Parallel Algorithms

List ranking

Jesper Larsson Träff
traff@par.tuwien.ac.at

Parallel Computing, 184-5
Favoritenstrasse 16, 3. Stock

Sprechstunde: by email-appointment

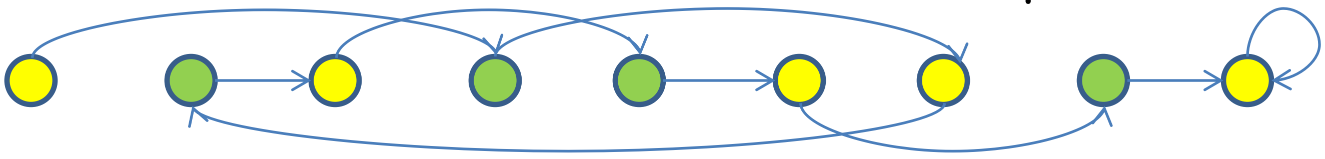
Part 3

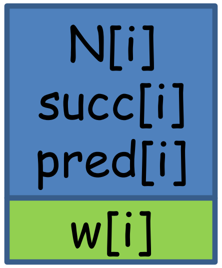
List ranking and applications to trees

1. List ranking: Cole-Vishkin
2. List ranking: Anderson-Miller
3. List ranking: Kruskal-Rudolph-Snir
4. Tree computations

The data-dependent prefix-sums problem: List ranking

Given list L defined by successor pointers p(i) pointing to next element of i for each i; last element has p(i)=†





List elements stored in arrays in some order. Weight w[i] associated with element i, associative operation „+" on weights. Other information for list elements stored in arrays indexed by i

- List ranking: compute for each i the „distance" to end of list
- Data-dependent prefix-sums: compute for each i w[i]+w[p(i)]+w[p(p(i))]+…+w[p^k(i)], where p(p^k)(i)=†,k≥0

Basic list ranking/data-dependent prefix with Wyllie's algorithm: W(n) = O(n log n), T(n) = O(log n) on an EREW PRAM

```
par (0≤i<n)  N[i] = p(i); d = 1;
while (d<n) {
   par (0≤i<n) {
      if (N[i]≠† and N[N[i]]≠†) {
         w[i] = w[i]+w[N[i]];
         N[i] = N[N[i]];
    }
    d = d*2;
  }
}
```

Since the sequential complexity of the problem is obviously O(n), Wyllie's algorithm is not work-optimal

Major obstacle

Goal: List ranking in O(log n) time and O(n) work, O(n/p+log n) steps

[J. C. Wyllie: The complexity of parallel computations. PhD thesis, 1979]

Basic list ranking/data-dependent prefix with Wyllie's algorithm: W(n) = O(n log n), T(n) = O(log n) on an EREW PRAM

```
par ($0\le i<n$)  N[i] = p(i); d = 1;
while ($d<n$) {
   par ($0\le i<n$) {
      if (N[i]≠† and N[N[i]]≠†) {
         w[i] = w[i]+w[N[i]];
         N[i] = N[N[i]];
    }
    d = d*2;
  }
}
```

Obervation:
If the list of length n can be shrunk to a list of length n/log n (in such a way that the ranks in the original list can be recomputed), Wyllie's algorithm can be used on in shorter list since O(log(n/log n)) = O(log n) and O((n/log n) log(n/log n)) = O(n)

[J. C. Wyllie: The complexity of parallel computations. PhD thesis, 1979]

Basic list ranking/data-dependent prefix with Wyllie's algorithm: $W(n) = O(n \log n)$, $T(n) = O(\log n)$ on an EREW PRAM

```
par (0≤i<n)  N[i] = p(i); d = 1;
while (d<n) {
   par (0≤i<n) {
      if (N[i]≠† and N[N[i]]≠†) {
        w[i] = w[i]+w[N[i]];
        N[i] = N[N[i]];
    }
    d = d*2;
  }
}
```

General outline:

1. Find work-optimal, fast way to shrink list
2. Apply Wyllie
3. Reverse step 1

[J. C. Wyllie: The complexity of parallel computations. PhD thesis, 1979]

Definition:
An independent set of a list is a set of nodes I, such that if i is in I, then p(i) is not in I

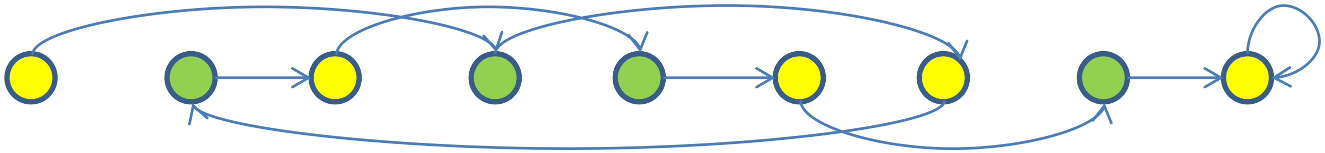
Given an independent set, all nodes of I can be removed (spliced out) in O(n) work and O(1) time steps:

```
par (i) if (succ[i]≠†) pred[succ[i]] = i; // make list doubly linked
```

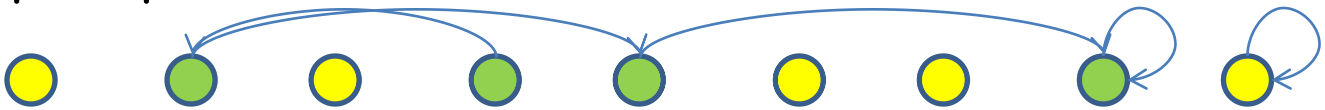
```
par (i in I) {
  if (pred[i]≠†) w[pred[i]] = w[pred[i]]+w[i];
  if (pred[i]≠†) succ[pred[i]] = succ[i]; // splice out i
  if (succ[i]≠†) pred[succ[i]] = pred[i];
  push(i,…,pred[i],succ[i]); // store pred,succ for later splice in
}
```

EREW

🟡 Nodes of independent set



After splice out (note: spliced-out elements still have succ and pred pointers)



Restoring spliced out elements can likewise be done in $O(1)$ time and $O(n)$ work

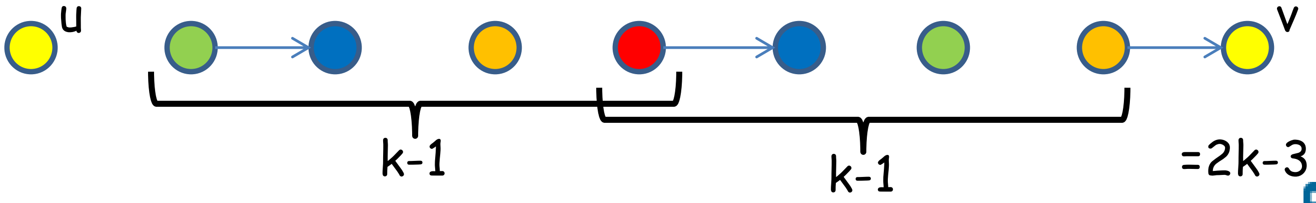
```
par (i in I) {
   if (pred[i]≠†) succ[pred[i]] = i;
   if (succ[i]≠†) pred[succ[i]] = i;
   if (succ[i]≠†) w[i] = w[i]+w[succ[i]];
}
```

Colorings to independent sets:

Let a k-coloring c of a list be given. A local minimum (maximum) is a node u such that the c(u)<c(pred(u)) and c(u)<c(succ(u))

Lemma: Given a k-coloring of a list, the set of local mimina (maxima) is an independent set of size $\Omega(n/k)$. The independent set can be computed in O(1) time steps and O(n) operations

Proof: Let u and v be local minima with no other local minima between them. They cannot be adjacent. The colors from u to v form an increasing and then a decreasing sequence, hence can be of length at most 2k-3. Therefore the independent set must be of size $\Omega(n/k)$. Determining that u is a local mimimum is trivially an O(1) operation



List ranking with coloring:

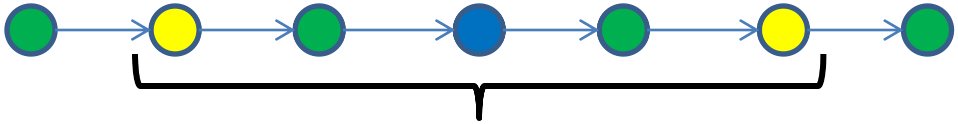
```
k = 0; nk = n; L is current list
while (nk>n/log n) {
  3-color L
  Determine local-mimima independent set Ik
  Contract L by splicing out all nodes of Ik
  k = k+1;
  nk = |L|;
  Compact list elements into array of size nk
}
Wyllie(L);
Splice in independent sets in reverse order
```

„CV86 algorithm“

Correct, since an independent set can be spliced out on an EREW PRAM

Complexity:

1. 3-coloring in O(log*n) time and O(nk log*nk) operations



2. Distance between local minimum independent nodes u and v is at most 5, thus size of |Ik| ≥ nk/5, so n(k+1) ≤ 4/5 nk. The number of iterations needed before nk≤n/log n is therefore (4/5)^k n ≤ n/log n ⇔ k log(5/4) ≥ log log n

3. Compacting the list into an array (excluding the spliced out elements) takes O(log n) time and O(nk) work (prefix-sums, compaction)

Lemma:

The list ranking problem can be solved (non-work optimally) in $T(n) = O(\log n \log\log n)$ time steps and $W(n) = O(n \log^*n)$ operations

Proof:

The number of operations is $O(\sum k \le \log\log n: nk \log^*nk) \le O(\sum k: ((4/5)^k n)\log^*n) = O(n \log^*n)$. The number of iterations of the contraction loop is $\log\log n$, each dominated by the prefix-sums compaction of $O(\log n)$ time steps.

The work is dominated by the ($\log^*n\ nk$) 3-coloring steps

To make the algorithm work-optimal, a work-optimal 3-coloring routine is needed

Colored prefix-sums problem: Given an array A of size n, elements from set S with an associative operation +, and an array of colors c, such that c[i] is the color of A[i]. The problem is to compute all colored prefix sums

$$B[i] = \sum_{0\leq j<i,\ c[j]=c[i]} A[j]$$

Theorem:
The colored prefix sums problem for R colors can be solved in O(R+log n) time and O(Rn/log n+n) operations. For R = O(log n) this is O(log n) time steps and O(n) work

Proof (somewhat difficult: pipelining): Exercise

Bucket-sort with colored prefix:
Given array X of n integers in the range [0,R-1] colored prefix operation can be used to sort the integers into an array Y

```
par (0≤i<n) {
  c[i] = X[i];
  A[i] = 1;
}
par (0≤i<R) {
  c[i+n] = i;
  A[i+n] = 1;
}
Colored-prefix(A,B,c,n+R);
par (0≤i<R) C[i] = B[i+n];
ExScan(C,R);
par (0≤i<n) Y[C[c[i]]+B[i]] = X[i]
```

Bucket-sort with colored prefix:
Given array X of n integers in the range [0,R-1] colored prefix operation can be used to sort the integers into an array Y

```
par (0≤i<n) {
  c[i] = X[i];
  A[i] = 1;
}
par (0≤i<R) {
  c[i+n] = i;
  A[i+n] = 1;
}
Colored-prefix(A,B,c,n+R);
par (0≤i<R) C[i] = B[i+n];
ExScan(C,R);
par (0≤i<n) Y[C[c[i]]+B[i]] = X[i]
```

🟨	🟩	🟨	🟧	🟧	🟩	🟥	🟥	🟥	🟨	🟦	🟧
0	0	1	0	1	1	0	1	2	2	0	2

C[0...R-1]	🟨 3	🟩 2	🟧 3	🟥 3	🟦 1
	🟨 0	🟩 3	🟧 5	🟥 8	🟦 11

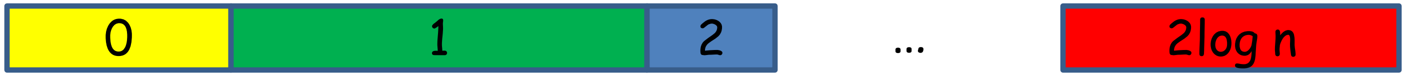
Not EREW

But can easily be made so (how?)

A work-optimal 3-coloring algorithm:

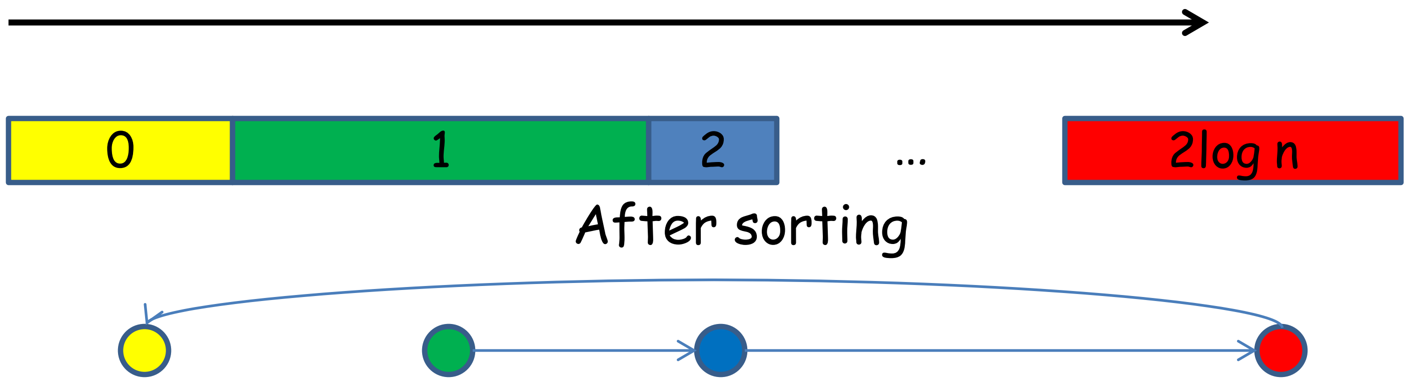
```
par (i) c[i] = i;
Reduce-color(C,c,n); // one step of deterministic coin tossing
Sort(C,c,n); // sort C (inplace) according to color
for (color=3; color<2ceil(log n); color++) {
  par (i) if (c[i]==color) c[i] = smallest possible color from {0,1,2}
}
```

Complexity: The first two steps are O(1) time, O(n) operations. Since O(log n) colors remain, sorting can be done in O(log n) time and O(n) operations. Now, since the vertices are in color order, the for loop can be carried out in a total of O(n) operations, obviously in O(log n) time steps.



After sorting

At most 2log n time steps, each in constant time with



After sorting

```
if (c[pred[i]]==0&&c[succ[i]]==0) c[i] = 1; else
if (c[pred[i]]==0&&c[succ[i]]==1) c[i] = 2; else
if (c[pred[i]]==0&&c[succ[i]]>1) c[i] = 1; else
…
```

Theorem:

The CV86 algorithm solves the list ranking problem work-optimally in $T(n) = O(\log n \log \log n)$ time steps and $W(n) = O(n)$ work on an EREW PRAM

[Richard Cole, Uzi Vishkin: Deterministic coin tossing with applications to optimal parallel list ranking. Information and Control, 70(1):32-53, 1986]

Digression:

A p-processor priority multi-prefix CRCW PRAM has an instruction

```
MPADD(register int x, int *a, register int prefix)
```

When processor i executes the operation in lock-step with other processors, the instruction computes in one time step

$$\text{prefix} = {*a} + \sum 0\le j<i,\ a(j)==a(i){:}\ x(j)$$
$${*a} = {*a}(\text{old value}) + \sum 0\le j<p,\ a(j)==a(i){:}\ x(j)$$

Observation: a p-processor priority multi-prefix CRCW PRAM can sort n integers in the range [0,n] in $O(n/p)$ time steps

Digression:

A p-processor priority multi-prefix CRCW PRAM has an instruction

```
MPADD(register int x, int *a, register int prefix)
```

When processor i executes the operation in lock-step with other processors, the instruction computes in one time step

$$\text{prefix} = {}^*a + \sum_{0\leq j<i,\ a(j)==a(i)} x(j)$$
$${}^*a = {}^*a(\text{old value}) + \sum_{0\leq j<p,\ a(j)==a(i)} x(j)$$

Historical facts:

- The SBPRAM (Ranade's simulation) realizes a p-processor priority multi-prefix CRCW PRAM
- Vishkin's XMT realizes a p-processor arbitrary multiprefix CRCW PRAM

List ranking in O(log n) time and O(n) work: Andersson-Miller

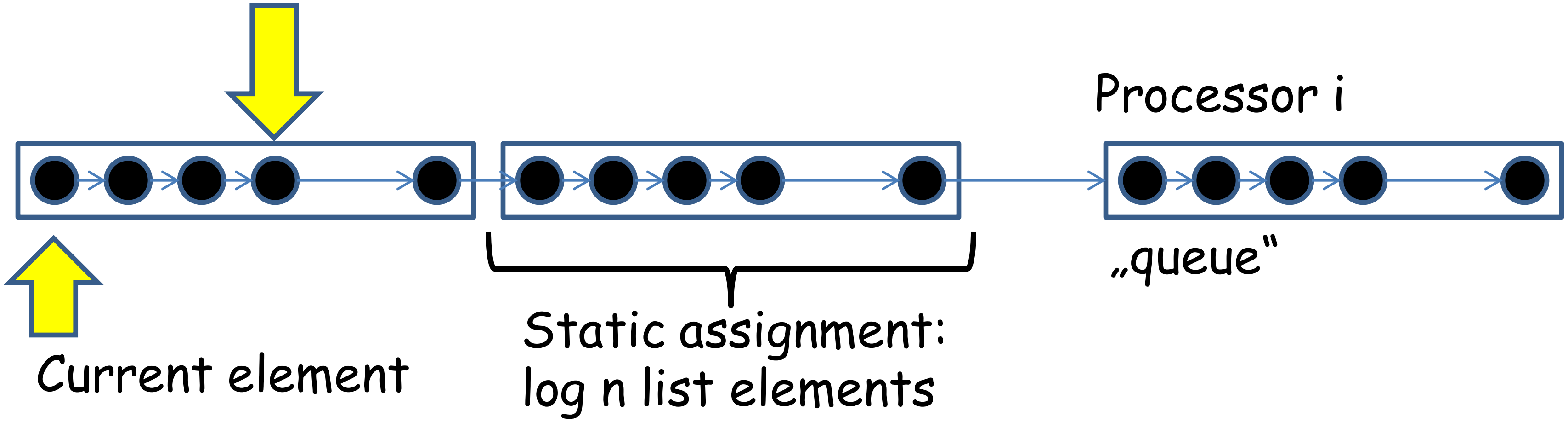
- Contraction of list to O(n/log n) elements, then Wyllie

- Static assignment of processors to n/p list elements („queue“)
- Simple processing (splice out or skip) of the assigned elements
- Coloring to bound work per processor

- Input list is doubly linked; can be done in O(1) time

[R. J. Anderson, G. L. Miller: Deterministic Parallel List Ranking. Algorithmica 6:859-868, 1991]

```
status: {ruler,subject,active,inactive,removed}
succ, pred:
```



Processor i

„queue"

Current element

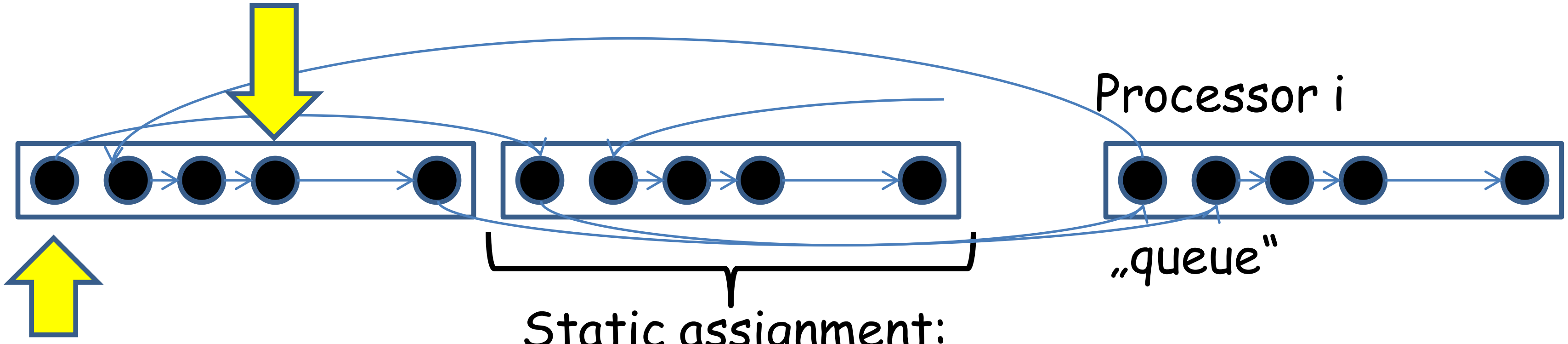
Static assignment:
log n list elements

- List element inactive->active, then ruler or subject: 🟡🟢🔵⚫🔴
- Ruler removes subjects
- Subjects turn to next element in queue

Processor k, 0≤k<n/p:

```
// initialization: only queue heads active
for (i=1; i<n/p; i++) queue[i].status = inactive;
i = 0; curr = &queue[0]; curr->status = active;
```

status: {ruler,subject,active,inactive,removed}
succ, pred:



Current element

Static assignment:
log n list elements

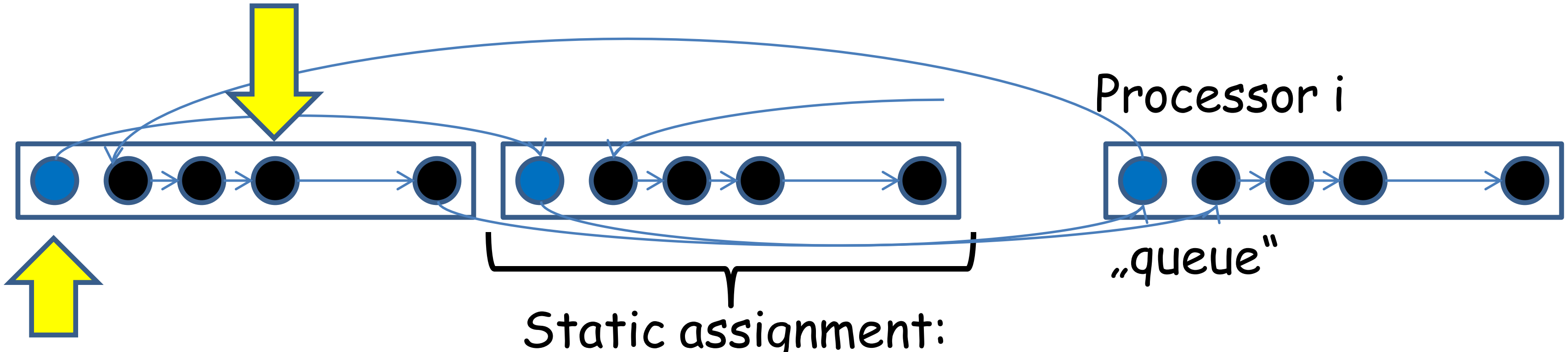
- List element inactive->active, then ruler or subject: 🟡🟢🔵⚫🔴
- Ruler removes subjects
- Subjects turn to next element in queue

Processor k, 0≤k<n/p:

```
// initialization: only queue heads active
for (i=1; i<n/p; i++) queue[i].status = inactive;
i = 0; curr = &queue[0]; curr->status = active;
```

First element in each queue becomes active

status: {ruler,subject,active,inactive,removed}
succ, pred:



Processor i

„queue“

Current element

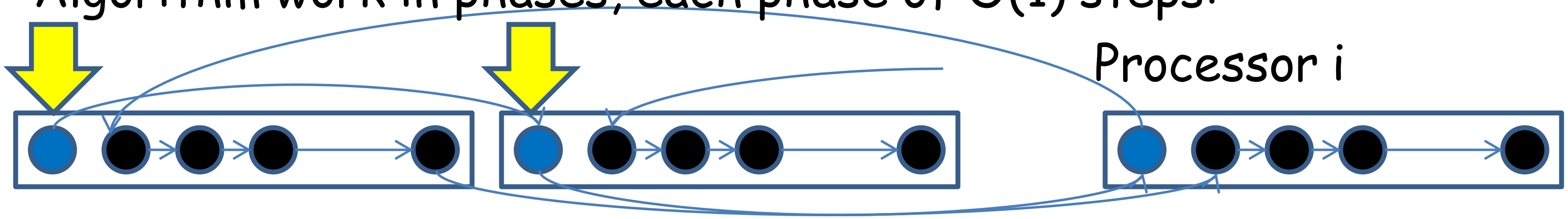
Static assignment:
log n list elements

- List element inactive->active, then ruler or subject: 🟡🟢🔵⚫🔴
- Ruler removes subjects
- Subjects turn to next element in queue

Processor k, 0≤k<n/p:

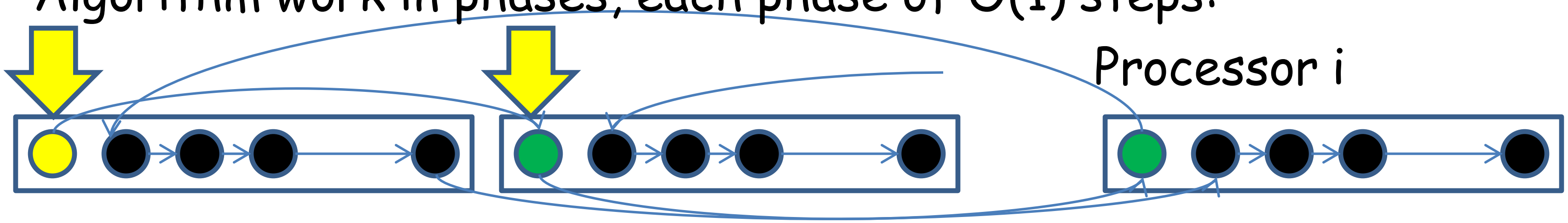
```
// initialization: only queue heads active
for (i=1; i<n/p; i++) queue[i].status = inactive;
i = 0; curr = &queue[0]; curr->status = active;
```

Algorithm work in phases, each phase of O(1) steps:



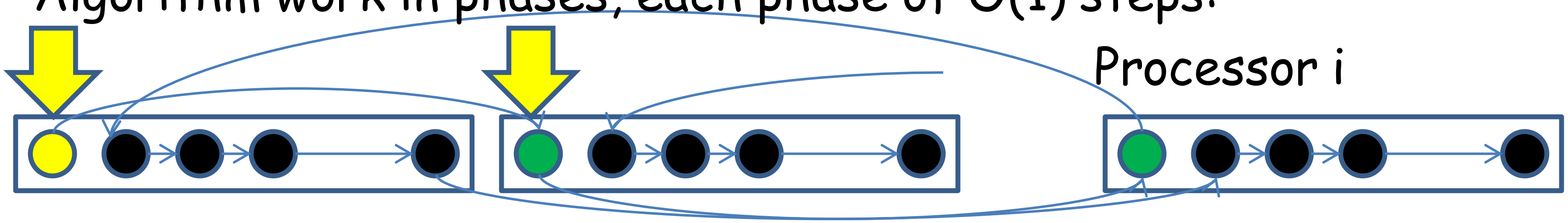
```
// Rulers splice out (see next)
// Promote to ruler/subject
if (curr->status==active) {
  if (curr->pred.status≠active&&curr->succ.status≠active) {
    SpliceOut(curr);
    curr->status = removed;
  } else {
    if (curr->pred.status≠active) curr->status = ruler;
    else curr->status = subject;
}
// advance to next element in queue (see next)
```

Algorithm work in phases, each phase of O(1) steps:



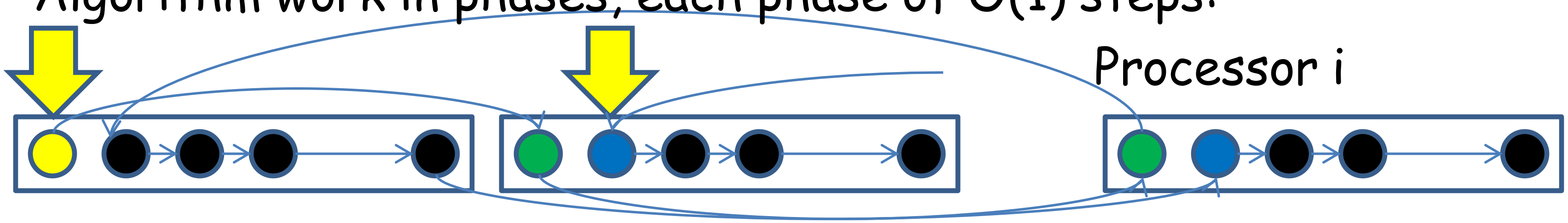
```
// Rulers splice out (see next)
// Promote to ruler/subject
if (curr->status==active) {
  if (curr->pred.status≠active&&curr->succ.status≠active) {
    SpliceOut(curr);
    curr->status = removed;
  } else {
    if (curr->pred.status≠active) curr->status = ruler;
    else curr->status = subject;
}
// Advance to next element in queue (see next)
```

Algorithm work in phases, each phase of O(1) steps:



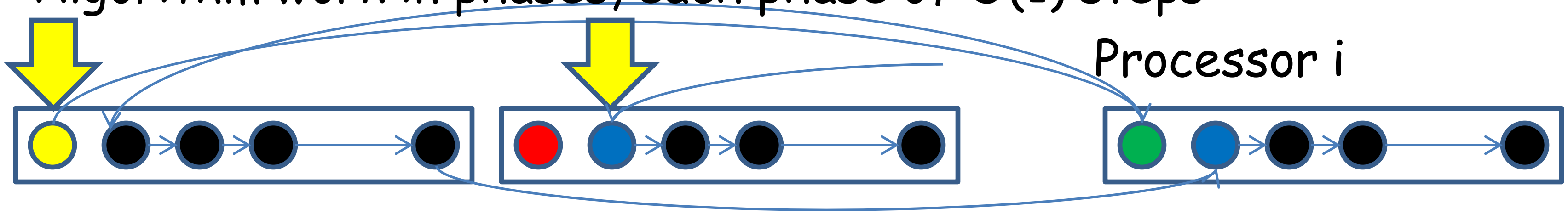
```
// Rulers splice out
if (curr->status==ruler) {
  SpliceOut(curr->succ); // succ pointer of curr updated
  if (curr->succ.status≠subject) curr->status = active;
}
// Promote to ruler/subject (see previous)
// Advance to next element in queue
if (curr->status==removed||curr->status==subject) {
  i++; if (i==n/p) done; else curr = &queue[i]; curr->status = active;
}
```

Algorithm work in phases, each phase of O(1) steps:



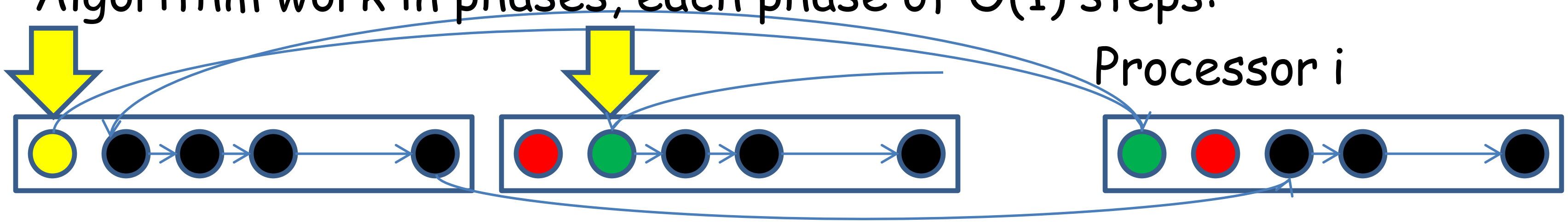
```
// Rulers splice out
if (curr->status==ruler) {
  SpliceOut(curr->succ); // succ pointer of curr updated
  if (curr->succ.status≠subject) curr->status = active;
}
// Promote to ruler/subject (see previous)
// Advance to next element in queue
if (curr->status==removed||curr->status==subject) {
  i++; if (i==n/p) done; else curr = &queue[i]; curr->status = active;
}
```

Algorithm work in phases, each phase of O(1) steps:



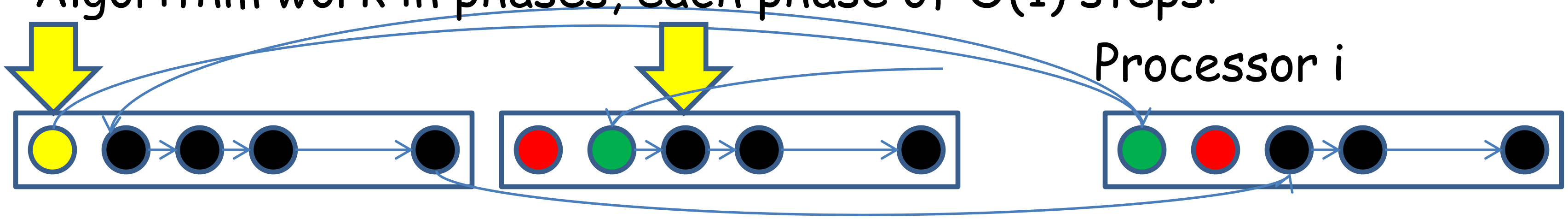
```
// Rulers splice out
if (curr->status==ruler) {
  SpliceOut(curr->succ); // succ pointer of curr updated
  if (curr->succ.status≠subject) curr->status = active;
}
// Promote to ruler/subject (see previous)
// Advance to next element in queue
if (curr->status==removed||curr->status==subject) {
  i++; if (i==n/p) done; else curr = &queue[i]; curr->status = active;
}
```

Algorithm work in phases, each phase of O(1) steps:



```
// Rulers splice out (see previous)
if (curr->status==active) {
  if (curr->pred.status≠active&&curr->succ.status≠active) {
    SpliceOut(curr);
    curr->status = removed;
  } else {
    if (curr->pred.status≠active) curr->status = ruler;
    else curr->status = subject;
}
```

Algorithm work in phases, each phase of O(1) steps:



```
// Rulers splice out (see previous)
// Promote to ruler (see rpevious)
// Advance to next element
if (curr->status==removed||curr->status==subject) {
  i++; if (i==n/p) done; else curr = &queue[i]; curr->status = active;
}
```

Complete phase of AM algorithm (version 1)

```
if (curr->status==ruler) {
  SpliceOut(curr->succ); // succ pointer of curr updated
  if (curr->succ.status≠subject) curr->status = active;
}
if (curr->status==active) {
  if (curr->pred.status≠active&&curr->succ.status≠active) {
    SpliceOut(curr);
    curr->status = removed;
  } else {
    if (curr->pred.status≠active) curr->status = ruler;
    else curr->status = subject;
}
if (curr->status==removed||curr->status==subject) {
  i++; if (i==n/p) done; else curr = &queue[i]; curr->status = active;
}
```

A phase has three steps

1. Splice out: Ruler splices out subject, if last becomes active
2. Promote: Active node splices itself out if adjacent elements both not active; otherwise becomes ruler if predecessor is not active, otherwise subject
3. Advance: subject or removed element goes to next element in queue

Observations:
Each phase takes O(1) steps; PRAM synchronization ensures that status changes take place at the same time. No adjacent elements are spliced out: EREW. Each processor has a queue of n/p elements – if no chain of subjects is longer than O(n/p), the algorithm would terminate with p elements after O(n/p) phases.

Lemma: Assume that (almost all of the) p processors are active at the start of each phase. The algorithm needs at most 4n/p phases to reduce the list to O(p) elements

Proof: Accounting scheme.

1. The current element is spliced out if it is isolated (adjacent elements not active): assign 1 unit
2. A ruler splices out a subject: assign $\frac{1}{2}$ unit
3. An element becomes subject: assign $\frac{1}{2}$ unit

To process all elements n units need to be assigned. Since in each phase a processor either splices out, or becomes ruler or subject; and since there are at least as many new subjects as rulers created in a phase, each phase assigns at least p/4 units. Number of phases is therefore k p/4 = n ⇔ k = 4n/p

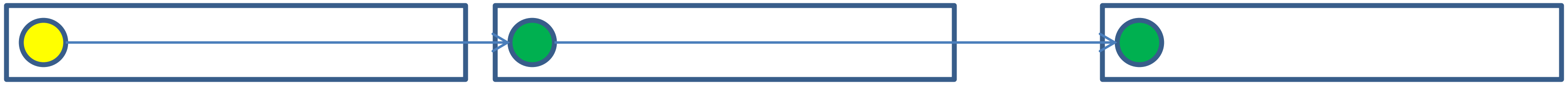
Problems:

1. Long chains (proportional to p elements) cause some processors to be active for $\Omega(n/p)$ phases
2. Many processors may run out of work after n/p phases (when queue becomes empty)

Solutions:

1. Coloring to break up long chains
2. Chains ordered according to estimated work, rulers are chosen as least work elements in chain

Chain: sublist of a ruler and its subjects, length of chain is O(p)



Complete phase of AM algorithm (version 1)

```
if (curr->status==ruler) {
  SpliceOut(curr->succ); // succ pointer of curr updated
  if (curr->succ.status≠subject) curr->status = active;
}
if (curr->status==active) {
  if (curr->pred.status≠active&&curr->succ.status≠active) {
    SpliceOut(curr);
    curr->status = removed;
  } else {
    if (curr->pred.status≠active) curr->status = ruler;
    else curr->status = subject;
}
if (curr->status==removed||curr->status==subject) {
  i++; if (i==n/p) done; else curr = &queue[i]; curr->status = active;
}
```

