07. Amortized Analysis and Fibonacci Heaps

CPSC 535 $\sim$ Spring 2019

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY FULLERTON

March 11, 2019

Amortized Efficiency

amortized performance

- *amortized efficiency:* average of worst-case time over any sequence of operations
- different from (probabalistic) expected efficiency, which is averaged over random choices made by a PRNG
- amortized time bound is a weaker math condition than the same worst-case bound
- e.g. $\Theta(1)$ amortized vs. $\Theta(1)$ worst-case
- $\implies$ downgrading to an amortized bound may admit upgrades to other aspects of a data structure

Big Idea in Algorithm Design: Work Smart not Hard

Lifecycle of an implemented algorithm:

Phase of Life	**Frequency of Creation**
design and analysis	once, by discoverer and peer reviewers
learned by students	annually, by thousands
implemented	once per programming language lifetime (decade)
run and maintained	millions-billions of times indefinitely

- given the choice, we'd prefer for the ongoing tasks to be easy
- even if that means the one-time phases are complicated
- holy grail: algorithm is tough to conceive and analyze, but easy to understand and implement
- example: universal hashing, open addressing

Amortized Analysis — What to Prove

A *splay tree* is a kind of binary search tree.

Lemma: the *INSERT*, *SEARCH*, and *DELETE* operations each take $\Theta(\log n)$ amortized time on a splay tree.

Would need to prove:

- average time/operation $= \Theta(\log n)$
- or, any sequence of n of these operations takes a grand total of $n \cdot \Theta(\log n) = \Theta(n \log n)$ worst-case time
- <u>any</u> sequence: includes the worst-case for the data structure
- three conventional proof techniques

Aggregate Analysis

- count total time $T(n)$ for any sequence of n operations
- $\implies$ each operation takes $T(n)/n$ amortized time
- pro: simple logic; from first principles
- con: only works when the goal is to prove all operations take the same time (e.g. *INSERT*, *SEARCH*, and *DELETE* are each $\Theta(\log n)$); wouldn't work if one of them is $\Theta(1)$
- con: not much inspiration for difficult analyses

The Accounting Method

- inspired by financial amortization, balanced budgets
- analyst picks a **cost** for each operation (ex.: *INSERT* costs $4 \cdot \lceil \log_2 n \rceil$ units)
- some operations **over-charge** and turn a **profit** that is deposited somewhere in the structure (e.g. in a node)
- other operations **under-charge** and incur a **loss** that is withdrawn from prior deposits
- show: never go bankrupt, i.e. withdrawn units always exist
- show: for every operation,

$$\text{(charge)} + \text{(withdrawal)} \leq \text{actual time spent}$$

- $\implies$ amortized time = charged cost

Accounting Method Pros/Cons

- pro: possible to prove different amortized efficiency classes for each operation (e.g. one is $\Theta(1)$, another $\Theta(\log n)$)
- pro: cost story helps us reason through analysis
- loss operation = procrastinating, deferring work to later
- profit operation = catching up on deferred work
- con: cost story may overcomplicate things
- con: sometimes awkard to store profits in specific data structure locations

Potential Method

- premise: for data structure D, **potential function** $\Phi(D)$
- (Φ pronounced like "fee")
- potential is stored energy, like a battery
- amortized time of operation = actual time + change to $\Phi(D)$
- procrastinating/loss operations decrease $\Phi(D)$
- catch-up/profit operations increase $\Phi(D)$
- show: $\Phi(D) \geq 0$ always
- ultimately a less structured way of thinking of the accounting method

Fibonacci Heap Overview

- alternative to a binary heap (as in heapsort)
- pro: faster INSERT and DECREASE-KEY operations (optimal in fact)
- pro: supports additional operations
- con: more complicated to describe, implement, *especially* analyze
- con: operations have amortized time bounds
- con: much worse constant factors, not in-place
- not currently practical

Mergeable Heap API

CREATE-HEAP(): initialize empty heap
INSERT(H, x): insert entry x into H
MINIMUM(H): return minimum-key entry
EXTRACT-MIN(H): remove and return minimum-key entry
UNION(H_1, H_2): consume heaps H_1, H_2 and return a new heap with their elements
DECREASE-KEY(H, x, k): lower key of x to k
DELETE(H, x): remove entry x from H

Heap Efficiency Comparison

Operation	Binary Heap (worst-case)	Fib. Heap (amort.)
CREATE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\log n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$

Fibonacci Heap Structure

Overall heap H has

- a *forest* of individual k-ary trees (nonbinary); each tree in min-heap-order
- $H.min$ = pointer to the root with the global minimum key
- $H.n$ = number of entries in heap

