ADTs Stack and Queue

After studying this chapter, you should be able to

- Describe a stack and its operations at a logical level
- Demonstrate the effect of stack operations using a particular implementation of a stack
- Implement the Stack ADT in two array-based implementations and a linked implementation
- Describe the structure of a queue and its operations at a logical level
- Demonstrate the effect of queue operations using a particular implementation of a queue
- Implement the Queue ADT using both an array-based implementation and a linked implementation
- Use inheritance to create a Counted Queue ADT

Stacks: Logical Level

- **Stack:** An ADT where items are added and removed only from the top of the structure; this behavior is called **LIFO** (**L**ast **I**n, **F**irst **O**ut)
- Items are “ordered” by when they were added to the stack
- Like lists, stacks store homogeneous items

Stack Operations

- **Push:** Adds an item to the top of the stack
- **Pop:** Removes the top item from the stack
- **Top:** Returns the item at the top of the stack but does not remove it
- **IsEmpty:** Returns true if the stack has no items
- **IsFull:** Stacks are logically unbounded, but implementations may be bounded

Stacks: Application Level

Stacks have many applications in software engineering. Some examples:

- Tracking the function calls in a program
- Performing syntax analysis on a program
- Traversing structures like trees and graphs
- Some programming languages are entirely stack-based, such as Forth

Stacks are useful in situations where we must process nested components.

Stacks: Implementation Level

- Since elements are homogeneous (the same type), an array-based approach can be used
- We can put elements into sequential slots in the array, placing the first element pushed into the first array position, th second element pushed into the second array position, and so on.

Definitions of Stack Operations

- In the Stack ADT, top indicates which element is on top. Thus the class constructor sets (top == -1) rather than 0. IsEmpty should compare top with -1, and IsFull should compare top with (MAX_ITEMS – 1)'

```
StackType::StackType()
{
    top = -1;
}

bool StackType::IsEmpty() const
{
    return (top == -1);
}

bool StackType::IsFull() const
{
    return (top ==  MAX_ITEMS-1);
}
```

Definitions of Stack Operations (cont.)

- We write the algorithm to Push an item on the top of the stack, Pop an item from the top of the stack, and return a copy of the top item. Push must increment top and store the new item into `items[top].` If the stack is already full when we invoke Push, the resulting condition s called stack overflow. The overflow causes and exception to be thrown; thus we can use a try/catch statement to handle the overflow.

Push

```
if stack if full
      throw an exception FullStack
else
      increment top
      Set items[top] to newItem
```

```
void StackType::Push(ItemType newItem)
{
  if( IsFull() )
    throw FullStack();
  top++;
  items[top] = newItem;
}
```

Definitions of Stack Operations (cont.)

```
void StackType::Pop()
{
   if( IsEmpty() )
     throw EmptyStack();
   top--;
}

ItemType StackType::Top()
{
   if (IsEmpty())
     throw EmptyStack();
   return items[top];
}
```

- Pop is essentially the reverse of Push. We decrement top. If the stack is empty when we invoke Pop or Top, a stack underflow results. As with the Push function, the specifications for the operations say to throw and exception in this event.

```cpp
class StackType
{
public:

   StackType();
   // Class constructor.
   bool IsFull() const;
   // Function: Determines whether the stack is full.
   // Pre:  Stack has been initialized.
   // Post: Function value = (stack is full)
   bool IsEmpty() const;
   // Function: Determines whether the stack is empty.
   // Pre:  Stack has been initialized.
   // Post: Function value = (stack is empty)
   void Push(ItemType item);
   // Function: Adds newItem to the top of the stack.
   // Pre:  Stack has been initialized.
   // Post: If (stack is full), FullStack exception is thrown;
   //     otherwise, newItem is at the top of the stack.
   void Pop();
   // Function: Removes top item from the stack.
   // Pre:  Stack has been initialized.
   // Post: If (stack is empty), EmptyStack exception is thrown;
   //     otherwise, top element has been removed from stack.
   ItemType Top();
   // Function: Returns a copy of top item on the stack.
   // Pre:  Stack has been initialized.
   // Post: If (stack is empty), EmptyStack exception is thrown;
   //     otherwise, top element has been removed from stack.


private:
   int top;
   ItemType  items[MAX_ITEMS];
};
```

Implementing a Stack as a Linked Structure

- Instead of using array, we can implement a stack using linked list structure.

Linked-List Implementation: Push

```
void StackType::Push(ItemType newItem)
// Adds newItem to the top of the stack.
// Pre:  Stack has been initialized.
// Post: If stack is full, FullStack exception is thrown;
//       else newItem is at the top of the stack.

{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```

- Create a new node for the new element
- Have it point to topPtr as its next element
- Update topPtr to point to the new node
- Additionally, must throw an exception if the stack is full when Push is called.

Linked-List Implementation: Pop

- Make a tempPtr that copies topPtr
- Update topPtr to point to the next node on the stack
- Delete the node pointed to by tempPtr
- If the stack is empty when Pop is called, an exception should be thrown.

```
void StackType::Pop()
// Removes top item from Stack and returns it in item.
// Pre:  Stack has been initialized.
// Post: If stack is empty, EmptyStack exception is thrown;
//       else top element has been removed.
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

Other Stack Functions

- **Top:** Returns `topPtr->info`
- **IsEmpty:** Returns `topPtr == NULL`
- **IsFull:** Attempts to allocate a new node, using a try-catch block to handle the `bad_alloc` exception; if an exception is thrown, returns true
- **Constructor:** Initializes `topPtr` to `NULL`
- **Destructor:** Walks the stack and deallocates every node

Big-O Comparison of Stack Implementations

- So which is better? It depends on the situation. The linked implementation certainly gives more flexibility, and in applications where the number of stack items can vary greatly, it wastes less space when the stack is small.
- When the stack size is unpredictable, he linked implementation is preferable, because size is largely irrelevant.
- The array implementation executes faster because it does not incur the run-time overhead of the new and delete operations. If the size is small and it can be sure that do not need to exceed the declared stack size, the array-base is a good choice.

	Static Array	Linked
Class constructor	O(1)	O(1)
IsFull	O(1)	O(1)
IsEmpty	O(1)	O(1)
Push	O(1)	O(1)
Pop	O(1)	O(1)
destructor	NA	O(N)

Queues

Queues: Logical Level

- **Queue:** An ADT in which elements are added to the rear and removed from the front; this behavior is called **FIFO** (**F**irst **I**n, **F**irst **O**ut)
- Items are homogeneous, like in stacks and lists
- Example: A line of people at a cash register

Queue Operations

- **Enqueue:** Add an item to the end of the queue
- **Dequeue:** Removes the item at the front of the queue and returns it
- **IsEmpty:** Returns true if the queue is empty
- **IsFull:** Returns true if the queue is full
- **MakeEmpty:** Removes all items from the queue

Queues: Application Level

Like stacks, queues are used in various ways by the OS and other systems:

- Scheduling jobs on the processor
- Buffering data between processes or other systems

Queues: Implementation Level

- Several implementations are possible
- As before, we’ll start with an array-based implementation

Fixed-Front Queue

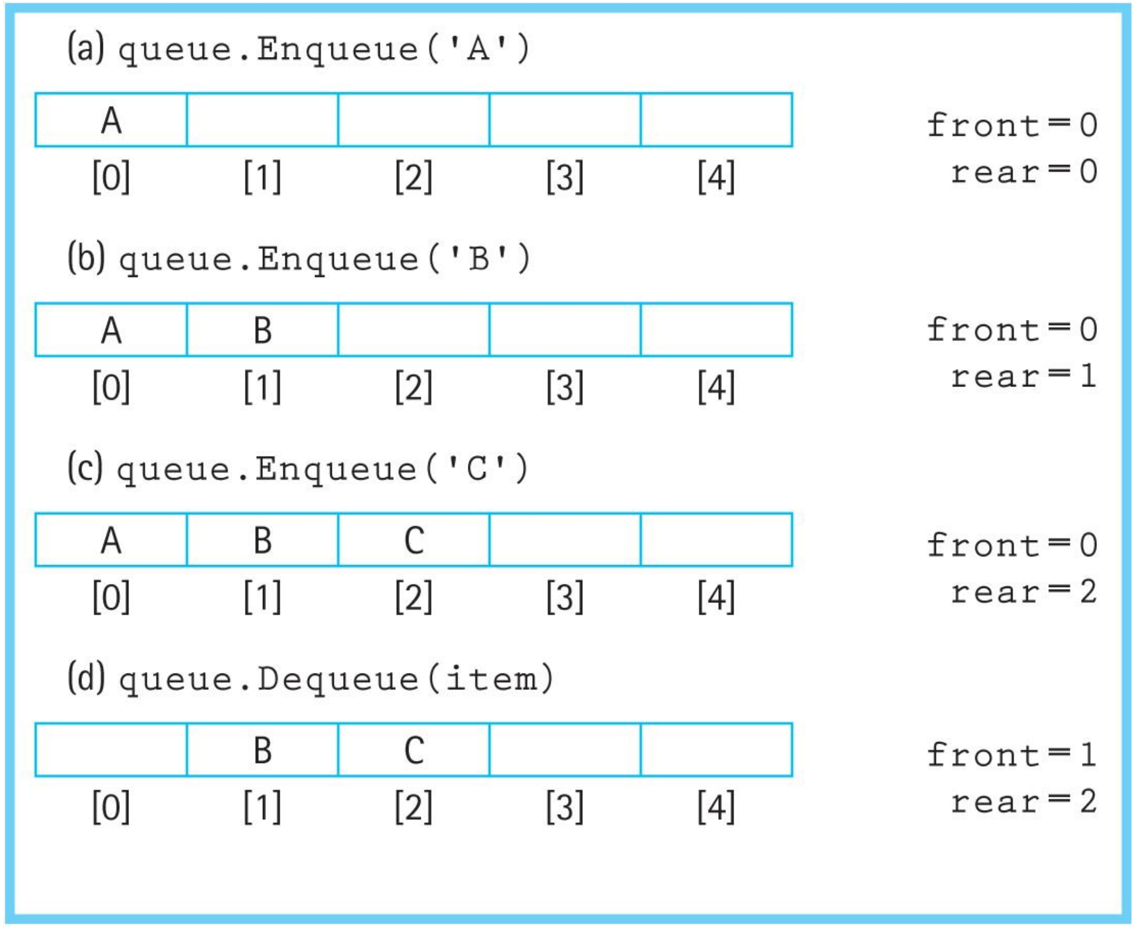
- Array-based implementation where index 0 is always the front of the queue
- Enqueue fills in the first empty slot, second and so on …
- Dequeue empties the first slot and moves all subsequent elements up
- Copying elements like this is inefficient

