论题 2-11 作业

姓名：陈劭源　　学号：161240004

1 [TC] Problem 12.1-2

In a binary search tree, every node is greater than or equal to all the elements in its left subtree, and is less than or equal to all the elements in its right subtree. However, in a min-heap, every node is less than or equal to its child(ren), both left and right, if exist.

Min-heap property can not be used to print out the keys in sorted order in $O(n)$ time. If it can, then we sort the n keys in $O(n)$ time, because building an n-node heap only takes a running time of $O(n)$, and this is contradictory to the $\Omega(n \log n)$ lower bound for comparison-based sorting algorithm.

2 [TC] Problem 12.1-5

Suppose, to the contrary, that there exists a comparison-based algorithm, that constructs an n-element binary search tree in $o(n \log n)$ time. We use this algorithm to build a binary search tree. The inorder traversal of the binary search tree gives the list of all the elements in the tree in sorted order, and it takes a running time of $O(n)$. That means, we can sort n elements in $o(n \log n)$ running time, which is contradictory to the $\Omega(n \log n)$ lower bound for comparison-based sorting algorithm.

3 [TC] Problem 12.2-5

When a node in a binary search tree has two children, then its successor is the minimum element in its right subtree. In TREE-MINIMUM, line 1, the **while** loop condition "$x.left \neq$ NIL" guarantees that when the loop terminates, the minimum element, x must not have a left child.

Likewise, the predecessor is the maximum element in its left subtree, and the **while** loop condition in TREE-MAXIMUM guarantees that the maximum element must not have a right child.

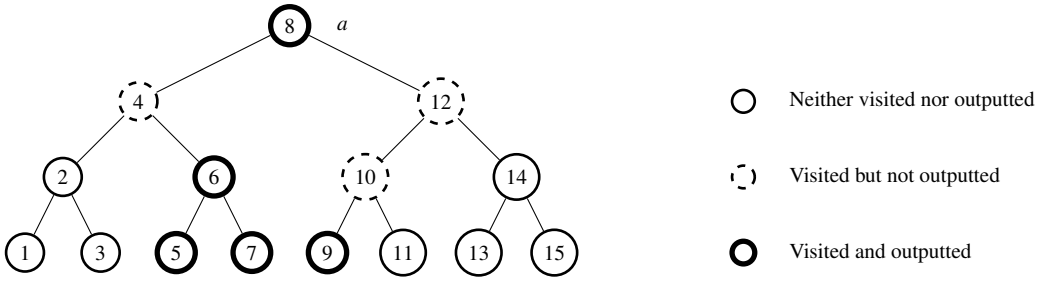
4 [TC] Problem 12.2-8

The k successive calls to TREE-SUCCESSOR output a consecutive subsequence of the inorder traversal of the tree. During this process, every node, say x, in the tree, is visited at most three times: entering the tree rooted in x from its parent, then visiting its left subtree; leaving its left subtree, outputting x itself, and then visiting its right subtree; leaving the right subtree, then returning to its parent. The k elements visited and outputted takes a running time of $O(k)$. Now, we are going to consider the elements visited but not outputted.

Assume, the lowest common ancestor of the outputted elements is a, then a must be outputted according to the binary search tree property. In the left subtree of a, we claim that there do not exist two elements b_1, b_2

at the same level visited but not outputted. Otherwise, let a' be their lowest common ancestor, and assume b_1 is in the left subtree of a' and b_2 is in the right one. Since both b_1 and b_2 have been visited, the procedure must have left the left subtree of a' and entered the right subtree of a', and thus a' must have been outputted. Now, we have proved that both a and a' have been outputted, but b_2, between a and a', is not outputted, which leads to a contradiction. So, there are at most h elements visited but not outputted in the left subtree of a, and they take a running time of $O(h)$. Likewise the elements visited but not outputted in the right subtree of a take a running time of $O(h)$.

Therefore, the total running time is $O(k+h)$.



Both in left and right subtree of a, there do not exist two elements visited but not outputted in the same level.

5 [TC] Problem 12.2-9

If x is the left child of y, since x is a leaf, x is the rightmost node in x's left subtree, so $x.key$ is the largest key in T smaller than $y.key$, i.e. $y.key$ is the smallest key in T larger than $x.key$.

If x is the right child of y, x is the leftmost node in x's right subtree because x is a leaf, so $x.key$ is the smallest key in T larger than $y.key$, i.e. $y.key$ is the largest key in T smaller than $x.key$.

