CS 261: Graduate Data Structures

Week 8: Navigating in trees

David Eppstein
University of California, Irvine

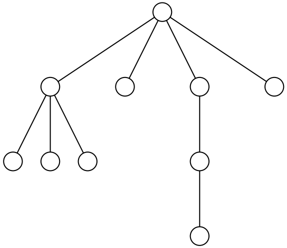
Spring Quarter, 2021

Representing trees succinctly

Arbitrary trees with ordered children

Write nested parentheses for each node,
surrounding string representing the subtree below it

Convert to binary: “(” = 0, “)” = 1



```
0 00101011  01  000111  01 1
(  ( ( ) ( ) ( ) )  ( )  ( ( ( ) ) )  ( )  )
```

left child and its
three grandchildren

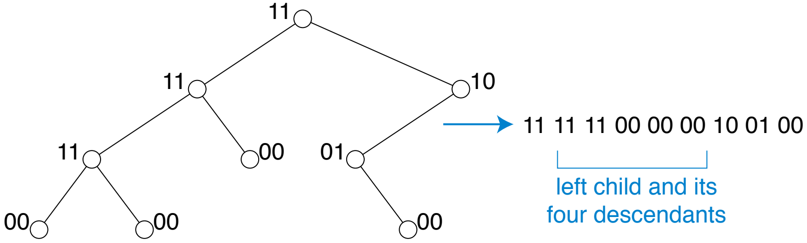
n-node tree ⇒ $2n$-bit binary number

Binary trees

Parentheses don't work: can't tell if single child is left or right

Traverse tree in preorder (root first, then children)

For each node, write down two bits:
Does it have a left child (0 for no, 1 for yes)?
Does it have a right child (0 for no, 1 for yes)?



n-node tree $\Rightarrow$ $2n$-bit binary number

Both codes are near-optimal

In each case, # trees is a Catalan number

$$C_n = \binom{2n}{n} \Big/ n + 1 \approx \frac{4^n}{\sqrt{n^3 \pi}}$$

(C_n for n-node binary trees, C_{n-1} for ordered trees)

So in order to have a different code for each tree, # bits must be

$$\geq \log_2 C_n = 2n - O(\log n)$$

Navigation in succinct trees

Extension of succinct representations (not enough time in schedule to include details in this class) allow fast parent/child lookups or other operations in representations with $2n + o(n)$ bits

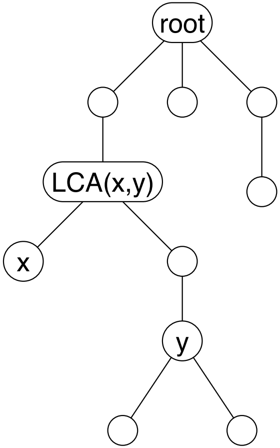
Typical strategy:

- Break tree into blocks of logarithmic size (either subtrees or substrings of succinct representation)
- Handle navigation within a block by table lookup with succinct representation as index
- Handle block-to-block navigation by a non-succinct data structure on fewer elements (the number of blocks rather than the number of tree nodes)

We will see similar block structure in some of the other structures we study this week

Common ancestors and their applications

Lowest common ancestor



Lowest common ancestor of two nodes x and y in a rooted tree is:

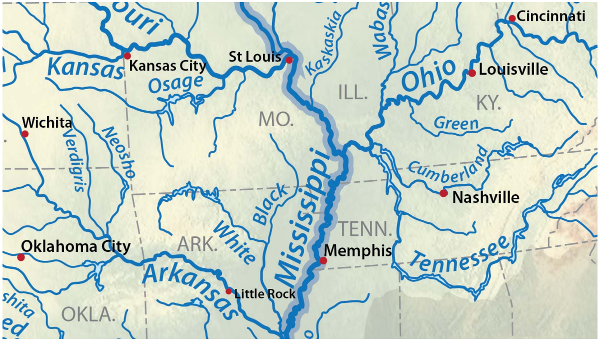
- An ancestor of x (on path to root)
- An ancestor of y
- Lower (farther from root) than any other common ancestor

It may be x or y itself, if one is an ancestor of the other

Shortest paths in undirected trees

Data: An undirected tree with lengths on edges

Goal: Process so we can quickly look up distances between pairs of vertices, or find shortest path between vertices



E.g. what are the river-distances between these nine cities?

Cropped from File:Mississippiriver-new-01.png by Shannon1 on Wikimedia commons

Shortest path solution

Choose a root for the tree (the mouth of the river)

Build a data structure for finding lowest common ancestors
(the point where the streams from any two cities meet)

Store the distance to the root at each node

Then:

- $\text{distance}(x, y) =$
  $\text{distance}(x, \text{root}) + \text{distance}(y, \text{root}) - 2 \cdot \text{distance}(\text{ancestor}, \text{root})$
- Path from x to y can be found by following parent links from x to common ancestor, and from y to common ancestor, and then gluing together these two paths to the ancestor

Bandwidth in computer networks

Network: Graph of computers and routers with communication links (edges) between them

Different communication links have different bandwidths (number of bits/second you can transfer across the link)

Bandwidth of a path = minimum bandwidth of any link

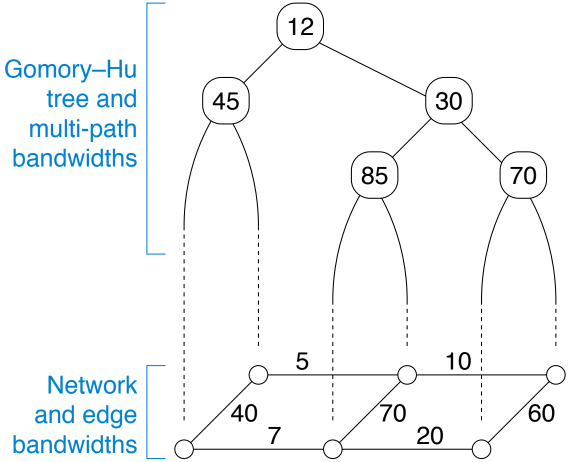
Can also route messages along multiple paths as long as total traffic on any edge is at most its bandwidth

Gomory–Hu tree

Hierarchical clustering of vertices of a graph, with clusters labeled by numbers

Bandwidth between any two nodes is the label of their ancestral cluster

Can be defined for either single or multi-path routing



Gomory–Hu tree and multi-path bandwidths
12
45
30
85
70
Network and edge bandwidths
5
10
40
70
60
7
20

The remaining question

Both shortest paths and bandwidth motivate the question:

If we are given as data a rooted tree,

How can we design and build a data structure

For quickly answering *lowest common ancestor* queries?

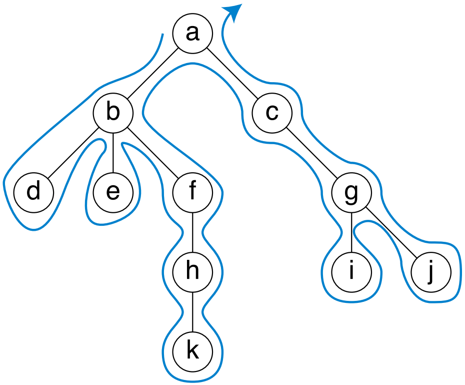
From ancestors to range minima

Euler tours (intuition)

Draw a tree in the plane, from the root down

Trace your finger around the boundary of the drawing

What order does it reach each vertex (multiple times)?



a, b, d, b, e, b, f, h, k, h, f, b, a, c, g, i, g, j, g, c, a

Each edge traced twice $\Rightarrow$ # vertices in tour $= 2n-1$

Euler tours (pseudocode)

To produce an Euler tour of a subtree rooted at x:

- Output x
- For each child y of x:
  - Recursively produce an Euler tour for the subtree rooted at y
  - Output x again

Like

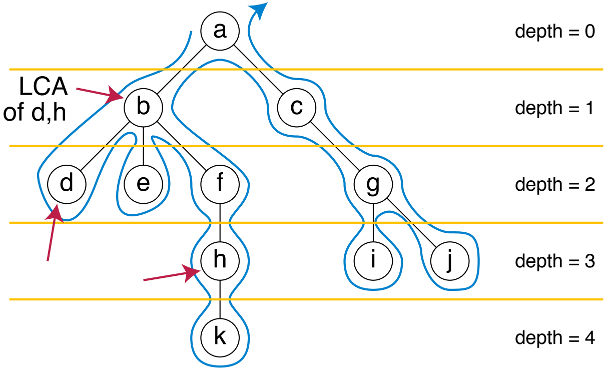
preorder (root comes before children),
inorder (root comes between children), and
postorder (root comes after children),

all merged into one

Ancestors and Euler tours

Number tree vertices by depth (distance from root)

LCA(x, y) = smallest depth node between first occurrence of x and first occurrence of y in the Euler tour



a:0, b:1, d:2, b:1, e:2, b:1, f:2, h:3, k:4, h:3, f:2, ...

range between 1st occurrences of d, h

From ancestors to range minima

Given a tree, to be processed for ancestor queries:

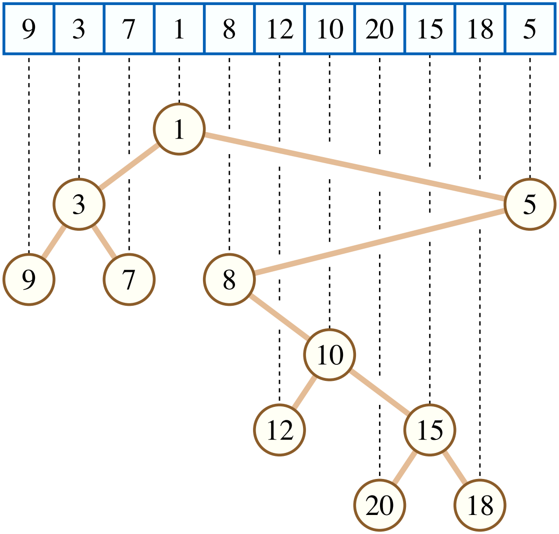
- Number vertices by depth (distance from root)
- Make array E of $2n-1$ vertices, where $E[i] =$ ith vertex of Euler tour
- Make array D of $2n-1$ numbers, $D[i] = \text{depth}(E[i])$
- Store for each vertex v the position $P[v]$ of its first occurrence in the tour
- Build a data structure for range minimum queries in D

Then $\text{LCA}(x, y) = E[\text{rangemin}(P[x], P[y])]$

So how quickly can we answer range minimum queries?

Cartesian trees

Example



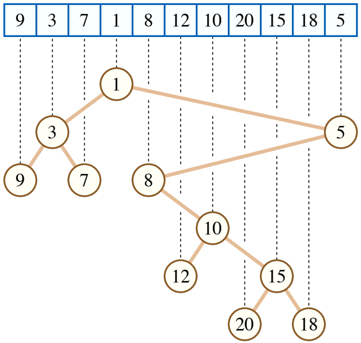
Recursive definition

Defined from an array of numbers

Root is the minimum number (leftmost copy if there are ties)

Left subtree defined recursively from subarray to left of root

Right subtree defined recursively from subarray to right of root



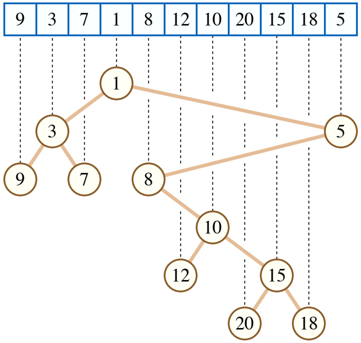
Axiomatic definition

Binary tree whose nodes are the numbers in a given array

(but not a binary search tree)

Inorder traversal order of the tree = array order

Heap-ordered

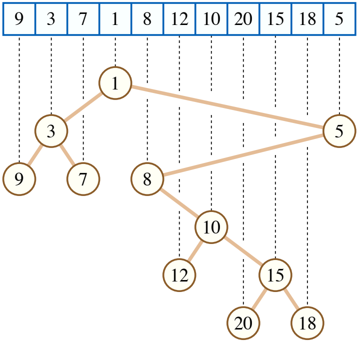


Linear time construction

Add numbers to tree in left-to-right order

For each number $A[i]$

- Follow parent links from $A[i-1]$ until finding ancestor that is $\leq A[i]$
- Add x as right child
- Add previous right child as left child of x



Amortized time for adding $A[i] = O(1)$ with potential function Φ = length of path from root to rightmost node

E.g. when adding 5 to this tree, path of parent links = 18—15—10—8—1, ancestor = 1, and previous right child = 8

From range minima to ancestors

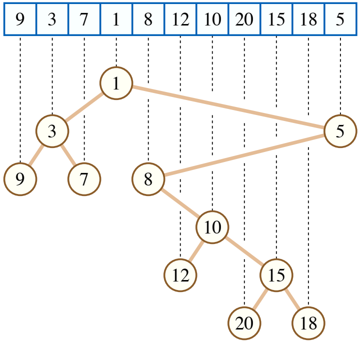
For any array A of numbers

Minimum of A in range $[i, j]$ $=$ LCA of $A[i]$ and $A[j]$ in the Cartesian tree of A

Explanation: The recursive definition chooses roots of subtrees in sorted order

$A[i]$ and $A[j]$ stay together in the same subtree until we choose a root between them

That root must be the smallest number in the range



Circular logic

By constructing an Euler tour, we can convert ancestor problems into range minimum problems

tree with n nodes $\Rightarrow$ array with $2n - 1$ numbers

By constructing a Cartesian tree, we can convert range minimum problems into ancestor problems

array with n numbers $\Rightarrow$ tree with n nodes

How is this helpful?

Ancestor and range minimum solutions

History

- Aho, Hopcroft, and Ullman (1973): Define ancestor problem
- Dov Harel (UCI student!) and Tarjan (1984): $O(n)$ space, $O(1)$ query time, but complicated
- Schieber and Vishkin (1984): use bitwise operations
- Gabow, Bentley, and Tarjan (1984): convert range-minimum to LCA using Cartesian tree
- Berkman and Vishkin (1993): convert LCA to range-min by Euler tour; solve by table lookup + common substructures
- Bender and Farach-Colton (2000), Alstrup et al (2004), Fisher and Heun (2006): additional simplifications

Table lookup

Easy but space-inefficient solution for range minimum:

Construct a two-dimensional array $M[i, j]$
where $M[i, j]$ = the index of the minimum in range $[i, j]$

11	27	17	99	31	43
0	1	2	3	4	5

$\longrightarrow$

	$j = 0$	1	2	3	4	5
$i = 0$	0	0	0	0	0	0
1		1	2	2	2	2
2			2	2	2	2
3				3	4	4
4					4	4
5						5

Query = look up array value

Table lookup construction

$M[i,j]$ = the index of the minimum in range $[i,j]$
To construct M, loop over each pair (i,j) with $i \leq j$

- If $i = j$, range has only one element, so $M[i,j] = i$
- Otherwise, compare $A[M[i,j-1]]$
  (minimum element of subrange $[i,j-1]$) to $A[j]$
- If $A[M[i,j-1]] \leq A[j]$, whole range has same minimum as subrange, so set $M[i,j] = M[i,j-1]$
- If not, set $M[i,j] = j$

