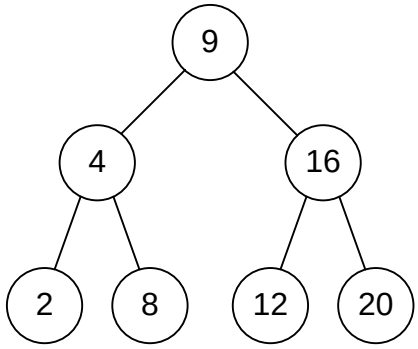
Binary Search Tree

Binary search tree (BST) is a binary tree for which following conditions hold (search tree properties):

- Both subtrees (*left* and *right*) are binary search trees.
- The nodes of left subtree of arbitrary node of *x* have **smaller** key values than those which node *x* has.
- The nodes of right subtree of arbitrary node of *x* have **greater** key values than those which node *x* has.



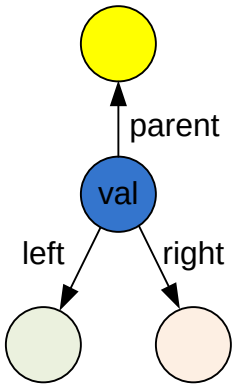
Obviously, data from each node should have keys on which the comparison operation is less defined. Usually, data, which defines a node, is a record, not unique field of information. However, it concerns the implementation, not the nature of a binary search tree.

Declare the node of BST:

```
class TreeNode
{
public:
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

Here *val* – some data which is linked to a node, *left* и *right* – pointers to nodes which are children of current node – left and right nodes correspondingly. For optimization of algorithms concete realizations also assume definitions in each node of *parent* field – pointer to parent element:

```
class TreeNode
{
public:
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode *parent;
  TreeNode(int x, TreeNode *prev) :
           val(x), left(NULL), right(NULL), parent(prev) {}
};
```



The data can have a ***key***, where the operation less (<) is defined. In concrete realizations it can be a pair (*key*, *value*), or a link to such pair, or simple definition of comparison operation on the needed structure or link to it.

For arbitrary *x* the following binary tree properties hold:

$$key[left[x]] < key[x] \leq key[right[x]]$$

The main benefit of binary search tree (which differs it from other structures) is the high efficiency of realization of search and sort algorithms based on it.

Main operations on binary search tree

Main interface of binary search tree consists of 3 operations:
Insert(*key*, *value*) – adding a pair into tree (*key*, *value*).
Remove(*key*) – delete a node which holds a pair (*key*, *value*).
Find(*key*) – find a node where this pair holds (*key*, *value*).

Starting from this point we will look at trees, whose keys are integer values showed as *data* variable.

Adding an element **(Insert)**
Given: *tree* and *val*
Problem: add given *val* into the *tree*.

```
void Insert(TreeNode *&tree, int val)
{
   // if tree is empty, change it to a tree with
   // one root node (data, null, null) and stop.
  if (tree == NULL)
  {
    tree = new TreeNode(val);
    return;
  }

   // otherwise,
   // compare data with with the key of root node tree->val
  if (val < tree->val)
     // if val < tree->val,
     // recursively add val into left subtree of tree
    Insert(tree->left, val);
  else
```

```
    // if val >= tree->val,
    // recursively add val to the right subtree of tree
    Insert(tree->right, val);
}
```

If we need to insert element to the tree that supports *parent* pointer, we pass *prev* pointer (*parent* of a *tree*) to Insert function.

