Lesson 9
Stacks and Queues:

Pure Knowledge Has
Infinite Organizing Power

Wholeness Statement

Knowledge of data structures allows us to pick the most appropriate data structure for any computer task, thereby maximizing efficiency.

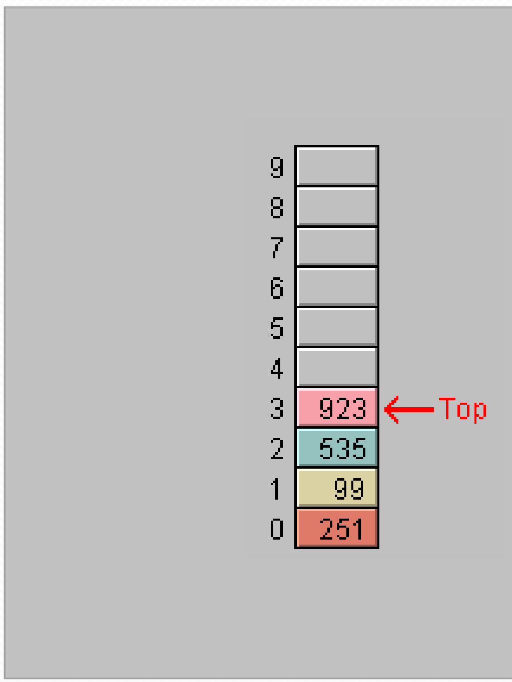
Pure knowledge has infinite organizing power, and administrate the whole universe with minimum effort.

Stack (LIFO)

- A STACK is a LIST in which insertions and deletions can occur relative to just one designated position (called the *top of the stack*).
- Applications :Used for recursive method calls, evaluate expressions, backtracking approaches.
- Stack Operations

pop	remove top of the stack and return the object)
push	insert object in the top of stack
peek	view object at top of the stack without removing it

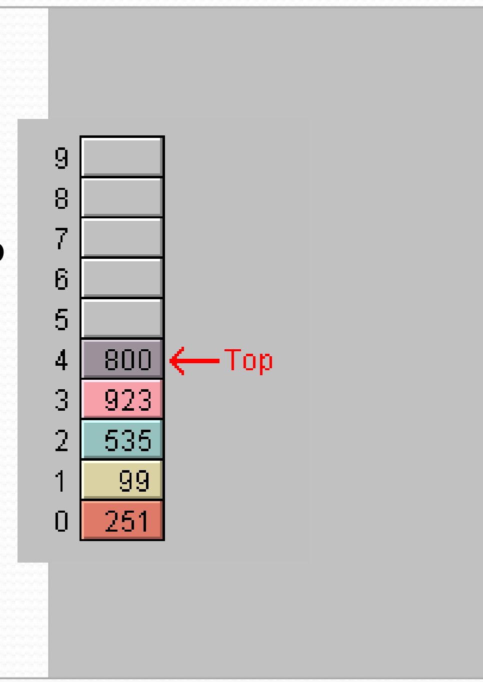
Stack : An array implementation



→

after pushing the element 800

→



Implementing Stacks and Queues

- Use an array to implement Stack
- Use a linked list to implement Queue
- Since the insertion and deletion operations on a stack are made only at the end of the stack, using an array to implement a stack is more efficient than a linked list.
- Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list.

Array Implementation of a Stack

- Designate the top of the stack to be the element with the highest index.
- Declare an int field to hold the index of the top element of the stack
- *push* operation
  - Increment the index of the top element
  - Store the element in the array
- *pop* operation
  - Decrement the index of the top element
  - Return the top element of the stack
- *peek* operation
  - Return top element of the stack

  See :ArrayStackDemo.java

Implementation of STACK

- The standard Java distribution comes with a Stack class, which is a subclass of Vector.
- Vector is an array-based implementation of LIST.

Class Stack

`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

```
Stack<Integer> s = new Stack<Integer>();
s.push(42);
s.push(-3);
s.push(17);       // bottom [42, -3, 17] top

System.out.println(s.pop()); // 17
```

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.

