CSE 331
Software Design & Implementation

Kevin Zatloukal

Spring 2022

ADT Implementation: Abstraction Functions

Administrivia

- HW2 due by 11pm
  - be sure use reasoning, *not trial & error*, on problem 9
    - ask questions if it is unclear what the invariant says
  - fill in three parts from the invariant, as we saw in lecture:
    - initialize the variables so the invariant is vacuous initially
    - set the loop condition so it exits when the postcondition holds
    - compare the invariants before & after progress step
      then fill in code to ensure the extra conditions required after

- HW3 released tonight

Specifying an ADT

Different types of methods:

1. **`creators`**
2. **`observers`**
3. **`producers`**
4. **`mutators`** (if mutable)

Described in terms of how they change the **abstract state**

- abstract description of what the object means
  - difficult (unless concept is already familiar) but vital
- specs have no information about concrete representation
  - leaves us free to change those in the future

IntSet, a mutable data type

```
// Overview: An IntSet is a mutable,
// unbounded set of integers.  A typical
// IntSet is { 1, 2, 7, 10 }.
class IntSet {
```

(Note: Javadoc is highly simplified...)

IntSet: mutators

```
// modifies: this
// effects:  this = this U {x}
public void add(int x)

// modifies: this
// effects:  this = this - {x}
public void remove(int x)
```

Specifications written in terms of how the **abstract state** changes

Implementing a Data Abstraction (ADT)

To implement an ADT:

- select the representation of instances
- implement operations using the chosen representation

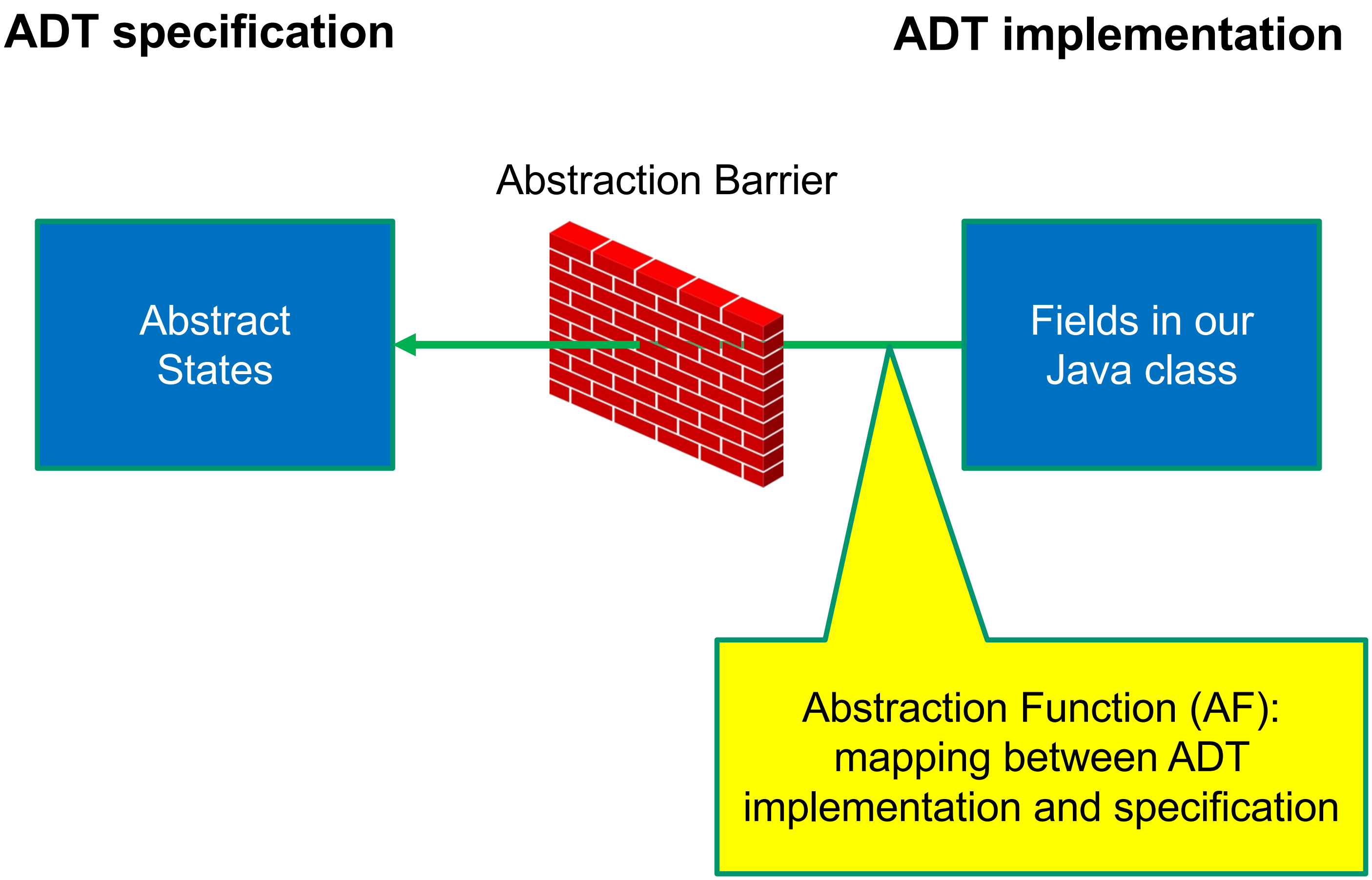
Choose a representation so that:

