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!

Sets

An unordered collection

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

```java
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?

We’d like to make these operations fast: O(1)!
What if we assume that they’re integers.

Any ideas how to do this fast?

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

put(7)  Where should we put it?

containsKey(4)?

put(7)  Where should we put it?

containsKey(4)?  No!
Integer sets

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

containsKey(7)?

Array: [null, null, null, null, 5, 7, null, null, null, null, null, null, null, null, null, null, null, null, null, null]

containsKey(7)? Yes!

Array: [null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null]

remove(5)?

Array: [null, null, null, null, null, 7, null, null, null, null, null, null, null, null, null, null, null, null, null, null]

remove(5)?
### Integer sets

```java
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**

```
null null null null null null null null
```

### Integer sets

```java
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**

```
null null null null null null null null
```

### Keys

- **Universe of keys - U**

For whatever key we're using (e.g., integers), there is an universe of all possible keys
Why not just arrays?

- Array must be as large as the universe of keys.

- Space of actual keys used is often much smaller than the universe of keys.

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

Hashtables

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

If $m < |U|$, what can happen?
Collisions

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

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?

Collision resolution by chaining

Hashtable consists of an array of linked lists.

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

When a collision occurs, the element is added to the 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.

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?
Hash function is a mapping from the key to some value < m
More methods

```java
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();
}
```

containsKey

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

ternary operator

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

containsKey

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

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

containsKey

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

containsKey

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

containsKey

public boolean containsKey(E key) {
    Linkedlist<E> entry = table[h(key)];
    return entry == null ? false : entry.contains(key);
}
public E remove(E key) {
    if (entry != null && entry.remove(key)) {
        return key;
    } else {
        return null;
    }
}
```java
public E remove(E key) {
    LinkedList<E> entry = table[h(key)];
    if (entry != null && entry.remove(key)) {
        return key;
    } else {
        return null;
    }
}
```
Length of the chain

Worst case?

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