Solutions
Homework #9
Math 188, Winter 2001

7.43 Suppose we are using a radix r, and m columns (digits). Then we can represent r^m distinct keys. Since we have n distinct keys, we must have that $n \leq r^m$, or $m \geq \log_r(n)$.

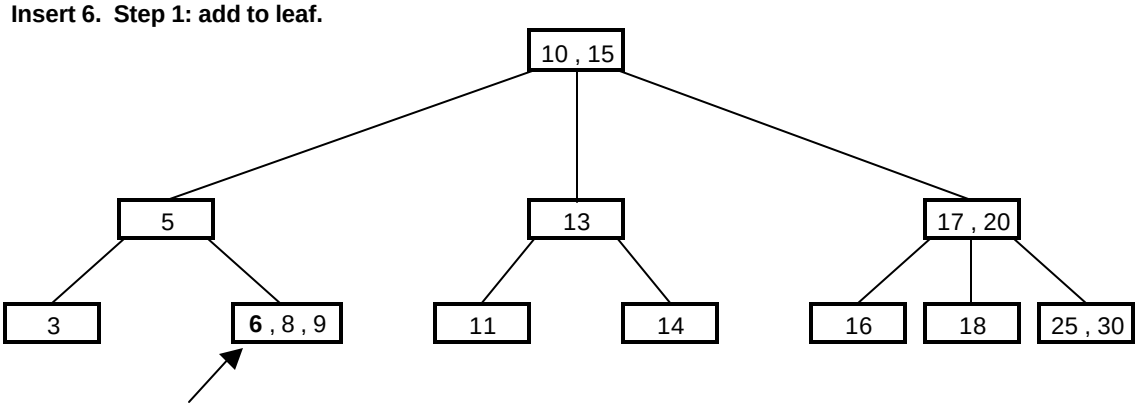
Now we follow the every-case time analysis in the book. We see that traversal of *masterlist* requires n passes through the **while** loop. Addition of all the lists to *masterlist* requires r passes through the **for** loop in *coalesce*. Each of these procedures is called m times from *radixsort*. Thus

$$\begin{aligned} T(n, m, r) &= m \cdot (n + r) \\ &\geq \log_r(n) \cdot (n + r). \end{aligned}$$

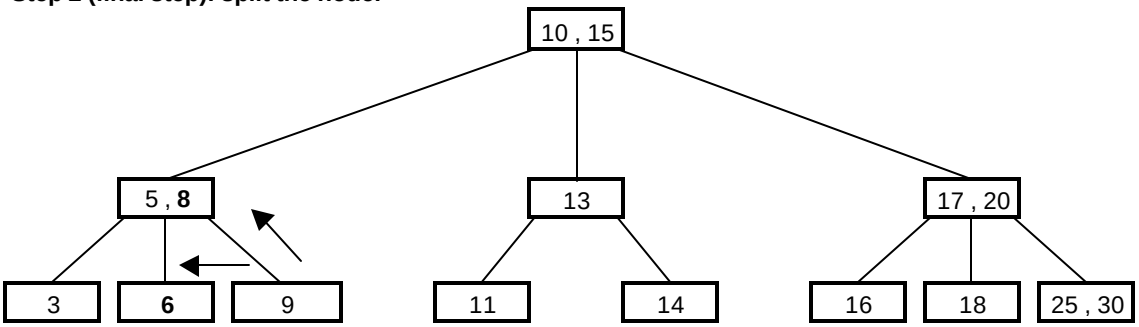
Since r is constant, $\log_r(n) \in \Theta(\lg(n))$ and therefore in terms of n alone, $T(n) \in \Omega(\lg(n) \cdot n)$ (since we have $T(n)$ greater than something.) The best case would be when $\log_r(n) = m$, in which case $T(n) \in \Theta(n \cdot \lg(n))$, as desired.

