CHAPTER 3

Lists, Stacks, and Queues

3.1

```
public static <AnyType> void printLots(List<AnyType> L,
List<Integer> P)
{
  Iterator<AnyType> iterL = L.iterator();
  Iterator<Integer> iterP = P.iterator();

  AnyType itemL=null;
  Integer itemP=0;
  int start = 0;

  while ( iterL.hasNext() && iterP.hasNext() )
  {
    itemP = iterP.next();

    System.out.println("Looking for position " + itemP);
    while ( start < itemP && iterL.hasNext() )
    {
      start++;
      itemL = iterL.next();
    }
    System.out.println( itemL );
  }
}
```

3.2 **(a)** For singly linked lists:

```
// beforeP is the cell before the two adjacent cells that are to be
   swapped.
// Error checks are omitted for clarity.

public static void swapWithNext( Node beforep )
{
  Node p, afterp;

  p = beforep.next;
  afterp = p.next;   // Both p and afterp assumed not null.

  p.next = afterp.next;
  beforep.next = afterp;
  afterp.next = p;
}
```

(b) For doubly linked lists:

```
// p and afterp are cells to be switched.  Error checks as before.
```

```
public static void swapWithNext( Node p )
{
  Node beforep, afterp;

  beforep = p.prev;
  afterp = p.next;

  p.next = afterp.next;
  beforep.next = afterp;
  afterp.next = p;
  p.next.prev = p;
  p.prev = afterp;
  afterp.prev = beforep;
}
```

3.3

```
public boolean contains( AnyType x )
{
   Node<AnyType> p = beginMarker.next;
   while( p != endMarker && !(p.data.equals(x) ))
    {
          p = p.next;
    }

   return (p != endMarker);
}
```

3.4

```
public static <AnyType extends Comparable<? super AnyType>>
       void intersection(List<AnyType> L1, List<AnyType> L2,
                                            List<AnyType> Intersect)
{

  ListIterator<AnyType> iterL1 = L1.listIterator();
  ListIterator<AnyType> iterL2 = L2.listIterator();

  AnyType itemL1=null, itemL2=null;

  // get first item in each list
  if ( iterL1.hasNext() && iterL2.hasNext() )
  {
    itemL1 = iterL1.next();
    itemL2 = iterL2.next();
  }

  while ( itemL1 != null && itemL2 != null )
  {
    int compareResult = itemL1.compareTo(itemL2);

    if ( compareResult == 0 )
    {
      Intersect.add(itemL1);
      itemL1 = iterL1.hasNext()?iterL1.next():null;
      itemL2 = iterL2.hasNext()?iterL2.next():null;
    }
    else if ( compareResult < 0 )
    {
      itemL1 = iterL1.hasNext()?iterL1.next():null;
```

```
      }
      else
      {
        itemL2 = iterL2.hasNext()?iterL2.next():null;
      }
    }
  }
```

3.5

```
  public static <AnyType extends Comparable<? super AnyType>>
         void union(List<AnyType> L1, List<AnyType> L2,
                                              List<AnyType> Result)
  {

    ListIterator<AnyType> iterL1 = L1.listIterator();
    ListIterator<AnyType> iterL2 = L2.listIterator();

    AnyType itemL1=null, itemL2=null;

    // get first item in each list
    if ( iterL1.hasNext() && iterL2.hasNext() )
    {
      itemL1 = iterL1.next();
      itemL2 = iterL2.next();
    }

    while ( itemL1 != null && itemL2 != null )
    {
      int compareResult = itemL1.compareTo(itemL2);

      if ( compareResult == 0 )
      {
        Result.add(itemL1);
        itemL1 = iterL1.hasNext()?iterL1.next():null;
        itemL2 = iterL2.hasNext()?iterL2.next():null;
      }
      else if ( compareResult < 0 )
      {
        Result.add(itemL1);
        itemL1 = iterL1.hasNext()?iterL1.next():null;
      }
      else
      {
        Result.add(itemL2);
        itemL2 = iterL2.hasNext()?iterL2.next():null;
      }
    }
  }
```

