Today: Amortization
- aggregate method
- accounting method
- charging method
- potential method

Different approaches/techniques for amortized analysis—all related, but one often easier than others

- table doubling
- binary counter
- 2-3 trees

Examples of amortized analysis

Powerful technique for data structure analysis
- often, what you really care about

Recall: table doubling [6.006]
- $n$ elements in table of $m$ slots
- want $m = \Omega(n)$ for $1 + \frac{m}{n} = O(1)$ expected performance (with hashing with chaining)

- idea: if $n$ grows $\geq m$, double $m$
- cost: $\Theta(m+n) = \Theta(n)$ to build new table

\[\Rightarrow \text{pay } \Theta(2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^\lceil \log n \rceil) = \Theta(n)\]

Total to resize table over $n$ insertions

\[\Rightarrow \Theta(1) \text{ amortized cost per insertion}\]
Aggregate method: “just add it up”

\[
\text{total cost of } k \text{ operations} = \frac{k}{\text{amortized cost per operation}}
\]
- Common only for simple analyses

**Amortized bounds:**
- Assign an “amortized cost” to each operation such that “preserve total”:
  \[\Sigma \text{amortized costs} \geq \Sigma \text{actual costs}\]
  (over all operations, for any operation sequence
  average is just one option)
- E.g. can say 2-3 trees achieve
  \[O(1) \text{ worst-case per create-empty}\]
  \[O(lg n^*) \text{ amortized per insert}\]
  \[O \text{ amortized per delete (assuming exists)}\]

where \(n^* = \text{maximum size of set at any time}\)
because \(c\) creations, \(i\) insertions, \(d\) deletions

cost \[O(c + (i+d) lg n^*) = O(c + i lg n^* + \theta d)\]

\[\leq 2i\]

- We'll tighten to \(O(lg n)\) where
  \(n = \text{current set size}\), below
Accounting method: "planning ahead for rainy day"
- allow an operation to store credit (like bank)
  ⇒ amortized cost > actual cost
- allow operations to pay using existing credit
  ⇒ amortized cost < actual cost

Example: table doubling
- when inserting an element, add a coin to it representing \( c = \Theta(1) \) work
- when table needs to double \( n \rightarrow 2n \), \( n/2 \) new elements still with coins

\[
\begin{array}{cccccccc}
\times & \times & \times & \times & \times & \times & \times & \times \\
\circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ
\end{array}
\]

- use up those coins to pay for \( \Theta(n) \) rebuild

\[
\begin{array}{ccccccccc}
\times & \times & \times & \times & \times & \times & \times & \times & \times \\
\circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ
\end{array}
\]

⇒ \( \Theta(n) - \frac{n}{2} \cdot c \) amortized rebuild cost
  = 0 for large enough \( c \)
- \( O(1) + c = \Theta(1) \) amortized cost per insert

Counterexample: free deletion in 2-3 trees
- claim: \( O(\log n) \) am. insert, 0 am. delete
- attempt: put coin worth \( \Theta(\log n) \)
  on inserted element
- trouble: when deleting that element, \( n \) might be bigger ⇒ coin worth too little
Charging method: (blaming the past) (not in CLRS)
- allow operations to charge cost retroactively to past operations (not future ops)
- amortized cost of op. = actual cost + total charge to past ops. + total charge by future ops. to this op.

Example: table doubling
- when table doubles \( n \rightarrow 2n \), charge \( \Theta(n) \) cost to \( n/2 \) inserts since last doubling
  \( \Rightarrow \) each of these elements charged \( \Theta(n)/n/2 = \Theta(1) \)
  \( \Rightarrow \Theta(1) \) amortized per insert

Example: table doubling & halving
- motivation: want \( \Theta(n) \) space even with deletes
- if table down to \( 1/4 \) full \( (n = m/4) \):
  shrink to half size \( (m \rightarrow m/2) \) at \( \Theta(m) \) cost
  \( \Rightarrow \) still half full after any resize
  \( \Rightarrow \) still \( \geq m/2 \) inserts to charge to on growth
- also \( \geq m/4 \) deletes to charge to on shrink
- each operation charged \( \leq \) once, by \( \Theta(1) \)
  \( \Rightarrow \Theta(1) \) amortized per insert & delete

