Searching II: Hashtables

- put
- get
- delete

- Intuition and Motivation
- Hash Functions
- Collision Resolution
- Hashtable Operations
- Analysis

If keys were just small, unique integers...

0	1	2	3	4	5	6	7	8	9	10
"apple"					~~"pear"~~ "peach"					

put(0, "apple")

put(5, "pear")

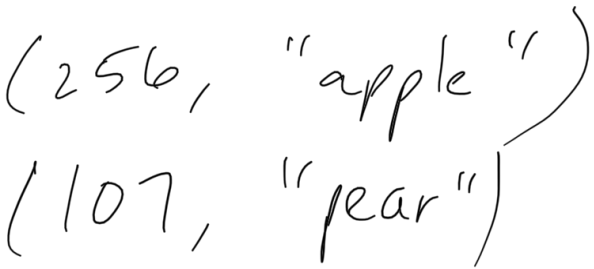
put(5, "peach")

What are some problems with this approach?

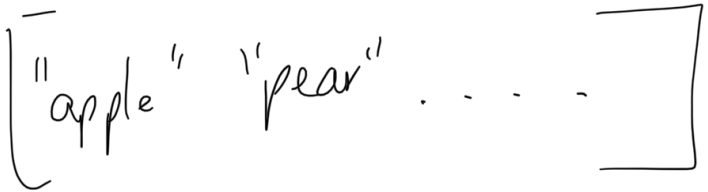
keys must be small integers

Time-Space Tradeoff

If we had unlimited space…



If we had unlimited time…



A Compromise

Hashtable Overview

- One way to implement a (unordered) dictionary
- Two parts:
  - table (size M)
  - hash function

# of pairs: N

Example

M = 11

0	1	2	3	4	5	6	7	8	9	10
				"BW B"	"CD CM"	"CK SP"	"MM G"			"GS Sting"

put("Clark Kent", "Superman")

put("Bruce Wayne", "Batman")

put("Gordon Sumner", "Sting")

* put("Marshall Mathers", "Eminem")

put("Carol Danvers", "Captain Marvel")

h("MM") = 6

What new questions does this raise?

Hash Functions

1. hash a value to get an integer
2. modular hashing

Hash Functions, continued

- The Java **hashCode** contract:

If $k_1 = k_2$, then $h(k_1) = h(k_2)$.

If $h(k_1) \neq h(k_2)$, then $k_1 \neq k_2$.

~~If $h(k_1) = h(k_2)$, then $k_1 = k_2$.~~

- Two additional goals for a good hash function:
  - → efficient
  - → uniformly distribute the keys

What does Java do?

```
public class Transaction
{
   ...
   private final String who;
   private final Date when;
   private final double amount;

   public int hashCode()
   {
      int hash = 17;
      hash = 31 * hash + who.hashCode();
      hash = 31 * hash + when.hashCode();
      hash = 31 * hash
         + ((Double) amount).hashCode();
      return hash;
   }
   ...
}
```

Picture from [1]

How to ~~avoid~~ minimize collisions…

1. using more of the key in the hash function
2. make M prime

1. Use as much of the key as possible.

2. Make the table size prime.

Experimenting with modular hashing…

- The following shows the input and output for an experiment with modular hashing to see why a prime modulus is better than a composite modulus.
- Input: a bunch of integers
- Output a: the number of times a particular value *v* occurs where *v* = *n % 96* and *n* is a given number from the input file.
- Output b: the number of times a particular value *v* occurs where *v* = *n % 97* and *n* is a given number from the input file.

Input 1: Random Data

test_output1a.txt

```
0:      5
1:      13
2:      15
3:      11
4:      13
5:      10
6:      6
7:      7
8:      16
9:      4
10:     15
11:     6
12:     12
13:     11
14:     11
15:     12
16:     14
17:     11
18:     13
19:     5
20:     11
21:     10
22:     12
23:     9
24:     11
25:     11
26:     11
27:     9
28:     12
29:     16
```

test_output1b.txt

```
0:      13
1:      11
2:      6
3:      9
4:      8
5:      9
6:      8
7:      15
8:      8
9:      6
10:     13
11:     15
12:     11
13:     12
14:     9
15:     9
16:     15
17:     9
18:     9
19:     9
20:     7
21:     11
22:     11
23:     13
24:     10
25:     10
26:     11
27:     14
28:     14
29:     13
```

Input 2: Multiples of 2

test_output2a.txt

```
0:      20
1:      0
2:      26
3:      0
4:      24
5:      0
6:      17
7:      0
8:      19
9:      0
10:     19
11:     0
12:     24
13:     0
14:     18
15:     0
16:     30
17:     0
18:     23
19:     0
20:     29
21:     0
22:     15
23:     0
24:     34
25:     0
26:     18
27:     0
28:     22
29:     0
```

test_output2b.txt

```
0:      19
1:      18
2:      9
3:      13
4:      9
5:      6
6:      17
7:      10
8:      7
9:      11
10:     9
11:     11
12:     14
13:     10
14:     13
15:     8
16:     10
17:     10
18:     17
19:     11
20:     18
21:     8
22:     8
23:     9
24:     13
25:     8
26:     12
27:     8
28:     7
29:     7
```

Input 3: Multiples of 2 & 3

test_output3a.txt

```
0:      22
1:      0
2:      7
3:      21
4:      10
5:      0
6:      22
7:      0
8:      11
9:      22
10:     5
11:     0
12:     23
13:     0
14:     13
15:     11
16:     11
17:     0
18:     23
19:     0
20:     11
21:     8
22:     10
23:     0
24:     32
25:     0
26:     18
27:     19
28:     9
29:     0
```

test_output3b.txt

```
0:      4
1:      8
2:      14
3:      8
4:      8
5:      10
6:      10
7:      10
8:      16
9:      10
10:     14
11:     11
12:     12
13:     16
14:     11
15:     11
16:     10
17:     9
18:     9
19:     10
20:     14
21:     12
22:     12
23:     15
24:     11
25:     7
26:     11
27:     9
28:     7
29:     7
```

Goal: Even distribution of hash values.

Hash value frequencies for words in *Tale of Two Cities* (10,679 keys, $M = 97$)

Picture from [1]

Hash Functions in Cryptography

NOTE: This is a side issue. The hash functions used in hashtables do not have to be cryptographically secure and are much simpler to design.

- maps data of arbitrary size to a fixed-length bit string (e.g. SHA-256 maps keys to 256 bits)
- easy to compute but hard to “undo”
- used for masking sensitive data (e.g. passwords)
- definitely want to minimize collisions as much as possible

Summary of Hash Functions

- Should be **consistent** (i.e. equal keys always hash to the same value)
- Should be **easy to compute** (if it's too complex, it could add significantly to the performance because the operations of the hashtable depend on computing the hash value.)

  What is the runtime for the Java hashcode function for Strings?

- Should **uniformly distribute** the set of keys (as much as possible)
- Can be used with modular hashing to get index values

Collision Managemer

- Separate Chaining
- Linear Probing
- Quadratic Probing

Image from https://clipartart.com/wallpaper/getimg.html

Method 1: Separate Chaining

0	1	2	3	4	5	6	7	8	9	10
		(13, "apple")				(17, "pear")				

(13, "apple")

(17, "pear")

(6, "peach")

(6, "peach")

