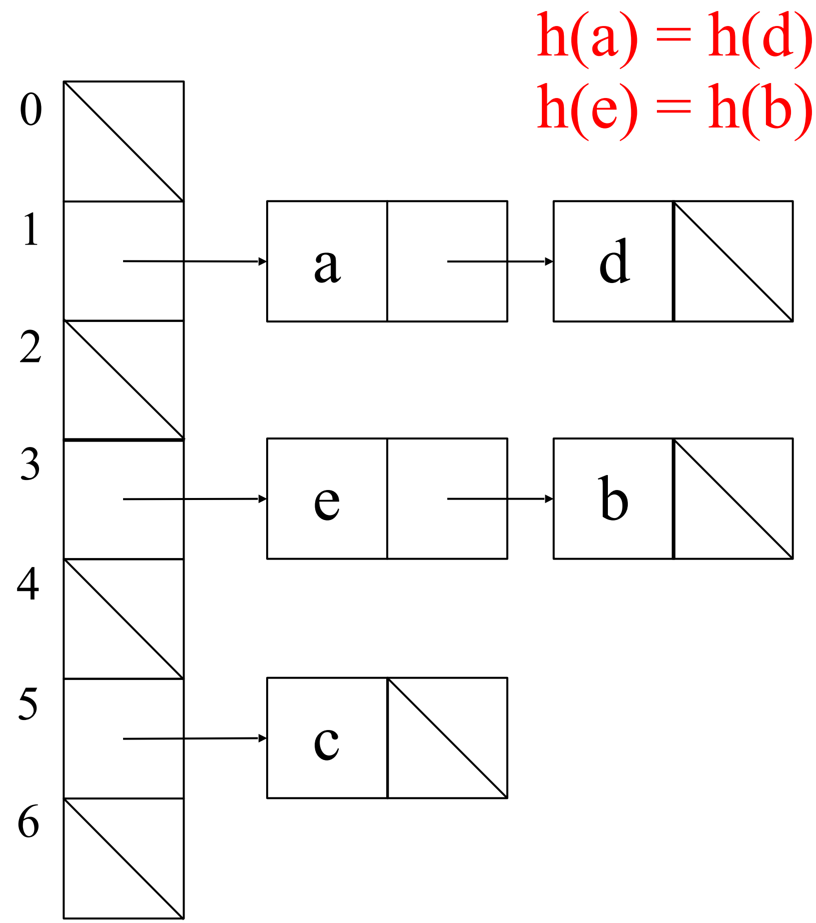
- A collision occurs when two different keys hash to the same value
  - E.g. For *TableSize* = 17, the keys 18 and 35 hash to the same value
  - 18 mod 17 = 1 and 35 mod 17 = 1
- Cannot store both data records in the same slot in array!
- Two different methods for collision resolution:
  - Separate Chaining: Use a dictionary data structure (such as a linked list) to store multiple items that hash to the same slot
  - Closed Hashing (or *probing*): search for empty slots using a second function and store item in first empty slot that is found

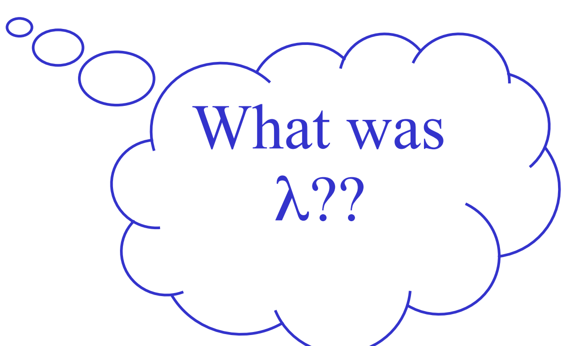
A Rose by Any Other Name…

- Separate chaining = Open hashing
- Closed hashing = Open addressing

Hashing with Separate Chaining

- Put a little container at each entry
  - choose type as appropriate
  - common case is unordered linked list (*chain*)
- Properties
  - performance degrades with length of chains
  - **λ can be greater than 1**





Load Factor with Separate Chaining

- Search cost (assuming simple uniform hashing)

- Load factor:

Load Factor with Separate Chaining

- Search cost (assuming simple uniform hashing)

  linear in terms of λ, $O(\lambda)$

- Load factor:
  - λ is not bound by 1; it can be >1.
  - But if λ is between ½ and 1 is fast and makes good use of memory.

Alternative Strategy: Closed Hashing

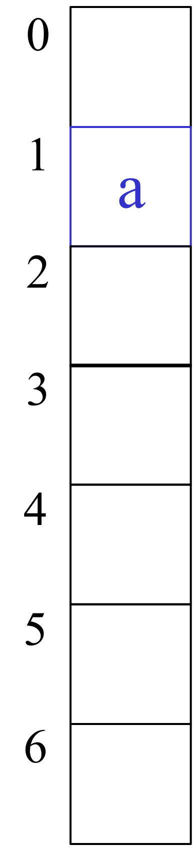
Problem with separate chaining:

Memory consumed by pointers –
32 (or 64) bits per key!

What if we only allow one Key at each entry?

- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must *go in another spot*

$h(a) = 1$

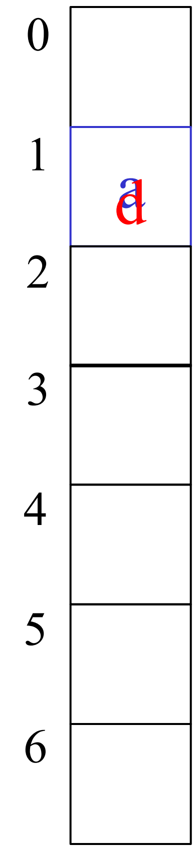


Alternative Strategy: Closed Hashing

Problem with separate chaining:

Memory consumed by pointers – 32 (or 64) bits per key!

h(a) = 1
h(d) = 1



What if we only allow one Key at each entry?

- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must *go in another spot*

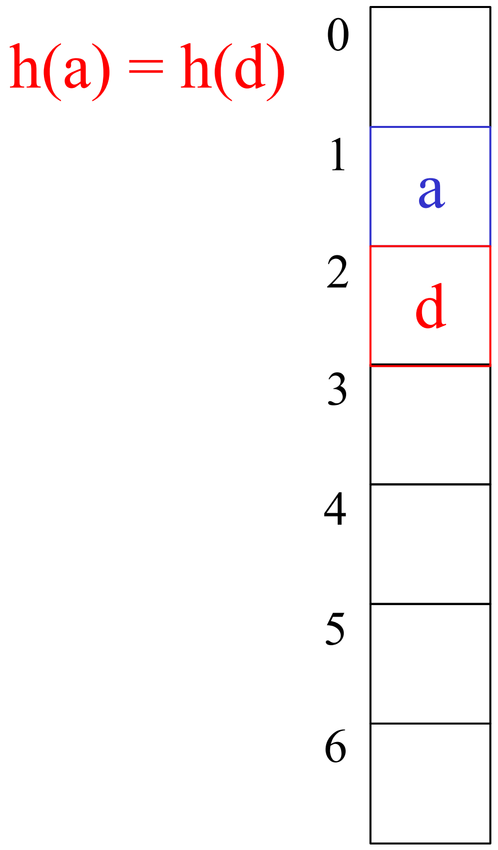
Alternative Strategy: Closed Hashing

Problem with separate chaining:

Memory consumed by pointers –
32 (or 64) bits per key!

What if we only allow one Key at each entry?

- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must *go in another spot*

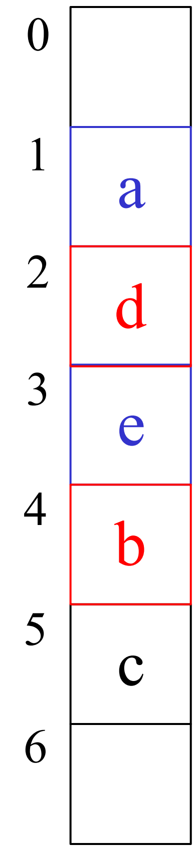


Alternative Strategy: Closed Hashing

Problem with separate chaining:

Memory consumed by pointers –
32 (or 64) bits per key!

$h(a) = h(d)$
$h(e) = h(b)$

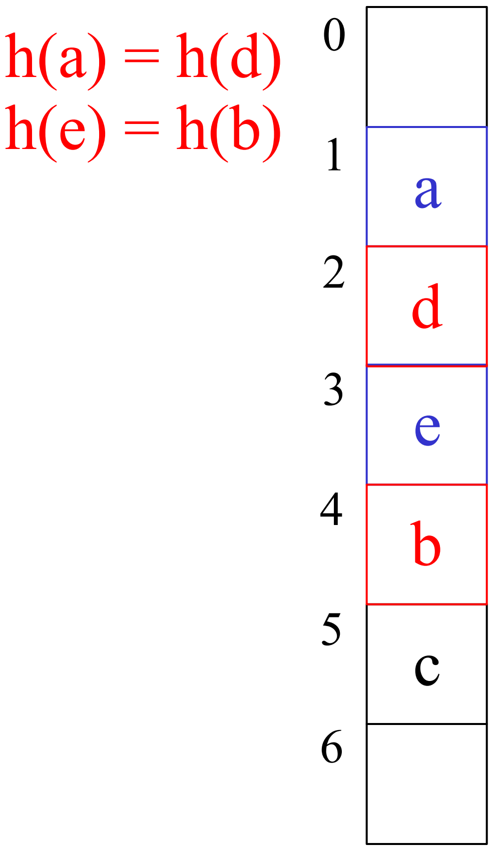


What if we only allow one Key at each entry?
- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must *go in another spot*

Alternative Strategy: Closed Hashing

Problem with separate chaining:

Memory consumed by pointers – 32 (or 64) bits per key!



