《数据结构》上机报告

2019 年 10 月 14 日

姓名：马家昱 学号：1950509 班级：计科 1 班 得分：400

<table>
<tr><td>实验题目</td><td colspan="2">链表</td></tr>
<tr><td>问题描述</td><td colspan="2">链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。</td></tr>
<tr><td rowspan="2">基本要求</td><td colspan="2">1. (p1)实现链表的基本操作，包括顺序表的初始化、头插法与尾插法生成链表、第 i 个元素前插入一个新的元素、删除第 i 个元素、查找某元素、链表的销毁。
2. (p2)实现对链表进行去重，即在多个重复元素中只保留一个，删除其他重复元素节点的功能。
3. (p3)实现对链表指定范围的节点进行逆置。
4. (p4)实现使用单链表表示一元多项式，并实现多项式的加法与乘法功能。</td></tr>
<tr><td>已完成基本内容（序号）：</td><td>1，2，3，4，5</td></tr>
<tr><td rowspan="2">选做要求</td><td colspan="2">1. 程序要增添适当的注释，程序的书写要采用缩进格式。
2. 程序拥有一定的健壮性，对非法输入有相应的提示。</td></tr>
<tr><td>已完成选做内容（序号）</td><td>1，2</td></tr>
<tr><td>数据结构设计</td><td colspan="2">本实验均采用带头结点的单链表结构，每个节点包含数据域与指针域，其中，头结点的数据域用来存储链表长度信息。前三题中的节点数据域均为一个整数型数据，一元多项式的操作中，数据域包含两个整数型数据，分别表示系数与指数。指针域中的指针指向该节点的下一个节点。</td></tr>
</table>

功能（函数）说明	1. 链表的基本操作
	节点类与节点类初始化
	`class LNode {  //节点`
	`public:`
	`  int data;`
	`  LNode* next;`
	`public:`
	`  LNode();`
	`};`
	
	`LNode::LNode() {  //节点初始化`
	`  data = 0;`
	`  this->next = NULL;`
	`}`
	链表类与链表类初始化
	`class Llist {  //链表，头结点元素储存长度`
	`public:`
	`  LNode* head;`
	`public:`
	`  Llist();//初始化`
	`  void printList();//打印链表中所有元素`
	`  bool insert(int i, int x);//插入，i为位置，x为元素`
	`  bool remove(int j);//移除，j为位置`
	`  int check(int y);//查找并返回位置`
	`  void uniqueList();//去重`
	`  bool reverse(int begin, int end);//逆置，begin与end为起始位置`
	`};`
	
	`Llist::Llist() { //给头节点分配存储空间，长度置0，next指针为NULL`
	`  this->head = (LNode*)malloc(sizeof(LNode));`
	`  this->head->data = 0;`
	`  this->head->next = NULL;`
	`}`
	Void printList（）打印链表中所有元素
	`void Llist::printList() { //遍历并打印`
	`  LNode* temp = this->head;`
	`  for (int i = 0; i < this->head->data; i++) {`
	`  temp = temp->next;`
	`  cout << temp->data << ' ';`
	`  }`
	`  cout << endl;`
	`}`
	Bool insert（int i，int x）在位置 i 上插入新节点，其数据域为 x

```
bool Llist::insert(int i, int x) { //插入
    if (i<1 || i>this->head->data + 1) //位置合法性判断
        return false;
    LNode* newNode = (LNode*)malloc(sizeof(LNode));
    if (!newNode)
        return false;
    LNode* temp = this->head;
    for (int t = 0; t < i - 1; t++) //将指针移动到需要插入位置的前一个节点
        temp = temp->next;
    newNode->next = temp->next; //插入
    temp->next = newNode;
    newNode->data = x;
    this->head->data++; //长度加1
    return true;
}
```

Bool remove（int j）删除 j 位置上的节点

```
bool Llist::remove(int j) {
    if (j<1 || j>this->head->data) //位置合法性判断
        return false;
    LNode* temp = this->head;
    for (int t = 0; t < j - 1; t++) //将指针移动到需要删除位置的前一个节点
        temp = te
    temp->next =                    ->next; //删除
    this->head->d                   度减1
    return true;
}
```

(int)0
联机搜索

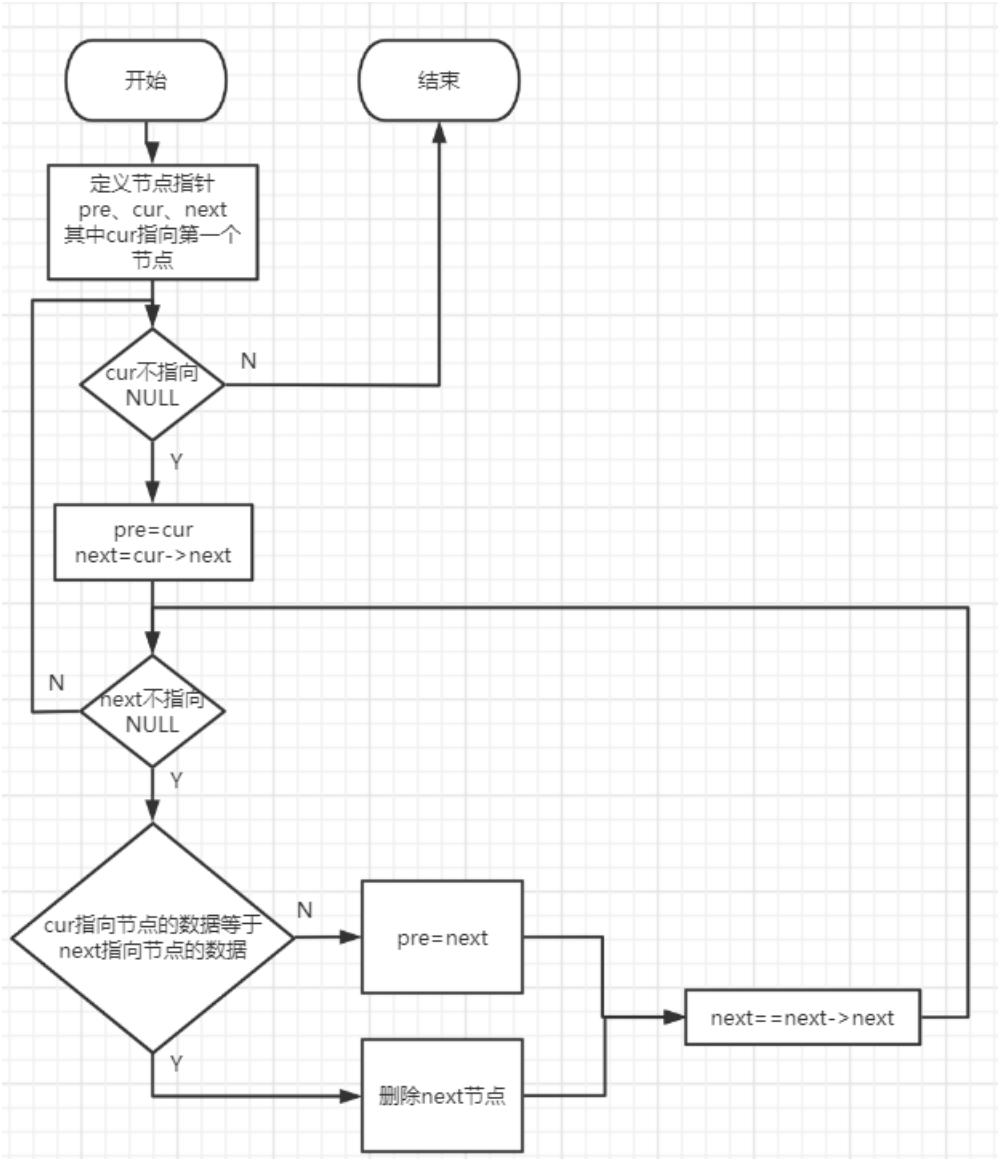
在链表中查找元素 y，并返回其位置

```
int Llist::check(int y) { //遍历并查找
    LNode* temp = this->head;
    for (int i = 0; i < this->head->data; i++) {
        temp = temp->next;
        if (temp->data == y)
            return i + 1;
    }
    return -1;
}
```

2. 链表的去重 void uniqueList（）

```
void Llist::uniqueList() { //去重
    LNode* cur = this->head->next;
    LNode* pre = NULL;
    LNode* next = NULL;
    while (cur != NULL) {
        pre = cur;
        next = cur->next;
        while (next != NULL) {
            if (cur->data == next->data) {//若找到重复元素，将其删除
                pre->next = next->next;
                this->head->data--;
            }
            else pre = next;//否则重新定位pre指针
            next = next->next;
        }
        cur = cur->next;
    }
}
```

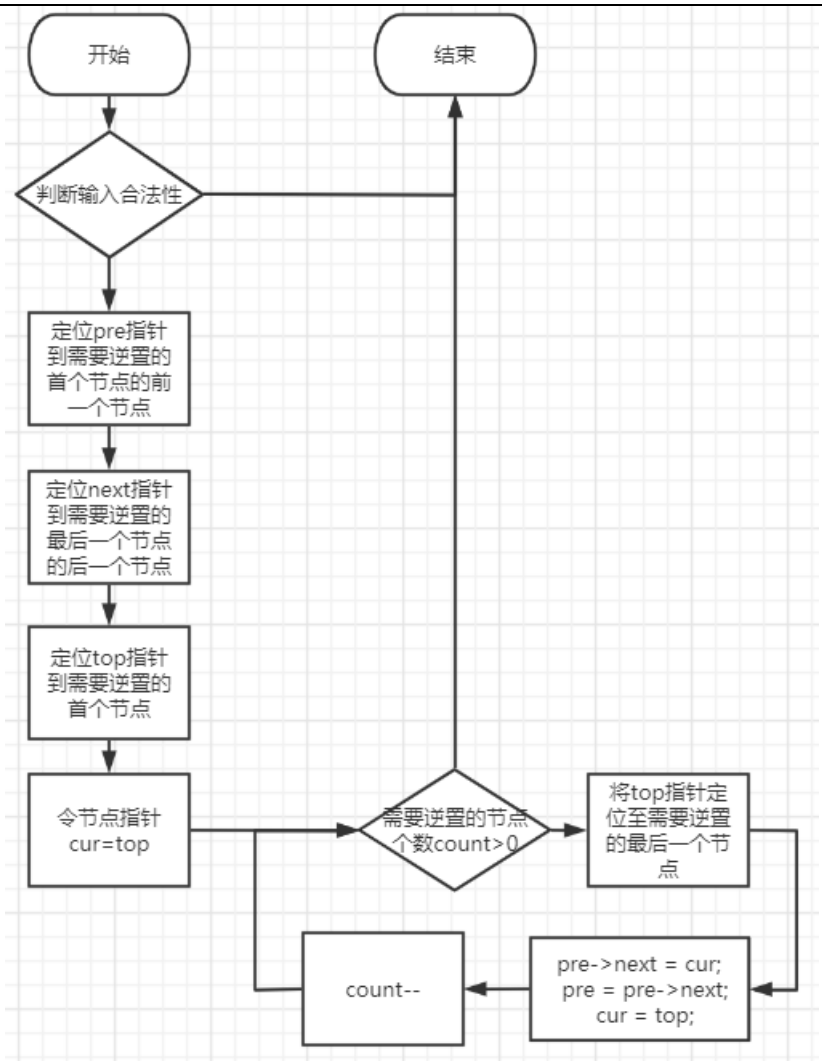
流程图：



3. 链表的逆置 void reverse（int begin，int end）begin 与 end 为需要逆置的节点始末位置。

```
bool Llist::reverse(int begin, int end) {//逆置
    if (begin >= end)
        return false;
    if (begin<1 || begin>this->head->data - 1)
        return false;
    if (end<2 || end>this->head->data)//输入合法性判断
        return false;
    int count = end - begin;
    LNode* pre = this->head;
    for (int i = 0; i < begin - 1; i++)//定位pre指针到需要逆置的首个元素的前一个节点
        pre = pre->next;
    LNode* next = pre;//定位next指针到需要逆置的最后一个元素的下一个节点
    for (int i = 0; i < end - begin + 2; i++)
        next = next->next;
    LNode* top = pre->next;//top指针指向需要逆置的首个元素
    LNode* cur = top;
    for (++count; count; count--) {//开始逆置
        for (int i = 0; i < count - 1; i++)
            cur = cur->next;
        pre->next = cur;
        pre = pre->next;
        cur = top;
    }
    pre->next = next;
    return true;
}
```

流程图：



4. 一元多项式的加法与乘法

节点类与节点的初始化

```
class LNode { //项类，p、e分别表示系数与指数
public:
    int p;
    int e;
    LNode* next;
public:
    LNode();
};

LNode::LNode() { //项的初始化
    p = 0;
    e = 0;
    this->next = NULL;
}
```

链表类与链表的初始化

```
class Llist { //多项式类
public:
    LNode* head;
public:
    Llist();
    void printList(); //打印多项式
    bool insert(int p, int e); //按指数升序插入项
    void addPolyn(Llist L); //多项式加法
    Llist multiPolyn(Llist L); //多项式与项的乘法
    Llist singleMulti(LNode* term); //多项式与多项式的乘法
};

Llist::Llist() { //多项式链表的初始化，头结点的数据域存储链表长度
    this->head = (LNode*)malloc(sizeof(LNode));
    this->head->p = 0;
    this->head->next = NULL;
}
```

Void printList（）打印整个多项式，系数为 0 的项忽略

```
void Llist::printList() { //打印多项式，其中系数为0的项忽略不计
    LNode* temp = this->head;
    for (int i = 0; i < this->head->p; i++) {
        temp = temp->next;
        if (temp->p != 0)
            cout << temp->p << ' ' << temp->e << ' ';
    }
    cout << endl;
}
```

Bool insert（int p，int e）插入项，自动按指数升序插入

```
bool Llist::insert(int p, int e) { //插入项，自动按照指数升序插入
    LNode* newNode = (LNode*)malloc(sizeof(LNode));
    if (!newNode)
        return false;
    LNode* temp = this->head;
    for (; temp->next && e > temp->next->e;)//将指针移动到应该插入的节点的前一个节点
        temp = temp->next;
    newNode->next = temp->next;
    temp->next = newNode;
    newNode->p = p;
    newNode->e = e;
    this->head->p++;
    return true;
}
```

Bool addPolyn（Llist L）将多项式 L 加入到此多项式中

```
void Llist::addPolyn(Llist L) { //多项式加法
    LNode* rear = this->head;
    bool isInsert = 0;
    if (this->head->p == 0) { //如果为空表，则直接将另一个多项式的每一个项依次插入
        for (LNode* cur = L.head->next; cur; cur = cur->next)
            this->insert(cur->p, cur->e);
        return;
    }
    for (int i = 0; i < this->head->p; i++)
        rear = rear->next;
    LNode* thisCur = this->head->next;
    for (LNode* cur = L.head->next; cur; cur = cur->next) {
        if (cur->e<thisCur->e || cur->e>rear->e) //如果要加入的项的指数小于原来表中项的最小的指数，大于最大的指数，则直接插入
            this->insert(cur->p, cur->e);
        else {
            isInsert = 0;
            for (LNode* temp = thisCur; temp; temp = temp->next) //指数相等，则系数相加
                if (cur->e == temp->e) {
                    temp->p += cur->p;
                    isInsert = 1;
                    break;
                }
            if (isInsert == 0)//没有指数相等的项，则直接插入
                this->insert(cur->p, cur->e);
        }
    }
}
```

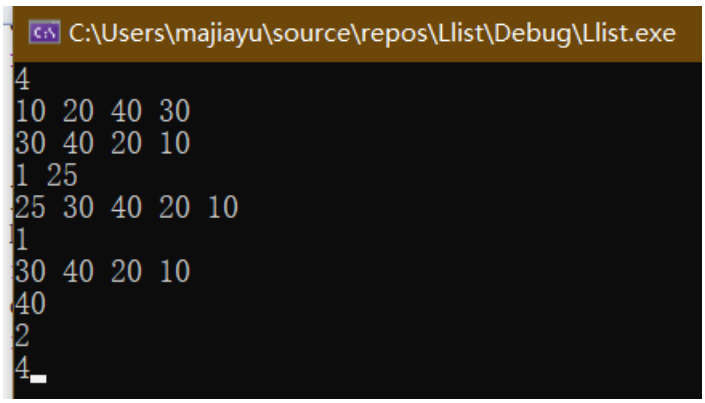
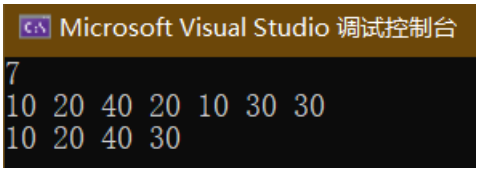
Llist singleMulti（LNode term）将此多项式与单个项相乘，并返回结果

```
Llist Llist::singleMulti(LNode* term) { //多项式与一个项的乘法，返回所得的多项式
    Llist L;
    LNode* cur = this->head;
    for (int i = 0; i < this->head->p; i++) {
        cur = cur->next;
        L.insert(cur->p * term->p, cur->e + term->e);
    }
    return L;
}
```

Llist multiPolyn（Llist L）将此多项式与多项式 L 相乘，并返回结果

```
Llist Llist::multiPolyn(Llist L) { //多项式与多项式的乘法，原多项式分别与另一个多项式的每一个项相乘，结果累加并返回
    Llist result;
    for (LNode* cur = L.head->next; cur; cur = cur->next) {
        Llist temp = this->singleMulti(cur);
        result.addPolyn(temp);
    }
    return result;
}
```

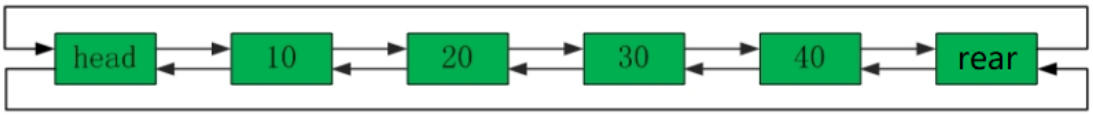
开发环境	Windows10 Microsoft VS code 2019

调试分析	（运行结果截图） 1. 链表的基本操作  2. 链表的去重  3. 链表的逆置  4. 一元多项式的加法与乘法 
心得体会	（对整个实验过程做出总结，对重要的算法做出性能分析。） 时间复杂度分析： 1. 链表的基本操作中，在插入、删除、查找功能的最坏情况下，均只需要遍历一遍链表即可，因此其时间复杂度为 O（n）。 2. 链表的去重操作中，需要对每一个元素遍历一遍整个链表来查找重复元素，需使用到两层循环嵌套，因此时间复杂度为 O（n^2） 3. 链表的逆置中，在对每一个需要逆置的节点进行操作时，需要定位指针至需要逆置的节点处，因此需要使用两层循环嵌套，时间复杂度为 O（n^2）。 4. 一元多项式的加法，此算法为表的合并，两个链表的指针同时向前推进，时间复

杂度为 O（n）。

5. 一元多项式的乘法，需要将一个链表中的每个项与另一个链表相乘，并累加，只需要遍历一遍链表即可，时间复杂度为 O（n^2）

带头指针和尾指针及长度属性的双向循环链表的存储结构描述：



循环链表的指针域中存在两个指针 pre 与 next，分别指向该节点的前一个节点与后一个节点，循环链表的长度属性储存在 head 节点的数据域中。其循环属性表现在 rear 节点的 next 指针指向 head 节点，head 节点的 pre 指针指向 head 节点。

链表的优缺点：

优点：1.链表节点的存储空间可以动态申请与释放，对空间的利用率高。
2.插入与删除效率高，不需要移动其他元素即可快速插入与删除。

缺点：1.失去了顺序表随机存储的特点，访问元素时只能从一个节点开始顺次访问。

心得体会：通过本次实验，我对链表这种数据结构有了更进一步的了解，从底层开始搭建一个链表，也使我再次熟悉了链表的基本操作，同时分析了各种算法的时间复杂度。链表的应用中，一元多项式的加法与乘法作为一个实例，使我的编程能力得到了锻炼，考虑了多种边界条件，自己独立设计了加法与乘法的算法。