Data structures and algorithms Spring 2025

STACK AND QUEUE

Lecturer: Do Thuy Duong

Contents

- STACK ADT
- Stack implementations
- Queue ADT
- Queue implementations

Stack

What **it is** (conceptual)

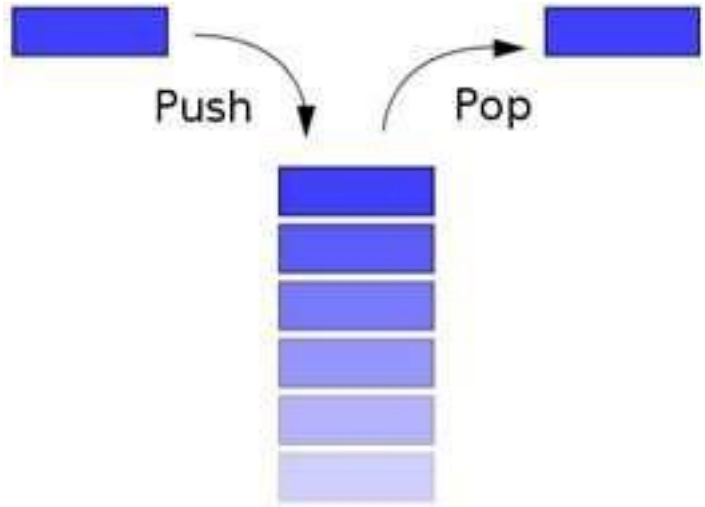
How **we implement it** (implementation)

Why **we use it** (applications)

Stack ADT

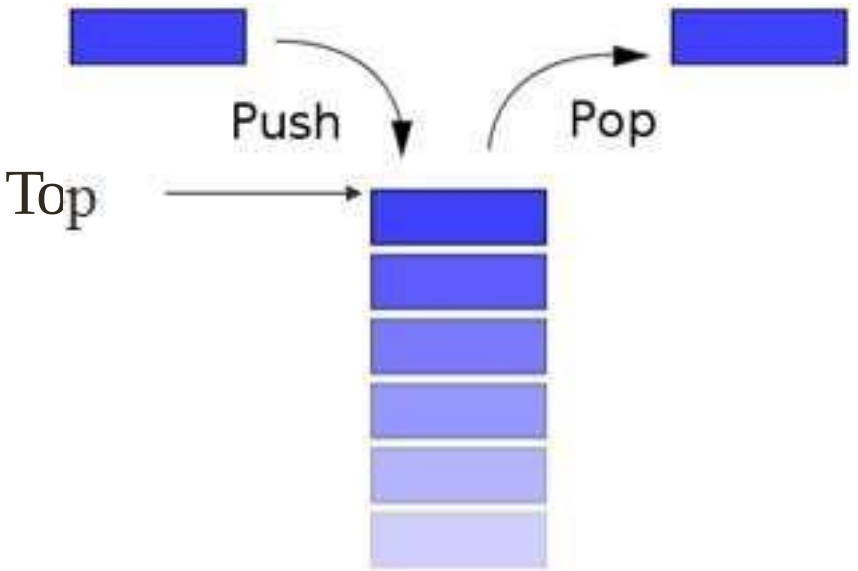
Definition:

- Stack is a linear sequence (**list**) for which all insertions and deletions (and usually all accesses) are made at **one end of the list**. (**LIFO: Last-in-first-out**)
- Also called:
  - Last-in-first-out (LIFO) list
  - Pushdown list.



Stack ADT

- The position that insertions and deletions can be performed at the end of the list, called the **Top**.
- Data:
  - Primitive data type or other (object)
- The 2 basic operations:
  - **push (insert)**
  - **pop (delete)**

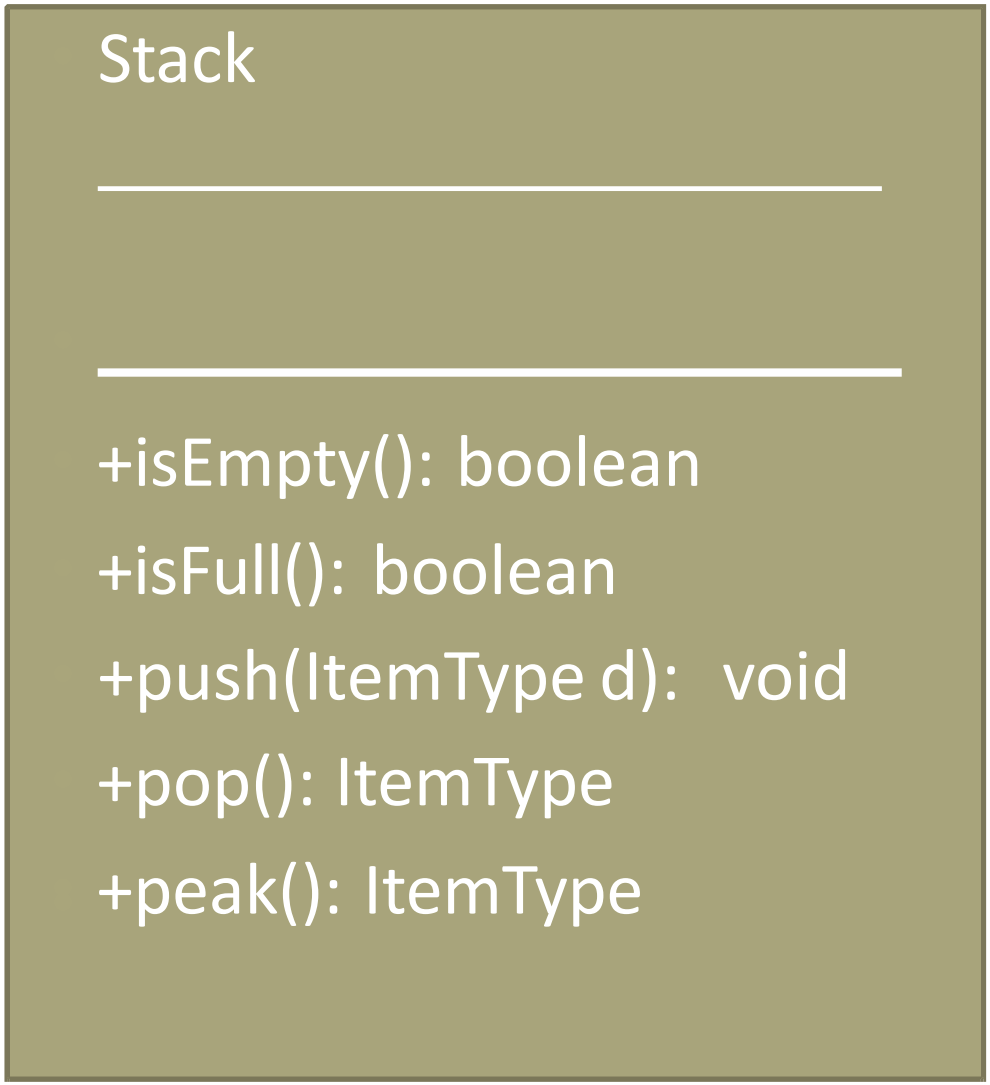


Stack ADT

Stack operations:

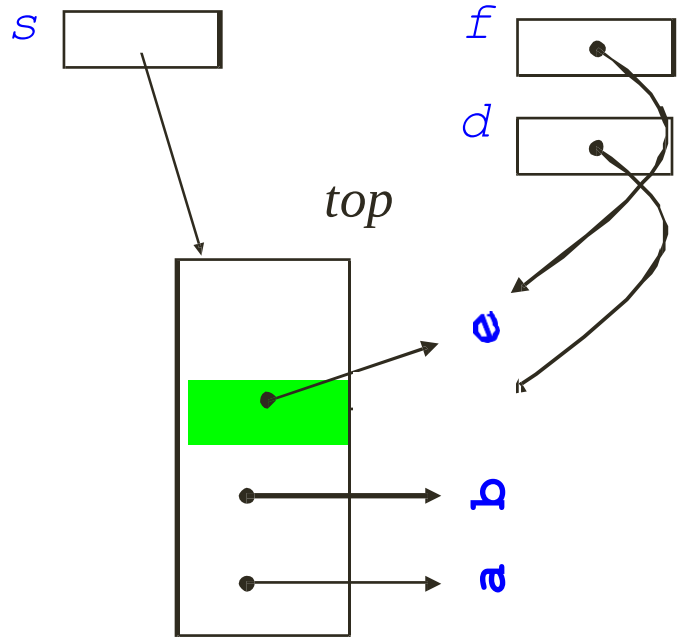
- Create an empty stack.
- Determine whether a stack is empty.
- Determine whether a stack is full.
- Add a new item into the stack (**Push**).
- Take out the most recently added (**Pop**).
- Retrieve the item most recently added (**Peak**)

Stack ADT


Stack
+isEmpty(): boolean
+isFull(): boolean
+push(ItemType d): void
+pop(): ItemType
+peak(): ItemType

Stack ADT: Example

- Stack s = new Stack();
- s.push(a);
- s.push(b);
- s.push(c);
- d = s.pop();
- s.push(e);
- f=s.peak();

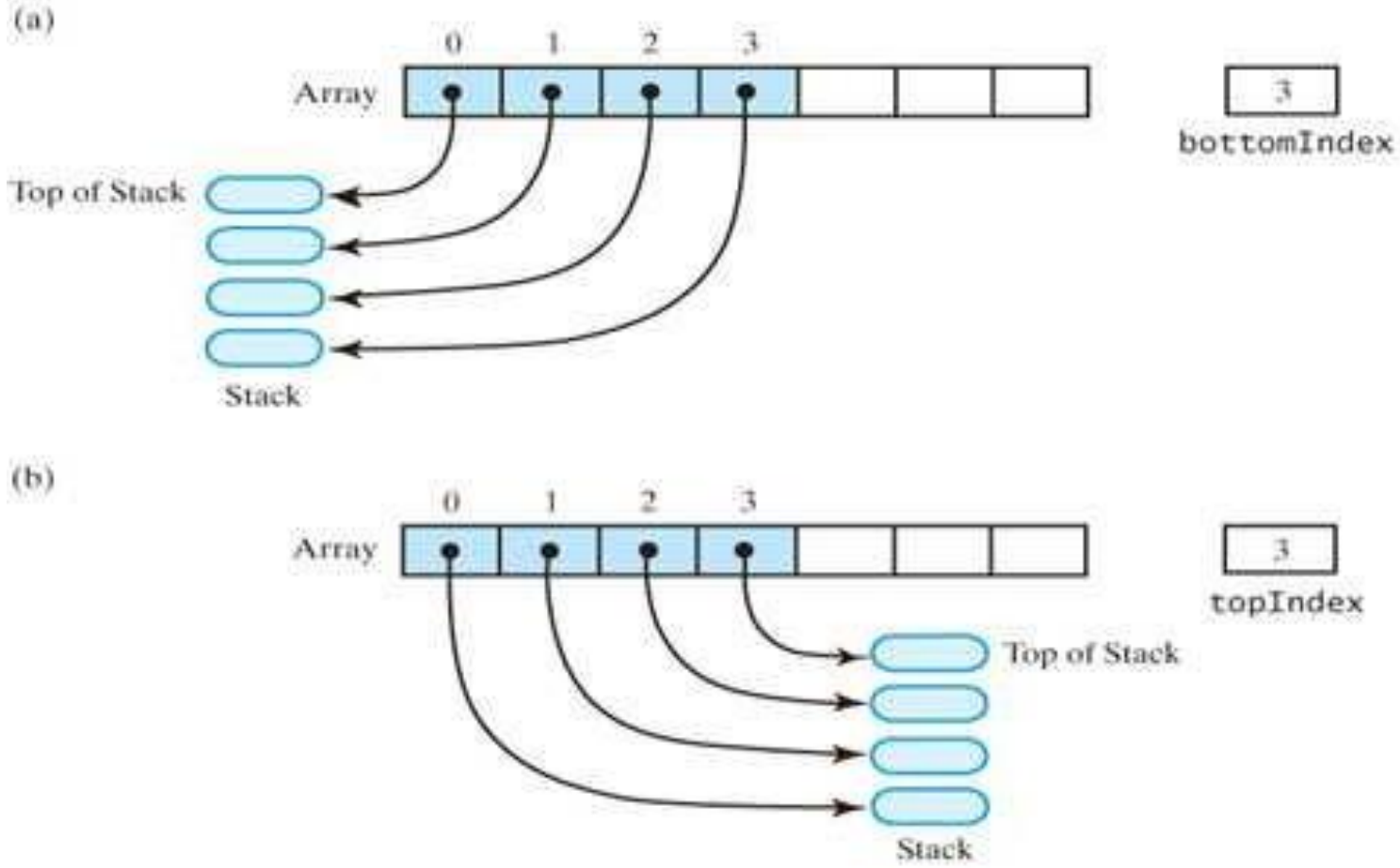


Stack Implementation

- Array-based implementation
- List-based implementation

Array-based stack [1]

- **Array-based:**
  - An array is used for storing stack items

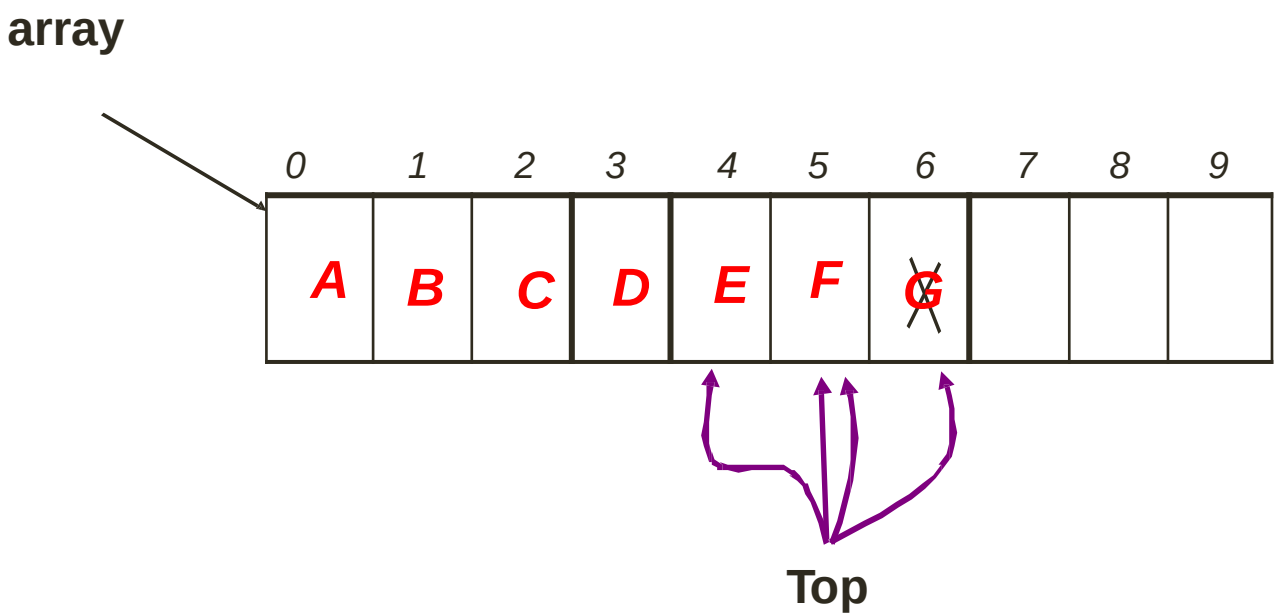


Array-based stack [2]

- **Array-based:**
  - An array is used for storing stack items
  - The array's **first element should represent the bottom of the stack**
  - The **last occupied location in the array represents the stack's top**
- This avoids shifting of elements of the array when we push or pop items from stack

Array-based stack [3]

- **Array-based:**



Array-based stack [4]

- **isEmpty(): boolean**
- **isFull(): boolean**
- **Push(item): void**

Array-based stack [4]

- **isEmpty(): boolean**
  - If top equals -1
- **isFull(): boolean**
  - If top equals maxSize-1
- **Push(item): void**
  - if stack is not full
    - Increate top
    - Add item to stack

```
if (top == -1)
        return true;
else
        return false;
```

```
if (top == maxSize-1)
        return true;
else
        return false;
```

```
if (!isFull())
{
        top++;
        s[top]=item;
}
```

Array-based stack [5]

- **Pop(): ItemType**
- **Peak(): ItemType**

Array-based stack [5]

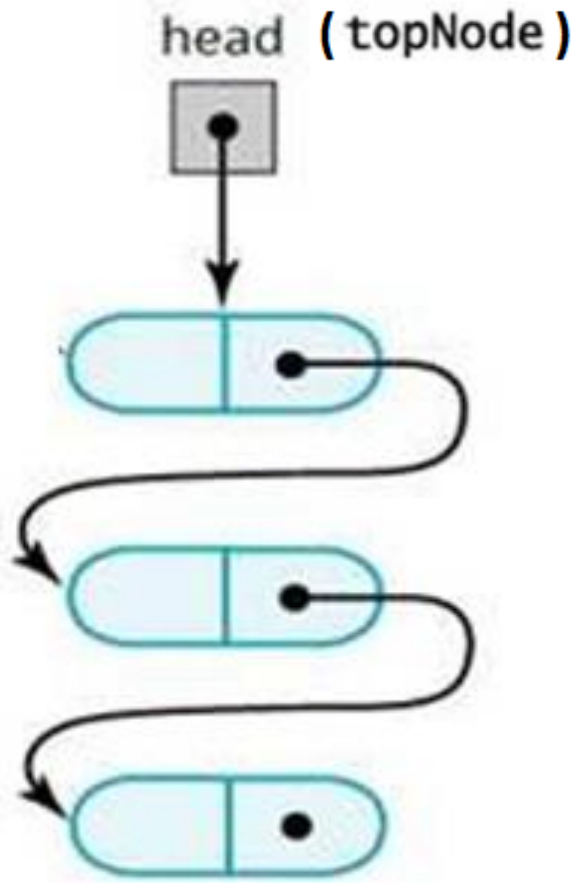
- **Pop(): ItemType**
  - If stack is not empty
    - Take out and return item at the top position
    - Decrease top
- **Peak(): ItemType**
  - If stack is not empty
    - Return item at the top position

```
if (!isEmpty())
{
        tmp=s[top];
        top--;
        return tmp;
}
else
        return null;
```

```
if (!isEmpty())
        return s[top];
else
        return null;
```

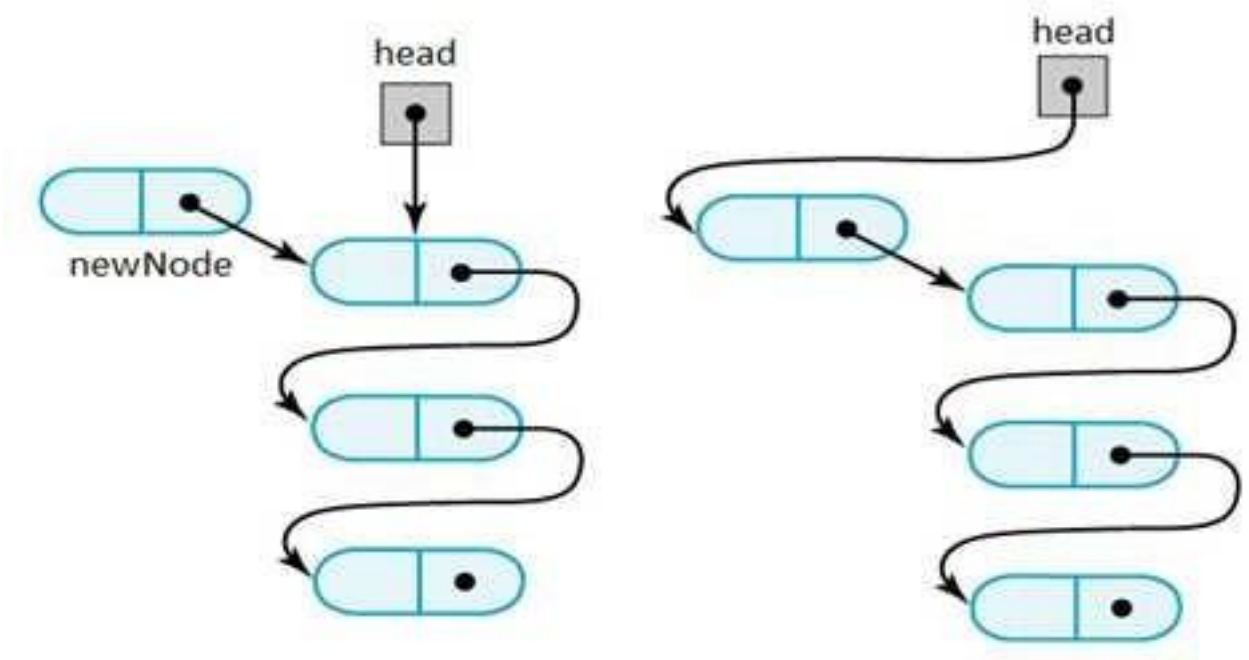
List-based stack[1]

- Using a SL List to implement a stack with each node reference one item in stack



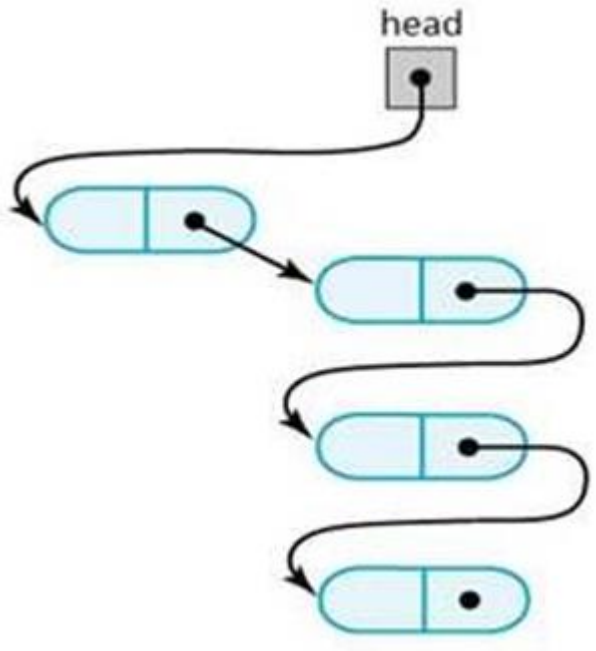
List-based stack[2]

- **The first node should reference the stack's top**:
  - With the head reference, we can add a new node.

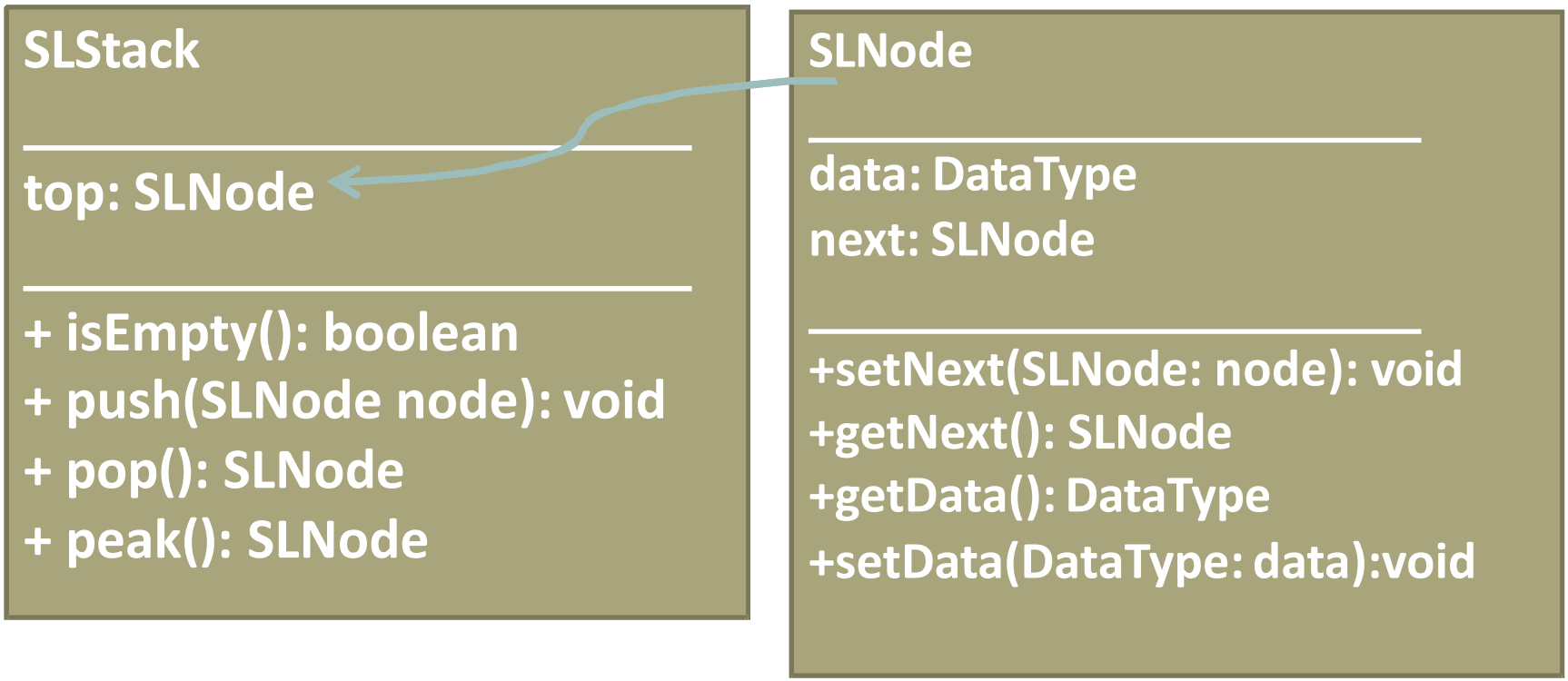


List-based stack[3]

- **The first node should reference the stack's top**:
  - With the head reference, we can remove a node.



List-based stack[4]



List-based stack[5]

- **isEmpty(): boolean**

- **Push(SLNode newNode): void**

List-based stack[5]

- **isEmpty(): boolean**
  - If top equals null

```
if (top == null)
        return true;
else
        return false;
```

- **Push(SLNode newNode): void**
  - Link newNode to the stack
  - Update head reference

```
newNode.setNext(top);
top=newNode;
```

List-based stack[6]

- **Pop(): ItemType**

- **Peak(): ItemType**

List-based stack[6]

- **Pop(): ItemType**
- **Peak(): ItemType**

```
if (!isEmpty())
{
   SLNode topNode=top;
   top=top.getNext();
   return topNode;
}
else
   return null;
```

```
if (!isEmpty())
       return top;
else
       return null;
```

List-based stack[5]

- **isEmpty(): boolean**

- **Push(SLNode newNode): void**

List-based stack[5]

- **isEmpty(): boolean**
  - If top equals null

```
if (top == null)
        return true;
else
        return false;
```

- **Push(SLNode newNode): void**
  - Link newNode to the stack
  - Update head reference

```
newNode.setNext(top);
top=newNode;
```

Queue

What **it is** (conceptual)

How **we implement it** (implementation)

Why **we use it** (applications)

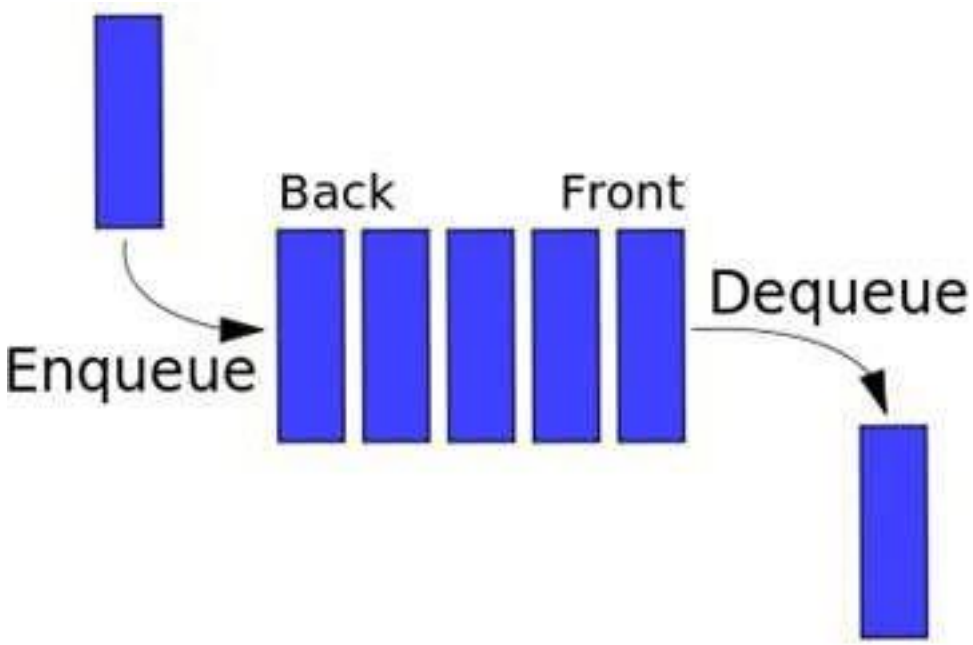
Queue ADT

Definitions:

- A **FIFO Queue: ** is a **linear list** for which all insertions are made at one end of the list (**rear**). All deletions (or all accesses) are made at the other end (**front**).

Queue ADT

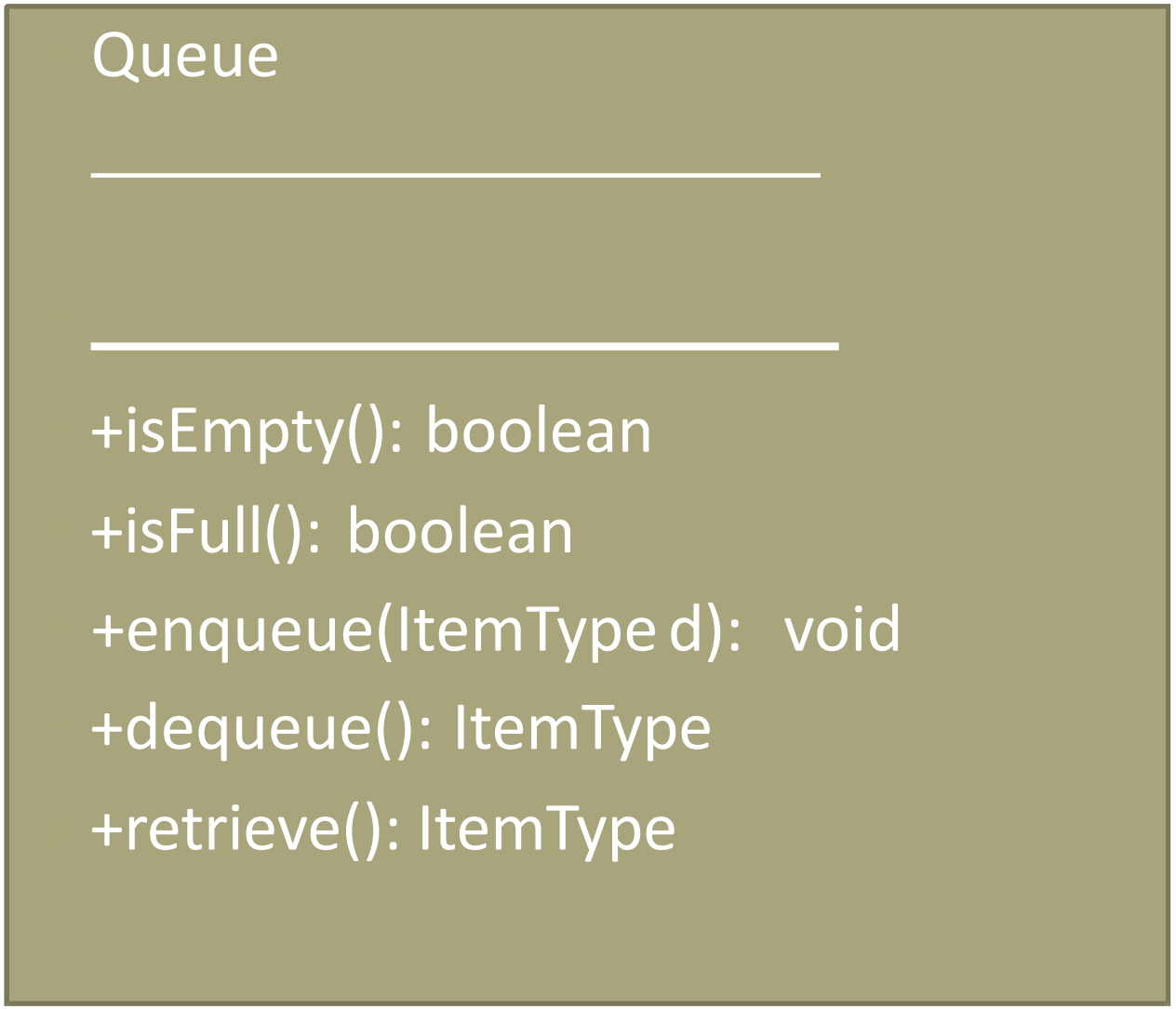
- **FIFO Queue**
  - Front: dequeue() or remove()operation
  - Rear (or Back): enqueue() or insert() operation



Queue ADT

- **Operations on Queue**
  - Create an empty queue
  - Determine whether a queue is empty
  - Determine whether a queue is full
  - Add a new item to the queue (**enqueue**)
  - Remove the item that was added earliest (**dequeue**)
  - Retrieve the item that was added earliest

