Algorithm 1 Insert

```
next_cid := 0                                    ▷ next chunk identifier

procedure INSERT(node, key, val)                 ▷ node is a pointer to root
    return INSERT_AUX(node, key, val, ϵ)▷ return ptr to new node

procedure INSERT_AUX(node, key, val, last_chunk)
    if node = ϵ then                             ▷ if the tree is empty:
        chunk := NEW_CHUNK(next_cid)             ▷ first chunk
        node := INSERT_IN_CHUNK(chunk, key, val)
        node→chunk := chunk                      ▷ node becomes chunk root
        chunk→root :=node                        ▷ ditto
        return node
    if node→chunk ≠ ϵ then                       ▷ if node is the chunk root:
        c := node→chunk                          ▷ let c be this chunk
        if c →size = C_p then                    ▷ if c is a full chunk:
            SPLIT_CHUNK(node)                    ▷ split node (i.e., c's root)
    else
        c := last_chunk                          ▷ chunk root is a higher node
    if node→is_leaf then                         ▷ if node is a leaf:
        node_chunk := node→chunk                 ▷ keep its chunk, if any
        node→chunk := ϵ                          ▷ node can't be chunk root
        if key < node→key then                   ▷ normal AVL left insert
            left := INSERT_IN_CHUNK(c, key, val)
            node := NEW_INNER(node→key, left, node, 1)
        else                                     ▷ normal AVL right insert
            right := INSERT_IN_CHUNK(c, key, val)
            node := NEW_INNER(key, node, right, 1)
        node→right→inner_node := node ▷ leaf points to its inner
        node→chunk := node_chunk                 ▷ inner gets kept chunk
        if node→chunk ≠ ϵ then
            c→root := node
    else                                         ▷ if node is an inner node:
        if key < node→key then                   ▷ go down left or...
            node→left := INSERT_AUX(node→left, key, val, c)
        else                                     ▷ ...go down right...
            node→right := INSERT_AUX(node→right,key,val,c)
    UPDATE_HEIGHT(node)                          ▷ new height from children heights
    return BALANCE(node)                         ▷ if needed, rotate to keep balance

procedure UPDATE_HEIGHT(node)
    node→height :=                               ▷ height gets max children height plus one
              max(node→left→height,node→right→height) + 1

procedure BALANCE(node)
    h := node→left→height - node→right→height
    if h < -1 then                               ▷ if right branch is bigger than left branch:
        return ROTATE_RL(node)                   ▷ rotate [right] left
    if h > 1 then                                ▷ if left branch is bigger than right branch:
        return ROTATE_LR(node)                   ▷ rotate [left] right
    return node
```

Algorithm 2 Auxiliary procedures

```
procedure NEW_INNER(key, ln, rn, ht)             ▷ create inner node
    new_inode := allocate new inner node
    new_inode→key := key
    new_inode→left := ln
    new_inode→right := rn
    new_inode→height := ht
    return new_inode

procedure NEW_CHUNK(cid)                         ▷ create chunk
    new_c := allocate new chunk
    new_c→cid := cid
    new_c→size := 0
    new_c→root := ϵ
    return new_c

procedure INSERT_IN_CHUNK(c, key, val)           ▷ insert key,val
    c→leaf[c→size].key := key
    c→leaf[c→size].value := val
    node_addr := pointer to c→leaf[c→size]
    c→size := c→size + 1
    return node_addr

procedure DELETE_FROM_CHUNK(c, key)
    for node in c→leaf do
        if node.key = key then                   ▷ found leaf
            SWAP(node, c→leaf[c→size])           ▷ swap with the last
    deallocate c→leaf[c→size]                    ▷ free last leaf
    c→size := c→size - 1                         ▷ decrement number of leaves
    if c→size = 0 then                           ▷ chunk is empty
        next_cid := next_cid - 1                 ▷ decrement number of chunks
        last_chunk := GET_CHUNK(next_cid)        ▷ get chunk with
highest id
        last_chunk→cid := c→cid                  ▷ replace chunk's id
        deallocate c                             ▷ free empty chunk

procedure SPLIT_CHUNK(node)                      ▷ split chunk rooted at node
    new_c := NEW_CHUNK(node→chunk→cid)
    DFS(node→left, node, new_c)
    node→left→chunk := new_c
    new_c→root := node→left                      ▷ assign left chunk
    next_cid := next_cid + 1
    new_c := NEW_CHUNK(next_cid)
    DFS(node→right, node, new_c)
    new_c→root := node→right                     ▷ assign right chunk
    node→right→chunk := new_c
    deallocate node→chunk
    node→chunk := ϵ                              ▷ x is no longer chunk root
    return

procedure DFS(ptr, pnt, c)
    if ptr→is_leaf then
        c→leaf[c→size].key := ptr→key
        c→leaf[c→size].value := ptr→value
        c→leaf[c→size].i_node := ptr→i_node
        if ptr→key < pnt→key then                ▷ assign parent
            pnt→left := pointer to chunk→leaf[c→size]
        else
            pnt→right := pointer to chunk→leaf[c→size]
        c→size := c→size + 1                     ▷ one more leaf in chunk
    else
        DFS(ptr→left, ptr, c)
        DFS(ptr→right, ptr, c)
    return
```

Algorithm 3 Delete

```
procedure DELETE(node, key)
    if node.key = key and node.is_leaf then        ⊳ only one node
        DELETE_FROM_CHUNK(node.chunk, key)
        return ϵ
    return DELETE_AUX(ϵ, node, key, ϵ)⊳ return ptr to new node

procedure DELETE_AUX(node, key, last_chunk)
    if node→chunk ≠ ϵ then        ⊳ if node is the chunk root:
        c := node→chunk        ⊳ let c be this chunk
    else
        c := last_chunk        ⊳ chunk root is a higher node
    if node→left→key = key and node→left→is_leaf then
        if c = ϵ then        ⊳ chunk is necessarily on the left
            c := node→left→chunk
        DELETE_FROM_CHUNK(c, key)
        promoted := node→right
        if node→chunk ≠ ϵ then
            promoted→chunk := node→chunk        ⊳ node is a chunk root
        deallocate node        ⊳ no need for inner-node
        return promoted
    if node→key = key and node→right→is_leaf then
        if c = ϵ then        ⊳ chunk is necessarily on the right
            c := node→right→chunk
        DELETE_FROM_CHUNK(c, key)
        promoted := node→left
        if node→chunk ≠ ϵ then
            promoted→chunk := node→chunk        ⊳ node is a chunk root
        deallocate node        ⊳ no need for inner-node
        return promoted
    if node→key = key and node→right→left→is_leaf then
        if c = ϵ then        ⊳ chunk is lower
            if node→right→chunk ≠ ϵ then        ⊳ chunk on right
                c := node→right→chunk
                node→rightNode→rightNode→chunk = c
                node→chunk = ϵ
            else
                c := node→right→left→chunk ⊳ chunk on the left
        DELETE_FROM_CHUNK(c, key)
        aux := node→right
        node→key := aux→key
        node→right := aux→right
        deallocate aux
        return node
    if node→key = key then
        n, p := DELETE_LEAF(node→right, c)
        node→key = n→key
        node→right := p
        deallocate n
    else
        if key < node→key then
            node→left := DELETE_AUX(node→left, key, c)
        else
            node→right := DELETE_AUX(node→right, key, c)
    UPDATE_HEIGHT(node)        ⊳ new height from children heights
    return BALANCE(node)        ⊳ if needed, rotate to keep balance
```

```
procedure DELETE_LEAF(node, c)
    if node→chunk ≠ ϵ then        ⊳ if node is the chunk root:
        c := node→chunk        ⊳ let c be this chunk
    else
        c := last_chunk        ⊳ chunk root is a higher node
    if node→left→is_leaf then        ⊳ found leaf
        if c = ϵ then
            c := node→left→chunk
        DELETE_FROM_CHUNK(c, node→left→key)
        if node→chunk ≠ ϵ then
            node→right→chunk := node→chunk
        return node, node→right
    else
        n, new_root := DELETE_LEAF(node→left, c)
        node→left := new_root
        UPDATE_HEIGHT(node)⊳ new height from children heights
        return n, BALANCE(node)        ⊳ if needed, rotate to keep balance
```

Algorithm 4 Rotations

```
procedure ROTATE_RL(node)   ⊳ normal AVL [right] left rotation
    rl_height := node→right→left→height
    rr_height := node→right→right→height
    if rl_height > rr_height then
        node→right := ROTATE_R(node→right)
        UPDATE_HEIGHT(node)
    return ROTATE_L(node)

procedure ROTATE_LR(node)   ⊳ normal AVL [left] right rotation
    ll_height := node→left→left→height
    lr_height := node→left→right→height
    if ll_height < lr_height then
        node→left := ROTATE_L(node→left)
        UPDATE_HEIGHT(node)
    return ROTATE_R(node)

procedure ROTATE_L(node)        ⊳ node is the pivot
    if node→chunk ≠ ϵ then⊳ if subtree rooted at node in a chunk:
        node→chunk→root := node→right
        node→right→chunk := node→chunk⊳ node's right child...
        node→chunk := ϵ        ⊳ ...becomes new chunk root
    else        ⊳ else, rotation may involve two chunks
        if node→right→chunk ≠ ϵ then   ⊳ if r child is chunk root:
            SPLIT_CHUNK(node→right)
    new_pivot := node→right        ⊳ rotation in three steps: one, ...
    node→right := new_pivot→left        ⊳ ...two, and...
    new_pivot→left := node        ⊳ ...three!
    UPDATE_HEIGHT(node)        ⊳ update moved node height
    UPDATE_HEIGHT(new_pivot)        ⊳ update moved node height
    return new_pivot

procedure ROTATE_R(node)        ⊳ node is the pivot
    if node→chunk ≠ ϵ then⊳ if subtree rooted at node in a chunk:
        node→left→chunk := node→chunk   ⊳ node's left child...
        node→chunk := ϵ        ⊳ ...becomes new chunk root
    else        ⊳ else, rotation may involve two chunks
        if node→left→chunk ≠ ϵ then   ⊳ if l child is chunk root:
            SPLIT_CHUNK(node→left)
    new_pivot := node→left        ⊳ rotation in three steps: one, ...
    node→left := new_pivot→right        ⊳ ...two, and...
    new_pivot→right := node        ⊳ ...three!
    UPDATE_HEIGHT(node)        ⊳ update moved node height
    UPDATE_HEIGHT(new_pivot)        ⊳ update moved node height
    return new_pivot
```

Algorithm 5 Reconstruction

```
hash: trusted Merkle-root hash, stored in the block header
n_chunks: number of chunks of tree, stored in the block header
c_roots: set with all chunk roots, initially empty
t_parts: set with all tree parts, initially empty

procedure RECEIVE(chunk, cproof)
    c_root, s_parts :=                  ▹ build subtree rooted at chunk root and...
            BUILD_SUBTREE(chunk)                            ▹ ...all tree parts
    if VALID(hash, c_root, cproof) then                     ▹ is chunk valid?
        c_roots := c_roots ∪ {c_root}                   ▹ keep all chunk roots
        t_parts := t_parts ∪ s_parts                       ▹ and all tree parts
        if |c_roots| = n_chunks then                    ▹ received all chunks?
            SORT(t_parts)                   ▹ sort parts by height in desc order
            t_root := t_parts[0]             ▹ tree root is the deepest node
            for i in 1..|t_parts| do                       ▹ include all parts
                INSERT_NODE(t_root, t_parts[i])
            COMPUTE_HASH(t_root)                  ▹ compute all tree hashes
            return t_root
    return ϵ

procedure BUILD_SUBTREE(chunk)
    c_root := chunk→root                                ▹ chunk root in chunk
    if c_root→height = 0 then                   ▹ if root has no inner node...
        return c_root, {c_root}                          ▹ no subtree to build
    c_root :=                               ▹ create inner node for chunk root
                NEW_INNER(c_root→key, ϵ, ϵ, root→height)
    parts := {c_root}               ▹ all tree parts, initially only chunk root
    L := chunk→leaves                                  ▹ prepare to sort leaves:
    SORT(L)                               ▹ sort by height in descending order
    for i in 0..(|L|-1) do                     ▹ first pass: create inner nodes
        if L[i]→height > c_root→height then                       ▹ above root?
            parts := parts ∪    ▹ to be used after all chunks received
                        {NEW_INNER(L[i]→key, ϵ, ϵ, l→height)}
        if 0 < L[i]→height < c_root→height then                   ▹ below root?
            inner_node :=                                   ▹ create inner node
                        NEW_INNER(L[i]→key, ϵ, ϵ, L[i]→height)
            INSERT_NODE(c_root, inner_node)▹ insert it in subtree
    for i in 0..(|L|-1) do                     ▹ second pass: insert leaf nodes
        INSERT_NODE(c_root, L[i])                       ▹ insert leaf in subtree
        chunk→leaf[i].height := 0                           ▹ adjust leaf height
    return c_root, parts

procedure INSERT_NODE(x, r)                       ▹ insert node r in (sub)tree x
    if x = ϵ then
        return r                                 ▹ insert here, end recursion
    if r→key < x→key then                                   ▹ search down left
        x→left := INSERT_NODE(x→left, r)
    else                                                   ▹ search down right
        x→right := INSERT_NODE(x→right, r)
```