
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS390 Foundamental Programming Practices (FPP)
Professor Paul Corazza

Lecture 9:
Stacks and Queues

Wholeness of the Lesson

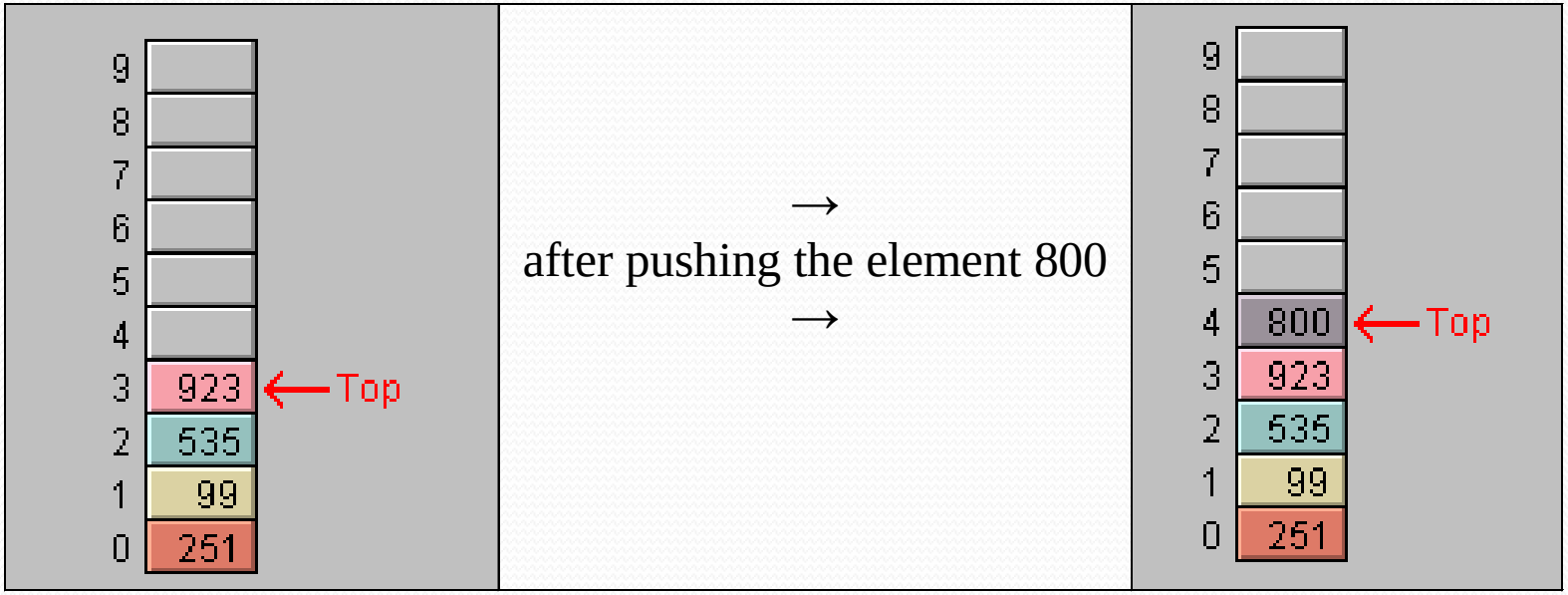
Stacks and Queues are, essentially, a special kind of list with a highly restricted interface that permit rapid insertion and rapid access to elements, according to a "last in, first out" (Stacks) or "first in, first out" (Queues) scheme. These data structures express the MVS principle that creation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

- **Model:**



- The Stack operations (under usual implementations) are extremely efficient.

Implementation of STACK Using a Linked List

- The usual add operation in a LinkedList adds the new element to the front of the list. Therefore, the push operation can be implemented by simply calling the Linked List's add 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)`.

Implementation of STACK Using an Array

- **Usual strategy**: Designate the rightmost array element to be the top of the stack.
- **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
- **Detail**: To avoid traversing the array in search of the current top of the Stack, maintain a pointer to the rightmost element.

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.

- 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) report an error
    ii. if the popped symbol doesn't match the symbol just read, report an error
  - After scanning is complete, if the Stack is not empty, report an error

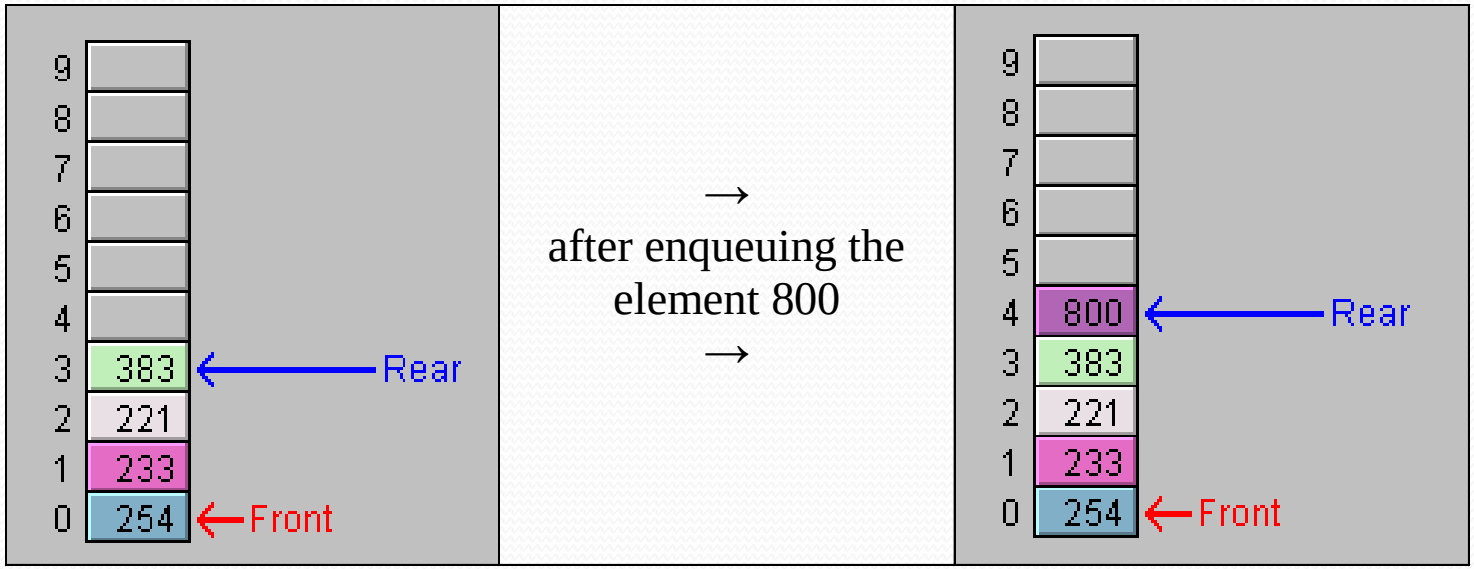
The QUEUE ADT

- **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*).

- **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

- **Model**

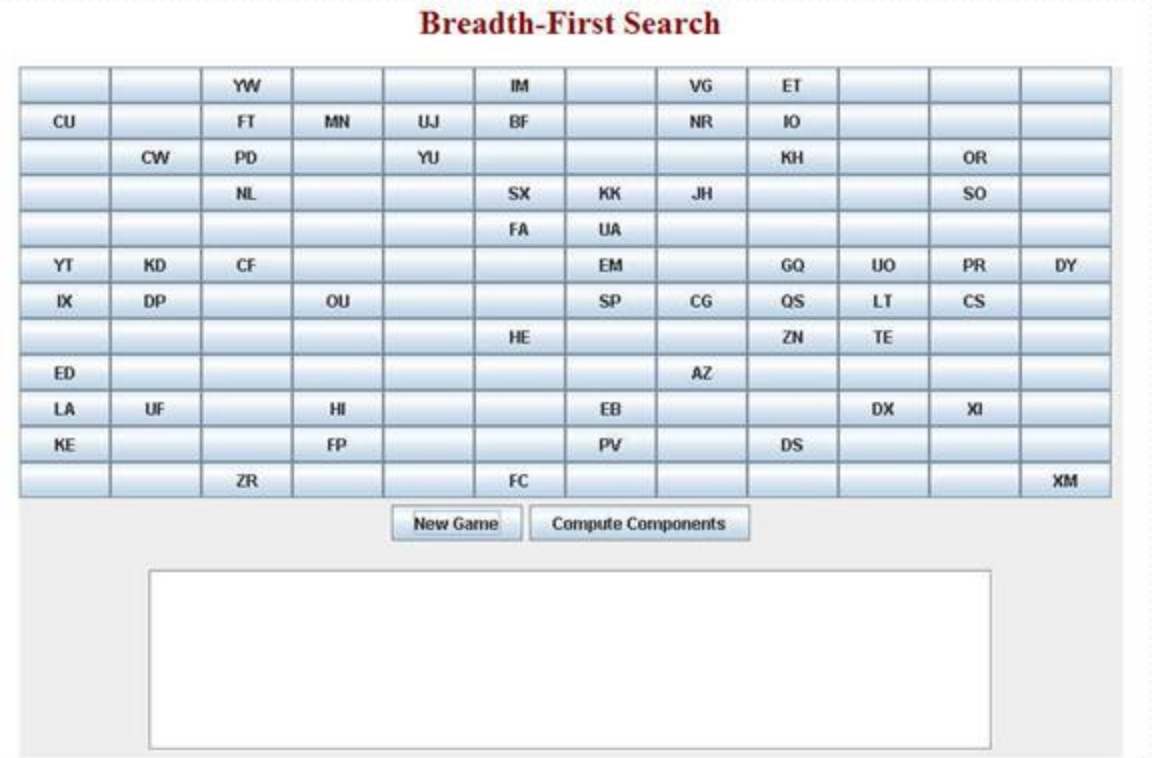

9
8
7
6
5
4
3 383
2 221
1 233
0 254
Rear
Front
→
after enqueuing the
element 800
→
9
8
7
6
5
4 800
3 383
2 221
1 233
0 254
Rear
Front

Implementations of Queues

- **Using a Linked List**
  - Implementation is straightforward
  - Good idea to use a circular Linked List to facilitate fast enqueuing
- **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.

- **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)

An Application of Queues: Breadth-First Search



- Breadth-first search can be used to list all the "connected components" in this grid – and more generally, in any graph. For instance, ED, LA, UF, KE form a connected component.

- *Procedure:*

  Output: A collection of component lists

  Algorithm:

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

  a) Create a queue Q (to handle the component that contains S)
  b) Place all cells adjacent to S in Q and add S 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.

Breadth-First Search

		YW			IM		VG	ET			
CU		FT	MN	UJ	BF		NR	IO			
	CW	PD		YU				KH		OR	
		NL			SX	KK	JH			SO	
					FA	UA					
YT	KD	CF				EM		GQ	UO	PR	DY
IX	DP		OU			SP	CG	QS	LT	CS	
					HE			ZN	TE		
ED							AZ				
LA	UF		HI			EB			DX	XI	
KE			FP			PV		DS			
		ZR			FC						XM

New Game | Compute Components

Component 0: [KH, IO, ET, NR, VG]
Component 1: [NL, PD, FT, CW, YW, MN, UJ, YU, BF, IM]
Component 2: [UA, KK, FA, EM, SX, JH, SP, CG, QS, GQ, ZN, LT, UO, TE, CS, PR, DY]
Component 3: [VG]
Component 4: [OR, SO]
Component 5: [HI, FP]
Component 6: [CG]
Component 7: [UO]

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 dyamic 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.