一、（1）简述线性表的两种存储结构的主要优缺点及各自适用的场合。

（2）在折半查找和表插入排序中，记录分别应使用哪种存储结构，并用一句话简述理由。

答：（1）顺序存储是按索引（如数组下标）来存取数据元素，优点是可以实现快速的随机存取，缺点是插入与删除操作将引起元素移动，降低效率。对于链式存储，元素存储采取动态分配，利用率高。缺点是须增设指针域，存储数据元素不如顺序存储方便。优点是插入与删除操作简单，只须修改指针域。

（2）在折半查找中，记录使用顺序存储，可以快速实现中点的定位；在表插入排序中，记录使用静态链表，可以降少移动记录的操作。

二、设 T 是一棵具有 n 个节点的二叉树，若给定二叉树 T 的先序序列和中序序列，并假设 T 的先序序列和中序序列分别放在数组 PreOrder[1..n]和 InOrder[1..n]中，设计一个构造二叉树 T 的链式存储结构的算法。以下为结点类型：

```
typedef struct BiTNode{
    TElemType data;
    Struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;
```

解：设 T 的先序序列和中序序列分别放在数组 PreOrder[1..n]和 InOrder[1..n]中，根据先序遍历的特点可知，PreOrder[1]为根节点，设中序序列中 InOrder[i]=PreOrder[1]，则在 InOrder[i]的左面的 InOrder[1..i-1]应为二叉树的左子树，而 InOrder[i+1..n]为根的右子树。相对应的是，在先序序列中根的左子树应为 PreOrder[2..i]，而右子树为 PreOrder[i+1..n]。而左子树的根为 PreOrder[2]，右子树的根为 PreOrder[i+1]。依次递归，用同样的方法可以确定子树的左子树和右子树，从而逐步确定整个二叉树。

```
typedef struct BiTNode{
    TElemType data;
    Struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;

void Create(BiTree T, TElemType PreOrder[], int i1, int j1, TElemType InOrder[], int i2, int j2)
{
    int i=i2;
    if (i1<=j1){
        T=(BiTree) malloc(sizeof(BiNode));
        T→data=PreOrder[i1];
        T→lchild=NULL；
        T→rchild=NULL;
        while (InOrder[i]!=PreOrder[i1]) i++;
        Create(T→lchild, PreOrder, i1+1, i-i2+i1, InOrder, i2, i-1);
        Create(T→rchild, PreOrder, i-i2+i1+1, j1, InOrder, i+1, j2);
```

```
	}
	else T=NULL;
}
```

三、用孩子兄弟链表作为树的存储结构，请编写算法计算树的深度。

解：算法思路：一棵树的深度可以递归定义为：若树为空，则深度为 0，否则树的深度为根结点的所有子树深度的最大值加 1。

数据结构为：

```
typedef struct CSNode{
	ElemType data;
	struct CSNode *firstchild, * nextsibling;
} CSNode, *CSTree;
```

算法如下：

```
int depth(CSNode * t)
{
	CSNode *p; int m, d;
	if (t==NULL) return 0;
	p=t→firstchild; m=0;
	while (p) {
		d=depth(p);
		if (d>m) m=d;
		p=p→nextsibling;
	}
	return m+1;
}
```

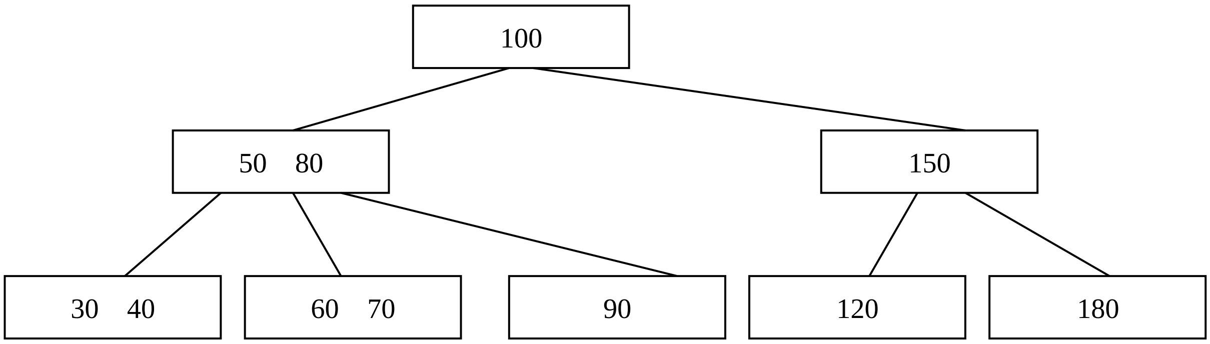
四、设计一个算法，判断无向图 G（图中有 n 个顶点）是否是一棵树。

解：算法思路：从第 v 个顶点出发，对图进行深度优先搜索。若在算法结束之前，又访问了某一已访问过的顶点，则图 G 中必定存在环，G 不是一棵以 v 为根的树。若在算法结束之后，所访问的顶点数小于图的顶点个数 n，则图 G 不是连通图，G 也不是一棵以 v 为根的树。