```
void Insert(TreeNode *&tree, TreeNode *prev, int val)
{
  // if tree is empty, change it to a tree with
  // one root node and stop.
  if (tree == NULL)
  {
    tree = new TreeNode(val, prev);
    return;
  }

  // otherwise,
  // compare val with with the key of the root node tree->val
  if (val < tree->val)
    // if val < tree->val,
    // recursively add val to the left subtree of tree
    Insert(tree->left, tree, val);
  else
    // if val >= tree->val,
    // recursively add val to the right subtree of tree
    Insert(tree->right, tree, val);
}
```

If *tree* is a root, to insert *val* to the *tree* we must call a function (parent of the root is NULL):

```
Insert(tree, NULL, val);
```

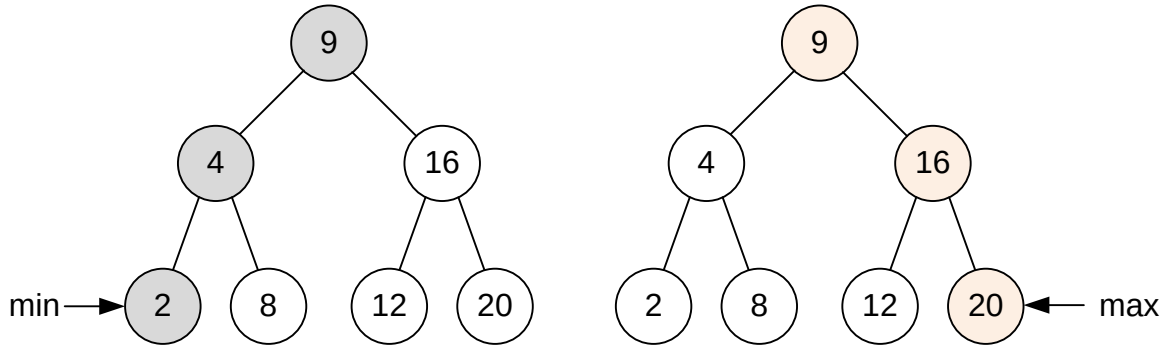
Find the element
Given: *tree* and *element*
Problem: find a node with *element* in a *tree* and return a pointer to it.

```
TreeNode *Find(TreeNode *tree, int element)
{
  // if tree is empty, return null
  if (tree == NULL) return NULL;

  // if element is found, return pointer to the node
  if (element == tree->val) return tree;

  // Otherwise continue the search in the left or in the right subtree
  if (element < tree->val) return Find(tree->left, element);
  else return Find(tree->right, element);
}
```

E-OLYMP **10063. Tree find** Find element in the tree.
► Use the idea described above.

Starting from the root, we compare *element* with the value in the current vertex. If they are equal, search is finished. If element < tree → data, search continues in left subtree, otherwise, in the right subtree. The length of the search is no more than the height of tree.

Find min and max element



To find the vertex with minimum element, we need to move to the left subtree until we reach the vertex, which pointer to the left subtree equals to NULL.

Function Minimum returns pointer to the vertex with minimum element.

```
TreeNode *Minimum(TreeNode *tree)
{
  if (tree == NULL) return tree;
  // while we have left subtree, go there
  while (tree->left != NULL) tree = tree->left;
  return tree;
}
```

To find the vertex with maximum element, we need to move to the right subtree until we reach the vertex, which pointer to the right subtree equals to NULL.

E-OLYMP **10061. Tree minimum element** Return the pointer to minimum element.

► Use the idea described above.

E-OLYMP **10062. Tree maximum element** Return the pointer to maximum element.

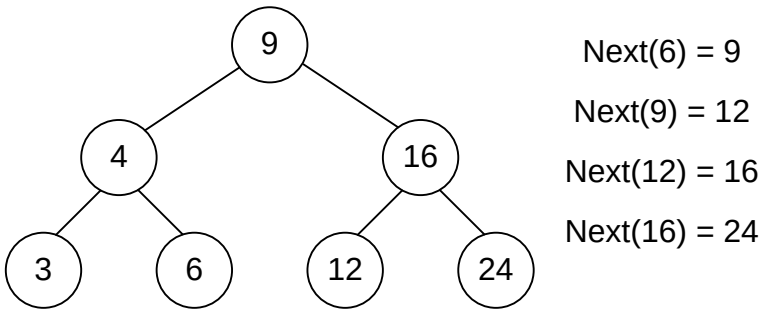
► Use the idea described above.

Find next and previous element

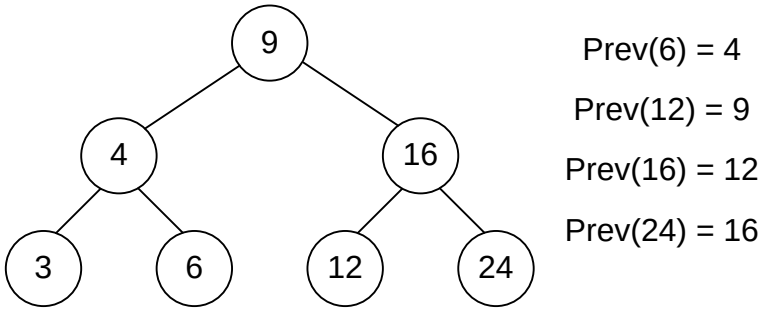
If right subtree is not empty, **next** element will be *min* element of right subtree. Otherwise, we need to move up until we find the vertex, which is left child of its own parent. Actually, this parent is the next element.