Demo code

StackDemo.java

ArrayStackDemo.java

ExpressionDemo.java

Main Point 1

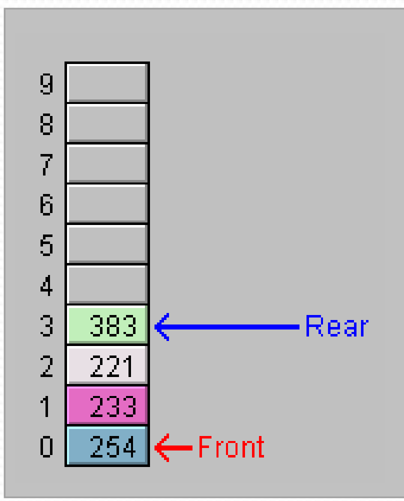
Stacks are data structures that allow very specific and orderly insertion, access, and removal of their individual elements; only the top element can be inserted, accessed, or removed. Similarly, nature is orderly; an apple seed will yield only an apple tree.

Queue (FIFO)

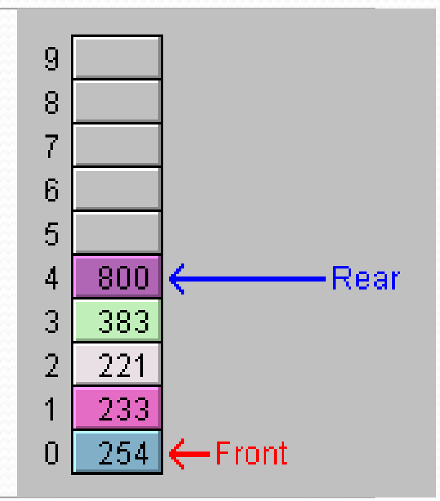
- **Definition.** Like a STACK, a QUEUE is a specialized LIST in which insertions may occur only at a designated position (the rear) and deletions may occur only at a designated position (the *front*).
- Applications : Used for printer queues, queue of network data packets to send, Breadth First Search.
- Queue 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

Queue : An array implementation



→
after enqueuing (adding) the element 800
→



Queue Implementation

- Can be based on either an array or a linked list
- Linked List
  - Implementation is straightforward
- Array
  - Need to maintain pointers to index of front and rear elements and need to enlarge the array

The Queue Operations

- Create an empty queue
  - new object constructor call
- Determine whether the queue is empty
  - isEmpty
- Add an item to the end of the queue
  - enqueue
- Remove an item from the front of the queue
  - dequeue
- Retrieve the item at the front of the queue
  - peek
- Remove all items from the queue
  - removeAll

Predefined Queue Interface

`Queue<`**`E`**`>()`	constructs a new Queue with elements of type **E**
`add(Value)`	place the given value at back of queue
offer(Value)	This inserts the specified element into the queue.
remove()	removes value from front of queue and returns it; throws a NoSuchElementException if queue is empty
poll()	This method retrieves and removes the head of this queue, or return null if this queue is empty.
peek()	returns front value from queue without removing it; returns null if queue is empty
element()	This method retrieves the head of the queue. Throws a NoSuchElementException if queue is empty
size()	returns number of elements in queue
isEmpty()	returns true if queue has no elements

Queue in Java

Refer : QueueDemo.java, LinkedQueueDemo.java

Example :

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(42);
q.add(-3);
q.add(17);          // front [42, -3, 17] back

