

03 | 顺序表（下）：常用操作合集与复杂度分析

2023-02-17 王健伟 来自北京

《快速上手C++数据结构与算法》



你好，我是王健伟。

上节课，我们实现了向顺序表中插入元素的操作。

这节课，我们继续探讨顺序表的不同操作，和上节课一样，先从抽象模型开始理解，分析元素在不同操作下可能会发生的情况以及我们需要注意到的细节，再去理解操作的实现的代码。通过时间复杂度的分析，为我们提供优化操作的思路。

shikey.com 转载分享

我们先从顺序表中元素的删除操作开始说起。

顺序表元素删除操作

因为顺序表中每个数据元素在内存中是连续存储的，所以如果删除某个位置的元素，则需要依次把该位置后面的元素依次向前移动。如图 5 所示：

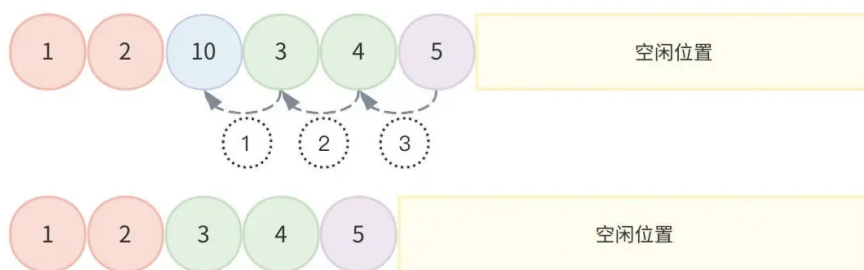


图5 顺序表删除元素10前后的元素位置对比图

在图 5 中，如果要将第 3 个位置的元素 10 删除，为了保证元素之间内存的连续性，需要将原来第 4 个位置以及第 4 个位置之后的所有元素依次向前移动 1 个位置，以保证元素之间的内存紧密相连。

那么这里就有几个需要考虑的问题了。

先从谁开始移动呢？

在移动 3、4、5 这几个元素时，需要先把元素 3 移动到第 3 个位置，再把元素 4 移动到第 4 个位置，最后把元素 5 移动到第 5 个位置，也就是先从数组中要删除元素位置的后面一个位置的元素开始依次向前移动，且不可先把元素 5 移动到第 5 个位置，因为这样会把本来在第 5 个位置的元素 4 直接覆盖掉。

另一方面，所要删除的位置必须有元素才可以删除。

理清头绪之后，我们看一下删除操作 ListDelete 的实现代码。

shikey.com转载分享

复制代码

```
1 //删除第i个位置的元素
2 template < typename T>
3 bool SeqList<T>::ListDelete(int i)
4 {
5     if (m_length < 1)
6     {
7         cout << "当前顺序表为空，不能删除任何数据!" << endl;
8         return false;
9     }
10    if (i < 1 || i > m_length)
```


```

11  {
12      cout << "删除的位置" << i << "不合法, 合法的位置是1到" << m_length << "之间!" << endl;
13      return false;
14  }
15  cout << "成功删除位置为" << i << "的元素, 该元素的值为" << m_data[i - 1] << "!" << endl;
16  //从数组中第i+1个位置开始向后遍历所有元素, 分别将这些位置中原有的元素向前移动一个位置
17  for (int j = i ; j < m_length; ++j)
18  {
19      m_data[j-1] = m_data[j];
20  }
21  m_length--;          //实际表长-1
22  return true;
23  }

```

在 main 主函数中, 继续增加代码测试元素删除操作。

```
1 seqobj.ListDelete(1);
```

 复制代码

新增代码的执行结果如下。

成功删除位置为1的元素, 该元素的值为15!

