数据结构第五讲

一、算法

1.1 定义

程序 = 数据结构 + 算法
算法：计算机解决问题的方法或者步骤
结构设计是程序的肉体、算法设计是灵魂

1.2 算法的特性及设计要求

```
算法特性：
                    确定性：每一条语句要有确定的含义，不能模棱两可
                    有穷性：执行一定的时间后会自动结束
                    输入：0个或多个输入
                    输出：至少一个或多个输出
                    可行性：能够执行、经济可行性
算法设计要求：
                    正确性：要求代码结果正确、不能有报错（所有测试数据能得到正确的结果）
                    健壮性：对不合理的输入给出合理的处理方式：例如switch中的default语句
                    可读性：要求关键代码有注释、命名合法、代码有缩进
                    高效率：算法时间复杂度尽可能低
                    低存储：降低对内存空间的开辟
```

1.3 时间复杂度

1>　是一个算法执行所耗费时间量度的函数
2>　计算公式为：T(n)=O(f(n))

```
T（n）：时间复杂度
n：问题规模----数据的范围或大小。
f（n）：问题规模的函数，是基本操作重复执行次数与n的关系。
用大写O（）来体现时间复杂度的记法称大O阶记法。
如：在T（n）=O（n3）中
                f（n）=n3------表示某语句的重复执行次数为n3次
```

3>　O阶记法推导过程

```
用常数1取代运行时间中的所有常数项。
          如：f（n）=2n2+5n+6
          则：T（n）=O（2n2+5n+1）
在修改后的运行次数函数中，只保留最高阶项。
                    T（n）=O（2n2）
如果最高阶项存在且系数不为1，则去除与这个项相乘的系数。
                    T（n）=O（n2）
得到的结果就是大O阶。
                      时间复杂度为O（ n2 ）阶
```

4> 总结：没有循环的话，时间复杂度为常数阶、一层循环线性阶、两层循环平方阶。。。

当循环体中，涉及到对循环变量的改变时，则该层循环变成对数阶

二、排序算法

2.1 排序定义

所谓排序，就是根据关键字，按照升序或者降序的方式，给数据进行重新排列的过程

2.2 排序的分类

1> 交换类排序：冒泡排序、快速排序
2> 选择类排序：简单选择排序、堆排序
3> 插入类排序：直接插入排序、折半插入排序、希尔排序
4> 归并排序：二路归并、三路归并。。。
5> 基数排序

2.3 冒泡排序（O(N^2)）

1> 定义：在排序过程中，越大（越小）的数据，经由交换后会慢慢“浮”到顶端、如同水中的起泡最终会上浮到顶端一样，故名为冒泡排序

2> 原理

```
1、比较相邻的两个元素，如果前面的比后面的大（小）,则交换
2、对每一组相邻的元素做同样的操作，从开始的第一对到结尾的最后一对
3、针对于所有的元素继续重复上述操作
4、直到没有一组数据需要交换为止
```

3> 算法

```
void pop_sort(int *arr, int n)
```

```
{
    int i,j;                //循环变量
    int t;                  //交换变量

    for(i=1; i<n; i++)              //外层循环控制趟数
    {
        int flag = 0;               //排序开始之前设立一个旗帜

        for(j=0; j<n-i; j++)        //控制元素下标、以及比较次数
        {
            if(arr[j] > arr[j+1])       //大升小降    if(*(arr+j) > *(arr+j+1))
            {
                //交换三部曲
                t=arr[j];  arr[j]=arr[j+1];  arr[j+1]=t;
                flag = 1;                                   //说明当前趟中进行了交换
            }
        }

        if(flag == 0)
        {
            break;              //说明当前趟没有数据进行交换，也反应了数列已经有序，后面就不需要再进行排了
        }
    }
}
```

2.4 选择排序（O(N^2)）

1> 定义：指在待排序的序列中找出最大（小）值，然后将其与待排序的序列中第一个元素进行交换

2> 原理：

```
1、从待排序的序列中找出最值
2、如果最值元素不是待排序序列的第一个元素，则将其和第一个元素进行交换
3、从剩余的待排序序列中，重复指向1、2两步，直到排序出结果
```

