# Rehashing,
# Cuckoo Hashing,
# Cichelli’s Algorithm

# Rehashing

- What if you run out of space in the hash table, or the table is so full that finding an empty slot is very hard?
- Rehashing: create a new larger hash table, and copy the elements over
  - Same or different hash function with the new T can be used
  - Double the size, or, even better, choose new T = large prime closest to old T * 2

Original Hash Table

| | |
|---|---|
| 0 | 6 |
| 1 | 15 |
| 2 | |
| 3 | 24 |
| 4 | |
| 5 | |
| 6 | 13 |

After Inserting 23

| | |
|---|---|
| 0 | 6 |
| 1 | 15 |
| 2 | 23 |
| 3 | 24 |
| 4 | |
| 5 | |
| 6 | 13 |

h(K) = K % 7

After Rehashing

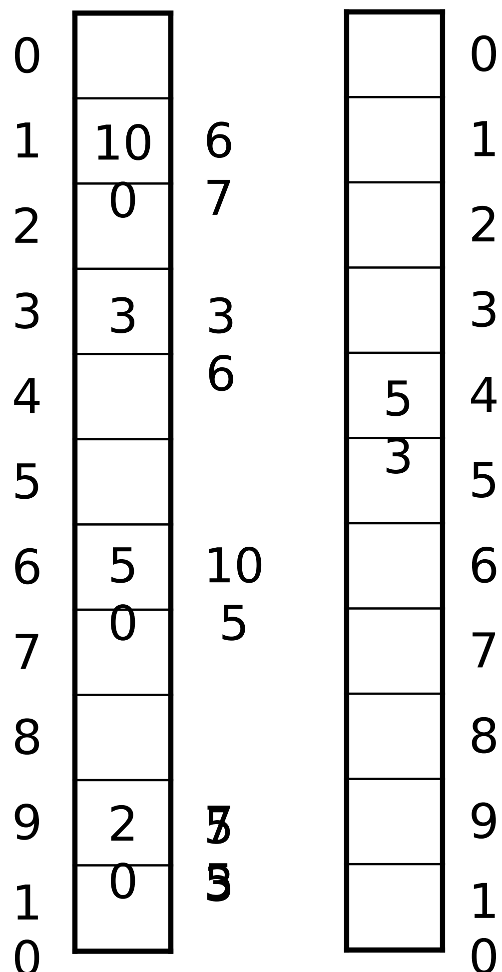
| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | 6 |
| 7 | 23 |
| 8 | 24 |
| 9 | |
| 10 | |
| 11 | |
| 12 | |
| 13 | 13 |
| 14 | |
| 15 | 15 |
| 16 | |

h(K) = K % 17

# Cuckoo hashing

- Collision resolution algorithm invented in 2001
- Guaranteed O(1) look-up time for all entries
- Two hash tables are used, each associated with its own hash function
- Each K can be either in one or in other table

| k | h(k) | h'(k) |
|---|---|---|
| **20** | 9 | 1 |
| **50** | 6 | 4 |
| **53** | 9 | 4 |
| **75** | 9 | 6 |
| **100** | 1 | 9 |
| **67** | 1 | 6 |
| **105** | 6 | 9 |
| **3** | 3 | 0 |
| **36** | 3 | 3 |




