# Lecture 9:
# Stacks and Queues

# Wholeness of the Lesson

Stacks and Queues are, essentially, a special kind of list with a highly restricted interface that permit rapid insertion and rapid access to elements, according to a "last in, first out" (Stacks) or "first in, first out" (Queues) scheme. These data structures express the MVS principle that creation emerges in the collapse of infinity to a point.

# The STACK ADT

- **Definition:** A STACK is a LIST in which insertions and deletions can occur relative to just one designated position (called the *top of the stack*).
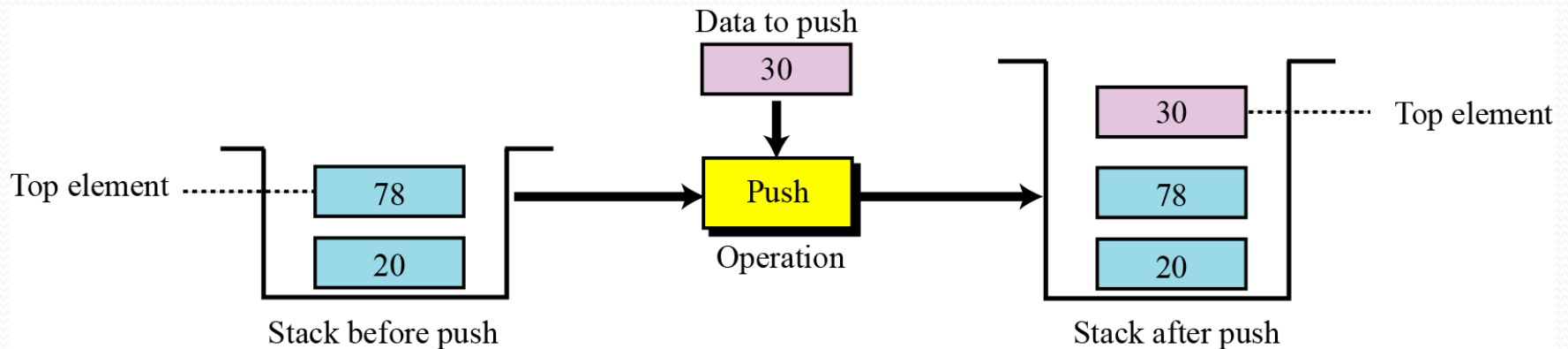
- **Example of Stack:**

# The STACK ADT

- **Operations:**

| pop | remove top of the stack and return  this object) |
|-----|--------------------------------------------------|
| push | insert object as new top of stack |
| peek | view object at top of the stack without removing it |

# The STACK ADT

- **push operation:**



Data to push

| 30 |

Top element ········ | 78 |
| 20 |

Stack before push

Push
Operation

| 30 | ········ Top element
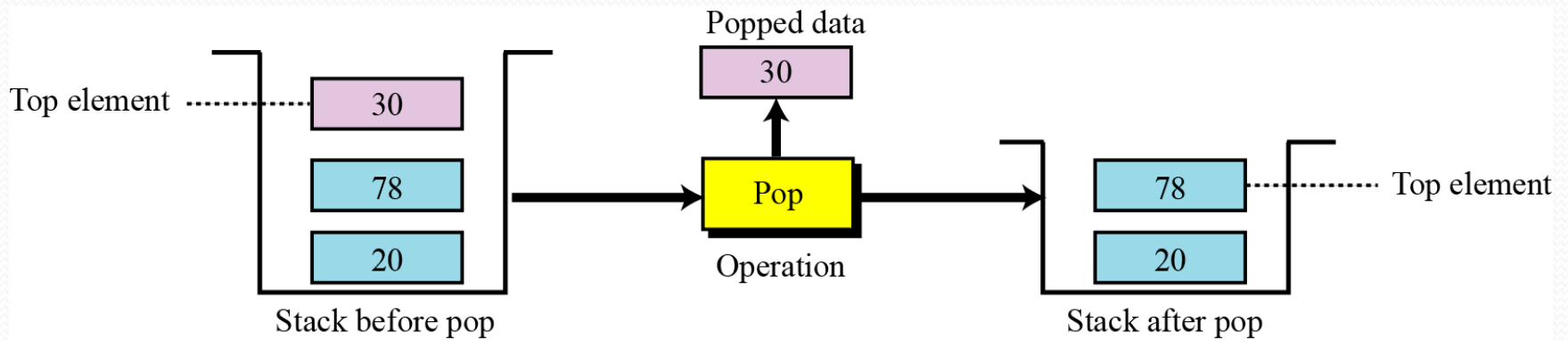| 78 |
| 20 |

Stack after push

# The STACK ADT

- **pop operation:**

# Implementation of STACK Using an Array

- **Usual strategy**: Designate the rightmost array element to be the top of the stack.
- **Detail**: To avoid traversing the array in search of the current top of the Stack, maintain a pointer to the rightmost element.
- **Advantage**:
  - Avoids the usual cost of copying array elements that is required in insertion and deletion of arbitrary array elements
- **Disadvantage**:
  - If usage requires many more pushes than pops, the underlying array will have to be resized often, and this is costly
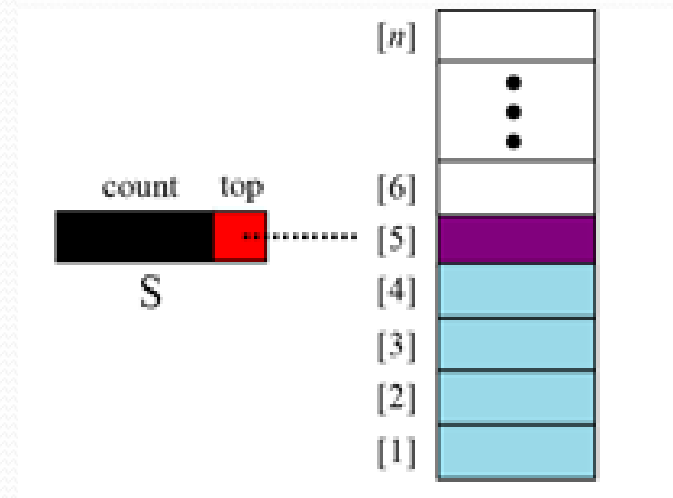
# Implementation of STACK Using a Linked List

- Imagine the front of the LinkedList as the top of the stack. The push operation can be implemented by simply calling the Linked List's `insert(data,0)` operation.

- The peek operation is equivalent to *find0th* (in a LinkedList, it is the call `get(0)`).

- The pop operation is equivalent to *find0th* followed by a call to `remove(0)`.

front of linkedlist
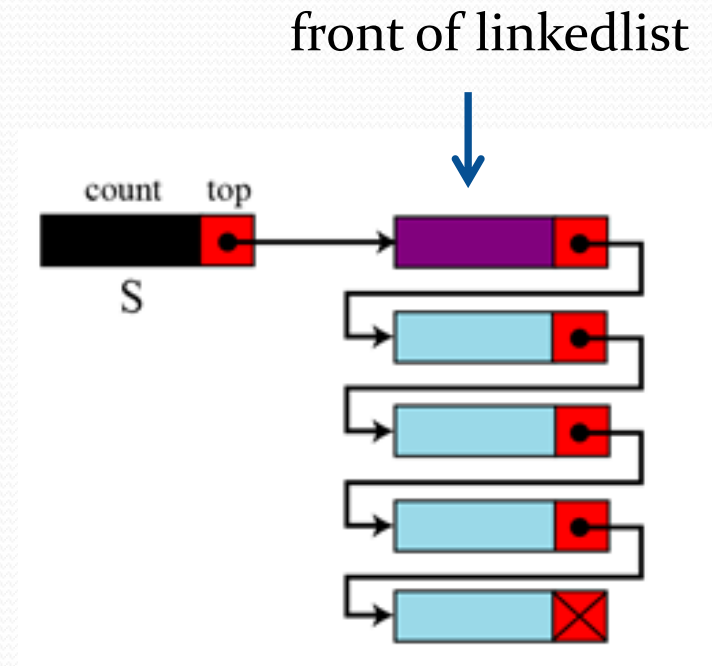
# Java's Implementation of Stack

- The Java distribution comes with a `Stack` class, which is a subclass of `Vector`.

- `Vector` is an array-based implementation of `List`. Therefore, for implementations that require many more pushes than pops, a stack based on a Linked List should be used.

- Exercise: Implement your own class `MyStringStack` that uses `MyStringLinkedList`.

