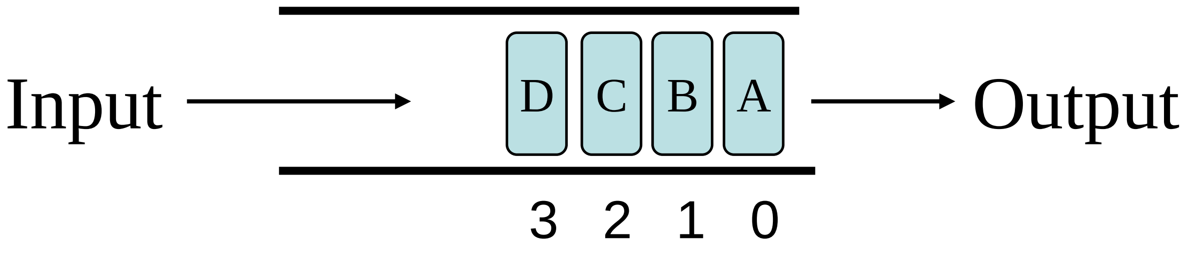
Data Structure

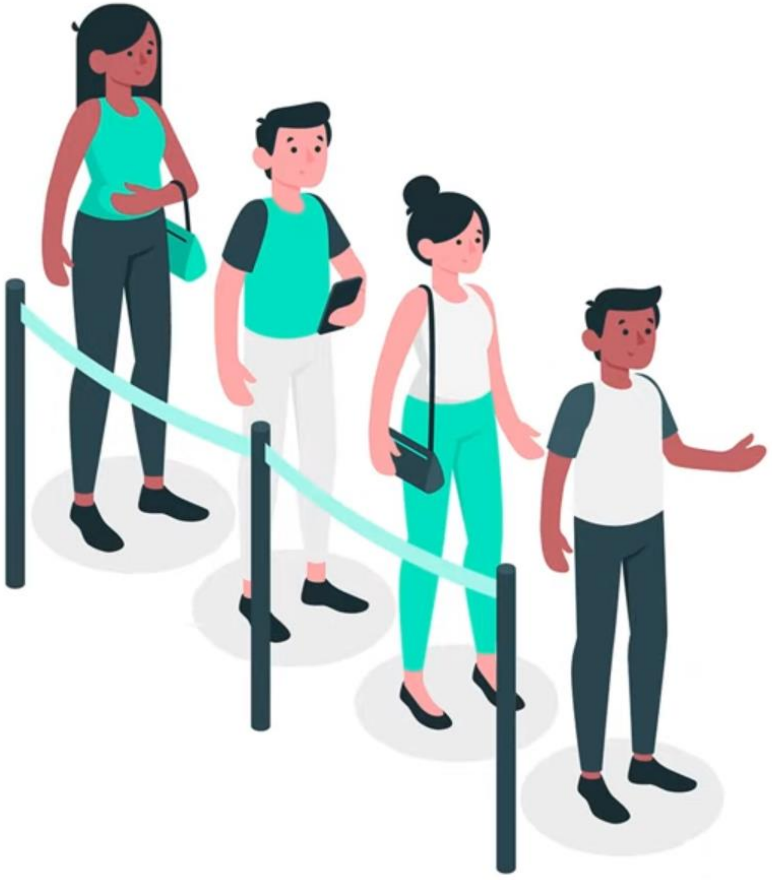
Lecture 4

Dr. Ahmed Fathalla

Motivation: Why `Queue`? *First In First Out (FIFO)*

- In a queue,
  - **new** values are always **added** at the tail of the list
  - values are **removed** from the opposite end of the list, the front

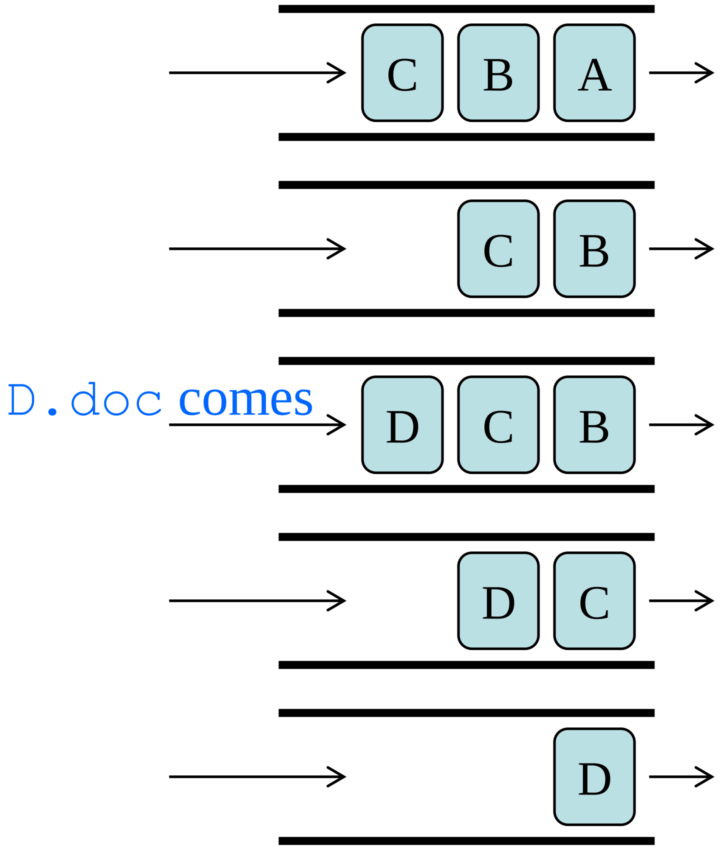




- **Examples of queues**
  - Checkout at supermarket
  - Toll Station
    - Car comes, pays, leaves
  - Check-out in Big super market
    - Customer comes, checks out and leaves
  - More examples: Printer, Office Hours, …

E.g., Printing Queue

- `A.doc, B.doc, C.doc` arrive to printer.



Now printing A.doc

A.doc is finished. Now printing B.doc

Now still printing B.doc

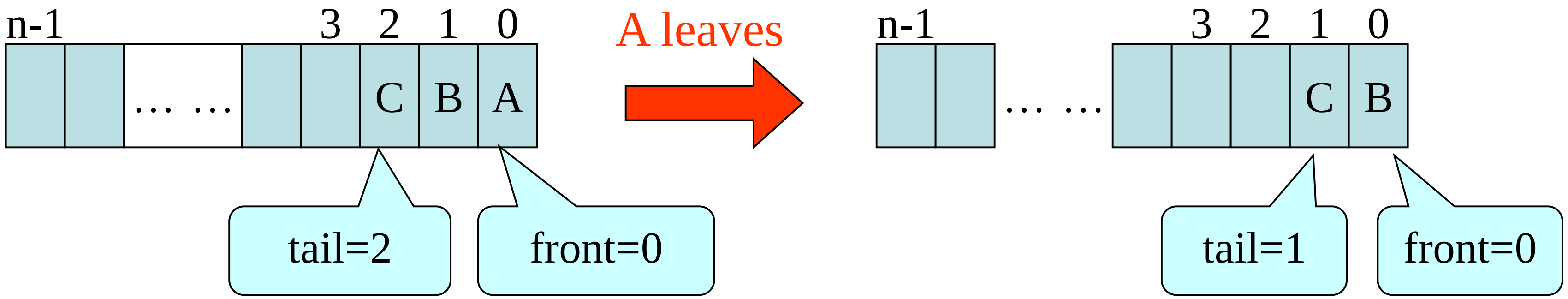
B.doc is finished. Now printing C.doc

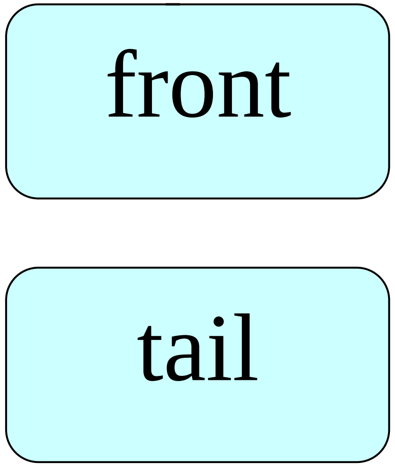
C.doc is finished. Now printing D.doc

Array Implementation1: Physical Model (Front is fixed as in physical lines)

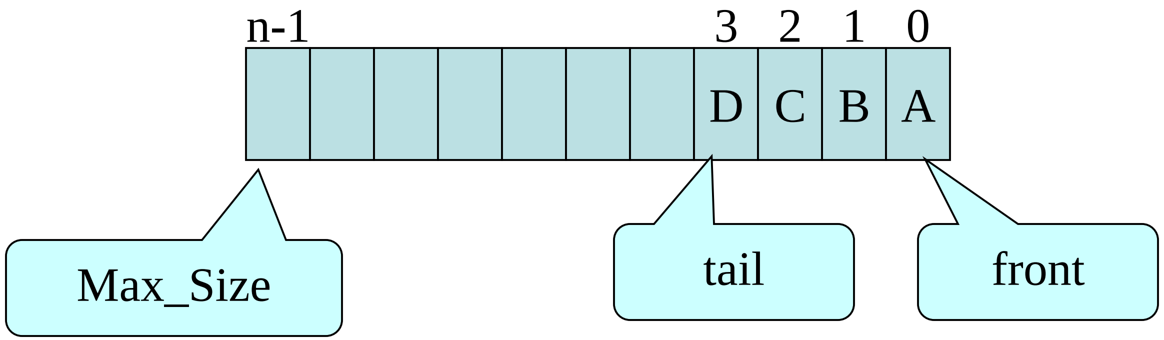
- Shifting all items to front in the array when **dequeue** operation. (Too Costly…)

- Why this was not a problem in the array implementation of Stacks?

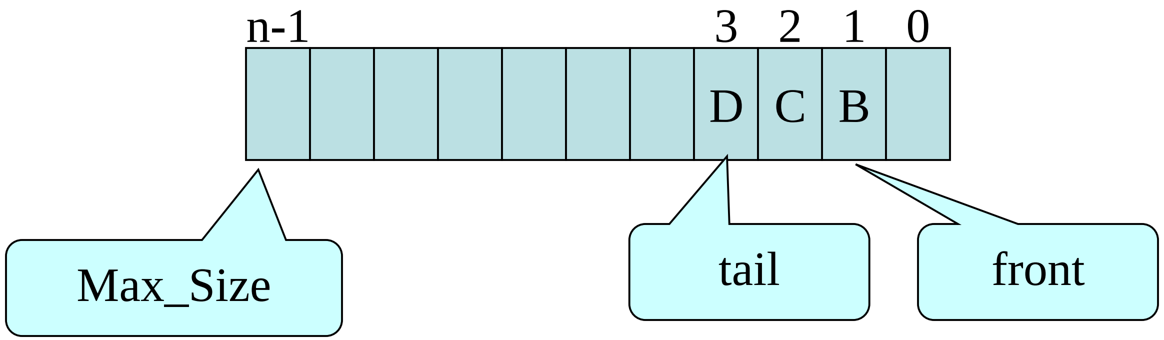




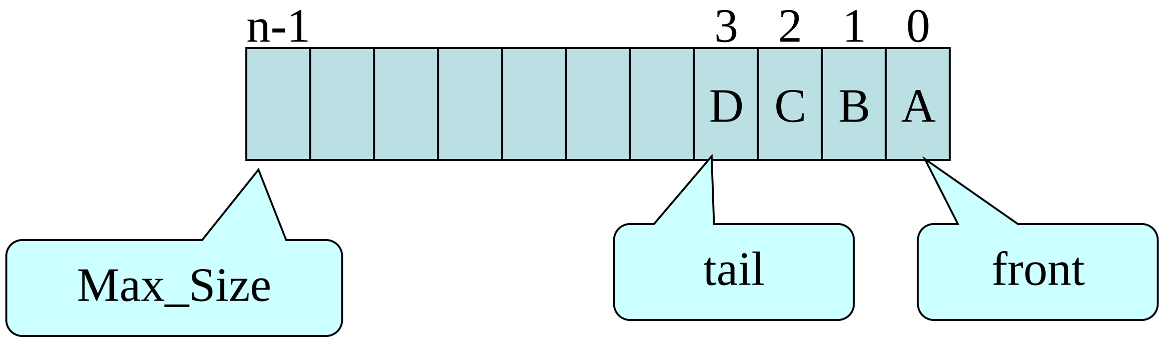
Array Implementation2: Linear Model (Two indices, front and tail)



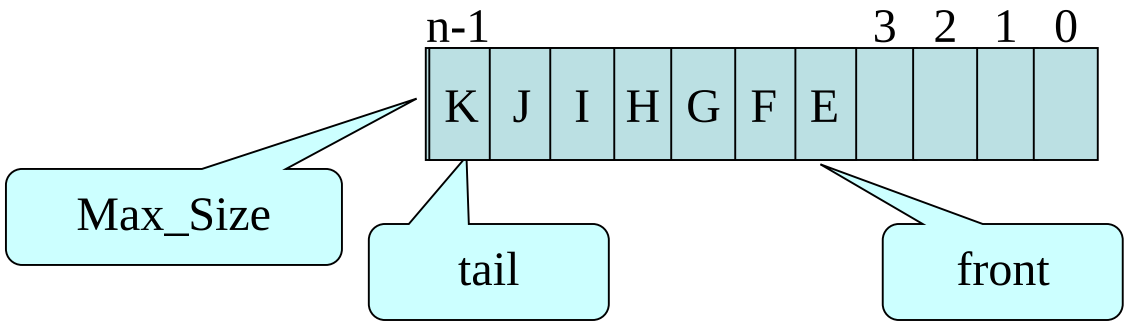
After A leaves,



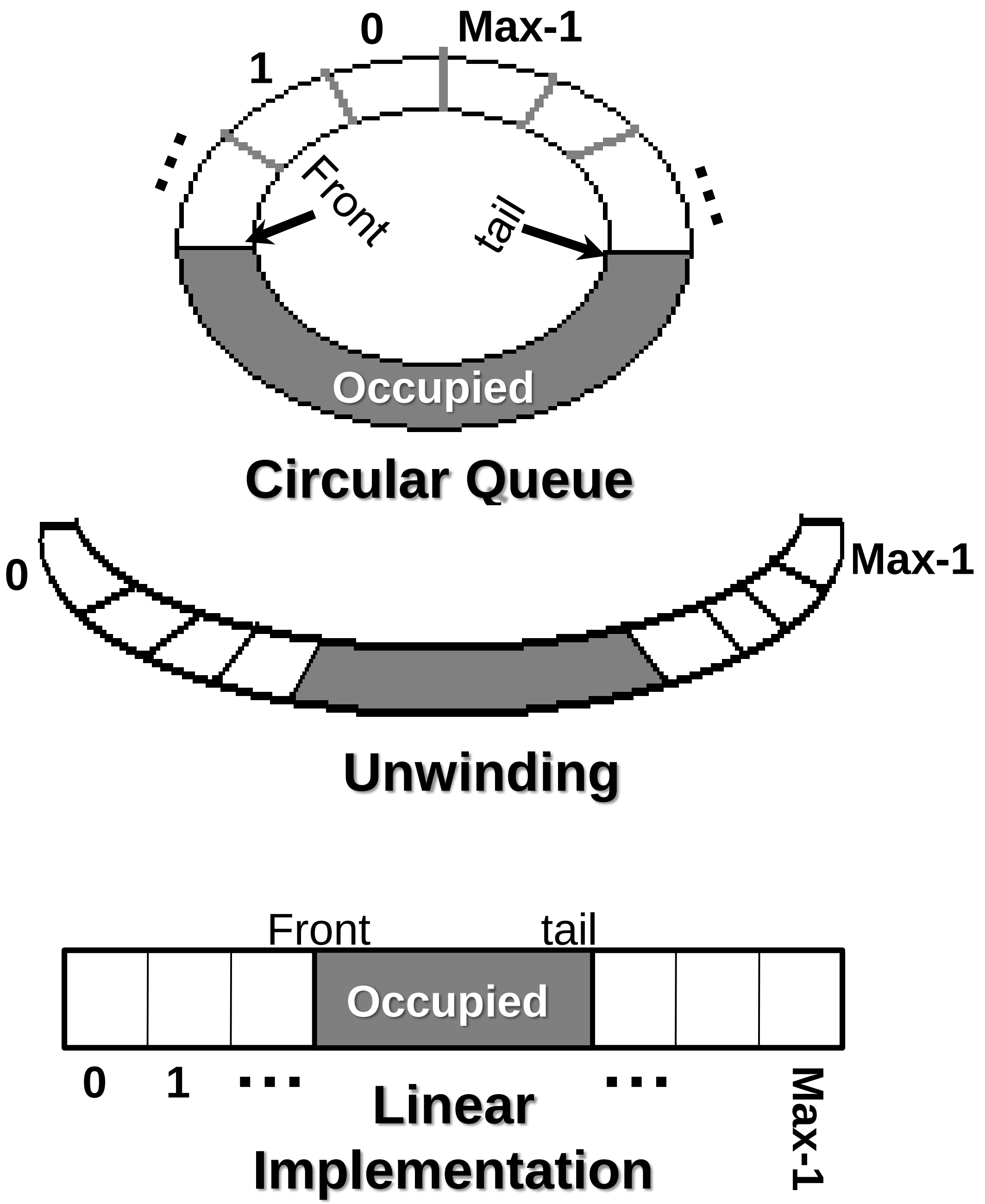
The problem is that there will be:
- Many **empty places** in the front of array, and
- tail is always incremented.



