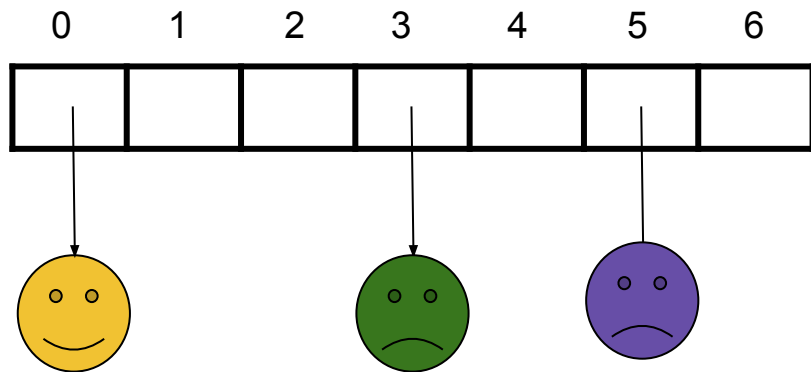


# Hashtables: *unordered* symbol tables

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

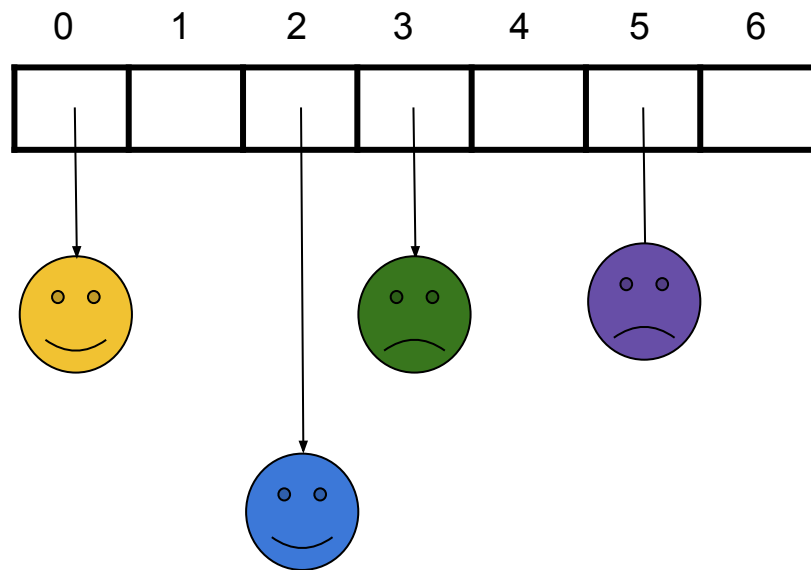
# If keys were just (small, unique) integers...

- ...we could just use an array to store the data
- ...then finding an item would be EASY because the key would just be the index!
- If you wanted to add a new (key, value) pair, that would be EASY (as long as there isn't already an item with that key)!



# If Keys were just (small) integers...

- ...we could just use an array to store the data
- ...then finding an item would be EASY because the key would just be the index!
- If you wanted to add a new (key, value) pair, that would be EASY (as long as there isn't already an item with that key)!



# Time-Space Tradeoff

If we had unlimited space, we could use a huge array and have **unique** integer keys for all possible data items (which may or may not be used!)

If we had unlimited time, we could use an unordered array and just use sequential search ( $O(N)$ ) to find an item by mapping each of the  $N$  keys to a **unique** index.

# A Compromise

- Since neither space nor time are unlimited, we can make a sort of compromise between the two and create a new structure called a *hashtable*.
- We will limit the space but we will use it wisely so that the searches are still possible in a reasonable amount of time.

# Hashtable Overview

- One way to implement an *unordered* symbol table
- Two parts:
  - Hash function  **$h(k)$**  that turns a key  **$k$**  into an integer (the result of  $h(k)$  is called the *hash value* of  $k$ )
  - An array (called a table) of size  **$M$**
- Main idea: Store item  **$(k, v)$**  at index  **$i = h(k)$**  in the array
- That means we need a hash function that turns  **$k$**  into an integer in the interval  $[0, M - 1]$ .

# Example

Table:  $N = 11$  (possible indices:  $[0, 10]$ )

|  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|

Values:



Keys: yellow, green, purple, blue

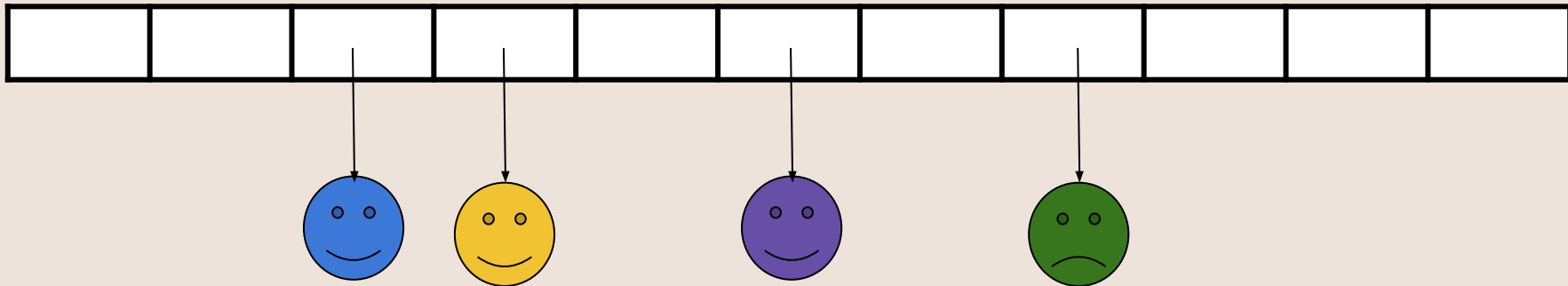
**Hash Function:** Let  $x$  be the numerical position in the alphabet of the first letter in the key. Return  $x \bmod 11$ .

