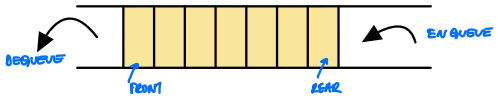
QUEUE

- 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

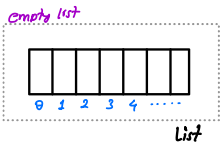
- SOME OPERATIONS

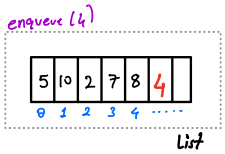
OPERATION	QUEUE AFTER OPERATION
Queue q	empty queue
q.enqueue(5)	5 ← enqueue (FRONT, BACK)
q.enqueue(3)	5 3 ← enqueue (FRONT, BACK)
q.enqueue(2)	5 3 2 ← enqueue (FRONT, BACK)
q.peekFront()	5 3 2 (FRONT, BACK) — 5 IS AT THE FRONT
q.dequeue()	dequeue ← 5, 3 2 (FRONT, BACK)
q.dequeue()	dequeue ← 3, 2 (FRONT,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

QUEUE IMPLEMENTATIONS

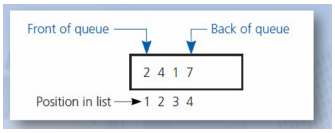
1. QUEUE IMPLEMENTATION USING ADT LIST (ARRAYLIST OR LINKED LIST)
2. LINKED-BASED IMPLEMENTATION
3. ARRAY-BASED IMPLEMENTATION

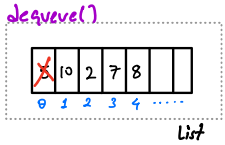
1 QUEUE IMPLEMENTATION USING ADT LIST (ARRAYLIST OR LINKED LIST)





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, REDUCE 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
```