CSORW4231 HOMEWORK 4

Due Mon, Mar 27

Jun Hu

(jh3846)

Problem 1. Prove the following useful property of a binary search tree (with distinct keys): Property 1. Let x be a node in a BST T. Let max and min denote the largest and smallest keys in the subtree rooted at x, respectively. For any node y outside the subtree rooted at x, show that either $y.key > max$ or $y.key < min$ This implies that if there is a key k in the tree that satisfies $minkmax$ then it must lie inside the subtree rooted at x. (Here the subtree rooted at x includes x itself.) Use it to solve Exercise 12.2-5, 12.2-6 and 12.2-9 on page 293. In all three exercises, we assume the BST has distinct keys.

Solution.

Property 1 Proof. For a BST T with distinct keys:

(1) If x is rooted at y:

① If x is in the left subtree of y, max is also in the left subtree of y, then $y.key > max$;

② If x is in the right subtree of y, min is also in the right subtree of y, then $y.key < min$;

(2) If x and y are rooted at z (z is the lowest common ancestor of x and y):

① If x is in the left subtree of z, max is also in the left subtree of z, and y must be in the right subtree of z, then $y.key > z.key > max$;

② If x is in the right subtree of z, min is also in the right subtree of z, and y must be in the left subtree of z, then $y.key < z.key < min$;

In conclusion, either $y.key > max$ or $y.key < min$. So when a key k satisfies $min \leq k \leq max$, k can not be outside of the subtree rooted at x, such that k must lie inside the subtree rooted at x.

12.2-5 Let T be the BST with distinct keys: a subtree with the root node x, and x has two children, and min is the minimum key rooted at x, max is the maximum key rooted at x. Apparently,

$$min \leq x.l.key \leq x.predecessor.key < x.key < x.successor.key \leq x.r.key \leq max$$

As we have proved, $x.predecessor$ and $x.successor$ must lie in the subtree rooted at x.

Especially, $x.predecessor$ is in the left subtree of x, and $x.successor$ is in the right subtree of x.

If $x.predecessor$ has a right child $x.pre.r$, then $x.predecessor.key < x.pre.r.key < x.key$, which violates the predecessor definition.

If $x.successor$ has a left child $x.suc.l$, then $x.key < x.suc.l.key < x.successor.key$, which violates the successor definition.

12.2-6 Let max be the maximum key rooted at x. Because x does not have a right subtree, which means:

$$y.key > x.key = max$$

As we have proved, as long as x has a successor y, y must be outside the subtree rooted at x, and only two cases fit the $y.key > max$:

① y is an ancestor of x, and x is in the left subtree. If y is not the lowest ancestor of x, then there exists a $y.l$, $y.key > y.l.key > max = x$, $y.l$ becomes the successor. So y has to be the lowest ancestor of x, which means $y.l$ is x, so y's left child is also an ancestor of x

② z is the lowest common ancestor of x and y, x is in the left subtree of z, however because of $y.key > z.key > max = x$, z becomes the successor, this case does not hold.

12.2-9 Let min be the minimum key rooted at y, max be the maximum key rooted at y, and z be a node outside the subtree rooted at y. there exists two cases:

① $y.key > x.key$:

$$min = x.key < y.key \leq y.r.key \leq max$$

We have proved, either $z.key < min = x.key < y.key \leq y.r.key \leq max$ or $min = x.key < y.key \leq y.r.key \leq max < z.key$, implies $y.key$ is the smallest key in T larger than $x.key$;

② $y.key < x.key$:

$$min \leq y.l.key \leq y.key < x.key = max$$

We have proved, either $z.key < min \leq y.l.key \leq y.key < x.key = max$ or $min \leq y.l.key \leq y.key < x.key = max < z.key$, implies $y.key$ is the largest key in T smaller than $x.key$.

□

Problem 2. Problem 13-2: Join operation. Skip e) and f). Replace d).

Solution.

a **1)** During insertion, the red node violation may push upwards to the root, so the root node becomes red with two black children, then the root has to be recolored back to black. This is the only scenario that the $T.bh$ increases by one.

2) If the child node is red, $child.red.bh = h$. If the child node is black, $child.black.hb = h+1$. In $O(1)$ time.

b The core idea is searching the node along the right edge of T_1 from the root until reach the black height of $T_2.bh$.

Specifically, start with the root node of T_1, initiate a loop with $T_2.bh$ as i, while $i > 0$, move to the right child (move to the left child only if no right child exists), and i decreases by 1 when encountering a black node. Because $T_2.bh \leq T_1.bh$, this is true in every path in T_1, i will eventually equal to 0. Return that node, it must be the black node with the black height of $T_2.bh$.

The while loop iterates at most $T_1.bh$ times, so the worst case is when only one node in T_2, $n-1$ nodes in T_1, which is $O(\lg n)$.

c Create a new subtree T_x, let x be the root, y be the left child of x, T_2 be the right subtree of x. This only requires $O(1)$ time. And both T_y and T_2 are unchanged, maintaining their local binary-search-tree-property. Additionally, elements in T_y are less than or equal to x, and x is less than or equal to elements in T_2, the elements in T_x are larger or equal than those in $T_1 - T_y$. So binary-search-tree-property holds overall.

d If $T_1.bh = T_2.bh$, simply make x black to get a red-black tree.

If $T_1.bh > T_2.bh$, then make x red.

1. As we described in ⓑ, find the node y in T_1 with the height of $T_2.bh$.
2. We replace T_y in T_1 with T_x as we described in ⓒ.
3. Fix property 4 from x, call RB-INSERT-FIXUP (T, x), with $O(T_1.bh - T_2.bh)$, which is in $O(\lg n)$ time.

□

Problem 3. Exercise 14.1-8 on page 345.

Solution.

We can use an order-statistic tree OST to solve the problem:

Each chord with two endpoints 'starting' – s and 'ending' – e, such that for i from 1 to n: $c_i = (s_i, e_i)$, $s, e \in [0, \pi)$, and $s_i < e_i$.

Sort all endpoints e and s, totally $2n$ elements in to a list L, in $O(n \lg n)$ time.
For the endpoints from the smallest to the largest in L:
    If the endpoint is s_i of a chord c_i
        Insert s_i into OST, in $O(\lg n)$ time.
    If the endpoint is e_i of a chord c_i
        Count the number(n_i) of elements in OST larger than s_i, in $O(\lg n)$ time.
        Delete c_i, in $O(\lg n)$ time.
Return $\sum_{i=1}^{n} n_i$

Overall, as described above, the algorithm takes $O(n \lg n)$ time.

□

Problem 4. Exercise 14.3-6 on page 354.

Solution.

We augment a red-black tree T with the elements of Q as the keys of nodes.

1). Extra information to store in each node x:

min_gap: minimum gap in the subtree rooted at x, ∞ for the leaf node.

min_key: minimum key in the subtree rooted at x.

max_key: maximum key in the subtree rooted at x.

2). $T.root.min\_gap$ can answer MIN-GAP(Q) directly, which only takes $O(1)$ time.

3). Extra information for a node x can computer from:

$$min\_gap = \text{MIN} \begin{cases} x.l.min\_gap & \infty \text{ if no left subtree} \\ x.key - x.l.max\_key & \infty \text{ if no left subtree} \\ x.r.min\_gap & \infty \text{ if no right subtree} \\ x.r.min\_key - x.key & \infty \text{ if no right subtree} \end{cases}$$

$$min\_val = \begin{cases} x.l.min\_val & \text{with left subtree} \\ x.key & \text{without left subtree} \end{cases}$$

$$max\_val = \begin{cases} x.r.max\_val & \text{with right subtree} \\ x.key & \text{without right subtree} \end{cases}$$

In conclusion, all the extra information for a node x can be derived from the information stored in its two children in O(1) time. Hence by Theorem 14.1, insertion and deletion of a node can still be done in $O(\lg n)$ time.

