BASIC DATA STRUCTURES

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

9: Linked Lists Catchup + Stacks and Queues

Tom Yeh
he/him/his

ArrayList Review

Worst-case performance of `add()` is $O(n)$

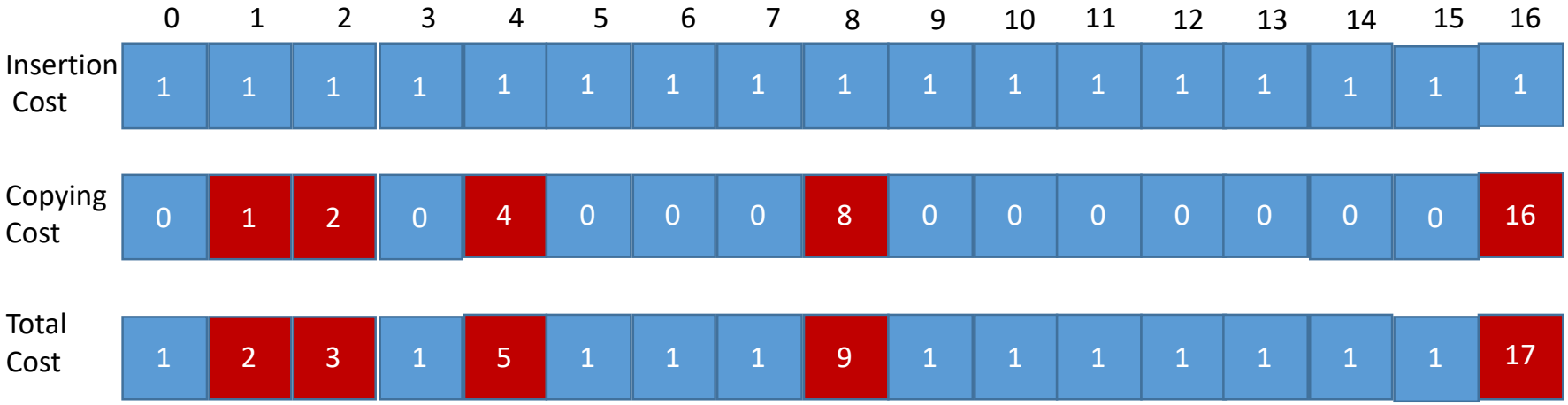
- Cost model: 1 for insertion, n for copying n items to a new array.
- Worst-case: If ArrayList is full, `add()` will need to call `resize` to create a new array of double the size, copy all items, insert new one.
- Total cost: $n + 1 = O(n)$.

- Realistically, this won't be happening often and worst-case analysis can be too strict. We will use amortized time analysis instead.

Amortized analysis

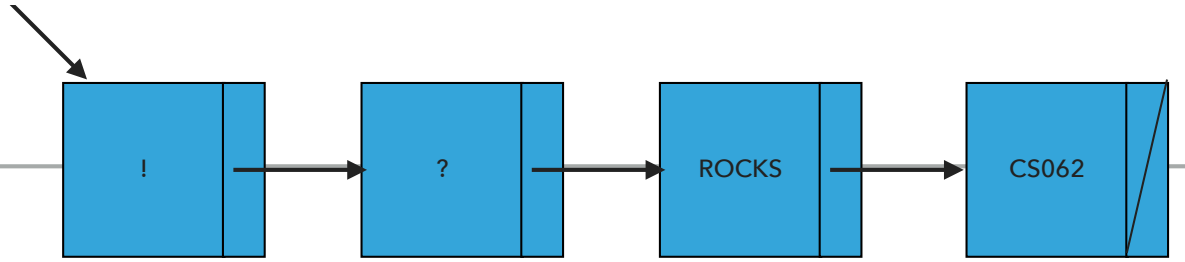
- Amortized cost per operation: for a sequence of n operations, it is the total cost of operations divided by n.
  - Simplest form of amortized analysis called aggregate method. More complicated methods exist, such as accounting (banking) and potential (physicist's).

Amortized analysis for n add() operations



- As the ArrayList increases, doubling happens *half as often* but *costs twice as much.*
- O(total cost)= $\sum$("cost of insertions") + $\sum$("cost of copying")
- $\sum$("cost of insertions") $= n$.
- $\sum$("cost of copying") = $1 + 2 + 2^2 + \ldots 2^{\lfloor \log 2^n \rfloor} \leq 2n$.
- O(total cost) $\leq 3n$, therefore amortized cost is $\leq \frac{3n}{n} = 3 = O(1)$, but "lumpy".

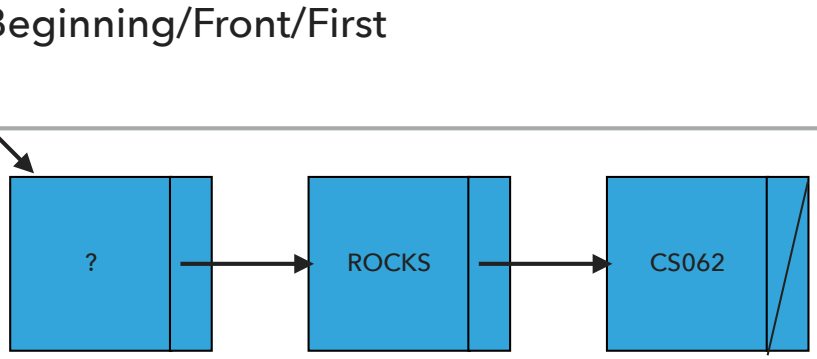
Quiz



Insert item at a specified index

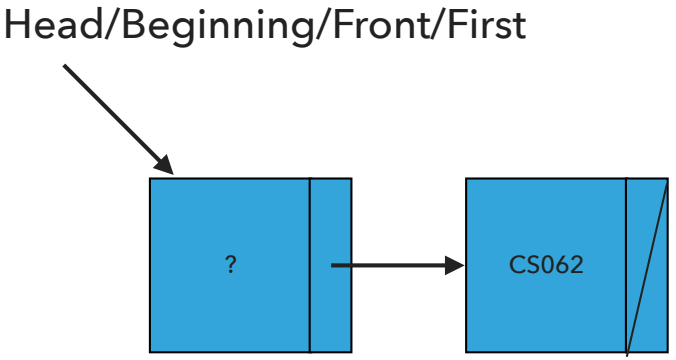
```
    // Inserts the specified item at the specified index.
    public void add(int index, Item item) {
            // check that index is within range
        rangeCheck(index);
            // if index is 0, then call one-argument add
        if (index == 0) {
            add(item);
            // else
        } else {
                // make two pointers, previous and finger. Point previous to null and finger to
head
            Node previous = null;
            Node finger = first;
            // search for index-th position by pointing previous to finger and advancing finger
            while (index > 0) {
                previous = finger;
                finger = finger.next;
                index--;
            }
            // create new node to insert in correct position. Set its pointers and contents
            Node current = new Node();
            current.next = finger;
            current.item = item;
            // make previous point to newly created node.
            previous.next = current;
            // increase number of nodes
            n++;
        }
    }
```

