Complexity and Advanced Algorithms
Monsoon 2011

Parallel Algorithms
Lecture 4

Advanced Optimal Solutions

$$\begin{tikzcd}
1 \arrow[r] & 8 \arrow[r] & 5 \arrow[r] & 11 \arrow[r] & 2 \arrow[r] & 6 \arrow[r] & 10 \arrow[r] & 4 \arrow[r] & 3 \arrow[r] & 7 \arrow[r] & 12 \arrow[r] & 9
\end{tikzcd}$$

- General technique suggests that we solve a smaller problem and extend the solution to the larger problem.
- To apply our technique we should use the pointer jumping based solution on a sub-list of size n/log n.
- How to identify such a sublist?

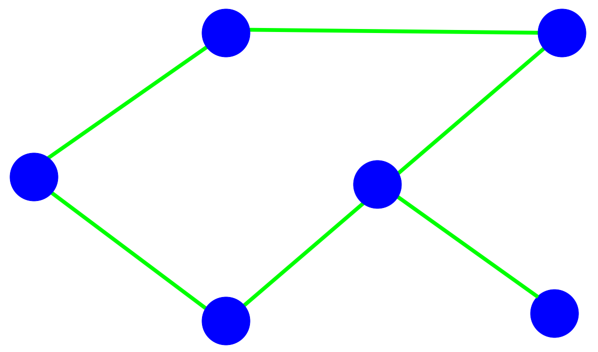
Advanced Solutions

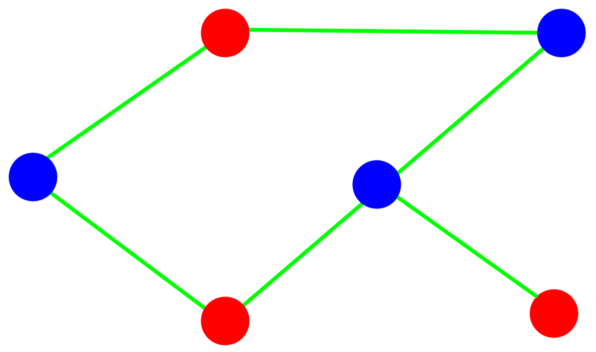
$$\begin{tikzcd}
1 \arrow[r] & 8 \arrow[r] & 5 \arrow[r] & 11 \arrow[r] & 2 \arrow[r] & 6 \arrow[r] & 10 \arrow[r] & 4 \arrow[r] & 3 \arrow[r] & 7 \arrow[r] & 12 \arrow[r] & 9
\end{tikzcd}$$

$$\begin{tikzcd}
1 \arrow[r] & 5 \arrow[r] & 11 \arrow[r] & 6 \arrow[r] & 4 \arrow[r] & 3 \arrow[r] & 12 \arrow[r] & 9
\end{tikzcd}$$

- Cannot pick equidistant as earlier.
- However, can pick independent nodes.
  - Removing independent nodes is easy!
  - Formally, an independent set of nodes.
  - Can extend the solution easily in such a case.

Advanced Solutions

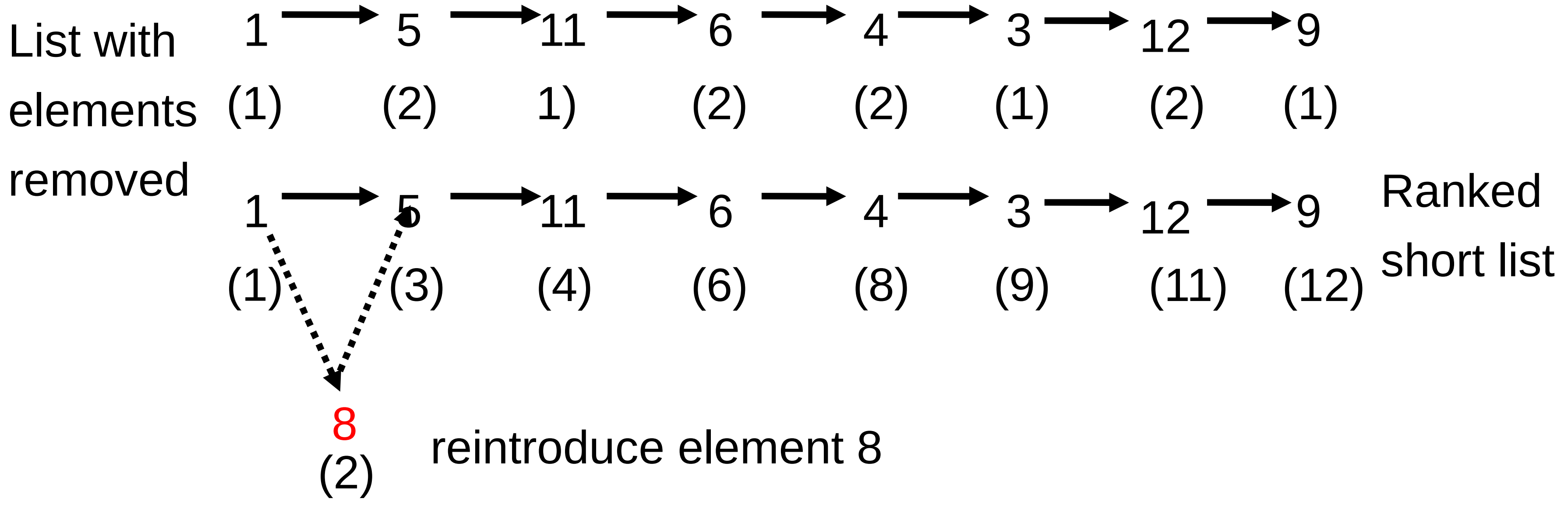




- Formally, in a graph G = (V, E), a subset of nodes U ⊆V is called an independent set if for ever pair of vertices u,v in U, (u,v) ∉ E.
- Linked lists (viewed as a graph) have the property that they have large independent sets.

Advanced Solutions

$$\begin{tikzcd}
1 \arrow[r] & 8 \arrow[r] & 5 \arrow[r] & 11 \arrow[r] & 2 \arrow[r] & 6 \arrow[r] & 10 \arrow[r] & 4 \arrow[r] & 3 \arrow[r] & 7 \arrow[r] & 12 \arrow[r] & 9
\end{tikzcd}$$



- Transfer current rank along with successor during removal.

Advanced Optimal Solutions

$$\begin{tikzcd}1 \arrow[r] & 8 \arrow[r] & 5 \arrow[r] & 11 \arrow[r] & 2 \arrow[r] & 6 \arrow[r] & 10 \arrow[r] & 4 \arrow[r] & 3 \arrow[r] & 7 \arrow[r] & 12 \arrow[r] & 9\end{tikzcd}$$

$$\begin{tikzcd}1 \arrow[r] & 5 \arrow[r] & 11 \arrow[r] & 6 \arrow[r] & 4 \arrow[r] & 3 \arrow[r] & 12 \arrow[r] & 9\end{tikzcd}$$

- Algorithm outline:
  - Remove an independent set of nodes in the linked list.
  - Rank the remaining list, and then
  - Rank the removed elements.

- Several algorithms use this technique with variations: Anderson-Miller, Hellman-JaJa, Reid-Miller,...

Advanced Solutions

- Issue 1: How to find a large independent set of nodes in parallel?
- Issue 2: How many iterations needed to reduce the size of the list?
- Issue 3: How to rank the removed elements?

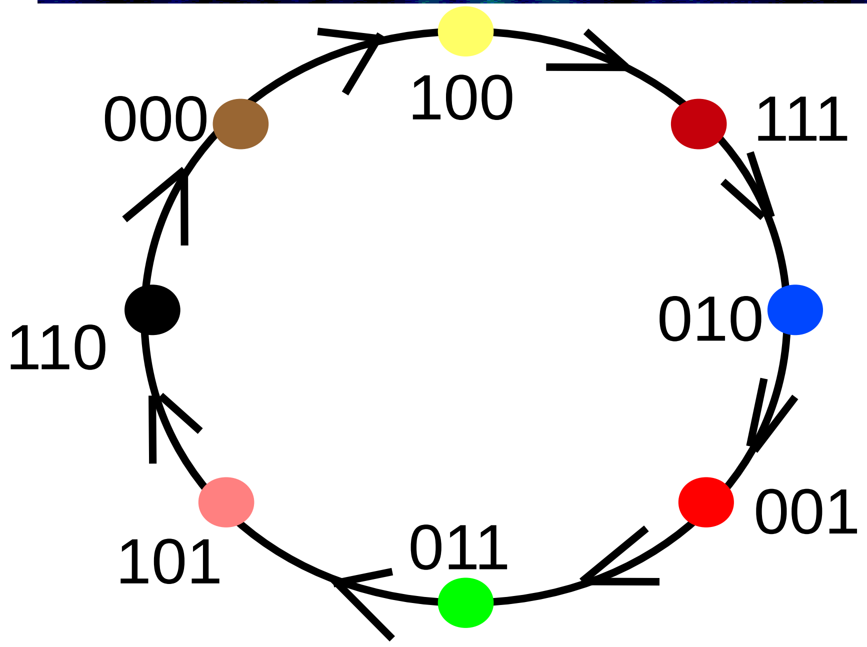
Advanced Solutions

- Issue 1:
  - Use techniques from parallel symmetry breaking.
  - Can obtain an independent set of size $\geq$ n/3.
- Issue 2:
  - The naïve algorithm is slightly non-optimal (by a factor of O(log n)
  - Hence, reduce the size of the list from n to n/log n.
- Issue 3:
  - Bookkeep enough details during removal
  - Reintroduce in the reverse order.

Symmetry Breaking

- A way to induce differences between like (symmetric) participants.
- Useful in applications such as graph coloring
  - Generally, difficult using deterministic techniques.
    - Need randomization
- Special cases where fast, deterministic symmetry breaking can be achieved.
    - Linked lists and directed cycles are an example.

Coloring by Symmetry Breaking

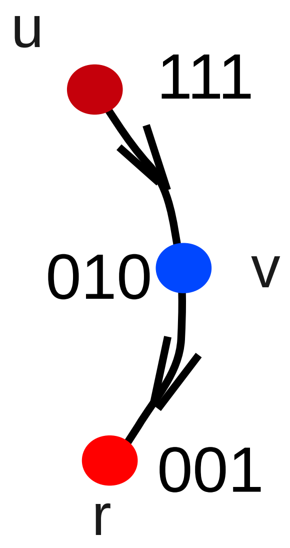


u	v	New Color(u)
110	000	10 (k = 1)
000	100	100 (k = 2)
100	111	00 (k = 0)
010	001	00 (k = 0)
001	011	10 (k = 1)
011	101	11 (k = 1)
111	010	11 (k = 0)
101	110	01 (k = 0)

- Consider a directed cycle of n nodes numbered 1 to n.
- Treat the number of the node as its initial color.
- Can reduce colors to log n in one step.
  - Compare color with parent, Newcolor(u) = 2k + $\text{color}(u)_k$
  - k is the index of the first bit position that u and v differ from LSB

Coloring by Symmetry Breaking

- Claim: New colors are valid.
- Proof: Suppose that for u and v such that u → v, NewColor(u) = NewColor(v).
- Let NewColor(u) = 2k + color(u)$_k$, and NewColor(v) = 2r + color(v)$_r$.
- Let k = r. However, color(u)$_k$ = color(v)$_k$. Why?
- Let k = r. Then, color(u)$_k$ – color(v)$_r$ = 2(r – k).
  - The LHS has an absolute value of 1 and the RHS has an absolute value of 2.



Coloring by Symmetry Breaking

- In one iteration, can reduce the number of colors from n to 2log n.
  - Initial colors are log n bits
  - New colors are only $1+\lceil \log\log n \rceil$ bits.
- Can we repeat again?
  - Yes.
  - Reduces number of bits from t to $1+\lceil \log t \rceil$.
  - But, at some point $t < 1 + \lceil \log t \rceil$. No advantage any further.
  - Happens at t = 3.
- So, repeat till only 8 colors are used.

Coloring by Symmetry Breaking

- At that point, can still reduce the number of colors as follows:
- For i = 8 downto 3 in sequence
  - If node u is colored i, then u chooses a color among {1,2,3} that is not same as the colors of its neighbors.
- Possible to do so. Why?

Coloring by Symmetry Breaking

- At that point, can still reduce the number of colors as follows:
- For i = 8 downto 3 in sequence
  - If node u is colored i, then u chooses a color among {1,2,3} that is not same as the colors of its neighbors.
- Possible to do so. Why?
  - Each node has only two neighbors.
  - So, only two colors amongst {1,2,3} can be used up already.

Coloring by Symmetry Breaking

- Total time analyzed as follows:
  - Each iteration of symmetry breaking reduces number of bits from to $1+\lceil \log t \rceil$.
  - The recurrence relation is $T(n) = T(\log n) + 1$
  - Solution: $T(n) = O(\log^* n)$.
  - In the next phase, only 5 iterations.
  - So, overall time = $O(\log^* n)$
- Work however is $O(n\log^* n)$.
- $\text{Log}^* n = i$ such that

$$\underbrace{\log(\log(\ldots\ldots(\log n)))}_{i} = 1;$$

- The algorithm extends to lists and trees also.

Coloring to Independent Sets

- For bounded degree graphs colored with O(1) colors, a coloring is equivalent to finding a large independent set.

- Iterate on each color and count the number of nodes with a given color.
- Pick the subset of like colored nodes of the largest size.
  - Clearly, an independent set.
  - Has a size of at least a fraction of n.

Back to List Ranking

- Our algorithm outline:

```
Algorithm Rank(L)
$L_1$ = L;
For r iterations do
    Color the list L with 3 colors
    Pick an independent set $U_i$ of nodes of size $\geq n/3$
    $L_i$ = Remove nodes in $U_i$ from $L_{i-1}$;
End-for

Rank the list $L_r$ using pointer jumping.

For i = r down to 1 do
    Reinsert the nodes in $U_i$ into $L_i$
End-for
End.
```

- Question: How many iterations required?

Back to List Ranking

- Each iteration of coloring the list can give an MIS of size at least n/3.
- We require that only n/log n nodes remain at the end.
- Hence, O(log log n) iterations are required.
  - $(n/3)^r$ = n/log n at r = O(log log n).

Back to List Ranking

- Time taken:
  - To shrink the list: Each iteration is O(log* n). At O(log log n) iterations, this takes O(log *n . log log n) time.
  - To rank the remaining list using pointer jumping: O(log n) time
  - To reintroduce the removed elements: Over r = O(log log n) iterations, O(log log n) time.
  - Total = O(log n).
- Work: O(nlog log n)
  - Dominated by the work done in reintroducing the removed elements

Back to List Ranking

- A different way:
  - Reintroducing can be slowed down to make it optimal.
  - Use only n/log n processors, with each iteration taking O(log n) time.
  - Total time = O(log n log log n).
  - Total work = O(nlog* n).
    - Dominated by the time taken to find an independent set.

Back to List Ranking

- Yet another way:
  - Use an optimal approach to finding an independent set.
  - Takes O(log n) time and O(n) work.
- Overall time and work:
  - Time = O(log n log log n)
  - Work = O(n)
  - Optimal!

Back to List Ranking

- In general, if one can spend O(t) time and O(n) work in each iteration of removing nodes, then
  - Time = O(t log log n + log n)
  - Work = O(n).
- Have to lower t to get O(log n) optimal list ranking.
- There are such algorithms.
  - Anderson-Miller is one such example.
- Further reading
  - Anderson Miller described in JaJa's book
  - Hellman-JaJa is another popular approach
    - Used by most practical papers in recent times.

After a long break, welcome again.

Have to make up two lectures and one exam.

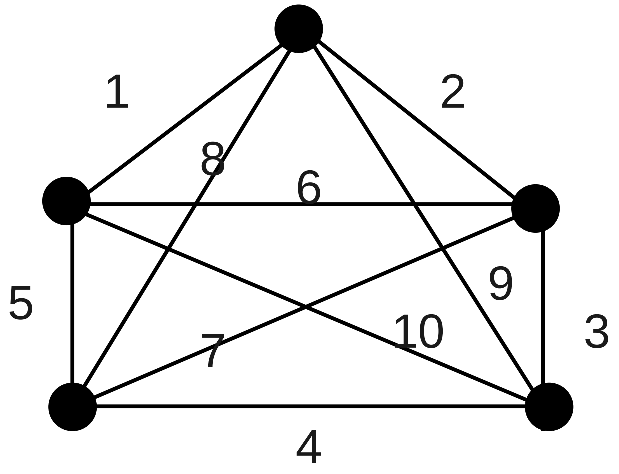
Lectures: What if we continue our classes till 4 PM for the next 6 lectures? With a small break after 1 hour.

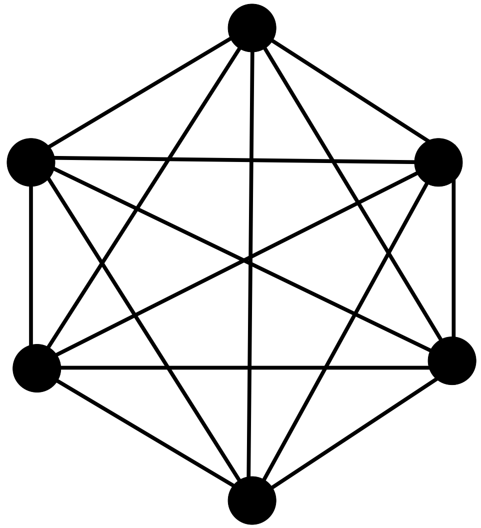
Exam: Will hold it some day after possibly 2 weeks from now. May be one hour exam, with only 15% weightage.

Tree Processing

- Now that we know how to process linked lists, let us consider tree algorithms.
- Problems we will consider:
  - Traversal
  - Expression evaluation
  - Least common ancestor, range minima

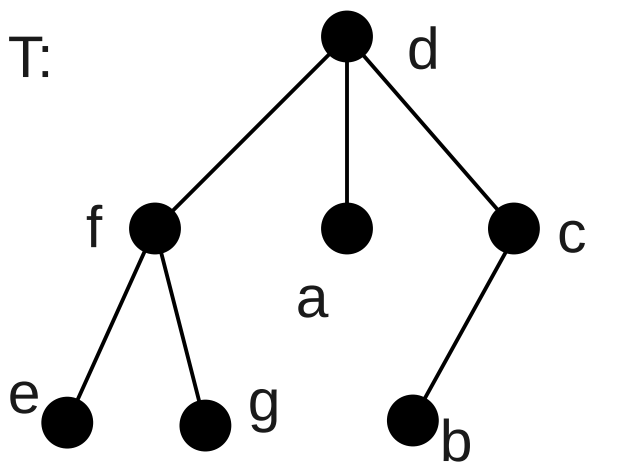
Traversal via Euler Tour

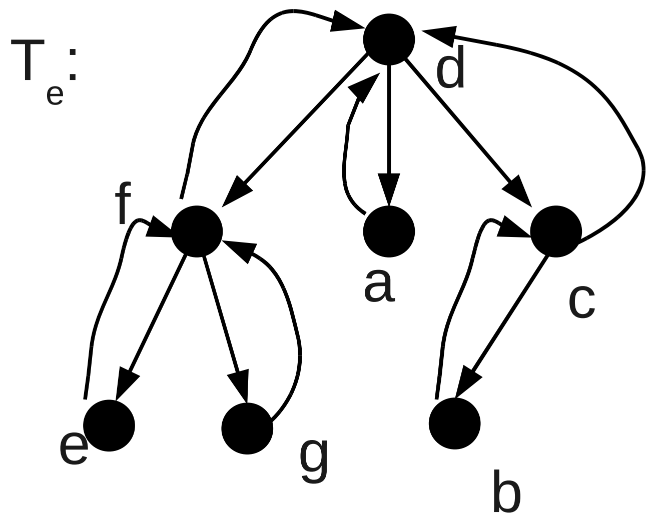




- Given a tree T = (V, E), we use an Euler tour as a primitive for tree traversal algorithms.
- Euler tour: Given a graph, an Euler tour is a cycle that includes every edge exactly once.
  - A directed graph G has an Euler tour if and only if for every vertex, its in-degree equals its out-degree.

Euler Tour of a Tree





- For a tree T = (V,E) to define an Euler tour, we make it a directed tree:
  - Define $T_e = (V_e, E_e)$ with $V_e = V$.
  - For each edge uv in E, add two directed edges (u,v) and (v,u) to E_e.
  - T_e has an Euler tour.

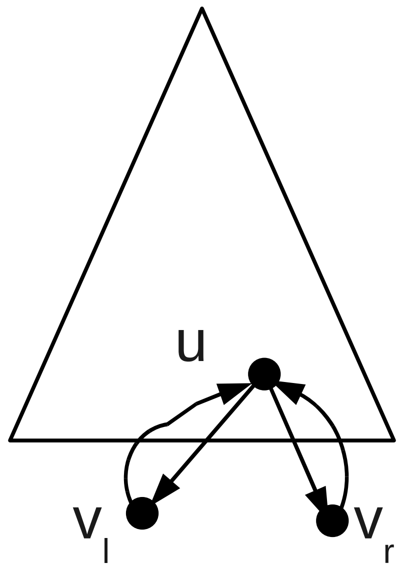
Defining an Euler Tour on a Tree

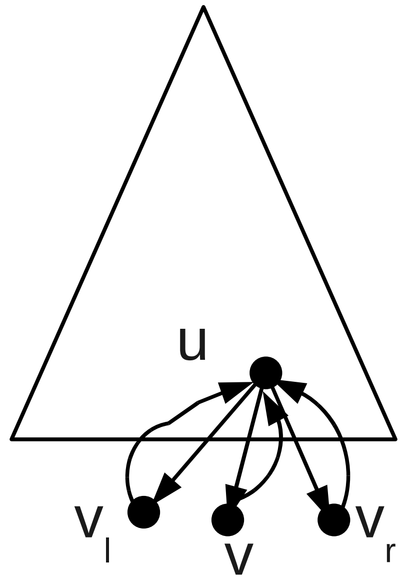
- Just have to define a successor. Here, successor for an edge.
- For a node u in T_e order its neighbors v_1, v_2, ..., v_d.
  - Can be done independently at each node.
  - For e = (v_i, u), set s(e) = e' where e' = (u, v_{i+1}).
    - Compute indices modulo the degree of u.

Euler Tour on a Tree

- Claim: The above definition of s: E → E is a tour.
- Proof: By induction. Let n = 1. Obviously true.
- For n = 2, at most one edge present. The tour is well defined according to s().
- Let the tour be well defined for n = k.
- Step: n = k+1.
  - Every tree has at least one leaf, say v.
  - T' = T \ {v} has an Euler tour defined by s':E(T') → E(T') as |V(T')| = k.
  - We now extend this definition of s' to define a function s: E(T) →E(T).

Euler Tour on a Tree

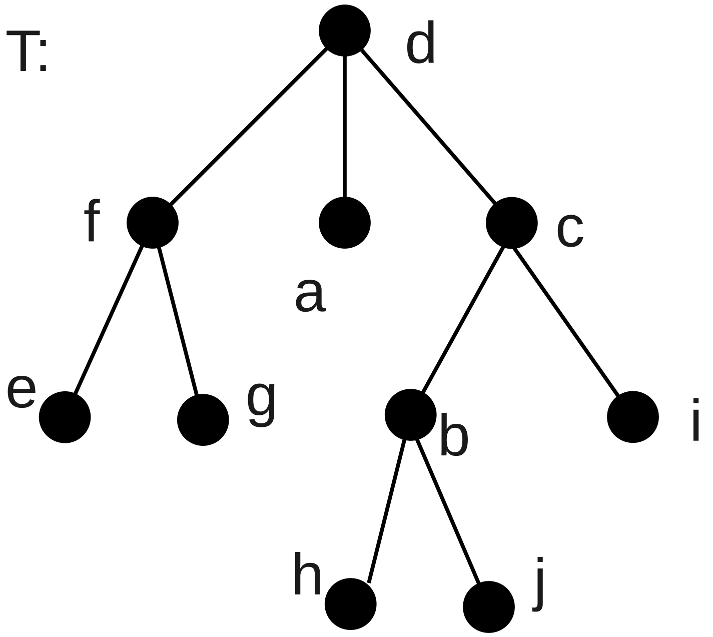




- Let u be the neighbor of v in T.
  - Let $N(u) = \{v_0, \ldots, v_{i-1}, v_i{=}v, v_{i+1}, \ldots v_d\}$.
- Set $s(u, v) := (v,u)$. Set $s(v_{i-1},u) := (u,v)$.
- At all other edges $e \in T$, $s(e) := s'(e)$.