3> 算法：

```
void select_sort(int *arr, int n)
{
    int i,j;            //循环变量
    int min;           //记录最小值下标
```

```
    int t;              //交换变量

    for(i=0; i<n; i++)        //将所有元素遍历一遍
    {
        min = i;          //将待排序的第一个元素的下标当做最小值下标
        for(j=i+1; j<n; j++)   //从待排序的序列中找最小值
        {
            if(arr[min] > arr[j])
            {
                min = j;                //更新最小值下标
            }
        }

        //判断最小值下标是不是待排序序列的第一个元素
        if(min != i)
        {
            //交换三部曲
            t=arr[min]; arr[min]=arr[i]; arr[i]=t;
        }

    }
    printf("排序成功\n");

}
```

2.5 快速排序（O(N*log2N)：N乘以以2为底的n的对数)

1> 定义：将序列元素与选定的基准进行比较，通过基准将序列分为大小两个部分的交换类排序

2> 原理：

```
1、从待排序的序列中，任意选择一个元素作为基准
2、将其他元素与基准进行比较，分为大小两个部分
3、再对各个部分重新选定基准，并以上述两步重复进行，直到每个部分只剩一个元素为止
```

3> 算法：

```
/**********************定义一趟快排函数*********************/
int part(int *arr, int low, int high)
{
```

```
    int x = arr[low];       //将第一个设为基准

    while(low<high)             //控制循环不至于一遍过后就结束
    {
        while(arr[high]>=x && low<high)         //防止low超过high
        {
            high--;
        }
        arr[low] = arr[high];

        while(arr[low]<=x  && low<high)     //防止low超过high
        {
            low++;
        }
        arr[high] = arr[low];
    }

      arr[low] = x;                 //将基准放入中间位置
      return low;                   //将基准所在的位置返回
}

/**********************定义快排函数***********************************/
void quick_sort(int *arr, int low, int high)
{
    if(low < high)
    {
        int mid = part(arr, low, high);           //调用一趟排序函数

        quick_sort(arr, low, mid-1);           //对基准左侧的序列进行快速排序
        quick_sort(arr, mid+1, high);          //对基准右侧的序列进行快速排序
    }
}
```

三、查找算法

3.1 定义

根据所给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素（或一条记录）的操作

3.2 顺序查找(O(N))

所谓顺序查找，就是将所有数据元素遍历一遍，与要查找的值进行比较，目前你所掌握的查找就是顺序查找

3.3 折半查找(O(1/2 N))

所谓折半查找，是在顺序存储的有序序列中，通过逐次减半查询范围，查找某个特定元素的查找方法
要求：数据元素必须有序、顺序存储

算法：

```
int half(int *arr, int n, int key)              //key是要查找的数据，n是数组长度
{
    int low = 0;            //定义变量接受最小下标
    int high = n-1;         //定义变量接受最大下标
    int mid;

    while(low<=high)
    {
        mid = (low+high)/2;           //找到中间元素的下标

        if(arr[mid] == key)
        {
            return mid;               //将查找到的数据下标返回
        }else if(arr[mid] < key)
        {
            low = mid+1;          //将最小下标更新，舍弃前面一半数据
        }else if(arr[mid] > key)
        {
            high = mid-1;        //将最大值下标更新，舍弃后面一般数据
        }
    }

    return -1;            //说明没有找到
}
```

3.4 哈希查找

1> 定义：先将要查找的序列元素，按照关键字特点，存储到哈希表中，然后再到哈希表中进行查找的方法

2> 哈希表：哈希表是借助哈希函数将序列存储于连续存储空间的查找表

3> 哈希函数：哈希函数是根据关键字确定存储位置的函数

4> 哈希冲突：哈希冲突是不同关键字由哈希函数得到相同存储位置的现象

5> 构造哈希函数的方法的方法：

```
直接定址法
数字分析法
平方取中法
折叠法
除留余数法：取关键字被某个不大于哈希表表长m的数p除后所得余数为哈希地址的方法
随机数法
```

6> 哈希表表长的设定：一般取元素个数除以0.75之后的最大素数（质数）

7> 解决哈希冲突的方法：

```
开放定址法
    线性探测法
    二次探测法
    伪随机探测法
再哈希法
链地址法：将所有哈希函数值相同的记录存储在同一线性链表中
建立公共溢出区
```

8> 全部代码

```
1、hash.h
#ifndef __HASH_H__
#define __HASH_H__

#define N 13

//定义节点结构体类型
typedef struct Node
{
    int data;       //数据域
    struct Node *next;    //指针域
}Node;

//初始化哈希表
void hash_init(Node *hash[]);
```

```
16
17  //将数据存入哈希表
18  void hash_insert(Node *hash[], int key);
19
20  //输出哈希表内容
21  void hash_show(Node *hash[]);
22  //哈希查找
23  void hash_search(Node *hash[], int key);
24  #endif
25
26
27  2、hash.c
28  #include<stdio.h>
29  #include<stdlib.h>
30  #include"hash.h"
31
32
33  //初始化哈希表
34  void hash_init(Node *hash[])
35  {
36      for(int i=0; i<N; i++)
37      {
38          hash[i] = NULL;
39      }
40  }
41
42  //将数据存入哈希表
43  void hash_insert(Node *hash[], int key)
44  {
45      int pos = key%N;        //找到要存储的链表位置
46
47      //申请节点封装数据
48      Node *p = (Node *)malloc(sizeof(Node));
49      if(NULL==p)
50      {
51          printf("节点申请失败\n");
52          return ;
53      }
54      p->data = key;
55      p->next = NULL;
56
57      //头插法将数据放入表中
58      p->next = hash[pos];
59      hash[pos] = p;
60
```

```c
    printf("存入成功\n");

}

//输出哈希表内容
void hash_show(Node *hash[])
{
    //遍历每一条链表
    for(int i=0; i<N; i++)
    {
        printf("%d:", i);
        //定义遍历指针，将当前链表遍历
        Node *q = hash[i];
        while(q!=NULL)
        {
            printf("%d ---> ", q->data);
            q=q->next;
        }
        printf("NULL\n");
    }
}
//哈希查找
void hash_search(Node *hash[], int key)
{
    //通过哈希函数，确定要查找值所在的链表
    int pos = key%N;

    //定义遍历指针遍历当前链表
    Node *q = hash[pos];
    while(q!=NULL && q->data!=key)
    {
        q = q->next;
    }

    //判断q是否为空
    if(q == NULL)
    {
        printf("您要查找的数据不在表中\n");
    }else
    {
        printf("您要查找的数据在表中\n");
    }
}
```

```
3、main.c
#include"hash.h"
#include<stdio.h>
int main(int argc, const char *argv[])
{
    int arr[10] = {25,51,8,22,26,67,11,16,54,41};

    Node *hash[N];          //定义哈希表

    hash_init(hash);

    //循环调用函数插入
    for(int i=0; i<10; i++)
    {
        hash_insert(hash, arr[i]);
    }

    //调用输出哈希表函数
    hash_show(hash);

    //调用hash查找函数
    hash_search(hash, 26);
    hash_search(hash, 100);


    return 0;
}
```

作业

第一题：使用哈希存储将数据存入哈希表中，并进行查找

◆1、已知一组关键字序列为（25，51，8，22，26，67，11，16，54，41）

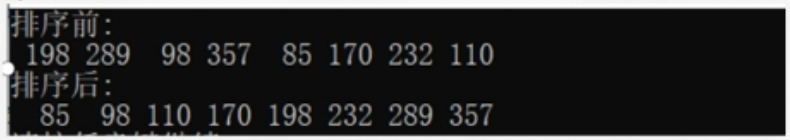
◆2、其哈希地址空间为[0,...,12]

◆3、Hash函数定义为：H(key) = key MOD 13

◆4、使用链地址法处理冲突，画出对应的哈希表，并编码实现

第二题：使用快速排序完成下面案例

■案例效果图



■案例要求

◆2）使用快速排序对序列{198，289，98，357，85，170，232，110}从小到大排序

◆3）排序后根据效果图正确输出

◆4）功能代码加入注释

◆5）分析时间复杂度

第三题：若已知一颗二叉树先序序列为ABCDEFG，中序序列为CBDAEGF画出该二叉树，并写出后序遍历顺序

四、图

4.1 定义

图是指由一个或多个数据元素及其关系构成的图形结构
注意：
1、数据元素至少有一个
2、数据元素之间的关系有0个或多个

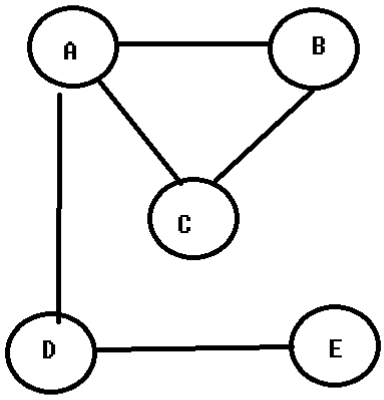
4.2 相关概念

顶点：是指图中的数据元素
边：图中两个顶点之间的关系
无向边：是指没有方向的边
有向边（弧）：是指有方向的边
入边：以当前顶点为终点的有向边
出边：以当前顶点为起点的有向边
弧头：有向图中箭头指向的顶点
弧尾：有向图中箭头起始位置的顶点
无向图：所有边均为无向边的图
有向图：所有边均为有向边的图
无向完全图：任意两个顶点之间都 存在边的无向图
有向完全图：任意两个顶点之间都存在方向相反的两条有向边的有向图
顶点的度：与顶点相连的所有边的数目
入度：有向图中以当前顶点为终点的所有边的数目
出度：有向图中以当前顶点为起点的所有边的数目
路径：从一个顶点到另一个顶点所经过的顶点序列
路径长度：路径上边的数目
子图：由顶点集和边集的子集组成的图

连通图：任意两个顶点之间 都有路径的无向图
强连通图：任意两个顶点之间都有往返路径的有向图

4.3 图的存储

1> 链式存储：邻接表
2> 邻接矩阵



图的链式存储：邻接表

```
A-->B-->C-->D
B-->A-->C
C-->a-->B
D-->A-->E
E-->D
```

图的顺序存储：邻接矩阵

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	1	0	0
C	1	1	0	0	0
D	1	0	0	0	1
E	0	0	0	1	0

4.4 邻接矩阵的结构体定义

一个顶点数组和一个存储关系的二维整形数组

```c
typedef char datatype;
#define N 5
typedef struct
{
    datatype data[N];          //存放定点数据
    int rel[N][N];            //存放任意两个顶点之间的关系
}Graph;
```

4.5 图的创建

```c
Graph *graph_create()
{
    Graph *G = (Graph*)malloc(sizeof(Graph));
    if(NULL==G)
    {
        printf("创建失败\n");
        return G;
    }
```

```
    //初始化顶点数组
    for(int i=0; i<N; i++)
    {
        G->data[i] = 'A'+i;
    }

    //初始化关系数组
    for(int i=0; i<N; i++)
    {
        for(int j=0; j<N; j++)
        {
            G->rel[i][j] = 0;
        }
    }

    printf("创建成功\n");
    return G;
}
```

4.6 添加关系

```
//定义获得顶点下标函数
int get_pos(Graph *G, datatype v)
{
    //判断逻辑
    if(NULL==G)
    {
        printf("获取失败\n");
        return -1;
    }

    //查找
    for(int i=0; i<N; i++)
    {
        if(G->data[i] == v)
        {
            return i;
        }
    }

    return -1;        //没有找到该顶点
}

```

```
23
24  //添加关系
25  void rel_add(Graph *G, datatype v1, datatype v2)
26  {
27      int pos1 = get_pos(G, v1);
28      int pos2 = get_pos(G, v2);
29
30      //判断逻辑
31      if(NULL==G || pos1==-1 || pos2==-1 || pos1==pos2)
32      {
33          printf("关系添加失败\n");
34          return ;
35      }
36
37      //将关系数组赋值为1
38      G->rel[pos1][pos2] = G->rel[pos2][pos1] = 1;
39      printf("%c--%c关系建立成功\n", v1, v2);
40  }
```