3.6 A basic algorithm is to iterate through the list, removing every *M*th item. This can be improved by two observations. The first is that an item *M* distance away is the same as an item that is only *M* mod *N* away. This is useful when *M* is large enough to cause iteration through the list multiple times. The second is that an item *M* distance away in the forward direction is the same as an item (*M* – *N*) away in the backwards

direction. This improvement is useful when M is more than $N/2$ (half the list). The solution shown below uses these two observations. Note that the list size changes as items are removed. The worst case running time is $O(N\min(M, N))$, though with the improvements given the algorithm might be significantly faster. If $M = 1$, the algorithm is linear.

```
public static void pass(int m, int n)
{
  int i, j, mPrime, numLeft;

  ArrayList<Integer> L = new ArrayList<Integer>();

  for (i=1; i<=n; i++)
    L.add(i);

  ListIterator<Integer> iter = L.listIterator();
  Integer item=0;

  numLeft = n;
  mPrime = m % n;

  for (i=0; i<n; i++)
  {
    mPrime = m % numleft;
    if (mPrime <= numLeft/2)
    {
      if (iter.hasNext())
            item = iter.next();
      for (j=0; j<mPrime; j++)
      {
        if (!iter.hasNext())
          iter = L.listIterator();

        item = iter.next();
      }
    }
    else
    {
      for (j=0; j<numLeft-mPrime; j++)
      {
        if (!iter.hasPrevious())
          iter = L.listIterator(L.size());

        item = iter.previous();
      }
    }
    System.out.print("Removed " + item + " ");
    iter.remove();
    if (!iter.hasNext())
      iter = L.listIterator();

    System.out.println();
    for (Integer x:L)
```

```
            System.out.print(x + " ");
          System.out.println();
          numLeft--;
        }

        System.out.println();
      }
```

3.7 $O(N^2)$. The trim method reduces the size of the array, requiring each add to resize it. The resize takes $O(N)$ time, and there are $O(N)$ calls.

3.8 **(a)** Because the remove call changes the size, which would affect the loop.

(b) $O(N^2)$. Each remove from the beginning requires moving all elements forward, which takes $O(N)$ time.

(c) $O(N)$. Each remove can be done in $O(1)$ time.

(d) No. Since it is always the first element being removed, finding the position does not take time, thus an iterator would not help.

3.9

```
public void addAll( Iterable<? extends AnyType> items )
{
  Iterator<? extends AnyType> iter = items.iterator();
  while ( iter.hasNext() )
  {
    add(iter.next());
  }
}
```

This runs in $O(N)$ time, where N is the size of the items collection.

3.10

```
public void removeAll( Iterable<? extends AnyType> items )
{
  AnyType item, element;
  Iterator<? extends AnyType> iterItems = items.iterator();

  while ( iterItems.hasNext ( ) )
  {
    item = iterItems.next();
    Iterator<? extends AnyType> iterList = iterator();
    while ( iterList.hasNext ( ) )
    {
      element = iterList.next();
      if ( element.equals(item) )
        iterList.remove();
    }
  }
}
```

This runs in $O(MN)$, where M is the size of the items, and N is the size of the list.

3.11

```
import java.util.*;

public class SingleList
```

```
{
  SingleList()
  { init(); }

  boolean add( Object x )
  {
    if (contains(x))
      return false;
    else
    {
      Node<Object> p = new Node<Object>(x);
      p.next = head.next;
      head.next = p;
      theSize++;
    }
    return true;
  }

  boolean remove(Object x)
  {
    if (!contains(x))
      return false;
    else
    {
      Node<Object> p = head.next;
      Node<Object> trailer = head;
      while (!p.data.equals(x))
      {
        trailer = p;
        p = p.next;
      }
      trailer.next = p.next;
      theSize--;
    }
    return true;
  }

  int size()
  { return theSize; }

  void print()
  {
    Node<Object> p = head.next;
    while (p != null)
    {
      System.out.print(p.data + " ");
      p = p.next;
    }
    System.out.println();
  }

  boolean contains( Object x )
  {
    Node<Object> p = head.next;
    while (p != null)
    {
      if (x.equals(p.data))
```

```
        return true;
      else
        p = p.next;
    }
    return false;
  }

  void init()
  {
    theSize = 0;
    head = new Node<Object>();
    head.next = null;
  }

  private Node<Object> head;
  private int theSize;

  private class Node<Object>
  {
    Node()
    {
      this(null, null);
    }
    Node(Object d)
    {
      this(d, null);
    }
    Node(Object d, Node n)
    {
      data = d;
      next = n;
    }
    Object data;
    Node next;
  }
}
```

