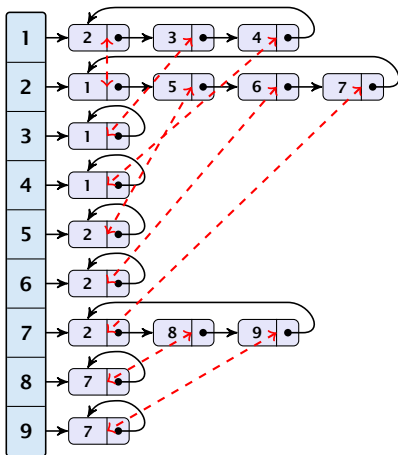
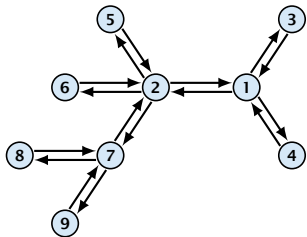


Tree Algorithms



Euler Circuits

Every node v fixes an arbitrary ordering among its adjacent nodes:

$$u_0, u_1, \dots, u_{d-1}$$

We obtain an Euler tour by setting

$$\text{succ}((u_i, v)) = (v, u_{(i+1) \bmod d})$$

Lemma 1

An Euler circuit can be computed in constant time $\mathcal{O}(1)$ with $\mathcal{O}(n)$ operations.

Rooting a tree

- ▶ split the Euler tour at node r
- ▶ this gives a list on the set of directed edges (Euler path)
- ▶ assign $x[e] = 1$ for every edge;
- ▶ perform parallel prefix; let $s[\cdot]$ be the result array
- ▶ if $s[(u, v)] < s[(v, u)]$ then u is parent of v ;

Postorder Numbering

- ▶ split the Euler tour at node r
- ▶ this gives a list on the set of directed edges (Euler path)
- ▶ assign $x[e] = 1$ for every edge $(v, \text{parent}(v))$
- ▶ assign $x[e] = 0$ for every edge $(\text{parent}(v), v)$
- ▶ perform parallel prefix
- ▶ $\text{post}(v) = s[(v, \text{parent}(v))]$; $\text{post}(r) = n$

Level of nodes

- ▶ split the Euler tour at node r
- ▶ this gives a list on the set of directed edges (Euler path)
- ▶ assign $x[e] = -1$ for every edge $(v, \text{parent}(v))$
- ▶ assign $x[e] = 1$ for every edge $(\text{parent}(v), v)$
- ▶ perform parallel prefix
- ▶ $\text{level}(v) = s[(\text{parent}(v), v)]$; $\text{level}(r) = 0$

Number of descendants

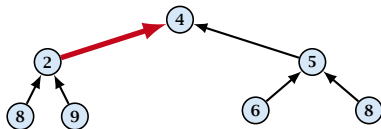
- ▶ split the Euler tour at node r
- ▶ this gives a list on the set of directed edges (Euler path)
- ▶ assign $x[e] = 0$ for every edge $(\text{parent}(v), v)$
- ▶ assign $x[e] = 1$ for every edge $(v, \text{parent}(v))$, $v \neq r$
- ▶ perform parallel prefix
- ▶ $\text{size}(v) = s[(v, \text{parent}(v))] - s[(\text{parent}(v), v)]$

Rake Operation

Given a binary tree T .

Given a leaf $u \in T$ with $p(u) \neq r$ the **rake-operation** does the following

- ▶ remove u and $p(u)$
- ▶ attach sibling of u to $p(p(u))$



We want to apply rake operations to a binary tree T until T just consists of the root with two children.

Possible Problems:

1. we could **concurrently** apply the rake-operation to two siblings
2. we could **concurrently** apply the rake-operation to two leaves u and v such that $p(u)$ and $p(v)$ are connected

By choosing leaves carefully we ensure that none of the above cases occurs

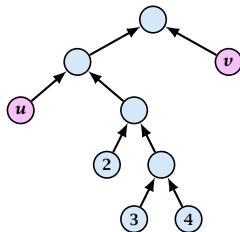
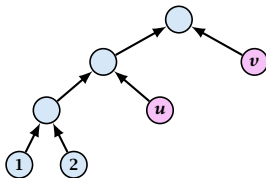
Algorithm:

- ▶ label leaves consecutively from left to right (excluding left-most and right-most leaf), and store them in an array A
- ▶ for $\lceil \log(n + 1) \rceil$ iterations
 - ▶ apply rake to all odd leaves that are left children
 - ▶ apply rake operation to remaining odd leaves (odd at start of round!!!)
 - ▶ A =even leaves

