算法分析与设计基础 第十周作业

徐浩博 软件02 2020010108

Problem 1

我们采用势函数$\Phi(T) = |2 \cdot T.num - T.size|$，来看TABEL-DELETE的操作摊还代价.

i)当$\alpha_{i-1} < 1/2$，且DELETE操作不会引起表的收缩，则$c_i = 1$，$\alpha_i < 1/2$，有：

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&= 1 + size_i - 2 \cdot num_i - size_{i-1} + 2 \cdot num_{i-1} \\
&= 1 + (size_i - size_{i-1}) - 2(num_i - num_{i-1}) \\
&= 1 + 1 - 0 = 2
\end{aligned}$$

ii)当$\alpha_{i-1} < 1/2$，且DELETE操作会引起表的收缩，则$c_i = num_i + 1$（新建slot并且复制原有元素），$size_i = 2size_{i-1}/3$（size收缩），$num_i < size_{i-1}/3$（触发收缩的条件）有：

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&= num_i + 1 + size_i - 2num_i - size_{i-1} + 2num_{i-1} \\
&= 1 + (size_i - size_{i-1}) - num_i + 2num_{i-1} \\
&= 1 - size_{i-1}/3 - num_i + 2num_{i-1} \\
&< 1 - num_i - num_i + 2num_{i-1} \\
&= 1 + 2(num_{i-1} - num_i) \\
&= 1 + 2 = 3
\end{aligned}$$

iii)当$\alpha_{i-1} \geq 1/2$，即$num_{i-1} \geq size_{i-1}/2$.

若DELETE引起表收缩，则说明$num_i = num_{i-1} - 1 < size_{i-1}/3$，结合两个不等式，有$size_{i-1}/2 - 1 < size_{i-1}/3$，推出$size_{i-1} < 6$，由此我们进行如下较为宽松的放缩证明摊还代价上界是个常数：

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&< size_{i-1} + 1 + (2 \cdot num_i + size_i) + 0 \\
&< size_{i-1} + 1 + 2size_{i-1} + size_{i-1} \\
&< 6 + 1 + 12 + 6 = 25
\end{aligned}$$

若DELETE不引起表收缩，且$2 \cdot num_i - size_i < 0$，那么考虑到$num_{i-1} \geq size_{i-1}/2$，则有$|2 \cdot num_i -$

$size_i| = |2 \cdot (num_{i-1} - 1) - size_{i-1}| \geq 2$,，故该绝对值式子的最大值为2，有

$$
\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&= 1 + 2 - 2num_{i-1} + size_{i-1} \\
&= 3 - 2(num_{i-1} + size_{i-1}/2) \\
&\leq 3
\end{aligned}
$$

若DELETE不引起表收缩，且$2 \cdot num_i - size_i \geq 0$，那么:

$$
\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&= 1 + 2num_i - size_i - 2num_{i-1} + size_{i-1} \\
&= 1 - 2(num_{i-1} - num_i) + (size_{i-1} - size_i) \\
&= 1 - 2 + 0 = -1
\end{aligned}
$$

综合以上，TABLE_DELETE操作的摊还代价上界是个常数.

Problem 2

a. SEARCH

```
SEARCH(A, n, key):
    for i from k - 1 to 0:
        if A_i is not empty (i.e. n_i == 1):
            result = BINARY_SEARCH(A_i, key)
            if result != -1:
                return i, result
    return -1, -1

BINARY_SEARCH(a, key):
    low = 0, high = a.size - 1
    while low <= high:
        mid = (low + high) / 2
        if a[mid] == key:
            return key
        if a[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
```

```
    return −1
```

首先，BINARY_SEARCH，也即二分查找的最坏复杂度为O(log(length))，其中length是该数组的长度.这些有序数组均没有找到该数是一种最差情况，且其他的情况不会差于此情况；在这种情况下将$A_0 \cdots A_{k-1}$的非空数组均进行了二分查找，那么显然，这些数组均非空时时间开销最大，在这种情况下时间开销$T(n) = \sum_{i=0}^{k} log(A_i.size) = 1 + 2 + \cdots + (log(n) - 1) = (log(n) - 1)log(n)/2 = O(log^2 n)$.

