Week 10:
Hash Table

Readings

- Required
  - [Weiss] ch20
- Exercise
  - 20.5

nus.soc.cs1102b.week10

6

Recap

	Unsorted Array/List	Sorted Array	BST	Hash Table
Insert	O(1)	O(N)	O(log N)	**O(1) avg**
Delete	O(N)	O(N)	O(log N)	**O(1) avg**
Find	O(N)	O(logN)	O(log N)	**O(1) avg**
findMin	O(N)	O(1)	O(log N)	**O(N)**
findMax	O(N)	O(1)	O(log N)	**O(N)**

nus.soc.cs1102b.week10

9

Hash Table is a data structure that support the most common dynamic set operations in constant time on average. It has many many applications.

Direct Addressing Table

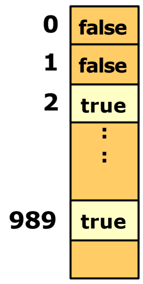
Direct address table, is a simplified version of hash table.

SBS Bus Problem

- **find(N)**
  - Does bus service no. N exist?
- **insert(N)**
  - Introduce a new bus service no. N
- **delete(N)**
  - Remove bus service no. N

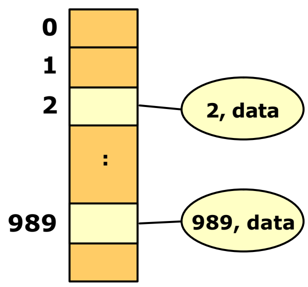
Consider the problem of maintaining information about SBS (and TIBS) bus services. We want to support three operations find, insert and delete.

SBS Bus Problem



Since bus numbers are integers between 0 – 999, we can create an array with 1000 booleans, initialized to false. If bus service N exists, just set position N to true. All find, delete, and insert can be done in O(1) time.

Direct Addressing Table



We can extends this idea, if we want to maintain additional data about a bus. Use an array of 1000 slots, each can reference to an Object.

Direct Addressing Table

insert (key, data)
  a[key] = data

delete (key)
  a[key] = null

search (key)
  return a[key]

Restrictions

- Keys must be integer
- Range of keys must be small

This works only if keys are integers, (cannot keep track of bus no NR10, 162M) and the range for the keys must be small (if keys are phone numbers, you need an array of size 10 million).

Hash Table

Hash Table is a generalization of direct addressing table, to remove these restrictions.

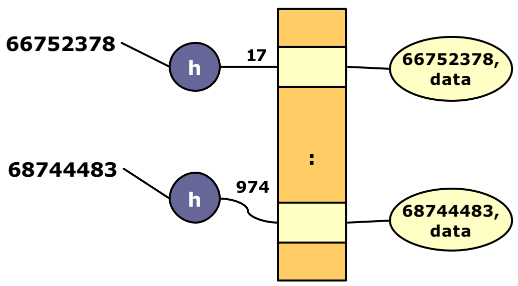
Idea

- Map non-integer keys to integers
- Map large integers to smaller integers

HASHING

The idea is to map any keys to small integers. We call this hashing. The function that map keys to integers are call hash function.

Hash Table



h is a hash function. This example shows how we map phone numbers to slot numbers between 0 and 999.

Hash Table

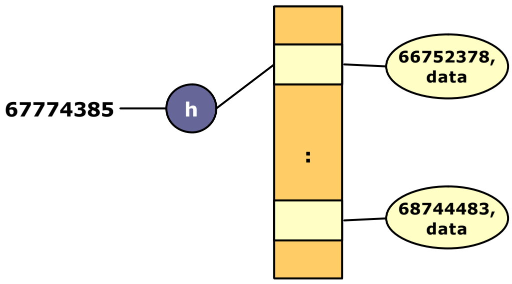
insert (key, data)
  a[h(key)] = data

delete (key)
  a[h(key)] = null

search (key)
  return a[h(key)]

Here is the pseudocode: notice that we have replaced key with h(key).
(This does not work! See next slide)

Hash Table



But a hash function does not guarantee that two different keys goes into different slots! This is called a “collision”.

Problem

- Two keys can have the same hash value

COLLISION

Overview of This Lecture

- How to hash?
- How to resolve collision?

To implement hash table, we need to answer two questions: how to define a hash function and how to resolve collision. They are important issues that can affect the efficiency of hash table.

Hash Functions

Good Hash Functions

- appear random
- fast
- depends on all information in the key
- keys that are close have hash values that are far apart

Perfect Hashing Function

- One-to-one mapping between keys and hash values.
- Maybe possible if all keys are known

It is possible to have a perfect hash function: where collision is guaranteed not to occur.

Uniform Hashing Function

- Distributes keys evenly
- Example
  - if k are integers uniformly distributed among 0 and X-1

$$k \in [0, X)$$

$$hash(k) = \left\lfloor \frac{km}{X} \right\rfloor$$

A uniform hashing function put a key into a slot with equal probability.

Hashing Integers

There are many ways to hash an integer.

Division Method

- Mapped into table of m slots

$$hash(k) = k \% m$$

The most popular one is the division method: where we use the mod operator (% in Java) to map an integer to values between 0 and m-1 (inclusive).

mod operator

- n mod m = remainder of n divided by m

How to pick m?

- m = 16
- m = 10
- m = 13

The choice of m (or hash table size) is important. If m is power of two, say 2^n, then key modulo of m is the same of last n bits of the key.
If m is 10^n, then our hash values is the last n digit of keys.
We usually pick m to be a prime number close to a power of two.

Rule

- Pick m to be a prime number **not too** close to power of two.

nus.soc.cs1102b.week10 31

Multiplication Method

1. Multiply by a number 0 <= A < 1
2. Extract the fractional part
3. Multiply by m

$$hash(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

nus.soc.cs1102b.week10 32

Another method is the multiplication method. The golden ratio = (sqrt(5) – 1)/2 seems to be a good choice for A.

Hashing Strings

Hashing of Strings

```
hash(s, m)
  sum = 0
  foreach character c in s
     sum += c
  return sum % m
```

nus.soc.cs1102b.week10 34

To hash a string, we can just sum up all ascii values of ecah characters.

hash("Tan Ah Teck", 11)

= ("T" + "a" + "n" + " " +
"A" + "h" + " " +
"T" + "e" + "c" + "k") % 11

= (84 + 97 + 110 + 32 +
65 + 104 + 32 +
84 + 101 + 99 + 107) % 11

= 825 % 11
= 0

Hashing of Strings

- Lee Chin Tan
- Chen Le Tian
- Chan Tin Lee

Does not depend on position of characters!

This only depends on the characters that are present in a string, not their positions.

Hashing of Strings

hash(s)

```
sum = 0
foreach character c in s
    sum += sum*37 + c
return sum % m
```


A better way is to "shift" the sum everytime, so that the position affects the calculated hash values. (Note: Java's String.hashCode( ) uses 31 instead of 37)

Collision Resolution

Probability of Collision

- von Mises Paradox: "How many people must be in a room before the probability that some share a birthday, ignoring the year and leap days, becomes at least 50 percent?"

nus.soc.cs1102b.week10

39

Probability of Collision

Q(n) = Probability of unique birthday for n people

$$= \frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \ldots \frac{365-n+1}{365}$$

P(n) = Probability of collisions for n people
= 1 – Q(n)

P(23) = 0.507

nus.soc.cs1102b.week10

40

Probability of Collision

Collision is very likely!

nus.soc.cs1102b.week10

41

If we more than 23 keys into a table with 365 slots, more than half of the time we get collision.

Collision Resolutions

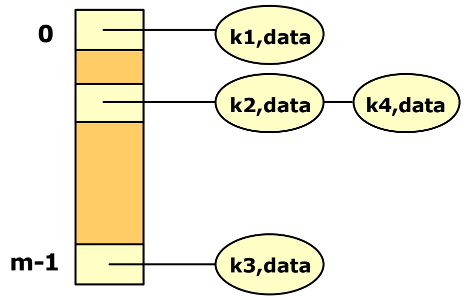
- Separate Chaining
- Linear Probing
- Quadratic Probing
- Double Hashing

nus.soc.cs1102b.week10

42

Separate Chaining

Idea



Separate Chaining is the most straight forward method, using a linked-list to store the collided keys.

Hash Table

insert (key, data)
insert data into the list a[h(key)]

delete (key)
delete data from the list a[h(key)]

search (key)
find key from the list a[h(key)]

Insertion can be done in O(1) time. But deletion and search takes O(n) time where n is the length of the list.

Analysis

- n: number of keys
- m: number of slots
- L: load factor

- L = n/m
- Average length of list = L

Average Running Time

- Search O(1 + L)
- Insert O(1)
- Delete O(1 + L)

- If L is bounded by some constant, then all three operations are O(1)

However, we can bound the length of the chain by a constant.

Rehashing

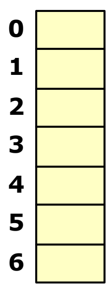
- To keep L bounded, we may need to reconstruct the whole table

When ever the load factor exceeds the bound, we need to rehash all keys into a bigger table (increase m to reduce L)

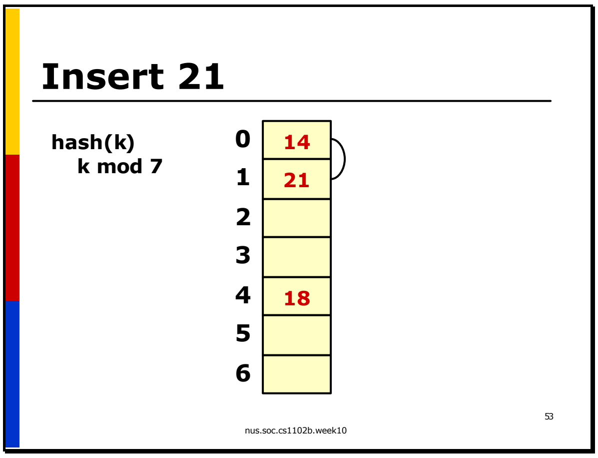
Linear Probing

Linear Probing

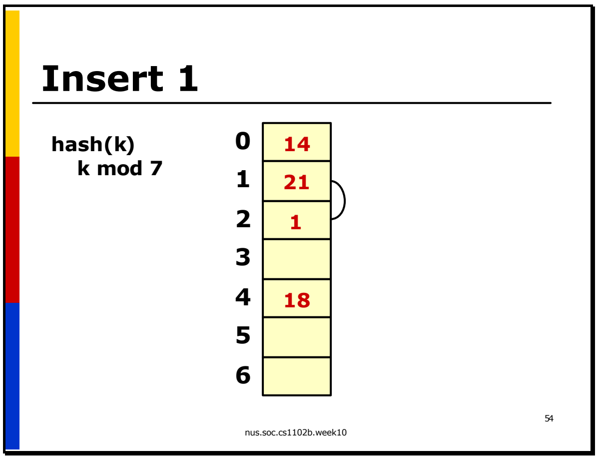
hash(k)
k mod 7



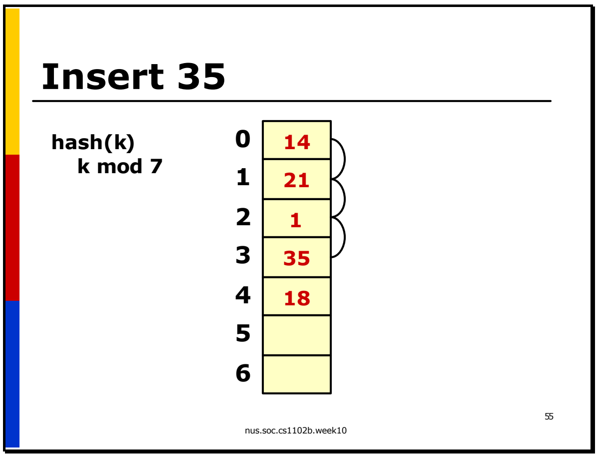
In linear probing, when we get a collision, we scan through the table looking for an empty slot (wrapping around when we reach the last slot)

