Question 1. (6 points) (a) What is the *load factor* of a hash table? (Give a definition.)

The load factor is *number of items in the table / size of the table (number of buckets)*

(b) What is a reasonable value for the load factor of a hash table if the operations on the table are to be efficient (i.e., *O*(1) instead of something significantly slower)?

Generally it should be less than 1.0, usually no more than 0.5 to 0.75.

(c) What needs to be true about a hash function if operations using that function to locate items in a hash table are to be efficient?

The main requirement is if we compute hash values for many keys, they should be distributed evenly among the different buckets. Also, the hash computation itself should be efficient – *O*(1).

Question 2. (3 points) You have been asked to design a hash function for a *set* data structure (an unordered collection with no duplicates). There are two possibilities. One would be to calculate a "polynomial" using the hashcodes of the elements of the set, i.e., if the set elements are `s1`, `s2`, …, `sn`, then use (…((`s1.hashCode()` * 37 + `s2.hashCode()`) * 37 + …) * 37 + `sn.hashCode()`. The other possibility is to just add up the individual hashcodes without any additional calculations, i.e., `s1.hashCode()` + `s2.hashCode()` + … + `sn.hashCode()`. Which of these functions would be the right hashcode to use for a set, or would either one be equally good? Give a brief reason for your answer.

The sum of the individual hash codes is needed here. If we use the polynomial computation, we will get a different result if the hashcodes for the set elements are added in different orders. But a set is unordered, so we should have the same hashcode for any sets that contain the same elements, no matter what order we use the hashcodes of the individual elements to calculate the set's hashcode.

Question 3. (6 points) Suppose that we have the following class containing a few instance variables and an `equals` method.

```
public class Thing {
   private Gizmo g;
   private int number;
   private Thing nextThing;
   private Blob b;
   private Whazzit w;

   /** return true if this Thing equals other */
   public boolean equals(Object other) {
      if (!other instanceof Thing) {
         return false;
      }
      Thing t = (Thing) other;
      return this.g.equals(t.g) && this.b.equals(t.b)
                  && this.w.equals(t.w);
      }
      ...
   }
```

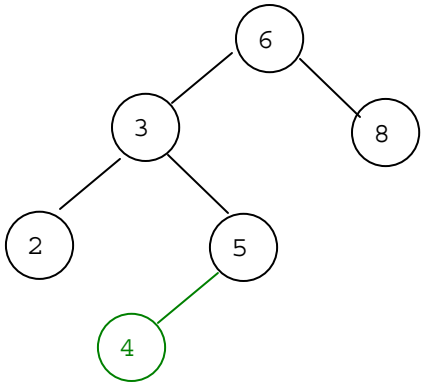
Complete the definition of a suitable `hashCode()` method for class `Thing`, below. For full credit, your `hashCode()` function should compute a high-quality hash code, but should also be consistent with the `equals()` method defined above for class `Thing`.

```
/** return a suitable hash code for Thing objects */
public int hashCode() {

   return (g.hashCode() * 37 + b.hashCode()) * 37
                                          + w.hashCode()

}
```

Key points: A polynomial computation is likely to give a better hashcode. More importantly, the hashcode computation should only involve the fields `g`, `b`, and `w`, since these are the only ones referenced in `equals()`. If other fields in a `Thing` object are used in the hashcode computation, we could wind up in a situation where two `Things` are equal, but have different hashcodes.

One other note: 37 isn't the only possible prime number to use in a calculation like this, but it is one of the ones that is known to produce reasonable results in lots of cases.

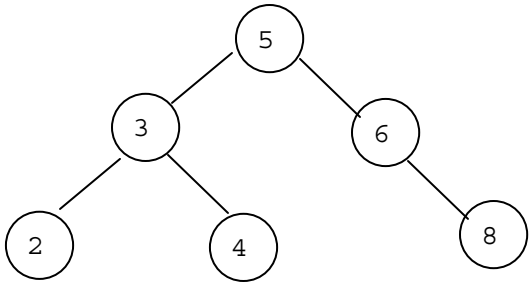
Question 4. (6 points) Consider the following AVL tree



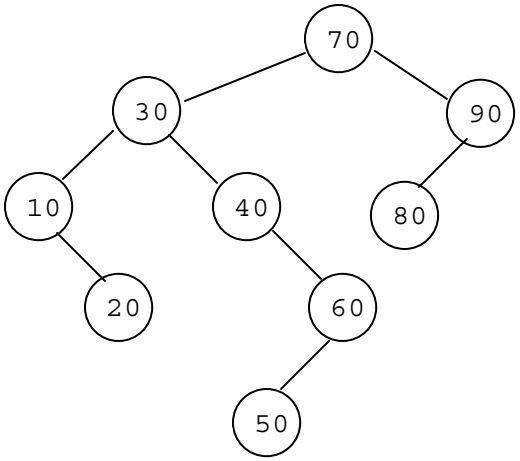
Now suppose we insert a new node containing 4 into this tree.