4.7 输出图

```
1   void graph_show(Graph *G)
2   {
3       //判断逻辑
4       if(NULL==G)
5       {
6           printf("输出失败\n");
7           return ;
8       }
9
10      //输出图
11      for(int i=0; i<N; i++)
12      {
13          printf("\t%c", G->data[i]);
14      }
15      printf("\n");
16
17      //输出关系
18      for(int i=0; i<N; i++)
19      {
20          printf("%c\t", G->data[i]);
21          for(int j=0; j<N; j++)
22          {
23              printf("%d\t", G->rel[i][j]);
```

```
        }
        printf("\n");
    }
}
```

4.8 全部代码

1> graph.h

```
#ifndef __GRAPH_H__
#define __GRAPH_H__

//定义图的结构体类型
typedef char datatype;
#define N 5
typedef struct
{
    datatype data[N];         //存放定点数据
    int rel[N][N];           //存放任意两个顶点之间的关系
}Graph;

//创建图
Graph *graph_create();


//定义获得顶点下标函数
int get_pos(Graph *G, datatype v);

//添加关系
void rel_add(Graph *G, datatype v1, datatype v2);

//输出图
void graph_show(Graph *G);


#endif
```

2> graph.c

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


//创建图
```

```c
Graph *graph_create()
{
    Graph *G = (Graph*)malloc(sizeof(Graph));
    if(NULL==G)
    {
        printf("创建失败\n");
        return G;
    }

    //初始化顶点数组
    for(int i=0; i<N; i++)
    {
        G->data[i] = 'A'+i;
    }

    //初始化关系数组
    for(int i=0; i<N; i++)
    {
        for(int j=0; j<N; j++)
        {
            G->rel[i][j] = 0;
        }
    }

    printf("创建成功\n");
    return G;
}

//定义获得顶点下标函数
int get_pos(Graph *G, datatype v)
{
    //判断逻辑
    if(NULL==G)
    {
        printf("获取失败\n");
        return -1;
    }

    //查找
    for(int i=0; i<N; i++)
    {
        if(G->data[i] == v)
        {
            return i;
        }
```

```c
    }

    return -1;        //没有找到该顶点
}


//添加关系
void rel_add(Graph *G, datatype v1, datatype v2)
{
    int pos1 = get_pos(G, v1);
    int pos2 = get_pos(G, v2);

    //判断逻辑
    if(NULL==G || pos1==-1 || pos2==-1 || pos1==pos2)
    {
        printf("关系添加失败\n");
        return ;
    }

    //将关系数组赋值为1
    G->rel[pos1][pos2] = G->rel[pos2][pos1] = 1;
    printf("%c--%c关系建立成功\n", v1, v2);
}

//输出图
void graph_show(Graph *G)
{
    //判断逻辑
    if(NULL==G)
    {
        printf("输出失败\n");
        return ;
    }

    //输出图
    for(int i=0; i<N; i++)
    {
        printf("\t%c", G->data[i]);
    }
    printf("\n");

    //输出关系
    for(int i=0; i<N; i++)
    {
        printf("%c\t", G->data[i]);
```

```
        for(int j=0; j<N; j++)
        {
            printf("%d\t", G->rel[i][j]);
        }
        printf("\n");
    }
}
```

3> main.c

```c
#include"graph.h"
#include<stdio.h>
int main(int argc, const char *argv[])
{
    Graph *G = graph_create();
    if(NULL==G)
    {
        return -1;
    }
    //调用输出图函数
    graph_show(G);

    rel_add(G, 'A', 'B');
    rel_add(G, 'A', 'C');
    rel_add(G, 'A', 'D');
    rel_add(G, 'B', 'C');
    rel_add(G, 'D', 'E');

    graph_show(G);

    return 0;
}
```