# HASH TABLES

http://fscked.org/writings/225notes/week13/week13.html

http://en.wikipedia.org/wiki/Hash_table

# Running Times

| | Worst Case | | | Average Case | | |
|---|---|---|---|---|---|---|
| | Search | Insert | Delete | Search | Insert | Delete |
| Unordered array | | | | | | |
| Ordered array | | | | | | |
| Unordered list | | | | | | |
| Ordered list | | | | | | |

Can we do Better?

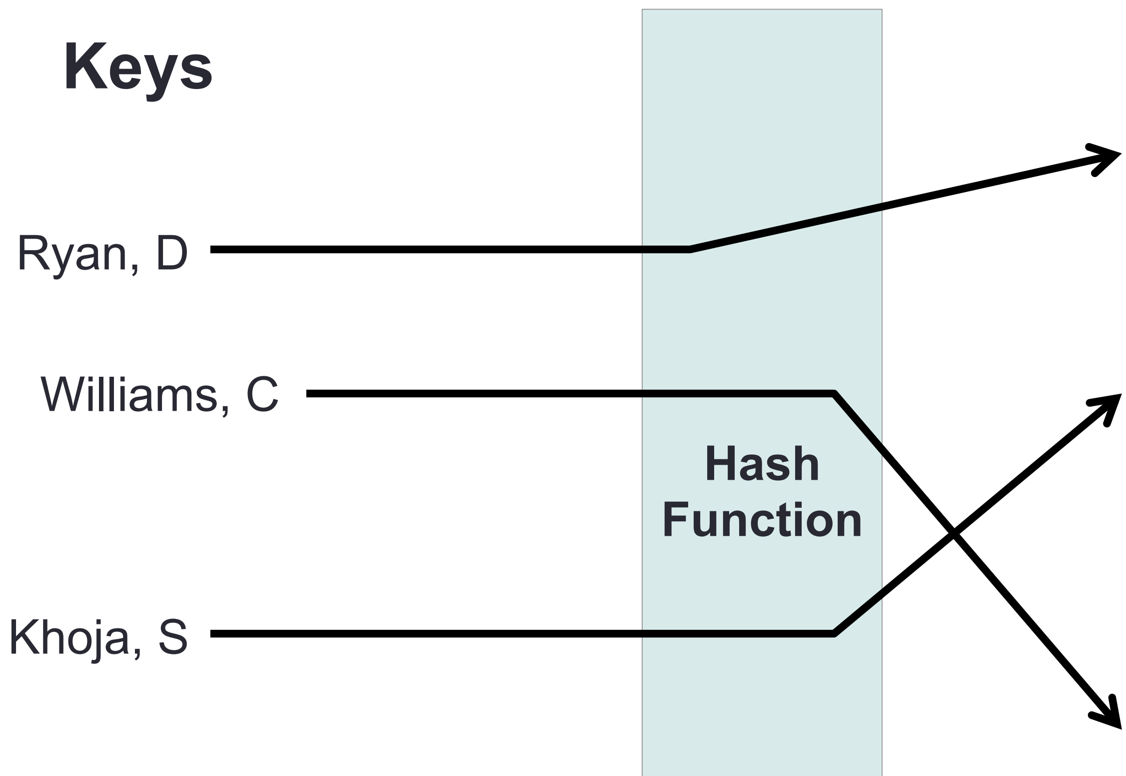
# Hash Table

- A hash table (or hash map) is a data structure that maps keys (identifiers) into a certain location (bucket)

- A hash function changes the key into an index value (or hash value)

# Hash Tables

The Hash Table has a fixed length. We'll see how to add space dynamically later.




| Bucket | Data |
|---|---|
| 01 | Ryan, D |
| 02 | |
| 03 | Khoja, S |
| 04 | |
| 05 | |
| 06 | Williams, C |
| 07 | |

