CSCI 241

Lecture 13
Interface vs Implementation
Set, Priority Queue
Intro to Heaps

Announcements

- Quiz 3 is graded, available on Gradescope. Grades are also in Canvas.

- Dr. Hearne is putting together teams for for the ACM programming contest coming up on November 11th
  - He is hoping to form multiple teams, including some consisting of lower division students (you!)
  - Email or talk to Dr. Hearne if you are interested and/or want to learn more.

Goals

- Understand how **public** and **private** members in Java reflect the separation between **interface** and **implementation**.

- Understand the purpose and interface of the Set ADT, and how AVL trees implement it.

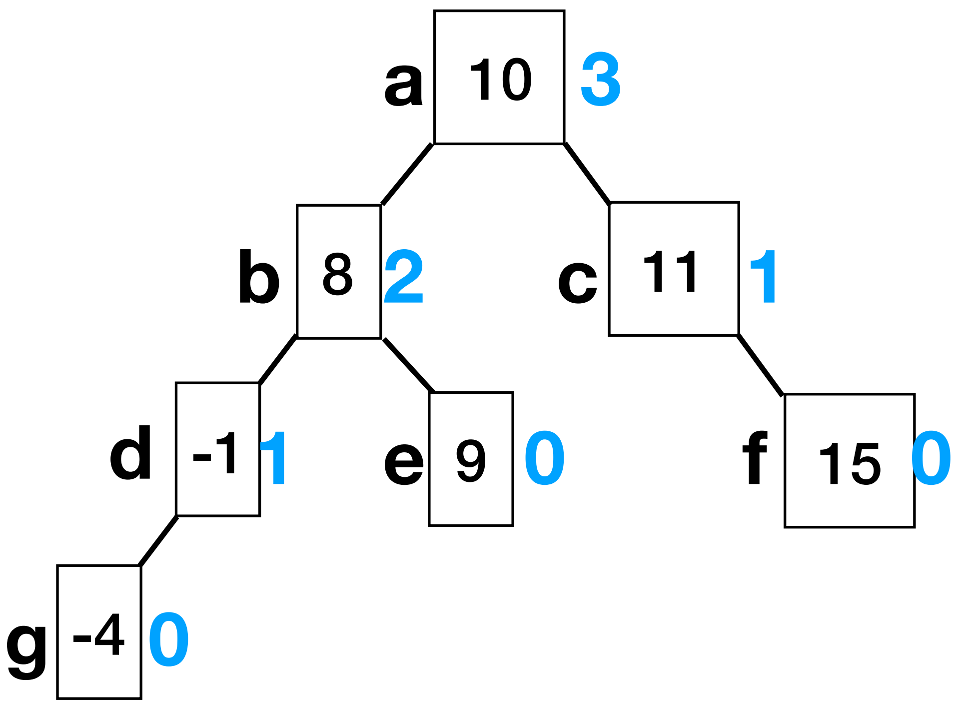
- Understand the purpose and interface of the Priority Queue ADT.

- Know the definition and properties of a heap.

A2: Updating Heights

1. **Ignore** height field until implementing rebalance.
2. Heights change when the tree structure changes: insertions and rotations.
3. Strategy - insertion: update n’s height using its childrens’ heights before calling rebalance.
4. Strategy - rotations: only x and y can change height. Update them after rotation is complete.

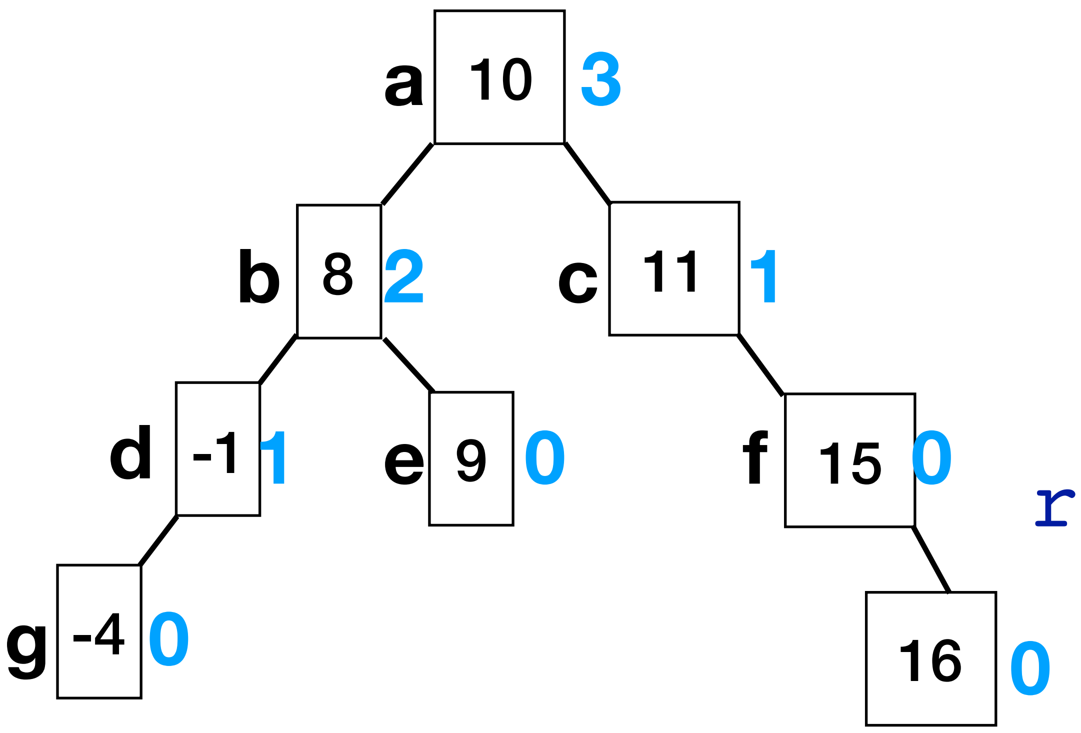
AVL Insertion




```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
  rebalance(n);
```

```
insert(a, 16)
=>insert(c, 16)
  =>insert(f, 16)
    =>attach new node
      rebalance(f)
    rebalance(c)
  rebalance(a)
```

