# ECS 32B - Hashing

*Aaron Kaloti*

UC Davis - Summer Session #2 2020

# Overview

- How a hash table works. Hash function.
- Collision resolution.
  - Separate chaining.
  - Open addressing.
    - Linear probing.
    - Quadratic probing.
    - Double hashing.
- Deletion; lazy deletion.
- Analysis. Worst-case time complexity. Table size.
- Rehashing.
- Hash table vs. self-balancing BST.

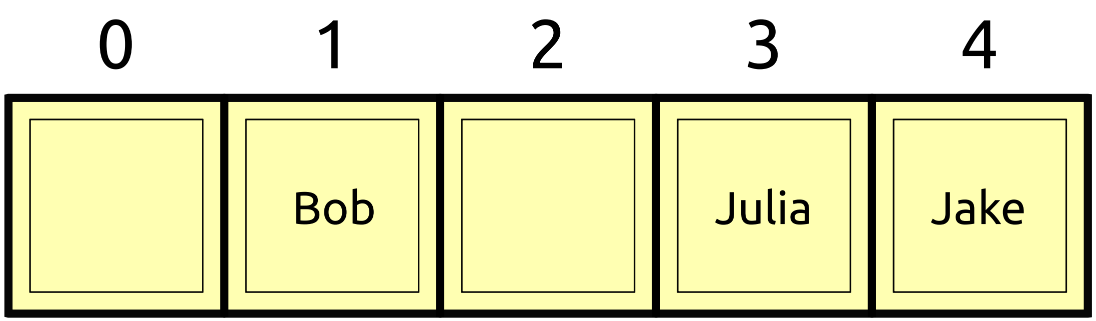
# Hash Table

## Description

- Support less operations than self-balancing BST in exchange for speed; sorted order of elements isn't maintained.
- Hash table's underlying implementation is Python list of $m$ slots/buckets.
- As with BSTs, location of element is influenced by its key.
- Hash function maps key to some number in range $[0, m - 1]$.

## Example

- Keys are names (strings).
- Hash function:
  - Maps "Bob" to 1, i.e. "Bob" *hashes* to 1.
  - Maps "Julia" to 3, i.e. "Julia" *hashes* to 3.
  - Maps "Jake" to 4, i.e. "Jake" *hashes* to 4.



# Hash Function

## Description

- Range of possible keys usually much larger than $m$ (number of slots/buckets).
- Want to distribute keys as evenly as possible.
- Assume keys are always integers for now $\Rightarrow$ typical hash function is $hash(x) = x \% m$.

## Example (Insertions)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

insert(14) -> place at 14 % 10 = 4:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 14 | | | | | |

insert(67) -> place at 67 % 10 = 7:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 14 | | | 67 | | |

insert(40) -> place at 40 % 10 = 0:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 40 | | | | 14 | | | 67 | | |

# Hash Function

- Can find an element by using hash function.

## Example: Found

find(97) -> check 97 % 10 = 7 -> 97 is there.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 21 | | 13 | | | 16 | 97 | | |

## Example: Not Found

find(46) -> check 46 % 10 = 6 -> 46 is not there.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 21 | | 13 | | | 16 | 97 | | |

# Collision Resolution

## Motivation

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 21 | | 13 | | | 16 | 97 | | |

insert(53) -> 53 % 10 = 3 -> but 13 is already there...

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 21 | | 13 | | | 16 | 97 | | |

- We must resolve this **collision**.

# Collision Resolution

- Two[1] ways to handle collision:
    1. Separate chaining.
    2. Open addressing.
        - Linear probing.
        - Quadratic probing.
        - Double hashing.

1. There are others; we won't get into them.

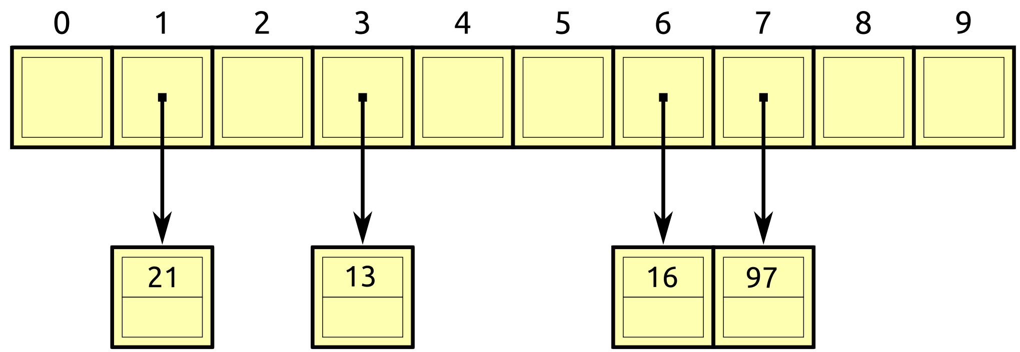
# Collision Resolution

## Separate Chaining

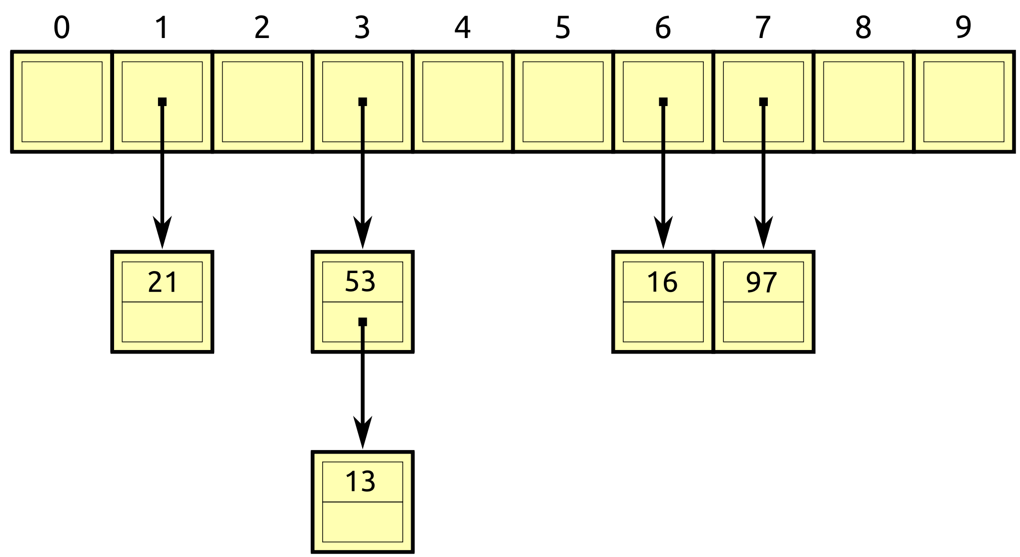
### Description

- Each bucket contains a linked list.
- Use hash function to determine which list to check.

### Example




insert(53) -> 53 % 10 = 3

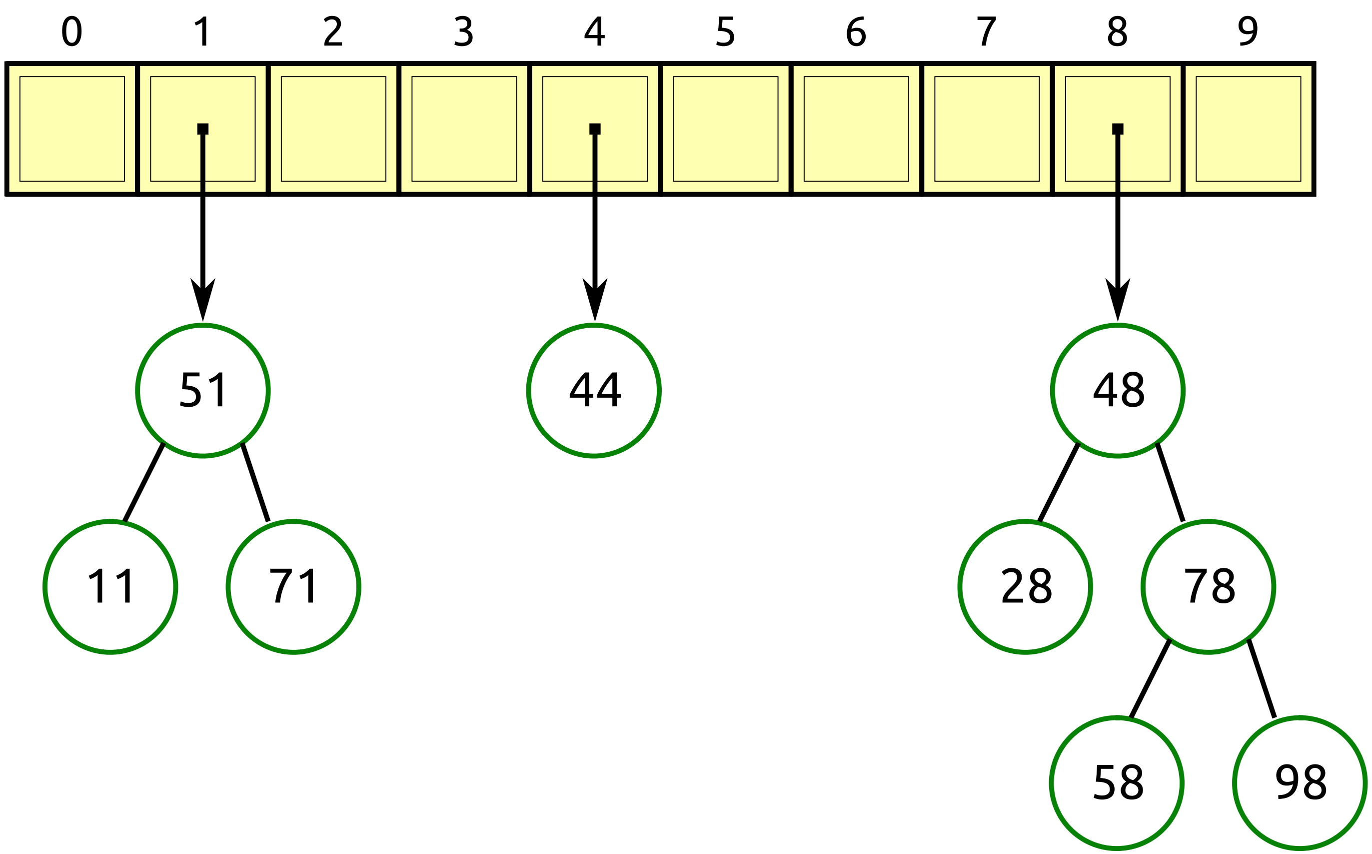


# Collision Resolution

## Separate Chaining

- Doesn't have to use linked lists, but probably should since elements per bucket is small if large table and good hash function.