hash(Ryan, D) = 01
hash(Williams, C) = 06
hash(Khoja, S) = 03

# A Problem

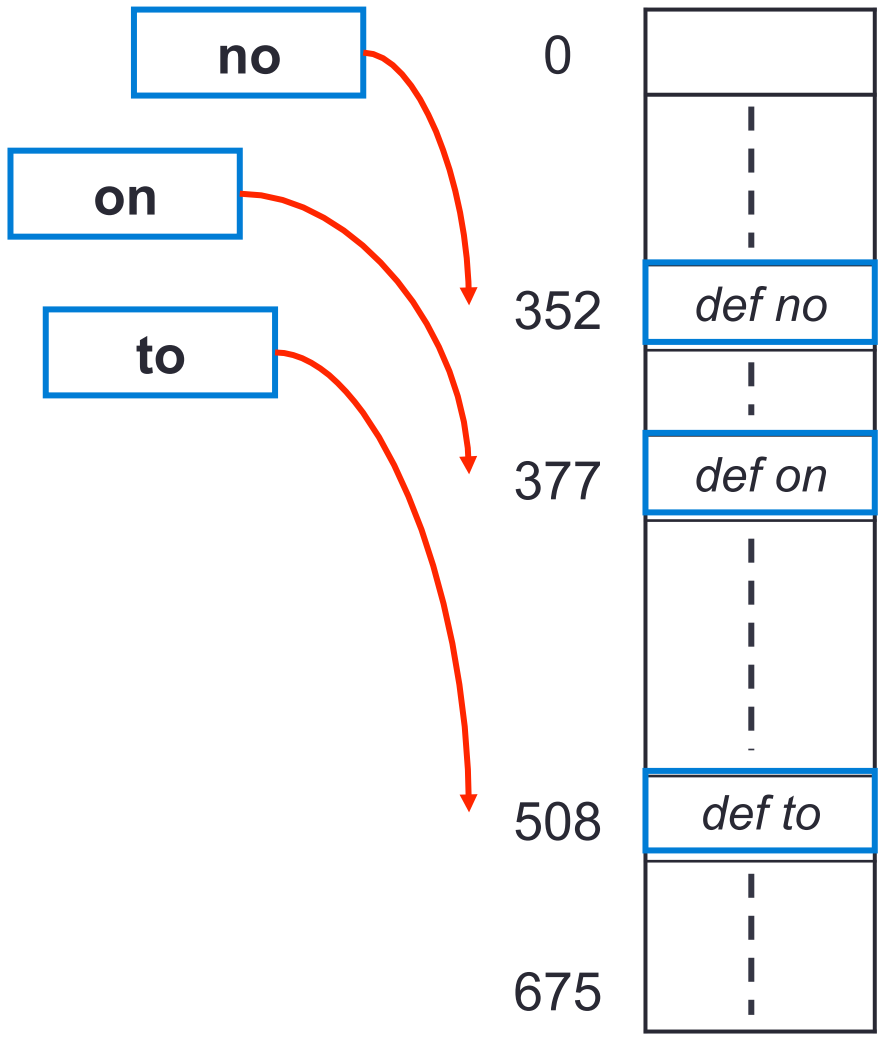
- We want to create a fast dictionary that we can use to look up the definition of 2-letter words (e.g. on, to, so, no).
- The number of possible words is 26 * 26 = 676.
- Create an array with 676 entries.
- Write a function to map each 2-letter word to a unique integer number within the array range [0, 675].
- Use the integer as an index to the array, and place the definition of the word into that array position.

# A Solution

- Function:

$26 * ( c_1 - \text{'a'} ) + ( c_2 - \text{'a'} )$

- Result for “no”:
  $26 * ( \text{'n'} - \text{'a'} ) + ( \text{'o'} - \text{'a'} ) =$
  $26 * ( 14 - 1 ) + ( 15 - 1 ) = 352$
- Result for “on” = 377
- Result for “to” = 508



