MAHARISHI UNIVERSITY of MANAGEMENT

Engaging the Managing Intelligence of Nature

Computer Science Department

CS390 Fundamental 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 permits 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 Maharishi Vedic Science 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 a Stack in the Real World:**

a stack of dishes

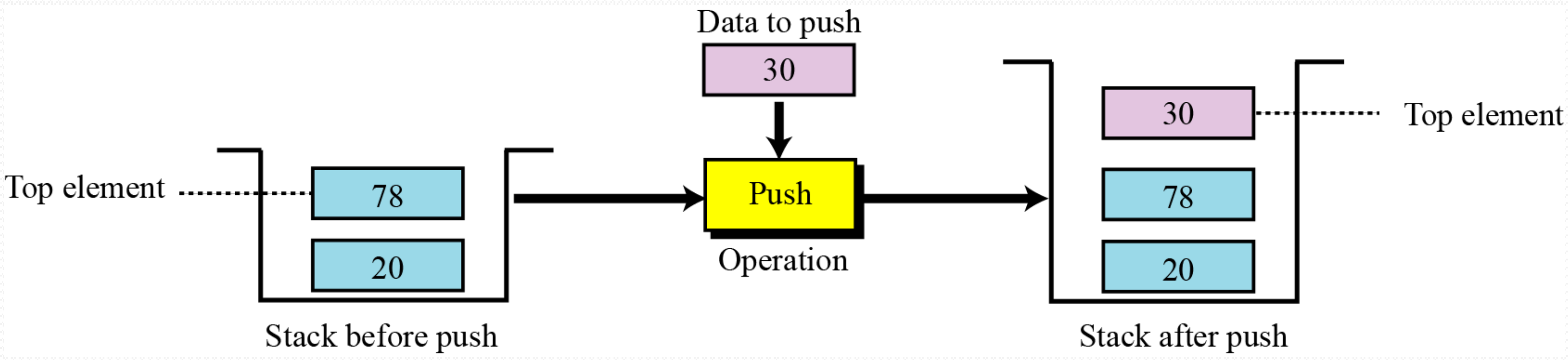
The STACK ADT

- **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

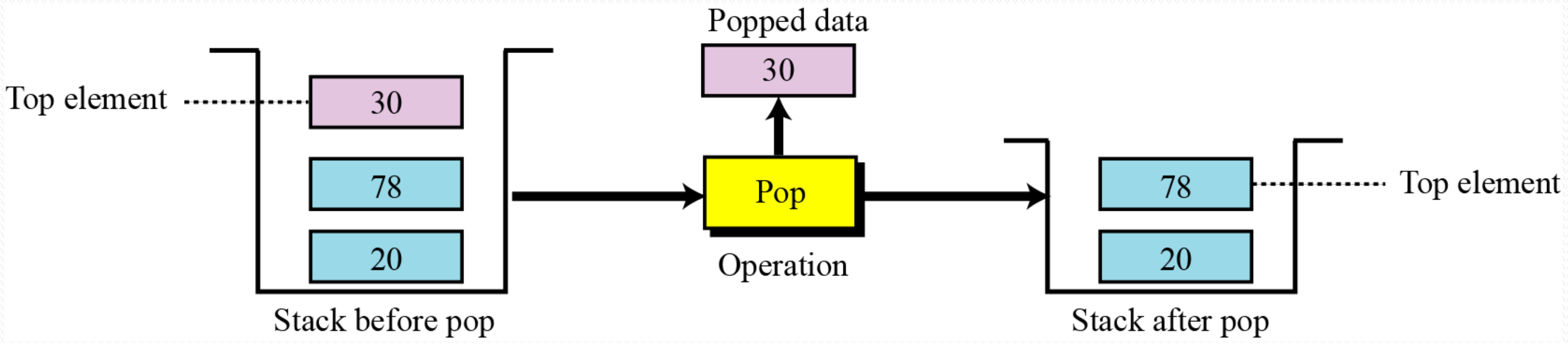
The Stack ADT

- **push operation:**



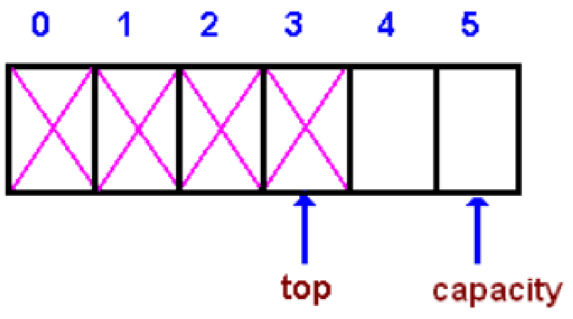
The Stack ADT

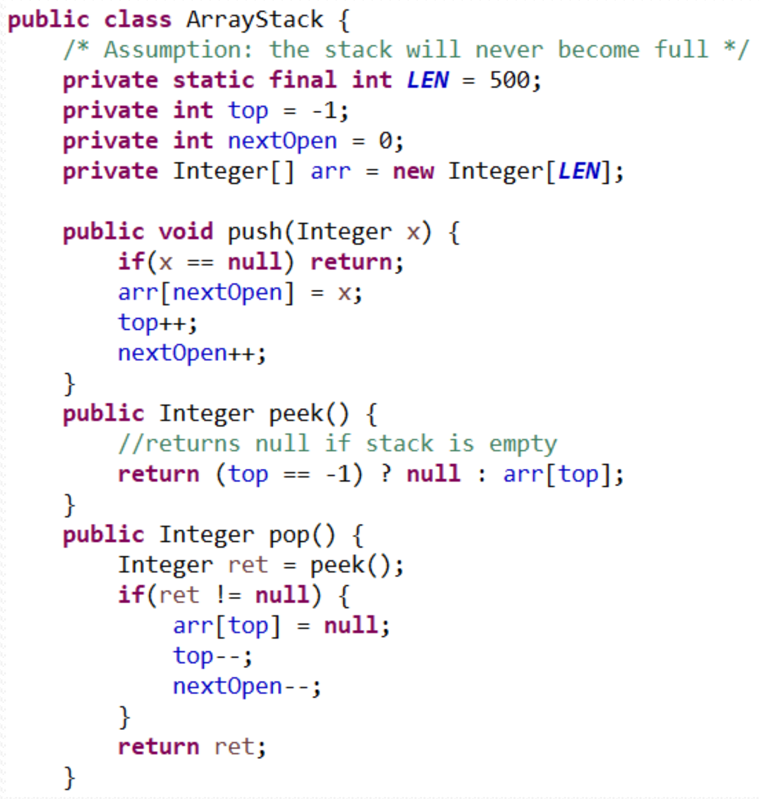
- **pop operation:**



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 (the "top").
- **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.
  - <u>Best Practice</u>: If resizing might become necessary, choose a different implementation





```java
public class ArrayStack {
    /* Assumption: the stack will never become full */
    private static final int LEN = 500;
    private int top = -1;
    private int nextOpen = 0;
    private Integer[] arr = new Integer[LEN];

    public void push(Integer x) {
        if(x == null) return;
        arr[nextOpen] = x;
        top++;
        nextOpen++;
    }
    public Integer peek() {
        //returns null if stack is empty
        return (top == -1) ? null : arr[top];
    }
    public Integer pop() {
        Integer ret = peek();
        if(ret != null) {
            arr[top] = null;
            top--;
            nextOpen--;
        }
        return ret;
    }
```

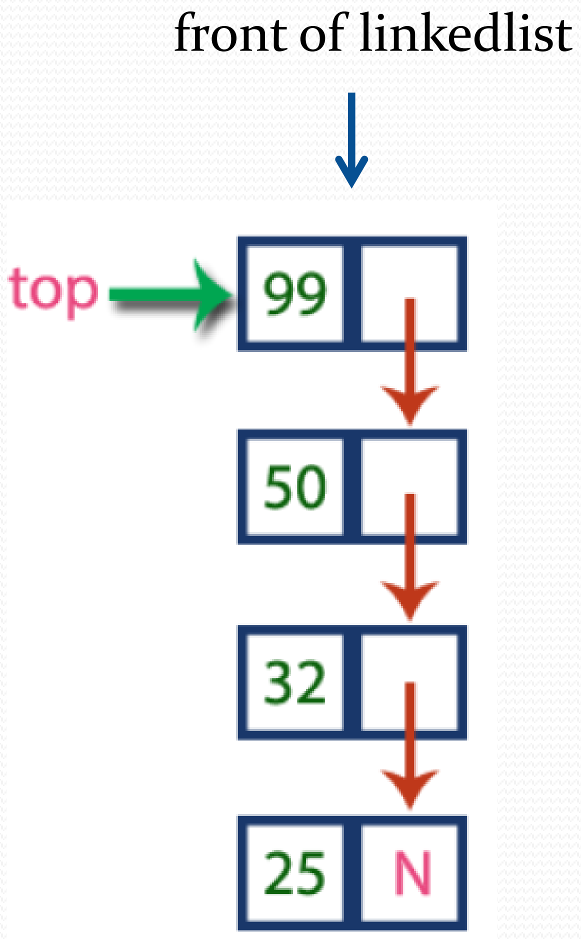
Implementation of STACK Using a LinkedList

- The usual `addFirst` operation in a Java `LinkedList` adds the new element to the front of the list. Therefore, an object `S` can be pushed onto a Java LinkedList `linked` with the call
  `linked.addFirst(S)`

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

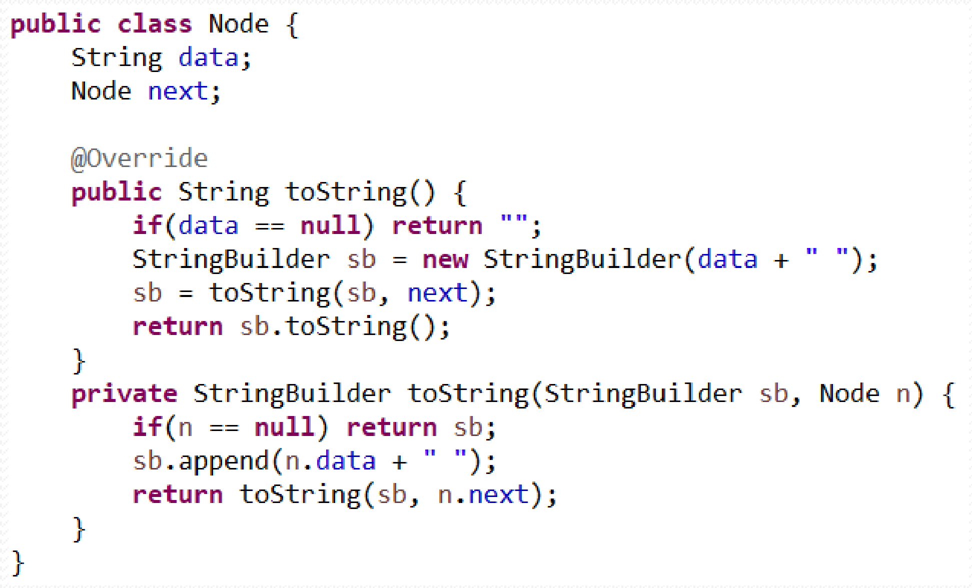
- The pop operation is equivalent to `remove(0)` (which returns the element removed).

- *Note:* This `LinkedList` implementation is essentially the same as the `Node`-based implementation, but hides the underlying `Node` operations behind the `LinkedList` API



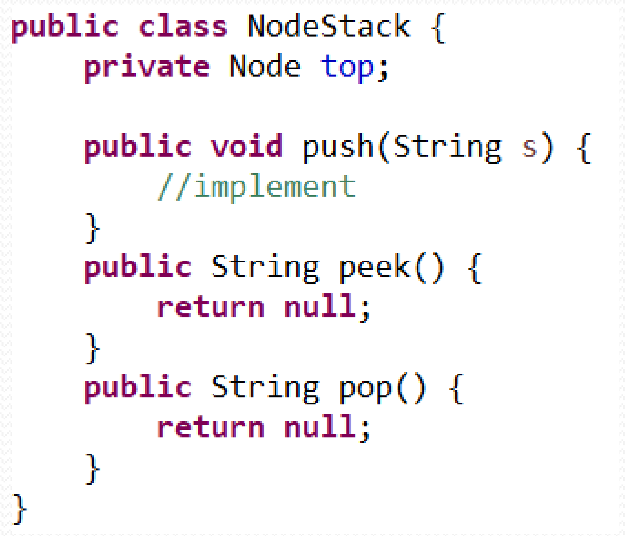
Exercise 9.1: Implementing a Stack with a Node

Implement the stack operations in NodeStack (see the startup code in InClassExercises project).



```java
public class Node {
    String data;
    Node next;

    @Override
    public String toString() {
        if(data == null) return "";
        StringBuilder sb = new StringBuilder(data + " ");
        sb = toString(sb, next);
        return sb.toString();
    }
    private StringBuilder toString(StringBuilder sb, Node n) {
        if(n == null) return sb;
        sb.append(n.data + " ");
        return toString(sb, n.next);
    }
}
```



```java
public class NodeStack {
    private Node top;

    public void push(String s) {
        //implement
    }
    public String peek() {
        return null;
    }
    public String pop() {
        return null;
    }
}
```

Exercise 9.1 - Solution

```java
public class Node {
    String data;
    Node next;

    @Override
    public String toString() {
        if(data == null) return "";
        StringBuilder sb = new StringBuilder(data + " ");
        sb = toString(sb, next);
        return sb.toString();
    }
    private StringBuilder toString(StringBuilder sb, Node n) {
        if(n == null) return sb;
        sb.append(n.data + " ");
        return toString(sb, n.next);
    }
}
```

```java
public class NodeStack {
    private Node top;
    public void push(String s) {
        Node newTop = new Node();
        newTop.data = s;
        newTop.next = top;
        top = newTop;
    }
    public String peek() {
        if(top != null) {
            return top.data;
        }
        else {
            return null;
        }
    }
    public String pop() {
        if(top != null) {
            String s = peek();
            top = top.next;
            return s;

        } else {
            return null;
        }
    }
}
```

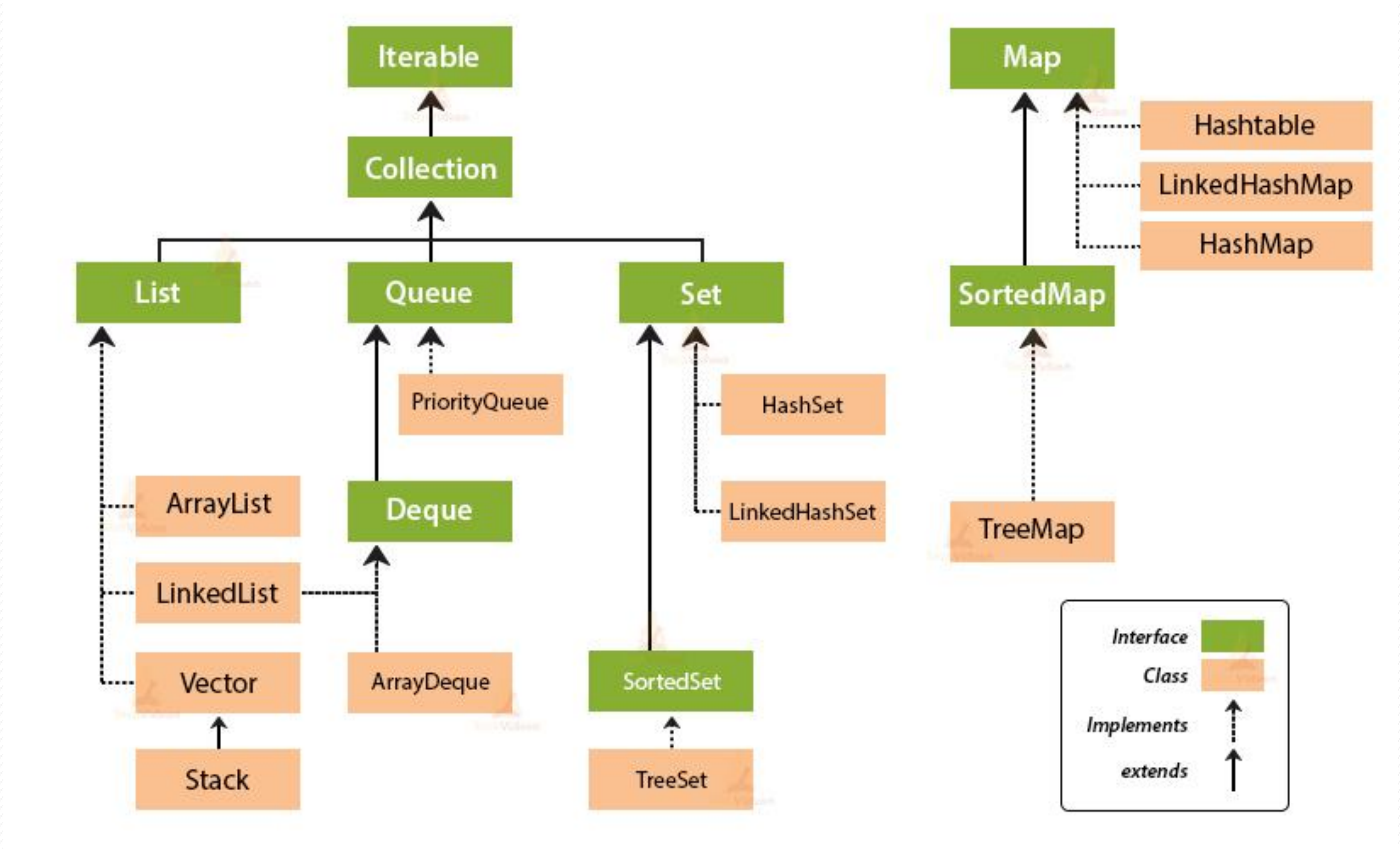
Java's Implementation of Stack

- The Java distribution comes with a `Stack` class, which is a subclass of `Vector`.(See next slide)

- `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 or on Nodes should be used instead.

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

Collection Hierarchy



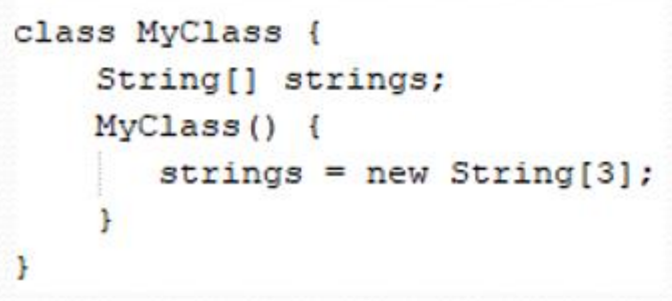
Application of Stacks: Symbol Balancing

- 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.
- This type of check is part of what the Java compiler does when it scans Java code.
- Simple Example:

  {a, b, f(c)} – balanced

  {a, b, f(c}) – not balanced

Another Example:



```
class MyClass {
    String[] strings;
    MyClass() {
        strings = new String[3];
    }
}
```

