Chapter 5

ADTs Stack and Queue

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: Implementation Level

- Since elements are homogeneous, an array-based approach can be used
- Elements are inserted into subsequent indices in the array
- Very similar to the Unsorted List ADT!
- We can use the same implementation strategy, but keep in mind: A stack is not the same as an unsorted list

Stack Operation Implementation

- The `top` field is the index of the top of the stack
- **Push:** Increment top and insert the element at the index given by `top`
  - If the array is full, attempting to add an item triggers **stack overflow** and an exception is thrown
- **IsEmpty:** `top` == -1
- **IsFull:** `top` == MAX_ITEMS – 1
- Push increments `top` before adding an item, so `top` is set back one from the start

Stack Operation Implementation (cont.)

- **Pop:** The opposite of Push; simply decrement top, treating the removed item as garbage
- **Top:** Returns items[top]; if the stack is empty, triggers stack underflow
- What if the stack is empty? Calling Pop or Top on an empty stack triggers **stack underflow** and an exception is thrown

Stacks: Push and Pop

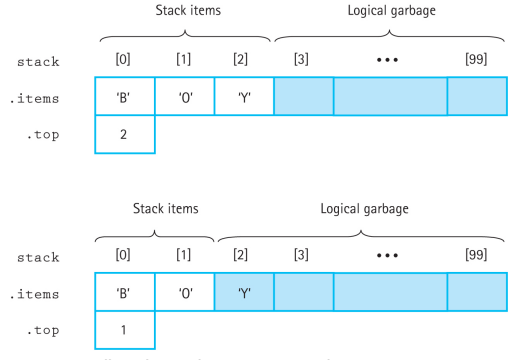

Figure 5.4 The effect of a Pop following a series of Pushes

Dynamic Array Implementation

- As with lists, we can implement the stack with a dynamically allocated array
- Add parameterized and default constructors
- Add a destructor to clean up the array
- Replace references to MAX_ITEMS with maxStack, which is now a field in the class

Linked List-Based Stack

- Linked Lists allow our stack to be physically unbounded (or only bounded by memory)
- The main change to the class definition is adding a pointer to the top element
- This gives us direct access to the top of the stack

Linked-List Implementation: Push

- Essentially the same as Unsorted List
  - 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

- The algorithm is simple:
  - 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

Linked-List Implementation: Pop (cont.)

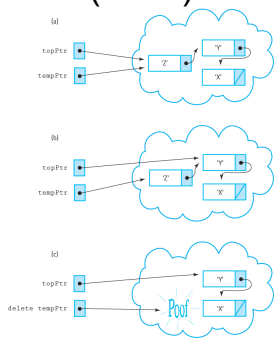

Figure 5.5 Popping the stack

Popping the Last Stack Element

- The algorithm correctly handles removing the last element of the stack

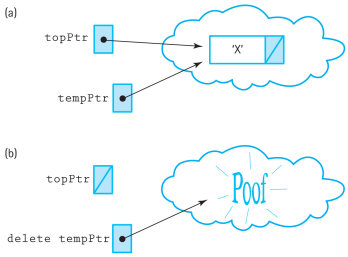

Figure 5.6 Popping the last element on the stack

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

Comparing Stack Implementations

Table 5.1 Big-O Comparison of Stack Operations

	Static Array Implementation	Dynamic Array Implementation	Linked Implementation
class constructor	O(1)	O(1)	O(1)
IsFull	O(1)	O(1)	O(1)
IsEmpty	O(1)	O(1)	O(1)
Push	O(1)	O(1)	O(1)
Pop	O(1)	O(1)	O(1)
destructor	NA	O(1)	O(*N*)

Table 5.1 Big-O Comparison of Stack Operations

Comparing Stack Implementations (cont.)

- Array-based version is small, simple, and efficient
  - May allocate too much memory (stack size small)
  - May not have enough memory (stack size large)
- Linked list-based version is very flexible and more memory efficient
  - Only allocates memory as needed
  - Some operations have higher overhead

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
- Dequeue empties the first slot and moves all subsequent elements up
- Copying elements like this is inefficient

Floating Queue

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

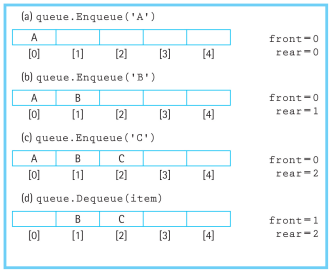

Figure 5.9 The effect of Enqueue and Dequeue

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`**:

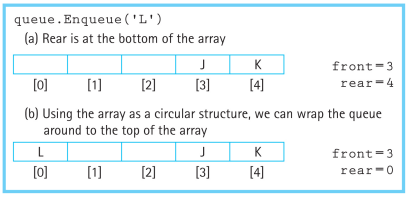

Figure 5.10 Wrapping the queue elements around

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

Floating Queue: IsFull and IsEmpty (cont.)