Retrieve and remove head




```
/**
     * Retrieves and removes the head of the singly linked list.
     *
     * @return the head of the singly linked list.
     */
    public Item remove() {
        // Make a temporary pointer to head
        Node temp = first;
        // Move head one to the right
        first = first.next;
        // Decrease number of nodes
        n--;
        // Return item held in the temporary pointer
        return temp.item;
    }
```

Retrieve and remove element from a specific index




```
//Retrieves and removes the item at the specified index.
public Item remove(int index) {
       // check that index is within range
     rangeCheck(index);
       // if index is 0, then call remove
     if (index == 0) {
          return remove();
       // else
     } else {
       // make two pointers, previous and finger. Point previous to null and finger to head
          Node previous = null;
          Node finger = first;
          // search for index-th position by pointing previous to finger and advancing finger
          while (index > 0) {
               previous = finger;
               finger = finger.next;
               index--;
          }
          // make previous point to finger's next
          previous.next = finger.next;
          // reduce number of items
          n--;
          // return finger's item
          return finger.item;
     }

}
```

add() in singly linked lists is $O(1)$ for worst case

```
public void add(Item item) {
   // Save the old node
   Node oldfirst = first;

   // Make a new node and assign it to head. Fix pointers.
   first = new Node();
   first.item = item;
   first.next = oldfirst;

   n++; // increase number of nodes in singly linked list.
}
```

get() in singly linked lists is $O(n)$ for worst case

```
public Item get(int index) {
    rangeCheck(index);

    Node finger = first;
    // search for index-th element or end of list
    while (index > 0) {
        finger = finger.next;
        index--;
    }
    return finger.item;
}
```

`add(int index, Item item)` in singly linked lists is $O(n)$ for worst case

```
public void add(int index, Item item) { // What is the worst case?
        rangeCheck(index);

        if (index == 0) {
            add(item);
        } else {

            Node previous = null;
            Node finger = first;
            // search for index-th position
            while (index > 0) {
                previous = finger;
                finger = finger.next;
                index--;
            }
            // create new value to insert in correct position.
            Node current = new Node();
            current.next = finger;
            current.item = item;
            // make previous value point to new value.
            previous.next = current;

            n++;
        }
    }
```

remove() in singly linked lists is $O(1)$ for worst case

```
public Item remove() {
   Node temp = first;
   // Fix pointers.
   first = first.next;

   n--;

   return temp.item;
}
```

`remove(int index)` in singly linked lists is $O(n)$ for worst case

```
public Item remove(int index) {
    rangeCheck(index);

    if (index == 0) {
        return remove();
    } else {
        Node previous = null;
        Node finger = first;
        // search for value indexed, keep track of previous
        while (index > 0) {
            previous = finger;
            finger = finger.next;
            index--;
        }
        previous.next = finger.next;

        n--;
        // finger's value is old value, return it
        return finger.item;
    }

}
```

Readings:

- Textbook:
  - Chapter 1.3 (Page 142–146)
- Textbook Website:
  - Linked Lists: https://algs4.cs.princeton.edu/13stacks/

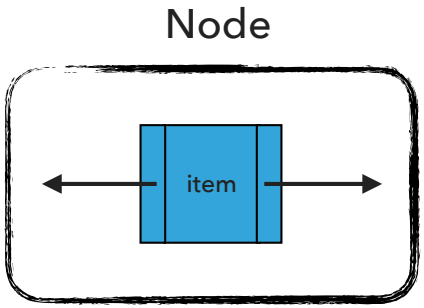
Practice Problems:

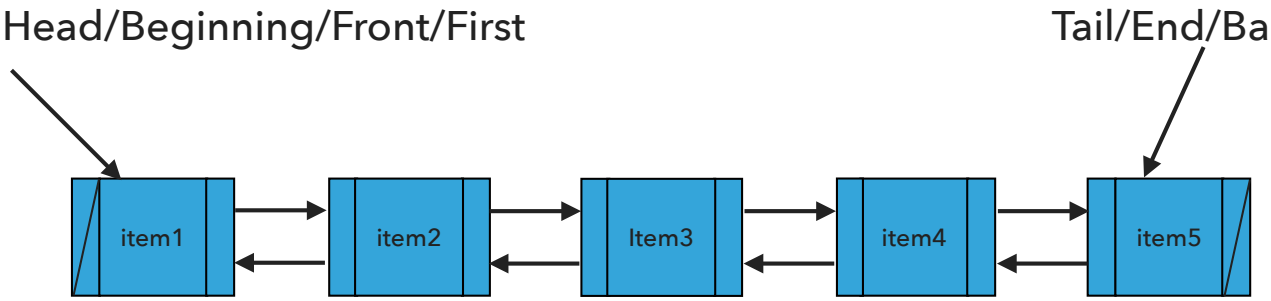
- 1.3.18–1.3.27

TEXT

Recursive Definition of Doubly Linked Lists

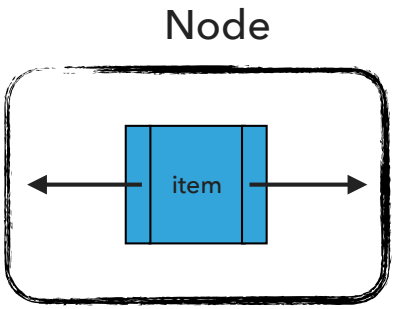
- A doubly linked list is either empty (null) or a node having a reference to a doubly linked list.
- Node: is a data type that holds any kind of data and two references to the previous and next node.





Node

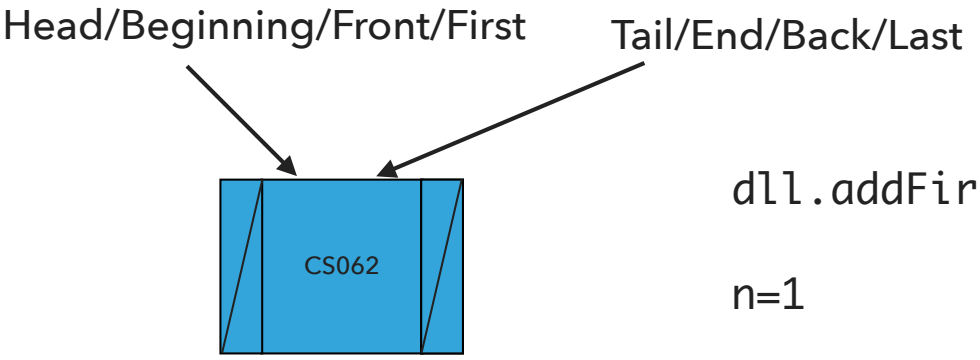
```
private class Node {
    Item item;
    Node next;
    Node prev;
  }
```



