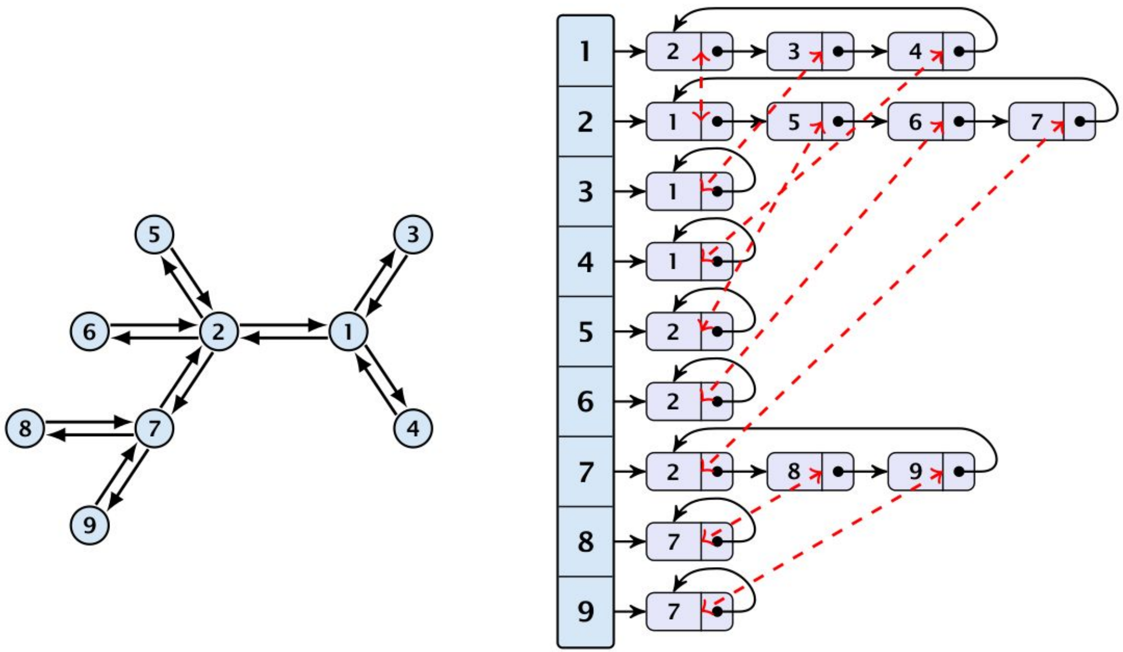
Tree Contraction

More of parallel pointer manipulation

Euler Tour Technique

Recap

Circular adjacency lists with twin pointers



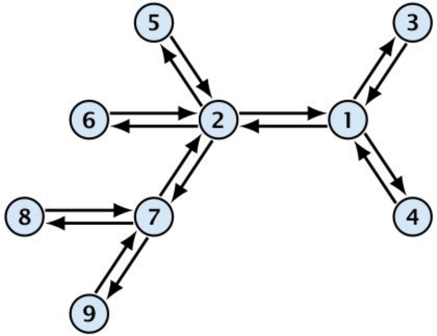
Find Euler tour

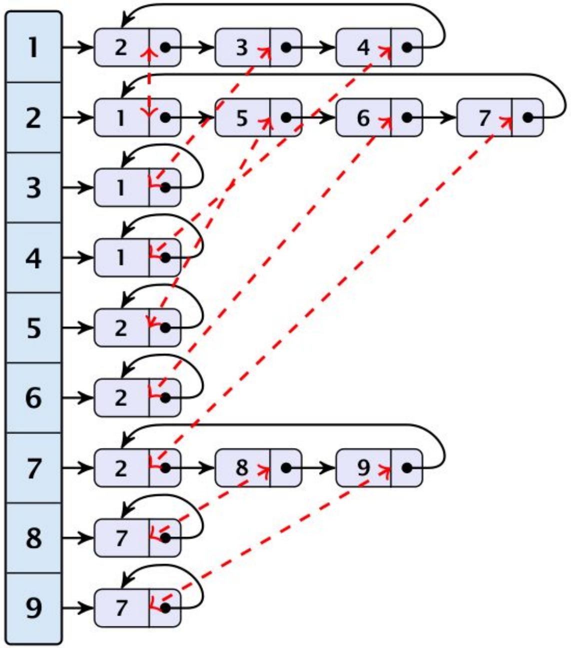
Every node v fixes an arbitrary ordering among its adjacent nodes:

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

We obtain an Euler tour by setting

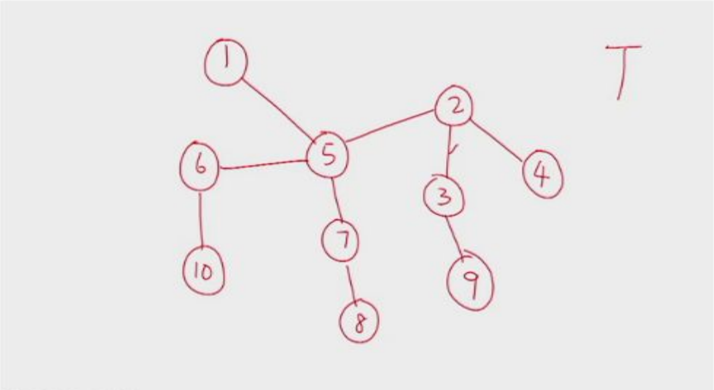
$$\mathrm{succ}((u_i, v)) = (v, u_{(i+1) \bmod d})$$

