# 北京大学信息科学技术学院

# 2007－2008 秋季学期

## 数据结构与算法 A 参考答案

## 任课教师： 张铭、赵海燕、王腾蛟、宋国杰

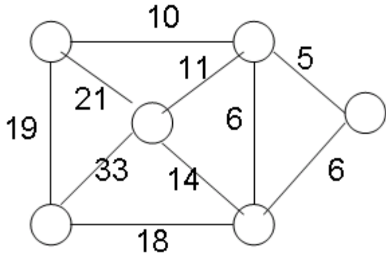
一、填空

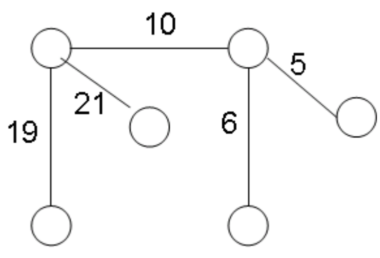
1. D
2. 3
3. (1) 1234, 1324, 2134 (2) 添加弧$v_3v_2$ 或$v_3v_2$
4. ACD
5. {60，51，121，142，15，575，185，46，57，168，89}
6. $\lceil n/m \rceil$
7. $\sum_{i=1}^{n} i * \frac{1}{2^i} = 2 - \frac{n+2}{2^n}$
8. 4M/(4+4)= 512K，30*512K=15M

二．简答题：

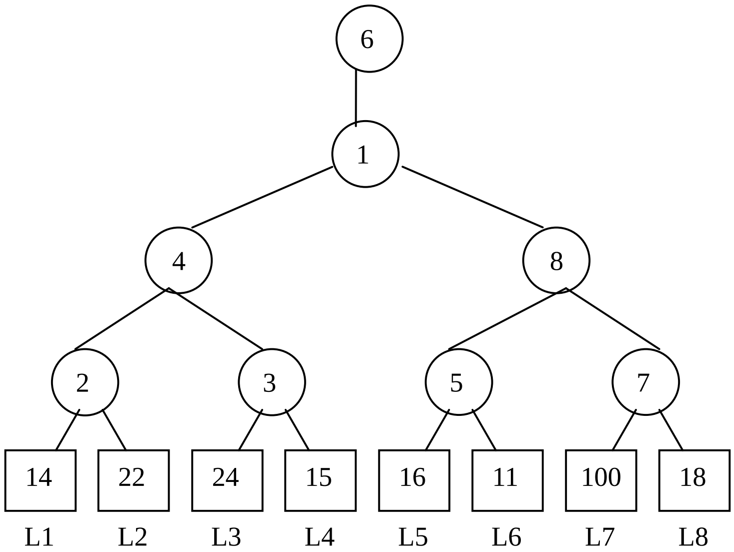
1．Dijkstra最短路径算法对于已知图G=(V,E)，给定源顶点s∈V，找出s到图中其他各顶点的最短路径。而如果此图为连通图。根据Dijkstra算法，对于任意一个顶点s，可以找到其他每个顶点到s的最短路径，查找过程中将所有顶点分成第一组和第二组，第一组为跟s相关联的顶点，对于每次新加入第一组的顶点Vi，在处理过程中将中间顶点Vm设为其父结点，并将s到Vi的最短路径纪录下来，这样查找完毕时就会发现，除了s没有父结点外，其他每个结点都有且仅有一个父结点。这便构成了一棵支撑树。因此对于一个连通图来说，Dijkstra最短路径算法可以给出一棵支撑树。

但是 Dijkstra 算法不能保证得到一棵最小支撑树（MST)。(举反例）对图 1，对于左上角的顶点 s 找到的最短路径如图 2，总权值为 19+21+10+6+5=61，而该图的最小支撑树其权值为 10+11+5+6+18=50。所以 Dijkstra 最短路径算法不一定能够得到最小支撑树。

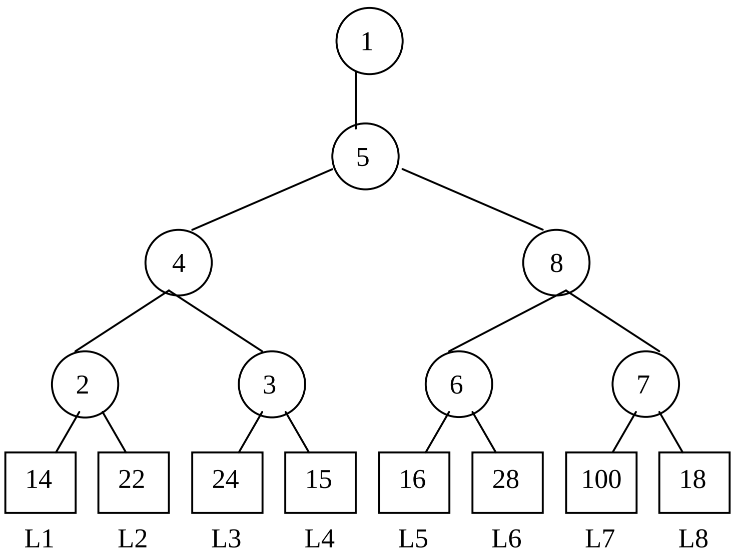



2. 初始：




重构：




3.
倒排索引中的记录地址可以是记录的实际存放地址，也可以是记录的关键码。试比较这两种方式的优缺点。

答案：

在倒排索引中的记录地址用记录的实际存放地址，搜索的速度快；但以后在文件中插入或删除记录对象时需要移动文件中的记录对象，从而改变记录的实际存放地址，这将对所有的索引产生影响：修改所有倒排索引的指针，不但工作量大而且容易引入新的错误或遗漏，使得系统不易维护。

记录地址采用记录的关键码，缺点是寻找实际记录对象需要再经过主索引，降低了搜索速度；但以后在文件中插入或删除记录对象时，如果移动文件中的记录对象，导致许多记录对象

的实际存放地址发生变化，只需改变主索引中的相应记录地址，其他倒排索引中的指针一律不变，使得系统容易维护，且不易产生新的错误和遗漏。

3．（1）散列表：

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 78 | | 15 | 03 | | 57 | 45 | 20 | 32 | 33 | 23 | | 12 |

搜索成功的平均搜索长度为：

ASLsucc = (1 + 1 + 1 + 1 + 1 + 1 + 3 + 1 + 3+ 1)/10 = 14/10=1.4

搜索不成功的平均搜索长度为：

ASLunsucc = (2 + 1 + 3 + 2 + 1 + 7 + 6 + 5+ 4 + 3 + 2 + 1 + 3)/13 = 40/13=3.07

（2）散列表：

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 78 | | 15 | 03 | | 57 | 45 | 20 | 33 | | 23 | 32 | 12 |

