HASHTABLES 2

David Kauchak
CS 62 – Spring 2021

1

Admin

Lab tomorrow

- Midterm recap (save questions for then)
- Course feedback discussion
- Start next assignment (2 week assignment)

Quiz on Thursday

2

Sets

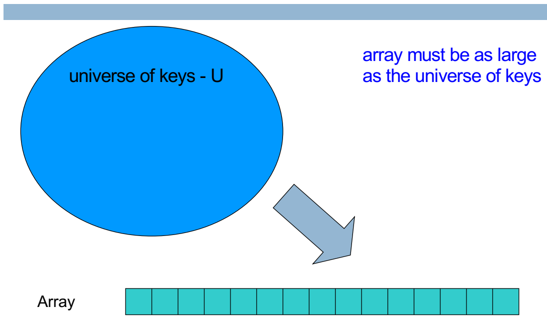
An unordered collection

- Things can be added and removes
- Check if things are in the set

```
public interface Set<E> {
    public void put(E key);
    public boolean containsKey(E key);
    public E remove(E key);
    public boolean isEmpty();
    public int size();
}
```

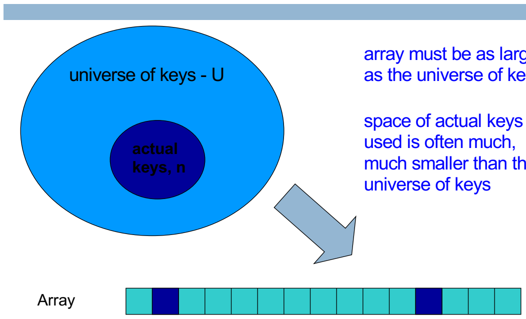
3

Why not just arrays?



4

Why not just arrays?



array must be as large as the universe of keys

space of actual keys used is often much, much smaller than the universe of keys

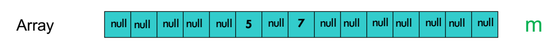
5

Hashtables

```
public interface Set<E> {
    public void put(E key);
    public boolean containsKey(E key);
    public E remove(E key);
    public boolean isEmpty();
    public int size();
}
```

Using an array is still a good idea

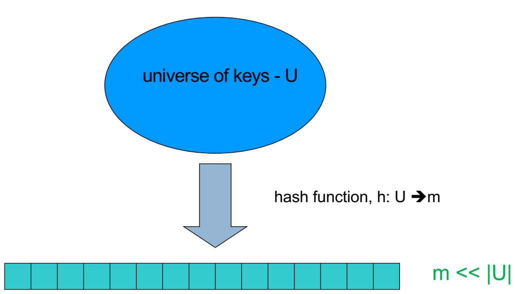
Key idea: need to translate from the key into an index in the array



6

Hash function, *h*

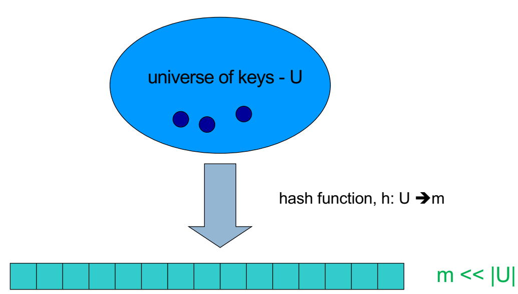
A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array)



7

Hash function, *h*

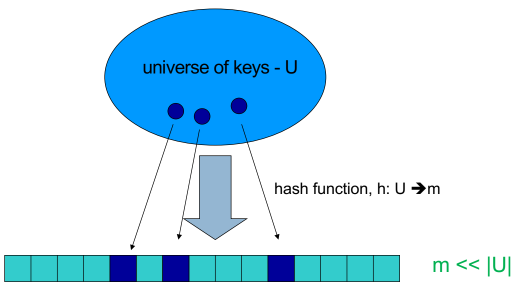
A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array)



8

Hash function, *h*

A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array)



9

Collisions

A collision occurs when h(x) = h(y), but x ≠ y

A good hash function will minimize the number of collisions

Because the number of hashtable (array) entries is less than the possible keys (i.e. m < |U|) collisions are inevitable!