Floating Queue

- We keep track of the index of the front as well as the rear, we can let both ends of the queue float in the array.
- The Enqueue operation has the same effect as in fix queue operation; they add elements to subsequent slots in the array and increment the index of the rear indicator.
- The Dequeue operation is simpler, however. Instead of moving elements up to the beginning of the array, it merely increments the front indicator to the next slot.

Floating Queue

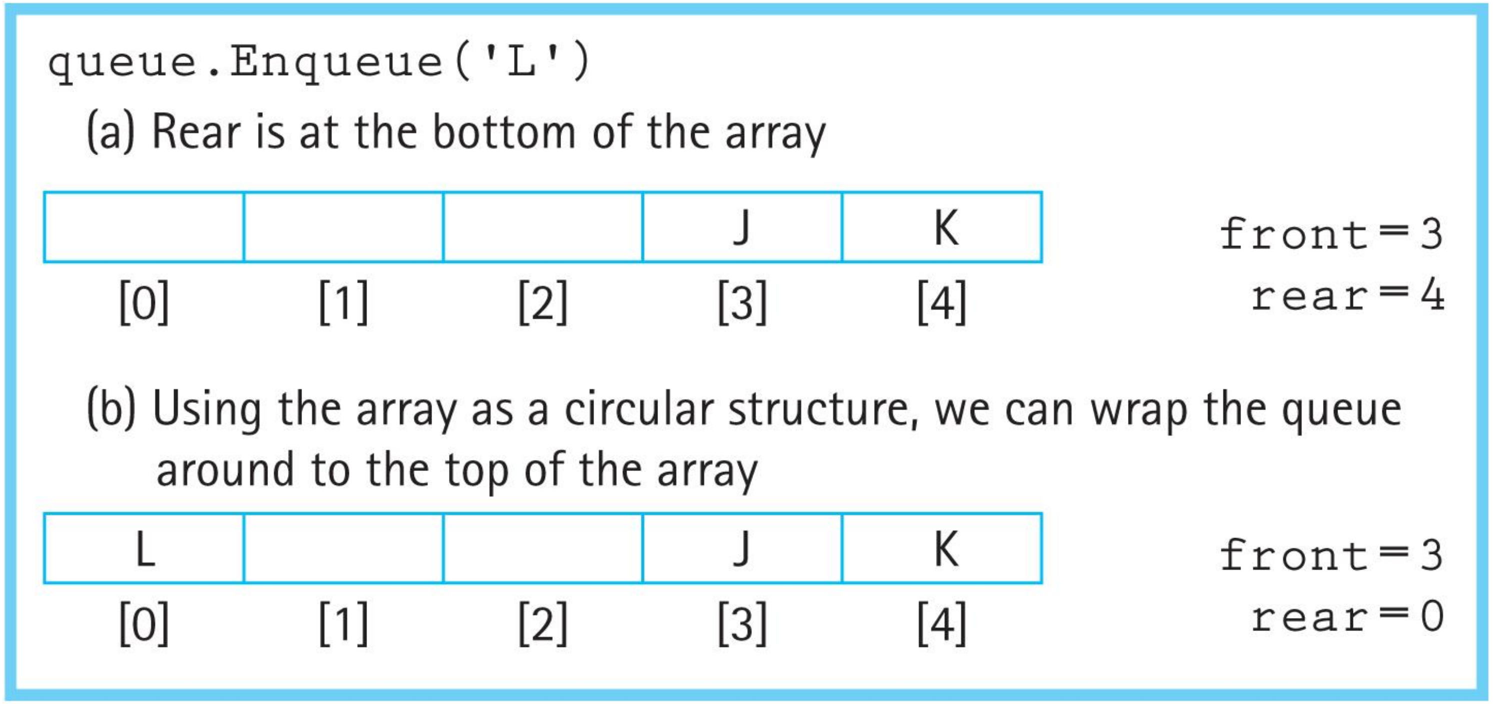
Both the front and end of the queue float in the array



