第 1 章 绪论

一、基础题

1. A　　2. C　　3. C　　4. A　　5. C

二、扩展题

1. 数据是计算机加工处理的对象；数据元素是数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理；数据项是组成数据元素的、不可分割的最小单位。

2. 数据结构是按某种逻辑关系组织起来的数据元素的集合，使用计算机语言描述并按一定的存储方式存储在计算机中，并在其上定义了一组运算。

3. 集合结构、线性结构、树形结构和图形结构。集合结构中，元素之间没有关系；线性结构中，元素之间存在一对一的关系；树形结构中，元素之间存在一对多的关系，其中最多只有一个元素没有前驱元素，这个元素就是根；图形结构中，元素之间存在多对多的关系。

4. 顺序存储、链式存储、索引存储和散列存储。

5. 一个算法是对特定问题的求解步骤的一种描述，是指令的有限序列。其特征包括：

- 输入：算法有零个或多个输入
- 输出：算法至少产生一个输出
- 确定性：算法的每一条指令都有确切的定义，没有二义性。
- 能行性/可行性：可以通过已经实现的基本运算执行有限次来实现
- 有穷性：算法必须总能在执行有限步之后终止

6. 联系：程序是计算机指令的有序集合，是算法用某种程序设计语言的表述，是算法在计算机上的具体实现。

区别：在语言描述上不同，程序必须是用规定的程序设计语言来写，而算法的描述形式包括自然语言、伪代码、流程图和程序语言等；算法所描述的步骤一定是有限的，而程序可以无限地执行下去，比如一个死循环可以称为程序，但不能称为算法。

7.

正确性：算法的执行结果应当满足功能需求，无语法错误，无逻辑错误
简明性：思路清晰、层次分明、易读易懂，有利于调试维护
健壮性：当输入不合法数据时，应能做适当处理，不至于引起严重后果
效率：有效使用存储空间和有高的时间效率
最优性：解决同一个问题可能有多种算法，应进行比较，选择最佳算法
可使用性：用户友好性

8

(1)　执行次数为 **n**，时间复杂度为O(n)。

(2) 执行次数为 $\lceil \boldsymbol{log_3 n} \rceil$；时间复杂度为$O(logn)$

(3)　执行次数为$\boldsymbol{n^2}$；时间复杂度为$O(\boldsymbol{n^2})$

(4) 执行次数为$\lfloor \sqrt{n} \rfloor + 1$；时间复杂度为$O(\sqrt{n})$

第 2 章　线性表

一、基础题

1．A

2．D

3．B

4．C

5．B

6．D

7．D

8．C

9．A

10．D

二、扩展题

1．编写程序实现对顺序表逆置。

```
void Invert(seqList *L)
{
      ElemType temp;
      int i;
      for ( i=0; i<L->n/2; i++)
      {
            temp =L->element[i];
            L->element[i] = L->element[L->n-i-1];
                  L->element[L->n-i-1] = temp;
      }
}
```

2．编写程序将有序递增的单链表中数据值在 a 到 b(a<=b)之间的元素删除。

```
Status DeleteAb(SingleList *L,ElemType a, ElemType b)
{
   Node *p,*q;
   if (!L->n) return ERROR;
   p=L->first;
   q=L->first;
    while (p && p->element<b)
   {
   if (p->element<a)
      {
       q=p;
        p=p->link;
      }
   else if (q==L->first)
         {
               q=p->link;
                first=q;
               L->free(p);
                L->n--;
                p=q;
```

```
        }
        else{
            q->link=p->link;
            free( p);
            L->n--;
            p=q->link;
        }
    }
return OK;
}
```

3．编写程序删除单链表中所有关键字值为 x 的元素。

```
Status DeleteX( SingleList *L,ElemType x)
{
  Node *q,*p, *temp;
  if (!L->n) return ERROR;
  q=NULL;
  p=L->first;
  while(p!=NULL)
  {
     if (p->element!=x)
     {
       q=p;
       p=p->link;
     }
     else
     {
       if (q==NULL)
       L->first=L->first->link; //删除的是头结点
       else
       {
         p=q->link;
         q->link=p->link; //从单链表中删除 p 所指向的结点
       }
       temp=p;
       p=p->link;
       free(temp);
       L->n--;
     }
  }
}
```

4．编写程序实现对单链表的逆置。

```
void invert(singleList *L)
{
    Node *p=L->first,*q;
    L->first=NULL;
```

```
  while (p)
 {
     q=p->link;
     p->link=L->first;
     L->first=p;
     p=q;
    }
}
```

5．编写程序实现将数据域值最小的元素放置在单链表的最前面。

```
Status Swap(SingleList *L)
{
Node *p,*pFront,*min,*minFront;
ElemType temp;
if(!L->first)
     return ERROR;
p= L->first->link;
pFront =NULL;
min = L->first;
minFront=NULL;
while(p)
{
     if(p->element<min->element)
         {
           min = p;
           minFront=pFront;
         }
     pFront=p;
     p = p->link;
}
if(min != L->first)
{
     minFront->link=min->link;
     min->link= L->first;
     L->first=min;
}
return OK;
}
```

第 3 章 堆栈和队列

一 基础题

1. D

2. ABC, ACB, BAC, BCA CBA

3 堆栈遵循先进后出原则，队列遵循先进先出原则，都属于线形结构

4 判断队列为空的条件：front==rear

判断队列为满的条件：(rear+1)%M == front

5 (1) 5+(3*2+3)/3，最多有 3 个

(2) 4+(2^(4*1-1*3))/(2*1)，最多有 5 个

6 (1) ab+cd+/ (2) b2^4a*c*-

(3) ac*bc2^/- (4) ab+c*def+/+

(5) ab+cd*e+*ac*-

7 递归算法是算法过程中存在调用自身的结构，递归算法的优点是程序非常简洁和清晰，且易于分析。但它的缺点是费时间、费空间。非递归算法的优缺点与之相反。

二 扩展题

1 (2)和(3)不能。对(2)中的 E，B，D 而言，E 最先出栈则表明，此时 B 和 D 均在栈中，由于，B 先于 D 进栈，所以应有 D 先出栈。同理(3)也不能。

(1)能，操作序列:push(A),pop(),push(B),pop(),push(C),pop(),push(D),pop(),push(E),pop()

(4)能，操作序列:push(A),push(B),push(C),push(D),push(E),pop() ,pop(),pop(),pop(),pop()

3 只需在教材给定的循环队列算法的基础上，通过修改数组下标即可实现，该题较为简单，此处略过。

4

```
void ReverseStack(Stack S){
    Queue Q;                    //定义队列 Q
    while(!IsEmpty(S))          //
        EnQueue(Q, pop(S));     //栈顶元素进队
    while(!IsEmpty(Q))
        Push(S, DeQueue(Q));    //队头元素进栈
}
```