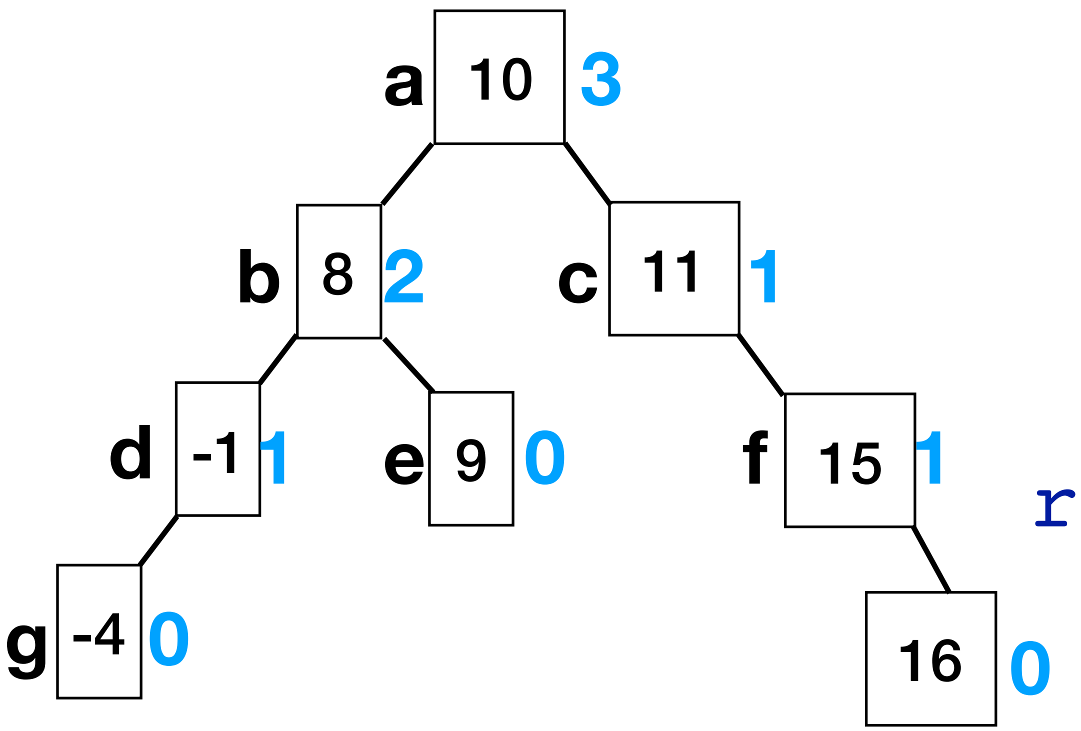
AVL Insertion




```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
  rebalance(n);
```

```
insert(a, 16)
=>insert(c, 16)
  =>insert(f, 16)
    =>attach new node
      rebalance(f)
    rebalance(c)
  rebalance(a)
```

