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 "$x.\mathit{left} \neq \text{NIL}$" 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.

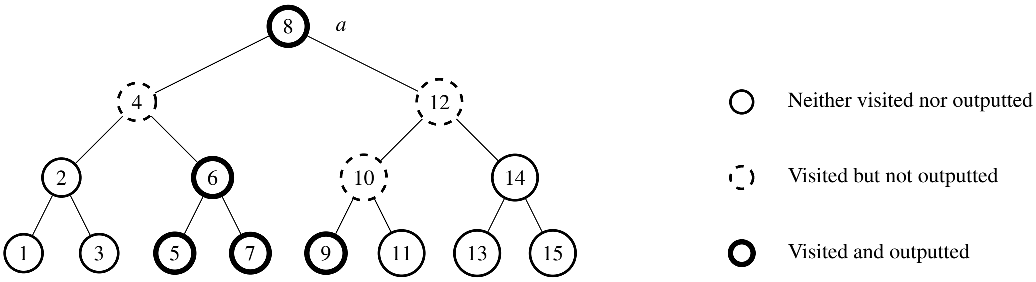
4 [TC] Problem 12.2-8

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 is at most one visited but not outputted element in a 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)
y = x
while y.right ≠ NIL
    y = y.right
y = y.succ
if y ≠ NIL and y.left == x // x is the left child of its parent
    return y
else // x is the right child of its parent
    if y == NIL
        y = T.root
    else
        y = y.left
    while y.right ≠ x
        y = y.right
    return y
```

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

TODO: