Illinois Institute of Technology
Department of Computer Science

Homework 4 Solutions

CS 430 Introduction to Algorithms
Fall Semester, 2010

1. **Problem 13.2 on page 332–333**.

   Solution:

   (a) If we augment the tree so that each node has a black height attribute bh, then we can use Theorem 14.1 (page 346). The black height of a non-leaf node x can be determine by

   $$x.bh = \begin{cases} 1 & \text{if one or both of } x\text{'s children is a NIL leaf,} \\ x.left.bh & \text{if } x.left \text{ is red,} \\ x.left.bh + 1 & \text{if } x.left \text{ is black.} \end{cases}$$

   Since the black height of a node x only depends on the color and black height of its children, by Theorem 14.1, we can maintain the black height of each node in the tree during insertions and deletions without affecting the $O(\lg n)$ runtime of these operations.

   (b) Since the largest child of a node is always the right child in a BST, we start at the root of T_1 and travel down the rightmost path (always selecting the right child), until we reach a black node y with the same black height as T_2. In the worst case we may need to travel down to the bottom rightmost node in T_1 which will take $O(\lg n)$ time since the a red-black tree is guaranteed to have height of at most $O(\lg n)$.

   (c) Since $y.key \leq x.key \leq T_2.root.key$ and the black heights of y and T_2 are the same, we can join T_y and T_2 by making y a left child of x and $T_2.root$ the right child of x. Since $x.key \geq y.key$ we can replace y in T_1 with x by making x the right child of y's parent. Since we are only modifying attributes of x, y, $y.p$, and $T_2.root$, replacing T_y with $T_y \cup \{x\} \cup T_2$ only takes $O(1)$ time.

   (d) Since y and $T_2.root$ are both black, if we make x red it will not affect the black heights of y and $T_2.root$ and we will satisfy red-black properties 1, 3, and 5. However it is possible that properties 2 and 4 could be violated as x could now be the root (violating property 2) or x could now have a red parent (violating property 4). If x is the root, which could happen if the black height of T_1 is equal to the black height of T_2, we can simply color x black. If x has a red parent, we can call RB-INSERT-FIXUP at x which will restore the red-black properties from x to the root in $O(\lg n)$ time.

   (e) If $T_1.bh \leq T_2.bh$ then we simply find the black node y in T_2 with the *smallest* key among those whose black height is $T_1.bh$. This is done in a similar manner to that described in part (b) only now we always follow left child until we reach a node with black height $T_1.bh$. This node is now y and instead of making it a left child of x as in part (c), we make it the right child of x. The rest of the algorithm is the same.

   (f) RB-JOIN consists of three steps, which are described in parts (b), (c), and (d). Part (b) takes $O(\lg n)$, part (c) takes $O(1)$, and part (d) takes $O(\lg n)$. Hence, the total running time for RB-JOIN is $O(\lg n) + O(1) + O(\lg n) = O(\lg n)$.

(g) The algorithm RB-SPLIT takes a red-black tree T and a value x and returns two trees T_1 and T_2 such that every key in T_1 is less than or equal to x and every key in T_2 is greater than or equal to x. RB-SPLIT starts at the root of T and travels down the tree, removing subtree rooted at nodes with keys less than x or greater than $> x$ and joining them to T_1 or T_2.

As k moves along a path from the root to a leaf, the loop at Lines 5-22 executes $O(\lg n)$ times in the worst case. Hence, this algorithm also performs $O(\lg n)$ joins in the worst case. However, if as k descends down T, we simultaneously travel down the rightmost path of T_1 and leftmost path of T_2 and locate nodes with the same black height as k, joins can be carried out in constant amortized time. Thus, the total running time for RB-SPLIT is $O(\lg n)$.

Algorithm 1 RB-SPLIT(T, x)

```
1: let T_1 and T_2 be empty trees
2: v_1 = NIL
3: v_2 = NIL
4: k = T.root
5: while k ≠ NIL do
6:   if x < k.key then
7:     Join the subtree rooted at k.right to T_2 using node v_2
8:     v_2 = k
9:     k = k.left
10:  else
11:    if x > k.key then
12:      Join the subtree rooted at k.left to T_1 using node v_1
13:      v_1 = k
14:      k = k.right
15:    else {x == k.key}
16:      Join the subtree rooted at k.left to T_1 using node v_1
17:      Join the subtree rooted at k.right to T_2 using node v_2
18:      v_1 = v_2 = NIL
19:      Finished splitting, terminate while loop
20:    end if
21:  end if
22: end while
23: if v_1 ≠NIL then
24:   insert it as the maximum node of T_1
25: end if
26: if v_2 ≠NIL then
27:   insert it as the minimum node of T_2
28: end if
29: return T_1, T_2
```

2. **Problem 15-2 on page 405**.

