Lecture 13:
Trees

Modern Parallel Computing
John Owens
EEC 289Q, UC Davis, Winter 2018

Credits

- Based on a lecture by Kerry Seitz

- Tero Karras
  - "Thinking Parallel" blog posts (https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-i-collision-detection-gpu/)
  - "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees (https://research.nvidia.com/publication/maximizing-parallelism-construction-bvhs-octrees-and-k-d-trees)

Overview

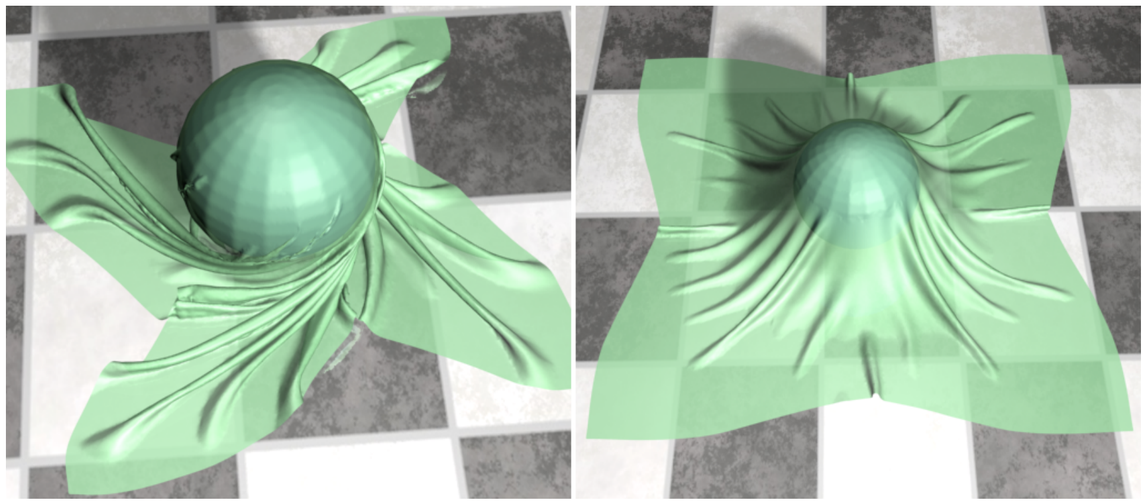
- Motivating Application: Collision Detection
- Bounding Volume Hierarchy (BVH)
- Tree Traversal
- Tree Construction

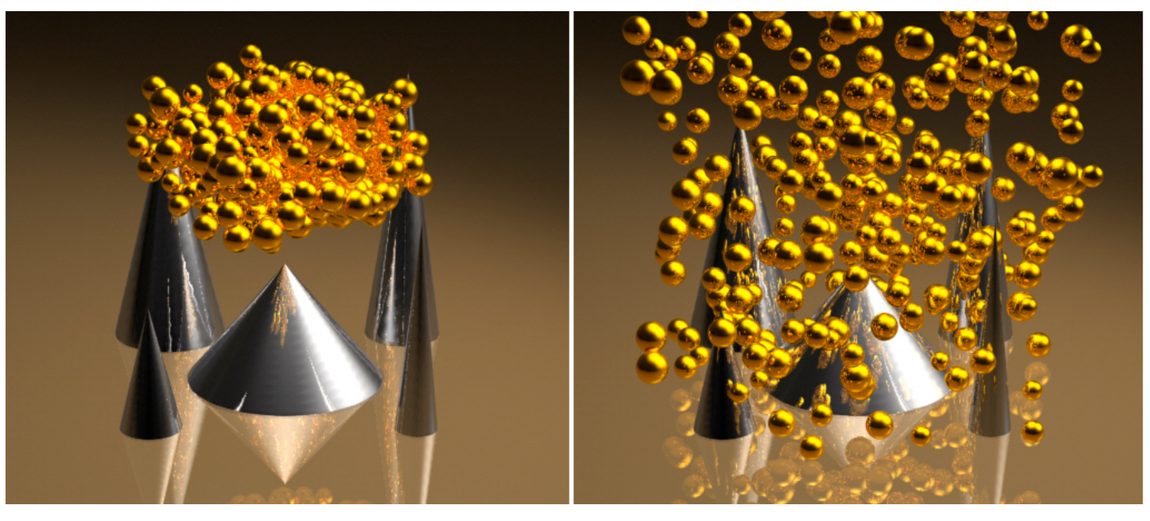
Collision Detection

- Query: Does object X intersect with anything in the scene?

- Difficulty: Both X and the scene are dynamic (constantly changing)

- Goal: Data structure that makes this query efficient (in parallel)





Images from HPCCD: Hybrid Parallel Continuous Collision Detection, Kim et al., Pacific Graphics 2009

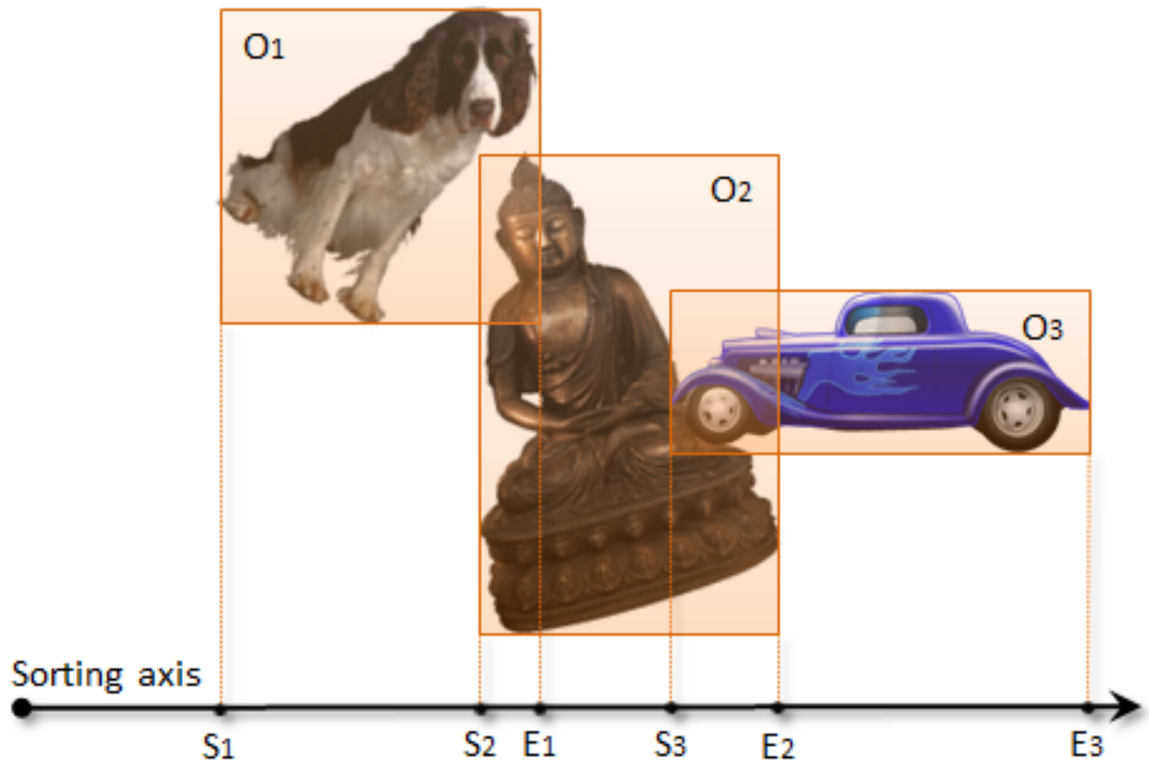
Collision Detection

- Naive approach: Full collision detection on all *n* items pairwise
  - $O(n^2)$ (this is a form of *n-body* computation)
  - Very inefficient

- Better approach: Use a "broad phase" and a "narrow phase"
  - Broad phase – identifies pairs of items that potentially collide
    - Often implemented on GPU, our focus today
  - Narrow phase – looks at the candidate pairs and determines actual collisions

Sort and Sweep

- Assign axis-aligned bounding boxes (AABB) for each object
- Project AABB to one axis
- Two items potentially collide if their projected boxes overlap

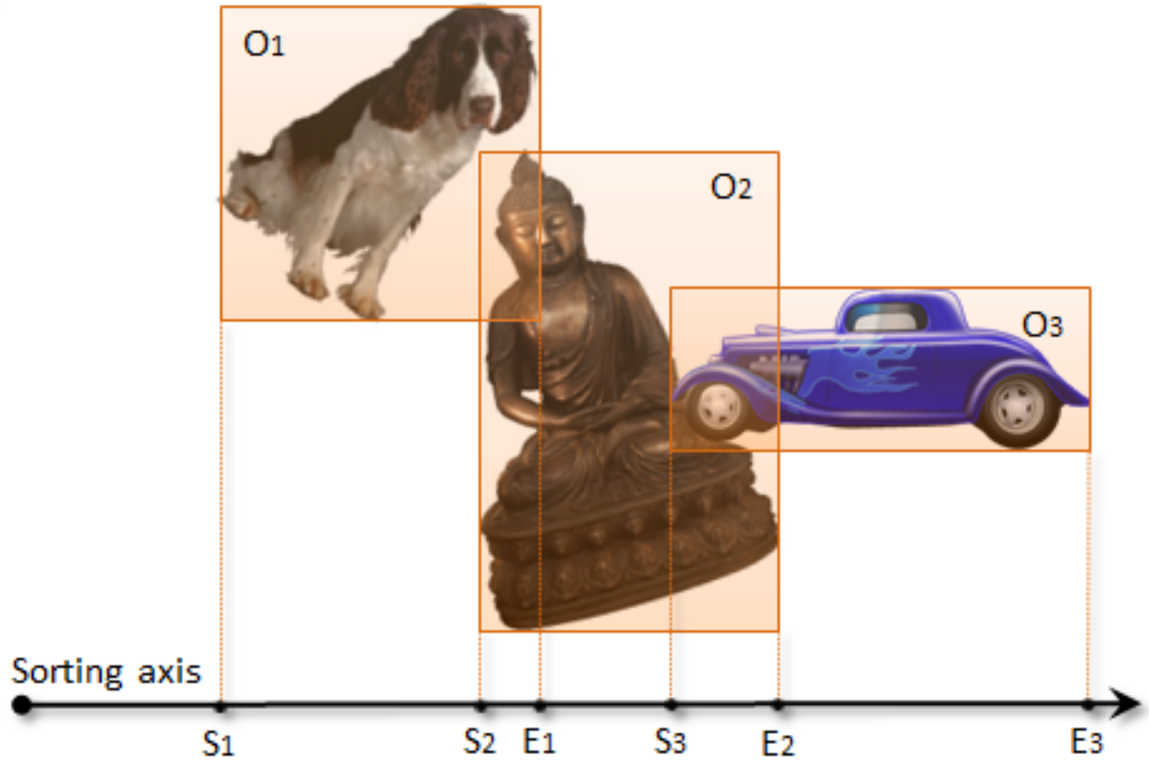


Sort and Sweep – Parallel

- Kernel 0: One thread per object
  - calculate AABB
  - project to axis
  - write start and end points into array

- Kernel 1: Sort the array

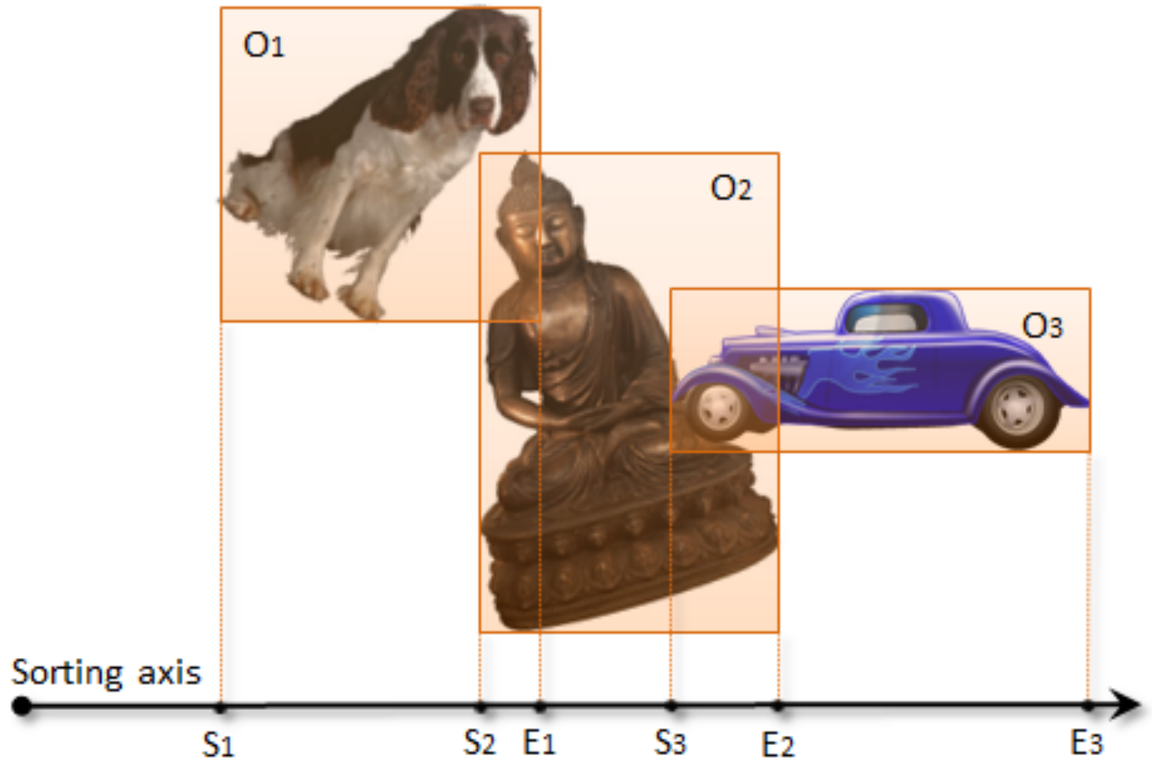
- Kernel 2: One thread per array element
  - If end point, exit
  - If start point, walk forward, outputting other start points until corresponding end point is reached



Sort and Sweep – Downsides

- Potentially output $O(n^2)$ pairs (if all object overlap on chosen axis)

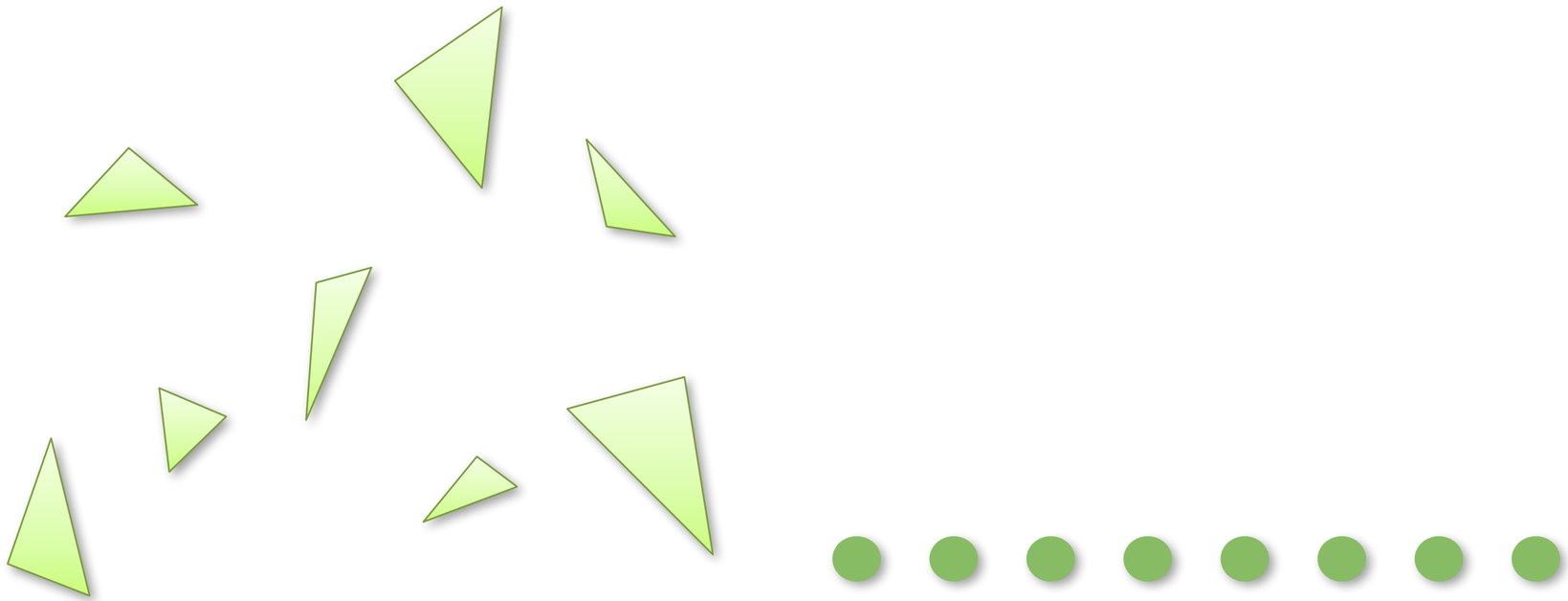
- Lots of divergence
  - Thread divergence
  - Data divergence (uncoalesced)



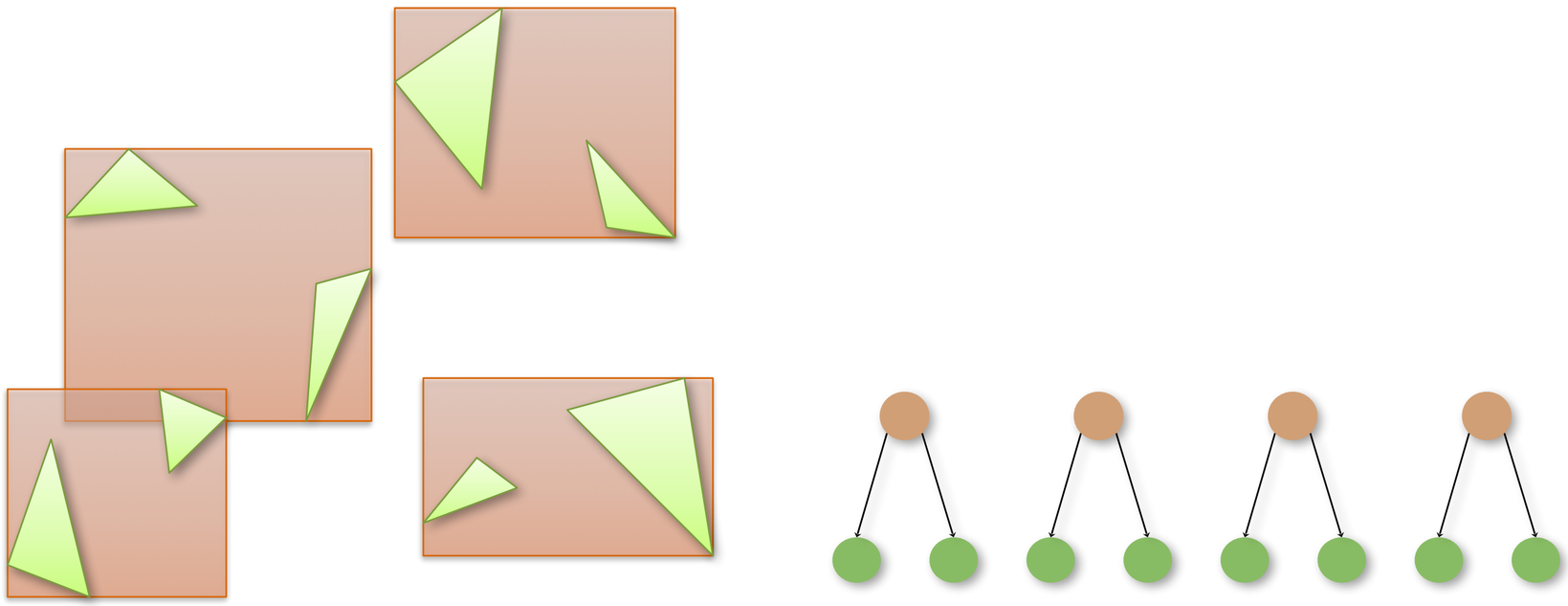
Bounding Volume Hierarchy

Bounding volume hierarchy

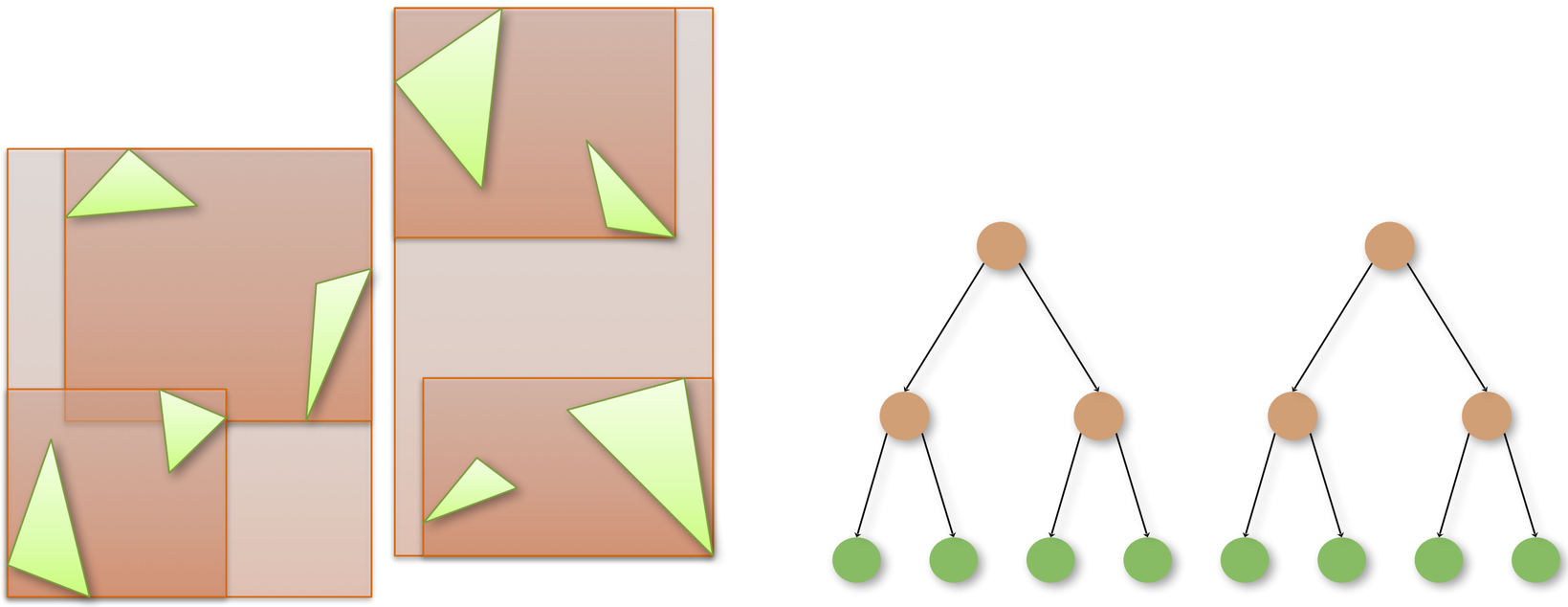
Bounding volume hierarchy



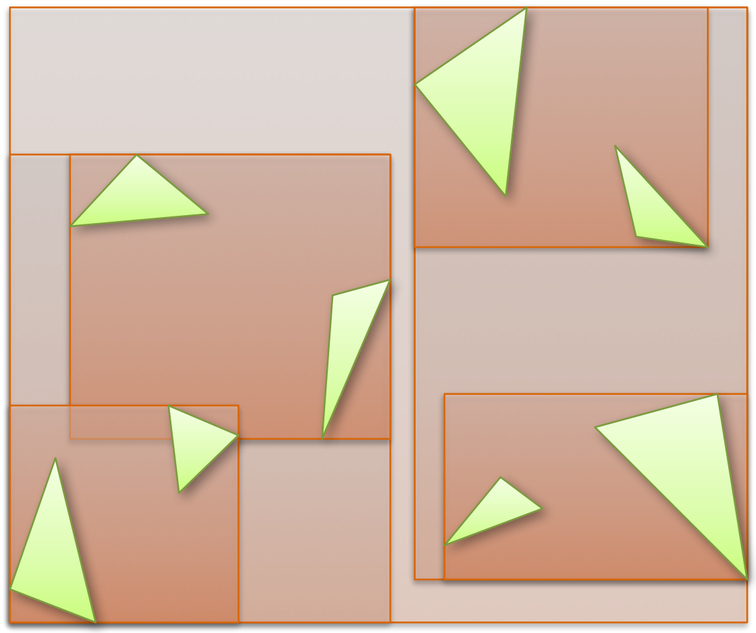
Bounding volume hierarchy

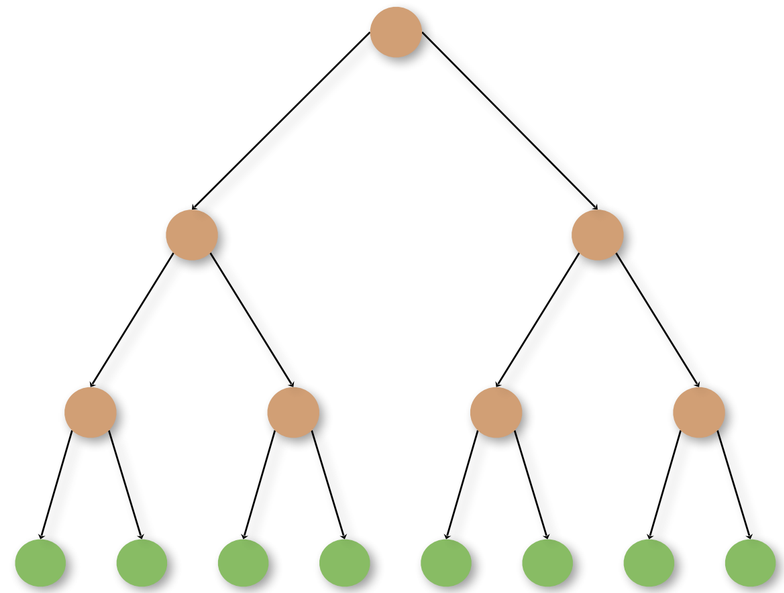


Bounding volume hierarchy

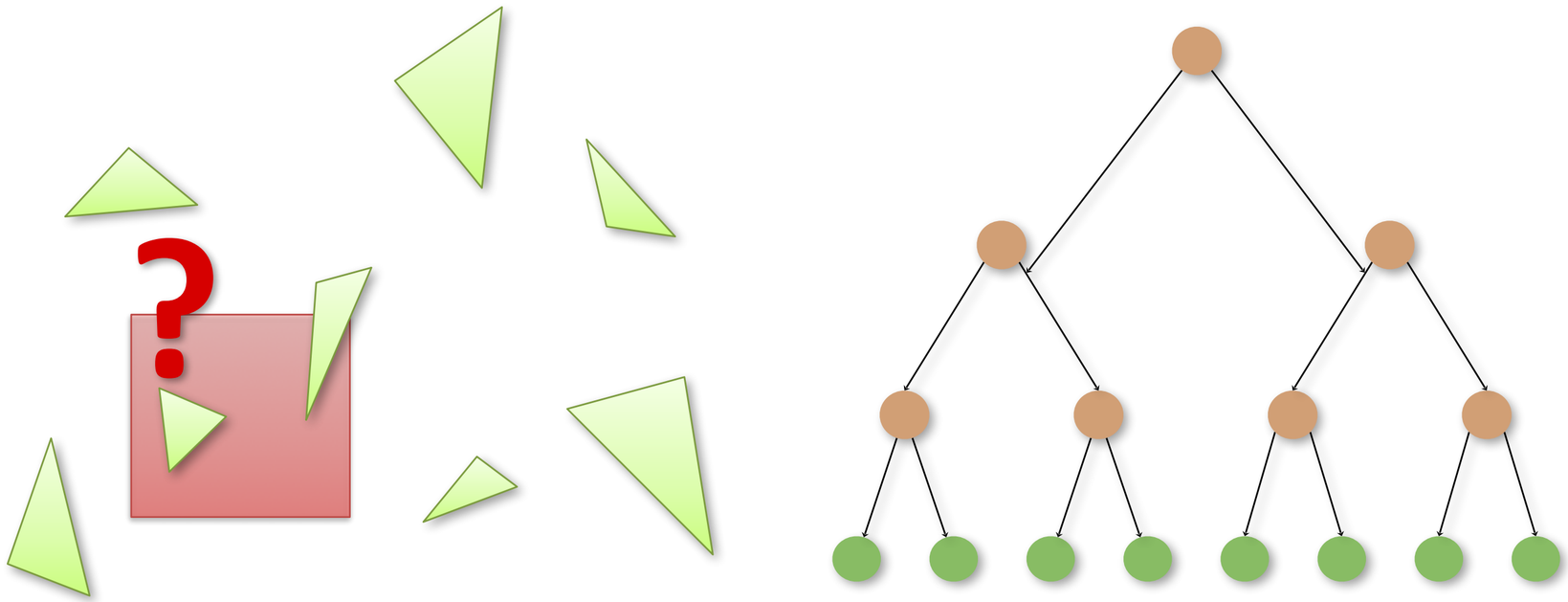


Bounding volume hierarchy

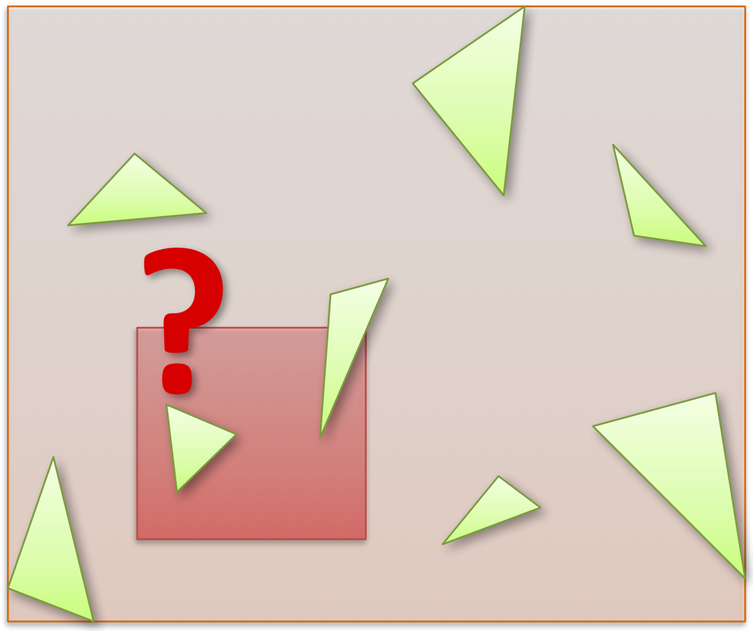


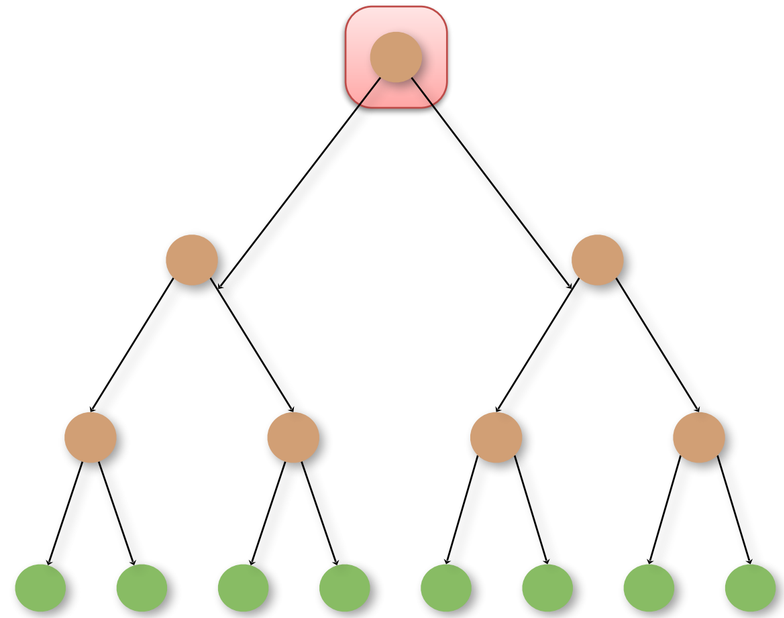


Bounding volume hierarchy

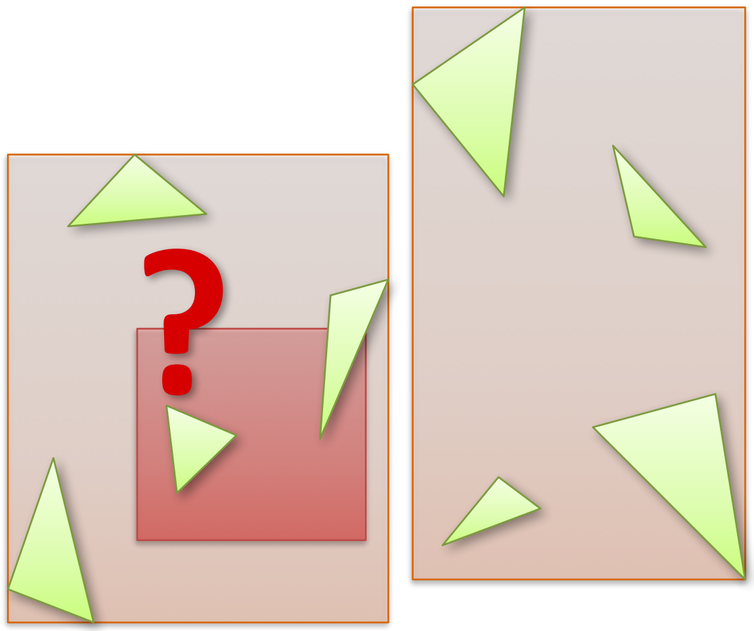


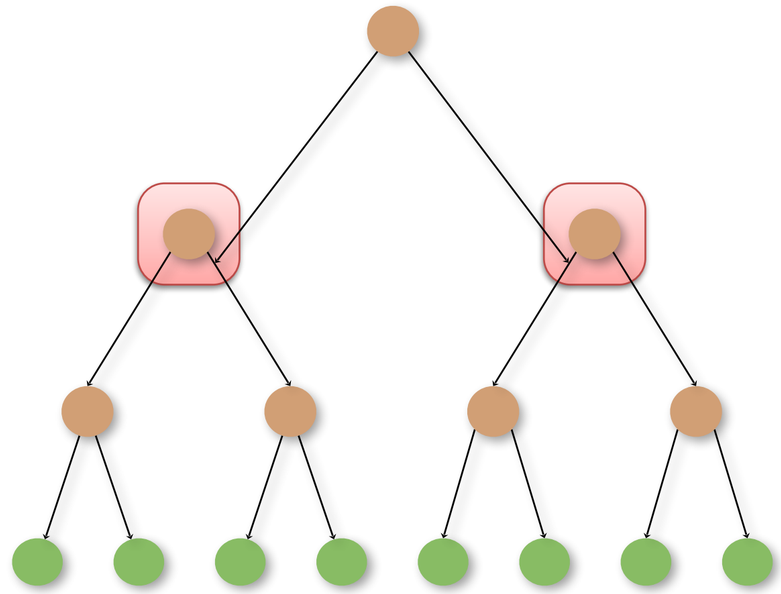
Bounding volume hierarchy



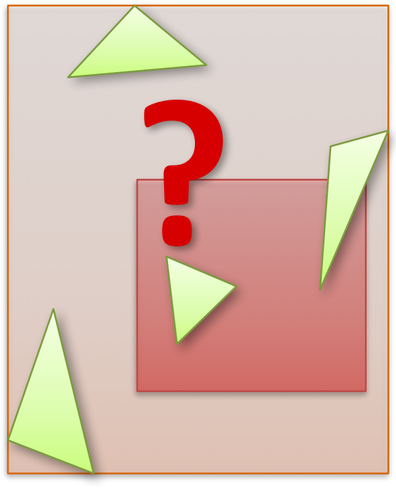


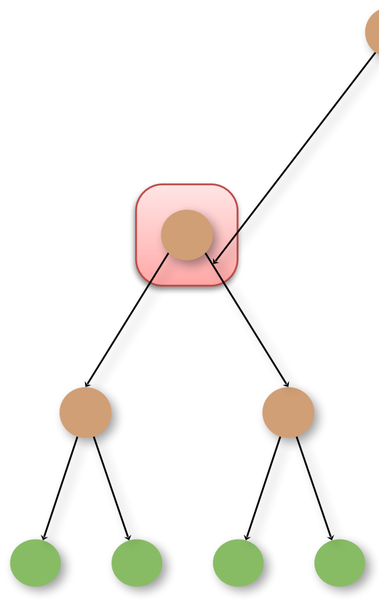
Bounding volume hierarchy



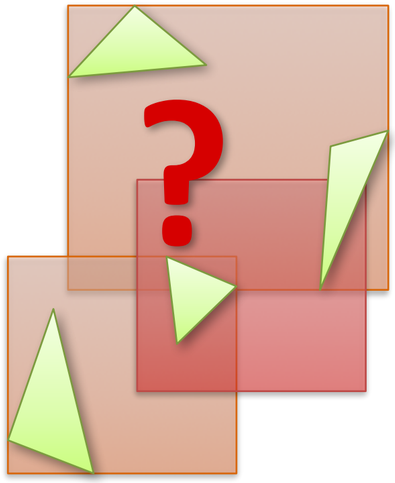


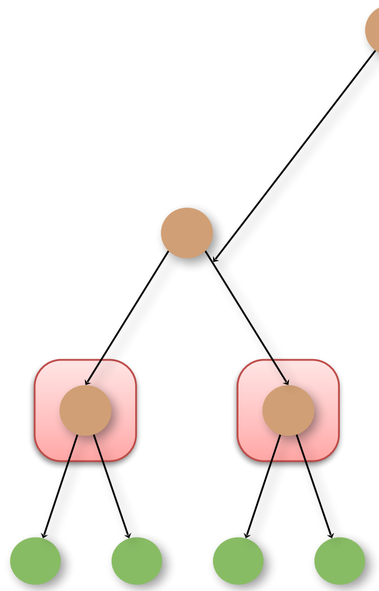
Bounding volume hierarchy



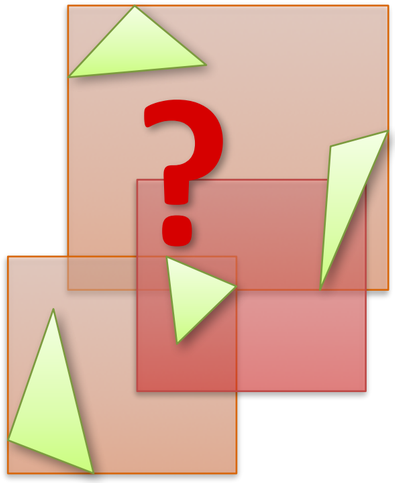


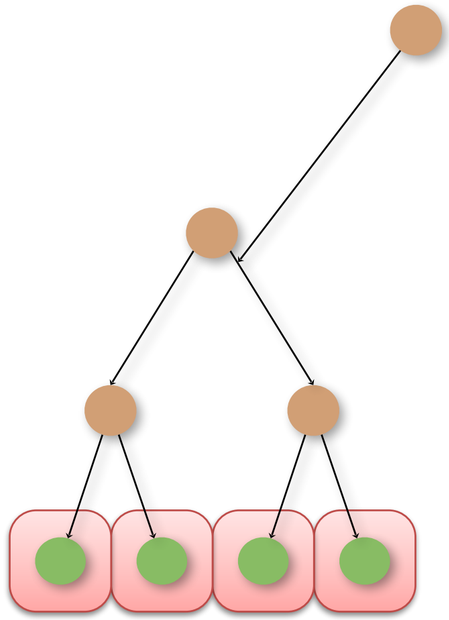
Bounding volume hierarchy



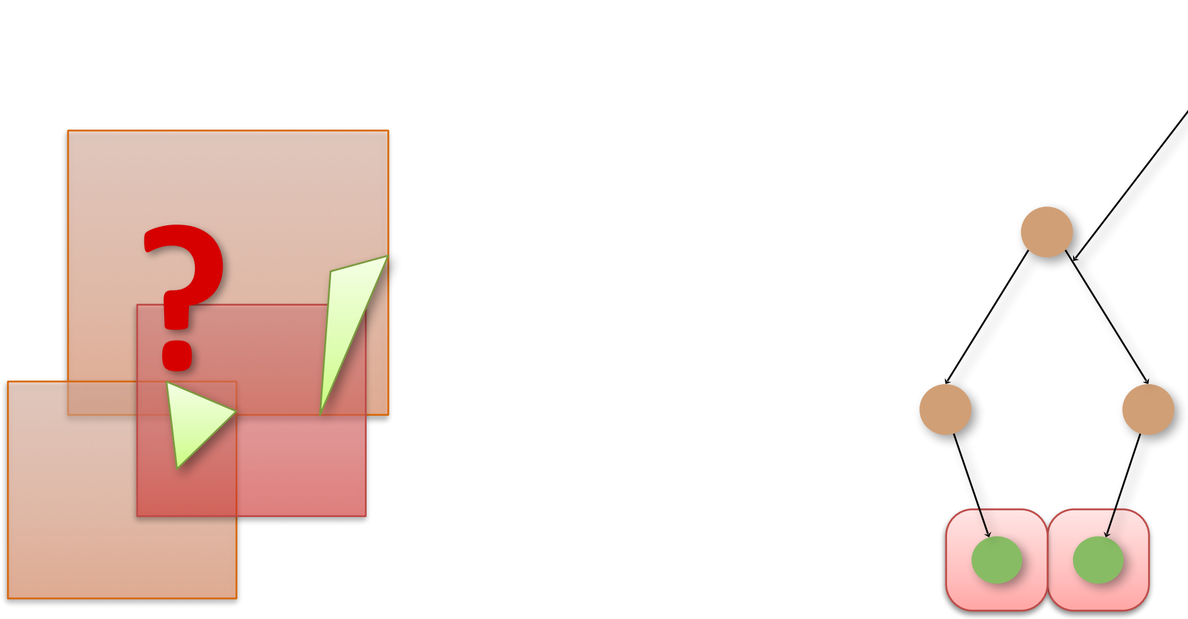


Bounding volume hierarchy



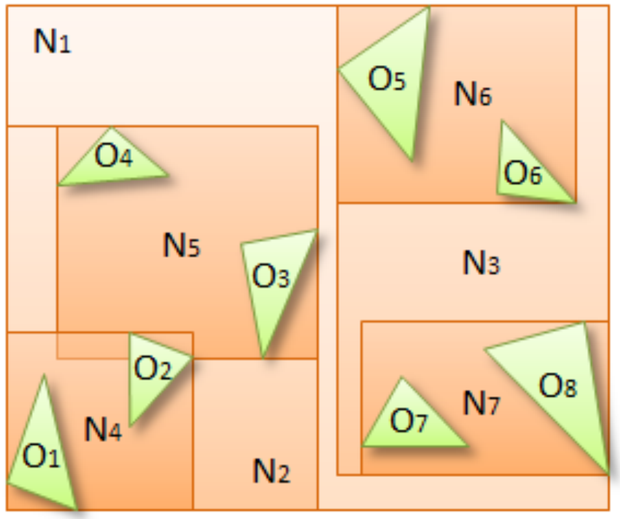


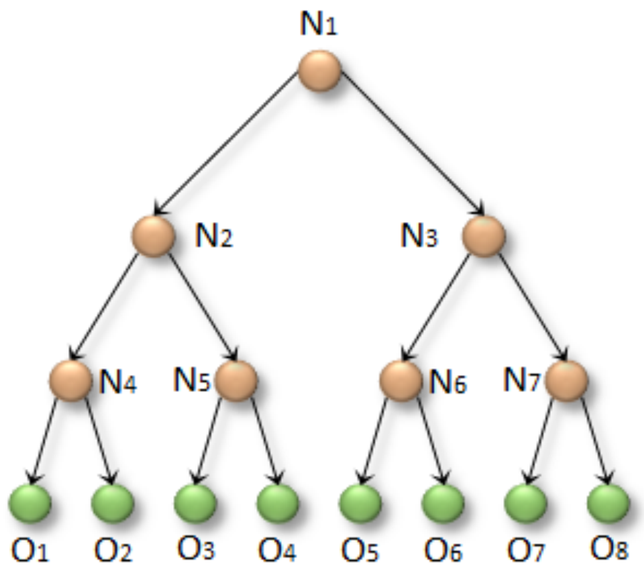
Bounding volume hierarchy



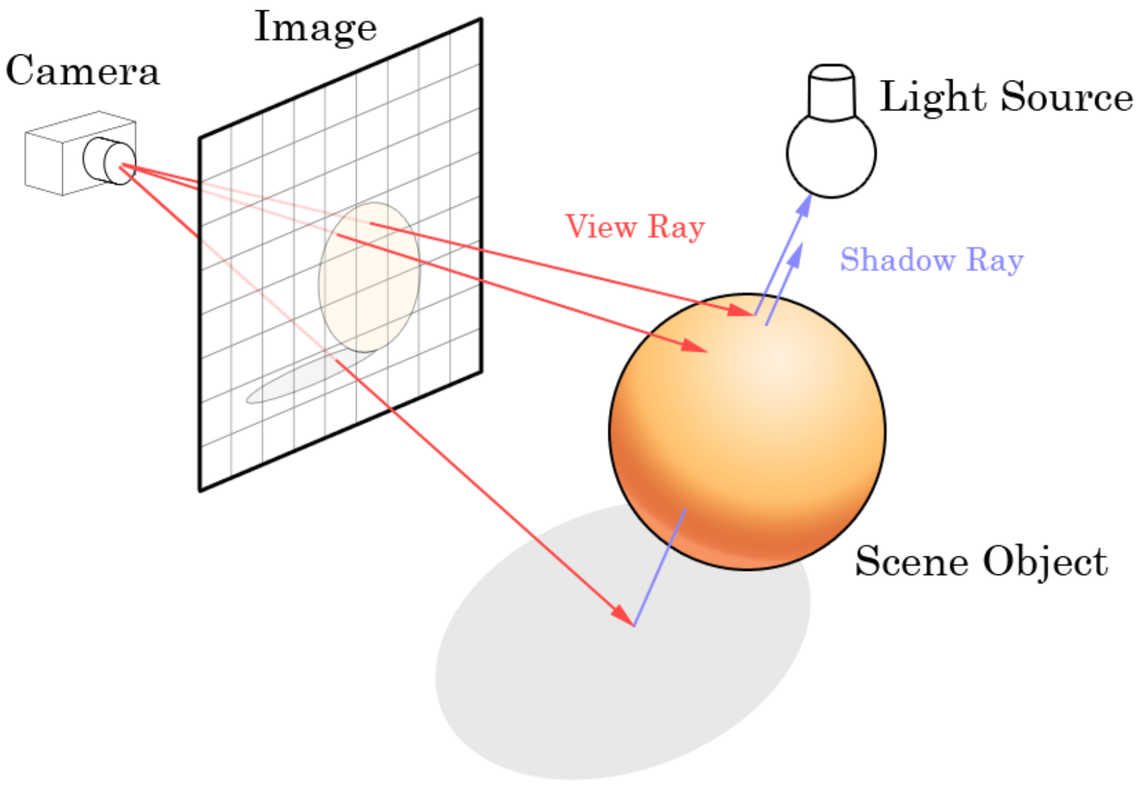
Bounding Volume Hierarchy (BVH)