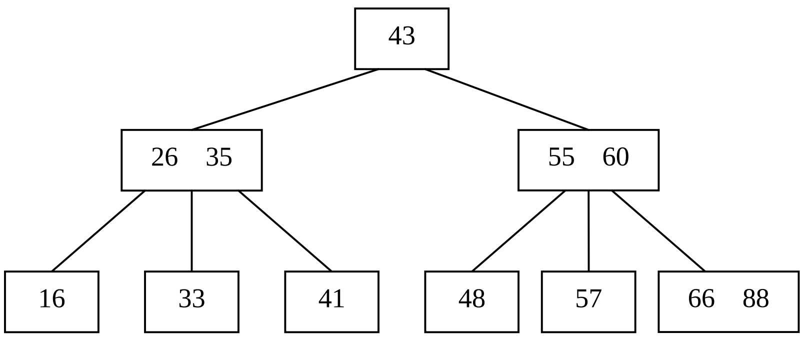
搜索成功的平均搜索长度为：

ASLsucc = (1 + 1 + 1 + 1 + 1 + 1 + 2 + 1 + 2+ 1) = 12/10=1.2

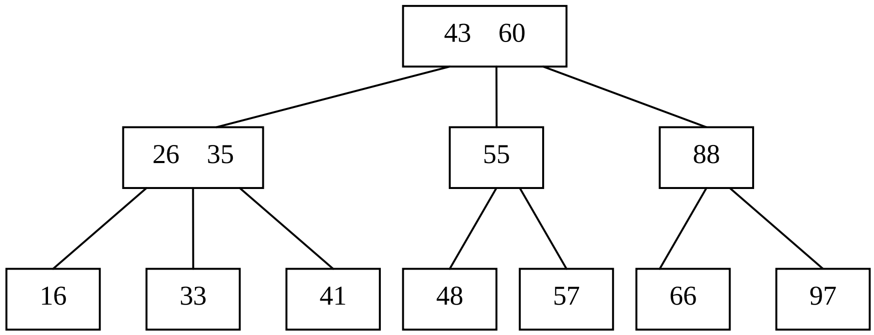
4. 设有 3 阶 B 树如下图所示：

（1） 在该 B 树上插入关键字 97，画出插入后的 B 树。

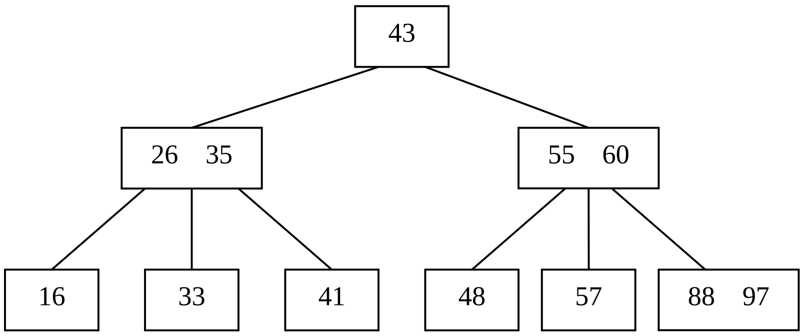
（2） 在（1）得到的 B 树基础上删除 66，画出删除后的 B 树。




答案：（1）




（2）




三．算法填空评分标准

（1）s == t || a[i + 1] >= b[j] && b[j + 1] >= a[i]　// 写错一个条件扣 1 分

（2）search(a, b, i, t)
（3）search(a, b, s, i);　　　　// （2）（3）即使有错。但有折半查找思想的给一半分数
（4）a[0] > b[n - 1]　　　　　　// 判断A[n-1]或者B[n-1]是否
（5）b[0] > a[n - 1]

四、算法设计

1. 评分标准

a) 算法思想 ＋ 注释 占 2 + 1 分

b) 代码框架 7 分，按照正确比例给成绩

思想：深度优先，回溯。

```
int visited[MAXSIZE];                       // 初始化为 0
int GetPathNum_Len(Graph& G, int i, int j, int len)    {
    if (i == j && len == 0) return 1;        // 找到了一条路径,且长度符合要求
    sum = 0;    //sum 表示通过本结点的路径数
    visited[i] = 1;
    for (Edge e = G.FirstEdge(i); G.IsEdge(e); e = G.NextEdge(e)) {
        int l = G.ToVertex(e);
```

```
        if (!visited[l]) sum += GetPathNum_Len(G, l, j, len - 1)    //剩余路径长度减一
    }//for
    visited[i] = 0;                          //本题允许曾经被访问过的结点出现在另一条路径中
    return sum;
} //GetPathNum_Len
```

2. 教材原题。

注意需要遍历整个探查序列，记录第一个墓碑位置。有重复码不插入，没有重复码则插入在第一个墓碑或者序列尾部。

```
template <class Key, class Elem, class KEComp, class EEComp>
bool hashdict<Key, Elem, KEComp, EEComp>::hashInsert(const Elem &e)
{
    int   home= h(getkey(e));        // home 记录基位置
    int   i=0, inplace;
    bool tomb_pos=FALSE;
    int   pos = home;                  //  探查序列的初始位置
    while(!EEComp::eq(EMPTY, HT[pos]))
    {
        if(EEComp::eq(e, HT[pos]))
            return false;
        if (EEComp::eq(TOMB, HT[pos])&& !tomb_pos )
        {
            inplace=pos;          //记录第一个墓碑的位置
            tomb_pos=TRUE;
        }
        i++;
        pos = (home + p( getkey(e) , i ) ) % M;
    }
    if( !tomb_pos )
        inplace = pos;
    H[inplace] = e;
    return TRUE;
}
```

3. 评分标准（总分 10 分）

   c) 算法思想 + 注释 2 + 1 分

   d) 代码框架 7 分 ，其中

      1) 增量序列的生成 3 分

      2) 之后对按照不同增量对原始序列分成的若干子序列进行排序 4 分

参考代码如下：

```
// Shell 排序，Array[]为待排序数组，n 为数组长度
template <class Record>
void ShellSort(Record Array[], int n)   {
    int i, j, k, h, hCnt;
    int   increments[] = new int[20];
    Record tmp;

    for (h = 1, i = 0; h < n;   i++) {      //  建立增量序列，存放在数组中
```

```
            increments[i] = h;
            h = 3*h + 1;
        }
        for (i--;    i >= 0;    i--) {                  //  对不同的增量值循环，
            h = increments[i];
            for (hCnt = h; hCnt < 2*h; hCnt++) {    //  对第 i 趟的各个小序列排序
                for (j = hCnt ; j < n; ) {                  //  采用直接插入排序，此是
                    tmp = Array[j];
                    k = j;
                    while (k-h >= 0 && tmp <Array[k-h]) {
                        Array[k] = Array[k-h];
                        k -= h;
                    }
                    Array[k] = tmp;
                     j += h;
                }
            }
        }
}
```