CSE 548: Analysis of Algorithms

Guest Lecture
(The α Technique)

Inspiration Comes from Lectures Given by
Jeff Erickson, Seth Pettie, Vijaya Ramachandran and Raimund Seidel

Guest Lecturer: Shih-yu Tsai
(Slides: Rezaul A. Chowdhury, Shih-yu Tsai)
Department of Computer Science
SUNY Stony Brook
Spring 2019

Iterated Functions

$$f^*(n) = \min\left\{i \geq 0: \underbrace{f\Big(f\big(f(\ldots f(n)\ldots)\big)\Big)}_{i\ times} \leq 1\right\}$$

$$= \min\{i \geq 0: f^{(i)}(n) \leq 1\},$$

where $$f^{(i)}(n) = \begin{cases} n & if\ i = 0 \\ f\left(f^{(i-1)}(n)\right) & if\ i > 0 \end{cases}$$

Example: If $f = \log$, we have:

$$\log^{(0)}(65536) = 65536$$
$$\log^{(1)}(65536) = 16$$
$$\log^{(2)}(65536) = 4$$
$$\log^{(3)}(65536) = 2$$
$$\log^{(4)}(65536) = 1$$
$$\therefore \log^*(65536) = 4$$

Iterated Functions

$f(n)$	$f^*(n)$
$n-1$	$n-1$
$n-2$	$\frac{n}{2}$
$n-c$	$\frac{n}{c}$
$\frac{n}{2}$	$\log_2 n$
$\frac{n}{c}$	$\log_c n$
$\log n$	$\log^* n$

$\log^* (n)$ grows extremely slowly

$$\log^* 2 = 1$$

$$\log^* 2^2 = 2$$

$$\log^* 2^4 = 3$$

$$\log^* 2^{16} = 4$$

$$\log^* 2^{65536} = 5$$

$$\log^* 2^{2^{65536}} = 6 \ldots.$$

The Inverse Ackermann Function: $\alpha(n)$

$\alpha(n)$ rows

$f(n)$	$f^*(n)$	
$\log n$	$\log^* n$	> 3
$\log^* n$	$\log^{**} n$	> 3
$\log^{**} n$	$\log^{***} n$	> 3
...	...	
$\log^{\overbrace{*\cdots*}^{k-2}} n$	$\log^{\overbrace{*\cdots*}^{k-1}} n$	> 3
$\log^{\overbrace{*\cdots*}^{k-1}} n$	$\log^{\overbrace{*\cdots*}^{k}} n$	≤ 3

$$\alpha(n) = \min\left\{k \geq 1 : \log^{\overbrace{*\cdots*}^{k}} n \leq 3\right\}$$

Example: $\alpha(65536)$

	$f(n)$	$f^*(n)$
$\alpha(65536)$ rows	$\log 65536$	$\log^* 65536 = 4 \quad > 3$
	$\log^* 65536$	$\log^{**} 65536 = 3 \quad \leq 3$

$$\alpha(65536) = \min\left\{k \geq 1 : \log^{\overbrace{*\cdots*}^{k}} 65536 \quad \leq 3\right\} = 2$$

$$\log^{(0)}(65536) = 65536 \qquad (\log^*)^{(0)}(65536) = 65536$$

$$\log^{(1)}(65536) = 16 \qquad (\log^*)^{(1)}(65536) = 4$$

$$\log^{(2)}(65536) = 4 \qquad (\log^*)^{(2)}(65536) = \log^*(4) = 2$$

$$\log^{(3)}(65536) = 2 \qquad (\log^*)^{(3)}(65536) = \log^*(2) = 1$$

$$\log^{(4)}(65536) = 1 \qquad \therefore \log^{**}(65536) = 3$$

$$\therefore \log^*(65536) = 4$$

The Partial Sums Data Structure

Example:
The Partial Sums on Array of numbers

3	4	6	2	11	7	3	5	5	2

$$4 + 6 + 2 + 11 + 7 + 3 = ?$$

Semigroups

Semigroup ($\boldsymbol{\Pi}$, $\bigoplus$)**:** A set Π together with an associative binary operation $\bigoplus$: $\Pi \times \Pi \rightarrow \Pi$.

Examples:

$$(\Re, max)$$

$$(\{ true, false \}, logical\ OR)$$

$$(k \times k\ matrices, matrix\ multiplication)$$

Partial Semigroup Sums

Given (i) a semigroup ($\Pi, \bigoplus$), and

(ii) an array $A[1 \ldots n]$ with each entry $A[i] \in \Pi$

Goal: Preprocess A using as little space as possible so that for all $1 \leq i \leq j \leq n$, queries of the form $A[i] \bigoplus A[i+1] \bigoplus \cdots \bigoplus A[j]$ can be answered efficiently.

Query Complexity: #times the $\bigoplus$ operation is applied

Space Complexity: #values from Π stored in the data structure

$\boldsymbol{k}$-op structure: A data structure with query complexity k

$\boldsymbol{S_k(n)}$: #values from Π to be stored so that every partial sum query can be answered using at most k applictions of the $\bigoplus$ operation

Bound 0

Bound 0: $S_1(n) \leq n \log n$.

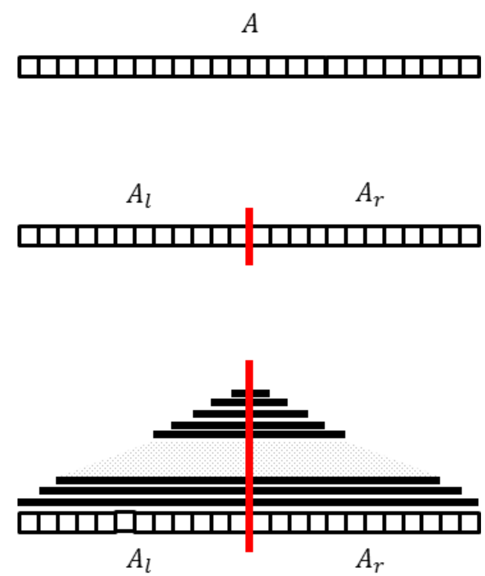
Construction of a 1-op structure:

Input array A of size n

Split A into A_l and A_r of size $\frac{n}{2}$ each

Compute: all suffix-sums of A_l, and
all prefix-sums of A_r

Recurse: 1-op structure for A_l, and
1-op structure for A_r



Query: Either crosses A's midpoint (return suffix-sum $\oplus$ prefix-sum),
or lies completely inside A_l (recurse) or A_r (recurse)

Bound 0

Bound 0: $S_1(n) \leq n \log n$.

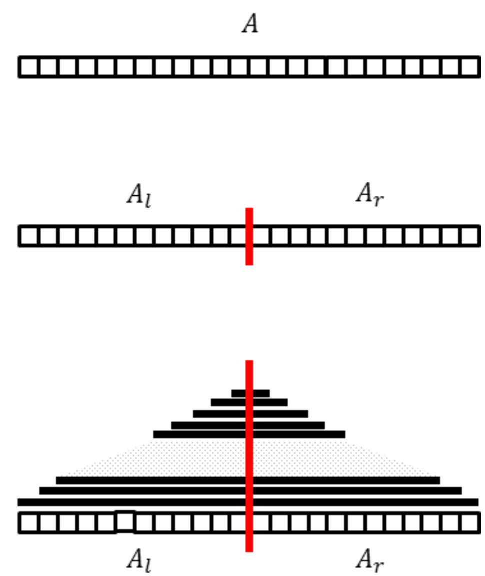
Construction of a 1-op structure:

Input array A of size n

Split A into A_l and A_r of size $\frac{n}{2}$ each

Compute: all suffix-sums of A_l, and
all prefix-sums of A_r

Recurse: 1-op structure for A_l, and
1-op structure for A_r



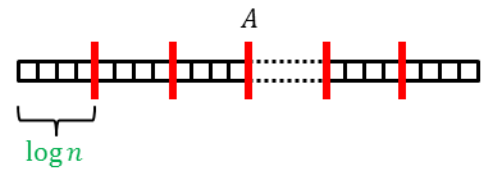
Space: $S_1(n) \leq n + 2S_1\left(\frac{n}{2}\right)$
$\leq n \log n$

Bound 1

Bound 1: $S_3(n) \le 3n \log^* n$.

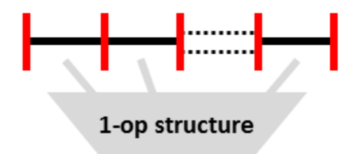
Construction of a 3-op structure:

Split A into $\frac{n}{\log n}$ subarrays of size $\le \log n$ each

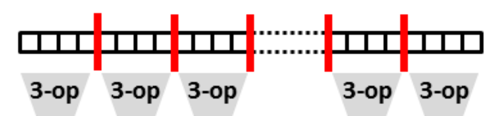


Compute: all suffix- and prefix- sums within each subarray

Build: 1-op structure for $\frac{n}{\log n}$ subarray sums



Recurse: 3-op structure for each subarray



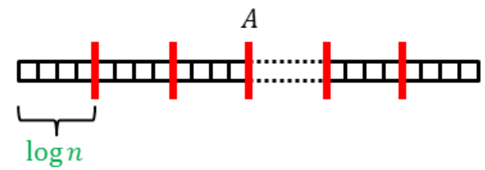
Query: Either completely inside a subarray (recurse), or crosses subarray boundaries (return suffix-sum $\oplus$ answer from 1-op structure $\oplus$ prefix-sum)

Bound 1

Bound 1: $S_3(n) \le 3n \log^* n$.