# Application of Stacks: Symbol Balancing

- A Stack can be used to verify whether all occurrences of symbol pairs (for symbol pairs like (), [], {}) are properly matched and occur in the correct order.

- For example:

| VALID INPUTS | INVALID INPUTS |
|---|---|
| { } | { ( } |
| ( { [ ] } ) | ( [ ( ( ) ] ) |
| { [ ] ( ) } | { } [ ] ) |

# Application of Stacks: Symbol Balancing

- The following procedure can be used:
  - Begin with an empty Stack
  - Scan the text (will ignore all non-bracketing symbols)
  - When an open symbol is read, push it
  - When a closed symbol is read, pop the Stack –
    i. if the stack is empty (so it can't be popped) return false.
    ii. if the popped symbol doesn't match the symbol just read, return false.
  - After scanning is complete, if the Stack is not empty, return false.

# The QUEUE ADT

- **Definition.** Like a STACK, a QUEUE is a specialized LIST in which insertions may occur only at a designated position (the *back*) and deletions may occur only at a designated position (the *front*).
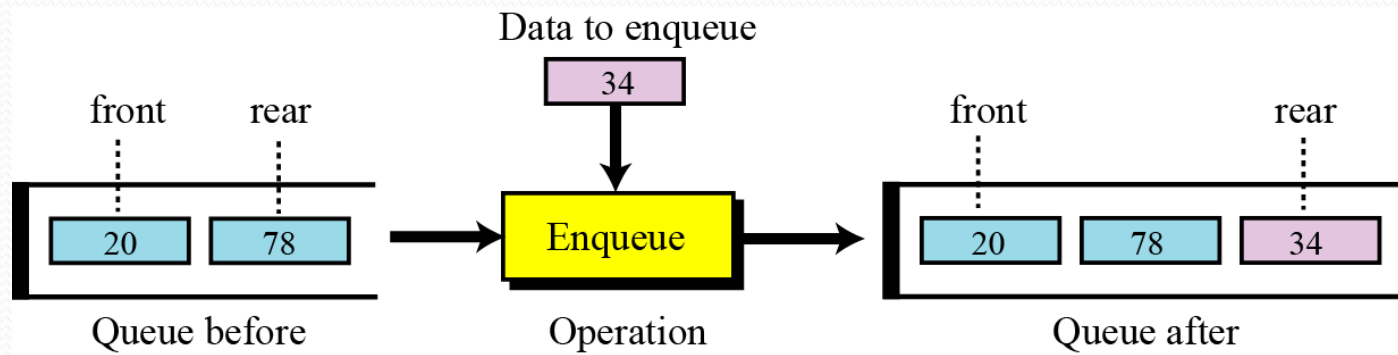


A queue of people

A computer queue
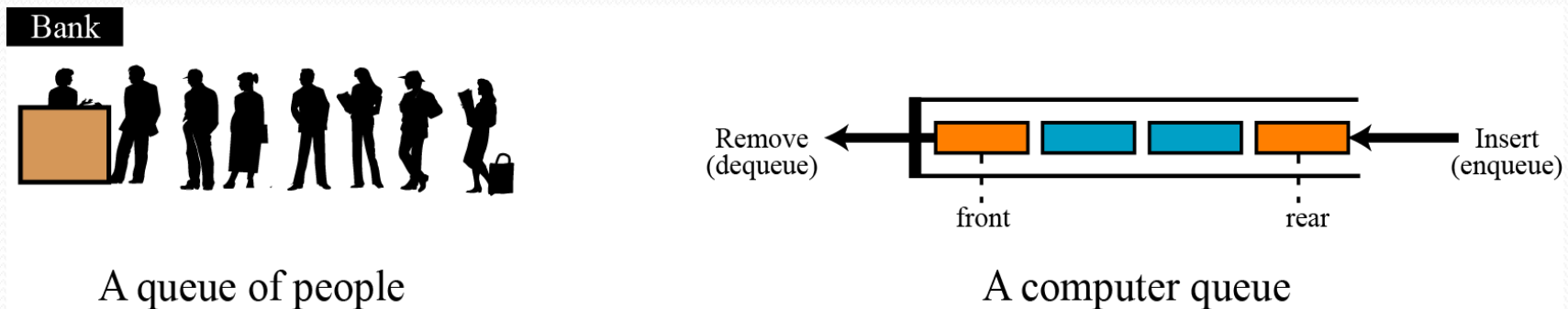
# The QUEUE ADT

- **Operations**:

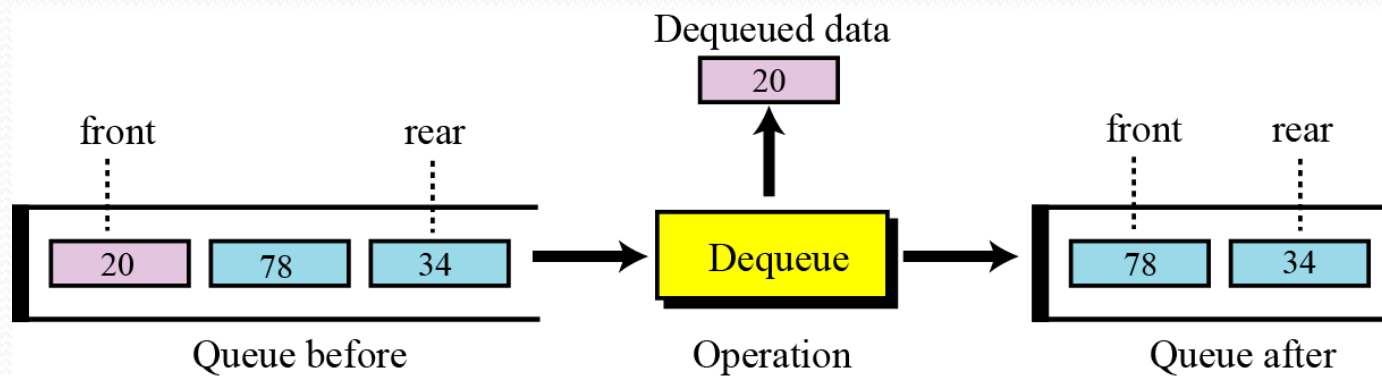| dequeue | remove the element at the front (usually also returns this object) |
|---------|-------------------------------------------------------------------|
| enqueue | insert object at the back |
| peek | view object at front of queue without removing it |

# The QUEUE ADT

- **enqueue operation**:

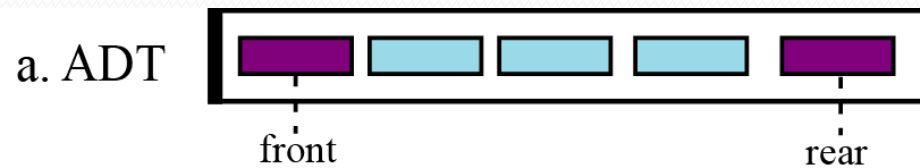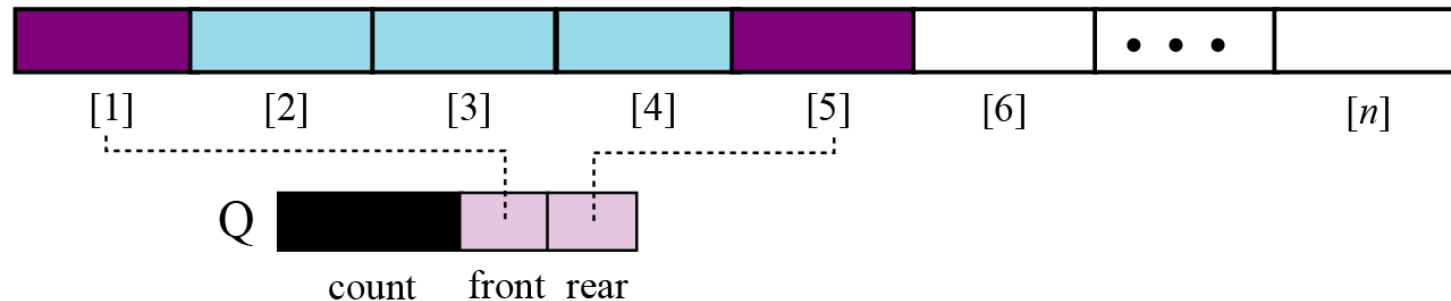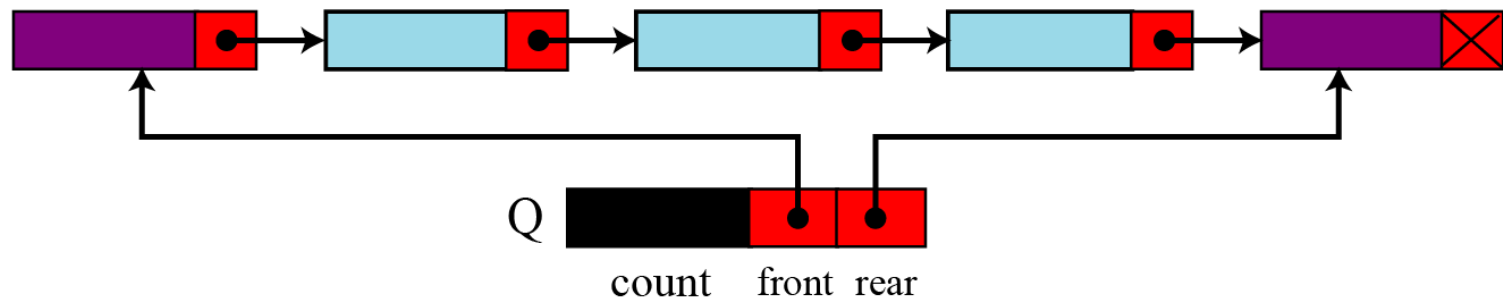# The QUEUE ADT

