CS200: Queues

- Prichard Ch. 8
Queues

- First In First Out (FIFO) structure
- Imagine a checkout line
- So removing and adding are done from opposite ends of structure.
  - add to tail (back), remove from head (front)
- Used in operating systems (e.g. print queue).
Possible Queue Operations

- **enqueue**(in `newItem`: `QueueItemType`)
  - Add new item at the **back** of a queue

- **dequeue()**: `QueueItemType`
  - Retrieve and remove the item at the **front** of a queue

- **peek()**: `QueueItemType`
  - Retrieve item from the **front** of the queue. Retrieve the item that was added earliest.

- **isEmpty()**: `boolean`

- **createQueue()**
A linked list with two external references

- A reference to the front
- A reference to the back

At which end do we enqueue / dequeue?

WHY?

Reference of the First Node
Reference of the Last Node
Reference-Based Implementation 2

A circular linked list with one external reference

- lastNode references the back of the queue
- lastNode.getNext() references the front

Last Node: node reference
Inserting an item into a nonempty queue

1. newNode.next = lastNode.next;
2. lastNode.next = newNode;
3. lastNode = newNode;

New node
Inserting a New Item

- Insert a new item into the empty queue
public void enqueue (Object newItem){
    Node newNode = new Node(newItem);
    if (isEmpty()){
        newNode.next = newNode;
    } else {
        newNode.next = lastNode.next;
        lastNode.next = newNode;
    }
    lastNode = newNode;
}
Removing an item from queue

public Object dequeue() throws QueueException{
    if (!isEmpty()){
        Node firstNode = lastNode.next;
        if (firstNode == lastNode) {
            lastNode = null;
        }
        else{
            lastNode.next = firstNode.next;
        }
        return firstNode.item;
    }
    else { exception handling..
}
}
Removing an Item

What happens to this node?

First Node

Last Node

Node firstNode = lastNode.next;
if (firstNode == lastNode) {
    lastNode = null;
} else{
    lastNode.next = firstNode.next;
}
return firstNode.item;
Naïve Array-Based Implementation

Drift wastes space

How do we initialize front and back?
(Hint: what does a queue with a single element look like?
what does an empty queue look like?)
Solving Drift:
Circular implementation of a queue

MAX_QUEUE-1

0 1 2 3 4 5 6

FRONT
BACK

a e i o
Solving Drift:
- Delete
Solving Drift:

- Delete

MAX_QUEUE-1

FS200 - Queues
Solving Drift

- Insert u

When either front or back advances past MAX_QUEUE-1, it wraps around (to 0: using % MAX_QUEUE)
Queue with Single Item

- *back* and *front* are pointing at the same slot.
Empty Queue: remove Single Item

Remove last item.

- \textit{front} passed \textit{back}.

When the queue is EMPTY, \textit{front} is one slot ahead of \textit{back}.
Insert the last item

*back* catches up to *front* when the queue becomes **full**.

When the queue is FULL, *front* is one slot ahead of *back* as well.

Problem?

Solution?

Maintain size:

0:empty
max_queue: full

CS200 - Queues
Wrapping the values for front and back

- Initializing
  front = 0
  back = MAX_QUEUE - 1
  count = 0

- Adding
  back = (back + 1) % MAX_QUEUE;
  items[back] = newItem;
  ++count;

- Deleting
  deleteItem = items[front];
  front = (front + 1) % MAX_QUEUE;
  --count;
public void enqueue(Object newItem) throws QueueException{

    if (!isFull()){
        back = (back+1) % (MAX_QUEUE);
        items[back] = newItem;
        ++count;
    }
    else {
        throw new QueueException(your_message);
    }
}
public Object dequeue() throws QueueException{

    if (!isEmpty()){
        Object queueFront = items[front];
        front = (front+1) % (MAXQUEUE);
        --count;
        return queueFront;
    }else{
        throw new QueueException (your_message);
    }
}
You can implement operation `dequeue()` as the list operation `remove(0)`.

`peek()` as `get(0)`

`enqueue()` as `add(newItem) // at tail`
Questions

What is an advantage of the circular array implementation over linked list?

