Q13
It makes sense to store heaps in an array because a heap is built and maintained so that it is always a complete tree. Hence, if there are 10 items in the heap, then array locations 1-10 are used in the array . By comparison, a general binary tree could have a lot of empty space. For example, consider a “tree” with 10 items in which only the right link is ever used. Then the 10 items would be stored at positions: 0, 2, 6, 14, 30, 62, 126, 254, 510, 1022. Hence, you would need an array with size 1023 just to be sure to be able to store a tree of height 10 that actually has only 10 items. Clearly this wastes a lot of space, and the space wastage only gets worse with increasing depth. So, it you know the max height of a tree, or are not worried about limiting the height, it may make sense to use an array to store a general binary tree. Otherwise, not.

Q14 Very little change is really required from the standard remove function. I could have simplified things a little by rather than using the code to get the inorder successor, I write code to get a leaf. In this case, I could then immediately delete the found left, return its payload and not have to do more recursion. But, by not making this change, I saved some code writing.

```
public boolean removeUnordered(E element) {
        if (root == null)
            return false;
        return removeunorderedUtil(root, null, element);
    }

    /**
     * Internal, recursive implementation of remove
     *
     * @param treepart    the root of the current subtree
     * @param parent      the parent of the root of the current subtree
     * @param toBeRemoved the element to be removed.
     * @return true iff toBeRemoved is in the tree.
     */
      private boolean removeUnorderedUtil(Node<E> treepart, Node<E>
parent, E toBeRemoved) {
        if (treepart == null)
            return false;
        int cmp = treepart.payload.compareTo(toBeRemoved);
        if (cmp!=0) {
            boolean b = removeUnorderedUtil(treepart.left, treepart,
toBeRemoved);
            if (b) {
                return true;
            } else {
                return removeUnorderedUtil(treepart.right, treepart,
toBeRemoved);
            }
        } else {
            // This code is identical to the code for remove
        }

    }
```

Q17:

```
      public int countFullNodes() {
        return countFullNodesUtil(root);
    }

    private int countFullNodesUtil(Node<E> node) {
        if (node == null)
            return 0;
        int incr = (node.right != null && node.left != null) ? 1 : 0;
        return incr + countFullNodesUtil(node.left) +
countFullNodesUtil(node.right);
    }
```

Q21
Consider building a BST with the items 1,2,3,4,5 in that order. This results in a tree with the root at 1 and only right links used until you get to 5. On the other hand, if you receive the same number in the order 5,4,3,2,1 then you would have a root at 5 and only left links used until you get to 1. So BSTs are certainly order dependent
Balancing eliminates some but not all order dependence. For instance, given the numbers 1,2,3 it does not matter the order in which they were received. Balancing will always result in a tree with a root at 2 with 1 to the left and 3 to the right. However, given just 1,2 the order does matter. In one case the tree has a root of 2 and 1 to the left; in the other case the root is at 1 with 2 to the right.

Q8.

java.lang.String is final and cannot be extended. Hence, this question could be considered a “trick” question, or just poorly thought out. Lets assume poorly thought out. So, rather than

```
public class StringFrequencyOrder extends String
```

lets just make a class with a method

```
public int freqOrder(String s1, String s2)
```

which returns the order of these two strings using the method described in the question.

```
public class FString {
    HashMap<Character, Integer> lookup;
    final char[] corder = { 'e', 't', 'a', 'o', 'i', 'n', 's', 'h',
'r', 'd', 'l', 'c', 'u', 'm', 'w', 'f', 'g', 'y',
            'p', 'b', 'v', 'x', 'j', 'x', 'q', 'z' };

    public FString() {
        lookup = new HashMap<>();
        for (int i = 0; i < corder.length; i++)
            lookup.put(corder[i], i);
    }

    int freqOrder(String s1, String s2) {
        int l1 = s1.length();
```

```
        int l2 = s2.length();
        int luse = (l1 > l2) ? l2 : l1;
        for (int i = 0; i < luse; i++) {
            int c1 = lookup.get(s1.charAt(i));
            int c2 = lookup.get(s2.charAt(i));
            if (c1 < c2)
                return -1;
            if (c1 > c2)
                return 1;
        }
        // only here is every char is the same
        if (l1 < l2)
            return -1;
        if (l1 > l2)
            return 1;
        return 0;
    }
// Bonus - some tests of the correctness of the implemenation
public static void main(String[] args) {
        FString ff = new FString();
        System.out.println(ff.freqOrder("e", "e"));
        System.out.println(ff.freqOrder("e", "a"));
        System.out.println(ff.freqOrder("efghi", "efght"));
    }
}
```

Q8. Yes and No. In terms of asymptotic analysis, using a O(n) procedure to build the heap does not matter because the time to remove items from the heap remains O(log n). Hence the overall complexity of sorting is O(n) + O(n*log n) = O(n + n * log n). That is the complexity remains O(n*log n). On the other hand, the change will affect the observed runtime of the algorithm. (It will speed the algorithm). So, it does matter in a practical sense .. speedup is good. In a theoretical sense you have not changed the core behavior of the algorithm.

Q5.
Note a nice main function illustrating that the 2 queue system works as a stack. This code used the class java.util.LinkedList for its queue implementation and java.util.Queue for the queue interface. I could have used classed discussed in class instead. It was simply more convenient to use these java standards.
In terms of algorithmic analysis, the push is clearly O(1), there is only one statement. Actually, that one statement could hide anything, but in this case it is safe to assume thatthe one statement hides O(1) implementation of offer. The pop method is O(n) again assuming an O(1) implementation of queue poll. There are two loops over the entire contents of the queue, each of which therefore takes O(n) time. But the two loops do not interact. So we have O(n) +O(n)=O(n+n)=O(n)

```
import java.util.LinkedList;
import java.util.Queue;

public class SQueue<E> {
```

```
    Queue<E> queueA;
    Queue<E> queueB;

    public SQueue() {
        queueA = new LinkedList<E>();
        queueB = new LinkedList<E>();
    }

    public void push(E e) {
        queueA.offer(e);
    }

    public E pop() {
        if (queueA.size() == 0)
            return null;
        if (queueA.size() == 1) {
            return queueA.poll();
        }
        for (int i = 0; i < queueA.size() - 0; i++)
            queueB.offer(queueA.poll());
        E tmp = queueA.poll();
        while (!queueB.isEmpty())
            queueA.offer(queueB.poll());
        return tmp;
    }

    public static void main(String[] args) {
        SQueue<String> sqs = new SQueue<>();
        sqs.push("AA");
        sqs.push("BB");
        sqs.push("CC");
        System.out.println(sqs.pop());
        System.out.println(sqs.pop());
        sqs.push("DD");
        sqs.push("EE");
        System.out.println(sqs.pop());
        System.out.println(sqs.pop());
        System.out.println(sqs.pop());
        System.out.println(sqs.pop());
    }
}
```

Q4:
The worst case running time is O(n). This will tend to occur when C==n-1. That is when the number of items in the hashtable is equal to 1 less that the size of the hashtable. For linear probing this situation will definitely result in an average run time of n/2 with a min of 0 and a max of n. Quadratic probing is harder to analyze but should be similar (in fact, quadratic probing from a given starting position using the standard implementation will only ever see about half of the possible positions, so it this case the time might be undefined. With some minor modifications (not discussed in class) quadratic probing will see all locations in about 3n

probes. So the time to put in that last item is between 1 and 3n which is O(n). (Note that this answer goes a lot deeper than I would expect, indeed than I thought when I wrote the question. Simply arguing that the time would be O(n) is enough.)

Problem 1:

```
      public int countChildren() {
        return countChildrenUtil(root);
    }
    private int countChildrenUtil(Q1Node<S> n) {
        if (n==null) return 0;
        int cc = 1;
        Q1Node<S> clist = n.firstChild;
        while (clist!=null) {
            cc += countChildrenUtil(clist);
            clist = clist.next;
        }
        return cc;
    }
```

Question 5:
I understand that this question want to know the maximum number of links between the root and a node that has two children.

```
   /* This will return -1 if there are not nodes in the
    * tree with two children.
    */
   public int distFullNodes() {
        return distFullNodesUtil(root, 0);
    }

    private int distFullNodesUtil(Node<E> node, int d) {
        if (node == null)
            return -1;
        int incr = (node.right != null && node.left != null) ? d : -1;
        int ll = distFullNodesUtil(node.left, d + 1);
        int lr = distFullNodesUtil(node.right, d + 1);
        if (ll > 0) {
            if (ll >= lr)
                return ll;
        }
        if (lr > 0) {
            if (lr >= ll)
                return lr;
        }
        return incr;
    }
      public static void main(String[] args) {
```

```
        LinkedBinaryTree<Integer> tr = new LinkedBinaryTree<>();
        int[] dt = { 10, 5, 2, 7, 1, 12, 50, 11, 3, 6, 8, 100, 60,
200, 160, 300 };
        for (int ii : dt)
            tr.insert(ii);
        System.out.println(tr.distFullNodes());
    }
```