5

(1) 对整数数组 A[n]设计递归算法求数组中的最大整数。

```
int max(int a[], int i, int n){
    int k;
    if (i<n){
        k=max(a, i+1,n);
        if (a[i]>k)  return a[i];
        else   return k;
    }
    else    return a[n];
}
```

(2)求数组中 n 个数的平均值

```
int avg(int a[], int n){
        if (n>0)
            return (a[n-1]+ avg(a, n-1)*(n-1))/n;
        return 0;
}
```

6 在线性表 L 中搜索关键字值为 x 的元素,未找到返回-1，否则返回 x 所在位置下标。

```
int find(SeqList *L, int i, int x){
        if(i>=L->n)     return -1;
        if(L->element[i]==x)     return i;
        else return find(L, i+1, x);
}
```

第四章 数组和字符串

一、基础题

1. B　　2. （勘误）没有正确选项，答案应该是228　　3. D　　4. C　　5. C

二、扩展题

1.

```
ADT SymMatrix{
数据：n*n的矩阵，其中下标为<i,j>的元素与下标为<j,i>的元素值相等
运算：
        CreateSymMatrix(A,n) 创建运算；
        DestroySymMatrix(A) 清除运算；
        Retrieve(A,i,j) 查询运算；
        Output(A) 输出运算；
        Copy(A) 复制运算；
}
```

2.

（1） 查找运算

```
Status Retrieve(TriangularMatrix A, int i, int j, int* x)
{
        if(i<0||j<0||i>=A.n||j>=A.n) return Error;
        if(i>=j) *x=A.array[i*(i+1)/2+j];
        else *x= A.c; //c为零元
        return OK;
}
```

（2） 赋值运算

```
Status Store(TriangularMatrix *A, int i, int j, ElemType x)
{
        if(i<0||j<0||i>=A.n||j>=A.n) return Error;
        if(i>=j) *(A->array+i*(i+1)/2+j);
        else else *x= A.c;//c为零元
        return OK;
}
```

（3） 输出运算

```
void Output(TriangularMatrix A)
{
    int i,j;
    for(int i=0;i<A.n;i++)
        for(int j=0;j<A.n;j++)
        {
            ElemType value;
            Retrive(A,i,j,&value);
            PrintElement(value,i,j);//PrintElement的实现取决于value的类型，
                                    //此处不详列
        }
}
```

3.

行三元组表为:

i	j	value
0	0	-1
0	6	9
1	1	2
1	4	5
2	2	3
3	6	12
4	5	11

列三元组表为:

i	j	value
0	0	-1
1	1	2
2	2	3
1	4	5
4	5	11
0	6	9
3	6	12

4.

	0	1	2	3	4	5	6
num	1	1	1	0	1	1	2
k	0	1	2	3	3	4	5

5. 稀疏矩阵的查找运算

```
Status RetrieveSparseMatrix(SparseMatrix A, int i, int j, ElemType *x)
{
        if(!A||i<0||j<0||i>=A.m||j>=A.n) return ERROR;//要判断A是否存在
        int k;
        for(k=0;k<A.t;k++)
        {
            If(A.table[k].row==i&& A.table[k].col==j)
            {
            *x = A.table[k].value;
                return OK;
            }

    }
    *x = ZERO;//ZERO是预定义好的零元
    return OK;
}
```

时间复杂度: O(ta), ta是矩阵A的非零元数量。

第 5 章 树和二叉树

一 基础题

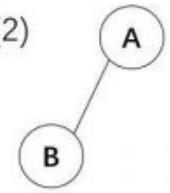
1 D	2 B	3 D	4 D	5 C
6 C	7 C	8 C	9 B	10 B
11 A	12 C			

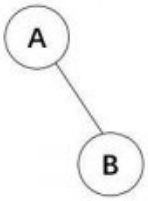
二 扩展题

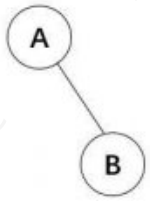
1 无序树 9 棵，有序树 12 棵，二叉树 30 棵

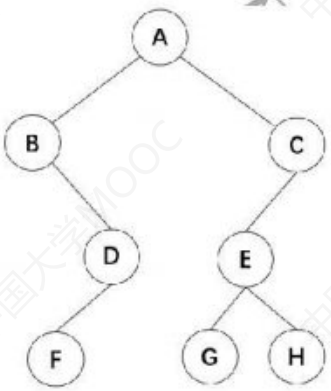
2. 举例表示：

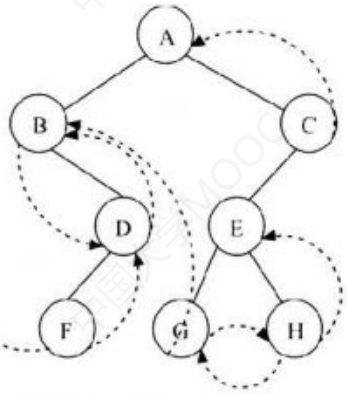
(1) 

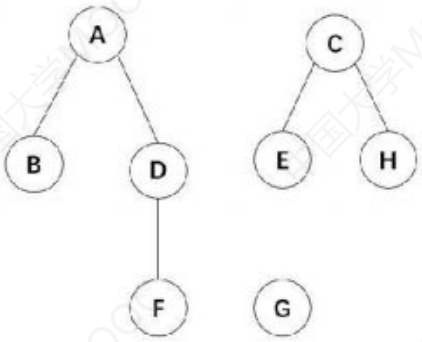
(2) 

(3) 

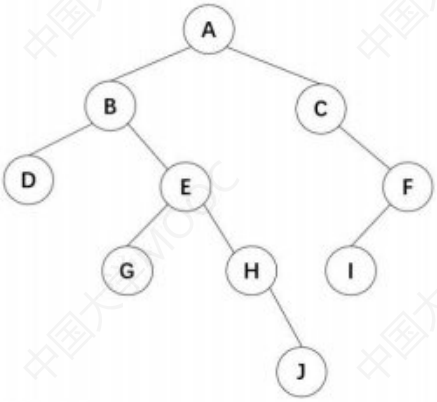
(4) 

3. (1) 

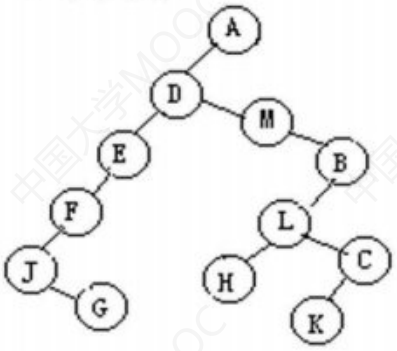
(2) 

(3) 

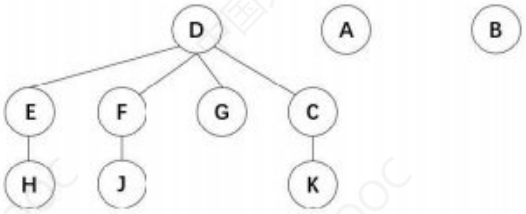
4.