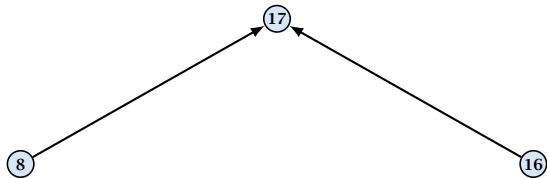
Observations

- ▶ the rake operation does not change the order of leaves
- ▶ two leaves that are siblings do not perform a rake operation in the same round because one is even and one odd at the start of the round
- ▶ two leaves that have adjacent parents either have different parity (even/odd) or they differ in the type of child (left/right)

Cases, when the left edge btw. $p(u)$ and $p(v)$ is a left-child edge.



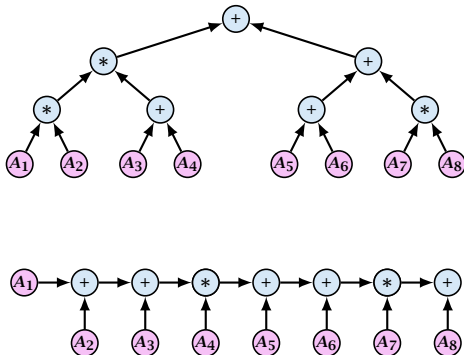
Example



- ▶ one iteration can be performed in constant time with $\mathcal{O}(|A|)$ processors, where A is the array of leaves;
- ▶ hence, **all** iterations can be performed in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ work;
- ▶ the initial parallel prefix also requires time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$

Evaluating Expressions

Suppose that we want to evaluate an expression tree, containing additions and multiplications.



If the tree is not balanced this may be time-consuming.

We can use the rake-operation to do this quickly.

Applying the rake-operation changes the tree.

In order to maintain the value we introduce parameters a_v and b_v for every node that still allows to compute the value of a node based on the value of its children.

Invariant:

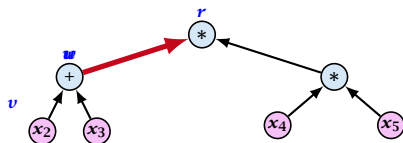
Let u be internal node with children v and w . Then

$$\text{val}(u) = (a_v \cdot \text{val}(v) + b_v) \otimes (a_w \cdot \text{val}(w) + b_w)$$

where $\otimes \in \{*, +\}$ is the operation at node u .

Initially, we can choose $a_v = 1$ and $b_v = 0$ for every node.

Rake Operation



Currently the value at u is

$$\begin{aligned}\text{val}(u) &= (a_v \cdot \text{val}(v) + b_v) + (a_w \cdot \text{val}(w) + b_w) \\ &= x_1 + (a_w \cdot \text{val}(w) + b_w)\end{aligned}$$

In the expression for r this goes in as

$$\begin{aligned}a_u \cdot [x_1 + (a_w \cdot \text{val}(w) + b_w)] + b_u \\ = \underbrace{a_u a_w}_{a'_w} \cdot \text{val}(w) + \underbrace{a_u x_1 + a_u b_w + b_u}_{b'_w}\end{aligned}$$

If we change the a and b -values during a rake-operation according to the previous slide we can calculate the value of the root in the end.

Lemma 2

We can evaluate an arithmetic expression tree in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$ regardless of the height or depth of the tree.

By performing the rake-operation in the reverse order we can also compute the value at each node in the tree.

Lemma 3

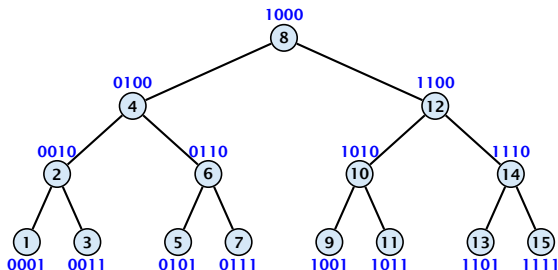
We compute *tree functions* for arbitrary trees in time $\mathcal{O}(\log n)$ and a linear number of operations.

proof on board...

In the **LCA (least common ancestor) problem** we are given a tree and the goal is to design a data-structure that answers LCA-queries in constant time.

Least Common Ancestor

LCAs on complete binary trees (inorder numbering):

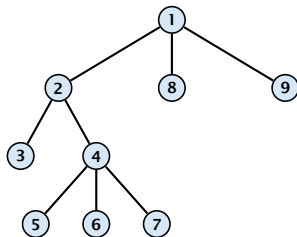


The least common ancestor of u and v is

$$z_1 z_2 \dots z_i 1 0 \dots 0$$

where z_{i+1} is the first bit-position in which u and v differ.

Least Common Ancestor



nodes

1	2	3	2	4	5	4	6	4	7	4	2	1	8	1	9	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

levels

0	1	2	1	2	3	2	3	2	3	2	1	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$\ell(v)$ is index of first appearance of v in node-sequence.

$r(v)$ is index of last appearance of v in node-sequence.

$\ell(v)$ and $r(v)$ can be computed in constant time, given the node- and level-sequence.

Least Common Ancestor

Lemma 4

1. *u is ancestor of v iff $\ell(u) < \ell(v) < r(u)$*
2. *u and v are not related iff either $r(u) < \ell(v)$ or $r(v) < \ell(u)$*
3. *suppose $r(u) < \ell(v)$ then $\text{LCA}(u, v)$ is vertex with minimum level over interval $[r(u), \ell(v)]$.*

Range Minima Problem

Given an array $A[1 \dots n]$, a **range minimum query** (ℓ, r) consists of a left index $\ell \in \{1, \dots, n\}$ and a right index $r \in \{1, \dots, n\}$.

The answer has to return the index of the minimum element in the subsequence $A[\ell \dots r]$.

The goal in the **range minima problem** is to preprocess the array such that range minima queries can be answered quickly (constant time).

Observation

Given an algorithm for solving the range minima problem in time $T(n)$ and work $W(n)$ we can obtain an algorithm that solves the LCA-problem in time $\mathcal{O}(T(n) + \log n)$ and work $\mathcal{O}(n + W(n))$.

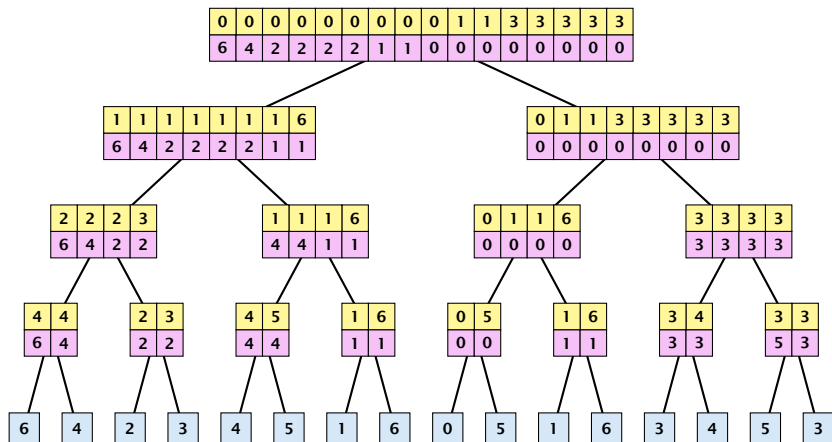
Remark

In the sequential setting the LCA-problem and the range minima problem are equivalent. This is not necessarily true in the parallel setting.

For solving the LCA-problem it is sufficient to solve the **restricted range minima problem** where two successive elements in the array just differ by $+1$ or -1 .

Prefix and Suffix Minima

Tree with prefix-minima and suffix-minima:



- ▶ Suppose we have an array A of length $n = 2^k$
- ▶ We compute a complete binary tree T with n leaves.
- ▶ Each internal node corresponds to a subsequence of A . It contains an array with the prefix and suffix minima of this subsequence.

Given the tree T we can answer a range minimum query (ℓ, r) in constant time.

- ▶ we can determine the LCA x of ℓ and r in constant time since T is a complete binary tree
- ▶ Then we consider the suffix minimum of ℓ in the left child of x and the prefix minimum of r in the right child of x .
- ▶ The minimum of these two values is the result.

Lemma 5

We can solve the range minima problem in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n \log n)$.

Reducing the Work

Partition A into blocks B_i of length $\log n$

Preprocess each B_i block separately by a sequential algorithm so that range-minima queries within the block can be answered in constant time. (**how?**)

For each block B_i compute the minimum x_i and its prefix and suffix minima.

Use the previous algorithm on the array $(x_1, \dots, x_{n/\log n})$.

Answering a query (ℓ, r) :

- ▶ if ℓ and r are from the same block the data-structure for this block gives us the result in constant time
- ▶ if ℓ and r are from different blocks the result is a minimum of three elements:
 - the suffix minimum of entry ℓ in ℓ 's block
 - the minimum among $x_{\ell+1}, \dots, x_{r-1}$
 - the prefix minimum of entry r in r 's block