ADS2 — Week 8: Amortized Analysis (Exam Notes & Solutions)

Metadata

Field	Value
Title	ADS2 — Week 8: Amortized Analysis
Date	2025-11-09
Author	Mads Richardt
Sources used	weekplan-1.png; weekplan-2.png; week8_slides-1..8.png; week8_text.pdf (Amortized Analysis chapter)
Week plan filename	weekplan-1.png, weekplan-2.png

General Methodology and Theory

- **Amortized analysis** studies average cost per operation over a worst-case sequence. Three equivalent tools:
- **Aggregate:** bound total cost T(m) over m operations, report T(m)/m.
- **Accounting:** overcharge cheap ops, store credits on data items to pay for future expensive ops. Credits must never go negative.
- **Potential:** define Φ(state) ≥ 0 with Φ(initial)=0. Amortized cost ĉ_i = c_i + Φ(D_i) − Φ(D_{i−1}). Sum telescopes to T(m) + Φ(D_m) − Φ(D_0).
- **Slides-first anchors:** Stacks with MultiPop, Dynamic Binary Search (binomial arrays), Aggregate/Accounting/Potential templates, and Heap potentials (sum of depths).
- **Design tips:**
- Attach credits/potential to *elements* that migrate upward/downward (e.g., array merges, heap swims/sinks).
- When merging "small into large", each element's set size doubles ⇒ ≤ $\lfloor \log_2 n \rfloor$ migrations.
- For binary counters/array towers, the i-th level flips/merges every 2^i inserts ⇒ total merges O(m).

Notes (slides → key takeaways)

- **Stack + MultiPop (and MultiPush):** Each element is pushed once and popped at most once ⇒ total pops ≤ total pushes. Aggregate gives O(1) amortized.
- **Aggregate/Accounting/Potential equivalence:** choose the view that makes invariants obvious (e.g., coin per level for merges; Φ = sum of coins).
- **Dynamic Ordered Sets via Dynamic Binary Search (DBS):** Maintain sorted arrays A_0…A_{h−1} where |A_i| is either 0 or 2^i; merges mimic a binary counter. SEARCH: binary search each

nonempty array ($O(\log^2 n)$) or with fractional cascading $O(\log n)$. INSERT: amortized $O(\log n)$. DELETE: lazy deletion + rebuild when tombstones $\geq$ ½ fills, amortized $O(\log n)$.

- **Heaps:** With $\Phi = c \cdot$ (sum of node depths), INSERT increases Φ by $O(\log n)$; EXTRACT-MAX decreases Φ by $\Omega(\log n)$. Tuned c yields amortized $O(1)$ extract.

Coverage Table

Weekplan ID	Canonical ID	Title/Label (verbatim)	Assignment Source	Text Source	Status
1	—	MultiPush on a stack with MultiPop: effect on amortized time	weekplan-1.png (p.1)	slides 3–4 (Stack); week8_text.pdf §Stacks	Solved
2	—	Queues with two stacks: ENQUEUE/ DEQUEUE amortized O(1)	weekplan-1.png (p.1)	week8_text.pdf §Queues; slides 1 overview	Solved
3.1	—	Set Union (coloring): standard analysis	weekplan-1.png (p.1)	week8_text.pdf §Union-by-size idea	Solved
3.2	—	Set Union (coloring): amortized analysis	weekplan-1.png (p.1)	slides 1 overview; text §Union	Solved
4.1	—	Binary heap basics, INSERT/ EXTRACT-MAX standard bounds	weekplan-1.png (p.1)	week8_text.pdf §Heaps	Solved
4.2	—	Heap: amortized analysis with potential (INS O(log n), EX-MAX O(1))	weekplan-1.png (p.1)	slides 8 (Potential); text §Heaps	Solved
5.1	—	Sequence T_i: aggregate method	weekplan-1.png (p.1)	slides 1–2 (methods)	Solved
5.2	—	Sequence T_i: accounting method	weekplan-1.png (p.1)	slides 6 (Accounting)	Solved

Weekplan ID	Canonical ID	Title/Label (verbatim)	Assignment Source	Text Source	Status
5.3	—	Sequence T_i: potential method	weekplan-1.png (p.1)	slides 8 (Potential)	Solved
6.1	—	DBS: duplicates during INSERT — issue & algorithm	weekplan-2.png (p.2)	slides 5–7 (DBS); text §DBS	Solved
6.2	—	DBS: DELETE in O(log n) amortized	weekplan-2.png (p.2)	slides 7 (costs)	Solved
7.1	—	Billy the Rabbit: rounds until carrots (n steps away)	weekplan-2.png (p.2)	week8_text.pdf §Exercises	Solved
7.2	—	Billy: total steps before finding carrots (asymptotic)	weekplan-2.png (p.2)	week8_text.pdf §Exercises	Solved
7.3	—	Billy: amortized steps per round (asymptotic)	weekplan-2.png (p.2)	week8_text.pdf §Exercises	Solved
—	Puzzle	Princesses (truth, liar, random): one yes/no question then choose	weekplan-2.png (p.2)	—	Solved (best-possible guarantee)

Solutions

Exercise 1 — MultiPush on a stack with MultiPop

Assignment Source: weekplan-1.png (p.1)
Text Source: slides 3–4 (Stack with MultiPop); week8_text.pdf §Stacks

Claim. Adding **MultiPush(x,k)** (push k copies) preserves **O(1) amortized** time for PUSH, POP, MULTIPOP, and MULTIPUSH over any sequence.