5. (1)



(2)

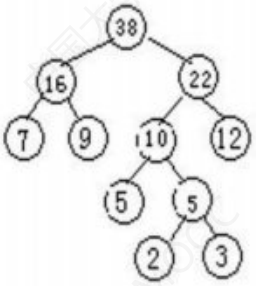


6. 由于哈夫曼树中只有度为 0 的叶子结点和度为 2 的结点，不妨设度为 2 的结点数量为 n_2，并假设树中结点的总结点数为 n，则有 $n=n_0+n_2$。又因为二叉树性质：任意二叉树中度为 0 的结点数 n_0 和度为 2 的结点数 n_2 间的关系是 $n_2=n_0-1$，则 $n=n_0+n_2=2n_0-1$。

7.（1):最大深度 6，最小深度 4　　(2):非叶结点数 5

8.

(1)



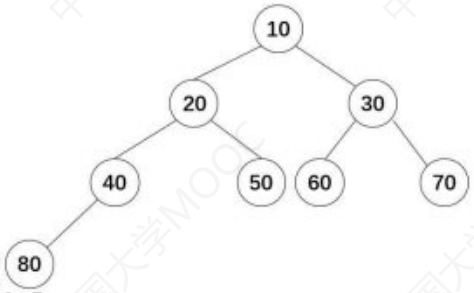
(2): 加权路径长度：WPL=(2+3)×4+5×3+(7+9+12)×2=91

(3): A: 1010　　B: 1011　　C: 100

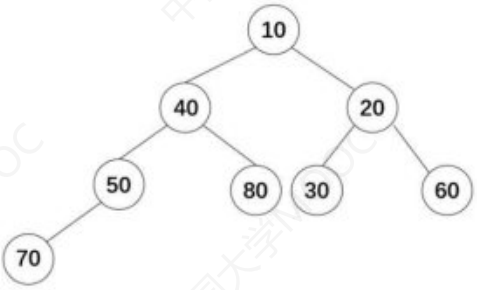
D: 00　　E: 01　　F: 11

9.

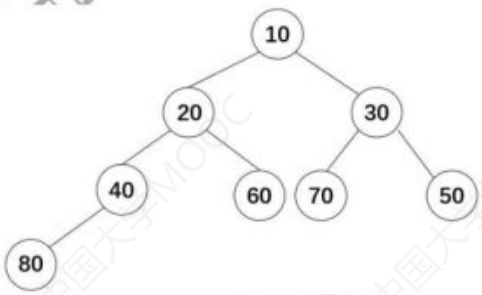
(1)



(2)

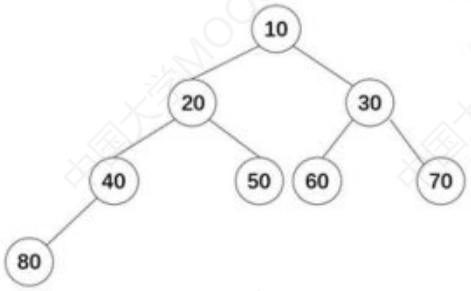


(3)

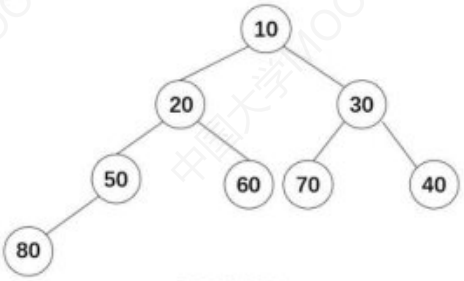


10.

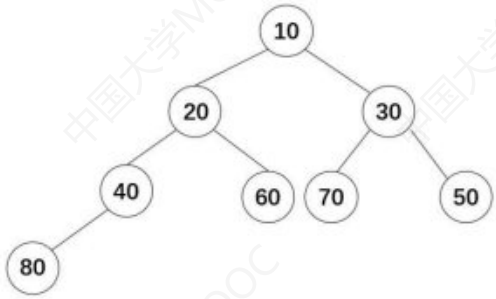
(1)



(2)



(3)



12. (1):

```
int   Height(BTNode *t) {       // 计算二叉树的高度
        if (!t)       return 0;     //边界条件
        int l = Height(t->lChild);
        int r = Height(t->rChild);
        return l>r ? (l + 1) : (r + 1);
}
```

(2):

```
void Map(BTNode *t, BTNode *arr[], int i) {
    if(!t)      return;
    arr[i]=t;
    Map(t->lChild, arr, 2*i+1);
    Map(t->rChild, arr, 2*i+2);
}
BOOL IsCompleteTree(BTNode *t){
    int  h=Height(t);
    int  i, n=1;     BOOL  flag = TRUE;
    for(i = 0; i<h; i++)  n=n*2;
    //可能的最多的节点数量为 2^n-1
    BTNode **arr = (BTNode **) malloc((n-1)*sizeof(BTNode *));
    for(i = 0; i++; i<n)    arr[i]=NULL
    Map(t, arr, 0);
    for(i=0; i<n; i++){
        if(!arr[i]&&flag)  flag = FALSE;
        if(arr[i]&&!flag)  return FALSE;
    }
    return TRUE;
}
```

(3):

```
//求内路径长度和外路径长度
void PL(BTNode *t, int *ipath, int *epath, int level)
{
    if(!t)     return;  //边界条件
    if(!t->lChild && !t->rChild)  //叶子结点，计入外路径长度
        *epath += level;
    else                  //非叶节点，计入内路径长度
        *ipath += level;
    PL(t->lChild, ipath, epath, level +1);  //递归计算左子树
    PL(t->rChild, ipath, epath, level +1);  //递归计算右子树
}
//ipath 和 epath 分别记录二叉树的内路径和外路径长度，并返回
void PathLength(BinaryTree *tree, int *ipath, int *epath)
{
    *ipath = *epath = 0;
    if(!tree)  return;
    int level=0;    //记录节点的路径长度
    PL(tree->root, ipath, epath, level);
}
```

第 6 章 集合和搜索

一、基础题

1.B　2.B　3.D　4.C　5.B

二、扩展题

1. 判断是否是递增

```
BOOL IsSorted(SeqList *L)
{
    BOOL flag = TRUE;
    for (int i = 1; i < L->n; i++)
        if (L->element[i] < L->element[i-1]) {
            flag = FALSE;
            break;
        }
    return flag;
}
```

