Northern Illinois University

Hash Tables & Hashing

Dr. Maoyuan Sun – smaoyuan@niu.edu

Hash Table

- The ideal hash table data structure is an array of some fixed size containing items
- A **key** is used for the lookup
- The size of the hash table is known as the *tablesize* and common convention has the table run from 0 to (*tablesize* - 1)
- Each key is mapped into some number in the range 0 to (*tablesize* - 1) which corresponds to a cell in the array
- The mapping is called a hash function and has the properties:
  - it is easy to compute
  - ensures that any two keys result in a different cell location

Hash Table Example

- John hashes to 3
- Phil hashes to 4
- Dave hashes to 6
- Mary hashes to 7

- Don't worry about ***hash function*** here, just understand that the value of name is converted to array index
- What is the size of this hash table? 10

Key	Data
0	
1	
2	
3	John
4	Phil
5	
6	Dave
7	Mary
8	
9	

Hash Function

- If our data is as simple as integers, then simply returning key mod table size is a reasonable strategy
- Why would we use the size of the hash table in mod?

Provides us with an easy method for generating index between **0** and **tablesize – 1**.

Hash Function

- If our data is as simple as integers, then simply returning key mod table size is a reasonable strategy
- Why would we use the size of the hash table in mod?

Provides us with an easy method for generating index between **0** and **tablesize – 1** but this simplistic approach can cause problems:

key 100	tablesize 10	location 0
key 20	tablesize 10	location 0
key 40	tablesize 10	location 0

Collision at location 0 and considered bad!!

Hash Function

- One solution is to make the hash table size a prime number
- If the input keys are random integers, the **function is easy to compute** and **distributes the keys evenly**
- We don’t always have integers; strings are often used as keys
- What could be a function that would take a string as input and return us an index into the hash table? A simple strategy is to add up the ASCII values of the characters in the string

Hash Function

- ASCII values to index:

```
int
hash(const string &key, int tablesize)
{
    int hashvalue = 0;
    for(char ch : key)
        hashvalue += ch;

    return hashvalue%tablesize;
}
```

- Simple to implement, easy to compute
- Does it give us a nice distribution of index values?

Hash Function

- ASCII values to index:

```
int
hash2(const string &key, int tablesize)
{
    return (key[0]+27*key[1]+729*key[2])%tablesize;
}
```

- Assume key has at least 3 characters, we use the fact that 26 characters in English alphabet plus a blank (27) and that 27^2 is 729
- With a table size of 10007, we could expect a better distribution
- Problem still?

Hash Function

- ASCII values to index:

```
int
hash3(const string &key, int tablesize)
{
    int hashvalue = 0;
    for(char ch : key)
        hashvalue = 37 * hashvalue + ch;

    return hashvalue%tablesize;
}
```

- Back to using all characters, simple to implement, easy to compute

Hash Function

- You can also do clever things; say your string is street address, what could you do?

  9700 S. Cass Avenue, Argonne IL 60439

- Key could be complete address, with the hash function using a couple characters from street address, couple characters from city name and maybe the zipcode
- Other approaches use only the odd characters

Take away – a good hash function is important!!

Collisions

- If/when an element is inserted and it hashes to an index that is already in use, it is known as a collision.

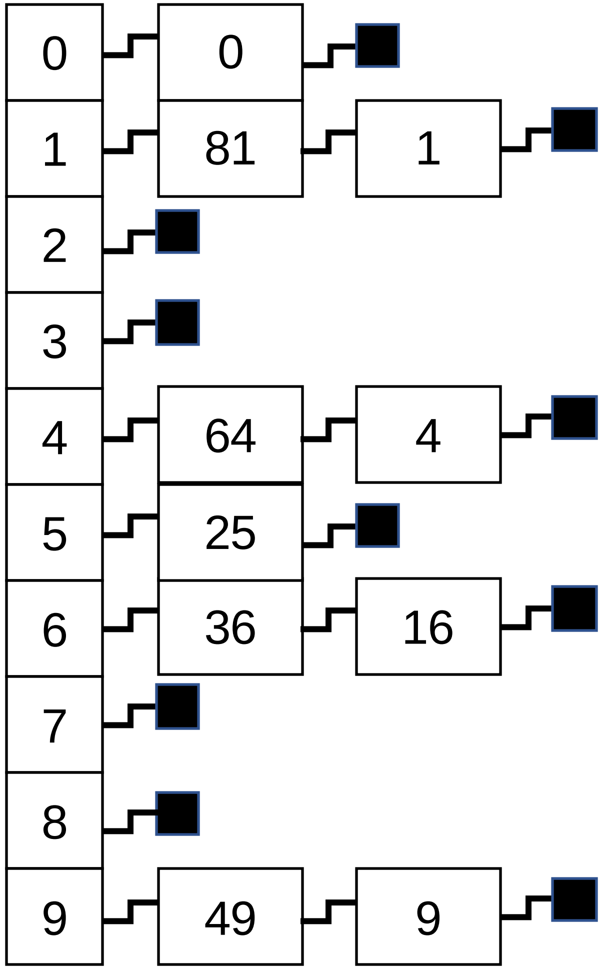
Collisions (Separate Chaining)

