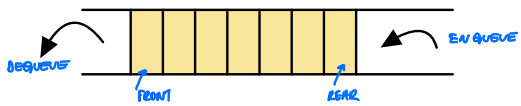
QUEUE

- A QUEUE IS A LIST FROM WHICH ITEMS ARE DELETED FROM ONE END (FRONT) AND INTO WHICH ITEMS ARE INSERTED AT THE OTHER END (REAR, OR BACK)



FIRST IN FIRST OUT (FIFO)

- APPLICATIONS: ANY APPLICATION WHERE A GROUP OF ITEMS IS WAITING TO USE A SHARED RESOURCE WILL USE A QUEUE.
  - JOBS IN A SINGLE PROCESSOR COMPUTER
  - PRINT SPOOLING
  - INFORMATION PACKETS IN COMPUTER NETWORKS

- QUEUE ADT

Queue
+isEmpty(): boolean +enqueue (newEntry: T): boolean +dequeue (): boolean +peekFront(): T

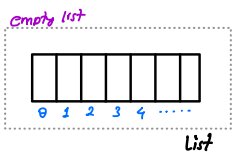
- SOME OPERATIONS

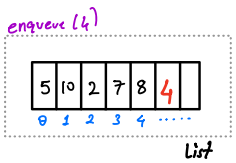
OPERATION	QUEUE AFTER OPERATION
Queue q	empty queue
q.enqueue(5)	front 5 back ← enqueue
q.enqueue(3)	front 5 3 back ← enqueue
q.enqueue(2)	front 5 3 2 back ← enqueue
q.peekFront()	front 5 3 2 back — 5 IS AT THE FRONT
q.dequeue()	dequeue ← 5, front 3 2 back
q.dequeue()	dequeue ← 3, front 2 back

QUEUE IMPLEMENTATIONS

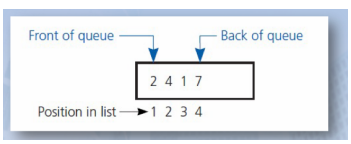
1. QUEUE IMPLEMENTATION USING ADT LIST (ARRAYLIST OR LINKED LIST)
2. LINKED-BASED IMPLEMENTATION
3. ARRAY-BASED IMPLEMENTATION

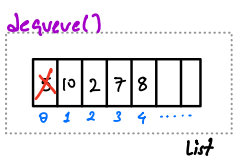
1 QUEUE IMPLEMENTATION USING ADT LIST (ARRAYLIST OR LINKED LIST)





list->insert(list->getLength()+1, 4)





list->remove(1);

READ CHAPTER INTERLUDE-4 FOR MORE INFORMATION ABOUT SMART POINTERS

SMART POINTER → ACT LIKE RAW POINTERS
→ PROVIDE AUTOMATIC MEMORY MANAGEMENT

```
#ifndef LIST_QUEUE_H
#define LIST_QUEUE_H

#include "QueueInterface.h"
#include "ArrayList.h"
#include <memory>

template <class T>
class ListQueue: public QueueInterface<T>{
    private:
        std::unique_ptr<ArrayList<T>> listPtr;      //unique smart pointer to list of queue items
    public:
        ListQueue();
        ListQueue(const ListQueue<T>& other);
        ~ListQueue();
        bool isEmpty() const;
        bool enqueue(const T& newEntry);
        bool dequeue();

        T peekFront() const throw(PrecondViolatedException);
};
```

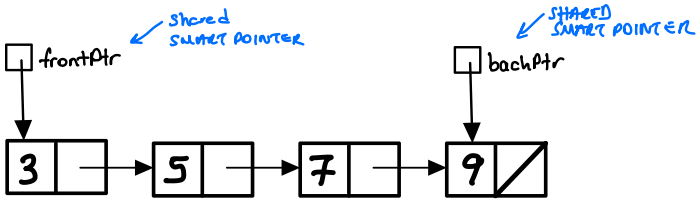
```
template<class T>
ListQueue<T>::ListQueue():listPtr(std::make_unique<ArrayList<T>>())
{
}

template <class T>
ListQueue<T>::ListQueue(const ListQueue<T>& other):listPtr(other.listPtr)
{
}

template <class T>
ListQueue<T>::~ListQueue()
{
}

template <class T>
bool ListQueue<T>::isEmpty() const{
    return listPtr->isEmpty();
}
```

MAKES UNIQUE SMART POINTER

NO OTHER POINTER CAN REFERENCE SAME OBJECT

```
template <class T>
bool ListQueue<T>::isEmpty() const{
    return listPtr->isEmpty();
}

template <class T>
bool ListQueue<T>::enqueue(const T& newEntry){
    return listPtr->insert(listPtr->getLength() + 1, newEntry);
}

template <class T>
bool ListQueue<T>::dequeue(){
    return listPtr->remove(1);
}

template <class T>
T ListQueue<T>::peekFront() const throw(PrecondViolatedException)
{
    if(isEmpty())
        throw PrecondViolatedException("peekFront() called with empty queue");

    return listPtr->getEntry(1);
}


#endif
```

2 LINKED-BASED IMPLEMENTATION



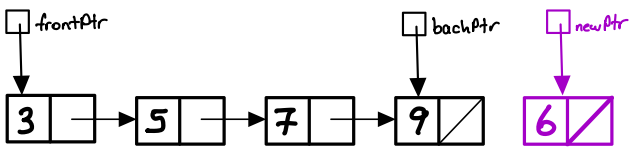

```
template <class T>
class LinkedQueue: public QueueInterface<T> {
    private:
        std::shared_ptr<NodeType> frontPtr;
        std::shared_ptr<NodeType> backPtr;
        .
        .
        .
```

SMART POINTERS

```
template <class T>
class LinkedQueue: public QueueInterface<T> {
    private:
        Node<T>* frontPtr;
        Node<T>* backPtr;
        .
        .
        .
```

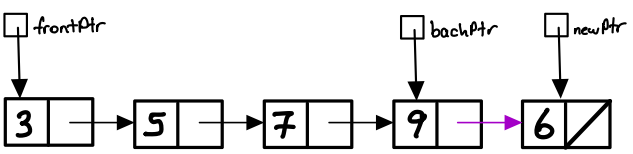
RAW POINTERS

enqueue(6)

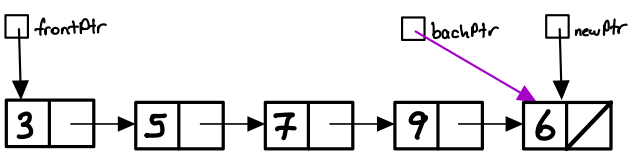


CREATE A NEW NODE
MAKE IT A SHARED POINTER

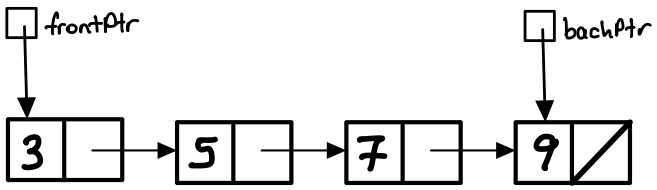
```
newPtr = std::make_shared<Node<T>>(entry)
```



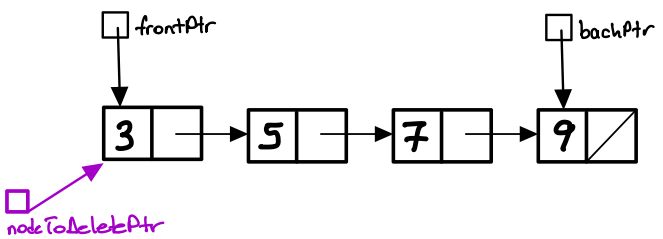

```
backPtr->setNext(newPtr)
```



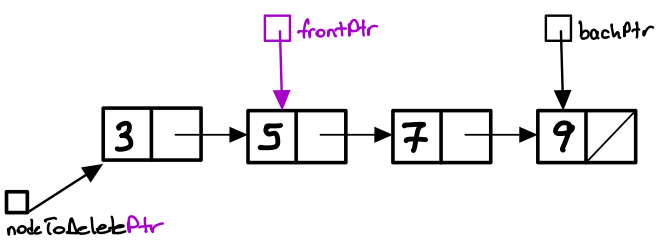

```
backPtr = backPtr->getNext()
```



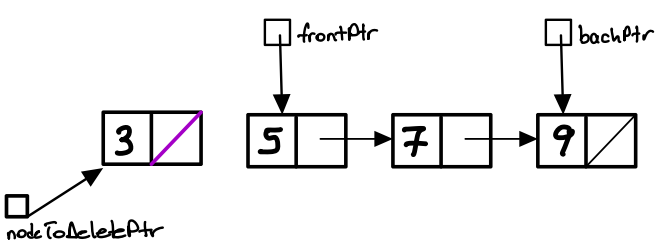
dequeue()



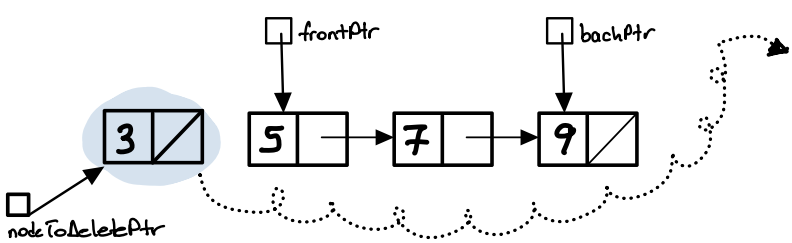
nodeToDeletePtr = frontPtr



frontPtr = frontPtr->getNext()



nodeToDeletePtr.setNext(nullptr)



