Hashing

Introduction

- How fast can we search?
  - Vector
  - Unordered or Ordered list
  - Binary Search Trees
  - AVL trees
- When log(n) is just too big …
  - Real-time databases
  - Air traffic control
  - Packet routing
- Can we break log(n) barrier?

Hash Tables

- Use a **key** (arbitrary string or number) to index directly into an array. O(1) time to access records.
- A[“kreplach”] = “tasty stuffed dough”
  - Need a *hash function* to convert the key to an integer

	Key	Data
0	kim chi	spicy cabbage
1	kreplach	tasty stuffed dough
2	kiwi	Australian fruit

Properties of Good Hash Functions

1. Must return **number** [0, …, tablesize-1]
2. Should be **efficiently** computable – O(1) time
3. Should not waste space unnecessarily (**even distribution**).
   1. For every index, there is at least one key that hashes to it
   2. Load factor lambda λ = (number of keys / TableSize)
4. Should **minimize collisions** (different keys hashing to same index).

Integer Keys

- **Hash(x) = x % TableSize**
- Good idea to make TableSize ***prime***. Why?
  - Because keys are typically not randomly distributed, but usually have some *pattern*
    - mostly even
    - mostly multiples of 10
    - in general: mostly multiples of some k
  - If k is a factor of TableSize, then only (TableSize/k) slots will ever be used.
  - Since the only factor of a prime number is itself, this phenomena only hurts in the (rare) case where k=TableSize

Strings as Keys

- If keys are strings, can get an integer by adding up ASCII values of characters in *key (from 0 – 127)* http://www.asciitable.com/

```
for (i=0;i<key.length();i++)
      hashVal += key.charAt(i);
```

- **Problem 1**: What if *TableSize* is 10,000 and all keys are 8 or less characters long? Keys will hash only to positions 0 through 8*127 = 1016, and the rest of the table is empty.
- **Problem 2**: What if keys often contain the same characters (“abc”, “bca”, etc.)? Similar summation

Collisions and their Resolution

- A **collision** occurs when two different keys hash to the same value
  - **Example:** For *TableSize* = 17 and hash function (X mod 17), the keys 18 and 35 hash to the same value. 18 mod 17 = 1 and 35 mod 17 = 1
- Cannot store both data records in the same slot in array.
- Two different methods for collision resolution:
  1. **Open hashing (separate Chaining):** Use a dictionary data structure (such as a linked list) to store multiple items that hash to the same slot
  2. **Closed Hashing (or *probing, open addressing*):** search for empty slots using a second function and store item in first empty slot that is found

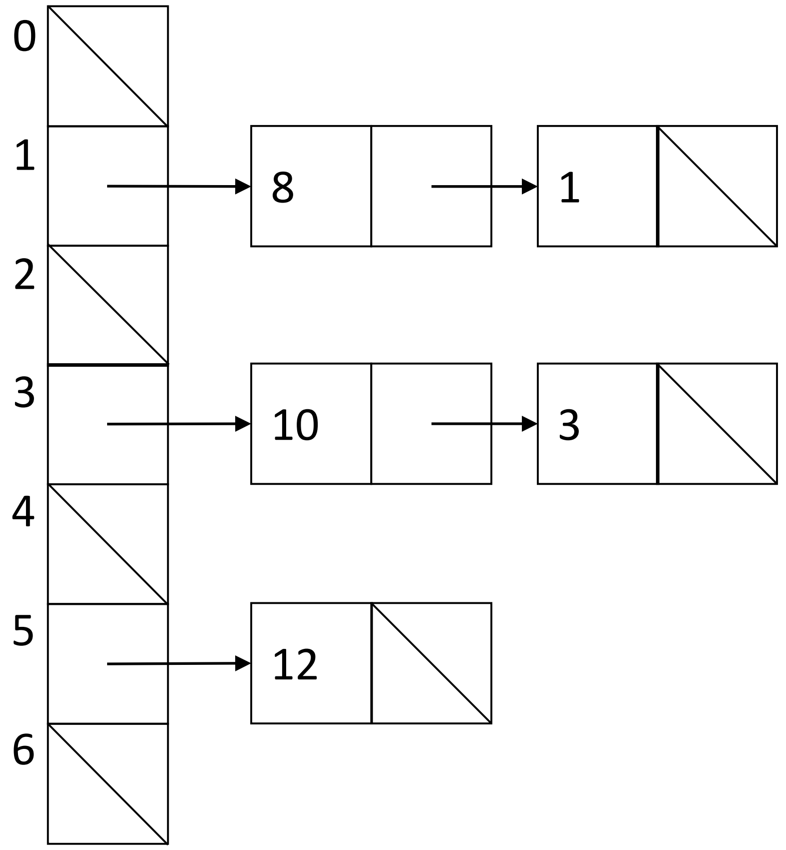
(1) Open Hashing

- The idea is to **create an unordered linked list (chain)** of all elements that hash to the same value.
- The array elements are pointers to the first nodes of the lists.
- A new item is inserted to the **front** of the list

