数据结构与算法I 作业8

2019201409 于倬浩

11.2-3

首先这种修改并不会改变每个链表的期望长度，期望长度依然是$\Theta(1+\alpha)$。

对于成功的查询，给出键值，新的算法依然需要$\Theta(1+\alpha)$来遍历链表，得到键值对应的地址，因此这种情况不会变优。

对于失败的查询，给出键值后，只要在链表中遍历到大于要查找元素键值的位置后，即可返回，因此可以减少链表的总遍历次数，但最坏遍历次数依然是$\Theta(1+\alpha)$，因此这种情况会有常数因子的改善。

对于插入操作，原算法只需要插入到表头即可，时间复杂度$\Theta(1)$，然而新的算法需要最坏$\Theta(1+\alpha)$来寻找可以插入的位置，因此新算法插入操作的时间复杂度更高。

对于删除操作，实际上和查询类似，如果给出的是需要删除元素的键值，当我们遍历到第一个大于所指定的键值时，即可知道链表中不存在对应元素，因此直接返回，然而最坏遍历次数并没有改变，因此这种情况下依然只有常数因子的改善，为$\Theta(1+\alpha)$。然而，如果给出需要删除的元素的指针，且链表实现为双向链表，可以直接用$\Theta(1)$的时间复杂度进行单次删除。

11.4-2

```
void Hash_Delete(int key) {
    int pos = h(key), checked = 0;
    while(s[pos].key ≠ key || (s[pos].key == key && s[pos].deleted == true)) {
        if(s[pos].key == -1) break; // 出现未使用的空间，可知key在表中不存在
        ++pos, ++checked; //linear-probing 直接找下一个
        if(pos ≥ M) pos -= M; //越过了边界，等价于取模。
        if(checked ≥ M) return; //当前已经完整遍历了整个表依然找不到，说明key在表中不存在
    }
    s[pos].deleted = true; //找到了要被删除的元素，打上删除标记
}
```

```
void Hash_Insert(int key, int val) {
    if(size ≥ M) throw runtime_error; //表已满，不能继续插入
    int pos = h(key);
    while(s[pos].key ≠ -1 && s[pos].deleted == false) {
        ++pos; //此时并不需要判断是否完整遍历了整个表，因为已经在最开始判断表不满，一定存在可用空间
        if(pos ≥ M) pos -= M;
    }
    s[pos] = data(false, key, val); // 在找到的位置插入元素，其删除标记为假。
}
```