分析一下 ListDelete 的时间复杂度, 只需要关注 for 循环的执行次数与问题规模 n 的关系, 问题规模 n 在这里指的是顺序表的当前长度 m_length 。

最好情况时间复杂度

如果要删除顺序表尾部的元素, 则其他已有的顺序表中元素都不需要移动, for 循环一次都不会执行, 这是最好情况时间复杂度 $O(1)$ 。

最坏情况时间复杂度

如果删除顺序表头部的元素, 则其他已有的顺序表中所有元素 ($n-1$ 个) 都需要依次前移, for 循环执行的次数就是 $n-1$ 次, 这是最坏情况时间复杂度 $O(n)$ 。

平均情况时间复杂度

如果假设删除任何一个位置元素的概率相同，那么删除位置 1, 2,, m_length 的概率就都为 $\frac{1}{n}$ ，如果删除第一个位置的元素，则需要把后面 n-1 个元素依次前移，也就是 for 循环会执行 n-1 次，如果删除第二个位置的元素，则需要把后面 n-2 个元素依次前移，也就是 for 循环会执行 n-2 次.....以此类推，如果删除最后一个位置（尾部）的元素，则 for 循环会执行 0 次。

把每种情况下循环的次数累加起来，再除以 n，就得到了数组中元素前移次数的平均值（平均循环次数） $\frac{n-1}{2}$ 。又因为大 O 时间复杂度表示法中，系数、常量可以忽略掉，所以平均情况时间复杂度为 O(n)。

$$\text{前移次数平均值} = \frac{1 + 2 + 3 + \dots + (n - 1)}{n}$$

$$\text{根据等差数列求和公式 } 1+2+3+\dots+n = \frac{n(n+1)}{2},$$

如果把该公式中的 n 用 n-1 替换，

$$\text{则可以得到 } 1+2+3+\dots+(n-1) \text{ 等于 } \frac{(n-1)n}{2}$$

把新得到的这个公式带入到前移次数平均值中去，

$$\text{就有 前移次数平均值} = \left(\frac{(n-1)n}{2}\right) \frac{1}{n} = \frac{n-1}{2}。$$

极客时间

你会发现，时间开销主要源于元素的移动。

我们算了这么多的时间复杂度，有什么用呢？

shickey.com 转载分享


思考一下，如果是连续删除顺序表中几个紧挨着的元素，那么每删除其中一个元素就会做一次剩余元素的移动操作，效率显然比较低。

所以，我们可以将几个连续的元素全部删除后，一次性完成剩余元素的移动操作，以此提高程序的执行效率。

顺序表元素获取操作

关于元素获取操作，我们分为两种情况来讨论：按位置获取，以及按元素值获取。

首先看一下如何**按位置获取**顺序表中元素值，下面是具体代码。

 复制代码

```
1 //获得第i个位置的元素值
2 template<class T>
3 bool SeqList<T>::GetElem(int i, T& e) //参数e是引用类型参数，确保将该值带回调用者
4 {
5     if (m_length < 1)
6     {
7         cout << "当前顺序表为空，不能获取任何数据!" << endl;
8         return false;
9     }
10
11     if (i < 1 || i > m_length)
12     {
13         cout << "获取元素的位置" << i << "不合法，合法的位置是1到" << m_length << "之间!" << endl;
14         return false;
15     }
16     e = m_data[i-1];
17     cout << "成功获取位置为" << i << "的元素，该元素的值为" << m_data[i - 1] << "!" << endl;
18     return true;
19 }
```

在 main 主函数中，继续增加如下代码测试按位置进行元素获取操作。

 复制代码

```
1 int eval = 0;
2 seqobj.GetElem(1, eval); //如果GetElem()返回true，则eval中保存着获取到的元素值
```


shikey.com转载分享

新增代码的执行结果如下。

成功获取位置为1的元素，该元素的值为10!