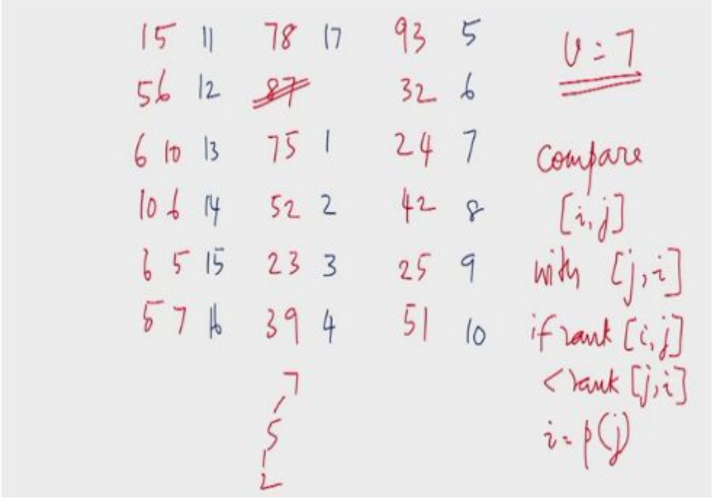




Each node's **parent** in "rooted" tree using Euler circuit

Take the Euler circuit
break the circuit by deleting one incoming edge of v
Now the ckt has become a list
Rank the list
Use the ranks ?





Application 1

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;

Each node's **level** in "rooted" tree using Euler circuit

The levels of the nodes of T
(T is rooted at vertex v)

level (v) = 0

level (w) = 1 where w is a child of v

level (w) = 2 — " — is a grand-child of v

15	11	78	17	93	5
56	12	~~87~~		32	6
6 10	13	75	1	24	7
10 6	14	52	2	42	8
6 5	15	23	3	25	9
5 7	16	39	4	51	10

7
|
5
|
2

v = 7

compare [i, j] with [j, i]
if rank [i, j] < rank [j, i]
i = p(j)

5 – 1
2 – 2
6 – 2

75	1	1
52	1	2
23	1	3
39	1	4
93	−1	3
32	−1	2
24	1	3
42	−1	2

25	−1	1
51	1	2
15	−1	1
56	1	2
6 10	1	3
10 6	−1	2
6 5	−1	1
5 7	−1	0
78	1	1

for parent-child edges : 1
child parent edges : −1

Application 2

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
- $\text{level}(v) = s[(\text{parent}(v), v)]$; $\text{level}(r) = 0$

Application 3

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
- $\text{post}(v) = s[(v, \text{parent}(v))]$; $\text{post}(r) = n$

Application 4

Number of descendants

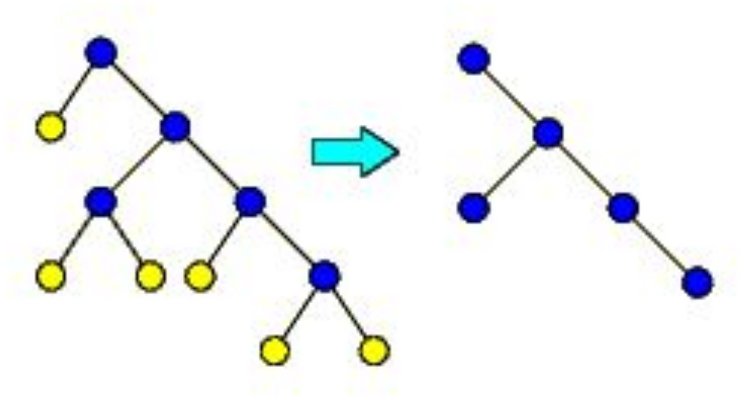
- split the Euler tour at node r
- this gives a list on the set of directed edges (Euler path)
- assign $x[e] = 0$ for every edge $(\text{parent}(v), v)$
- assign $x[e] = 1$ for every edge $(v, \text{parent}(v))$, $v \neq r$
- perform parallel prefix
- $\text{size}(v) = s[(v, \text{parent}(v))] - s[(\text{parent}(v),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

Tree Contraction: SHUNT

Comprises

- An operation of node removal called RAKE
- An operation of pointer jumping called COMPRESS
- Euler tour and prefix sum (to decide which nodes to RAKE and COMPRESS)