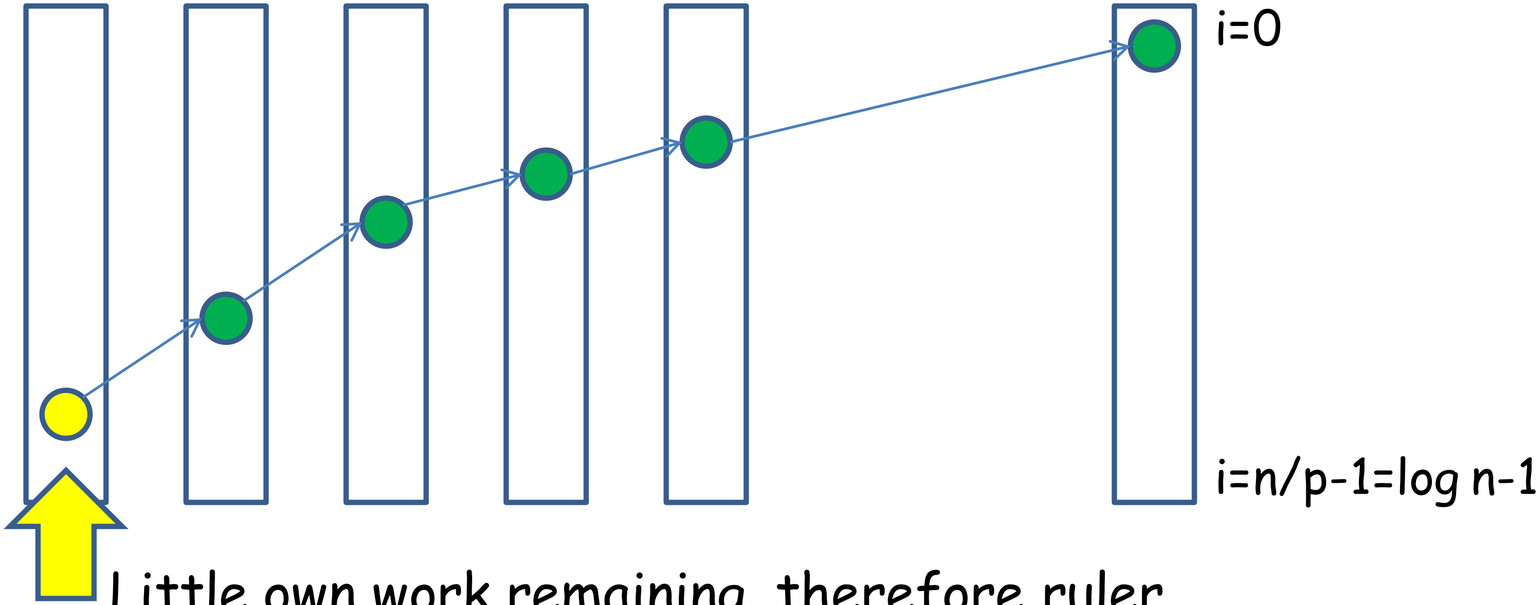
Break chains by O(log log n) chain coloring

Recap:
Start with coloring c[i] = i for each element, apply deterministic coin tossing twice to end up with a coloring with O(log log n) colors. Takes O(n) operations and O(1) time steps on an EREW. (Problem/exercise: make sure the recoloring works for sublists)

The coloring is used to find an independent set R with the property that for any node not in R, the distance to a node in R is at most log log n; called a log log n ruling set

Remedy 1: make subjects that are in R into new rulers, thus breaking chains into shorter chains of O(log log n) elements

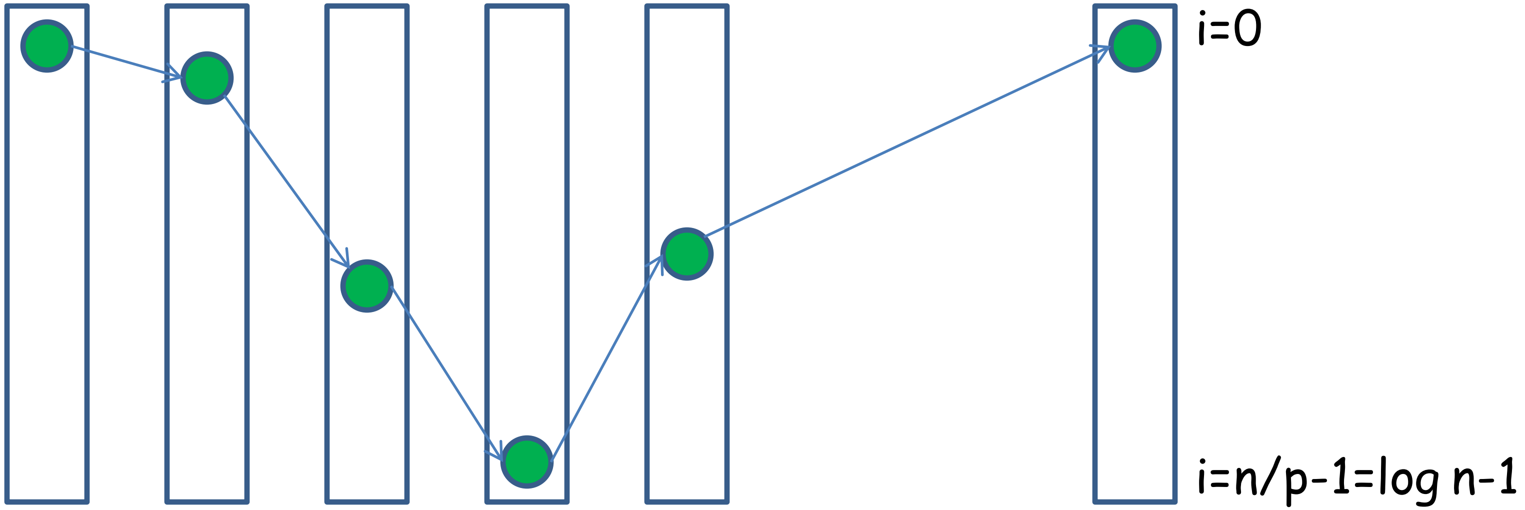
Remedy 2: Estimate the remaining work of each ruler by its position in the queue, and break chains into smaller chains of decreasing work



Little own work remaining, therefore ruler

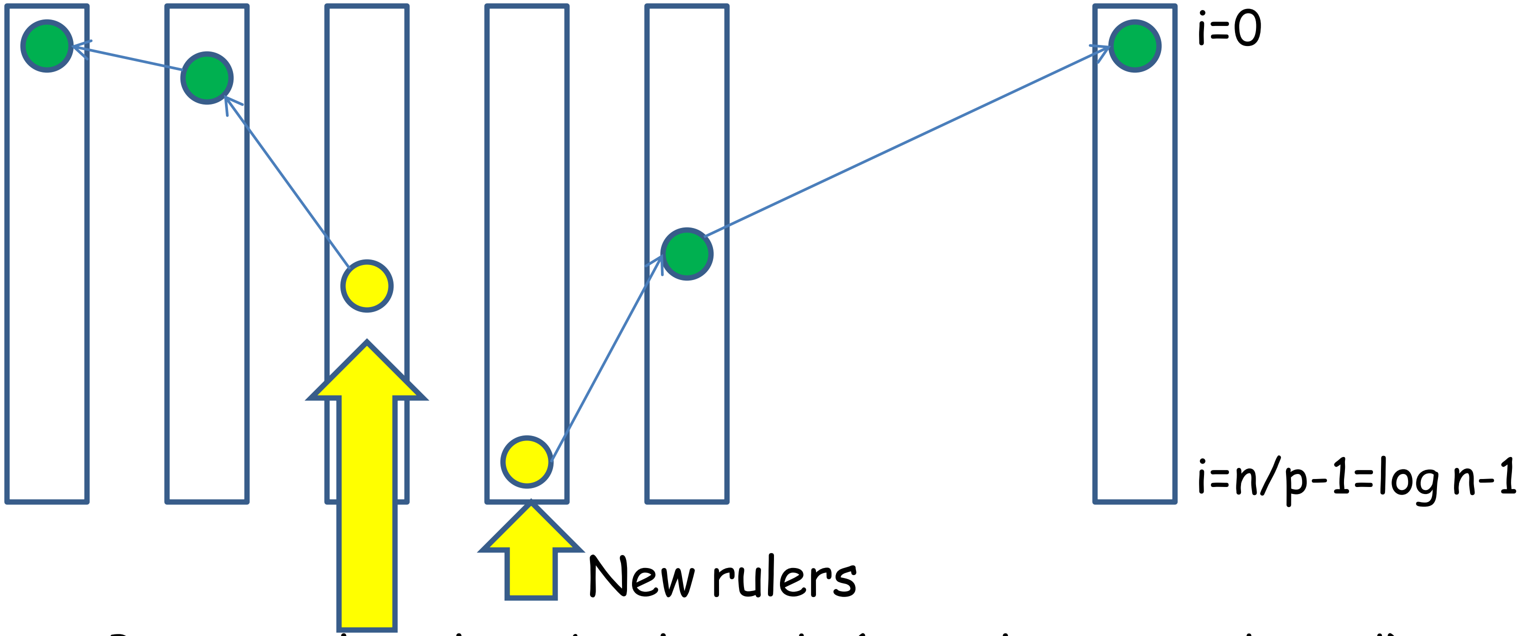
Remedy 2: Estimate the remaining work of each ruler by its position in the queue, and break chains into smaller chains of decreasing work

Determine local maximum and minimum elements wrt. position in queue. Break chains here.



Remedy 2: Estimate the remaining work of each ruler by its position in the queue, and break chains into smaller chains of decreasing work

Determine local maximum and minimum elements wrt. position in queue. Break chains here. Role and direction can be determined in O(1) time and O(p) work



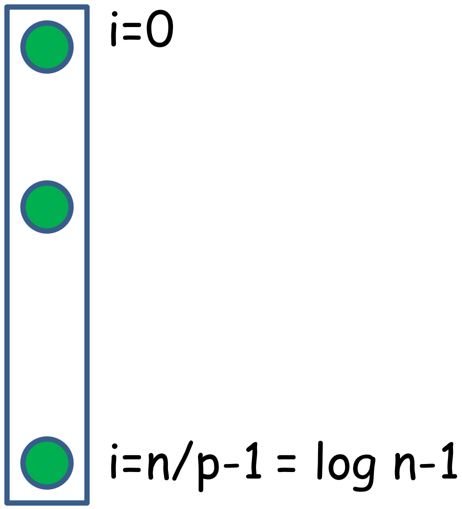
Process this chain backwards (switch succ and pred)

Set q = 1/log log n, and let p = n/log n (number of processors)

Assign weight wi = (1-q)^i to the i'th queue element (i=0,1,…,n/p-1).

Total weight per queue is at most ∑0≤j<i:(1-q)^j < 1/q (recall: geometric series with q<1). Thus, total weight of all elements at most p/q = (n/log n)/q (for p = n/log n).

The smallest weight of an element is (1-q)^(n/p-1) = (1-q)^(log n -1)



Modified accounting scheme:

- Removal of isolated element: decrease by weight of element
- Element becomes subject: decrease by ½ weight of element
- Element is removed by ruler: decrease by ½ weight
- Element becomes ruler: no change

Main Claim:

Each phase reduces the total weight of the queues by at least a factor of (1-q/4)

Theorem:
The AM algorithm solves the list ranking problem in O(log n) time steps and O(n) work on an EREW PRAM

Proof: Correctness is clear from basic version of the algorithm.

Assume the claim holds, then after $k \geq 5\log n$ phases:

$$(n/\log n)/q\ (1-q/4)^k \quad < (n/\log n)(1-q)^{(\log n)}$$
$$< (n/\log n)(1-q)^{(\log n - 1)}$$

which follows from $1/q\ (1-q/4)^{(c \log n)} < (1-q)^{(\log n)}$ for $c \geq 5$
which follows using the fact that $(1+q/x)^x \to e^x$

Theorem:
The AM algorithm solves the list ranking problem in $O(\log n)$ time steps and $O(n)$ work on an EREW PRAM

Proof: Correctness is clear from basic version of the algorithm.

Assume the claim holds, then after $k \geq 5\log n$ phases:

$$\underbrace{(n/\log n)/q}\,(1-q/4)^k \quad < (n/\log n)(1-q)^{(\log n)}$$
$$< (n/\log n)\underbrace{(1-q)^{(\log n-1)}}$$

which follows from $1/q\,(1-q/4)^{(c \log n)} < (1-q)^{(\log n)}$ for $c \geq 5$
which follows using the fact that $(1+q/x)^x > e^x$

Initial total weight

Smallest weight

This implies that at most $n/\log n$ elements remain after k phases

Proof of main claim:

For the proof the weights are distributed to the processors as follows:

- Ruler: weight of remaining elements in queue, weight of queue head and all subjects
- Subjects: weight of remaining elements in queue

Observation 1:

First note that for a queue with current element at position i

$$W = \sum i\le j<\log n: (i-q)^j < \sum i\le j: (1-q)^j = 1/q - (1-(1-q)^i)/q$$
$$= ((1-q)^i)/q$$

Case 1: Processor is idle (because its queue is empty), no change in queue weight, but since the weight of an empty queue is 0, claim holds

Case 2: Isolated element removed from position i queue of some processor. Weight of queue before phase is $W=\sum i\leq j<\log n: (1-q)^j$, and after phase $W' = W-(1-q)^i$. This gives:

$W'/W = (W-(1-q)^i)/W = 1 - ((1-q)^i)/W < 1 - q$ (Observation 1)

Thus, the weight of the queue has changed by a factor $(1-q) < (1-q/4)$

Case 3: Element becomes ruler or subject

Let the chain of the subject have queue positions $i0, i1, i2, \ldots, ik$ with $i0$ being the position of the ruler. By the algorithm $i0 \geq ij$ (remedy 2: rulers have smallest amount of work), $j=1,2,\ldots,k$. Recall that $k \leq \log \log n$ (remedy 1, log log n ruling set)

The weight of the j'th subject's queue after the phase is $wj' = wj-(1-q)^{ij}$ where wj is the weight before the phase.

The weight of the ruler is
$w0' = w0+\frac{1}{2}\sum_{1 \leq j < k} (1-q)^{ij}$
since by the accounting scheme the weight of the ruler is the weight of elements in its own queue and the weight of the ruled subjects, and their weights have been decreased by $\frac{1}{2}$.

Now (Observation 1)

$$wj'+\tfrac{1}{2}(1-q)^{ij} = wj -\tfrac{1}{2}(1-q)^{ij}$$
$$< (1-q)^{ij}/q - \tfrac{1}{2}(1-q)^{ij} = (1-q/2)(1-q)^{ij}/q = (1-q/2)wj$$

The total weight of the queues of the chain after the phase is therefore

$$W' = w0+\textstyle\sum 1\leq j<k\text{: } wj-\tfrac{1}{2}(1-q)^{ij} < w0+\sum 1\leq j<k\text{: } (1-q/2)wj$$
$$= (1-q/2)W-(1-(1-q/2))w0$$
$$= w0(q/2) + (1-q/2)W$$

where W is the total weight before the phase. Since there is at least one subject, and since $w0\leq wj$ by the construction of the chains

$$w0(q/2) + (1-q/2)W \leq q/4\ W + (1-q/2)W = (1-q/4)W$$

Therefore it holds that $W'/W \leq (1-q/4)$

Case 4: Ruler removes next subject

In this case it is shown that after k phases, $1 \leq k \leq \log \log n$, it holds that $W'/W \leq (1-q/4)^k$, where W' and W are the weights of the rulers queue after and before the k phases

$W = w0+ \frac{1}{2} \sum 1 \leq j < k: (1-q)^{ij}$ and $W' = w0$, where w0 is the sum of the weights of the elements in the rulers queue, and the ruled elements are assigned half their original weight by the accounting scheme

Since $i0 \geq ij$ this gives
$W'/W = w0/W \leq 1/(1+1/(2w0)\ k(1-q)^i) \leq 1/(1+kq/2)$
since by Observation 1 $w0 < ((1-q)^i)/q$

It now only remains to verify that $1/(1-kq/2) \leq (1-q/4)^k$

This follows by multiplying

$(1-kq/2)(1.-q/4)^k \geq (1-kq/2)(1-kq/4) = 1+kq/4-((kq)^2)/8 \geq 1$

Case 1-4 completes the proof of the claim, and establishes the O(log n) bound on the number of phases for the AM algorithm

Key result of PRAM parallel computing:
List ranking can be done in O(log n) time steps and O(n) work with no concurrent reading or writing

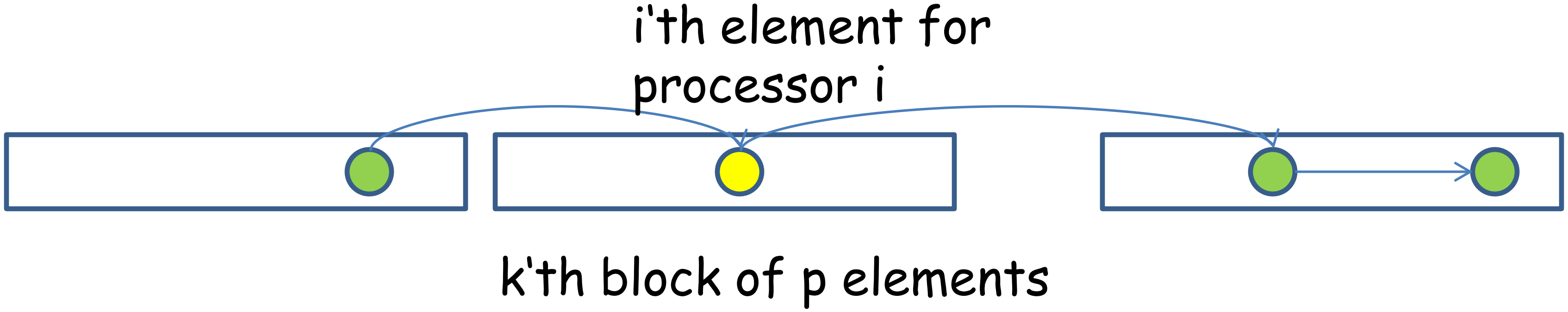
Is the AM algorithm a practical list ranking algorithm?

List ranking without coloring: Kruskal-Rudolph-Snir (KRS)

List stored compactly in array, L[i] a list element with succ[i] pointer. Algorithm uses p processors to shrink list by a constant factor, iterates until O(p) elements remain, then use Wyllie

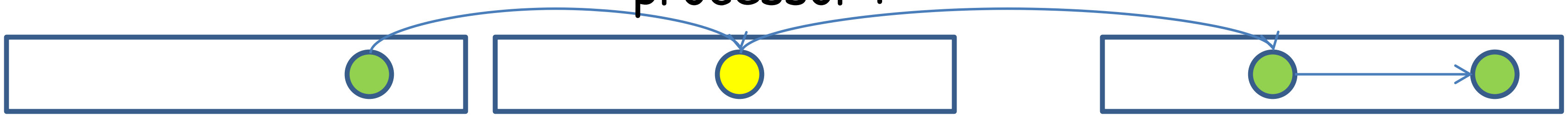
1. If p=1 rank L sequentially, else
2. Divide L into n/p blocks of p list elements, process the blocks one after another
3. Block k, k=0,…,n/p-1: assign processor i to i'th element of block; if succ[i] is in another block, splice out succ[i]
4. Recursively apply the algorithm to the n/p blocks in parallel, assigning max(1,p^2/n) processors to each

[C. P. Kruskal, L. Rudolph, M. Snir: The power of parallel prefix. IEEE Tr. Computers, C-34(10): 965-968, 1985]



Block k:
If succ[i] in different block and element i has not been marked, splice out succ[i], mark i

Observation:
After k'th block has been processed, all unmarked elements i have succ[i] inside k'th block. Thus, the n/p blocks can be treated independently (in parallel); use as many processors as possible per block



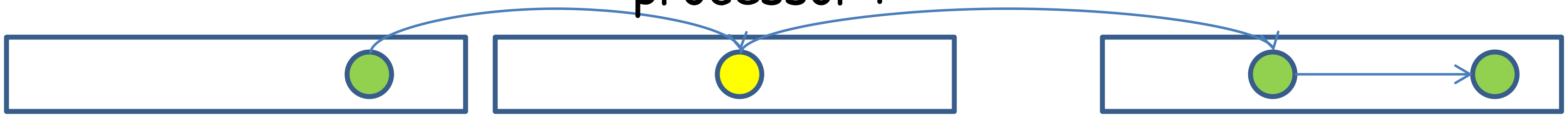
AllOut(p,n,…):

```
if (p==1) { // one processor only
  for (i=0; i<n; i++)
    if (!Del[i]&&!M[i]) SpliceOut(i)
} else { // p>1 processors
  for (k=0; k<n; k+=p)
    par (k≤i<k+p)
      if (!Del[i]&&!M[i]&&!(k≤succ[i]<k+p))
        SpliceOut(i);
   par (0≤k<n/p) AllOut(p^2/n,p,block k);
}
```

SpliceOut(i):

```
w[i] = w[i]+w[succ[i]];
Del[succ[i]] = true;
succ[i] = succ[succ[i]];
M[i] = true;
```

i'th element for processor i



k'th block of p elements

Paper additionally checks successor?

SpliceOut(i):

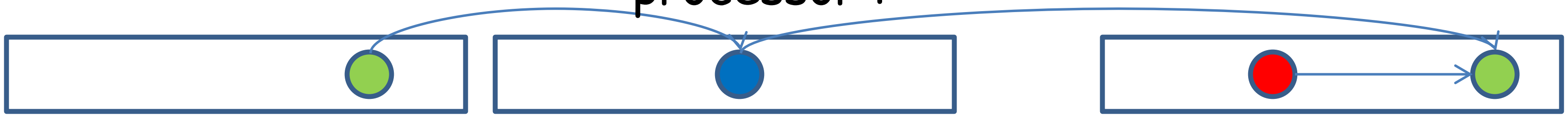
```
w[i] = w[i]+w[succ[i]];
Del[succ[i]] = true;
succ[i] = succ[succ[i]];
M[i] = true;
```

AllOut(p,n,…):

```
if (p==1) { // one processor only
  for (i=0; i<n; i++)
    if (!Del[i]&&!M[i]&&!M[succ[i]])
      SpliceOut(i)
} else { // p>1 processors
  for (k=0; k<n; k+=p)
    par (k≤i<k+p)
      if (!Del[i]&&!M[i]&&!(k≤succ[i]<k+p))
        SpliceOut(i);
   par (0≤k<n/p) AllOut(p^2/n,p,block k);
}
```

succ[i] in another block, splice out succ[i], mark i

i'th element for processor i



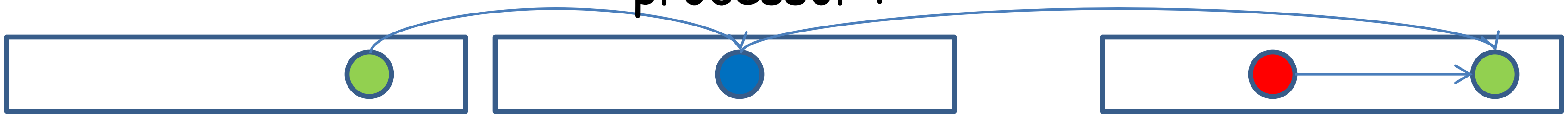
k'th block of p elements

SpliceOut(i):

```
w[i] = w[i]+w[succ[i]];
Del[succ[i]] = true;
succ[i] = succ[succ[i]];
M[i] = true;
```

AllOut(p,n,…):

```
if (p==1) { // one processor only
  for (i=0; i<n; i++)
    if (!Del[i]&&!M[i]) SpliceOut(i)
} else { // p>1 processors
  for (k=0; k<n; k+=p)
    par (k≤i<k+p)
      if (!Del[i]&&!M[i]&&!(k≤succ[i]<k+p))
        SpliceOut(i);
   par (0≤k<n/p) AllOut(p^2/n,p,block k);
}
```



i'th element for processor i

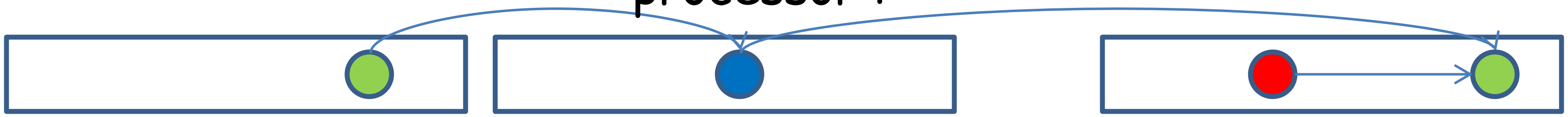
k'th block of p elements

AllOut(p,n,…):

```
if (p==1) { // one processor only
  for (i=0; i<n; i++)
    if (!Del[i]&&!M[i]) SpliceOut(i)
} else { // p>1 processors
  for (k=0; k<n; k+=p)
    par (k≤i<k+p)
      if (!Del[i]&&!M[i]&&!(k≤succ[i]<k+p))
        SpliceOut(i);
  par (0≤k<n/p) AllOut(p^2/n,p,block k);
}
```

Recurse, n/p blocks, so p/(n/p) = max(1,p^2/n) processors can be allocated to each

i'th element for processor i



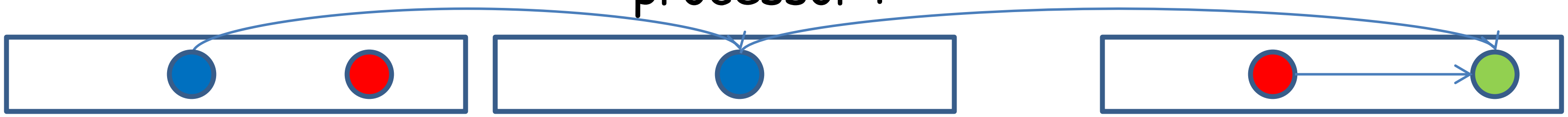
k'th block of p elements

AllOut(p,n,…):

```
if (p==1) { // one processor only
  for (i=0; i<n; i++)
    if (!Del[i]&&!M[i]) SpliceOut(i)
} else { // p>1 processors
  for (k=0; k<n; k+=p)
    par (k≤i<k+p)
      if (!Del[i]&&!M[i]&&!(k≤succ[i]<k+p))
        SpliceOut(i);
   par (0≤k<n/p) AllOut(p^2/n,p,block k);
}
```

Observation: No conflicts, so EREW PRAM suffices. The procedure reduces the number of elements by a factor of 2/3: if an element has not been spliced out, both neighbors have

i'th element for processor i



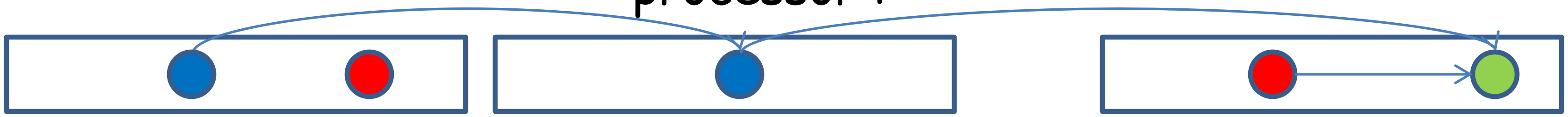
k'th block of p elements

AllOut(p,n,…):

```
if (p==1) { // one processor only
  for (i=0; i<n; i++)
    if (!Del[i]&&!M[i]) SpliceOut(i)
} else { // p>1 processors
  for (k=0; k<n; k+=p)
    par (k≤i<k+p)
      if (!Del[i]&&!M[i]&&!(k≤succ[i]<k+p))
        SpliceOut(i);
   par (0≤k<n/p) AllOut(p^2/n,p,block k);
}
```

Observation: No conflicts, so EREW PRAM suffices. The procedure reduces the number of elements by a factor of 2/3: if an element has not been spliced out, both neighbors have

i'th element for processor i



k'th block of p elements

AllOut(p,n,…):

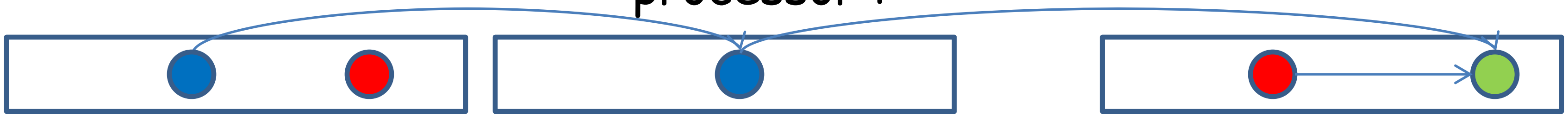
```
if (p==1) { // one processor only
  for (i=0; i<n; i++)
    if (!Del[i]&&!M[i]) SpliceOut(i)
} else { // p>1 processors
  for (k=0; k<n; k+=p)
    par (k≤i<k+p)
      if (!Del[i]&&!M[i]&&!(k≤succ[i]<k+p))
        SpliceOut(i);
   par (0≤k<n/p) AllOut(p^2/n,p,block k);
}
```

Claim: Time with p processors for AllOut is

$O(n/p * \log n/\log(2n/p))$

Proof: exercise…

i'th element for processor i

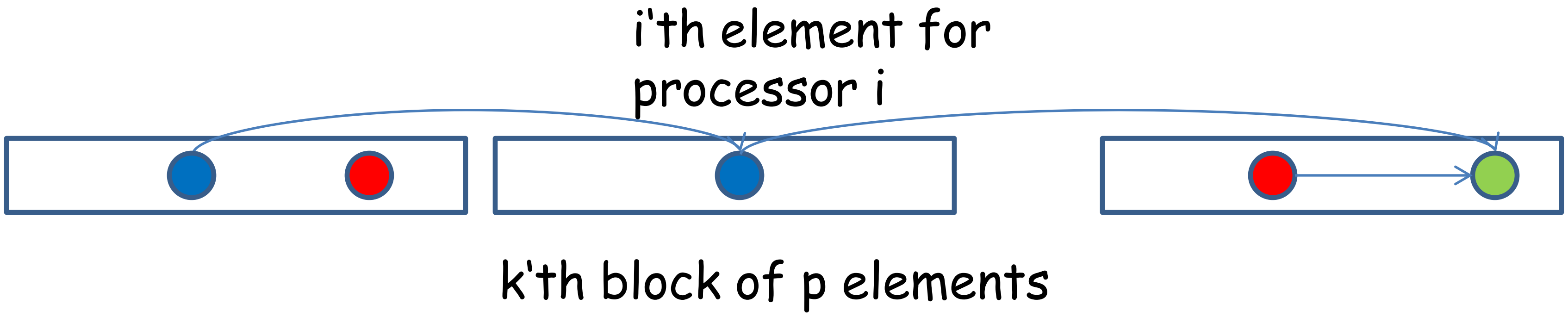


k'th block of p elements

AllOut(p,n,…):

```
if (p==1) { // one processor only
  for (i=0; i<n; i++)
    if (!Del[i]&&!M[i]) SpliceOut(i)
} else { // p>1 processors
  for (k=0; k<n; k+=p)
    par (k≤i<k+p)
      if (!Del[i]&&!M[i]&&!(k≤succ[i]<k+p))
        SpliceOut(i);
   par (0≤k<n/p) AllOut(p^2/n,p,block k);
}
```

List ranking done by repeatedly applying AllOut followed by a compaction of the non-deleted elements until O(p) elements remain; then Wyllie



AllOut(p,n,…):

```
if (p==1) { // one processor only
  for (i=0; i<n; i++)
    if (!Del[i]&&!M[i]) SpliceOut(i)
} else { // p>1 processors
  for (k=0; k<n; k+=p)
    par (k≤i<k+p)
      if (!Del[i]&&!M[i]&&!(k≤succ[i]<k+p))
        SpliceOut(i);
   par (0≤k<n/p) AllOut(p^2/n,p,block k);
}
```

Theorem: The KRS algorithm solves the list ranking problem on a p processor EREW PRAM in O(n/p * log n/log(2n/p)) time steps. This is cost optimal for p<n^(1-ε) for any constant ε>0

Highlights in history of deterministic EREW list ranking

Work-optimal

- Steven Fortune, James Wyllie: Parallelism in Random Access Machines. STOC 1978: 114-118
- Clyde P. Kruskal, Larry Rudolph, Marc Snir: The Power of Parallel Prefix. IEEE Trans. Computers 34(10): 965-968 (1985)
- Yijie Han, Robert A. Wagner: Parallel Algorithms for Bucket Sorting and the Data Dependent Prefix Problem. ICPP 1986: 924-930
- Richard Cole, Uzi Vishkin: Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. Information and Control 70(1): 32-53 (1986)

Work-optimal, logarithmic

- Richard Cole, Uzi Vishkin: Approximate Parallel Scheduling. Part I: The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time. SIAM J. Comput. 17(1): 128-142 (1988)
- Richard J. Anderson, Gary L. Miller: Deterministic Parallel List Ranking. Algorithmica 6(6): 859-868 (1991)

Journal version, actual results earlier

Computations on trees: Euler tour technique

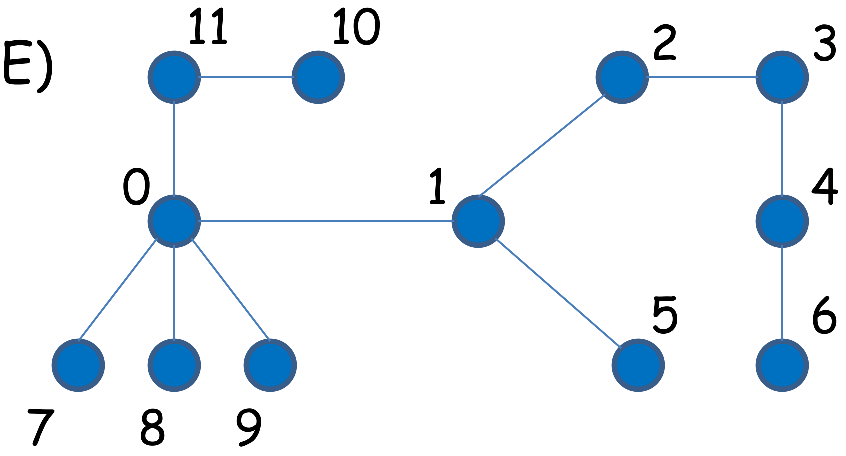
A tree T=(V,E) is an undirected graph that is connected and contains no cycles. The following three conditions are equivalent

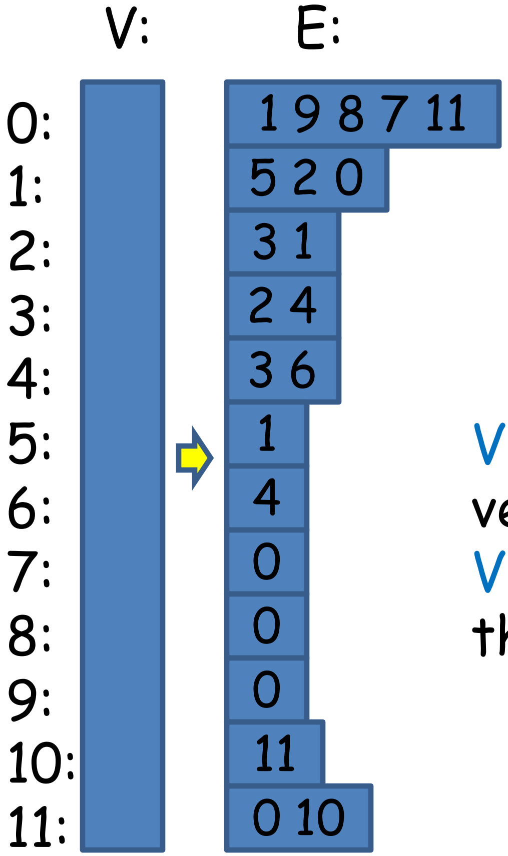
1. T is a tree
2. T is minimal connected (removing an edge will disconnect T)
3. T is maximal acyclic (adding an edge will create a cycle)

A tree with |V|=n nodes has|E|= n-1 edges. An oriented tree can be represented by a parent pointer for each node.

For now: undirected tree represented as a set of adjacency lists, stored in arrays

T=(V,E)





	V:	E:
0:		1 9 8 7 11
1:		5 2 0
2:		3 1
3:		2 4
4:		3 6
5:		1
6:		4
7:		0
8:		0
9:		0
10:		11
11:		0 10

V[i].degree: number of edges adjacent to vertex i

V[i].edges: array of i's edges, v=V[i][j] is the j'th edge in some order

Given undirected tree T=(V,E). The adjacency list representation represents T as a directed graph, each edge (u,v) adjacent to u is also an edge (v,u) adjacent to v. A directed edge <u,v> from u to v is called an arc

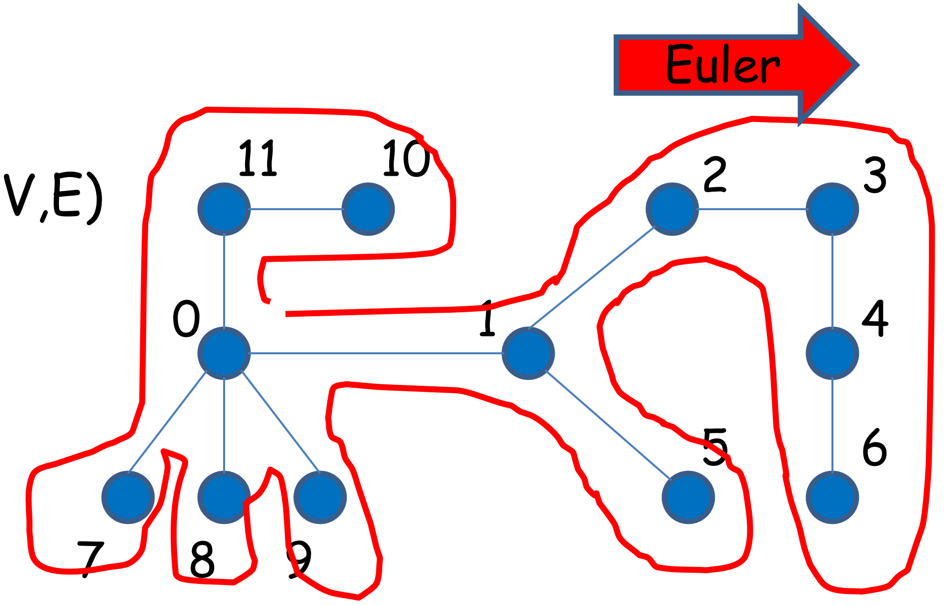
Since each node u has the same number of incoming and outgoing arcs, there exists a circuit that traverses all arcs exactly once.

Such a circuit is called an Euler tour

An Euler tour can be represented by a successor function that maps each arc e=<u,v> into the next arc on the tour s(e)

	V:	
0:		1 9 8 7 11
1:		2 5 0
2:		3 1
3:		2 4
4:		3 6
5:		1
6:		4
7:		0
8:		0
9:		0
10:		11
11:		0 10

T=(V,E)



<0,1>, <1,2>, <2,3>, <3,4>, <4,6>,
<6,4>, <4,3>, <3,2> <2,1>, <1,5>, <5,1>,
<1,0>, <0,7>, <7,0>, <0,8>, <8,0>,
<0,9>, <9,0>, <0,11>, <11,10>, <10,11>,
<11,0>

Proposition: Given a T=(V,E) in adjacency list representation, an Euler tour can be computed in O(1) time steps and O(n) work on an EREW PRAM

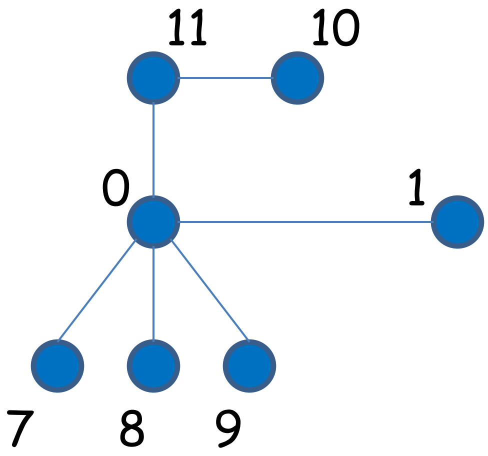
Proof: The adjacency list representation determines an ordering of the arcs of each node v, say
V[v].edges = <u0,u1,…,u(deg(v)-1)>.

