Week 11: Hashing

Thursday, November 19, 2020 11:23 PM

Hashtables

- keys/elements
- array

→ lookup, add, remove
→ O(1)

Hash Code

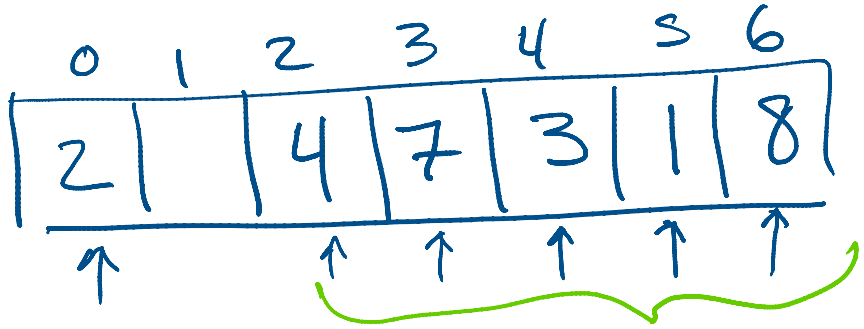
- every object has a hash code
- ideally unique identifier
  - an object should have same hash code throughout its life
- hashCodes are calculated using fields of an object
  - * objects should keep track of their hash ID
  - * need to be reliable
  - * objects that are equal will have same hashcode

* useful to find index in an array
* → if unique: O(1) search.

* → if unique: O(1) search, add, remove

Open-Address Hashing

→ just involves array
→ collisions: two objects have same hashed index



*what do we do?

→ linear probing: go 1 by 1 to find empty index
  → causes clustering
    * O(n) → 1,000,000
    1

→ double hashing:
  → calculate a second hash value to add to index

hash value ?
repeatedly add to index
until we find an
open spot
*avoids clustering

object.hashCode()
% array len
↖ [0, arraylen)

int hash(key){} -> return original
int hash2(key){} -> return diff value

↙
[1, array length - 2]

*array length must be prime !
larger of twin primes

(5, 7) or (11, 13) *Donald Knuth

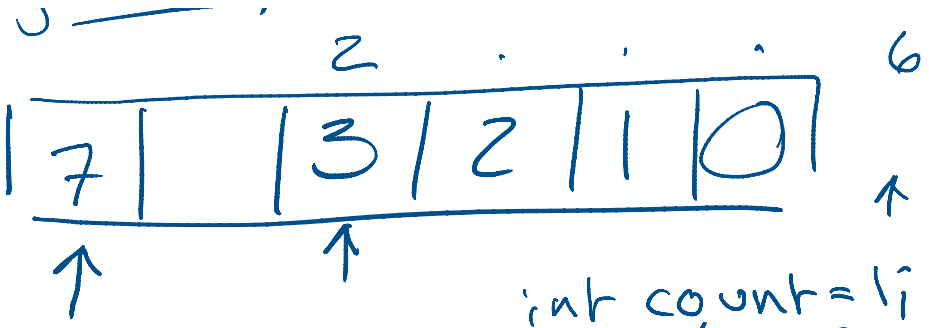
⇒quadratic probing:
$h+1, h+2^2, h+3^2, \ldots$
$h+k^2$

Original hash index

*avoids clustering
* O(1) maintained

ger Index / add

2 . . . 6



index: 2
index: 3
index: 4
index: 5

```
int count = 1;
While (arr[index] is not empty)
    index = hash + count
    *make sure valid/
      circle around
```

```
int count = 1
while (arr[index] is not
      empty) {
    index = hash + Math.pow(count, 2)
    ++count;
    *make sure index is valid
}
```

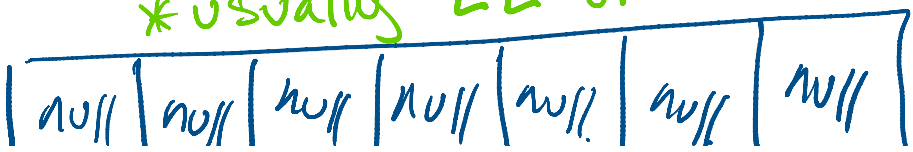
Chained/Bucket Hashing

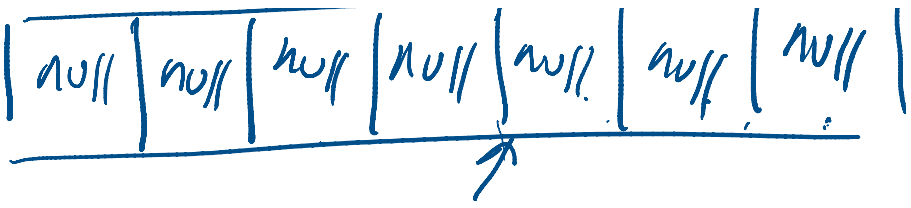
→ if there is a collision, just add element to a "bucket"

*no probing

→ buckets: dynamic arrays in hw

*usually LL or balanced BST





add: 7; index: 4

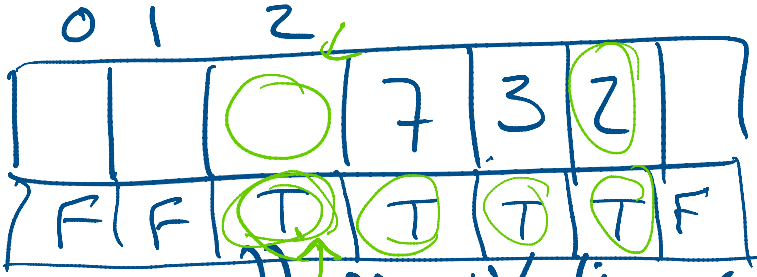
arr[index] = new ArrayList()
arr[index].add(7)

*buckets are not sorted
* ops take O(n) time

LL/DA: O(n)
BST: O(log n)

LL: doesn't waste space
DA: wastes a lot of space

Removal w/ Open-Address



→open-address w/ linear probing

→remove 5.

*all existing elements hashed to 2

→delete 5

→ does hashtable contain 2?

→ does hash table contain 2?
→ no

→ need to keep track of whether there was an element @ each index for removal
→ "zombie" placeholder
→ sentinel value
→ special object
* place when removing
→ boolean array w/ same length as hash table
→ when adding: set same index to T

*** if hash table is too full, runtime slows down
→ load factor: 70% or 3/4 of array length
* → if # elements reaches load factor, resize

rehash

* essential for efficiency:
  - -> good hash codes *
  - -> good collision resolution
  - -> keep low # of elements compared to array size