(a) (1 point) Show where a new node containing 4 would be inserted in the above tree before any rotations are done to rebalance the tree. (Just draw it in the correct place in the above diagram.)

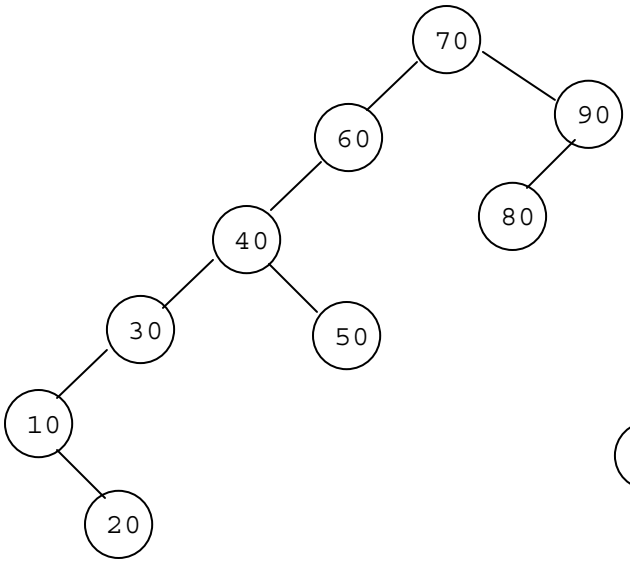
(b) (5 points) Assuming that the tree that results from adding a node containing 4 in part (a) is out of balance, show the results of the rotations that would be performed to restore the AVL property. If more than one rotation step is needed, draw diagrams that show the successive results of the rotations performed. (i.e., show your work)

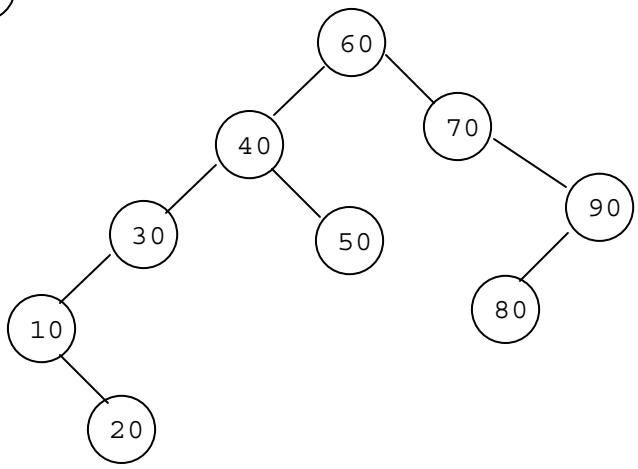


Question 5. (6 points) Consider the following splay tree.



Show the tree that results if we search for `60` in this tree and perform the splay operations that are implied by that search. If more than one splay operation is required, please draw a sequence of diagrams showing each of the splay operations (this will help you organize your answer, and will help the grader figure out what happened if something isn't quite right.)





The next questions concern binary search trees whose nodes are defined as follows.

```
class BSTNode {
   public Comparable item;     // item referred to by this node
   public BSTNode left;        // left subtree; null if none
   public BSTNode right;       // right subtree; null if none
   public BSTNode parent;      // this node's parent; null if
                               //   none (i.e., null if this is
                               //   the root node)
}
```

In this representation, each node contains a reference to its parent in the tree, if there is one, in addition to its left and right subtrees.

Question 6. (12 points) Complete the definition of the following Java method so that, given a reference to some node in a binary search tree (implemented with the above node data structure) the method returns a reference to the node that precedes it in an in-order traversal of the tree. The result should be `null` if there is no node preceding `p`.

