考研数据结构代码

1. 线性表的结构体定义

1.1）顺序表的结构体定义

```
typedef struct Sqlist{
    int data[maxSize];
    int length;
}Sqlist;
```

1.2）考试中顺序表定义简写法

```
int A[maxSize];
    int n;
```

1.3）单链表的结构体定义

```
typedef struct LNode{
    int data;
    struct LNode *next;
}LNode;
```

1.4）双链表结构体定义

```
typedef struct DLNode {
    int data;
    struct DLNode *prior;
    struct DLNode *next;
}DLNode;
```

2. 顺序表的基本操作

2.1）初始化顺序表

```
void initList(Sqlist &L) {
    L.length = 0;
}
```

2.2）求指定位置元素

```
int getElem(Sqlist L, int p, int &e){
    if (p < 0 || p >= L.length) {
        return 0;
    }
    e = L.data[p];
    return 1;
}
```

2.3）插入数据元素

```
int insertElem(Sqlist &L, int p, int e) {
    if (p < 0 || p > L.length || L.length == maxSize) {
        return 0;
    }
    for (int i = L.length-1; i >= p; i--) {
        L.data[i+1] = L.data[i];
    }
    L.data[p] = e;
    ++(L.length);
    return 1;
}
```

2.4）按元素值查找

```
int findElem(Sqlist L, int e) {
    for (int i = 0; i < L.length; i++) {
        if (L.data[i] == e) {
            return i;
        }
    }
    return -1;
}
```

2.5）删除数据元素

```
int deleteElem(Sqlist &L, int p, int &e) {
    if (p < 0 || p >= L.length) {
        return 0;
    }
    e = L.data[p];
    for (int i = p; i < L.length-1; i++) {
        L.data[i] = L.data[i+1];
    }
    --(L.length);
    return 1;
```

2.6）顺序表的元素逆置

```
void reverse(Sqlist &L) {
    int i, j, temp;
    for (i = 0, j = L.length - 1; i < j; ++i, --j) {
        // 交换
        temp = L.data[i];
        L.data[i] = L.data[j];
        L.data[j] = temp;
    }
}
```

2.7）删除下标为i~j的数据元素

```
void deleteRange(Sqlist &L, int i, int j) {
    assert(0 <= i && 0 <= j && i < L.length && j < L.length);
    // 用j+1后面的元素去覆盖往前数第j-i+1个元素
    int delta = j - i + 1;
    for (int k = j+1; k < L.length; ++k) {
        L.data[k-delta] = L.data[k];
    }
    L.length -= delta;
}
```

2.8）Partition操作

```
void partition(Sqlist &L) {
    assert(L.length != 0);
    int p = L.data[0];
    int i = 0, j = L.length-1;
    while (i < j) {
        while (i < j && L.data[j] > p) --j;
        if (i < j) {
            L.data[i] = L.data[j];
            --i;
        }
```

```
        while (i < j && L.data[i] < p) ++i;
        if (i < j) {
            L.data[j] = L.data[i];
            --j;
        }
    }
    L.data[i] = p;
}
```

3. 单链表的基本操作

3.1）初始化单链表

```
void InitLinkList(LNode *list) {
    assert(list != NULL);
    list->next = NULL;
}
```

3.2）尾插法建立单链表

```
void createlistR(LNode *&list, int a[], int n) {
    LNode *s, *r; // s用来指向新申请的节点，r始终指向list的终端节点
    int i;
    list = (LNode*)malloc(sizeof(LNode));
    list->next = NULL;
    r = list;
    for (int i = 0; i < n; i++) {
        s = (LNode*)malloc(sizeof(LNode));
        s->data = a[i];
        r->next = s;  // 让当前的终端节点指向新来的节点
        r = r->next;  // 更新r，让r指向新的终端节点
    }
    r->next = NULL;
```

3.3）头插法建立单链表

```
void createlistF(LNode *&list, int a[], int n) {
    LNode *s;  // s用来指向新申请的节点
    list = (LNode*)malloc(sizeof(LNode));
    list->next = NULL;
    for (int i = 0; i < n; i++) {
        s = (LNode*)malloc(sizeof(LNode));
        s->data = a[i];
        s->next = list->next;
        list->next = s;
    }
}
```

3.4）合并递增单链表

```
void mergeR(LNode *A, LNode *B, LNode *&C) {
    // 由于不能改变了A，B自己的内容，所以得设定p，q来进行操作
    LNode *p = A->next;
    LNode *q = B->next;

    LNode *r;  // r始终指向C的终端节点
    C = A;  // 将A的头结点用来做C的头结点
    C->next = NULL;
    free(B);
    r = C;  // 初试的时候C也是终端节点
    while (p != NULL && q != NULL) {
        if (p->data <= q->data) {
            r->next = p; // 让当前的终端节点指向新的终端节点
            p = p->next; // p自然要往后挪一步
            r = r->next; // 更新r的指向，让r指向新的终端节点
        } else {
            r->next = q;
            q = q->next;
            r = r->next;
        }
    }

    // p还有剩余
    if (p != NULL) {
        r->next = p;
    }

    // q还有剩余
    if (q != NULL) {
        r->next = q;
    }
}
```

3.5）合并递减单链表

```
void mergeF(LNode *A, LNode *B, LNode *&C) {
    LNode *p = A->next;
```

```
    LNode *q = B->next;
    LNode *s;
    C = A;
    C->next = NULL;
    free(B);
    while (p != NULL && q != NULL) {
        if (p->data <= q->data) {
            // 这里就体现出s的作用了
            // 如果没有s，
            // 则直接p->next = C->next,
            // 那么改变了p的指向，p无法再往后挪了
            s = p;
            p = p->next;
            s->next = C->next;
            C->next = s;
        } else {
            s = q;
            q = q->next;
            q->next = C->next;
            C->next = s;
        }
    }

    // 由于头插，所以这里必须循环将每个剩余元素放到链表C的前面去
    while (p != NULL) {
        s = p;
        p = p->next;
        s->next = C->next;
        C->next = s;
    }

    while (q != NULL) {
        s = q;
        q = q->next;
        s->next = C->next;
        C->next = s;
    }
}
```

3.6）查找元素并删除

```
int findAndDelete(LNode *list, int x) {
    // 先找到该元素
    LNode *p;
    p = list;
    while (p->next != NULL) {
        if (p->next->data == x) {
            break;
        }
        p = p->next;
    }

    // 然后删除
    if (p->next == NULL) {
        return 0;
```

```
    } else {
        LNode *del;
        del = p->next;
        p->next = p->next->next;
        free(del);
        return 1;
    }
}
```

3.7）对于一个递增单链表，删除其重复的元素

```
void deleteDuplicate(LNode *list) {
    LNode *p, *q;  // q用来释放被删除的元素
    p = list->next;
    while (p->next != NULL) {
        if (p->data == p->next->data) {
            q = p->next;
            p->next = p->next->next;
            free(q);
        } else {
            p = p->next;
        }
    }
}
```

3.8）删除单链表中的最小值

```
void deleteMin(LNode *list) {
    // 这题的关键是要弄清楚需要哪些指针：
    // p用来扫描链表，pre指向p的前驱
    // minp用来指向最小值节点，premin用来指向minp的前驱
    LNode *p = list->next, *pre = list, *minp = p, *premin = pre;
    while (p != NULL) {
        if (p->data < minp->data) {
            minp = p;
            premin = pre;
        }
        pre = p;
        p = p->next;
    }
    premin->next = minp->next;
    free(minp);
}
```

3.9）单链表的原地逆置

```
// 将L->NULL设置为空，然后将剩余的节点一一用头插法插入到L中
void reverseList(LNode *list) {
    LNode *p = list->next, *q;
    list->next = NULL;  // 断开头节点
    while (p != NULL) {
        q = p->next;  // 首先得让q临时保存一下p的下一个节点，待会儿还得用呢
        p->next = list->next;  // 头插法
        list->next = p;
        p = q;  // 拿回下一个节点
    }
}
```

3.10）将一个数据域为整数的单链表分割为两个分别为奇数和偶数的链表

```
void split2(LNode *A, LNode *&B) {
    // p用来扫描，但是指向的是要删除节点的前驱
    // q用来临时保存要删除的节点
    // r用来指向B中的终端节点
    LNode *p, *q, *r;
    B = (LNode*)malloc(sizeof(LNode));
    B->next = NULL; // 每申请一个新的节点，都将指针域设置为空，这样可以避免出事儿
    p = A;
    r = B;
    while (p != NULL) {
        if (p->next->data%2 == 0) {
            q = p->next;
            p->next = p->next->next;
            r->next = q;
            r = q;
            q->next = NULL;  // 不写这句话绝对出事
        } else {
            p = p->next;
        }
    }
}
```

3.11）逆序打印单链表中的节点

```
// 逆序，入栈不就逆序了吗，所以可以考虑递归打印
void reprintList(LNode *list) {
    if (list != NULL) {
        reprintList(list->next);
        printf("%d ", list->data);
    }
}
```

3.12）查找链表中倒数第k个节点

```
int findkLast(LNode *list, int k) {
    LNode *p, *q;
    p = list->next;
    q = list;
    int i = 1;
    while (p != NULL) {
        p = p->next;
        ++i;  // 这里的计数器i自己画图给弄清楚
        if (i > k) q = q->next;
    }

    if (q == list) return 0;  // 链表的节点不到k个
    else {
        printf("%d ", p->data);
        return 1;
    }
```

4. 双链表的基本操作

4.1）尾插法建立双链表

```
void createDListR(DLNode *&list, int a[], int n) {
    // 准备工作...
    DLNode *s, *r;  // s指向新申请的节点，r指向终端节点
    list = (DLNode*)malloc(sizeof(DLNode));  // 创建头结点
    list->prior = NULL;
    list->next = NULL;
    r = list;  // 初始状态，r指向头结点

    // 开始插入元素
    int i;
    for (int i = 0; i < n; i++) {
        s = (DLNode*)malloc(sizeof(DLNode));
        s->data = a[i];

        // 最关键的三句话
        r->next = s;
        s->prior = r;
        r = s;
    }
    r->next = NULL;  // 如果不加这句话，一定会出事的
}
```

4.2）查找结点的算法

```
DLNode* findNode(DLNode *list, int x) {
    DLNode *p = list->next;
    while (p != NULL) {
        if (p->data == x) {
            break;
        }
        p = p->next;
    }
    return p;
}
```