Problem H-7(a)

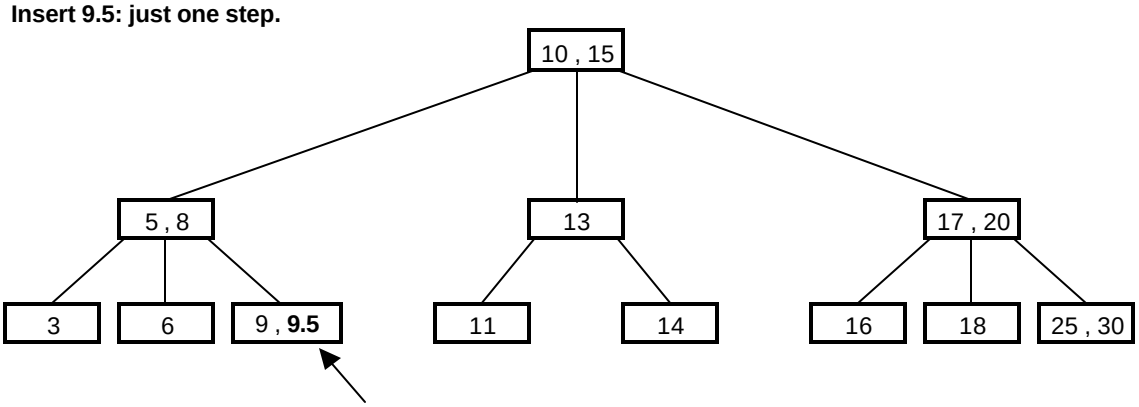
Insert 6. Step 1: add to leaf.



Step 2 (final step): split the node.

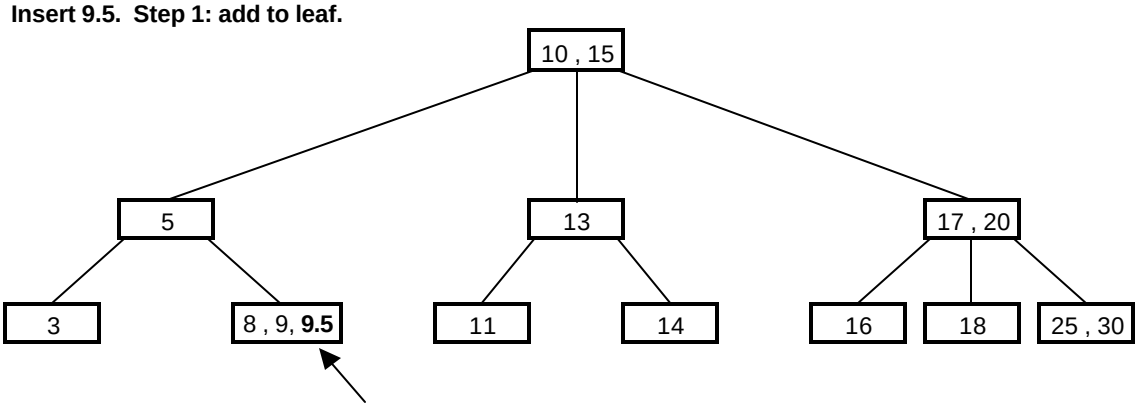


Insert 9.5: just one step.

