第一次研讨课作业

AUTHOR: 林日中(1951112), DATE: 11/07/2020

0. 目录

1. 静态结构和动态结构的本质区别。

本质区别在于所需存储空间是否连续。

静态结构的存储空间是连续的。所有元素存放在一个连续的存储空间中，用物理上的相邻表达逻辑上的相邻关系。动态结构的存储空间是不连续的。每个元素的存储地址动态分配，通过指针来表达逻辑上的相邻关系。

2. 对于单链表，带头结点和不带头结点的优缺点。

带头结点的单链表

- 优点： 对结点的操作一致，即不需要对第一个有有效数据的结点进行特殊判断和操作；在第一个有效元素前插入元素、删除第一个有效元素时，不需要特殊处理。
- 缺点： 头结点的数据域未被利用，特别在结点的数据域特别大的情况下，会有较大的空间浪费。补救方法可以是在头结点的数据域记录一些链表的基本信息，如链表的总长度、链表的描述等。

不带头结点的单链表

- 优点： 不会造成空间浪费。
- 缺点： 若元素的添加、修改、删除等操作涉及到第一个结点，需要进行特殊处理。若链表为空，需要做特殊判断。

3．学生成绩管理，按照学号顺序输入，建立成绩表，按学号大小逆置。

> *可采用哪些数据结构？如何做？算法复杂度分析。*

题目的前提是“按照学号顺序输入”，那么 此时成绩表的顺序已经是按学号从小到大 。若要实现按学号大小逆置，只需反转建立的成绩表即可。针对成绩表反转的需求，我们可以想到以下几种比较简便的方法，分别使用了顺序队列、链式队列、双向链式队列。

① 反转顺序队列

此方法使用的数据结构为 顺序队列 。顺序队列反转的相关类C++代码如下。

```
new_base = new Student[len];
for (int i = len; i; i--)
{
    new_base[i - 1] = base[len - i];
}
delete[] base;
base = new_base;
```

相关算法的时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

② 反转链式队列

此方法使用的数据结构为 链式队列 。链式队列反转的相关类C++代码如下。

```
while (cur)
{
    tmp = cur->next;
    cur->next = tmp_pre;
    tmp_pre = cur;
    cur = tmp;
}
```

相关算法的时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

③ 带有方向标记的双向队列

此方法使用的数据结构为**链式双向队列**。该结构中含有一个代表队列顺序的标志 `bool from_head_to_tail` ，若为 `true` ，则队列的遍历顺序为从 `head` 到 `tail` ；否则为从 `tail` 到 `head` 。 具体的实现代码如下。

```cpp
struct Student
{
    char *no = nullptr;
    char *name = nullptr;
    struct Student *prior = nullptr;
    struct Student *suc = nullptr;
    Student(const char *_no = "", const char *_name = "")
        : no(new char[strlen(_no) + 1]), name(new char[strlen(_name) + 1])
    {
        strcpy(no, _no);
        strcpy(name, _name);
    }
    struct Student *next(bool to_next = true) const { return to_next ? suc : prior; }
}; // struct Student

class StudentQueue
{
protected:
    struct Student *head = nullptr;
    struct Student *tail = nullptr;
    bool from_head_to_tail = true; // when it is false, the sequence of the queue is from tail to head

public:
    StudentQueue() : head(new Student()), tail(head) { /* create the student queue */ }
    void reverse() { from_head_to_tail = !from_head_to_tail; }
    // 💡 this is an example of traversing the list
    bool traverse(bool (*visit)(struct Student &))
    {
        struct Student *cur = from_head_to_tail ? head : tail;
        while (cur && visit(*cur))
        {
            cur = cur->next(from_head_to_tail);
        }
        return !cur; // when the current node is not nullptr, the traverse process does not end normally
    }
}; // class StudentQueue
```

相关算法的时间复杂度为 $O(1)$ 。

④ 栈

此方法使用的数据结构为 栈。学生对象按照学号顺序进栈，然后会按照 LIFO 的原则逆序出栈。相关代码如下。

```
std::stack<Student *> stu_stack;
stu_stack.push(/* ... */);
// ...
if (!stu_stack.empty())
{
    Student *cur = stu_stack.top();
    stu_stack.pop();
}
```

进栈、出栈操作的时间复杂度都为 $O(1)$ 。

4. 医院看病排队管理。

病人们在医院里排队的序列，可以被抽象为一个队列，队列元素是病人，元素先进先出，即先排队的病人先得到诊治。然而，会有许多需要特殊处理的情况发生——在医院的急诊室里，根据病人的病情严重程度对病人进行诊治；军人等特殊群体需要先得到诊治等等。这可以用 优先队列 来模拟。本程序利用 STL 中的 `std::priority_queue` 来模拟该过程。

病人在医院排队，可以被抽象为2个阶段：

1. 病人挂号，确定进入队列的时间 `enter_time` 和紧急程度 `severity_level` ，进入队列。
2. 医生或者服务台的其他医护人员负责从队列中取出元素（病人），即队首元素出队就诊。

不同元素间的先后关系，由两个成员变量 `enter_time` 和 `severity_level` 决定，具体的规则是：紧急程度越高的病人越优先；若紧急程度相同，则比较他们进入队列的先后顺序，先排队先就诊。

病人的基本结构如下。

```
struct Patient
{
    /* ... name, age, sex, symptoms and other basic information ... */
    time_t enter_time = (time_t)-1;
    unsigned int severity_level = 0U;
    Patient(/* ... basic information ... */, const time_t e_time, const unsigned int s_level) : enter_time(e_time), severity_level(s_level) {}
}; // struct Patient
```

病人之间的比较函数对象的结构如下。

```
// highest severity level has highest priority; if levels of the two patients are equal, first in line, first seeing the doctor
struct cmp
{
    bool operator()(const Patient *left, const Patient *right) const { return left->severity_level > right->severity_level || (left->severity_level == right->severity_level && left->enter_time < right->enter_time); }
}; // struct cmp
```

优先队列的基本结构如下。

```
std::priority_queue<Patient *, std::vector<Patient *>, cmp> pq;
```

病人挂号，进入排队时的相关操作如下。

```
Patient *new_patient = new Patient(/* ... */);
pq.push(new_patient);
```

队首元素出队的相关操作如下。

```
if (!pq.empty())
{
    Patient *current_patient = pq.top();
    pq.pop();
}
else
{
    /* corresponding actions */
}
```

时间复杂度

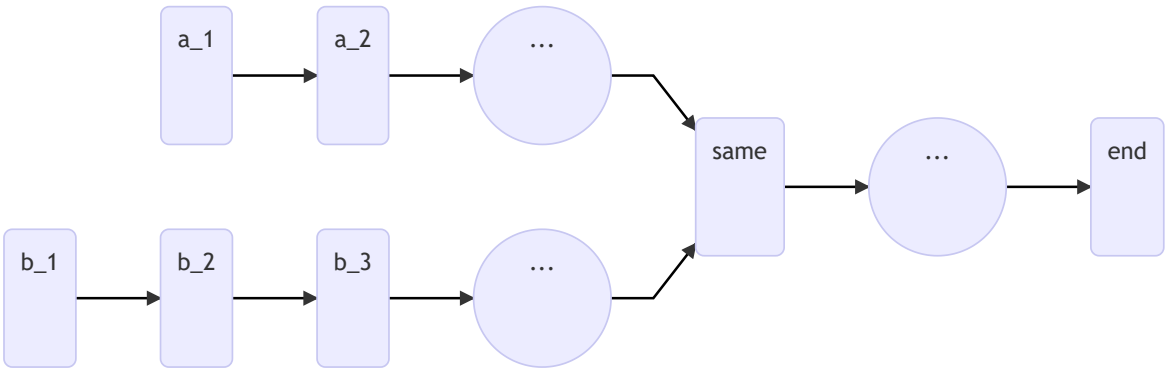
普通的队列是一种先进先出的数据结构，元素在队列尾追加，而从队列头删除。在优先队列中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先队列具有最高级先出 （first in, largest out） 的行为特征。通常采用堆数据结构来实现。

priority queue operation:

- add : adds in order; $O(1)$ average, $O(\log n)$ worst
- peek : returns **minimum (or maximum)** element; $O(1)$
- remove : removes/returns **minimum (or maximum)** element; $O(\log n)$ worst
- isEmpty , clear , size , iterator : $O(1)$

5. 算法题：编程判断两个单向链表是否相交。

（1）请画出链表相交示意图。



（2）给出算法思想，并分析时间复杂度。

记两个链表为 `a` 和 `b` ，它们的长度分别为 `m` 和 `n` 。

① “暴力”解法

从头到尾分别遍历两个链表，若发现相同元素则两链表交；否则不相交。

- 时间复杂度： $O(n*m)$

② 使用栈

创建两个栈，第一个栈存储第一个链表的结点，第二个栈储第二个链表的结点。每遍历到一个结点时，就将该结点栈。两个链表都入栈结束后。则通过 `top()` 判断栈顶结点是否相等即可判断两个单链表是否相交。

