Introduction to Computation and Problem Solving

Class 23: Introduction to Data Structures: Stacks and Queues

Prof. Steven R. Lerman

and

Dr. V. Judson Harward

Algorithms

An *algorithm* is a precise but not necessarily formal description of how to solve a class of computational problem. Examples: insertion sort, Newton's method,

A *data structure* is a general method of storing and accessing data that optimizes or organizes one or more aspects of data access, e.g., speed, ordering, etc. Examples, arrays, `Vector`

A *pattern* is a strategy for solving a problem, specific enough to be recognizable, but general enough so that the programmer will customize it for each case. Examples: next lecture!

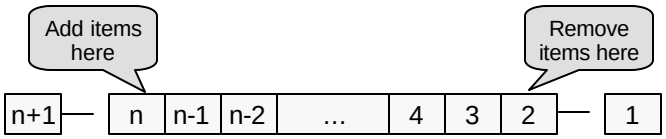
Data Structures

- **A good example is a queue. We encounter queues all the time in every day life.**
- **What makes a queue a queue? What is the essence of queueness?**

3

Queues

A *queue* is a data structure to which you add new items at one end and remove old items from the other.

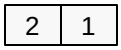


4

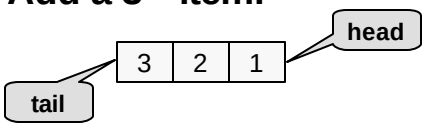
Queue Operations

Add an item:

Add another item:



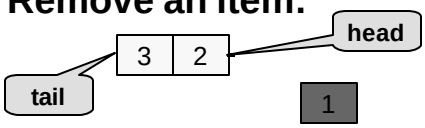
Add a 3rd item:



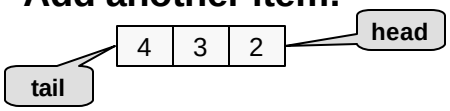
5

Queue Operations, 2

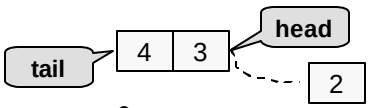
Remove an item:



Add another item:



Remove another item:



6

Queue Interface

```
public interface Queue
{
    public boolean isEmpty();
    public void add( Object o );
    public Object remove() throws
          NoSuchElementException;
    public void clear();
}
```

7

Abstract vs Concrete Data Types

The "data structure" actually combines two object-oriented concepts:

- **the data structure *interface*, which defines an *abstract data type (ADT),* and**
- **a particular implementation of that interface that provides a *concrete data type (CDT)*.**

8

Uses for Queues

- Queues are useful in many algorithms and in situations where two systems pass information to each other but will operate more efficiently without handshaking on each information transfer.
- Queues buffer the information transfer and allow the queue source and destination to process the information at independent rates.
- The Java® event queue is a good example.
- Derived types: priority queues, streams

9

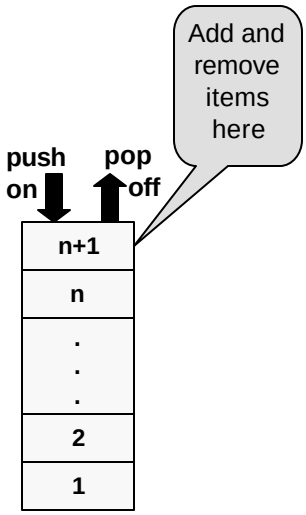
The Right Data Structure for the Job

- Suppose you are writing a graphics editor.
- You want to implement an unlimited Undo operation.
- You cleverly design a class that captures each modification you make as an object.
- You want to use these objects to undo the mods.
- What data structure do we want? A queue?

10

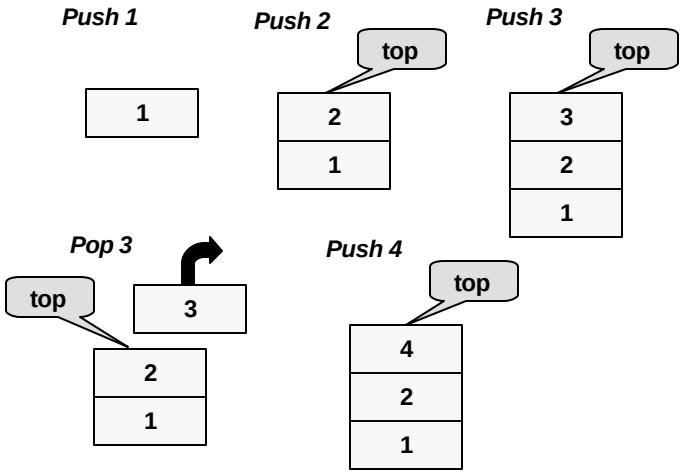
Stacks

- **Stacks resemble queues except that items are added and removed from the same end.**
- **Stacks are sometimes referred to as *LIFO queues* (last in first out) as opposed to *FIFO queues* (first in first out) which describe the base queue type.**



11

Stack Operations



12

Uses of Stacks

- Stacks provide an orderly way to postpone the performance of subsidiary tasks encountered during program execution.
- They are often associated with recursive algorithms.
- Derived types: program stacks, parser stacks

13

Stack Interface

```
public interface Stack
{
    public boolean isEmpty();
    public void push( Object o );
    public Object pop() throws
          EmptyStackException;
    public void clear();
}
```

14

Using a Stack to Reverse an Array

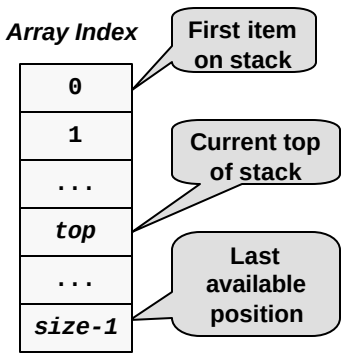
```
public class Reverse {
    public static void main(String args[]) {
        int [] array = { 1, 2, 3, 4, 5 };
        int i;
        Stack stack = new ArrayStack();

        for ( i = 0; i < array.length; i++ )
            stack.push( new Integer( array[ i ] ));
        i = 0;
        while ( !stack.isEmpty() ) {
            array[ i ] =
              ((Integer) stack.pop()).intValue();
            System.out.println( array[ i++ ] );
        }
    }
}
```

15

Stack Implementation Based on an Array

- In our array-based implementation, the top of the stack will move down the array as we push items onto it and will move back up as we pop them.
- The bottom of the stack will always lie at element 0 of the array.
- In a sense, we are building the stack "upside down" in the array.



16

ArrayStack,1

```
import java.util.*;
public class ArrayStack implements Stack {
  static public final int DEFAULT_CAPACITY = 8;
  private Object[] stack;
  private int top = -1;
  private int capacity;

  public ArrayStack(int cap) {
    capacity = cap;
    stack = new Object[capacity];
  }
  public ArrayStack() {
    this( DEFAULT_CAPACITY );
  }
```

17

ArrayStack,2

```
  public boolean isEmpty() {
    return ( top < 0 );
  }

  public void clear() {
    for ( int i = 0; i < top; i++ )
      stack[ i ] = null;  // for garbage collection
    top = -1;
  }
```

18

ArrayStack,3

```
public void push(Object o) {
  if (++top == capacity)
    grow();
  stack[top] = o;
}

private void grow() {
  capacity *= 2;
  Object[] oldStack = stack;
  stack = new Object[capacity];
  System.arraycopy(oldStack, 0, stack, 0, top);
}
```

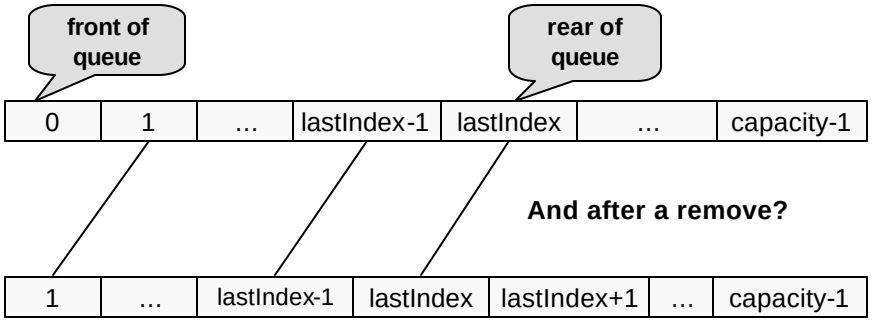
19

ArrayStack,4

```
public Object pop()
  throws EmptyStackException
{
  if ( isEmpty() )
    throw new EmptyStackException();
  else {
    // your code goes here
    // remove and return item on top of the stack;
    // adjust private variables; free memory
    return null;
  }
}
```

20

A Naïve Queue Implementation

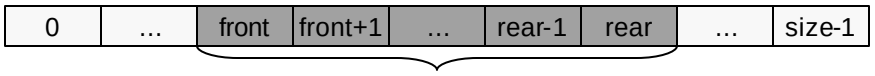


21

A Ring Queue Implementation Based on Array

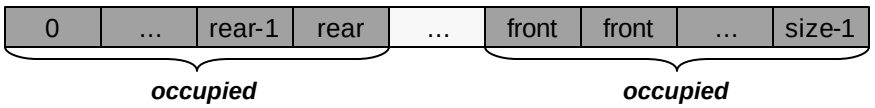
Two cases:

1. `front < rear`



occupied

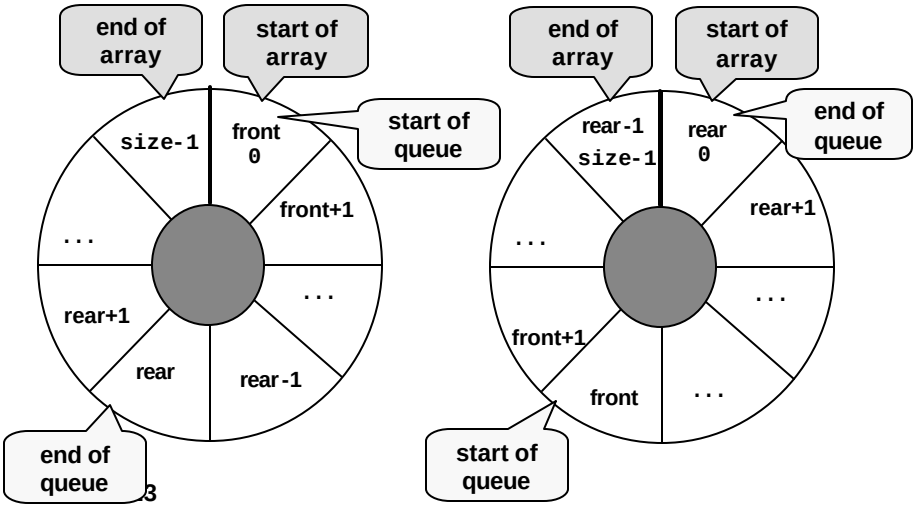
2. `rear < front`, queue has wrapped round



occupied ***occupied***

22

Ring Structure of Array-Based Queue



Implementing a Ring Queue

- Implementing a `RingQueue` is surprisingly difficult. The trick is to use the concept of *invariants*.
- An invariant is a property of a class that is true whenever one of the methods of the class is not executing. While a class method is executing, the class can violate the invariant for a moment, but it must restore it before exiting the method.

```
public class RingQueue implements Queue {
  private Object[] queue;
  private int front;
  private int rear;
  private int capacity;
  private int size = 0;
```

RingQueue Invariants

When not executing a static or instance method of RingQueue all the following must hold:

queue: holds a reference to the ring array

size: Always >=0. Holds the # of items on the queue

front: if size>0, holds the index to the next item to be removed from the queue

rear: if size>0, holds the index to the last item to be added to the queue

capacity: Holds the size of the array referenced by queue

Sample RingQueue Methods

```
public boolean isEmpty() {
      return ( size == 0 );
}

public Object remove() {
  if ( isEmpty() )
    throw new NoSuchElementException();
  else {
    Object ret = queue[ front ];
    queue[ front ] = null;  //for garbage collection
    front = (front + 1) % capacity;
    size--;
    return ret;
  }
}
```

Stack Exercise, 1

- You will write the `pop()` method for `ArrayStack` and write the method in UndoViz that implements Undo.
- Download Lecture23.zip from the web site.
- Open it in Winzip and extract it to a new directory;
  1. Click the extract button, which will open the extract dialogue.
  2. Use the browser on the right to navigate to where you want to locate the new directory.
  3. Click "New Folder" and name it.
  4. Then click the extract button in the upper right corner of the dialogue.
- Open a new project in Forte and mount the directory you just created

27

Stack Exercise, 2

- Look over ArrayStack.java.
- The user interface is all contained in UndoViz.java. This contains the main method to run the visualization. You don't need to understand the code in most of the methods. Just note how the undo button calls `do_undo()` when it is pressed.
- Compile the project and execute UndoViz. (Right button down and select execute). Confirm that the Undo button does not work.
- Now complete the `pop()` method in `ArrayStack` and the `do_undo()` method in `UndoViz`. You shouldn't have to change any other methods.
- Recompile and check that Undo now works.

28

Stack Exercise, 3

```
public class UndoViz extends JPanel {
  ...
  public UndoViz() {
    ...
    undoButton.addActionListener(new ActionListener()
      {
        public void actionPerformed(ActionEvent evt) {
          do_undo();
      }
    });
    ...
  }
  ...
  public void do_undo() {
    // Your code goes here.
    // Undo the last graphic addition and redraw.
  }
```

29

`RingQueue` Exercise

This exercise is more challenging. Don't worry if you can't complete it. Solutions will be posted on the web.

- **You will write the `add()` method for `RingQueue`.**
- **Build the whole project. Execute `RingQueueViz` and confirm that the add button doesn't work. Now write the `RingQueue` `add()` method. Make sure that by the time you exit the methods all invariants still hold. What will you do if `size == capacity` before you add an element?**
- **Compile and test by running `RingQueueViz`.**

30