COMP20007 Design of Algorithms
Semester 1 2015

Map/Dictionary data structures

Hash tables

Lecture Objectives

- Learn about the Hash Table data structure

- After this lecture you should be able to
  - implement a hash table with separate chaining
  - choose a suitable hash function for your data
  - understand different approaches to collision handling
    - Open Addressing
    - Cuckoo Hashing
    - Hopscotch Hashing

Map/Dictionary abstract data type

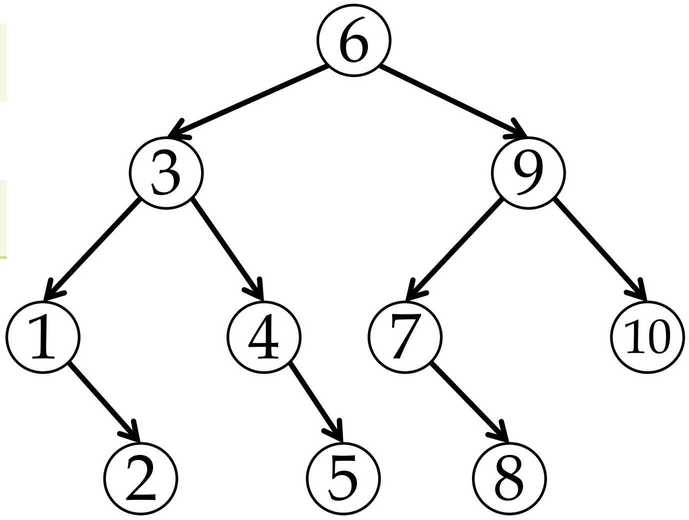
- What is required of a map?
  - create empty map
  - insert (key, value)
  - delete (key, value)
  - find(key)

Maybe:
- get size
- get an iterator
- get a sorted iterator

Map/dictionary as a BBST

Operation	Worst
create	$\Theta(1)$
insert	$\Theta(?)$
delete	$\Theta(?)$
find	$\Theta(?)$
size	$\Theta(?)$
iterate over	$\Theta(?)$
find smallest	$\Theta(?)$
find largest	$\Theta(?)$
find successor	$\Theta(?)$

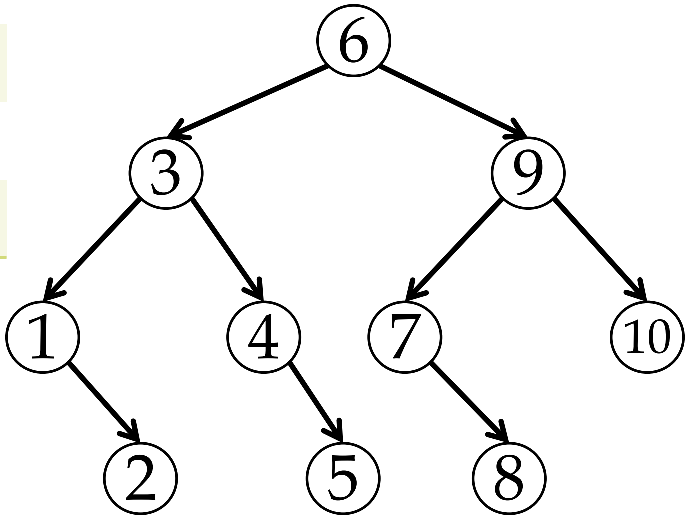
Successor = smallest thing in right sub-tree, or parent of left-child ancestor.



Map/dictionary as a BBST

Operation	Worst
create	$\Theta(1)$
insert	$\Theta(\log n)$
delete	$\Theta(\log n)$
find	$\Theta(\log n)$
size	$\Theta(n)$
iterate over	$\Theta(n)$
find smallest	$\Theta(\log n)$
find largest	$\Theta(\log n)$
find successor	$\Theta(\log n)$

Successor = smallest thing in right sub-tree, or parent of left-child ancestor.



