

HASHTABLES

David Kauchak
CS 62 – Fall 2026

1

Admin

No mentor/office hours over fall break

Wordle due Friday @ 5pm

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();
}
```

Could we do this with any of our data structures so far?
Big-O runtime of methods?

3

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();
}
```

We'd like to make these operations fast: $O(1)$!

4

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

What if we assume that they're integers.

Any ideas how to do this fast?

5

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

Array

null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null
------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

6

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

put(5)

Where should we put it?

Array

null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null
------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

7

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

put(5)

Array

null	null	null	null	null	5	null	null	null	null	null	null	null	null	null	null	null	null	null	null
------	------	------	------	------	---	------	------	------	------	------	------	------	------	------	------	------	------	------	------

8

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

put(7) Where should we put it?

Array

null	null	null	null	null	5	null	null	null	null	null	null	null	null	null
------	------	------	------	------	---	------	------	------	------	------	------	------	------	------

9

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

put(7) Where should we put it?

Array

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

10

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

containsKey(4)?

Array

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

11

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

containsKey(4)? No!

Array

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

12

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

containsKey(7)?

Array

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

13

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

containsKey(7)? Yes!

Array

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

14

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

remove(5)?

Array

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

15

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

remove(5)?

Array

null	null	null	null	null	null	7	null	null	null	null	null	null	null	null
------	------	------	------	------	------	---	------	------	------	------	------	------	------	------

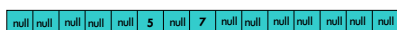
16

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

What is the big-O
runtime of the methods? $O(1)$

Array



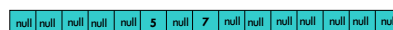
17

Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

Any problems?

Array



18

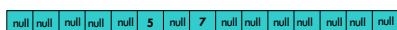
Integer sets

```
public interface IntegerSet {
    public void put(int key);
    public boolean containsKey(int key);
    public int remove(int key);
    public boolean isEmpty();
    public int size();
}
```

Any problems?

- Negative numbers (could fix this by adding a constant)
- The array would have to be huge!
(ints range from -2^{31} to $2^{31}-1$,
 $2^{31} \approx 2$ billion)

Array



19

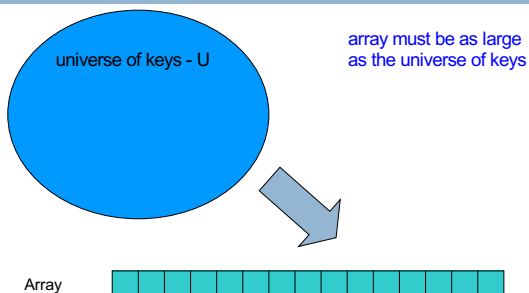
Keys



For whatever key we're using (e.g., integers), there is a universe of all possible keys

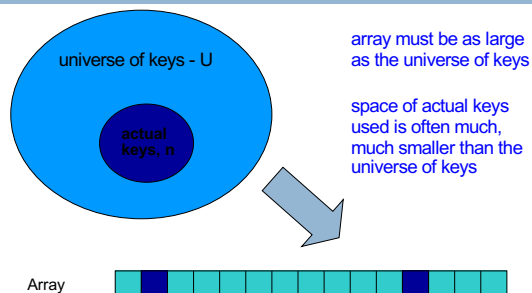
20

Why not just arrays?



21

Why not just arrays?



22

Why not arrays?

Think of indexing all last names < 10 characters

- ▣ Census listing of all last names
https://www.census.gov/topics/population/genealogy/data/2020_names.htm
- ▣ 156,621 last names
- ▣ What is the size of our space of keys?
 - 26^{10} = a big number
- ▣ Not feasible!
- ▣ Even if it were, not space efficient

23

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

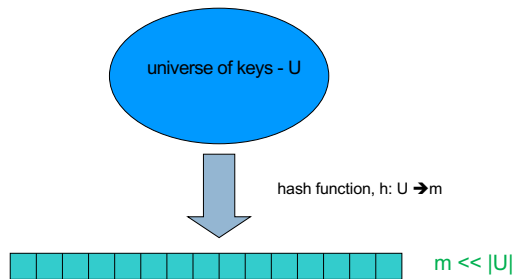
Key idea: need to translate from the key into an index in the array



