天津大学

数据结构实验报告

实验名称：线性表

学院名称 智能与计算学部

专　　业 软件工程

学生姓名 陈昊昆

学　　号 3021001196

年　　级 2021 级

班　　级 软工 3 班

时　　间 2023 年 5 月 18 日

1. 实验内容

1. 完成顺序表的基本操作，class sequenceList

- 初始化顺序表 sequenceList (const int&, const int&, float[]);//顺序表容量，初始化数组长度，初始化数组
- 销毁顺序表, ~ sequenceList ()
- 添加元素, bool addItem(const float&);//添加元素顺序表尾，成功返回 true，失败返回 false
- 插入元素, bool insertItem(const int& index, const float&);//插入元素到 index 位置，成功返回 true，失败返回 false
- 删除元素,int deleteItem(const float&)//返回删除位置，找不到返回-1
- 查找元素（按序号找）, bool locate(const int&, float& val)// 成功返回 true，值放在 val 中，失败返回 false
- 查找元素（按值找）, int locate(const float&)//返回找到位置，找不到返回-1
- 倒序排列元素（使用原空间）, void reverse();
- 按序输出元素, void print()//元素个数:元素 1, 元素 2…

2. 实现有表头结点的链表的基本操作，class linkList

- 初始化链表, linkList(const int&, float[]);//初始化数组长度，初始化数组
- 销毁链表，~ linkList ()
- 插入元素(头插法)，bool headInsertItem(const float&);成功返回 true，失败返回 false
- 插入元素(尾插法)，bool tailInsertItem(const float&);成功返回 true，失败返回 false
- 插入元素，int insertItem(const int& index, const float&); //插入元素到 index 位置，成功返回插入位置，错误返回-1
- 删除元素,int deleteItem(const float&)//返回删除位置，找不到返回-1
- 查找元素（按序号找）, bool locate(const int&, float& val)//成功返回 true，值放在 val 中，失败返回 false
- 查找元素（按值找）, int locate(const float&)//返回找到位置，找不到返回-1
- 递增排序，void ascendingOrder()//
- 倒序排列元素，void reverse();
- 按序输出元素, void print()//元素个数:元素 1, 元素 2…

3. 基于上面的链表实现，设计一个算法 void merge(linkList& A, linkList& B)，读入 A 和 B 两个线性表，输入不需有序，输入后按元素值递增次序排列。编写程序将这两个单链表归并为一个按元素值递减次序排列的单链表，利用原来两个单链表的结点存放归并后的单链表,结果存在 A 线性表中。

2. 程序实现

```
#include "linearList.h"
using namespace std;

// 初始化顺序表
sequenceList::sequenceList(const int &capacity, const int &initsize,
float a[])
{
    myList = new float;
    if (capacity <= 0)
    {
        cout << "Capacity cannot be zero or negative";
    }

    if (initsize > capacity)
    {
        cout << "Cannot initialize over the capacity";
    }

    curNumberOfItem = initsize;
    maxCapcity = capacity;

    for (int i = 0; i < initsize; i++)
    {
        myList[i] = a[i];
    }
}

// 销毁顺序表
sequenceList::~sequenceList()
{
    delete myList;
}

// 添加元素
bool sequenceList::addItem(const float &n)
{
```

```
    if (curNumberOfItem == maxCapcity)
        return false;
    else
    {
        myList[curNumberOfItem] = n;
        curNumberOfItem++;
        return true;
    }
}

// 插入元素
bool sequenceList::insertItem(const int &index, const float &n)
{
    if (curNumberOfItem == maxCapcity)
        return false;
    else
    {
        for (int i = curNumberOfItem - 1; i >= index; i--)
            myList[i + 1] = myList[i];
        myList[index] = n;
        curNumberOfItem++;
        return true;
    }
}

// 删除元素
int sequenceList::deleteItem(const float &n)
{
    for (int i = 0; i < curNumberOfItem; i++)
    {
        if (myList[i] == n)
        {
            for (int j = i; j < curNumberOfItem - 1; j++)
                myList[j] = myList[j + 1];
            curNumberOfItem--;
            return i;
        }
    }
    return -1;
}

// 查找元素（按序号找）
bool sequenceList::locate(const int &index, float &val)
{
    if (index >= 0 && index <= curNumberOfItem - 1)
```

```
    {
        val = myList[index];
        return true;
    }
    else
    {
        return false;
    }
}

// 查找元素（按值找）
int sequenceList::locate(const float &n)
{
    for (int i = 0; i < curNumberOfItem; i++)
    {
        if (myList[i] == n)
        {
            return i;
        }
    }
    return -1;
}

// 倒序排列元素（使用原空间）
void sequenceList::reverse()
{
    for (int i = 0; i < curNumberOfItem / 2; i++)
    {
        float a = myList[i];
        myList[i] = myList[curNumberOfItem - 1 -i];
        myList[curNumberOfItem - 1 - i] = a;
    }
}

// 顺序表打印
void sequenceList::print()
{
    cout << curNumberOfItem << ":";

    for (int i = 0; i < curNumberOfItem - 1; i++)
    {
        cout << myList[i] << ",";
    }

    cout << myList[curNumberOfItem - 1] << endl;
```

```
}

listNode::~listNode()
{
    // do nothing
}

listNode::listNode(float nodeData, listNode* succ)
{
    data = nodeData;
}

// 初始化链表
linkList::linkList(const int &n, float a[])
{
    if (n == 0)
    {
        firstNode = new listNode;
        listSize = 0;
        lastNode = firstNode;
        lastNode->next = NULL;
    }

    if (n == 1)
    {
        firstNode = new listNode;
        curNode = new listNode;
        lastNode = curNode;
        listSize = 1;
        firstNode->next = curNode;
        lastNode->next = NULL;
        curNode->data = a[0];
    }

    else
    {
        firstNode = new listNode;
        curNode = new listNode;
        lastNode = new listNode;
        listSize = n;
        firstNode->next = curNode;
        for (int i = 0; i < n - 2; i++)
        {
            curNode->data = a[i];
            listNode *p = new listNode;
```

```
            curNode->next = p;
            curNode = p;
        }
        curNode->data = a[n - 2];
        curNode->next = lastNode;
        lastNode->data = a[n - 1];
        lastNode->next = NULL;
    }
}

// 销毁链表
linkList::~linkList()
{
    listNode *p = firstNode;
    listNode *q = firstNode->next;
    while (q != lastNode)
    {
        delete p;
        p = q;
        q = p->next;
    }
    delete p;
    delete q;
}

// 插入元素(头插法)
bool linkList::headInsertItem(const float &n)
{
    listSize++;
    listNode *p = new listNode(n);
    if (p == nullptr)
    {
        return false;
    }
    if (listSize == 0)
    {
        lastNode = p;
    }
    p->next = firstNode->next;
    //p->data = n;
    firstNode->next = p;
    return true;
}

// 插入元素(尾插法)
```

```cpp
bool linkList::tailInsertItem(const float &n)
{
    if (listSize == 0)
    {
        listNode *p = new listNode;
        p->data = n;
        p->next = NULL;
        lastNode = p;
        firstNode->next = p;
        listSize++;
    }
    else
    {
        curNode = lastNode;
        lastNode = new listNode;
        if (lastNode == nullptr)
        {
            return false;
        }
        lastNode->data = n;
        lastNode->next = NULL;
        curNode->next = lastNode;
        listSize++;
    }
    return true;
}

// 插入元素
int linkList::insertItem(const int &index, const float &n)
{
    listSize++;
    listNode *p = firstNode;
    for (int i = 0; i < index; i++)
    {
        p = p->next;
    }
    listNode *q = new listNode;
    if (q == nullptr)
    {
        return -1;
    }
    q->next = p->next;
    q->data = n;
    p->next = q;
```

```
    if (index == listSize - 1)
    {
        lastNode = q;
    }
    return index;
}

// 删除元素
int linkList::deleteItem(const float &n)
{
    curNode = firstNode;
    int cnt = 0;

    while (curNode != lastNode)
    {
        if (curNode->next->data == n)
        {
            listSize--;
            listNode *p = curNode->next;
            curNode->next = curNode->next->next;
            delete p;
            return cnt;
        }
        curNode = curNode->next;
        cnt++;
    }

    return -1;
}

// 查找元素（按序号找）
bool linkList::locate(const int &index, float &val)
{
    if (index > listSize - 1)
        return false;
    curNode = firstNode->next;
    for (int i = 0; i < index; i++)
    {
        curNode = curNode->next;
    }
    val = curNode->data;
    return true;
}

// 查找元素（按值找）
```

```
int linkList::locate(const float &n)
{
    curNode = firstNode;
    int cnt = 0;

    while (curNode != lastNode)
    {
        if (curNode->next->data == n)
        {
            return cnt;
        }
        curNode = curNode->next;
        cnt++;
    }

    return -1;
}

// 递增排序
void linkList::ascendingOrder()
{
    if (listSize <= 1)
    {
        return;
    }

    listNode *p = firstNode->next;
    firstNode->next = nullptr;
    lastNode = p;
    while (p != nullptr)
    {
        listNode *nextNode = p->next;
        listNode *q = firstNode;
        while (q->next != nullptr && q->next->data < p->data)
        {
            q = q->next;
        }
        p->next = q->next;
        q->next = p;
        if (p->next == nullptr)
        {
            lastNode = p;
        }
        p = nextNode;
    }
```

```
}

// 倒序排列元素
void linkList::reverse()
{
    if (listSize <= 1)
    {
        return;
    }

    listNode *last = lastNode;
    curNode = firstNode->next;
    lastNode = curNode;
    firstNode->next = last;
    listNode *nextNode = curNode->next;
    listNode *nnNode = curNode->next->next;
    while (true)
    {
        if (nnNode == nullptr)
            break;
        nextNode->next = curNode;
        curNode = nextNode;
        nextNode = nnNode;
        nnNode = nextNode->next;
    }
    last->next = curNode;
    lastNode->next = NULL;
}

// 链表打印
void linkList::print()
{
    curNode = firstNode;

    cout << listSize << ":";

    while (curNode != lastNode)
    {
        if (curNode->next == lastNode)
            cout << curNode->next->data << endl;
        else
        {
            cout << curNode->next->data << ",";
        }
```

```
        curNode = curNode->next;
    }
}

// 获得节点指针
listNode *linkList::get()
{
    return firstNode->next;
}

// 清空链表
void linkList::clear()
{
    listNode *p = firstNode->next;
    while (p != nullptr)
    {
        listNode *q = p->next;
        delete p;
        p = q;
    }
    curNode = new listNode;
    lastNode = curNode;
    lastNode->data = 789.456;
    lastNode->next = NULL;
    firstNode->next = curNode;
    listSize = 1;
}

linkList::linkList()
{
}

void merge(linkList &A, linkList &B)
{
    float lll[] = {789.456};
    linkList C(1, lll);
    A.ascendingOrder();
    B.ascendingOrder();
    listNode *a = A.get();
    listNode *b = B.get();

    while (a != nullptr && b != nullptr)
    {
        if (a->data < b->data)
        {
```

```
            C.tailInsertItem(a->data);
            a = a->next;
        }
        else
        {
            C.tailInsertItem(b->data);
            b = b->next;
        }
    }

    if (a == nullptr)
    {
        while (b != nullptr)
        {
            C.tailInsertItem(b->data);
            b = b->next;
        }
    }
    else
    {
        while (a != nullptr)
        {
            C.tailInsertItem(a->data);
            a = a->next;
        }
    }

    C.deleteItem(789.456);
    C.reverse();

    A.clear();

    listNode *c = C.get();

    while (c != nullptr)
    {
        A.tailInsertItem(c->data);

        c = c->next;
    }

    A.deleteItem(789.456);

    return;
}
```

3. 实验结果

顺序表

```
5:-2.2,1.2,0,-0.0001,0
6:-2.2,1.2,0,-0.0001,0,-2.2
7:1,-2.2,1.2,0,-0.0001,0,-2.2
7:1,-2.2,1.2,0,-0.0001,0,-2.2
7:1,-2.2,1.2,0,-0.0001,0,-2.2
7:-2.2,0,-0.0001,0,1.2,-2.2,1
8:1,-2.2,0,-0.0001,0,1.2,-2.2,1
```

有表头结点的链表

```
5:-2.2,1.2,0,-0.0001,0
6:-2.2,1.2,0,-0.0001,0,-2.2
6:-2.2,1.2,0,-0.0001,0,-2.2
6:-2.2,1.2,0,-0.0001,0,-2.2
6:-2.2,0,-0.0001,0,1.2,-2.2
```

链表合并

```
3:1.3,4.5,0.4
4:1.5,8.5,0.1,6.2
3:0.4,1.3,4.5
4:0.1,1.5,6.2,8.5
7:8.5,6.2,4.5,1.5,1.3,0.4,0.1
```

4. 实验中遇到的问题及解决方法

理解了链表和顺序表的存储结构和操作原理。链表通过链式存储，节点之间通过指针连接；顺序表通过数组连续存储。这两种存储方式各有优缺点，需要根据实际应用场景选择。

实现了链表的头插法、尾插法两种插入方式。头插法适用于实现栈，尾插法适用于实现队列。这两个插入方式也是链表操作的基础。

实现了两个链表的合并操作。这个操作也比较典型，通过比较两个链表节点的值，选择插入较小的值，直至一个链表为空，然后将剩余节点直接插入。这个操作实际上实现了两个有序链表的合并。

理解了指针在链表操作中的作用。很多链表操作都是通过指针移动、创建、删除来实现的。所以对指针的运用很重要。

实现过程中也体会到一些注意事项，比如尾节点的维护，空链表的初始化等。这些细节都是实现链表和顺序表时需要注意的地方。