Announcements

- PA1 due Sunday @ 11:59PM
Recap

- **ADT**: Abstract Data Type, defines what a particular data structure can be used without specifying how it is implemented
  - ie: `Seq, mutable.Seq`
- **Array**: A type of sequence with a fixed element size and fixed number of elements, allocated as a contiguous block of memory
  - Immutable
  - Constant time random access (base + index * element size)
- **ArrayBuffer**: The mutable form of an array, allows insert and remove
<table>
<thead>
<tr>
<th><strong>ADT</strong></th>
<th><strong>Data Structure</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The interface to a data structure</td>
<td>The implementation of one (or more) ADTs</td>
</tr>
<tr>
<td>Defines <em>what</em> the data structure can do</td>
<td>Defines <em>how</em> the different tasks are carried out</td>
</tr>
<tr>
<td>Many data structures can implement the same ADT</td>
<td>Different data structures will excel at different tasks</td>
</tr>
</tbody>
</table>
Types of Collections in Scala

**Iterable** - Any collection of items

**Seq** - A collection of items in a specific order

**IndexedSeq** - A Seq where there is guaranteed O(1) access to items

**Set** - A collection of unique items

**Map** - A collection of items identified by a key (associative collection)
Types of Sequences in Scala

**mutable.Seq** - Like Seq.....but mutable

**mutable.Buffer** - Like mutable.Seq, but "efficient" appends.

**Queue** - Like mutable.Seq but "efficient" append and remove first.  
   *Think like a queue of people*

**Stack** - Like mutable.Seq but "efficient" prepend and remove first.  
   *Think like a stack of papers*
apply(idx: Int): [A]
   Get the element (of type A) at position idx

iterator: Iterator[A]
   Get access to view all elements in the sequence, in order, once

length: Int
   Count the number of elements in the seq

insert(idx: Int, elem: A): Unit
   Insert an element at position idx with value elem

remove(idx: Int): A
   Remove the element at position idx, and return the removed value
What does an Array of n items of type T actually look like?

- 4 bytes for n (optional)
- 4 bytes for sizeof(T) (optional)
- n * sizeof(T) bytes for the data
Challenge: Operations that modify the array size require copying the array!
Challenge: Operations that modify the array size require copying the array!

Solution: Reserve extra space!
What does an `ArrayBuffer` of \( n \) items of type \( T \) actually look like?

- 4 bytes for \( n \) (optional)
- 4 bytes for `sizeof(T)` (optional)
- 4 bytes for the number of `used` fields
- \( n \times \text{sizeof}(T) \) bytes for the data

<table>
<thead>
<tr>
<th>n</th>
<th><code>sizeof(T)</code></th>
<th><code>u</code></th>
<th>a(1) or None</th>
<th>a(2) or None</th>
<th>a(3) or None</th>
<th>a(4) or None</th>
<th>...</th>
</tr>
</thead>
</table>
class ArrayBuffer[T] extends Buffer[T] {
  var used = 0
  var data = Array[Option[T]].fill(INITIAL_SIZE) { None }

  def length = used

  def apply(i: Int): T = {
    if(i < 0 || i >= used) { throw new IndexOutOfBoundsException(i) }
    return data(i).get
  }

  /* ... */
}
What is Option[T]...a brief digression

- Let's say we have a function that we know can possibly return `null`
- What can go wrong in the following code snippet?

```scala
val x = functionThatCanReturnNull()
x.doAThing()
```
What is Option[T]... a brief digression

- Let's say we have a function that we know can possibly return `null`
- What can go wrong in the following code snippet?

```java
val x = functionThatCanReturnNull()
x.doAThing()
```

`java.lang.NullPointerException` (runtime error)
What is Option[T]...a brief digression

- Let's say we have a function that we know can possibly return null
- What can go wrong in the following code snippet?

```scala
val x = functionThatCanReturnNull()
if(x == null) { /* do something special */ }
else { x.doAThing() }
```

It's very easy in practice to miss doing this test!
What is Option[T]...a brief digression

- What if instead that function returns something called an Option?

```scala
val x = functionThatReturnsOption()
x.doAThing()
```

error: value doAThing is not a member of Option[MyClass]
What is Option[T]...a brief digression

- What if instead that function returns something called an Option?

```scala
val x = functionThatReturnsOption()
x.doA Thing()
```

error: value doAThing is not a member of Option[MyClass]

Now it's a compile time error...Easier to catch
What is Option[T]...a brief digression

- But what is an Option (in Scala)?

**Some (x)**
- Subclass of Option[T]
- `value.isDefined == true`
- A valid value exists and we can access it with `value.get`

**None**
- Subclass of Option[T]
- `value.isEmpty == true`
- Analogous to `null`. No value.
Now back to ArrayBuffers...
class ArrayBuffer[T] extends Buffer[T] {
  var used = 0
  var data = Array[Option[T]].fill(INITIAL_SIZE) { None }

  def length = used

  def apply(i: Int): T = {
    if(i < 0 || i >= used) { throw new IndexOutOfBoundsException(i) }
    return data(i).get
  }

  /* ... */
}
def remove(target: Int): T = {
  /* Sanity-check inputs */
  if(target < 0 || target >= used) {
    throw new IndexOutOfBoundsException(target)
  }
  /* Shift elements left */
  for(i <- target until (used-1)) {
    data(i) = data(i+1)
  }
  /* Update metadata */
  data(used-1) = None
  used -= 1
}
def remove(target: Int): T = {
  /* Sanity-check inputs */
  if(target < 0 || target >= used) {
    throw new IndexOutOfBoundsException(target)
  }
  /* Shift elements left */
  for(i <- target until (used-1)) {
    data(i) = data(i+1)
  }
  /* Update metadata */
  data(used-1) = None
  used -= 1
}
def remove(target: Int): T = {
  /* Sanity-check inputs */
  if(target < 0 || target >= used) {
    throw new IndexOutOfBoundsException(target)
  }
  /* Shift elements left */
  for(i <- target until (used-1)) {
    data(i) = data(i+1)
  }
  /* Update metadata */
  data(used-1) = None
  used -= 1
}

What is the complexity?
O(data.size)

or
Θ(used - target)
Analysis of `remove(i)`

\[
T_{\text{remove}}(n) = \begin{cases} 
1 & \text{if } target = used - 1 \\
2 & \text{if } target = used - 2 \\
3 & \text{if } target = used - 3 \\
\vdots & \vdots \\
n - 1 & \text{if } target = 0
\end{cases}
\]
Analysis of $\text{remove}(i)$

$$T_{\text{remove}}(n) = \begin{cases} 
1 & \text{if } target = used - 1 \\
2 & \text{if } target = used - 2 \\
3 & \text{if } target = used - 3 \\
\vdots & \vdots \\
n - 1 & \text{if } target = 0 
\end{cases}$$

$T_{\text{remove}}(n)$ is $O(n)$ and $\Omega(1)$
def append(elem: T): Unit = {
    if(used == data.size){ /* Sad case 😞 */
        val newData = Array.copyOf(original = data, newLength = ???)
        for(i <- data.size until newData.size){ newData(i) = None }
    }
    /* Happy case 😊 */
    /* Append element, update data and metadata */
    newData(used) = Some(elem)
    data = newData
    used += 1
}
def append(elem: T): Unit = {
  if(used == data.size){ /* Sad case 😞 */
    /* assume newLength > data.size, but pick it later */
    val newData = Array.copyOf(original = data, newLength = ???)
    /* Array.copyOf doesn't init elements, so we have to */
    for(i <- data.size until newData.size){ newData(i) = None }
  }
  /* Happy case 😊 */
  /* Append element, update data and metadata */
  newData(used) = Some(elem)
  data = newData
  used += 1
}

What is the complexity?

...and what is newLength?
```scala
def append(elem: T): Unit = {
  if (used == data.size)
    /* Sad case 😞 */
    /* assume newLength > data.size, but pick it later */
    val newData = Array.copyOf(original = data, newLength = ???)
    /* Array.copyOf doesn't init elements, so we have to */
    for (i <- data.size until newData.size)
      newData(i) = None

  /* Happy case 😊 */
  /* Append element, update data and metadata */
  newData(used) = Some(elem)
  data = newData
  used += 1
}
```

*What is the complexity?*

\[ O(data.size) \text{ (ie } O(n) \text{)} \] ...but...
Analysis of `append(elem)`

\[ T_{append}(n) = \begin{cases} 
\ n & \text{if used} = n \\
\ 1 & \text{otherwise} 
\end{cases} \]
Analysis of \texttt{append(elem)}

\[ T_{\text{append}}(n) = \begin{cases} 
  n & \text{if used} = n \\
  1 & \text{otherwise}
\end{cases} \]

\[ T_{\text{append}}(n) \text{ is } O(n) \text{ and } \Omega(1) \]
Analysis of `append(elem)`

\[
T_{append}(n) = \begin{cases} 
n & \text{if used} = n \quad \text{🙁 case} \\
1 & \text{otherwise} \quad \text{😃 case} 
\end{cases}
\]

\[
T_{append}(n) \text{ is } O(n) \text{ and } \Omega(1)
\]

How often do we hit the 😞 case?
Analysis of \texttt{append(elem)}

\[
T_{\text{append}}(n) = \begin{cases} 
    n & \text{if used} = n \quad \frown case \\
    1 & \text{otherwise} \quad \smile case 
\end{cases}
\]

\[T_{\text{append}}(n) \text{ is } O(n) \text{ and } \Omega(1)\]

How often do we hit the \frown case? \textit{Depends on newLength}
A Note on Runtime Complexity

So far, when we've discussed runtime bounds we have done so without taking any extra information/context into account.

For example, the worst-case runtime of `ArrayBuffer.append` is $O(n)$.

We haven't considered the fact that oftentimes it is faster than $O(n)$.
A Note on Runtime Complexity

So far, when we've discussed runtime bounds we have done so without taking any extra information/context into account.

For example, the worst-case runtime of ArrayBuffer.append is $O(n)$

We haven't considered the fact that oftentimes it is faster than $O(n)$

We refer to this as the unqualified runtime...it is the runtime without any extra qualifications, caveats, etc
A Note on Runtime Complexity

So far, when we've discussed runtime bounds we have done so without taking any extra information/context into account.

For example, the worst-case runtime of ArrayBuffer.append is $O(n)$

We haven't considered the fact that oftentimes it is faster than $O(n)$

We refer to this as the **unqualified runtime**...it is the runtime without any extra qualifications, caveats, etc

*But, sometimes the extra context can be relevant... how can we include this in our analysis?*
Analysis of `append(elem)`

\[ T_{append}(n) = \begin{cases} 
  n & \text{if used = n} \\
  1 & \text{otherwise}
\end{cases} \]

\( T_{append}(n) \) is \( O(n) \) and \( \Omega(1) \)

How often do we hit the \( 😞 \) case?

*Depends on newLength...how should we calculate newLength?*
newLength = data.size + 1

Assume the initial size of our array is \(X\), and we want to insert \(n\) items in a row.
newLength = data.size + 1

For $n$ appends into an empty buffer...

While $\text{used} \leq \text{Initial\_Size}$:

$$\sum_{i=0}^{\text{IS}} \Theta(1)$$

And after:

$$\sum_{i=\text{IS}+1}^{\text{n}} \Theta(i)$$
newLength = data.size + 1

For $n$ appends into an empty buffer...

While $\text{used} \leq \text{Initial\_Size}$:

\[
\sum_{i=0}^{\text{IS}} \Theta(1)
\]

And after:

\[
\sum_{i=\text{IS}+1}^{n} \Theta(i)
\]

Total: $\Theta(n^2)$
newLength = data.size * 2
newLength = data.size * 2
newLength = data.size * 2

X * Θ(1) for X appends
newLength = data.size * 2

X * Θ(1) for X appends

Θ(1)  ...  Θ(1)  (X)Θ  Θ(1)  ...  Θ(1)  Θ(2)  Θ(1)  ...  Θ(1)  Θ(1)  ...

1  X  X+1  X+2  2X  2X+1  2X+2  n-1  n
newLength = data.size * 2

X * Θ(1) for X appends

Θ(X) + (X-1) * Θ(1) for X appends

Θ(X*2) + (X*2-1) * Θ(1) for X*2 appends

Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(4)

1 X X+1 X+2 2X 2X+1 2X+2 n-1 n
newLength = data.size * 2

So...how many red boxes for $n$ inserts?
newLength = data.size * 2

So...how many red boxes for n inserts? $\Theta(\log(n))$
newLength = data.size \times 2

So...how many red boxes for $n$ inserts? $\Theta(\log(n))$

How much work for box $j$?
newLength = data.size * 2

So...how many red boxes for $n$ inserts? $\Theta(\log(n))$

How much work for box $j$? $\Theta(IS \cdot 2^j) + \sum_{1}^{IS \cdot 2^j} \Theta(1)$
newLength = data.size * 2

So...how many red boxes for $n$ inserts? $\Theta(\log(n))$

How much work for box $j$?

$$\Theta(IS \cdot 2^j) + \sum_{i=1}^{IS \cdot 2^j} \Theta(1) = \Theta(2^j)$$
newLength = data.size * 2

So...how many red boxes for \( n \) inserts? \( \Theta(\log(n)) \)

How much work for box \( j \)?

\[ \Theta(IIS \cdot 2^j) + \sum_{1}^{IS \cdot 2^j} \Theta(1) = \Theta(2^j) \]

How much work for \( n \) inserts?
newLength = data.size * 2

So...how many red boxes for \( n \) inserts? \( \Theta(\log(n)) \)

How much work for box \( j \)?

\[
\Theta(IS \cdot 2^j) + \sum_{1}^{IS \cdot 2^j} \Theta(1) = \Theta(2^j)
\]

How much work for \( n \) inserts?

\[
\sum_{j=0}^{\Theta(\log(n))} \Theta(2^j)
\]
newLength = data.size * 2

So...how many red boxes for \( n \) inserts? \( \Theta(\log(n)) \)

How much work for box \( j \)?

\[ \Theta(IS \cdot 2^j) + \sum_{1}^{IS \cdot 2^j} \Theta(1) = \Theta(2^j) \]

How much work for \( n \) inserts?

\[ \sum_{j=0}^{\Theta(\log(n))} \Theta(2^j) \]

Total for \( n \) insertions: \( \Theta(n) \)
Amortized Runtime

append(elem) is $O(n)$

$n$ calls to append(elem) are $O(n)$
Amortized Runtime

$append(elem)$ is $O(n)$

$n$ calls to $append(elem)$ are $O(n)$

The cost of $n$ calls is guaranteed to be $O(n)$. 
Amortized Runtime

If $n$ calls to a function take $O(T(n))$...

We say the **Amortized Runtime** is $O(T(n) / n)$

The **amortized runtime** of `append` on an `ArrayBuffer` is: $O(n/n) = O(1)$

The **unqualified runtime** of `append` on an `ArrayBuffer` is: $O(n)$