

Recitation 7

Hashing

Sets

Sets

```

Set<E>
add(E e);
remove(Object o);
contains(Object o);
size();
    
```

Set: collection of *distinct* objects

Sets

How to implement a set?

Array List of *values*?

VA	NY	CA	
0	1	2	3

Method	Runtime
add	$O(n)$
contains	$O(n)$
remove	$O(n)$

Have to search through the list *linearly* to find the values

Have to shift all values down

Hashing 101

Hashing

Hashing — an implementation of a Set

Idea: finding an element in an array takes constant time when you know which index it is stored in

value

→

Hash Function

→

int

↓

b						
	0	1	2	3	4	5

Hashing

Hashing

Idea: finding an element in an array takes constant time when you know which index it is stored in

VA

→

Hash Function

→

5

add("VA")

↓

b

						VA
	0	1	2	3	4	5

Hashing

Hashing

Idea: finding an element in an array takes constant time when you know which index it is stored in

The diagram shows an oval labeled 'NY' with an arrow pointing to an oval labeled 'Hash Function'. From the 'Hash Function', an arrow points to a circle containing the number '3'. Below this, a horizontal array labeled 'b' has six cells indexed 0 to 5. The cell at index 3 contains 'NY' and the cell at index 5 contains 'VA'. An arrow points from the '3' in the circle to the 'NY' in the array cell at index 3. The text 'add("NY")' is written to the right of the 'Hash Function' oval.

Hashing

Load factor: b's saturation

Load factor: $\lambda = \frac{\# \text{ of entries}}{\text{length of array}} = \frac{3}{6}$

The diagram shows an oval labeled 'MA' with an arrow pointing to an oval labeled 'Hash Function'. From the 'Hash Function', an arrow points to a circle containing the number '0'. Below this, a horizontal array labeled 'b' has six cells indexed 0 to 5. The cell at index 0 contains 'MA', the cell at index 3 contains 'NY', and the cell at index 5 contains 'VA'. An arrow points from the '0' in the circle to the 'MA' in the array cell at index 0. The text 'add("MA")' is written to the right of the 'Hash Function' oval.

Hashing

We can hash any type of object!

Every object in Java has this method.
Default behavior is its object's memory address.

```

class Point {
    int x;
    int y;
    int hashCode() {
        return x + y;
    }
}
    
```

Hashing

Remainder Operator!

What if hashCode returns an int out of the array's bounds?

```

int hashInBounds(Object val) {
    return Math.abs(val.hashCode() % b.length);
}
    
```

For all operations, start by hashing to a valid index

Hashing

Basic set operations with hashing

```

add(val) {
    b[hashInBounds(val)] = val;
}
remove(val) {
    b[hashInBounds(val)] = null;
}
contains(val) {
    return b[hashInBounds(val)] != null;
}
    
```

Note: these are *very* simplified versions!

Operations take time proportional to hash function. Constant with respect to size of the array!

Collisions are a big problem: 2 vals hash to same index!

Collision Resolution

Collision Resolution

Problem: Collisions

```

class Point {
    int x;
    int y;
    int hashCode() {
        return x + y;
    }
}
    
```

Point p1 = new Point(1, 2);
 Point p2 = new Point(2, 1);

The diagram illustrates a collision. Two points, p1 (1, 2) and p2 (2, 1), are processed by separate Hash Function boxes. Both functions output the value 3. Below, a hash table array with indices 0 through 5 is shown. An arrow from the value 3 points to index 3 in the array, indicating that both p1 and p2 would map to the same location.

Collision Resolution

Solution 1: Perfect hash function

Map each value to a different index in the hash table

Impossible in practice

- don't know the size of the array
- Number of possible values far far exceeds the array size
- no point in a perfect hash function if it takes O(n) to compute

Collision Resolution

Solution 2: Collision resolution

Two ways of handling collisions:

1. Chaining
2. Open Addressing

The diagram shows two methods for collision resolution. On the left, 'Chaining' is illustrated with a vertical array of buckets. The second bucket has a pointer to a linked list of three nodes. On the right, 'Open Addressing' is shown with a vertical array of buckets. The second bucket has a circular arrow around it, indicating a search for the next available slot.

Collisions: Chaining

Collisions: Chaining

Chaining example

NY → Hash Function → 3 add("NY")

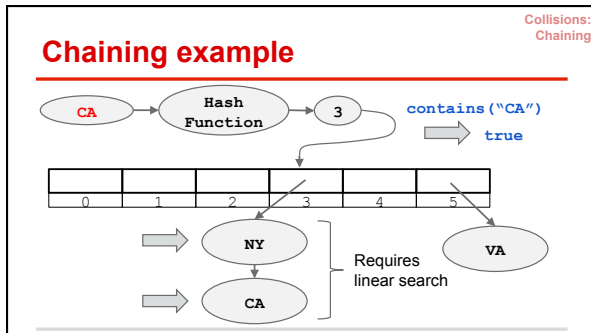
The diagram shows a hash table with indices 0-5. Index 3 contains 'VA'. A new node 'NY' is added to a linked list starting at index 3. The Hash Function for 'NY' outputs 3, which points to the bucket containing 'VA'. The bucket at index 3 now has a pointer to a linked list containing 'NY'.

Collisions: Chaining

Chaining example

CA → Hash Function → 3 add("CA")

The diagram shows a hash table with indices 0-5. Index 3 contains 'VA'. A new node 'CA' is added to a linked list starting at index 3. The Hash Function for 'CA' outputs 3, which points to the bucket containing 'VA'. The bucket at index 3 now has a pointer to a linked list containing 'NY' and 'CA'. A bracket on the left labels this linked list as 'bucket/chain (linked list)'.



Collisions:
Chaining

Inner class HashEntry

```

class HashSet<V> {
    LinkedList<HashEntry<V>>[] b;

    private class HashEntry<V> {
        V value;
    }
}
    
```

inner class to store value

Collisions:
Chaining

Set operations

For `add`, `contains`, `remove` always start by finding correct bucket:

- `b[hashInBounds(value)]`

`add(value)`

1. If bucket already contains value, do nothing
2. Else add new `HashEntry` to bucket

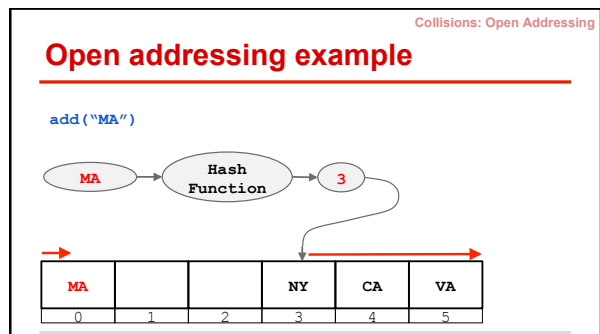
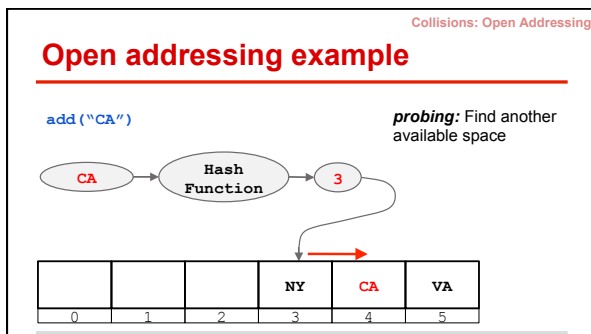
`contains(value)`

1. If bucket contains value, return true
2. Else return false

`remove(value)`

1. If bucket contains value, remove entry from list

Collisions: Open Addressing



Collisions: Open Addressing

Open addressing example

contains("SC")

How far do we search? Once we reach an empty (null) cell, we know it's not there.

MA	⊗		NY	CA	VA
0	1	2	3	4	5

Collisions: Open Addressing

Finding where a key belongs

Keep searching until we hit null or we find the value in question

```
int getPosition(val) {
    int i = hashInBounds(val);
    while (b[i] != null && !val.equals(b[i].val)) {
        i = (i+1) % b.length;
    }
    return i;
}
```

linear probing - searching the array in order: i, i+1, i+2, i+3...

Collisions: Open Addressing

Efficiency of linear probing

Average number of probes

$$= \frac{1}{1-\lambda} = \frac{1}{1-\frac{\# \text{ of entries}}{\text{length of array}}} = \frac{1}{1-\frac{\text{null entries}}{\text{length of array}}} = \frac{\text{length of array}}{\text{null entries}}$$

Array half full? add(value) expected to need only 2 probes! Wow! Beats linear search!

Collisions: Open Addressing

Deleting elements

contains("MA") ⇒ false

What happens if we remove VA and then try to lookup MA?

MA			NY	CA	⊗
0	1	2	3	4	5

Collisions: Open Addressing

Deleting elements

contains("MA") ⇒ true

Solution: The VA entry is still there, but marked as removed

MA	⊗			NY	CA	⊗
0	1	2	3	4	5	6

Collisions: Open Addressing

Deleting elements

```
class HashSet<V> {
    HashEntry<V>[] b;

    private class HashEntry<V> {
        V value;
        boolean isInSet = true;
    }
}
```

Set isInSet to false to remove it

Collisions: Open Addressing

Set operations

For `add`, `contains`, `remove`, always start by finding correct index using probing: `pos = getPosition(key)`

```

add(value)
    1. If b[pos] is null, add new HashEntry at pos
    2. Else mark isInSet as true
contains(value)
    1. Return b[pos] != null && b[pos].isInSet
remove(value)
    1. If b[pos] is not null and isInSet is true,
       mark isInSet as false
    
```

Collisions: Open Addressing

Linear vs quadratic probing

When a collision occurs, how do we search for an empty space?

linear probing: search the array in order: $i, i+1, i+2, i+3 \dots$	quadratic probing: search the array in nonlinear sequence: $i, i+1^2, i+2^2, i+3^2 \dots$	clustering: problem where nearby hashes have very similar probe sequence so we get more collisions
--	--	--

Collisions

Collision resolution summary

Open Addressing <ul style="list-style-type: none"> store all entries in table use linear or quadratic probing to place items uses less memory clustering can be a problem - need to be more careful with choice of hash function 	Chaining <ul style="list-style-type: none"> store entries in separate chains (linked lists) can have higher load factor/degrades gracefully as load factor increases
---	---

Rehashing

Rehashing

Resizing

What happens as the array becomes too full?
i.e. load factor gets a lot bigger than $\frac{1}{2}$?

$O(1) \rightarrow O(n)$ operations

Solution: **Dynamic resizing**

- reinsert / **rehash** all elements to an array *double* the size.
 - Now is the time where we remove the entries where `!b[pos].isInSet`
- Why not** simply copy into first half?

Rehashing

Load factor

Load factor $\lambda = \frac{\# \text{ of entries}}{\text{length of array}}$

Rehashing happens when λ reaches **load factor threshold**

Big O!

Big O of Hashing

Runtime analysis

	Chaining	Open Addressing
Expected	$O(\text{hash function}) + O(\text{load factor})$	$O(\text{hash function}) + O\left(\frac{\text{length of array}}{\# \text{ of null slots}}\right)$
Worst	$O(n)$ (all elements in one bucket)	$O(n)$ (array almost full)

Big O of Hashing

Amortized runtime

Insert n items: $n + 2n$ (from copying) = $3n$ inserts $\rightarrow O(3n) \rightarrow O(n)$
 Amortized to constant time per insert

	Copying Work
Everything has just been copied	n inserts
Half were copied in previous doubling	$n/2$ inserts
Half of those were copied in doubling before previous one	$n/4$ inserts
...	...
Total work	$n + n/2 + n/4 + \dots \leq 2n$

Hash Functions

- Hash Functions
- ### Requirements
- Hash functions MUST:
- have the same hash for two equal objects
 - In Java: if `a.equals(b)`, then `a.hashCode() == b.hashCode()`
 - if you override equals and plan on using object in a HashMap or HashSet, override hashCode too!
 - be deterministic
 - calling hashCode on the same object should return the same integer
 - important to have immutable values if you override equals!

- Hash Functions
- ### Good hash functions
- As often as possible, if `!a.equals(b)`, then `a.hashCode() != b.hashCode()`
 - this helps avoid collisions and clustering
 - Good distribution of hash values across all possible keys
 - FAST. add, contains, and remove are proportional to speed of hash function
- A bad hash function won't break a hash set but it could seriously slow it down

String.hashCode()

Don't hash very long strings, not $O(1)$ but $O(\text{length of string})!$

```
/** Returns a hash code for this string.
 * Computes it as
 *  $s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$ 
 * using int arithmetic.
 */
public int hashCode() { ... }
```

Designing good hash functions

```
class Thingy {
    private String s1, s2;

    public boolean equals(Object obj) {
        return s1.equals(obj.s1)
            && s2.equals(obj.s2);
    }

    public int hashCode() {
        return 37 * s1.hashCode() + 97 * s2.hashCode();
    }
}
```

Limitations of hash sets

1. Due to rehashing, adding elements will sometimes take $O(n)$
 - a. not always ideal for time-critical applications
2. No ordering among elements, very slow to find nearby elements

Alternatives (out of scope of the course):

1. hash set with incremental resizing prevents $O(n)$ rehashing
2. self-balancing binary search trees are worst case $O(\log n)$ and keep the elements ordered

Hashing Extras

Hashing has wide applications in areas such as security

- cryptographic hash functions are ones that are very hard to invert (figure out original data from hash code), changing the data almost always changes the hash, and two objects almost always have different hashes
- md5 hash: `md5 filename` in Terminal

By	Size	MD5
erc73	1.449 MB	13188ff26829b7cc4f4a45b84ee6deb7