$h(K) = K \% 11$

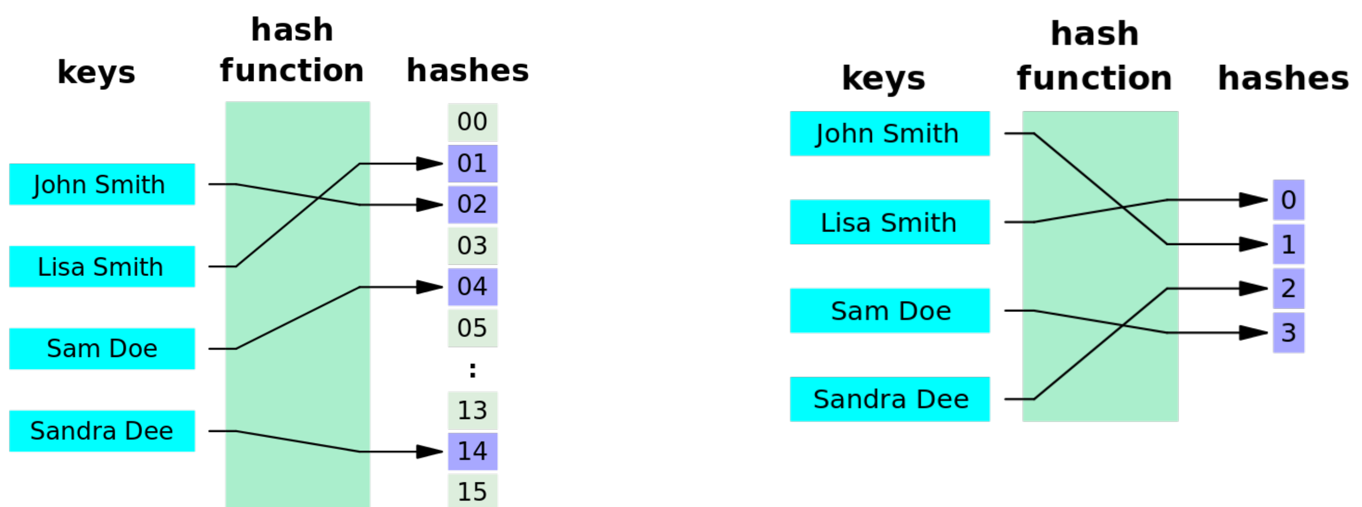
$h'(K) = (K / 11) \% 11$

Problem: cuckoo hashing may produce an infinite loop

Add max iteration number to the algorithm; in case of deadlock, **rehash** both tables

# Perfect hash functions

- A perfect hash function is a hash function that assigns a unique hash code to every key (no collisions)
- A minimal perfect hash function is a perfect hash function that uses every slot in the hash table and wastes no space




- Why are hash functions not perfect/minimal?
  - Because the input space is unlimited – can't prepare for everything!
  - What if we knew exactly what data and how much data needs to be stored?
  - Turns out you can design a perfect hash function if data (input space) is known beforehand!

# Cichelli’s Algorithm

- An algorithm that constructs a perfect minimal hash function for a set of strings
- Suppose we want to store different cities in a hash table:

Johannesburg
Pretoria
Cape Town
Durban

  - We are re-inventing google maps: need to store a lot of information about each city, and be able to access it quick
  - Every city has a unique name: city names can be the keys
  - The number of cities is predetermined: reserve just enough space (and no extra!) to store them all
- Cichelli’s hash function:

```
h(S) = [length(S) + firstLetter(S) + lastLetter(S)] % T
```

  - length(S) is the length of the string
  - firstLetter(S) and lastLetter(S) return the values assigned to the first and last letter of the string
  - Task: Assign values to all first/last characters to create a perfect hash

# Cichelli’s Algorithm

1. Count the frequencies of first/last characters

| | |
|---|---|
| Pretoria | P = 1, A = 1 |
| Johannesburg | J = 1, G = 1 |
| Durban | D = 1, N = 1 |
| Cape Town | C = 1, N = 2 |

P = 1
A = 1
J = 1
G = 1
D = 1
N = 2
C = 1

2. Sort the strings based on cumulative frequency (freq(first) + freq(last))

| | |
|---|---|
| Durban | 3 |
| Cape Town | 3 |
| Pretoria | 2 |
| Johannesburg | 2 |

Why sort:
Same letters may cause clashes, thus place the words that share letters first

# Cichelli's Algorithm

3. assignHash(wordlist)
   I. Remove first word on the list
   II. Choose values for first(w), last(w)
   III. Compute hash(w) = [length(w) + first(w) + last(w)]%T
      - if(hash is not taken)
         - use the hash
         - assignHash(wordlist) // recursive step
      - else
         - Go to (II) and try other values for first(w), last(w)

| | |
|---|---|
| Durban | 3 |
| Cape Town | 3 |
| Pretoria | 2 |
| Johannesburg | 2 |

D = 0
N = 0

| | |
|---|---|
| 0 | |
| **1** | |
| **2** | **Durban** |
| **3** | |

`h(Durban) = [6 + 0 + 0] % 4 = 2`

# Cichelli's Algorithm

3. assignHash(wordlist)
   I. Remove first word on the list
   II. Choose values for first(w), last(w)
   III. Compute hash(w) = [length(w) + first(w) + last(w)]%T
      - if(hash is not taken)
         - use the hash
         - assignHash(wordlist) // recursive step
      - else
         - Go to (II) and try other values for first(w), last(w)

| | |
|---|---|
| Cape Town | 3 |
| Pretoria | 2 |
| Johannesburg | 2 |

D = 0
N = 0
C = 0

| | |
|---|---|
| 0 | |
| **1** | **Cape Town** |
| **2** | **Durban** |
| **3** | |

```
h(Cape Town) = [9 + 0 + 0] % 4 = 1
```

# Cichelli’s Algorithm

3. assignHash(wordlist)
   I. Remove first word on the list
   II. Choose values for first(w), last(w)
   III. Compute hash(w) = [length(w) + first(w) + last(w)]%T
      - if(hash is not taken)
         - use the hash
         - assignHash(wordlist) // recursive step
      - else
         - Go to (II) and try other values for first(w), last(w)

| | |
|---|---|
| Pretoria | 2 |
| Johannesburg | 2 |

D = 0
N = 0
C = 0
P = 0
A = 0

| | |
|---|---|
| 0 | **Pretoria** |
| **1** | **Cape Town** |
| **2** | **Durban** |
| **3** | |

```
h(Pretoria) = [8 + 0 + 0] % 4 = 0
```

# Cichelli's Algorithm

Johannesburg | 2

D = 0
N = 0
C = 0
P = 0
A = 0
J = 0 | J = 1
G = 0 | G = 1 | G = 2 | ?