4.3）插入结点的算法

```
int insertNode(DLNode *&list, int index, int e) {
    if (index < 0) return 0;
    DLNode *p = list;
    for (int i = 0; i < index; ++i) {
        p = p->next;
    }

    DLNode *s = (DLNode*)malloc(sizeof(DLNode));
    s->data = e;
    // 最关键的四句话:将新节点插入到p指向节点的后面
    s->next = p->next;
    s->prior = p;
    p->next = s;
    s->next->prior = s;
    return 1;
}
```

4.4）删除结点的算法

```
int deleteNode(DLNode *&list, int index) {
    DLNode *p = list;
    for (int i = 0; i < index; ++i) {
        p = p->next;
    }
    if (p->next == NULL) return 0;

    // 最关键的四句话：删除p的后继结点
    DLNode *del;
    del = p->next;
    p->next = del->next;
    if (del->next != NULL) {  // 不加这个判断也是会出事的
        del->next->prior = p;
    }
    free(del);
    return 1;
}
```

5. 栈和队列的结构体定义

5.1）顺序栈的定义

```
typedef struct SqStack{
    int data[maxSize];
    int top;
} SqStack;
```

5.2）链栈结点定义

```
typedef struct LNode{
    int data;
    struct LNode *next;
} LNode;
```

5.3）顺序队列的定义

```
typedef struct {
    int data[maxSize];
    int front;  // 队首指针
    int rear;  // 队尾指针
}SqQueue;
```

5.4）链队定义

（5.4.1）队结点类型定义

```
typedef struct QNode {
    int data;  // 数据域
    struct QNode *next;   // 指针域
}QNode;
```

（5.4.2）链队类型定义

```
typedef struct {
    QNode *front;  // 队头指针
    QNode *rear;  // 队尾指针
}LiQueue;
```

6. 顺序栈的基本操作

6.1）顺序栈的初始化

```
void initStack(SqStack &st) {
    st.top = -1;
}
```

6.2）判断栈空

```
int isEmpty(SqStack st) {
    return st.top == -1;
}
```

6.3）进栈代码

```
int push(SqStack &st, int x) {
    if (st.top == maxSize) return 0;
    st.data[++st.top] = x;
    return 1;
}
```

6.4）出栈代码

```
int pop(SqStack &st, int &x) {
    if (st.top == -1) return 0;
    x = st.data[st.top--];
    return 1;
}
```

6.5）考试中顺序栈用法的简洁写法

```
int stack[maxSize]; int top = -1;  // 这两句话连定义带初始化都有了
stack[++top] = x;  // 一句话实现进栈
x = stack[top--];  // 一句话实现出栈
```

7. 链栈的基本操作

7.1）链栈的初始化

```
void initStack(LNode *&lst) {
    lst = (LNode*)malloc(sizeof(LNode));
    lst->next = NULL;  // 还是老规矩：申请新的节点后一定要为其指针域设置为空
}
```

7.2）判断栈空

```
int isEmpty(LNode *lst) {
    return lst->next == NULL;
}
```

7.3）进栈代码

```
void push(LNode *lst, int x) {
    // 步骤一：申请新的节点，存放x的值
    // 步骤二：头插法将新的节点插入链栈中

    LNode *p = (LNode*)malloc(sizeof(LNode));
    p->next = NULL;

    p->data = x;
    p->next = lst->next;
    lst->next = p;
}
```

7.4）出栈代码

```
int pop(LNode *lst, int &x) {
    // 步骤一：判空
    // 步骤二：删除节点

    if (lst->next == NULL) return 0;

    x = lst->data;
    LNode *p;
    p = lst->next;
    lst->next = lst->next->next;
    free(p);
    return 1;
}
```

8. 顺序队列的基本操作

8.1）顺序队列的初始化

```
void initQueue(SqQueue &qu) {
    qu.front = qu.rear = 0;  // 队首和队尾指针重合，并且指向0
}
```

8.2）判断队空

```
int isQueueEmpty(SqQueue qu) {
    return qu.front == qu.rear;  // 只要重合，即为空队
}
```

8.3）进队算法

```
int enQueue(SqQueue &qu, int x) {
    if ((qu.rear+1)%maxSize == qu.front) return 0;  // 队满则不能入队
    qu.rear = (qu.rear+1)%maxSize;  // 若未满，则先移动指针
    qu.data[qu.rear] = x;  // 再放入元素
    return 1;
}
```

8.4）出队算法

```
int deQueue(SqQueue &qu, int &x) {
    if (qu.front == qu.rear)  return 0;  // 若队空，则不能删除
    qu.front = (qu.front+1)%maxSize;  // 若队不空，则先移动指针
    x = qu.data[qu.front];  // 再取出元素
    return 1;
}
```

9. 链队的基本操作

9.1）链队的初始化

```
void initQueue(LiQueue *&lqu) {
    lqu = (LiQueue*)malloc(sizeof(LiQueue));
    lqu->front = lqu->rear = nullptr;
}
```

9.2）判断队空算法

```
int isQueueEmpty(LiQueue *lqu) {
    return (lqu->rear == nullptr || lqu->front == nullptr);  // 注意有两个
}
```

9.3）入队算法

```
void enQueue(LiQueue *lqu, int x) {
    QNode *p;
    p = (QNode*)malloc(sizeof(QNode));
    p->data = x;
    p->next = nullptr;
    if (lqu->rear == nullptr) {  // 若队列为空，则新结点是队首结点，也是队尾结点
        lqu->front = lqu->rear = p;
    } else {
        lqu->rear->next = p;  // 将新结点链接到队尾，rear指向它
        lqu->rear = p;
    }
}
```

