CPS 130 Homework 18 - Solutions

Problem 1:

(a) Is it true that in the worst case, a red-black tree insertion requires $O(1)$ rotations?

Solution: True, at most two rotations are performed.

(b) Is it true that in the worst case a red-black tree deletion requires $O(1)$ node recolorings?

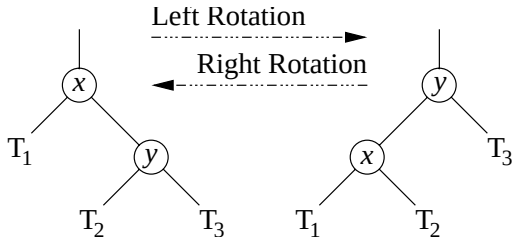
Solution: False, the number of recolorings can be at most $\Theta(\log n)$.

(c) Is it true that walking a red-black tree with n nodes in pre-order takes $\Theta(n \log n)$ time?

Solution: False, a pre-order tree walk takes $\Theta(n)$ time.

(d) Draw a left rotation and a right rotation.

Solution:



(e) What type of tree-walk on a red-black tree outputs the elements in sorted order?

Solution: In-order traversal.

(f) Given a red-black tree with n elements, how fast can you sort them using the tree?

Solution: $\Theta(n)$ using an in-order traversal.

(g) How fast can we build a red-black tree with n elements?

Solution: Each insertion into a red-black tree takes $O(\log n)$ time and we insert n elements, so we can build the tree in time $O(n \log n)$.

(h) If a data structure supports an operation FOO such that a sequence of n FOO's takes $O(n \log n)$ time in the worst case, then the amortized time of a FOO operation is $\Theta($ $)$ while the actual time of a single FOO operation could be as low as $\Theta($ $)$ and as high as $\Theta($ $)$.

Solution: The amortized time of a FOO operation is $\Theta(\log n)$ while the actual time of a single FOO operation could be as low as $\Theta(1)$ and as high as $\Theta(n \log n)$.

(i) In order for dynamic programming to be applicable to optimization problems the structure of the optimal solution must satisfy a certain condition. What is this condition and what does it mean?

Solution: A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems.

Problem 2:

In this problem we consider a data structure $\mathcal{D}$ for maintaining a set of integers under the normal INIT, INSERT, DELETE, and FIND operations, as well as a COUNT operation, defined as follows:

- INIT($\mathcal{D}$): Create an empty structure $\mathcal{D}$.
- INSERT($\mathcal{D}$,x): Insert x in $\mathcal{D}$.
- DELETE($\mathcal{D}$,x): Delete x from $\mathcal{D}$.
- FIND($\mathcal{D}$,x): Return pointer to x in $\mathcal{D}$.
- COUNT($\mathcal{D}$,x): Return number of elements larger than x in $\mathcal{D}$.

Describe how to modify a standard red-black tree in order to implement $\mathcal{D}$ such that INIT is supported in $O(1)$ time and INSERT, DELETE, FIND, and COUNT are supported in $O(\log n)$ time.

Solution: We can implement $\mathcal{D}$ using a red-black tree with an additional field *size* stored at each node. The *size* field maintains the size of the subtree rooted at x. If we set the size of leaf (NIL) nodes to zero then we may define *size* for any node x by

$$size(x) = size(right(x)) + size(left(x)) + 1,$$

and we can maintain *size* in $O(1)$ time per node.

By Theorem 14.4 in CLRS, if we augment a red-black tree of n nodes with a field f at each node x such that f can be computed using only the information stored in x, $left(x)$, and $right(x)$, then we can maintain f at all nodes during insertion and deletion in $O(\log n)$ time. In particular, we can maintain *size* so that INSERT($\mathcal{D}$,x) and DELETE($\mathcal{D}$,x) are supported in $O(\log n)$ time. INIT($\mathcal{D}$) requires creating a NIL node (null pointer) with *size* zero, which is done in $O(1)$ time. FIND($\mathcal{D}$,x) is supported in $O(\log n)$ time, as we know a tree search requires time proportional to the height of the tree, which for a red-black tree is $O(\log n)$, and that *size* is not affected at any node during the search. An implementation for COUNT($\mathcal{D}$,x) could be:

```
Count[D,x]
    if x = nil return
    c    = 0
    r    = root(D)
    while r != nil
        if key(x) < key(r) then
            c = c + size(right(r)) + 1
            r = left(r)
        else if key(x) > key(r)
            r = right(r)
        else if key(x) = key(r)
            return c + size(right(r))
        end if
    end while
    return c
```

The value c returned by COUNT will be the number of elements larger than x in $\mathcal{D}$. This implementation of COUNT correctly handles the case when x is not in $\mathcal{D}$. It suffices to explain in words without pseudocode the main idea of the COUNT implementation. We maintain a count variable c which is initially zero and start at the root of the tree. At any node r, if the key of x is less than the key of r, we add to c the size of r's right subtree plus one (to count r) and move left. If they key of x is greater than the key of r, then we move right. If the keys are equal we return c plus the size of r's right subtree. If x is not in the tree, then we will travel to a leaf, r will be NIL, and we return c. We now have to show that COUNT is supported in $O(\log n)$ time. Starting at the root, we compare the key of x with the key of r. The procedure terminates if the keys are equal or we encounter a NIL node. Else it travels to one of the children of r. At each node $O(1)$ work is performed. The nodes encountered form a path downward from the root and thus the running time of COUNT is proportional to the height of the tree, which is $O(\log n)$.

