CSE 250

Data Structures

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

Lec 04: ADTs and Data Structures

Announcements

- AI Quiz, PA0, WA1 due Sunday @ 11:59PM
- Bring something to write with/on to recitations!
    - You will be turning in work for participation

Exercise:
Make a list of your favorite movies…

Abstract Data Type

The concept of a **List** is abstract…an ordered collection of things

How we choose to implement it is more concrete:

- Write it down on a piece of paper
- Write it on your computer (in what program?)
- Write it on your phone
- Make it in your head and remember it
- Sculpt it out of spaghetti noodles and glue

Abstract Data Type

Given a **List** we also have an idea of what we can do with it/ask about it:

- How many items are on the list?
- What are the items on the list?
- What is the first thing on the list? The last? The third?
- Where is "Halloween" on the list? Is it even on the list?
- Add something to the list/remove something from the list

Abstract Data Type

Details of our implementation will affect how we perform these tasks:

- Did we number the list? Then it's easier to find what is at a certain position/how many elements there are
- Did we write in pen, pencil, or digitally? That affects how easily we can change it
- How much space is on our paper? That affects how many items we can easily add

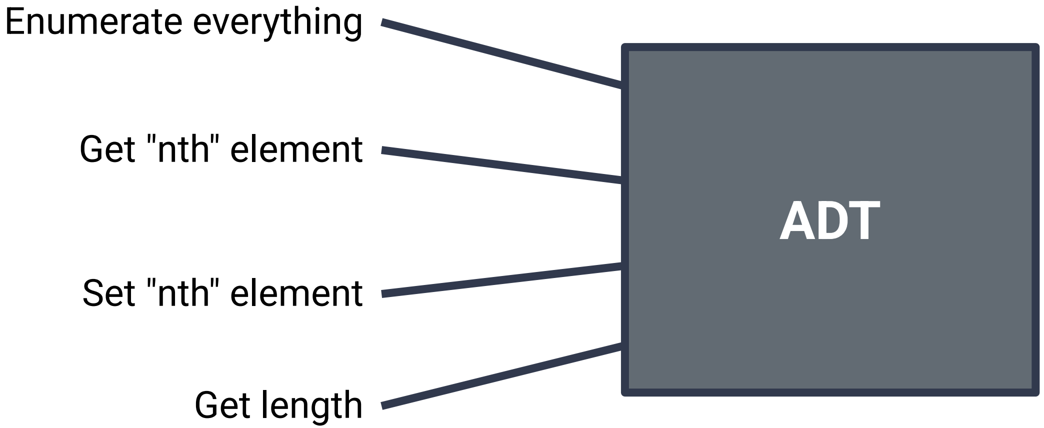
Abstract Data Type

Knowing what we will use the list for may guide our decisions

- What if I told you to write your **top 10** movies? Now we know we only need space for 10 things
- What if I told you to list everything you ate yesterday? Now we know the list will never have to change

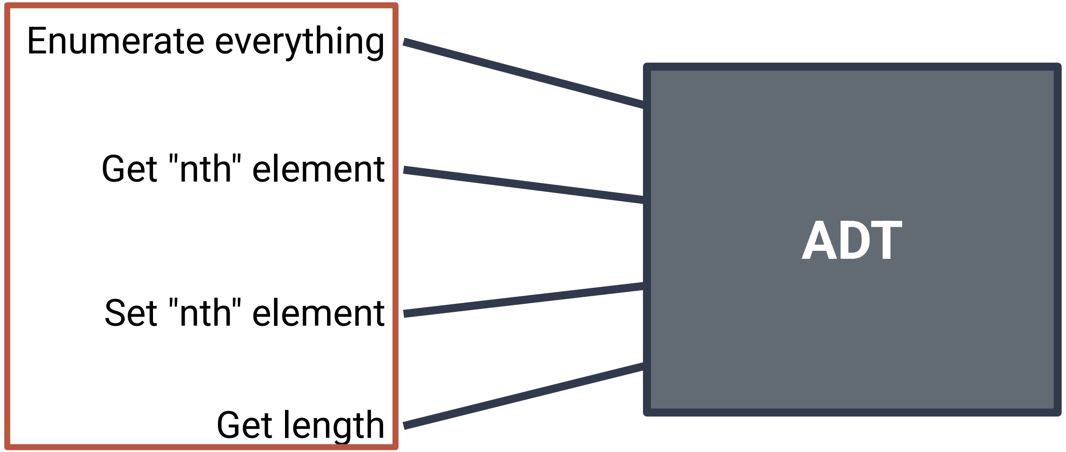
Abstract Data Types (ADTs)