- Keeps a list of all the elements that hash to the same value

- Example:
  - Let's assume the keys are the first 10 perfect squares
  - Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81
  - Hashing Function: hash(x) = x%10, where 10 is tablesize [yes its not prime]
  - Then index looks like 0, 1, 4, 9, 6, 5, 6, 9, 4, 1

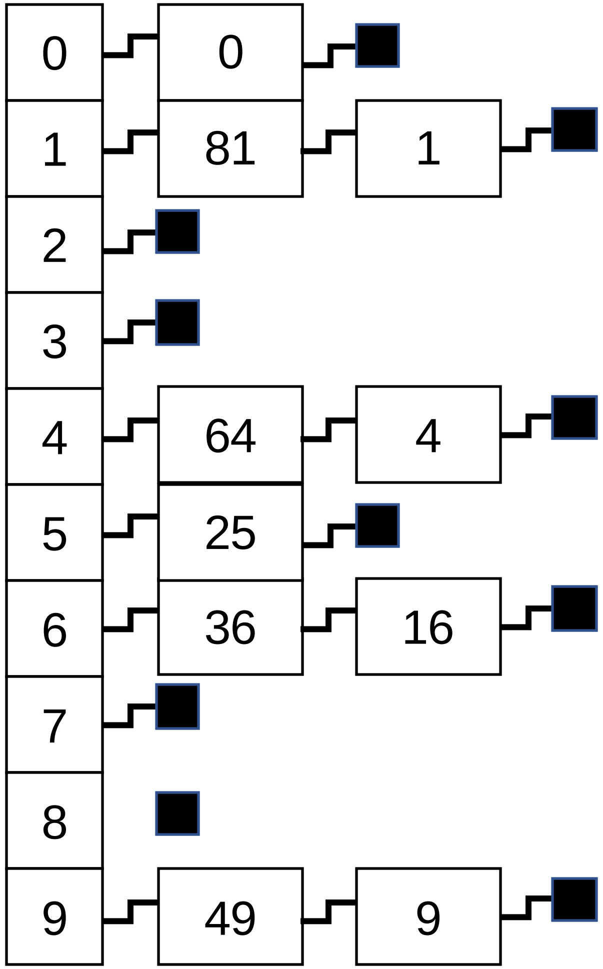
Collisions (Separate Chaining)

- Keeps a list of all the elements that hash to the same value
- Example:
  - Let’s assume the keys are the first 10 perfect squares
  - Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81
  - Hashing Function: hash(x) = x%10, where 10 is tablesize
  - Then index looks like 0, 1, 4, 9, 6, 5, 6, 9, 4, 1



Collisions (Separate Chaining)

- Insertion
  - If value at key (index) is NULL we create linked list and add element
  - If value at key (index) is !NULL we insert element into linked list
- Find
  - Traverse list till you find value or hit NULL
- Delete
  - Same as delete/remove from linked list



Collisions (Probing)

- In **probing**, you look for alternative cells until an empty cell is found
- Given cells:

$h_0(x)$, $h_1(x)$, $h_2(x)$, … are tried in succession where:

$$h_i(x) = (hash(x) + f(i))\%tablesize \text{ with } f(0) = 0$$

The function f is the *collision resolution strategy*

- When using probing all elements go into the table so the table is generally larger
- Types of probing
  - Linear
  - Quadratic
  - Double hashing

Collisions (Linear Probing)

- In linear probing, f is a linear function of i

$$f(i) = i$$

- Cells are tried sequentially (with wrap around) in search of empty cell
- Example:
  - Keys: 89, 18, 49, 58, 69
  - Hashing Function: hash(x) = x%10, where 10 is tablesize

Collisions (Linear Probing)

Keys: 89, 18, 49, 58, 69 Hashing Function: hash(x) = x%10, where 10 is tablesize

0					
1					
2					
3					
4					
5					
6					
7					
8					
9	89				

Collisions (Linear Probing)

Keys: 89, 18, 49, 58, 69 Hashing Function: hash(x) = x%10, where 10 is tablesize

0					
1					
2					
3					
4					
5					
6					
7					
8		18			
9	89	89			

Collisions (Linear Probing)

Keys: 89, 18, 49, 58, 69 Hashing Function: hash(x) = x%10, where 10 is tablesize

0			49		
1					
2					
3					
4					
5					
6					
7					
8		18	18		
9	89	89	89		

Collisions (Linear Probing)

Keys: 89, 18, 49, 58, 69 Hashing Function: hash(x) = x%10, where 10 is tablesize

0			49	49	
1				58	
2					
3					
4					
5					
6					
7					
8		18	18	18	
9	89	89	89	89	

Collisions (Linear Probing)

Keys: 89, 18, 49, 58, 69 Hashing Function: hash(x) = x%10, where 10 is tablesize

0			49	49	49
1				58	58
2					69
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Collisions (Linear Probing)

- Works as long as the table is large for a free cell to always be found
- What would be an issue?

Collisions (Linear Probing)

- Works if the table is large for a free cell to always be found
- What would be an issue? **Could take a lot of time!**
- Even for a relatively empty table, continuous blocks of occupied cells begin to form – this effect is known as **clustering**
- If a key hashes into a cluster, it will often require several attempts to resolve the collision

Collisions (Quadratic Probing)

- In quadratic probing, f is a quadratic function of i

$$f(i) = i^2$$

- Cells are tried sequentially (with wrap around) in search of empty cell
- Example:
  - Keys: 89, 18, 49, 58, 69
  - Hashing Function: hash(x) = x%10, where 10 is tablesize

Collisions (Quadratic Probing)

Keys: 89, 18, 49, 58, 69 Hashing Function: hash(x) = x%10, where 10 is tablesize

0					
1					
2					
3					
4					
5					
6					
7					
8					
9	89				

Collisions (Quadratic Probing)

Keys: 89, 18, 49, 58, 69 Hashing Function: hash(x) = x%10, where 10 is tablesize

0					
1					
2					
3					
4					
5					
6					
7					
8		18			
9	89	89			

Collisions (Quadratic Probing)

Keys: 89, 18, 49, 58, 69 Hashing Function: hash(x) = x%10, where 10 is tablesize

0			49		
1					
2					
3					
4					
5					
6					
7					
8		18	18		
9	89	89	89		

Collisions (Quadratic Probing)

Keys: 89, 18, 49, 58, 69 Hashing Function: hash(x) = x%10, where 10 is tablesize

0			49	49	
1					
2				58	
3					
4					
5					
6					
7					
8		18	18	18	
9	89	89	89	89	

f(1) = 1
f(2) = 4

Collisions (Quadratic Probing)

Keys: 89, 18, 49, 58, 69 Hashing Function: hash(x) = x%10, where 10 is tablesize

0			49	49	49
1					
2				58	58
3					69
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

$f(1) = 1$
$f(2) = 4$

Collisions (Double Hashing)

- Double hashing uses a second hash function when a collision occurs.

$$f(i) = i * hash_2(x)$$

which mean, we apply a second hash function to x probe at a distance $hash_2(x)$, then $2*hash_2(x)$, then $3*hash_2(x)$, …

- NOTE: Poor choice of $hash_2$ can be disastrous!
- Example: $hash_2(x) = x\%9$

$$f(i) = i * hash_2(x)$$

if building on our last example we inserted 99, it would conflict with 89 which when executed the second has would be $i * 0$ for all i's.

Take Away: $hash_2$ can **NEVER** return 0!!!!!

Collisions (Double Hashing)

- Important that all cells can be probed (not possible in example because the table is not prime)
- This can be achieved by designing a $hash_2$ that looks like this:

$$hash_2(x) = R - (x\%R)$$

where R is a prime number smaller than the size of the hash table.

- Example:

$$hash_1(x) = x\%10$$

$$hash_2(x) = 7 - (x\%7)$$, where $R = 7$ in $hash_2$

Collisions (Double Hashing)

Keys: 89, 18, 49, 58, 69, 60 $hash_1(x) = x\%10$ $hash_2(x) = 7 - (x\%7)$, where 10 is tablesize

0						
1						
2						
3						
4						
5						
6						
7						
8						
9	89					

Collisions (Double Hashing)

Keys: 89, 18, 49, 58, 69, 60 $hash_1(x) = x\%10$ $hash_2(x) = 7 - (x\%7)$, where 10 is tablesize

0						
1						
2						
3						
4						
5						
6						
7						
8		18				
9	89	89				

Collisions (Double Hashing)

Keys: 89, 18, 49, 58, 69, 60 $hash_1(x) = x\%10$ $hash_2(x) = 7 - (x\%7)$, where 10 is tablesize

0			1			
1			2			
2			3			
3			4			
4			5			
5			6			
6			7 49			
7						
8		18	18			
9	89	89	89			

$hash_2 = 7 - (49\%7)$
$hash_2 = 7 - 0$
$hash_2 = 7$

Collisions (Double Hashing)

Keys: 89, 18, 49, 58, 69, 60 $hash_1(x) = x\%10$ $hash_2(x) = 7 – (x\%7)$, where 10 is tablesize

0				2		
1				3		
2				4		
3				5 58		
4						
5						
6			49	49		
7						
8		18	18	18		
9	89	89	89	1 89		

$hash_2 = 7 - (58\%7)$
$hash_2 = 7 - 2$
$hash_2 = 5$

Collisions (Double Hashing)

Keys: 89, 18, 49, 58, 69, 60 $hash_1(x) = x\%10$ $hash_2(x) = 7 - (x\%7)$, where 10 is tablesize

0					1 69	
1						
2						
3				58	58	
4						
5						
6			49	49	49	
7						
8		18	18	18	18	
9	89	89	89	89	89	

$hash_2 = 7 - (69\%7)$
$hash_2 = 7 - 6$
$hash_2 = 1$

Collisions (Double Hashing)

Keys: 89, 18, 49, 58, 69, 60 $hash_1(x) = x\%10$ $hash_2(x) = 7 - (x\%7)$, where 10 is tablesize

0					69	69
1						1
2						2 60
3				58	58	58
4						
5						
6			49	49	49	49
7						
8		18	18	18	18	18
9	89	89	89	89	89	89

$hash_2 = 7 - (60\%7)$
$hash_2 = 7 - 4$
$hash_2 = 3$
COLLISION!
$hash_2 = 2 * 3$
$hash_2 = 6$
COLLISION!
$hash_2 = 3 * 3$
$hash_2 = 9$
COLLISION!
$hash_2 = 4 * 3$
$hash_2 = 12$

$12\%10 = 2$

Rehashing

- What do you do if the table gets full?
- Build a new table that is twice as big (with a new hash function).
  - Scan the original table computing new key
  - Place element at new location based on key

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

h(x) = x%7

insert 23, now the table is 70% full, time to create new table twice as big

Rehashing

- Create a table that is twice as big!

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

Rehashing

- Create a table that is twice as big!
- Is 17 twice as big as 6?

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

Rehashing

- Create a table that is twice as big!
- Is 17 twice as big as 6? No, it's the first prime after 12.
- New hash function:

$$h(x) = x \% 17$$

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

Load Factor

- **Load factor** is the ratio of the number of elements in the hash table to the table size
- Ideal case is if **load factor** is less than 1
- As **load factor** approaches or is greater than 1 you might want to rehash

Load Factor

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

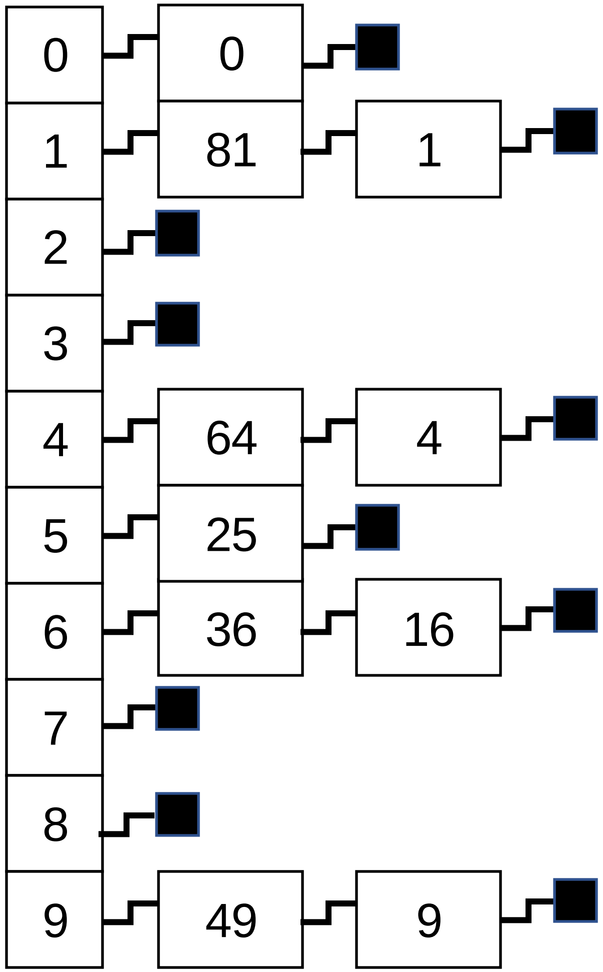
load factor = 5 / 7 = .71

Load Factor

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

←load factor = 5 / 7 = .71

load factor = 10 / 10 = 1→



Hash Table in STL

- STL provides a hash table implementation of sets and maps, they are unordered_set and unordered_map

Acknowledgement

These slides have been adapted and borrowed from books on the right as well as the CS340 notes of NIU CS department (Professors: Alhoori, Hou, Lehuta, and Winans) and many google searches.