After A,B,C,D leave

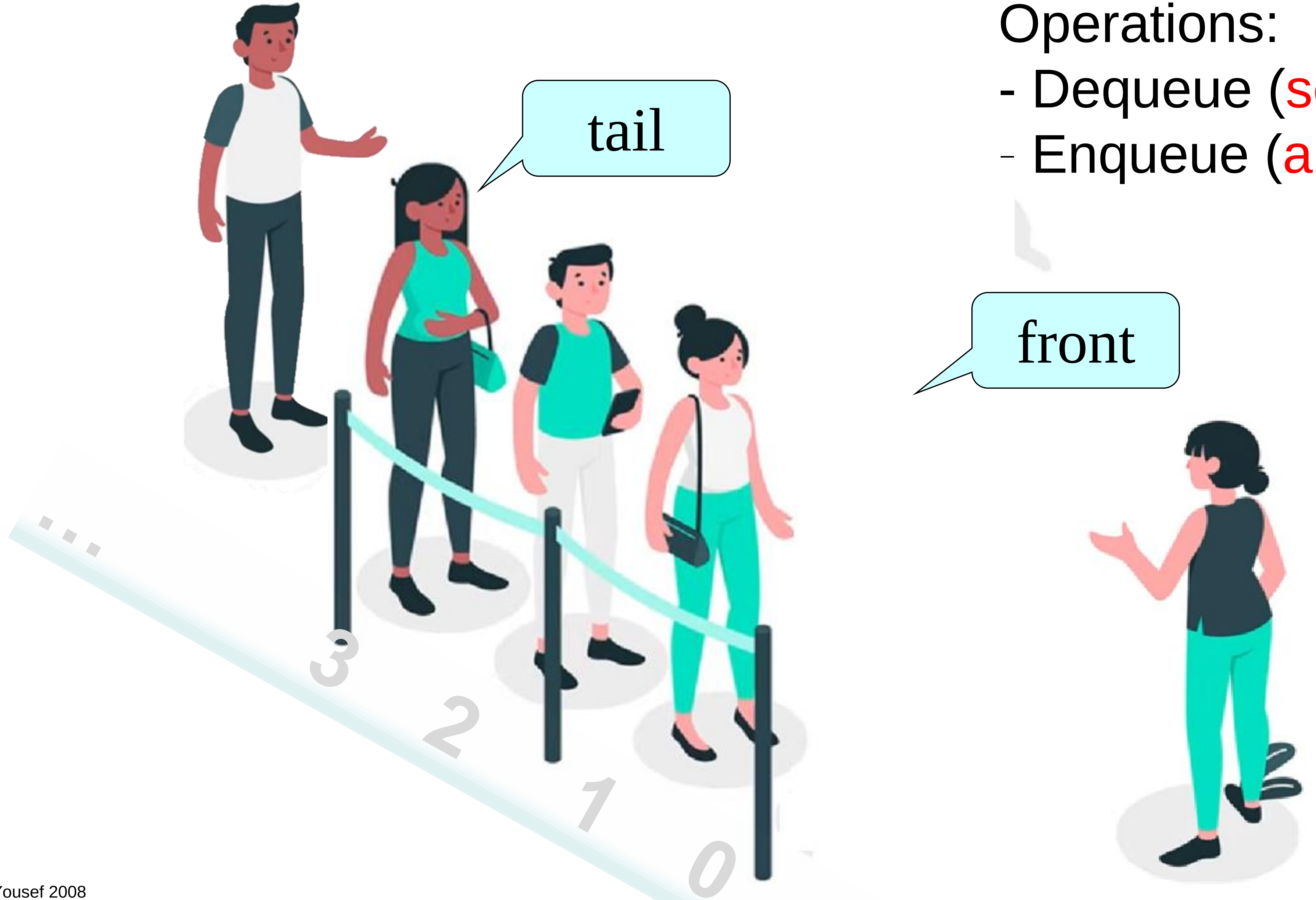


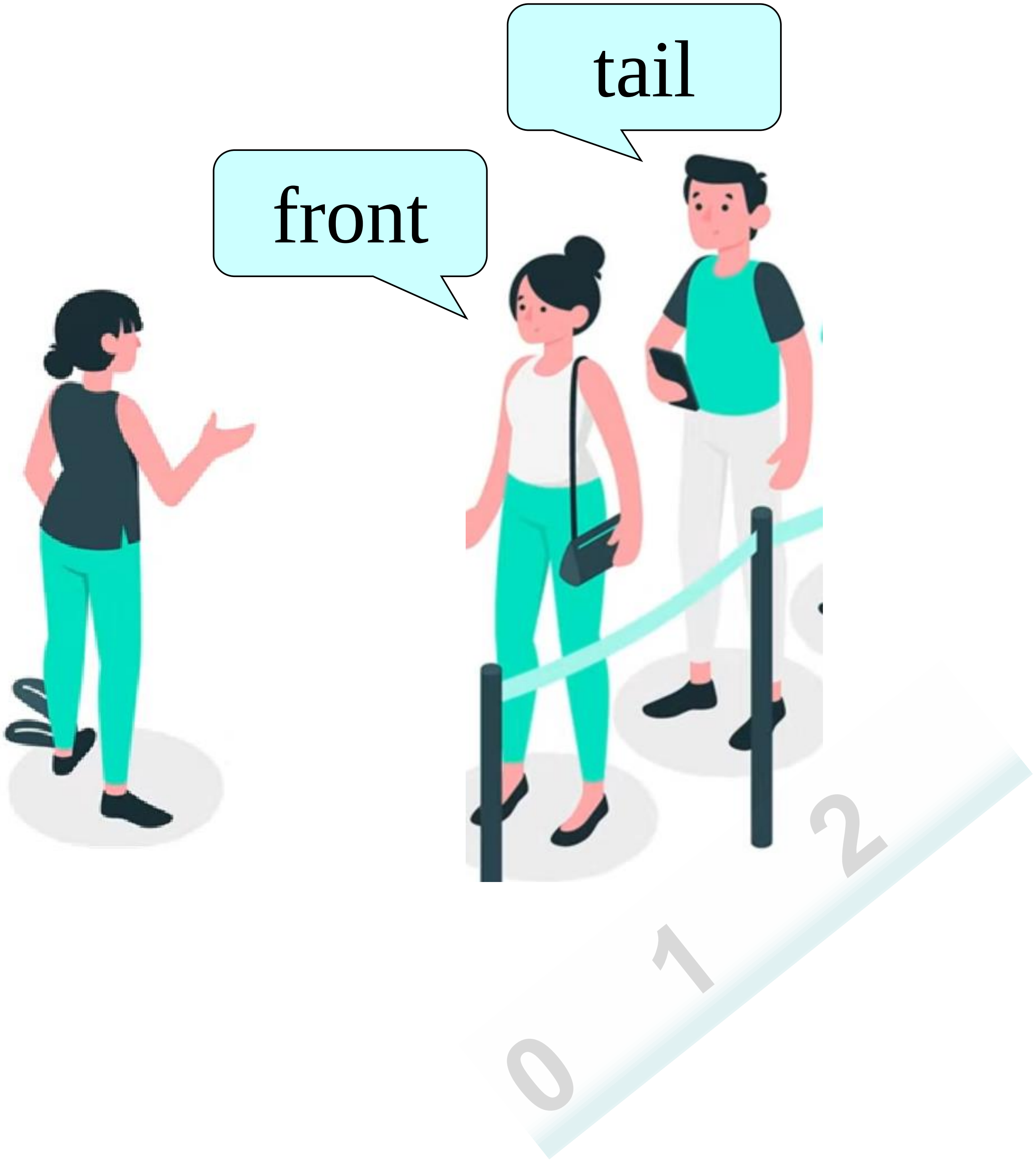
Array Implementation 3: Circular Implementation



Operations:

- Dequeue (serving)
- Enqueue (append)




tail
front
0
1
2

Checking the Boundary conditions

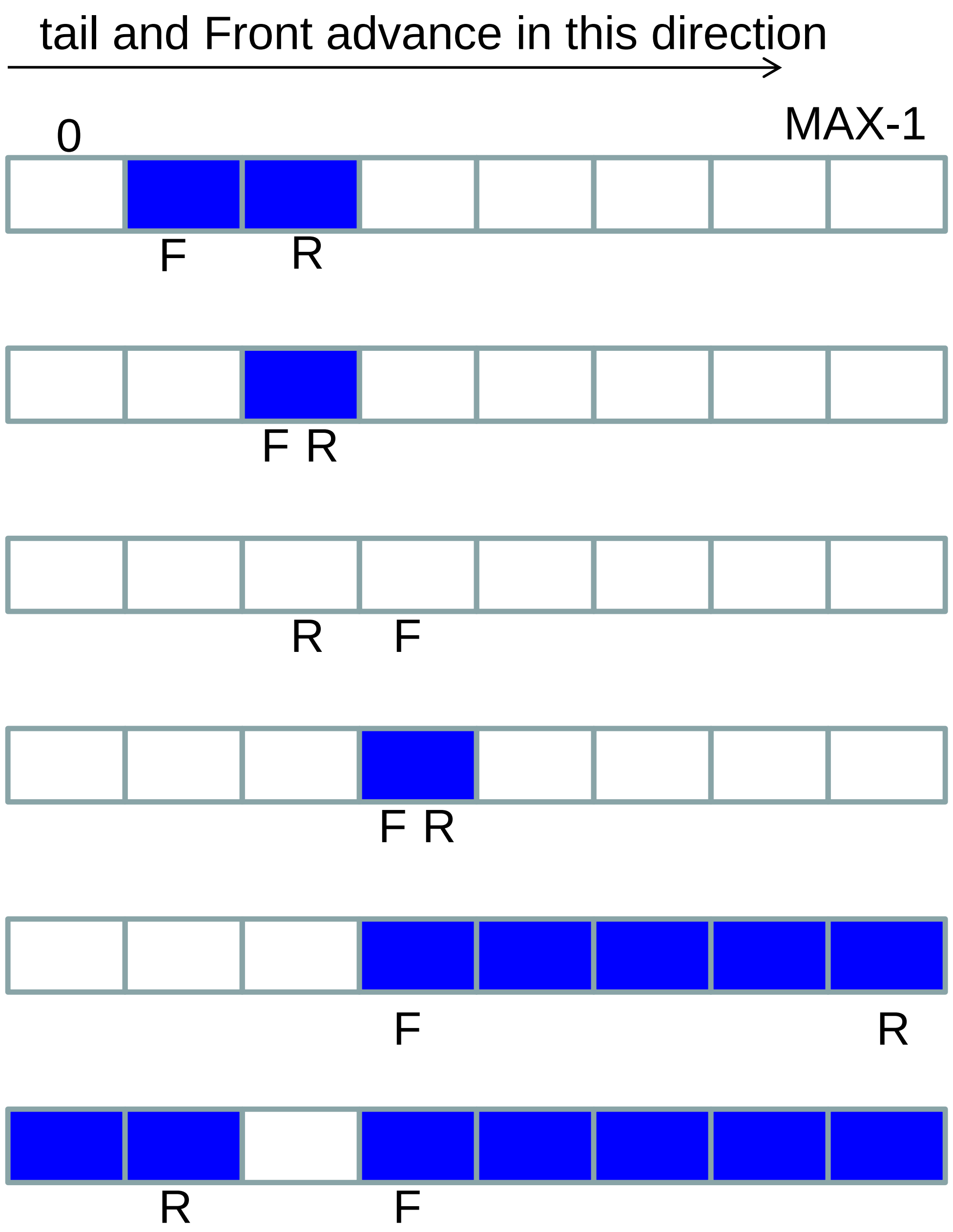
Queue with two elements

Queue with one element

With Empty condition:
Next to tail = Front.

Queue with one element

Full Condition:
Next to tail = Front
Therefore we waste one location

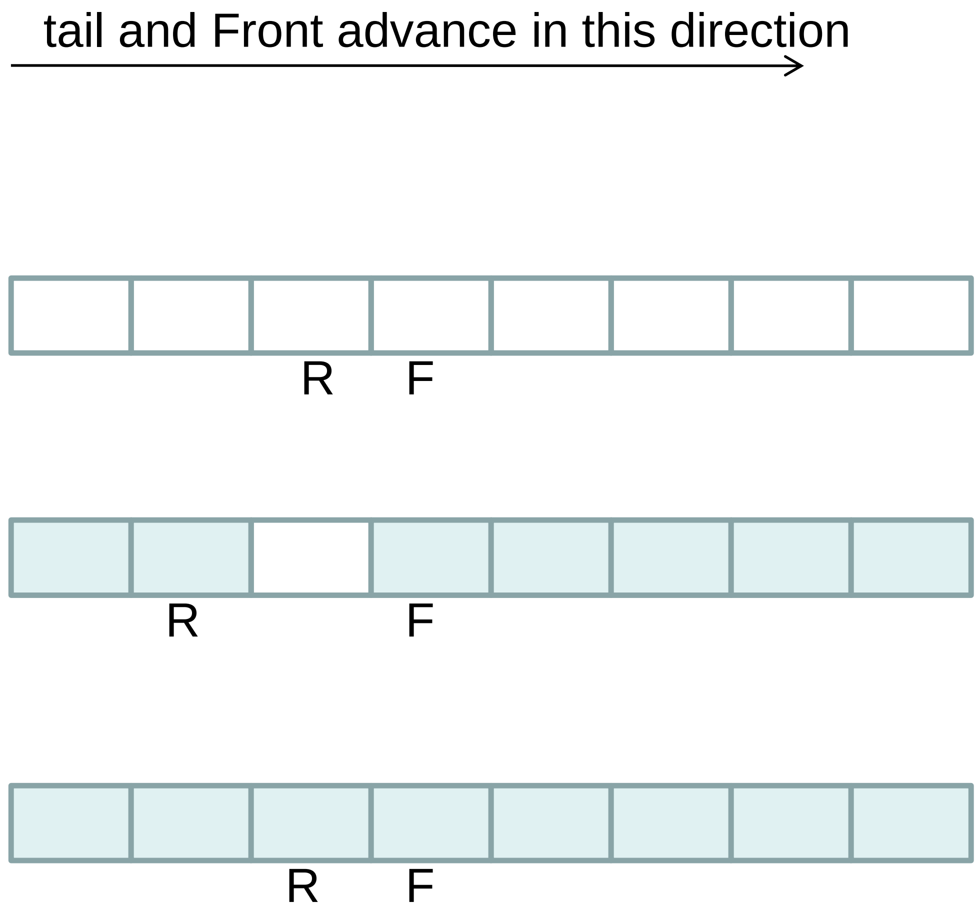


Circular implementation

Better solution: Use indicator variable to distinguish between Empty and Full conditions.