Instance variables and inner class

```
public class DoublyLinkedList<Item> implements Iterable<Item> {
   private Node first; // head of the doubly linked list
   private Node last; // tail of the doubly linked list
   private int n; // number of nodes in the doubly linked list

   /**
    * This nested class defines the nodes in the doubly linked list with a value
    * and pointers to the previous and next node they are connected.
    */
   private class Node {
      Item item;
      Node next;
      Node prev;
   }
```

addFirst() in doubly linked lists is $O(1)$ for worst case

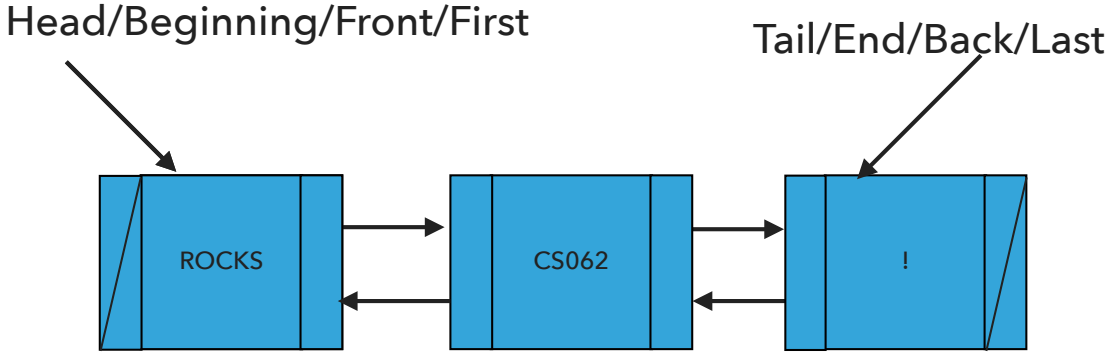


dll.addFirst("CS062")

n=1

```
public void addFirst(Item item) {
      // Save the old node
      Node oldfirst = first;

      // Make a new node and assign it to head. Fix pointers.
      first = new Node();
      first.item = item;
      first.next = oldfirst;
      first.prev = null;

      // if first node to be added, adjust tail to it.
      if (last == null)
         last = first;
      else
         oldfirst.prev = first;

      n++; // increase number of nodes in doubly linked list.
   }
```

addLast() in doubly linked lists is $O(1)$ for worst case

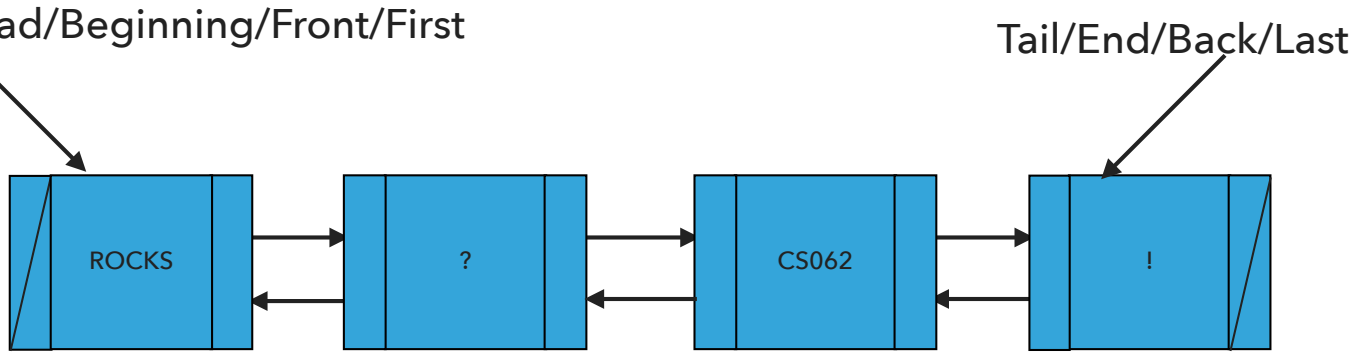



```
public void addLast(Item item) {
      // Save the old node
      Node oldlast = last;

      // Make a new node and assign it to tail. Fix pointers.
      last = new Node();
      last.item = item;
      last.next = null;
      last.prev = oldlast;

      // if first node to be added, adjust head to it.
      if (first == null)
         first = last;
      else
         oldlast.next = last;

      n++;
   }
```

dll.addLast(“!”)

n=3

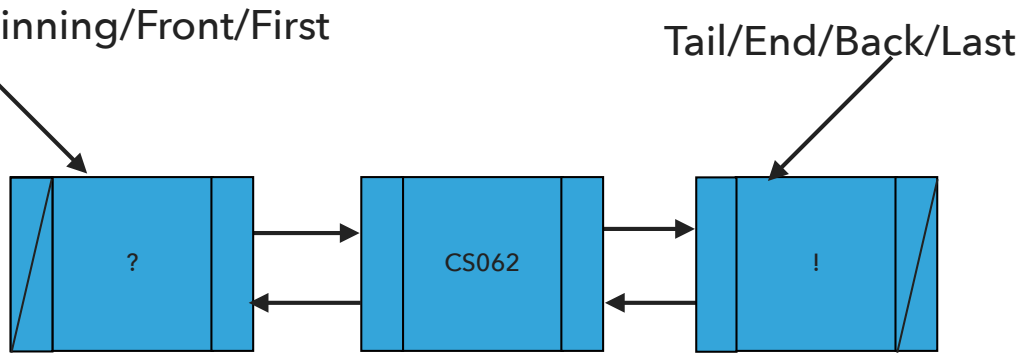
`add(int index, Item item)` in doubly linked lists is $O(n)$ for worst case



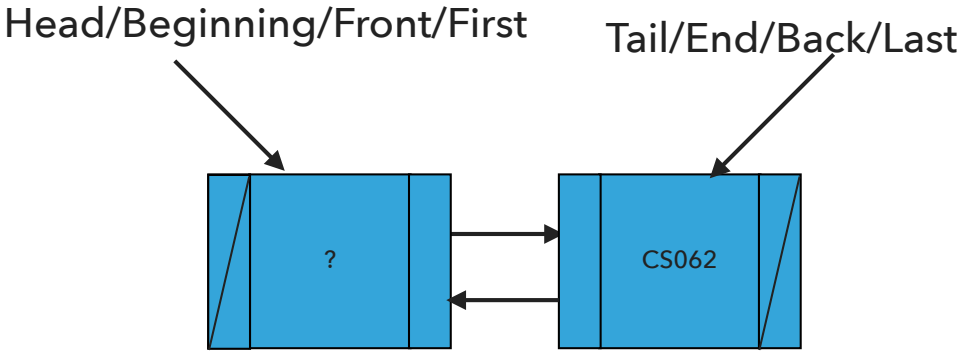

```
public void add(int index, Item item) {
        rangeCheck(index);

        if (index == 0) {
            addFirst(item);
        } else if (index == size()) {
            addLast(item);
        } else {

            Node previous = null;
            Node finger = first;
            // search for index-th position
            while (index > 0) {
                previous = finger;
                finger = finger.next;
                index--;
            }
            // create new value to insert in correct position
            Node current = new Node();
            current.item = item;
            current.next = finger;
            current.prev = previous;
            previous.next = current;
            finger.prev = current;

            n++;
        }
    }
```