9.4）出队算法

```
int deQueue(LiQueue *lqu, int &x) {
    QNode *p;
    if (lqu->rear == nullptr) {  // 队空不能出队
        return 0;
    } else {
        p = lqu->front;
    }
    if (lqu->front == lqu->rear) {  // 队列中只有一个结点时的出队操作需要特殊处理
        lqu->front = lqu->rear = nullptr;
    } else {
        lqu->front = lqu->front->next;
    }
    x = p->data;
    free(p);
    return 1;
}
```

10. 串的结构体定义

10.1）定长顺序存储表示

```
typedef struct {
    char str[maxSize+1];  // 多出一个'\0'作为结束标记
    int length;
}Str;
```

10.2）变长分配存储表示

```
typedef struct {
    char *ch;  // 指向动态分配存储区首地址的字符指针
    int length;  // 串长度
}Str;
```

11. 串的基本操作

11.1）赋值操作

```
int strassign(Str& str, char* ch) {
    if (str.ch) {
        free(str.ch);  // 释放原空间
    }
    int len = 0;
    char *c = ch;
    while (*c) {  // 求ch串的长度
        ++len;
        ++c;
    }
    if (len == 0) {  // 如果ch为空串，则直接返回空串
        str.ch = nullptr;
        str.length = 0;
        return 1;
    } else {
        str.ch = (char*)malloc(sizeof(char)*(len+1));  // 取len+1是为了'\0'
        if (str.ch == nullptr) return 0;  // 分配失败
        else {
            c = ch;
            for (int i = 0; i <= len; ++i, ++c) {
                str.ch[i] = *c;   // 之所以取<=也是为了将'\0'复制过去
            }
            str.length = len;
            return 1;
        }
    }
}
```

11.2）取串长度操作

```
int strlength(Str str) {
    return str.length;
}
```

11.3）串比较操作

```
int strcompare(Str s1, Str s2) {
    for (int i = 0; i < s1.length && i < s2.length; ++i) {
        if (s1.ch[i] != s2.ch[i]) return s1.ch[i] - s2.ch[i];
    }
    return s1.length - s2.length;
}
```

11.4）串连接操作

```
int concat(Str &str, Str str1, Str str2) {
    if (str.ch) {
        free(str.ch);  // 释放原空间
        str.ch = nullptr;
    }
    str.ch = (char*)malloc(sizeof(char)*(str1.length+str2.length));
    if (str.ch == nullptr) return 0;
    int i = 0;
    while (i < str1.length) {
        str.ch[i] = str1.ch[i];
        ++i;
    }
    int j = 0;
    while (j <= str2.length) {
        str.ch[i+j] = str2.ch[j];  // 之所以取<=也是为了将'\0'复制过去
        ++j;
    }
    str.length = str1.length + str2.length;
    return 1;
}
```

11.5）求子串操作

```
int substring(Str& substr, Str str, int pos, int len) {
    if (pos<0 || pos>=str.length || len<0 || len>str.length-pos) return 0;
    if (substr.ch) {
        free(substr.ch);
        substr.ch = nullptr;
    }
    if (len == 0) {
        substr.ch = nullptr;
        substr.length = 0;
        return 1;
    } else {
        substr.ch = (char*)malloc(sizeof(char)*(len+1));
        int i = pos;
        int j = 0;
        while (i<pos+len) {
            substr.ch[j++] = str.ch[i++];
        }
        substr.ch[j] = '\0';
```

```
        substr.length = len;
        return 1;
    }
}
```

11.6）串清空操作

```
int clearstring(Str& str) {
    if (str.ch) {
        free(str.ch);
        str.ch = nullptr;
    }
    str.length = 0;
    return 1;
}
```

12. 串的模式匹配

12.1）简单模式匹配算法

```
int index(Str str, Str substr) {
    int i = 1, j = 1, k = i;
    while (i<=str.length && j<=substr) {
        if (str.ch[i] == substr[j]) {
            ++i;
            ++j;
        } else {
            j = 1;
            i = ++k;  // 匹配失败，i从主串的下一位置开始，k中记录了上一次的起始位置
        }
    }
    if (j>substr.length) return k;
    else return 0;
}
```

12.2）KMP算法

> 注：这里的代码全部都认为下标从0开始

（12.2.1）求next数组的代码

```
void GetNext(char* p,int next[])
{
    int pLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < pLen - 1)
    {
```

```
        //p[k]表示前缀，p[j]表示后缀
        if (k == -1 || p[j] == p[k])
        {
            ++k;
            ++j;
            next[j] = k;
        }
        else
        {
            k = next[k];
        }
    }
}
```

(12.2.2) 求nextval数组的代码

```
//优化过后的next 数组求法
void GetNextval(char* p, int next[])
{
    int pLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < pLen - 1)
    {
        //p[k]表示前缀，p[j]表示后缀
        if (k == -1 || p[j] == p[k])
        {
            ++j;
            ++k;
            //较之前next数组求法，改动在下面4行
            if (p[j] != p[k])
                next[j] = k;   //之前只有这一行
            else
                //因为不能出现p[j] = p[ next[j ]]，所以当出现时需要继续递归，k =
next[k] = next[next[k]]
                next[j] = next[k];
        }
        else
        {
            k = next[k];
        }
    }
}
```

