H311: Honors Colloquium – Introduction to Algorithms

Lecture 2: Amortized Analysis

Marius Minea

University of Massachusetts Amherst

9 February 2021

Amortized Analysis: Overview

Usually, we analyze *worst-case* execution time:

- for an algorithm
- or individual data structure operations

Sometimes, the cost of an operation varies significantly

Counting *each* operation at *maximum* cost would needlessly overestimate total running time

Amortized analysis computes the cost *per operation*, for the *entire algorithm*

Sources and extra material:

CMU 15-451 Spring 2007 (Manuel Blum)
https://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0206.pdf

Duke COMPSCI 330 Spring 2017 (D. Panigrahi)
https://www2.cs.duke.edu/courses/spring17/compsci330/Notes/UnionFindAmortize.pdf

Example: Extensible Array (Stack)

Consider an array that can grow with a `push()` operation

- if not full, `push()` is $O(1)$
- if full, array must be reallocated and copied $\implies$ high cost

What is the overall cost of doing n `push()` operations?

Depends on when / how much the array is extended.

If every resize doubles the array, cost of resizing is
$1 + 2 + 4 + \ldots + 2^i$ with $2^i < n$ (end size). Total $< 2n$

Amortized cost per operation is $< 1 + 2 = 3$.

Our computation used *aggregate method* (sum, then average)

Accounting Method ("piggy-banking")

Idea: we have *cheap* and *expensive* operations.

For each cheap operation, *budget* more than its actual cost

Use savings to pay for expensive operations (never go negative)

Budget cost 3 instead of 1 for array `push()`

When growing L to $2L$, $L/2$ elements are new since last growth
$\implies$ accumulated $2 \cdot L/2$ pays for copying cost

Variant: *potential method*: define nonnegative function that depends only on state of the system (like accumulated savings)

Binary Counter

Count: 0, 1, 10, 11, 100, 101, . . .

Every bit flip has cost 1.
Each increment is $O(\log n)$ coarse bound, flips may be few
What is the *amortized* cost per increment ?

Aggregate method:
bit 0 flipped n times
bit 1 flipped every other time: $n/2$, etc.
Total: $n + n/2 + n/4 + ... < 2n \implies$ amortized cost 2 ($O(1)$)

Accounting method:
Budget 2 (not 1) when flipping 0 to 1 (once on every increment)
Use 1 for every flip from 1 to 0
Can't go negative (can flip at most all ones) $\implies$ same result

Binary Counter (high bit, high cost)

Now assume cost 2^k to flip bit k.

Single increment could cost $1 + 2 + ... + 2^{\log n - 1} \simeq n$

Amortized cost is $\log n$ (by aggregation):

$n + 2 \cdot n/2 + 4 \cdot n/4 + \ldots = n \log n$

Amortized Dictionary (Searchable Array)

Alternative to balanced search tree.

Keep all items (numbers) in sorted array segments of size 2^k, given by binary representation of element count n.
(e.g. for 19, lengths 16 + 2 + 1)

Search time is $\log 2 + \log 4 + ... + \log n = O(\log^2 n)$

Adding a new element: create array segment of length 1 and propagate up by merging equal-length arrays

e.g., for 20, merge 1 + 1 = 2, merge 2 + 2 = 4. Result: 16 + 4.

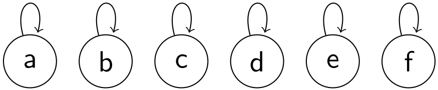
Cost to merge: $2 \cdot 2^k$ to merge arrays of length 2^k
$\implies$ like binary counter with cost 2^k for bit k.

$\implies$ amortized cost is $O(\log n)$ per insert.