`removeFirst()` in doubly linked lists is $O(1)$ for worst case

```
public Item removeFirst() {
       Node oldFirst = first;
       // Fix pointers.
       first = first.next;
       // at least 1 nodes left.
       if (first != null) {
          first.prev = null;
       } else {
          last = null; // remove final node.
       }
       oldFirst.next = null;

       n--;

       return oldFirst.item;
   }
```



removeLast() in doubly linked lists is $O(1)$ for worst case

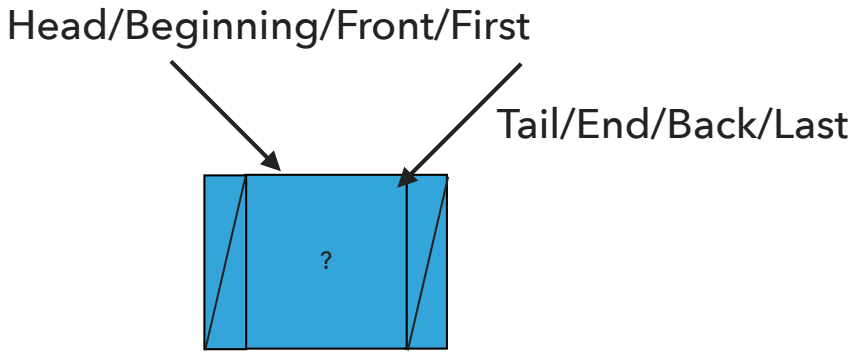



```
public Item removeLast() {

    Node temp = last;
    last = last.prev;

    // if there was only one node in the doubly linked list.
    if (last == null) {
        first = null;
    } else {
        last.next = null;
    }
    n--;
    return temp.item;
}
```

dll.removeLast()

n=2

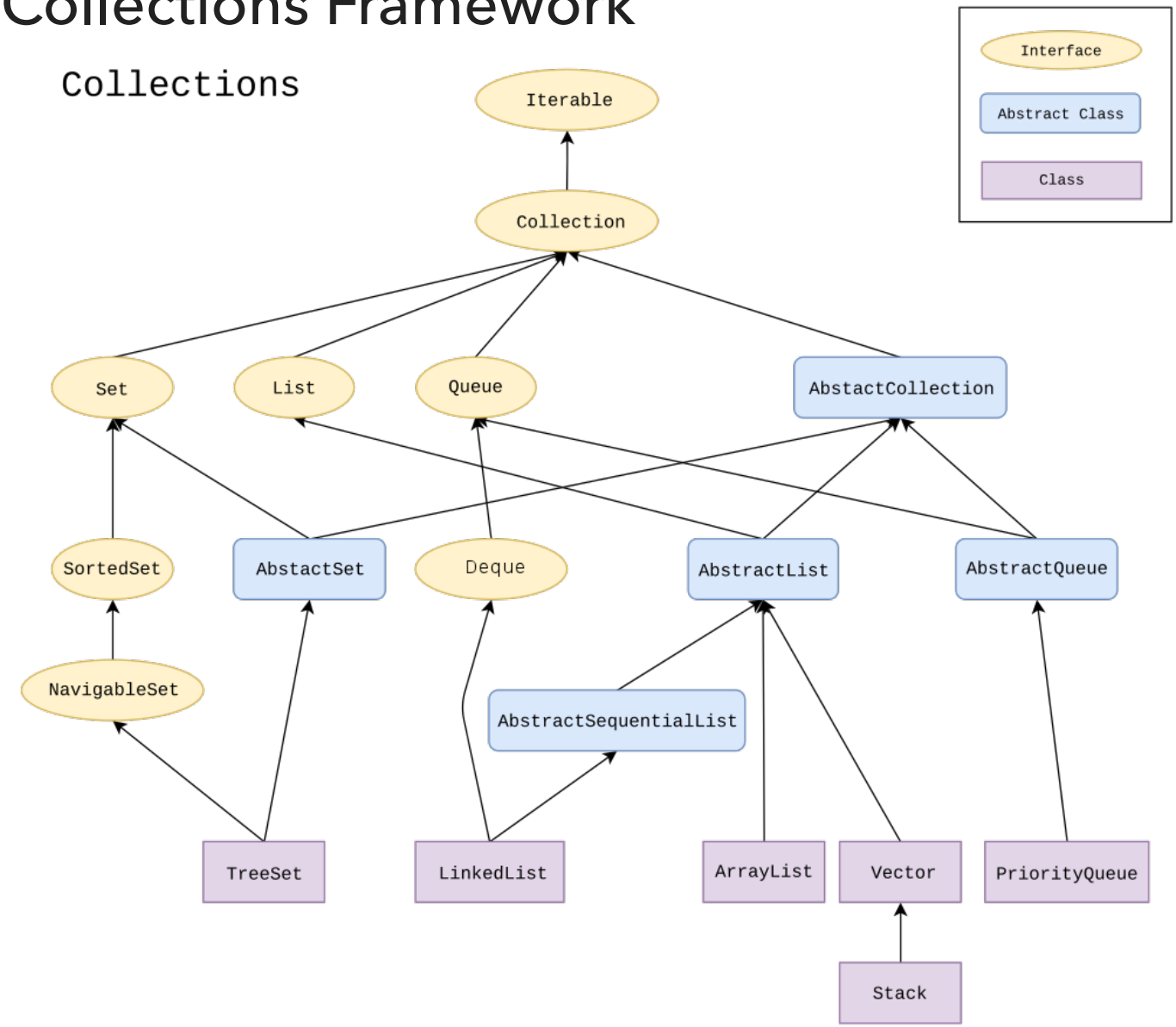
`remove(int index)` in doubly linked lists is $O(n)$ for worst case




```
public Item remove(int index) {
     rangeCheck(index);

     if (index == 0) {
          return removeFirst();
     } else if (index == size() - 1) {
          return removeLast();
     } else {
          Node previous = null;
          Node finger = first;
          // search for value indexed, keep track of previous
          while (index > 0) {
               previous = finger;
               finger = finger.next;
               index--;
          }
          previous.next = finger.next;
          finger.next.prev = previous;

          n--;
          // finger's value is old value, return it
          return finger.item;
     }

}
```

TEXT

The Java Collections Framework



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

`LinkedList` in Java Collections

- Doubly linked list implementation of the `List` and `Deque` (stay tuned) interfaces.

```
java.util.LinkedList;
```

```
public class LinkedList<E> extends
AbstractSequentialList<E> implements List<E>, Deque<E>
```

https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

Readings:

- Oracle's guides:
  - Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
  - Linked Lists: https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html
- Textbook:
  - Chapter 1.3 (Page 142–146)
- Textbook Website:
  - Linked Lists: https://algs4.cs.princeton.edu/13stacks/

Practice Problems:

- 1.3.18–1.3.27 (approach them as doubly linked lists).

