Lists and Trees (Ch. 3)

List Ranking Problem

Input: A linked list of \( n \) nodes specified by a successor array \( S[1..n] \), i.e., \( S(i) = \) index of node \( i \)'s successor and successor of last node (tail) is 0.
Output: A distance (rank) array \( R[1..n] \) s.t. \( R(i) = \) dist from node \( i \) to end of list.

Pointer Jumping Algorithm (we've seen before)

Alg 1

\[
\text{for } 1 \leq i \leq n \text{ pardo} \quad /\!* \text{init } R[\cdot] \text{ array }*/
\]

\[
\text{if } S(i) = 0 \\
\quad \text{then } R(i) := 1 \\
\quad \text{else } R(i) := 0
\]

\end pardo

\[
\text{for } 1 \leq i \leq n \text{ pardo} \quad /\!* \text{do ptr jumping }*/
\]

\[
Q(i) := S(i) \\
\text{while } (Q(i) \neq 0) \text{ AND } (Q(Q(i)) = 0) \text{ do}
\]

\[
R(i) := R(i) + R(Q(i)) \\
Q(i) := Q(Q(i))
\]

\endwhile

\end pardo

\end /* Alg 1 */

\[
T(n) = O(\log n) \text{ b/c while loop "halves" dist to end each iter} \\
W(n) = O(n\log n) \text{ b/c do } O(\log n) \text{ work each } n \text{ itts} \\
CREW b/c many procs reach tail simultaneously (can make CREW)
\]
Sequentially, problem takes $O(n)$ time (work)
So Alg 1 is not work-optimal

Strategy for work-optimal list ranking
0. "Shrink" list to only $O(n/\log n)$ elts
2. Apply Alg 1 to "short" list
   - $T(n/\log n) = O(\log (n/\log n)) = O(\log n)$
   - $W(n/\log n) = O((n/\log n) \cdot \log (n/\log n)) = O((n/\log n)^2) = O(n)$
3. Restore original list & rank nodes removed in 0.

We'll look at how to do step 1 in $O(n)$ work,
Step 3 is pretty much just reversal of step 1.

Independent Sets

Basic Idea: To shrink list $L$, remove an independent set of elts - iteratively - until $L$ is small enough

Definition: A set $I$ of elts in independent if whenever $i \in I$, then $S(i) \neq I$

Removing node $i$ from $L$
- adjust successor ptr of predecessor of $i$
- update $R$ value of predecessor of $i$ (to reflect dist/rank info known so far about elt $i$)
- keep info re removed nodes so can add back later (put in extra $U[1..n]$ array)
Subroutine Remove Nodes in Independent Set

**Input:** arrays \( S[i..n] \) and \( P[i..n] \) for succ and pred of elts in list
- independent set \( I \) s.t. \( P(i), S(i) \neq 0 \) for all \( i \in I \)
- array \( R[i..n] \) w/ \( R \) value of each node (initially all 1 except tail = 0)

**Output:** list obtained after removal of nodes in \( I \)
- updated \( R \) value
- updated array \( U[i..n] \)

1. Assign consecutive "serial numbers" \( N(i) \) to elts of \( I \)

2. for all \( i \in I \) pardo
   \[
   U(N(i)) := (i, S(i), R(i)) \quad \text{/* save so can get back later */}
   R(P(i)) := R(P(i)) + R(i) \quad \text{/* update } R \text{ for Predecessor */}
   S(P(i)) := S(i) \quad \text{/* unlink } i \text{ */}
   P(S(i)) := P(i)
   \]

EX

```
checkmark 7 8 1 2 3 6 5 4
  [1] [1] [2] [2] [2] [3] [0] [0]
```

Complexity:

Step 1: Time \( O(\log(n)) \), Work \( O(n) \) by Prefix Sums computation
where assign 1 to nodes in \( I \) and 0 to others

Step 2: Time \( O(1) \), Work \( O(n) \)
Determining an Independent Set

Via Coloring

Recall a $k$-coloring maps nodes $\to \{0, 1, \ldots, k-1\}$ s.t.
no two adj nodes get same color

Given a $k$-coloring, a node $e$ is a local minimum (max)
with the $k$-coloring if the color of $e$ is
smaller (larger) than the colors of its pred + succ.

Lemma Given a $k$-coloring of the nodes of a list $L$
of size $n$, the Set $I$ of local minima (maxima)
is an independent set of size $\Omega(\sqrt{n/k})$.
- I can be identified in $O(1)$ time w/ $O(n)$ work.

Proof

- Let $u_1 + u_2$ be two consecutive local minima in $L$
- nodes between $u_1 + u_2$ must form increasing seq.

\[ \Rightarrow \text{at most } 2k-3 \text{ nodes between } \left( \text{all diff colors except color } u_1 \text{ or color } u_2 \right) \]

\[ \subseteq \text{local minima form set size } \Omega(\sqrt{n/k}) \]
and also for max

Complexity

- Determine if node local min or max in $O(1)$ time
w/ $O(n)$ work (EREW PRAM, read right then left+
Lemma: set $I$ of local minima of $K$-coloring form an \( IS \) of size \( \Omega(\sqrt[3]{k}) \)

More Precisely: \( |I| \geq \left\lfloor \frac{n}{2k-2} \right\rfloor \geq \frac{n}{2k-1} \) (Since at most 2k-3 nodes between consecutive local minima)

Work-Optimal List Ranking Alg

input: linked list $L$ w/ $S(i)$ + $P(i)$ arrays, $n$ elts
output: ranks $R(i)$ of all $n$ input nodes
1. init $R(i)$ values, $n_0 := n$, $k := 0$
2. while $n_k > \frac{n}{\log n}$ do /* reduce size of $L$ to $\leq \frac{n}{\log n}$/
   (a) $k := k + 1$
   (b) 3-color list + identify $I$ (local minima)
   (c) Remove nodes in $I$ from list + update $R(i)$ values
   (d) $n_k :=$ size of remaining list
   (e) Compact elts into $n_k$ consecutive memory locations
endwhile
3. Apply Pointer Jumping list ranking alg to "short" list
4. Restore orig. list & fix-up $R(i)$ values ("Reverse" Step 2)

Complexity

Step 1: \( O(1) \) time, \( O(n) \) work \ EREW PRAM
Step 3: \( O(\log n_k) = O(\log(\frac{n}{\log n})) = O(\log n) \) time
\( O(n_k \log n_k) = O(\frac{n}{\log n} \cdot \log n) = O(n) \) work
CREW PRAM
(But can be adapted to run on EREW PRAM by making sure only 1 guy reads succ of tail each iter)

by doing this make each elt read succ every time so don't do succ(read)
Steps 2 (πk):

\[
\text{# iters of while loop}
\]

* by lemma \( |I| \geq \frac{n_k}{2^{3-k}} = \frac{n_k}{5} \) \( \Rightarrow n_{k+1} \leq \frac{4}{5} n_k \)

\( \Rightarrow n_k \leq \left( \frac{4}{5} \right)^k n \)

Therefore, \( n_k \leq \frac{n}{\log n} \) after \( O(\log \log n) \) iterations.

Each iter of while loop:

(b) \( O(\log n_k) \) time, \( O(n_k) \) work, EREW PRAM

(c) (Removing I) \( O(\log n_k) \) time, \( O(n_k) \) work, EREW PRAM

(e) (Compacting) can be done by prefix sums computation by labeling nodes in \( I \) w/ 0 + others w/ 1.

- \( O(\log n_k) \) time, \( O(n_k) \) work, EREW PRAM

Total for each iter: \( O(\log n_k) \) time, \( O(n_k) \) work.

Total time:

\[
T(n) = O \left( \sum_{K=0}^{\log \log n} \log n_k \right) = O \left( \sum_{K=0}^{\log \log n} (\frac{4}{5})^k n \right) = O \left( \sum_{K=0}^{\log \log n} \log n + \log \left( \frac{4}{5} \right)^k \right) = O \left( \log n \log \log n \right)
\]

Total work: (compaction (e) is important for this)

\[
W(n) = O \left( \sum_{K=0}^{\log \log n} n_k \right) = O \left( \sum_{K=0}^{\log \log n} (\frac{4}{5})^k n \right) = O \left( n \sum_{K=0}^{\log \log n} (\frac{4}{5})^k \right) = O(n)
\]

EREW PRAM (w/ modified Pointer Jumping List Ranking alg)

**Note:** Book shows \( O(\log n) \) time, \( O(n) \) work, EREW PRAM.

We'll use this result, but not prove it...
Trees

Euler Tour Technique

- $T = (V, E)$ is a tree (undirected)
- $T' = (V, E')$ is directed graph (digraph) obtained by replacing each $(x, y) \in E$ w/ arcs $(x, y) \rightarrow (y, x)$

Fact: $T' = (V, E')$ is Eulerian since indegree = outdegree for every $v \in V$. Thus $T'$ has a directed circuit that traverses every arc in $E'$ exactly once.

