数据结构考试题（样例一）

一．选则题（5分）

1. 数据结构中，与所使用的计算机无关的是数据的 （3） 结构。

（1）存储　　（2）物理　　（3）逻辑　　（4）物理和存储

2. （4）在链表中进行比在顺序表中进行效率高。

（1）顺序查找　　（2）折半查找　　（3）分块查找　　（4）插入

3. 树结构最适合于用来表示 （3） 。

（1）有序数据　　（2）无序数据　　（3）元素间具有分支层次关系的数据

（4）元素间无关联的数据

4. 散列存储中，碰撞（或称冲突）指的是 （2） 。

（1）两个元素具有相同序号　　（2）不同的关键字对应于相同的存储地址

（3）两个记录的关键字相同　　（4）数据元素过多

5. n个结点的连通图至少有 （1） 条边。

（1）n-1　　（2）n　　（3）n(n-1)/2　　（4）2n

6. 直接选择排序的时间复杂性为 （4） （n为元素的个数）。

（1）O(n)　　（2）O(log2 n)　　（3） O(n log2 n)　　（4） O(n2)

二．判断下列各命题是否正确，若正确在（　）内打√，否则打×。（5分）

1. 使用双向链表存储数据，可以提高查找（定位运算）的速度。 （√）

2. B+ 树是一种特殊的二叉树。 （×）

3. 栈可视为一种特殊的线性表。 （√）

4. 最小生成树是边数最少的生成树。（√）

5. 快速排序总是比其它排序方法快。（×）

三．填空（10分）

1. 设二维数组A[10 . . 20, 5 . . 10] 按行优先存储，每个元素占 4 个单元，A[10, 5] 的地址是 2000，则A[15,10]的地址是 2140 。

2. 深度为k的满二叉树有 $2^k - 1$ 个结点。

3．设 s = ‘I AM A STUDENT’， t = ‘GOOD WORKER’，CONCAT(Substr(s，6，2)，t) = ‘A GOOD WORKER’。

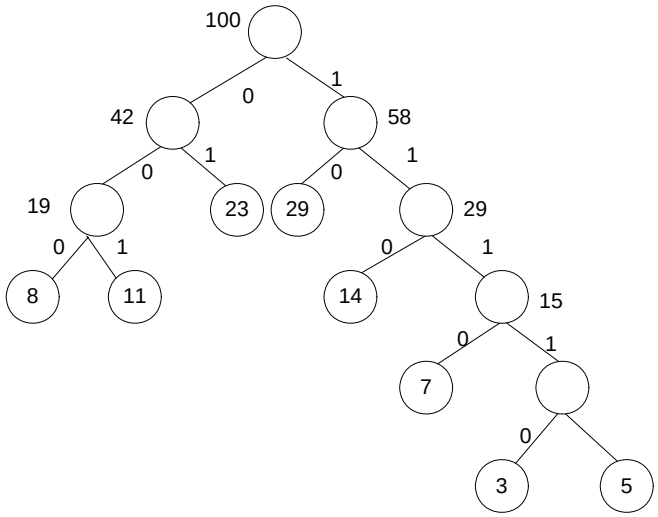
4. ISAM文件由 主索引，柱面索引，磁道索引和主文件组成。

5. 散列既是一种 存储 方式，又是一种查找方法。

四．应用题（40 分）

1.（15 分）已知某系统在通信联络中只可能出现 8 种字符。 其频率分别为 0.05, 0.29, 0.07, 0.08, 0.14, 0.23, 0.03, 0.11。试设计哈夫曼编码。

答案:



哈夫曼编码: 0.05：11111，0.29：10，0.07：1110，0.08：000，0.14：110，0.23：01，0.03：11110，0.11：001；

2.（15 分）试编写算法实现下述运算：①图的广度优先搜索；②图的连通分量计算。

答案:

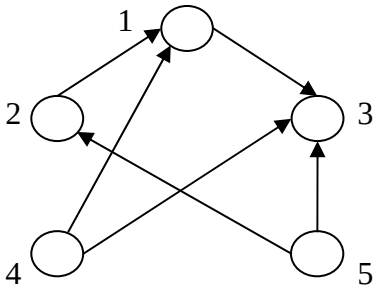
```
const    vnum =  …;      //图的顶点数
struct    graph
{      int vex[vnum];
       int arcs[vnum][vnum];
};    graph ga;

void    dfs (vexnode *g[], int v1)          //从 V1 出发深度优先遍历用链接表表示的图 g
{      edgenode *p;
       bool visited[n];          //访问标记数组，n 为顶点数
       cout<<v1;
       visited[v1] = true;       //标志 V1 访问
       p = g[v1]→link;           //找 V1 的第一个邻接点
       while (p != NULL)
       {      if (!(visited [p→vertex])    dfs(g, p→vertex);
              p = p→next;        //回溯 - 找 V1 邻接点
       }
}
```

在广度优先搜索中，若对 x 的访问先于 y，则对 x 邻接点的访问也先于对 y 邻接点的访问。因此，可采用队列来暂存那些刚访问过，但可能还有未访问的邻接点的顶点。

```
void    bfs (vexnode *g[], int v1)
{  //以邻接表为存储结构的广度优先搜索。q 为队列，假设 visited 的各分量已置为 false}
      int v;
      init_linkedque (q);       //设置一个空队 q
      visited[v1] = true;
      cout<<v1;
      in_linkedque (q, v1);     //v1 入队
      while (!(empty (q))
      {     v = out_linkedeque (q);       //出链队列
            p = g[v]→link;
            while (p != NULL)
            {     if (!visited[p→vertex])
                  {     visited [p→vertex] = true;
                        cout<<p→vertex<<" ";
                        in _linkedque (q, p→vertex);
                  }
                  p = p→next;
            }
    }
}
```

