EECE Department

ELC 303-B

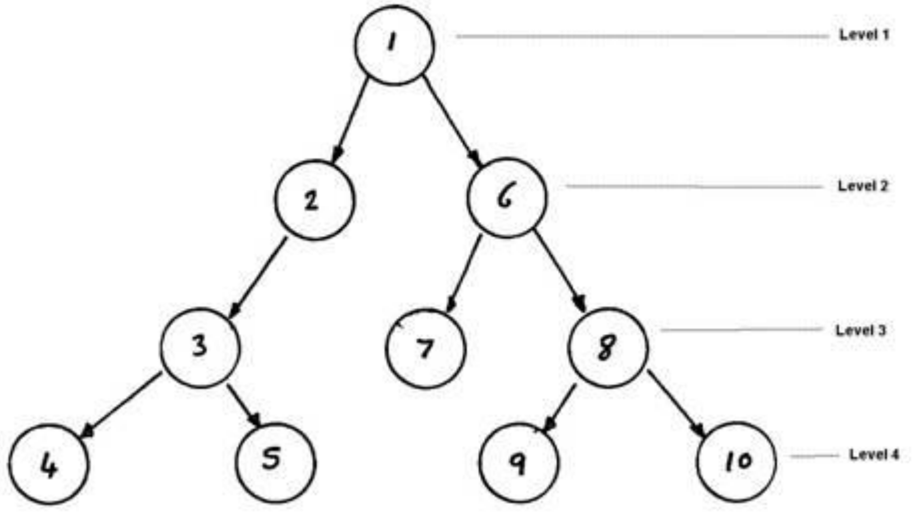
Trees

Dr. Ahmed Khattab

Overview

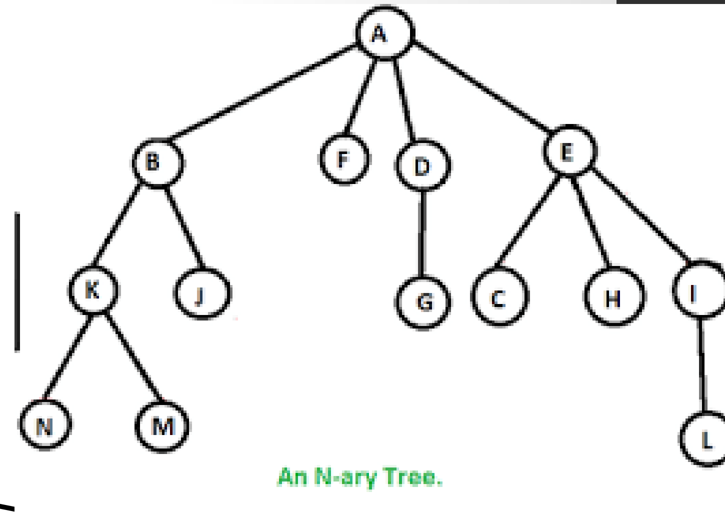
- Trees are a flexible data structure useful for solving a wide range of problems.
- Trees represent data in a hierarchical manner, not linear
- Binary search trees allow rapid retrieval by key, plus in-order processing.
- Recursion is used frequently

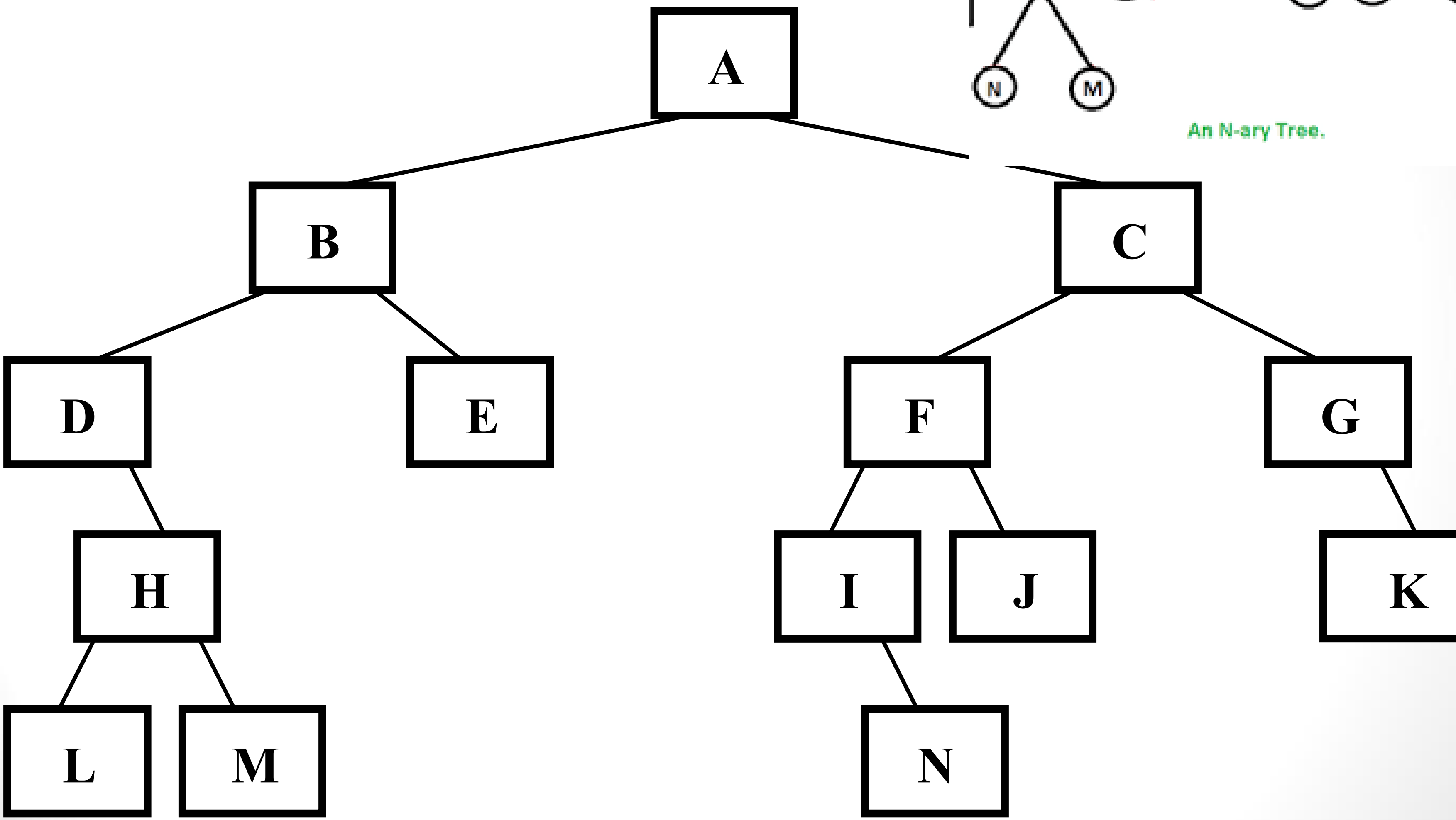
A Tree



- Node - tree element
- Root - the node at the top
- Parent/child nodes
- Degree - number of child nodes
- Subtree
- Siblings - have the same parent
- Ancestor/ Descendant
- Leaves - nodes without children
- Internal nodes - nodes with children
- Edge/branch/link/arc: connection between one node and another
- Path: sequence of edges from root to node
- Length of the path: number of edges in the path
- Depth of a node: length of path from root to node
- Levels: 1 + the number of edges between the node the root.
- Height – maximum level of a node in a tree

A Binary Tree vs. N-ary tree





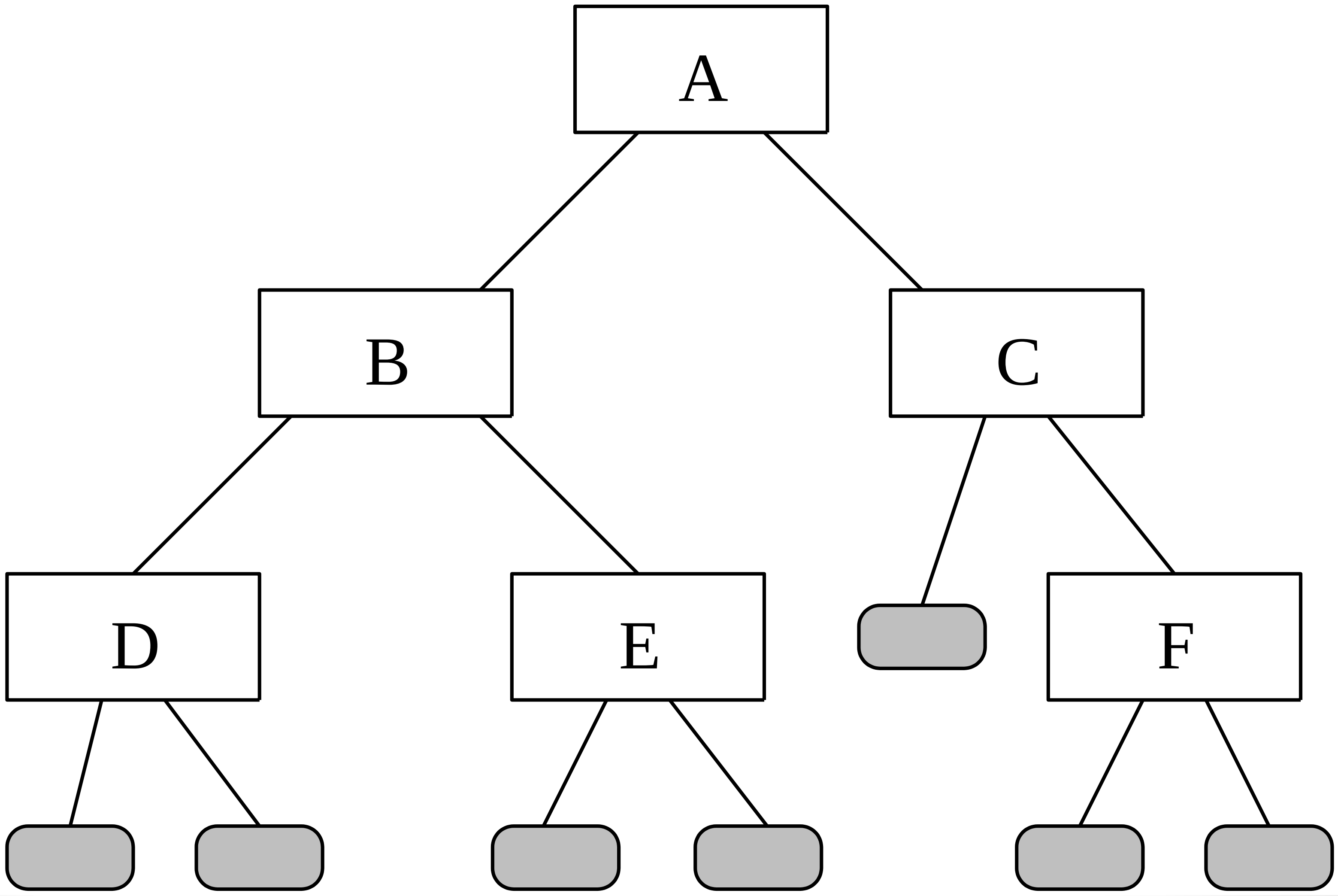
Binary Tree Definition

- A binary tree is either:

    1. an empty tree; or

    2. consists of a node, called a root, and two children, left and right, each of which are themselves binary trees.