Aggregate proof. Let the sequence contain P individual pushes total (counting all copies from MultiPush) and let Q be the number of pops performed (counting elements popped by MULTIPOP). Each element contributes O(1) once when pushed and O(1) once when popped; an element cannot be

popped more than once. Thus total cost T = O(P + Q) $\leq$ O(2P) since Q $\leq$ P. Over m operations, T = O(P) $\leq$ O(m·$\bar{k}$) but each MultiPush contributes exactly k to P, so **T = O(number of pushed elements)** and **T/m = O(1)**. Hence all operations are **O(1) amortized**.

Accounting view. Charge 2 to each individual push (including copies). Spend 1 now; store 1 credit on the pushed copy. POP/MULTIPOP consume the stored credit. MULTIPUSH(x,k) simply performs k charged pushes. No deficit.

Potential view. Φ = (# elements on stack). PUSH increases Φ by 1; POP/MULTIPOP decrease by number popped. Amortized cost $\hat{c}$ = c + ΔΦ $\leq$ 2 for push and $\leq$ 0 for pops, giving O(1).

Transfer Pattern. Dynamic bag where each item is created once and destroyed once (stack/queue/multiset). Recognition cues: "batch insert/delete", "each element removed at most once", "no duplication after insertion". Mapping: attach 1 credit per item upon insertion.

✅**Answer:** All stack ops remain **O(1) amortized** under MultiPush.

Exercise 2 — Queues with two stacks (amortized O(1))

Assignment Source: weekplan-1.png (p.1)
Text Source: week8_text.pdf §Queues; slides overview

Method. Maintain two stacks: **IN** (push on enqueue) and **OUT** (pop on dequeue).

```
Algorithm: queue_with_two_stacks
Input: stream of enqueue/dequeue operations
Output: queue semantics

init empty stacks IN, OUT

enqueue(x): push(IN, x)

dequeue():
  if OUT empty:
    while IN not empty: push(OUT, pop(IN))
  return pop(OUT)
// Time: O(1) amortized per op; Space: O(n)
```

Amortized proof (accounting). Charge 3 to each enqueue: 1 to push onto IN and 2 credits stored on the element to pay for the single transfer to OUT and the eventual pop from OUT. Each element is moved **at most once** between stacks ⇒ total T = O(#elements). Dequeues that trigger a refill are paid by stored credits.

Transfer Pattern. Two-phase buffer: inexpensive append, expensive flush, but each element flushed once. Cues: "reverse order by bulk move", "flush when empty".

✅**Answer:** ENQUEUE and DEQUEUE are **O(1) amortized**.

Exercise 3.1 — Set Union (coloring), standard analysis

Assignment Source: weekplan-1.png (p.1)
Text Source: week8_text.pdf §Union-by-size

Model. Each set has a color id. UNION(A,B): recolor all elements in the *smaller* set to the color of the larger; ties arbitrary. SAMESET(x,y): compare colors.

Standard (worst-case) cost. UNION may recolor $\Theta(n)$ elements; SAMESET is $O(1)$; INIT is $O(n)$. Over a sequence of u unions, worst case is $\Theta(u \cdot n)$ by adversarial alternating sizes (standard analysis ignores the size-doubling effect).

✅**Answer:** Worst-case per UNION = $\Theta(n)$; SAMESET = $O(1)$; INIT = $\Theta(n)$.

Exercise 3.2 — Set Union (coloring), amortized analysis

Assignment Source: weekplan-1.png (p.1)
Text Source: slides overview; text §Union

Key invariant. When recoloring, the smaller set moves into the larger. Each element's set size at least doubles whenever it is recolored.

Aggregate bound. An element can be recolored at most $\lfloor \log_2 n \rfloor$ times ($1 \to 2 \to 4 \to \ldots \leq n$). Thus total recolorings $\leq n\lfloor \log_2 n \rfloor$. Over u UNIONs, $T \leq O(n \log n + u)$ and amortized recolorings per UNION $\leq O(\log n)$ in the worst mix, while SAMESET remains $O(1)$.

Accounting. Place 1 credit on each element of a set per unit of current set size "level". Recoloring an element consumes one level credit; as the set doubles, new credits are assigned. No deficits.

Transfer Pattern. "Small-to-large" merges (DSU/union-by-size/rank). Cues: repeated merges, size-aware choice. Mapping: attach log levels to elements; each merge reduces a level counter.

✅**Answer:** Total recolorings $\leq n \log n \Rightarrow$ amortized recolorings **O(log n)**; SAMESET stays **O(1)**.

Exercise 4.1 — Binary heaps: recap

Assignment Source: weekplan-1.png (p.1)
Text Source: week8_text.pdf §Heaps

- Structure: array-based complete binary tree, parent at $\lfloor i/2 \rfloor$, children at 2i, 2i+1.
- **INSERT(x):** place at next leaf, **swim** up while parent < x. Worst case O(log n).
- **EXTRACT-MAX():** swap root with last leaf, remove leaf, **sink** root down. Worst case O(log n).
- Heap property maintained after each swap.

✅**Answer:** Standard bounds: both operations **O(log n)** worst-case.