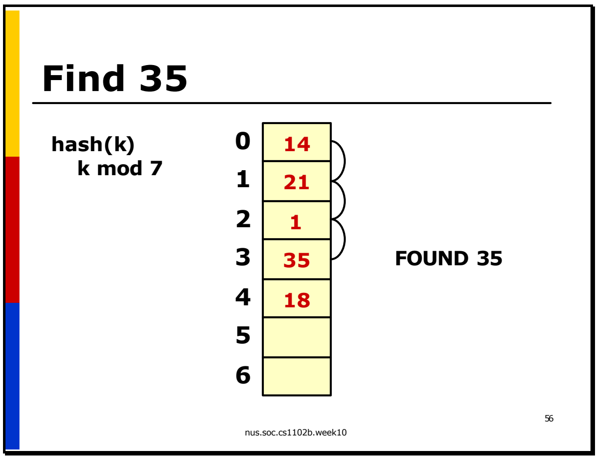


21 collides with 14. Look for the next empty slot.

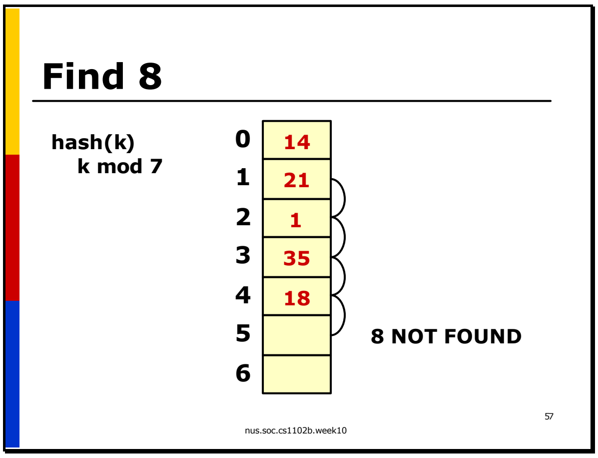


1 collided with 21. Look for an empty slot.

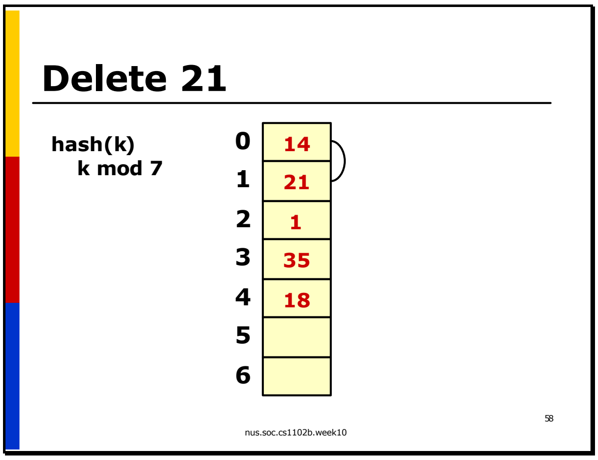




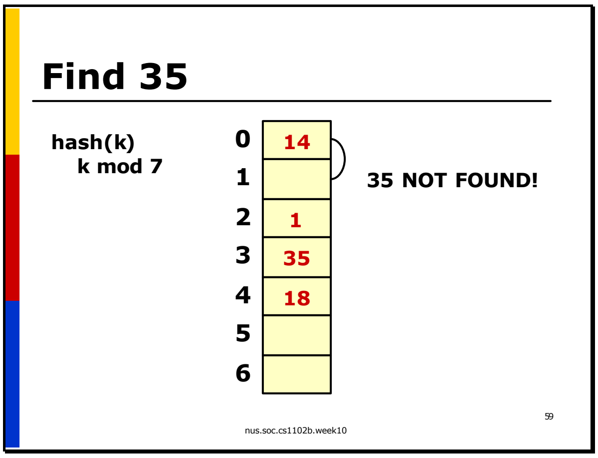
Find a values is similar to find. We probe the array starting from the original hash position (in this case hash(35) = 0)



When probing, if we reach an empty slot, we know that the value does not exist in the hash table.



To delete, we first find the value, and remove it from the table.



We cannot simply remove a value, because it can affect find( ) !