Why do we care?

An Euler circuit for $T'$ can be used to compute many functions on tree $T$ efficiently in parallel.

- Rooting a tree at some vertex $v \in V$, i.e., computing \text{parent}(v) for each $v \in V$ when $T$ rooted at $v$.

- Postorder numbering, i.e., given rooted tree, compute postorder numbering for other nodes in $T$.

- Computing levels, i.e., given rooted tree, compute distance of each $v \in V$ from root.

- Many more...
Euler Circuit for \( T' = (V, E') \)

- \( S(e) \) is arc following \( e \) in circuit, for every \( e \in E' \)
- \( S(e) \) can be defined locally (important for parallelization)
  - for each \( v \in V \) fix ordering of vertices adjacent to \( v \) in \( T \)
    \[ \text{adj}(v) = (u_0, u_1, \ldots, u_{d-1}) \]
    \( d = \text{degree}(v) \)
  - for each arc \( e = (u_i, v) \)
    \[ S(e) = (v, u_{(i+1) \mod d}) \]

\[ S(1,4) = (4,2) \]
\[ S(2,4) = (4,3) \]
\[ S(3,4) = (4,8) \]
\[ S(4,1) = (1,4), S(4,2) = (2,4), S(4,3) = (3,4) \]

Eulerian Circuit: \( \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 4 \)

\textbf{Lemma:} The function \( S(c) \) defines EC for \( T' = (V, E') \)

\textbf{Proof:}

First note each arc \( e \in E' \) is assigned a unique successor.
Thus, we need only argue that \( S(c) \) produces a single circuit.
The proof is by induction on \( n \), the number of vertices, in \( T' \).

\underline{basis}: \( n = 2 \), trivial

\underline{assume}: \( S(c) \) defines EC for all trees \( w/ < n \) nodes.

Now, consider a tree \( T \) with \( n \) nodes:

- let \( v \) be a leaf in \( T \) and \( u \) a vertex adjacent to \( v \)
- let \( \text{adj}(u) = (\ldots, v', v, v'', \ldots) \)
- by def of \( S(c) \):

  \[ S(v', u) = (u, v) \]
  \[ S(u, v) = (v, u) + S(v, u) = (u, v') \]

- if we delete \( v \) only change to \( S(c) \) is \( S(v', u) = (u, v'') \)
  + by ind. hyp. \( T - v \) has EC \( C \) defined by \( S(c) \)
- Thus, \( C - (5) + (6) + (2) \) is EC for \( T' \) (i.e. we just splice in \( v \)?)
Computing an Eulerian Circuit - SC) Values

Assume \( T = (V,E) \) represented by adjacency list \( L[1..n] \)
- \( \text{ptr} \) from arc \( (x,y) \) to \( (y,x) \)
- \( \text{ptr} \) from last to first elt of each list \( L[i] \)

e.g.,

\[
\begin{align*}
L & = \{(4,2), (4,3), (4,1), (1,4), (2,4), (3,4)\} \\
EC: & = 1 \rightarrow 4 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 4
\end{align*}
\]

Algorithm
- Assign one processor to each arc \( e = (x,y) \in E' \) (i.e., to each element of adjacency lists)
- Set \( S(x,y) = \text{arc following } (y,x) \text{ in } L[y] \)

Complexity
- \( O(1) \) time, \( O(n) \) work, EREW PRAM

Alternative Representation
- if each "list" \( L[i] \) is an array, can do in same time + work so long as know degree of each node.
  \( \Rightarrow \) CREW PRAM unless duplicate degree w/ each elt of \( L[i] \)
  \( \Rightarrow \) EREW PRAM if each elt has index of successor in \( L[i] \)
Rooting a Tree at any desired vertex \( r \)

- For each \( v \neq r \in V \) determine a unique parent \( p(v) \) when \( T \) is rooted at \( r \).

\[
\begin{align*}
\text{Time} & \quad \text{Work} \\
O(1) & \quad O(n) \quad \text{1. Compute Euler circuit } C \text{ of } T \\
O(1) & \quad O(l) \quad \text{2. Break } C \text{ into list by setting } S(u_{d-1}, r) = 0 \\
O(l) & \quad O(m) \quad \text{3. Assign wt. 1 to each arc and apply parallel prefix alg on the list given by } s() \text{ function} \\
O(1) & \quad O(n) \quad \text{4. For each arc } (x, y) \text{ do} \\
& \quad \quad \text{if } (\text{prefix-sum}(x, y) < \text{prefix-sum}(y, x)) \\
& \quad \quad \quad \text{then } p(y) := x \\
& \quad \quad \text{end if} \\
& \quad \text{end for}
\end{align*}
\]

REW PRAM

\[
\text{EC: } \quad 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \\
S(1, 4) = (4, 2) \\
S(2, 4) = (4, 3) \\
S(3, 4) = (4, 1) \\
S(4, 1) = (1, 4), S(4, 2) = (2, 4), S(4, 3) = (3, 4)
\]

\[
\text{root } @ 2: \text{ Set } S(4, 2) = 0 \\
\begin{align*}
1 / 1 & \rightarrow 1 / 2 \\
2 & \rightarrow 4 \\
3 & \rightarrow 4 \\
4 & \rightarrow 1 \\
5 & \rightarrow 4 \\
6 & \rightarrow 2 \\
\end{align*}
\]

- \( ps(2, 4) = 1 < ps(4, 2) = 6 \Rightarrow p(4) := 2 (1, 4) \)
- \( ps(4, 1) = 4 < ps(1, 4) = 5 \Rightarrow p(1) := 4 (4, 1) \)
- \( ps(4, 3) = 2 < ps(3, 4) = 3 \Rightarrow p(3) := 4 (4, 2) \)

Why does this work?

List obtained in (2) is Depth-First-Search traversal of \( T \). So lower value is for first time we reach a vertex \( v \) higher when we return (parent to child).
Post-order Numbering/Traversal of tree $T$ rooted at $r$

Postorder ($r$)

for $x :=$ leftmost to rightmost child of $r$ do
  Postorder ($x$)
end for

visit ($x$)
end /* postorder $x$*/

Eq.

\[
\begin{array}{cccc}
\text{EP} & \text{Weight} & \text{Prefix Sums} \\
(1,2) & 0 & 0 \\
(2,5) & 0 & 0 \\
(5,2) & 1 & 1 \Rightarrow \text{post}(5) = 1 \\
(2,6) & 0 & 1 \\
(6,2) & 1 & 2 \Rightarrow \text{post}(6) = 2 \\
(2,7) & 0 & 2 \\
(7,8) & 0 & 2 \\
(8,7) & 1 & 3 \Rightarrow \text{post}(8) = 3 \\
(7,9) & 0 & 3 \\
(9,7) & 1 & 4 \Rightarrow \text{post}(9) = 4 \\
(7,2) & 1 & 5 \Rightarrow \text{post}(7) = 5 \\
(2,1) & 1 & 6 \Rightarrow \text{post}(2) = 6 \\
(1,3) & 0 & 6 \\
(3,1) & 1 & 7 \Rightarrow \text{post}(3) = 7 \\
(1,4) & 0 & 7 \\
(4,1) & 1 & 8 \Rightarrow \text{post}(4) = 8 \\
\end{array}
\]

- define left-to-right order
  of children of $v$ as
  order visited in EP

- EP visits each node several
  times: + 1st time $(p(v), v)$
  + last time $(v, p(v))$

$\Rightarrow$ ordered list obtained by keeping last occurrence of $v$
(Each vertex give postorder numbering).

Time | Work
--- | ---
0(n) | 0(n)
0(n) | 0(n)

Algorithm (rooted tree $T$)

1. Compute EP for $T$ rooted at $r$
2. For each $v \neq r$
   \[ W(v, p(v)) := 1 \]
   \[ W(p(v), v) := 0 \]
3. Perform Prefix Sums w/ wts from 2
4. For each $v \neq r$
   \[ \text{post}(v) := \text{prefix-sum}(v, p(v)) \]
   // last occurrence of $v$
5. $\text{post}(r) := n$

Complexity

Dominated by Step 2
O($\log n$) time, O($n$) Work
EREW PRAM
Computing Vertex Level for rooted tree $T$

