STACK & QUEUES
Queues

- If we wanted to simulate customers waiting in a line to be served,
  - We wouldn’t use a stack...
    - LIFO is only going to make the person that got in line first mad.
Queues

- We would want to use FIFO
  - First In First Out, or 1st in line 1st one to get served.
- Instead of push and pop, we have the operations
  - Enqueue and Dequeue that add/remove elements from the list.
Sidenote: Abstract Data Type

- Queues are another example of an abstract data type (ADT)
  - ADT - Something that is not built into the language, and it is defined in terms of its behavior.
  - So if I tell you to use *MY* implementation of a queue to simulate customer wait times
    - You wouldn’t need to know how I implemented it, you could just call the functions – Enqueue, Dequeue, etc.
Queue Basic Operations

- **Enqueue:**
  - Inserts an element at the back of the queue
  - Returns 1 if successful, 0 otherwise.

- **Dequeue:**
  - Removes the element at the front of the queue.
  - Returns the removed element.

- **Peek**
  - Looks at the element at the front of the queue without removing it.
  - Returns the front element.

- **isEmpty**
  - Checks to see if the queue is empty.
  - Returns true or false.

- **isFull**
  - Checks to see if the queue is full.
  - Returns true or false.
Queue Example

Starting Queue:

<table>
<thead>
<tr>
<th>Front</th>
<th>Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Enqueue(13)</td>
</tr>
<tr>
<td>2</td>
<td>Dequeue()</td>
</tr>
<tr>
<td>3</td>
<td>Enqueue(15)</td>
</tr>
<tr>
<td>4</td>
<td>Dequeue()</td>
</tr>
<tr>
<td>5</td>
<td>Dequeue()</td>
</tr>
</tbody>
</table>

Time 1:

<table>
<thead>
<tr>
<th>Front</th>
<th>Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

Time 2:

<table>
<thead>
<tr>
<th>Front</th>
<th>Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

Time 3:

<table>
<thead>
<tr>
<th>Front</th>
<th>Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>15</td>
</tr>
</tbody>
</table>

Time 4:

<table>
<thead>
<tr>
<th>Front</th>
<th>Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>15</td>
</tr>
</tbody>
</table>

Time 5:

<table>
<thead>
<tr>
<th>Front</th>
<th>Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>15</td>
</tr>
</tbody>
</table>
Queues - Array Implementation

- What would we need for an array implementation?
  - We need an array obviously
  - And we need to keep track of the front and the back.
BAD Queue Implementation Example

Starting Queue: 3 5 7 9 11
front back

Time 1: 3 5 7 9 11 13
front back

Time 2: 5 7 9 11 13
front back

Time 3: 5 7 9 11 13 15
front back

Time 4: 7 9 11 13 15
front back

Time 5: 9 11 13 15
front back

<table>
<thead>
<tr>
<th>TIME</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Enqueue(13)</td>
</tr>
<tr>
<td>2</td>
<td>Dequeue()</td>
</tr>
<tr>
<td>3</td>
<td>Enqueue(15)</td>
</tr>
<tr>
<td>4</td>
<td>Dequeue()</td>
</tr>
<tr>
<td>5</td>
<td>Dequeue()</td>
</tr>
</tbody>
</table>

Notice that you have to Shift the contents of the Array over each time front changes
Queues: Array Implementation

- We will use the following revamped idea to store our queue structure:
  - Keep track of the array, the front, and the current number of elements.

```c
struct queue {
    int *elements;
    int front;
    int numElements;
};
```
Queues: Array Implementation

- Enqueue:
  - We’ll simply add the given element to the index “back” in the array.
  - BUT we’re not storing “back”!!!!!
  - What must we do instead?
    - Add it to the index: front + numElements
    - But what if this goes outside the bounds of our array?

```c
struct queue {
    int *elements;
    int front;
    int numElements;
};
```

```
| 9 | 11 | 13 | 15 |   |
```

numElements = 4
Queues: Array Implementation

- **Enqueue(17):**
  - Add it to the index: front + numElements
  - But what if this goes outside the bounds of our array?
  - Front = 2, plus numElements = 4, gives us 6
  - We can mod by the queueSize
  - \((\text{front} + \text{numElements}) \mod \text{queueSize} = 0\)

```c
struct queue {
    int *elements;
    int front;
    int numElements;
    int queueSize;
};
```

```
17 9 11 13 15
```

```c
numElements = 5
```
So we’re allowing our array to essentially wrap around.

- This way we don’t have to copy the contents of our array over if front or back moves.

```c
struct queue {
    int *elements;
    int front;
    int numElements;
    int queueSize;
};
```

```
numElements = 5

17 9 11 13 15
front
```
Queues: Array Implementation

- Dequeue
  - If the numElements > 0
    - numElements--;
    - front = (front + 1) % queueSize

```c
struct queue {
    int *elements;
    int front;
    int numElements;
    int queueSize;
};
```

numElements = 4

```
17 11 13 15
```
front front
What if our numElements == queueSize?

- We can realloc more memory for our array and update queueSize!
- But we also need to make sure we copy over the wraparound values correctly.

```c
struct queue {
    int *elements;
    int front;
    int numElements;
    int queueSize;
};
```
What if our `numElements == queueSize`?
- We can realloc more memory for our array and update `queueSize`!
- But we also need to make sure we copy over the wraparound values correctly.

```c
struct queue {
    int *elements;
    int front;
    int numElements;
    int queueSize;
};
```

```c
6 3 4 5
```

```c
say queueSize = 4
Enqueue(12);
```

```c
elements = (int *)realloc(elements, 2*queueSize*sizeof(int));
queueSize = 2*queueSize;
```

BUT where do front and back go? Does this look right?
So what we really need to do, is reset front = 0

And copy the elements accordingly:

In code we could do:

```c
struct queue {
    int *elements;
    int front;
    int numElements;
    int queueSize;
};

for (i=front, j=0; i<queueSize; i++, j++)
    temp[j] = values[i];
for (i=0; i<front; i++, j++)
    temp[j] = values[i];
```
Queues - Linked List Implementation

- We are going to need a linked list
  - So we’ll use the same node implementation as before.
- But we’ll need to keep track of the front and the back.
  - Otherwise either enqueue or dequeue would require an O(n) traversal each time.
- So we’ll keep a front and back pointer inside of a structure called queue.

```c
struct node {
    int data;
    struct node *next;
};

struct queue {
    struct node *front;
    struct node *back;
};
```
2 examples:

1) Checking if we have matching parentheses
2) Reading in a list of numbers from a user and printing it in backwards order.

// Either prints (1) More right paren’s than left, (2) More left paren’s than right, or (3) Paren’s are balanced
void ParenMatch();

void main(){
    printf("Give input expression without blanks: \n");
    char *InputExpression = malloc(100*sizeof(char));
    scanf("%s", InputExpression);
    ParenMatch(InputExpression);
}