AVL Insertion

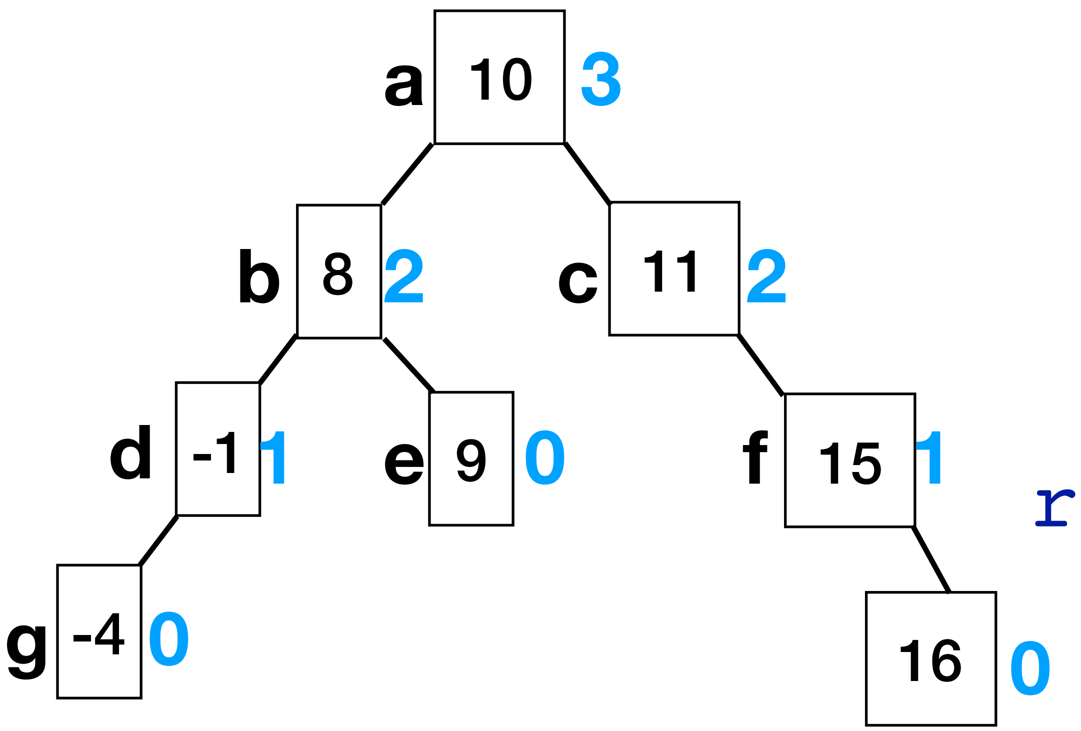



```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
  rebalance(n);
```

```
insert(a, 16)
=>insert(c, 16)
  =>insert(f, 16)
    =>attach new node
      rebalance(f)
    rebalance(c)
  rebalance(a)
```

update f's height

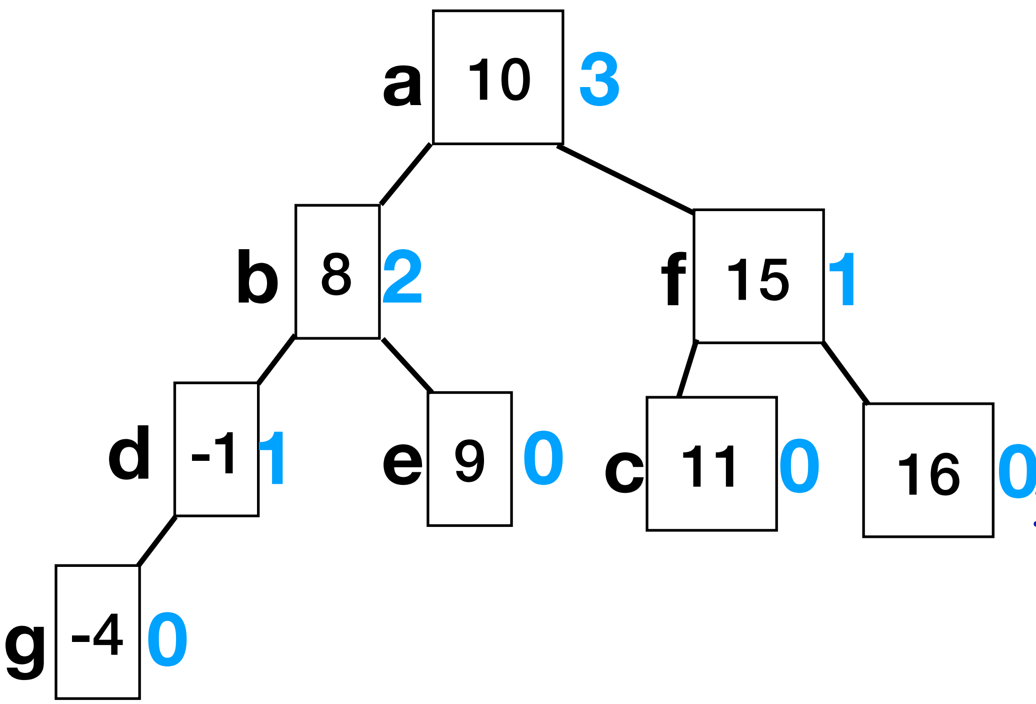
AVL Insertion




```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
  rebalance(n);
```

```
insert(a, 16)
=>insert(c, 16)
  =>insert(f, 16)
    =>attach new node
      rebalance(f)    <- update f's height
    rebalance(c)      <- update c's height
  rebalance(a)
```

AVL Insertion




```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
rebalance(n);
```

```
insert(a, 16)
=>insert(c, 16)
  =>insert(f, 16)
    =>attach new node
      rebalance(f)
    rebalance(c)
  rebalance(a)
```