Problem 3:

A pharmacist has W pills and n empty bottles. Let $\{p_1, p_2, ..., p_n\}$ denote the number of pills that each bottle can hold.

(a) Describe a greedy algorithm, which, given W and $\{p_1, p_2, ..., p_n\}$, determines the fewest number of bottles needed to store the pills. Prove that your algorithm is correct (that is, prove that the *first* bottle chosen by your algorithm will be in some optimal solution).

Solution: Sort the n bottles in non-increasing order of capacity. Pick the first bottle, which has largest capacity, and fill it with pills. If there are still pills left, fill the next bottle with pills. Continue filling bottles until there are no more pills. We can sort the bottles in $O(n \log n)$ and it takes time $O(n)$ to fill the bottles, so our greedy algorithm has running time $O(n \log n)$.

To show correctness, we want to show there is an optimal solution that includes the first greedy choice made by our algorithm. Let k be the fewest number of bottles needed to store the pills and let S be some optimal solution. Denote the first bottle chosen by our algorithm by p'. If S contains p', then we have shown our bottle is in some optimal solution. Otherwise, suppose p' is not contained in S. All bottles in S are smaller in capacity than p' (since p' is the largest bottle) and we can remove any of the bottles in S and empty its pills into p', creating a new set of bottles $S' = S - \{p\} \cup \{p'\}$ that also contains k bottles – the same number of bottles as in S'. Thus S' is an optimal solution that includes p' and we have shown there is always an optimal solution that includes the first greedy choice.

Because we have shown there always exists an optimal solution that contains p', the problem is reduced to finding an optimal solution to the subproblem of finding $k - 1$ bottles in $\{p_1, p_2, ..., p_n\} - \{p'\}$ to hold $W - p'$ pills. The subproblem is of the same form as the original problem, so that by induction on k we can show that making the greedy choice at every step produces an optimal solution.

(b) How would you modify your algorithm if each bottle also has an associated cost c_i, and you want to minimize the total cost of the bottles used to store all the pills? Give a recursive formulation of this problem (formula is enough). You do not need to prove correctness.

Hint: Let $MinPill[i, j]$ be the minimum cost obtainable when storing j pills using bottles among 1 through i. Thinking of the 0-1 KNAPSACK PROBLEM *formulation may help.*

Solution: We want to find the minimum cost obtainable when storing j pills using bottles chosen from the set bottle 1 through bottle i. This occurs either with or without bottle i.

The first case is given by the cost c_i of storing p_i pills in bottle i plus the minimum cost to store $j - p_i$ pills among some subset of bottles 1 through $i - 1$. The second case is given by the minimum cost obtainable when storing j pills among some subset of bottles 1 through $i - 1$. The minimum of the first and second cases is the optimal solution at the i^{th} bottle.

$$MinPill[i, j] = \begin{cases} \min\{c_i, MinPill[i-1, j]\} & p_j > j, \\ \min\{c_i + MinPill[i-1, j-p_i], MinPill[i-1, j]\} & \text{otherwise.} \end{cases}$$

$MinPill[n, W]$ solves our problem.

An implementation for $MinPill[i, j]$ could be as follows:

```
MinPill[i,j]
    if p_i <= j then
        with = c_i + MinPill[i-1, j-p_i]
    else
        with = c_i
    end if
    without = MinPill[i-1,j]
    return min{with, without}
```

(c) Describe briefly how you would design an algorithm for it using dynamic programming and analyze its running time.

Solution: We want to design a dynamic programming algorithm to compute $MinPill[n, W]$. The idea is to avoid repeated calculations of subproblems by solving every subproblem $MinPill[i, j]$ just once and saving its solution in a table. We create such a table of size $n \times W$ and initialize its entries to null. We modify the function $MinPill[n, W]$ to check the table before making a recursive call to see if the value has been computed already. If so, we return the value. Else we have to make the recursive call and store the result in the table. From the recursive formulation given in (b) we see the cost to compute $MinPill[i, j]$ is $O(1)$ (we are finding the minimum of two values) not counting recursive calls and we fill each entry in the table at most once. The running time of the dynamic programming algorithm is then $O(nW)$ to create the table added to the $O(1)$ work to compute each of the nW entries, for a total running time of $O(nW)$.

