Queues

CS284

Structure of this week's classes

Queues

Applications

Implementation

Queue

- The queue, like the stack, is a widely used data structure
- A queue differs from a stack in one important way
  - A stack is LIFO list – *Last-In, First-Out*
  - While a queue is FIFO list – *First-In, First-Out*

Example: Print Queue

- Operating systems use queues to
  - keep track of tasks waiting for a scarce resource
  - ensure tasks are carried out in the order they were generated
- Print queue: printing is much slower than the process of selecting pages to print, so a queue is used

The Queue Interface (Sample) – `java.util` (1/2)

```java
public interface Queue<E> extends Collection<E> {

// Returns entry at front of queue without removing it. If the
// queue is empty, throws NoSuchElementException
E element()

// Insert an item at the rear of a queue
boolean offer(E item)

// Adding an item at the rear of a queue
boolean add(E item)

// Return element at front of queue without removing it;
// returns null if queue empty
E peek()

// Remove and return  entry from front of queue;
// returns null if queue empty
E poll()

// Removes entry from front of queue and returns it if
// queue not empty; otherwise throws NoSuchElementException
E remove()
}
```

Difference between `add` and `offer`

- **`offer`**: tries to add an element to a queue, and returns false if the element can't be added (like in case when a queue is full), or true if the element was added, and doesn't throw any specific exception.
- **`add`**: tries to add an element to a queue, returns true if the element was added, or throws an IllegalStateException if no space is currently available.

The Queue Interface – `java.util` (2/2)

Note:

- `Stack<E>` is a class (derived from Vector) but `Queue<E>` is an interface (derived from Collection)
- Stacks have a canonical behaviour, Queues do not (eg. priority queues)

Queues

Applications

Implementation

Implementing a Stack using Queue(s)

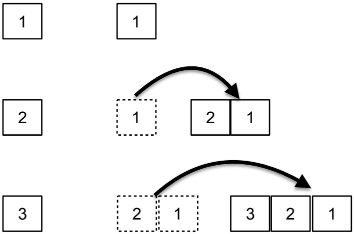
- Stack - LIFO; queue - FIFI;
- Operations we can use:
  - `poll();`
  - `add();`
  - `peek();`
  - `isempty();`
- Operations we need to implement
  - `push();`
  - `pop();`
  - `peek();`
  - `isempty();`

Analysis

- After `push` and `pop`, the output element needs to be the *last* element being pushed in;
- The `poll` operation can only remove the *first* element in the queue;
- Therefore, must place the last added element in the first position;
- We can implement this by `poll`ing and `add`ing all the previous elements;

push

```java
    /** Push element x onto stack. */
    public void push(int x) {
        queue.add(x);
        for(int i = 0; i < queue.size() - 1; i++){
            queue.add(queue.poll());
        }
    }
```



pop, peek, isempty

```java
/** Removes the element on top of the stack and returns
that element. */
public int pop() {
    return queue.poll();
}

/** Get the top element. */
public int top() {
    return queue.peek();
}

/** Returns whether the stack is empty. */
public boolean empty() {
    return queue.isEmpty();
}
```

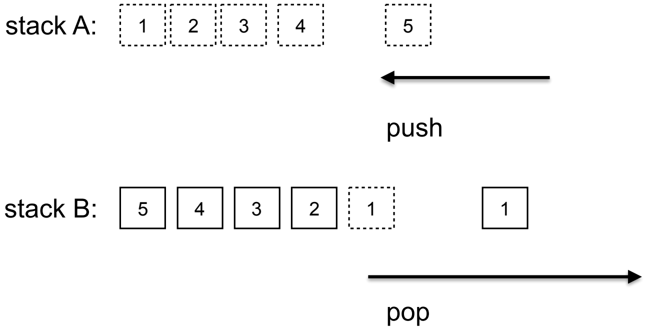
Implementing Queue using Two Stacks

- Stack - LIFO; queue - FIFI;
- Operations we can use:
  - `push();`
  - `pop();`
  - `peek();`
  - `isempty();`
- Operations we need to implement:
  - `poll();`
  - `add();`
  - `peek();`
  - `isempty();`

Analysis

- One stack reverses the order, two stacks maintain the same order;
- Stack for pushing: always push to stack A;
- Stack for popping: pop and push stack A's element to stack B, pop from stack B;