$\text{level}(v) =$ distance (# edges) from $v$ to $r$

**Example**

- $E_P$: $(1,2)$, $(2,5)$, $(5,2)$, $(2,6)$, $(6,7)$, $(7,6)$, $(6,2)$, $(2,1)$, $(1,3)$, $(3,1)$, $(1,4)$, $(4,1)$
- Weight: $1$, $1$, $-1$, $1$, $1$, $-1$, $1$, $1$, $1$, $1$, $-1$
- Prefix Sums: $1 \Rightarrow \text{level}(2) = 1$, $2 \Rightarrow \text{level}(5) = 2$, $3 \Rightarrow \text{level}(6) = 2$, $3 \Rightarrow \text{level}(7) = 3$, $3 \Rightarrow \text{level}(3) = 1$, $3 \Rightarrow \text{level}(4) = 1$

**Idea**

- Level increases when $E_P$ goes $(p(v), v)$
- Level decreases when $E_P$ goes $(v, p(v))$

**Algorithm** (rooted tree $T$)

1. Compute $E_P$ for $T$ rooted at $r$
2. For each $v \neq r$
   - $w(p(v), v) := +1$
   - $w(v, p(v)) := -1$
3. Perform Prefix Sums $w$ wts from 2
4. For each $v \neq r$
   - $\text{level}(v) := \text{prefix-sums}(p(v), v)$
5. $\text{level}(r) := 0$

**Complexity**

$O(\log n)$ time, $O(n)$ work - Step 3

EREW PRAM
Computing # descendants

<table>
<thead>
<tr>
<th>EP</th>
<th>Weight</th>
<th>Prefix Sums</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(2,5)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(5,2)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(2,6)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(6,7)</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>(7,6)</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>(6,2)</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>(2,1)</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>(1,3)</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>(3,1)</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>(1,4)</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>(4,1)</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

IDEA

- EP is DFS Traversal, so visits v's whole subtree before leave on arc (v, p(v)).

**Algorithm (rooted tree T)**

1. Compute EP for T rooted at r
2. for each v ≠ r do
   - \( w(p(v), v) := 0 \)
   - \( w(v, p(v)) := 1 \)
3. Perform Prefix Sums w/ wts from 2
4. for each v ≠ r do
   - \( \text{size}(v) := \text{prefixsum}(v, p(v)) - \text{prefixsum}(p(v), v) \)
5. \( \text{size}(r) := n \)

Complexity

- \( O(\log n) \) time, \( O(n) \) work - Step 3
- EREW PRAM
Lowest Common Ancestor Problem

given rooted tree $T$, the lowest common ancestor of two nodes $u + v$ in $T$, $\text{lca}(u,v)$, is the lowest node $w$ that is ancestor of both $u + v$.

- often we want to perform many lca queries
  $\Rightarrow$ preprocess $T$ so we can answer them quickly - $O(1)$ time

$T$ is a path (list)

lca's can be computed in $O(1)$ time if we know the distance of each node from root (preprocessing = list ranking)

$T$ is a complete binary tree

lca's can be computed in $O(1)$ time if we know inorder traversal number of each node

- $\text{lca}(x,y) \Rightarrow$ express $x + y$ in binary (number bits left to right)
  - let $i$ be the 1st bit $x + y$ differ
    i.e., they agree $b_1, b_2 ... b_{i-1}$
  - $\text{lca}(x,y) = b_1, b_2 ... b_{i-1} , 10 ... 0$

E.g.

$3 = 0011$  $11 = 1011$  $\text{lca}(3,11) = 1000 = 8 \checkmark$

$3 = 0011$  $5 = 0101$  $\text{lca}(3,5) = 0100 = 4 \checkmark$

$9 = 1001$  $11 = 1011$  $\text{lca}(9,11) = 1010 = 10 \checkmark$
T is a general tree

Use Euler Tour Technique + Range Minima (later)

1. Compute Euler Tour C of T
2. Form an array A of vertices by replacing each arc (x,y) in ET C by vertex y + insert root in A[0]
   - |A| = 2n-1, |V| = n
3. For each v ∈ V compute
   (i) level(v) in T
   (ii) ll(v) = index 1st occur. of v in A
   (iii) rl(v) = index last occur. of v in A
4. Form an array B[0..(2n-2)] where
   B[i] = level(A[i+1]) in T