Union-Find Data Structure

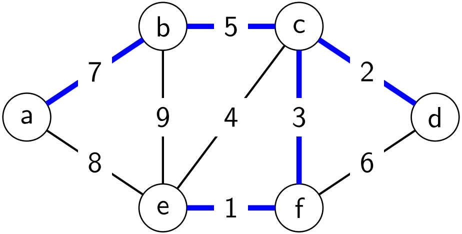
Clever data structure to maintain equivalence classes:

- Find(v): return name of set containing v
- Union(A, B): merge two sets
- Each set elects a representative to act as the “name” of the set
- Nodes point to their representative
- Initially, every node points to itself



Union-Find Data Structure

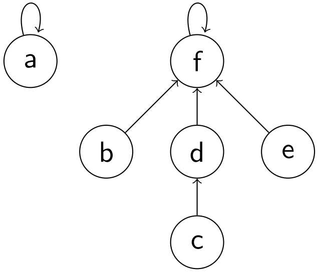
Example: gradually link all nodes into a *spanning tree*



- Union(e, f)
- Union(c, d)
- Union(c, f)
- Union(b, c)
- Union(a, b)

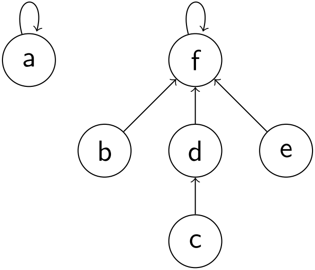
- Time for union? $O(1)$: update one pointer

Union-Find Data Structure



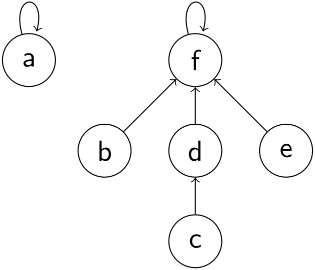
- Union(a, f): which pointer should be updated?
- **Convention**: smaller set changes its name
- Time for Find? Equal to depth of tree

Union-Find Data Structure



- **Claim**: let $d =$ depth and $k = \#$ nodes in set.
- Then $d \leq \log_2(k) \implies$ Find is $O(\log n)$
- Proof by induction

Union-Find Data Structure



- **Invariant**: let d = depth and k = # nodes in a given set. Then $k \geq 2^d$
- Base case: $d = 0, k = 1$ ✓
- Induction step: consider union of sets of size $k_L < k_R$ with depths d_L and d_R
- New depth is $d = \max\{d_L + 1, d_R\}$
  - $k = k_L + k_R \geq 2k_L \geq 2 \cdot 2^{d_L} \geq 2^{d_L+1}$
  - $k = k_L + k_R \geq k_R \geq 2^{d_R}$
- Therefore $k \geq 2^d \implies d \leq \log_2(k)$

Union-Find: Constant-Time Find

Alternate goal:
Find in $O(1)$, using star graphs (each node points directly to root).

Time for Union ?

Budget one extra unit when joining T_s with T_l $(|T_s| \leq |T_l|)$

An element can be joined $\leq \log n$ times (since size doubles)

$\implies$ total cost of union $\leq n \log n$, amortized $\log n$.

Union-Find with Path Compression

Use trees with parent pointer. When going up on Find,
all nodes on path get linked directly to root (trees get "bushier")

Amortized complexity of Find: $\log^* n$ (iterated logarithm), where:

$\log^* 1 = 0$, $\log^* n = 1 + \log^*(\log n)$, grows very slowly:
$\log^* 2 = 1, \log^* 2^2 = 2, \log^* 2^4 = 3, \log^* 2^{16} = 4, \log^* 2^{65536} = 5, ...$

Keep doing union by rank (depth); if merging two equal-rank trees, rank of root increases by 1.

Tree of rank r has at least 2^r nodes. At most $n/2^r$ roots of rank 2^r (if all roots of rank r have trees with 2^r nodes).

Group nodes into "buckets" by $\log^* r$. In bucket $[k+1, 2^k]$ at most $n/2^{k+1} + n/2^{k+2} + ... = n/2^k$ nodes. Path cost for each $\leq 2^k$.
$\implies$ cost n per bucket.

Total effort spent is $O(n \log^* n) \implies O(\log^* n)$ amortized