TEXT

Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators

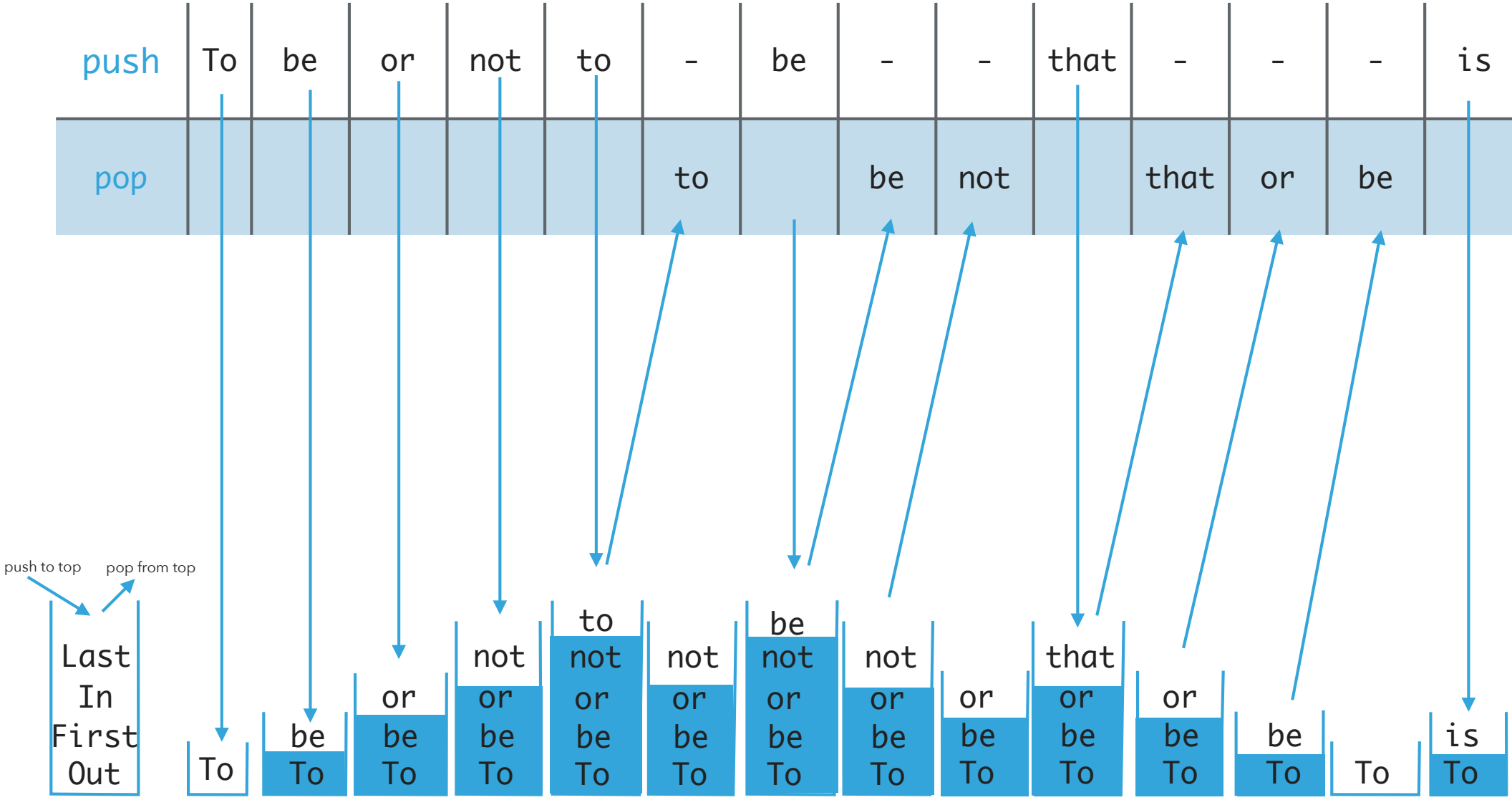
Some slides adopted from Algorithms 4th Edition and Oracle tutorials

Stacks

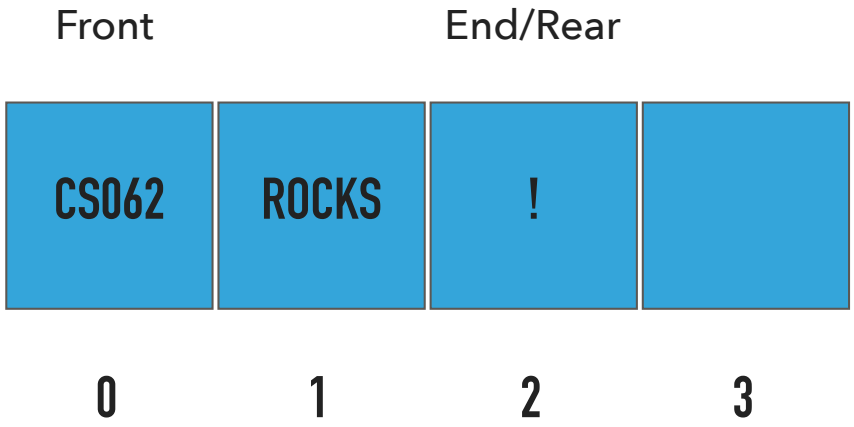
- Dynamic linear data structures.
- Items are inserted and removed following the LIFO paradigm.
- LIFO: Last In, First Out.
- Similar to lists, there is a sequential nature to the data.
- Remove the *most* recent item.