We need to handle collisions!

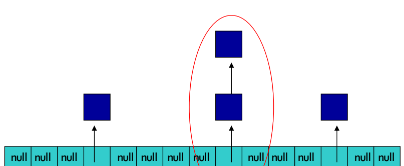
Collision resolution techniques?

10

Collision resolution by chaining

Hashtable consists of an array of linked lists

```
private LinkedList<E>[] table;
```

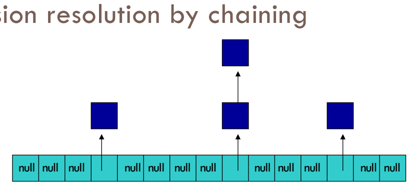


When a collision occurs, the element is added to linked list at that location

If two entries x ≠ y have the same hash value h(x) = h(y), then table[h(x)] will contain a linked list with both values

11

Collision resolution by chaining



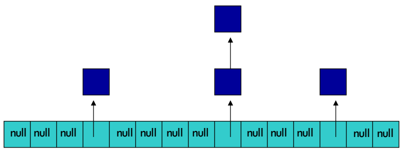
put?

containsKey?

remove

12

Collision resolution by chaining



put: addFirst h(key)

containsKey: contains h(key)

remove: remove h(key)

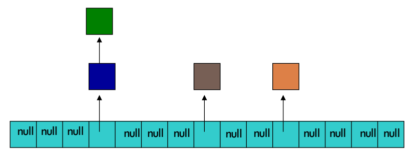
13

Running time?

put: addFirst h(key)

containsKey: contains h(key)

remove: remove h(key)



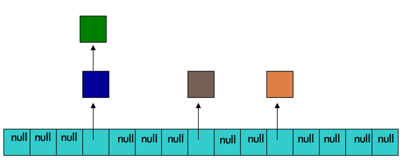
14

Running time?

put: O(1)

containsKey: O(length of linked list)

remove: O(length of linked list)



15

Length of the chain

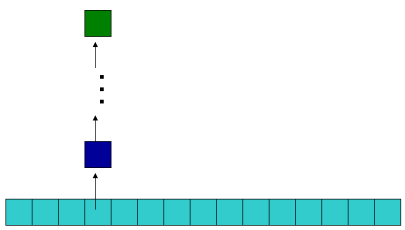
Worst case?

16

Length of the chain

Worst case?

- All elements hash to the same location
- h(k) = 4
- **n**



17

Length of the chain

Average case:

Depends on how well the hash function distributes the keys

What is the best we could hope for a hash function?

- simple uniform hashing: an element is equally likely to end up in any of the *m* slots

Under simple uniform hashing what is the average length of a chain in the table?

- *n* keys over *m* slots = $n / m = \alpha$

18

The load of a table/hashtable

m = number of possible entries in the table

n = number of keys stored in the table

$\alpha = n/m$ is the **load factor** of the hashtable

The smaller α, the more wasteful the table

The load also helps us talk about run time

19

Average chain length

If you roll a fair *m* sided die *n* times, how many times are we likely to see a given value?

For example, 10 sided die:

1 time
- 1/10

100 times
- 100/10 = 10

20

containsKey average running time

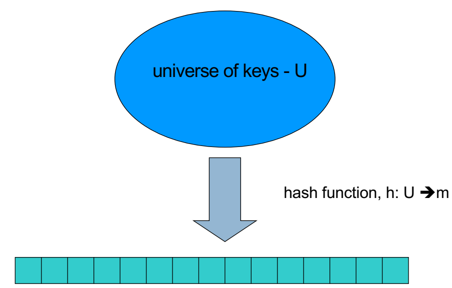
Two cases:

- Key is **not** in the table
  - must search all entries
  - O(1 + α)
- Key **is** in the table
  - on average search half of the entries
  - O(1 + α)

21

Hash functions

Function that takes as input a key and return a value from 0 to m-1 (the size of the hashtable)



22

Hash functions

What makes a good hash function?