- it is possible to implement required operations
- the most frequently used operations are efficient / simple / …
  - abstraction allows the rep to change later
  - almost always better to start simple

Use **reasoning** to verify the operations are correct

- specs are written in terms of *abstract states* not *actual fields*
- need a new tool for this...

Data abstraction outline



ADT specification

ADT implementation

Abstraction Barrier

Abstract States

Fields in our Java class

Abstraction Function (AF): mapping between ADT implementation and specification

Connecting implementations to specs

For implementers / debuggers / maintainers of the implementation:

Abstraction Function: maps Object → abstract state

- says what the data structure *means* in vocabulary of the ADT
- maps the fields to the abstract state they represent
  - can check that the abstract value after each method meets the postcondition described in the specification

Representation Invariant: (next lecture)

Example: Circle

```java
/** Represents a mutable circle in the plane. For example,
  * it can be a circle with center (0,0) and radius 1. */
public class Circle {

  // Abstraction function:
  // AF(this) = a circle with center at this.center
  //   and radius this.rad
  private Point center;
  private double rad;

  //  ...

}
```

Example: Circle 2

```java
/** Represents a mutable circle in the plane. For example,
  * it can be a circle with center (0,0) and radius 1. */
public class Circle {

  // Abstraction function:
  // AF(this) = a circle with center at this.center
  //   and radius this.center.distanceTo(this.edge)
  private Point center, edge;

  //  ...

}
```

Example: Polynomial

```
/** An immutable polynomial with integer coefficients.
  * Examples include 0, 2x, and 3x^2 + 5x + 6. */
public class IntPoly {

  // Abstraction function:
  // AF(this) = sum of coeffs[i] * x^i
  //                for i = 0 .. coeffs.length-1
  private final int[] coeffs;

  //  ...

}
```

Example: Polynomial 2

```
/** An immutable polynomial with integer coefficients.
  * Examples include 0, 2x, and 3x^2 + 5x + 6. */
public class IntPoly {

  // Abstraction function:
  // AF(this) = sum of monomials in this.terms
  private final LinkedList<IntTerm> terms;

  //  ...

}
```

