

Topics 15-16: Amortized Analysis & Union-Find Solutions

Copyright (c) 2020 Daniel D. Suthers & Peter Sadowski. All rights reserved. These solution notes may only be used by faculty, TAs and students in ICS 311 Fall 2020 at the University of Hawaii.

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 $\epsilon = .5$, 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. Here we will analyze the effect of Prof. Efficiency's proposal on the time to insert into such a table.

1. Parameterizing the Algorithm (0.5pt). The code for Table-Insert from the lecture notes is below, with ϵ added as a parameter `epsilon`. Change the code to use `epsilon`, **boldfacing** your changes. Remember that table size is an integer.

```
Table-Insert (T,x,epsilon)
1  if T.size == 0
2      allocate T.table with 1 slot
3      T.size = 1
4  if T.num == T.size
5      allocate newTable with ceiling((1+epsilon)*T.size) slots
6      insert all items in T.table into newTable
7      free T.table
8      T.table = newTable
9      T.size = ceiling((1+epsilon)*T.size)
10 insert x into T.table
11 T.num = T.num + 1
```

2. Deriving the Amortized Time per Operation. Let's derive a general formula for the time to insert into a table, given ϵ . Assume we increase the size of the table by an arbitrary constant epsilon ϵ . Assume the table T contains m elements and is full. (Use m in your formulas, not $T.size$.)

(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: $(1 + \epsilon)m$... or $\text{ceiling}((1 + \epsilon)m)$

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

Answer: ϵm ... or $\text{ceiling}(\epsilon m)$

(c, 0.5 pt) Credit Available for Next Resize: Assume each insertion into the table costs us \$1, but we charge customers $\$c > \1 for each insertion in Part (b) in order to raise credit for covering the cost of resizing the table when it gets full. How much credit will we have accumulated by the time T' has to be resized? Give your answer in terms of c , m , and ϵ .

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

(d, 1.0 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 c as a function of ϵ ? Using the expressions you wrote above, set up an equation where $\text{credit} \geq \text{cost}$ and solve for c .

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

$$(c-1)\epsilon m \geq (1 + \epsilon)m$$

which solves to

$$c \geq 1 + \frac{1+\epsilon}{\epsilon} \quad \text{or} \quad 2 + 1/\epsilon \quad (\text{or equivalent})$$

3. 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) $\epsilon = 0.5$ (Prof. Efficiency's proposal.)

$$\text{Answer: } 2 + \frac{1}{0.5} = 4$$

(b, 0.25 pt) $\epsilon = 0.1$ (Prof. Stingy thinks that Prof. Efficiency has not gone far enough.)

$$\text{Answer: } 2 + \frac{1}{0.1} = 12$$

(c 0.25 pt) $\epsilon = 1.0$ (Doubling of the table, as in the CLRS version.)

Answer: $2 + \frac{1}{1} = 3$

(d, 0.25 pt) $\epsilon = 4.0$ (Prof. Generous has plenty of space to spare.)

Answer: $2 + \frac{1}{4} = 2.25$

(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 the amortized cost is still $O(1)$ - 0.5 pt.

This is another classic time/space tradeoff, although there are diminishing returns for large ϵ since it is in the denominator.

4. Challenge Problem: Iterative Find-Set

The Find-Set method for Dynamic Disjoint Sets as written in CLRS Section 21.3 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 that uses path compression.

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. The recursive solution does this implicitly with the runtime stack, suggesting that we use an explicit stack to record vertices on the path, and then pop the stack to make them point to the root once it is found.

```
Find-Set-Iterative(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
}
```

But one can also do it without a stack, just by climbing the parent pointers twice. This is more efficient: why allocate a stack object when you just need the path?

```
Find-Set(x)
    r = x          // we need the x
    while r != r.p
        r = r.p
    // now r is the root of the tree
    // set all parents on path to r to do path compression
    while x != r
        p = x.p    // preserve the parent
        x.p = r    // compress
        x = p      // go to parent
    return x       // or r
```

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