(l,v)  ET  A  Index  B
(1,17)  (1,2)  1  1  0
(1,14)  (2,5)  2  2  1
(3,2)  (5,1)  2  3  2
(2,1)  (2,16)  3  4  2
(4,7)  (6,16)  4  5  2
(6,7)  (7,6)  5  6  2
(7,6)  (6,1)  6  7  3
(6,1)  (8,6)  7  8  3
(6,1)  (6,9)  8  9  2
(9,6)  (9,6)  9  10  3
(7,6)  (9,6)  6  11  2
(6,1)  (6,2)  2  12  1
(2,1)  (2,1)  1  13  0
(1,13)  (1,3)  3  14  1
(3,1)  (1,14)  1  15  0
(1,14)  (4,1)  4  16  1
(4,1)  (4,1)  1  17  0
Lemma: Let \( u \neq v \in V \)

1. \( u \) is an ancestor of \( v \) \( \iff \) \( l(u) < l(cv) < r(u) \)
2. \( u + v \) are not related \( \iff \) \( r(u) < l(v) \) or \( r(v) < l(u) \)
3. if \( r(u) < l(v) \) then \( \text{lca}(u,v) \) is vertex in \( A[r(u), l(v)] \) with minimum level \# \text{ (Range Minima in } B[r(u), l(v)] \text{ )}

Proof: Based on fact ET is DFS traversal of \( T \). 

Total Preprocessing Time for LCA queries

- Euler tour computations (need tour + level)
  \( O(\log n) \) time, \( O(n) \) work, EREW PRAM
- Range Minima Preprocessing for B array
  \( O(\log n) \) time, \( O(n) \) work, EREW PRAM

\( \Rightarrow \) \( O(\log n) \) time, \( O(n) \) work, EREW PRAM

Answering LCA queries

- reduced to a range minima query in B array.
- \( O(1) \) time
Range Minima Problem

given array $B[1..n]$, $n=2^k$, of elts from linearly ordered set, preprocess $B$ so that given any two integers $1 \leq i < j \leq n$ we can determine $\min (b_i, b_{i+1}, \ldots, b_j)$ in $O(1)$ seq. time.

Simple Idea (that doesn't work)

- at each internal node $v$ store $\min$ of elts that are at leaves of subtree rooted at $v$

• OK if $RM$ values $i < j$ correspond exactly to range of leaves in some subtree, e.g. $i=5$, $j=8$
  (for now assume know which internal node "covers" $(b_i, b_{i+1}, \ldots, b_j)$)
• Otherwise, may need to look along leaf to root path and take $O(\log n)$ time, e.g. $i=2$, $j=7$

"Patching up" this (wrong) approach

define: prefix minima of $B$ as elts $P = (p_1, p_2, \ldots, p_n)$ s.t.
\[ p_i = \min (p_1, p_2, \ldots, p_i) \]

suffix minima of $B$ as elts $S = (s_1, s_2, \ldots, s_n)$ s.t.
\[ s_i = \min (s_i, s_{i+1}, \ldots, s_n) \]

⇒ at each internal node $v$ store
- $P_v$ - prefix minima of leaves in subtree rooted at $v$
- $S_v$ - suffix minima of leaves in subtree rooted at $v$
Answering RM Query \( 1 \leq i < j \leq n \), i.e. \( \min(b_i, b_{i+1}, \ldots, b_j) \)

- let \( v \) be lowest level node whose subtree contains \( B_v = (b_k, b_{k+1}, \ldots, b_2) \) s.t. \( k \leq i \) and \( j \leq l \)
- let \( u \) and \( w \) be left and right children, resp., of \( v \)
- subarrays \( B_u + B_w \) partition \( B_v \) into
  \( B_u = (b_k, \ldots, b_p) \) + $\$
  \( B_w = (b_{p+1}, \ldots, b_j) \) s.t. \( i \leq p < j \)
- \( \min(b_i, \ldots, b_j) = \min[\min(b_i, \ldots, b_p), \min(b_{p+1}, \ldots, b_j)] \)

\( \Rightarrow \) Can answer RM query in \( O(1) \) seq. time given \( P_v + S_v \) arrays

(* assuming identify \( v \) in \( O(1) \) time, later \)

\( \text{and already know \( B \) array.} \)

**Algorithm: Range Minima Preprocessing**

(Compute \( P_v + S_v \) arrays)

1. **for** \( 1 \leq i \leq n \) **par do**
   
   \( P_{0,i} := B(i) \) (\( P_{i,j} \) is Prefix Min for node \( j \) on \( i^{th} \) level)
   
   \( S_{0,i} := B(i) \) (\( S_{i,j} \) is Suff Min for node \( j \) on \( i^{th} \) level)

   **end par do**

2. **for** \( h := 1 \) to \( \log n \) **do**
   
   **for** \( 1 \leq j \leq n/2^h \) **par do**
   
   \( P_{h,j} := \text{PMerge}(P_{h-1,2j-1}, P_{h-1,2j}) \)

   \( S_{h,j} := \text{SMerge}(S_{h-1,2j-1}, S_{h-1,2j}) \)

   **end par do**

**end for**

\( \text{(* alg)} \)
PMerge (SMerge analogous)

1. Copy $P_{h-1, 2^{i-1}}$ to 1st half of $P_{hi}$
2. Copy $P_{h-1, 2^i}$ to 2nd half of $P_{hi}$
3. Replace each elt of 2nd half of $P_{hi}$ w/ min of itself & last elt in first half $P_{hi}$.

⇒ $O(1)$ time, $O(2^h)$ work
CREW PRAM (Step 3)

Total Complexity for Preprocessing

Step 1: $O(1)$ time, $O(n)$ work

Step 2:
- Each level $h$ of tree:
  - $O(1)$ time
  - $O(\frac{1}{2^h} \cdot 2^h) = O(n)$ work
- $O(\log n)$ levels
- Total: $O(\log n)$ time, $O(n \log n)$ work
CREW PRAM

Modifying for EREW PRAM

- Only need CR for Step 3 in PMerge + SMerge
- Have each element in $P_{ij}$ & $S_{ij}$ arrays hold minimum value in subarray in addition to prefix or suffix min.
  - then can do Step 3 in ER (each elt compares to diff copy)
  - Can update min elt in new subarray using this info too
  - Time & work bounds remain the same.
Reducing Preprocessing Work to $O(n)$

- apply accelerated cascading technique (p. 74)
  - reduce problem size to $O(n \log n)$
  - then use work-suboptimal method we've just seen

- slightly complicates query process (not too much)
- time still $O(\log n)$

Note: Faster alg possible, $O(\log \log n)$ time (Ex. 2.19)

- can also be made $O(n)$ work w/ accelerated cascading
- requires CRCW PRAM
Tree Contraction

- shrink tree into a single vertex by successively
  - merging a leaf w/ its parent
  - merging a degree 2 node w/ its parent

*expression evaluation*

expression represented as binary tree
- leaves are constants
- internal nodes are arithmetic operators

\[(a+b) \cdot c\]

- when contract expression tree, incorporate values of shrunk nodes into data field of merged node

**rake** - primitive operation for tree contraction
- let \( T = (V,E) \) be binary tree rooted at \( r \)
  (assume each internal node has exactly 2 kids)
- let \( u \) be a leaf s.t. \( p(u) \neq r \)

rake operation applied to \( u \)
- make \( \text{sib}(u) \) a child of \( p(p(u)) \)
- remove \( u + p(u) \) from \( T \)

\[\text{rake}(u) \Rightarrow \]

\[
\begin{array}{c}
\text{u} \\
\text{z}
\end{array}
\begin{array}{c}
\text{x} \\
\text{y}
\end{array}
\begin{array}{c}
\text{w} \\
\text{z} \\
\text{y}
\end{array}
\]
Problem: can't simultaneously rake two leaves w/ same parent
⇒ apply rake only to non-consecutive leaves

Algorithm: Tree Contraction
input: rooted binary tree T, w/ p(v) + r(v) for each v \( \neq r 
output: 3-node binary tree (root + leftmost + rightmost leaves)
0. Label leaves of T consecutively left to right (except leftmost + rightmost) + place in array A[1...n] (n+2 leaves)
1. for i:=1 to \( \log(n+2) \) do
   0.1 Apply rake concurrently to each elt of A and
   that is a left child
   0.2 Apply rake concurrently to other elements of A and
   0.3 \( A[i] = A[i+1] \)
   endfor

Complexity (+ Correctness)

# iterations: iteration starts w/ m leaves, decreases to \( m/2 \)
\( \log(m+2) \) iters to reduce n leaves to 2 (leftmost + rightmost)

Step 1: Euler Tour technique, O(1) time, O(m) work
prefix sums to label leaves, O(log m) time, O(n) work

Step 2
- each iter w/ m leaves
  0.1 + 0.2: O(1) time, O(m) work
  0.3: comparison O(1) time, O(m) work [Remark: don't really need to compute]

Total Time: O(log m) - each iter O(1) time, O(log m) iters
Total Work: \( O\left( \sum_{i=1}^{\log m} \frac{n}{2^i} \right) = O(n) \)
EREW PRAM
After
Step 1

Iter #1
rake f

Iter #2
rake e, p

Iter #3
rake n

rake i, o, g

rake n

(2.2 nothing)
**Evaluating Arithmetic Expressions**

expression represented by rooted binary tree
- leaves are constants
- internal nodes are arithmetic operators + or \( \cdot \)

\[(a+b) \cdot c \Rightarrow \]

```
  o
 / \  \\
+   c  \\
\   \  \\
 a   b
```

goal: evaluate value of expression at root of \( T \), \( \text{val}(T) \)

**Simple Approach**
evaluate in parallel all subexpressions at nodes \( v \) both of whose kids are leaves + replace \( v \) by new leaf constant
- good if \( T \) reasonably balanced (best case \( \Theta(\log n) \) time)
- bad if \( T \) not balanced (worst case chain \( \Theta(n) \) time)

**Better Approach**
partially evaluate + then eliminate nodes w/ 1 leaf child
- associate a label \( (a_v, b_v) \) w/ each node representing linear expression \( a_v x + b_v \)
- \( a_v \) + \( b_v \) are constants
- \( x \) is an indeterminate that stands for possibly unknown value of subexpression at node \( v \)

idea: use basic tree contraction alg but augment rake operation to update values of \( (a_v, b_v) \) labels
invariant: for internal node $u$ with kids $v$ and $w$ and operator $o_u$

\[
\text{val}(u) = (a_v \cdot \text{val}(v) + b_v) \cdot o_u (a_w \cdot \text{val}(w) + b_w)
\]

- initially, label of each node is $(1,0)$

**Rake operation**

- $v$ leaf, $w = \text{sib}(v)$, $u = p(v)$, $c_v$ constant stored at $v$
  - remove $v$ and $u$
  - incorporate their contributions into $(a_w, b_w)$

**E.g.**

\[
\begin{align*}
\text{val}(x) &= (1 \cdot 3 + 0) + (1 \cdot \text{val}(w) + 0) \\
&= 3 + (1 \cdot (12 + 0) + 1 \cdot \text{val}(w) + 0) \\
&= 3 + 2 \cdot \text{val}(w)
\end{align*}
\]

**In general**

- $\text{val}(u) = (a_v \cdot c_v + b_v) \cdot o_u (a_w \cdot \text{val}(w) + b_w)$
- contrib. of $\text{val}(w)$ to $p(u)$ is
  \[
  E = a_w \cdot \text{val}(u) + b_w = a_u [(a_v \cdot c_v + b_v) \cdot o_u (a_w \cdot \text{val}(w) + b_w)] + b_u
  \]
  \[E \text{ is linear expression in } \text{val}(w) \text{ so can evaluate to find } (a_w', b_w')\]

**Case 1 $o_u =$**

\[
\begin{align*}
a_w' &= a_u (a_v \cdot c_v + b_v) a_w \\
b_w' &= a_u (a_v \cdot c_v + b_v) b_w + b_u
\end{align*}
\]

**Case 2 $o_u =$**

\[
\begin{align*}
a_w' &= a_u a_w \\
b_w' &= a_u (a_v \cdot c_v + b_v) + a_u b_w + b_u
\end{align*}
\]
Algorithm for expression evaluation

- expression represented as binary tree with nodes having labels \((a_v, b_v)\)
  - initialize labels to \((1,0)\)
- use basic tree contraction algorithm
  - augment rake operation to update labels \((a_v, b_v)\)

Complexity - same as tree contraction

\(O(k\log n)\) time, \(O(n)\) work, EREW PRAM

What if expression tree \(T\) is not binary?

\[\Rightarrow\text{ Convert } T \text{ into binary tree}\]

- replace each internal node \(v\) w/ degree \((v) > 2\)
  - by a binary tree w/ degree \((v) - 1\) internal nodes
  - each internal node has same operation as before

E.g.

\[
(1 + (2 \cdot 4) + 3 + 5) \cdot 9 \cdot (6 + 7 + 8) \Rightarrow [(1 + (2 \cdot 4)) + (3 + 5)] \cdot 9 \cdot [(6 + 7) + 8]
\]

(can view it as adding (unnecessary) parenthesis)