Queue ADT

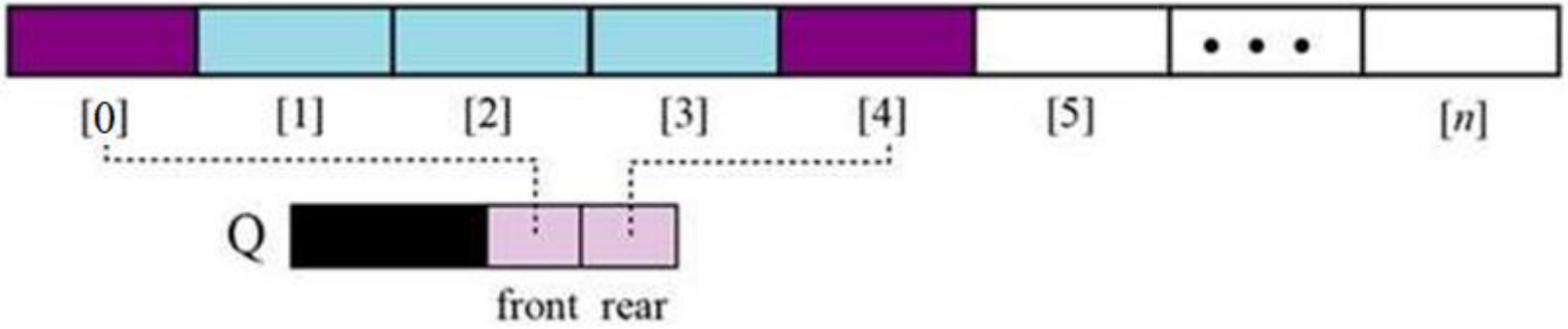

Queue
+isEmpty(): boolean
+isFull(): boolean
+enqueue(ItemType d): void
+dequeue(): ItemType
+retrieve(): ItemType

Queue implementation

- Array-based implementation
- List-based implementation

Array-based Queue [1]

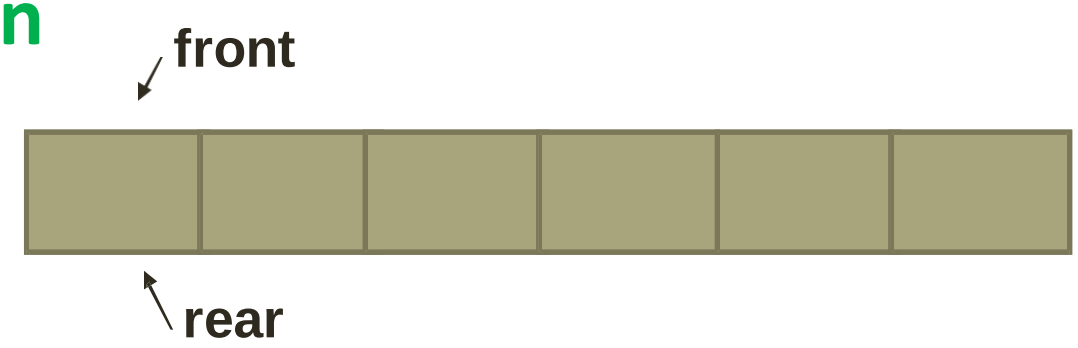
- An array of the size maxSize is used to implement the queue:
  - Two indexing front and rear must be maintained



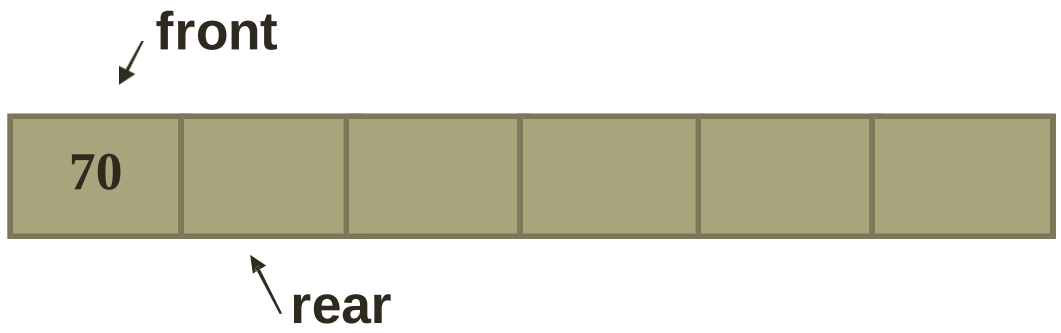
  - When a new item is inserted at the rear, the rear index moves upward.
  - Similarly, when an item is deleted from the queue the front index moves upward.

Array-based Queue [2]

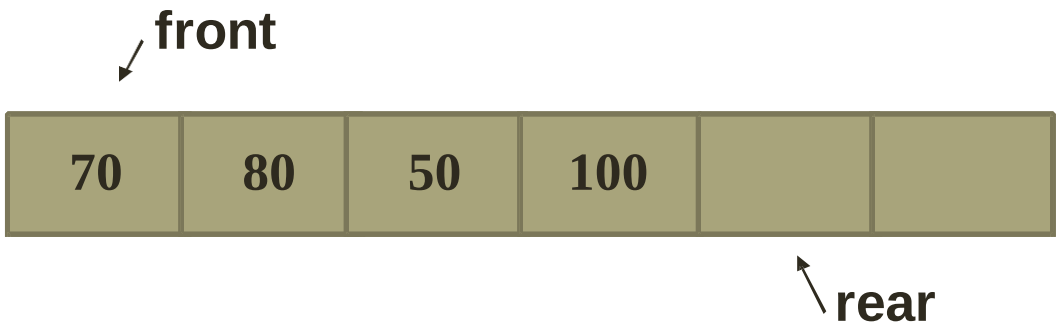
- **Queue operation**
  - Create an empty queue



  - enqueue(70)

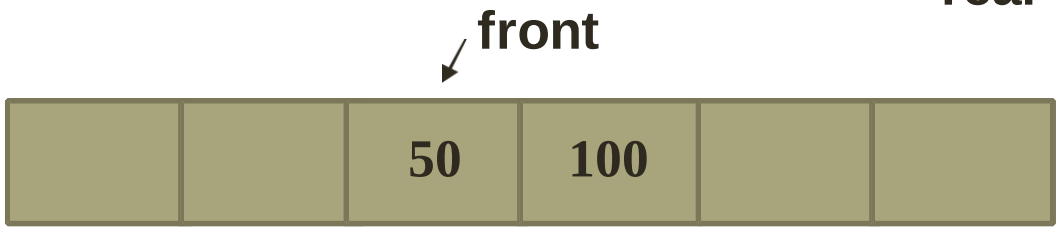


  - enqueue(80)
  - enqueue(50)
  - enqueue(100)

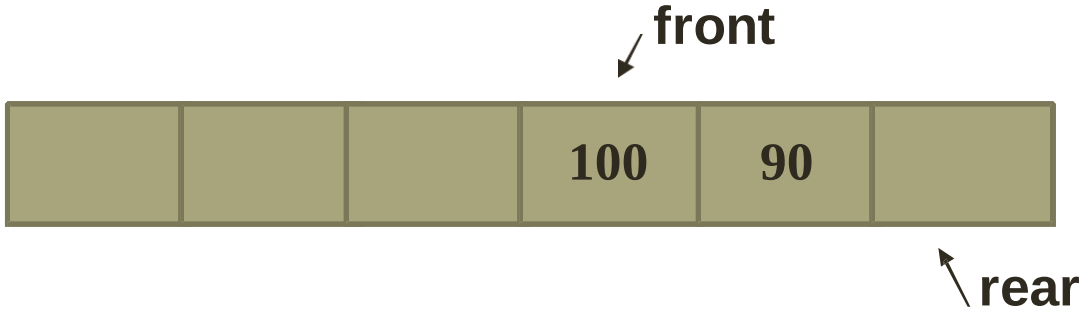


Array-based Queue [3]

- **Queue operation**
  - x=dequeue()
  - y=dequeue()



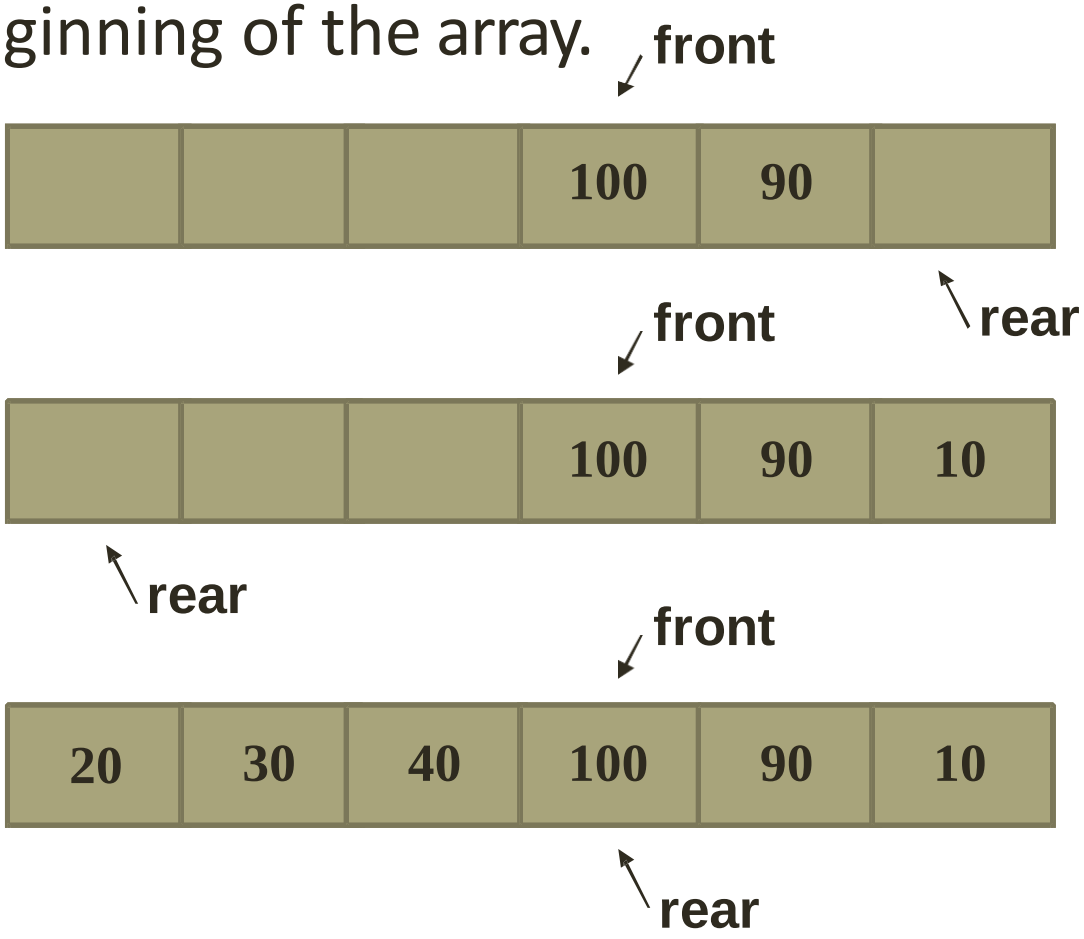
  - enqueue(90)
  - z=dequeue()



  - After a few enqueue and dequeue operations the **rear might reach the end of the queue** and no more items can be inserted although there is space in the queue.

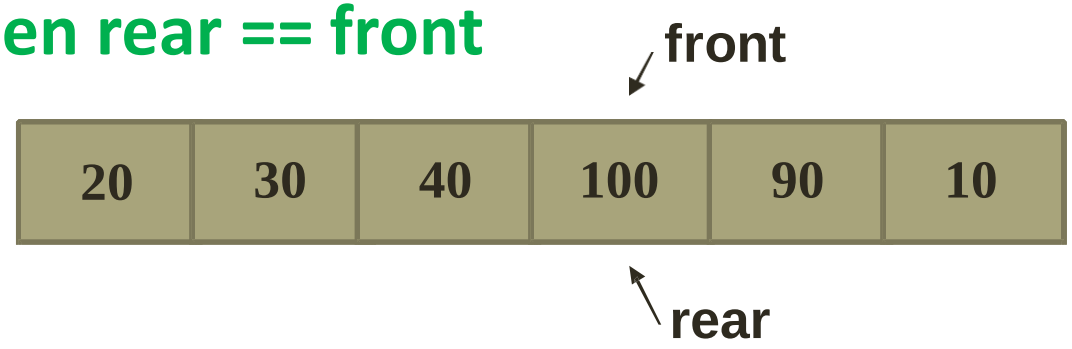
Array-based Queue [4]

- **Circular queue**
  - Allow both the front and the rear index **wrap around** to the beginning of the array.
  - enqueue(10)
  - enqueue(20)
  - enqueue(30)
  - enqueue(40)

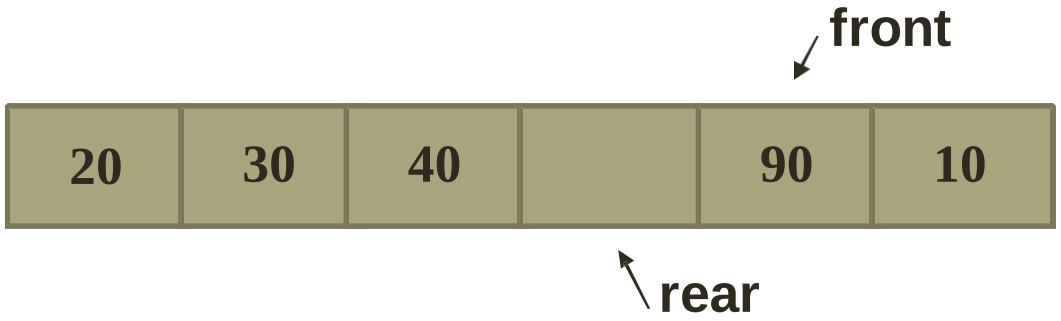


Array-based Queue [5]

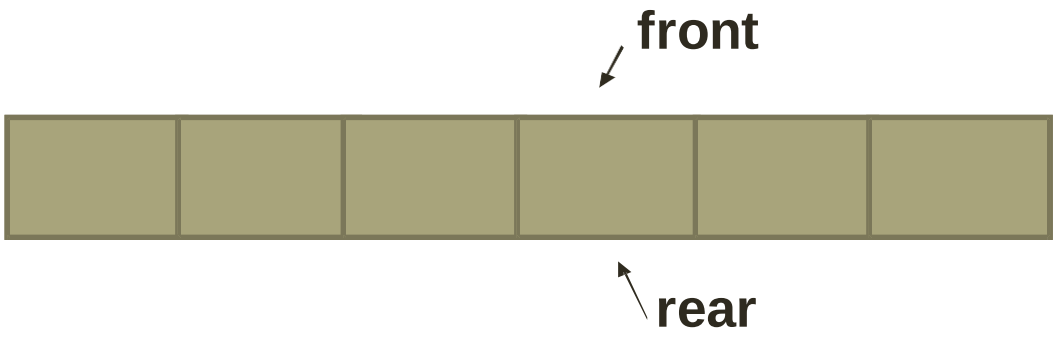
- **Circular queue**
  - **Queue is full when rear == front**



  - x1=dequeue()

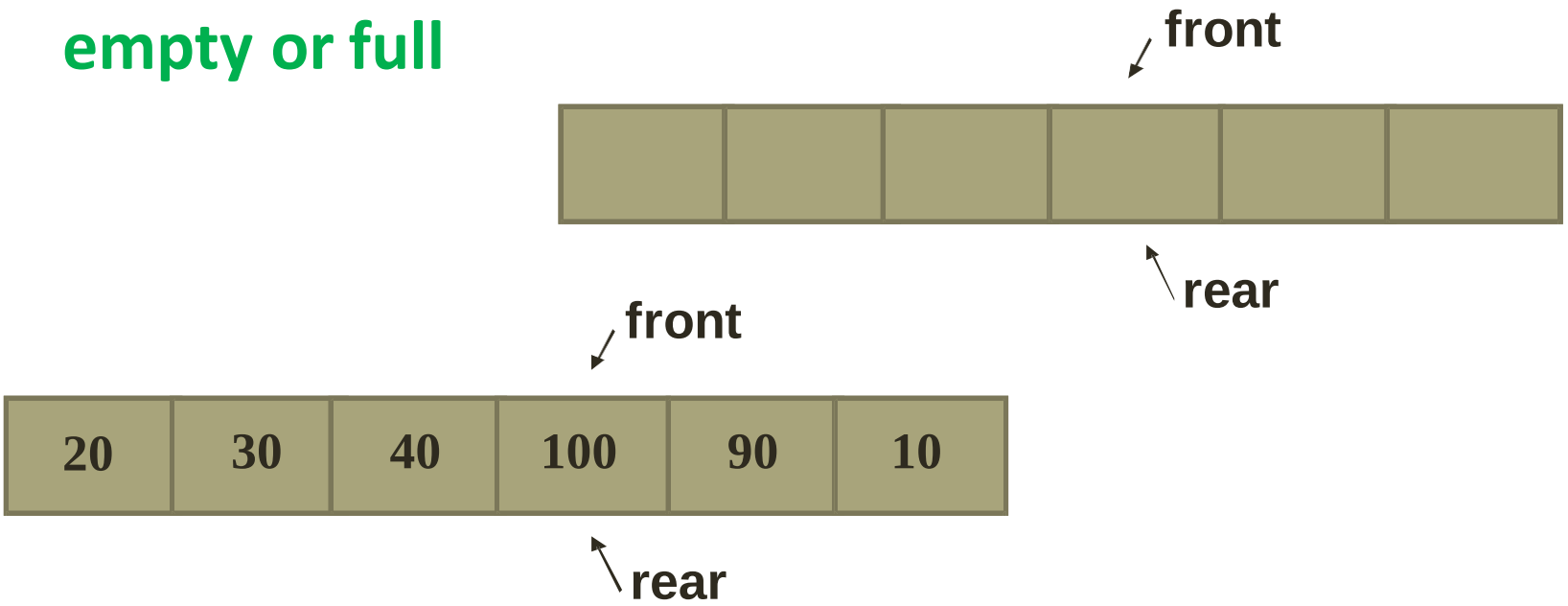


  - x2=dequeue()
  - …
  - x6=dequeue()



Array-based Queue [6]

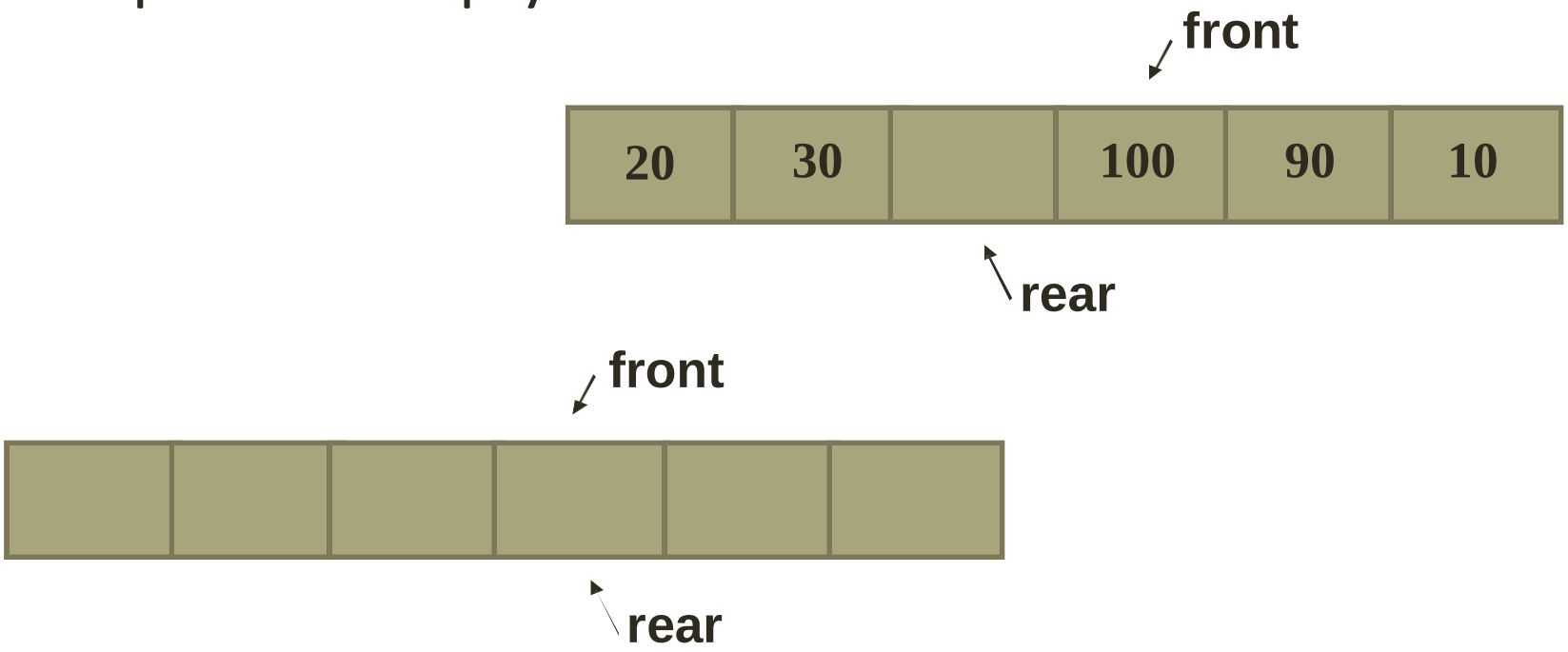
- **Circular queue**
  - **When rear==front, the queue can be either empty or full**



  - Thus, we cannot distinguish between empty and full.

Array-based Queue [7]

- Circular queue
  - We may avoid this situation by **maintaining one empty position**, so that front will never equal to rear unless the queue is empty



Array-based Queue [8]

- **isEmpty()**
  - front equals to rear

```
if (front == rear)
        return true;
else
        return false;
```

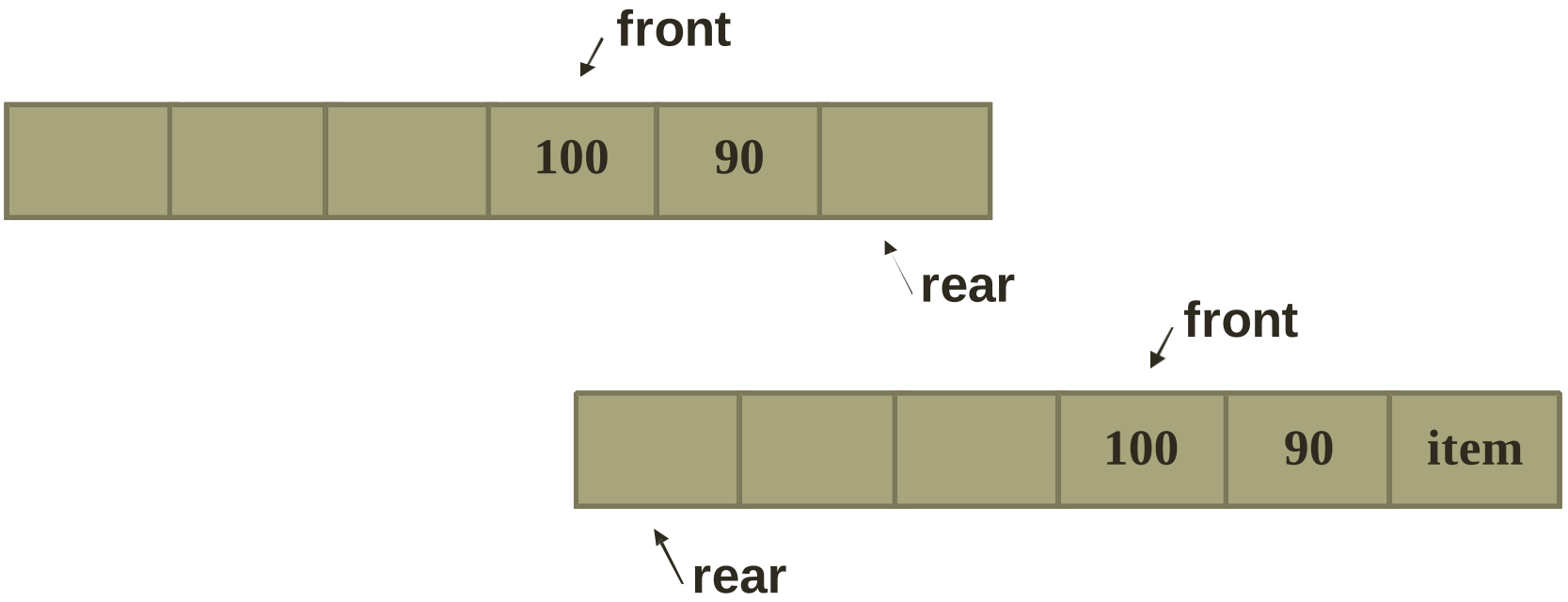
- **isFull()**
  - Front equals *(rear+1) % maxSize*

```
if (front == (rear+1)% maxSize)
        return true;
else
        return false;
```

Array-based Queue [9]

- **enqueue(item)**
  - add item at rear position
  - update new rear

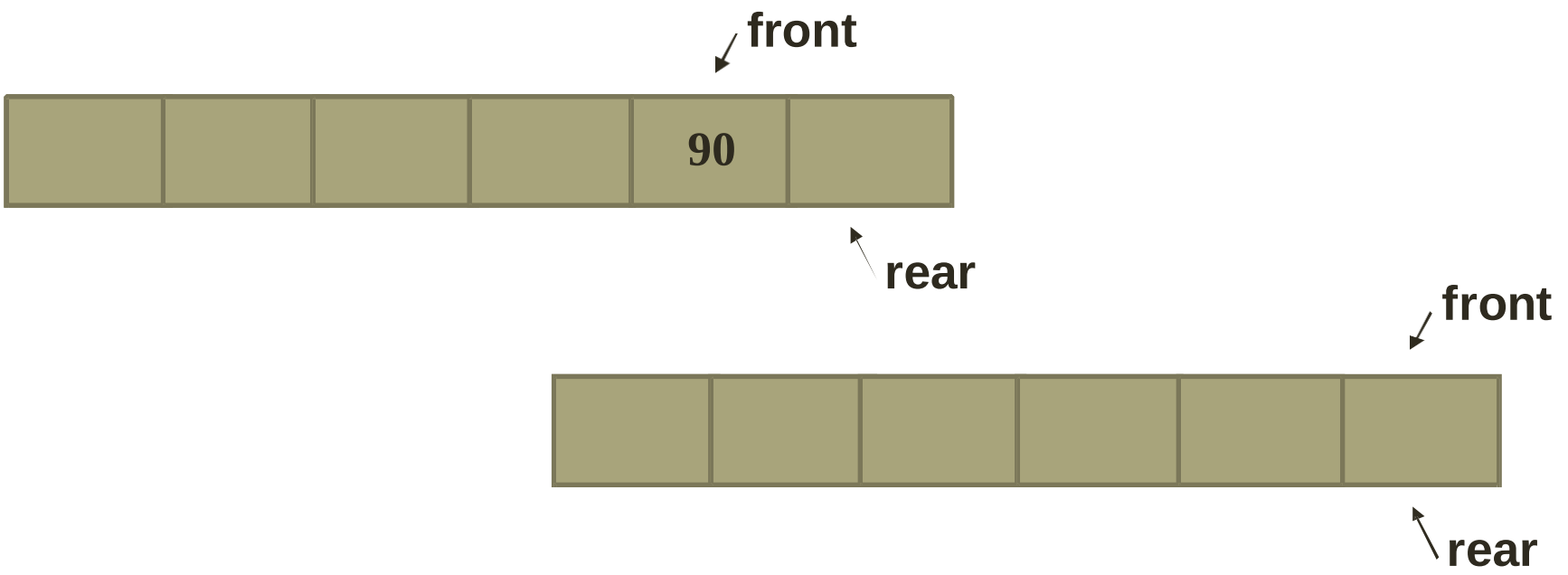
```
if (!isFull())
{
   q[rear]=item;
   rear=(rear+1) % maxSize;
}
```



Array-based Queue [10]

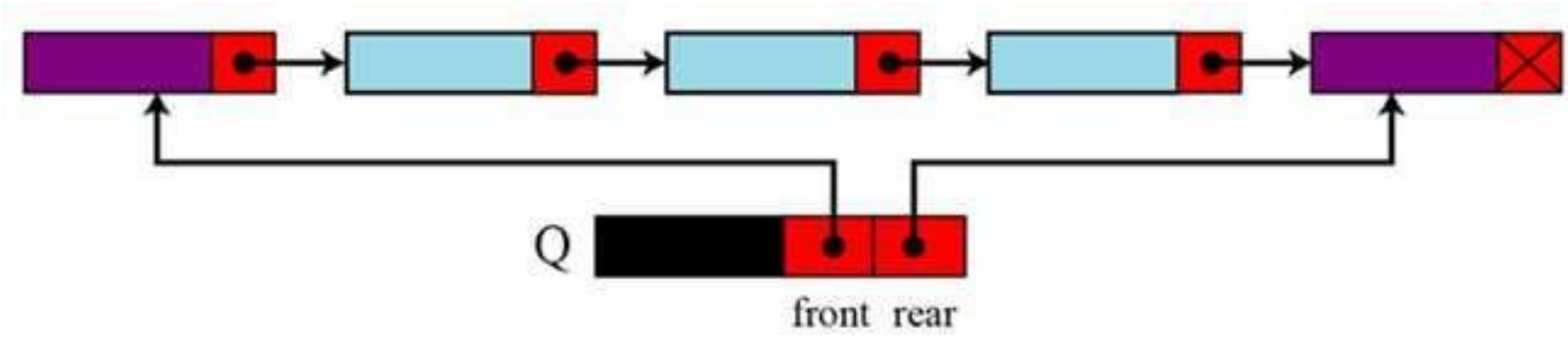
- **dequeue()**
  - return item at front position
  - update new front

```
if (!isEmpty())
{
  pos=front;
  front=(front+1) % maxSize;
  return q[pos];
}
```



List-based Queue [1]

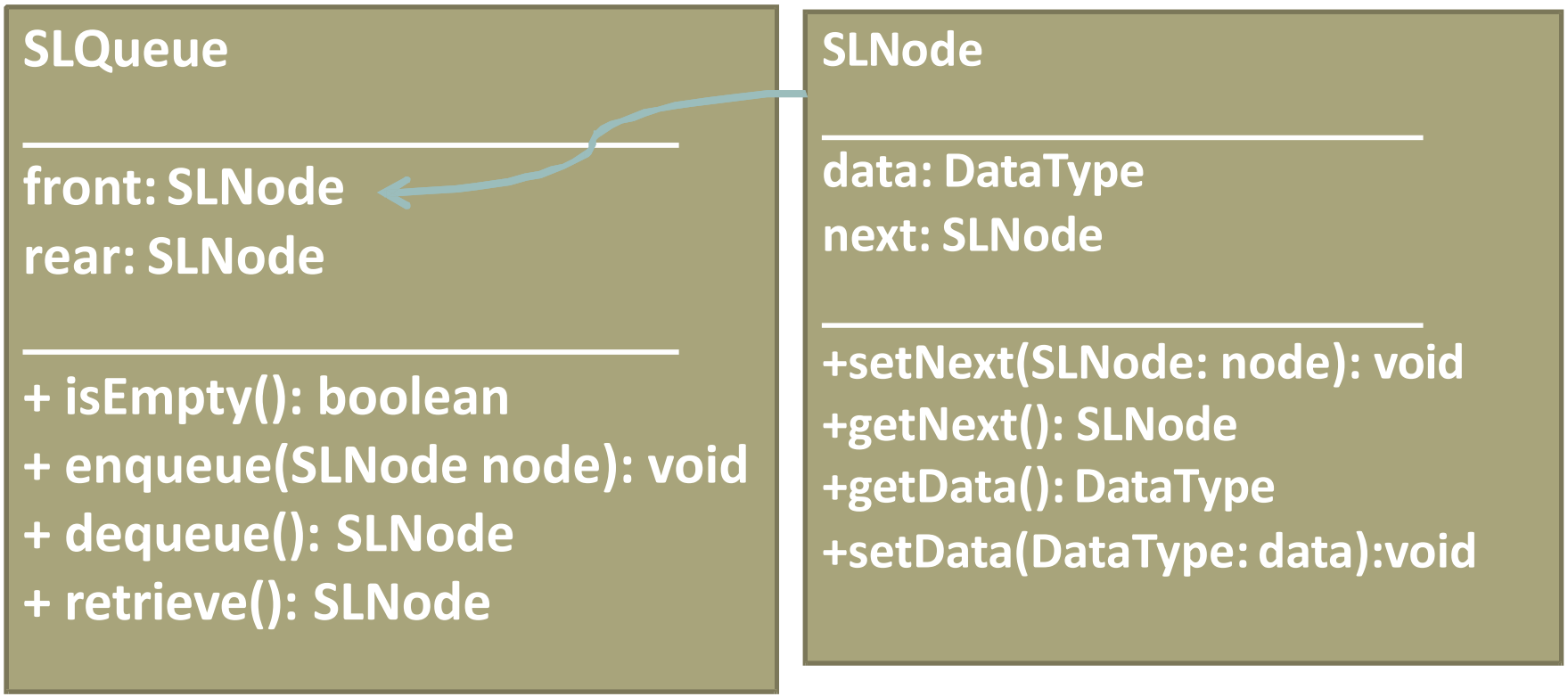
- A linked list is used to implement the queue:



  - front is the head of the list
  - rear is the tail of the list
  - Doesn’t have isFull() operation

List-based Queue [2]

- **ADT**



List-based Queue [3]

- **Create an empty queue**
  - Set front = rear = null
- **isEmpty()**
  - If the queue is empty, then **front==rear==null**
- **enqueue(newNode)**
  - Considering two case:
  - If queue is empty
    - front=rear=newNode
  - If queue is not empty
    - Add newNode to the last position of the list
    - Update rear reference

List-based Queue [4]

- **enqueue(newNode)**

```
newNode.setNext(null);
if (isEmpty())
{
  front=rear=newNode;
}
else
{
  rear.setNext(newNode);
  rear=newNode;
}
```

List-based Queue [5]

- dequeue()
  - Considering two cases:
  - If the queue has one item
    - front=rear=null
  - If the queue has more than one item
    - Get item at front and return
    - Update front reference

```
if (!isEmpty())
{
   SLNode tmp;
   if (front==rear)
   {
        tmp=front;
        front=rear=null;
   }
   else
   {
        tmp=front;
        front=front.getNext();
   }
   return tmp;
}
else
   return null;
```

Tutorial & next topic

- **Preparing for the tutorial:**
  - Practice with examples and exercises in Tutorial 7
- **Preparing for next topic:**
  - Read textbook chapter 3 (3.6 & 3.7) Stack and Queue.