DS

Sort

	平均复杂度	最优复杂度	最坏复杂度	辅助空间	稳定性
Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	Stable
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Unstable
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	Stable
Shell Sort	$O(n^{\frac{7}{6}})$	$O(n)$	$O(n^{\frac{4}{3}})$	$O(1)$	Unstable
Heap Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	Unstable
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	Stable
Quick Sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$	Unstable
Bucket Sort	$O(n+\frac{n^2}{k}+k)$	$O(n)$	$O(n^2)$	$O(k+n)$	Stable

通过交换相邻元素的排序算法需要$\Omega(n^2)$

Quick Sort 的辅助空间是递归额外使用的栈空间。

对于归并排序的总体复杂度有：

$$T(N)=2T(N/2)+N$$

求解后:

$$T(N)=N+N\log N=O(N\log N)$$

我们总共需要的Merge Times 是$O(\log N)$, 每次Merge的时间开销大概是$O(\log N)$

杂项

斐波那契数列：递归：$O(2^n)$ 迭代：$O(n)$, 算法：二叉树叶子数就是运行时间，等价于求解 $T(n)=T(n-1)+T(n-2)$

Tree

Nodes = Degree + 1

Hash Table

Hash Table Find的平均长度

处理办法	查找成功	查找失败	插入
Separate Chaining	$1+\frac{\lambda}{2}$	$\lambda+e^{-\lambda}$	与失败相同
Linear Probing	$\frac{1}{2}(1+\frac{1}{(1-\lambda)})$	$\frac{1}{2}(1+\frac{1}{(1-\lambda)^2})$	与失败相同
Quadratic Probing	$-\frac{1}{\lambda}\ln(1-\lambda)$	$\frac{1}{1-\lambda}$	与失败相同
Double Hashing	$-\frac{1}{\lambda}\ln(1-\lambda)$	$\frac{1}{1-\lambda}$	失败相同

Hash表的平均查找与插入时间复杂度为$O(1)$，但真正的复杂度取决于他的冲突解决方法，最坏复杂度为$O(n)$

查找失败总是比查找成功的平均长度长

定理：如果hash table的长度是质数，且表至少有一半是空的，那么使用Quadratic probing 总是能插入一个新的元素。

如果这个素数的形式是4k+3，且使用probing方法是$F(i)=\pm i^2$，那么整个哈希表都可以被探测到。

$F(i)=\pm i^2$的含义是每次探测先加再减

程序填空题

2015-2016秋冬期末

5-1 Please fill the array with the results after the following union/find operations.

```
union( find(4), find(6) )
union( find(2), find(7) )
union( find(0), find(4) )
union( find(7), find(6) )
union( find(7), find(1) )
```

Note: Assume `union-by-size` (if two sets are equal-sized, the first root will be the root of the result) and `find-with-path-compression`. `S[i]` is initialized to be -1 for all $0 \le i \le 7$.

i	0	1	2	3	4	5	6	7
S[i]	4	-6 (1分)	6 (1分)	−1	1 (1分)	−1	4	2 (1分)

4 4 -6 4

评测结果：答案错误（0 分）

序号	结果	得分
0	答案错误	0
1	答案错误	0
2	答案错误	0
3	答案错误	0

5-2 A binary tree is said to be "height balanced" if both its left and right subtrees are height balanced, and the heights of its left and right subtrees can differ by at most 1. That is, $|H_L - H_R| \le 1$ where H_L and H_R are the heights of the left and right subtrees, respectively. An empty binary tree is defined to be height balanced.

The function `IsBalanced` is to judge if a given binary tree `T` is height balanced. If the answer is yes then return `true` and store the tree height in the parameter `pHeight`, else simply return `false`. The height of an empty tree is defined to be 0.

```
typedef struct TNode *BinTree;
struct TNode{
    int Key;
    BinTree Left;
    BinTree Right;
};

bool IsBalanced ( BinTree T, int *pHeight )
{
    int  LHeight, RHeight, diff;

    if( T == NULL) {
        *pHeight = 0;
        return true;
    }
    else if ( IsBalanced(T->Left, &LHeight) && IsBalanced(T->Right, &RHeight) ) {
        diff = LHeight - RHeight;
        if ( (diff<=1)&&(diff>=-1) (6分) ) {
            *pHeight = 1 + ( diff<0 ? -1*diff:diff (6分) );
            return true;
        }
        else return false;
    }
    return false;
}
```

(RH : LH)

评测结果：部分正确（6 分）

序号	结果	得分
0	答案正确	6
1	答案错误	0

5-3 The function is to sort the list { `r[0] … r[n-1]` } in non-increasing order. Unlike selection sort which places only the maximum unsorted element in its correct position, this algorithm finds both the minimum and the maximum unsorted elements and places them into their final positions.

```
void  sort( list r[], int n )
{
   int i, j, mini, maxi;

   for (i=0; i<n-i-1; i++) {
      mini = maxi = i;
      for( j=i+1; j< n-i-1 (3分); ++j ){
         if( r[j]->key<r[mini]->key (3分) ) mini = j;
         else if(r[j]->key > r[maxi]->key) maxi = j;
      }
      if( maxi != i ) swap(&r[maxi], &r[i]);
      if( mini != n-i-1 ){
         if( r[maxi]->key == r[mini]->key (3分) ) swap(&r[maxi], &r[n-i-1]);
         else swap(&r[mini], &r[n-i-1]);
      }
   }
}
```

j<n-i

mini==i

评测结果：部分正确（3 分）

序号	结果	得分
0	答案错误	0
1	答案正确	3
2	答案错误	0

函数题

2015-2016秋冬期末

6-1 CheckBST[2]

Given a binary tree, you are supposed to tell if it is a binary search tree. If the answer is yes, try to find the K-th smallest key, else try to find the height of the tree.

Format of function:

```
int CheckBST ( BinTree T, int K );
```

where `BinTree` is defined as the following:

```
typedef struct TNode *BinTree;
struct TNode{
    int Key;
    BinTree Left;
    BinTree Right;
};
```

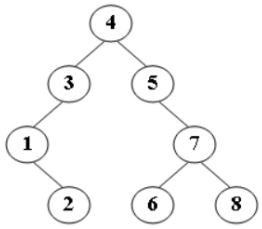
The function `CheckBST` is supposed to return the `K`-th smallest key if `T` is a binary search tree; or if not, return the negative height of `T` (for example, if the height is 5, you must return -5).

Here the height of a leaf node is defined to be 1. `T` is not empty and all its keys are positive integers. `K` is positive and is never more than the total number of nodes in the tree.

Sample program of judge:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct TNode *BinTree;
struct TNode{
    int Key;
    BinTree Left;
    BinTree Right;
};

BinTree BuildTree(); /* details omitted */
int CheckBST ( BinTree T, int K );

int main()
{
    BinTree T;
    int K, out;

    T = BuildTree();
    scanf("%d", &K);
    out = CheckBST(T, K);
    if ( out < 0 )
        printf("No.  Height = %d\n", -out);
    else
        printf("Yes.  Key = %d\n", out);

    return 0;
}
/* 你的代码将被嵌在这里 */
```

Sample Input 1: (for the following tree)

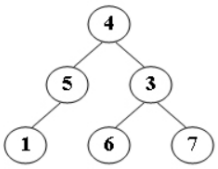



```
3
```

Sample Output 1:

```
Yes.  Key = 3
```

Sample Input 2: (for the following tree)




```
2
```

Sample Output 2:

```
No.  Height = 3
```

不想写了，写了几遍都看错，看成写判断平衡二叉树，艹。

2016-2017秋冬期末

函数题 总分：8 得分：8

6-1 LCA in BST

The lowest common ancestor (LCA) of two nodes `u` and `v` in a tree `T` is the deepest node that has both `u` and `v` as descendants. Given any two nodes in a binary search tree (BST), you are supposed to find their LCA.

Format of function:

```
int LCA( Tree T, int u, int v );
```

where `Tree` is defined as the following:

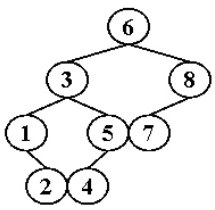
```
typedef struct TreeNode *Tree;
struct TreeNode {
    int   Key;
    Tree  Left;
    Tree  Right;
};
```

The function `LCA` is supposed to return the key value of the LCA of `u` and `v` in `T`. In case that either `u` or `v` is not found in `T`, return `ERROR` instead.

Sample program of judge:

```
#include <stdio.h>
#include <stdlib.h>

#define ERROR -1
typedef struct TreeNode *Tree;
struct TreeNode {
    int   Key;
    Tree  Left;
    Tree  Right;
};

Tree BuildTree(); /* details omitted */
int LCA( Tree T,  int u, int v );

int main()
{
    Tree T;
    int u, v, ans;

    T = BuildTree();
    scanf("%d %d", &u, &v);
    ans = LCA(T, u, v);
    if ( ans == ERROR ) printf("Wrong input\n");
    else printf("LCA = %d\n", ans);

    return 0;
}

/* Your function will be put here */
```

Sample Input 1 (for the tree shown in the Figure):




```
2 7
```

Sample Output 1:

```
LCA = 6
```

Sample Input 2 (for the same tree in Sample 1):

```
1 9
```

Sample Output 2:

```
Wrong input
```

AC代码

```
#define MAXN 128

int dfs(Tree T, int X, int *Stack, int *Sp);

int LCA(Tree T, int u, int v) {
    int stack1[MAXN];
    int stack2[MAXN];
    int sp1 = -1;
    int sp2 = -1;
    int f1 = dfs(T, u, stack1, &sp1);
    int f2 = dfs(T, v, stack2, &sp2);
    if (f1 * f2 == 0) {
        return ERROR;
    }
    int min = (sp1 - sp2 < 0) ? sp1 : sp2;
    for (int i = 0; i <min+1; ++i) {
        if (stack1[i]!=stack2[i])
        {
            return stack1[i-1];
        }
    }
    return stack1[min];


}

int dfs(Tree T, int X, int *Stack, int *Sp) {
    Tree cur = T;
    while (1) {
        if (cur == NULL) {
            return 0;
        }
        Stack[++*Sp] = cur->Key;
        if (cur->Key > X) {
            cur = cur->Left;
        } else if (cur->Key < X) {
            cur = cur->Right;
        } else {
            return 1;
        }
    }

}
```