6 [TC] Problem 12.3-5

PARENT(T,x)

```
1  y = x
2  while y.right ≠ NIL
3      y = y.right
4  y = y.succ
5  if y ≠ NIL and y.left == x // x is the left child of its parent
6      return y
7  else // x is the right child of its parent
8      if y == NIL
9          y = T.root
10     else
11         y = y.left
12     while y.right ≠ x
13         y = y.right
14     return y
```

TREE-SEARCH'(x,k)

// Since this procedure does not visit the parent of a node,

// the implementation is exactly the same as the original one.

TREE-INSERT'(T,z)

```
 1  y = NIL
 2  x = T.root
 3  pred = NIL
 4  while x ≠ NIL
 5      y = x
 6      if z.key < x.key
 7          x = x.left
 8      else
 9          pred = x
10          x = x.right
11  if y == NIL
12      T.root = z
13      z.succ = NIL
14  elseif z.key < y.key
15      y.left = z
16      z.succ = y // maintain the successor
17      pred.succ = z
18  else
19      y.right = z
20      z.succ = y.succ // maintain the successor
21      y.succ = z
```

TRANSPLANT'(T,u,v)

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

We should implement PREDECESSOR'(T,x) to maintain its successor when deleting a node. When x's left subtree is not empty, it is easy. However, when x's left tree is empty, we have to follow a path up the tree. Visiting x's parent takes a running time of $O(h)$, which is unacceptable. Instead, we try to search for x from the root of the tree. This might go wrong if some nodes have identical keys: the node we found might not be x, but has the same key. However, we notice that the implementation of TREE-INSERT' always insert a node to the rightmost empty child of the nodes with the same key. Hence, the nodes with the same key form a 'linked list'.

We could verify that the implementation of TREE-DELETE' preserves this property. Therefore, we could use this property to find a node correctly.

PREDECESSOR'(T,x)

```
1  if x.left ≠ NIL
2      return TREE-MAXIMUM(x.left)
3  else
4      y = T.root
5      pred = NIL
6      while x ≠ y
7          if x.key < y.key
8              y = y.left
9          else
10             pred = y
11             y = y.right
12     return pred
```

TREE-DELETE'(T,z)

```
1  pred = PREDECESSOR'(T,z)
2  if z.left == NIL
3      TRANSPLANT(T,z,z.right)
4  elseif z.right == NIL
5      TRANSPLANT(T,z,z.left)
6  else
7      y = TREE-MINIMUM(z.right)
8      if PARENT(T,y) ≠ z
9          TRANSPLANT(T,y,y.right)
10         y.right = z.right
11     TRANSPLANT(T,z,y)
12     y.left = z.left
13 pred.succ = z.succ
```

Every procedure above runs in $O(h)$ time.

7 [TC] Problem 12-1

a. When all the keys are identical, the binary search tree is degenerate, i.e. it becomes a linked list, so inserting n items with identical keys into an initially empty binary search tree takes a running time of $O(n^2)$.

b. The binary search tree is balanced, because this strategy ensures that, for every node, its larger subtree contains at most one more node than the other, which means, the height of an n-element tree is $\lceil \lg(n+1) \rceil$.

Inserting an element to a balanced tree of height h runs in $O(h)$ time. Therefore, inserting n identical elements takes a running time of

$$O\left(\sum_{i=1}^{n} \lceil \lg i \rceil\right) = O(n \lg n)$$

c. We have only to insert the node to the head of the list, which takes $O(1)$ time, and inserting n items takes $O(n)$ time in total.

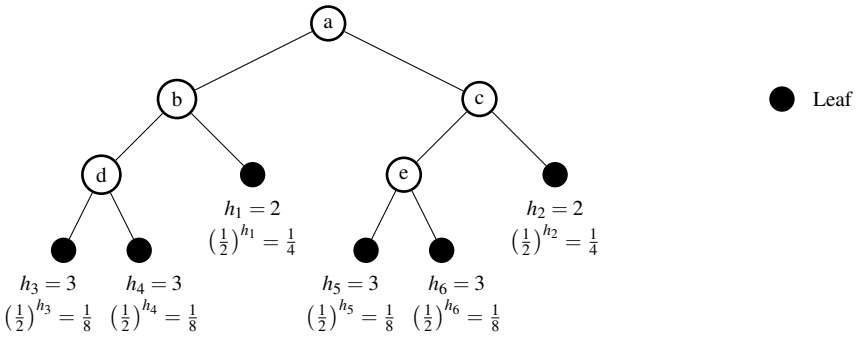
d. The worst case, is that, the tree is degenerate and becomes a linked list, and inserting n items with identical keys takes a running time of $O(n^2)$.