Empty queue:
Next to tail = Front.
Size=0

Full queue:
Next to tail = Front.
Size=MAX

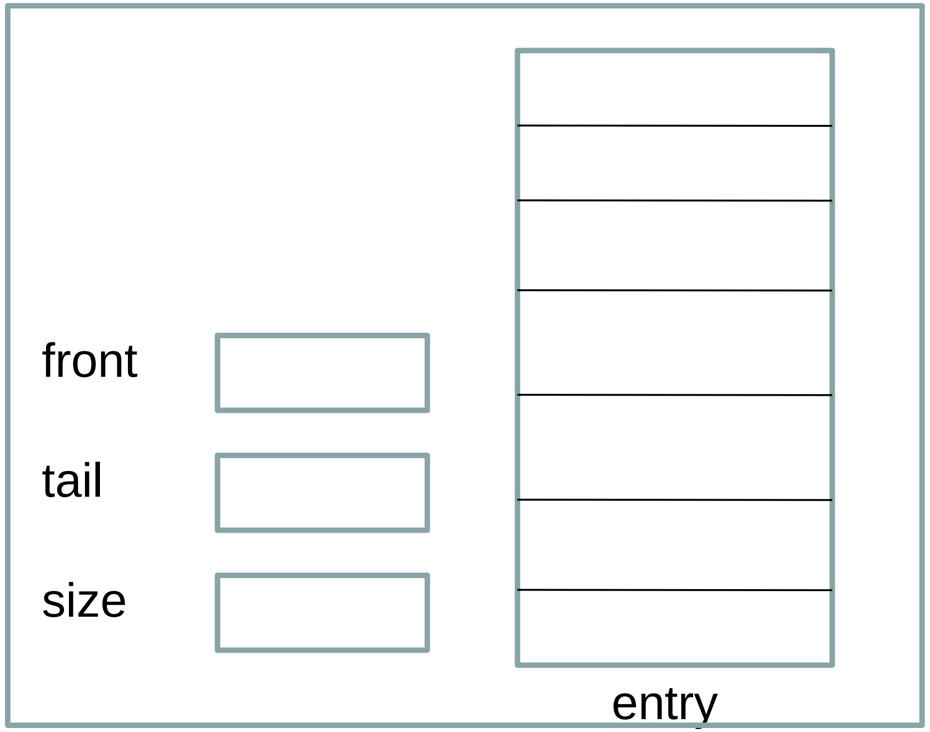


<u>Definition:</u> A ***queue*** of elements of type T is a finite sequence of elements of T together with the following operations:

1. **Create** the queue, leaving it empty.
2. Determine whether the queue is **empty** or not
3. Determine whether the queue is **full** or not
4. Find the **size** of the queue
5. **Append** (**Enqueue**) a new entry onto the top of the queue, provided the queue is not full.
6. **Serve** (**dequeue**) (and remove) the front entry from the queue, provided the queue is not empty.
7. **Traverse** the queue, visiting each entry
8. **Clear** the queue to make it empty

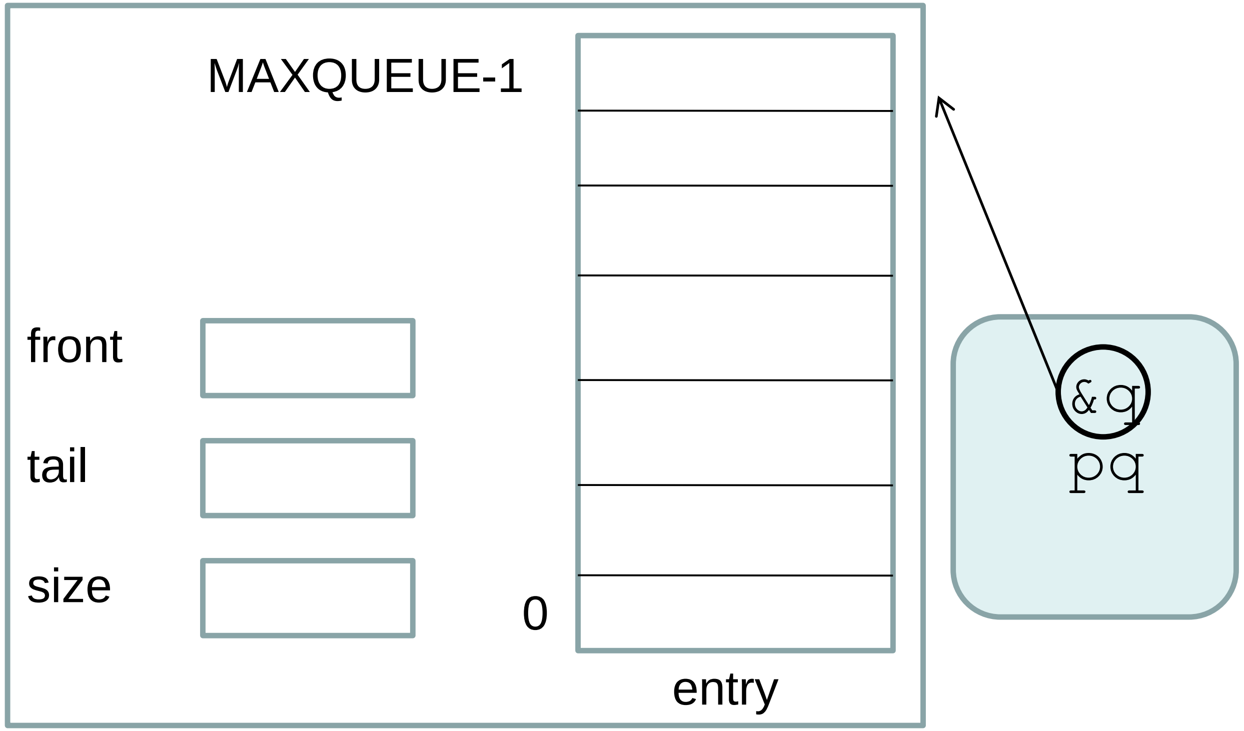
User Level (interface)

```
struct Queue{
    int front;
    int tail;
    int size;
    QueueEntry entry[MAXQUEUE];
};
```




```
void main(){

Queue q;


}
```

```c
void CreateQueue(Queue *pq){
    pq->front= 0;
    pq->tail = -1;
    pq->size = 0;
}
//Initializing front =5 and tail
=4 will work if MAXQUEUE >=6. But,
since MAXQUEUE can be 1 we
intialize as above.
```

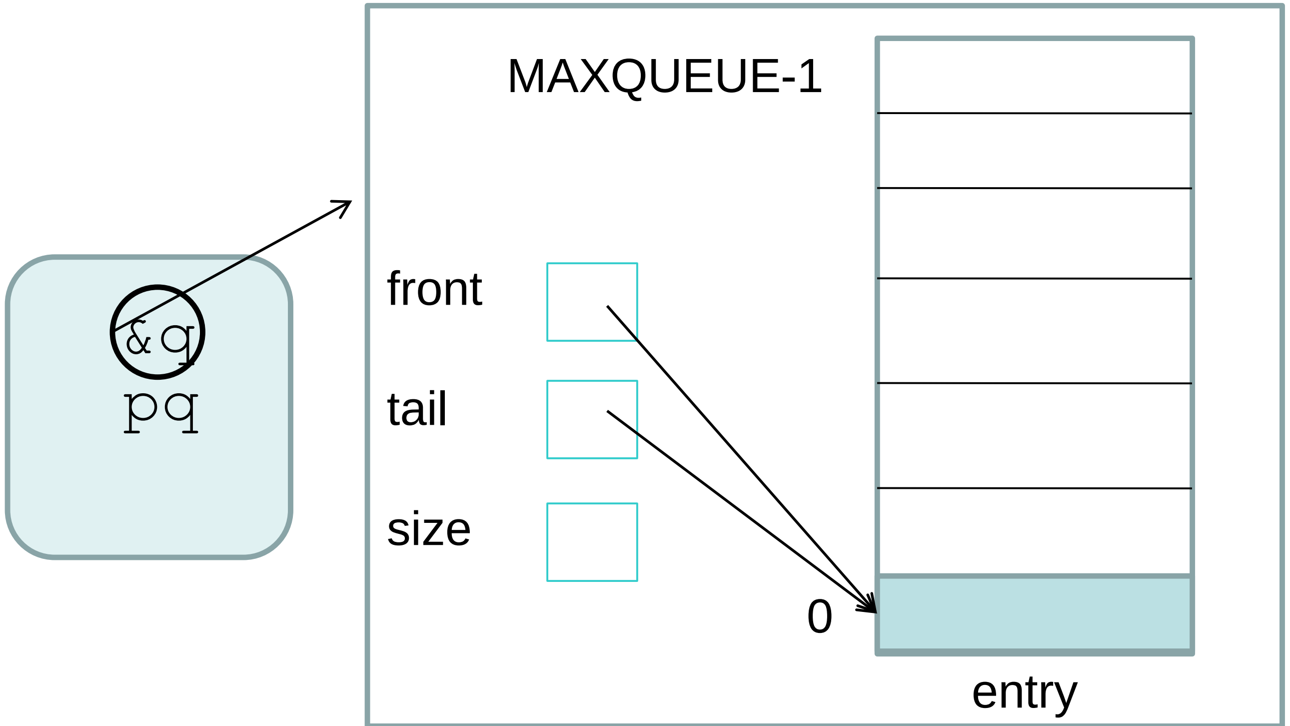


User Level (interface)

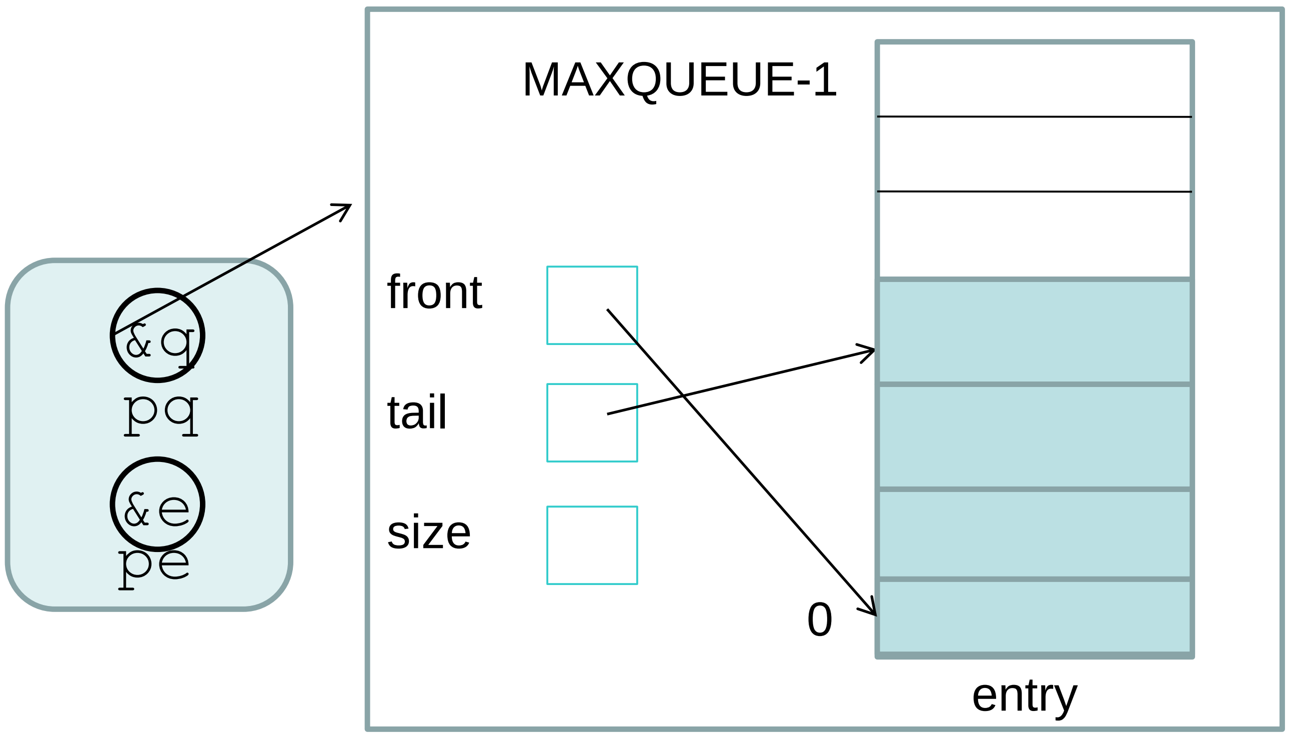
```c
void main(){

Queue q;

CreateQueue(&q);


}
```

```
void Append(QueueEntry e, Queue* pq){
  pq->tail = (pq->tail + 1) % MAXQUEUE;
  pq->entry[pq->tail] = e;
  pq->size++;
}
```

