CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

9: Stacks, Queues, and Iterators

Alexandra Papoutsaki
she/her/hers

Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators

Some slides adopted from Algorithms 4th Edition and Oracle tutorials

Stacks

- Dynamic linear data structures.
- Items are inserted and removed following the LIFO paradigm.
- LIFO: Last In, First Out.
- Similar to lists, there is a sequential nature to the data.
- Remove the most recent item.

- Metaphor of cafeteria plate dispenser.
  - Want a plate? Pop the top plate.
  - Add a plate? Push it to make it the new top.
  - Want to see the top plate? Peek.
  - We want to make push and pop as time efficient as possible

Example of stack operations

push	To	be	or	not	to	-	be	-	-	that	-	-	-	is
pop						to		be	not		that	or	be	

push to top

pop from top

Last In First Out

Implementing stacks with ArrayLists

- Where should the top go to make push and pop as efficient as possible?
- The *end/rear* represents the top of the stack.
- To push an item `add(Item item)`.
  - Adds at the end. Average $O(1)$.
- To pop an item `remove()`.
  - Removes and returns the item from the end. Average $O(1)$.
- To peek `get(size()-1)`.
  - Retrieves the last item. $O(1)$.
- If the front/beginning were to represent the top of the stack, then:
  - Push, pop would be $O(n)$ and peek $O(1)$.

Implementing stacks with singly linked lists

- Where should the top go to make push and pop as efficient as possible?
- The *head* represents the top of the stack.
- To push an item `add(Item item)`.
  - Adds at the head. $O(1)$.
- To pop an item `remove()`.
  - Removes and retrieves from the head. $O(1)$.
- To peek `get(0)`.
  - Retrieves the head. $O(1)$.
- If the *tail* were to represent the top of the stack, then:
  - Push, pop, peek would all be $O(n)$.

Implementing stacks with doubly linked lists

- Where should the top go to make push and pop as efficient as possible?
- The head represents the top of the stack.
- To push an item `addFirst(Item item)`.
  - Adds at the head. $O(1)$.
- To pop an item `removeFirst()`.
  - Removes and retrieves from the head. $O(1)$.
- To peek `get(0)`.
  - Retrieves the head's item. $O(1)$.
- Unnecessary memory overhead with extra pointers.
- If the *tail* were to represent the top of the stack, we'd need to use `addLast(Item item)`, `removeLast()`, and `get(size()-1)` to have $O(1)$ complexity.

Implementation of stacks

- `Linear.java`: simple interface with `add`, `remove`, `peek`, `isEmpty`, and `size` methods.
- `Stack.java`: simple interface with `push`, `pop`, `peek`, `isEmpty`, and `size` methods. Extends `Linear` interface.
- `ArrayListStack.java`: for implementation of stacks with ArrayLists. Must implement methods of `Stack` interface (and as a consequence of `Linear` interface).
- `LinkedStack.java`: for implementation of stacks with singly linked lists. Must implement methods of `Stack` interface (and as a consequence of `Linear` interface).

Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators

Queues

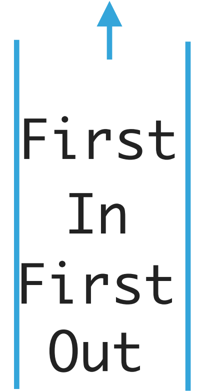
- Dynamic linear data structures.
- Items are inserted and removed following the FIFO paradigm.
- FIFO: First In, First Out.
- Similar to lists, there is a sequential nature to the data.
- Remove the *least* recent item.

- Metaphor of a line of people waiting to buy tickets.
- Just arrived? Enqueue person to the end of line.
- First to arrive? Dequeue person at the top of line.
- We want to make enqueue and dequeue as time efficient as possible.

