CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 11: List Review and Sets/Bags

Announcements

- PA1 Implementation due Sunday @ 11:59PM
- WA2 will be released after the PA1 deadline, due 2/27 @ 11:59PM

Review of Amortized Runtime

With **amortized** analysis we look at the total cost of a series of operations and imagine that total cost spread evenly over the operations

Definition: If calling a function, `foo`, n times takes $O(f(n))$ steps, the **amortized runtime** of `foo` is $O(f(n)/n)$

Review of Amortized Runtime

With **amortized** analysis we look at the total cost of a series of operations and imagine that total cost spread evenly over the operations

Definition: If calling a function, `foo`, n times takes $\Theta(f(n))$ steps, the **amortized runtime** of `foo` is $\Theta(f(n)/n)$

Review of Amortized Runtime

With **amortized** analysis we look at the total cost of a series of operations and imagine that total cost spread evenly over the operations

Definition: If calling a function, `foo`, n times takes $\Omega(f(n))$ steps, the **amortized runtime** of `foo` is $\Omega(f(n)/n)$

Example

The ArrayList.add(v) function has **unqualified runtime** $O(n)$

If we call ArrayList.add(v) n times in a row, it takes $\Theta(n)$ steps total

Therefore the **amortized runtime** of ArrayList.add(v) is $\Theta(n/n) = \Theta(1)$

Another Example

Imagine a coffee shop that sells coffee for $1

They also sell a reusable cup for $3.50 and refilling it only costs $0.50

How much does it cost to buy coffee for a week w/o the reusable cup?

How much does it cost to buy coffee for a week w/ the reusable cup?

Another Example

Imagine a coffee shop that sells coffee for $1

They also sell a reusable cup for $3.50 and refilling it only costs $0.50

How much does it cost to buy coffee for a week w/o the reusable cup? $7

How much does it cost to buy coffee for a week w/ the reusable cup?

Another Example

Imagine a coffee shop that sells coffee for $1

They also sell a reusable cup for $3.50 and refilling it only costs $0.50

How much does it cost to buy coffee for a week w/o the reusable cup? $7

How much does it cost to buy coffee for a week w/ the reusable cup? $7

Another Example

Ima

They

How **? $7**

How **$7**

The **amortized cost** of buying coffee with the reusable cup for the week ends up being $1 per cup of coffee, even though on the first day you spend a lot more ($4).

If you know you are going to be buying a lot of coffee, buying the cup works out in the long run.

If you only end up buying one or two coffees, the reusable cup is more expensive.

Amortized Example

```
for (int i = 0; i < n; i++) {
  foo(i);
}
```

Imagine the **unqualified runtime** of `foo` is $O(n^3)$...what is worst-case runtime of the above code?

Amortized Example

```
for (int i = 0; i < n; i++) {
  foo(i);
}
```

With an unqualified upper bound, the shortcut of iter_cost * num_iters may not give a tight bound!

Imagine the **unqualified runtime** of **`foo`** is $O(n^3)$...what is worst-case runtime of the above code? $O(n^4)$

Is that a tight bound? **We don't know!**

Amortized Example

```
for (int i = 0; i < n; i++) {
  foo(i);
}
```

With an unqualified upper bound, the shortcut of iter_cost * num_iters may not give a tight bound!

Imagine the **unqualified runtime** of `foo` is $O(n^3)$...what is worst-case runtime of the above code? $O(n^4)$

Is that a tight bound? **We don't know!**

If the **amortized runtime** of foo is $\Theta(n)$, what is the runtime?

Amortized Example

```
for (int i = 0; i < n; i++) {
  foo(i);
}
```

With an unqualified upper bound, the shortcut of iter_cost * num_iters may not give a tight bound!

Imagine the **unqualified runtime** of `foo` is $O(n^3)$...what is worst-case runtime of the above code? $O(n^4)$

Is that a tight bound? **We don't know!**

If the **amortized runtime** of foo is $\Theta(n)$, what is the runtime? $\Theta(n^2)$

But with amortized the shortcut always works!

List Summary So Far

	ArrayList	Linked List (by index)	Linked List (by reference)
`get(idx)`	$\Theta(1)$	$\Theta(\text{idx}) \subset O(n)$	$\Theta(1)$
`set(idx,v)`	$\Theta(1)$	$\Theta(\text{idx}) \subset O(n)$	$\Theta(1)$
`size()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(v)`	$O(n)$, Amortized $\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(idx,v)`	??	$\Theta(\text{idx}) \subset O(n)$	$\Theta(1)$
`remove(idx)`	$O(n)$	$\Theta(\text{idx}) \subset O(n)$	$\Theta(1)$

Follow-Up Questions

What is the amortized runtime of **`add`** for a **`LinkedList`**?

What is the runtime of **`add(int idx, E elem)`** for an **`ArrayList`**?

