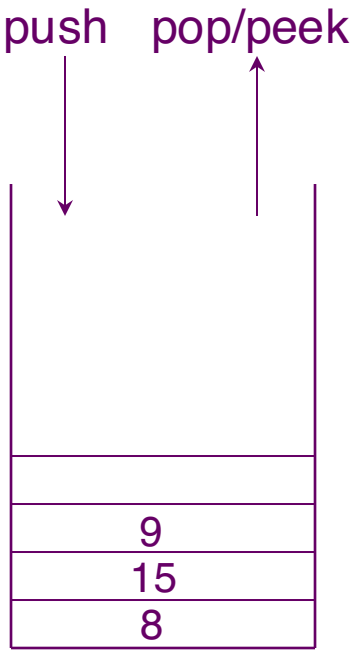
Stacks and Queues

EECS 233

Stack ADT

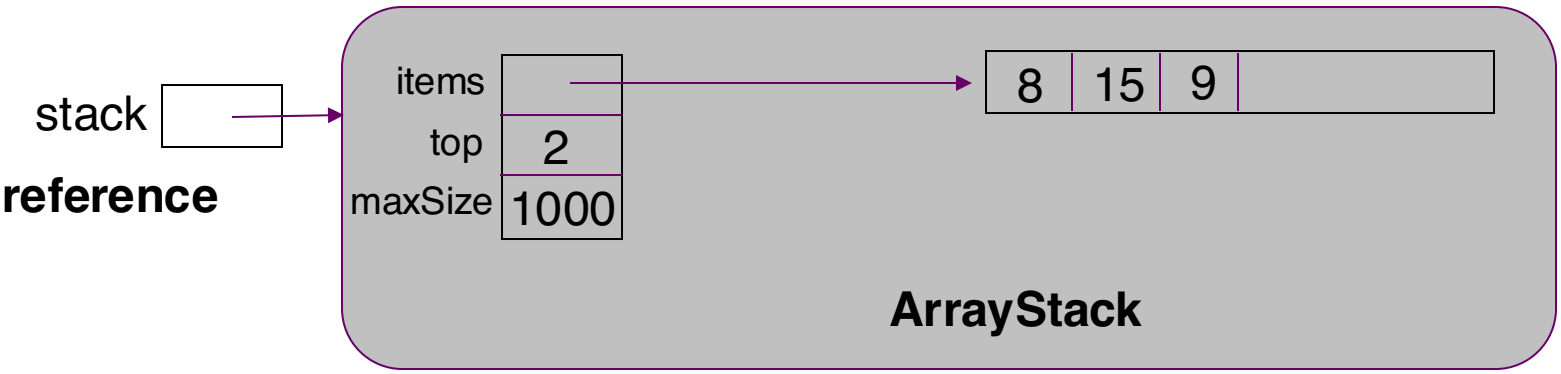


- A stack is a special sequence in which:
  - items can be added and removed only at one end (the *top*)
  - you can only access the item that is currently at the top
- Operations:
  - boolean push(ItemType i); add an item to the top of the stack
  - ItemType pop(); remove the item at the top of the stack
  - ItemType peek(); get the item at the top of the stack, but don't remove it
  - boolean isEmpty();
  - boolean isFull();
- The interface provides no way to access/insert/delete an item at an arbitrary position.
  - Enforced by encapsulation

push(8); push(15); push(9); pop() - returns 9

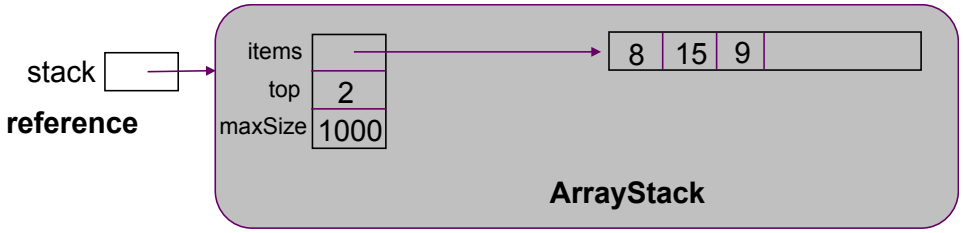
Array Implementation of Stacks

- Example: the integer stack | 9 / 15 / 8 | would be represented as follows:



Array Implementation: Constructors and Methods

```
public ArrayStack(int max) {
      items = new int[max];        // a stack of integers
      top = -1;
      maxSize = max;
}

public boolean isEmpty() {
      return (top == -1);
}

public boolean isFull() {
      return (top == maxSize – 1);
}
```



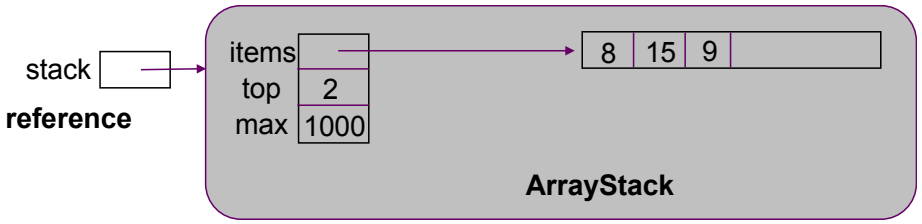
Generic ArrayStack Class

- Generic classes use *type variables* that serve as placeholders for actual types.

```
public class ArrayStack<T> {
        private T[ ] items;
        private int top;
        private int maxSize;
        …
        public boolean push(T item) {
        …
}
```

- Constructor:

```
public ArrayStack(int max) {
        items = (T[ ]) new Object[max];
        top = -1;
        maxSize = max;
}
```



- Methods: push(), pop(), peek()

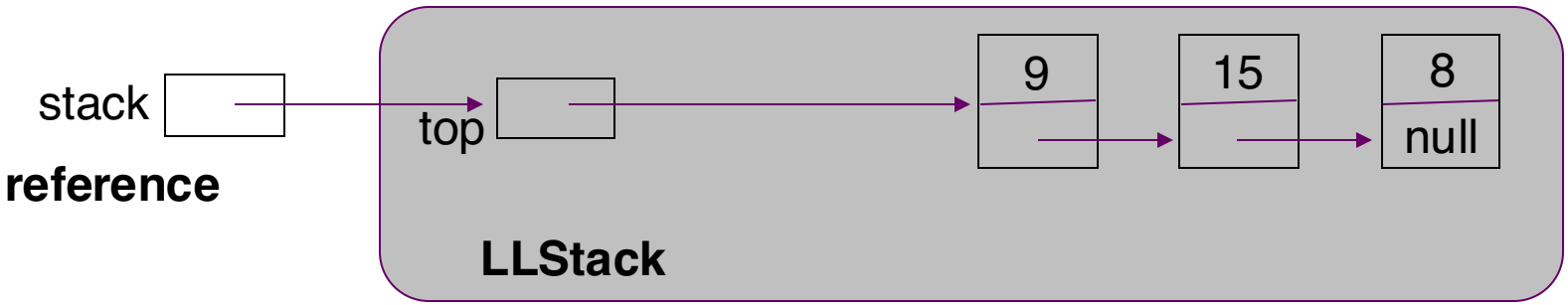
```
public boolean push(T item) {
        if (isFull())
                return false;
        items[++top] = item;
        return true;
}

public T pop() {
        if (isEmpty())
            throw new
                RuntimeException("Removing
                    from empty");
        return items[top--];
}

public T peek() {
        if (isEmpty())
            throw new
                RuntimeException("Stack is
                        empty");
        return items[top];
}
```

Usage:

ArrayStack<Bag> bagStack = new ArrayStack<Bag>(100);

Linked-List Implementation of Stacks

- The integer stack | 9 / 15 / 8 | in linked list:



Generic LLStack Class

```
public class LLStack<T> {
    private class Node {
        T item;
        Node next;
    }

    private Node top;
    ...

    public boolean push(T item) {
        ...
    }
}
```

Applications: Checking for Delimiter Balancing

- Making sure delimiters (e.g., parentheses, brackets) are balanced:
  - push open (i.e., left) delimiters onto a stack
  - when you encounter a close (i.e., right) delimiter, pop an item off the stack and see if it matches
  - example: 5 * [3 + {(5 + 16 – 2)]
    - push “[“; push “{“; push “(“;
    - pop “(“ when seeing “)”
    - pop “{“ when seeing “]” ???

Queue ADT

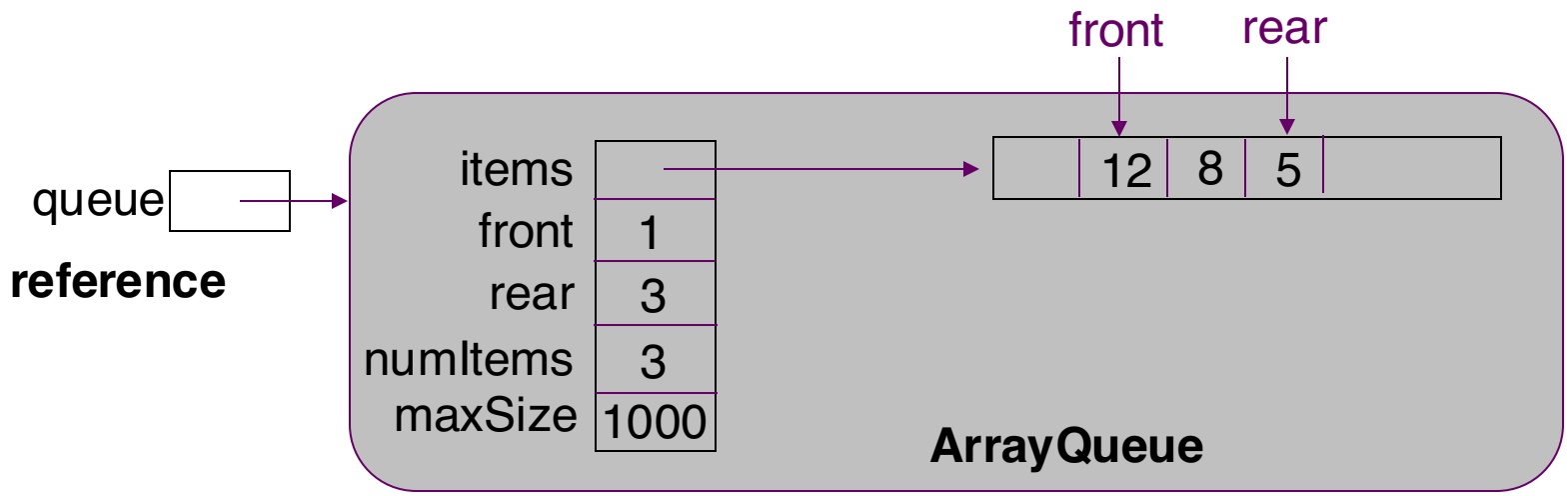
- A queue is a special sequence in which:
  - items are added at the rear and removed from the front
    - first in, first out (FIFO) (vs. a stack, which is last in, first out)
  - we can only access the item that is currently at the front
- Operations:
  - boolean insert(T item); add an item at the rear of the queue
  - T remove(); remove the item at the front of the queue
  - T peek(); get the item at the front of the queue, but don't remove it
  - boolean isEmpty(); test if the queue is empty
  - boolean isFull(); test if the queue is full

- Example: a queue of integers
  - *Starting state:* 12 8
  - *insert 5:* 12 8 5
  - *remove: get* 12, *state* 8 5

Array Implementation of Generic Queues

- Five instance variables:

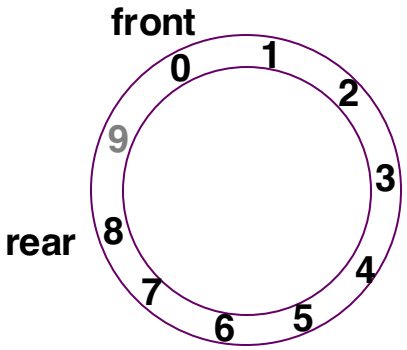
```
T[ ] items; // array of type T (type variable)
int front; // index of item at front of queue
int rear; // index of item at rear of queue
int numItems; // number of items in queue (optional for now)
int maxSize; // size of the array (optional - see array.length)
```



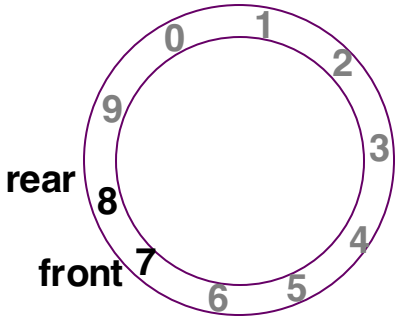
Problem: what do we do when we reach the end of the array?

Circular Array Implementation

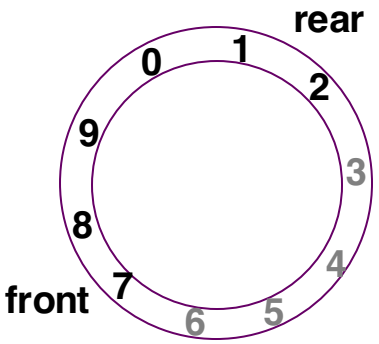
- Problem: what do we do when we reach the end of the array?
- Solution: a *circular array*.
  - When we reach the end of the array, we wrap around to the beginning.



After 9 insertons

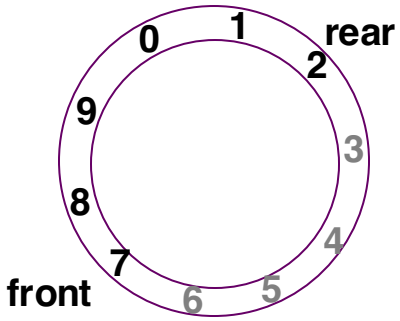


After 7 removals

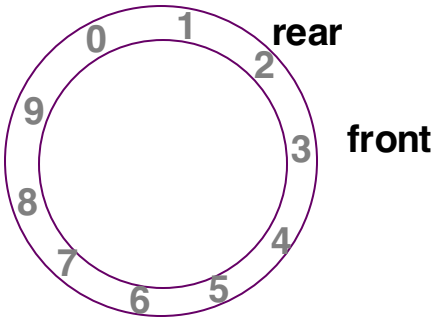


After 4 insertions

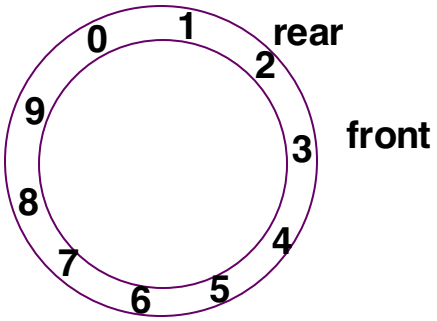
Circular Array: Distinguishing Full From Empty



Previous state



After 6 more removals: empty



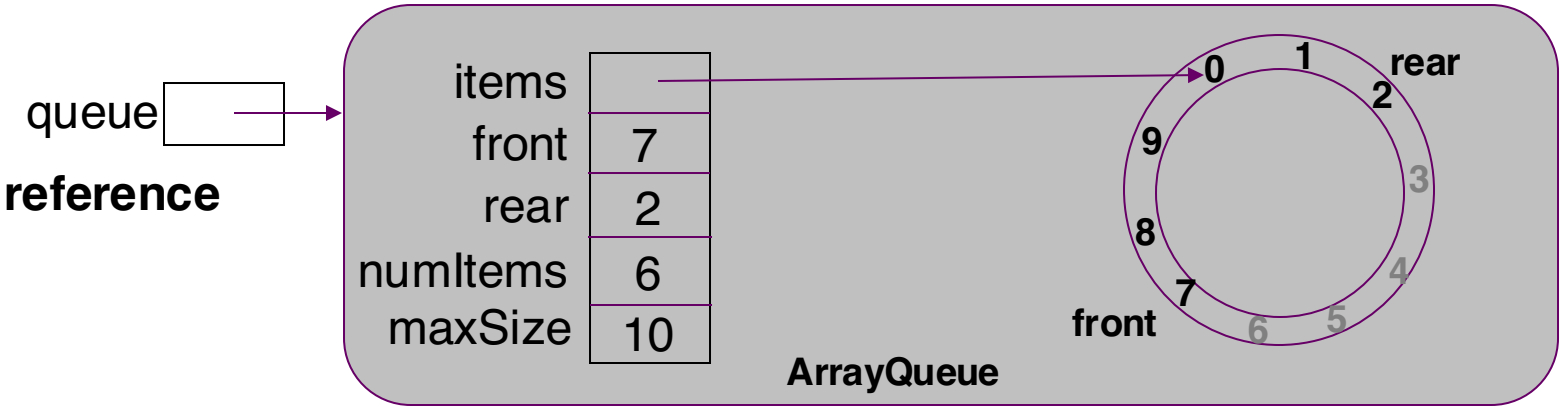
After 10 more insetions: full

- The queue is empty when front "overcomes" rear:
  - ((rear + 1) % maxSize) == front
- But how to distinguish from a full ArrayQueue?
  - we maintain numItems!
  - Test for (numItems == maxSize)

Constructors and Methods

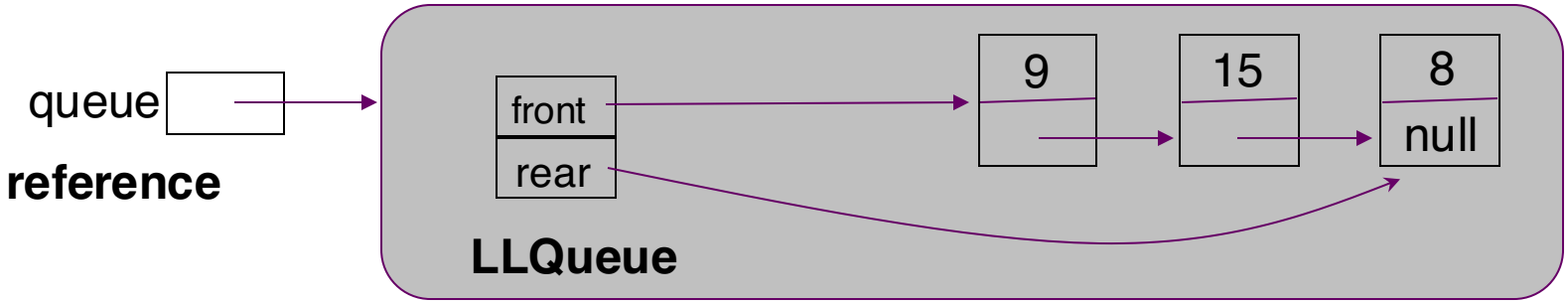
```
public ArrayQueue<T>(int max) {
        items = (T[ ]) new Object[max];
        maxSize = max;
        front = 0;
        rear = -1;
        numItems = 0;
}
```

```
public boolean insert(T item) {
        if (isFull())
                return false;
        rear = (rear + 1) % items.length;
        items[rear] = item;
        numItems++;
        return true;
}

public T remove() {
        if (isEmpty())
                throw new RuntimeException("Removing
                    from empty queue");
        T removed = items[front];
        front = (front + 1) % items.length;
        numItems--;
        return removed;
}
```



Linked-List Implementation of Queues

- Two instance variables:
  - Node front; // front of the queue
  - Node rear; // rear of the queue



- No capacity issue: no need for circular buffer.

Applications of Queues

- First-in first-out (FIFO) inventory control
- OS scheduling: processes, print jobs, packets, etc.
- Simulations of banks, supermarkets, airports, etc.
- Breadth-first traversal of a graph (stay tuned…)

Summary: Efficiency of Stacks and Queues

- Stack and Queue complexity
- Array and linked list implementation
  - Running time of insert (push), remove (pop), peek
  - Space complexity?
- Mind twister problem: emulate a queue using stacks

How to implement a queue using two stacks

- Let queue to be implemented be q and stacks used to implement q be stack1 and stack2
- Implement the enQueue and deQueue operations

Method 1 (costly enQueue operation)

Makes sure that oldest entered element is always at the bottom of stack 1
deQueue operation just pops from stack1
To put the element at the bottom of stack1, stack2 is used.

enQueue(q, x)
1) While stack1 is not empty, push everything from stack1 to stack2.
2) Push x to stack1 (assuming size of stacks is unlimited).
3) Push everything back to stack1.

dnQueue(q)
1) If stack1 is empty then error
2) Pop an item from stack1 and return it

How to implement a queue using two stacks

Method 2 (By making deQueue operation costly)

In enqueue operation, the new element is entered at the top of stack1

In dequeue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned

enQueue(q, x)

1) Push x to stack1 (assuming size of stacks is unlimited).

deQueue(q)

1) If both stacks are empty then error.
2) If stack2 is empty
   While stack1 is not empty, push everything from stack1 to stack2.
3) Pop the element from stack2 and return it.

Method 1 moves all the elements twice in enQueue operation

Method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty