第一次研讨课报告

1. 静态结构和动态结构的本质区别是什么?

Solution:

- 存储空间大小是否在使用前已经被定义好
  - 静态结构的存储空间大小在使用前已经被定义好，在程序执行的过程中无法改变。比如顺序表等等

    > 顺序表满后，再次申请不并入动态结构，而是视作重新申请了静态结构

- 总的数据存储量大小是否确定
  - 动态结构不确定总的数据存储量，在程序开始时有一个初始大小的空间。随着程序的运行，如果需要存储的数据数量超过初始空间大小的话，程序会动态内存申请更大的空间来存储数据（在机器内存允许的情况下）。
  - 动态结构存储空间会随着数据数量的改变而动态变化，而静态结构在数据量超过预期值的时候会发生数据的溢出。

2. 对于单链表，带有头结点与不带有头结点的优缺点是什么?

Solution:

- 头结点可以统一空链表与非空链表的操作
  - 带头节点的单链表的初始化以及判断为空:

```
bool initList(LinkList& head)
{
    head->next = NULL;
    return true;
}

bool isEmpty(LinkList head)
{
    return head->next == NULL;
}
```

  - 不带头节点的单链表初始化以及判断为空:

```
bool initList(LinkList& L)
{
     L = NULL;
    return OK;
}

bool isEmpty(LinkList L)
{
    return L == NULL;
}
```

- 头节点可以将在第一个元素节点前插入删除的操作与其他位置的插入删除操作统一起来
  - 带头节点的单链表的插入删除操作

```
bool insertNode(LinkList& L, Node* x)
{
    ... //找到要插入位置的前驱节点p
    if (x == NULL) return ERROR;
    x->next = p->next;
    p->next = x;
    return  OK;
}

bool deleteNode(LinkList& L)
{
    ... //找到要删除位置的前驱节点p
    Node* temp = (Node*)malloc(sizeof(Node));
    if (temp == NULL) return ERROR;
    temp = p->next;
    p->next = x->next;
    free(temp);
    return OK;
}
```

> 这里p可以是头节点，这样就是插入该链表的第一个元素
>
> 这样可以将第一个元素的插入与其他位置的插入统一起来
>
> 同样对于删除操作也同理

  - 不带头节点的单链表的插入删除操作

```
bool insertHead(LinkList& L, Node* x)
{
    x->next = L;
    L = x;
    return OK;
}

bool insertNotHead(LinkList& L, Node* x)
{
    .... //找到要插入位置的前驱节点p
```

```
    x->next = p->next;
    p->next = x;
    return OK;
}

bool deleteHead(LinkList& L)
{
    Node* temp = (Node*)malloc(sizeof(Node));
    if (temp == NULL) return ERROR;
    temp = L;
    L = L->next;
    free(temp);
    return OK;
}

bool deleteNotHead(LinkList& L)
{
    ... //找到要删除位置的前驱节点p
    Node* temp = (Node*)malloc(sizeof(Node));
    if (temp == NULL) return ERROR;
    temp = p->next;
    p->next = x->next;
    free(temp);
    return OK;
}
```

> 可以看到，在删除第一位置元素时进行的操作和在其他位置进行的操作并不相同，造成形式上的不统一

- 可以利用头节点的数据域进行链表整体数据的存储
  - 链表的长度(随链表增删节点动态修改)

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

Solution 1:

- 可以考虑在原链表存储的数据类型结构体中加一个 布尔型的变量作为标记，初始置为 false
- 在遍历链表的过程中每遍历到一个节点时都判断一下对应标记的值，如果标记的值为 false 就将其改为 true 并继续遍历
- 如果访问到的标记的值为 true 则说明单链表中有环存在
- 复杂度分析
  - 时间复杂度：若链表中有n个节点，则此方法时间复杂度为O(n)
  - 空间复杂度：对每一个结构体添加布尔型变量会有空间开销

- 节点类

```
typedef int EleType;  // 原有的数据域

struct ListNode
{
    EleType data;
    bool vis;  //标记是否被访问过的变量
};

typedef ListNode* LinkList;
```

- 判环函数

```
int isLoop(LinkList head)
{
    ListNode* ptr = (ListNode*)malloc(sizeof(ListNode));
    if (ptr == NULL) return ERROR;
    for (ptr = head->next; ptr->next != NULL; ptr = ptr->next)
    {
        ptr->vis = true;
        if (ptr->next->vis) return true;
    }
    return false;
}
```

Solution 2:

