자료구조

Chap 5-2. Queue

2018년 1학기

컴퓨터과학과
민 경 하

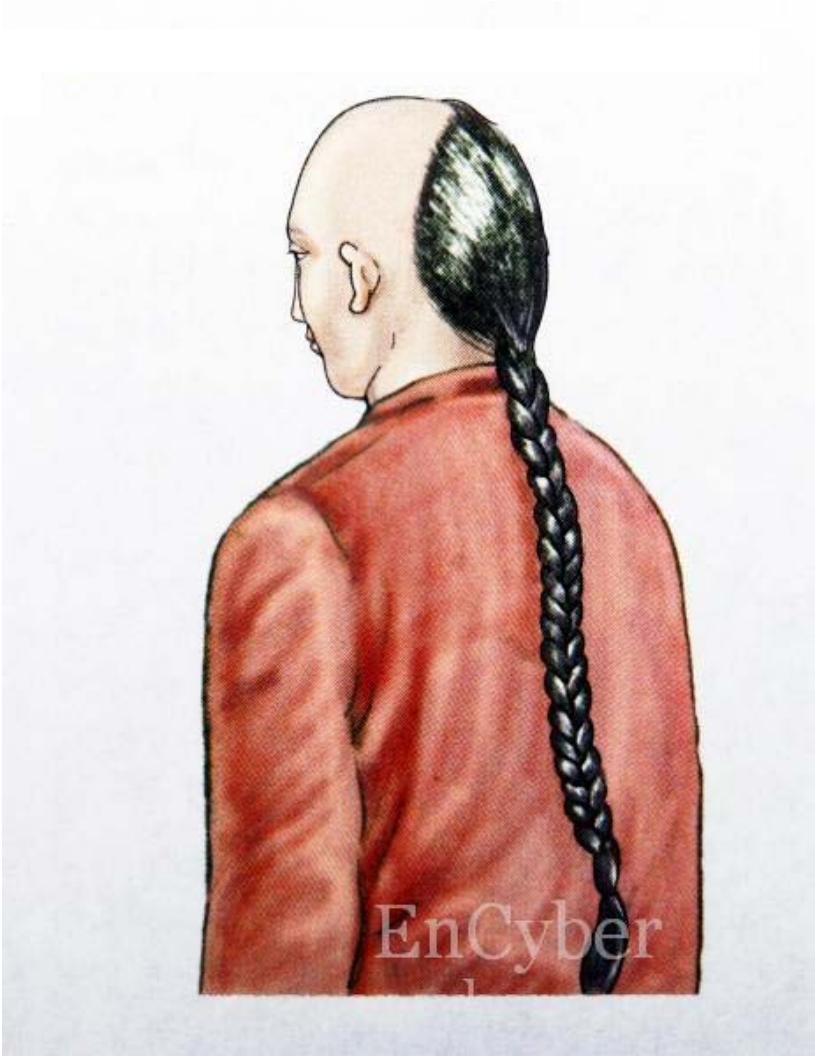
Contents

5. Queues

1. Definition of queue
2. Operations of queue
3. Data structure of queue
4. Implementation of operations
5. Special queues

1. Definition of Queue

- 辮髮



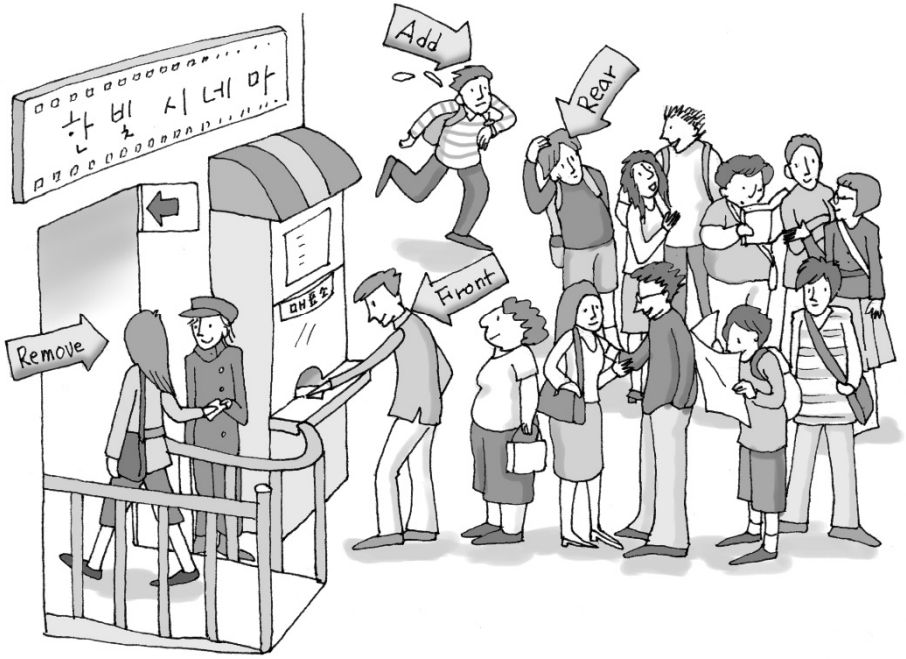
1. Definition of Queue

- Queue
  - A list that records the arrival time of its elements

```
다음의 공통점은?
(1) 맛집
(2) 뮤지컬 티켓
(3) 콘서트 티켓
(4) 수강신청
(5) 추석 귀성 기차표
```

1. Definition of Queue

- Waiting list
  - The order of arrival



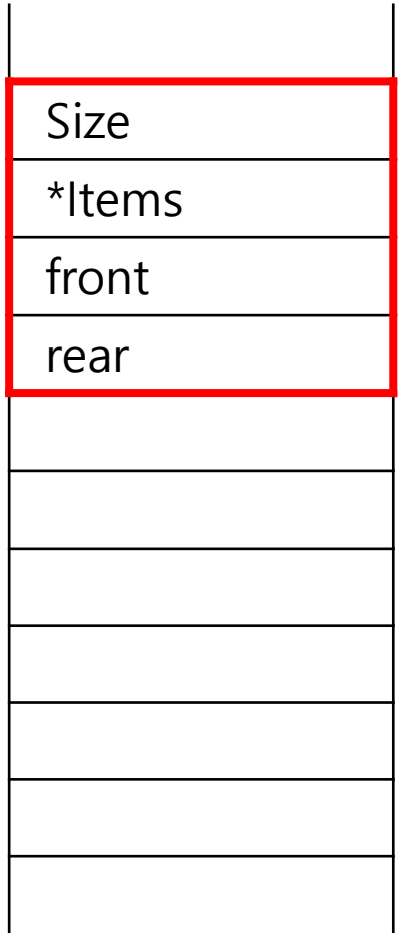
1. Definition of Queue

- Queue
  - An ordered list in which insertions and deletions are made at each end of the list
    - One end where data are inserted is called ***REAR***
    - The other end where data are deleted is called ***FRONT***
    - The insertion operation is called ***ADDQ*** (**ADD** or **ENQUEUE**)
    - The deletion operation is called ***DELETEQ*** (**REMOVE** or **DEQUEUE**)
  - FIFO (First-In-First-Out)

2. Data structure of Queue

- Data structure of queue
  - Size
  - List of elements
  - rear, front

```
Class queue{
        int size;
        DataType *Items;
        int rear, front;
};
```



3. Operations of queue

① CreateQueue

- Create a queue of size n

② IsEmpty

- Return True, if the queue is empty

③ IsFull

- Return True, if the queue is full

④ AddQ

- Insert a new element to a queue

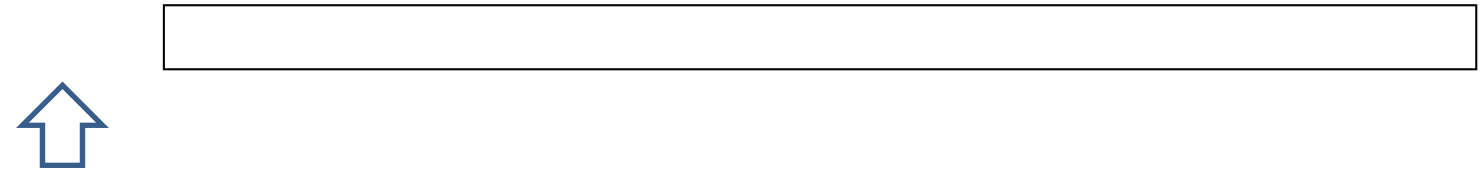
⑤ DeleteQ

- Delete an element from a queue

3. Operations of Queue

④ ADDQ

- Inserting a new element to a queue
- A new element can be added at ***REAR***
- Example:

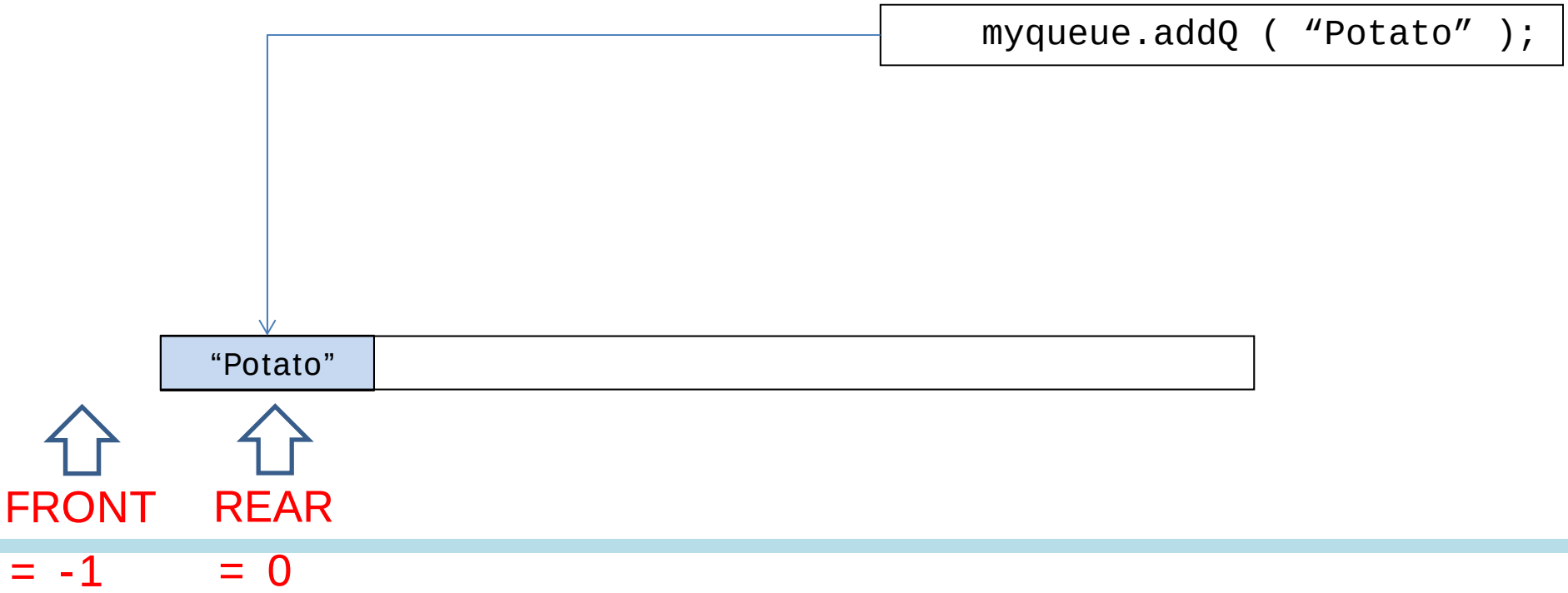


FRONT = REAR = -1

3. Operations of Queue

④ ADDQ

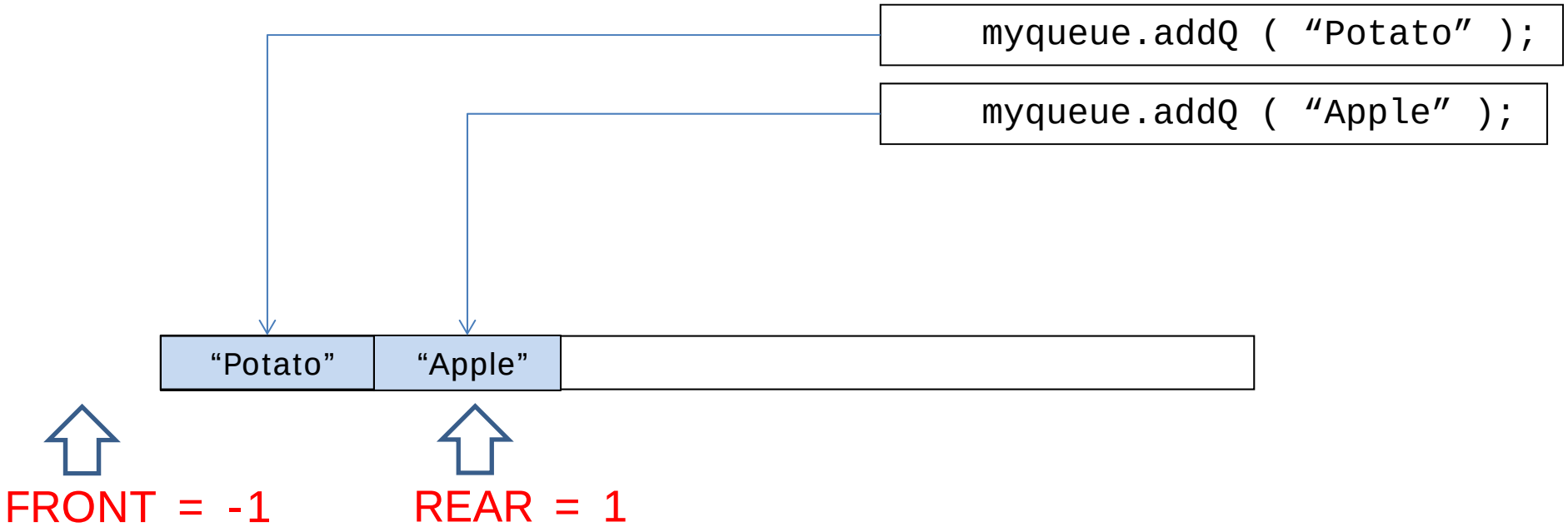
- Inserting a new element to a queue
- A new element can be added at ***REAR***
- Example:



3. Operations of Queue

④ ADDQ

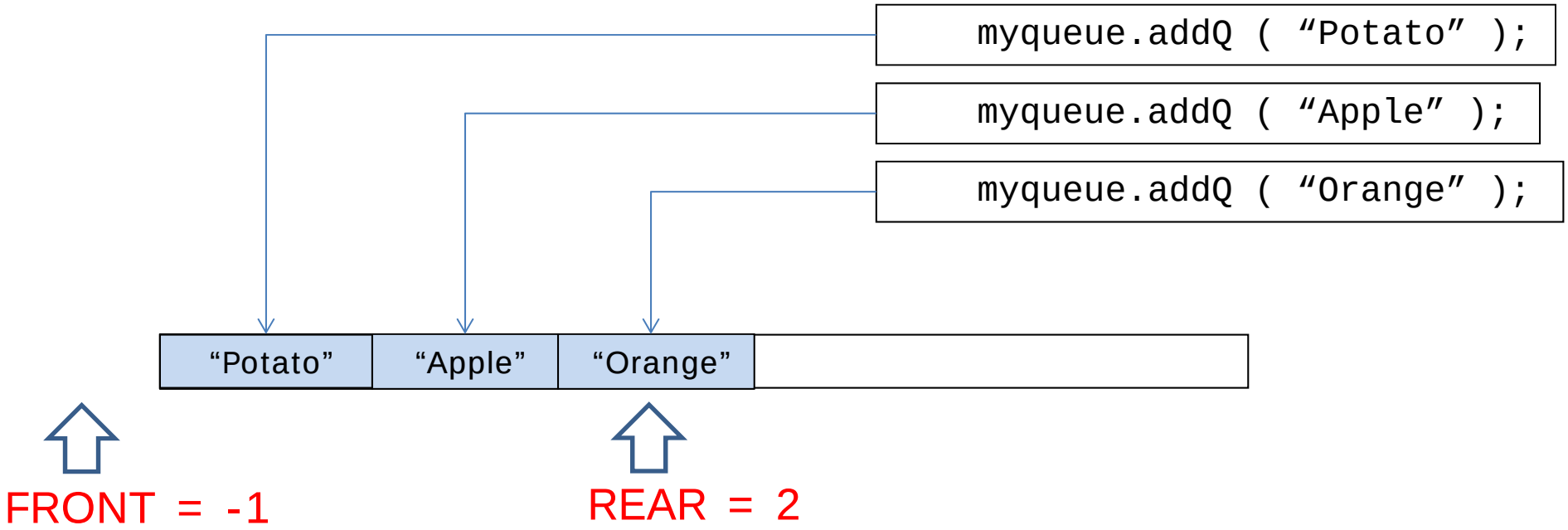
- Inserting a new element to a queue
- A new element can be added at ***REAR***
- Example:



3. Operations of Queue

④ ADDQ

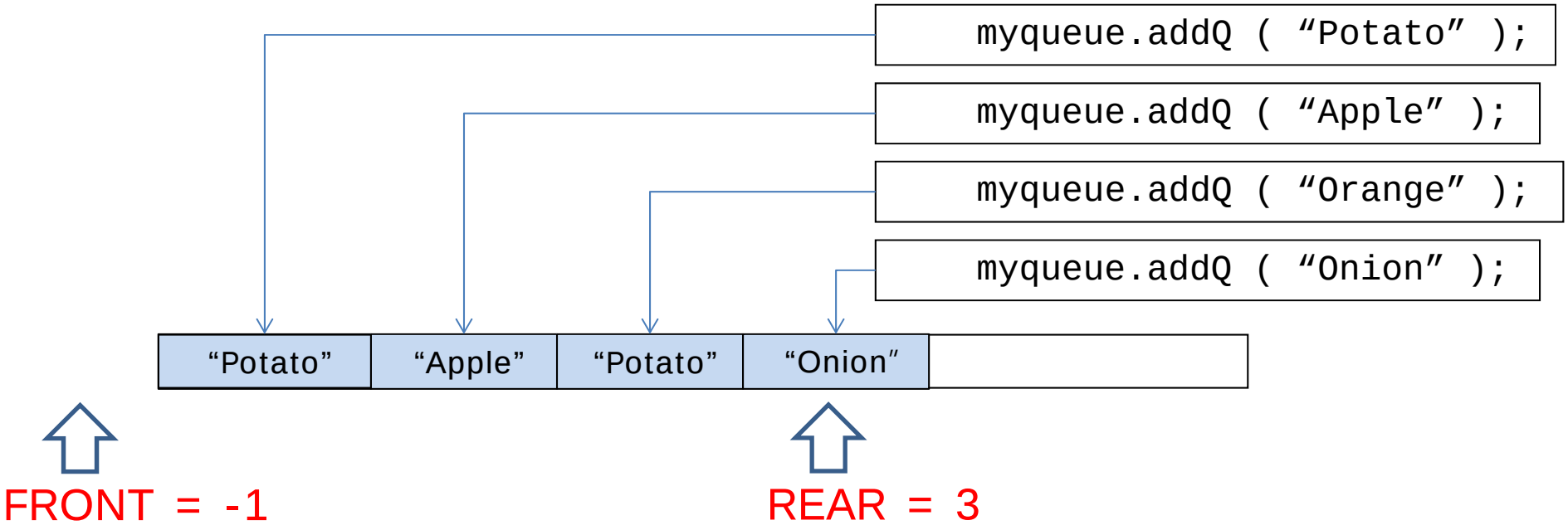
- Inserting a new element to a queue
- A new element can be added at ***REAR***
- Example:



3. Operations of Queue

④ ADDQ

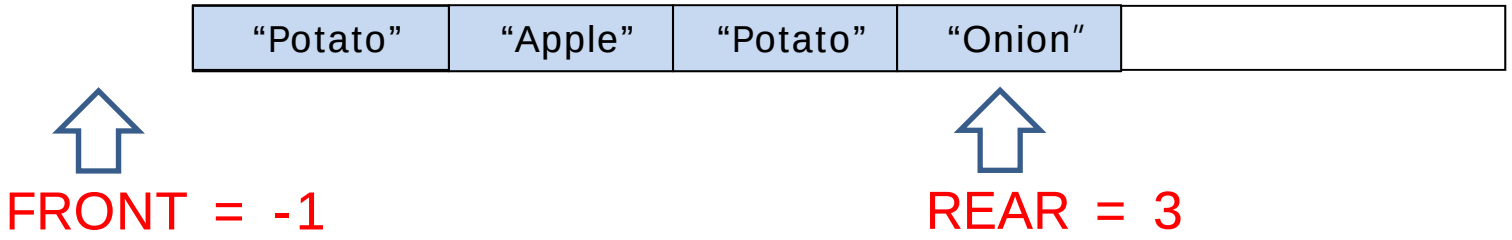
- Inserting a new element to a queue
- A new element can be added at ***REAR***
- Example:



3. Operations of Queue

⑤ DELETEQ

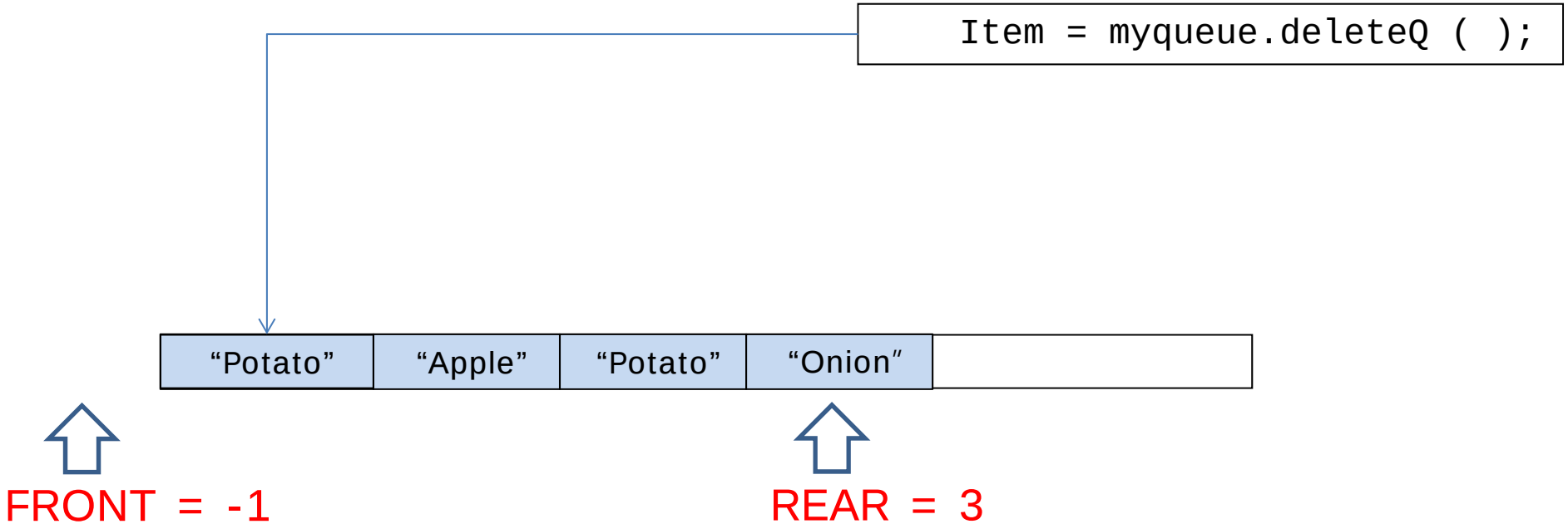
- Deleting an element from a queue
- An element is deleted at ***FRONT***
- Example:



3. Operations of Queue

⑤ DELETEQ

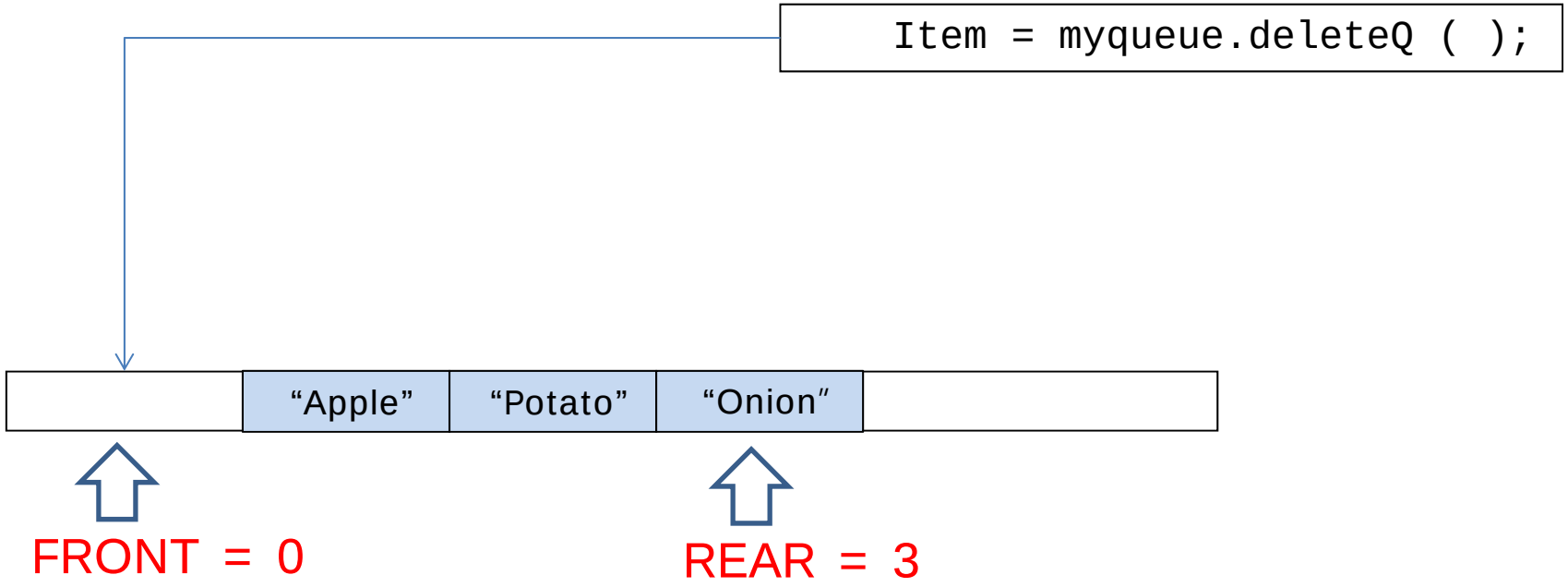
- Deleting an element from a queue
- An element is deleted at ***FRONT***
- Example:



3. Operations of Queue

⑤ DELETEQ

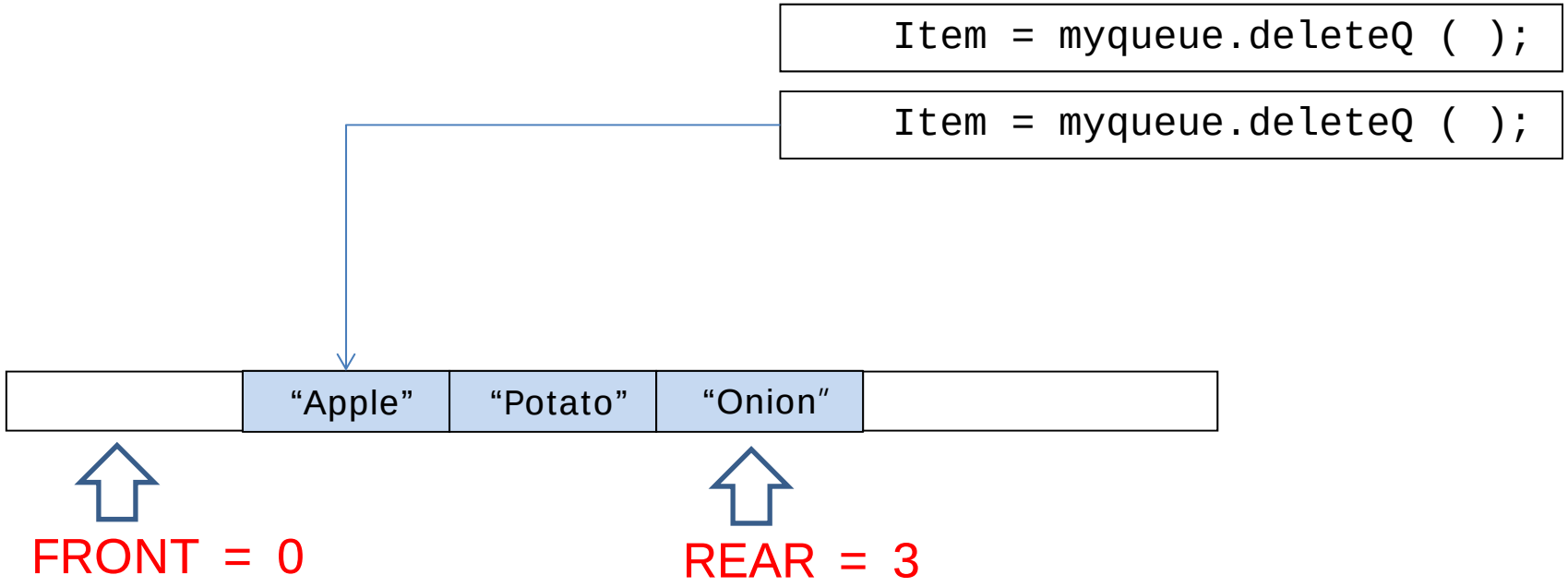
- Deleting an element from a queue
- An element is deleted at ***FRONT***
- Example:



3. Operations of Queue

⑤ DELETEQ

- Deleting an element from a queue
- An element is deleted at ***FRONT***
- Example:



3. Operations of Queue

⑤ DELETEQ

- Deleting an element from a queue
- An element is deleted at ***FRONT***
- Example:

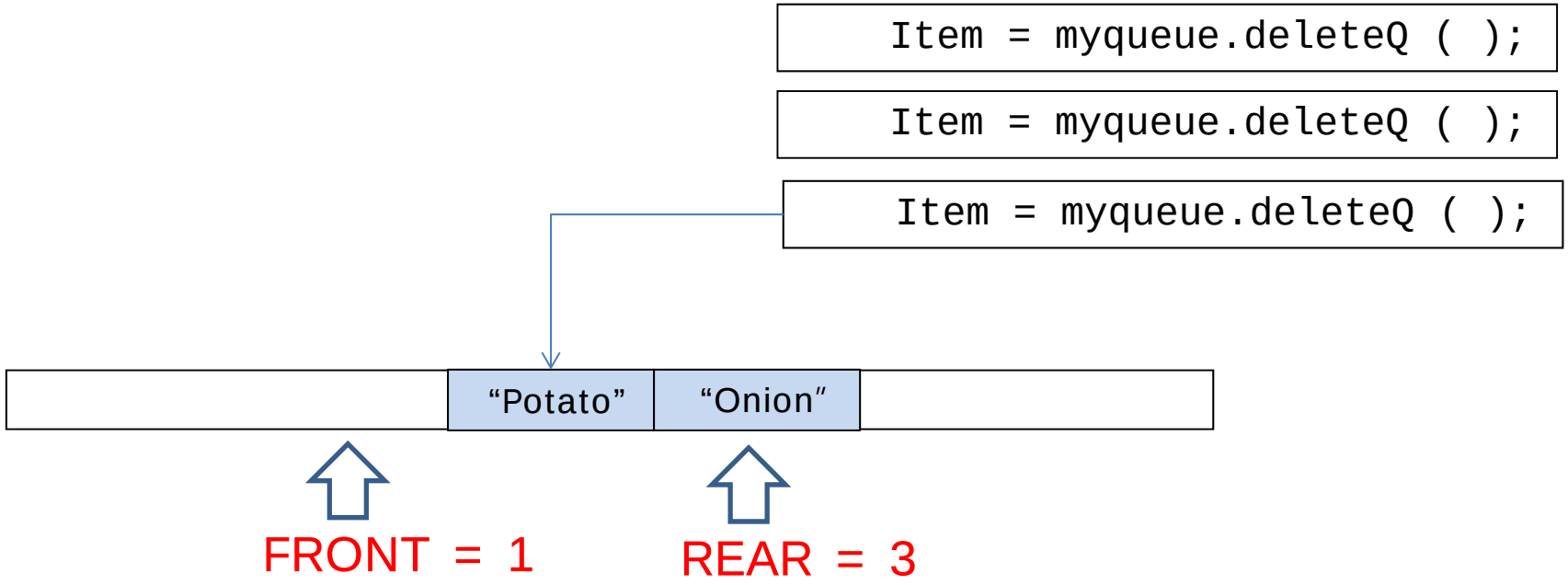
```
Item = myqueue.deleteQ ( );
```

```
Item = myqueue.deleteQ ( );
```

3. Operations of Queue

⑤ DELETEQ

- Deleting an element from a queue
- An element is deleted at ***FRONT***
- Example:



3. Operations of Queue

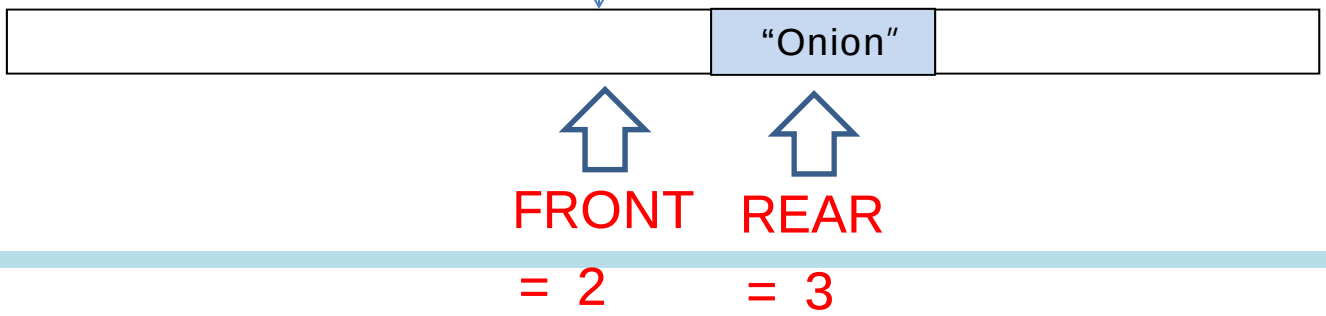
⑤ DELETEQ

- Deleting an element from a queue
- An element is deleted at ***FRONT***
- Example:

```
Item = myqueue.deleteQ ( );
```

```
Item = myqueue.deleteQ ( );
```

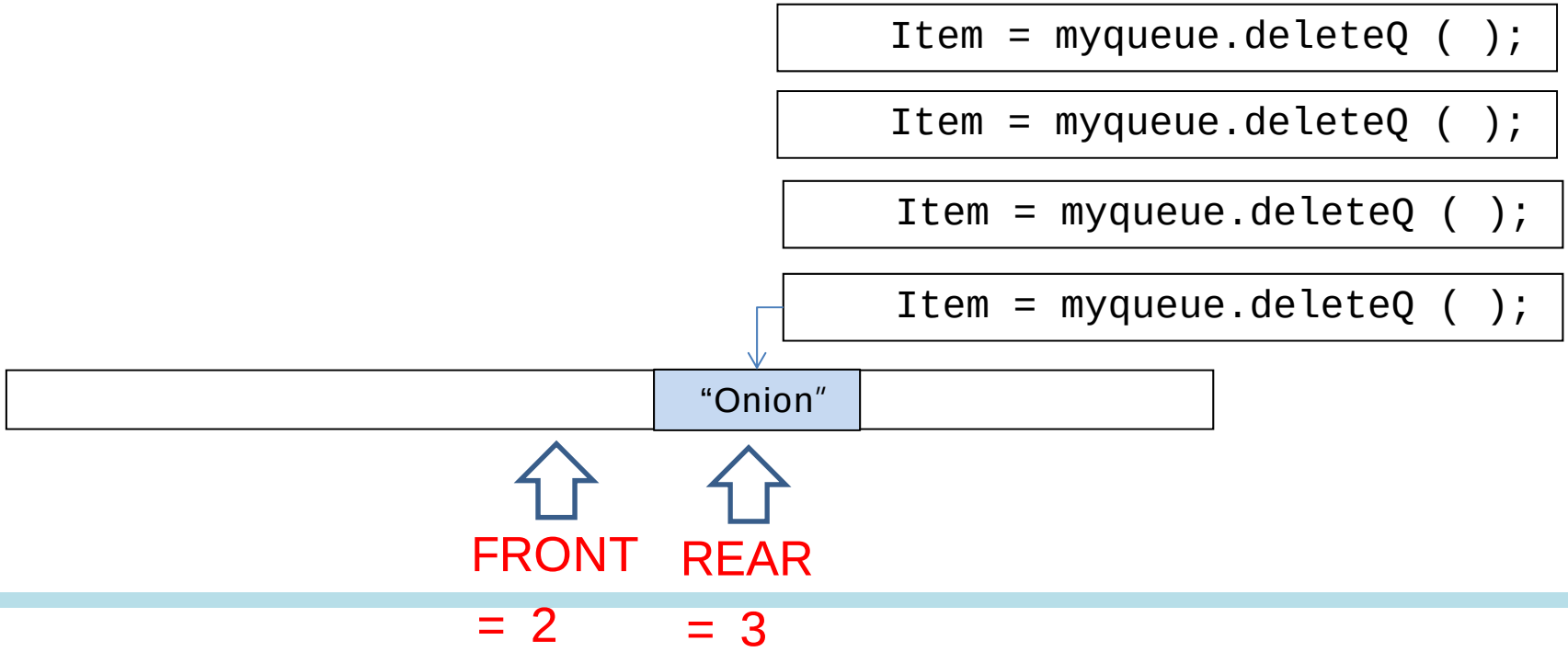
```
Item = myqueue.deleteQ ( );
```



3. Operations of Queue

⑤ DELETEQ

- Deleting an element from a queue
- An element is deleted at ***FRONT***
- Example:



3. Operations of Queue

⑤ DELETEQ

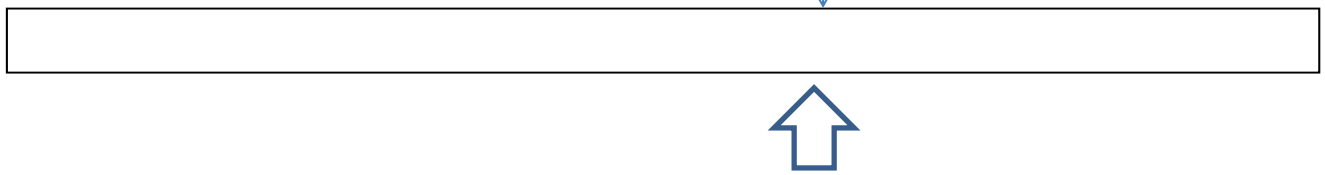
- Deleting an element from a queue
- An element is deleted at ***FRONT***
- Example:

```
Item = myqueue.deleteQ ( );
```

```
Item = myqueue.deleteQ ( );
```

```
Item = myqueue.deleteQ ( );
```

```
Item = myqueue.deleteQ ( );
```

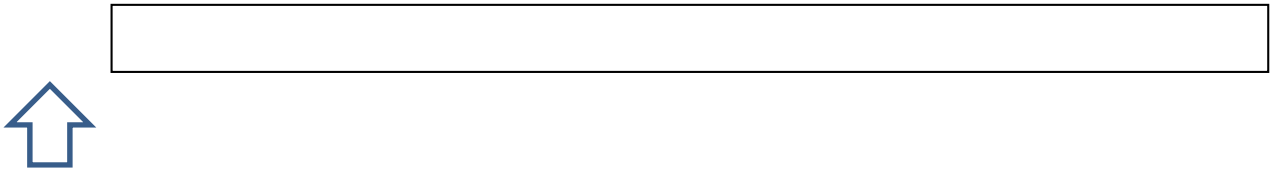


FRONT = REAR = 3

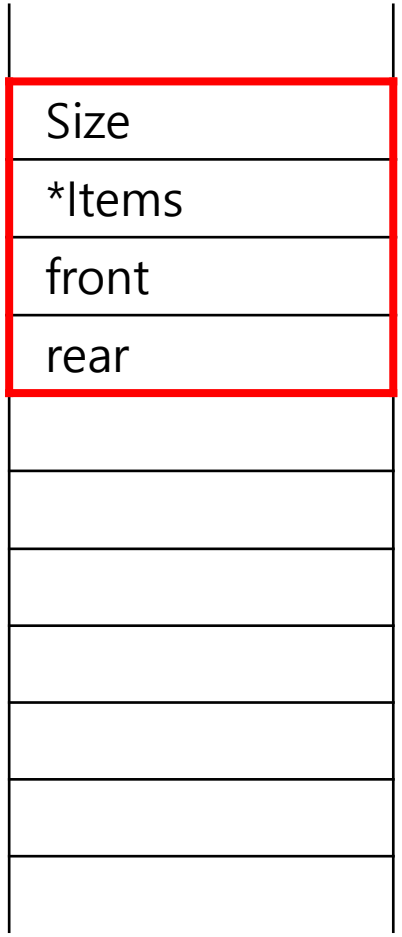
3. Data structure of Queue

- Data structure of queue
  - Size
  - Items
  - rear, front

```
Class queue{
        int size;
        DataType *Items;
        int rear, front;
};
```



FRONT = REAR = -1

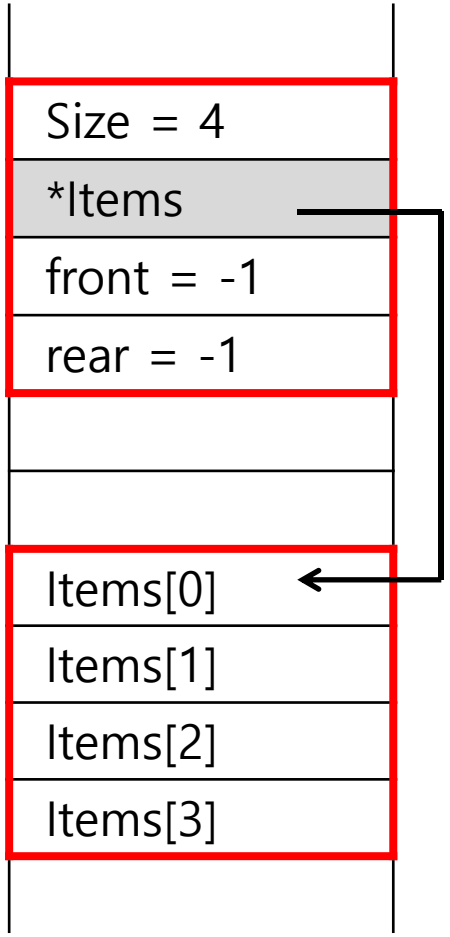


4. Implementation of operations

① CreateQueue (int n)

- Create a queue of size n

```
void queue::Create ( int maxQueueSize )
{
    Size = maxQueueSize ;
    Items = new Datatype[Size];
    front = rear = -1;
}
```

