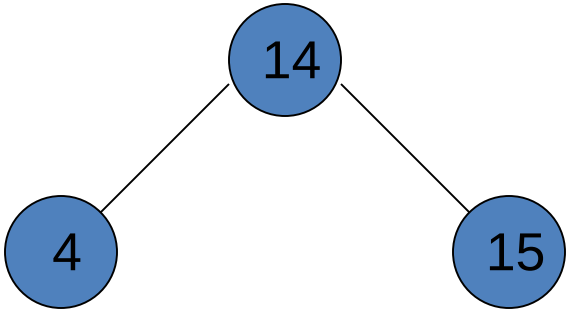
Data Structures & Algorithms

Week 10: Binary Search Tree

Traversing a Binary Tree

- Ways to print a 3 node tree:



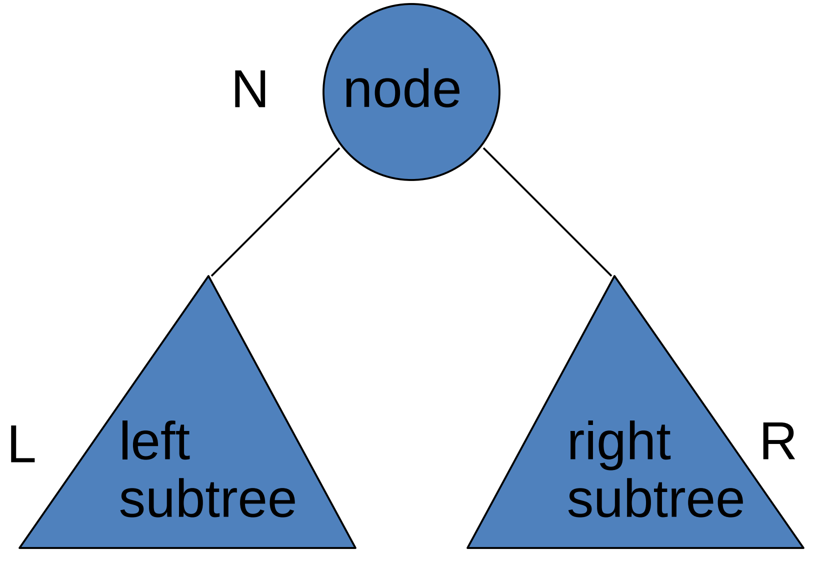
(4, 14, 15), (4,15,14)

(14,4,15), (14,15,4)

(15,4,14), (15,14,4)

Traversing a Binary Tree

- In case of the general binary tree:



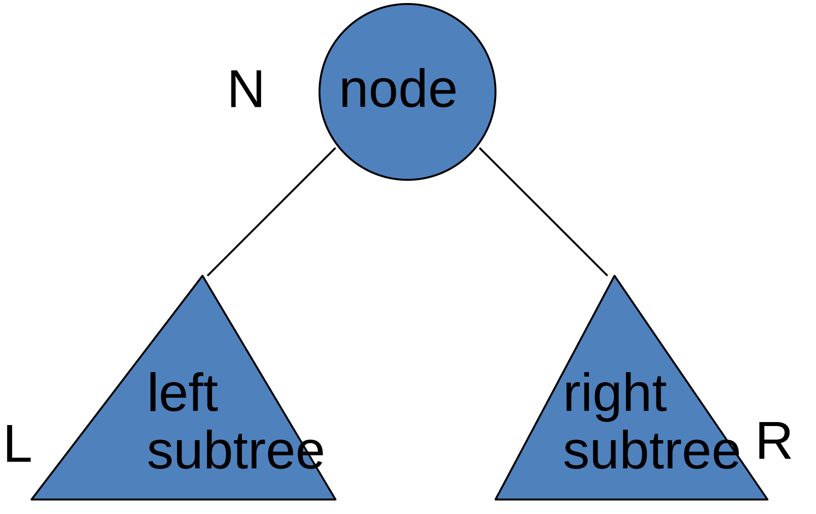
(L,N,R), (L,R,N)

(N,L,R), (N,R,L)

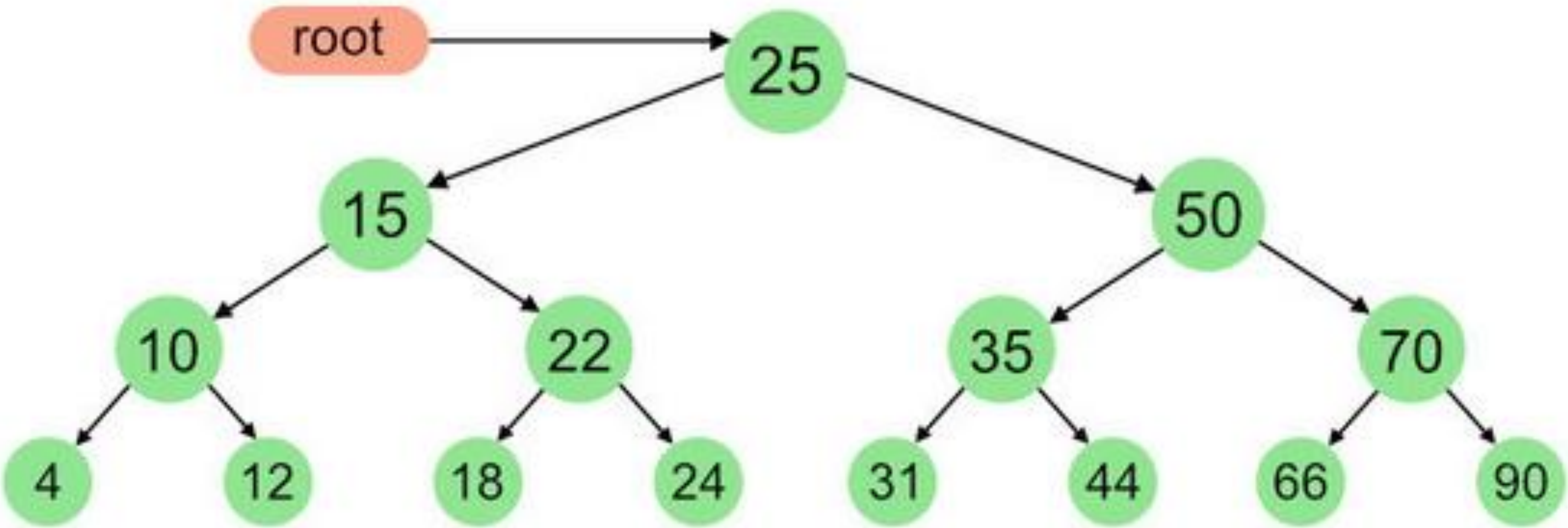
(R,L,N), (R,N,L)

Traversing a Binary Tree

- Three common ways

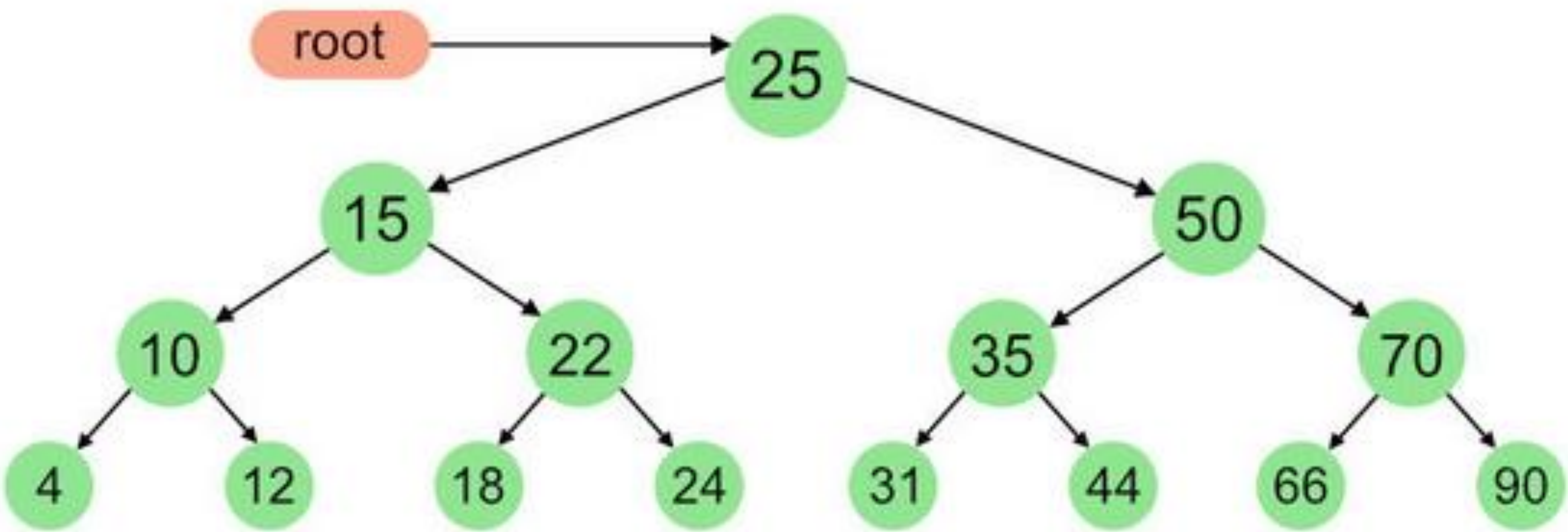


Preorder:	(N,L,R)
Inorder:	(L,N,R)
Postorder:	(L,R,N)


root
25
15
50
10
22
35
70
4
12
18
24
31
44
66
90

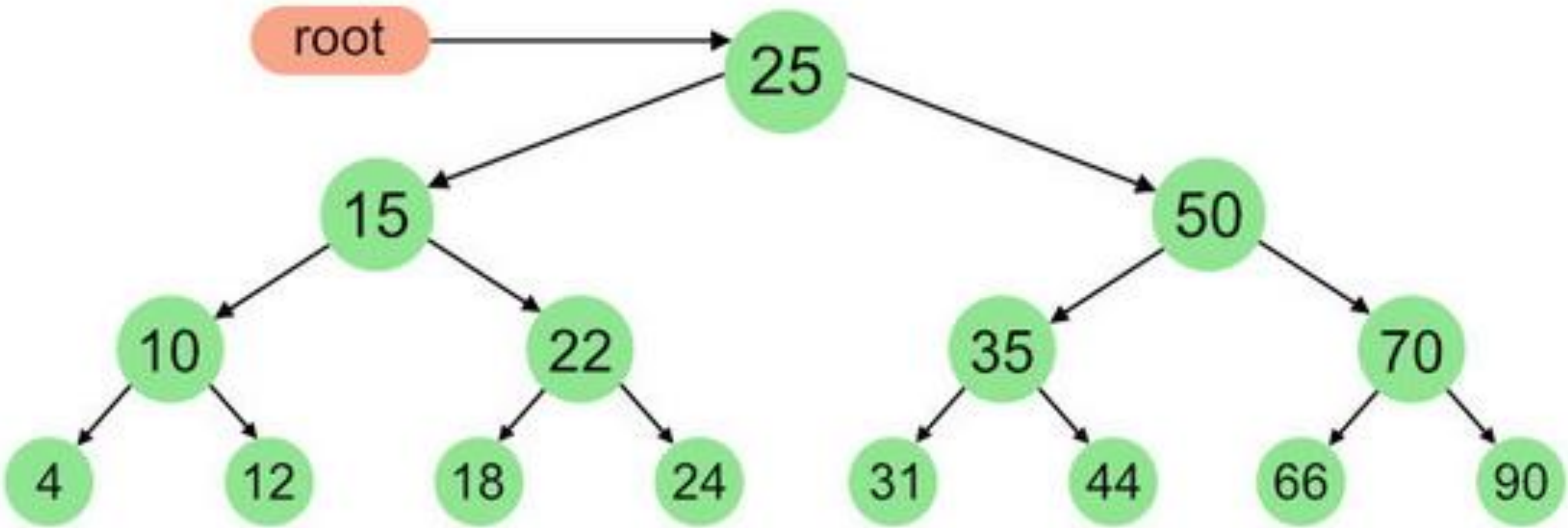
A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90



A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Traversing a Binary Tree

```
void preorder(BSTNode* BSTNode)
{
    if( BSTNode != NULL )
    {
        cout << BSTNode->getInfo()<<" ";
        preorder(BSTNode->getLeft());
        preorder(BSTNode->getRight());
    }
}
```

Traversing a Binary Tree

```
void inorder(BSTNode* BSTNode)
{
    if( BSTNode != NULL )
    {
        inorder(BSTNode->getLeft());
        cout << BSTNode->getInfo()<<" ";
        inorder(BSTNode->getRight());
    }
}
```

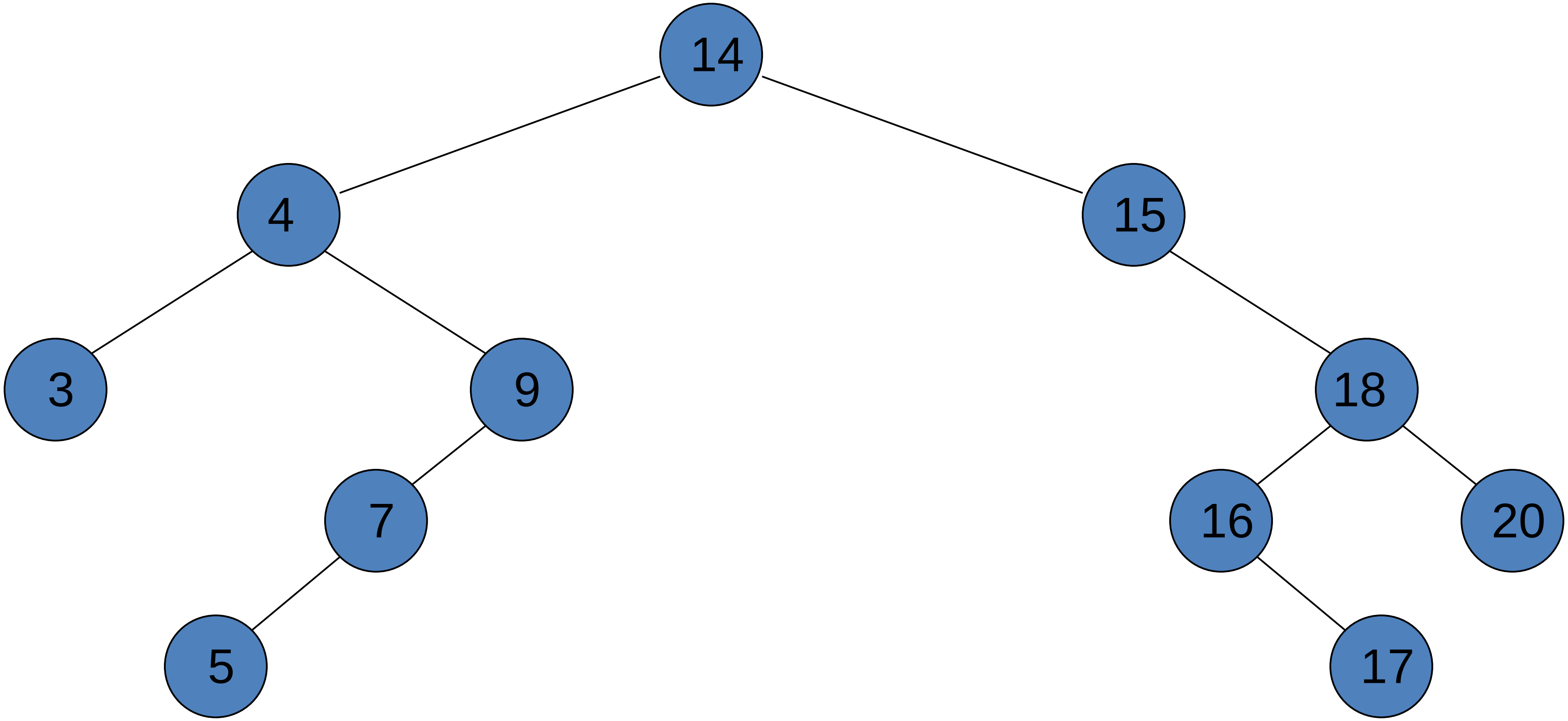
Traversing a Binary Tree

```
void postorder(BSTNode* BSTNode)
{
    if( BSTNode != NULL )
    {
        postorder(BSTNode->getLeft());
        postorder(BSTNode->getRight());
        cout << BSTNode->getInfo()<<" ";
    }
}
```

Traversing a Binary Tree

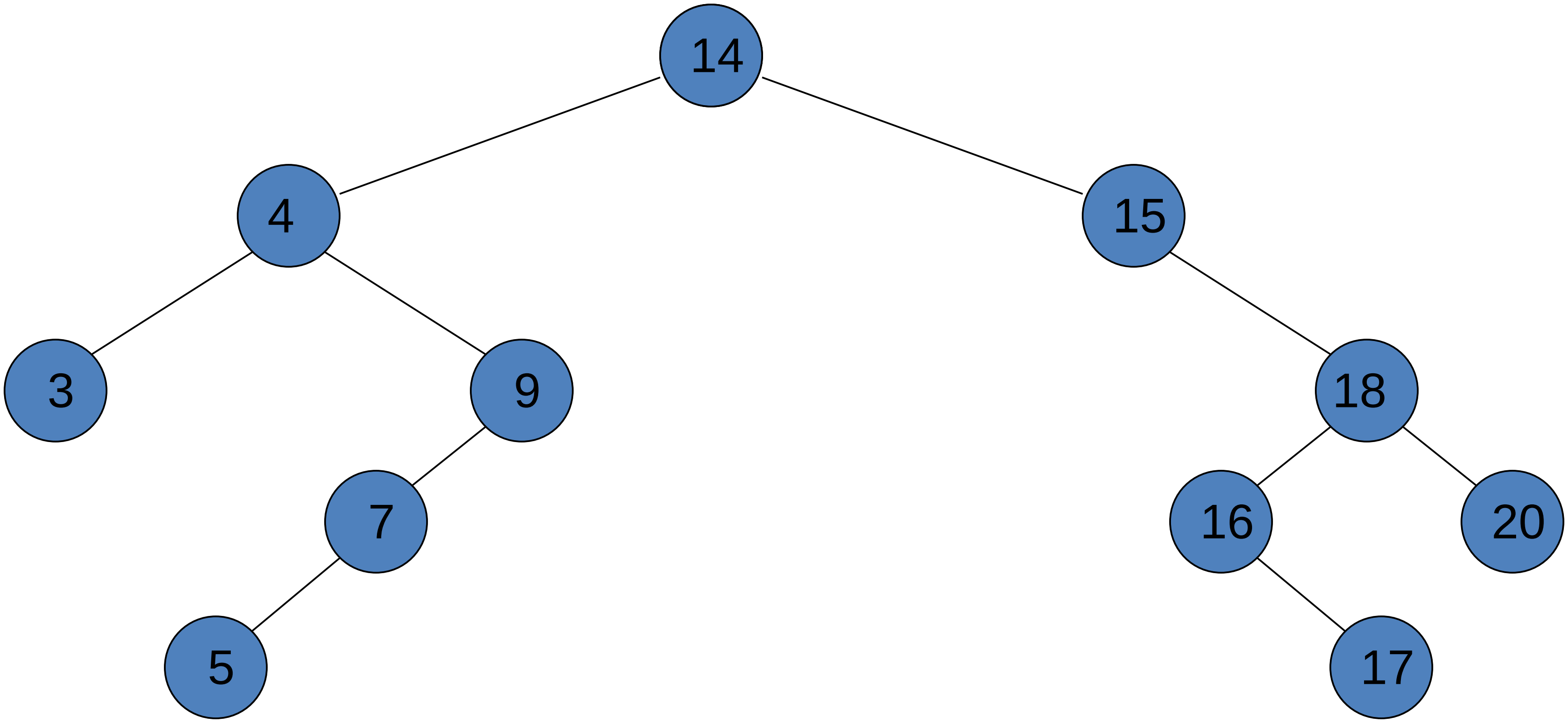
```
cout << "inorder: ";   preorder( root);
cout << "inorder: ";   inorder( root );
cout << "postorder: "; postorder( root );
```

Traversing a Binary Tree



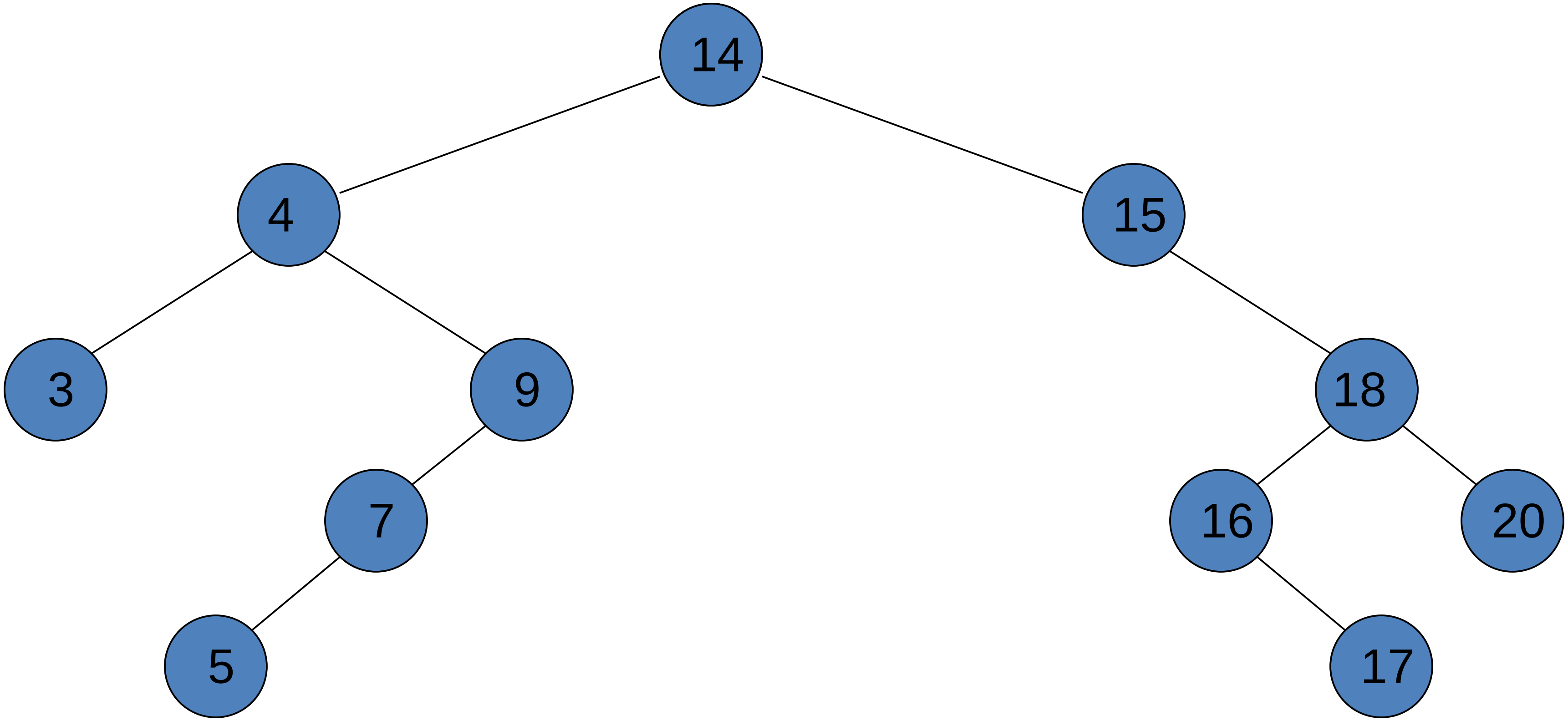
Preorder: 14 4 3 9 7 5 15 18 16 17 20

Traversing a Binary Tree



Inorder: 3 4 5 7 9 14 15 16 17 18 20

Traversing a Binary Tree



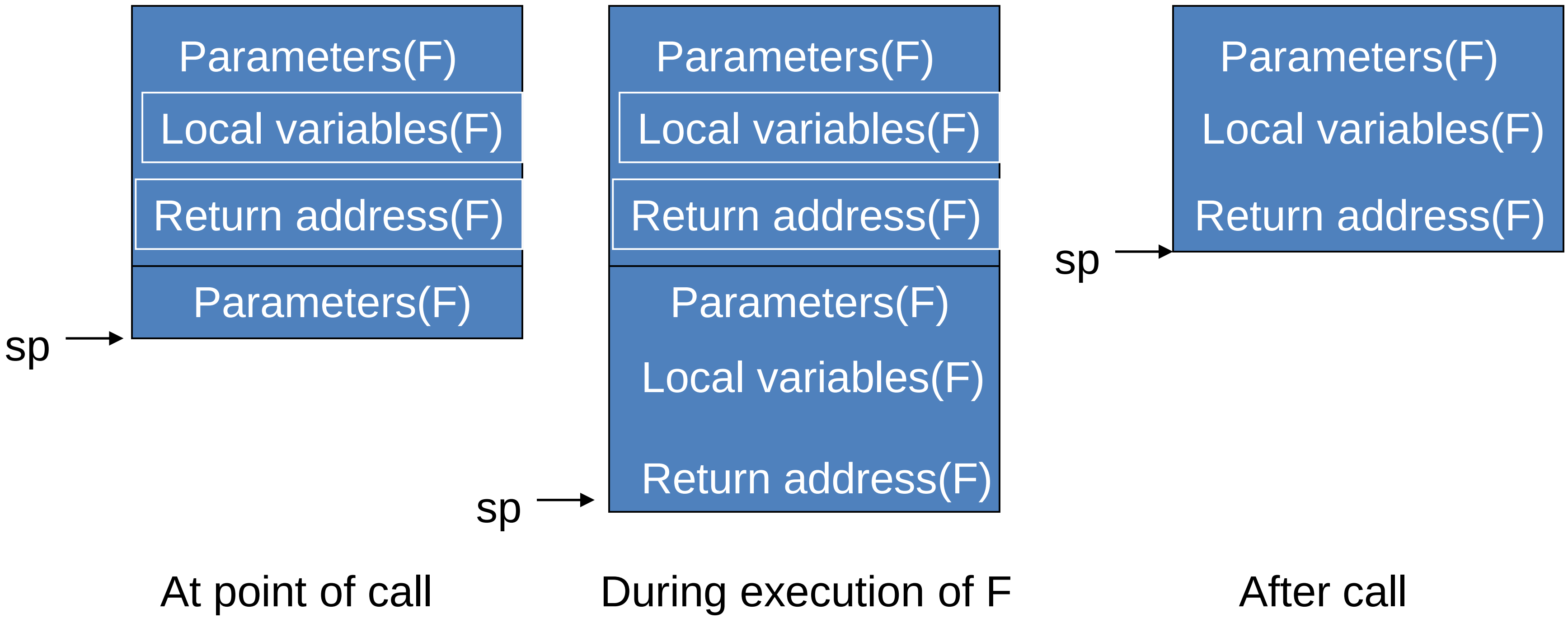
Postorder: 3 5 7 9 4 17 16 20 18 15 14

Recursive Call

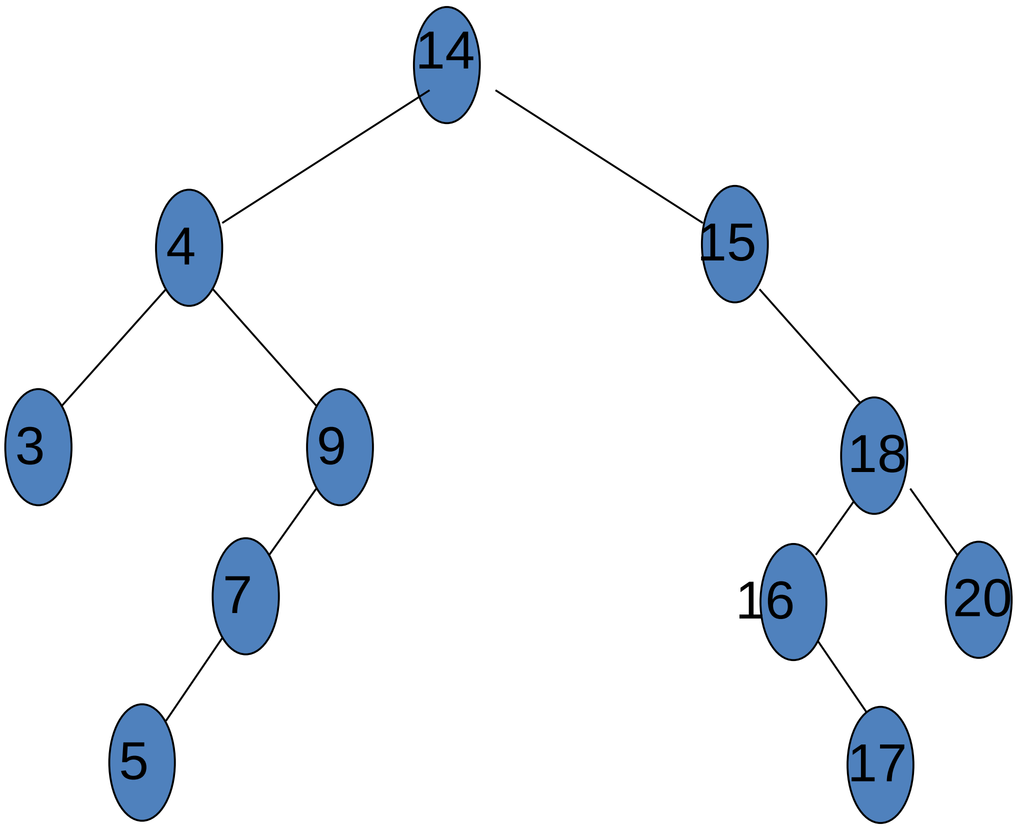
- Recall that a stack is used during function calls.
- The caller function places the arguments on the stack and passes control to the called function.
- Local variables are allocated storage on the call stack.
- Calling a function itself makes no difference as far as the call stack is concerned.

Stack Layout during a call

- Here is stack layout when function F calls function F (recursively):

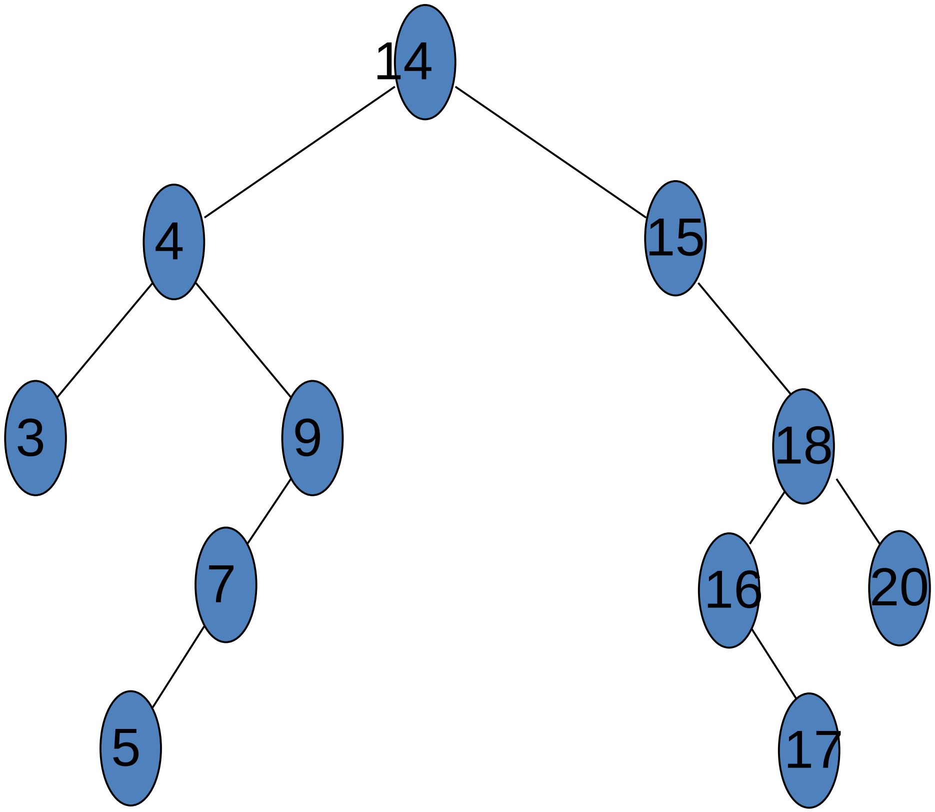


Recursion: preorder



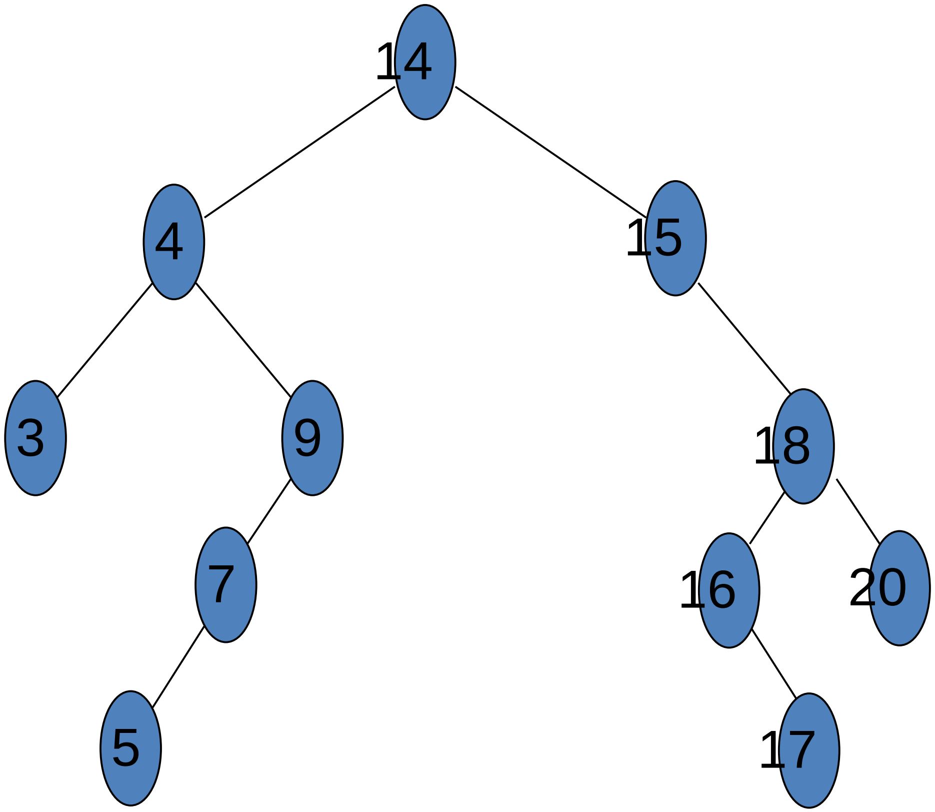

```
preorder(14)
14
..preorder(4)
4
....preorder(3)
3
......preorder(null)
......preorder(null)
....preorder(9)
9
......preorder(7)
7
........preorder(5)
5
..........preorder(null)
..........preorder(null)
........preorder(null)
......preorder(null)
```

Recursion: preorder



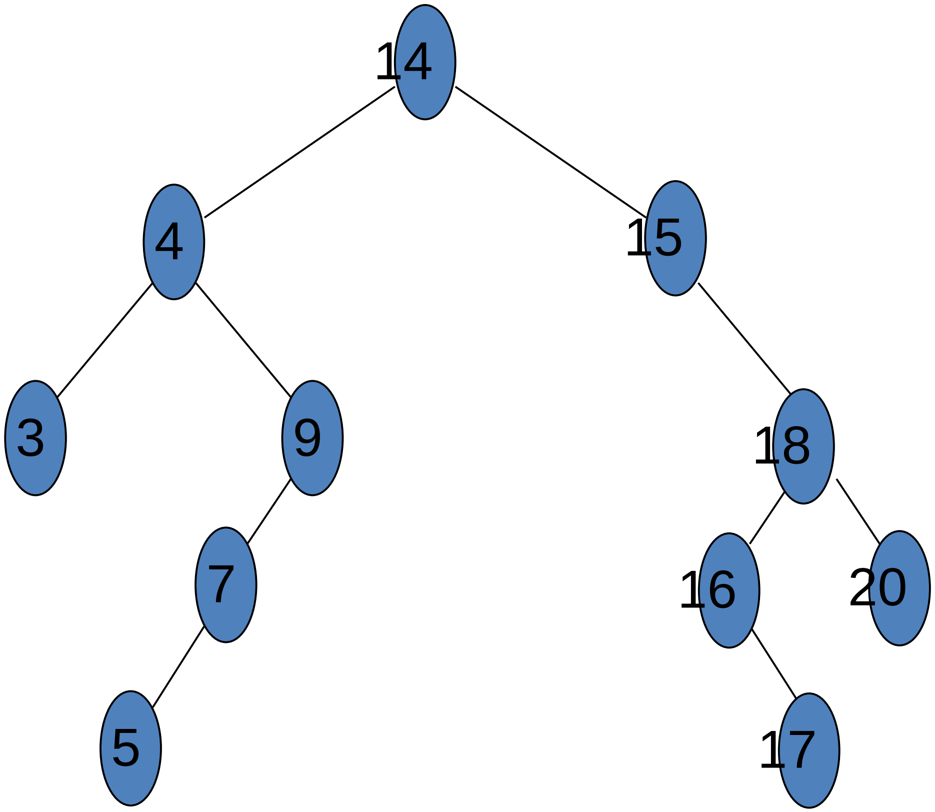

```
..preorder(15)
15
....preorder(null)
....preorder(18)
18
......preorder(16)
16
........preorder(null)
........preorder(17)
17
..........preorder(null)
..........preorder(null)
......preorder(20)
20
........preorder(null)
........preorder(null)
```

Recursion: inorder




```
inorder(14)
..inorder(4)
....inorder(3)
......inorder(null)
3
......inorder(null)
4
....inorder(9)
......inorder(7)
........inorder(5)
..........inorder(null)
5
..........inorder(null)
7
........inorder(null)
9
......inorder(null)
14
```

Recursion: inorder

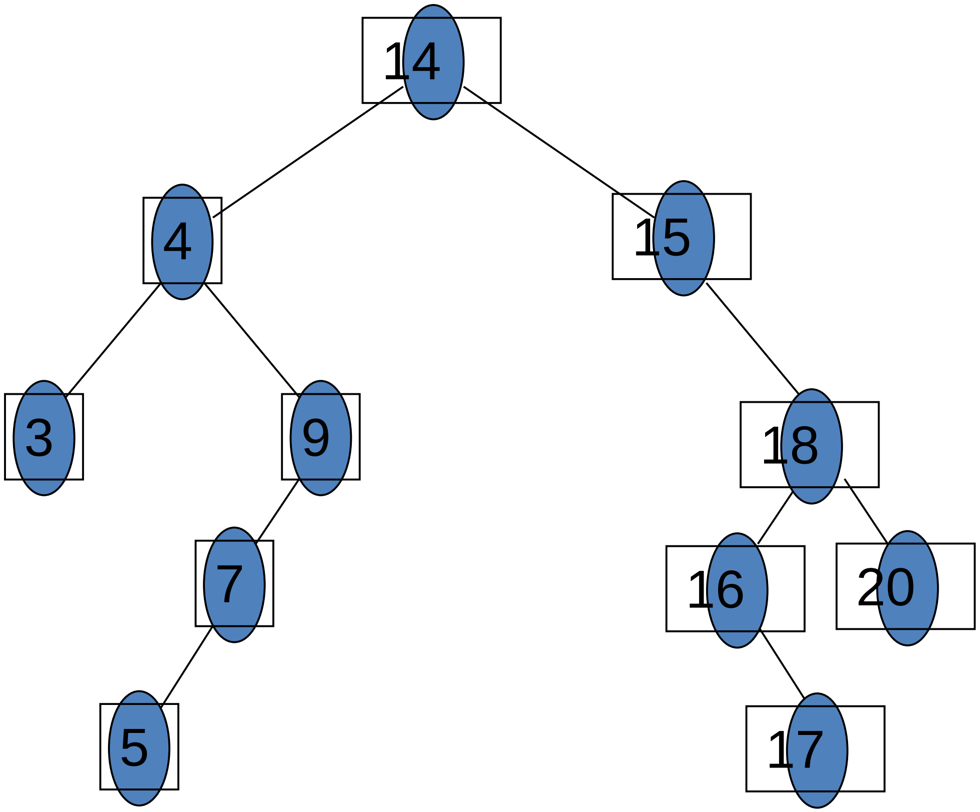



```
..inorder(15)
....inorder(null)
15
....inorder(18)
......inorder(16)
........inorder(null)
16
........inorder(17)
..........inorder(null)
17
..........inorder(null)
18
......inorder(20)
........inorder(null)
20
........inorder(null)
```

Non Recursive Traversal

- We can implement non-recursive versions of the preorder, inorder and postorder traversal by using an explicit stack.
- The stack will be used to store the tree nodes in the appropriate order.
- Here, for example, is the routine for inorder traversal that uses a stack.

