List Implementations (`ArrayList`, `Stack`, `Queue`)

MSCI 240: Algorithms & Data Structures

lecture summary

`ArrayList` implementation

`Stack` implementation

`Queue` implementation

`ArrayList` methods (10.1)

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

`ArrayListOfDouble` methods

`ArrayListOfDouble()`	constructs a new empty list of doubles
`add(`**`value`**`)`	appends value at end of list
`get(`**`index`**`)`	returns the value at given index, or if the index is invalid, throws an `IndexOutOfBoundsException`
`size()`	returns the number of elements in list

`ArrayListOfDouble` test code

```
ArrayListOfDouble testArray = new ArrayListOfDouble();
testArray.add(1.0);
testArray.add(3.14);
testArray.add(7.6);
testArray.add(-33);
testArray.add(100);

System.out.print("[");
for (int i = 0; i < testArray.size(); i++) {
    if (i > 0) {
        System.out.print(", ");
    }
    System.out.print(testArray.get(i));
}
System.out.println("]");
```

expected output:

```
[1.0, 3.14, 7.6, -33.0, 100.0]
```

```
public class ArrayListOfDouble {
    private int size;
    private double[] data;

    public ArrayListOfDouble() {
        size = 0;
        data = new double[2];
    }
    // ...


}
```

```
public class ArrayListOfDouble {
    private int size;
    private double[] data;

    // ...
    public int size() {
        return size;
    }


    // ...
}
```

```
public class ArrayListOfDouble {
    private int size;
    private double[] data;

    // ...
    public double get(int i) {
        if (i < 0 || i >= size) {
            throw new IndexOutOfBoundsException();
        }
        return data[i];
    }
}
```

```
public class ArrayListOfDouble {
    private int size;
    private double[] data;

    // ...
    public void add(double value) {
        if (size >= data.length) {
            grow(); // hide details in grow method
        }
        data[size] = value;
        size++;
    }
}
```

idea: “grow” the array whenever capacity is reached

reminder: arrays have a fixed size

need to create a new array and copy over data

```
public class ArrayListOfDouble {
    private int size;
    private double[] data;

    // ...
    private void grow() {
        double[] newData = new double[data.length * 2];
        for (int i = 0; i < data.length; i++) {
            newData[i] = data[i]; // copy data
        }
        data = newData; // data now points to array
                        // with twice the capacity
    }
}
```

`ArrayList` implementation summary

keeps track of size and stores data in an array (fields)

add method must check if array is big enough to store next element that is added

- if not, need to "grow" the list

need a (private) grow method to increase the capacity of the list whenever it needs to hold more elements

- requires a copy operation (can be expensive!)

`Stack` implementation

`Stack` methods

`Stack<`**E**`>()`	constructs a new stack with elements of type **E**
`push(`**value**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

idea: use array

push: add elements to last position

pop/peek: remove/look at element at last position

better idea: use `ArrayList`

```
public class Stack<E> {
    private ArrayList<E> data;

    public Stack() {
        data = new ArrayList<>();
    }

    public void push(E value) {
        data.add(value);
    }

    public E pop() {
        if (data.size() == 0) {
            throw new EmptyStackException();
        }
        return data.remove(data.size() - 1);
    }
}
```

`Queue` implementation

Queue class

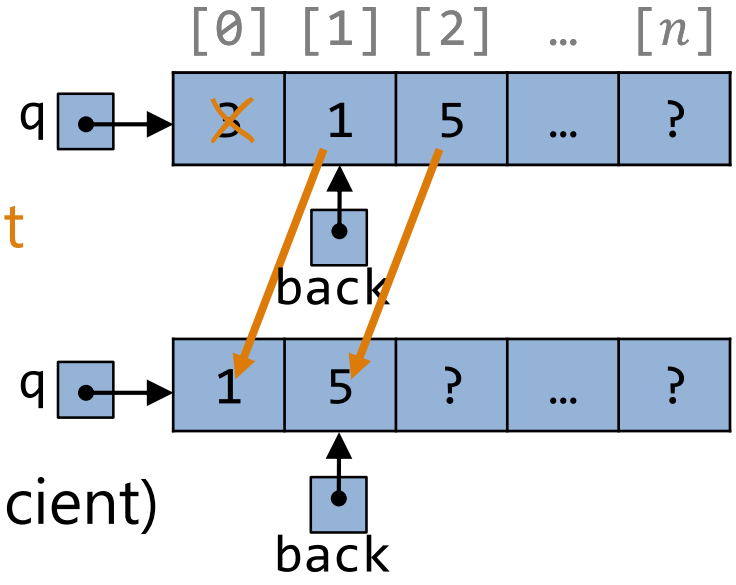
`add(`**`value`**`)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; returns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

idea (no code this time): use array

add elements to the last empty spot

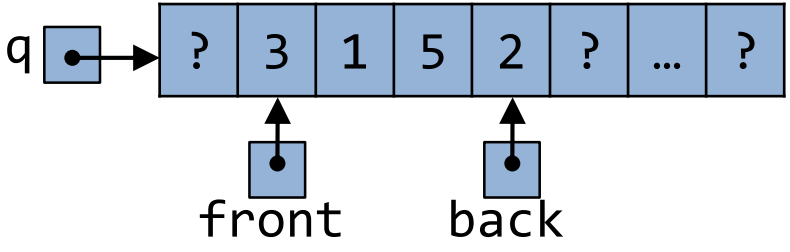
remove elements from the front

need to shift all elements left (inefficient)



alternatively: use circular array

i.e., keep track of start/end indexes and wrap around the end



`Stack` & `Queue` implementation summary

a `Stack` can be implemented using an `ArrayList`

- `push()` uses `add()`,
- `pop()` uses `remove()`
- `peek()` uses `get()`

a `Queue` can be implemented with an array/`ArrayList`, but requires a shift operation on dequeue (expensive)

a `Queue` can be implemented more efficiently with a circular array (no need to shift)

next class:

linked lists