The specification of **what** a data structure can do



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

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

Abstract Data Type vs Data Structure

ADT

The interface to a data structure

*Defines **what** the data structure can do*

Many data structures can implement the same ADT

Data Structure

The implementation of one (or more) ADTs

*Defines **how** the different tasks are carried out*

Different data structures will excel at different tasks

Abstract Data Type vs Data Structure

ADT

The interface to a data structure

*Defines **what** the*

can

Many data st

implement the same ADT

Data Structure

The implementation of one (or

) ADTs

different tasks are

ed out

Different data structures will excel at different tasks

The internal structure of our implementation and the conceptual model of our ADT do not have to be identical…more on this later

Collections

A **Collection** (of items) will be our most basic ADT

What can we do with a collection:

1. Get its size (the number of elements in it)
2. Enumerate the elements (iterate over the elements)

We aren't going to deal with collections directly, but instead look at a few more specific collection ADTs

Collection ADTs

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

Sequences (what are they?)

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?)

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 (of fixed size)

Sequences (what can you do with them?)

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
- Count how many elements you have

The Sequence ADT

`T get(int idx)`

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

`T set(int idx, T value)`

Set the element (of type **`T`**) at position **`idx`** to a new value

`int size()`

Get the number of elements in the seq

`Iterator<T> iterator()`

Get access to view all elements in the sequence, in order, once

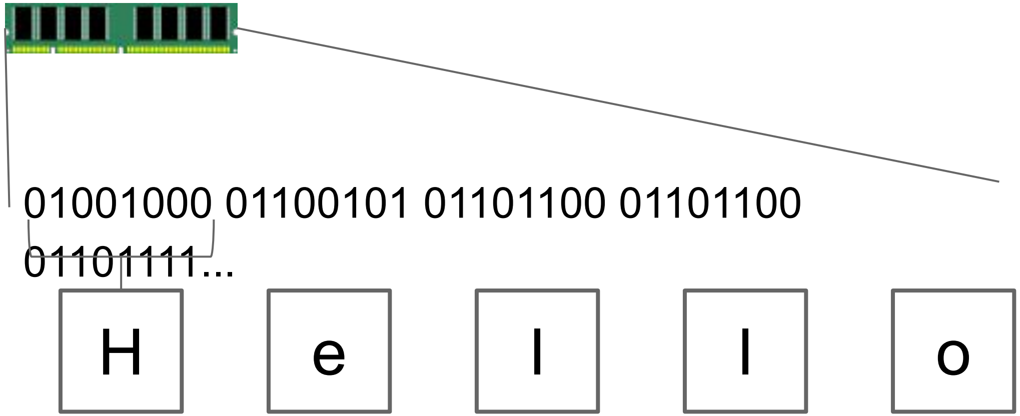
So…what's in the box?
(how do we implement it)

A Brief Aside on RAM (220 crossover)

A Brief Aside on RAM (220 crossover)

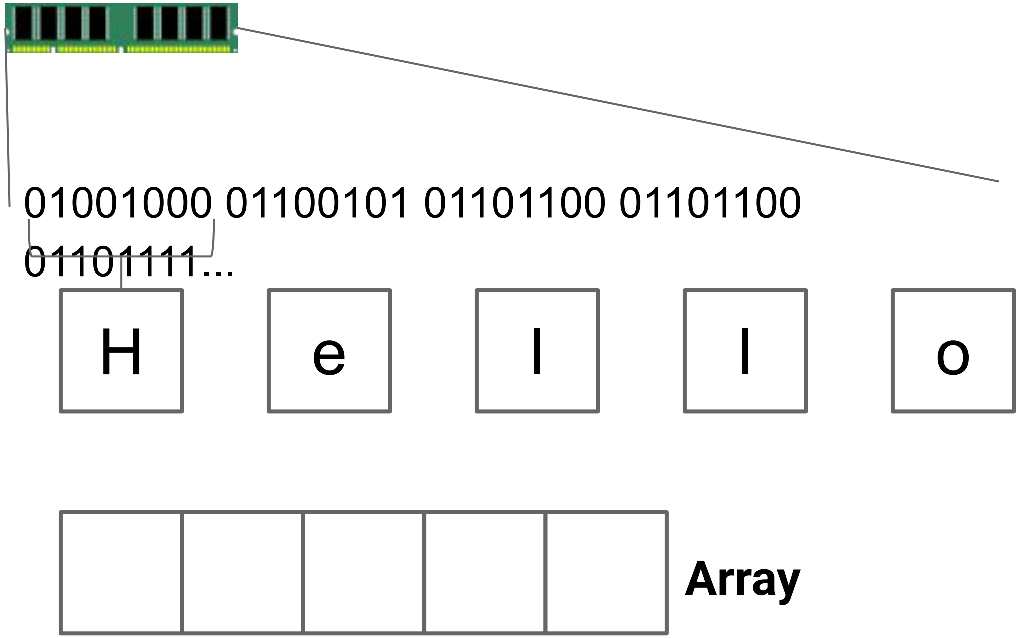
01001000 01100101 01101100 01101100 01101111...