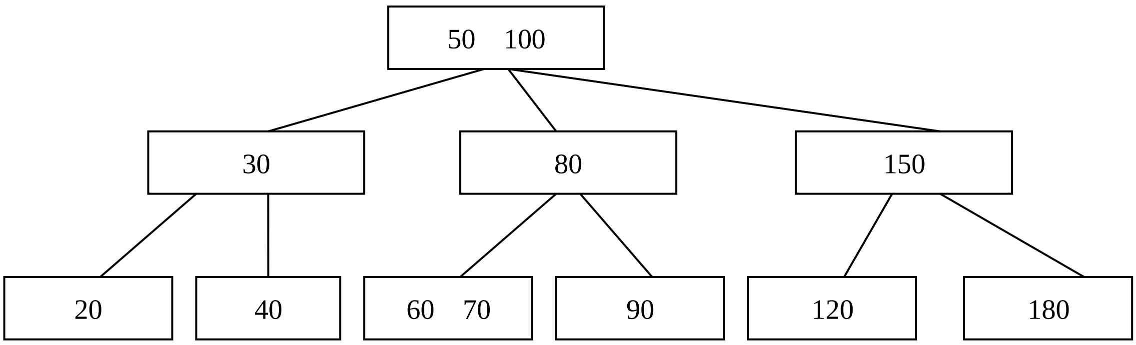
```
Boolean visited[MAX] ;                    //用于标识结点是否已被访问过
Status ( * VisitFunc) (int v);    //函数变量
void DFSTraverse( Graph G, Status ( * VisitFunc) (int v));
  { VisitFunc = Visit;
      for ( v=0; v <G.vexnum; ++v )    visited[v] = false;
      if (DFS(G, v )==FALSE) return FALSE;
      for ( v=0; v <G.vexnum; ++v )
          if (visited[v] == false) return FALSE;
      return OK;
    }
 Status DFS( Graph G,    int v );
 { Visited[v] = true;    VisitFunc(v);
      for ( w = FirstAdjVex(G, v) ; w ; w = NextAdjVex(G, v, w) )
```

```
                if (Visited[w]) return FALSE;
                    else DFS(G, w );
        return OK;
    }
```

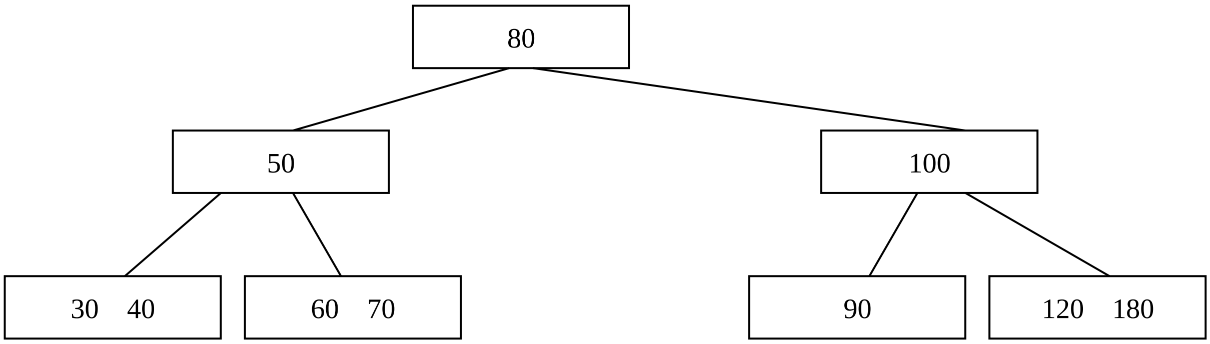
五、设有 3 阶 B一树，如下图所示，分别画出在该树插入关键字 20 和在原树删除关键字 150 得到的 B一树。



解：插入 20 后的 B一树为：



删除 150 后的 B一树为：



六、有一种简单的排序算法，叫做计数排序。这种排序算法对一个待排序的表进行排序，并将排序结果存放到另一个新的表中。必须注意的是，表中所有待排序的关键字互不相同。计数排序算法针对表中的每个记录，扫描待排序的表一趟，统计表中有多少个记录的关键字比

该记录的关键字要小。假设针对某一个记录，统计出的计算值为 c，那么这个记录在新的有序表中的合适的存放位置为 c+1。

（1） 编写实现计数排序的算法；

（2） 分析该算法的时间复杂性。

解：（1）假设数据结构如下：

```
#define MAXSIZE    20
typedef int KeyType；
typedef struct {
    KeyType    key;
    InfoType otherinfo;
} RedType;
typedef struct {
    RedType      r [MAXSIZE + 1 ] ; // r[0]  空或作哨兵
    int                        length;
} SqList;

void CountSort(SqList L1, SqList L2)
{//把 L1 计数排序后，结果放在 L2
     int i, j, n, count;
     n=L1.length;
     L2.length=L1.length;
     for (i=1; i<=n; i++){
          count=0;
          for (j=1; j<=n; j++)
               if (L1.r[j]<L1.r[i]) count++;
          L2.r[count+1]=L1.r[i];
     }
}
```

（2）基本操作是关键字比较操作和记录移动操作。其中关键字比较操作为 $O(n^2)$，记录移动操作为 O(n)。因此，总的时间复杂性为 $O(n^2)$。