Exercise 4.2 — Heap: amortized analysis (Φ via depths)

Assignment Source: weekplan-1.png (p.1)
Text Source: slides 8 (Potential); text §Heaps

Potential. Let Φ(D) = c · (sum of depths of all nodes in the heap), depth(root)=0. Choose constant c ≥ 1 so one swap changes Φ by at least the work saved.

- **INSERT.** Actual cost c_ins = #swims ≤ h = ⌊log₂(n+1)⌋. Each swim decreases depth by 1 ⇒ ΔΦ = −c per swap, but the inserted leaf initially increases Φ by its initial depth $d_0 \le h$. Net ĉ_ins = c_ins + ΔΦ ≤ h − c·h + c·h = O(log n).
- **EXTRACT-MAX.** Actual cost c_ext = #sinks ≤ h. Removing last leaf (moved to root) decreases Φ by its old depth (≈ h). Each sink increases depth by 1 ⇒ ΔΦ ≤ +c per level, but the removal contributes −c·h. With c large enough (e.g., c=2), ĉ_ext ≤ c_ext − c·h + c·h/2 ≤ O(1).

Rigorous tuning: pick c so that each **sink level**'s $+c$ is offset by the removed leaf's depth budget (≈ h). Summed over a sequence, Φ never negative.

Transfer Pattern. Tree of bounded height with symmetric up/down adjustments; potential = sum of node depths.

✅**Answer:** INSERT **O(log n)** amortized; EXTRACT-MAX **O(1)** amortized (with Φ as above).

Exercise 5.1 — Sequence T_i (aggregate)

Assignment Source: weekplan-1.png (p.1)
Text Source: slides 1–2

Let

$$ T_i = \begin{cases} 2i & \text{if } i \text{ is a power of two},\\
1 & \text{otherwise.} \end{cases} $$

Over m operations, expensive indices are i = 1,2,4,8,\dots,2^{\lfloor\log_2 m\rfloor}. Their total is $\sum_{k=0}^{\lfloor \log_2 m \rfloor} 2 \cdot 2^k = 2(2^{\lfloor \log_2 m \rfloor + 1} - 1) < 4m$. All other (at most m) operations cost 1 each. Hence total $T(m) < 5m$ and amortized cost is **O(1)**.

✅**Answer:** Aggregate ⇒ **O(1)** amortized.

Exercise 5.2 — Sequence T_i (accounting)

Assignment Source: weekplan-1.png (p.1)
Text Source: slides 6

Charge 5 to every operation. For non-power-of-two i, spend 1 and bank 4. At i=2^k, actual cost is 2·2^k; the preceding block of 2^k−1 operations contributed ≥ 4(2^k−1) > 2·2^k − 4 credits, and earlier blocks cover the remaining constant slack. Credits never negative.

✅**Answer:** With charge 5, bank covers spikes ⇒ **O(1)** amortized.

Exercise 5.3 — Sequence T_i (potential)

Assignment Source: weekplan-1.png (p.1)
Text Source: slides 8

Let i in binary have t trailing zeros (tz(i)). Define Φ after i operations as $\Phi_i = 2 \cdot (2^{\mathrm{tz}(i)} - 1)$. Bit-flip view: moving from i−1 to i flips t low bits from 1→0 and one bit from 0→1.

- For non-power-of-two (t = tz(i) < k): actual cost 1, potential change $\Delta\Phi \in [-2, +2] \Rightarrow \hat{c}$ = O(1).
- For i = 2^k: actual cost 2·2^k, but tz(i)=k and $\Delta\Phi = 2(2^k - 1) - (2(2^{\mathrm{tz}(i-1)} - 1)) \leq 2 \cdot 2^k$. Choose constants so $\hat{c} \leq$ constant.

(Any equivalent binary-counter potential suffices.)

✅**Answer: O(1)** amortized.

Exercise 6.1 — Dynamic Binary Search (DBS): duplicates on INSERT

Assignment Source: weekplan-2.png (p.2)
Text Source: slides 5–7 (DBS); week8_text.pdf §DBS

Issue. DBS keeps each nonempty level as a distinct **sorted array** of size 2^i. Duplicate keys across levels waste space and complicate DELETE.

Efficient fix. Store **(key, count)** pairs in arrays. During binary search, compare keys; INSERT(x): create (x,1) at A_0 and cascade merges as usual, but when merging arrays, **deduplicate runs**: if the same key appears in both inputs, output one (key, count_a+count_b). SEARCH unchanged.

Complexity. Merging still linear in total run length; run-compression is linear in the number of duplicate occurrences. Asymptotic costs unchanged: INSERT amortized O(log n); SEARCH O(log n) (with cascading) as assumed.

Transfer Pattern. From set to multiset: replace element unit weight with multiplicity counters; maintain invariant at merge points.

✅**Answer:** Use **(key,count)** with deduplication during merges; INSERT remains amortized **O(log n)**; SEARCH stays **O(log n)** under the given assumption.

Exercise 6.2 — DBS: DELETE in O(log n) amortized

Assignment Source: weekplan-2.png (p.2)
Text Source: slides 7 (Costs)

Strategy (lazy deletion + rebuild).

1. **SEARCH x** across O(log n) arrays to find a (key,count). If count>1: decrement and stop (O(log n)).
2. If count=1: mark as a **tombstone**. Maintain invariant per array A_i: when tombstones ≥ ½ of live items, **rebuild** A_i by compacting to a new sorted array (and possibly cascading merges downward to restore the 0/2^i fullness invariant).

Amortized analysis (accounting). Attach 2 credits to each newly inserted *live* item in A_i: 1 to pay for its eventual deletion mark, 1 to pay for its share of a future rebuild. Each element is compacted at most once before leaving the level; rebuild work O(2^i) is charged to Ω(2^i) tombstones.

Complexity. SEARCH O(log n). DELETE worst-case O(2^i) inside a rebuild, but **amortized O(log n)** over sequences.

Transfer Pattern. Maintain density by **lazy delete** + **repack when half empty**; classic in hash tables and ordered arrays.

✅**Answer:** DELETE can be implemented in **O(log n) amortized** using tombstones and thresholded rebuilds.

Exercise 7.1 — Billy the Rabbit: rounds

Assignment Source: weekplan-2.png (p.2)
Text Source: week8_text.pdf §Exercises

Round r moves forward 2^{r−1} steps before returning. He reaches the carrots in the first r with 2^{r−1} ≥ n, i.e., $r = \lceil \log_2 n \rceil$.

✅**Answer:** $\lceil \log_2 n \rceil$ rounds.

Exercise 7.2 — Billy: total steps (asymptotic)

Assignment Source: weekplan-2.png (p.2)
Text Source: week8_text.pdf §Exercises

Total before the final round: $2(1 + 2 + \cdots + 2^{r-2}) = 2^r - 2$. In the final round he walks exactly n steps forward and stops upon finding the carrots. Since $2^{r-2} < n \leq 2^{r-1}$, we have $2^r < 4n$. Thus total steps $< (4n - 2) + n = O(n)$ and also $> n$.

✅**Answer: Θ(n)** total steps.

Exercise 7.3 — Billy: amortized steps per round

Assignment Source: weekplan-2.png (p.2)
Text Source: week8_text.pdf §Exercises

With r = ⌈log₂ n⌉ rounds and Θ(n) total steps, the amortized steps/round are $\Theta\left(\frac{n}{\log n}\right)$.

✅**Answer: Θ(n / log n)**.

Puzzle — Princesses (truth, liar, random)

Assignment Source: weekplan-2.png (p.2)

Goal. Ask one yes/no question to one daughter, then choose whom to marry, preferring an extreme (oldest truth-teller or youngest liar) and avoiding the middle (random).

Fact (impossibility). With exactly one yes/no question to a possibly random respondent, it is **impossible** to deterministically identify an extreme in all cases: if you question the random daughter, Nature can force any answer, making any deterministic selection rule fail on some labeling.

Best-possible strategy (avoid asking the one you might pick). Pick two daughters B and C in front of you; ask A (the third):

> "If I asked you 'Is B the middle?', would you say yes?"

- If A is an extreme, this self-reference neutralizes truth/lie and yields a *reliable* answer to whether B is middle. Then **marry the non-middle among B and C** accordingly.
- If A is the middle, the answer is unreliable, but you still **never marry A**. Your choice between B and C succeeds with probability ≥ 1/2; no one-question strategy can guarantee 1.

✅**Answer:** No deterministic guarantee exists with one question if you might query the random sister. The above question maximizes reliability; never choose the respondent.

Summary

- **Stacks (with MultiPush/MultiPop):** O(1) amortized via "pushed once, popped once".
- **Queues via two stacks:** O(1) amortized; each element moves at most once.
- **Set Union (coloring small→large):** total recolorings ≤ n log n; SAMESET O(1).
- **Heaps:** With Φ=sum of depths, EXTRACT-MAX becomes amortized O(1), INSERT stays O(log n).
- **DBS:** INSERT O(log n) amortized; handle duplicates with (key,count); DELETE O(log n) amortized via lazy deletion + rebuild.
- **Spike sequence T_i:** O(1) amortized by aggregate/accounting/potential.
- **Billy:** rounds ⌈log₂ n⌉; total steps Θ(n); amortized per round Θ(n/log n).

Notation blurb. n = number of elements; m = number of operations; Φ = potential; "amortized" means per-operation cost averaged over sequences in the worst case.

Amortized Analysis

- Amortized Analysis
- Aggregate Method
- Accounting Method
- Potential Method

Philip Bille

Amortized Analysis

- Amortized Analysis
- Aggregate Method
- Accounting Method
- Potential Method

Amortized Analysis

- Idea.
  - Analyse of data structure operations whose time complexity vary over long sequences of operations.
  - Standard analysis may be too pessimistic, amortized analysis gives a more refined analysis.
- Definition.
  - Let T(m) be the time complexity for a worst-case sequence of m operations on some data structure D. The amortized time complexity of an operation is T(m)/m.

Amortized Analysis

- Applications.
  - Algorithms that use data structures. Total time is important, not individual operations.
    - Examples: Minimum spanning tree algorithms, Dijkstra's shortest path algorithm, ...
  - Simple and practical. Often simpler and faster than worst-case versions.
- Goals.
  - Techniques for showing bounds on amortized complexity.
  - New data structures and data structure design techniques.

Amortized Analysis

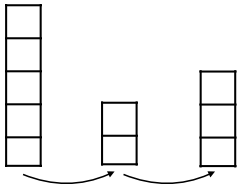
- Amortized Analysis
- Aggregate Method
- Accounting Method
- Potential Method

Aggregate Method

- Aggregate method.
  - Identify a worst-case sequence of m operations.
  - Compute the total time complexity T(m) of the sequence.
  - Compute T(m)/m as the amortized time complexity.