Hash tables: map keys into an array

- So we need…
  - An array $A[0 \ldots m\text{-}1]$
  - A function h : key ➔ integer in range $[0, m)$

- $h(\text{key}) = \text{key} \bmod m$
- Insert(key, data): $A[h(\text{key})] = \text{data}$ $\Theta(1)$
- Search(key): return $A[h(\text{key})]$ $\Theta(1)$
- Delete(key): $A[h(\text{key})] = \text{NULL}$ $\Theta(1)$

Fantastic! Why do we need BBST?

Operation	BBST Worst	Hash Table Worst
create	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(\log n)$	$\Theta(1)$
delete	$\Theta(\log n)$	$\Theta(1)$
find	$\Theta(\log n)$	$\Theta(1)$
size	$\Theta(n)$	$\Theta(1)$
iterate over	$\Theta(n)$	$O(m)$
find smallest	$\Theta(\log n)$	$O(m)$
find largest	$\Theta(\log n)$	$O(m)$
find successor	$\Theta(\log n)$	$O(m)$

Ok, so no order information. Who cares?

- Exercise
  - $m = 8$
  - $h(\text{key}) = \text{key} \bmod m$
  - Insert 15, 9, 4, 7

0	
1	
2	
3	
4	
5	
6	
7	

Ok, so no order information. Who cares?

- Exercise
  - $m = 8$
  - $h(\text{key}) = \text{key} \bmod m$
  - Insert 15, 9, 4, 7

0	
1	
2	
3	
4	
5	
6	
7	15

Ok, so no order information. Who cares?

- Exercise
  - $m = 8$
  - $h(\text{key}) = \text{key} \bmod m$
  - Insert 15, 9, 4, 7

0	
1	9
2	
3	
4	
5	
6	
7	15

Ok, so no order information. Who cares?

- Exercise
  - $m = 8$
  - $h(\text{key}) = \text{key} \bmod m$
  - Insert 15, 9, 4, 7

0	
1	9
2	
3	
4	4
5	
6	
7	15

Ok, so no order information. Who cares?

- Exercise
  - $m = 8$
  - $h(\text{key}) = \text{key} \bmod m$
  - Insert 15, 9, 4, 7

0	
1	9
2	
3	
4	4
5	
6	
7	15

Collision

How to handle collisions?

- Make m bigger
  - Pigeon hole principle: n pigeons, m holes, $m \geq n$
  - If we want to handle integer keys up to U, we need $m \geq U$
  - For integers, $U = 2^{32}$ or 2^{64}
  - For strings? Say 10 letter words, lower case: 26^{10}
  - Not viable for large U

- If $m \geq n$, make h "better"
  - If we know all keys in advance, can build a perfect hash function (uses graphs, but not studied here)

Even if $m \geq n$, can still get collisions

- Birthday "paradox": what's the probability of 2 people sharing a birthday (day/month) out of n people?

Even if $m \geq n$, can still get collisions

- Birthday "paradox": what's the probability of 2 people sharing a birthday (day/month) out of n people?

  $n = 1$, Pr no collision = 1

  $n = 2$, Pr no collision = $(m-1)/m$

  $n = 3$, Pr no collision = $(m-1)/m$ * $(m-2)/m$

  …

  n, Pr no collision = $\prod_{i=1}^{n} \frac{m-i+1}{m} = \frac{m!}{(m-n)!\,m^n}$
- $n = 23$, $m = 365$: 0.493
- $n = 50$, $m = 365$: 0.03

So even if *h* really randomises keys…

- …you will probably get collisions.
- A nice comparison of hash functions:

  "Which hashing algorithm is best for uniqueness and speed". stackexchange.com.