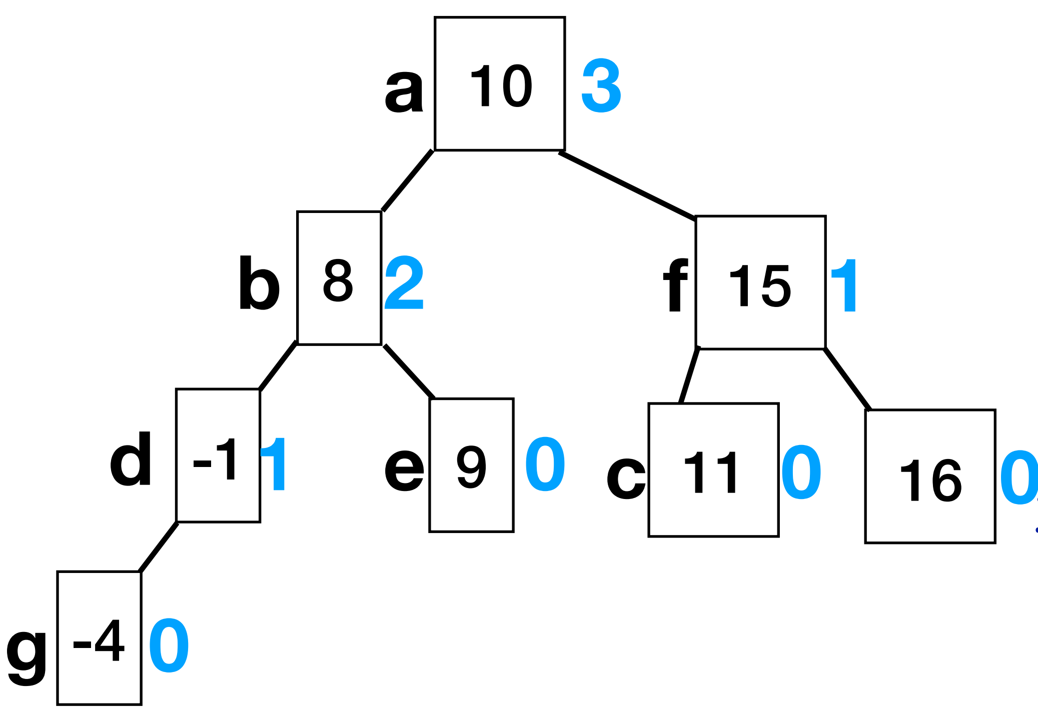
update f's height

update c's height

(rotation happens)

AVL Insertion

```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
rebalance(n);
```




```
insert(a, 16)
=>insert(c, 16)
  =>insert(f, 16)
    =>attach new node
      rebalance(f)
    rebalance(c)
  rebalance(a)
```

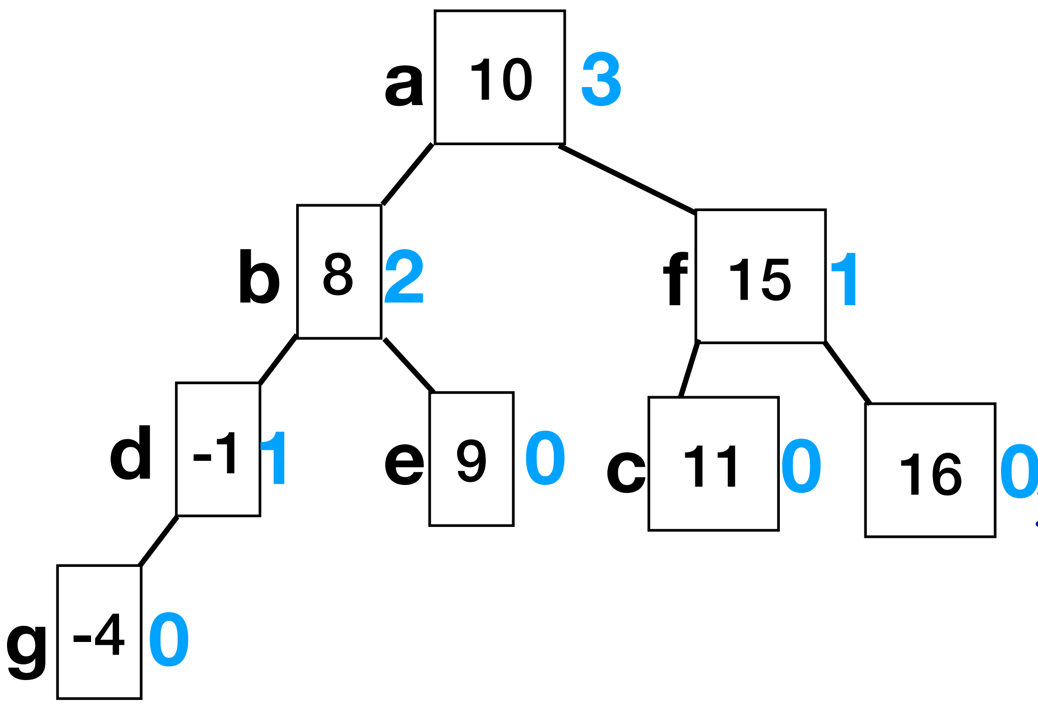
update f's height

update c's height

update a's height

AVL Insertion

```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
rebalance(n);
```