RAKE: removing leaves

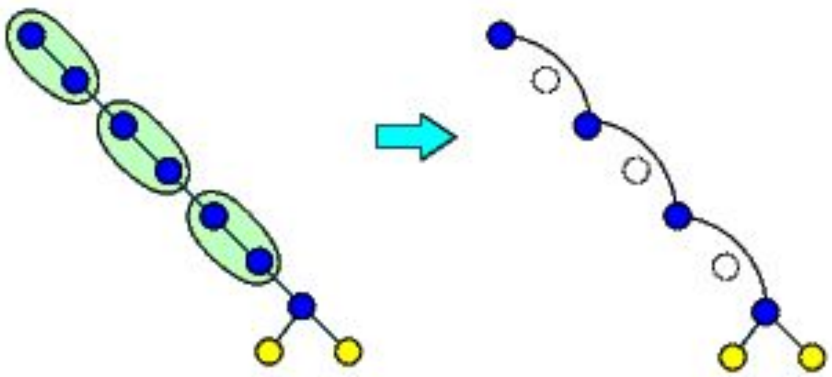


RAKE alone not sufficient for parallel tree contraction

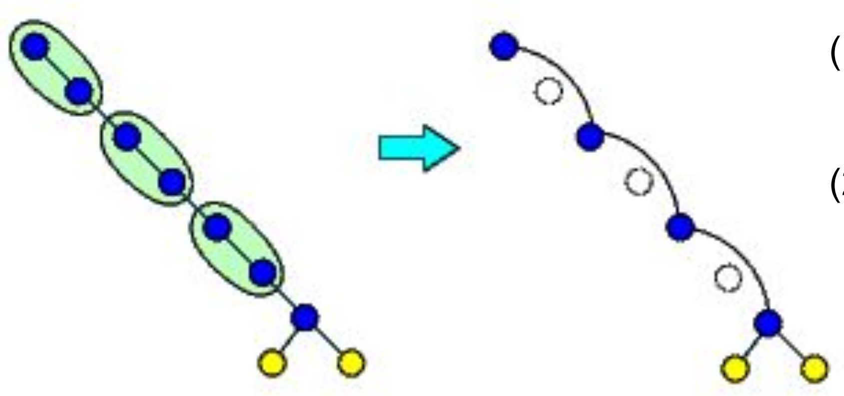
(1) If the tree is thin and tall, and therefore the height of the tree is linear, rather than logarithmic (the worst case is just a linked list), RAKE cannot be applied in parallel.

(2) RAKE itself tends to linearize the original tree.

COMPRESS: Pointer jumping

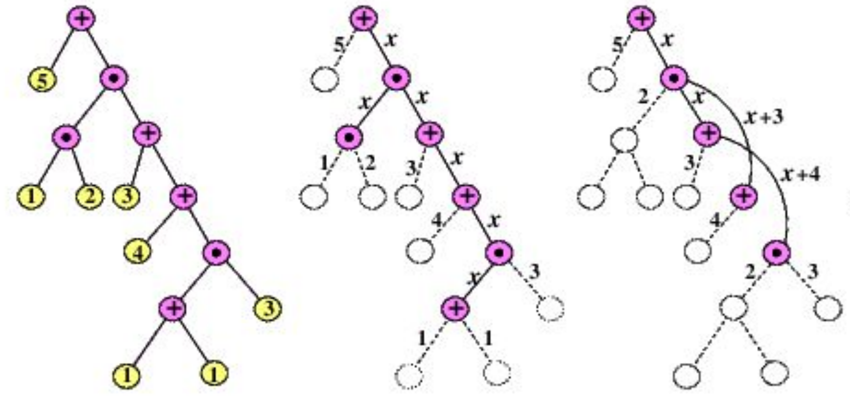


COMPRESS: Pointer jumping



(1) COMPRESS and RAKE can be applied in parallel to disjoint parts of the tree.

(2) COMPRESS produces leaves for RAKE and RAKE produces linear lists for COMPRESS.



Basic contraction algorithm

Input:	$P[1,\ldots,n]$; /* $P[x]$ is a pointer to the parent of x */
	$children[1,\ldots,n]$; /* $children[v] = \{v_1,\ldots,v_k\}$ – pointers to all children */
	$index[1,\ldots,n]$; /* $index[v_i] = i$ – each child v knows its index in $children[P[v]]$ */
Auxil:	$label[1,\ldots,n]$; /* $label[v] = \{f_1,\ldots,f_k\}$, where $f_i \in \{U, M\}$ */
	/* $f_i = M$ = marked iff a child supplied its value to its parent */
	$UnMarkChil(x)$ **returns** `int`; /* function returning the # of unmarked children */
Output:	the value accumulated in the root

```
/* initialize the data structures */
for all nodes v ∈ T do_in_parallel initialize(v);
while UnMarkChil(root) > 0 do
    { for all nodes v ∈ T do_in_parallel
        if P[v] ≠ nil then
          { case UnMarkChil[v] of
              0: { Rake(v); label[P[v]][index[v]] := M; P[v] := nil; }
              1: if UnMarkChil[P[v]] = 1 then { Compress(v); P[v] := P[P[v]]; }
          endcase }};
Rake(root);
```

Further reduce linearity of tree during contraction

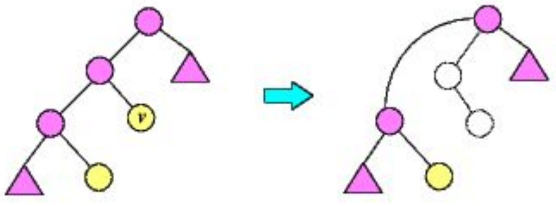
- Contraction should not produce linear chains
- A chain is produced when a tree contains a binary subtree, where each internal vertex has a child that is a leaf and one that is not.
- After a parallel RAKE on all leaves, such a subtree becomes a linear list
- If RAKE and COMPRESS is always applied simultaneously, i.e. every individual RAKE operation is followed immediately by COMPRESS of its sibling, chains cannot be formed
- This combined operation is called SHUNT

SHUNT: leaf RAKE with sibling COMPRESS

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 $p(p(u))$

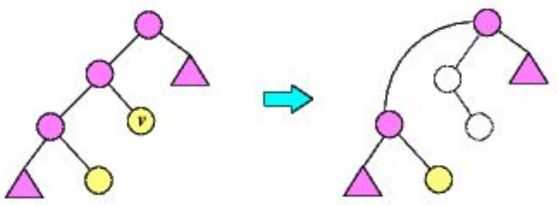


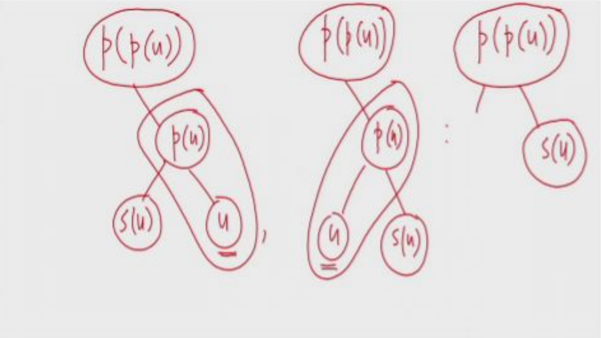
SHUNT: leaf RAKE with sibling COMPRESS

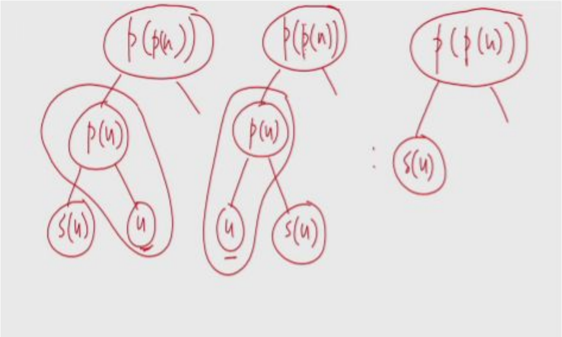
Given a binary tree T.