2.

```
//求 A 和 B 的交集
void Intersection(SeqList *A, SeqList *B)
{
    int i, j;
    for(i=0;i<B->n;i++)
        for(j=0;j<A->n;j++)
            if(B->element[i]==A->element[j])
                printf("%d ", B->element[i])
}
```

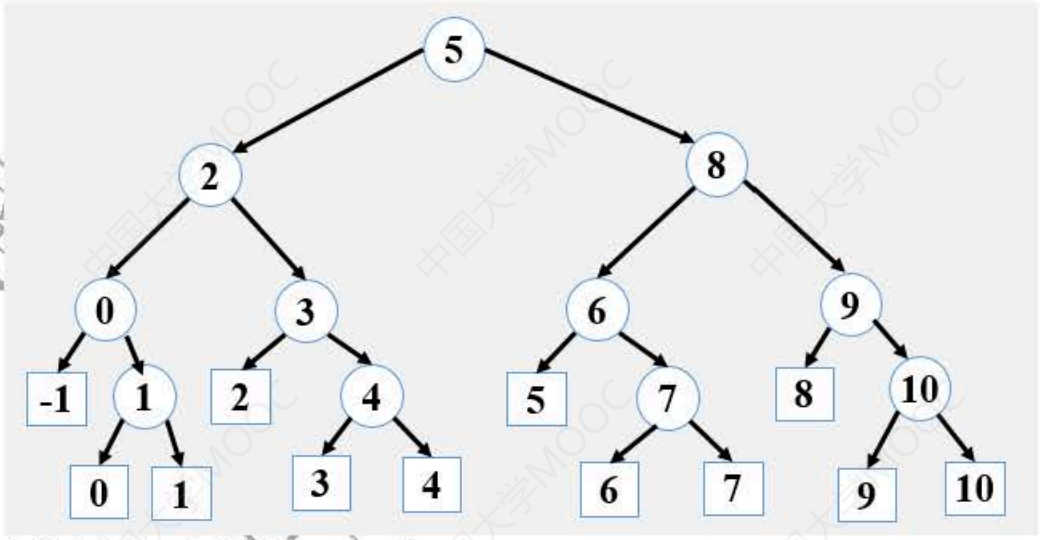
3. 查找 x 成功返回 TRUE，否则返回 FALSE

```
BOOL Search(LinkList *L, int x)
{
    Node* p = L->first;
    while (p) {
        if (p ->element == x)
            return TRUE;
        else if(p->element > x)
            break;
        p = p->next;
    }
    return FALSE;//查找失败
}
```

5.

(6,	17,	21,	27,	30,	36,	44,	55,	60,	67,	71)
0	1	2	3	4	5	6	7	8	9	10

二叉判定树:



查找成功的平均查找长度 ASL = (1 * 1 + 2 * 2 + 4 * 3 + 4 * 4) / 11 = 3

第 7 章 搜索树

一 基础题

1. B 解析：二叉树平衡因子可能为 1、0、-1.

2. C 解析：m-阶 B-树的概念，所有叶子节点在同一层。

3. C 解析：二叉搜索树（BST）的中序遍历升降有序，AVL 树是 BST 树。

4. B 解析：二叉搜索树只需要保证根节点比左子树都大，比右子树都小，有很多画法。

5. A 解析：高度最小的 BST 树应该是 AVL 树。

6. C 解析：一棵 m 阶 B 树中每个结点至多有 m 棵子树（即至多含有 m-1 个关键字）.

7. B 解析：考虑 BST 的极端情况，比如一右到底，故最坏情况下 BST 与线性表相同。

8. B 解析：AVL 属于 BST，完全二叉树当然不属于 BST，所以完全二叉树不一定是 AVL

9. B 解析：平衡因子为-1 的节点，新元素插入在其右子树上则不会引起平衡旋转；平衡因子为 1 的情况相反。

10. 12 解析：实际上可以用斐波那契数列推出来，层数为 1,2,3,4,5 时的最少节点数为：1,2,4,7,12.即是 F(N) = F(N-1) +F(N-2) +1，其中 N 为平衡二叉树层数。

二 扩展题

1、建立 37、45、91、25、14、76、56、65 为输入时的二叉搜索树，再从该树上依次删除 76、45，则树形分别如何？

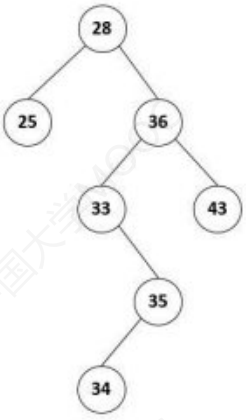
2、试写一个判定任意给定的二叉树是否是二叉搜索树的算法。

```
/**
* typedef int KeyType
* typedef struct entry{
*     KeyType Key;
*     ElemType Data;
* }Entry;
```

```
* typedef struct bstnode {
*     Entry Element;
*     struct bstnode *LChild, *RChild;
* }BSTNode, *BSTree;
 */
//参数 low、high 分别为当前结点元素的上限和下限
bool dfs(BSTree T, long low, long high) {
    if (!T) return true;
    long num = T->Element.Key;
    if (num <= low || num >= high) return false;
    return dfs(T-> LChild, low, num) && dfs(T-> RChild, num, high);
}

bool isValidBST(BSTree T){
    return dfs(T, LONG_MIN, LONG_MAX);
}
```

3、已知一棵二叉搜索树的先序遍历序列是 28,25,36,33,35,34,43，请画出此二叉搜索树。



4、编写一个从二叉搜索树中删除最大元素的算法，分析算法的时间复杂度。

```
/**
* typedef int KeyType
* typedef struct entry{
*     KeyType Key;
*     ElemType Data;
* }Entry;
* typedef struct bstnode {
*     Entry Element;
*     struct bstnode *LChild, *RChild;
* }BSTNode, *BSTree;
 */
bool DeleteMax(BSTree &T){
    BSTNode * p = T, *q = p;
    if(!p) return false;
```

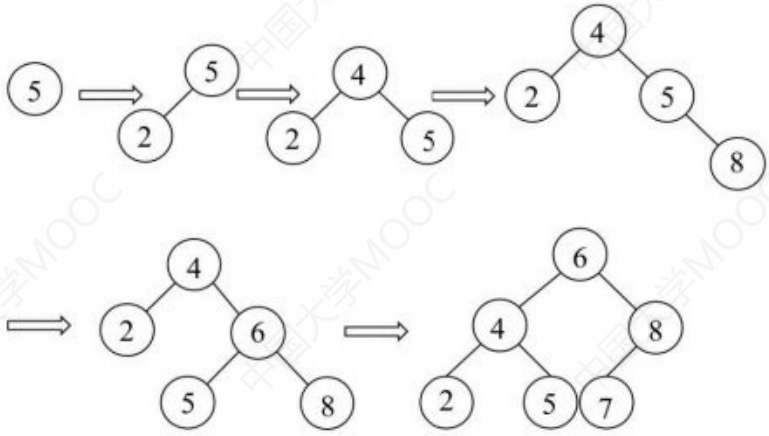
```
    while(p-> RChild){
        q=p;
        p=p-> RChild;
    }
    if(p==T){
        T=T->LChild;
    }
    else{
        q-> RChild =p-> LChild;
    }
    return true;
}
```