A. Faster to enqueue
B. Uses less memory
C. Can more easily fix and enforce a maximum size
D. Fewer allocations
Expressions: infix to postfix conversion

Prichard: 7.4
Let’s do some
2 + 3 * 4
2 * 3 + 4
2 + 3 - 4
2 + (3 - 4)
2 - 3 - 4
1 - (2 + 3 * 4) / 5

observations?
Expressions: infix to postfix conversion

\[
\begin{align*}
2 + 3 \times 4 & \rightarrow 2 3 4 \times + \\
2 \times 3 + 4 & \rightarrow 2 3 4 \times + \\
2 + 3 - 4 & \rightarrow 2 3 4 - \\
2 + (3 - 4) & \rightarrow 2 3 4 - + \\
2 - 3 - 4 & \rightarrow 2 3 4 - \\
1 - (2 + 3 \times 4) / 5 & \rightarrow 1 2 3 4 \times + 5 / - \\
\end{align*}
\]

1. operand order does not change
2. operators come after second operand and obey associativity and precedence rules
3. ( ) converts the inner expression to an independent postfix expression
infix to postfix implementation

- Use a **queue** to create the resulting postfix expression
  - the operands get immediately enqueued

- Use a **stack** to store the operators
  - operators get pushed on the stack

- when to pop and enqueue?
  - let’s play
\[
2 + 3 \times 4
\]

<table>
<thead>
<tr>
<th>Action</th>
<th>Stack</th>
<th>Queue</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(2 + 3 \times 4)</td>
<td></td>
<td>enqueue</td>
</tr>
<tr>
<td></td>
<td>(+ 3 \times 4)</td>
<td>2</td>
<td>push</td>
</tr>
<tr>
<td></td>
<td>(3 \times 4)</td>
<td>+</td>
<td>enqueue</td>
</tr>
<tr>
<td></td>
<td>(* 4)</td>
<td>+</td>
<td>push</td>
</tr>
<tr>
<td></td>
<td>(*)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(4)</td>
<td>+</td>
<td>enqueue</td>
</tr>
<tr>
<td></td>
<td>(*)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(+)</td>
<td>2 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(+)</td>
<td>2 3 4</td>
<td>pop; enqueue</td>
</tr>
<tr>
<td></td>
<td>(+)</td>
<td>2 3 4</td>
<td>pop; enqueue</td>
</tr>
<tr>
<td></td>
<td>(2 3 4 * +)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2 * 3 + 4

<table>
<thead>
<tr>
<th>stack</th>
<th>queue</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 * 3 + 4</td>
<td></td>
<td>enqueue</td>
</tr>
<tr>
<td>* 3 + 4</td>
<td>2</td>
<td>push</td>
</tr>
<tr>
<td>3 + 4</td>
<td>*</td>
<td>enqueue</td>
</tr>
<tr>
<td>+ 4</td>
<td>*</td>
<td>push?</td>
</tr>
</tbody>
</table>

NO!! Because * has higher precedence than + and so binds to 2 3

| + 4          | *     | 2 3    | pop; enqueue|
| + 4          |       | 2 3 * 4| push        |
| 4            | +     | 2 3 * 4| enqueue     |
| +            | 2 3 * 4| pop; enqueue|
|              | 2 3 * 4| +      |

CS200 - Queues
\[ 2 - 3 + 4 \]

<table>
<thead>
<tr>
<th>stack</th>
<th>queue</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 - 3 + 4</td>
<td></td>
<td>enqueue</td>
</tr>
<tr>
<td>- 3 + 4</td>
<td>2</td>
<td>push</td>
</tr>
<tr>
<td>3 + 4</td>
<td>-</td>
<td>enqueue</td>
</tr>
<tr>
<td>+ 4</td>
<td>-</td>
<td>push?</td>
</tr>
</tbody>
</table>

NO!! Because of left associativity – binds to 2 3

| + 4       | -     | 2 3    | pop; enqueue |
| + 4       |       | 2 3 -  | push        |
| 4         | +     | 2 3 -  | enqueue     |
|           | +     | 2 3 - 4| pop; enqueue|
|           |       | 2 3 - 4|             |
\[ 2 - ( 3 + 4 ) \]

<table>
<thead>
<tr>
<th>stack</th>
<th>queue</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-</td>
<td>enqueue</td>
</tr>
<tr>
<td>- ( 3 + 4 )</td>
<td>2</td>
<td>push</td>
</tr>
<tr>
<td>( 3 + 4 )</td>
<td>-</td>
<td>delete or push?</td>
</tr>
</tbody>
</table>

the expression inside the ( ) makes its own independent postfix, so we push the ( then use the stack as before until we see a ) then we pop all the operators off the stack and enqueue them, until we see a ( and delete the ( ( 3 + 4 ) | -     | enqueue     |
| ( 3 + 4 ) | -     | 2 3 | push        |
| + 4 ) | -     | 2 3 | enqueue     |

continued next page
\[ 2 - (3 + 4) \]

