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
  \( \Rightarrow \) pay \( \Theta(2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{\lceil \log n \rceil}) = \Theta(n) \)
  total to resize table over \( n \) insertions
  \( \Rightarrow \Theta(1) \) amortized cost per insertion
Aggregate method: “just add it up”

\[
\text{total cost of } k \text{ operations} = k
\]

- 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 \(n^*\) = maximum size of set at any time

because c creations, i insertions, d deletions

\[O(c + (i+d)lg n^*) = O(c + ilg n^* + \emptyset d)\]

\(\leq 2i\)

- we’ll tighten to \(O(lg n)\) where

\(n = \text{current set size}, \text{ below}\)
**Accounting method:** “planning ahead for rainy day”
- allow an operation to **store credit** (like bank)
  \[\Rightarrow \text{amortized cost} > \text{actual cost}\]
- allow operations to pay using existing credit
  \[\Rightarrow \text{amortized cost} < \text{actual cost}\]

**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

\[\times \times \times \times \times \times \times \times \times\]

\[\times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \boxtimes \]

\[\times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \boxtimes \]

\[\times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \boxtimes \]

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

**Counterexample:** free deletion in 2-3 trees
- **claim:** \(O(\lg n)\) am. insert, \(\Theta\) am. delete
- **attempt:** put coin worth \(\Theta(\lg n)\) on inserted element
- **trouble:** when deleting that element, \(n\) might be bigger \(\Rightarrow\) 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
- each of these elements charged \( \frac{\Theta(n)}{n/2} = \Theta(1) \) & won't be charged again
- \( \Theta(1) \) 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
  - still half full after any resize
  - 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 \( \Theta(1) \)
- \( \Theta(1) \) amortized per insert & delete

- could do this argument with coins instead, but less intuitive (to me)
Example: free deletion in 2-3 trees
- **claim**: $O(lg n)$ 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 $\rightarrow$ 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 $+ \Delta \Phi$
  $= \Phi(\text{DS after op.}) - \Phi(\text{DS before op.})$
⇒ sum of amortized costs telescopes
  $= \text{sum of actual costs} + \Phi(\text{final DS}) - \Phi(\text{initial DS})$
  $\geq \Phi$ initial balance
- so also need to pay $\Phi(\text{initial DS})$ at start
  - ideally $\Phi$ 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 \( \Phi = c \cdot \# \text{ 1 bits in counter} \)
  
  \( \Rightarrow \Delta \Phi \text{ from increment} = c(-\# \text{ trailing 1 bits} + 1) \)

  \( \Rightarrow \text{amortized cost} = \text{actual cost} + \Delta \Phi \)
  
  \( = \Theta(1 + \# \text{ trailing 1 bits}) + c(-\# \text{ trailing 1 bits} + 1) \)
  
  \( = O(1) \text{ for } c \text{ large enough} \)
- \( \Phi(\text{initial DS}) = \emptyset \text{ assuming we start @000...0} \)
  
  (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 \( 3 \rightarrow 2 \ 2 \)

  \( \Rightarrow \text{amortized} \# \text{ splits} = \text{actual} \# \text{ splits} + \Delta \Phi \)
  
  \( \leq \# \text{ splits} + (1 - \# \text{ splits}) = 1. \)
- \( \Phi(\text{initial DS}) = \emptyset \text{ if we start empty} \)

In B-trees: \( \Phi = \# \text{ nodes with } B \text{ children} \)
In \( (a,b) \)-trees: \( \Phi = \# \text{ nodes with } 6 \text{ children} \)
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}
5 \text{ keys} \\
6 \text{ children}
\end{array}
\quad \Rightarrow 
\begin{array}{c}
5 \text{ k} \\
3\text{-node}
\end{array}
\quad \begin{array}{c}
y\text{s} \\
3\text{-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}
\text{1 child} \\
\text{2-node}
\end{array}
\quad \Rightarrow 
\begin{array}{c}
\times 1 \\
3\text{-node}
\end{array}
$

$\Rightarrow$ 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 1s or 3/5-nodes.