What if we only allow one Key at each entry?

- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must *go in another spot*

- Properties
  - $\lambda \leq 1$
  - performance degrades with **difficulty of finding** right spot

Collision Resolution by Closed Hashing

- Given an item X, try cells $h_0(X), h_1(X), h_2(X), \ldots, h_i(X)$
- $h_i(X) = (Hash(X) + F(i)) \bmod \mathit{TableSize}$
  - Define $F(0) = 0$
- F is the *collision resolution* function. Some possibilities:
  - Linear: $F(i) = i$
  - Quadratic: $F(i) = i^2$
  - Double Hashing: $F(i) = i Hash_2(X)$

Closed Hashing I: Linear Probing

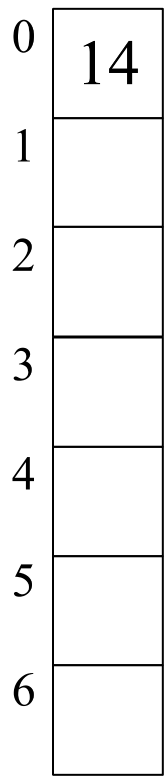
- Main Idea: When collision occurs, scan down the array one cell at a time looking for an empty cell
  - $h_i(X) = (\text{Hash}(X) + i) \bmod \textit{TableSize}$ $(i = 0, 1, 2, \ldots)$
  - Compute hash value and increment it until a free cell is found

Linear Probing Example

insert(14)

14%7 = 0

TableSize = 7



probes: 1

Linear Probing Example

TableSize = 7

	insert(14) 14%7 = 0	insert(8) 8%7 = 1
0	14	14
1		8
2		
3		
4		
5		
6		
probes:	1	1

Linear Probing Example

TableSize = 7

	insert(14)	insert(8)	insert(21)
	14%7 = 0	8%7 = 1	21%7 =0
0	14	14	14
1		8	8
2			21
3			
4			
5			
6			
probes:	1	1	3

(21+1)%7 =1

(21+2)%7 =2

Linear Probing Example

TableSize = 7

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2
0	14	14	14	14
1		8	8	8
2			21	12
3				2
4				
5				
6				
probes:	1	1	3	2

(2+1)%7 =3

Drawbacks of Linear Probing

- Works until array is full, but as number of items N approaches *TableSize* ($\lambda \approx 1$), access time approaches O(N)
- Very prone to clustering problem (as in our example)
  - As long as table is big enough, a free cell can always be found, but the time to do so can get quite large.
  - Worse: even if the cluster is relatively empty, blocks of occupied cells start forming. This effect is know as:
  - *Primary clustering – clusters grow when keys hash to values close to each other*
  - Does not satisfy good hash function criterion of *distributing keys uniformly*

Closed Hashing II: Quadratic Probing

- Main Idea: Spread out the search for an empty slot – Increment by i^2 instead of i

- $h_i(X) = (\mathrm{Hash}(X) + i^2) \ \% \ \mathit{TableSize}$

  h0(X) = Hash(X) % TableSize

  h1(X) = Hash(X) + 1 % TableSize
- h2(X) = Hash(X) + 4 % TableSize
- h3(X) = Hash(X) + 9 % TableSize

Quadratic Probing Example

insert(14)

$14\%7 = 0$

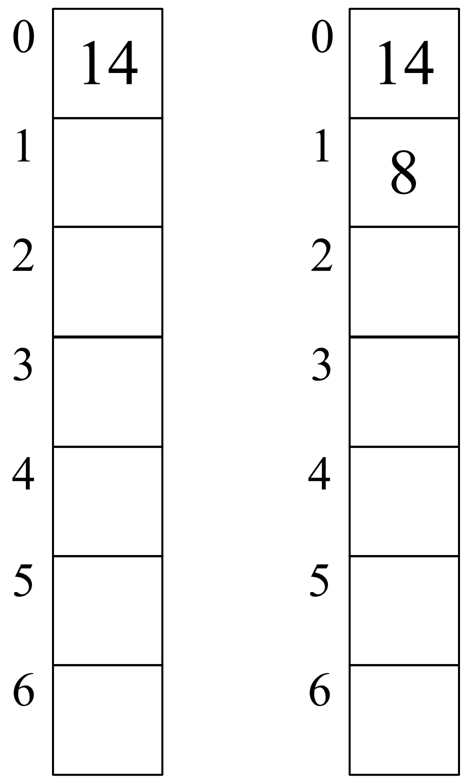
0	14
1	
2	
3	
4	
5	
6	

probes: 1

Quadratic Probing Example

insert(14) insert(8)

