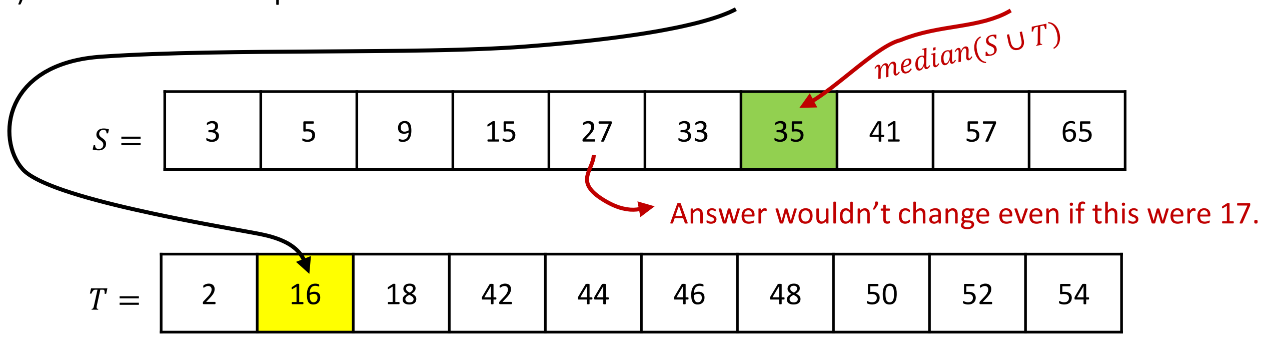
York University **EECS2011MZ: A4 Solutions** **Winter 2019**
Dept. of EECS **Instructor: Andy Mirzaian**

Problem 1: [35%] **[GTG] Exercise C-10.50, page 455 somewhat modified:**
Design and analyze an $O(\log n)$ worst-case time algorithm for the following:
Input: two sorted arrays S and T, each of size n, with a total of $2n$ distinct elements, and a positive integer $k \le n$.
Output: the k^{th} smallest element in $S \cup T$.

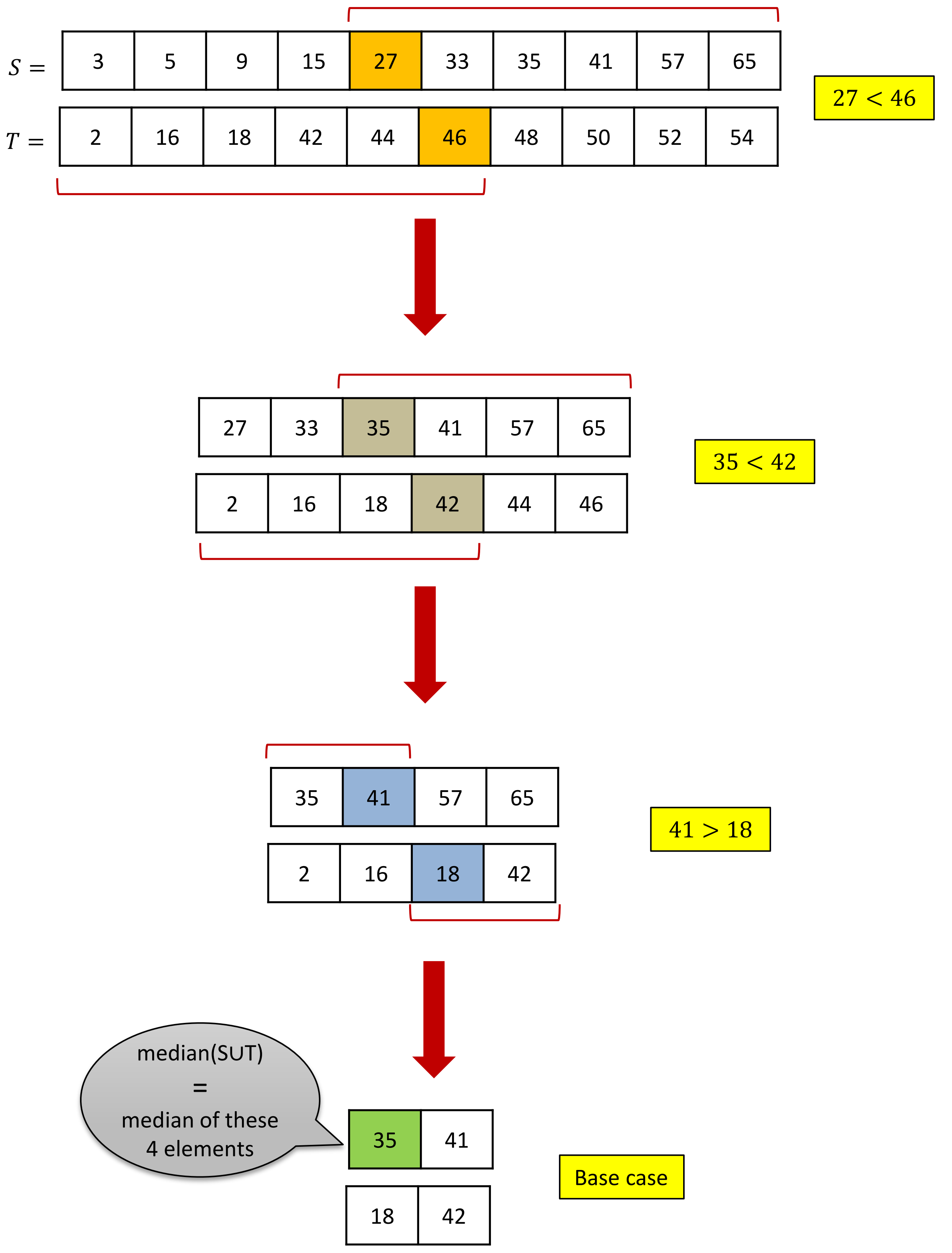
a) What is the output for the below instance with $k = 6$? What is it for $k = 10$?