24

Hash function, h

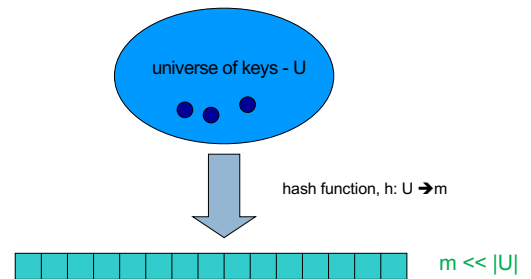
A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array)



25

Hash function, h

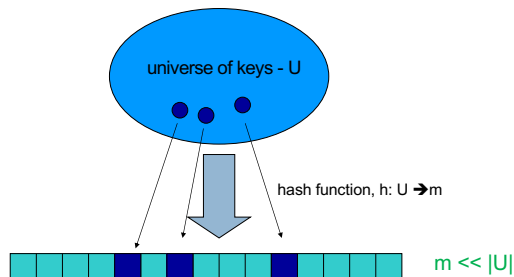
A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array)



26

Hash function, h

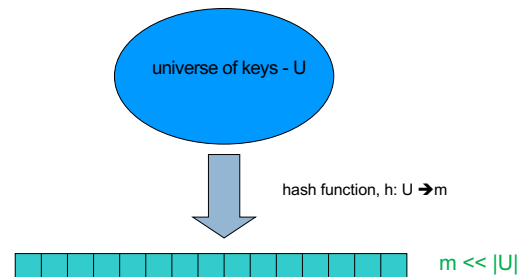
A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array)



27

Hash function, h

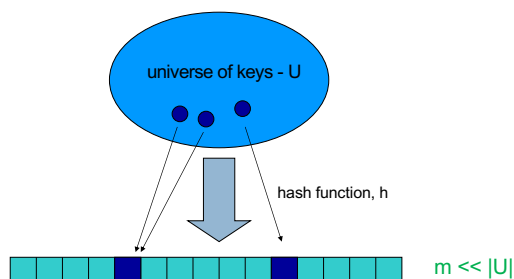
What can happen if $m < |U|$?



28

Collisions

If $m < |U|$, then two keys can map to the same position in the hashtable (pigeonhole principle)



29

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!

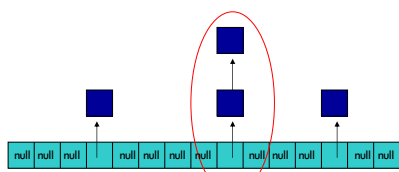
Collision resolution techniques?

30

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

31

put

```
public void put(E key) {
    LinkedList<E> entry = table[h(key)];

    if( entry == null ){
        entry = new LinkedList<E>();
        table[h(key)] = entry;
    }

    entry.addFirst(key);
}
```

What does this code do?

32

```

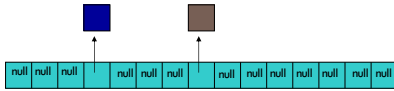
public void put(E key) {
    LinkedList<E> entry = table[h(key)];

    if( entry == null ){
        entry = new LinkedList<E>();
        table[h(key)] = entry;
    }

    entry.addFirst(key);
}

```

put(■)



33

```

public void put(E key) {
    LinkedList<E> entry = table[h(key)];

    if( entry == null ){
        entry = new LinkedList<E>();
        table[h(key)] = entry;
    }

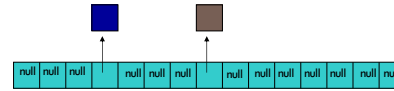
    entry.addFirst(key);
}

```

put(■)

h(■)

hash function is a mapping from the key to some value < m



34

```

public void put(E key) {
    LinkedList<E> entry = table[h(key)];

    if( entry == null ){
        entry = new LinkedList<E>();
        table[h(key)] = entry;
    }

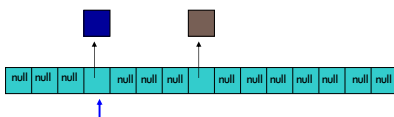
    entry.addFirst(key);
}

```

put(■)

h(■)

hash function is a mapping from the key to some value < m



35

```

public void put(E key) {
    LinkedList<E> entry = table[h(key)];

    if( entry == null ){
        entry = new LinkedList<E>();
        table[h(key)] = entry;
    }

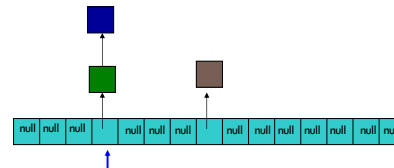
    entry.addFirst(key);
}

```

put(■)

h(■)

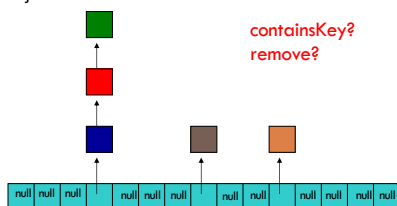
hash function is a mapping from the key to some value < m



36

More methods

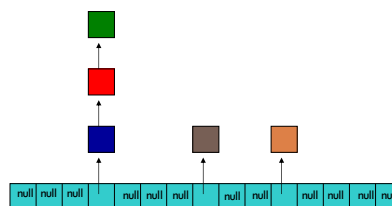
```
public interface Set<E> {
    public void put(E key);
    public boolean containsKey(E key);
    public E remove(E key);
    public boolean isEmpty();
    public int size();
}
```



37

containsKey

```
public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}
```



38

ternary operator

```
public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}
```

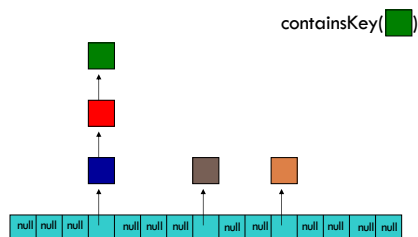
```
public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];

    if( entry == null ) {
        return false;
    } else {
        return entry.contains(key);
    }
}
```

39

containsKey

```
public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}
```



40

```
public boolean containsKey(E key){
    LinkedList<E> entry = table.get(key);
    return entry == null ? false : entry.contains(key);
}
```



```
public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}
```

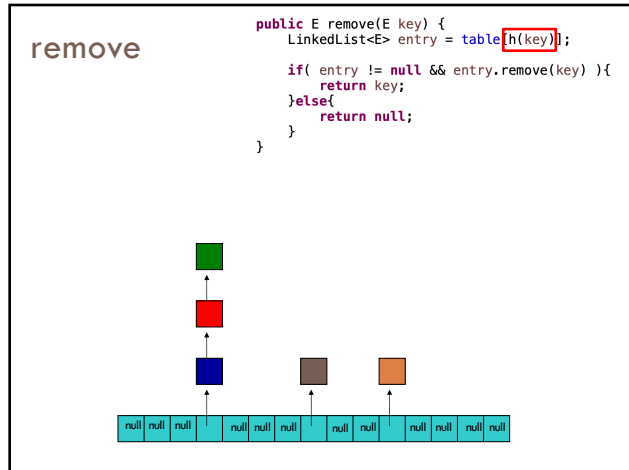


```
public boolean containsKey(E key){
    LinkedList<E> entry = table.get(key);
    return entry == null ? false : entry.contains(key);
}
```

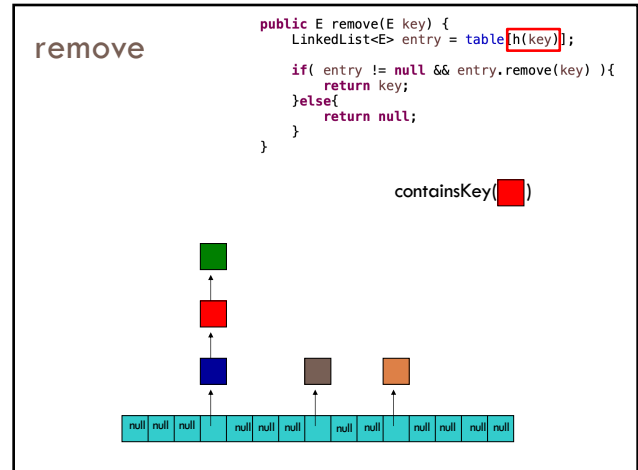


```
public boolean containsKey(E key){
    LinkedList<E> entry = table.get(key);
    return entry == null ? false : entry.contains(key);
}
```