- And the winner is MurmurHash
  http://code.google.com/p/smhasher/

- But watch out for hashDOS…
  http://emboss.github.io/blog/2012/12/14/breaking-murmur-hash-flooding-dos-reloaded/

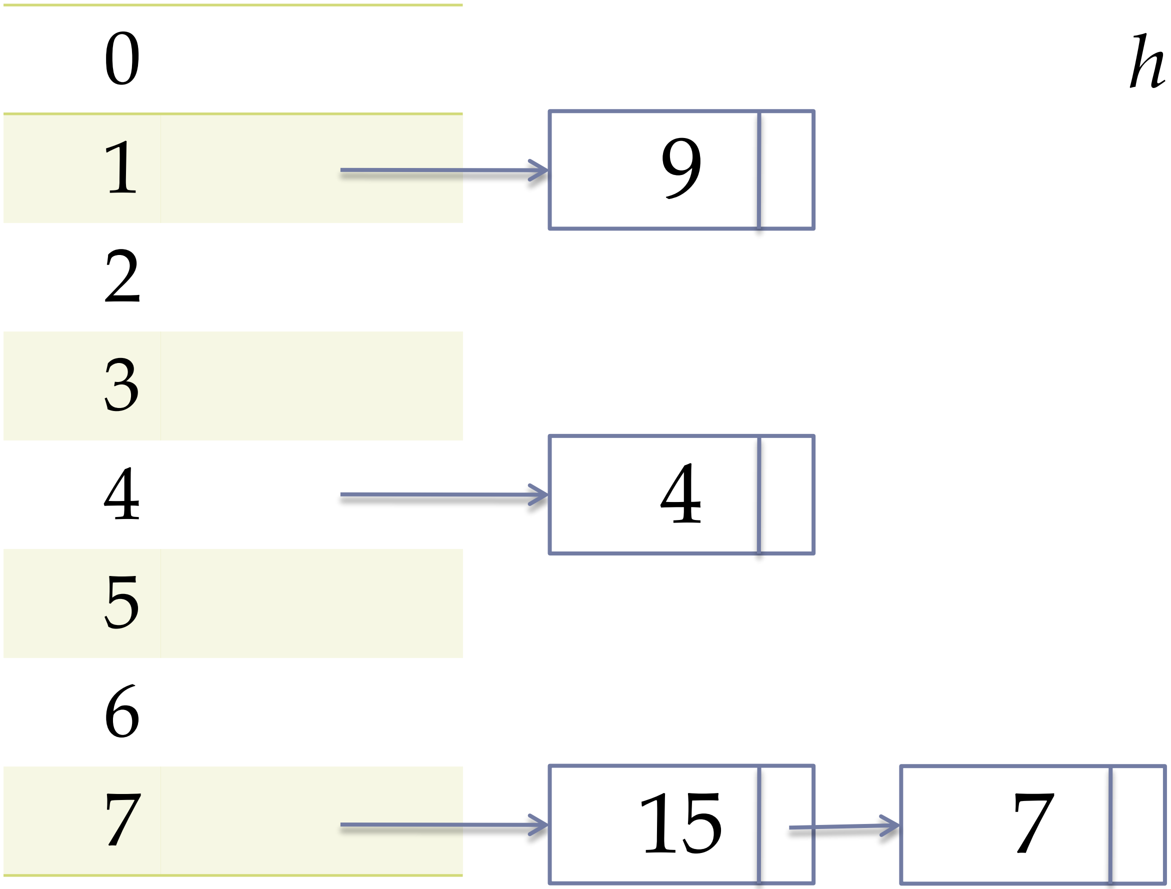
So we have to handle collisions

- Separate chaining
  - Each element in A is a linked list (perhaps move to front (MTF))
  - Or each element in A is a BBST (perhaps Splay)
  - Or just an unordered array (good cache hits)

- Open Addressing
  - Linear probing: $h(\text{key}, i) = (h(\text{key}) + i) \bmod m$
  - Double hashing: $h(\text{key}, i) = (h_1(\text{key}) + i.h_2(\text{key})) \bmod m$
  - Cuckoo hashing
    - Move current element into B & vice versa
    - What happens when A and B are full: rehash everything into bigger tables
    - Visualisation at http://www.lkozma.net/cuckoo_hashing_visualization/

