Problem 5 (12 points): A queue of integers is said to be a \textit{breadth-first queue} if all the elements have the same value or if all of the elements at the front of the queue are one less than all the elements at the end and there is nothing else in between (so \([\ ]\), \([1\ 1\ 1]\), and \([1\ 1\ 1\ 2\ 2]\) are all breadth-first queues but \([1\ 2\ 2\ 3\ 3]\) and \([1\ 1\ 3\ 2\ 2]\) are not).

(a) Write a method \texttt{int massDequeue(ItemType& dequeued)} that removes all equal values from the front of the queue, returning the number removed and copying the value to the reference parameter. The precondition for \texttt{massDequeue} is that the queue is not empty.

\begin{verbatim}
int Queue::massDequeue(ItemType& dequeued)
{
    dequeue(dequeued); // get the first element
    int count = 1; // keep track of how many were dequeued

    // get elements that are equal to the first dequeued
    while (first->info == dequeued)
    {
        dequeue(dequeued);
        count++;
    }

    return count;
}
\end{verbatim}

(b) Write a client function that takes a queue of \texttt{ints} as a parameter and returns true exactly when that queue is a breadth-first queue. The queue should not be changed. (Hint: use \texttt{massDequeue}.)

\begin{verbatim}
bool isBFQueue(IQueue q)
{
    if (q.isEmpty())
        return true;

    int first;
    q.massDequeue(first);

    if (q.isEmpty())
        return true;

    int second;
    q.massDequeue(second);

    return (q.isEmpty() && first == second - 1);
}
\end{verbatim}
Problem 6 (14 points): To save space when the same value is repeatedly added to a queue, we can keep track of a repeat count for each item that records how many consecutive values equal to that item are on the queue. Consider such an implementation that uses a dynamically allocated array of (info, count) pairs. [] would be a queue with an empty array. [1 2 ] and [1 1 2 2 2 2] would both be stored using two array elements; the counts would be 1 and 1 in the first case and 2 and 3 in the second. [1 2 1] would use 3 array elements with counts all 1.

(a) Write the enqueue method for this new implementation. You may assume that you can use the == and = operators with ItemType. enqueue should resize the array if necessary.

```cpp
void IQueue::enqueue(const ItemType& toAdd)
{
    if (numNodes == 0 || items[numNodes - 1].info != toAdd)
    {
        if (numNodes == maxNodes)
        {
            maxNodes = maxNodes * 2;
            Node* newItems = new Node[maxNodes];
            for (int i = 0; i < numNodes; i++)
                newItems[i] = items[i];
            delete[] items;
            items = newItems;
        }
        items[numNodes].info = toAdd;
        items[numNodes].count = 1;
        numNodes++;
    }
    else
    {
        items[numNodes - 1].count++;
    }
    numItems++;
}
```

(b) Write a member function that determines if a Queue is a breadth-first queue. You should assume that ItemType is int. The resulting function should have a better asymptotic worst-case running time than the client function you wrote in part (a).

```cpp
bool IQueue::isBFQueue() const
{
    return (numNodes < 2
            || (numNodes == 2 && items[0].info == items[1].info - 1));
}
```

(c) What is the asymptotic worst-case running time of your member function? $O(1)$
Problem 7 (12 points): Suppose there is an ADT that has two different implementations, X and Y. The ADT specifies three methods foo, bar, and baz. The asymptotic worst-case running times for each method using each implementation are given in the table below.

<table>
<thead>
<tr>
<th></th>
<th>foo</th>
<th>bar</th>
<th>baz</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Y</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

(a) Suppose algorithm A makes $O(n \log n)$ calls to foo, $O(1)$ calls to bar and $O(n)$ calls to baz. Which implementation of the ADT gives a better asymptotic worst-case running time for A? Explain your answer.

Using X: $O(n \log n + n + n \log n) = O(n \log n)$. Using Y: $O(n \log^2 n + n \log n + n) = O(n \log^2 n)$.

X is better.

(b) Fill in the blank to make the following statement true: if algorithm B makes $O(n^2)$ calls to foo, $O(1)$ calls to bar, and $O(n^3)$ calls to baz then using implementation Y gives a better worst-case asymptotic running time for B.
Problem 8 (12 points):

(a) Write a function that takes a pointer to a Node in a binary tree not a binary search tree as a parameter and returns true exactly when all nodes in the subtree rooted at that node have keys that are less than the keys in their children (that is, the nodes obey the heap ordering property). The Node structure has three members key, left, and right. You may assume that the key values are ints.

```cpp
bool List::hasHeapOrderProperty(const Node* tree)
{
    return (tree == NULL || ((tree->left == NULL
        || (tree->info < tree->left->info
            && hasHeapOrderProperty(tree->left)))
        && (tree->right == NULL
            || (tree->info < tree->right->info
                && hasHeapOrderProperty(tree->right))));
}
```

(b) Write a function void traverseBottom(Node *tree, Queue& q, int depth) that adds to the queue, in right to left order, the depths of all nodes in the subtree rooted at tree that don’t have two children.

```cpp
void List::traverseBottom(Node* tree, IQueue& q, int depth)
{
    if (tree != NULL)
    {
        traverseBottom(tree->right, q, depth + 1);
        if (tree->left == NULL || tree->right == NULL)
            q.enqueue(depth);
        traverseBottom(tree->left, q, depth + 1);
    }
}
```