EC 504
Spring, 2022
Due Date: Saturday, Feb. 5, 8 pm
Submit on Gradescope

1. (20 pts) Here is a classical problem that is often part of interviews. You are given n nuts and n bolts, such that one and only one nut fits each bolt. Your only means of comparing these nuts and bolts is with a function TEST(x, y), where x is a nut and y is a bolt. The function returns +1 if the nut is too big, 0 if the nut fits, and -1 if the nut is too small. Note that you cannot compare nuts to nuts and bolts to bolts.

   The usual question is to design and analyze an algorithm for sorting the nuts and bolts from smallest to largest using the TEST function. While there are many approaches that work, the smart answer is to combine a pivot operation from quicksort to solve this as follows: Pick a bolt, and partition the nut array into nuts that are smaller, nut that fits, and nuts that are larger. Then, pick the matching nut, and partition the nut array similarly. Below is pseudo-code for this algorithm.

```
Procedure quicknutboltsort(nutarray,boltarray,m,n):
if m >=n, return;
new_nutarray[0:n-m] = -1; new_boltarray[0:n-m]=-1;
testbolt = boltarray[n];
low = 0; best_fit = -1; high = n-m;
For k in m to n,
     if Test(nutarray[k],testbolt) < 0,  // nut too small, move to front
          new_nutarray[low++]=nutarray[k];
     else if Test(nutarray[k],testbolt) == 0, //nut fits, remember it
          best_fit = nutarray[k];
     else  //nut is too big, move to back
          new_nutarray[high--] = nutarray[k];
new_nutarray[low] = best_fit;
// Now pivot the bolts against the best_fit nut.
low = 0; testnut = best_fit; best_fit = -1; high = n-m;
For j in m to n,
     if Test(testnut,boltarray[j]) < 0, //bolt is too big, move to back
          new_boltarray[high--]=boltarray[j];
     else if Test(testnut,boltarray[j]) > 0, // bolt is too small, move to front
          new_boltarray[low++]=boltarray[j];
     else
         continue;
new_boltarray[low] = testbolt;
// now recur:
quicknutboltsort(new_nutarray, new_boltarray, 0, low-1);
quicknutboltsort(new_nutarray,new_boltarray,low+1,n-m);
for k in m to n:
    nutarray[k] = newnut_array[k-m];
    boltarray[k] = newbolt_array[k-m];
```

   (a) What is the tightest worst-case asymptotic growth of the above function as $n \to \infty$?
   Solution: The worst case happens when nuts are sorted in increasing order, and the bolts are sorted in decreasing order. In that case, you go through the entire list at each pass, and the recursion is

$$T(n) = T(n-1) + T(0) + cn$$

   This means $T(n) = c(1 + c + 2c + 3c + \ldots + nc \in O(n^2)$.

(b) The above algorithm uses lots of extra storage, copying arrays back and forth. Modify the above algorithm so that you use only the original nutarray and boltarray.

Solution:

See below:

```
Procedure quicknutboltsort(nutarray,boltarray,m,n):
if m >=n, return;
testbolt = boltarray[n];
low = m;
//Pivot the nuts against the last bolt, moving smaller nuts to front
For k in m to n-1,  //leave the last position for the nut that fits
     if Test(nutarray[k],testbolt) < 0, // nut too small, move to front
         swap(nutarray[k],nutarray[low])
           low++;
     else if Test(nut_array[k],testbolt) == 0, //nut fits, swap with last and retest k
           swap(nutarray[k],nutarray[n])
           k--;
swap(nutarray[low], nutarray[n]); // swap last nut into correct position

// Now pivot the bolts against the nut that fit testbolt
testnut = nutarray[low];
low = m;
For j in m to n-1, //we know the bolt in last position is the one that fits
     if Test(testnut,boltarray[j]) > 0, // bolt too small, swap down
          swap(boltarray[j],boltarray[low]);
          low++;
     else if Test(testnut,boltarray[j]) == 0, // not needed, left it in for symmetry
          swap(boltarra[j], boltarray[n])
          j--;
swap(boltarray[low], boltarray[n]) // put bolt that fits in correct position
// now recur, excluding the nut-bolt position that already fits.
quicknutboltsort(nutarray, boltarray, m, low-1);
quicknutboltsort(nutarray,boltarray,low+1,n);
```

2. (30 pts) Suppose you are given two sorted arrays A, B of integer values, in increasing order, sizes n and m respectively, and $m+n$ is an odd value (so we can define the median value uniquely). Assume $n < m$.

   (a) Assume arrays are indexed from 0, and that integer division rounds down. Show that, if $A[(n-1)/2] \leq B[(m-1)/2]$, the median value of the combined arrays is in the interval $[A[(n-1)/2], B[(m-1)/2]]$. Otherwise, show that the median value is in the interval $[B[(m-1)/2], A[(n-1)/2]]$.

   Solution:

   First, if $A[(n-1)/2] = B[(m-1)/2]$, that is the median, as the value is guaranteed to be greater than or equal to $(n+m)/2$ elements, and less than or equal to $(n+m)/2$ elements.

   Suppose without loss of generality that $A[(n-1)/2] < B[(m-1)/2]$, as we won't use the fact that $m > n$. Then, consider an element $B[(m-1)/2+1]$. It is greater than or equal to $(m-1)/2+1$ elements in array B, and greater than or equal to $(n-1)/2+1$ elements in array A, so it is greater than or equal to $(m+n)/2+1$ elements.

   The median of the combined array must be in index $(m+n)/2$ if you sorted the combined arrays together, so $B[(m-1)/2+1]$ cannot be in that position, and neither can any larger elements in array B. Similarly, consider element $A[(n-1)/2-1]$. It is less than or equal to $(n-1)/2+1$ elements in A, and less than or equal to $(m-1)/2+1$ elements in array B, so it is less than or equal to $(m+n)/2+1$ elements, which means it does not have to be put in the median position $(m+n)/2+1$.

   The above argument is a little loose, using integer division, but it works out fine.

(b) Assume that, if $n \leq 2$, we can find the median in $O(\log(m))$ using binary insertion of the elements of A into B. Develop an algorithm for finding the median of the combined sorted arrays, with worst case complexity $O(\log(m))$, and write your algorithm in pseudocode as an answer.

Solution:

```
Procedure median = findMedian(A[],n, B[], m)
Input: Sorted arrays A[0:n-1], B[0:n-1]
If n > 2,
    midA = (n-1)/2; midB = (m-1)/2;
    medA = A[midA]; medB = B[midB];
    if medA == medB, // have a  median, return it
        return medA;
    If medA < medB,  //remove first midA elements of A, last midA elements of B
        median = findMedian(A[midA:n-1], n/2 + 1,  B[0:m-midA-1], m - midA);
    else //remove from consideration last midA elements of A, first midA elements of B.
        median = findMedian(A[0:midA], n/2+1, B[midA:m-1], m - midA);
else //call special function to finish
    median = binaryInsertMedian(A,n,B,m);
```

(c) Write a recursion for the run time of the algorithm as a function of m, the length of the longer array. Solve the recursion to show that your algorithm is $O(\log(m))$.

Solution:

$$T(n) = c + T(n/2), n \geq 2$$

Solution is $T(n) \in O(log(n)) + O(T(2))$. Since we assume $O(T(2)) \in O(log(m))$, then $T(n) \in O(log(m))$.

(d) Demonstrate your algorithm by finding the median of the following two cases:

$$A = [0, 1, 2, 3]; \quad B = [4, 5, 6, 7, 8, 9, 10, 11, 12]$$

$$A = [7, 8, 9, 10]; \quad B = [0, 1, 2, 3, 4, 5, 6, 11, 12]$$

Solution: Depth 1. findMedian(A[],n, B[], m)
Input: $A = [0, 1, 2, 3]; n = 4; B = [4, 5, 6, 7, 8, 9, 10, 11, 12]; m = 9$
$midA = 1; medA = 1; midB = 4; medB = 8;$
Since $medA < medB$, recur:
Depth 2. findMedian([1,2,3], 3,[4,5,6,7,8,9,10,11] ,8)
$midA = 1; medA = 2; midB = 3; medB = 7;$
Since $medA < medB$, recur:
Depth 3. findMedian([2,3], 2, [4,5,6,7,8,9,10], 7)
Since $n < 3$, do binaryInsertMedian([2,3], 2, [4,5,6,7,8,9,10], 7), which returns median = 6.

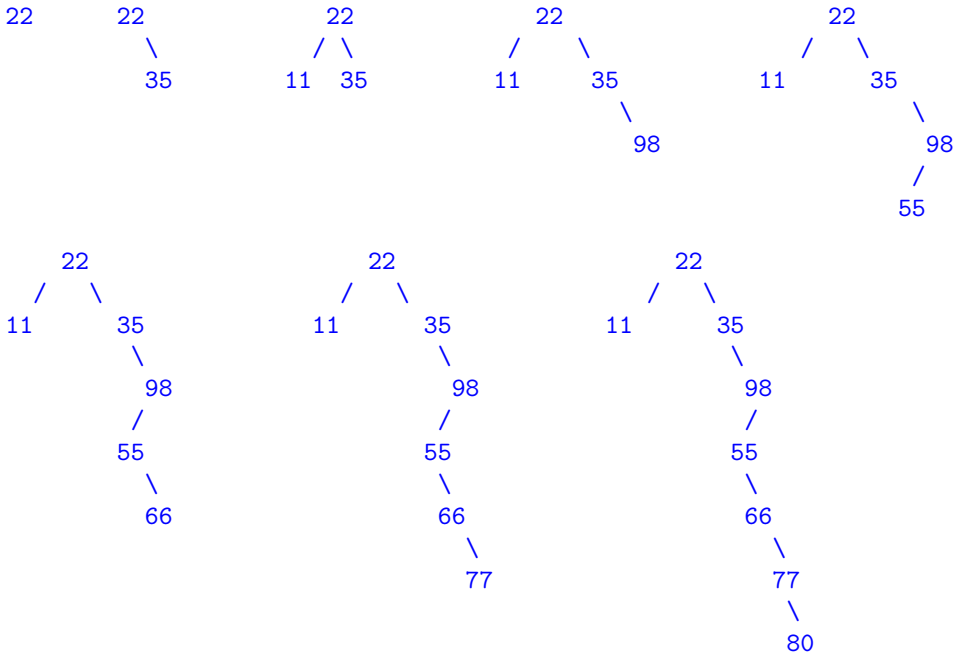
Now, for the second case:
Depth 1. findMedian(A[],n, B[], m)
Input: $A = [7, 8, 9, 10]; n = 4, B = [0, 1, 2, 3, 4, 5, 6, 11, 12]; m = 9$
$midA = 1; medA = 8; midB = 4; medB = 4;$
Since $medA > medB$ recur:
Depth 2. findMedian([7,8], 2, [2,3,4,5,6,11,12] ,7)
Since $n < 3$, do binaryInsertMedian([7,8], 2, [2,3,4,5,6,11,12], 7), which returns median = 6.

3. (20 pts) Consider the following list of elements:

$$22, 35, 11, 98, 55, 66, 77, 80$$

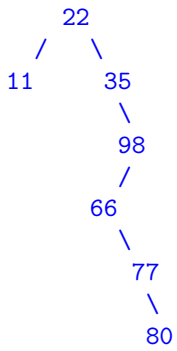
(a) Insert the following items into a binary search tree, in the order given. Show the binary trees after each insert.

Solution:



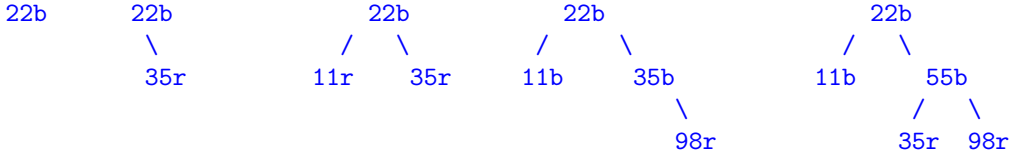
(b) From the last tree in the previous part, show the tree that remains when you delete 55.

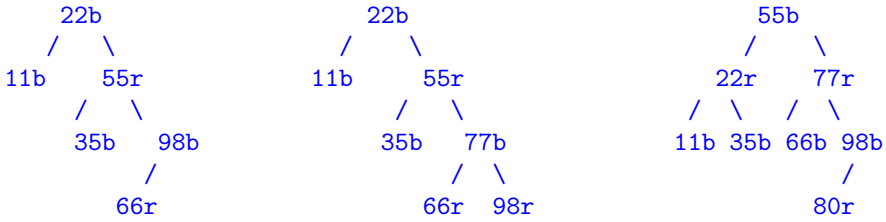
Solution:



(c) Insert the following items into a red-black tree. binary search tree, in the order given. Show the red-black trees after each insert. Put a b next to each black node, r next to each red node (or use color pens...)

Solution:





(d) From the last tree above, show the tree that remains when you delete 55.

Solution: