Search Trees

Juan Mendivelso

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Search Tree

- Structure that stores objects, each object identified by a key value, in a tree structure.
- The keys are drawn from an ordered set.
- Two keys can be compared in constant time.
- These comparisons are used to guide the access to a specific object by its key.
- The search starts from the root. Each node contains a key that is compared with the query key.
- One can go to different nodes depending on whether the query key is smaller or larger than the key in the node.

Search Tree

- This structure is fundamental to most data structure.
- It allows many variations.
- It is a building block for most more complex data structures.

- Search trees are one method to implement dictionaries:.
- A dictionary is a structure that stores objects, identified by keys, and supports the operations find, insert and delete.
- There are other ways to implement dictionaries, like hash tables.
- Also, there are applications of search trees different from dictionaries.

CONTENTS

1. TWO MODELS OF SEARCH TREES

Two Models of Search Tree

1. Leaf Tree

- Take left branch if query key is smaller than node key; otherwise take the right branch.
- Repeat until a leaf is found.
- The keys in the interior nodes are only for comparison.
- All objects are in the leaves.

2. Node Tree

- Take left branch if query key is smaller than node key.
- Take the right branch if the query key is larger than the node key.
- Take the object contained in the node if they are equal.

Two Models of Search Tree

1. Leaf Tree

- Each interior node has a left and a right subtree.
- It is a full binary tree: the degree of each node is either 0 or 2.
- The structure is more regular.
- One comparison for two possible outcomes: left or right.

2. Node Tree

- Left or right subtree may be missing.
- Two comparisons for three possible outcomes: node, left or right.

Two Models of Search Tree

1. Leaf Tree

- Interior nodes serve only for comparisons and may reappear in the leaves for the identification of the objects.
- It is possible that there are keys used for comparison that do not belong to any object, for example if the object has been deleted.
- All keys used for comparison must be distinct.
- Each key appears at most twice (one for comparison and as a leaf).

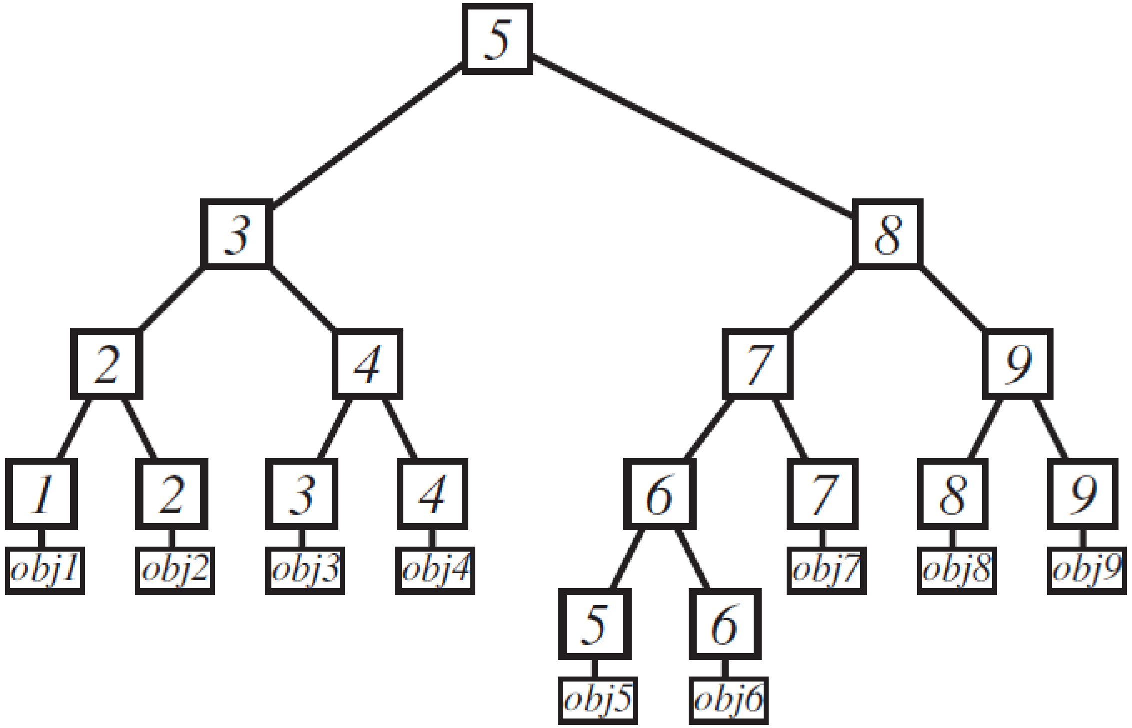
2. Node Tree

- Each node appears only once, together with its object.
- It is preferred for most textbooks because such books don't make the distinction between the keys and the objects

Two Models of Search Tree

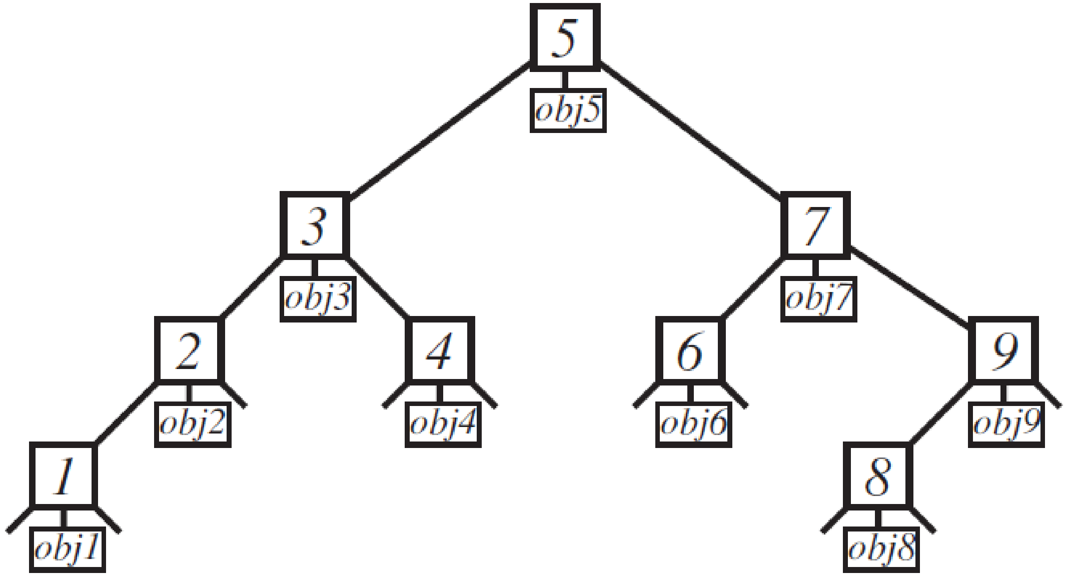
1. Leaf Tree

- A tree of height h contains at most 2^h objects (leaves).



2. Node Tree

- A tree of height h contains at most $2^{h+1} - 1$ objects (all nodes).



Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Two Models of Search Tree

1. Leaf Tree

- It is the preferred in these presentations.
- This is because it requires fewer number of comparisons, and its structure is more regular.
- It will be used for most data structures.

2. Node Tree

- It is used for splay trees.

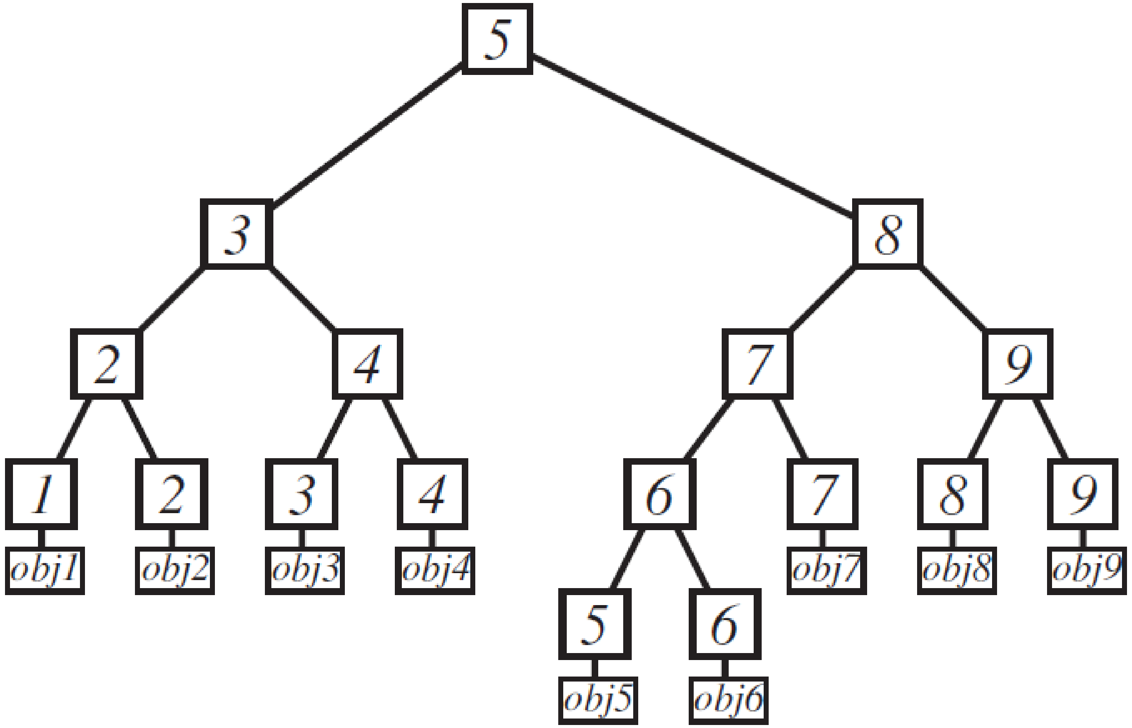
2. LEAF TREES

Leaf Trees

```
typedef struct tr_n_t {key_t          key;
                       struct tr_n_t   *left;
                       struct tr_n_t  *right;
                /* possibly additional information */
                     } tree_node_t;
```

- A node `*n` is a leaf if `n->right` = NULL. Then `n->left` points to the object stored in that leaf and `n->key` contains the object's key.
  - If `root->left` = NULL, then the tree is empty.
  - If `root->left` ≠ NULL and `root->right` = NULL, then `root` is a leaf and the tree contains only one object.
  - If `root->left` ≠ NULL and `root->right` ≠ NULL, then `root->right` and `root->left` point to the roots of the right and left subtrees. For each node `*left_node` in the left subtree, we have `left_node->key` < `root->key`, and for each node `*right_node` in the right subtree, we have `right_node->key` ≥ `root->key`.

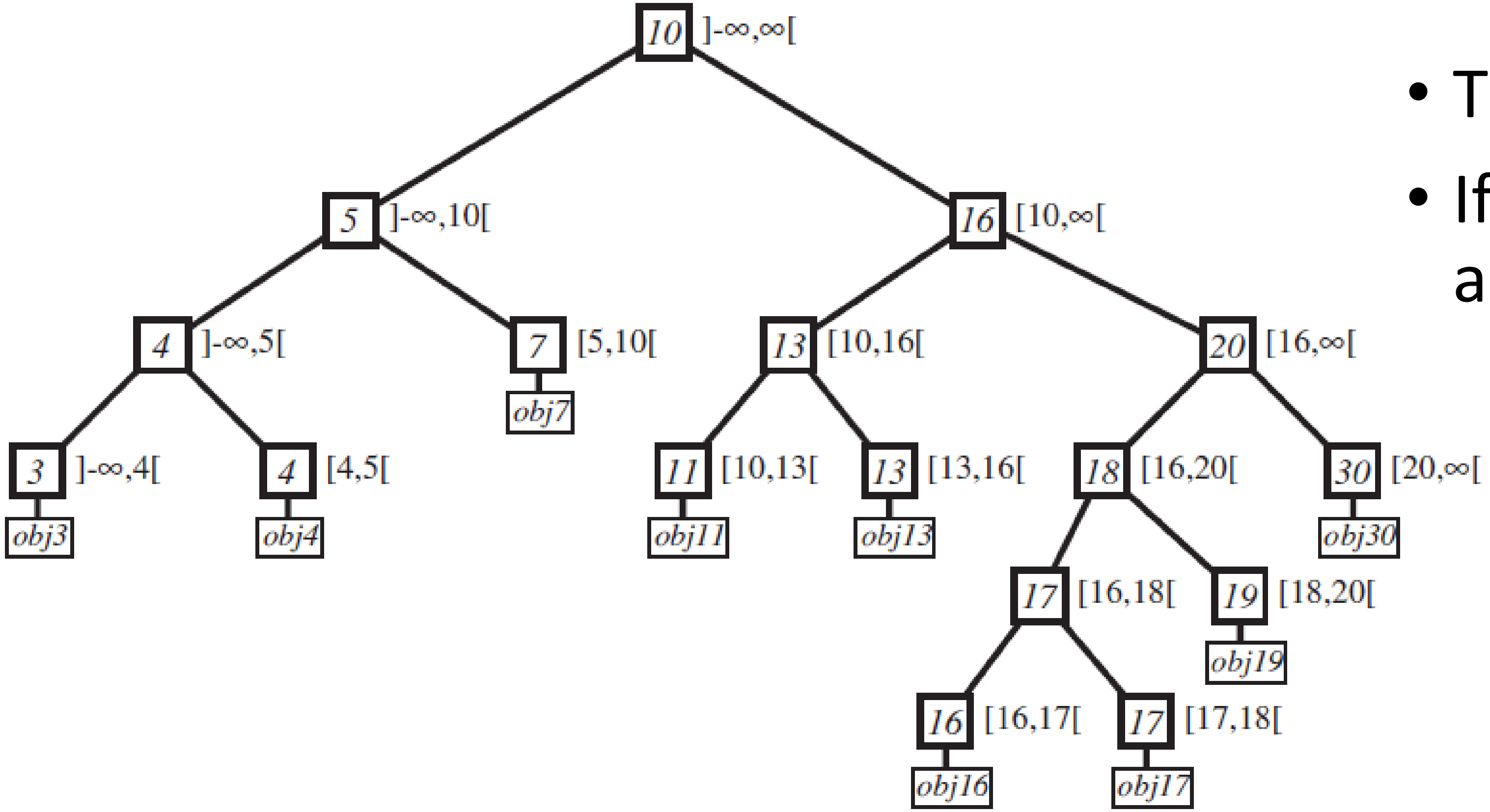
Leaf Trees




```
tree_node_t *create_tree(void)
{   tree_node_t *tmp_node;
    tmp_node = get_node();
    tmp_node->left = NULL;
    return( tmp_node );
}
```


Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Intervals in Leaf Trees



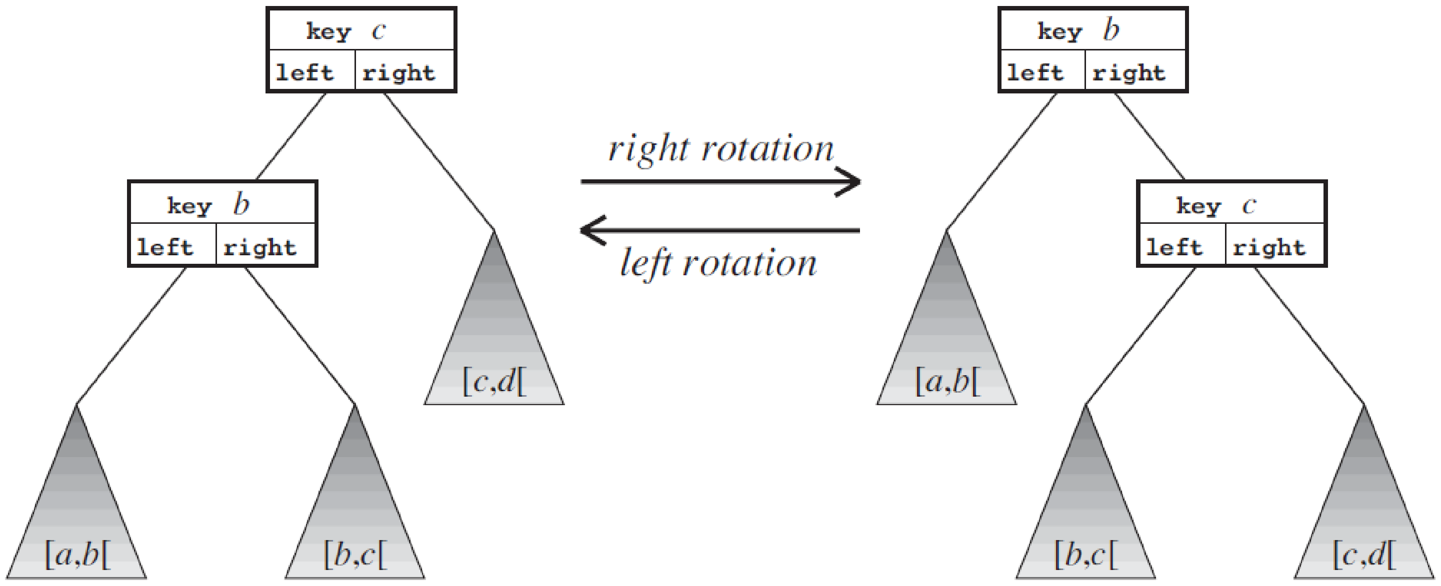
- We can associate each tree node with an interval:

  the interval of all possible key values that can be reached through this node.

- The interval of `root` is $]-\infty, \infty[$.
- If `*n` is an interior node associated with `[a,b[`, then
  - `n->key ∈ [a,b[`
  - `n->left ∈ [a,n->key[`
  - `n->right ∈[n->key,b[`

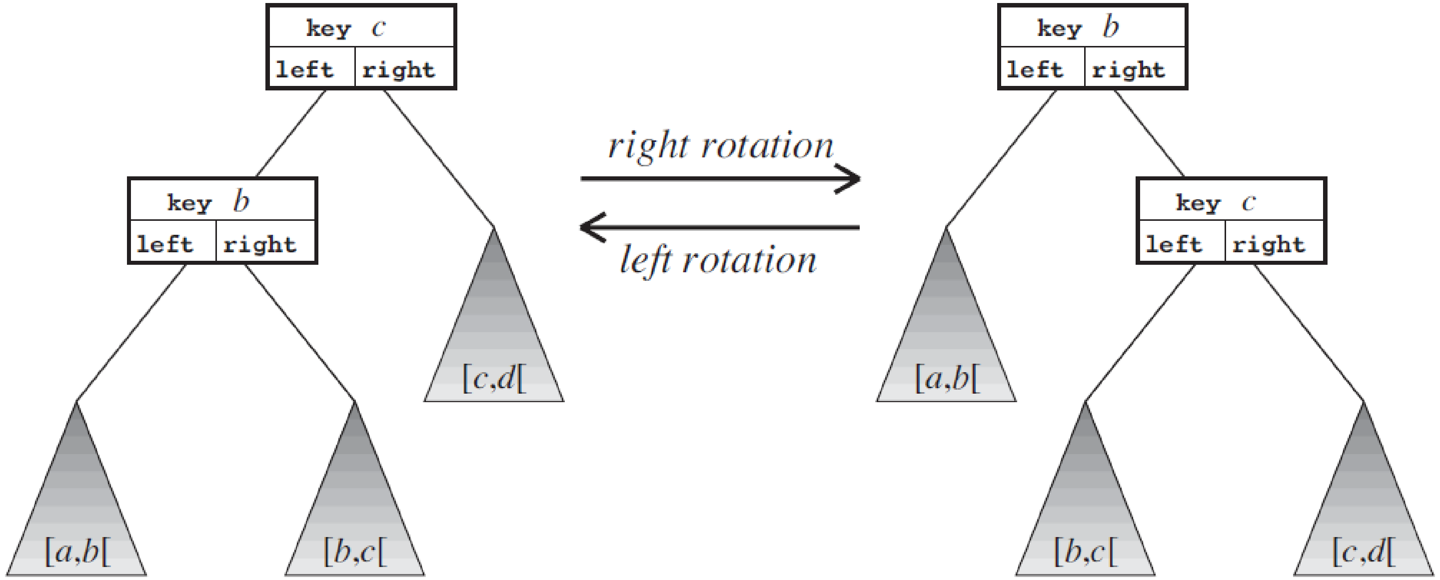
Rotations in Leaf Trees

- The same set of (key, object) pairs can be organized in many distinct correct search trees.
- There are two operations that transform a correct search tree in a different correct search tree for the same set.
- They are the left and right rotations.



Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Rotations in Leaf Trees



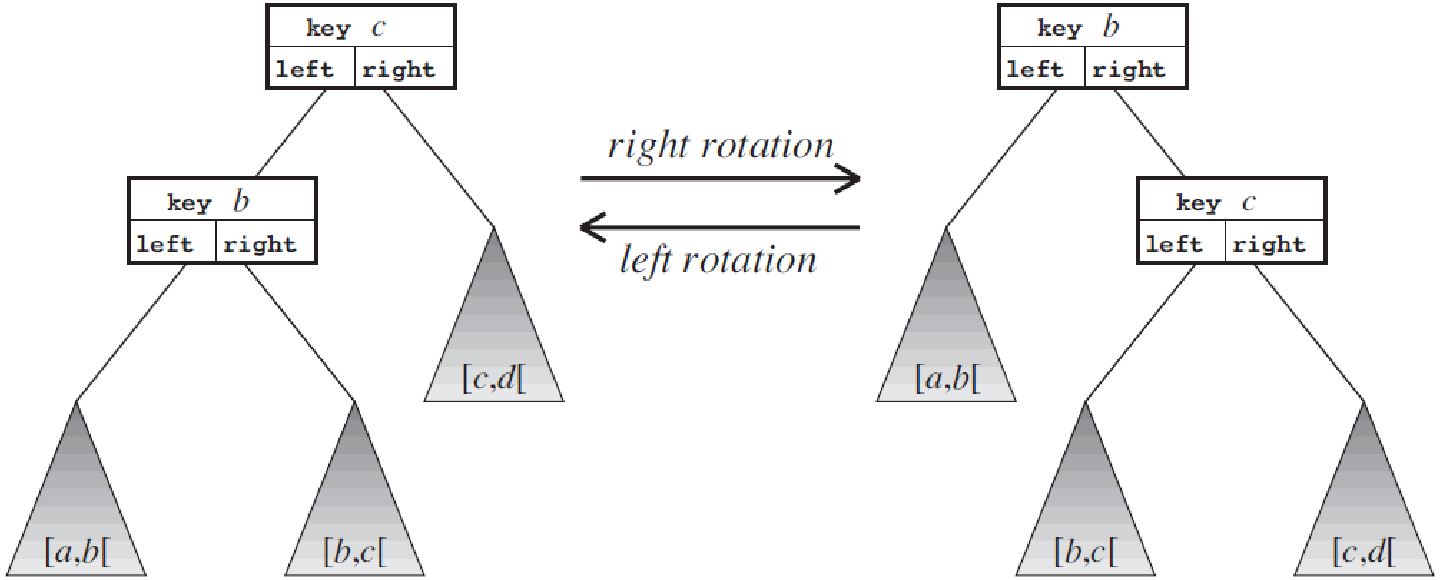

```
void left_rotation(tree_node_t *n)
{   tree_node_t *tmp_node;
    key_t        tmp_key;
    tmp_node = n->left;
    tmp_key  = n->key;
    n->left  = n->right;
    n->key   = n->right->key;
    n->right = n->left->right;
    n->left->right = n->left->left;
    n->left->left  = tmp_node;
    n->left->key   = tmp_key;
}
```

- We move the content of the nodes around, but the node `*n` needs to be the root of the subtree because there are pointers from higher levels in the tree that point to `*n`.
- This is because there are no pointers to the parent.

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Rotations in Leaf Trees

```
void right_rotation(tree_node_t *n)
{   tree_node_t *tmp_node;
    key_t          tmp_key;
    tmp_node = n->right;
    tmp_key  = n->key;
    n->right = n->left;
    n->key   = n->left->key;
    n->left  = n->right->left;
    n->right->left = n->right->right;
    n->right->right  = tmp_node;
    n->right->key   = tmp_key;
}
```



- Any correct search tree for some set of (key, object) pairs can be transformed into each other by sequence of rotations.

Height of a Leaf Tree

- The central property which makes some search trees good and others bad is the height.

- The **height** of a search tree is the maximum length of a path, in number of edges, from the root to a leaf.

- The **depth** of a node is the distance, in number of edges, from the root to such node.

Height of a Leaf Tree

- The maximum number of leaves of a search tree of height h is 2^h.
- The minimum number of leaves is $h + 1$ because
  - a tree of height h must have at least one interior node at each depth $0, \ldots, h - 1$.
  - a tree with h interior nodes has $h + 1$ leaves.

- **Theorem:** A leaf tree for n objects has height $\lceil \lg n \rceil \leq h \leq n - 1$.

- **Theorem:** A leaf tree for n objects has average depth of at least $\lg n$ and at most $\frac{(n-1)(n+2)}{2n} \approx \frac{1}{2}n$. Why?

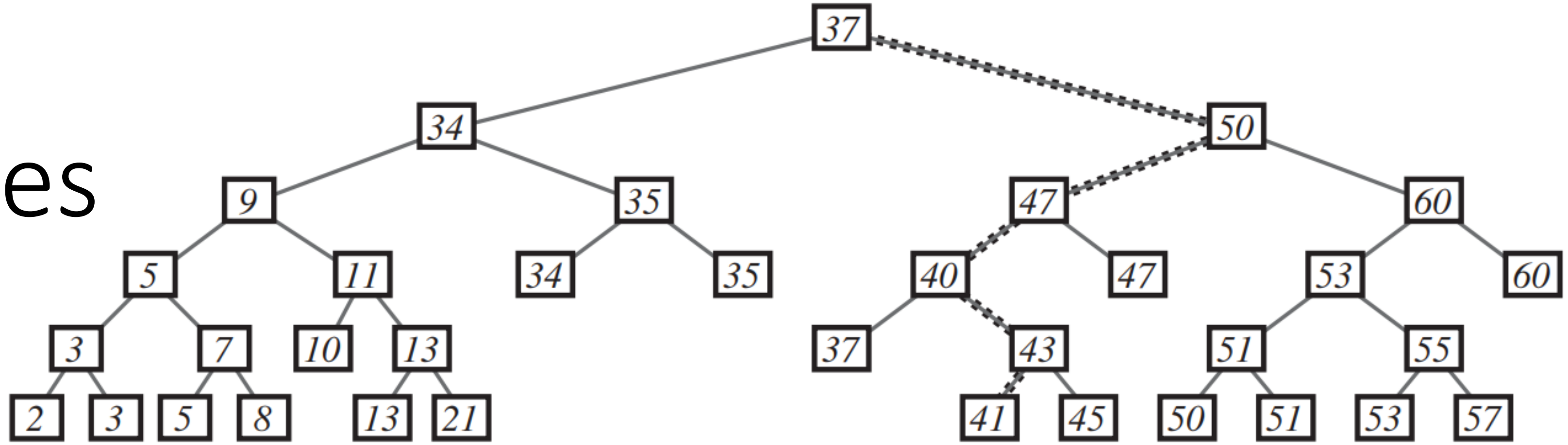
3. OPERATIONS ON LEAF TREES

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Operations on Leaf Trees

- `find( tree, query_key)`: Returns the object associated with `query_key`, if there is one;
- `insert( tree, key, object )`: Inserts the (key, object) pair in the tree; and
- `delete( tree, key)`: Deletes the object associated with `key` from the tree.

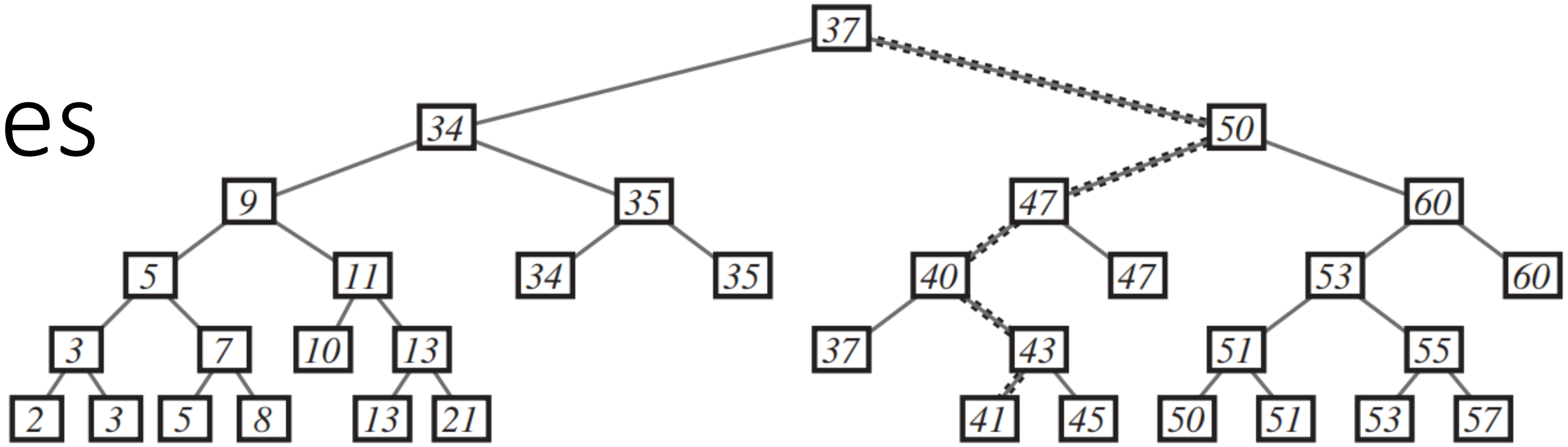
Find on Leaf Trees




```
object_t *find(tree_node_t *tree,
               key_t query_key)
{  tree_node_t *tmp_node;
   if( tree->left == NULL )
     return(NULL);
   else
   {  tmp_node = tree;
      while( tmp_node->right != NULL )
      {   if( query_key < tmp_node->key )
                tmp_node = tmp_node->left;
          else
                tmp_node = tmp_node->right;
      }
      if( tmp_node->key == query_key )
         return( (object_t *) tmp_node->left );
      else
         return( NULL );
   }
}
```

- First, we check if the tree is empty.
- Then, one just follows the associated interval structure to the leaf.
- If the object is there, return it; otherwise return `NULL`.

Find on Leaf Trees (Recursive)




```
object_t *find(tree_node_t *tree,
               key_t query_key)
{  if( tree->left == NULL ||
       (tree->right == NULL &&
        tree->key != query_key ) )
      return(NULL);
   else if (tree->right == NULL &&
            tree->key == query_key )
      return( (object_t *) tree->left );
   else
   {  if( query_key < tree->key )
         return( find(tree->left, query_key) );
      else
         return( find(tree->right, query_key) );
   }
}
```

- First, we check if the tree is empty or a leaf that does not contain the `query_key`.
- If it indeed contains the `query_key`, return the object.
- Otherwise, one just follows the associated interval structure to the leaf.

Insert on Leaf Trees

```
int insert(tree_node_t *tree, key_t new_key,
           object_t *new_object)
{  tree_node_t *tmp_node;
   if( tree->left == NULL )
   {  tree->left = (tree_node_t *) new_object;
      tree->key  = new_key;
      tree->right  = NULL;
   }
   else
   {  tmp_node = tree;
      while( tmp_node->right != NULL )
      {   if( new_key < tmp_node->key )
                tmp_node = tmp_node->left;
          else
                tmp_node = tmp_node->right;
      }
      /* found the candidate leaf. Test whether
         key distinct */
      if( tmp_node->key == new_key )
         return( -1 );
```

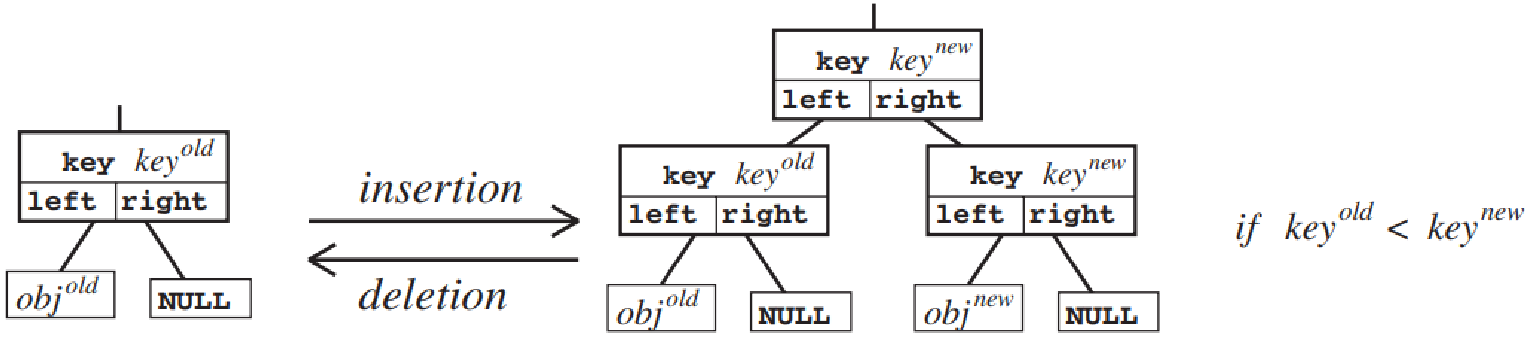
- If the tree is empty, the new node is added at the root.
- Then, it is similar to find.
- Let us call `tmp_node` as the leaf node where the new key must be inserted.
- It has to create a new interior node and a new leaf node.
- For the moment, assume all the keys are unique.
- Otherwise, it is an error.

Insert on Leaf Trees

```
/* key is distinct, now perform the insert */
{  tree_node_t *old_leaf, *new_leaf;
   old_leaf = get_node();
   old_leaf->left = tmp_node->left;
   old_leaf->key = tmp_node->key;
   old_leaf->right  = NULL;
   new_leaf = get_node();
   new_leaf->left = (tree_node_t *)
   new_object;
   new_leaf->key = new_key;
   new_leaf->right  = NULL;
   if( tmp_node->key < new_key )
   {    tmp_node->left  = old_leaf;
        tmp_node->right = new_leaf;
        tmp_node->key = new_key;
   }
   else
   {    tmp_node->left  = new_leaf;
        tmp_node->right = old_leaf;
   }
}
```

```
}
return( 0 );
}
```

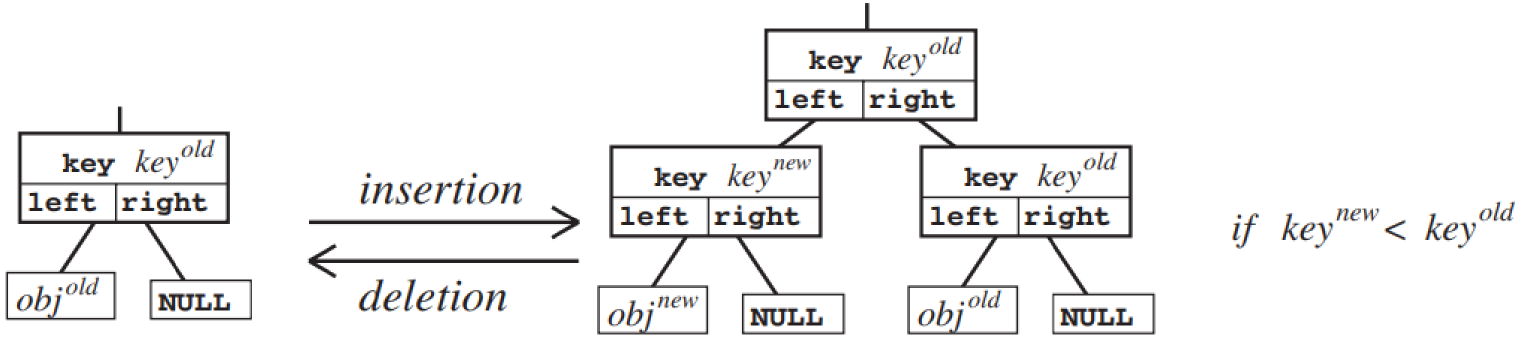
- We must create leaf nodes for:
  - former value of `tmp_node` (`old_leaf`)
  - new key inserted (`new_leaf`)
- The node `tmp_node`, which was a leaf, is now an interior node that points to `old_leaf` and `new_leaf`.
- Its key is the largest key of the children.
- The left an right pointers depend on whether `tmp_node->key < new_leaf->key`.



Insert on Leaf Trees

```
/* key is distinct, now perform the insert */
{  tree_node_t *old_leaf, *new_leaf;
   old_leaf = get_node();
   old_leaf->left = tmp_node->left;
   old_leaf->key = tmp_node->key;
   old_leaf->right  = NULL;
   new_leaf = get_node();
   new_leaf->left = (tree_node_t *)
   new_object;
   new_leaf->key = new_key;
   new_leaf->right  = NULL;
   if( tmp_node->key < new_key )
   {    tmp_node->left  = old_leaf;
        tmp_node->right = new_leaf;
        tmp_node->key = new_key;
   }
   else
   {    tmp_node->left  = new_leaf;
        tmp_node->right = old_leaf;
   }
 }
}
return( 0 );
}
```

- We must create leaf nodes for:
  - former value of `tmp_node` (`old_leaf`)
  - new key inserted (`new_leaf`)
- The node `tmp_node`, which was a leaf, is now an interior node that points to `old_leaf` and `new_leaf`.
- Its key is the largest key of the children.
- The left an right pointers depend on whether `tmp_node->key < new_leaf->key`.



Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Delete on Leaf Trees

```
object_t *delete(tree_node_t *tree,
                 key_t delete_key)
{   tree_node_t *tmp_node, *upper_node,
    *other_node;
    object_t *deleted_object;
    if( tree->left == NULL )
       return( NULL );
    else if( tree->right == NULL )
    {   if(  tree->key == delete_key )
        {   deleted_object =
                             (object_t *) tree->left;
            tree->left = NULL;
            return( deleted_object );
        }
        else
            return( NULL );
    }
    else
    {   tmp_node = tree;
```

- If the tree is empty or the key is not in the tree, we return NULL.
- If the root is a leaf and it contains the `delete_key`, we just return it and eliminate the object.
- Otherwise, we must find the corresponding leaf like in `find`.

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Delete on Leaf Trees

```
else
{   tmp_node = tree;
    while( tmp_node->right != NULL )
    {   upper_node = tmp_node;
        if( delete_key < tmp_node->key )
        {   tmp_node   = upper_node->left;
            other_node = upper_node->right;
        }
        else
        {   tmp_node   = upper_node->right;
            other_node = upper_node->left;
        }
    }
```

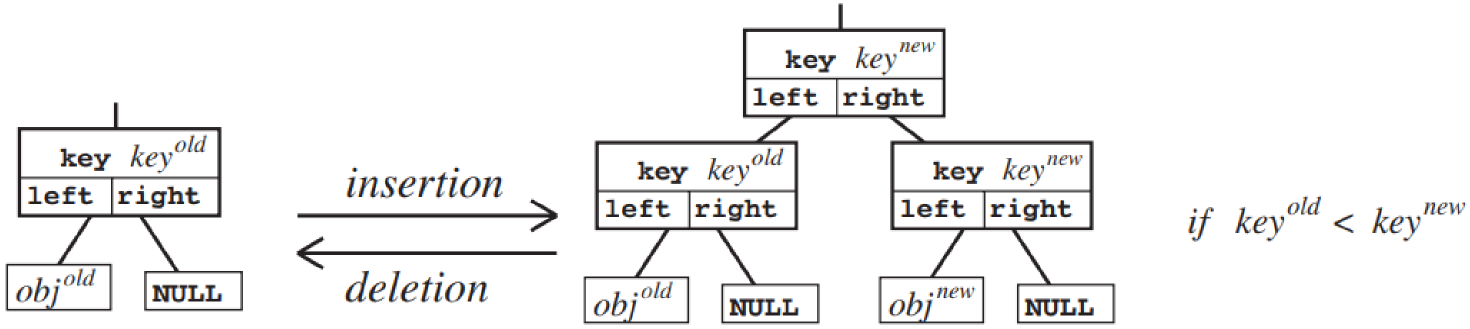
- Otherwise, we must find the corresponding leaf like in `find`.
- When we are deleting a leaf, we need to delete an interior node above it.
- So, we must keep track of the current node (`tmp_node`) and its upper neighbor (`upper`).
- The sibling of `tmp_node` is `other_node`.
- At the end of the loop, `tmp_node` points to a leaf where `delete_key` may be in.
- But `other_node` may point to a leaf or an interior node.

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Delete on Leaf Trees

```
if( tmp_node->key != delete_key )
    return( NULL );
else
{   upper_node->key   = other_node->key;
    upper_node->left  = other_node->left;
    upper_node->right = other_node->right;
    deleted_object = (object_t *)
    tmp_node->left;
    return_node( tmp_node );
    return_node( other_node );
    return( deleted_object );
  }
 }
}
```

- We must store and return the deleted object.
- Also, we must delete a leaf (tmp_node) and an interior node (other_node).
- But before, we copy the key and children of other_node in upper_node.

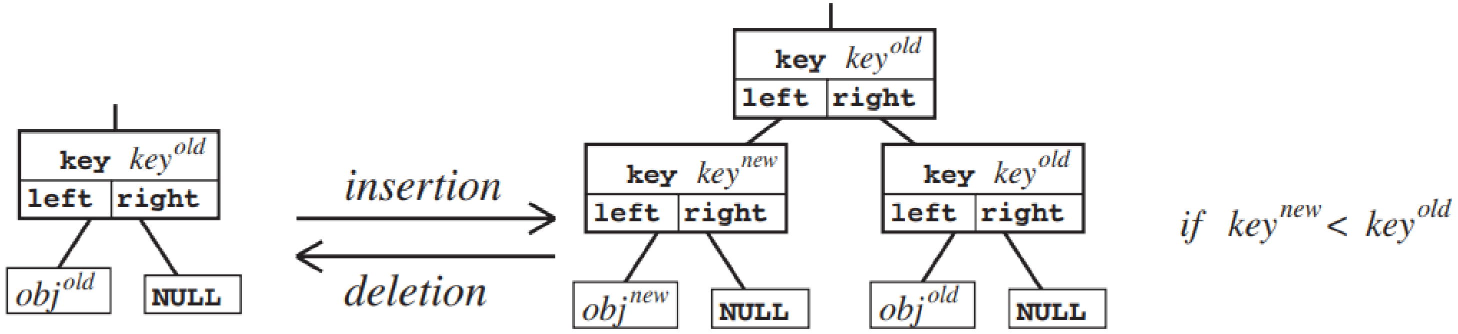


Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Delete on Leaf Trees

- We must store and return the deleted object.
- Also, we must delete a leaf (tmp_node) and an interior node (other_node).
- But before, we copy the key and children of other_node in upper_node.

```
    if( tmp_node->key != delete_key )
       return( NULL );
    else
    {  upper_node->key   = other_node->key;
       upper_node->left  = other_node->left;
       upper_node->right = other_node->right;
       deleted_object = (object_t *)
       tmp_node->left;
       return_node( tmp_node );
       return_node( other_node );
       return( deleted_object );
    }
  }
}
```



Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

4. RETURNING FROM LEAF TO ROOT

Returning from Leaf to Root

- We normally start from the root and follow pointers toward a leaf.
- Sometimes, it is necessary to do the opposite. For example, for rebalancing.
- We can do this with any of the following methods:
  1. A stack
  2. Back pointers
  3. Back pointer with lazy update
  4. Reversing the path

1. A Stack

- If we push pointers to all traversed nodes on a stack during descent to a leaf, then we can take the nodes from the stack in the correct reversed order afterwards.
- This is the most economic solution: it does not put additional information into the tree structure.
- The maximum size of the stack needed is the height of the tree.
- This solution is implicitly used in any recursive implementation of the search trees.

2. Back Pointers

- If each node contains also a pointer to its parent, then we have a path up from any node back to the root.

- It requires additional memory. However, now it is not a problem.

- However, this pointer has to be corrected in each operation. This requires time and may generate errors.

3. Back Pointer with Lazy Update

- Each node has a pointer to its parent.
- However, it is only entered during descent in the tree.
- Then, we have a correct path from the leaf we reached to the root.
- We don’t need to correct back pointers during all operations of the tree.
- But then, a back pointer is only assumed to be correct for the nodes on the path from the leaf we just reached.

4. Reversing the Path

- We can keep back pointers for the path even without an extra entry for a back pointer.
- We just reverse the forward pointers as we go down the tree.
- For instance, if we go left, the left pointer is used as a back pointer.
- When we go up again, the correct forward pointers must be restored.
- This method does not use extra space.
- But it causes many additional problems because the leaf tree structure is temporarily destroyed.
- It is not recommended to use it.

5. DEALING WITH NONUNIQUE KEYS

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Dealing with Nonunique Keys

- In practical applications, it is not uncommon that there are several objects with the same key.
- In database applications, there are queries to list all objects with a given attribute value.
- Thus, we must adapt the `find`, `insert` and `delete` operations.

Dealing with Nonunique Keys

{ `find` returns all objects whose key is the given query key in output-sensitive time $O(h + k)$, where h is the height of the tree and k is the number of elements that `find` returns.

{ `insert` always inserts correctly in time $O(h)$, where h is the height of the tree.

{ `delete` deletes all items of that key in time $O(h)$, where h is the height of the tree.

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Dealing with Nonunique Keys

The obvious way to realize this behavior is to keep all elements of the same key in a linked list below the corresponding leaf of the leaf tree.

- `find` just returns all the elements of that list.
- `insert` always inserts at the beginning of that list.
- `delete` requires additional information.
  - We need and additional node, between the leaf and the linked list, which contains pointers to the beginning and the end of the list.
  - Then, we can transfer the entire list with $O(1)$ operations to the free list of our dynamic memory allocation structure.

6. QUERIES FOR THE KEYS IN AN INTERVAL

Queries for the Keys in an Interval

- If the keys are subject to small errors, we might not know the key exactly, so we may want to look for an interval of keys instead.
- We give an interval [a,b[and want to find all keys that are contained in this interval.
- This can be adapted in several ways:
  1. Organize the leaves in a doubly linked list.
  2. Change the query and not the data structure.

1. Leaves in a Doubly Linked List

- We can move in $O(1)$ time from a leaf to the next larger and the next smaller leaf.
- This require a change in the insertion and deletion functions to maintain the list.
- But it is an easy change that takes only $O(1)$ additional time. How?
- The query method is also almost the same. It takes $O(k)$ additional time if it lists a total of k keys in the interval.

2. Change in the Query and not in the Tree

- We go down the tree with the query interval instead of the query key:
  - We go left if `[a,b[ < node->key`.
  - We go right if `node->key ≤ [a,b[`.
  - We go both left and right if `a < node->key ≤ b`.
- We store all the branches that we still need to explore on a stack. The nodes we visit this way are:
  - Nodes on the search path for a.
  - Nodes on the search path for b.
  - Nodes on the tree between these paths.

2. Change in the Query and not in the Tree

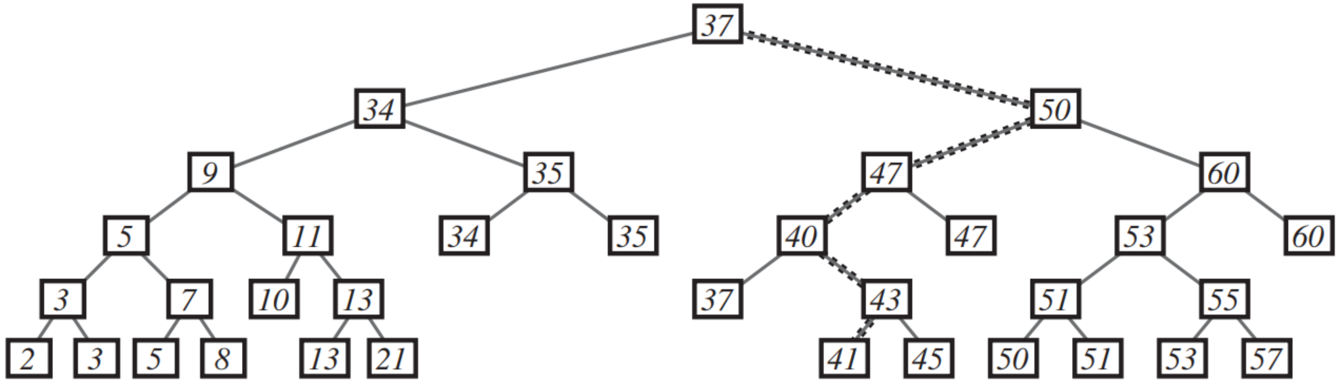
- If there are i interior nodes between these paths, there must be at least $i + 1$ leaves between these paths.
- So if this method lists k leaves, the total number of nodes visited is at most twice the number of nodes visited in a normal find operation plus $O(k)$. Why?
- This method is slightly slower than the first one, but it requires no change in the `insert` and `delete` operations.

2. Implementation

- The output of the operation is potentially long.
- Thus, we return man objects instead of a single result.
- For that, we create a linked list of the (key, object) pairs found in the query interval, which is linked by the right pointers.
- In particular, each node of such list is a tree node `node`, where the key is stored in `node->key` and the object is stored in `node->right`.

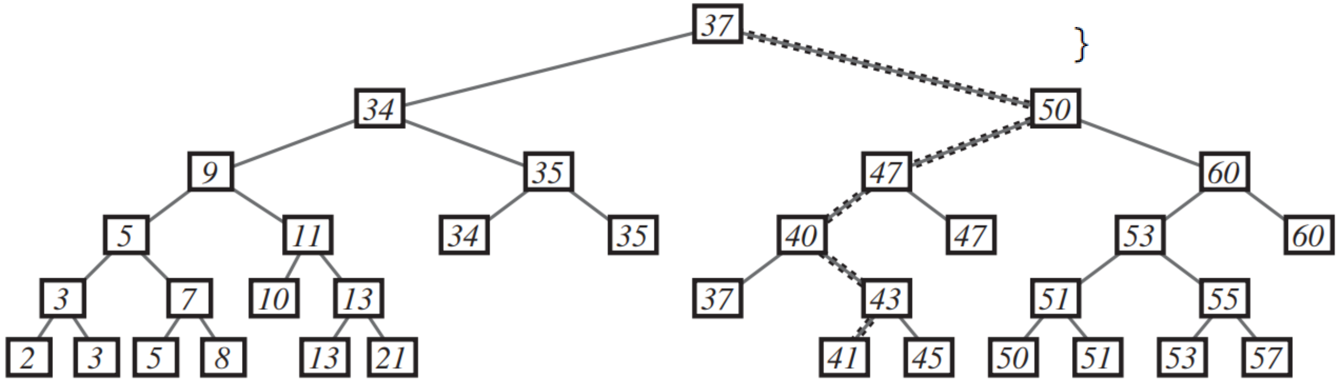
2. Implementation

```
tree_node_t *interval_find(tree_node_t *tree,
                           key_t a, key_t b)
{  tree_node_t *tr_node;
   tree_node_t *result_list, *tmp;
   result_list = NULL;
   create_stack();
   push(tree);
   while( !stack_empty() )
   {  tr_node = pop();
      if( tr_node->right == NULL )
      {  /* reached leaf, now test */
         if( a <= tr_node->key &&
             tr_node->key < b )
         {  tmp = get_node();
            /* leaf key in interval */
            tmp->key  = tr_node->key; /*
            copy to output list */
            tmp->left = tr_node->left;
            tmp->right = result_list;
            result_list = tmp;
         }
      } /* not leaf, might have to follow down */
```



2. Implementation

```
    } /* not leaf, might have to follow down */
    else if ( b <= tr_node->key )
    /* entire interval left */
       push( tr_node->left );
    else if ( tr_node->key <= a )
    /* entire interval right */
       push( tr_node->right );
    else    /* node key in interval,
                  follow left and right */
    {  push( tr_node->left );
       push( tr_node->right );
    }
 }
 remove_stack();
 return( result_list );
}
```



Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

7. OPTIMAL SEARCH TREES

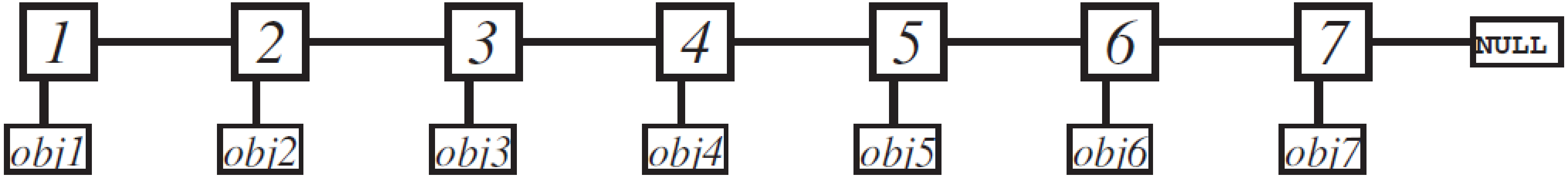
Optimal Search Trees

- We consider search trees as a static structure: no inserts and no deletes.
- Then, there is no problem of rebalancing the tree.
- We should build it as good as possible: minimum height.
- Because a tree of height h has at most 2^h, an optimal search tree for n items has height $\lceil \lg n \rceil$.
- There are two ways to construct it:
    1. Bottom up
    2. Top Down

Bottom Up Construction

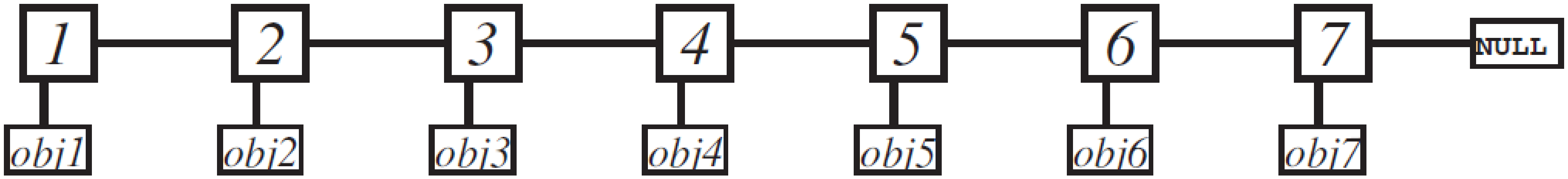
```
typedef struct tr_n_t {key_t        key;
                       struct tr_n_t   *left;
                       struct tr_n_t  *right;
                    /* possibly additional information */
                       } tree_node_t;
```

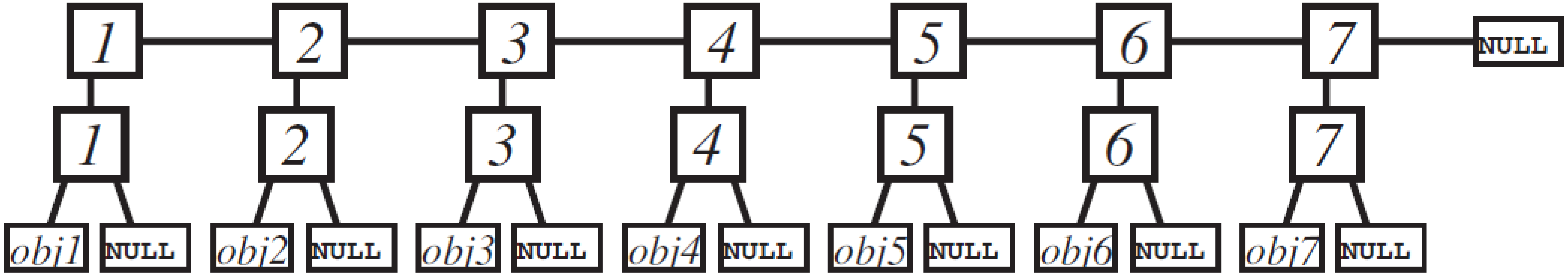
- The input is a sorted list of the (key, object pairs).
- In particular, it is a list of `tree_node_t` nodes where each node contains:
  - `key`
  - `left` points to the object
  - `right` points to the next element (or NULL at the end of the list).



Bottom Up Construction

- Create a list as a list of one-element trees (leaves).

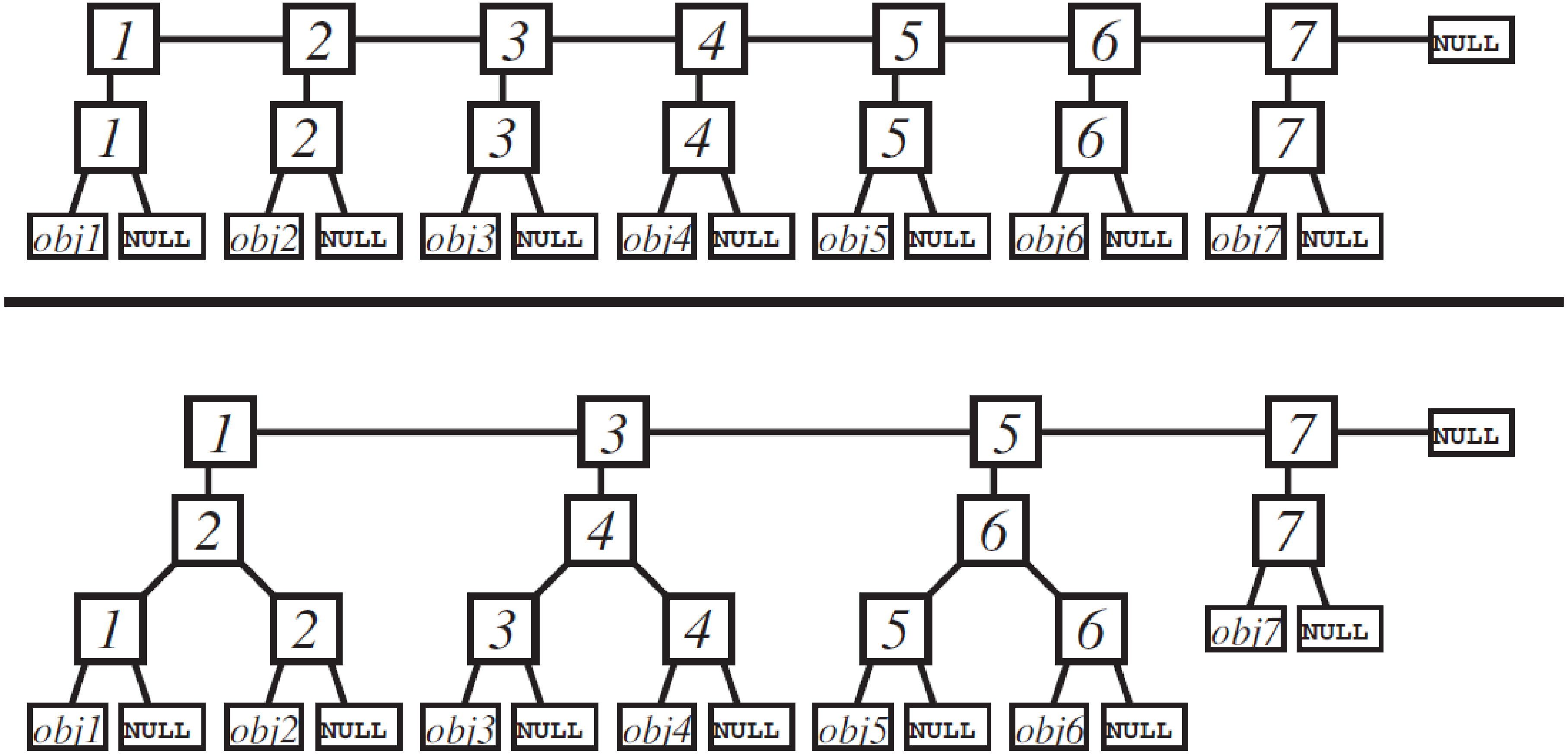




Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

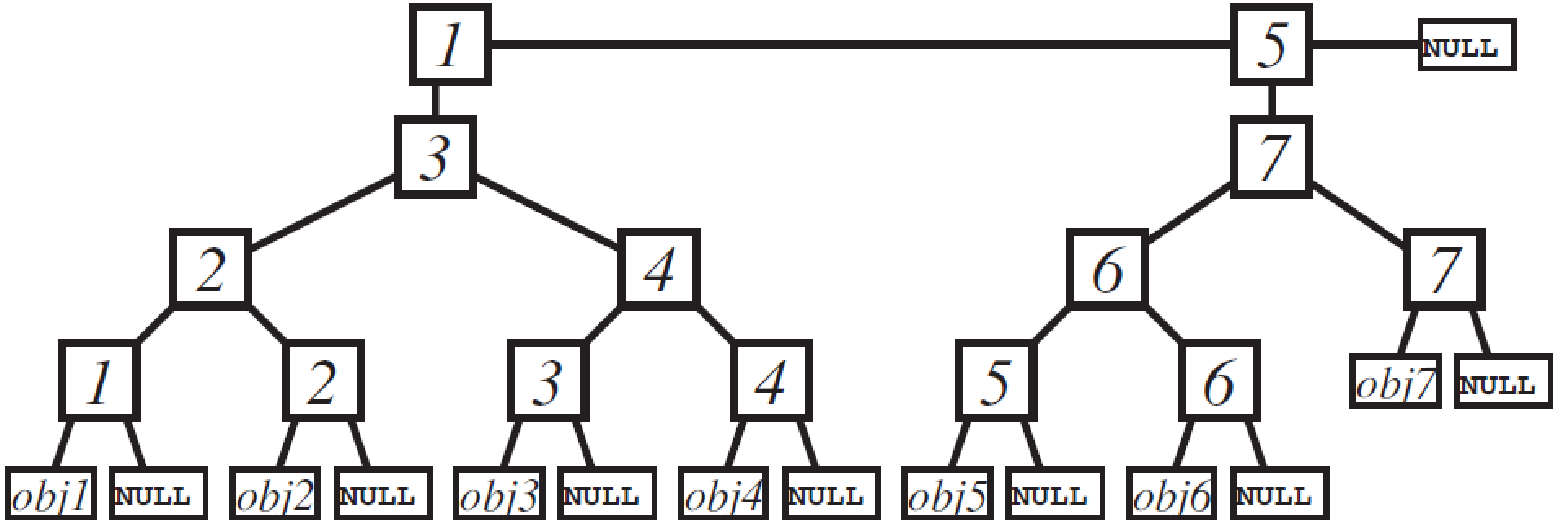
Bottom Up Construction

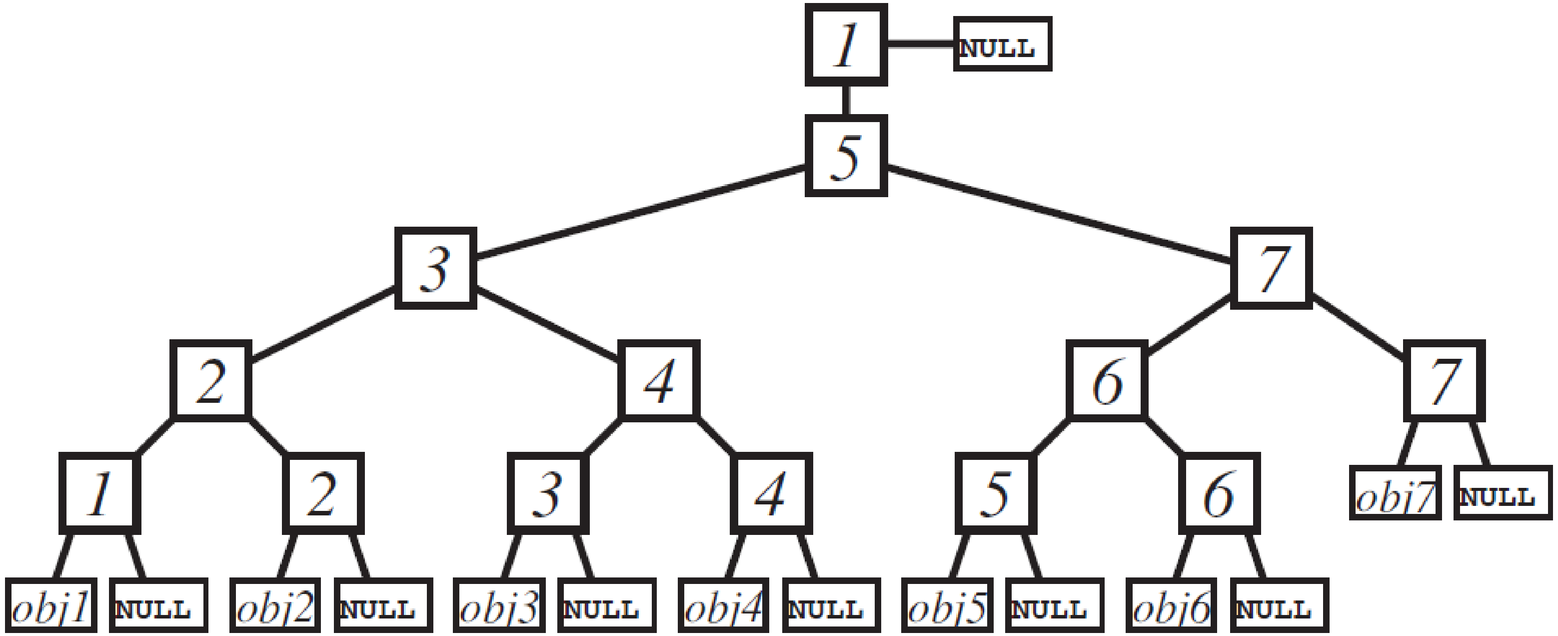
- Go through the list , joining two consecutive trees.
- The nodes of the previous list are attached as leaves.



Bottom Up

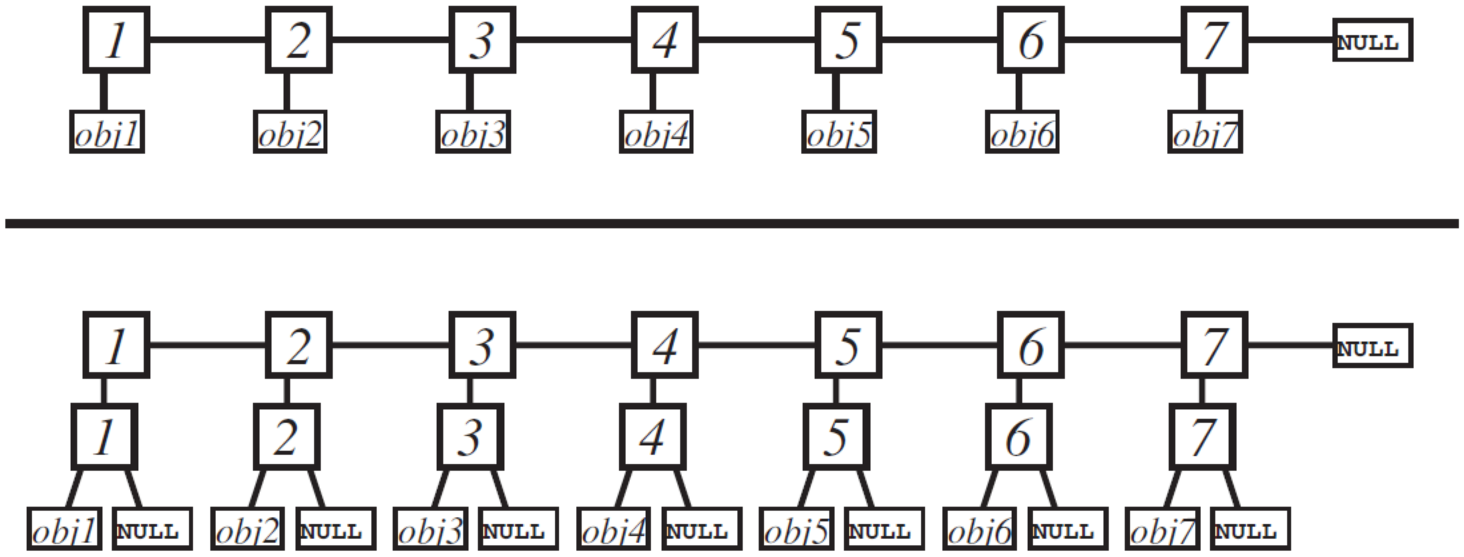
- Repeat the process until there is only one tree left.





Bottom Up

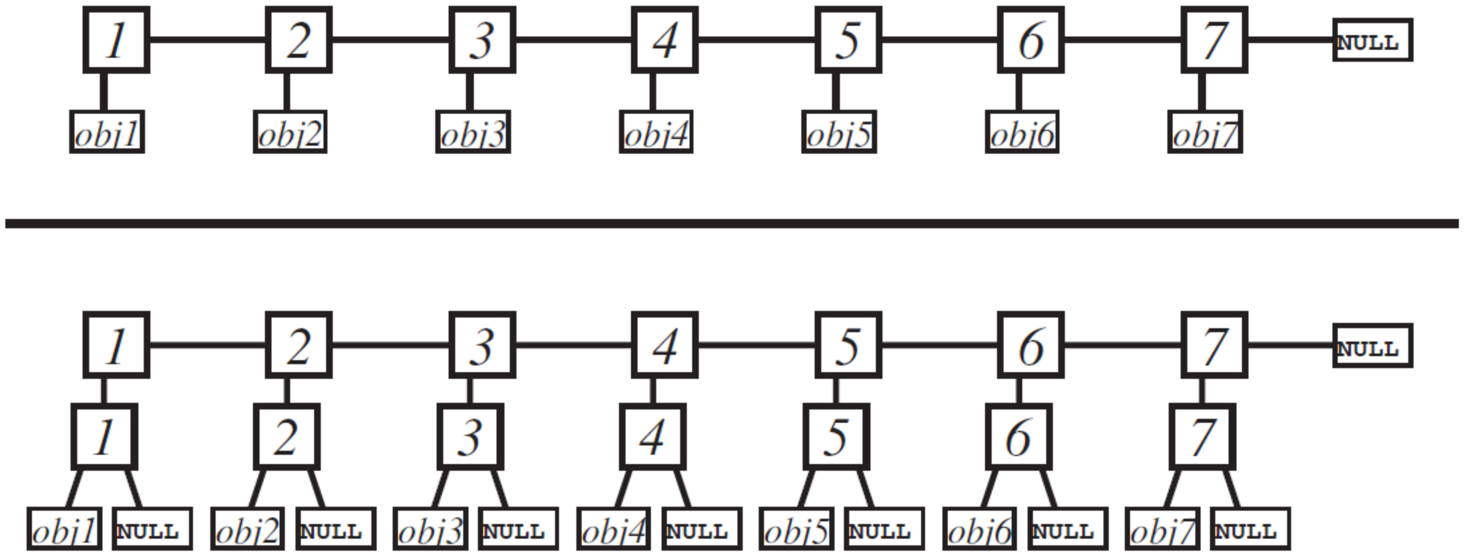
- If the list is empty, create an empty tree.
- Otherwise, create a list as a list of one-element trees (leaves).



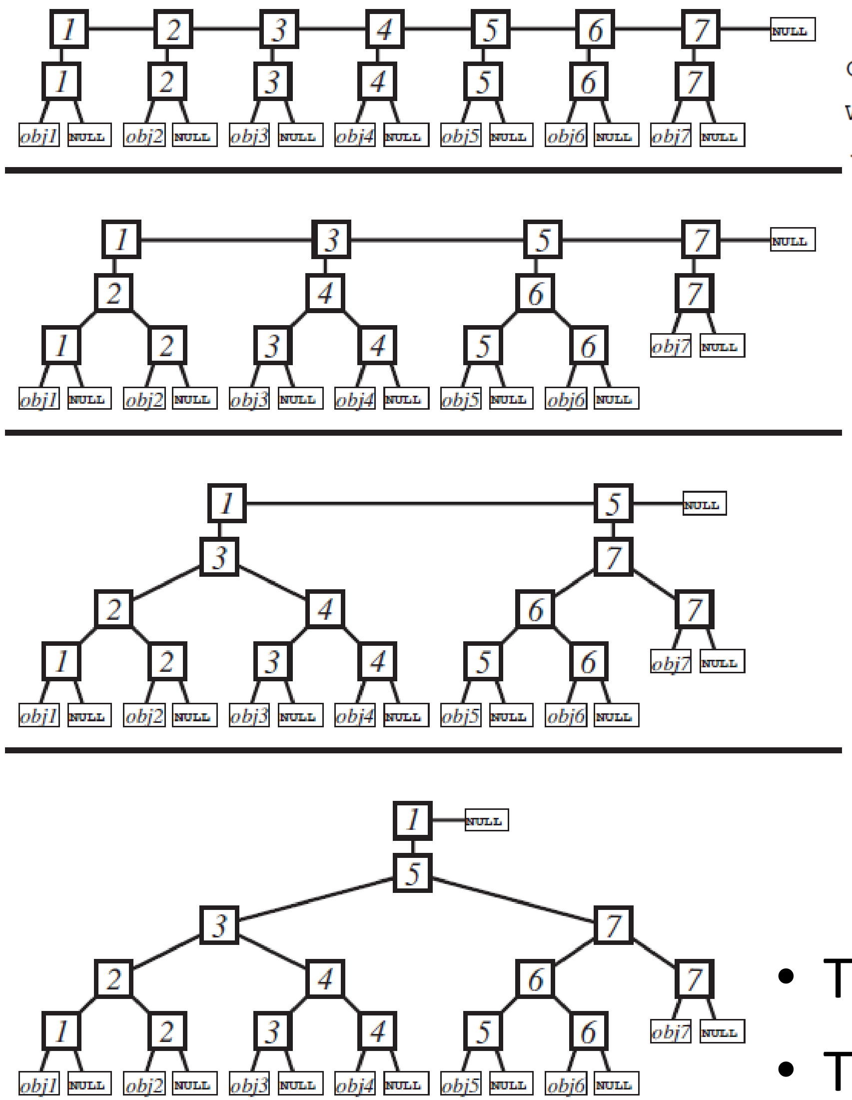

```
tree_node_t *make_tree(tree_node_t *list)
{  tree_node_t *end, *root;
   if( list == NULL )
   {  root = get_node(); /* create empty tree */
      root->left = root->right = NULL;
      return( root );
   }
   else if( list->right == NULL )
      return( list ); /* one-leaf tree */
   else /* nontrivial work required: at least
            two nodes */
   {  root = end = get_node();
      /* convert input list into leaves below
         new list */
      end->left = list;
      end->key = list->key;
      list = list->right;
      end->left->right = NULL;
      while( list != NULL )
      {  end->right = get_node();
         end = end->right;
         end->left = list;
         end->key = list->key;
         list = list->right;
         end->left->right = NULL;
      }
      end->right = NULL;
```

Bottom Up

- We create the upper part of the new list with pointer `end`, i.e. the nodes that point to the leaves.




```
tree_node_t *make_tree(tree_node_t *list)
{   tree_node_t *end, *root;
    if( list == NULL )
    {   root = get_node(); /* create empty tree */
        root->left = root->right = NULL;
        return( root );
    }
    else if( list->right == NULL )
        return( list ); /* one-leaf tree */
    else /* nontrivial work required: at least
                two nodes */
    {   root = end = get_node();
        /* convert input list into leaves below
           new list */
        end->left = list;
        end->key = list->key;
        list = list->right;
        end->left->right = NULL;
        while( list != NULL )
        {   end->right = get_node();
            end = end->right;
            end->left = list;
            end->key = list->key;
            list = list->right;
            end->left->right = NULL;
        }
        end->right = NULL;
```



```
{   tree_node_t *old_list, *new_list, *tmp1,
                *tmp2;
old_list = root;
while( old_list->right != NULL )
{   /* join first two trees from
       old_list */
    tmp1 = old_list;
    tmp2 = old_list->right;
    old_list = old_list->right->right;
    tmp2->right = tmp2->left;
    tmp2->left  = tmp1->left;
    tmp1->left  = tmp2;
    tmp1->right = NULL;
    new_list = end = tmp1;
    /* new_list started */
    while( old_list != NULL )
    /* not at end */
    {   if( old_list->right == NULL )
        /* last tree */
        {   end->right = old_list;
            old_list = NULL;
        }
        else /* join next two trees of
                old_list */
        {   tmp1 = old_list;
            tmp2 = old_list->right;
            old_list =
                  old_list-> right->right;
            tmp2->right = tmp2->left;
            tmp2->left  = tmp1->left;
            tmp1->left  = tmp2;
            tmp1->right = NULL;
            end->right  = tmp1;
            end =   end->right;
        }
    } /* finished one pass through
         old_list */
    old_list = new_list;
} /* end joining pairs of trees
     together */
root = old_list->left;
return_node( old_list );
}
return( root );
}
}
```

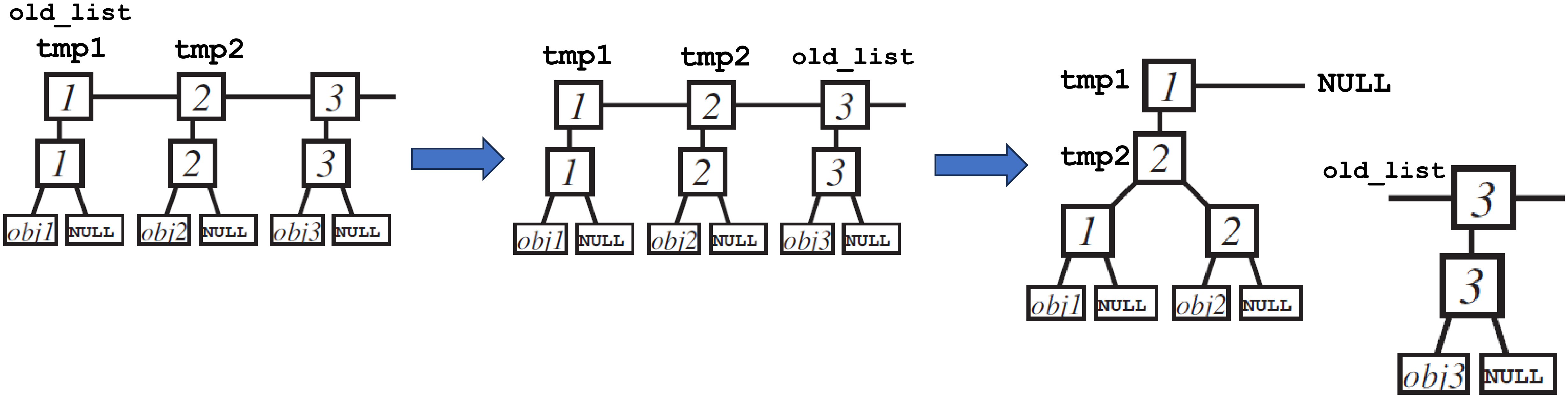
- The inner while goes through the list joining two consecutive trees.
- The outer while repeats the process until there is only one tree left.

- In each iteration of the outer while, `old_list` is used to construct a `new_list` that merges the consecutive pairs of trees of `old_list`.
- During the process, `new_list` points to what has been merged until `end`; `old_list` points to what is left to be merged.

Merging two consecutive trees

- In each iteration of the outer while, `old_list` is used to construct a `new_list` that merges the consecutive pairs of trees of `old_list`.
- During the process, `new_list` points to what has been merged until `end`; `old_list` points to what is left to be merged.
- The objects that are to be merged are `tmp1` and `tmp2`.

```
tmp1 = old_list;
tmp2 = old_list->right;
old_list = old_list->right->right;
tmp2->right = tmp2->left;
tmp2->left  = tmp1->left;
tmp1->left  = tmp2;
tmp1->right = NULL;
```



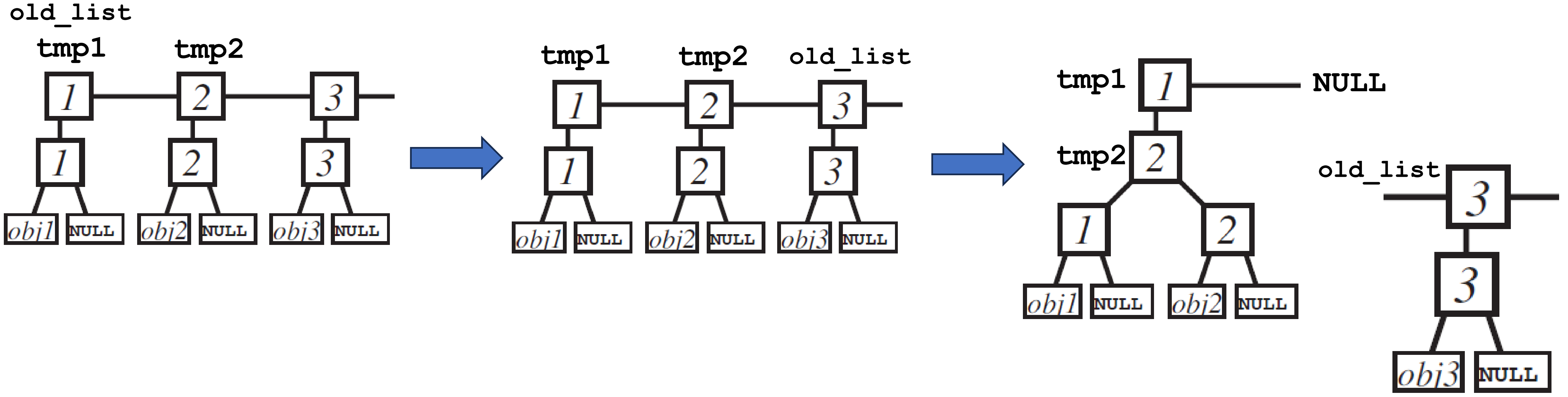
Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

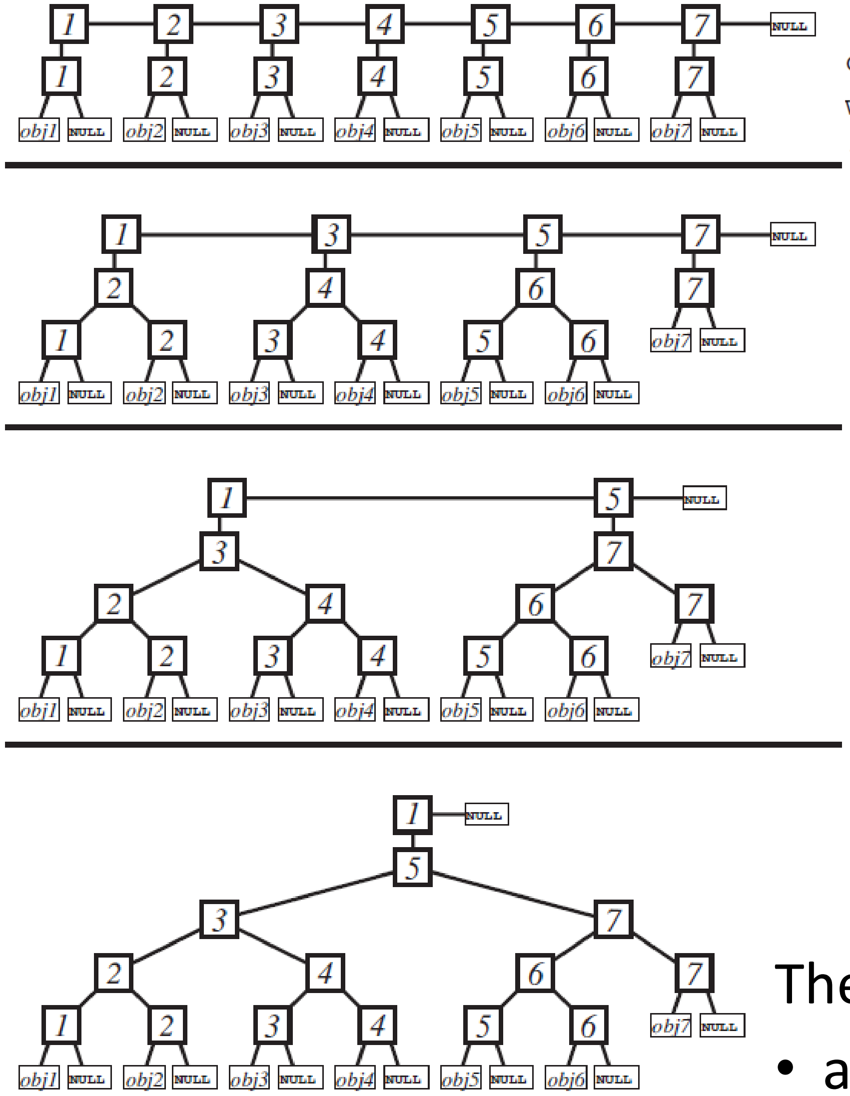
Merging two consecutive trees

This operation is performed:

- at the beginning of each iteration of the outer loop.
  - We keep the beginning of the list in `new_list`.
- when there are still trees to merge in the inner loop.
  - We append the merged nodes into `new_list` at the `end` and update `end`.

```
tmp1 = old_list;
tmp2 = old_list->right;
old_list = old_list->right->right;
tmp2->right = tmp2->left;
tmp2->left  = tmp1->left;
tmp1->left  = tmp2;
tmp1->right = NULL;
```

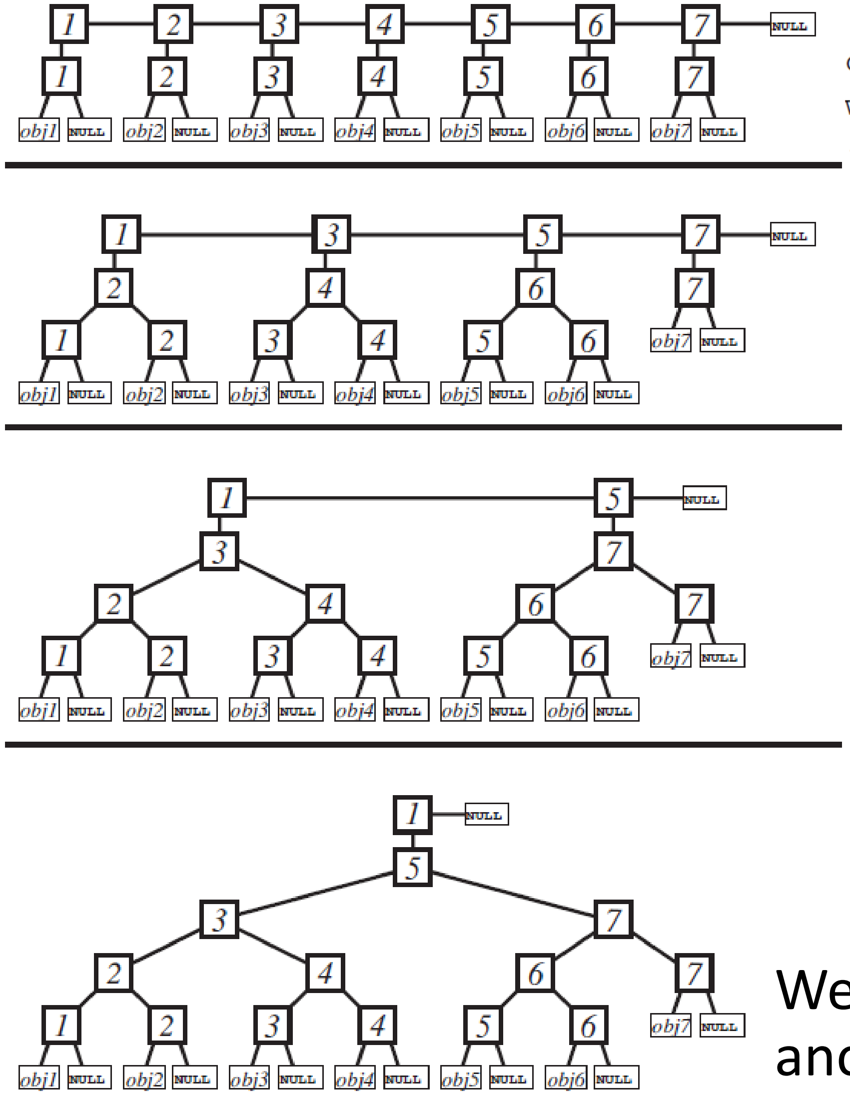





```
{  tree_node_t *old_list, *new_list, *tmp1,
               *tmp2;
   old_list = root;
   while( old_list->right != NULL )
   {  /* join first two trees from
         old_list */
      tmp1 = old_list;
      tmp2 = old_list->right;
      old_list = old_list->right->right;
      tmp2->right = tmp2->left;
      tmp2->left  = tmp1->left;
      tmp1->left  = tmp2;
      tmp1->right = NULL;
      new_list = end = tmp1;
      /* new_list started */
      while( old_list != NULL )
      /* not at end */
      {  if( old_list->right == NULL )
         /* last tree */
         {  end->right = old_list;
            old_list = NULL;
         }
         else /* join next two trees of
                 old_list */
         {  tmp1 = old_list;
            tmp2 = old_list->right;
            old_list =
                  old_list-> right->right;
            tmp2->right = tmp2->left;
            tmp2->left  = tmp1->left;
            tmp1->left  = tmp2;
            tmp1->right = NULL;
            end->right  = tmp1;
            end =   end->right;
         }
      } /* finished one pass through
           old_list */
      old_list = new_list;
   } /* end joining pairs of trees
        together */
   root = old_list->left;
   return_node( old_list );
}
return( root );
}
}
```

The merge operation is performed:

- at the beginning of each iteration of the outer loop.
  - We keep the beginning of the list in `new_list`.
- when there are still trees to merge in the inner loop.
  - We append the merged nodes into `new_list` at the `end` and update `end`.




```
{  tree_node_t *old_list, *new_list, *tmp1,
               *tmp2;
   old_list = root;
   while( old_list->right != NULL )
   {  /* join first two trees from
         old_list */
      tmp1 = old_list;
      tmp2 = old_list->right;
      old_list = old_list->right->right;
      tmp2->right = tmp2->left;
      tmp2->left  = tmp1->left;
      tmp1->left  = tmp2;
      tmp1->right = NULL;
      new_list = end = tmp1;
      /* new_list started */
      while( old_list != NULL )
      /* not at end */
      {  if( old_list->right == NULL )
         /* last tree */
         {  end->right = old_list;
            old_list = NULL;
         }
         else /* join next two trees of
                 old_list */
         {  tmp1 = old_list;
            tmp2 = old_list->right;
            old_list =
                  old_list-> right->right;
            tmp2->right = tmp2->left;
            tmp2->left  = tmp1->left;
            tmp1->left  = tmp2;
            tmp1->right = NULL;
            end->right  = tmp1;
            end =   end->right;
         }
      } /* finished one pass through
           old_list */
      old_list = new_list;
   } /* end joining pairs of trees
        together */
   root = old_list->left;
   return_node( old_list );
}
return( root );
}
}
```

We have a special case when there is an element that does not have another node to be merged with.

- We just add it as the final node of `new_list`.

- At the end of an iteration of the outer loop, we just set the new list as the old one.
- When the process is finished, we return the resulting tree and free the list that points to it.

Complexity Analysis of the Bottom Up Construction

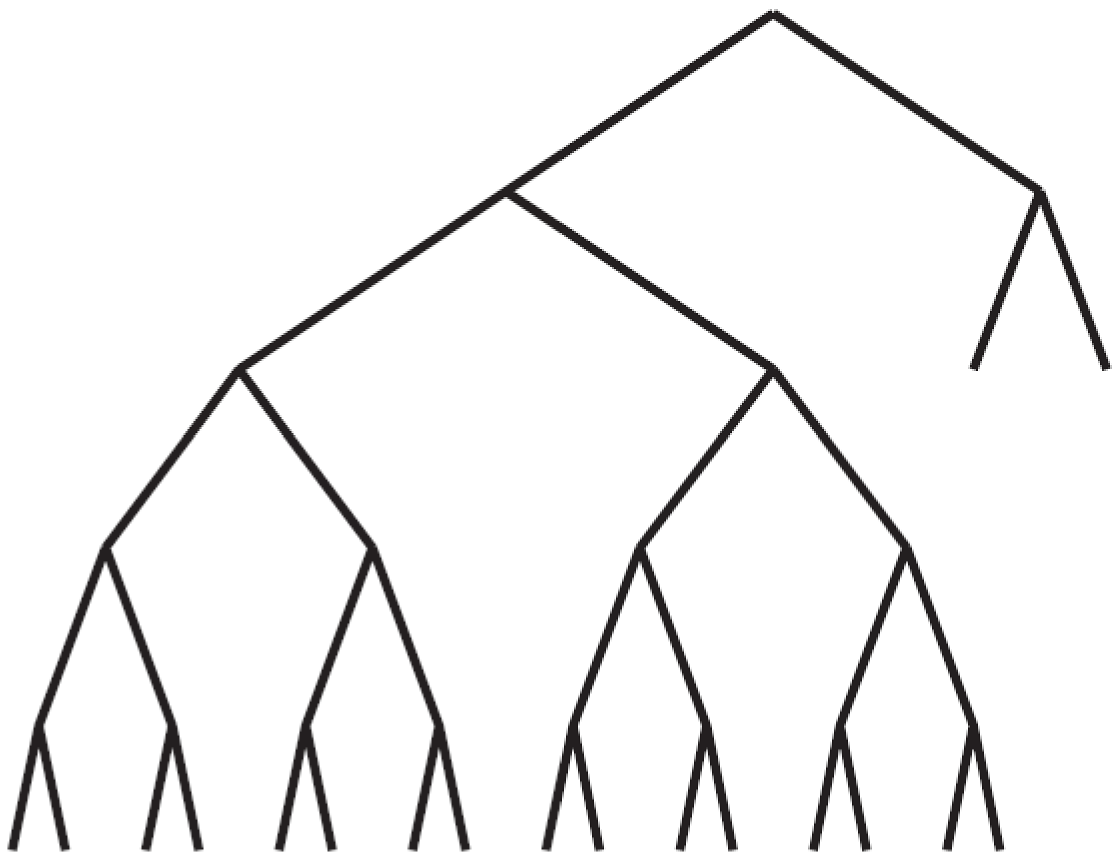
- This method constructs a search tree of optimal height from an ordered list in time $O(n)$.

- **First Part:** Duplicating the list and converting it in a list of leaves takes $O(n)$.
- **Second Part:** In each iteration of the innermost while, one of the n interior nodes created in the first part if removed from the current list and put into a finished subtree. Thus, it is executed only $O(n)$ times.

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Analysis of Bottom Up Construction

- This method produces a tree of optimal height.
- The disadvantage is that the resulting tree might be quite unbalanced.
- If we start with a set of $2^m + 1$ items, then the root of the tree has
  - A subtree of 2^m elements.
  - A subtree of 1 item.



Top Down Construction

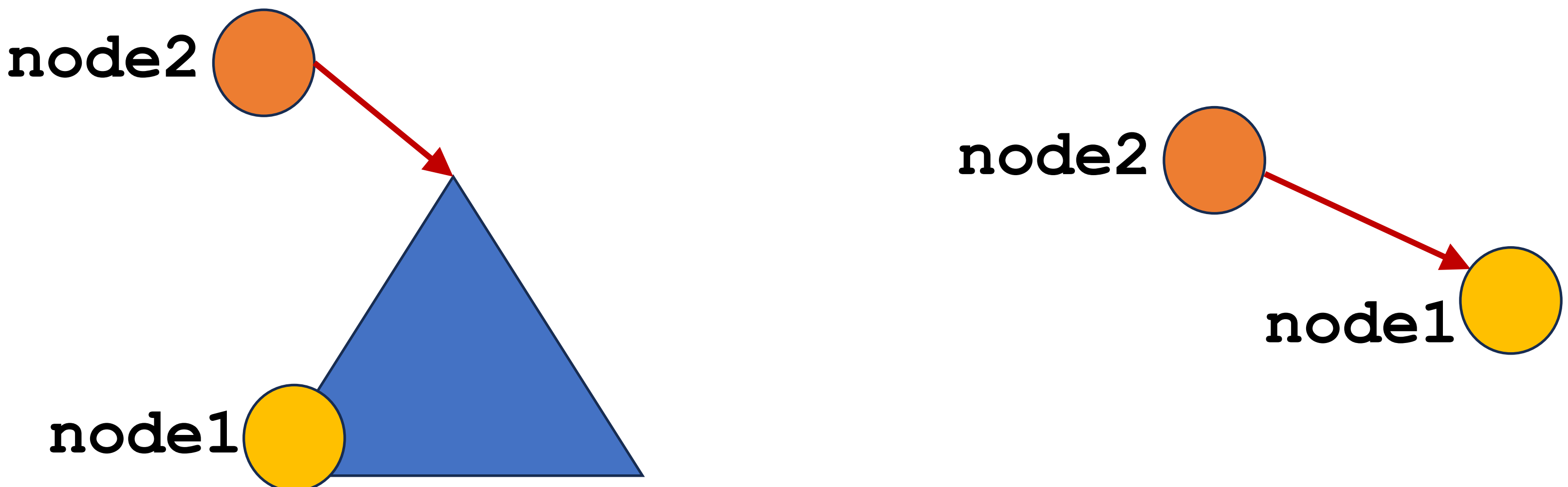
- Recursive description
  - Divide the data set in the middle.
  - Create optimal trees for the lower and the upper halves.
  - Join them together.
- This division is balanced: the number of items in the left and in the right subtree differs by at most one.
- It results in a tree of optimal height.
- But if we implement it this way, the complexity is $\Theta(n \lg n)$ due to the $O(n)$ overhead in each recursion to find the middle of the list.
- But we can implement it in $O(n)$ using a stack. The best option is an array-based stack.

Top Down Construction

- We first construct the tree “in the abstract” without filling any key values or pointers to objects.
- We don’t need to find the middle of the list; we just need to keep track of the number of elements that should go into the left and right subtrees.
- We can build this abstract tree of the required shape easily using a stack.
- We initially put the root on the stack labeled with the required tree size.
- Then, we continue until the stack is empty, to
  - Pop a node from the stack.
    - If its size is 1 (it’s a leaf),it should be filled with the next key and object from the list.
    - Otherwise,
      - Attach it to two newly created nodes labeled with half the size.
      - Put the new nodes on the stack.

Top Down Construction

- The problem is to fill in the keys of the interior nodes, which become available only when a leaf is reached.
- For this, each item on the stack needs two pointers:
  - `node1`: The node that still needs to be expanded. The one we are considering.
  - `node2`: The node higher up in the tree, where the smallest key of leaves below that node's right subtree should be inserted as comparison key.



Top Down Construction

- The problem is to fill in the keys of the interior nodes, which become available only when a leaf is reached.
- For this, each item on the stack needs two pointers:
  - `node1`: The node that still needs to be expanded. The one we are considering.
  - `node2`: The node higher up in the tree, where the smallest key of leaves below that node's right subtree should be inserted as comparison key.
- Also, each stack item contains a `number`: the number of leaves that should be created below that node.
- When we pop a node from the stack and create its two children, we first insert the right child first.
- Then, when we reach a leaf, it is the leftmost unfinished leaf of the tree.

Top Down Construction

- The problem is to fill in the keys of the interior nodes, which become available only when a leaf is reached.
- For this, each item on the stack needs two pointers:
  - `node1`: The node that still needs to be expanded. The one we are considering.
  - `node2`: The node higher up in the tree, where the smallest key of leaves below that node's right subtree should be inserted as comparison key.
- The pointer for the missing key value propagates into the left subtree of the current node (where the smallest node comes from).
- The smallest key from the right subtree should become the comparison key of the current node.

Top Down Construction

- The entries for a stack item are:
  - `node1`: The node that still needs to be expanded. The one we are considering.
  - `node2`: Pointer to the node higher up in the tree, where the smallest key of leaves below that node's right subtree should be inserted as comparison key.
  - `number`: the number of leaves that should be created below that node.
- First, we count the number of nodes.
- We create the root and its corresponding stack item. Then, we push it into the stack.

```
tree_node_t *make_tree(tree_node_t *list)
{  typedef struct { tree_node_t  *node1;
                   tree_node_t  *node2;
                   int          number; }
                   st_item;
   st_item current, left, right;
   tree_node_t *tmp, *root;
   int length = 0;
   for( tmp = list; tmp != NULL;
   tmp = tmp->right )
      length += 1; /* find length of list */
   create_stack(); /* stack of st_item:
                      replace by array */
   root = get_node();
   /* put root node on stack */
   current.node1 = root;
   current.node2 = NULL;
   /* root expands to length leaves */
   current.number = length;
   push( current );
```

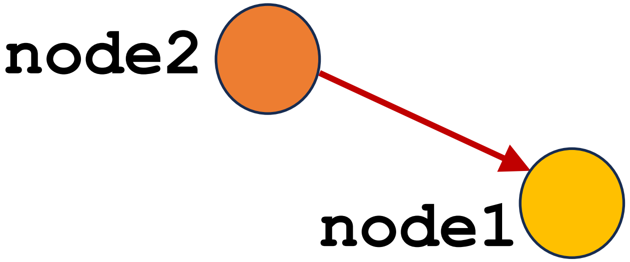
Top Down Construction

- When we pop a stack item `current` and it does not correspond to a leaf, we create its children `left` and `right` with their corresponding sizes.
- `node1` corresponds to each created child that may need to be expanded.
- `node2` is set to:
  - `current.node2` (for `left`)
  - `current` (for `right`)
- We set the pointers between `current` and `left` and `right`.
- Then, we push right and left into the stack.

```
while( !stack_empty() )
/* there is still unexpanded node */
{   current = pop();
    if( current.number > 1 )
    /* create (empty) tree nodes */
    { left.node1 = get_node();
      left.node2 = current.node2;
      left.number = current.number / 2;
      right.node1 = get_node();
      right.node2 = current.node1;
      right.number = current.number -
                     left.number;
      (current.node1)->left  = left.node1;
      (current.node1)->right = right.node1;
      push( right );
      push( left );
    }
    else /* reached a leaf, must be filled
            with list item */
```

Top Down Construction

- `node2` is set to:
  - `current.node2` (for `left`)
  - **`current` (for `right`)**
- Note that for the right child, `node2` points to its parent `current`.
- <u>Notice that `current` is the closest ancestor that is to the left of `right`.</u>
- Thus, if `right` is a leaf, its key should be the comparison value of its parent `current`.

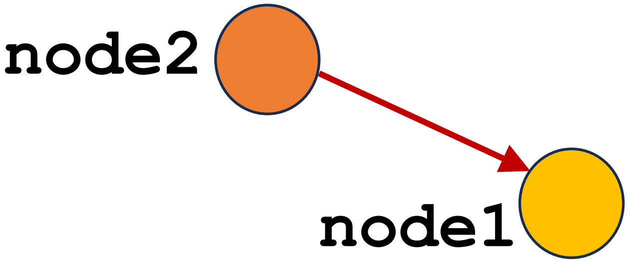



```
while( !stack_empty() )
/* there is still unexpanded node */
{   current = pop();
    if( current.number > 1 )
    /* create (empty) tree nodes */
    { left.node1 = get_node();
      left.node2 = current.node2;
      left.number = current.number / 2;
      right.node1 = get_node();
      right.node2 = current.node1;
      right.number = current.number -
                     left.number;
      (current.node1)->left  = left.node1;
      (current.node1)->right = right.node1;
      push( right );
      push( left );
    }
    else /* reached a leaf, must be filled
            with list item */
```


Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Top Down Construction

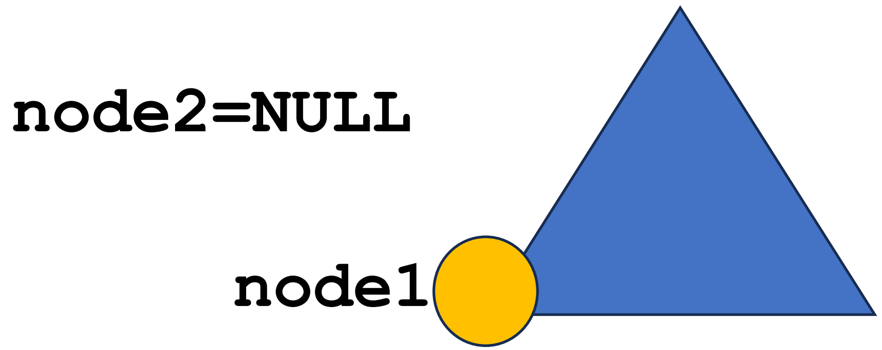
- `node2` is set to:
  - `current.node2` (for `left`)
  - **`current` (for `right`)**
- <u>All the internal nodes whose right children are leaves are getting a comparison value.</u>




```
while( !stack_empty() )
/* there is still unexpanded node */
{   current = pop();
    if( current.number > 1 )
    /* create (empty) tree nodes */
    { left.node1 = get_node();
      left.node2 = current.node2;
      left.number = current.number / 2;
      right.node1 = get_node();
      right.node2 = current.node1;
      right.number = current.number -
                     left.number;
      (current.node1)->left  = left.node1;
      (current.node1)->right = right.node1;
      push( right );
      push( left );
    }
    else /* reached a leaf, must be filled
            with list item */
```

Top Down Construction

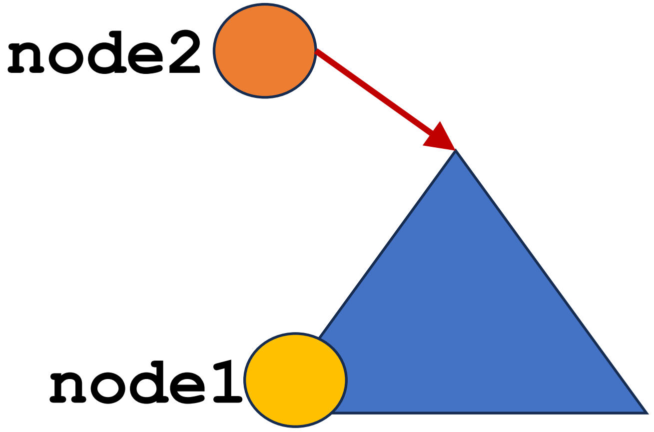
- `node2` is set to:
  - **`current.node2` (for `left`)**
  - `current` (for `right`)
- Note that for the left child, `node2` may point to either NULL or a non immediate ancestor.
- If it points to NULL it is because `left` is the leftmost node so far.




```
while( !stack_empty() )
/* there is still unexpanded node */
{   current = pop();
    if( current.number > 1 )
    /* create (empty) tree nodes */
    { left.node1 = get_node();
      left.node2 = current.node2;
      left.number = current.number / 2;
      right.node1 = get_node();
      right.node2 = current.node1;
      right.number = current.number -
                     left.number;
      (current.node1)->left  = left.node1;
      (current.node1)->right = right.node1;
      push( right );
      push( left );
    }
    else /* reached a leaf, must be filled
            with list item */
```

Top Down Construction

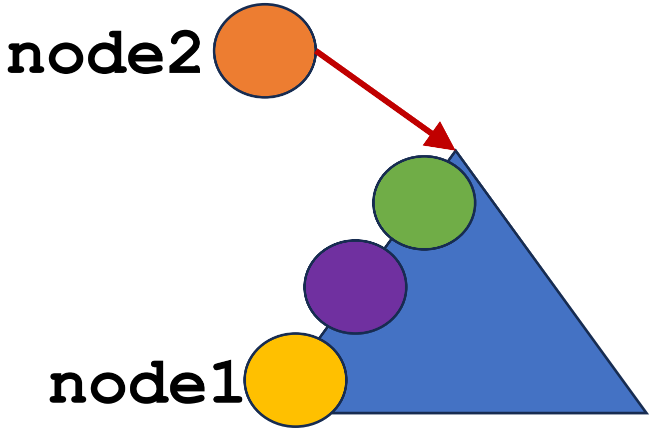
- `node2` is set to:
  - **`current.node2`** **(for `left`)**
  - `current` (for `right`)
- Otherwise, it points to the closest ancestor that is to the left of `left`.
- Thus, if `left` is a leaf, its key should be the comparison value of `current.node2`.




```
while( !stack_empty() )
/* there is still unexpanded node */
{   current = pop();
    if( current.number > 1 )
    /* create (empty) tree nodes */
    { left.node1 = get_node();
      left.node2 = current.node2;
      left.number = current.number / 2;
      right.node1 = get_node();
      right.node2 = current.node1;
      right.number = current.number -
                     left.number;
      (current.node1)->left  = left.node1;
      (current.node1)->right = right.node1;
      push( right );
      push( left );
    }
    else /* reached a leaf, must be filled
            with list item */
```

Top Down Construction

- `node2` is set to:
  - **`current.node2` (for `left`)**
  - `current` (for `right`)
- Otherwise, it points to the closest ancestor that is to the left of `left`.
- Note that `node2` was propagated to the left until it reaches a leaf.




```
while( !stack_empty() )
/* there is still unexpanded node */
{   current = pop();
    if( current.number > 1 )
    /* create (empty) tree nodes */
    { left.node1 = get_node();
      left.node2 = current.node2;
      left.number = current.number / 2;
      right.node1 = get_node();
      right.node2 = current.node1;
      right.number = current.number -
                     left.number;
      (current.node1)->left  = left.node1;
      (current.node1)->right = right.node1;
      push( right );
      push( left );
    }
    else /* reached a leaf, must be filled
            with list item */
```

Top Down Construction

- `node2` is set to:
  - **`current.node2` (for `left`)**
  - `current` (for `right`)
- Note that for the left child, `node2` may point to a non immediate ancestor, i.e. the closest ancestor that is to the left of `left`.
- Thus, if `left` is a leaf, its key should be the comparison value of `current.node2`.
- <u>All the internal nodes are getting a comparison value because there is always a leaf for which it is the closest ancestor to the left of it.</u>

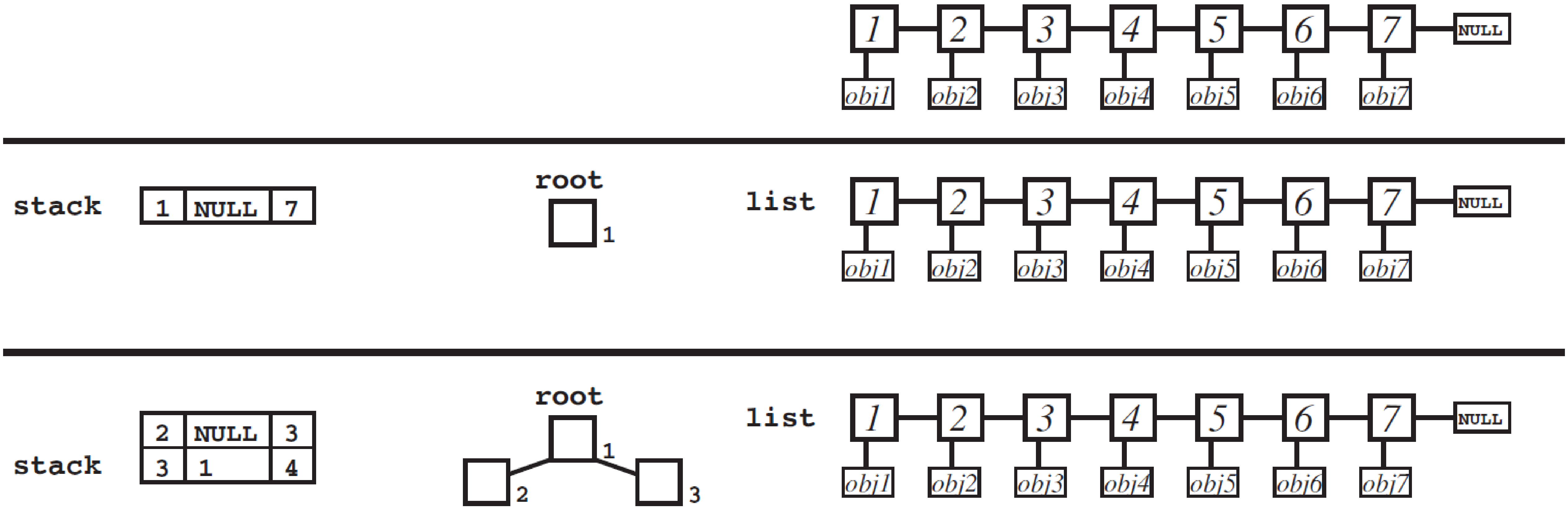
```
while( !stack_empty() )
/* there is still unexpanded node */
{   current = pop();
    if( current.number > 1 )
    /* create (empty) tree nodes */
    { left.node1 = get_node();
      left.node2 = current.node2;
      left.number = current.number / 2;
      right.node1 = get_node();
      right.node2 = current.node1;
      right.number = current.number -
                     left.number;
      (current.node1)->left  = left.node1;
      (current.node1)->right = right.node1;
      push( right );
      push( left );
    }
    else /* reached a leaf, must be filled
            with list item */
```

Top Down Construction

- When we pop a stack item `current` and it corresponds to a leaf, we fill the corresponding information.
- If `current.node2` is not NULL, we fill the comparison value of such node.
- We remove the inserted element from the list and move on to the next element.
- After the loop is completed, the root of the tree is returned.

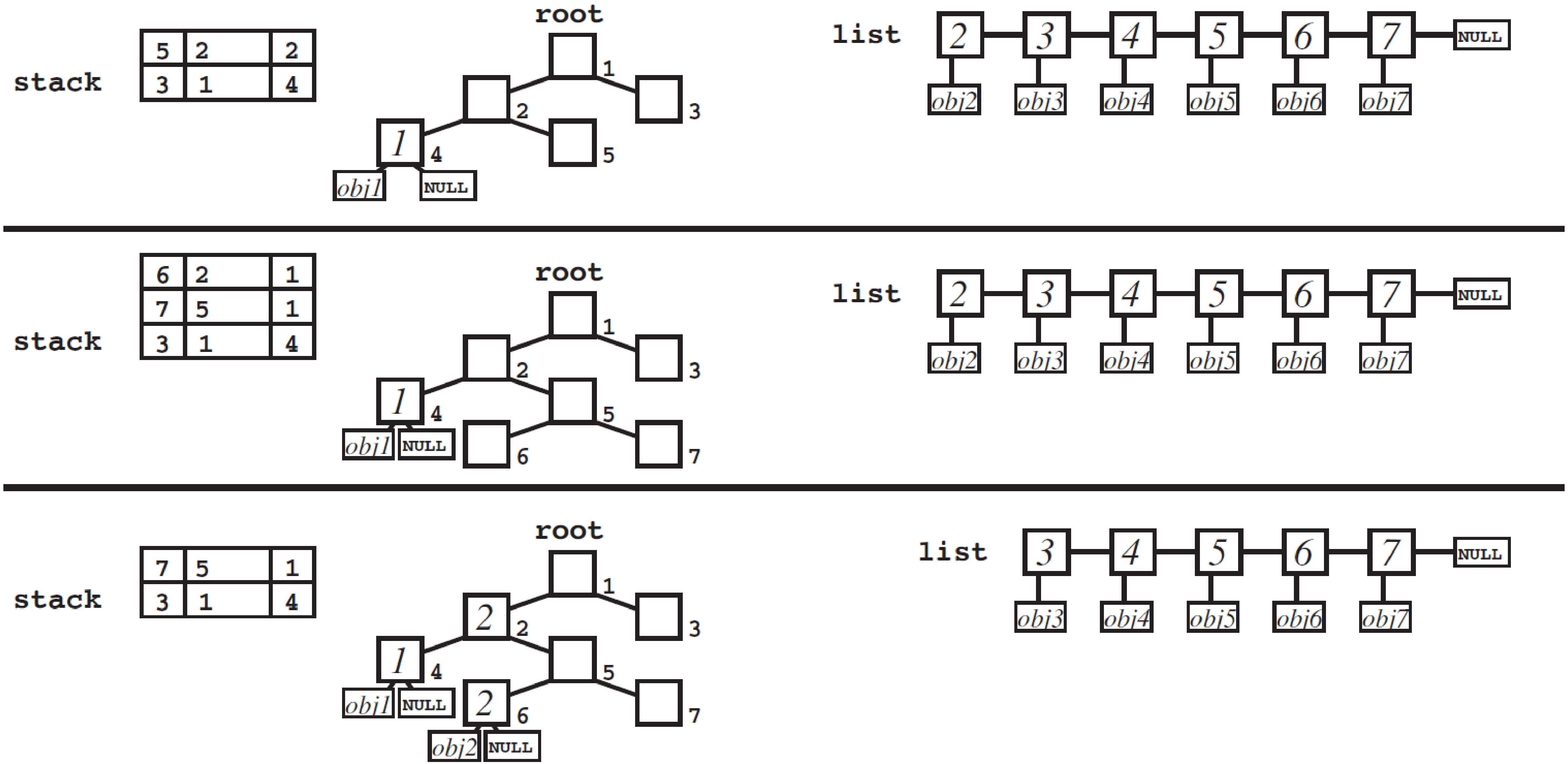
```
        else /* reached a leaf, must be filled
                with list item */
        { (current.node1)->left  = list->left;
          {/* fill leaf from list */}
          (current.node1)->key   = list->key;
          (current.node1)->right = NULL;
          if( current.node2 != NULL )
             /* insert comparison key in
                interior node */
             (current.node2)->key   = list->key;
          tmp = list;
          /* unlink first item from list */
          list = list->right;
          /* content has been copied to */
          return_node(tmp);
          /* leaf, so node is returned */
        }
    }
    return( root );
}
```

Top Down Construction

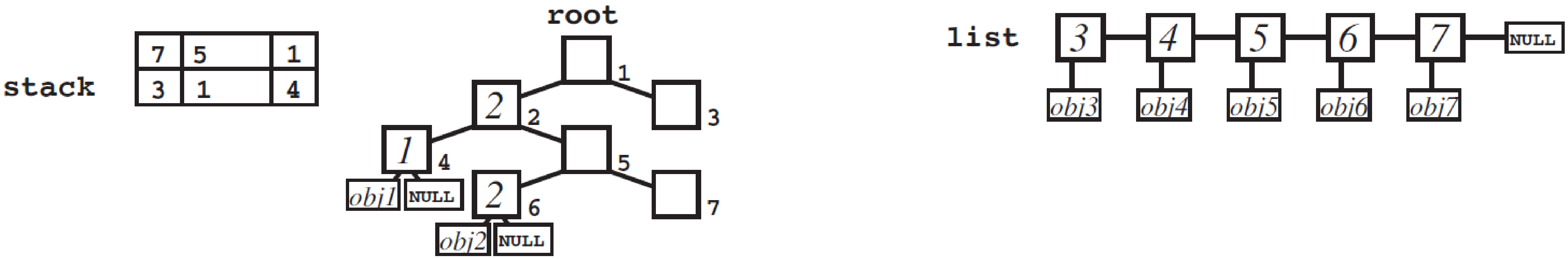


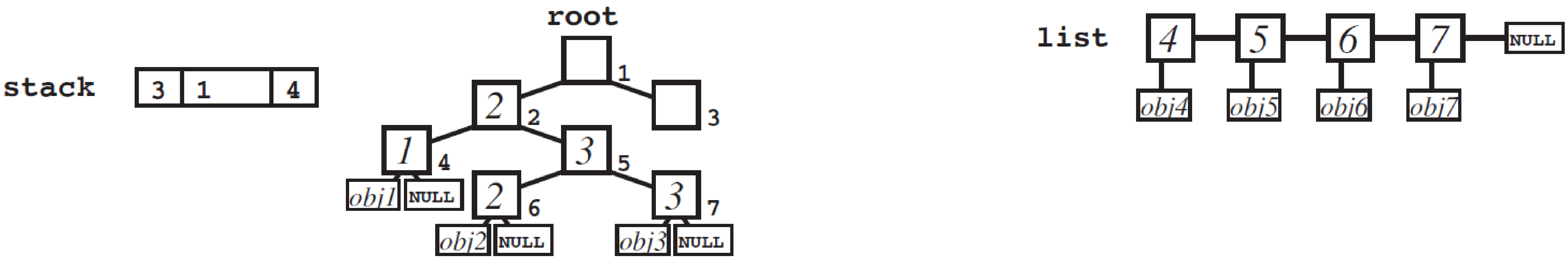
Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Top Down Construction



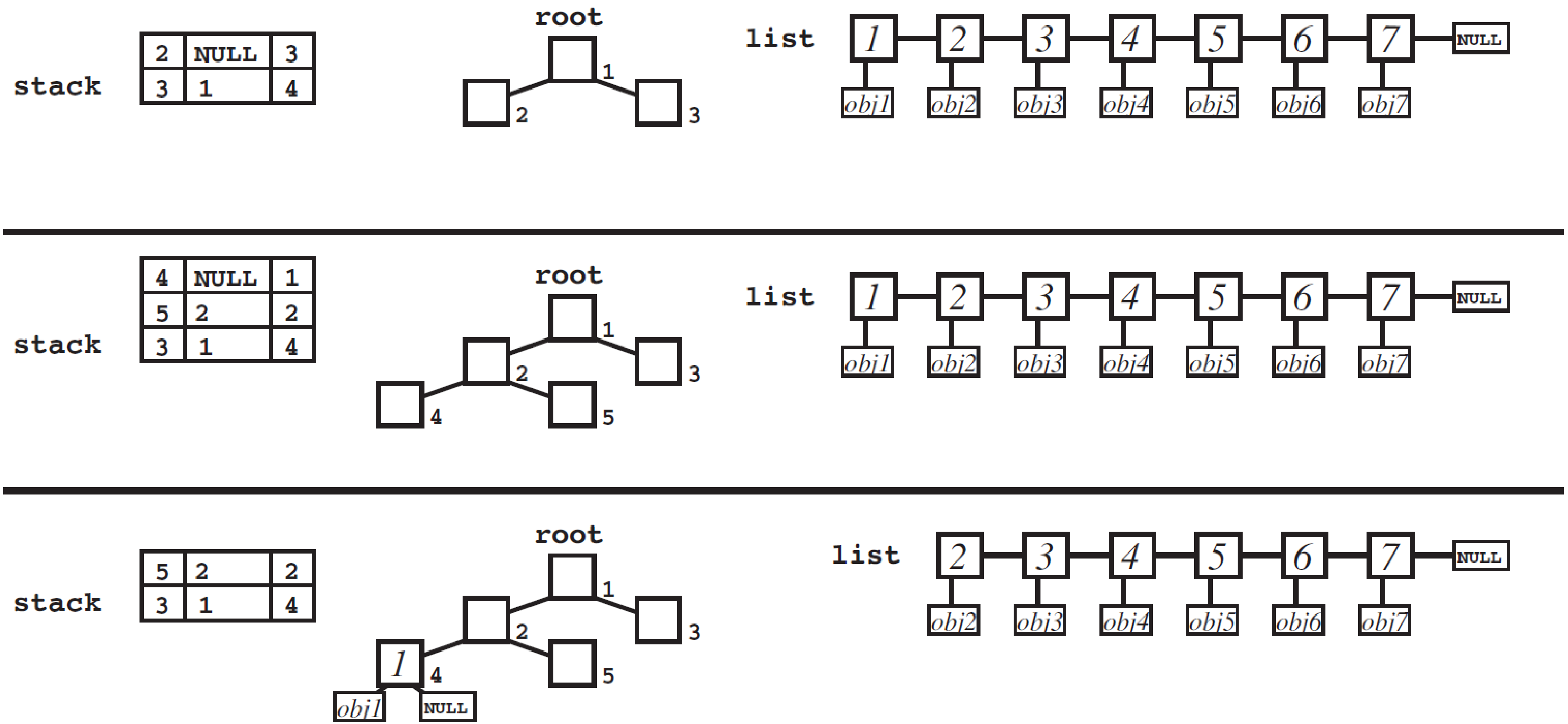
Top Down Construction





Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Top Down Construction



Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Complexity Analysis of the Top Down Construction

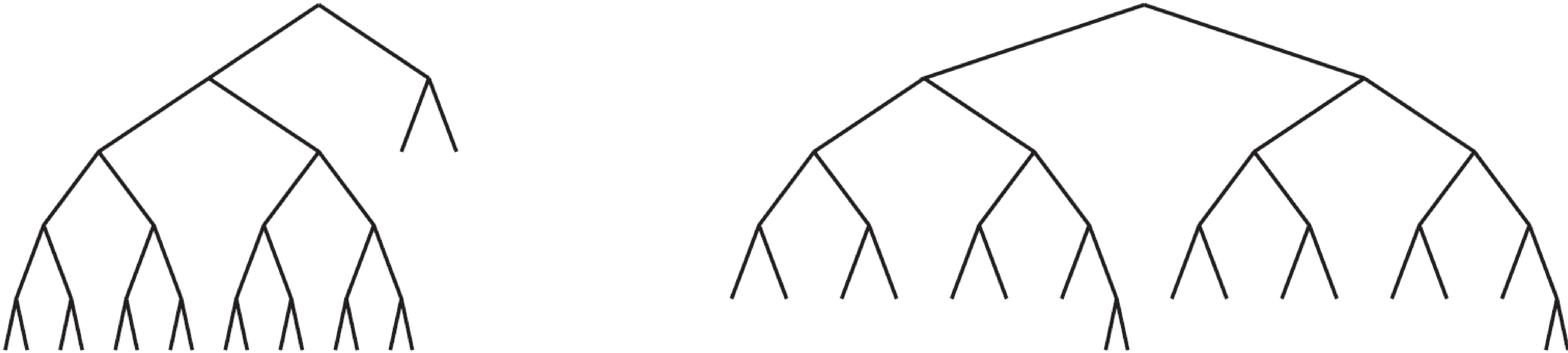
- This method constructs a search tree of optimal height form an ordered list in $O(n)$ time.

- Observe that in each step on the stack we either
  - create two new nodes, and there are only $n-1$ nodes created in total, or
  - we attach a list item as a leaf, and there are only n list items.

- There are other methods to construct the top-down optimal tree.
- They differ only in the amount of additional space needed.
- In this algorithm, it is the size of the stack: $\lceil \lg n \rceil$.

Analysis of the Top Down Construction

- It is more complicated of the bottom up construction, but it constructs a more balanced tree.



BOTTOM-UP AND TOP-DOWN OPTIMAL TREE WITH 18 LEAVES

8. CONVERTING A TREE INTO A SORTED LIST

Converting a Tree into a Sorted List

- We can use a stack for a trivial depth-first search enumeration of the leaves in decreasing order.
- This is because the right children will be popped first.
- We insert each node in the front of the list.
- Thus, we obtain an increasing order list of elements.
- The size of the stack is the height of the tree.
- It takes $O(n)$ time.

```
tree_node_t *make_list(tree_node_t *tree)
{  tree_node_t *list, *node;
   if( tree->left == NULL )
   { return_node( tree );
     return( NULL );
   }
   else
   {  create_stack();
      push( tree );
      list = NULL;
      while( !stack_empty() )
      {  node = pop();
         if( node->right == NULL )
         { node->right = list;
           list = node;
         }
         else
         { push( node->left );
           push( node->right );
           return_node( node );
         }
      }
      return( list );
   }
}
```

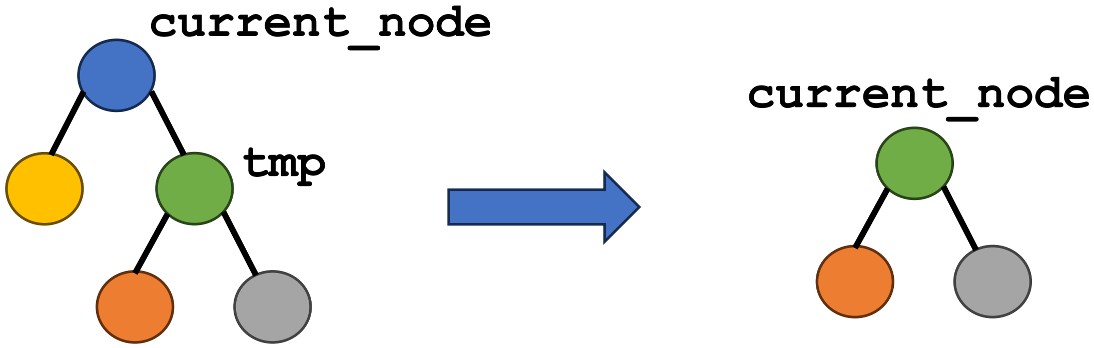
9. REMOVING A TREE

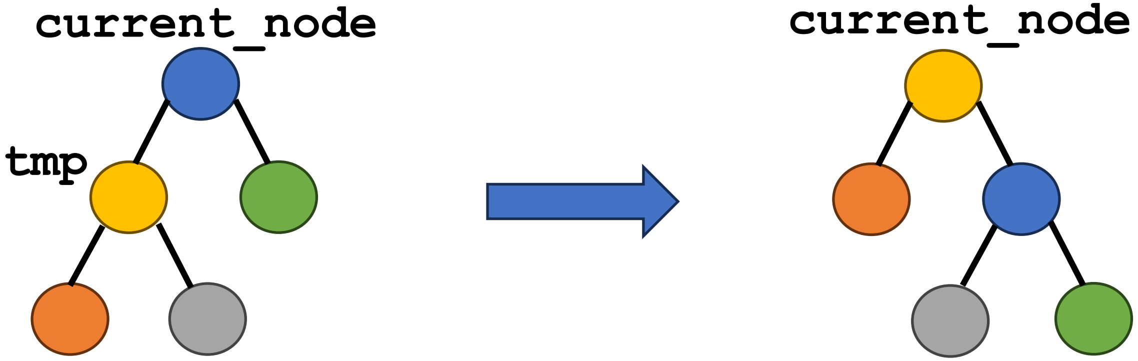
Removing a Tree

- It is important to free all nodes to avoid a memory leak.
- We can do it in $O(n)$ time, i.e. constant time per freed node.
- We can do it using a stack like in the previous algorithm.
- Alternatively, we can perform right rotations in the root until the left-lower neighbor is a leaf.
- Then, it returns the leaf, moves the root down to the right and return the previous root.

```
void remove_tree(tree_node_t *tree)
{   tree_node_t *current_node, *tmp;
    if( tree->left == NULL )
       return_node( tree );
    else
    {   current_node = tree;
       while(current_node->right != NULL )
       {   if( current_node->left->right == NULL )
           {   return_node( current_node->left );
               tmp = current_node->right;
               return_node( current_node );
               current_node = tmp;
           }
           else
           {   tmp = current_node->left;
               current_node->left = tmp->right;
               tmp->right = current_node;
               current_node = tmp;
           }
       }
       return_node( current_node );
    }
}
```

Removing a Tree






```
void remove_tree(tree_node_t *tree)
{   tree_node_t *current_node, *tmp;
    if( tree->left == NULL )
       return_node( tree );
    else
    {  current_node = tree;
       while(current_node->right != NULL )
       {  if( current_node->left->right == NULL )
          {  return_node( current_node->left );
             tmp = current_node->right;
             return_node( current_node );
             current_node = tmp;
          }
          else
          {  tmp = current_node->left;
             current_node->left = tmp->right;
             tmp->right = current_node;
             current_node = tmp;
          }
       }
       return_node( current_node );
    }
}
```

BIBLIOGRAPHY

- Peter Brass. Advanced Data Structures. Cambridge University Press. 2008.