Analysis



Protocol

- Push: Always push to stack A;
- Pop: Always pop from stack B. If stack B is empty, dump *all* stack A's element to stack B;

poll

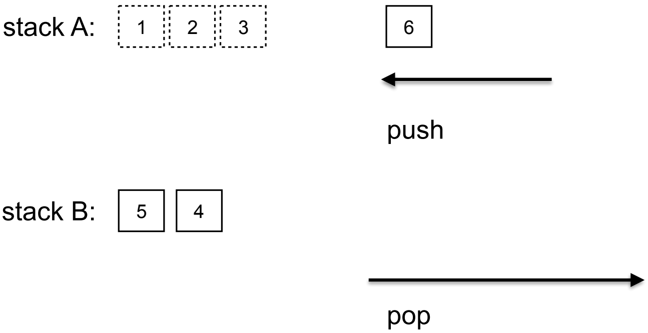
```java
/** adding/pushing to stack A */
public void add(int element) {
    stack1.push(element);
}

/** popping from/poll from top of stack B
 * when stack B is empty, always dump *all*
 * elements from stack A to stack B
 * @return
 */
public int poll() {
    if(stack2.empty()){
        while(!stack1.empty()){
            stack2.push(stack1.pop());
        }
    }
    return stack2.pop();
}
```

peek

```java
/** getting the first element from stack B
 * when stack B is empty, always dump *all*
 * elements from stack A to stack B
 * @return
 */
public int peek() {
    if(stack2.empty()){
        while(!stack1.empty()){
            stack2.push(stack1.pop());
        }
    }
    return stack2.peek();
}
```

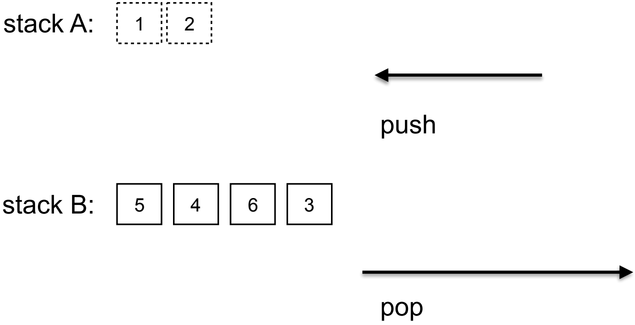
Discussion: Why Dumping *all* Elements from Stack A?

Is it *necessary* we dump all the elements?



Discussion: Why Dumping *all* Elements from Stack A?

Is it *necessary* we dump all the elements?



Queues

Applications

Implementation

Class `LinkedList` Implements the Queue Interface

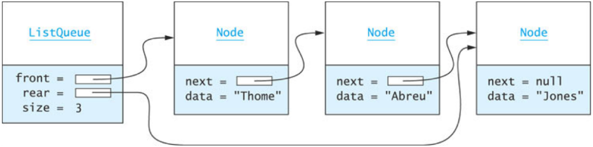
- The `LinkedList` class provides methods for inserting and removing elements at either end of a double-linked list, which means all Queue methods can be implemented easily
- The Java 5.0 LinkedList class implements the Queue interface

```
Queue<String> names = new LinkedList<String>();
```

  - creates a new Queue reference, names, that stores references to String objects

Using a Single-Linked List to Implement a Queue

- Insertions are at the rear of a queue and removals are from the front
- We need a reference to the last list node so that insertions can be performed at $\mathcal{O}(1)$
- The number of elements in the queue is changed by methods insert and remove



Using a Single-Linked List to Implement a Queue

- A comment before beginning
- One might expect to start out with something like:

```
public class ListQueue<E> implements Queue<E> {
   ...
}
```

- However, since `Queue` is a subinterface (i.e., parent interface) of other interfaces (namely, `Collection<E>` and `Iterable<E>`), many additional operations would have to be implemented

Using a Single-Linked List to Implement a Queue

- It is best to start off with the abstract class `AbstractQueue` since it implements all operations except for:
  - public boolean offer(E item)
  - public E poll()
  - public E peek()
  - public int size()
  - public Iterator<E> iterator()
- Our implementation shall concentrate on these

```java
public class ListQueue<E> extends AbstractQueue<E>
    implements Queue<E> {
   ...
}
```

Using a Single-Linked List to Implement a Queue

```java
import java.util.*;
public class ListQueue<E> extends AbstractQueue<E>
    implements Queue<E> {

  // Data Fields
  /** Reference to front of queue. */
  private Node<E> front;
  /** Reference to rear of queue. */
  private Node<E> rear;
  /** Size of queue. */
  private int size;
```