3.12

```
import java.util.*;

public class SingleListSorted
{
  SingleListSorted()
  { init(); }

  boolean add( Comparable x )
  {
    if (contains(x))
      return false;
    else
    {
      Node<Comparable> p = head.next;
      Node<Comparable> trailer = head;
      while (p != null && p.data.compareTo(x) < 0)
      {
        trailer = p;
        p = p.next;
      }
```

```
    trailer.next = new Node<Comparable>(x);
    trailer.next.next = p;
    theSize++;
  }
  return true;
}

boolean remove(Comparable x)
{
  if (!contains(x))
    return false;
  else
  {
    Node<Comparable> p = head.next;
    Node<Comparable> trailer = head;
    while (!p.data.equals(x))
    {
      trailer = p;
      p = p.next;
    }
    trailer.next = p.next;
    theSize--;
  }
  return true;
}

int size()
{ return theSize; }

void print()
{
  Node<Comparable> p = head.next;
  while (p != null)
  {
    System.out.print(p.data + " ");
    p = p.next;
  }
  System.out.println();
}

boolean contains( Comparable x )
{
  Node<Comparable> p = head.next;
  while (p != null && p.data.compareTo(x) <= 0)
  {
    if (x.equals(p.data))
      return true;
    else
      p = p.next;
  }
  return false;
}

void init()
{
  theSize = 0;
```

```
    head = new Node<Comparable>();
    head.next = null;
  }

  private Node<Comparable> head;
  private int theSize;

  private class Node<Comparable>
  {
    Node()
    {
      this(null, null);
    }
    Node(Comparable d)
    {
      this(d, null);
    }
    Node(Comparable d, Node n)
    {
      data = d;
      next = n;
    }
    Comparable data;
    Node next;
  }
}
```

3.13

```
  public java.util.Iterator<AnyType> iterator()
  { return new ArrayListIterator( ); }

  public java.util.ListIterator<AnyType> listIterator()
  { return new ArrayListIterator( ); }

  private class ArrayListIterator implements
  java.util.ListIterator<AnyType>
  {
     private int current = 0;
     boolean backwards = false;

     public boolean hasNext()
     { return current < size(); }

     public AnyType next()
     {
        if ( !hasNext() )
           throw new java.util.NoSuchElementException();
        backwards = false;
        return theItems[ current++ ];
     }

     public boolean hasPrevious()
     { return current > 0; }

     public AnyType previous()
     {
        if ( !hasPrevious() )
           throw new java.util.NoSuchElementException();
```

```
            backwards = true;
            return theItems[ --current ];
        }

        public void add( AnyType x )
        { MyArrayList.this.add( current++, x ); }

        public void remove()
        {
            if (backwards)
                MyArrayList.this.remove( current-- );
            else
                MyArrayList.this.remove( --current );
        }

        public void set( AnyType newVal )
        { MyArrayList.this.set( current, newVal ); }

        public int nextIndex()
        {
            throw new java.lang.UnsupportedOperationException();
        }

        public int previousIndex()
        {
            throw new java.lang.UnsupportedOperationException();
        }
    }
```

