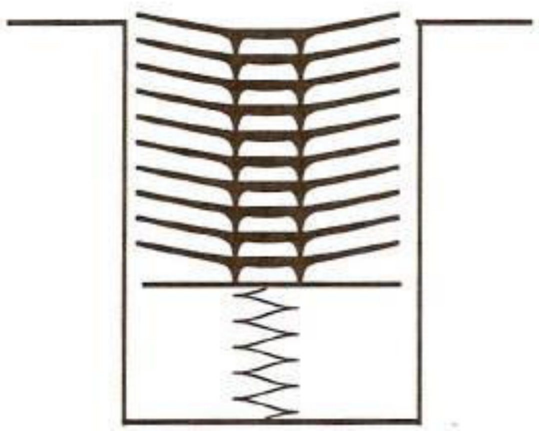
12.3 Stacks and Queues

As we said when we introduced the idea of a linear list, different applications require lists with different operations. It turns out that, for many applications, insertions and deletions are only required at the ends of the list. For such applications, it is often more efficient if we use lists that only have the operations necessary for the problem. In this section, we will examine two of the most important types of linear lists for which there are restrictions on the points at which insertions and deletions can be made.

Stacks

The first of these is the *stack* - a linear list in which all insertions, deletions, and inspections take place at one end. The end at which activity takes place is called the *top* of the stack; the other end is called the *bottom* of the stack. Stacks have a physical analogy in the spring-operated mechanisms sometimes used to store stacks of plates in cafeterias. In these devices, if you place a new plate on the top of the stack, the added pressure pushes the stack down a bit; if you remove a plate from the top, the decreased pressure allows the remaining plates to pop up a bit.



Following this analogy, to insert an item into a stack, we *push* it onto the stack; to delete an item, we *pop* it from the stack. Because of this, stacks are sometimes referred to as *pushdown stores*. They are also referred to as *LIFO* (Last In, First Out) lists because the last item pushed onto the stack will be the first item popped from it.

To make the concept of a stack as an ADT more formal, we can make the following definition: a stack is a linear list with the following operations:

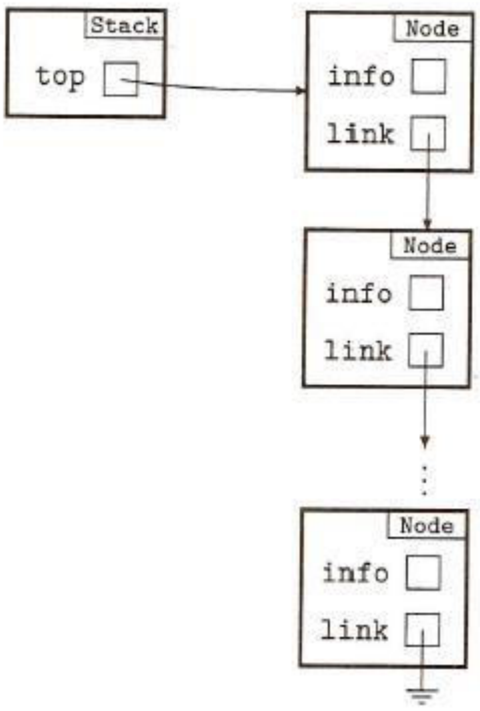
create	create a new, empty stack

-

push	add a new item onto one end of the list (the top)
pop	delete and return the item at the top
isEmpty	return true if and only if the stack is empty
peek	return the value of the item at the top

Java actually supplies a `Stack` class in the package `java.util` with methods for each of these operations but we will not be using it as it would spoil the fun of creating our own class for implementing this data type.

To implement a stack, we *could* use a partially-filled array, a process that we explored in Chapter 8. Here, however, we will examine an alternative structure for stack implementation - dynamic allocation using a linked list. The form of such a linked list is the usual one that we have been using up to now, with each node consisting of `info` and `link` fields. We often draw a stack as shown in the following diagram to reinforce the idea that the first node is at the "top" of the stack but, in fact, there is nothing new in the way that the list is defined.



Example 1

To insert an item in such a stack, we can use the following method (in the class `Stack`):

```
public void push (int item)
{
   top = new Node(item,top);
}
```

Example 2

This method (for the class `Stack`) returns the value currently at the top of the stack without altering the stack.

```
public int peek ()
{
   if (top == null)
      throw new RuntimeException("peek: empty stack");
   else
      return top.info;
}
```

Notice that if the method finds that the stack is empty, there is no value for the method to return. In such a situation, the method throws an exception. Unless we take some action to handle the exception, the program's execution will terminate. Exceptions and exception handling are discussed in Appendix E.

The other methods required for a stack (`pop` and `isEmpty`) are also quite simple. Their implementations are left as exercises.

Queues

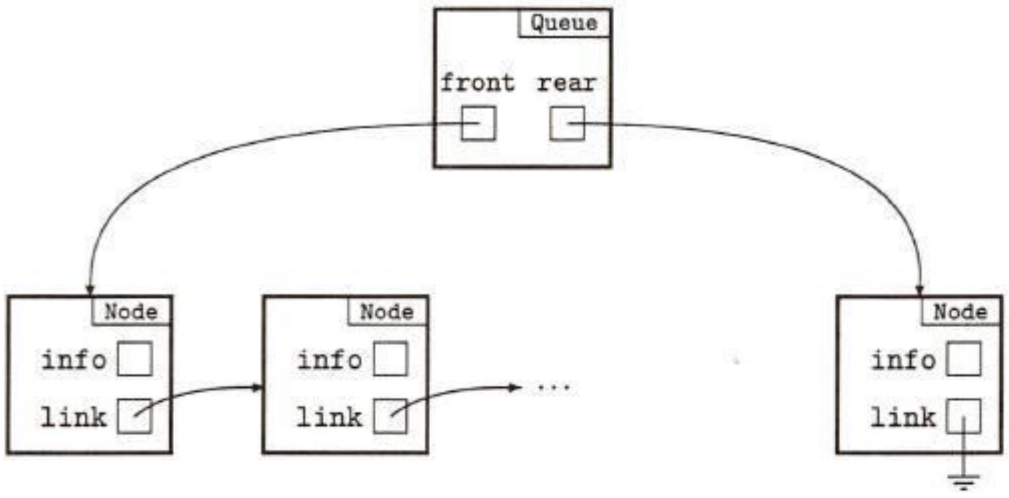
A *queue* is another linear list with restrictions on insertions and deletions. A queue operates like a human queue at a checkout line in a store. All insertions take place at one end, the *rear* of the queue, and all deletions take place at the other end, the *front* of the queue. Because the first item inserted into a queue will be the first item deleted, a queue is also known as a First In, First Out, or *FIFO* list. As we did for a stack, we can make the idea of a queue as an ADT more precise. A queue is a linear list with the following operations:

create	create a new, empty stack

enqueue	add a new item onto one end (the rear) of the list

• dequeue	delete and return the item at the other end (the front) of the list
• isEmpty	return true if and only if the queue is empty
• peek	

To implement a queue, we can again use either sequential allocation with an array or linked allocation. We will look only at linked allocation in this section. Because insertions and deletions take place at opposite ends of the queue, it is useful to have separate references to the front and the rear of the queue. This is illustrated in the next diagram in which the object at the top of the diagram is of type `Queue`.



We can begin to create a `Queue` class by setting up fields for the references to the two ends of the lists along with our usual `Node` inner class.

```
class Queue
{
   private Node front;
   private Node rear;

   class Node
   {
      int info;
      Node link;

      Node (int i, Node n)
```

```
        {
           info = i;
           link = n;
        }
    }
}
```

To insert a new node in a queue, we must examine two cases. If the queue is empty, then both `front` and `rear` must be set to refer to the new node. Otherwise, the new node must be inserted after the node referred to by `rear` after which `rear` must be redirected to refer to the newly inserted node.

Example 3

This method inserts a new node into a list of the `Queue` class.

```
public void enqueue (int item)
{
   Node temp = new Node(item,null);
   if (rear == null)
      // queue was empty
      front = rear = temp;
   else
      // add node at rear of queue
      rear = rear.link = temp;
}
```

Exercise 12.3

1.
   a. The *Back* operation on a Web browser is an application that uses a stack. Give two other common applications of stacks.
      1. **Text Editors**: The undo feature in every text editor is an example of stacks as it undoes the last change that happened to the document. This is the same idea as LIFO.
      2. **Calculators:**

         - If the user decides to insert a value into the calculator, it is usually inserted at the end of other values unless the calculator does not contain any values.
         - If the user decides to click “Back” or “DEL”, the last value that was inserted is deleted.

   b. Find two common applications of queues.
      1. **Printer Application** – All printing applications print in order from the first output to the last output.