14%7 = 0 8%7 = 1



probes: 1 1

Quadratic Probing Example

	insert(14)	insert(8)	insert(21)
	$14\%7 = 0$	$8\%7 = 1$	$21\%7 = 0$
0	14	14	14
1		8	8
2			
3			
4			21
5			
6			
probes:	1	1	3

$(21+1)\%7 = 1$

$(21+2^2)\%7 = 4$

Quadratic Probing Example

	insert(14)	insert(8)	insert(21)	insert(2)
	14%7 = 0	8%7 = 1	21%7 =0	2%7 = 2
0	14	14	14	14
1		8	8	8
2				2
3				
4			21	21
5				
6				
probes:	1	1	3	1

Problem With Quadratic Probing

	insert(14)	insert(8)	insert(21)	insert(2)	insert(7)
	14%7 = 0	8%7 = 1	21%7 =0	2%7 = 2	7%7 = 0
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					
probes:	1	1	3	1	??

$(7+1)\%7 = 1$

$(7+2^2)\%7 = 4$

$(7+3^2)\%7 = 2$

$(7+4^2)\%7 = 2$

$(7+5^2)\%7 = 4$

$(7+6^2)\%7 = 4$

$(7+7^2)\%7 = 0$

Closed Hashing II: Quadratic Probing

- Although quadratic probing works better than linear probing regarding to clustering problem, it is still prone to clustering problem which in this case is called *secondary clustering*

- Quadratic probing needs very large TableSize and cannot use the whole space of hash-table (just like the last example)
  - Usually ($\lambda \approx 0.5$) (means we just use half of the hast-table)

Closed Hashing III: Double Hashing

- **Idea**: Spread out the search for an empty slot by using a second hash function
  - *No primary or secondary clustering*
- $h_i(X) = (Hash_1(X) + i*Hash_2(X)) \bmod \mathit{TableSize}$
  for $i = 0, 1, 2, \ldots$
- Good choice of $Hash_2(X)$ can guarantee does not get "stuck" as long as $\lambda < 1$
  - Integer keys:
    $Hash_2(X) = R - (X \bmod R)$
    where R is a prime smaller than *TableSize*

Double Hashing Example

insert(14)

$14\%7 = 0$

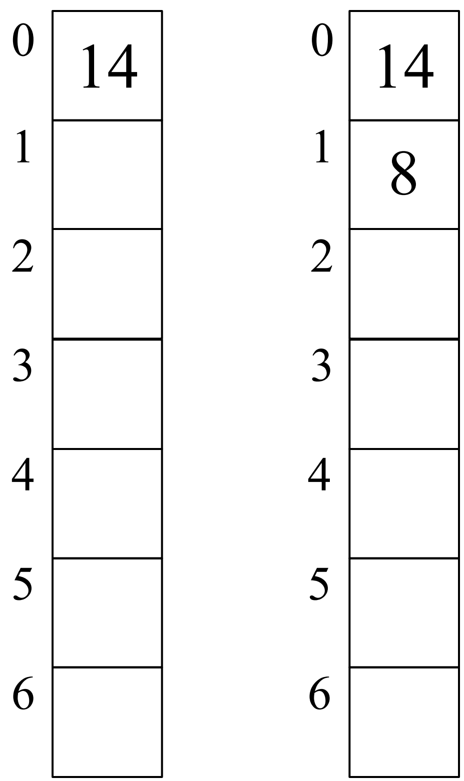
0	14
1	
2	
3	
4	
5	
6	

probes: 1

Double Hashing Example

insert(14) insert(8)

14%7 = 0 8%7 = 1



probes: 1 1

Double Hashing Example

tableSize:7

R:5

	insert(14)	insert(8)	insert(21)
	14%7 = 0	8%7 = 1	21%7 =0
			5-(21%5)=4
0	14	14	14
1		8	8
2			
3			
4			21
5			
6			
probes:	1	1	2

Double Hashing Example

tableSize:7

R:5

	insert(14)	insert(8)	insert(21)	insert(2)
	14%7 = 0	8%7 = 1	21%7 =0 5-(21%5)=4	2%7 = 2
0	14	14	14	14
1		8	8	8
2				2
3				
4			21	21
5				
6				
probes:	1	1	2	1

Double Hashing Example

tableSize:7

R:5

	insert(14)	insert(8)	insert(21)	insert(2)	insert(7)
	14%7 = 0	8%7 = 1	21%7 =0	2%7 = 2	7%7 = 0
			5-(21%5)=4		1*(5-(7%5))=3
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					7
4			21	21	21
5					
6					
probes:	1	1	2	1	2

compare it to quadratic probing