Follow-Up Questions

What is the amortized runtime of **`add`** for a **`LinkedList`**?

Each **`add`** is **Θ(1)**. Total for ***n*** calls is **Θ(*n*)**. Amortized is **Θ(*n*/*n*) = Θ(1)**

Note: This is the same as the amortized runtime of **`ArrayList`** **`add`**!

That means that even though **`LinkedList`** and **`ArrayList`** **`add`** may perform differently for a single call, they'll perform the same in a loop!

Follow-Up Questions

What is the amortized runtime of **`add`** for a **`LinkedList`**?

Each **`add`** is $\Theta(1)$. Total for n calls is $\Theta(n)$. Amortized is $\Theta(n/n) = \Theta(1)$

What is the runtime of **`add(int idx, E elem)`** for an **`ArrayList`**?

Follow-Up Questions

What is the amortized runtime of **`add`** for a **`LinkedList`**?

Each **`add`** is $\Theta(1)$. Total for n calls is $\Theta(n)$. Amortized is $\Theta(n/n) = \Theta(1)$

What is the runtime of **`add(int idx, E elem)`** for an **`ArrayList`**?

To **`add`** between two elements requires the rest of the elements to be shifted to the right (opposite of **`remove`**), so runtime is always $O(n)$.

(Either we are out of space so we copy n, or we have space so we shift n)

`List` Summary So Far

	ArrayList	Linked List (by index)	Linked List (by reference)
`get(idx)`	$\Theta(1)$	$\Theta(\text{idx}) \subset \boldsymbol{O}(n)$	$\Theta(1)$
`set(idx,v)`	$\Theta(1)$	$\Theta(\text{idx}) \subset \boldsymbol{O}(n)$	$\Theta(1)$
`size()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(v)`	$\boldsymbol{O}(n)$, Amortized $\boldsymbol{\Theta(1)}$	$\Theta(1)$	$\Theta(1)$
`add(idx,v)`	$\boldsymbol{O}(n)$	$\Theta(\text{idx}) \subset \boldsymbol{O}(n)$	$\Theta(1)$
`remove(idx)`	$\boldsymbol{O}(n)$	$\Theta(\text{idx}) \subset \boldsymbol{O}(n)$	$\Theta(1)$

What Data Structure is Best?

Scenario #1: You need to read in the lines of a CSV file, store them in a List, and later be able to access individual records based on index.

What Data Structure is Best?

Scenario #1: You need to read in the lines of a CSV file, store them in a List, and later be able to access individual records based on index.

`ArrayList`

Since the amortized runtime of add for **`ArrayList`** and **`LinkedList`** is Θ (1), adding the n lines of the CSV file will take $\Theta(n)$ time for both…

But **`ArrayLists`** will then have an advantage because looking up records by index will be $O(1)$ whereas **`LinkedLists`** will be $O(n)$

What Data Structure is Best?

Scenario #2: Users logging onto an online game need to be efficiently added to a List in the order they log on. From time to time you must be able to iterate through the list from beginning to end.

What Data Structure is Best?

Scenario #2: Users logging onto an online game need to be efficiently added to a List in the order they log on. From time to time you must be able to iterate through the list from beginning to end.

`LinkedList`

The enumeration will cost a total of $\Theta(n)$ for both types of List

But some users will experience longer waits being added to the List if implemented as an **`ArrayList`** due to the need for it to occasionally resize

Sets and Bags

Collection ADTs

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

Sets

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

(order doesn't matter, and at most one copy of each item)

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 doesn't matter, and multiple copies of the same item is OK)

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 multiset. Java does not have a native Bag/Multiset class.

Implementation

- **`LinkedLists`** and **`ArrayLists`** are **data structures**
- **`Sequences`**, **`Lists`**, **`Sets`** and **`Bags`** are **ADTs**
- We've already seen how we can implement **`Sequences`** and **`Lists`** with both **`LinkedLists`** and **`Arrays`**
- Now let's implement **`Sets`** and **`Bags`**

This idea of taking a given data structure and implementing a given ADT will be an important skill in this class!

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;

add(elem):
```

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;

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

Is this correct?

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;

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

Is this correct?

`void add(T element)`

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

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;

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

Runtime?

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;
add(elem):
  if(!contains(elem)):
    Θ(1)
```

Runtime?

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;
add(elem):
  if(!contains(elem)):
    Θ(1)
```

Runtime? Depends on **`contains`**…

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;

contains(elem):
```

Set Pseudocode (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 Pseudocode (w/LinkedList)

```
LinkedList<T> data;
contains(elem):
  Θ(1)
  while curr.isPresent():
    Θ(1)
  Θ(1)
```

Runtime?

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;

contains(elem):

  Θ(1)

  while curr.isPresent():

    Θ(1)

  Θ(1)
```

Runtime? We are not guaranteed to do all ***n*** iterations! So runtime is $O(n)$ but not $\Theta(n)$

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;

add(elem):
  if(!contains(elem)):
    Θ(1)
```

Runtime? Depends on **`contains`**…

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;

add(elem):

  if(!contains(elem)):

    Θ(1)
```

Runtime? *O*(*n*)

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;

remove(elem):
```

Set Pseudocode (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
```

Set Pseudocode (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
```

Remove by reference!

Set Pseudocode (w/LinkedList)

```
LinkedList<T> data;

remove(elem):
  Θ(1)
  while curr.isPresent():
    Θ(1)
  Θ(1)
```

Runtime? $O(n)$

Implementing Bag (w/LinkedList)

What changes if we are implementing a Bag instead?

Implementing Bag (w/LinkedList)

What changes if we are implementing a Bag instead?

- **`add`** doesn't need to check if **`elem`** is already in the bag...it's now $\Theta(1)$
- **`contains`** returns the number of occurrences; $\Theta(n)$ instead of $O(n)$
- **`remove`** doesn't change

Set Pseudocode (w/ArrayList)

```
ArrayList<T> data;

add(elem):
```

Set Pseudocode (w/ArrayList)

```
ArrayList<T> data;
add(elem):
  if(!contains(elem)):
    data.add(elem)
```

Runtime?

Set Pseudocode (w/ArrayList)

```
ArrayList<T> data;
add(elem):
  if(!contains(elem)):
    O(n), Amortized Θ(1)
```

Runtime? Still depends on **`contains`**

Set Pseudocode (w/ArrayList)

```
ArrayList<T> data;

contains(elem):
```

Set Pseudocode (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 Pseudocode (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*)

Set Pseudocode (w/ArrayList)

```
ArrayList<T> data;
add(elem):
  if(!contains(elem)):
```

$O(n)$, Amortized $\Theta(1)$

Runtime? Still depends on **`contains`**

Set Pseudocode (w/ArrayList)

```
ArrayList<T> data;
add(elem):
  if(!contains(elem)):
    O(n), Amortized Θ(1)
```

Runtime? $O(n)$

What about amortized?

Set Pseudocode (w/ArrayList)

```
ArrayList<T> data;
add(elem):
  if(!contains(elem)):
    O(n), Amortized Θ(1)
```

Runtime? *O*(*n*)

What about amortized? ***O*(*n*)**

Set Pseudocode (w/ArrayList)

```
ArrayList<T> data;

remove(elem):
```

Set Pseudocode (w/ArrayList)

Runtime?

```
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
```

Set Pseudocode (w/ArrayList)

Runtime? $O(n)$

What about Θ?

```
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
```

Set Pseudocode (w/ArrayList)

Runtime? $O(n)$

What about Θ? For this code…$\Theta(\mathtt{i})$ steps to find the element at index $\mathtt{i}$, $\Theta(n - \mathtt{i})$ steps to remove it. $\Theta(\mathtt{i}) + \Theta(n - \mathtt{i}) = \Theta(n)$

```
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
```

Implementing Bag (w/ArrayList)

What changes if we are implementing a Bag instead?

Implementing Bag (w/ArrayList)

What changes if we are implementing a Bag instead?

- `add` doesn't need to check if `elem` is already in it: amortized $\Theta(1)$
- `contains` returns the number of occurrences; $\Theta(n)$ instead of $O(n)$
- `remove` doesn't 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`	$O(n)$	$\Theta(n)$

Potential Improvements

How could we improve these implementations?

Thought…does order matter for sets?

Potential Improvements

How could we improve these implementations?

Thought…does order matter for sets? **No!**

Can we somehow take advantage of that?

Small Improvement

Notice how the ArrayList version of remove was $\Theta(n)$ because we had to shift over elements to fill the hole after removing the target…

If we don't need to maintain order, we don't need to shift everything to fill the hole, we can just fill it with the last item!

Set Pseudocode (w/ArrayList)

Runtime?

```
ArrayList<T> data;
remove(elem):
  idx ← 0
  while idx < data.size():
    if data[idx] == elem:
      data[idx] =
          data[data.size()-1]
      data.remove(data.size()-1)
      return true
    idx = idx + 1
  return false
```

Set Pseudocode (w/ArrayList)

Runtime? Still $O(n)$...but now $\Omega(1)$

Just a tactical optimization, doesn't change the asymptotic runtime...

```
ArrayList<T> data;
remove(elem):
  idx ← 0
  while idx < data.size():
    if data[idx] == elem:
      data[idx] =
          data[data.size()-1]
      data.remove(data.size()-1)
      return true
    idx = idx + 1
  return false
```

Slightly Better Improvement

What if we were to store elements in sorted order instead of the order they were added…

More on that in a future lecture