Define s(<ui,v>) = <v,u((i+1) mod deg(v)>

Example:

V[0].edges=<1,9,8,7,11>

s(<11,0>) = <0,1>
s(<1,0>) = <0,9>
s(<7,0>) = <0,11>
s(<8,0>) = <0,7>
s(<9,0>) = <0,8>



Proposition: Given a T=(V,E) in adjacency list representation, an Euler tour can be computed in O(1) time steps and O(n) work on an EREW PRAM

Proof: The adjacency list representation determines an ordering of the arcs of each node v, say
V[v].edges = <u0,u1,…,u(deg(v)-1)>.

Define s(<ui,v>) = <v,u((i+1) mod deg(v)>

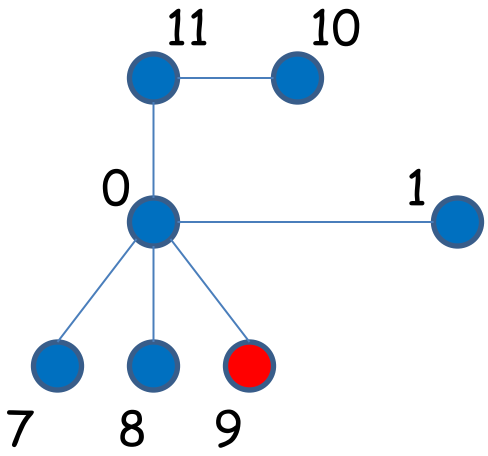
Claim: s determines an Euler tour. Proof by induction on the number of nodes n of T.

Induction base |V|=2 is clear, s(<u,v>) = <v,u>, since
V[u],edges=<v> and V[v].edges=<u>

Proposition: Given a T=(V,E) in adjacency list representation, an Euler tour can be computed in O(1) time steps and O(n) work on an EREW PRAM

Proof: The adjacency list representation determines an ordering of the arcs of each node v, say
V[v].edges = <u0,u1,…,u(deg(v)-1)>.

Define s(<ui,v>) = <v,u((i+1) mod deg(v)>



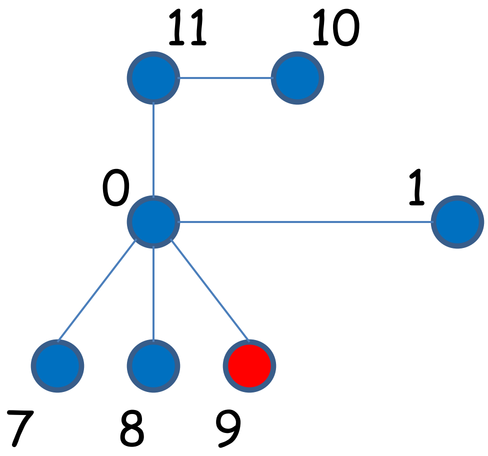
Induction step, |V|>2. Consider a leaf v adjacent to some vertex u, and let V[u].edges = <…,v',v,v'',…> (v' and v'' could be same node). By construction s(<v',u>) = <u,v> and s(<v,u>) = <u,v''>. Now remove v; then V[u].edges = <…,v',v'',…>. Successor function is unchanged, except for s(<v',u>) = <u,v''>

Proposition: Given a T=(V,E) in adjacency list representation, an Euler tour can be computed in O(1) time steps and O(n) work on an EREW PRAM

Proof: The adjacency list representation determines an ordering of the arcs of each node v, say
V[v].edges = <u0,u1,…,u(deg(v)-1)>.

Define s(<ui,v>) = <v,u((i+1) mod deg(v)>

Now remove v; then V[u].edges = <…,v',v'',…>.
Successor function is unchanged, except for
s(<v',u>) = <u,v''>
By induction hypothesis, s defines an Euler tour in the T without v. Replacing s(<v',u>) = <u,v''> by s(<v',u>) = <u,v> and adding s(<v,u>) = <u,v'') result in an Euler tour for T



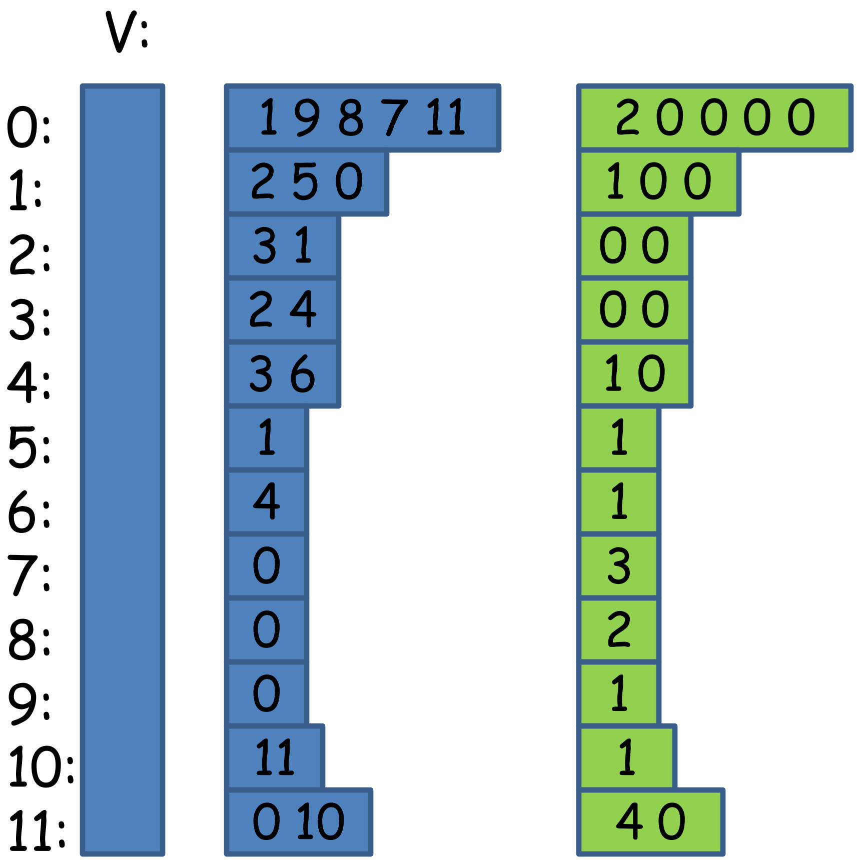
Proposition: Given a T=(V,E) in adjacency list representation, an Euler tour can be computed in O(1) time steps and O(n) work on an EREW PRAM

Proof: The adjacency list representation determines an ordering of the arcs of each node v, say
V[v].edges = <u0,u1,…,u(deg(v)-1)>.

Define s(<ui,v>) = <v,u((i+1) mod deg(v)>

To compute the successor function, it is needed for each arc <ui,v> into node v to know the next node after ui that is adjacent to v.

The adjacency list representation must be augmented with the index of u in the adjacency list of v for each node v adjacent to u:

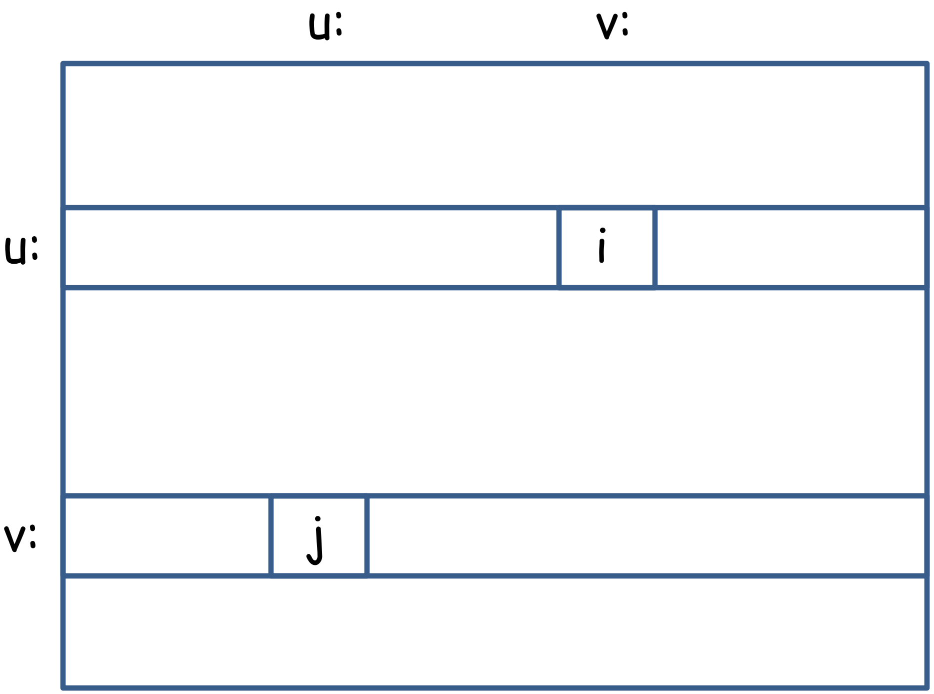


V:		
0:	1 9 8 7 11	2 0 0 0 0
1:	2 5 0	1 0 0
2:	3 1	0 0
3:	2 4	0 0
4:	3 6	1 0
5:	1	1
6:	4	1
7:	0	3
8:	0	2
9:	0	1
10:	11	1
11:	0 10	4 0

This additional information („backpointers“) can be computed in parallel in O(1) time and O(n) operations

O(n^2) space is required, but does not have to be initialized

```
par (u in V) {
  par (vi in V[u].edges) {
    indices[u][vi] = i;
  }
}
par (u in V) {
  par (vi in V[u].edges) {
    V[u].back[i] =
      indices[vi][u] ;
  }
}
```



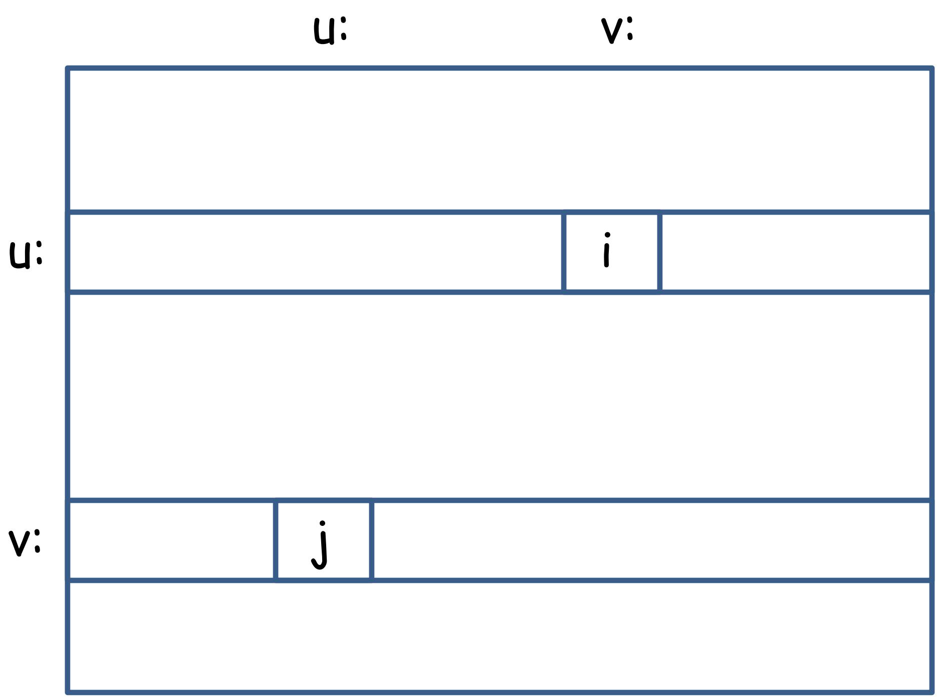
|V|x|V| indices matrix

V[u].edges = <…,vi, …>
V[v].edges = <…,uj, …>

Note: some prefix-sums computations necessary to schedule the processors for the nested **par** constructs. Do as exercise!

Euler tour computation: determining successor for each arc

```
par (u in V) {
  par (vi in V[u].edges) {
    indices[u][vi] = i;
  }
}
par (u in V) {
  par (vi in V[u].edges) {
    V[u].back[i] =
      indices[vi][u] ;
  }
}
```



|V|x|V| indices matrix

```
par (u in V) {
  par (vi in V[u].edges) {
    s(<u,vi>) = <vi,V[vi].edges[(V[u].back[i]+1) mod V[vi].size);
  }
}
```

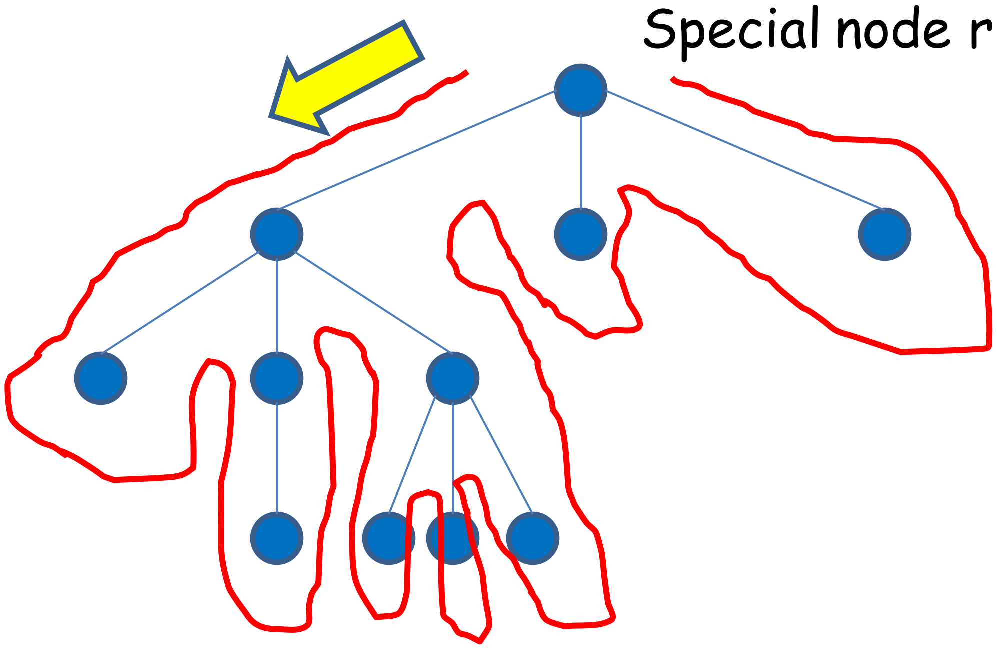
Using the Euler tour: rooting and traversing a tree

T=(V,E) a tree, special vertex r, task: to root T at r and compute for each v in V its parent p(v) in the rooted version of T

Compute an Euler tour of T, break at the arc <v,r> into r (no successor); this is a list of all arcs of T corresponding to a depth-first traversal of T: e0=<r,v0> visits v0, the successor edges visit the subtree rooted at v0 before returning to r (recall Euler tour construction, V[v0].edges = <…,r,v1,v2, …>), the next arc <r,v1> out of r is followed by a traversal of the subtree rooted at v1, etc.

Observation: for each node v≠r, the arc <p(v),v> appears before the arc <v,p(v)> on the Euler tour

T=(V,E)



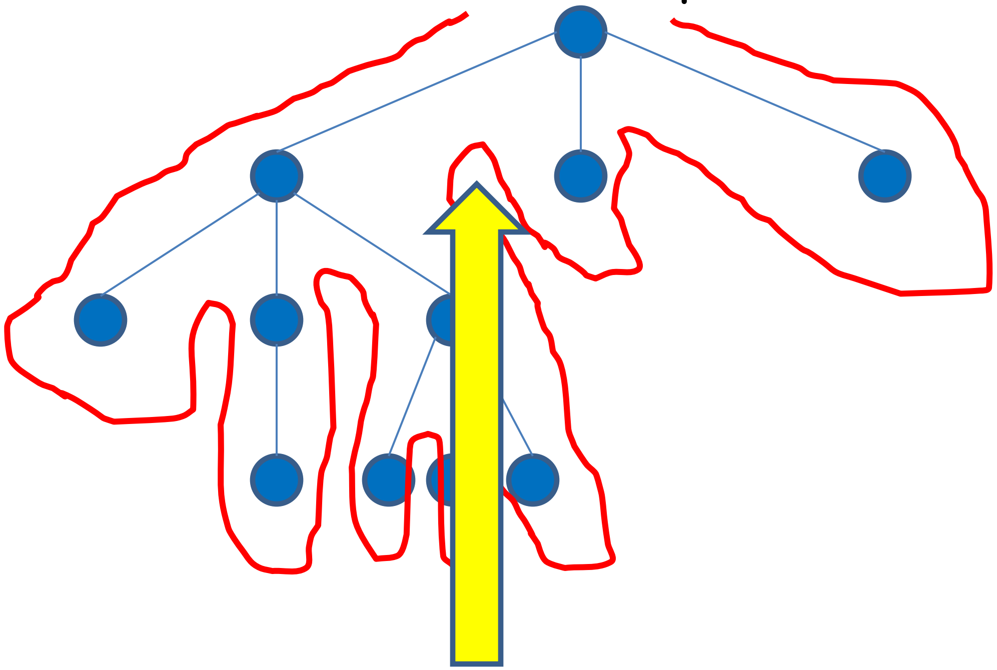
Rooting a tree T:

Input: A tree T represented by adjancency lists stored in arrays, a special node r
Output: A rooted version of T represented by parent pointers p(v) for v≠r

1. Set s(<v,r>) = †, build a list from the Euler tour
2. Assign weight to each arc, w[<u,v>] = 1
3. Perform list-ranking/prefix sums computation
4. For each arc, if w[<u,v>] < w[<v,u>] assign p(v) = u

Except for the list ranking step, all steps take O(1) time and O(n) operations; list ranking O(log n) steps and O(n) operations. All steps can be done on an EREW PRAM

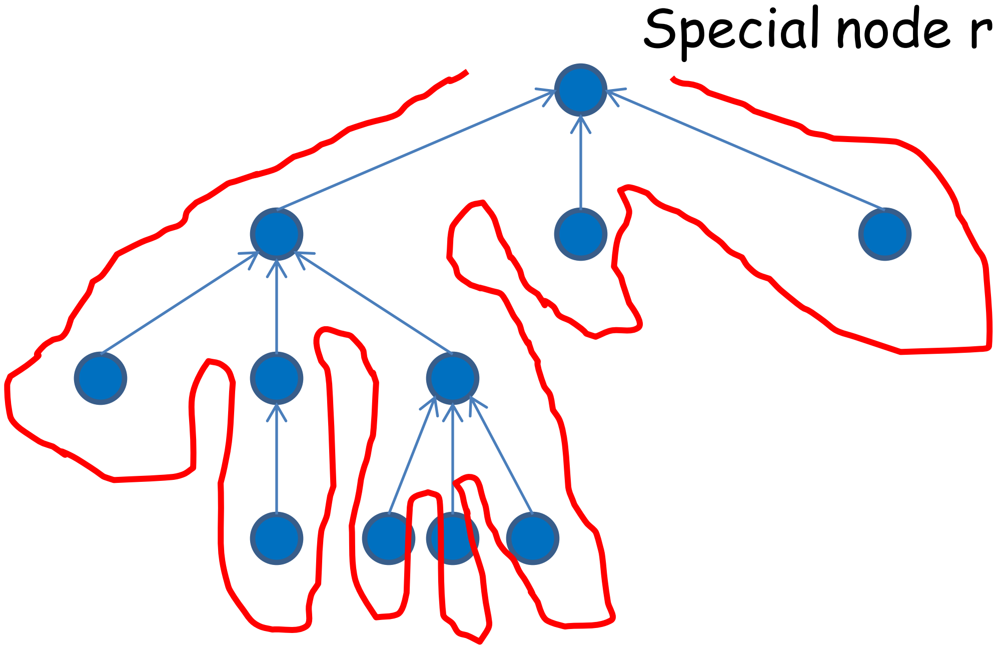
T=(V,E)