Using a Single-Linked List to Implement a Queue

```
/** Node is building block for single-linked list. */
private static class Node<E> {
  private E data;
  private Node next;

  /** Creates a new node with a null next field.
      @param dataItem The data stored
   */
  private Node(E dataItem) {
    data = dataItem;
    next = null;
  }
  /** Creates a new node that references another node.
      @param dataItem The data stored
      @param nodeRef The node referenced by new node
   */
  private Node(E dataItem, Node<E> nodeRef) {
    data = dataItem;
    next = nodeRef;
  }
} //end class Node
```

Using a Single-Linked List to Implement a Queue

```java
/** Insert an item at the rear of the queue.
    post: item is added to the rear of the queue.
    @param item The element to add
    @return true (always successful)    */
public boolean offer(E item) {
  // Check for empty queue.
  if (front == null) {
    rear = new Node<E> (item);
    front = rear;
  }
  else {
```

Using a Single-Linked List to Implement a Queue

```java
    else {
      // Allocate a new node at end, store item in
      // it, and
      // link it to old end of queue.
      rear.next = new Node<E>(item);
      rear = rear.next;
    }
    size++;
    return true;
  }
```

Using a Single-Linked List to Implement a Queue

```
  /** Return the item at the front of the queue without removi
      @return The item at the front of the queue if successful
   */
  public E peek() {
    if (size == 0)
      return null;
    else
      return front.data;
  }
}
```

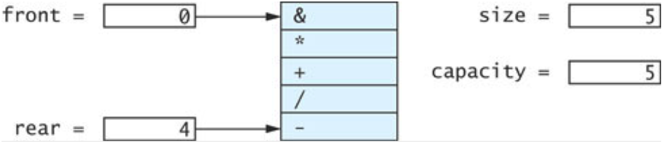
Using a Single-Linked List to Implement a Queue

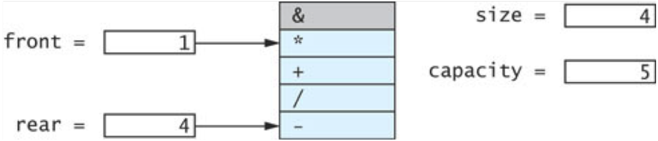
```java
/** Remove the entry at the front of the queue and
     return it if the queue is not empty.
     post: front references item that was 2nd in queue.
     @return Item removed if successful, null othw    */
 public E poll() {
   E item = peek(); // Retrieve item at front.
   if (item == null)
     return null;
   if (size==1) {   // Queue has one item
       front = null;
       rear  = null;
   } else {         // Queue has two or more items
       front = front.next;
   }
   size--;
   return item; // Return data at front of queue.
 }
```

Implementing a Queue Using a Circular Array

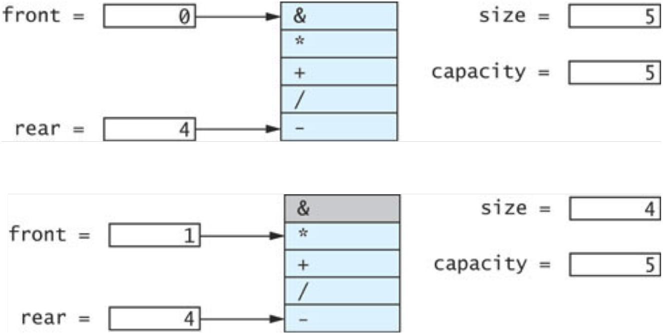
- The time efficiency of using a single- or double-linked list to implement a queue is acceptable
- However, there are some space inefficiencies
- Storage space is increased when using a linked list due to references stored in the nodes
- Array Implementation
  - Insertion at rear of array is constant time $\mathcal{O}(1)$
  - Removal from the front is linear time $\mathcal{O}(n)$ if we shift all elements
  - Removal from rear of array is constant time $\mathcal{O}(1)$
  - Insertion at the front is linear time $\mathcal{O}(n)$ if we shift all elements
- We can avoid these inefficiencies in a circular array

Implementing a Queue Using a Circular Array (cont.)





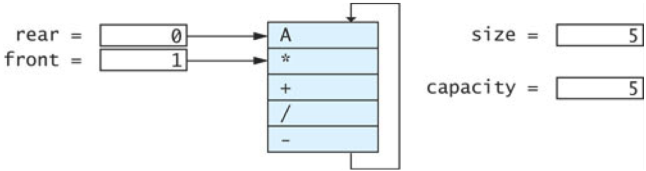
Implementing a Queue Using a Circular Array (cont.)



Now we add A

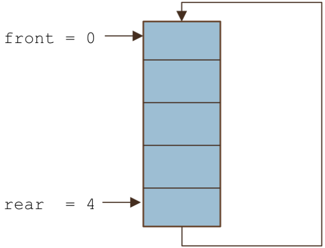
Implementing a Queue Using a Circular Array (cont.)

We add A



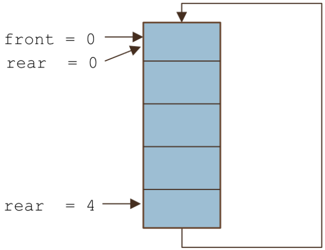
Implementing a Queue Using a Circular Array (cont.)

```
ArrayQueue q = new ArrayQueue(5);
```




```
public ArrayQueue(int initCapacity) {
  capacity = initCapacity;
  theData = (E[])new Object[capacity];
  front = 0;
  rear = capacity - 1;
  size = 0;
}
```

Implementing a Queue Using a Circular Array (cont.)

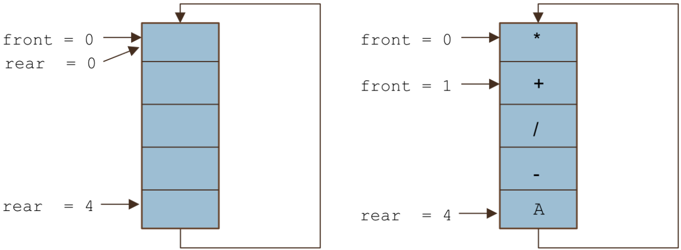



```java
public boolean offer(E item) {
  if (size == capacity) {
    reallocate();
  }
  size++;
  rear = (rear + 1) % capacity;
  theData[rear] = item;
  return true;
}
```

Let's see an example

Implementing a Queue Using a Circular Array (cont.)

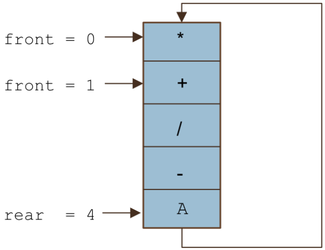
`q.offer('*');q.offer('+');q.offer('/');q.offer('-');q.offer('A');`




```
public boolean offer(E item) {
  if (size == capacity) {
    reallocate();
  }
  size++;
  rear = (rear + 1) % capacity;
  theData[rear] = item;
  return true;
}
```

Implementing a Queue Using a Circular Array (cont.)

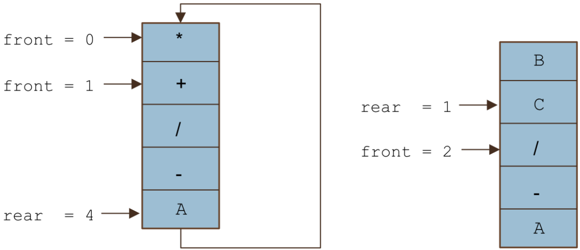
`next = q.poll();next = q.poll();`




```
public E poll() {
  if (size == 0) {
    return null
  }
  E result = theData[front];
  front = (front + 1) % capacity;
  size--;
  return result;
}
```

Implementing a Queue Using a Circular Array (cont.)

`q.offer('B');q.offer('C')`




```java
public boolean offer(E item) {
  if (size == capacity) {
    reallocate();
  }
  size++;
  rear = (rear + 1) % capacity;
  theData[rear] = item;
  return true;
}
```

Implementing a Queue Using a Circular Array (cont.)

```
private void reallocate() {
  int newCapacity = 2 * capacity;
  E[] newData = (E[])new Object[newCapacity];
  int j = front;
  for (int i = 0; i < size; i++) {
    newData[i] = theData[j];
    j = (j + 1) % capacity;
  }
  front = 0;
  rear = size - 1;
  capacity = newCapacity;
  theData = newData;
}
```

Comparing the Three Implementations

Computation time

- All three implementations (double-linked list, single-linked list, circular array) are comparable in terms of computation time
- All operations are $\mathcal{O}(1)$ regardless of implementation
- Although reallocating an array is $\mathcal{O}(n)$, it is amortized over n items, so the cost per item is $\mathcal{O}(1)$

Comparing the Three Implementations

Storage

- Linked-list implementations require more storage due to the extra space required for the links
  - Each node for a single-linked list stores two references (one for the data, one for the link)
  - Each node for a double-linked list stores three references (one for the data, two for the links)
- A double-linked list requires 1.5 times the storage of a single-linked list
- A circular array that is filled to capacity requires half the storage of a single-linked list to store the same number of elements, but a recently reallocated circular array is half empty, and requires the same storage as a single-linked list
- All three implementations (double-linked list, single-linked list, circular array) are comparable in terms of computation time