A Brief Aside on RAM (220 crossover)



01001000 01100101 01101100 01101100 01101111...

A Brief Aside on RAM (220 crossover)

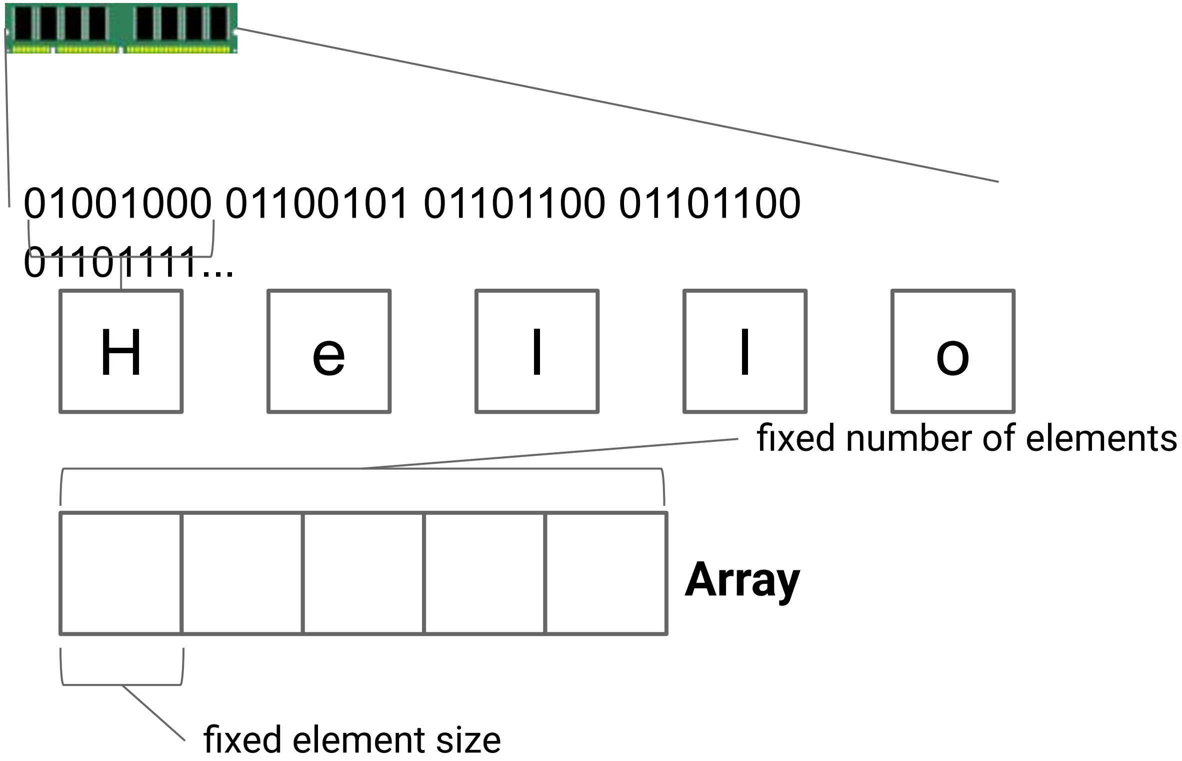


01001000 01100101 01101100 01101100 01101111...

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

Allocation with `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

Allocation with `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.

```java
int[] arr = new int[50];
```

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

** slightly more actually…see next slide*

Arrays in Detail

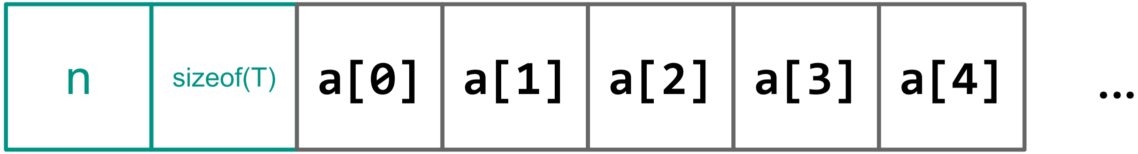
What does an array of *n* items of type `T` actually look like?

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

Arrays in Detail

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



Arrays in Detail

How would we implement the methods of the Sequence ADT for an Array:

```
T get(int idx)
```

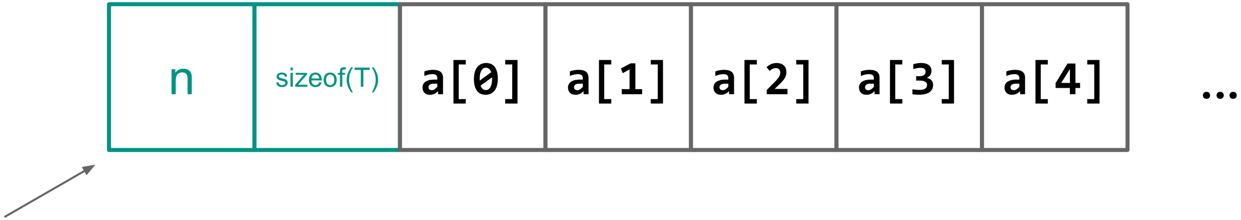
```
T set(int idx, T value)
```

```
int size()
```

Arrays in Detail

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



n

sizeof(T)

a[0]

a[1]

a[2]

a[3]

a[4]

...