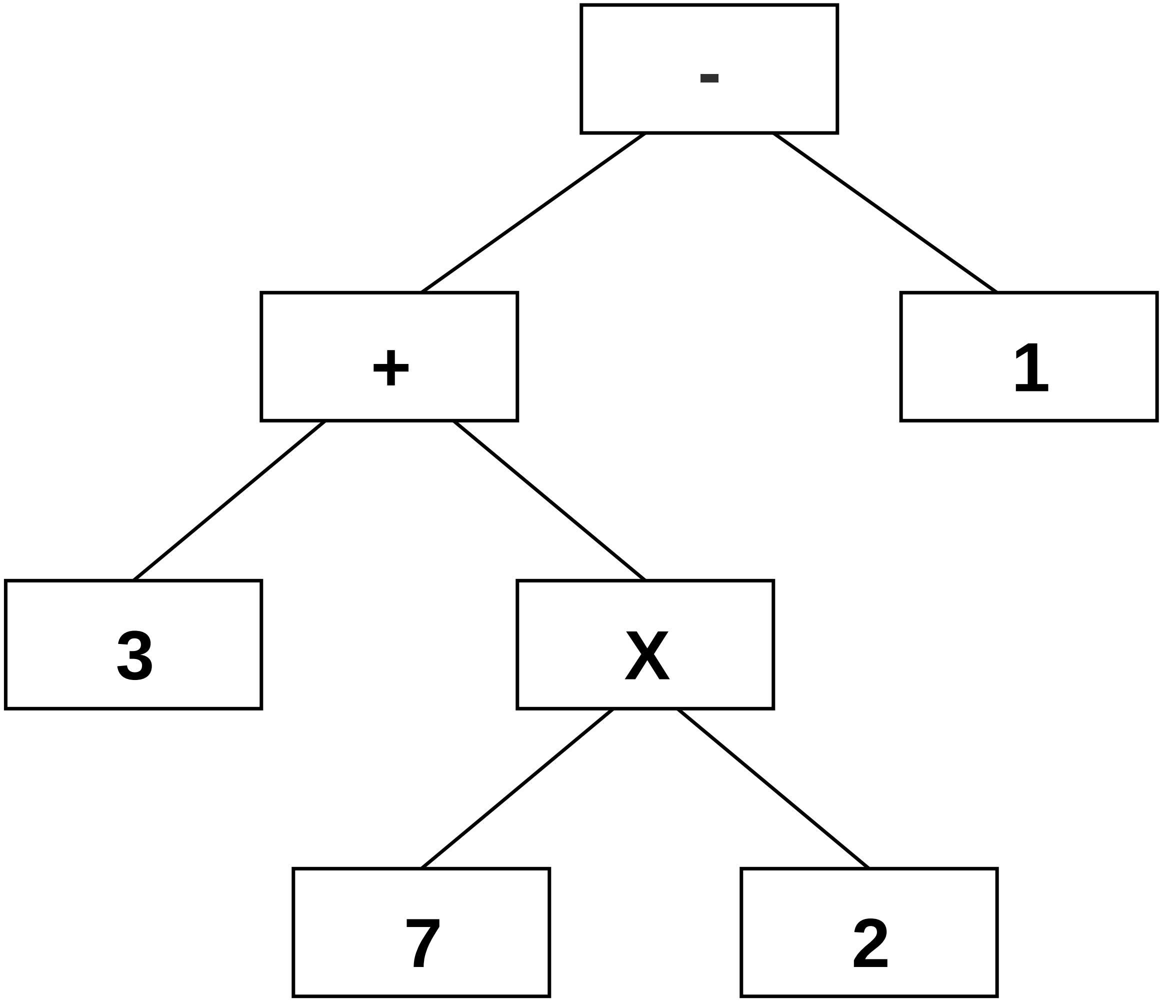
A Binary Tree



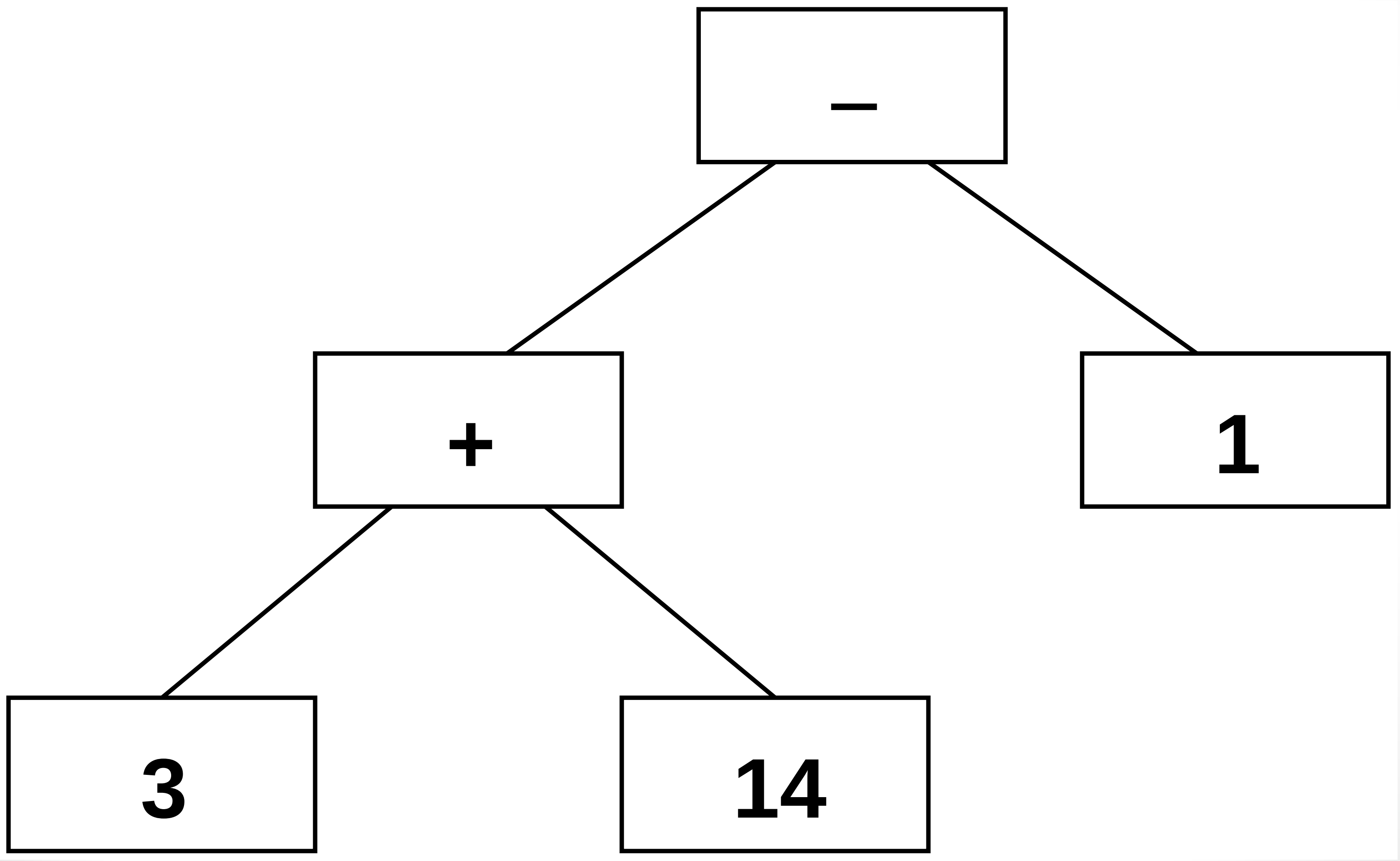
Expression Trees

- 3+7*2-1
- 3+(7*2)-1 grouping for * precedence
- (3+(7*2))-1 left->right associative + vs. -

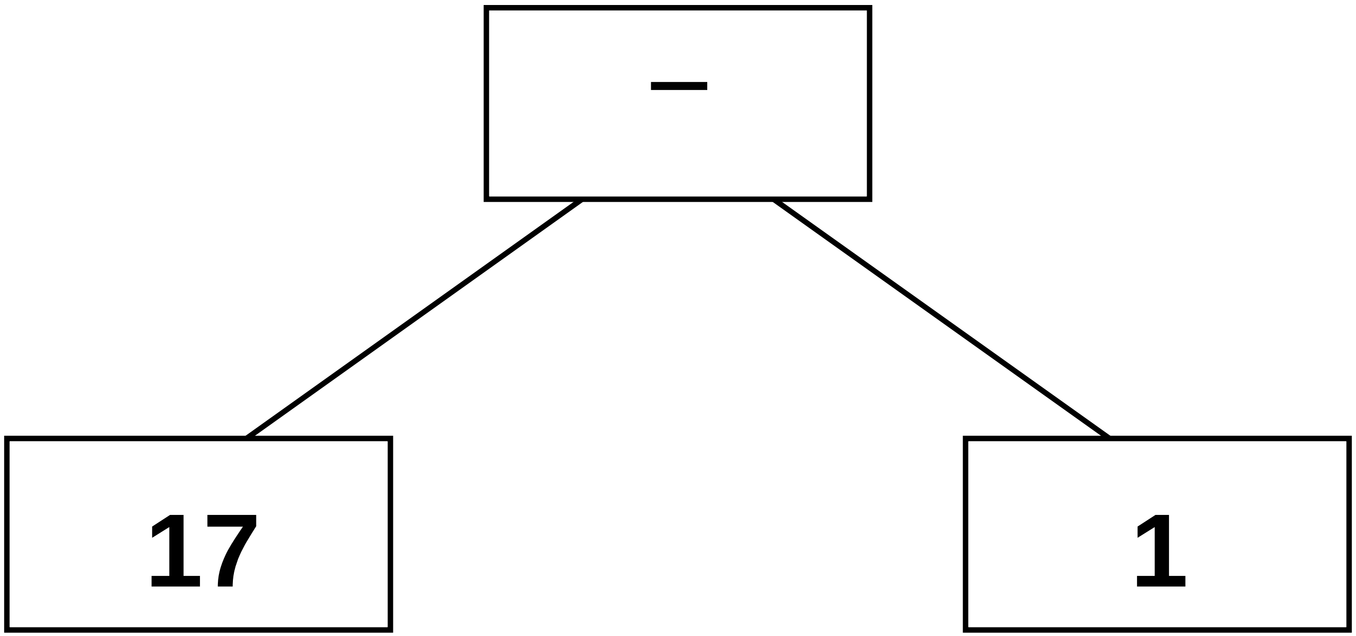
Expression tree for 3 + 7 * 2 - 1



Expression Tree After First Subtree Eliminated



Expression Tree After Second Subtree Eliminated



Final Value of Expression Tree

16

Implementing Binary Trees

- Two methods:

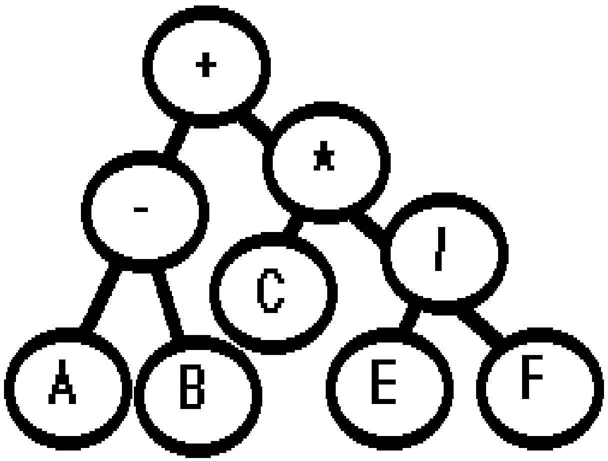
    1. Linear representation - array

    2. Linked representation - pointers

BT Linear representation

1. Allocate an array of size $2^{(depth+1)}-1$
2. Store root in location 1
3. For node in location n, store left child in location 2n, right child in location 2n+1

Example: What is the needed array size for depth=3?



1	+
2	-
3	*
4	A
5	B
6	C
7	/
8	
9	
10	
11	
12	
13	
14	E
15	F

Tradeoffs

- Fast access (given a node, its children and parent can be found very quickly)
- Slow updates (inserts and deletions require physical reordering)
- Wasted space (partially filled trees)

Binary Tree ADT

- **Characteristics**
  - A Binary Tree ADT T stores data of some type (btElementType)
- **Operations**
  - isEmpty
  - getData
  - insert
  - left
  - right
  - makeLeft
  - makeRight

Binary Tree Header File

```
template < class btElementType >
class BinaryTree {
public:
   BinaryTree();
   bool isEmpty() const;
   // Precondition: None.
   // Postcondition: None.
   // Returns: true if and only if T is an empty tree
```

getData( ), insert( )

```
btElementType getData() const;
  // getData is an accessor
  // Precondition: !this->isEmpty()
  // Postcondition: None
  // Returns: data associated with the root of the tree

void insert(const btElementType & d);
  // Precondition: none
  // Postconditions: this->getData() == d;     !this->isEmpty()
```

left( ) and right( )

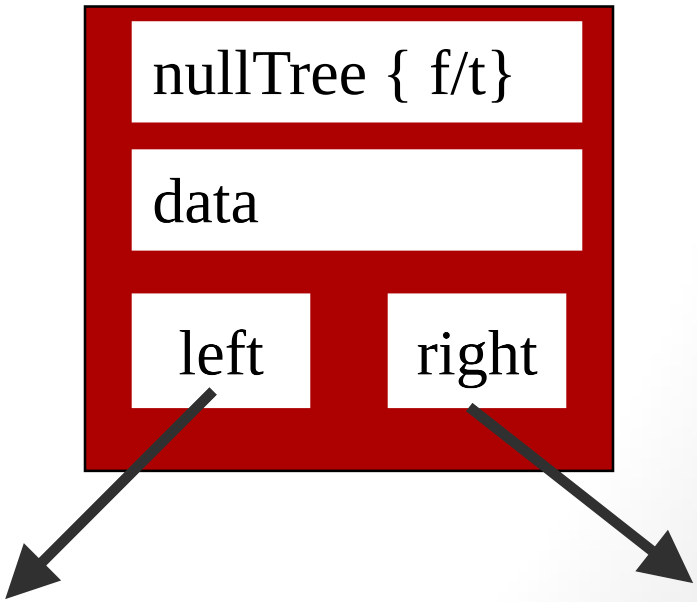
```
BinaryTree * left();
  // Precondition: !this->isEmpty()
  // Postcondition: None
  // Returns: (a pointer to) the left child of T

BinaryTree * right();
  // Precondition: !this->isEmpty()
  // Postcondition: None
  // Returns: (a pointer to) the right child of T
```

makeLeft( ), makeRight( )

```
void makeLeft(BinaryTree * T1);
  // Precondition: !this->isEmpty();
       this->left()->isEmpty()
  // Postcondition: this->left() == T1

void makeRight(BinaryTree * T1);
  // Precondition: !this->isEmpty();
       this->right()->isEmpty()
  // Postcondition: this->right() == T1
```

Private Section

```
private:
  bool nullTree;
  btElementType treeData;
  BinaryTree * leftTree;
  BinaryTree * rightTree;
};
```



Implementation file: Constructor

```
template < class btElementType >
BinaryTree < btElementType > :: BinaryTree()
{
   nullTree = true;
   leftTree = NULL;
   rightTree = NULL;
}
```

isEmpty( )

```
template < class btElementType >
bool
BinaryTree < btElementType > :: isEmpty() const
{
   return nullTree;
}
```

getData( )

```
template < class btElementType >
btElementType
BinaryTree < btElementType > :: getData() const
{
   assert(!isEmpty());
   return treeData;
}
```

insert( )

```
template < class btElementType >
void BinaryTree < btElementType >
:: insert(const btElementType & d)
{
  treeData = d;
  if (nullTree) {
    nullTree = false;
    leftTree = new BinaryTree;
    rightTree = new BinaryTree;
  } }
```

left( )

```
template < class btElementType >
BinaryTree < btElementType > *
BinaryTree < btElementType > :: left()
{
   assert(!isEmpty());
   return leftTree;
}
```

right( )

```
template < class btElementType >
BinaryTree < btElementType > *
BinaryTree < btElementType > :: right()
{
    assert(!isEmpty());
   return rightTree;
}
```

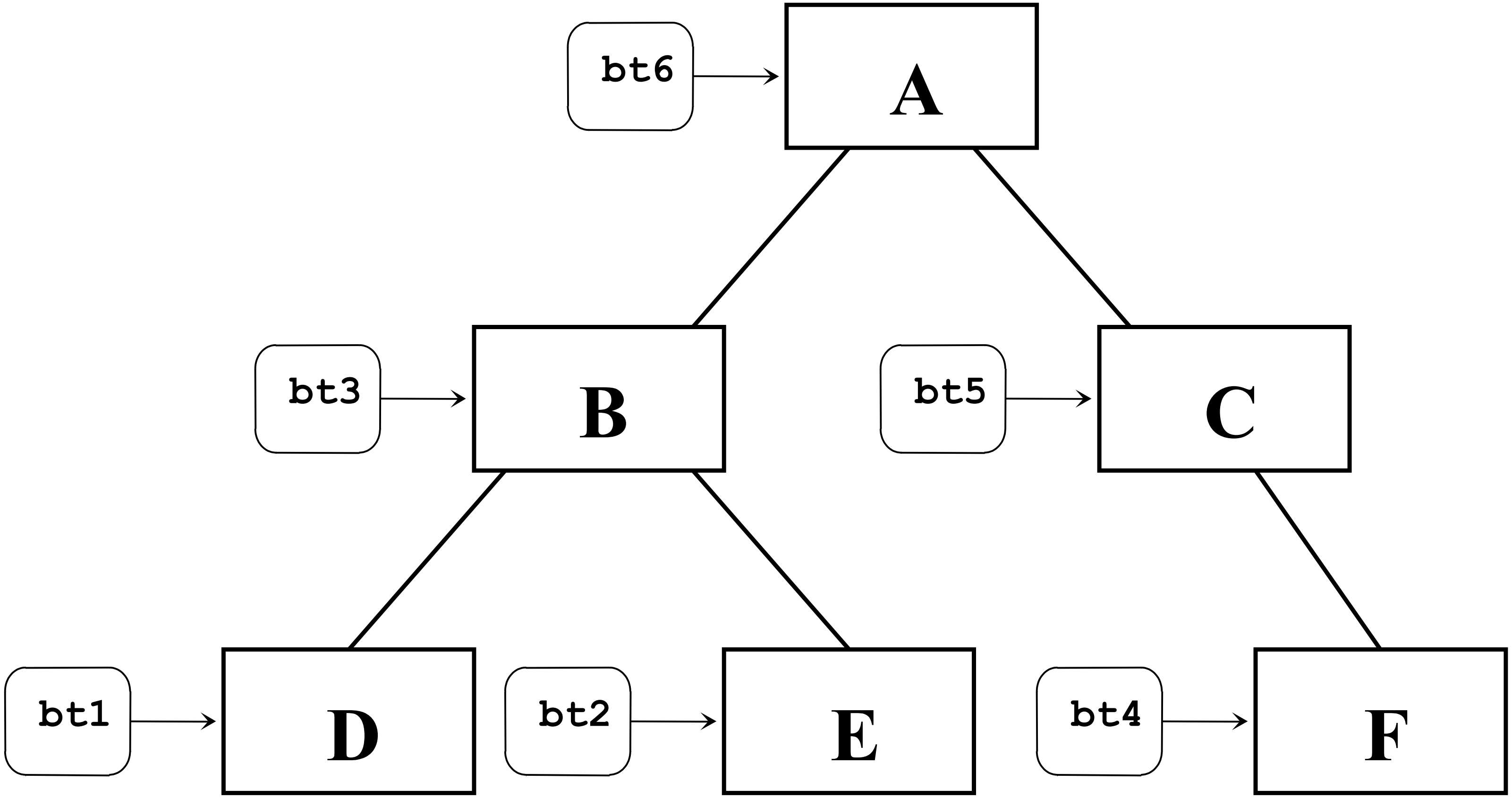
makeLeft( )

```
template < class btElementType >
void BinaryTree < btElementType >
:: makeLeft(BinaryTree * T1)
{
  assert(!isEmpty());
  assert(left()->isEmpty());
  delete left();  // could be nullTree true, w/data
  leftTree = T1;
}
```

makeRight( )

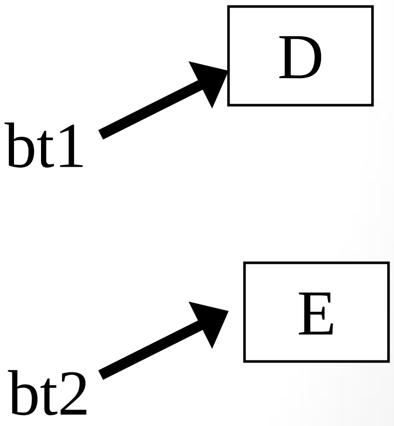
```
template < class btElementType >
void BinaryTree < btElementType >
:: makeRight(BinaryTree * T1)
{
  assert(!isEmpty());
  assert(right()->isEmpty());
  delete right();
  rightTree = T1;
}
```

The Operation of Client Code Example



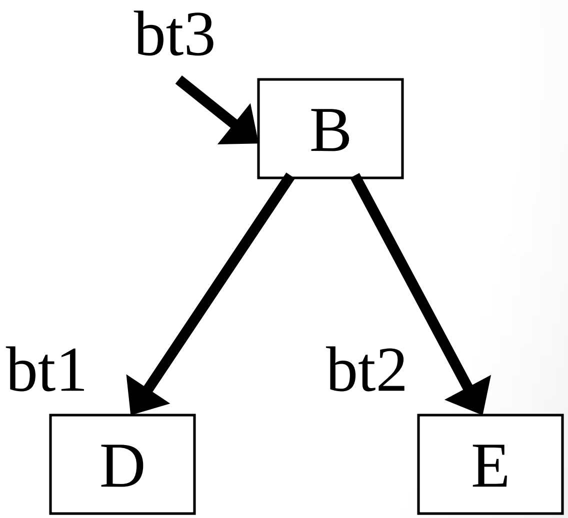
Simple Client for Binary Tree

```
int main()
{ typedef BinaryTree < char > charTree;
  typedef charTree * charTreePtr;
// Create left subtree (rooted at B)
// Create B's left subtree
  charTreePtr bt1=new charTree;
  bt1->insert('D');
  // Create B's right subtree
  charTreePtr bt2=new charTree;
  bt2->insert('E');
```



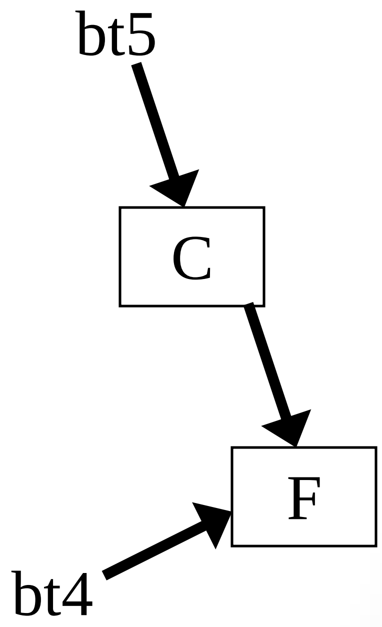
Create Tree

```
// Create node containing B, and link
// up to subtrees
charTreePtr bt3=new charTree;
bt3->insert('B');
bt3->makeLeft(bt1);
bt3->makeRight(bt2);
// ** done creating left subtree
```



Create Right Subtree

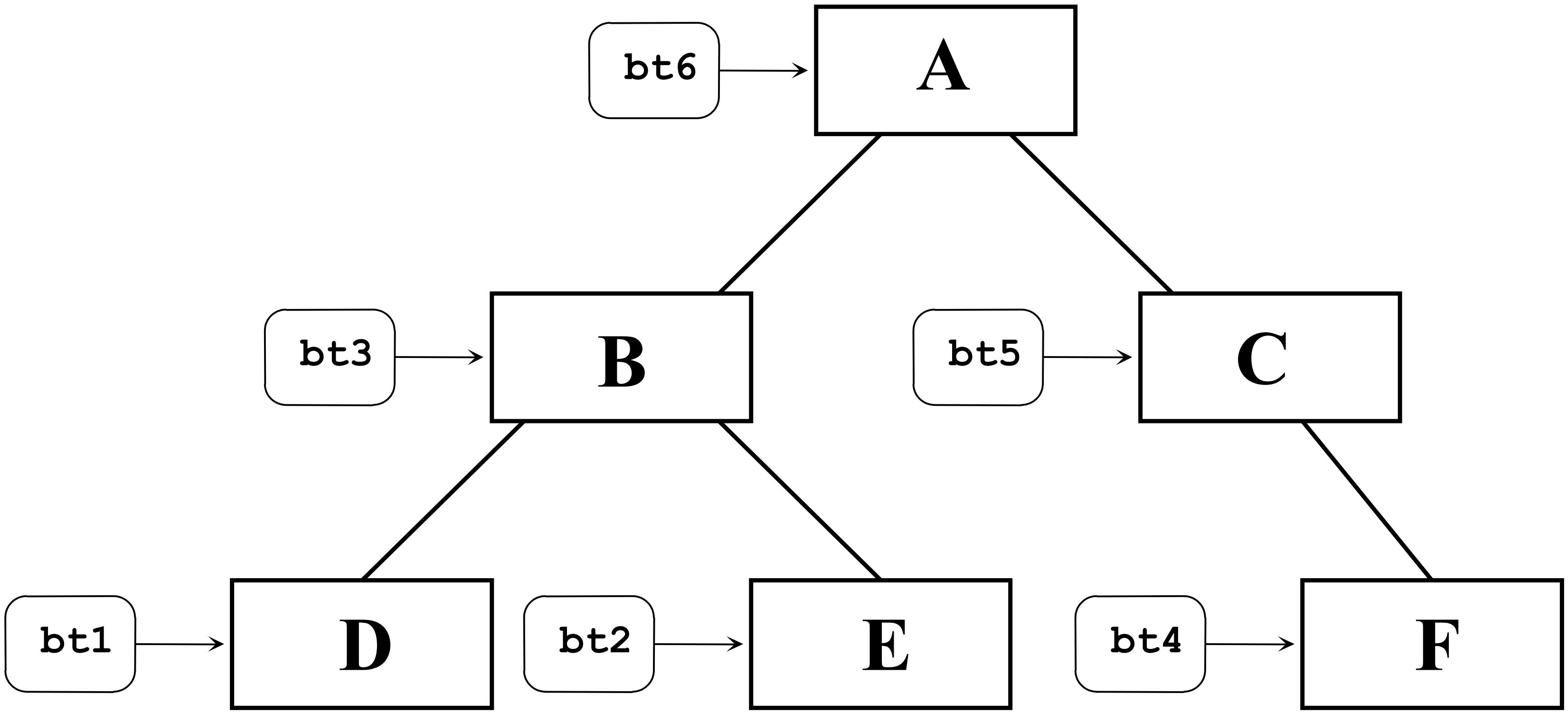
```
// Create C's right subtree
  charTreePtr bt4=new charTree;
  bt4->insert('F');
  // Create node containing C, and link
  // up its right subtree
  charTreePtr bt5=new charTree;
  bt5->insert('C');
  bt5->makeRight(bt4);
  // ** done creating right subtree
```



Create the Root of the Tree

```
charTreePtr bt6(new charTree);
 bt6->insert('A');
 bt6->makeLeft(bt3);
 bt6->makeRight(bt5);

 // print out the root
 cout << "Root contains: " << bt6->getData() << endl;
```

Final Product



Print Left and Right Subtrees

```
// print out root of left subtree
  cout << "Left subtree root: " <<
        bt6->left()->getData() << endl;

// print out root of right subtree
  cout << "Right subtree root: " <<
        bt6->right()->getData() << endl;
```

Print Extreme Child Nodes

```
    cout << "Leftmost child is: " <<
      bt6->left()->left()->getData() << endl;

    cout << "Rightmost child is: " <<
      bt6->right()->right()->getData() << endl;

    return 0;
}
```

Methods of Tree Traversal

- Must visit every element once
- Must not miss any
- Three basic types
  - preorder
  - inorder
  - postorder

Preorder Traversal

- if the tree is not empty
- visit the root
- preOrderTraverse(left child)
- preOrderTraverse(right child)