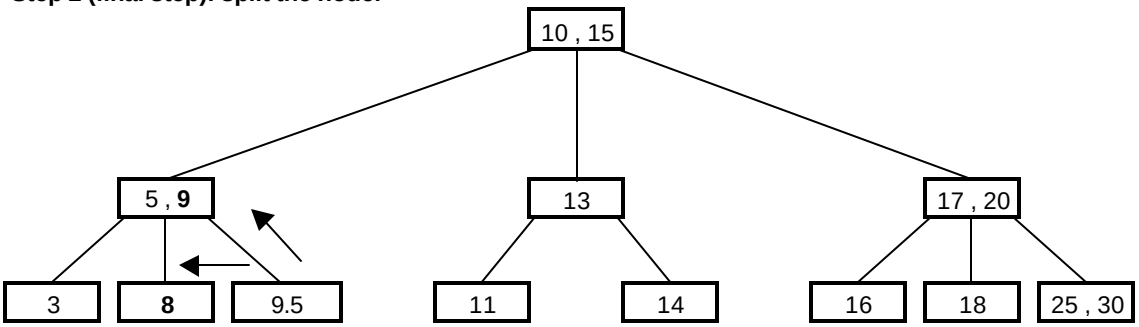


Problem H-7(b)

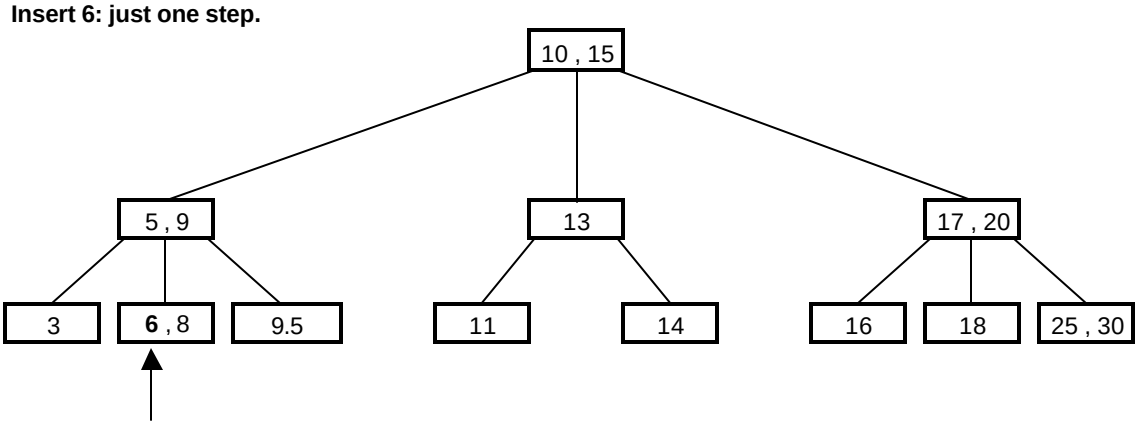
Insert 9.5. Step 1: add to leaf.



Step 2 (final step): split the node.



Insert 6: just one step.



H-8 **(a)** If the root has value x ($x = 0$ or 1), then two or more of its children have value x, two or more of each of their children have value x, and so on. In the luckiest of situations, the first two children examined for each internal node have the same value, so that we don't have to examine the third. If this happens, only two nodes are computed at depth 1, only 4 nodes are computed at depth 2, and so on, doubling for each level, down to $\boxed{2^d}$ nodes at depth d.

(b) The root's value is the majority of the values of the three children, a, b, c from left to right. The standard strategy first goes through a subset of the first 3^{d-1} leaves (examining $A(d-1)$ of them on average), giving the value a. Then it goes through a subset of the next 3^{d-1} leaves (examining $A(d-1)$ of them on average), giving the value b. Note the first and second group of 3^{d-1} leaves have not yet interacted with each other, so the number of leaves examined in the second group is not impacted by the leaves examined in the first group. Now, there is a 50% chance that $a = b$, in which case the root equals this same value (a) and we do not have to examine any more leaves, but there is also a 50% chance that $a \neq b$, in which case the root has value c, and we must examine a subset of the last 3^{d-1} leaves ($A(d-1)$ of them on average) to determine c.

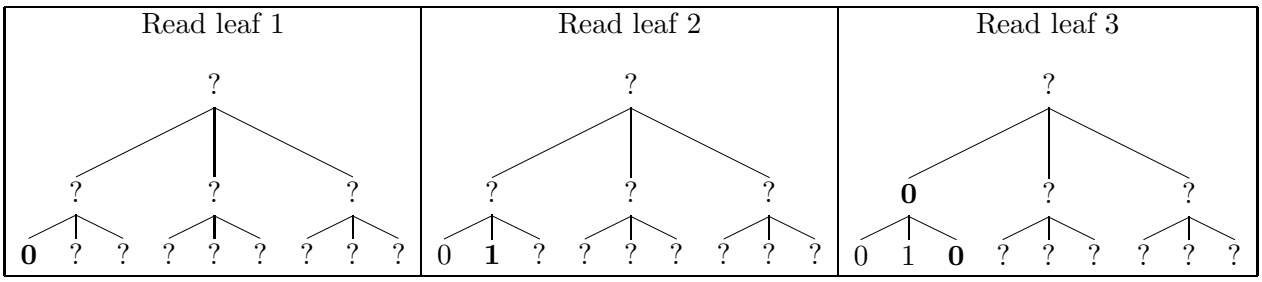
So the expected number of leaves examined is given by the recursion

$$A(d) = A(d-1) + A(d-1) + .5(0 + A(d-1)) = 2.5A(d-1).$$

The initial condition is $A(0) = 1$ because if the tree has depth 0, the root is the only leaf, and it must be examined. Solving the recursion gives $A(d) = \boxed{2.5^d}$.

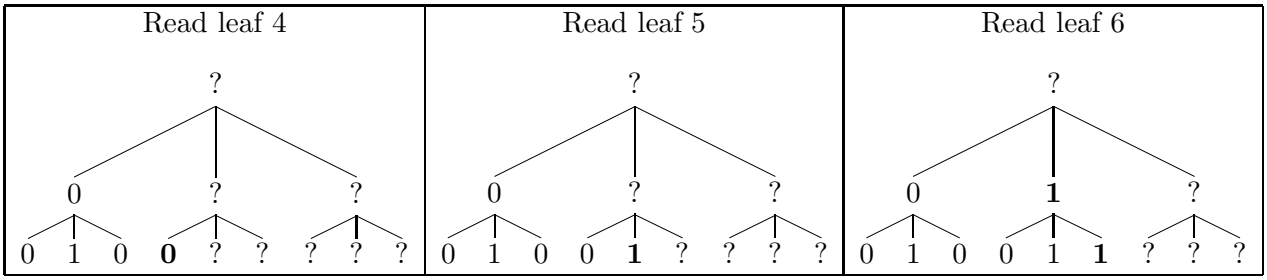
(c) Let $L(m)$ be the value of leaf m.

Example. First consider how to make a worst case for the Standard Strategy with $d = 2$ as above. Say $L(1) = 0$. If $L(2) = 0$, their parent would have value 0, so you could skip leaf 3; to prolong things, make $L(2) = 1$. Now there's one 0 and one 1, and the tie is broken by reading leaf 3. Whatever value leaf 3 has (say $L(3) = 0$) is then the majority value of the three leaves, and gets assigned to the parent of these 3 leaves.

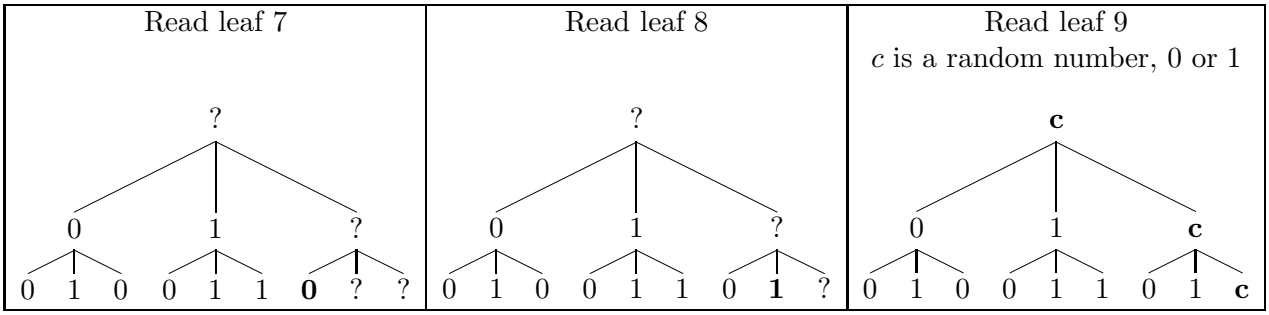


Use similar reasoning to pick $L(4) = 0$, $L(5) = 1$. This forces us to read leaf 6. Its value $L(6) = b$ will break the tie among leaves 4, 5, 6, and will be assigned to the parent of these three leaves. Then the root will have two children whose values are known, 0 and b. If $b = 0$, the root would be 0, and we wouldn't have to read the third subtree of the

root; so make $b = 1$ to force having to read the next group of 3 leaves.



For the final group of 3 leaves, we again make the first two (leaves 7 and 8) have values 0 and 1 so we are forced to read the last one, leaf 9, with value c. The parent of these three then is assigned the tie-breaking value c. Now the root has three children with values $0, 1, c$, so c is the tie-breaker, and the root gets assigned the value c. If the adversary *always* chooses $c = 0$, (or always chooses $c = 1$) we would be able to defeat the adversary's attempts to prolong the questioning by asking no questions at all; we would just use the value c always given at the end. So, the adversary randomly chooses 0 or 1:



General solution. The adversary begins by making a ternary tree of depth d in which to record all the leaves whose values been reported so far, as well as the values of any internal nodes that can be computed from these.

For any group of three siblings (including 3 leaves that are siblings, as well as 3 internal nodes that are siblings), the first node whose value is computable (based on the order of the queries) should have value 0. The second node whose value is computable should have value 1. (Or, the first one has a random value 0 or 1, and the second has the other value.) That forces the questioner to ask more questions to get the value of the third node, and the majority value of these 3 nodes will be the value assigned to this third node.

So, when a leaf is queried, if it was already queried earlier, we return the value we said before (and the questioner has just wasted a question). If it has not been queried yet, we count how many siblings have been assigned values already ($s = 0$, 1, or 2). If $s = 0$ or 1, we assign s as its value, and return it. If $s = 2$, this is a tie-breaker, so we go up the tree to see what value its parent, grandparent, or other ancestor would require to prolong activity among *its* group of siblings. Specificially, say the parent has s' siblings whose values are known.

(i) If the parent already has a value assigned to it, the questioner is wasting questions because this leaf is not necessary to compute the value of the parent. So we can assign the leaf any value. To be definite, just use the value the parent already has.

(ii) If $s' = 0$ or 1, we want the parent to have value s', so we assign this leaf s'. Now the majority of 0, 1, s' is s', so the parent is assigned the correct value.

(iii) If $s' = 2$, the parent's siblings have been assigned 0 and 1, and the parent is the tie-breaker among its siblings. We go up to the parent's parent and repeat this process recursively until we encounter an ancestor that either has a value assigned to it (and we use that value), or which is in a group of siblings with only 0 or 1 of them assigned values (and we use that number, 0 or 1). If we recurse all the way up to the root (which has no siblings), we pick a random number, 0 or 1. (If we always used 0 or always used 1, this adversary could be foiled by anyone who knew that; there would be no need to examine any leaves at all.)