The length is stored in the memory allocated for the array…

Arrays in Detail

How would we implement the methods of the Sequence ADT for an Array:

`T get(int idx)`

`T set(int idx, T value)`

`int size()`

Return the **`length`** field

Implementing get/set

```java
int[] arr = new int[50];
```

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

Implementing get/set

```java
int[] arr = new int[50];
```

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

- $a + 19 * 4$

Implementing get/set

```java
int[] arr = new 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`**?)

Implementing get/set

```
int[] arr = new 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**? **No**)

Implementing get/set

```
int[] arr = new 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`**? **No**)

What about **`a[55]`**?

Implementing get/set

```
int[] arr = new 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`**? **No**)

What about **`a[55]`**?

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

Arrays in Detail

How would we implement the methods of the Sequence ADT for an Array:

`T get(int idx)`

Compute the address of the element and return the value there

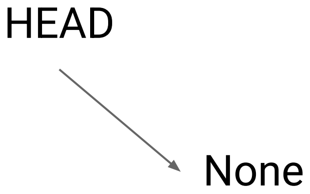
`T set(int idx, T value)`

Compute the address of the element and change the value there

`int size()`

Access the **`length`** field

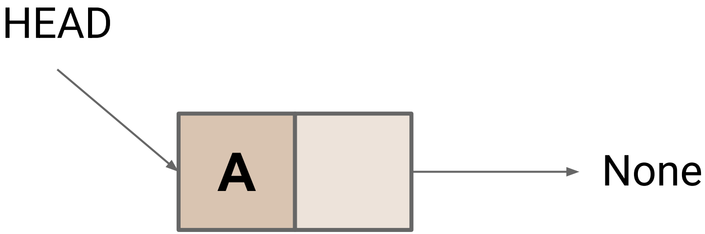
Linked Lists