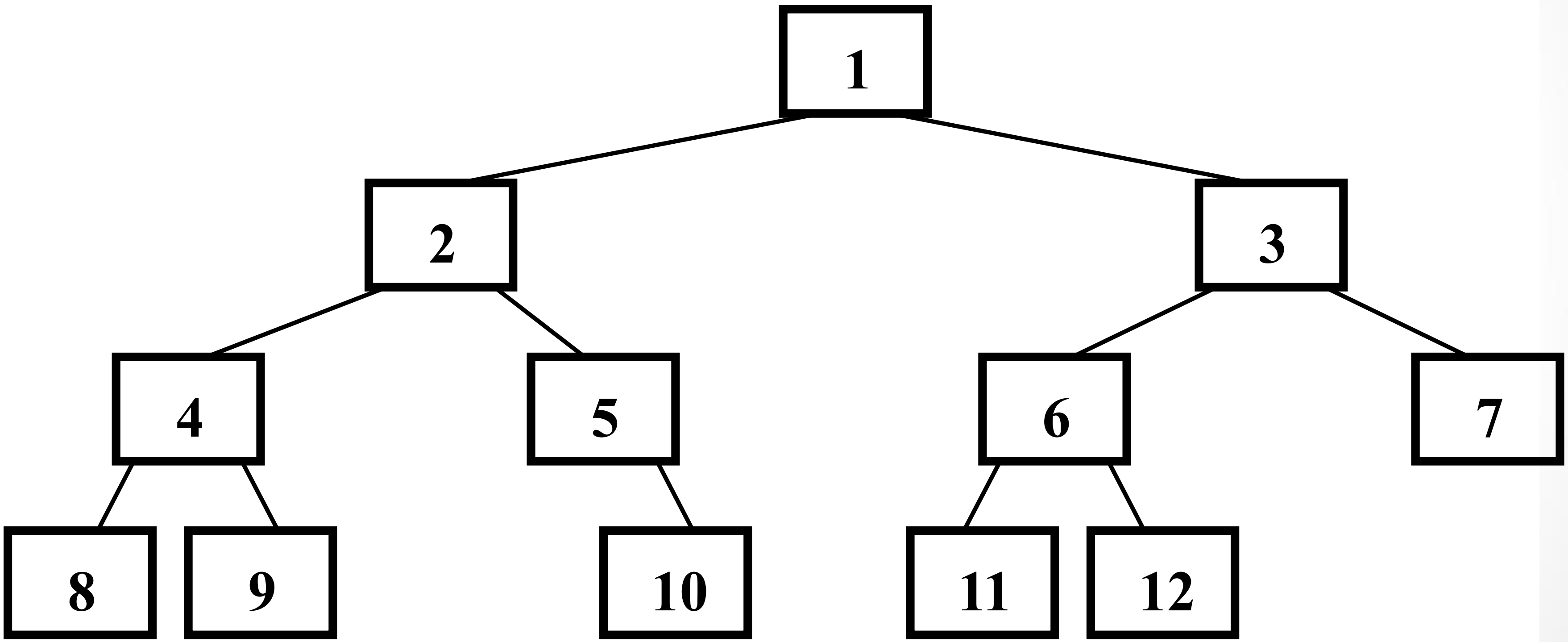
Inorder traversal

- if the tree is not empty
- inOrderTraverse(left child)
- visit the root
- inOrderTraverse(right child)

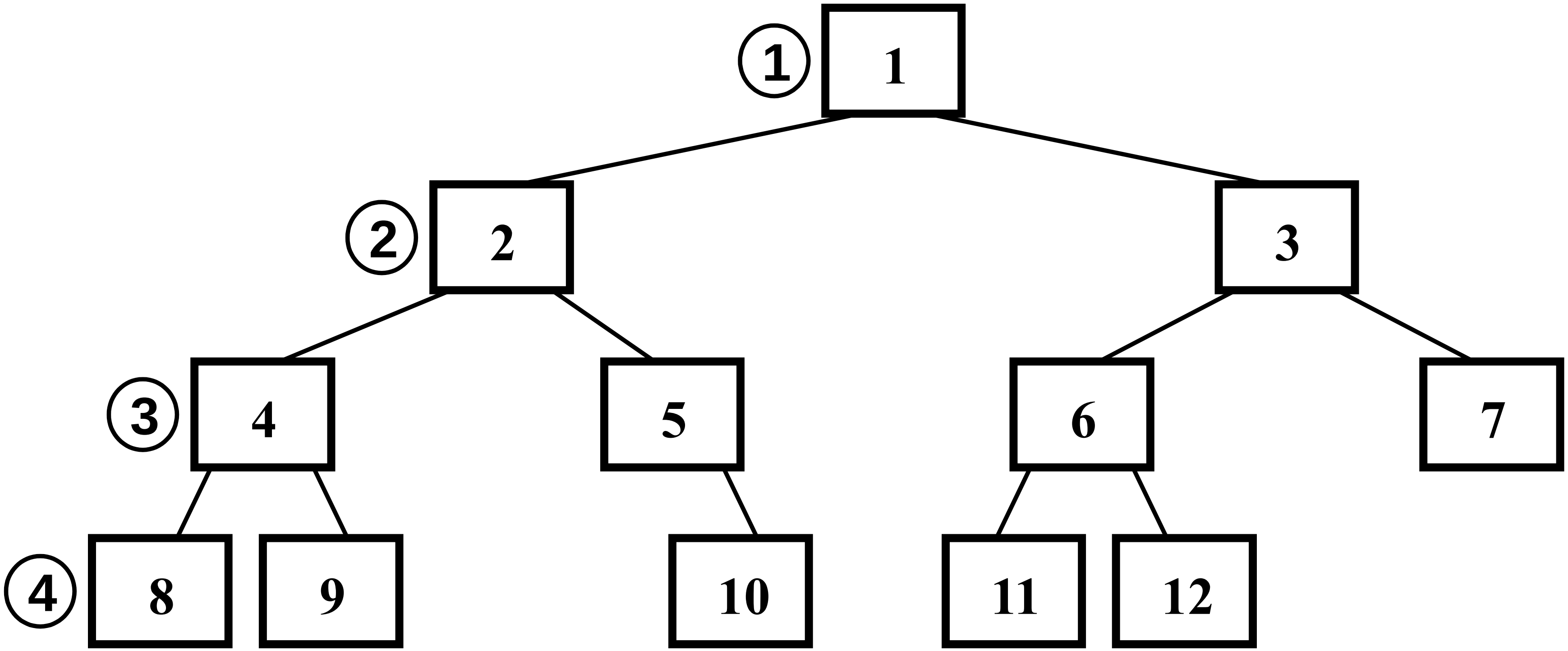
Postorder traversal

- if the tree is not empty
- postOrderTraverse(left child)
- postOrderTraverse(right child)
- visit the root

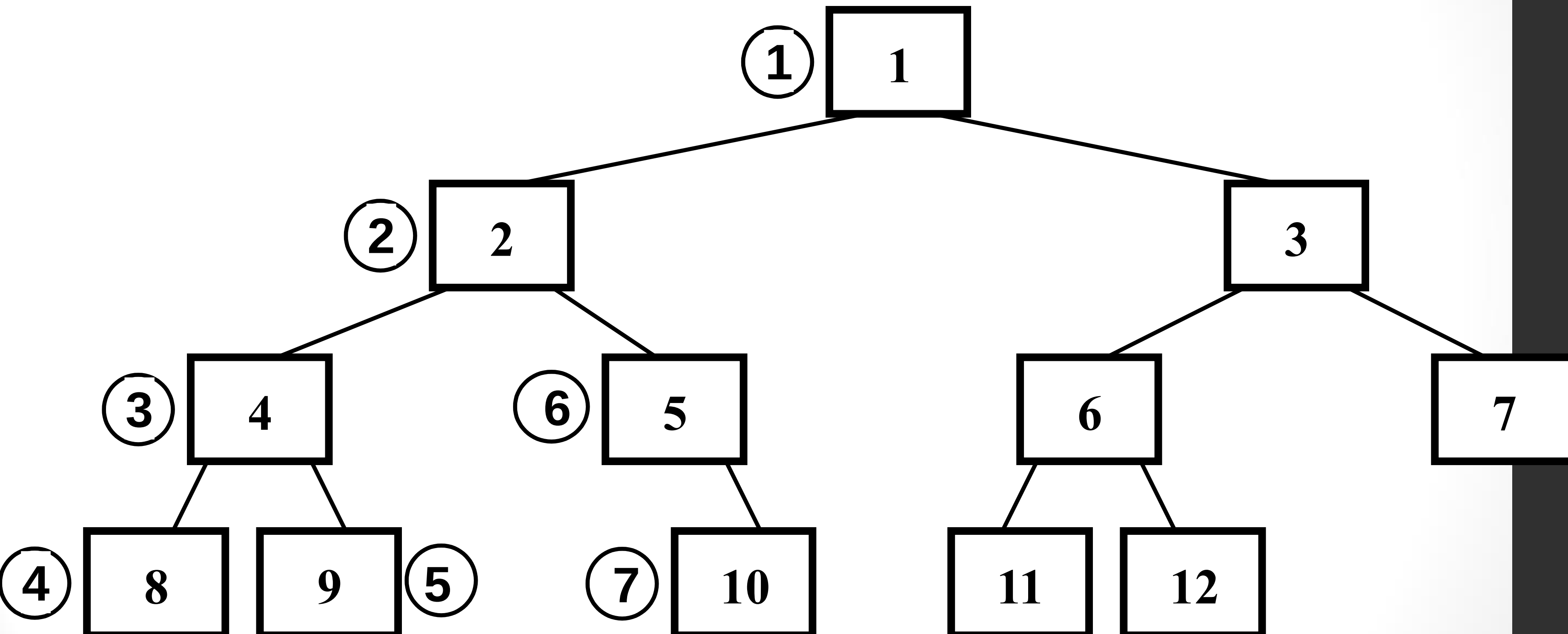
Sample Tree to Illustrate Tree Traversal



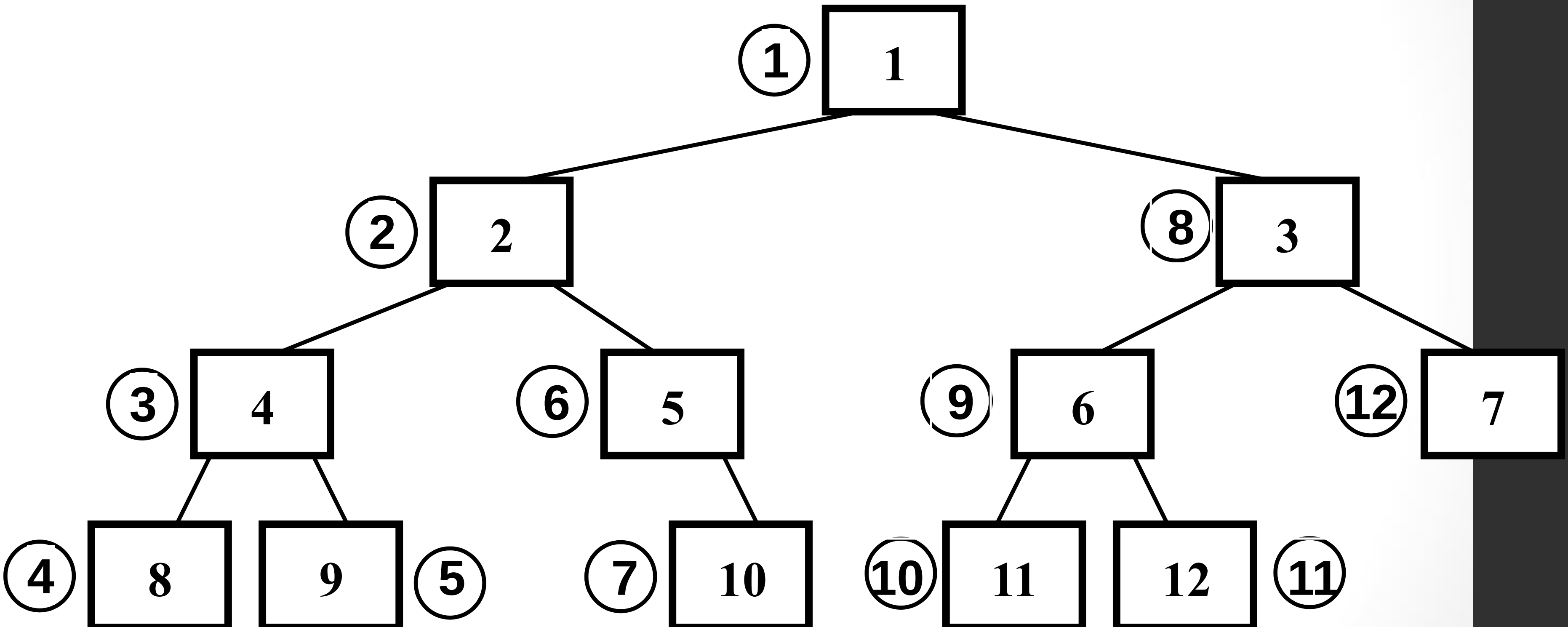
Tree after Four Nodes Visited in Preorder Traversal