An implementation could be as follows:

```
MinPill[i,j]
   if table(i,j) != null then
        return table(i,j)
    end if
    if p_i <= j then
        with = c_i + MinPill[i-1, j-p_i]
    else
        with = c_i
    end if
    without    = MinPill[i-1,j]
    table(i,j) = min{with,without}
    return table(i,j)
```

Problem 4:

In this problem we look at the amortized cost of insertion in a dynamic table. Initially the size of the table is 1. The cost of insertion is 1 if the table is not full. When an item is inserted into a full table, it first expands the table and then inserts the item in the new table. The expansion is done by allocating a table of size 3 times larger than the old one and copying all the elements of the old table into the new table.

(a) What is the cost of the i^{th} insertion?

Solution: We are given that the cost of an insertion is 1 if the table is not full. If the table is full then we have to copy the elements in the (old) table into the new one and insert the i^{th} element into the new table, so the cost of the (expensive) i^{th} insertion is i. More precisely, we know that the size of a new table is three times that of the old table and initially the table is of size 1, so after k expansions the table is of size 3^k, $k = 0, 1, 2, \ldots$ and the actual cost is $i = 3^k + 1$.

$$\text{cost of insertion } i = \begin{cases} i & \text{if } i = 3^k + 1, k = 0, 1, 2, \ldots \\ 1 & \text{otherwise} \end{cases}$$

(b) Using the accounting method, prove that the amortized cost of an insert in a sequence of n inserts starting with an empty table is $O(1)$.

Solution: In the accounting method we assign differing charges to the various operations performed. The amount we charge an operation is its amortized cost. If the amortized cost of an operation is higher than its actual cost, then the difference is referred to as credit that is distributed among elements in the system to pay for future operations.

In this problem the only operation is insertion. To calculate the amortized cost for insertion, we assume that after an expensive operation is performed – expanding the table – all the credit is used up and we need to accumulate enough credit from the insertions between table expansions to pay for next expansion. Expanding a table of size 3^k gives a new table of size $3 \cdot 3^k = 3^{k+1}$. Following this expansion, the 3^k elements we just copied have no credit to pay for future operations. There will $2 \cdot 3^k$ inserts into the table before it becomes full again and the cost of inserting element $2 \cdot 3^k + 1$ is $3^{k+1} + 1$.

Thus, to pay for the expensive insert the minimum credit c needed per each of the $2 \cdot 3^k$ inexpensive insertions is the solution to the equation

$$c\left(2 \cdot 3^k\right) + 1 = 3^{k+1} + 1,$$

which is $c = 3/2$. This is the smallest c for which the total credit in the system at any time is non-negative. The actual cost of an insertion when the table is not full is 1, so we can assign an insertion operation an amortized cost of $5/2$.

(c) Prove the same amortized cost by defining an appropriate potential function. You can use the standard notation $num(T)$ for the number of elements in the table T and $size(T)$ for the total number of slots (maximum size) of the table.

Solution: We assume that the credit in the system ('potential') is 0 after performing an expensive operation and increases with each subsequent inexpensive operation to the actual cost of the next expensive operation.

For $i = 1..n$, let c_i be the actual cost of operation i and let D_i represent the system (or data structure) that results after the i^{th} operation is performed on D_{i-1}. The amortized cost a_i of the i^{th} operation with respect to Φ is

$$a_i = c_i + \Phi(D_i) - \Phi(D_{i-1}),$$

where $\Phi(D_i)$ represents the potential in the system after operation i.

We know from (a) and (b) that if all elements inserted between table expansions have 3/2 credits then we can pay for an expensive insertion into a full table (the table is full when $num(T) = size(T)$). The table is 1/3 full after expansion, that is, $num(T) = size(T)/3$, and the potential in the system is 0 (the potential contributed by the $size(T)/3$ elements in the table is already used). In general, when we have $num(T)$ elements in the table, $num(T) - size(T)/3$ insertions have each contributed 3/2 to the potential in the system. Thus we can define our potential function as

$$\Phi(D_i) = \frac{3}{2}\left(num(T) - \frac{size(T)}{3}\right).$$

When $num(T) = size(T)/3$, $\Phi(D_i) = 0$ and when $num(T) = size(T)$, $\Phi(D_i) = size(T)$. The table is at least 1/3 filled at any time, so that $num(T) \geq size(T)/3$, and $num(T) \leq size(T)$ since when $num(T) = size(T)$ a new table is created at the next insertion, thus $\Phi(D_i) \geq 0$ for all i.

To prove the same amortized cost as in (b), we consider the two possible cases when performing an insert operation. If the table is not full, we have

$$\begin{aligned} a_{i+1} &= c_{i+1} + \Phi(D_{i+1}) - \Phi(D_i) \\ &= 1 + 3/2\,(num(T) + 1 - size(T)/3) - 3/2\,(num(T) - size(T)/3) \\ &= 5/2. \end{aligned}$$

If the table is full (i.e. $num(T) = size(T)$),

$$\begin{aligned} a_{i+1} &= c_{i+1} + \Phi(D_{i+1}) - \Phi(D_i) \\ &= num(T) + 1 + 3/2\,(num(T) + 1 - size(T)) - 3/2\,(num(T) - size(T)/3) \\ &= 5/2 + num(T) - size(T) \\ &= 5/2. \end{aligned}$$