CHAPTER 7

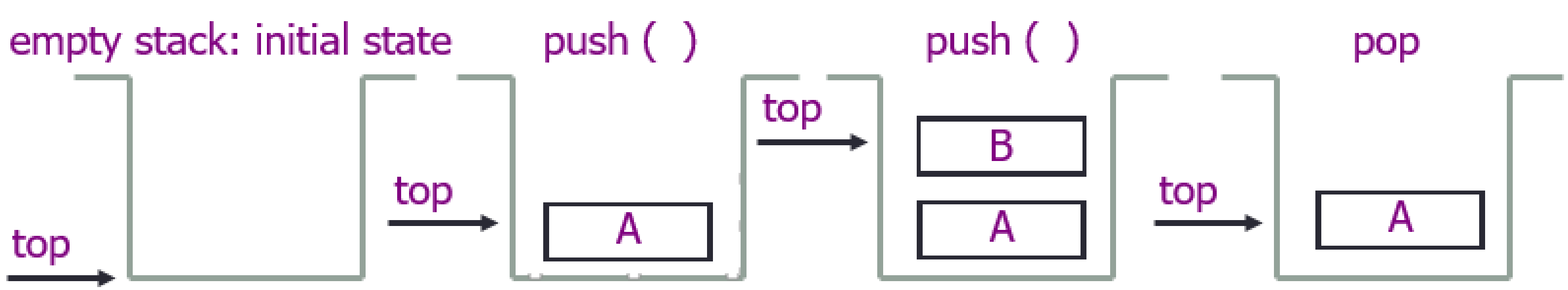
STACKS AND QUEUES

What is a Stack?

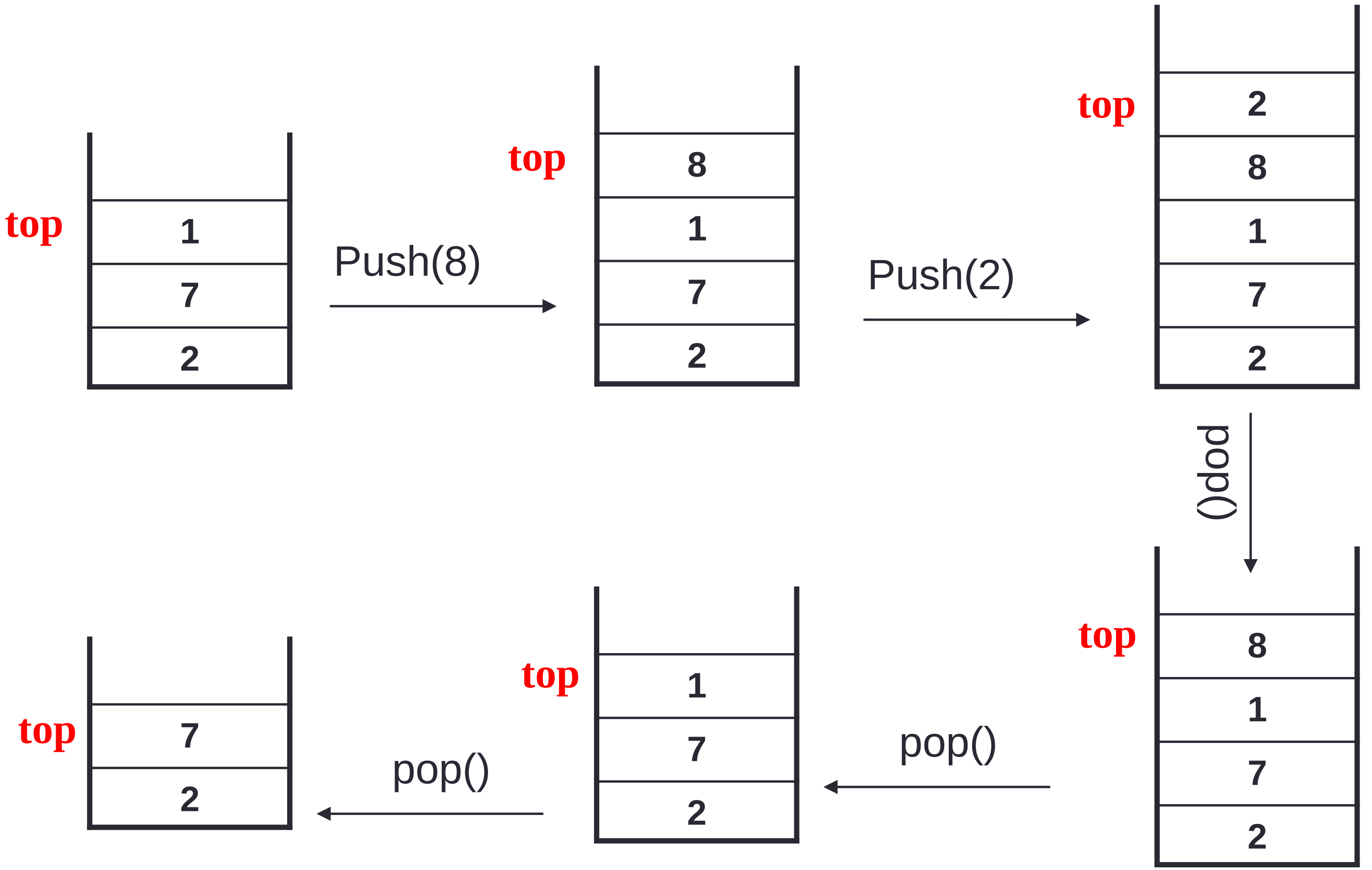
- Stack is a data structure in which data is added and removed at only one end called the top
- Examples of stacks are:
  - Stack of books
  - Stack of trays in a cafeteria