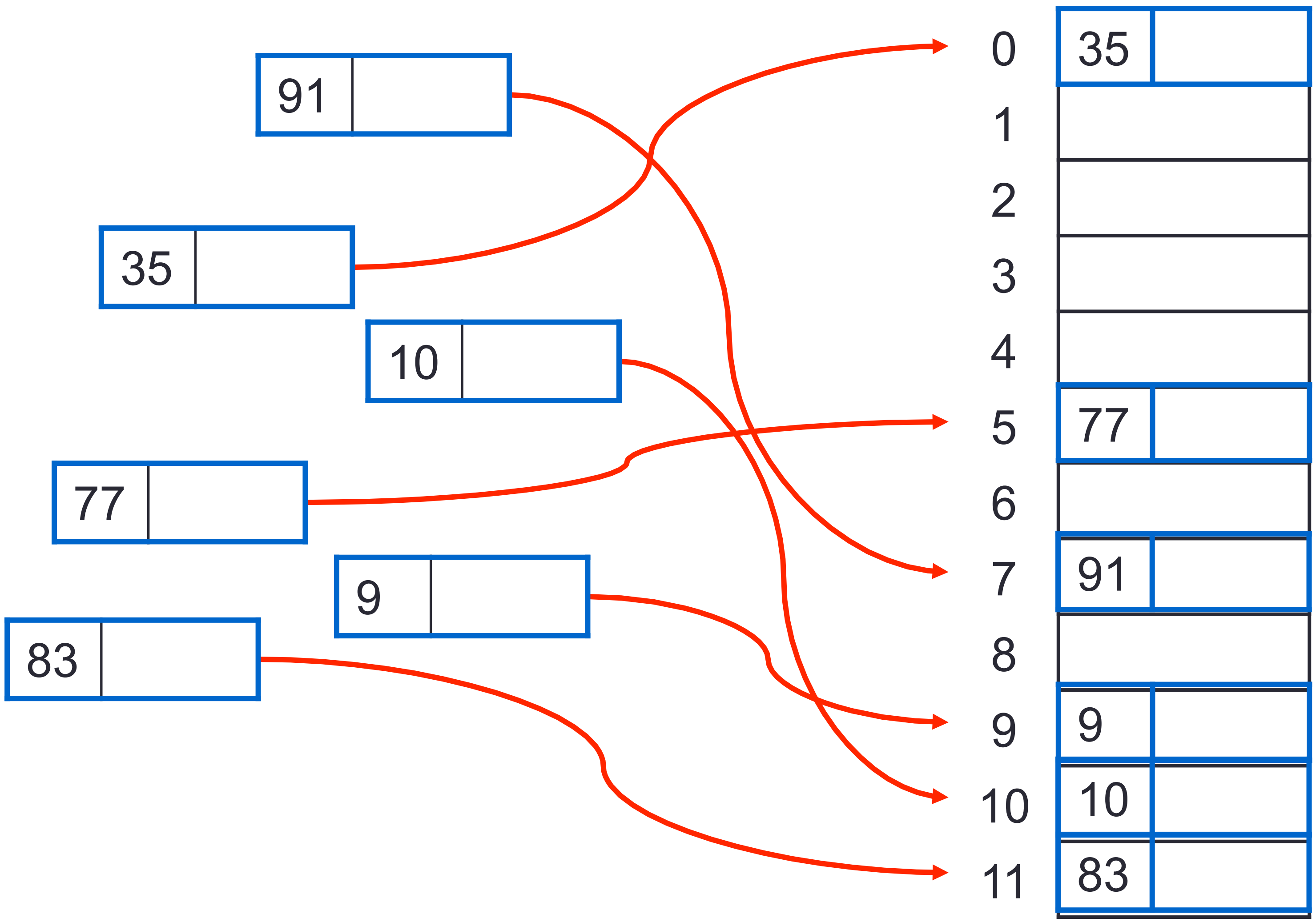
# Another Problem

- Q. How big would the hash table be if we wanted to store every word in the English language? Keep in mind that the longest word in the dictionary has 45 letters.
- The longest word in the dictionary is *pneumonoultramicroscopicsilicovolcanoconioisis* and it's a type of lung disease.
- The size of the table needed to store all English words of 45 letters or less is $26^{45}$.
- Of course it is impossible to create and store such a large table, and it would also be wasteful since fewer than a million of those words would be valid English words.
- What is the solution? Use a different hash function!

# The Solution

- Create an array large enough to hold all the words in the English language.
- Map the words to the array indexes using a hash function.
- H ( key ) = key *mod* table-size
- A hash function should:
  - Be easy to implement and quick to compute.
  - Achieve an even distribution over the hash table.
- The table-size is normally chosen to be a prime number to avoid uneven distribution

# A Simple Example



# Hash Functions

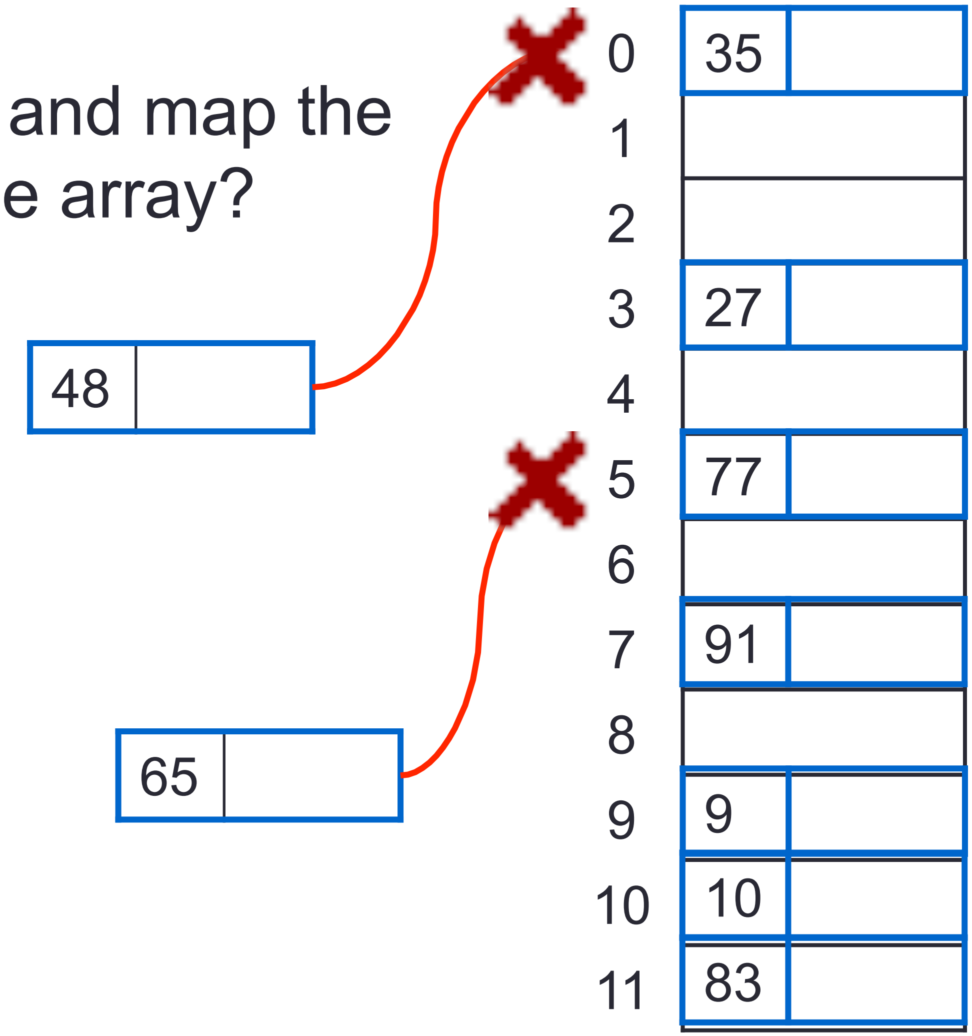
- Hash function:
  - key % table-size = array index

- 10 % 12 = 10
  91 % 12 = 7
  83 % 12 = 11
  36 % 12 = 0
  77 % 12 = 5
  9 % 12 = 9

| | | |
|---|---|---|
| 0 | 35 | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | 77 | |
| 6 | | |
| 7 | 91 | |
| 8 | | |
| 9 | 9 | |
| 10 | 10 | |
| 11 | 83 | |

# Collision

- What happens when we try and map the following keys onto the same array?




- This is called a collision!

# Avoiding Collision

- We could try to avoid this collision by using a different hash function.

- Choosing a good hash functions is hard, so instead, we concentrate on how to resolve collisions.

# Collisions

- Perfect Hash - each key maps to an empty bucket
  - Rare!
- Collisions occur where two different keys map to the same bucket
- Solution?

# Hash Function

- Hash function – compute the key's bucket address from the key
  - some function h(K) maps the domain of keys K into a range of addresses 0, 1, 2, … M-1

- The Problem
  - Finding a suitable function h
  - Determining a suitable M
  - Handling collisions

# Hash Function

- Mid Square
  - (turn the key into an integer)
  - square the key
  - take some number of bits from the center to form the bucket address

**Advantages?**

**Disadvantages?**

# Example

- Problem: Let's assume that the key value is simply the sum of the ASCII values squared. If the key value is 16-bits and we take the middle 8-bits:
  - How big is the hash table?
  - What is the range of bucket addresses?
  - Where does the key AB map to in the hash table?

# Implementation

section 2.9 of K&R

- How do we access the middle 8 in an integer?

```
// assume 4 byte integers
unsigned int key = 0x1231a456;
unsigned int middle;

middle = (key & 0x000ff000) >> 12;

printf("%08x %08x\n", key,
middle);
```

One Hex-digit is 4 bits

pad with zero    8 wide    hexadecimal output

http://www.eskimo.com/~scs/cclass/int/sx4ab.html

# Hash Function

- Division Hashing
  - bucket = key % N
  - N is the length of the hash table AND a prime number

a) How big is the hash table?

b) What is the range of bucket addresses?

c) Where does the key AB map to in the hash table?

**Advantages?**

**Disadvantages?**

# Collision Handling Techniques

- Open Addressing: if a collision occurs, then an empty slot is used to store the record. Techniques for selecting the empty slot include:
  - Linear Probing.
  - Quadratic Probing.
  - Re-hashing (double hashing).

- Separate Chaining: if a collision occurs, then this new record is attached to the existing record in the form of a chain (linked list). This way, several records will share a slot.

# Collision Handling

- Open Addressing
  - If both K and C map to the same bucket we have a collision
    - K and C are distinct
    - K is inserted first
  - To resolve using OA, find another unoccupied space for C

    BUT: We must do this systematically so we can find C again easily!
- Analysis: (summation of the # of probes to locate each key in the table) / # of keys in the table

# Open Addressing

- Find another open bucket
- **`bucket = ( h(K) + f(i) ) % N`**
  - N is the length of the table
  - h(K) : original hash of key K
  - f(i) : i is the number of times you have hashed and failed to find an empty slot
  - First hash is:
    - **`bucket = ( h(K) + f(0) ) % N`**
    - **`f(0) = 0`**

# Linear Probing

- If a collision occurs, then the next empty slot that is found is used to store the data item.

- If position h(n) is occupied, then try
  - ( h(n) + 1 ) mod table-size,
  - ( h(n) + 2 ) mod table-size,
  - and so on until an empty slot is found.

- Problems with linear probing include:
  - Formation of bad clusters.
  - Since gaps are being filled, the collision problem is exacerbated.

# Linear Probing

- `f(i) = i`
- Example:
  `h(Kn) = n % 11`
- Insert
  M13
  G7
  Q17
  Y25
  R18
  Z26
  F6

| Bucket | Data |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |

Primary Clustering!

# Primary Clustering

- Primary Clustering - this implies that all keys that collide at address b will extend the cluster that contains b

# Quadratic Probing

- With quadratic probing we try to overcome the problem of clustering.

- So, when a collision occurs while trying to insert an item in the table, instead of looking at every cell until an empty one is found, a function is applied to find an empty cell.

- If h(n) is occupied, try
  - ( h(n) + $1^2$ ) mod table-size,
  - ( h(n) + $2^2$ ) mod table-size,
  - and so on until an empty cell is found.

- Quadratic probing works well if the size of the table is a prime number and the the table is less than half full. Quadratic probing may not get anywhere.

# Quadratic Probing

- `f(i) = i^2`
- Example:
  `h(Kn) = n % 11`

- Insert
  M13
  G7
  Q17
  Y25
  R18
  Z26
  F6

| Bucket | Data |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |

Secondary Clustering!

# Secondary Clustering

- Secondary Clustering - is when adjacent clusters join to form a composite cluster

# Re-hashing

- With rehashing, if a collision occurs while inserting a new data item into the hash table, then a second hash function is applied to the result of the first hash function to find an empty cell in the table.
- The re-hashing function can either be a new function or a re-application of the original one. As long as the functions are applied to a key in the same order, then a sought key can always be located.
- The second hash function has to be chosen with care:
  - The sequence should be able to visit all slots in the table.
  - The function must be different from the first to avoid clustering.
  - It must be very simple to compute.

# Double Hash

- `f(i) = h2(k) * i`
  - h2(k) is some second hash function
  - unique probe sequence for every key
  - `bucket = ( h(K) + h2(K) * i ) % N`
  - h2(k) should be relatively prime to N for all k
    - don't produce zero
  - Example
    - `h(k) = k % N`
      `h2(k) = 1 + (k % ( N - 1 ))`

# Rehash

- Reallocate the table larger and reinsert every element

# Chaining

- Chaining is also called the bucket approach.
- It differs from the collision avoidance already discussed in that instead of each cell holding a single data item, several data items can be stored in the form of a linked list, and only the header of the linked list is placed in the table.
- The disadvantages of chaining include implementing a separate data structure, and dynamically allocating memory.

# Chaining (Open Hashing)

- Each bucket is the head of a linked list
  - if you hash a key to a bucket, insert the data into the list
  - insert at front, back, or in sorted order.
    - why would this decision matter?

# Problem

- Hash the keys M13, G7, Q17, Y25, R18, Z26, and F6 using the hash formula h(Kn) = n mod 9 with the following collision handling technique: (a) linear probing, (b) chaining

- Compute the average number of probes to find an arbitrary key K for both methods.

- avg = (summation of the # of probes to locate each key in the table) / # of keys in the table