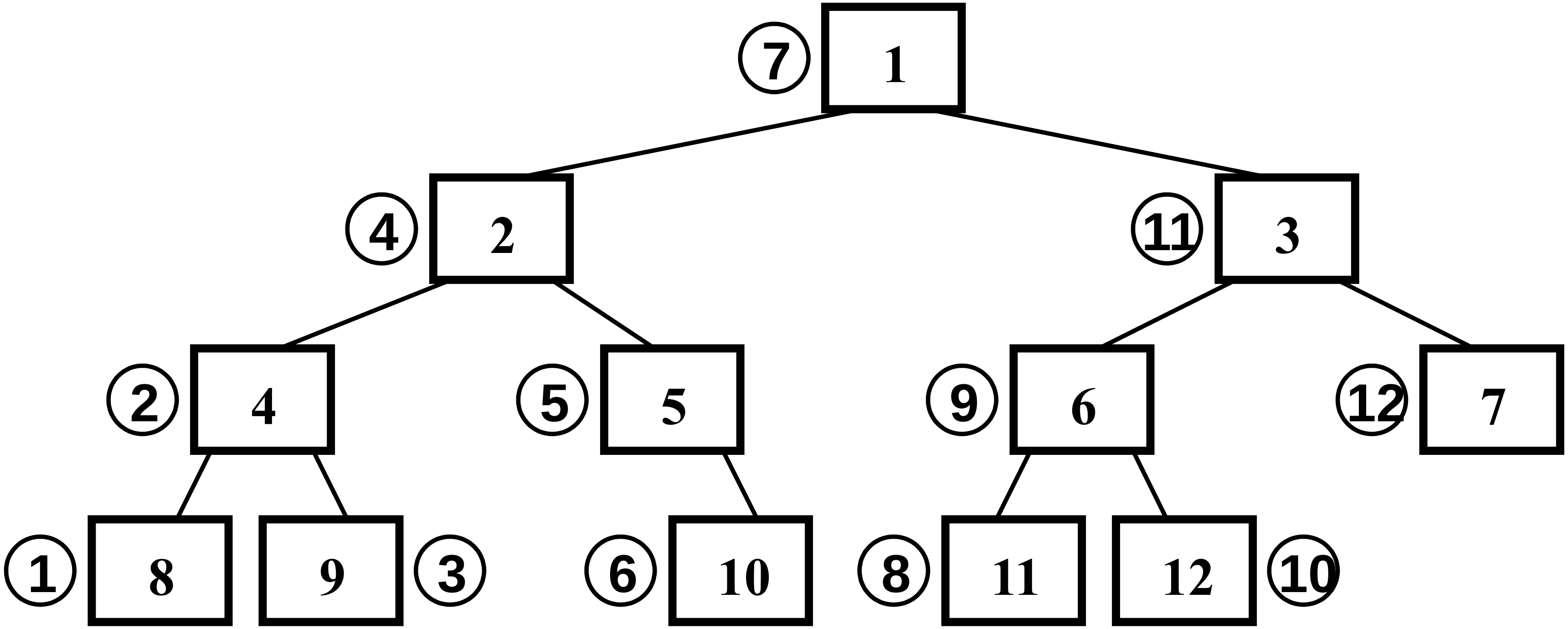
Tree after Left Subtree Visited Using Preorder Traversal



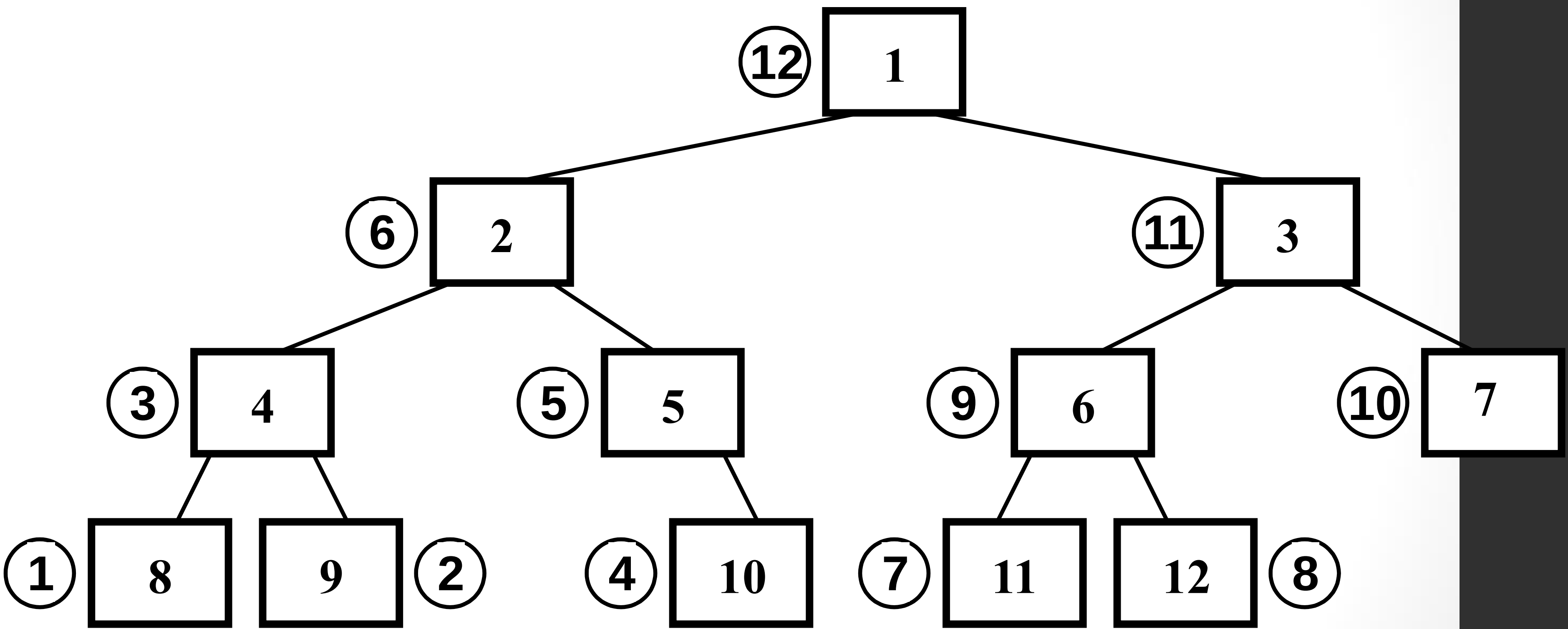
Tree after Completed Preorder Traversal



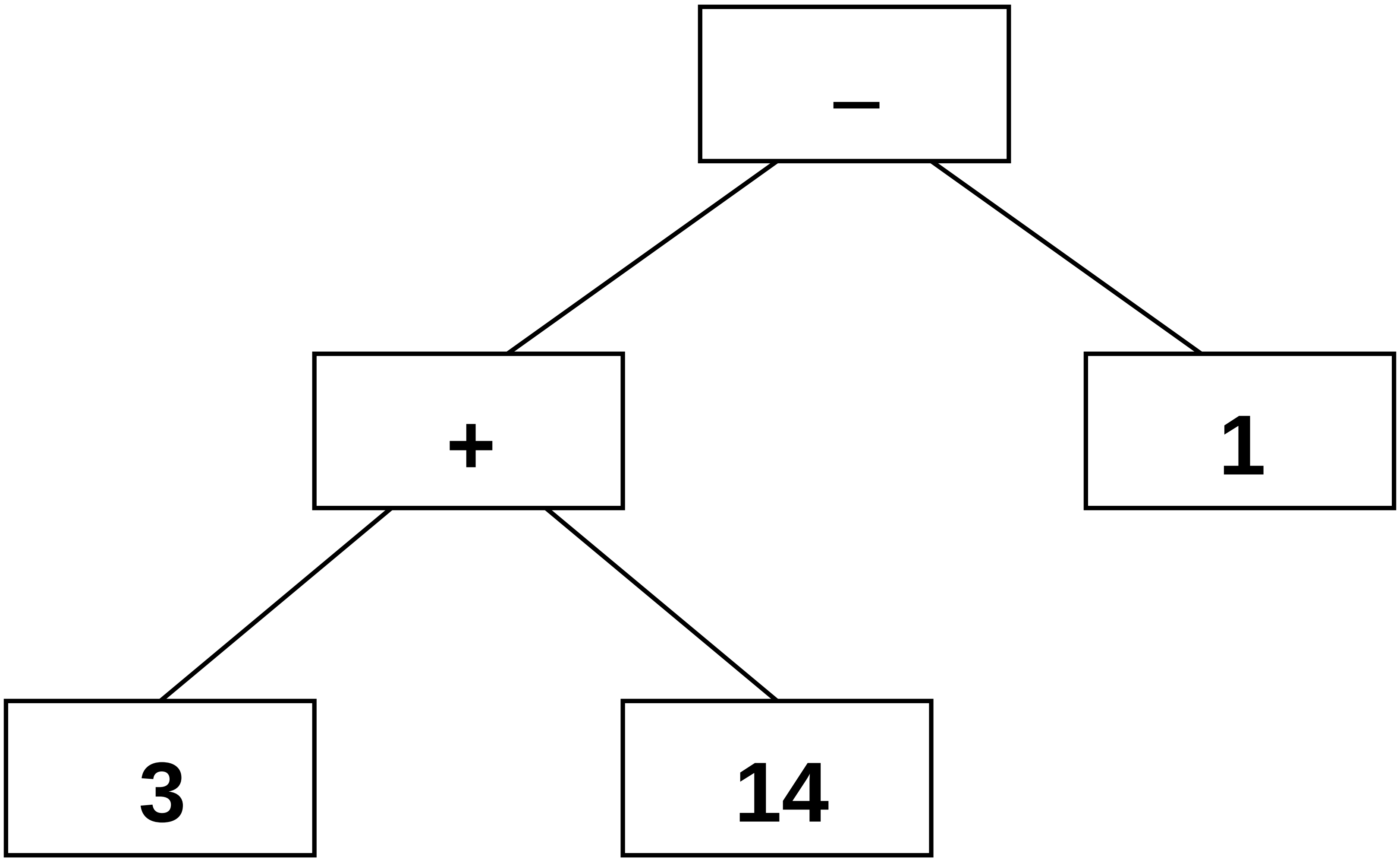
Tree visited using inorder traversal



Tree visited using postorder Traversal



Expression Tree for 3 + 7 *2 - 1



Binary Tree Traversals: preorder

```
typedef BinaryTree < int > btint;
typedef btint * btintp;
void preOrderTraverse(btintp bt)
{   if (!bt->isEmpty()) {  // if not empty
      // visit tree
      cout << bt->getData() << '\t';
     // traverse left child
      preOrderTraverse(bt->left());
      // traverse right child
      preOrderTraverse(bt->right());
   }
}
```

Inorder traversal

```
void inOrderTraverse(btintp bt)
{
  if (!bt->isEmpty()) {
    // traverse left child
    inOrderTraverse(bt->left());
    // visit tree
    cout << bt->getData() << '\t';
    // traverse right child
    inOrderTraverse(bt->right());
  }
}
```

For the following tree, answer the following

1) The number of internal nodes in the tree are

a) 2 b) 8 c) 6 d) none of the above

2) The height of the tree is

a) 4 b) 3 c) 2 d) none of the above

3) If a postorder traverse is used, what is the order of accessing the root?

a) 1 b) 14 c) 3 d) none of the above

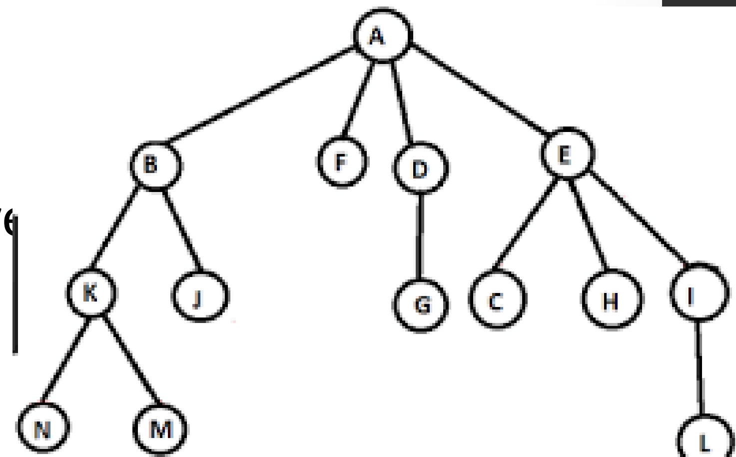
4) Is B node considered a tree?

