Massachusetts Institute of Technology — Handout 20
6.046J/18.410J: Introduction to Algorithms — March 21, 2002
Professors Michel Goemans and Piotr Indyk

Problem Set 3 Solutions

Problem 3-1. Inversible Traversibles

(a) No, it is possible to have more than one tree generate the same inorder walk. See Figure 1 for possible trees. An inorder walk of both trees generates the sequence 123, but the trees are distinct.

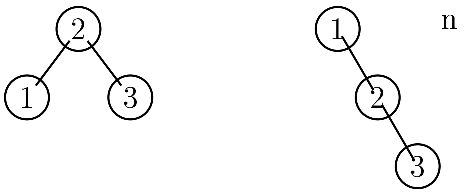

Figure 1: Two trees with the same in-order walk

(b) Yes, an preorder walk uniquely defines the tree that generated it. The following algorithm reconstructs a tree from its preorder walk. In the algorithm, we are using one variable (called *endindex*) to pass values back from the caller to the callee, as opposed to the usual passing of values from the callee to the caller. A call to TreeFromPreorderRecursive $(A, parent, startindex, endvalue)$
constructs the balanced binary subtree of *parent* whose preorder traversal is given by the following portion of the array A: starting at $A[startindex]$ (inclusive) and ending with the first element of A with value greater than *endvalue* (exclusive). The procedure returns the root of the tree and the index of the first element of A that was not included in the tree.

TreeFromPreorder (A)
$(T, dummy) \leftarrow$ TreeFromPreorderRecursive $(A, \text{NIL}, 1, \infty)$
return T

TreeFromPreorderRecursive $(A, parent, startindex, endvalue)$
if $startindex > length(A)$ or $A[startindex] > endvalue$
  return $(\text{NIL}, startindex)$
Create a new node T
parent$[T] \leftarrow parent$
key$[T] \leftarrow A[startindex]$
(left$[T]$, $leftendindex) \leftarrow$ TreeFromPreorderRecursive $(A, T, startindex + 1, \text{key}[T])$
(right$[T]$, $endindex) \leftarrow$ TreeFromPreorderRecursive $(A, T, leftendindex, endvalue)$
return $(T, endindex)$

The algorithm is correct by induction on the size of the portion of the array from which the tree is being constructed. The precise statement we want to prove by induction is given in the comments starting on the first line of TREEFROMPREORDERRECURSIVE. Base case: the portion of the array out of which we are supposed to construct the tree is empty. In that case, we return NIL and the index where we stopped is the same as the one where we were supposed to start, which is correct. Inductive case: if the portion of the array is non-empty, then the first element must be the root of the tree (because the traversal is preorder). The left subtree consists of all remaining the elements of the array until the first one greater than the root; the right subtree consists of the remainder of the original portion of the array. Both of these portions of the array are smaller than the original portion: they start at least one element later and end no later (to show that the left portion ends no later, we need to note that because key$[T] \leq$ *endvalue*, the first element larger than key$[T]$ comes no later than the first element larger than *endvalue*). Therefore, by the inductive hypothesis, the left subtree and the right subtree will be constructed correctly.

The running time of the algorithm can be analyzed as follows. For each invocation of TREEFROMPREORDERRECURSIVE (except the first one), we fill in one child-pointer in a node of the tree. Therefore, there are as many invocations as child-pointers in the tree plus 1, and the number of child-pointers is $2n$. Each invocation takes constant time, so the running time is $\Theta(n)$.

(c) The answer to this part is yes as well. In fact, postorder and preorder traversals are quite similar. We could solve this problem by giving pseudocode similar to part (b) to reconstruct the tree from the traversal. However, we will instead use the algorithm for part (b) as a "black box" to build the tree.

Notice how a postorder walk is essentially the "opposite" of a preorder walk. In a preorder walk we print our own key before recursively printing both children, and in a postorder walk we print it after recursing. So, we should be able to manipulate the array A in such a way that when we feed it into our algorithm for making a preorder tree, we can easily change it into a postorder tree. In the following pseudocode, swap(left$[T]$, right$[T]$) just swaps the left and right pointers of a node in the tree, so that the new left child is the old right child, and vice-versa.

```
FIX-TREE (T)
if T = NIL return
swap (left[T], right[T])
key[T] ← −key[T]
FIX-TREE(left[T])
FIX-TREE(right[T])
```

TreeFromPostorder (A)
for $i \leftarrow 1$ **to** n
    $B[n-i+1] \leftarrow -A[i]$
$T \leftarrow$ TreeFromPreorder(B)
Fix-Tree(T)

What this procedure does is takes the array A, reverses it, negates every element, and puts the result in an array B. It then performs a preorder traversal of B. The resulting tree T_B is "fixed" by swapping left and right pointers of each node, and negating the elements again, to obtain the final tree T. Clearly the elements that end up in T are exactly the original elements of A, since each one is negated twice in the process. We just have to prove that the postorder traversal of T is A.

Consider performing a preorder walk of T_B, where we print out B. At each node we print our own key, then our left subtree, then our right subtree. Since T is just T_B with all the left and right pointers flipped (and the elements negated), this walk is equivalent to a walk of T where we print out our own key, then the right subtree, then the left subtree; performing this walk on T results in printing $-B$. Now consider this same on T walk in reverse: we print our left subtree, then our right subtree, then our own key. This results in printing out the reverse of $-B$, which of course is just A. Since what we described is a postorder traversal of T, we conclude that the postorder traversal of T results in printing out A.

Problem 3-2. Dot-Com Office Manager

(a) For the Hire operation, we will first try to put the new employee in an open spot. If there are no open spots, then we will expand the office from $n \times n$ to $2n \times 2n$; four times its original size. The method to find an open spot just affects the running time, not the "cost," so it is not central to this problem. In the following implementation, we maintain a global variable E holding the current number of employees. We also have access to the current office dimension n. Initially, $E = n = 0$.

Hire (x)
if $E = n^2$
    Expand$(2n)$ // Cost: \$$(3n^2)$. This also performs assignment $n \leftarrow 2n$
for $i \leftarrow 1$ **to** n
    for $j \leftarrow 1$ **to** n
        if $M(i,j) = \langle$ empty $\rangle$, Assign(x,i,j) // Cost: \$1. This sets $E \leftarrow E+1$.

This algorithm runs in time $\Theta(n^2)$ in the worst case, but again, running time is not what we're concerned with. We would like to prove that the total cost of m calls to Hire is \$$(\Theta(m))$.

The intuition behind our amortized analysis is as follows. If we do not need to expand, then our running time is constant, so that is the easy case. Suppose that whenever a new employee sits at a desk, we supply the desk with some money to pay for future expands. Since we are expanding by a factor of 4, perhaps \$4 is the right amount of money to leave. If we use all the money to expand, then in our new office, the desks in the lower-left-hand corner will not have any money left; however, we have lots of time until our next expand, and by that time there will be money on all but the lower-left-hand 1/4 of the office. This should be enough to pay for an expand.

We make this intuition formal using the potential method. We must define a function $\Phi_a(M)$ that is always non-negative, such that for every operation, the real cost plus the change in potential is at most some constant. This would show that any sequence of m operations costs $\$(\Theta(m))$. Let $\Phi_a(M) = \$4(E - n^2/4)$; this is four times the number of non-empty desks NOT in the lower-left-hand-corner of the office. For this to be a useful potential function, $\Phi_a(M)$ should always be non-negative. Since the office is always at least 1/4 full, E will always be at least $n^2/4$, so $\Phi_a(M)$ will always be non-negative.

Now we just have to argue that each call to Hire has a constant amortized cost. If $E \neq n^2$ then the algorithm just inserts one employee, so the real cost is \$1. The variable E increases by 1, so $\Phi_a(M)$ increases by \$4. Thus the amortized cost is \$5 in this case. If $E = n^2$, then Expand is called, n doubles, and E increases by 1. The real cost is $(2n)^2 - n^2$ for the expand and 1 for the assign for a total of $3n^2 + 1$. The potential $\Phi_a(M)$ goes from $3n^2$ to 4, so it decreases by $3n^2 - 4$. Thus the amortized cost is $(3n^2 + 1) - (3n^2 - 4) = 5$.

(b) A bit of notation first: M' is the $n/4 \times n/4$ sub-matrix $M(1 \ldots n/4, 1 \ldots n/4)$. In other words, M' is the "safe" corner of the office, such that if we called Downsize$(n/4)$ (which we are not going to do, but soppose we do...), employees in M' do not have to be moved. During this procedure, we will maintain the invariant that M' is full of employees (and we may save a bit on our heating bill in the process...). This invariant will make our analysis much cleaner; we refer to it from now on as the *packing invariant.*

The Lay-Off operation will essentially be the opposite of our Hire operation. When Lay-Off(x) is called, we just remove the employee from the office using Remove. If the person we removed was in M', then we find another employee to move into that spot that was just vacated, in order to maintain the packing invariant. If the office is now exactly 1/16 full, we perform Downsize$(n/2)$. We can legally perform this downsize, since all the employees are in M'. We again maintain the variable E to hold the current number of employees in the office, and the dimension n. We assume that we have access to the location of x in the office. We denote this by Loc$[x]$.

```
Increment(x, y, N)
    if x < N
        return (x + 1, y)
    else
        return (1, y + 1)
```

// Finds an employee no in M' and swaps them into $M(i, j)$

FIND-SWAP(i, j)
  $i' \leftarrow 1$, $j' \leftarrow 1$
  while ($(M(i', j') = \langle \text{empty} \rangle)$ **or** ($i' \leq n/4$ **or** $j' \leq n/4$))
    INCREMENT(i', j', n)
  SWITCH(i, j, i', j')

LAY-OFF (x)
  REMOVE(Loc$[x]$) // Cost: \$1. This sets $E \leftarrow E - 1$
  if Loc$[x] \in M'$, FIND-SWAP(Loc$[x]$)
  if $E = n^2/16$
    DOWNSIZE$(n/2)$ // Cost: \$$(3n^2/4)$. This sets $n \leftarrow n/2$.

The running time of the algorithm is $\Theta(n^2)$ in the worst case, but we will focus on the amortized cost of the procedure. We start with a full office. If we lay-off someone and we do not need to downsize, then it is very cheap; we just pay \$1 for the call to REMOVE, and perhaps \$1 for the switch. Once the office is less than 1/4 full, then we are "close" to having to downsize. At that point, we can just start putting \$4 on every desk when we remove someone from it. This \$4 will be used to downsize in the same way that the \$4 was used to expand.

Formally, define $\Phi_b(M) = \max\{0, 4(n^2/4 - E)\}$. Clearly $\Phi_b(M)$ is always non-negative, since it is always at least 0. Now let's analyze our LAY-OFF operation. If $E > n^2/16 + 1$, then we just call REMOVE once, and perhaps SWITCH once, for a real cost of at most \$2. We call this a "simple" layoff. The value of E decreases by 1, so $\Phi_b(M)$ either increases by 4 (if $E < n^2/4$) or stays the same (if $E \geq n^2/4$). Thus in this case the amortized cost is at most $6 = \Theta(1)$. Our packing invariant is maintained, since if this layoff made a vacant spot in M' it was immediately filled.

Now for the harder case, where $E = n^2/16 + 1$. This is the only other possible case, since when we down-size E jumps back up to $n^2/4$. In this case the LAY-OFF operation consists of a simple layoff, followed by a call to DOWN-SIZE$(n/2)$. We argued above that the simple layoff has constant amortized cost, and maintains the packing invariant. If we argue that the down-size has constant amortized cost, and maintains our packing invariant, then the amortized cost of the overall operation is also constant.

Before the down-size, $E = n^2/16$, so $\Phi_b(M) = 4(n^2/4 - n^2/16) = 3n^2/4$. The down-size itself costs exactly $3n^2/4$, so our amortized cost $= 3n^2/4 - \Phi_b(M) = 0$. By the packing invariant, before we down-sized, all employees were in M'. After down-sizing, the new M' is a sub-matrix of the previous M', thus the new M' is full, and our packing invariant is maintained.

(c) We will use the same operations as we defined in the previous two parts. We just have to show that any sequence of m operations of either kind has amortized constant cost. When $E = n^2/4$, we are "far away" from any expensive operations, since we'd have to add $3n^2/4$ employees in order to trigger an expand, and we'd have to remove $3n^2/16$ employees in order to trigger a down-size. So, we'd like our potential to increase as we move away from $E = n^2/4$, in either direction. We define our potential function as $\Phi_c(M) = \max\{\Phi_a(M), \Phi_b(M)\} = 4|n^2/4 - E|$. This is always non-negative, since we are taking an absolute value.

