Trees

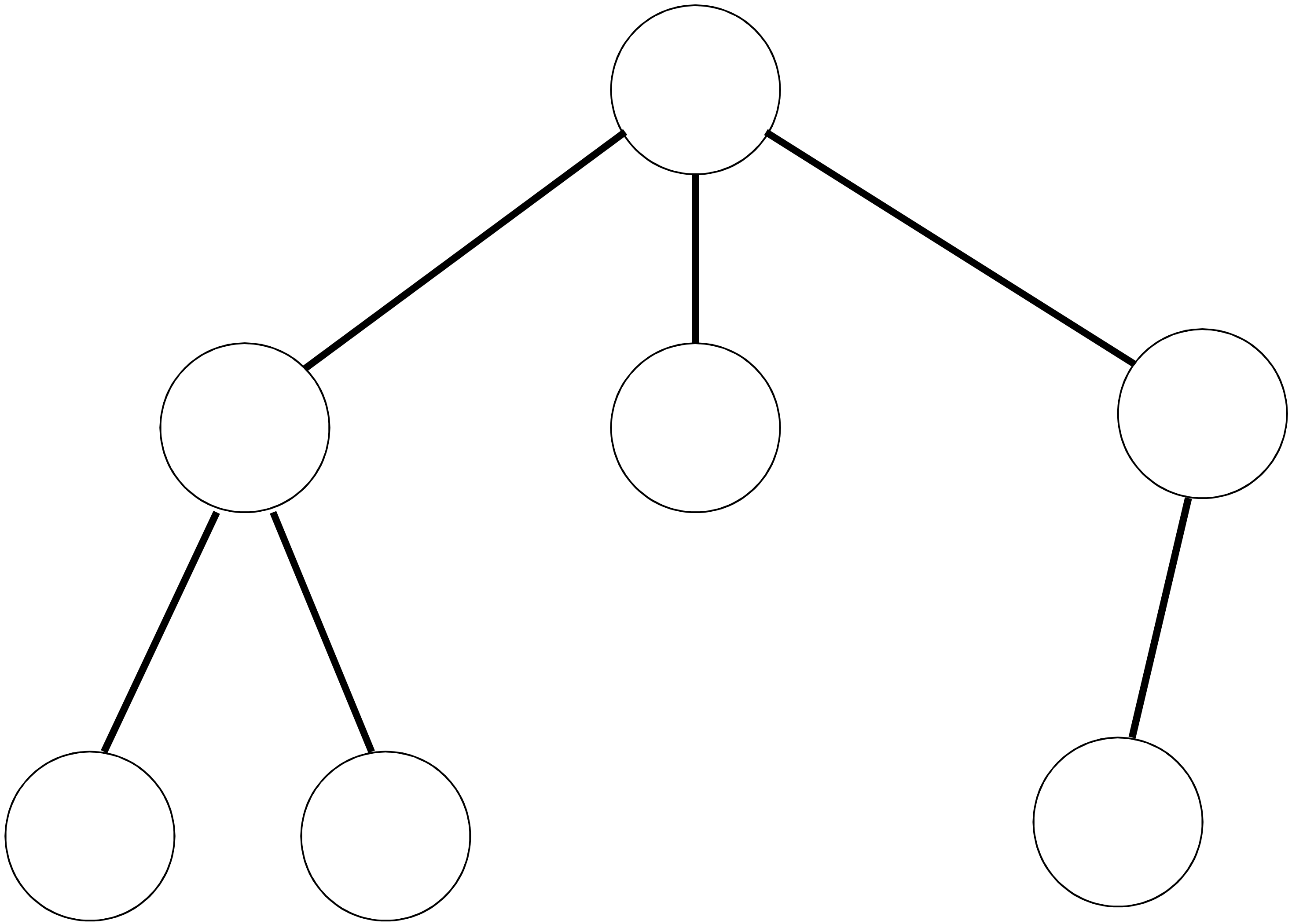
Introduction

Trees

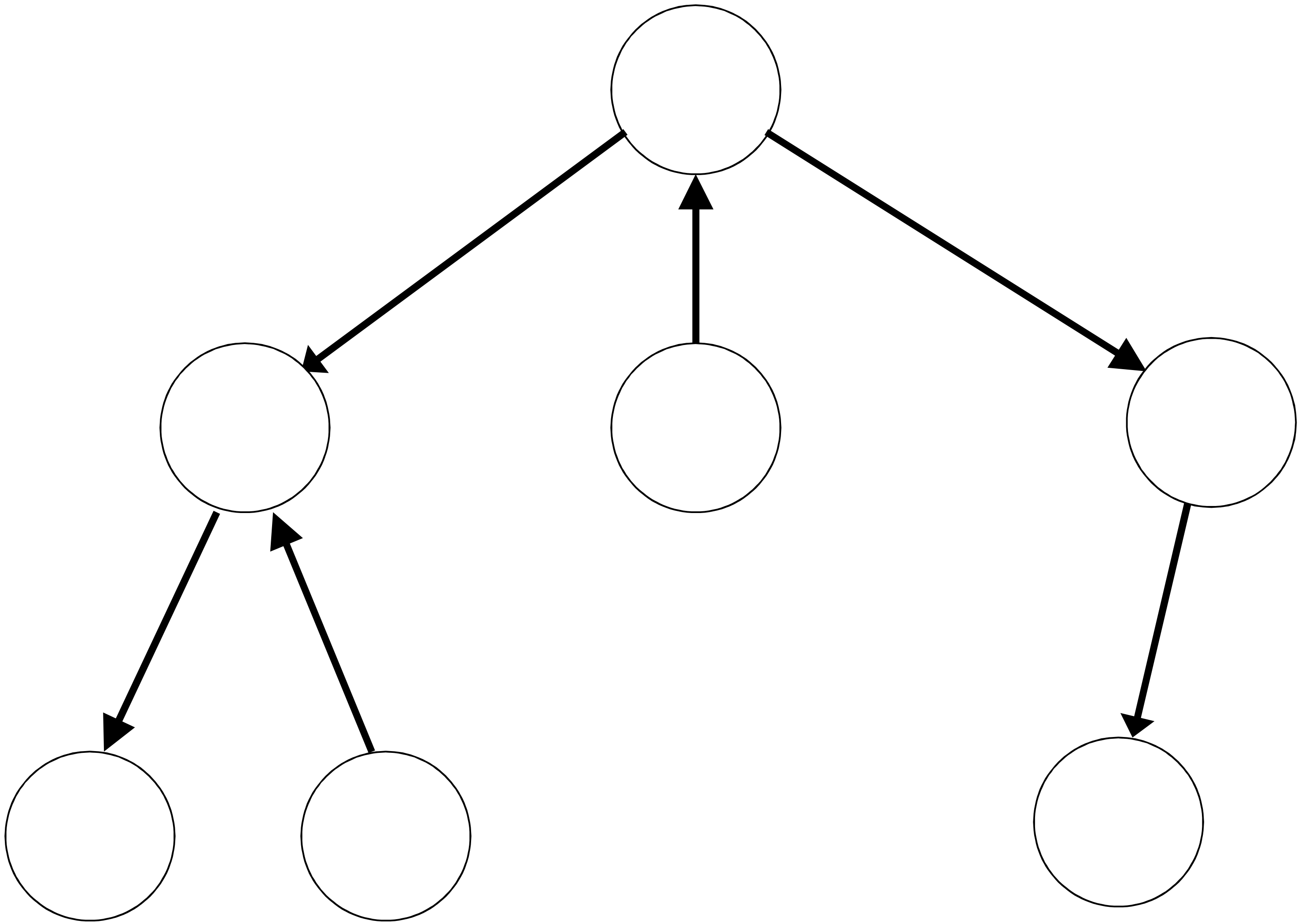
Tree:

- connected graph without cycles (acyclic)
- usage:
  - computer graphic
  - effective searching
  - computational trees
  - decision trees

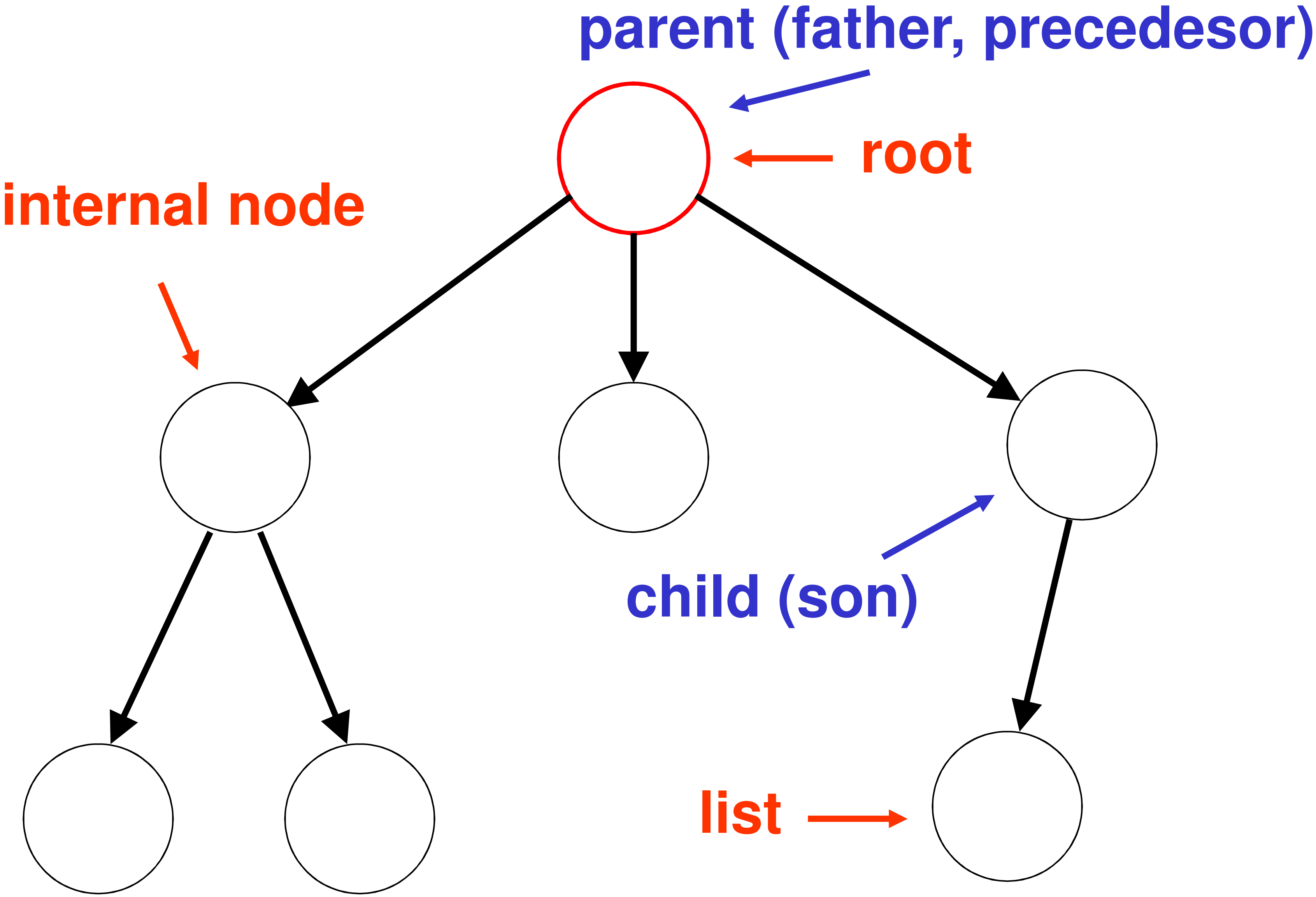
Undirected tree



Directed tree



Rooted tree

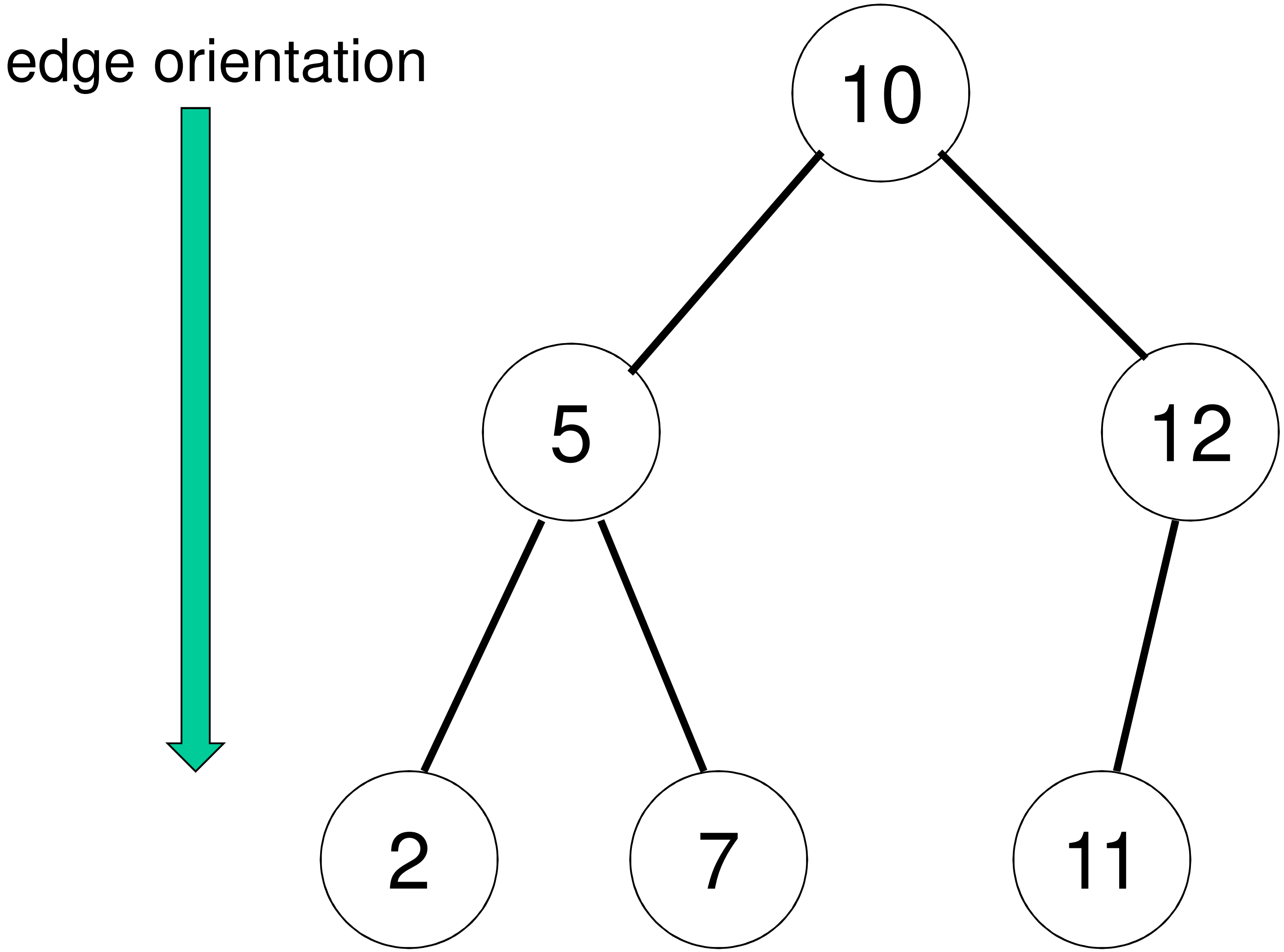

parent (father, precedesor)
root
internal node
child (son)
list

- **rooted tree** with the root **u**
  - directed tree where each path P(**u**,v) is an oriented path
- **node depth** d(x) of the node x in the root tree
  - distance from tree
- **depth of the tree**
  - max d(x) $x \in V$

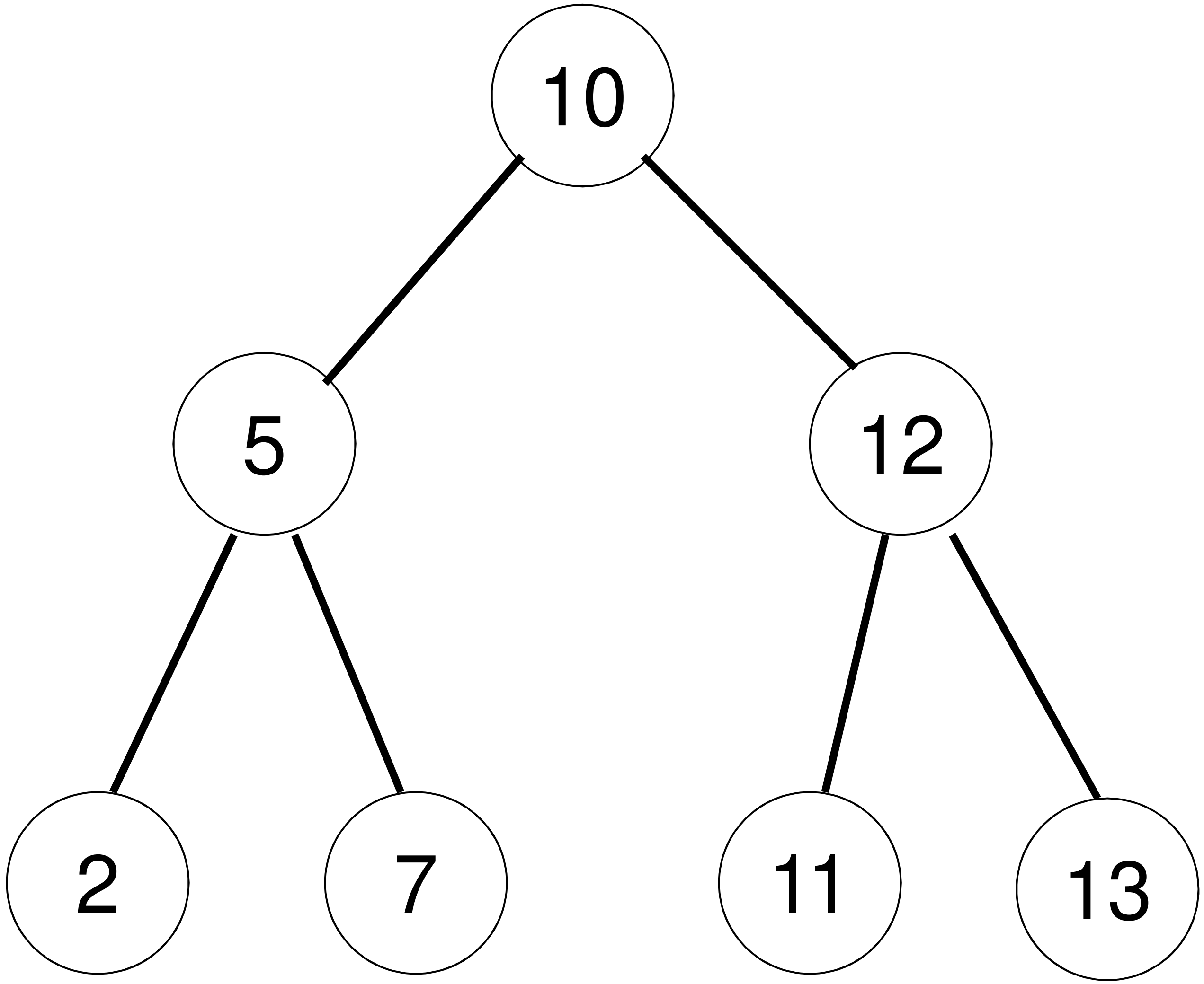
Binary tree

- each node has max. 2 children
- ordered binary tree
  - nodes are labeled (by numbers,…)
  - the label of the left child is always less or equal than parent
  - the label of the right child is always greater than parent
  - ordered binary trees are effective for searching: complexity is $log_2 n$ (height of the tree)

Ordered binary tree



Full binary tree of the depth 2



- full binary tree of the depth *k* has count of nodes

$$n = 2^{k+1}-1$$

- the minimal depth of binary tree (not necessarily full) with n nodes is

$$k = \lfloor \log_2(n) \rfloor$$

floor

– example: tree with $n = 5$ nodes

$$k = \lfloor \log_2(5) \rfloor = \lfloor 2{,}32 \rfloor = 2$$

Tree representation

- pointer to the root (root node)
- root is represented with the structure:
  - label of the node
  - pointers to left and right children (subtrees)

```
typedef struct TNode {
  int label;
  TNode *left;
  TNode *right;
  } TNode;
```

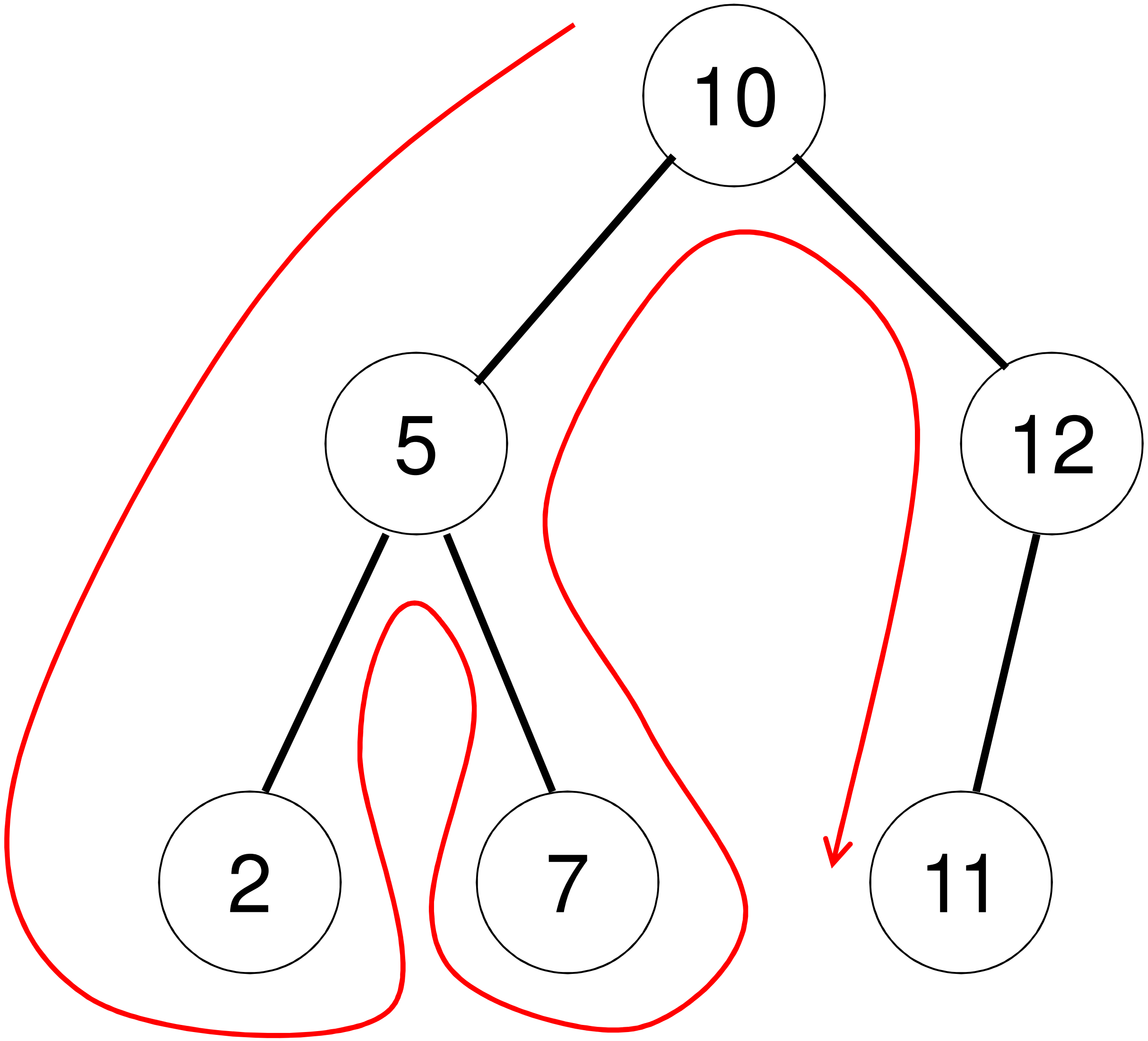
- the list has both pointers left, rights set to `NULL`

Operations over tree

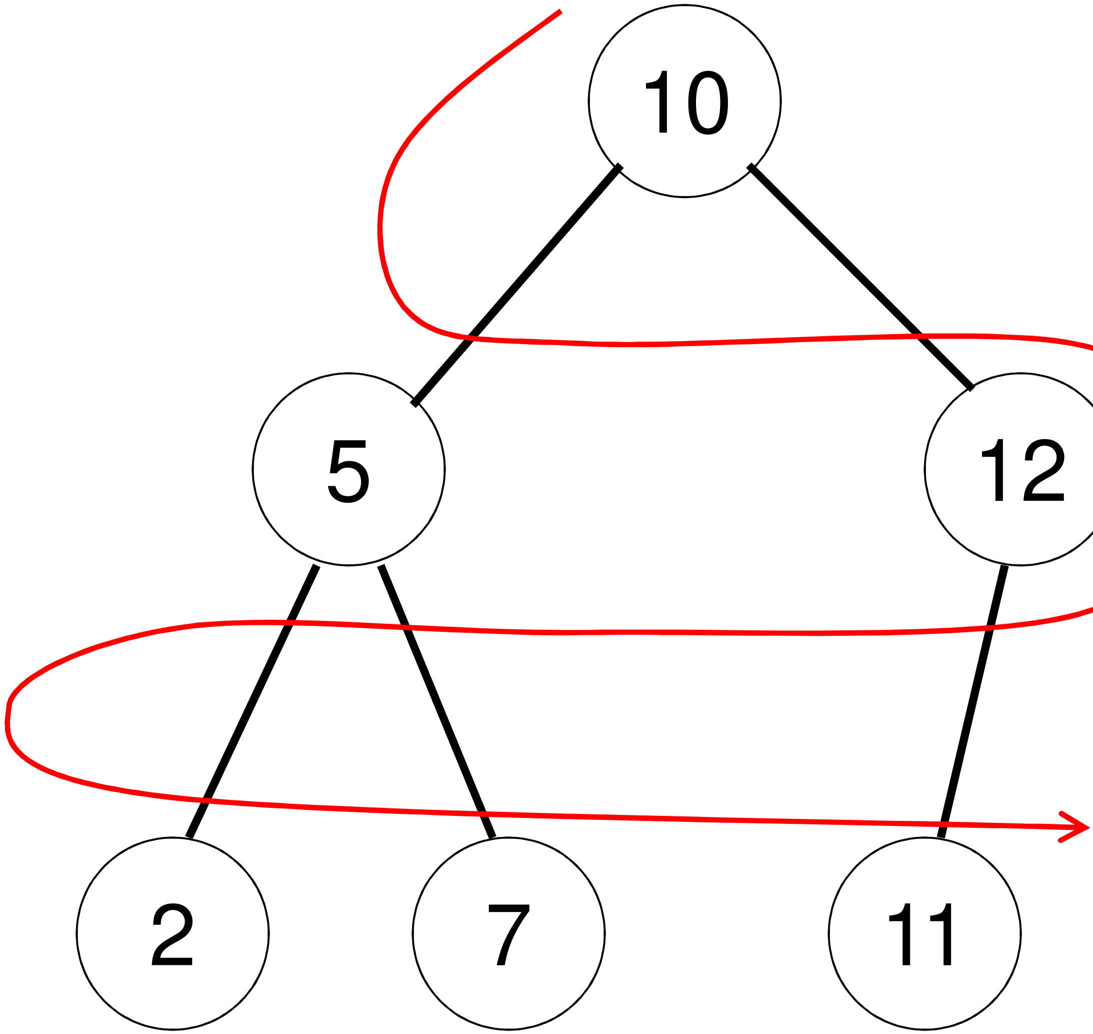
- operations are (usually) recursive
  - depends on the problem
    - complete traversal must be programmed using recursion (tracing depth)
    - searching element (only) can be implemented non-recursively

- operations over tree
  - tree traversal (can be combined with some action)
    - depth-first
    - breadth-first
  - searching node
  - insert new node as a list
  - delete node
  - delete tree

Depth-first Tree Traversal



Breadth-first tree traversal



General depth-first tree traversal recursive algorithm

```
void traverse(TNode *u)
{
  if (u==NULL) return;
  action(u->label);
  traverse(u->left);
  traverse(u->right);
}
```

Variants of depth-tree traversal

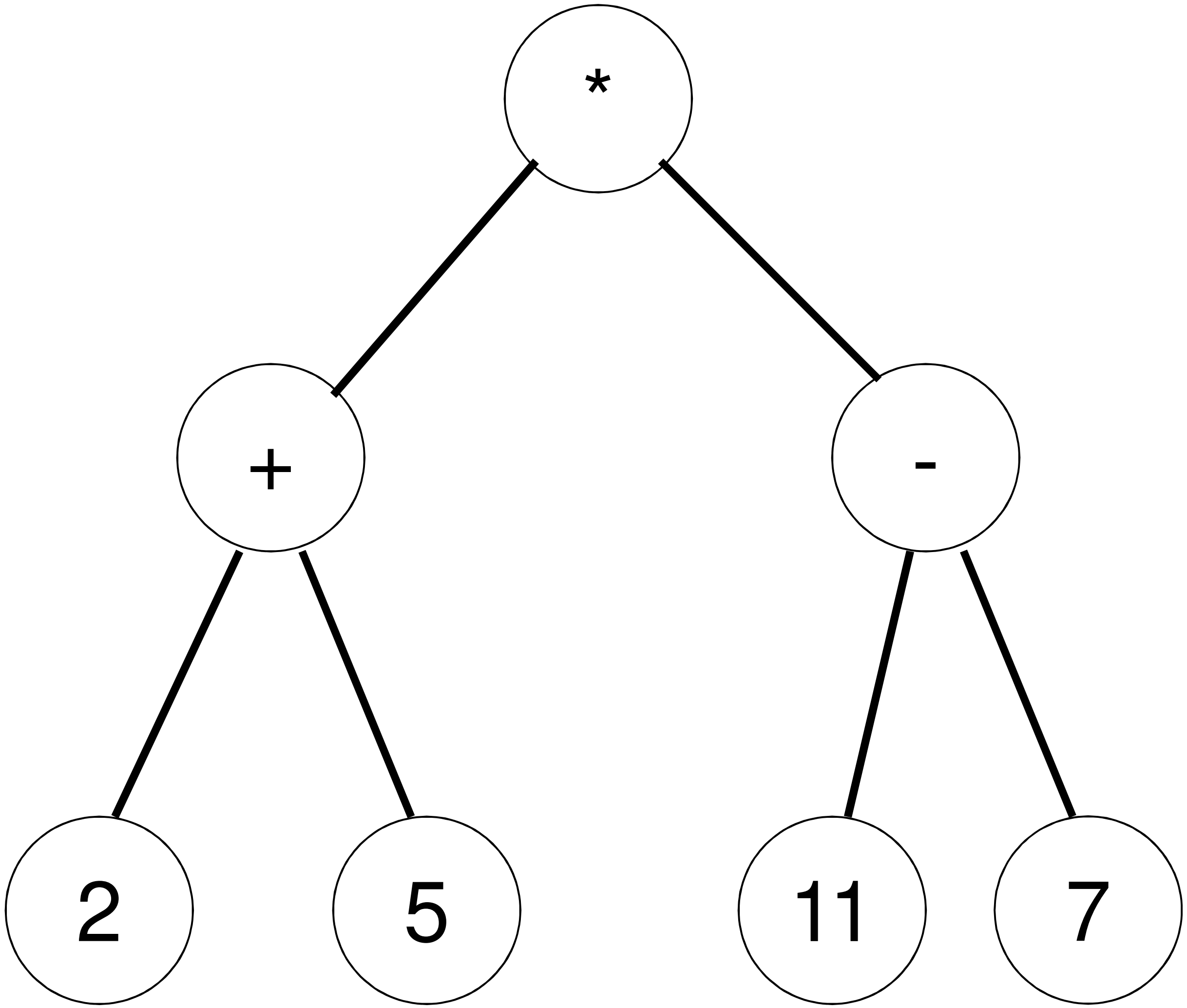
- *left order*
  - left subtree, node action, right subtree
- *right order*
  - right subtree, node action, left subtree
- *preorder*
  - node action, left subtree, right subtree
- *another permutations, if they make sense*

Example – left order

- print all nodes of ordered tree

```
void print_node(TNode *u)
{
  if (u==NULL) return;
  print_node(u->left);
  printf("%d ",u->label);
  print_node(u->right);
}
```

Example – pre order
(expression representation by tree)



- the tree represents expression:

$$(2+5)*(11-7)$$

- print it in pre-order form (Polish notation)

$$* + 2\ 5 - 11\ 7$$

```
void print_node_pre(TNode *u)
{
  if (u==NULL) return;
  printf("%c ",u->label);
  print_node_pre(u->left);
  print_node_pre(u->right);
}
```

Note:

- tree representation is usually used in compilers to represent expressions
- the tree is traversed post-order to evaluate an expression

Node search

- returns 1, if the value is found

```
int find(TNode *u, int x)
{
  if (u==NULL) return 0;
  if (u->label==x) return 1;
  if (x < u->label)
    return find(u->left,x);
  else
    return find(u->right,x);
}
```

Non-recursive version

- returns 1, if the value is found

```
int find_nonrecurs(Node *root, int x)
{
  while(root != NULL && root->label != x)
  {
    if (x < root->label)
      root = root -> left;
    else
      root = root -> right;
  }
  if (root == NULL) return 0;
  else return 1;
}
```

Insert new list

```
void insert_new(TNode **u, int x)
{
 if (*u == NULL)
 {
   *u = (TNode*)malloc(sizeof(TNode));
   (*u) -> label = x;
   (*u) -> left = NULL;
   (*u) -> right = NULL;
 }
 else
  if (x <= (*u)->label)
    insert_new (&((*u)->left),x);
  else insert_new (&((*u)->right),x);
}
```

Delete tree

```
void delete(TNode *u)
{
  if (u==NULL) return;
  delete(u->left); delete(u->right);
  free(u);
}
```

```c
void main(void)
{
  TNode *tree = NULL;

  insert_new(&tree,10);
  insert_new(&tree,5);
  insert_new(&tree,7);
  insert_new(&tree,2);
  insert_new(&tree,12);
  insert_new(&tree,11);
  find(tree,5);
  delete(tree);
}
```

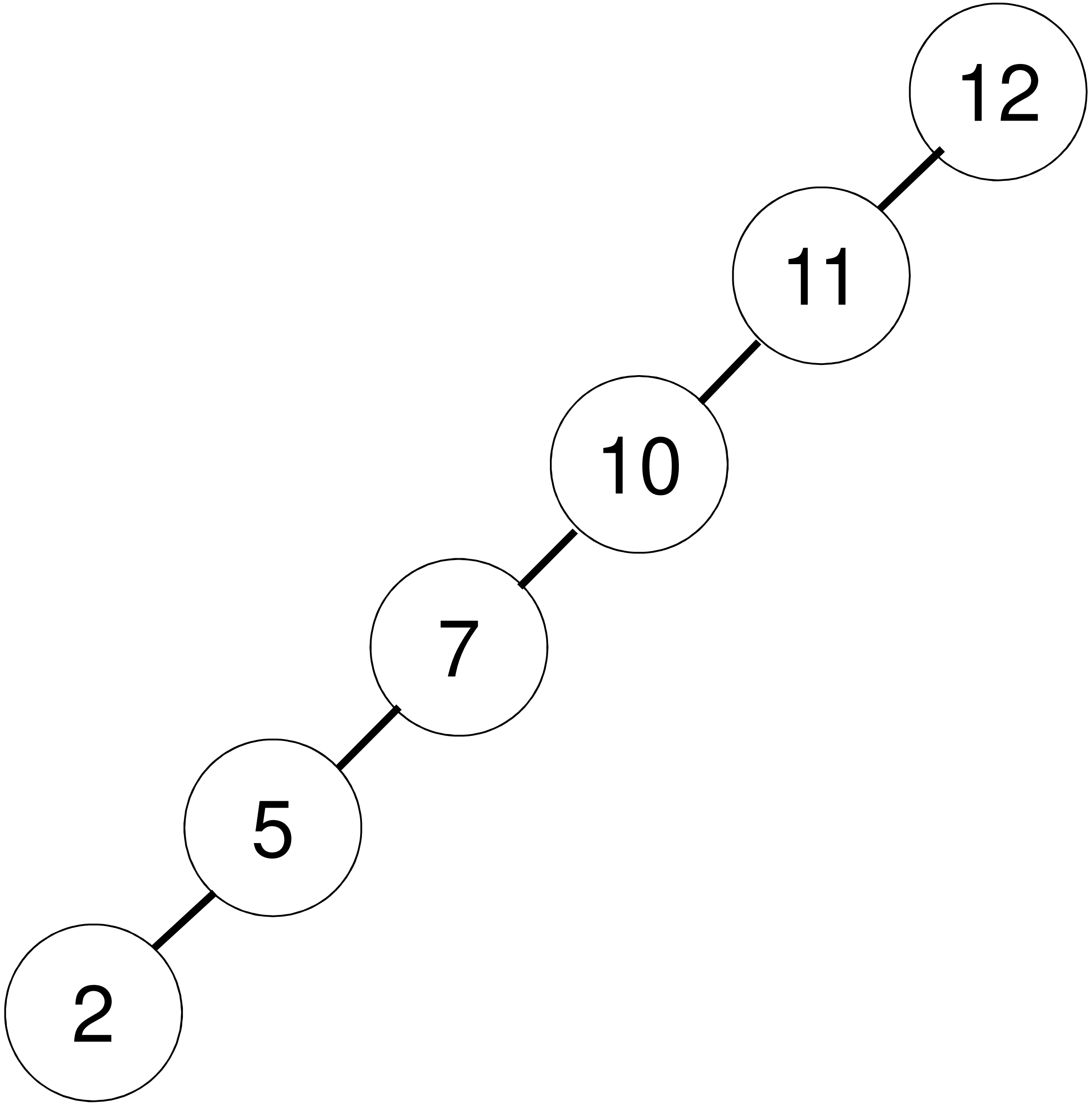
Which tree is created?

```
void main(void)
{
  TNode *tree = NULL;
  insert_new(&tree,10);
  insert_new(&tree,5);
  insert_new(&tree,7);
  insert_new(&tree,2);
  insert_new(&tree,11);
  insert_new(&tree,12);

}
```

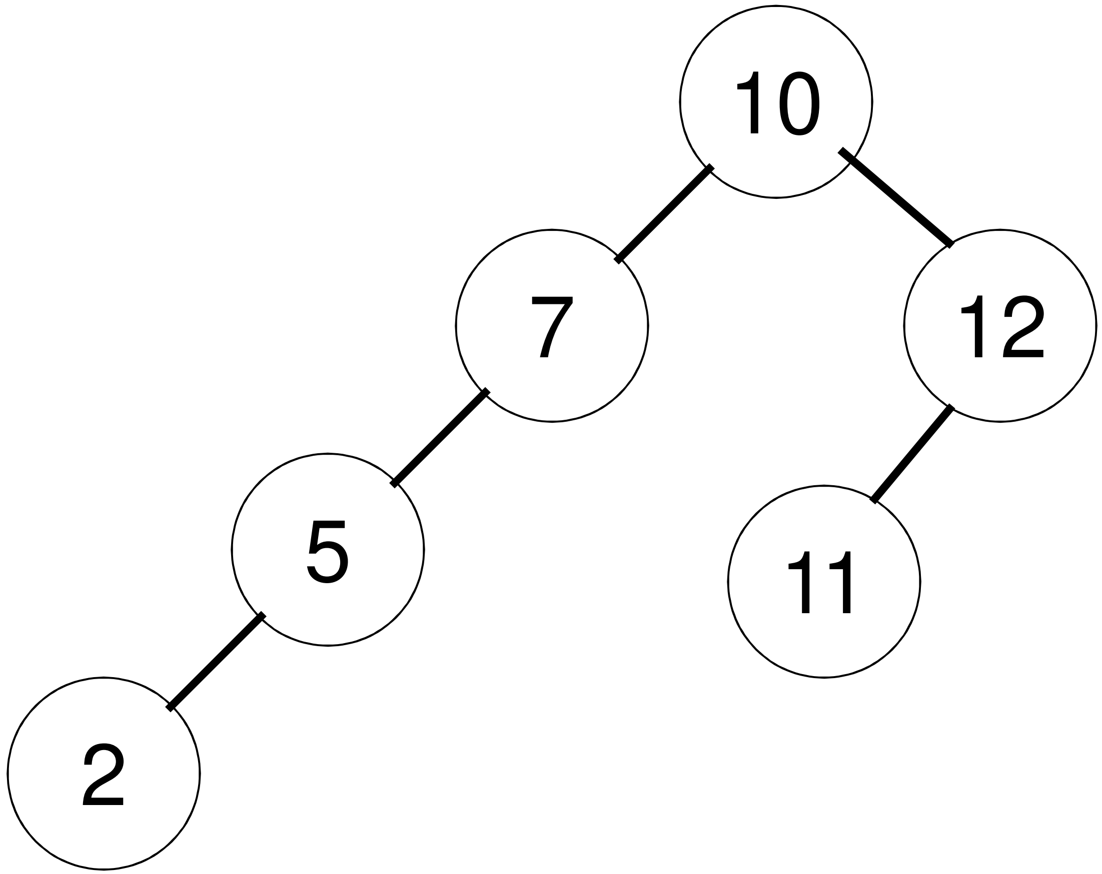
And now?

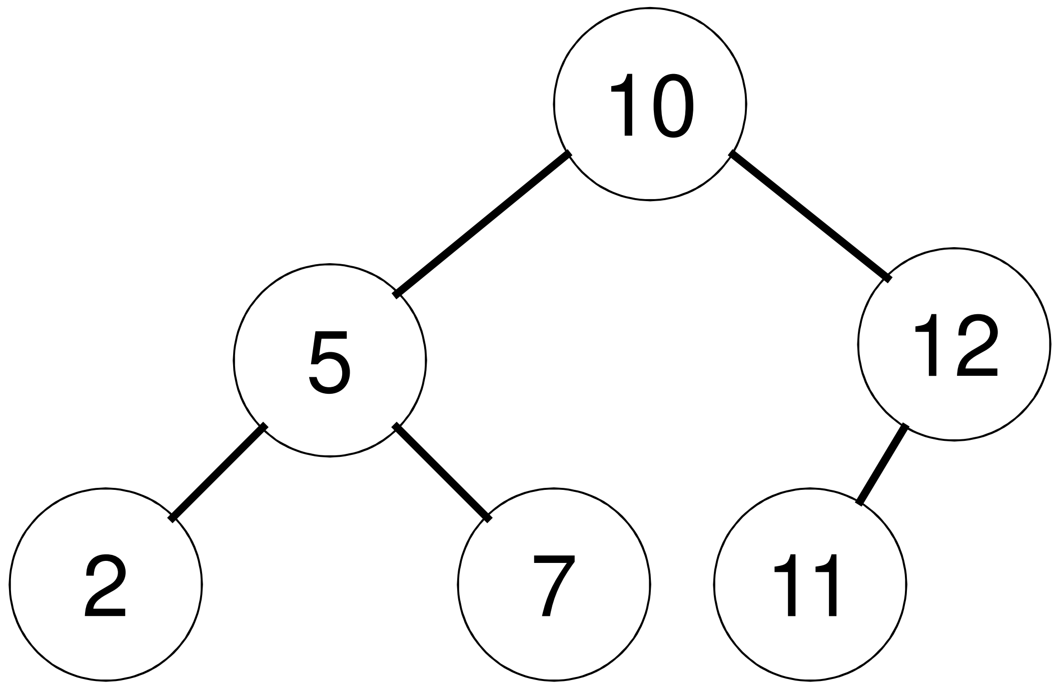
```
void main(void)
{
  TNode *tree = NULL;
  insert_new(&tree,12);
  insert_new(&tree,11);
  insert_new(&tree,10);
  insert_new(&tree,7);
  insert_new(&tree,5);
  insert_new(&tree,2);

}
```

- the tree degrades to the linear list



The aim is to create **balanced** tree, where the height of the left and right tree differs max. 1. It must be satisfied for each subtree (node). If it is not satisfied the operation balancing is executed.




10
5
12
2
7
11