CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
Capen 208

Lecture 12
The Set ADT

Announcements

- PA1 Implementation due last night, AutoLab open until Tuesday
- WA2 out now, due Sunday @ 11:59PM
  - Includes similar content to the first midterm

Collection ADTs

Property	`Sequence`	`List`	`Set`	`Bag`
Explcit Order	✓	✓		
Enforced Uniqueness			✓	
Fixed Size	✓			
Iterable	✓	✓	✓	✓

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order does not matter, and only one copy of each item is allowed)

The Set ADT

`void add(T element)`

Store one copy of **`element`** if not already present

`boolean contains(T element)`

Return **`true`** if **`element`** is present in the set

`boolean remove(T element)`

Remove **`element`** if present, or return **`false`** if not

Bags

A **Bag** is an **unordered** collection of **non-unique** elements.

(order does not matter, and there can be multiple copies of an item)

The Bag ADT

`void add(T element)`

Store one copy of **`element`**

`int contains(T element)`

Return the number of copies of **`element`** in the bag

`boolean remove(T element)`

Remove one copy of **`element`** if present, or return **`false`** if not

Note: *Sometimes referred to as a multiset. Java does not have a native Bag/Multiset class*

Recap

- `LinkedLists`, `ArrayLists`, and `Arrays` are **data structures**

Recap

- **`LinkedLists`**, **`ArrayLists`**, and **`Arrays`** are **data structures**
- **`Sequences`**, **`Lists`**, **`Sets`**, and **`Bags`** are **ADTs**

Recap

- `LinkedLists`, `ArrayLists`, and `Arrays` are **data structures**
- `Sequences`, `Lists`, `Sets`, and `Bags` are **ADTs**
- We've implemented `Sequences` and `Lists` with multiple data structures

Recap

- **`LinkedLists`**, **`ArrayLists`**, and **`Arrays`** are **data structures**
- **`Sequences`**, **`Lists`**, **`Sets`**, and **`Bags`** are **ADTs**
- We've implemented **`Sequences`** and **`Lists`** with multiple data structures
- **Now let's implement `Sets` and `Bags`!**

This idea of taking a given data structure, and implementing a given ADT is a core skill in this class!

Set Implementation (w/`LinkedList`)

```
LinkedList<T> data


add(elem):
```

Set Implementation (w/LinkedList)

```
LinkedList<T> data


add(elem):
    data.add(elem)
```

Is this implementation correct?

Set Implementation (w/LinkedList)

```
LinkedList<T> data

add(elem):
    data.add(elem)
```

Is this implementation correct?

From the ADT:

`void add(T element)`

Store one copy of **`element`** **<u>if not already present</u>**

Set Implementation (w/LinkedList)

```
LinkedList<T> data

add(elem):
  if (!contains(elem))
    data.add(elem)
```

From the ADT:

`void add(T element)`
Store one copy of **`element`** **if not already present**

Runtime?

Set Implementation (w/LinkedList)

```
LinkedList<T> data

add(elem):
  if (!contains(elem))
    data.add(elem)
```

From the ADT:

`void add(T element)`

Store one copy of **`element`** **if not already present**

Runtime?

We need to know how **`contains`** works!

Set Implementation (w/LinkedList)

```
LinkedList<T> data


contains(elem):
```

Set Implementation (w/LinkedList)

```
LinkedList<T> data

contains(elem):
    curr = data.head
    while curr.isPresent():
        if curr.value == elem:
            return true
        curr = curr.next
    return false
```

Runtime?

Set Implementation (w/LinkedList)

```
LinkedList<T> data

contains(elem):
    curr = data.head
    while curr.isPresent():
        if curr.value == elem:
            return true
        curr = curr.next
    return false
```

Runtime?

$\Theta(1)$ per iteration...how many iterations?

Set Implementation (w/LinkedList)

```
LinkedList<T> data

contains(elem):
    curr = data.head
    while curr.isPresent():
        if curr.value == elem:
            return true
        curr = curr.next
    return false
```

Runtime?