- 时间复杂度： $O(n+m)$

③ 哈希表法

两链表一旦相交，相交结点一定有相同的内存地址。遍历一个链表，并利用地址建立哈希表；遍历第二个链表，看地址哈希值是否和第一个表中的结点地址值有相同。若有同，则相交；否则不相交。

- 时间复杂度： $O(n+m)$

④ 遍历法

此方法与 方法 ② 类似，但是不需要使用栈。直接遍历两个链表至最后一个结点。若两链表最后一个结的地址值相同，则两链表相交；否则不相交。

- 时间复杂度： $O(n+m)$

6． 数据结构设计：队列中取最大值的问题。

> 假设有这样一个拥有三个操作的队列 ：
>
> - （1） `Enqueue(v)` ： v 入队
> - （2） `DeQueue()` ：使队首元素删除并返回此元素
> - （3） `GetMax()` ：返回队列中的最大元素
>
> 请设计一种数据结构和算法，让 `GetMax` 操作的时间复杂度尽可能地低。

当一个元素进入队列的时候，它前面所有比它小的元素就不会再对队列中的最大值产生影响。

如果我们向队列中插入数字序列 `1 1 1 1 2` ，那么在第一个数字 `2` 被插入后，数字 `2` 前面的所有数字 `1` 将不会对结果产生影响。因为按照队列的取出顺序，数字 `2` 只能在所有的数字 `1` 被取出之后才能被取出，因此如果数字 `1` 如果在队列中，那么数字 `2` 必然也在队列中，使得数字 `1` 对结果没有影响。

按照上面的思路，我们可以设计这样的方法：从队列尾部插入元素时，我们可以提前取出队列中所有比这个元素小的元素，使得队列中只保留对结果有影响的数字。这样的方法等价于要求维持队列单调递减，即要保证每个元素的前面都没有比它小的元素。

以下是数据结构的实现代码。考虑到描述简洁的问题，该结构利用了 STL 内的 `std::queue` 和 `std::deque` 。经检验，该数据结构能够正常运行，并且其 `GetMax()` 操作的时间复杂度也成功达到了 $O(1)$ 。

```
class MaxQueue
{
    std::queue<int> data;
    std::deque<int> max_value;

public:
    MaxQueue() {}
    int GetMax() const { return max_value.empty() ? -1 :
max_value.front(); }
    void EnQueue(int value)
    {
        while (!max_value.empty() && max_value.back() < value)
        {
            max_value.pop_back();
        }
        max_value.push_back(value);
        data.push(value);
    }
    int DeQueue()
    {
        if (data.empty())
        {
            return -1;
        }
        int ans = data.front();
        if (ans == max_value.front())
        {
            max_value.pop_front();
        }
        data.pop();
        return ans;
    }
}; // class MaxQueue
```

- 时间复杂度：$O(1)$（插入，删除，求最大值）
- 空间复杂度：$O(n)$，需要用队列存储所有插入的元素。

7. 进一步思考和对课程的建议

数据结构课程研究的主要是研究非数值计算的研究的程序设计问题中所出现的计算机处理对象以及它们之间关系和操作。在理论课上，我们学习的是各种数据结构的概念、实现方法和应用场景；但是在研讨课上，我才比较彻底地理解清楚概念，开始真正地思考各种场景对应的最佳数据结构，应该如何根据具体场景实现、如何把时间 / 空间复杂度降到最低。

一开始我对静态链表是静态还是动态结构不太清楚。它的特点是不是和动态 / 静态结构的特点相悖？它的存储空间是一开始开辟好的，但数据元素间的逻辑关系并不是靠数据的存储位置反映的。但是经过了研讨课，我知道了静态、动态结构的本质区别是所需存储结构是否连续。所以，我现在认为静态链表是静态结构，因为它使用的内存是连续的。

第3题，要求的是实现一个具有反转功能的学生表数据结构，所以有很多种可用的基本数据结构：顺序表、单链表、双向链表、栈等。但是在实际应用中，对学生的管理肯定不是只有逆置操作，在考虑其他操作的情况下，一些数据结构就变得不适用—比如栈，除了适合逆置，其他任何操作都非常麻烦。

我的建议是，在理论课中，老师在讲解一种数据结构之前和之后，都可以系统地整合数据结构的各种概念，这样我们能对基础概念理解得更加透彻；在研讨课上，老师可以考虑在讨论课前先将题目发布，以便提前进行相关的调查准备，交流时可以更充分有理。