Example

T:



$$\begin{tikzcd}
a \arrow[r] & d & & \\
b \arrow[r] & j \arrow[r] & h \arrow[r] & c \\
c \arrow[r] & i \arrow[r] & d \arrow[r] & b \\
d \arrow[r] & c \arrow[r] & a \arrow[r] & f \\
e \arrow[r] & f & & \\
f \arrow[r] & g \arrow[r] & d & \\
g \arrow[r] & f & & \\
h \arrow[r] & b & & \\
i \arrow[r] & c & & \\
j \arrow[r] & b & &
\end{tikzcd}$$

Example

$$\begin{tikzcd}
a \arrow[r] & d \\
b \arrow[r] & j \arrow[r] & h \arrow[r] & c \\
c \arrow[r] & i \arrow[r] & d \arrow[r] & b \\
d \arrow[r] & c \arrow[r] & a \arrow[r] & f \\
e \arrow[r] & f \\
f \arrow[r] & g \arrow[r] & d \\
g \arrow[r] & f \\
h \arrow[r] & b \\
i \arrow[r] & c \\
j \arrow[r] & b
\end{tikzcd}$$

s(b,c) = (c,i)

s(c,i) = (i,c)

s(i,c) = (c,d)

s(c,d) = (d,a)

s(d,a) = (a,d)

s(a,d) = (d,f)

s(d,f) = (f,g)

s(f,g) = (g,f)

s(g,f) = (f,d)

s(f,d) = (d,c)

s(d,c) = (c,b)

s(c,b) = (b,j)

s(b,j) = (j,b)

s(j,b) = (b,h)

s(b,h) = (h,b)

s(h,b) = (b,c)

s(b,c) = (c,i)

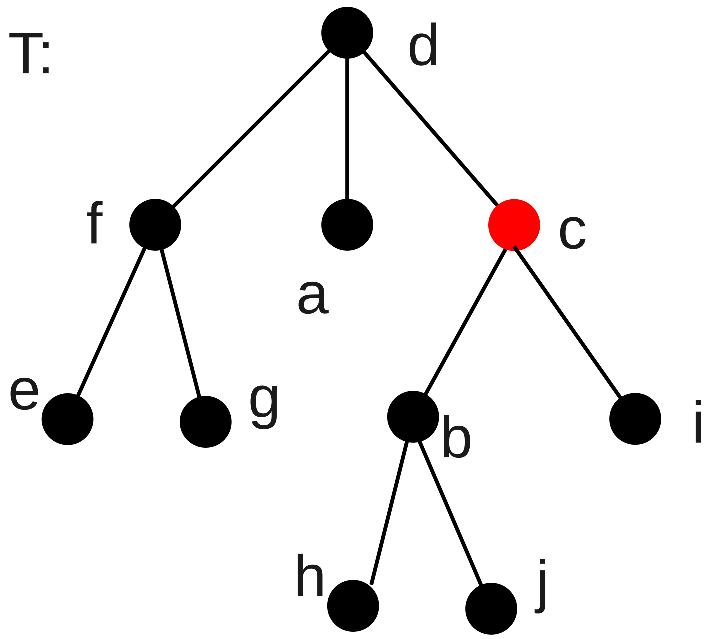
Euler Tour:

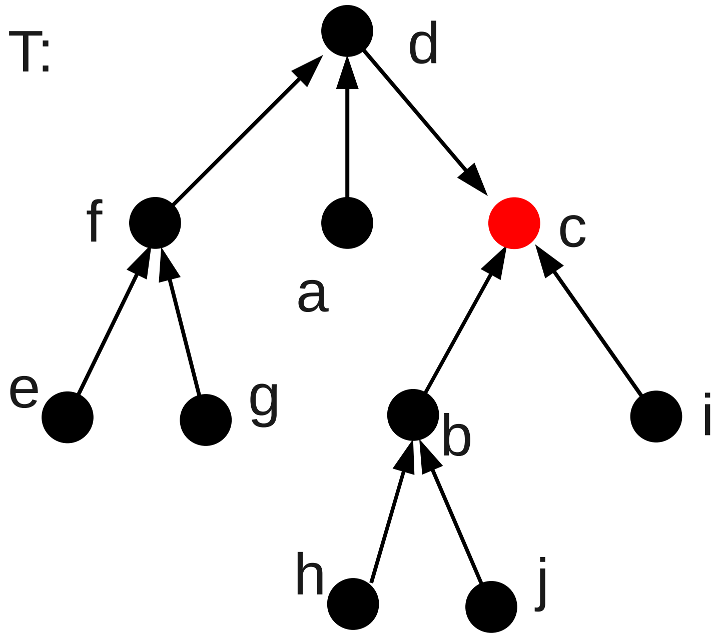
(b,c) → (c,i) → (i,c) → (c,d) → (d,a) → (a,d) → (d,f) → (f,g) → (g,f) → (f,d) → (d,c) → (c,b) → (b,j) → (j,b) → (b,h) → (h,b) → (b,c)

Applications of Euler Tour to Traversal

- We now show why the Euler tour is an important construct for trees.
- Operations on a tree such as rooting, perorder and postorder traversal can be converted to routines on an Euler tour.

Rooting a Tree

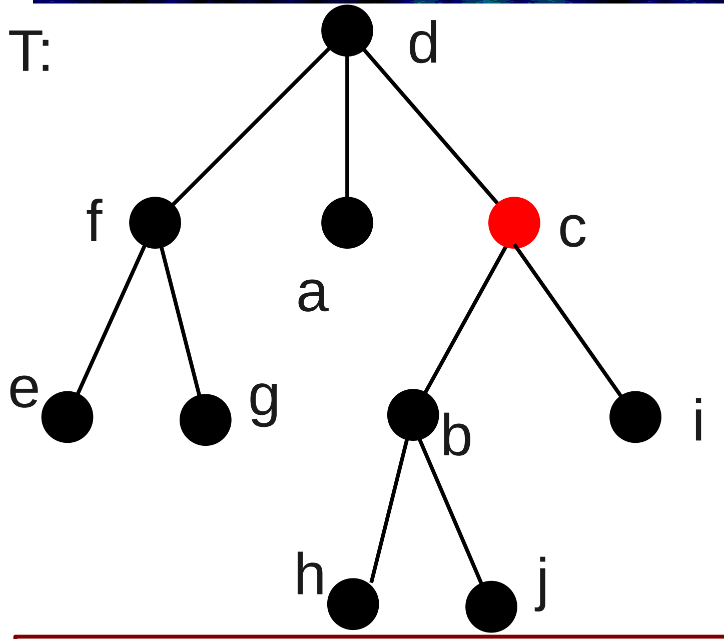


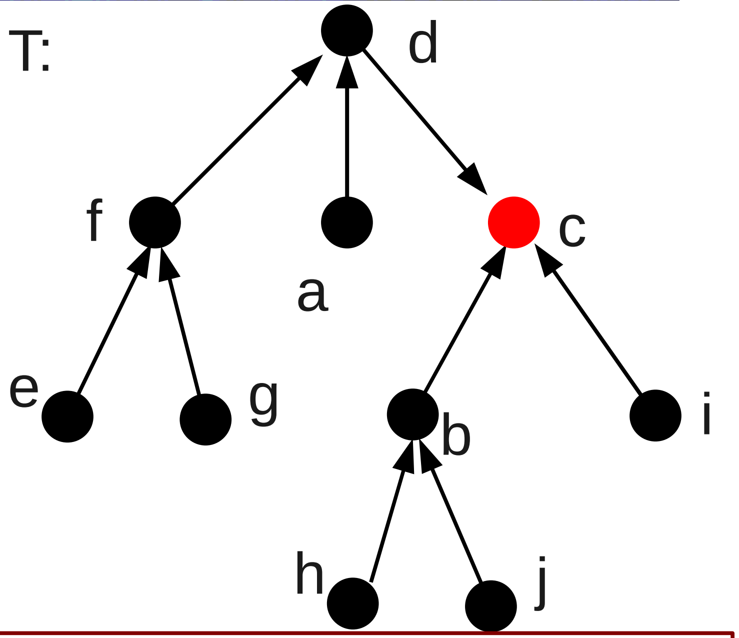


Euler Tour:
(b,c) → (c,i) → (i,c) → (c,d) → (d,a) → (a,d) → (d,f) → (f,g) → (g,f) → (f,d) → (d,c) → (c,b) → (b,j) → (j,b) → (b,h) → (h,b) → (b,c)

- Designate a node in a tree as the root.
- All edges are directed towards the root.

Rooting a Tree



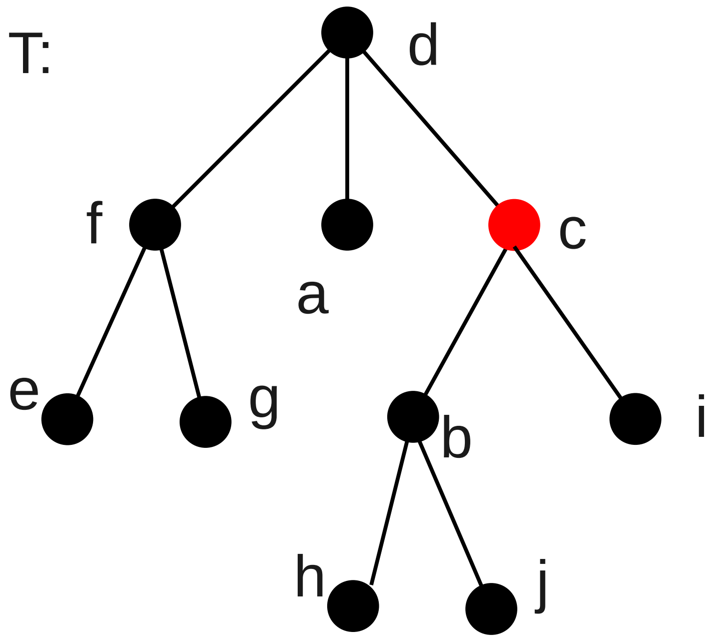


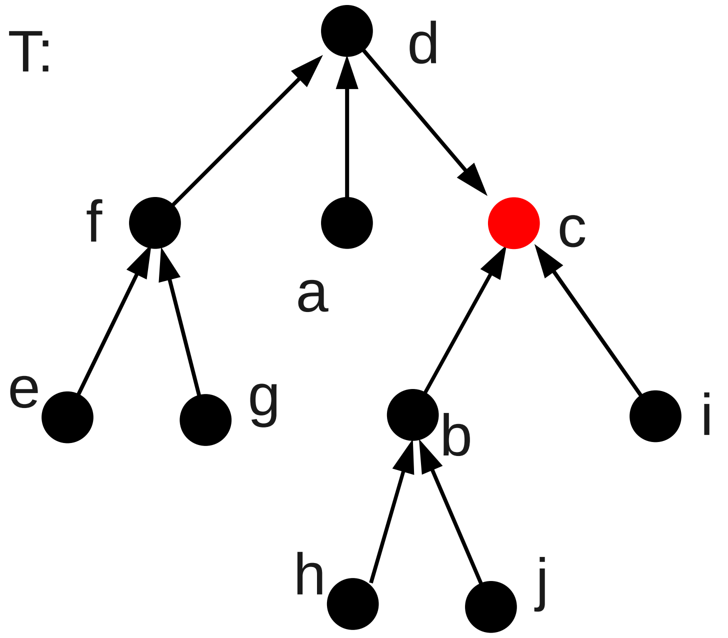
Euler Tour:

(b,c) → (c,i) → (i,c) → (c,d) → (d,a) → (a,d) → (d,f) → (f,g) → (g,f) → (f,d) → (d,c) → (c,b) → (b,j) → (j,b) → (b,h) → (h,b) → (b,c)

- Let (v_1, v_2, ..., v_d) be the neighbors of the root node r. In this case, say (i, d, b)
- Set s(e) = NULL for e = (v_d, r). In this case, s((b,c)).