时间复杂度：平均 $O(log_2 n)$，最坏 $O(n)$

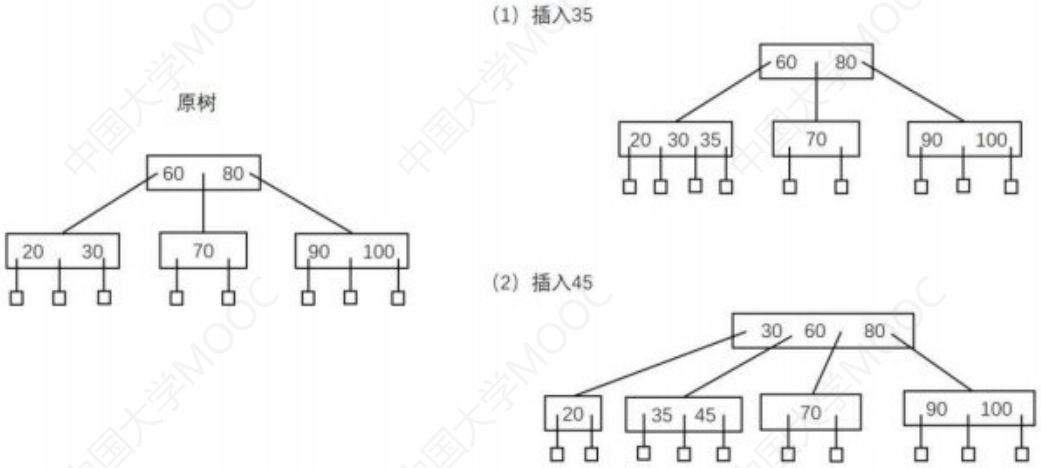
5、编写一个递归算法，实现在一棵二叉搜索树上插入一个元素。

```
/**
* typedef int KeyType
* typedef struct entry{
*     KeyType Key;
*     ElemType Data;
* }Entry;
* typedef struct bstnode {
*     Entry Element;
*     struct bstnode *LChild, *RChild;
* }BSTNode, *BSTree;
 */
BSTNode * insertIntoBST(TreeNode* root, Entry e)
{
    KeyType k = e.Key;
    if(!root)       //找到了插入位置，新建结点，初始化结点信息，插入二叉搜索树
    {
        BSTNode *p=( BSTNode *)malloc(sizeof(BSTNode));
        p->LChild=NULL;
        p->RChild=NULL;
        p->Element.Key=k;
        return p;
    }
    if(k >root->Element.Key)      //新节点应该插入在根节点的右子树上
        root->RChild=insertIntoBST(root->RChild, e);
    else if(k<root->Element.Key) //新节点应该插入在根节点的左子树上
        root->LChild =insertIntoBST(root->LChild, e);
    return root;
}
```

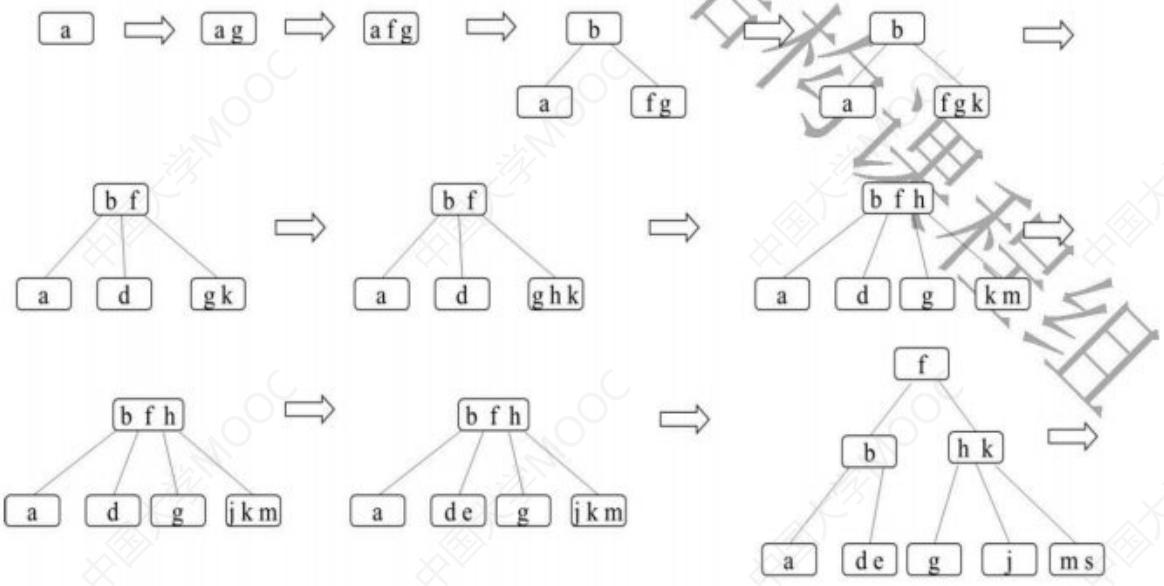
7、向空的 AVL 树中依次插入关键字 5、2、4、8、6 和 7，画出最终生成的 AVL 树。

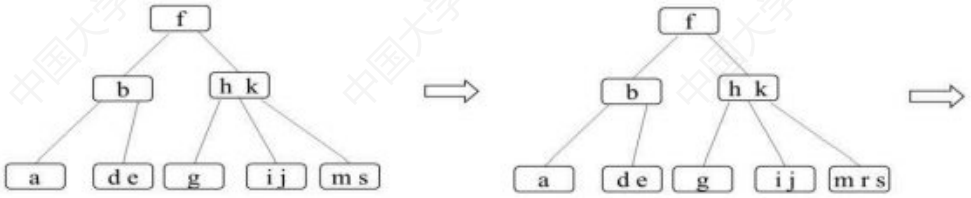


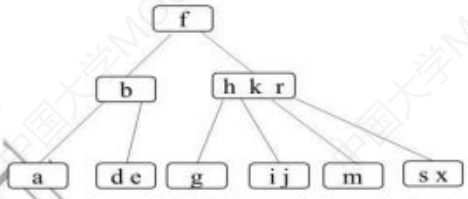
8、图 7.25 是一棵 4 阶 B-树，请画出向该树中依次插入关键字 35、45 后最终所得的 B-树。