3.14

```
    public java.util.Iterator<AnyType> iterator()
    {
      return new LinkedListIterator( );
    }

    public java.util.ListIterator<AnyType> listIterator()
    {
      return new LinkedListIterator( );
    }

    private class LinkedListIterator implements
    java.util.ListIterator<AnyType>
    {
      private Node<AnyType> current = beginMarker.next;
      private int expectedModCount = modCount;
      private boolean okToRemove = false;

      public boolean hasNext()
      {  return current != endMarker; }

      public AnyType next()
      {
        if( modCount != expectedModCount )
            throw new java.util.ConcurrentModificationException( );
        if( !hasNext( ) )
            throw new java.util.NoSuchElementException( );

        AnyType nextItem = current.data;
```

```
    current = current.next;
    okToRemove = true;
    return nextItem;
  }

  public boolean hasPrevious()
  { return current.prev != beginMarker; }

  public AnyType previous()
  {
    if ( modCount != expectedModCount )
      throw new java.util.ConcurrentModificationException();
    if ( !hasPrevious() )
      throw new java.util.NoSuchElementException();

    current = current.prev;
    AnyType previousItem = current.data;
    okToRemove = true;
    return previousItem;
  }


  public void add( AnyType x )
  {
    if ( modCount != expectedModCount )
      throw new java.util.ConcurrentModificationException();

    MyLinkedList.this.addBefore( current.next, x );
  }

  public void remove()
  {
    if( modCount != expectedModCount )
      throw new java.util.ConcurrentModificationException( );
    if( !okToRemove )
      throw new IllegalStateException( );

    MyLinkedList.this.remove( current.prev );
    okToRemove = false;
  }

  public void set( AnyType newVal )
  {
    if ( modCount != expectedModCount )
       throw new java.util.ConcurrentModificationException();

    MyLinkedList.this.set( current.next, newVal );
  }

  public int nextIndex()
  {
    throw new java.lang.UnsupportedOperationException();
  }

  public int previousIndex()
  {
    throw new java.lang.UnsupportedOperationException();
```

```
    }

  }

// and change MyLinkedList as follows:

  public AnyType set( int idx, AnyType newVal )
  {   return set( getNode( idx ), newVal ); }

  private AnyType set( Node<AnyType> p, AnyType newVal )
  {
     AnyType oldVal = p.data;
     p.data = newVal;
     return oldVal;
  }
```

3.16

```
  Iterator<AnyType> reverseIterator()
  { return new ArrayListReverseIterator( ); }

  private class ArrayListReverseIterator implements
  java.util.Iterator<AnyType>
  {
    private int current = size( ) - 1;

    public boolean hasNext()
    { return current >= 0; }

    public AnyType next()
    {
      if ( !hasNext() )
         throw new java.util.NoSuchElementException();
      return theItems[ current-- ];
    }

    public void remove()
    { MyArrayList.this.remove( --current ); }

  }
```

3.18

```
  public void addFirst( AnyType x )
  { addBefore( beginMarker.next, x ); }

  public void addLast( AnyType x )
  { addBefore( endMarker, x ); }

  public void removeFirst( )
  { remove( beginMarker.next ); }

  public void removeLast( )
  { remove( endMarker.prev ); }

  public AnyType getFirst( )
  { return get( 0 ); }

  public AnyType getLast( )
  { return get( size( ) - 1 ); }
```

3.19 Without head or tail nodes the operations of inserting and deleting from the end becomes an $O(N)$ operation where the N is the number of elements in the list. The algorithm must walk down the list before inserting at the end. Without the head node insert needs a special case to account for when something is inserted before the first node.

3.20 **(a)** The advantages are that it is simpler to code, and there is a possible saving if deleted keys are subsequently reinserted (in the same place). The disadvantage is that it uses more space, because each cell needs an extra bit (which is typically a byte), and unused cells are not freed.