- Approximates the assumption of simple uniform hashing
- Deterministic – h(x) should always return the same value
- Low cost – if it is expensive to calculate the hash value (e.g. log n) then we don't gain anything by using a table

Challenge: we don't generally know the distribution of the keys

- Frequently data tend to be clustered (e.g. similar strings, run-times, SSNs). A good hash function should spread these out across the table

23

Division method

h(k) = k mod m

m	k	h(k)
11	25	
11	1	
11	17	
13	133	
13	7	
13	25	

24

Division method

h(k) = k mod m

m	k	h(k)
11	25	3
11	1	1
11	17	6
13	133	3
13	7	7
13	25	12

25

Division method

Don't use a power of two. Why?

m	k	bin(k)	h(k)
8	25	11001	
8	1	00001	
8	17	10001	

26

Division method

Don't use a power of two. Why?

m	k	bin(k)	h(k)
8	25	11001	1
8	1	00001	1
8	17	10001	1

if h(k) = k mod 2^p, the hash function is just the lower *p* bits of the value

27

Division method

Good rule of thumb for *m* is a prime number not too close to a power of 2

Pros:

- quick to calculate
- easy to understand

Cons:

- keys close to each other will end up close in the hashtable

28

Multiplication method

Multiply the key by a constant 0 < A < 1 and extract the fractional part of *kA*, then scale by *m* to get the index

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

extracts the fractional portion of *kA*

29

Multiplication method

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

Common choice is for *m* as a power of 2 and

$$A = (\sqrt{5} - 1)/2 = 0.6180339887$$

Why a power of 2?

Book has other heuristics

30

Multiplication method

m	k	A	kA	h(k)
8	15	0.618		
8	23	0.618		
8	100	0.618		

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

31

Multiplication method

m	k	A	kA	h(k)
8	15	0.618	9.27	floor(0.27*8) = 2
8	23	0.618	14.214	floor(0.214*8) = 1
8	100	0.618	61.8	floor(0.8*8) = 6

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

32

Other hash functions

http://en.wikipedia.org/wiki/List_of_hash_functions

cyclic redundancy checks (i.e. disks, cds, dvds)

Checksums (i.e. networking, file transfers)

Cryptographic (i.e. MD5, SHA)

33

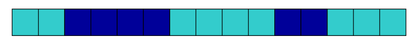
Open addressing

Keeping around an array of linked lists can be inefficient and a hassle

Like to keep the hashtable as just an array of elements (no pointers)

How do we deal with collisions?

- compute another slot in the hashtable to examine



34

Hash functions with open addressing

Hash function must define a **probe sequence** which is the list of slots to examine when a put or containsKey

The hash function takes an additional parameter *i* which is the number of collisions that have already occurred

The probe sequence **must** be a permutation of every hashtable entry. Why?

{ h(k,0), h(k,1), h(k,2), …, h(k, m-1) } is a permutation of { 0, 1, 2, 3, …, m-1 }

35

Hash functions with open addressing

Hash function must define a **probe sequence** which is the list of slots to examine when doing a put or containsKey

The hash function takes an additional parameter *i* which is the number of collisions that have already occurred

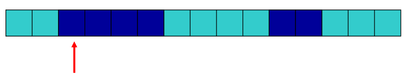
The probe sequence **must** be a permutation of every hashtable entry. Why?

If not, we wouldn't explore all the possible location in the table!