Stack

- A Last In First Out (LIFO) data structure
- Primary operations: Push and Pop
- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element from the top of the stack
- An example



Building Stack Step-by-Step



Stack Errors

- Stack Overflow
  - An attempt to add a new element in an already full stack is an error
  - A common mistake often made in stack implementation
- Stack Underflow
  - An attempt to remove an element from the empty stack is also an error
  - Again, a common mistake often made in stack implementation

Applications of Stacks

- Some direct applications:
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Evaluating postfix expressions (e.g., xy+)

- Some indirect applications
  - Auxiliary data structure for some algorithms (e.g., Depth First Search algorithm)
  - Component of other data structures

The Stack Abstract Data Type

- Stacks are the simplest of all data structures
- Formally, a stack is an abstract data type (ADT) that supports the following two methods:
  - push(*e*): Insert element *e* to the top of the stack
  - pop(): Remove from the stack and return the top element on the stack;
      - an error occurs if the stack is empty – what error?
- Additionally, let us also define the following methods:
  - size(): Return the number of elements in the stack
  - isEmpty(): Return a Boolean indicating if the stack is empty
  - top(): Return the top element in the stack, without removing it
    - an error occurs if the stack is empty

The Stack Abstract Data Type

- **Example :** The following table shows a series of stack operations and their effects on an initially empty stack S of integers.

Operation	Output	Stack Contents
push(5)	-	(5)
push(3)	-	(5, 3)
pop()	*3*	(5)
push(7)	-	(5, 7)
pop()	*7*	(5)
top()	*5*	(5)
pop()	*5*	()
pop()	*"error“*	()
isEmpty()	*true*	()
push(9)	-	(9)
push(7)	-	(9, 7)
push(3)	-	(9, 7, 3)
push(5)	-	(9, 7, 3, 5)
size()	*4*	(9, 7, 3, 5)
pop()	*5*	(9, 7, 3)
push(8)	-	(9, 7, 3, 8)
pop()	*8*	(9, 7, 3)
pop()	*3*	(9, 7)

A Stack Interface in Java

- The stack data structure is included as a "built-in" class in the java.util package of Java.
- Class java.util.Stack is a data structure that stores generic Java objects and includes, among others, the following methods:
  - push(),
  - pop()
  - peek() (equivalent to top()),
  - size(), and empty() (equivalent to isEmpty()).
  - Methods pop() and peek() throw exception EmptyStackException if they are called on an empty stack.

