Given an array of integers, and x, find how many pairs of elements of the array sum to x, i.e., how many indexes $i \neq j$ are there with $arr[i] + arr[j] = x$? Array is not sorted, and may contain duplicate elements. Solve the problem using balanced BST with RT = $O(n \log n)$. For example, If arr = {3,3,4,5,3,5,4} then howMany(A,8) returns 7.

Answer #1

```
static int howMany(int[] arr, int x) {
    Map<Integer, Integer> map = new TreeMap<>();
    for(int e : arr) {
            Integer c = map.get(e);
            if (c==null) map.put(e, 1);
            else map.put(e, c+1);
    }
    int count=0;
    for (Map.Entry<Integer,Integer> me : map.entrySet()) {
        int e = me.getKey(); int c = me.getValue();
        if (e*2 > x) break;
        int c2 = map.get(x-e);
        if (c2==null) continue;
        else {  if (e == x-e) { count += c*(c-1)/2;}
            else{if (2*e <= x) count += c * c2;}
        }
    }
    return count;
```

slick answer:

```
Map...
for (e : arr) { map.put(e,0); map.put(x-e,0
}
    count = 0;
for(e : arr) {
    count += map.get(x-e);
    map.put(e, map.get(e) + 1);
}
return count;
```

Given an array, return those elements that occur exactly once, in the same order in which they appear in the given array. Solve the problem using balanced BST, with RT = $O(n \log n)$.

```
        T extends Comparable<? super T>
static<T> Object[ ] exactlyOnce(T[ ] arr) {
  TreeMap<T, Integer> map = new TreeMap<>();
  int unique = 0;
  for(T e : arr) {
        Integer c = map.get(e);
        if (c==null) {map.put(e, 1); unique++;}
        else {map.put(e, c+1); if(c==1) unique--;}
  }
  // unique = number of unique elements
  Object[] result = new Object[unique];
  int i=0;
  for(T e: arr) {
        Integer c = map.get(e);
          if (c==1) {result[i++] = e; }
  }
  return result;
```

Given an array of integers, find the length of a longest streak of consecutive integers that occur in it (not necessarily contiguously). For example, if arr = {1,7,9,4,1,7,4,8,7,1}. longestStreak(arr) returns 3, corresponding to the streak {7,8,9} of consecutive integers that occur in the array. Solve using balanced BST, with RT = $O(n \log n)$.

```
static int longestStreak(int[] arr) {
    TreeSet<Integer> set = new TreeSet<>();
    for ( int e : arr) set.add(e);
    prev = set.first();
    max = 1; //just prev          current = 0;
    //LI: max = Length of max streak seen so far, current = length of streak upto current elem
    for (int e : set) {          // in sorted order for TreeSet
        if (prev+1 == e) {current++;}
        else { current = 1; }
        prev = e;
        if (current > max)  max = current;
    }
    return max;
}
```

Build height-balanced BST from sorted array.

```
treeFromArray( arr ):
   tree ← treeFromArray( arr, 0, arr.length )
   return new BST( tree, arr.length )

// Build a balanced tree from arr[ i..i+n-1 ]
treeFromArray( arr, i, n ):  // RT = O(n).
   if n <= 0 then
         return null
   else
         Ln ←  n / 2     // arr[ i..i+Ln-1 ]
         Rn ← n - Ln - 1  // arr[ i+Ln+1..i+n-1 ]
         root ←  i + Ln
         left ←  treeFromArray ( arr, i, Ln )
         right ←  treeFromArray ( arr, root+1, Rn )
         return new Tree( arr[ root ], left, right )
```

Let the running time of treeFromArray(arr, i, n) be $T(n)$. The recurrence for the running time is:
$T(0) = 1$ (base case of the algorithm)
$T(n) = T(Ln) + T(Rn) + c$, where c is a constant with the running time of the other lines of the program, other than the recursive calls.

When $n+1$ is an exact power of 2, the array splits equally, and a complete binary tree is returned. For this case, the recurrence is $T(n) = 2T(n/2) + c$. In class we used the Master method to show that the solution to this recurrence is $T(n) = O(n)$.

Hashing: subset of dictionary operations: add, contains, remove.
A function h, known as hash function, maps elements to non-negative integers in [0, n-1] where the table size is chosen to be n. Then x will be placed in table[h(x)], if possible.

Design goals:

1. Choose n proportional to number of elements in dictionary: λ = size / n, the load factor, is O(1).
2. For any two keys x and y, Pr{ h(x) = h(y) } = 1/n.
3. Pseudorandom function: h(1), h(2), h(3), $\cdots$ should be indistinguishable from a random sequence.
4. Deterministic, and easy to compute.

Implementation sketch:

add(x): Place x in table[h(x)]	**contains**(x): Is x in table[h(x)]?	**remove**(x): remove x from table[h(x)]

Collision resolution: What do you do if add(x) finds table[h(x)] is already occupied by another element?
(1) Separate chaining (known as open hashing): each entry of the hash table is a linked list of elements.
(2) Open addressing (closed hashing): each entry of the hash table can store only one element (or a small, fixed number of elements). Many schemes are available for collision resolution.

Java: Hash tables use separate chaining. Hash function is called hashCode(), and h(x) is a function of x.hashCode() and n. Table size is automatically adjusted based on load factor, and system tries to keep the load factor to be less than 0.5. In the base class of the object hierarchy, Object, hashCode is defined to be the address of the object. This is not a good hash function. Wrapper classes override it. User-defined classes that need to be used as keys in hashing should implement hashCode( ) and equals( ) methods.

The lengths of Java's hash tables are powers of 2 to simplify calculations. Bit operations are used to mangle the integer given by hashCode( ) to avoid problems created by poorly defined hash functions.

```
// Code extracted from Java's HashMap:
static int hash(int h) {
   // This function ensures that hashCodes that differ only by
   // constant multiples at each bit position have a bounded
   // number of collisions (approximately 8 at default load factor).
   h ^= (h >>> 20) ^ (h >>> 12);
   return h ^ (h >>> 7) ^ (h >>> 4);
}
static int indexFor(int h, int length) {      // length = table.length is a power of 2
   return h & (length-1);
}
// Key x is stored at table[ hash( x.hashCode( ) ) & ( table.length − 1 ) ].
```

Java hash tables: HashSet, HashMap, LinkedHashSet, ConcurrentHashMap, HashTable.
HashSet: implementation of Set interface. Main operations: add, contains, remove, iterator. add(x) is rejected if x is already in the set. HashSet is implemented using HashMap.
HashMap: Implementation of Map interface (key/value pairs). Main ops: : get, put, containsKey, remove, iterator. put operation replaces value if key already exists in map. get returns null if key does not exist.
LinkedHashSet: like HashSet, but iterator goes through elements in order of add.
ConcurrentHashMap, **HashTable**: synchronized, suitable for multi-threaded applications.