CSE 250

Data Structures

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

Collections, Sequences and ADTs

Textbook Ch. 7.1, 1.7.2

Announcements

- PA1 due Sunday at midnight
  - Be aware that course staff is not guaranteed to be available after 5PM or on weekends

Sequences (what are they?)

- Examples

Fibonacci Sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Characters in a String: 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'

Lines in a File

People in a queue

Sequences (what are they?)

- Examples

Fibonacci Sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Characters in a String: 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'

Lines in a File

People in a queue

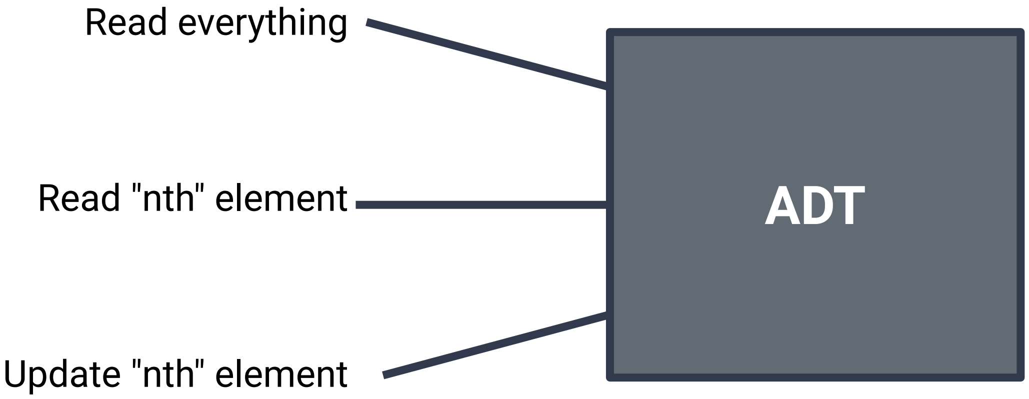
An "ordered" collection of elements

Sequences (what can you do with them?)

- Enumerate every element in sequence
  - ie: print out every element, sum every element
- Get the "nth" element
  - ie: what is the first element? what is the 42nd element?
- Modify the "nth" element
  - ie: set the first element to x, set the third element to y

Abstract Data Types (ADTs)

