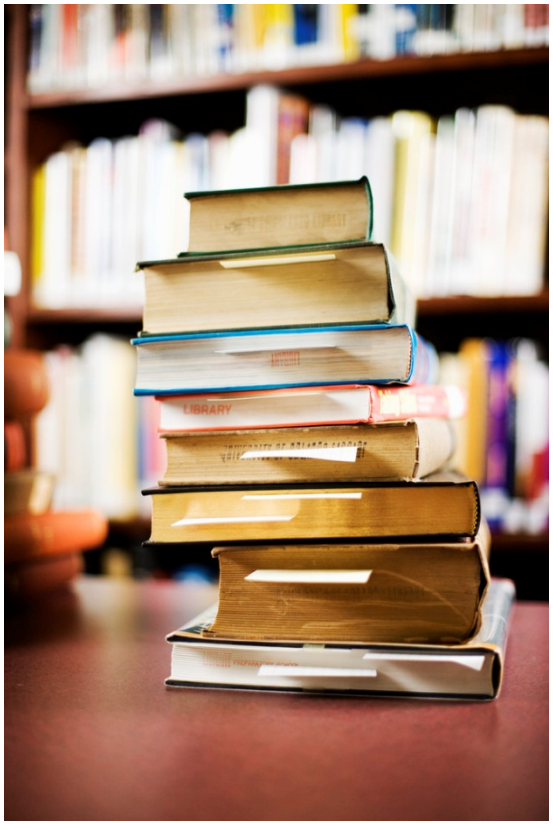
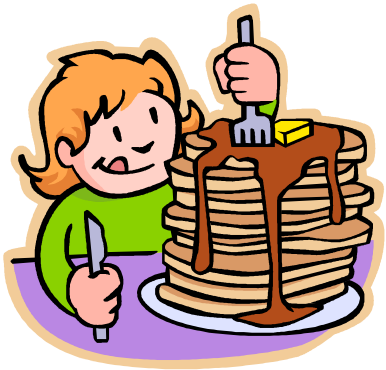
Implementing Stacks and Queues

