HASHTABLES

David Kauchak
CS 62 – Spring 2021

1

Admin

No mentor hours: Friday (3/5) – Tuesday (3/16)

Mentors will reach out regarding learning community meetings.

Office hours today: 3:30-4pm

No office hours next week

No assignment over spring break!

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();
}
```

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 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();
}
```

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 

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 

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 

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?

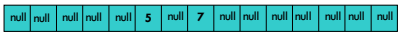
Array 

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?

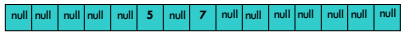
Array 

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)?

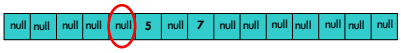
Array 

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 

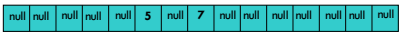
12

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



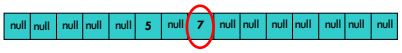
13

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



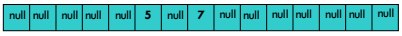
14

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



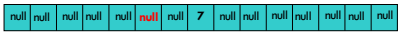
15

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



16

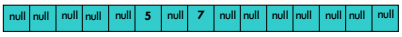
Integer sets

```
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



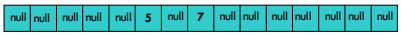
17

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

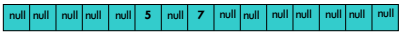
Integer sets

```
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?

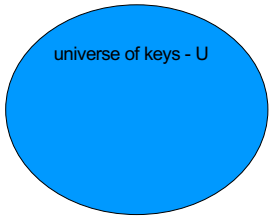
- Negative numbers (could fix this by adding a constant)
- The array would have to be huge!
  (ints range from -2^{31} to $2^{31}-1$,
  $2^{31} = \sim 2$ billion)

Array



19

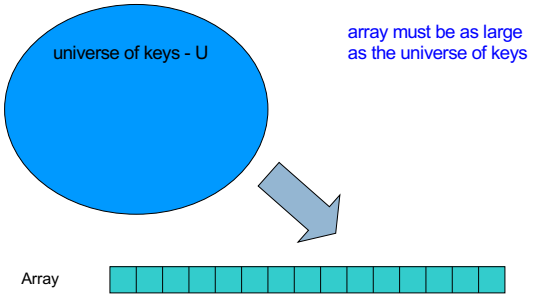
Keys



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

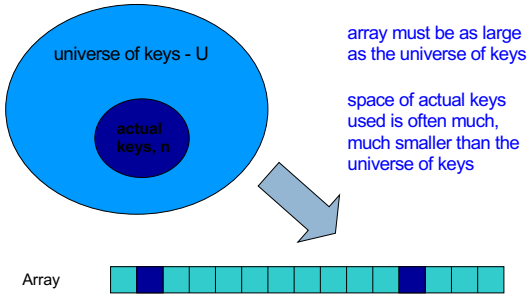
20

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
  http://www.census.gov/genealogy/names/dist.all.last
  - 88,799 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();
}
```

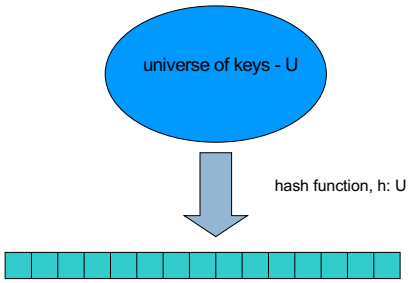
Using an array is still a good idea

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

24

Hash function, *h*

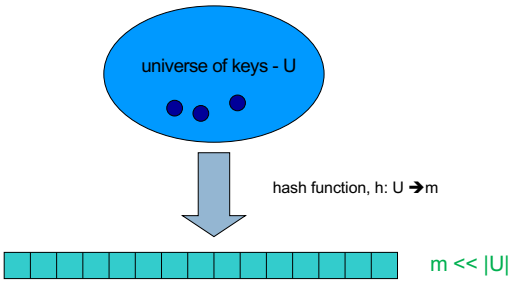
A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array)



25

Hash function, *h*

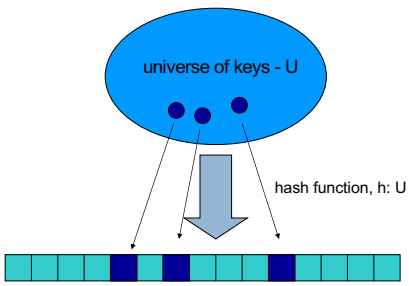
A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array)



26

Hash function, *h*

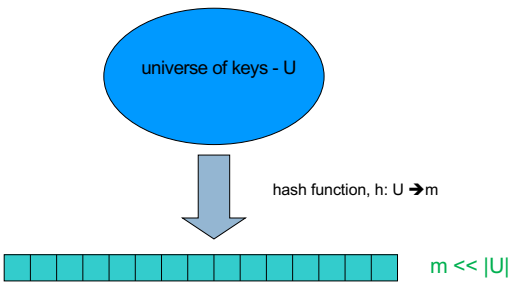
A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array)



27

Hash function, *h*

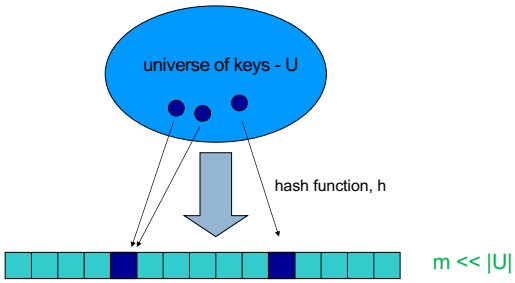
What can happen if m < |U|?



28

Collisions

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



29

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!

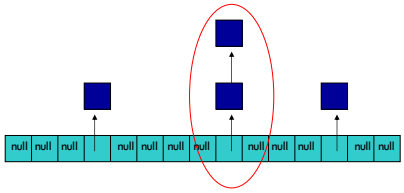
Collision resolution techniques?

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

31

put

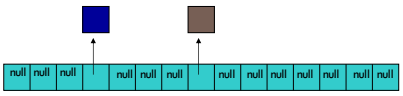
```
public void put(E key) {
    LinkedList<E> entry = table[h(key)];