The Stack Abstract Data Type

- Implementing an abstract data type in Java involves two steps. The first step is the definition of a Java ***Application Programming Interface*** (API), or simply ***interface***, which describes the names of the methods that the ADT supports and how they are to be declared and used.

- In addition, we must define exceptions for any error conditions that can arise. For instance, the error condition that occurs when calling method pop() or top() on an empty stack is signaled by throwing an exception of type **EmptyStackException**,

```
public class EmptyStackException extends RuntimeException {
  public EmptyStackException(String err) {
    super(err);
  }
}
```

The Stack Abstract Data Type

Code Fragment 7.1 : Interface Stack documented with comments in Javadoc style. Note also the use of the generic parameterized type, E, which implies that a stack can contain elements of any specified class.

```
public interface Stack<E> {
  /**
   * Return the number of elements in the stack.
   * @return number of elements in the stack.
   */
  public int size();
  /**
   * Return whether the stack is empty.
   * @return true if the stack is empty, false otherwise.
   */
  public boolean isEmpty();
  /**
   * Inspect the element at the top of the stack.
   * @return top element in the stack.
   * @exception EmptyStackException if the stack is empty.
   */
  public E top()
    throws EmptyStackException;
  /**
   * Insert an element at the top of the stack.
   * @param element to be inserted.
   */
  public void push (E element);
  /**
   * Remove the top element from the stack.
   * @return element removed.
   * @exception EmptyStackException if the stack is empty.
   */
  public E pop()
    throws EmptyStackException;
}
```

A Simple Array-Based Stack Implementation

- We can implement a stack by storing its elements in an array.
- Specifically, the stack in this implementation consists of
  - an *N*-element array *S*
  - plus an integer variable *t* that gives the index of the top element in array *S*.

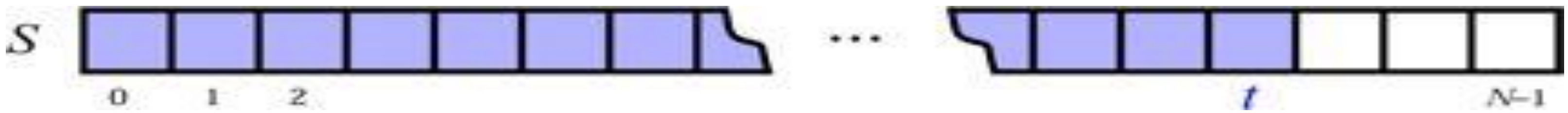


Figure 5.2: Implementing a stack with an array *S*. The top element in the stack is stored in the cell *S*[*t*].

- Recalling that arrays start at index 0 in Java,
  - we initialize *t* to −1, and we use this value for *t* to identify an empty stack.
- Likewise, we can use *t* to determine the number of elements (*t* + 1).
- FullStackException, to signal the error that arises if we try to insert a new element into a full array.
- Exception FullStackException is specific to this implementation and is not defined in the stack ADT.

Code Fragment 7.2: Implementing a stack using an array of a given size, N.

```
Algorithm size():
    return t + 1
Algorithm isEmpty():
    return (t < 0)
Algorithm top():
    if isEmpty() then
        throw a EmptyStackException
    return S[t]
Algorithm push(e):
    if size() = N then
        throw a FullStackException
    t ← t + 1
    S[t] ← e
Algorithm pop():
    if isEmpty() then
        throw a EmptyStackException
    e ← S[t].
    S[t] ← null
    t ← t − 1
    return e
```

A Drawback with the Array-Based Stack Implementation

- The array implementation of a stack is simple and efficient.
- This implementation has one negative aspect
  - it must assume a fixed upper bound, CAPACITY, on the ultimate size of the stack.
- Stacks serve a vital role in a number of computing applications, so it is helpful to have a fast stack ADT implementation such as the simple array-based implementation.

- Thus, even with its simplicity and efficiency, the array-based stack implementation is not necessarily ideal.
- Fortunately, there is another implementation, which we discuss next,
  - that does not have a size limitation
  - and use space proportional to the actual number of elements stored in the stack.

Implementing a Stack with a Generic Linked List

- Using a singly linked list to implement the stack ADT.

- In designing such an implementation, we need to decide if
  - the top of the stack is at the head
  - or at the tail of the list.

- Rather than use a linked list that can only store objects of a certain type, we would like, in this case, to implement a generic stack using a ***generic*** linked list.

- Thus, we need to use a generic kind of node to implement this linked list. We show such a Node class in **Code Fragment 7.3**

Implementing a Stack with a Generic Linked List

- **Code Fragment 7.3**: Class Node, which implements a generic node for a singly linked list.
- ```
  public class Node<E> {
    // Instance variables:
    private E element;
    private Node<E> next;
    /** Creates a node with null references to its element and next node. */
    public Node() {
      this(null, null);
    }
    /** Creates a node with the given element and next node. */
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }
    // Accessor methods:
    public E getElement() {
      return element;
    }
    public Node<E> getNext() {
      return next;
    }
    // Modifier methods:
    public void setElement(E newElem) {
      element = newElem;
    }
    public void setNext(Node<E> newNext) {
      next = newNext;
    }
  }
  ```
  ```

A Generic NodeStack Class

- A Java implementation of a stack, by means of a generic singly linked list, is given in Code Fragment 7.4

- All the methods of the Stack interface are executed in constant time.

- In addition to being time efficient, this linked list implementation has a space requirement that is $O(n)$, where n is the current number of elements in the stack.

Code Fragment 7.4: Class NodeStack, which implements the Stack interface using a singly linked list, whose nodes are objects of class Node from Code Fragment 7.3

```
public class NodeStack<E> implements Stack<E> {
  protected Node<E> top;       // reference to the head node
  protected int size;          // number of elements in the stack
  public NodeStack() { // constructs an empty stack
    top = null;
    size = 0;
  }
  public int size() { return size; }
  public boolean isEmpty() {
    if (top == null) return true;
    return false;
  }
  public void push(E elem) {
    Node<E> v = new Node<E>(elem, top); // create and link-in a new node
    top = v;
    size++;
  }
  public E top() throws EmptyStackException {
    if (isEmpty()) throw new EmptyStackException("Stack is empty.");
    return top.getElement();
  }
  public E pop() throws EmptyStackException {
    if (isEmpty()) throw new EmptyStackException("Stack is empty.");
    E temp = top.getElement();
    top = top.getNext();       // link-out the former top node
    size--;
    return temp;
  }
}
```

Reversing an Array Using a Stack

Code Fragment 7.5: A generic method that reverses the elements in an array of type E objects, using a stack declared using the Stack<E> interface.

- The basic idea is simply to push all the elements of the array in order into a stack and then fill the array back up again by popping the elements off of the stack.

```
/** A nonrecursive generic method for reversing an array */
public static <E> void reverse(E[] a) {
  Stack<E> S = new ArrayStack<E>(a.length);
  for (int i=0; i < a.length; i++)
    S.push(a[i]);
  for (int i=0; i < a.length; i++)
    a[i] = S.pop();
}
```

Code Fragment 7.6: A test of the reverse method using two arrays.