### Example



# Collision Resolution

## Separate Chaining: Analysis

- **load factor** ($\lambda = \frac{n}{m}$): ratio of number of elements in hash table to table size.
- Average length of list is $\lambda$.
- Search involves: find list-to-traverse (takes constant time) and traverse said list.
  - On average, unsuccessful search checks $\lambda$ nodes.
- For separate chaining, load factor is more directly important than table size.

# Side Note: Prime Table Size

- Should keep $m$ as a prime number.
- Helps distribution of keys.

# Collision Resolution

## Open Addressing

- No linked lists.
- If collision occurs, try to place key at another bucket as determined by open addressing scheme. Repeat until successful.
- Three kinds:
  - Linear probing.
  - Quadratic probing.
  - Double hashing.

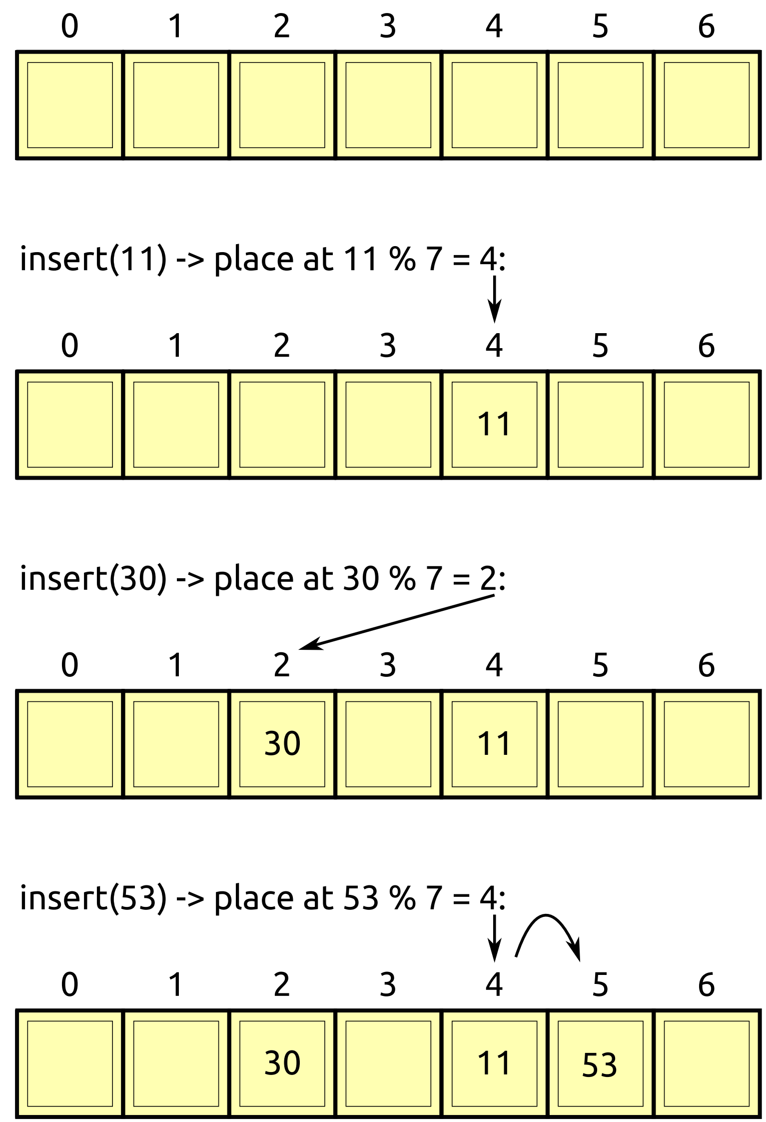
# Collision Resolution

## Open Addressing: Linear Probing

### Description

- If can't place key at bucket, try to place at next bucket (with wraparound). If that doesn't work, try next bucket. And so on...

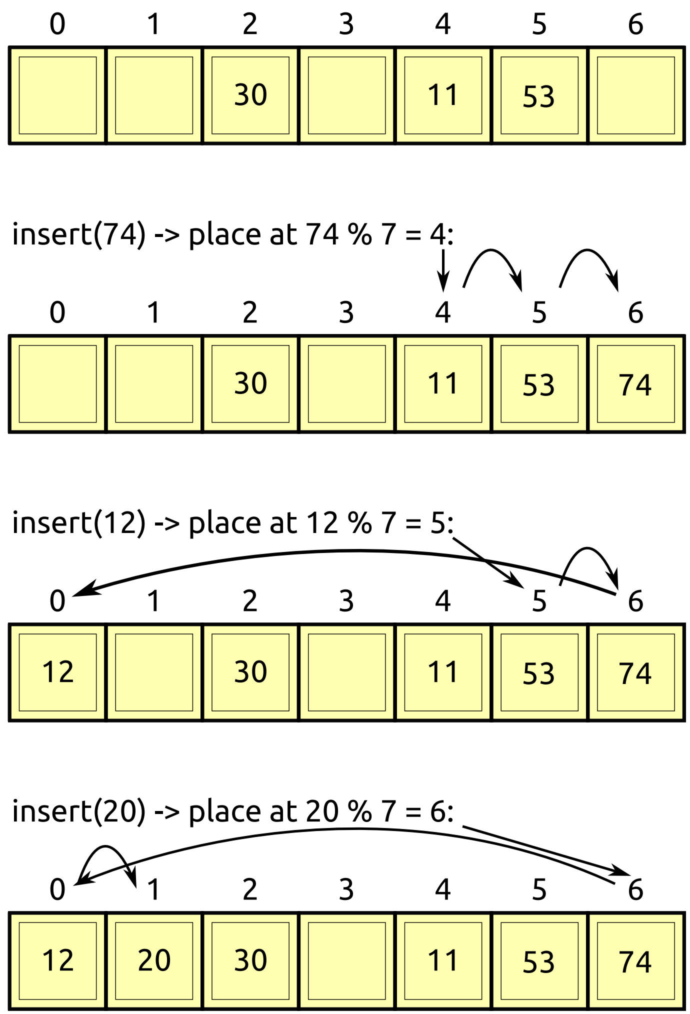
### Example



# Collision Resolution

## Open Addressing: Linear Probing

### Example (Continued)

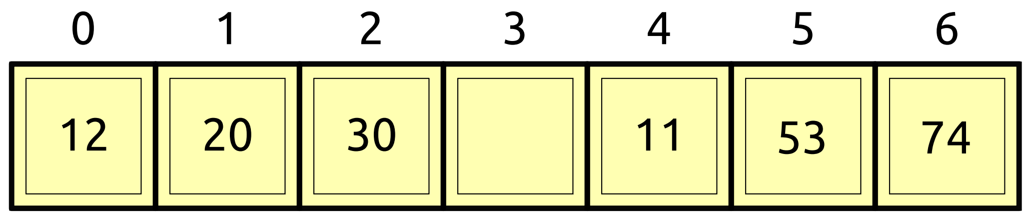



- By the way, $\lambda = \frac{6}{7}$ (bad) at end.

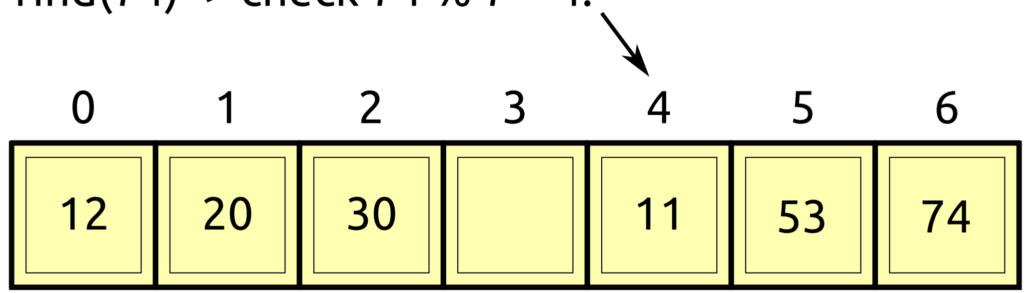
# Collision Resolution

## Open Addressing: Linear Probing

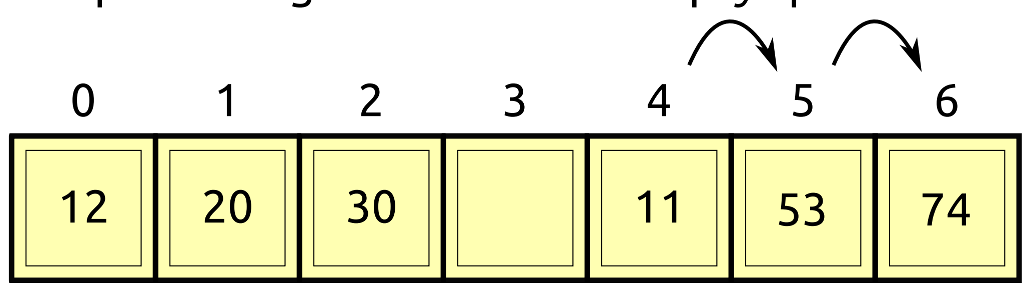
### Example of Find Operation




find(74) -> check 74 % 7 = 4:



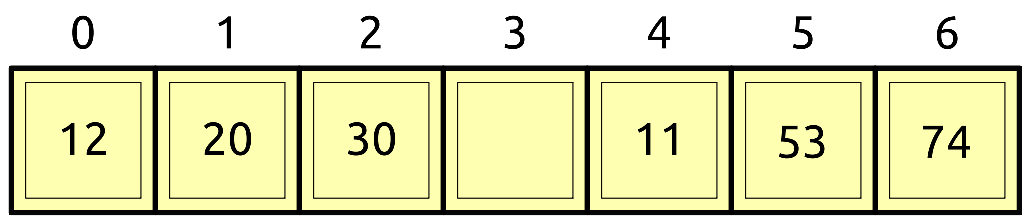

Keep checking until found or empty spot



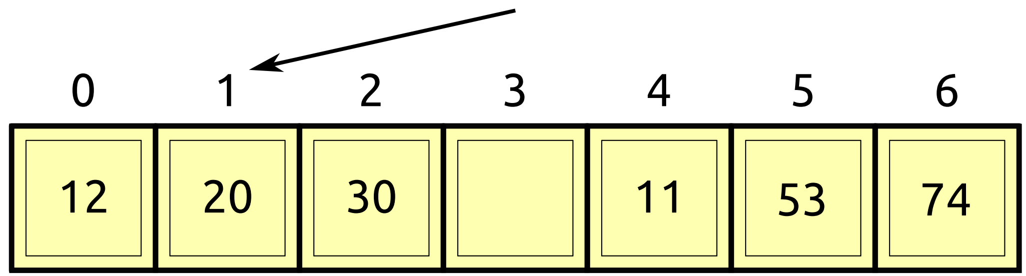
# Collision Resolution

## Open Addressing: Linear Probing

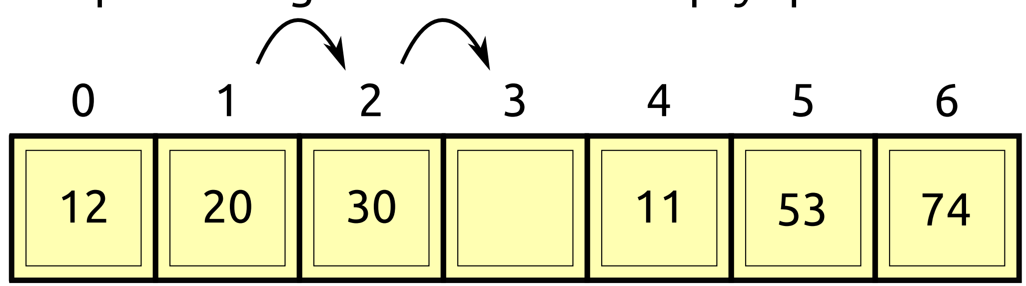
### Example of Failed Find Operation




find(8) -> check 8 % 7 = 1:




Keep checking until found or empty spot



# Collision Resolution

## Open Addressing: Linear Probing

### Deletion: Bad Way

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 12 | 20 | 30 | | 11 | 53 | 74 |

delete(11) -> check 11 % 7 = 4:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 12 | 20 | 30 | | 11 | 53 | 74 |

find(53) -> check 53 % 7 = 4

Not found!

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 12 | 20 | 30 | | | 53 | 74 |

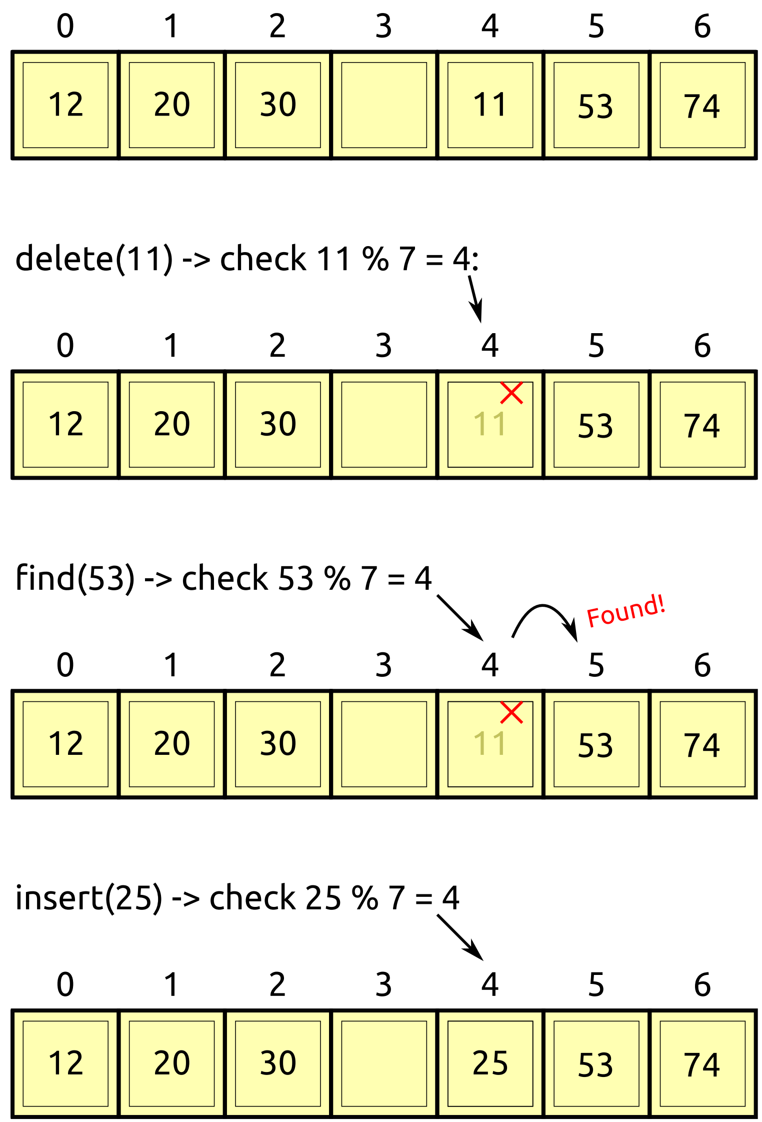
- Falsely reports not found.

# Collision Resolution

## Open Addressing: Linear Probing

### Deletion: Good Way (Lazy Deletion)

- Mark that 11 is deleted.
  - Needn't actually remove it. (Save time.)
- Lazily deleted node can be replaced.



# Collision Resolution

## Open Addressing: Linear Probing

### Weaknesses

- **primary clustering**: Blocks of nearby occupied buckets tend to form.
- New key may take several collision resolution attempts (and then adds to the cluster).