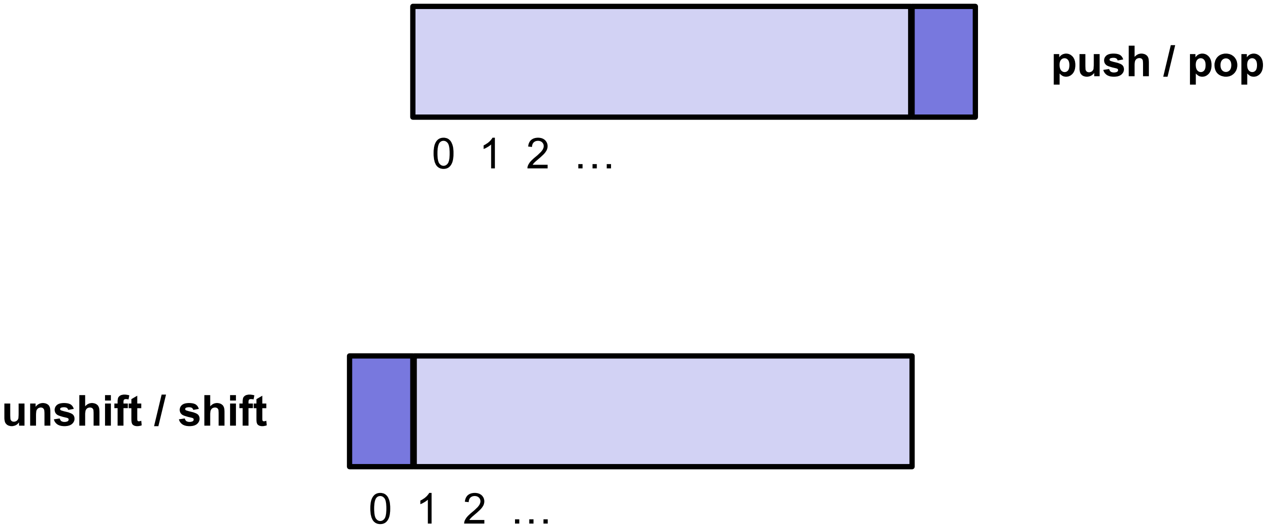
The abstraction function

- Purely conceptual (not a Java function)

- Allows us to check correctness
  - use reasoning to show that the method leaves the abstract state such that it satisfies the postcondition