To derive the expected running time, consider the leaves (external nodes) in the tree. Let $h_1, h_2, \cdots h_k$ denote the height of the leaves in an n-element tree. We claim that $k = n+1$ and $\sum_{i=1}^{n+1} (1/2)^{h_i} = 1$. This can be easily proved by mathematical induction: when the tree is empty, the only external node is the root, whose height is 0; for any tree with n elements, a lowest internal node, assuming its height being h, just occupies a leaf of a tree with $n-1$ elements, but gives two new leaves of height $h+1$.

The expected running time of inserting a node into an n-element binary search tree is $E(X) = \sum_{i=1}^{n+1} h_i (1/2)^{h_i}$. Substitution $H_i = (1/2)^{h_i}$ gives

$$E(X) = -\sum_{i=1}^{n+1} H_i \lg H_i \qquad \text{with} \quad \sum_{i=1}^{n+1} H_i = 1$$

By Jensen's inequality, $E(X)$ is maximized when $H_1 = H_2 = \cdots = H_{n+1} = 1/(n+1)$, thus $E(X) \leq \lg(n+1) = O(\lg n)$. Therefore, the expected running time of inserting n elements is $O(n \lg n)$.



8 [TC] Problem 13.1-5

The shortest simple path and the longest simple path contain the same number of black nodes, and the shortest possible path contains only black nodes, and the longest possible path contains alternately arranged red and black nodes. Since the leaf must be black, the longest simple path is at most twice as long as the shortest simple path.

9 [TC] Problem 13.1-6

The smallest possible number is $2^k - 1$, which occurs when all the nodes are black.

The largest possible number is $4^k - 1$, which occurs when the red and black nodes are alternately arranged in every path.

10 [TC] Problem 13.1-7

TODO:

11 [TC] Problem 13.2-2

If a node has a right child which is not $T.nil$, we can perform a left rotation on it; if it has a left child which is not $T.nil$, we can perform a right rotation on it. So the number of possible rotations is the number of the children which is not $T.nil$. Note that every node in a binary search tree is a child of other node, except the root, so there are $n-1$ children which is not $T.nil$, thus there are exactly $n-1$ possible rotations in every n-node rotation binary search tree.

12 [TC] Problem 13.3-1

If we set z's color to black, though property 4 would not be violated, property 5 will be violated, because the path to z would have one more black nodes than other paths.

13 [TC] Problem 13.3-5

If a red-black tree has no red node, it must be a tree with every level completely filled, otherwise, property 5 is violated. Suppose, to the contrary that inserting n nodes with RB-INSERT can form such a tree. Consider the last element z inserted to the tree. z's parent must be black before insertion, otherwise, the other child of z's parent must be black, and the property 5 is violated. Therefore, property 2 and property 4 will not be violated after insertion, the procedure RB-INSERT-FIXUP does nothing and z remains red, which leads to contradiction.

14 [TC] Problem 13.4-1

In case 1, x's sibling is red, the procedure switches the colors of x's sibling and x's parent, and then performs a left rotation on x's parent, after which the parent node is black.

In case 2, the procedure only changes the color of x's sibling, which will not affect the color of the root.

In case 3, the procedure switches the colors of w and w's left child, then performs a right rotation, after which the parent node is black.

In case 4, x is assigned the root of the tree, which causes the **while** loop to terminate, and then the color of x is set black.

Therefore, after executing RB-DELETE-FIXUP, the root of the tree must be black.

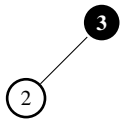
15 [TC] Problem 13.4-2

When both x and $x.p$ are red, the **while** loop in RB-DELETE-FIXUP will not be executed, and the color of x is set black, thus property 4 is restored.

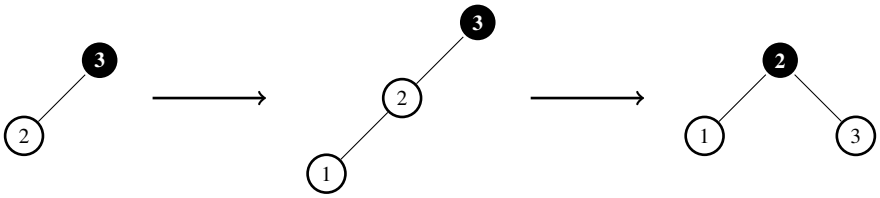
16 [TC] Problem 13.4-7

The resulting red-black tree might be changed. Here's an example.

Before insertion:



Inserting 1 to the red-black tree:



RB-DELETE-FIXUP, case 3 occurs:
Change the colors of nodes 2 and 3, then right rotate.

Deleting 1 from the red-black tree: