Algorithms

FOURTH EDITION

PART I

ROBERT SEDGEWICK | KEVIN WAYNE

Algorithms

FOURTH EDITION

PART I

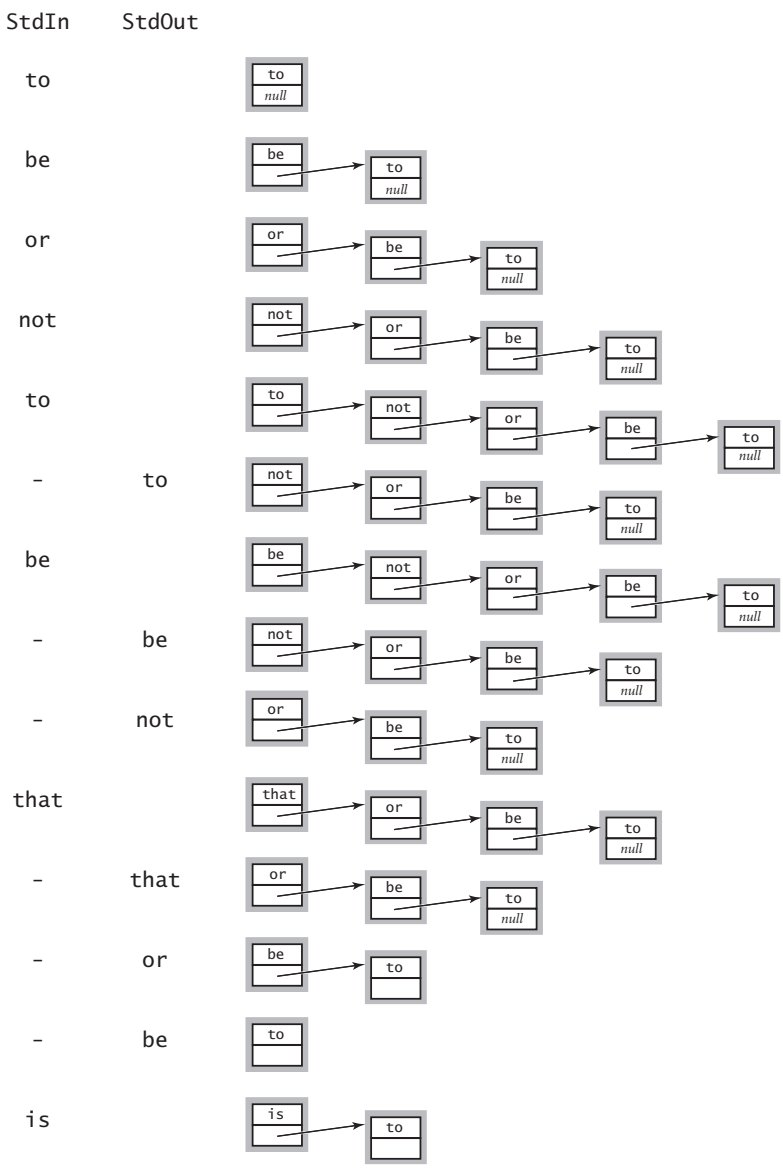
Stack implementation. Given these preliminaries, developing an implementation for our `Stack` API is straightforward, as shown in ALGORITHM 1.2 on page 149. It maintains the stack as a linked list, with the top of the stack at the beginning, referenced by an instance variable `first`. Thus, to `push()` an item, we add it to the beginning of the list, using the code discussed on page 144 and to `pop()` an item, we remove it from the beginning of the list, using the code discussed on page 145. To implement `size()`, we keep track of the number of items in an instance variable `N`, incrementing `N` when we push and decrementing `N` when we pop. To implement `isEmpty()` we check whether `first` is `null` (alternatively, we could check whether `N` is `0`). The implementation uses the generic type `Item`—you can think of the code `<Item>` after the class name as meaning that any occurrence of `Item` in the implementation will be replaced by a client-supplied data-type name (see page 134). For now, we omit the code to support iteration, which we consider on page 155. A trace for the test client that we have been using is shown on the next page. This use of linked lists achieves our optimum design goals:

- It can be used for any type of data.
- The space required is always proportional to the size of the collection.
- The time per operation is always independent of the size of the collection.