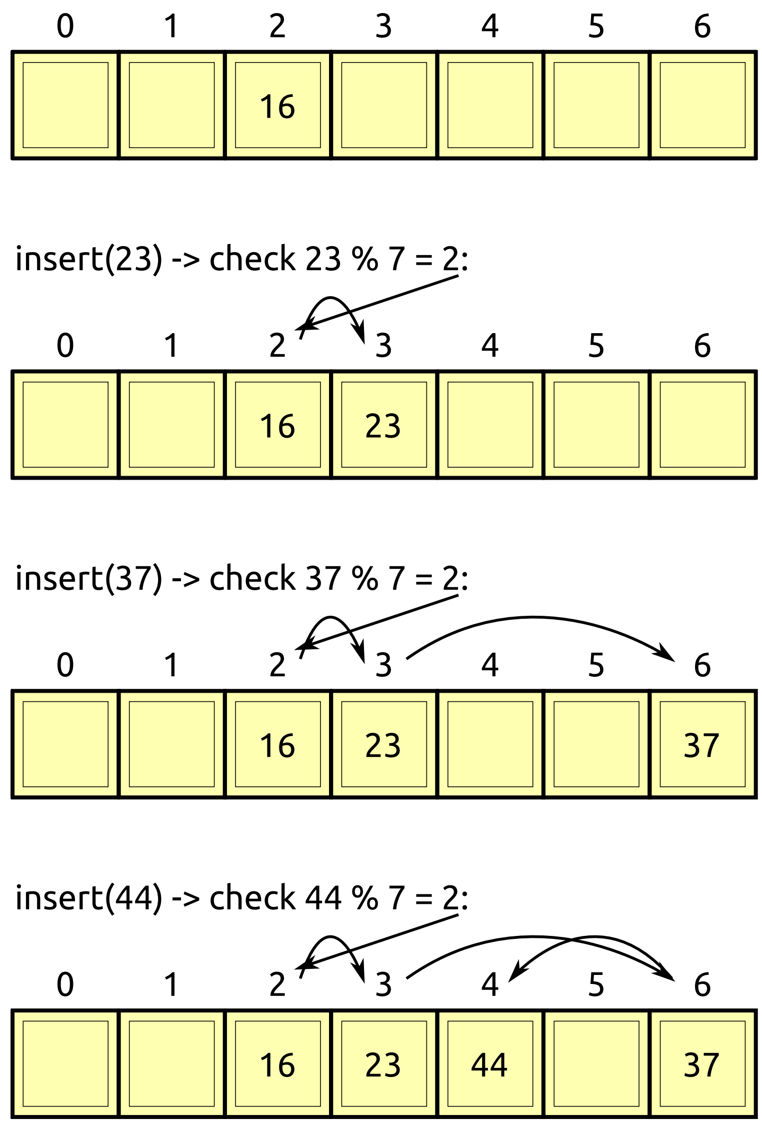
# Collision Resolution

## Open Addressing: Quadratic Probing

### Description

- Eliminates primary clustering issue.
- If can't place key at bucket $u$, try to place at bucket $1^2 = 1$ after that one. If that doesn't work, try to place at bucket $2^2 = 4$ after $u$. If that doesn't work, try to place at bucket $3^2 = 9$ after $u$. And so on... (wraparound when appropriate).
- Alternative way of thinking: check next bucket, then check $3$ buckets later, then check $5$ buckets later, then check $7$ buckets later, etc.

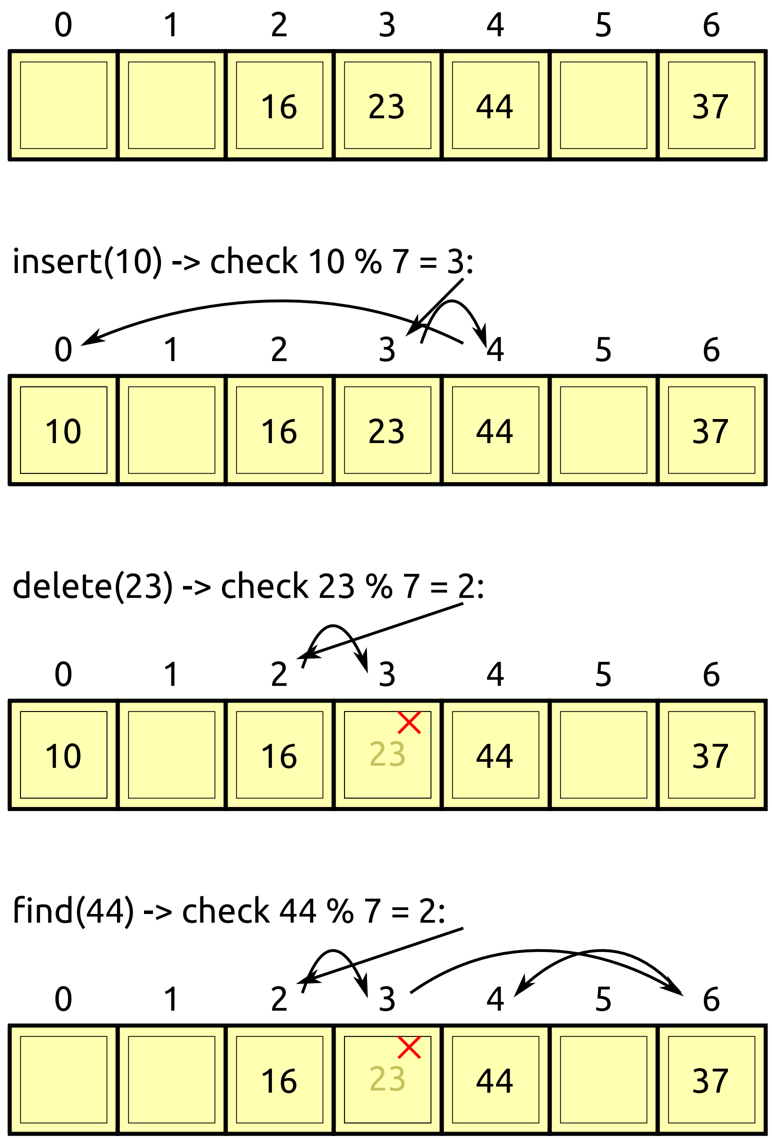
### Example



# Collision Resolution

## Open Addressing: Quadratic Probing

### Example (Continued)



# Collision Resolution

## Open Addressing: Quadratic Probing

### Analysis

- For linear probing, high $\lambda$ degraded performance.
- For quadratic probing, $\lambda > \frac{1}{2}$ can make it ***impossible*** to find empty bucket.
  - If table size not *prime*, can happen *even with* $\lambda \leq \frac{1}{2}$.
    - Example: If insert $8$ into below table, will check indices $0$, $1$, and $4$ forever.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 16 | 25 | | | 84 | | | |

- Can *prove* that if table is half empty and table size is prime, guaranteed to find empty bucket[1].
- Vulnerable to secondary clustering[2]: elements hashed to same location will probe same buckets.

1. See p.204 of *Data Structures and Algorithm Analysis in C++* by Mark Allen Weiss (Fourth Edition) for proof.
2. Weiss' book says this isn't a big concern.

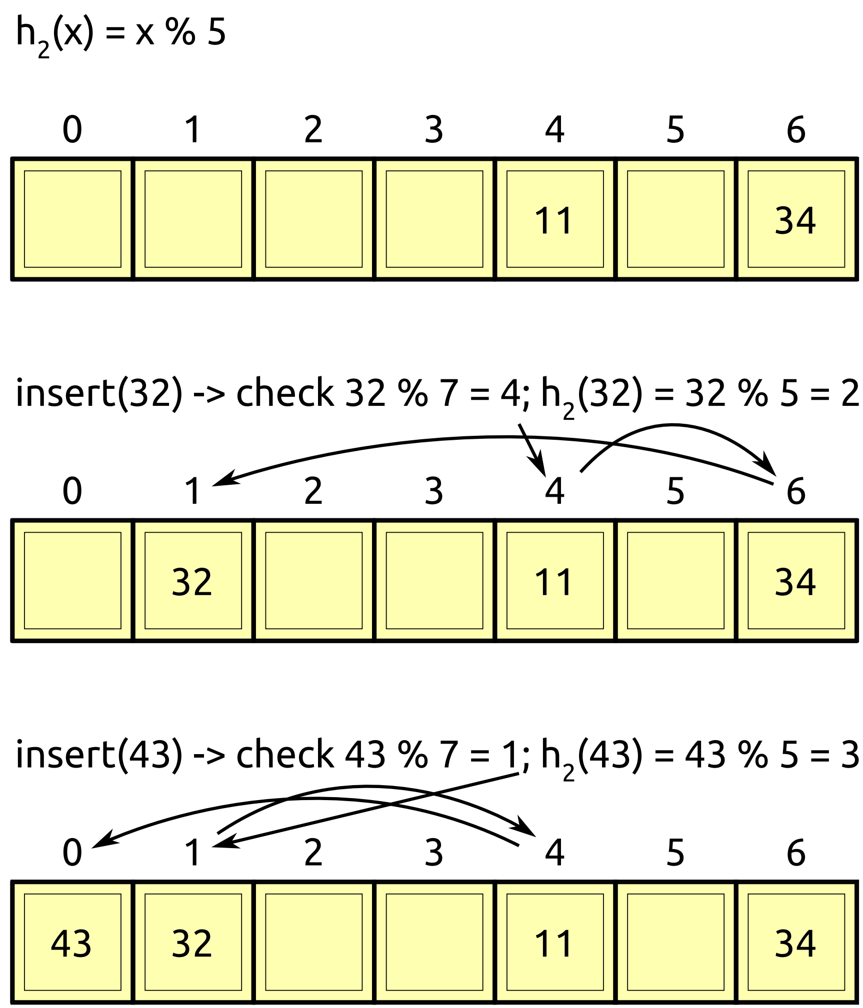
# Collision Resolution

## Open Addressing: Double Hashing

### Description

- Can eliminate secondary clustering issue.
- Requires second hash function $h_2(x)$.
- If can't place key $k$ at bucket, try to place $h_2(k)$ spots later. If doesn't work there, try $h_2(k)$ spots later. And so on... (wraparound when appropriate).
- Slower than quadratic probing in practice, because second hash function.

### Example



# Rehashing

## Description

- If $\lambda$ too high, can **rehash**.
- Steps: (let $m$ be old table size, $m'$ new table size)
    1. Create new table of size $m' = nextPrime(2m)$, where $nextPrime(x)$ returns lowest prime number above $x$.
    2. Insert each element in old table into new table, using new hash function, $h_1(k) = k\%m'$.

## Example

- Rehash when $\lambda \geq \frac{1}{2}$.

(Using linear probing.)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | 51 | 10 | 18 | | |

insert(27) -> We'll rehash first.
insert(27) -> 27 % 17 = 10
insert(51) -> 51 % 17 = 0
insert(10) -> 10 % 17 = 10
insert(18) -> 18 % 17 = 1

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 51 | 18 | | | | | | | | | 27 | 10 | | | | | |

# Usage

## In Python

- Set and dictionary.
- Why `in` operator takes constant time.