Here is such an algorithm. We use coordinates (k, m), where at depth $k = 0, \ldots, d$, there are 3^k nodes, numbered $m = 1, 2, \ldots, 3^k$ from left to right. The leaves are at coordinates (d, m) where $1 \le m \le 3^d$, and the root is at coordinate $(0, 1)$. The parent of (k, m) is $(k-1, \lceil m/3 \rceil)$. The children of (k, m) are $(k+1, 3m-2)$, $(k+1, 3m-1)$, $(k+1, 3m)$. The value at node (k, m) is $f(k, m)$.

```
// queries are made with readleaf(m)
int readleaf(int m) {
  int value = adversary(d, m);
  set f(d,m) = value, and update all ancestors whose values are now computable;
  return value;
}

// f(k,number) is the value assigned to the node at the given level k and
// number.  It is 0 or 1 if it computes to that based on the values
// already assigned to the leaves underneath it, or -1 if its value is
// not yet known.

int adversary(int k, int number) {
  // value already known
  if (f(k,number) != -1) return f(k,number);

  // recursed all the way up to the root, make a random assignment
  if (k == 0) return (a random number, 0 or 1);

  s = number of siblings whose values are already known;
  if (s==0 || s==1) return s;

  return adversary(k-1, parent(number));
}
```

Note. This generalizes to r-ary trees (r odd) in which we assign values 0 or 1 to the leaves arbitrarily, and assign 0 or 1 to a internal node according to whether the majority of its children have the value 0 or 1. Let $r = 2t - 1$, so that a majority consists of $t = (r+1)/2$ or more of the same value.

The total number of ways to set the leaves is 2^{r^d}.

(a) The minimum number of queries conceivable is t^d.

(b) The average number of children that have to be examined till a majority is found is harder to compute than before. Suppose we examine all r-tuples of 0's and 1's. The bits of each one are examined from left to right, stopping when a majority is found. Let α_r be the average number of bits that must be examined (for the original problem, $\alpha_3 = 2.5$). Then the average number of leaves the standard strategy will explore is ${\alpha_r}^d$. To compute α_r, we need to know how many sequences have the t-th 0 in position k for each k (if 0 is the majority value), and the same for 1's. This number is $\binom{k-1}{t-1}$ ways to have the first $k-1$ positions filled so that $t-1$ of them are 0's, times 2^{r-k} ways to set the remaining bits. So the expected number of bits explored is

$$\begin{aligned}\alpha_r &= \frac{2\ (\text{majority bit could be 0 or 1})}{\#\text{ of sequences of length } r} \\ &\qquad \times \sum_{k=t}^{r} k \times (\#\text{ of sequences with } t\text{-th majority bit in position } k) \\ &= \frac{2}{2^r}\sum_{k=t}^{r} k\binom{k-1}{t-1}2^{r-k} \;=\; 2\sum_{k=t}^{2t-1} k\binom{k-1}{t-1}2^{-k}\end{aligned}$$

This does not simplify in terms of ordinary functions you know, it simply has to be computed for each $r = 2t-1$.

(c) Any strategy for choosing an order in which to examine the leaves can be forced to check all r^d leaves through an adversary argument. Within any group of r sibling nodes, when it is time to set a node's value and $s < r-1$ of its siblings have already been set, we set this node's value to 0 for half the values of s and to 1 for the other half of the values (examples: 0 or 1 according to whether s is even or odd; 0 if $s < (r-1)/2$, 1 if $(r-1)/2 \le s < r-1$; random choices of 0's and 1's until one more of a number would make a majority, and then just use the other number). When $s = r-1$, we recurse on the ancestors of the node until we find one that has a value assigned to it, or which has fewer than $r-1$ siblings assigned values so that it may now be assigned a value by the preceding rule. If we recurse all the way up to the root, we randomly pick a value 0 or 1.

Another way to generalize the problem is to allow more than two values. However, this does not work. Say we have r-ary trees, and assign one of three values 0, 1, 2 to each leaf. When we take the majority of a group of r values, if r is even and $r/2$ of the values are 0, $r/2$ are 1, and none are 2, there is a tie. Similarly, if r is odd and $(r-1)/2$ values are 0, $(r-1)/2$ are 1, and one value is 2, there is a tie. The only way to extend this problem to multiple values (or to even values of r) is to have a biased majority, such as taking the least value among the ones that tie for a plurality.