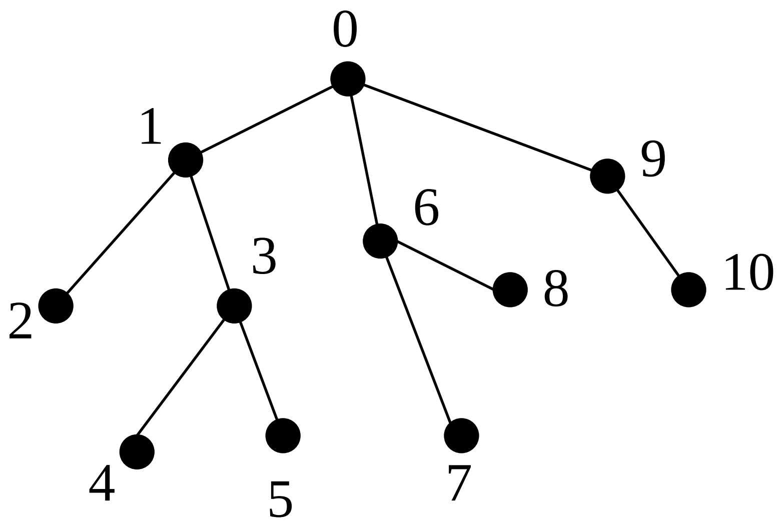
2-INF-237 Vybrané partie z dátových štruktúr

2-INF-237 Selected Topics in Data Structures

- Instructor: Broňa Brejová
- E-mail: brejova@fmph.uniba.sk
- Office: M163

- Course webpage: http://compbio.fmph.uniba.sk/vyuka/vpds/

Lowest common ancestor (LCA), najnižší spoločný predok



v is ancestor of u if it is on the path from u to the root

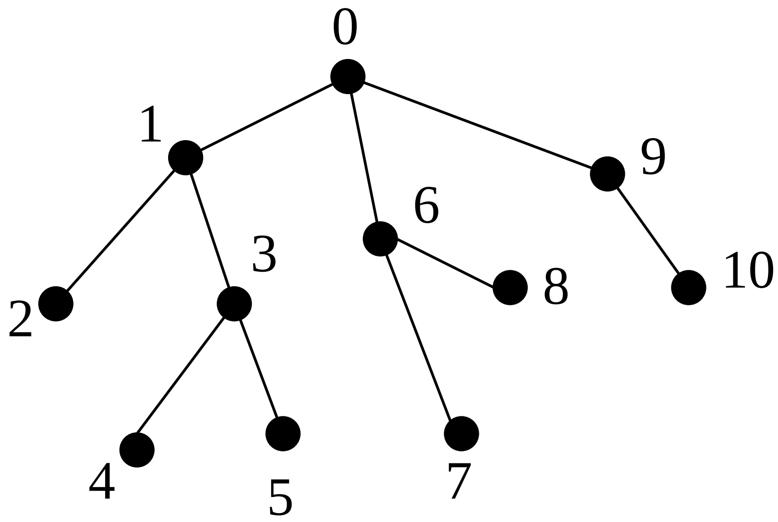
lca(u, v): node of greatest depth in ancestors$(u) \cap$ ancestors(v)

Task: preprocess tree T in $O(n)$, answer lca(u, v) in $O(1)$

Harel and Tarjan 1984, Schieber a Vishkin 1988 (Gusfield book),

Bender and Farach-Colton 2000 (this lecture)

Back to static trees



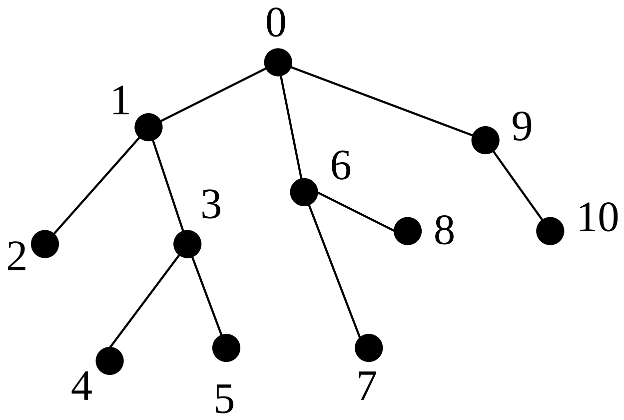
Task: preprocess tree T in $O(n)$, answer lca(u, v) in $O(1)$

Trivial solutions:

– no preprocessing, $O(n)$ time per lca

– $O(n^3)$ preprocessing, $O(n^2)$ memory, $O(1)$ time per lca

Lowest common ancestor (LCA)



lca(u, v): node of greatest depth which is ancestor of both u and v

Task: preprocess tree T in $O(n)$, answer lca(u, v) in $O(1)$

Range minimum query (RMQ)

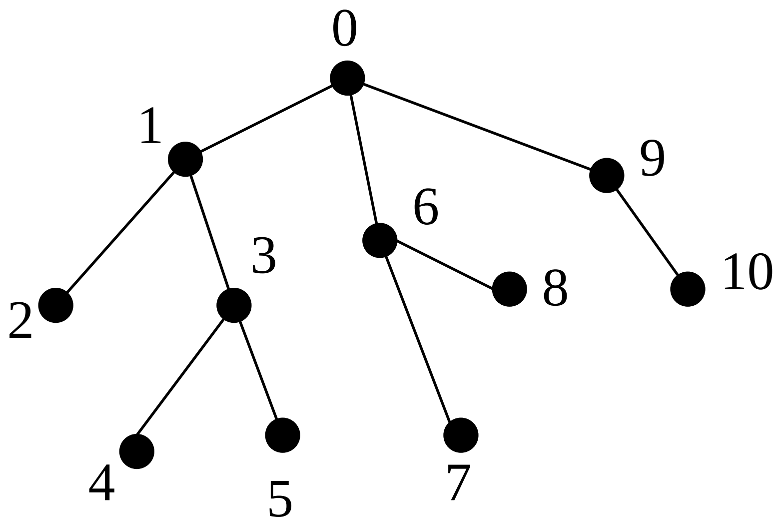
Array A of size n

$\mathrm{RMQ}(i, j) = \arg\min_{k \in \{i, i+1, \ldots, j\}} A[k]$

Task: preprocess array A in $O(n)$, then answer RMQ(i, j) in $O(1)$

Solving LCA using RMQ

Preprocess tree to arrays V, D, R



V – visited nodes

D – their depths

R – first occurrence of node in V

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V:	0	1	2	1	3	4	3	5	3	1	0	6	7	6	8	6	0	9	10	9	0
D:	0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0	1	2	1	0

i:	0	1	2	3	4	5	6	7	8	9	10
R:	0	1	2	4	5	7	11	12	14	17	18

Solving LCA using RMQ

```
search(root, 0); // call recursion

void search(node v, int depth) {
  R[v] = V.size;
  V.push_back(v);
  D.push_back(depth);
  foreach child u of v {
      search(u, depth+1);
      V.push_back(v);
      D.push_back(depth);
  }
}
```

RMQ algorithm 3

$M[i,k]$: index of minimum in $A[i..i+2^k-1]$ for $k=1,\ldots,\lfloor \lg n \rfloor$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
A[i]	0	1	2	1	2	3	2	1	0	1	0	1	0
k=1	0	1	3	3	4	6	7	8	0	10	10	12	-
k=2	0	1	3	3	7	8	8	8	8	10	-	-	-
k=3	0	8	8	8	8	8	-	-	-	-	-	-	-

RMQ algorithm 3

$M[i,k]$: index of minimum in $A[i..i+2^k-1]$ for $k = 1, \ldots, \lfloor \lg n \rfloor$

Preprocessing:

if $A[M[i,k-1]] < A[M[i+2^{k-1},k-1]]$, $M[i,k] = M[i,k-1]$

else $M[i,k] = M[i+2^{k-1},k-1]$

RMQ(i,j):

let $k = \lfloor \lg(j-i+1) \rfloor$

if $A[M[i,k]] < A[M[j-2^k+1,k]]$, return $M[i,k]$

else return $M[j-2^k+1,k]$

Time: $O(n \log n)$ preprocessing, $O(1)$ query

LCA algorithm overview

- Compute arrays V,D,R by depth-first search in the tree
- Enumerate all possible $+1, -1$ blocks of length $m-1$, precompute answers for all intervals in each type
- Split D into blocks of length $m = \log_2(n)/2$, precompute minimum and its index in each block (A', M'), find type of each block
- Precompute $O(n' \log n')$ data structure for RMQ in A'
- For $\mathrm{lca}(u, v)$ a query:
  $i = R[u], j = R[v]$, find position k of minimum in $D[i..j]$ as follows:
  – find block b_i containing i, block b_j containing j
  – compute minimum in $b_i \cap [i, j]$, $b_j \cap [i, j]$
  – compute minimum in $A'[b_{i+1} \dots b_{j-1}]$
  – find minimum of three numbers, let k be its index in D
  return $V[k]$

Lowest common ancestor (LCA)

Range minimum query (RMQ):

Alg.1 no preprocessing, $O(n)$ query

Alg.2 $O(n^2)$ preprocessing, $O(1)$ query

Alg.3 $O(n \log n)$ preprocessing, $O(1)$ query

± 1RMQ: $O(n)$ preprocessing, $O(1)$ time

split to blocks, use alg.2 within blocks, alg.3 between blocks

many blocks repeat in input – save time

LCA: $O(n)$ preprocessing, $O(1)$ time

use ± 1RMQ on array of depths in depth-first search

RMQ: want $O(n)$ preprocessing, $O(1)$ time

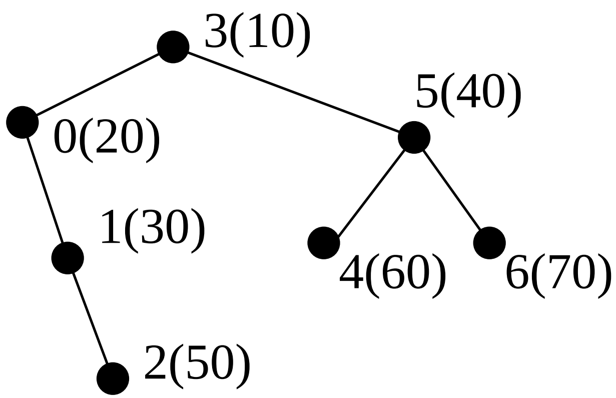
convert back to LCA!

RMQ using LCA

Cartesian tree for A: root: minimum in A (at position k)

left subtree: recursively for $A[1..k-1]$

right subtree: recursively for $A[k+1..n]$



i	0	1	2	3	4	5	6
$A[i]$	20	30	50	10	60	40	70

$A \longrightarrow$ Cartesian tree in $O(n)$: add elements from left to right

min $A[i..j] =$ lca(i,j)

Building a Cartesian tree

Use auxiliary value $a[-1] = -\infty$

```
root = new node(-1, null);
r = root;
for(int i=0; i<n; i++) {
   while(a[r.id]>a[i]) {
      r = r.parent;
   }
   v = new node(i, r);
   v.left = r.right;
   r.right = v;
   r = v;
}
```

Lowest common ancestor (LCA)

Range minimum query (RMQ):

Alg.2 $O(n^2)$ preprocessing, $O(1)$ query

Alg.3 $O(n \log n)$ preprocessing, $O(1)$ query

± 1RMQ: $O(n)$ preprocessing, $O(1)$ time

split to blocks, use alg.2 within blocks, alg.3 between blocks

LCA: $O(n)$ preprocessing, $O(1)$ time

use ± 1RMQ on array of depths in depth-first search

RMQ: $O(n)$ preprocessing, $O(1)$ time

use LCA on Cartesian tree

Direct method: split to blocks,

represent blocks by Cartesian trees (Fischer and Heun 2006)

Precomputing values over intervals

Operation $\circ$, compute $R_\circ(i,j) = A[i] \circ A[i+1] \circ \cdots \circ A[j]$

- Precompute all answers: $O(n^2)$ preprocessing, $O(1)$ query
- Precompute prefix "sums" $R_\circ(0,i)$

  good for groups (e.g. $\circ = +$ over Z,Q,R,etc.)

  $R_\circ(i,j) = R_\circ(0,j)) \circ R_\circ(0,i-1))^{-1}$

  Optional HW: what about multiplication?

- Precompute intervals of sizes 2^i

  combine 2 overlapping answers e.g. for min

- Segment trees: precompute non-overlapping intervals of sizes 2^i

  combine several intervals to cover each element exactly once

  good for any associative $\circ$, e.g. matrix multiplication

  also good in case of dynamic updates of the array

Segment tree

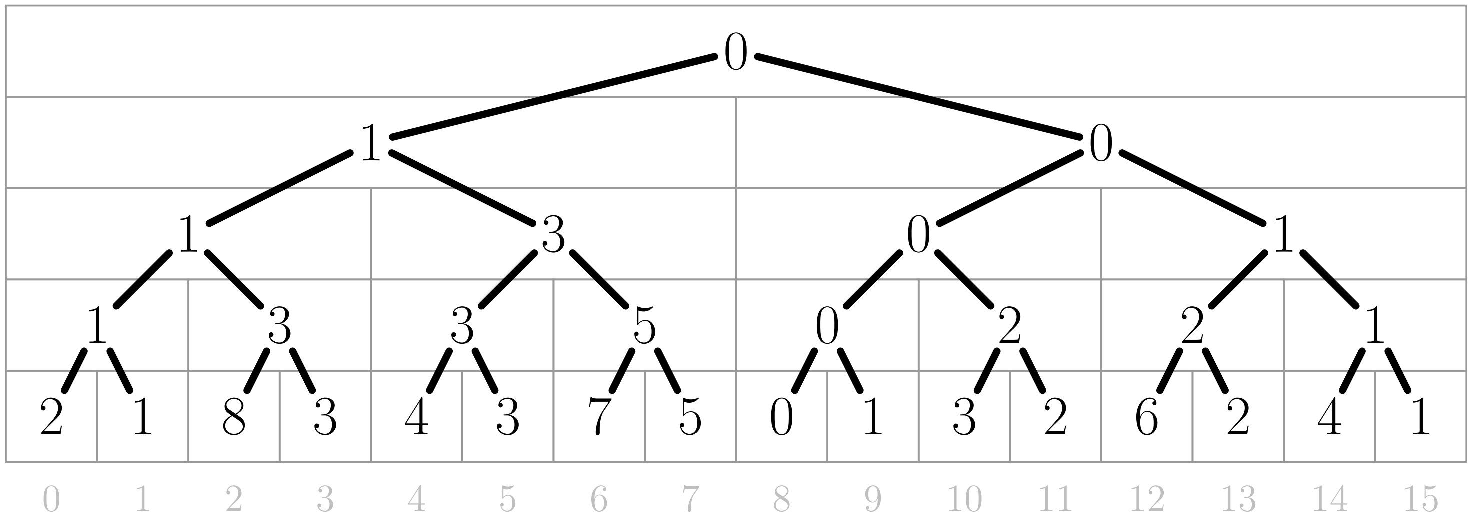
- Root correspods to interval $[0, n)$
- Leafs correspond to intervals $[i, i + 1)$
- If a node corresponds to $[i, j)$

  left child corresponds to $[i, k)$, right child to $[k, j)$

  where $k = \lfloor (i + j)/2 \rfloor$
- For each node $[i, j)$ store $R_\circ(i, j - 1) = A[i] \circ \cdots \circ A[j - 1]$
- Total number of nodes $2n - 1$, height $\lceil \lg n \rceil$

Segment tree

Example for $\circ = \min$

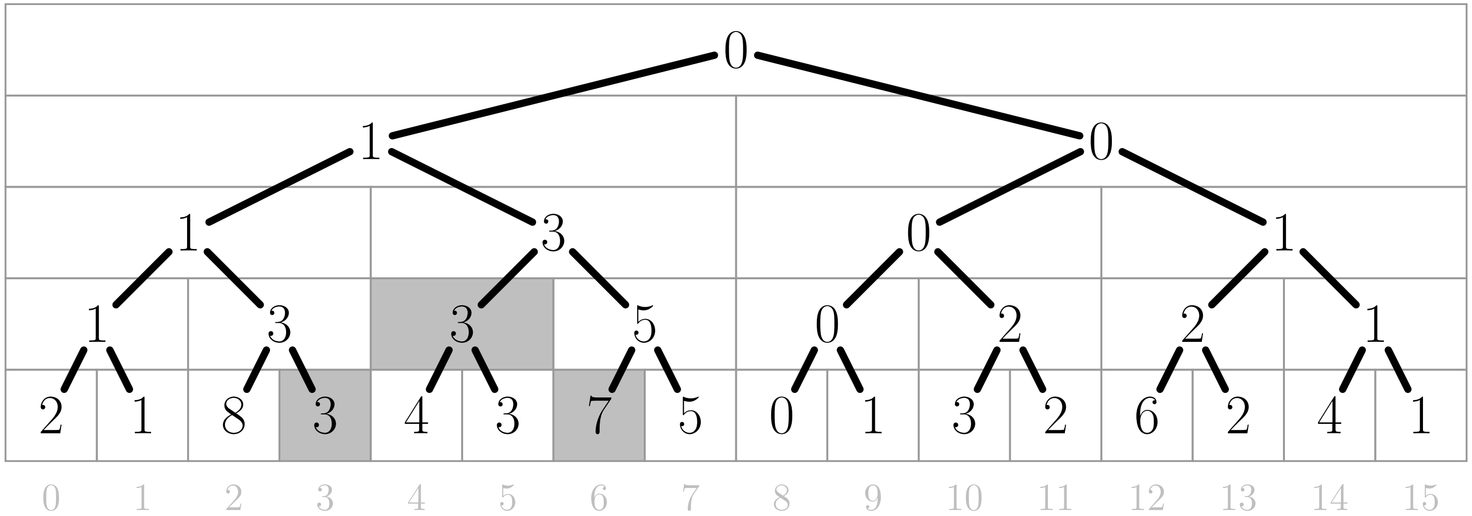


It might be more useful to store position of minimum, rather than value

The structure can be stored in an array similarly as binary heap

Canonical decomposition

Decompose query interval $[x, y)$ to a set of disjoint tree intervals



Canonical decomposition

Decompose query interval $[x, y)$ to a set of disjoint tree intervals

- Current node $[i, j)$, and its left child $[i, k)$
  invariant: $[i, j)$ overlaps with $[x, y)$
- If $[i, j) \subseteq [x, y)$, return $\{[i, j)\}$
- $R = \emptyset$
- If $[i, k)$ overlaps with $[x, y)$, recurse on left child, add to R
- If $[k, j)$ overlaps with $[x, y)$, recurse on right child, add to R
- Return R

Segment tree (summary)

- Tree of intervals, height $O(\log n)$
- Root: entire array; leaves: intervals of length 1
- Each node stores the result of operation $\circ$ on its interval
- Each internal node split into two disjoint intervals, left and right child
- Canonical decomposition: Each query interval can be written as union of $O(\log n)$ disjoint intervals, these can be found in $O(\log n)$ time
- To compute $A[i] \circ \cdots \circ A[j]$, we need $O(\log n)$ time for any associative $\circ$
- Update of an element in A can be done in $O(\log n)$ time

Finding all small numbers

We have array A precomputed for RMQ.

For given i, j, x find all indices $k \in \{i, \dots, j\}$ s.t. $A[k] \leq x$.

```
void small(i, j, x) {
  if(j > i) return;
  k = rmq(i, j);
  if(a[k] <= x) {
    print k;
    small(i, k-1, x);
    small(k+1, j, x);
  }
}
```

Running time $O(p)$, where p is the number of printed indices