Grammar Compression and Random Access

- Grammar Compression
- Random Access

Philip Bille

Grammar Compression

- Statistical compression.
  - Huffman, arithmetic encoding,...
- Dictionary compression.
  - Lempel-Ziv, ...
- Grammar compression.
  - Repair, sequitur, greedy, bisection, …
- Kolmogorov complexity.

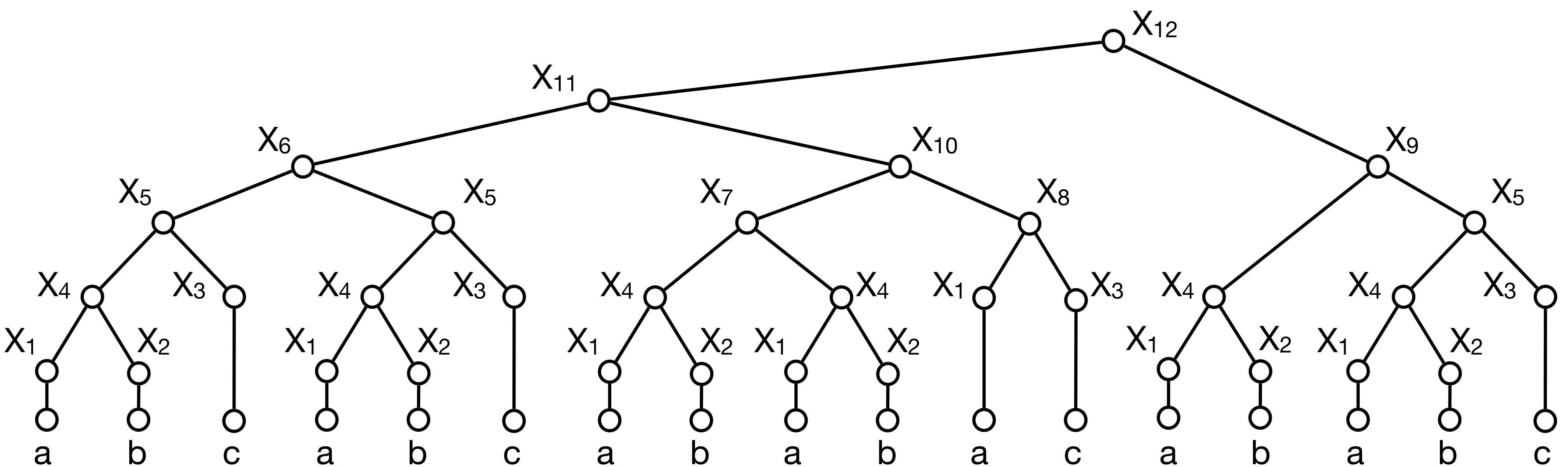
Grammar Compression

- Grammar compression. Encode string S as an grammar G that generates S.
- Straight-line program. Assume G is a straight-line program.
  - G is acyclic.
  - Each production in G is either $X_i \rightarrow X_j X_k$ or $X_i \rightarrow \tau$.
- Encoding. Re-pair, bisection, greedy, …
- Decoding. Unfold productions top-down.

$X_{12} \rightarrow X_{11}X_9$
$X_{11} \rightarrow X_6X_{10}$
$X_{10} \rightarrow X_7X_8$
$X_9 \rightarrow X_4X_5$
$X_8 \rightarrow X_1X_3$

$X_6 \rightarrow X_5X_5$
$X_5 \rightarrow X_4X_3$
$X_4 \rightarrow X_1X_2$
$X_3 \rightarrow c$
$X_2 \rightarrow b$
$X_1 \rightarrow a$

abcabcababacababc



Grammar Compression

- Re-pair compression [Larsson and Moffat 2000].
  - Start with string S.
  - Replace a most frequent pair ab by new character X_i. Add production $X_i \rightarrow ab$.
  - Repeat until string is a single character.

X_9	
X_8X_6	$X_9 \rightarrow X_8X_6$
$X_3X_7X_6$	$X_8 \rightarrow X_3X_7$
$X_3X_4X_5X_6$	$X_7 \rightarrow X_4X_5$
$X_3X_4X_5X_1X_2$	$X_6 \rightarrow X_1X_2$
$X_3X_4acX_1X_2$	$X_5 \rightarrow ac$
$X_3X_1X_1acX_1X_2$	$X_4 \rightarrow X_1X_1$
$X_2X_2X_1X_1acX_1X_2$	$X_3 \rightarrow X_2X_2$
$X_1cX_1cX_1X_1acX_1X_1c$	$X_2 \rightarrow X_1c$
abcabcabababacababc	$X_1 \rightarrow ab$

Grammar Compression

- Grammar compression properties.
  - Many dictionary schemes can be viewed as grammar compressors.
  - Smallest grammar is NP-hard.
  - LZ77 is lower bound on the smallest grammar.
  - LZ77 can be converted to grammar with blowup by logarithmic factor.
  - Grammar very useful for compressed computation.

Random Access

- Random Access Problem. Represent grammar G of size n generating string S of length N to support
  - access(i): return S[i]

$X_{12} \rightarrow X_{11}X_9$	$X_6 \rightarrow X_5X_5$
$X_{11} \rightarrow X_6X_{10}$	$X_5 \rightarrow X_4X_3$
$X_{10} \rightarrow X_7X_8$	$X_4 \rightarrow X_1X_2$
$X_9 \rightarrow X_4X_5$	$X_3 \rightarrow c$
$X_8 \rightarrow X_1X_3$	$X_2 \rightarrow b$
	$X_1 \rightarrow a$

abcabcabababacababc

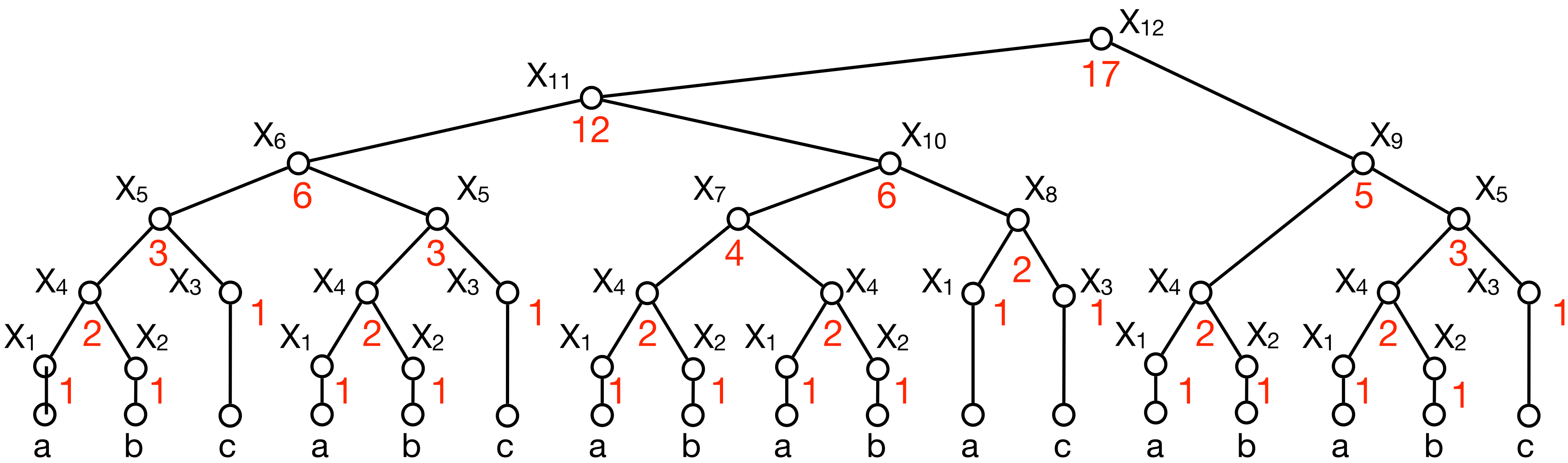
Random Access

- Applications.
  - Most basic computational task on compressed data.
  - Component in most algorithms and data structures that work directly on compressed data (compressed computing).
  - Interesting selection of elegant and useful data structural techniques.

Random Access

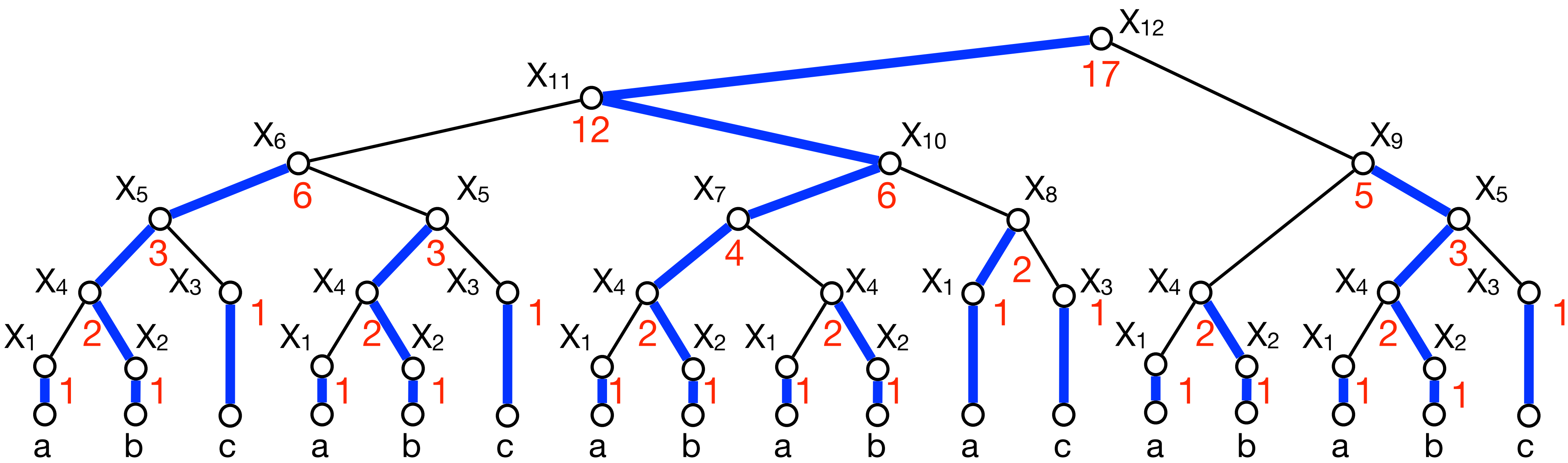
- Goal. Random access with O(n) space O(log N) query time.
- Solution in 4 steps.
    - Top-down search. Slow but only linear space.
    - Heavy-path decompositions. Almost fast but too much space.
    - Heavy-path redundancy. Almost fast with linear space.
    - Interval-biased search. Fast and linear space.

Solution 1: Top Down Search



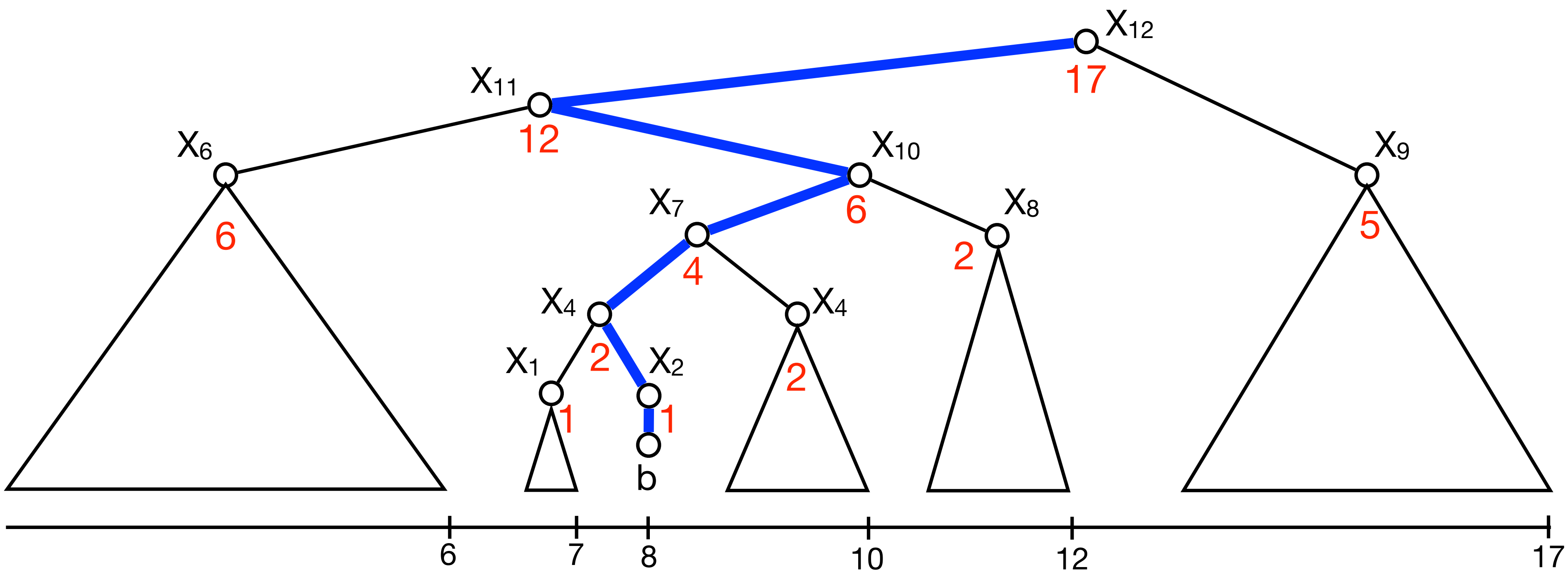
- Data structure. Store size of string generated by each node.
- Access(x): Top-down search for x.
- Time. O(h) = O(n)
- Space. O(n)

Solution 2: Heavy Path Decomposition



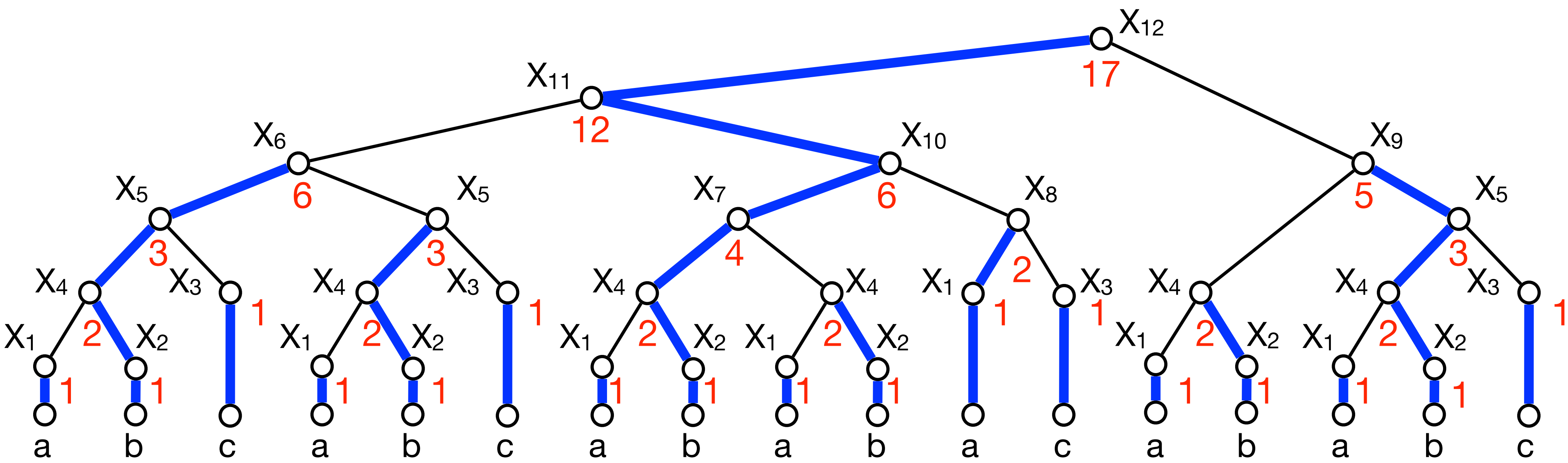
- Heavy-path decomposition.
  - Start at root. Choose a child of maximum size repeatedly until we reach leaf.
  - Repeat for subtrees hanging off tree.
- Lemma. O(log N) heavy paths on any root-to-leaf path.
- Proof: Size decrease by at least half on each light edge.

Solution 2: Heavy Path Decomposition

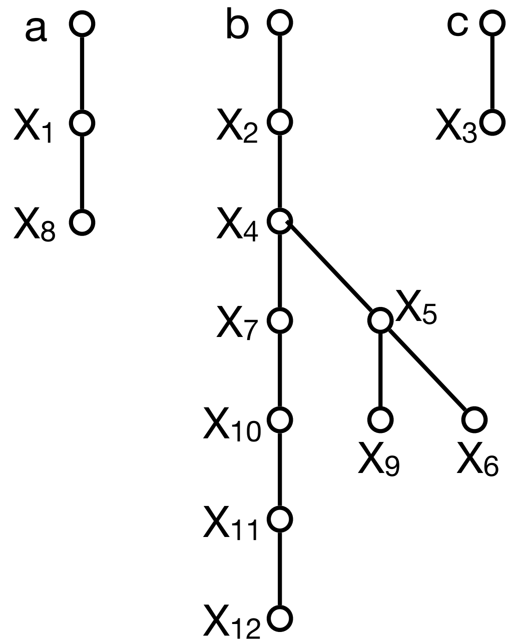


- Data structure. For each heavy path store list of values + char at end of heavy path.
- Access(x): Predecessor search on each heavy-path on root-to-leaf path.
- Time. O(log log N log N)
- Space. $O(n^2)$

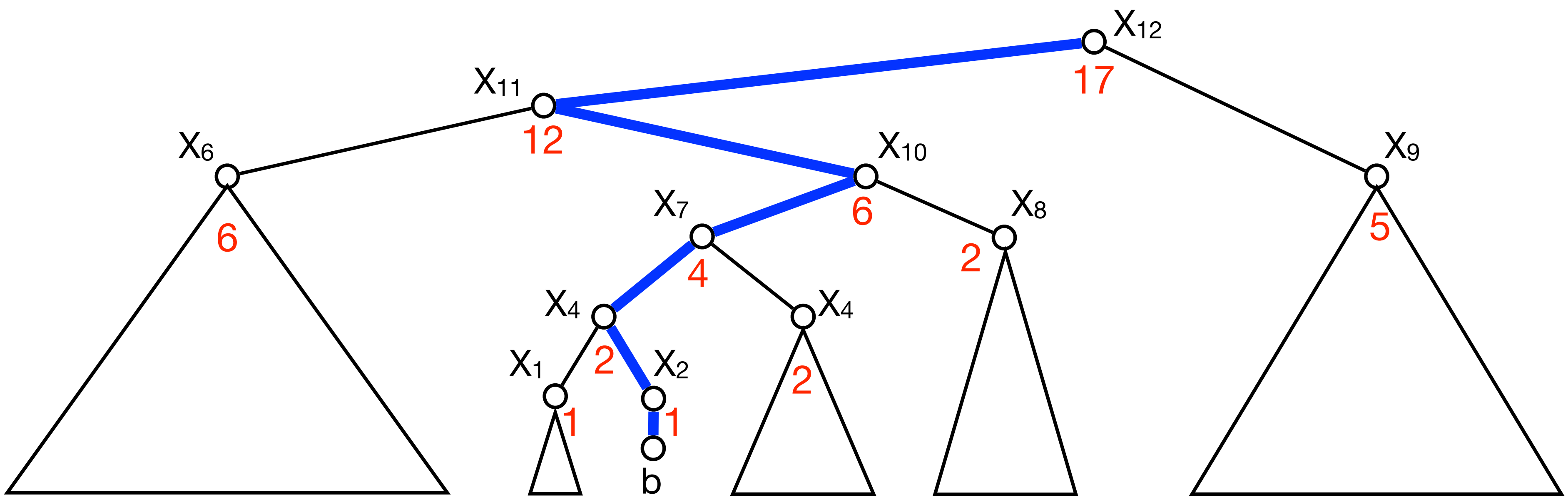
Solution 3: Heavy-Path Redundancy



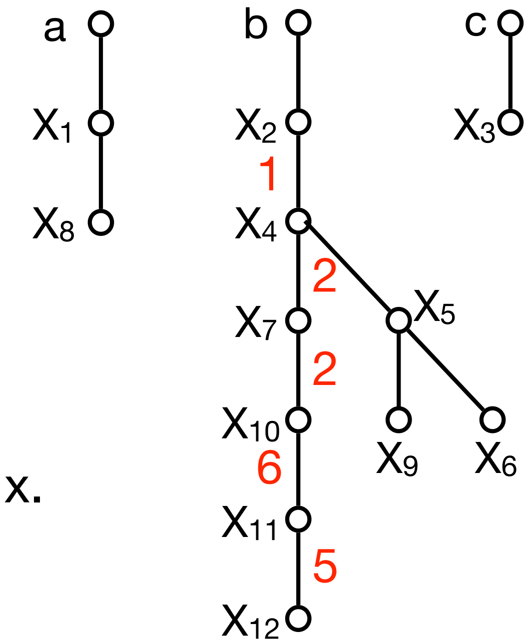
- Idea. Exploit overlaps in heavy-paths to get compact representation.
- Heavy-path suffix forest.
    - Tree of all suffixes of heavy-paths.
    - v is a parent of u iff u is heavy child of v.
    - Only n nodes.



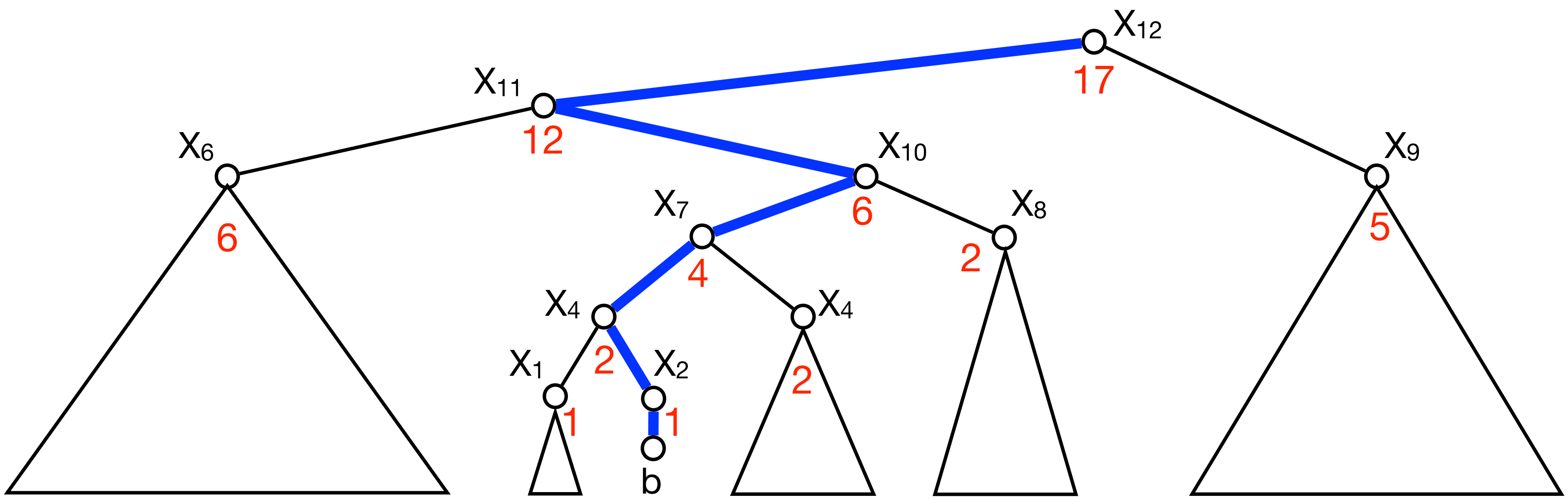
Solution 3: Heavy-Path Redundancy



- Predecessor on heavy path.
  - Weighted ancestor problem on heavy path suffix forest.
  - Weigh each edge with size of off-path subtree.
  - Keep left and right edge weights separate.
  - Search for x to the left = closest ancestor of distance ≥ x.
  - Similar for search to the right.

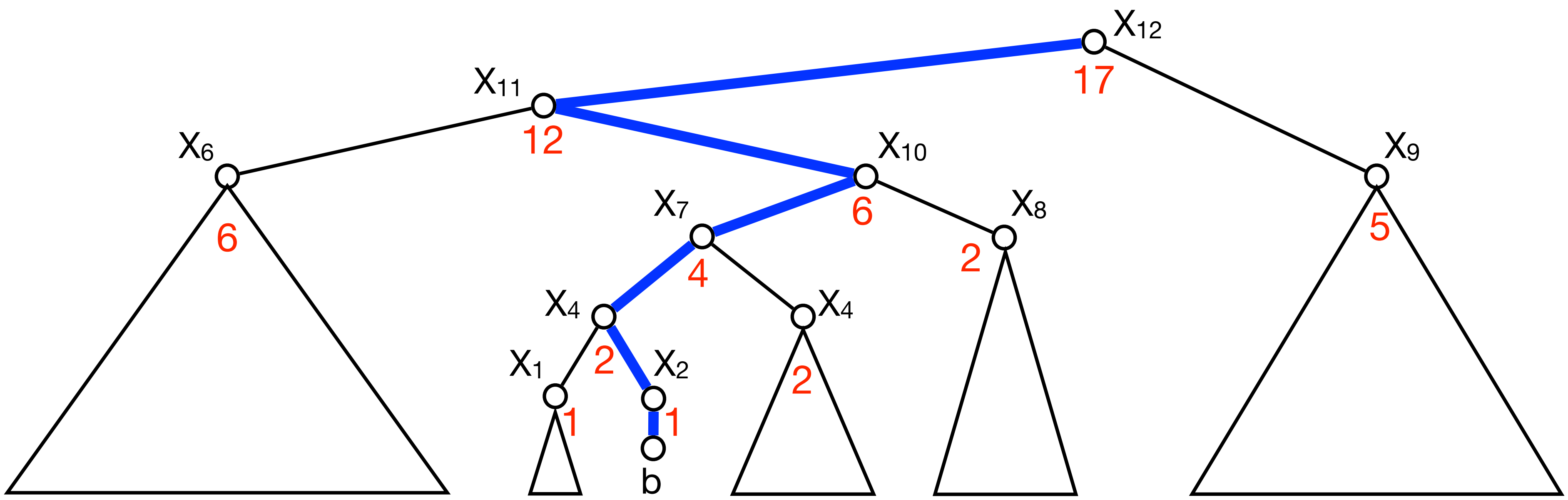


Solution 3: Heavy-Path Redundancy



- Lemma. For a tree with n nodes and edge weights from universe [0…N] we can solve the weighted ancestor problem in O(n) space and O(log log N) time.
- Access(x): Weighted ancestor query on each heavy-path on root-to-leaf path.
- Time. O(log log N log N)
- Space. O(n)

Solution 4: Interval Biased Search



- Lemma. For a tree with n nodes and edge weights from universe [0…N] we can solve the weighted ancestor problem in O(n) space and O(log (N/S)) time, where S is size of subtree hanging off path.
- Access(x): Weighted ancestor query on each heavy path on root to leaf path.
- Time. $\log (N/S_1) + \log (S_1/S_2) + \log (S_2/S_3) + \log (S_3/S_4) + ... + O(1)$
- $= \log N - \log S_1 + \log S_1 - \log S_2 + \log S_2 - \log S_3 + \log S_3 + ... + O(1)$
- $= O(\log N)$

Random Access

	Space	Time
Top down search	O(n)	O(h) = O(n)
Heavy path decomposition	$O(n^2)$	O(log N log log N)
Heavy path redundancy	O(n)	O(log N log log N)
Interval biased search	O(n)	O(log N)
Lower bound	$n \log^{O(1)} N$	$\Omega(\log^{1-\varepsilon} N)$

Grammar Compression and Random access

- Grammar Compression
- Random Access