Based on slides by Prof. Burton Ma

Stack

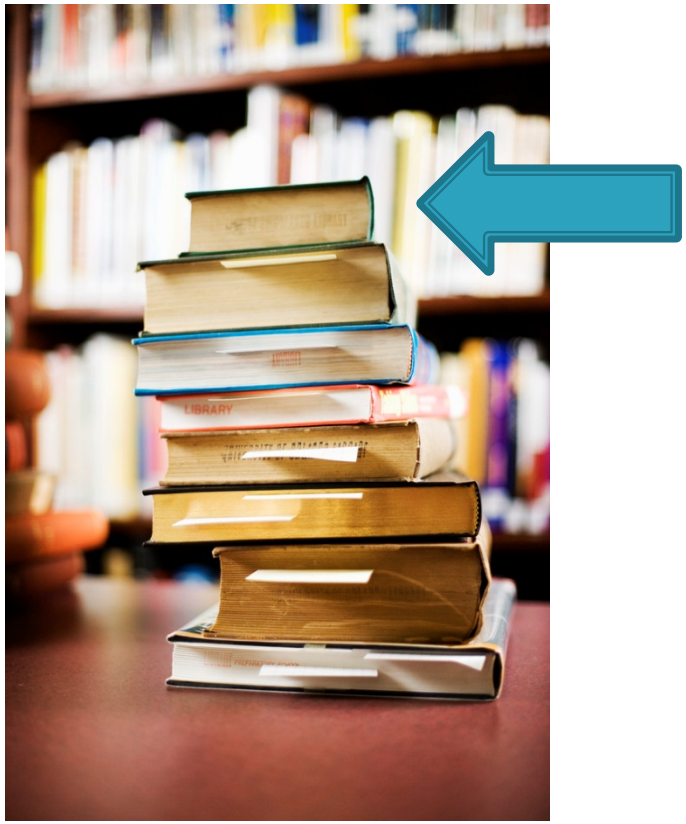
- Examples of stacks





Top of Stack

- Top of the stack



Stack Operations

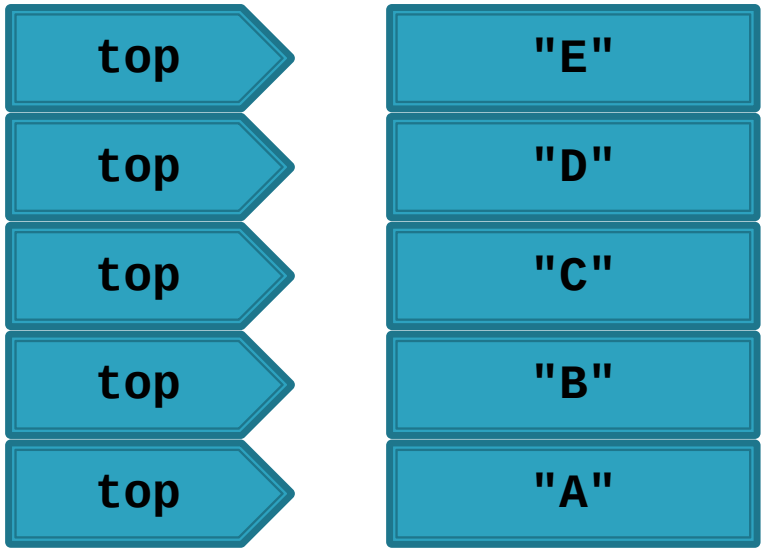
- Classically, stacks only support two operations
  1. Push
     - Add to the top of the stack
  2. Pop
     - Remove from the top of the stack

Stack Optional Operations

- Optional operations
  1. Size
     - Number of elements in the stack
  2. isEmpty
     - Is the stack empty?
  3. peek
     - Get the top element (without removing it)
  4. search
     - Find the position of the element in the stack
  5. isFull
     - Is the stack full? (for stacks with finite capacity)
  6. capacity
     - Total number of elements the stack can hold (for stacks with finite capacity)

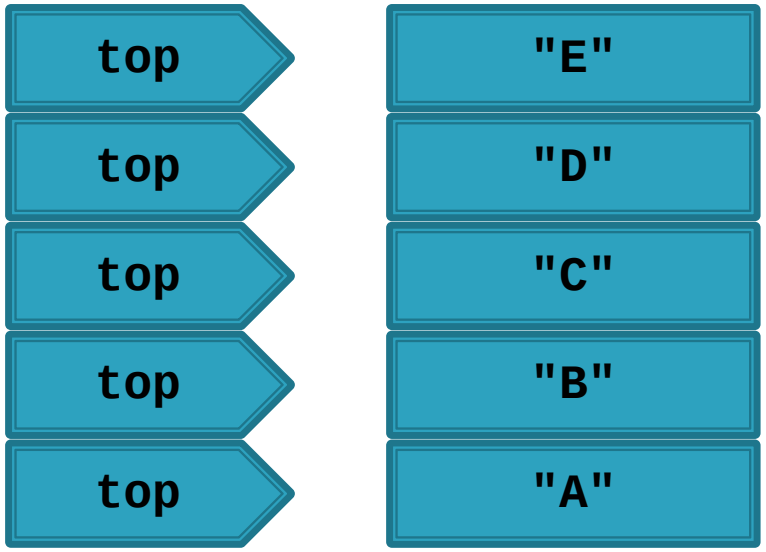
Push

1. `st.push("A")`
2. `st.push("B")`
3. `st.push("C")`
4. `st.push("D")`
5. `st.push("E")`



Pop

1. `String s = st.pop()`
2. `s = st.pop()`
3. `s = st.pop()`
4. `s = st.pop()`
5. `s = st.pop()`



LIFO

- Stack is a Last-In-First-Out (LIFO) data structure
  - The last element pushed onto the stack is the first element that can be accessed from the stack

Implementation with LinkedList

- A linked list can be used to efficiently implement a stack
- The head of the list becomes the top of the stack
  - Adding (push) and removing (pop) from the head of a linked list requires O(1) time

```
public class Stack<E>
{
  private LinkedList<E> stack;

  public Stack()
  {
    this.stack = new LinkedList<E>();
  }

  public push(E element)
  {
    this.stack.addFirst(element);
  }

  public E pop()
  {
    return this.stack.removeFirst();
  }
}
```

Implementation with ArrayList

- **`ArrayList`** can be used to efficiently implement a stack
- The end of the list becomes the top of the stack
  - Adding and removing to the end of an **`ArrayList`** usually can be performed in O(1) time

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

  public Stack()
  {
    this.stack = new ArrayList<E>();
  }

  public push(E element)
  {
    this.stack.add(element);
  }

  public E pop()
  {
    return this.stack.remove(this.stack.size() - 1);
  }
}
```

Implementations in java.util

- java.util.Stack provides a stack class

Applications

- Stacks are used widely in computer science and computer engineering
  - A call stack is used to store information about the active methods in a Java program
  - Undo/Redo
  - Back/Forward history
  - Widely used in parsing

Queue

Queue

Queue Operations

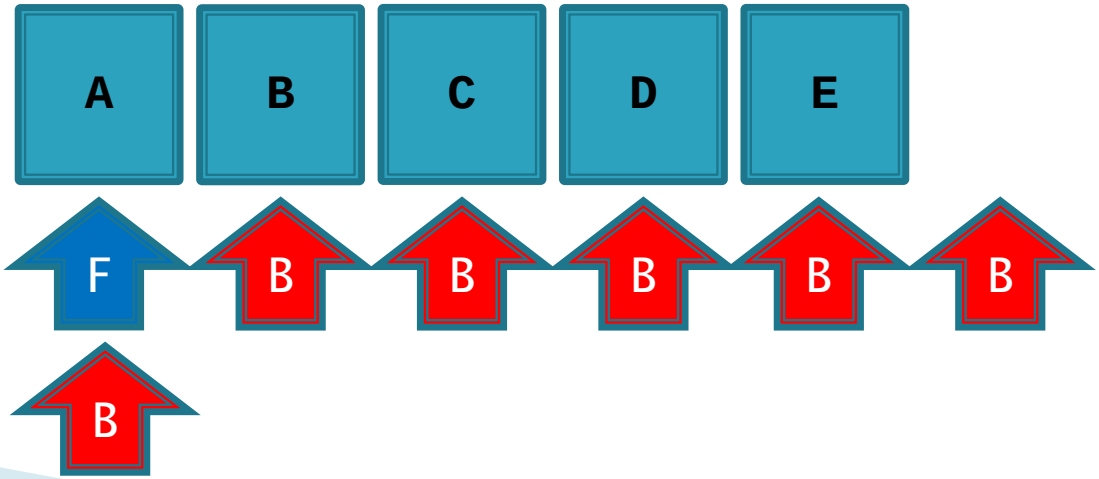
- Classically, queues only support two operations
  1. Enqueue
     - Add to the back of the queue
  2. Dequeue
     - Remove from the front of the queue

Queue Optional Operations

- Optional operations
  1. size
     - Number of elements in the queue
  2. isEmpty
     - Is the queue empty?
  3. peek
     - Get the front element (without removing it)
  4. search
     - Find the position of the element in the queue
  5. isFull
     - Is the queue full? (for queues with finite capacity)
  6. capacity
     - Total number of elements the queue can hold (for queues with finite capacity)

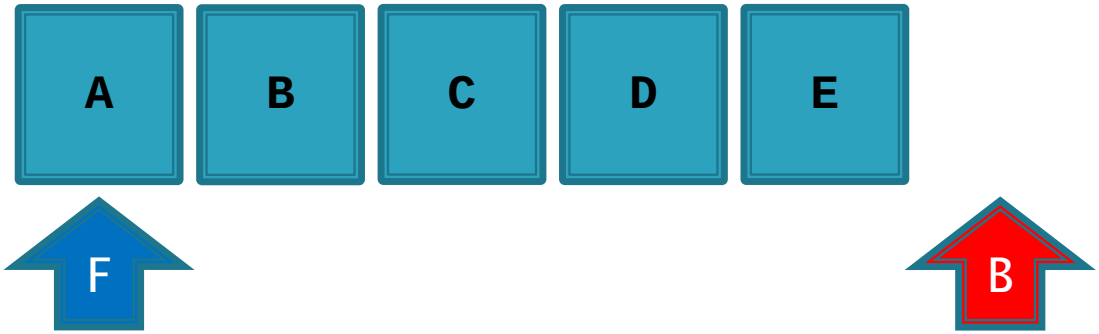
Enqueue

1. `q.enqueue("A")`
2. `q.enqueue("B")`
3. `q.enqueue("C")`
4. `q.enqueue("D")`
5. `q.enqueue("E")`



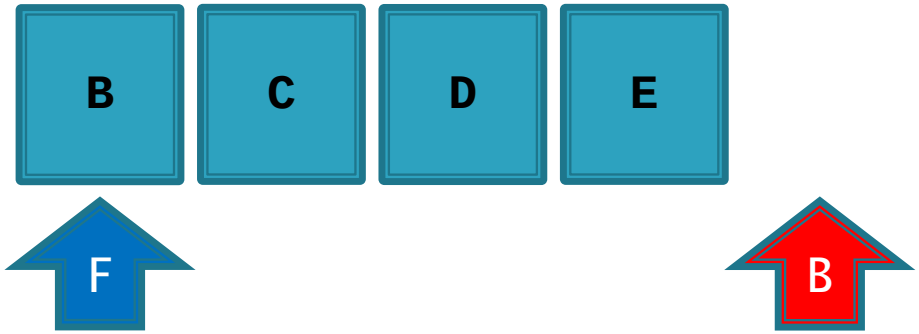
Dequeue

1. `String s = q.dequeue()`



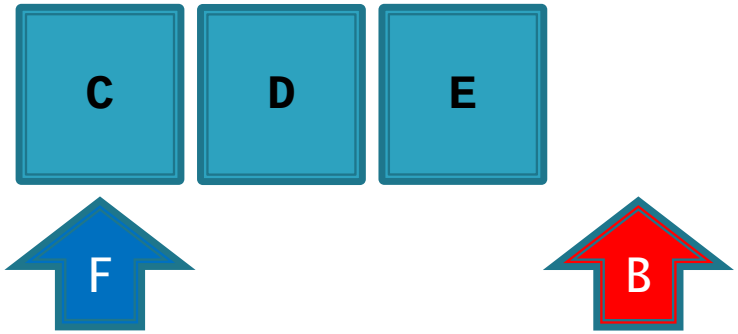
Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`



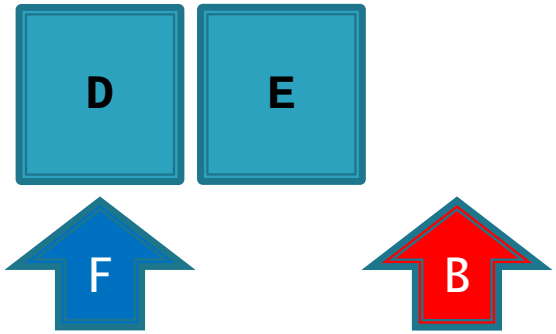
Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`
3. `s = q.dequeue()`



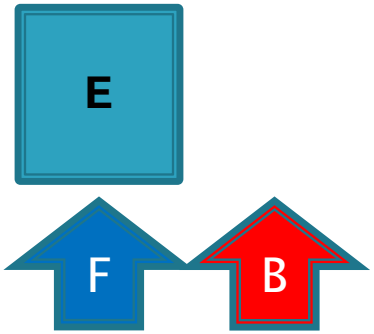
Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`
3. `s = q.dequeue()`
4. `s = q.dequeue()`



Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`
3. `s = q.dequeue()`
4. `s = q.dequeue()`
5. `s = q.dequeue()`



FIFO

- Queue is a First-In-First-Out (FIFO) data structure
  - The first element enqueued in the queue is the first element that can be accessed from the queue

Implementation with LinkedList

- A linked list can be used to efficiently implement a queue as long as the linked list keeps a reference to the last node in the list
  - Required for enqueue
- The head of the list becomes the front of the queue
  - Removing (dequeue) from the head of a linked list requires O(1) time
  - Adding (enqueue) to the end of a linked list requires O(1) time if a reference to the last node is available
- java.util.LinkedList is a doubly linked list that holds a reference to the last node

```
public class Queue<E>
{
  private LinkedList<E> q;

  public Queue()
  {
    this.q = new LinkedList<E>();
  }

  public enqueue(E element)
  {
    this.q.addLast(element);
  }

  public E dequeue()
  {
    return this.q.removeFirst();
  }
}
```

Implementation with LinkedList

- Note that there is no need to implement your own queue as there is an existing interface
  - The interface does not use the names enqueue and dequeue however

java.util.Queue

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

boolean	**add(E e)** Inserts the specified element into this queue...
E	**remove()** Retrieves and removes the head of this queue...
E	**peek()** Retrieves, but does not remove, the head of this queue...

- Plus other methods
  - http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html

java.util.Queue

- **`LinkedList`** implements **`Queue`** so if you ever need a queue you can simply use:
  - E.g. for a queue of strings

```
Queue<String> q = new LinkedList<String>();
```

Queue applications

- Queues are useful whenever you need to hold elements in their order of arrival
  - Serving requests of a single resource
    - Printer queue
    - Disk queue
    - CPU queue
    - Web server