Pseudocode
This material is intended to help implementing the technique. It is not necessary for a high level understanding of the algorithm.

```
def advance(stream) : /* move read/write head forward  */
def allocRoot() : /* Create the root node              */
def allocNode(node, oct) : /* Create a child in an octant */
def allocQueue() : /* Create a queue                   */
def back(queue) : /* Get the item in the back of the q. */
def contains(node, point) : /* Is point inside the node? */
def fill(queue, instream) : /* Fill q. with leaf_max+1 pts */
def free(item) : /* Deallocates the item from memory  */
def front(queue) : /* Get the item at the front of the q. */
def isEmpty(queue) : /* Check if the queue is empty    */
def isSubdivisible(node) : /* Not at max. Morton depth? */
def level(node) : /* The depth of the node in the tree */
def next(stream) : /* Read the next item from the stream */
def pop(queue) : /* Remove an elem. from the queue    */
def push(queue, item) : /* Put item to the back of the q. */
def tell(stream) : /* Get write/read head location     */
def write(stream, item) : /* Write an item into stream  */
def chunk_levels = 3 : /* Subtree depth for chunking    */
def leaf_max = 16 : /* The refinement criteria          */
```

Algorithm 1: Various utility functions and constants used in Algorithms 3-4.

```
def finalizeInner(node):
for octant = 0...7 do
    if node.child[octant] != null then
        node.cluster += node.child[octant].cluster
        node.childIdx[octant] = node.child[octant].idx
    else
        node.childIdx[octant] = noIndex
    end
    free(node.child[octant])
end

def finalizeLeaf(queue, node, pointsProcessed):
node.firstPoint = pointsProcessed
node.numPoints = 0
while contains(node, front(pointQueue)) do
    pop(pointQueue)
    pointsProcessed++
    node.numPoints++
end
return pointsProcessed
```

Algorithm 2: Node finalization, i.e. computation of clusters and setting the point or child indices. The inner nodes can only be finalized once the children have been written out. The traversal order of Algorithms 3-4 guarantees that.

Input: pointInStream: Morton-sorted stream of points
Output: nodeOutStream: The output stream for the
            out-of-core node data
/* The main entry point for the algorithm. We do not repeat
   this part for the second version (Algorithm 4)   */

```
fill(queue, pointInStream)
pointsProcessed = 0
root = allocRoot()
buildRecurse(queue, root)
free(root)
return

def shouldRefine(node, queue):
a = contains(node, back(queue))
b = size(queue) > leafMax
c = isSubdivisible(node)
return a and b and c

def buildRecurse(queue, node):
if isEmpty(queue) then
    return
if !contains(node, front(queue)) then
    return
if shouldRefine(node, queue) then
    for octant = 0...7 do
        node.child[octant] = allocNode(node, octant)
        buildRecurse(queue, node.child[octant])
    end
    finalizeInner(node)
    node.idx = tell(nodeOutStream)
    write(nodeOutStream, node)
else
    /* The below while-loop is triggered only if the node
       was not subdivisible, i.e. the max Morton code bit
       depth was met                                    */
    while contains(node, next(pointInStream)) do
        push(queue, next(pointInStream))
        advance(pointInStream)
    end

    finalizeLeaf(queue, node, pointsProcessed)
    node.idx = tell(nodeOutStream)
    write(nodeOutStream, node)
    fill(queue, pointInStream)
end
```

Algorithm 3: The basic streaming octree building algorithm for Morton-sorted points. The algorithm is explained verbally in the actual paper. As discussed, in practice we use a slightly modified version (see Algorithm 4) that utilizes chunking to improve the coherency of the nodes on disk.

```
def buildRecurse2(queue, node):
if isEmpty(queue) then
    return
if !contains(node, front(queue)) then
    return
if level(node) % chunk_levels = 0 then
    writeOutQueue = allocQueue();
if shouldRefine(node, queue) then
    for octant = 0...7 do
        node.child[octant] = allocNode(node, octant)
        buildRecurse2(queue, node.child[octant])
    end
else
    /* The below while-loop is triggered only if the node
       was not subdivisible, i.e. the max Morton code bit
       depth was met                                     */
    while contains(node, next(pointInStream)) do
        push(queue, next(pointInStream))
        advance(pointInStream)
    end

    finalizeLeaf(queue, node, pointsProcessed)
    fill(queue, pointInStream)
end
push(writeOutQueue, node)
if level(node) % chunk_levels = 0 then
    while isEmpty(writeOutQueue) do
        wnode = front(writeOutQueue)
        if !isLeaf(wnode) then
            finalizeInner(wnode)
        wnode.idx = tell(nodeOutStream)
        write(nodesOutStream, wnode)
        pop(writeOutQueue)
    end
    free(writeOutQueue)
end
```

Algorithm 4: The out-of-core octree construction code with chunking. In comparison to Algorithm 3, this version queues up the nodes for write-out, and is able to organize the nodes of the octree in a more coherent order. This algorithm is bootstrapped the same way as Algorithm 3, so we do not repeat that code here.