This implementation is a prototype for many *algorithm* implementations that we consider. It defines the linked-list *data structure* and implements the client methods `push()` and `pop()` that achieve the specified effect with just a few lines of code. The algorithms and data structure go hand in hand. In this case, the code for the algorithm implementations is quite simple, but the properties of the data structure are not at all elementary, requiring explanations on the past several pages. This interaction between data structure definition and algorithm implementation is typical and is our focus in ADT implementations throughout this book.

```
public static void main(String[] args)
{  // Create a stack and push/pop strings as directed on StdIn.
   Stack<String> s = new Stack<String>();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
           s.push(item);
      else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
   }
   StdOut.println("(" + s.size() + " left on stack)");
}
```

Test client for Stack



Trace of Stack development client

ALGORITHM 1.2 Pushdown stack (linked-list implementation)

```
public class Stack<Item> implements Iterable<Item>
{
   private Node first; // top of stack (most recently added node)
   private int N;      // number of items

   private class Node
   {  // nested class to define nodes
      Item item;
      Node next;
   }

   public boolean isEmpty() {  return first == null; }
   public int size()        {  return N; }

   public void push(Item item)
   {  // Add item to top of stack.
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
      N++;
   }

   public Item pop()
   {  // Remove item from top of stack.
      Item item = first.item;
      first = first.next;
      N--;
      return item;
   }

   // See page 155 for iterator() implementation.

   // See page 147 for test client main().

}
```

This generic `Stack` implementation is based on a linked-list data structure. It can be used to create stacks containing any type of data. To support iteration, add the highlighted code described for `Bag` on page 155.

```
% more tobe.txt
to be or not to - be - - that - - - is

% java Stack < tobe.txt
to be not that or be (2 left on stack)
```

Queue implementation. An implementation of our `Queue` API based on the linked-list data structure is also straightforward, as shown in ALGORITHM 1.3 on the facing page. It maintains the queue as a linked list in order from least recently to most recently added items, with the beginning of the queue referenced by an instance variable `first` and the end of the queue referenced by an instance variable `last`. Thus, to `enqueue()` an item, we add it to the end of the list (using the code discussed on page 145, augmented to set both `first` and `last` to refer to the new node when the list is empty) and to `dequeue()` an item, we remove it from the beginning of the list (using the same code as for `pop()` in `Stack`, augmented to update `last` when the list becomes empty). The implementations of `size()` and `isEmpty()` are the same as for `Stack`. As with `Stack` the implementation uses the generic type parameter `Item`, and we omit the code to support iteration, which we consider in our `Bag` implementation on page 155. A development client similar to the one we used for `Stack` is shown below, and the trace for this client is shown on the following page. This implementation uses the same *data structure* as does `Stack`—a linked list—but it implements different *algorithms* for adding and removing items, which make the difference between LIFO and FIFO for the client. Again, the use of linked lists achieves our optimum design goals: it can be used for any type of data, the space required is proportional to the number of items in the collection, and the time required per operation is always independent of the size of the collection.

```
public static void main(String[] args)
{  // Create a queue and enqueue/dequeue strings.

   Queue<String> q = new Queue<String>();

   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
           q.enqueue(item);
      else if (!q.isEmpty()) StdOut.print(q.dequeue() + " ");
   }

   StdOut.println("(" + q.size() + " left on queue)");
}
```