a)True b) False

5) If a preorder traverse is used,

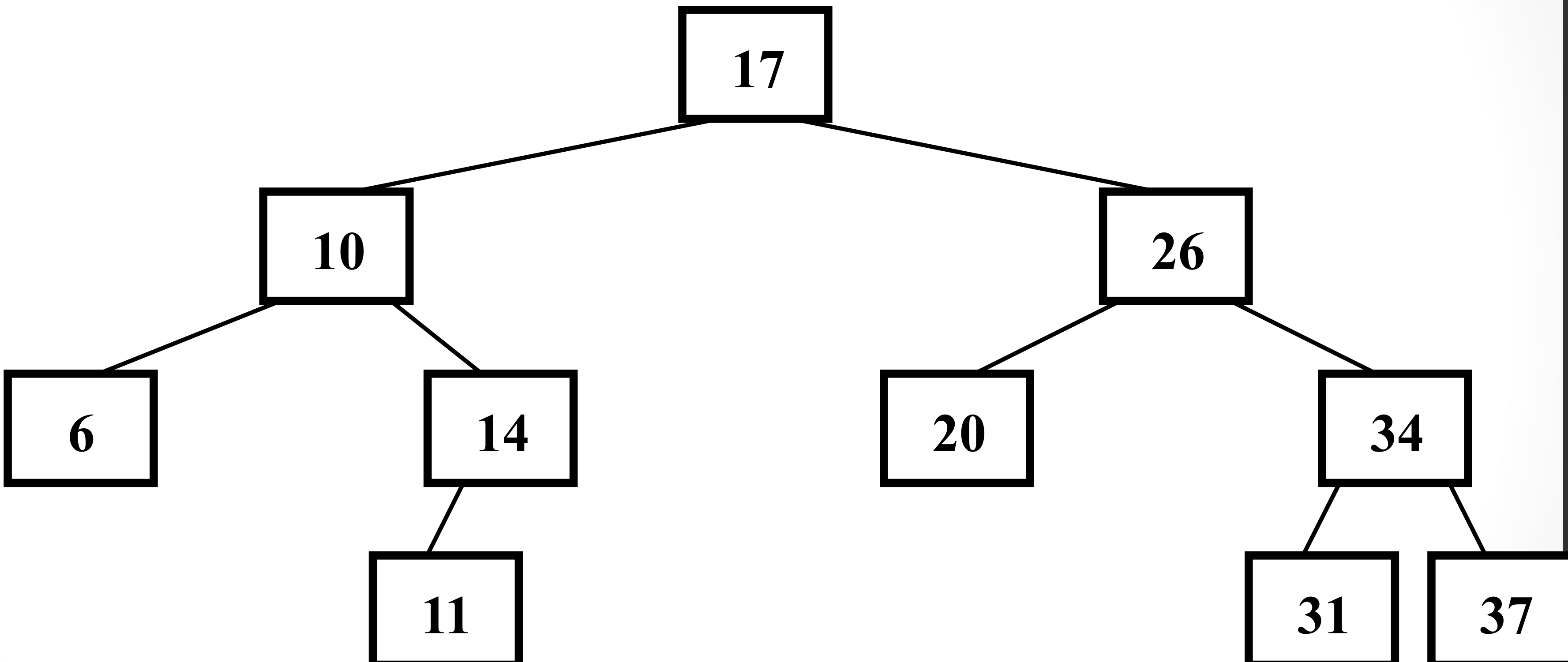
what is the order of accessing the root?

a) 1 b) 14 c) 3 d) none of the above



An N-ary Tree.

A Binary Search Tree



An Ordered Tree ADT

- BSTs are ordered
- BSTs provide for fast retrieval and insertion
- BSTs also support sequential processing of elements

Definition of BST

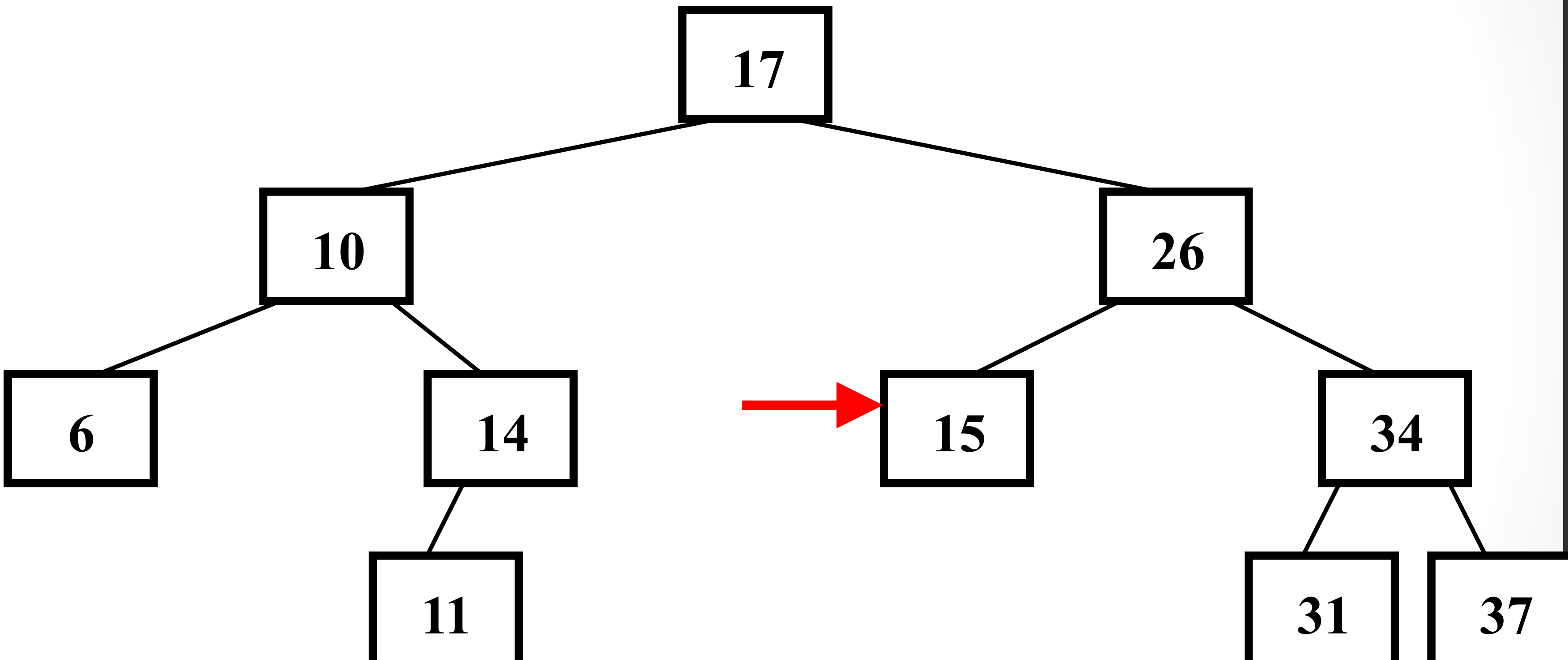
- A Binary Search Tree (BST) is either

  1. An empty tree, or

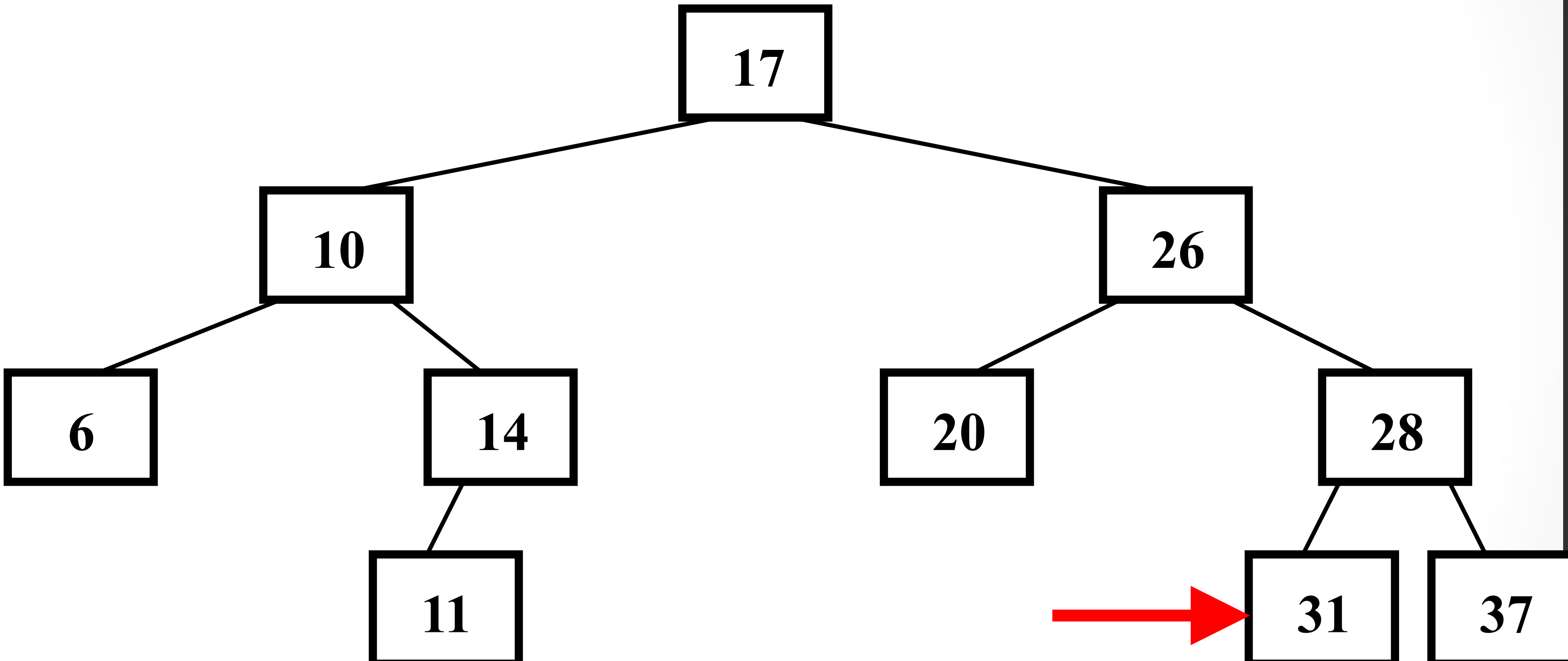
  2. A tree consisting of a node, called the root, and two children called left and right, each of which is also a BST. Each node contains a value such that the root is greater than all node values stored in its left subtree and less than all values stored in the right subtree.

- The BST **invariant**
  - The invariant is the ordering property
  - "less than goes left, greater than goes right."

Not a BST: invariant is violated



Not a BST: subtree is not a BST



Binary Search Tree ADT

- **Characteristics**

  A Binary Search Tree ADT T

  stores data of some type (btElementType)

  Obeys definition of BST (see earlier slide)

- **Prerequisites**

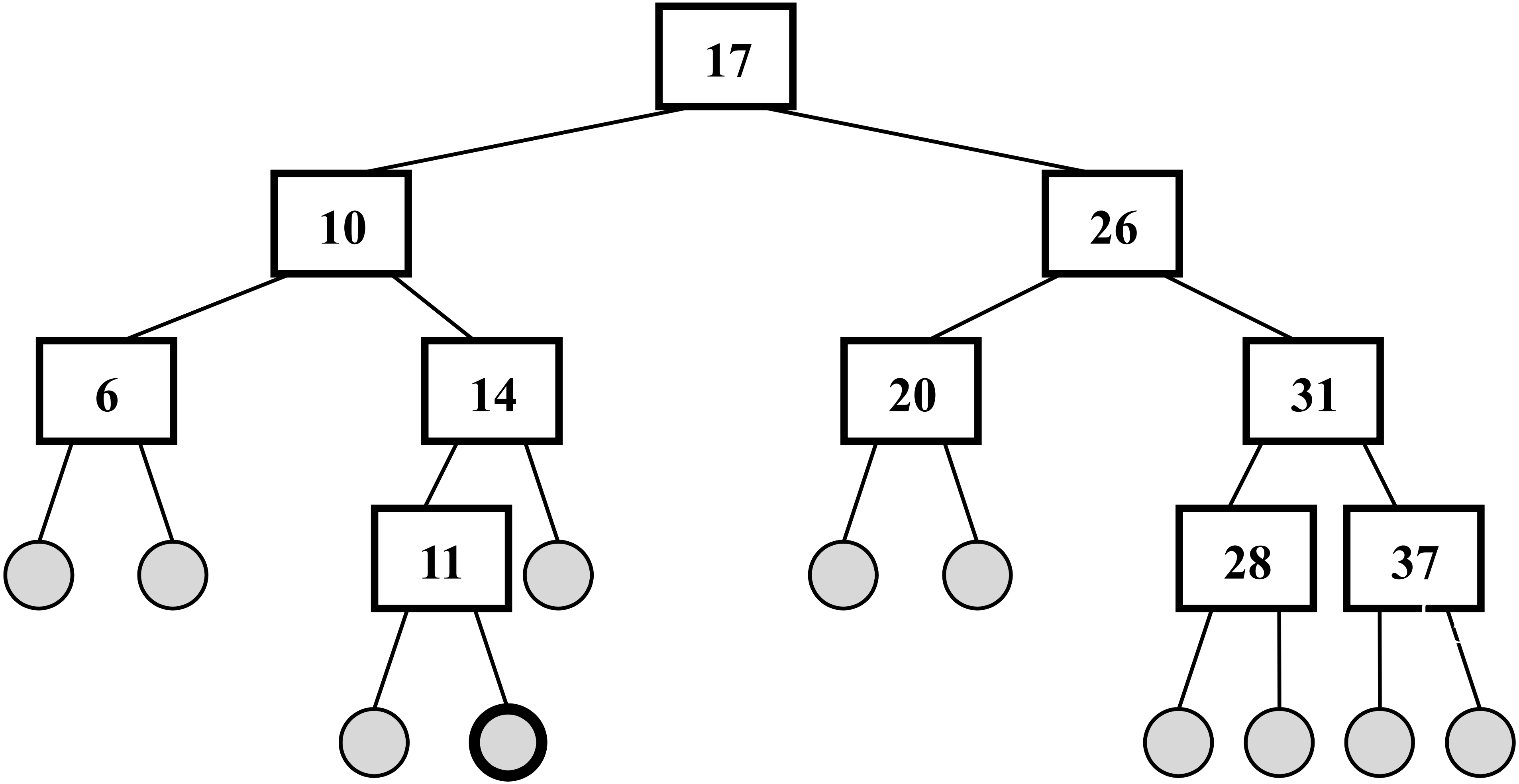
  The data type btElementType must

  implement the < and == operators. (operator overloading)