Floating Queue

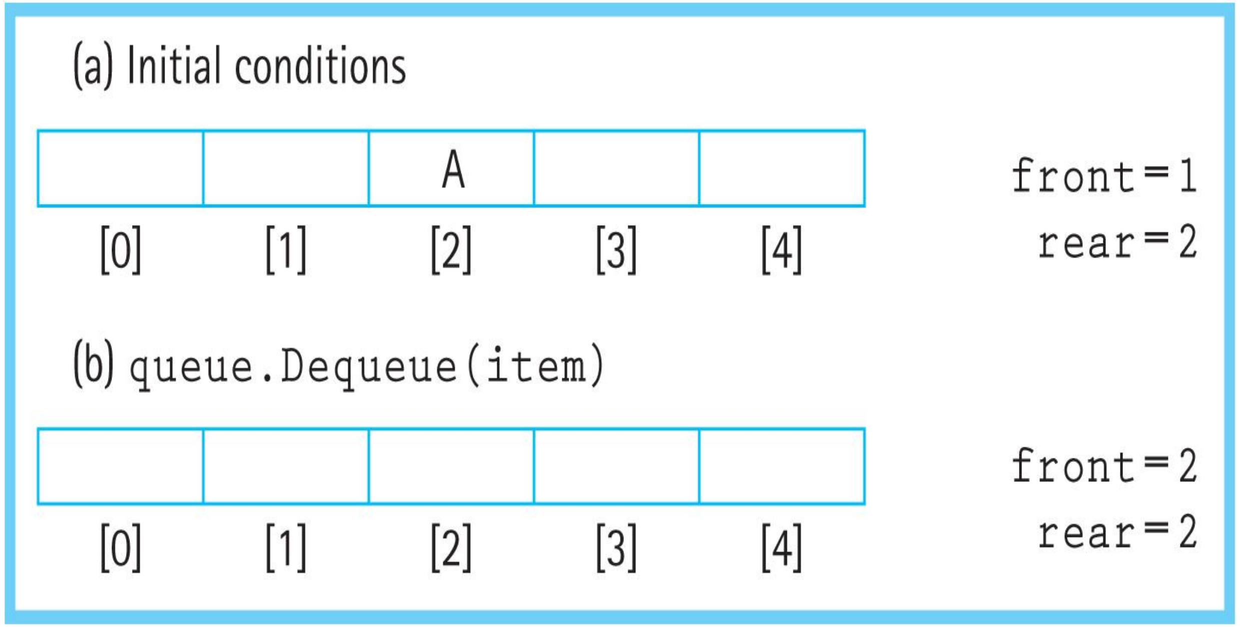
What happens when we reach the end of the array but the queue isn't full?

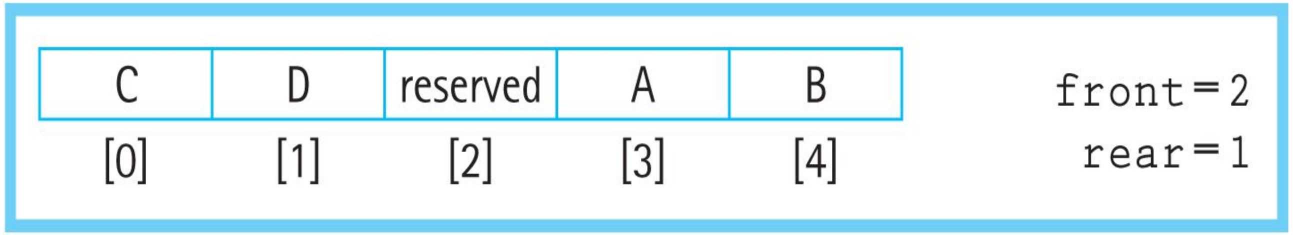
We treat the array as a circular structure by using **`rear = (rear + 1) % maxQue`**:



Wrapping the queue elements around

Floating Queue: IsFull and IsEmpty





How do we know if the queue is empty or full?

- One approach: Keep track of the number of elements in the queue
- Another approach: Make front point to the space *before* the actual first element
  - If front == rear, the queue is empty
  - The space indicated by front is reserved

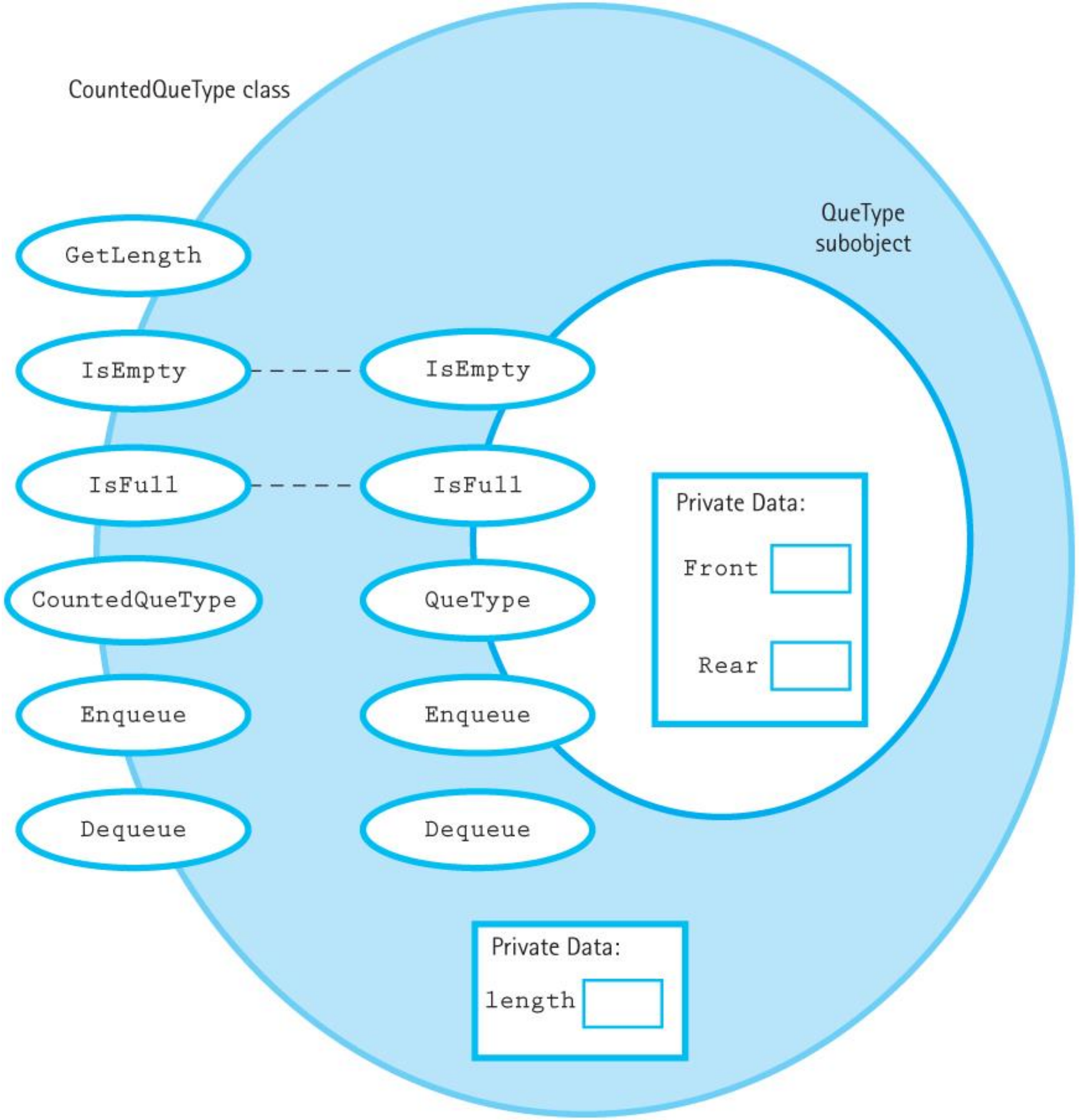
Comparing Array Implementations

- The fixed-front queue has a less efficient Dequeue implementation
- The floating circular queue is more complex but has a more efficient Dequeue
- All other operations are O(1)

Counted Queue

- We may want a count of items in the queue
- Instead of making an entirely new class, we'll instead derive CountedQueType from QueType
- Logically, it needs a field for the count, a method to return the count, a new constructor that initializes the count to 0, and slightly modified Enqueue and Dequeue that change the count
- Deriving the class lets us avoid most of the work

Counted Queue Class Diagram



Class interface diagram for CountedQueType class

Queue Inheritance

- CountedQueType is the **derived class** or subclass
- QueType is the **base class** or superclass
- CountedQueType can't access QueType's private members, but can call the public methods

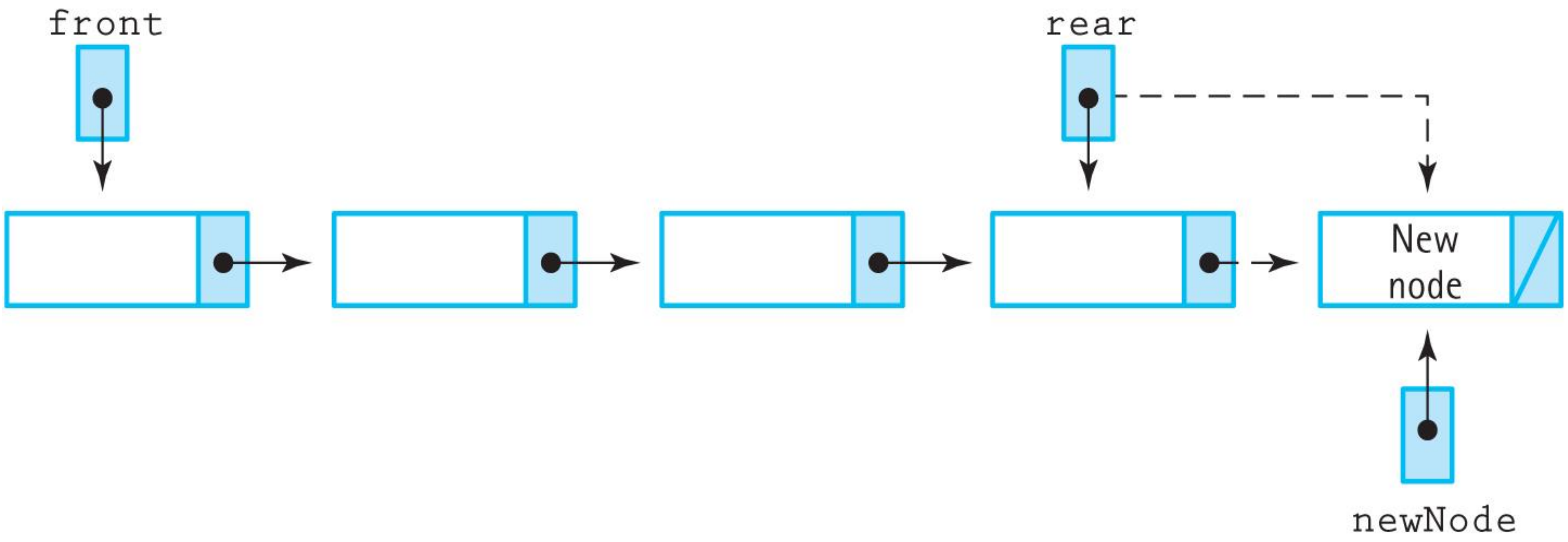
- CountedQueType's Enqueue and Dequeue call QueType's methods and then modify length
- CountedQueType uses QueType's IsFull and IsEmpty method instead of writing new ones

Linked-List Based Queues

- Conceptually similar to the circular queue
- Keep two pointers, tracking the front and rear of the queue

Linked-List Queue: Enqueue Operation

- Algorithm:
  - Allocated new node
  - Update rear->next to point to new node
  - Update rear to point to new node
  - If queue was empty, also update front



Alternate Enqueue – a bad design

- Could try to implement Enqueue the same as the stack's Push by reversing the rear and front pointers
- But this makes Dequeue impossible to implement because we can't go back up the queue

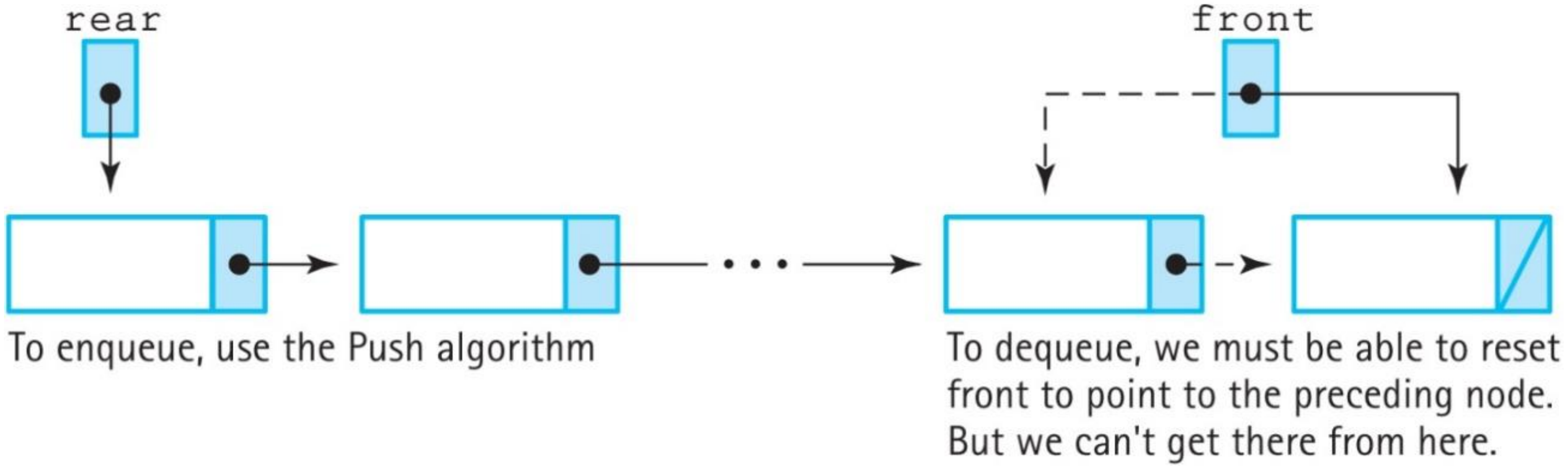
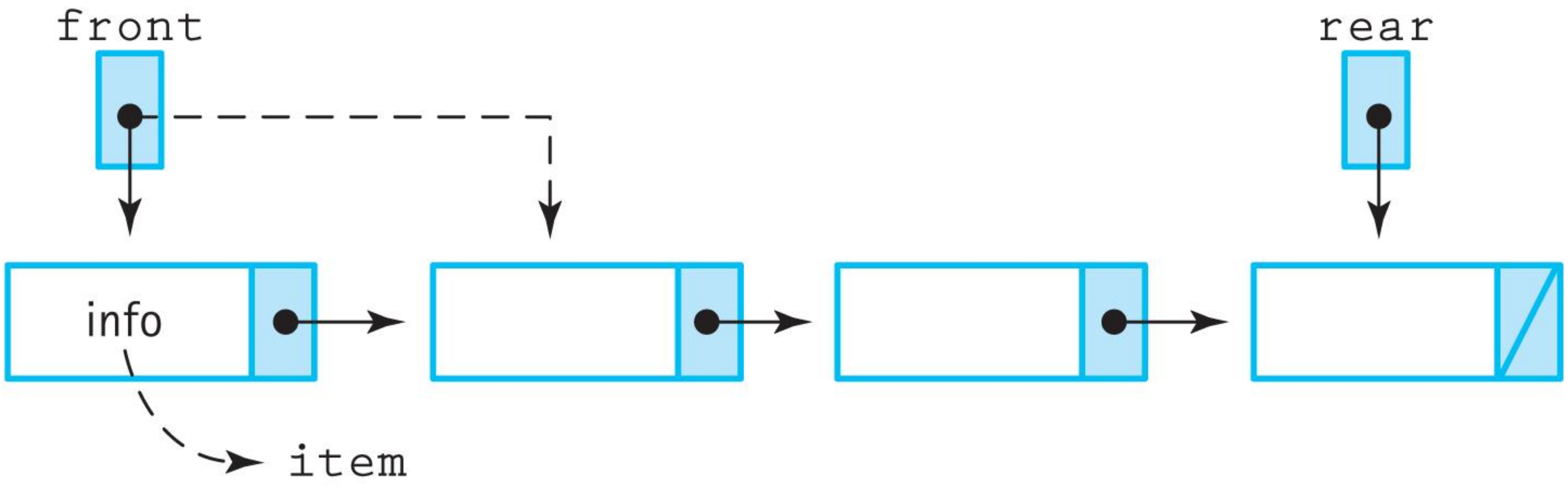


Figure 5.18 A bad queue design

Linked-List Queue: Dequeue Operation

- Similar to the linked list stack's Pop:
  - Use a temp pointer to track the front item
  - Advance the front pointer to front->next
  - If front is now NULL, set rear to NULL (queue is empty)



Other Queue Operations

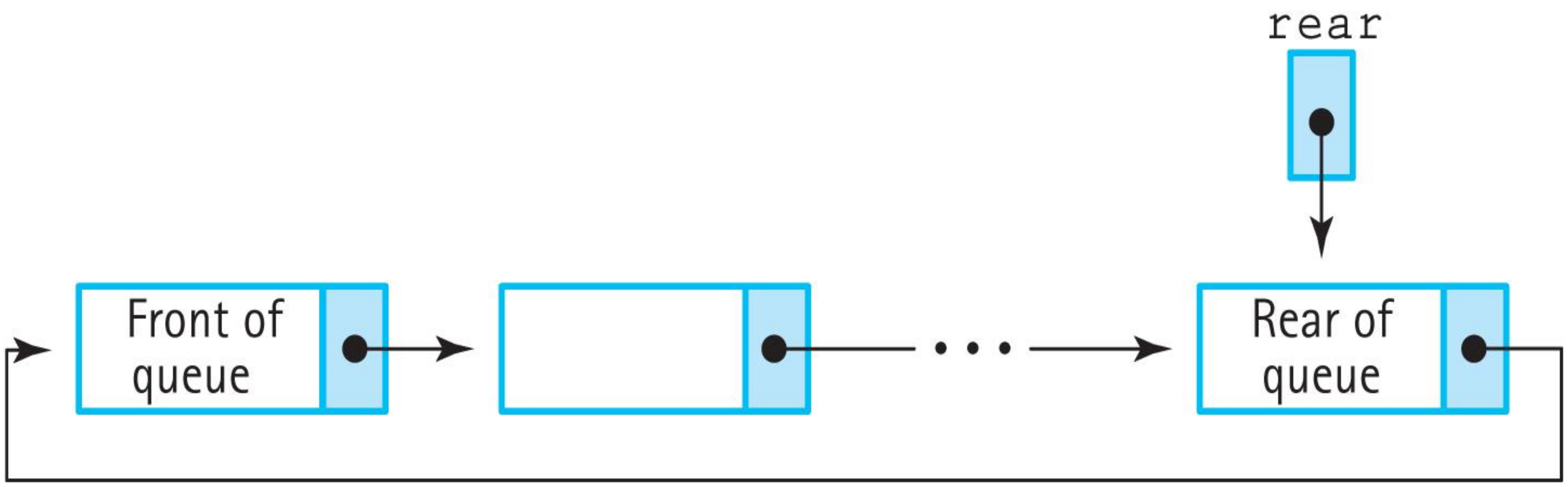
- **IsFull:** Attempts to allocate a new node
- **IsEmpty:** Checks if front is NULL
- **MakeEmpty:** Walks the linked structure and deallocates the nodes
- **Destructor:** Calls MakeEmpty

Circular Linked Queue

- Is it possible to have only one pointer in the QueType class?
  - Only front: Can access the rear by walking the links, which is O(*N*)
  - Only rear: Can’t access the front because the links only point towards the rear
- Yes, by implementing a circular linked list

Circular Linked Queue

- The class only has a pointer to the rear of the queue
- Instead of pointing to NULL, the last node's next points to the front of the queue



Comparing Queue Implementations

- All operations are O(1), except the linked list-based MakeEmpty and destructor are O(*N*)
- The linked list-based queue has memory overhead

	Dynamic Array Implementation	Linked Implementation
Class constructor	O(1)	O(1)
`MakeEmpty`	**O(1)**	**O(*N*)**
`IsFull`	O(1)	O(1)
`IsEmpty`	O(1)	O(1)
`Enqueue`	O(1)	O(1)
`Dequeue`	O(1)	O(1)
Destructor	**O(1)**	**O(*N*)**

Comparing Queue Implementations

- The array-based queue requires a fixed amount of memory no matter how many items are actually in the queue.
- The linked list-based queue consumes more memory as more elements are added.
- For cases with few items or large items, linked list-based queues can be more efficient.

The End!