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

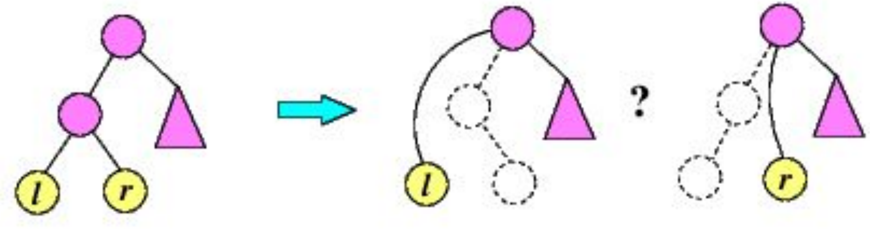
- remove u and $p(u)$
- attach sibling of u to $p(p(u))$



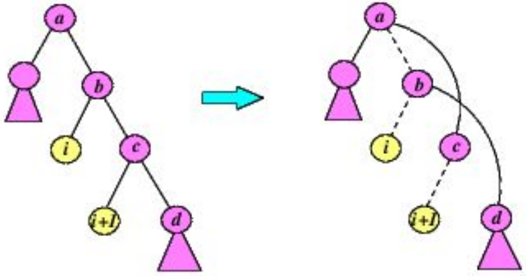




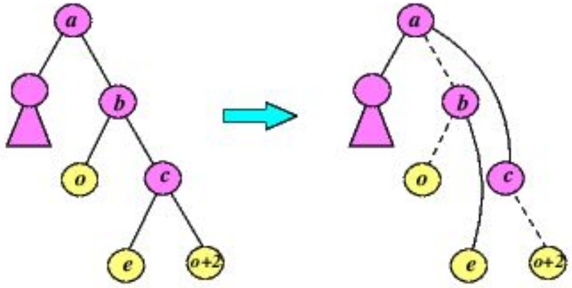
SHUNT conditions



Cannot be applied to two siblings simultaneously
(non-deterministic output)

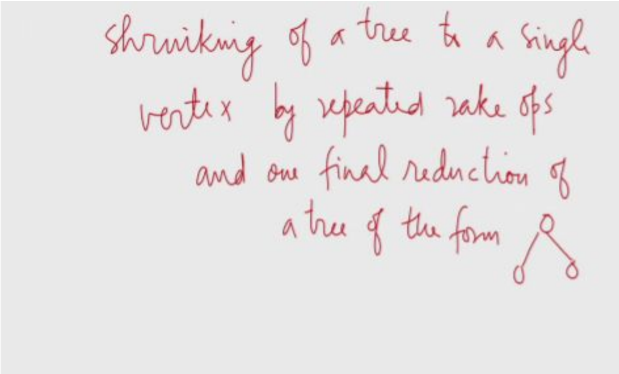


Cannot be applied to two consecutive leaves
(tree becomes disconnected)

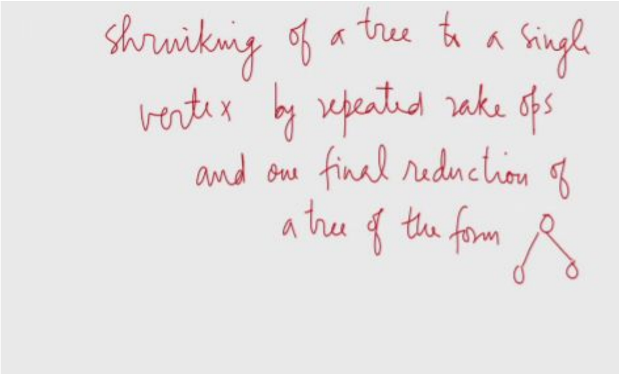


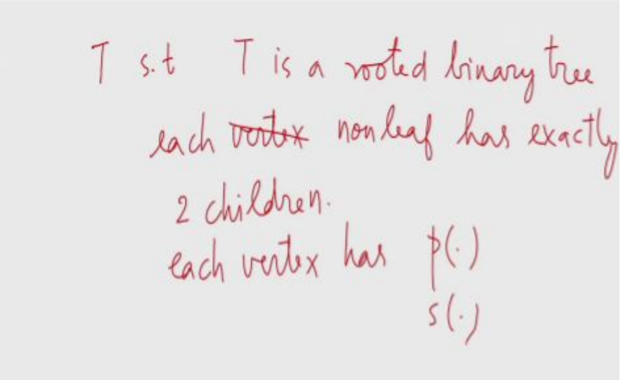
Cannot be applied to two consecutive odd leaves (tree
becomes disconnected, needs concurrent write)

The goal, the input tree and the algorithm



The goal, the input tree and the algorithm





The goal, the input tree and the algorithm

Shrinking of a tree to a single vertex by repeated rake ops and one final reduction of a tree of the form 

T s.t. T is a rooted binary tree

each ~~vertex~~ nonleaf has exactly 2 children.

each vertex has $p(\cdot)$, $s(\cdot)$

1. Get the adjacency list repn of T
2. Find an Euler ckt
3. Break the ckt open by deleting any incoming edge of r
4. Find a preorder traversal
5. Copy the traversal into an array
6. Mark all the leaf nodes
7. Compact the leaf nodes.