```
/** Tester routine for reversing arrays */
public static void main(String args[]) {
  Integer[] a = {4, 8, 15, 16, 23, 42};  // autoboxing allows this
  String[] s = {"Jack", "Kate", "Hurley", "Jin", "Boone"};
  System.out.println("a = " + Arrays.toString(a));
  System.out.println("s = " + Arrays.toString(s));
  System.out.println("Reversing. . .");
  reverse(a);
  reverse(s);
  System.out.println("a = " + Arrays.toString(a));
  System.out.println("s = " + Arrays.toString(s));
}
```

The output from this method is the following:

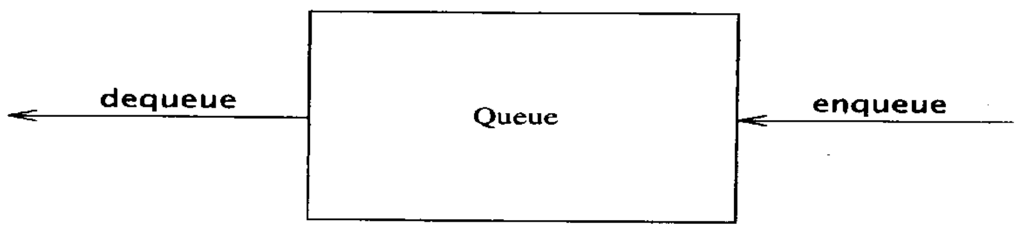
```
a = [4, 8, 15, 16, 23, 42]
s = [Jack, Kate, Hurley, Jin, Michael]
Reversing...
a = [42, 23, 16, 15, 8, 4]
s = [Michael, Jin, Hurley, Kate, Jack]
```

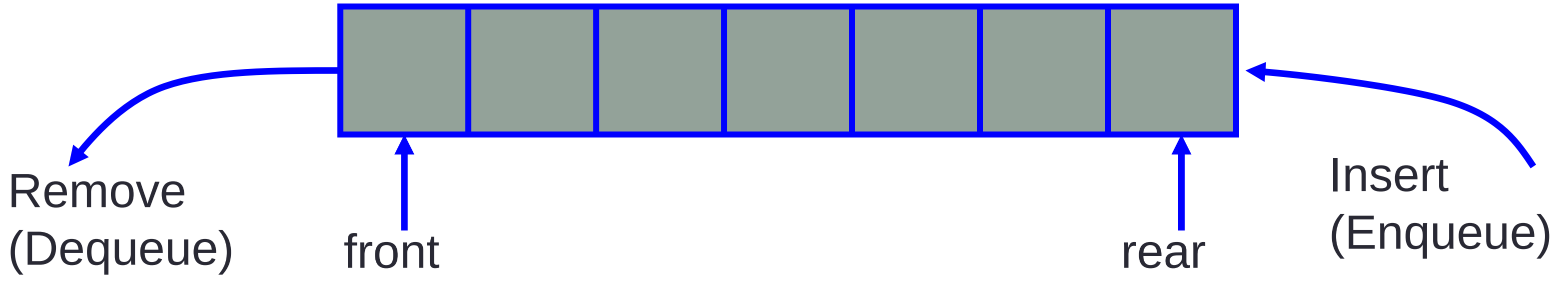
What is Queue?

- Queue is a data structure in which insertion is done at one end (Front), while deletion is performed at the other end (Rear)
  - Contrast with stack, where insertion and deletion at one and the same end
- It is First In, First Out (FIFO) structure
  - For example, customers standing in a check-out line in a store, the first customer in is the first customer served.

Queue operations

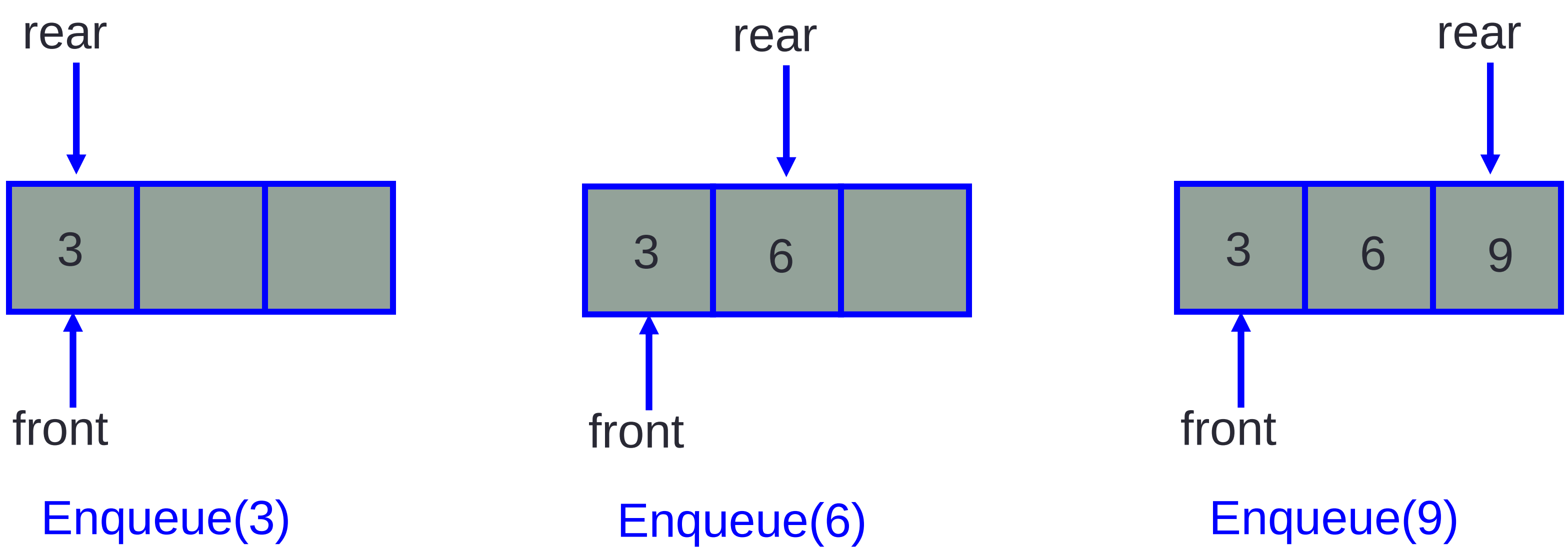
- Enqueue: insert an element at the rear of the list
- Dequeue: delete the element at the front of the list





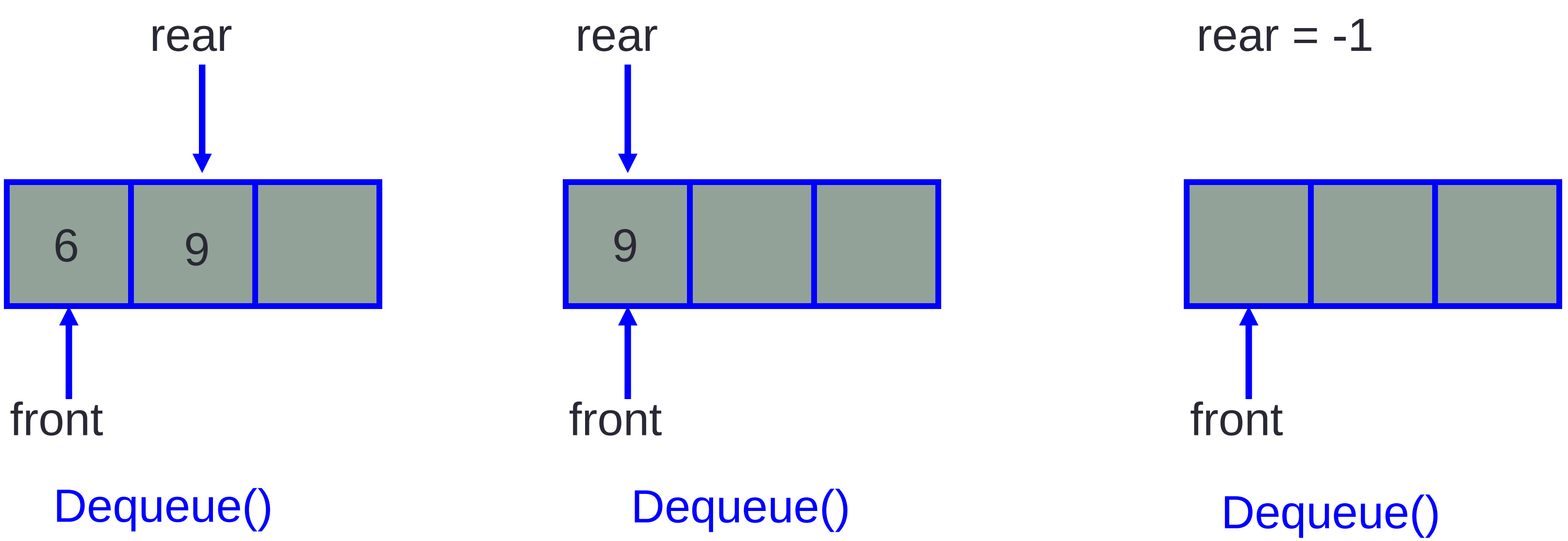
Building a Queue Step-by-Step

- There are several different algorithms to implement Enqueue and Dequeue
- Enqueuing
  - The front index is always fixed
  - The rear index moves forward in the array



Building a Queue Step-by-Step

- Dequeuing
  - The element at the front of the queue is removed
  - Move all the elements after it by one position



The Queue Abstract Data Type

- Formally, the queue abstract data type defines a collection that keeps objects in a sequence, where
  - element access and deletion are restricted to the first element in the sequence, the ***front*** of the queue,
  - and element insertion is restricted to the end of the sequence, the ***rear*** of the queue.
  - This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in first-out (FIFO) principle.
- The ***queue*** abstract data type (ADT) supports the following two fundamental methods:
  - enqueue(*e*): Insert element *e* at the rear of the queue.
  - dequeue(): Remove and return from the queue the object at the front;
    - an error occurs if the queue is empty.
- Additionally, similar to the case with the Stack ADT, the queue ADT includes the following supporting methods:
  - size(): Return the number of objects in the queue.
  - isEmpty(): Return a Boolean value that indicates whether the queue is empty.
- front(): Return, but do not remove, the front object in the queue;
    - an error occurs if the queue is empty.

The Queue Abstract Data Type

- Example : The following table shows a series of queue operations and their effects on an initially empty queue Q of integer objects. For simplicity, we use integers instead of integer objects as arguments of the operations.

Operation	Output	front ← Q ← rear
enqueue(5)	-	(5)
enqueue(3)	-	(5, 3)
dequeue( )	5	(3)
enqueue(7)	-	(3, 7)
dequeue( )	3	(7)
front( )	7	(7)
dequeue( )	7	( )
dequeue( )	"error"	( )
isEmpty( )	true	( )
enqueue(9)	-	(9)
enqueue(7)	-	(9, 7)
size()	2	(9, 7)
enqueue(3)	-	(9, 7, 3)
enqueue(5)	-	(9, 7, 3, 5)
dequeue( )	9	(7, 3, 5)

The Queue Abstract Data Type

- Example Applications
- There are several possible applications for queues.
  - Stores,
  - theaters,