$$h(\text{yellow}) = 25 \bmod 11 = 3$$

$$h(\text{green}) = 7 \bmod 11 = 7$$

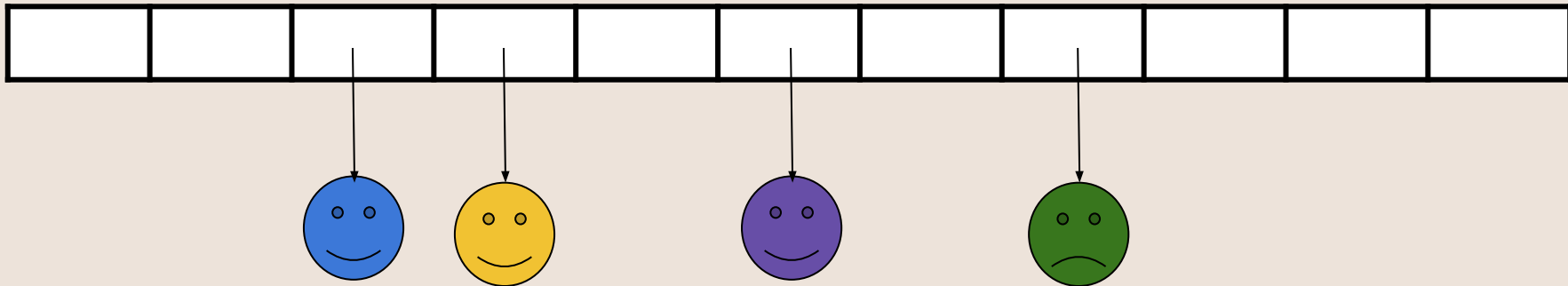
$$h(\text{purple}) = 16 \bmod 11 = 5$$

$$h(\text{blue}) = 2 \bmod 11 = 2$$



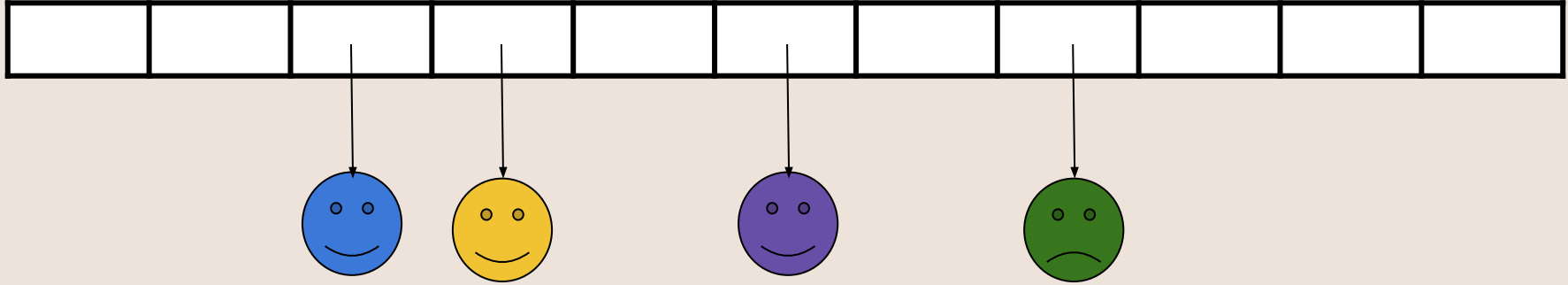
- **put(yellow, 😊):**
  - calculate  $h(\text{yellow}) = 3$
  - $a[3] = \text{😊}$
- **get(green):**
  - calculate  $h(\text{green}) = 7$
  - return  $a[7]$
- **delete(blue):**
  - calculate  $h(\text{blue}) = 2$
  - $a[2] = \text{null}$





- **put(yellow, 😊):**
  - calculate  $h(\text{yellow}) = 3$
  - $a[3] = \text{😊}$
- **get(green):**
  - calculate  $h(\text{green}) = 7$
  - return  $a[7]$
- **delete(blue):**
  - calculate  $h(\text{blue}) = 2$
  - $a[2] = \text{null}$