Rooting a Tree



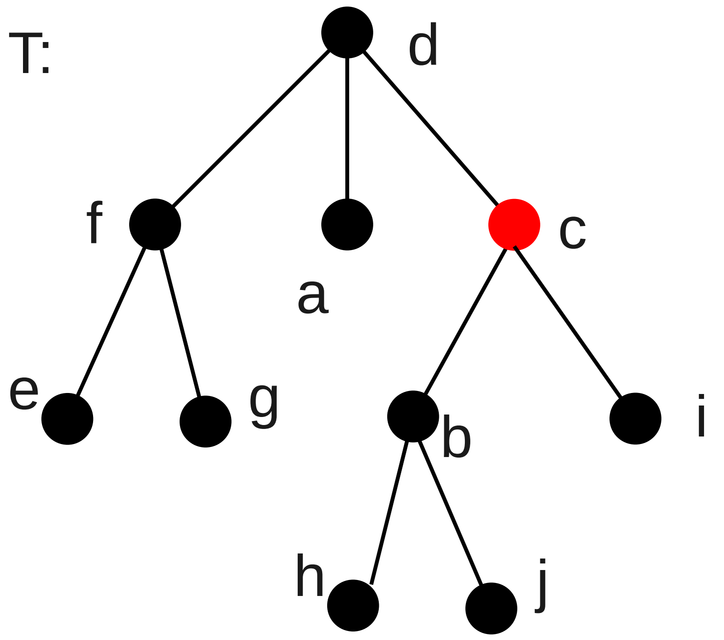


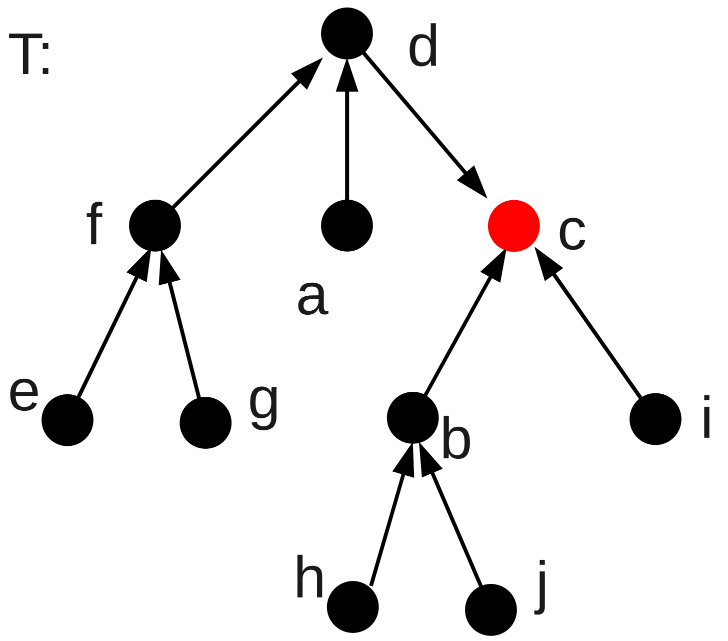
Euler Tour:

(c,i) → (i,c) → (c,d) → (d,a) → (a,d) → (d,f) → (f,g) → (g,f) → (f,d) → (d,c) → (c,b) → (b,j) → (j,b) → (b,h) → (h,b) → (b,c) → NIL

- The edge (r,v_i) appears before (v_i,r).
- So, if u precedes v, then u = p(v). Orient the edge uv from v to u.

Rooting a Tree





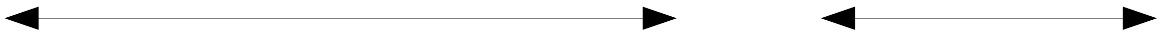
Euler Tour:

(c,i) → (i,c) → (c,d) → (d,a) → (a,d) → (d,f) → (f,g) → (g,f) → (f,d) → (d,c) → (c,b) → (b,j) → (j,b) → (b,h) → (h,b) → (b,c) → NIL

- So, if u precedes v, then u = p(v). Orient the edge uv from v to u.
- To know the above, use list ranking on the Euler path.

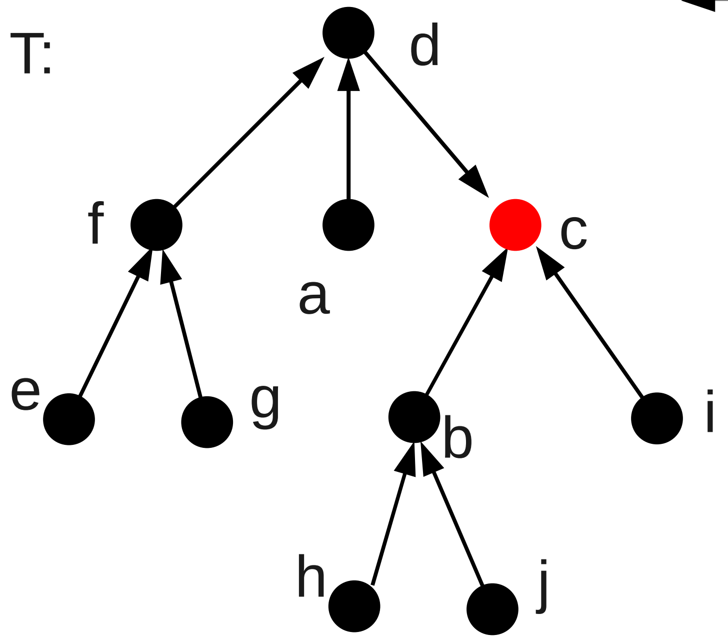
Preorder Traversal

ci-ic-cd-da-ad-df-fe-ef-fg-gf-fd-dc-cb-bj-jb-bh-hb-bc



Euler path Entering node d and its subtree

Euler path at node b and its subtree



- Euler tour can be used to get a preoder number for every node.
- Associate meaning to the Euler tour.

Preorder Traversal

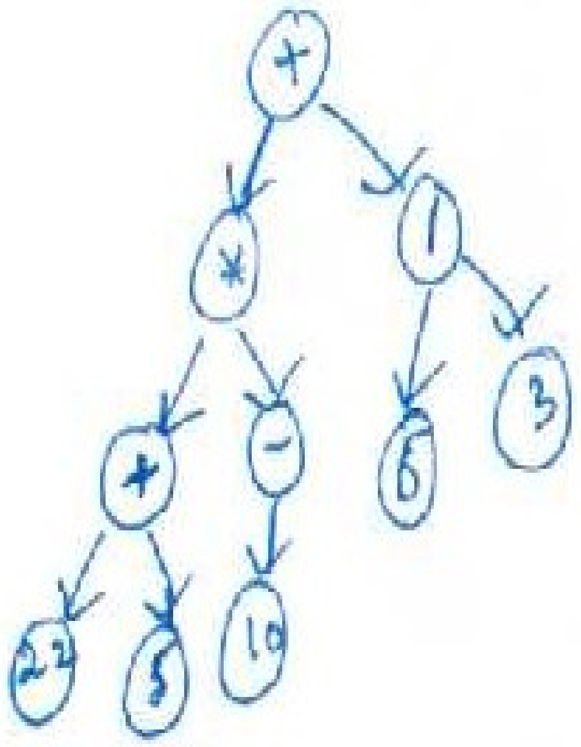
- In preorder traversal, a node is listed before any of the nodes in its subtree.
- In an Euler tour, nodes in a subtree are visited by entering those subtrees, and finally exiting to the parent.
- If we can therefore track the first occurrence of a node in the Euler path, then we can get the preorder traversal of the tree.

Postorder Traversal

- Similar rules can be designed for also postorder and inorder traversals.
- Inorder for binary trees only makes sense.
- Next we see how to process expression trees.

Expression Trees

- Expression trees are trees with operands at the leaf nodes, and operators at the internal nodes.
- Our interest is to evaluate the result of an expression represented by its expression tree.
- We would limit ourselves to binary operators.
  - Can also convert non-binary cases to the case of binary operators.
- However, the expression tree need not be balanced, or height in Θ(log n).



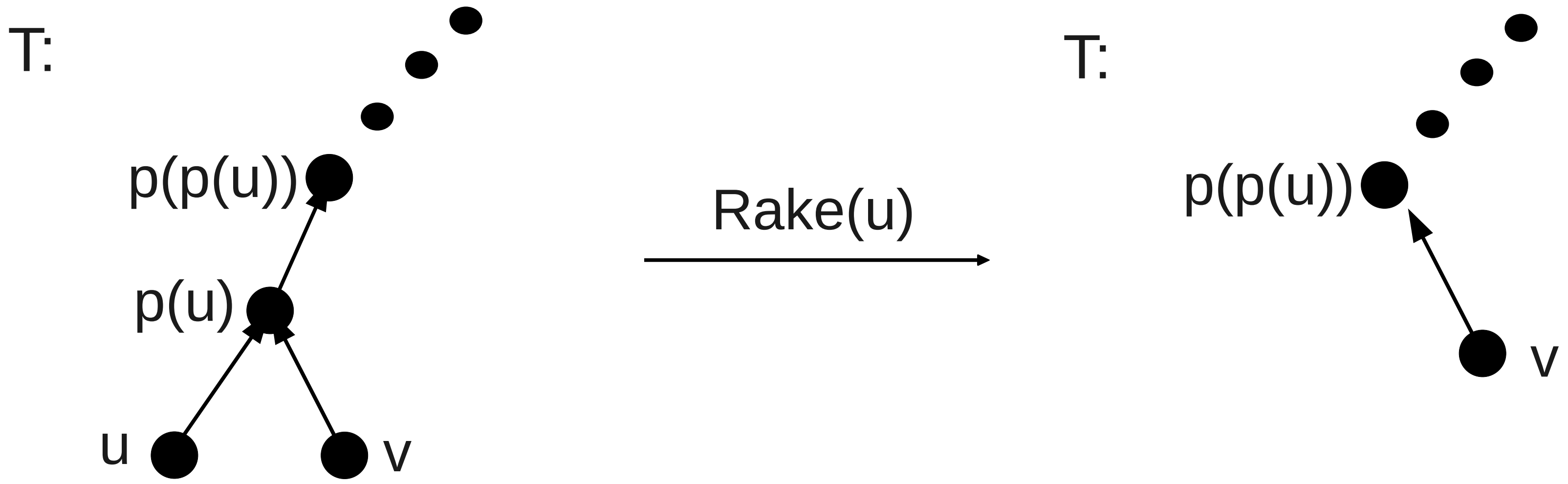
Expression Tree Evaluation

- Cannot directly apply the standard technique of
  - All the penultimate level nodes, then the next level etc.
    - Tree may not be balanced, and could have very few nodes at the penultimate level.
  - All leaves first cannot be processed at once
    - At an internal node, both operands may not be evaluated yet.

Two Steps

- A RAKE technique that contracts a tree
  - rake: 1.an agricultural implement with teeth or tines for gathering cut grass, hay, or the like or for smoothing the surface of the ground.
    2. any of various implements having a similar form, as a croupier's implement for gathering in money on a gaming table.
- Applying the rake technique to evaluate subexpressions.

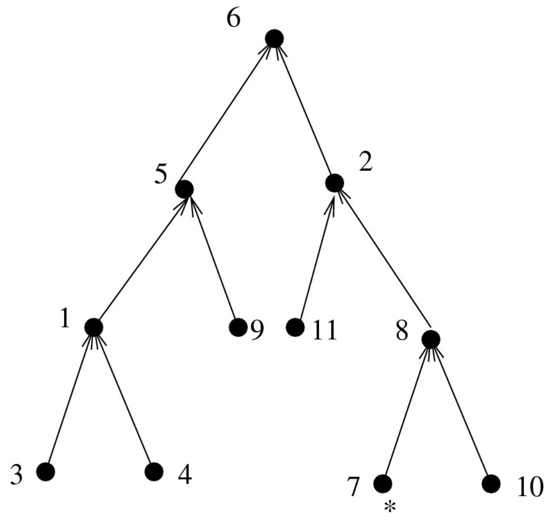
The Rake Technique



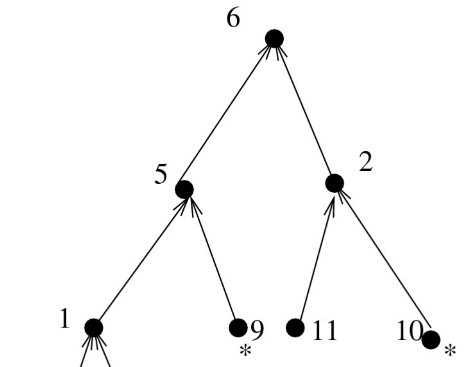
- T = (V, E) be a rooted tree with r as the root and p() be the parent function
- One step of the rake operation at a leaf u with p(u) $\neq$ r involves:
  - Remove nodes u and p(u) from the tree.
  - Make the sibling of v as the child of p(p(u)).

The Rake Technique

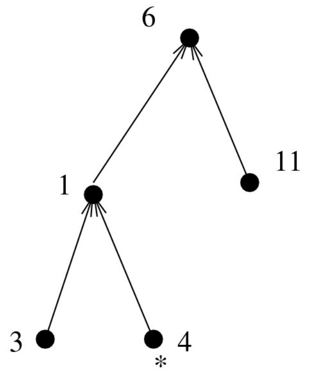
- Why is this good?
- Can be applied simultaneously at several leaf nodes in parallel.
- Which ones?
  - All leaves which do not share the same parent, essentially non-siblings.



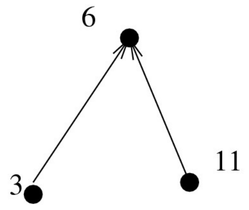
(a)



(b)



(c)



(d)

The Rake Technique

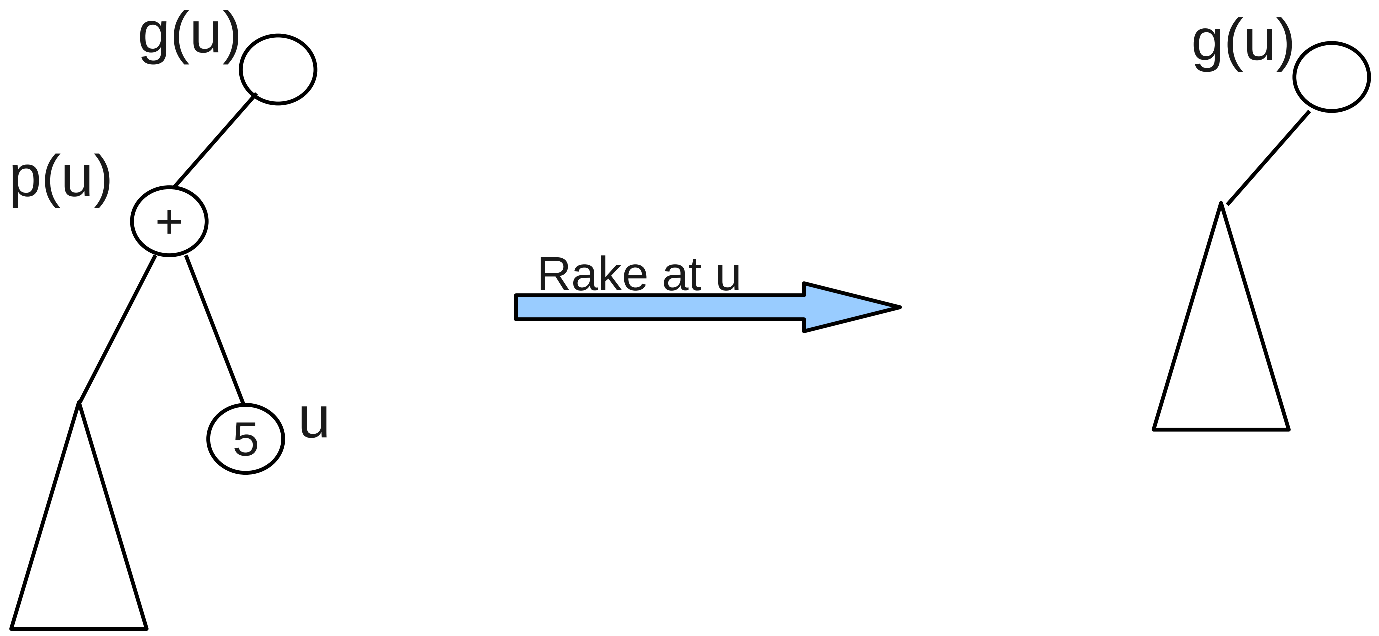
```
Algorithm ShrinkTree(T)
Step 1. Compute labels for the leaf nodes  consecutively,
        excluding the leftmost and the rightmost leaf
        nodes into an array A.
Step 2. for k iterations do
        2.1 Apply the rake operation to all the odd
            numbered leaves that are left children
        2.2 Apply the rake operation to all the odd
            numbered leaves that are right children
        2.3 Update A to be the remaining (even) leaf
  nodes.
end-for
End Algorithm.
```

- So, we apply the Rake technique on all non-adjacent sibling nodes.
- The algorithm is as shown.

Time Analysis

- Observation: Each application of the rake at all the leaves as given in the algorithm reduces the number of leaves by a half.
- Each application of Rake at a leaf node is an O(1) operation.
- So the total time is O(log n).
- The number of operations is O(n).
    - Similar observations hold.

Applying Rake to Expression Evaluation

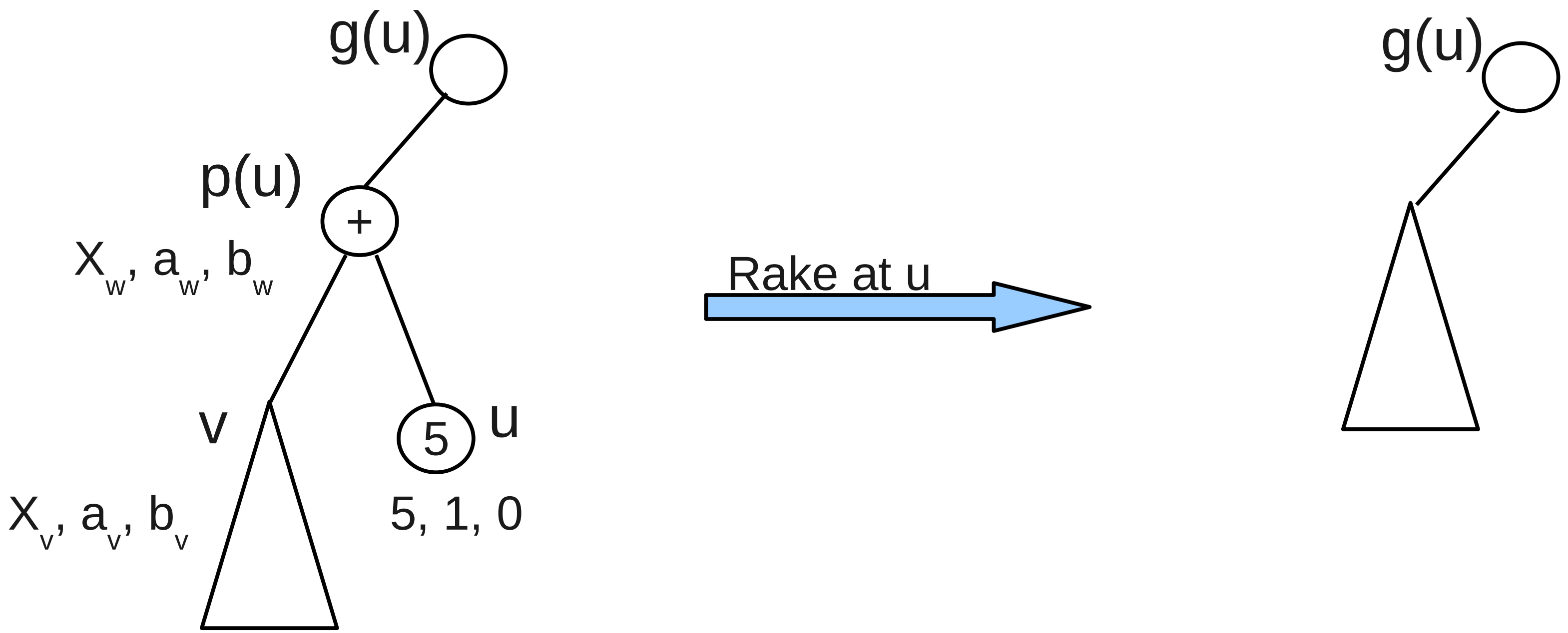


- Applying Rake means that we can process more than one leaf node at the same time.
- For expression evaluation, this may mean that an internal node with only one operand evaluated, also needs to be deleted.
  - Need to partially evaluate internal nodes.

Partial Evaluation

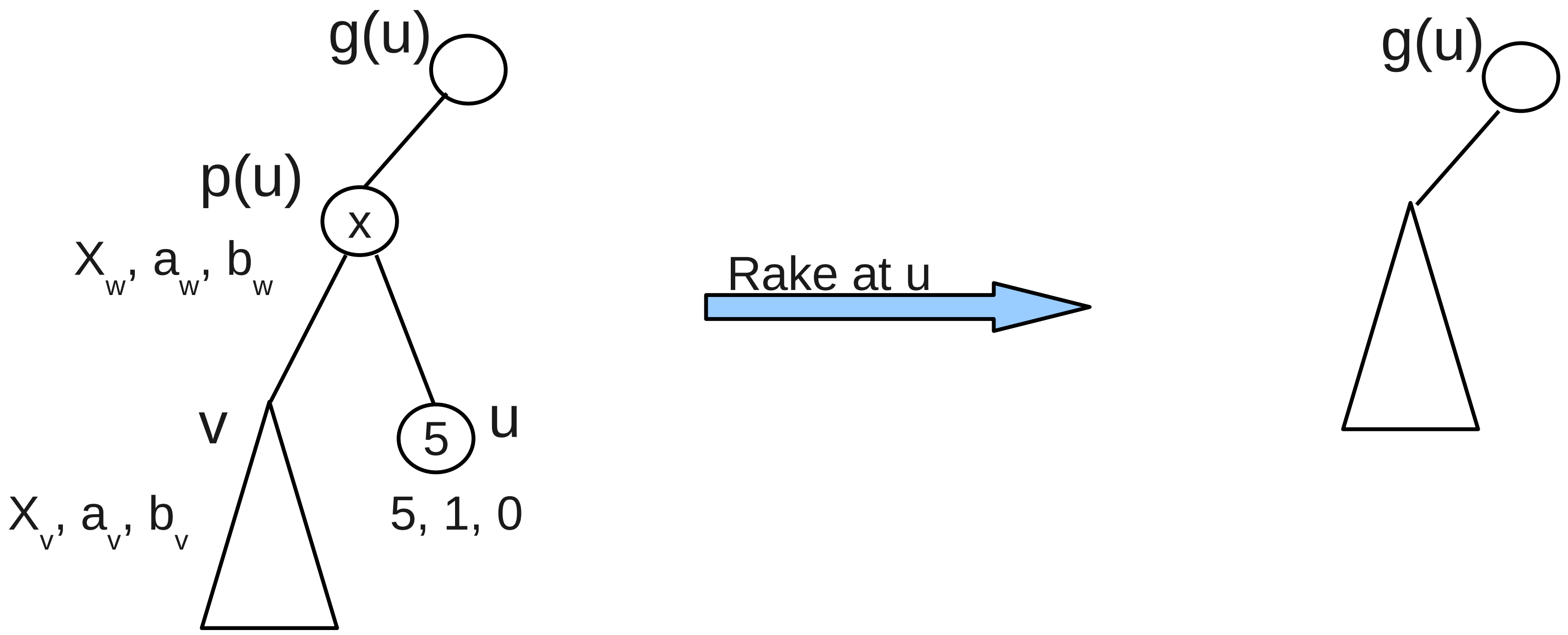
- Transfer the impact of applying the operator at p(u) to the sibling of u.
- Associate with each node u labels a_u and b_u so that $R_u = a_u X_u + b_u$.
  - X_u is the result of the subexpression, possibly unknown, at node u.
- Adjust the labels a_u and b_u during any rake operation appropriately.
- Initially, at each leaf node u, X_u equals the operand, $a_u = 1$, and $b_u = 0$.

Adjusting Labels



- Prior to rake at u, contribution of p(u) to g(u) is $a_w X_w + b_w$.
- $X_w = (a_u X_u + b_u) + (a_v X_v + b_v) = a_v X_v + (a_u X_u + b_u + b_v)$
- Therefore, adjust a_v and b_v as $a_w a_v$ and $a_w(a_u X_u + b_u + b_v)$.

Adjusting Labels



- Prior to rake at u, contribution of p(u) to g(u) is $a_w X_w + b_w$.
  - $X_w = (a_u X_u + b_u) \times (a_v X_v + b_v)$
- Therefore, adjust a_v and b_v as:
  - $a_v = a_w a_v (a_u X_u + b_u)$, $b_v = b_w + a_w b_v (a_u X_u + b_u)$.

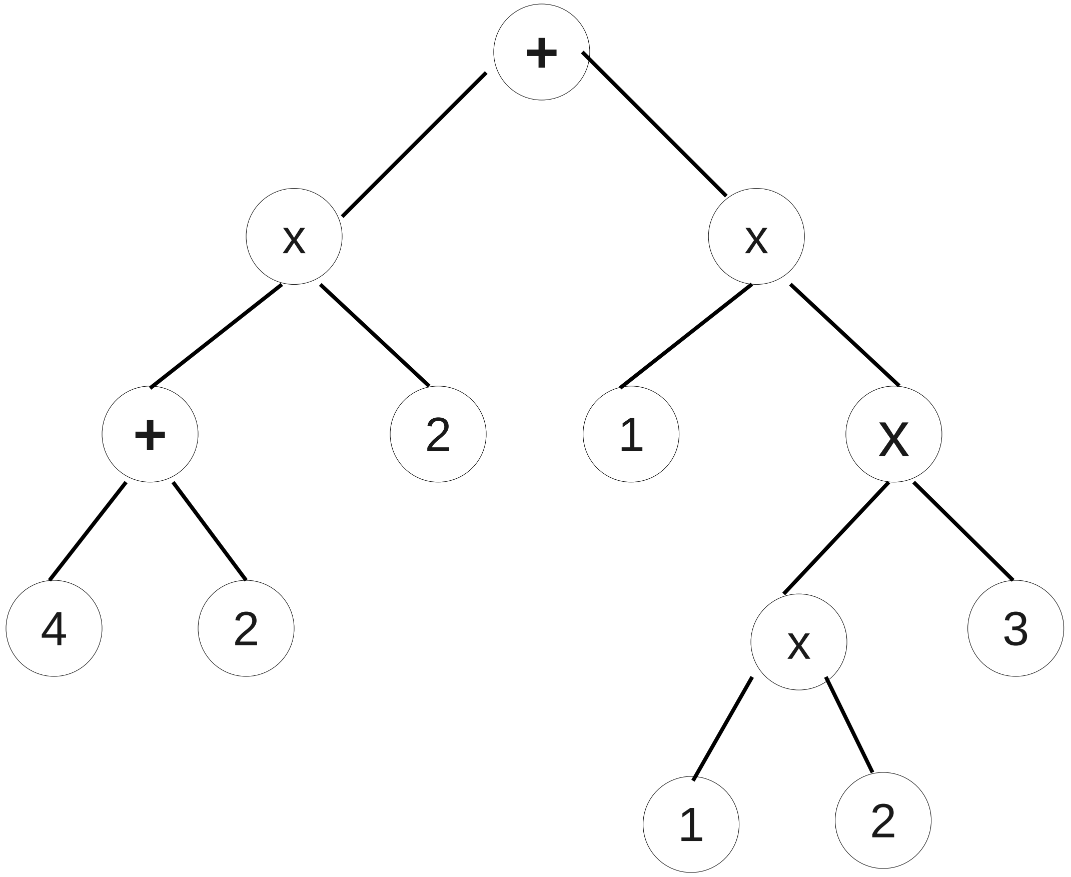
Adjusting Labels

- For other operators, proceed in a similar fashion.
- HW Problem.

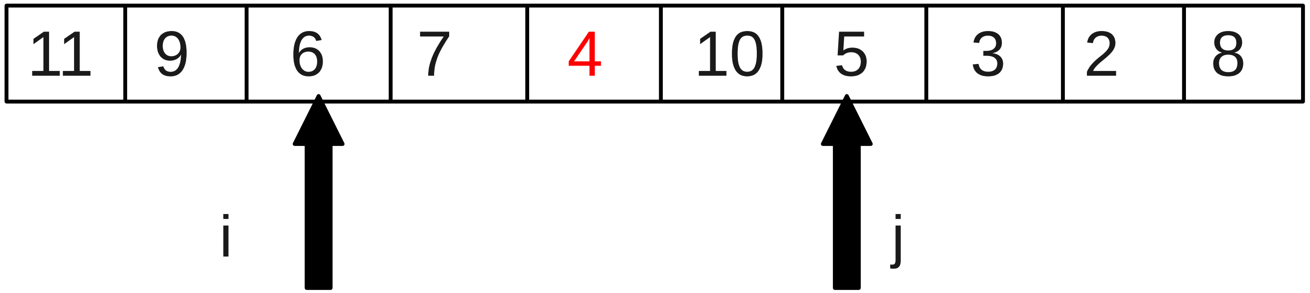
Expression Evaluation

- Parallel algorithm has the following main steps:
  - Rake the expression tree
  - Set up and adjust labels while raking
  - Stop when the tree has only three nodes, one operator and two operands as children.
  - Evaluate this three node tree.
- Theorem: Expression evaluation of an n-node expression tree can be done in parallel on an EREW PRAM using O(log n) time and O(n) operations.

Example

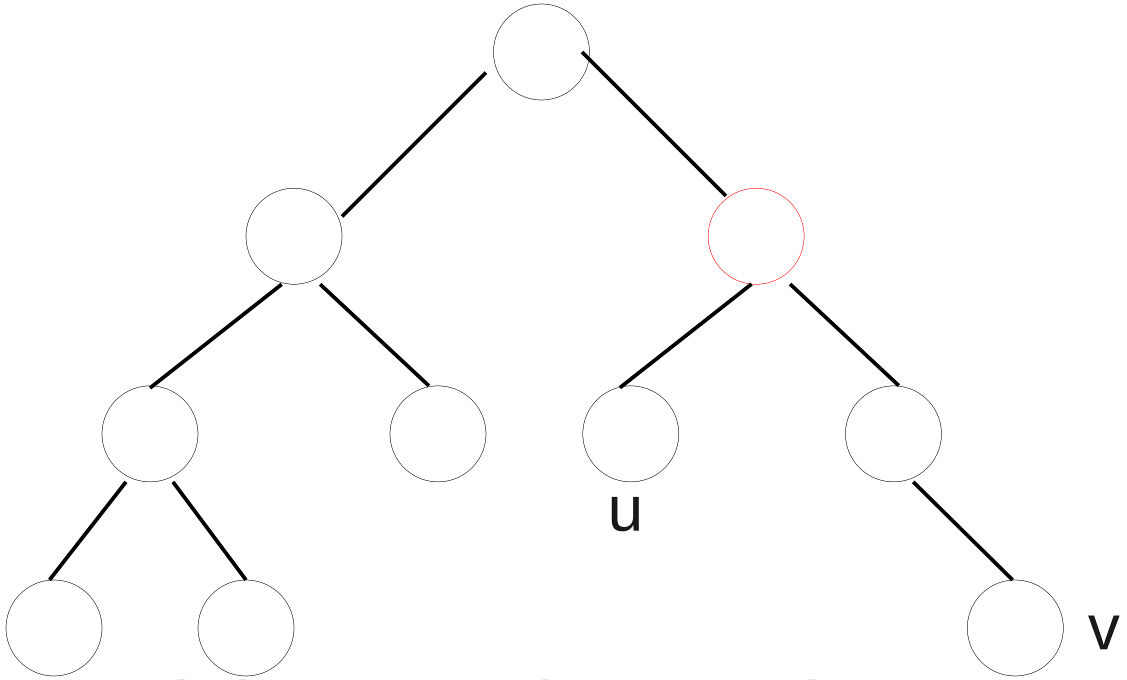


Range Minima



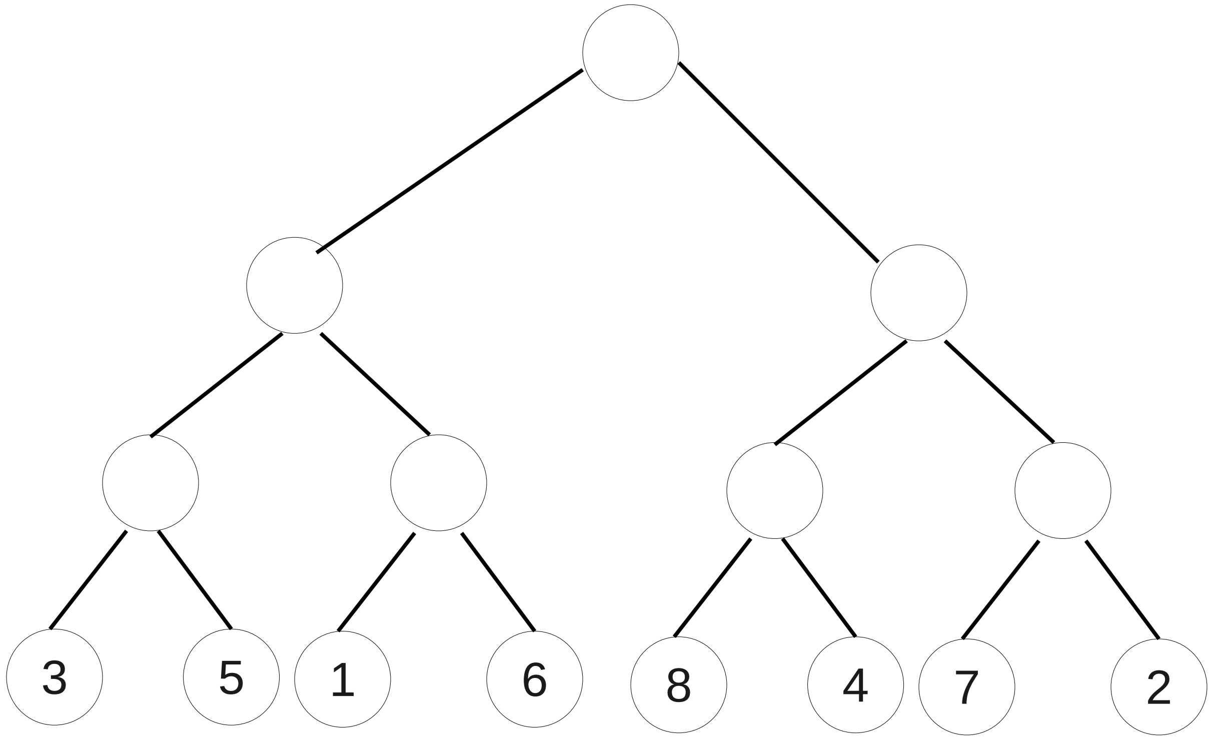
- Several geometric problems take the following flavor.
- Given a set of points S in a one dimensional space, preprocess S into a data structure D(S) so that:
  - Given a range of indices [i,j], report the element of S of least value between indices i and j.

Applications of Range Minima



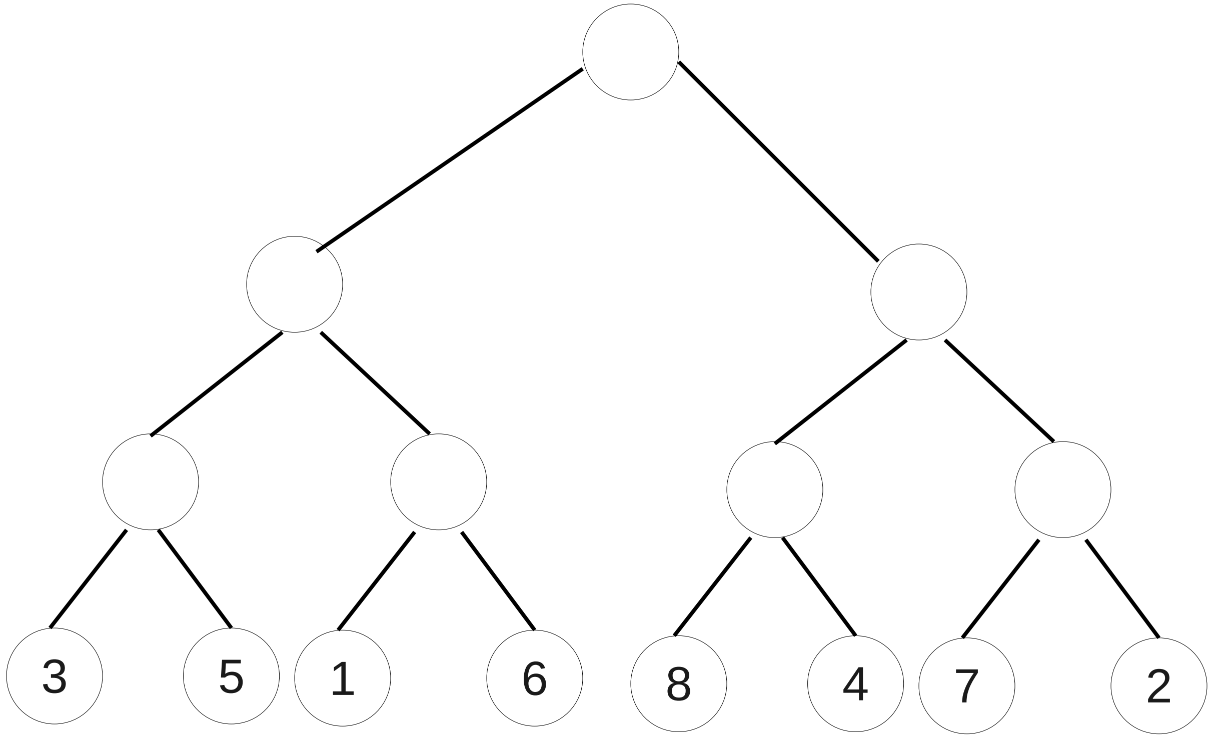
- Range minima can be used to solve the problem of finding the least common ancestor of two nodes in a tree.
  - Called as an LCA query, and is useful in several other settings.

Range Minima – Preprocessing



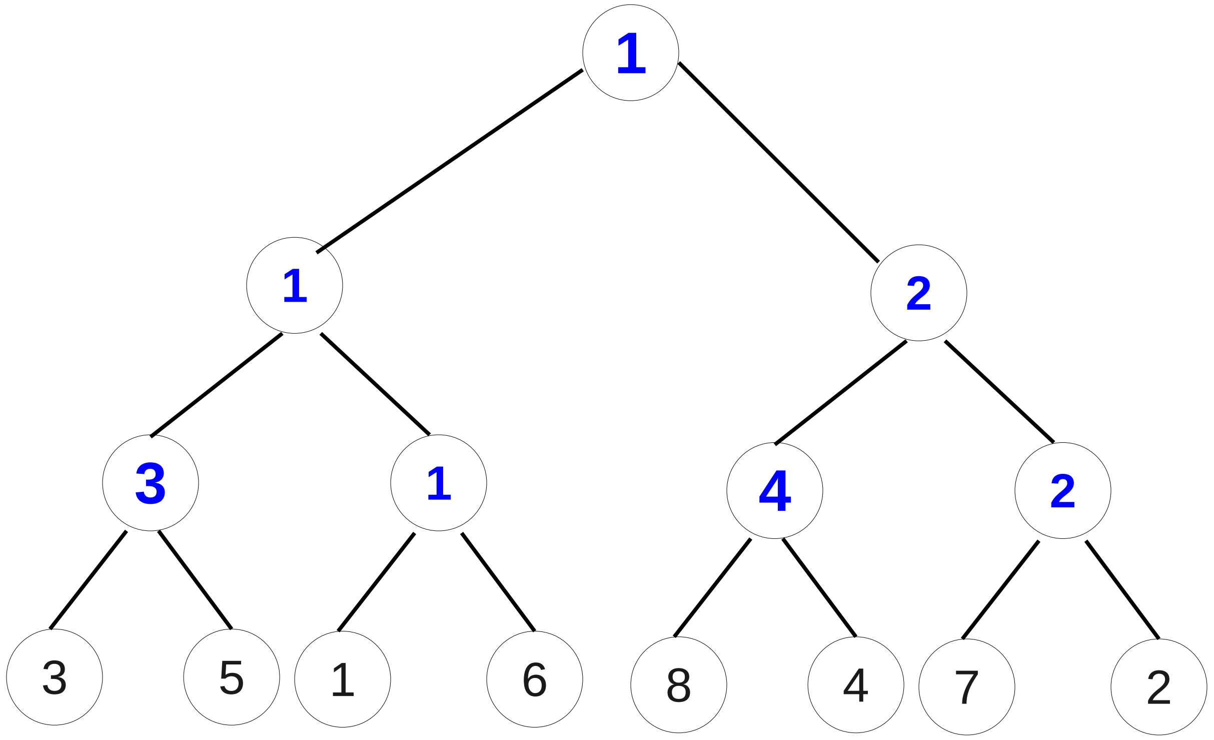
- Let n be the number of elements, and n = 2^k.
- Consider a full binary tree on the n elements.

Range Minima – Preprocessing



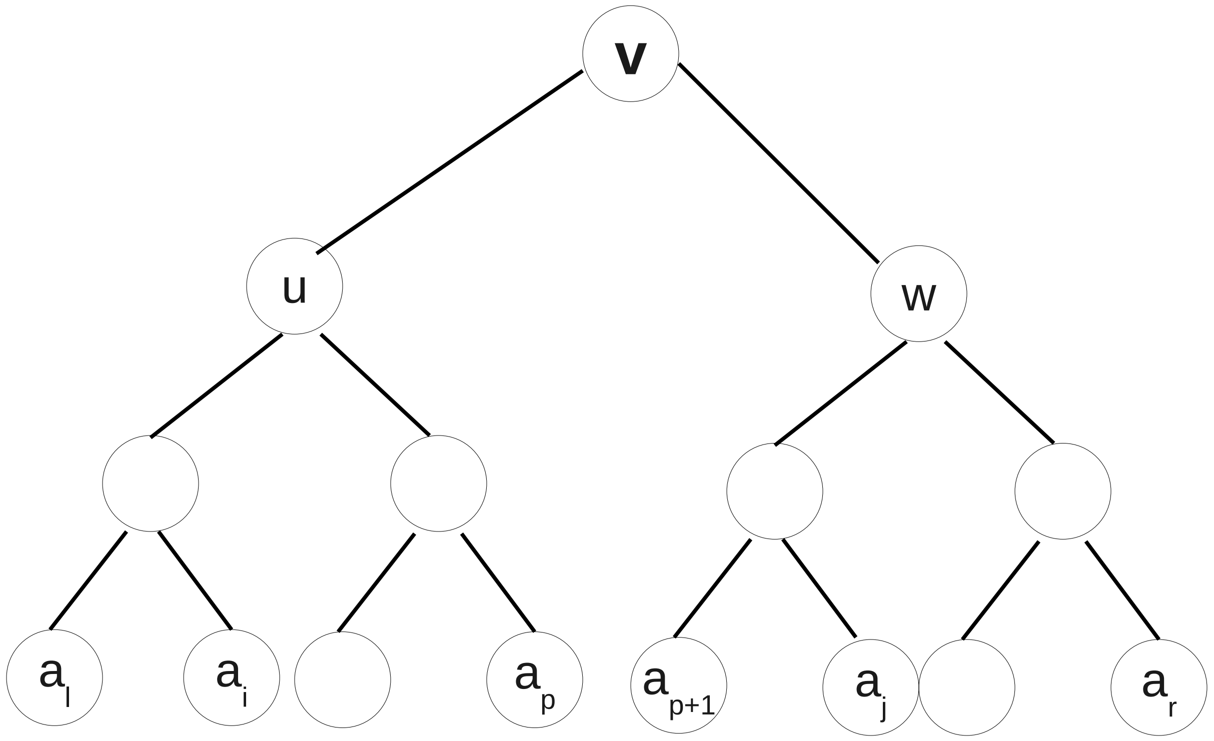
- Given two indices i and j, let v be the node in the tree that corresponds to the LCA of i and j.
  - No circular reasoning here. On a full binary tree, LCA is easy to find.

Range Minima – Preprocessing



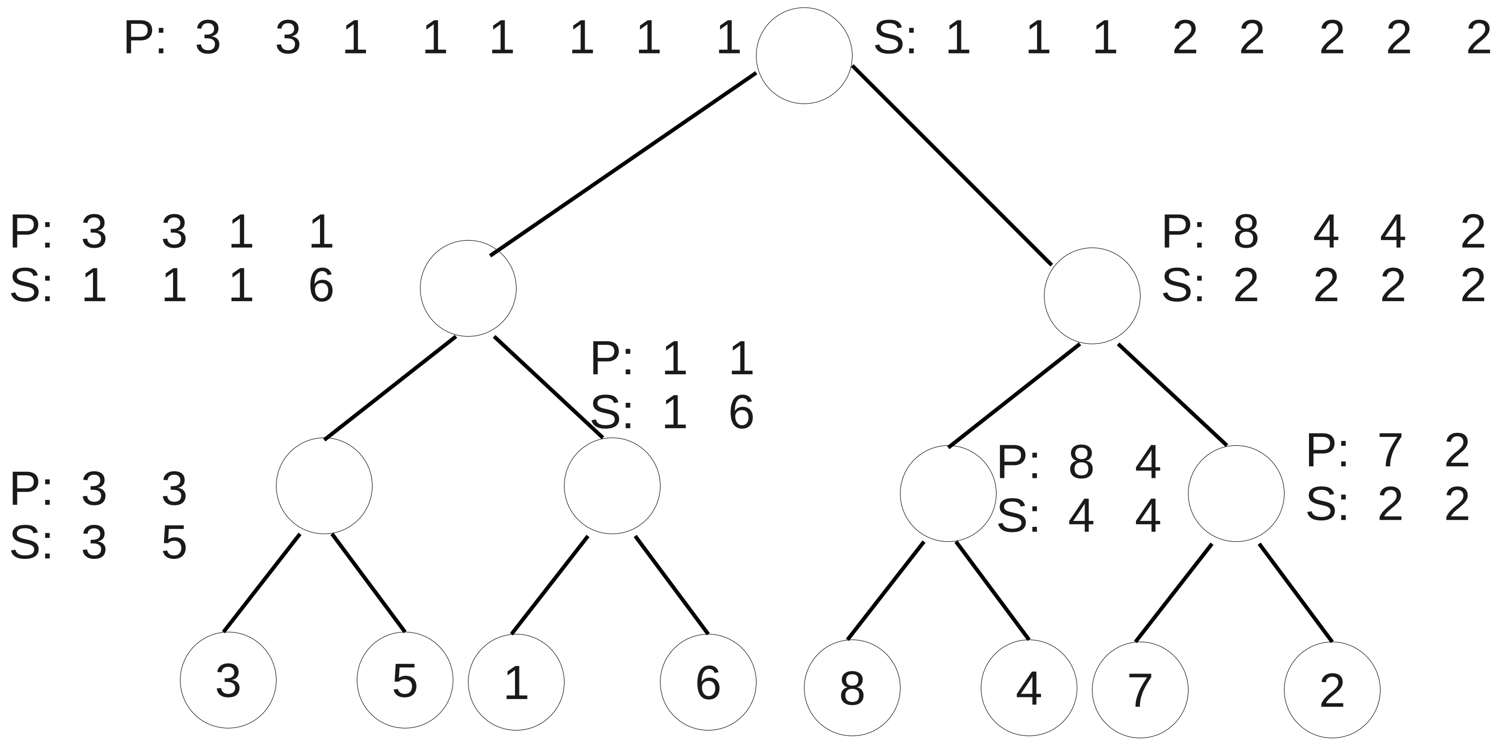
- Does not suffice if at every internal node, we just store the minima of the elements in that subtree.

