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
- (a,b)-trees

\{ examples of amortized analysis \}

Recall: table doubling
- want hash table to have \( m = \Omega(n) \) slots for \( 1 + \frac{m}{n} = O(1) \) expected performance (with hashing without chaining)
- idea: if \( n \) grows \( \geq m \), double \( m \)
- cost: \( \Theta(m+n) = \Theta(n) \) to build new table
\[ \Rightarrow \Theta(2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{\log n}) = \Theta(n) \]
- total to resize table over \( n \) insertions
\[ \Rightarrow \Theta(1) \text{ amortized cost per insertion} \]
Aggregate method: “just add it up”
- compute total cost of \( k \) operations
- divide by \( k \)
  \( \Rightarrow \) amortized cost per operation
- common only for simple analyses

Amortized bounds:
- operation has amortized cost \( T(n) \)
  if \( k \) operations cost \( \leq k \cdot T(n) \)
- e.g. average time, averaged over the \( k \) ops.
  (as in aggregate method)
- in general free to assign amortized costs
  so long as total cost is preserved
- e.g. can say 2-3 trees achieve
  \( O(1) \) worst-case per create-empty
  \( O(\lg n^*) \) amortized per insert
  \( \emptyset \) amortized per delete (assuming exists)
where \( n^* = \) maximum size of set at any time
because \( c \) creations, \( i \) insertions, \( d \leq i \) deletions
\[
\text{cost } \leq \frac{O(c + (i+d) \lg n^*)}{2i} = O(c + i \lg n^* + \emptyset d)
\]
- we’ll tighten to \( O(\lg n) \) where
  \( n = \) 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

\[ \begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
\text{coin} & & & & & & & \\
\end{array} \rightarrow \frac{n}{2} \rightarrow \]
- use up those coins to pay for \( \Theta(n) \) rebuild

\[ \begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
\text{coin} & & & & & & & \\
\end{array} \]

\[ \Theta(n) - \frac{n}{2} \cdot c \text{ amortized rebuild cost} \]
\[ = 0 \text{ for large enough } c \]
- \( O(1) + c = \Theta(1) \) 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 ⇒ 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 \( n \to 2n \), charge \( \Theta(n) \) cost to \( n/2 \) inserts since last doubling
  \( \Rightarrow \) each of these elements charged \( \Theta(n/2) = \Theta(1) \)
  \& won't be charged again
  \( \Rightarrow \Theta(1) \) 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 \( (m \to m/2) \) at \( \Theta(n) \) cost
  \( \Rightarrow \) still half full after any resize
  \( \Rightarrow \) still \( \geq n/2 \) inserts to charge to on growth
- also \( \geq n \) 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**: $O(\log n)$ 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
  \[ \Rightarrow \text{each insert charged at most once} \]
Potential method: (defining karma)
- define a potential function $\Phi$ mapping
data-structure configuration $\rightarrow$ nonnegative integer
- intuitively measuring "potential energy"
  $\Phi$ = potential high costs in the future
- equivalent to total unused credit
  ($\leq$ unused coins) stored by all past ops.
  $\Phi$ = bank account balance
- nonnegative $\Rightarrow$ never owe the bank
- amortized cost = actual cost $+ \Delta \Phi$
  $= \Phi($DS after op.$) - \Phi($DS before op.$)$
$\Rightarrow$ sum of amortized costs telescopes
= sum of actual costs $+ \Phi($final DS$) - \Phi($initial DS$)$$\geq \Phi$ initial balance
- so also need to pay $\Phi($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 $O(1 + \# \text{ trailing 1 bits})$
  so intuition is that 1 bits are bad
- define $\Phi = c \cdot \# \text{ 1 bits in counter}$
  \[\Rightarrow \Delta \Phi \text{ from increment} = c (1 - \# \text{ trailing 1 bits})\]
  \[\Rightarrow \text{amortized cost} = \text{actual cost} + \Delta \Phi\]
  \[= \Theta(1 + \# \text{ trailing 1 bits}) + c (1 - \# \text{ trailing 1 bits})\]
  \[= \Theta(1) \text{ for } c \text{ large enough}\]
- $\Phi(\text{initial DS}) = \emptyset$ assuming we start @ 000...0
  (necessary for $O(1)$ amortized bound)

Example: insert in (a,b)-trees
- $O(\log n)$ splits in worst case
- but claim only $O(1)$ amortized splits
- what causes splits? nodes overflowing
  - $\Phi = \# \text{ full nodes (nodes with b children)}$
  \[\Rightarrow \Delta \Phi \leq 1 - \# \text{ splits}\]
  add child @ top //each split turns full node $\Rightarrow 2$ nonfull
  \[\Rightarrow \text{amortized \# splits} = \text{actual \# splits} + \Delta \Phi\]
  \[\leq \# \text{ splits} + 1 - \# \text{ splits} = 1\]
- $\Phi(\text{initial DS}) = \emptyset$ if we start empty
Example: insert & delete in $(a,b)$-trees
- claim $O(1)$ amortized splits & merges
- overflows cause splits $\rightarrow$ full nodes
- underflows cause merges $\rightarrow$ bare nodes
- $D = \# \text{ full nodes} + \# \text{ bare nodes}$ $\downarrow \text{ children}$
- insert: $\Delta D \leq 1 - \# \text{ splits}$

Assuming split creates no bare nodes:

- Over
  - b keys
  - b+1 children

- Need $\left\lfloor \frac{b+1}{2} \right\rfloor > a$
  - i.e. $b > 2a$
  - i.e. $b \geq 2a+1$
  - Slightly bigger gap than Lecture 7
  - E.g. $(2,5)$-trees but not 2-3 or 2-3-4

- Delete: $\Delta D \leq 1 - \# \text{ merges}$

Assuming merge creates no bare nodes

- Under
  - a-2 keys
  - a-1 children

- Need $2a-1 < b$
  - i.e. $2a-1 \leq b-1$
  - i.e. $2a \leq b$ $\sim$ fine

$\Rightarrow$ amortized costs $= O(1)$