STUDENT OUTLINE

Lesson 39 – Queues

INTRODUCTION: Queues are another restricted-access data structure, much like stacks. A queue is like a one-way line where data enters the end of the line and exits from the front of the line. Implementation of the `Queue` interface will support operations similar to, but different from, stacks.

The key topics for this lesson are:

A. Queues
B. Operations on Queues
C. Implementing Queues as a Linked List

VOCABULARY: QUEUE ENQUEUE
DEQUEUE

DISCUSSION:

A. Queues

1. A queue is a linear data structure that simulates waiting in line. A queue has two ends, a front and a (rear) end.

2. Data must always enter the queue at the end and leave from the front of the line. This type of action can be summarized as FIFO (first-in, first-out).

3. A queue is the appropriate data structure when simulating waiting in line. A printer that is part of a multi-user network usually processes print commands on a FIFO basis. A queue would be used to maintain the order of the print jobs.

B. Operations on Queues

1. The following operations are supported by the `Queue` interface:

```
public interface Queue
{
  boolean isEmpty();
  void enqueue(Object obj);
  Object dequeue();
  Object peekFront();
}
```

2. Using a queue implemented through the `Queue` interface is very similar to using a stack. Here is a sample program that uses some of the key operations provided by a `Queue` implementation.

Program 39-1

```
public static void main(String[] args)
{
  ListQueue queue;

  for (int k = 1; k <= 5; k++)
    queue.enqueue(new Integer(k));

  while (!queue.isEmpty())
  {
    System.out.println(queue.dequeue());
  }
}

Run output:

1 2 3 4 5
```

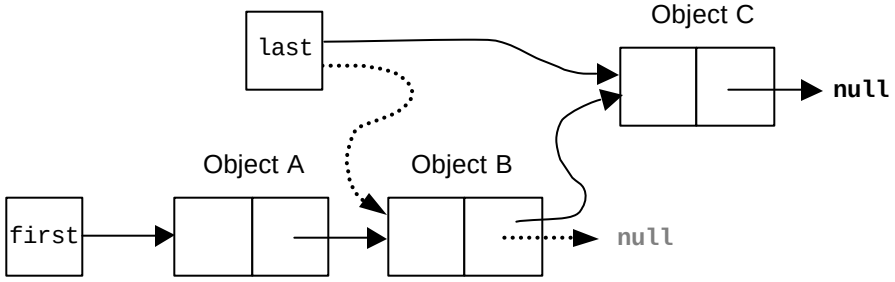
See Handout H.A.39.1, *Queue.*

3. See Handout H.A.39.1, *Queue Interface*[*] for the full specifications of the `Queue` interface.

C. Implementing Queues as a Linked List

1. A queue can be implemented as an array or a linked list. If we use a linked list to implement a queue we must deal with the following issues.

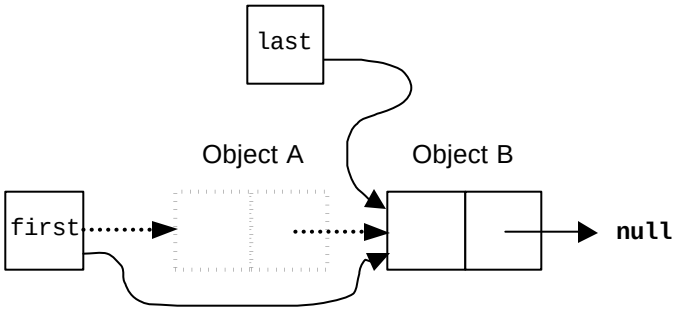
2. Two external pointers must be used to keep track of the front and end of the queue. When discussing a queue it is traditional to add new data at the end of the queue, as in "get in at the end of the line." The term *enqueue* is used to describe this operation.



[*] Adapted from the College Board's *AP Computer Science AB: Implementation Classes and Interfaces*.

An *Enqueue* Operation – Inserting at the End

3. Data will be removed from the "front" of the queue. This operation is called *dequeue*.



A *Dequeue* Operation – Deletion from the Front

4. A queue can be implemented as a singly linked list if the two external pointers are appropriately placed.

5. When a new piece of data is added to the tail of the queue (enqueue), the external pointer `last` must be changed to point to the new node added to the list.

6. When an old piece of data is extracted from the queue (dequeue), the external pointer `first` must be changed to point to the appropriate node after the data has been removed.

SUMMARY/ REVIEW: Queues serve a very useful function whenever a program needs to simulate waiting in line. One of the lab exercises will require queues to solve an intriguing problem regarding binary trees.

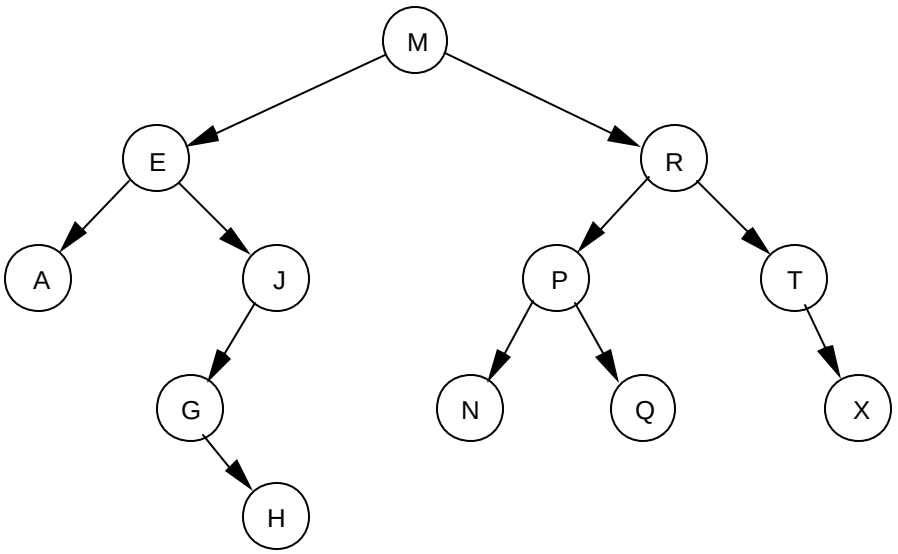
ASSIGNMENT: Lab Exercise L.A.39.1, *Printbylevel*
Lab Exercise L.A.39.2, *RPN*