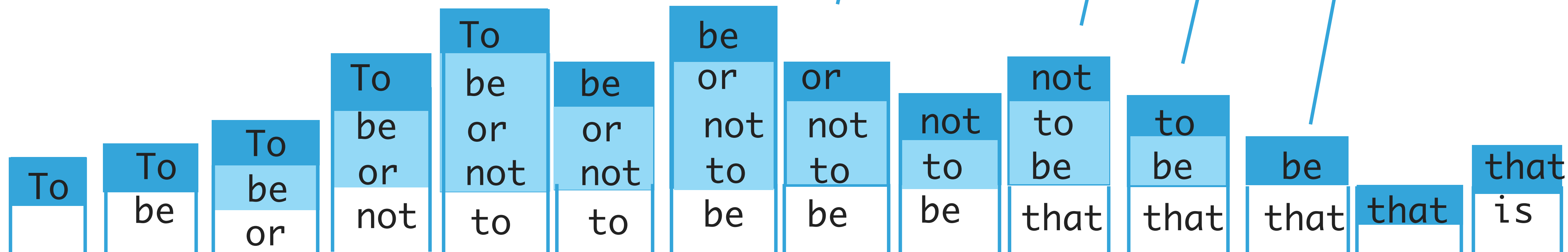
Example of queue operations

enqueue	To	be	or	not	to	-	be	-	-	that	-	-	-	is
dequeue						To		be	or		not	to	be	

dequeue from beginning



enqueue at end



Implementing queue with ArrayLists

- Where should we enqueue and dequeue items?
- To enqueue an item `add()` at the end of arrayList. Average $O(1)$.
- To dequeue an item `remove(0)`. $O(n)$.
- What if we add at the beginning and remove from end?
  - Now dequeue is cheap ($O(1)$) but enqueue becomes expensive ($O(n)$).

Implementing queue with singly linked list

- Where should we enqueue and dequeue items?
    - To enqueue an item `add()` at the *head* of SLL ($O(1)$).
    - To dequeue an item `remove(size()-1)` ($O(n)$).
- What if we add at the end and remove from beginning?
    - Now dequeue is cheap ($O(1)$) but enqueue becomes expensive ($O(n)$).
- $O(1)$ if we have a tail pointer.
    - Simple modification in code, big gains!
    - Version that recommended textbook follows.

Implementing queue with doubly linked list

- Where should we enqueue and dequeue items?
  - To enqueue an item `addLast()` at the tail of DLL ($O(1)$).
  - To dequeue an item `removeFirst()` ($O(1)$).
  - What if we add at the head and remove from tail?
    - Both are $O(1)$!

Implementation of queues

- `Linear.java`: simple interface that ensures that we can use stacks and queues interchangeably through the `add`, `remove`, `peek`, `isEmpty`, and `size` methods.
- `Queue.java`: simple interface with `enqueue`, `dequeue`, `peek`, `isEmpty`, and `size` methods. Extends `Linear` interface.
- `ArrayListQueue.java`: for implementation of queues with ArrayLists. Must implement methods of `Queue` interface (and as a consequence of `Linear` interface).
- `LinkedQueue.java`: for implementation of queues with doubly linked lists. Must implement methods of `Queue` interface (and as a consequence of `Linear` interface).

Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators

Stack applications

- Java Virtual Machine.
- Basic mechanisms in compilers, interpreters (see CS101).
- Back button in browser.
- Undo in word processor.
- Infix expression evaluation (Dijskstra’s algorithm with two stacks).
- Postfix expression evaluation.

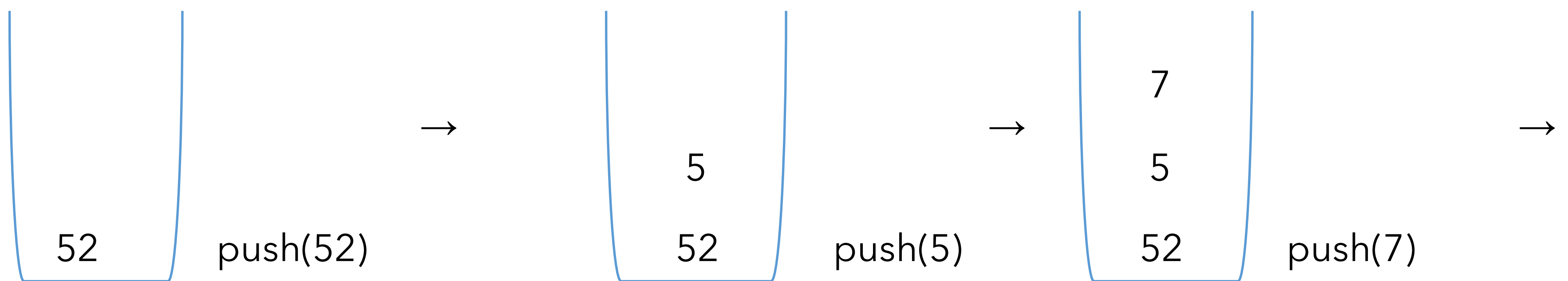
http://algs4.cs.princeton.edu

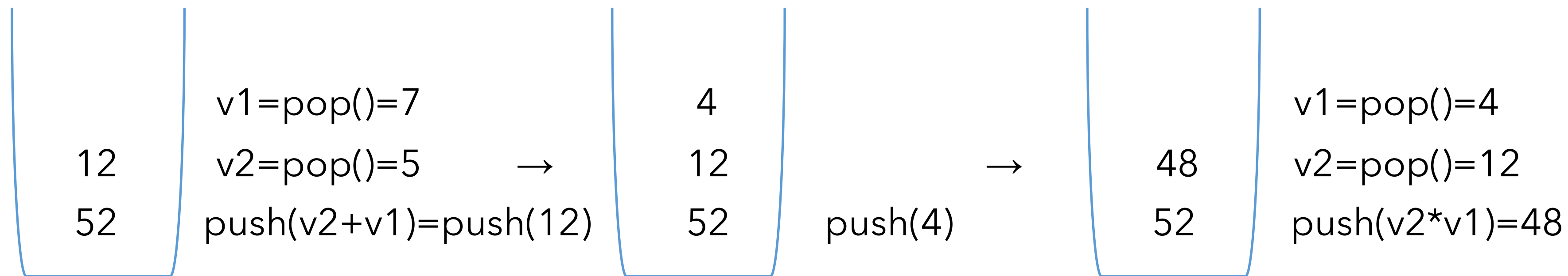
1.3 DIJKSTRA'S 2-STACK DEMO

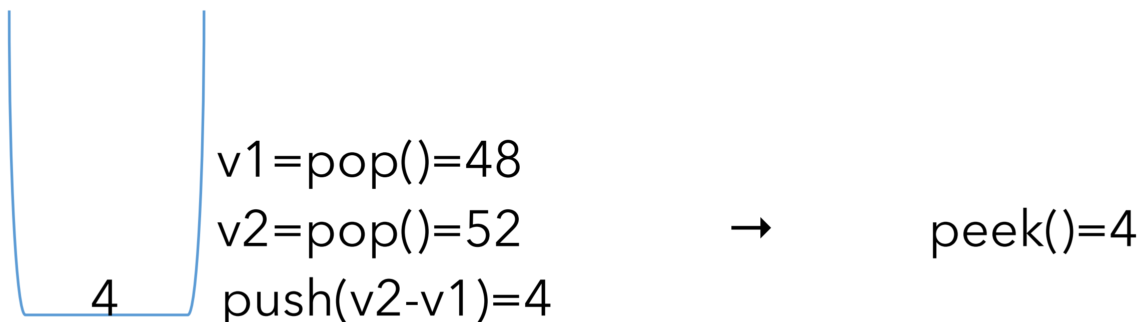
https://algs4.cs.princeton.edu/lectures/demo/13DemoDijkstraTwoStack.mov