2. Write instance methods to implement the pop and isEmpty operations of the `Stack` class. Have your pop method throw an exception (as in Example 2) if the stack is empty.

```
        /*
         * this method deletes and returns the item at the top.
         * pre: none
         * post: returns item at the top
        */
        public void pop () {
               if (top == null)
                      throw new RuntimeException("peek: empty stack");
               else
                      top = top.link;
        }

        /*
         * This method returns true if and only if the stack is empty.
         * pre: none
         * post: True (empty) or False (not empty)
        */
        public boolean isEmpty () {
               if (top == null)
                      return true;
               else
                      return false;

        }
```

3. Write instance methods for the `Queue` class that implement the peek and dequeue operations. Each of the methods should throw an exception (as in Example 2) if the queue is empty.

```
        /*
         *This method returns the value of the item at the front
         *pre: none
         *post: returns item at the front
        */
        public int peek () {
               if (rear == null)
                      throw new RuntimeException("peek: empty queue");
               else
                      return front.info;

        }

        /*
         * This method deletes and returns the item at the other end (the
front) of the list
         * pre: none
         * post: returns front item
        */
        public int dequeue () {
               if (rear == null)
                      throw new RuntimeException("peek: empty queue");
               else {
                      front = front.link;
                      return front.info;
               }

        }
```

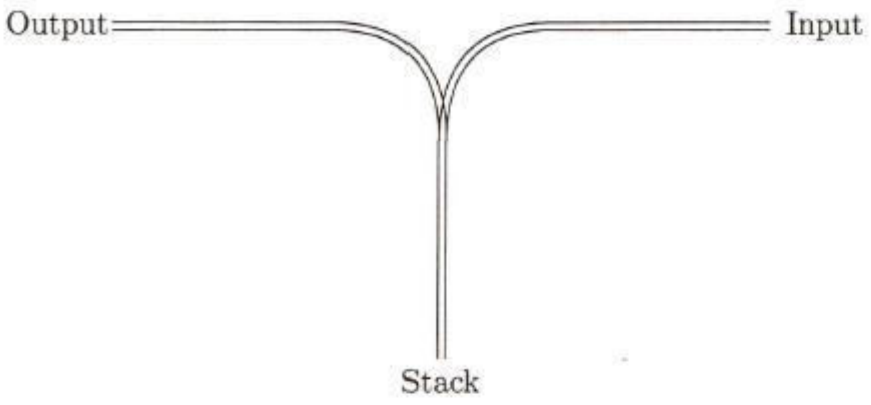
4. In our implementation of a queue, the links in the nodes pointed backward, from the front of the queue to the rear. If the links were in the other direction, would it still be possible to insert and/or delete items efficiently? Justify your answer in each case by writing a suitable method if you think that it is possible or by explaining why you think that it is not possible

   If the links were in the other direction, it would still be possible to insert and/or delete items from the list. However, it's not inefficient to delete an item this way as it requires the program to go through the entire list to find the second last Node or store the second last Node in a separate variable. This is inefficient because it occupies more space in the memory which can slow down the program.

```
/*
         * This method adds a new item onto one end (the rear) of the list
         * pre: item
         * post: none
        */
        public void enqueue (int item) {
           Node temp = new Node(item,rear);
           if (front == null)
              // queue was empty
              front = rear = temp;
           else
              // add node at rear of queue
              rear = temp;

        }

        /*
         * This method deletes and returns the item at the other end (the
front) of the list
         * pre: none
         * post: returns front item
        */
        public int dequeue () {
               if (front == null)
                      throw new RuntimeException("peek: empty queue");
```

```
                else {
                        Node current = rear;
                        while (current.link.link != null)
                                current = current.link;

                        front = current;
                        return front.info;
                }

        }
```

5. A model of a stack is provided by a railroad switching network as shown in the next diagram.



Railroad cars, numbered 1,2,3, ... ,n enter at the right and leave at the left. As cars enter, we can consider them to be pushed onto the stack; as they leave, we can consider them to be popped from the stack. The order in which they leave at the left can be altered (for a given arrival order) by choosing different sequences of push and pop operations. For example, if there are three cars, then the initial order (1,2,3) can be changed to 2,1,3 by performing the following sequence of operations: push, push, pop, pop, push, pop.

1,2

a. Find all the possible output sequences that can be produced from the input sequence 1,2,3.
b. Repeat part (a) for the input sequence 1,2,3,4.