1. Exercise 12 (page 329)

(a) Draw a diagram of how the stack might look.
smallTop: top for the stack of small values, initialized to -1 and incremented.
largeTop: top for the stack of large values, initialized to 200 and decremented.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td>197</td>
</tr>
<tr>
<td>198</td>
<td>199</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b)
class DStack
{
    public:
        void Push(int item);
        void PopLarge(int& item);
        void PopSmall(int& item);
    private:
        int small;
        int large;
        int item[200];
}

c(void Push(int item)
{
    if (item <= 1000)
    {
        small++;
        items[small] = item;
    }
    else
    {
        large--;
        items[large] = item;
    }
}

2. Exercises 19, 20

19. No, this sequence is not possible.
20. Yes, this sequence is possible.

3. Exercise 30 (page 335)

(a) Set secondElement to the second element in the queue, leaving the queue without its original front two elements.
{ queue.Dequeue(secondElement);
  queue.Dequeue(secondElement);
}

(b) Set last equal to the rear element in the queue, leaving the queue empty.
{
while (!queue.IsEmpty())
    queue.Dequeue(last);
}

(c) Set last equal to the rear element in the queue, leaving the queue unchanged.
{
    QueType tempQ;
    ItemType item;

    while (!queue.IsEmpty())
    {
        queue.Dequeue(last);
        tempQ.Enqueue(last);
    }
    while (!tempQ.IsEmpty())
    {
        tempQ.Dequeue(item);
        queue.Enqueue(item);
    }
}

(d) A copy of the queue, leaving the queue unchanged.
{
    QueType<ItemType> tempQ;
    ItemType item;

    while (!queue.IsEmpty())
    {
        queue.Dequeue(item);
        tempQ.Enqueue(item);
    }
    while (!tempQ.IsEmpty())
    {
        tempQ.Dequeue(item);
        queue.Enqueue(item);
        copy.Enqueue(item);
    }
}

5.

A binary search of 1 million elements would require $\log_2(1,000,000)$ or about 20 iterations at most (i.e., worst case). A linear search of 1000 elements would require 500 iterations on the average (i.e., going halfway through the array). Therefore, binary search would be $500/20=25$ faster (in terms of iterations) than linear search. However, since linear search iterations are twice as fast as binary search, binary search would be $25/2$ or about 12 times faster than linear search overall, on the same machine. Since we ran them on different machines, where an instruction on the 5-GHz machine is 5 times faster than an instruction on a 1-GHz machine, binary search would be $12/5$ or about 2 times faster than linear search! The key idea is that software improvements can make an algorithm run much faster without having to use more powerful software.