best Worst

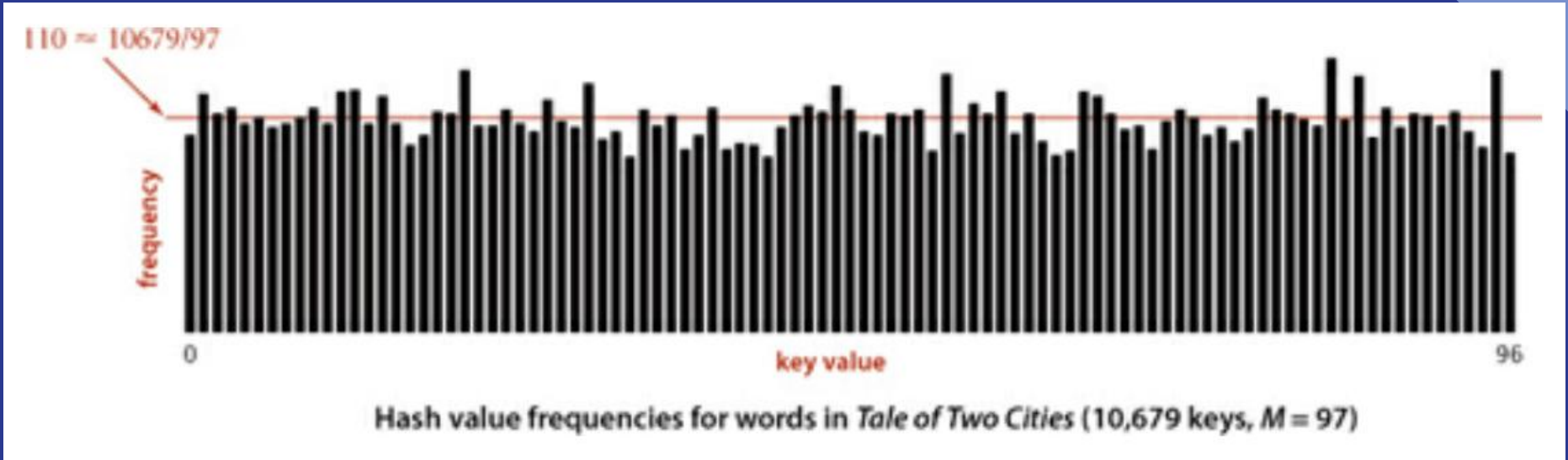
put
get
delete
} O(1) O(N)

expected: O(N/M)

Separate Chaining

- The hashtable is an array of lists.
- Collisions are managed by using the lists.
- This means that for *put* and *get*, the runtime is proportional to length of the list, which could be as good as O(1) and as bad as O(N), but is more accurately O(N/M) with a well-designed hash function and well-managed hashtable.
- The average length of the lists (N/M) is called the load factor (α) and is typically how you determine when to resize.

Example: Using Java's String *hashcode* function.



Hash value frequencies for words in *Tale of Two Cities* (10,679 keys, $M = 97$)

Picture from [1]

How do you choose a good M (size of table)?

- What is the relationship between **M** and the average length of the lists if you have **N** elements in the table?

How do you choose a good M (size of table)?

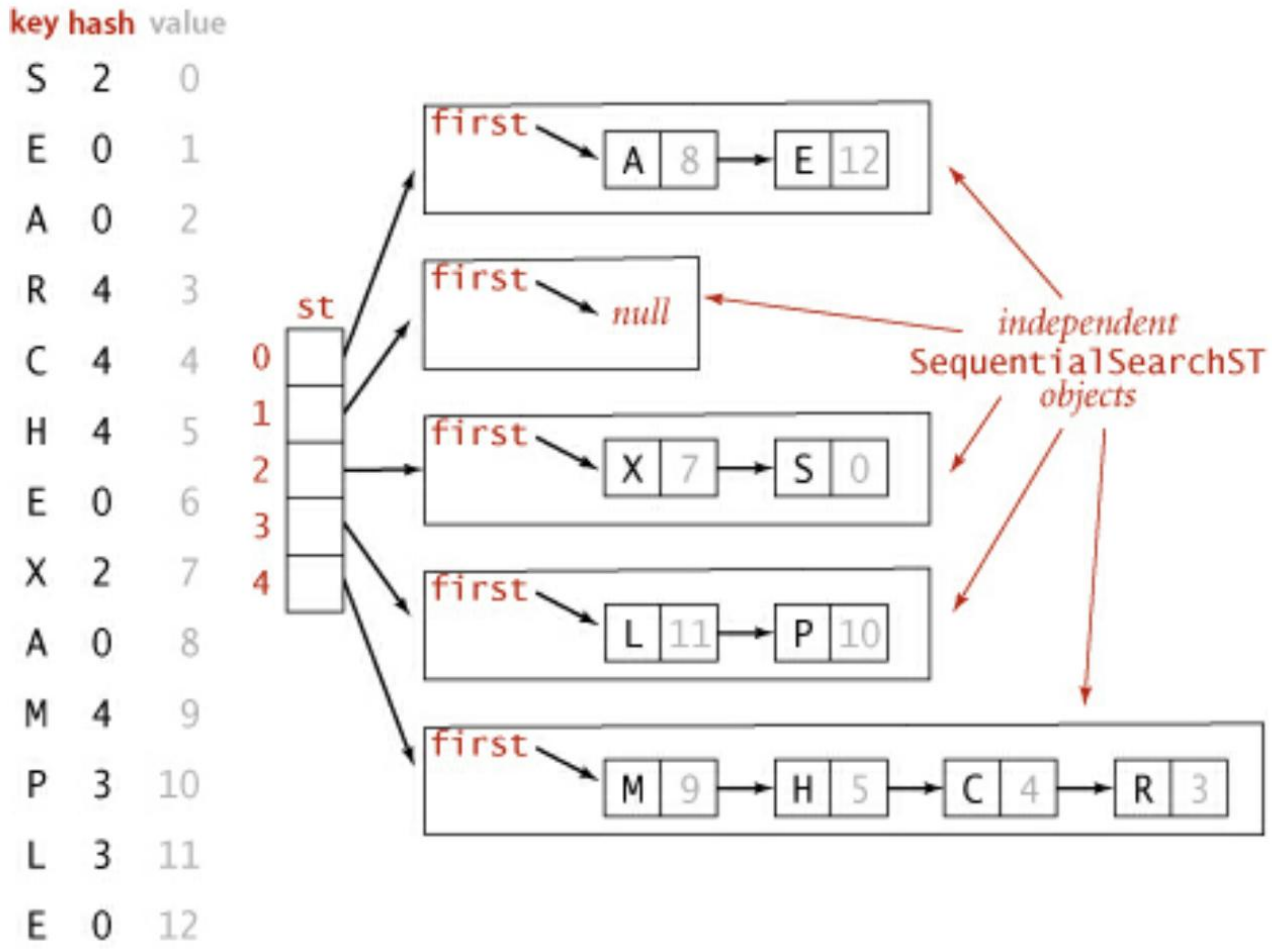
- What is the relationship between M and the average length of the lists?