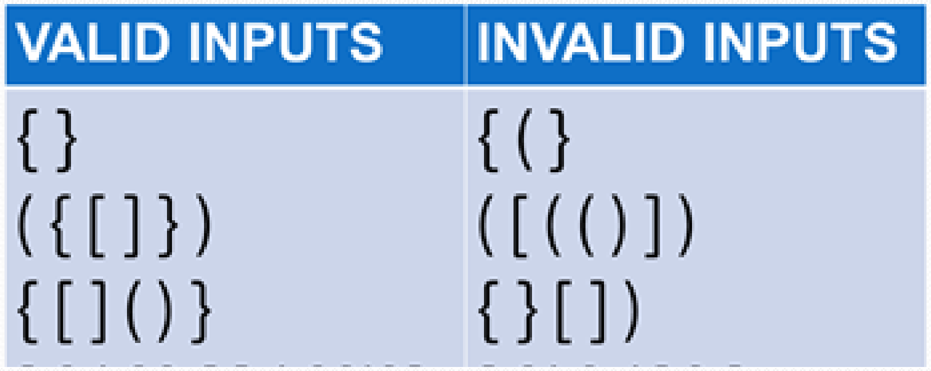
Removing non-bracketing characters produces:

{ [ ] ( ) { [ ] } }

This string of symbols is "balanced".

Symbol Balancing (cont)

- Here are some valid and some invalid strings of brackets (with all text removed)

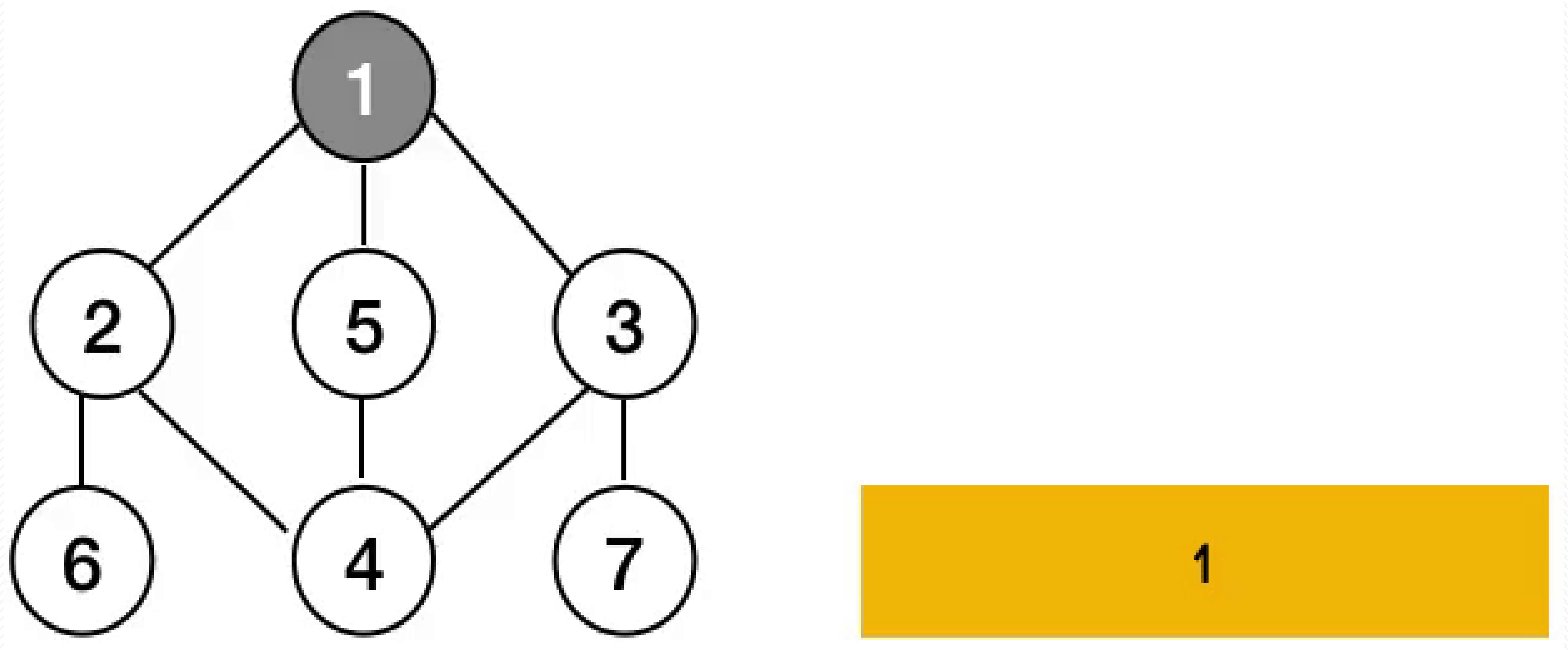


VALID INPUTS	INVALID INPUTS
{ }	{ (}
({ [ ] })	([(( )])
{ [ ] ( ) }	{ } [ ])

Symbol Balancing Algorithm

- The following procedure can be used:
  - Begin with an empty Stack
  - Scan the text (will ignore all non-bracketing symbols)
  - When an open symbol (like '(' or '[') is read, push it
  - When a closed symbol (like ')' or ']') is read, pop the Stack –
    1. if the stack is empty (so it can't be popped) return false.
    2. 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.
- See `Symbol Balancer Demo.pdf`

Application of Queues: Depth First Search



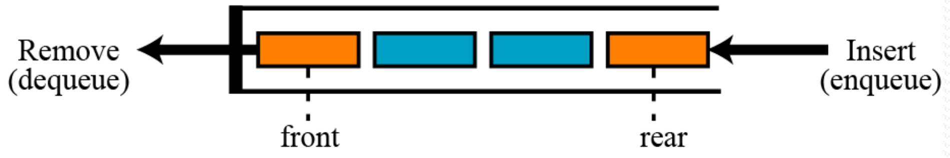
Depth First Search Algorithm

```
boolean DFS(int root, int target) {
    Set<Node> visited;
    Stack<Node> stack;
    add root to stack;
    while (s is not empty) {
        Node cur = the top element in stack;
        remove the cur from the stack;
        return true if cur is target;
        for (Node next : the neighbors of cur) {
            if (next is not in visited) {
                add next to visited;
                add next to stack;
            }
        }
    }
    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* or *rear*) and deletions may occur only at a designated position (the *front*).

A queue of people



A computer queue

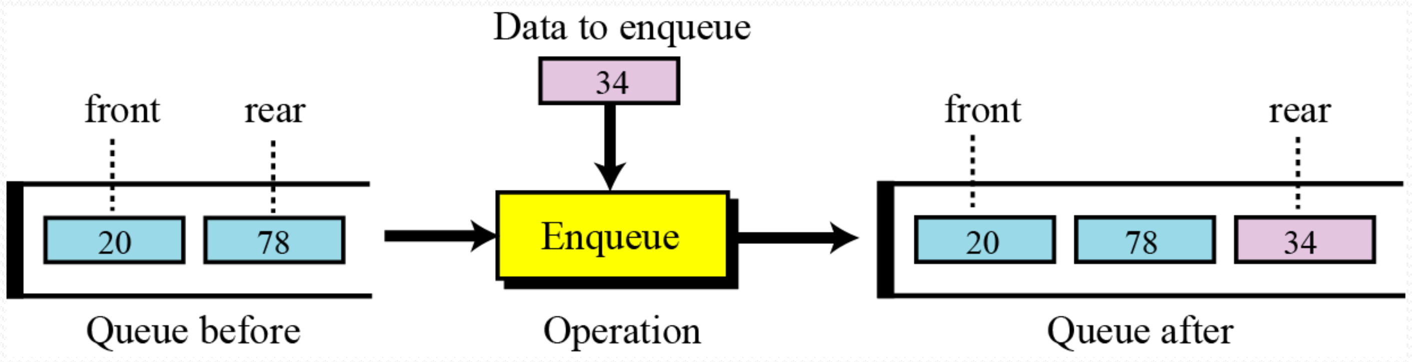
The Queue ADT

- **Operations**:

dequeue	remove and return the element at the front
enqueue	insert object at the back
peek	view object at front of queue without removing it

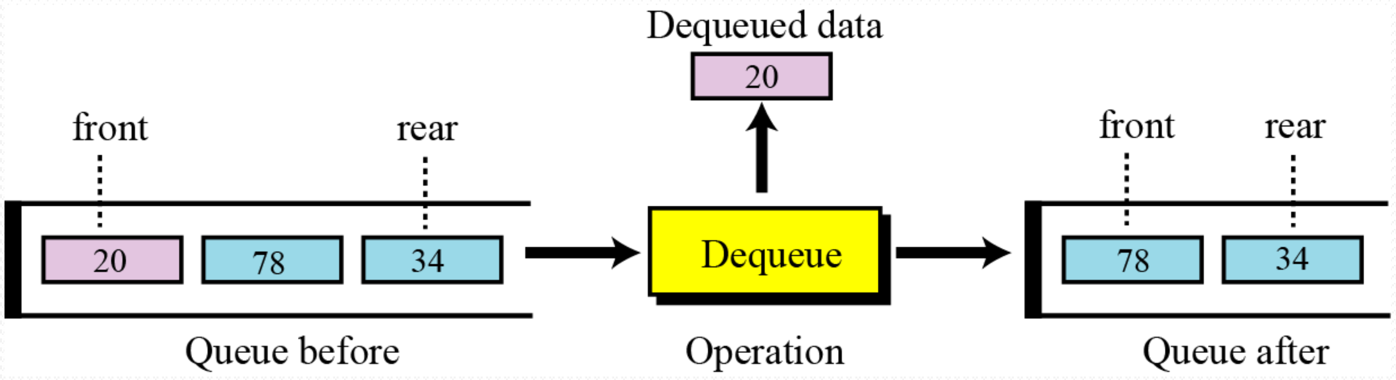
The QUEUE ADT

- **enqueue operation:**