- **dequeue operation:**

# Implementations of QUEUES



a. ADT

front       rear

b. Array implementation

[1]    [2]    [3]    [4]    [5]    [6]    [$n$]

Q

count   front   rear

c. Linked list implemenation

Q

count   front   rear

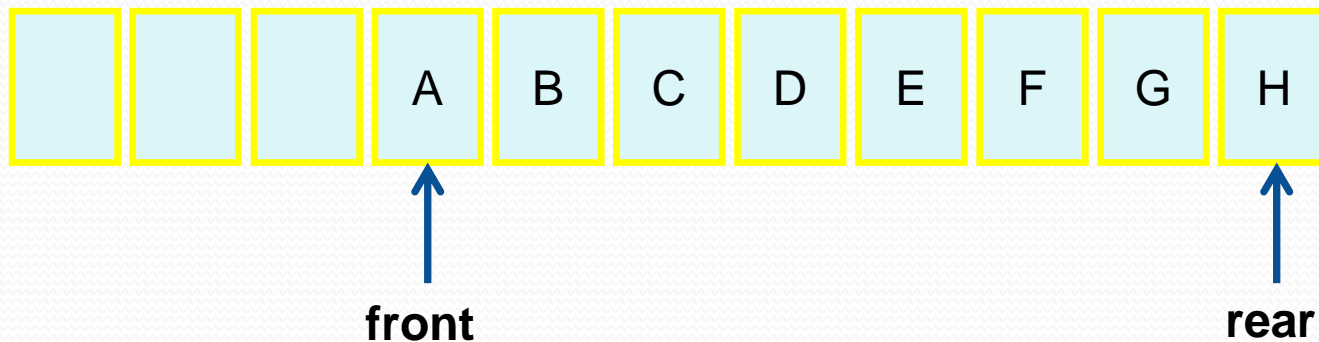16

# Implementations of QUEUES

- **Using a Linked List**
    - The enqueue operation is equivalent to adding the element to the last of a LinkedList.
    - The dequeue operation is equivalent to removing the element at the front of a LinkedList.
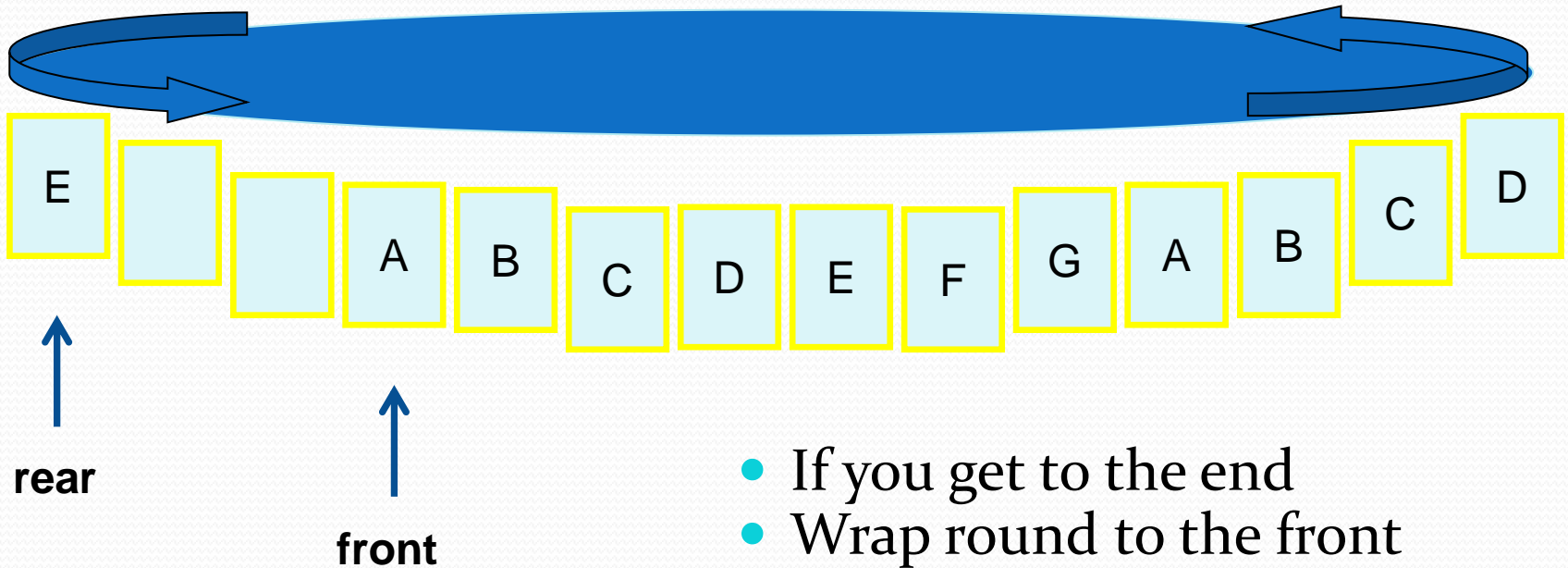
# Implementations of QUEUES

- **Using an Array**
  - Need to maintain pointers to front and back elements
  - Repeated enqueuing will fill the right half of the array prematurely—solution is to wrap around to the front.

|   |   |   | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|---|---|

**front**                                   **rear**

# Circular Queue

E | | | A | B | C | D | E | F | G | A | B | C | D

**rear**

**front**

- If you get to the end
- Wrap round to the front
- Until you hit the front pointer
- It's not really a circle!

- **Java's Implementation**
  - In j2se5.0, an interface `Queue<E>` (implemented by `LinkedList<E>`) is provided, with these declared operations:
    - `E peek()` –  returns but does not remove the front of the queue
    - `void add(E obj)` –  same as enqueue
    - `E remove()` –  returns and removes the front of the queue (same as dequeue)

