1. (20 points) Provide short answers to the following questions:

(a) **Question:** What does it mean for a binary search tree to be *balanced*?

**Answer:** The depth of the deepest leaf is no more than some constant factor $k$ (typically 2) of the shallowest leaf’s.

(b) **Question:** How is a *hash function* used within a hash table implementation? What makes a particular hash function good or bad?

**Answer:** A *hash function* maps each key to an array index (i.e., a hash table position). A good hash function uniformly distributes the keys among the available indices.

(c) **Question:** What is the worst-case running time (in Big-O terms) for performing a *lookup* operation on an $m$-entry hash table with chaining that contains $n$ keys? Explain.

**Answer:** $O(n)$. In the worst case, the hash function maps all of the keys to a single table position (that is, all keys collide in the table), placing all $n$ keys into a single linked list. The lookup on that linked list requires a linear search, which requires $O(n)$ operations.

(d) **Question:** What does it mean for the operations of a splay tree to take *amortized* $O(lg n)$ time?

**Answer:** Over $m$ operations on the splay tree, for a sufficiently large $m$, the total time for all of those operations is $O(m lg n)$. Thus, the average time per operation, over that sequence of operations, is $O(lg n)$. 
2. (20 points) Consider each of the following use cases of a Dictionary ADT of keys. For each, specify which data structure you would implement for that use case, and the relevant Big-O running times that motivated your choice.

(a) **Question:** The keys are arbitrary strings of characters. Insertions and removals happen regularly and with no particular pattern, and lookups are frequent.

**Answer:** Hash table (with or without chaining). Strings present an infinite key set, and there is no need for them to be ordered, making the average case $O(1)$ time for each operation a good fit.

(b) **Question:** The keys use the full range of 32-bit integers. Keys are regularly inserted in groups, and may be inserted in order; for a given key, it is frequently the case that its successor—the next largest key—needs to be found.

**Answer:** Balanced binary search tree. The need to find successors suggests the need for ordered storage of the keys, and allows that value to be found in $O(h)$ time (which $h$ is the height of the tree). Because insertions may be themselves in-order, balancing of the tree is likely necessary so that $h = O(lg n)$.

(c) **Question:** The key range is from 1 to 1000. After an initial set of insertions, there are frequent lookups. Periodic in-order printing of the keys does occur.

**Answer:** Array of booleans, with one entry per possible key value. The limited key range makes worst-case $O(1)$ on the insertions and lookups easy to achieve. The in-order printing requires $O(m)$ operations (where $m$ is the range of key values), which is worse than the $O(n)$ for in-order traversal of other structures, but the small value for $m$ limits the cost.
3. (20 points) **Question:** Consider an implementation of a *hash table with linear probing*. It uses the array `storage` to keep pointers to the keys; it also uses a boolean array, `used`, to record which locations in `storage` have *ever* contained a key.

Write the method `lookup()` for this hash table implementation, searching for a given key, and returning (as a boolean) whether it was found.

**Answer:**

```java
public boolean lookup (E key) {
    int index = hash(key);
    int original = index;
    while (storage[index] != null || used[index]) {
        if (key.equals(storage[index])) { return true; }
        index = (index + 1) % storage.length;
        if (index == original) { break; }
    }
    return false;
}
```
4. (40 points) Assume that the nodes of a red-black tree are defined like so:

```java
public class RBNode< E extends Comparable<E> > {

    public E key;
    public RBNode< E > parent;
    public RBNode< E > left;
    public RBNode< E > right;
    public boolean red;

    public RBNode< E >(RBNode< E > parent) {
        this.key = null;
        this.parent = parent;
        this.left = null;
        this.right = null;
        this.red = false;
    }

    public boolean isNullLeaf() {
        return key == null;
    }
}
```
(a) **Question:** Write a method `blackHeight()` that returns the *black height* of the subtree rooted at a given node `n`. This method should return `-1` if it finds that the black heights of the subtrees of `n` are not the same—that is, that the subtree is not part of a valid red-black tree.

**Answer:**

```java
public int blackHeight (Node n) {

    // Base case: No tree to traverse.
    if (n == null) { return 0; }

    // Base case: Null leaves are black.
    if (n.isNullLeaf()) { return 1; }

    // Find the black heights of the subtrees.
    int lh = blackHeight(n.left);
    int rh = blackHeight(n.right);

    // If either subtree is invalid, or the subtrees black heights don’t match, then this subtree is invalid.
    if (lh == -1 || rh == -1 || lh != rh) {
        return -1;
    }

    // The height of this tree is either:
    // (1) the height of either subtree, if n is red, or
    // (2) the height of either subtree + 1, if n is black.
    return (n.red ? lh : lh + 1);

}
```
(b) **Question:** Write a method `rotateLeft()` that performs a *left rotation* on a given node n.

**Answer:**

```java
public void rotateLeft (Node n) {
    // Grab the parent (p) and child (c). c will rotate
    // to take n’s place below p.
    Node p = n.parent;
    Node c = n.right;

    // c’s left subtree becomes n’s right subtree.
    n.right = c.left;
    c.left.parent = n;

    // n rotates down to become c’s left child.
    c.left = n;
    n.parent = c;

    // If p exists, have c replace n as the right or
    // left child, as appropriate.
    c.parent = p;
    if (p != null) {
        if (p.left == n) { p.left = c; }
        else { p.right = c; }
    }
}
```