Solution: The solution to the longest palindrome subsequence (LPS) problem is similar to the solution to the longest common subsequence in Section 15.4. First let us characterize the optimal substructure of an LPS. For a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$, we denote the subsequence starting at x_i and ending at x_j by $X_{ij} = \langle x_i, x_{i+1}, \ldots, x_j \rangle$. Now let $X = \langle x_1, x_2, \ldots, x_n \rangle$ be an input sequence, and let $Z = \langle z_1, z_2, \ldots, z_m \rangle$ be any LPS of X.

(a) If $n = 1$, then $m = 1$ and $z_1 = x_1$

(b) If $n = 2$ and $x_1 = x_2$, then $m = 2$ and $z_1 = z_2 = x_1 = x_2$

(c) If $n = 2$ and $x_1 \neq x_2$, then $m = 1$ and either $z_1 = x_1$ or $z_1 = x_2$

(d) If $n > 2$ and $x_1 = x_n$, then $m > 2$, $z_1 = z_m = x_1 = x_n$, and $Z_{2,m-1}$ is an LPS of $X_{2,n-1}$

(e) If $n > 2$ and $x_1 \neq x_n$, then $z_1 \neq x_1$ implies that $Z_{1,m}$ is an LPS of $X_{2,n}$

(f) If $n > 2$ and $x_1 \neq x_n$, then $z_m \neq x_n$ implies that $Z_{1,m}$ is an LPS of $X_{1,n-1}$

This tells us that we should examine either one or two subproblems when finding an LPS of X, depending on whether $x_1 = x_n$. Using our understanding of the optimal substructure of this problem, we define $p[i, j]$ to be the *length* of an LPS of the subsequence of X_{ij}.

$$p[i,j] = \begin{cases} 1 & \textbf{if } i = j, \\ 2 & \textbf{if } j = i+1 \textbf{ and } x_i = x_j, \\ 1 & \textbf{if } j = i+1 \textbf{ and } x_i \neq x_j, \\ p[i+1, j-1] + 2 & \textbf{if } j > i+1 \textbf{ and } x_i = x_j, \\ \max(p[i, j-1], p[i+1, j]) & \textbf{if } j > i+1 \textbf{ and } x_i \neq x_j. \end{cases}$$

Just like LCS-Length (page 394), our algorithm for finding a LPS uses two tables $p[i, j]$ and $b[i, j]$. We use to $p[i, j]$ to find an LPS and $b[i, j]$ to construct the LPS when we are done. Similar to the table b in LCS-Length, $b[i, j]$ points to the table entry corresponding optimal subproblem solution chosen when computing $p[i, j]$. To fill in the cells in tables p and b, we use the algorithm LONGEST-PALINDROME

We construct an LPS from table b in a manner similar to PRINT-LCS (page 395). We start with $b[1, n]$ and trace through the table by following the arrows. Whenever we encounter a "$\swarrow$" in entry $b[i, j]$, it implies that $x_i = x_j$ are the first and last elements of the LPS found by our algorithm.

Algorithm 2 LONGEST-PALINDROME(X)

```
1: n = X.length
2: let b[1...n, 1...n] and p[0...n, 0...n] be new tables
3: for i = 1 to n − 1 do
4:    p[i, i] = 1
5:    j = i + 1
6:    if x_i == x_j then
7:       p[i, j] = 2
8:       b[i, j] = " ↙ "
9:    else
10:      p[i, j] = 1
11:      b[i, j] = " ↓ "
12:   end if
13: end for
14: p[n, n] = 1
15: for i = n − 2 downto 1 do
16:   for j = i + 2 to n do
17:      if x_i == x_j then
18:         p[i, j] = p[i + 1, j − 1] + 2
19:         b[i, j] = " ↙ "
20:      else
21:         if p[i + 1, j] ≥ p[i, j − 1] then
22:            p[i, j] = p[i + 1, j]
23:            b[i, j] = " ↓ "
24:         else
25:            p[i, j] = p[i, j − 1]
26:            b[i, j] = " ← "
27:         end if
28:      end if
29:   end for
30: end for
31: return p, b
```

3. **Problem 16.1-5 on page 422**.