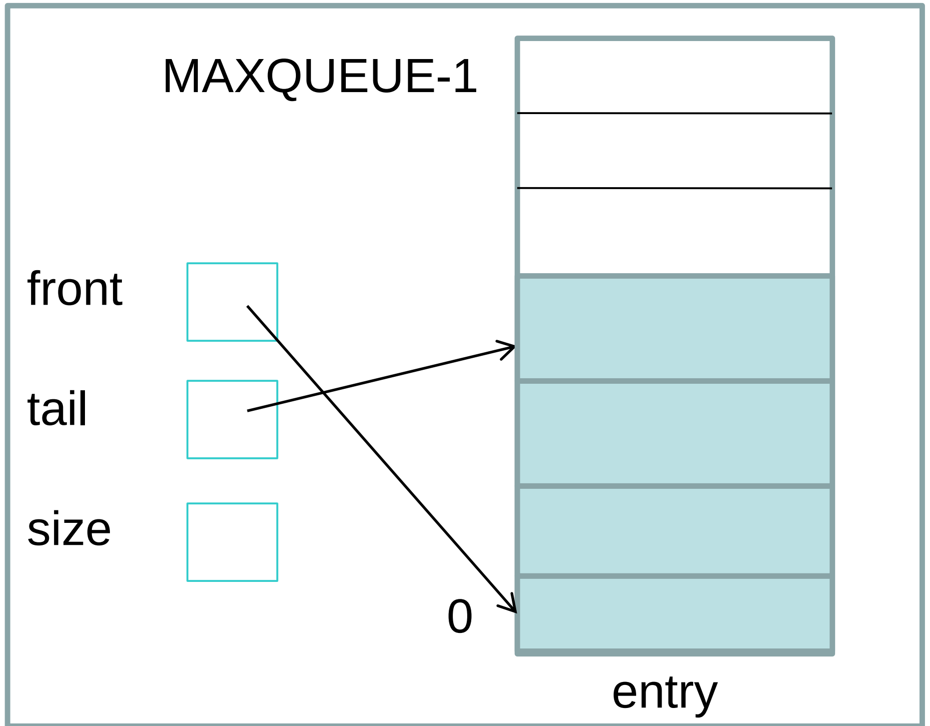
```
if (pq->tail == MAXQUEUE-1)
        pq->tail=0;
else
        pq->tail++;
```



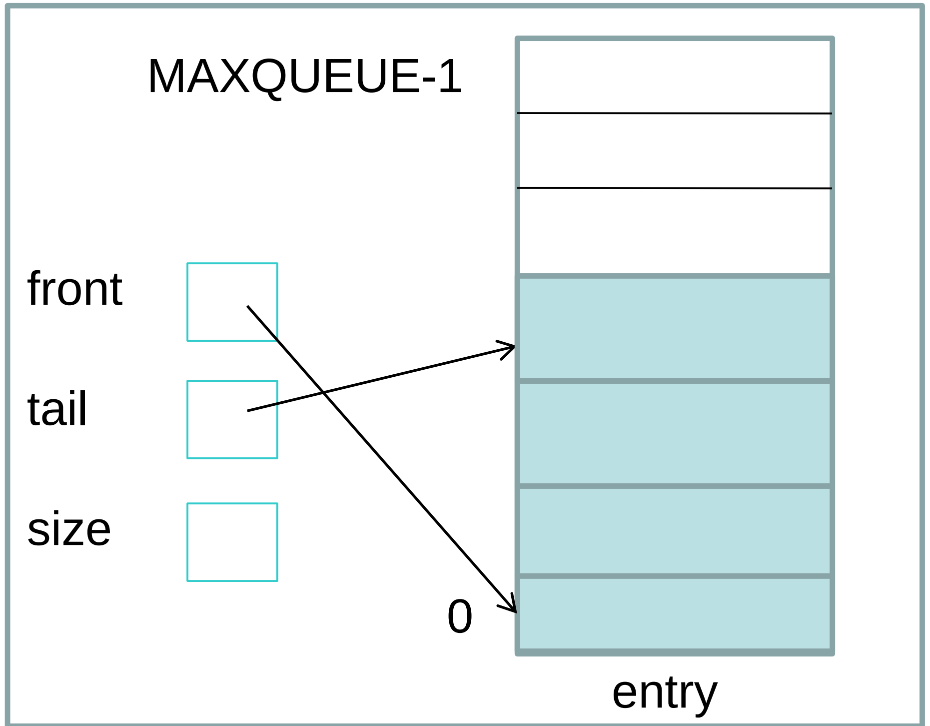
```c
void Serve(QueueEntry *pe, Queue* pq){
  *pe = pq->entry[pq->front];
  pq->front = (pq->front + 1) % MAXQUEUE;
  pq->size--;
}
```



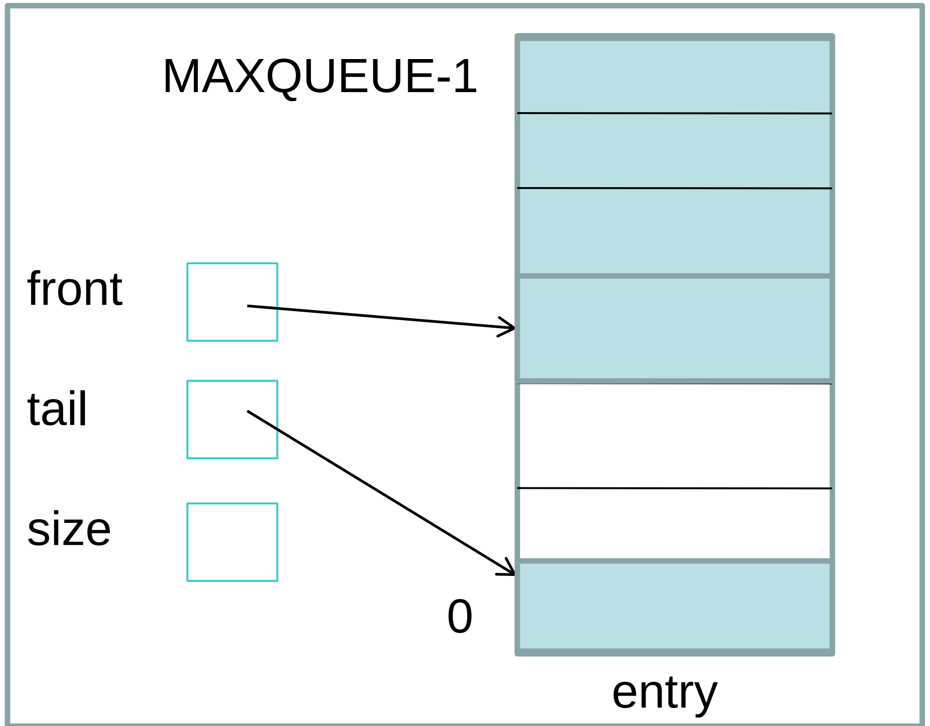
```c
int QueueEmpty(Queue* pq){
      return !pq->size;
}

int QueueFull(Queue* pq){
      return (pq->size == MAXQUEUE);
}
```



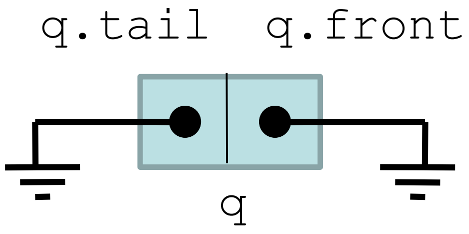
```c
int QueueSize(Queue* pq){
    return pq->size;
}
void ClearQueue(Queue* pq){
    pq->front = 0;
    pq->tail  = -1;
    pq->size  = 0;
}//same as CreateQueue. No nodes to free.
```



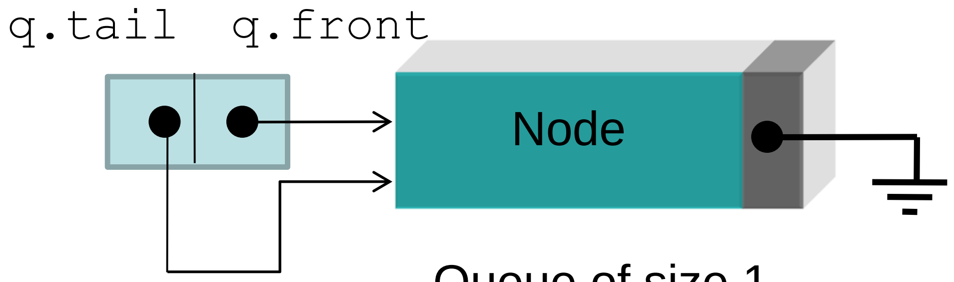
```
void TraverseQueue(Queue* pq){
      int pos, s;
      for(pos=pq->front, s=0; s < pq->size; s++)
      {
            cout<< pq->entry[pos];
            pos=(pos+1)%MAXQUEUE;
      }
}
```



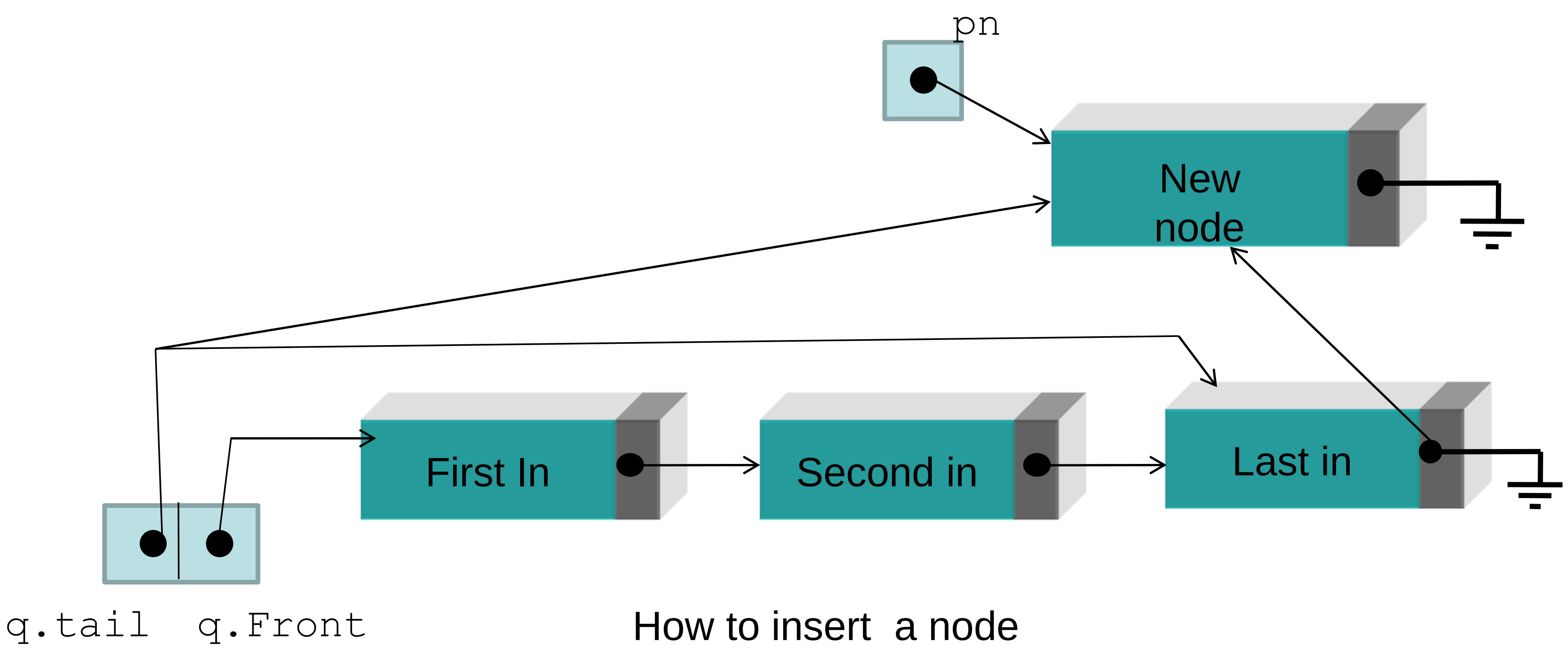
Linked Queues (to overcome fixed size limitations)