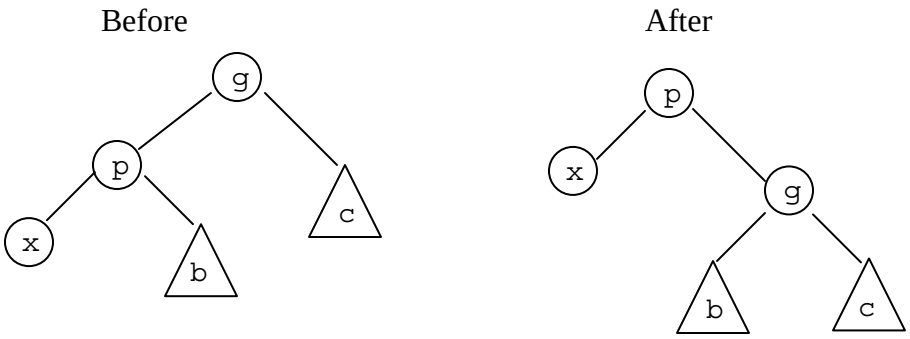
```
/* Given a non-null reference p to some node in a binary
 * search tree, return a reference to the node that PRECEEDES
 * p in an in-order traversal of the binary search tree.
 * Return null if p has no predecessor node. */
public BSTNode previousNode(BSTNode p) {
   // Strategy: if p has a left child, then the previous node
   // is the largest one on the left.  If not, then it is the
   // closest ancestor node whose item is less than p.item

   BSTNode q;
   if (p.left != null) {
      q = p.left;
      while (q.right != null) {
         q = q.right;
      }
   } else {
      q = p.parent;
      while (q != null && q.item.compareTo(p.item) > 0) {
         q = q.parent;
      }
   }
   return q;
}
```

Another way to search for the nearest ancestor that contains a smaller value is to ascend the tree as long as the current node is the left child of its parent.

```
   q = p;
   while (q.parent != null && q.parent.left == q) {
      q = q.parent;
   }
   return q.parent;
```

This is likely to be cheaper than the first solution, since it avoids a potentially expensive `compareTo` computation.

Question 7. (12 points) When a node is inserted in a red-black tree it is initially colored red. If it is inserted as the child of another red node, a rotation must be performed to restructure the tree to preserve the red-black property. One of the possible rotations is described by the following diagrams, where `x` is the new node, `p` is initially its parent, and `g` is initially its grandparent.



Assuming that variable `x` is initialized to point to the node labeled `x` in the left diagram above, write a sequence of assignment statements to perform this rotation. You do not need to write a complete method, and you do not need to worry about keeping track of the colors of the nodes. Just write a sequence of statements to change the pointers in the various nodes to restructure the nodes as shown in the diagram. You should assume that the nodes are instances of the `BTNode` class (data structure) defined on the previous page, with links `parent`, `left`, and `right` referring to adjacent nodes in the tree.

```
BSTNode x;    // initialized to point to node 'x'
...
// write your answer below

// name the ancestor nodes (optional, but helps clarify)
BSTNode p = x.parent;
BSTNode g = p.parent;

// fix child pointers
g.left = p.right;
p.right = g;

// fix parent pointers
p.parent = g.parent;
g.parent = p;
if (g.left != null) {
   g.left.parent = g;
}
```

Question 8. (9 points) We've now seen many possible ways to implement collections of data. For each of the following applications, explain which data structure from the given list you would choose and give your reasons for doing so. There may not be a single "right" answer to some of the questions. Your choices will be evaluated on whether they are appropriate for the given application and the quality of the (brief) reason(s) that you give in support of your choice.

Choose from the following list of data structures:

- Array-based list
- Linked list
- Binary tree, not a binary search tree
- Unbalanced binary search tree
- AVL tree
- Splay tree
- Hash table

(a) List of names of all UW students. Lookup operations tend to be random and are much more frequent than additions and deletions. Additions often arrive in large batches and are initially in alphabetical order. The application also needs to be able to list alphabetically all names in the directory starting at a particular name and ending at another name, e.g., list all names from Jones to Ngyuen, inclusive.

AVL tree. Access to any entry is reasonably fast ($O(\log n)$), and access to a range of entries is also quick for each successive entry. The rebalancing will ensure that the height remains $O(\log n)$, even if entries are added in alphabetical order.

(b) List of book titles sold by Amazon. There are hundreds of thousands of titles in the list, and access to any title needs to be relatively fast, but almost all accesses (greater than 99.5%) refer to a few dozen best sellers.

Splay tree. Amortized access to any entry will be $O(\log n)$, although it may be slower for some entries. However, access to the best sellers will be quite fast since the splay operations will move those to the top of the tree.

(c) Parts inventory in an automobile supply shop. There are hundreds of thousands of different parts in the inventory and, given a part number, information about the part with that number needs to be accessed quickly.

Hash table. Assuming a reasonable hash function and enough buckets, access to entries will be $O(1)$. The part number would be a suitable key for the hash function.