   Solution: A greedy approach can not be used for this problem. However, we can construct a dynamic programming solution. We will use a similar analysis to the one used in Section 16.1. Let S_{ij} be defined as in Section 16.1. An optimal solution to S_{ij} is a subset of mutually compatible events that together have maximum value. Let A_{ij} be an optimal slution to S_{ij}. Suppose A_{ij} includes an event a_k. Let A_{ik} and A_{kj} be defined as in Section 16.1. Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the value of the maximum value set is equal to the value of A_{ik} plus the value of A_{kj} plus v_k.

   We will denote the value of an optimal solution for the set S_ij by $val[i, j]$.

$$val[i,j] = \begin{cases} 0 & \textbf{if } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{val[i,k] + val[k,j] + v_k\} & \textbf{if } S_{ij} \neq \emptyset. \end{cases}$$

   We create two fictitious activities, a_0 with $f_0 = 0$ and a_{n+1} with $s_{n+1} = \infty$. We are interested in a maximum size set $A_{0,n+1}$ of mutually compatible activities in $S_{0,n+1}$. In addition to val we will also

use the table act, were $act[i, j]$ is the activity k that we choose to put into A_{ij}.

We use the algorithm MAX-VALUE-ACTIVITY-SELECTOR to fill in the values in tables val and act.

Algorithm 3 MAX-VALUE-ACTIVITY-SELECTOR(s, f, v, n)

```
1: let val[0...n+1, 0...n+1] and act[0...n+1, 0...n+1] be new tables
2: for i = 1 to n do
3:    val[i, i] = 0
4:    val[i, i+1] = 0
5: end for
6: val[n+1, n+1] = 0
7: for l = 2 to n+1 do
8:    for i = 0 to n-l+1 do
9:       j = i + l
10:      val[i, j] = 0
11:      k = j - 1
12:      while f[i] < f[k] do
13:         if f[i] ≤ s[k] and f[k] ≤ s[j] and val[i, k] + val[k, j] + v_k > val[i, j] then
14:            val[i, j] = val[i, k] + val[k, j] + v_k
15:            act[i, j] = k
16:         end if
17:         k = k - 1
18:      end while
19:   end for
20: end for
21: return val, act
```

The value of a maximum value set will be found in $val[0, n+1]$. The activities in this set can be found using the algorithm PRINT-ACTIVITIES with the function call PRINT-ACTIVITIES$(val, act, 0, n+1)$.

Algorithm 4 PRINT-ACTIVITIES(val, act, i, j)

```
1: if val[i, j] > 0 then
2:    k = act[i, j]
3:    print k
4:    PRINT-ACTIVITIES(val, act, i, k)
5:    PRINT-ACTIVITIES(val, act, k, j)
6: end if
```

4. **Problem 17.4-3 on page 471**

Solution: For this problem, our potential function is $\Phi(T) = |2 \cdot T.num - T.size|$. First consider the case in which the ith operation is a delete that does not cause a contraction. In this case, the actual cost of the operation is $c_i = 1$. Since there is no contraction, $size_i = size_{i-1}$. Since this is a delete operation, $num_i = num_{i-1} - 1$ and $num_{i-1} = num_i + 1$. If $2 \cdot num_i \geq size_i$ (i.e. $\alpha \geq 1/2$), then $|2 \cdot num_i - size_i| = 2 \cdot num_i - size_i$ and we have an amortized cost for the operation of

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&= 1 + |2 \cdot num_i - size_i| - |2 \cdot (num_i + 1) - size_i| \\
&= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_i + 2 - size_i) \\
&= 1 + 2 \cdot num_i - size_i - 2 \cdot num_i - 2 + size_i \\
&= -1
\end{aligned}$$

If the ith operation is a delete that does not cause a contraction, but $2 \cdot num_i < size_i$ (i.e. $\alpha_i < 1/2$), then $|2 \cdot num_i - size_i| = -(2 \cdot num_i - size_i)$. Also, $2 \cdot (num_i + 1) \leq size_i$ and $|2 \cdot (num_i + 1) - size_i| = -(2 \cdot (num_i + 1) - size_i)$. So now we have an amortized cost for the operation of

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&= 1 + |2 \cdot num_i - size_i| - |2 \cdot (num_i + 1) - size_i| \\
&= 1 + (-1)(2 \cdot num_i - size_i) - (-1)(2 \cdot num_i + 2 - size_i) \\
&= 1 - 2 \cdot num_i + size_i + 2 \cdot num_i + 2 - size_i \\
&= 3
\end{aligned}$$

If the ith operation is a delete that results in a contraction of the table, we have $c_i = num_i + 1$, since we must move num_i items and delete an item. We also know that $size_{i-1} = 3 \cdot num_{i-1} = 3(num_i + 1)$ and $size_i = \frac{2}{3} size_{i-1} = \frac{2}{3}(3 \cdot (num_i + 1)) = 2 \cdot num_i + 2$.

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&= num_i + 1 + |2 \cdot num_i - (2 \cdot num_i + 2)| - |2 \cdot (num_i + 1) - 3 \cdot (num_i + 1)| \\
&= num_i + 1 + |-2| - |-(num_i + 1)| \\
&= num_i + 1 + 2 - (num_i + 1) \\
&= num_i + 1 + 2 - num_i - 1 \\
&= 2
\end{aligned}$$

Hence, when the ith operation is a delete, the amortized cost is $\hat{c_i} \leq 3$.

5. **Problem 17-4 on page 474–475**

   Solution:

   (b) All cases except for case 1 of RB-INSERT-FIXUP and case 2 of RB-DELETE-FIXUP are terminating.

   (c) Case 1 of RB-INSERT-FIXUP reduces the number of red nodes by one. As Figure 13.5 shows, node z's parent and uncle change from red to black (-2 red), and z's grandparent changes from black to red (+1 red). Hence $\Phi(T') = \Phi(T) - 1$.

   (d) Lines 1-16 of RB-INSERT insert a red node which increases the potential by one. The nonterminating case of RB-INSERT-FIXUP (Case 1) makes three color changes and decreases the potential by 1. The terminating cases of RB-INSERT-FIXUP (cases 2 and 3) cause one rotation each which do not affect the potential. Case 3 makes color changes, but the potential does not change as the number of red nodes does not change.

(e) The number of structural modifications and amount of potential change resulting from lines 1-16 of RB-INSERT and from the terminating cases of RB-INSERT-FIXUP are $O(1)$, so the amortized number of structural changes is $O(1)$. The nonterminating case of RB-INSERT-FIXUP may repeat $O(\lg n)$ times, but its amortized number of structural modifications is zero, since by our assumption, the unit decrease in the potential pays for structural modifications needed. Therefore, the amortized number of structural modifications performed by RB-INSERT is $O(1)$.

(f) (extra credit) From Figure 13.5, we see that case 1 of RB-INSERT-FIXUP makes the following changes to the tree:

- Changes a black node with two red children (node C) to a red node, resulting in a potential change of -2.
- Changes a red node (node A in part (a) and node B in part (b)) to a black node with one red child, resulting in no potential change.
- Changes a red node (node D) to a black node with no red children, resulting in a potential change of 1.

The total change in potential is -1, which pays for the structural modications performed, and thus the amortized number of structural modications in case 1 (the nonterminating case) is 0. The terminating cases of RB-INSERT-FIXUP cause $O(1)$ structural changes. Because $w(v)$ is based solely on node colors and the number of color changes caused by terminating cases is $O(1)$, the change in potential in terminating cases is $O(1)$. Hence, the amortized number of structural modications in the terminating cases is $O(1)$. The overall amortized number of structural modications in RB-INSERT, therefore, is $O(1)$.

(g) (extra credit) Figure 13.7 shows that case 2 of RB-DELETE-FIXUP makes the following changes to the tree:

- Changes a black node with no red children (node D) to a red node, resulting in a potential change of -1.
- If B is red, then it loses a black child, with no effect on potential.
- If B is black, then it goes from having no red children to having one red child, resulting in a potential change of -1.

The total change in potential is either -1 or -2, depending on the color of B. In either case, one unit of potential pays for the structural modications performed, and thus the amortized number of structural modications in case 2 (the nonterminating case) is at most 0. The terminating cases of RB-DELETE cause $O(1)$ structural changes. Because $w(v)$ is based solely on node colors and the number of color changes caused by terminating cases is $O(1)$, the change in potential in terminating cases is $O(1)$. Hence, the amortized number of structural changes in the terminating cases is $O(1)$. The overall amortized number of structural modications in RB-DELETE-FIXUP, therefore, is $O(1)$.

(h) (extra credit) Since the amortized number structural modication in each operation is $O(1)$, the actual number of structural modications for any sequence of m RB-INSERT and RB-DELETE operations on an initially empty red-black tree is $O(m)$ in the worst case.