   Now the leaves are in a L to R order. Let A denote the array of leaves

A_{odd} : the seq. of the odd elements in A

A_{even} : ________ even

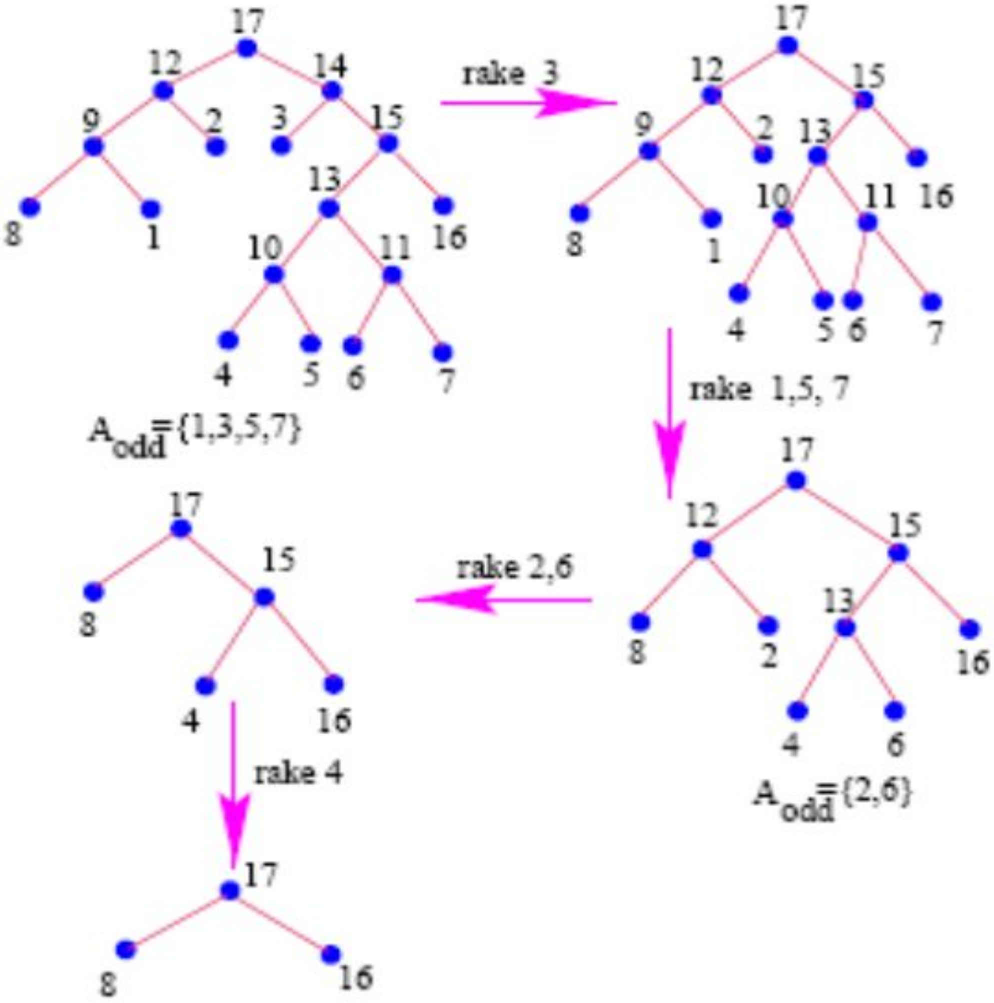
a b c d e f

| x | x | x

for $\lceil \log(n+1) \rceil$ iterations

- apply rake on all A_{odd} elements that are left children
- apply rake on the remaining elements of A_{odd}.
- $A := A_{even}$

Example run



17
12
14
9
2
3
15
13
8
1
16
10
11
4
5
6
7
A_odd={1,3,5,7}
rake 3
17
12
15
9
2
13
10
11
16
8
1
4
5
6
7
rake 1,5, 7
17
12
15
13
8
2
16
4
6
A_odd={2,6}
rake 2,6
17
15
8
4
16
rake 4
17
8
16

Why choose to RAKE in this order?

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

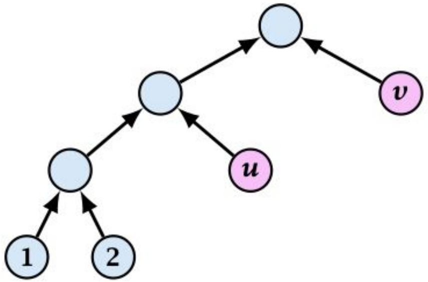
Possible Problems:

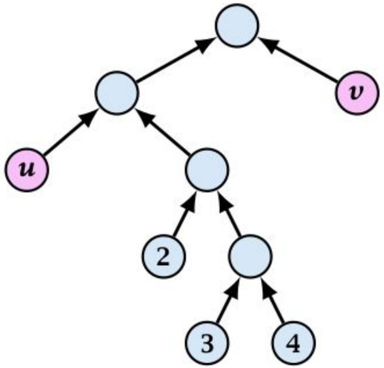
1. we could concurrently apply the rake-operation to two siblings
2. we could concurrently apply the rake-operation to two leaves u and v such that $p(u)$ and $p(v)$ are connected

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

Observations on the algorithm

- 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)





