作业HW1 报告

1.涉及数据结构和相关背景

1. 1 线性表

- 掌握关于线性表的逻辑结构定义，抽象数据类型定义以及各种存储结构的描述方法
- 在线性表的两类存储结构（顺序存储与链式存储）上完成实现增删改查的基本操作

1.2 线性表的分析

- 顺序表
  - 优点
    - 结构简单
    - 存储效率高
    - 直接存取结构
  - 缺点
    - 插入删除移动大量元素，效率低
    - 表长难以估计，容易溢出或者浪费空间
- 链表
  - 单链表、双链表、循环链表
  - 优点
    - 存储空间动态分配
    - 空间利用率高
  - 缺点
    - 查找时间劣于线性表
- 经常性
  - 查找：线性表
  - 插入删除：链表
- 综合运用两种数据结构能获得更好的时间空间性能

2. 实验内容

2.1 顺序表去重

2.1.1 问题描述

完成顺序表的去重运算，即将顺序表中所有重复的元素只保留第一个，其他均删除。

2.1.2 基本要求

- 输入要求：给定的线性表中的元素在32位int范围内，即给出的线性表元素均为整形，且给出的顺序表长度不超过50000
- 输出要求：输出一行表示去重后的列表中的所有元素

2.1.3 数据结构设计

定义一个结果数组（顺序表），每次输入一个数据时首先通过遍历结果数组判断输入的数字是否在结果数组内，如果在则打上标记，不会被加入结果数组中，否则加入结果数组中，这样也同时保证了保留的元素是第一个此元素。

2.1.4 功能说明

```
scanf("%d", &n);  //读入表长
for (int i = 1; i <= n; i++)
{
    scanf("%lld", &a);
    int flg = 0;
    for (int i = 0; i < vis.size(); i++)  //循环判断新进来的元素是否在结果数组内
    {
        if (vis[i] == a) flg = 1;
    }

    if (!flg) // 每日有在结果数组内就加入结果数组
    {
        vis[++total] = a;
    }

}

for (int i = 0; i < total; i++)
{
    printf("%lld ", vis[i]);
}
```

2.1.5 调试分析

- 在进行实验时，直接使用long long 型进行数组元素的接取，其实没有必要，因为数组元素是32位以内，都是int范围内的。
- 在尝试时曾经想要开一个访问数组，想要在已经访问过对应的元素上进行标记，查重时直接进行访问，降低时间复杂度到On级别， 但是由于后续对题目数据规模的修改，无法开到10^9级别的访问数组，原计划无法进行。

2.1.6 总结和体会

因为数据规模的原因，本题题解的时间复杂度达到了On^2级别，需要进行二重循环来进行查重操作，如果数据规模仅在10^6级别，可以将时间复杂度降低到On级别。

2.2 学生信息管理

2.2.1 问题描述

本题希望通过顺序表实现对学生信息的录入、删除、按学号查找、按姓名查找等操作。

2.2.2 基本要求

- 输入要求：输入学生总数，进行学生信息初始化录入

  分别实现以下函数

```
insert i 学号 姓名: 表示在第i个位置插入学生信息，若i位置不合法，输出-1，否则输出0
remove j:表示删除第j个元素，若元素位置不合适，输出-1，否则输出0
check name 姓名y：查找姓名y在顺序表中是否存在，若存在，输出其位置序号及学号、姓名，若不存在，输出-1。
check no 学号x：查找学号x在顺序表中是否存在，若存在，输出其位置序号及学号、姓名，若不存在，输出-1。
end: 操作结束，输出学生总人数，退出程序。
```

2.2.3 数据结构设计

- 本题用顺序表会超过时间限制，所以此处用链表来实现频繁的插入删除操作，时间效率更高，同时不用担心空间溢出等问题

2.2.4 功能说明

- 结构体存储学生节点信息

```
struct StudentNode
{
    string name;  //学生姓名
    string id;  //学生学号
    StudentNode* next;  //指针域
};
```

  也可以通过pair进行存储

```
pair<string, stirng>  //first: 学生姓名,  second: 学生学号
```

- 删除函数：将链表中的一个节点删除

```
int remove(StudentNode* head, int i)
{
    /*
    *@para head 链表的头节点
    *@para i 要进行删除的学生编号
    */
    /*
    实现：遍历到待删除节点前驱进行指针变换删除链表中编号为i的节点，同时释放内存，链表长度减一
    输出：删除成功返回1，否则返回0（没有查询到有这位学生）
    */
}
```

- 插入函数：

```
int insert(StudentNode* head, int i)
{
    /*
    *@para head 链表的头节点
    *@para i 要进行插入的学生编号
    */
    /*
    实现：遍历到相应插入位置，申请内存，进行指针变换插入相应的节点，链表长度加一
    输出：插入成功返回1，否则返回0（插入位置不正确，插入位置范围：[1, length + 1]）
    */
}
```

- 按照学生姓名查询函数

```
int checkName(StudentNode* head, string name)
{
    /*
    *@para head 链表的头节点
    *@para name 要查询的学生姓名
    */
    /*
    实现：遍历列表，检查链表每一个节点中存储的姓名是否与所给姓名相同
    输出：查找到此位学生返回1，否则返回0
    */
}
```

- 按照学生学号查询函数

```
int checkID(StudentNode* head, string id)
{
    /*
    *@para head 链表的头节点
    *@para id 要查询的学生学号
    */
    /*
    实现：遍历列表，检查链表每一个节点中存储的姓名是否与所给姓名相同
    输出：查找到此位学生返回1，否则返回0
    */
}
```

2.2.5 调试分析

- 最初尝试进行线性表的尝试，由于查询时间过长，导致测试段测试点超过时间限制，后续进行了存储优化，但在学生人数较多的情况下每一步删除和添加的操作都会造成时间超限，因此改用链表进行存储。
- 在尝试过程中使用过C++的STL中的List模板与pair数据结构结合存储学生信息，但由于List的迭代器无法进行直接存取，在迭代器进行遍历时利用List::erase()进行删除时会造成迭代器指针变化，故因remove函数实现困难而终止尝试。
- 最后自己实现了链表数据结构，并在题目基础上进行了相应的改动，实现了相应的函数功能。但在进行实际测试时，测试点1，3通过但测试点2内存出现问题，测试点3错误，其余测试点超过时间限

制。在重新仔细阅读题目后发现学生的学号姓名必须用字符串进行表示，而我之前的尝试均位整形变量来接受学生学号，导致了测试点发生错误。
- 在修正存储数据类型后，仍有一测试点，没有通过，经过试错发现是插入的位置范围问题，原来的插入范围在[1, length]，而在length + 1的位置是可以插入的，修正后通过了全部测试点。

2.2.6 总结和体会

- 在开始写代码前，需要先仔细分析题目，对题目需要的数据结构进行判断，在线性表中需要分析是否需要进行多次插入删除操作，如果需要，构造链表进行操作更佳。
- 需要仔细阅读题目，根据题目要求进行数据类型的选择。
- 将具体功能进行封装成函数利于后续bug检修。

2.3 一元多项式的相加和相乘

2.3.1 问题描述、

给出两个一元多项式（系数和指数），给出其相加或相乘后的一元多项式的形式

2.3.2 基本要求

- 输入：给出两个多项式的长度和系数 指数的顺序排列
- 输出：输出多项式，要求按照指数大小进行排列
  - 最后执行操作0：输出加法结果
  - 执行操作1：输出乘法结果
  - 执行操作2：输出两个结果
- 数据限制在2050以内， 结果为0 0不输出

2.3.3 数据结构设计

- 利用链表进行多项式表达，通过双指针在两个链表的遍历来进行数据的合并
- 利用内置数据类型pair来实现相应项的表达

```
pair<int, int>  res;
```

- 等价的也可以用结构体对存储的每一项进行表达

```
struct element
{
    int coeff;  //系数
    int exp;  // 指数
    element* next;  //指针域
};
```

- 在存储此元素的链表基础上进行操作，实现相应功能

2.3.4 功能说明

- 多项式加法函数

```
LinkedList pluser(LinkedList a1, LinkedList a2)
{
    /*
    *@para a1 第一个多项式链表
    *@para a2 第二个多项式链表
    */
    /*
    实现：给出两个指针，用来遍历两个多项式链表，给出空结果链表的头指针
    比较两指针对应多项式项的指数部分
    如果a1对应项指数部分比较大，则直接将其链接到结果链表中，同时a1链表对应指针向后移动一位，指向多项式的下一项
    如果a2对应项指数部分比较大，将a2对应项直接链接到结果链表中，同时a2链表对应指针向后移动一位，指向多项式的下一项
    如果二者相等，则将二者的系数相加，指数不变，链接到结果数组中，同时两个链表对应的指针同时移动到下一位
    当其中一个链表的指针NULL时，即将一个多项式遍历完成，此时需要将另一个链表（如果另一个链表对应的指针还不是NULL时）的剩余部分直接连接到答案链表中。
    注意此时的答案链表中的元素中系数可能为0，同时指数并不是按照从小到大进行排序的


    输出：返回答案链表的头指针
    */
}
```

- 为了使多项式按照指数进行排序，此处需要引入仿函数（或者二元谓词）
  - 仿函数

```
bool cmp(element p1, element p2)
{
    return p1.exp < p2.exp;
}
```

  - 二元谓词

```
class Comparison()
{
public:
    bool operator()(element p1, element p2)
    {
        return p1.exp < p2.exp;
    }
}
```

- 对应主函数调用

  l利用sort进行排序（可以通过将数据用pair进行存储来实现根据指数进行排序）

- 对应主函数输出

```
if (res_plus->next->coeff == 0 && res_plus->next->exp == 0) return 0;  // 0 0 的情况不输出
for (LinkedList ptr = res_plus; ptr != NULL; ptr = ptr->next)  //遍历链表
{
    if (ptr->coeff)  //注意系数为0不输出
        cout << ptr->coeff << " " << ptr->exp << " ";
}
```

- 多项式乘法函数

```
LinkedList multi(LinkedList a1, LinkedList a2)
{
     /*
    *@para a1 第一个多项式链表
    *@para a2 第二个多项式链表
    */
    /*
    实现：利用乘法分配律，将第一个多项式中的各个元素与第二个多项式相乘，得到多个多项式，利用多项式加法函数将其相加
    给出答案链表，
    遍历第一个链表，对于第一个链表中的每一项，
    给出临时多项式，将这一项分别与第二个多项式的每一项相乘，结果链接到临时多项式中，
    将答案链表与临时多项式相加，结果赋值给答案链表

    注意此时的答案链表中的元素中系数可能为0，同时指数并不是按照从小到大进行排序的

    输出：返回答案链表的头指针
    */
}
```

- 对应主函数调用

  利用sort进行排序（可以通过将数据用pair进行存储来实现根据指数进行排序）

- 对应主函数输出

```
if (res_mul->next->coeff == 0 && res_mul->next->exp == 0) return 0;  // 0 0 的情况不输出
for (LinkedList ptr = res_mul; ptr != NULL; ptr = ptr->next)  //遍历链表
{
    if (ptr->coeff)  //注意系数为0不输出
        cout << ptr->coeff << " " << ptr->exp << " ";
}
```

2.3.5 调试分析

- 需要注意加法函数双指针的应用，以及指针的移动时机，尤其要注意在二者指数相等时，加完后要将二者指针同时后移一位。
- 在第一次尝试中，输出时没有按照指数大小进行排序，后利用sort函数进行了相应的排序
- 后尝试中，没有注意到操作2是同时输出两个结果，导致一半的测试点没有通过，后完善了输出形式，通过了全部测试点

2.3.6 总结和体会

- 利用双指针进行多项式的相加是有效降低时间复杂度的一种方法，通常情况下，双指针可以将遍历的时间复杂度从On^2降低到On的级别。
- 此题的同一多项式输入项，指数可能有不重复的限制，若有重复限制，对于加法和乘法可能双指针无法给出正确结果（在双指针前进的过程中，重复指数的项并不会与之前合并，而是会直接链接到答案链表的后边，造成答案链表中指数的重复）
- 在进行乘法运算的时候用到了加法运算，我们可以通过数学运算将其简化为求解过的函数，从而简化运算。

2.4 求级数

2.4.1 问题描述

求出给定形式的级数

给出形如：

$$\sum_{i=1}^{N} iA^i = A + 2A^2 + \cdots + NA^N$$

的级数，求其值

2.4.2 基本要求

给定一个N，求出对应此级数的值

其中N小于等于150，A小于等于15

2.4.3 数据结构设计

注意到体重给出了两个数据范围，其中底数较小，而指数很大

估计：当A=2时，其数值范围已经超过了64位long long int的承载范围，甚至超过了 128位整数表示范围。

所以次题必须用高精度进行相应的乘法和加法运算，为保证乘方运算效率，使用快速幂进行乘方操作

2.4.4 功能说明

- 高精度乘法

  对人工运算乘法进行模拟，将每一位逆序记录在数组中，分别将两个数的每一位进行乘法运算，同时进行进位运算，将结果输出到结果数组的每一位中，最后将结果数组进行逆序输出即可

```
int* mul(int* A, int sizeA, int* B, int sizeB)
{
    /*
    *@para A 数组A
    *@para B 数组B
    *@para sizeA 数组A长度
    *@para sizeB 数组B长度
    */
    int C[N];  //初始化结果数组
```

```
    for (int i = 1; i <= N; i++)  //初始化为0
    {
        C[i] = 0;
    }

    for (int i = 0; i < sizeA; i++)  //双重循环对两个数的每一位进行遍历
    {
        for (int j = 0; j < sizeB; j++)
        {
            C[i + j] += A[i] * B[j]; //每一位的数相乘
            C[i + j + 1] += (C[i + j] / 10);  //下一位进位
            C[i + j] %= 10;  //原来这一位取个位数字
        }
    }

    return C;
}
```

- 高精度加法

  类似高精度乘法，通过模拟人工加法来计算结果中的每一位

```
int* add(int* A, int sizeA, int* B, int sizeB, int& places)
{
   /*
    *@para A 数组A
    *@para B 数组B
    *@para sizeA 数组A长度
    *@para sizeB 数组B长度
    *@para places 输出数组的长度
   */
    int C[N];  //初始化结果数组
    for (int i = 1; i <= N; i++)  //初始化为0
    {
        C[i] = 0;
    }

    int t = 0;  //记录进位
    for (int i = 0; i < sizeA || i < sizeB; i++)
    {
        if (i < sizeA) t += A[i];
        if (i < sizeB) t += B[i];
        C[i] = t % 10; //原来一位取个位数
        t /= 10; //进位取十位数（最多为1）
        places = i;
    }

    if (t) C[places + 1] = 1;  //如果进位有剩，最前面那一位补1

    while (places > 0 && C[places] == 0) places--; //去除前导0
    return C;
}
```

- 乘法快速幂

  普通求幂的方式是将底数求n遍，这样的复杂度是On，也就是

  $$a^n = \underbrace{a \cdot a \cdot a \cdots a}_{n个}$$

  也就是每次将幂次分为i - 1和1

  通过合理划分，我们也可以将其划分为当前指数的一半，也就是

  $$a^n = a^{\frac{n}{2}} \cdot a^{\frac{n}{2}} \quad a为偶数$$
  $$a^n = a^{\frac{n}{2}} \cdot a^{\frac{n}{2}+1} \quad a为奇数$$

  可以看到其中a^(n / 2)的部分是重复的，所以可以每次将a^(n / 2)存储起来，同时奇数与偶数的差异可以通过二进制的方式进行统一，通过判断指数每一位的二进制位是否是1来判断这一位是否进行乘方（类似于除以2）这样乘方的时间复杂度可以降低到O(logn)

  ```
  int* fps(int* A, int p)
  {
      /*
      *@para A 底数数组
      *@para p 指数
      */
      int ans[N];
      ans[0] = 1;

      while (p)
      {
          if (p & 1) ans = mul(ans, A);  //这一位有值，乘方结果乘以上一步结果
          A = mul(A, A);  //每一步自乘
          p >>= 1;  //从小到大依次检测指数的每一位
      }
      return ans;
  }
  ```

- 主函数调用对结果去除前导0

2.4.5 调试分析

- 其实对于这道题，没有必要用快速幂来进行乘方运算，遍历进行乘法运算就可以，但是快速幂可以进一步降低时间复杂度
- 高精度加法与乘法一定要注意最后前导0的剔除，同时要注意再删0过程中注意位数问题，当结果为0时，要保留一个0

2.4.6 总结和体会

本题很明显数量级必须要用高精度，其中高精度的运算还涉及到减法和除法，其中数据的存储都是逆序存储，为了保证在进位过程中往后可以顺利补位，如果顺位记录的话顺序表在前边添加进位的话需要往后挪位，产生时间浪费

2.5 扑克牌游戏

2.5.1 问题描述

- 扑克牌具有：花色、编号，其中编号从小到大为A,2,3,4,5,6,7,8,9,10,J,Q,K
- 实现对扑克牌堆进行一些操作

```
Append 添加一张扑克牌到牌堆的底部
Extract 从牌堆中抽取某种花色的所有牌，按照编号从小到大进行排序，并放到牌堆的顶部。
Revert 使整个牌堆逆序
Pop 如果牌堆非空，则除去牌堆顶部的第一张牌，并打印该牌的花色和数字；如果牌堆为空，则打印NULL
```

2.5.2 基本要求

- 初始时牌堆为空。输入n个操作命令（1 ≤ n ≤200），执行对应指令。所有指令执行完毕后打印牌堆中所有牌花色和数字（从牌堆顶到牌堆底），如果牌堆为空，则打印NULL
- 注意：每种花色和编号的牌数量不限
- 输入要求：命令数量 之后每一行输入一个命令
- 输出要求：输出若干行，每次收到Pop指令后输出一行（花色和数子或NULL），最后将牌堆中的牌从牌堆顶到牌堆底逐一输出（花色和数字），若牌堆为空则输出NULL

2.5.3 数据结构设计

通过双向链表来实现对于链表的前后端同时进行插入删除的操作，对于首尾指针的操作可以实现对于整个链表进行反转的操作。

- 记录牌的大小

```
int colorList[] = { 65 , 2, 3, 4, 5, 6, 7, 8, 9, 10, 74, 81, 75 }; //其中通过
```

- 牌类

```
class Poker
{
public:
    Poker(string color, int mnum)
    {
        this->color = mcolor;
        this->num = mnum;
    }

    string color;
    int num;
};
```

Poker类中记录牌的花色、点数，同时重载了类的构造函数

- 牌堆

通过建立双向链表表示牌堆，在尾部设立伪头指针，根据条件伪头指针成为头指针，往前进行遍历

FakeHead作为头节点时，prior就是之前的next

```
struct Node
{
    Node* next;
    Node* prior;
}Headhead, FakeHead;
Head->next = FakeHead->prior;
FakeHead->next = NULL;
```

2.5.4 功能说明

- append函数

```
void append(string color, int num)
{
    /*
    *@param color 牌色
    *@para num 点数
    */
    /*牌堆的Head和FakeHead之前都可以作为“牌堆底”*/
    if (flag == "reversed")
    {
        /*
        如果是已经反转的牌堆，要从尾部FakeHead进行遍历
        遍历到Head位置“前驱节点”（也就是原来列表的）头节点的后继节点
        进行插入
        */
    }
    else
    {
        /*
        遍历到Fakehead的前驱节点进行插入
        */
    }

}
```

- pop函数

```
void pop()
{
    if (flag == "reversed")
    {
        /*反转的牌堆对应的FakeHead的“后继节点就是排队需要pop的牌”*/
        /*通过指针域的操作把这个节点删除并把信息打印出来*/
    }
    else
    {
       /*把Head的后继节点删除并打印信息*/
    }
}
```

- revert函数

```
void revert(string )
{
    if (flag == "reversed")
    {
        flag = "direct";
    }
    else
    {
        flag = "reversed";
    }

}
```

- extract函数

```
void extarct(string color)
{
    /*
    *@param color 花色
    */
    /*
    预设一个结果链表（双向链表）
    指针从当前头部(Head或者FakeHead)开始遍历，当遍历到与输入花色相同的花色时，
    指针回退一位，
    将当前节点链接到结果链表中，同时在原牌堆中将此节点删除，（指针域操作）
    继续遍历，直到遍历到当前方向的尾部


    将结果列表根据点数进行排列
    （冒泡排序）从第一个节点开始，比较此节点与下一节点的大小关系，如果后边节点对应点数更大，就将二者位置互换（指针域进行操作）
    （或者用一个顺序表接挑选出来的节点，用sort进行排序）


    将排序好的链表尾部的节点（FakeHead之前的节点）链接到相应的头部后继节点，完成操作
    */
}
```

- 输入点数格式化存储函数

```
int getAscii(string s)
{
    if (s == "J") return 74;
    if (s == "Q") return 81;
    if (s == "K") return 75;
    if (s == "A") return 65;

    if (s == "10") return 10;
    if (s == "2") return 2;
    if (s == "3") return 3;
    if (s == "4") return 4;
    if (s == "5") return 5;
    if (s == "6") return 6;
    if (s == "7") return 7;
```

```
    if (s == "8") return 8;
    if (s == "9") return 9;
}
```

为了输入字母与数字花色的统一管理，需要进行这样的格式化

2.5.5 调试分析

- 充分利用双向链表可以对头尾进行的增删特性来实现牌堆的模拟
- 在实际操作过程中，添加了Head和FakeHead两个头节点，让反转和遍历更加方便
- 在存储字符时，最初的想法是用字符串数组进行存储，但是对于10来说char表示不了，所以后来字母用ascii进行存储，数字直接用相应整形进行存储。不使用map映射的话，也可以用二维数组进行字符与数字之间的对应。
- 若考虑STL的话：首先可以将其点数与相应的面值用map映射对应起来，用deque进行牌堆的模拟（双端队列可以进行头尾操作），revert以及extract的操作可以用双端栈进行实现逆序排列

2.5.6 总结和体会

- 注意如何进行数据的存储对应
- 当链表与首尾等操作结合时，应当用双向链表

3. 实验总结

本次实验主要是侧重于线性表的应用，通过使用两种线性表，通过使用两种类型的线性表解决一些实际问题，在实际应用过程中，我们应当时刻注意问题中经常进行的操作，如果经常进行的是删改操作，最好使用链表；如果是一次性导入信息后经常性查找，使用顺序表进行查找更佳。同时在链表构造过程中要注意指针的指向问题，尤其是双向链表的指针指向问题。在进行删除操作的同时要注意内存泄漏问题，及时清理内存。

线性表的不同存储方式使用具体取决于具体问题的分析，在实际选择线性表数据结构时，要根据情况选择不同的存储方式。同时对于线性表的操作如首尾操作也影响着具体的存储方式构建，如单链表与双向链表、循环链表的选择。