显然，按位置获取顺序表元素操作的时间复杂度为 $O(1)$ 。

再来看看**按元素值查找**其在顺序表中第一次出现的位置的代码。

 复制代码

```
1 //按元素值查找其在顺序表中第一次出现的位置
2 template<class T>
3 int SeqList<T>::LocateElem(const T& e)
4 {
5     for (int i = 0; i < m_length; ++i)
6     {
7         if (m_data[i] == e)
8         {
9             cout << "值为" << e << "的元素在顺序表中第一次出现的位置为" << i+1 << "!" << endl;
10            return i + 1; //返回的位置应该用数组下标值+1
11        }
12    }
13    cout << "值为" << e << "的元素在顺序表中没有找到!" << endl;
14    return -1; //返回-1表示查找失败
15 }
```

在 main 主函数中，继续增加如下代码测试按元素值查找其在顺序表中第一次出现的位置。

 复制代码

```
1 int findvalue = 10; //在顺序表中要找的元素值
2 seqobj.LocateElem(findvalue);
```

新增代码的执行结果如下。

shikey.com转载分享

值为10的元素在顺序表中第一次出现的位置为1!

同样，我们分析一下 LocateElem 的时间复杂度。这里只需要关注 for 循环的执行次数与问题规模 n 的关系，问题规模 n 在这里指的是顺序表的当前长度 m_length 。

如果要查找的元素值正好位于顺序表头部，则 for 循环只需要执行一次，这是最好情况时间复杂度 $O(1)$ 。

如果要查找的元素值正好位于顺序表尾部，则 for 循环需要执行 n 次，这是最坏情况时间复杂度 $O(n)$ 。

如果假设要查找的元素出现在任何一个位置的概率相同，因为顺序表中有 n 个元素，也就是出现在任何一个位置的概率都为 $\frac{1}{n}$ ，如果要查找的元素值位于第一个位置，则 for 循环会执行一次，如果要查找的元素位于第二个位置，则 for 循环会执行 2 次...，以此类推，如果要查找的元素位于最后一个位置，则 for 循环会执行 n 次，把每种情况下循环的次数累加起来，再除以 n ，就得到了查找元素次数的平均值 $\frac{n+1}{2}$ 。因为大 O 时间复杂度表示法中，系数、常量可以忽略掉，所以平均情况时间复杂度为 $O(n)$ 。

$$\begin{aligned}\text{查找元素次数平均值} &= \frac{1+2+3+\dots+n}{n} \\ \text{根据等差数列求和公式 } 1+2+3+\dots+n &= \frac{n(n+1)}{2}, \\ \text{就有 前移次数平均值} &= \left(\frac{(n+1)n}{2}\right) \frac{1}{n} = \frac{n+1}{2}.\end{aligned}$$

 极客时间

顺序表元素的其他常用操作

目前为止，我们已经了解了顺序表基本框架的搭建，元素的插入、删除、获取操作，除此之外，顺序表还有其他一些常用操作，比如输出所有元素、获取顺序表长度、翻转顺序表等等。我们来看一下它们的具体实现。

1. 输出顺序表中的所有元素 Displist

```
1 //输出顺序表中的所有元素，时间复杂度为O(n)
2 template<class T>
3 void SeqList<T>::Displist()
4 {
5     for (int i = 0; i < m_length; ++i)
6     {
```

 复制代码

```
7     cout << m_data[i] << " "; //每个数据之间以空格分隔
8 }
9     cout << endl; //换行
10 }
```

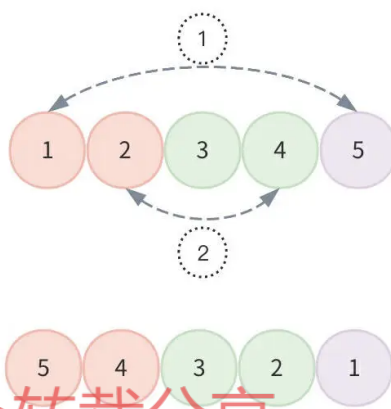
2. 获取顺序表的长度 ListLength

复制代码

```
1 //获取顺序表的长度，时间复杂度为O(1)
2 template<class T>
3 int SeqList<T>::ListLength()
4 {
5     return m_length;
6 }
```