(12.2.3) KMP算法

```
int KMP(char* s, char* p, int next[])
{
    int i = 0;
    int j = 0;
    int sLen = strlen(s);
```

```
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
    {
        //②如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++
        if (j == -1 || s[i] == p[j])
        {
            i++;
            j++;
        }
        else
        {
            //②如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j =
next[j]
            //next[j]即为j所对应的next值
            j = next[j];
        }
    }
    if (j == pLen)
        return i - j;  // 匹配成功，则返回匹配的位置
    else
        return -1;
}
```

13. 二叉树的结点定义

13.1）普通二叉树的链式存储结点定义

```
typedef struct BTNode {
    char data;   // 这里默认结点data域为char类型

    struct BTNode *lchild;  // 左儿子
    struct BTNode *rchild;  // 右儿子
}BTNode;
```

13.2）线索二叉树的结点定义

```
// 线索二叉树
typedef struct TBTNode{
    char data;
    int ltag, rtag;
    struct TBTNode *lchild;
    struct TBTNode *rchild;
}TBTNode;
```

13. 二叉树的遍历算法

13.1）先序遍历（递归）

```
void preorder(BTNode *p) {
    if (p != nullptr) {  // 一定要记得判空
        printf("%c ", p->data);
        preorder(p->lchild);
        preorder(p->rchild);
    }
}
```

13.2）中序遍历（递归）

```
void inorder(BTNode *p) {
    if (p != nullptr) {
        inorder(p->lchild);
        printf("%c ", p->data);
        inorder(p->rchild);
    }
}
```

13.3）后序遍历（递归）

```
void postorder(BTNode *p) {
    if (p != nullptr) {
        postorder(p->lchild);
        postorder(p->rchild);
        printf("%c ", p->data);
    }
}
```

13.4）层序遍历

```
void level(BTNode *p) {
    BTNode *que[maxSize];
    BTNode *q = nullptr;
    int front = 0, rear = 0;  // 定义一个循环队列

    if (p != nullptr) {
        rear = (rear+1) % maxSize;
        que[rear] = p;  // 让根节点入队
        while (front != rear) {  // 只要队列不空，则进行循环
            front = (front+1) % maxSize;
            q = que[front];  // 队头元素出队
            printf("%c\n", q->data);  // 访问队头元素

            if (q->lchild) {  // 左子树存在，则左子树根节点入队
                rear = (rear+1) % maxSize;
                que[rear] = q->lchild;
            }
            if (q->rchild) {  // 右子树存在，则右子树根节点入队
```

```
                rear = (rear+1) % maxSize;
                que[rear] = q->rchild;
            }
        }
    }
```

13.5）先序遍历（非递归）

```
void preorderNonrecursion(BTNode *bt) {
    if (bt != nullptr) {
        BTNode *Stack[maxSize];
        int top = -1;  // 定义人工栈
        Stack[++top] = bt;  // 根节点入栈
        BTNode *p;
        while (top != -1) {  // 判断不空
            p = Stack[top--];  // 出栈 并完成一次访问
            printf("%c\n", p->data);
            if (p->rchild != nullptr) {   // 记得，先序遍历一定是先右孩子，再左孩子
                Stack[++top] = p->rchild;
            }
            if (p->lchild != nullptr) {
                Stack[++top] = p->lchild;
            }
        }
    }
}
```

13.6）中序遍历（非递归）

```
void inorderNonrecursion(BTNode *bt) {
    BTNode *Stack[maxSize];
    int top = -1;
    BTNode *p = bt;
    if (bt != nullptr) {
        while (top != -1 || p != nullptr) {
            while (p != nullptr) {
                Stack[++top] = p;
                p = p->lchild;
            }

            if (top != -1) {
                p = Stack[top--];
                printf("%c\n", p->data);
                p = p->rchild;
            }
        }
    }
}
```

13.7）后序遍历（非递归）

```
void postorderNonrecursion(BTNode *bt) {
    if (bt != nullptr) {
        BTNode *Stack1[maxSize]; int top1 = -1;
        BTNode *Stack2[maxSize]; int top2 = -1;  // 定义两个栈
        BTNode *p;
        Stack1[++top1] = bt;
        while (top1 != -1) {
            p = Stack1[top1--];
            Stack2[++top2] = p;  // 注意这里与先序的区别，放入栈2中即可
            if (p->lchild) {
                Stack1[++top1] = p->lchild;
            }
            if (p->rchild) {
                Stack1[++top1] = p->rchild;
            }
        }
        // 这时候循环结束，则会将逆后序遍历的结果都存放到了栈2中
        // 所以对栈2进行输出即可得到后序遍历的结果
        while (top2 != -1) {
            p = Stack2[top2--];
            printf("%c\n", p->data);
        }
    }
}
```

14. 图的存储结构

14.1）邻接矩阵的结点定义

```
typedef struct {
    int no;  // 顶点编号
    char info;  // 顶点的其他信息，这里默认为char型。这一句一般在题目中很少用到，因此题目不做特殊要求可以不写
}VertexType;  // 顶点类型

typedef struct {
    int edges[maxSize][maxSize];  // 临街矩阵定义，如果是有权图，则在此句中将int改为float
    int n, e;  // 分别为定点数和边数
    VertexType vex[maxSize];  // 存放节点信息
}MGraph;  // 图的邻接矩阵类型
```

14.2）邻接表的结点定义

```
typedef struct ArcNode{
    int adjvex;  // 该边所指向的节点的位置
    struct ArcNode *nextarc;  // 指向下一条边的指针
    int info;  // 该边的相关信息（如权值）
```

```
}ArcNode;

typedef struct {
    char data;  // 定点信息
    ArcNode *firstarc;  // 指向第一条边的指针
}VNode;

typedef struct{
    VNode adjlist[maxSize];  // 邻接表
    int n, e;  // 定点数和边数
}AGraph;  // 图的邻接表类型
```

15. 图的遍历方式

15.1）DFS

```
int visit[maxSize];
void DFS(AGraph *G, int v) {
    ArcNode *p;
    visit[v] = 1;
    cout << v << endl;
    p = G->adjlist[v].firstarc;  // 让p指向顶点v的第一条边
    while (p != nullptr) {
        if (visit[p->adjvex] == 0) {
            DFS(G, p->adjvex);
            p = p->nextarc;
        }
    }
}
```

15.2）BFS

```
void BFS(AGraph *G, int v) {
    ArcNode *p;
    int que[maxSize], front = 0, rear = 0;  // 定义一个队列
    int j;
    cout << v << endl;
    visit[v] = 1;
    rear = (rear+1)%maxSize;  // 入队
    que[rear] = v;
    while (front != rear) {
        front = (front+1)%maxSize;  // 顶点出队
        j = que[front];
        p = G->adjlist[j].firstarc;  // p指向出队顶点j的第一条边
        while (p != nullptr) {  // 将p的所有邻接点未被访问的入队
            if (visit[p->adjvex] == 0) {
                cout << p->adjvex << endl;
                rear = (rear+1)%maxSize;
                que[rear] = p->adjvex;
            }
```

```
            p = p->nextarc;
        }
    }
}
```

16. 最小（代价）生成树

16.1）Prim算法

```
void Prim(MGraph g, int v0, int &sum) {
    int lowcost[maxSize], vset[maxSize], v;
    int i, j, k, min;
    v = v0;
    for (i = 0; i < g.n; ++i) {
        lowcost[i] = g.edges[v0][i];
        vset[i] = 0;
    }
    vset[v0] = 1;  // 将v0并入树中
    sum = 0;  // sum清零用来累计树的权值
    for (i = 0; i < g.n-1; ++i) {
        min = INF;  // INF是一个已经定义的比图中所有边权值都大的常量
        // 下面这个循环用于选出候选边中的最小者
        for (j = 0; j < g.n; ++j) {
            if (vset[j] == 0 && lowcost[j] < min) {  // 选出当前生成树其他顶点到最短边中最短的一条
                min = lowcost[j];
                k = j;
            }
            vset[k] = 1;
            v = k;
            sum += min;  // 这里用sum记录了最小生成树的权值
            // 下面这个循环以刚进入的顶点v为媒介更新候选边
            for (j = 0; j < g.n; ++j) {
                if (vset[j] == 0 && g.edges[v][j] < lowcost[j]) {
                    lowcost[j] = g.edges[v][j];
                }
            }
        }
    }
}
```

16.2）Kruskal算法

```
typedef struct {
    int a, b;  // a和b为一条边所连的两个顶点
    int w;  // 边的权值
}Road;
Road road[maxSize];
int v[maxSize];  // 定义并查集数组
int getRoot(int a) {
    while (a != v[a])  a = v[a];  // 在并查集中查找根结点的函数
```

```
    return a;
}

void Kruskal(MGraph g, int &sum, Road road[]) {
    int i, N, E, a, b;
    N = g.n;
    E = g.e;
    sum = 0;
    for (i = 0; i < N; ++i)  v[i] = i;
    sort(road, E);   // 对road数组中的E条边按其权值从小到大排序，假设该函数已定义好
    for (i = 0; i < E; ++i) {
        a = getRoot(road[i].a);
        b = getRoot(road[i].b);
        if (a != b) {
            v[a] = b;
            sum += road[i].w;
        }
    }
}
```

17. 最短路径算法

17.1）Dijkstra算法

```
void Dijkstra(MGraph g, int v, int dist[], int path[]) {
    int set[maxSize];
    int min, i, j, u;
    // 从这句开始对各数组进行初始化
    for (i = 0; i < g.n; ++i) {
        dist[i] = g.edges[v][i];
        set[i] = 0;
        if (g.edges[v][i] < INF)
            path[i] = v;
        else {
            path[i] = -1;
        }
    }
    set[v] = 1; path[v] = -1;
    //  初始化结束
    // 关键操作开始
    for (i = 0; i < g.n-1; ++i) {
        min = INF;
        // 这个循环每次从剩余顶点中选出一个顶点，通往这个顶点的路径在通往所有剩余顶点的路径中是长度最短的
        for (j = 0; j < g.n; ++j) {
            if (set[j] == 0 && dist[j] < min) {
                u = j;
                min = dist[j];
            }
        }
        set[u] = 1;  // 将选出的顶点并入最短路径中
        // 这个循环以刚并入的顶点作为中间点，对所有通往剩余顶点的路径进行检测
        for (j = 0; j < g.n; ++j) {
            // 这个if语句判断顶点u的加入是否为出现通往顶点j的更短的路径
```

```
            if (set[j] == 0 && dist[u]+g.edges[u][j] < dist[j]) {
                dist[j] = dist[u] + g.edges[u][j];
                path[j] = u;
            }
        }
    }
}
```