System.out.println(q.remove());    // 42
```

- **IMPORTANT**: When constructing a queue you must use a new `LinkedList` instead of a new `Queue` because it is an Interface.

Application of the Queue ADT

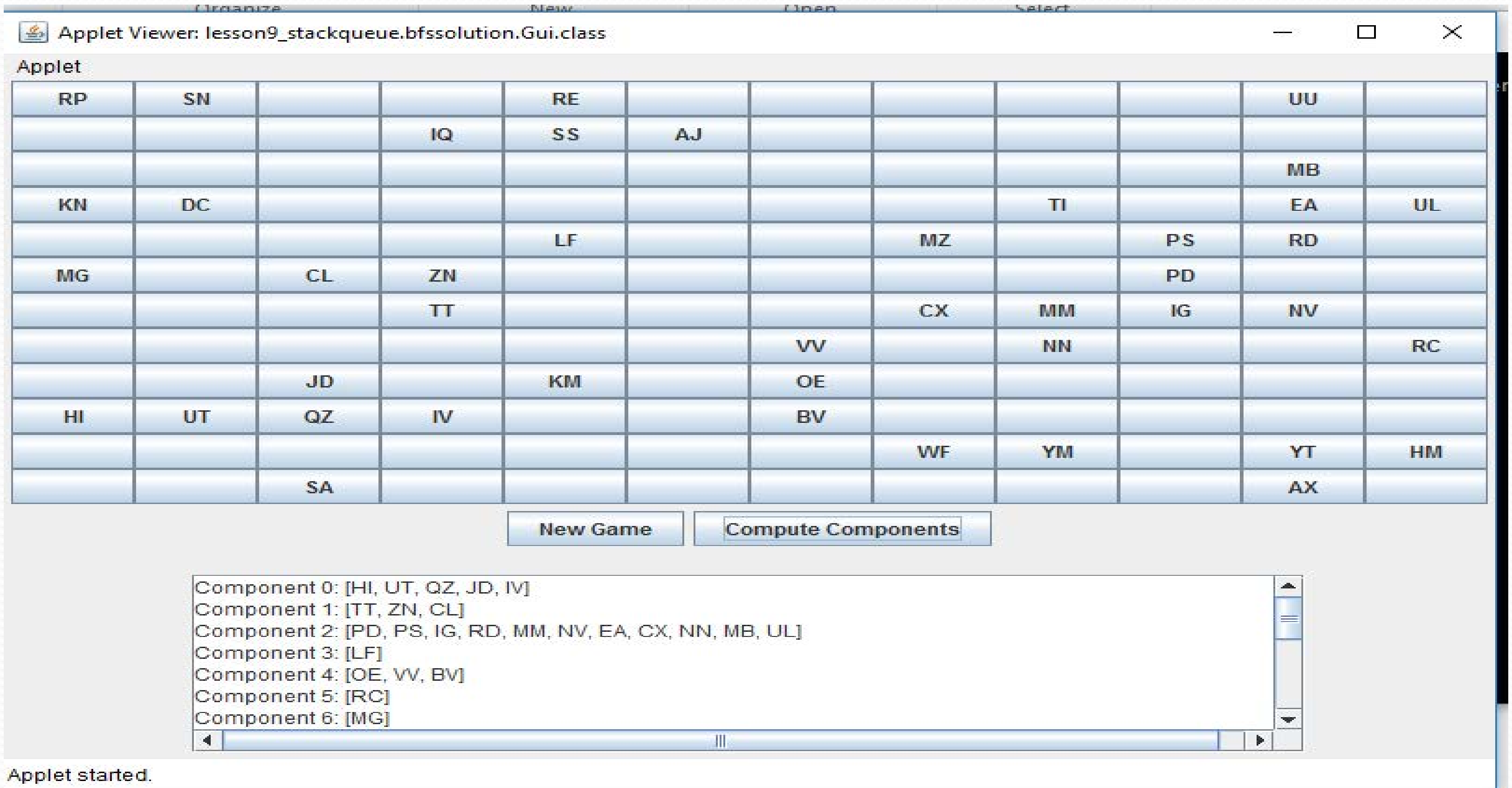
- Recognizing Palindromes
  - Strings that read the same from left to right as they do from right to left
  - E.g., aba, abba
- Breadth-First Search

Algorithm for Recognizing Palindromes

- Create an empty queue or stack
- Scan and insert characters one by one into both the queue or stack
- Remove and compare each character from the stack or queue
  - If they are different at any point, then the string is not a palindrome

Give an example – cbc, and cbc

Breadth-First Search Applet View



- Breadth-first search can be used to list all the "connected components"(nearest neighbors) in this grid – and more generally, in any graph.
- Queue is used to store the nearest component form left to right or from top to bottom. For instance, ED, LA, UF, KE form a connected component.

- ***Procedure:***

Output: A collection of component lists

Algorithm:

Look for the next unvisited cell Co (moving from left to right and then top to bottom) and mark Co as "visited"

a) Create a queue Q (to handle the component that contains Co)
b) Place all cells adjacent to Co in Q and add Co to a new component list
c) While Q is not empty
   - Remove the cell at the front of Q – call it C – and mark it as "visited"
   - Insert into Q all cells adjacent to C
   - Add C to the current component list
d) Look for the next unvisited cell and repeat steps (a) – (d) until there are no more unvisited cells.

Understand the concept by reading this article : http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/bfs.html

Main Point 2

The Queue ADT is a special ADT that supports access or removal from the front of the queue and insertion at the end. Queues achieve their high level of efficiency by concentrating on a single point of insertion (end) and a single point of removal and access (front). In a similar way, the dynamism of creation arises from the concentration of dynamic intelligence at a point.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

1. There are infinitely many ways to design large programs.
2. With the knowledge of data structures such as Lists, Stacks, and Queues, one can design programs that run most efficiently and simply.
3. <u>Transcendental Consciousness</u> is the unbounded field of pure awareness.
4. <u>Wholeness moving within itself</u> : In Unity Consciousness, creation is seen as the interaction of unboundedness and point value: the unbounded collapses to its point value; point value expands to infinity, all within the wholeness of awareness.