Assignment 3 Solution

reference: https://walkccc.me/CLRS/

1.1

10.1-2

At the beginning, top of the first stack is A[1], top of the second stack - A[n]. When pushing element to first stack, increase the iterator, when pushing to the second stack - decrease it.

10.1-6

```
class MyQueue {
    stack<int>s1;
    stack<int>s2;
public:
    /** Initialize your data structure here. */
    MyQueue() {

    }

    /** Push element x to the back of queue. */
    void push(int x) {
        s1.push(x);
    }

    /** Removes the element from in front of queue and returns that element. */
    int pop() {
        int x = peek();
        s2.pop();
        return x;
    }

    /** Get the front element. */
    int peek() {
        if(s2.empty()) transit();
        return s2.top();
    }

    void transit() {
        while(!s1.empty())
        {
            int v = s1.top();
            s1.pop();
            s2.push(v);
        }
    }

    /** Returns whether the queue is empty. */
    bool empty() {
        return s1.empty() && s2.empty();
    }
};

/**
 * Your MyQueue object will be instantiated and called as such:
 * MyQueue obj = new MyQueue();
 * obj.push(x);
```

```
 * int param_2 = obj.pop();
 * int param_3 = obj.peek();
 * bool param_4 = obj.empty();
 */
```

最坏情况pop需要O(n),但是pop的平均情况只需要O(1)

10.2-5

```
LIST-SEARCH'(L, k):
    key[nil[L]] = k
    x ← next[nil[L]]
    while(key[x] != k):
        x ← next[x]
    if x == nil[L]:
        return NULL
    return x
```

1.2

12.2-7

To show this bound on the runtime, we will show that using this procedure, we traverse each edge twice. This will suffice because the number of edges in a tree is one less than the number of vertices.

Consider a vertex of a BST, say x. Then, we have that the edge between x.p and x gets used when successor is called on x.p and gets used again when it is called on the largest element in the subtree rooted at x. Since these are the only two times that that edge can be used, apart from the initial finding of tree minimum. We have that the runtime is O(n). We trivially get the runtime is Ω(n) because that is the size of the output.

12.3-4

No, giving the following courterexample.

- Delete A first, then delete B:

```
  A        C        C
 / \      / \        \
B   D    B   D        D
   /
  C
```

- Delete B first, then delete A:

```
  A        A        D
 / \        \      /
B   D        D    C
   /        /
  C        C
```

12.3-5

We don't need to change SEARCH.

We have to implement PARENT, which facilitates us a lot.

```
PARENT(T, x)
    if x == T.root
        return NIL
    y = TREE-MAXIMUM(x).succ
    if y == NIL
        y = T.root
    else
        if y.left == x
            return y
        y = y.left
    while y.right != x
        y = y.right
```

```
    return y

INSERT(T, z)
    y = NIL
    x = T.root
    pred = NIL
    while x != NIL
        y = x
        if z.key < x.key
            x = x.left
        else
            pred = x
            x = x.right
    if y == NIL
        T.root = z
        z.succ = NIL
    else if z.key < y.key
        y.left = z
        z.succ = y
        if pred != NIL
            pred.succ = z
    else
        y.right = z
        z.succ = y.succ
        y.succ = z
```

We modify TRANSPLANT a bit since we no longer have to keep the pointer of p.

```
TRANSPLANT(T, u, v)
    p = PARENT(T, u)
    if p == NIL
        T.root = v
    else if u == p.left
        p.left = v
    else
        p.right = v
```

Also, we have to implement TREE-PREDECESSOR, which helps us easily find the predecessor in line 2 of DELETE.

```
TREE-PREDECESSOR(T, x)
    if x.left != NIL
        return TREE-MAXIMUM(x.left)
    y = T.root
    pred = NIL
    while y != NIL
        if y.key == x.key
            break
        if y.key < x.key
            pred = y
            y = y.right
        else
            y = y.left
    return pred
```

```
DELETE(T, z)
    pred = TREE-PREDECESSOR(T, z)
    pred.succ = z.succ
    if z.left == NIL
        TRANSPLANT(T, z, z.right)
    else if z.right == NIL
        TRANSPLANT(T, z, z.left)
    else
        y = TREE-MIMIMUM(z.right)
        if PARENT(T, y) != z
            TRANSPLANT(T, y, y.right)
            y.right = z.right
        TRANSPLANT(T, z, y)
        y.left = z.left
```

Therefore, all these five algorithms are still O(h) despite the increase in the hidden constant factor.

1.3

6.1-2

https://walkccc.me/CLRS/Chap06/6.1/

6.1-6

NO, because 7 > 6