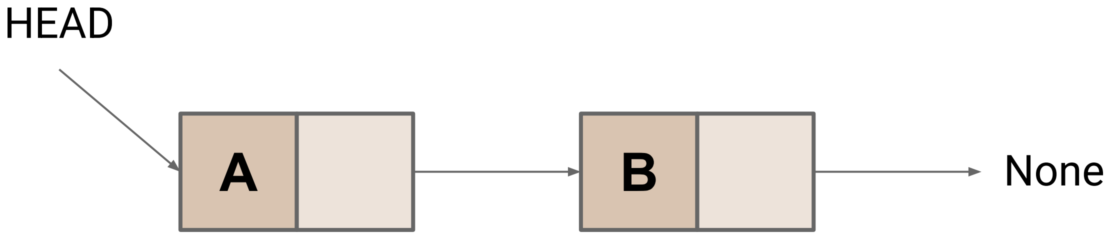
HEAD

None

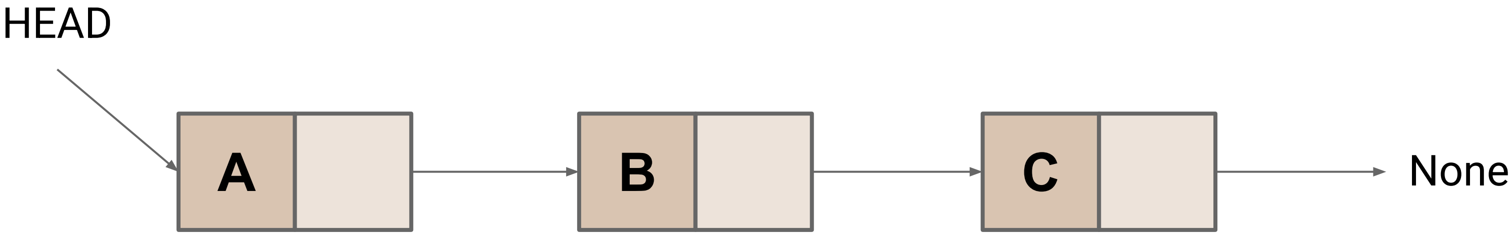
Linked Lists



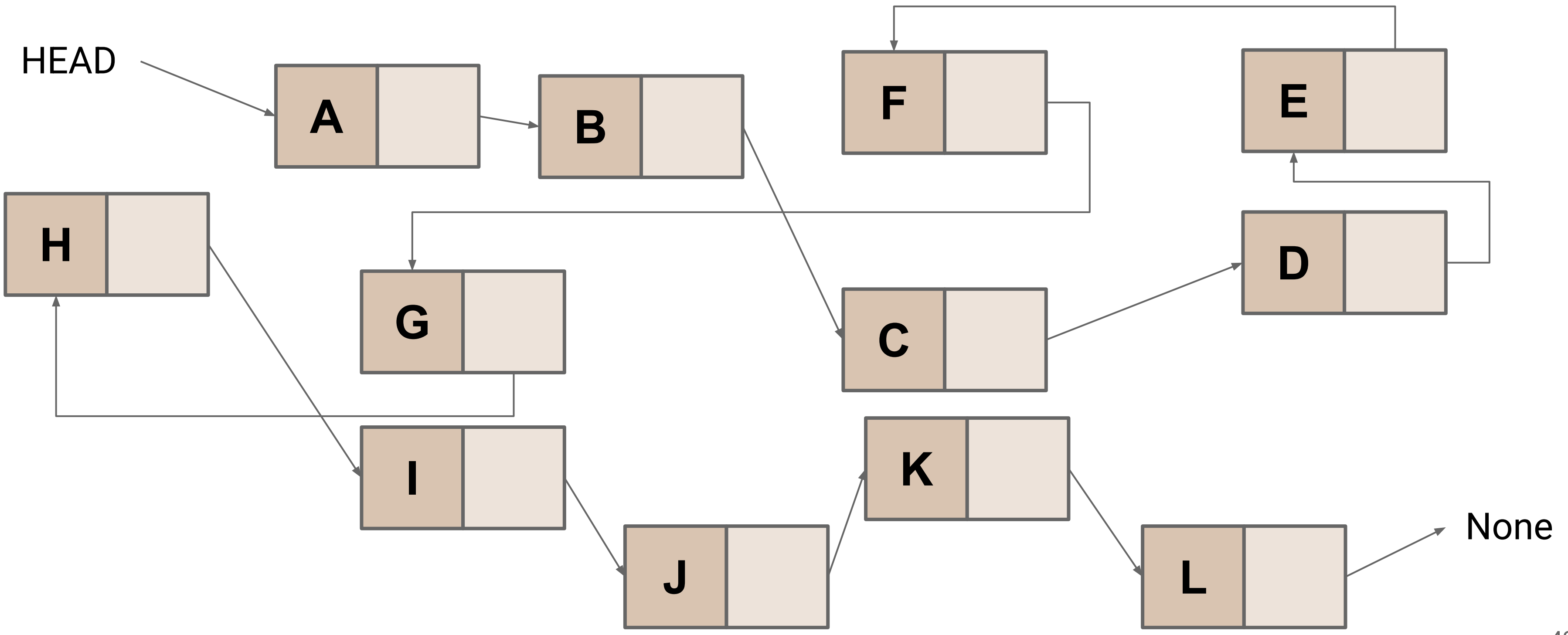
Linked Lists



Linked Lists



Linked Lists



Linked Lists in Detail

```
class LinkedList<T> {
  Optional<LinkedListNode<T>> head = Optional.empty();
  /* ... */
}
```

Class for our list, which right now just has a **`Optional`** reference to **`head`**

```
class LinkedListNode<T> {
  T value;
  Optional<LinkedListNode<T>> next = Optional.empty();
}
```

Class for a node in the list, which has a **`value`**, and an **`Optional`** reference to the **`next`** node

Linked Lists in Detail

```
class LinkedList<T> {
  Optional<LinkedListNode<T>> head = Optional.empty();
  /* ... */
}
```

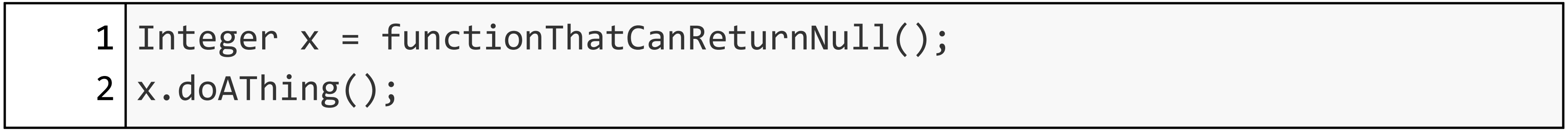
Class for our list, which right now just has a **`Optional`** reference to **`head`**

```
class LinkedListNode<T> {
  T value;
  Optional<LinkedListNode<T>> next = Optional.empty();
}
```

Class for a node in the list, which has a **`value`**, and an **`Optional`** reference to the **`next`** node

What is Optional<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?



```
Integer x = functionThatCanReturnNull();
x.doAThing();
```

What is Optional<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?

```
Integer x = functionThatCanReturnNull();
x.doAThing();
```

`java.lang.NullPointerException` (runtime error)

What is Optional<T>…a brief digression

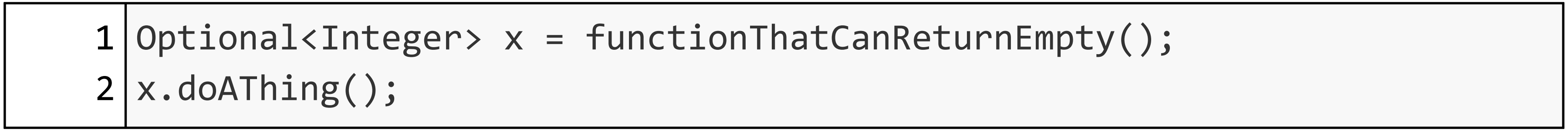
```
Integer x = functionThatCanReturnNull();
if (x == null) { /* do something special */ }
else { x.doAThing(); }
```

We need to add a check for **`null`** to avoid this…but this is easy to forget

What if our function returns **`Optional<Integer>`** instead?

What is Optional<T>...a brief digression

- Now our function returns **`Optional<Integer>`**
- What can go wrong in the following code snippet?



```
Optional<Integer> x = functionThatCanReturnEmpty();
x.doAThing();
```

What is Optional<T>...a brief digression

- Now our function returns **`Optional<Integer>`**
- What can go wrong in the following code snippet?

```
Optional<Integer> x = functionThatCanReturnEmpty();
x.doAThing();
```

`Cannot resolve method doAThing() in Optional`

(compile error)

What is Optional<T>...a brief digression

```java
Optional<Integer> x = functionThatCanReturnNull();
if (x.isPresent()) { x.get().doAThing(); }
else { /* do something special */ }
```

Java makes us do something sensible!

What is Option[T]...a brief digression

Creating `Optional` objects:

```
Optional.empty()       // Like null
Optional.of(x)         // Optional object w with value x
Optional.ofNullable(x) // If x is null same as .empty()
```

Using `Optional` objects:

```
.isPresent()        // True if there is a value
.get()              // gets the value
.orElse(y)          // return value if present, y if not
```

Linked Lists in Detail

How do we implement the methods of the Sequence ADT for a Linked List:

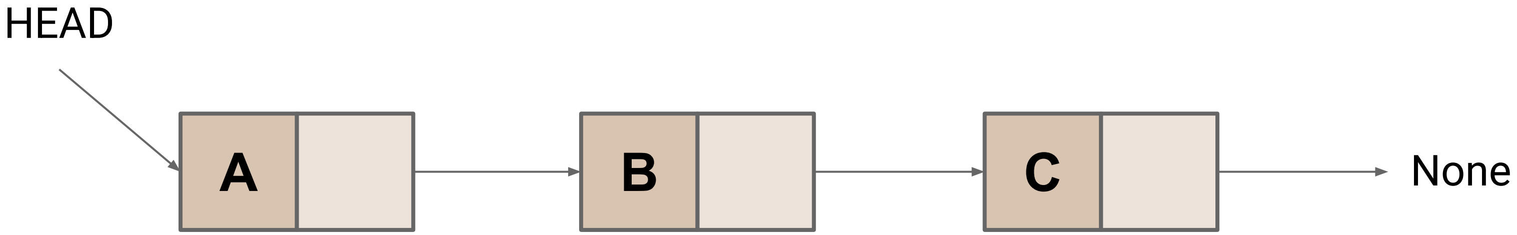
```
T get(int idx)
```

```
T set(int idx, T value)
```

```
int length
```

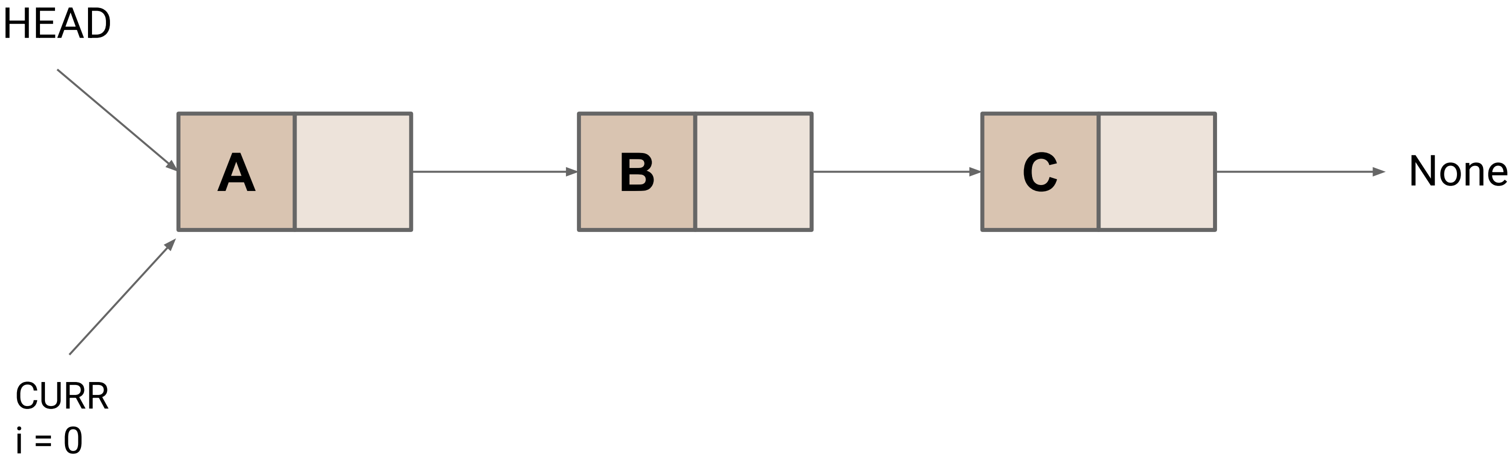
Implementing get/set

`get(2)`



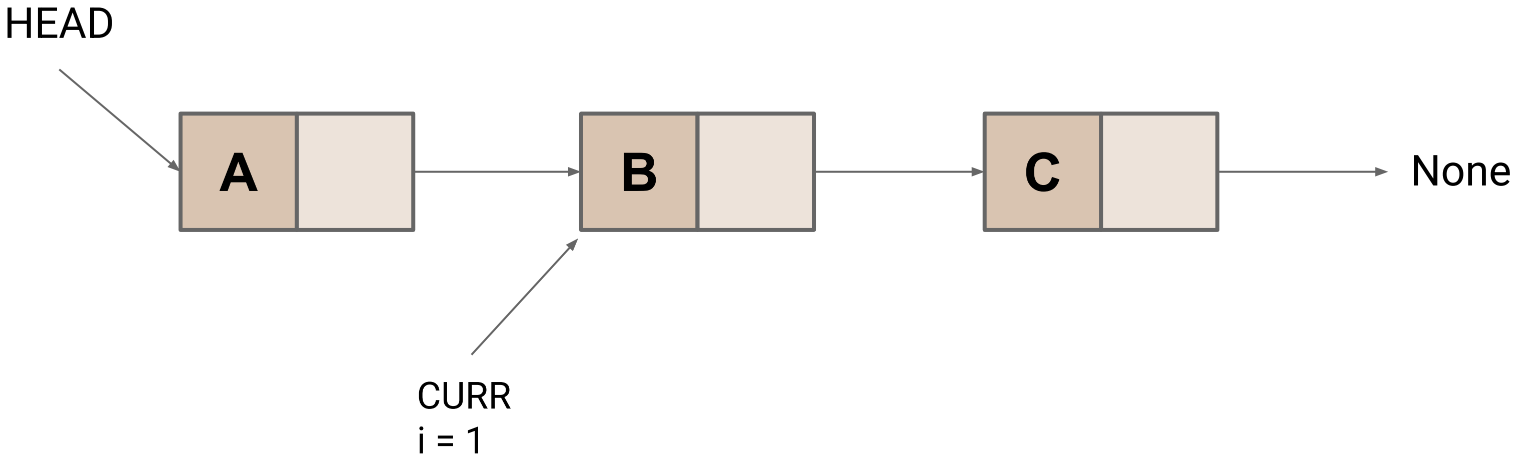
Implementing get/set

`get(2)`



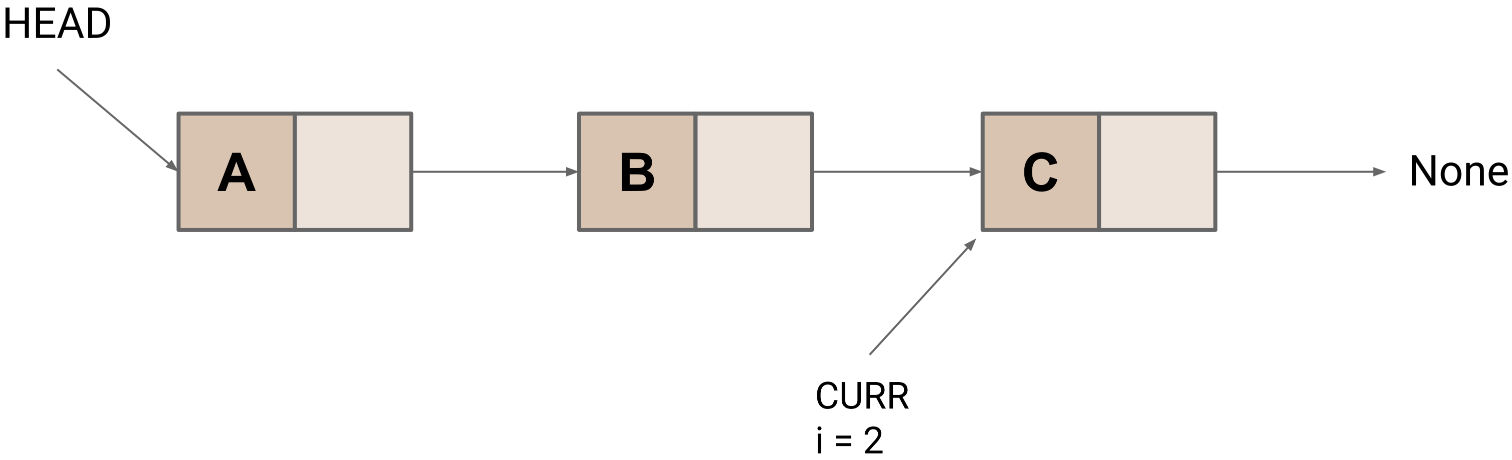
Implementing get/set

`get(2)`



Implementing get/set

`get(2)`



Implementing get/set

```java
public T get(int idx) {
  int i = 0;
  Optional<LinkedListNode<T>> curr = head;
  while(i < idx) {
    if (!curr.isPresent()) { throw new IndexOutOfBoundsException(); }
    i++;
    curr = curr.get().next;
  }
  if(!curr.isPresent()) { throw new IndexOutOfBoundsException(); }
  return curr.get().value;
}
```

Linked Lists in Detail

How do we implement the methods of the Sequence ADT for a Linked List:

`T get(int idx)`

Go node-by-node until you reach **`idx`** and return the value of that node

`T set(int idx, T value)`

Go node-by-node until you reach **`idx`** and set the value of that node

`int size()`

Implementing size

```java
public int size() {
    int i = 0;
    Optional<LinkedListNode<T>> curr = head;
    while(curr.isPresent()) { i++; curr = curr.get().next; }
    return i;
}
```