17.2）Floyd算法

```
void Floyd(MGraph g, int Path[][maxSize]) {
    int i, j, k;
    int A[maxSize][maxSize];
    // 这个双循环对数组A[][]和Path[][]进行了初始化
    for (i = 0; i < g.n; ++i) {
        for (j = 0; j < g.n; ++j) {
            A[i][j] = g.edges[i][j];
            Path[i][j] = -1;
        }
    }
    // 下面三层循环是主要操作，完成了以k为中间点对所有的顶点对{i, }进行检测和修改
    for (k = 0; k < g.n; ++k) {
        for (i = 0; i < g.n; ++i) {
            for (j = 0; j < g.n; ++j) {
                if (A[i][j] > A[i][k] + A[k][j]) {
                    A[i][j] = A[i][k] + A[k][j];
                    Path[i][j] = k;
                }
            }
        }
    }
}
```

18. 拓扑排序

18.1）拓扑排序中对邻接表表头结构的修改

```
typedef struct {
    char data;
    int count;  // 此处为新增代码，count用来统计顶点当前的入度
    ArcNode *firstarc;
}VNode;
```

18.2）拓扑排序算法

```
int TopSort(AGraph *G) {
    int i, j, n = 0;
    int stack[maxSize], top = -1;  // 定义并初始化栈
```

```
    ArcNode *p;
    // 这个循环将图中入度为0的顶点入栈
    for (i = 0; i < G->n; ++i) {  // 图中的顶点从0开始编号
        if (G->adjlist[i].count == 0) {
            stack[++top] = i;
        }
    }
    // 关键操作开始
    while (top != -1) {
        i = stack[top--];  // 顶点出栈
        ++n;  // 计数器加1，统计当前顶点
        cout << i << " ";  // 输出当前顶点
        p = G->adjlist[i].firstarc;
        // 这个循环实现了将所有由此顶点引出的边所指向的顶点的入度都减少1
        // 并将这个过程中入度变为0的顶点入栈
        while (nullptr != p) {
            j = p->adjvex;
            --(G->adjlist[j].count);
            if (G->adjlist[j].count == 0)
                stack[++top] = j;
            p = p->nextarc;
        }
    }
    // 关键操作结束
    return n == G->n;
}
```

19. 排序算法

19.1）直接插入排序

```
void InsertSort(int a[], int n) {
    int i, j;
    int temp;
    for (i = 1; i < n; ++i) {
        if (a[i] < a[i-1]) {
            temp = a[i];
            for (j = i; a[j-1] > temp && j >= 1; --j) {
                a[j] = a[j-1];
            }
            a[j] = temp;
        }
    }
}
```

19.2）折半插入排序

```
void BinaryInsertSort(int R[],int n) {
    int i,j,low,mid,high,temp;
    for(i=1; i<n; i++) {
        low=0;
```

```
        high=i-1;
        temp=R[i];
        // 下面的折半是为了在i元素的前面查找到合适的插入位置
        while(low <= high) {
            mid=(low+high)/2;
            if(R[mid]>temp) {
                high=mid-1;
            } else {
                low=mid+1;
            }
        }
        for(j=i-1;j>=high+1;j--) {
            R[j+1]=R[j];
        }
        R[j+1]=temp;
    }
}
```

19.3）希尔排序

```
// 更小间隔的排序并没有破坏之前的排序后的相对性，但是排序的结果可能不一样了
void ShellSort(int arr[], int n) {
    for (int d = n/2; d > 0; d /= 2) {
        // 下面的内容就是把上面插入排序中所有的1改为d，一共有5处修改，别忘了--j也要改
        for (int i = d; i < n; ++i) {
            int temp = arr[i];
            int j;
            for (j = i; j >= d && temp < arr[j-d]; j -= d) {
                arr[j] = arr[j-d];
            }
            arr[j] = temp;
        }
    }
}
```

19.4）起泡排序

```
void BubbleSort(int arr[], int n) {
    int temp;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n-i-1; ++j) {
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

19.5）快速排序

（19.5.1）两路快排

```
void __quickSort(int nums[], int l, int r) {
    if (l >= r) return;
    swap(nums[l], nums[rand()%(r-l+1)+l]);
    int pivot = nums[l];

    int low = l+1, high = r;
    while (low <= high) {
        if (nums[low] > pivot && nums[high] < pivot) {
            swap(nums[low++], nums[high--]);
        }
        if (low <= high && nums[low] <= pivot) ++low;  // 一定要带等号
        if (low <= high && nums[high] >= pivot) --high;
    }
    swap(nums[l], nums[high]);
    int p = high;

    __quickSort(nums, l, p-1);
    __quickSort(nums, p+1, r);
}
```

（19.5.2）三路快排

```
// 三路快排(性能最好)
template <typename T>
void __quickSort3Ways(T arr[], int l, int r) {
    if (l >= r) return;