Properties

- Performance degrades with length of chains
- **λ can be greater than 1**

Insert 1, 8, 12, 3, and 10 using
Hash(x) = x% 7



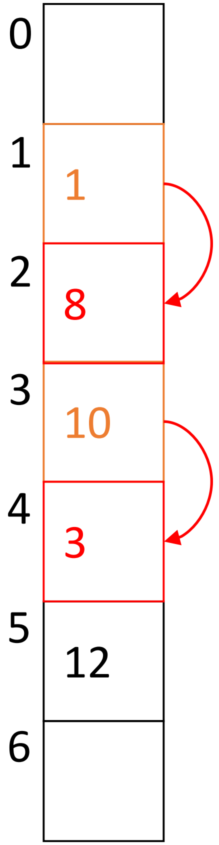
(2) Closed Hashing

Problem with separate chaining:

1. Memory consumed by pointers
2. Time to search the linked list

Insert 1, 8, 12, 10, and 3 using
Hash(x) = x% 7

h(1) = h(8)
h(10) = h(3)



- In an open addressing hashing system, all the data go inside the table. Thus a bigger table is needed.
- If a collision occurs, alternative cells are tried until an empty cell is found.

Properties

- performance degrades with **difficulty of finding** right spot
- $\lambda \leq \mathbf{1}$ (generally below 0.5)

Collision Resolution by Closed Hashing

- Given an item x, try cells $h_0(x)$, $h_1(x)$, $h_2(x)$, …, $h_i(x)$
- **$H_i(x)$ = (hash(x) + f(i)) mod *TableSize***
- ***f*** is the ***collision resolution*** **function**. Some possibilities:
  1. **Linear**: $f(i) = i$
  2. **Quadratic:** $f(i) = i^2$
  3. **Double Hashing:** $f(i) = i * hash_2(x)$

(2.1) Linear Probing

- When collision occurs, scan down the array one cell at a time looking for an empty cell
  - $H_i(x) = (hash(x) + i) \bmod \mathit{TableSize} \quad (i = 0, 1, 2, \ldots)$
  - Compute hash value and increment it until a free cell is found

Linear Probing Example

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2
0	14	14	14	14
1		8	8	8
2			21	21
3				2
4				
5				
6				
probes:	1	1	3	2

Exercise: Using linear probing, insert 89, 18, 49, 58 and 9 into a hash table of size 10.

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

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

Find

- The find algorithm follows the same probe sequence as the insert algorithm.
  - A find for 58 would involve 4 probes.
  - A find for 19 would involve 5 probes.

Drawbacks of Linear Probing

1. Works until array is full, but as number of items N approaches *TableSize* ($\lambda \approx 1$), **access time** approaches O(N)
2. Very prone to **cluster formation** (as in previous examples)
   - If a key hashes anywhere into a cluster, finding a free cell involves going through the entire cluster – and making it grow!
   - **Primary clustering** – clusters grow when keys hash to values close to each other
3. Can have cases where table is **empty** except for a few clusters
   - Does not satisfy good hash function criterion of *distributing keys uniformly*

(2.2) Quadratic Probing

- **Main Idea:** Spread out the search for an empty slot – Increment by i^2 instead of i

- $H_i(x) = (hash(x) + i^2)$ % *TableSize*

  $H_0(x) = (hash(x) + 0)$ % TableSize

  $H_1(x) = (hash(x) + 1)$ % TableSize

  $H_2(x) = (hash(x) + 4)$ % TableSize

  $H_3(x) = (hash(x) + 9)$ % TableSize

Quadratic Probing Example

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2
0	14	14	14	14
1		8	8	8
2				2
3				
4			21	21
5				
6				
probes:	1	1	3	1

Problem With Quadratic Probing

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2	insert(7) 7%7 = 0
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					
probes:	1	1	3	1	??

