Solutions, Class 03/28: Amortized Analysis & Union-Find

A space-efficient strategy for table growth

When we allocate a new table of twice the size and copy the old table into the new one, the new table is only half full. Prof. Efficiency thinks this is a waste of space and suggests the following solution: instead of doubling the size of the table when it becomes full, increase the table size by a fraction ε (epsilon) instead and copy the elements into this new table. For example, when ε = .5, if a table is full and contains *m* elements, the newly allocated table has capacity $\lceil 1.5m \rceil$. This way, we are using only at most 50% more space than actually is needed to store the elements in the table.

1. Deriving the Amortized Time per Operation. Let's analyze the effect of Prof. Efficiency's proposal on the time to insert into such a table. Assume we increase the size of the table by an arbitrary constant epsilon ε. Assume the table T contains *m* elements and is full.

(a, 0.5 pt) Total Capacity: What will be the size (total capacity) of the new the table T' after resizing? (This will also be the number of items that have to be copied when T' is resized)

Answer: $(1+\varepsilon)m$

(b, 0.5 pt) Available Space: How many elements can we insert into T' before it gets resized?

Answer: εm

(c, 1.0 pt) Credit Available for Next Resize: Assuming we charge *x* CyberDollars for each insertion, how much credit will we have by the time T' has to be resized? (Don't forget the cost of the insertion itself.)

Answer: We spend one CyberDollar to do the insertion, and the rest (x-1) are the credit. Therefore, we have $(x-1)\varepsilon m$ credit.

(d, 1 pt) Amortized Cost of Each Insertion: How much should we charge per each insertion to make sure we have enough credit to copy all elements next time we resize the table? That is, what is *x* as a function of ε? Using the expressions you wrote above, set up an equation where credit ≥ cost and solve for *x*.

Answer: To copy all elements at next resizing, we need to copy $(1+\varepsilon)m$ items. Thus, to have enough credit, we need to solve the equation $(x-1)\varepsilon m \geq (1+\varepsilon)m$, which solves to

$x \geq 1 + \frac{1+\varepsilon}{\varepsilon}$ or 2 + 1/ε (or equivalent)

2. Specific cases. Now let's evaluate the amortized time-cost of Prof. Efficiency's proposal for specific ε. Compute the cost of insertion if:

(a, 0.25 pt) ε = 50% (Prof. Efficiency's proposal.)

Answer: $1 + \frac{1+0.5}{0.5} = 4$

(b, 0.25 pt) ε = 10% (Prof. Stingy thinks that Prof. Efficiency has not gone far enough.)

Answer: $1 + \frac{1+0.1}{0.1} = 12$

(c 0.25 pt) ε = 100% (Doubling of the table, as in the CLRS version.)

Answer: $1 + \frac{1+1}{1} = 3$

(d, 0.25 pt) ε = 400% (Prof. Generous has plenty of space to spare.)

Answer: $1 + \frac{1+4}{4} = 2.25$

(e, 1 pt) Conclusions: What do the above results tell us about time/space tradeoff in tables with expansion? How does this tradeoff affect the big-O amortized cost of such tables?

- Allocating more space makes the amortized cost per operation faster (and less makes it slower) - 0.5 pt
- but it is still O(1) - 0.5 pt.

---

3. (Extra Credit) Iterative Find-Set

Find-Set as written in the text is not tail recursive, so a compiler won't be able to automatically generate efficient iterative executable code. Write an iterative version of Find-Set.

> Of course, many variations are possible, but they must (a) do path compression and then (b) return the root of the tree in the end. Since you don't know who the root is until you reach it, vertices passed along the way must be saved -- a stack is a convenient way to do this -- and then made to point to the root once it is found.

```
Find-Set(x) {
    S = new Stack()
    while x != x.p {
        S.push(x)
        x = x.p }
    // now x is at the root of the tree
    // pop all vertices visited and do path compression
    while !S.isEmpty() {
        S.pop().p = x }
   return x
}
```

> A common error is to forget to set the parent of *all* vertices on the path to x.