All  $O(1)$



**put(pink,  )**

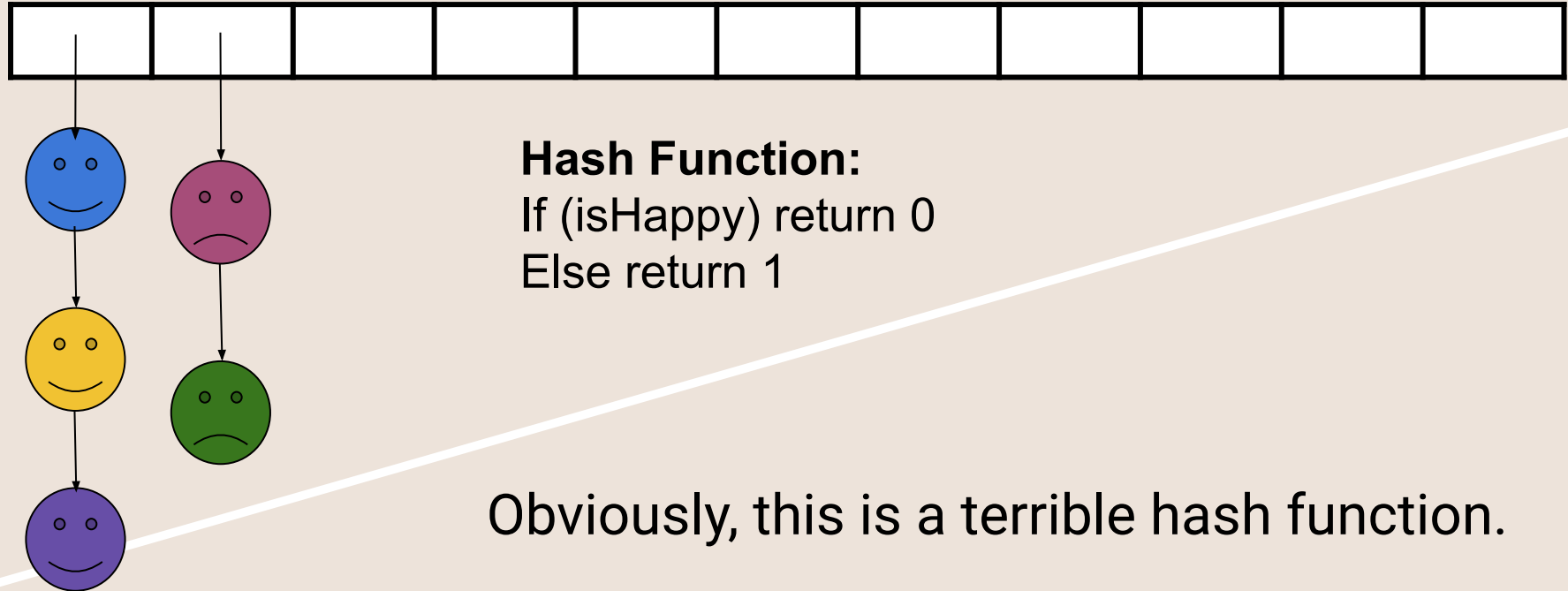
$$h(\text{pink}) = 16 \bmod 11 = 5$$

**Collision!!!**

# Collision Avoidance

- If possible, we would like to avoid collisions.
- To do that, we need
  - a good hash function
  - a well-managed table (particularly the ratio of items to the size of the table, which is called the *load factor*)
- Even so, we will not likely succeed in complete collision avoidance, so we will still have to deal with collision *management*.
- But in the meantime, let's consider what makes a good hash function.

# A good hash function should minimize collisions. This is not a good hash function.



So how can we design a hash function that minimizes collisions?

**Key Idea:** Use as much of the key as possible so as to make each key hash to something that is more likely to be unique.

**Example:** Design a hash function for Strings.

$$h(\text{"car"}) = 3 + 1 + 20 =$$

$$h(\text{"cab"}) \neq h(\text{"car"})$$

~~$$h(\text{"car"}) = h(\text{"arc"})$$~~

multiply the index

"a b c d e"  
└─┴─┘ └─┴─┘

"b c d e a"  
└─┴─┘ └─┴─┘

# Example: Java's String hashCode function.

Java:

[https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#hashCode\(\)](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#hashCode())

# Hash Function Basics

- The Java **hashCode** contract:
  - **hashCode(k)** always returns the same value as long as **k** hasn't changed (within the same application)
  - If  $k_1 = k_2$ , then  $h(k_1) = h(k_2)$  (IS THE CONVERSE TRUE?)
  - If  $h(k_1) \neq h(k_2)$ , then  $k_1 \neq k_2$  (Contrapositive)
- Two additional goals for a good hash function:
  - Easy to compute
  - Distribute keys uniformly (for each key, each hashcode value should be equally likely)



# Hash Function Basics

- The Java **hashCode** contract:
  - **hashCode(k)** always returns the same value as long as **k** hasn't changed (within the same application)
  - If  $k_1 = k_2$ , then  $h(k_1) = h(k_2)$  (IS THE CONVERSE TRUE?)
  - If  $h(k_1) \neq h(k_2)$ , then  $k_1 \neq k_2$  (Contrapositive)
- Two additional goals for a good hash function:
  - Easy to compute
  - Distribute keys uniformly (for each key, each hashcode value should be equally likely)

No, it isn't.  
That's why  
we have to  
handle  
collisions.

# Often, when using hashing for symbol tables...

- ...there are two parts to the hashing process:
  - First, the key gets hashed to an integer\* (this may be a large integer).
  - Then, the mod function is used to turn that integer into a smaller integer that is an index of the table. This is called *modular hashing*.
- Why is this a good approach?

\*The Java *hashCode* function can be used to turn a key into an integer. After that, you can just use modular hashing to fit the integer into the table according to the table size.