Empty Queue



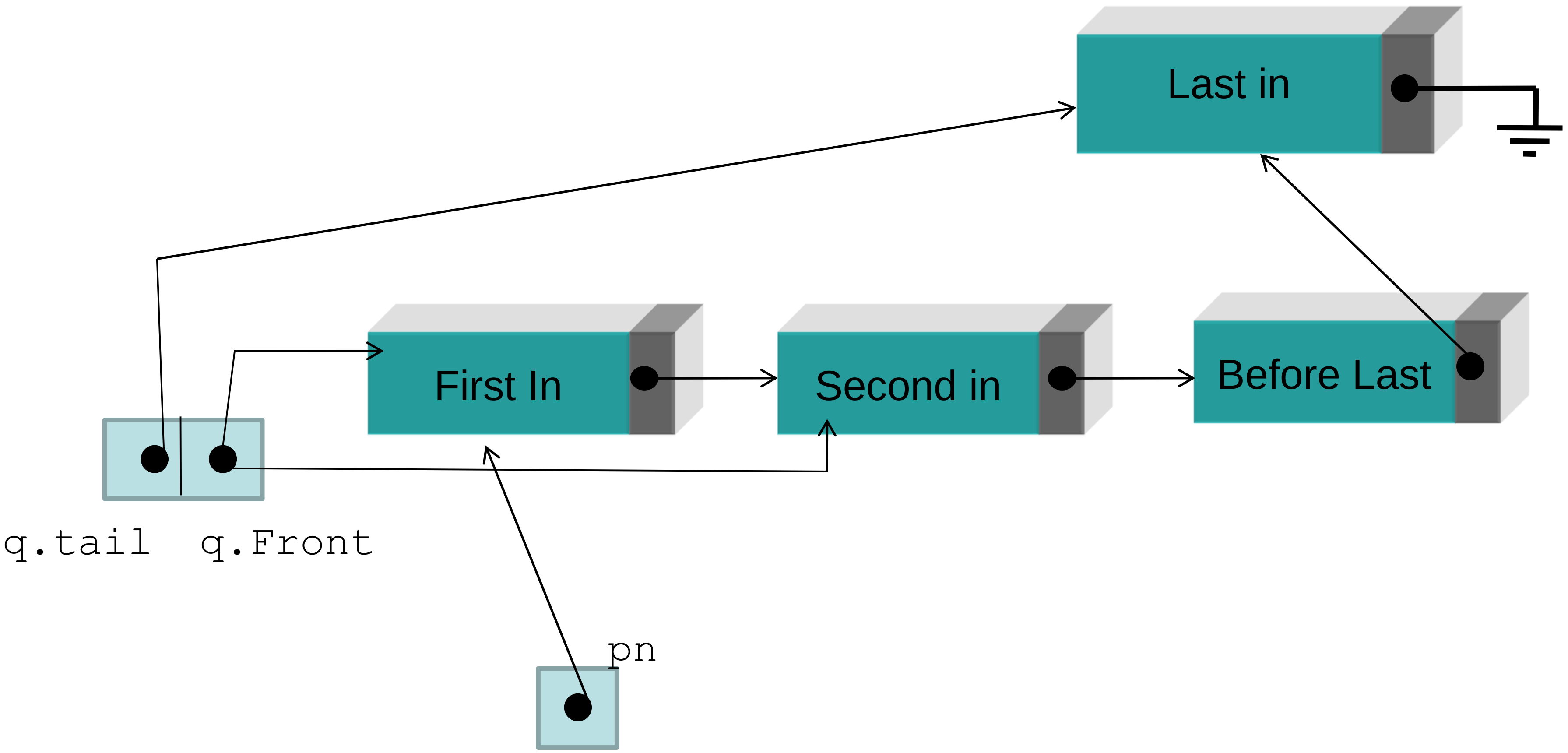
Queue of size 1



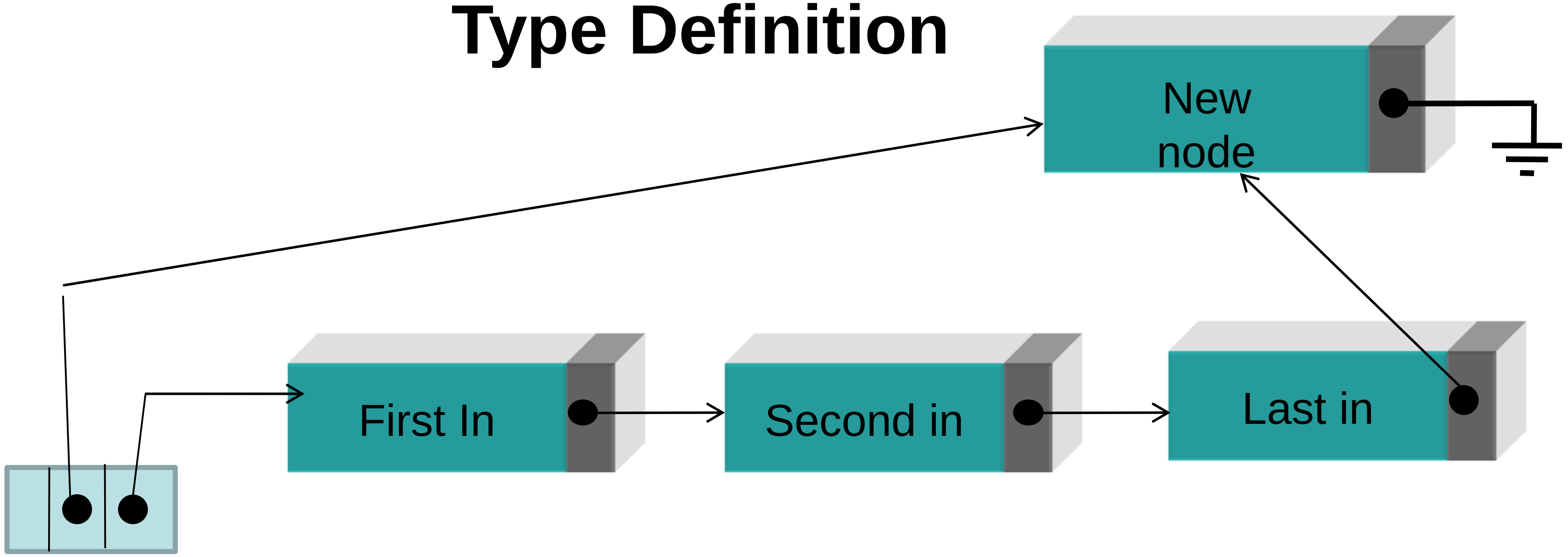
How to insert a node

Linked Queues (to overcome fixed size limitations)

How to serve a node



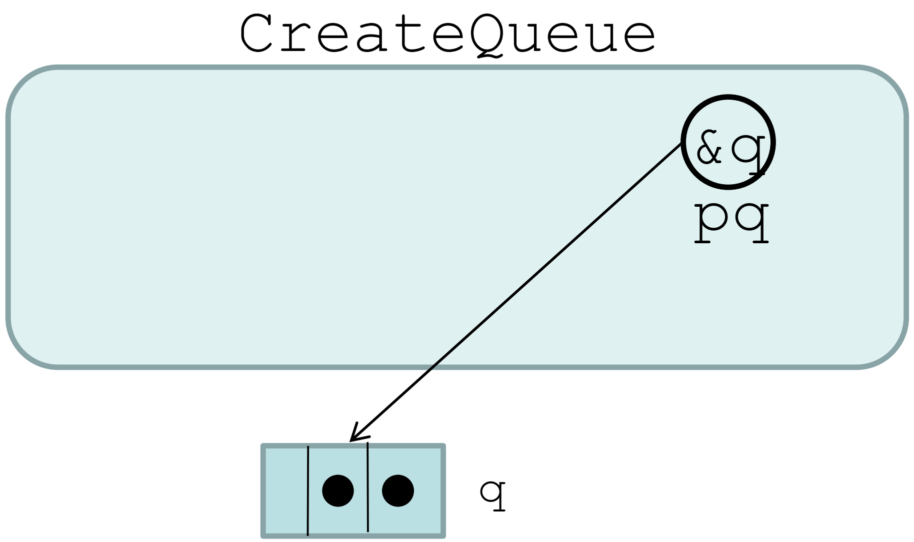
Type Definition




```
struct QueueNode{
      QueueEntry entry;
      struct queuenode *next;
};
struct Queue{
      QueueNode *front;
      QueueNode *tail;
      int  size; // same as stack size issue in the
                 // linked-based implementation
};
```

Implementation level (what really happens)

```
void CreateQueue(Queue *pq){
   pq->front=NULL;
   pq->tail=NULL;
   pq->size=0;
}
```



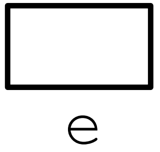
User Level (interface)