```
TreeNode *Next(TreeNode *tree)
{
  // If right subtree exists, next element is
```

```
  // min of right subtree
  if (tree->right != NULL) return Minimum(tree->right);

  // otherwise, we need to move up untill
  // we do not find a node which is left node of its parent
  TreeNode *Prev = tree->parent;
  while ((Prev != NULL) && (tree == Prev->right))
  {
    tree = Prev;
    Prev = Prev->parent;
  }
  return Prev;
}
```



If left subtree is not empty, **previous** element will be *max* element of left subtree. Otherwise, we need to move up until we find the vertex, which is right child of its own parent. Actually, this parent is the previous element.



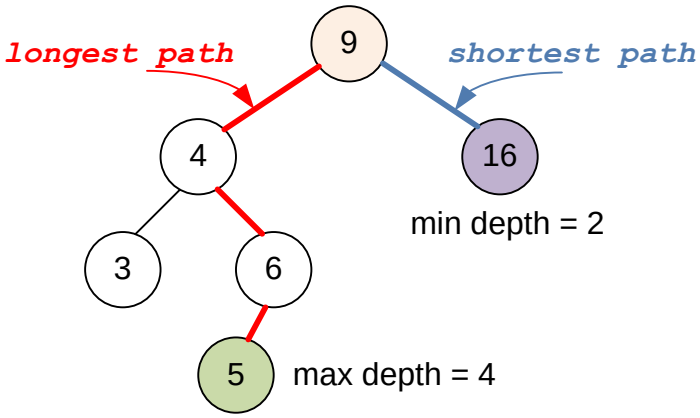
E-OLYMP **10146. Tree Next** Return the pointer to the next element.
► Use the idea described above.

E-OLYMP **10147. Tree Previous** Return the pointer to the previous element.
► Use the idea described above.

Maximum and minimum depth

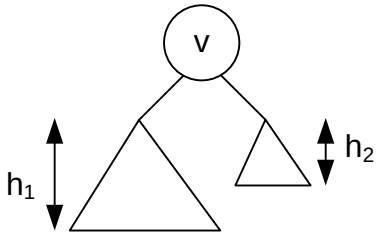
The **maximum depth** is the number of nodes along the *longest path* from the root node down to the farthest leaf node.

The **minimum depth** is the number of nodes along the *shortest path* from the root node down to the nearest leaf node.



The farthest leaf node is 5. The number of nodes along the longest path 9 → 4 → 6 → 5 is 4.

The nearest leaf node is 16. The number of nodes along the shortest path 9 → 16 is 2.



Let h_1 be the **maximum depth** of the left subtree.
Let h_2 be the **maximum depth** of the right subtree.

Then:

- **maximum depth** of the tree (with root at *v*) equals to max(h_1, h_2) + 1;
- **minimum depth** of the tree (with root at *v*) equals to min(h_1, h_2) + 1;

But we must be careful for the case of *minimum depth*:

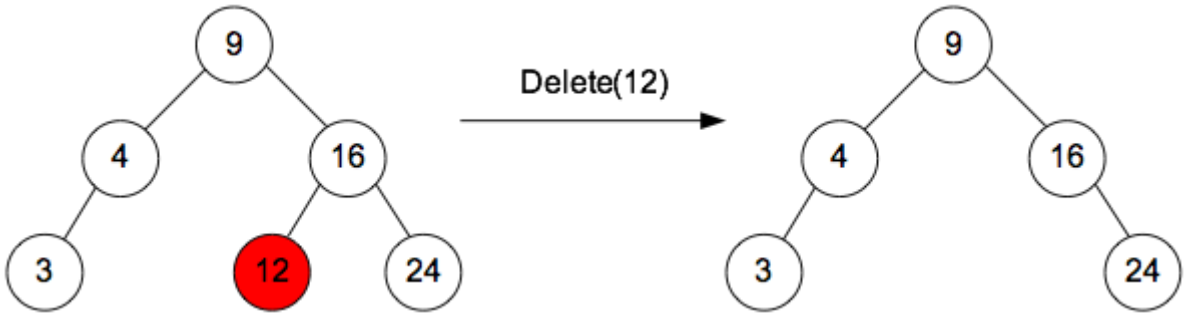
- if $h_1 = 0$ (*v* havn't left child), then minimum depth equals to $h_2 + 1$;
- if $h_2 = 0$ (*v* havn't right child), then minimum depth equals to $h_1 + 1$;

E-OLYMP **10109. Tree Minimum depth** Find the minimum depth of the tree.
► Use the idea described above.

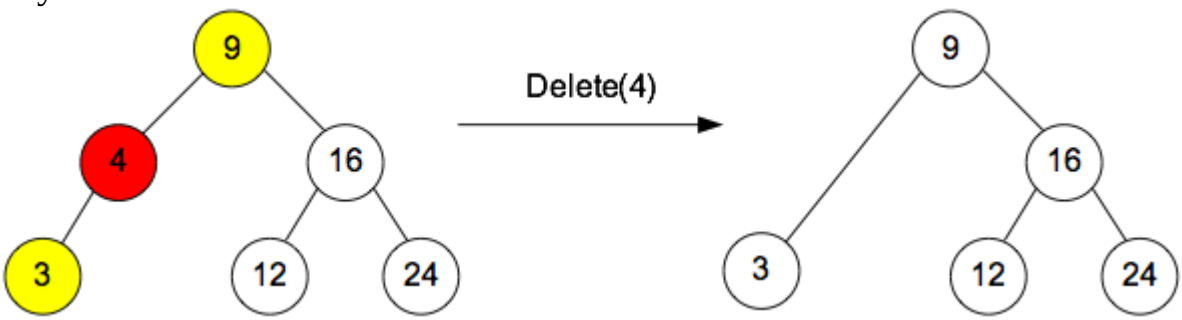
E-OLYMP **10110. Tree Maximum depth** Find the maximum depth of the tree.
► Use the idea described above.

Deleting an element
Given: tree T and given *data*
Problem: delete a node with *data* from the tree T.

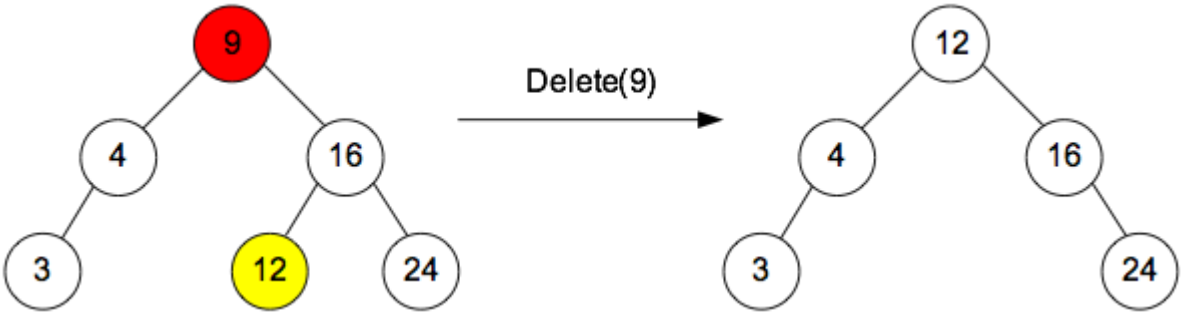
Let *z* is the value to be deleted. While deleting 3 cases may appear:
1. If z does not have children, for deleting it we need to put NULL into corresponding field of his parent.

2. 

2. If z has only 1 child, we can cut z by connecting his parent with his own child dirrectly.



3. Let z has 2 children. And *y* goes after z and it does not have left child. We copy information from the vertex *y* into vertex *z*, and delete *y* itself like we did before.



Tree traversals

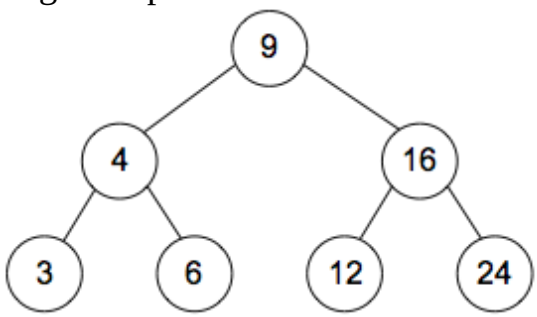
The following are three traversals of a tree:

- InOrder – centred: going through left subtree, root, right subtree;
- PreOrder – forward: going through root, left subtree, right subtree;
- PostOrder – reverse: going through left subtree, right subtree, root;

```
void InOrder(TreeNode *tree)
{
  if (tree == NULL) return;
  InOrder(tree->left);
  printf("%d ", tree->val);
  InOrder(tree->right);
}

void PreOrder(TreeNode *tree)
{
  if (tree == NULL) return;
```

```
  printf("%d ", tree->val);
  PreOrder(tree->left);
  PreOrder(tree->right);
}

void PostOrder(TreeNode *tree)
{
  if (tree == NULL) return;
  PostOrder(tree->left);
  PostOrder(tree->right);
  printf("%d ", tree->val);
}
```

Let's look at the following example:



PreOrder: 9, 4, 3, 6, 16, 12, 24.
InOrder: 3, 4, 6, 9, 12, 16, 24.
PostOrder: 3, 6, 4, 12, 24, 16, 9.

E-OLYMP **10057. Tree PreOrder Traversal** Implement a **PreOrder** traversal of a tree.

► Use the idea described above.

E-OLYMP **10059. Tree InOrder Traversal** Implement an **InOrder** traversal of a tree.

► Use the idea described above.

E-OLYMP **10060. Tree PostOrder Traversal** Implement a **PostOrder** traversal of a tree.

► Use the idea described above.

E-OLYMP **10064. Tree Sum of elements** Binary tree is given. Find the sum of values in all its nodes.

► Run any traversal and find the sum of values in all vertices.

E-OLYMP **10113. Tree Sum of leaves** Binary tree is given. Find the sum of all its leaves.

► Run any traversal. Vertex is a leaf if both its left and right child are NULL.

E-OLYMP **10111. Tree Sum of left leaves** Binary tree is given. Find the sum of all its left leaves.

► Run any traversal. But how to detect a left leaf? Let at this moment we are at some vertex *v*. If its left child does not have left and right child, then this left child is a left leaf.