Problem with Quadratic Probing

- Clustering may still happen with quadratic probing, and form so called **secondary clusters**, which are less harmful than primary clusters of linear probing.
- If TableSize is prime and Load Factor λ<= ½, quadratic probing will find an empty slot; **for greater λ, might not**.

Exercise: Using quadratic probing, insert 89, 18, 49, 58 and 9 into a hash table of size 10.

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

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

(2.3) Double Hashing

- Spread out the search for an empty slot by using a second hash function
  - *No primary or secondary clustering*
- **$H_i(x) = (hash_1(x) + i*hash_2(x))$ mod *TableSize***
  for $i = 0, 1, 2, \ldots$
- Good choice of $Hash_2(x)$ can guarantee does not get "stuck" as long as $\lambda < 1$.
- The function $hash_2(x)$ must never evaluate to **zero**.
- A function such as $\boldsymbol{hash_2(x) = R - (x \bmod R)}$ where R is a prime smaller than *TableSize* will work well.

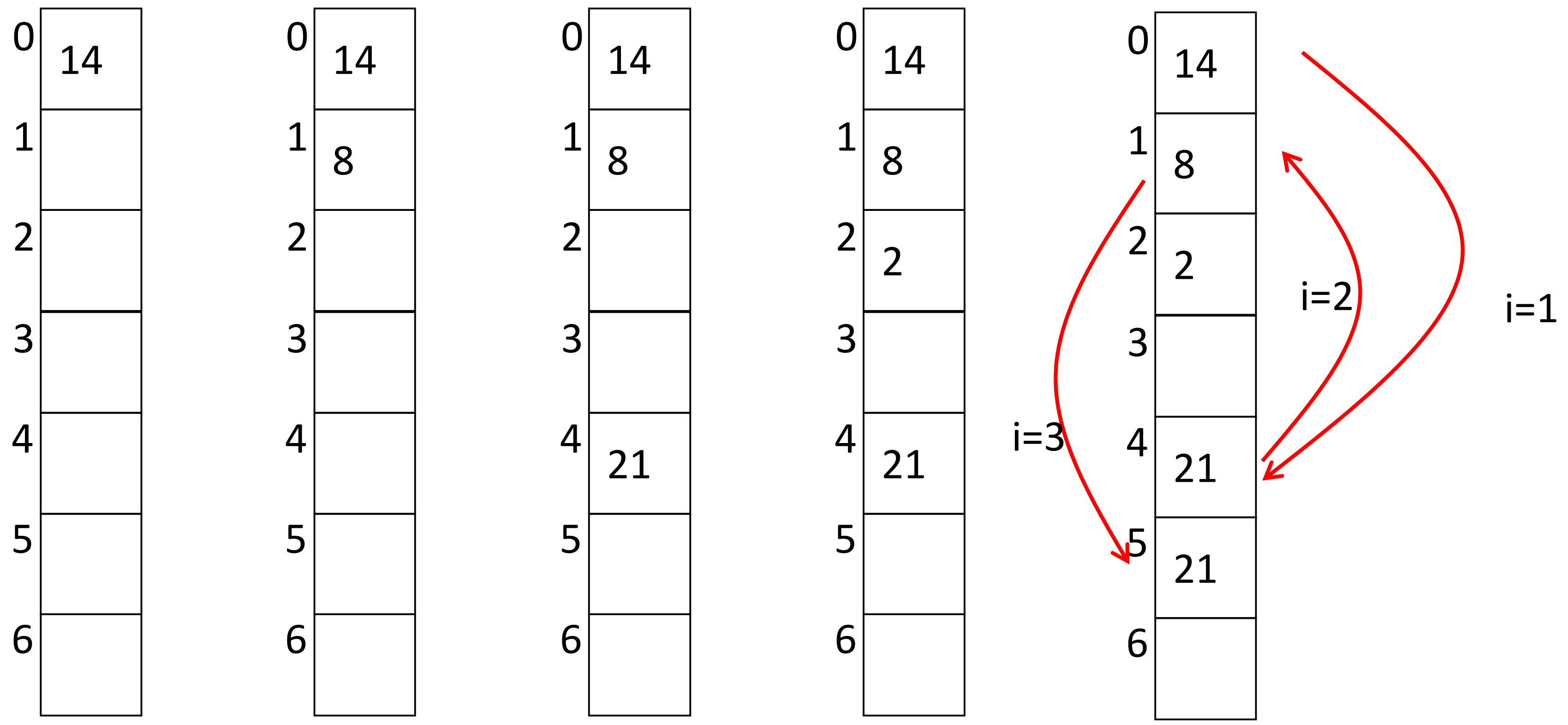
Double Hashing Example

$hash_2(x)$ = 5 – (x mod 5)

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0 5-(21%5)=4	insert(2) 2%7 = 2	insert(21) 21%7 = 0 5-(21%5)=4
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					
probes:	1	1	2	1	??

Double Hashing Example

insert(14)	insert(8)	insert(21)	insert(2)	insert(21)
14%7 = 0	8%7 = 1	21%7 =0 5-(21%5)=4	2%7 = 2	21%7 = 0 5-(21%5)=4



probes:	1	1	2	1	4

Exercise: Insert the following numbers (89,18,49,58,29) into an array of size 10 using the following hashing function:

$h_i(x) = (x \bmod 10 + i*(7 - x \bmod 7)) \bmod TableSize$

0	
1	
2	
3	58
4	
5	29
6	49
7	
8	18
9	89

Comments on performance

- When the table is full, clusters are more likely and search gets slow.
- Load factor > 65%, linear probing becomes unacceptable
- Load factor > 75%, quadratic probing becomes unacceptable
- Load factor > 80%, double hashing becomes unacceptable
- So: hash table cannot be too full.

Expected number of probes

Load factor	failure	success
.1	1.11	1.06
.2	1.28	1.13
.3	1.52	1.21
.4	1.89	1.33
.5	2.5	1.50
.6	3.6	1.75
.7	6.0	2.17
.8	13.0	3.0
.9	50.5	5.5

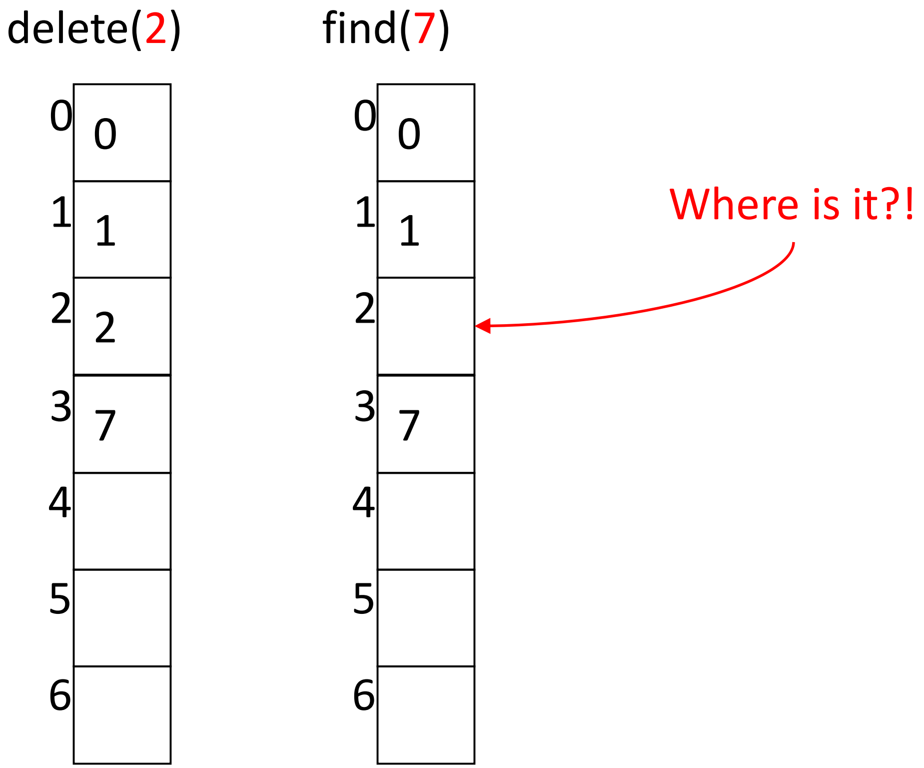
Deletion in Closed Hashing

- For chaining, it is simply removing an item.
- Can cause problem in probing.

– The probing may stop.

– To avoid, don't remove, but add a **marker**.

– Purge and remove after a while.



Lazy Deletion

- *Lazy deletion* (i.e. marking items as deleted)