浙大数据结构-C 代码

在浙大的数据结构课程里面，我们会学到

一、基本的算法知识：复杂度计算等

二、线性结构的各种操作

三、树和图的各种操作以及应用

四、基本的排序和查找算法

Chapter1 线性结构

1.1 顺式结构线性表定义和插入

```
typedef int Position;
typedef struct LNode *List;
struct LNode {
    ElementType Data[MAXSIZE];
    Position Last;
};

/* 初始化 */
List MakeEmpty()
{
    List L;

    L = (List)malloc(sizeof(struct LNode));
    L->Last = -1;

    return L;
}

```

```
/* 查找 */
#define ERROR -1

Position Find( List L, ElementType X )
{
    Position i = 0;

    while( i <= L->Last && L->Data[i]!= X )
        i++;
    if ( i > L->Last ) return ERROR; /* 如果没找到，返回错误信息 */
    else return i;  /* 找到后返回的是存储位置 */
}

/* 插入 */
/*注意:在插入位置参数 P 上与课程视频有所不同，课程视频中 i 是序列位序（从
1 开始），这里 P 是存储下标位置（从 0 开始），两者差 1*/
bool Insert( List L, ElementType X, Position P )
{ /* 在 L 的指定位置 P 前插入一个新元素 X */
    Position i;

    if ( L->Last == MAXSIZE-1) {
        /* 表空间已满，不能插入 */
```

```
40.         printf("表满");
41.         return false;
42.     }
43.     if ( P<0 || P>L->Last+1 ) { /* 检查插入位置的合法性 */
44.         printf("位置不合法");
45.         return false;
46.     }
47.     for( i=L->Last; i>=P; i-- )
48.         L->Data[i+1] = L->Data[i]; /* 将位置 P 及以后的元素顺序向后移动 */
49.     L->Data[P] = X;  /* 新元素插入 */
50.     L->Last++;        /* Last 仍指向最后元素 */
51.     return true;
52. }
53.
54. /* 删除 */
55. /*注意:在删除位置参数 P 上与课程视频有所不同，课程视频中 i 是序列位序（从
    1 开始），这里 P 是存储下标位置（从 0 开始），两者差 1*/
56. bool Delete( List L, Position P )
57. { /* 从 L 中删除指定位置 P 的元素 */
58.     Position i;
59.
60.     if( P<0 || P>L->Last ) { /* 检查空表及删除位置的合法性 */
```

```
        printf("位置%d 不存在元素", P );
        return false;
    }
    for( i=P+1; i<=L->Last; i++ )
        L->Data[i-1] = L->Data[i]; /* 将位置 P+1 及以后的元素顺序向前移动 */
    L->Last--; /* Last 仍指向最后元素 */
    return true;
}
```

1.2 链式结构线性表之定义和插入

```
typedef struct LNode *PtrToLNode;
struct LNode {
    ElementType Data;
    PtrToLNode Next;
};
typedef PtrToLNode Position;
typedef PtrToLNode List;

/* 查找 */
#define ERROR NULL

Position Find( List L, ElementType X )
{
```

```
    Position p = L; /* p 指向 L 的第 1 个结点 */

    while ( p && p->Data!=X )
        p = p->Next;

    /* 下列语句可以用 return p; 替换 */
    if ( p )
        return p;
    else
        return ERROR;
}

/* 带头结点的插入 */
/*注意:在插入位置参数 P 上与课程视频有所不同，课程视频中 i 是序列位序（从 1 开始），这里 P 是链表结点指针，在 P 之前插入新结点 */
bool Insert( List L, ElementType X, Position P )
{ /* 这里默认 L 有头结点 */
    Position tmp, pre;

    /* 查找 P 的前一个结点 */
    for ( pre=L; pre&&pre->Next!=P; pre=pre->Next ) ;
    if ( pre==NULL ) { /* P 所指的结点不在 L 中 */
```

```
35.         printf("插入位置参数错误\n");
36.         return false;
37.     }
38.     else { /* 找到了 P 的前一个结点 pre */
39.         /* 在 P 前插入新结点 */
40.         tmp = (Position)malloc(sizeof(struct LNode)); /* 申请、填装结点 */
41.         tmp->Data = X;
42.         tmp->Next = P;
43.         pre->Next = tmp;
44.         return true;
45.     }
46. }
47.
48. /* 带头结点的删除 */
49. /*注意:在删除位置参数 P 上与课程视频有所不同，课程视频中 i 是序列位序（从
    1 开始），这里 P 是拟删除结点指针 */
50. bool Delete( List L, Position P )
51. { /* 这里默认 L 有头结点 */
52.     Position tmp, pre;
53.
54.     /* 查找 P 的前一个结点 */
55.     for ( pre=L; pre&&pre->Next!=P; pre=pre->Next ) ;
```

```
    if ( pre==NULL || P==NULL) { /* P 所指的结点不在 L 中 */
        printf("删除位置参数错误\n");
        return false;
    }
    else { /* 找到了 P 的前一个结点 pre */
        /* 将 P 位置的结点删除 */
        pre->Next = P->Next;
        free(P);
        return true;
    }
}
```

1.3 堆栈

```
typedef int Position;
struct SNode {
    ElementType *Data; /* 存储元素的数组 */
    Position Top;      /* 栈顶指针 */
    int MaxSize;       /* 堆栈最大容量 */
};
typedef struct SNode *Stack;

Stack CreateStack( int MaxSize )
{
    Stack S = (Stack)malloc(sizeof(struct SNode));
```

```
12.     S->Data = (ElementType *)malloc(MaxSize * sizeof(ElementType));
13.     S->Top = -1;
14.     S->MaxSize = MaxSize;
15.     return S;
16. }
17.
18. bool IsFull( Stack S )
19. {
20.     return (S->Top == S->MaxSize-1);
21. }
22.
23. bool Push( Stack S, ElementType X )
24. {
25.     if ( IsFull(S) ) {
26.         printf("堆栈满");
27.         return false;
28.     }
29.     else {
30.         S->Data[++(S->Top)] = X;
31.         return true;
32.     }
33. }
```

```
34.
35. bool IsEmpty( Stack S )
36. {
37.     return (S->Top == -1);
38. }
39.
40. ElementType Pop( Stack S )
41. {
42.     if ( IsEmpty(S) ) {
43.         printf("堆栈空");
44.         return ERROR; /* ERROR 是 ElementType 的特殊值，标志错误 */
45.     }
46.     else
47.         return ( S->Data[(S->Top)--] );
48. }
```

```
1. typedef struct SNode *PtrToSNode;
2. struct SNode {
3.     ElementType Data;
4.     PtrToSNode Next;
5. };
6. typedef PtrToSNode Stack;
7.
```

```
8. Stack CreateStack( )
9. { /* 构建一个堆栈的头结点，返回该结点指针 */
10.     Stack S;
11.
12.     S = (Stack)malloc(sizeof(struct SNode));
13.     S->Next = NULL;
14.     return S;
15. }
16.
17. bool IsEmpty ( Stack S )
18. { /* 判断堆栈 S 是否为空，若是返回 true；否则返回 false */
19.     return ( S->Next == NULL );
20. }
21.
22. bool Push( Stack S, ElementType X )
23. { /* 将元素 X 压入堆栈 S */
24.     PtrToSNode TmpCell;
25.
26.     TmpCell = (PtrToSNode)malloc(sizeof(struct SNode));
27.     TmpCell->Data = X;
28.     TmpCell->Next = S->Next;
29.     S->Next = TmpCell;
```

```
    return true;
}

ElementType Pop( Stack S )
{ /* 删除并返回堆栈 S 的栈顶元素 */
    PtrToSNode FirstCell;
    ElementType TopElem;

    if( IsEmpty(S) ) {
        printf("堆栈空");
        return ERROR;
    }
    else {
        FirstCell = S->Next;
        TopElem = FirstCell->Data;
        S->Next = FirstCell->Next;
        free(FirstCell);
        return TopElem;
    }
}
```

1.4 队列

```
typedef int Position;
struct QNode {
```

```
    ElementType *Data;     /* 存储元素的数组 */
    Position Front, Rear;  /* 队列的头、尾指针 */
    int MaxSize;           /* 队列最大容量 */
};
typedef struct QNode *Queue;

Queue CreateQueue( int MaxSize )
{
    Queue Q = (Queue)malloc(sizeof(struct QNode));
    Q->Data = (ElementType *)malloc(MaxSize * sizeof(ElementType));
    Q->Front = Q->Rear = 0;
    Q->MaxSize = MaxSize;
    return Q;
}

bool IsFull( Queue Q )
{
    return ((Q->Rear+1)%Q->MaxSize == Q->Front);
}

bool AddQ( Queue Q, ElementType X )
{
```

```
    if ( IsFull(Q) ) {
        printf("队列满");
        return false;
    }
    else {
        Q->Rear = (Q->Rear+1)%Q->MaxSize;
        Q->Data[Q->Rear] = X;
        return true;
    }
}

bool IsEmpty( Queue Q )
{
    return (Q->Front == Q->Rear);
}

ElementType DeleteQ( Queue Q )
{
    if ( IsEmpty(Q) ) {
        printf("队列空");
        return ERROR;
    }
```

```
    else {
        Q->Front =(Q->Front+1)%Q->MaxSize;
        return Q->Data[Q->Front];
    }
}
```

```
typedef struct Node *PtrToNode;
struct Node { /* 队列中的结点 */
    ElementType Data;
    PtrToNode Next;
};
typedef PtrToNode Position;

struct QNode {
    Position Front, Rear;  /* 队列的头、尾指针 */
    int MaxSize;           /* 队列最大容量 */
};
typedef struct QNode *Queue;

bool IsEmpty( Queue Q )
{
    return ( Q->Front == NULL);
}
```

```
18.
19. ElementType DeleteQ( Queue Q )
20. {
21.     Position FrontCell;
22.     ElementType FrontElem;
23.
24.     if  ( IsEmpty(Q) ) {
25.         printf("队列空");
26.         return  ERROR;
27.     }
28.     else {
29.         FrontCell = Q->Front;
30.         if ( Q->Front == Q->Rear ) /* 若队列只有一个元素 */
31.             Q->Front = Q->Rear = NULL; /* 删除后队列置为空 */
32.         else
33.             Q->Front = Q->Front->Next;
34.         FrontElem = FrontCell->Data;
35.
36.         free( FrontCell );  /* 释放被删除结点空间  */
37.         return  FrontElem;
38.     }
39. }
```

Chapter 2 树

2.1 树的表示与二叉树的存储结构

```
typedef struct TNode *Position;
typedef Position BinTree; /* 二叉树类型 */
struct TNode{ /* 树结点定义 */
    ElementType Data; /* 结点数据 */
    BinTree Left;     /* 指向左子树 */
    BinTree Right;    /* 指向右子树 */
}
```

2.2 二叉树的遍历

```
void InorderTraversal( BinTree BT )
{
    if( BT ) {
        InorderTraversal( BT->Left );
        /* 此处假设对 BT 结点的访问就是打印数据 */
        printf("%d ", BT->Data); /* 假设数据为整型 */
        InorderTraversal( BT->Right );
    }
}

void PreorderTraversal( BinTree BT )
{
    if( BT ) {
        printf("%d ", BT->Data );
```

```
15.         PreorderTraversal( BT->Left );
16.         PreorderTraversal( BT->Right );
17.     }
18.}
19.
20.void PostorderTraversal( BinTree BT )
21.{
22.    if( BT ) {
23.         PostorderTraversal( BT->Left );
24.         PostorderTraversal( BT->Right );
25.         printf("%d ", BT->Data);
26.    }
27.}
28.
29.void LevelorderTraversal ( BinTree BT )
30.{
31.    Queue Q;
32.    BinTree T;
33.
34.    if ( !BT ) return; /* 若是空树则直接返回 */
35.
36.    Q = CreatQueue(); /* 创建空队列 Q */
```

```
    AddQ( Q, BT );
    while ( !IsEmpty(Q) ) {
        T = DeleteQ( Q );
        printf("%d ", T->Data); /* 访问取出队列的结点 */
        if ( T->Left )  AddQ( Q, T->Left );
        if ( T->Right ) AddQ( Q, T->Right );
    }
}
```

2.3 BST 二叉搜索树

```
BinTree Insert( BinTree BST, ElementType X )
{
    if( !BST ){ /* 若原树为空，生成并返回一个结点的二叉搜索树 */
        BST = (BinTree)malloc(sizeof(struct TNode));
        BST->Data = X;
        BST->Left = BST->Right = NULL;
    }
    else { /* 开始找要插入元素的位置 */
        if( X < BST->Data )
            BST->Left = Insert( BST->Left, X );  /*递归插入左子树*/
        else if( X > BST->Data )
            BST->Right = Insert( BST->Right, X ); /*递归插入右子树*/
        /* else X 已经存在，什么都不做 */
    }
```

```
15.     return BST;
16. }
17.
18. BinTree Delete( BinTree BST, ElementType X )
19. {
20.     Position Tmp;
21.
22.     if( !BST )
23.         printf("要删除的元素未找到");
24.     else {
25.         if( X < BST->Data )
26.             BST->Left = Delete( BST->Left, X );   /* 从左子树递归删除 */
27.         else if( X > BST->Data )
28.             BST->Right = Delete( BST->Right, X ); /* 从右子树递归删除 */
29.         else { /* BST 就是要删除的结点 */
30.             /* 如果被删除结点有左右两个子结点 */
31.             if( BST->Left && BST->Right ) {
32.                 /* 从右子树中找最小的元素填充删除结点 */
33.                 Tmp = FindMin( BST->Right );
34.                 BST->Data = Tmp->Data;
35.                 /* 从右子树中删除最小元素 */
36.                 BST->Right = Delete( BST->Right, BST->Data );
```

```
            }
            else { /* 被删除结点有一个或无子结点 */
                Tmp = BST;
                if( !BST->Left )       /* 只有右孩子或无子结点 */
                    BST = BST->Right;
                else                   /* 只有左孩子 */
                    BST = BST->Left;
                free( Tmp );
            }
        }
    }
    return BST;
}
```

2.4 AVL 平衡二叉树

```
typedef struct AVLNode *Position;
typedef Position AVLTree; /* AVL 树类型 */
struct AVLNode{
    ElementType Data; /* 结点数据 */
    AVLTree Left;     /* 指向左子树 */
    AVLTree Right;    /* 指向右子树 */
    int Height;       /* 树高 */
};

```

```
int Max ( int a, int b )
{
    return a > b ? a : b;
}

AVLTree SingleLeftRotation ( AVLTree A )
{ /* 注意：A 必须有一个左子结点 B */
  /* 将 A 与 B 做左单旋，更新 A 与 B 的高度，返回新的根结点 B */

    AVLTree B = A->Left;
    A->Left = B->Right;
    B->Right = A;
    A->Height = Max( GetHeight(A->Left), GetHeight(A->Right) ) + 1;
    B->Height = Max( GetHeight(B->Left), A->Height ) + 1;

    return B;
}

AVLTree DoubleLeftRightRotation ( AVLTree A )
{ /* 注意：A 必须有一个左子结点 B，且 B 必须有一个右子结点 C */
  /* 将 A、B 与 C 做两次单旋，返回新的根结点 C */

```

```
    /* 将 B 与 C 做右单旋，C 被返回 */
    A->Left = SingleRightRotation(A->Left);
    /* 将 A 与 C 做左单旋，C 被返回 */
    return SingleLeftRotation(A);
}

/*****************************************/
/* 对称的右单旋与右-左双旋请自己实现 */
/*****************************************/

AVLTree Insert( AVLTree T, ElementType X )
{ /* 将 X 插入 AVL 树 T 中，并且返回调整后的 AVL 树 */
    if ( !T ) { /* 若插入空树，则新建包含一个结点的树 */
        T = (AVLTree)malloc(sizeof(struct AVLNode));
        T->Data = X;
        T->Height = 0;
        T->Left = T->Right = NULL;
    } /* if (插入空树) 结束 */

    else if ( X < T->Data ) {
        /* 插入 T 的左子树 */
        T->Left = Insert( T->Left, X);
```

```
54.     /* 如果需要左旋 */
55.     if ( GetHeight(T->Left)-GetHeight(T->Right) == 2 )
56.         if ( X < T->Left->Data )
57.             T = SingleLeftRotation(T);      /* 左单旋 */
58.         else
59.             T = DoubleLeftRightRotation(T); /* 左-右双旋 */
60.   } /* else if (插入左子树) 结束 */
61.
62.   else if ( X > T->Data ) {
63.     /* 插入 T 的右子树 */
64.     T->Right = Insert( T->Right, X );
65.     /* 如果需要右旋 */
66.     if ( GetHeight(T->Left)-GetHeight(T->Right) == -2 )
67.         if ( X > T->Right->Data )
68.             T = SingleRightRotation(T);     /* 右单旋 */
69.         else
70.             T = DoubleRightLeftRotation(T); /* 右-左双旋 */
71.   } /* else if (插入右子树) 结束 */
72.
73.   /* else X == T->Data，无须插入 */
74.
75.   /* 别忘了更新树高 */
```

```
    T->Height = Max( GetHeight(T->Left), GetHeight(T->Right) ) + 1;

    return T;
}
```

2.5 堆 Heap

```
typedef struct HNode *Heap; /* 堆的类型定义 */
struct HNode {
    ElementType *Data; /* 存储元素的数组 */
    int Size;          /* 堆中当前元素个数 */
    int Capacity;      /* 堆的最大容量 */
};
typedef Heap MaxHeap; /* 最大堆 */
typedef Heap MinHeap; /* 最小堆 */

#define MAXDATA 1000  /* 该值应根据具体情况定义为大于堆中所有可能元素的值 */

MaxHeap CreateHeap( int MaxSize )
{ /* 创建容量为 MaxSize 的空的最大堆 */

    MaxHeap H = (MaxHeap)malloc(sizeof(struct HNode));
    H->Data = (ElementType *)malloc((MaxSize+1)*sizeof(ElementType));
    H->Size = 0;
```

```
    H->Capacity = MaxSize;
    H->Data[0] = MAXDATA; /* 定义"哨兵"为大于堆中所有可能元素的值*/

    return H;
}

bool IsFull( MaxHeap H )
{
    return (H->Size == H->Capacity);
}

bool Insert( MaxHeap H, ElementType X )
{ /* 将元素 X 插入最大堆 H，其中 H->Data[0]已经定义为哨兵 */
    int i;

    if ( IsFull(H) ) {
        printf("最大堆已满");
        return false;
    }
    i = ++H->Size; /* i 指向插入后堆中的最后一个元素的位置 */
    for ( ; H->Data[i/2] < X; i/=2 )
        H->Data[i] = H->Data[i/2]; /* 上滤 X */
```

```
    H->Data[i] = X; /* 将 X 插入 */
    return true;
}

#define ERROR -1 /* 错误标识应根据具体情况定义为堆中不可能出现的元素值 */

bool IsEmpty( MaxHeap H )
{
    return (H->Size == 0);
}

ElementType DeleteMax( MaxHeap H )
{ /* 从最大堆 H 中取出键值为最大的元素，并删除一个结点 */
    int Parent, Child;
    ElementType MaxItem, X;

    if ( IsEmpty(H) ) {
        printf("最大堆已为空");
        return ERROR;
    }

```

```
61.     MaxItem = H->Data[1]; /* 取出根结点存放的最大值 */
62.     /* 用最大堆中最后一个元素从根结点开始向上过滤下层结点 */
63.     X = H->Data[H->Size--]; /* 注意当前堆的规模要减小 */
64.     for( Parent=1; Parent*2<=H->Size; Parent=Child ) {
65.         Child = Parent * 2;
66.         if( (Child!=H->Size) && (H->Data[Child]<H->Data[Child+1]) )
67.             Child++;  /* Child 指向左右子结点的较大者 */
68.         if( X >= H->Data[Child] ) break; /* 找到了合适位置 */
69.         else  /* 下滤 X */
70.             H->Data[Parent] = H->Data[Child];
71.     }
72.     H->Data[Parent] = X;
73. 
74.     return MaxItem;
75. }
76. 
77. /*---------- 建造最大堆 ----------*/
78. void PercDown( MaxHeap H, int p )
79. { /* 下滤：将 H 中以 H->Data[p]为根的子堆调整为最大堆 */
80.     int Parent, Child;
81.     ElementType X;
82. 
```

```
    X = H->Data[p]; /* 取出根结点存放的值 */
    for( Parent=p; Parent*2<=H->Size; Parent=Child ) {
        Child = Parent * 2;
        if( (Child!=H->Size) && (H->Data[Child]<H->Data[Child+1]) )
            Child++;  /* Child 指向左右子结点的较大者 */
        if( X >= H->Data[Child] ) break; /* 找到了合适位置 */
        else  /* 下滤 X */
            H->Data[Parent] = H->Data[Child];
    }
    H->Data[Parent] = X;
}

void BuildHeap( MaxHeap H )
{ /* 调整 H->Data[]中的元素，使满足最大堆的有序性  */
  /* 这里假设所有 H->Size 个元素已经存在 H->Data[]中 */

    int i;

    /* 从最后一个结点的父节点开始，到根结点 1 */
    for( i = H->Size/2; i>0; i-- )
        PercDown( H, i );
}
```

2.6 Huffman 树

```
typedef struct TreeNode *HuffmanTree;
struct TreeNode{
  int Weight;
  HuffmanTree Left, Right;
}
HuffmanTree Huffman( MinHeap H )
{    /* 假设H->Size个权值已经存在H->Elements[]->Weight里 */
     int i;  HuffmanTree T;
     BuildMinHeap(H); /*将H->Elements[]按权值调整为最小堆*/
     for (i = 1; i < H->Size; i++) { /*做H->Size-1次合并*/
          T = malloc( sizeof( struct TreeNode) ); /*建立新结点*/
          T->Left = DeleteMin(H);
                    /*从最小堆中删除一个结点，作为新T的左子结点*/
          T->Right = DeleteMin(H);
                    /*从最小堆中删除一个结点，作为新T的右子结点*/
          T->Weight = T->Left->Weight+T->Right->Weight;
                    /*计算新权值*/
          Insert( H, T ); /*将新T插入最小堆*/
     }
     T = DeleteMin(H);
     return T;
}
```

整体复杂度为O(N logN)

2.7 并查集

```
1. #define MAXN 1000                /* 集合最大元素个数 */
2. typedef int ElementType;          /* 默认元素可以用非负整数表示 */
3. typedef int SetName;              /* 默认用根结点的下标作为集合名称 */
4. typedef ElementType SetType[MAXN]; /* 假设集合元素下标从 0 开始 */
5.
6. void Union( SetType S, SetName Root1, SetName Root2 )
7. { /* 这里默认 Root1 和 Root2 是不同集合的根结点 */
8.    /* 保证小集合并入大集合 */
9.    if ( S[Root2] < S[Root1] ) { /* 如果集合 2 比较大 */
10.       S[Root2] += S[Root1];     /* 集合 1 并入集合 2  */
11.       S[Root1] = Root2;
12.   }
```

```
    else {                          /* 如果集合 1 比较大 */
        S[Root1] += S[Root2];       /* 集合 2 并入集合 1  */
        S[Root2] = Root1;
    }
}

SetName Find( SetType S, ElementType X )
{ /* 默认集合元素全部初始化为–1 */
    if ( S[X] < 0 ) /* 找到集合的根 */
        return X;
    else
        return S[X] = Find( S, S[X] ); /* 路径压缩 */
}
```

Chapter3 图

3.1 图的定义与构建

```
/* 图的邻接矩阵表示法 */

#define MaxVertexNum 100    /* 最大顶点数设为 100 */
#define INFINITY 65535        /* ∞设为双字节无符号整数的最大值 65535*/
typedef int Vertex;           /* 用顶点下标表示顶点,为整型 */
typedef int WeightType;       /* 边的权值设为整型 */
typedef char DataType;        /* 顶点存储的数据类型设为字符型 */