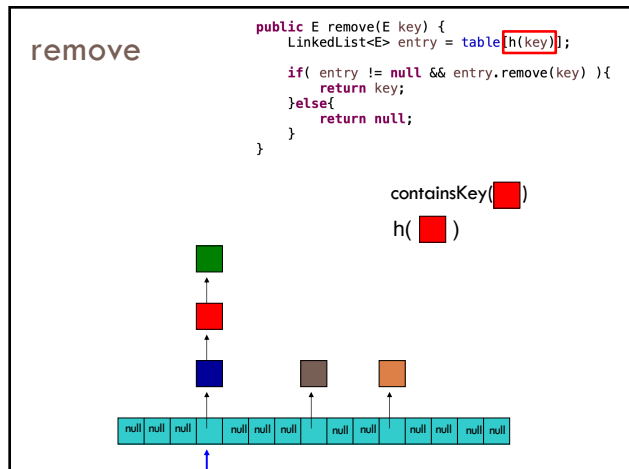




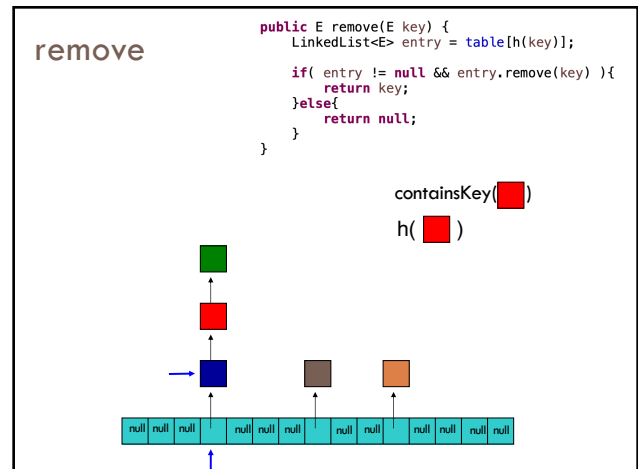
45



46



47



48

remove

```

public E remove(E key) {
    LinkedList<E> entry = table[h(key)];
    if( entry != null && entry.remove(key) ){
        return key;
    }else{
        return null;
    }
}

```

containsKey (red box)
h(red box)

49

remove

```

public E remove(E key) {
    LinkedList<E> entry = table[h(key)];
    if( entry != null && entry.remove(key) ){
        return key;
    }else{
        return null;
    }
}

```

containsKey (red box)
h(red box)

50

Average case running time?

put

```

public void put(E key) {
    LinkedList<E> entry = table[h(key)];
    if( entry == null ){
        entry = new LinkedList<E>();
        table[h(key)] = entry;
    }
    entry.addFirst(key);
}

```

containsKey

```

public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}

```

remove

```

public E remove(E key) {
    LinkedList<E> entry = table[h(key)];
    if( entry != null && entry.remove(key) ){
        return key;
    }else{
        return null;
    }
}

```

51

Average case running time?

put: $O(1)$

containsKey:
 $O(\text{length of linked list})$

remove:
 $O(\text{length of linked list})$

```

public void put(E key) {
    LinkedList<E> entry = table[h(key)];
    if( entry == null ){
        entry = new LinkedList<E>();
        table[h(key)] = entry;
    }
    entry.addFirst(key);
}

```

```

public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}

```

```

public E remove(E key) {
    LinkedList<E> entry = table[h(key)];
    if( entry != null && entry.remove(key) ){
        return key;
    }else{
        return null;
    }
}

```

52

Length of the chain

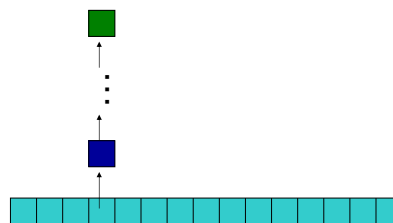
Worst case?

53

Length of the chain

Worst case?

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



54

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

55

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
 - $100/10 = 10$

56

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 (table size m with n elements)?

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

57

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

58

containsKey average running time

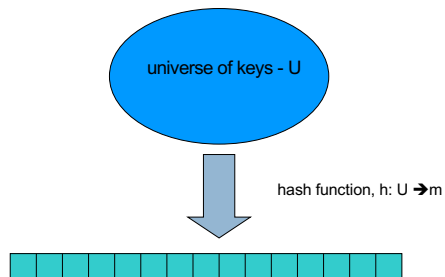
Two cases:

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

59

Hash functions

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



60

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

61

Division method

$$h(k) = k \bmod m$$

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

62

Division method

$$h(k) = k \bmod m$$

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

63

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	

64

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 $h(k) = k \bmod 2^p$, the hash function is just the lower p bits of the value

65

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

66

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

67

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

68

Multiplication method

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

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

69

Multiplication method

m	k	A	kA	h(k)
8	15	0.618	9.27	
8	23	0.618	14.214	
8	100	0.618	61.8	

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

70

Multiplication method

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

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

71

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)

72