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

Example of stack operations

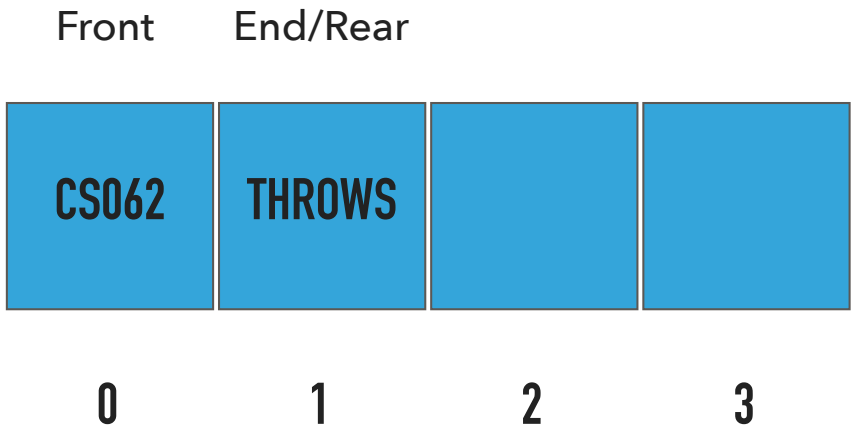


`add(Item item)`: Appends the item to the end of the ArrayList



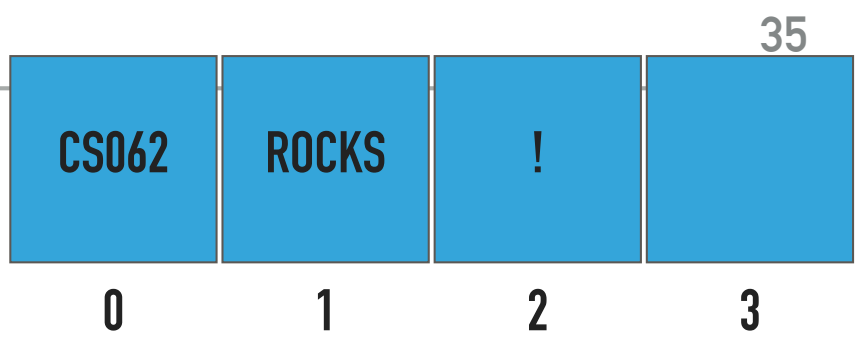

```
al.add("!");
```

`remove()`: Retrieves and removes item from the end of ArrayList




```
al.remove();
```

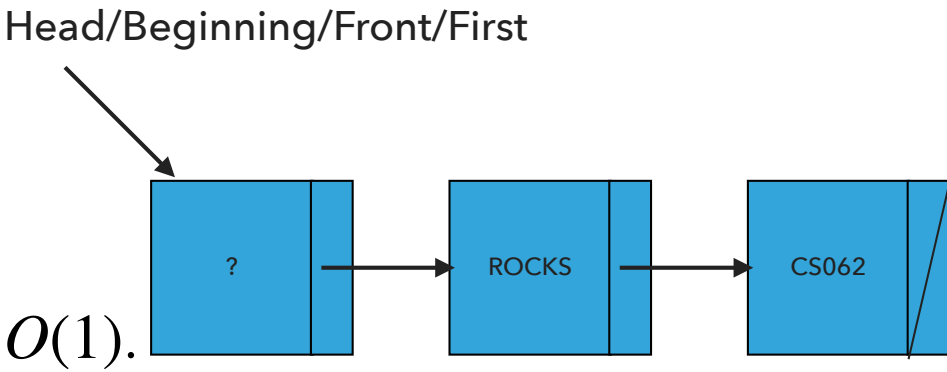
Implementing stacks with **ArrayLists**

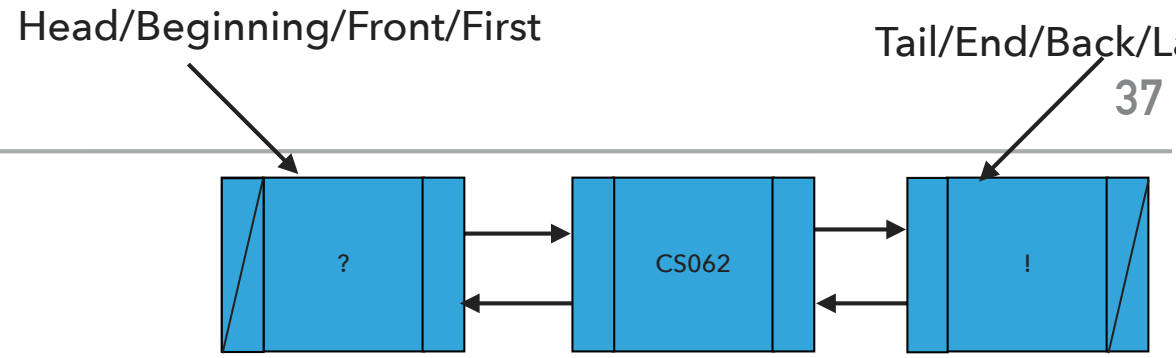


- Where should the top go to make push and pop as efficient as possible?
- The *end/rear* represents the top of the stack.
- To push an item `add(Item item)`.
  - Adds at the end. Average $O(1)$.
- To pop an item `remove()`.
  - Removes and returns the item from the end. Average $O(1)$.
- To peek `get(size()-1)`.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

Implementing stacks with **singly linked lists**

- Where should the top go to make push and pop as efficient as possible?
- The *front* 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 *end* 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 *front* 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 `head.item`.
  - Retrieves the head. $O(1)$.
- Unnecessary memory overhead with extra pointers.
- If the *end* were to re[resent the top of the stack, we'd need to use `addLast(Item item)`, `removeLast()`, and `tail.item` to have $O(1)$ complexity.

Textbook implementation of stacks

- ResizingArrayStack.java: for implementation of stacks with ArrayLists.
- LinkedStack.java: for implementation of stacks with singly linked lists.

TEXT

Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators

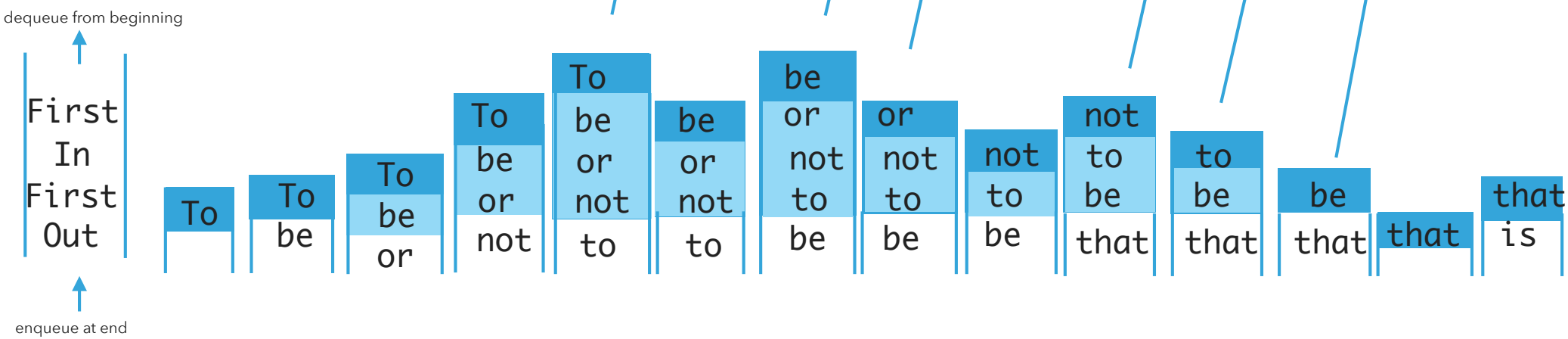
Queues

- 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.