LAB EXERCISE

PrintByLevel

Background:

1. A different kind of binary tree traversal scheme is to visit the nodes level by level. Your task in this lab exercise is to print out a binary tree by level from left to right. For example, this binary tree of letters will result in the following output of letters:



M E R A J P T G N Q X H

Assignment:

1. Use the `ListQueue` class to provide the necessary queue routines for this lab exercise.

2. Starting with the earlier binary tree lab exercise in Lesson 37, L.A.37.1, *TreeStats*, build a binary tree of characters ordered by letter.

3. Write a method `printLevel` that prints out the tree, level by level from left to right. The output can be formatted in one line as in the above example.

Instructions:

1. Use the same data files (*fileA.txt*) and (*fileB.txt*) as in the earlier binary tree lab in Lesson 37, L.A.37.1, *TreeStats*.

2. Turn in your source code and the two run outputs.

LAB EXERCISE

RPN

Background:

1. Most calculators use infix notation, in which the operator is typed between the operands. You have grown up solving infix math expressions such as `2 + 3` or `9 - 5`.

2. There are two other types of binary math notation systems that are called prefix and postfix. In prefix the expressions look like `+ 2 3` or `- 9 5`. Notice that the operator comes before ("pre") the operands.

3. Postfix binary expressions are used on some calculators, as complex math expressions can be entered without the need for parentheses. Postfix math is also called reverse polish notation (RPN). These statements look like `2 3 +` or `9 5 -`. The advantage of postfix (and prefix) is the elimination of parentheses to solve more complex problems.

4. Here is a comparison of infix versus postfix math expressions:

infix	postfix
`5 + ((7 + 9) * 2)`	`5 7 9 + 2 * +`

   A postfix expression is solved in this manner. If a value is entered it is placed on a stack. If an operation is entered (`+`, `-`, `*`, `/`) the stack is popped twice and the operation is applied to those two numbers. The resulting answer is placed back on the stack. The expression

   `5 7 9 + 2 * +` is solved in this order:

```
5  7  9  +  2  *  +
5  16  2  *  +
5  32  +
39
```

   Notice that postfix expressions are simply solved left to right and that parentheses are not needed.

5. The following special cases must be addressed. If the problem to be solved is `7 - 5` (infix), the problem is entered on an RPN calculator in this order:

```
7  (enter)
5  (enter)
-
```

   This causes the stack to be popped twice and the correct math to be solved is `7 5 -`, which equals `2`.

   A similar ordering issue surrounds the (`/`) operator. The infix problem `9 / 2` is entered on an RPN calculator as:

```
9
```

```
2
/
```
which returns `4`.

Assignment:

1. Write a program to implement a simple RPN calculator that processes only this type of input:
   single-digit integers
   the four integer math operations: +, -, *, /

2. The user will type in single character input until 'Q' or 'q' is entered. As described above, if a digit is entered, the number is placed on the stack. If an operation is entered, pop two values off the stack, do the math, then return the answer back onto the stack. As keyboard input is entered the program should keep track of all keystrokes entered to be replayed after 'Q' or 'q' is entered.

3. When 'Q' or 'q' is entered, the program will print out the entire problem and the answer. A queue and a stack would be a good thing to use in this lab.

Instructions:

1. Enter the following 5 postfix problems:

```
9  5  -
7  3  *  6  /
8  4  7  +  -
7  3  9  4  5  *  +  -  -
8  4  3  *  *  6  4  2  -  +  +
```

2. Your run output should look something like this:

```
9  5  -  =  4
7  3  *  6  /  =  3
8  4  7  +  -  =  -3
7  3  9  4  5  *  +  -  -  =  33
8  4  3  *  *  6  4  2  -  +  +  =  104
```

Queue Interface* and Implementation

```
public interface Queue
{
    // postcondition: returns true if queue is empty, false otherwise
  boolean isEmpty();

    // precondition:  queue is [e1, e2, ..., en] with n >= 0
    // postcondition: queue is [e1, e2, ..., en, x]
  void enqueue(Object x);

    // precondition:  queue is [e1, e2, ..., en] with n >= 1
    // postcondition: queue is [e2, ..., en]; returns e1
    //                throws an unchecked exception if the queue is empty
  Object dequeue();

    // precondition:  queue is [e1, e2, ..., en] with n >= 1
    // postcondition: returns e1
    //                throws an unchecked exception if the queue is empty
  Object peekFront();
}

public class ListQueue implements Queue
{
  private java.util.LinkedList list;

  public ListQueue() { list = new java.util.LinkedList(); }
  public boolean isEmpty() { return list.size() == 0; }
      // Or: ... isEmpty() { return list.isEmpty(); }
  public void enqueue(Object obj) { list.addLast(obj); }
  public Object dequeue() { return list.removeFirst(); }
  public Object peekFront() { return list.getFirst(); }
}
```

* Adapted from the College Board's *AP Computer Science AB: Implementation Classes and Interfaces*.