- could do this argument with coins instead, but less intuitive (to me)
Example: free deletion in 2-3 trees
- **claim**: $O(lgn)$ am. insert, $\emptyset$ am. delete
- insert charges nothing
- delete charges one insert:
  - not the insertion of same element
  - (same problem as accounting method)
  - insertion that brought $n$ to its current value
- before $n$ can reach this value again, must have another insert
  $\Rightarrow$ each insert charged at most once
Potential method: (defining karma)
- define a potential function $\Phi$ mapping data-structure configuration $\to$ nonnegative integer
  - intuitively measuring “potential energy”
    = potential high costs in the future
    = equivalent to total unused credit (≥ unused coins) stored by all past ops.
    = bank account balance
  - nonnegative $\Rightarrow$ never owe the bank
- amortized cost = actual cost + $\Delta \Phi$
  = $\Phi(\text{DS after op.}) - \Phi(\text{DS before op.})$
$\Rightarrow$ sum of amortized costs telescopes
= sum of actual costs + $\Phi(\text{final DS}) - \Phi(\text{initial DS})$
  $\geq 0$ initial balance
- so also need to pay $\Phi(\text{initial DS})$ at start
  ~ ideally 0 or $O(1)$ ~ else another amortization

- in accounting method, specify offset ($\Delta \Phi$) between actual cost & amortized cost, which determines total stored value ($\Phi$)
- in potential method, specify total stored value $\Phi$, which determines changes per op.: $\Delta \Phi$
- sometimes one is more intuitive than other
- potential method feels most powerful (to me) but also the hardest to come up with proof($\Phi$)
Example: binary counter
- operation: increment
- increment costs $\Theta(1 + \#\text{trailing \ 1 bits})$
  
  So intuition is that 1 bits are bad
- define $\Phi = c \cdot \#1 \text{ bits in counter}$
  
  $\Rightarrow \Delta \Phi$ from increment $= c(-\#\text{trailing \ 1 bits} + 1)$
  
  $\Rightarrow$ amortized cost $= \text{actual cost} + \Delta \Phi$
  
  $= \Theta(1 + \#\text{trailing \ 1 bits}) + c(-\#\text{trailing \ 1 bits} + 1)$
  
  $= O(1)$ for $c$ large enough
- $\Phi(\text{initial DS}) = 0$ assuming we start $@000...0$
  
  (necessary for $O(1)$ amortized bound)

Example: insert in 2-3 trees
- $O(\lg n)$ splits in worst case
  
  but claim only $O(1)$ amortized splits
  
  what causes splits? nodes overflowing
- $\Phi = \# \text{ nodes with 3 children}$
  
  $\Rightarrow \Delta \Phi \leq 1 - \#\text{splits}$
  
  add child @ top $\Rightarrow$ each split turns $\bigtriangledown$ into $\bigtriangleup$
  
  $\Rightarrow$ amortized $\#\text{splits} = \text{actual } \#\text{splits} + \Delta \Phi$
  
  $\leq \#\text{splits} + (1 - \#\text{splits}) = 1$
- $\Phi(\text{initial DS}) = 0$ if we start empty

In B-trees: $\Phi = \# \text{ nodes with B children}$
In (a,b)-trees: $\Phi = \# \text{ nodes with 6 children}$
Example: insert & delete in (2,5)-trees
- claim $O(1)$ amortized splits & merges
- overflows cause splits $\rightarrow$ 5-nodes
- underflows cause merges $\rightarrow$ 2-nodes
- $\Phi = \# 5$-nodes + $\# 2$-nodes
- insert: $\Delta \Phi \leq 1 - \#$ splits
  make a 5-node
  from final merge
  destroy 5-nodes (\& no new 2-nodes)

OVERFULL:
\[
\begin{array}{c}
\text{5 keys} \\
\text{6 children}
\end{array}
\Rightarrow
\begin{array}{c}
\text{5 keys} \\
\text{3-node}
\end{array}
\begin{array}{c}
y, s \\
\text{3-node}
\end{array}
\]
- delete: $\Delta \Phi \leq 1 - \#$ merges
  make a 2-node
  from final steal
  destroy 2-nodes (\& no new 5-nodes)

UNDERFULL:
\[
\begin{array}{c}
0 \\
\text{1 child}
\end{array}
\Rightarrow
\begin{array}{c}
\times 1 \\
\text{3-node}
\end{array}
\begin{array}{c}
1 \\
\text{2-node}
\end{array}
\]
\Rightarrow \text{amortized costs} = O(1)
- $\Phi$(initial DS) = $\emptyset$ if we start empty

In (a,b)-trees: need $b > 2a$

Potential examples could also be done with accounting method: coins on 1$'$s or 3/5-nodes.