```

```
/* 边的定义 */
typedef struct ENode *PtrToENode;
struct ENode{
    Vertex V1, V2;      /* 有向边<V1, V2> */
    WeightType Weight;  /* 权重 */
};
typedef PtrToENode Edge;

/* 图结点的定义 */
typedef struct GNode *PtrToGNode;
struct GNode{
    int Nv;  /* 顶点数 */
    int Ne;  /* 边数   */
    WeightType G[MaxVertexNum][MaxVertexNum]; /* 邻接矩阵 */
    DataType Data[MaxVertexNum];      /* 存顶点的数据 */
    /* 注意：很多情况下，顶点无数据，此时 Data[]可以不用出现 */
};
typedef PtrToGNode MGraph; /* 以邻接矩阵存储的图类型 */


MGraph CreateGraph( int VertexNum )
```

```
{ /* 初始化一个有 VertexNum 个顶点但没有边的图 */
    Vertex V, W;
    MGraph Graph;

    Graph = (MGraph)malloc(sizeof(struct GNode)); /* 建立图 */
    Graph->Nv = VertexNum;
    Graph->Ne = 0;
    /* 初始化邻接矩阵 */
    /* 注意：这里默认顶点编号从 0 开始，到(Graph->Nv - 1) */
    for (V=0; V<Graph->Nv; V++)
        for (W=0; W<Graph->Nv; W++)
            Graph->G[V][W] = INFINITY;

    return Graph;
}

void InsertEdge( MGraph Graph, Edge E )
{
    /* 插入边 <V1, V2> */
    Graph->G[E->V1][E->V2] = E->Weight;
    /* 若是无向图，还要插入边<V2, V1> */
    Graph->G[E->V2][E->V1] = E->Weight;
```

```
}

MGraph BuildGraph()
{
    MGraph Graph;
    Edge E;
    Vertex V;
    int Nv, i;

    scanf("%d", &Nv);   /* 读入顶点个数 */
    Graph = CreateGraph(Nv); /* 初始化有 Nv 个顶点但没有边的图 */

    scanf("%d", &(Graph->Ne));   /* 读入边数 */
    if ( Graph->Ne != 0 ) { /* 如果有边 */
        E = (Edge)malloc(sizeof(struct ENode)); /* 建立边结点 */
        /* 读入边，格式为"起点 终点 权重"，插入邻接矩阵 */
        for (i=0; i<Graph->Ne; i++) {
            scanf("%d %d %d", &E->V1, &E->V2, &E->Weight);
            /* 注意：如果权重不是整型，Weight 的读入格式要改 */
            InsertEdge( Graph, E );
        }
    }
```

```
75.
76.     /* 如果顶点有数据的话，读入数据 */
77.     for (V=0; V<Graph->Nv; V++)
78.         scanf(" %c", &(Graph->Data[V]));
79.
80.     return Graph;
81. }
```

```
1. /* 图的邻接表表示法 */
2.
3. #define MaxVertexNum 100    /* 最大顶点数设为 100 */
4. typedef int Vertex;          /* 用顶点下标表示顶点,为整型 */
5. typedef int WeightType;      /* 边的权值设为整型 */
6. typedef char DataType;       /* 顶点存储的数据类型设为字符型 */
7.
8. /* 边的定义 */
9. typedef struct ENode *PtrToENode;
10. struct ENode{
11.     Vertex V1, V2;      /* 有向边<V1, V2> */
12.     WeightType Weight;  /* 权重 */
13. };
14. typedef PtrToENode Edge;
15.
```

```
/* 邻接点的定义 */
typedef struct AdjVNode *PtrToAdjVNode;
struct AdjVNode{
    Vertex AdjV;        /* 邻接点下标 */
    WeightType Weight;  /* 边权重 */
    PtrToAdjVNode Next;    /* 指向下一个邻接点的指针 */
};

/* 顶点表头结点的定义 */
typedef struct Vnode{
    PtrToAdjVNode FirstEdge;/* 边表头指针 */
    DataType Data;          /* 存顶点的数据 */
    /* 注意：很多情况下，顶点无数据，此时 Data 可以不用出现 */
} AdjList[MaxVertexNum];    /* AdjList 是邻接表类型 */

/* 图结点的定义 */
typedef struct GNode *PtrToGNode;
struct GNode{
    int Nv;     /* 顶点数 */
    int Ne;     /* 边数   */
    AdjList G;  /* 邻接表 */
};
```

```
typedef PtrToGNode LGraph; /* 以邻接表方式存储的图类型 */

LGraph CreateGraph( int VertexNum )
{ /* 初始化一个有 VertexNum 个顶点但没有边的图 */
    Vertex V;
    LGraph Graph;

    Graph = (LGraph)malloc( sizeof(struct GNode) ); /* 建立图 */
    Graph->Nv = VertexNum;
    Graph->Ne = 0;
    /* 初始化邻接表头指针 */
    /* 注意：这里默认顶点编号从 0 开始，到(Graph->Nv - 1) */
        for (V=0; V<Graph->Nv; V++)
            Graph->G[V].FirstEdge = NULL;

    return Graph;
}

void InsertEdge( LGraph Graph, Edge E )
{
```

```
60.     PtrToAdjVNode NewNode;
61.
62.     /* 插入边 <V1, V2> */
63.     /* 为 V2 建立新的邻接点 */
64.     NewNode = (PtrToAdjVNode)malloc(sizeof(struct AdjVNode));
65.     NewNode->AdjV = E->V2;
66.     NewNode->Weight = E->Weight;
67.     /* 将 V2 插入 V1 的表头 */
68.     NewNode->Next = Graph->G[E->V1].FirstEdge;
69.     Graph->G[E->V1].FirstEdge = NewNode;
70.
71.     /* 若是无向图，还要插入边 <V2, V1> */
72.     /* 为 V1 建立新的邻接点 */
73.     NewNode = (PtrToAdjVNode)malloc(sizeof(struct AdjVNode));
74.     NewNode->AdjV = E->V1;
75.     NewNode->Weight = E->Weight;
76.     /* 将 V1 插入 V2 的表头 */
77.     NewNode->Next = Graph->G[E->V2].FirstEdge;
78.     Graph->G[E->V2].FirstEdge = NewNode;
79. }
80.
81. LGraph BuildGraph()
```

```
{
    LGraph Graph;
    Edge E;
    Vertex V;
    int Nv, i;

    scanf("%d", &Nv);   /* 读入顶点个数 */
    Graph = CreateGraph(Nv); /* 初始化有 Nv 个顶点但没有边的图 */

    scanf("%d", &(Graph->Ne));   /* 读入边数 */
    if ( Graph->Ne != 0 ) { /* 如果有边 */
        E = (Edge)malloc( sizeof(struct ENode) ); /* 建立边结点 */
        /* 读入边，格式为"起点 终点 权重"，插入邻接矩阵 */
        for (i=0; i<Graph->Ne; i++) {
            scanf("%d %d %d", &E->V1, &E->V2, &E->Weight);
            /* 注意：如果权重不是整型，Weight 的读入格式要改 */
            InsertEdge( Graph, E );
        }
    }

    /* 如果顶点有数据的话，读入数据 */
    for (V=0; V<Graph->Nv; V++)
```

```
104.         scanf(" %c", &(Graph->G[V].Data));
105.
106.     return Graph;
107. }
```

3.2 图的遍历

```
1. /* 邻接表存储的图 - DFS */
2.
3. void Visit( Vertex V )
4. {
5.     printf("正在访问顶点%d\n", V);
6. }
7.
8. /* Visited[]为全局变量，已经初始化为 false */
9. void DFS( LGraph Graph, Vertex V, void (*Visit)(Vertex) )
10.{   /* 以 V 为出发点对邻接表存储的图 Graph 进行 DFS 搜索 */
11.    PtrToAdjVNode W;
12.
13.    Visit( V ); /* 访问第 V 个顶点 */
14.    Visited[V] = true; /* 标记 V 已访问 */
15.
16.    for( W=Graph->G[V].FirstEdge; W; W=W->Next ) /* 对 V 的每个邻接点
    W->AdjV */
17.        if ( !Visited[W->AdjV] )    /* 若 W->AdjV 未被访问 */
```

```
            DFS( Graph, W->AdjV, Visit );    /* 则递归访问之 */
}
/* 邻接矩阵存储的图 - BFS */

/* IsEdge(Graph, V, W)检查<V, W>是否图 Graph 中的一条边, 即 W 是否 V 的邻接点。 */
/* 此函数根据图的不同类型要做不同的实现，关键取决于对不存在的边的表示方法。*/
/* 例如对有权图, 如果不存在的边被初始化为 INFINITY, 则函数实现如下:        */
bool IsEdge( MGraph Graph, Vertex V, Vertex W )
{
    return Graph->G[V][W]<INFINITY ? true : false;
}

/* Visited[]为全局变量，已经初始化为 false */
void BFS ( MGraph Graph, Vertex S, void (*Visit)(Vertex) )
{  /* 以 S 为出发点对邻接矩阵存储的图 Graph 进行 BFS 搜索 */
    Queue Q;
    Vertex V, W;

```

```
    Q = CreateQueue( MaxSize ); /* 创建空队列, MaxSize 为外部定义的常数 */
    /* 访问顶点 S：此处可根据具体访问需要改写 */
    Visit( S );
    Visited[S] = true; /* 标记 S 已访问 */
    AddQ(Q, S); /* S 入队列 */

    while ( !IsEmpty(Q) ) {
        V = DeleteQ(Q);  /* 弹出 V */
        for( W=0; W<Graph->Nv; W++ ) /* 对图中的每个顶点 W */
            /* 若 W 是 V 的邻接点并且未访问过 */
            if ( !Visited[W] && IsEdge(Graph, V, W) ) {
                /* 访问顶点 W */
                Visit( W );
                Visited[W] = true; /* 标记 W 已访问 */
                AddQ(Q, W); /* W 入队列 */
            }
    } /* while 结束*/
}
```

3.3 最短路径

```
/* 邻接表存储 - 无权图的单源最短路算法 */

/* dist[]和 path[]全部初始化为-1 */
```

```
void Unweighted ( LGraph Graph, int dist[], int path[], Vertex S )
{
    Queue Q;
    Vertex V;
    PtrToAdjVNode W;

    Q = CreateQueue( Graph->Nv ); /* 创建空队列, MaxSize 为外部定义的常数 */
    dist[S] = 0; /* 初始化源点 */
    AddQ (Q, S);

    while( !IsEmpty(Q) ){
        V = DeleteQ(Q);
        for ( W=Graph->G[V].FirstEdge; W; W=W->Next ) /* 对 V 的每个邻接点 W->AdjV */
            if ( dist[W->AdjV]==-1 ) { /* 若 W->AdjV 未被访问过 */
                dist[W->AdjV] = dist[V]+1; /* W->AdjV 到 S 的距离更新 */
                path[W->AdjV] = V; /* 将 V 记录在 S 到 W->AdjV 的路径上 */
                AddQ(Q, W->AdjV);
            }
    } /* while 结束*/
}
```

```
/* 邻接矩阵存储 - 有权图的单源最短路算法 */

Vertex FindMinDist( MGraph Graph, int dist[], int collected[] )
{ /* 返回未被收录顶点中 dist 最小者 */
    Vertex MinV, V;
    int MinDist = INFINITY;

    for (V=0; V<Graph->Nv; V++) {
        if ( collected[V]==false && dist[V]<MinDist) {
            /* 若 V 未被收录，且 dist[V]更小 */
            MinDist = dist[V]; /* 更新最小距离 */
            MinV = V; /* 更新对应顶点 */
        }
    }
    if (MinDist < INFINITY) /* 若找到最小 dist */
        return MinV; /* 返回对应的顶点下标 */
    else return ERROR;  /* 若这样的顶点不存在，返回错误标记 */
}

bool Dijkstra( MGraph Graph, int dist[], int path[], Vertex S )
{
    int collected[MaxVertexNum];
```

```
23.     Vertex V, W;
24.
25.     /* 初始化：此处默认邻接矩阵中不存在的边用 INFINITY 表示 */
26.     for ( V=0; V<Graph->Nv; V++ ) {
27.         dist[V] = Graph->G[S][V];
28.         if ( dist[V]<INFINITY )
29.             path[V] = S;
30.         else
31.             path[V] = -1;
32.         collected[V] = false;
33.     }
34.     /* 先将起点收入集合 */
35.     dist[S] = 0;
36.     collected[S] = true;
37.
38.     while (1) {
39.         /* V = 未被收录顶点中 dist 最小者 */
40.         V = FindMinDist( Graph, dist, collected );
41.         if ( V==ERROR ) /* 若这样的 V 不存在 */
42.             break;      /* 算法结束 */
43.         collected[V] = true;  /* 收录 V */
44.         for( W=0; W<Graph->Nv; W++ ) /* 对图中的每个顶点 W */
```

```
            /* 若 W 是 V 的邻接点并且未被收录 */
            if ( collected[W]==false && Graph->G[V][W]<INFINITY ) {
                if ( Graph->G[V][W]<0 ) /* 若有负边 */
                    return false; /* 不能正确解决，返回错误标记 */
                /* 若收录 V 使得 dist[W]变小 */
                if ( dist[V]+Graph->G[V][W] < dist[W] ) {
                    dist[W] = dist[V]+Graph->G[V][W]; /* 更新 dist[W] */
                    path[W] = V; /* 更新 S 到 W 的路径 */
                }
            }
    } /* while 结束*/
    return true; /* 算法执行完毕，返回正确标记 */
}
```

```
/* 邻接矩阵存储 - 多源最短路算法 */

bool Floyd( MGraph Graph, WeightType D[][MaxVertexNum], Vertex path[][
    MaxVertexNum] )
{
    Vertex i, j, k;

    /* 初始化 */
    for ( i=0; i<Graph->Nv; i++ )
```

```
        for( j=0; j<Graph->Nv; j++ ) {
            D[i][j] = Graph->G[i][j];
            path[i][j] = -1;
        }

    for( k=0; k<Graph->Nv; k++ )
        for( i=0; i<Graph->Nv; i++ )
            for( j=0; j<Graph->Nv; j++ )
                if( D[i][k] + D[k][j] < D[i][j] ) {
                    D[i][j] = D[i][k] + D[k][j];
                    if ( i==j && D[i][j]<0 ) /* 若发现负值圈 */
                        return false; /* 不能正确解决，返回错误标记 */
                    path[i][j] = k;
                }
    return true; /* 算法执行完毕，返回正确标记 */
}
```