Each node x has

- $x.parent$ = parent node (NIL if root)
- $x.child$ = pointer to an arbitrary child
- $x.left, x.right$ = pointers to siblings
- (siblings are in circular, unsorted, doubly-linked lists)
- $x.degree = \#$ children
- $x.mark$ = boolean, true iff x has lost a child since the last time x was assigned a new parent (manages procrastination)

Potential Function and Degree Bound

For analysis purposes, we define the potential function

$$\Phi(H) = t(H) + 2 \cdot m(H)$$

where

- $t(H) = \#$ trees in root list
- $m(H) = \#$ marked nodes

Claim: If Fibonacci heap H has n nodes, then the maximum degree ($\#$ children) of any node, written $D(n)$, is

$$D(n) = O(\log n).$$

CREATE-HEAP

```
function CREATE-HEAP
    H.min = NIL
    H.n = 0
    return H
end function
```

Takes $\Theta(1)$ time; $t(H) = m(H) = 0$, so $\Phi(H) = 0$; no profit or loss.

INSERT Pseudocode

Just create node, initialize node, insert into root list, update $H.min$

```
function INSERT(H, x)
    x.degree = 0
    x.parent = x.child = NIL
    x.mark = false
    if H.min == NIL then
        H.min = x
        x.left = x.right = NIL
    else
        insert x into root list as H.min's right sibling
        if x.key < H.min.key then
            H.min = x
        end if
    end if
    H.n++
end function
```

INSERT Analysis

Actual time steps are $\Theta(1)$.
Let H be heap before INSERT, and H' after; then

$t(H') = t(H) + 1$ (one tree created)
$m(H') = m(H)$ (no marking)
$\implies \Phi(H') = \Phi(H) + 1$

$\therefore$ INSERT takes $O(1) + 1 = O(1)$ amortized time.

MINIMUM

Just follow the $H.min$ pointer.

$O(1)$ actual time.

No new trees, no new marked nodes, so $O(1)$ amortized time.

UNION

```
function UNION(H_1, H_2)
    if H_1.n == 0 then
        return H_2
    else if H_2.n == 0 then
        return H_1
    else
        H = new heap object
        H.min = min(H_1.min, H_2.min)
        H.n = H_1.n + H_2.n
        concatenate root lists of H_1, H_2 into one linked list
        return H
    end if
end function
```

$O(1)$ actual time; trees and marked nodes move around, but their number is unchanged, so no change to Φ; $O(1)$ amortized time.

So Far So Good

So far all operations have been simple, and either potential-neutral (CREATE-HEAP, MINIMUM), or increased potential (INSERT)

Indeed, n INSERTs creates a glorified n-element linked list.

$\therefore$ need to expect remaining operations to be more complicated, decrease potential, decrease length of root list.

EXTRACT-MIN

Follow $H.min$ pointer; promote all children to roots; remove from root list; CONSOLIDATE to shorten root list and find new $H.min$.

```
function EXTRACT-MIN(H)
    z = H.min
    for each child c of z do
        insert c into H's root list
        c.parent = NIL
    end for
    delete z from root list
    if H.n == 1 then
        H.min = NIL                          ▷ heap just became empty
    else
        CONSOLIDATE(H)        ▷ compact root list, recompute H.min
    end if
    H.n − −
    return z
end function
```

Zooming in to CONSOLIDATE

Before moving on, observe EXTRACT-MIN takes, aside from CONSOLIDATE, $\Theta(\text{degree}(H.min))$ time; claimed this is $\Theta(\log n)$

Contract for CONSOLIDATE:

1: **function** CONSOLIDATE(H)
Require: $H.min$ invalid and $H.n == \#\text{nodes} + 1$
Ensure: $H.min$ valid and $\#\text{trees} \in O(\log n)$
2: **end function**

CONSOLIDATE Pseudocode

```
function CONSOLIDATE(H)
    A = UNIQUE-DEGREE-ARRAY(H)
    ARRAY-TO-ROOT-LIST(H, A)
    free A
    RECOMPUTE-MIN(H)
end function
```

Clearly $\Theta(1)$ time except for the three subroutines.