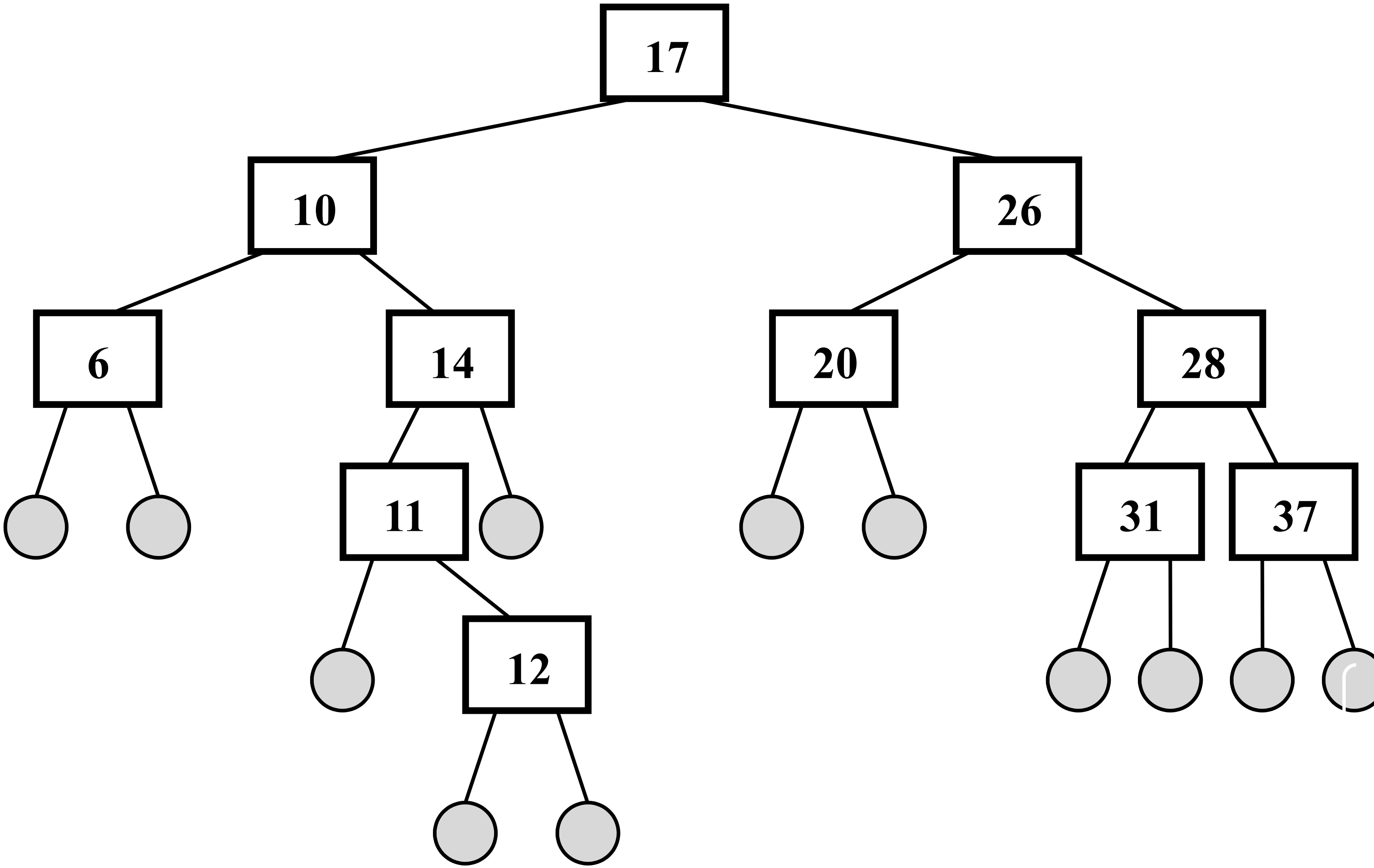
BST ADT Operations

- **isEmpty()   // check for empty BST**
- **getData()   // accessor**
- **insert()   // inserts new node**
- **retrieve()   // returns pointer to a BST**
- **left()   // returns left child**
- **right()   // returns right child**

Where 12 would be inserted

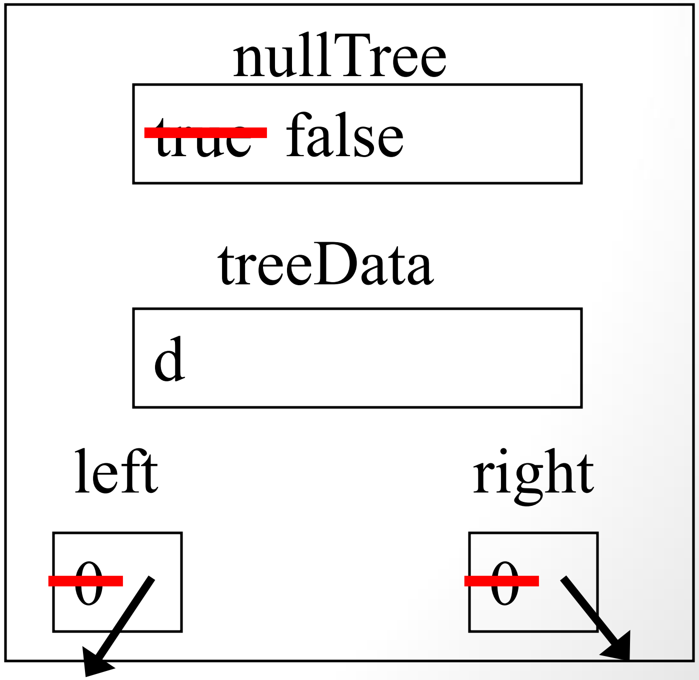


After 12 inserted



insert( )

```
template < class btElementType >
void BST < btElementType >
:: insert(const btElementType & d)
{
  if (nullTree) {
    nullTree = false;
    leftTree = new BST;
    rightTree = new BST;
    treeData = d;
  }
```



insert (if not empty)

```
else if (d == treeData) ; // do nothing -- it's already here!
else if (d < treeData)
    leftTree->insert(d);  // insert in left subtree
else
    rightTree->insert(d);  // insert in right subtree
}
```

retrieve( )

- **BinaryTree T.retrieve(btElementType d)**
- *Precondition:* T meets the BST invariant.
- *Postcondition:* T meets the BST invariant.
- *Returns:* if T contains a node with data d,
  then T.retrieve(d).getData() == d;
  otherwise, T.retrieve(d).isEmpty().

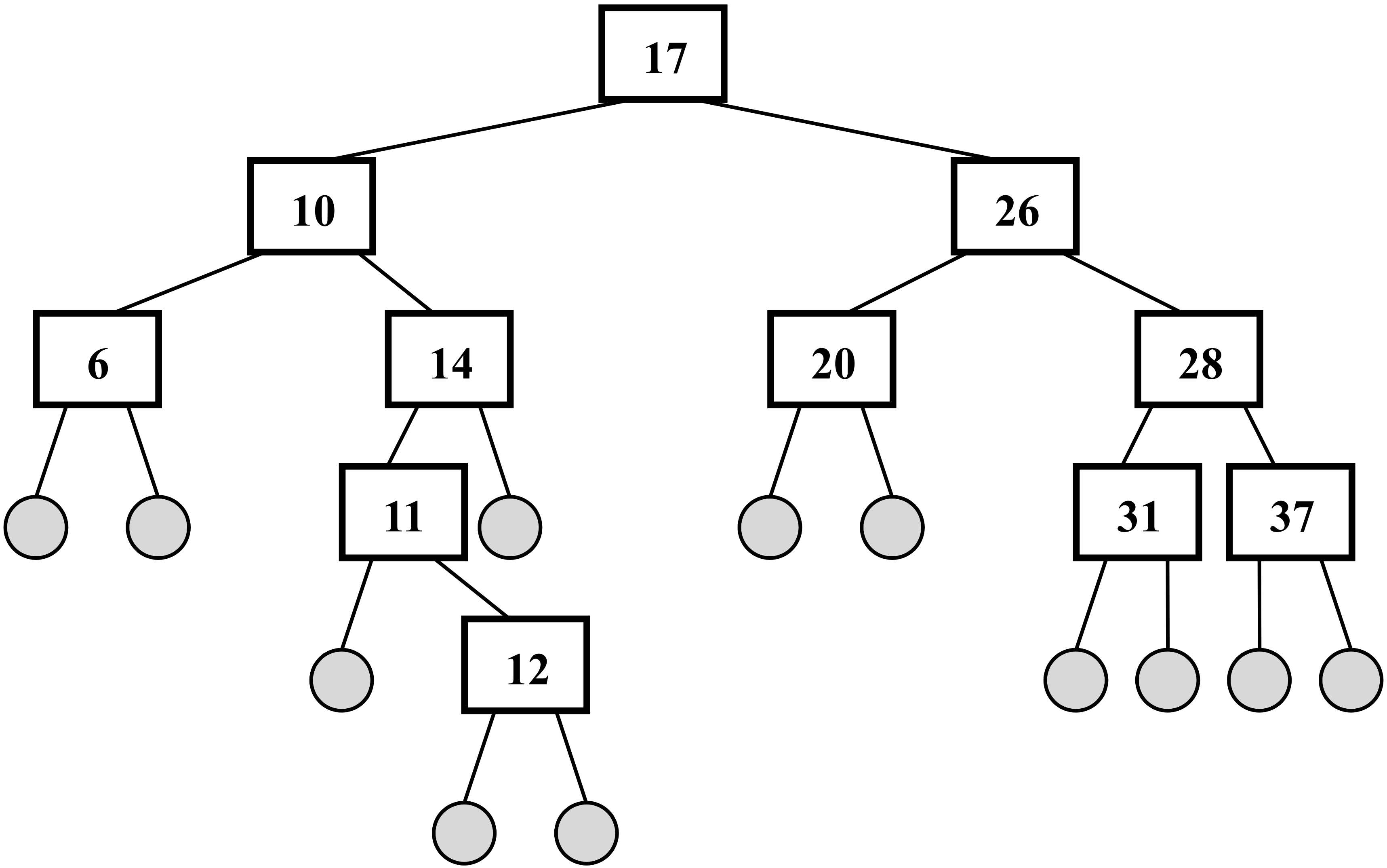
retrieve( )

```
template < class btElementType > BST < btElementType > *
BST < btElementType > :: retrieve(const btElementType & d)
{
  if (nullTree || d == treeData)
    // return pointer to tree for which retrieve was called
    return this;
  else if (d < treeData)
    return leftTree->retrieve(d);  // recurse left
  else
    return rightTree->retrieve(d);  // recurse right
}
```

Client of BST

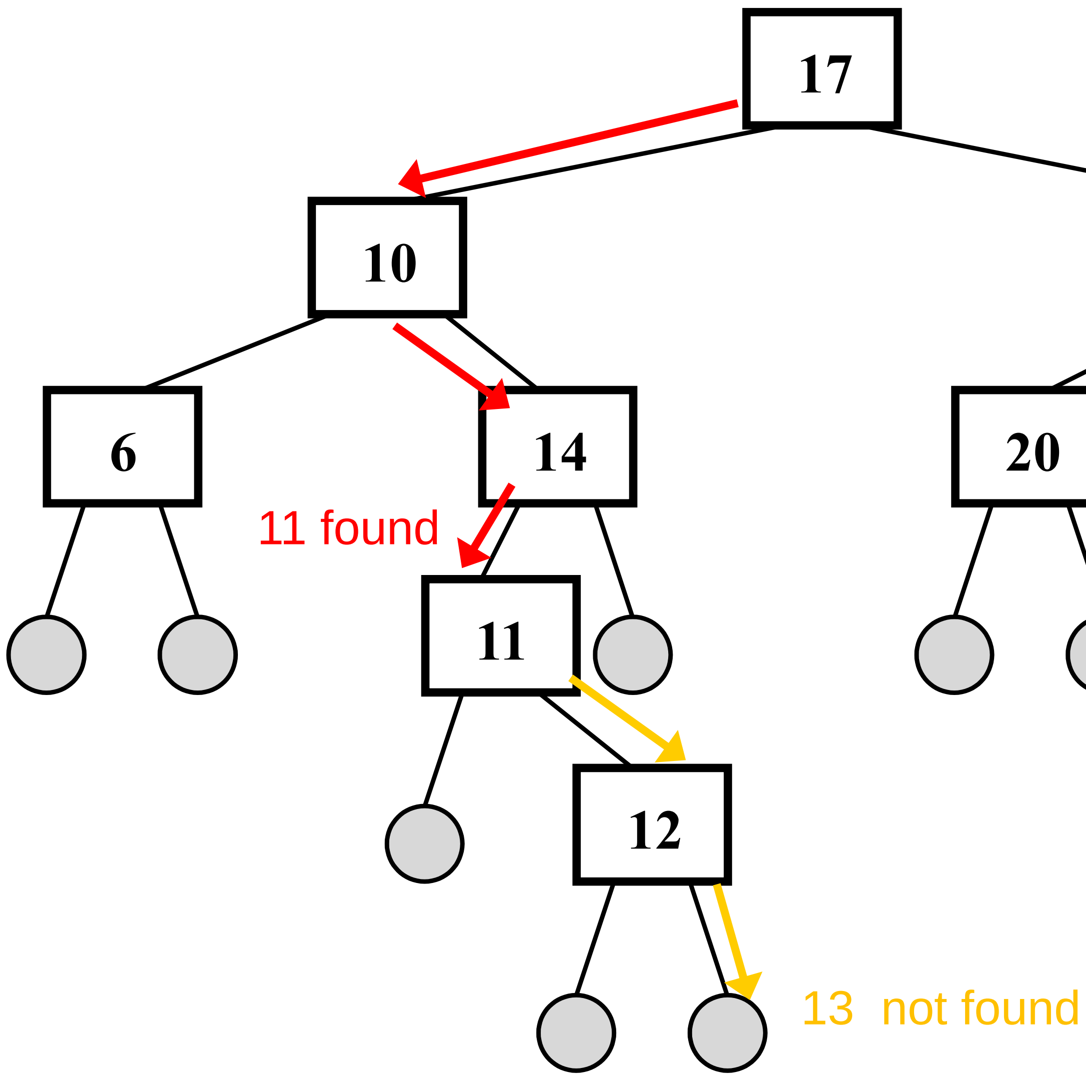
```
int main()
{
  typedef BST < int > intBST;
  typedef intBST * intBSTPtr;

  intBSTPtr b(new intBST);
  b->insert(17);  b->insert(10);  b->insert(26);
  b->insert(6);  b->insert(14); b->insert(20);
  b->insert(28);  b->insert(11); b->insert(31);
  b->insert(37); b->insert(12);
```