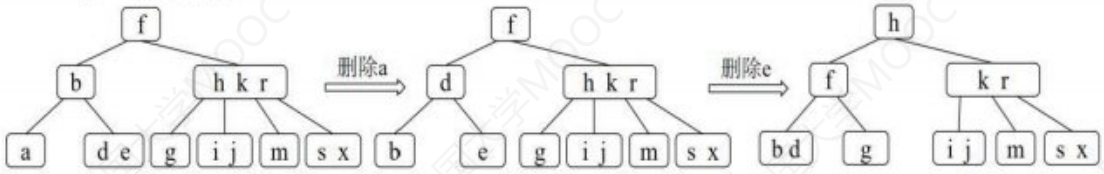
9、从空树开始，使用关键字序列 a,g,f,b,k,d,h,m,j,e,s,i,r,x 建立 4 阶 B-树。

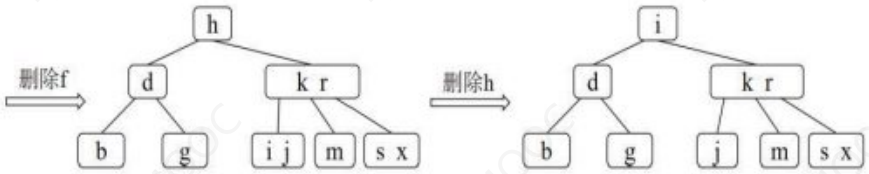






10、从上题的 4 阶 B-树上依次删除 a，e，f，h。





12、说明 B-树适用于外搜索的理由。

B 树的每个节点可以存储多个关键字，它将节点大小设置为磁盘页的大小，充分利用了磁盘预读的功能。每次读取磁盘页时就会读取整个节点。也正因每个节点存储着非常多个关键字，树的深度就会非常的小。进而要执行的磁盘读取操作次数就会非常少，更多的是在内存中对读取进来的数据进行查找。

13、5 阶 B-树的高度为 2 时，树中元素个数最少为多少？

5 个。即：1(根结点关键字 1 个)+2*2(两个孩子，每个结点至少 2 个关键字)=5 个

第八章 散列表

一 基础题:

1. 3，2

2. D

3. 拉链法(或开散列法)

二、扩展题:

1.

线性探查法:

0	1	2	3	4	5	6	7	8	9	10
55	45	35	25	70	80	60	50			

二次探查法:

0	1	2	3	4	5	6	7	8	9	10
45	35	80	25	70	60	50				55

2.

0	1	2	3	4	5	6	7	8	9	10
55	80	35	25	70	60	45	50			

3.

```
 //采用线性探查法的散列表查找算法
int HashSearch1(HashTable HT,KeyType K,int m){
    int d,temp;
    d = h(K,m);                         //调用散列函数获取散列地址
    temp = d;                           //temp 防止进入重复循环
    while(HT[d].key!=-32768)            //如果 Key 不为空
    {
        if(HT[d].Key == K){             //查找成功
            return d;
        }else{
            d = (d+1)%m;                //计算下一个地址
        }
        if(d==temp){                    //循环一次
            return -1;                      //找不到可用地址返回-1
        }
        return d;
    }
}
```

```
//在散列表上插入一个节点的算法
int HashInsert1(HashTable HT,NodeType s,int m){
    int d;
    d = HashSearch1(HT,s,m);          //查找可用地址
    if (d==1)return -1;
    else{
        if (HT[d].key == s.key){return 0;}          //表中已有该节点
        else{
            HT[d] =s ;                              //插入结点
            return 1;                               //返回 1 表示成功
        }
    }
}

4.
//拉链法查找算法
HTNode* HashSeaerch2(HT T,KeyType K,int m){
    HTNode* p = T[h(K,m)];                  //取 K 所在链表的头指针
    while(p!=NULL && p->key!=K){//如果当前链表头指针不为空，且不为待查找结点
        p=p->next;                          //遍历
    }
    return p;
}

//拉链法插入算法
int HashInsert2(HT T,HTNode* s,int m)
{
    int d;
    HTNode* p = HashSearch2(T,s->key,m);        //查找当前结点
    if(p!=NULL)return 0;                        //表明表中没有待插入结点
    else{                                           //将*s 插入在相应链表的表头上
        d = h(s->key,m);                        //获取待插入结点的头指针
        s->next=T[d];
        T[d]=s;
        return 1;                               //插入成功
    }
}
```

第 9 章　图

一、基础题

1．B

2．B

3．A

4．D

5．A

6．A

二、扩展题

1．对于图 9.24 中的有向图，求：

（1）每个顶点的入度；

顶点	0	1	2	3	4	5
入度	2	2	2	2	0	0

（3）强连通分量为：

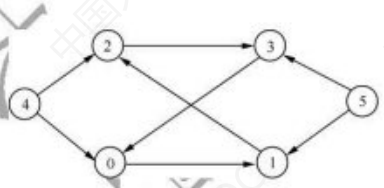
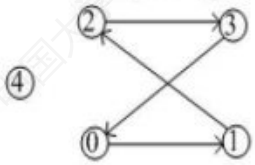


图 9.24

答：



2．画出图 9.25 中的无向图的邻接矩阵。

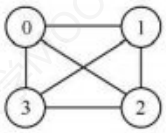


图 9.25

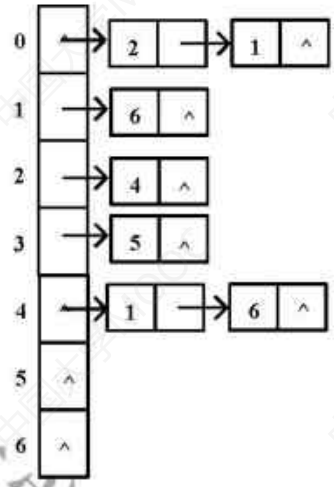
答：无向图的邻接矩阵为

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

3. 设有 7 个顶点没有边的图以邻接表存储，使用程序 9.9 的 Insert 函数插入以下边：

<0,1>,<0,2>,<1,6>,<2,4>,<3,5>,<4,6>,<4,1>，请画出所构建的邻接表。

答：所构建的邻接表为



4．设计一个算法计算邻接表表示的图中各个顶点的出度。

```
void OutDegree (int* outDegree,LGraph *g)
{
  int i;
  ENode *p;
  for( i=0;i<g->n;i++) outDegree[i]=0;
  for(i=0;i<g->n;i++)
  for( p=g->a[i];p;p=p->nextArc)
      outDegree[i]++;
}
```

5．设计一个算法计算邻接表表示的图中各个顶点的入度。

```
void InDegree (int* inDegree,LGraph *g)
{
  int i;
  ENode *p;
  for( i=0;i<g->n;i++) inDegree[i]=0;
  for(i=0;i<g->n;i++)
  for( p=g->a[i];p;p=p->nextArc)
      inDegree[p->adjVex]++;
}
```

6．设计一个算法计算邻接表表示的图中任意顶点 u 的入度。

```
int InDegreeU(LGraph g, int u)
{   ENode *p;
    int i,indegree= 0;
    if (u<0||u>g.n-1) return -1;

        for(i=0;i<g.n;i++)
        {
                p = g.a[i];
                while (p)
                {
                        if(p->adjVex==u)
                        indegree++;
                        p = p->nextArc;
                }
}
return indegree;
}
```

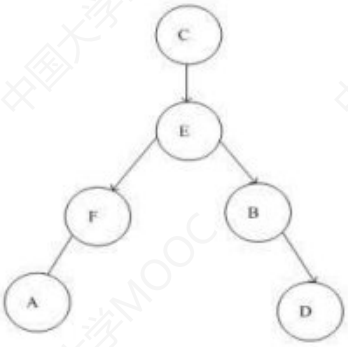
7．设计一个算法计算邻接表表示的图中任意顶点 u 的出度。

```
int OutDegreeU(LGraph g ,int u )
{
     int out=0;ENode *p;
     if (u<0||u>g.n-1) return -1;
     p=g.a[u];
     while(p)
     {
        out++;
        p=p->nextArc;
     }
     return out;
     }
}
```

10．已知图 G 的邻接表表示如图 9.23 所示，在此邻接表上进行以顶点 C 为起始顶点的深度优先遍历，画出深度优先遍历顶点序列以及生成森林(或生成树)。

答：深度优先遍历顶点序列为 C E F A B D

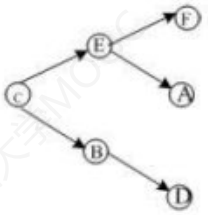
其生成树为：



11．已知图 G 的邻接表表示如图 9.23 所示，在此邻接表上进行以顶点 C 为起始顶点的宽度优先遍历，画出宽度优先遍历顶点序列以及生成森林(或生成树)。

答：序列为 C E BFAD

其生成树为：



15．对图 9.26 所示有向图以 B 为起点，给出该有向图的所有拓扑排序序列。

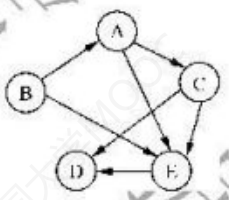


图 9.26

答：拓扑序列为 BACED

18．以顶点 A 作为起始顶点，请用 Prim 算法构造图 9.27 所示的连通图的最小代价生成树。

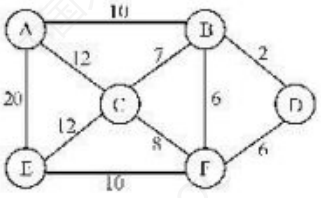
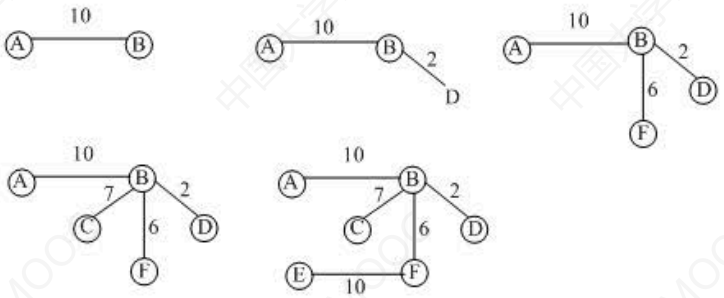


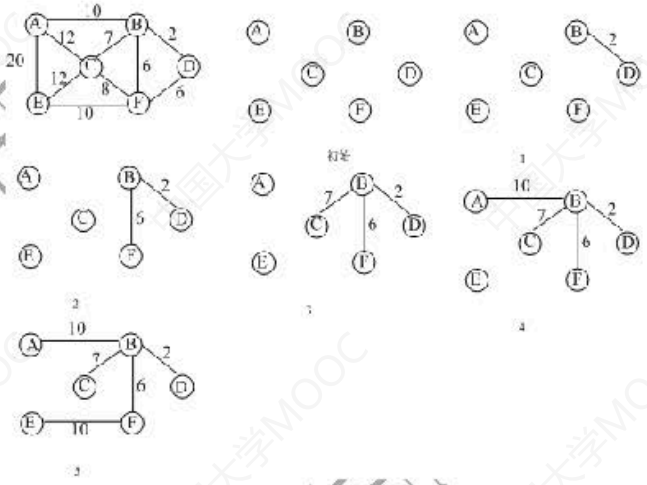
图 9.27

答：本题答案不唯一。其中的一种答案为：



19．用 Kruskal 算法构造图 9.27 所示的连通图的最小代价生成树。

答：本题答案不唯一。其中的一种答案为：



21．使用迪杰斯特拉算法求图 9.29 所示的有向图中以顶点 1 为源点的单源最短路径。

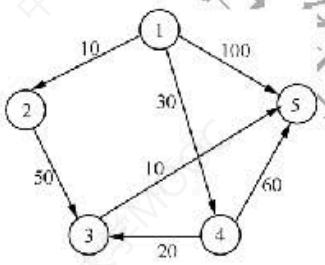


图 9.29

答：

S	d[1] path[1]	d[2] path[2]	d[3] path[3]	d[4] path[4]	d[5] path[5]
1	0,-1	10,1	∞,-1	30,1	100,1
2	0,-1	10,1	60,2	30,1	100,1
4	0,-1	10,1	50,4	30,1	90,4
3	0,-1	10,1	50,4	30,1	60,3
5	0,-1	10,1	50,4	30,1	60,3

第十章 排序

一、基础题

1.

