COMP 250
Lecture 18
queue ADT
ADT (abstract data type)

- **List**
  - add(i,e), remove(i), get(i), set(i), ..

- **Stack**
  - push, pop(), ..

- **Queue**
  - enqueue(e), dequeue()
Queue

decode
(remove from front)

enqueue
(add at back)

e.g. server

clients
Examples

• keyboard buffer
  (delay when you type)

• CPU processes
  (different applications do not run in parallel; they line up and each gets a certain amount of time on the CPU and then they have to line up again)

• web server
  (many customers trying to access the same web site)

• cafeteria ...
Stack
push(e)
pop()
LIFO
(last in, first out)

Queue
enqueue( e )
dequeue()
FIFO
(first in, first out)
“first come, first serve”
Queue Example

enqueue( a )
enqueue( b )
dequeue( )
enqueue( c )
enqueue( d )
enqueue( e )
dequeue( )
enqueue( f )
enqueue( g )

returns a
Queue Example

enqueue(a)
enqueue(b)
dequeue()
enqueue(c)
enqueue(d)
enqueue(e)

time returns a

a
ab
b
bc
bcd
bcde
Queue Example

enqueue( a )
enqueue( b )
dequeue( )
enqueue( c )
enqueue( d )
enqueue( e )
dequeue( )
enqueue( f )
enqueue( g )
Queue Example

enqueue(a)
enqueue(b)
dequeue()
enqueue(c)
enqueue(d)
enqueue(e)
dequeue()
enqueue(f)
enqueue(g)
returns b
### How to implement a queue?

<table>
<thead>
<tr>
<th></th>
<th>enqueue(e)</th>
<th>dequeue()</th>
</tr>
</thead>
<tbody>
<tr>
<td>singly linked list</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>doubly linked list</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>array list</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
How to implement a queue?

<table>
<thead>
<tr>
<th></th>
<th>enqueue(e)</th>
<th>dequeue()</th>
</tr>
</thead>
<tbody>
<tr>
<td>singly linked list</td>
<td>addLast(e)</td>
<td>removeFirst()</td>
</tr>
<tr>
<td>doubly linked list</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>array list</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
How to implement a queue?

- **enqueue(e)**
  - *singly linked list*: addLast(e)
  - *doubly linked list*: same, or addFirst() & removeLast()
  - *array list*: ?

- **dequeue()**
  - *singly linked list*: removeFirst()
  - *doubly linked list*: ?
  - *array list*: ?
# How to implement a queue?

<table>
<thead>
<tr>
<th>Method</th>
<th>Singly Linked List</th>
<th>Doubly Linked List</th>
<th>Array List*</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue(e)</td>
<td>addLast(e)</td>
<td>same, or addFirst() &amp; removeLast()</td>
<td>removeFirst() (inefficient)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>removeFirst()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*array lists generally don’t have addLast or removeFirst methods, but they have other methods that can do the same thing.*
Implementing a queue with an array list. (inefficient)

length = 4

0123 indices

enqueue( a ) a---
enqueue( b ) ab--
dequeue( ) b---
enqueue( c ) bc--
enqueue( d ) bcd-
enqueue( e ) bcde
enqueue( f )
enqueue( g ) cde-

requires a shift

requires a shift
Circular array

length = 4

0123

length = 8

01234567
Circular array

enqueue( a )

head=0  tail=0
Circular array

enqueue(a)
enqueue(b)

head=0   tail=1

0123

ab--

1 2 3

a b
enqueue( a )
enqueue( b )

decqueue()  ?

head=0  tail=1
Circular array

enqueue( a )
enqueue( b )
dequeue()
enqueue( c )

head=1  tail=1

\[ \text{tail} = (\text{head} + \text{size} - 1) \mod \text{length} \]

Note: if size = 1, then tail = head.
Circular array

enqueue(a)
enqueue(b)
dequeue()
enqueue(c)
enqueue(d)

head=1 \; \text{tail}=2

\text{tail} = (\text{head} + \text{size} - 1) \mod \text{length}

Note: if size = 1, then tail = head.
Circular array

enqueue( a )
enqueue( b )
dequeue()  
enqueue( c )
enqueue( d )

enqueue( e )

head=1   tail=3

0123
-bcd
Circular array

enqueue( a )
enqueue( b )
dequeue( )
enqueue( c )
enqueue( d )
enqueue( e )

dequeue() ?

tail=0  head=1

tail = (head + size – 1) mod length
Circular array

enqueue( a )
enqueue( b )
dequeue( )
enqueue( c )
enqueue( d )
enqueue( e )
dequeue()

enqueue( f )?

tail = (head + size - 1) mod length
Circular array

enqueue(a)
enqueue(b)
dequeue()
enqueue(c)
enqueue(d)
enqueue(e)
dequeue()
enqueue(f)

tail = 1  head = 2

tail = (head + size – 1) mod length

NOTE: When size = length, we have tail = (head – 1) mod length.
enqueue( a )
enqueue( b )
dequeue()
enqueue( c )
enqueue( d )
enqueue( e )
dequeue()
enqueue( f )
enqueue( g )

\[ \text{tail} = (\text{head} + \text{size} - 1) \mod \text{length} \]
How to \texttt{enqueue}(g) ?
Increase length of array and copy? \textbf{NO}

```
head = 2
tail = 1
size = 4
```
How to enqueue \( (g) \)?
Increase length of array. Copy such that head stays as is. Add the new element \( g \).
How to enqueue($g$) ? (Alternative)
Increase length of array. Copy elements so that head is 0. Add the new element $g$. 

![Diagram showing array manipulation]

- $head = 2$
- $tail = 1$
- $size = 4$

- $head = 0$
- $tail = 4$
- $size = 5$
**enqueue( element )**{ // using a circular array
  if ( size == length ) { 

  Increase length of array.
  Copy elements so that head is 0.

  }

  tail = (tail + 1) mod length
  queue[ tail ] = element
  size = size + 1

}

**NOTE:**
We don’t actually need the tail variable, since tail = (head + size – 1) mod length.
enqueue( element ){                         // using a circular array
    if ( size == length ) {                  // increase length of array
        create a bigger array tmp[ ]     // e.g. 2*size
        for i = 0 to size - 1
            tmp[i] = queue[ (head + i) mod size ]
        head = 0
        tail = size - 1
        queue = tmp
    }
    tail = (tail + 1) mod length
    queue[ tail ] = element
    size = size + 1
}

NOTE:
We don’t actually need the tail variable, since tail = (head + size – 1) mod length.
dequeue( ){                   // using a circular array

    element = queue[head]
    size = size – 1
    head = (head+1) mod length
    return element

}

Do you modify the head? the tail? both?
dequeue() {  // using a circular array

  // check that queue.size > 0  (omitted)

  element = queue[head]
  size =  size – 1
  head = (head+1) mod length
  return  element
}

Note:  this does not affect the tail.
What is the relation between head and tail when size == 0?
Suppose length = 4.

Initial state  ----  (0, 3, 0)

array  (head, tail, size)

\[
tail = (head + size - 1) \mod length
\]
What is the relation between head and tail when size == 0? Suppose length = 4.

\[ \text{tail} = (\text{head} + \text{size} - 1) \mod \text{length} \]

Initial state
- \( (0, 3, 0) \)

enqueue( a )
- \( (0, 0, 1) \)

enqueue( b )
- \( (0, 1, 2) \)

dequeue()
- \( (1, 1, 1) \)

dequeue()
- \( (2, 1, 0) \)
Recall: ADT (abstract data type)

Defines a data type by the values and operations from the user’s perspective only. It ignores the details of the implementation.

Examples:
- list
- stack
- queue
- ...

Exercise: what can we do with just the operations of an ADT?
Exercise: Implement a queue using a stack(s).
Hint: you want FIFO (first in, first out) behavior.

enqueue(e) {
    // use only push(e), pop(), isEmpty()
}

dequeue() {
    // use only push(e), pop(), isEmpty()
}

You are also allowed temporary variables, loops, etc.
Suppose we implement `enqueue` as above.

The example on the left shows what happens when we “enqueue” elements `a` to `i`, namely we push them onto a stack.

But how do we implement `dequeue()` in this case?
Hint: Use a second stack.

Write pseudocode that uses only operations `push(e)`, `pop()`, `isEmpty()`.

```plaintext
dequeue()

Write pseudocode that uses only operations `push(e)`, `pop()`, `isEmpty()`.
```
dequeue()
    while (!s.isEmpty()){
        tmpS.push(s.pop())
    }
    tmp = tmpS.pop()
}
dequeue(){
    while ( ! s.isEmpty() ){
        tmpS.push( s.pop() )
    }
    element = tmpS.pop()
}

s  tmpS
```c
int dequeue()
{
    while (!s.isEmpty())
    {
        tmpS.push( s.pop() )
    }
    element = tmpS.pop()
    while (!tmpS.isEmpty())
    {
        s.push( tmpS.pop() )
    }
    return element
}
```
### Coming up...

#### Lectures

- Fri. Feb. 18  Mathematical Induction
- Next week... recursion

#### Assessments

- Assignment 2 is due on Fri. Feb. 25.
- Assignment 3 will be released shortly after that. (READING WEEK)