# 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

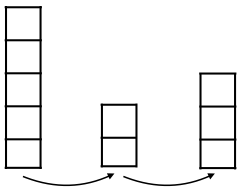
- Amortized Analysis
- Aggregate Method
- Accounting Method
- Potential Method

## 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.



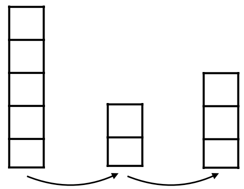
## Stack

- 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

$$D_0 \xrightarrow{o_1} D_1 \xrightarrow{o_2} D_2 \xrightarrow{o_3} D_3 \quad .... \quad D_{m-1} \xrightarrow{o_m} D_m$$

- 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 ⇒ $\hat{c}_i$ 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 ⇒ amortized cost is amortized running time.

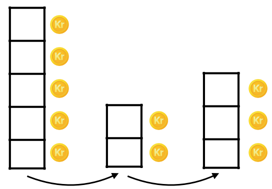

- Challenge. How to find a good credit scheme?

## 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.

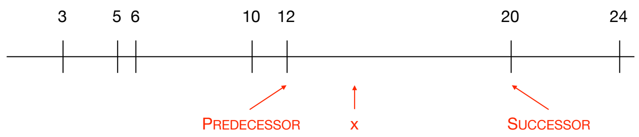
| | Actual Cost | Assigned Cost |
|---|---|---|
| PUSH | 1 | 2 |
| MULTIPOP | k | 0 |

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

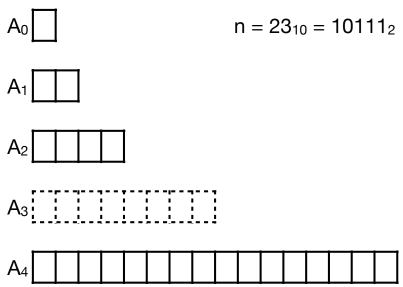



## Dynamic Ordered Sets

- Applications.
  - Dictionaries, indexes, databases, filesystem, ...
- What solutions do we know?

## Dynamic Binary Search

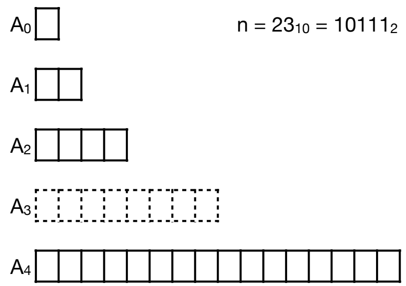
- Dynamic binary search.
  - Maintain arrays $A_0, A_1, A_2, \ldots, A_{h-1}$. $A^i$ has size $2^i$ and $h \approx \log(n)$.
  - Each array is either full or empty.
  - Full arrays correspond to the binary representation of $n = |S|$.
  - Each full array stores elements from S in sorted order.




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

## Dynamic Binary Search

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

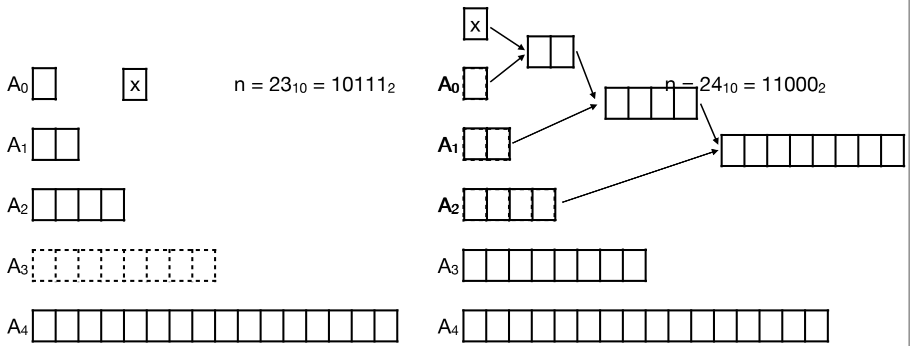



$n = 23_{10} = 10111_2$

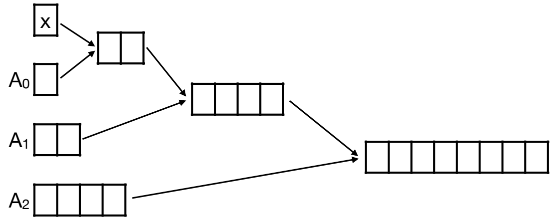
- Time. $O(\log^2 n)$.
- Similar idea for PREDECESSOR and SUCCESSOR.