36

Probe sequence

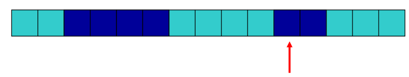
h(k, 0)



37

Probe sequence

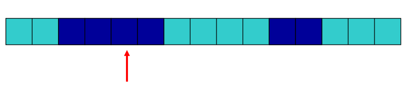
h(k, 1)



38

Probe sequence

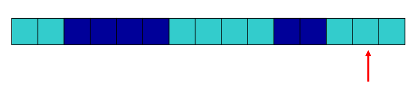
h(k, 2)



39

Probe sequence

h(k, 3)

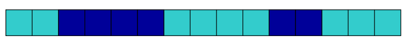


40

Probe sequence

h(k, …)

must visit all locations



…

41

Open addressing: put

```
public void put(E key) {
    int i = 0;
    int next = probeSequence(key, i);

    while( i < table.length &&
           table[next] != null ){
        i++;
        next = probeSequence(key, i);
    }

    table[next] = key;
    count++;
}
```

What does this code do?

42

Open addressing: put

```
public void put(E key) {
    int i = 0;
    int next = probeSequence(key, i);

    while( i < table.length &&
           table[next] != null ){
        i++;
        next = probeSequence(key, i);
    }

    table[next] = key;
    count++;
}
```

get the first entry to check

43

Open addressing: put

```
public void put(E key) {
    int i = 0;
    int next = probeSequence(key, i);

    while( i < table.length &&
           table[next] != null ){
        i++;
        next = probeSequence(key, i);
    }

    table[next] = key;
    count++;
}
```

as long as we haven't check all entries and the entry isn't empty

44

Open addressing: put

```
public void put(E key) {
    int i = 0;
    int next = probeSequence(key, i);

    while( i < table.length &&
           table[next] != null ){
        i++;
        next = probeSequence(key, i);
    }

    table[next] = key;
    count++;
}
```

get the next entry to check

45

Open addressing: put

```
public void put(E key) {
    int i = 0;
    int next = probeSequence(key, i);

    while( i < table.length &&
           table[next] != null ){
        i++;
        next = probeSequence(key, i);
    }

    table[next] = key;
    count++;
}
```

put the key into the table
(assumes table wasn't full)

46

Open addressing

containsKey?

remove?

47

Open addressing: containsKey

```
public boolean containsKey(E key){
    int i = 0;
    int next = probeSequence(key, i);

    while( i < table.length &&
           table[next] != null &&
           !table[next].equals(key)){
        i++;
        next = probeSequence(key, i);
    }

    // only 3 ways to exit the while loop
    // the two of which below mean we didn't find it
    return !(i == table.length || table[next] == null);
}
```

48

Open addressing: containsKey

```
public boolean containsKey(E key){
    int i = 0;
    int next = probeSequence(key, i);

    while( i < table.length &&
           table[next] != null &&
           !table[next].equals(key)){
        i++;
        next = probeSequence(key, i);
    }

    // only 3 ways to exit the while loop
    // the two of which below mean we didn't find it
    return !(i == table.length || table[next] == null);
}
```

very similar to put!

also need to check if we've found the key

49

Open addressing: containsKey

```
public boolean containsKey(E key){
    int i = 0;
    int next = probeSequence(key, i);

    while( i < table.length &&
           table[next] != null &&
           !table[next].equals(key)){
        i++;
        next = probeSequence(key, i);
    }

    // only 3 ways to exit the while loop
    // the two of which below mean we didn't find it
    return !(i == table.length || table[next] == null);
}
```

return false if we searched the whole table or
we got to a null entry

50

Open addressing: remove

Two options:

- mark node as "deleted" (rather than null)
  - modify containsKey to continue looking if a "deleted" node is seen
  - modify put to fill in "deleted" entries
  - increases search times!
- if a lot of deleting will happen, use chaining

51

Probing schemes

Linear probing – if a collision occurs, go to the next slot

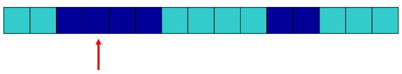
- h(k,i) = (h(k) + i) mod m
- Does it meet our requirement that it visits every slot?
- for example, m = 7 and h(k) = 4

h(k,0) = 4
h(k,1) = 5
h(k,2) = 6
h(k,3) = 0
h(k,3) = 1

52

Linear probing: put

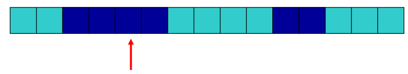
h(, 0)