Why shouldn't we use the recursive height() method from lab3 to update heights?

```
insert(a, 16)
=>insert(c, 16)
  =>insert(f, 16)
    =>attach new node
      rebalance(f)     <- update f's height
    rebalance(c)       <- update c's height
  rebalance(a)         <- update a's height
```

A2: Updating Heights

1. **Ignore** height field until implementing rebalance.
2. Heights change when the tree structure changes: insertions and rotations.
3. Strategy - insertion: update n's height **using its childrens' heights** before calling rebalance.
4. Strategy - rotations: only x and y can change height. Update them after rotation is complete.

Recap: **Interface** vs **Implementation**

- Abstract data structures are **interfaces**
  - they specify only the **interface** to a class (method names and specifications)
  - not its **implementation** (method bodies, fields, …)

- Abstract data structures can have multiple possible **implementations.**

Interface vs **Implementation**: in Java

- A **class** is a “blueprint” for an **object**.
- It contains **members** including
  - **fields** (variables)
  - **methods** (functions)
- Fields and methods can be **public** or **private**

- **Private** members **can’t** be seen outside the class.
- **Public** members **can** be seen outside the class.

- Public members provide the class’s **interface.**
- Private members are used to **implement** the interface.

Interface vs **Implementation**: in Java

- Public members provide the class's **interface.**
- Private members are used to **implement** the interface.

Important consequence:

- If a public method has a specification, it has to implement that specification **precisely** and **completely**.
- Just because **you** never encounter an edge case doesn't mean **someone else** using your class won't.

Interface vs **Implementation**: in Java

- Public members provide the class's **interface.**
- Private members are used to **implement** the interface.

Important consequence:

- If a public method has a specification, it must implement that specification **precisely** and **completely**.
- Just because **you** never encounter an edge case doesn't mean **someone else** using your class won't.

(e.g., me grading your code using unit tests)

Interface vs **Implementation**: A pertinent example.

```
/** partition A around the pivot A[pivIndex].
 *  return the pivot's new index.
 *  precondition: start <= pivIndex < end
 *  postcondition: A[start..i] <= A[i] <= A[i+1..end]
 *     where i is the return value */
public int partition(int[] A, int start, int end, int pivIndex) {
```

Even if **your** `quickSort` always calls this with

```
pivIndex = end-1
```

`partition` is **public** and **must** implement the spec exactly!
It needs to work if pivIndex is **any** value where
`start <= pivIndex < end` (the precondition says so!)

Interface vs **Implementation**: Javadoc Comments

There's a **big** difference between

Appears in documentation that tells people how to use your class!

```
/** this comment */
public void myMethod() {
```

and

Does not appear elsewhere - "merely" helpful to someone reading your code.

```
/* this comment */
private void myMethod() {
```

Scanner.nextInt() documentation   Scanner.nextInt() source code

Interface vs **Implementation**: Tips for Assignments (and life)

- Public method specifications, names, return values, and parameters should **never** be changed.
- All public methods must implement its specification **completely**, even if your use case doesn't require it.
- You can do basically **whatever you want** with private methods, as long as you stick to good coding style.
    - It's still a good idea to write a specification and test private methods.
    - Use them to make your code easier to read (and write).
    - In my A1 solution, I wrote a helper to find the pivot:

```
/* put the median of A[start], A[middle], A[end-1] at the start *
private void medOfThree(int[] A, int start, int end) {
```

Interface vs Implementation

What the operations do

An abstract data type specifies only **interface**, not **implementation**

How they are accomplished

Interface vs Implementation

What the operations do

An abstract data type specifies only **interface**, not **implementation**

How they are accomplished

Abstract data types can have multiple possible **implementations.**

Abstract Data Types

- **interface** List defines an "abstract data type"
- It has public methods: add, get, remove, …
- Various classes **implement** List:

Class:	ArrayList	LinkedList
Backing storage:	array	chained nodes
add(i, val)	O(n)	O(n)
add(0, val)	O(n)	O(1)
add(n, val)	O(1)	O(1)
get(i)	O(1)	O(n)
get(0)	O(1)	O(1)
get(n)	O(1)	O(1)

Stacks and Queues are **restricted** Lists

Stack	Class:	ArrayList	LinkedList
	Backing storage:	array	chained Node objects
	add(i, val)	O(n)	O(n)
push(val)	add(0, val)	O(n)	O(1)
	add(n, val)	O(1)	O(1)
	get(i)	O(1)	O(n)
peek()	get(0)	O(1)	O(1)
	get(n)	O(1)	O(1)
pop()	remove(0)	O(n)	O(1)
	remove(i)	O(n)	O(n)
	remove(n)	O(1)	O(1)