b. INSERT

```
INSERT(A, k, key):
    i = 0
    let aux be an auxiliary array
    while A_i is full (i.e. n_i == 1):
        i = i + 1
    if i == k:
        k = k + 1
    aux[0] = key
    for j from 0 to i − 1:
        merge(A_j, aux, aux)
        clear all the elements in A_j
    copy aux to A_i
    return

merge(a, b, result):
    length1 = length of a
    length2 = length of b
    i = 0, j = 0, k = 0
    let aux be an auxiliary array
    while i < length1 and j < length2:
        if a[i] < b[j]:
            aux[k] = a[i]
            k = k + 1
            i = i + 1
        else:
            aux[k] = b[j]
            k = k + 1
            j = j + 1
    while i < length1:
        aux[k] = a[i]
        k = k + 1
```

```
        i = i + 1
    while j < length2:
        aux[k] = b[j]
        k = k + 1
        j = j + 1
    copy aux to result
    return result
```

我们先分析进位时若干$A_0 \cdots A_{m-1}$合并的时间开销. 可以看到，我们的策略是从A_0开始向上合并；每调用一次merge合并的时间开销线性于a，b两个数组的长度，记为O(c·length)，那么合并这i个数组的时间开销是$\sum_{i=1}^{m-1} c(\sum_{j=0}^{i-1} length(A_j) + length(A_i)) = \sum_{i=1}^{m-1} c(2^{i-1} + c^i) < c(m2^0 + (m-1)2^1 + \cdots + 1 \cdot 2^{m-1}) < c2^{m+1} = O(2^m)$.

最坏情况：显然是所有的低位均需进位，$m = \lfloor log_2(n+1) \rfloor$，则合并数组的开销就是$O(2^m) = O(n)$.

摊还分析：我们采用聚集法进行分析，对于每一次进位合并，时间开销为O(2^m)正比于m个数组的总元素个数，因此我们可以认为每次进位，均摊到每个元素上的时间开销都是O(1)的，对于A_i来说，复杂度就是$c2^i$. 下面我们对每个数组A_i分析，它每次需要合并时的时间开销均是$c2^i$，共需进位$\lfloor \frac{n}{2^i} \rfloor$次，因此插入n个元素时，把所有的$A_i$时间开销相加，得：$\sum_{i=0}^{k-1} c2^i \lfloor \frac{n}{2^i} \rfloor < c\sum_{i=0}^{k-1} n = cnk = cn\lfloor log(n+1) \rfloor = O(nlogn)$. 每个操作的平均代价就是O(nlogn/n)=O(logn).

c. DELETE

```
DELETE(A, n, key):
    i, pos = SEARCH(A, n ,key)
    if(i == -1) return
    n = n - 1
    j = 0
    while A_j is empty (i.e. n_j == 0):
        j++
    if i != j:
        SPLIT(A, j, 0)
        k = pos
        while k > 0:
            if A_i[k - 1] <= A_j[0]:
                A_i[k] = A_j[0]
                break
            else:
                A_i[k] = A_i[k - 1]
            k = k - 1
        if k == 0:
            A_i[k] = A_j[0]
```

```
    else:
    SPLIT(A, j ,pos)

SPLIT(A, i, pos):
    j = i - 1
    k = 0
    while A_j is empty (i.e. n_j == 0):
        size = 2^j
        if k + size - 1 < pos or k > pos:
            copy A_i[k ... k + size - 1] to A_j
        else
            copy A_i[k ... pos - 1] and A_i[pos + 1 ... k + pos] to A_j
```

算法可以总结为：先搜索要删除的元素位置，这一步是O(logn)的. 假设要删除的元素在A_i内，而A_j为j最小的非空数组，那么删除时需要将A_j一个元素给A_i（此处我们给出$A_j[0]$），将$A_j[0]$在A_i中安插好，然后将A_j分裂成$A_0 \cdots A_{j-1}$. 一种特殊的情况是$i == j$，则直接将A_i分裂即可. 考虑到数组均是有序的，因此无论是安插$A_j[0]$还是分裂A_j，都和元素个数是呈线性的. 考虑到A_j元素不多于A_i，因此删除操作的复杂度近似和A_i的元素个数成正比，为$O(2^i)$.

下面我们估算删除已有的每个元素平均用时：对于A_i，他有2^i个元素，因此删除的元素在2^i内的概率为$2^i/2^k$，删除的时间开销为$c2^i$. 因此计算平均时间开销有$\sum_{i=0}^{k-1} c2^i \times p(A_i) = c/2^k \cdot \sum_{i=0}^{k-1} 2^{2i} < c2^k = O(2^{logn}) = O(n)$. 即平均时间开销是O(n)的.