Example: IntDeque

```
// List that only allows insert/remove at ends.
```



push / pop

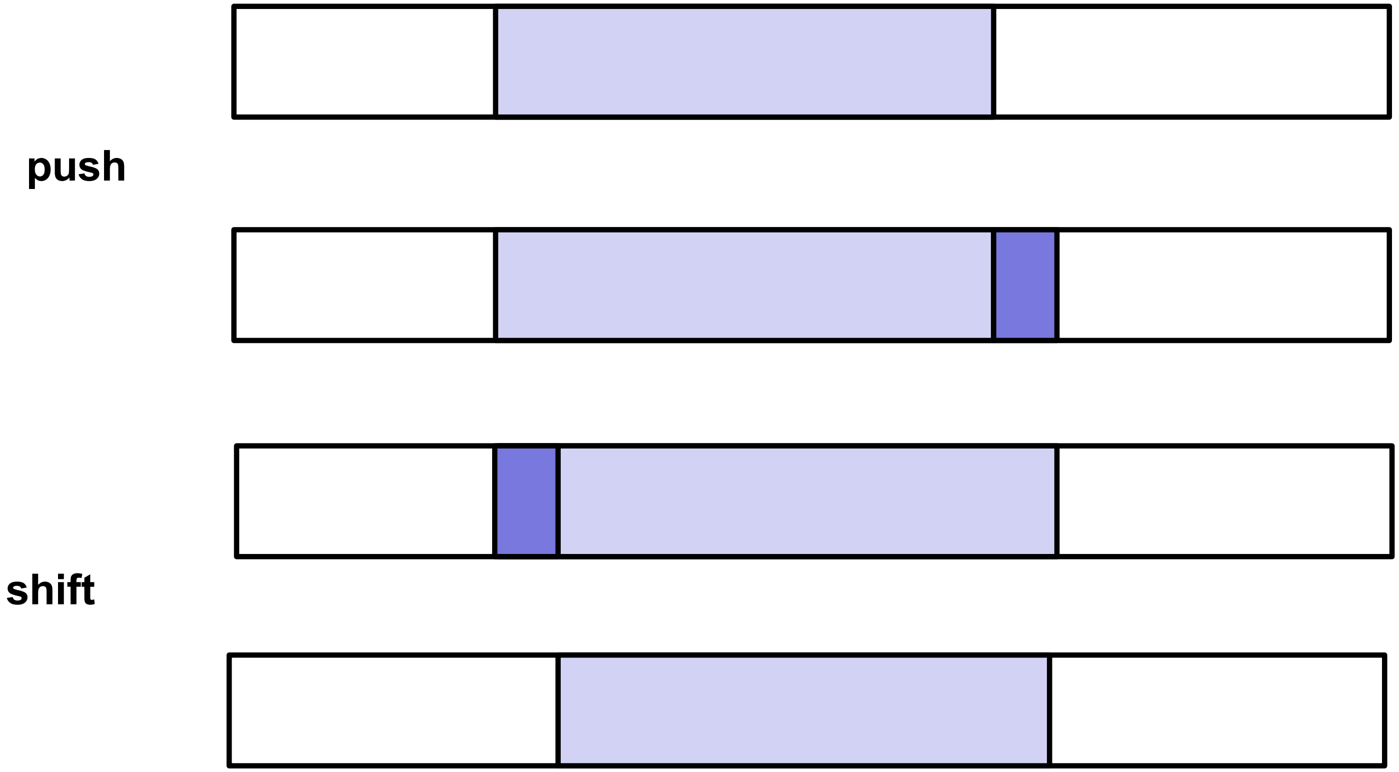
0 1 2 …

unshift / shift

0 1 2 …

Example: IntDeque

```
// List that only allows insert/remove at ends.
```

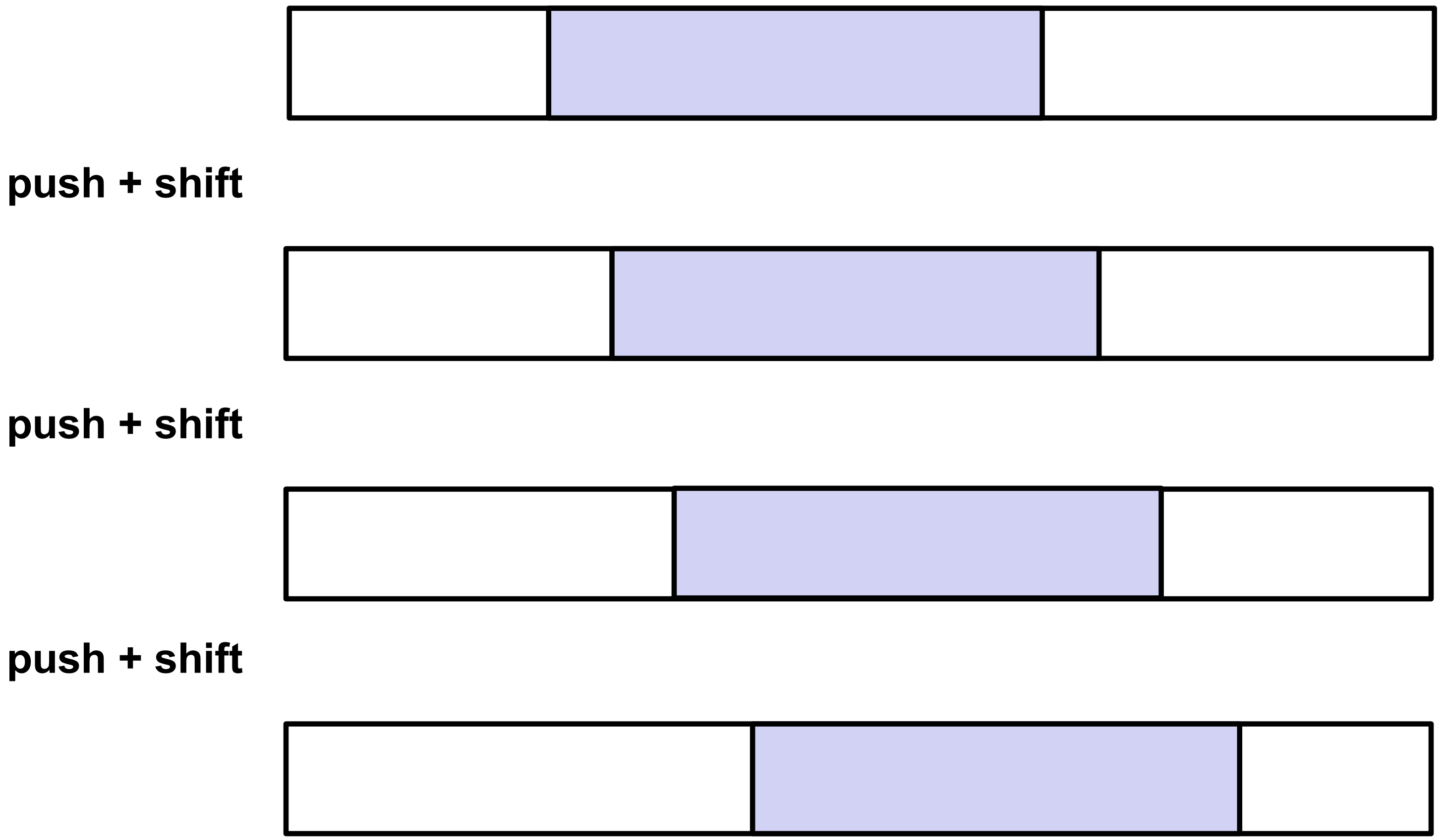


push

shift

Example: IntDeque

```
// List that only allows insert/remove at ends.
```



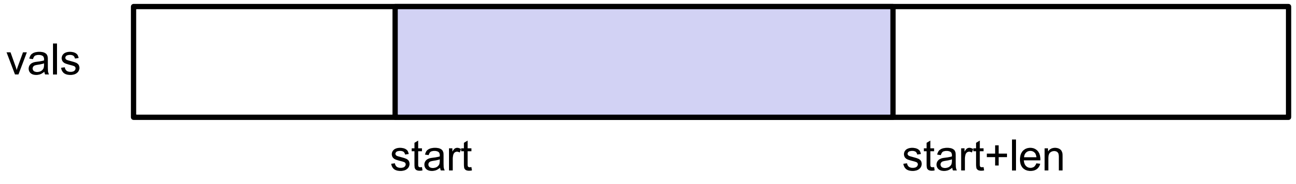
push + shift

push + shift

push + shift

Example: IntDeque

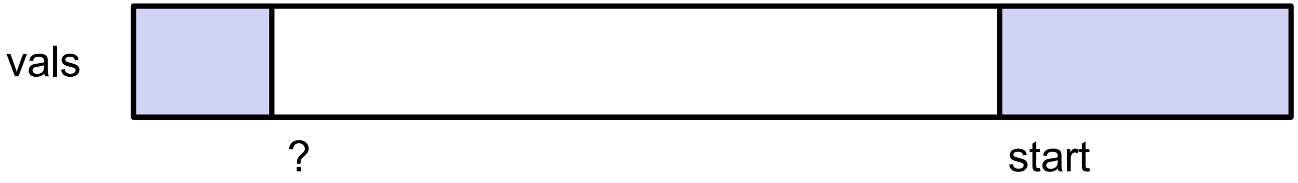
```
// List that only allows insert/remove at ends.
```



vals

start

start+len



vals

?

start

= (start+len) **mod** vals.length

= start + len – vals.length

Example: IntDeque

```
/** List that only allows insert/remove at ends. */
public class IntDeque {

  // AF(this) =
  //   vals[start..start+len-1]      if start+len <= vals.length
  //   vals[start..] + vals[0..?]    otherwise
  private int[] vals;
  private int start, len;

  // Creates an empty list.
  public IntDeque() {
    vals = new int[3];
    start = len = 0;
  }
```

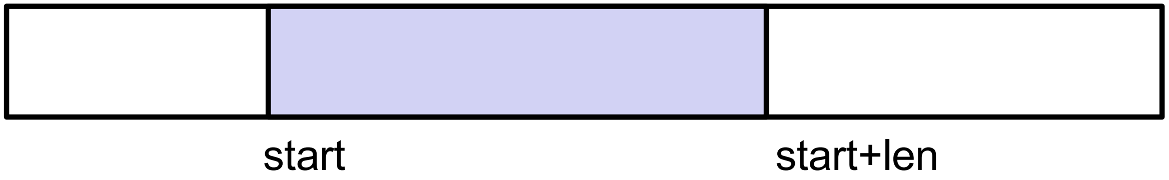
AF(this) = vals[0..-1] = []

Example: IntDeque

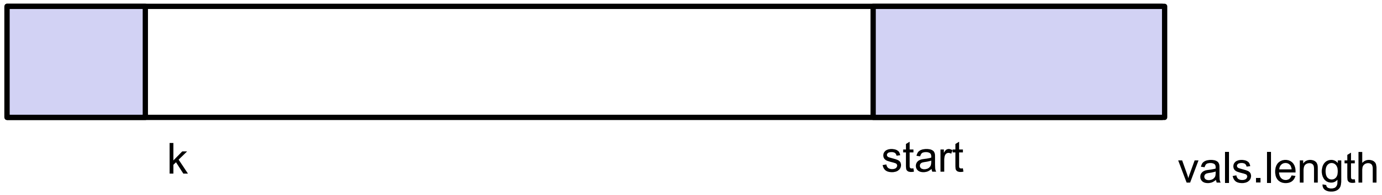
```
/** List that only allows insert/remove at ends. */
public class IntDeque {

  // AF(this) =
  //   vals[start..start+len-1]      if start+len <= vals.length
  //   vals[start..] + vals[0..?]    otherwise
  private int[] vals;
  private int start, len;

  // ...

  // @returns length of the list
  public int getLength() {
    return len;
  }
```

Example: IntDeque

```
// List that only allows insert/remove at ends.
```



#items = len



#items = vals.length – (start – k)   (= len?)

holds iff k = start + len – vals.length

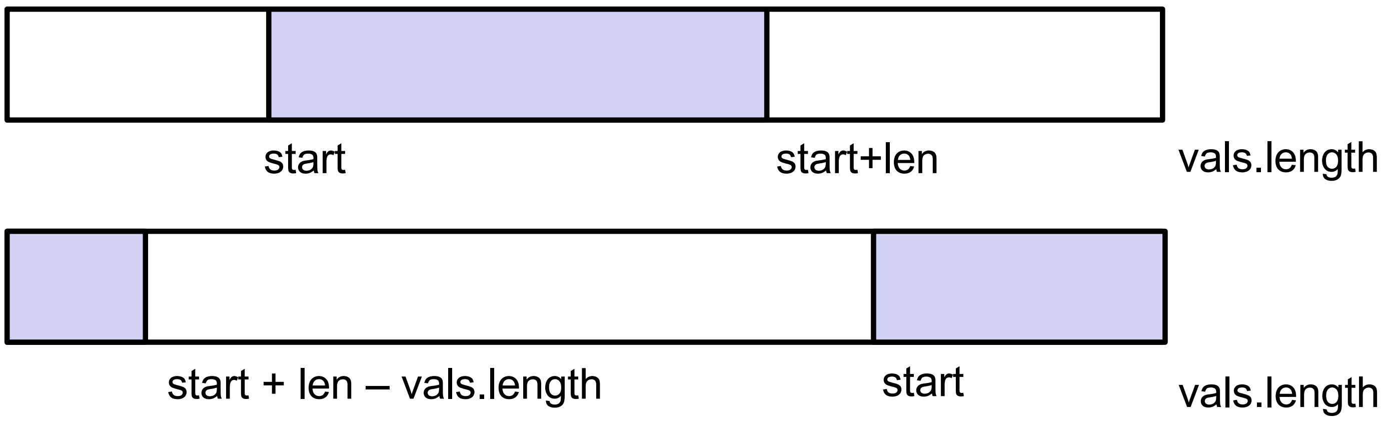
Example: IntDeque

```
/** List that only allows insert/remove at ends. */
public class IntDeque {

  // AF(this) =
  //   vals[start..start+len-1]     if start+len <= vals.length
  //   vals[start..] + vals[0..k]   otherwise
  private int[] vals;
  private int start, len;

  // ...

  // @returns length of the list
  public int getLength() {
    return len;
  }
```

1 line of code
but 2 cases for reasoning

Example: IntDeque

```
/** List that only allows insert/remove at ends. */
public class IntDeque {

  // @requires 0 <= i < length
  // @returns this[i]
  public int get(int i) { ... }
```



Example: IntDeque

```
/** List that only allows insert/remove at ends. */
public class IntDeque {

  // @requires 0 <= i < length
  // @returns this[i]
  public int get(int i) {
    if (start + len <= vals.length) {
      return vals[start + i];
    } else {
      return vals[(start + i) % vals.length];
    }
  }
```

Example: IntDeque

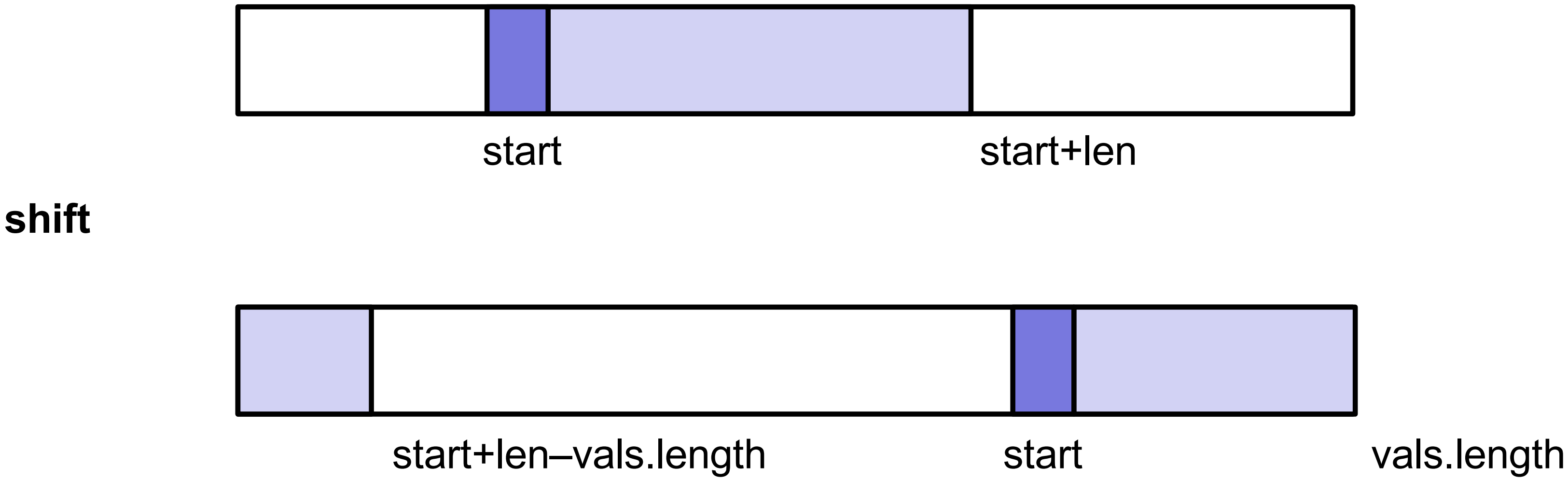
```
/** List that only allows insert/remove at ends. */
public class IntDeque {

  // @requires 0 <= i < length
  // @returns this[i]
  public int get(int i) {
    return vals[(start + i) % vals.length];
  }
```

Example: IntDeque

```
/** List that only allows insert/remove at ends. */
public class IntDeque {

  // @requires list length > 0
  // @modifies this
  // @effects first element of list is removed
  // @returns value at the front of the list
  public int shift() { ... }
```

Example: IntDeque

```
// List that only allows insert/remove at ends.
```



Example: IntDeque

```
// AF(this) =
//   vals[start..start+len-1]     if start+len <= vals.length
//   vals[start..] + vals[0..k]   otherwise

// @requires list length > 0
// @modifies this
// @effects first element of list is removed
public void shift() {
  if (start + 1 < vals.length) {
    start += 1;
  } else {
    start = 0;
  }
  len -= 1;
}
```

Example: IntDeque

```
// AF(this) =
//   vals[start..start+len-1]     if start+len <= vals.length
//   vals[start..] + vals[0..k]    otherwise

// @requires list length > 0
// @modifies this
// @effects first element of list is removed
public void shift() {
  start = (start + 1) % vals.length;
  len -= 1;
}
```

Example: IntDeque

```
// AF(this) =
//   vals[start..start+len-1]      if start+len <= vals.length
//   vals[start..] + vals[0..k]    otherwise

// @requires list length > 0
// @modifies this
// @effects first element of list is removed
// @returns value at the front of the list
public int shift() {
  int val = get(0);
  start = (start + 1) % vals.length;
  len -= 1;
  return val;
}
```

IntDeque.java

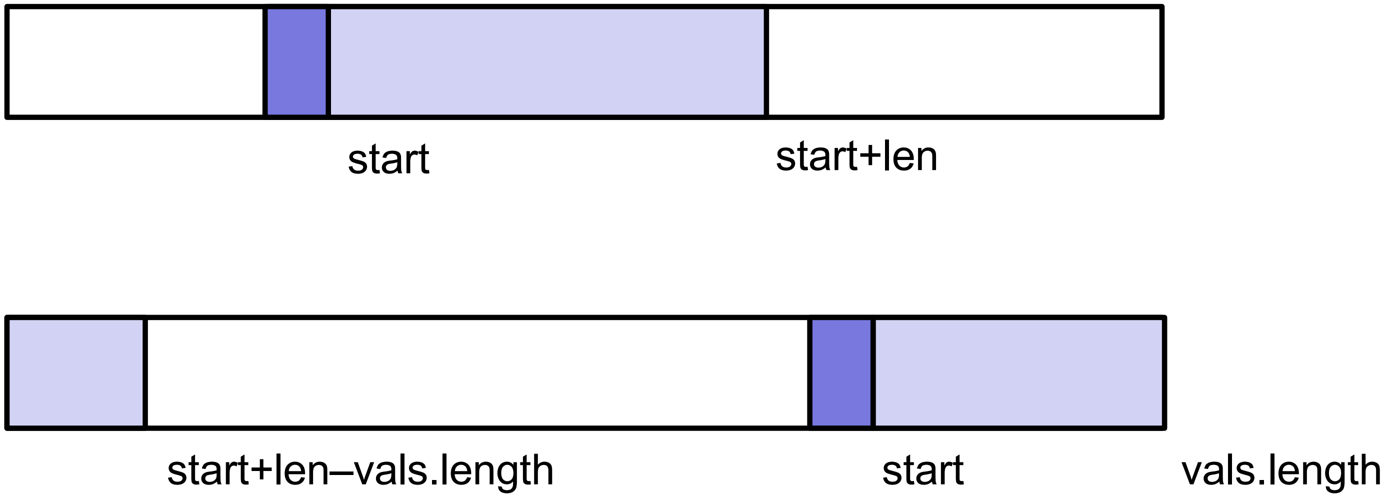
```
/** @modifies this
  * @effects this is unchanged and len < vals.length */
private void ensureMoreSpace() {
```

Example: IntDeque

```
// AF(this) =
//   vals[start..start+len-1]       if start+len <= vals.length
//   vals[start..] + vals[0..k]     otherwise

// @modifies this
// @effects insert val at the beginning of this
//          (i.e., this = [val] + this)
public int unshift(int val) { ... }
```

Example: IntDeque

```
// List that only allows insert/remove at ends.
```



unshift

Example: IntDeque

```
// AF(this) =
//   vals[start..start+len-1]       if start+len <= vals.length
//   vals[start..] + vals[0..k]     otherwise

// @modifies this
// @effects insert val at the beginning of this
//          (i.e., this = [val] + this)
public int unshift(int val) {
  ensureMoreSpace();
  start = (start > 0) ? start – 1 : vals.length – 1;
  len += 1;
  vals[start] = val;
}
```