BST Result



Retrieval

```
// is 11 in the tree?
  intBSTPtr get11(b->retrieve(11));
  if (get11->isEmpty())
    cout << "11 not found.\n";
  else  cout << "11 found.\n";
  // is 13 in the tree?
  intBSTPtr get13(b->retrieve(13));
  if (get13->isEmpty())
    cout << "13 not found.\n";
  else cout << "13 found.\n";
  return 0;
}
```


17
10
26
6
14
20
28
11 found
11
31
37
12
13 not found

Reuse through Inheritance

- Important OOP language features

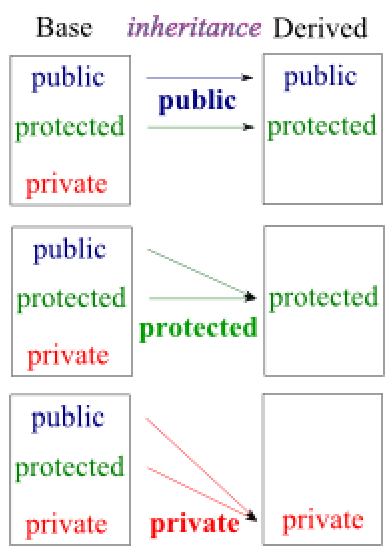
1. **Encapsulation**: for abstractions like classes, types, objects.
2. **Inheritance**: abstractions can be reused
3. **Polymorphism**: statements that use more than one abstraction, with the language choosing the right one while the program is running.

“is-a” Relations

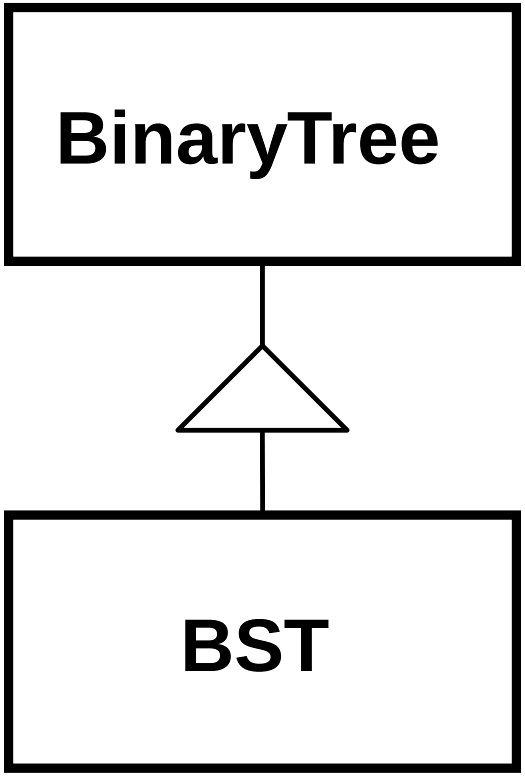
- Defining one abstraction in terms of another.
- The Binary Tree ADT is a general class of which binary search trees are one type.
- Therefore, if we define a Binary Tree ADT we should be able to define a BST as a special case which inherits the characteristics of the Binary Tree ADT.
- A BST “is a” Binary Tree, with special features
- So, the BST is a derived class of the Binary Tree base class

Inheritance Terminology

- Base class - the one from which others inherit
- Derived class - one which inherits from others
- So, the BST is a derived class of the Binary Tree base class.



Inheritance Diagram



Implementing inheritance in C++

- Base classes are designed so that they can be inherited from.
- One crucial aspect of inheritance is that the derived class may wish to implement member functions of the base case a little differently.
- Good examples are insertion for BST (which must follow the ordering principle) and constructors.

Protected Members

- We have used public and private sections of our class definitions. There is also ‘protected’.

- Protected means that the members are hidden from access (private) for everything except derived classes, which have public access rights.

- Derived classes can use the data members of base classes directly, without accessors

Public Inheritance

- Inheritance notation is done like this

  class BST : public BinaryTree

- public means that BST client code must be able to access the public members of the base class. (public inheritance)

- It means that derived functions will only redefine member functions that they must (like insert() for BST). All other functions will be the public ones in the base class.

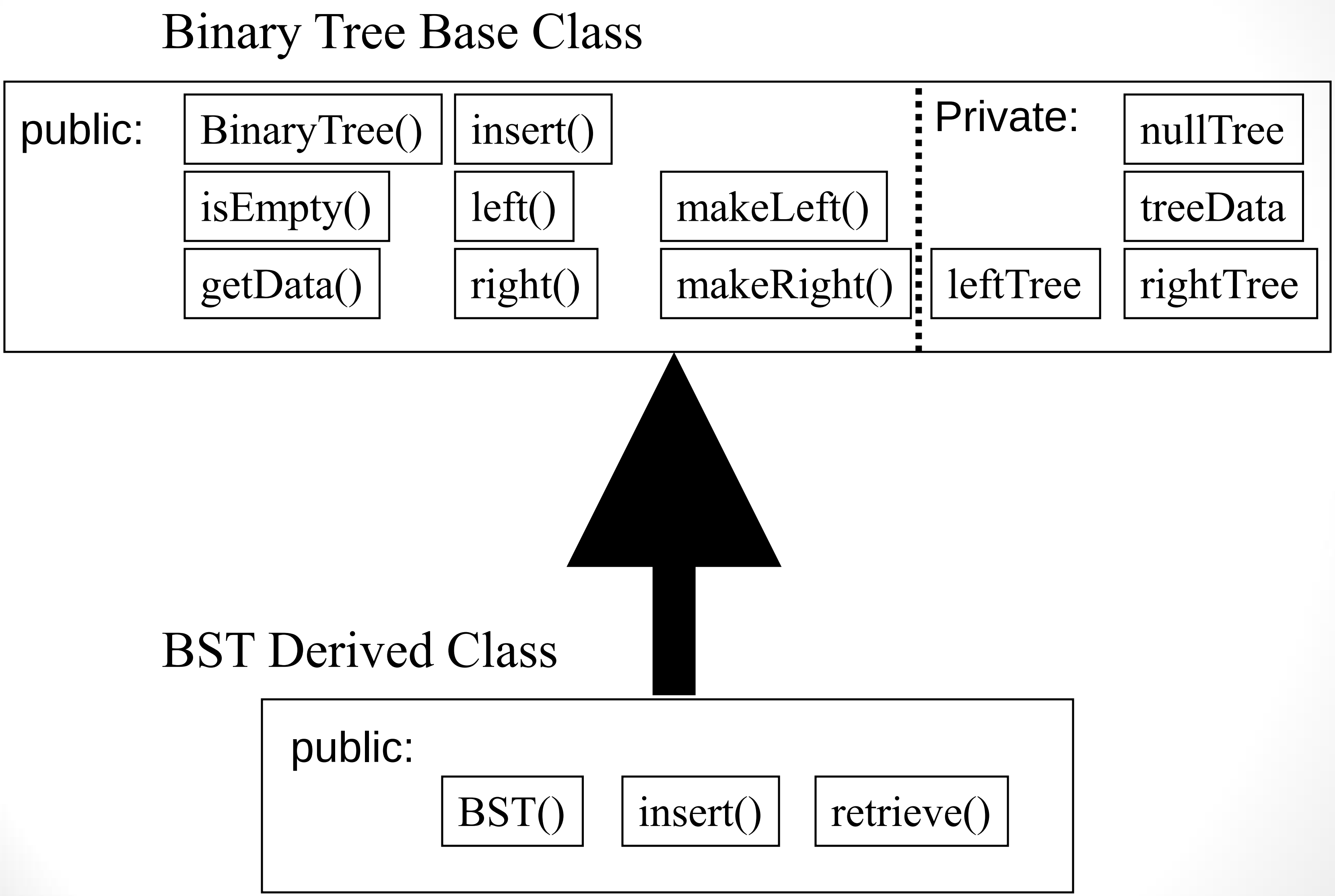
BST Inherited Class

```
template < class btElementType >
class BST : public BinaryTree < btElementType > {
public:
  BST();
  void insert(const btElementType & d);
  BinaryTree < btElementType > *
        retrieve(const btElementType & d);
};
```

Why so short?

- The BST derived class definition was so short because it only needed to declare member functions that represented changes from the public functions of the base class.

- All other functions come from the already defined base class.


Binary Tree Base Class
public:
BinaryTree()
insert()
isEmpty()
left()
makeLeft()
getData()
right()
makeRight()
Private:
nullTree
treeData
leftTree
rightTree
BST Derived Class
public:
BST()
insert()
retrieve()

Derived class constructors

- To construct a BST we must first call the Binary Tree constructor (the base class)
- Once this has been called, then the body of the BST constructor can add additional things on to it.
- In this case, we have nothing to add (see next slide)

Constructor (inheritance)

```
template < class btElementType >
BST < btElementType >
:: BST() : BinaryTree < btElementType >()
{
}
```

Chapter Summary

- Tree represent data in a hierarchical manner.
- In a Binary Tree, each node has a left and a right child.
- Expression Trees are a Binary Tree that can represent arithmetic expressions.
- The data in a tree can be visited using in-order, pre-order, and post-order traversals.
- Function pointers can be used to pass one function to another as an argument.
- The Binary Search Tree is an ordered tree ADT.

Chapter Summary

- Binary Search Trees support efficient retrieval by key
- Inheritance can be used to model is-a relations.
- Inheritance can be implemented in C++ through base and derived classes.
- Inheritance supports code reuse.
- Binary Search Tree can have bad performance if they're unbalanced, but very good performance when balanced.

Using Function Pointers

- We are used to sending pointers to variables
- Anything that has an address has a pointer
- Functions are addressable
- Therefore we can send functions into other functions by sending in their pointers
- Similarly, we can call functions by dereferencing these pointers

Visit function

```
void visit(btintp bt)
{
  cout << bt->getData() << '\t';
}
```

preorder traversal w/ *

```
typedef BinaryTree < int > btint;
typedef btint * btintp;  // pointer to integer binary tree
void preOrderTraverse(btintp bt, void visit(btintp))
{  if (!bt->isEmpty()) {
     (* visit)(bt);    // visit tree
     // traverse left child
      preOrderTraverse(bt->left(), visit);
      // traverse right child
      preOrderTraverse(bt->right(), visit);
   }
}
```

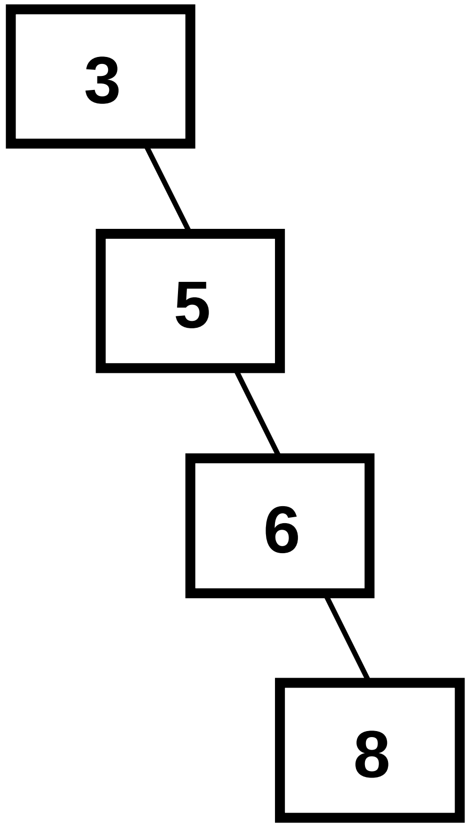
Inorder traversal w/*

```
void inOrderTraverse(btintp bt, void visit(btintp))
{  if (!bt->isEmpty()) {
     // traverse left child
     inOrderTraverse(bt->left(), visit);
     (* visit)(bt);       // visit tree
     //traverse right child
     inOrderTraverse(bt->right(), visit);
   }
}
```

Performance of Binary Trees

- Shape and balance are very important
- Short and wide trees are better than long and narrow ones.
- The depth of the tree is the main consideration in all traversal routines.
- Traversals are used by insertion and retrieval functions which must first look up a location in the tree.
- Examples using the preorder traversal

A Tree That is Expensive to Process



An Efficient Tree