The analysis of each operation is essentially the same as covered in parts (a) and (b). The amortized cost of a simple call to HIRE or LAY-OFF (no expanding or down-sizing) is at most \$2 for the real cost, and changes the potential by exactly \$4, for an amortized cost of at most \$6. A call to a simple HIRE certainly can't affect the packing invariant, since we are just adding an employee, and not changing the size of the office.

An expand operation costs $3n^2$ in real cost and changes the potential from $3n^2$ to 0, thus its amortized cost is 0. The new matrix M', although 4 times as big, is a sub-matrix of (the office before it expanded). Since we expand only when the office is full, M' is still full, thus our packing invariant is maintained.

A down-size operation costs $3n^2/4$ is real cost, and changes the potential from $3n^2/4$ to 0, thus the amortized cost is 0. As argued in part (b), the packing invariant is maintained. Since $\Phi_c(M)$ is always positive, and begins at zero, we may conclude that the cost of any sequence of m operations is at most $\$6 = \$(\Theta(m))$.

Problem 3-3. Amortized weight-balanced trees

(a) Do an inorder walk of the subtree roted at x to get the elements sorted. That will take $\Theta(size[x])$. Now build a $\frac{1}{2}$-balanced tree by making the median the new root and continuing recursively.

(b) It takes $O(\log_{1/\alpha} n)$ to search an α-balanced tree. For α constant, this is $O(\log n)$. We can show this by induction on n. If $n = 1$ the height of the tree is $0 = \log_{1/\alpha} 1$. Suppose the claim is true for $k < n$. The height of a tree of size n rooted at x is

$$1 + max(\text{ height}[left[x]], \text{height}[right[x]])$$

We know $size[left[x]] \leq \alpha n$ and $size[left[x]] \leq \alpha n$, since the tree is α-balanced, so by induction, height$[left[x]]$ and height$[right[x]]$ are both at most $\log_{1/\alpha}(\alpha n)$. So the total height is at most

$$1 + \log_{1/\alpha}(\alpha n) = \log_{1/\alpha}(1/\alpha) + \log_{1/\alpha}(\alpha n) = \log_{1/\alpha}((1/\alpha)(\alpha n)) = \log_{1/\alpha} n$$

(c) We know $\Delta(x) \geq 0$ because $\Delta(x)$ is just the absolute value of something, so $\Phi(T) \geq 0$ for all T. If T is $\frac{1}{2}$-balanced then $\Delta(x) < 2 \ \forall x$, so $\Phi(T) = 0$.

(d) Suppose we have an m-node subtree rooted at x that is not α-balanced, and say the left size is the one that violates the balanced property,i.e., $size[left] > \alpha \cdot size[x]$. This means that $size[right] < (1 - \alpha) \cdot size[x]$, and so $size[left] - size[right] > (\alpha - (1 - \alpha))m = (2\alpha - 1)m$. Since $\Phi(T) \geq c\Delta(x)$, $\Phi(T) > c(2\alpha - 1)m$.

Rebuilding an m-node subtree to be $1/2$-balanced causes the potential of every node in the subtree to drop to 0, including x. So, the change in potential is at least $c(2\alpha - 1)m$. If we make $c \geq 1/(2\alpha - 1)$, then the potential will drop by at least m. Since rebuilding the tree costs m units of potential (as the problem states), the amortized time to rebuild is $O(1)$.

(e) Inserting a node into the tree consists of searching for a place for the node, attaching it to the tree, then rebuilding all subtrees that are not α-balanced. The searching takes real time $O(\log n)$, as argued in part (b). Each "rebuilding" takes $O(1)$ amortized time, as argued in part (c).

How many "rebuilds" must we perform? It turns out we only need one. The algorithm specified in the problem says that we rebuild the subtree rooted at the highest unbalanced node. All the nodes that could potentially become unbalanced when we insert the new node are all on the path from the root to the point of insertion. So, if there are some unbalanced nodes, one such node is "higher" than all the others on this path, and this is the root of the subtree we rebuild. This rebuild makes all the other unbalanced nodes balanced, so it is the only rebuild we need to perform. Thus the total amortized time for insert is $O(\log n)$.