# Often, when using hashing for symbol tables...

- ...there are two parts to the hashing process:
  - First, the key gets hashed to an integer\* (this may be a large integer).
  - Then, the mod function is used to turn that integer into a smaller integer that is an index of the table. This is called *modular hashing*.
- Why is this a good approach?

It means you don't have to write an entirely new hash function every time you resize the table.

\*The Java *hashCode* function can be used to turn a key into an integer. After that, you can just use modular hashing to fit the integer into the table according to the table size.

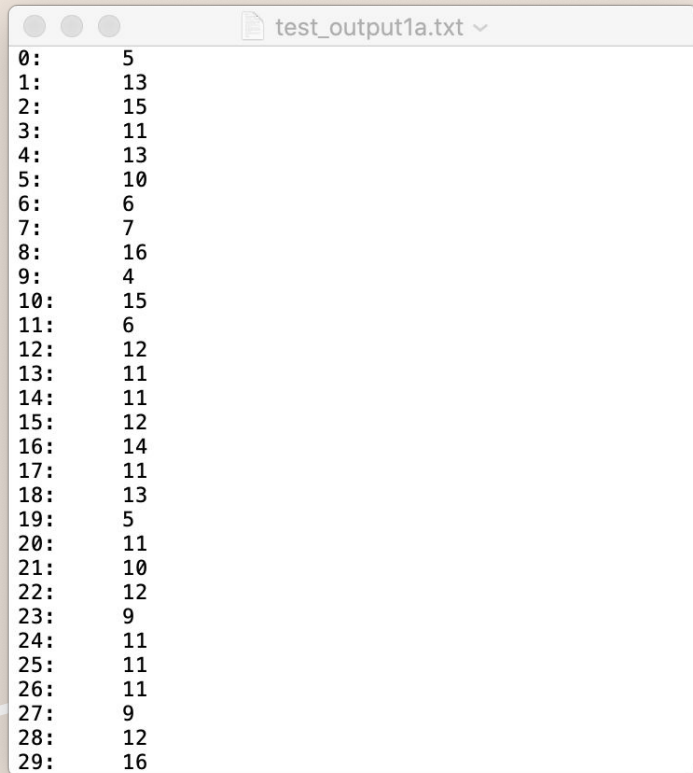
# Modular Hashing

- Typically choose  $M$  to be prime: WHY?
- Often used to turn a hashcode value of a key into an index that fits the hashtable.

# Experimenting with modular hashing...

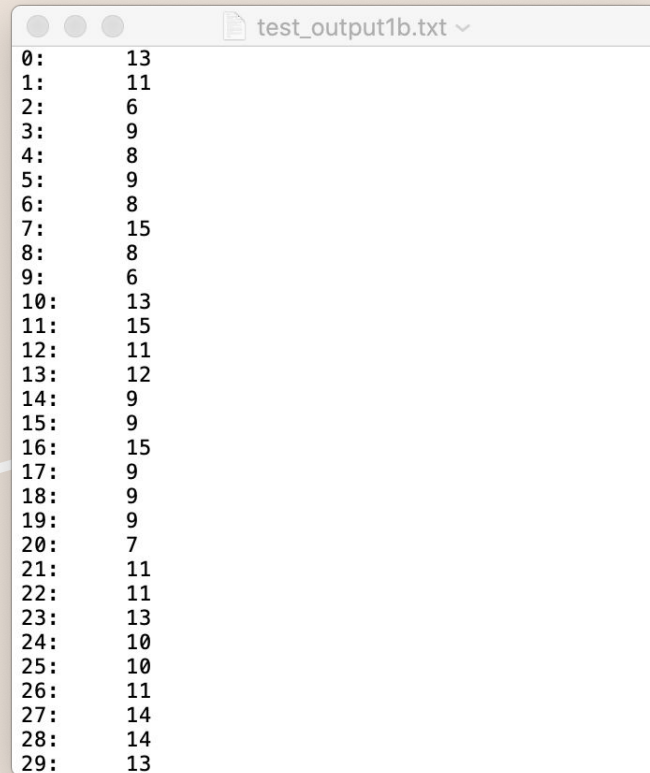
- The following shows the input and output for an experiment with modular hashing to see why a prime modulus is better than a composite modulus.
- Input: a bunch of integers
- Output a: the number of times a particular value  $v$  occurs where  $v = n \% 96$  and  $n$  is a given number from the input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



# Modular Hashing

- Typically choose  $M$  to be prime because it distributes the keys better when they aren't completely random.
- Can be used for various types of Keys: HOW? First turn the key into any integer, then do the modular hashing.
  - Integers
    - Use the actual value
  - Floating point values
    - Use the binary representation
  - Strings
    - Use both the integer representation of each character and the location of each character.

