研讨报告 1

姓名：郑博远　学号：2154312　日期：2022 年 10 月 21 日

1. 研讨内容

1. 1 静态结构与动态结构的本质区别是什么？

组内讨论：

静态数据结构，如固定大小的数组，由编译器在编译时分配固定大小的、连续的存储空间。其缺点在于分配后空间的长度固定，所分配到的存储空间在程序运行的过程中，其位置和容量都不会再改变。由于无法重新申请空间，需要事先确定所需的最大空间大小。其优点在于，由于内存分配是固定的，所以在添加新项或删除现有项时，不容易产生内存溢出、内存泄漏等问题。

动态数据结构，如链表，针对不确定的总数据存储量来申请空间，分配的内存往往不连续。在程序运行过程中，若问题的数据量发生变化，数据的存储空间的大小能够随之发生变化：若数据量增加，则能够重新向系统申请新的空间；若数据量减少，则也能够将现有的多余的空间归还给系统。其缺点在于可能出现内存溢出、内存泄漏的情况；其优点在于能够更有效地利用内存空间，较于静态结构在空间申请时更为灵活。

个人思考：

在研讨课后查阅相关资料时，我观察到网络上部分资料将数据结构中元素的物理关系作为区分静态结构与动态结构的依据；如有资料指出，静态结构能够 O(1)访问每个元素，而动态结构则需 O(n)。但我认为静态结构与动态结构的本质区别应该在于是否可以申请的空间大小是否能动态改变。上述表述的反例有：静态链表采用数组存储链表，使用与动态申请链表无异，但应该归类于静态结构。

1. 2 对于单链表，带头结点与不带头结点的优缺点是什么？

带头结点的链表优点：

1. 头结点能够存储诸如链表长度等信息；
2. 对于链表中第一个元素的插入与删除会更为便捷，不需要做特殊处理；

1）插入操作

若存在头结点，操作如下：

```
Status List_Insert(Linklist& L, int i, ElemType e){
    p = L; j = 0;
    while(p && j < i - 1){
        p = p->next;
        ++j;
    }
    if(!p || j > i - 1)
        return ERROR;
    s = (Linklist) malloc (sizeof (LNode))
    s->data = e;
    s->next = p->next;
    p->next = s;
    return OK;
}
```

若不存在头结点，操作如下：

```
Status List_Insert(Linklist& L, int i, ElemType e){
    p = L; j = 0;
    while(p && j < i - 2){
        p = p->next;
        ++j;
    }
    if(!p || j > i - 1)
        return ERROR;
    s = (Linklist) malloc (sizeof (LNode))
    s->data = e;
    if(i == 1){
        L = s;      //更新链表的头指针
        s->next = p;
    }
    else{
        s->next = p->next;
        p->next = s;
    }
    return OK;
}
```

2) 删除操作

若存在头结点，操作如下：

```
Status List_Delete(Linklist& L, int i, ElemType e){
    p = L; j = 0;
    while(p->next && j < i - 1){
        p = p->next;
        j++;
    }
    if(!(p->next) || j > i - 1)
        return ERROR;
    q = p->next; p->next = q->next;
    e = q->data; free(q);
    return OK;
}
```

若不存在头结点，操作如下：

```
Status List_Delete(Linklist& L, int i, ElemType e){
    p = L; j = 0;
    while(p && j < i - 2){
        p = p->next;
        ++j;
    }
    if(!(p->next) || j > i - 1)
        return ERROR;
    if(i == 1){
        L = p->next;
        free(p)
    }
    else{
        q = p->next; p->next = q->next;
        e = q->data; free(q);
    }
    return OK;
}
```

3. 能够统一空表与非空表的操作。

使用头结点时，无论表是否为空，头指针都指向头结点，二者操作一致；不使用头结点时，若表非空则头指针指向第一个结点，否则头指针为 NULL。

带头结点的链表缺点：

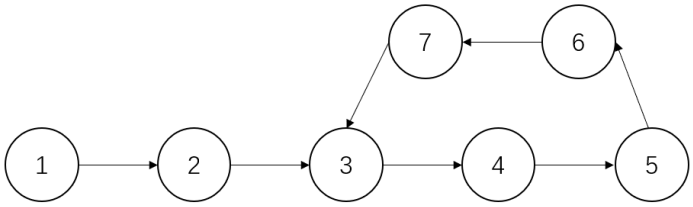
头结点占用多余的内存空间。当链表长度很短或所占空间很小时，头结点是对内存空间的浪费。