- The specification of what a data structure can do



Read everything

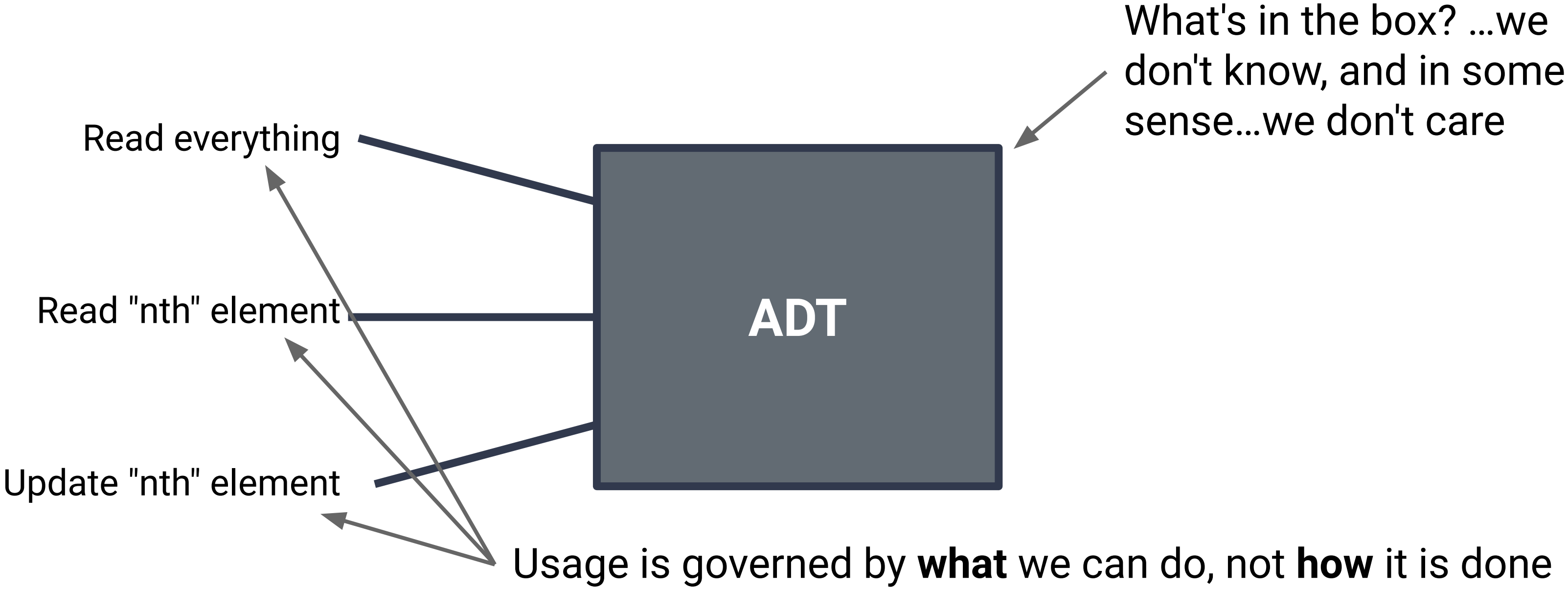
Read "nth" element

Update "nth" element

ADT

Abstract Data Types (ADTs)

- The specification of what a data structure can do



What's in the box? …we don't know, and in some sense…we don't care
Read everything
Read "nth" element
Update "nth" element
ADT

Usage is governed by **what** we can do, not **how** it is done

The **`Seq`** ADT

`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`

Get the number of elements in the seq

The `mutable.Seq` ADT

`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

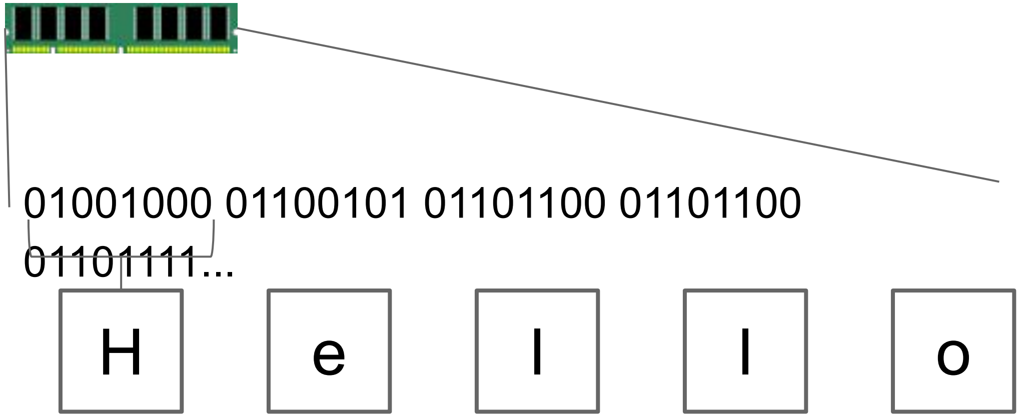
So…what's in the box?
(how do we implement it)

A Brief Aside on RAM (220 crossover)

A Brief Aside on RAM (220 crossover)

01001000 01100101 01101100 01101100 01101111...

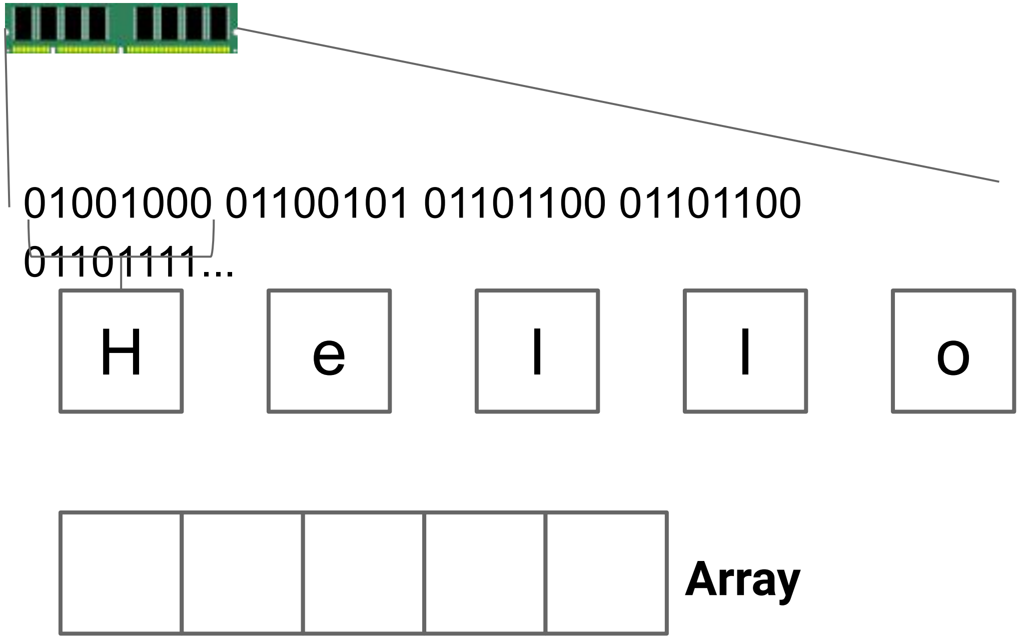
A Brief Aside on RAM (220 crossover)



01001000 01100101 01101100 01101100 01101111...

H e l l o

A Brief Aside on RAM (220 crossover)

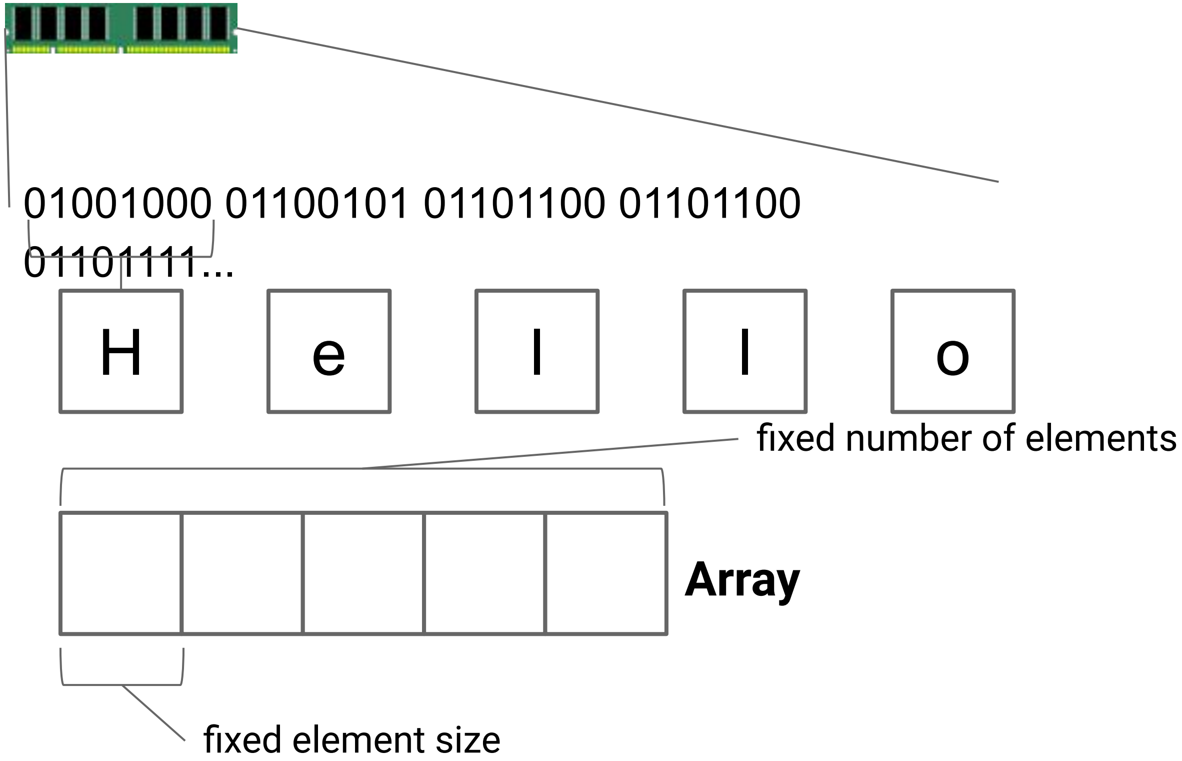


01001000 01100101 01101100 01101100 01101111...

H e l l o

Array

A Brief Aside on RAM (220 crossover)



01001000 01100101 01101100 01101100 01101111...

H e l l o

fixed number of elements

Array

fixed element size

RAM

`new T()`

Go find some unused part of memory that is big enough to fit a `T`, mark it as used, and return the ***address*** of that location in memory.

RAM

`new T()`

Go find some unused part of memory that is big enough to fit a `T`, mark it as used, and return the ***address*** of that location in memory.

```
var arr = new Array[Int](50)
```

The above code allocates 50 * 4 = 200 bytes of memory (a single Scala `Int` takes of 4 bytes in memory)

Element Access

```
var arr = new Array[Int](50)
```

If **`arr`** is at address *a*, where should you look for **`arr(19)`**?

Element Access

```
var arr = new Array[Int](50)
```

If **`arr`** is at address *a*, where should you look for **`arr(19)`**?

- *a* + 19 * 4   (does this computation depend on the size of **`arr`**?)

Element Access

```
var arr = new Array[Int](50)
```

If **`arr`** is at address *a*, where should you look for **`arr(19)`** ?

- *a* + 19 * 4   (does this computation depend on the size of **`arr`**?)
  - What is the complexity?

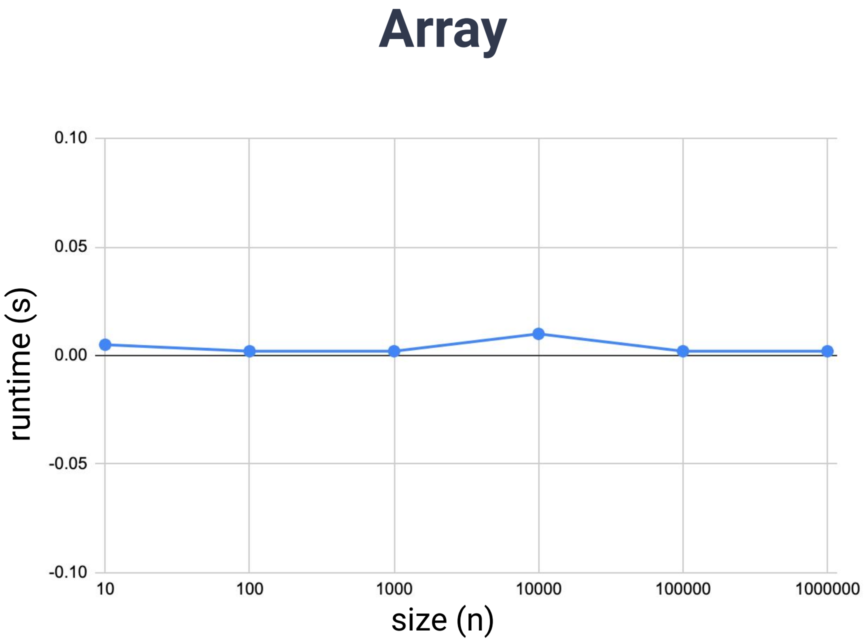
Element Access

```
var arr = new Array[Int](50)
```

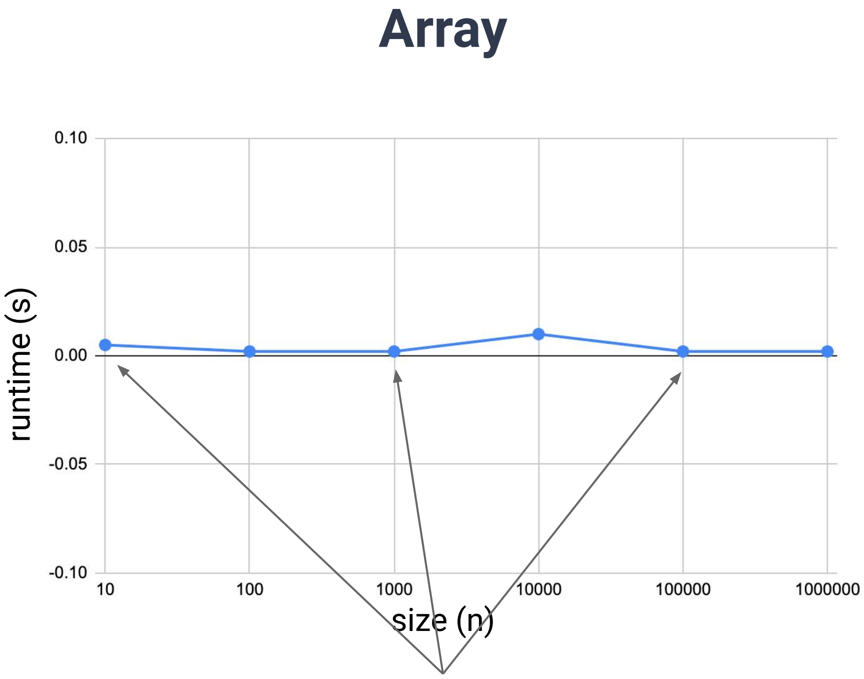
If `arr` is at address *a*, where should you look for `arr(19)`?

- *a* + 19 * 4   (does this computation depend on the size of `arr`?)
  - What is the complexity? **Θ(1)**

Random Access for an Array (Lecture 04)



Random Access for an Array (Lecture 04)



Notice how our runtime doesn't depend on the size of the array

Element Access

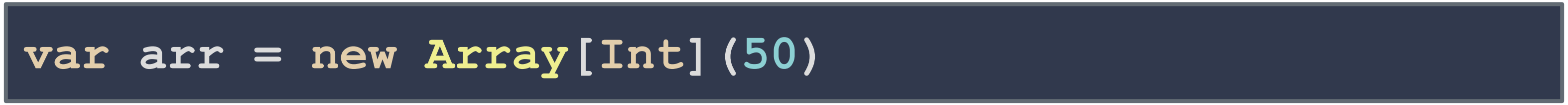
```
var arr = new Array[Int](50)
```

If **`arr`** is at address *a*, where should you look for **`arr(19)`**?

- *a* + 19 * 4   (a constant number of steps to compute…)

What about **`a(55)`**?

Element Access



```
var arr = new Array[Int](50)
```

If **`arr`** is at address *a*, where should you look for **`arr(19)`**?

- *a* + 19 * 4   (a constant number of steps to compute…)

What about **`a(55)`**?

- *a* + 55 * 4   …but that memory was not reserved for this array.
- Scala will prevent you from accessing an *out of bounds* element

`Array[T]:Seq[T]`

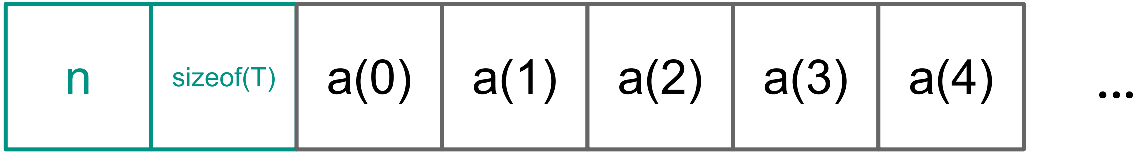
What does an **`Array`** of *n* items of type **`T`** actually look like?

- 4 bytes for *n* (optional)
- 4 bytes for **`sizeof(T)`** (optional)
- *n* * **`sizeof(T)`** bytes for the data

`Array[T]:Seq[T]`

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



`Array[T]:Seq[T]`

Given the structure of an **`Array`**, how would we implement the methods of the **`Seq`** ADT:

`apply(idx: Int): [A]`

Get the element (of type **`A`**) at position **`idx`**

`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

`Array[T]:Seq[T]`

Given the structure of an **`Array`**, how would we implement the methods of the **`Seq`** ADT:

`apply(idx: Int): [A]`
Get the element (of type **`A`**) at position **`idx`**

`length: Int`
Count the number of elements in the seq

Insert and remove don't make sense on arrays…

`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

How can we make it mutable?

IDEA: What if we reserve extra space?

`ArrayBuffer[T]:Buffer[T]`

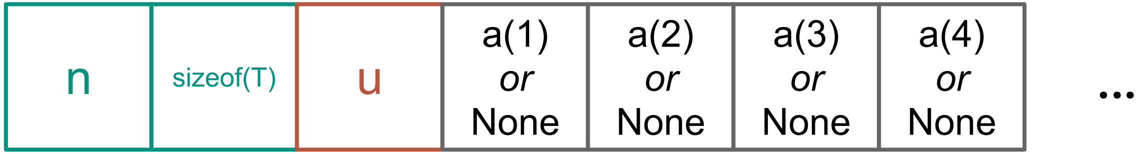
What does an **`ArrayBuffer`** of *n* items of type **`T`** actually look like?

- 4 bytes for *n* (optional)
- 4 bytes for **`sizeof(T)`** (optional)
- 4 bytes for the number of **used** fields
- *n* * **`sizeof(T)`** bytes for the data

`ArrayBuffer[T]:Buffer[T]`

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* * **`sizeof(T)`** bytes for the data



n	sizeof(T)	u	a(1) *or* None	a(2) *or* None	a(3) *or* None	a(4) *or* None	...