## 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.



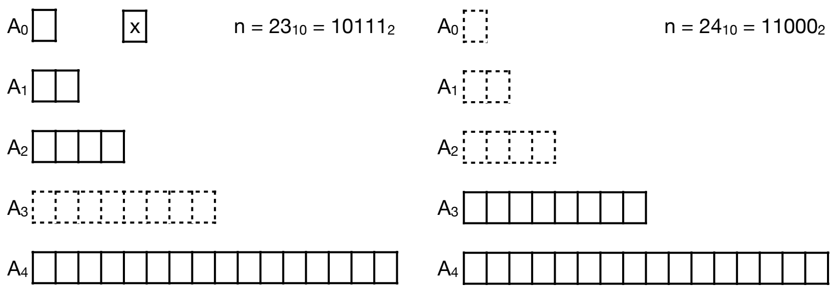

## Dynamic Binary Search




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

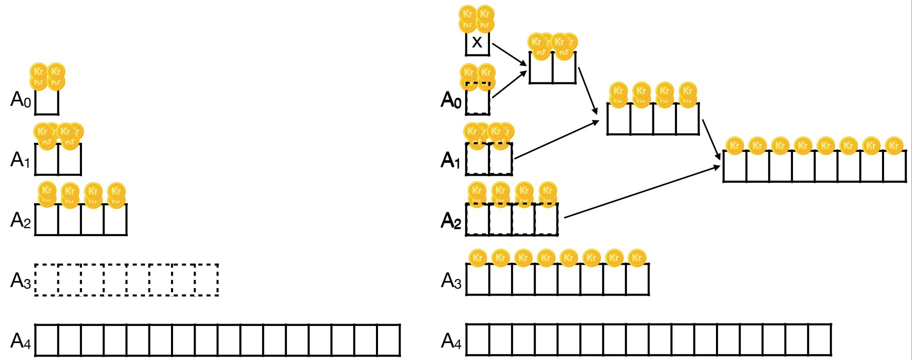
## Dynamic Binary Search

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



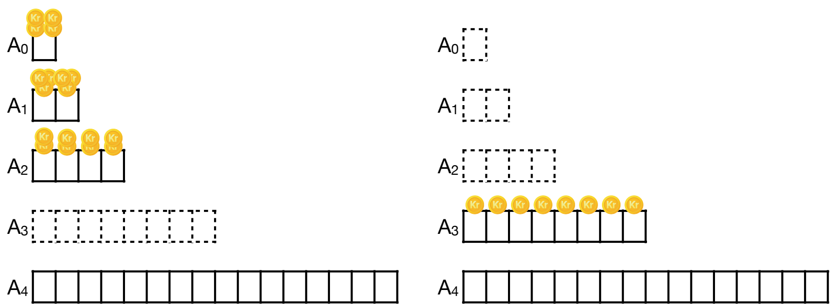

## Dynamic Binary Search

- 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

$$D_0 \xrightarrow{o_1} D_1 \xrightarrow{o_2} D_2 \xrightarrow{o_3} D_3 \quad .... \quad D_{m-1} \xrightarrow{o_m} D_m$$

- Potential function.
  - Define a potential function Φ(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

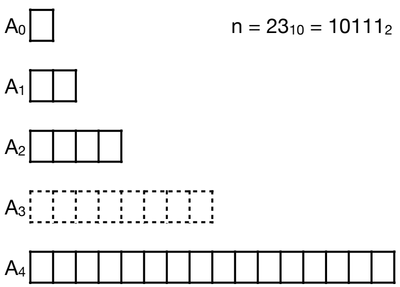
- Potential method.
  - Define a potential function $\Phi(D)$.
  - Compute the corresponding amortized cost ⇒ amortized cost is amortized running time.
- Challenge. How to find a good potential function?

## 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.



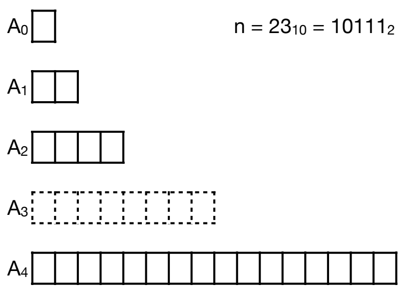

$n = 23_{10} = 10111_2$

## 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.




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

$\Phi$

$b_0 \cdot 2^0 \cdot (h-1) = 1 \cdot 1 \cdot 4 = 4$

$b_1 \cdot 2^1 \cdot (h-2) = 1 \cdot 2 \cdot 3 = 6$

$b_2 \cdot 2^2 \cdot (h-3) = 1 \cdot 4 \cdot 2 = 8$

$b_3 \cdot 2^3 \cdot (h-4) = 0 \cdot 8 \cdot 1 = 0$

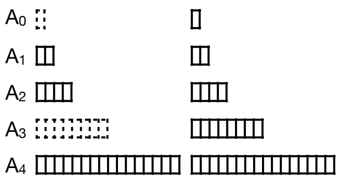
$b_4 \cdot 2^4 \cdot (h-5) = 1 \cdot 16 \cdot 0 = 0$

## 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:

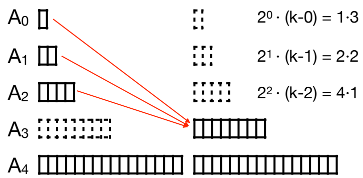
$$\sum_{j=0}^{k-1} 2^j(k-j) = k \cdot \sum_{j=0}^{k-1} 2^j - \sum_{j=0}^{k-1} j \cdot 2^j$$

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

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

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

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




## 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