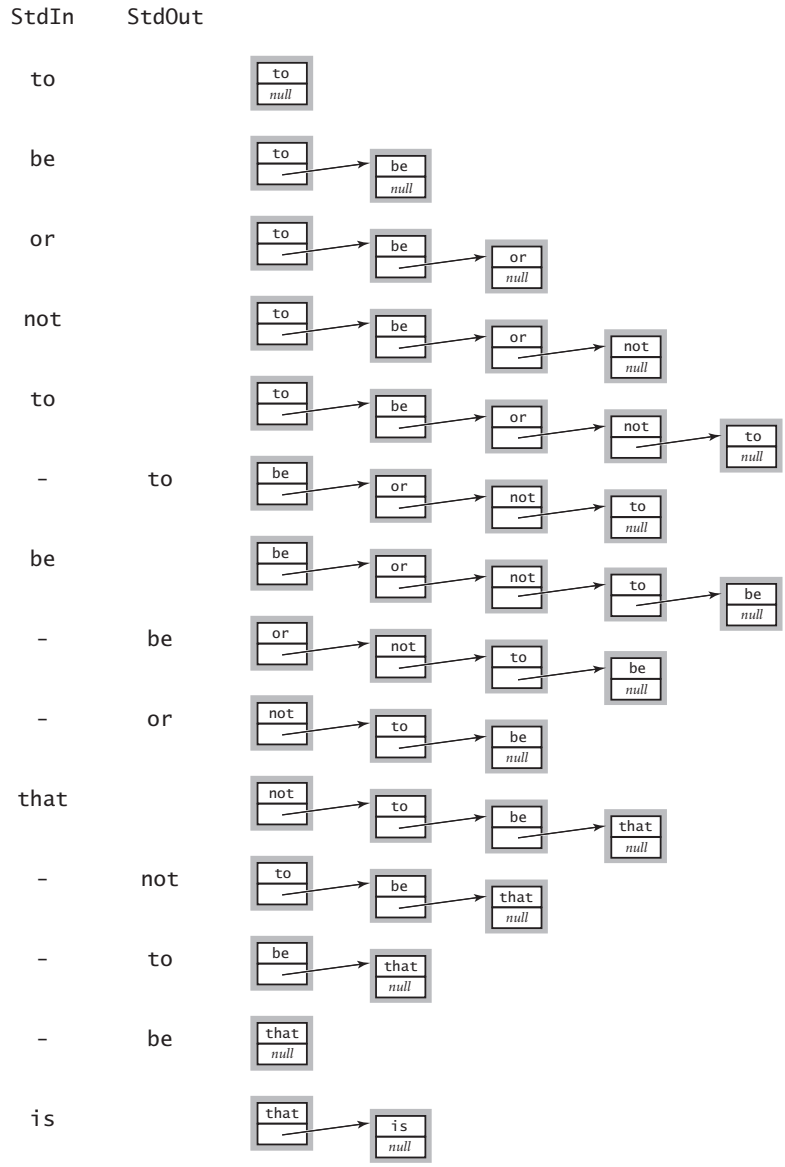
Test client for Queue

```
% more tobe.txt
to be or not to - be - - that - - - is

% java Queue < tobe.txt
to be or not to be (2 left on queue)
```

ALGORITHM 1.3 FIFO queue

```
public class Queue<Item> implements Iterable<Item>
{
   private Node first; // link to least recently added node
   private Node last;  // link to most recently added node
   private int N;      // number of items on the queue

   private class Node
   {  // nested class to define nodes
      Item item;
      Node next;
   }

   public boolean isEmpty() {  return first == null;  }
   public int size()        {  return N;  }

   public void enqueue(Item item)
   {  // Add item to the end of the list.
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
      N++;
   }

   public Item dequeue()
   {  // Remove item from the beginning of the list.
      Item item = first.item;
      first = first.next;
      N--;
      if (isEmpty()) last = null;
      return item;
   }

   // See page 155 for iterator() implementation.

   // See page 150 for test client main().

}
```

This generic `Queue` implementation is based on a linked-list data structure. It can be used to create queues containing any type of data. To support iteration, add the highlighted code described for `Bag` on page 155.



Trace of Queue development client