1. **Initialization.** Don’t forget to do this.

2. **Memory.**

   (a) 32

   *Each Pair object uses 32 bytes of memory:*
   - 16 bytes of object overhead
   - 4 bytes for the integer **key**
   - 8 bytes for the double **value**
   - 4 bytes of padding (to make memory usage a multiple of 8)

   (b) $\sim 40n$

   *A SortedArray object uses $\sim 8m + 32n$ bytes of memory when the array **pairs[]** is of length $m$ and has $n$ non-null entries.*
   - 16 bytes of object overhead
   - 4 bytes for the integer $n$
   - 4 bytes of padding
   - 8 bytes for the reference to **pairs[]**
   - $24 + 8m$ bytes for the array of references of length $m$.
   - $32n$ bytes for the $n$ Pair objects.

   ![SortedArray Memory Diagram]

   When the array is full, $m = n$, which is $\sim 40n$.

   (c) $\sim 64n$

   *When the array is one-quarter full, $m = 4n$, which is $\sim 64n$.***
3. Five sorting algorithms.

(3.1) selection sort after 12 iterations
(3.2) mergesort just before the last call to merge()
(3.3) insertion sort after 16 iterations
(3.4) quicksort after first partitioning step
(3.5) heapsort immediately after the heap construction phase

4. Analysis of algorithms.

(4.1) $\sim \frac{1}{2}n^2$
Selection sort makes $\sim \frac{1}{2}n^2$ compares to sort any array of length $n$.

(4.2) $\sim \frac{1}{8}n^2$
Let’s consider the even and odd iterations of insertion sort separately.
- In the odd iterations (when we are inserting the integers 1, 2, …, $n$), there are no exchanges because each integer is larger than every element to its left.
- In the even iterations (when we are inserting the 0s), the 0 must be exchanged with all of the positive integers to its left.
So, the total number of exchanges is $0 + 1 + 2 + \ldots + (n/2 - 1) \sim \frac{1}{8}n^2$.
The number of compares in insertion sort is always within an additive factor of $n$ of the number of exchanges.

(4.3) $\sim \frac{3}{4}n \log_2 n$
In each merge, the left subarray contains $n/4$ 0s followed by $n/4$ smaller integers; and the right subarray contains $n/4$ 0s followed by $n/4$ larger integers. Here is an example when $n = 4$:

| 0 0 0 0 1 2 3 4 | 0 0 0 0 5 6 7 8 |

Merging two subarrays of this form involves $\sim \frac{3}{4}n$ compares because the left subarray is exhausted before taking any of the integers from the right subarray. This is true at every level in the recursion.

(4.4) $\Theta(n)$, $O(n)$, $O(n \log n)$, $O(n^2)$

$f(n) = n + \frac{1}{2}n + \frac{1}{3}n + \frac{1}{4}n + \ldots + 1 = 2n - 1$.
Big $O$ and big Theta notations discard both lower-order terms and the leading coefficient. The main difference is that big $O$ notation includes functions that grow more slowly. So, $O(n \log n)$ includes not only functions like $2n \log_2 n$ and $\frac{1}{2}n \log_2 n$, but also $2n - 1$ and $\frac{1}{8}\sqrt{n}$.
5. **Predecessor search in a BST.**

```
C D C E F D C
```

This is identical to the `floor()` function from lecture, except for when the search key is equal to the key in the node, in which case you should find the predecessor in the left subtree.

```java
private Key pred(Node x, Key k, Key champ) {
    if (x == null) return champ;
    int cmp = k.compareTo(x.key);
    if (cmp < 0) return pred(x.left, k, champ);
    else if (cmp > 0) return pred(x.right, k, x.key);
    else return pred(x.left, k, champ);
}
```

6. **Mystery key.**

   (6.1) 65, 70

   The constraints of the binary heap imply that $60 \leq x \leq 80$.

   (6.2) 0, 6, 7

   When it is time for E to be inserted, the linear-probing hash table will have the following structure:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   (6.3) (55, 65), (75, 55)

   The constraints of the k-d tree imply that $50 \leq x \leq 80$ and that $40 \leq y \leq 70$.

7. **Why did Java do that?**

   (7.1) X O X O O X

   Like mergesort, in the worst case, Timsort makes $\sim n \log_2 n$ compares and uses $\Theta(n)$ extra space. Timsort is optimized for input arrays that have a small number of runs (either in increasing or decreasing order); it makes $\Theta(n)$ compares in such cases.

   (7.2) X X X O O X

   A doubly linked list supports adding/removing from the front or back in $\Theta(1)$ time, as in a Deque. Accessing the element at index $n/2$ takes $\Theta(n)$ time. Each iterator uses $\Theta(1)$ extra memory—it only needs to maintain a reference to the current node in the iteration.
8. **Triple sum.**

The main idea is to put the integers in \( c[] \) into a hash table and then iterate over all pairs of elements \( a[i] \) and \( b[j] \) and check whether \(-(a[i] + b[j])\) is in the hash table. This solution takes \( O(n^2) \) time on typical inputs and uses \( \Theta(n) \) extra space.