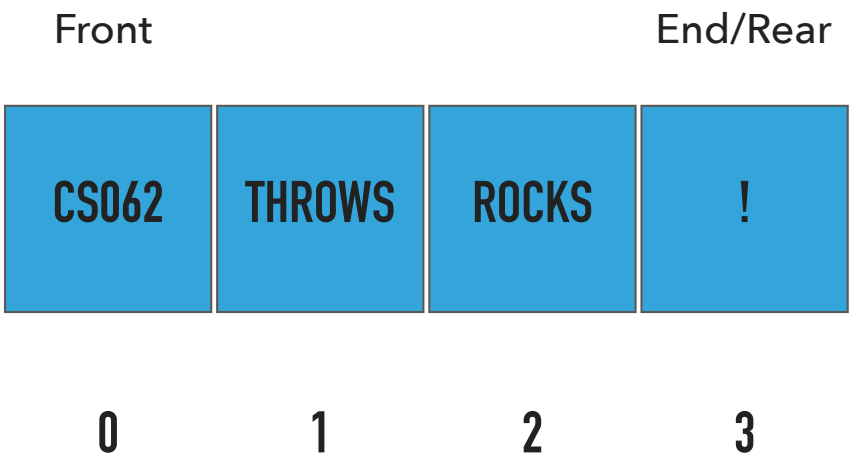
Example of queue operations

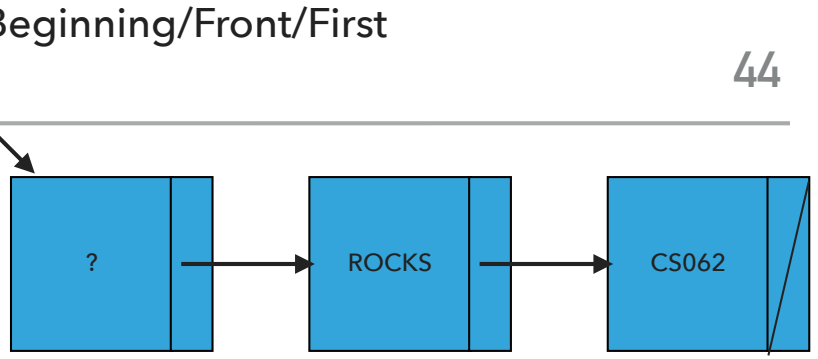
enqueue	To	be	or	not	to	-	be	-	-	that	-	-	-	is
dequeue						To		be	or		not	to	be	



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 ($O(n)$).



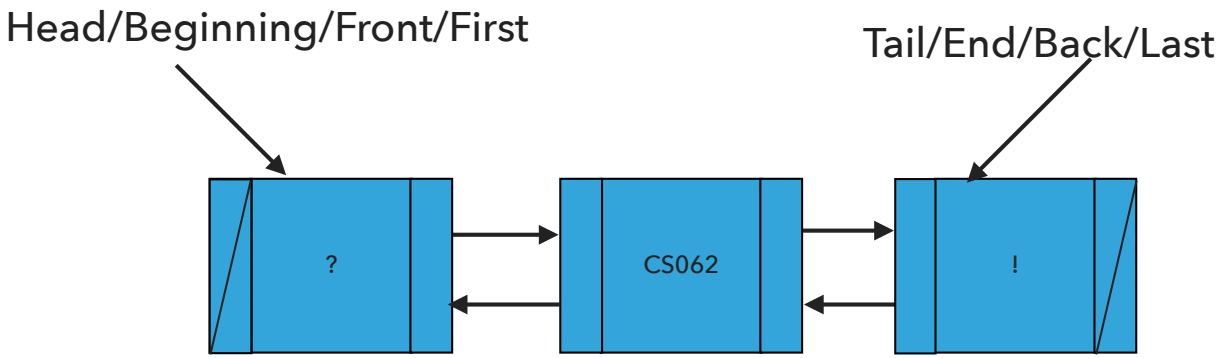


Implementing queue with singly linked list

- 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 Head?
  - 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 textbook follows.

Implementing queue with doubly linked list

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



Textbook implementation of queues

- ResizingArrayQueue.java: for implementation of queues with ArrayLists.
- LinkedQueue.java: for implementation of queues with singly linked lists.

TEXT

Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators

Stack applications

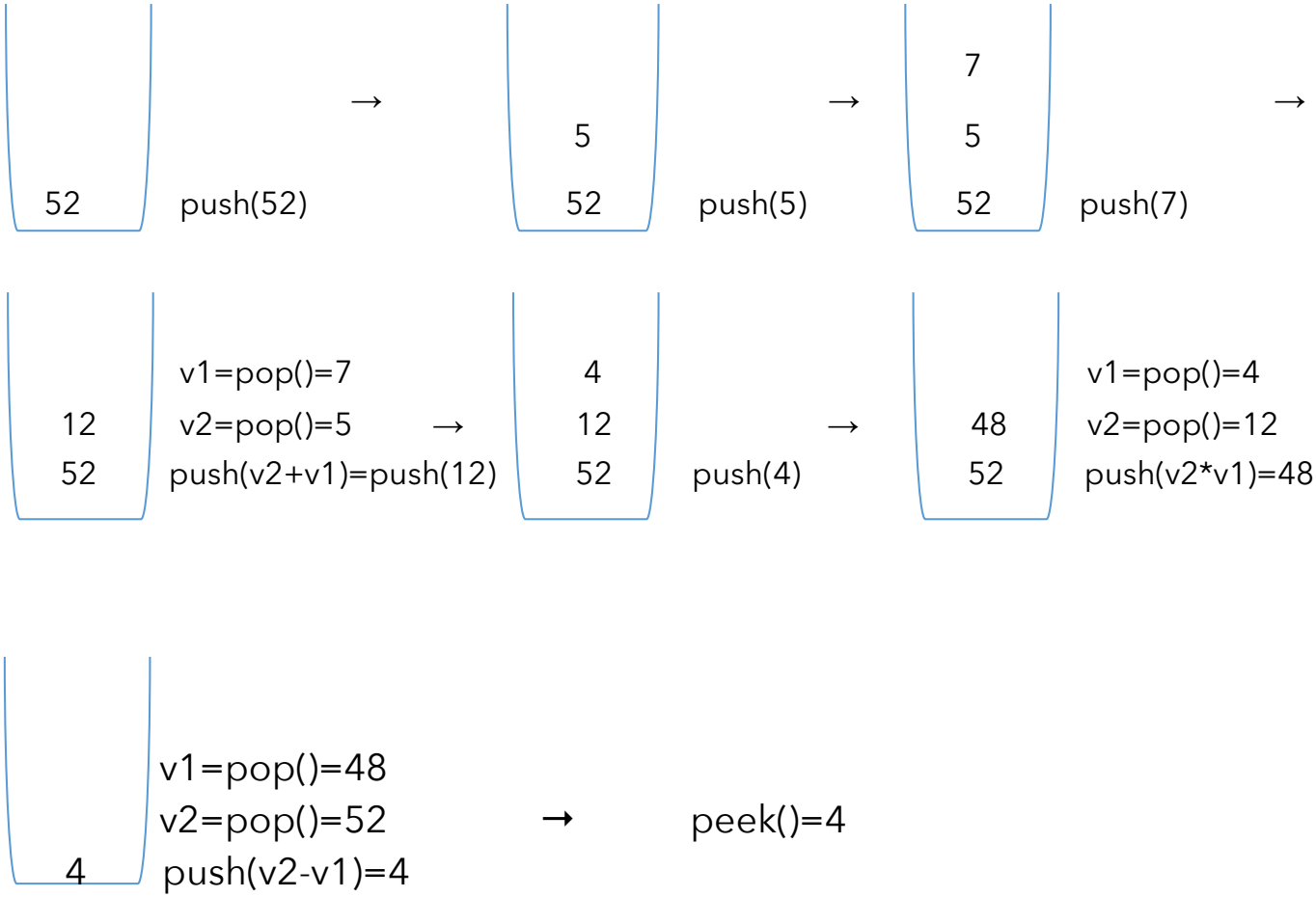
- Java Virtual Machine.
- Basic mechanisms in compilers, interpreters (see CS101).
- Back button in browser.
- Undo in word processor.
- Infix expression evaluation (Dijskstra's algorithm with two stacks).
- Postfix expression evaluation.

http://algs4.cs.princeton.edu

1.3 DIJKSTRA'S 2-STACK DEMO

Postfix expression evaluation example

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



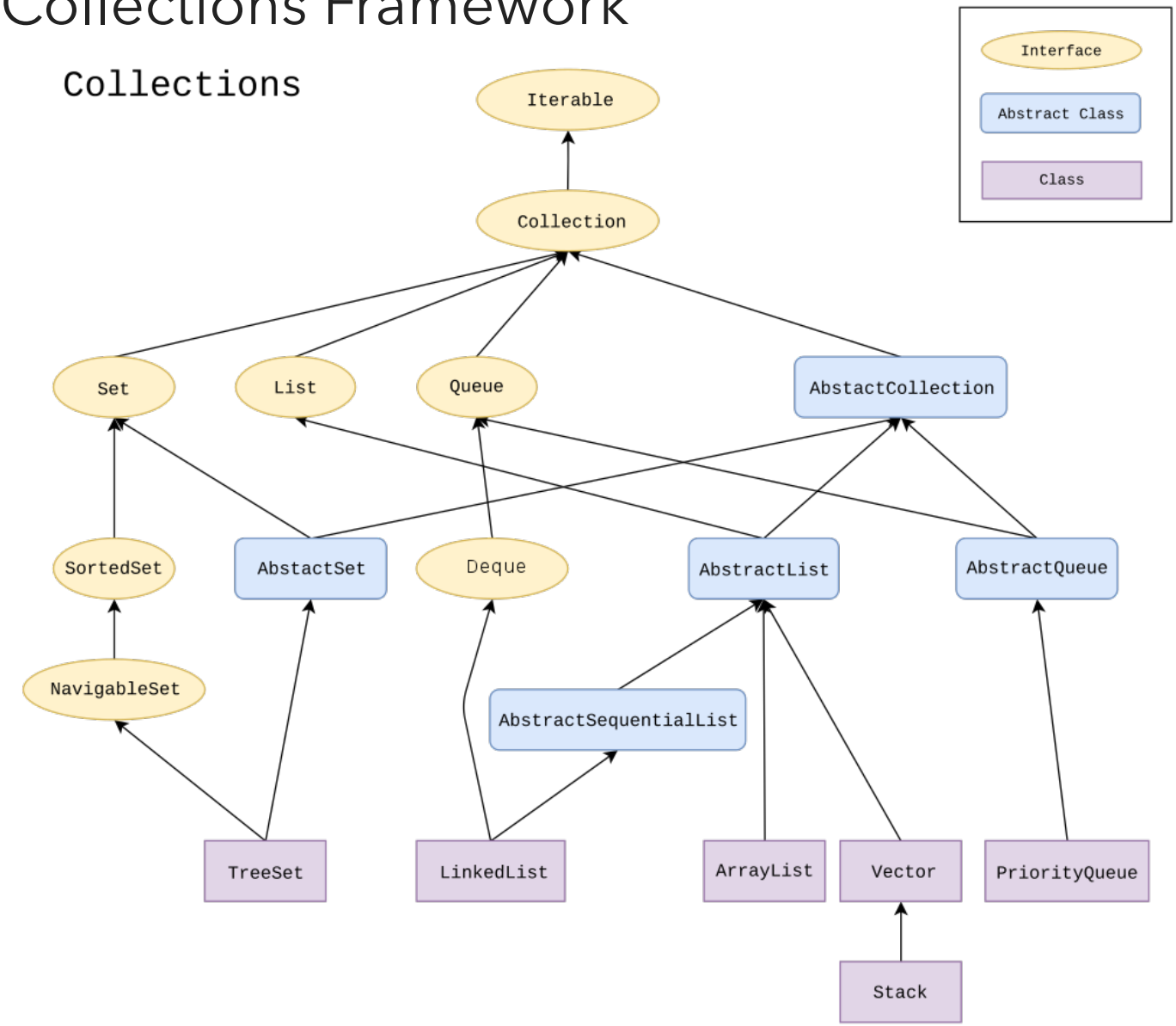
Queue applications

- 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`.
- `Queue` is an interface…
- It's recommended to use `Deque` 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<String> myList = new ArrayList<String>();
//… operations on myList

Iterator listIterator = myList.iterator();

while(listIterator.hasNext()){
  String 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(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

ITERATORS

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 < n;

          }

          public Item next() {

               return a[i++];

          }

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

          }

     }
```

Traversing `ArrayList`

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

```
for(String elt:a1) {
    System.out.println(elt);
}

a1.forEach(System.out::println);
a1.forEach(elt->{System.out.println(elt);});

a1.iterator().forEachRemaining(System.out::println);
a1.iterator().forEachRemaining(elt->{System.out.println(elt);});
```

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
  - 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
- Textbook:
  - Chapter 1.3 (Page 126–157)
- Website:
  - Stacks and Queues: https://algs4.cs.princeton.edu/13stacks/

Practice Problems:

- 1.3.2–1.3.8, 1.3.32–1.3.33