Sample Solution for Problem Set 3

Data Structures and Algorithms, Fall 2019

November 10, 2020

Contents

1 Problem 1

(a) The third and the fifth.

(b) No. There is a counterexample. A binary search tree with 3 nodes, root (1), root's right child (3), right child of root's right child (2). Then the search path for value 3 makes $A = \{2\}, B = \{1, 3\}, C = \emptyset$, then $2 \in A, 1 \in B$ but $2 > 1$.

2 Problem 2

Algorithm 1: PARENT(T,x)

```
1  if x = T.root then
2  |   return NIL;
3  end
4  y=TREE-MAXIMUM(x).succ;
5  if y = NIL then
6  |   y=T.root
7  end
8  else
9  |   if y.left = x then
10 |   |   return y;
11 |   end
12 |   y=y.left;
13 end
14 while y.right != x do
15 |   y=y.right
16 end
17 return y
```

SEARCH No changes.

Algorithm 2: INSERT(T,z)

```
1  y = NIL ;
2  x = T.root ;
3  pred = NIL ;
4  while x != NIL do
5  |   y = x ;
6  |   if z.key < x.key then
7  |   |   x = x.left ;
8  |   end
9  |   else
10 |   |   pred = x ;
11 |   |   x = x.right ;
12 |   end
13 end
14 if y == NIL then
15 |   T.root = z ;
16 |   z.succ = NIL ;
17 end
18 else if z.key < y.key then
19 |   y.left = z ;
20 |   z.succ = y ;
21 |   if pred != NIL then
22 |   |   pred.succ = z ;
23 |   end
24 end
25 else
26 |   y.right = z ;
27 |   z.succ = y.succ ;
28 |   y.succ = z ;
29 end
```

Algorithm 3: TRANSPLANT(T,u,v)

```
1  p = PARENT(T, u) if p == NIL then
2  |   T.root = v
3  end
4  else if u == p.left then
5  |   p.left = v
6  end
7  else
8  |   p.right = v
9  end
```

Algorithm 4: TREE-PREDECESSOR(T, x)

```
1  if x.left != NIL then
2  |  return TREE-MAXIMUM(x.left)
3  end
4  y = T.root;
5  pred = NIL;
6  while y != NIL do
7  |  if y.key == x.key then
8  |  |  break
9  |  end
10 |  if y.key < x.key then
11 |  |  pred = y;
12 |  |  y = y.right;
13 |  end
14 |  else
15 |  |  y = y.left
16 |  end
17 end
18 return pred
```

Algorithm 5: DELETE(T, z)

```
1  pred = TREE-PREDECESSOR(T, z);
2  pred.succ = z.succ;
3  if z.left == NIL then
4  |  TRANSPLANT(T, z, z.right)
5  end
6  else if z.right == NIL then
7  |  TRANSPLANT(T, z, z.left)
8  end
9  else
10 |  y = TREE-MIMIMUM(z.right);
11 |  if PARENT(T, y) != z then
12 |  |  TRANSPLANT(T, y, y.right);
13 |  |  y.right = z.right;
14 |  end
15 |  TRANSPLANT(T, z, y);
16 |  y.left = z.left;
17 end
```

3 Problem 3

(3.a)

If all nodes in the binary search tree have identical keys, the TREEINSERT procedure always inserts the new node to the right end, which implys $h = O(n)$. $T(n) = T(n-1) + O(h) = T(n-1) + O(n) = O(n^2)$.

(3.b)

Since new nodes always enter the left and right subtrees of x **in turn**, we can find out that the differ from the size of left-subtree to right-subtree is no more than 1, which implys $h = O(\lg n)$. $T(n) = T(n-1) + O(h) = T(n-1) + O(\lg n) = O(n \lg n)$.

(3.c)

There is only one node in the binary search tree, $h = O(1)$. And we need $O(1)$ time to insert z into the list. $T(n) = T(n-1) + O(h) + O(1) = O(n)$.

4 Problem 4

(4.a)

- largest: 2 (a black root node with two red child).
  Proof: Only black nodes' children can be red.
- smallest: 0 (a black root node).
  Proof: Zero is the smallest non-negative number.