## Worst-Case Time Complexity

- Find/insert/delete:
  - $\Theta(n)$.
    - Rehashing.
  - Amortized $\Theta(1)$.
    - Rehashing occurs infrequently.
    - **To be clear**: a hash table is the first data structure you should consider when a normal Python list doesn't suffice.
- Can use number of buckets $m$ for rehash time; analysis is similar.

# Usage

## vs. Other Data Structures

| | **Find** | **Insert** | **Delete** |
|---|---|---|---|
| Unordered linked list | $\Theta(n)$ | $\Theta(1)$ | $\Theta(n)$ |
| Ordered/sorted linked list | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| Unordered Python list | $\Theta(n)$ | $\Theta(1)$ (amortized) | $\Theta(n)$ |
| Ordered/sorted Python list | See Conceptual HW 2. | $\Theta(n)$ | $\Theta(n)$ |
| Hash Table | $\Theta(1)$ (amortized) | $\Theta(1)$ (amortized) | $\Theta(1)$ (amortized) |
| BST | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| AVL Tree | $\Theta(\lg n)$ | $\Theta(\lg n)$ | $\Theta(\lg n)$ |
| Splay Tree | $\Theta(\lg n)$ (amortized) | $\Theta(\lg n)$ (amortized) | $\Theta(\lg n)$ (amortized) |

# Usage

## vs. Self-Balancing BST

| | **Find** | **Insert** | **Delete** |
|---|---|---|---|
| Hash Table | $\Theta(1)$ (amortized) | $\Theta(1)$ (amortized) | $\Theta(1)$ (amortized) |
| AVL Tree | $\Theta(\lg n)$ | $\Theta(\lg n)$ | $\Theta(\lg n)$ |
| Splay Tree | $\Theta(\lg n)$ (amortized) | $\Theta(\lg n)$ (amortized) | $\Theta(\lg n)$ (amortized) |

- Why bother using a BST?
- Hash tables are bad at any operation involving the ordering of the keys.
    - Examples:
        - Find min.
        - Find max.
        - Find all keys within a certain range.
        - Print keys in sorted order.

# Hashing Strings

- Harder to choose a hash function for strings.

## Approach #1: ASCII Values[1]

- Each character has an ASCII/integer value[2].

| Dec | Hex | Name | Char | Ctrl-char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | Null | NUL | CTRL-@ | 32 | 20 | Space | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | Start of heading | SOH | CTRL-A | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | Start of text | STX | CTRL-B | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | End of text | ETX | CTRL-C | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | End of xmit | EOT | CTRL-D | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | Enquiry | ENQ | CTRL-E | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | Acknowledge | ACK | CTRL-F | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | Bell | BEL | CTRL-G | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | Backspace | BS | CTRL-H | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | Horizontal tab | HT | CTRL-I | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | Line feed | LF | CTRL-J | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | Vertical tab | VT | CTRL-K | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | Form feed | FF | CTRL-L | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | Carriage feed | CR | CTRL-M | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | Shift out | SO | CTRL-N | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | Shift in | SI | CTRL-O | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | Data line escape | DLE | CTRL-P | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | Device control 1 | DC1 | CTRL-Q | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | Device control 2 | DC2 | CTRL-R | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | Device control 3 | DC3 | CTRL-S | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | Device control 4 | DC4 | CTRL-T | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | Neg acknowledge | NAK | CTRL-U | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | Synchronous idle | SYN | CTRL-V | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | End of xmit block | ETB | CTRL-W | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | Cancel | CAN | CTRL-X | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | End of medium | EM | CTRL-Y | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | Substitute | SUB | CTRL-Z | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | Escape | ESC | CTRL-[ | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | File separator | FS | CTRL-\ | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | Group separator | GS | CTRL-] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | Record separator | RS | CTRL-^ | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | Unit separator | US | CTRL-_ | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | DEL |

1. The two discussed approaches are directly from Weiss' book, but the code is translated from C++ to Python.
2. In C programming languages, characters (and booleans) *are* integers!

# Hashing Strings

## Approach #1: ASCII Values

- In Python, can translate between character and ASCII value.

```
>>> ord("A")
65
>>> ord("a")
97
>>> ord("y")
121
>>> chr(121)
'y'
>>> chr(97)
'a'
>>> ord('5')
53
```

# Hashing Strings

## Approach #1: ASCII Values

- Possible hash function using ASCII values:

```
def hash(s, table_size):
    hash_val = 0
    for c in s:
        hash_val += ord(c)
    return hash_val % table_size
```

```
>>> hash("abc", 17)
5
```

- Bad if `table_size` too big.

# Hashing Strings

## Approach #2

- Only consider first three characters.
    - 27 represents number of characters in English alphabet plus blank.

```
def hash(s, table_size):
    return (ord(s[0]) + 27 * ord(s[1]) + 729 * ord(s[2])) % table_size
```

## Examples

- `hash("abc", 17)` $\Rightarrow 97 + 27 \cdot 98 + 729 \cdot 99 = 97 + 2646 + 72171 = 74914$
  $\Rightarrow 74914 \,\%\, 17 = 12.$

```
>>> hash("abc", 17)
12
>>> hash("abc", 10000)
4914
```

- $26^3 = 17576$ combinations seems good.
- If check dictionary, find that only $2851$ of the combinations are used.

# References / Further Reading


- Chapter 5 of *Data Structures and Algorithm Analysis in C++* by Mark Allen Weiss (Fourth Edition).
    - I removed *extendible hashing* from the course content because it's hard to appreciate its purpose without learning a lower level language like C. You can read about it in section 5.9.
- Section 6.5 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.