**张宇杰 2022113573**

1. 顺序表的相关算法

删除给定元素

//删除线性表中所有给定元素

template <typename Elem, typename Position>

void AbstractLinearList<Elem, Position>::remove\_elem(const Elem& obj)

{

    Position p = begin();

    while (p != end())

    {

        if (at(p) == obj)

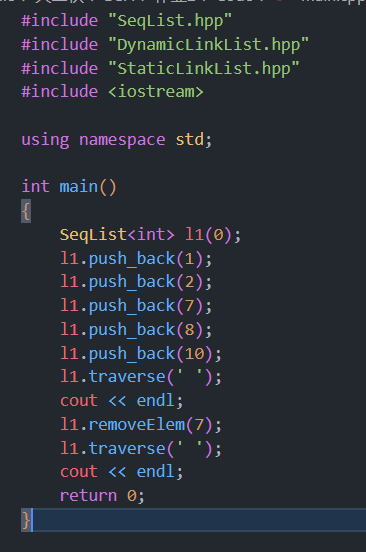
            remove(p);

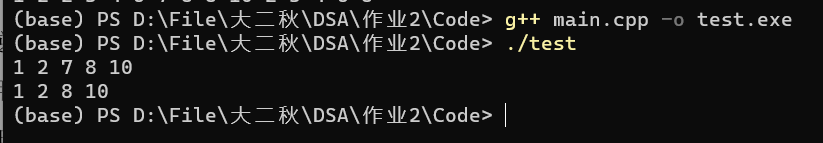
        else

            p = next(p);

    }

}





删除重复元素

//对排序好的线性表，删除所有重复元素

template <typename Elem, typename Position>

void AbstractLinearList<Elem, Position>::remove\_duplicate()

{

    Position p = begin();

    while ((p != end()) && (next(p) != end()))

    {

        //比较当前元素是否与下个元素相等

        if (at(p) == at(next(p)))

            //若相等，删除当前元素

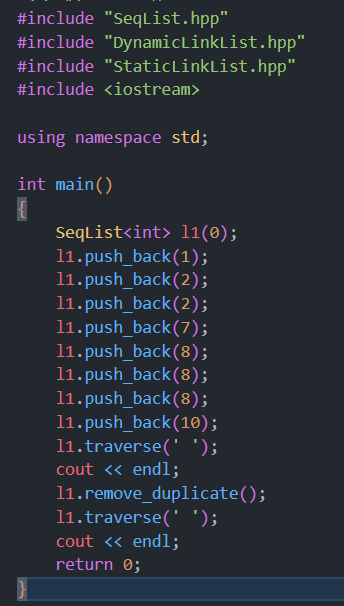
            remove(p);

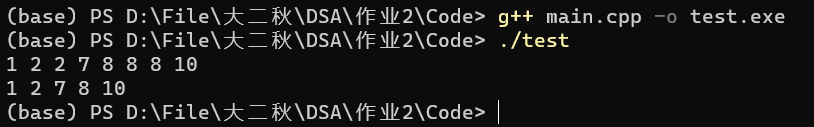
        else

            p = next(p);

    }

}





就地逆置

//顺序表就地倒置

template <typename Elem>

void SeqList<Elem>::reverse()

{

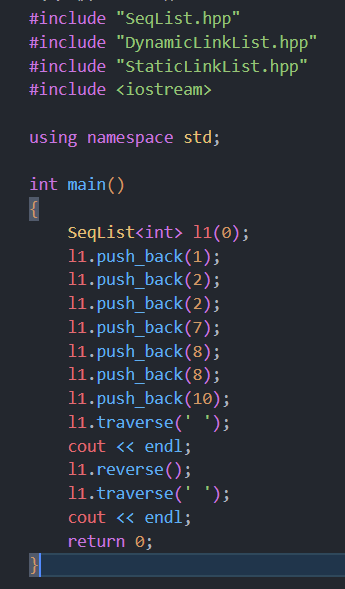
    if (!empty())

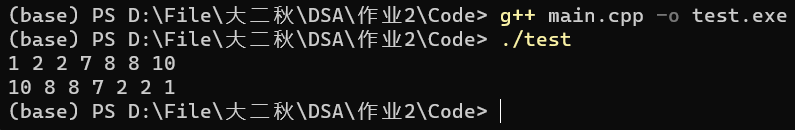
        //双指针交换

        for (int i = 0, j = used\_len - 1; i < j; i++, j--)

            std::swap(at(i), at(j));

}





循环移动k位

//顺序表循环移动k位，向左为正

template <typename Elem>

void SeqList<Elem>::cylic\_move(int k)

{

    if (!empty())

    {

        k = ((k % used\_len) + used\_len) % used\_len;

        // 求 k 对 len 在数学上的余数

        // e.g. 5 % 2 = 1, 7 % 4 = 3, -7 % 4 = 1

        //循环左移k位

        while (k > 0)

        {

            //将表头元素复制, 追加至表尾

            this->push\_back(at(0));

            //删除表头元素

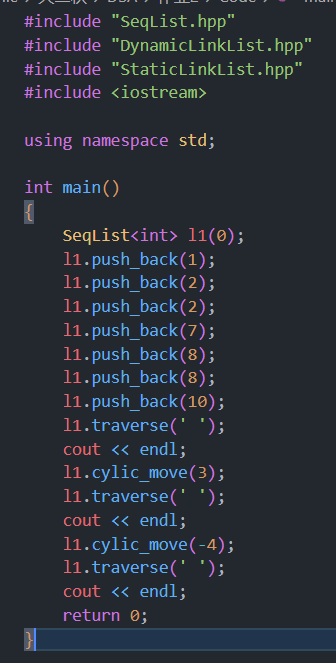
            remove(0);

            k--;

        }

    }

}



文本

描述已自动生成

合并排序好的两顺序表

//对排序好的两个顺序表进行合并

template <typename Elem>

void SeqList<Elem>::merge(const SeqList<Elem>& list2, bool (\*cmp)(const Elem&, const Elem&))

{

    //对本对象扩容，使之能够装下list2

    try\_extend(list2.used\_len);

    //给本对象起别名list1

    SeqList<Elem>& list1 = \*this;

    //将两个顺序表的元素根据cmp函数倒序复制到list1中

    int i = used\_len - 1, j = list2.used\_len - 1, k = used\_len + list2.used\_len - 1;

    while (i >= 0 && j >= 0)

    {

        if (cmp(list1[i], list2[j]))

            list1[k--] = list2[j--];

        else

            list1[k--] = list1[i--];

    }

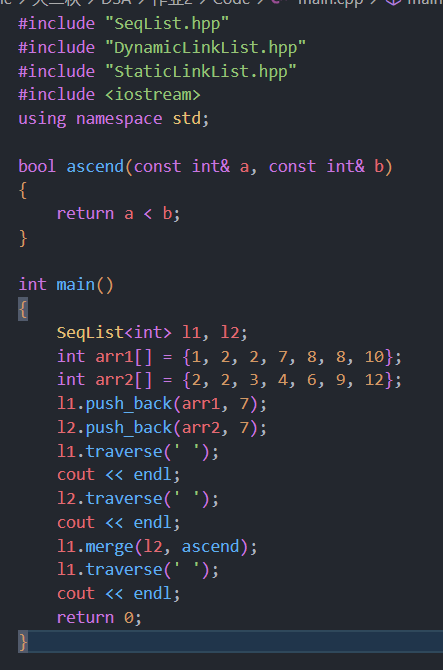
    //将list2剩下的元素倒序复制至list1

    while (j >= 0)

        list1[k--] = list2[j--];

    used\_len += list2.used\_len;

}



文本

描述已自动生成

二、动态链表相关算法

删除给定元素

（代码复用AbstractLinearList类中的remove\_elem方法）

文本

描述已自动生成

文本

描述已自动生成

删除重复元素

（代码复用AbstractLinearList类中的remove\_duplicate方法）

文本

描述已自动生成

文本

描述已自动生成

链表就地逆置

//链表就地倒置

template <typename Elem, typename Position>

void AbstractLinkList<Elem, Position>::reverse()

{

    //p指向第一个节点

    Position p = \_next(begin());

    //当p不为空（表存在）

    if (p != null())

    {

        //p\_prior为 p指向的前一个元素 的指针

        Position p\_prior = p;

        p = \_next(p);

        //第一个节点指向空

        \_next(p\_prior) = null();

        while (p != null())

        {

            //p\_next为 p指向的后一个元素 的指针

            Position p\_next = \_next(p);

            //让p指向的元素 指向前一个元素

            \_next(p) = p\_prior;

            //p\_prior 与 p 的指向 均 向后移动一个节点

            p\_prior = p;

            p = p\_next;

        }

        //当p为空时，p\_prior正好指向表尾

        //让表头指向表尾

        \_next(begin()) = p\_prior;

    }

}文本

描述已自动生成

文本

描述已自动生成

循环移动k位

//链表表循环移动k位，向左为正

template <typename Elem, typename Position>

void AbstractLinkList<Elem, Position>::cylic\_move(int k)

{

    int len = length();

    if (k % len != 0)

    {

        \_next(end()) = \_next(begin());

        // 尾节点指向第一个节点

        k = ((k % len) + len) % len;

        // 求 k 对 len 在数学上的余数

        // e.g. 5 % 2 = 1, 7 % 4 = 3, -7 % 4 = 1

        Position new\_head = begin();

        for (int i = 0; i < k; i++)

        {

            new\_head = \_next(new\_head);

        }

        // 此时new\_head指向 新的第一个节点 的 前一个节点

        \_next(begin()) = \_next(new\_head);

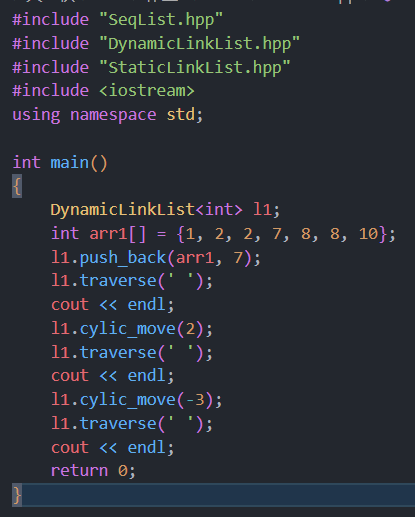
        // 头节点指向新的第一个节点

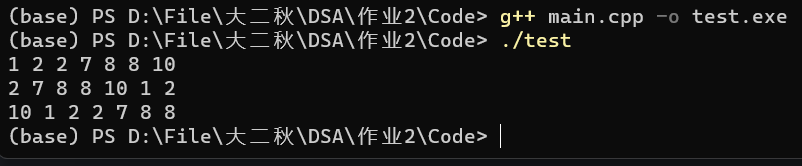
        \_next(new\_head) = null();

        // 新的第一个节点的前一个节点（即新的尾节点）指向空

    }

}





合并排序好的链表

//对排序好的两个链表进行合并

template <typename Elem, typename Position>

void AbstractLinkList<Elem, Position>::merge(const AbstractLinkList<Elem, Position>& list, bool (\*cmp)(const Elem&, const Elem&))

{

    Position p = \_next(begin());

    Position q = \_next(list.begin());

    //从本对象的表头开始，依次将两个链表按cmp顺序连在一起

    Position temp = begin();

    while (p != null() && q != null())

    {

        //谁更cmp，temp就与谁连接

        //被连接的一方应后移一个节点

        if (cmp(data(p), data(q)))

        {

            \_next(temp) = p;

            p = \_next(p);

        }

        else

        {

            \_next(temp) = q;

            q = \_next(q);

        }

        //temp向后移动，始终指向新链表的表尾

        temp = \_next(temp);

    }

    //当一方链表为空时，temp连接另一方剩下的链表

    if (p == null())

        \_next(temp) = q;

    else

        \_next(temp) = p;

    //在链表的合并操作后，第二个表已失去意义，应置空

    \_next(list.begin()) = null();

}

文本

描述已自动生成

文本

描述已自动生成

三、静态链表相关算法

就地逆置

（代码复用AbstractLinkList类中的reverse方法）

文本

描述已自动生成

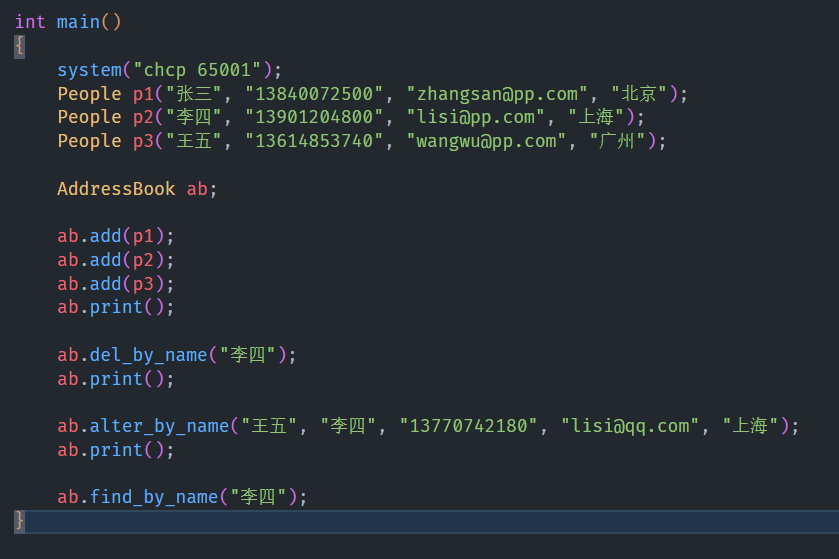
图片包含 文本

描述已自动生成

四、通讯录管理系统

（代码见源码）

目前未实现与用户交互的界面，但已经提供了相关接口



增

文本

描述已自动生成

删



改



查