(4.b)

- Lemma: At most $n-1$ right rotations suffice to transform the tree into a right-going chain.
- Proof of lemma:
  - I.B: For $n = 1$, no rotation is required.
  - I.H: For $1 \leq n \leq k$, at most $n-1$ right rotations suffice to transform the tree into a right-going chain.
  - I.S: For $n = k+1$, assume the size of the left subtree is m ($1 \leq m \leq k$), then the size of the right subtree is $k-m$. From **I.H**, we can take at most

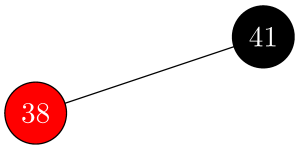
    $$t = (m-1) + \max\{0, (k-m-1)\} \leq k-1$$

    right rotations to transform them into right-going chains. Then take 1 right rotation at root, the whole tree transform into a right-going chain, also. $t+1 \leq n-1$.
- Proof of the original proposition:
  - We can use a stack to record the sequence of nodes that have been taken right rotations, take left rotations in reverse order can restore the original binary search tree.
  - Assume that we need to transform T_1 into T_2. Transform T_1 and T_2 to the right chain, record the right rotation order of T_2. Take these left rotations to T_1, which is a right-going chain in this moment.

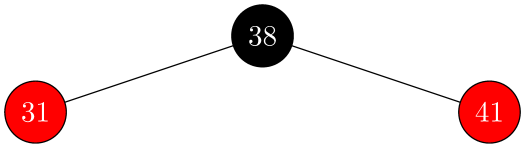
5 Problem 5

- insert 41:

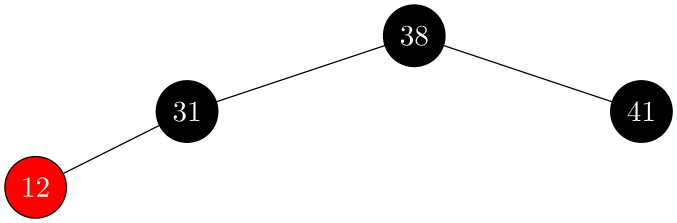
- insert 38:



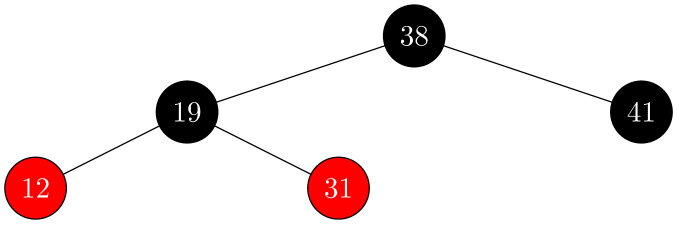
- insert 31:



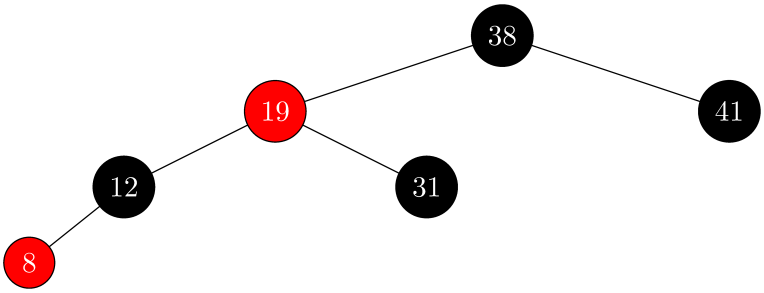
- insert 12:



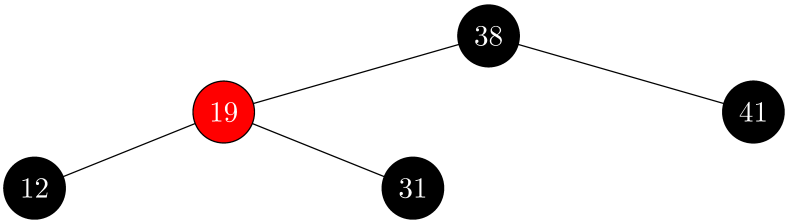
- insert 19:



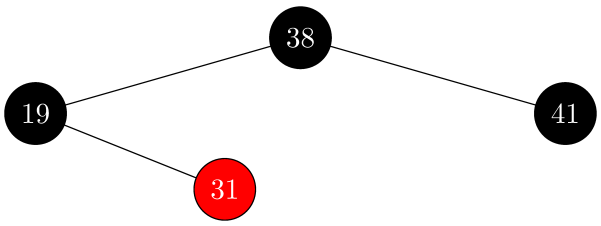
- insert 8:

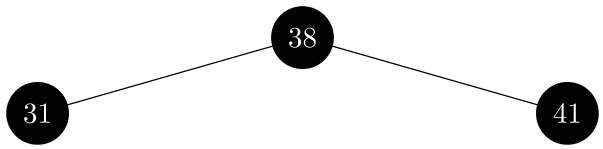


- delete 8:

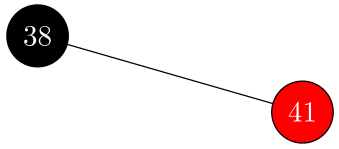


- delete 12:





- delete 19:



- delete 31:
- delete 41: (38)(black)

6 Problem 6

(a)

Let T_h be the minimum number of vertices that an AVL tree with height h has. It can be seen that $T_n \geq T_{n-1} + T_{n-2} + 1$, which shows that there exists some constant $C > 1$, so that $T_n \geq \exp(Cn)$ when n is large enough.