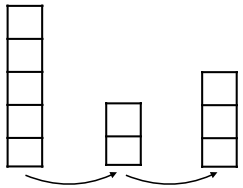
Stack

- Stack with MultiPop. Maintain a sequence (stack) S supporting the following operations:
  - PUSH(x): add x to S.
  - MULTIPOP(k): remove and return the k most recently added elements in S.
  - POP() = MULTIPOP(1).
- Assume POP/MULTIPOP always has enough elements on stack.



Stack

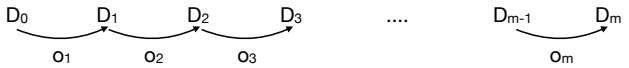
- Stack with MultiPop. Maintain a stack S supporting PUSH(x) and MULTIPOP(k).
- Consider a sequence of m operations.
- Standard analysis.
  - PUSH in O(1) time.
  - MultiPop in O(k) = O(m) time.
- Amortized analysis.
  - An element can only be POPped once for each time it is PUSHed.
  - ⇒ Total number of POPs is ≤ total number of PUSHes ≤ m.
  - ⇒ Total time is O(m)
  - ⇒ Amortized time of MULTIPOP is O(1).



Amortized Analysis

- Amortized Analysis
- Aggregate Method
- Accounting Method
- Potential Method

Amortized Cost



- Actual cost.
  - c_i = actual cost of operation i = time complexity of operation i.
- Amortized cost.
  - Assign a cost $\hat{c}_i$ to operation i.
  - $\hat{c}_i$ is an amortized cost for D if for all sequences $O = o_1, ..., o_m$.
  - $\sum_{i=1}^{m} \hat{c}_i \geq \sum_{i=1}^{m} c_i$
  - If $\hat{c}_i$ is an amortized cost $\Rightarrow$ $\hat{c}_i$ is also amortized running time.
- Challenge. How to find a good amortized cost?

Accounting Method

- Accounting method.
  - Assign a cost $\hat{c}_i$ to each operation.
    - $\hat{c}_i > c_i$: store the difference as credits to objects in the data structure.
    - $\hat{c}_i < c_i$: use the stored credits to pay for operation.
  - Show that cost is an amortized cost $\Rightarrow$ amortized cost is amortized running time.

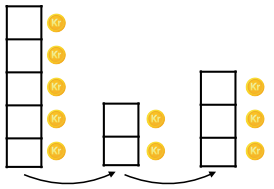
- Challenge. How to find a good credit scheme?

Stack

- Stack with MultiPop. Maintain a stack S supporting PUSH(x) and MULTIPOP(k).
- Costs.
  - A credit pays for a PUSH or POP of an element.
  - PUSH(x): use 1 credit to PUSH element. Assign 1 credit to element.
  - MULTIPOP(k): use stored credits on top k elements to pay.

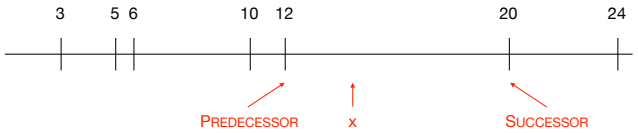
	Actual Cost	Assigned Cost
PUSH	1	2
MULTIPOP	k	0

- Amortized analysis.
  - Always enough credits to pay for POP/MULTIPOP.
  - $\Rightarrow \sum_{i=1}^{m} \hat{c}_i \geq \sum_{i=1}^{m} c_i$
  - $\Rightarrow$ Assigned cost are amortized costs.
  - $\Rightarrow$ Amortized running time of MULTIPOP is O(1).



Dynamic Ordered Sets

- Dynamic Ordered Sets. Maintain a dynamic set S of numbers supporting the following operations.
  - SEARCH(x): return true if $x \in S$.
  - PREDECESSOR(x): return the largest element in S that is $\leq x$.
  - SUCCESSOR(x): return the smallest element in S that is $\geq x$.
  - INSERT(x): add x to S.
  - DELETE(x): remove x from S.



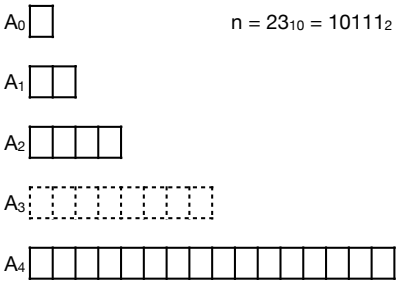
Dynamic Ordered Sets

- Applications.
  - Dictionaries, indexes, databases, filesystem, ...

- What solutions do we know?

Dynamic Binary Search

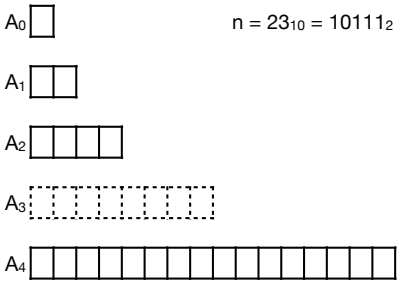
- Dynamic binary search.
  - Maintain arrays $A_0, A_1, A_2, \ldots, A_{h-1}$. A^i has size 2^i and $h \approx \log(n)$.
  - Each array is either full or empty.
  - Full arrays correspond to the binary representation of $n = |S|$.
  - Each full array stores elements from S in sorted order.



$n = 23_{10} = 10111_2$

Dynamic Binary Search

- SEARCH(x): Do binary search in each array.

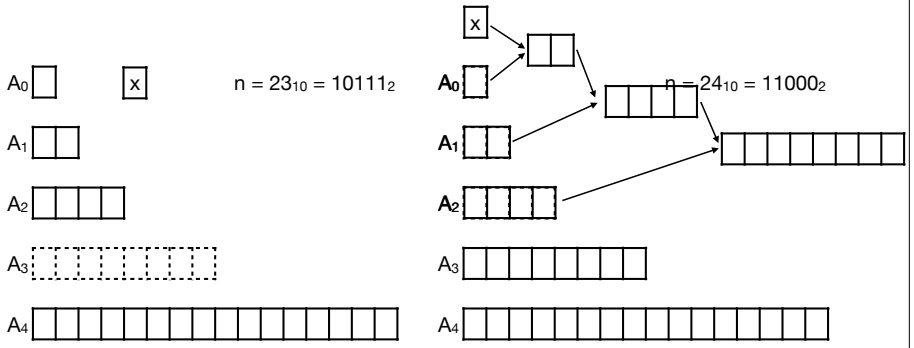


$n = 23_{10} = 10111_2$

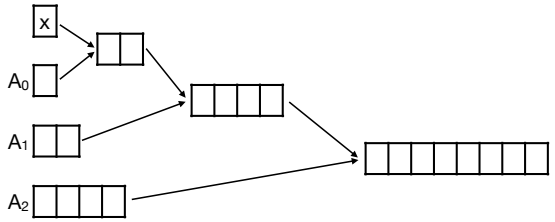
- Time. $O(\log^2 n)$.
- Similar idea for PREDECESSOR and SUCCESSOR.

Dynamic Binary Search

- INSERT(x):
  - If A_0 is empty, fill it with x and stop.
  - Create singleton array containing x. Merge arrays pairwise top-down until we fill empty array.
  - Corresponds to incrementing a binary number.



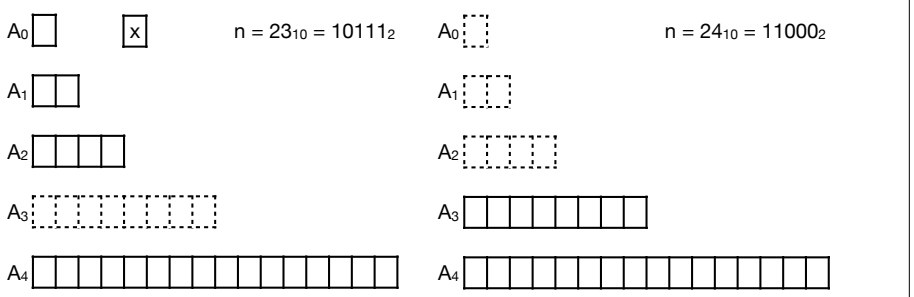
Dynamic Binary Search



- Standard analysis.
  - Create singleton array: O(1) time.
  - Merge arrays $A_0,\ldots,A_{k-1}$: $O(2^0 + 2^1 + \ldots + 2^k) = O(2^{k+1} - 1) = O(n)$ time.
  - ⇒ O(n) time in total.

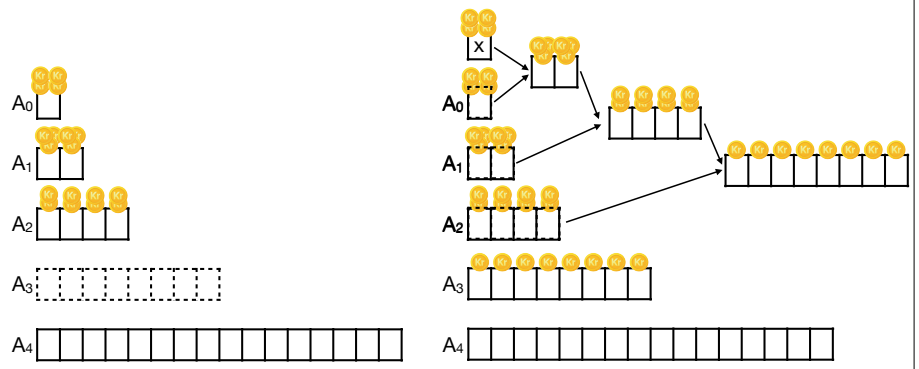
Dynamic Binary Search

- Observation.
  - Most insertions are fast.
  - Elements always start at top and move down monotonically.



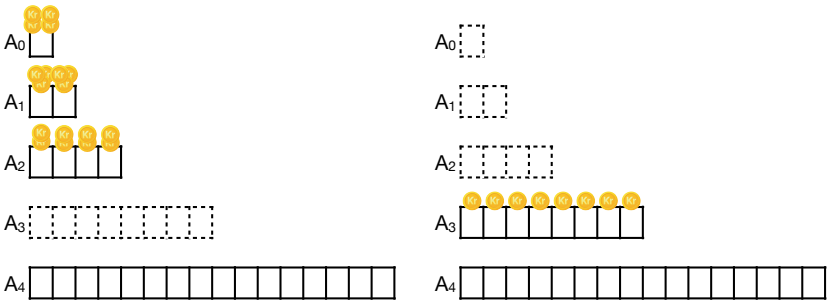
Dynamic Binary Search

- Costs.
  - A credit pays for a single element being part of a merge.
  - INSERT(x): Assign h-1 credits to element. Use credits to pay for merges.



Dynamic Binary Search

- Amortized analysis
  - Enough credits to pay for merges ⇒ assigned cost are amortized costs.
  - ⇒ Amortized running time of INSERT is h-1 = O(log n).



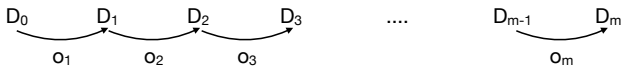
Dynamic Binary Search

- Dynamic binary search.
  - SEARCH, PREDECESSOR, and SUCCESSOR in $O(\log^2 n)$ time.
  - INSERT and DELETE in O(log n) amortized time.
- With fractional cascading technique can do SEARCH, PREDECESSOR, and SUCCESSOR in O(log n) time.
- Key component in database indexes called a log-structured merge.
- General idea for transforming static data structures into dynamic data structures.

Amortized Analysis

- Amortized Analysis
- Aggregate Method
- Accounting Method
- Potential Method

Potential Method



- Potential function.
  - Define a potential function $\Phi(D)$ that maps (the state of) data structure D to a real value.
  - Require that $\Phi(D_i) \geq 0$ for all i and $\Phi(D_0) = 0$.
  - Assign cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.
  - Corresponds to potential energy.
- Amortized cost.
  - $\hat{c}_i$ is an amortized cost:
  - $$\sum_{i=1}^{m} \hat{c}_i = \sum_{i=1}^{m} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right) = \sum_{i=1}^{m} c_i + \Phi(D_m) - \Phi(D_0) \geq \sum_{i=1}^{m} c_i$$

Potential Method

- Potential method.
  - Define a potential function $\Phi(D)$.
  - Compute the corresponding amortized cost ⇒ amortized cost is amortized running time.

- Challenge. How to find a good potential function?

Stack

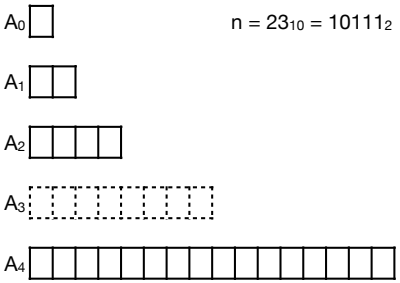
- Stack with MultiPop. Maintain a stack S supporting PUSH(x) and MULTIPOP(k).

- Potential function.
  - Define $\Phi(D_i)$ = number of elements on stack.
  - $\Phi(D_i) \geq 0$ for all i and $\Phi(D_0) = 0$.

- Amortized analysis.
  - PUSH(x): $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.
  - MULTIPOP(k): $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k - k = 0$.
  - ⇒ amortized running time of MULTIPOP is O(1).

Dynamic Binary Search

- Dynamic binary search.
  - Maintain arrays $A_0, A_1, A_2, \ldots, A_{h-1}$. A^i has size 2^i and $h \approx \log(n)$.
  - Each array is either full or empty.
  - Full arrays correspond to the binary representation of $n = |S|$.
  - Each full array stores elements from S in sorted order.

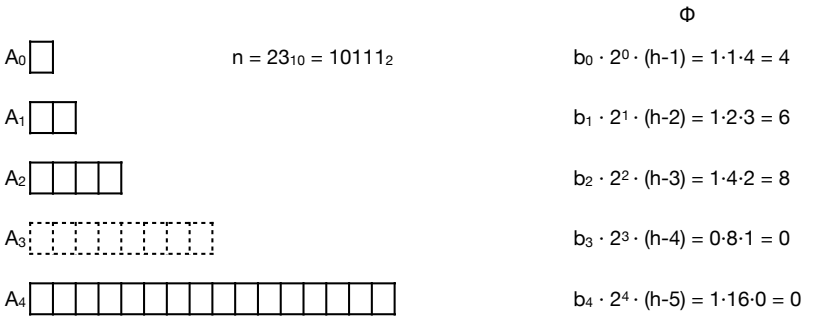


Dynamic Binary Search

- Potential function. Let $b_{h-1}b_{h-2}\ldots b_0$ be the binary representation of n and define

$$\Phi(D) = \sum_{j=0}^{h-1} b_j \cdot 2^j \cdot ((h-1)-j)$$

- Intuition. Individual elements have potential corresponding to their height.

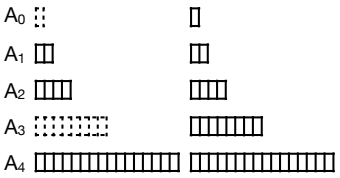


Dynamic Binary Search

- Amortized analysis (case 1). No merges.
  - Actual cost: $c_i = 1$.
  - Increase in potential:
  - $\Phi(D_i) - \Phi(D_{i-1}) = 2^0(h-1) = h-1$

$$\Phi(D) = \sum_{j=0}^{h-1} b_j \cdot 2^j \cdot ((h-1)-j)$$

- Amortized cost.
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + h - 1 = h = O(\log n)$



Dynamic Binary Search

- Amortized analysis (case 2). Merge arrays $A_0,\ldots,A_{k-1}$.
  - Actual cost: $c_i = \sum_{j=0}^{k} 2^j = 2^{k+1} - 1$.
  - Decrease in potential:

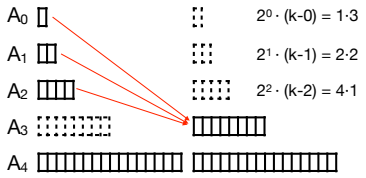
$$\sum_{j=0}^{k-1} 2^j(k-j) = k \cdot \sum_{j=0}^{k-1} 2^j - \sum_{j=0}^{k-1} j \cdot 2^j$$

$$= k \cdot (2^k - 1) - \left((k-2)2^k + 2\right)$$

$$= 2^{k+1} - k - 2$$

- Amortized cost.
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - $= 2^{k+1} - 1 - (2^{k+1} - k - 2) = k + 1 = O(\log n)$

$$\Phi(D) = \sum_{j=0}^{h-1} b_j \cdot 2^j \cdot ((h-1)-j)$$



Dynamic Binary Search

- ⇒ Amortized running time of INSERT is O(log n) in both cases.