    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);
}
```

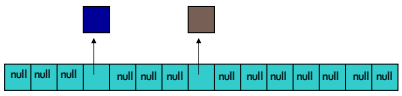
put( )



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);
}
```

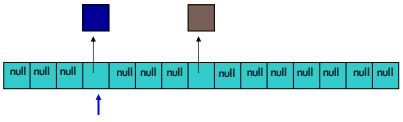
put( ) h( ) hash function is a mapping from the key to some value < m



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);
}
```

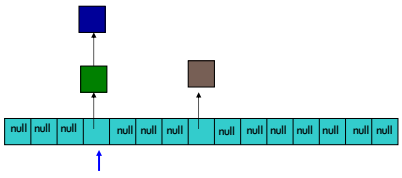
put( ) h( ) hash function is a mapping from the key to some value < m



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);
}
```

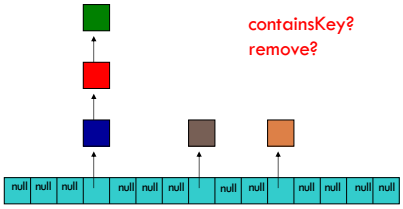
put( ) h( ) hash function is a mapping from the key to some value < m



36

More methods

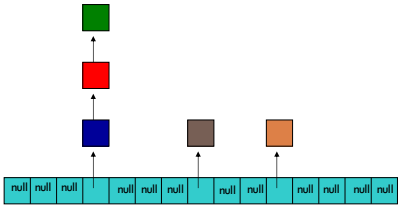
```
public interface Set<E> {
    public void put(E key);
    public boolean containsKey(E key);
    public E remove(E key);
    public boolean isEmpty();
    public int size();
}
```



37

containsKey

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



38

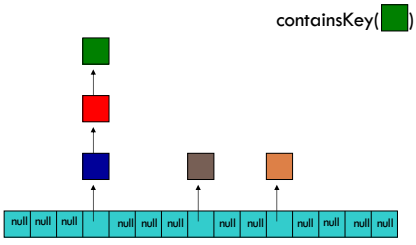
ternary operator

```
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

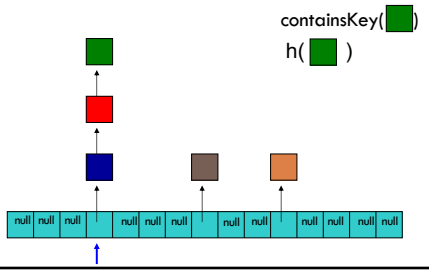
```
public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}
```



40

containsKey

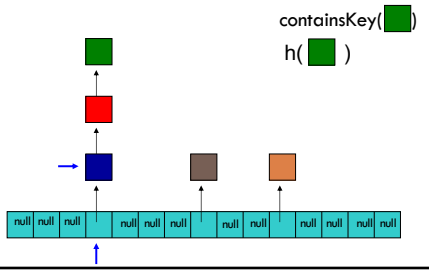
```
public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}
```



41

containsKey

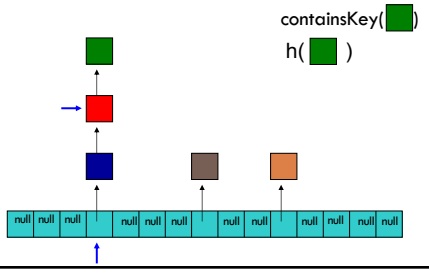
```
public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}
```



42

containsKey

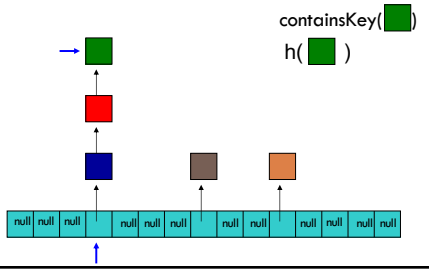
```
public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}
```



43

containsKey

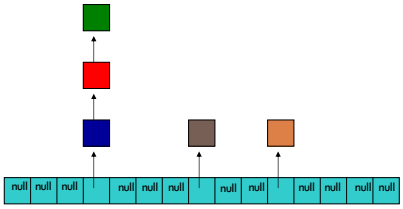
```
public boolean containsKey(E key){
    LinkedList<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}
```



44

remove

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

    if( entry != null && entry.remove(key) ){
        return key;
    }else{
        return null;
    }
}
```

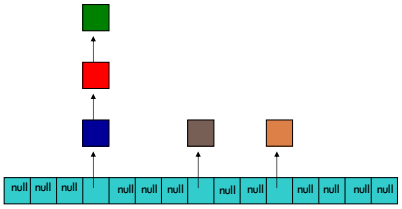


45

remove

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

    if( entry != null && entry.remove(key) ){
        return key;
    }else{
        return null;
    }
}
```

containsKey( )



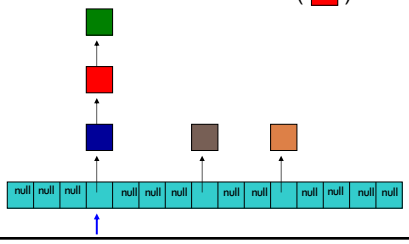
46

remove

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

    if( entry != null && entry.remove(key) ){
        return key;
    }else{
        return null;
    }
}
```

containsKey( )

h( )



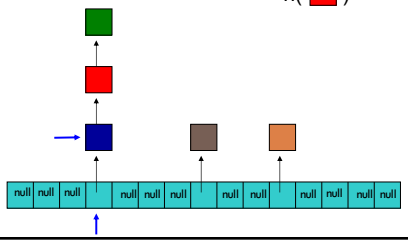
47

remove

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

    if( entry != null && entry.remove(key) ){
        return key;
    }else{
        return null;
    }
}
```

containsKey( )

h( )



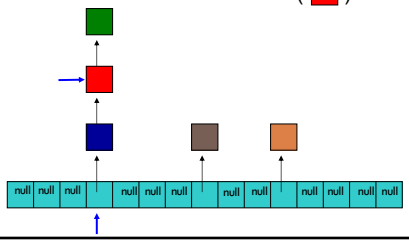
48

remove

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

    if( entry != null && entry.remove(key) ){
        return key;
    }else{
        return null;
    }
}
```

containsKey( )

h( )



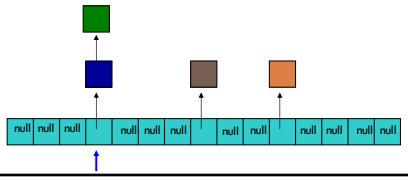
49

remove

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

    if( entry != null && entry.remove(key) ){
        return key;
    }else{
        return null;
    }
}
```

containsKey( )

h( )



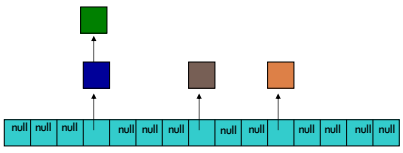
50

Running time?

put

containsKey

remove



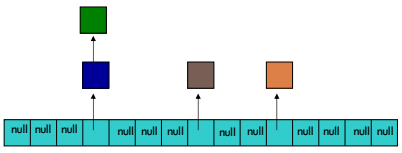
51

Running time?

put: O(1)

containsKey: O(length of linked list)

remove: O(length of linked list)



52

Length of the chain

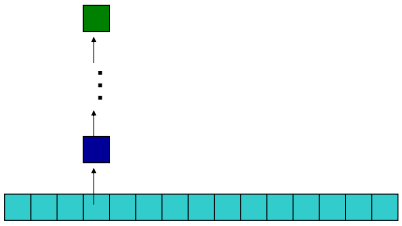
Worst case?

53

Length of the chain

Worst case?

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



54