(b)

Algorithm

Algorithm 6: Balance(x)

Input: node x

if $x.left.h > x.right.h + 1$ **then**
  if $x.left.left.h < x.left.right.h$ **then**
    $Left - Rotate(x.left)$;
  end
  $Right - Rotate(x)$;
end
if $x.right.h > x.left.h + 1$ **then**
  if $x.right.right.h < x.right.left.h$ **then**
    $Right - Rotate(x.right)$;
  end
  $Left - Rotate(x)$;
end

Correctness

WLOG, we will only prove correctness when $x.left.h > x.right.h + 1$.

- If $x.left.left.h < x.left.right.h$, it can be seen that $x.left.right.h = x.left.left.h + 1$ from our basic assumption. Hence, We may perform $Left - Rotate$ operation on $x.left$, which maintains balance property for all vertices in the left subtree of x, while negating the fact that $x.left.left.h < x.left.right.h$.
- After first operation, it is guaranteed that $x.left.left.h \geq x.left.right.h$. We may perform $Right - Rotate$ operation on vertice x, achieving the goal.

(c)

(d)

We only needs to check it performs $O(1)$ rotations. It follows from the fact that Balance operation will reduce the height of unbalanced subtree by exactly 1.

Remark

We assume that Rotation operation will maintain information such as the height of subtree and so on.

Algorithm 7: Insert(x, z)

Input: node x, z

```
if x == NIL then
    z.h = 0;
    return;
end
if x.key < z.key then
    Insert(x.right, z);
    if x.right == NIL then
        x.right = z;
    end
end
else
    Insert(x.left, z);
    if x.left == NIL then
        x.left = z;
    end
end
left_h = −1, right_h = −1;
if x.left ≠ NIL then
    left_h = x.left.h;
end
if x.right ≠ NIL then
    right_h = x.right.h;
end
x.h = max(left_h, right_h) + 1;
Balance(x);
```

7 Problem 7

(a) Create n nodes $P[1...n]$ such that $P[i].key = A[i]$, and assign each $P[i]$ a random priority $P[i].pri$ (For example, let $P[i].pri$ be a random real number in $[0, 1]$, we assume this can be done in $O(1)$ time). Recall that each node in a Treap contain two attributes key and $priority$. Now we give a $O(n)$ algorithm to build a Treap T containing nodes $P[1...n]$.

Remark 1. *The structure of a $Treap$ is uniquely fixed when key and $priority$ are fixed for each node, by the following way: pick the node with the largest $priority$ as the root, split the array into its left subtree and right subtree, and do this recursively to build the subtrees. However, the trivial implementation of the idea will cost $O(n^2)$ time in worst case. We will show how to solve this in $O(n)$ time. The first method use rotations to maintain the treap property. The second method use **Monotonous stack** which is much more intricate but useful, it is highly recommanded to understand the second method .*

Method 1: Let $T.root = P[1]$. Build a right chain by the following loop:

- For $i = 1$ to $n - 1$, let $P[i].rightchild = P[i + 1]$ and $P[i + 1].p = P[i]$.

Now the BST property is satisfied, we use the following procedure to adjust the tree to maintain the heap property (suppose we want a **Max-heap**):

1 For $i = 2$ to n
   - Let $u \leftarrow P[i]$.
   - **While** u is not root and $u.pri > (u.p)\ .pri$, **do** $left - rotate(u.p)$ and let $u \leftarrow u.p$.

The correctness of the algorithm is guaranteed by the following loop invariance:

Invariance: After each loop in line 1, the Treap composed by $P[1...i]$ satisfies the heap property.

Initially the invariance is true since $P[1]$ is automatically a one node Treap. The third line of the algorithm try to add $P[i]$ into Treap $P[1...i - 1]$ while maintain the heap property. The proof is left to the reader.

Since rotations will not destroy the BST property, and according to the invariance, the heap property is satisfied, the algorithm correctly build a Treap.

Complexity: Consider the length of the right most chain of the tree T. Each rotation will eliminate one one from the right most chain. Since there are n nodes in the right most chain initially, there can be at most n times of rotations. The running time of the algorithm is dominated by the number of rotations, thus, the complexity is $O(n)$.