3. 翻转顺序表 ReverseList

所谓翻转顺序表，就是把顺序表中元素的排列顺序反过来，比如原来存放的元素是 1、2、3、4、5，那么翻转后存放的元素就是 5、4、3、2、1。如图 6 所示：



极客时间

图6 顺序表中元素翻转前后对比图

解决这种问题并不难，在图 6 中，只需要将第 1 个元素和第 5 个（最后一个）元素交换位置，第 2 个元素跟第 4 个（倒数第二个）元素交换位置，第 3 个元素保持不动即可。

复制代码

```
1 //翻转顺序表，时间复杂度为O(n)
```



```

2  template<class T>
3  void SeqList<T>::ReverseList()
4  {
5      if (m_length <= 1)
6      {
7          //如果顺序表中没有元素或者只有一个元素，那么就不用做任何操作
8          return;
9      }
10     T temp;
11     for (int i = 0; i < m_length / 2; ++i)
12     {
13         temp = m_data[i];
14         m_data[i] = m_data[m_length - i - 1];
15         m_data[m_length - i - 1] = temp;
16     }
17 }

```


另外，我们还可以扩展出其他常用操作，相关代码你可以自行实现。

LocateElem 和 ListDelete 配合可以实现按值删除顺序表中指定元素。

如果 ListLength 返回 0 就可以确定顺序表为空或者专门实现一个判断顺序表是否为空的成员函数。

在顺序表头部或者尾部插入或者删除元素的成员函数。

在 main 主函数中，继续增加代码对上述已经实现的函数进行测试。

 复制代码

```

1  seqobj.ListInsert(2, 100);
2  seqobj.DispList();
3  cout << seqobj.ListLength() << endl;
4  seqobj.ReverseList();
5  seqobj.DispList();


```

顺序表的扩展操作

最后，我们说一下顺序表的扩展操作。

扩展是什么意思呢？比如在前面针对插入函数 ListInsert 的代码实现中，如果顺序表已经存满了数据，那就不允许再插入新数据了，这造成了一些使用中的不便，这个时候，我们当然希望顺序表能够自动扩容。

具体的实现思路，就是重新 new 一块比原顺序表所需内存更大一些的内存以便容纳更多的元素，然后把原来内存中的元素拷贝到新内存（这一步动作如果元素很多将很耗费时间）并把原内存释放掉（当然，这样做也是比较影响程序执行效率的）。为此，引入成员函数 IncreaseSize，代码我也放在了下面。

 复制代码

```
1 //当顺序表存满数据后可以调用此函数为顺序表扩容，时间复杂度为O(n)
2 template<class T>
3 void SeqList<T>::IncreaseSize()
4 {
5     T* p = m_data;
6     m_data = new T[m_maxsize + IncSize]; //重新为顺序表分配更大的内存空间
7     for (int i = 0; i < m_length; i++)
8     {
9         m_data[i] = p[i];                //将数据复制到新区域
10    }
11    m_maxsize = m_maxsize + IncSize;      //顺序表最大长度增加IncSize
12    delete[] p;                          //释放原来的内存空间
13 }
```


现在，就可以修改插入函数 ListInsert 的代码，以达到当顺序表满之后再插入数据时能够自动扩容的目的，你可以看一下原 ListInsert 代码的第一个 if 判断语句（判断顺序表是否已满）内容。

 复制代码

shikekey.com 转载分享


```
1 //如果顺序表已经存满了数据，则不允许再插入新数据了
2 if (m_length >= m_maxsize)
3 {
4     cout << "顺序表已满，不能再进行插入操作了!" << endl;
5     return false;
6 }
```

只需要对上述 if 判断语句进行简单修改，其他代码不变，下面是修改后的代码。

 复制代码

```
1 //如果顺序表已经存满了数据，则不允许再插入新数据了
2 if (m_length >= m_maxsize)
3 {
4     //cout << "顺序表已满，不能再进行插入操作了!" << endl;
5     //return false;
6     IncreaseSize();
7 }
```

在 main 主函数中，继续增加如下代码对上述函数进行测试就可以了。

 复制代码

```
1 for (int i = 3; i < 30; ++i)
2 {
3     seqobj.ListInsert(i, i*2);
4 }
5 seqobj.DispList();
```