Runtime

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

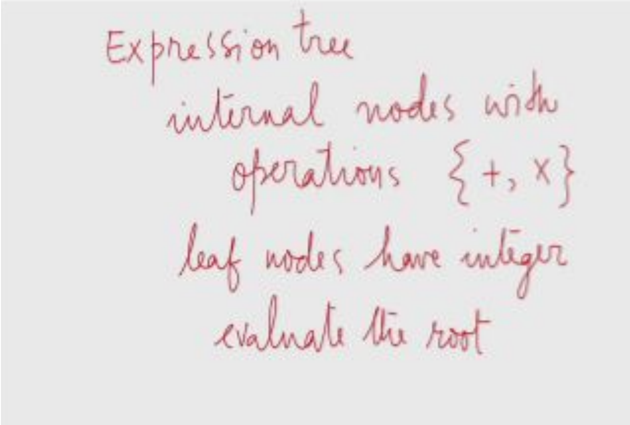
Runtime

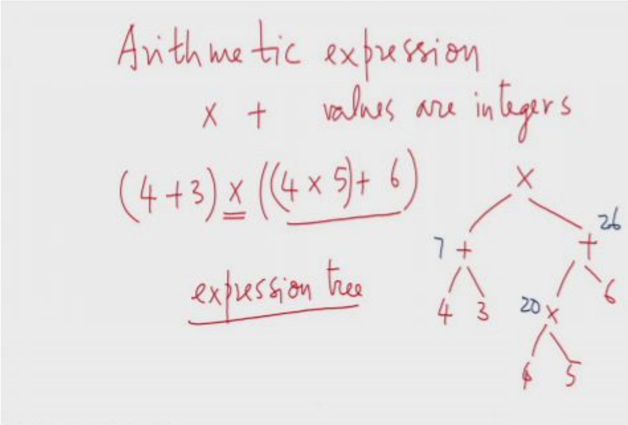
Algorithm:

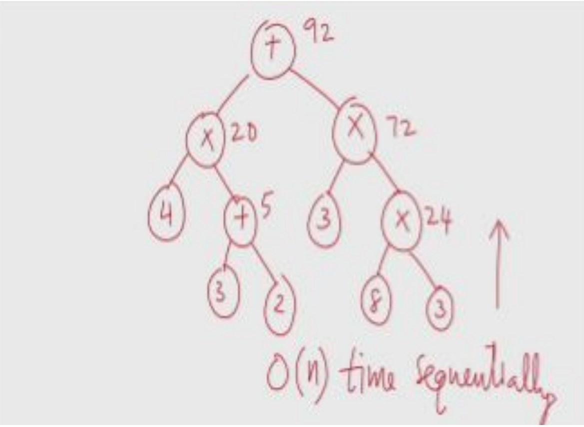
- label leaves consecutively from left to right (excluding left-most and right-most leaf), and store them in an array A
- for $\lceil \log(n+1) \rceil$ iterations
  - apply rake to all odd leaves that are left children
  - apply rake operation to remaining odd leaves (odd at start of round!!!)
  - A=even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

- 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 intial parallel prefix also requires time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$

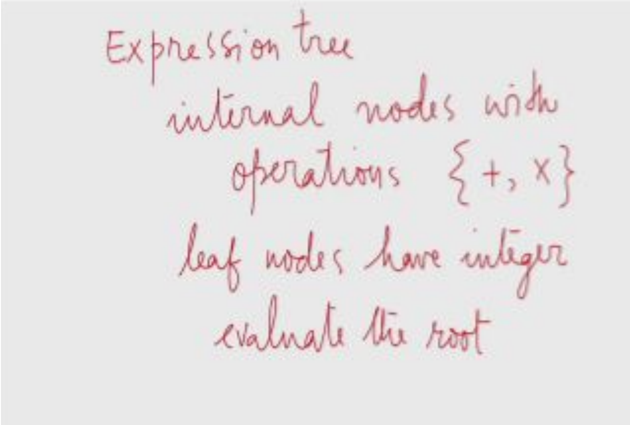
Application of tree contraction: expression evaluation

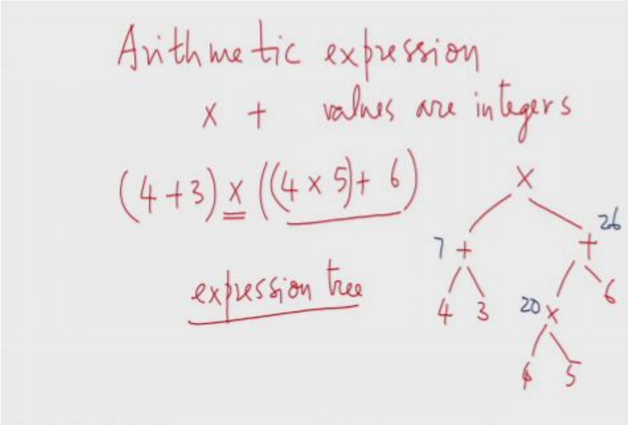


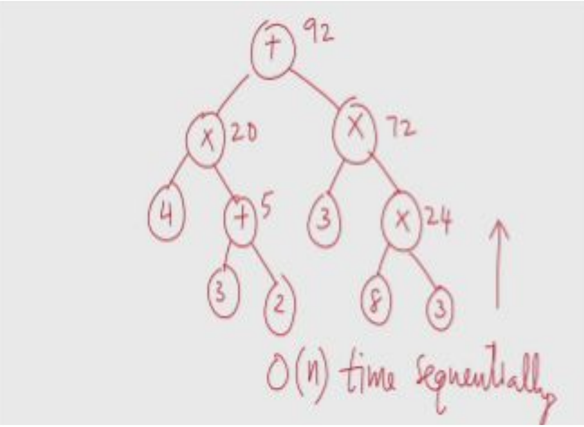


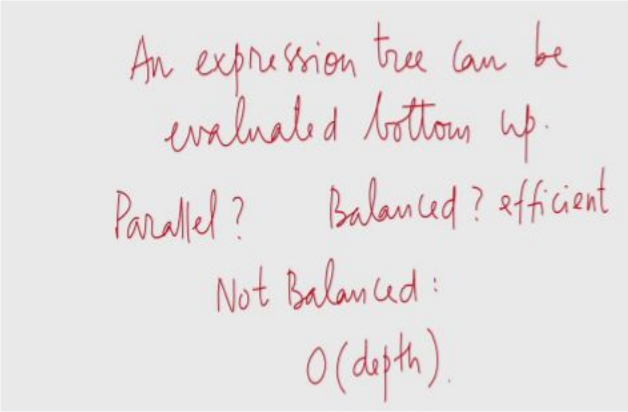


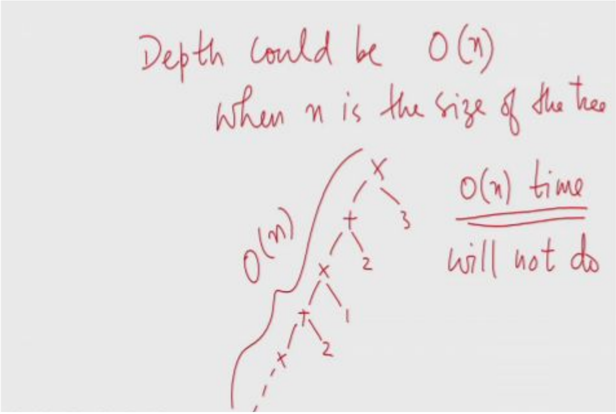
Application of tree contraction: expression evaluation











Maintaining expression values in nodes

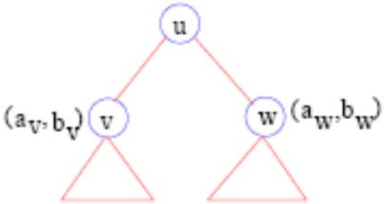
- We do not completely evaluate each internal node. We evaluate the internal nodes partially.
- For each internal node v, we associate a label (a_v, b_v). a_v and b_v are constants.
- The value of the expression at node is:

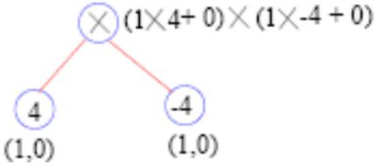
  $(a_v X + b_v)$, where X is an unknown value for the expression of the subtree rooted at v.

Invariant:

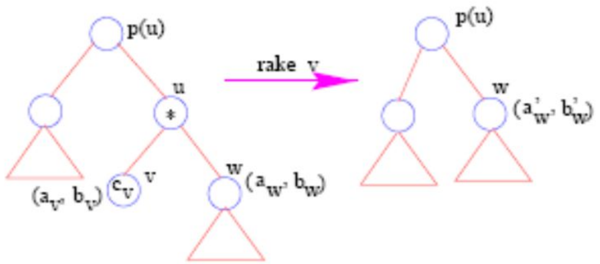
- Let u be an internal node which holds the operation $\oplus \in \{+, \times\}$.
- Let v and w are the children of u with labels (a_v, b_v) and (a_w, b_w).
- Then the value at u is:

  $val(u) = (a_v val(v) + b_v) \oplus (a_w val(w) + b_w)$





Propagating values while shunting



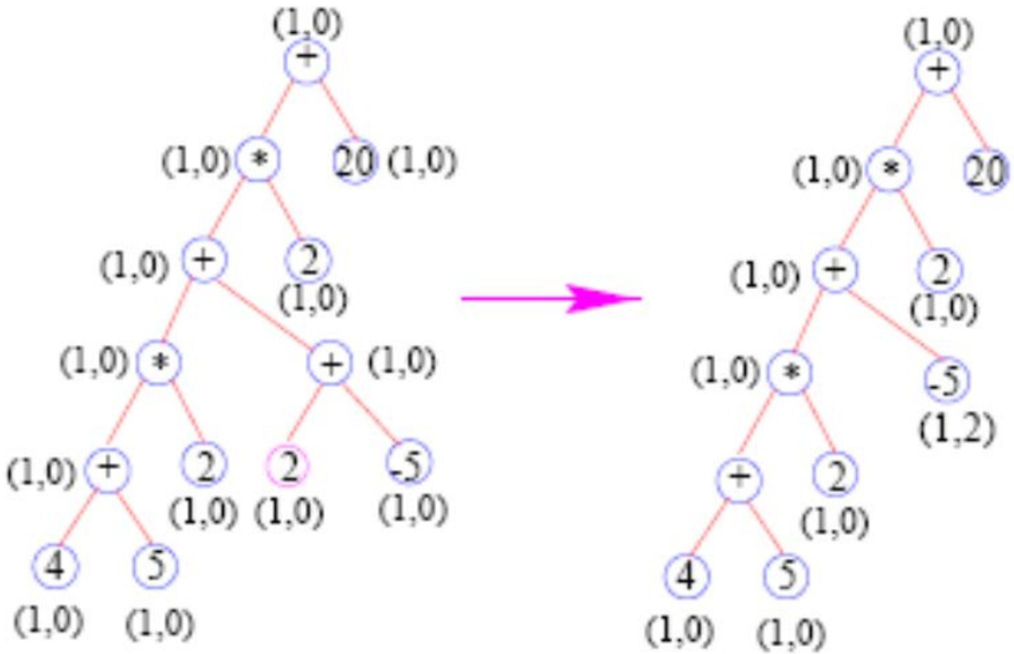
- The value at node u is:

  $val(u) = (a_v c_v + b_v) \times (a_w X + b_w)$
- X is the unknown value at node w.