Stacks and Queues are **restricted** Lists

Queue	Class:	ArrayList	LinkedList
	Backing storage:	array	chained Node objects
	add(i, val)	O(n)	O(n)
	add(0, val)	O(n)	O(1)
enQ(val)	add(n, val)	O(1)	O(1)
	get(i)	O(1)	O(n)
peek()	get(0)	O(1)	O(1)
	get(n)	O(1)	O(1)
deQ()	remove(0)	O(n)	O(1)
	remove(i)	O(n)	O(n)
	remove(n)	O(1)	O(1)

The **Set** ADT

- A **Set** maintains a collection of **unique** things.
- Java has this ADT built in as an interface:

  `java.util.Set`
- Some methods from **`java.util.Set:`**
  - **`boolean add(Object ob)`**
  - **`boolean contains(Object ob)`**
  - **`boolean remove(Object ob)`**

The **Set** ADT

Methods from **`java.util.Set:`**

- **`boolean add(Object ob)`**
- **`boolean contains(Object ob)`**
- **`boolean remove(Object ob)`**

Possible implementations:

- array
- linked list
- BST
- AVL Tree

The **Set** ADT

Methods from `java.util.Set:`

- `boolean add(Object ob)`
- `boolean contains(Object ob)`
- `boolean remove(Object ob)`

Possible implementations:

- **array**
- linked list
- BST
- AVL Tree

Array operations:

- indexing/assignment (A[i] = v)
- length (A.length)

How can we implement Set's **add** method?

The **Set** ADT

A **Set** maintains a collection of **unique** things.

Methods from `java.util.Set:`

- `boolean add(Object ob)`
- `boolean contains(Object ob)`
- `boolean remove(Object ob)`

In small groups:

1. Write **English** descriptions of how to implement `contains` and `remove` using an **array**.

2. Do the same, but using the operations provided by an **AVL Tree**.

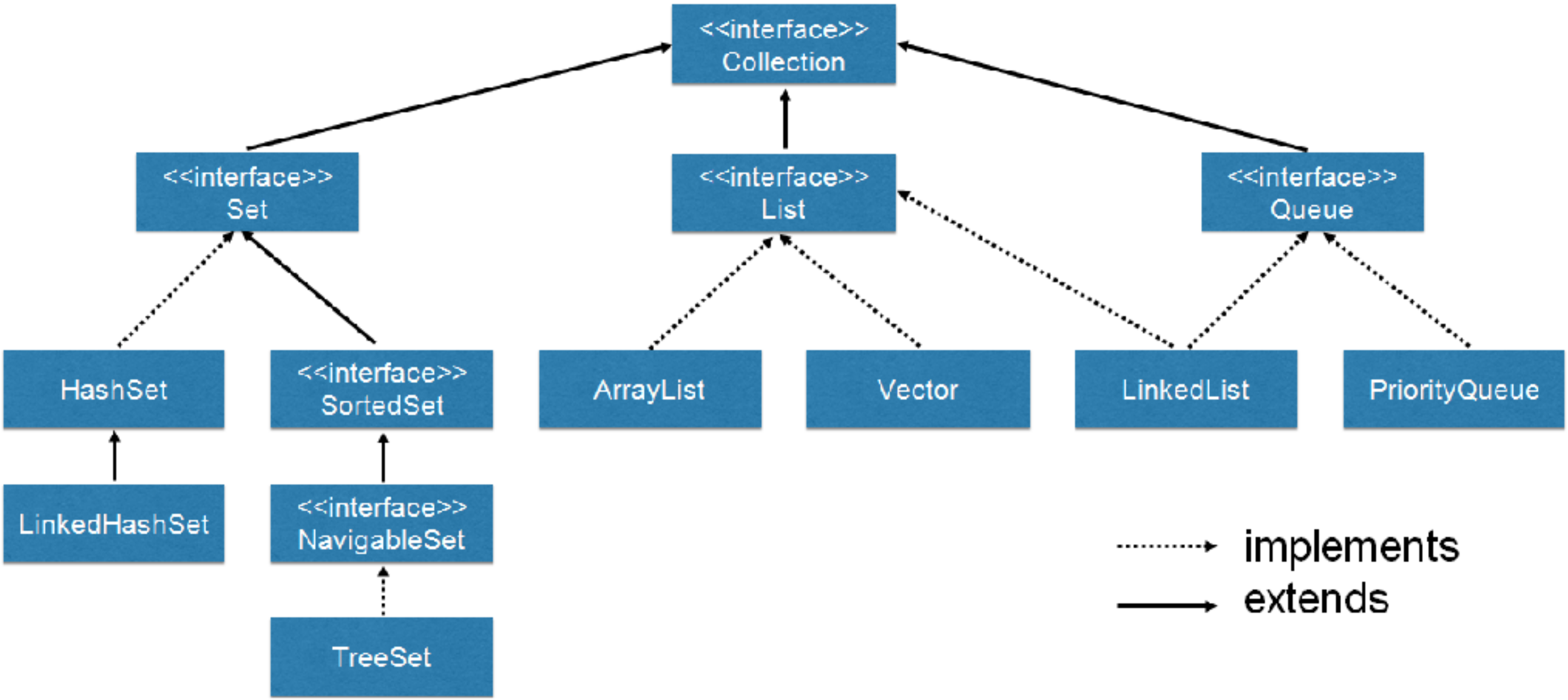
Array operations:

- indexing/assignment (A[i] = v)
- length (A.length)

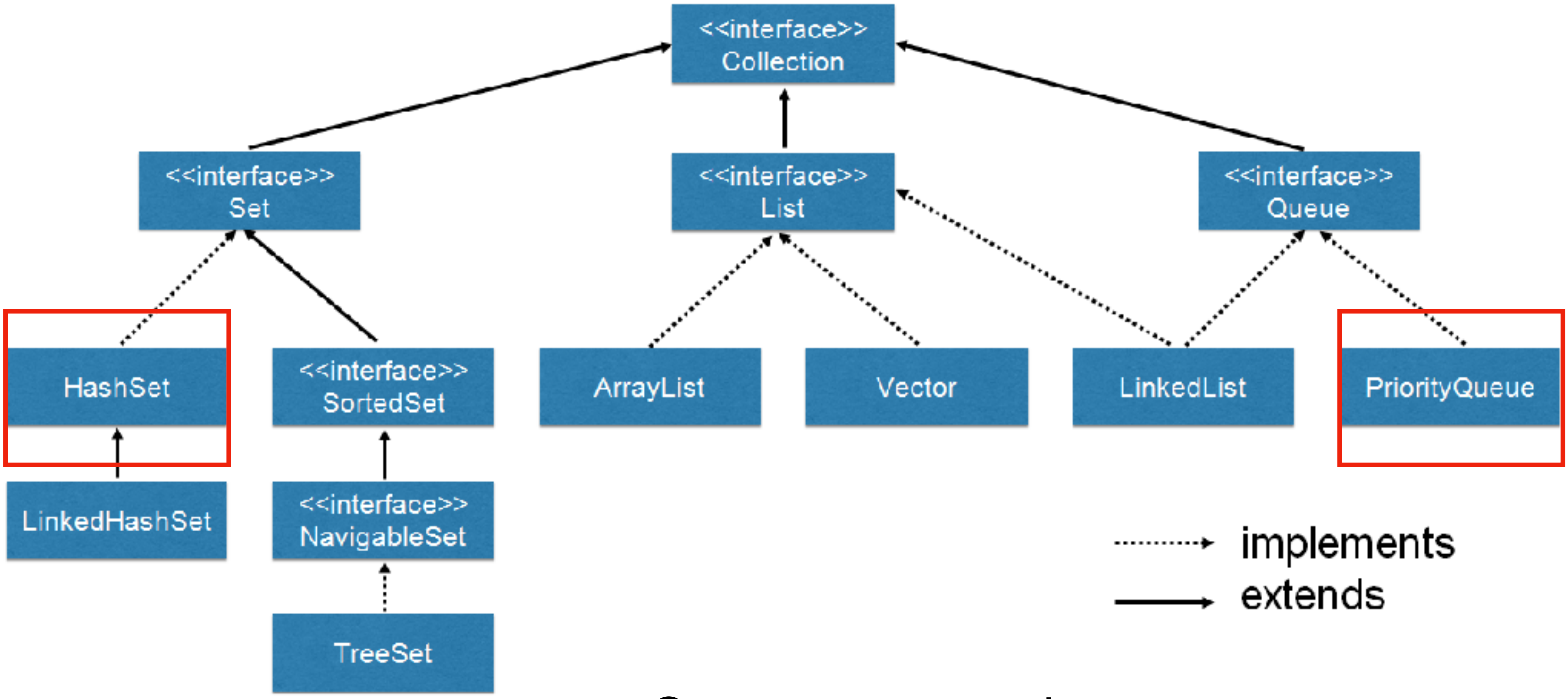
AVL operations:

- search(Object ob)
- insert(Object ob)
- remove(Object ob)