小结

这节课我们继续通过代码详细实现了顺序表的元素删除、获取操作以及各种其他常用操作，包括输出顺序表中的所有元素、获取顺序表的长度、翻转顺序表。在顺序表扩展操作这个话题中，用代码实现了向顺序表插入数据时如果发现顺序表已满如何进行顺序表的自动扩容。

这里可以总结一下顺序表的几个重要的特点。

通过下标**随机访问**数组元素的时间复杂度仅为 **$O(1)$** 。

存储的数据**紧凑**，表中的元素在内存中紧密相连，无须为维持表中元素之间的前后关系而增加额外的存储空间（后面学习到链表时会看到增加额外存储空间来维持表中元素前后关系的情形）。

插入和删除操作可能会**移动大量元素**导致这两个动作**效率不高**。

需要大片**连续**的内存空间来存储数据。

动态分配内存的方式实现的顺序表在扩展顺序表容量时扩展的空间大小不好确定，分配太多内存容易造成浪费，分配太少内存就会导致 new 和 delete 被频繁调用，影响程序执行效率。而且扩展顺序表容量操作的时间复杂度也比较高。

正如我们所见，顺序表要求内存中数据连续存储，这既带来了定位元素的便利，同时也拖慢了插入和删除元素的速度。

值得说明的是，对于多数需要使用顺序表的场合，直接使用标准模板库中的 vector 容器即可，其丰富的接口会给开发带来更多的便利。

如果是对于性能要求极其严苛的底层开发，而且通过测试确定了自己编写的代码执行效率比 vector 容器更高，也可以利用这节讲解的知识自行实现顺序表。

课后思考

在 STL（标准模板库）中，提供了一个基于数组的容器 vector，你知道 vector 容器存储数据是什么样子的吗？vector 容器的优缺点和使用场合是什么呢？此外，vector 容器中的 reserve 方法、capacity 方法是用来做什么的呢？

欢迎你在留言区和我互动。如果觉得有所收获，也可以分享给更多的朋友一起学习。我们下一讲见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (3) shikey.com 转载分享



徐曙辉

2023-02-17 来自湖南

vector跟Java的ArrayList、Go的slice作用类似。

以go的slice举例，它是在静态数组基础上增加扩容机制后的动态数组，存储的数据在静态数组上。由3个部分组成，data 是指向数组的指针；len 是当前slice的长度；cap 是当前slice的容量。

优点是自动扩容机制让开发者不用手动管理内存，在业务开发中不确定数据数量的时候用slic

e。

缺点是如果存储的数据很多，要经常扩容，每次扩容需要 1.开辟更大内存空间，2.移动所有元素到新数组，3.释放旧数组空间内存。扩容对性能影响比较大，扩容次数的时间复杂度是 $O(\log n)$ ，所以我们在初始化的时候如果元素数量是确定的就要指定容量，避免扩容，优化性能。

vector 容器中的 reserve 方法设置容量大小，capacity 方法获取当前vector 容量

作者回复: 我懂你的意思，但作为传播知识的人，把知识简单讲是他的责任。我们用new扩容一下，总比我们给学友们介绍容器的size,capacity,reserve,是什么含义要简单直观得多，在我看来，很多时候讲的少就是讲的多。



👍 2



阿阳

2023-02-18 来自江苏

老师，在这节课中，定义模板函数前面使用template<class T>，而上一节课中使用的是template < typename T>。请问这两种方法有什么联系和区别吗？

作者回复: 没区别，在模板定义开头这个位置，class和typename可互换，当然，其他位置class 和typename一般是不互换的。



👍 1



ikel

2023-03-14 来自上海

vector 容器存储数据类似于数组，reserve 方法相当于new，capacity相当于m_length

作者回复: vector容器用起来简单，但误用也挺影响效率。reverse 理解为new，可以的。但capacity我记得返回的是new出来的大小。size才相当于m_length，capacity相当于m_maxsize。



shikey.com转载分享