53

Linear probing: put

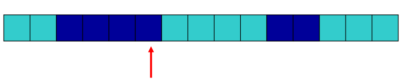
h(, 1)



54

Linear probing: put

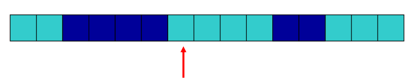
h(, 2)



55

Linear probing: put

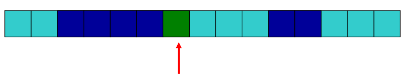
h(, 3)



56

Linear probing: put

h(, 3)

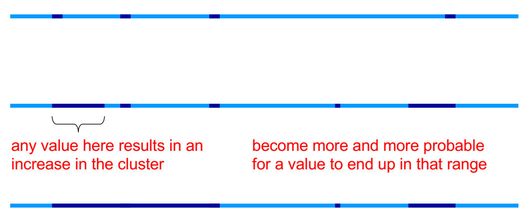


57

Linear probing

Problem:

primary clustering – long runs of occupied slots tend to build up and these tend to grow



58

Quadratic probing

$h(k,i) = (h(k) + c_1 i + c_2 i^2) \bmod m$

Rather than a linear sequence, we probe based on a quadratic function

Problems:

- must pick constants and *m* so that we have a proper probe sequence
- if $h(x) = h(y)$, then $h(x,i) = h(y,i)$ for all i
- secondary clustering

59

Double hashing

Probe sequence is determined by a second hash function

$h(k,i) = (h_1(k) + i(h_2(k))) \bmod m$

Problem:

- $h_2(k)$ must visit all possible positions in the table

60

Running time of put and containsKey for open addressing

Depends on the hash function/probe sequence

Worst case?

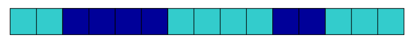
O(n) – probe sequence visits every full entry first before finding an empty

61

Running time of put and containsKey for open addressing

Average case?

We have to make at least one probe



62

Running time of put and containsKey for open addressing

Average case?

What is the probability that the first probe will **not** be successful (assume uniform hashing function)?

α

63

Running time of put and containsKey for open addressing

Average case?

What is the probability that the first **two** probed slots will **not** be successful?

why '~' ?

$\sim\alpha^2$

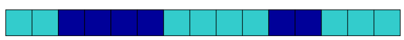
64

Running time of put and containsKey for open addressing

Average case?

What is the probability that the first **two** probed slots will **not** be successful?

Technically, second probe is: $\frac{n-1}{m-1}$ $\sim\alpha^2$



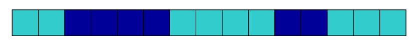
65

Running time of put and containsKey for open addressing

Average case?

What is the probability that the first **three** probed slots will **not** be successful?

$\sim\alpha^3$



66

Running time of insert and search for open addressing

Average case: expected number of probes

sum of the probability of making 1 probe, 2 probes, 3 probes, …

$$E[probes] = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$$

$$= \sum_{i=0}^{m} \alpha^i$$

$$< \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1-\alpha}$$

67

Average number of probes

$$E[probes] = \frac{1}{1-\alpha}$$

α	Average number of searches
0.1	$1/(1-.1) = 1.11$
0.25	$1/(1-.25) = 1.33$
0.5	$1/(1-.5) = 2$
0.75	$1/(1-.75) = 4$
0.9	$1/(1-.9) = 10$
0.95	$1/(1-.95) = 20$
0.99	$1/(1-.99) = 100$

68

How big should a hashtable be?

A good rule of thumb is the hashtable should be around half full

What happens when the hashtable gets full?

- Copy: Create a new table and copy the values over
  - results in one expensive put
  - simple to implement

- Amortized copy: When a certain ratio is hit, grow the table, but copy the entries over a few at a time with every insert
  - no single put is expensive and can guarantee per put performance
  - more complicated to implement

69

To the code…

abstract classes!

Making your classes hashable:

- hashCode
- equals

HashSet:

https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

HashMap:
https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

70