上机报告一

顺序表

头

```
#include<iostream>

#define ERROR   0
#define OK  1
#define LIST_INIT_SIZE   100
#define LISTINCREMENT  10

using namespace std;

typedef int Status;

typedef  struct {
    int     key;
}ElemType;

typedef  struct {//顺序表结构
    ElemType *elem;
    int length;
    int listsize;
}SqList;
int InitList_SqList(SqList &L) {//创建一个空顺序表
    L.elem = (ElemType*)malloc(LIST_INIT_SIZE*sizeof(ElemType));
    if (!L.elem)
        exit(OVERFLOW);
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;
    return OK;
}
```

递增顺序表去重复

算法思想:每个元素与其前一个元素比较,如果相同则删除这个元素

```
int RemoveDuplicates(SqList &L) {
    //删除递增数组的重复数
    //当i的值与index的值不同时，elem[index + 1]的地址存入elem[i]的地址，
index + 1
    int i, index = 0;
    if (L.length == 0) return 0;
    for (i = 1; i<L.length; i++) {
        if (L.elem[index].key != L.elem[i].key) {
            L.elem[index + 1] = L.elem[i];
            index++;
        }
    }
    L.length = index + 1;
    for (i = 0; i<L.length; i++) {
        if (L.elem[i].key == 100)
            break;
        cout << L.elem[i].key << ' ';
    }
}
```

- 时间复杂度O(n).遍历一次数组
- 空间复杂度O(1).不占用额外的空间

排除所有0元素

算法思想:逐个检索,遇到0就删除

```
void remove_zero(SqList& L) {
    int i; int e;
    for (i = 0; i < L.length; i++) {
        if (L.elem[i].key == 100)
            break;
        if (L.elem[i].key == 0) {
            ListDelete_Sq(L, i, e);
            --i;
        }
    }
    for (i = 0; i<L.length; i++) {
```

```
        if (L.elem[i].key == 100)
            break;
        cout << L.elem[i].key << ' ';
    }
}
```

- 时间复杂度O(n)
- 空间复杂度O(1)

冒泡排序

算法思想:从序列的最右端开始,比较相邻的两个数,如果右端的数字更小,则交换两个数的位置,从右往左比较一轮后,最小的数字已经排到了序列的最左端.重复操作,直到所有数字都被排序.

```
void bubble_sort(SqList L){//冒泡排序，排为递增序列
    int i, j, swap;
    for (i = 0; i < L.length; i++){
        for (j = L.length - 1; j > i; j--){
            if (L.elem[j].key < L.elem[j - 1].key){
                swap = L.elem[j].key;
                L.elem[j].key = L.elem[j - 1].key;
                L.elem[j - 1].key = swap;
            }
        }
    }
    for (i = 0; i < L.length; i++)
        cout << L.elem[i].key << ' ';
    cout << endl;
}
```

- 时间复杂度O(n^2).即 $(n-1)+(n-2)+\ldots+1=\frac{n*(n-1)}{2}$
- 空间复杂度O(1)

选择排序

算法思想:线性搜索序列中的最小值,将其与列中最左端的数字进行交换,重复直到所有数字都被排序

```
void select_sort(SqList L){
```

```
    //选择排序，递增
    int i, j, min, swap;
    for (i = 0; i < L.length; i++) {
        min = i;
        for (j = i + 1; j < L.length; j++) {
            if (L.elem[j].key < L.elem[min].key)
                min = j;
        }
        swap = L.elem[i].key;
        L.elem[i].key = L.elem[min].key;
        L.elem[min].key = swap;
    }
    for (i = 0; i < L.length; i++)
        cout << L.elem[i].key << ' ';
    cout << endl;
}
```

- 时间复杂度O(n^2).线性遍历找最小值需要O(n^2),将最小值放在序列最前需要O(n^2)
- 空间复杂度O(1)

插入排序

算法思想:取出序列中的数字,将其与已经操作的左侧的数字进行比较,直到出现一个较小的数字或者数字到达左端,将取出的这个数字插入.

```
void insert_sort(SqList L) {
    //插入排序，递增
    int e;//用于存放取出的元素
    int i, j;
    for (i = 1; i < L.length; i++)
        for (j = 0; j < i; j++) {
            if (L.elem[i].key < L.elem[j].key) {
                ListDelete_Sq(L, i, e);
                ListInsert_Sq(L, j, e);
            }
        }
    for (i = 0; i < L.length; i++) {
        cout << L.elem[i].key << ' ';
    }
}
```

- 时间复杂度O(n^2)
- 空间复杂度O(1)

模式匹配(KMP算法)

算法思想:由于模式串可能具有一定的对称性,因此失配时不必从第一个元素重新开始匹配,主串也不用回退.关键是根据模式串的重复序列找到下一个匹配位置,即next函数.

next()函数相当于是将模式串看作主串,失配前的序列看作模式串.因此可以利用递归求next()函数.

1. next函数

```
void get_next(SString t, int next[]) {//用于KMP算法的next函数
    int i = 1, j = 0;
    next[1] = 0;//第一个位置失配的next置为0,表示主串应当前进一位
    while (i < t[0]) {//t[0]存储的是串的长度
        if (j == 0 || t[i] == t[j]) {
            ++i;
            ++j;
            next[i] = j;
        }
        else j = next[j];
    }
}
```

2. KMP算法

```
int KMPmatch(SString s, SString t,int next[]) {
    //kmp算法，返回子串第一个元素在主串中出现的位置，如果不匹配则返回0
    int j = 0;
    int i = 1;
    while (i <= s[0] && j <= t[0]) {
        if (j == 0 || s[i] == t[j]) {
            ++i;
            ++j;
        }
        else j = next[j];
    }
    if (j > t[0]) {
        cout << i - t[0];
```

```
        return i - t[0];
    }//子串从主串的第i-n个元素开始匹配
    else return 0;//不匹配
}
```

- 时间复杂度O(n + m),匹配过程为O(n),next函数O(m)
- 空间复杂度O(m),next[j]需要m个存储空间

单链表

头

```
#include<iostream>

#define OK 1
#define ERROR 0

using namespace std;

typedef int Status;
typedef struct LNode {
    int data;
    struct LNode* next;
}LNode, *LinkList;
void CreateList(LinkList& L, int a[], int n) {
    //假设数据都是int型变量
    //利用数组创建一个链表
    LNode* s;
    int i;
    LNode* r;
    L = (LinkList)malloc(sizeof(LinkList));  //创建头结点
    r = L; // r始终指向终端节点，开始时指向头节点
    for (i = 0; i < n; i++) {
        s = new(LNode); //创建新结点
        s->data = a[i];
        s->next = NULL;
        r->next = s;   //将s插入在r之后
        r = s;
    }
    r->next = NULL;
}
```

单链表逆置(不额外使用结点)

算法思想:每个结点往前指.例如,原本的顺序是p->q->r,改为p<-q r.因此需要三个指针.

```
LinkList reverse(LinkList& L){
    LNode *p, *q, *r;
    p = L->next;
    if (p == NULL || p->next == NULL){
        cout << "No reverse needed!\n";
    }
    q = p->next;
    p->next = NULL;
    while (q != NULL){
        r = q->next;
        q->next = p;
        p = q;
        q = r;
    }
    L->next = p;
    return L;
}
```

- 时间复杂度O(n).
- 空间复杂度O(1).设置了3个指针,为了保证不丢失结点.

找单链表中点

算法思想:设置两个指针p和q,p每次前进一步,q每次前进两步.当q走到表尾的NULL时,p正好指向链表的中点(如果链表中的元素的个数是偶数个,则指向的是靠后的那个元素).

```
void LinkList_mid(LinkList L) {
    LNode* p, * q;
    p = L->next;
    q = L->next;
    while (q->next != NULL && q != NULL) {
        p = p->next;
        q = q->next->next;
    }
    cout << p->data;
}
```

- 时间复杂度O(n).
- 空间复杂度O(1).

找单链表倒数第K个点

算法思想:逆置链表,找逆置后的第K个结点

1. 访问带头结点的链表中的第i个元素

```
Status ListGet_L(LinkList L, int i) {
    LNode* p;
    p = L;
    int j = 0;
    while (p->next != NULL && j < i - 1) {
        //p指向第i个结点的前驱
        p = p->next;
        ++j;
    }
    if (p->next == NULL || j > i - 1)
        return ERROR;
    cout << p->next->data;
    return OK;
}
```

2. 访问逆置后的链表的第k个元素,就是原链表的倒数第k个元素

```
    reverse(L);
    int k;
    ListGet_L(L, k);
```

- 时间复杂度O(n).访问第i个节点O(n),单链表逆置算法O(n).
- 空间复杂度O(1).逆置链表没有花费额外的空间.仅需要一个计数的空间

删除单链表倒数第K个点

算法思想:删除逆置后的链表中的第K个结点

1. 在带头结点的单链表L中删除第i个元素

```
Status ListDelete_L(LinkList& L, int i) {
    LNode* p, *q;
    p = L;
    int j = 0;
    while (p->next != NULL && j < i - 1) {
        p = p->next;
        ++j;
    }
    if (p->next == NULL || j > i - 1)
        return ERROR;
    q = p->next;
    p->next = q->next;
    free(q);
    p = L->next;
    while (p != NULL) {
        cout << p->data << ' ';
        p = p->next;
    }
    return OK;
}
```

2. 链表逆置,并删除逆置后链表的第K个元素,即原链表的倒数第K个元素

```
    reverse(L);
    ListDelete_L(L, k);
```

- 时间复杂度O(n).
- 空间复杂度O(1).

判断单链表是否有环，如有，找到交点

算法思想:设置两个指针p,q.p每次前进一个结点,q每次前进两个结点.如果链表有环,则p,q会重合.接下来演算如何找到交叉位置.

设交点前的链包含a个结点(不包括交点,因为交点属于环),设环上有b个结点.

设p,q相遇时,p移动了n次,则q移动了$2*n$次.

且$2*n=n+k*b,k$是整数,因为q应该领先p整数倍个环

且$n<a+b$.

因此有$a=k*b-(n-a)$.

令指针p指向头结点,q不变.二者每次都前进一个结点,最终会在交点相遇.

```
void loop_find (LinkList& L) {
    LNode* p, * q;
    p = L->next;
    q = p->next;
    bool l;
    while (p != q) {
        if (q == NULL) {
            l = 0;
            break;
        }
        p = p->next;
        q = q->next->next;
    }
    if (l == 0) {
        cout << "没有环" << endl;
    }
    else {
        p = L->next;
        while (p != q) {
            p = p->next;
            q = q->next;
        }
        cout << "重复节点" <<p << endl;
    }
}
```

- 时间复杂度O(n).p指针一共移动$n+a$次,$a<n$.
- 空间复杂度O(1).仅需要存两个指针p和q.

判断两个单链表是否相交，相交则找出交点

算法思想

```
void cross_find() {
    LinkList A, B;
    int i = 0, j = 0, k = 0;//记录链的长度
    LNode* p = A->next, * q = B->next;
    while (p != NULL) {
        i++;
        p = p->next;
    }
    while (q != NULL) {
        j++;
        q = q->next;
    }
    if (p != q) {//最后一个结点也不相交,说明两条链没有交点
        cout << "不重复" << endl;
        return;
    }
    reverse(B);//将链表B逆置
    p = A->next;
    while (p != NULL) {
        k++;
        p = p->next;
    }
    int a;
    p = A->next;
    for (a = 1; a <= (i + j - k) / 2 + 1; a++) {
        p = p->next;
    }
    cout << p;//输出第一个交叉点的地址
}
```

- 时间复杂度O(n).一共进行了3次遍历,一次逆置.
- 空间复杂度O(1).仅需要存储3个链长度信息

对于有序单链表，删除重复节点

保留一个

```
void rm_repeat_1(LinkList& L) {//去重复，保留一个
    LNode* p, * q;
    p = L->next;
    q = p->next;
    while (q != NULL) {
        if (p->data == q->data) {
            p->next = q->next;
            q = p->next;
        }
        else {
            p = p->next;
            q = q->next;
        }
    }
}
```

无保留

```
void rm_repeat_2(LinkList& L) {
    //去重复，不保留
    LNode* p, * q, * r;
    p = L->next;
    q = p->next;
    r = q;
    while (q != NULL) {
        while (r->next != NULL) {
            if (r->next->data == r->data) {
                r = r->next;
            }
            else
                break;
        }
        if (q != r) {
            p->next = r->next;
            q = p->next;
            r = q;
        }
        else {
            p = p->next;
```

```
            q = p->next;
            r = q;
        }
    }
}
```

- 时间复杂度O(n)
- 空间复杂度O(1)

约瑟夫问题

有损

设置一个计数,指针q每次前进一个结点,计数加一,模三.每次为0时就删除当前的结点.同时计数k用于记录剩余的人数.要剩余两个结点则k > 2.

```
void josephus_1() {//Josephus问题,有损
    int m = 9, n = 3;//问题的规模.m个人,每隔n个删除一人
    LinkList L;
    L = (LinkList)malloc(sizeof(LinkList));
    LNode* s, * r;
    int i;
    r = L;
    for (i = 0; i < m; i++) {
        s = new(LNode); //创建新结点
        s->data = i + 1;
        s->next = NULL;
        r->next = s;   //将s插入在r之后
        r = s;
    }
    r->next = NULL;
    display(L);
    //上述内容是创造m-n的约瑟夫问题
    int k = m;//用于记录存活人数
    LNode* p = L, * q = p->next;
    int j = 1;
    while (k > 2) {
        if (j == 0) {
            if (q->next != NULL) {
                cout << q->data << ' ';
```

```
                p->next = q->next;
                q = p->next;
                j = (j + 1) % 3;
                k--;
            }
            else {//到链尾结点,则重新返回头结点
                cout << q->data << ' ';
                p->next = NULL;
                p = L;
                q = p->next;
                j = (j + 1) % 3;
                k--;
            }
        }
        else {
            if (q->next != NULL) {
                p = p->next;
                q = p->next;
                j = (j + 1) % 3;
            }
            else {
                p = L;
                q = p->next;
                j = (j + 1) % 3;
            }
        }
    }
    cout << endl;
    display(L);
}
```

- 时间复杂度O($n * m$).
- 空间复杂度O(1).

无损

跟有损问题相比,无损问题中不删除结点,而是另外设置一个布尔型数组用于标记应当删除的人.在访问到标记为删除的结点时,就跳过.

```
void josephus_2() {//Josephus问题,无损
    const int m = 9, n = 3;
```

```
    LinkList L;
    L = (LinkList)malloc(sizeof(LinkList));
    LNode* s, * r;
    int i;
    r = L;
    for (i = 0; i < m; i++) {
        s = new(LNode); //创建新结点
        s->data = i + 1;
        s->next = NULL;
        r->next = s;    //将s插入在r之后
        r = s;
    }
    r->next = NULL;
    display(L);
    //上述内容是创造m--n的约瑟夫问题
    int k = m;//用于记录存活人数
    bool death[m];//标记
    for (int i = 0; i < m; i++)
        death[i] = 0;
    LNode* p = L, * q = p->next;
    int l = 1, j = 1;
    while (k > 2) {
        while (death[l % m] == 1) {//人已经删除了，应该跳过
            if (q->next != NULL) {
                p = p->next;
                q = q->next;
                l = (l + 1) % m;
            }
            else {
                p = L;
                q = p->next;
                l = 1;
            }
        }
        if (j == 0) {
            if (q->next != NULL) {
                cout << q->data << ' ';//显示删除的元素
                death[l%m] = 1;
                p = p->next;
                q = p->next;
                j = (j + 1) % 3;
                l = (l + 1) % m;
                k--;
```

```
            }
            else {
                cout << q->data << ' ';
                death[l%m] = 1;
                p = L;
                q = p->next;
                j = (j + 1) % 3;
                l = (l + 1) % m;//这里应该刚好等于1，否则是错误的
                k--;
            }
        }
        else {
            if (q->next != NULL) {
                p = p->next;
                q = p->next;
                j = (j + 1) % 3;
                l = (l + 1) % m;
            }
            else {
                p = L;
                q = p->next;
                j = (j + 1) % 3;
                l = (l + 1) % m;
            }
        }
    }
    cout << endl;
    p = L->next;
    l = 1;
    while (p != NULL) {
        if (death[l] == 0)
            cout << p->data << ' ';
        l++;
        p = p->next;
    }
}
```

- 时间复杂度O($n * m$).
- 空间复杂度O(m).要存储标记

合并两个升序单链表

升序

类似于插入排序.将B链表中的元素按顺序拿出来,和A中的元素比较,首次遇到更大的元素时,插入到这个元素前面;如果遇不到更大的元素,则直接插入到表尾.由于A和B链表都是升序的,因此插入下一个元素时可以从当前插入位置开始比较

```
LinkList joint(LinkList &A, LinkList &B) {//升序链表合并为升序链表
    LinkList C;
    //考虑都不设置头节点
    //将B中的元素插入到A中
    LinkList p = B->next;//point
    LinkList q = A->next;//search
    LNode* r;
    C = A;
    bool T;
    while (q->next != NULL) {
        if (p->data < q->data) {
            r = p->next;
            p->next = q;
            A->next = p;
            p = r;
        }
        else {
            A = q;
            q = A->next;
        }
        if (p->next == NULL) {
            T = true;
            break;
        }
    }
    if (q->next == NULL) {
        while (p->next != NULL) {
            if (p->data < q->data) {
                r = p->next;
                p->next = q;
                A->next = p;
                p = r;
            }
        }
```

```
        q->next = p;
    }
    A = C;
    return C;
}
```

降序

降序则是将升序的链表逆置,逆置函数:

```
LinkList reverse(LinkList& L){//单链表逆置,不改变原来的结点
    LNode *p, *q, *r;
    p = L->next;
    if (p == NULL || p->next == NULL){
        cout << "No reverse needed!\n";

    }
    q = p->next;
    p->next = NULL;
    while (q != NULL){
        r = q->next;
        q->next = p;
        p = q;
        q = r;
    }
    L->next = p;
    return L;
}
```

- 时间复杂度O($n * m$).
- 空间复杂度O(1).

判断一个单链表是否对称

算法思想:头插法构建一个新的链表,是原本的链表的逆序,之后比较这两个链表即可.

```
void symmetry(LinkList L) {
    LNode* b = L->next, * a;
    LinkList S;
    S = (LinkList)malloc(sizeof(LNode));
```

```
    S->next = NULL;
    a = (LinkList)malloc(sizeof(LNode));
    while (b != NULL) {
        a->data = b->data;
        a->next = S->next;
        S->next = a;
        b = b->next;
    }
    LNode* p = L->next, * q = S->next;
    while (p != NULL) {
        if (p->data == q->data) {
            p = p->next;
            q = q->next;
        }
        else break;
    }
    if (p == NULL)
        cout << " 对称" << endl;
    else
        cout << " 不对称" << endl;
}
```

- 时间复杂度O(n)
- 空间复杂度O(n)