b) First solve the problem for the special case $k = n$, i.e., find median of $S \cup T$.

Design Idea: We solve the problem even if the elements are non-distinct. Solution of the instance in part (a) is illustrated on page 2. The recursive algorithm is shown on page 3. Initial call is to $median(S, 0, n-1, T, 0, n-1)$. In general we have the call to $median(S, s_1, f_1, T, s_2, f_2)$ with pre and post conditions as stated. We compare the medians $S[m_1]$ and $T[m_2]$ of the two sub-arrays $S[s_1..f_1]$ and $T[s_2..f_2]$. If these medians are equal, then they are both the overall median. If not, consider the sub-array with the smaller median and eliminate its elements before its median (these are smaller than half the elements in both arrays, hence, have lower rank than the overall median), and eliminate elements of the other sub-array after its median (these have higher rank than the overall median). Because of the "floor" and "ceiling" in the computation of the two median indices, equal number of low and high rank elements are eliminated from the two sub-arrays. So, median of the remaining elements is still the original median. Now we recur on the remaining two half-size sub-arrays. The base case is when we have at most 2 elements in each sub-array. The latter is important to avoid getting into infinite recursion, since if we also recur when there are 2 elements in each sub-array, we might not eliminate any more elements (because of the "floor" and "ceiling") and hence enter an infinite recursion (recall LS4, pages 64-66).

Running Time Analysis: The recurrence is $T(n) = T(n/2) + O(1)$, with the boundary condition $T(n) = O(1)$ for $n \le 2$. By the Master Method the solution is $T(n) = O(\log n)$.



$S =$ 3 5 9 15 27 33 35 41 57 65
$T =$ 2 16 18 42 44 46 48 50 52 54
$27 < 46$
27 33 35 41 57 65
2 16 18 42 44 46
$35 < 42$
35 41 57 65
2 16 18 42
$41 > 18$
median(SUT) = median of these 4 elements
35 41
18 42
Base case

```
Algorithm  $median(S, s_1, f_1, T, s_2, f_2)$
// Pre-Cond:  $S[s_1..f_1]$ and $T[s_2..f_2]$ are non-empty, equal length, sorted arrays.
//            ( Note: non-empty equal length means $f_1 - s_1 = f_2 - s_2 \geq 0$ .)
// Post-Cond:  returns median of $S[s_1..f_1] \cup T[s_2..f_2]$ .

    // Base case: 1 or 2 elements in each sub-array:
    if ( $f_1 - s_1 \leq 1$ ) then return $max(min(S[s_1], T[f_2]), min(S[f_1], T[s_2]))$

    // Recursive case: at least 3 elements in each sub-array:
    $m_1 \longleftarrow \lfloor (s_1 + f_1)/2 \rfloor$      // lower  mid-point index of $S[s_1 ... f_1]$
    $m_2 \longleftarrow \lceil (s_2 + f_2)/2 \rceil$      // upper mid-point index of $T[s_2 ... f_2]$
    if  ( $S[m_1] = T[m_2]$ )  then return  $S[m_1]$
    if  ( $S[m_1] < T[m_2]$ )  then return  $median(S, m_1, f_1, T, s_2, m_2)$
    return  $median(S, s_1, m_1, T, m_2, f_2)$
end
```

c) Now solve the problem for the general case of k as expressed in the problem statements.

Design Idea: For any given $k \leq n$, return $median(S, 0, k-1, T, 0, k-1)$. Why? Because the k^{th} smallest element of $S[0..n-1] \cup T[0..n-1]$ cannot be larger than the k^{th} smallest in either array. So, it must be an element of $S[0..k-1] \cup T[0..k-1]$. But the latter collection contains $2k$ elements. So, its median is the correct answer.

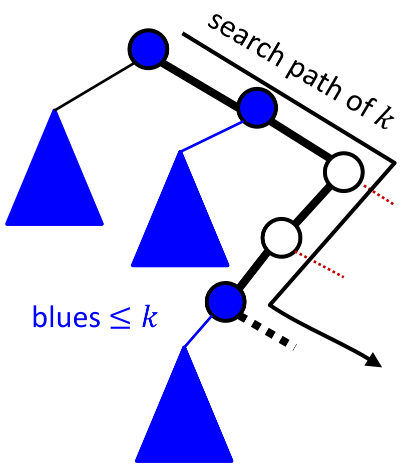
Running Time Analysis: We call the median algorithm on two subarrays each of size $k \leq n$. As in part (b), the running time is $O(\log k)$ which is at most $O(\log n)$.

Problem 2: [35%] [GTG] Exercise C-11.37, page 527.

Design Idea: Let T be an augmented BST, i.e., a BST in which each node v has an augmented field $size(v)$ that denotes the number of descendants of v (including itself). We are also given a pair (k_1 , k_2) specifying the key range $[\mathrm{k}_1, \mathrm{k}_2] = \{ k \mid k_1 \le k \le k_2 \}$. The method $countRange$(k_1 , k_2) shown below returns the number of elements in T with key in the given key range. It uses the BST method $search$ (or $treeSearch$) as well as a new method $rank(k\ ,\ v)$. The latter, also shown below, returns the number of elements in the subtree rooted at v with key $\le k$. It follows the search path of k and accumulates the sizes of the left sub-trees hanging from this path as well as the number of those nodes on the path that have a right child on the path or have key $= k$.
Now notice that $rank(k_2\ ,\ r) - rank(k_1\ ,\ r)$ is the number of elements with key $\in (k_1, k_2] = \{ k \mid k_1 < k \le k_2 \}$. To make sure key k_1 is also counted in case it is in T, we add $(\,(search(k_1,r) = null)\ ?\ 0\ :\ 1)$ to the count .

```
Algorithm  $countRange\ (\ k_1\ ,\ k_2\ )$
// Pre-Cond:    instance tree is an augmented BST, and $[k_1\ ,k_2]$ is a key range.
// Post-Cond:   returns the number of items in the BST whose key $k$ satisfies $k_1 \le k \le k_2$.
1.  $r \longleftarrow this.root(\,)$      // root of the augmented BST
2.  return $rank(\ k_2\ ,\ r) - rank(k_1\ ,\ r\ ) + \big((search(k_1\ ,\ r) = null)\ ?\ \ 0\ :\ 1\big)$
end
```

```
Algorithm  $rank(k\ ,\ v)$
// Pre-Cond:    $k$ is a key, $v$ is a node of the augmented BST.
// Post-Cond:   returns the number of items in the sub-tree
                rooted at $v$ with key $\le k$.
3.  if ( $v = null$ ) then return  0
4.  $vRank \longleftarrow 1 + size(left(v))$
5.  if $\big(k = key(v)\big)$ then return  $vRank$
6.  if $\big(k < key(v)\big)$ then return  $rank\big(k\ ,left(v)\big)$
7.  return  $vRank + rank\big(k\ ,right(v)\big)$
end
```



Maintaining the augmented field: The methods $rank$ and $search$ are access methods; they do not change the link structure of the tree, and hence do not affect the augmented field $size$ in any node. But what about the dictionary update methods $insert$ and $delete$ that change the tree structure? We should modify these update methods so that they also update the affected $size$ fields consistently. The idea is simple; $insert$ adds a new leaf node with $size$ initialized to one. This causes each of its ancestors to gain one more descendant. So, after the insertion, we use the parent pointers to climb up along the ancestral path of the inserted node and increment their size fields by one. For the $delete$ operation, the reverse is the case. The ancestors of the removed node have lost a descendant, so their $size$ fields should be decremented by one. This can also be done by climbing up along the ancestral path of the removed node to update those affected $size$ fields.

Running Time Analysis: $rank$ and $search$ take $O(h)$ time each. So, $countRange$ also takes $O(h)$ time, where h is the height of the augmented BST. The extra upward traversal along the ancestral path, in both cases of $insert$ and $delete$, also takes at most $O(h)$ time. Hence, the worst-case asymptotic running times of the dictionary operations $search$, $insert$, and $delete$ are not affected and remain $O(h)$ time.

Problem 3: [30%] [GTG] Exercise C-12.36, page 569.

Design Idea: Use an **AVL tree** T with each node storing an entry of the type $(ID, count)$, where ID is used as key, and $count$ is the vote count as the value field. Since there are $k < n$ distinct candidates, the $size$ of T is at most k. Therefore, each dictionary operation on T will take $O(\log k)$ time in the worst-case. For each of the n votes cast, we first search T for the vote ID. If the search is successful, we simply increment the count field of the located node in T. If the search is unsuccessful, then we insert the ID with its $count$ field initialized to 1. The obvious pseudo-code is omitted here and left to the reader.

Running Time Analysis: There are $O(n)$ dictionary operations each taking at most $O(\log k)$ time for a total of $O(n \log k)$ time in the worst-case.

Remark 1: Instead of **AVL tree**, we can use a **Splay tree** with the same guaranteed total time bound $O(n \log k)$. The reason is that the *amortized* time per dictionary operation is $O(\log k)$, and the aggregate amortized time is an upper bound on the aggregate actual time. As mentioned in class, we will fully analyze Splay trees in EECS4101.

Remark 2: Here are some other techniques that one might think of. **Bucketting** techniques cannot be used, since candidate IDs are arbitrarily large numbers and don't fit in a short integer interval range for bucket indexing. **Hashing** techniques are also out of the question, since their worst-case performance is bad and cannot meet the required worst-case bound. Some of you, as posted on the forum, may have found an internet site that suggests improving the worst-case performance of chained hashing by implementing the chains with AVL trees. Even though this would accomplish the stated worst-case bound, it is frankly a silly hybrid structure. What is the role of the hashing then? Why use a collection of AVL trees when a single one will do? The whole point behind hashing was its simplicity and that it offers good expected performance. The suggested hybrid is like replacing a bicycle's pedal structure with a jet engine, or a plastic surgeon doing a nose job on an elephant!

My advice: study the works of the original thinkers, and not fame seeking internet hackers. We have a lot more to study and learn in the area of algorithms and data structures in upper level courses. I hope to see you there. Best of luck with the exam, and have a great summer.