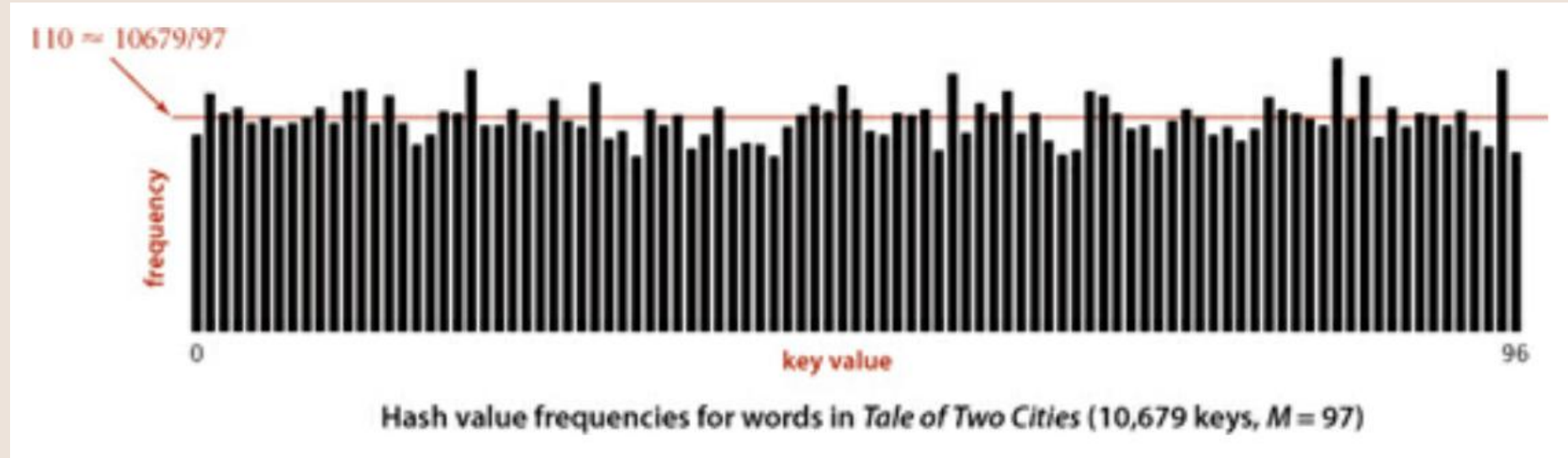
# What does Java do?

- **Integer**: uses the primitive **int** value
- **Double**: exclusive OR of the two halves of the **long** integer bit representation
- **String**:  $s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$  where **s[i]** is the numerical value of the  $i^{\text{th}}$  character and **n** is the length of the String
- **File**: system dependent (e.g. on UNIX, the exclusive OR of the hashCode of its pathname String and the decimal number 1234321)
- **User-defined Objects?** Use the hashCode functions for the individual fields and combine them in some way.
- The result of hashCode alone may not work as array index, so use modular hashing with it to get the array indices that you need.

```
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;
    }
    ...
}
```

Goal: Even distribution of hash values.



# 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?**

$O(\text{length of String})$

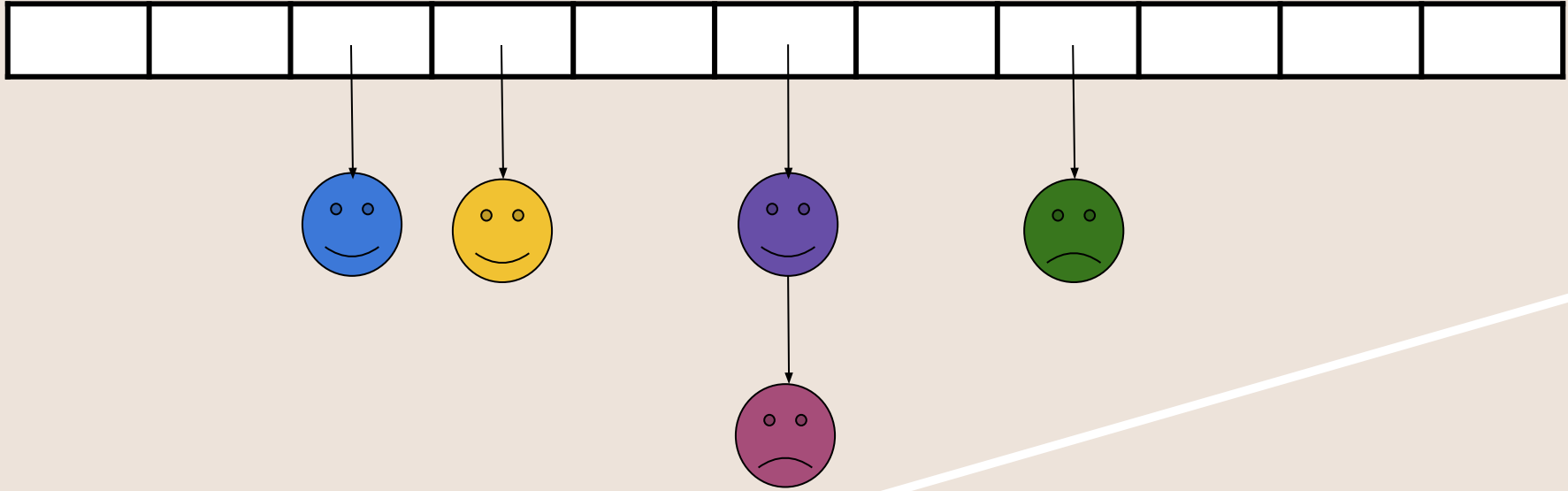
- Should **uniformly distribute** the set of keys (as much as possible)
- Can be used with modular hashing to get index values