$\Theta(1)$ per iteration...how many iterations? $O(n)$, $\Omega(1)$

Set Implementation (w/LinkedList)

```
LinkedList<T> data

contains(elem):
    curr = data.head
    while curr.isPresent():
        if curr.value == elem:
            return true
        curr = curr.next
    return false
```

Runtime? $O(n), \Omega(1)$

Set Implementation (w/LinkedList)

```
LinkedList<T> data

add(elem):
    if (!contains(elem))
        data.add(elem)
```

Runtime?

Set Implementation (w/LinkedList)

```
LinkedList<T> data

add(elem):
    if (!contains(elem))
        data.add(elem)
```

Runtime?

$$T_{\text{add}}(n) = \begin{cases} O(n),\ \Omega(1) & \text{if elem in set} \\ \Theta(1) & \text{otherwise} \end{cases}$$

Set Implementation (w/LinkedList)

```
LinkedList<T> data

add(elem):
    if (!contains(elem))
        data.add(elem)
```

Runtime? $O(n)$, $\Omega(1)$

$$T_{\text{add}}(n) = \begin{cases} O(n),\ \Omega(1) & \text{if elem in set} \\ \Theta(1) & \text{otherwise} \end{cases}$$

Set Implementation (w/LinkedList)

```
LinkedList<T> data


remove(elem):
```

Set Implementation (w/LinkedList)

```
LinkedList<T> data

remove(elem):
    curr = data.head
    while curr.isPresent():
        if curr.value == elem:
            data.remove(curr)
            return true
        curr = curr.next
    return false
```

Runtime?

Set Implementation (w/LinkedList)

```
LinkedList<T> data

remove(elem):
    curr = data.head
    while curr.isPresent():
        if curr.value == elem:
            data.remove(curr)
            return true
        curr = curr.next
    return false
```

Runtime?

Cost per iteration?

Set Implementation (w/LinkedList)

```
LinkedList<T> data

remove(elem):
    curr = data.head
    while curr.isPresent():
        if curr.value == elem:
            data.remove(curr)
            return true
        curr = curr.next
    return false
```

Runtime?

Cost per iteration? $\Theta(1)$

Set Implementation (w/LinkedList)

```
LinkedList<T> data

remove(elem):
    curr = data.head
    while curr.isPresent():
        if curr.value == elem:
            data.remove(curr)
            return true
        curr = curr.next
    return false
```

Runtime? $O(n)$, $\Omega(1)$

Cost per iteration? $\Theta(1)$

Bag Implemtation (w/`LinkedList`)

What changes do we need to make to implement a Bag instead?

Bag Implemtation (w/`LinkedList`)

What changes do we need to make to implement a Bag instead?

`void add(T elem)`

  No longer need to check if **`elem`** is already present, runtime becomes $\Theta(1)$

`int contains(T elem)`

  Returns the ***number*** of occurrences – runtime becomes $\Theta(n)$ instead of $O(n)$

`boolean remove(T elem)`

  No change

Set Implementation (w/ArrayList)

```
ArrayList<T> data


add(elem):
```

Set Implementation (w/ArrayList)

```
ArrayList<T> data

add(elem):
  if (!contains(elem))
    data.add(elem)
```

Runtime?

Set Implementation (w/ArrayList)

```
ArrayList<T> data

add(elem):
  if (!contains(elem))
    data.add(elem)
```

Runtime?

We need to know how **`contains`** works!

Set Implementation (w/ArrayList)

```
ArrayList<T> data


contains(elem):
```

Set Implementation (w/ArrayList)

```
ArrayList<T> data

contains(elem):
    idx = 0
    while idx < data.size():
        if data[idx] == elem:
            return true
        idx = idx + 1
    return false
```

Runtime?

Set Implementation (w/ArrayList)

```
ArrayList<T> data

contains(elem):
    idx = 0
    while idx < data.size():
        if data[idx] == elem:
            return true
        idx = idx + 1
    return false
```

Runtime? $O(n), \Omega(1)$

Set Implementation (w/ArrayList)

```
ArrayList<T> data

add(elem):
    if (!contains(elem))
        data.add(elem)
```

Runtime?

Set Implementation (w/ArrayList)

```
ArrayList<T> data

add(elem):
    if (!contains(elem))
        data.add(elem)
```

Runtime? $O(n)$, $\Omega(1)$

Set Implementation (w/`ArrayList`)

```
ArrayList<T> data

remove(elem):
```

Set Implementation (w/ArrayList)

```
ArrayList<T> data

remove(elem):
    idx = 0
    while idx < data.size():
        if data[idx] == elem:
            data.remove(idx)
            return true
        idx = idx + 1
    return false
```

Runtime?

Set Implementation (w/ArrayList)

```
ArrayList<T> data

remove(elem):
    idx = 0
    while idx < data.size():
        if data[idx] == elem:
            data.remove(idx)
            return true
        idx = idx + 1
    return false
```

Runtime?

What is the cost per iteration?

Set Implementation (w/ArrayList)

```
ArrayList<T> data

remove(elem):
    idx = 0
    while idx < data.size():
        if data[idx] == elem:
            data.remove(idx)
            return true
        idx = idx + 1
    return false
```

Runtime?

What is the cost per iteration? $O(n)$

How many iterations?

Set Implementation (w/ArrayList)

```
ArrayList<T> data

remove(elem):
    idx = 0
    while idx < data.size():
        if data[idx] == elem:
            data.remove(idx)
            return true
        idx = idx + 1
    return false
```

Runtime?

What is the cost per iteration? $O(n)$

How many iterations? $O(n)$

Set Implementation (w/ArrayList)

```
ArrayList<T> data

remove(elem):
    idx = 0
    while idx < data.size():
        if data[idx] == elem:
            data.remove(idx)
            return true
        idx = idx + 1
    return false
```

Runtime?

What is the cost per iteration? $O(n)$

How many iterations? $O(n)$

STOP! Think big picture...

How many operations do we perform per element?

Set Implementation (w/ArrayList)

```
ArrayList<T> data

remove(elem):
    idx = 0
    while idx < data.size():
        if data[idx] == elem:
            data.remove(idx)
            return true
        idx = idx + 1
    return false
```

Runtime?

What is the cost per iteration? $O(n)$

How many iterations? $O(n)$

STOP! Think big picture...

How many operations do we perform per element? Exactly one! Either we check it for a match, or we move it to fill the hole

Runtime?

Set Implementation (w/ArrayList)

```
ArrayList<T> data

remove(elem):
    idx = 0
    while idx < data.size():
        if data[idx] == elem:
            data.remove(idx)
            return true
        idx = idx + 1
    return false
```

Runtime?

What is the cost per iteration? $O(n)$

How many iterations? $O(n)$

STOP! Think big picture...

How many operations do we perform per element? Exactly one! Either we check it for a match, or we move it to fill the hole

Runtime? $\Theta(n)$

Bag Implemtation (w/`ArrayList`)

What changes do we need to make to implement a Bag instead?

Bag Implemtation (w/`ArrayList`)

What changes do we need to make to implement a Bag instead?

`void add(T elem)`

Don't check if **`elem`** is already present, runtime becomes amortized $\Theta(1)$

`int contains(T elem)`

Returns the ***number*** of occurrences – runtime becomes $\Theta(n)$ instead of $O(n)$

`boolean remove(T elem)`

No change

Sets and Bags (...so far)

	`LinkedList`	`ArrayList`
`Set.add`	$O(n)$	$O(n)$
`Set.contains`	$O(n)$	$O(n)$
`Set.remove`	$O(n)$	$\Theta(n)$
`Bag.add`	$O(1)$	$O(n)$, Amortized $\Theta(1)$
`Bag.contains`	$\Theta(n)$	$\Theta(n)$
`Bag.remove`	$O(n)$	$\Theta(n)$