1.3 如何判断一个链表是否有环存在？请给出可能的几种解决方案，并进行算法复杂性分析。

组内讨论：

1. 简单的穷举遍历。

将遍历过的所有结点地址存入顺序表中，每遍历到一个新的结点便遍历一次顺序表，若该地址已经被访问过则有环。如下图中，当第二次访问到节点 3 时，顺序表中已经有节点 3 的地址，则说明有环。

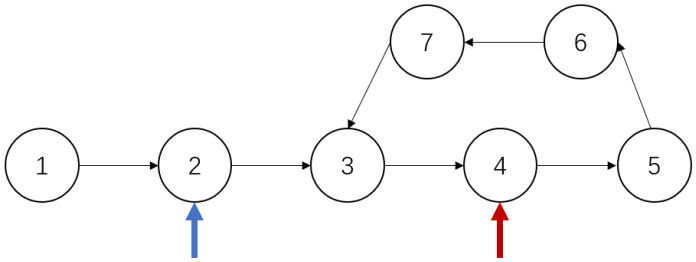


此算法的时间复杂度为 $O(n^2)$，空间复杂度为 O(n)。

2. 哈希表存储。

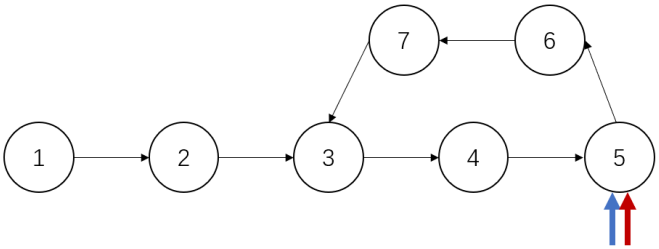
遍历方式与第一种方式类似。但是不采用顺序表来存储，而采用哈希表存储。这样每一次查找结点是否被访问时，可以达到 O(1)的时间复杂度，使得总时间复杂度为 O(n)，空间复杂度为 O(n)。

3. 快慢指针。



用两个指针共同遍历链表。其中快指针（上图红色箭头）每次前进两个结点，

慢指针（上图蓝色箭头）每次前进一个结点。若两个指针相遇，则说明链表中存



在环。如下图中，快慢指针在第五个结点相遇。

若有环循环的次数取决于环的长度，若无环则取决于表长。因此可以近似理解为时间复杂度为 O(n)，空间复杂度为 O(1)。

个人思考：

以下是我对快慢指针方式伪代码的实现：

```
slowpointer = fastpointer = head
while(fastpointer && slowpointer){
    slowpointer = slowpointer->next;
    fastpointer = fastpointer->next;
    if(!fastpointer)    //此处要判断 防止是NULL导致错误
        return false;
    fastpointer = fastpointer->next;

    if(slowpointer == fastpointer)
        return true;
}
return false;
```

若不使用快慢指针方式，则一定有一个指针遍历产生 O(n)的时间复杂度，且最后的总体复杂度取决于每次访问到一个新结点怎么判断当前结点是否被访问过。方式一的顺序表、方式二的哈希表查找元素的时间复杂度分别为 O(n)与 O(1)，因此总时间复杂度为 $O(n^2)$与 O(n)。对于穷举遍历的方式，若不采用顺序表存储，而是每遍历一个结点都另取新指针从头指针开始遍历到当前指针（可以计数来判断是不是同一次访问），则能将空间复杂度降至 O(1)。

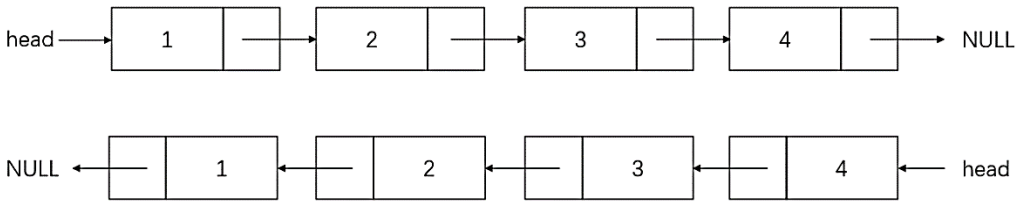
1.4 学生成绩管理：按学号顺序输入，建立成绩表；将其按学号从大到小逆置。可以采用哪些数据结构？如何做？算法复杂性分析。