直接插入排序：

初始： 65 78 21 30 80 7 79 57 35 26
一： 65 78 21 30 80 7 79 57 35 26
二： 21 65 78 30 80 7 79 57 35 26
三： 21 30 65 78 80 7 79 57 35 26
四： 21 30 65 78 80 7 79 57 35 26
五： 7 21 30 65 78 80 79 57 35 26
六： 7 21 30 65 78 79 80 57 35 26
七： 7 21 30 57 65 78 79 80 35 26
八： 7 21 30 35 57 65 78 79 80 26
九： 7 21 26 30 35 57 65 78 79 80

简单选择排序：

初始：65 78 21 30 80 7 79 57 35 26
一： 7 78 21 30 80 65 79 57 35 26
二： 7 21 78 30 80 65 79 57 35 26
三： 7 21 26 30 80 65 79 57 35 78
四： 7 21 26 30 80 65 79 57 35 78
五： 7 21 26 30 35 65 79 57 80 78
六： 7 21 26 30 35 57 79 65 80 78
七： 7 21 26 30 35 57 65 79 80 78
八： 7 21 26 30 35 57 65 78 80 79
九： 7 21 26 30 35 57 65 78 79 80
最终序列：
7 21 26 30 35 57 65 78 79 80

冒泡排序：

初始：65 78 21 30 80 7 79 57 35 26
一： 65 21 30 78 7 79 57 35 26 80
二： 21 30 65 7 78 57 35 26 79 80
三： 21 30 7 65 57 35 26 78 79 80
四： 21 7 30 57 35 26 65 78 79 80
五： 7 21 30 35 26 57 65 78 79 80
六： 7 21 30 26 35 57 65 78 79 80
七： 7 21 26 30 35 57 65 78 79 80
八： 7 21 26 30 35 57 65 78 79 80
九： 7 21 26 30 35 57 65 78 79 80
十： 7 21 26 30 35 57 65 78 79 80
最后结果：
7 21 26 30 35 57 65 78 79 80

快速排序：

65 78 21 30 80 7 79 57 35 26
一： {57 26 21 30 35 7} 65 {79 80 78}

二： {7 26 21 30 35} 57 65 {79 80 78}
三： 7 {26 21 30 35} 57 65 {79 80 78}
四： 7 {21} 26 {30 35} 57 65 {79 80 78}
五： 7 21 26 30 35 57 65 {79 80 78}
六： 7 21 26 30 35 57 65 78 79 78}

两路合并排序：

65 78 21 30 80 7 79 57 35 26
一： {65 78} {21 30} {7 80} {57 79} {26 35}
二： {21 30 65 78} {7 57 79 80} {26 35}
三： {7 21 30 57 65 78 79 80} {26 35}
四： 7 21 26 30 35 57 65 78 79 80

堆排序：

65 78 21 30 80 7 79 57 35 26
初始堆
80 78 79 57 65 7 21 30 35 26
一：79 78 26 57 65 7 21 30 35 80
二：78 65 26 57 35 7 21 30 79 80
三：65 57 26 30 35 7 21 78 79 80
四：57 35 26 30 21 7 65 78 79 80
五：35 30 26 7 21 57 65 78 79 80
六： 30 21 26 7 35 57 65 78 79 80
七：26 21 7 30 35 57 65 78 79 80
八： 21 7 26 30 35 57 65 78 79 80
九：7 21 26 30 35 57 65 78 79 80

2.

算法	最好时间复杂度	最坏时间复杂度	平均时间复杂度	空间复杂度	稳定性	提前确定最终有序位置
直接插入	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定	不能
简单选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定	能
冒泡排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定	能
快速排序	O(nlogn)	$O(n^2)$	O(nlogn)	O(1)（勘误，O(log2n)	不稳定	能
两路合并	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定	不能
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	不稳定	能

3. A
4. C, B
5. D
6. A (勘误，直接选择改成简单选择)
7. B, C（勘误，C选项为“被排序的数据完全无序”）
8. 7, 5(第五趟就无交换了)
9. 50, 60, 40, 20
11.3

二、扩展题

1. 将 n 个元素存放在一个数组中，设计算法输出关键字最小的前 k (k<n) 个元素。

```
public static int[] getMinKNumsByHeap(int[] arr, int k) {
        if (k < 1 || k > arr.length) {
            return arr;
        }
        int[] heap = new int[k];
        for (int i = 0; i != k; i++) {
            heapInsert(heap, arr[i], i);
        }
        for (int i = k; i < arr.length; i++) {
            if (arr[i] < heap[0]) {
                heap[0] = arr[i];
                heapify(heap, 0, k);
            }
        }
        return heap;
    }
    private static void heapInsert(int[] heap, int value, int index) {
        heap[index] = value;
        while (index != 0) {
            int parent = (index - 1) / 2;
            if (heap[parent] < heap[index]) {
                swap(heap, parent, index);
                index = parent;
            } else {
                break;
            }
        }
    }
    private static void heapify(int[] heap, int index, int heapSize) {
        int left = index * 2 + 1;
        int right = index * 2 + 2;
        int largest = index;
        while (left < heapSize) {
            if (heap[left] > heap[index]) {
                largest = left;
            }
```

```
            if (right < heapSize && heap[right] > heap[largest]) {
                largest = right;
            }
            if (largest != index) {
                swap(heap, largest, index);
            } else {
                break;
            }
            index = largest;
            left = index * 2 + 1;
            right = index * 2 + 2;
        }
    }
    private static void swap(int[] heap, int parent, int index) {
        int tmp = heap[index];
        heap[index] = heap[parent];
        heap[parent] = tmp;
    }
```

2. 双向冒泡

```
void DoubleBubble (List *list, int *max, int *min)//假设待排序元素都是整数
{
   int i=0;
   int j=n-1;
   for(;i<n-1;i++,j--)
   {
       if(list->D[i].key > list->D[i+1].key)
           Swap(list->D, i,i+1);
       if(list->D[j].key < list->D[j-1].key)
           Swap(list->D, j,j-1);
    }
      *max = list->D[n-1].key;
      *min = list->D[0].key;
}
```

3. 设计待表头结点的单链表上实现稳定的简单选择排序和直接插入排序算法

```
template   <class   T>   //单链表上的简单选择排序程序如下
void   SingleList<T>::SelSort()
{   Node<T> *p=first, *q,*s; int n=length; T min;
```

```
    while (p->link)
    {   q=p;   s=p->link;   min=s->data;
        While(s->link）
        {   if (min>s->link->data)
            {   min=s->link->data;   q=s;   }
            s=s->link;
        }
        s=q->link;   q->link=s->link;
        q=p->link;   p->link=s;
        s->link=q;   p=s;
    }
}
```

```
//单链表上的直接插入排序如下
template <class T>
void SingleList<T>::DirInsert()
{   Node<T> *p,*q=first->link->link,*t; int n=length;
    if   (!q) return;
    t=new Node <T>;     t->link=0;
    t->data=Maxnum;     first->link->link=t;
    while (q)
    {   t=q; q=q->link; p=first;
        while (p->link->data<t->data)   p=p->link;
        t->link=p->link;   p->link=t;
    }
}
```

4.当待排序的序列为正序或逆序排列时，且每次划分只得到一个比上一次划分少一个记录的子序列，注意另一个为空。如果递归树画出来，它就是一棵斜树。此时需要执行 n－1 次递归调用，且第 i 次划分需要经过 n－i 次关键字的比较才能找到第 i 个记录，也就是枢轴的位置，因此比较次数为 $\sum_{i=1}^{n-1}(n-i)=n-1+n-2+\cdots+1=\frac{n(n-1)}{2}$，最终其时间复杂度为 O(n^2)。

5. 1