UNIQUE-DEGREE-ARRAY Pseudocode

```
function UNIQUE-DEGREE-ARRAY(H)
Ensure: returns A[0..D(H.n)] where A[d] = only root w/ degree d
    A[0..D(H.n)] = new array of node pointers, all NIL
    for each root node r in H do                          ▷ move r into A
        p = r                        ▷ parent node that needs to move into A
        while A[p.degree] ≠ NIL do              ▷ another node in the way
            c = A[p.degree]                          ▷ p, c have same degree
            A[p.degree] = NIL              ▷ we will link them into one tree
            if p.key > c.key then
                swap(p, c)                  ▷ ensure c should be p's child
            end if
            make c a child of p, incrementing p.degree
            c.mark = false
        end while
    end for
    return A
end function
```

UNIQUE-DEGREE-ARRAY Analysis

Create A: $O(D(n))$ time

for loop — aggregate analysis

- # iterations = length of root list before CONSOLIDATE = $t(H) + D(n) - 1$
- each iteration of inner **while** loop links two roots into one, decrementing # roots
- # roots at end of CONSOLIDATE $\leq$ size of $A = D(n) + 1$
- $\implies$ total time in all **while** iterations is

$$\big(t(H) + D(n) - 1\big) - \big(D(n) + 1\big) = t(H) - 2$$

- $\implies$ total time in **for** loop is

$$\big(t(H) + D(n) - 1\big) + \big(t(H) - 2\big) \leq 2 \cdot t(H) + D(n)$$

total for UNIQUE-DEGREE-ARRAY is $O(t(H) + D(n))$

ARRAY-TO-ROOT-LIST

```
function ARRAY-TO-ROOT-LIST(H, A)
    clear H's root list to empty
    for index i of A do
        if A[i] ≠ NIL then
            insert A[i] into H's root list
        end if
    end for
end function
```

$O(D(n))$ time

RECOMPUTE-MIN

```
function RECOMPUTE-MIN(H)
    H.min = NIL
    for node r in H's root list do
        if H.min == NIL then
            H.min = A[i]
        else if H.min.key > r.key then
            H.min = r
        end if
    end for
end function
```

$O(D(n))$ time

CONSOLIDATE Worst-Case Analysis

```
function CONSOLIDATE(H)
    A = UNIQUE-DEGREE-ARRAY(H)        ▷ O(D(n) + t(H))
    ARRAY-TO-ROOT-LIST(H, A)          ▷ O(D(n)))
    free A                            ▷ O(1)
    RECOMPUTE-MIN(H)                  ▷ O(D(n))
end function
```

Time spent is

$$O(D(n)+t(H)+D(n)+1+D(n)) = O(3\cdot D(n)+t(H)) = O(\log n+t(H)).$$

Want this to be $O(\log n)$; the $t(H)$ overage can be withdrawn from the potential function.

CONSOLIDATE Amortized Analysis

Recall potential function

$$\Phi(H) = t(H) + 2 \cdot m(H)$$

Let H' be H after EXTRACT-MIN
EXTRACT-MIN does not mark any nodes, so $m(H') = m(H)$.
roots decreases: $t(H') = D(n) + 1 \leq t(H)$

$$\Phi(H') = t(H') + 2 \cdot m(H') = \big(D(n) + 1\big) + 2 \cdot m(H)$$

$$\Phi(H) - \Phi(H') = t(H) - \big(D(n) + 1\big)$$

Amortized cost of EXTRACT-MIN

$$= \Delta\Phi = O(t(H) + D(n)) - O(t(H) - D(n) - 1) = O(2 \cdot D(n)) = O(\log n)$$

Where did that time come from? Accounting Perspective

- potential function's $t(H)$ term over-charges INSERT and under-charges EXTRACT-MIN
- when $t(H) = O(\log n)$, EXTRACT-MIN takes $O(\log n)$ worst-case time
- first $\log n$ roots are "clean", remaining $t(H) - \log n$ are "mess"
- deposit $O(1)$ time when INSERTing a messy root
- withdraw those deposits in CONSOLIDATE
- 1 deposit pays for making 1 messy root a child of a clean parent
- (recall that link operation takes $O(1)$ time)
- hard to use accounting method directly because the identity of the clean nodes changes during CONSOLIDATE based on heap-order

DELETE

function DELETE(H, x)
  DECREASE-KEY$(H, x, -\infty)$
  EXTRACT-MIN(H)
end function

Know EXTRACT-MIN is $O(\log n)$ amortized time.

Claim: DECREASE-KEY is $O(1)$ amortized time.

$\implies$ DELETE is $O(\log n)$ amortized time.

DECREASE-KEY Sketch

DECREASE-KEY(H, x, k)

- update $x.key = k$
- if x is a root, or heap-order maintained, done
- else **cut** x
    - make x a root, no longer a child of parent p
    - update $H.min$ if x is new minimum
    - mark p; but if p was already marked, recursively cut p
- $O(1)$ time not counting recursive cuts
- recursive cuts are paid for by withdrawing potential

Heap Efficiency Comparison

Operation	Binary Heap (worst-case)	Fib. Heap (amort.)
CREATE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\log n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$