- The contribution of $val(u)$ to the value of node $p(u)$ is:

  $a_u \times val(u) + b_u = a_u[(a_v c_v + b_v) \times (a_w X + b_w)] + b_u$
- We can adjust the labels of node w to (a'_w, b'_w)
- $a'_w = a_u(a_v c_v + b_v)\, a_w$
- $b'_w = a_u(a_v c_v + b_v)\, b_w + b_u$

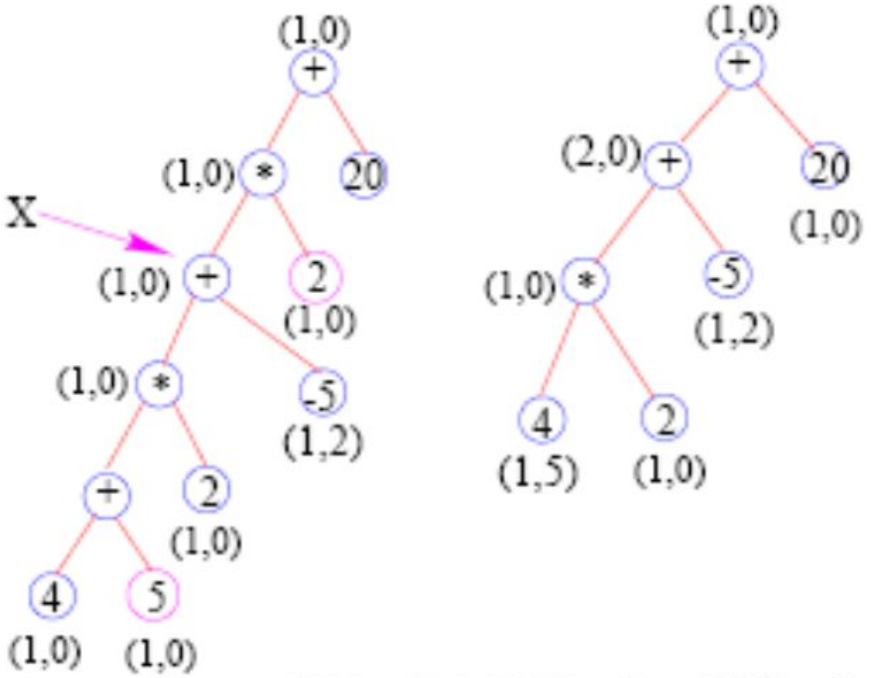
Example run through



$$(1*2 + 0) + (-5*1 + 0) = (1* -5 + 2)$$

(1,2)

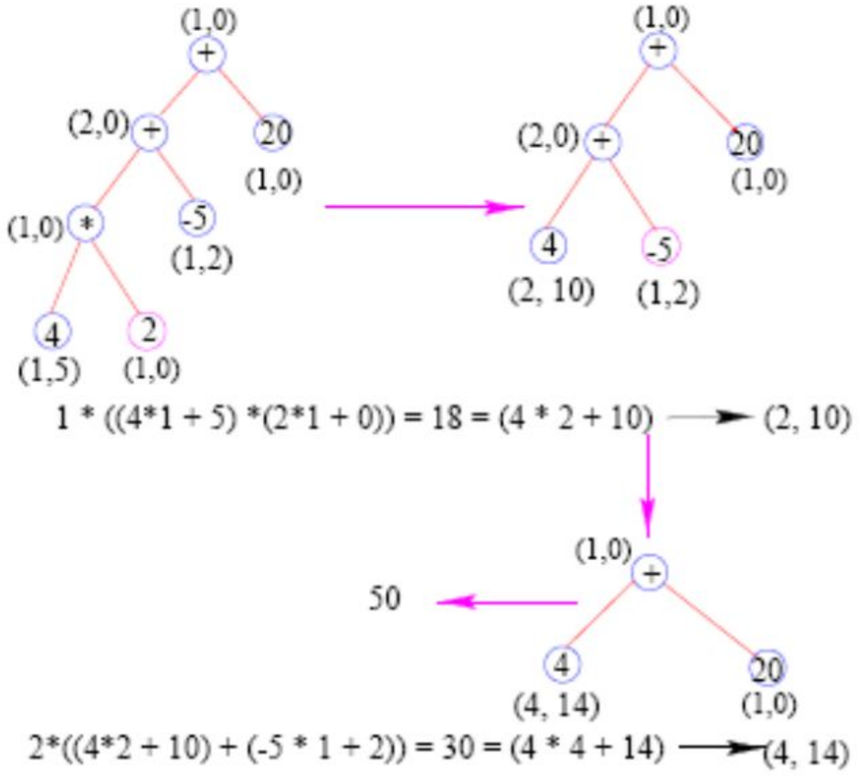
Example run through



(2*1 + 0) * (1*X + 0) = (2*X + 0) ⟶ (2,0)

(4*1+0) + (5*1 +0) = (4*1 + 5) ⟶ (1,5)

Example run through



Parallel algorithmic techniques

- **Divide and conquer (mergesort, parallel sum and other reductions, prefix-sum)**
- Parallel pointer manipulation
  - **Pointer jumping**
  - **Euler tour**
  - **Graph contraction (Tree contraction with rake, compress (pointer jumping), shunt)**
  - Ear decomposition
- Randomization
  - Sampling
  - Symmetry breaking
  - Load balancing