- 链表中的节点的地址总是不相同的，则可以在遍历过程中将遍历过的节点存储到其他容器中
- 在这个容器中遍历，看是否此地址已经在容器中
- 如果在则说明这个地址已经被访问过，则原链表是有环的
- 如果不再就把当前节点的地址存到这个容器中，继续遍历下一个节点
- 复杂度分析
    - 时间复杂度：
        - 顺序表作为容器则时间复杂度为O(n^2)
        - 将存储地址按照大小进行排列，利用二分查找的方式寻找，则进行排序时间复杂度可以为O(nlogn), 查找的时间复杂度为O(logn)
        - 利用STL中集合进行判重，如果添加前后的集合元素个数没有变化，则说明有重复的元素，时间复杂度接近O(n)
    - 空间复杂度
        - 由于有另外存储的容器，则会有容器大小的空间开销

- 节点类：同**Solution 1**
- 判环函数

```
#include <set>

bool isLoop(LinkList head)
{
    LinkNode* ptr = head;
    set<ListNode*> s;

    while (ptr != NULL)
    {
        if (s.count(ptr)) return true;

        s.insert(ptr);
        ptr = ptr->next;
    }
    return false;
}
```

Solution 2优化

- 查找地址时可以使用哈希表进行查找，即使用unordered_set容器进行查找
- 复杂度：
    - 时间复杂度：大致接近于O(n)
    - 空间复杂度：另需容器的空间开销

Solution 3:

- 快慢指针法
- 创建两指针指向链表头节点，开始遍历
- 遍历时快指针向下移动两个节点，满指针向下移动一个节点
- 如果链表中有环，则当两个指针都进入到这个环当中移动后一定会有一时刻两个指针指向一个节点
- 如果链表中没有环，则两个指针不会相遇
- 复杂度分析
    - 时间复杂度：O(n) 主要取决于环的大小（若速度差能改变同时取决于速度差），其决定了相遇时间
    - 空间复杂度：指针操作，除指针空间外不会产生额外开销

- 节点类：同**Solution 1**
- 判环函数

```
bool isLoop(LinkList head)
{
    ListNode* slowptr, fastptr;
```

```
    slowptr = fastptr = head;

    while (fastptr != NULL && fastptr->next != NULL)
    {
        slowptr = slowptr->next;
        fastptr = fastptr->next;

        //in case of overflow && the linkedlist ends where NULL can be
reached
        if (fastptr == NULL || fastptr->next == NULL) return false;

        fastptr = fastptr->next;

        if (slowptr == fastptr) return true;

    }
    return false;
}
```

> 可以提前进行NULL的判断，如果能到达NULL，其实就能够说明无环了

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。

Solution 1:

- 顺序表原地操作，将首尾对应元素交换，达到逆置的效果
- 复杂度分析
  - 时间：O(n / 2)，因为进行的是表长一半的操作

- 逆序函数

```
void reverseList(int& arr[], int len)  //学号顺序表与表长
{
    for (int i = 0; i < len / 2; i++)
    {
        swap(arr[i], arr[len - 1 - i]);
    }
}
```

Solution 2:

- 用双向链表存储学号，逆序遍历 或者 单链表直接头插法就可以将其逆置
- 复杂度分析
  - 时间复杂度：O(n)，涉及到建表和遍历的过程
  - 空间复杂度：链表的长度即为空间开销

- 采用头插法
- 节点

```
struct Node
{
    Node* next;
    int data;
};

typedef Node* LinkList;
```

- 逆序函数

```
int reverseList(int n)   //学生数量
{
    //set up
    LinkList head = (Node*)malloc(sizeof(Node));
    head->next = NULL;

    //head insert
    for (int i = 1; i <= n; i++)
    {
        Node* temp = (Node*)malloc(sizeof(Node));
        if (temp == NULL) return ERROR;
        temp->next = head->next;
        head->next = temp;
    }

    //print node
    for (Node* ptr = head->next; ptr != NULL; ptr = ptr->next)
    {
        cout << ptr->data << endl;
    }

    return OK;

}
```

Solution 3:

- 用双向链表进行存储，但此双向链表具有特殊结构：
- 节点中存储含有两个指针的一维数组，此数组第一位是指向前驱的指针，第二位是指向后继的指针
- 前序和后序遍历只需要更改取用指针数组的下标，就可以实现前序和后序遍历

- 节点类

```
struct Node
{
    Node* ptr[2];
    int data;
};

typedef Node* LinkList;
```