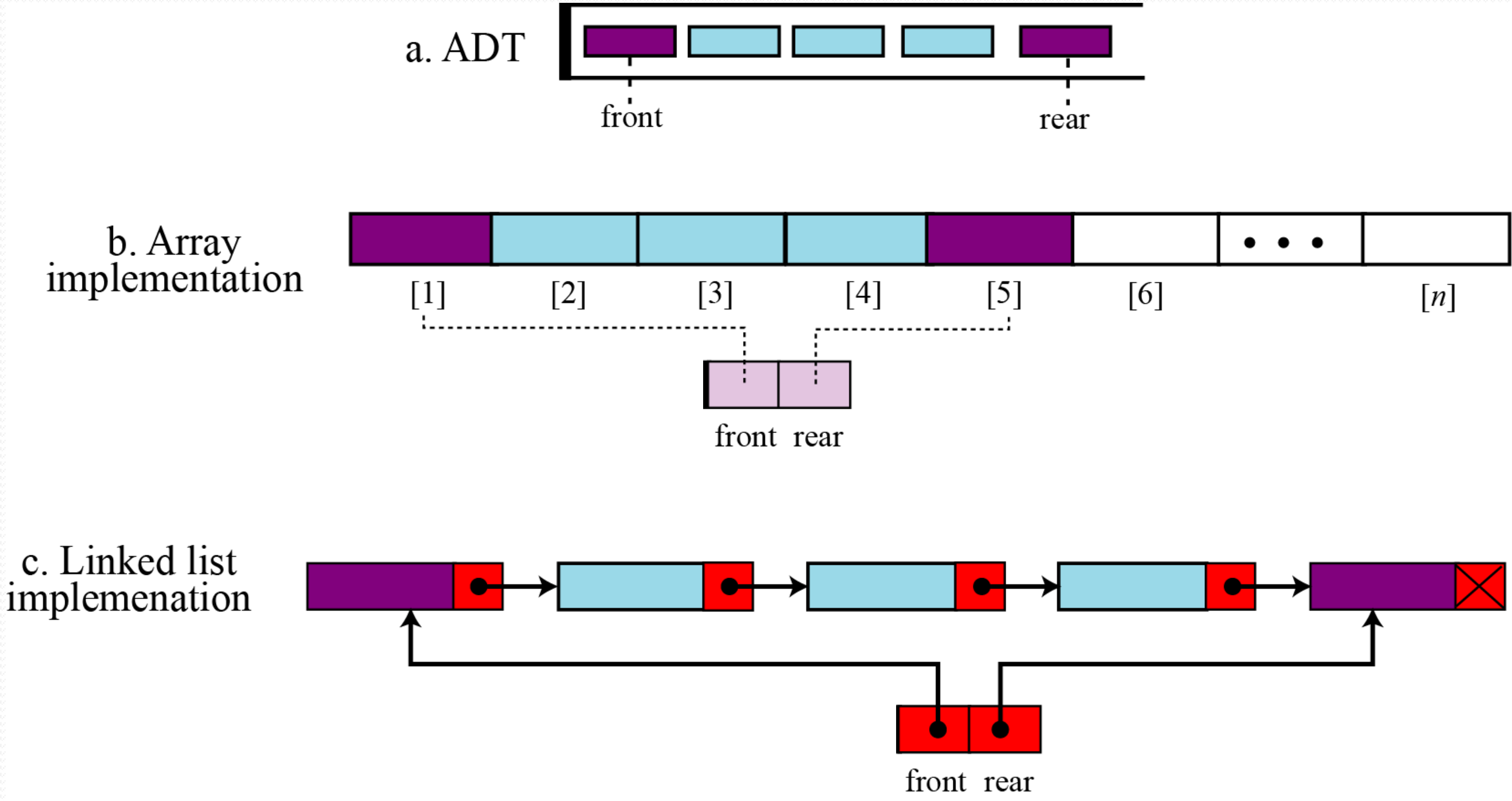


The QUEUE ADT

- **dequeue operation:**



Implementations of QUEUES

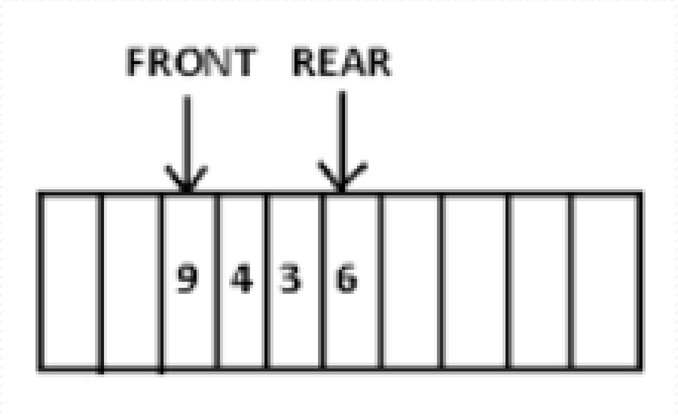


Implementations of QUEUES

- **Using a Linked List**
  - The enqueue operation is equivalent to adding each element to the end of a LinkedList.
  - The dequeue operation is equivalent to removing the element at the front of a LinkedList.
  - It is possible to implement a queue using Nodes, as we did with stack. See the labs.

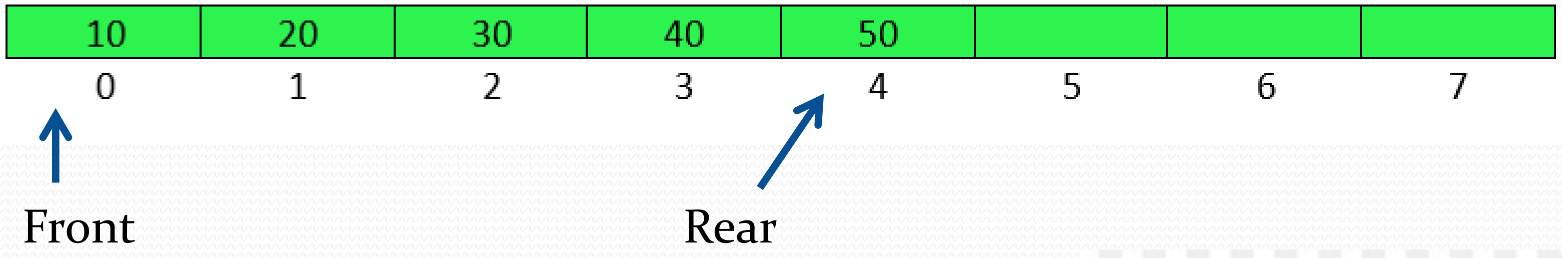
Implementations of QUEUES

- **Using an Array**
  - Need to maintain pointers to front and back elements

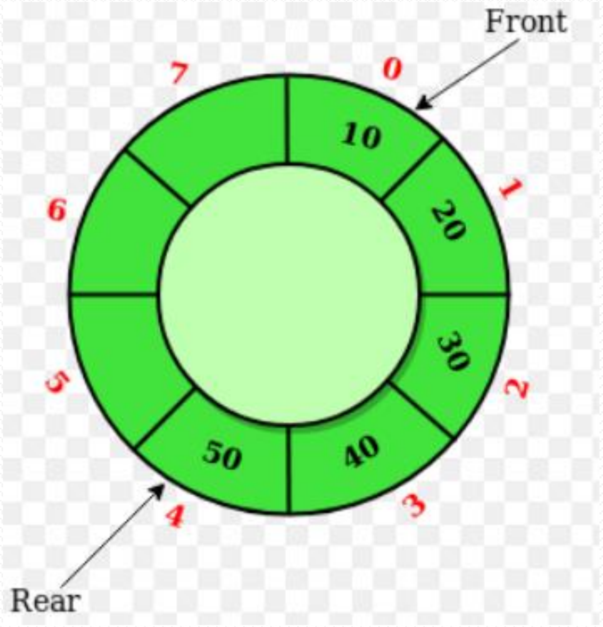


  - Repeated enqueuing will fill the right half of the array prematurely—solution is a *circular queue*.

Circular Queue



Linear queue as a circular queue



Java's Implementation

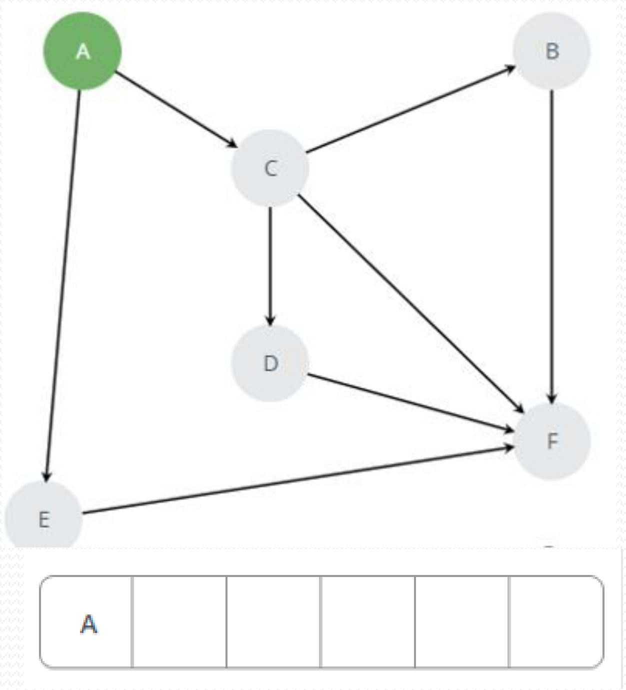
In j2se5.0, an interface `Queue<E>` (implemented by `LinkedList<E>`) is provided, with these declared operations:

- `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)

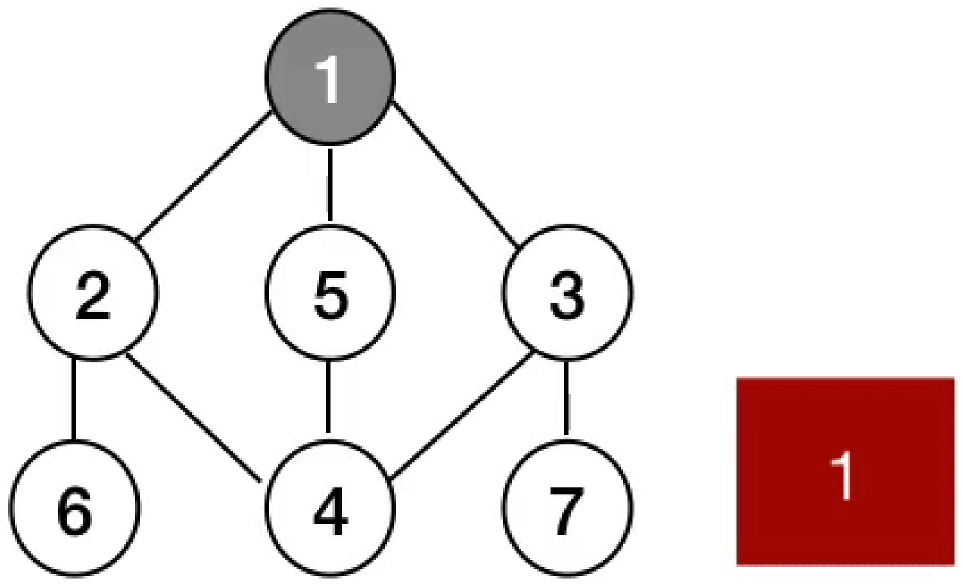
Application of Queues: Breadth First Search

Breadth First Search is a strategy for visiting every vertex in a graph.
https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/visualize/

Idea. Pick a starting vertex. Visit every adjacent vertex. Then take each of those vertices in turn and visit every one of its adjacent vertices. And so forth. Use a Queue to keep track of recently visited vertices.

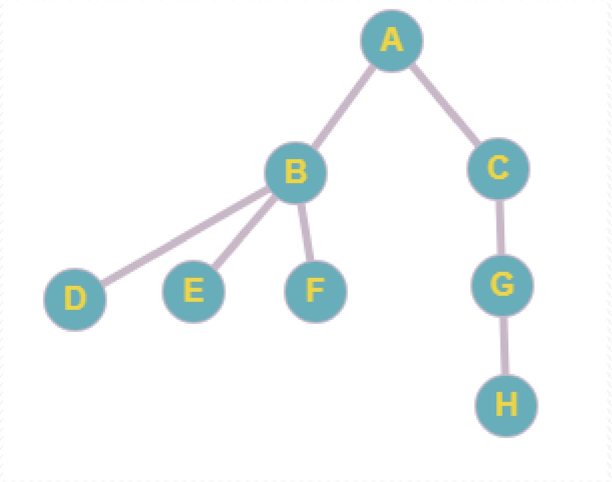


Application of Queues: Breadth First Search



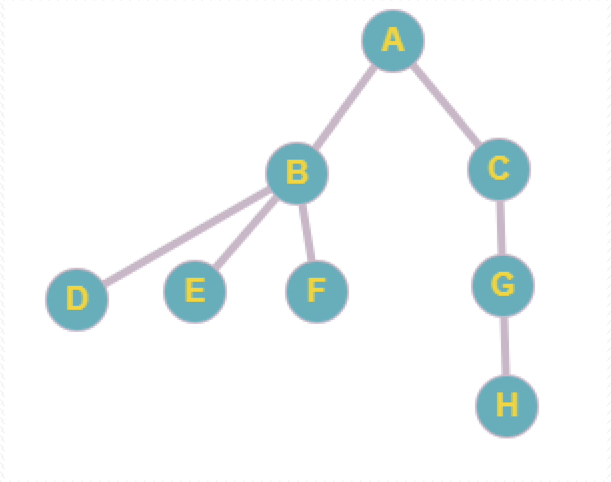
Exercise 9.2

Without using a queue, write down the vertices in the order in which the BFS algorithm will visit them, starting from vertex A, and choosing adjacent vertices in alphabetical order



Exercise 9.2 - Solution

Without using a queue, write down the vertices in the order in which the BFS algorithm will visit them, starting from vertex A, and choosing adjacent vertices in alphabetical order



A, B, C, D, E, F, G, H

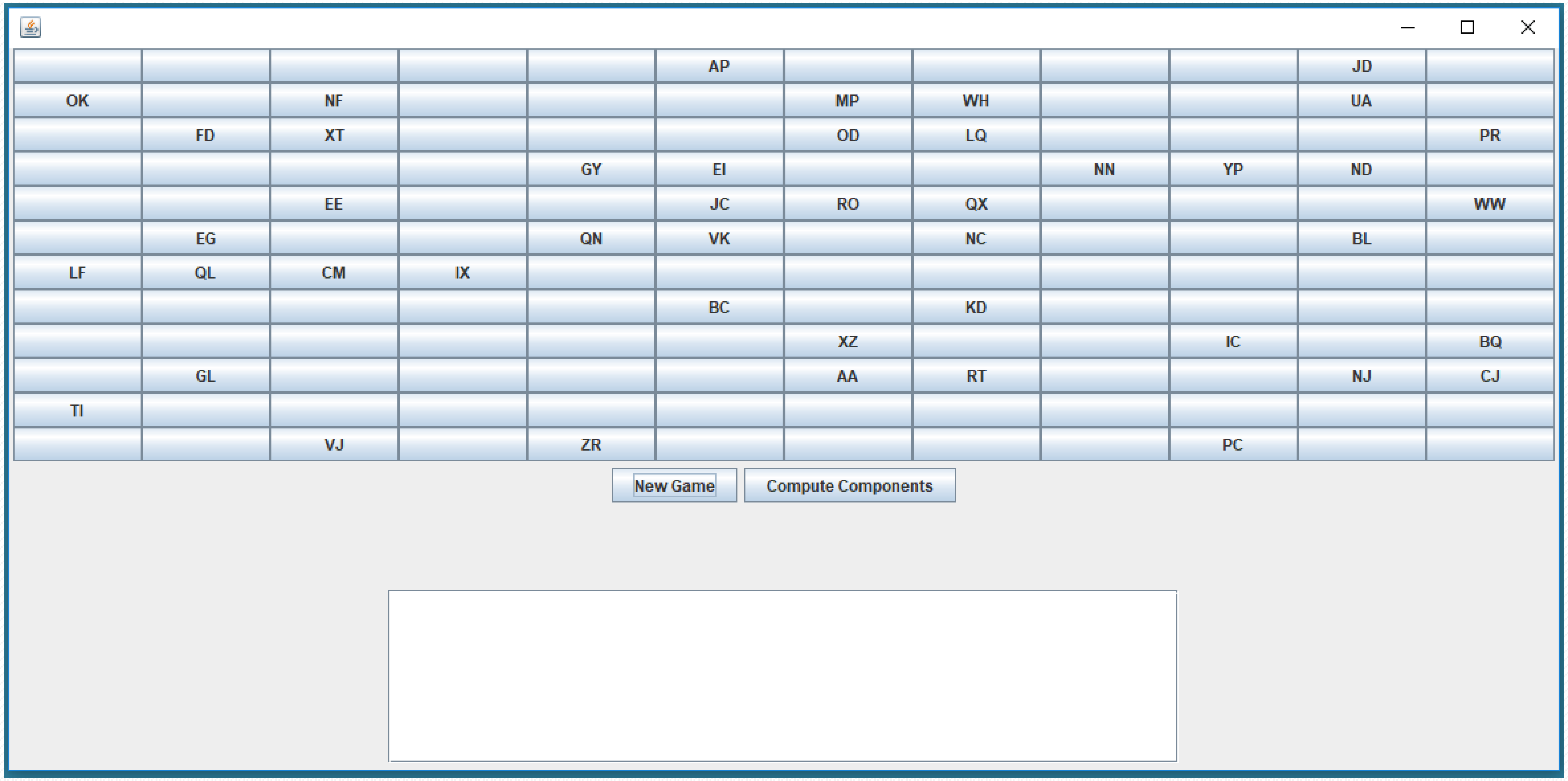
Breadth First Search Algorithm

```
int BFS(Node root, Node target) {

    Queue<Node> queue;  // store all nodes which are waiting to be processed
    int step = 0;       // number of steps neeeded from root to current node
    // initialize
    add root to queue;

    // BFS
    while (queue is not empty) {
        step = step + 1;
        // iterate the nodes which are already in the queue
        int size = queue.size();
        for (int i = 0; i < size; ++i) {
            Node cur = the first node in queue;
            return step if cur is target;
            for (Node next : the neighbors of cur) {
                add next to queue;
            }
            remove the first node from queue;
        }
    }
    return -1;          // there is no path from root to target
}
```

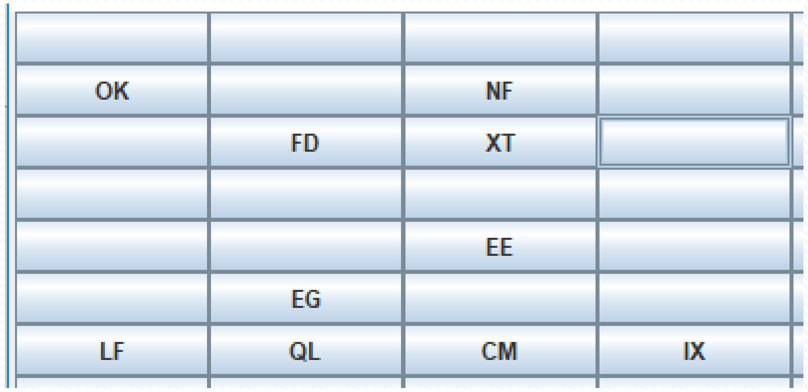
The Queue Game



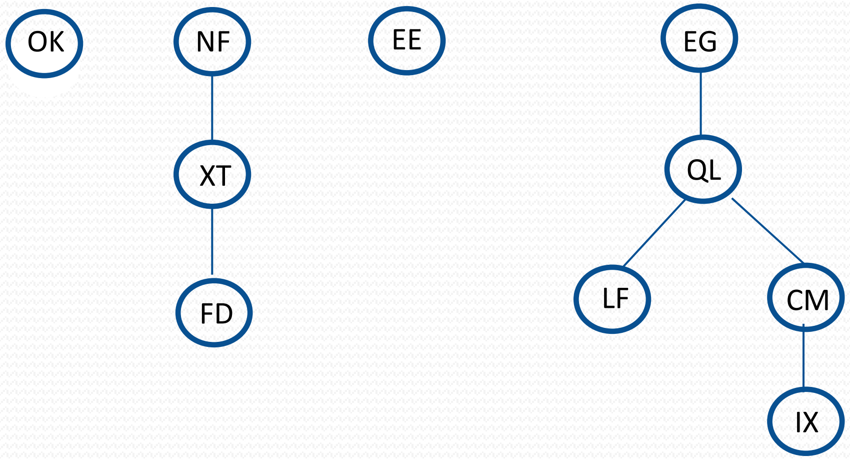
					AP					JD	
OK		NF				MP	WH			UA	
	FD	XT				OD	LQ				PR
				GY	EI			NN	YP	ND	
		EE			JC	RO	QX				WW
	EG			QN	VK		NC			BL	
LF	QL	CM	IX								
					BC		KD				
						XZ			IC		BQ
	GL					AA	RT			NJ	CJ
TI											
		VJ		ZR					PC		

(see queueGame.jar)

Solving the Queue game with BFS



OK		NF	
	FD	XT	
		EE	
	EG		
LF	QL	CM	IX



1. Represent the board with its occupied cells as a graph. An occupied cell is a vertex in the graph. Adjacent occupied cells will be treated as adjacent vertices.
2. Start with some occupied cell, viewed as a vertex, and peform BFS. When finished, the first component has been discovered.
3. Move to an unvisited occupied cell (viewed as a vertex again). Perform BFS. When finished, the second component has been discovered.
4. Continue till no unvisited vertices remain.

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 Maharishi Vedic Science 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.