- Dynamic binary search.
  - SEARCH, PREDECESSOR, and SUCCESSOR in $O(\log^2 n)$ time.
  - INSERT and DELETE in O(log n) amortized time.

Amortized Analysis

- Amortized Analysis
- Aggregate Method
- Accounting Method
- Potential Method

Reading

See webpage.

Exercises

1 [w] **MultiPush** Consider the stack with MULTIPOP and add the following additional operation:

- MULTIPUSH(x, k): add k duplicates of x onto the stack S.

Determine how this affects the amortized running time of (all of) the stack operations. Is the amortized running time still constant?

2 [w] **Queues** Show how to implement a queue using two stacks (and no other data structures) such that both the ENQUEUE and DEQUEUE operations take amortized constant time.

3 **Set Union** The *set union problem* is to maintain a dynamic family of sets supporting the following operations.

- INIT(n): Create n singleton sets.
- UNION(A, B): Create the union $C = A \cup B$ and return a pointer to C. The sets A and B are destroyed.
- SAMESET(x, y): Return true, if x and y are in the same set, and false otherwise.

Note that the problem is closely related to the union find problem (how exactly?). Consider the following solution idea. For INIT(n) we create the n singleton sets and assign each element a distinct *color*. For UNION(A, B) we re-color the elements in the smaller set with the color of the larger set (breaking ties arbitrarily). Finally, for SAMESET(x, y), we compare the colors of x and y and return true if and only if they match.

Solve the following exercises.

3.1 Do a standard analysis of the running time of the data structure for all of the operations.

3.2 Do an amortized analysis of the data structure for the relevant operations. *Hint:* Give a bound on the number of times an element can be recolored.

4 **Binary Heaps** Consider a binary (max-)heap with n elements.

4.1 [w] Recall the basic structure of a heap and how the INSERT and EXTRACT-MAX operations work. Recall the standard analysis for the running time of INSERT and EXTRACT-MAX.

4.2 Do an amortized analysis to show that the amortized running time of INSERT is $O(\log n)$ and the amortized running time of EXTRACT-MAX is constant. Use the potential method. *Hint:* Use the depth of the nodes as part of your potential function.

5 **Amortized Analysis Methods** Consider a data structure where the running time T_i of the ith operation is:

$$T_i = \begin{cases} 2i & \text{if } i \text{ is a power of 2,} \\ 1 & \text{otherwise.} \end{cases}$$

Solve the following exercises.

5.1 Analyze the amortized running time of the operation using the aggregate method.

5.2 Analyze the amortized running time of the operation using the accounting method.

5.3 [$*$] Analyze the amortized running time of the operation using the potential method.

6 Dynamic Ordered Sets Consider the dynamic binary search data structure. Assume that we can support SEARCH in $O(\log n)$ time. Solve the following exercises.

6.1 In the lecture, we ignored how to handle *duplicates* during INSERT operations, i.e., inserting the same value multiple times. Explain why this is an issue and give an efficient algorithm to handle duplicates.

6.2 [$*$] Show how to implement the DELETE operation in $O(\log n)$ amortized running time.

7 Billy the Rabbit (Exam 2017) The rabbit Billy lives with his family in a rabbit hole. The rabbit Billy is very shy, and he loves carrots. He knows that there are carrots somewhere on the path going from the rabbit hole into the woods. But he does not know how far away the carrots are.

Billy is afraid to leave home alone, so he comes up with the following strategy. In the first round, he takes 1 step and then goes back to the hole. In the next round, he takes 2 steps and then goes back; in the third round, he takes 4 steps, and so on. That is, in each round he takes twice as many steps as in the previous round. Assume the carrots are n steps away. Solve the following exercises.

7.1 How many rounds does it take before Billy finds the carrots?

7.2 What is the total number of steps Billy takes before he finds the carrots? (An asymptotic answer is satisfactory.) Explain your answer.

7.3 What is the amortized number of steps that Billy takes in a round? (An asymptotic answer is satisfactory.) Explain your answer.

Puzzle of the week: Princesses You are a young Prince from the country Algo. The King in the neighboring country Logic has 3 daughters. The oldest one always tells the truth, the youngest one always lies, and the middle one sometimes lies, sometimes tells the truth.

You want to marry either the oldest one or the youngest one (since you know she *always* lies that is as good as the one always telling the truth). The only one you don't want to marry is the middle one.

The king is a sneaky man, and he tells you, that you can ask *one* of the daughters *one* question. The question should be one with a yes/no answer. After that you have to choose which one to marry. They all look alike, so it is not possible for you to determine which one is which by looking at them.

What question should you ask, and which one should you then pick?