组内讨论：

1. 顺序表：设表的长度为 length。每次交换顺序表中第 i 个和第 length-i 个结点的内容，需要执行执行 length/2 次，算法时间复杂度为 O(n)；

2. 单链表：具体实现方式如下方伪代码所示。用三个指针来记录，只需要遍历链表一次。因此，算法的时间复杂度为 O(n)。伪代码如下：

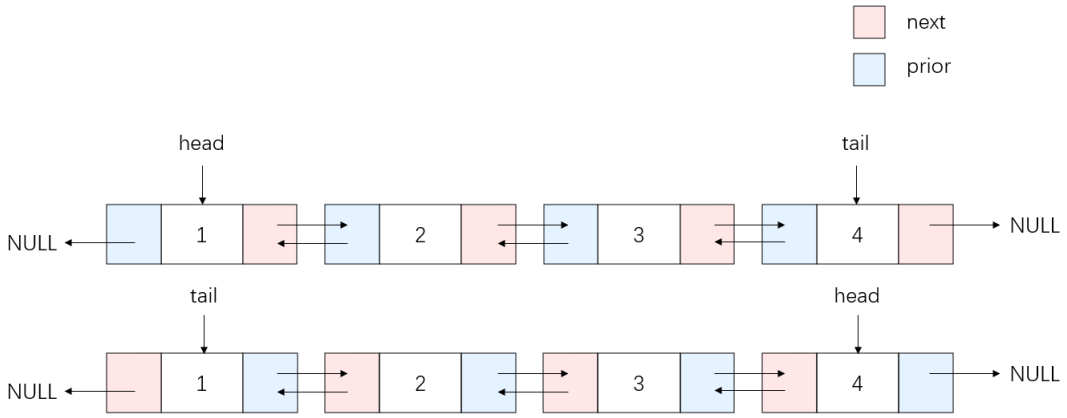
```
pre = NULL;
p = head;
while(p){
    q = p->next;
    p->next = pre;
    pre = p;
    p = q;
}
head = pre;
```



3. 双向链表：先交换头尾指针。仅需一个额外指针变量用于交换，在遍历时逐个交换当前节点的前驱指针和后继指针；因此时间复杂度也为 O(n)；

```
for(p = head; p; p = p->prior)  //交换后的prior即之前的next
    swap(p->next, p->prior);

swap(head, tail);  //交换头尾指针
```



个人思考：

我在 HW1 作业的牌堆倒置中，使用过带头结点的双向循环链表。可以采用一个 bool 变量来表示向后或向前遍历链表。当 bool 值为 true 时，从头结点开始用 p=p->next 来遍历，否则使用 p=p->prior 来遍历链表。逆置时仅需改变 bool 变量即可，因此时间复杂度为 O(1)。需要注意的是，逆置方向后的增添与删除也要对应变化，要注意是使用的是 p->prior 还是 p->next。

实际操作中，学生学号的输入在一次性输入后仍可能有许多次重新增减，会导致输入的学生学号不是严格递增的情况。若要使倒置操作更具有普适性，就需要引入排序算法了。如时间复杂度为 $O(n^2)$的冒泡排序、选择排序，时间复杂度为 O(nlogn)的堆排序、快速排序等。

1. 5 医院看病排队管理

组内讨论：

采用的数据结构：队列。

队列中队首元素代表当前正在排队的第一个病人，队尾元素代表当前正在排队的最后一个病人。由于医院中有许多不同的科室，每个科室单独排队；因此每个科室应该单独配置上述队列。该队列支持如下操作：

