

HASHTABLES 2

David Kauchak
CS 62 – Fall 2026

1

Admin

Assignment 6 out (OnDiskSort)

Drop **two** lowest quizzes

2

Sets

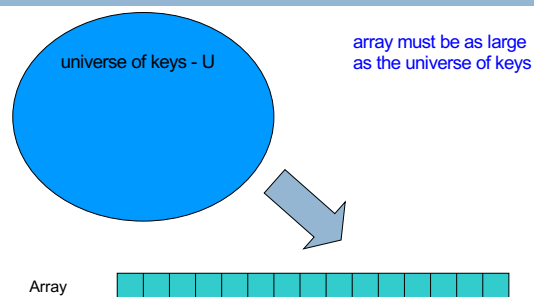
An unordered collection

- Things can be added and removed
- 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();
}
```

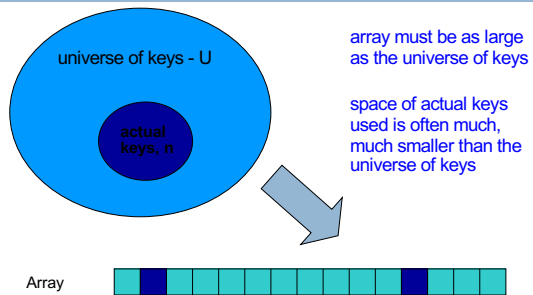
3

Why not just arrays?



4

Why not just arrays?



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

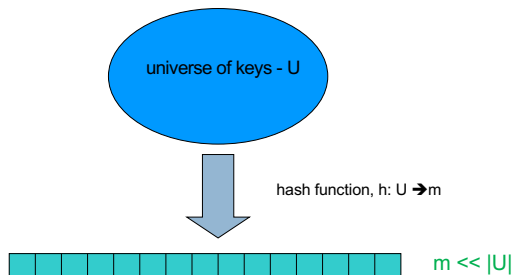
Array

null	null	null	null	5	7	null	null	null	null	null	null	null	null	m
------	------	------	------	---	---	------	------	------	------	------	------	------	------	---

6

Hash function, h

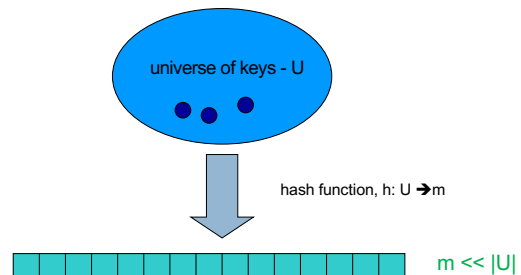
A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array)



7

Hash function, h

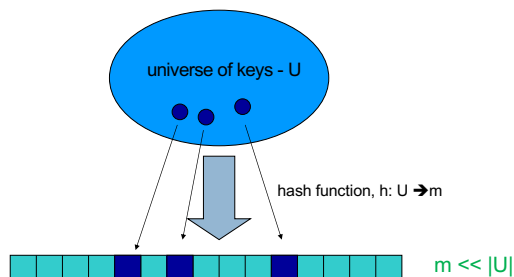
A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array)



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 $x \neq y$

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!

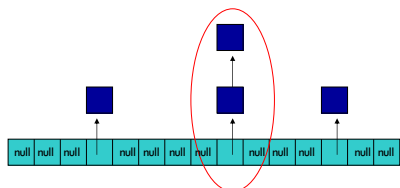
Collision resolution techniques?

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 $x \neq y$ have the same hash value $h(x) = h(y)$, then `table[h(x)]` will contain a linked list with both values

11

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



12

Hash functions with open addressing

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

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

$$h(\text{key}, i)$$

$h(k, 0)$: first place to look

$h(k, 1)$: if there is a collision in the first place, where to look next

$h(k, 2)$: If both first and second places are full, where to look next
...

13

Hash functions with open addressing

Hash function must define a **probe sequence** which is the list of slots to examine when calling 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?**

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

14

Hash functions with open addressing

Hash function must define a **probe sequence** which is the list of slots to examine when calling 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 locations in the table!

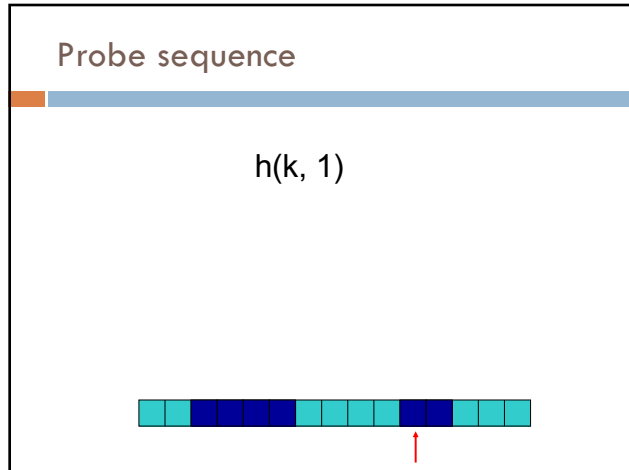
15

Probe sequence

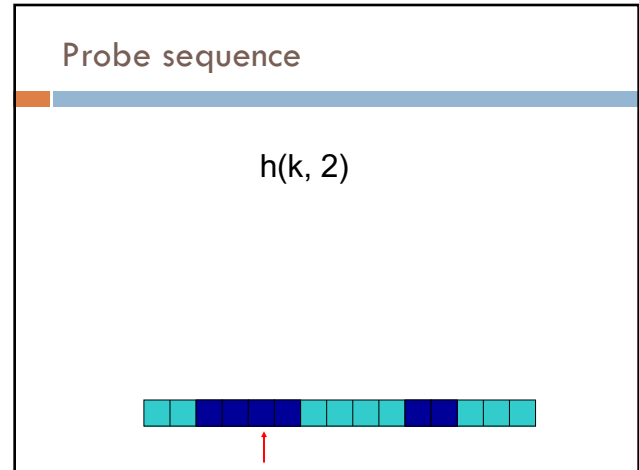
$$h(k, 0)$$



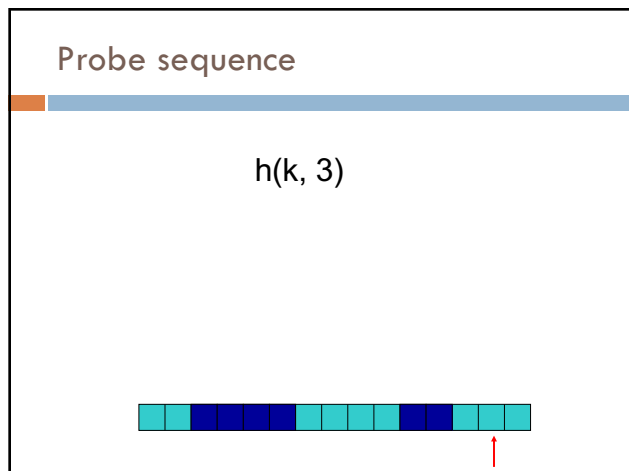
16



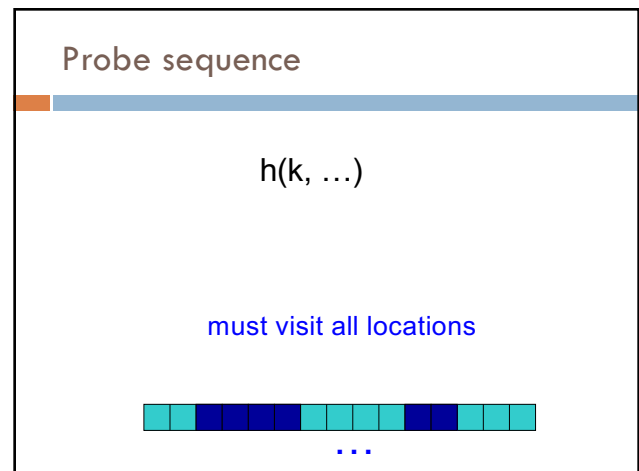
17



18



19



20

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?

21

Open addressing: put

```

public void put(E key) {
    int i = 0;
    int next = probeSequence(key, i);    get the first entry to check

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

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

```

22

Open addressing: put

```

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

    while( i < table.length &&
           table[next] != null ){    as long as we haven't
        i++;                          check all entries and the
        next = probeSequence(key, i); entry isn't empty
    }

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

```

23

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);    get the next entry to check
    }

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

```

24

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)

25

Open addressing

containsKey?

remove?

```
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++;
}
```

26

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);
}
```

27

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

28

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
```

29

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

30

Probing schemes

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

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

$$h(k,0) = 4$$

$$h(k, 1) = 5$$

$$h(k,2) = 6$$

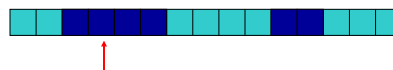
$$h(k,3) = 0$$

$$h(k,3) = 1$$

31

Linear probing: put

$$h(\blacksquare, 0)$$



32

Linear probing: put

h(■, 1)



33

Linear probing: put

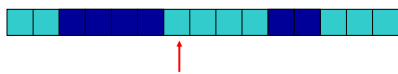
h(■, 2)



34

Linear probing: put

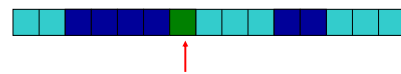
h(■, 3)



35

Linear probing: put

h(■, 3)

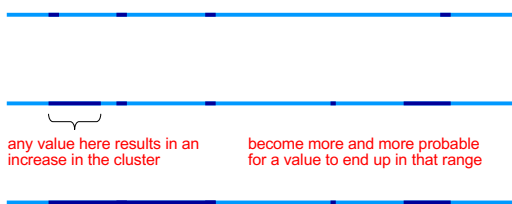


36

Linear probing

Problem:

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



37

Quadratic probing

$$h(k,i) = (h(k) + c_1i + c_2i^2) \bmod m$$

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

Problems/challenges:

- 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

38

Double hashing

Probe sequence is determined by a second hash function

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

Problems/challenges:

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

39

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

40

Running time of put and containsKey for open addressing

Average case?

We have to make at least one probe



41

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)?

$$n/m = \alpha$$



42

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
'~'? $\sim \alpha^2$



43

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: $\frac{n-1}{m-1} \sim \alpha^2$



44

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?

$$\sim \alpha^3$$



45

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, ...

$$\begin{aligned} E[\text{probes}] &= 1 + \alpha + \alpha^2 + \alpha^3 + \dots \\ &= \sum_{i=0}^{\infty} \alpha^i \\ &< \sum_{i=0}^{\infty} \alpha^i \\ &= \frac{1}{1 - \alpha} \end{aligned}$$

46

Average number of probes

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

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	$1/(1 - .99) = 100$

47

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

48

Java implementations

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>

49

To the code... Abstract classes

A blend of a traditional class and an interface

Some method implementations are defined

Others, have only the method header defined (called "abstract" methods)

Cannot instantiate an abstract class: must inherit from them and implement the abstract methods

For example: <https://github.com/pomonaCS62-fa2026/LectureCode/blob/main/Hashtables/OpenAddressedHashTable.java>

50

Abstract classes in Java already

AbstractList:

<https://docs.oracle.com/javase/8/docs/api/java/util/AbstractList.html>

AbstractSet:

<https://docs.oracle.com/javase/8/docs/api/java/util/AbstractSet.html>

51

To the code...

<https://github.com/pomonaCS62-fa2026/LectureCode/tree/main/Hashtables>

52