3.22 The following function evaluates a postfix expression, using + ,–, *, /, and ^ ending in =. It requires spaces between all operators and =.

```
static double evalPostFix()
{
  Stack<Double> s = new Stack<Double>();
  String token;
  Double a, b, result=0.0;
  boolean isNumber;

  Scanner sc = new Scanner(System.in);
  token = sc.next();
  while (token.charAt(0) != '=')
  {
    try
    {
      isNumber = true;
      result = Double.parseDouble(token);
    }
    catch (Exception e)
    {
      isNumber = false;
    }
    if (isNumber)
      s.push(result);
    else
    {
      switch (token.charAt(0))
      {
        case '+': a = s.pop(); b = s.pop();
              s.push(a+b); break;
        case '-': a = s.pop(); b = s.pop();
              s.push(a-b); break;
        case '*': a = s.pop(); b = s.pop();
              s.push(a*b); break;
        case '/': a = s.pop(); b = s.pop();
              s.push(a/b); break;
        case '^': a = s.pop(); b = s.pop();
              s.push(Math.exp(a*Math.log(b)));
              break;
      }
```

```
    }

    token = sc.next();
  }

  return s.peek();
}
```

3.23 **(a, b)** This function will read in from standard input an infix expression of single lower case characters and the operators +, –, /, *, ^, and ( ), and outputs a postfix expression.

```
static void InFixToPostFix()
{
  Stack<Character> s = new Stack<Character>();
  String expression;
  Character token;
  int i=0;

  Scanner sc = new Scanner(System.in);
  expression = sc.next();

  while ((token = expression.charAt(i++)) != '=')
  {
    if (token >= 'a' && token <= 'z')
       System.out.print(token + " ");
    else
    {
       switch (token)
       {
        case ')': while ( !s.empty() && s.peek() != '(' )
                  { System.out.print(s.pop() + " "); }
                  s.pop();
                  break;
        case '(': s.push(token);
                  break;
        case '^': while ( !s.empty() && !(s.peek() == '^' ||
                                          s.peek() == '(') )
                  { System.out.print(s.pop()); }
                  s.push(token);
                  break;
        case '*':
        case '/': while ( !s.empty() && s.peek() != '+' &&
                         s.peek() != '-' && s.peek() != '(' )
                  { System.out.print(s.pop()); }
                  s.push(token);
                  break;
        case '+':
        case '-': while ( !s.empty() && s.peek() != '(' )
                  { System.out.print(s.pop() + " "); }
                  s.push(token);
                  break;
       }
    }
  }
```

```
    while (!s.empty())
    { System.out.print(s.pop()); }
    System.out.println();
  }
```

3.24 Two stacks can be implemented in an array by having one grow from the low end of the array up, and the other from the high end down.

3.25 **(a)** Let E be our extended stack. We will implement E with two stacks. One stack, which we'll call S, is used to keep track of the *push* and *pop* operations, and the other M, keeps track of the minimum. To implement *E.push* (x), we perform *S.push* (x). If x is smaller than or equal to the top element in stack M, then we also perform *M.push* (x). To implement *E.pop*( ) we perform *S.pop*( ). If x is equal to the top element in stack M, then we also *M.pop*( ). *E.findMin*( ) is performed by examining the top of M. All these operations are clearly $O(1)$.

(b) This result follows from a theorem in Chapter 7 that shows that sorting must take $\Omega(N \log N)$ time. $O(N)$ operations in the repertoire, including *deleteMin*, would be sufficient to sort.

3.26 Three stacks can be implemented by having one grow from the bottom up, another from the top down and a third somewhere in the middle growing in some (arbitrary) direction. If the third stack collides with either of the other two, it needs to be moved. A reasonable strategy is to move it so that its center (at the time of the move) is halfway between the tops of the other two stacks.

3.27 Stack space will not run out because only 49 calls will be stacked. However, the running time is exponential, as shown in Chapter 2, and thus the routine will not terminate in a reasonable amount of time.

3.28 Since the `LinkedList` class supports adding and removing from the first and end of the list, the `Deque` class shown below simply wraps these operations.

```
public class Deque<AnyType>
{
  Deque()
  { L = new LinkedList<AnyType>(); }

  void push(AnyType x)
  { L.addFirst(x); }

  AnyType pop()
  { return L.removeFirst(); }

  void inject(AnyType x)
  { L.addLast(x); }
```

```
  AnyType eject()
  { return L.removeLast(); }

  LinkedList<AnyType> L;
}
```

