Chapter 12

12.1-2

BST: $x.left \leq x$ and $x.right > x$
Minimum Heap: $x.left \geq x$ and $x.right \geq x$
It's not possible, since for a heap, its left child and right child's keys doesn't have any greater or less relationship. So only by traversing the heap for $O(n)$ time is unable to output all keys in order.

12.1-5

Proof. Building a BST using sorting algorithm based on comparing is just like sort these data. Since the lower bound of comparison-based sorting algorithm is $\Omega(nlgn)$, the lower bound of building a BST by comparison is also $\Omega(nlgn)$. □

12.2-5

Proof. Denote this Node as x, then $x.left.key \leq x.key$ and $x.right.key \geq x.key$.
a.Suppose $S = SUCCESSOR(x)$, then if S.left exists, $S.left.key \leq S$, and S is located in x's right subtree. So $S.left.key \geq x.key$ and $S.left.key \leq S.key$, in this case, S.left is x's successor, which contradicts our assumption. As a result, x's successor doesn't have a left child.
b.Suppose $P = PREDECESSOR(x)$, and P has a right child P.right. We have $P.key \leq x.key$ and $P.right.key \geq P.key$. Besides, P.right is located in x's left subtree. So $P.right.key \leq x.key$, in this case, P.right is the predecessor of x, which contradicts our assumption. Therefore, x's predecessor doesn't have a right child. □

12.2-8

Proof. For the first time the function PRE invoked, it takes $O(h)$ time. For the next k-1 times, it traverses adjacent k-1 nodes. There are k-2 sides connecting them, and in the traversal, each side is passed at most twice. So the total time is $O(h) + 2(k-2) = O(h+k)$. □

12.2-9

Proof. If x is the left child of y, then $x.key < y.key$ and $y.right.key > y.key > x.key$. For since x is the left child of y, all decedents in y's right subtree is larger than x and y. So y.key is the least key larger than x.key. Similarly, if x is the right child of y, y.key is the largest key than x.key. □

12.3-5

12.1
a:$O(h+n)$
b:

```
1  void Tree_Insert(tree *T, node *z)
2  {
3      node *x = T->root, *y = NULL;
4      while(x != NULL)
5      {
6          y = x;
7          if(z->key == x->key)
8          {
9              if(x->b == 0)
10                 x = x->left;
11             else
12                 x = x->right;
13             x->b = !x->b;
14         }
15         else if(z->key < x->key)
16             x = x->left;
```

```
17          else
18              x = x->right;
19      }
20      z->p = y;
21      if(y == NULL)
22          T->root = z;
23      else if(z->key == y->key)
24      {
25          if(y->b == 0)
26              y->left = z;
27          else y->right = z;
28          y->b = !y->b;
29      }
30      else if(z->key < y->key)
31          y->left = z;
32      else y->right = z;
33  }
```

c:

```
1   void Tree_Insert(tree *T, node *z)
2   {
3       node *x = T->root, *y = NULL;
4       while(x != NULL)
5       {
6           y = x;
7           if(z->key == x->key)
8           {
9               z->next = x->next;
10              x->next = z;
11              return;
12          }
13          else if(z->key < x->key)
14              x = x->left;
15          else
16              x = x->right;
17      }
18      z->p = y;
19      if(y == NULL)
20          T->root = z;
21      else if(z->key == y->key)
22      {
23          z->next = y->next;
24          y->next = z;
25      }
26      else if(z->key < y->key)
27          y->left = z;
28      else y->right = z;
29  }
```

d:

```
1   void Tree_Insert(tree *T, node *z)
2   {
3       node *x = T->root, *y = NULL;
4       while(x != NULL)
5       {
```

```
6           y = x;
7           if(z->key == x->key)
8           {
9               if(rand()%2 == 0)
10                  x = x->left;
11              else
12                  x = x->right;
13          }
14          else if(z->key < x->key)
15              x = x->left;
16          else
17              x = x->right;
18      }
19      z->p = y;
20      if(y == NULL)
21          T->root = z;
22      else if(z->key == y->key)
23      {
24          if(rand()%2 == 0)
25              x = x->left;
26          else
27              x = x->right;
28      }
29      else if(z->key < y->key)
30          y->left = z;
31      else y->right = z;
32  }
```

Chapter 13

13.1-5

Proof. Since this tree has a certain number of black node along every way downwards, the shortest way's length is the number of black nodes. According to the theorem, red nodes can't be adjacent, and red node's child must be black, the longest is that black and red nodes come up one after another. That is twice the number of the certain number black number has along one way downwards. So the longest way is at most twice the length of the shortest one. □

13.1-6

According to lema 13.1, there is at most $2^{2k} - 1$, at least $2^k - 1$ inside nodes.

13.1-7

If black and red nodes come up one after another, and all nodes at bottom is red, the ratio at most is 2:1. If all this tree's nodes is black, the ratio at least is 0.

13.2-2

Proof. There are n nodes, then there are n-1 sides, every side can be rotated left or right, so there are n-1 possible ways of rotation. □

13.3-1

If that node is black, then the fifth property of a RB Tree is violated, making it much more difficult to adjust.

13.3-5

Proof. If an insertion doesn't violate any property of a RB Tree, it's obvious that this node is red. Then suppose the worst case, all of the n-1 nodes are adjusted and made black. Wherever the last node is inserted, since it's first made red, it won't violate any property. So the last node must be red. Therefore, there is at least one red node. □

13.4-1

Proof. If x is the root, then it will be made black.If x is red and is not root, the loop won't take effect, and the root remains black. If x is black and is not root, then the program will enter the loop. x.p will be made red(see line 6). After case 1, it will enter case 2, 3, 4. In case 2, x.p is the new x node and made red. Loop terminates. At last x is made black. If entering case 3, 4, (see line 18)x.p is made black, if x.p is root, it remains black. If it directly enters case 2, then x.p is replaced by x. If the new x is red, loop terminates, at last it is made black. If it directly enters case 3, 4, if x.p is root, it will made black in line 18, then in case terminates. After all, the root is still black. □

13.4-2

Proof. If x and x.p are both red, it won't enter loop. Instead, x is made black to maintain the fourth property. □

13.4-7
Not the same.

Proof. □