（1）开始问诊：允许开始病人挂号；

（2）挂号：将病人的个人信息（病历号、姓名、年龄等）入队；

（3）就诊：将队首病人信息出队。若当前出队的病人不在，则继续出队当前队首（队列为空除外），过号的病人需要重新挂号；

（5）屏幕打印就诊信息：遍历从队首到队尾的所有病人信息并打印；

（6）停止挂号：禁止新病人信息入队，其余功能正常；

（7）结束问诊：清空当日队列信息。

个人思考：

考虑到有军人优先等特殊情况，为军人等有优先就诊权利的病人另开一个队列，功能与上述相同。当优先就诊队列为空时，再进行普通就诊队列的就诊。

```
struct info{
    string ID;      //病人病历号
    string name;    //病人姓名
    int age;        //病人年龄
    bool priority;  //病人是否有优先就诊特权
}

class HospitalQueue{
private:
    SqQueue NormalQueue;      //普通排队
    SqQueue PriorityQueue;    //优先排队
    bool IsEnableRegister;    //当前是否支持挂号

public:
    Status Registration(info& e);                              //挂号
    Status Visit(info&e, bool (*Iswaiting)(info& e));          //就诊
    Status Print();                                            //屏幕打印就诊信息
    Status DisableRegister();                                  //停止挂号
    Status StartVisiting();                                    //问诊开始
    Status QuitVisiting();                                     //问诊结束
 }
```

```
Status Registration(info& e){
    //当日还能挂号
    if(IsEnableRegister){
        if(e.priority)
            ret = PriorityQueue.enqueue(e);
        else
            ret = NormalQueue.enqueue(e);
        return ret;
    }
    else
        return ERROR;
}

Status Visit(info&e, bool (*Iswaiting)(info& e)){
    //若病人不在等待，则就诊队首下一位病人
    while(true){
        //没有排队的病人
        if(PriorityQueue.empty() && NormalQueue.empty())
            return ERROR;
        //先考虑有优先就诊特权的队列（军人、老年人）
        if(!PriorityQueue.empty()){
            ret = PriorityQueue.pop(e);
            if(Iswaiting(e))
                return ret;
        }
        else{
            ret = NormalQueue.pop(e);
            if(Iswaiting(e))
                return ret;
        }
    }
}


Status Print(){
    if (PriorityQueue.print() == ERROR)
        return ERROR;
    if (NormalQueue.print() == ERROR)
        return ERROR;
    return OK;
}

Status DisableRegister(){
    //禁止新挂号
    IsEnanbleRegister = false;
}

Status StartVisiting(){
    //允许开始挂号
    IsEnanbleRegister = true;
}

Status QuitVisiting(){
    //清空剩余的病人信息
    PriorityQueue.clear();
    NormalQueue.clear()
    IsEnanbleRegister = false;
}
```

2. 研讨总结与建议

本次研讨课我很荣幸有机会作为线上小组的代表参加了问题讨论结果的汇报。由于疫情的原因，我在线上小组与未能返校的同学们一起参加；尽管不能面对面交流，但大家的讨论热情十分高涨，都能积极提出自己的看法。本次研讨课营造了良好的讨论氛围，既有组内的看法交流，也有不同组之间互相分享解题思路，让我在巩固所学的知识点之余也能够更深入地去思考问题。

对于之后的研讨课，我有几点小建议：

1. 题目难度的设置上可以更有梯度。开始的题目难度可以设计的较为简单，激发同学们的讨论热情。题目难度呈阶梯上升，让同学们在讨论的过程中能够循序渐进地拓宽思路。不同梯度难度的题目能够保证同学们基本的参与度，同时也能够拓宽知识面，学习到自己所不了解的数据结构知识；

2. 对于部分难度较大或者涉及背景比较复杂的讨论题目，可以在研讨课前事先公布。可以选择小组轮流或者自愿报名的方式，对应小组在课前做好准备工作，在上课时向其他同学以介绍的方式进行知识的传授；

3. 可以在研讨课前通过共享文档的方式邀请同学们填写自己在学习该章节时所遇到的困惑之处，或是想要提出的认为有讨论价值的问题。老师或者助教学长学姐们可以挑选同学们比较有共鸣的问题或是含金量比较高的问题作为研讨课的讨论题目。通过这样的方式，能够更好地解决同学们在数据结构课程学习过程中所遇到的难点与不解之处；

4. 由于人数较多，同学们分散在不同的教室，可能有许多闪光的思路只被所在教室的部分同学所了解。可以考虑将比较优秀的思路整理成文档发送在班级的 QQ 群中，让所有同学都能够更深入地了解到优秀的解法思路，有所收获。