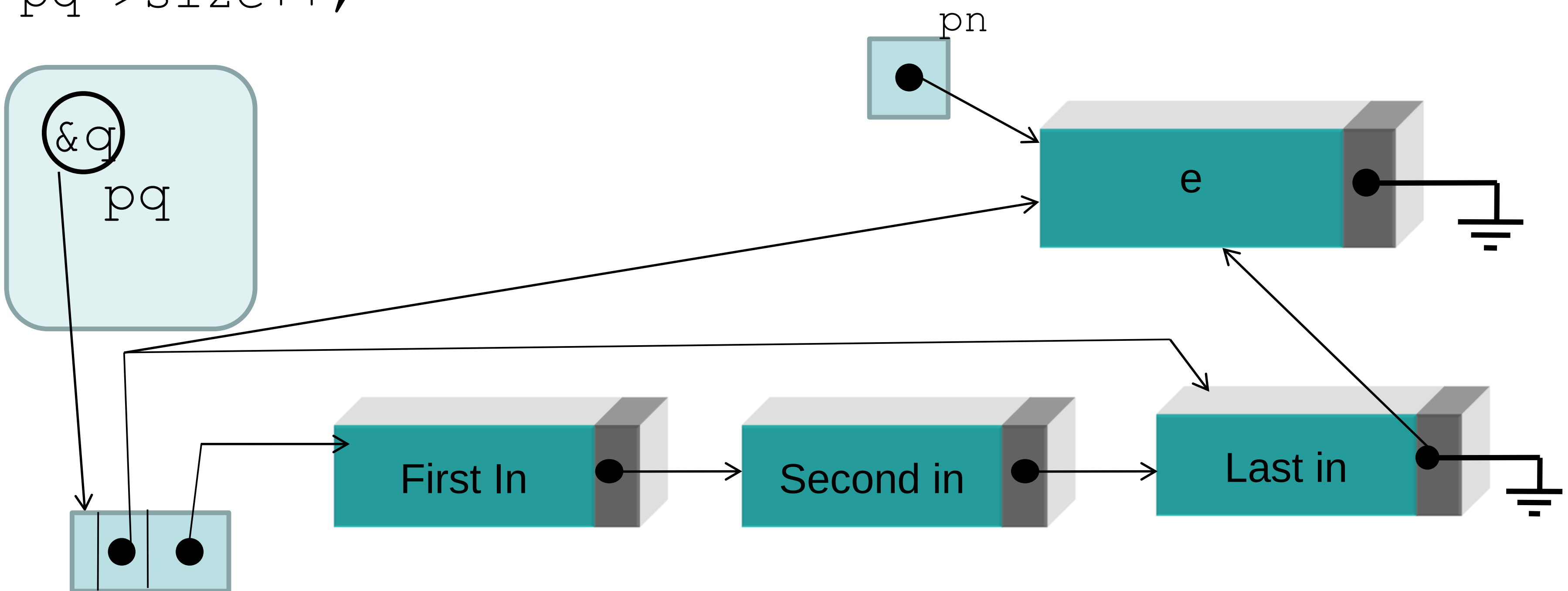
```
void main(){

Queue q;

CreateQueue(&q);


}
```

```
void Append(QueueEntry e, Queue* pq){
  QueueNode* pn= new QueueNode;
  pn->next=NULL;
  pn->entry=e;



    pq->tail->next=pn;
  pq->tail=pn;
  pq->size++;
}
```

User Call:

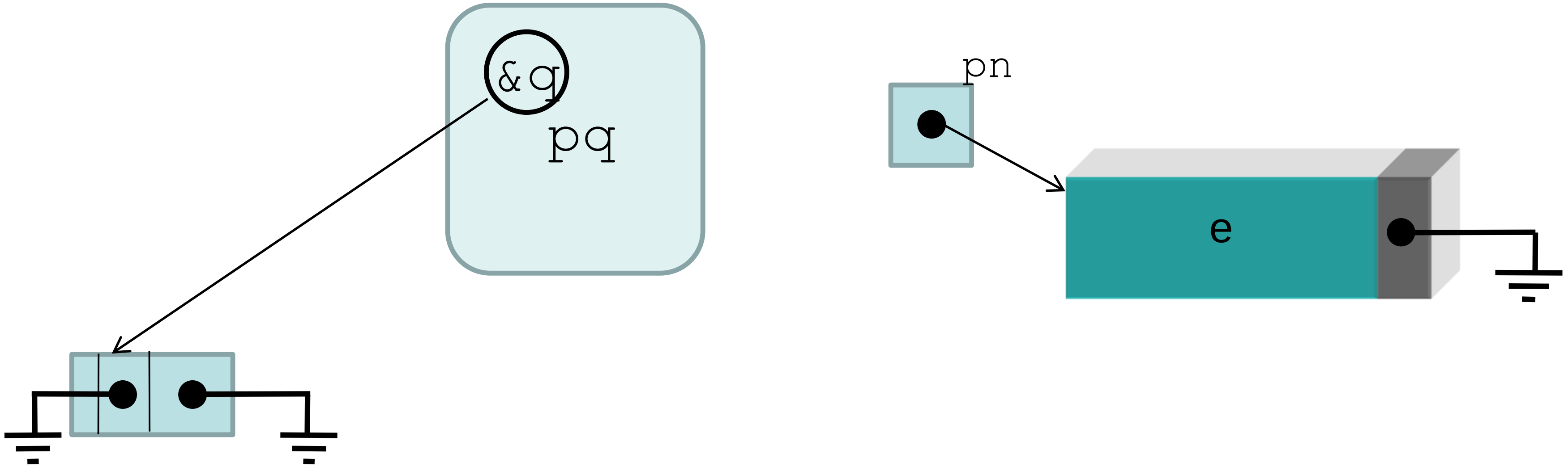
```
Queue q;
QueueEntry e;

Append(e, &q);
```



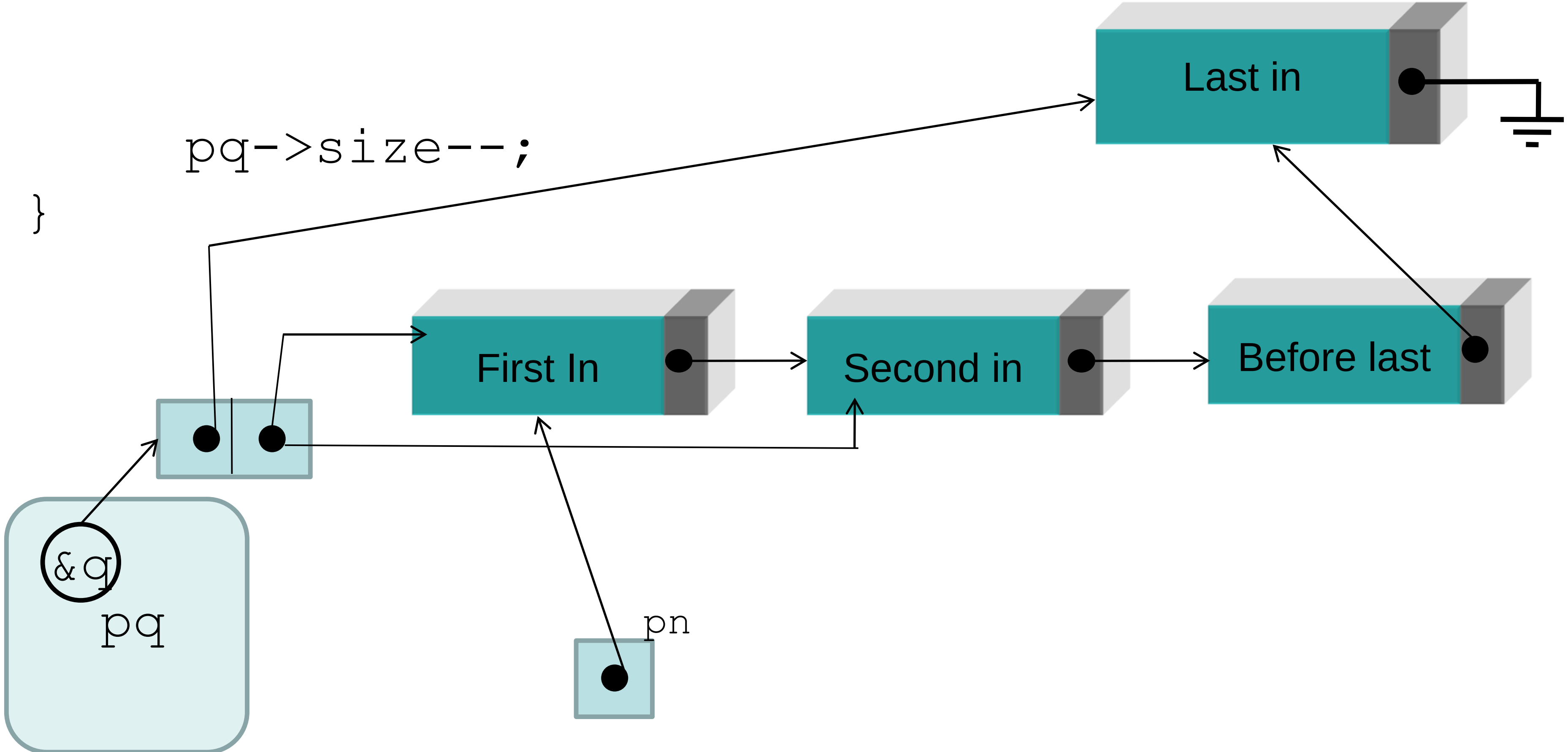


Always take care of special cases (queue is empty)

```
void Append(QueueEntry e, Queue* pq){
  QueueNode* pn= new QueueNode;
  pn->next=NULL;
  pn->entry=e;
  if (!pq->tail) // if the queue is empty
    pq->front=pn;
  else
    pq->tail->next=pn; // run time error for empty queue
  pq->tail=pn;
  pq->size++;
}
```

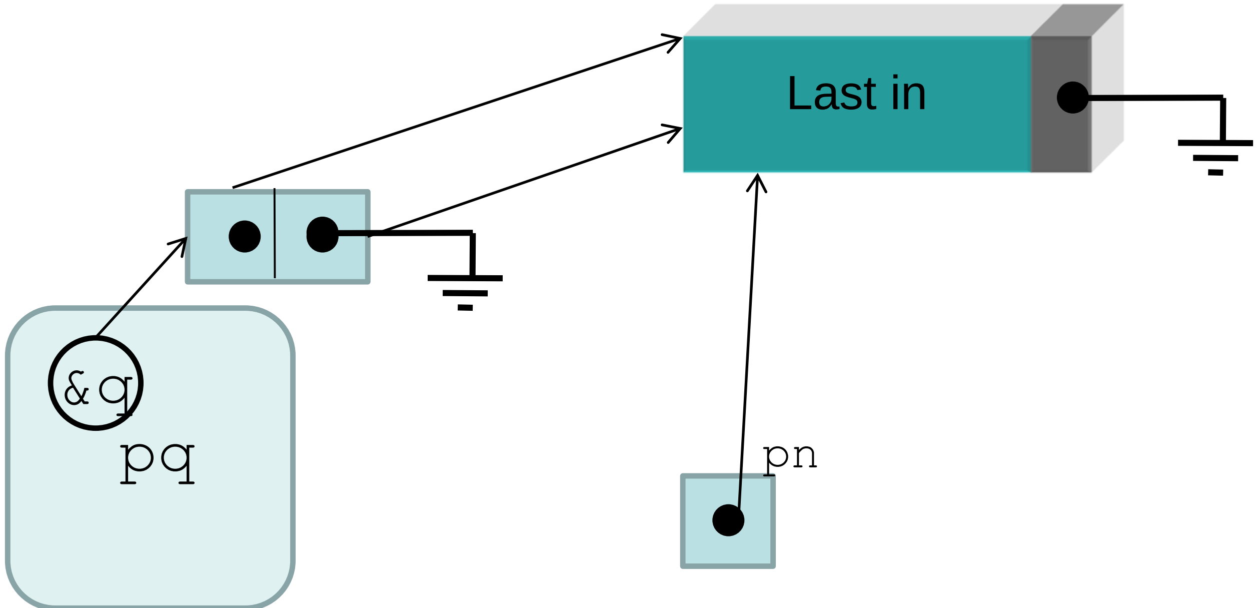


```c
void Serve(QueueEntry *pe, Queue* pq){
      QueueNode *pn=pq->front;
      *pe=pn->entry;
      pq->front=pn->next;
      delete pn;


      pq->size--;
}
```



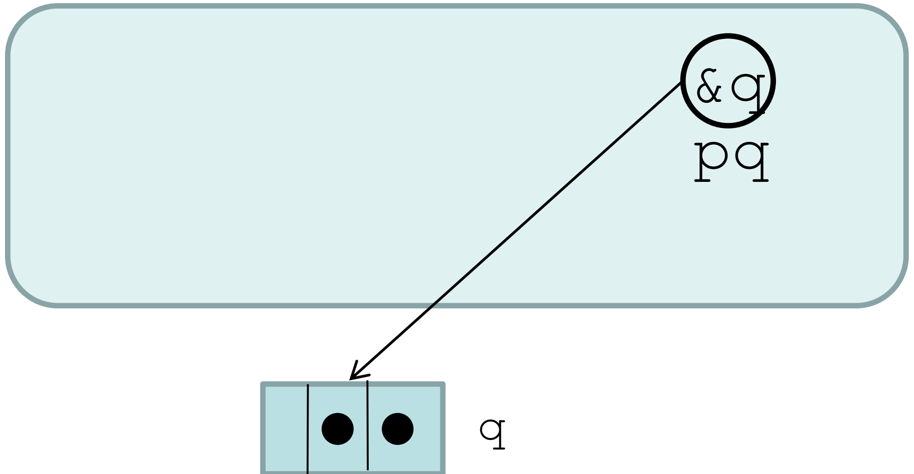
Always take care of special cases: Only one element exists

```
void Serve(QueueEntry *pe, Queue* pq){
     QueueNode *pn=pq->front;
     *pe=pn->entry;
     pq->front=pn->next;
     delete pn;
     if (!pq->front) // if the queue became empty
          pq->tail=NULL;
     pq->size--;
}
```



Implementation level (what really happens)

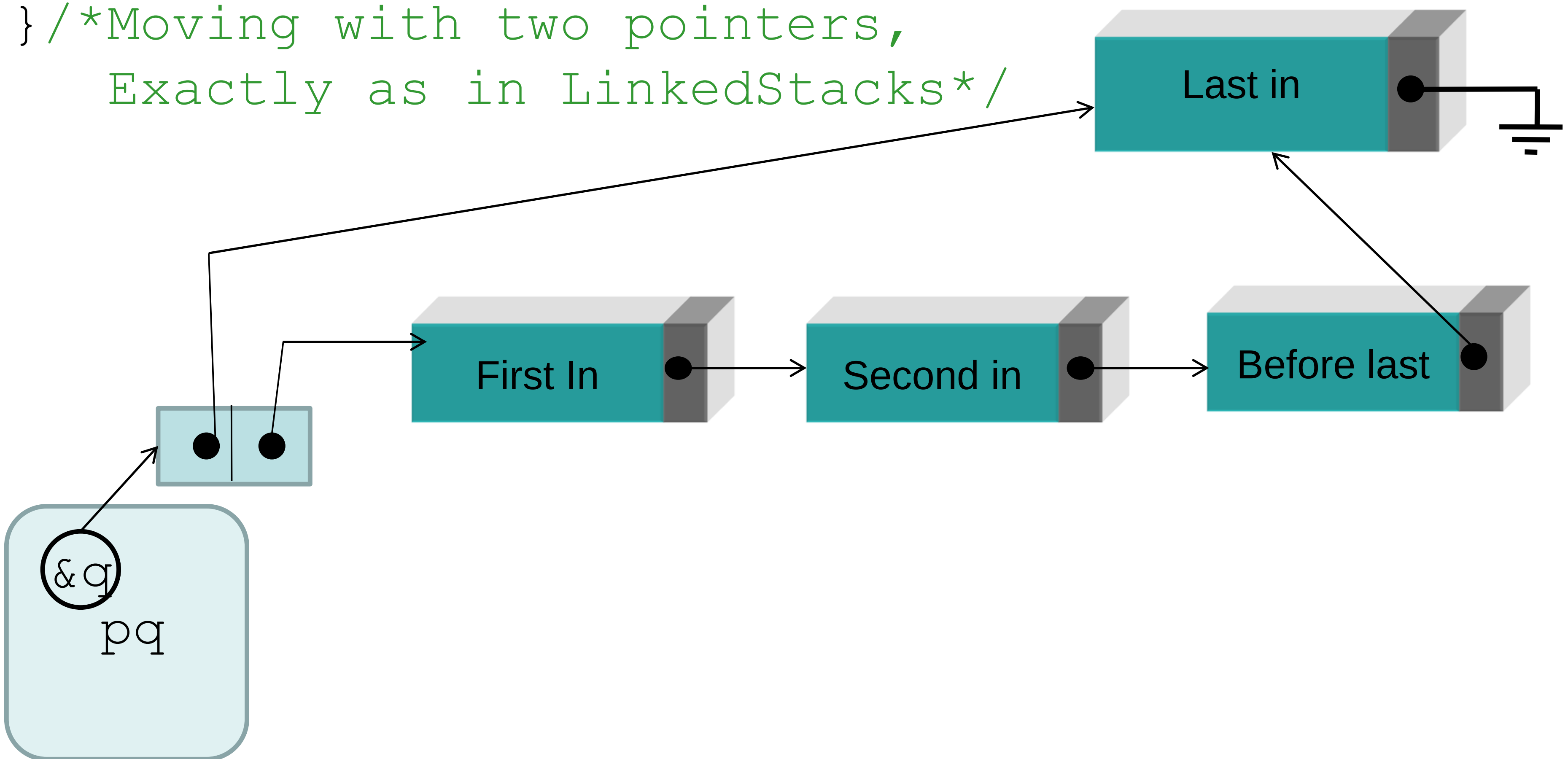
```
int QueueEmpty(Queue* pq){
      return !pq->front;
}

int QueueFull(Queue* pq){
      return 0;
}

int QueueSize(Queue* pq){
      return pq->size;
}
```



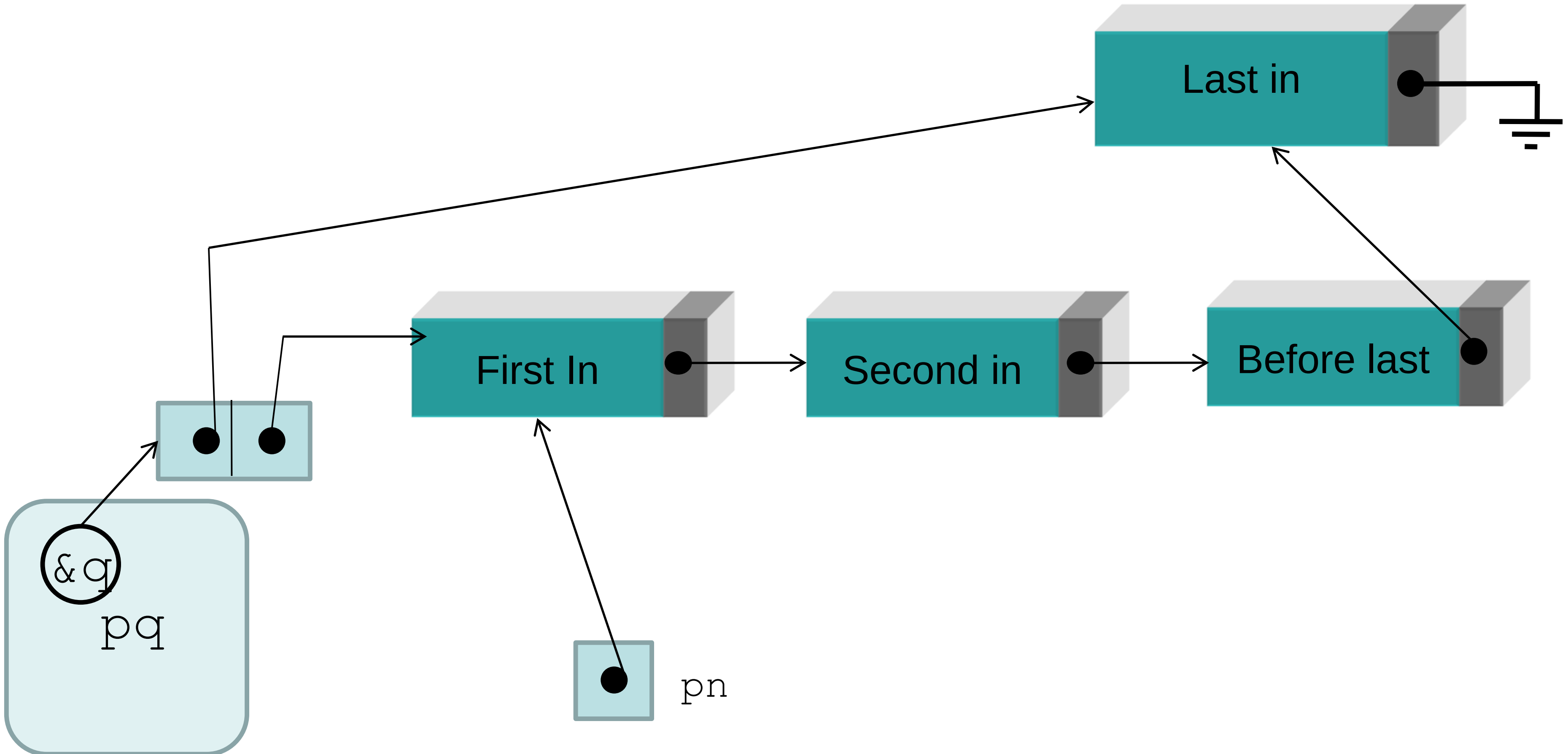
User Level (interface)

```
void main(){

Queue q;

CreateQueue(&q);



}
```

```
void ClearQueue(Queue* pq){
     while(pq->front){
          pq->tail=pq->front->next;
          delete pq->front;
          pq->front=pq->tail;
     }
     pq->size  = 0;
}/*Moving with two pointers,
   Exactly as in LinkedStacks*/
```



```
void TraverseQueue(Queue* pq){
      for(QueueNode *pn=pq->front; pn; pn=pn->next)
            cout<<pn->entry<<"\t";
}
```



Very important note for all linked structures. E.g., in Queues:

In `Push` and `Append` we have to check for exhausted memory.

The code can be modified to:

```
int Append(QueueEntry e, Queue* pq){
  QueueNode* pn = new QueueNode;
  if (!pn)
    return 0;
    /*This is much better than the Error message of
      the book because this is more flexible. Also,
      the same function for contiguous implementation
      has to return 1 always to have consistent
      interface*/
  else{
    //Put here exactly all of the remaining code
    return 1;
  }
```

```
...;
If (!Append(e, &q)){
     ...;
}
```