Problem

Cannot Delete!

nus.soc.cs1102b.week10

60

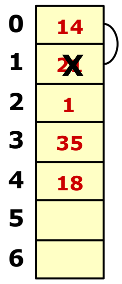
How to delete?

- Lazy Deletion
- Three different states
  - occupied
  - occupied but mark as deleted
  - empty

When a value is removed from linear probed hash table, we just mark it as “deleted”, instead of emptying the slot.

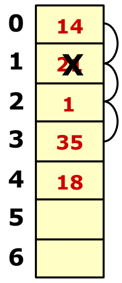
Delete 21

hash(k)
k mod 7



Find 35

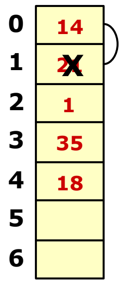
hash(k)
k mod 7



FOUND 35

Insert 15

hash(k)
k mod 7



When we insert, we can put a value into either an empty slot, or a slot that has been marked as deleted.

Insert 15

hash(k)
k mod 7

0	14
1	15
2	1
3	35
4	18
5	
6	

Problem

Primary Clustering

The problem with linear probing is that it can create many consecutive occupied slots, increasing the running time of find/insert/delete. This is called primary clustering.

Quadratic Probing

An improvement to linear probing is quadratic probing.

Linear Probing

hash(key)
(hash(key) + **1**) % m
(hash(key) + **2**) % m
(hash(key) + **3**) % m
:

The probe sequence for linear probing is this.

Quadratic Probing

hash(key)
(hash(key) + **1**) % m
(hash(key) + **4**) % m
(hash(key) + **9**) % m
:

For quadratic probing, we use this probe sequence.

Insert 3

hash(k)
k mod 7

0	
1	
2	
3	**3**
4	**18**
5	
6	

Insert 38

hash(k)
k mod 7

0	**38**
1	
2	
3	**3**
4	**18**
5	
6	

Notice that the calculation of +1 +4 +9 .. starts from the *original* hash position. If we were to start from the *previous* probe position, the probe sequence should be +1 +3 +5 ..+ (2i -1).

(Q: Show mathematically that they are the same)

Theorem

- If L < 0.5, and m is prime, then we can always find an empty slot if table is not full.

How can we be sure that quadratic probing always terminate? Insert 12 into the previous example, follow by 10. See what happen?

Problems

- If two keys have the same initial position, their probe sequence is the same.
- Secondary clustering.

Using quadratic probing requires more careful design of hash table. It also suffers from a (less minor) problem – if two keys has the same initial position, they have the same probe sequence.

Double Hashing

Double hashing uses a second hash function to calculate the probe sequence, so unless two keys have the same hash values for both hash functions, they have different probe sequences.

Double Hashing

hash(key)
(hash(key) + $hash_2$(key)) % m
(hash(key) + 2*$hash_2$(key)) % m
(hash(key) + 3*$hash_2$(key)) % m
:

$hash_2$(key) is the secondary hash function.

Insert 21

hash(k)
k mod 7

$hash_2$(k)
k mod 5

0	14
1	21
2	
3	
4	18
5	
6	

We use k%5 as the secondary hash function in this example. Can you give two keys that have the same probe sequence in this example?

If we insert 21, the probe sequence is the same as linear probing.

Insert 4

hash(k)
k mod 7

$hash_2(k)$
k mod 5

0	14
1	21
2	
3	
4	18
5	4
6	

If we insert 4, the probe sequence is 4, 8, 12 … (from the first probe position) or 4, 4, 4, … (from the previous probe position).

Insert 35

hash(k)
k mod 7

$hash_2(k)$
k mod 5

0	14
1	21
2	
3	
4	18
5	4
6	

But if we insert 35, the probe sequence is 0, 0, 0, …
What is wrong?

Warning

- **Secondary hash function must not evaluate to 0 !**
- Change $hash_2$(key) to

  $hash_2$(key) = 5 – (key % 5)

Good Collision Resolution

- Minimize clustering
- Can find an empty slot if L is small
- Give different probe sequence when initial probe is the same
- Fast