# Main Point

The Stack ADT is a special ADT that supports insertion of an element at "the top" and the removal of the top element, by way of operations *push* and *pop*, respectively. Similarly, the Queue ADT is a special ADT that supports insertion of an element at "the rear" (called *enqueuing*) and removal of an element from the "front" (called *dequeuing*). Both ADT's, when implemented properly, are extremely efficient. Sun provides a Stack class and a Queue interface in its Collections API.

Stacks and Queues make use of the MVS principle that the dynamism of creation arises in the concentration of dynamic intelligence to a point value ("collapse of infinity to a point"); stacks and queues achieve their high level of efficiency by concentrating on a single point of input (top of stack or rear of queue) and a single point of output (top of stack or front of queue).

# Connecting the Parts of Knowledge With the Wholeness of Knowledge

*Collapse of infinity to a point embodied in Stacks and Queues*

1. Lists may be used as an all-purpose collection class. Nearly any need for storing collections of objects can be met by using some kind of list, though in some cases, other choices of data structures could improve performance. Lists have a more "unbounded" range of applicability.

2. Stacks and Queues are extremely specialized data structures, designed to accomplish (primarily) two operations with optimum efficiency. These data structures have a restricted range of applicability that is like a "point".

___

3. **Transcendental Consciousness:** Transcendental Consciousness is the unbounded value of awareness.

4. **Wholeness moving within itself**: In Unity Consciousness, creation is seen as the teraction of unboundedness and point value: the unbounded collapses to its point value; point value expands to infinity; all within the wholeness of awareness.