Special node r

w[<r,v>] < w[<v,r>]

T=(V,E)



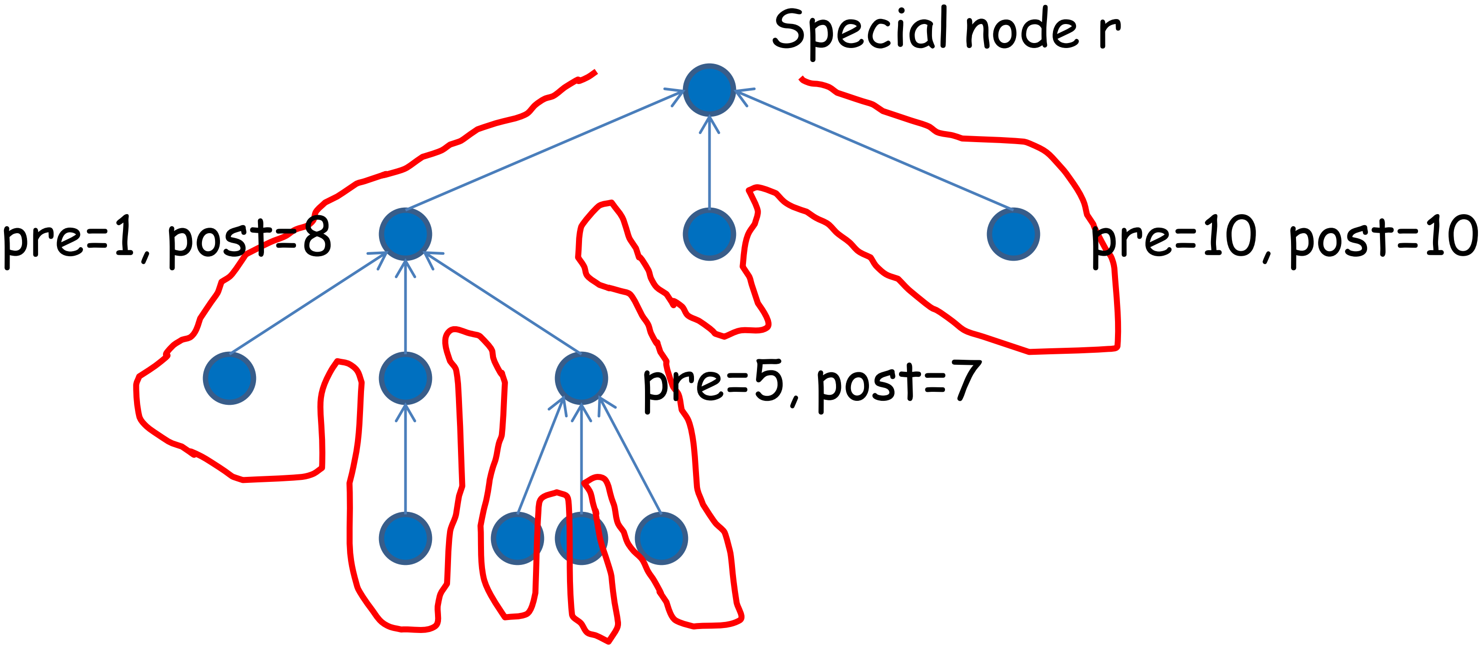
Using the Euler tour: rooting and traversing a tree

T=(V,E) a tree, special vertex r, task: to root T at r and compute for each v in V its parent p(v) in the rooted version of T

Compute an Euler tour of T, break at the arc <v,r> into r (no successor); this is a list of all arcs of T corresponding to a depth-first traversal of T

Observation: The first time a node is visited determines its preorder traversal number, the last time a vertex is visited its postorder number; and the difference between the first and the last time a node is visited the number of descendants

T=(V,E)



Computing pre- and postorder numbers of T, number of descendants:

1. Preorder: w-pre[<p(v),v>] = 1, w-pre[<v,p(v)>] = 0
2. Postorder: w-post[<v,p(v)>] = 1, w-post[<p(v),v>] = 0

3. List rank/parallel prefix-sums on w-pre, w-post

4. For each v:
   preorder(v) = w-pre[<p(v),v>]
   postorder(v) = w-post[<v,p(v)>]
   descendants(v) = w-pre[<v,p(v)>] - w-pre[<p(v),v>]

Using the Euler tour: rooting and traversing a tree

T=(V,E) a tree, special vertex r, task: to root T at r and compute for each v in V its parent p(v) in the rooted version of T

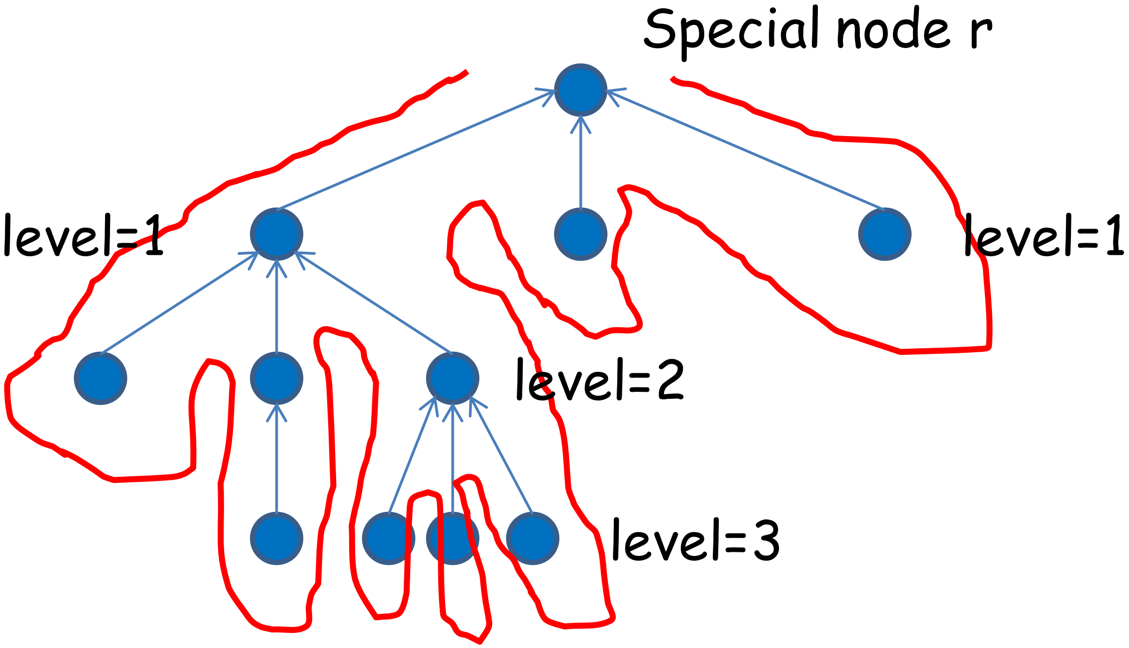
Compute an Euler tour of T, break at the arc <v,r> into r (no successor); this is a list of all arcs of T corresponding to a depth-first traversal of T

Definition: The depth (level) of node v is its distance from the root.
Observation: The level of v is the difference between the number of down- and up-pointing edges before v is reached by the Euler tour

Computing level number for a tree T:

1. Level: w[<p(v),v>] = 1, w[<v,p(v)>] = -1
2. List rank/parallel prefix-sums
3. For each v: level(v) = w[<p(v),v>]

T=(V,E)



Theorem:

Given a tree T=(V,E) represented as adjacency lists, and a special vertex r, the tree can be rooted at r as represented by a parent pointer p(v) for each v in V in O(log n) time steps and O(n) work on an EREW PRAM.

Theorem:

Given a tree T=(V,E) represented as adjacency lists, rooted at vertex r, the following can be obtained in O(log n) time steps and O(n) work for all nodes of T on an EREW PRAM:

1. Pre- and postoder traversal numbers
2. Depth in tree (level)
3. Number of descendants

Range minimum problem and the LCA problem

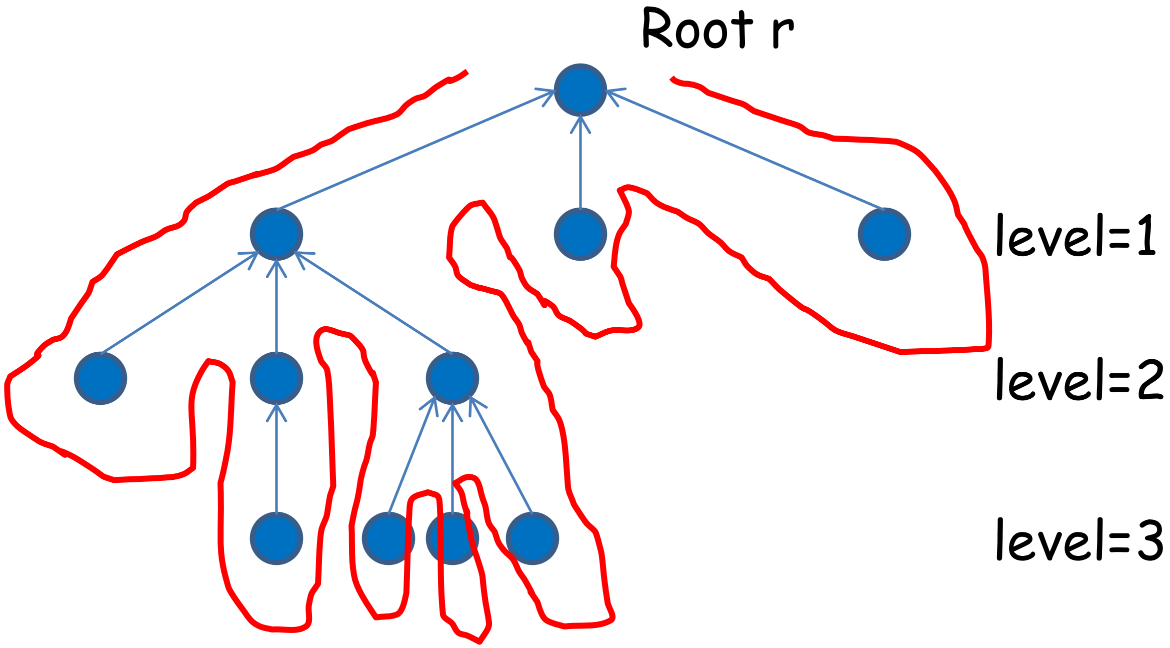
Task:
For any two nodes u and v in a rooted tree T, determine the lowest common ancestor (LCA) of u and v in T: LCA(u,v) is first node on the path from u to r that is also on the path from v to r

Strategy:
Preprocess T such that queries LCA(u,v) can be answered fast

Outcome: LCA(u,v) queries can be answered in O(1) time, on CREW PRAM batches of queries can be done in parallel

T=(V,E)

|V|=n, number of arcs m=2(n-1)

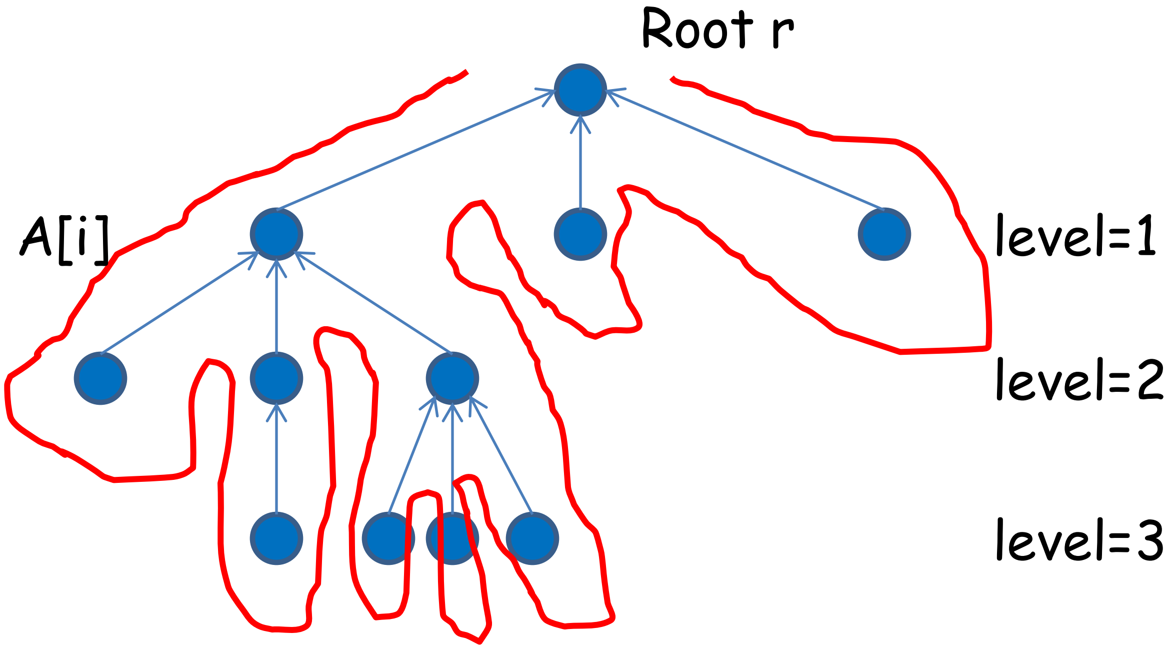


Using the Euler tour techniques construct arrays:

- A[0…m+1]: nodes v of arc <u,v> in the order they appear on the tour, in addition A[0]=r
- L[0…m+1]: level(A[i]) for i=0,…,m
- l[v],r[v]: first (leftmost) and last (rightmost) apperance of v in A

T=(V,E)

|V|=n, number of arcs m=2(n-1)

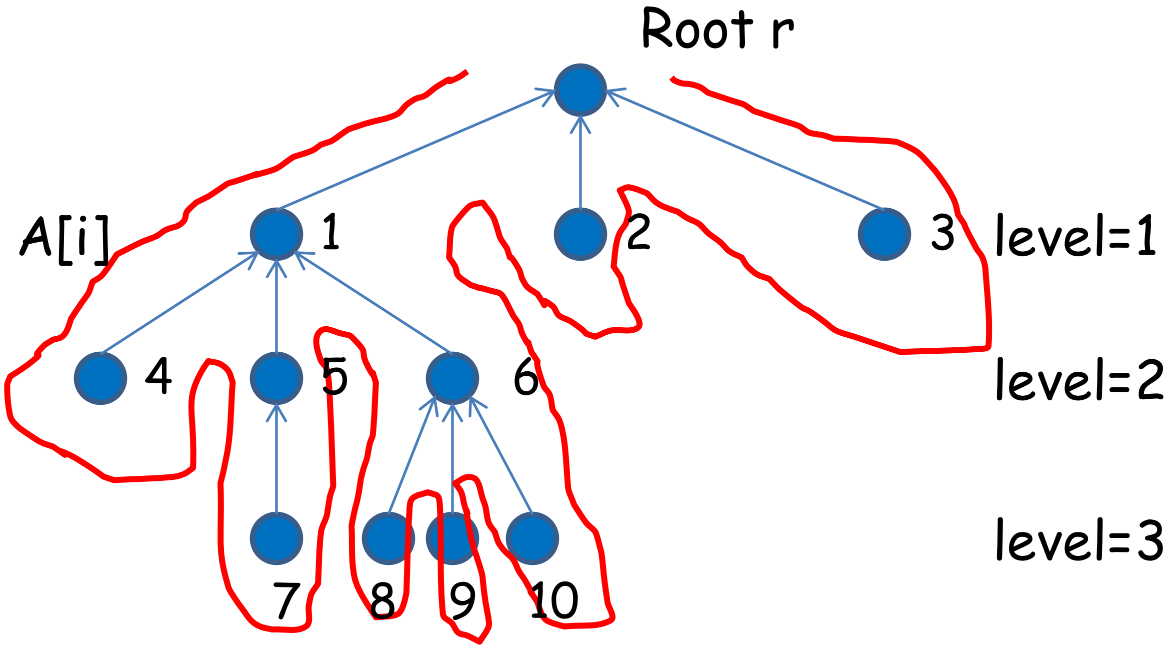


Observation:

v=A[i] is first appearance of v in A iff L[A[i]]==L[A[i-1]]+1

v=A[i] is last appearance of v in A iff L[A[i]]==L[A[i+1]]+1

T=(V,E) |V|=n, number of arcs m=2(n-1)

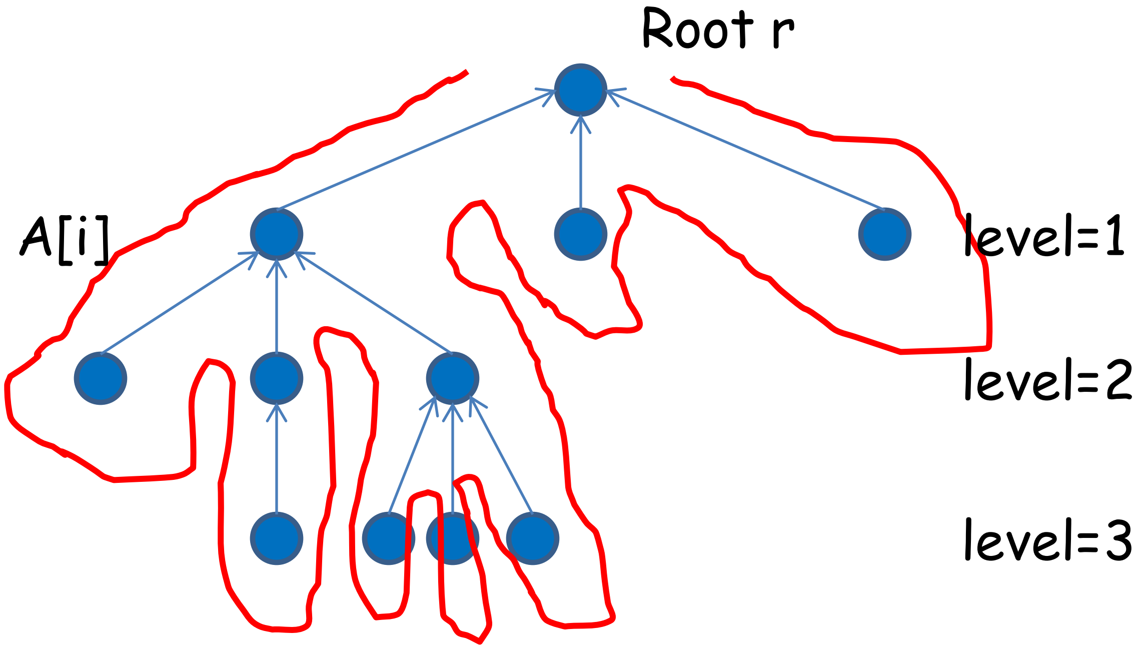


Example:
A[0,…] = [0,1,4,1,5,7,5,1,6,8,6,9,6,10,6,1,0,…]

first last

T=(V,E)

|V|=n, number of arcs m=2(n-1)



```
par (1≤i<n-1) {
   if (L[A[i]]==L[A[i-1]]+1) l[A[i]] = i;
   if (L[A[i]]==L[A[i+1]]+1) r[A[i]] = i;
}
l[0] = 0; r[0] = n-1;
```