MAX = 2

| | |
|---|---|
| 0 | **Pretoria** |
| **1** | **Cape Town** |
| **2** | **Durban** |
| **3** | **Johannesburg** |

Final:

D = 0
N = 0
C = 0
P = 0
A = 0
J = 1
G = 2

```
h(Johannesburg) = [12 + 0 + 0] % 4 = 0
h(Johannesburg) = [12 + 0 + 1] % 4 = 1
h(Johannesburg) = [12 + 0 + 2] % 4 = 2

h(Johannesburg) = [12 + 1 + 0] % 4 = 1
h(Johannesburg) = [12 + 1 + 1] % 4 = 2
h(Johannesburg) = [12 + 1 + 2] % 4 = 3
```

# Cichelli's Algorithm

If values have been assigned to letters, they should be re-used by all words

3. assignHash(wordlist)
   - Remove first word on the list
   - for(first(w) = 0; first(w) < MAX; first(w)++) // unless first(w) has a value
     - for(last(w) = 0; last(w) < MAX; last(w)++) // unless last(w) has a value
       - Compute hash(w) = [length(w) + first(w) + last(w)]%T
       - if(hash(w) not taken)
         - Put the word in position hash(w)
         - success = assignHash(wordlist) // recursive step
         - if(success) return success;
   - Put word back on the list
   - Return failure // backtracking

MAX is chosen to be a small value;
# words / 2

# Cichelli’s Algorithm

Frequencies:

| | |
|---|---|
| Anna | 4 + 4 = 8 |
| Maria | 2 + 4 = 6 |
| Jane | 3 + 1 = 4 |
| John | 3 + 2 = 5 |
| Julia | 3 + 4 = 7 |
| Megan | 2 + 2 = 4 |

A = 4
M = 2
J = 3
E = 1
N = 2

| | |
|---|---|
| 0 | |
| **1** | |
| **2** | |
| **3** | |
| **4** | |
| **5** | |

| | |
|---|---|
| 8 | Anna |
| 7 | Julia |
| 6 | Maria |
| 5 | John |
| 4 | Jane |
| 4 | Megan |

Assign values to first/last characters!

# Cichelli's Algorithm

MAX = T / 2 = 6 / 2 = 3

Anna
Julia
Maria
John
Jane
Megan

Anna (4 + 0 + 0) % 6 = 4
Julia (5 + 0 + 0) % 6 = 5
Maria (5 + 0 + 0) % 6 = 5 X
Maria (5 + 1 + 0) % 6 = 0
John (4 + 0 + 0) % 6 = 4 X
John (4 + 0 + 1) % 6 = 5 X
John (4 + 0 + 2) % 6 = 0 X
John (4 + 0 + 3) % 6 = 1
Jane (4 + 0 + 0) % 6 = 4 X
Jane (4 + 0 + 1) % 6 = 5 X
Jane (4 + 0 + 2) % 6 = 0 X
Jane (4 + 0 + 3) % 6 = 1 X
Maria (5 + 2 + 0) % 6 = 1
// John will go to zero, but Jane won't find a place
Maria (5 + 3 + 0) % 6 = 2
// John will go to zero
// Jane will go to one

A = 0
J = 0
M = 3
N = 3
E = 3

| | |
|---|---|
| 0 | Maria |
| **1** | John |
| **2** | |
| **3** | |
| **4** | Anna |
| **5** | Julia |

Nothing else to try – backtrack!

# Cichelli's Algorithm

MAX = T / 2 = 6 / 2 = 3

Anna
Julia
Maria
John
Jane
Megan

Anna (4 + 0 + 0) % 6 = 4
Julia (5 + 0 + 0) % 6 = 5
Maria (5 + 3 + 0) % 6 = 2
John (4 + 0 + 2) % 6 = 0
Jane (4 + 0 + 3) % 6 = 1
Megan (5 + 3 + 2) % 6 = 4 X
John (4 + 0 + 3) % 6 = 1
Jane (4 + 0 + 2) % 6 = 0
Megan (5 + 3 + 3) % 6 = 5 X
Julia (5 + 1 + 0) % 6 = 0

| | |
|---|---|
| 0 | John / Jane |
| **1** | Jane |
| **2** | Maria |
| **3** | |
| **4** | Anna |
| **5** | Julia |

A = 0
J = 1
M = 3
N = 3
E = 3

Repeat the steps till the perfect hash is created

Complexity: exponential

What is the problem with this approach?

Infeasible for large sets

Backtracking can be very time-consuming

What if you had to store Jane and Jade?