3.29 Reversal of a linked list can be done recursively using a stack, but this requires $O(N)$ extra space. The following solution is similar to strategies employed in garbage collection algorithms. At the top of the while loop, the list from the start to *previousPos* is already reversed, whereas the rest of the list, from *currentPos* to the end is normal. This algorithm uses only constant extra space. This solution reverses the list which can then be printed in reverse order.

```
void reverseList()
{
  Node currentPos, nextPos, previousPos;

  previousPos = null;
  currentPos = head.next;  // skip header node
  nextPos = currentPos.next;

  while( nextPos != null)
  {
    currentPos.next = previousPos;
    previousPos = currentPos;
    currentPos = nextPos;
    nextPos = nextPos.next;
  }

  currentPos.next = previousPos;

  head.next = currentPos;
}
```

3.30 **(c)** This follows well-known statistical theorems. See Sleator and Tarjan's paper in Chapter 11 for references.

3.31

```
public class SingleStack<AnyType>
{

  SingleStack()
  {
    head = null;
  }

  void push(AnyType x)
  {
    Node<AnyType> p = new Node<AnyType>(x, head);
    head = p;
  }

  AnyType top()
```

```
  {
    return head.data;
  }

  void pop()
  {
    head = head.next;
  }

  private class Node<AnyType>
  {
    Node()
    { this(null, null); }
    Node(AnyType x)
    { this(x, null); }
    Node(AnyType x, Node p)
    {
      data = x;
      next = p;
    }
    AnyType data;
    Node next;
  }

  private Node<AnyType> head;

}
```

3.32

```
public class SingleQueue<AnyType>
{
  SingleQueue()
  {
    front = null;
    rear = null;
  }

  void enqueue(AnyType x)
  {
    Node<AnyType> p = new Node<AnyType>(x, null);
    if (rear != null)
       rear = rear.next = p;
    else
       front = rear = p;
  }

  AnyType dequeue()
  {
    AnyType temp = front.data;
    Node<AnyType> p = front;
    if (front.next == null)  // only 1 node
       front = rear = null;
    else
       front = front.next;

    return temp;
  }
```

```
  private class Node<AnyType>
  {
    Node()
    { this(null, null); }
    Node(AnyType x)
    { this(x, null); }
    Node(AnyType x, Node p)
    {
      data = x;
      next = p;
    }
    AnyType data;
    Node next;
  }

  private Node<AnyType> front, rear;
}
```

3.33

```
import java.util.*;

public class SingleQueueArray<AnyType>
{

  SingleQueueArray()
  { this(101); }  // note:  actually holds one less than given size

  SingleQueueArray(int s)
  {
    maxSize = s;
    front = 0;
    rear = 0;
    elements = new ArrayList<AnyType>(maxSize);
  }

  void enqueue(AnyType x)
  {
    if ( !full() )
    {
      if (elements.size() < maxSize)  // add elements until size is
      reached
        elements.add(x);
      else
        elements.set(rear, x);  // after size is reached, use set

      rear = (rear + 1) % max Size;
    }
  }

  AnyType dequeue()
  {
    AnyType temp=null;

    if ( !empty() )
    {
      temp = elements.get(front);
      front = (front+1) % max Size;
    }
```

```
    return temp;
  }

  boolean empty()
  { return front == rear; }

  boolean full()
  { return (rear + 1) % max Size; == front; )


  private int front, rear;
  private int maxSize;
  private ArrayList<AnyType> elements;
}
```

3.34 **(b)** Use two iterators *p* and *q*, both initially at the start of the list. Advance *p* one step at a time, and *q* two steps at a time. If *q* reaches the end there is no cycle; otherwise, *p* and *q* will eventually catch up to each other in the middle of the cycle.

3.35 **(a)** Does not work in constant time for insertions at the end.

(b) Because of the circularity, we can access the front item in constant time, so this works.

3.36 Copy the value of the item in the next node (that is, the node that follows the referenced node) into the current node (that is, the node being referenced). Then do a deletion of the next node.

3.37 **(a)** Add a copy of the node in position *p* after position *p*; then change the value stored in position *p* to *x*.

(b) Set `p.data = p.next.data` and set `p.next = p.next.next`. Note that the tail node guarantees that there is always a next node.