Linear BVH construction on GPUs using parent search

Jackson Lee

Basic concept based on:
"Fast BVH Construction on GPUs" by C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha [Lauterbach et al. 2009]

"Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d trees" by T. Karras [Karras 2012]

March 2014

Algorithm overview

1) Find merged AABB [1]
2) Assign Morton codes [1]
3) Sort Morton codes [1]
4) Build binary radix tree [2]* ← focus of these slides
5) Assign AABBs for internal nodes [2]

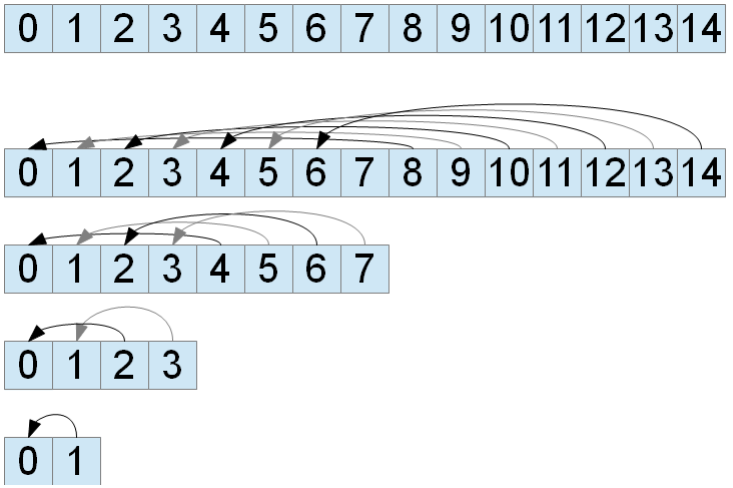
[1] More details in [Lauterbach et al. 2009]

[2] More details in [Karras 2012]

*Exact algorithm presented here is not covered in the papers

Find merged AABB

- Use parallel reduction to find the merged AABB of all leaf nodes




```
b3AabbCL aabb = out_mergedAabb[aabbIndex];
b3AabbCL otherAabb = out_mergedAabb[otherAabbIndex];

b3AabbCL mergedAabb;
mergedAabb.m_min = b3Min(aabb.m_min, otherAabb.m_min);
mergedAabb.m_max = b3Max(aabb.m_max, otherAabb.m_max);
out_mergedAabb[aabbIndex] = mergedAabb;
```

Assign Morton codes

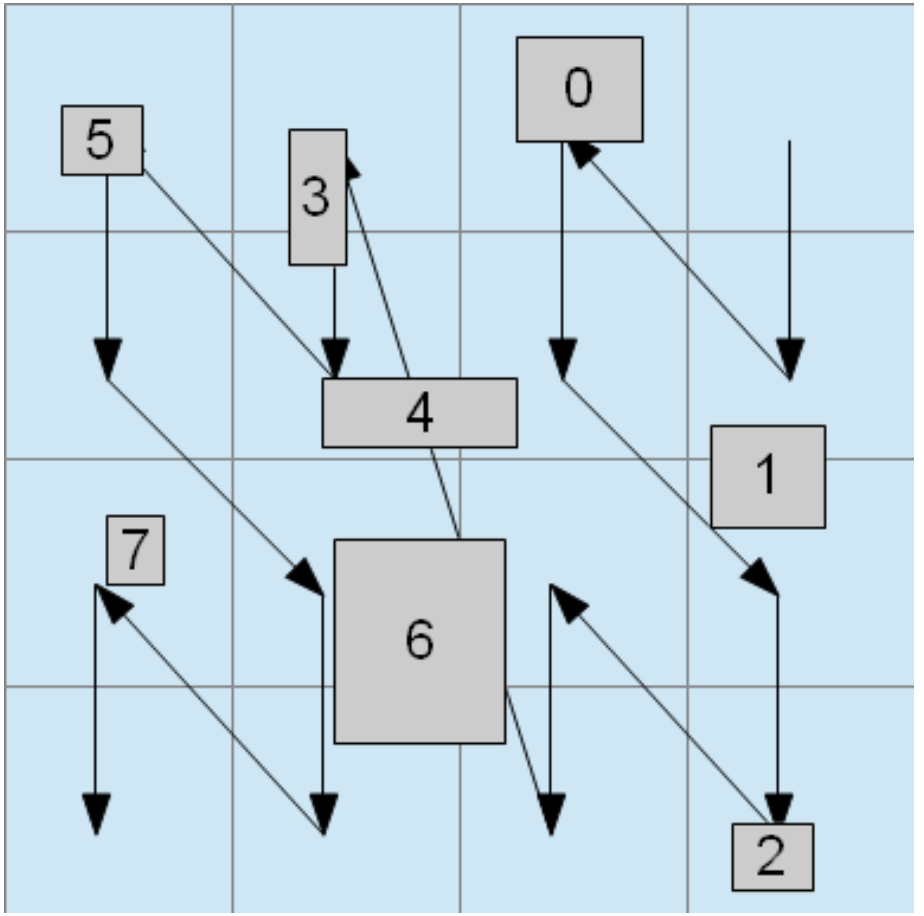
- Assign grid coordinates to each AABB using its center
  - Use min, max of merged AABB as grid extents
  - 1024 cells(10 bits) in each dimension
- Convert grid coordinates into Morton codes (z-curve)

$$\{ x_3x_2x_1x_0,\ y_3y_2y_1y_0,\ z_3z_2z_1z_0 \} \rightarrow \{ z_3y_3x_3z_2y_2x_2z_1y_1x_1z_0y_0x_0 \}$$

$$\{ 1101, 1011, 0001 \} \rightarrow \{\ 011\ 001\ 010\ 111\ \}$$

Sort Morton codes

- Use radix sort
- Objects that are closer in space (3D) become closer in memory (1D)



Build binary radix tree

- Connections between the BVH nodes are established by constructing a binary radix tree
- http://en.wikipedia.org/wiki/Radix_tree
  - 'Binary' in binary radix tree means that the tree only contains 0 and 1

Build binary radix tree

- Why a binary radix tree?
  - A radix tree organizes its nodes by a common prefix
  - Combining this characteristic with sorted Morton codes is equivalent to splitting the nodes along an axis-aligned plane

Common prefix

- For example, the common prefix of

  1110 1001 1011 0100, and

  1110 1011 0011 0100 is 1110 10 (length 6)
- Definition: contiguous shared bits of 2 bit strings starting from the most significant bit
- Length of the common prefix is also important
  - Lower common prefixes are closer to the root of the binary radix tree, and higher common prefixes are closer to the leaves

Common prefix

- More examples

  0010 1101 and

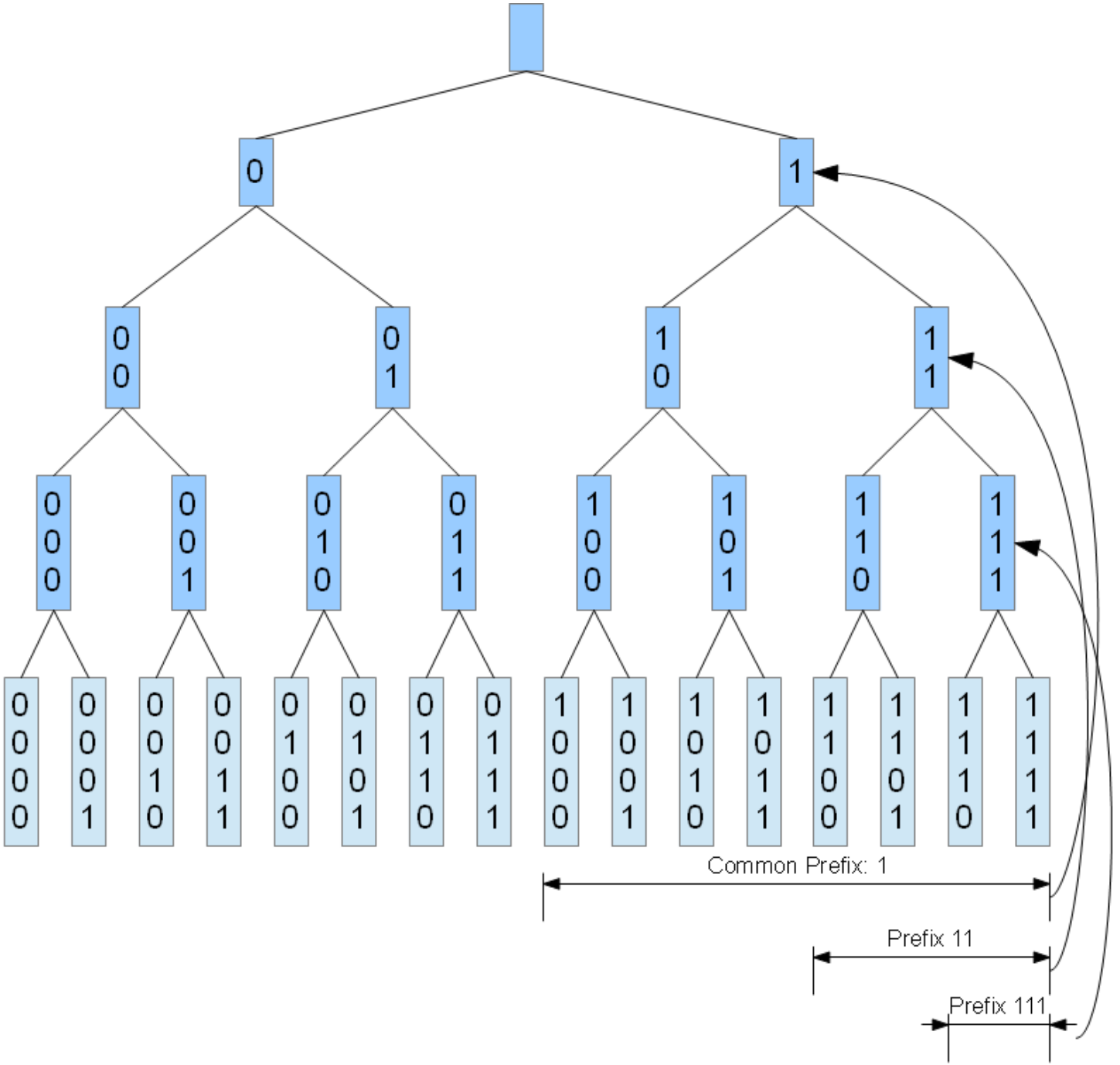
  1010 1101 has no common prefix (length 0)

  1010 0110 and

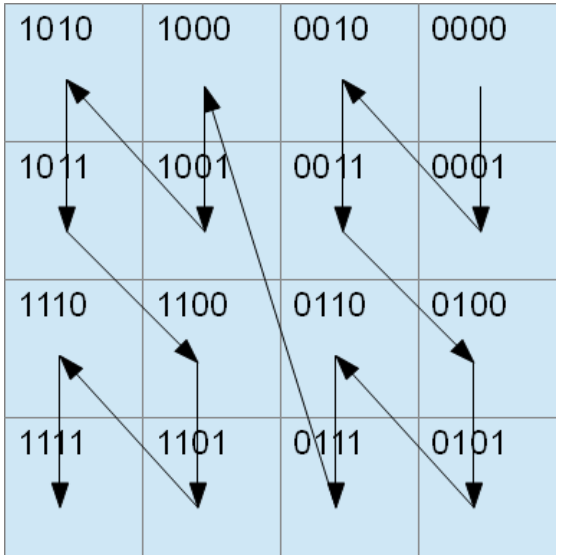
  1011 1010 is 101 (length 3)

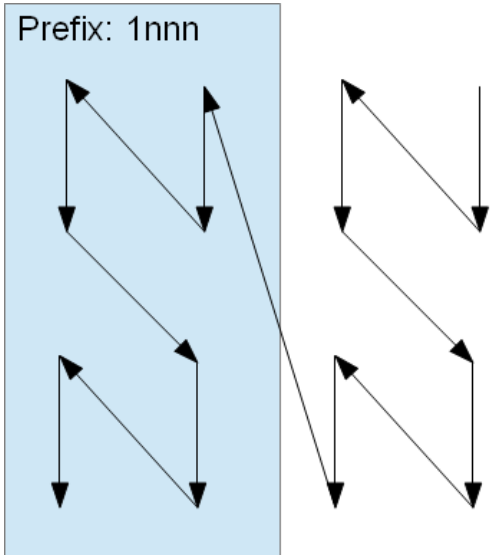
  1110 0100 and 1110 0100 is 1110 0100 (length 8)

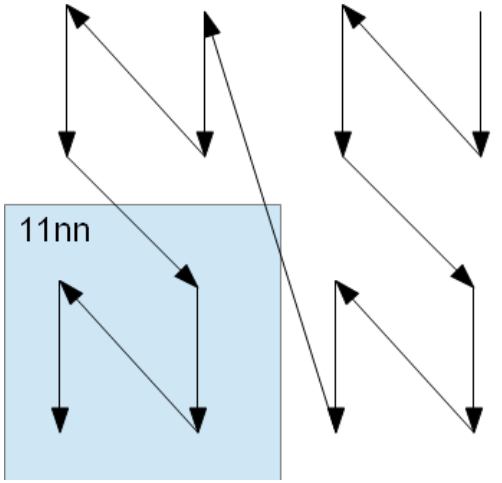
The child nodes of each internal node in a binary radix tree share a common prefix

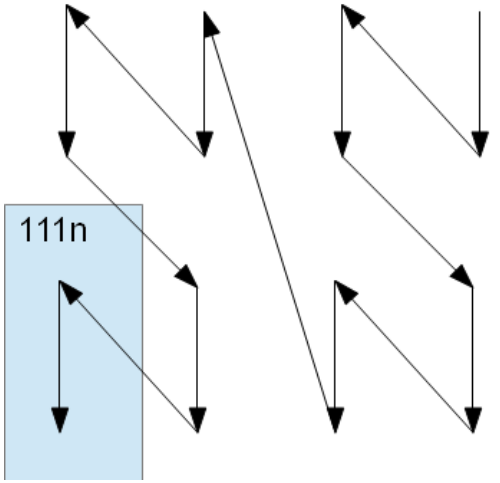


Forming a binary radix tree with Morton codes splits along an axis-aligned plane









Build binary radix tree

- How to construct?
  - A recursive method that splits the leaf nodes by moving from lower to higher common prefixes is the natural solution, but maps poorly to GPUs
  - [Karras 2012] shows that combining sorted Morton codes with a specific layout for the internal nodes allows efficient construction on the GPU

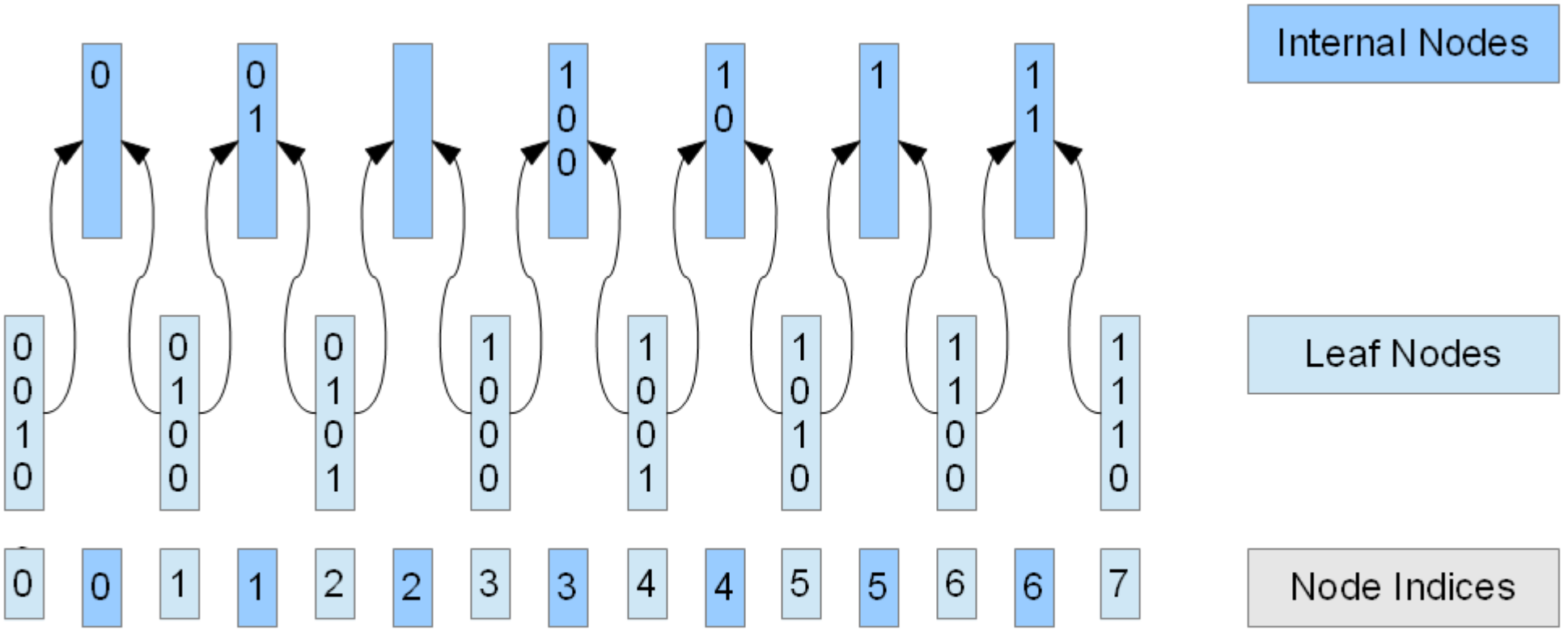
Build binary radix tree

- Following [Karras 2012], we use a specific layout for the tree, but our method has considerable differences
  - Search for the parent of each node instead of searching for the children
  - Position of each internal node does not always coincide with the beginning or end of its range
- Similar computational complexity and performance characteristics
  - 2 binary searches

Build binary radix tree

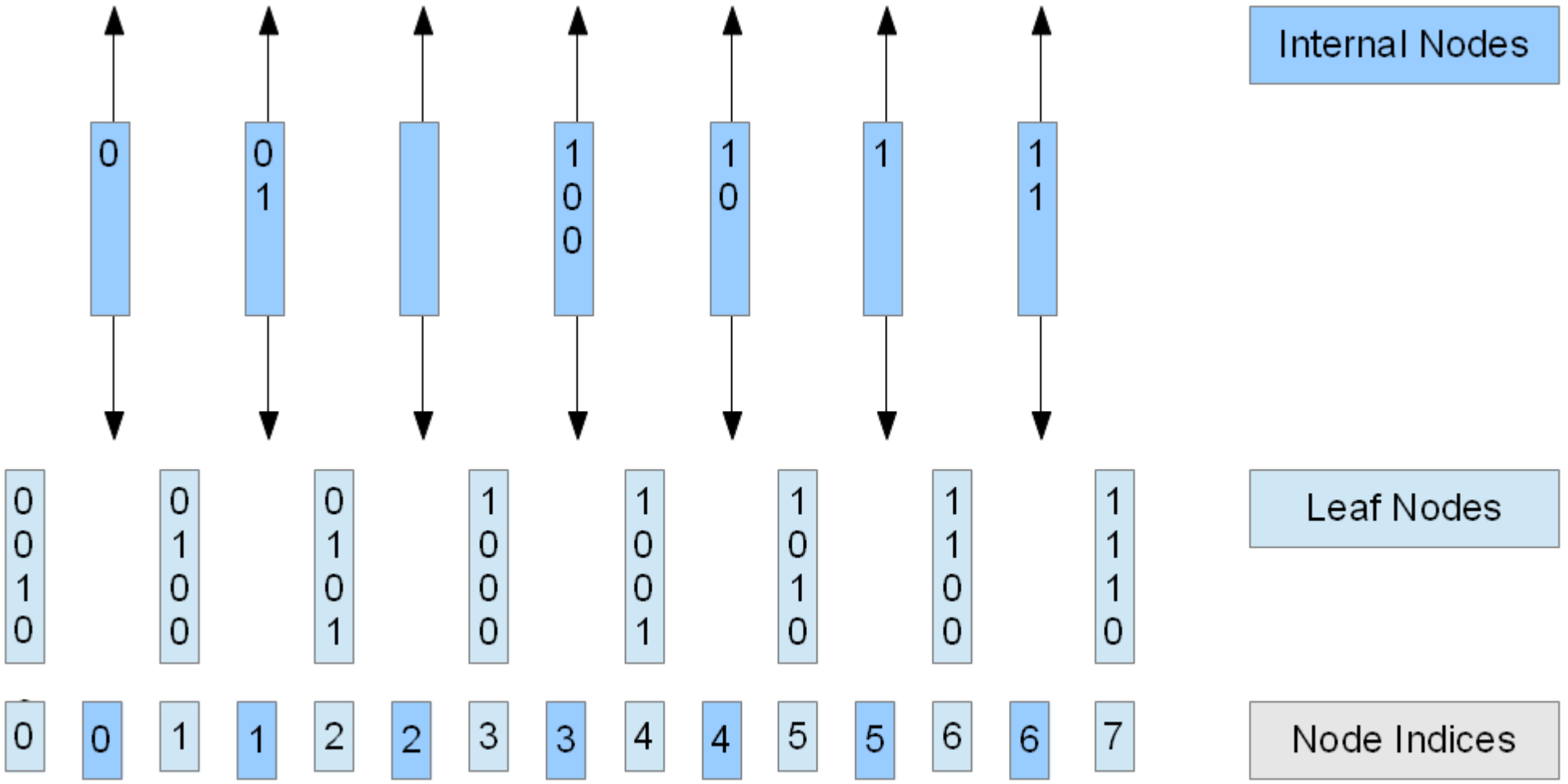
- Tree layout: place internal nodes in between leaf nodes
  - For a binary tree there are always N – 1 internal nodes, where N is the number of leaf nodes
- Compute the common prefix and common prefix length at each internal node by looking at its leaf node neighbors
- The problem to solve at this point is: how do we form connections among the internal nodes so that the lowest common prefixes are at the top, and highest are at the bottom?

Tree Layout



- Internal nodes are placed in between leaf nodes, and are assigned a common prefix by comparing their neighboring leaf nodes

Tree Layout

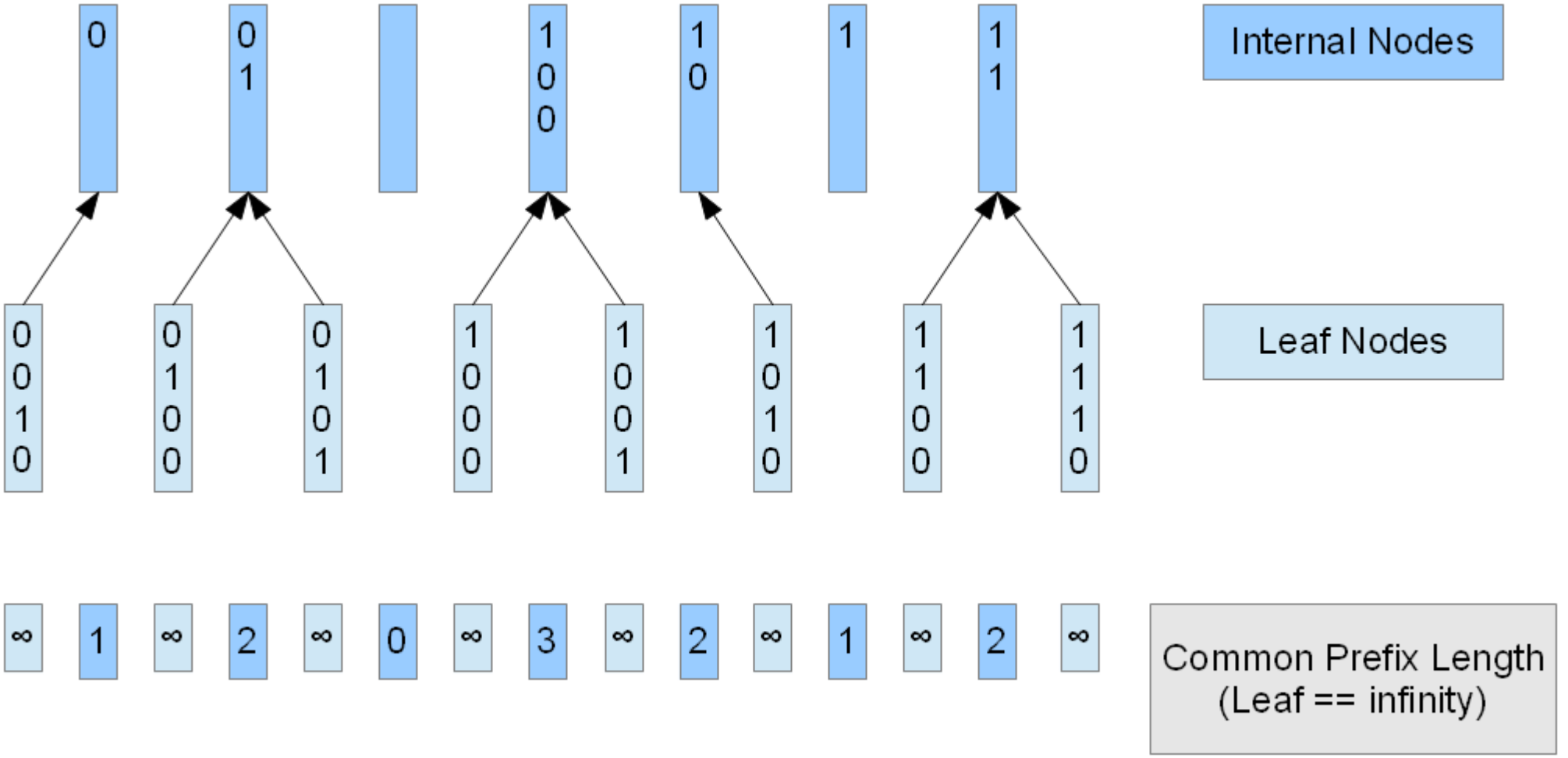


- Problem becomes one of determining how low or high in the hierarchy each internal node is

Build binary radix tree

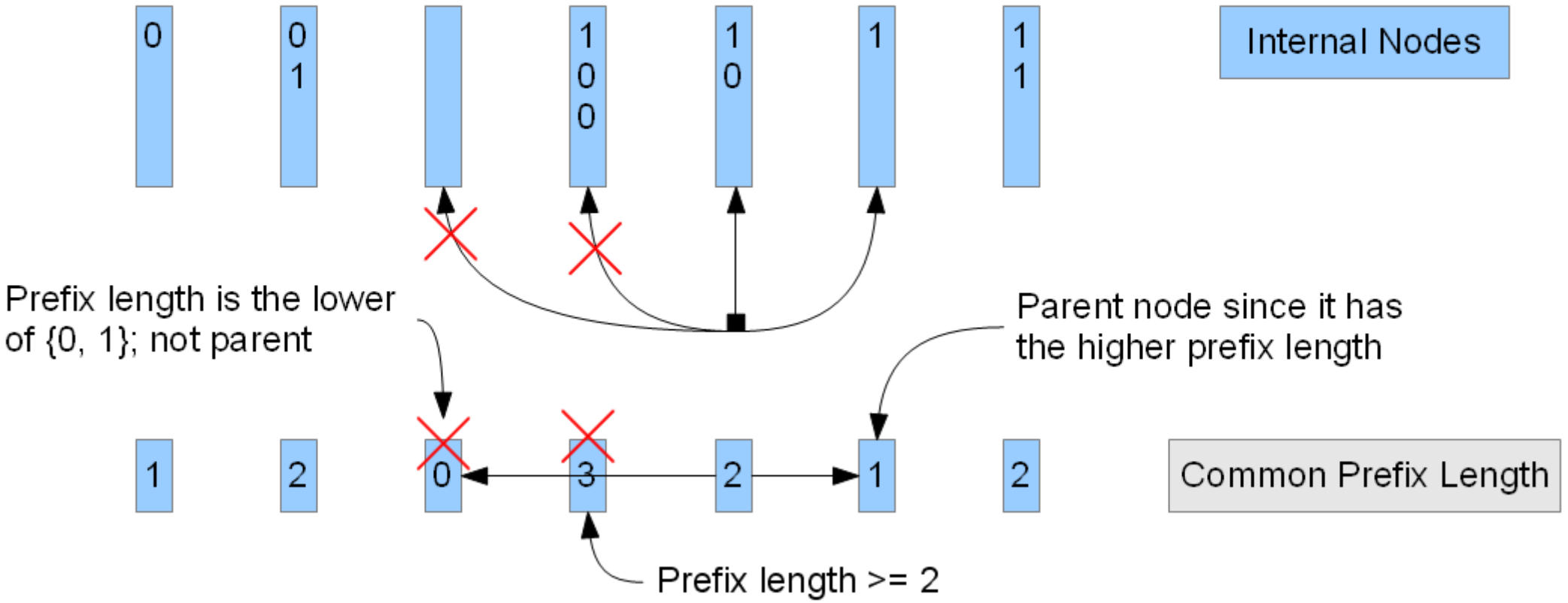
- Key observation: the parent node of any node is either the nearest left node with a lower prefix length, or the nearest right node with a lower prefix length
  - Of these 2 nodes, the parent is the node with the higher prefix length
- Treat leaf nodes as having an infinitely high prefix length (128 in the implementation)
  - Since the leaf nodes always have a higher prefix length than the internal nodes, their parents can be computed by looking at the left and right internal nodes that are directly adjacent

Finding parent nodes for leaf nodes



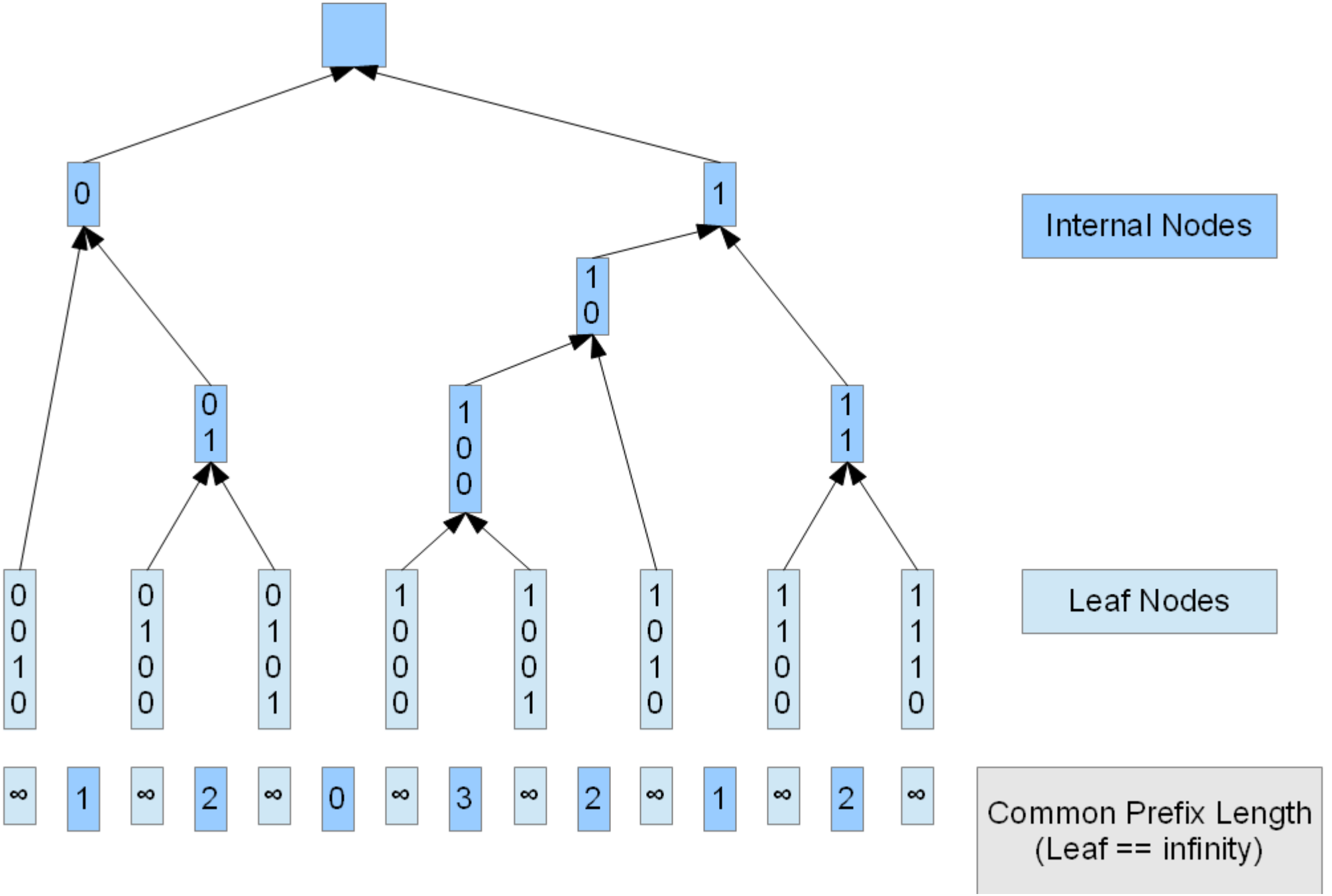
- Each leaf node checks the nearest left and right internal node and selects the one with the higher common prefix length as its parent

Finding parent nodes for internal nodes

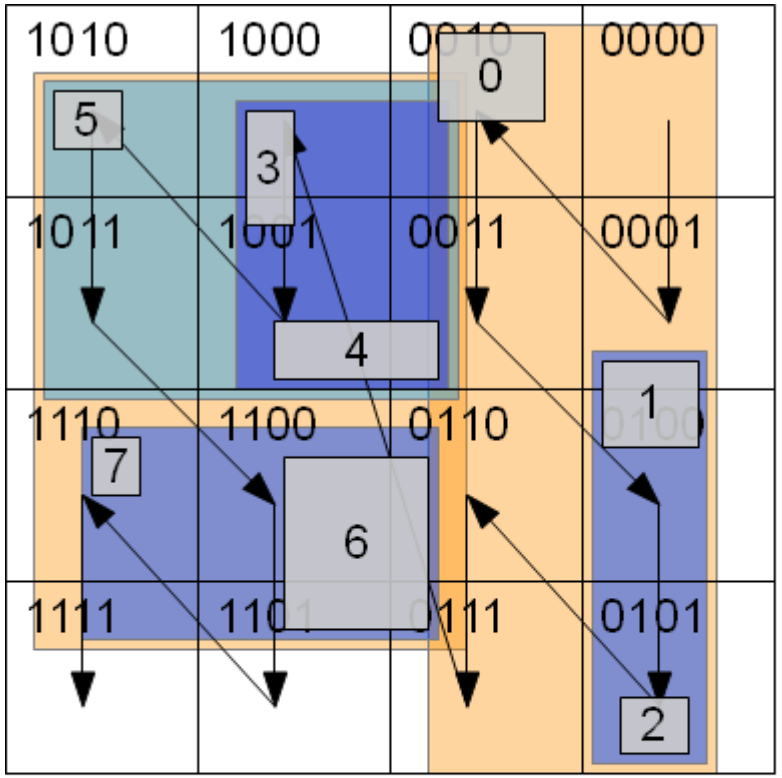


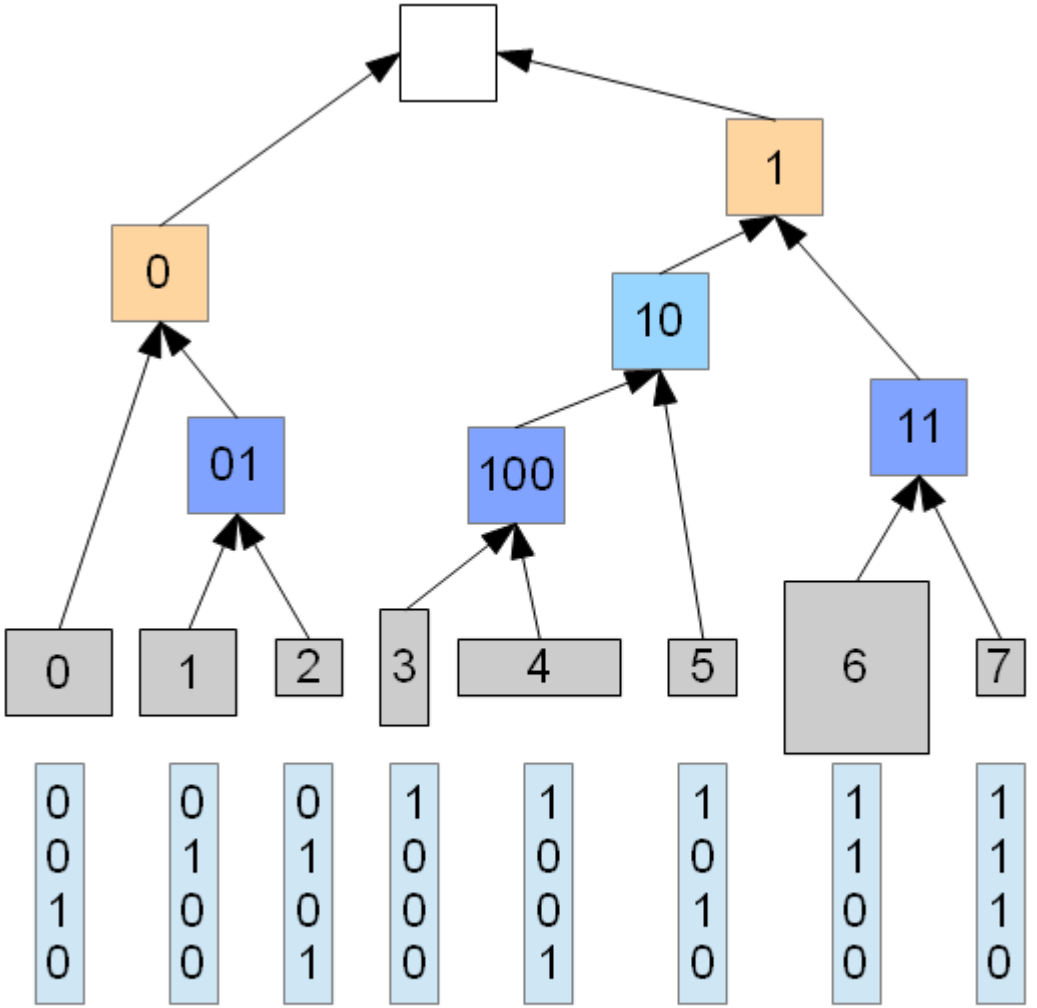
- The potential parent nodes of an internal node are the nearest nodes to the left and right with a lower common prefix length
- Of these 2 nodes, the actual parent is the node with the higher common prefix length

Resulting Tree



Resulting Linear BVH

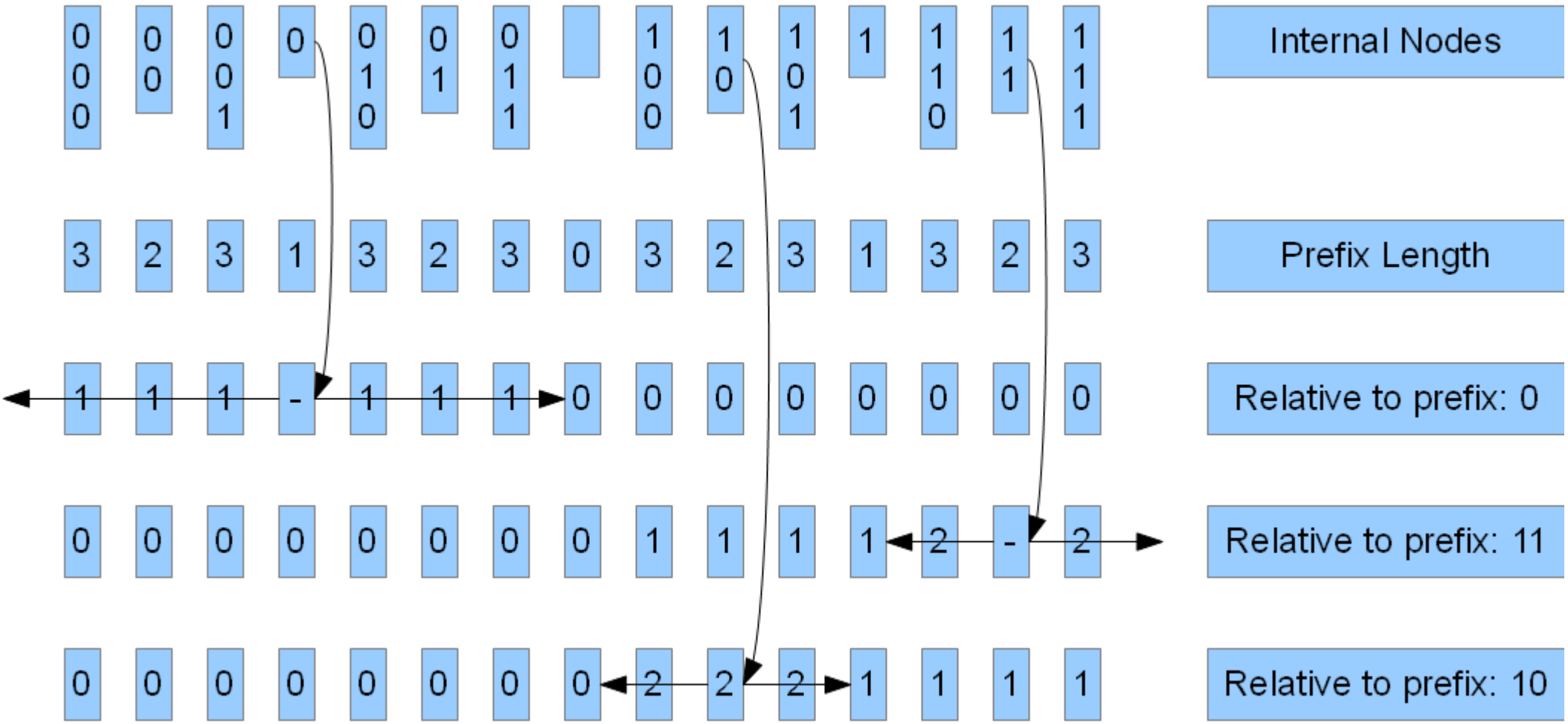




Build binary radix tree

- Problem: how to find the 2 nearest nodes with lower common prefix for internal nodes?
  - Common prefix lengths are not sorted; sorting is expensive and also does not provide much additional information
- Solution: compute the common prefixes of all internal nodes relative to a single internal node
  - If 2 prefixes are of different length, calculate the prefix using the shorter length
  - Allows finding the 2 nearest nodes with 2 binary searches; one to the left and one to the right
  - Does not need to be stored, compute during binary search iteration
- Now that we have a parent node for each node, it is trivial to assign the child nodes of each internal node

Relative common prefix

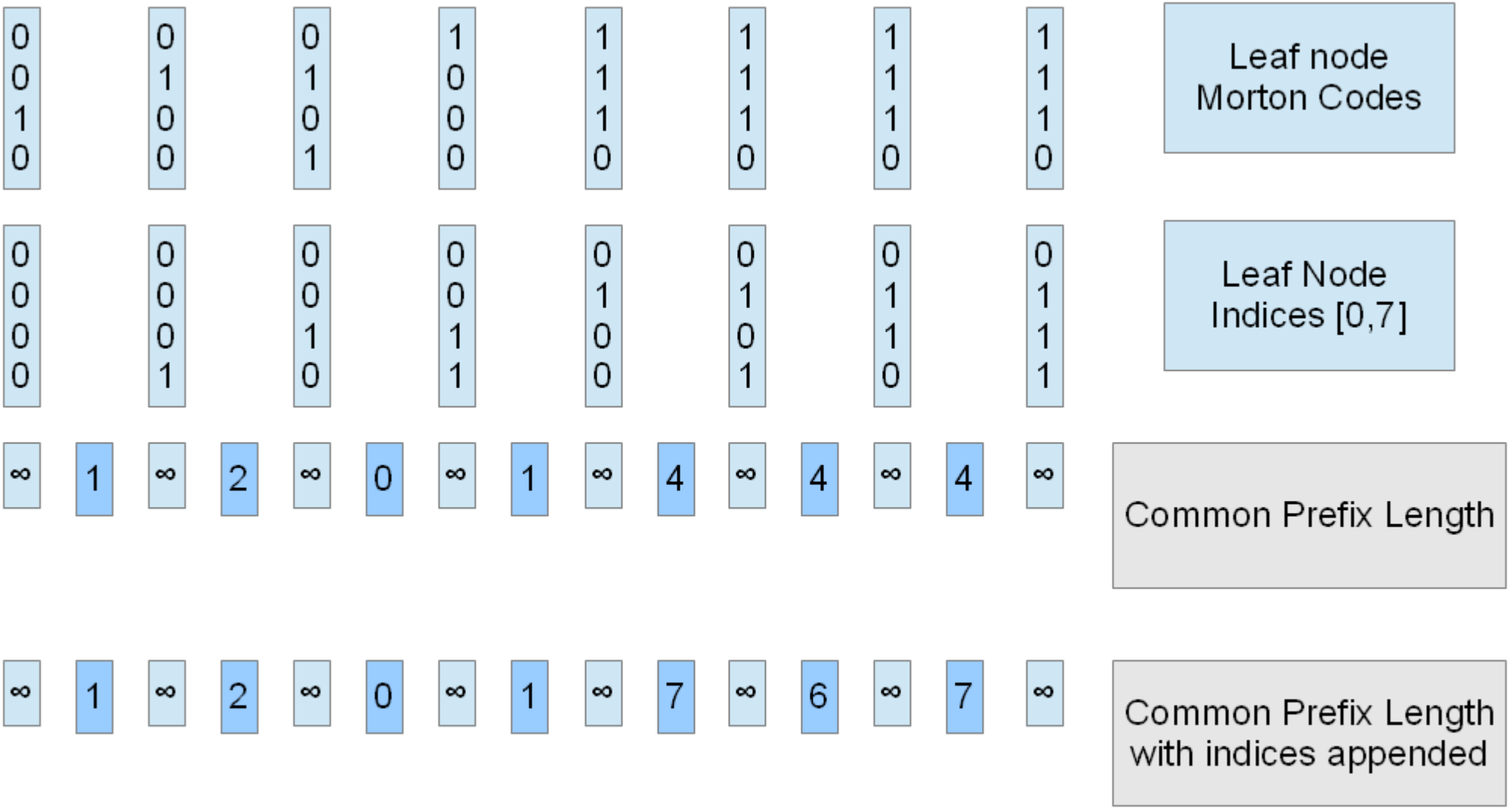


- Computing the common prefixes of all nodes relative to a single node makes it possible to use a binary search to find the nearest left and right nodes with lower prefixes

Duplicate common prefixes

- Algorithm requires that the Morton codes are sorted, and that there are no duplicate common prefixes
- Use same method as in [Karras 2012]
  - Append leaf index to each Morton code
  - Common prefixes are still in sorted order
- In contrast to [Karras 2012], the combined prefix has to be stored since we compare the internal node prefixes
- Additionally, since internal node prefixes are compared, it does not matter if there are 2 leaf nodes with the same prefix; only if there are 2 internal nodes with the same prefix
  - Occurs when there are 3+ leaf nodes with the same Morton code

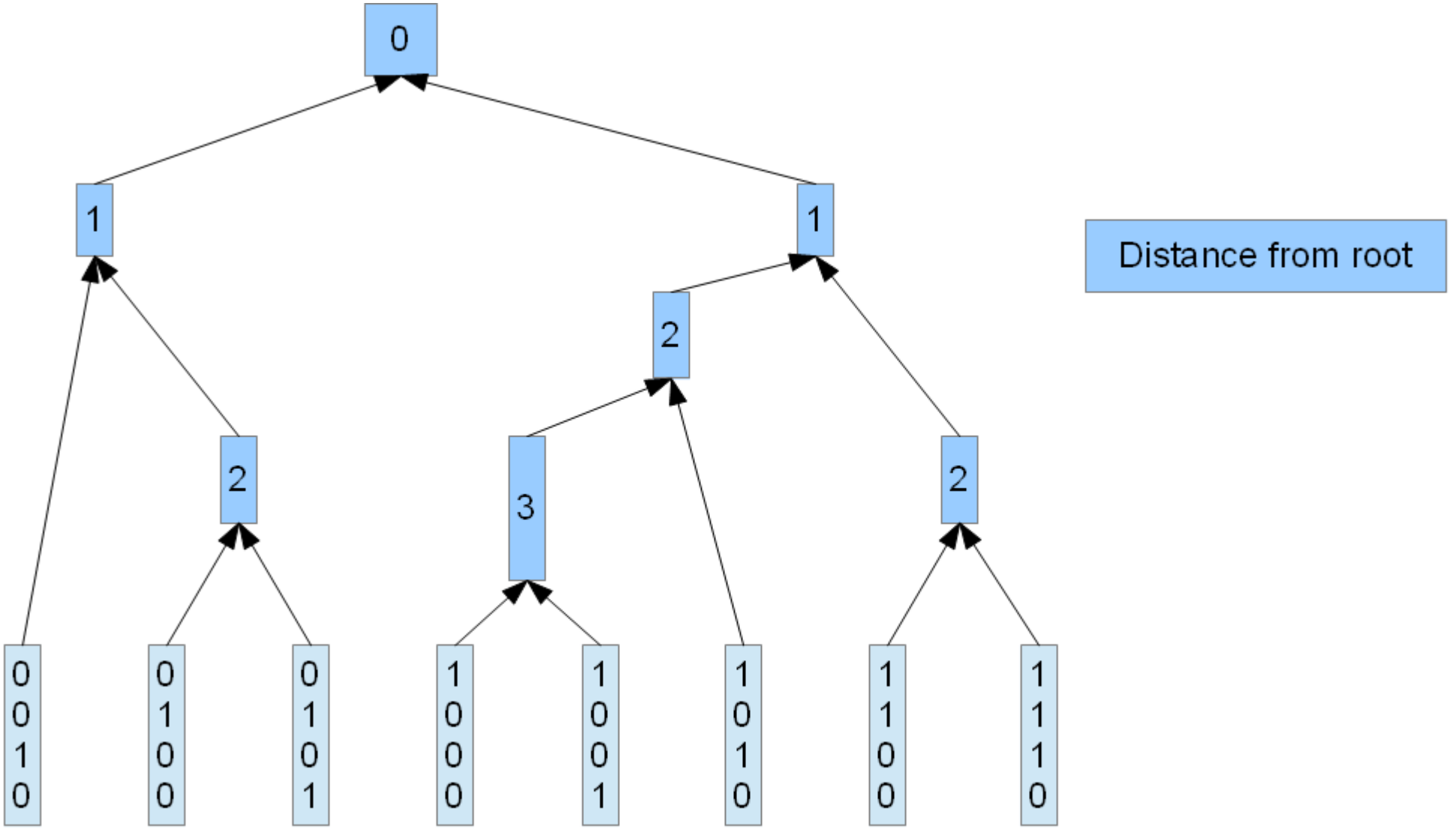
Handling duplicate prefixes



Assign internal node AABBs

- Find distance from root for each node
  - Start from leaves, traverse towards root
- Launch kernels, 1 kernel launch per distance
  - Each internal node merges the AABBs of its child nodes
  - OpenCL does not have a mechanism for ensuring that writes in 1 workgroup are visible to another workgroup within a kernel

Assigning internal node AABBs



- In this example, 4 kernel launches are needed to set the AABBs for all internal nodes
- Kernels are launched from highest to lowest distances

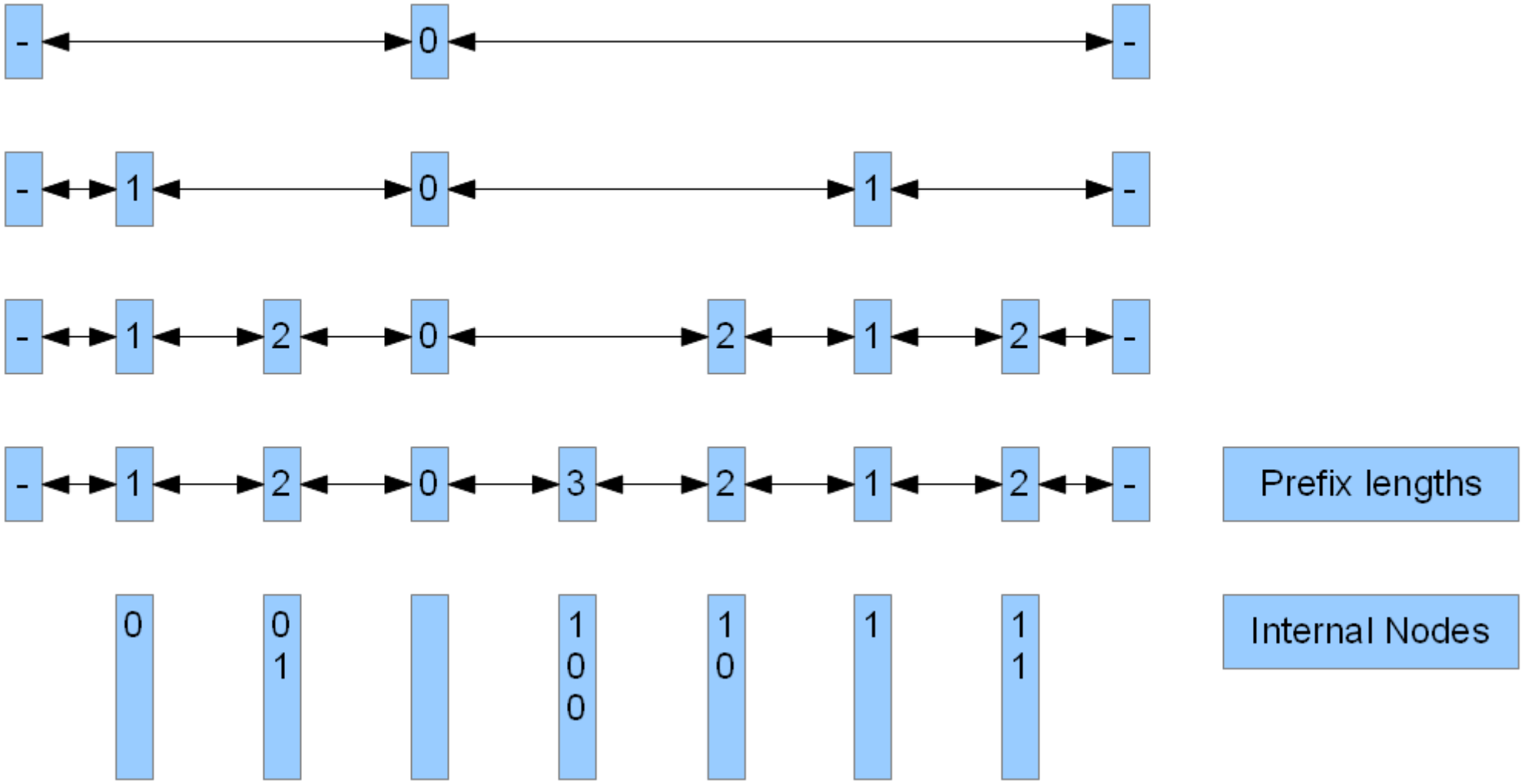
Summary / Implementation details

- 2 kernels to assign Morton code
  - Parallel reduction to find grid extent
  - For each leaf node, compute grid coordinates and convert into Morton code
- Use radix sort to sort the Morton codes
- 3 kernels to build binary radix tree
  - For each internal node, calculate its common prefix by looking at the 2 neighboring leaf nodes
  - For each leaf node, find its parent by comparing the nearest 2 internal nodes
  - For each internal node, find its parent with 2 binary searches
- 2 kernels to set AABB
  - For each internal node, traverse the tree to find the distance to the root
  - (1 kernel launch per distance) For each internal node, check its 2 child nodes to get its AABB

Other notes

- Could also construct the tree using pointers
  - Same common prefix length never repeated twice if no duplicates
  - This property is maintained if prefixes are removed in passes, starting from the highest prefixes and proceeding to the lowest prefixes
  - Means that a linked list can be used
- Remove all nodes with a certain common prefix per kernel launch
  - Up to 32-64 kernel launches
- Could also use the atomic traversal of [Karras 2012] and update the linked list after both child nodes access a node
  - However, this method is potentially unstable since OpenCL does not guarantee that the writes are visible to other workgroups
- Slightly slower (for instance, 2.0ms compared to 1.5ms for 27000 objects)

Construction using pointers



- By using a linked list and removing the nodes from highest to lowest common prefixes, the nearest nodes with lower prefixes are easily accessible