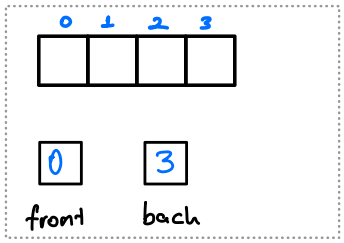
REMOVED NODE WILL BE DEALLOCATED WHEN METHOD ENDS SINCE WE USE SMART POINTERS. OTHERWISE, WE WOULD NEED TO DELETE THE NODE.

3 ARRAY-BASED IMPLEMENTATION

EX: CAPACITY = 4

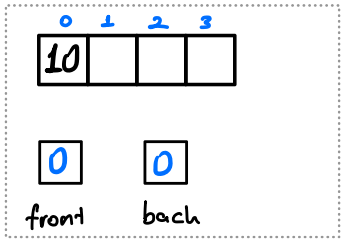
EMPTY QUEUE

1)



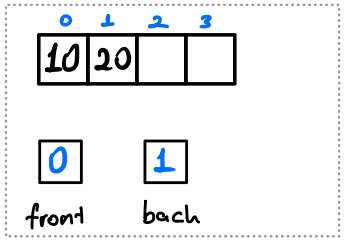
enqueue(10)

2)



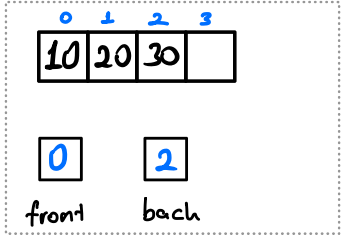
enqueue(20)

3)



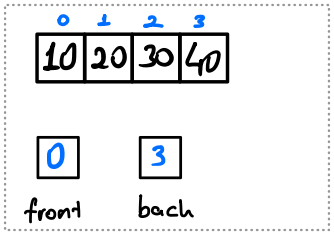
enqueue(30)

4)



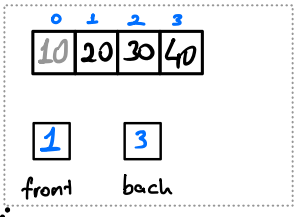
enqueue(40)

5)



dequeue()

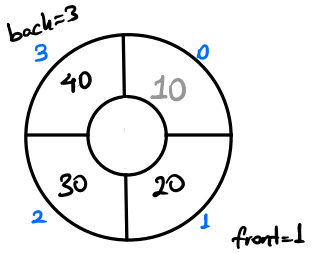
6)



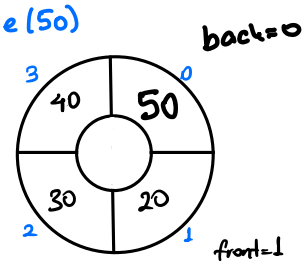
RIGHT DRIFT CAN CAUSE THE QUEUE TO APPEAR FULL EVEN THOUGH THE QUEUE CONTAIN FEW ENTRIES

WE MAY SHIFT THE ELEMENTS TO LEFT, BUT SHIFTING IS EXPENSIVE

SOLUTION : A CIRCULAR ARRAY



enqueue(50)

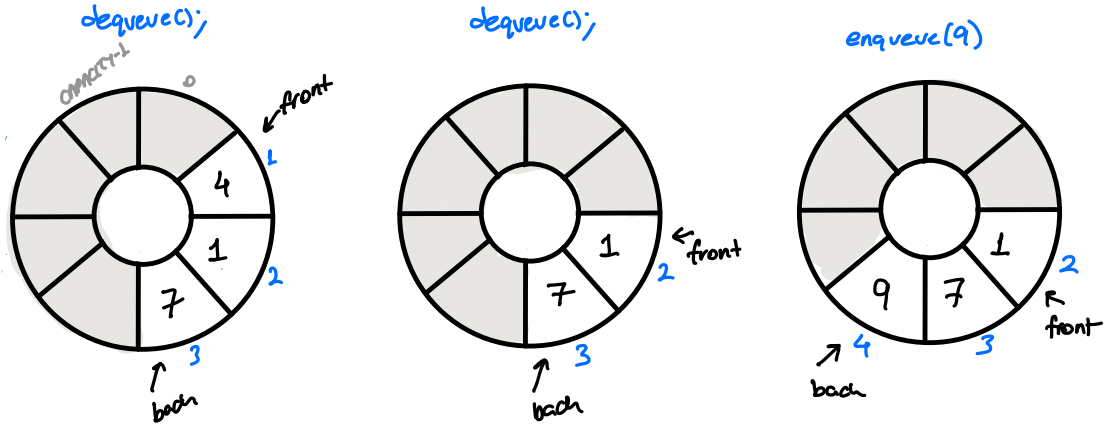


((WHEN EITHER FRONT OR BACK ADVANCES PAST CAPACITY-1, IT WRAPS AROUND 0.))

