Recitation 7

Hashing

Sets

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

Set: collection of distinct objects

How to implement a set?

Array List of values?

<table>
<thead>
<tr>
<th>VA</th>
<th>NY</th>
<th>CA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

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

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

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

Hashing

add("VA")
Idea: finding an element in an array takes constant time when you know which index it is stored in.

**Load factor:** b’s saturation
\[ \lambda = \frac{\text{# of entries}}{\text{length of array}} = \frac{3}{5} \]

We can hash any type of object!

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

Default behavior is its object’s memory address.

**Remainder Operator!**

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

For all operations, start by hashing to a valid index.

Basic set operations with hashing

```java
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!
**Problem: Collisions**

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

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

**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

**Solution 2: Collision resolution**

Two ways of handling collisions:

1. Chaining
2. Open Addressing

**Collisions: Chaining**

**Chaining example**

```
NY
```

**Chaining example**

```
CA
```

bucket/chain (linked list)
**Chaining example**

Collisions: Chaining

```plaintext
contains("CA")
true

Requires linear search
```

**Inner class HashEntry**

Collisions: Chaining

```java
class HashSet<V> {
    LinkedList<HashEntry<V>>[] b;
    private class HashEntry<V> {
        V value;
    }
}
```

inner class to store value

**Set operations**

For add, contains, remove always start by finding correct bucket:
- \( b[\text{hashInBounds(value)}] \)

- **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**

**Open addressing example**

**probing:** Find another available space

add("CA")

```
<table>
<thead>
<tr>
<th></th>
<th>NY</th>
<th>CA</th>
<th>VA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

add("MA")
Open addressing example

contains("SC")

SC

Hash Function

3

MA NY CA VA
0 1 2 3 4 5

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

Finding where a key belongs

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

Efficiency of linear probing

Average number of probes

\[ \frac{1}{1-\frac{1}{2 \text{ of array length}}} \approx \frac{1}{1-\frac{1}{\text{array length}}} = \frac{1}{\text{array length} - \text{null entries}} \]

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

Deleting elements

contains("MA") \iffalse

MA NY CA VA
0 1 2 4

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

class HashSet<V> {
    HashEntry<V>[] b;
    private class HashEntry<V> {
        V value;
        boolean isInSet= true;
    }
}

Deleting elements

contains("MA") \iffalse

Set isInSet to false to remove it

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

For **add**, **contains**, **remove**, always start by finding correct index using probing: \( \text{pos} = \text{getPosition(key)} \)

**add(value)**
1. If \( b[\text{pos}] \) is null, add new HashEntry at pos
2. Else mark isInSet as true

**contains(value)**
1. Return \( b[\text{pos}] \neq \text{null} \) && \( b[\text{pos}].\text{isInSet} \)

**remove(value)**
1. If \( b[\text{pos}] \) 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: \( i, i+1, i+2, i+3 \ldots \)
- **quadratic probing**: search the array in nonlinear sequence: \( i, i+1^2, i+2^2, i+3^2 \ldots \)

Collision resolution summary

**Open Addressing**
- 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**
- store entries in separate chains (linked lists)
- can have higher load factor/degrades gracefully as load factor increases

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?

Load factor

\[ \lambda = \frac{\# \text{ of entries}}{\text{length of array}} \]

Rehashing happens when \( \lambda \) reaches load factor threshold

- best range
- too slow

Load factor threshold

- waste of memory
Big O!

Runtime analysis

<table>
<thead>
<tr>
<th></th>
<th>Chaining</th>
<th>Open Addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>$O(\text{hash function}) + O(\text{load factor})$</td>
<td>$O(\text{hash function}) + O\left(\frac{\text{length of array}}{\text{# of null slots}}\right)$</td>
</tr>
<tr>
<td>Worst</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>(all elements in one bucket)</td>
<td>(array almost full)</td>
</tr>
</tbody>
</table>

Amortized runtime

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

<table>
<thead>
<tr>
<th></th>
<th>Copying Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Everything has just been copied</td>
<td>$n$ inserts</td>
</tr>
<tr>
<td>Half were copied in previous doubling</td>
<td>$n/2$ inserts</td>
</tr>
<tr>
<td>Half of those were copied in doubling before previous one</td>
<td>$n/4$ inserts</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Total work</td>
<td>$n + n/2 + n/4 + ... \leq 2n$</td>
</tr>
</tbody>
</table>

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!

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
Don’t hash very long strings, not O(1) but O(length of string)!

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

Designing good hash functions

```java
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