Postfix expression evaluation example

Example: (52 - ((5 + 7) * 4) ⇒ 52 5 7 + 4 * -







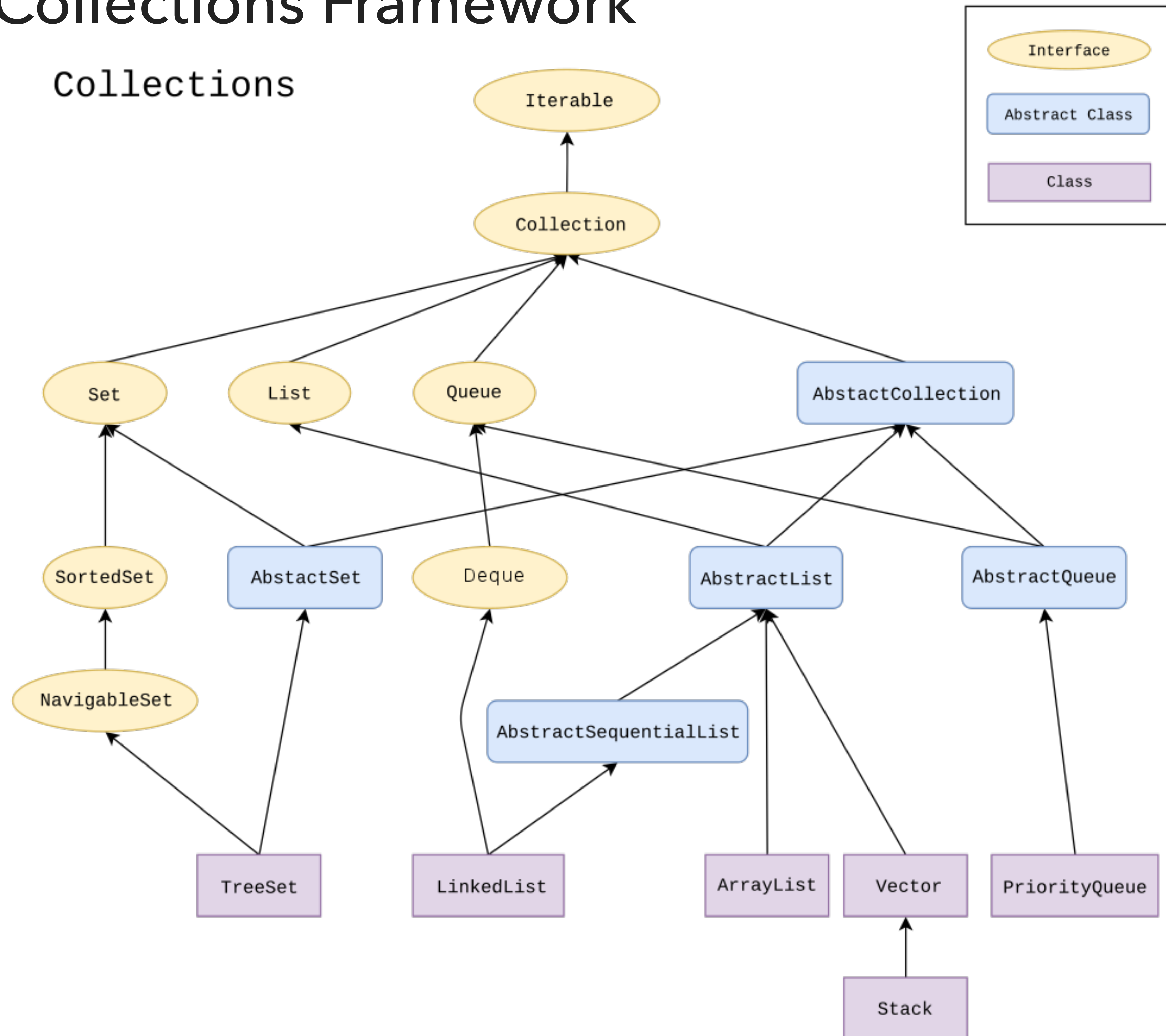
Queue applications

- Spotify playlist.
- Data buffers (netflix, Hulu, etc.).
- Asynchronous data transfer (file I/O, sockets).
- Requests in shared resources (printers).
- Traffic analysis.
- Waiting times at calling center.

Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators

The Java Collections Framework



https://en.wikipedia.org/wiki/Java_collections_framework

Deque in Java Collections

- Do not use `Stack`. Obsolete class.
- `Queue` is an interface…
- It's recommended to use the `Deque` interface instead.
  - Double-ended queue (can add and remove from either end).

```
java.util.Deque;

public interface Deque<E> extends Queue<E>
```

- You can choose between `LinkedList` and `ArrayDeque` implementations.
  - `Deque deque = new ArrayDeque(); //preferable`

https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html

Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators

`Iterator` Interface

- Interface that allows us to traverse a collection one element at a time.

```
public interface Iterator<E> {
  //returns true if the iteration has more elements
  //that is if next() would return an element instead of throwing an exception
  boolean hasNext();

  //returns the next element in the iteration
  //post: advances the iterator to the next value
  E next();

  //removes the last element that was returned by next
  default void remove(); //optional, better avoid it altogether
}
```

https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Iterator Example

```
List<Integer> myList = new ArrayList<Integer>();
//… operations on myList

Iterator<Integer> listIterator = myList.iterator();
while(listIterator.hasNext()){
  Integer elt = listIterator.next();
  System.out.println(elt);
}
```

Java8 introduced lambda expressions

- `Iterator` interface now contains a new method.
- `default void forEachRemaining(Consumer<? super E> action)`
- Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
- `listIterator.forEachRemaining(s -> System.out.println(s));`
- or
- `listIterator.forEachRemaining(System.out::println);`

Iterable Interface

- Interface that allows an object to be the target of a for-each loop:

```
for(String elt: myList){
  System.out.println(elt);
}
```

```
interface Iterable<E>{
  //returns an iterator over elements of type E
  Iterator<E> iterator();

  //Performs the given action for each element of the Iterable until all elements have
  //been processed or the action throws an exception.
  default void forEach(Consumer<? super E> action);
}
myList.forEach(elt-> System.out.println(elt));
myList.forEach(System.out::println);
```

https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html

How to make your data structures iterable?

1. Implement `Iterable` interface.
2. Make a private class that implements the `Iterator` interface.
3. Override `iterator()` method to return an instance of the private class.

Example: making ArrayList iterable

```
public class ArrayList<Item> implements Iterable<Item> {
     //…
     public Iterator<Item> iterator() {

          return new ArrayListIterator();
     }


     private class ArrayListIterator implements Iterator<Item> {

          private int i = 0;

          public boolean hasNext() {
               return i < size;

          }

          public Item next() {

               return data[i++];

          }

          public void remove() {
               throw new UnsupportedOperationException();

          }

     }
```

Traversing ArrayList

- All valid ways to traverse ArrayList and print its elements one by one.

```
// because it implements the Iterable interface
for(int elt:myList) {
     System.out.println(elt);
 }

// because it implements the Iterable interface
myList.forEach(elt -> System.out.println(elt));
myList.forEach(elt -> {System.out.println(elt);});
myList.forEach(System.out::println);

// because it contains a private class that implements the Iterator interface
Iterator<Integer> listIterator = myList.iterator();
while(listIterator.hasNext()){
     Integer elt = listIterator.next();
     System.out.println(elt);
 }

 // because it contains a private class that implements the Iterator interface
Iterator<Integer> listIterator = myList.iterator();
listIterator.forEachRemaining(elt-> System.out.println(elt));
listIterator.forEachRemaining(elt->{System.out.println(elt);});
listIterator.forEachRemaining(System.out::println);
```

Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators

Readings:

- Oracle's guides:
  - Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
  - Deque: https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html
  - ArrayList: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
  - Iterator: https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
  - Iterable: https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html
- Recommended Textbook:
  - Chapter 1.3 (Page 126-157)
- Recommended Textbook Website:
  - Stacks and Queues: https://algs4.cs.princeton.edu/13stacks/

Code

- Lecture 9 code

Practice Problems:

- 1.3.2-1.3.8, 1.3.32-1.3.33