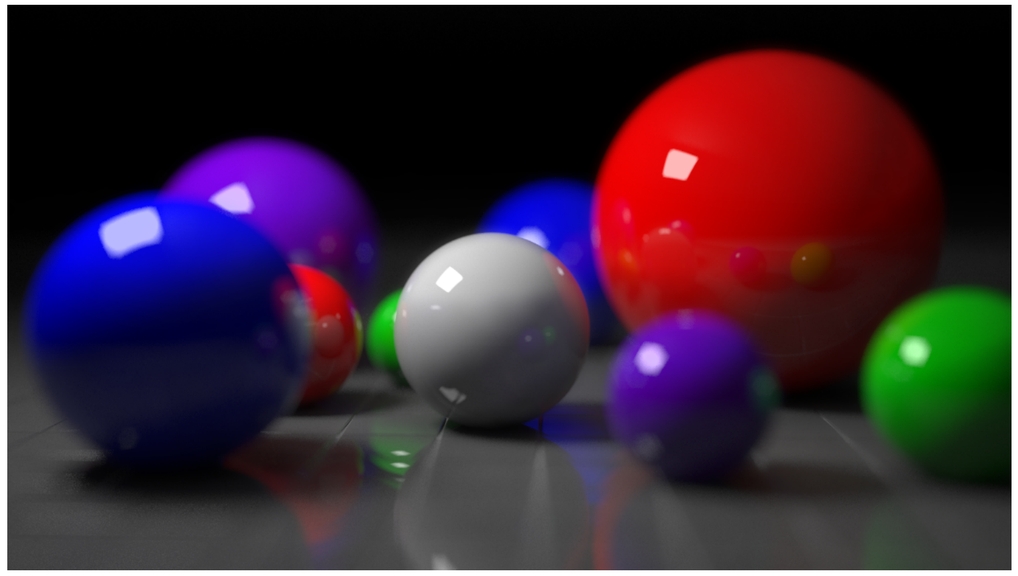
- Hierarchical grouping of 3D objects
- Nodes:
  - Leaf nodes – individual objects
  - Internal nodes – groupings of objects based on bounding volume
  - Root node – entire scene
- Bounding volumes can be any shape, but generally use axis-aligned bounding boxes (AABB) for simplicity
- Used in collision detection and ray tracing





BVH for Ray Tracing



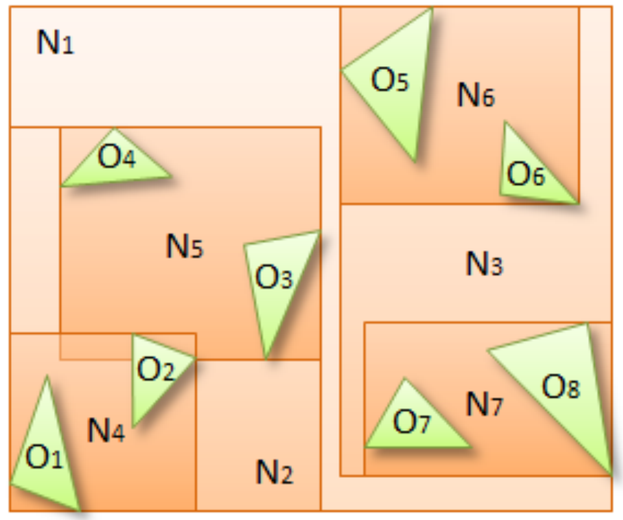


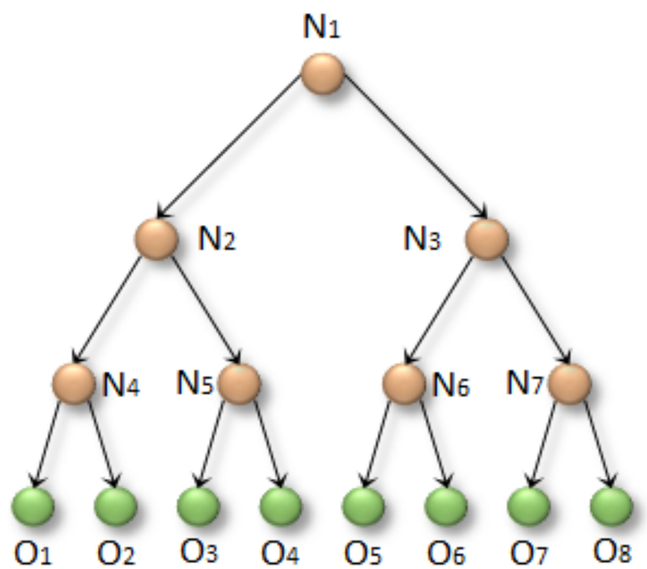
Tree Traversals

Scenario & things to consider

- Does AABB-specified object intersect any object in BVH?

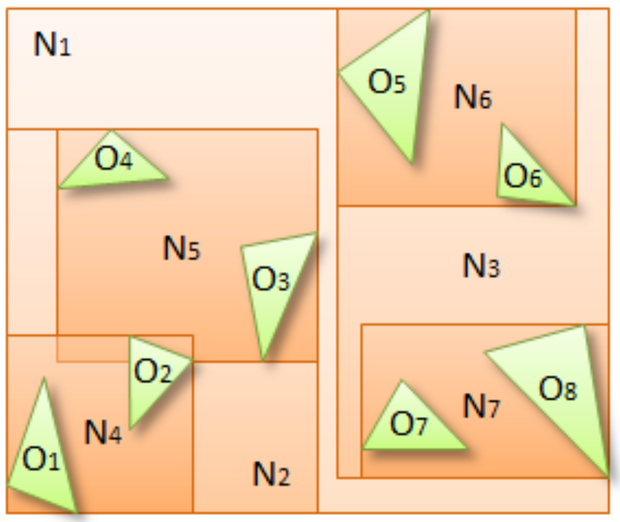
- Consider:
  - Amount of work
  - Occupancy
  - Thread divergence
  - Data access patterns

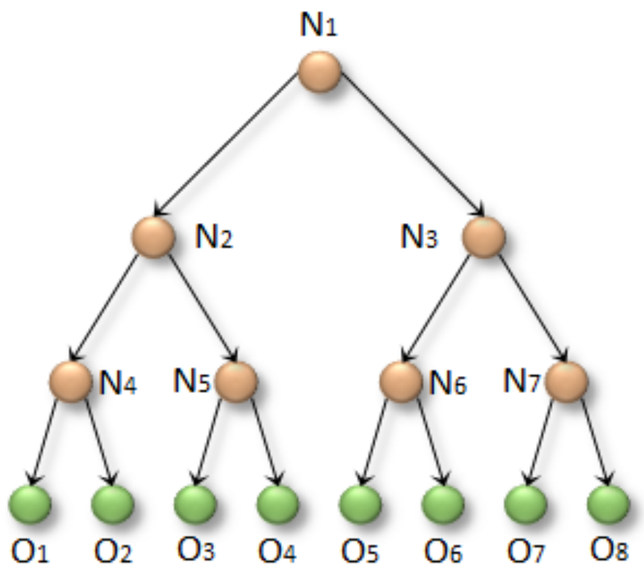




Options to Parallelize Traversal

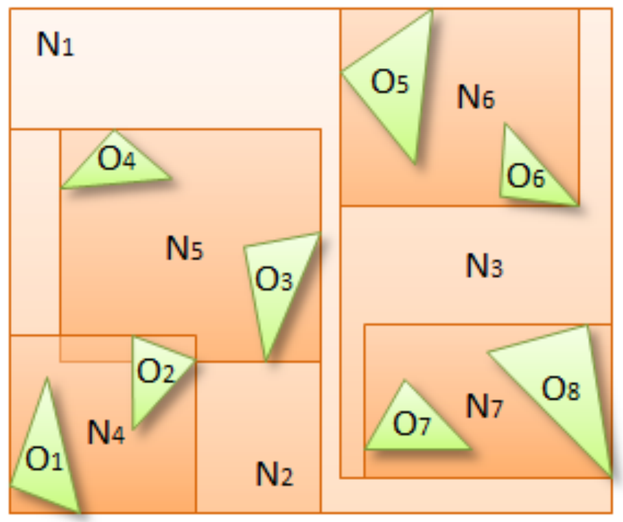
- One thread per object
  - E.g., [O_2, O_7, O_1, O_4, …]
  - Different objects take different paths down the tree

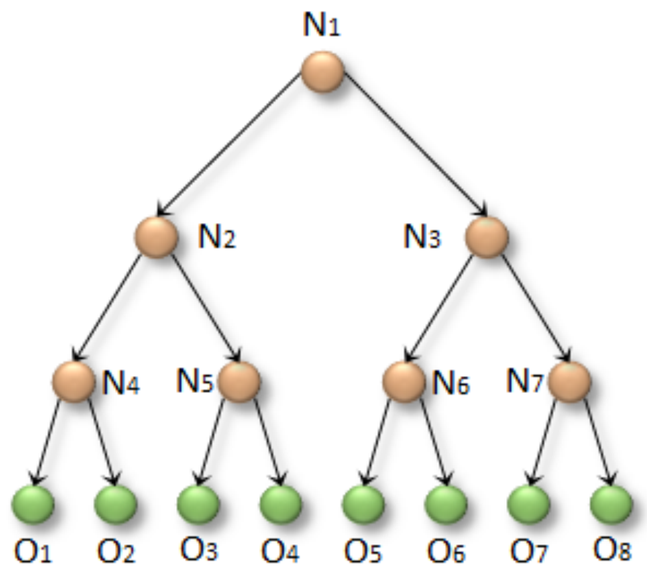




Options to Parallelize Traversal

- One thread per object
  - E.g., [O_2, O_7, O_1, O_4, …]
  - Different objects take different paths down the tree
  - Lots of divergence





BVH Traversal – First Attempt (C++)

query object

```
void traverseRecursive( CollisionList& list,
                        const BVH&     bvh,
                        const AABB&    queryAABB,
                        int            queryObjectIdx,
                        NodePtr        node) {
    // Bounding box overlaps the query => process node.
    if (checkOverlap(bvh.getAABB(node), queryAABB)) {
        // Leaf node => report collision.
        if (bvh.isLeaf(node))
            list.add(queryObjectIdx, bvh.getObjectIdx(node));
        // Internal node => recurse to children.
        else {
            NodePtr childL = bvh.getLeftChild(node);
            NodePtr childR = bvh.getRightChild(node);
            traverseRecursive(bvh, list, queryAABB,
                              queryObjectIdx, childL);
            traverseRecursive(bvh, list, queryAABB,
                              queryObjectIdx, childR);
        }
    }
}
```

BVH Traversal – First Attempt (CUDA)

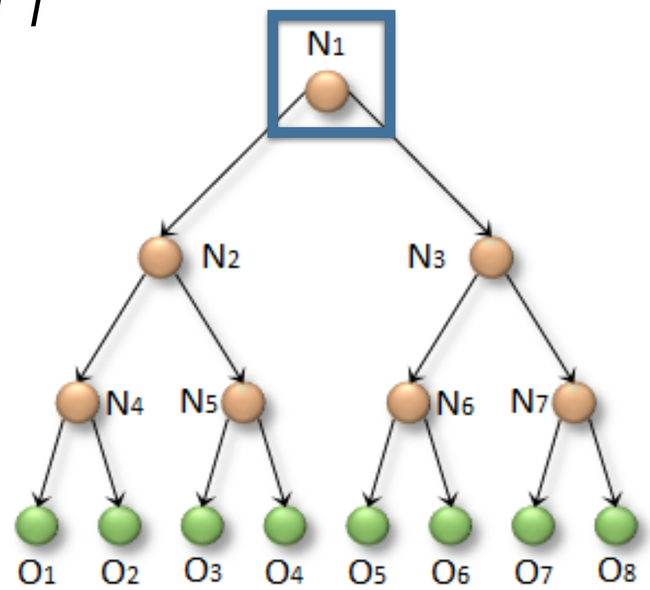
```
__device__ void traverseRecursive( CollisionList& list,
                                   const BVH&     bvh,
                                   const AABB&    queryAABB,
                                   int            queryObjectIdx,
                                   NodePtr        node)
{
    // same as before...
}
```

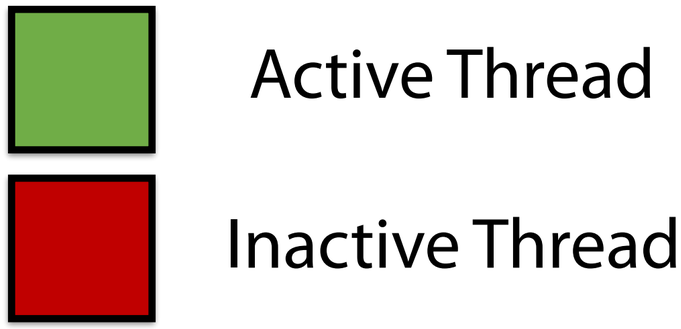
parallelize over query objects

```
__global__ void findPotentialCollisions( CollisionList list,
                                         BVH           bvh,
                                         AABB*         objectAABBs,
                                         int           numObjects)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < numObjects)
        traverseRecursive(bvh, list, objectAABBs[idx],
                          idx, bvh.getRoot());
}
```

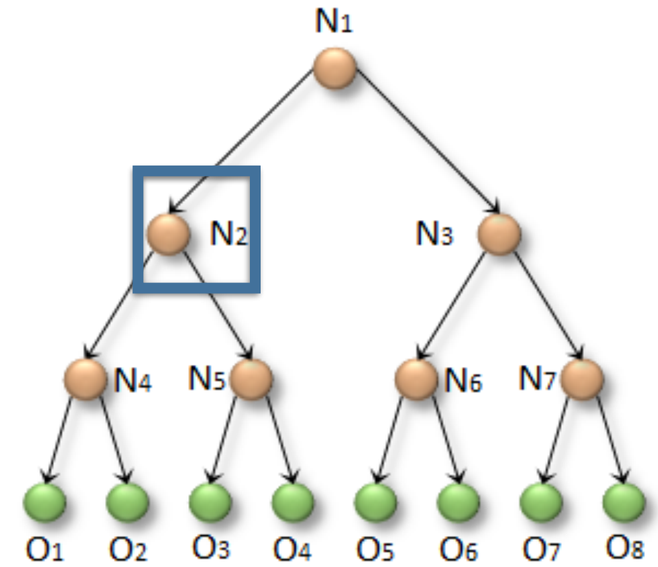
BVH Traversal – First Attempt

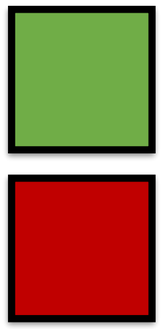
Note we're querying the same geometry that's in the BVH





BVH Traversal – First Attempt

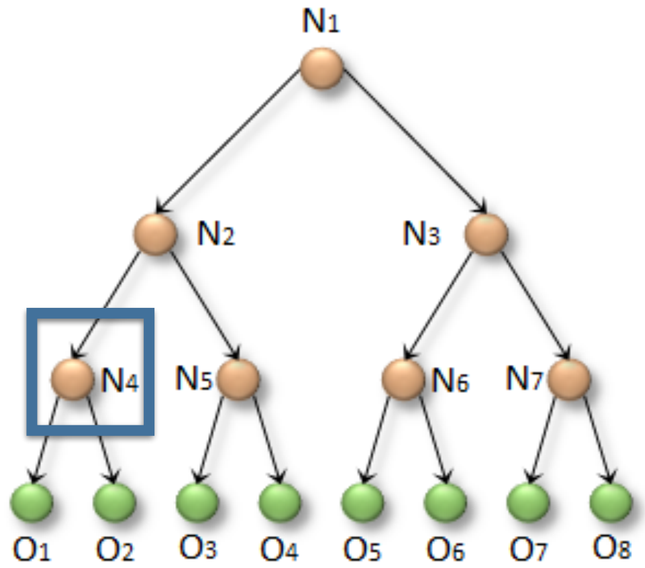


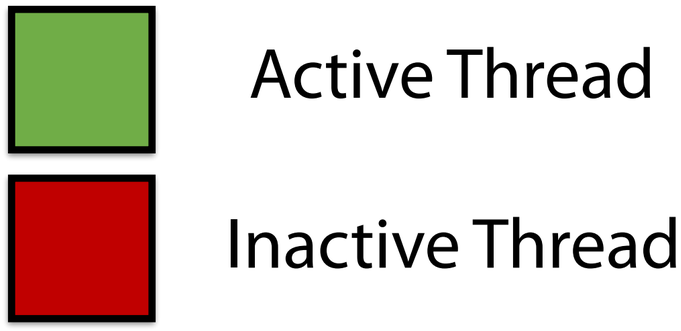


Active Thread

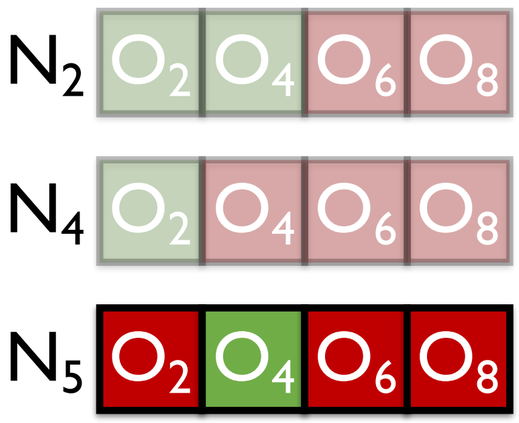
Inactive Thread

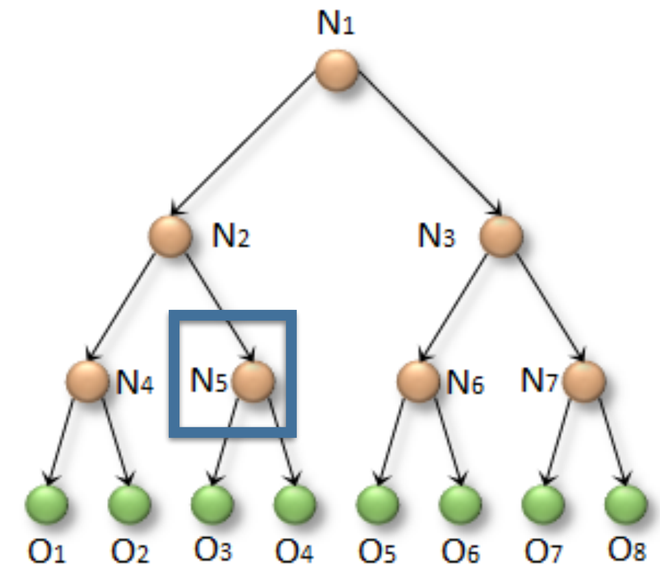
BVH Traversal – First Attempt

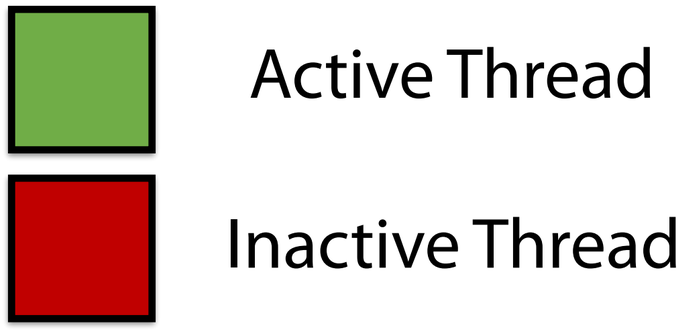




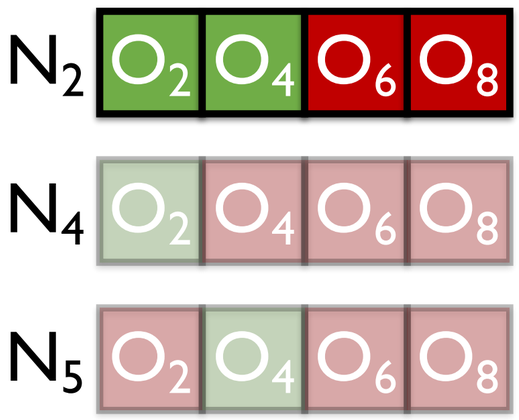
BVH Traversal – First Attempt

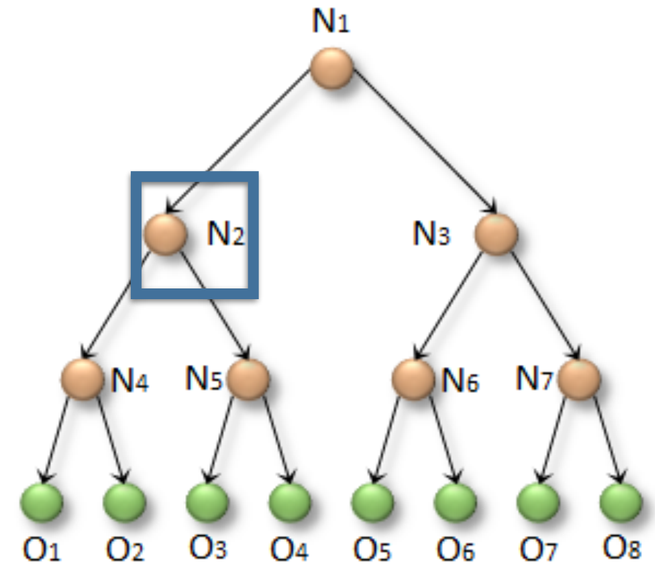


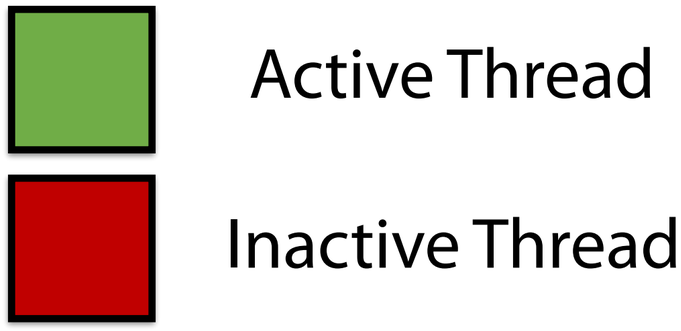




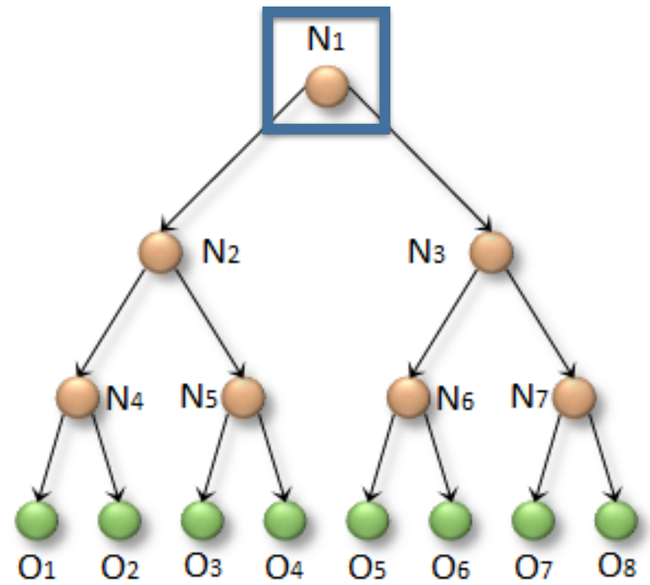
BVH Traversal – First Attempt

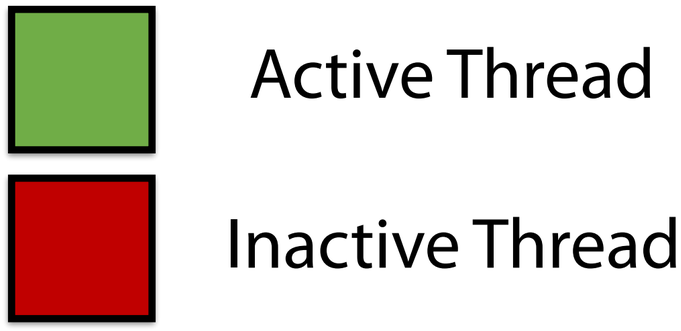




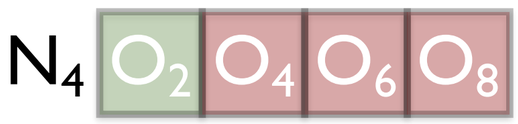


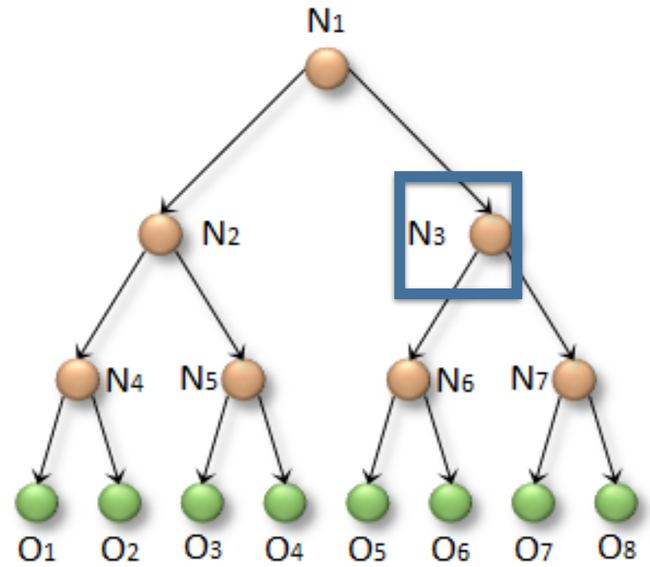
BVH Traversal – First Attempt

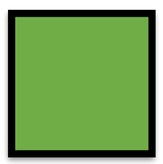




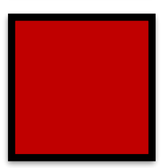
BVH Traversal – First Attempt





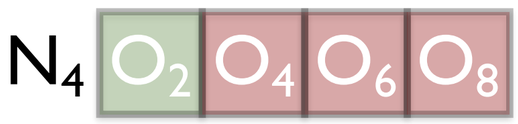


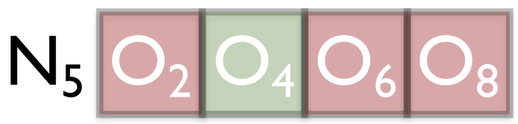
Active Thread

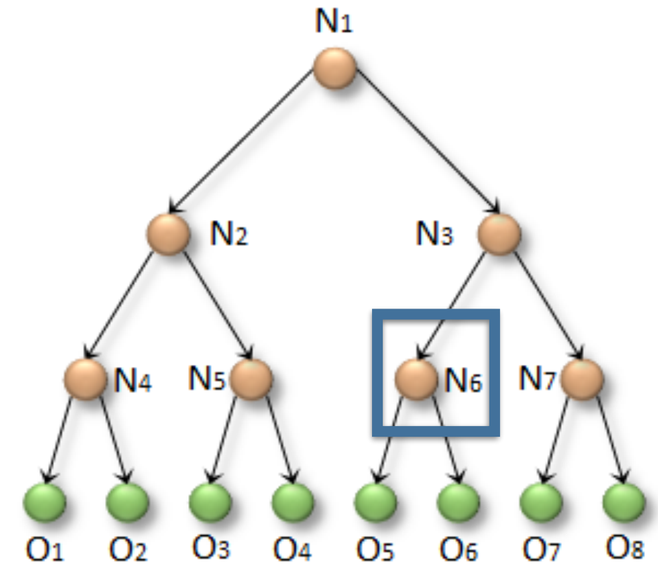


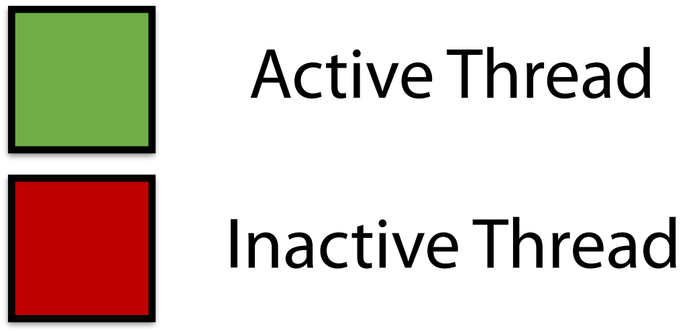
Inactive Thread

BVH Traversal – First Attempt

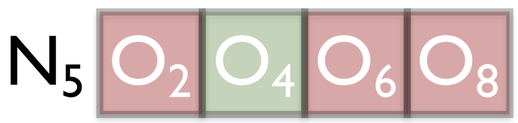


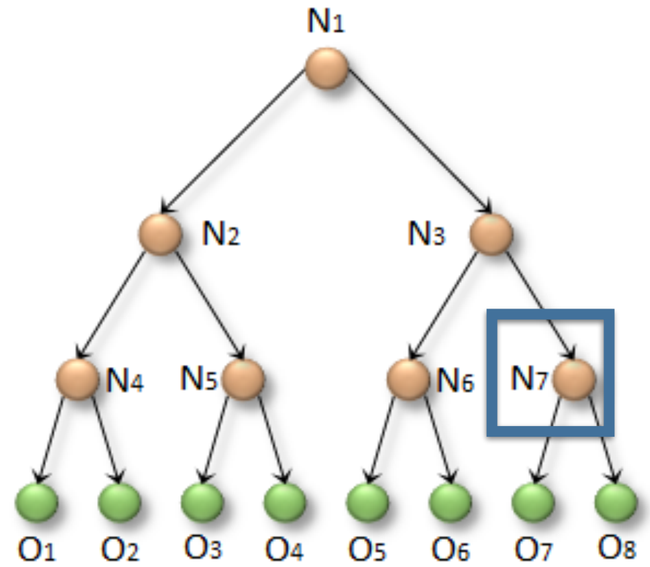


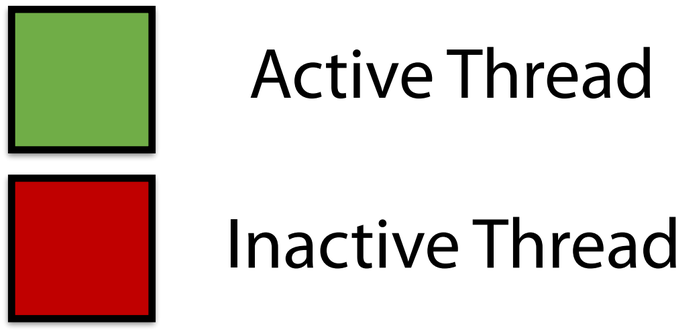




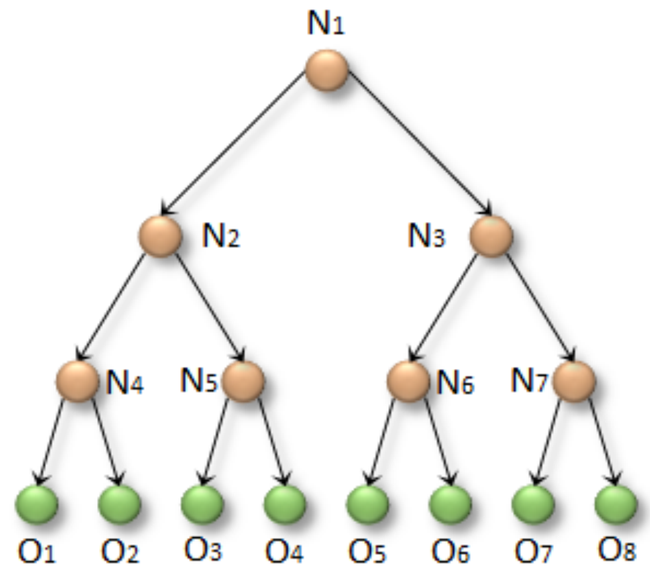
BVH Traversal – First Attempt

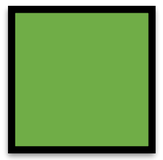




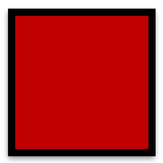


BVH Traversal – First Attempt





Active Thread



Inactive Thread

BVH Traversal – First Attempt Problems

- **Lots of thread divergence**

```
// Internal node => recurse to children.
else {
    NodePtr childL = bvh.getLeftChild(node);
    NodePtr childR = bvh.getRightChild(node);
    traverseRecursive(bvh, list, queryAABB, queryObjectIdx, childL);
    traverseRecursive(bvh, list, queryAABB, queryObjectIdx, childR);
}
```

If only 1 thread takes this path, the other 31 have to wait until that thread finishes its full depth traversal

BVH Traversal – First Attempt Performance

- Dataset from APEX Destruction
  - 12,486 objects
  - 73,704 pairs of potentially colliding objects
- Tested on NVIDIA GeForce GTX 690
- Execution time: 3.8 milliseconds
  - Very slow!
  - Only one part of collision detection
  - Only one part of a simulation ideally running at 60 FPS (16 ms/frame)
- Try 2: Convert to iterative; manage call stack explicitly

BVH Traversal – Second Attempt

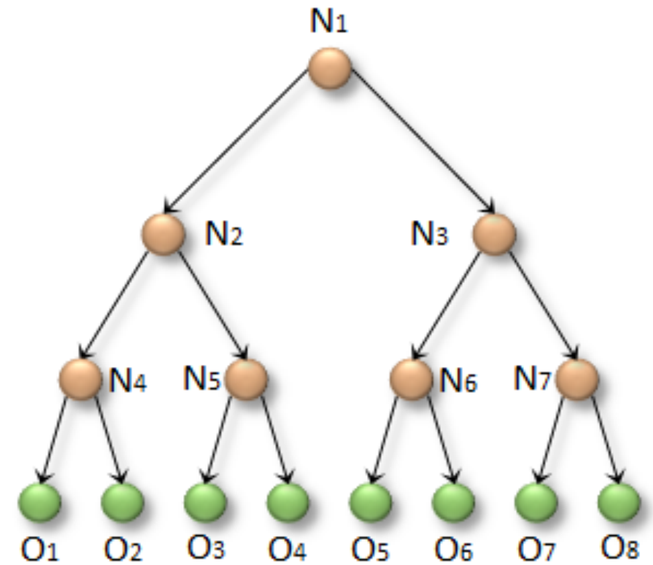
```
__device__ void traverseIterative( CollisionList& list,
                                   BVH& bvh,
                                   AABB& queryAABB,
                                   int queryObjectIdx)
{
    // Allocate traversal stack from thread-local memory,
    // and push NULL to indicate that there are no postponed nodes.
    NodePtr stack[64];
    NodePtr* stackPtr = stack;
    *stackPtr++ = NULL; // push

    // Traverse nodes starting from the root.
    NodePtr node = bvh.getRoot();
    do {
        // see next
    }
    while (node != NULL);
}
```

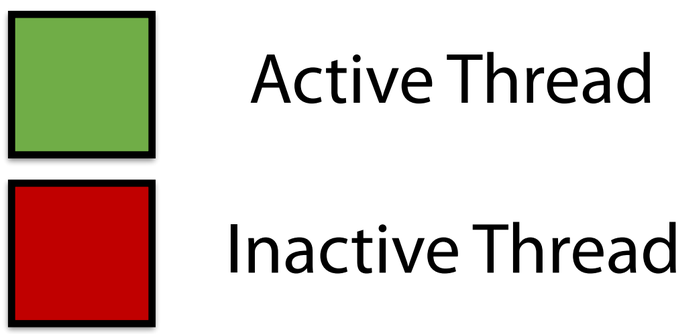
BVH Traversal – Second Attempt

```
__device__ void traverseIterative(

{
    // Allocate traversal stack fro
    // and push NULL to indicate th
    NodePtr stack[64];
    NodePtr* stackPtr = stack;
    *stackPtr++ = NULL; // push

    // Traverse nodes starting from
    NodePtr node = bvh.getRoot();
    do {
        // see next
    }
    while (node != NULL);
}
```

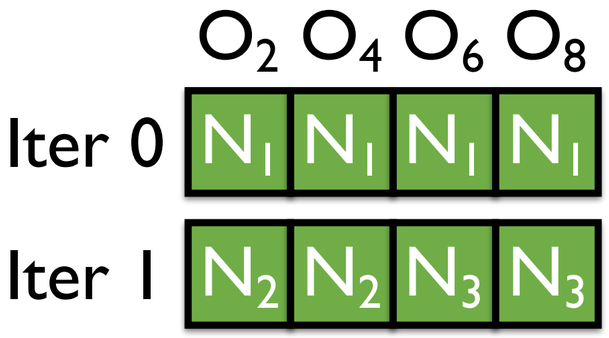
```
// Check each child node for overlap.
NodePtr childL = bvh.getLeftChild(node);
NodePtr childR = bvh.getRightChild(node);
bool overlapL = ( checkOverlap(queryAABB,
                               bvh.getAABB(childL)) );
bool overlapR = ( checkOverlap(queryAABB,
                               bvh.getAABB(childR)) );

// Query overlaps a leaf node => report collision.
if (overlapL && bvh.isLeaf(childL))
    list.add(queryObjectIdx, bvh.getObjectIdx(childL));

if (overlapR && bvh.isLeaf(childR))
    list.add(queryObjectIdx, bvh.getObjectIdx(childR));

// Query overlaps an internal node => traverse.
bool traverseL = (overlapL && !bvh.isLeaf(childL));
bool traverseR = (overlapR && !bvh.isLeaf(childR));

if (!traverseL && !traverseR)
    node = *--stackPtr; // pop
else {
    node = (traverseL) ? childL : childR;
    if (traverseL && traverseR)
        *stackPtr++ = childR; // push
}
```

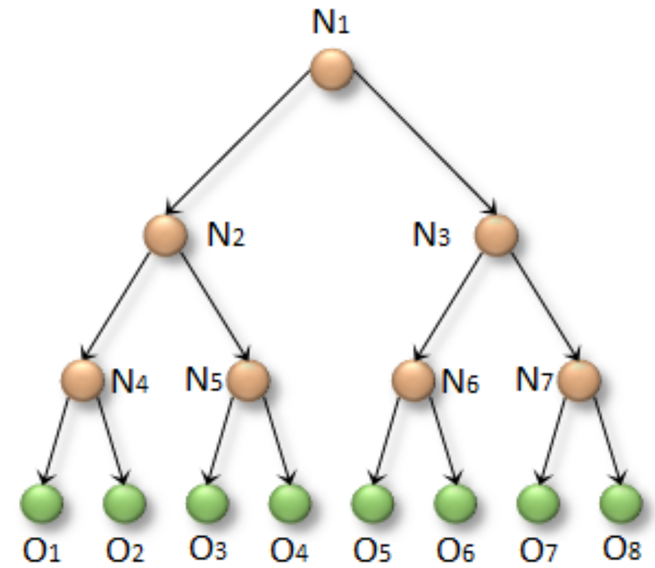
BVH Traversal – Second Attempt

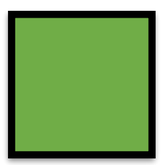




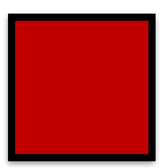
BVH Traversal – Second Attempt





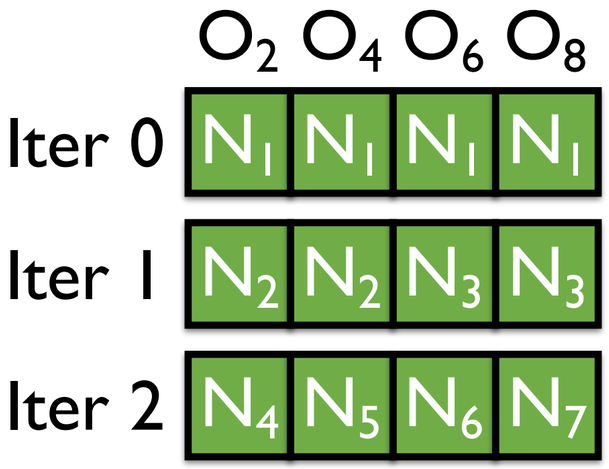


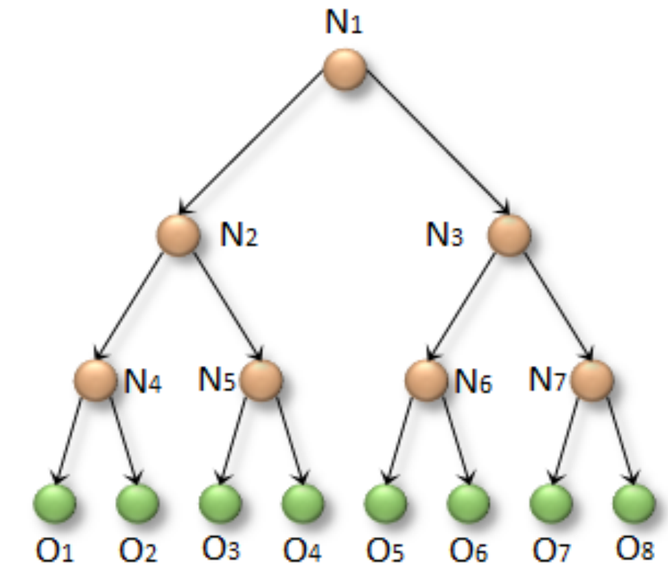
Active Thread

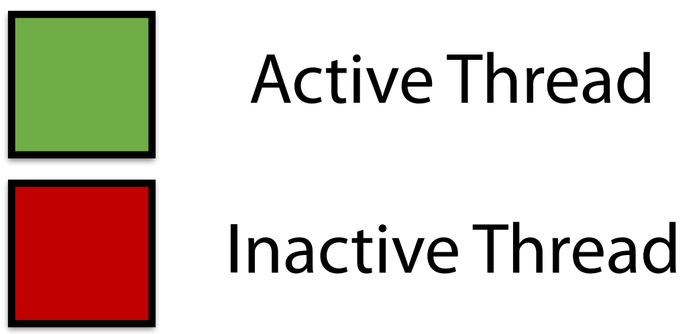


Inactive Thread

BVH Traversal – Second Attempt







BVH Traversal – Second Attempt Performance

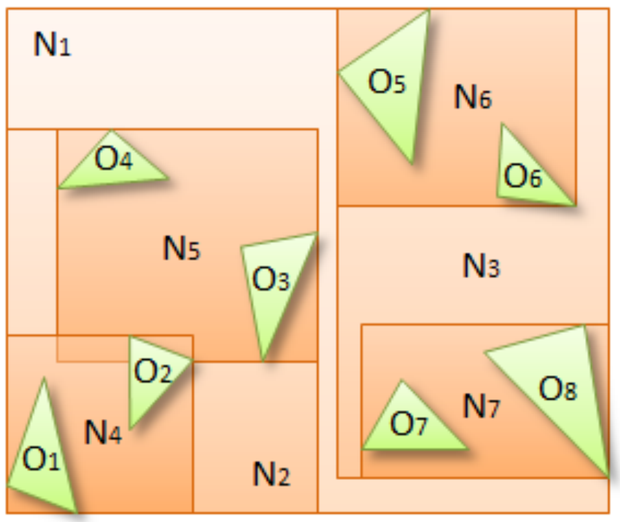
- Execution time: 0.91 milliseconds
  - vs. 3.8 ms for the recursive kernel

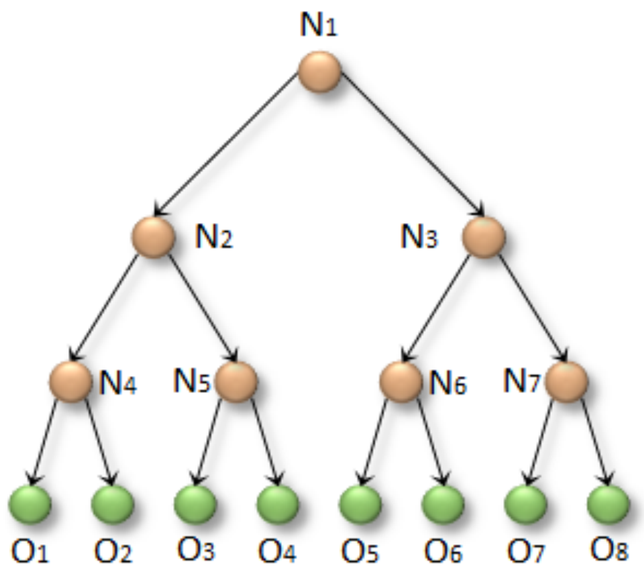
- Improvement over recursive: less thread divergence
  - Each thread executes the same loop repeatedly, regardless of traversal decisions

- Problem: data divergence
  - Threads might be accessing different parts of the tree
  - Threads may execute different number of iterations

Options to Parallelize Traversal

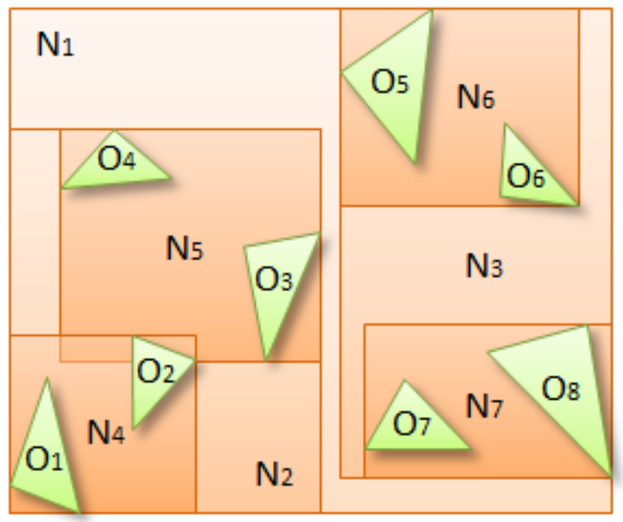
- One thread per object
  - E.g., [O_2, O_7, O_1, O_4, …]
  - Different objects take different paths down the tree

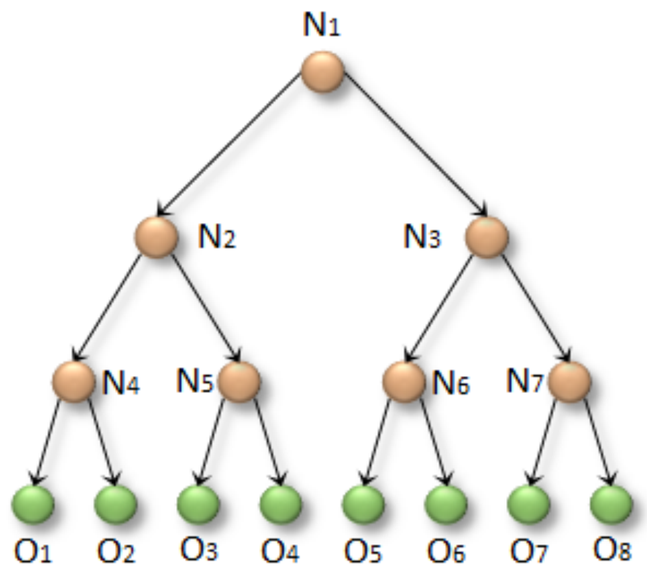




Options to Parallelize Traversal

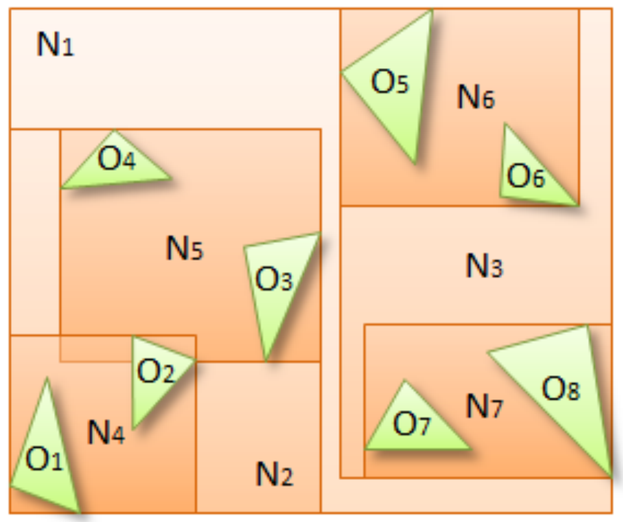
- One thread per object
  - E.g., [O_2, O_7, O_1, O_4, …]
  - Different objects take different paths down the tree
  - Lots of divergence



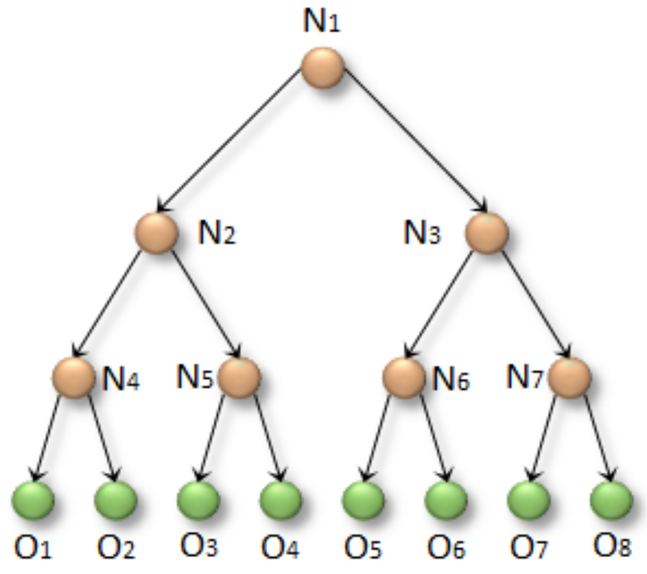


Options to Parallelize Traversal

- One thread per object
  - E.g., [O_2, O_7, O_1, O_4, …]
  - Different objects take different paths down the tree
  - Lots of divergence

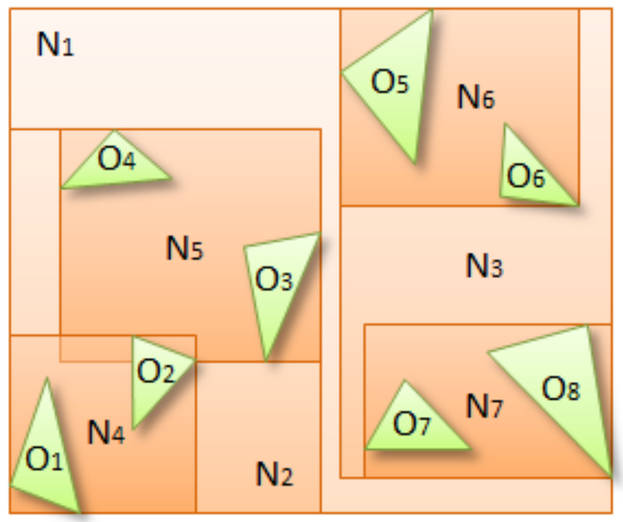


- One thread per leaf node
  - E.g., [O_1, O_2, O_3, O_4, …]
  - Still one thread per object

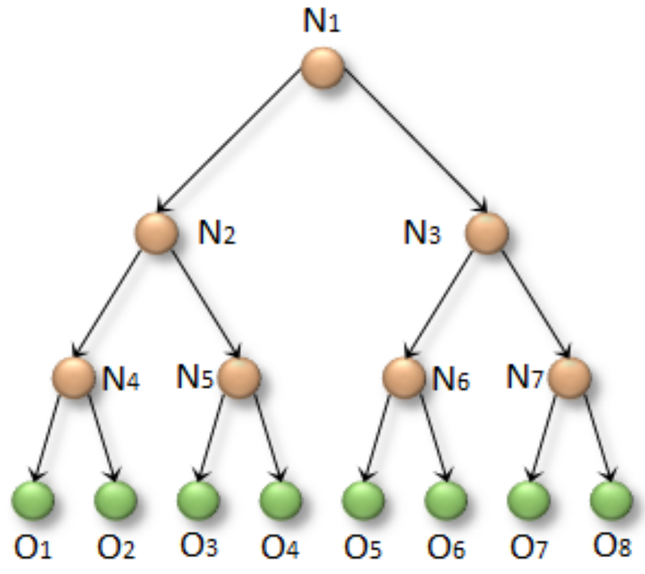


Options to Parallelize Traversal

- One thread per object
  - E.g., [O_2, O_7, O_1, O_4, …]
  - Different objects take different paths down the tree
  - Lots of divergence



- One thread per leaf node
  - E.g., [O_1, O_2, O_3, O_4, …]
  - Still one thread per object
  - But order of objects changed



BVH Traversal – Third Attempt

One thread per object (Second Attempt)

random order

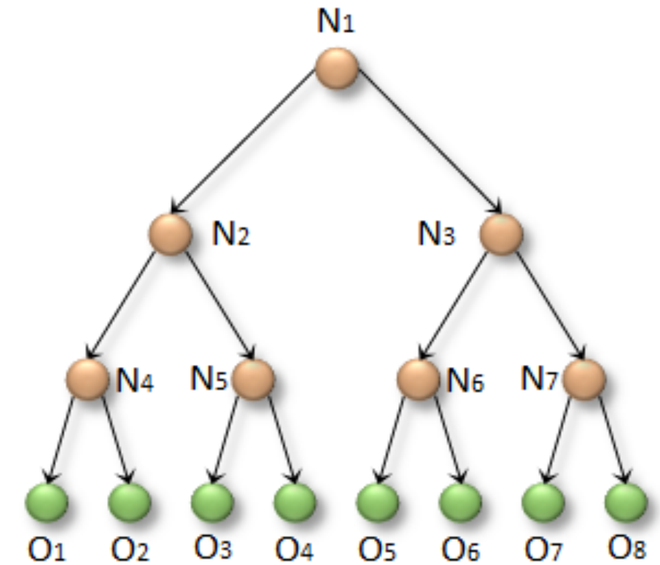
```
__global__ void findPotentialCollisions( CollisionList list, BVH bvh,
                                         AABB* objectAABBs, int numObjects) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < numObjects)
        traverseIterative(list, bvh, objectAABBs[idx], idx);
}
```

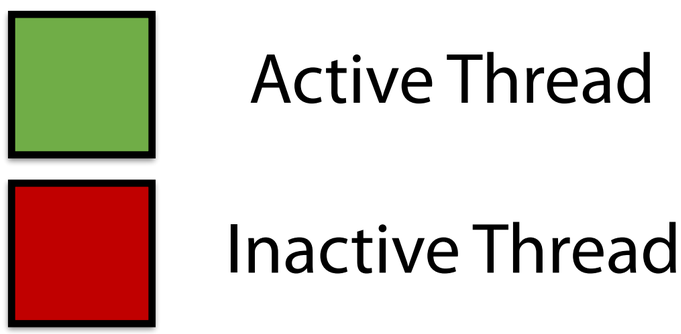
One thread per leaf node (Third Attempt)

```
__global__ void findPotentialCollisions( CollisionList list,
                                         BVH           bvh) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < bvh.getNumLeaves())
    {
        NodePtr leaf = bvh.getLeaf(idx);
        traverseIterative(list, bvh,
                          bvh.getAABB(leaf),
                          bvh.getObjectIdx(leaf));
    }
}
```

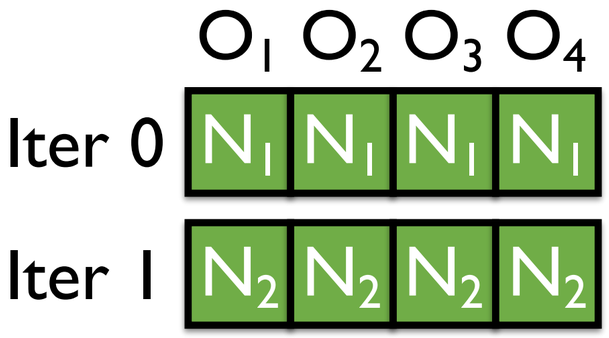
leaf order

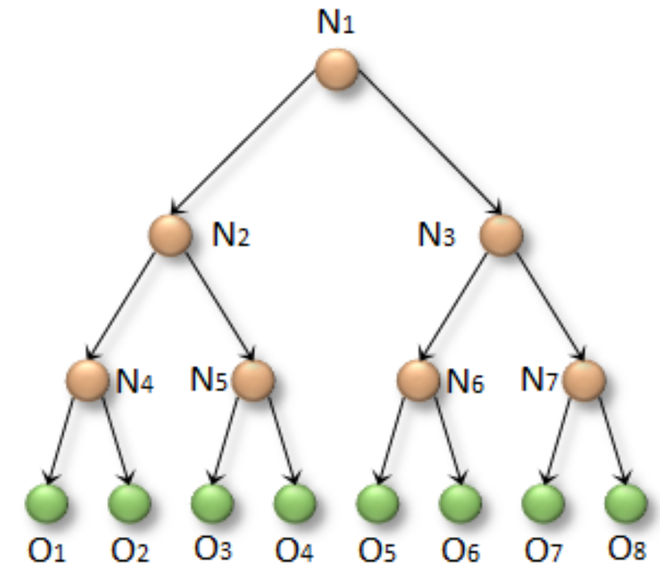
BVH Traversal – Third Attempt





BVH Traversal – Third Attempt

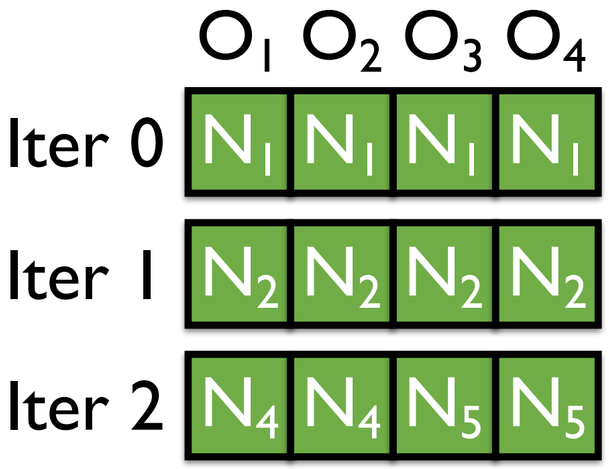


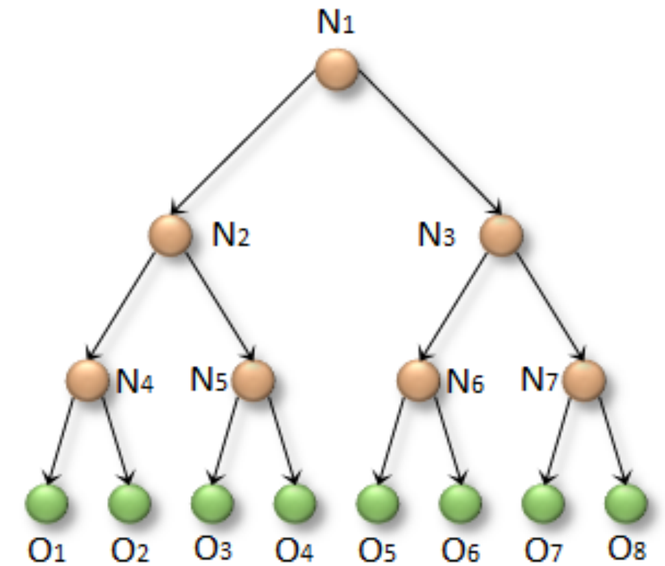


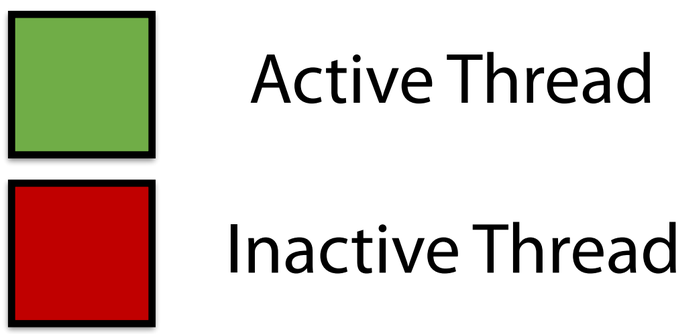
Active Thread

Inactive Thread

BVH Traversal – Third Attempt







BVH Traversal – Third Attempt Performance

- Execution time: 0.43 milliseconds
  - vs. 0.91 ms for second attempt
  - vs. 3.8 ms for the recursive kernel
- Improvement over second: better data access patterns
  - Objects close to each other (and, thus, likely to collide) are likely located in nearby leaf nodes
    - Improvement from coalescing, cache efficiency
- Problem: each collision reported twice
  - Twice as much work!
  - Solution: Require that A collides with B only if A appears before B in the tree (see blog post for detail)

Simultaneous Traversal

- Basic idea: group nearby objects together and traverse tree all at once for these groups
  - "Packet tracing"

- Works really well on CPU

- Worse performance on GPU
  - Individual traversal is faster
  - Divergence is the reason

Tree Construction

Motivation for GPU Tree Construction

- Constantly changing scene -> need to update BVH at every time step

- Could recycle the same BVH and make updates
  - But tree can deteriorate in unpredictable ways over time
  - Results in poor traversal performance
  - Also this is a challenging data structure problem

- Instead, we will recreate the BVH from scratch every time step

Linear BVH (LBVH)

- Basic algorithm:
  - Choose an order for leaf nodes (objects) to appear in the tree
  - Generate internal nodes that respect this order

For more details, see: C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," Computer Graphics Forum, vol. 28, no. 2, 2009.

Linear BVH (LBVH)

- We want objects in close 3D proximity to be nearby in the BVH
- So, sort them along a space-filling curve
- Specifically, we will use a Z-order curve

Z-order Curve

- Maps multidimensional data to 1D
- Preserves locality of data points
- Defined using Morton codes
  - Interleave binary representation of its coordinate values

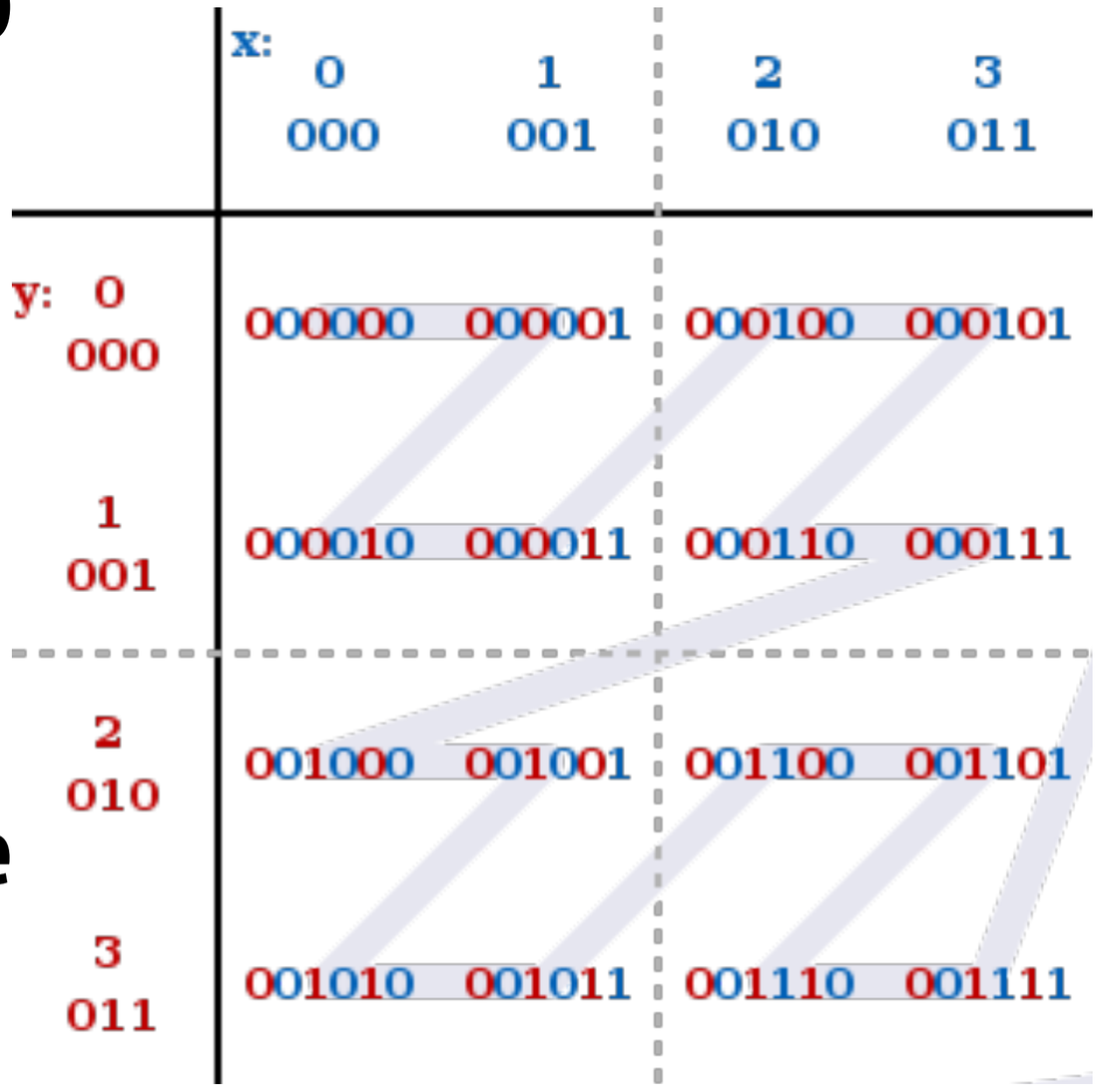


Image from https://en.wikipedia.org/wiki/Z-order_curve

Finding the leaf node order

- Compute Morton code for each object
  - Use the centroid of an object's AABB for its coordinates
  - Interleave the coordinates of the object as described previously
  - (Clever code to compute the codes can be found here: https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-iii-tree-construction-gpu/)

- Sort the Morton codes
  - Radix sort will work well

LBVH - Lauterbach et al. [2009]

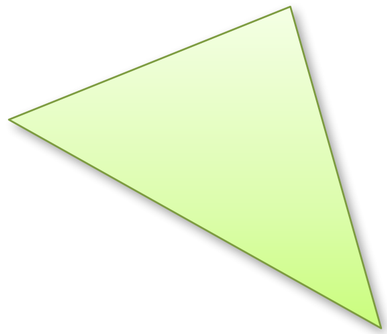
1. Assign Morton codes
2. Sort primitives
3. Generate hierarchy
4. Fit bounding boxes

LBVH - Lauterbach et al. [2009]

1. Assign Morton codes
2. Sort primitives
3. Generate hierarchy
4. Fit bounding boxes

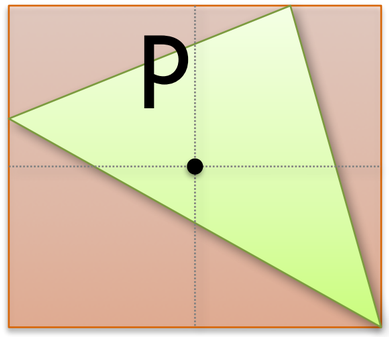
LBVH - Lauterbach et al. [2009]

1. Assign Morton codes
2. Sort primitives
3. Generate hierarchy
4. Fit bounding boxes



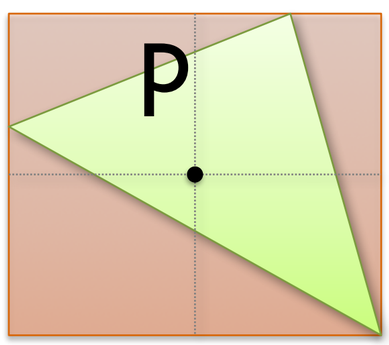
LBVH - Lauterbach et al. [2009]

1. **Assign Morton codes**
2. **Sort primitives**
3. **Generate hierarchy**
4. **Fit bounding boxes**



LBVH - Lauterbach et al. [2009]

1. **Assign Morton codes**
2. **Sort primitives**
3. **Generate hierarchy**
4. **Fit bounding boxes**



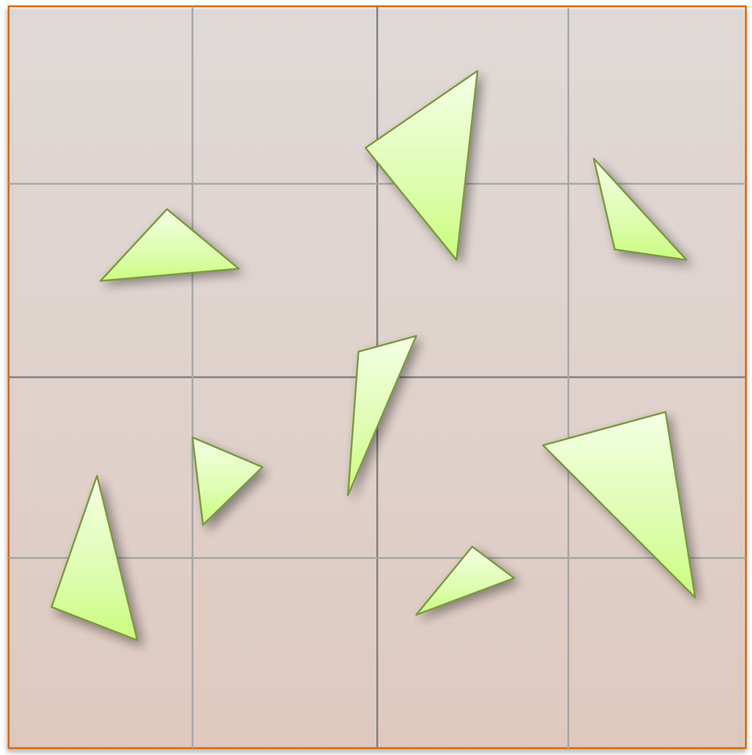
$p_x = 0.\ 1010$
$p_y = 0.\ 0111$
$p_z = 0.\ 1100$

LBVH - Lauterbach et al. [2009]

1. Assign Morton codes
2. Sort primitives
3. Generate hierarchy
4. Fit bounding boxes

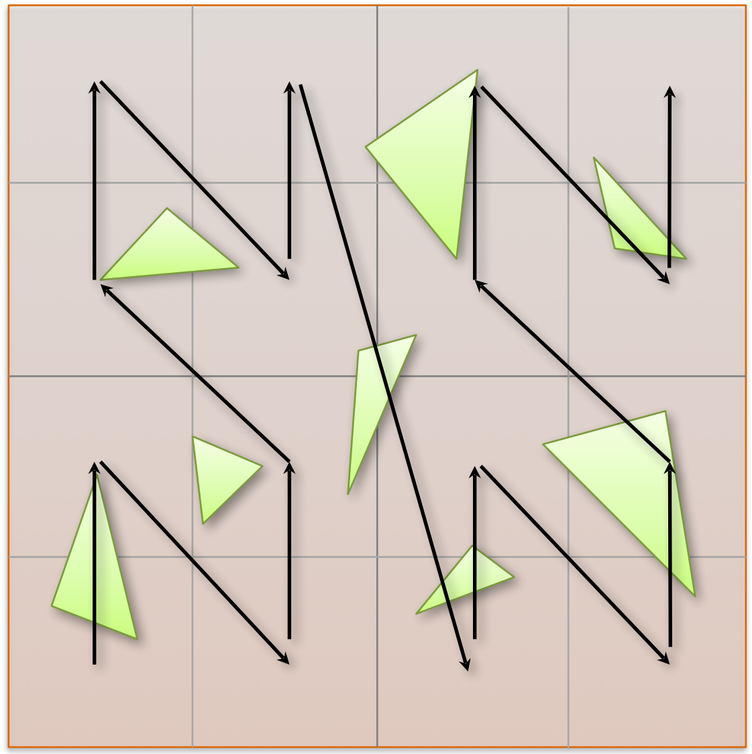
LBVH - Lauterbach et al. [2009]

1. Assign Morton codes
2. Sort primitives
3. Generate hierarchy
4. Fit bounding boxes



LBVH - Lauterbach et al. [2009]

1. Assign Morton codes
2. Sort primitives
3. Generate hierarchy
4. Fit bounding boxes



Finding the leaf node order: Performance

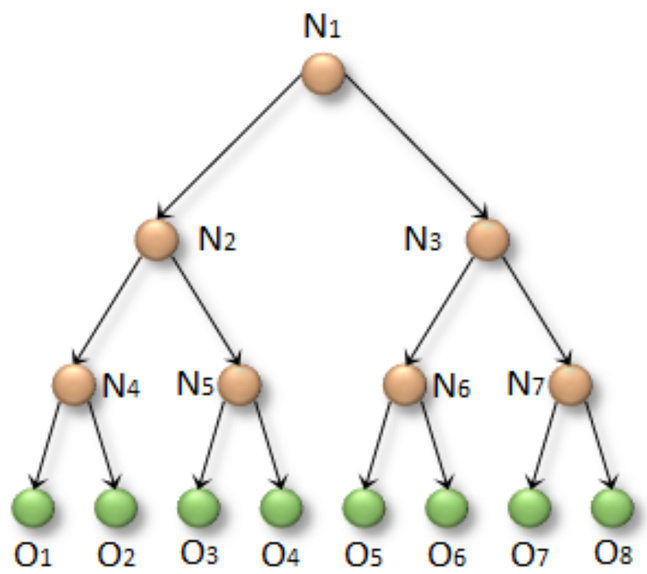
- Compute Morton code for each object
  - 0.02 milliseconds

- Sort the Morton codes
  - 0.18 milliseconds

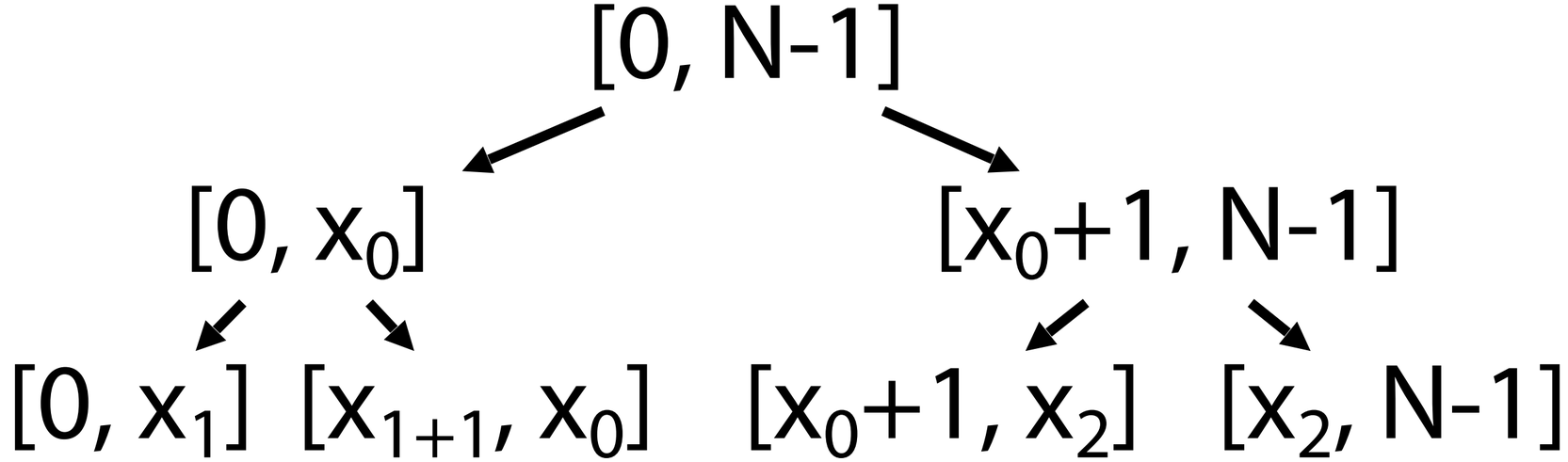
- Tested on GeForce GTX 690
  - Reminder: Traversal execution time is 0.25 milliseconds

Top-Down Hierarchy Generation

- Now that we have the nodes in order, we have to generate the hierarchy (the tree)
- With leaf nodes in order, internal nodes are just linear ranges over them
  - But not necessarily balanced linear ranges!
- N1: [01, 08]
- N2: [01, 04]
- N3: [05, 08]
- N4: [01, 02]
- etc.



Top-Down Hierarchy Generation

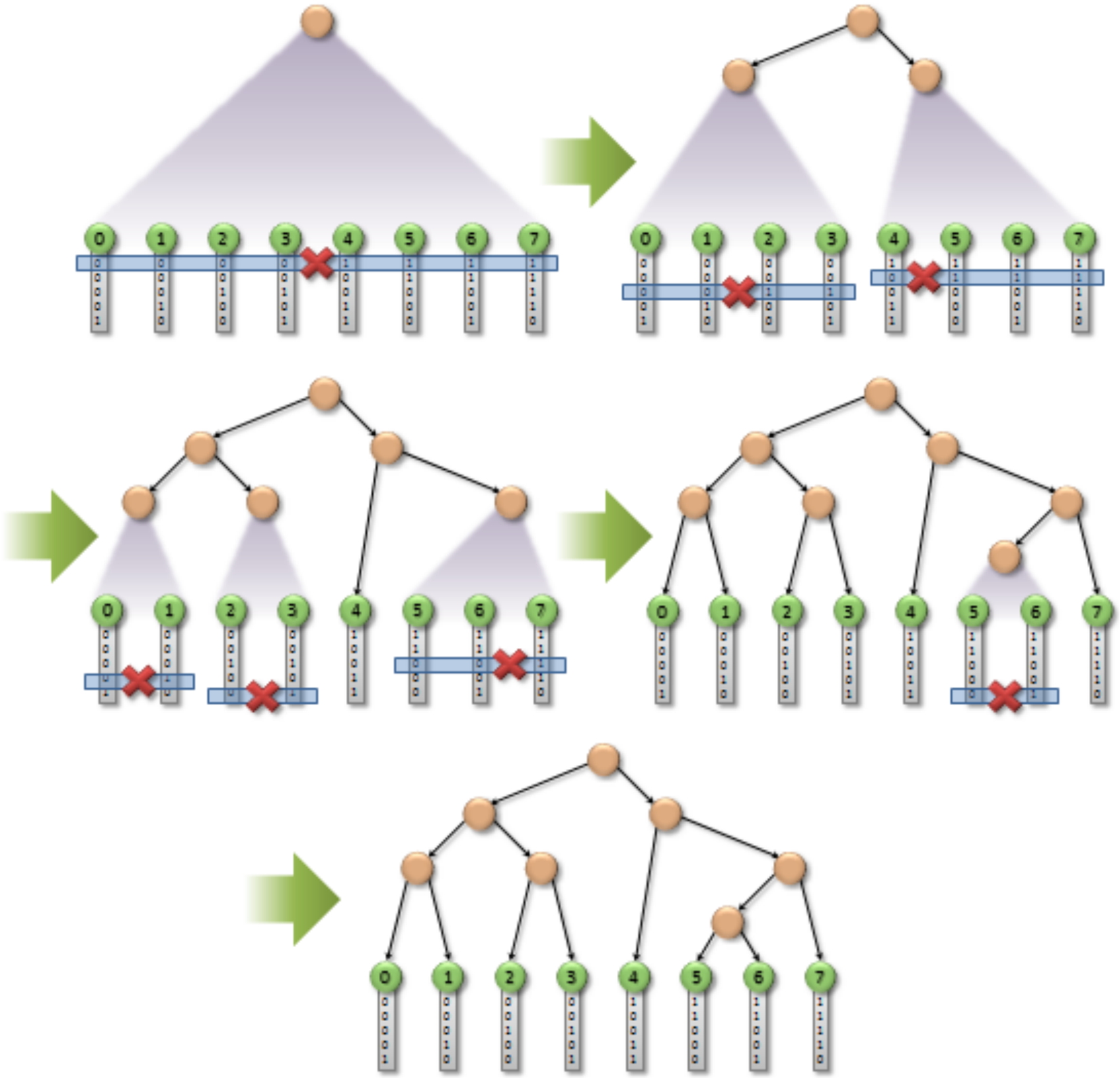


Top-Down Hierarchy Generation

```
Node* generateHierarchy( unsigned int* sortedMortonCodes,
                         int*          sortedObjectIDs,
                         int           first,
                         int           last)
{
    // Single object => create a leaf node.

    if (first == last)
        return new LeafNode(&sortedObjectIDs[first]);

    // Determine where to split the range.

    int split = findSplit(sortedMortonCodes, first, last);

    // Process the resulting sub-ranges recursively.

    Node* childA = generateHierarchy(sortedMortonCodes, sortedObjectIDs,
                                     first, split);
    Node* childB = generateHierarchy(sortedMortonCodes, sortedObjectIDs,
                                     split + 1, last);
    return new InternalNode(childA, childB);
}
```

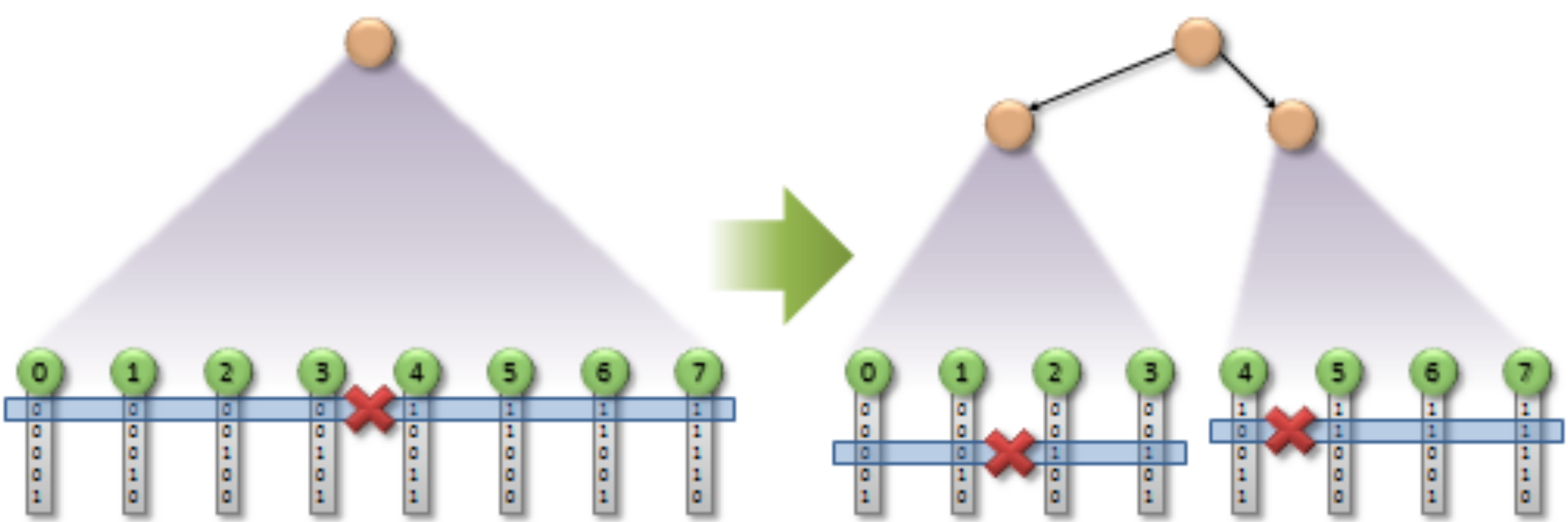
the interesting part!

Top-Down Hierarchy Generation

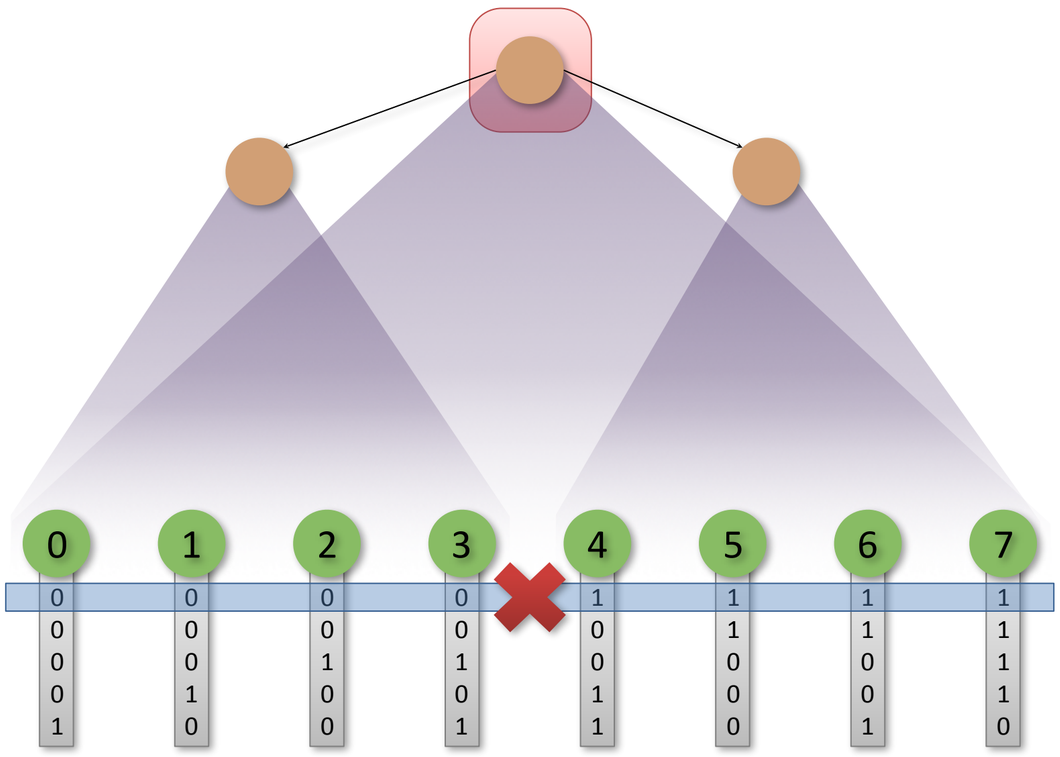


How should we choose the split points?

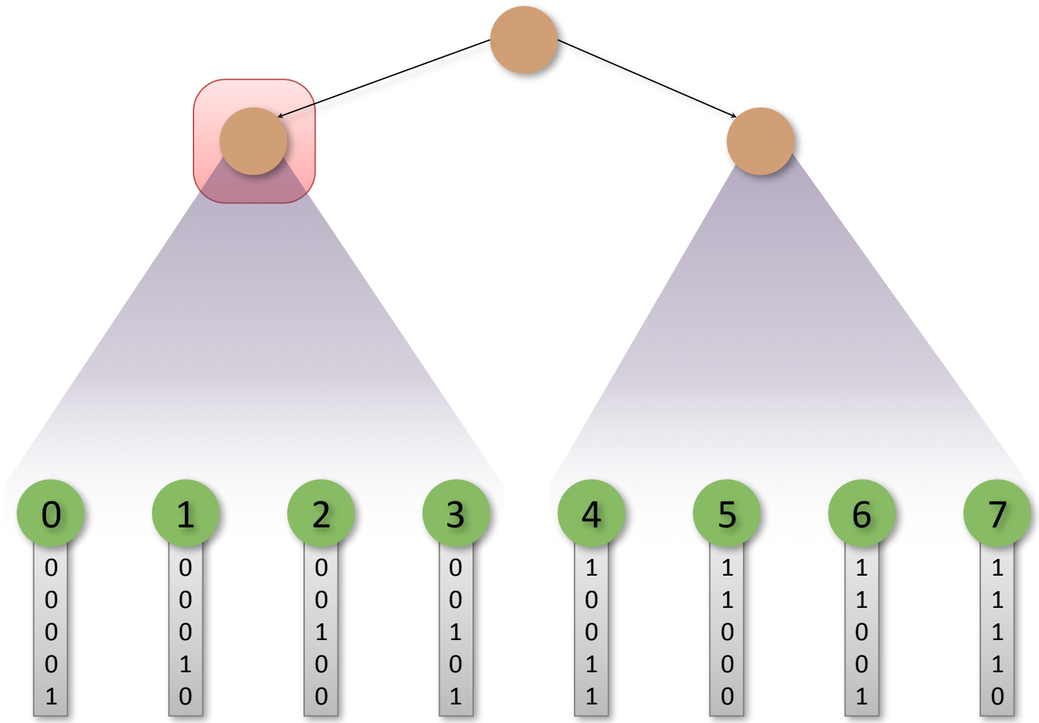
- LBVH chooses split point based on the highest differing bit



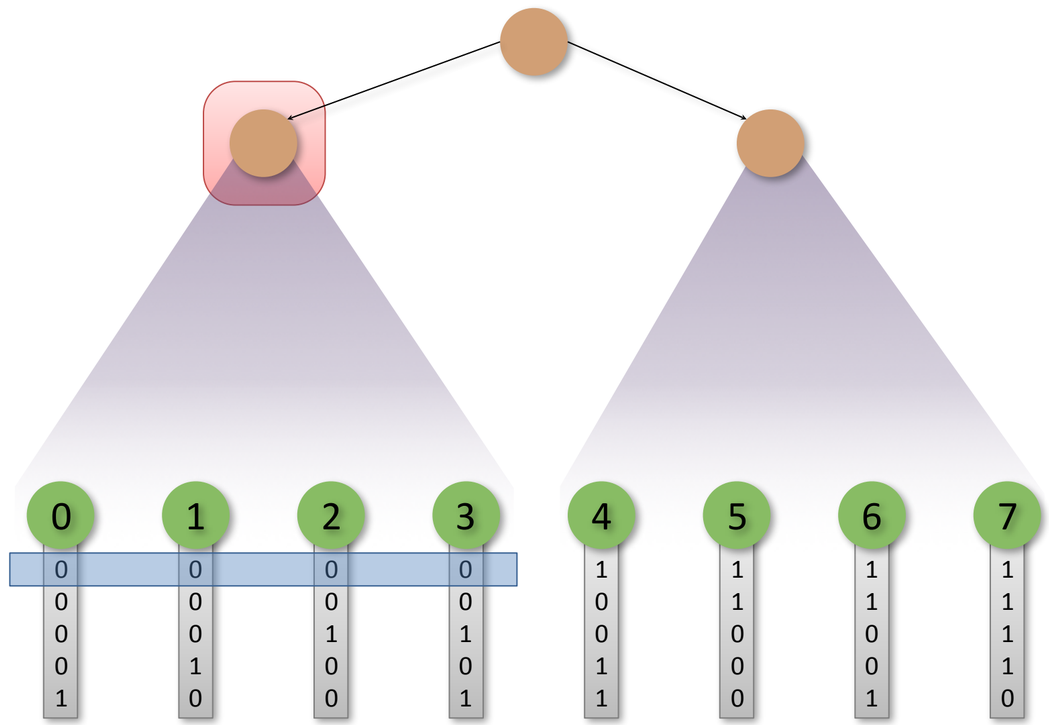
Binary radix tree



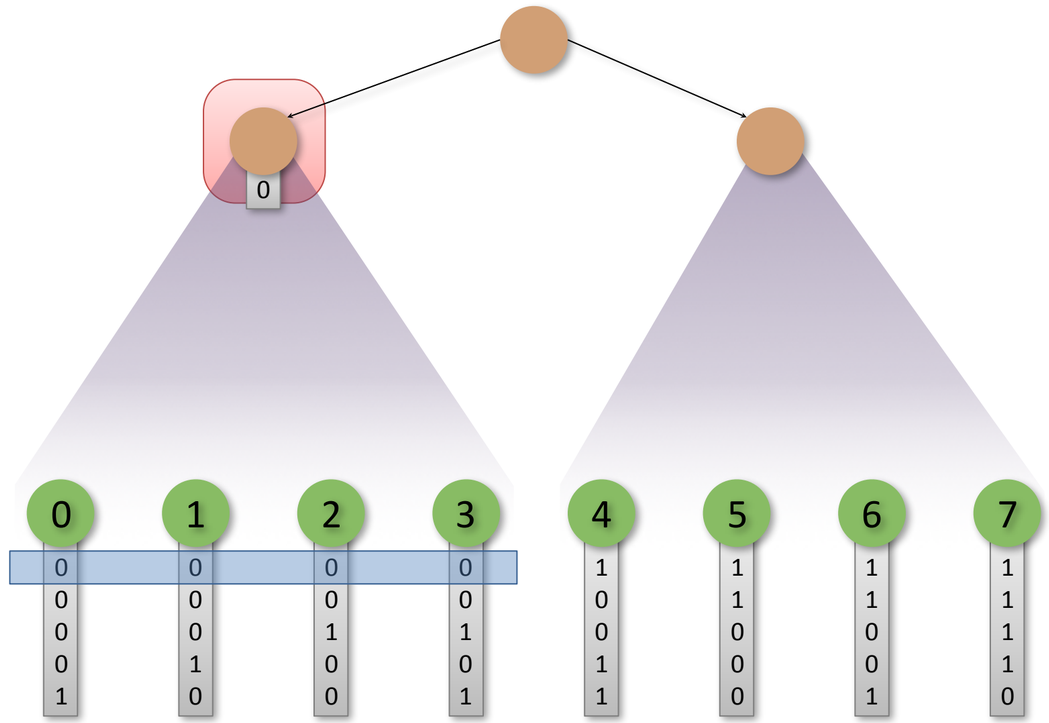
Binary radix tree



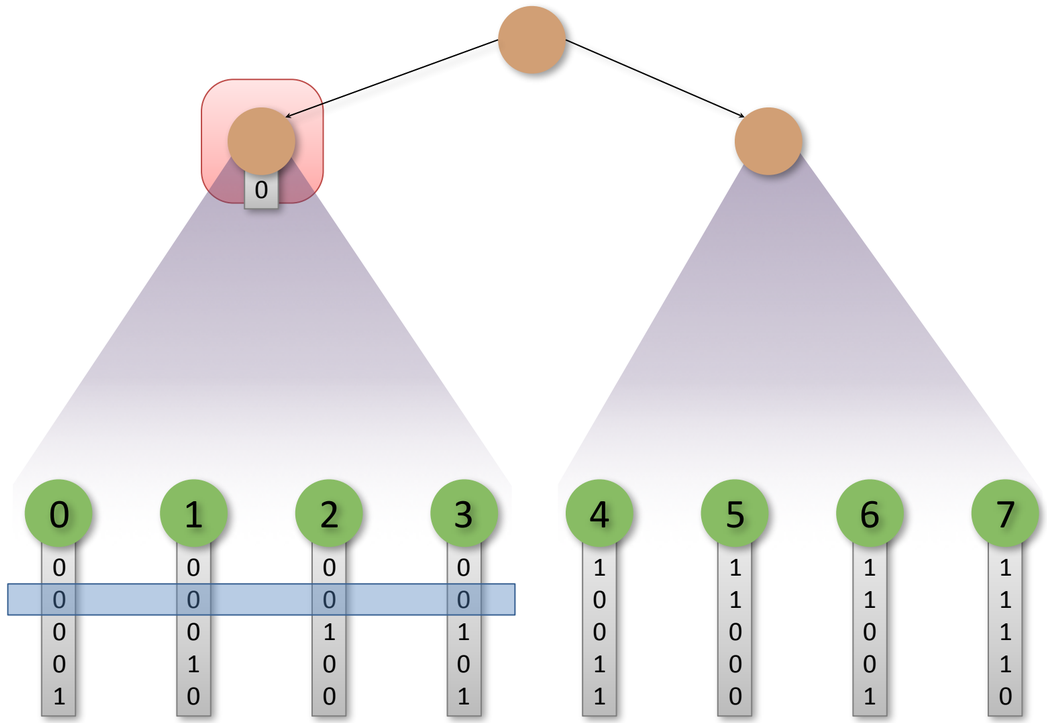
Binary radix tree



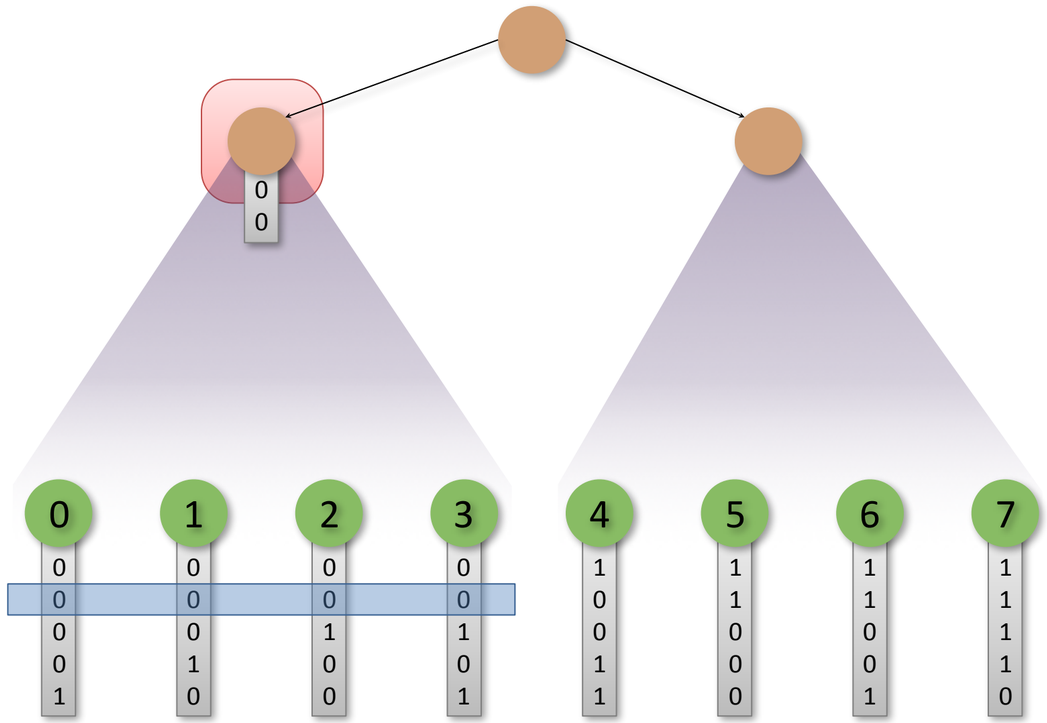
Binary radix tree



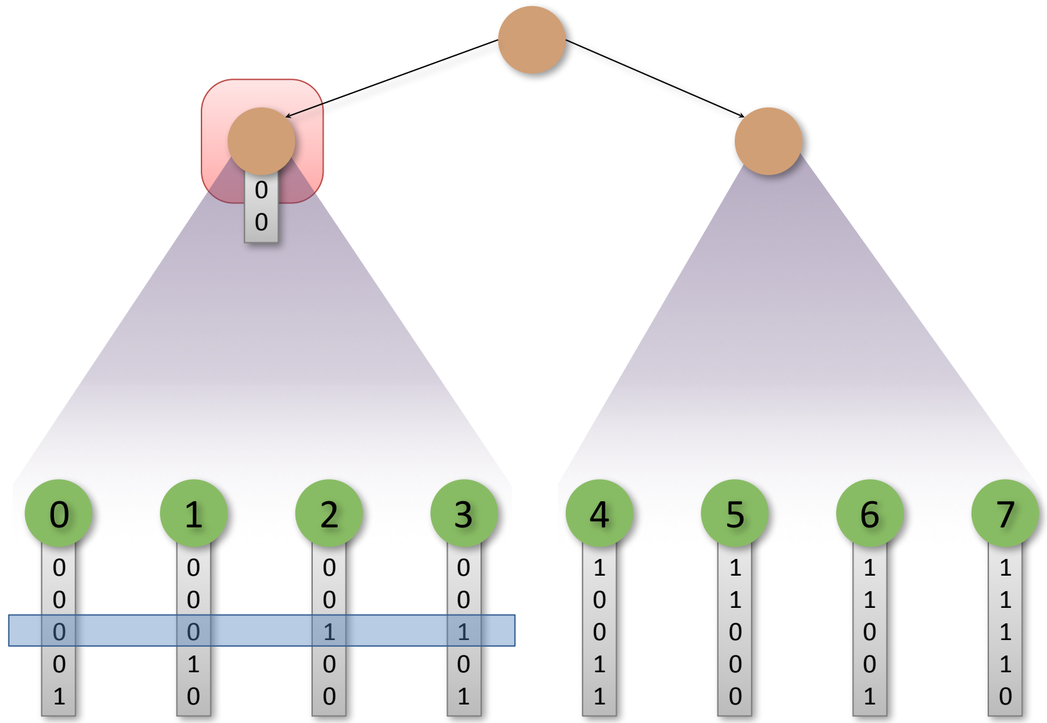
Binary radix tree



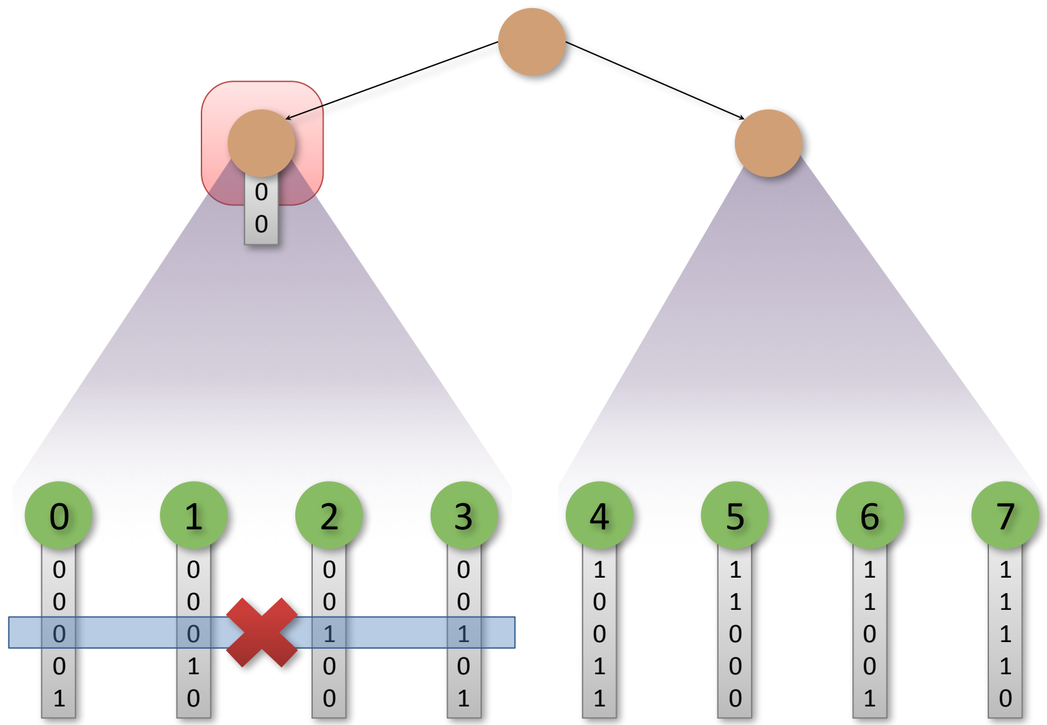
Binary radix tree



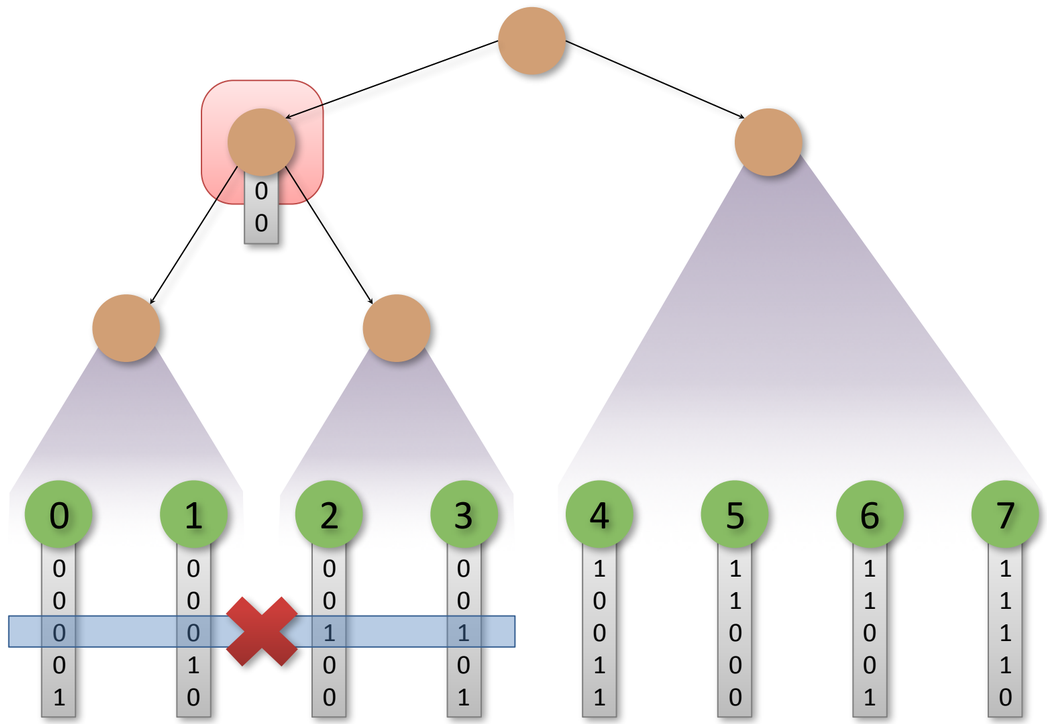
Binary radix tree



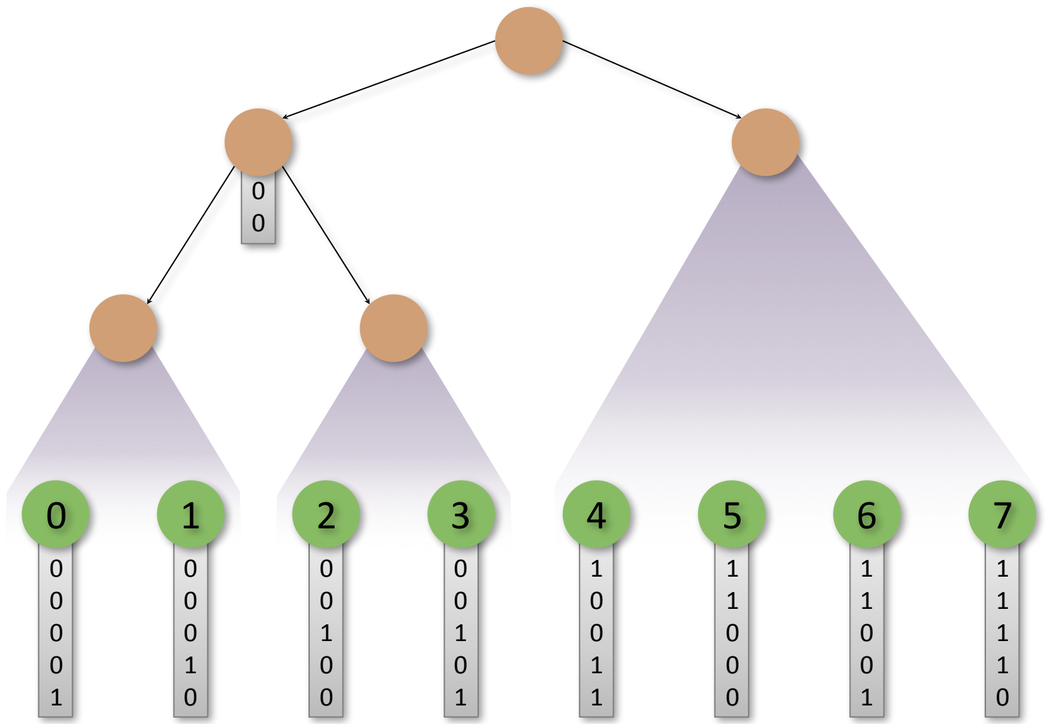
Binary radix tree



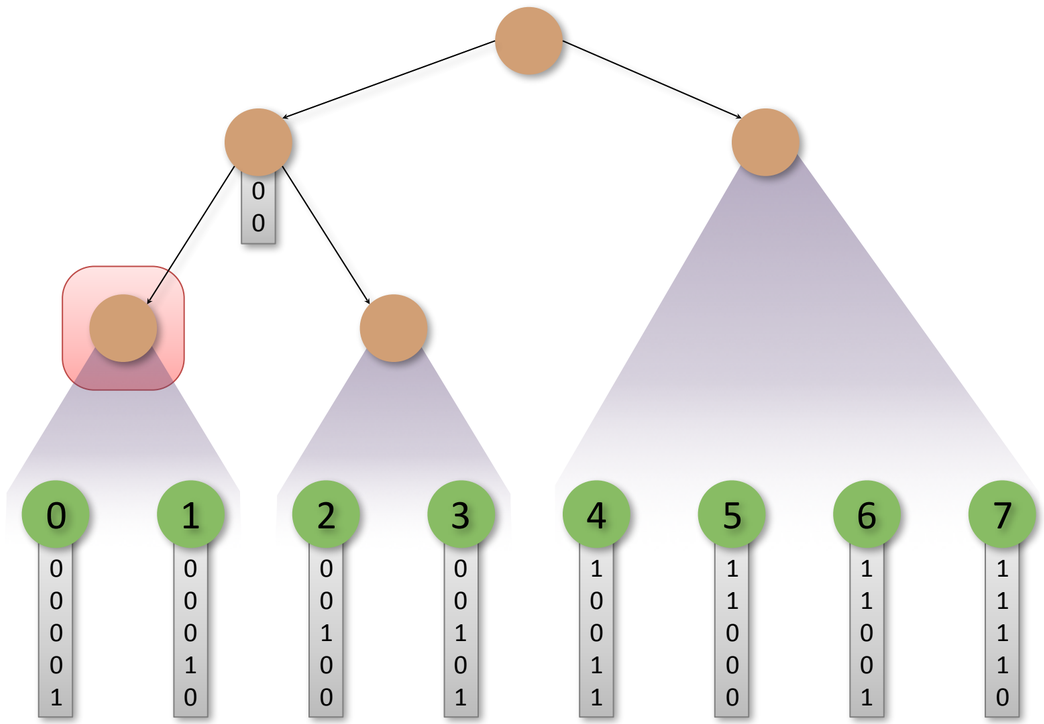
Binary radix tree



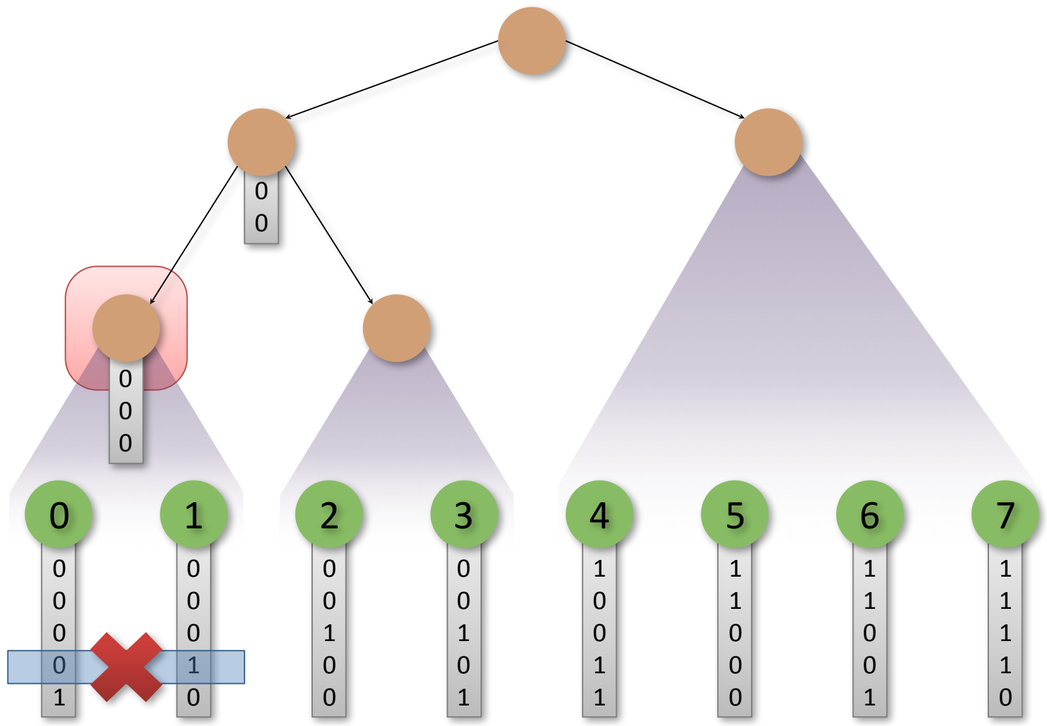
Binary radix tree



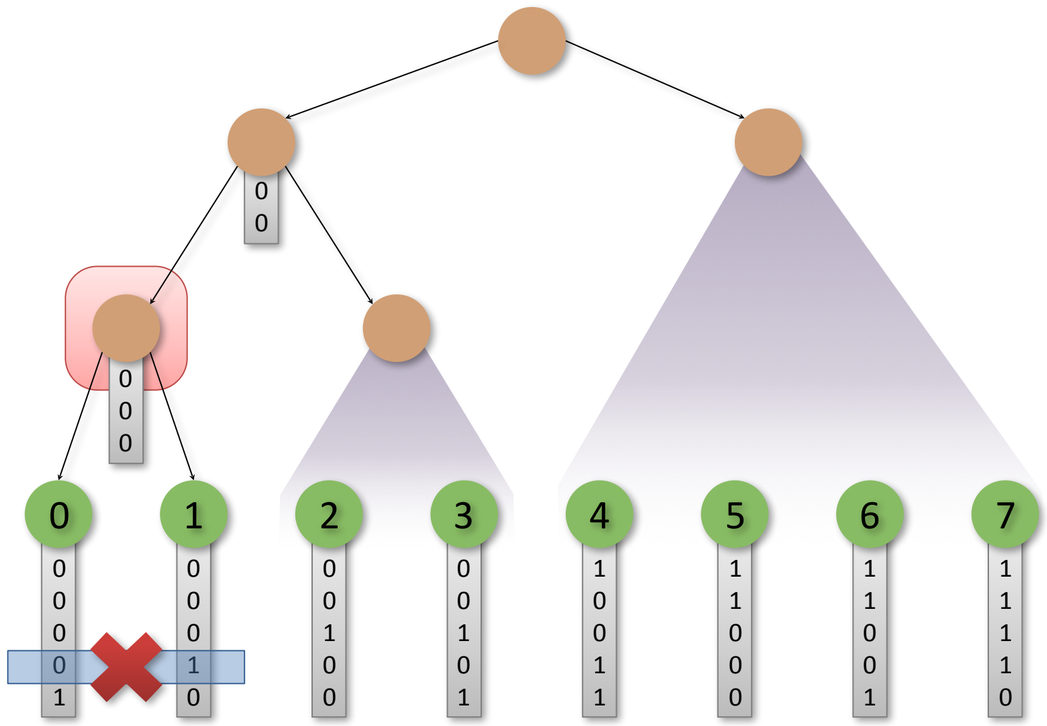
Binary radix tree



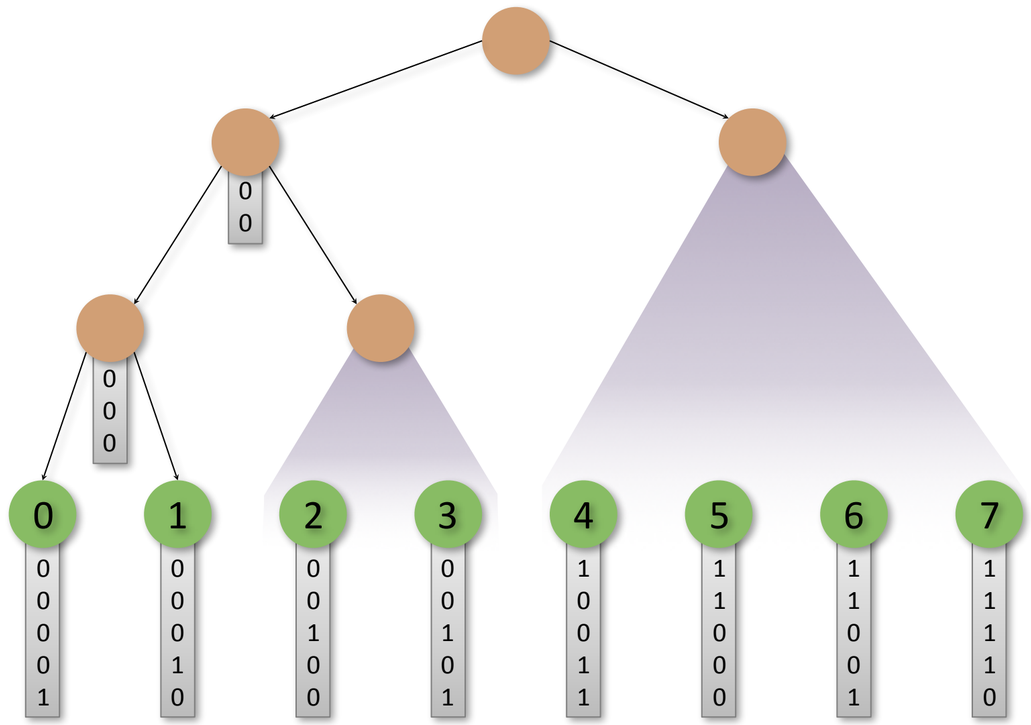
Binary radix tree



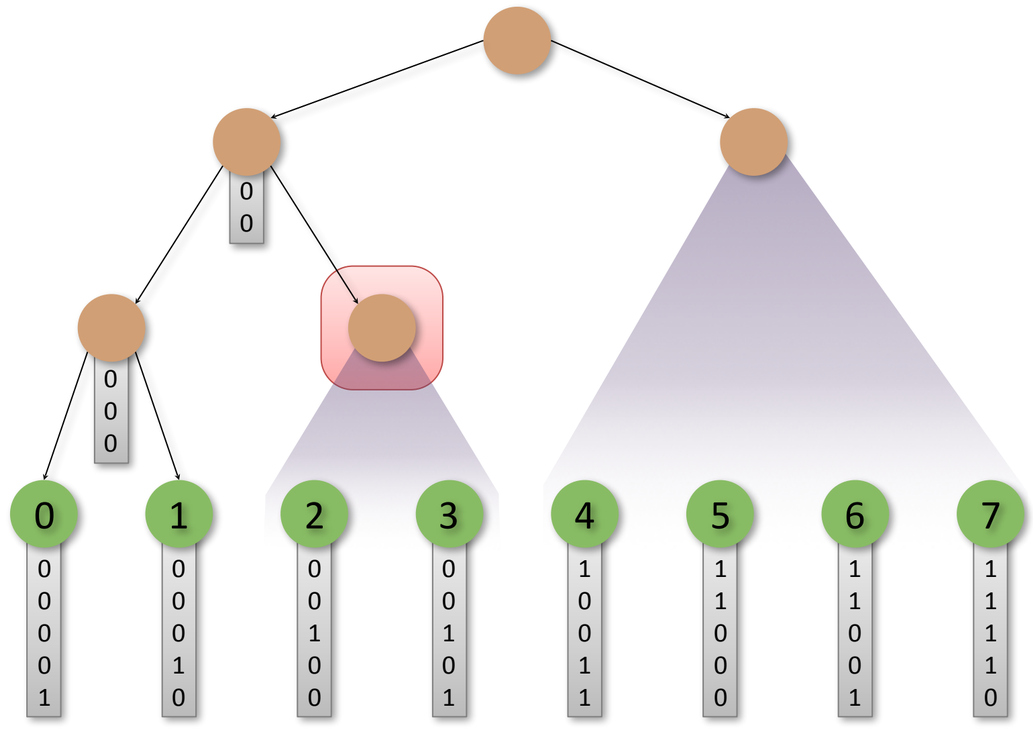
Binary radix tree



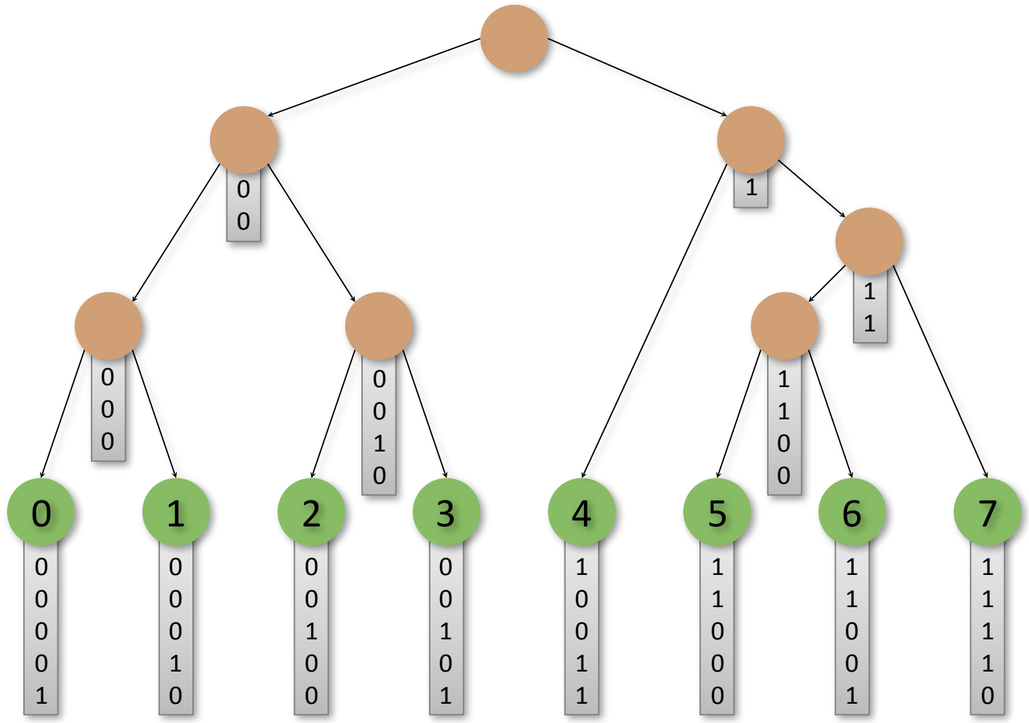
Binary radix tree



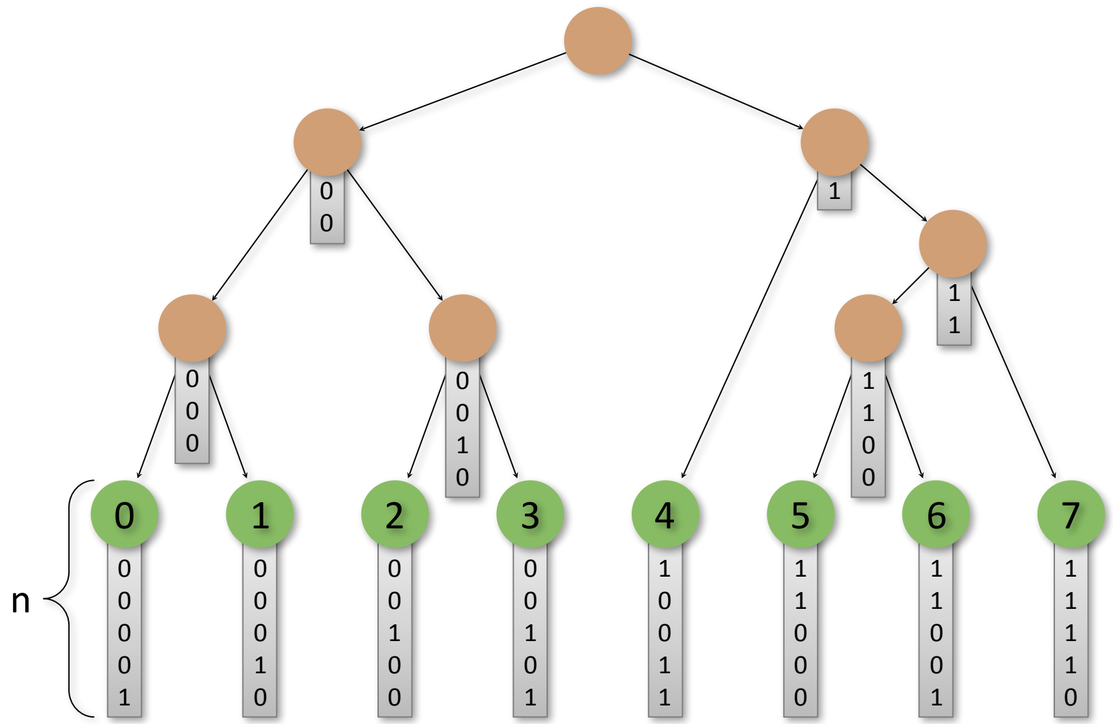
Binary radix tree



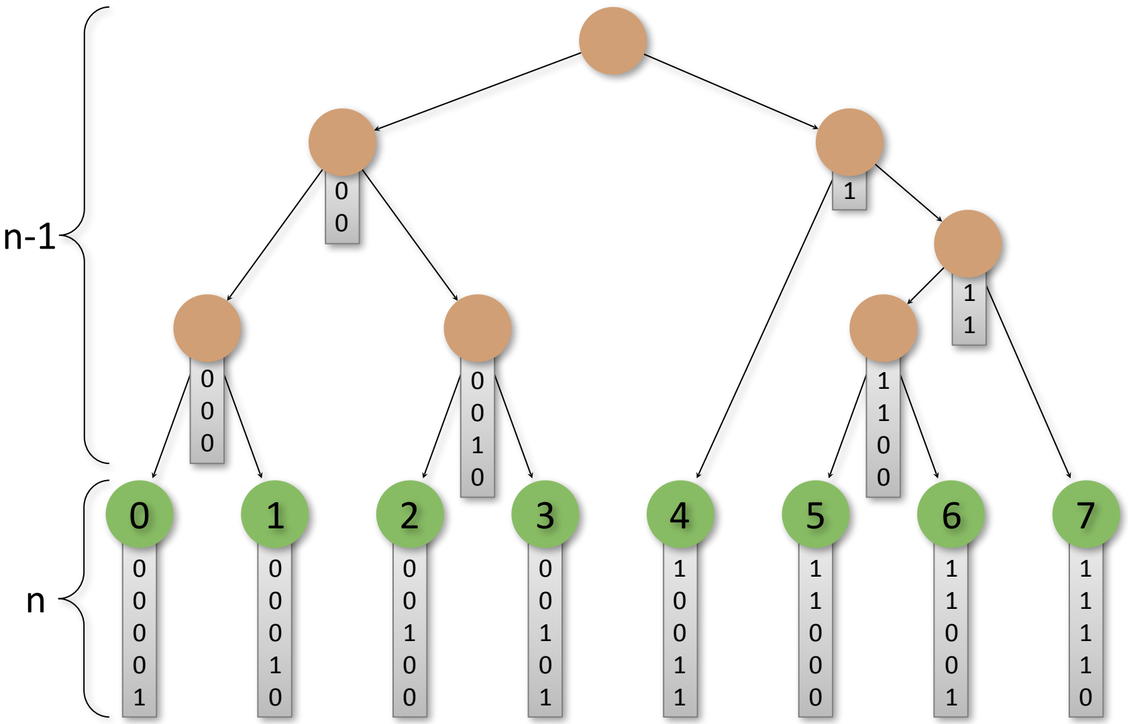
Binary radix tree



Binary radix tree



Binary radix tree



How should we choose the split points?

- LBVH chooses split point based on the highest differing bit
  - Corresponds to classifying them on either size of an axis-aligned plane in 3D

- Find split points one level at a time

- Use binary search to find the highest differing bit

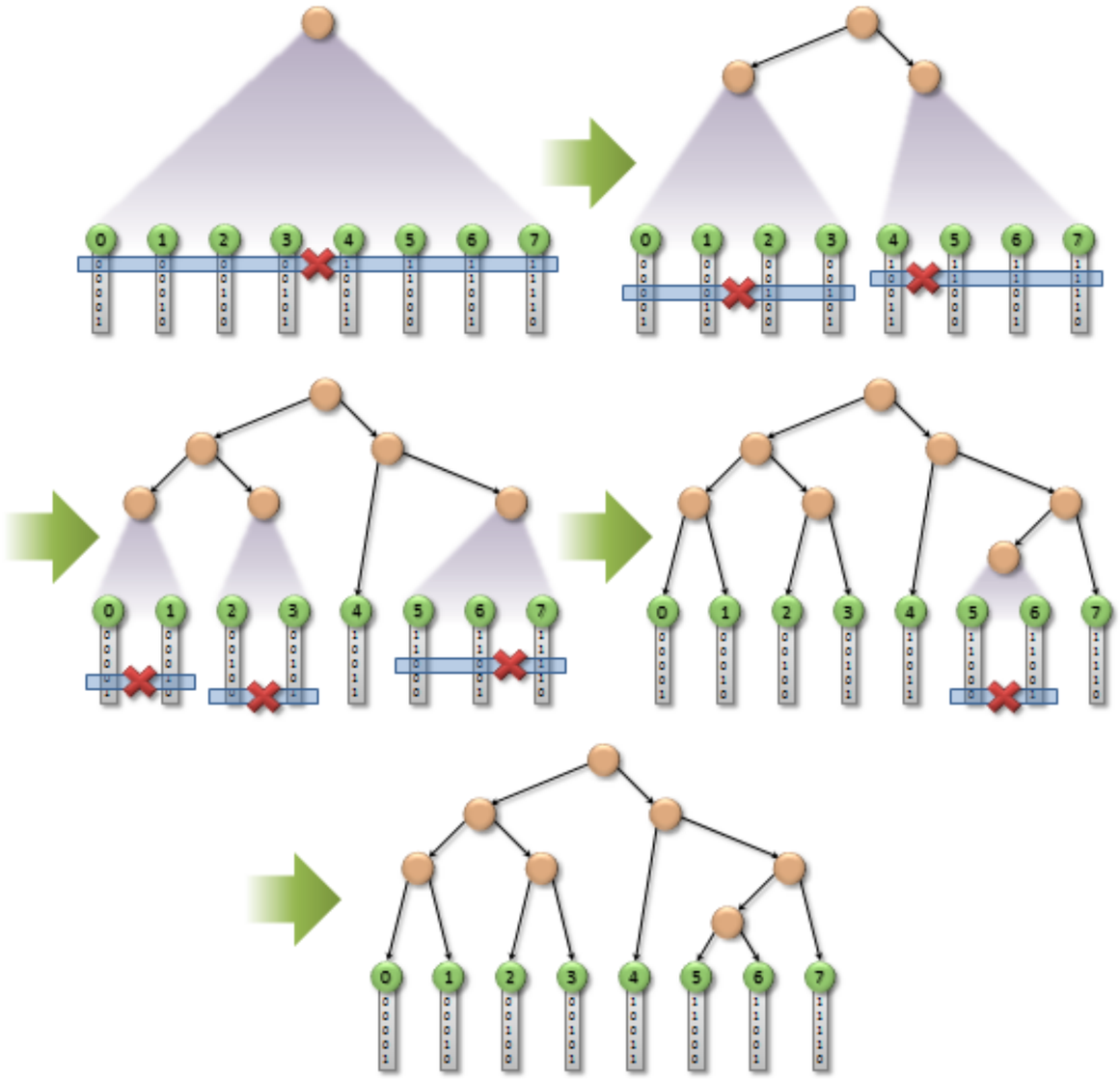
- Some clever code to find the split points can be found here: https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-iii-tree-construction-gpu/

Parallel BVH Construction

- Process each level of nodes sequentially, starting from root

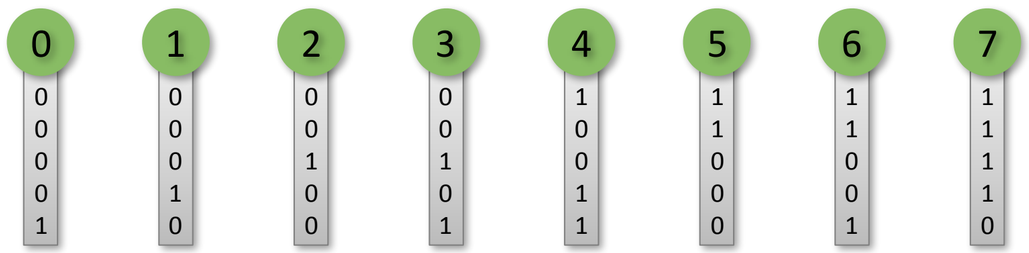
- Launch one thread per node in the level
  - Read first and last from input array
  - Call findSplit()
  - Append resulting child nodes to array using atomic counter
  - Write out corresponding sub-ranges

- Synchronize and repeat for subsequent levels

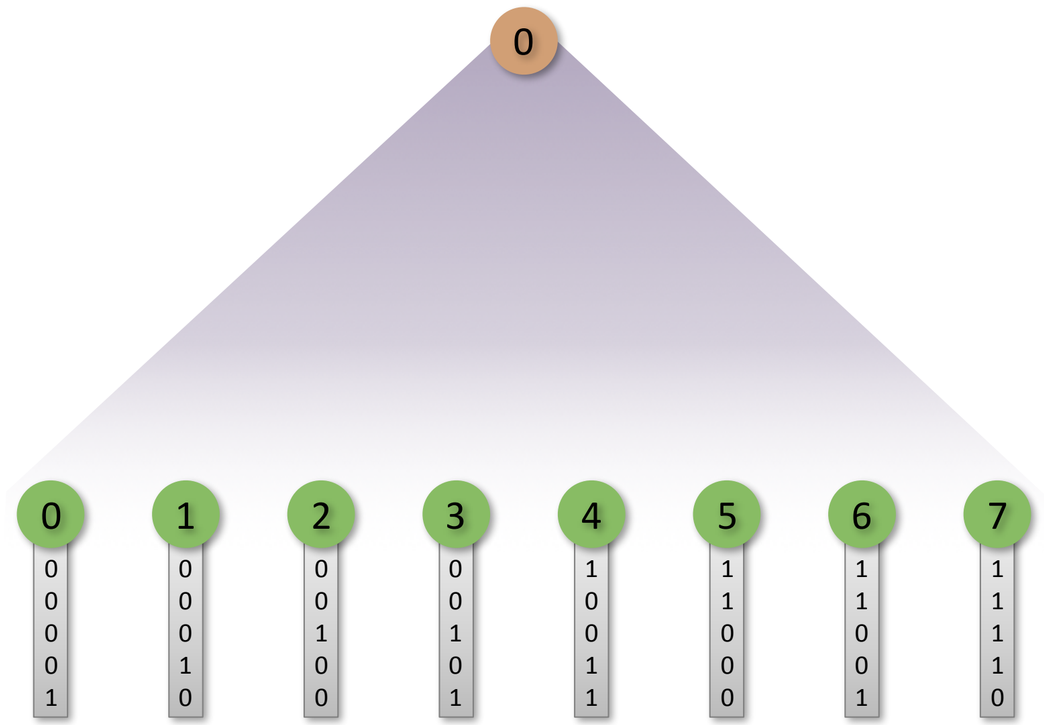


For more details, see: K. Garanzha, J. Pantaleoni, and D. McAllister, "Simpler and faster HLBVH with work queues," HPG'11: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics

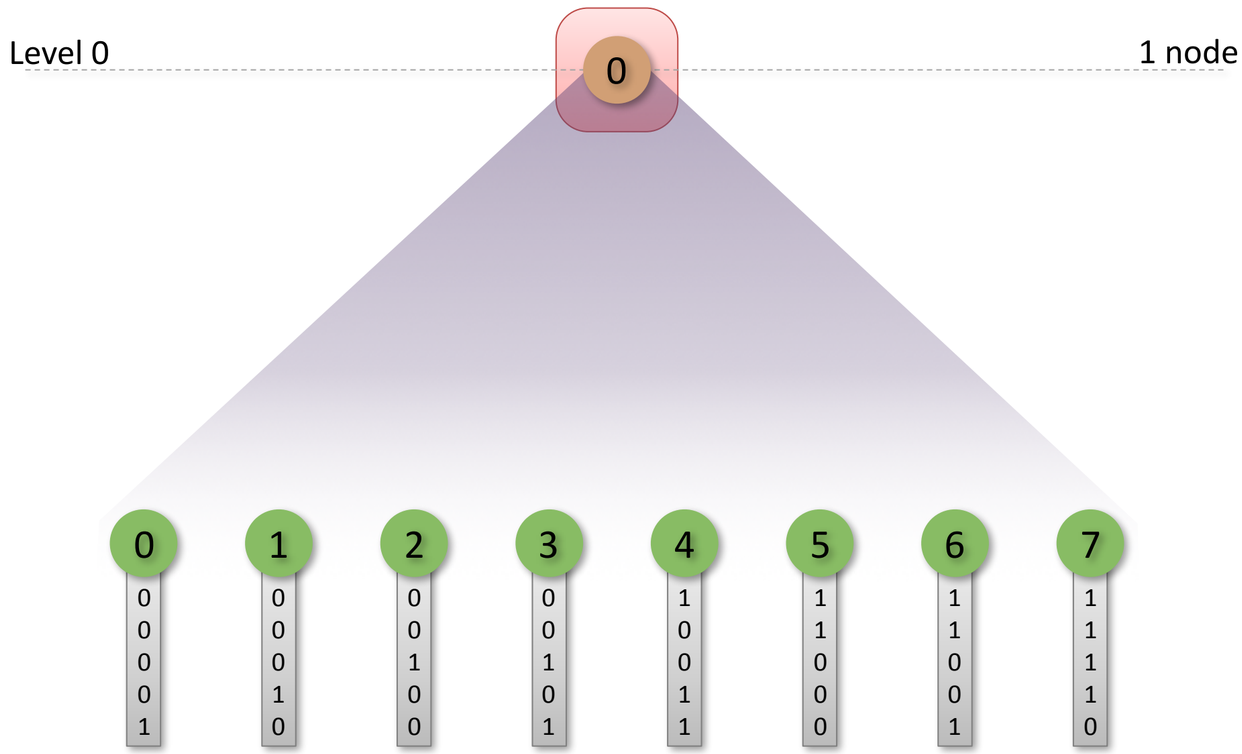
Garanzha et al. [2011]



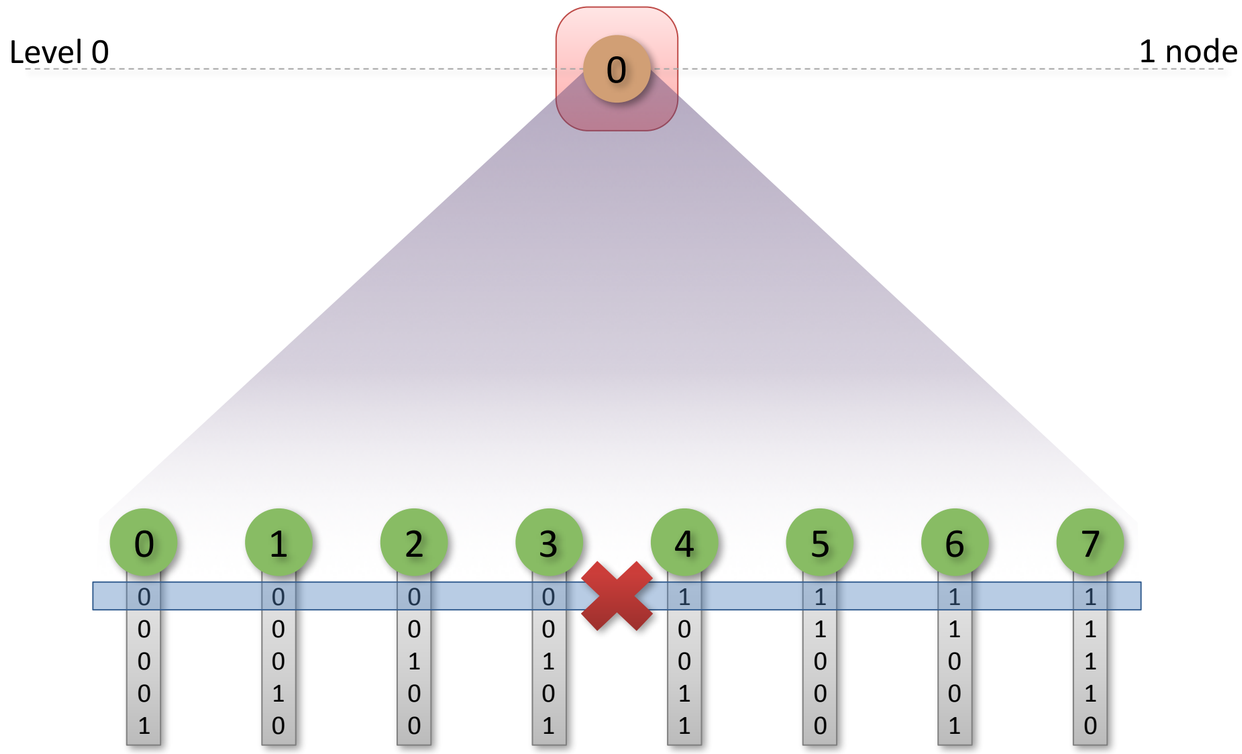
Garanzha et al. [2011]



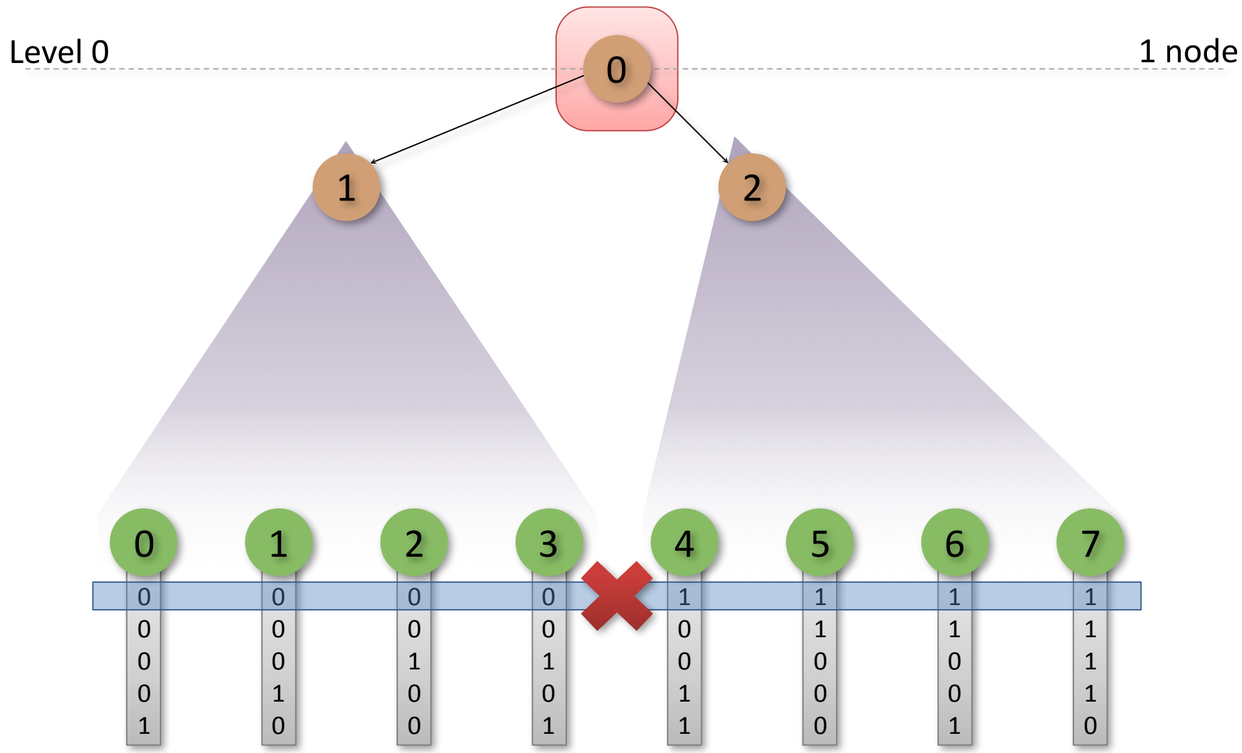
Garanzha et al. [2011]



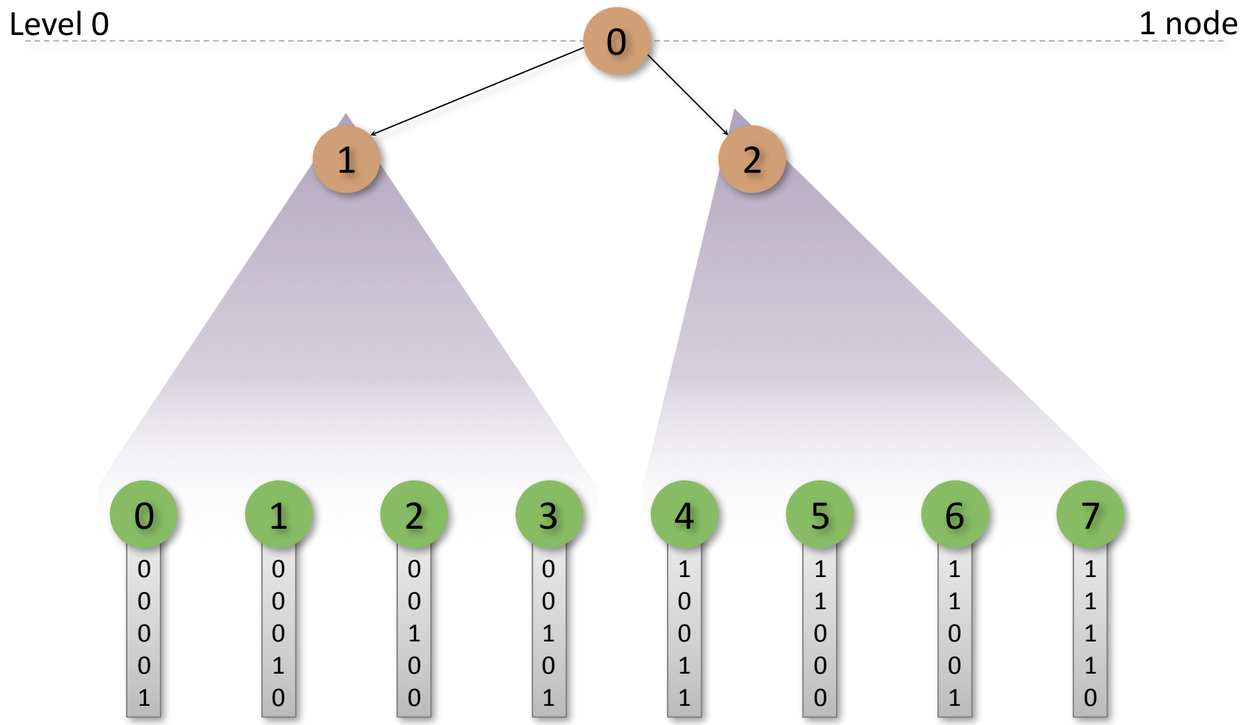
Garanzha et al. [2011]



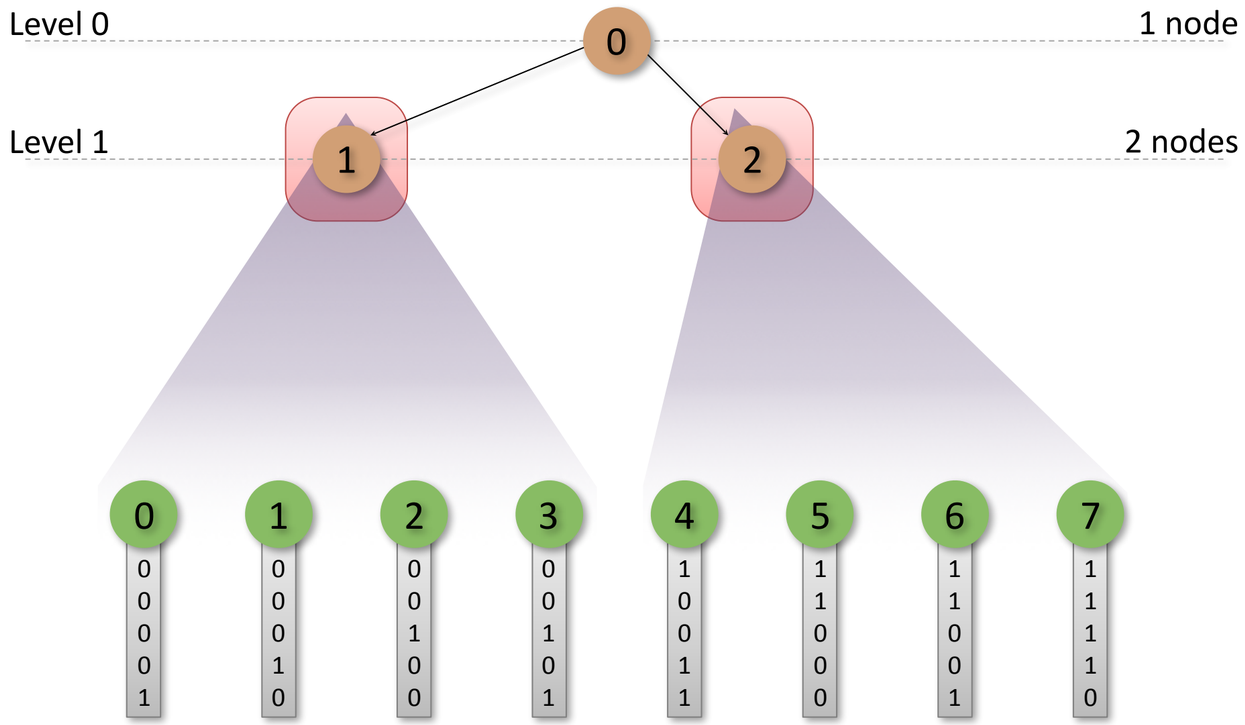
Garanzha et al. [2011]



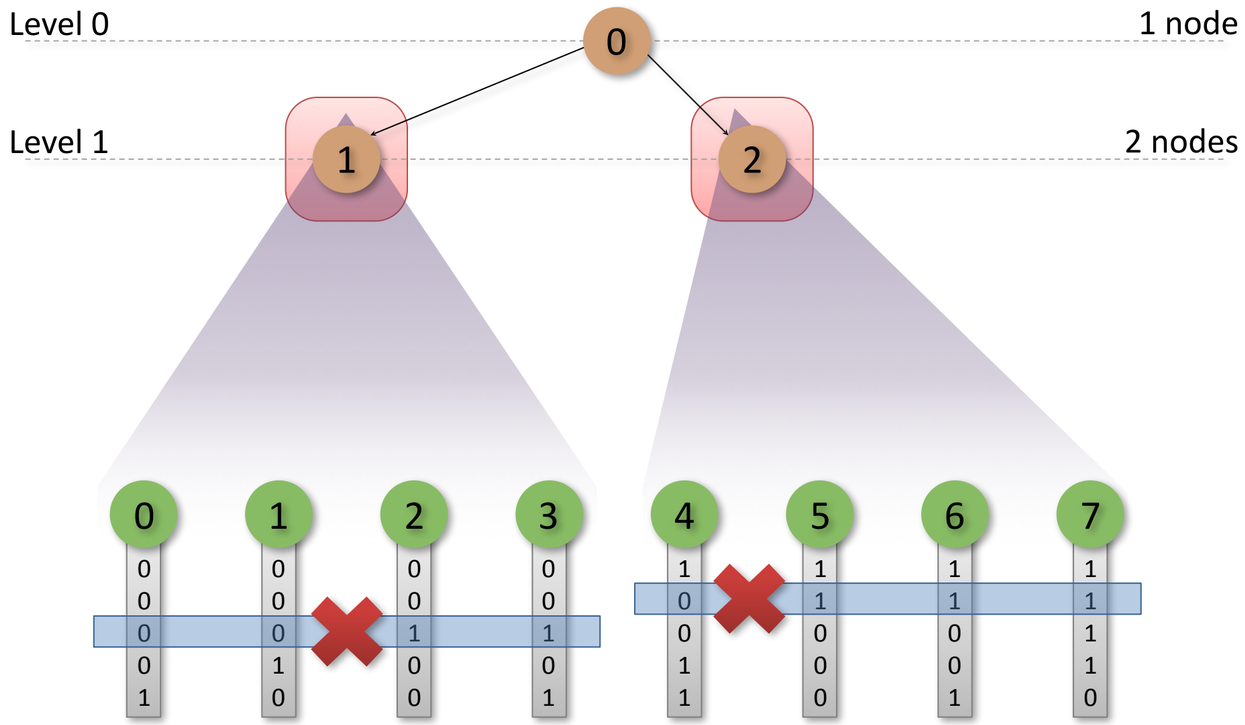
Garanzha et al. [2011]



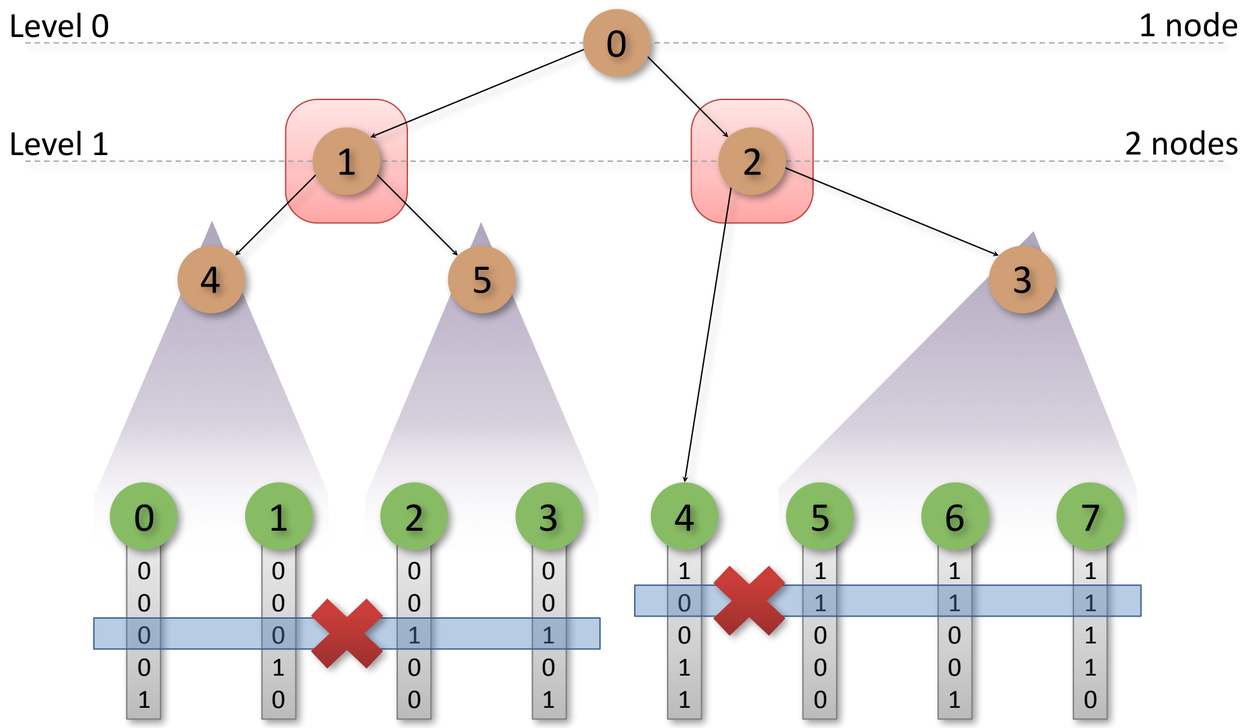
Garanzha et al. [2011]



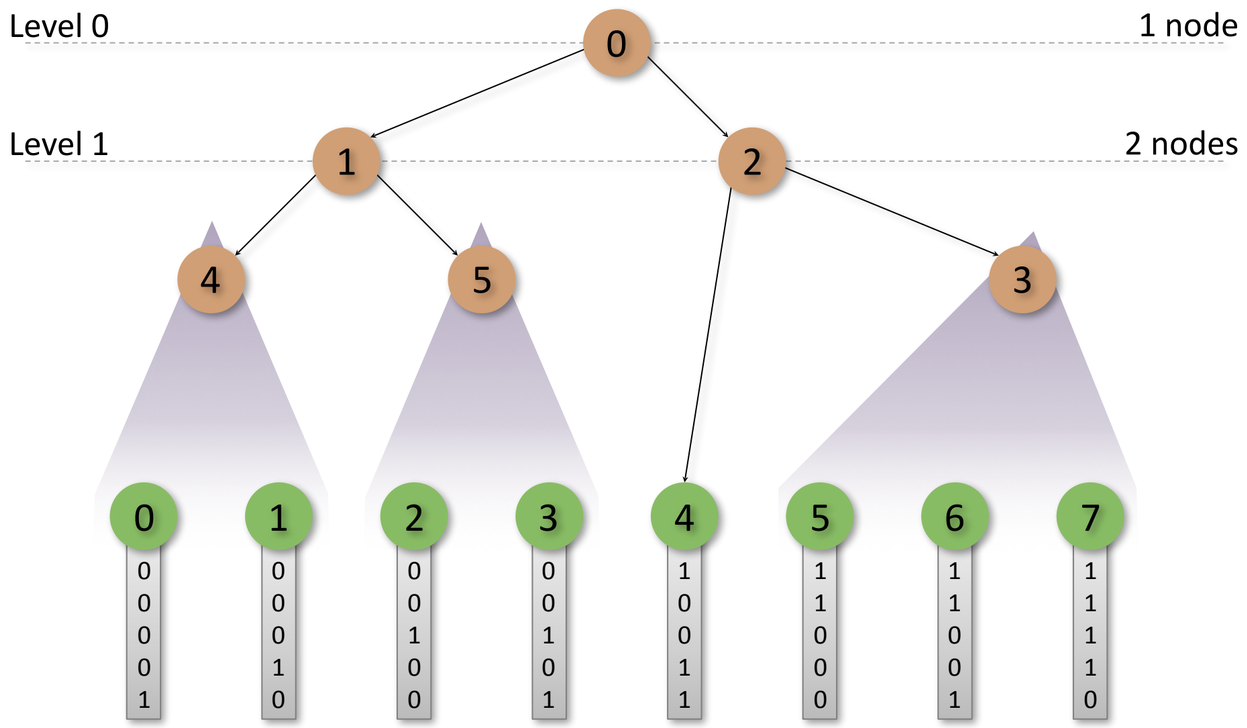
Garanzha et al. [2011]



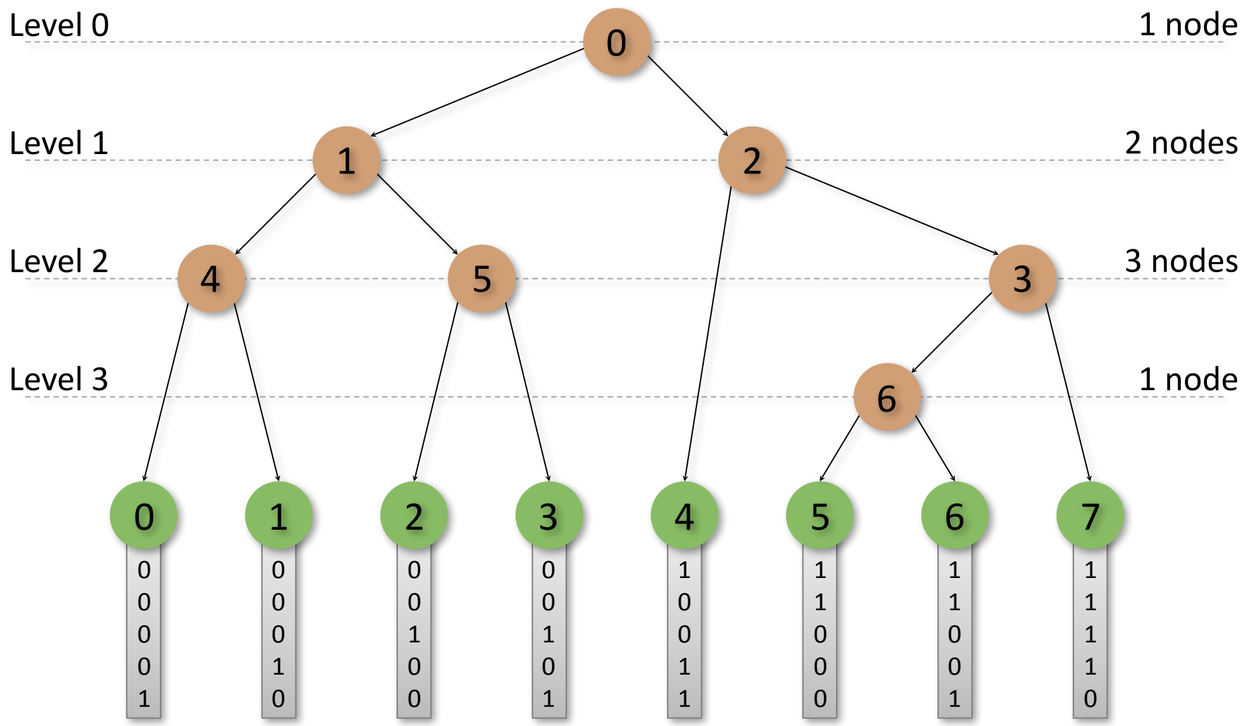
Garanzha et al. [2011]



Garanzha et al. [2011]



Garanzha et al. [2011]



Parallel BVH Construction Performance

- When there are millions of objects, it is very fast
  - Algorithm spends majority of execution time the bottom levels of the tree, which contains plenty of work to fill the GPU
  - Some divergence at higher tree levels, but these levels take significantly less time considering overall execution time (Amdahl's Law)

- Less efficient for fewer objects

Parallel BVH Construction Performance

- When there are millions of objects, it is very fast
  - Algorithm spends majority of execution time the bottom levels of the tree, which contains plenty of work to fill the GPU
  - Some divergence at higher tree levels, but these levels take significantly less time considering overall execution time (Amdahl's Law)

- Less efficient for fewer objects
  - E.g., 12K objects from the previous example
  - Execution time: 1.04 milliseconds

Occupancy

- # threads executing / maximum # threads the processor can support

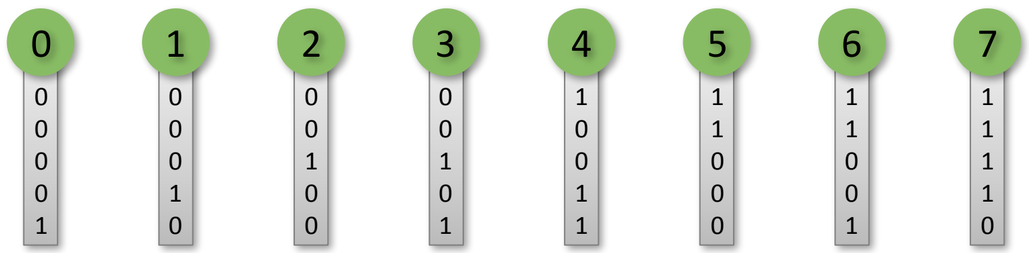
- Want to keep occupancy high
  - Then performance is limited by other factors
  - E.g., instruction throughput, memory bandwidth

Parallel BVH Construction Occupancy

- GeForce GTX 690 has maximum of 16k threads
- Application has 12k objects

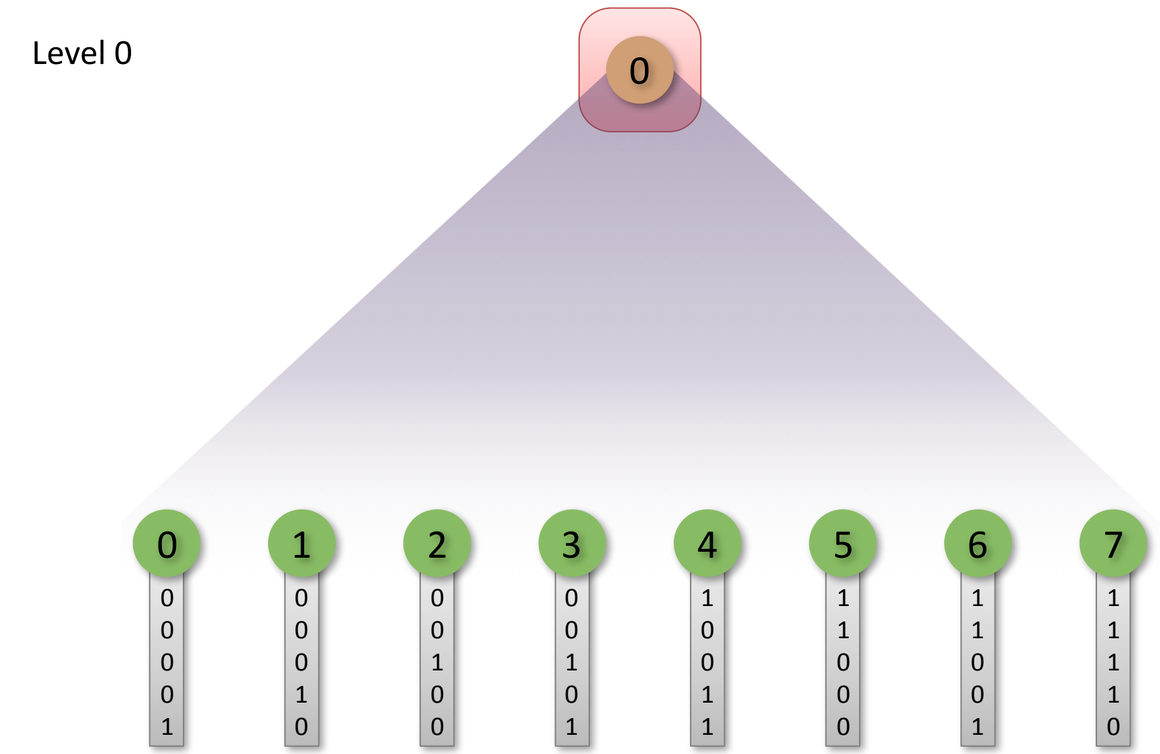
Tree Level	Number of active	Occupancy
0 (Root)	1	0.006%
1	2	0.013%
2	4	0.025%
3	8	0.050%
…	…	…
12	4,096	25.6%
13	8,192	52.2%
14	12,000	75%

Garanzha et al.: Challenge



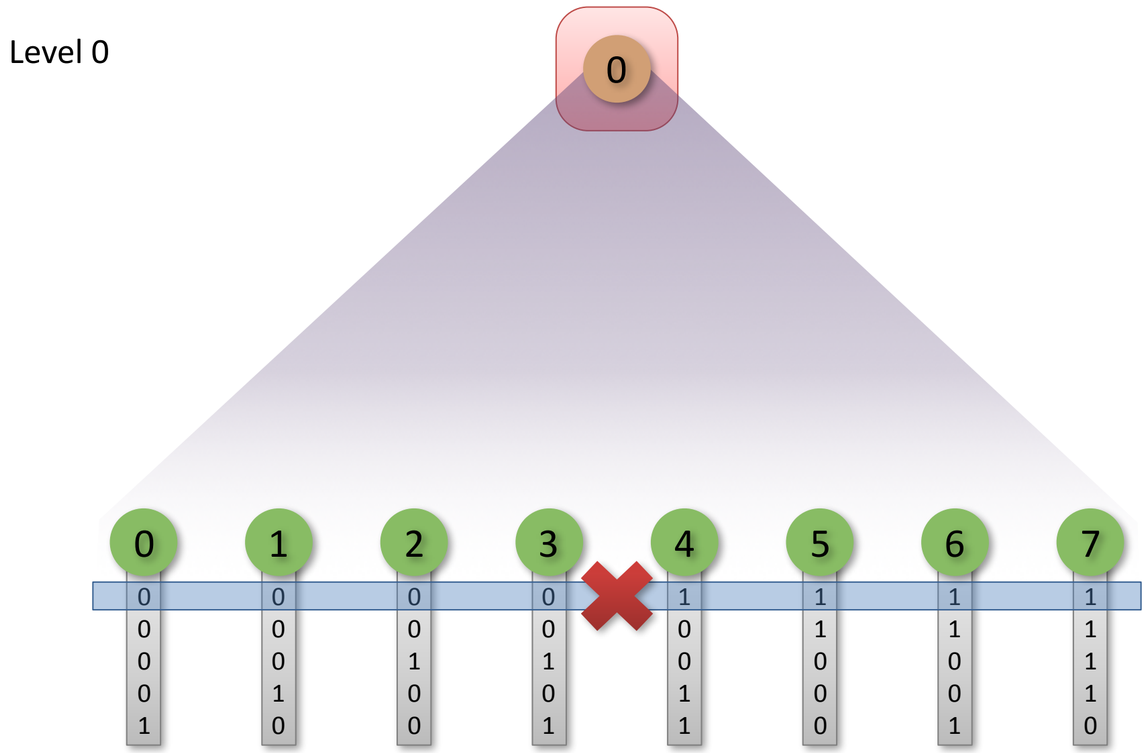
- Building nodes seems serial by level
- Could we generate *all* internal nodes at once?

Garanzha et al.: Challenge



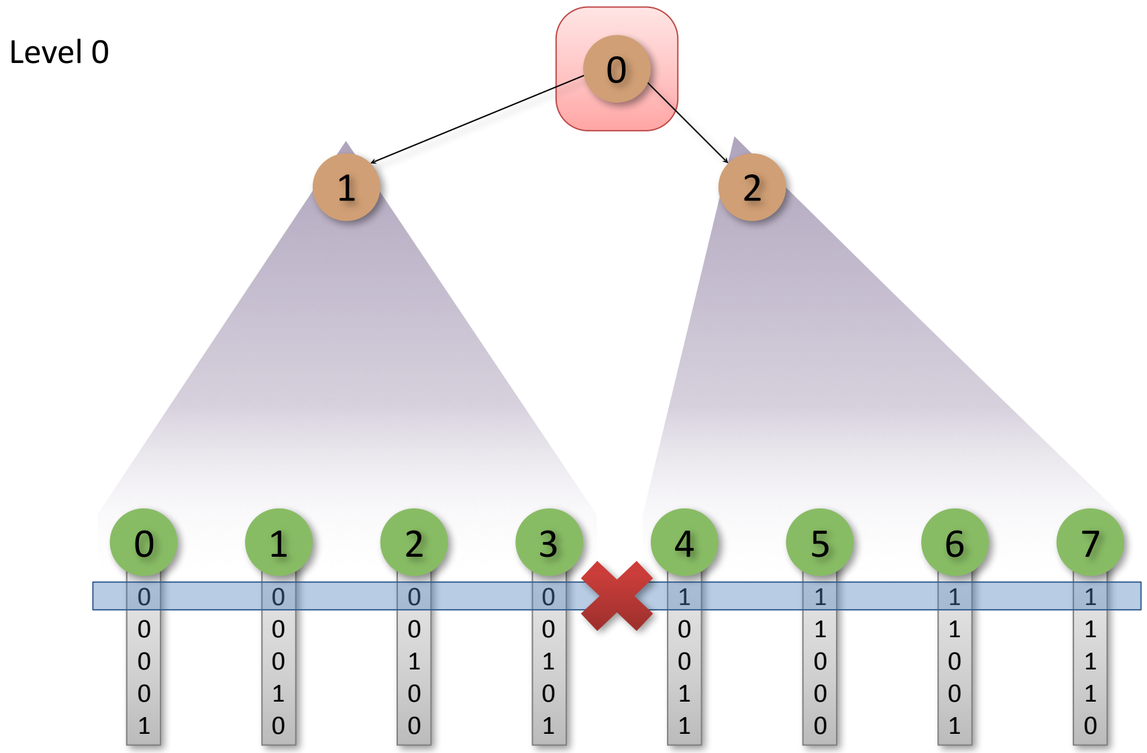
- Building nodes seems serial by level
- Could we generate *all* internal nodes at once?

Garanzha et al.: Challenge



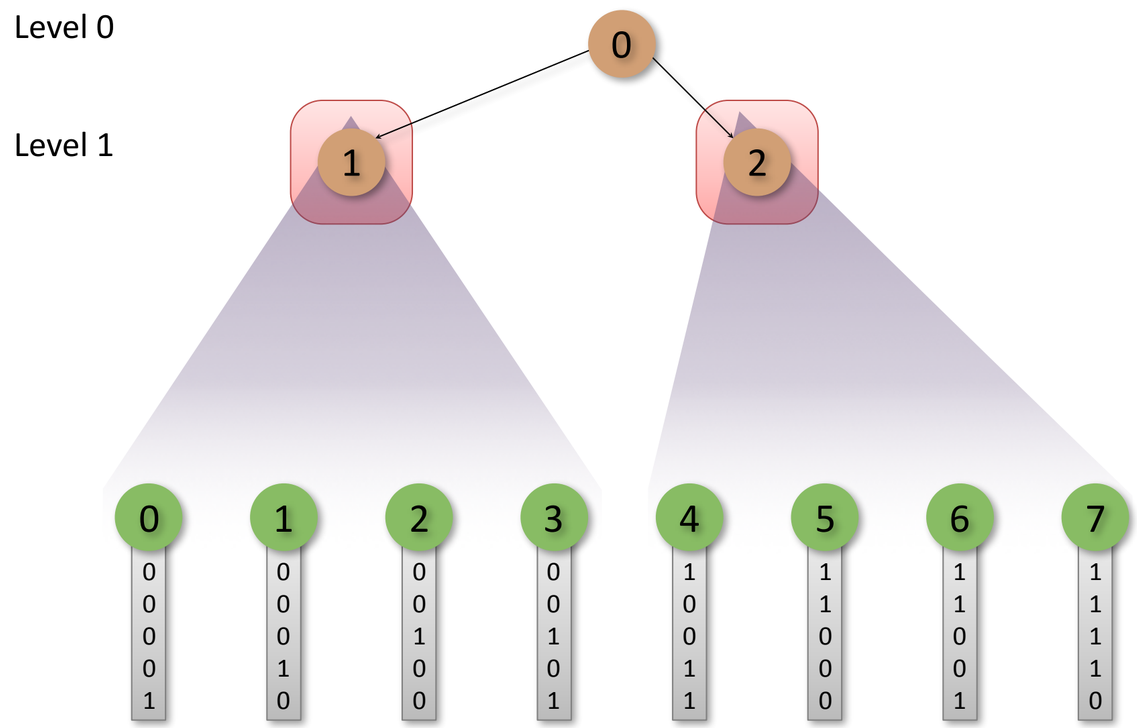
- Building nodes seems serial by level
- Could we generate *all* internal nodes at once?

Garanzha et al.: Challenge



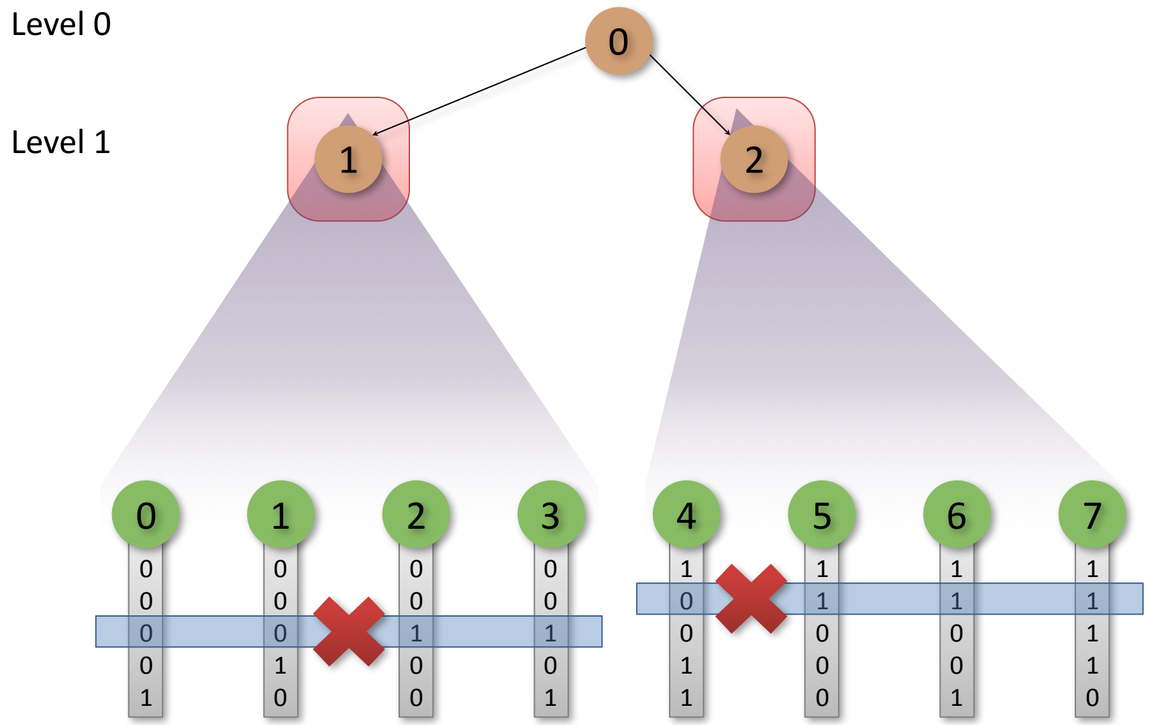
- Building nodes seems serial by level
- Could we generate *all* internal nodes at once?

Garanzha et al.: Challenge



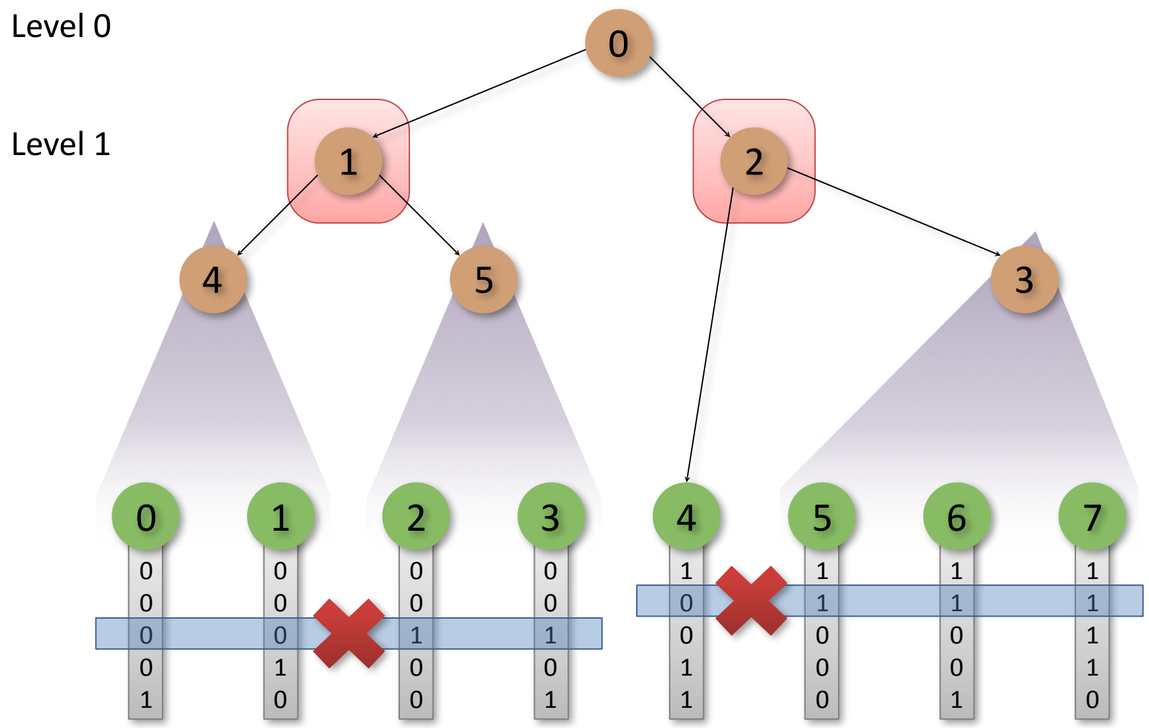
- Building nodes seems serial by level
- Could we generate *all* internal nodes at once?

Garanzha et al.: Challenge



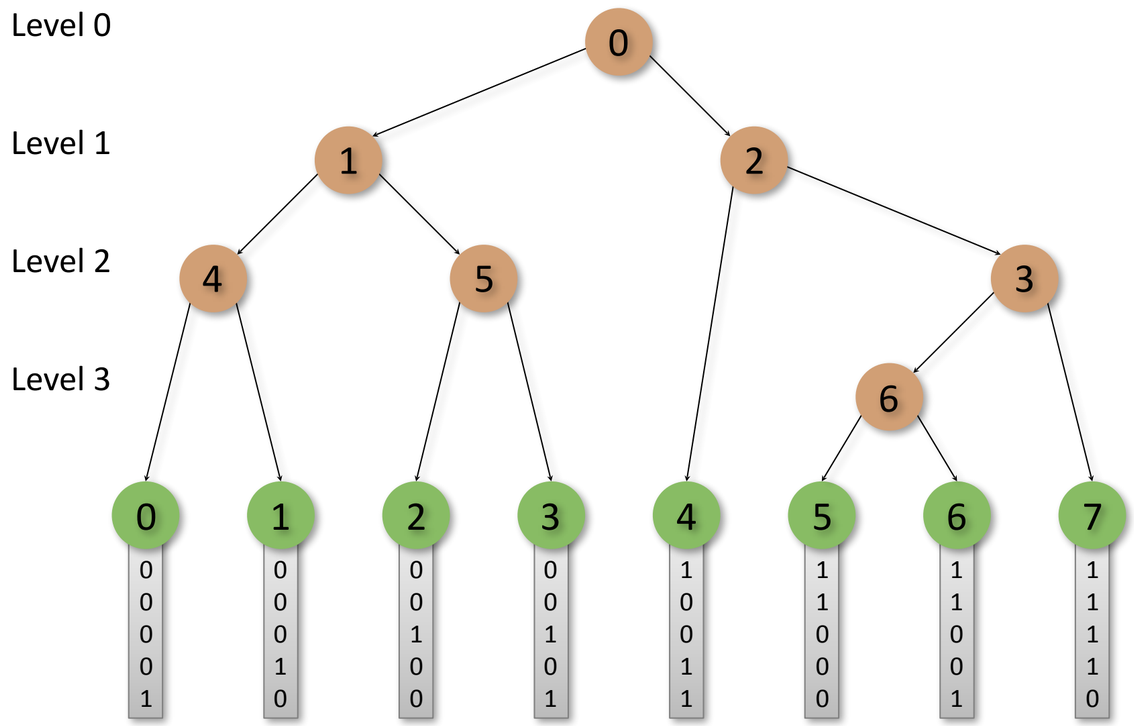
- Building nodes seems serial by level
- Could we generate *all* internal nodes at once?

Garanzha et al.: Challenge



- Building nodes seems serial by level
- Could we generate *all* internal nodes at once?

Garanzha et al.: Challenge



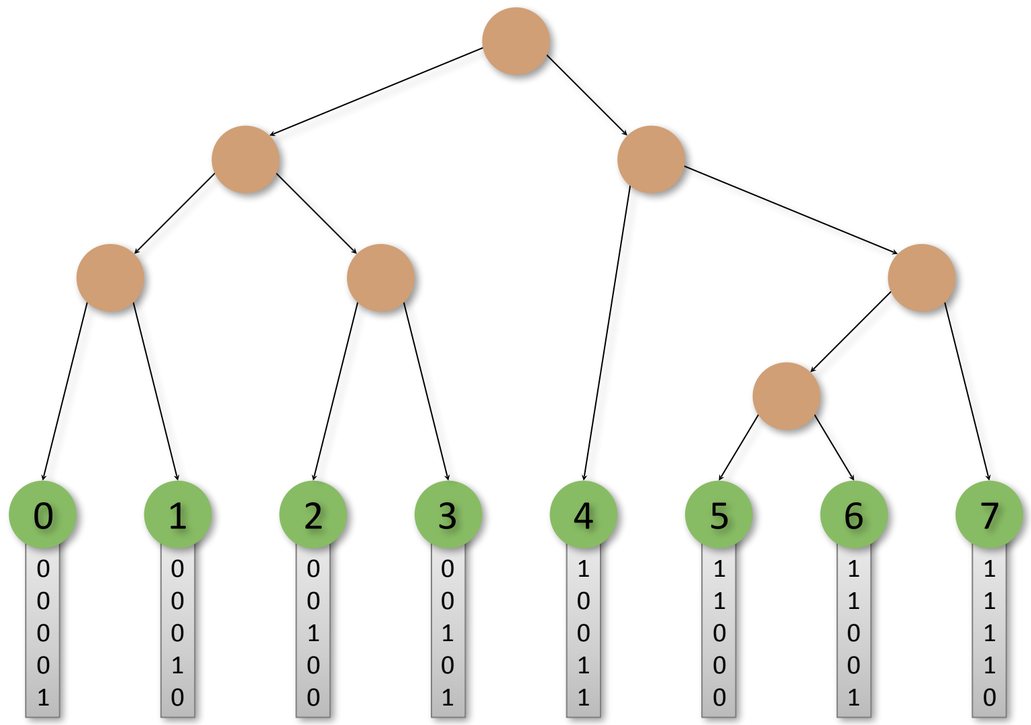
- Building nodes seems serial by level
- Could we generate *all* internal nodes at once?

Karras's method

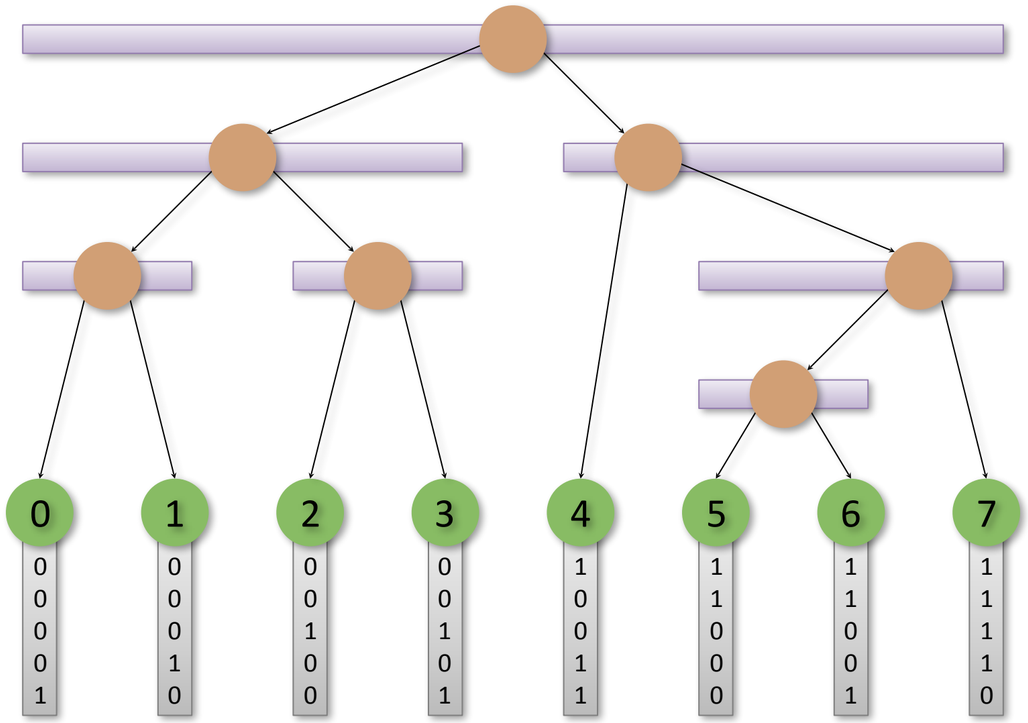
- Define a numbering scheme for the internal nodes
  - Gain some knowledge of their identity
    - What keys do they cover?
  - Connect internal-node indices and leaf (key) indices
- Find the children of a given node
  - Only look at node index and nearby keys
- Do this for all nodes in parallel
  - If we do it right: No dependencies!

T. Karras, "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees," HPG'12: Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics, 2012.

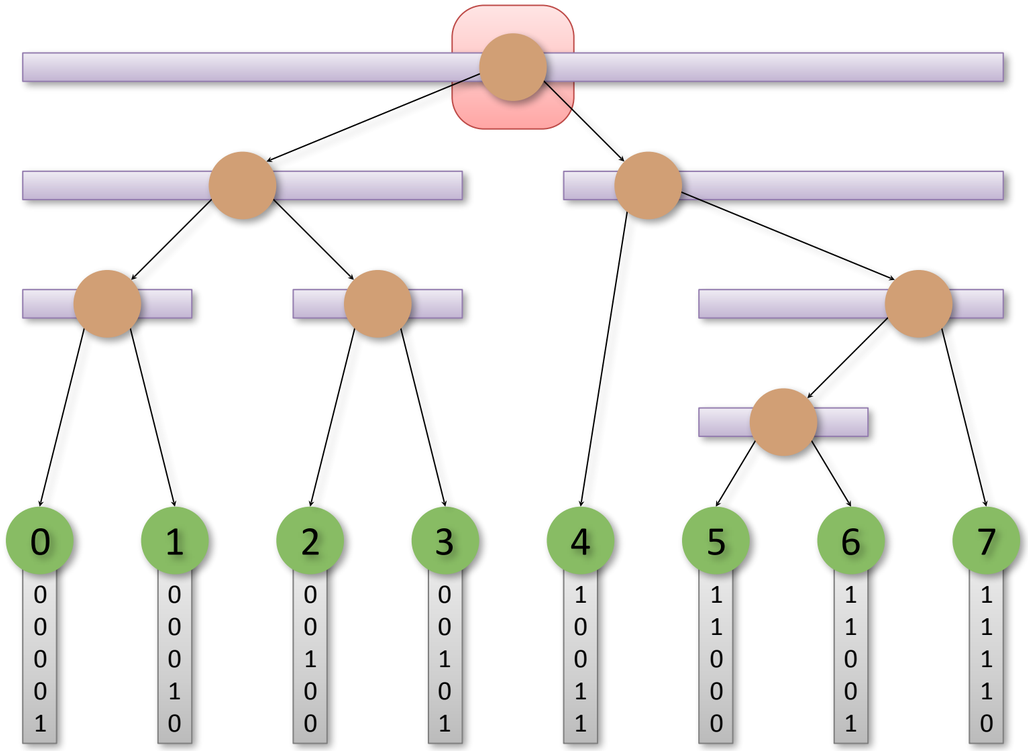
Numbering scheme



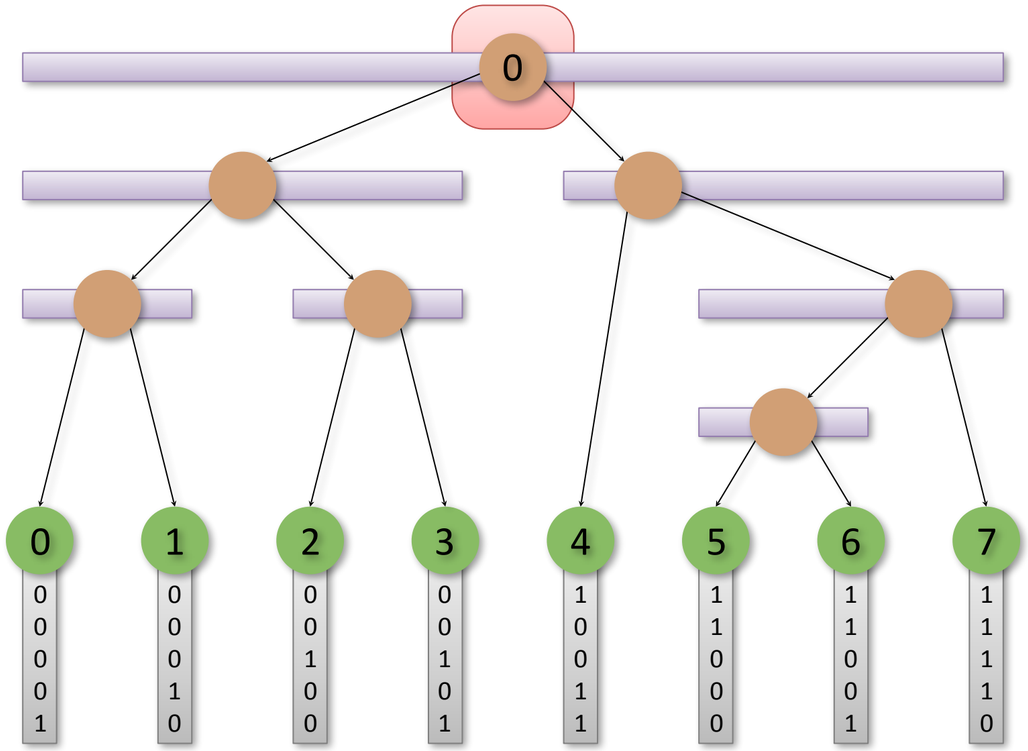
Numbering scheme



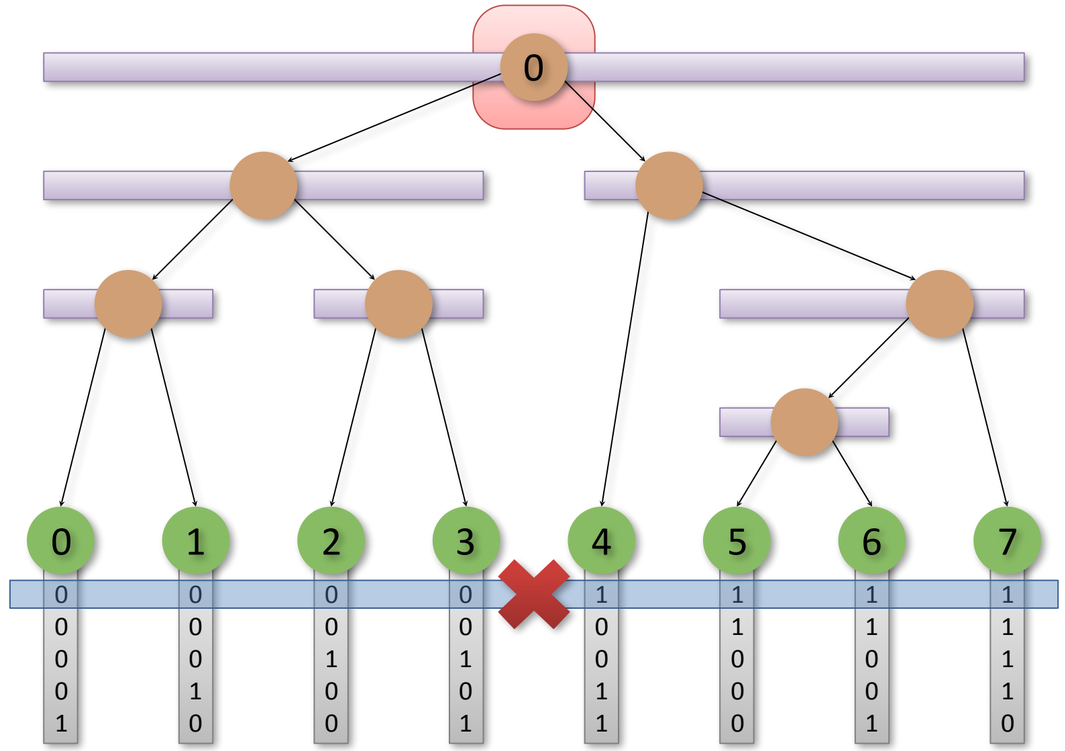
Numbering scheme



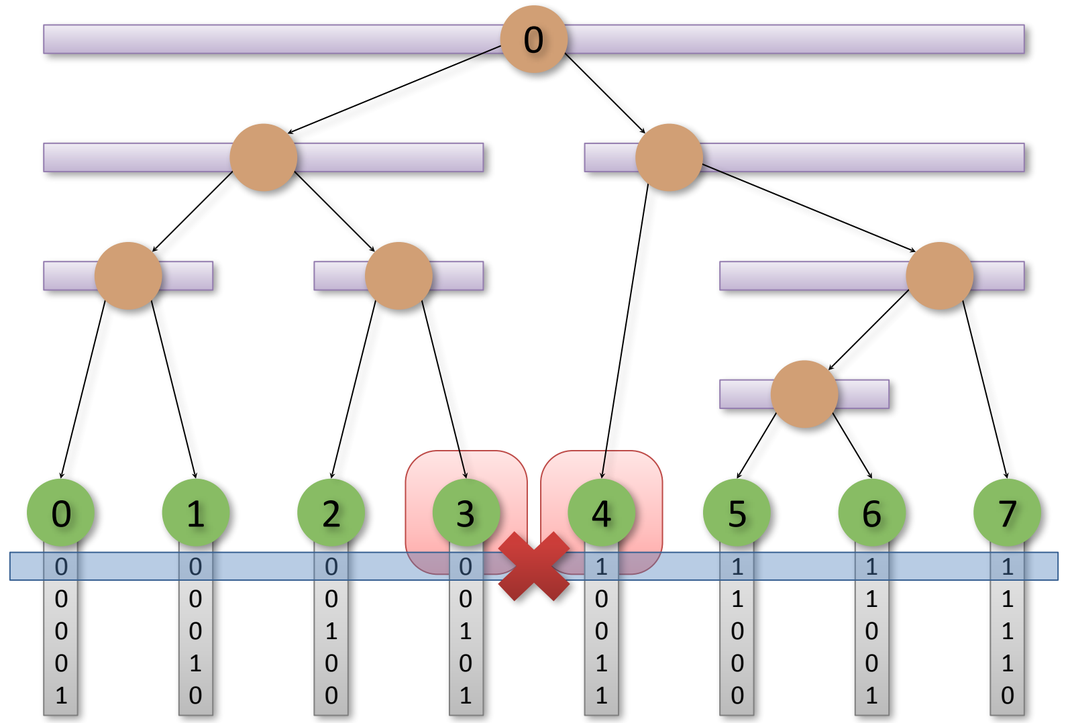
Numbering scheme



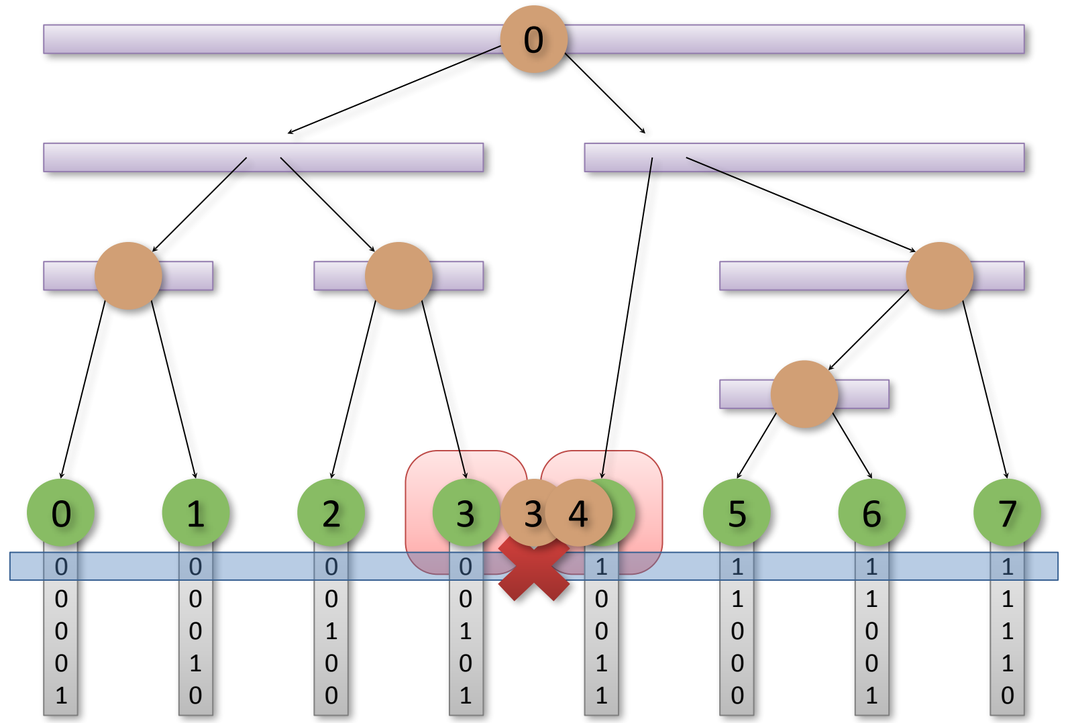
Numbering scheme



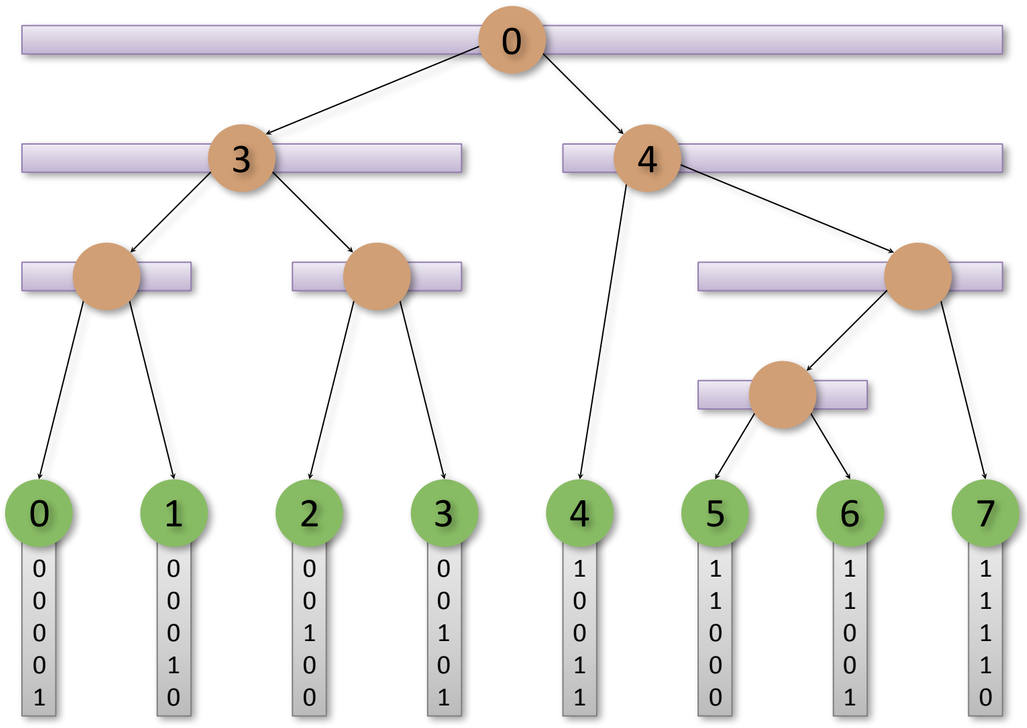
Numbering scheme



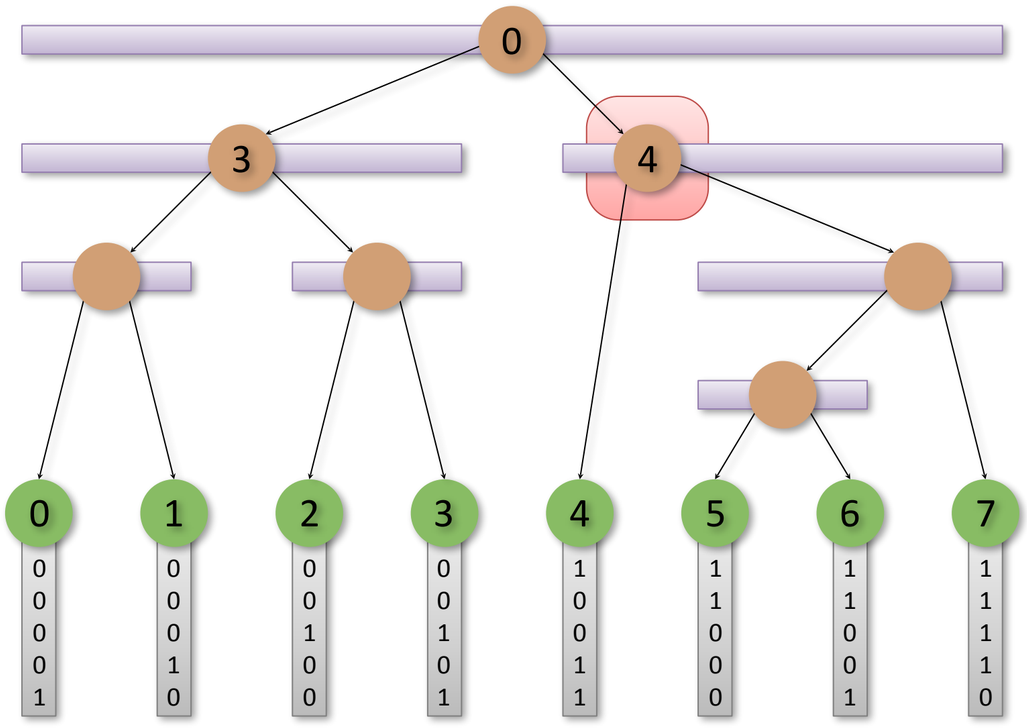
Numbering scheme



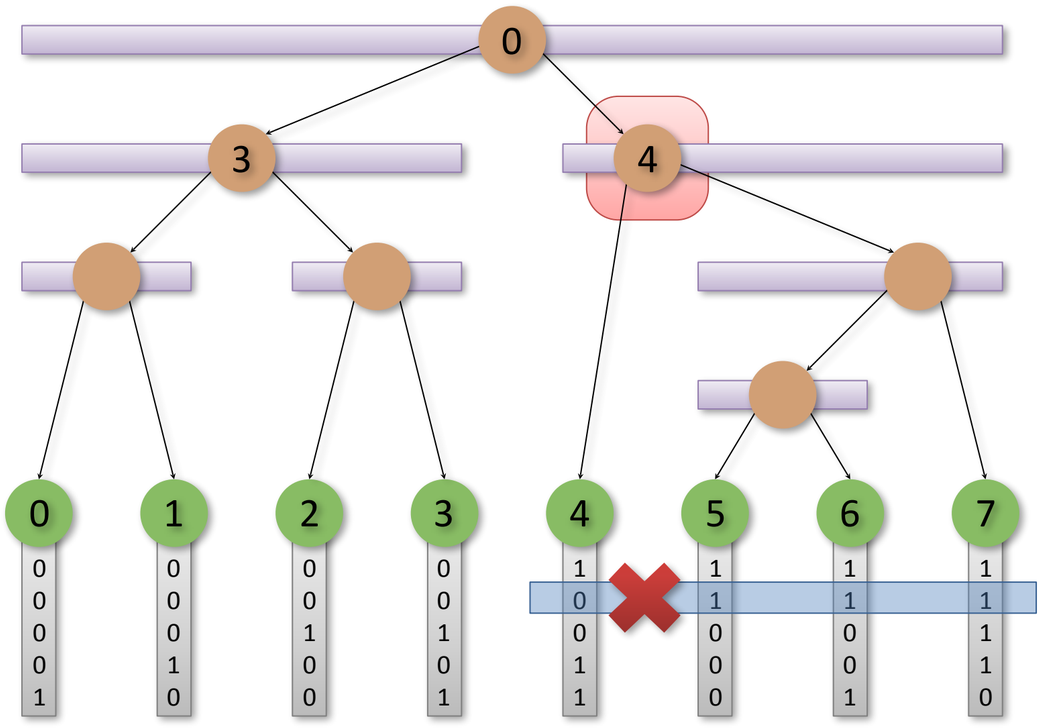
Numbering scheme



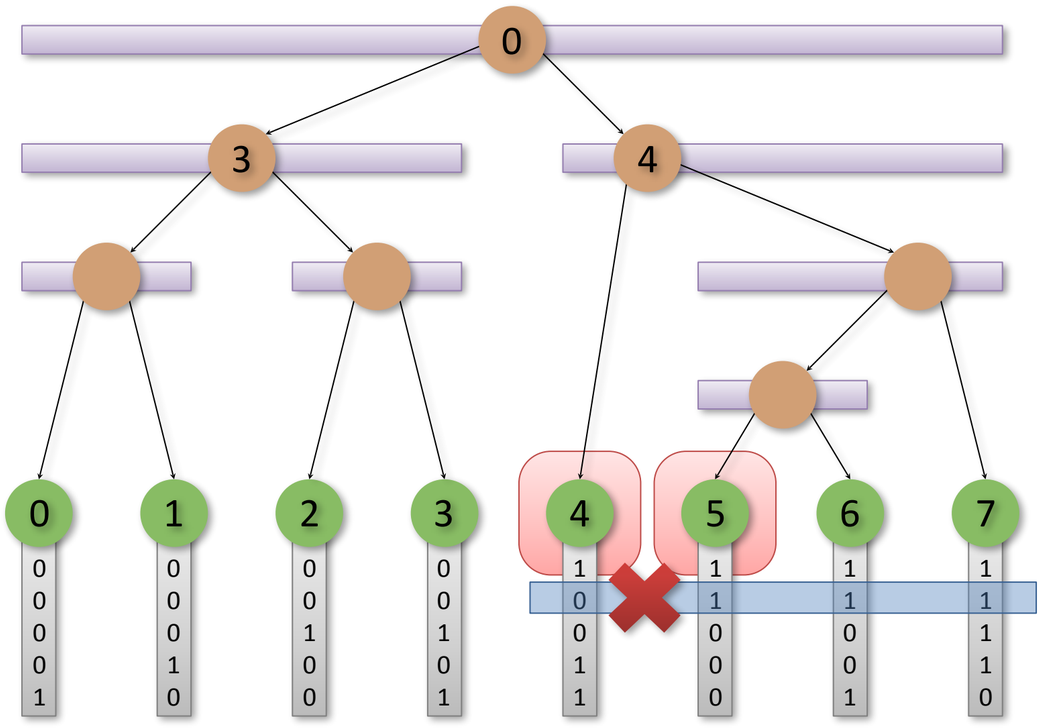
Numbering scheme



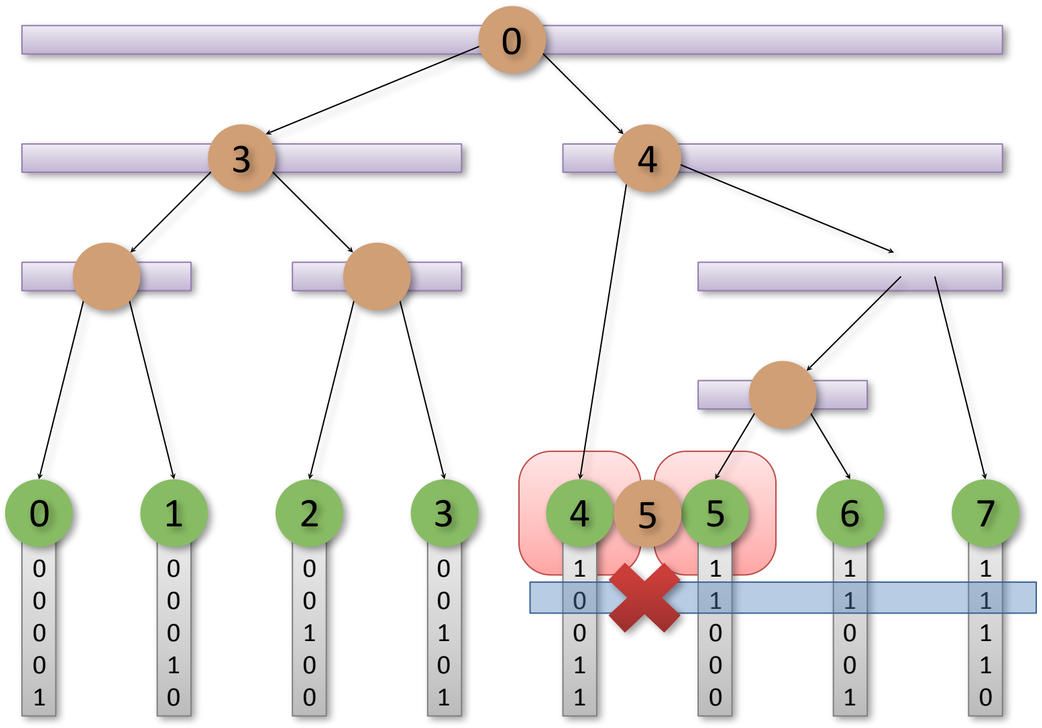
Numbering scheme



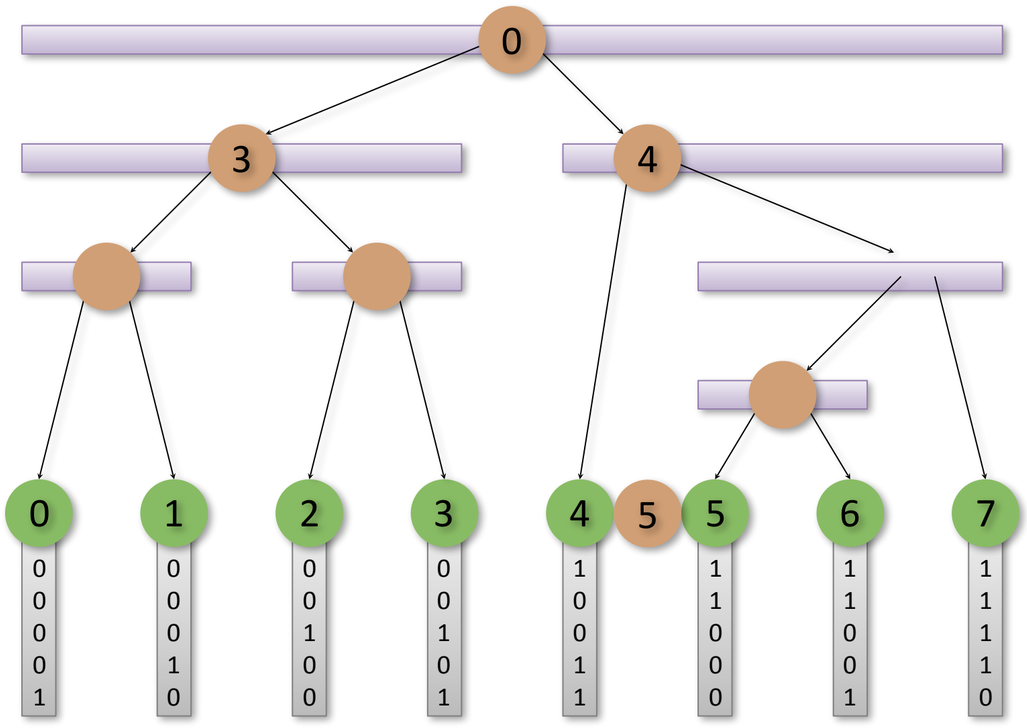
Numbering scheme



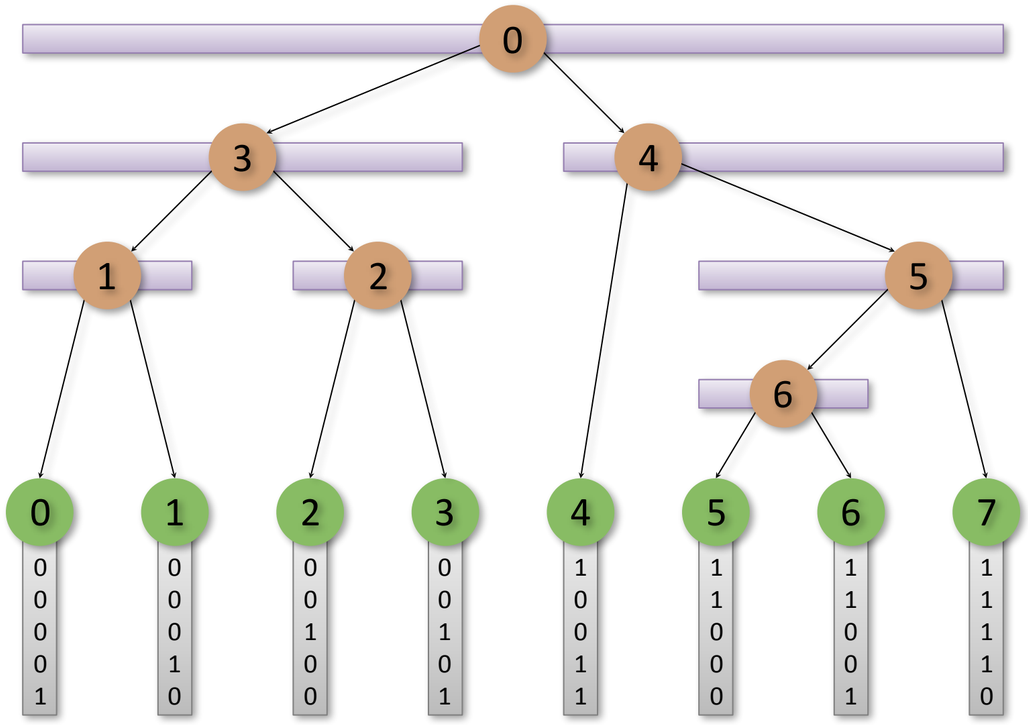
Numbering scheme



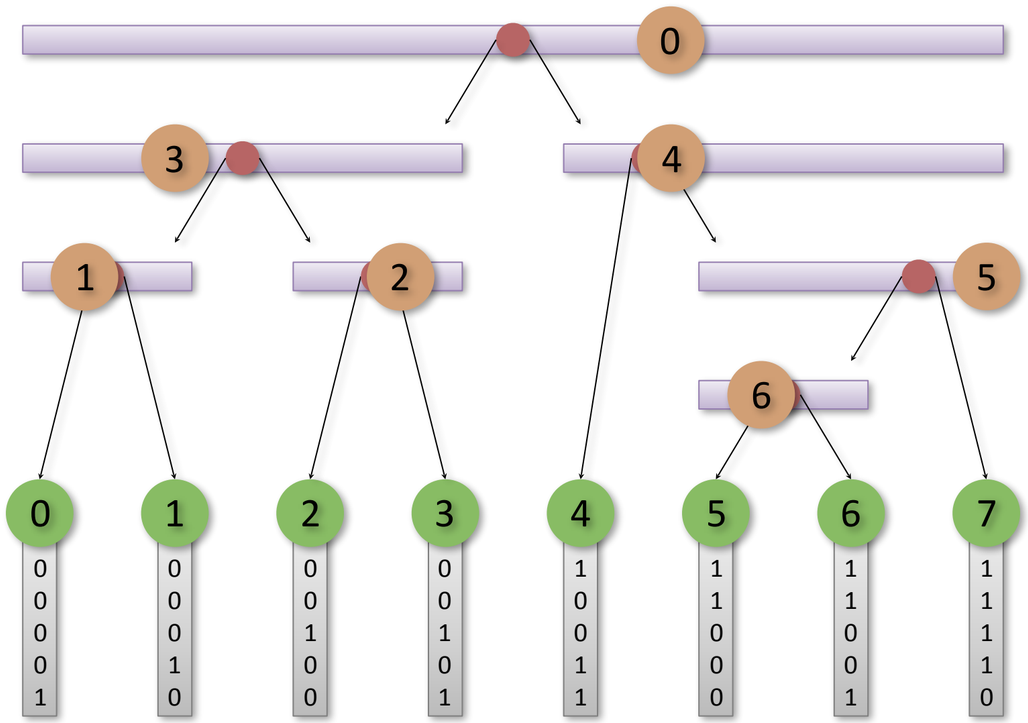
Numbering scheme



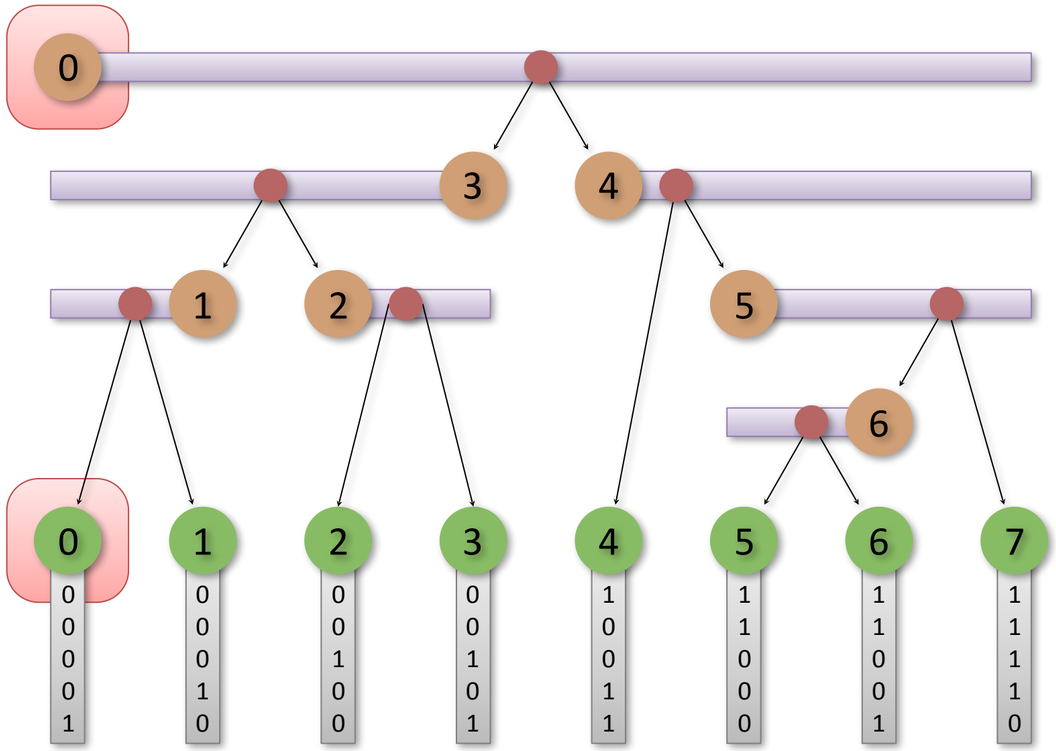
Numbering scheme



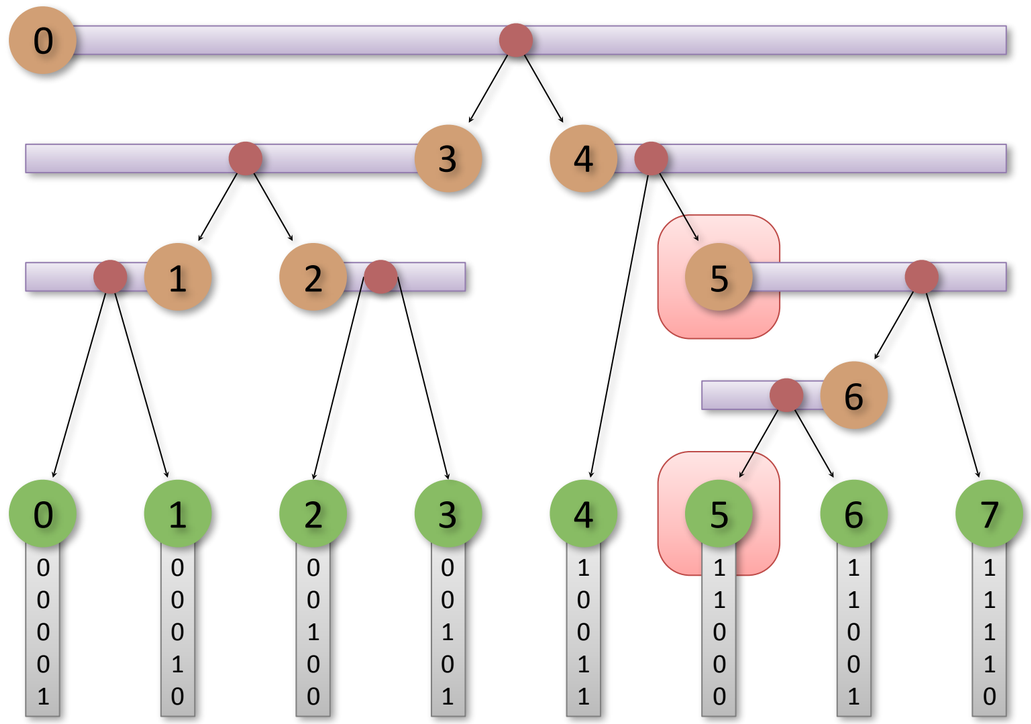
Numbering scheme



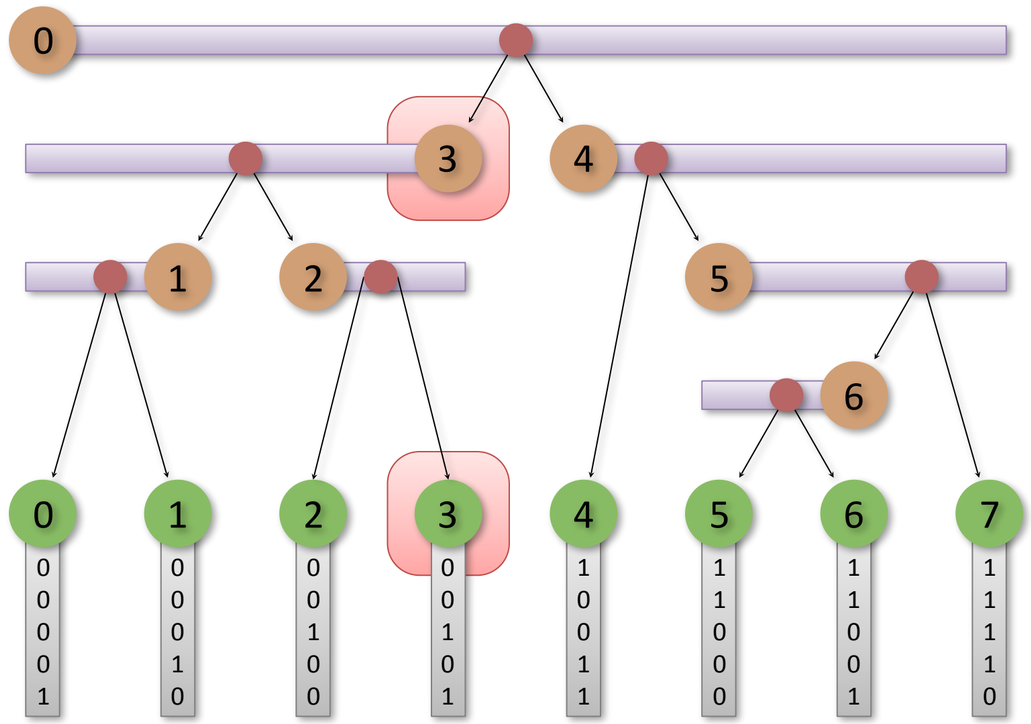
Numbering scheme



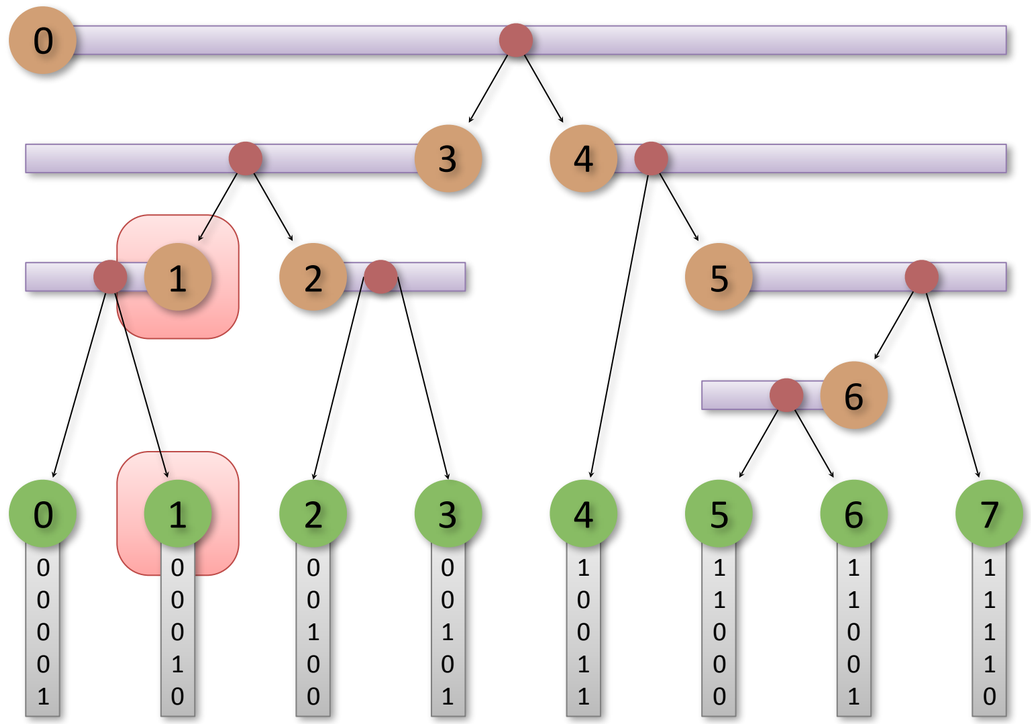
Numbering scheme



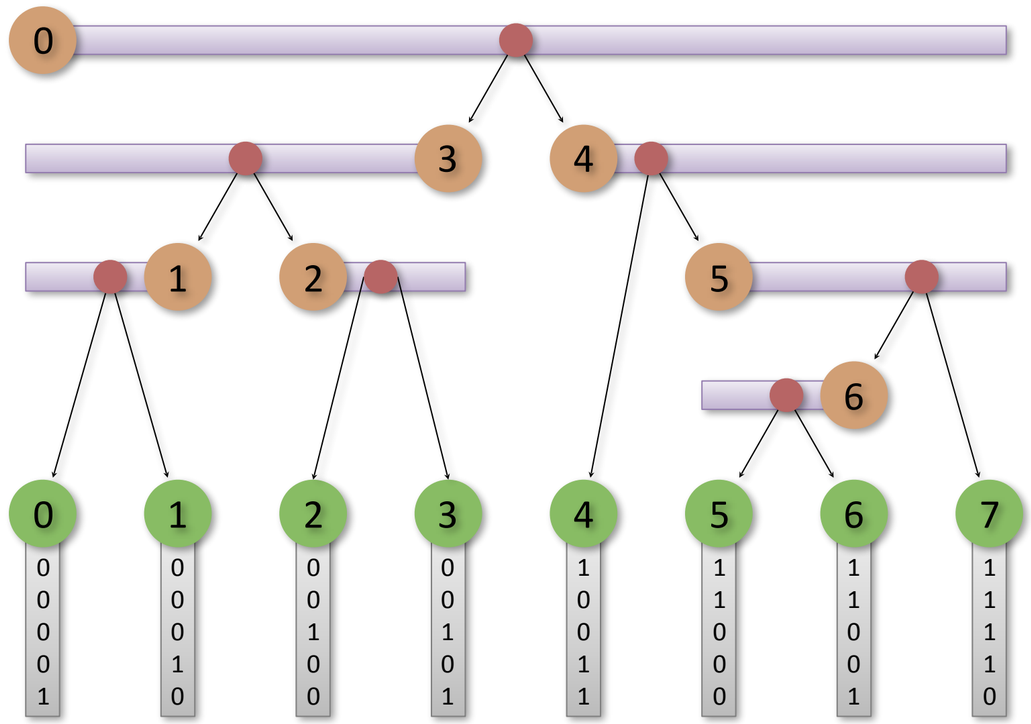
Numbering scheme



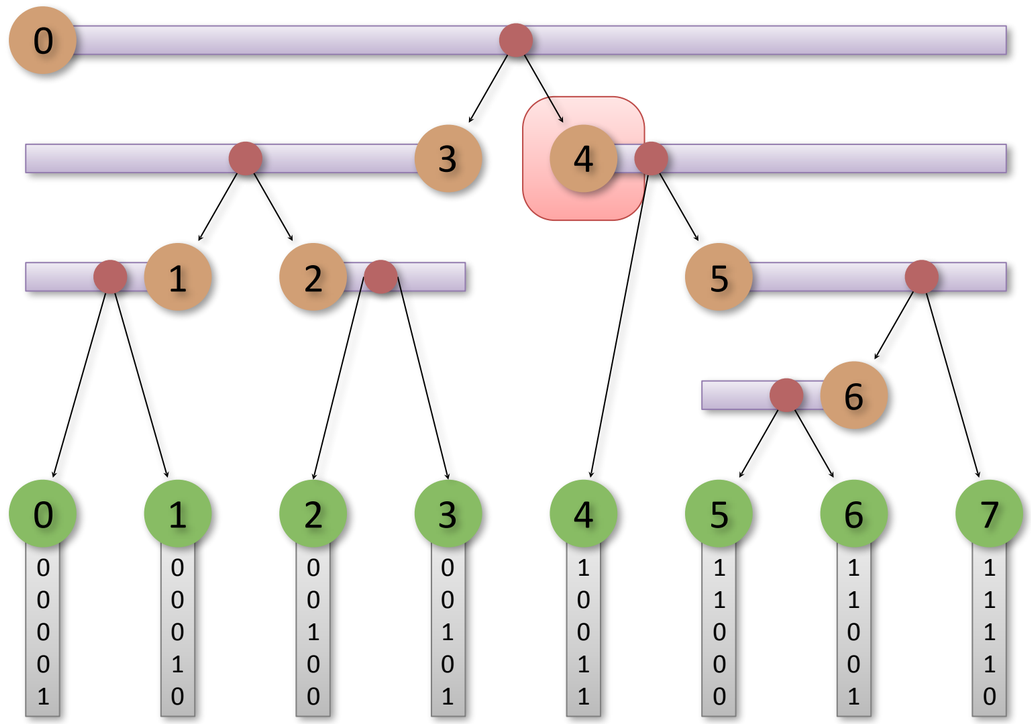
Numbering scheme



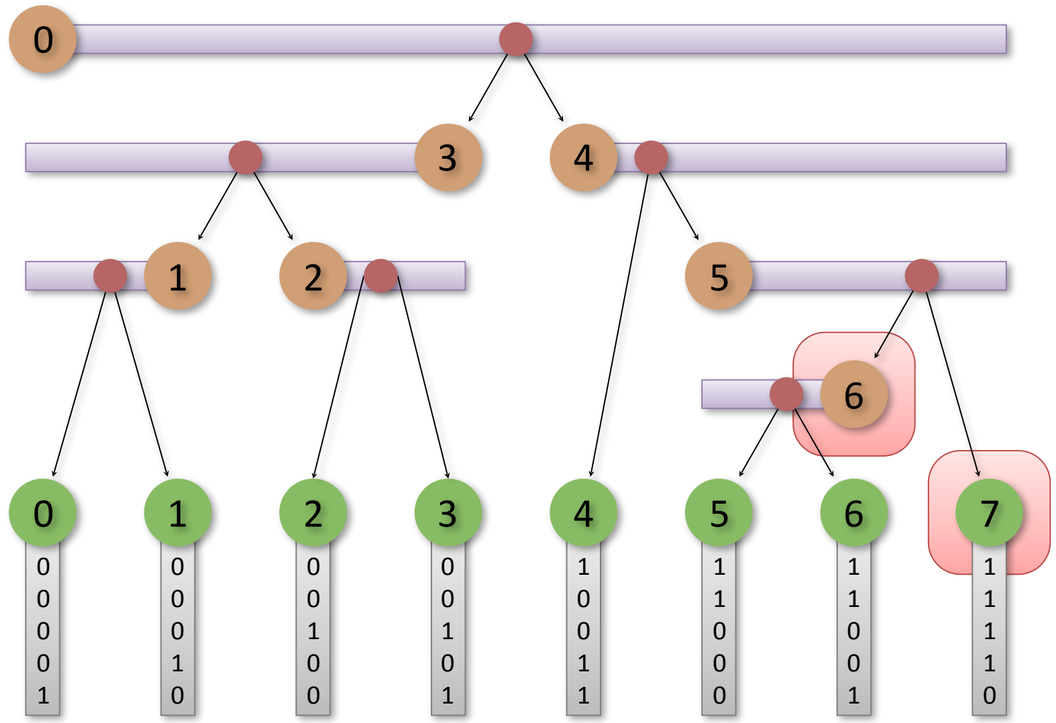
Numbering scheme



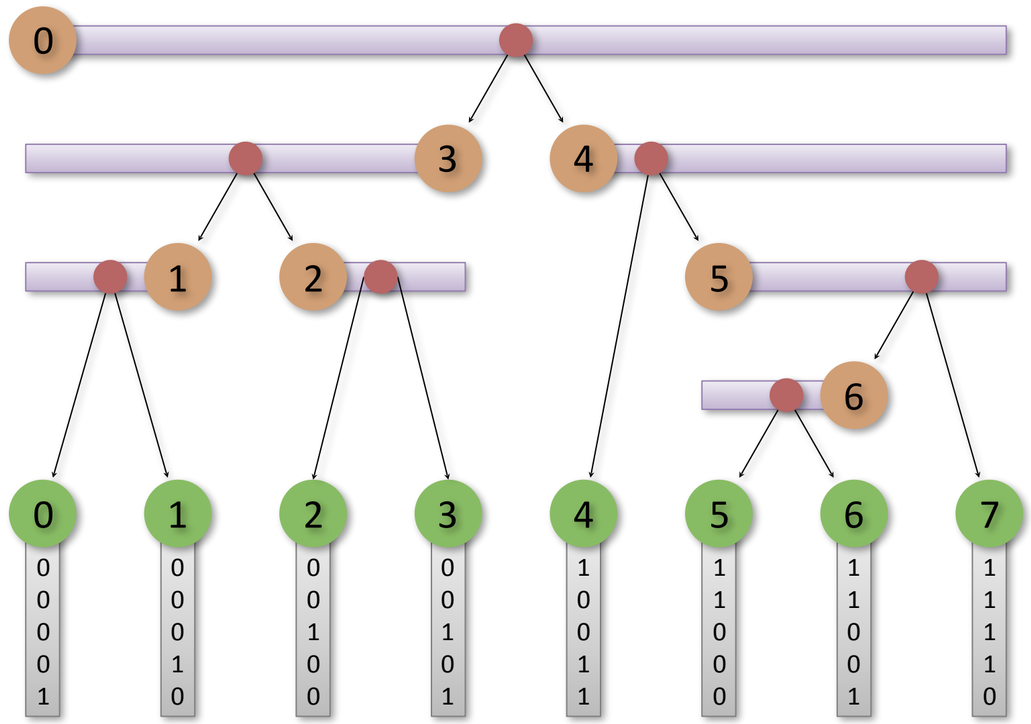
Numbering scheme



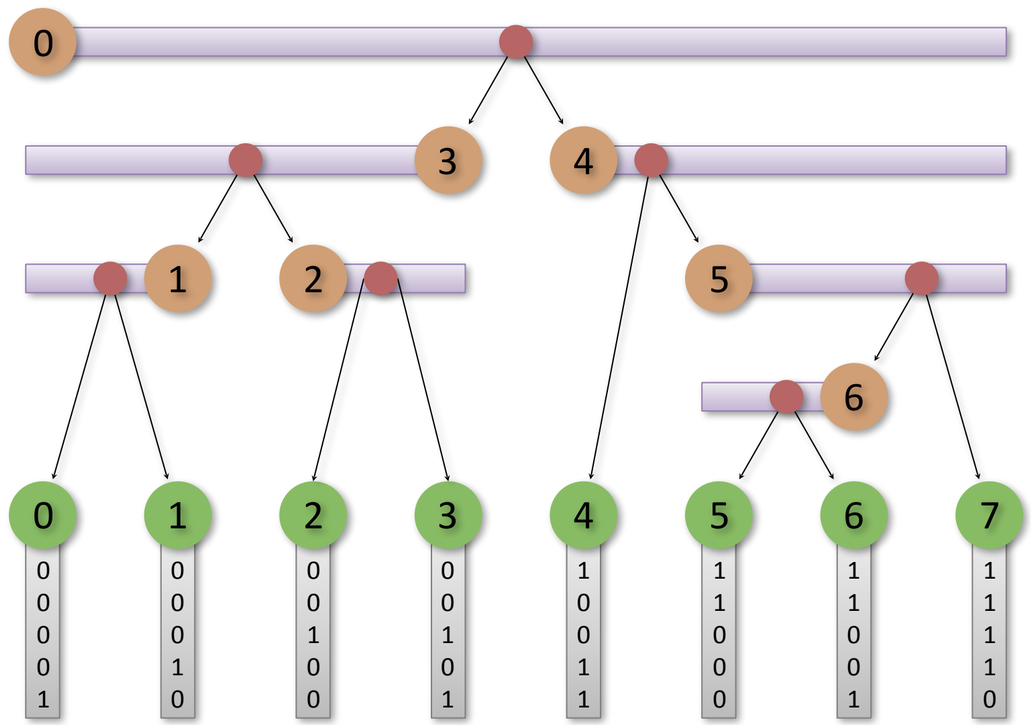
Numbering scheme



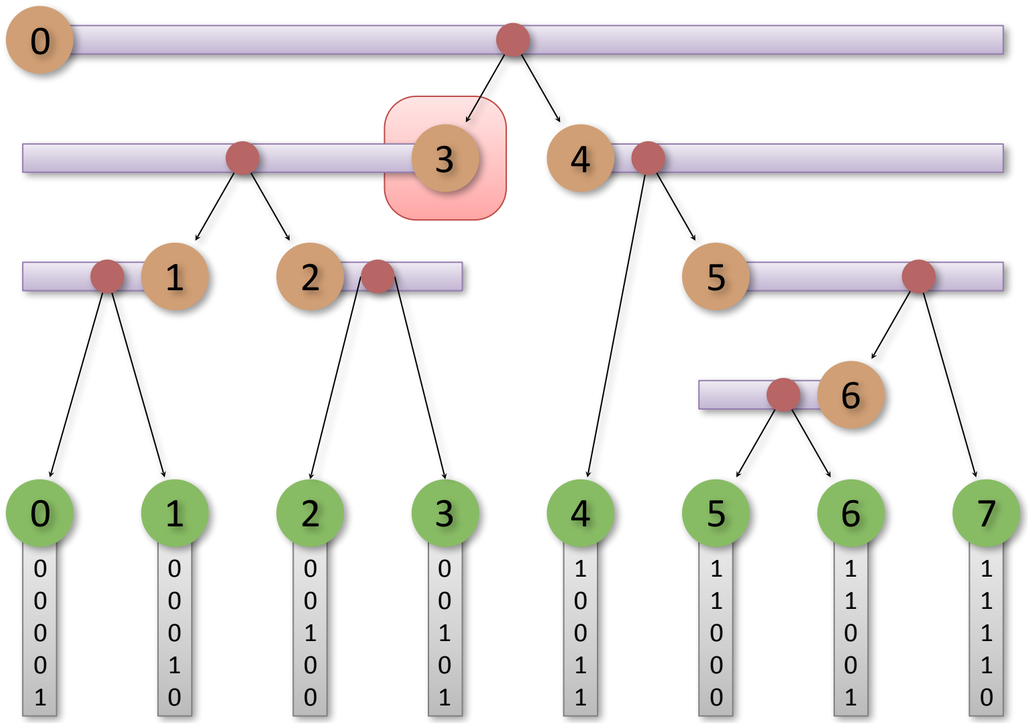
Numbering scheme



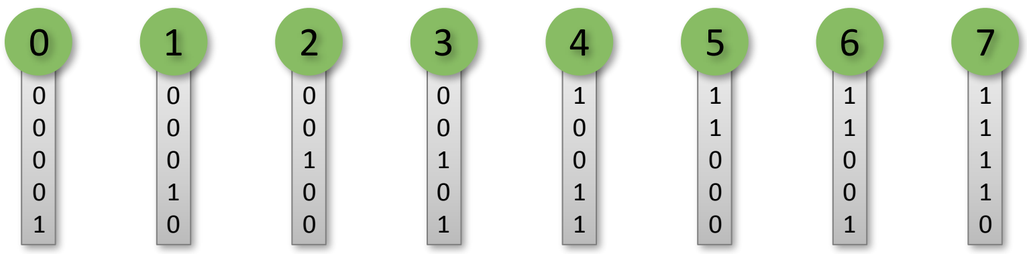
Algorithm


0
3
4
1
2
5
6
0
1
2
3
4
5
6
7
0 0 0 0 0 1
0 0 0 1 1 0
0 0 1 0 0 0
0 0 1 0 1 1
1 0 0 1 1 1
1 1 0 0 0 0
1 1 0 0 0 1
1 1 1 1 1 0

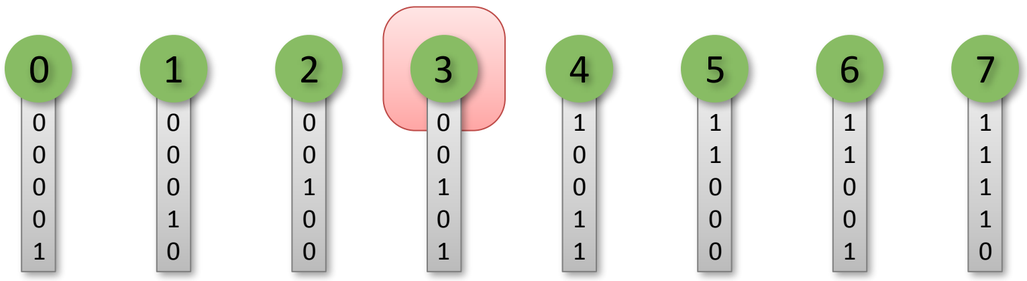
Algorithm


0
3
4
1
2
5
6
0
1
2
3
4
5
6
7
0 0 0 0 1
0 0 0 1 0
0 0 1 0 0
0 0 1 0 1
1 0 0 1 1
1 1 0 0 0
1 1 0 0 1
1 1 1 1 0

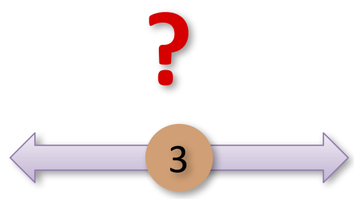
Algorithm

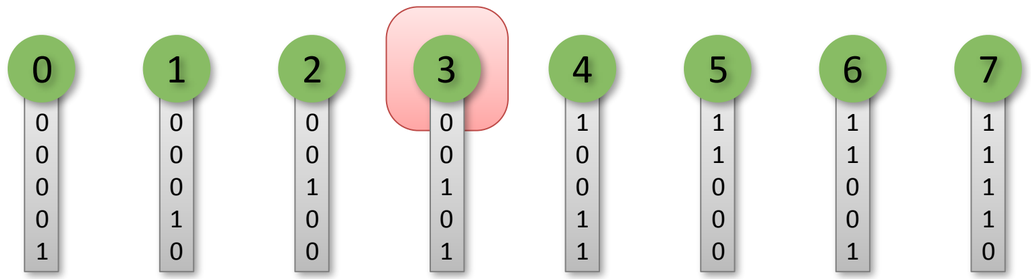


Algorithm

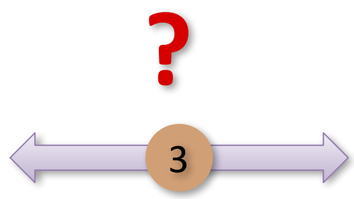


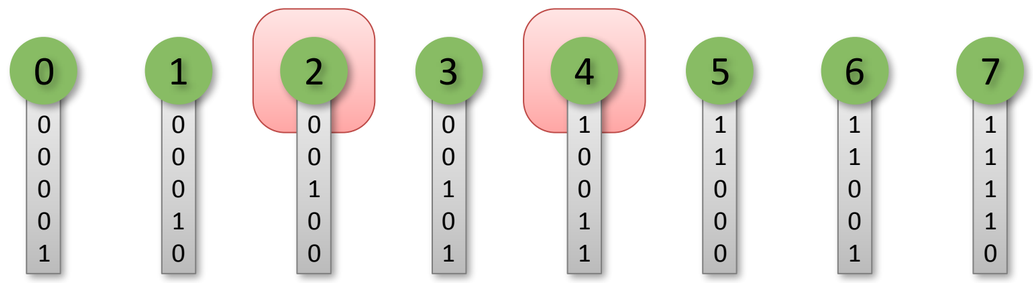
Algorithm



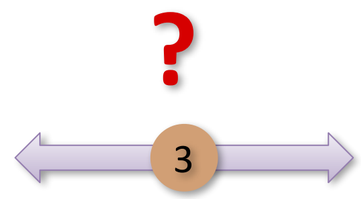


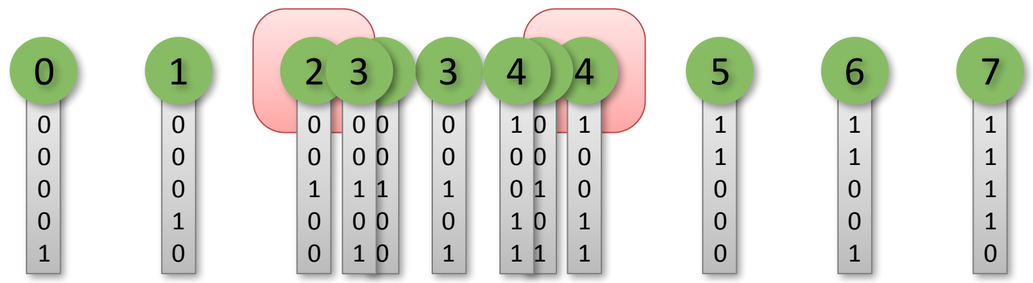
Algorithm



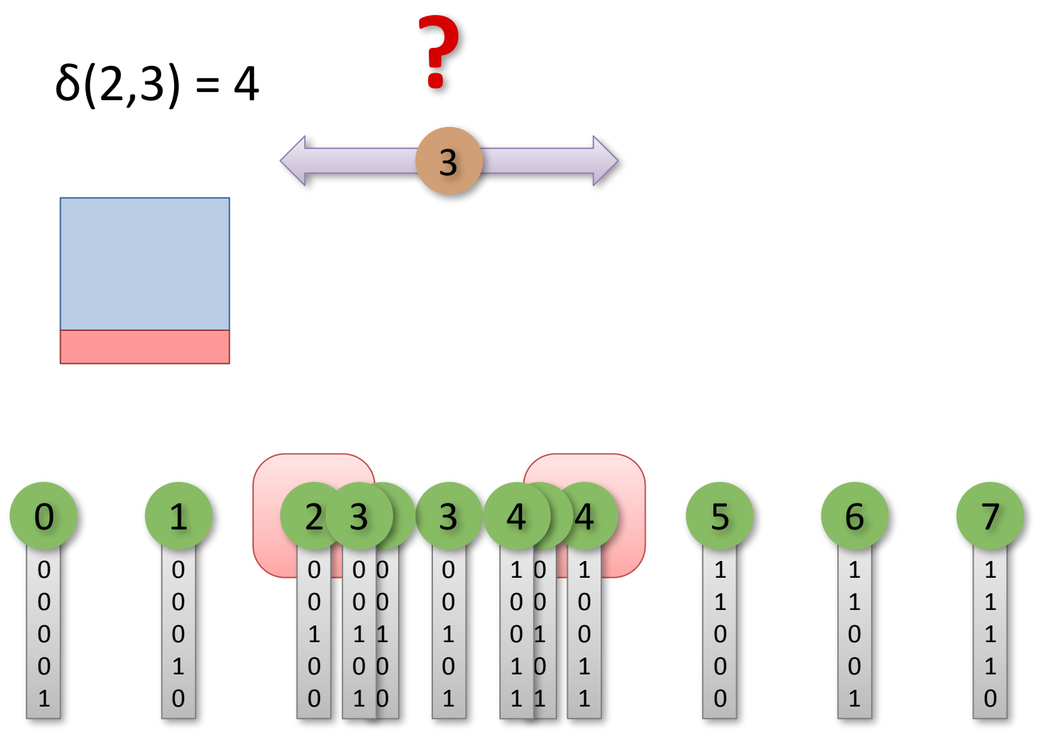


Algorithm

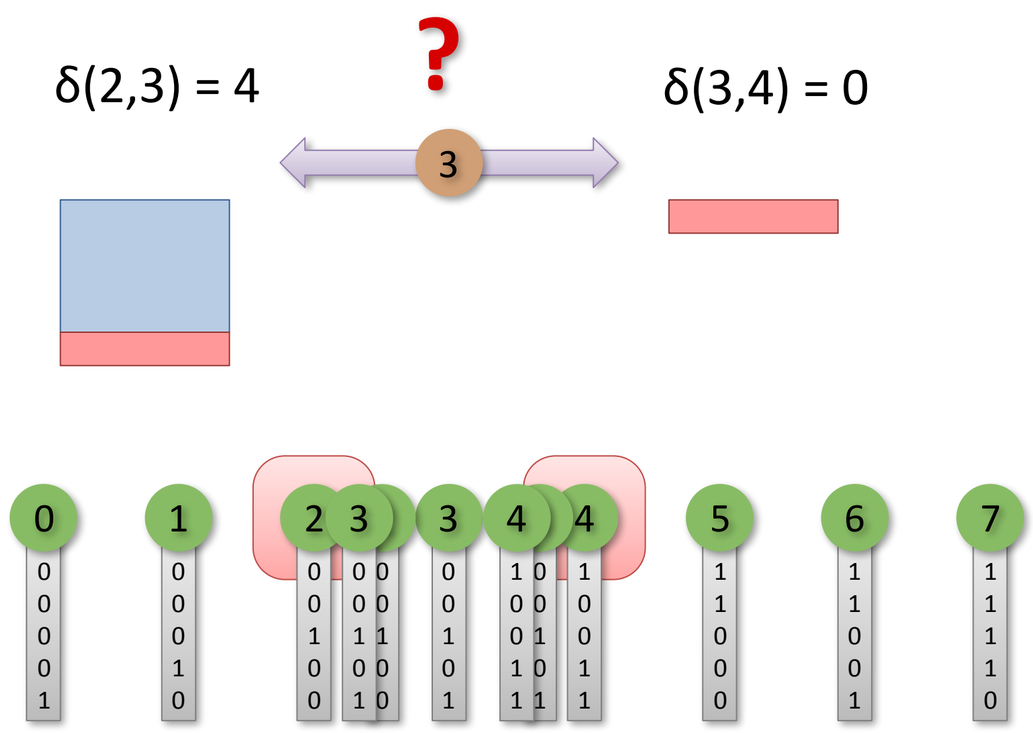

?
3


0
0 0 0 0 1
1
0 0 0 1 0
2
0 0 1 0 0
3
0 0 1 0 1 0
3
0 0 1 0 1
4
1 0 0 1 1
4
0 0 1 0 1
4
1 0 0 1 1
5
1 1 0 0 0
6
1 1 0 0 1
7
1 1 1 1 0

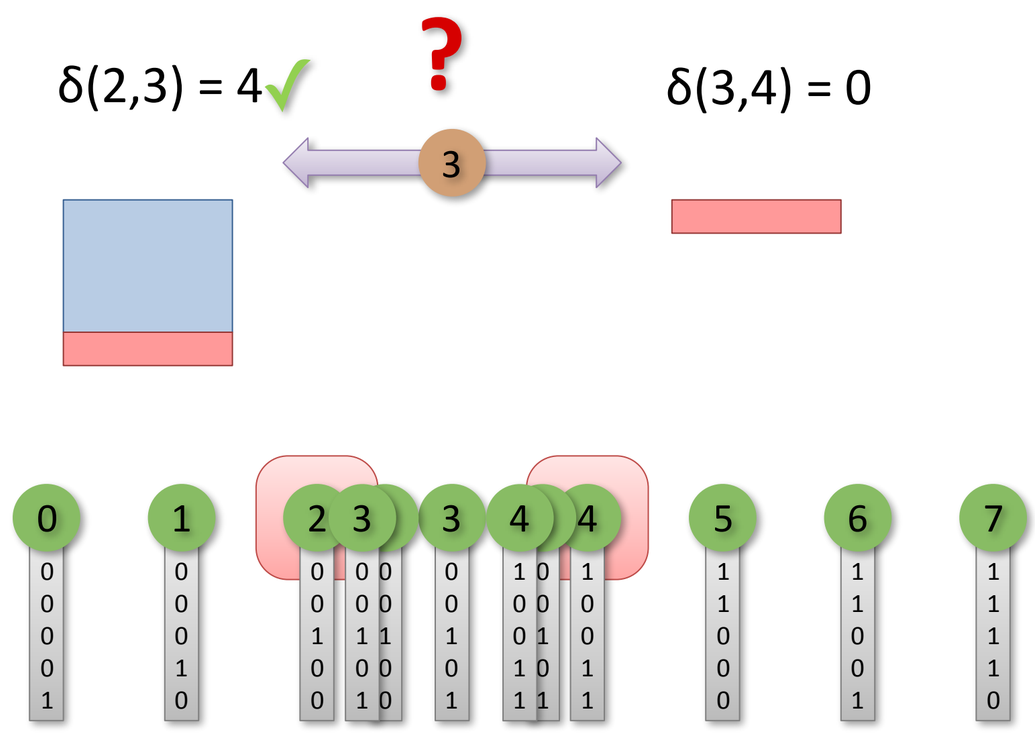
Algorithm



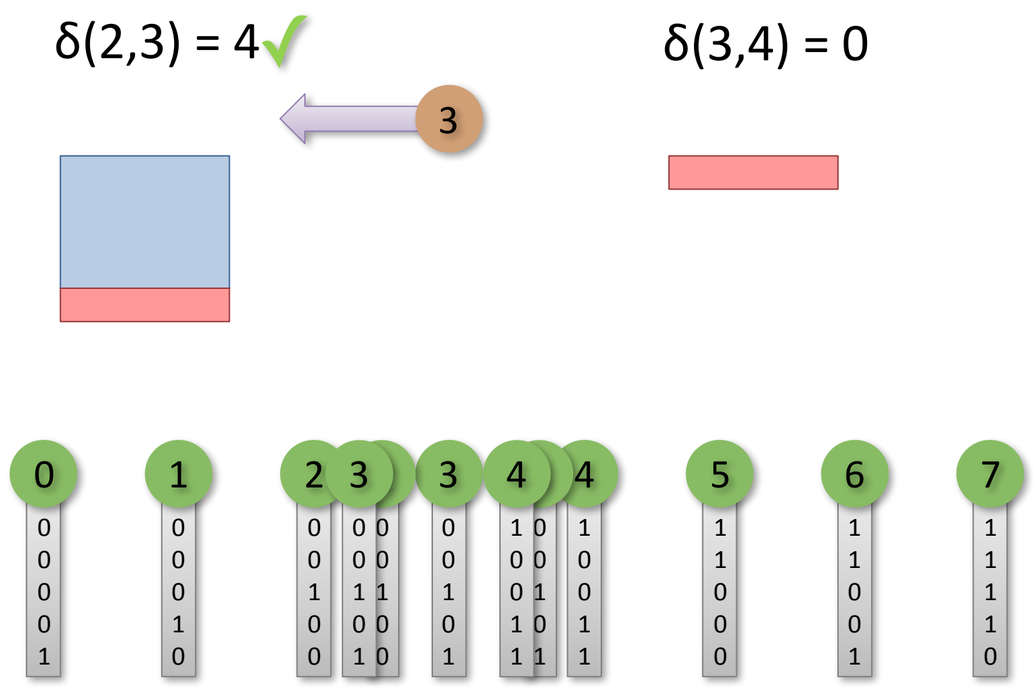
Algorithm


δ(2,3) = 4
?
δ(3,4) = 0
3
0
1
2
3
3
4
4
5
6
7
0 0 0 0 1
0 0 0 1 0
0 0 1 0 0
0 0 1 0 1
0 0 1 0 0
0 1 0 1
1 0 0 1 1
0 0 1 0 1
1 0 0 1 1
1 1 0 0 0
1 1 0 0 1
1 1 1 1 0

Algorithm



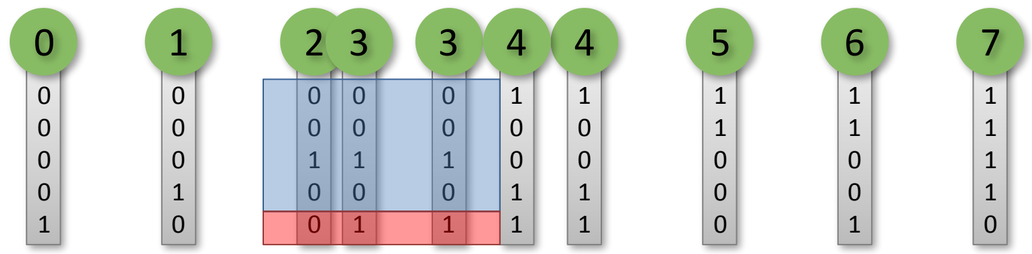
Algorithm


δ(2,3) = 4
δ(3,4) = 0
3
0
0 0 0 0 1
1
0 0 0 1 0
2
0 0 1 0 0
3
0 0 1 0 0
3
0 0 1 0 1
4
1 0 0 1 1
4
0 0 1 0 1
4
1 0 0 1 1
5
1 1 0 0 0
6
1 1 0 0 1
7
1 1 1 1 0

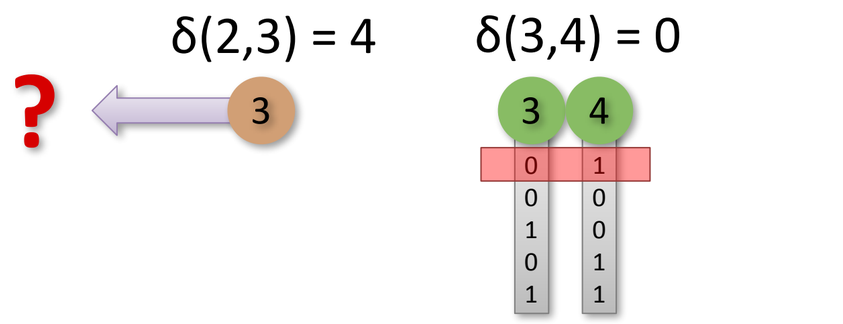
Algorithm

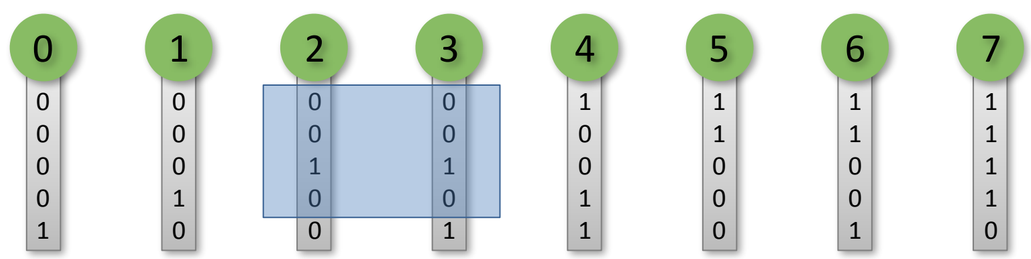
δ(2,3) = 4

δ(3,4) = 0

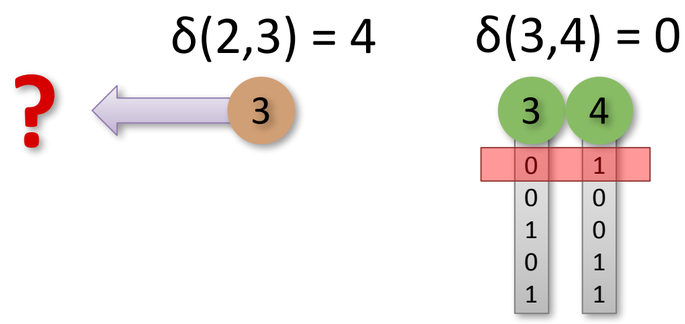


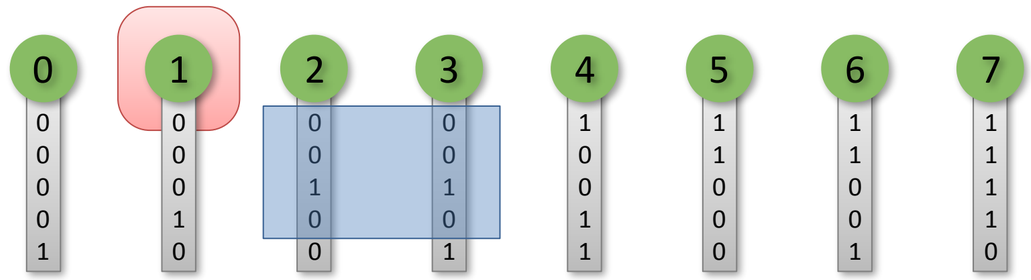
Algorithm


δ(2,3) = 4
δ(3,4) = 0
?
3
3
4
0
1
0
1
0
1
1
0
0
1
0
1
1

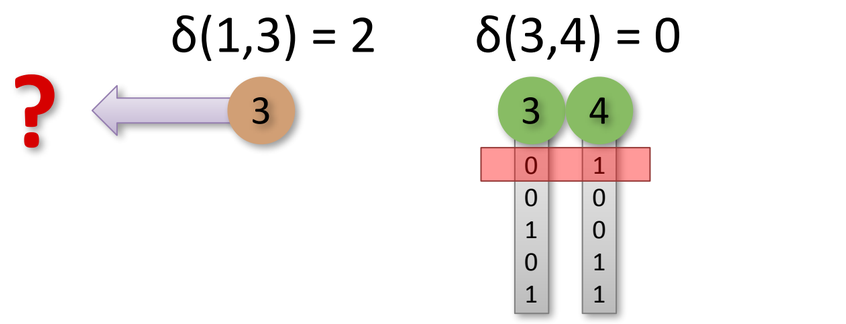

0
1
2
3
4
5
6
7
0 0 0 0 1
0 0 0 1 0
0 0 1 0 0
0 0 1 0 1
1 0 0 1 1
1 1 0 0 0
1 1 0 0 1
1 1 1 1 0

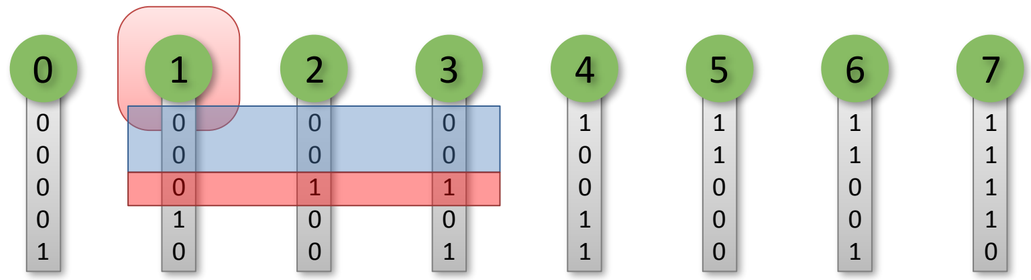
Algorithm


δ(2,3) = 4
δ(3,4) = 0
?
3
3
4
0
1
0
0
1
0
1
1
0
0
1

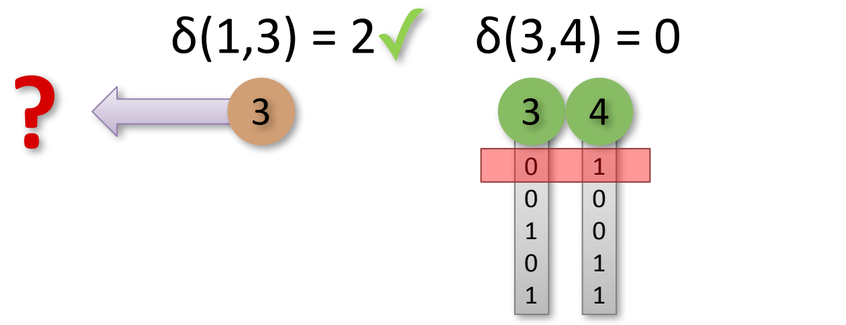

0
1
2
3
4
5
6
7
0 0 0 0 1
0 0 0 1 0
0 0 1 0 0
0 0 1 0 1
1 0 0 1 1
1 1 0 0 0
1 1 0 0 1
1 1 1 1 0

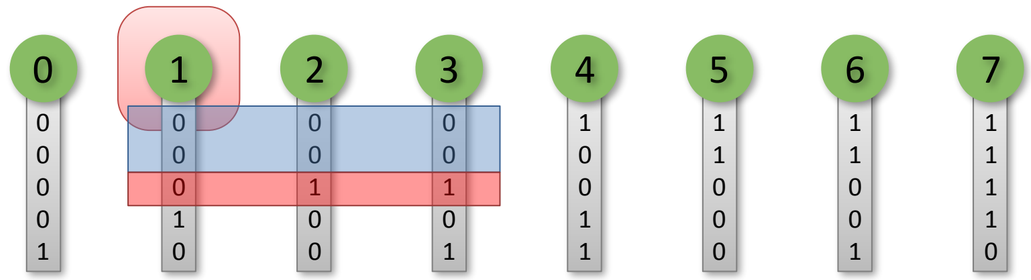
Algorithm


δ(1,3) = 2
δ(3,4) = 0
?
3
3
4
0
1
0
0
1
0
1
1
0
0
1
1

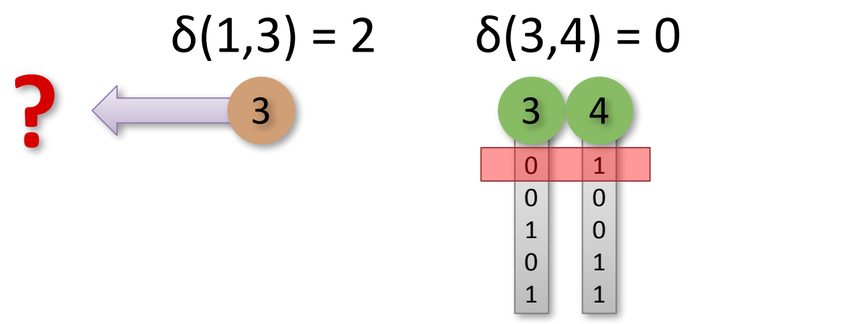

0
1
2
3
4
5
6
7
0
0
0
0
1
0
0
0
1
0
0
0
1
0
0
1
0
0
1
0
1
1
0
0
0
1
1
0
0
1
1
1
0
0
0
1
1
1
1
0

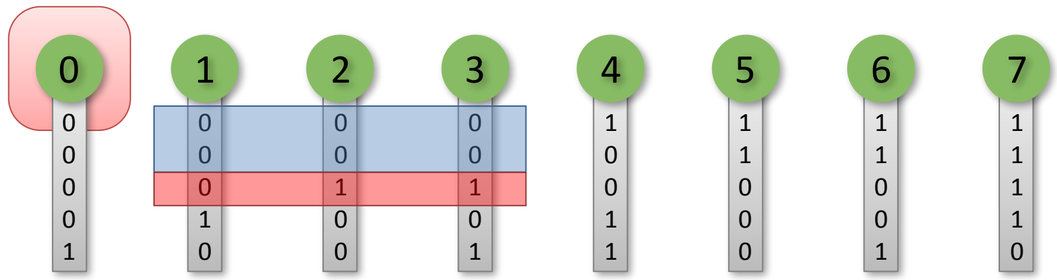
Algorithm



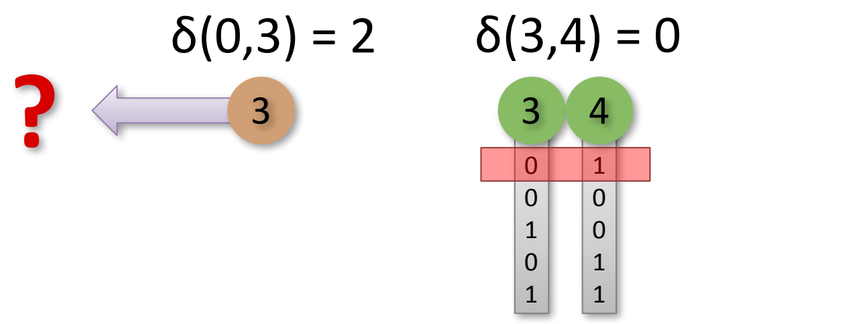


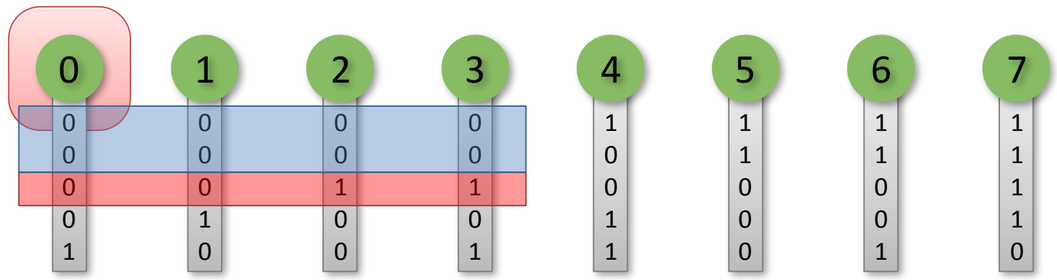
Algorithm


δ(1,3) = 2
δ(3,4) = 0
?
3
3
4
0
1
0
1
0
1
0
1
0
1
1
0
0
1
1

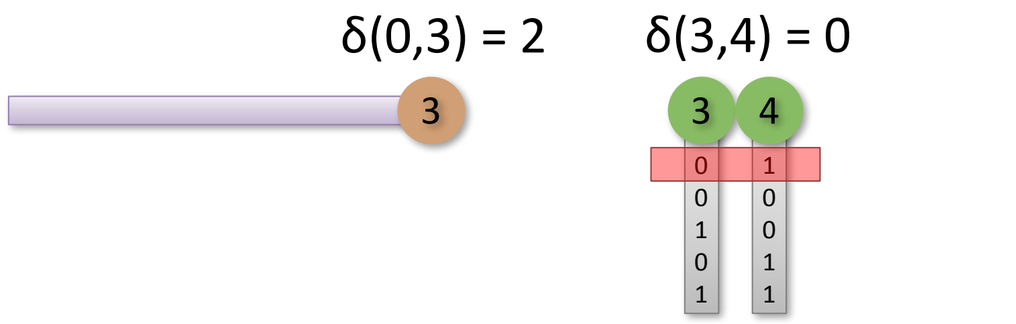

0
1
2
3
4
5
6
7
0
0
0
0
1
0
0
0
1
0
0
1
0
0
0
1
0
0
1
0
1
1
0
0
1
1
0
0
0
1
1
1
0
0
1
1
1
1
0

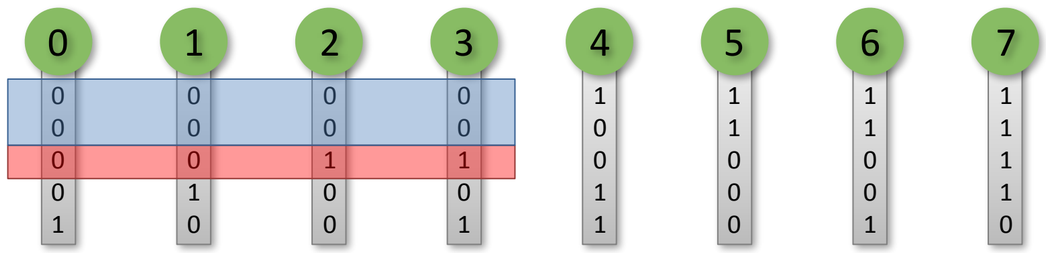
Algorithm


δ(0,3) = 2
δ(3,4) = 0
?
3
3
4
0
1
0
1
0
1
0
1
1
0
0
1
1

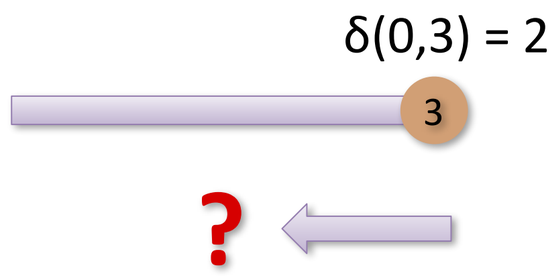

0
1
2
3
4
5
6
7
0
0
0
1
0
0
0
1
0
0
1
0
0
0
1
0
1
1
0
0
1
0
0
1
1
0
0
1
1
0
0
0
1
1
0
0
1
1
1
1
1
0

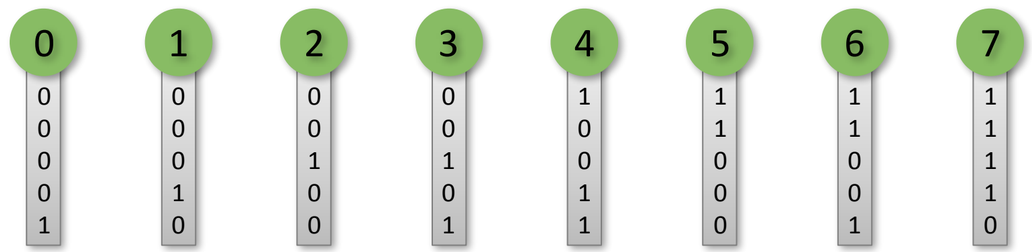
Algorithm


δ(0,3) = 2
δ(3,4) = 0
3
3
4
0
1
0
0
1
0
1
1
0
0
1
1

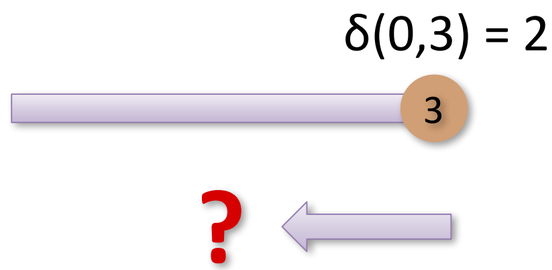

0
1
2
3
4
5
6
7
0
0
0
1
0
0
1
0
0
0
1
0
0
1
0
1
1
0
0
1
1
1
0
0
0
1
1
1
0
0
1
1
1
1
1
0

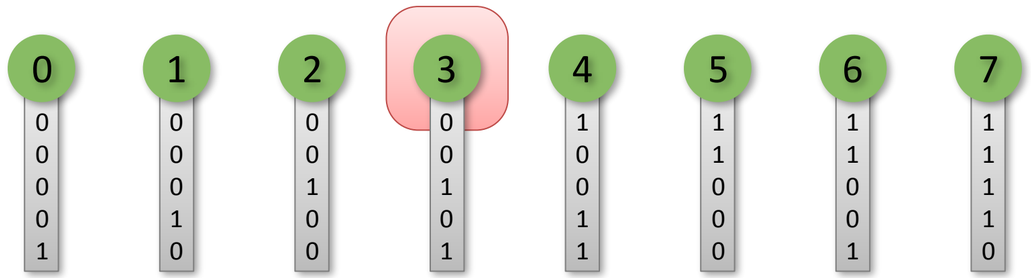
Algorithm


δ(0,3) = 2
3
?

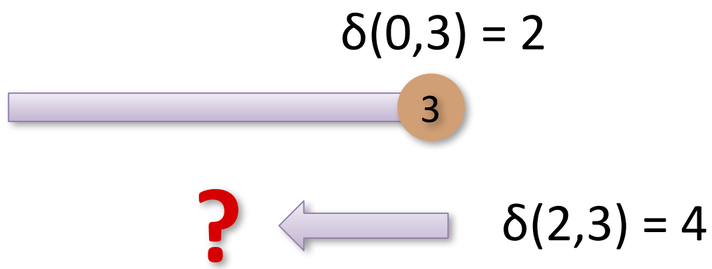

0
0 0 0 0 1
1
0 0 0 1 0
2
0 0 1 0 0
3
0 0 1 0 1
4
1 0 0 1 1
5
1 1 0 0 0
6
1 1 0 0 1
7
1 1 1 1 0

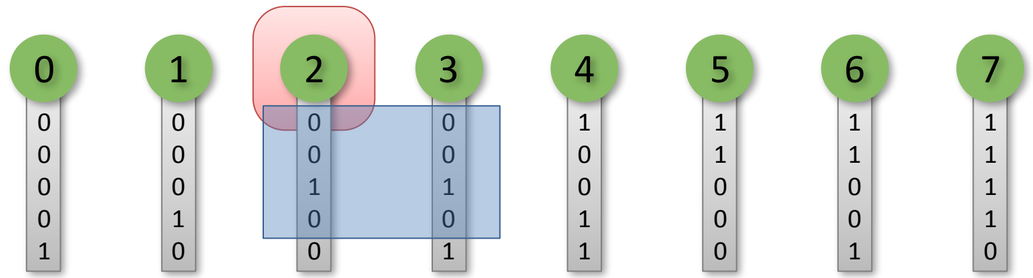
Algorithm


δ(0,3) = 2
3
?

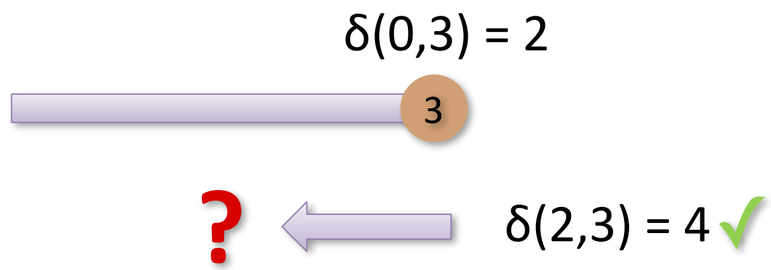

0
0 0 0 0 1
1
0 0 0 1 0
2
0 0 1 0 0
3
0 0 1 0 1
4
1 0 0 1 1
5
1 1 0 0 0
6
1 1 0 0 1
7
1 1 1 1 0

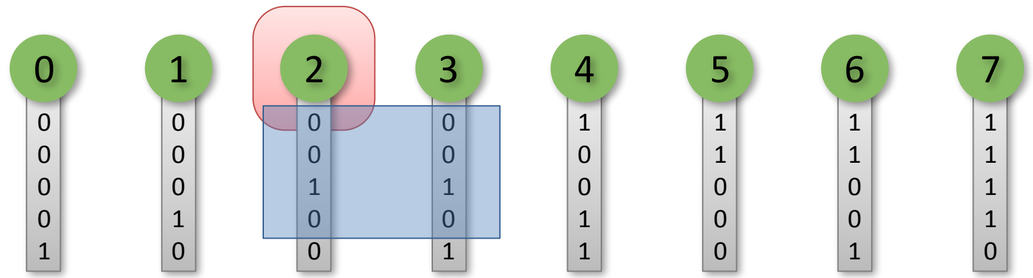
Algorithm


δ(0,3) = 2
3
?
δ(2,3) = 4

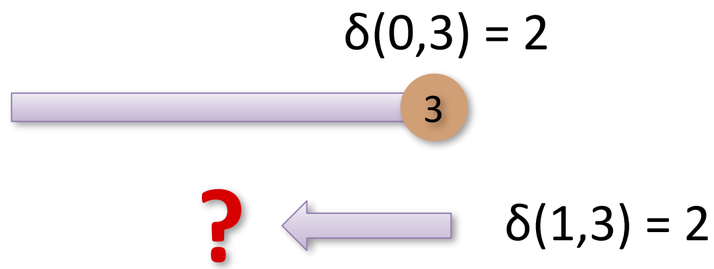

0
0 0 0 0 1
1
0 0 0 1 0
2
0 0 1 0 0
3
0 0 1 0 1
4
1 0 0 1 1
5
1 1 0 0 0
6
1 1 0 0 1
7
1 1 1 1 0

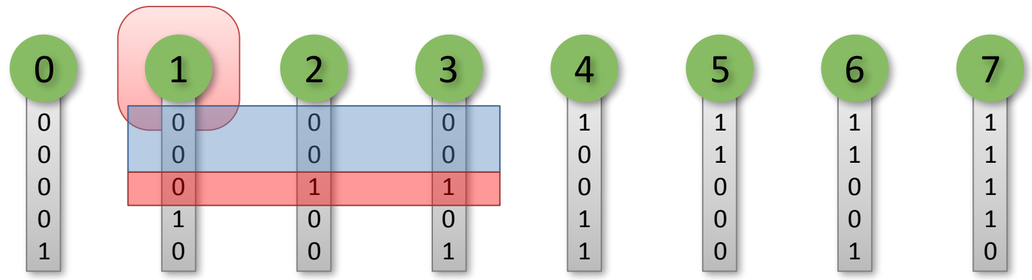
Algorithm


δ(0,3) = 2
3
?
δ(2,3) = 4

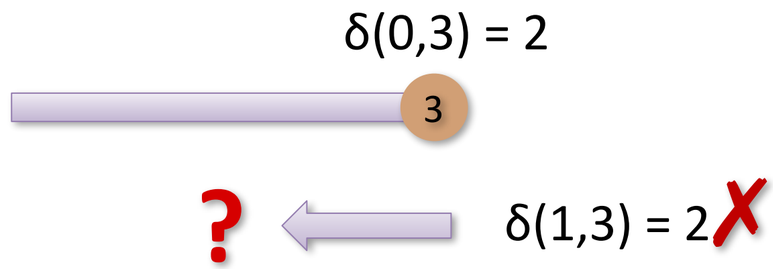

0
1
2
3
4
5
6
7
0 0 0 0 1
0 0 0 1 0
0 0 1 0 0
0 0 1 0 1
1 0 0 1 1
1 1 0 0 0
1 1 0 0 1
1 1 1 1 0

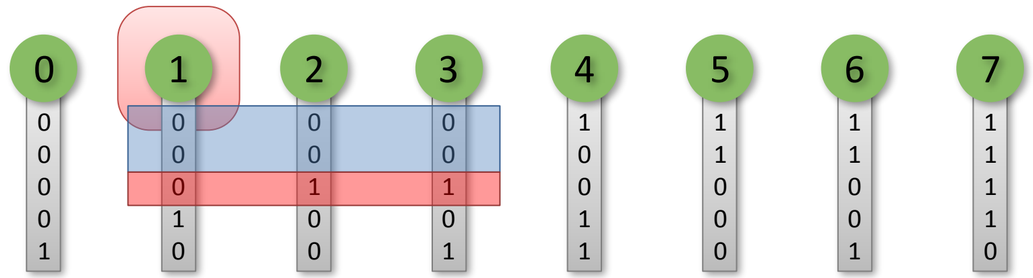
Algorithm


δ(0,3) = 2
3
?
δ(1,3) = 2

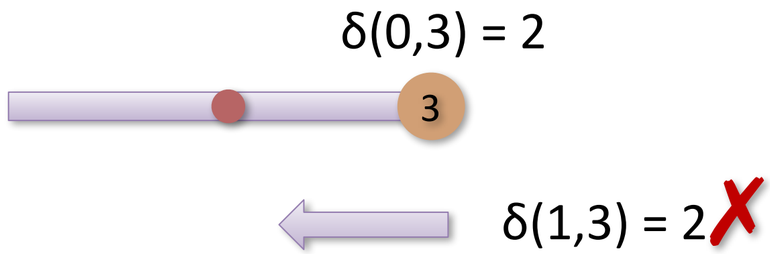

0
1
2
3
4
5
6
7
0 0 0 0 1
0 0 0 1 0
0 0 0 1 0
0 0 1 0 0
1 0 0 1 1
1 1 0 0 0
1 1 0 0 1
1 1 1 1 0

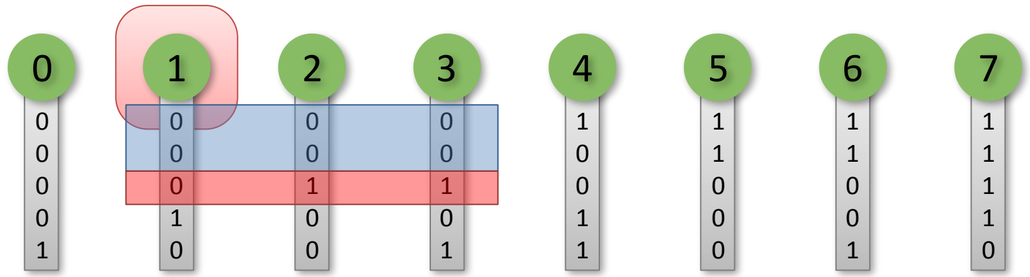
Algorithm


δ(0,3) = 2
3
?
δ(1,3) = 2

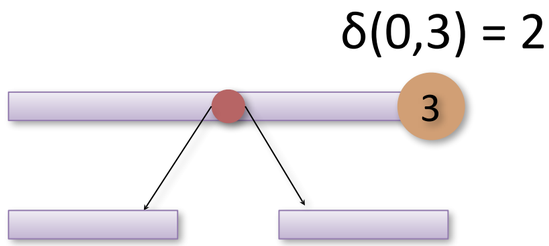

0
1
2
3
4
5
6
7
0 0 0 0 1
0 0 0 1 0
0 0 0 1 0
0 0 1 0 0
1 0 0 1 1
1 1 0 0 0
1 1 0 0 1
1 1 1 1 0

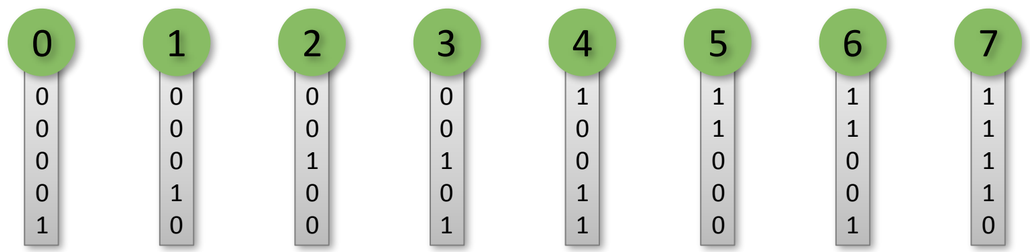
Algorithm


δ(0,3) = 2
3
δ(1,3) = 2

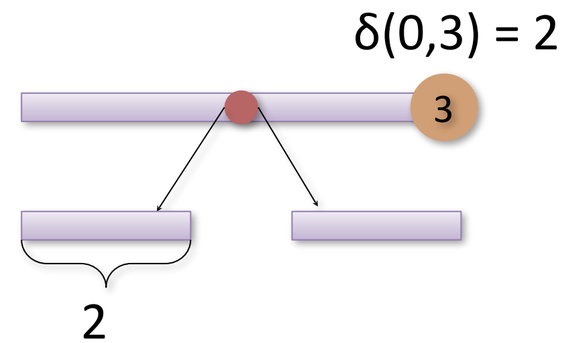

0
1
2
3
4
5
6
7
0 0 0 0 1
0 0 0 1 0
0 0 1 0 0
0 0 1 0 1
1 0 0 1 1
1 1 0 0 0
1 1 0 0 1
1 1 1 1 0

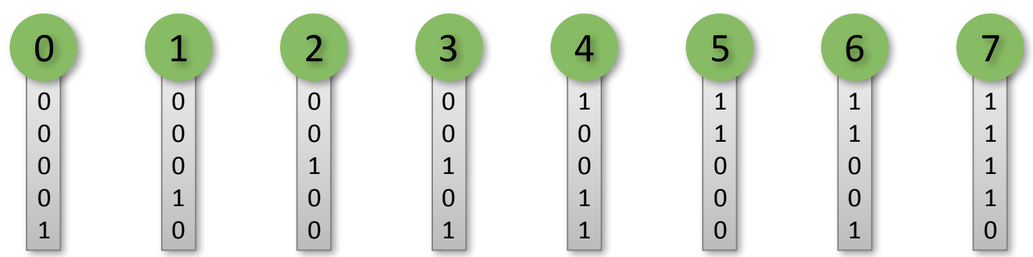
Algorithm


δ(0,3) = 2
3

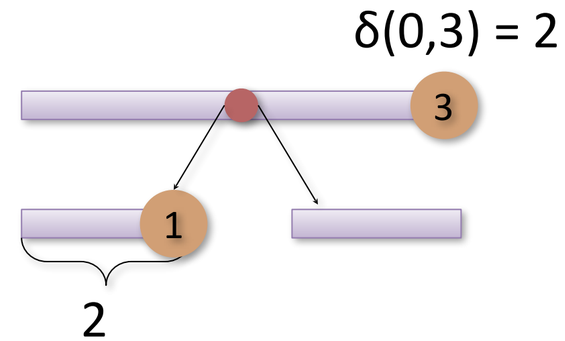

0
0
0
0
0
1
1
0
0
0
1
0
2
0
0
1
0
0
3
0
0
1
0
1
4
1
0
0
1
1
5
1
1
0
0
0
6
1
1
0
0
1
7
1
1
1
1
0

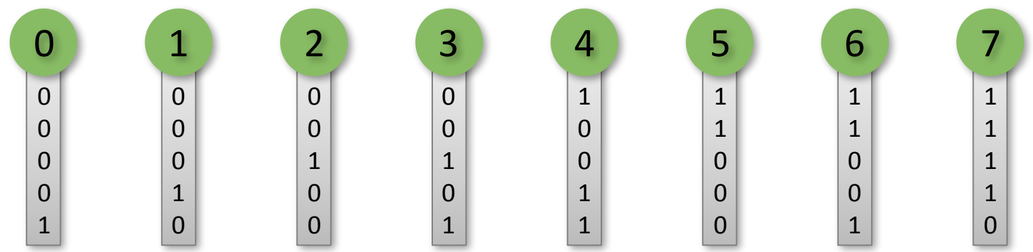
Algorithm


δ(0,3) = 2
3
2

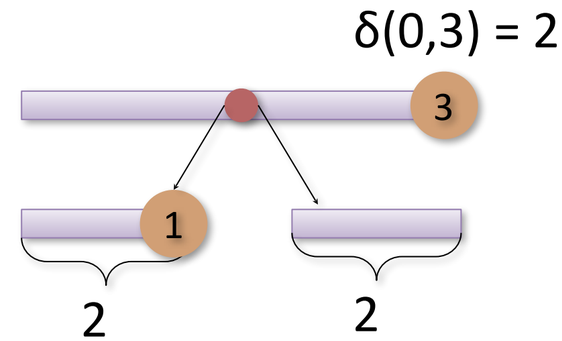

0
0 0 0 0 1
1
0 0 0 1 0
2
0 0 1 0 0
3
0 0 1 0 1
4
1 0 0 1 1
5
1 1 0 0 0
6
1 1 0 0 1
7
1 1 1 1 0

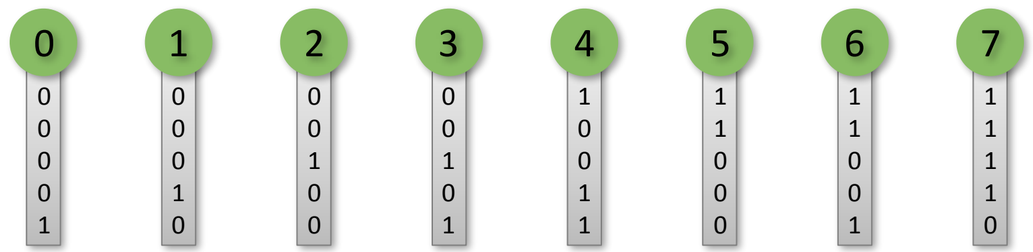
Algorithm


δ(0,3) = 2
3
1
2

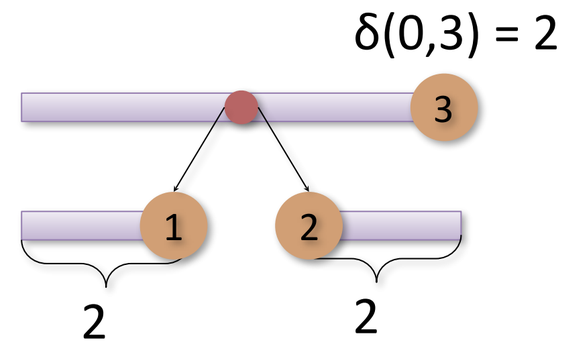

0
0 0 0 0 1
1
0 0 0 1 0
2
0 0 1 0 0
3
0 0 1 0 1
4
1 0 0 1 1
5
1 1 0 0 0
6
1 1 0 0 1
7
1 1 1 1 0

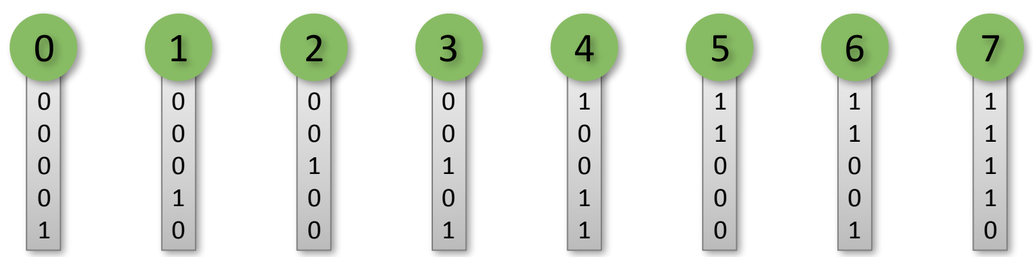
Algorithm


δ(0,3) = 2
3
1
2
2


0
0
0
0
0
1
1
0
0
0
1
0
2
0
0
1
0
0
3
0
0
1
0
1
4
1
0
0
1
1
5
1
1
0
0
0
6
1
1
0
0
1
7
1
1
1
1
0

Algorithm


δ(0,3) = 2
3
1
2
2
2


0
0
0
0
0
1
1
0
0
0
1
0
2
0
0
1
0
0
3
0
0
1
0
1
4
1
0
0
1
1
5
1
1
0
0
0
6
1
1
0
0
1
7
1
1
1
1
0

Algorithm

For each node i=0..n-2 in parallel:

1. Determine direction of the range
2. Expand the range as far as possible
3. Find where to split the range
4. Identify children

Algorithm

For each node i=0..n-2 in parallel:

1. **Determine direction of the range**
2. **Expand the range as far as possible**
3. **Find where to split the range**
4. **Identify children**

Algorithm

For each node i=0..n-2 in parallel:

1. **Determine direction of the range**
2. **Expand the range as far as possible**
3. **Find where to split the range**
4. **Identify children**

Binary search

$$\mathcal{O}(n \log h)$$

Fully Parallel BVH Construction Performance

- Performs more work than previous approach
- But, keeps full occupancy because of increased parallelism
- Execution time: 0.02 milliseconds
  - 50x improvement over top-down algorithm
  - Still performs better than top-down as # objects increases
    - Due to less divergence in this algorithm

Calculating Bounding Boxes

- Traverse tree in parallel (bottom-up)
  - Start from leaves, advance towards root
  - Each node reads bb from its two children, computes bb, outputs bb
- Terminate threads using per-node atomic flags
  - Kill first thread that enters, allow second
  - Each node is processed once after both its children are done

Calculating Bounding Boxes Performance

- Execution time: 0.06 milliseconds

- Lots of divergence, but not really important because:
  - The execution time is still relatively low compared to other parts (e.g., sort)
  - The processing mainly consists of memory reads and writes (thus, memory bandwidth is the limiting factor)

Summary of BVH Traversal and Construction

- Traversal
  - One thread per leaf node: use iterative algorithm to find potential collisions (0.25 ms)
- Construction
  1. One thread per object: calculate bounding box and Morton code (0.02 ms)
  2. Parallel radix sort: sort the objects according to Morton code (0.18 ms)
  3. One thread per internal node: generate BVH node hierarchy (0.02 ms)
  4. One thread per object: calculate node bounding boxes by walking tree towards root (0.06 ms)
- Total time: 0.28 ms

Trees, More Broadly

- Lots of things are trees!
- *Dynamic* (runtime) decision-making
- Different tree structures:
  - Fixed vs. arbitrary depth
  - Static (known) structure vs. dynamic structure
  - Fixed vs. maximum vs. arbitrary branching factor
    - e.g., 2-deep + dynamic structure + arbitrary branching factor is nested data parallelism == sparse matrix
  - Recursive or not