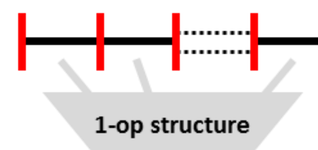
Construction of a 3-op structure:

Split A into $\frac{n}{\log n}$ subarrays of size $\le \log n$ each

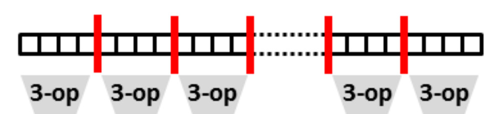


Compute: all suffix- and prefix- sums within each subarray

Build: 1-op structure for $\frac{n}{\log n}$ subarray sums



Recurse: 3-op structure for each subarray



Space: $S_3(n) \le 2n + S_1\left(\frac{n}{\log n}\right) + \frac{n}{\log n} S_3(\log n)$

$$\le 3n + \frac{n}{\log n} S_3(\log n) \le 3n \log^* n$$

Bound k

Bound k: $S_{2k+1}(n) \leq (2k+1)n \log^{\overbrace{*\cdots*}^{k}} n.$

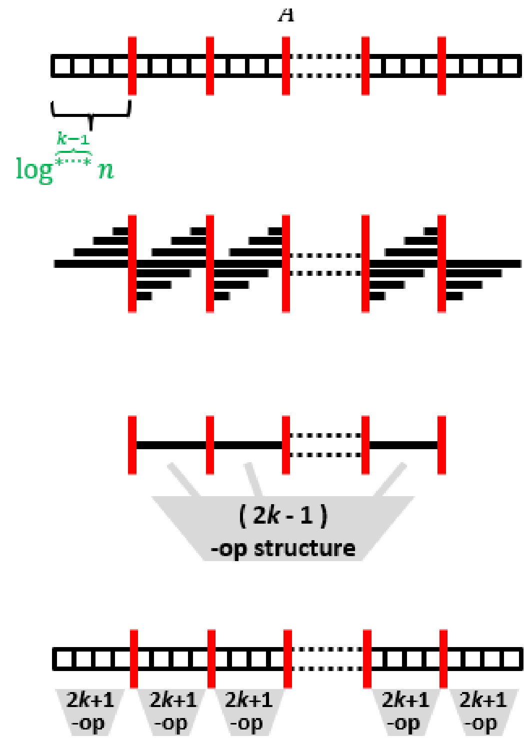
Construction of a $(2k+1)$-op structure:

Split A into $n/\log^{\overbrace{*\cdots*}^{k-1}} n$ subarrays of size $\leq \log^{\overbrace{*\cdots*}^{k-1}} n$ each

Compute: all suffix- and prefix- sums within each subarray

Build: $(2k-1)$-op structure for $n/\log^{\overbrace{*\cdots*}^{k-1}} n$ subarray sums

Recurse: $(2k+1)$-op structure for each subarray

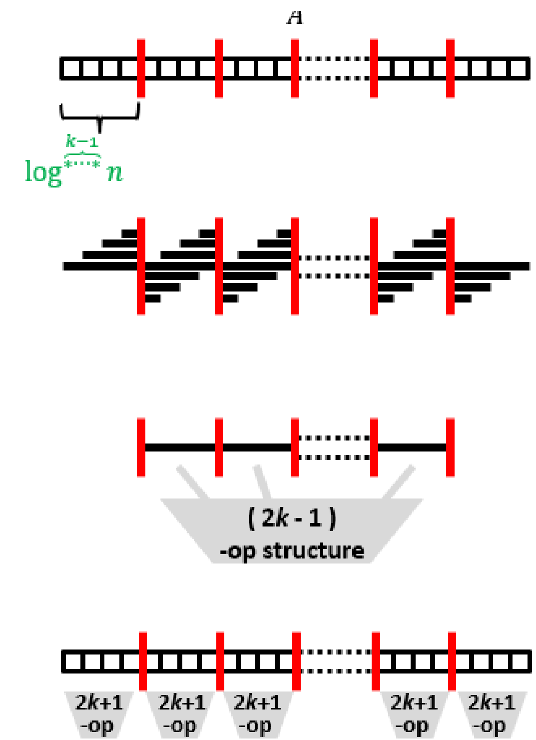


Query: Either completely inside a subarray (recurse), or crosses subarray boundaries (return suffix-sum $\oplus$ answer from $(2k-1)$-op structure $\oplus$ prefix-sum)

Bound k

Bound k: $S_{2k+1}(n) \le (2k+1)n \log^{\overbrace{*\cdots*}^{k}} n.$

Construction of a $(2k+1)$-op structure:



Split A into $n / \log^{\overbrace{*\cdots*}^{k-1}} n$ subarrays of size $\le \log^{\overbrace{*\cdots*}^{k-1}} n$ each

Compute: all suffix- and prefix- sums within each subarray

Build: $(2k-1)$-op structure for $n / \log^{\overbrace{*\cdots*}^{k}} n$ subarray sums

Recurse: $(2k+1)$-op structure for each subarray

Space: $S_{2k+1}(n) \le 2n + S_{2k-1}\left(\frac{n}{\log^{\overbrace{*\cdots*}^{k-1}} n}\right) + \frac{n}{\log^{\overbrace{*\cdots*}^{k-1}} n} S_{2k+1}\left(\log^{\overbrace{*\cdots*}^{k-1}} n\right)$

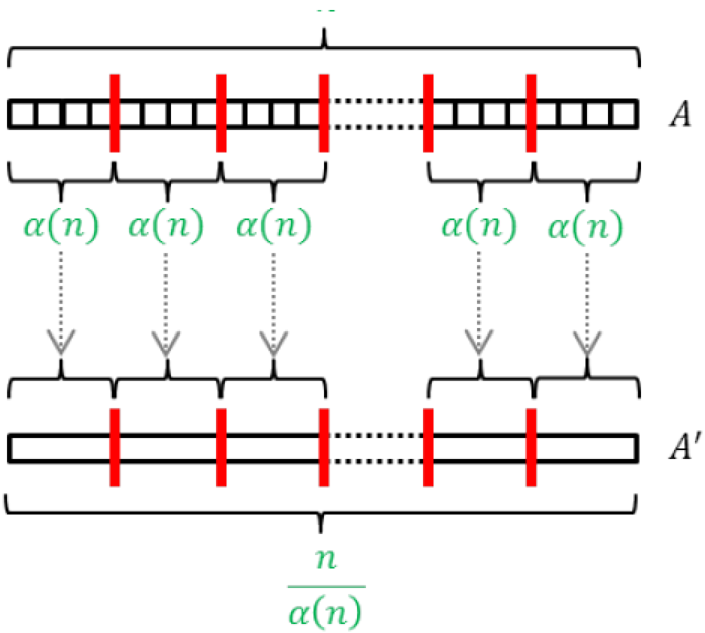
$\le (2k+1)n + \frac{n}{\overbrace{\cdots}^{k-1}} S_{2k+1}\left(\log^{\overbrace{*\cdots*}^{k-1}} n\right) \le (2k+1)n \log^{\overbrace{*\cdots*}^{k}} n$

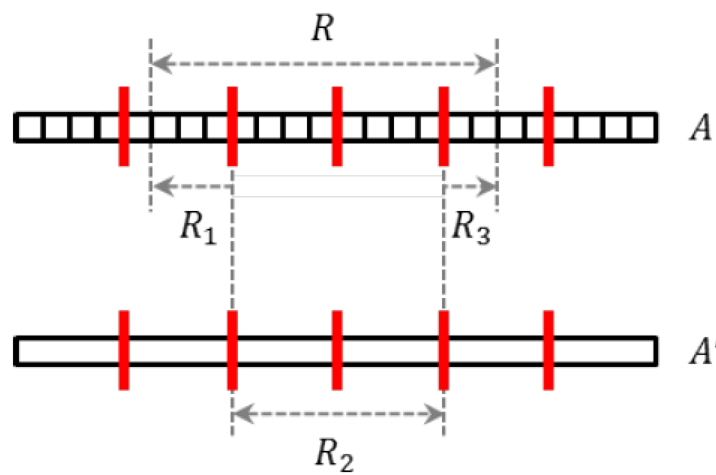
The α Bound

Bound k: $S_{2k+1}(n) \le (2k+1)n \log^{\overbrace{*\cdots*}^{k}} n.$

Putting $k = \alpha(n)$, we have:

Bound α: $S_{2\alpha(n)+1}(n) \le 3(2\alpha(n)+1)n = \mathrm{O}\big(n\alpha(n)\big).$





Linear Space: Use the α-bound to show that the space complexity of the data structure can be reduced to $\mathrm{O}(n)$ while still supporting range queries in $\mathrm{O}\big(\alpha(n)\big)$ time.

Union-Find:
A Disjoint-Set Data Structure

Disjoint Set Operations

A *disjoint-set data structure* maintains a collection of disjoint dynamic sets. Each set is identified by a *representative* which must be a member of the set.

The collection is maintained under the following operations:

MAKE-SET(x): create a new set $\{x\}$ containing only element x. Element x becomes the representative of the set.

FIND(x): returns a pointer to the representative of the set containing x

UNION(x, y): replace the dynamic sets S_x and S_y containing x and y, respectively, with the set $S_x \cup S_y$

Union-Find Data Structure with *Union by Rank* and *Find with Path Compression*

MAKE-SET (x)

1. $\pi(x) \leftarrow x$
2. $rank(x) \leftarrow 0$

LINK (x, y)

1. *if* $rank(x) > rank(y)$ *then* $\pi(y) \leftarrow x$
2. *else* $\pi(x) \leftarrow y$
3. *if* $rank(x) = rank(y)$ *then* $rank(y) \leftarrow rank(y) + 1$

UNION (x, y)

1. LINK (FIND (x), FIND (y))

FIND (x)

1. *if* $x \neq \pi(x)$ *then* $\pi(x) \leftarrow$ FIND ($\pi(x)$)
2. *return* $\pi(x)$

Some Useful Properties of Rank

- If x is not a root then $rank(x) < rank(\pi(x))$
- Node ranks strictly increase along any simple path towards a root
- Once a node becomes a non-root its rank never changes
- If $\pi(x)$ changes from y to z then $rank(z) > rank(y)$
- If the root of x's tree changes from y to z then $rank(z) > rank(y)$
- If x is the root of a tree then $size(x) \geq 2^{rank(x)}$
- If there are only n nodes the highest possible rank is $\lfloor \log_2 n \rfloor$
- There are at most $\frac{n}{2^r}$ nodes with rank $r \geq 0$

Some Useful Properties of Rank

- We will analyze the total running time of m' MAKE-SET, UNION and FIND operations of which exactly n $(\leq m')$ are MAKE-SET
- But each UNION can be replaced with two FIND and one LINK
- Hence, we can simply analyze the total running time of m MAKE-SET, LINK and FIND operations of which exactly n $(\leq m)$ are MAKE-SET and where $m' \leq m \leq 3m'$

Compress

COMPRESS (x, y) { y is an ancestor of x }

1. *if* $x \neq y$ *then* $\pi(x) \leftarrow$ COMPRESS ($\pi(x), y$)
2. *return* $\pi(x)$

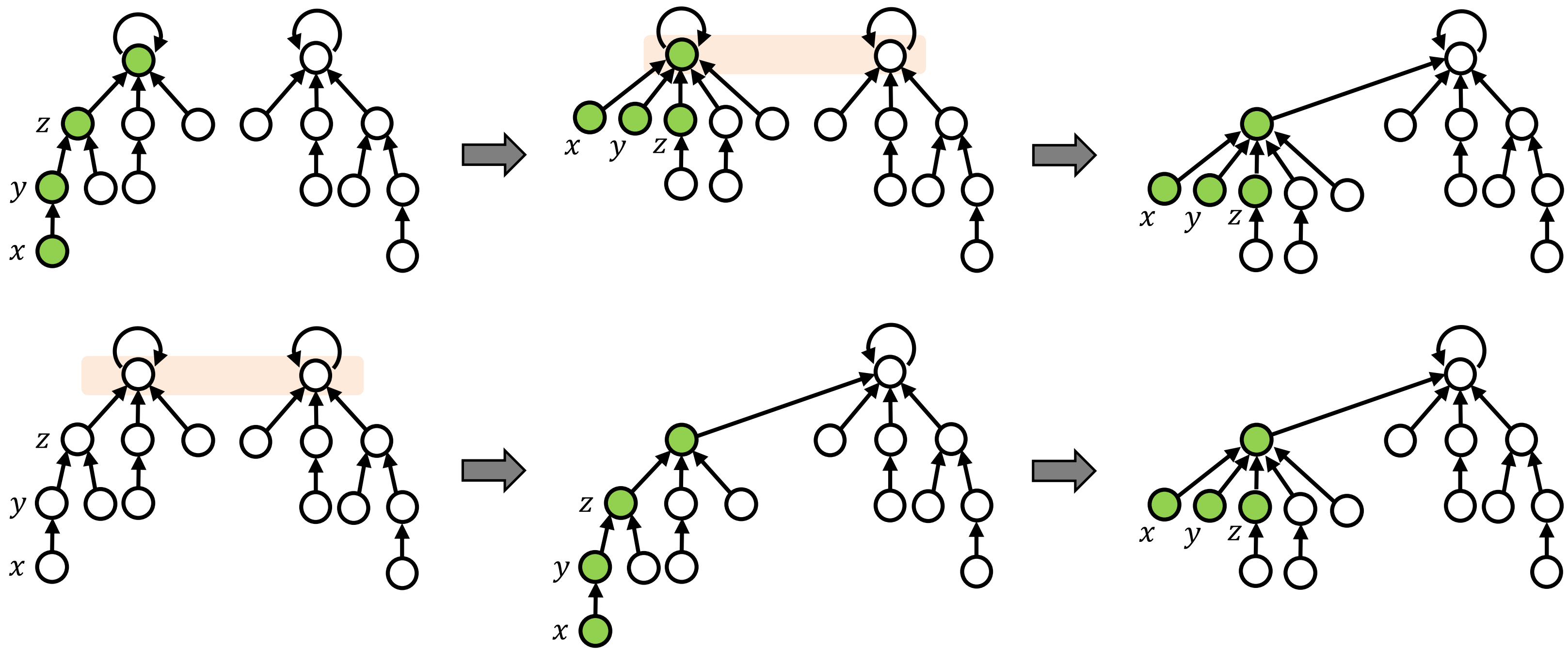
- We will analyze the total running time of m MAKE-SET, UNION and FIND operations of which exactly n $(\leq m)$ are MAKE-SET
- But FIND(x) is nothing but COMPRESS(x, y), where y is the root of the tree containing x
- Hence, we can analyze the total running time of m MAKE-SET, LINK and COMPRESS operations of which exactly n $(\leq m)$ are MAKE-SET

Compress

$\text{COMPRESS}(x, y)$ { y is an ancestor of x }

1. *if* $x \neq y$ *then* $\pi(x) \leftarrow \text{COMPRESS}(\pi(x), y)$
2. *return* $\pi(x)$

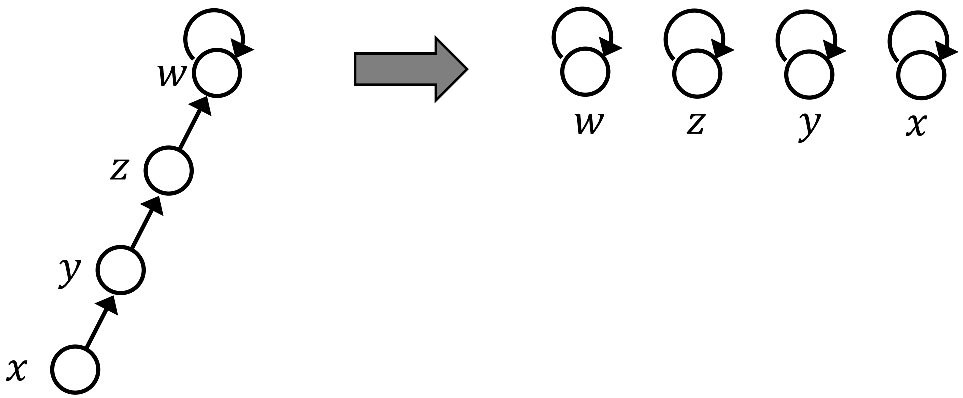
We can reorder the sequence of LINK and COMPRESS operations so that all LINK's are performed before all COMPRESS operations without changing the number of parent pointer reassignments!



Shatter

SHATTER (x)

1. *if* $x \neq \pi(x)$ *then* SHATTER ($\pi(x)$)
2. $\pi(x) \leftarrow x$



Bound 0

Let $T(m, n, r) =$ worst-case number of parent pointer assignments

- during any sequence of at most m COMPRESS operations
- on a forest of n nodes
- with maximum rank r

Bound 0: $T(m, n, r) \leq nr$.

Proof: Since there are at most r distinct ranks, and each new parent of a node has a higher rank than its previous parent, any node can change parents fewer than r times.

Bound 1

Bound 1: $T(m, n, r) \leq m + 2n \log^* r$.

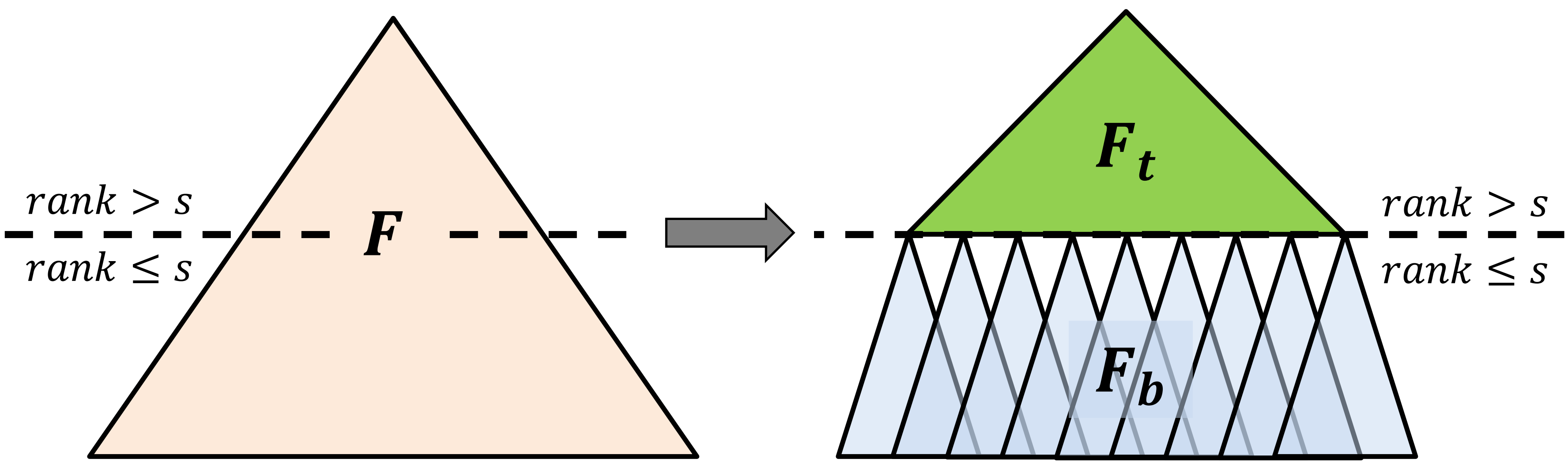
Proof: Let F be the forest, and C be the sequence of COMPRESS operations performed on F.

Let $T(F, C)$ be the number of parent pointer assignments by C in F.

Let s be an arbitrary rank. We partition F into two subforests:

F_b containing all nodes with rank $\leq s$, and

F_t containing all nodes with rank $> s$.



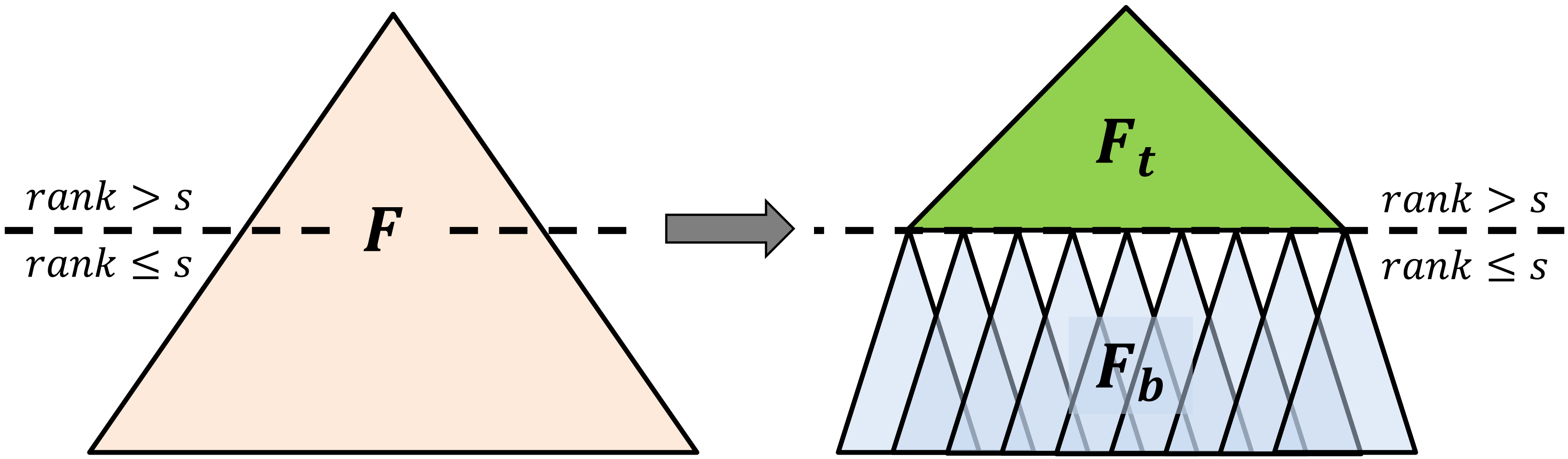
Bound 1

Bound 1: $T(m, n, r) \leq m + 2n \log^* r$.

Proof: Let s be an arbitrary rank. We partition F into two subforests:

F_b containing all nodes with rank $\leq s$, and

F_t containing all nodes with rank $> s$.



Let n_t = #nodes in F_t, and n_b = #nodes in F_b

Let m_t = #COMPRESS operations with at least one node in F_t, and

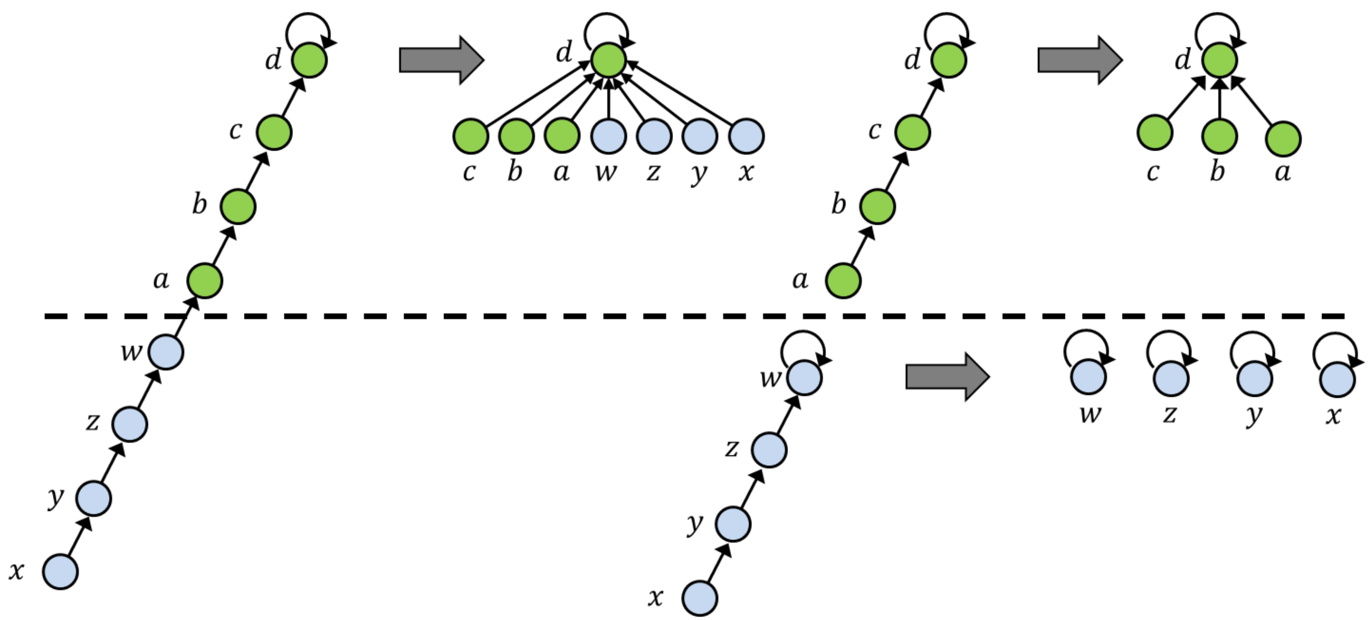
$m_b = m - m_t$

Bound 1

Bound 1: $T(m, n, r) \leq m + 2n \log^* r$.

Proof: The sequence C on F can be decomposed into

- a sequence of COMPRESS operations in F_t, and
- a sequence of COMPRESS and SHATTER operations in F_b



Suppose, this decomposition partitions C into two subsequences

- C_t in F_t, and
- C_b in F_b

Bound 1

Bound 1: $T(m,n,r) \leq m + 2n\log^* r$.

Proof: We get the following recurrence:

$$T(F,C) \leq T(F_t,C_t) + T(F_b,C_b) + m_t + n_b$$

Cost on Left Side	Corresponding Cost on Right Side
node $\in F_t$ gets new parent $\in F_t$	$T(F_t,C_t)$
node $\in F_b$ gets new parent $\in F_b$	$T(F_b,C_b)$
node $\in F_b$ gets new parent $\in F_t$ (for the first time)	n_b
node $\in F_b$ gets new parent $\in F_t$ (again)	m_t

Bound 1

Bound 1: $T(m, n, r) \le m + 2n \log^* r$.

Proof: We get the following recurrence:

$$T(F, C) \le T(F_t, C_t) + T(F_b, C_b) + m_t + n_b$$

Now $n_t \le \sum_{i>s} \frac{n}{2^i} = \frac{n}{2^s}$, and $r_t = r - s < r$.

Hence, using bound 0: $T(F_t, C_t) \le n_t r_t < \frac{nr}{2^s}$

Let $s = \log r$. Then $T(F_t, C_t) < n$.

Hence, $\quad T(F, C) \le T(F_b, C_b) + m_t + 2n$

$\Rightarrow T(F, C) - m \le T(F_b, C_b) - m_b + 2n$

Bound 1

Bound 1: $T(m,n,r) \le m + 2n\log^* r.$

Proof:

We got $T(F,C) - m \le T(F_b, C_b) - m_b + 2n$

Let $T_1(m,n,r) = T(m,n,r) - m$

Then $T_1(m,n,r) \le T_1(m_b, n_b, r_b) + 2n$

$\Rightarrow T_1(m,n,r) \le T_1(m,n,\log r) + 2n$

Solving, $T_1(m,n,r) \le 2n\log^* r$

Hence, $T(m,n,r) \le m + 2n\log^* r$

Bound 2

Bound 2: $T(m, n, r) \le 2m + 3n \log^{**} r$.

Proof: Similar to the proof of bound 1.

But we solve $T(F_t, C_t)$ using bound 1, instead of bound 0!

We fix $s = \log^* r$ (instead of $\log r$ for bound 1)

Then using bound 1: $T(F_t, C_t) \le m_t + 2n_t \log^* r_t$

$$\le m_t + 2\frac{n}{2^{\log^* r}} \log^* r$$

$$\le m_t + 2n$$

Then from $T(F, C) \le T(F_t, C_t) + T(F_b, C_b) + m_t + n_b$, we get

$$T(F, C) \le T(F_b, C_b) + 2m_t + 3n_b$$

Bound 2

Bound 2: $T(m,n,r) \leq 2m + 3n\log^{**} r$.

Proof: Our recurrence:

$$T(F,C) \leq T(F_b, C_b) + 2m_t + 3n_b$$

$$\Rightarrow T(F,C) - 2m \leq T(F_b, C_b) - 2m_b + 3n_b$$

Let $T_2(m,n,r) = T(m,n,r) - 2m$

Then $T_2(m,n,r) \leq T_2(m_b, n_b, r_b) + 3n$

$\Rightarrow T_2(m,n,r) \leq T_2(m,n,\log^* r) + 3n$

Solving, $T_2(m,n,r) \leq 3n\log^{**} r$

Hence, $T(m,n,r) \leq 2m + 3n\log^{**} r$

Bound k

Bound k: $T(m,n,r) \le km + (k+1)n\log^{\overbrace{*\cdots*}^{k}} r$.

Observation: As we increase k:

- the dependency on m increases
- the dependency on r decreases

When $k = \alpha(r)$, we have $\log^{\overbrace{*\cdots*}^{k}} r \le 3$!

Bound α: $T(m,n,r) \le m\alpha(r) + 3(\alpha(r)+1)n$.

The α Bound

Bound $\boldsymbol{\alpha}$: $T(m, n, r) \leq m\alpha(r) + 3(\alpha(r) + 1)n$.

Observing that $r < n$, we have:

Bound $\boldsymbol{\alpha}$: $T(m, n, r) \leq (m + 3n)\alpha(n) + 3n = \mathrm{O}\big((m + n)\alpha(n)\big)$.

Assuming $m \geq n$, we have:

Bound $\boldsymbol{\alpha}$: $T(m, n, r) = \mathrm{O}\big(m\alpha(n)\big)$.

So, amortized complexity of each operation is only $\mathrm{O}\big(\alpha(n)\big)$!