Range Minima – Preprocessing



- Let u and w be the left and right child of v.
  - Also, let $A_v = (a_l, a_{l+1},\ldots,a_i, a_{i+1},\ldots a_j, a_{j+1},\ldots a_r)$.
- The required minima is the minimum of $\min\{a_i, a_{i+1},\ldots, a_p\}$ and $\min\{a_{p+1}, a_{p+2}, \ldots, a_j\}$.

Range Minima – Preprocessing

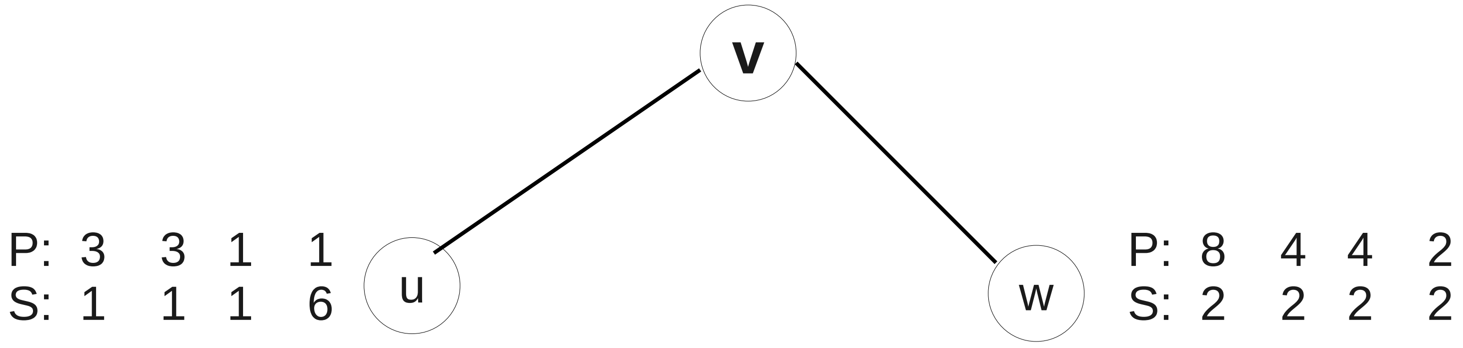


- For each node u, suppose we store the suffix and prefix minima of nodes in A_u.
- The required answer can be computed quickly.

Range Minima – Preprocessing

- For each node u, let P_u and S_u be the prefix and suffix minima arrays of elements in the subtree at node u.

Range Minima – Preprocessing



- Given a node v with children u and w, can actually compute P_v and S_v from P_u, P_w, and S_u, S_w respectively quickly.
  - How?

Range Minima – Preprocessing

- Given a node v with children u and w, can actually compute P_v and S_v from P_u, P_w, and S_u, S_w respectively quickly.
  - Let P_v be the concatenation of P_u and P_w.
  - The P_w part of P_v may change depending on the last element in P_u.
  - Similar rules apply for $S_v = S_u \text{ o } S_w$

Range Minima – Preprocessing

- Theorem: Given n elements in an array A, the array can be preprocessed in O(log n) parallel time and O(nlog n) operations so that range minima queries can be answered in O(1) time.
    - Requires CREW model.
        - Where do we require concurrent read?
    - Can be made to use O(n) operations. Use standard technique 1.
    - On the CRCW model, can actually reduce the parallel time to O(log log n) in O(n) operations.

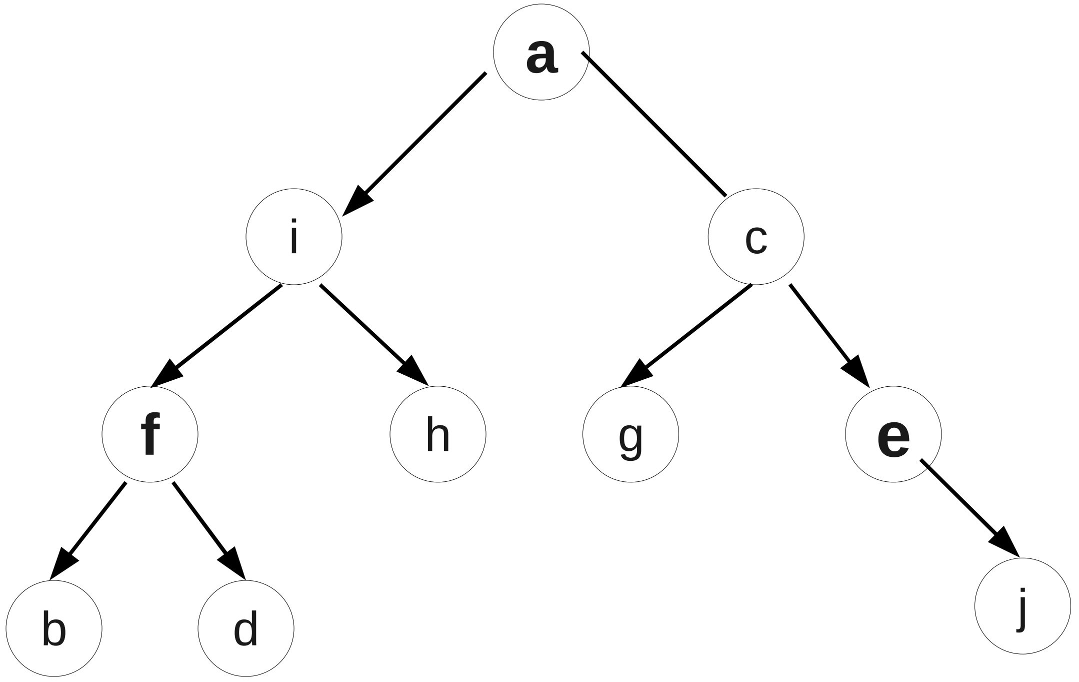
From Range Minima to LCA

- For a tree T rooted at r, let P be its Euler path with the edge (u,v) replaced by v.
- Compute the level of every node in the tree, with root at level 0.
  - Call this as the array Level[ ].
- Compute the leftmost and the rightmost occurrence of each node in P.
  - Call them as L(v) and R(v) for a node v.

From Range Minima to LCA

- Theorem: Let u and v be two vertices in T and L and R are given. Then,
  - L(u) < L(v) < R(u) if and only if u is an ancestor of v.
  - u and v do not share an ancestor-descendant relationship iff R(u) < L(v) or R(v) < L(u).
  - If R(u) < L(v) then the vertex with the minimum **Level** in the range [R(u), L(v)] is the LCA of u and v.
    - Therefore preprocess the Level array for range minima.

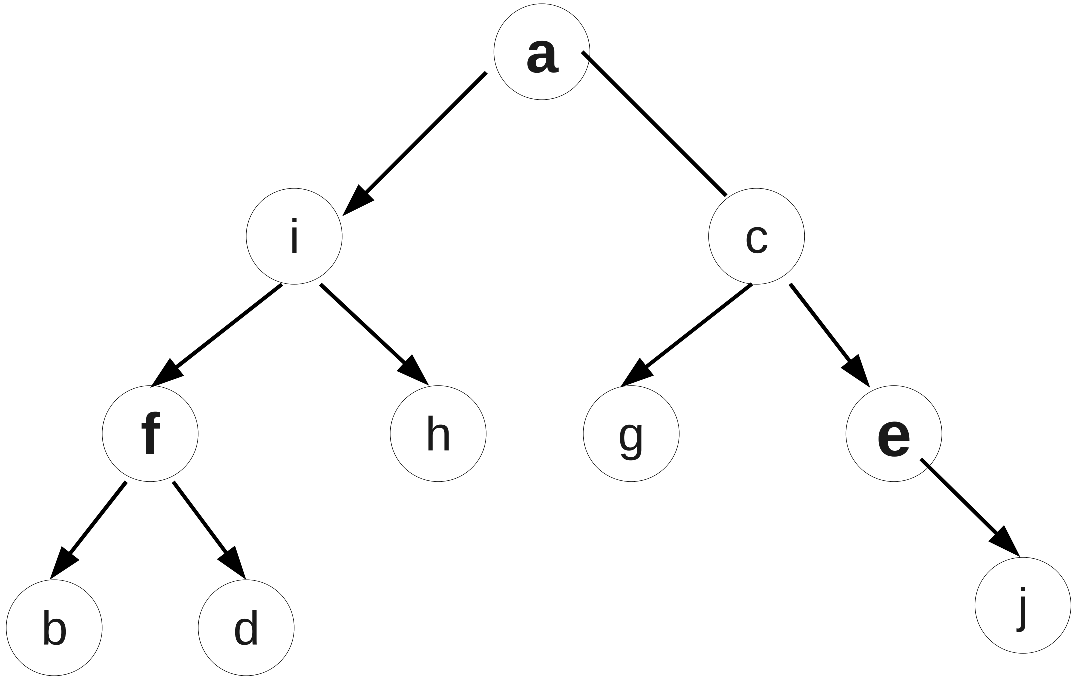
Example



(a,i) →(i,f) →(f,b)→ (b,f)→ (f,d)→ (d,f)→ (f,i)→ (i,h)→ (h,i)→ (i,a)→ (a,c)→ (c,g)→ (g,c)→ (c,e)→ (e,j)→ (j,e)→ (e,c)→ (c,a)

P: i →f →b→ f→ d→ f→ i→ h→ i→ a→ c→ g→c→ e→ j→ e→ c→ a

Example

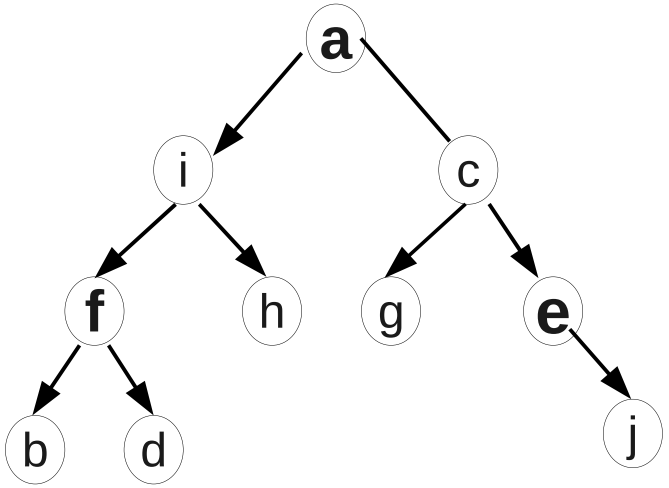


Node	L	R
a	0	18
b	3	3
c	11	17
d	5	5
e	14	16
f	2	6
g	12	12
h	8	8
i	1	9
j	15	15

P: a →i →f →b→ f→ d→ f→ i→ h→ i→ a→ c→ g→c→ e→ j→ e→ c→ a

Level **0 1 2 3 2 3 2 1 2 1 0 1 2 1 2 3 2 1 0**

Example



LCA of nodes g and j : R(g) = 12, L(j) = 15, $RM_{12,15}$(Level) = 1, LCA(g,j) = c

Node	L	R
a	0	18
b	3	3
c	11	17
d	5	5
e	14	16
f	2	6
g	12	12
h	8	8
i	1	9
j	15	15

P:	a →	i →	f →	b →	f →	d →	f →	i →	h →	i →	a →	c →	g →	c →	e →	j →	e →	c →	a
Level	**0**	**1**	**2**	**3**	**2**	**3**	**2**	**1**	**2**	**1**	**0**	**1**	**2**	**1**	**2**	**3**	**2**	**1**	**0**

Graph Algorithms