华东师范大学数据科学与工程学院上机实践报告

课程名称：算法设计与分析	**年级：22级**	**上机实践成绩：**
指导教师：金澈清	**姓名**：朱天祥	
上机实践名称：数据结构	**学号**：10225501461	**上机实践日期**：23/4/6
上机实践编号：No.6	组号：1-461	

一、目的

1．熟悉算法设计的基本思想。

2．掌握栈、链表等基本数据结构。

3．掌握使用非显式指针来表示链表等数据结构的方法。

二、实验内容

1. 用多重数组实现单链表。
2. 实现在该单链表上的“增删查改排”操作，插入并不一定在头部插入，排序可以用任意一种排序方法。注意释放对象。
3. 和用指针实现的链表相比，本次实验中实现的链表有什么使用和性能上的异同呢？（选做）请通过实验来验证你的猜想。

三、使用环境

可以使用你自己喜欢的编程语言和环境来实现。

四、实验过程

1. *写出多重数组单链表的源代码。*

定义一个List类，并定义好成员变量，包括key数组（用于存储数据），next和pre数组（代表指针，由于书上的图用的是双向链表所以我这里是定义了两个指针数组，用于指向前一个结点和后一个结点）

```
class List{
private:
    vector<int> next;
    vector<int> key;
    vector<int> pre;
    int free = 0;
    int head = -1;
};
```

两个私有方法

1.allocate_object用于分配内存

```
    int allocate_object(){
        if(free == -1)
            return -1;
        int obj = free;
        free = next[free];
        return obj;
    }
```

2.free_obj释放内存

```
    int free_obj(int obj){
        next[obj] = free;
        free = obj;
    }
```

接下来的所有方法均为public方法

1.构造器

```
public:
    List(int size){
        next = vector<int>(size);
        key = vector<int>(size);
        pre = vector<int>(size);
        for(int i=0;i<size-1;i++)
            next[i] = i + 1;
        next[size - 1] = -1;
    }
```

size为链表最大容量，初始化next数组让其一一指向后面的索引，并让next数组最后一个元素指向空（链表表尾）

1.增(由于是双向链表，要处理大量指针指向，并且因为是数组的形式，无法使用dummy head假头等优化链表代码的办法，难以避免代码的繁琐)

```
    bool add(int index,int val){
        if(free == -1)
            return false;
        int obj = allocate_object();
        key[obj] = val;
```

```
        if(head == -1){
            head = obj;
            pre[obj] = -1;
            next[obj] = -1;
            return true;
        }
        int p = head;
        while(index > 0 && next[p] != -1){
            p = next[p];
            index--;
        }
        if(index > 0){
            next[p] = obj;
            pre[obj] = p;
            next[obj] = -1;
        }
        else{
            if(p == head){
                head = obj;
                next[head] = p;
                pre[p] = head;
            }
            else{
                next[obj] = p;
                pre[obj] = pre[p];
                next[pre[obj]] = obj;
                pre[p] = obj;
            }
        }
        return true;
    }
```

2.删(双向链表的删除倒是比较方便，找到要删除的结点直接用next和pre指针改变前后结点指向，然后free掉即可)

```
    void remove(int index){
        int p = head;
        while(index > 0){
            index--;
            p = next[p];
        }
        if(p == head){
            if(next[p] != -1)
                pre[next[p]] = -1;
            head = next[p];
        }
        else if(next[p] == -1){
            next[pre[p]] = -1;
        }
        else{
            next[pre[p]] = next[p];
            pre[next[p]] = pre[p];
        }
        free_obj(p);
    }
```

3.改(找到要修改值的结点然后直接赋值即可，while循环条件中其实应该加一个p!=-1，防止数组越界)

```
void update(int index,int val){
    int p = head;
    while(index > 0){
        p = next[p];
        index--;
    }
    key[p] = val;
}
```

4.查(和update逻辑基本一致)

```
int search(int index){
    int p = head;
    while(index > 0){
        p = next[p];
        index--;
    }
    return key[p];
}
```

5.排序(我这里用的是插入排序，链表排序最基本的方法，和add方法一样，由于数组没法加假头并且是双向链表，代码比较难优化)

```
void sortList(){
    int p = next[head];
    next[head] = -1;
    while(p != -1){
        int s = next[p];
        int t = head;
        while(next[t] != -1 && key[t] < key[p])
            t = next[t];
        if(key[t] < key[p]){
            next[t] = p;
            pre[p] = t;
            next[p] = -1;
        }
        else if(t == head){
            head = p;
            next[p] = t;
            pre[t] = p;
        }
        else{
            next[p] = t;
            pre[p] = pre[t];
            next[pre[p]] = p;
            pre[t] = p;
        }
        p = s;
    }
}
```

其实可以考虑用冒泡排序的，我这里就简单讲述一下思路。定义两个指针指向头和头的下一个结点，二者比较完值后可以直接交换二者在key数组中的值，甚至都不用修改next和pre指针，这是冒泡排序的一个非常大的优势，和插入排序相比用不着指针的修改就能完成排序，但是这个方法也有一定的弊端，假如说一个节点内的值key是一个非常复杂的数据类型，包括几十几百几千个属性，那么此时就不能再用这个冒泡排序了，光是值的交换就有大量开销。

二者的复杂度理论上是相同的，都是n^2的时间开销，但是应该还有更好的方法。

链表其实也可以使用归并排序，大致思路是找到链表的中心结点，然后递归地把左边和右边的链表都排好序，然后进行merge操作即可，找中心结点的开销是O（n），merge开销为O（n），左右侧递归复杂度均为T（n/2），故根据master公式可以知道归并排序的复杂度是nlgn，比上述两种排序算法都更适合链表排序

至此list的所有方法就都定义好了

2. *验证实现的正确性，给出各种情况的测试数据和运行结果。*

使用指针实现链表增删改查以及排序

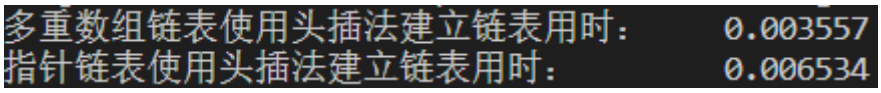
```
class LinkedList {
public:
    struct Node{
        int val;
        Node * next;
        Node(int val ):val(val), next(nullptr){};
        Node(int val , Node* next):val(val), next(next){}
    };

    LinkedList() {
        //虚拟节点方便计算
        _dummyHead = new Node(0);
        _maxindex = -1;
    }

    int get(int index) {

        if(index < 0 || index >_maxindex)
            return -1;
        Node * cur = _dummyHead;
        for(int i = 0; i <= index; i++){
            cur = cur->next;
        }
        return cur->val;
    }

    void printList(){
        auto p = _dummyHead->next;
        while(p){
            cout << p->val << ' ';
            p = p->next;
        }
        cout << endl;
    }

    void addAtHead(int val) {
```

```
        Node * newNode = new Node(val, _dummyHead->next);
        _dummyHead->next = newNode;
        _maxindex++;
    }

    void addAtTail(int val) {
        Node * cur = _dummyHead;
        while(cur->next != nullptr){
            cur = cur->next;
        }
        Node * newNode = new Node(val);
        cur->next = newNode;
        _maxindex++;
    }

    void addAtIndex(int index, int val) {
        if(index == _maxindex +1 )
            addAtTail(val);
        else if(index > _maxindex+1)
            return ;
        else{
            Node * cur = _dummyHead;
            for(int i = 0 ; i < index; i++){
                cur = cur->next;
            }
            Node * newNode = new Node(val);
            newNode->next = cur->next;
            cur->next = newNode;
            _maxindex++;
        }

    }

    void deleteAtIndex(int index) {
        if(index < 0 || index > _maxindex)
            return ;
        Node * cur = _dummyHead;
        for(int i = 0 ; i < index; i++){
            cur = cur->next;
        }
        Node * p = cur->next;
        cur->next = p->next;
        delete p;
        _maxindex--;
    }

    void sortList()
    {
        Node* head = _dummyHead->next;
        if (head == NULL || head->next == NULL)
        {
            return;
        }
        Node* left = head;//记录已排序序列的最后一个节点
        Node* cur = head->next;//记录待排序列的第一个节点
```

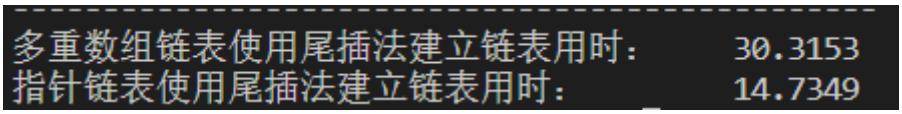
```
        Node* pre = NULL;
        while (cur)
        {
            if (cur->val >= left->val)
            {
                left = left->next;
            }
            else
            {
                pre = _dummyHead;
                while (pre->next->val < cur->val)
                {
                    pre = pre->next;
                }
                left->next = cur->next;
                cur->next = pre->next;
                pre->next = cur;
            }
            cur = left->next;
        }
    }

private:
    int _maxindex;
    Node * _dummyHead;
};
```

为了保证数据准确性，这里的排序用的也是插入排序，并且以下所有的数据集均为随机生成且规模为十万的整形数组，具体代码在.cpp文件中

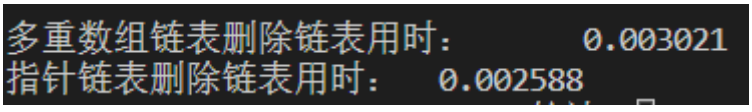
1.头插法建立链表用时对比：



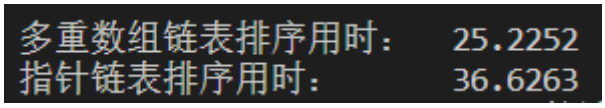
2.尾插法建立链表用时对比：



3.删除全部链表用时对比：



4.链表排序用时对比



3. *分析并提出假设并验证你的猜想。*

1.从头插法建立链表用时对比我们可以知道多重数组链表的构建链表时间是非常快的，其原因显然是因为指针结点构建链表需要耗费大量时间在malloc或者是new上，而数组链表则是用了一个数组来模拟了一个简易的内存分配，就因为它的简易使得数组链表的free和malloc非常的快。

2.从尾插法构建链表用时对比我们发现指针链表更快，二者主要的不同在于访址的方式不同，多重数组链表通过数组下标来简洁访址，而链表通过指针值直接访址，显然前者需要进行内存地址的计算，而后者可以直接访问内存，使得指针链表省去了时间来计算不必要的地址值，而且这个时间掩盖了malloc的缺陷，使得指针链表在索引或者说是访址的速度有着较大的优势

3.删除全部链表用时对比的话其实看不出什么，二者的速度基本上差不多，我这里的多重数组链表因为用的是双向链表，而指针链表用的是单链表，因此多重数组链表用时是偏大的，实际可能差不多也就是0.002秒左右，和指针链表速度差不多。

4.关于链表排序用时对比，发现多重数组链表更快，这个具体原因我其实也不是很清楚，大概可能是给整形变量赋值会比给指针变量赋值要快一些？另外一个可能的原因是程序在水杉上跑的，内存可能比较大导致指针变量字节数更大，cpu需要处理的字节数更多导致排序用时高也是一个可能的原因