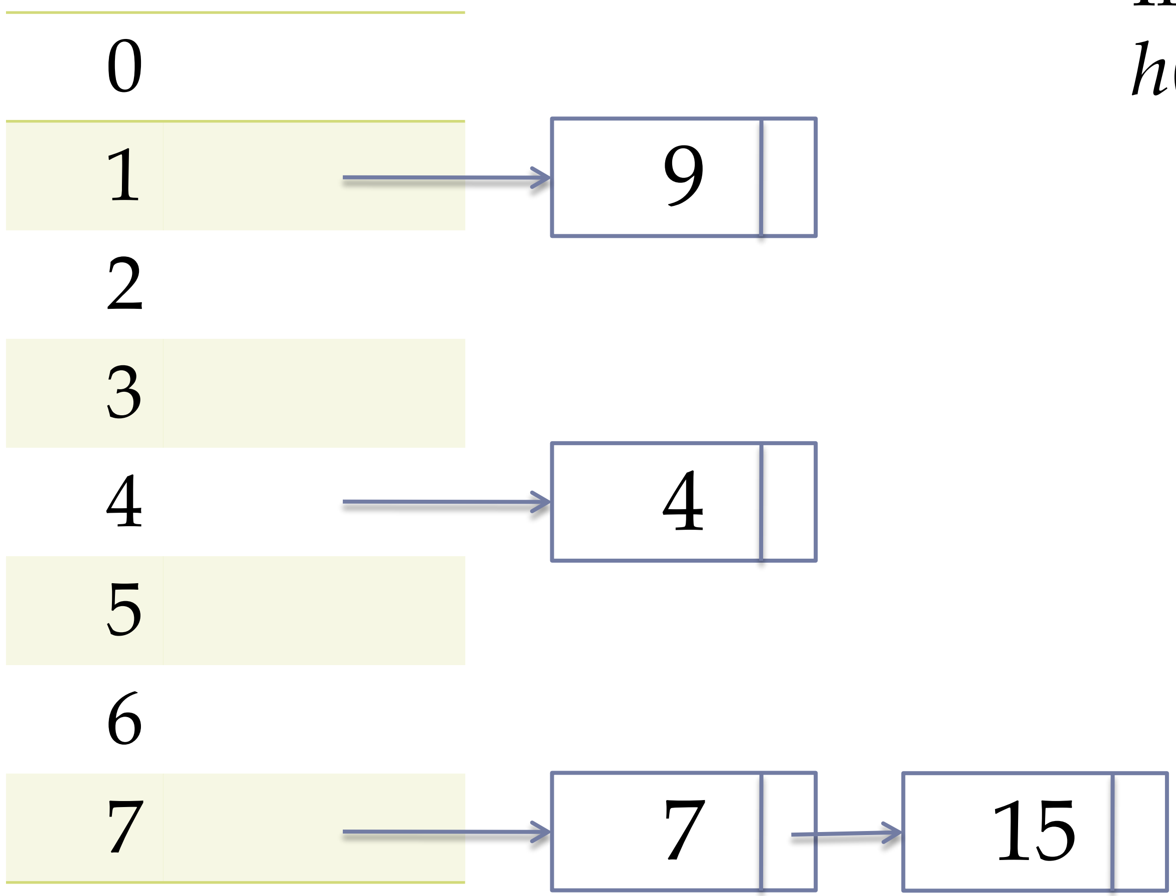
Separate Chaining

Insert 15, 9, 4, 7
$h(\text{key}) = \text{key} \bmod 8$

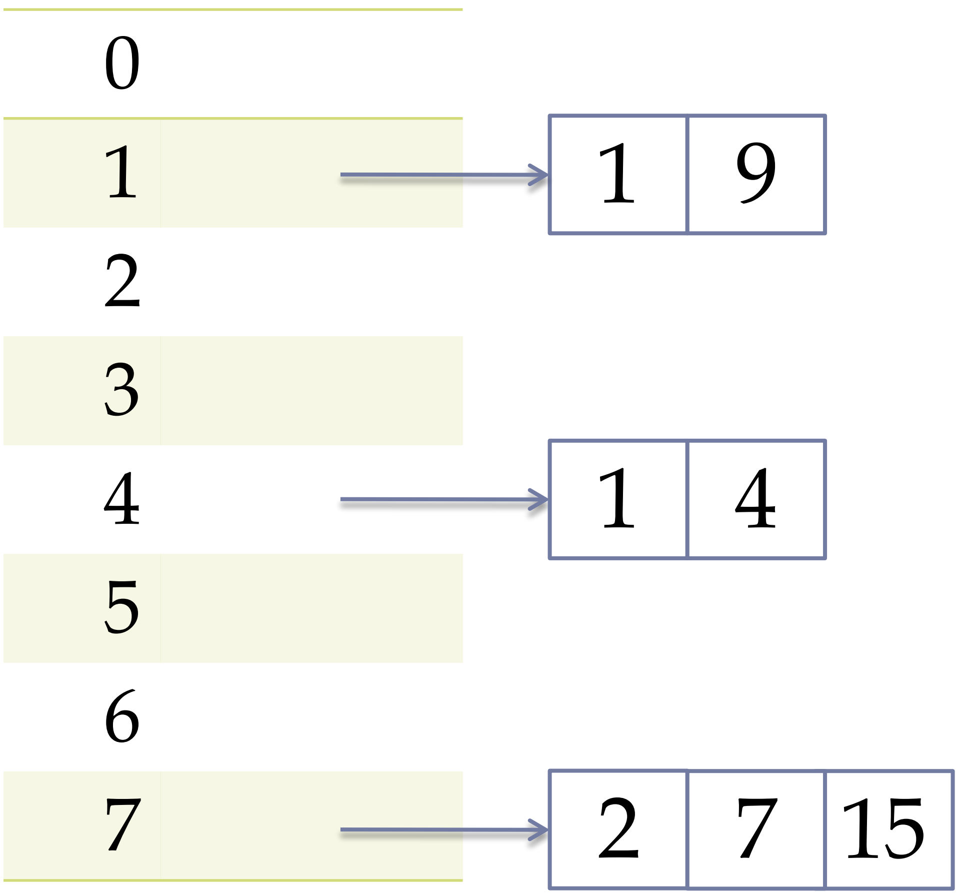


Separate Chaining with MTF

Insert 15, 9, 4, 7
$h(\text{key}) = \text{key} \bmod 8$



Separate Chaining with MTF array



Insert 15, 9, 4, 7

$h(\text{key}) = \text{key} \bmod 8$

First element is number of elements in the rest of array.

Separate Chaining - Analysis

- Assuming *h* is O(1) and spreads keys
- Insert: Θ(1) if MTF, usually O(1) if n/m low
- Search: expect O(1) if n/m low
- Delete: expect O(1) if n/m low

- Array may be faster than linked list (caching)
- MTF will adapt to skew access patterns

Open Addressing – Linear Probing

0	7
1	9
2	
3	
4	4
5	
6	
7	15

Insert 15, 9, 4, 7
$h(\text{key}) = \text{key} \bmod 8$

If collide, just search +1 (with wrap around – mod m) until find a gap

Open Addressing – Linear Probing

0	7 0
1	9
2	
3	
4	4
5	
6	
7	15

Insert 15, 9, 4, 7
$h(\text{key}) = \text{key mod } 8$

Insert 0. Clash!

Open Addressing – Linear Probing

0	7	
1	9	0
2		
3		
4	4	
5		
6		
7	15	

Insert 15, 9, 4, 7
$h(\text{key}) = \text{key} \bmod 8$

Search +1. Another clash!

Open Addressing – Linear Probing

0	7
1	9
2	0
3	
4	4
5	
6	
7	15

Insert 15, 9, 4, 7, 0
$h(\text{key}) = \text{key} \bmod 8$

Search +2. Success.

Open Addressing – Linear Probing

- Good cache behaviour
  - Scanning left to right except for wrap around
- Keys tend to cluster

- Generally in OA we want a hash function of two arguments: key and *i*, the rehash attempt number

- Linear probing: $h(\text{key}, i) = (h(\text{key}) + i) \bmod m$

Original hash function

Open Addressing – Double hashing

- $h(\text{key}, i) = (h_1(\text{key}) + i.h_2(\text{key})) \bmod m$

2 hash functions

Open Addressing – Double Hashing

0	
1	9
2	
3	
4	4
5	
6	
7	15

Insert 15, 9, 4, 7

$h_1(\text{key}) = \text{key mod } 8$

$h_2(\text{key}) = \text{key mod } 3 + 1$

$h(\text{key}, i) = (h_1(\text{key}) + i.h_2(\text{key})) \text{ mod } m$

Note h_2 should not evaluate to 0

Q: Why have second hash function, not just a fixed offset, e.g. $h_2(\text{key}) = 7$?

Open Addressing – Double Hashing

0	
1	9
2	
3	
4	4
5	
6	
7	15 7

Insert 15, 9, 4, 7

$h_1(\text{key}) = \text{key} \bmod 8$

$h_2(\text{key}) = \text{key} \bmod 3 + 1$

$h(\text{key}, i) = (h_1(\text{key}) + i.h_2(\text{key})) \bmod m$

$i = 0$. Clash!

$h_1(7) = 7 \bmod 8 = 7$

$h_2(7) = 7 \bmod 3 + 1 = 2$

Open Addressing – Double Hashing

0	
1	9 7
2	
3	
4	4
5	
6	
7	15

Insert 15, 9, 4, 7

$h_1(\text{key}) = \text{key mod } 8$

$h_2(\text{key}) = \text{key mod } 3 + 1$

$h(\text{key}, i) = (h_1(\text{key}) + i.h_2(\text{key})) \text{ mod } m$

i = 1. Clash!

$h_1(7) = 7 \text{ mod } 8 = 7$

$h_2(7) = 7 \text{ mod } 3 + 1 = 2$

Open Addressing – Double Hashing

0	
1	9
2	
3	7
4	4
5	
6	
7	15

Insert 15, 9, 4, 7

$h_1(\text{key}) = \text{key mod } 8$

$h_2(\text{key}) = \text{key mod } 3 + 1$

$h(\text{key}, i) = (h_1(\text{key}) + i.h_2(\text{key})) \text{ mod } m$

$i = 2$. *Success!*

$h_1(7) = 7 \text{ mod } 8 = 7$

$h_2(7) = 7 \text{ mod } 3 + 1 = 2$

Open Addressing – Double Hashing

- Must be careful choosing h_1 and h_2
- eg If m=1024 and h_2(key)=256, probe sequence would be: h_1(key) + {256, 512, 768, 0, 256, …}
  which only examines 4/1024 = 1/256 slots
- Generally, h_2(key) must be relatively prime to m
- Relatively Prime = no common divisor other than 1
- Easy way 1
  - Choose m to be a power of 2
  - Design h_2(key) to always return an odd number
- Easy way 2
  - Choose m to be prime
  - Design h_2 to be in (0,m): $h2$(key) = 1 + (key mod (m-1))

Summary

- Open Addressing does not require extra structures, but number of keys is limited to table size.
  - Watch out for delete!
- Chaining allows expansion beyond m
- Can be slow? Let's see in the Workshop.
- Access is O(1) if hash function is O(1)
- Insert/Delete can be more than O(1) if table is heavily loaded (n/m high)