Today's Outline

• Queues, Stacks, and Lists
  › Fun with Arrays and Links
First Example: Queue ADT

- FIFO: First In First Out
- Queue operations
  - create
  - destroy
  - enqueue
  - dequeue
  - is_empty

Circular Array Queue Data Structure

```java
enqueue(Object x) {
    Q[back] = x;
    back = (back + 1) % size
}

decueue() {
    x = Q[front];
    front = (front + 1) % size;
    return x;
}
```

How test for empty list?
How to find K-th element in the queue?
What is complexity of these operations?
Limitations of this structure?
Linked List Queue Data Structure

```
void enqueue(Object x) {
    if (is_empty())
        front = back = new Node(x)
    else
        back->next = new Node(x)
        back = back->next
}

Object dequeue() {
    assert(!is_empty)
    return_data = front->data
    temp = front
    front = front->next
    back = back->next
    delete temp
    return return_data
}

bool is_empty() {
    return front == null
}
```

Circular Array vs. Linked List

- Too much space, or not enough space
- Too much space, or not enough space
- Not as complex
- Can grow as needed
- Could make array more robust
- More memory per item in the queue
- Linked list code more complex
Second Example: Stack ADT

- LIFO: Last In First Out
- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty

Stacks in Practice

- Function call stack
- Removing recursion
- Balancing symbols (parentheses)
- Evaluating Postfix Notation
  
  Infix: \[7 + 8 \times 4 - 30\]
  
  Postfix: \[7 \ 8 \ 4 \ * \ + \ 30 \ -\]
Infix to Postfix with a Stack

7 + 8 * 4 – 30

7

Infix to Postfix with a Stack

7 + 8 * 4 – 30

+ 7
Infix to Postfix with a Stack

\[ 7 + 8 * 4 - 30 \]

\[ + \]

\[ 7 \ 8 \]
Infix to Postfix with a Stack

\[ 7 + 8 \times 4 - 30 \]

\[ \times \]
\[ + \]
\[ + \]

\[ 7 \ 8 \ 4 \]

Infix to Postfix with a Stack

\[ 7 + 8 \times 4 - 30 \]

\[ \times \]
\[ + \]
\[ + \]
\[ + \]

\[ 7 \ 8 \ 4 \ \times \]
Infix to Postfix with a Stack

\[
7 + 8 \times 4 - 30
\]

*  
+  +  +  #  
7 8 4 * +
Infix to Postfix with a Stack

\[ 7 + 8 \times 4 - 30 \]

\[ \times \]

\[ + \quad + \quad + \quad \# \quad - \]

\[ 7 \quad 8 \quad 4 \quad \times \quad + \quad 30 \]

Infix to Postfix with a Stack

\[ 7 + 8 \times 4 - 30 \# \]

\[ \times \]

\[ + \quad + \quad + \quad \# \quad - \]

\[ 7 \quad 8 \quad 4 \quad \times \quad + \quad 30 \quad \# \quad - \]
Infix to Postfix with a Stack

\[ 7 + 8 \times 4 - 30 \]

\[ * \]

\[ + + + # - \]

\[ 7 \ 8 \ 4 \ * \ + \ 30 \ - \]

Third Example: List ADT

- Insert, Query, and Delete by *Position*

\[ \text{EACBD} \]

\[ \text{F} \]

\[ \text{E} \ \text{A} \ \text{C} \ \text{B} \ \text{D} \]
### Array vs. Linked List

<table>
<thead>
<tr>
<th>Array Pros</th>
<th>Linked List Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Too much space, or not enough space</td>
<td>Can grow as needed</td>
</tr>
<tr>
<td>Not as complex</td>
<td>More memory per item in the queue</td>
</tr>
<tr>
<td>Could make array more robust</td>
<td>Linked list code more complex</td>
</tr>
</tbody>
</table>

- What else now that we’re implementing list functionality?

<table>
<thead>
<tr>
<th>Array Cons</th>
<th>Linked List Pros</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kth element accessed “easily”</td>
<td>Can grow as needed</td>
</tr>
<tr>
<td>Insert requires shift</td>
<td>More memory per item in the queue</td>
</tr>
<tr>
<td></td>
<td>Linked list code more complex</td>
</tr>
</tbody>
</table>

- Kth element access requires linear scan??
Some Other Approaches

- Encapsulated position objects that store reference to linked nodes, together with next() and prev() methods
- Lists of arrays, reducing shift costs but increasing access costs
- And really wacky stuff:

![Diagram showing a list structure]

Projects / Homework

1) Look at and get started on Project 1
   Ask Eric Questions in Section Tomorrow

2) Homework 1 will be released shortly