Problem 3-4. Error-Correcting Codes

(a) The following chart shows the output of ENCODER for each $x \in A$, where $k = 3, n = 8$. The first column is x, and the next eight columns are $h_{(0,0,0)}(x)$, $h_{(0,0,1)}(x)$, $\ldots$, $h_{(1,1,1)}(x)$.

```
-----------------------
000     0 0 0 0 0 0 0 0
001     0 1 0 1 0 1 0 1
010     0 0 1 1 0 0 1 1
011     0 1 1 0 0 1 1 0
100     0 0 0 0 1 1 1 1
101     0 1 0 1 1 0 1 0
110     0 0 1 1 1 1 0 0
111     0 1 1 0 1 0 0 1
```

Each pair of code words has Hamming distance exactly 4. If we view the above as an $n \times n$ matrix, what we get is a *Hadamard* matrix. An $n \times n$ matrix is Hadamard if every row has Hamming distance exactly $n/2$.

(b) A codeword for x has one bit $h_a(x)$ for each $h \in \mathcal{H}$. Since $\mathcal{H}$ is universal, he have that for all pairs (x, x'), where $x \neq x'$, $\Pr_{a \in A}[h_a(x) = h_a(x')] = 1/2$. This is equivalent to saying that if we pick a particular bit (corresponding to some $a \in A$) in the codewords for x and x', the probability that they differ is exactly $1/2$. This implies that the codewords for x and x' differ on exactly half their bits. Summing up, we have that for any (x, x'), where $x \neq x'$, the Hamming distance between the length-n codewords for x and x' is exactly $n/2$.

(c) We want to compute the quantity $Q = h_a(x) - h_b(x) \mod 2$. Plugging in the definition of h, we get:

$$
\begin{aligned}
Q &= \sum_{j=0}^{k-1} a_j x_j - \sum_{j=0}^{k-1} b_j x_j \mod 2 \\
&= \sum_{j=0}^{k-1} a_j x_j - b_j x_j \mod 2 \\
&= \sum_{j=0}^{k-1} x_j (a_j - b_j) \mod 2 \qquad (1)
\end{aligned}
$$

We know that a and b differ only on bit i. This means that every term in the sum in equation 1 is zero, except for the i^{th} term. So, we have that $Q = x_i(a_i - b_i)$. Since $a_i \neq b_i$, $(a_i - b_i) \mod 2 = 1$. Therefore, $Q = x_i$.

(d) We can use our fun fact from part (c) to design an algorithm often called *majority logic decoding*. The idea behind the algorithm is to derive "votes" for each information bit x_i based on pairs $[h_a(x), h_b(x)]$ in the codeword where a and b differ only on bit i. In fact, for each x_i, all the bits of the codeword can be divided into unique pairs, each one constituting a vote for x_i, for a total of $n/2$ votes. After computing all the votes, we simply take the majority value. Why does this always work? We know that $\delta(y, y') < n/4$, so fewer than $n/4$ bits were corrupted by the channel. This means that fewer than $n/4$ of our "votes" were corrupted, and therefore more than $n/4$ were correct. Since more than half the votes were correct, picking the majority value will always yield the correct value for x_i. We make the algorithm more exact below.

The function INCREMENT-WITH-SKIP(a, i) is a normal binary counter increment, but the increment operation imagines that the k-bit vector a is actually a $k-1$-bit vector by ignoring the i^{th} bit. This function enables us to iterate through all the pairs a, b that constitute independent votes.

```
DECODE (y')
for i ← 1 to k
    P ← 0, Q ← 0, a ← (0, ..., 0), V ← 0
    while (V < n/2)
        b ← a
        b_i ← 1 − b_i // Now b and a differ only on bit i
        vote = y'_BinaryValue(a) − y'_BinaryValue(b)  mod 2
        if vote = 0
            P ← P + 1
        else
            Q ← Q + 1
        a ← INCREMENT-WITH-SKIP(a, i)
        V ← V + 1
    if (P ≥ Q)
        x'_i = 0
    else
        x'_i = 1
return x'
```

The outer loop runs k times, the inner loop runs $n/2$ times. All internal operations other than INCREMENT-WITH-SKIP take constant time. So, the total time is $\Theta(nk)$ plus the time it takes for INCREMENT-WITH-SKIP. The INCREMENT-WITH-SKIP operation is executed $n/2$ times. We learned in class (and from the book) that incrementing a binary counter costs $\Theta(m)$ time for a sequence of m increments. Skipping bit i adds at most a constant amount of time for each operation, so the total time taken executing INCREMENT-WITH-SKIP is $\Theta(n/2) = \Theta(n)$. Thus the total time taken by the decoding algorithm is $\Theta(nk)$.