Question 1. [14 MARKS]

For each of the following operations briefly describe the most efficient (in the big-O sense) algorithm. Give the worst-case runtime efficiency in big-O (using the smallest, simplest expression).

Part (a) [2 MARKS] Given a string of length n, return the letter that occurs the most times.

Algorithm:

Runtime efficiency:

Part (b) [2 MARKS] Given a binary search tree of n nodes containing integers, return the difference between the smallest and the largest values of the tree.

Algorithm:

Runtime efficiency:

Part (c) [2 MARKS] Given a binary search tree of n nodes containing integers, return the sum of all the integers in the tree.

Algorithm:

Runtime efficiency:

Part (d) [2 MARKS] Given a sorted array of n integers, return true if the array contains the number 42.

Algorithm:

Runtime efficiency:

Part (e) [2 MARKS] Given a linked list of n nodes containing integers, remove the smallest m integers.

Algorithm:

Runtime efficiency:

Part (f) [2 MARKS] Given an empty binary search tree, insert n values into the tree.

Algorithm:

Runtime efficiency:

Part (g) [2 MARKS] Traverse a binary search tree of n nodes in order.

Algorithm:

Runtime efficiency:

Question 2. [10 MARKS]

Consider the following JumpNode class:

```
class JumpNode {
  public Comparable value;
  public JumpNode next;
  public JumpNode jump;
}
```

This class is used to construct sorted singly-linked lists, called jump lists.

The instance variables value and next are as usual for nodes of a singly-linked list.

The jump instance variable refers to some node after the node, but not necessarily the next node. It can be used to iterate more quickly through the list by "jumping" over multiple nodes.

Part (a) [5 MARKS]

Complete the following method.
For full marks you must take advantage of the jumps.

```
/* Return whether the list referred to by front contains value.
   Requires: the values in the list are sorted in increasing order. */
public static boolean contains(JumpNode front, Comparable value) {









}
```

Part (b) [5 MARKS]

Complete the following method.

```
/* Insert value at an appropriate place in the list referred to by front,
    and return the front of the updated list.
   Make the new node's jump refer to the node 5 nodes after it,
    or null if there are less than 5 nodes after it.
   Requires: the values in the list are sorted in increasing order. */
public static JumpNode insert(JumpNode front) {









}
```

Question 3. [25 MARKS]

In this question you will write some code for linked lists, where the end of the list is indicated by a special node with no data and no link to a next node.

Part (a) [5 MARKS]

Write an interface Node containing exactly these two methods:

```
// Return whether this list contains e.
boolean contains(int e)

// Insert e2 immediately after the first occurrence of e1.
void addAfter(int e1, int e2)
```

Include header comments in proper Javadoc format, and clearly indicate any preconditions.

Part (b) [10 MARKS]

Write two classes Element and End implementing the interface.
Element is for nodes containing data and a link to the next node.
End is for a special node to indicate the end of a linked list.

Element contains one constructor that takes an int as the data.
End just has its default constructor.

Element and End only have the methods required by the interface.

Element may have instance variables, which must be private.
End has no instance variables.

You must use recursion to implement the methods: do not use any loops. And you may not use instanceof.

Comments are not required.

Part (c) [5 MARKS]

Write two classes for exceptions:
MissingElement, extending Exception
EmptyList, extending MissingElement

Part (d) [10 MARKS]

Write new versions of addAfter for Node, Element and End, so that if e1 is not in the list:
addAfter(e1, e2) in Element throws a MissingElement exception
addAfter(e1, e2) in End throws an EmptyList exception

Question 4. [10 MARKS]

Consider the following algorithm:

```
public static traverse(TreeNode root) {
  C container = new C();
  container.ADD(root);
  while(!container.isEmpty()) {
    TreeNode current = container.REMOVE();
    if (current.left == null && current.right == null) {
      System.out.print(current.value + " ");
    }
    else {
      TreeNode left = current.left;
      TreeNode right = current.right;
      current.left = null;
      current.right = null;
      if (right != null) {
        container.ADD(right);
      }
      container.ADD(current);
      if (left != null) {
        container.ADD(left);
      }
    }
  }
}
```

What is printed out when C is a class implementing a stack, ADD = push and REMOVE = pop?

What is the name for this traversal?