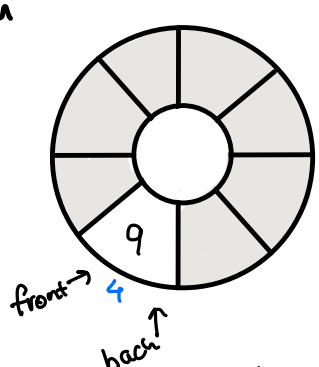
INITIALIZE → FRONT=0 BACK=CAPACITY-1

INSERT → (BACK+1) % CAPACITY

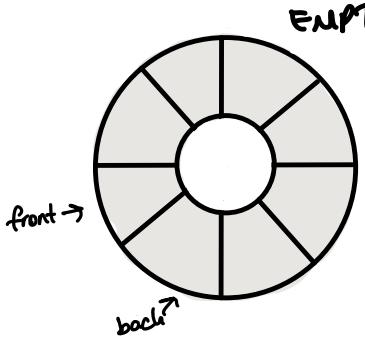
DELETE → (FRONT+1) % CAPACITY



QUEUE WITH SINGLE ITEM



QUEUE BECOMES EMPTY



// WE CANNOT USE FRONT AND BACK TO DISTINGUISH BETWEEN QUEUE-FULL AND QUEUE-EMPTY CONDITIONS //

↓

SOLUTIONS

USING A COUNTER

count == 0 ⇒ EMPTY

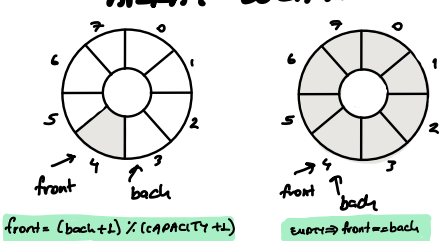
count == CAPACITY ⇒ FULL

USE BOOLEAN FLAGS

isFULL

isEMPTY

USE AN EXTRA ARRAY LOCATION



```cpp
#ifndef ARRAY_QUEUE
#define ARRAY_QUEUE

#include "QueueInterface.h"
#include "PreCondVialotedException.h"

template <class T>
class ArrayQueue:public QueueInterface<T>{
    private:
        static const int DEFAULT_CAPACITY = 50;
        T items[DEFAULT_CAPACITY];
        int front;
        int back;
        int count;
    public:
        ArrayQueue();
        bool isEmpty() const;
        bool enqueue(const T& newEntry);
        bool dequeue();
        T peekFront() const throw(PrecondViolatedException);
};
template <class T>
ArrayQueue<T>::ArrayQueue():front(0), back(DEFAULT_CAPACITY-1), count(0)
{
}

template <class T>
bool ArrayQueue<T>::isEmpty() const{
    return count == 0;
}

template <class T>
bool ArrayQueue<T>::enqueue(const T& item){
    bool isAbleToEnqueue = false;
    if(count < DEFAULT_CAPACITY){
        back = (back + 1) % DEFAULT_CAPACITY;
        count++;    → items[back]=item;
        isAbleToEnqueue = true;
    }
    return isAbleToEnqueue;
}
```

```cpp
template <class T>
bool ArrayQueue<T>::dequeue(){
    bool isAbleToDequeue = false;
    if(!isEmpty()){
        front = (front + 1) % DEFAULT_CAPACITY;
        count--;
        isAbleToDequeue = true;
    }

    return isAbleToDequeue;
}

template <class T>
T ArrayQueue<T>::peekFront() const{
    if(isEmpty()){
        throw PrecondViolatedException("peekFront() is called with an empty queue");
    }
    return items[front];
}

#endif
```

throw (PreconditionViolatedException)

EXAMPE: TO RECOGNIZE A PALINDROME, A QUEUE CAN BE USED IN CONJUNCTION WITH A STACK.

```
bool isPalindrome(string str){

    ArrayQueue<char> q;
    ArrayStack<char> s;

    for(int i=0; i<str.length(); i++){
        s.push(str.at(i));
        q.enqueue(str.at(i));
    }
    while(!q.isEmpty()){
        char item1 = q.peekFront();
        char item2 = s.peek();
        if(item1 != item2)
            return false;
        q.dequeue();
        s.pop();
    }

    return true;

}
```