Thus: l and r arrays can be computed in O(n) operations and O(1) time

Lemma: Let u,v be distinct nodes of tree T=(V,E); then

1. u is an ancestor of v iff $l(u)<l(v)\leq r(v)<r(u)$
2. u and v are not related (neither ancestor of the other) iff $r(u)<l(v)$ or $r(v)<l(u)$
3. If $r(u)<l(v)$ then LCA(u,v) is a node with the minimum level in $A[r(u),\ldots,l(v)]$

Proof (1):
If u is ancestor of v then the first appearance of v on the tour (=DFS traversal) is after the first appearance of u; also the subtree of v is completely visited before the tour returns to u, so the last appearance of v is before the last appearance of u. Conversely, assume $l(u)<l(v)\leq r(v)<r(u)$ and that u is not ancestor of v. Since $l(u)<l(v)$ and the complete subtree of u is visited before v, this gives $r(u)<l(v)$ - contradiction

Lemma: Let u,v be distinct nodes of tree T=(V,E); then

1. u is an ancestor of v iff $l(u)<l(v)\leq r(v)<r(u)$
2. u and v are not related (neither ancestor of the other) iff $r(u)<l(v)$ or $r(v)<l(u)$
3. If $r(u)<l(v)$ then LCA(u,v) is a node with the minimum level in $A[r(u),\dots,l(v)]$

Proof (2):
If u and v are not related, then either the complete subtree of u is visited before v, $r(u)<l(v)$, or the other way around, $r(v)<l(u)$. If $r(u)<l(v)$, this means that the complete subtree of u has been visited before v is visited for the first time, so u and v are not related; similar for the other case

Lemma: Let u,v be distinct nodes of tree T=(V,E); then

1. u is an ancestor of v iff $l(u)<l(v)\leq r(v)<r(u)$
2. u and v are not related (neither ancestor of the other) iff $r(u)<l(v)$ or $r(v)<l(u)$
3. If $r(u)<l(v)$ then LCA(u,v) is a node with the minimum level in $A[r(u),\ldots,l(v)]$

Proof (3):
If $r(u)<l(v)$ then all nodes in $A[r(u),\ldots,l(v)]$ are either nodes on the path from u to v, or descendants of such nodes (since the Euler tour is a DFS traversal). Per definition LCA(u,v) is the node on the path from u to v with the smallest level, the claim follows

Lemma: Let u,v be distinct nodes of tree T=(V,E); then

1. u is an ancestor of v iff $l(u)<l(v)\leq r(v)<r(u)$
2. u and v are not related (neither ancestor of the other) iff $r(u)<l(v)$ or $r(v)<l(u)$
3. If $r(u)<l(v)$ then LCA(u,v) is a node with the minimum level in $A[r(u),\ldots,l(v)]$

Conclusion:
The LCA query problem can be reduced to the range-minimum query problem: for any interval [j,k] of an array B, determine the minimum element in $B[j,\ldots,k]$

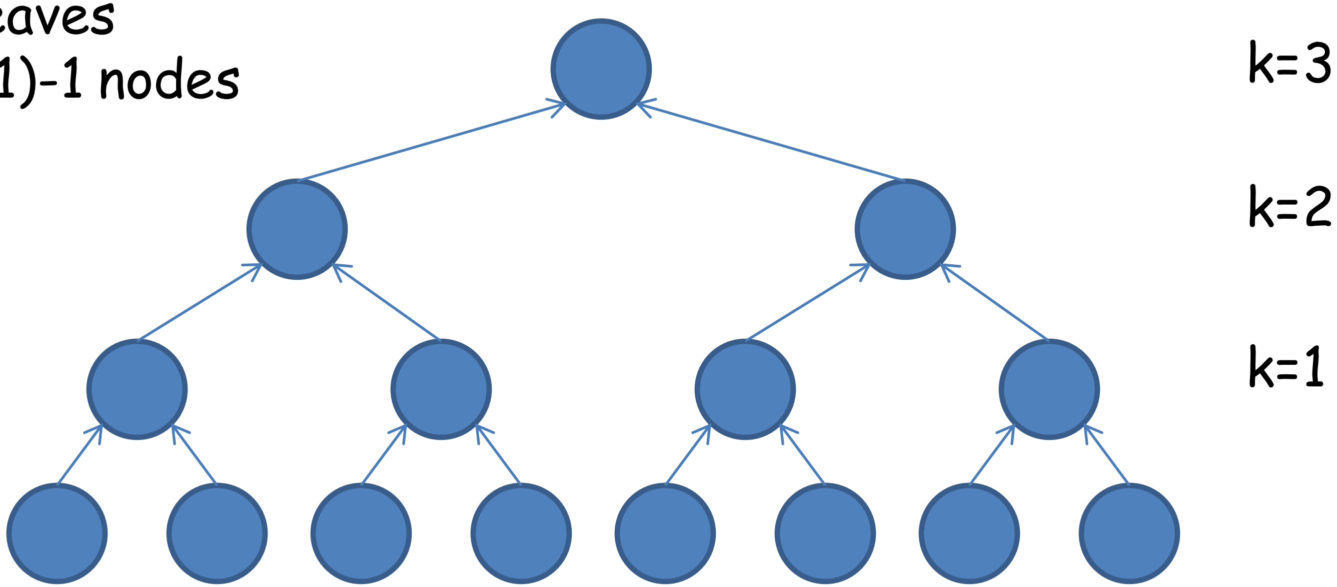
Note: There is a direct solution to the LCA query problem, see [Schieber, Vishkin: On finding lowest common ancestors: simplification and parallelization. SIAM Jour. Comp. 17(6): 1253-1262, 1988]

Theorem:
With a preprocessing step requiring O(n log n) work and O(log n) time steps on an EREW PRAM, arbitrary range minimum queries can be answered in O(1) time steps; concurrent queries require a CREW PRAM

Proof:
By the following algorithm and observation on complete, balanced trees

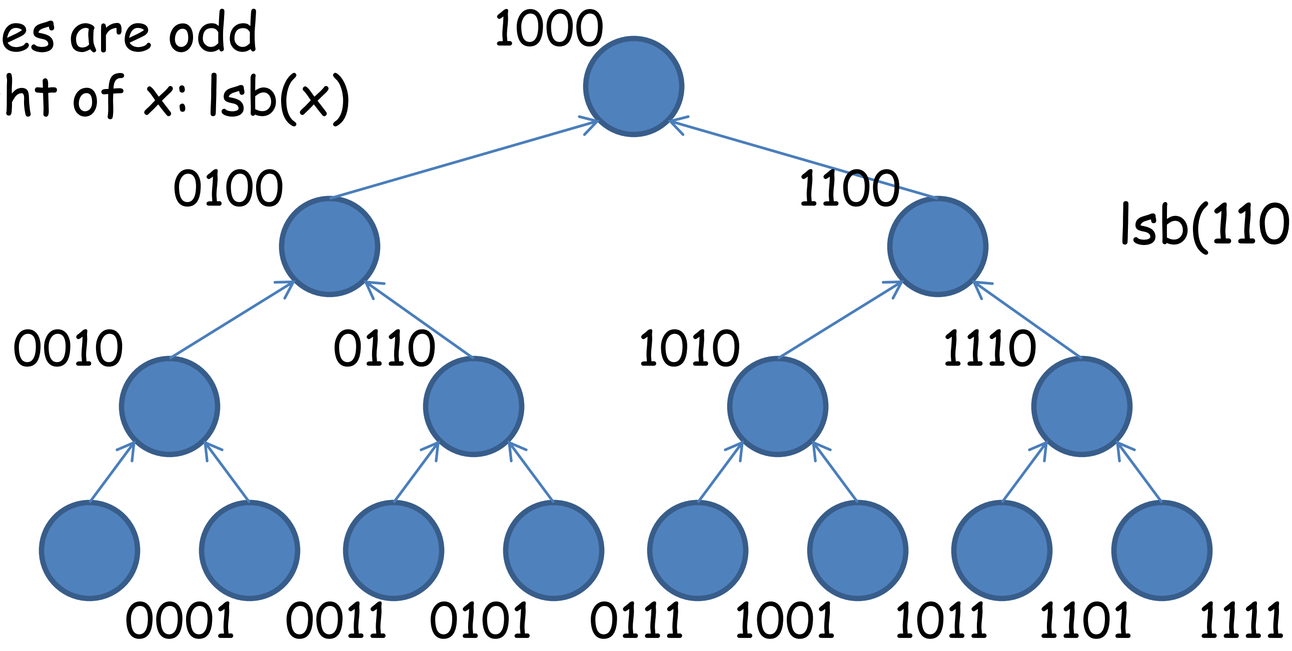
Complete binary trees: Facts and observations

Tree of height k:
- 2^k leaves
- 2^(k+1)-1 nodes



Complete binary trees: Facts and observations

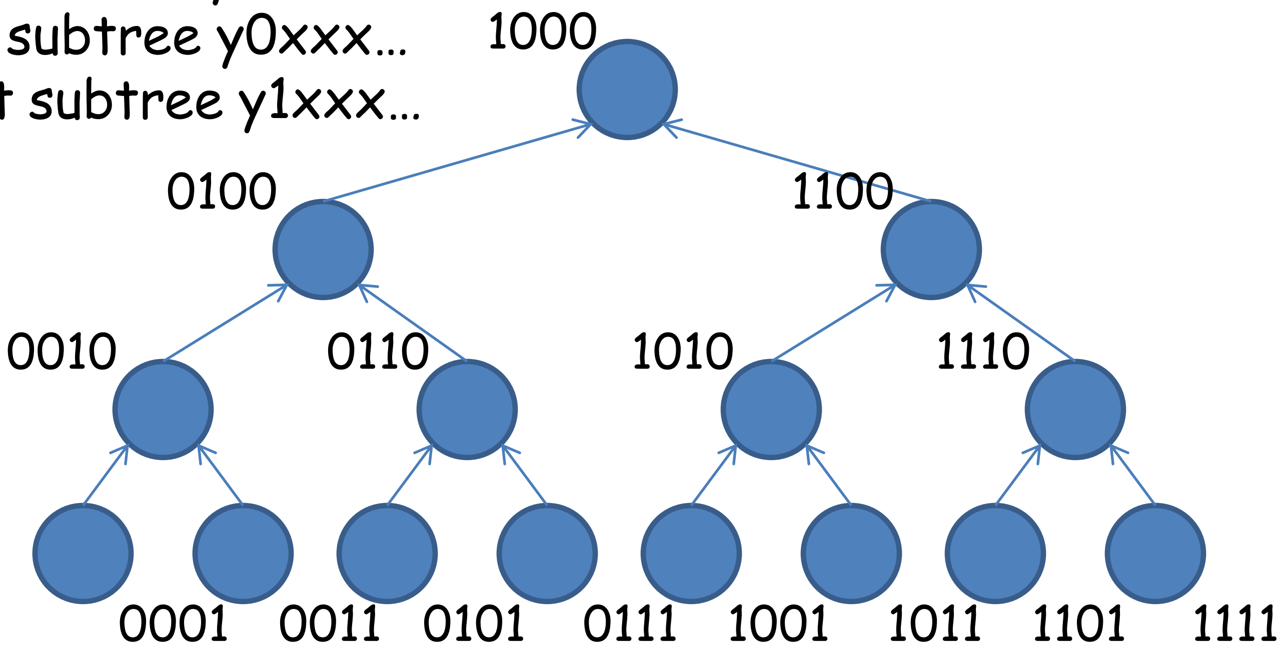
Inorder traversal, :
- Leaves are odd
- Height of x: lsb(x)



lsb(1100)=2

Complete binary trees: Facts and observations

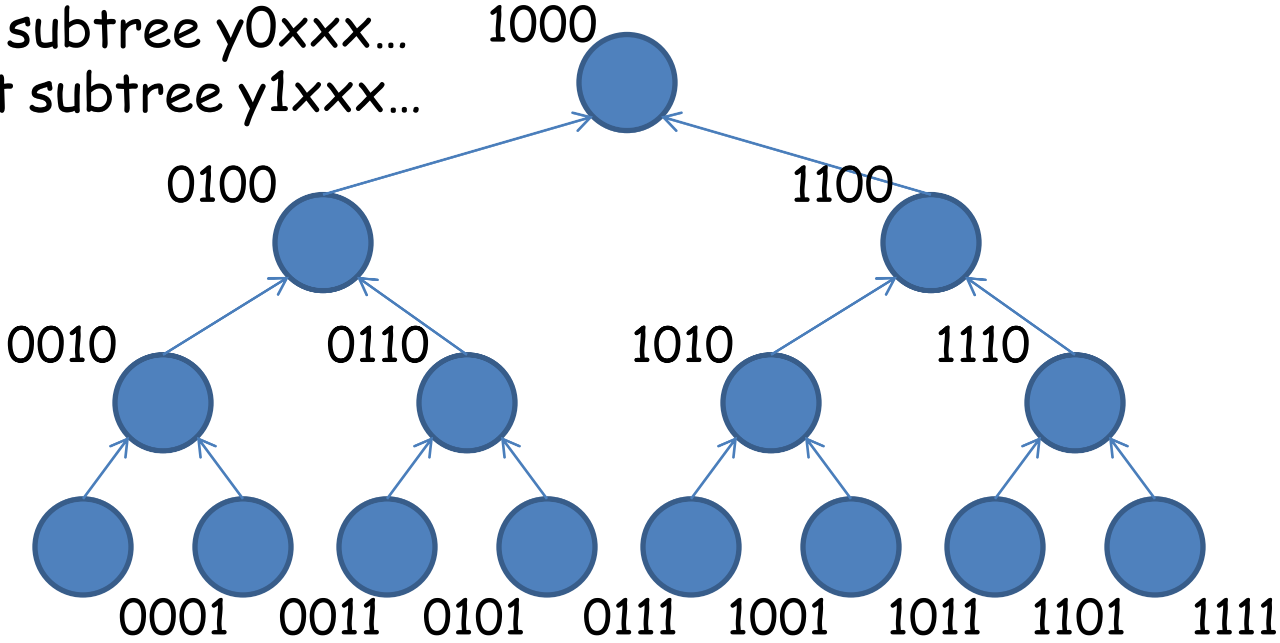
Inorder node y1000…
- Left subtree y0xxx…
- Right subtree y1xxx…



Let k=lsb(inorder(v)); the inorder numbers of the nodes in the left subtree take the form y(n-1)…y(k+1)0x(k-1)…x(0), where y(n-1)…y(k+1) are the bits of inorder(v) before k, and each x(i)=0/1

Complete binary trees: Facts and observations

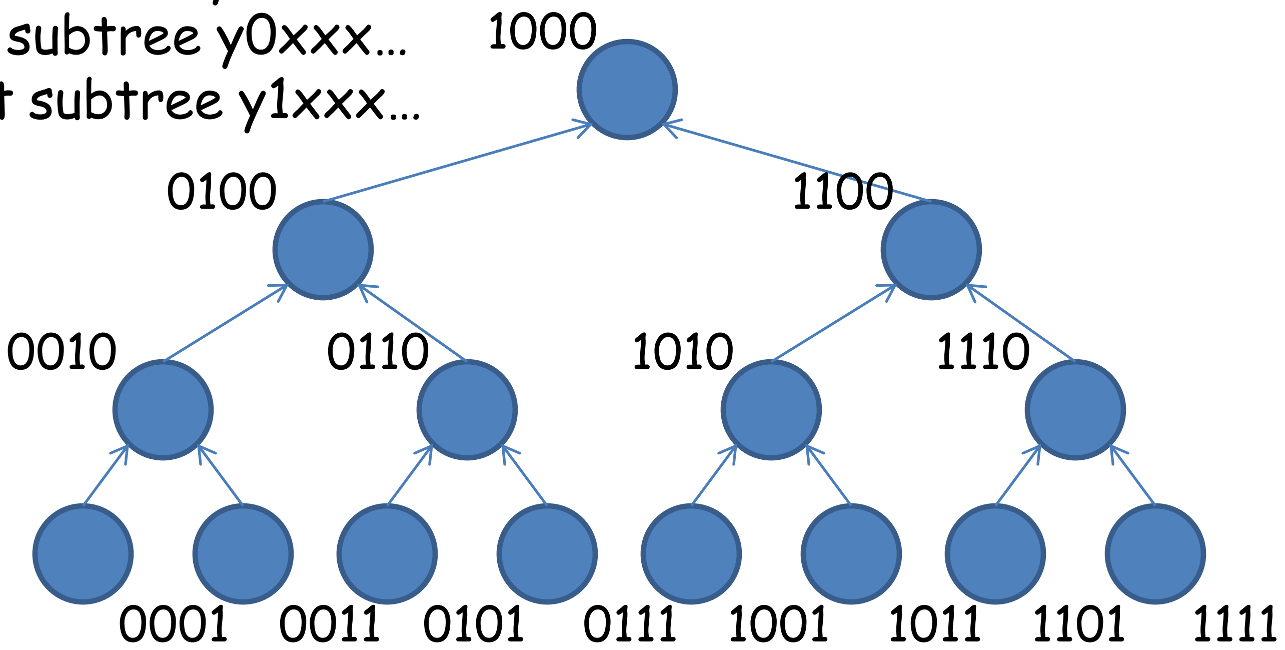
Inorder node y1000…
•Left subtree y0xxx…
•Right subtree y1xxx…



Let k = msb(inorder(u) **xor** inorder(v))
Then LCA(u,v) = (inorder(u) **and** complement(2^k-1)) **or** 2^k

Complete binary trees: Facts and observations

Inorder node y1000…
- •Left subtree y0xxx…
- •Right subtree y1xxx…



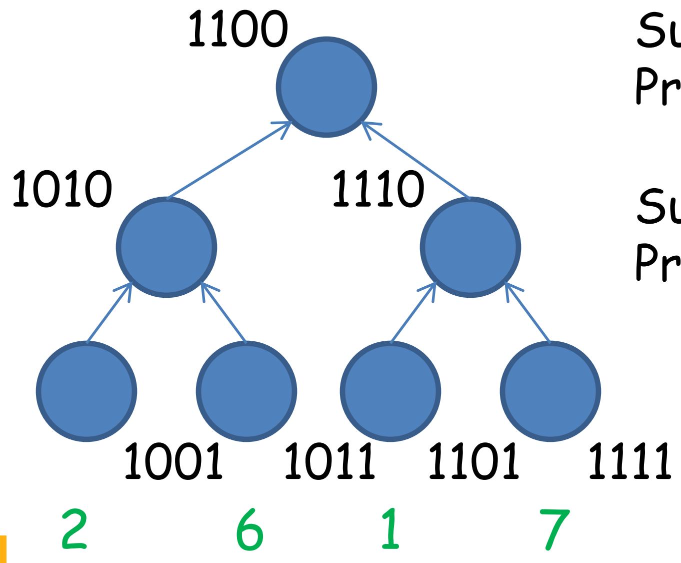
Examples:

LCA(0001,1011) = (0001&1000)|1000 = 1000
LCA(0010,0101) = (0010&1100)|0100 = 0100
LCA(1011,1110) = (1011&1100)|0100 = 1100

Let B be an array of 2^k elements. Build complete binary tree with the elements of B as leaves, B[0], B[1], … Process the tree from leaves upwards, compute for each node x the arrays of

- Suffix minima [s0,s1,…,sn] where si = min(B[i],B[i+1],…,B[n])
- Prefix minima [p0,p1,…,pn] where pi = min(B[0],…,B[i-1],B[i])

for the leaves of x



Suffix: [1,1,1,7]
Prefix: [2,2,1,1]

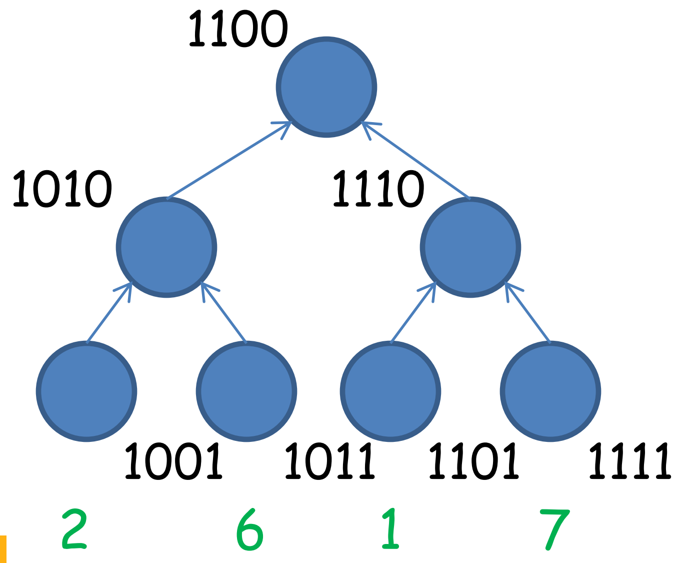
Suffix: [2,6]
Prefix: [2,2]

Suffix: [1,7]
Prefix: [1,1]

Let B be an array of 2^k elements. Build complete binary tree with the elements of B as leaves, B[0], B[1], … Process the tree from leaves upwards, compute for each node x the arrays of

- •Suffix minima [s0,s1,…,sn] where si = min(B[i],B[i+1],…,B[n])
- •Prefix minima [p0,p1,…,pn] where pi = min(B[0],…,B[i-1],B[i])

for the leaves of x

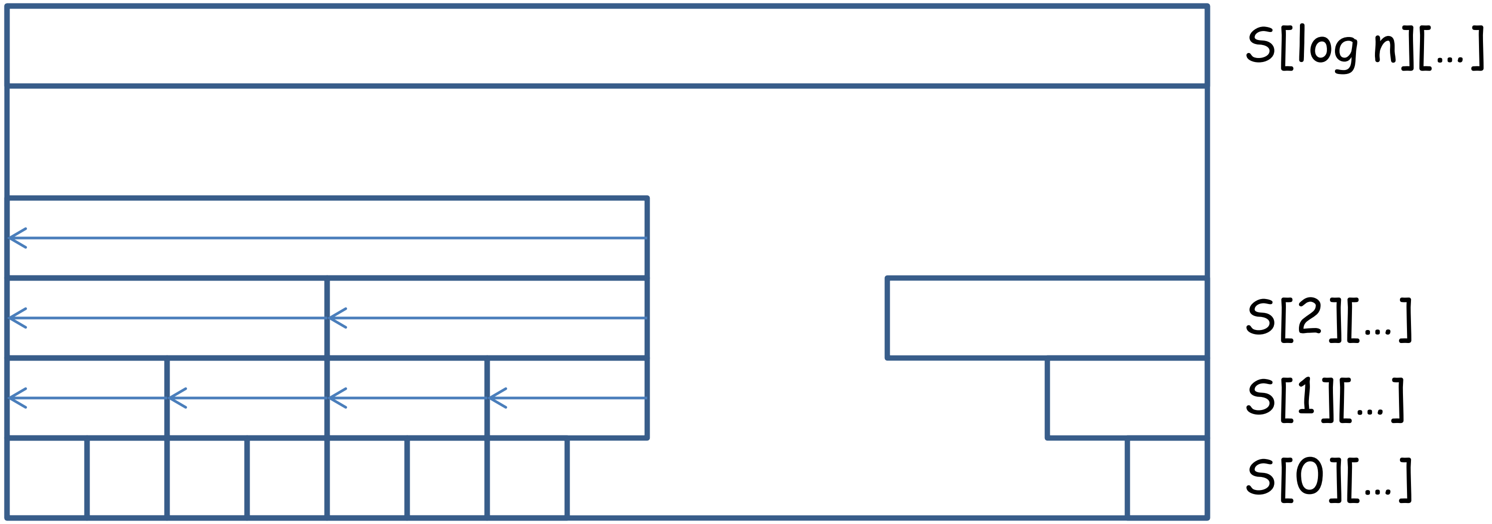


To answer RangeMinimum(x,y) query: find lowest common ancestor of x and y, range minimum is

min(suffixmin(x),Left(LCA(x,y)),
      prefixmin(y),Right(LCA(x,y)))

where Right(z) and Left(z) are right and left children of z

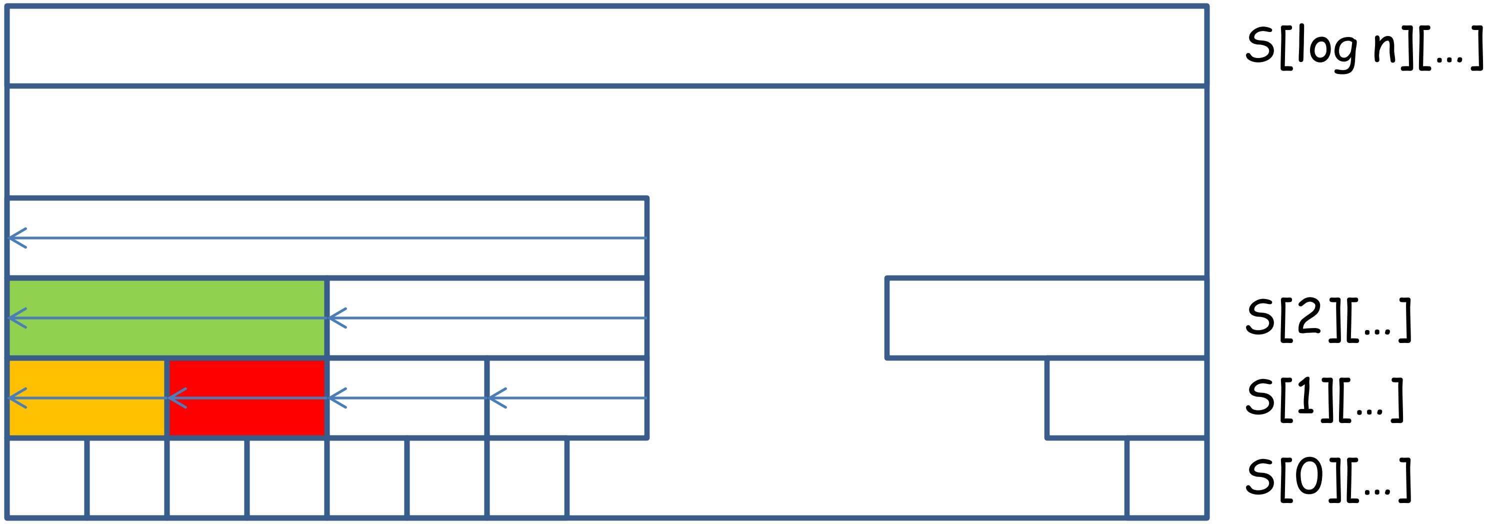
Implementation: suffix minima construction



```
par (0≤i<n) S[0][i] = B[i];
```

Initialization

Implementation: suffix minima construction



```
par (2^(k-1)≤i<2^k) S[k][i] = S[k-1][i];
par (0≤i<2^(k-1))
  S[k][i] = min(S[k-1][i],S[k-1][2^(k-1)]);
```

From level k-1 to k (one segment)

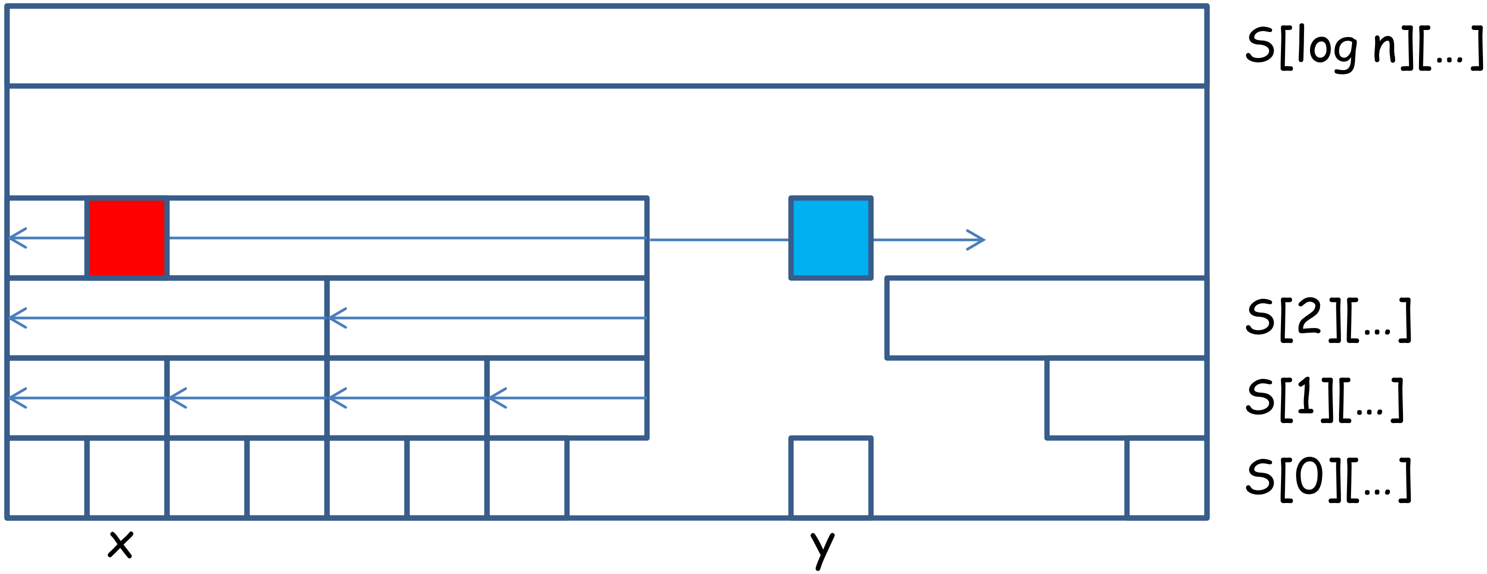
All levels, all segments (prefix minima equivalent)

```
par (0≤i<n) S[0][i] = B[i];
for (k=1; k<log n; k++) {
   par (j=0,2^k,2*2^k,3*2^k, …) {
      par (2^(k-1)≤i<2^k) S[k][j+i] = S[k-1][j+i];
      par (0≤i<2^(k-1))
        S[k][j+i] = min(S[k-1][j+i],S[k-1][j+2^(k-1)]);
  }
}
```

The algorithm computes suffix minima for all blocks of size 2^k, k=0,…,log n in O(n log n) operations and O(log n) time steps. Concurrent reading, but this can easily be eliminated

Exercise: Complete the algorithm for Suffix/Prefix minima for any n, give an EREW PRAM implementation

Implementation: range query



RangeMin(x,y) = min(S[k][x],P[k][y]) where k = msb(x **xor** y)

The algorithm can be made optimal by blocking (blocks of size O(log n)), but requires a linear time sequential algorithm (not trivial)

Theorem: With O(n) work and O(log n) time preprocessing on an EREW PRAM, range minimum queries can be answered in O(1) time

Theorem: With O(n) work and O(log log n) time preprocessing on an CRCW PRAM, range minimum queries can be answered in O(1) time

Hint: fast minimum, accelerated cascading

Exercise?

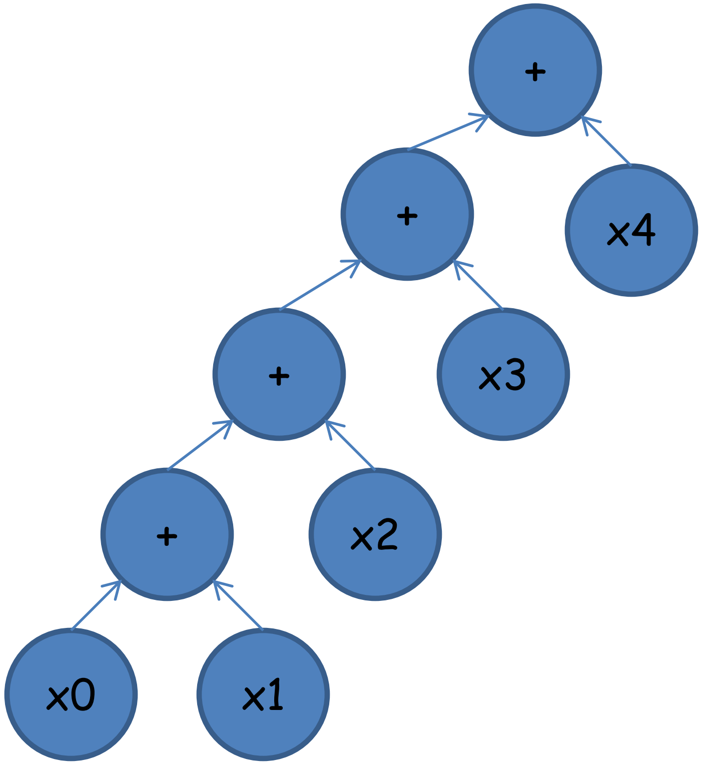
Work can be reduced using the techniques and ideas seen so far

Proposition: With O(n log log n) work and O(log n) time preprocessing on an EREW PRAM, range minimum queries can be answered in O(1) time

Proof sketch: Block array into n/log n pieces of log n elements. Apply the LCA preprocessing sequentially within each block. Apply LCA preprocessing on the n/log n block minima/maxima. Total work is O((n/log n) (log n log log n) + n/log n log(n/log n)) = O(n log log n). To answer LCA queries: combine query within local blocks with query in global structure

Exercise: work out the details

Tree contraction

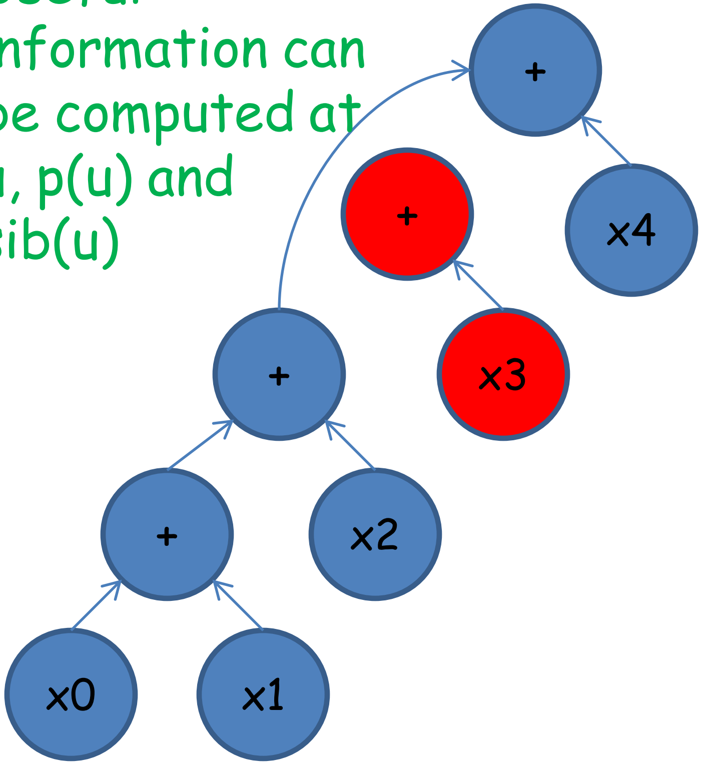


Contract tree by rake operations:

Given leaf u, p(u)≠r, with sibling sib(u), remove u and p(u), set p(sib(u)) = p(p(u))

Tree contraction

Useful information can be computed at u, p(u) and sib(u)



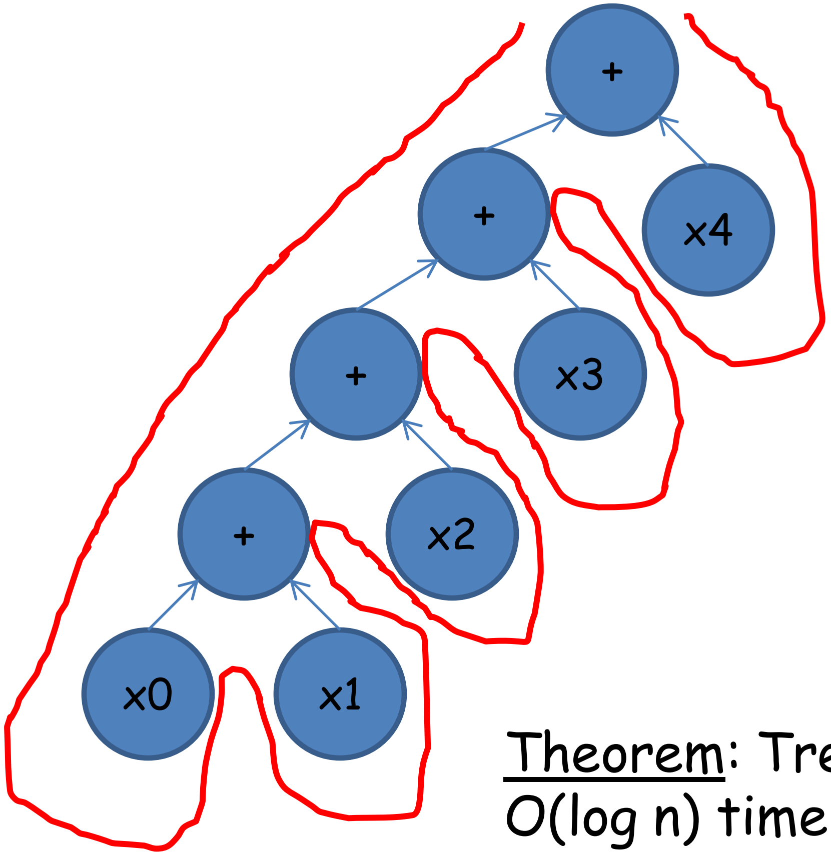
Contract tree by rake operations:

Given leaf u, p(u)≠r, with sibling sib(u), remove u and p(u), set p(sib(u)) = p(p(u))

Use Euler tour to number leaves, schedule such that up to half the remaining leaves can be removed in each parallel step

Example application: Expression tree evaluation

Tree contraction: detail sketch



Input: complete, binary tree
Output: contracted tree, root with 2 leaves

For ceil(log n) iterations:

1. Number leaves consecutively, 0,…,n-1
2. Apply rake to even-numbered, left-leaves
3. Apply rake to remaining, even-numbered leaves

<u>Theorem</u>: Tree contraction can be done in O(log n) time steps with O(n) operations on an EREW PRAM