# Collision Management

- Separate Chaining
- Linear Probing
- Quadratic Probing



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



## Option 1: Separate Chaining

`put(pink, )`

$$h(\text{pink}) = 16 \bmod 11 = 5$$



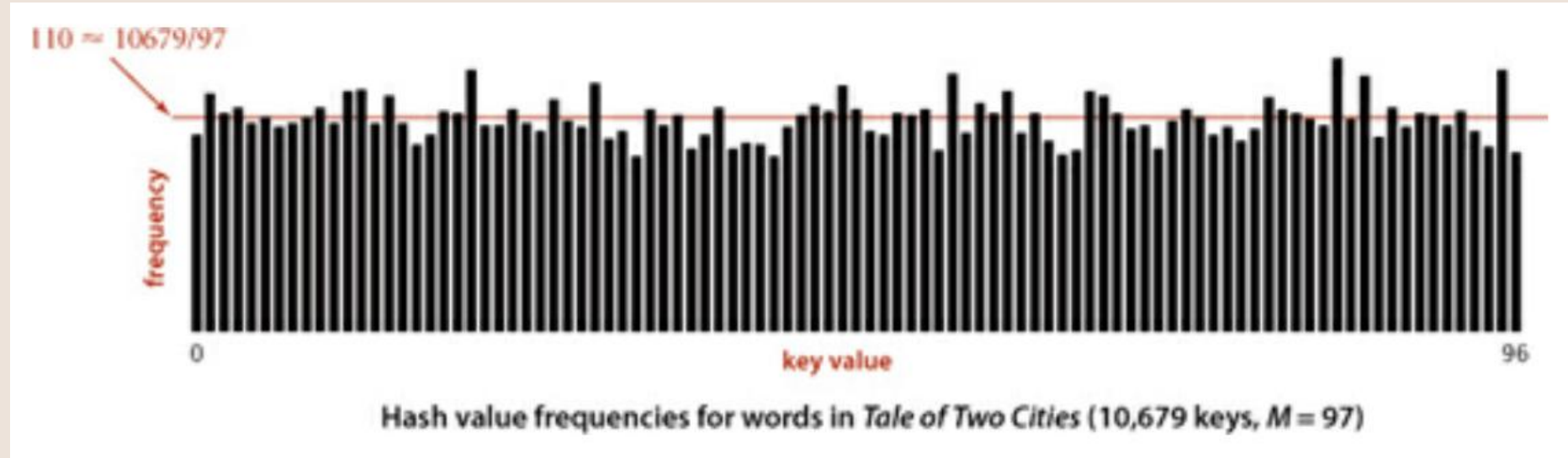
# Separate Chaining

- The hashtable is an array of lists.
- Collisions are managed by adding to lists when items hash to an existing list.
- This means that for *put* and *get*, the runtime is proportional to the length of the list in question, which could be...?

# Separate Chaining

- The hashtable is an array of linked lists.
- Collisions are managed by adding to lists when items hash to an existing list.
- This means that for *put* and *get*, the runtime is proportional to the length of the list in question, which could be... **$N$  in the worst case...BUT...**
- ...with a good hash function and a good table size, we expect the length of each list to be about  $N/M$ . This is called the load factor, indicated by  $\alpha$ .

# Example: Using Java's String *hashCode* function.



# 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?

**Bigger  $M$  means shorter 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 to 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:  **$O(1)$**
  - worst-case:  **$O(N)$**
  - expected case, assuming a good hash function that uniformly distributes keys,  **$O(N/M)$**
- **delete(k)**: same as **put**
- **get(k)**: same as **put**

What if  **$M \approx N$** ?

# Analysis

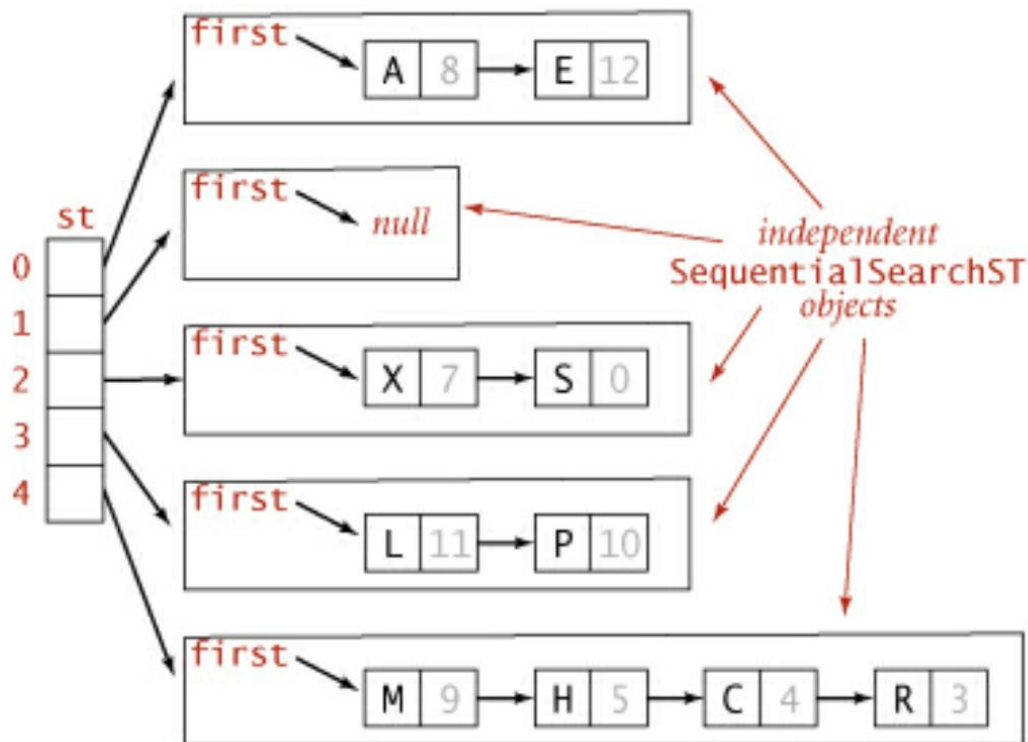
Analyze these operations where **M** is the size of the table and **N** is the number of key-value pairs:

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

What if  **$M \approx N$** ? The expected runtime becomes  **$O(1)$** !!!

key hash value

|   |   |    |
|---|---|----|
| S | 2 | 0  |
| E | 0 | 1  |
| A | 0 | 2  |
| R | 4 | 3  |
| C | 4 | 4  |
| H | 4 | 5  |
| E | 0 | 6  |
| X | 2 | 7  |
| A | 0 | 8  |
| M | 4 | 9  |
| P | 3 | 10 |
| L | 3 | 11 |
| E | 0 | 12 |



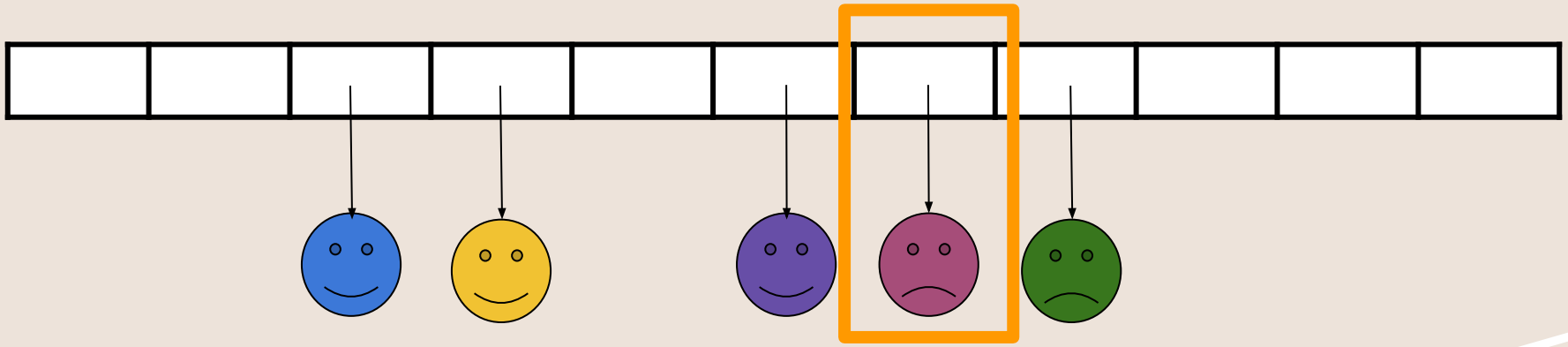
Hashing with separate chaining for standard indexing client



# Analysis

- Average length of lists:  $N/M$
- **If the hashCode distributes keys uniformly**, then it is extremely likely that each list will have about  $N/M$  values.
- In fact, the probability is close to 1 that the number of keys in a list is within a small factor of  $N/M$