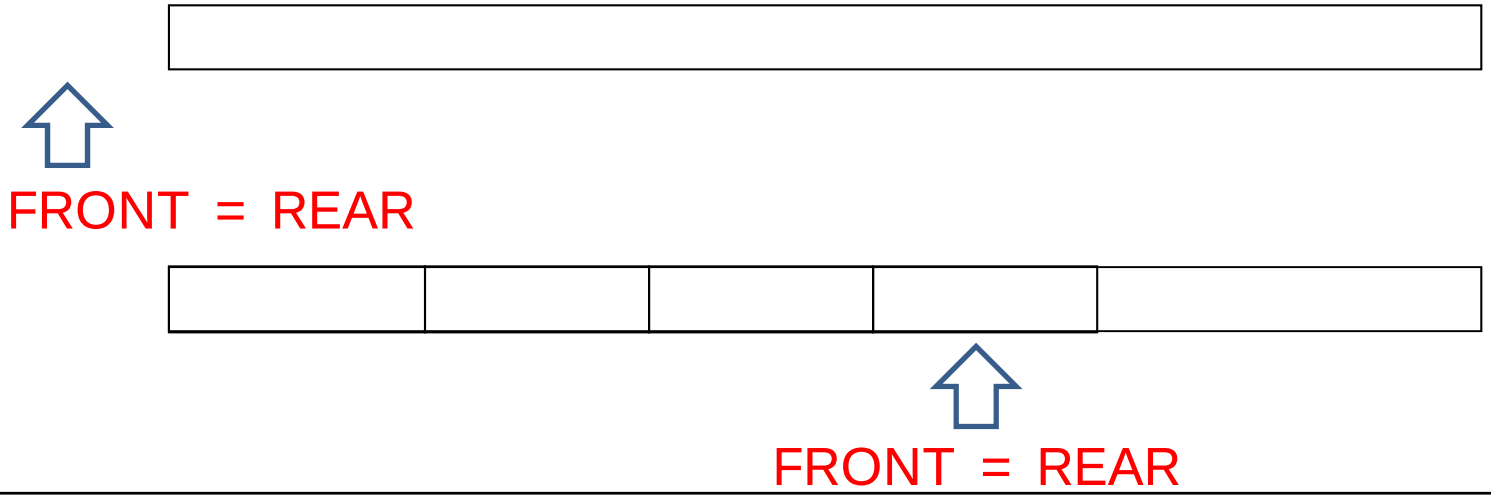



```
void main ( ) {
    Queue myQueue.Create ( 4 );
}
```

4. Implementation of operations

② is_Empty ( )

- Check underflow
- Returns TRUE if the queue is EMPTY

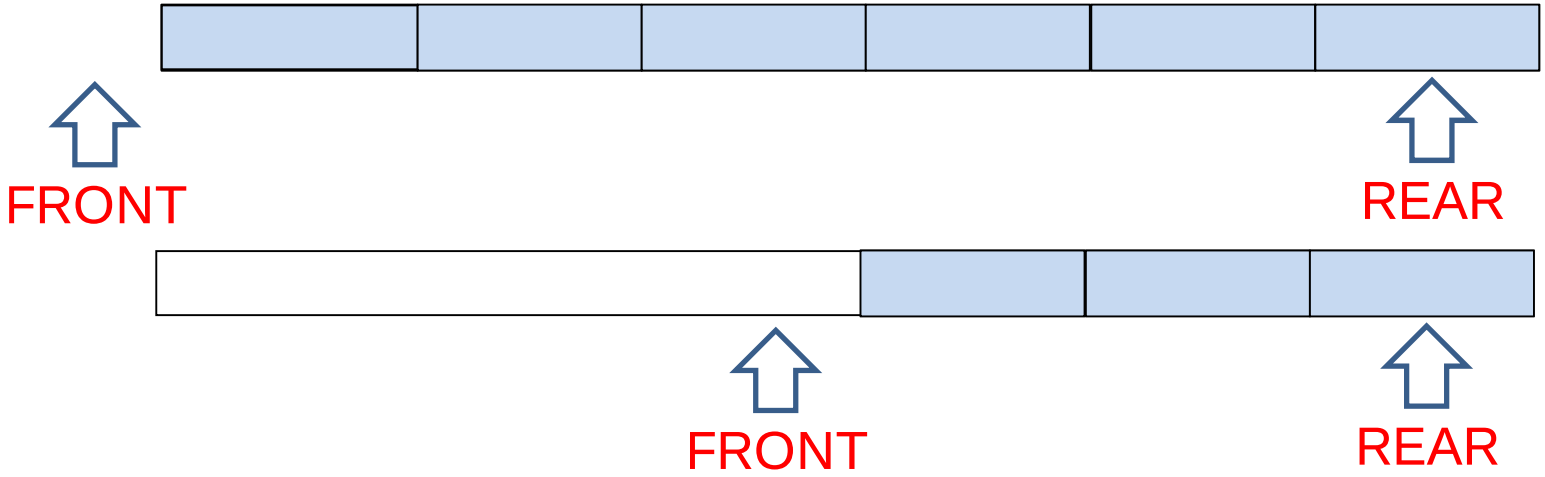



```
Int queue::IsEmpty ( )
{
        return (front == rear);
}
```

4. Implementation of operations

③ is_Full ( )

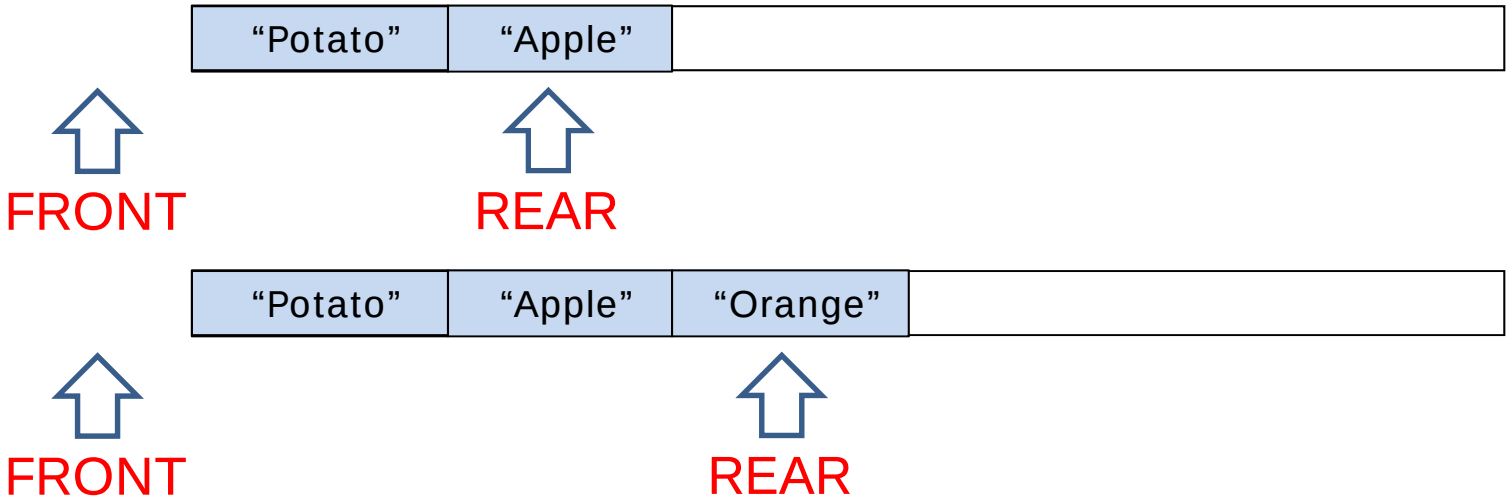
- Check overflow
- Returns TRUE if the stack is FULL




```
int queue::IsFull ( )
{
        return (rear == size-1);
}
```

4. Implementation of operations

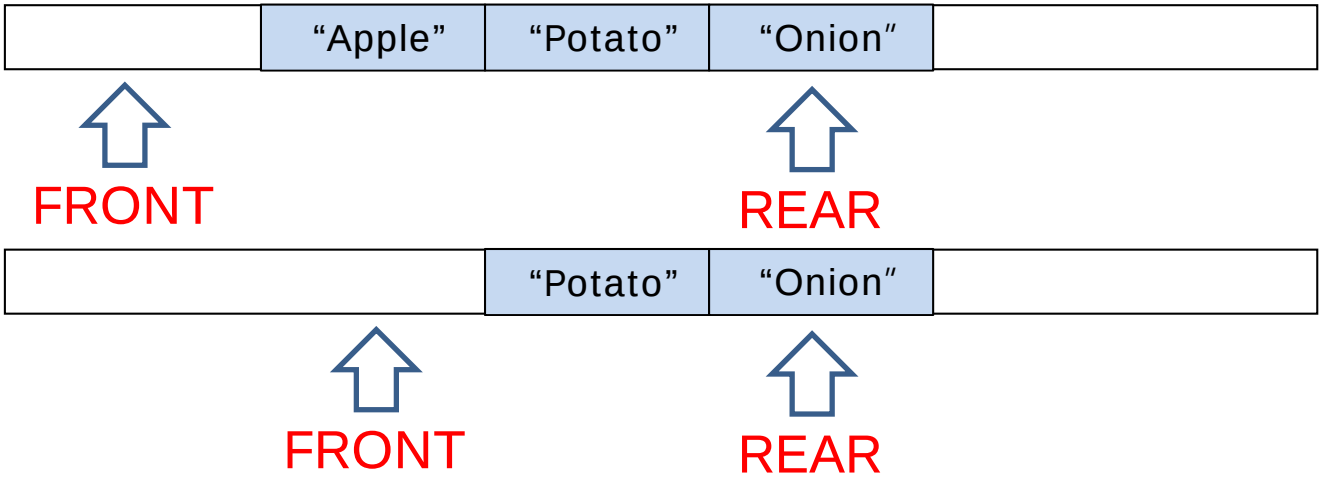
④ AddQ ( )




```
void queue::AddQ ( element item )
{
        if ( isFull ( ) )
                printf("Cannot add an element to a full queue);
        Items[ ++rear ] = item;
}
```

4. Implementation of operations

⑤ DeleteQ ( )




```
element queue::DeleteQ ( )
{
        if ( isEmpty ( ) )
                printf("You cannot delete from an empty queue);
        return Items[ ++front ];
}
```

5. Special Queues

5.1 Circular queue

5.2 DEQ (Doubly-Ended Queue)

5.3 Priority Queue

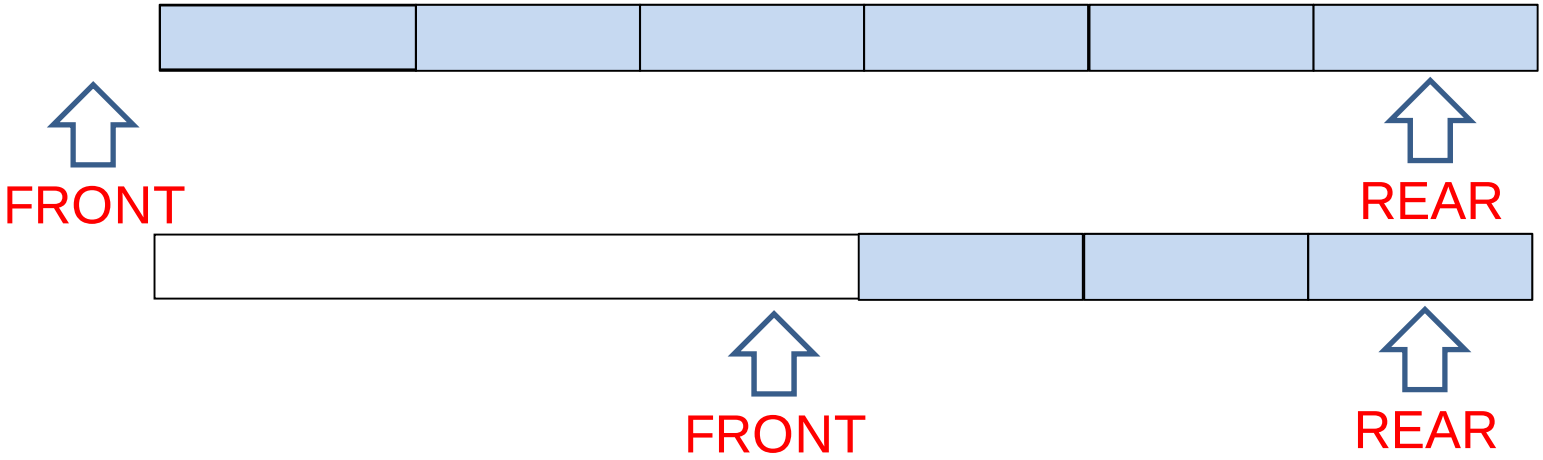
5.1 Circular queue

- Problems of queue?
  - Even though a queue is not FULL,

    IsFullQ ( ) returns TRUE,

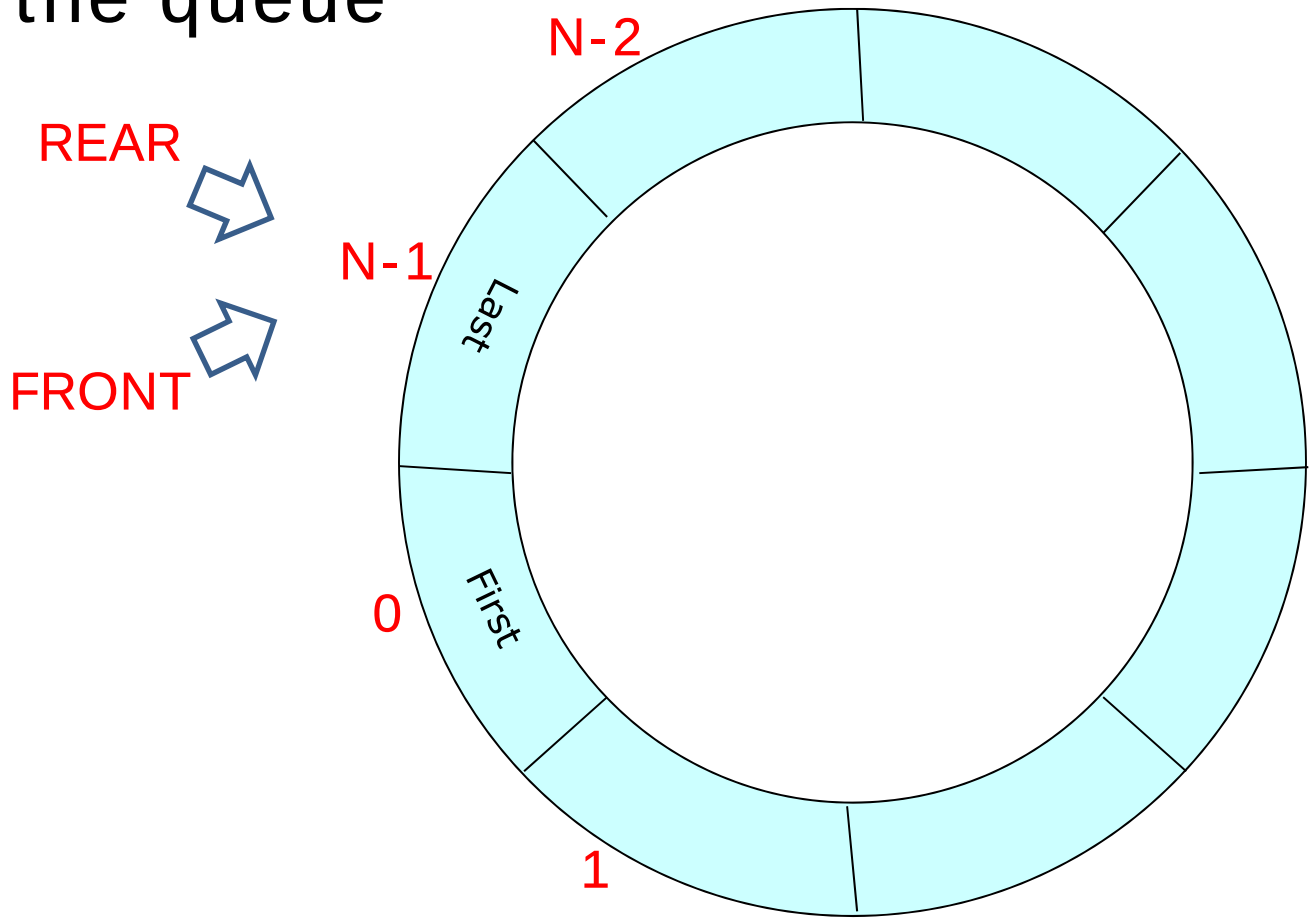
    if REAR == size-1

  - How to solve the problem?