□

Problem 5. Exercise 16.1-5 on page 422 and Exercise 16.3-5 on page 436.

Solution.

16.1-5 Each activity $a_k \in A$ contains start time s_i, finish time f_i and in addition a value v_i.

Considering we need to solve compatible problem, we first sort the activities by their finish time f, such that $f_1 \leq f_2 \leq \cdots \leq f_n$. So the number of compatible activities before a_j is c_j, such that the number of all activities a_i with $f_i < s_j$.

If we can always pickup an optimal activity from the end of the list and solve the remained compatible activities ahead, by recursively call the method, we will eventually solve the original problem. So we come up the following algorithms:

```
MAX-WEIGHTED-ACTIVITIES-VALUE(A)
    sort A by f
    MWAV[0] = 0
    for j from 1 to n
        calculate c_j
        MWAV[j] = MAX(MWAV(j − 1), MWAV(c_j) + v_j)
    return MWAV[n]
```

This will return the maximized value in $O(n)$ time.

```
OPT-ACTIVITIES(j)
    if MWAV(c_j) + v_j > MWAV(j − 1)
        print j
        OPT-ACTIVITIES(c_j)
    else
        OPT-ACTIVITIES(j − 1)
```

This will determine the optimal solution in $O(n)$ time.

16.3-5 Consider we invoke a priority queue pQ to store the characters and their joint combinations. At the beginning, the characters are sorted, then pushed into the pQ in reversed-alphabet order. Then the characters are popped out by their frequency, the lower frequency has higher priority. The joint combination will be pushed back into the qQ. So a character c_i or combination co_i will only be popped out the pQ when the characters and combinations with lower frequencies increased their frequencies by joining upto the c_i or co_i. This method is obviously optimal as Huffmans coding. Each joining operation will increase the codeword length by 1. To reach the final frequency, the lower frequency character will have to be joined more than a higher one. So their codeword lengths are longer. Because the characters frequencies are monotonically decreasing, these operations will generate monotonically increased codeword lengths.

□

Problem 6. Problem 16-2 on page 447.

Solution. In general, to minimize the average completion time, we always want to finish the task which is the shortest at the moment.

a Because:

$$c_1 = p_1, c_2 = p_1 + p_2, c_3 = p_1 + p_1 + p_2 + p_3, ...$$

The average completion time ACT:

$$\frac{1}{n}\sum_{i=1}^{n} c_i = \frac{1}{n}\sum_{i=1}^{n}(n - i + 1)p_i$$

To minimize it, we only need to sort S by p_i, such that $p'_1 \leq p'_2 \leq p'_3 \leq \cdots \leq p'_n$ Proof:

step 1. If n = 1 or 2, trivial.

step 2. Let S denote an optimal solution as described. But S' is not identical to S, with ACT'. There exists a pair a_i ahead of a_j, such that $p_i > p_j$. When we switch a_i and a_j, we will have the average completion time $ACT'' = ACT' - p_i + p_j < ACT'$. In conclusion, an optimal solution must be identical to S.

We can use `QUICK-SORT(S)` to sort S in $O(n \lg n)$, overall the time will be $O(n \lg n)$.

b The general idea is the same. The difference is there will be a new task a_i coming up at its releasing time r_i, so the current situation has to be initiated again. In order to initiate the current situation, we invoke a priority queue pQ to store tasks, always popping out the task with the smallest p for processing unless a new a new task a_i pushed into the pQ, then push the current running task with the its remaining processing time back into pQ. Then keep on popping and pushing and recording c_i until all tasks are done. The priority queue operations will run in $O(n \lg n)$.

We have already proved algorithm in **a** is correct and optimal. In this problem, all sub-problems are solved with the same idea that processing the task with the smallest processing time at the moment, so all the sub-problems must be correct and optimal. Such that the original problem must be correct and optimal.

□