<table>
<thead>
<tr>
<th>stack</th>
<th>queue</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>[+]</td>
<td>[2]</td>
<td>enqueue</td>
</tr>
<tr>
<td>[(4)]</td>
<td>[3]</td>
<td>pop, enqueue until [,], delete [()]</td>
</tr>
<tr>
<td>[(-)]</td>
<td>[2]</td>
<td>pop, enqueue until stack empty</td>
</tr>
<tr>
<td>[(-)]</td>
<td>[3]</td>
<td></td>
</tr>
<tr>
<td>[(-)]</td>
<td>[4]</td>
<td></td>
</tr>
<tr>
<td>[(-)]</td>
<td>[+]</td>
<td></td>
</tr>
<tr>
<td>[(-)]</td>
<td>[-]</td>
<td></td>
</tr>
</tbody>
</table>

CS200 - Queues

31
in2post algorithm

when encountering

**operand:** enqueue

**open:** push

**close:**
  pop and enqueue operators, until open on stack
  pop open

**operator:**
  if stack empty or top is open push, else
  pop and enqueue operators with greater or equal
  precedence, until operator with lower precedence on
  stack, or open on stack, or stack empty

**end of input:**
  pop and enqueue all operators until stack empty

Do it for: \( 1-(2+3*4)/5 \)
What about unary operators?

- e.g. `not` in logic expressions such as:
  - `not true and false`
  - `not ( true or false )`
  - `not not true`

`not` has higher priority than `and`,

- `true and not false` is `true and (not false)`

`and` has higher priority than `or`

`not` is right associative

- `not not true` is `not ( not true )`
<table>
<thead>
<tr>
<th>stack</th>
<th>queue</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>not true and false</td>
<td></td>
<td>push</td>
</tr>
<tr>
<td>true and false</td>
<td>not</td>
<td>enqueue</td>
</tr>
<tr>
<td>and false</td>
<td>not true</td>
<td>not higher priority</td>
</tr>
<tr>
<td>and false</td>
<td>true</td>
<td>pop, enqueue not</td>
</tr>
<tr>
<td>false</td>
<td>and true not</td>
<td>push</td>
</tr>
<tr>
<td></td>
<td>true not false</td>
<td>enqueue</td>
</tr>
<tr>
<td></td>
<td>true not false</td>
<td>pop, enqueue</td>
</tr>
</tbody>
</table>
not(true or false)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Queue</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>not (true or false)</td>
<td></td>
<td>push</td>
</tr>
<tr>
<td>(true or false)</td>
<td>not</td>
<td>push</td>
</tr>
<tr>
<td>(true or false)</td>
<td>not</td>
<td>enqueue</td>
</tr>
<tr>
<td>or false</td>
<td>not</td>
<td>true</td>
</tr>
<tr>
<td>or</td>
<td>(false)</td>
<td>not</td>
</tr>
</tbody>
</table>
not(true or false)  continued

<table>
<thead>
<tr>
<th>stack</th>
<th>queue</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>not</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enqueue</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>not</td>
<td>true false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>until (</td>
</tr>
<tr>
<td>not</td>
<td>true</td>
<td>false or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pop, enqueue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pop, enqueue</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>or not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td>queue</td>
<td>action</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
<td>true</td>
</tr>
<tr>
<td>not true</td>
<td>not</td>
<td>push</td>
</tr>
<tr>
<td>not true</td>
<td>not</td>
<td>push or enqueue?</td>
</tr>
<tr>
<td>push!</td>
<td>not is right associative, its operand is ahead of it</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>not</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>not</td>
<td>enqueue</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
<td>enqueue</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
<td>pop and enqueue</td>
</tr>
</tbody>
</table>

true not not
in2post algorithm

when encountering

**operand:** enqueue

**open:** push

**close:**

pop and enqueue operators, until open on stack
pop open

**end of input:**

pop and enqueue all operators until stack empty
in2post continued

when encountering

**and, or:**
- if stack empty or top is open push, else pop and enqueue operators with greater or equal precedence, until operator with lower precedence on stack, or open on stack, or stack empty

**not:**
- push

do it for not (not true or false)