Collection Interface



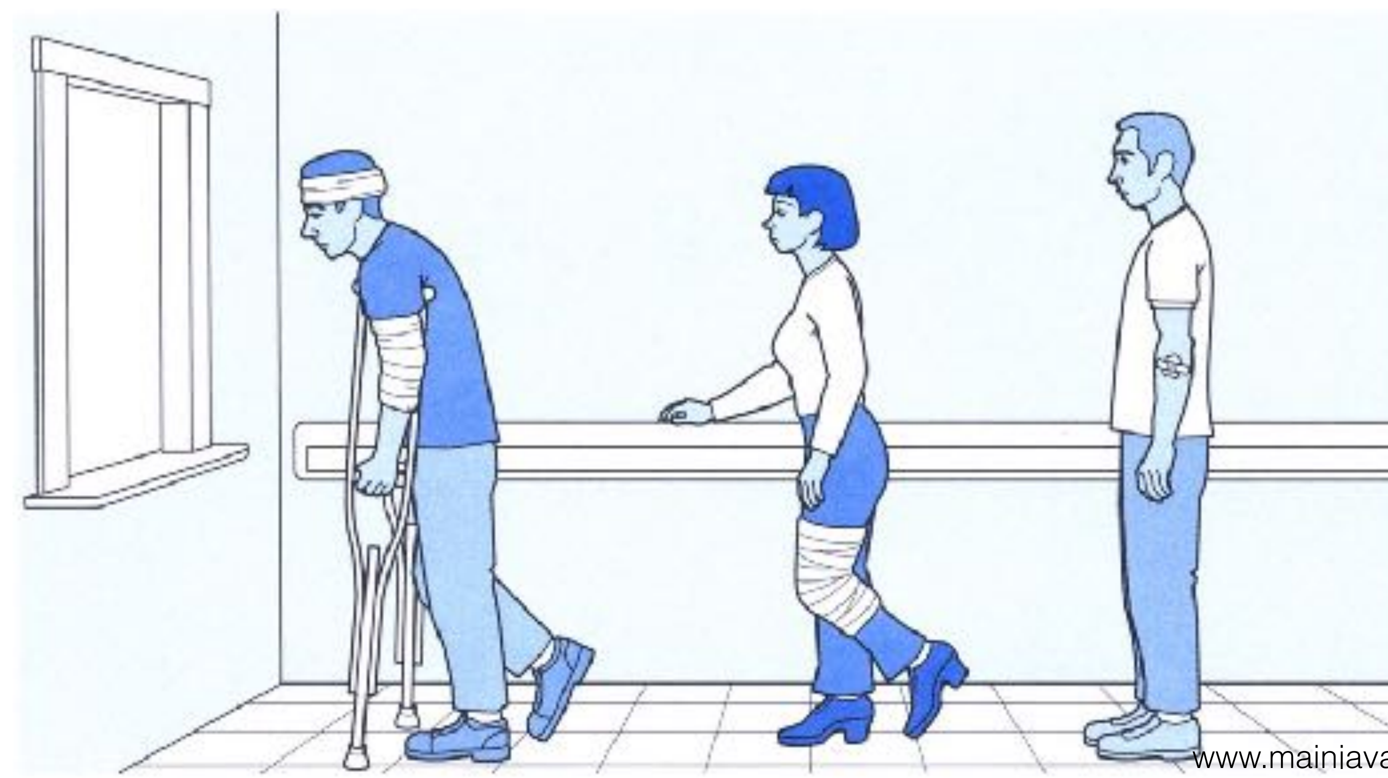
Collection Interface



Our next two topics:

- **Priority Queues**
- Hashing, **HashSets**, HashMaps

Queue vs Priority Queue



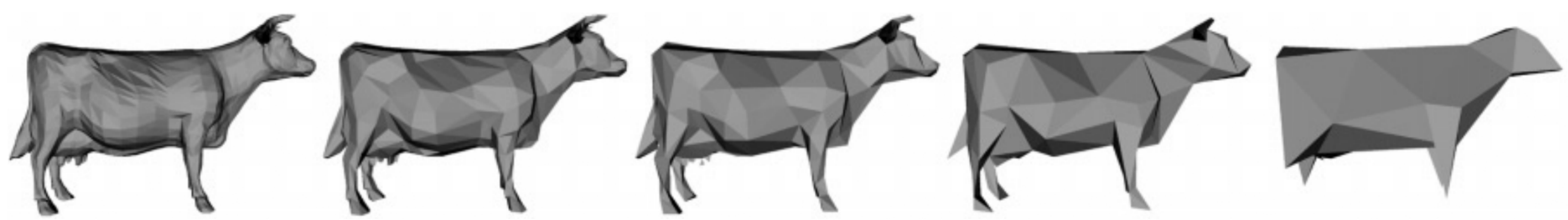
add (enqueue):
  inserts an item into the queue

remove (dequeue):
  removes the first item to be inserted (FIFO)

add (enqueue):
  inserts an item into the queue

remove (poll):
  remove the **highest-priority** item from the queue

Uses for Priority Queues



- Surface simplification [Garland and Heckbert 1997]
- Graph searching: Dijkstra's algorithm, Prim's algorithm
- Statistics: maintain largest M values in a sequence
- Graphics and simulation: "next time of contact" for colliding bodies
- AI Path Planning: A* search (e.g., Map directions)
- Operating systems: load balancing, interrupt handling
- Discrete optimization: bin packing, scheduling