*Here's the corresponding Java code.*

```java
public boolean hasTripleSum(int n, long[] a, long[] b, long[] c) {
    // add elements of c[] to hash table
    // key = integer, value = array index (but not used here)
    HashMap<Long, Integer> st = new HashMap<>();
    for (int k = 0; k < n; k++)
        st.put(c[k], k);

    // for each a[i] and b[j], check whether it sums to 0 with an integer from c[]
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (st.containsKey(-(a[i] + b[j])))
                return true;
    return false;
}
```

*Here's an alternative solution that uses sorting and a carefully orchestrated search. This solution takes \( \Theta(n^2) \) time in the worst case and uses only \( \Theta(1) \) extra memory.*

```java
public static boolean hasTripleSum(int n, long[] a, long[] b, long[] c) {
    Heap.sort(a); // assume overloaded method for sorting long[]
    Heap.sort(b);

    // for each c[k], check whether it sums to 0 with an integer from a[] and b[]
    for (int k = 0; k < n; k++) {
        int i = 0, j = n - 1;
        while (i < n && j >= 0) {
            if (a[i] + b[j] + c[k] > 0) j--;
            else if (a[i] + b[j] + c[k] < 0) i++;
            else return true;
        }
    }
    return false;
}
```

**Half-credit solution.** Create a RedBlackBST<Long, Integer> where the key is the integer in the multiset and the corresponding value is the number of times that integer appears.

```java
public class MultisetHalfCredit {
    private RedBlackBST<Long, Integer> st = new RedBlackBST<>();

    // add k to the multiset
    public void add(long k) {
        if (st.contains(k)) st.put(k, st.get(k) + 1);
        else st.put(k, 1);
    }

    // number of integers in the multiset equal to k
    public int count(String item) {
        if (st.contains(k)) return st.get(k);
        else return 0;
    }

    // number of integers in the multiset strictly less than k
    public int rank(int k) {
        int sum = 0;
        for (long i : st.keys())
            if (i < k) sum += st.get(i);
        else break;
        return sum;
    }
}
```

The `add()` and `count()` methods take $\Theta(\log n)$ time in the worst case. However, the `rank()` method takes $\Theta(n)$ time in the worst case.
Full-credit solution. The main idea to achieve a \( \Theta(\log n) \) performance guarantee for \( \text{rank}() \) is to re-implement a red–black BST, keeping the frequency counts in the nodes and modifying the subtree counts so that they account for duplicate integers.

For example, here a red–black BST corresponding to a multiset with the 10 integers 10, 20, 20, 30, 30, 30, 50, 60, 60, 80.

The specific modifications to \texttt{RedBlackBST} are as follows:

- **Value field.** Replace the \texttt{value} field with a frequency count field.
- **Add method.** Same as \texttt{put()} in a red–black BST except that if the key to be added is not in the BST, it sets the frequency in the node to 1; if the key is already in the BST, it increments the frequency in the corresponding node by 1.
- **Count method.** Same as \texttt{get()} in a BST except that it returns the frequency of the key in the BST, and 0 otherwise.
- **Rank method.** Same as \texttt{rank()} in a BST except to account for duplicate keys. Specifically, if the search key is greater than the key in the node, it should return
  \[
  \text{size}(x\text{.left}) + \text{rank}(\text{key}, x\text{.right}) + x\text{.frequency};
  \]
  instead of
  \[
  \text{size}(x\text{.left}) + \text{rank}(\text{key}, x\text{.right}) + 1;
  \]
- **Maintaining subtree counts.** Same as in \texttt{add()}, \texttt{rotateLeft()}, and \texttt{rotateRight()} except to account for duplicate keys. Specifically, when updating the subtree counts, use
  \[
  x\text{.size} = \text{size}(x\text{.left}) + \text{size}(x\text{.right}) + x\text{.frequency};
  \]
  instead of
  \[
  x\text{.size} = \text{size}(x\text{.left}) + \text{size}(x\text{.right}) + 1;\]
For reference, here’s the relevant Java code. All instance methods take $\Theta(\log n)$ time in the worst case.

```java
public class Multiset {
    private Node root;

    private class Node {
        private long key; // key
        private int frequency; // number of occurrences of key
        private Node left, right; // links to left and right subtrees
        private boolean color; // color of parent link
        private int size; // subtree count
    }

    private Node add(Node x, long key) {
        if (x == null) return new Node(key, RED);
        if (key < x.key) x.left = add(x.left, key);
        else if (key > x.key) x.right = add(x.right, key);
        else x.frequency++;
        if (isRed(x.right) && !isRed(x.left)) x = rotateLeft(x);
        if (isRed(x.left) && isRed(x.left.left)) x = rotateRight(x);
        if (isRed(x.left) && isRed(x.right)) flipColors(x);
        x.size = size(x.left) + size(x.right) + x.frequency;
        return x;
    }

    public int count(long key) {
        Node x = root;
        while (x != null) {
            if (key < x.key) x = x.left;
            else if (key > x.key) x = x.right;
            else return x.frequency;
        }
        return 0;
    }

    public int rank(long key) {
        return rank(key, root);
    }
    // number of keys less than key in the subtree rooted at x
    private int rank(long key, Node x) {
        if (x == null) return 0;
        if (key < x.key) x = x.left;
        else if (key > x.key) x = x.right;
        else return size(x.left) + rank(key, x.right) + x.frequency;
    }
}
```
Alternative full-credit solution. Modify the RedBlackBST to allow duplicate keys. The key modifications are as follows:

- **Value field.** No need for the value field.
- **Add method.** Same as `put()` in a red–black BST except that if the key to be added is equal to the key in the node, insert it in the left subtree.
- **Rank method.** Same as `rank()` in a BST except that if the search key is equal to the key in the node, return the rank in the left subtree.
- **Count method.** Not easy to do efficiently directly because the equal keys can be scattered throughout the BST. But, it’s easy to implement `count()` efficiently with two calls to `rank()`.

```java
public int count(int k) {
    return rank(k + 1) - rank(k);
}
```

*This solution uses Θ(n) space, where n is the number of integers in the multiset. The other solutions uses Θ(m) space, where m is the number of distinct integers in the multiset.*