3.4 最小生成树

```
/* 邻接矩阵存储 - Prim 最小生成树算法 */

Vertex FindMinDist( MGraph Graph, WeightType dist[] )
{ /* 返回未被收录顶点中 dist 最小者 */
    Vertex MinV, V;
    WeightType MinDist = INFINITY;
```

```
7.
8.     for (V=0; V<Graph->Nv; V++) {
9.         if ( dist[V]!=0 && dist[V]<MinDist) {
10.            /* 若 V 未被收录，且 dist[V]更小 */
11.            MinDist = dist[V]; /* 更新最小距离 */
12.            MinV = V; /* 更新对应顶点 */
13.        }
14.    }
15.    if (MinDist < INFINITY) /* 若找到最小 dist */
16.        return MinV; /* 返回对应的顶点下标 */
17.    else return ERROR;  /* 若这样的顶点不存在，返回-1 作为标记 */
18.}
19.
20.int Prim( MGraph Graph, LGraph MST )
21.{ /* 将最小生成树保存为邻接表存储的图 MST，返回最小权重和 */
22.    WeightType dist[MaxVertexNum], TotalWeight;
23.    Vertex parent[MaxVertexNum], V, W;
24.    int VCount;
25.    Edge E;
26.
27.    /* 初始化。默认初始点下标是 0 */
28.        for (V=0; V<Graph->Nv; V++) {
```

```
29.         /* 这里假设若 V 到 W 没有直接的边，则 Graph->G[V][W]定义为
    INFINITY */
30.         dist[V] = Graph->G[0][V];
31.         parent[V] = 0; /* 暂且定义所有顶点的父结点都是初始点 0 */
32.     }
33.     TotalWeight = 0; /* 初始化权重和     */
34.     VCount = 0;      /* 初始化收录的顶点数 */
35.     /* 创建包含所有顶点但没有边的图。注意用邻接表版本 */
36.     MST = CreateGraph(Graph->Nv);
37.     E = (Edge)malloc( sizeof(struct ENode) ); /* 建立空的边结点 */
38.
39.     /* 将初始点 0 收录进 MST */
40.     dist[0] = 0;
41.     VCount ++;
42.     parent[0] = -1; /* 当前树根是 0 */
43.
44.     while (1) {
45.         V = FindMinDist( Graph, dist );
46.         /* V = 未被收录顶点中 dist 最小者 */
47.         if ( V==ERROR ) /* 若这样的 V 不存在 */
48.             break;   /* 算法结束 */
49.
```

```
50.         /* 将 V 及相应的边<parent[V], V>收录进 MST */
51.         E->V1 = parent[V];
52.         E->V2 = V;
53.         E->Weight = dist[V];
54.         InsertEdge( MST, E );
55.         TotalWeight += dist[V];
56.         dist[V] = 0;
57.         VCount++;
58.
59.         for( W=0; W<Graph->Nv; W++ ) /* 对图中的每个顶点 W */
60.             if ( dist[W]!=0 && Graph->G[V][W]<INFINITY ) {
61.             /* 若 W 是 V 的邻接点并且未被收录 */
62.                 if ( Graph->G[V][W] < dist[W] ) {
63.                 /* 若收录 V 使得 dist[W]变小 */
64.                     dist[W] = Graph->G[V][W]; /* 更新 dist[W] */
65.                     parent[W] = V; /* 更新树 */
66.                 }
67.             }
68.     } /* while 结束*/
69.     if ( VCount < Graph->Nv ) /* MST 中收的顶点不到|V|个 */
70.         TotalWeight = ERROR;
71.     return TotalWeight;   /* 算法执行完毕，返回最小权重和或错误标记 */
```

```
}

/* 邻接表存储 - Kruskal 最小生成树算法 */

/*---------------------- 顶点并查集定义 ----------------------*/
typedef Vertex ElementType; /* 默认元素可以用非负整数表示 */
typedef Vertex SetName;     /* 默认用根结点的下标作为集合名称 */
typedef ElementType SetType[MaxVertexNum]; /* 假设集合元素下标从 0 开始 */

void InitializeVSet( SetType S, int N )
{ /* 初始化并查集 */
    ElementType X;

    for ( X=0; X<N; X++ ) S[X] = -1;
}

void Union( SetType S, SetName Root1, SetName Root2 )
{ /* 这里默认 Root1 和 Root2 是不同集合的根结点 */
    /* 保证小集合并入大集合 */
    if ( S[Root2] < S[Root1] ) { /* 如果集合 2 比较大 */
        S[Root2] += S[Root1];     /* 集合 1 并入集合 2  */
        S[Root1] = Root2;
```

```
    }
    else {                          /* 如果集合 1 比较大 */
        S[Root1] += S[Root2];       /* 集合 2 并入集合 1  */
        S[Root2] = Root1;
    }
}

SetName Find( SetType S, ElementType X )
{ /* 默认集合元素全部初始化为–1 */
    if ( S[X] < 0 ) /* 找到集合的根 */
        return X;
    else
        return S[X] = Find( S, S[X] ); /* 路径压缩 */
}

bool CheckCycle( SetType VSet, Vertex V1, Vertex V2 )
{ /* 检查连接 V1 和 V2 的边是否在现有的最小生成树子集中构成回路 */
    Vertex Root1, Root2;

    Root1 = Find( VSet, V1 ); /* 得到 V1 所属的连通集名称 */
    Root2 = Find( VSet, V2 ); /* 得到 V2 所属的连通集名称 */

```

```
43.     if( Root1==Root2 ) /* 若 V1 和 V2 已经连通，则该边不能要 */
44.         return false;
45.     else { /* 否则该边可以被收集，同时将 V1 和 V2 并入同一连通集 */
46.         Union( VSet, Root1, Root2 );
47.         return true;
48.     }
49. }
50. /*---------------------- 并查集定义结束 ----------------------*/
51.
52. /*---------------------- 边的最小堆定义 ----------------------*/
53. void PercDown( Edge ESet, int p, int N )
54. { /* 改编代码 4.24 的 PercDown( MaxHeap H, int p )    */
55.   /* 将 N 个元素的边数组中以 ESet[p]为根的子堆调整为关于 Weight 的最小
    堆 */
56.     int Parent, Child;
57.     struct ENode X;
58.
59.     X = ESet[p]; /* 取出根结点存放的值 */
60.     for( Parent=p; (Parent*2+1)<N; Parent=Child ) {
61.         Child = Parent * 2 + 1;
62.         if( (Child!=N-1) && (ESet[Child].Weight>ESet[Child+1].Weight) )
63.             Child++;  /* Child 指向左右子结点的较小者 */
```

```
64.         if( X.Weight <= ESet[Child].Weight ) break; /* 找到了合适位置 */
65.         else  /* 下滤 X */
66.             ESet[Parent] = ESet[Child];
67.     }
68.     ESet[Parent] = X;
69. }
70.
71. void InitializeESet( LGraph Graph, Edge ESet )
72. { /* 将图的边存入数组 ESet，并且初始化为最小堆 */
73.     Vertex V;
74.     PtrToAdjVNode W;
75.     int ECount;
76.
77.     /* 将图的边存入数组 ESet */
78.     ECount = 0;
79.     for ( V=0; V<Graph->Nv; V++ )
80.         for ( W=Graph->G[V].FirstEdge; W; W=W->Next )
81.             if ( V < W->AdjV ) { /* 避免重复录入无向图的边，只收 V1<V2 的边 */
82.                 ESet[ECount].V1 = V;
83.                 ESet[ECount].V2 = W->AdjV;
84.                 ESet[ECount++].Weight = W->Weight;
85.             }
```

```
    /* 初始化为最小堆 */
    for ( ECount=Graph->Ne/2; ECount>=0; ECount-- )
        PercDown( ESet, ECount, Graph->Ne );
}

int GetEdge( Edge ESet, int CurrentSize )
{ /* 给定当前堆的大小 CurrentSize，将当前最小边位置弹出并调整堆 */

    /* 将最小边与当前堆的最后一个位置的边交换 */
    Swap( &ESet[0], &ESet[CurrentSize-1]);
    /* 将剩下的边继续调整成最小堆 */
    PercDown( ESet, 0, CurrentSize-1 );

    return CurrentSize-1; /* 返回最小边所在位置 */
}
/*-------------------- 最小堆定义结束 --------------------*/


int Kruskal( LGraph Graph, LGraph MST )
{ /* 将最小生成树保存为邻接表存储的图 MST，返回最小权重和 */
    WeightType TotalWeight;
```

```
107.     int ECount, NextEdge;
108.     SetType VSet; /* 顶点数组 */
109.     Edge ESet;    /* 边数组 */
110.
111.     InitializeVSet( VSet, Graph->Nv ); /* 初始化顶点并查集 */
112.     ESet = (Edge)malloc( sizeof(struct ENode)*Graph->Ne );
113.     InitializeESet( Graph, ESet ); /* 初始化边的最小堆 */
114.     /* 创建包含所有顶点但没有边的图。注意用邻接表版本 */
115.     MST = CreateGraph(Graph->Nv);
116.     TotalWeight = 0; /* 初始化权重和     */
117.     ECount = 0;      /* 初始化收录的边数 */
118.
119.     NextEdge = Graph->Ne; /* 原始边集的规模 */
120.     while ( ECount < Graph->Nv-1 ) {  /* 当收集的边不足以构成树时 */
121.         NextEdge = GetEdge( ESet, NextEdge ); /* 从边集中得到最小边的
    位置 */
122.         if (NextEdge < 0) /* 边集已空 */
123.             break;
124.         /* 如果该边的加入不构成回路，即两端结点不属于同一连通集 */
125.         if ( CheckCycle( VSet, ESet[NextEdge].V1, ESet[NextEdge].V2 )==
    true ) {
126.             /* 将该边插入 MST */
```

```
            InsertEdge( MST, ESet+NextEdge );
            TotalWeight += ESet[NextEdge].Weight; /* 累计权重 */
            ECount++; /* 生成树中边数加 1 */
        }
    }
    if ( ECount < Graph->Nv-1 )
        TotalWeight = -1; /* 设置错误标记，表示生成树不存在 */

    return TotalWeight;
}
```

3.5 拓扑排序

```
/* 邻接表存储 - 拓扑排序算法 */

bool TopSort( LGraph Graph, Vertex TopOrder[] )
{ /* 对 Graph 进行拓扑排序，TopOrder[]顺序存储排序后的顶点下标 */
    int Indegree[MaxVertexNum], cnt;
    Vertex V;
    PtrToAdjVNode W;
    Queue Q = CreateQueue( Graph->Nv );

    /* 初始化 Indegree[] */
    for (V=0; V<Graph->Nv; V++)
        Indegree[V] = 0;
```

```
13.
14.    /* 遍历图，得到 Indegree[] */
15.    for (V=0; V<Graph->Nv; V++)
16.        for (W=Graph->G[V].FirstEdge; W; W=W->Next)
17.            Indegree[W->AdjV]++; /* 对有向边<V, W->AdjV>累计终点的入度 */
18.
19.    /* 将所有入度为 0 的顶点入列 */
20.    for (V=0; V<Graph->Nv; V++)
21.        if ( Indegree[V]==0 )
22.            AddQ(Q, V);
23.
24.    /* 下面进入拓扑排序 */
25.    cnt = 0;
26.    while( !IsEmpty(Q) ){
27.        V = DeleteQ(Q); /* 弹出一个入度为 0 的顶点 */
28.        TopOrder[cnt++] = V; /* 将之存为结果序列的下一个元素 */
29.        /* 对 V 的每个邻接点 W->AdjV */
30.        for ( W=Graph->G[V].FirstEdge; W; W=W->Next )
31.            if ( --Indegree[W->AdjV] == 0 )/* 若删除 V 使得 W->AdjV 入度为
    0 */
32.                AddQ(Q, W->AdjV); /* 则该顶点入列 */
33.    } /* while 结束*/
```

```
34.
35.     if ( cnt != Graph->Nv )
36.         return false; /* 说明图中有回路, 返回不成功标志 */
37.     else
38.         return true;
39. }
```

Chapter4 排序算法

4.1 简单排序

```
1. void InsertionSort( ElementType A[], int N )
2. { /* 插入排序 */
3.     int P, i;
4.     ElementType Tmp;
5.
6.     for ( P=1; P<N; P++ ) {
7.         Tmp = A[P]; /* 取出未排序序列中的第一个元素*/
8.         for ( i=P; i>0 && A[i-1]>Tmp; i-- )
9.             A[i] = A[i-1]; /*依次与已排序序列中元素比较并右移*/
10.        A[i] = Tmp; /* 放进合适的位置 */
11.    }
12. }
```

4.2 Shell 排序

```
1. void ShellSort( ElementType A[], int N )
2. { /* 希尔排序 - 用 Sedgewick 增量序列 */
3.     int Si, D, P, i;
```

```
    ElementType Tmp;
    /* 这里只列出一小部分增量 */
    int Sedgewick[] = {929, 505, 209, 109, 41, 19, 5, 1, 0};

    for ( Si=0; Sedgewick[Si]>=N; Si++ )
        ; /* 初始的增量 Sedgewick[Si]不能超过待排序列长度 */

    for ( D=Sedgewick[Si]; D>0; D=Sedgewick[++Si] )
        for ( P=D; P<N; P++ ) { /* 插入排序*/
            Tmp = A[P];
            for ( i=P; i>=D && A[i-D]>Tmp; i-=D )
                A[i] = A[i-D];
            A[i] = Tmp;
        }
}
```

4.3 堆排序

```
void Swap( ElementType *a, ElementType *b )
{
    ElementType t = *a; *a = *b; *b = t;
}

void PercDown( ElementType A[], int p, int N )
{ /* 改编代码 4.24 的 PercDown( MaxHeap H, int p )    */
```

```
/* 将 N 个元素的数组中以 A[p]为根的子堆调整为最大堆 */
    int Parent, Child;
    ElementType X;

    X = A[p]; /* 取出根结点存放的值 */
    for( Parent=p; (Parent*2+1)<N; Parent=Child ) {
        Child = Parent * 2 + 1;
        if( (Child!=N-1) && (A[Child]<A[Child+1]) )
            Child++;  /* Child 指向左右子结点的较大者 */
        if( X >= A[Child] ) break; /* 找到了合适位置 */
        else  /* 下滤 X */
            A[Parent] = A[Child];
    }
    A[Parent] = X;
}

void HeapSort( ElementType A[], int N )
{ /* 堆排序 */
    int i;

    for ( i=N/2-1; i>=0; i-- )/* 建立最大堆 */
        PercDown( A, i, N );
```

```
30.
31.     for ( i=N-1; i>0; i-- ) {
32.         /* 删除最大堆顶 */
33.         Swap( &A[0], &A[i] ); /* 见代码 7.1 */
34.         PercDown( A, 0, i );
35.     }
36.}
```

4.4 归并排序

```
1. /* 归并排序 - 递归实现 */
2.
3. /* L = 左边起始位置, R = 右边起始位置, RightEnd = 右边终点位置*/
4. void Merge( ElementType A[], ElementType TmpA[], int L, int R, int RightE
   nd )
5. { /* 将有序的 A[L]~A[R-1]和 A[R]~A[RightEnd]归并成一个有序序列 */
6.     int LeftEnd, NumElements, Tmp;
7.     int i;
8.
9.     LeftEnd = R - 1; /* 左边终点位置 */
10.    Tmp = L;        /* 有序序列的起始位置 */
11.    NumElements = RightEnd - L + 1;
12.
13.    while( L <= LeftEnd && R <= RightEnd ) {
14.        if ( A[L] <= A[R] )
```

```
15.             TmpA[Tmp++] = A[L++]; /* 将左边元素复制到 TmpA */
16.         else
17.             TmpA[Tmp++] = A[R++]; /* 将右边元素复制到 TmpA */
18.     }
19.
20.     while( L <= LeftEnd )
21.         TmpA[Tmp++] = A[L++]; /* 直接复制左边剩下的 */
22.     while( R <= RightEnd )
23.         TmpA[Tmp++] = A[R++]; /* 直接复制右边剩下的 */
24.
25.     for( i = 0; i < NumElements; i++, RightEnd -- )
26.         A[RightEnd] = TmpA[RightEnd]; /* 将有序的 TmpA[]复制回 A[] */
27. }
28.
29. void Msort( ElementType A[], ElementType TmpA[], int L, int RightEnd )
30. { /* 核心递归排序函数 */
31.     int Center;
32.
33.     if ( L < RightEnd ) {
34.         Center = (L+RightEnd) / 2;
35.         Msort( A, TmpA, L, Center );            /* 递归解决左边 */
36.         Msort( A, TmpA, Center+1, RightEnd );    /* 递归解决右边 */
```

```
        Merge( A, TmpA, L, Center+1, RightEnd );  /* 合并两段有序序列 */
    }
}

void MergeSort( ElementType A[], int N )
{ /* 归并排序 */
    ElementType *TmpA;
    TmpA = (ElementType *)malloc(N*sizeof(ElementType));

    if ( TmpA != NULL ) {
        Msort( A, TmpA, 0, N-1 );
        free( TmpA );
    }
    else printf( "空间不足" );
}
```

```
/* 归并排序 – 循环实现 */
/* 这里 Merge 函数在递归版本中给出 */

/* length = 当前有序子列的长度*/
void Merge_pass( ElementType A[], ElementType TmpA[], int N, int length )
{ /* 两两归并相邻有序子列 */
    int i, j;
```

```
8.
9.     for ( i=0; i <= N-2*length; i += 2*length )
10.        Merge( A, TmpA, i, i+length, i+2*length-1 );
11.    if ( i+length < N ) /* 归并最后 2 个子列*/
12.        Merge( A, TmpA, i, i+length, N-1);
13.    else /* 最后只剩 1 个子列*/
14.        for ( j = i; j < N; j++ ) TmpA[j] = A[j];
15.}
16.
17.void Merge_Sort( ElementType A[], int N )
18.{
19.    int length;
20.    ElementType *TmpA;
21.
22.    length = 1; /* 初始化子序列长度*/
23.    TmpA = malloc( N * sizeof( ElementType ) );
24.    if ( TmpA != NULL ) {
25.        while( length < N ) {
26.            Merge_pass( A, TmpA, N, length );
27.            length *= 2;
28.            Merge_pass( TmpA, A, N, length );
29.            length *= 2;
```

```
        }
        free( TmpA );
    }
    else printf( "空间不足" );
}
```

4.5 快速排序

```
/* 快速排序 – 直接调用库函数 */

#include <stdlib.h>

/*----------------简单整数排序--------------------*/
int compare(const void *a, const void *b)
{ /* 比较两整数。非降序排列 */
    return (*(int*)a – *(int*)b);
}
/* 调用接口 */
qsort(A, N, sizeof(int), compare);
/*----------------简单整数排序--------------------*/


/*---------------- 一般情况下，对结构体 Node 中的某键值 key 排序 ----------------*/
struct Node {
```

```
    int key1, key2;
} A[MAXN];

int compare2keys(const void *a, const void *b)
{ /* 比较两种键值：按 key1 非升序排列；如果 key1 相等，则按 key2 非降序排列 */
    int k;
    if ( ((const struct Node*)a)->key1 < ((const struct Node*)b)->key1 )
        k = 1;
    else if ( ((const struct Node*)a)->key1 > ((const struct Node*)b)->key
1 )
        k = -1;
    else { /* 如果 key1 相等 */
        if ( ((const struct Node*)a)->key2 < ((const struct Node*)b)->key2
)
            k = -1;
        else
            k = 1;
    }
    return k;
}
/* 调用接口 */
qsort(A, N, sizeof(struct Node), compare2keys);
```

```
/*---------------- 一般情况下，对结构体 Node 中的某键值 key 排序 ----------------*/
/* 快速排序 */

ElementType Median3( ElementType A[], int Left, int Right )
{
    int Center = (Left+Right) / 2;
    if ( A[Left] > A[Center] )
        Swap( &A[Left], &A[Center] );
    if ( A[Left] > A[Right] )
        Swap( &A[Left], &A[Right] );
    if ( A[Center] > A[Right] )
        Swap( &A[Center], &A[Right] );
    /* 此时 A[Left] <= A[Center] <= A[Right] */
    Swap( &A[Center], &A[Right-1] ); /* 将基准 Pivot 藏到右边*/
    /* 只需要考虑 A[Left+1] … A[Right-2] */
    return  A[Right-1];  /* 返回基准 Pivot */
}

void Qsort( ElementType A[], int Left, int Right )
{ /* 核心递归函数 */
    int Pivot, Cutoff, Low, High;
```

```
21.
22.     if ( Cutoff <= Right-Left ) { /* 如果序列元素充分多，进入快排 */
23.         Pivot = Median3( A, Left, Right ); /* 选基准 */
24.         Low = Left; High = Right-1;
25.         while (1) { /*将序列中比基准小的移到基准左边，大的移到右边*/
26.             while ( A[++Low] < Pivot ) ;
27.             while ( A[--High] > Pivot ) ;
28.             if ( Low < High ) Swap( &A[Low], &A[High] );
29.             else break;
30.         }
31.         Swap( &A[Low], &A[Right-1] );   /* 将基准换到正确的位置 */
32.         Qsort( A, Left, Low-1 );    /* 递归解决左边 */
33.         Qsort( A, Low+1, Right );   /* 递归解决右边 */
34.     }
35.     else InsertionSort( A+Left, Right-Left+1 ); /* 元素太少，用简单排序 */
36. }
37.
38. void QuickSort( ElementType A[], int N )
39. { /* 统一接口 */
40.     Qsort( A, 0, N-1 );
41. }
```

4.6 基数排序

```
1. /* 基数排序 - 次位优先 */
```

```
2.
3. /* 假设元素最多有 MaxDigit 个关键字，基数全是同样的 Radix */
4. #define MaxDigit 4
5. #define Radix 10
6.
7. /* 桶元素结点 */
8. typedef struct Node *PtrToNode;
9. struct Node {
10.     int key;
11.     PtrToNode next;
12. };
13.
14. /* 桶头结点 */
15. struct HeadNode {
16.     PtrToNode head, tail;
17. };
18. typedef struct HeadNode Bucket[Radix];
19.
20. int GetDigit ( int X, int D )
21. { /* 默认次位 D=1, 主位 D<=MaxDigit */
22.     int d, i;
23.
```

```
    for (i=1; i<=D; i++) {
        d = X % Radix;
        X /= Radix;
    }
    return d;
}

void LSDRadixSort( ElementType A[], int N )
{ /* 基数排序 – 次位优先 */
    int D, Di, i;
    Bucket B;
    PtrToNode tmp, p, List = NULL;

    for (i=0; i<Radix; i++) /* 初始化每个桶为空链表 */
        B[i].head = B[i].tail = NULL;
    for (i=0; i<N; i++) { /* 将原始序列逆序存入初始链表 List */
        tmp = (PtrToNode)malloc(sizeof(struct Node));
        tmp->key = A[i];
        tmp->next = List;
        List = tmp;
    }
    /* 下面开始排序 */
```

```
46.     for (D=1; D<=MaxDigit; D++) { /* 对数据的每一位循环处理 */
47.         /* 下面是分配的过程 */
48.         p = List;
49.         while (p) {
50.             Di = GetDigit(p->key, D); /* 获得当前元素的当前位数字 */
51.             /* 从 List 中摘除 */
52.             tmp = p; p = p->next;
53.             /* 插入 B[Di]号桶尾 */
54.             tmp->next = NULL;
55.             if (B[Di].head == NULL)
56.                 B[Di].head = B[Di].tail = tmp;
57.             else {
58.                 B[Di].tail->next = tmp;
59.                 B[Di].tail = tmp;
60.             }
61.         }
62.         /* 下面是收集的过程 */
63.         List = NULL;
64.         for (Di=Radix-1; Di>=0; Di--) { /* 将每个桶的元素顺序收集入 List */
65.             if (B[Di].head) { /* 如果桶不为空 */
66.                 /* 整桶插入 List 表头 */
67.                 B[Di].tail->next = List;
```

```
                List = B[Di].head;
                B[Di].head = B[Di].tail = NULL; /* 清空桶 */
            }
        }
    }
    /* 将 List 倒入 A[]并释放空间 */
    for (i=0; i<N; i++) {
        tmp = List;
        List = List->next;
        A[i] = tmp->key;
        free(tmp);
    }
}
```

```
/* 基数排序 - 主位优先 */

/* 假设元素最多有 MaxDigit 个关键字，基数全是同样的 Radix */

#define MaxDigit 4
#define Radix 10

/* 桶元素结点 */
typedef struct Node *PtrToNode;
```

```
10. struct Node{
11.     int key;
12.     PtrToNode next;
13. };
14.
15. /* 桶头结点 */
16. struct HeadNode {
17.     PtrToNode head, tail;
18. };
19. typedef struct HeadNode Bucket[Radix];
20.
21. int GetDigit ( int X, int D )
22. { /* 默认次位 D=1, 主位 D<=MaxDigit */
23.     int d, i;
24.
25.     for (i=1; i<=D; i++) {
26.         d = X%Radix;
27.         X /= Radix;
28.     }
29.     return d;
30. }
31.
```

```
void MSD( ElementType A[], int L, int R, int D )
{ /* 核心递归函数: 对 A[L]...A[R]的第 D 位数进行排序 */
    int Di, i, j;
    Bucket B;
    PtrToNode tmp, p, List = NULL;
    if (D==0) return; /* 递归终止条件 */

    for (i=0; i<Radix; i++) /* 初始化每个桶为空链表 */
        B[i].head = B[i].tail = NULL;
    for (i=L; i<=R; i++) { /* 将原始序列逆序存入初始链表 List */
        tmp = (PtrToNode)malloc(sizeof(struct Node));
        tmp->key = A[i];
        tmp->next = List;
        List = tmp;
    }
    /* 下面是分配的过程 */
    p = List;
    while (p) {
        Di = GetDigit(p->key, D); /* 获得当前元素的当前位数字 */
        /* 从 List 中摘除 */
        tmp = p; p = p->next;
        /* 插入 B[Di]号桶 */
```

```
54.         if (B[Di].head == NULL) B[Di].tail = tmp;
55.         tmp->next = B[Di].head;
56.         B[Di].head = tmp;
57.     }
58.     /* 下面是收集的过程 */
59.     i = j = L; /* i, j 记录当前要处理的 A[]的左右端下标 */
60.     for (Di=0; Di<Radix; Di++) { /* 对于每个桶 */
61.         if (B[Di].head) { /* 将非空的桶整桶倒入 A[], 递归排序 */
62.             p = B[Di].head;
63.             while (p) {
64.                 tmp = p;
65.                 p = p->next;
66.                 A[j++] = tmp->key;
67.                 free(tmp);
68.             }
69.             /* 递归对该桶数据排序, 位数减 1 */
70.             MSD(A, i, j-1, D-1);
71.             i = j; /* 为下一个桶对应的 A[]左端 */
72.         }
73.     }
74. }
75.
```

```
void MSDRadixSort( ElementType A[], int N )
{ /* 统一接口 */
    MSD(A, 0, N-1, MaxDigit);
}
```

排序方法	平均时间复杂度	最坏情况下时间复杂度	额外空间复杂度	稳定性
简单选择排序	$O(N^2)$	$O(N^2)$	O(1)	不稳定
冒泡排序	$O(N^2)$	$O(N^2)$	O(1)	稳定
直接插入排序	$O(N^2)$	$O(N^2)$	O(1)	稳定
希尔排序	$O(N^d)$	$O(N^2)$	O(1)	不稳定
堆排序	O(NlogN)	O(NlogN)	O(1)	不稳定
快速排序	O(NlogN)	$O(N^2)$	O(logN)	不稳定
归并排序	O(NlogN)	O(NlogN)	O(N)	稳定
基数排序	O(P(N+B))	O(P(N+B))	O(N+B)	稳定

Chapter5 散列表

5.1 散列表定义

```
#define MAXTABLESIZE 100000 /* 允许开辟的最大散列表长度 */
typedef int ElementType;    /* 关键词类型用整型 */
typedef int Index;          /* 散列地址类型 */
typedef Index Position;     /* 数据所在位置与散列地址是同一类型 */
/* 散列单元状态类型，分别对应：有合法元素、空单元、有已删除元素 */
typedef enum { Legitimate, Empty, Deleted } EntryType;

typedef struct HashEntry Cell; /* 散列表单元类型 */
```

```
9. struct HashEntry{
10.     ElementType Data; /* 存放元素 */
11.     EntryType Info;   /* 单元状态 */
12. };
13.
14. typedef struct TblNode *HashTable; /* 散列表类型 */
15. struct TblNode {   /* 散列表结点定义 */
16.     int TableSize; /* 表的最大长度 */
17.     Cell *Cells;   /* 存放散列单元数据的数组 */
18. };
19.
20. int NextPrime( int N )
21. { /* 返回大于 N 且不超过 MAXTABLESIZE 的最小素数 */
22.     int i, p = (N%2)? N+2 : N+1; /*从大于 N 的下一个奇数开始 */
23.
24.     while( p <= MAXTABLESIZE ) {
25.         for( i=(int)sqrt(p); i>2; i-- )
26.             if ( !(p%i) ) break; /* p 不是素数 */
27.         if ( i==2 ) break; /* for 正常结束，说明 p 是素数 */
28.         else  p += 2; /* 否则试探下一个奇数 */
29.     }
30.     return p;
```

```
}

HashTable CreateTable( int TableSize )
{
    HashTable H;
    int i;

    H = (HashTable)malloc(sizeof(struct TblNode));
    /* 保证散列表最大长度是素数 */
    H->TableSize = NextPrime(TableSize);
    /* 声明单元数组 */
    H->Cells = (Cell *)malloc(H->TableSize*sizeof(Cell));
    /* 初始化单元状态为“空单元” */
    for( i=0; i<H->TableSize; i++ )
        H->Cells[i].Info = Empty;

    return H;
}
```

5.2 散列表查找和插入

```
Position Find( HashTable H, ElementType Key )
{
    Position CurrentPos, NewPos;
    int CNum = 0; /* 记录冲突次数 */
```

```
5.
6.    NewPos = CurrentPos = Hash( Key, H->TableSize ); /* 初始散列位置 */
7.    /* 当该位置的单元非空，并且不是要找的元素时，发生冲突 */
8.    while( H->Cells[NewPos].Info!=Empty && H->Cells[NewPos].Data!=Key )
      {
9.                                  /* 字符串类型的关键词需要 strcmp 函数!! */
10.       /* 统计 1 次冲突，并判断奇偶次 */
11.       if( ++CNum%2 ){ /* 奇数次冲突 */
12.           NewPos = CurrentPos + (CNum+1)*(CNum+1)/4; /* 增量为
      +[(CNum+1)/2]^2 */
13.           if ( NewPos >= H->TableSize )
14.               NewPos = NewPos % H->TableSize; /* 调整为合法地址 */
15.       }
16.       else { /* 偶数次冲突 */
17.           NewPos = CurrentPos - CNum*CNum/4; /* 增量为
      -(CNum/2)^2 */
18.           while( NewPos < 0 )
19.               NewPos += H->TableSize; /* 调整为合法地址 */
20.       }
21.   }
22.   return NewPos; /* 此时 NewPos 或者是 Key 的位置，或者是一个空单元的
      位置（表示找不到）*/
```

```
}

bool Insert( HashTable H, ElementType Key )
{
    Position Pos = Find( H, Key ); /* 先检查 Key 是否已经存在 */

    if( H->Cells[Pos].Info != Legitimate ) { /* 如果这个单元没有被占，说明 Key
可以插入在此 */
        H->Cells[Pos].Info = Legitimate;
        H->Cells[Pos].Data = Key;
        /*字符串类型的关键词需要 strcpy 函数!! */
        return true;
    }
    else {
        printf("键值已存在");
        return false;
    }
}
```

```
#define KEYLENGTH 15                /* 关键词字符串的最大长度 */
typedef char ElementType[KEYLENGTH+1]; /* 关键词类型用字符串 */
typedef int Index;                  /* 散列地址类型 */
/******** 以下是单链表的定义 ********/
```

```
typedef struct LNode *PtrToLNode;
struct LNode {
    ElementType Data;
    PtrToLNode Next;
};
typedef PtrToLNode Position;
typedef PtrToLNode List;
/******** 以上是单链表的定义 ********/

typedef struct TblNode *HashTable; /* 散列表类型 */
struct TblNode {   /* 散列表结点定义 */
    int TableSize; /* 表的最大长度 */
    List Heads;    /* 指向链表头结点的数组 */
};

HashTable CreateTable( int TableSize )
{
    HashTable H;
    int i;

    H = (HashTable)malloc(sizeof(struct TblNode));
    /* 保证散列表最大长度是素数，具体见代码 5.3 */
```

```
    H->TableSize = NextPrime(TableSize);

    /* 以下分配链表头结点数组 */
    H->Heads = (List)malloc(H->TableSize*sizeof(struct LNode));
    /* 初始化表头结点 */
    for( i=0; i<H->TableSize; i++ ) {
        H->Heads[i].Data[0] = '\0';
        H->Heads[i].Next = NULL;
    }

    return H;
}

Position Find( HashTable H, ElementType Key )
{
    Position P;
    Index Pos;

    Pos = Hash( Key, H->TableSize ); /* 初始散列位置 */
    P = H->Heads[Pos].Next; /* 从该链表的第 1 个结点开始 */
    /* 当未到表尾，并且 Key 未找到时 */
    while( P && strcmp(P->Data, Key) )
```

```
49.         P = P->Next;
50.
51.     return P; /* 此时 P 或者指向找到的结点，或者为 NULL */
52. }
53.
54. bool Insert( HashTable H, ElementType Key )
55. {
56.     Position P, NewCell;
57.     Index Pos;
58.
59.     P = Find( H, Key );
60.     if ( !P ) { /* 关键词未找到，可以插入 */
61.         NewCell = (Position)malloc(sizeof(struct LNode));
62.         strcpy(NewCell->Data, Key);
63.         Pos = Hash( Key, H->TableSize ); /* 初始散列位置 */
64.         /* 将 NewCell 插入为 H->Heads[Pos]链表的第 1 个结点 */
65.         NewCell->Next = H->Heads[Pos].Next;
66.         H->Heads[Pos].Next = NewCell;
67.         return true;
68.     }
69.     else { /* 关键词已存在 */
70.         printf("键值已存在");
```

```
71.         return false;
72.     }
73. }
74.
75. void DestroyTable( HashTable H )
76. {
77.     int i;
78.     Position P, Tmp;
79.
80.     /* 释放每个链表的结点 */
81.     for( i=0; i<H->TableSize; i++ ) {
82.         P = H->Heads[i].Next;
83.         while( P ) {
84.             Tmp = P->Next;
85.             free( P );
86.             P = Tmp;
87.         }
88.     }
89.     free( H->Heads ); /* 释放头结点数组 */
90.     free( H );        /* 释放散列表结点 */
91. }
```