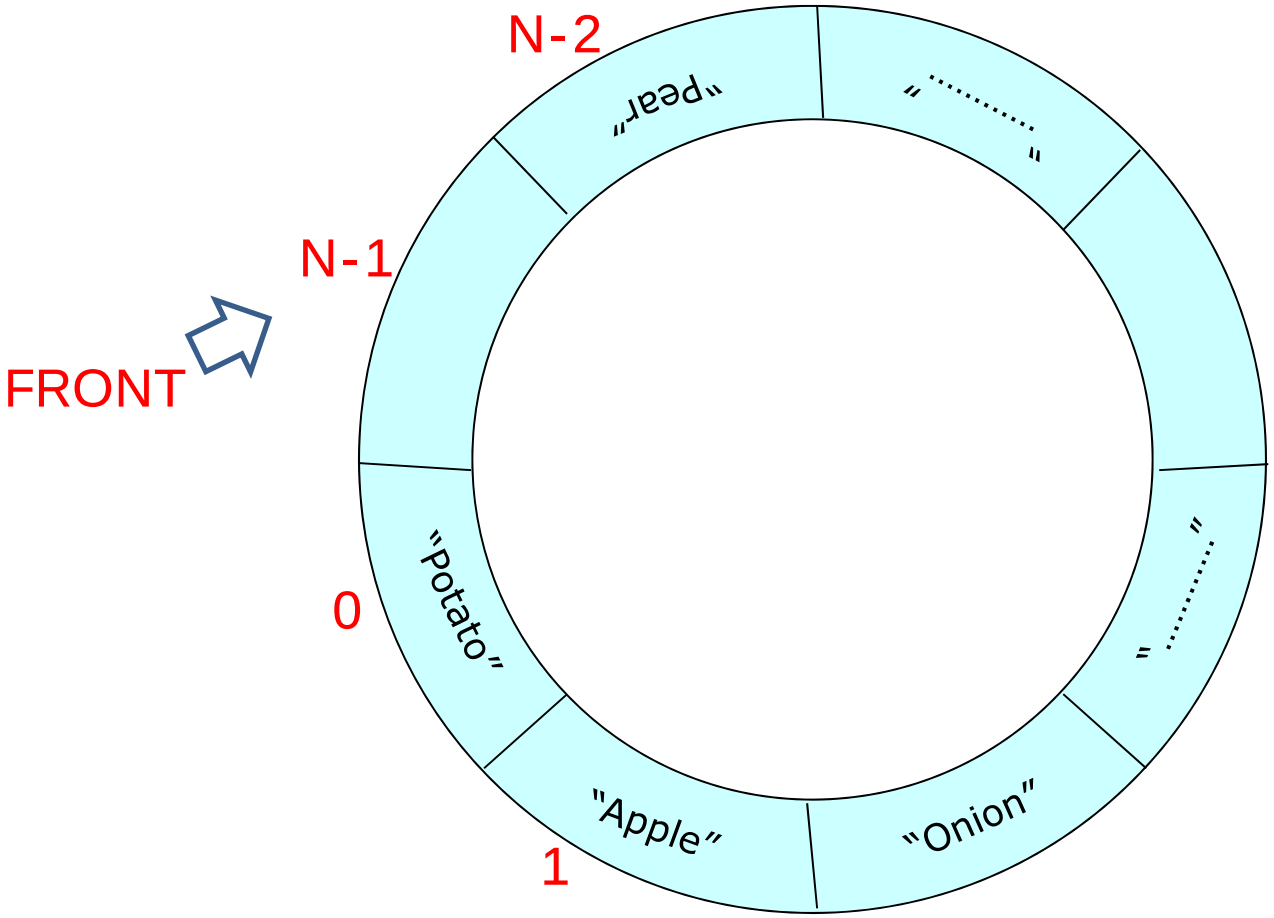
5.1 Circular queue

– Connecting the first and the last element of the queue



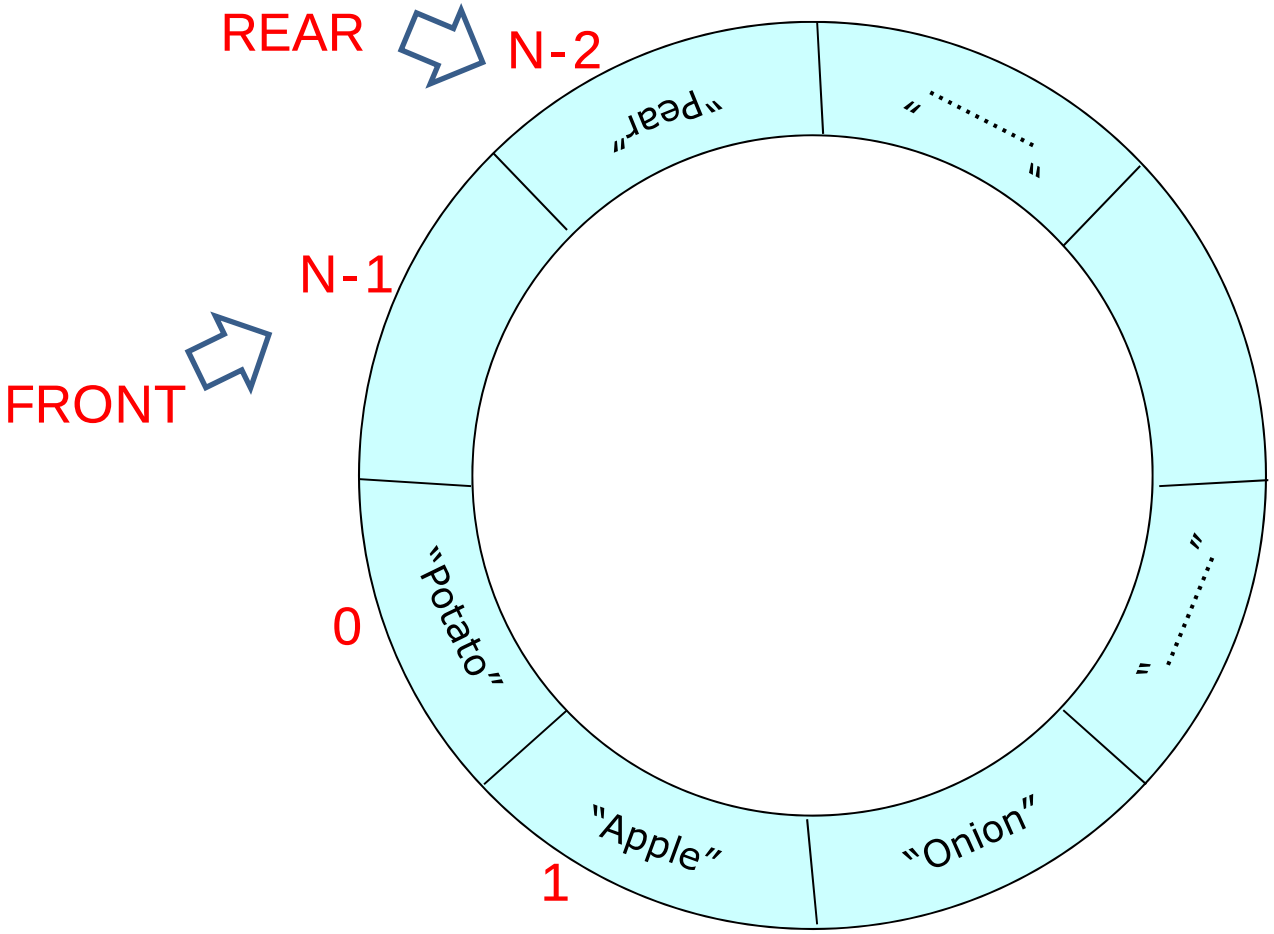
5.1 Circular queue

– (N-1) times AddQ ( )



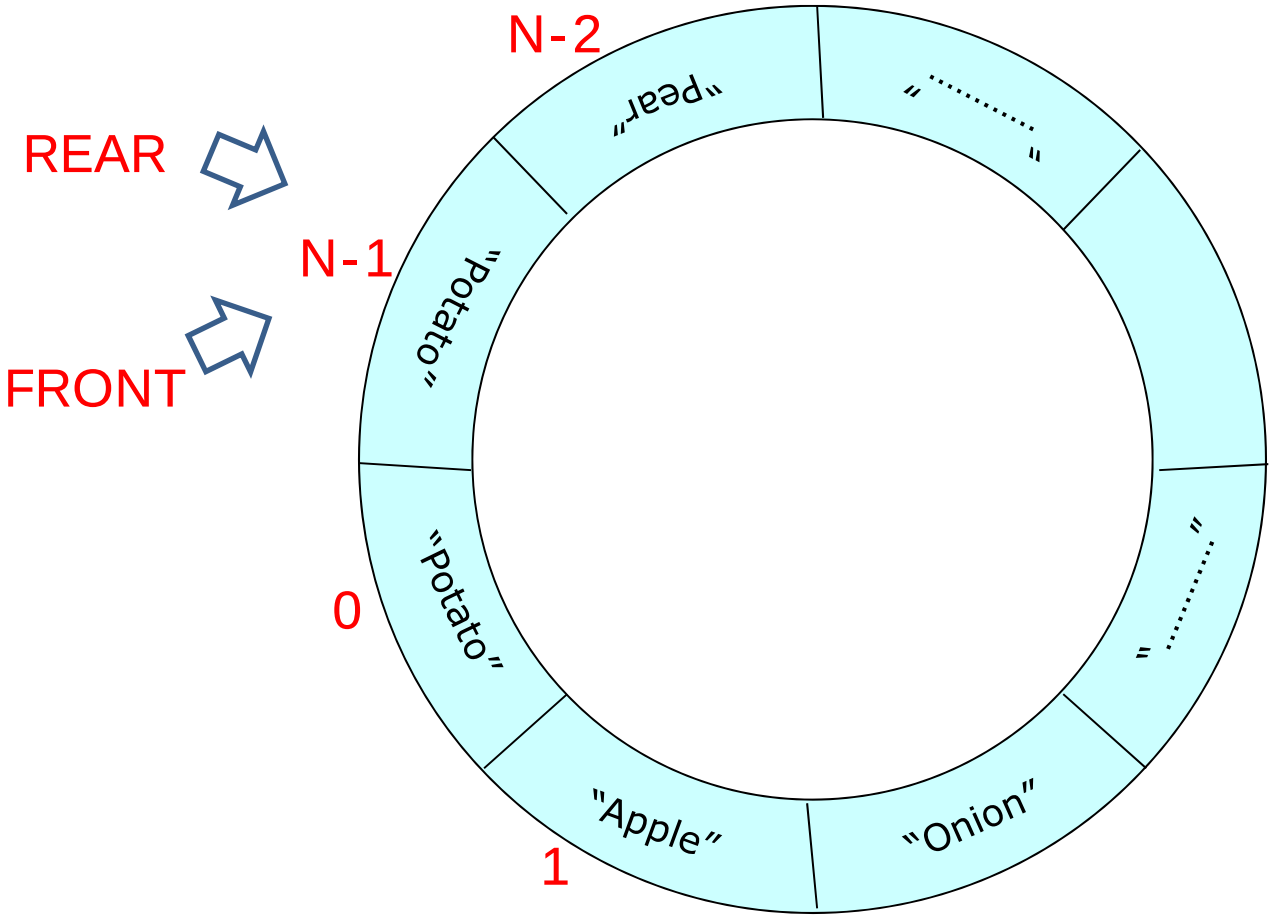
5.1 Circular queue

– (N-1) times AddQ ( )



5.1 Circular queue

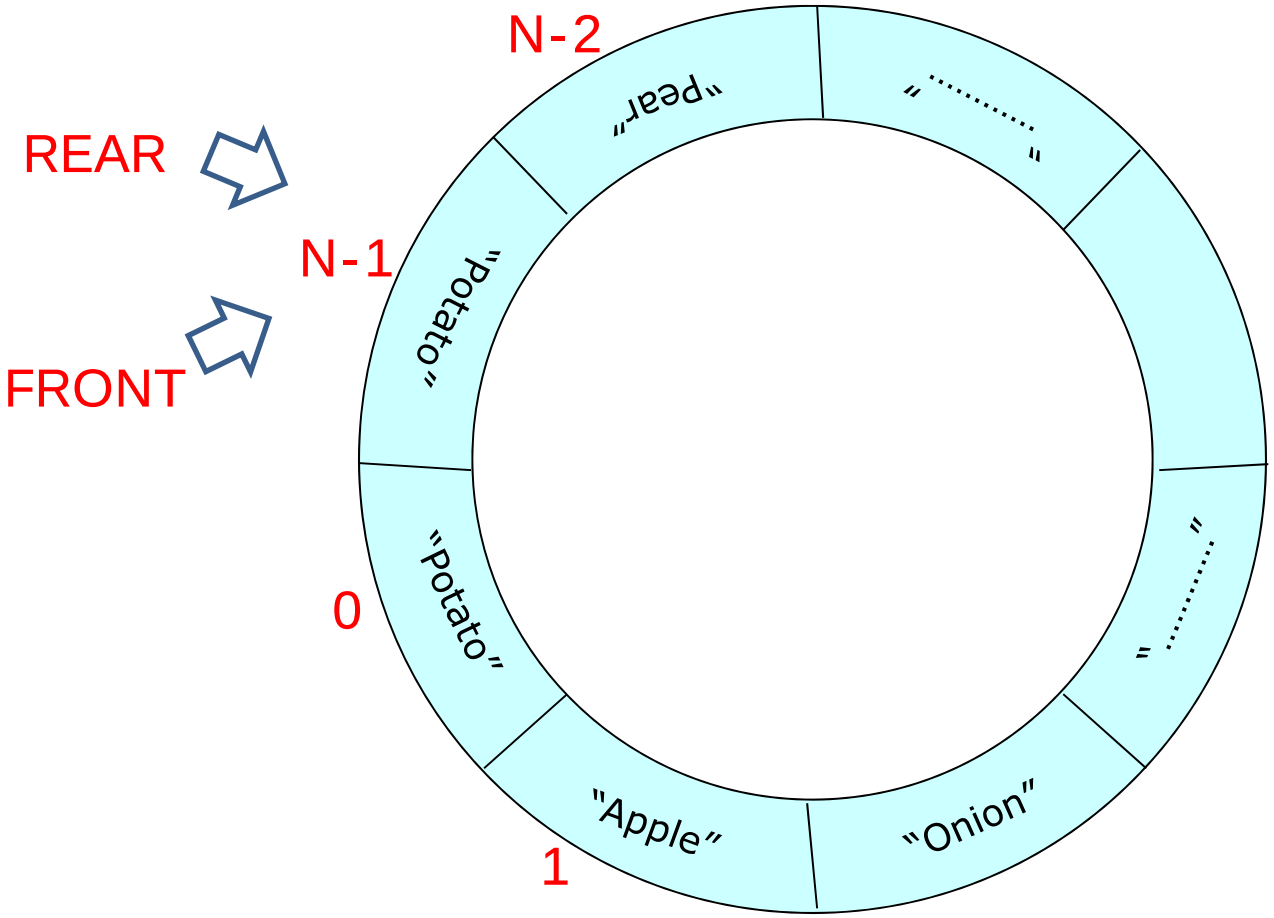
– One more AddQ ( ) ➔



5.1 Circular queue

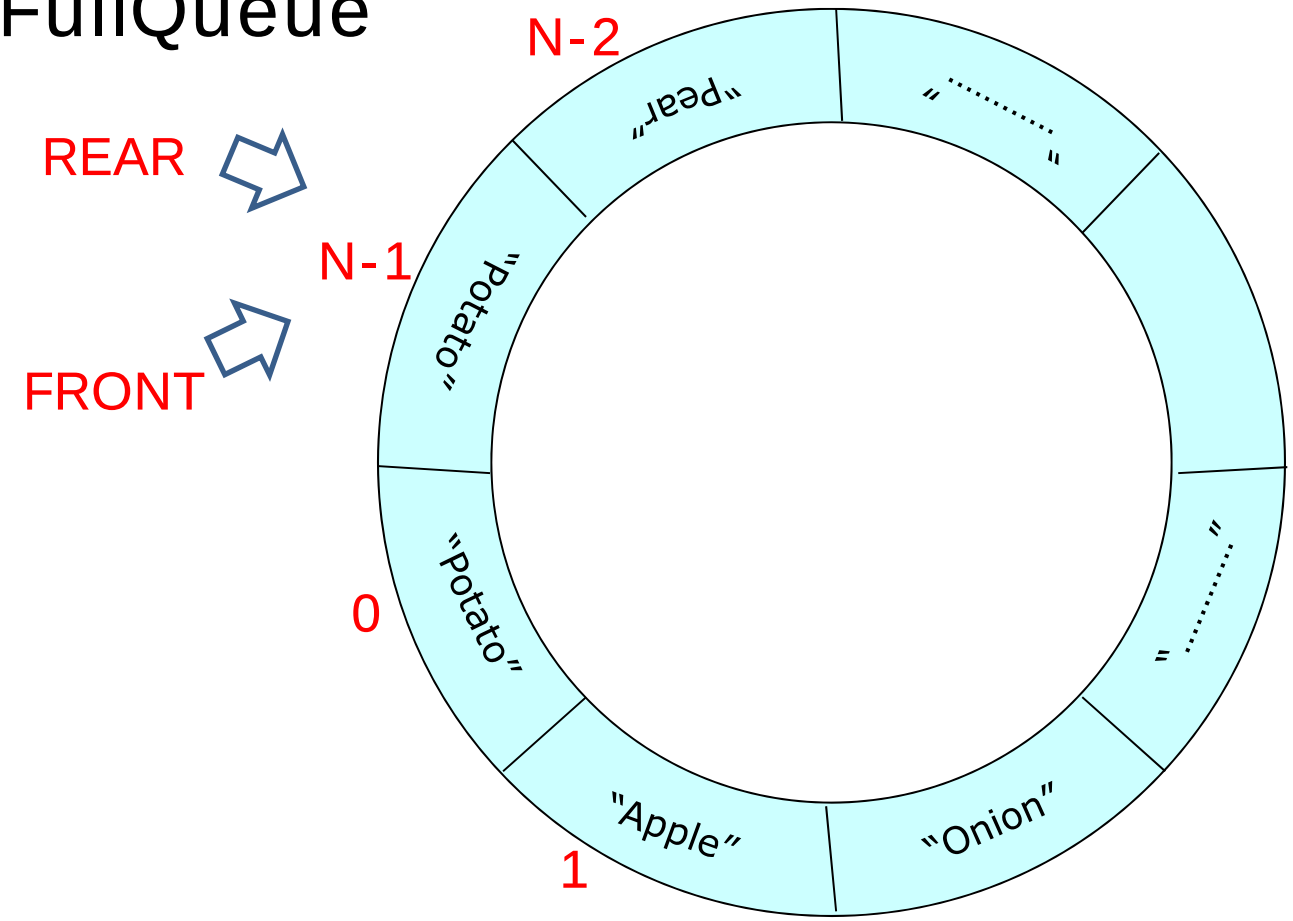
– One more AddQ ( ) ➔ REAR == FRONT

➔



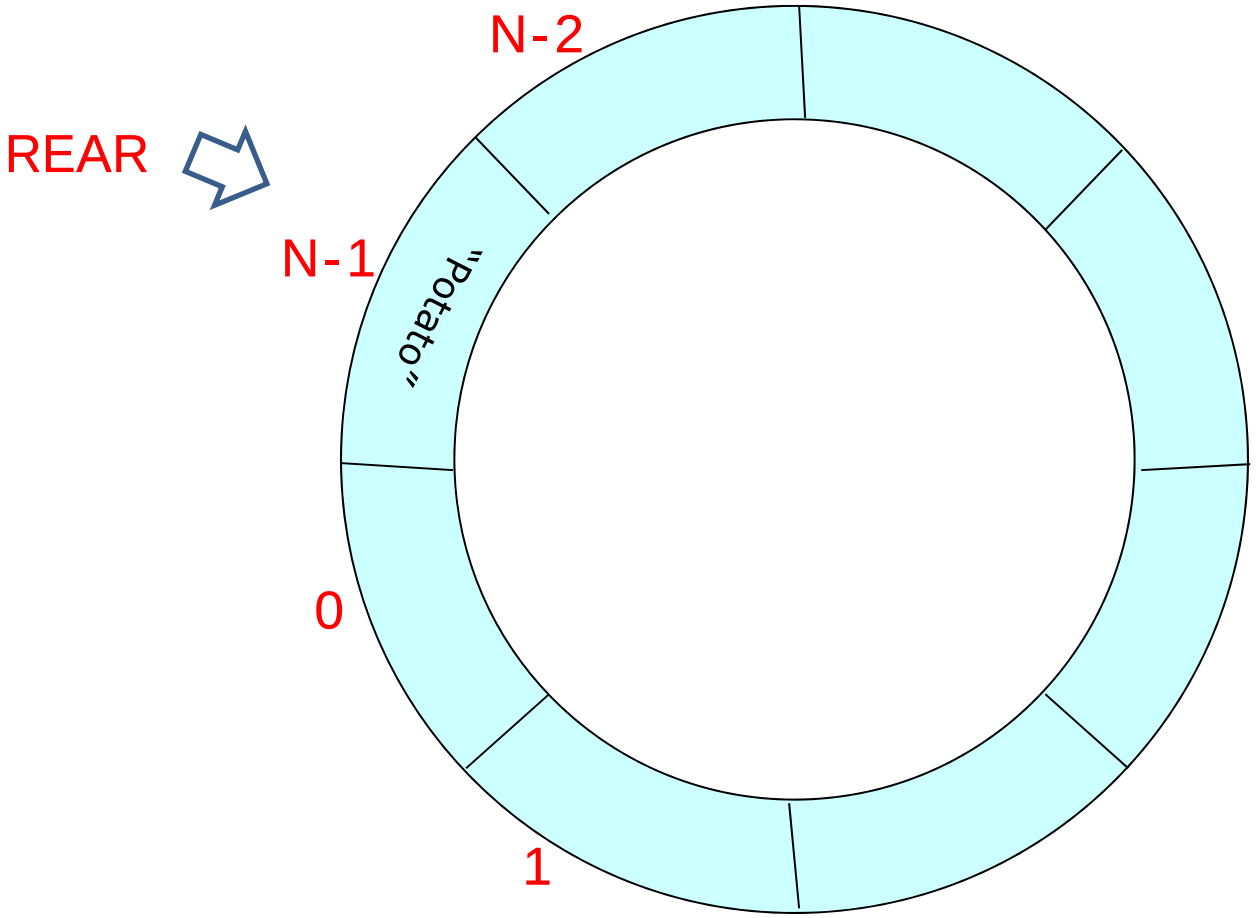
5.1 Circular queue

– One more AddQ ( ) ➔ REAR == FRONT
➔ FullQueue



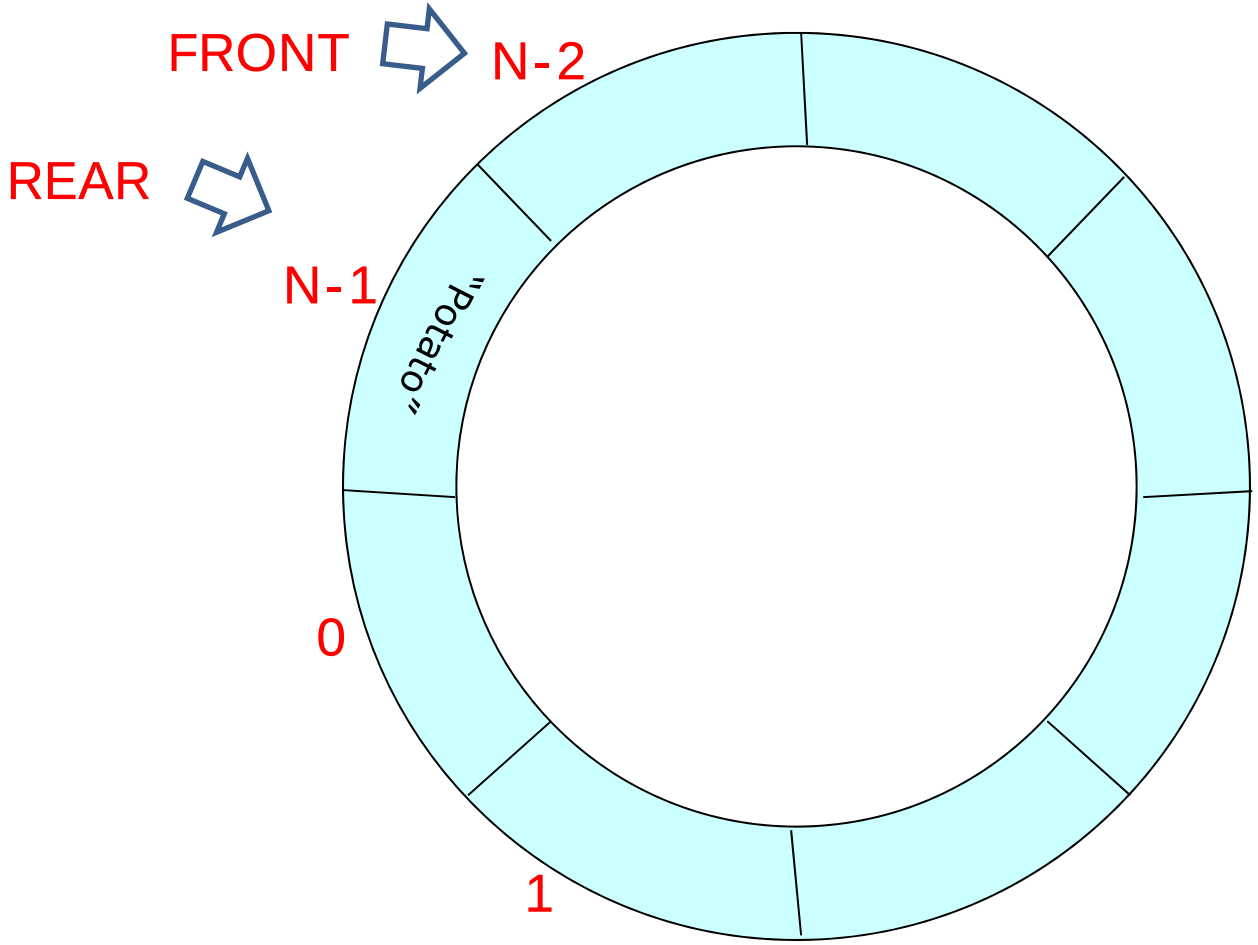
5.1 Circular queue

– (N-1) times DeleteQ ( )



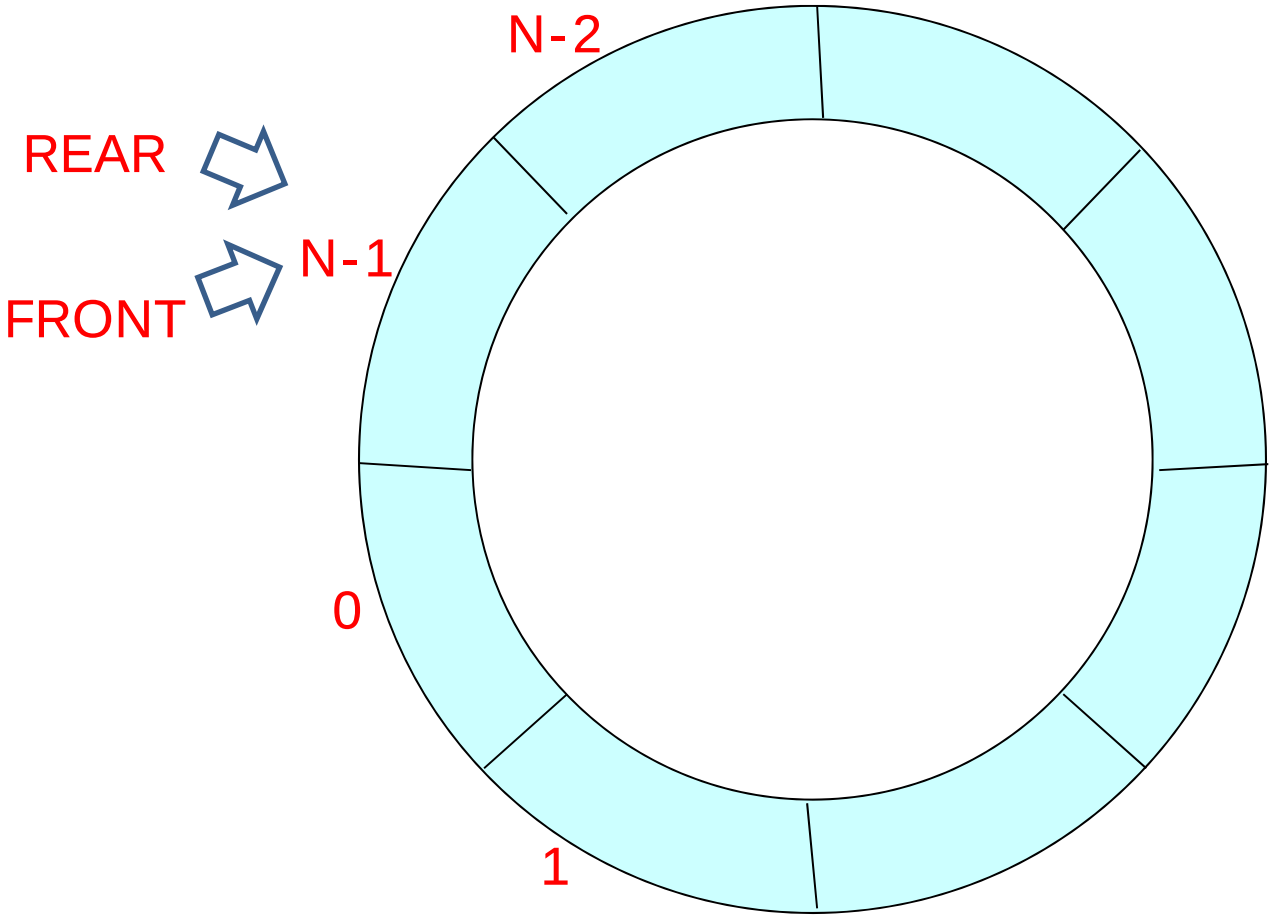
5.1 Circular queue

– (N-1) times DeleteQ ( )



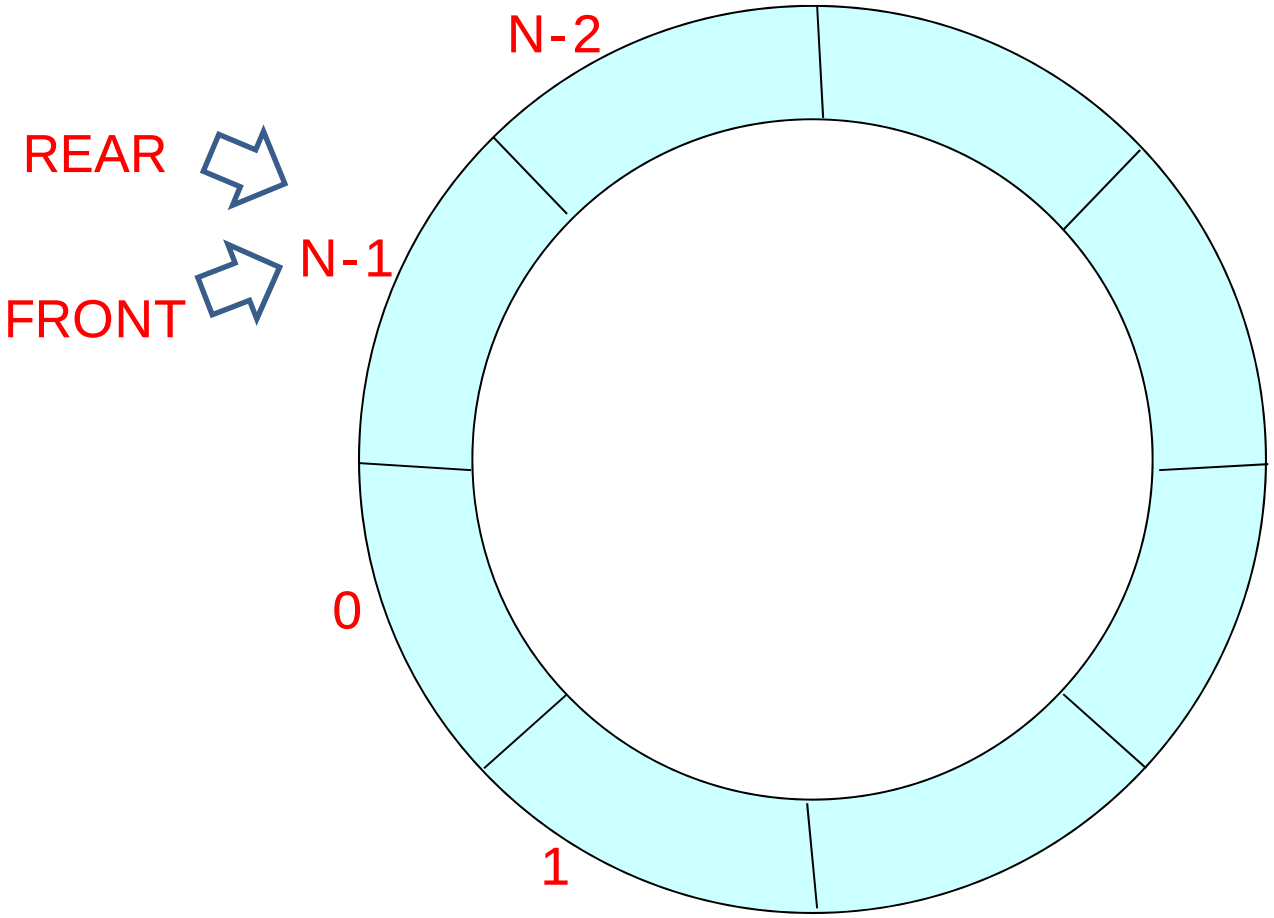
5.1 Circular queue

– One more DeleteQ ( ) ➔



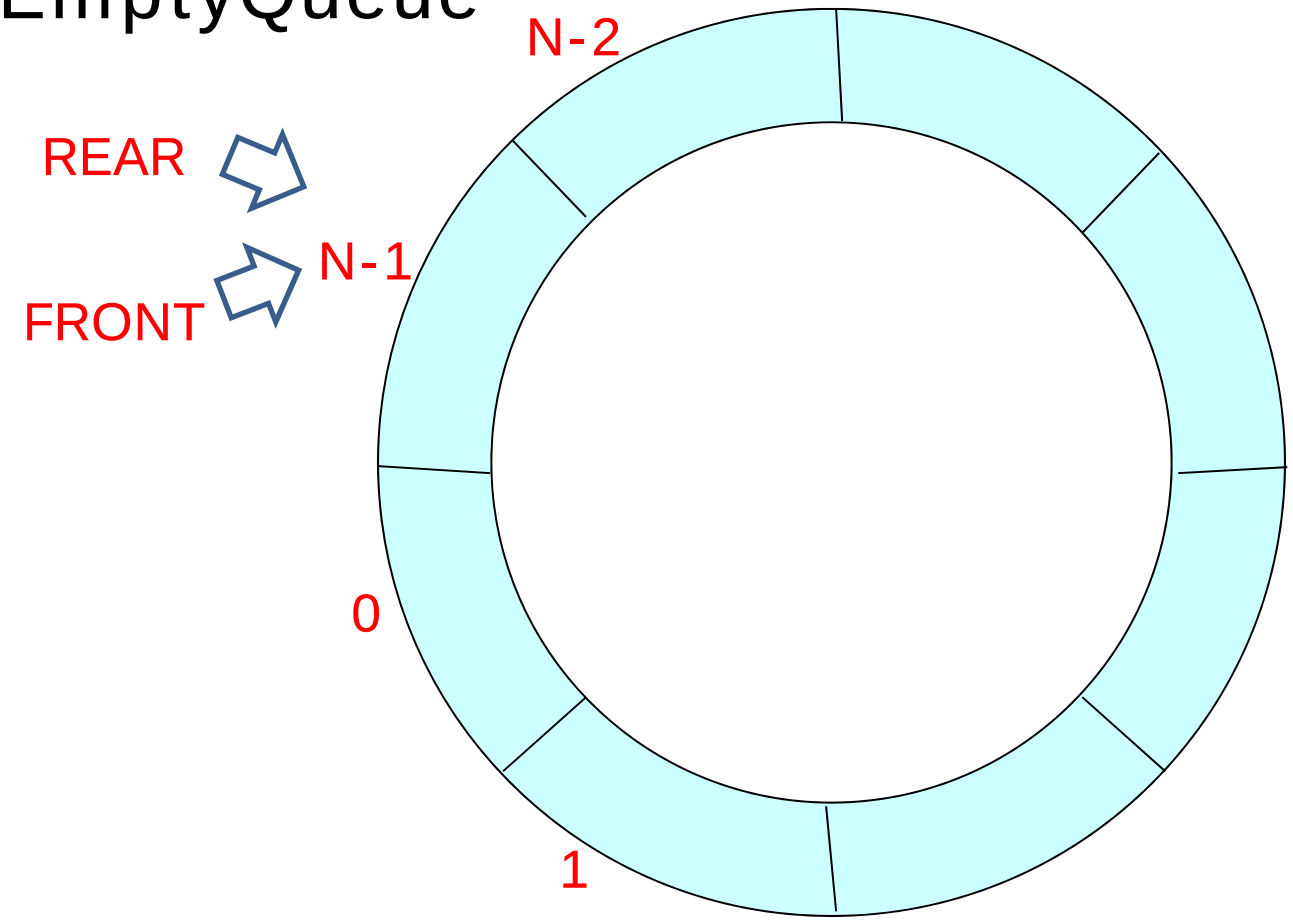
5.1 Circular queue

– One more DeleteQ ( ) ➔ REAR == FRONT ➔



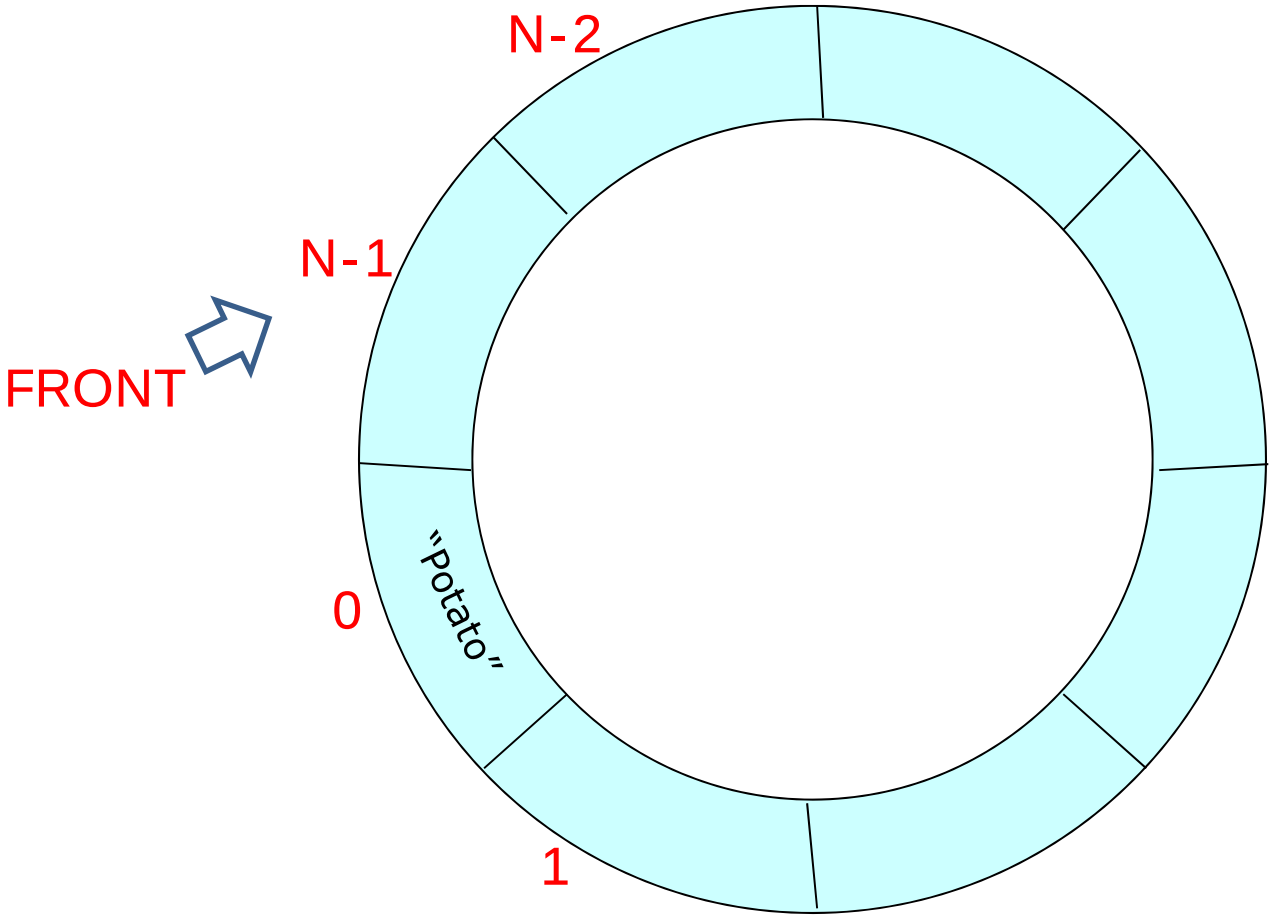
5.1 Circular queue

– One more DeleteQ ( ) ➔ REAR == FRONT ➔ EmptyQueue



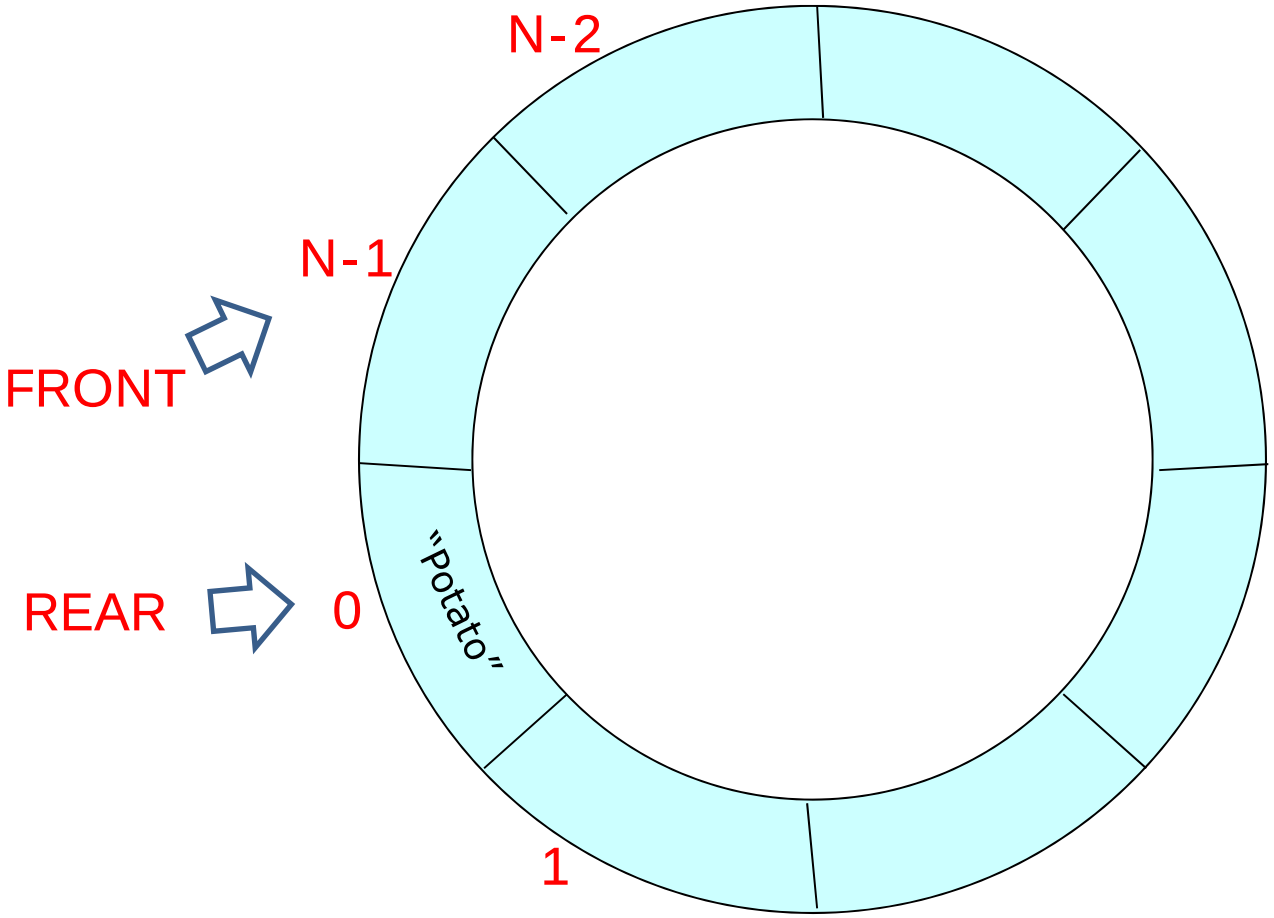
5.1 Circular queue

– One more AddQ ( )



5.1 Circular queue

– One more AddQ ( )



5.1 Circular queue

- Advantage
  - Infinite AddQ ( ) & DeleteQ ( ) is allowed
- Disadvantage
  - Condition for FullQueue

    [FRONT == REAR]
  - Condition for EmptyQueue

    [FRONT == REAR]

5.1 Circular queue

- Same condition for FullQueue and EmptyQueue !!
  - We need another variable “No_of_element” that notifies the number of elements in the circular queue

5.1 Circular queue

- Data structure

```
Class queue{
        int size;
        DataType *Items;
        int rear, front;
       int No_of_element;
};
```

5.1 Circular queue

- isFull ( ) for circular queue

```
void queue::isFull ( )
{
        return ( No_of_element == Size );
}
```

5.1 Circular queue

- isEmpty ( ) for circular queue

```
void queue::isEmpty ( )
{
        return ( No_of_element == 0 );
}
```

5.1 Circular queue

- AddQ ( ) for circular queue

```
void queue::AddQ ( )
{
        if ( isFull ( ) )
                printf("Cannot add an element to a full queue);
        rear = (rear+1)%size;
        Items[rear] = item;
        No_of_element++;
}
```

5.1 Circular queue

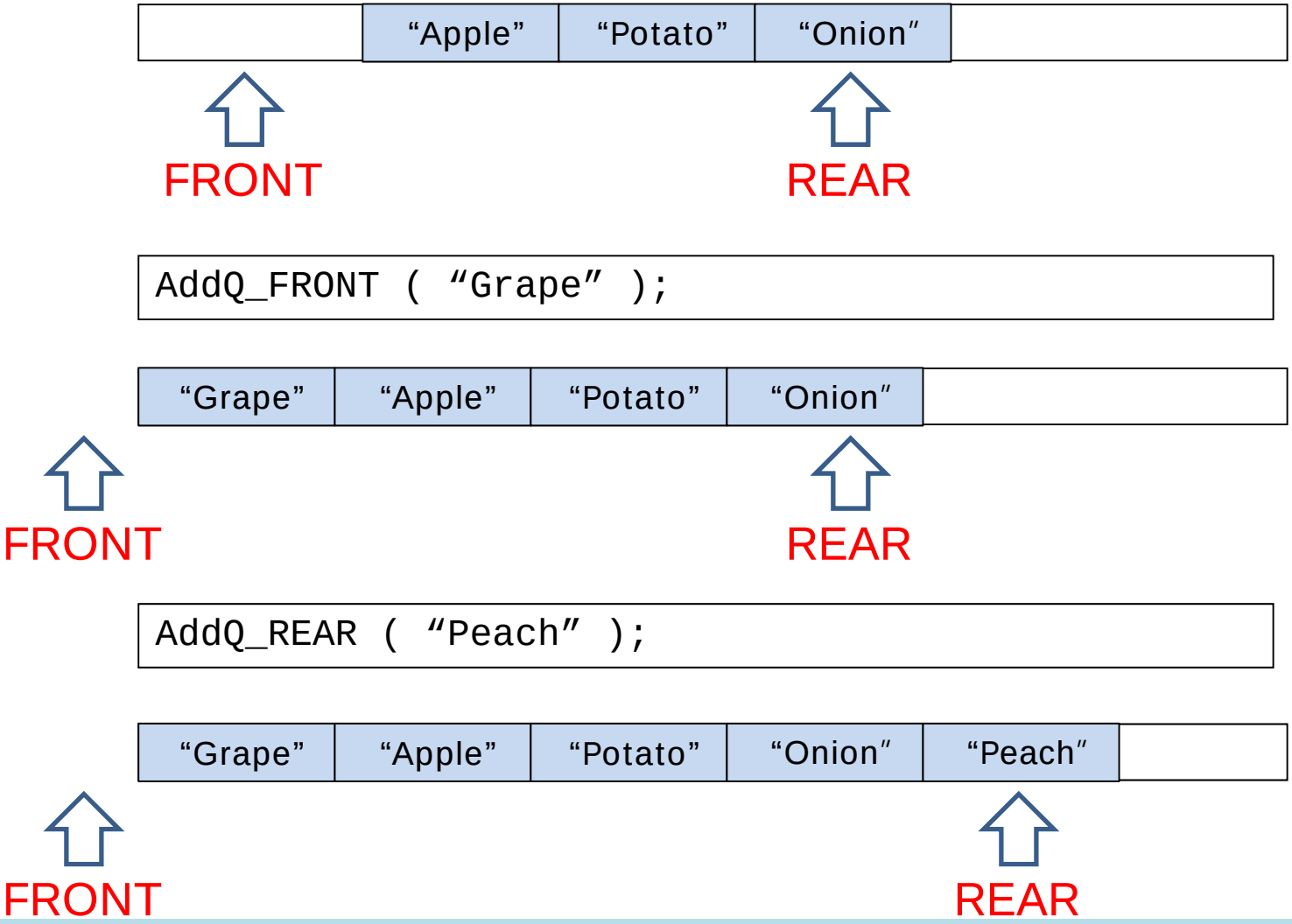
- DeleteQ ( ) for circular queue

```
element queue::DeleteQ ( )
{
        if ( isEmptyQ ( ) )
                printf("You cannot delete from an empty queue);
        No_of_element--;
        front = (front + 1)%size;
        return Items[front];
}
```

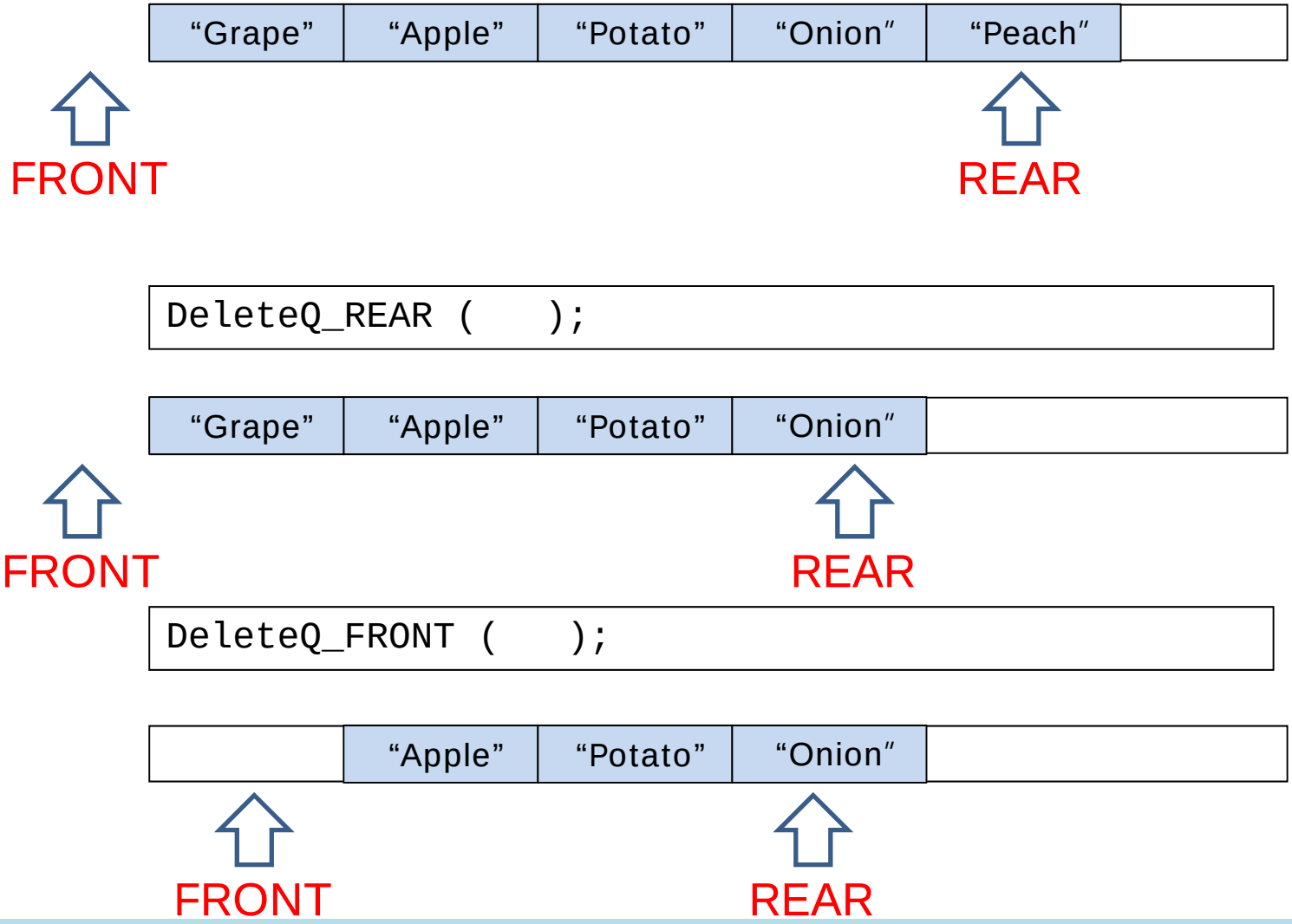
5.2 DEQ

- Doubley-Ended Queue
- A queue whose AddQ and DeleteQ is allowed at both ends of the queue
  - **AddQ_FRONT ( );**
    - Add a new element at FRONT
  - AddQ_REAR ( );
    - Add a new element at REAR
  - DeleteQ_FRONT ( );
    - Delete an element at FRONT
  - **DeleteQ_REAR ( );**
    - Delete an element at REAR

5.2 DEQ


"Apple"
"Potato"
"Onion"
FRONT
REAR
AddQ_FRONT ("Grape");
"Grape"
"Apple"
"Potato"
"Onion"
FRONT
REAR
AddQ_REAR ("Peach");
"Grape"
"Apple"
"Potato"
"Onion"
"Peach"
FRONT
REAR

5.2 DEQ


"Grape"
"Apple"
"Potato"
"Onion"
"Peach"
FRONT
REAR
DeleteQ_REAR ( );
"Grape"
"Apple"
"Potato"
"Onion"
FRONT
REAR
DeleteQ_FRONT ( );
"Apple"
"Potato"
"Onion"
FRONT
REAR

5.3 Priority Queue

- A queue whose deletions are allowed at arbitrary position.
  - The element of highest priority is deleted from the queue
  - Heap

Overall summary

Type	Data structure	Operations		
		search	insert	delete

Array (sorted)	`int size;`
	`int n;`
	`int *arr;`
	4 | 6 | 7 | 9 | |
		linear search ( )
		
		binary search ( )
			1. find the location
			2. move to right (→)
			3. insert an element
			4. increase the count
				1. find the location
				2. move to left (←)
				3. reduce the count

Linked list (sorted)	`class node {`
	`int element;`
	`node *link;`
	`}`
	4 → 6 → 7 → 9
		linear search ( )	1. find the location
			2. build a new node
			3. change the links
				1. find the location
				2. change the links
				3. free a delete node

Stack	`Class stack {`
	`int size;`
	`int *Items;`
	`int TOP;`
	`}`
	7 | 4 | 6 | | |
		No search operation	**Push**
			
			`Items[TOP++] = item`
				Pop
				
				`return Items[--Top];`

Queue	`Class queue {`
	`int size;`
	`int *Items;`
	`int front, rear;`
	`}`
	| 4 | 6 | | |
		No search operation	**AddQ**
			
			`Items[++rear] = item;`
				DeleteQ
				
				`return Items[++front];`

Contents