    // partition
    swap(arr[l], arr[rand()%(r-l+1)+l]);
    T v = arr[l];
    int lt = l;  // arr[l+1...lt] < v
    int gt = r+1;  // arr[gt...r] > v
    int i = l+1;  // arr[lt+1...i) == v
    while (i < gt) {
        if (arr[i] < v) {
            swap(arr[++lt], arr[i++]);
        } else if (arr[i] > v) {
            swap(arr[--gt], arr[i]);
        } else {
            ++i;
        }
    }

    swap(arr[l], arr[lt]);
    __quickSort3Ways(arr, l, lt-1);
    __quickSort3Ways(arr, gt, r);
}
```

19.6）简单选择排序

```
void SelectSort(int arr[], int n) {
    int temp;
    for (int i = 0; i < n; ++i) {
        for (int j = i+1; j < n; ++j) {
            if (arr[i] > arr[j]) {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

19.7）堆排序

(19.7.1）shiftDown操作

```
// 在原地堆排序中，元素是从数组下标0的位置开始存储的，因此i的左孩子应该为2*i+1
template <typename T>
void __shiftDown(T arr[], int n, int k) {
    while (2*k+1 < n) {
        int j = 2*k + 1;
        if (j + 1 < n && arr[j+1] > arr[j]) {
            ++j;
        }
        if (arr[k] >= arr[j]) break;
        swap(arr[j], arr[k]);
        k = j;
    }
}
```

(19.7.2）堆排序代码

```
template <typename T>
void heapSort3(T arr[], int n) {
    // heapify
    for (int i = (n-1)/2; i >= 0; --i) {
        __shiftDown(arr, n, i);
    }

    for (int i = n-1; i > 0; --i) {
        swap(arr[0], arr[i]);
        __shiftDown(arr, i, 0);
    }
}
```

19.8）归并排序

(19.8.1) merge操作

```
// 将arr[l...mid]和arr[mid+1...r]进行归并
template <typename T>
void __merge(T arr[], int l, int mid, int r) {
    T aux[r-l+1];
    for (int i = l; i <= r; ++i) {
        aux[i-l] = arr[i];
    }
    int i = l, j = mid+1;
    for (int k = l; k <= r; ++k) {
        // 首先处理i, j越界
        if (i > mid) {
            arr[k] = aux[j-l];
            ++j;
        } else if (j > r) {
            arr[k] = aux[i-l];
            ++i;
        } else if (aux[i-l] <= aux[j-l]) {
            arr[k] = aux[i-l];
            ++i;
        } else {
            arr[k] = aux[j-l];
            ++j;
        }
    }
}
```

(19.8.2）递归的归并排序算法

```
// 递归使用归并排序，对arr[l...r]的范围进行排序
template <typename T>
void __mergeSort(T arr[], int l, int r) {
    if (l >= r) return;
    int mid = l + (r-l)/2;
    __mergeSort(arr, l, mid);
    __mergeSort(arr, mid+1, r);
    if (arr[mid] > arr[mid+1]) {  // 优化处理，使之能够更好处理近乎有序的数组
        __merge(arr, l, mid, r);
    }
}
```

(19.8.3）非递归的归并排序算法

```
// 使用循环进行自底向上的归并排序
template <typename T>
void mergeSortBU(T arr[], int n) {
    for (int sz = 1; sz <= n; sz += sz) {
        // 对arr[i...i+sz-1]和arr[i+sz...i+sz+sz-1]进行归并
        for (int i = 0; i + sz < n; i += sz+sz) {
```

```
            // 细节一：i+sz < n是为了防止前一段的右端点i+sz-1不会越界
            if (arr[i+sz-1] > arr[i+sz]) {
                __merge(arr, i, i+sz-1, min(i+sz+sz-1, n-1));
                // 细节二：min在这里的作用是为了防止后一段的右端点i+sz+sz-1不会越界
            }
        }
    }
}
/*由于自底向上的归并排序没有直接使用索引对数据进行操作，
因此可以方便应用于对链表这种结构进行排序的过程中*/
```

20. 折半查找法

```
int Bsearch(int arr[], int low, int high, int k) {
    int mid;
    while (low < high) {
        mid = (low+high) / 2;
        if (arr[mid] == k) {
            return mid;
        } else if (arr[mid] > k) {
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return 0;
}
```

21. 二叉排序树

21.1）二叉排序树的结点定义

```
typedef struct BTNode {
    int key;
    struct BTNode *lchild;
    struct BTNode *rchild;
}BTNode;
```

21.2）二叉排序树的查找算法

```
BTNode* BSTSearch(BTNode *bt, int key) {
    if (bt == nullptr) return nullptr;
    if (bt->key == key) {
        return bt;
    } else if (key < bt->key) {
        return BSTSearch(bt->lchild, key);
    } else {
        return BSTSearch(bt->rchild, key);
    }
}
```

21.3）二叉排序树的插入算法

```
int BSTInsert(BTNode *&bt, int key) {  // 因为bt要改变，所以要用引用型指针
    if (bt == nullptr) {
        bt = (BTNode*)malloc(sizeof(BTNode));  // 创建新结点
        bt->lchild = bt->rchild = nullptr;
        bt->key = key;
        return 1;
    } else {
        if (key == bt->key) return 0;   // 关键字已经存在于树中
        else if (key < bt->key) return BSTInsert(bt->lchild, key);
        else return BSTInsert(bt->rchild, key);
    }
}
```

21.4）二叉排序树的构造算法

```
void CreateBST(BTNode *&bt, int key[], int n) {
    bt = nullptr;  // 将树清空
    for (int i = 0; i < n; ++i) {
        BSTInsert(bt, key[i]);
    }
}
```