What is printed out when C is a class implementing a queue, ADD = enqueue and REMOVE = dequeue?

Question 5. [10 MARKS]

Consider the following class for binary trees:

```
class BTree {
  public left BTree;
  public right BTree;
}
```

Complete the following method (which is useful for telling if a BST is not too tall).
Hint: write a helper method that returns the height of a subtree, or -1 if the subtree is unbalanced.

```
/* Return whether: for every node in t the difference in height
    of its left and right subtrees is at most 1.
   An empty tree is represented by t == null. */
public static boolean balanced(BTree t) {
```

Question 6. [10 MARKS]

Consider the following class for nodes in a Binary Search Tree:

```
class BSTNode {
  public int key;
  public BSTNode left;
  public BSTNode right;
}
```

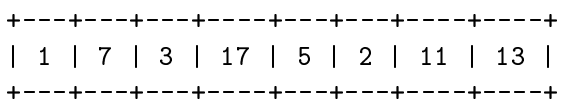
Complete the following method.

```
/* Remove the largest key from the binary search tree rooted at t.
   Return the root of the (modified) binary search tree.
   Requires: t != null. */
public static Object removeLargest(BSTNode t) {









}
```

Question 7. [10 MARKS]

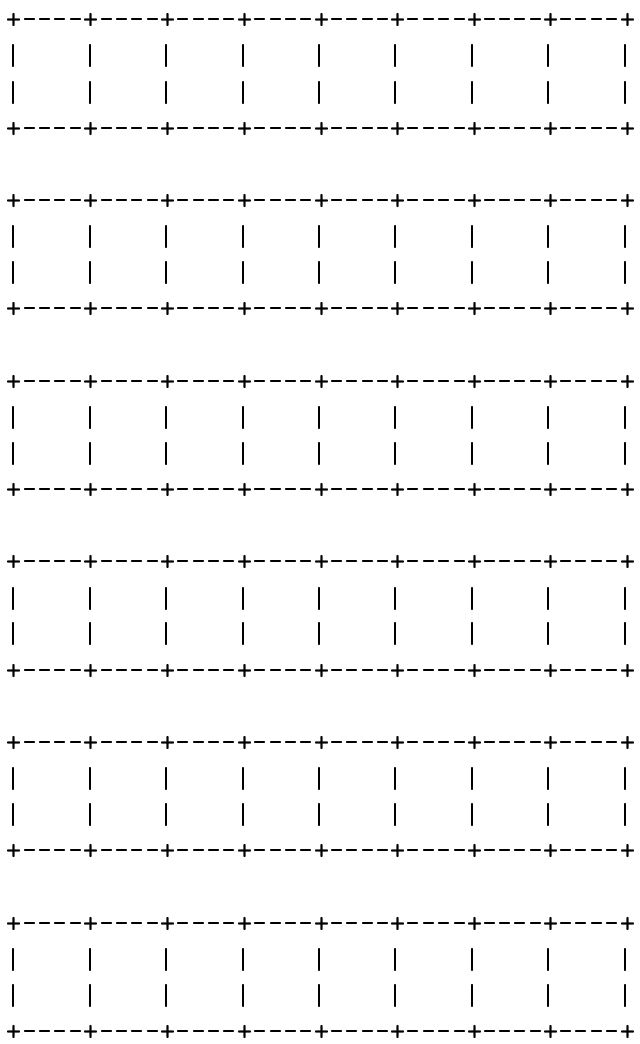
Part (a) [5 MARKS]

Consider the following array of elements:



When mergesort is called on this array, the array's contents will be sorted from smallest (on the left) to largest (on the right).

Show how the contents of this array will change as mergesort reorders the array's elements. Fill in the array diagrams below with snapshots of the array's contents each time the elements are reordered because of a call to merge. Use as many of the diagrams as you think you need.



Part (b) [5 MARKS]

Consider the alphaMerge method:

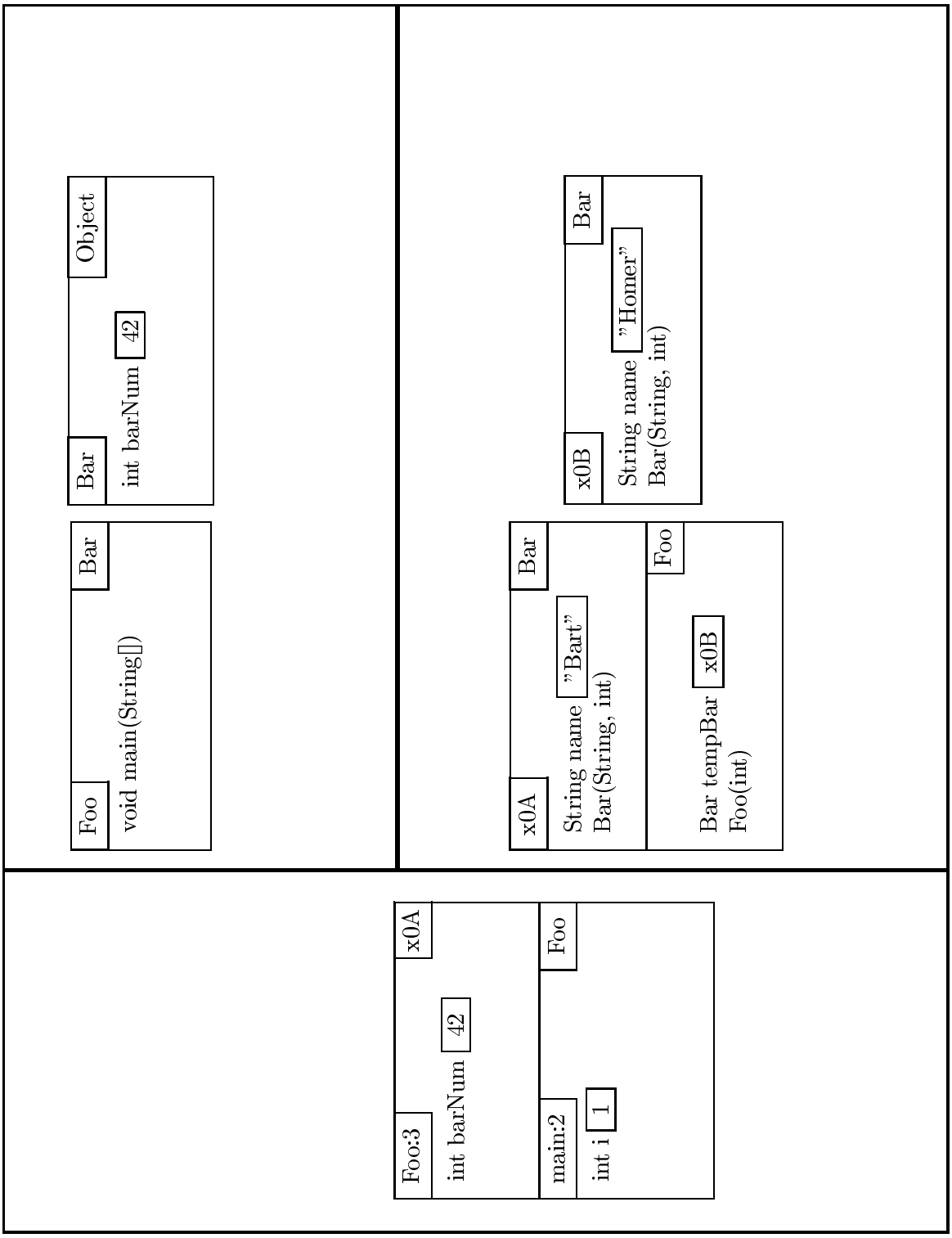
```
private static String alphaMerge(String a, String b) {

  String result = "";
  int aPos = 0, bPos = 0;
  while (aPos < a.length() && bPos < b.length()) {
    if (a.charAt(aPos) < b.charAt(bPos)) {
      result += a.charAt(aPos++);
    } else {
      result += b.charAt(bPos++);
    }
  }
  if (aPos == a.length()) {
    return result + b.substring(bPos);
  } else {
    return result + a.substring(aPos);
  }

}
```

Complete the following method with an algorithm that runs in $O(n \log n)$ time.

```
/**
 *  Return a String with the same letters as in input,
 *   but in alphabetic order.
 */
public static String alphaOrder(String input) {









}
```

Question 8. [15 MARKS]

Write a Java program which will generate the following memory model during its execution.



Total Marks = 104