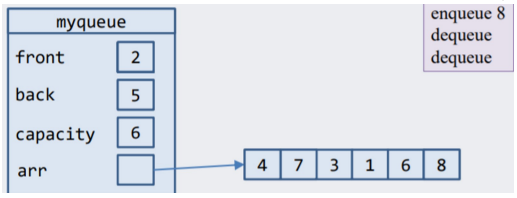
# Queues

October 23, 2019 8:14 AM

**Queues**

- satisfies required FIFO behavior, first in first out
- ex. printer jobs, CPU job scheduling, database requests
  - **enqueue:** insert item to back of queue
  - **dequeue:** remove item from front of queue
  - **peek:** return element at front of queue
  - **isEmpty:** is the queue empty
- **Implementation**
  - insertion happen at back of queue
  - if front is always index 0, need to shuffle with every dequeue O(n)
    - insertions will increment back index, removals will icrement fornt index





  - Use circular array to insert and remove items from queue in constant time
- **Modulo operator**
  - calculates remainders
    - ex. 1%5 = 1; 2%5 = 2, 5%5 = 0, 8%5 = 3
  - can be used to calculate front and back positions in circular array
    - avoiding comparisons to array size
    - back of queue: (front + num)%capacity
    - front of queue (after removing item): (front+1)%capacity
- **Queue implementation**

```
typedef struct {
  int front;
  int num;
  int capacity;
  int* arr;
} Queue;
```

```
void initializeQueue(Queue* q) {
  q->front = 0;
  q->num = 0;
  q->capacity = 6; // or some other value
  q->arr = (int*) malloc(q->capacity * sizeof(int));
}
```

```
int isEmpty(Queue* q) {
  if (q->num == 0)
    return TRUE;
  else
    return FALSE;
}
```

```
int isFull(Queue* q) {
  if (q->num == q->capacity)
    return TRUE;
  else
    return FALSE;
}
```

*void destroyQueue(Queue* q) {*
*free(q->arr);*
*}*

```
int enqueue(Queue* q, int val) {
  if (isFull(q))
    return FALSE;
  else {
    q->arr[(q->front + q->num) % q->capacity] = val;
    q->num++;
    return TRUE;
  }
}
```

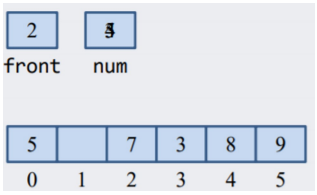
```
int dequeue(Queue* q) {
```

```
int dequeue(Queue* q) {
  if (isEmpty(q))
    return FALSE;
  else {
    q->arr[q->front] = -1;
    q->front = (q->front + 1) % q->capacity;
    q->num--;
    return TRUE;
  }
}
```

- does not let enqueue more elements once full
- enqueue is possible as long as array is not full

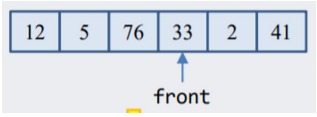
- ex.

```
Queue myq;
initQueue(&myq);
enqueue(&myq, 6);
enqueue(&myq, 4);
enqueue(&myq, 7);
enqueue(&myq, 3);
enqueue(&myq, 8);
dequeue(&myq);
dequeue(&myq);
enqueue(&myq, 9);
enqueue(&myq, 5);
```




**Array queue resizing**

- perform more enqueues to fill array
- How should we resize the array to allow for more enqueue operations
- resize the same way as stack, copy elements down, but may have gaps in in array




**Linked list and Queue implementation**

- use a back pointer to avoid traversing

```
typedef struct {
  struct Node* front;
  struct Node* back;
  int num;
} Queue;
```

```
void initializeQueue(Queue* q) {
  q->front = NULL;
  q->back = NULL;
  q->num = 0;
}
```

```
struct Node {
  int data;
  struct Node* next;
};
```

```
int isEmpty(Queue* q) {
  if (q->front == NULL)
    return TRUE;
  else
    return FALSE;
}
```