顺序表常用的功能

```
// 基本增删查改接口
// 顺序表初始化
void SeqListInit(SeqList* psl);
// 顺序表销毁
void SeqListDestory(SeqList* psl);
// 顺序表打印
void SeqListPrint(SeqList* psl);
// 检查空间，如果满了，进行增容
void CheckCapacity(SeqList* psl);
// 顺序表尾插
void SeqListPushBack(SeqList* psl, SLDataType x);
// 顺序表尾删
void SeqListPopBack(SeqList* psl);
// 顺序表头插
void SeqListPushFront(SeqList* psl, SLDataType x);
// 顺序表头删
void SeqListPopFront(SeqList* psl);
// 顺序表查找
int SeqListFind(SeqList* psl, SLDataType x);
// 顺序表在pos位置插入x
void SeqListInsert(SeqList* psl, size_t pos, SLDataType x);
// 顺序表删除pos位置的值
void SeqListErase(SeqList* psl, size_t pos);
```

动态顺序表结构定义

让数组可以存任意类型的数据 -- 类型重定义

```
typedef int SLDataType;//类型重定义，int float double...
typedef struct Seqlist
{
    SLDataType* a;//数组
    int size;//表示数组中存储了多少个数据
    int capacity;//数组实际能存数据的空间容量是多大
}SL;
```

新建顺序表

```
void TestSeqList1()
{
    SL s1;
    SeqListPrint(&s1);
}
```

初始化顺序表

```
void SeaListInit(SL* ps)//初始化
{
    ps->a = NULL;
    ps->size = 0;
    ps->capacity = 0;
}
```

扩容接口

判断容量是否为满并扩容，由于头插和尾插都涉及这步操作，为了减少代码冗余，封装一个扩容函数

```
//扩容
void SeqlistCheckCapacity(SL* ps)
{
    if (ps->size == ps->capacity)
    {
        //如果是0，就给四个空间，如果capacity满了就扩大n倍，随便自己扩大
        int newcapacity = ps->capacity == 0 ? 4 : ps->capacity * 2;
        SLDataType* tmp = (SLDataType*)realloc(ps->a, newcapacity *
sizeof(SLDataType));
        {
            //如果空间开辟失败
            if (tmp == NULL)
            {
                printf("realloc fail\n");
                exit(-1);
            }
            //空间开辟成功
            ps->a = tmp;
            ps->capacity = newcapacity;
```

```
        }
    }
}
```

尾插法

1.空间足够，直接插入数据即可

```
void SeqListPushBack(SL* ps, SLDataType x)//尾插
{
    //空间足够的处理

    ps->a[ps->size] = x;
    ps->size++;
}
```

2.整个顺序表没有空间

3.空间不够扩容

```
void SeqListPushBack(SL* ps, SLDataType x)//尾插
{
    SeqlistCheckCapacity(ps);

    //空间足够的处理
    ps->a[ps->size] = x;
    ps->size++;
}
```

头插法

1.空间足够，直接插入数据即可

```
void SeqListPushFront(SL* ps, SLDataType x)//头插
{
    //把所有元素往后平移一个，把要插入的元素放到前面
    int end = ps->size - 1;
    while (end >= 0)
    {
        ps->a[end + 1] = ps->a[end];
        end--;
```

```
    }
    ps->a[0] = x;
    ps->size++;
}
```

2.整个顺序表没有空间

3.空间不够扩容

```
void SeqListPushFront(SL* ps, SLDataType x)//头插
{
    SeqlistCheckCapacity(ps);
    //把所有元素往后平移一个，把要插入的元素放到前面
    int end = ps->size - 1;
    while (end >= 0)
    {
        ps->a[end + 1] = ps->a[end];
        end--;
    }
    ps->a[0] = x;
    ps->size++;
}
```

尾删法

1.删的数据小于size

2.删的数据大于size就会越界访问

两种解决方案：

第一种：if语句判断是否越界 -- 温柔

```
void SeqListPopBack(SL* ps)//尾删
{
    if (ps->size > 0)
    {
        //ps->a[ps->size - 1] = 0;//最后一个数据置0,这句可以不要，因为就算置0，开辟的内存还在
        ps->size--;
    }
}
```

第二种：assert断言，如果越界程序终止 -- 粗暴

```
#include <assert.h>
void SeqListPopBack(SL* ps)//尾删
{
    assert(ps->size > 0);
    ps->size--;
}
```

头删法

越界的处理方式和尾删完全一致

```
#include <assert.h>
void SeqListPopFront(SL* ps)//头删
{
    assert(ps->size > 0);
    //挪动数据
    int start = 1;
    while (start<ps->size)
    {
        start++;
    }
}

```

顺序表查找

```
int SeqListFind(SL* ps, SLDataType x)//找到了返回x位置下标，没有找到返回-1
{
    //多次出现x的情况也可以解决
    for (int i = 0; i < ps->size; i++)
    {
        if (ps->a[i] == x)
        {
            return i;
        }
    }
        return -1;
```

```
}
```

任意位置插入，并改造头插和尾插

```
void SeqListInsert(SL* ps, int pos, SLDataType x)//在指定下标位置插入
{
    //温柔处理方式
    if (pos > ps->size || pos < 0)
    {
        printf("pos invalid\n");
        return;
    }
    //暴力处理方式
    assert(pos <= ps->size && pos >= 0);
    //插入先判断是否要增容
    SeqlistCheckCapacity(ps);
    int end = ps->size - 1;
    while (end >= pos)
    {
        ps->a[end + 1] = ps->a[end];
        end--;
    }
    ps->a[pos] = x;
    ps->size++;
}
```

头插尾插改造

```
void SeqListPushFront2(SL* ps, SLDataType x)//头插
{
    SeqListInsert(ps, 0, x);
}

void SeqListPushBack2(SL* ps, SLDataType x)//尾插
{
    SeqListInsert(ps, ps->size, x);
}
```

任意位置删除，并改造头删和尾删

```
void SeqListErase(SL* ps, int pos)//删除任意位置元素
{
    assert(pos >= 0 && pos < ps->size);
    int begin = pos + 1;
    while (begin < ps->size)
    {
        ps->a[begin - 1] = ps->a[begin];
        begin++;
    }
    ps->size--;
}
```

头删尾删改造

```
//头删尾删改造
void SeqListPopFront2(SL* ps)//头删
{
    SeqListErase(ps, 0);
}

void SeqListPopBack2(SL* ps)//尾删
{
    SeqListErase(ps, ps->size-1);
}
```

打印顺序表

```
void SeqListPrint(SL* ps)
{
    for (int i = 0; i < ps->size; i++)
    {
        printf("%d ", ps->a[i]);
    }
    printf("\n");
}
```

销毁空间

```c
void SeqListDestory(SL* ps)
{
    free(ps->a);
    ps->a = NULL;
    ps->capacity = ps->size = 0;
}
```

主函数

```c
void TestSeqList1()
{
    SL s1;
    SeaListInit(&s1);
    SeqListPushBack(&s1, 1);
    SeqListPushBack(&s1, 2);
    SeqListPushBack(&s1, 3);
    SeqListPushBack(&s1, 4);
    SeqListPushBack(&s1, 5);

    SeqListPrint(&s1);
}

int main()
{
    TestSeqList1();
    return 0;
}
```

顺序表的缺陷：

1.空间不够要增容，增容是要付出代价

2.避免频繁扩容，基本都是扩2倍，可能就会导致一定空间的浪费

3.顺序表要求数据从开始位置连续存储那么我们在头部或者中间位置插入删除数据就需要挪动数据效率不高

针对线顺序表缺陷就设计出链表