线性表

麻珂珂，蒋开慧，高浚哲，皇有为

XUPT

2019 年 9 月 16 日

线性表基本介绍

线性表的定义：

定义

一个线性表是 n 个具有相同特性的数据元素的有限序列。数据元素是一个抽象的符号，其具体含义在不同的情况下一般不同
线性表的相邻元素之间存在着序偶关系。如用
$(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ 表示一个顺序表，则表中 a_{i-1} 领先于 a_i，a_i 领先于 a_{i+1}，称 a_{i-1} 是 a_i 的直接前驱元素，a_{i+1} 是 a_i 的直接后继元素。当 $i = 1, 2, \ldots, n-1$ 时，a_i 有且仅有一个直接后继，当
$i = 2, 3, \ldots, n$ 时，a_i 有且仅有一个直接前驱

线性表的实现主要有两种方式：

- **顺序表** 所有的元素都是相邻的
- **链表** 元素与元素之间不一定相邻

顺序表的定义及接口

定义

```
typedef struct List{
    ElemType * lst;
    int len;
    int maxlen;
} List;
```

顺序表的构造和销毁

顺序表的构造

构造

```
List * initList(int max_length){
    List * ans = (List *)malloc(sizeof(List));
    if(ans == NULL) return NULL;
    ans -> lst = (ElemType *)malloc(sizeof(ElemType) * max_len
    if(ans -> lst == NULL) return NULL;
    ans -> len = 0;
    ans -> maxlen = max_length;
    return ans;
}
```

构造一个顺序表的所有操作时间都是已知的，故构造顺序表的时间复杂度是 $\mathcal{O}(1)$。

顺序表的销毁

销毁顺序表的时候，释放顺序表所占的内存。

顺序表的随机访问

随机访问

```
template<typename T>
T & List<T>::operator[](int idx){
    assert(0 <= idx && idx < len);  // 检查idx处是否有元素
    return lst[idx];   // 返回顺序表idx处的元素
}
```

顺序表是一种限制元素的地址是连续的的线性表。由于这个限制，我们可以通过一次加法运算得到顺序表内任意元素的地址（顺序表首地址 + 偏移量），故顺序表访问任意元素的时间复杂度是 $\mathcal{O}(1)$。

顺序表的插入

将元素插入尾部

```
int insert(List * lt, int idx, ElemType val){
    for(int i = lt -> len; i > idx; i --)
        (lt -> lst)[i - 1] = (lt -> lst)[i];
    (lt -> lst)[idx] = val;
    lt -> len ++;
    return 0;
}
```

由于顺序表中的元素是顺序排列的，故将元素插入尾部的时间复杂度是 $\mathcal{O}(1)$（当顺序表长度大于最大长度，进行扩张时，时间复杂度为 $\mathcal{O}(n)$ (其中 n 为表的长度)）

顺序表的插入

将元素插入指定位置

```
template<typename T>
void List<T>::insert(T * pos, T val){
    assert(pos >= begin() && pos <= end());  // 越界检查
    if(len == maxlen)  // 若表已满，则扩张表
        assert(allocater() == 0);
    for(T * iter = end(); iter != pos; iter --)
        *iter = *(iter - 1);  // 将指定位置前的元素后移一位
    *pos = val;  // 将元素插入指定位置
    len ++;  // 更新长度
}
```

在顺序表中，将一个元素插入指定位置，需要移动后面的所有元素，故其时间复杂度为 $\mathcal{O}(n)$。

顺序表的删除

删除指定位置的元素

```
template<typename T>
void List<T>::remove(T * pos){
    assert(begin() <= pos && pos < end());  //越界检查
    for(T * iter = pos; iter + 1 != end(); iter ++)
        *iter = *(iter + 1);  // 将元素向前移一位
    len --;  // 更新长度
}
```

与插入操作类似，在顺序表中删除操作需要移动指定位置后的所有元素，故其时间复杂度同为 $\mathcal{O}(n)$。

顺序表的查找

查找

```
template<typename T>
T * List<T>::find(const T & val){
    for(int i = 0; i < len; i ++){   // 线性查找
        if(lst[i] == val)
            return & (lst[i]);  // 若找到，则返回元素地址
    }
    return NULL;     // 若未找到，则返回 NULL
}
```

同样，在顺序表中查找时，通常的做法是便利顺序表，若找到指定的元素则返回，故其时间复杂度为 $\mathcal{O}(n)$。

顺序表总结

顺序表主要操作的时间复杂度：

- **getitem** 随机访问 $\mathcal{O}(1)$
- **find** 查找指定元素 $\mathcal{O}(n)$
- **insert** 插入元素到指定位置 $\mathcal{O}(n)$
- **remove** 移除指定位置处的元素 $\mathcal{O}(n)$
- **push_back** 在表尾插入元素 $\mathcal{O}(1)$

顺序表适用的场景：

由于顺序表进行随机访问的代价很小，故顺序表适用于进行频繁随机访问的场景（如将连续的字符映射成不同的元素，或存储密集的数据（如稠密的矩阵））

思考题

思考题

在插入排序算法中，我们使用一种线性查找来（反向）扫描已排好的子序列 $A[1,\ldots,j-1]$，我们可以使用二分查找来把插入排序的最坏情况总运行时间改进到 $\Theta(nlgn)$ 吗？