Dynamic Array Queue and Deque
Queues

```
int isEmpty();
void addBack(TYPE val);  // Add value at end of queue.
TYPE front();            // Get value at front of queue.
void removeFront();      // Remove value at front.
```
Queue Applications

• Also good for ‘remembering’, just in a different order. We’ll revisit this when we study graphs and search!
• Operating systems – process scheduling
• Print Queue
Queue with Dynamic Array

- `int isEmpty();`
- `void addBack(TYPE val); // Add value at end of queue.
- `TYPE front(); // Get value at front of queue.
- `void removeFront(); // Remove value at front.

Removal from front is expensive!
void addFront(TYPE val);
void removeFront();
TYPE front();
void addBack(TYPE val);
void removeBack();
TYPE back();

Dequeue (Double Ended Queue) ADT

Can simulate a stack or queue using as deque!
Deque Application

- Finite Length Undo
Dynamic Array Deque

Adding to Front

Removing from Front

Big-Oh?

Big-Oh?
Let the partially filled block “float”

• One solution: Rather than always use index zero as our starting point, allow the starting index to “float”

• Maintain two integer values:
  – Starting or beginning index (beg)
  – Count of elements in the collection (size)

• Still need to reallocate when size equal to capacity
Dynamic Array Deque with Circular Buffer

- First filled element no longer always at index 0
- Filled elements may wrap around back to the front end of array
Problem: Elements can wrap around from beg to end

Index of front element of deque is just \texttt{beg}.

How can we compute the index of the last/back element?
Index of Back

• Calculate offset:

\[ \text{index} = \text{beg} + \text{size} - 1 \]

• If larger than or eq to capacity, subtract capacity

\[
\text{if (index} \geq \text{cap)} \\
\text{index} = \text{index} - \text{cap};
\]

• Or..combine into single statement with mod:

\[
/* \text{Convert logical index to absolute index.} */ \\
\text{index} = (\text{beg} + \text{size} - 1) \% \text{cap};
\]

beg = 6
cap = 8
size = 5
struct ArrDeque {
    TYPE *data;  /* Pointer to data array. */
    int size;    /* Number of elements in collection. */
    int beg;     /* Index of first element. */
    int cap;     /* Capacity of array. */
};

void initArrDeque(struct ArrDeque *d, int cap) {
    d->data = malloc(cap * sizeof(TYPE));
    assert(d->data != 0);
    d->size = d->beg = 0;
    d->cap = cap;
}
Adding/removing to/from back is easy, just adjust size

- Still need to reorganize if adding and size = capacity

**Add**

**Remove**
Adding/Removing from Front

Changes to front are easy, just adjust size and starting location

**Add**

![Diagram showing addition from the front]

**Remove**

![Diagram showing removal from the front]

What about when beg = 0?
# Operations Analysis

<table>
<thead>
<tr>
<th>Operation</th>
<th>Best</th>
<th>Worst</th>
<th>Ave</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddBack</td>
<td>1</td>
<td>n</td>
<td>1+</td>
</tr>
<tr>
<td>RemoveBack</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AddFront</td>
<td>1</td>
<td>n</td>
<td>1+</td>
</tr>
<tr>
<td>RemoveFront</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Your Turn...

• Complete Worksheet 20
// return back element of deque

TYPE back (struct deque * d) {
    int index = d->start + d->size - 1;
    if (index >= d->capacity) {
        index -= d->capacity
    }
    return d->data[index];
}
void removeBack (struct deque *d) {
  assert(d->size > 0);
  d->size--;
}
void addBack (struct deque * d, TYPE newValue) {
    int index;
    if (d->size >= d->capacity)
        _dequeSetCapacity(d, 2*d->capacity); // next slide
    index = d->start + d->size;
    if (index >= d->capacity)
        index -= d->capacity;
    d->data[index] = newValue;
    d->size++;
}
• Can we simply copy the elements to a larger array?
• Do this on your own in WS #20
Inserting & Removing from Front

Changes to front are easy, just adjust count and starting location

**Add**

**Remove**
Worksheet

• dequeFront
• dequeRemoveFront
• dequeAddFront

Do these on your own.