- It’s a good idea to manage your table to maintain a reasonable load factor so that each of the operations can be done in a reasonable amount of time. The goal is to have it close to O(1). **How could you get it close to O(1)?**

Analysis

Analyze these operations where **M** is the size of the table and **N** is the number of key-value pairs:

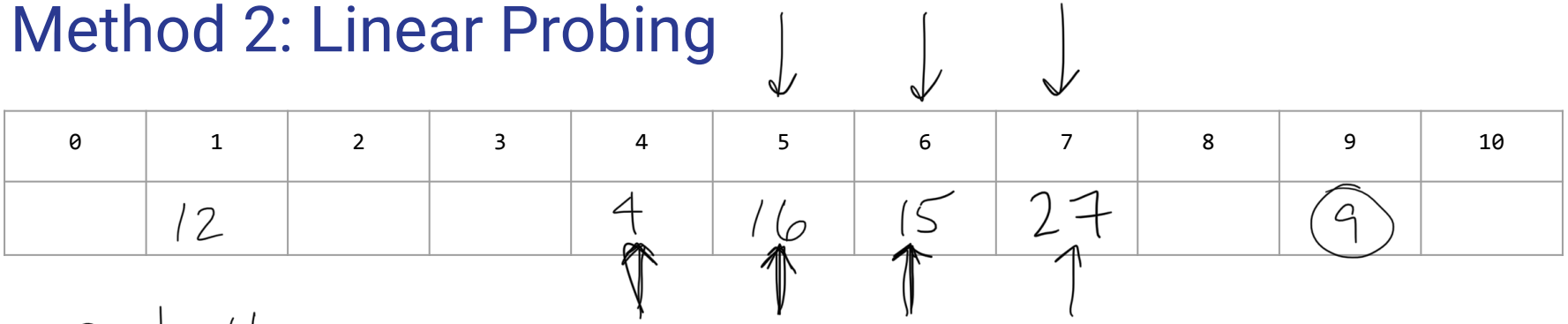
- **put(k, v)**
  - best case: $\Theta(1)$
  - worst-case: $O(N)$
  - expected case, assuming a good hash function that uniformly distributes keys: $\Theta(N/m)$
- **delete(k):** same
- **get(k):**



Hashing with separate chaining for standard indexing client

Picture from [1]

Method 2: Linear Probing



put 4
put 16
put 12
put 15

get 15
get 26
put 27

put 9

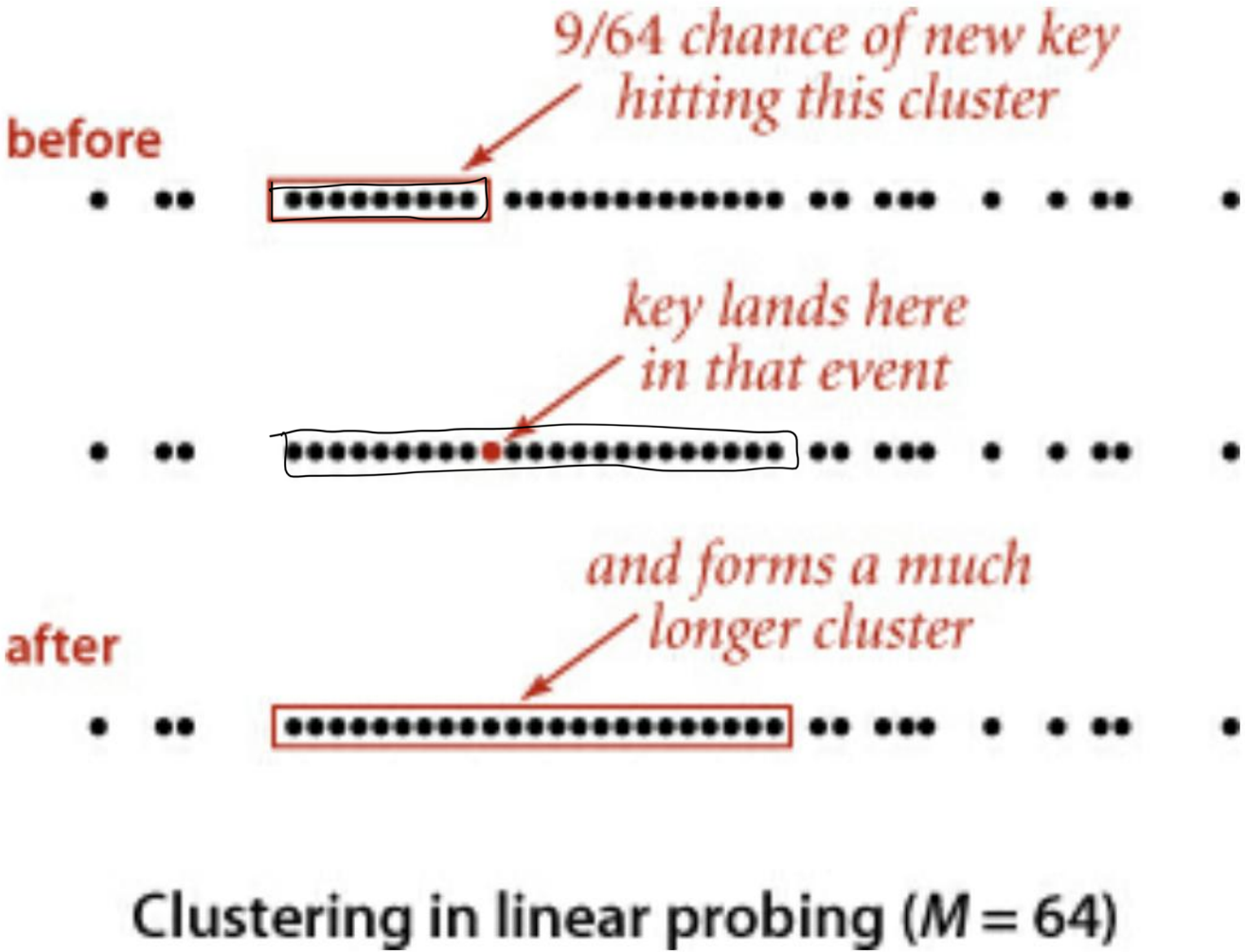
best: O(1)
worst: O(N)

Linear Probing

- The hashtable is an array.
- Collisions are managed by searching for an open spot as follows:
  - try h(k)
  - if that's taken, try h(k)+1 mod M
  - if that's taken, try h(k)+2 mod M
  - and so on…
- Typically, you want to maintain the hashtable to be between $\frac{1}{8}$ (α) and $\frac{1}{2}$ full to get good average runtimes for the basic operations.
- The percentage of the table that is filled is called the load factor and is used to determine when the table needs to be resized. N/M

Properties of Linear Probing

- **Null** used to mark division between clusters
- The longer a cluster, the longer the search time will be
- The longer a cluster, the more likely its length will increase



Clustering in linear probing ($M = 64$)

Picture from [1]

Load Factor: α

Separate Chaining

- How long are the lists?
- α = N/M (the average number of keys per list), which is often larger than 1

Linear Probing

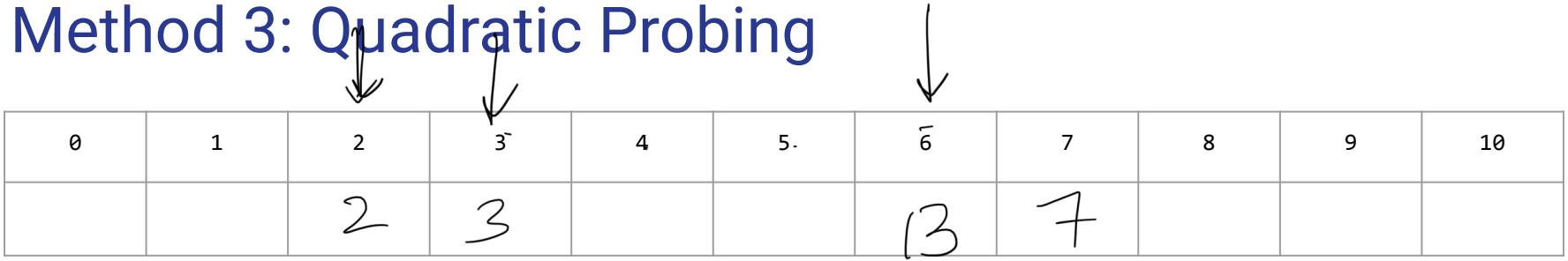
- How full is the array?
- α = N/M = fraction of table entries that are occupied (cannot be greater than 1)

Analysis of Hashtable with Linear Probing

- Good strategy: resize table to ensure that **α** is between ⅛ and ½ of the table (validated by mathematical analysis)
- When **α** is about ½, the average number of probes for search hit is about 3/2 and for search miss 5/2

expected runtime for get: ~O(1)

Method 3: Quadratic Probing



0	1	2	3	4	5	6	7	8	9	10
		2	3			13	7			

put 2
put 3
put 7
put 13

$h(k)$
$h(k) + 1^2 \pmod{M}$
$h(k) + 2^2 \pmod{M}$

Quadratic Probing

- Using linear probing, collisions are resolved by searching for an open spot in the following way:

$h(k), h(k)+1^2 \pmod M, h(k)+2^2 \pmod M, \ldots$

- Quadratic probing, the following spots are searched:

Deletion in hashtable using probing…

(linear)

0	1	2	3	4	5	6	7	8	9	10
		2	3	4	13					

put 13

delete 3

get 13

Summary of Hashtables

- Under **generous** assumptions, **insert** and **find** *average* runtimes can be reasonably estimated as constant
- Must have a good hash function for each type of key
- **Performance depends on the hash function**
- Hash functions may be difficult/expensive to compute
- Unless a hash function guarantees a one-to-one relationship between keys and hash values (unlikely), there needs to be some kind of strategy for handling collisions
  - Separate chaining
  - Probing (linear, quadratic, etc.)

Example. Using separate chaining and simple modular hashing, show how the following operations would be performed.

```
insert 7, insert 8, insert 17, delete 8, insert 14, insert 18, insert 11, insert 19,
insert 20, delete 17, insert 1, insert 3, insert 6
```

0	1	2	3	4	5	6	7	8	9	10

Example. Using linear probing and simple modular hashing, show how the following operations would be performed.

`insert 7, insert 8, insert 17, delete 8, insert 14, insert 18, insert 11, insert 19, insert 20, delete 17, insert 1, insert 3, insert 6`

0	1	2	3	4	5	6	7	8	9	10

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	

Example. Using quadratic probing and simple modular hashing, show how the following operations would be performed.

`insert 7, insert 8, insert 17, delete 8, insert 14, insert 18, insert 11, insert 19, insert 20, delete 17, insert 1, insert 3, insert 6`

0	1	2	3	4	5	6	7	8	9	10

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	

References

[1] *Algorithms, Fourth Edition;* Robert Sedgewick and Kevin Wayne (and associated slides)
[2] Book slides from Goodrich and Tamassia