顺序表

麻珂珂，蒋开慧，高浚哲，皇有为

XUPT

2019 年 9 月 24 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;
```

其中，ElemType 为顺序表中元素的类型，len 为线性表中当前元素的个数，maxlen 为当前线性表中所能容纳的最多的元素个数。

顺序表的构造和销毁

顺序表的构造

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

顺序表的构造和销毁

顺序表的销毁

```
void  destoryList(List * lt){
    free(lt -> lst);
    free(lt);
}
```

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 $\mathcal{O}(1)$。

销毁顺序表的时候，释放顺序表所占的内存。时间复杂度：$\mathcal{O}(1)$

顺序表的随机访问

随机访问

```
ElemType getItem(List * lt, int idx){
    return (lt -> lst)[idx];
}
```

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 push_back(List * lt, ElemType val){
    if(lt -> len == lt -> maxlen) return 1;
    (lt -> lst)[(lt -> len) ++] = val;
    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 $\mathcal{O}(1)$

顺序表的插入

将元素插入指定位置

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

顺序表插入元素时，需要移动后面的所有元素，时间复杂度为 $\mathcal{O}(n)$。

顺序表的删除

删除指定位置的元素

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

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

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

同样，在顺序表中查找时，通常的做法是遍历顺序表，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？

提示：对比线性表与链表在随机访问和插入的时间复杂度之间的区别。

思考题

朴素的的插入排序的时间复杂度为 $\mathcal{O}(n^2)$，其中，朴素算法每个部分的时间复杂度为：

- 寻找插入位置：$\mathcal{O}(n)$
- 将元素插入指定位置：$\mathcal{O}(n)$

插入排序每次把一个元素插入到一个有序的线性表中直到所有元素插入完毕，故时间复杂度为 $\mathcal{O}(n^2)$

要将插入排序的时间复杂度降低到到 $\mathcal{O}(nlogn)$ 的级别，则必须将两个操作降到至多 $\mathcal{O}(logn)$ 的水平。

由于插入排序要插入的线性表是有序的，我们很容易想到使用二分查找来寻找插入的位置。对于一个有序的顺序表，寻找一个元素的插入位置仅需 $\mathcal{O}(logn)$ 的事件复杂度。看起来我们成功的将插入排序的时间复杂度降到了 $\mathcal{O}(nlogn)$ ：)

思考题

但一个明显的问题是，在使用 $\mathcal{O}(logn)$ 来找到插入位置后，我们仍需 $\mathcal{O}(n)$ 来将元素插入到指定位置，那么整体的时间复杂度仍会是 $\mathcal{O}(n^2)$。如何解决插入的问题呢？也许你想到了使用刚刚学习的**朴素**链表（单向，双向链表）：在已知插入位置的情况下，链表插入一个元素的时间复杂度为 $\mathcal{O}(1)$，问题解决！但真的如此么？

事实上，时间复杂度为 $\mathcal{O}(logn)$ 的二分查找隐含了一个前提，即进行二分查找的线性表需要支持 $\mathcal{O}(1)$ 时间复杂度的随机访问（一个更弱的前提是在 $\mathcal{O}(1)$ 的时间内完成区间中点元素值的查询）。而朴素链表进行随机访问的时间复杂度是 $\mathcal{O}(n)$，最终的时间复杂度仍为 $\mathcal{O}(n^2)$：（

(PS: 通过对朴素链表的一些修改，我们能得到一种名为跳跃表 (Skip List) 的数据结构，可以完美的满足我们的需求，但在此不进行讨论)