- 遍历操作

```
void reverseList(int n)
{
    int dir = 1;  //指针数组第二位指向后继

    LinkList head = (Node*)malloc(sizeof(Node));
    head->ptr[dir] = head->ptr[1 - dir] = NULL;


    //rear insert
    Node* rear = (Node*)malloc(sizeof(Node));
    rear = head;
    for (int i = 1; i <= n; i++)
    {
        Node* temp = (Node*)malloc(sizeof(Node));
        if (temp == NULL) return ERROR;
        temp->ptr[dir] = NULL;
        rear->ptr[dir] = temp;
        temp->ptr[1-dir] = rear;
        rear = temp;
    }

    //output
    dir = 1 - dir;  //切换模式
    for (Node* bptr = rear; bptr != NULL; bptr = bptr->ptr[dir])
    {
        cout << bptr->data << endl;
    }

}
```

5.医院看病排队管理

Solution 1:

- 使用队列的数据结构
- 对于病情不同程度的一系列病人建立不同优先级的队列

- 病人结构体

```
struct Patient
{
  int pri;
  string name;
};
```

- 队列容器

```
vector<queue<Patient>> sys;
int N = 5;  //优先级个数
for (int i = 1; i <= N; i++)
{
    queue<Patient> q;
    sys.push_back(q);
}
```

- 按照病人优先级进入队列

```
int n;  //病人数量
for (int i = 1; i <= n; i++)
{
    Patient* pa = (Patient*)malloc(sizeof(Patient));
    ...//input info
    sys[pa->pri - 1].push(*pa);
}
```

- 按照队列优先级出队列

```
for (int i = 0; i <= N; i++)
{
    cout << sys[i].front()->name << endl;
    sys[i].pop();
}
```

Solution 1 改进：

- 方便特殊情况进行插入，则只需将队列改为链表即可
- 对于同一优先级的病人按照到来的先后顺序进行链表的尾部插入
- 需要特殊擦汗如的情况可以直接再链表中进行插入即可。
- 系统顺序表中存储的时每一个链表的头节点

- 排队系统数据结构
- 病人

```
struct Patient
{
    Patient() : next(NULL){}
    int pri;
    string name;
    Patient* next;
};

struct waitQueue()
{
    waitQueue() : head(NULL), rear(NULL){}
    Patient* head;
    Patient* rear;

};
```

- 排队系统

```
vector<waitQueue> sys;
int N = 5;  //优先级个数
for (int i = 1; i <= N; i++)
{
    Patient* pa = (Patient*)malloc(sizeof(Patient));
    sys.push_back(pa);
}
```

- 录入信息

```
int n;  //病人个数

//链表插入
void insert(waitQueue& wq, Patient* pa)
{
    pa->next = NULL;
    rear->next = pa;
    rear = pa;
}
```

```
for (int i = 1; i <= n; i++)
{
    Patient* pa = (Patient*)malloc(sizeof(Patient));
    scanf("%d%s", &pa->pri, &pa->name);
    insert(sys[pa->pri - 1], pa);
}
```

Solution 2:

- 可以将不同病情都统一成统一的一个队列来维护，可使用优先队列的方式来进行实时的排序
- 排序方式是多关键字排序，根据病情严重程度进行第一关键字排序，第一关键词相等时通过挂号时间进行排序
- 当医生空闲时，可以自行从等待优先级队列中获取患者信息（生产者消费者模式），拥有优先队列作为缓冲区来作为两个进程（患者挂号和医生问诊）之间的沟通桥梁

- 患者结构体信息

```
struct Patient
{
    Patient(int pri, int t) : priority(pri), time(t){}

    bool operator< (const Patient& p) const
    {
        if (this->priority == p.priority)
        return this->time > p.time;
        return this->priority < p.priority;
    }

    int priority;   //病情优先级
    int time;       //到达时间
    //... 其他信息
};
priority_queue<patient> q;
```

- 获取信息,实现医院看病排队管理

```
int n;  //患者个数
for (int i = 1; i <= n; i++)
{
    int pri, t;
    scanf("%d%d", &pri, &t);
    q.push(Patient(pri, t));
}
```

6. 研讨课建议

- 各个小组需要有记录员，将小组提出的想法进行记录汇总，防止反复多遍对于之前讨论内容的重复
- 可以先按照某一固定小组顺序讲题，进行到新的问题时先让轮到的小组汇报，轮到的小组没有汇报内容其他小组再进行补充，防止出现某几个小组过于积极或者小组都不积极的情况
- 在进行完一个题目汇报之后，老师可以适当的进行一些总结，给出一些建议，防止出现思路错误但是还作为方法被记录的情况
- 同时如果在同学汇报过程中有一些超前的知识点，老师可以简单的进行一些讲解，让同学们简单了解算法/思路即可
- 研讨课越办越好！