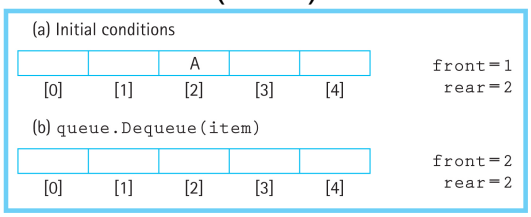

Figure 5.13 Testing for an empty queue

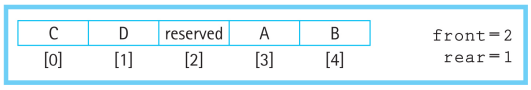

Figure 5.14 Testing for a full queue

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

- Users 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

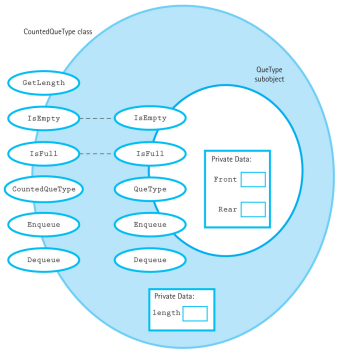

Figure 5.15 Class interface diagram for CountedQueType class

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

Queue Inheritance

- 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

- Algorithm:
  - Allocated new node
  - Update rear->next to point to new node
  - Update rear to point to new node

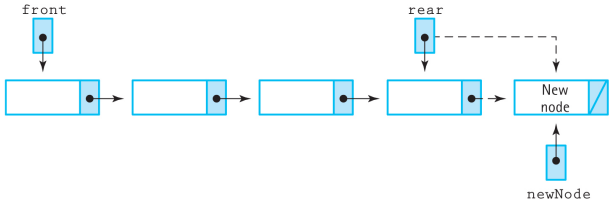

Figure 5.17 The Enqueue operation

Alternate Enqueue

- 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

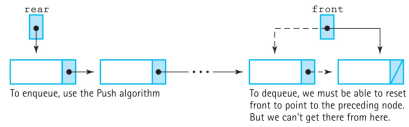

Figure 5.18 A bad queue design

Linked-List Queue: Dequeue

- 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)

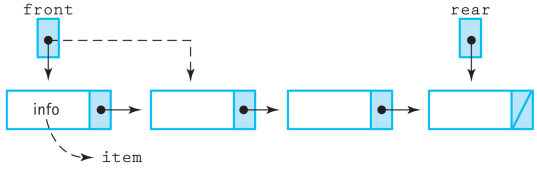

Figure 5.19 The Dequeue operation

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

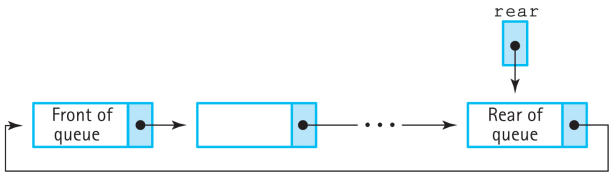

Figure 5.20 A circular linked queue

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

Table 5.2 Big-O Comparison of Queue Operations

	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*)

Table 5.2 Big-O Comparison of Queue Operations

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

Array vs. Linked-List Memory

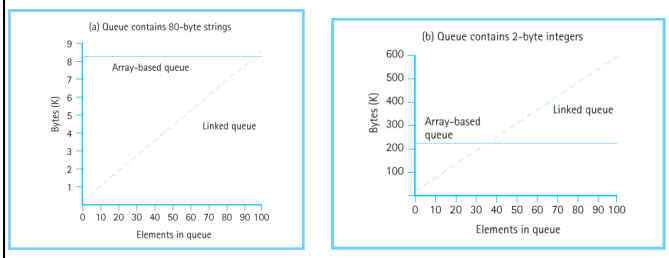

Figure 5.21 Comparison of storage requirements: (a) Queue contains 80-byte strings (b) Queue contains 2-byte integers