

Amortized Analysis

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

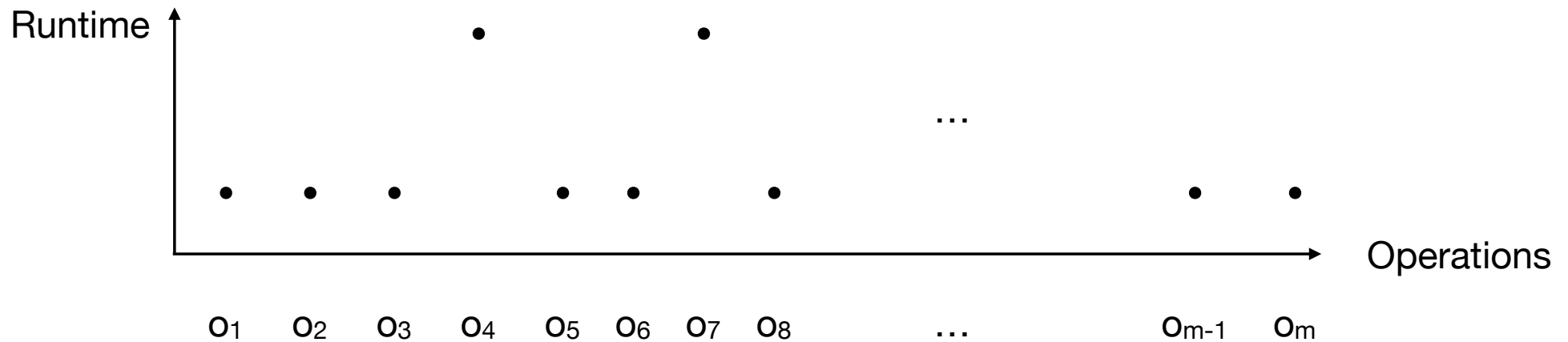
Philip Bille

Amortized Analysis

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

Amortized Analysis

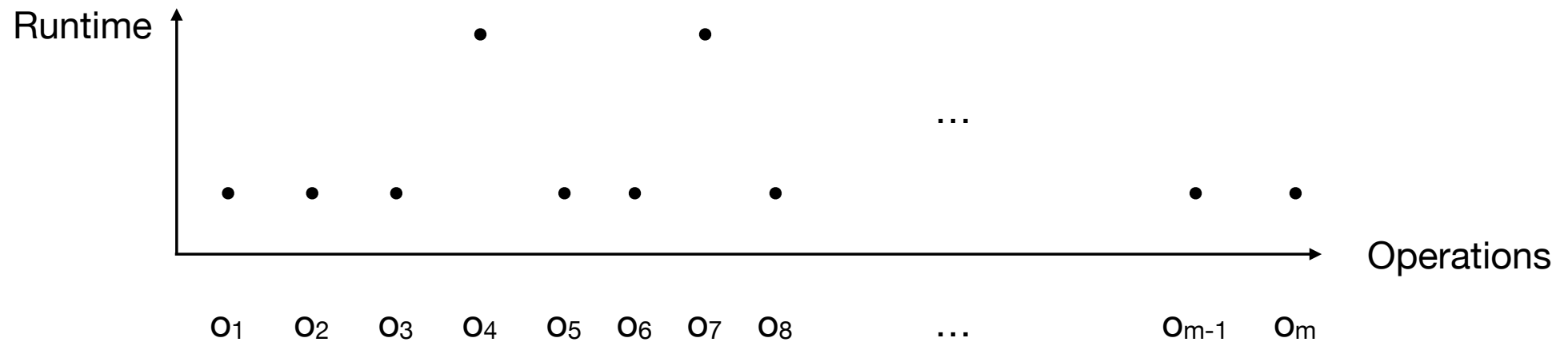
- **Motivating example.**
 - Consider an algorithm A that consists of m operations on data structure D.
 - The individual operations on D vary in running time. Many are fast, but a few are slow.
 - How do we analyze A's running time?
- **Standard analysis.** Number of operations $\times$ worst-case time for an operation.
- **Amortized analysis.** Total time of all operations.



Amortized Analysis

- Amortized time complexity.

- Let $T(m)$ be the worst-case time complexity for **any sequence of m operations** on a data structure. 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

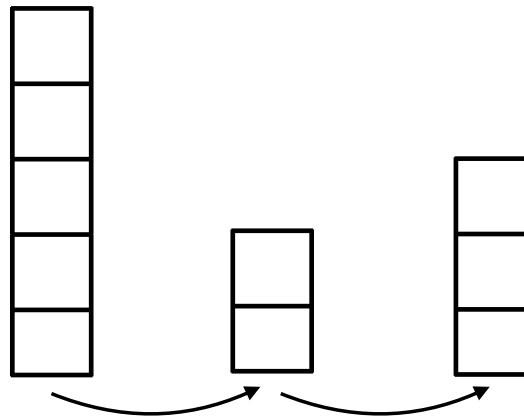
- Amortized Analysis
- **Aggregate Method**
- Accounting Method
- Potential Method

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.

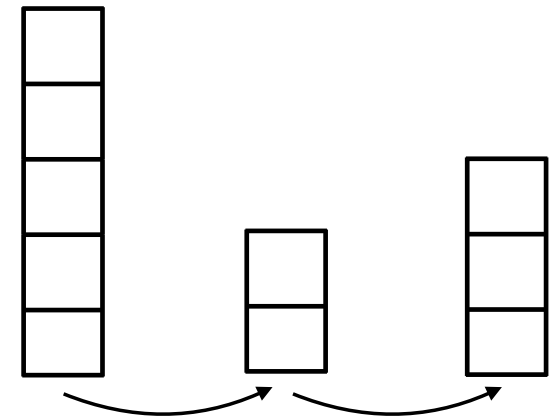
Stack

- **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 POP ped once for each time it is $PUSH$ ed.
 - $\Rightarrow$ Total number of $POPs$ is $\leq$ total number of $PUSHes$ $\leq m$.
 - $\Rightarrow$ Total time is $O(m)$
 - $\Rightarrow$ Amortized time of $MULTIPOP$ is $O(1)$.

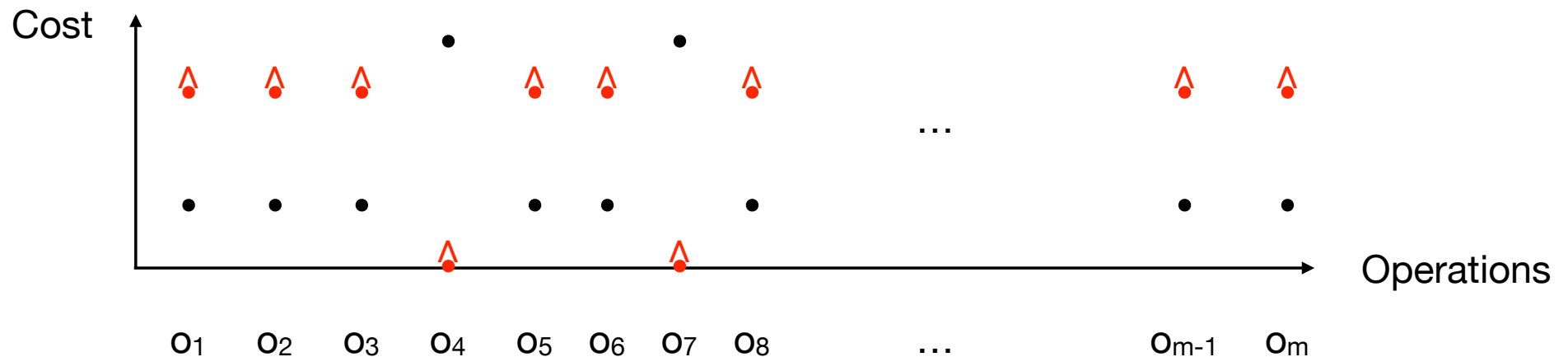


Amortized Analysis

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

Amortized Analysis

- **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 $O = o_1, \dots, o_m$.
$$\sum_{i=1}^m \hat{c}_i \geq \sum_{i=1}^m c_i.$$
- If $\hat{c}_i$ is an amortized cost $\Rightarrow \hat{c}_i$ is also amortized running time.



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 $MULTIPOP(k)$.

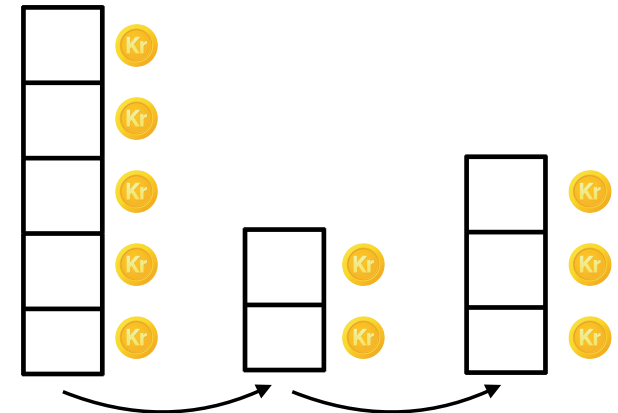
- **Costs.**

- A credit pays for a $PUSH$ or POP .
- $PUSH(x)$: 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

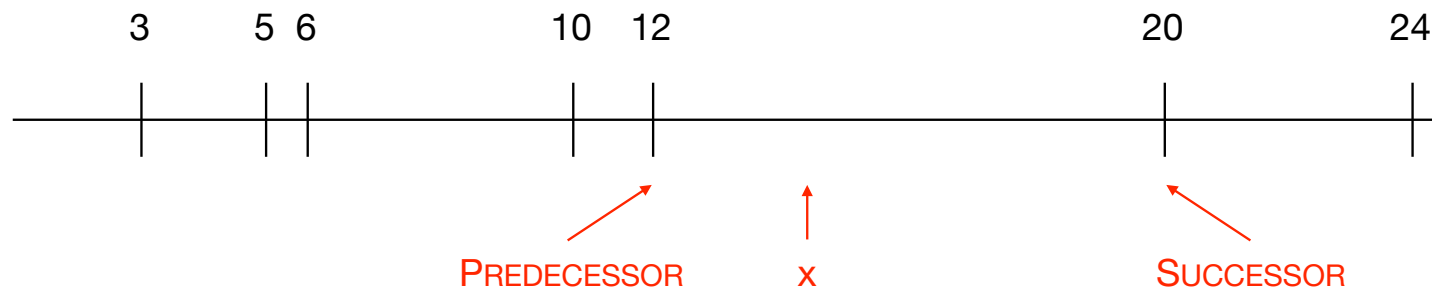
- **Amortized analysis.**

- Always enough credits to pay for $MULTIPOP$.
- $\Rightarrow \sum_{i=1}^m \hat{c}_i \geq \sum_{i=1}^m c_i$
- $\Rightarrow$ 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.
 - $\text{SEARCH}(x)$: return true if $x \in S$.
 - $\text{PREDECESSOR}(x)$: return the largest element in S that is $\leq x$.
 - $\text{SUCCESSOR}(x)$: return the smallest element in S that is $\geq x$.
 - $\text{INSERT}(x)$: add x to S .
 - $\text{DELETE}(x)$: 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 $A_0, A_1, A_2, \dots, A_{h-1}$. 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.

A_0 

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

A_1 

A_2 

A_3 

A_4 

Dynamic Binary Search

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

A_0 

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

A_1 

A_2 

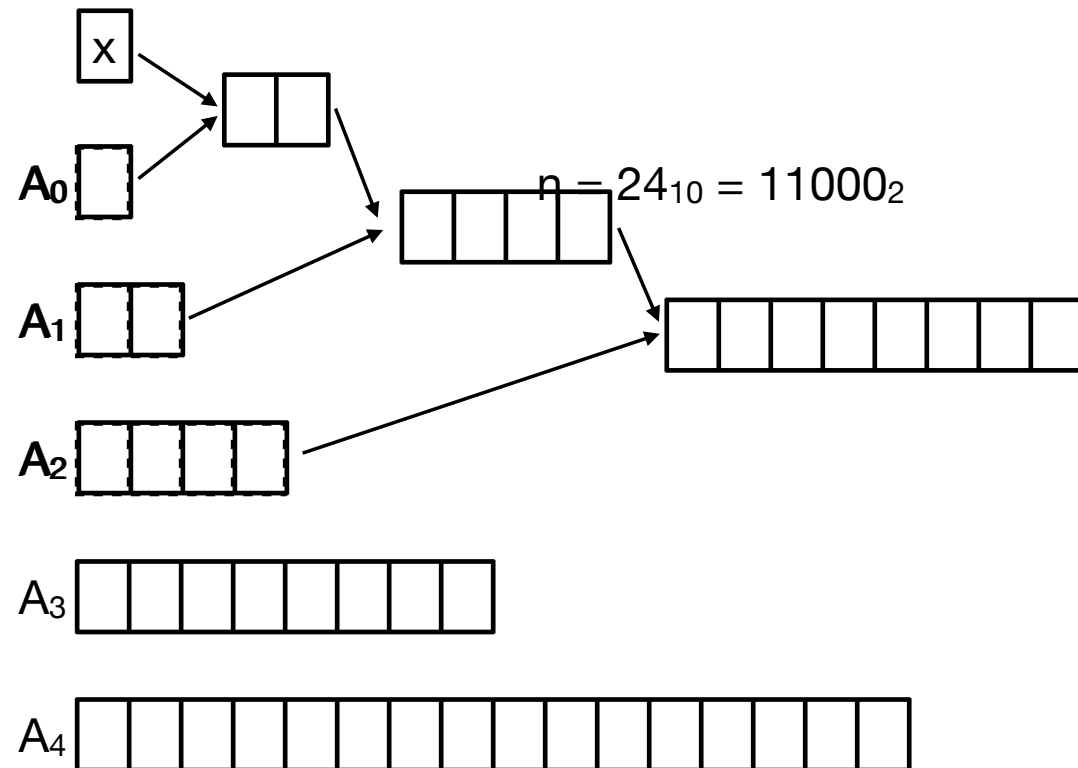
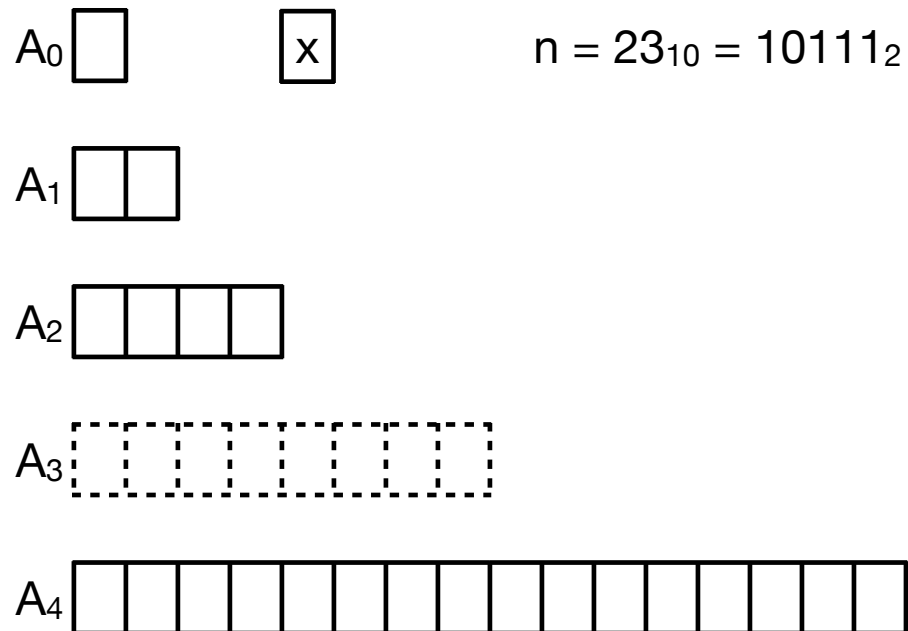
A_3 

A_4 

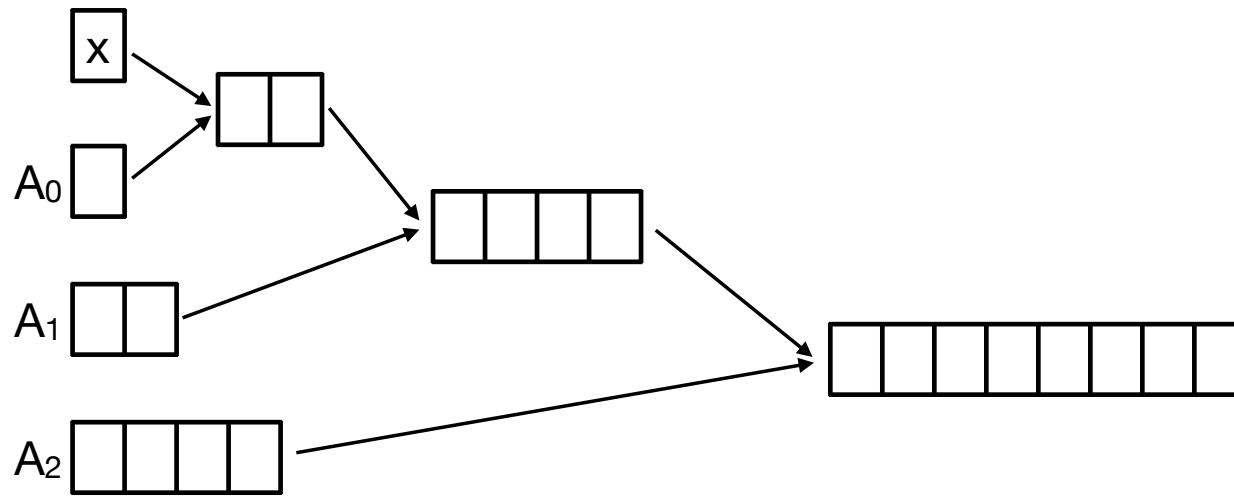
- Time. $O(\log^2 n)$.
- Similar idea for PREDECESSOR and SUCCESSOR.

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

- Observations.

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

A_0  $n = 23_{10} = 10111_2$

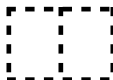
A_1 

A_2 

A_3 

A_4 

A_0  $n = 24_{10} = 11000_2$

A_1 

A_2 

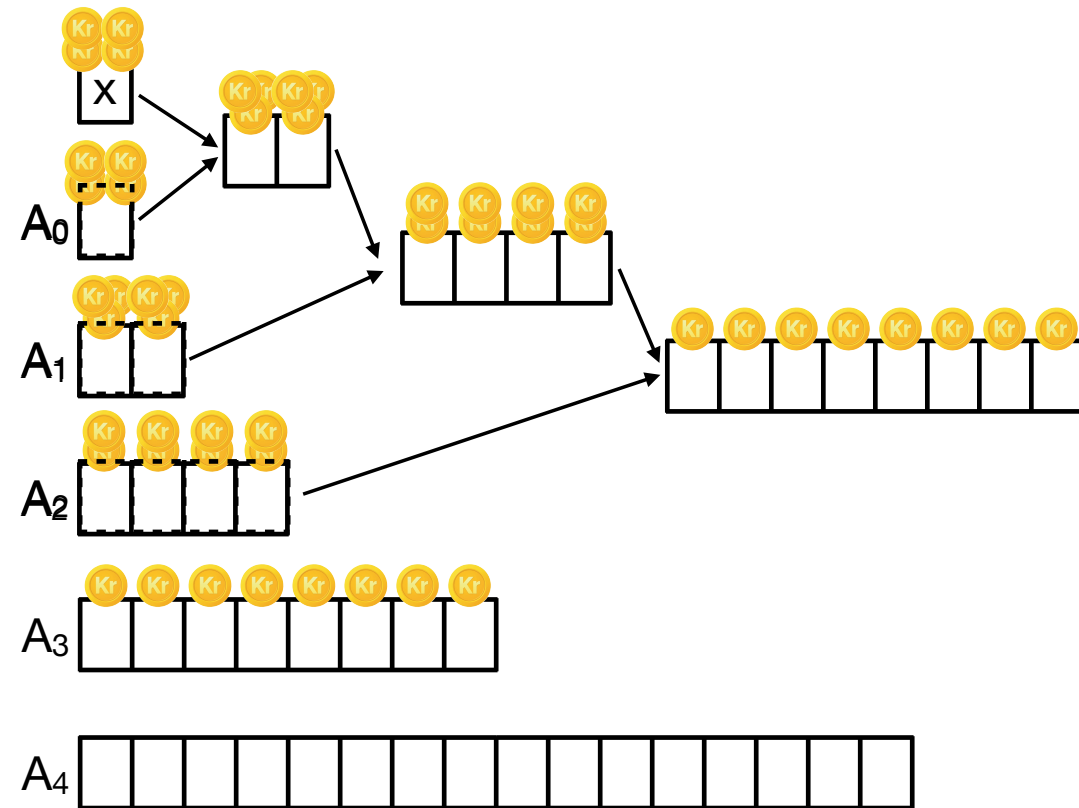
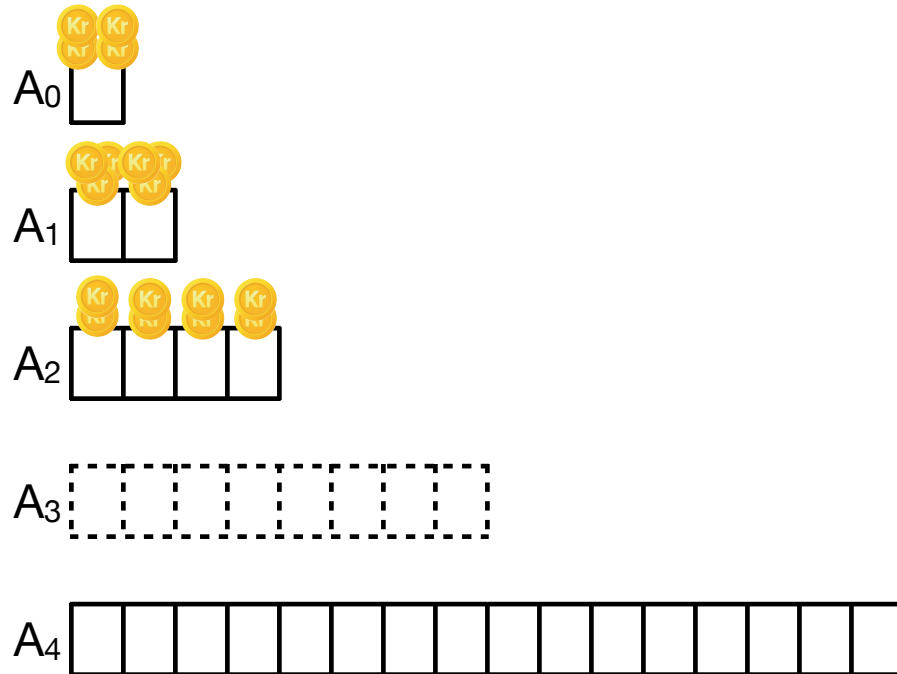
A_3 

A_4 

Dynamic Binary Search

- Costs.