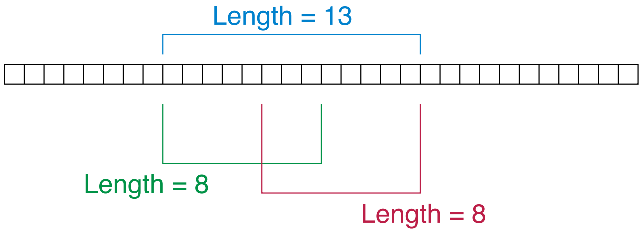
Query time: $O(1)$

Space: $O(n^2)$

Construction time: $O(n^2)$

Decomposition into power-of-two ranges

Key insight: Any interval $[i, j]$ is the union of two overlapping intervals whose length is a power of two



Answer queries in overlapping ranges and combine results

Few ranges have power-of-two lengths $\Rightarrow$ smaller lookup table

Formulas for power-of-two ranges

The largest power of two that will fit inside range $[i,j]$:

$$k = \lfloor \log_2(j-i+1) \rfloor$$

Decomposition of $[i,j]$ into two length-2^k ranges:

$$[i,j] = [i, i+2^k-1] \cup [j-2^k+1, j]$$

Make table of only length-2^k ranges, for $k = 0 \ldots \log_2 n$

$$P[i,k] = M[i, i+2^k-1]$$

(this is just definition of P; we'll see faster calculation next slide)

Answer queries in overlapping ranges and combine results

$$\mathrm{rangemin}(i,j) = \min(A[P[i,k]], A[P[j-2^k+1,k]])$$

Table construction for power-of-two ranges

To compute $P[i, k]$, the index of the minimum of the length-2^k range starting at i, simply divide it into two length-2^{k-1} ranges and compare their minima

For $k = 0, 1, \ldots \lfloor \log_2 n \rfloor$:

- If $A[P[i, k-1]] \leq A[P[i + 2^{k-1}, k-1]]$, set $P[i, k] = P[i, k-1]$
- Otherwise, set $P[i, k] = P[i + 2^{k-1}, k-1]$

Query time: $O(1)$

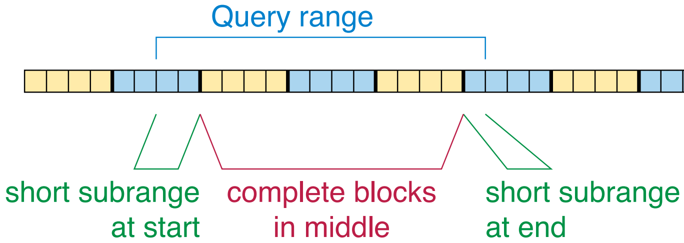
Space: $O(n \log n)$

Construction time: $O(n \log n)$

Blocking for linear space

Idea: Partition input array into blocks of some length b

Query range $\Rightarrow$ two short ranges at ends
$+$ one longer range of complete blocks in the middle



If blocks are short enough, many have similar structures
$\Rightarrow$ re-use same lookup table for short ranges of multiple blocks

Longer middle range only uses the minimum value in each block
$\Rightarrow$ less space because shorter sequence, length $\approx n/b$

Blocking for linear space

Choose block size $b \approx \frac{1}{3}\log_2 n$

Label each block by succinct representation t of its Cartesian tree

- $2b$ bits per label
- Blocks with same label behave the same

Precompute tables $M_t[i,j]$ for queries inside blocks with label t

- Number of tables $= 2^{2b} = n^{2/3}$
- Time and space per table $= b^2 = O((\log n)^2)$
- Total $= n^{2/3}(\log n)^2 = o(n)$

Use power-of-two solution for ranges of complete blocks

- Number of complete blocks $= n/b$
- Time and space $= O((n/b)\log(n/b)) = O(n)$

So space is linear and query time is constant

Maintaining order in a list

List ordering vs tree ancestors

We can view lists of elements as trees, rooted at the start of the list, with one child per node

Lowest common ancestor = which of two list elements is earlier?

Very easy solution: Number the elements by position, so the earliest element is the one with the smaller number

But what about dynamic lists, with insertion and deletion?

Formalization of the problem

Maintain a collection of elements ordered into a list

Operations:

- Insert x at the start or end of the list
- Insert x immediately before or after another element y
- Find the element immediately before or after x
- Remove x
- Test whether element x is earlier than or later than element y

Most operations can easily be done in constant time,
for example by using doubly linked lists

The only missing one: testing relative ordering

House numbering problem

Typical properties of numbers of buildings in US streets:

- They are ordered: number tells you relative position along street
- They are (usually) small integers
- They are not necessarily consecutive:
  there may be gaps in the numbering
- Renumbering is expensive, so don't do it very often

Intuition: apply similar scheme to list ordering by numbering list elements and using numbers to test relative position

Application of house numbering

Dynamic arrays allow fast lookup of the element at position i, and fast insertion or deletion at the end of the array

What if we want to extend arrays to allow insertion or deletion at other positions?

- Augmented binary search trees with ranking and unranking: $O(\log n)$ per operation (don't need house numbering)
- Maintain ordered numbering of elements, and use integer ranking and unranking structure: $O(\log n / \log \log n)$ per operation

Partial history

- Dietz 1982: logarithmic update, $O(1)$ order-comparison
- Tsakalidis 1984: constant amortized update and comparison
- Dietz and Sleator 1987: maintain ordered numbering, all numbers polynomially large, constant amortized update, complicated
- Bender, Cole, Demaine, Farach-Colton, and Zito 2002: simplification of same results
- Devanny, Fineman, Goodrich, Kopelowitz 2017: few relabelings per element

We will follow BCDFZ 2002.

Terminology

Key: The elements of our list

Tag: The number assigned to a key

We want to maintain a correspondence keys $\rightarrow$ tags so that the numerical ordering of tags $=$ the list ordering of keys

Main idea

If we delete a key, the tags of other keys do not need to change

If we insert key x, and there is an available tag i between tags of its predecessor and successor, set $\mathrm{tag}(x) = i$

Remaining case: Partition ranges of tags recursively into subranges of power-of-two sizes

Find the smallest range of tags (size 2^k) surrounding new element that is used by few keys: fewer than c^k for some $1 < c < 2$

Renumber the keys evenly within this range

Main idea analysis

The range of all possible tags needs to be small enough that n keys cannot fill it $\Rightarrow$ numbers are polynomial

When we renumber a range of size 2^k, one of its two child ranges is full but it is not, so we renumber $\Theta(c^k)$ keys and take time $\Theta(c^k)$

After renumbering, the children are below full by a factor of $c/2 \Rightarrow$ it will take $\Omega(c^k)$ insertions before they fill up and we renumber the same range again $\Rightarrow$ amortized time for this size level is $O(1)$

The same analysis holds separately for each choice of i, but there are $O(\log n)$ choices $\Rightarrow$ total amortized time per update is $O(\log n)$

Log-shaving

To achieve constant instead of logarithmic amortized time per update, again use a blocking strategy:

- Group keys into dynamic blocks of logarithmic length (similar to bottom level of a B-tree with $B = \Theta(\log n)$)
- Use main idea to give blocks disjoint ranges of polynomially many tags each
- Within blocks number new keys as average of predecessor and successor tags
- Renumber entire blocks when block structure changes or when a new block element has no available tag; this happens only once every $O(\log n)$ insertions

Image Cutting lumber with a swingblade sawmill.jpg by liempdma from Wikimedia commons

The level-ancestor problem

The level-ancestor problem

Data: A tree

Query:

- Given a vertex v and a number k, find the ancestor of v that is k steps higher in the tree
- Equivalently: Given a vertex v and a number d find the ancestor whose depth (number of steps from root) is d

We could just follow parent links up the tree in time $O(k)$ but we want small query time even when k is large

Inefficient solution

Each node v stores $O(\log n)$ ancestors, the ones k steps higher for $k = 1, 2, 4, 8, \ldots 2^i, \ldots$

To find the ancestor k steps higher when k is not a power of two:

- Round k down to a power of two, 2^i for $i = \lfloor \log_2 k \rfloor$
- Let w be ancestor of v, 2^i steps higher, as stored at v
- Recursively find ancestor $k - 2^i$ steps higher from w

Space $O(n \log n)$, query time $O(\log k)$

We can shave the log in the space (later) but first we need another idea to speed up the query time

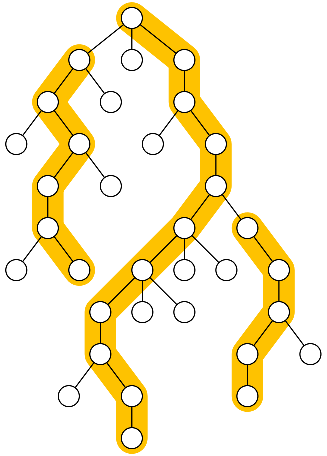
An easy special case

If the tree is a path, rooted at left end:

- Store an array of the vertices in the path
- Each node records its position in the array
- The ancestor k steps up from v is the vertex stored in the array at index $= \text{position}(v) - k$

Linear space, constant query time

Decomposition into long paths



Each non-leaf node selects one child, the one leading to the deepest leaf

The selected edges form a system of paths covering all non-leaf nodes and some of the leaves

Remaining leaves form one-vertex paths

But if we use the path solution for these paths, how do we find ancestors on different paths?

Longer paths for shallow-enough queries

For each path P of the path decomposition, extend P upwards to contain twice as many vertices (or all the way to the root, whichever comes first)

Store each extended path in an array, and for each tree vertex store both which array it belongs to and its position in the array

Total length of all arrays $\leq 2n \Rightarrow$ linear space

When vertex v is h steps above its deepest leaf descendant, its extended path extends another $h+1$ steps above it
$\Rightarrow$ can answer queries for $k \leq h+1$

Combined data structure

Store both jumps to 2^i steps higher and extended paths

To find the ancestor k steps higher from v:

- Round k down to a power of two, 2^i for $i = \lfloor \log_2 k \rfloor$
- Let w be ancestor of v, 2^i steps higher, as stored at v
- Because w is high enough above v, we can use the extended paths to find the ancestor $k - 2^i$ steps higher from w directly, rather than recursing

Constant query time but space is still $O(n \log n)$

(The paths use linear space but the jump tables are too big)

Log-shaving

Ancestor of v at height d = next vertex with height d in the Euler tour of the tree after the last occurrence of v

Structure:

- Break Euler tour into blocks of length $b = \frac{1}{2}\log_2 n$
- Within block, depths differ by ± 1; label each block by its pattern of $\pm$ choices, a $(b-1)$-bit binary number
- Precompute tables for queries with answer in same block
- Store jump tables only at the last vertex of each block

Query:

- Use table to find answer in v's block (if it is there)
- If not, vertex u at block end has same level-d ancestor as v
- Use the jump table stored at u to find an ancestor w high enough that we can use the extended paths for w

Summary

Summary

- Representation of trees and binary trees with $2n$ bits
- Blocking and table lookup strategy for saving logarithmic factors in the space bounds for many data structures
- Common ancestor problem and its applications to shortest paths and bandwidth maximization
- Equivalence between common ancestors and range minima
- Common ancestors in $O(n)$ space and $O(1)$ query time
- Maintaining order in a list in $O(1)$ amortized time
- Level ancestors in $O(n)$ space and $O(1)$ query time