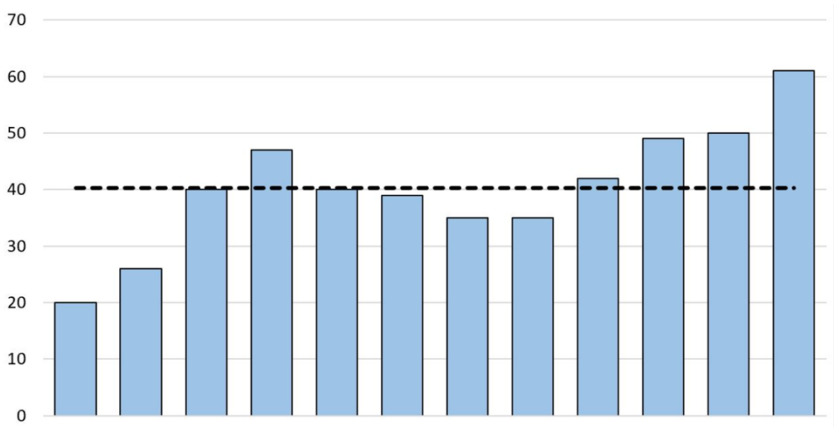
Amortized Analysis

Michael T. Goodrich
CS 263
Univ. of California, Irvine

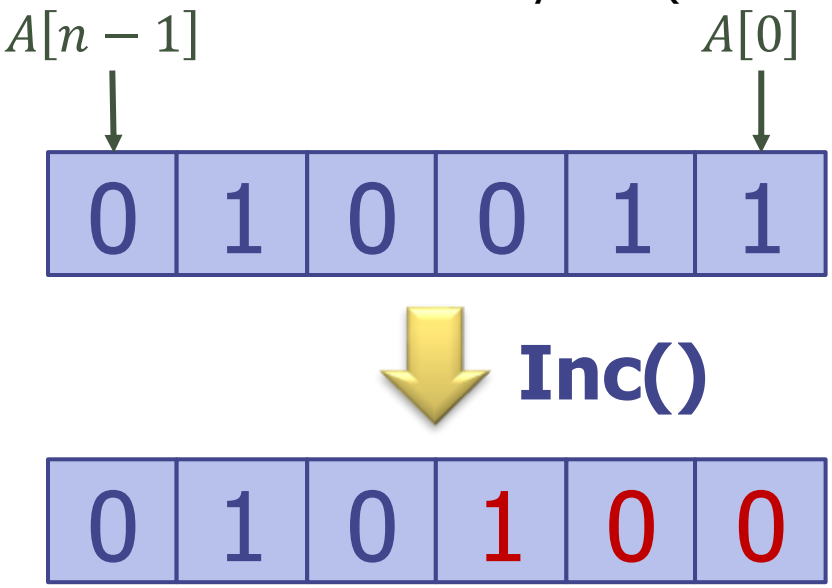


Some slides adapted from Nanjing University, Basheer Qolomany, Michael Bender, Haodong Hu, and Jeff Erickson.

The Accounting Method

Example: Binary Counter

- Use length n binary array A to represent a number.
- The number is 0 initially, and **Inc** op. adds 1 to this number.
- Cost of **Inc**: number of bits it flipped.
- Average cost of k **Inc** operations?
  - Easy answer: $O(n)$
  - More careful analysis… (Amortized analysis…)

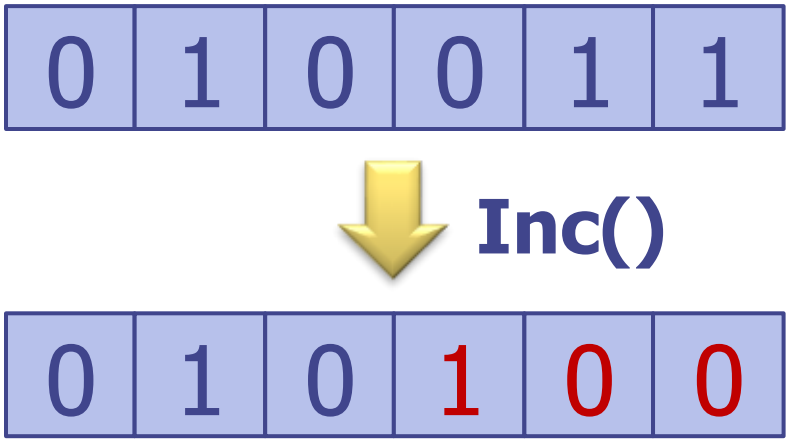


Inc(A):

```
i=0
while (i<n and A[i]==1)
  A[i]=0
  i=i+1
if (i<n)
  A[i]=1
```

Example: Binary Counter

- The number is 0 initially, and **Inc** op. adds 1 to this number.
- Cost of **Inc**: number of bits it flipped.
- In each **Inc**: $0 \to 1$: at most 1 bit; $1 \to 0$: many bits.
- But a bit has to be set to 1 before it resets to 0!
- If we deposit 1 whenever we $0 \to 1$, later $1 \to 0$ are "**free**"!
- Each **Inc** does $0 \to 1$ at most once, so amortized cost is $2 = O(1)$



Inc(A):

```
i=0
while (i<n and A[i]==1)
  A[i]=0
  i=i+1
if (i<n)
  A[i]=1
```

	Actual Total Cost	Amortized Total Cost
0 0 0 0 0 0		
	1	 × 2
0 0 0 0 0 1		
	$1 + 2 = 3$	 × 4
0 0 0 0 1 0		
	$3 + 1 = 4$	 × 6
0 0 0 0 1 1		
	$4 + 3 = 7$	 × 8
0 0 0 1 0 0		

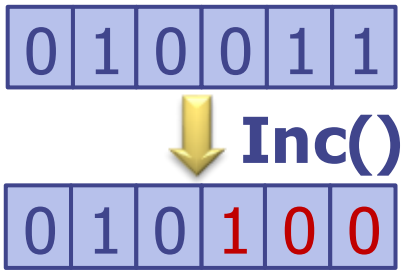
The Potential Method

- Consider a sequence operations:
  c_i = actual cost of i^{th} op.; $\widehat{c_i}$ = amortized cost of i^{th} op.
- For the amortized cost to be valid:
  $\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \widehat{c_i}$ for any $k \in \mathbb{N}^+$

- Design a **potential function** Φ that maps D.S. status to real values.
  - $\Phi(D_0)$: initial potential of D.S., usually set to 0.
  - $\Phi(D_i)$: potential of D.S. after i^{th} operation.
- Define $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- For amortized cost to be valid, need $\Phi(D_k) \geq \Phi(D_0)$ for all k.
- "Potential" is like the **balance in account** in "Counting Method".
  - Potential slowly accumulates during "cheap" operations (deposit).
  - Potential drops a lot after an "expensive" operation (withdraw).
- But the Potential Method could be more powerful in general…

Example: Binary Counter

- Design a **potential function** Φ that maps D.S. status to real values.
  - $\Phi(D_0)$: initial potential of D.S., usually set to 0.
  - $\Phi(D_i)$: potential of D.S. after i^{th} operation.
- Define $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$, need $\Phi(D_k) \geq \Phi(D_0)$ for all k.
- How to define $\Phi(D_i)$ for Binary Counter? (Recall potential is like "balance".)
- $\Phi(D_i)$ = number of 1s in the array after the i^{th} **Inc** operation.
- Clearly "$\Phi(D_k) \geq \Phi(D_0)$ for all k" is satisfied, how large is $\widehat{c_i}$?
- $c_i = (\#\text{ of bits } 0 \to 1) + (\#\text{ of bits } 1 \to 0)$
  $\Phi(D_i) - \Phi(D_{i-1}) = (\#\text{ of bits } 0 \to 1) - (\#\text{ of bits } 1 \to 0)$
- $\widehat{c_i} = 2 \cdot (\#\text{ of bits } 0 \to 1) \leq 2$



Making Structures Dynamic

- Two types of data structures for solving searching problems:
  - A *static* structure is built once and then searched many times; insertions and deletions of elements are not allowed.
  - *dynamic* structure: This structure is initially empty, and the three operations available on it are for inserting a new element, for deleting a current element, and for performing a search.

Static & Dynamic Structures

- To describe the performance of the static structure A we give three functions of n :
  - P(n) = the *preprocessing time* required to build A,
  - Q(n) = the *query time* required to perform a search in A,
  - S(n) = the *storage* required to represent A.
- We analyze the performance of the dynamic structure B by giving the functions
  - I(n) = the *insertion time* for B,
  - D(n) = the *deletion time* for B,
  - Q(n) = the *query time* required to perform a search in B,
  - S(n) = the *storage* required to represent B.

Decomposable search problems

- The notion of decomposable search problems, and the idea of a static-to-dynamic transformation, goes back to Bentley and Saxe (1980).
- A search problem is **decomposable** if, for any pair of disjoint data sets D and D′, the answer to a query over $D \cup D'$ can be computed in constant time from the answers to queries over the individual sets; that is,

$$Q(x, D \cup D') = Q(x, D) \diamond Q(x, D'),$$

- for some commutative and associative binary function $\diamond$ that maps $A \times A \rightarrow A$ and can be computed in $O(1)$ time.

Examples

- **Dictionary:** Data objects and query objects have the same arbitrary type; a query asks whether a query object x is a member of the data set D. Here, $\mathscr{A}$ is the set of booleans, $\diamond = \vee$, and $\perp = \textsc{False}$.
- **Range minimum queries:** Data objects are elements of some totally ordered universe $\mathscr{U}$, stored in an array $A[1..n]$, and query objects are pairs (i, j) of indices where $i \leq j$; a query asks for the minimum element of the subarray $A[i..j]$. Here, $\mathscr{A} = \mathscr{U}$, $\diamond = \min$, and $\perp = \infty$.
- **Rectangle counting:** Data objects are points in the plane; query objects are rectangles; a query asks for the number of points in a given rectangle. Here, $\mathscr{A} = \mathbb{N}$, $\diamond = +$, and $\perp = 0$.

Examples, continued

- **Nearest neighbor:** Data objects are points in some metric space; query objects are points in the same metric space; a query asks for the distance from a given query point to the nearest point. Here, $\mathscr{A}$ is the set of non-negative real numbers, $\diamond = \min$, and $\perp = \infty$.
- **Triangle emptiness:** Data objects are points in the plane; query objects are triangles; a query asks whether any data point lies in a given query triangle. Here, $\mathscr{A}$ is the set of booleans, $\diamond = \vee$, and $\perp = \textsc{False}$.
- **Interval stabbing:** Data objects are intervals on the real line; query objects are points on the real line; a query asks for the subset of data intervals that contain a given query point. Here, $\mathscr{A}$ is the set of all finite sets of real intervals, $\diamond = \cup$, and $\perp = \varnothing$.

Insertions Only - Build & Query

Our data structure consists of $\ell = \lfloor \lg n \rfloor$ *levels* $L_0, L_1, \ldots, L_{\ell-1}$. Each level L_i is either empty or a static data structure storing exactly 2^i items. Observe that for any value of n, there is a unique set of levels that must be non-empty, specified by the 1s in the binary representation of n. To answer a query, we perform queries in every non-empty level and combine the results. (This is where we require the queries to be decomposable.) For simplicity, the following pseudocode assumes that $\textsc{Query}(x, L_i)$ returns $\perp$ if L_i is empty; recall that $ans \diamond \perp = ans$.

```
NewQuery(x):
  ans ← ⊥
  for i ← 0 to ℓ − 1
      ans ← ans ⋄ Query(x, L_i)
  return ans
```

The total query time is clearly at most $\sum_{i=0}^{\ell-1} Q(2^i) < \ell \cdot Q(n) = O(\log n) \cdot Q(n)$

Insertions Only - Updates

The insertion algorithm follows the standard algorithm for incrementing a binary counter, where the presence or absence of each L_i plays the role of the ith least significant bit. We find the smallest empty level k; build a new data structure L_k containing the new item and all the items stored in $L_0, L_1, \ldots L_{k-1}$; and finally discard all the levels smaller than L_k.

```
INSERT(x):
  Find minimum k such that L_k = ∅
  L_k ← {x} ∪ ⋃_{i<k} L_i        ⟨⟨takes P(2^k) time⟩⟩
  for i ← 0 to k − 1
      destroy L_i
```

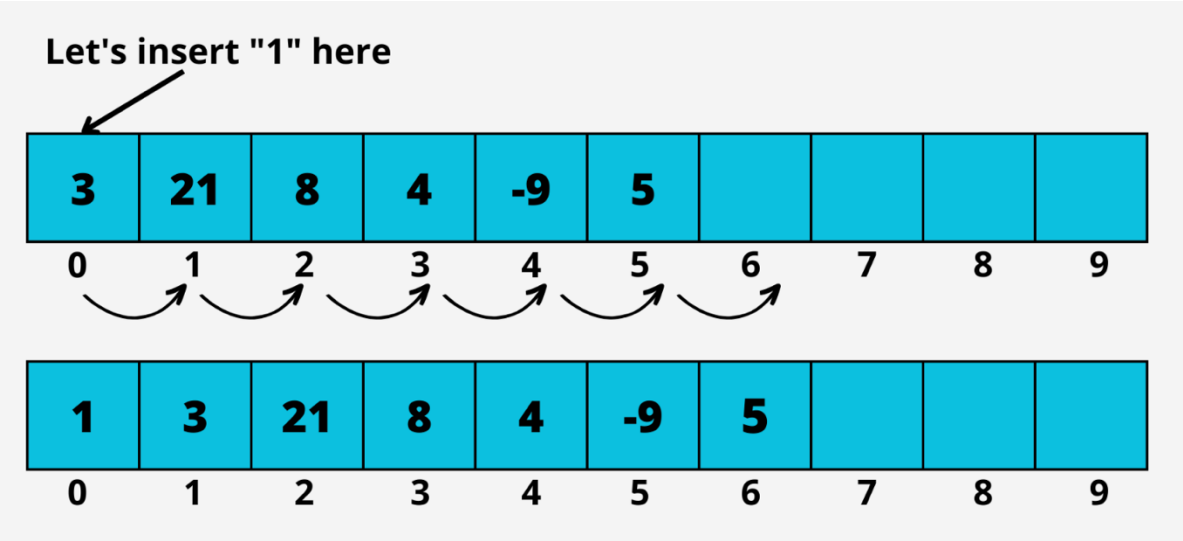
During the lifetime of the data structure, each item will take part in the construction of $\lg n$ different data structures. Thus, if we charge

$$I'(n) = \sum_{i=0}^{\lg n} \frac{P(2^i)}{2^i} = O(\log n)\frac{P(n)}{n}.$$

for each insertion, the total charge will pay for the cost of building all the static data structures. If $P(n) > n^{1+\varepsilon}$ for any $\varepsilon > 0$, the amortized insertion time is actually $O(P(n)/n)$.

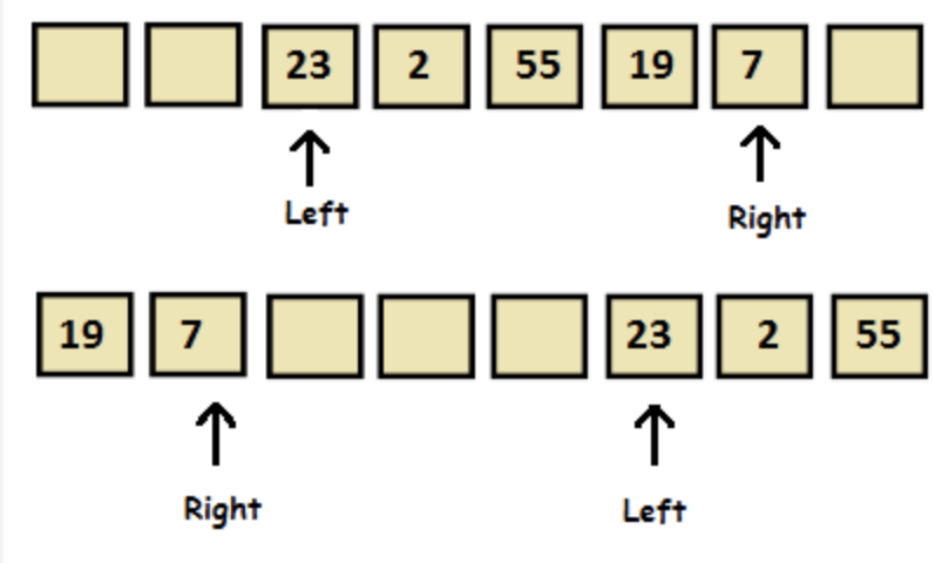
Array Order Maintenance

- Problem: Maintain an array, A, of elements according to a specific order.
- This is often a low-level data structure used by other structures.
- Performance: insert & delete take O(n) time in the worst case (and average case).



Array Order Maintenance

- Best case is O(1) time: insert/delete at the end.
- Another best case in O(1) time: insert/delete at the beginning or end.
  - How?



Tiered Vector

- The worst-case performance for insert and delete can be improved to $O(n^{\varepsilon})$ for any constant $\varepsilon>0$.
- E.g., for $\varepsilon=1/2$, we keep $n^{1/2}$ subarrays of size exactly $n^{1/2}$ each (so indexing still takes O(1) time).

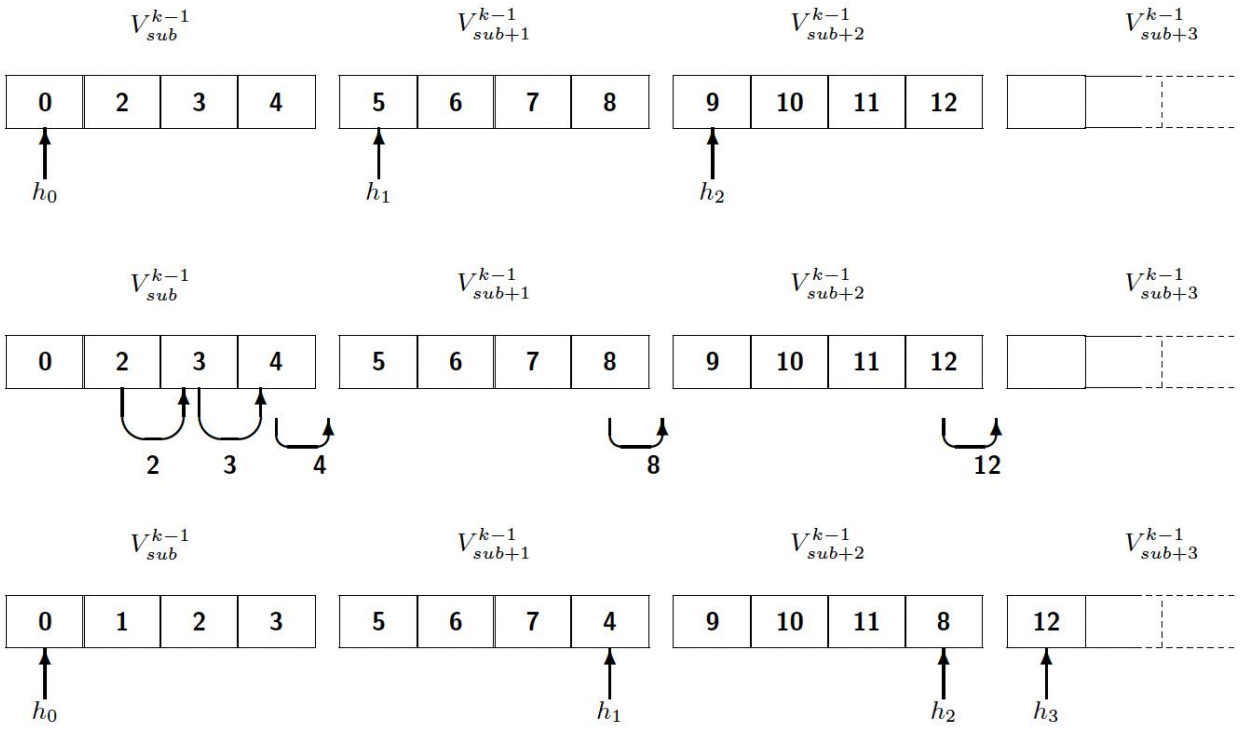


Fig. 2. Insertion of element **1** at rank r in a 2-level tiered vector.

Tiered Vector Amortization

- If we grow or shrink by more than a factor of 2, we rebuild the whole structure.
- This gives amortized overhead of O(1) using an argument similar to that for growing a dynamic array.

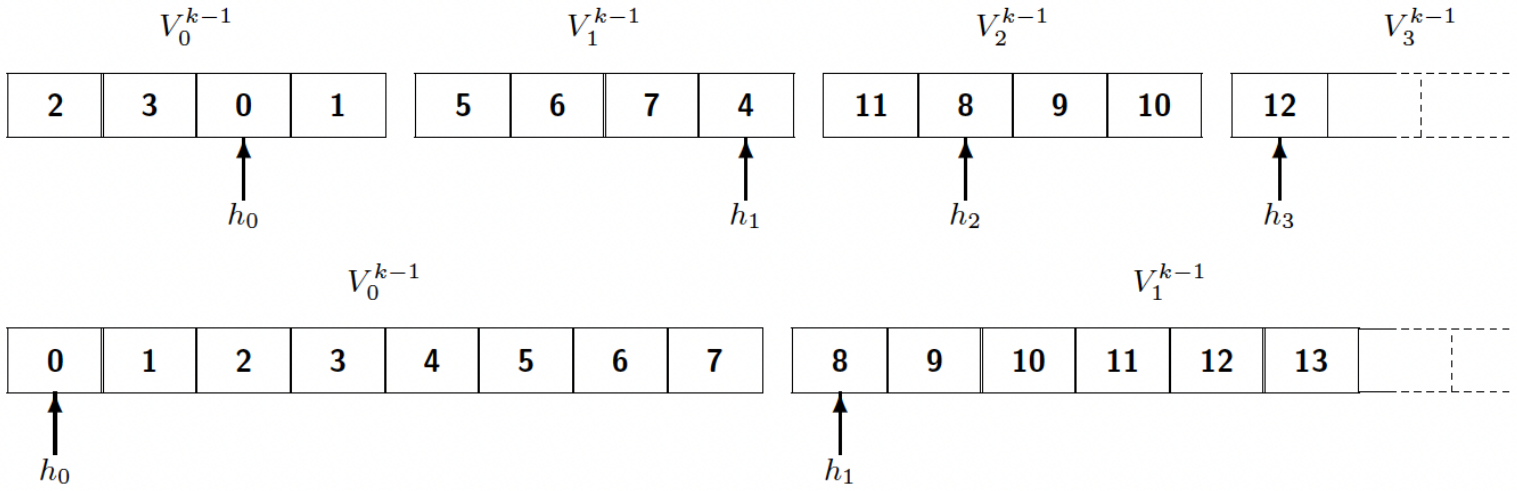
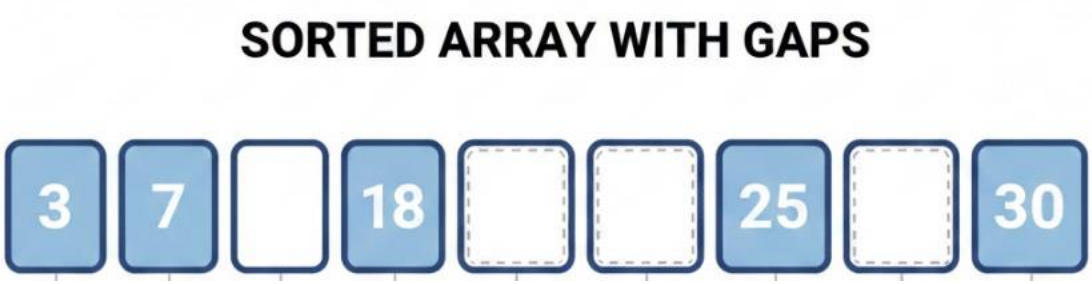


Fig. 3. Expansion and reordering of a 2-level tiered vector..

Packed-Memory Array

- A **packed-memory ar**ray (PMA) maintains a dynamic set of N elements in sorted order in a Θ(N)-sized array.
- The idea is to intersperse Θ(N) empty spaces or gaps among the elements so that only a small number of elements need to be shifted around on an insert or delete.
- The PMA maintains the **density invariant** that in any region of size S (for S greater than some small constant value), there are Θ(S) elements stored in it. To scan L elements after a given element x costs Θ(1+L/B) memory transfers, where B is the block size. Also, given the position of any element, we can find the next or previous one in O(1) time.



Segments and Windows

We now give some terminology. We divide the PMA into $\Theta(N/\log N)$ ***segments***, each of size $\Theta(\log N)$, and we let the number of segments be a power of 2. We call a contiguous group of segments a ***window***. We view the PMA in terms of a tree structure, where the nodes of the tree are windows. The root node is the window containing all segments, and a leaf node is a window containing a single segment. A node in the tree that is a window of 2^i segments has two children, a left child that is the window of the first 2^{i-1} segments and a right child that is the window of the last 2^{i-1} segments.

Density Thresholds

We let the height of the tree be h, so that $2^h = \Theta(N/\log N)$ and $h = \lg N - \lg\lg N + O(1)$. The nodes at each height ℓ have an ***upper density threshold*** τ_ℓ and a ***lower density threshold*** ρ_ℓ, which together determine the acceptable density of keys within a window of 2^ℓ segments. As the node height *increases*, the upper density thresholds *decrease* and the lower density thresholds *increase*. Thus, for constant minimum and maximum densities $D_{\min}$ and $D_{\max}$, we have

$$D_{\min} = \rho_0 < \cdots < \rho_h < \tau_h < \cdots < \tau_0 = D_{\max}. \qquad (1)$$

Density is distributed by height

More formally, upper and lower density thresholds for nodes and height ℓ are defined as follows:

$$\tau_\ell = \tau_h + (\tau_0 - \tau_h)(h - \ell)/h \quad (2)$$

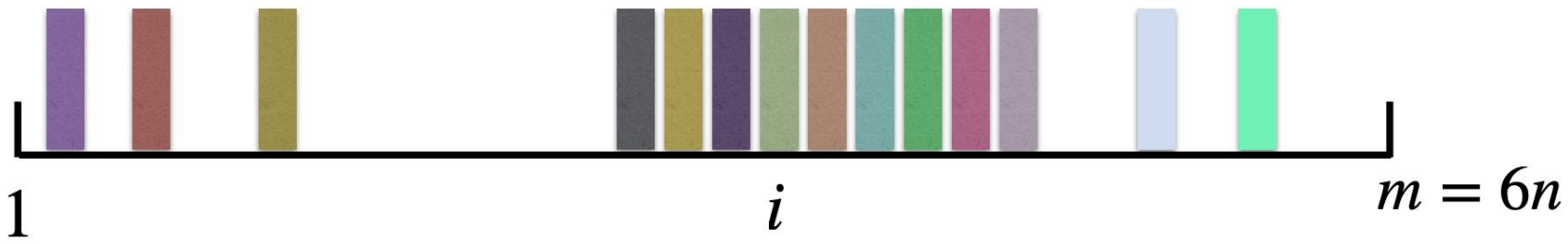
$$\rho_\ell = \rho_h - (\rho_h - \rho_0)(h - \ell)/h. \quad (3)$$

Moreover,

$$2\rho_h < \tau_h, \quad (4)$$

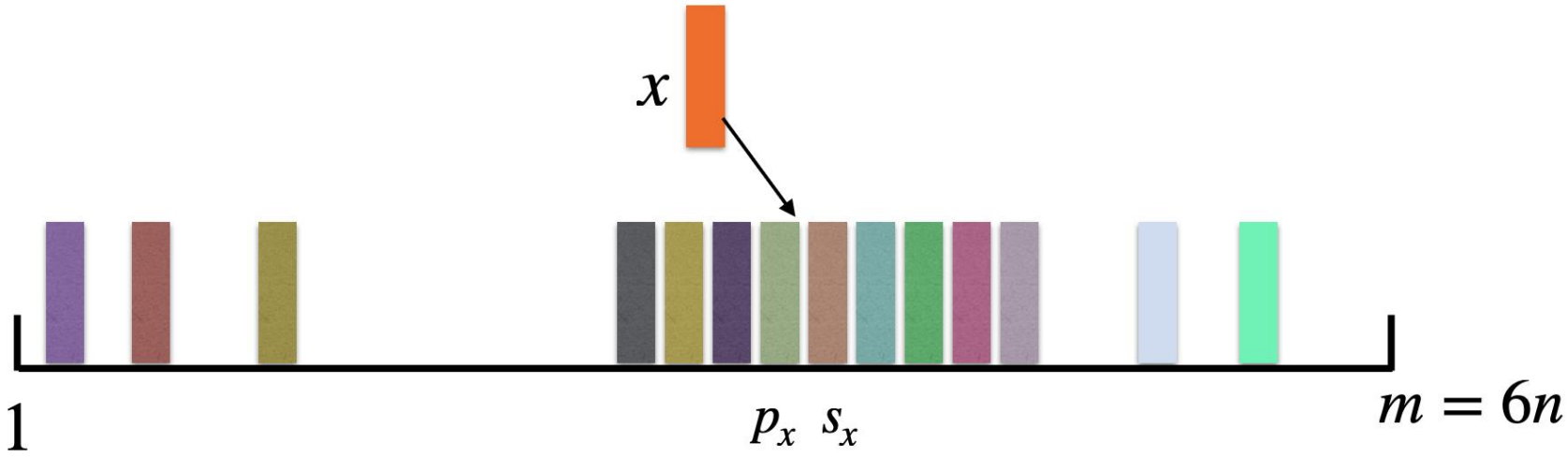
Insert in a PMA

- If there is enough space in the leaf (segment) containing x, then we rearrange the elements evenly within the leaf to make room for y.
- If the leaf is full, then we find the closest ancestor of the leaf whose density is within the permitted thresholds and rebalance.



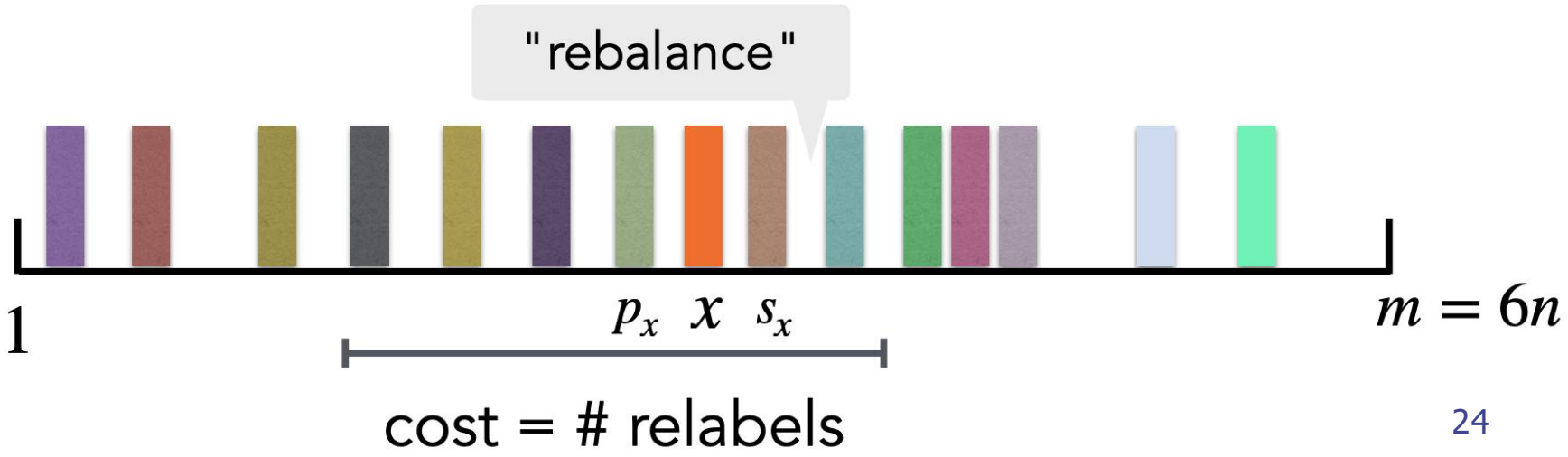
Insert in a PMA, Part 2

- If there is enough space in the leaf (segment) containing x, then we rearrange the elements evenly within the leaf to make room for y.
- If the leaf is full, then we find the closest ancestor of the leaf whose density is within the permitted thresholds and rebalance



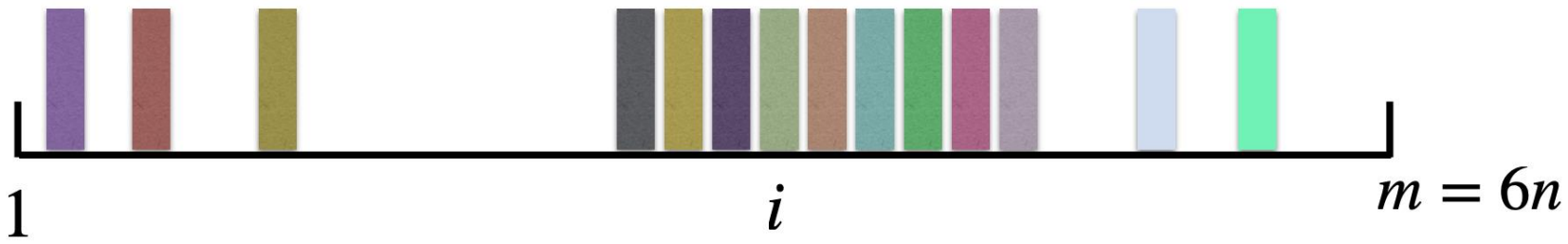
Insert in a PMA: Rebalance

- If there is enough space in the leaf (segment) containing x, then we rearrange the elements evenly within the leaf to make room for y.
- If the leaf is full, then we find the closest ancestor of the leaf whose density is within the permitted thresholds and rebalance.



Delete in a PMA

- To delete an element x, we remove x from its segment. If the segment falls below its density threshold, then, as before, we find the smallest enclosing window whose density is within threshold and rebalance.
- If the entire array is above the maximum density threshold (resp., below the minimum density threshold), then we recopy the keys into a PMA of twice (resp., half) the size.



Some Notation

We introduce further notation. Let ***Cap***(u_ℓ) denote the number of array positions in node u_ℓ of height ℓ. Since there are 2^ℓ segments in the node, the capacity is $\Theta(2^\ell \log N)$. Let ***Gaps***(u_ℓ) denote the number of gaps, i.e., unfilled array positions in node u_ℓ. Let ***Density***(u_ℓ) denote the fraction of elements actually stored in node u_ℓ, i.e., $\text{Density}(u_\ell) = 1 - \text{Gaps}(u_\ell)/\text{Cap}(u_\ell)$.

Amortized Analysis

THEOREM 1. *If the rebalance in a PMA satisfies the rebalance property, then updates take $O(\log^2 N)$ amortized element moves and $O(1+(\log^2 N)/B)$ amortized memory transfers.*

PROOF. Let u_ℓ be a node at level ℓ. A rebalance of u_ℓ is trigged by an insert or delete that pushes one descendant node u_i at each height $i = 0, \ldots, \ell - 1$ above its upper threshold τ_i or below its lower threshold ρ_i. (If this were not the case, then we would rebalance a node of a lower height than ℓ.)

Consider one particular such node u_i. Before the sweep of u_i's parent u_{i+1},

$$\text{Density}(u_i) > \tau_i \quad \text{or} \quad \text{Density}(u_i) < \rho_i .$$

After the sweep of u_{i+1}, by the rebalance property,

$$\rho_{i+1} \leq \text{Density}(u_i) \leq \tau_{i+1} .$$

Amortized Analysis, part 2

Therefore we need at least

$$(\tau_i - \tau_{i+1})\mathrm{Cap}(u_i)$$

inserts or at least

$$(\rho_{i+1} - \rho_i)\mathrm{Cap}(u_i)$$

deletes before the next sweep of node u_{i+1}. Therefore the amortized size of a sweep of node u_{i+1} per insert into child node u_i is at most

$$\begin{aligned} & \max\left\{ \frac{\mathrm{Cap}(u_{i+1})}{(\tau_i - \tau_{i+1})\mathrm{Cap}(u_i)}, \frac{\mathrm{Cap}(u_{i+1})}{(\rho_{i+1} - \rho_i)\mathrm{Cap}(u_i)} \right\} \\ = \quad & \max\left\{ \frac{2}{\tau_i - \tau_{i+1}}, \frac{2}{\rho_{i+1} - \rho_i} \right\} \\ = \quad & O(\log N). \end{aligned}$$

Amortized Analysis, conclusion

When we insert an element into the PMA, we actually insert into $h = \Theta(\log N)$ such nodes u_i, one at each level in the tree. Therefore the total amortized size of a rebalance per insertion into the PMA is $O(\log^2 N)$. Thus, the amortized number of element moves per insert is $O(\log^2 N)$. Because a rebalance is composed of a constant number of sequential scans, the amortized number of memory transfers per insert is $O(1 + (\log^2 N)/B)$, as promised.