如果要遍历一个非连通图，则需多次调用 dfs 或 bfs。每一次都得到一个连通分量；调用 dfs 或 bfs 的次数就是连通分量的个数。

3.（10 分）给出下图的所有拓扑排序序列。



答案：

④→⑤→②→①→③；

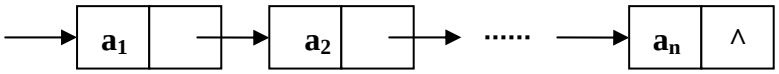
⑤→④→②→①→③；

⑤→②→④→①→③。

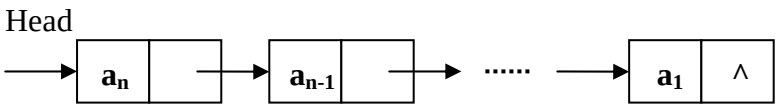
五．设计题（40 分）

1.（10 分）写一个算法，借助栈将一个单链表倒置（可以用栈的基本运算）。也即，将下面的单链表：

Head



倒置成:



答案：无标准答案，评分标准参照如下两点要求:

1）正确地写出了把单链表中各元素依次取出，用 PUSH 操作进行压栈处理给 5 分;

2)正确地写出了用 POP 操作把栈中各元素依次取出，并正确放入单链表给 5 分;

2.（15 分）试编写选择排序算法实现直接选择排序。

答案：直接选择排序是一种很简单的排序方法，它的做法是：首先在所有的记录中选出键值最小的记录，把它与第一个记录交换;然后在其余的记录中再选出键值最小的记录与第二个记录交换;依次类推，直至所有记录排序完成。在第 i 趟中，通过 n - i 次键值比较选出所需记录。

```
void    select (records r[n+1], int n)
{       //函数 swap(r[k], r[i])的功能是交换 r[k]和 r[i]记录的位置
        int i, j, k;
        for (i = 1; i < n; i++)       //每次循环，选择出一个最小键值
        {     k = i;
              for (j = i+1; j <= n; j++)
              if (r[j].key < r[k].key)    k = j;
              if (k != i)    swap(r[k], r[i]);
        }
}
```

3.（15 分）试编写选择排序算法实现堆排序。

答案：堆排序是树形选择排序的进一步改进。首先给出堆的定义：堆是一键值序列 $\{k_1,k_2\cdots\cdots,k_n\}$，对所有i = 1，2……[n/2] 满足; $\begin{cases} ki \le k2i \\ ki \le k2i+1 \end{cases}$ 。堆排序的基本思想是：对一组待排序记录的键值，首先把它们按堆的定义排成一个堆，这称为建堆。这就找到最小的键值。然后将最小键值取出，用剩下的键值再建堆，便得到次最小的键值。如此反复进行，直到最大健值。从而将全部键值排好序为止。实现堆排序需要解决两个问题：1）如何由一个无序序列建成一个堆？；2）如何输出堆顶元素之后，调整剩余元素成为一个新堆?。

第一问题的基本方法是：首先将要排序的所有键值分放到一棵完全二叉树的

各个结点中（这时的完全二叉树并不一定具备堆的特性）。显然，所有 i>[n/2]的结点 ki 都没有孩子结点。因此，以这样的 ki 为根的子树已经是堆。然后从 i=[n/2]的结点 ki 开始，逐步把以 i=[n/2]，[n/2]-1……为根的子树排成堆，直到以 k1 为根的树排成堆，就完成了建堆过程，实际上，即从第[n/2]个元素开始筛选。

第二个问题的基本方法是：在输出堆顶元素之后，以堆中最后一个元素替代之。此时根结点的左、右子树均为堆，则需要自上而下进行调整，得到一个新堆。具体做法是：首先堆顶元素和其左、右子树根结点的值比较，由于右子树根结点的值小于左子树根结点的值，且小于根结点的值，则将右子树根结点与二叉树的根结点交换;由于交换后破坏了右子树的堆，则需进行和上述相同的调整，直至叶子结点。此时，调整后的堆顶为 n-1 个元素中的最小值。重复上述过程（称为“筛选”），便可得有序的输出。

```
void    sift (records r[n+1], int k, int m)
{       //假设 r[k]...r[m]是以 r[k]为根的完全二叉树，且分别以 r[2k]和 r[2k+1]为根的左、右子树满足堆的性质，该算法调整 r[k]使整个序列 r[k]...r[m]满足堆的性质
        int i, j, x;
        bool finished;
        records t;
        i = k, j = 2*i;
        x = r[k].key;
        finished = false;
        t = r[k];               //暂存“根”记录 r[k]
        while ((j <= m) && (! finished))
        {     if ((j < m) && (r[j].key > r[j+1].key))    j++;
              //若存在右子树，且右子树根的关键字小，则沿右分支筛选
              if (x <= r[j].key)    finished = true;       //筛选完毕
              else     {     r[i] = r[j];    i = j;    j = 2*i;     }
        }
        r[i] = t;     //r[k]填入恰当位置
}

void    heapsort (records r[n+1])
{       //对 r[1] 到 r[n]进行堆排序，排序完成后，r 中记录按关键字自大至小有序排列
        int i;
        for (i = n / 2; i >= 1; i--)
        sift (r, i, n);          //自第[ n/2]个记录开始进行筛选建堆
        for(i = n; i >= 2; i--)
        {     swap (r[1], r[i]);       //将堆顶记录和堆中最后一个记录互换
              sift (r, 1, i -1)   //调整 r[1], 使 r[1]到[i-1]变成堆
```