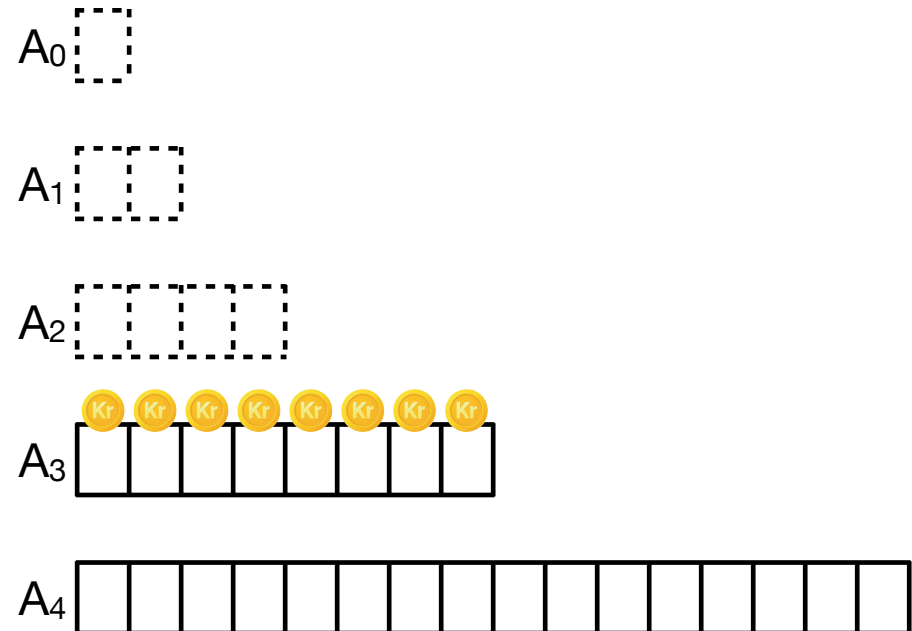
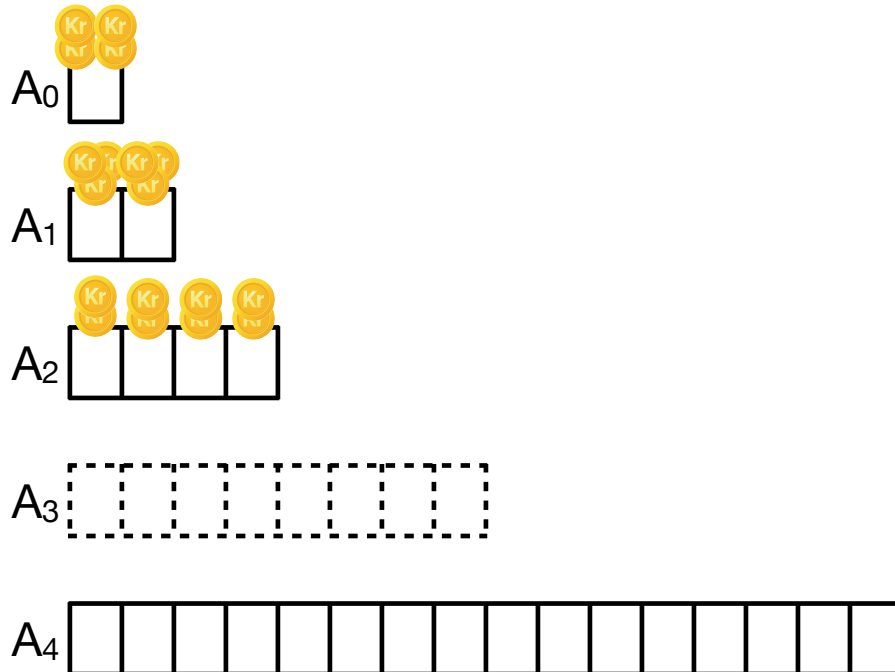
- A credit pays for a single element being part of a merge.
- INSERT(x): Assign $h-1$ credits to element. Use credits to pay for merges.



Dynamic Binary Search

- Amortized analysis.

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



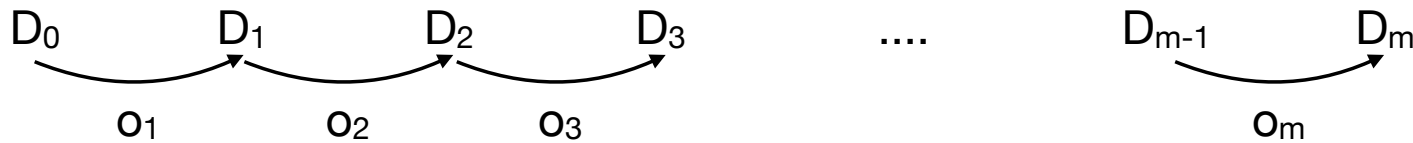
Dynamic Binary Search

- 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 we 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:

$$\sum_{i=1}^m \hat{c}_i = \sum_{i=1}^m (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^m c_i + \Phi(D_m) - \Phi(D_0) \geq \sum_{i=1}^m c_i$$

Potential Method

- Potential method.
 - Define a potential function $\Phi(D)$.
 - Compute the corresponding amortized cost $\Rightarrow$ 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.**
 - $PUSH(x)$: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.
 - $MULTIPOP(k)$: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k - k = 0$.
 - $\Rightarrow$ amortized running time of $MULTIPOP$ is $O(1)$.

Dynamic Binary Search

- Dynamic binary search.

- Maintain arrays $A_0, A_1, A_2, \dots, A_{h-1}$. 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.

A_0 

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

A_1 

A_2 

A_3 

A_4 

Dynamic Binary Search

- **Potential function.** Let $b_{h-1}b_{h-2}\dots b_0$ 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.

Φ

A_0 

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

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

A_1 

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

A_2 

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

A_3 

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

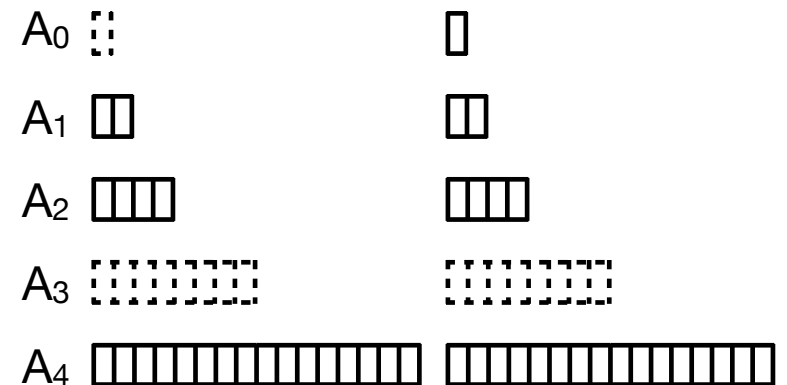
A_4 

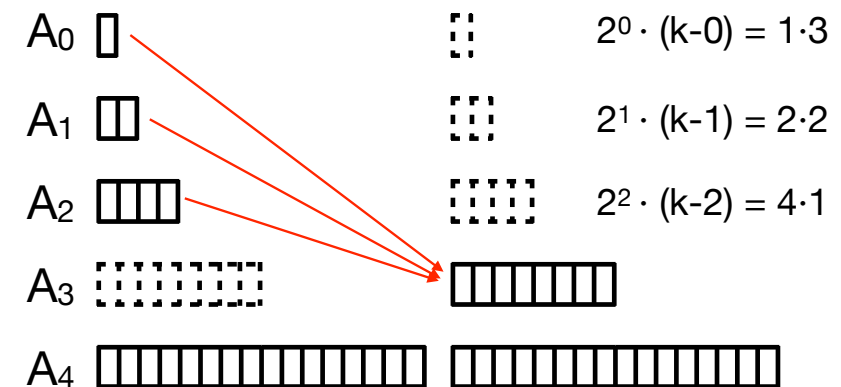
$$b_4 \cdot 2^4 \cdot (h-5) = 1 \cdot 16 \cdot 0 = 0$$

Dynamic Binary Search

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

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





Dynamic Binary Search

- $\Rightarrow$ 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