  - reservation centers,
  - and other similar services typically process customer requests according to the FIFO principle.
- A queue would therefore be a logical choice for a data structure to handle transaction processing for such applications.
- For example, it would be a natural choice for handling calls to the reservation center of an airline or to the box office of a theater.

Implementing a Queue with a Generic Linked List

- We can efficiently implement the queue ADT using a generic singly linked list.
- For efficiency reasons,
  - the front of the queue to be at the head of the list,
  - and the rear of the queue to be at the tail of the list.
  - In this way, we remove from the head and insert at the tail. Note that we need to maintain references to both the head and tail nodes of the list. Rather than go into every detail of this implementation, we simply give a Java implementation for the fundamental queue methods in Code Fragment 7.7

Code Fragment 7.7: Methods enqueue and dequeue in the implementation of the queue ADT by means of a singly linked list, using nodes from class Node of Code Fragment 7.3

```
public void enqueue(E elem) {
  Node<E> node = new Node<E>();
  node.setElement(elem);
  node.setNext(null); // node will be new tail node
  if (size == 0)
    head = node; // special case of a previously empty queue
  else
    tail.setNext(node); // add node at the tail of the list
  tail = node; // update the reference to the tail node
  size++;
}
...
public E dequeue() throws EmptyQueueException {
  if (size == 0)
    throw new EmptyQueueException("Queue is empty.");
  E tmp = head.getElement();
  head = head.getNext();
  size--;
  if (size == 0)
    tail = null; // the queue is now empty
  return tmp;
}
```

End