**Which makes the number of compares for search misses and inserts  $\sim N/M$ .**



## Option 2: Linear Probing

`put(pink, )`

$$h(\text{pink}) = 16 \bmod 11 = 5$$

# Linear Probing

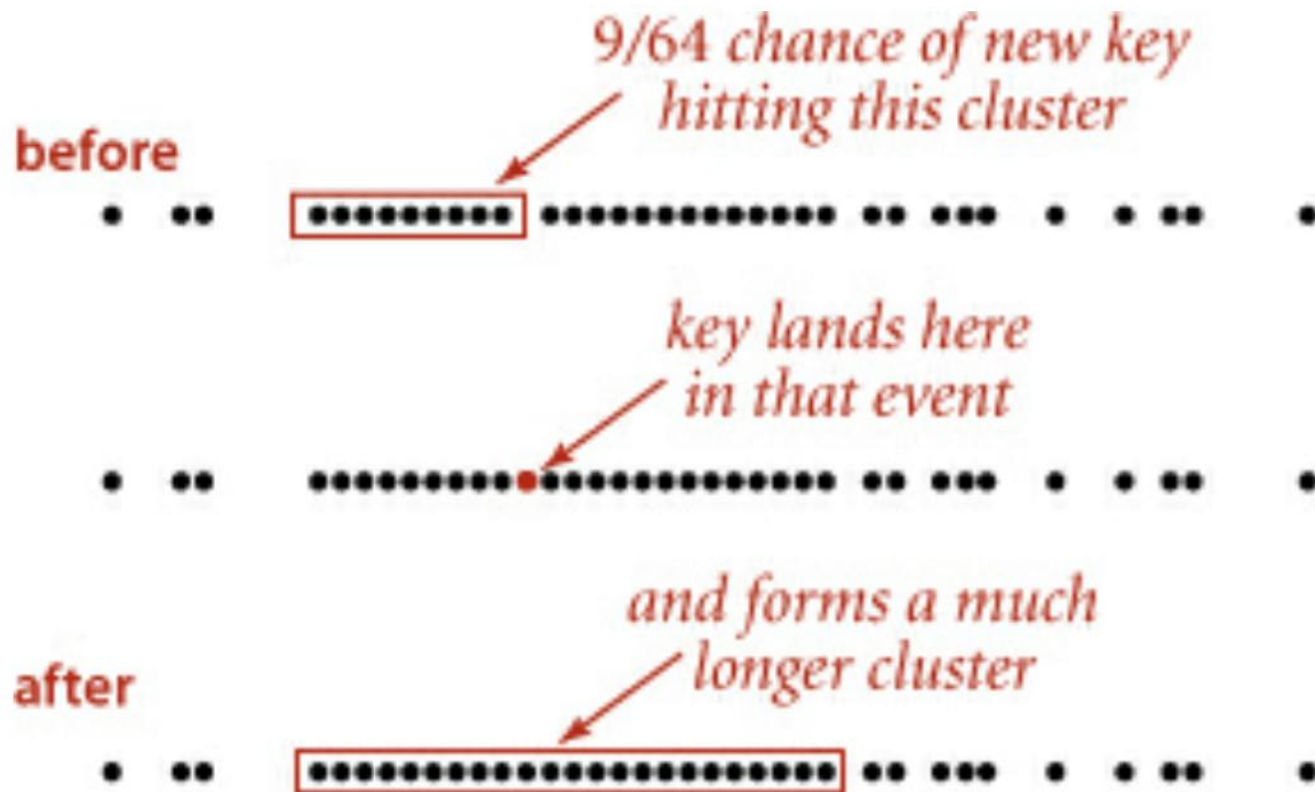
- The hashtable is an array.
- Collisions are managed by searching for an open spot as follows:
  - try  $h(k)$
  - if that's taken, try  $(h(k) + 1) \% M$
  - if that's taken, try  $(h(k) + 2) \% M$
  - and so on...
- Typically, you want to maintain the hashtable to be between  $\frac{1}{8}$  and  $\frac{1}{2}$  full to get good average runtimes for the basic operations. **This is called the load factor, indicated by  $\alpha$ . Note, although in this case  $\alpha < 1$ , it can still be calculated by  $N/M$ .**

# Finding a key

- Key is equal to search key (DONE!):  $O(1)$ —search hit
- Empty position at that index (DONE!):  $O(1)$ —search miss
- Key not equal to search key: Search next, next, etc...until...

# 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 ( $M = 64$ )

# Load Factor: $\alpha$

## Separate Chaining

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

## Linear Probing

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

# Analysis of Hashtable with Linear Probing

- Mathematically beyond the scope of this course
- Good strategy: resize table to ensure that  $\alpha$  is between  $\frac{1}{8}$  and  $\frac{1}{2}$  of the table (validated by mathematical analysis)
- When  $\alpha$  is about  $\frac{1}{2}$ , the average number of probes for search hit is about  $\frac{3}{2}$  and for search miss  $\frac{5}{2}$

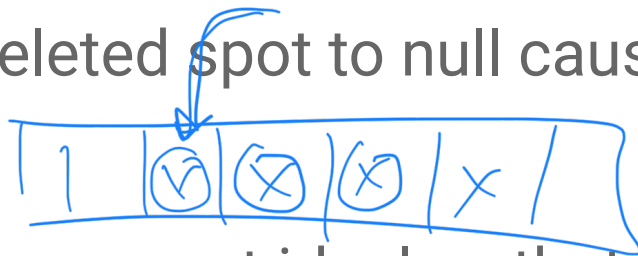


# Quadratic Probing

- Using linear probing, collisions are resolved by searching for an open spot in the following way:  $h(k)$ ,  $h(k)+1$ ,  $h(k) + 2...$
- Quadratic probing, the following spots are searched:  $h(k)$ ,  $h(k) + 1^2$ ,  $h(k) + 2^2$ ,  $h(k) + 3^2...$
- Introduces the possibility for infinite (or at least very long) loops in searching for an open spot, so you may need a limit on how many probes it will try before giving up and re-sizing the array.

# Deletion in a Hashtable that uses probing

Challenge: Simply setting the deleted spot to null causes problems for subsequent searches.



- ~~Option 1: re-hash all the values~~ → not ideal as that means  $O(N)$  time
- Option 2: leave the value in but indicate that it has been deleted (can be updated) → works well time-wise, and if you keep track of the number of “undeleted” items in the table as your “size,” then you will still resize as necessary

# 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 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 |
|----|---|---|----|---|---|----|---|----|----|----|
| 11 |   |   | 14 |   |   | 17 | 7 | 18 | 19 |    |

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

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 |
|----|----|---|----|---|---|----|---|---|----|----|
| 18 | 11 |   | 14 |   |   | 17 | 7 | 8 | 19 |    |

| 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 |
|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  |    | 3  |    |    | 6  | 7  |    |    |    | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |    |
|    |    | 14 |    |    | 17 | 18 | 19 | 20 |    |    | 53 |

# References

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