Method 2: Build an empty stack S and do the following procedure:

- For $i = 1$ to n, do
   - **While** S is not empty and $S.top().pri < P[i].pri$, **do** $u \leftarrow S.pop()$.
   - If S is empty, then let $T.root \leftarrow P[i]$.
   - Otherwise let $S.top().rightchild = P[i]$ and $P[i].p \leftarrow S.top()$. Let $u.p \leftarrow P[i]$ and $P[i].leftchild \leftarrow u$.
   - $S.push(P[i])$.

The stack S is call **Monotonous stack**, due to the fact that the priority of the nodes in S is always decreasing from bottom to the top. The correctness of the algorithm depend on the same loop invariance as method 1:

Invariance: After each loop, the Treap composed by $P[1...i]$ satisfies the heap property.

The proof is left to the reader.

The complexity is $O(n)$ since each element can be push and pop at most once.

(b) Suppose the skip list contains element $A[1...n]$, and it will raise an element with probability p, where p is a constant.

1. Let F_i be the event that $A[i]$ has been raised at least $2\log_{\frac{1}{p}} n$ times, then the probability for F_i to happen is $p^{2\log_{\frac{1}{p}} n} = \frac{1}{n^2}$. And according to **union bound** (An important method that you should learn carefully)

$$\Pr[\vee_{1\le i\le n} F_i] \le \sum_{1\le i\le n} \Pr[F_i] = n\cdot\frac{1}{n^2} = \frac{1}{n}$$

Thus, the probability for none of F_i happened is at leat $1-\Pr[\vee_{1\le i\le n}F_i] \ge 1-\frac{1}{n}$, which means with probability at least $1-\frac{1}{n}$ all elements are raised at most $2\log_{\frac{1}{p}} n = O(\log n)$ times.

2. Let X_i be the random variable of the number of nodes generated by $A[i]$. Obviously $E[X_i] = \frac{1}{1-p}$. Thus, we have

$$E[\text{number of nodes}] = E\left[\sum_{1\le i\le n} X_i\right] = \sum_{1\le i\le n} E[X_i] = \frac{n}{1-p} = O(n)$$

3. Chernoff bound has several useful forms. In this problem we need the multiplicative chernoff bound.

Lemma 2 (Multiplicative Chernoff bound)**.** *Suppose $X_1, X_2, ..., X_n$ are n independent random variables such that $a \le X_i \le b$ for all $1 \le i \le n$, let $X = \sum_{1\le i\le n} X_i$ and set $\mu = \mathbb{E}[X]$, then for all $\delta > 0$, we have*

$$\Pr\left[X > (1+\delta)\mu\right] \le e^{-\frac{2\delta^2\mu^2}{n(b-a)^2}}$$

The chernoff bound need X_i to be bounded ($a \le X_i \le b$).But recall that we define X_i as the random variable of the number of nodes generated by $A[i]$, which means X_i **is not bounded above**. But according to (a) we know they are bounded by $C \triangleq 4\log_{\frac{1}{p}} n$ with probability at lest $1-\frac{1}{n^2}$. That inspires us to define

$$X_i' = \begin{cases} 0 & X_i > C \\ X_i & X_i \le C \end{cases} \qquad X' = \sum_{1\le i\le n} X_i' \tag{1}$$

Now X_i is bounded by C. And according to (a), the probability for $X' = X$ is at least $1-\frac{1}{n^2}$.

Now we have $\mathbb{E}[X_i'] = c_0$ for some constant $1 < c_0 < 2$. Thus, we get $\mu = \mathbb{E}[X'] = c_0 n$. Apply Multiplicative Chernoff bound to $X_1', ..., X_n'$, we get

$$\Pr[X' > 2\mu] \le e^{-\frac{8c_0^2n^2}{nC^2}}$$

Note that $e^{-\frac{8c_0^2n^2}{nC^2}} = e^{-\Omega\left(\frac{n}{\log^2 n}\right)}$. For sufficiently large n we have $\Pr[X' > 2\mu] \le \frac{1}{n^2}$.

Let $\mathcal{A}$ be the event that $X' \ne X$, and let $\mathcal{B}$ be the event that $X' > 2\mu$. According to **union bound**, the probability for at least one of $\mathcal{A}$ and $\mathcal{B}$ to happen is at most $\frac{2}{n^2}$, which means with high probability, both $\mathcal{A}$ and $\mathcal{B}$ do not happen. Thus, with high probability, X is bounded by $2\mu = O(n)$.