Implementing `length`

Alternate Idea: Have the Linked List class store the length

```
class LinkedList<T> {
  Optional<LinkedListNode<T>> head = Optional.empty();
  int length;
  /* ... */
}
```

Why might this be a good idea?

Implementing `length`

Alternate Idea: Have the Linked List class store the length

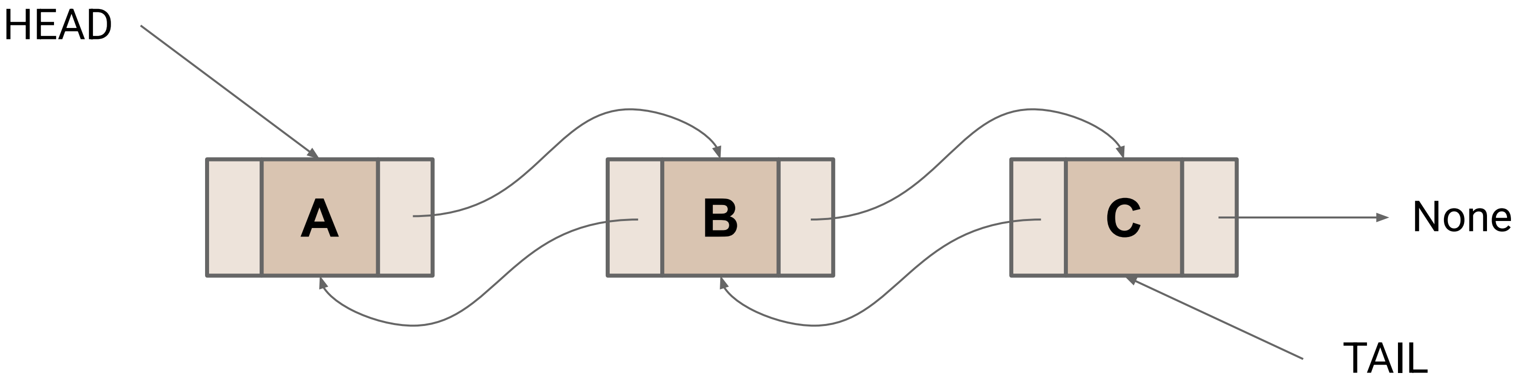
```java
class LinkedList<T> {
  Optional<LinkedListNode<T>> head = Optional.empty();
  int length;
  /* ... */
}
```

Why might this be a good idea? Faster…?

What do we mean by faster? How much faster? How do we quantify that?

Doubly Linked Lists

- Can also be doubly linked (a next AND a prev pointer per node)
- PA1 will have you implementing a **Sorted Doubly Linked List** with some minor twists



Doubly Linked Lists

```
class LinkedList<T> {
  Optional<LinkedListNode<T>> head = Optional.empty();
  Optional<LinkedListNode<T>> tail = Optional.empty();
  int length;
}
```

```
class LinkedListNode<T> {
  T value;
  Optional<LinkedListNode<T>> next = Optional.empty();
  Optional<LinkedListNode<T>> prev = Optional.empty();
}
```