Nonrecursive Inorder

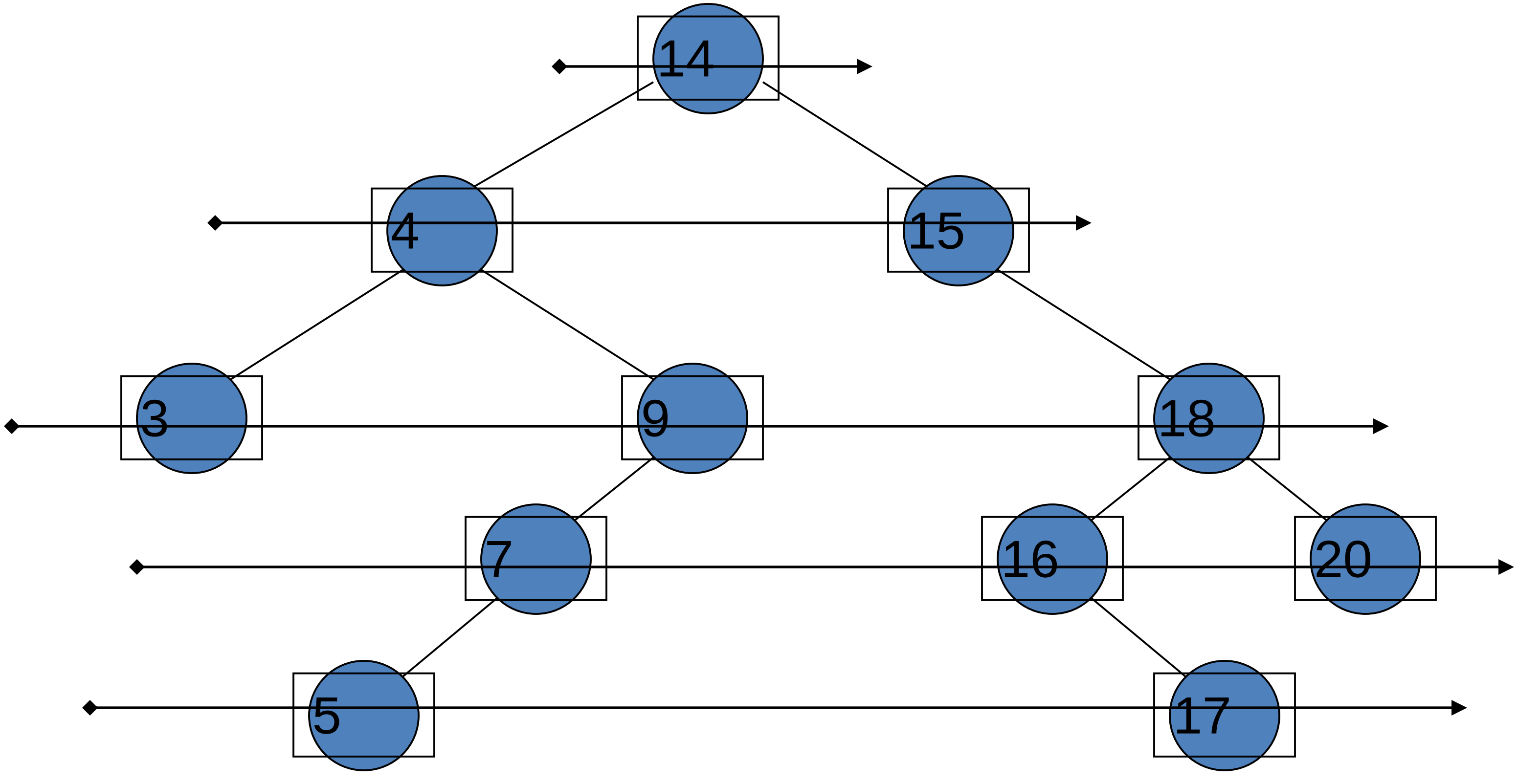



```
push(14)
..push(4)
....push(3)
3
4
..push(9)
....push(7)
......push(5)
5
7
9
14
push(15)
15
push(18)
..push(16)
16
..push(17)
17
18
push(20)
20
```

Level-order Traversal

- There is yet another way of traversing a binary tree that is not related to recursive traversal procedures discussed previously.
- In level-order traversal, we visit the nodes at each level before proceeding to the next level.
- At each level, we visit the nodes in a left-to-right order.

Level-order Traversal

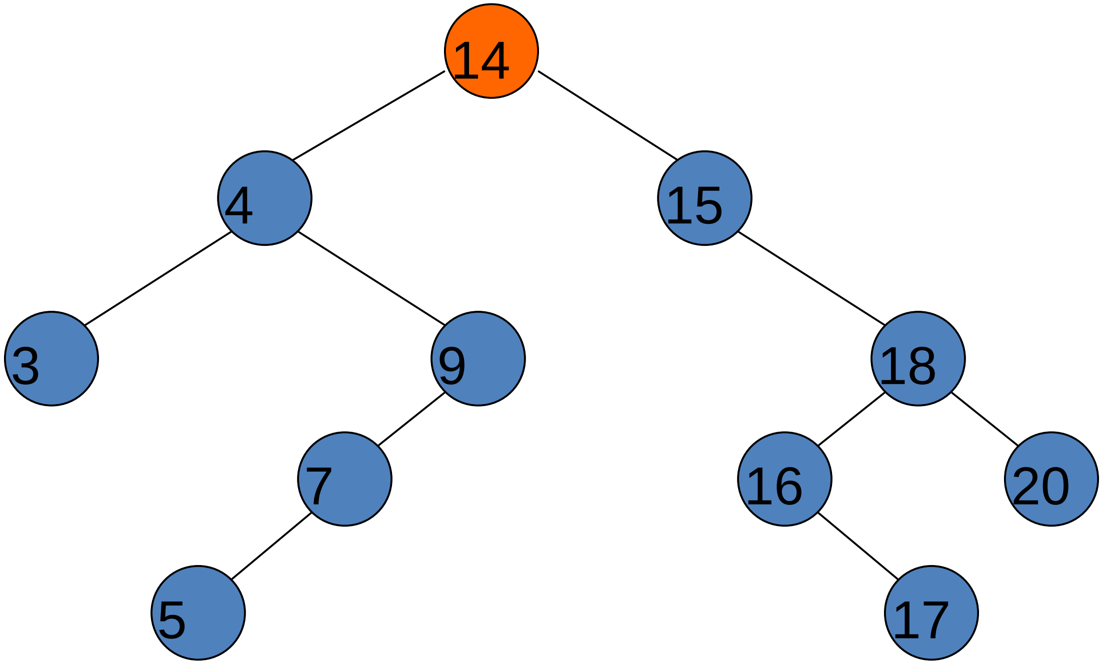


Level-order: 14 4 15 3 9 18 7 16 20 5 17

Level-order Traversal

- How do we do level-order traversal?
- Surprisingly, if we use a queue instead of a stack, we can visit the nodes in level-order.
- Here is the code for level-order traversal:

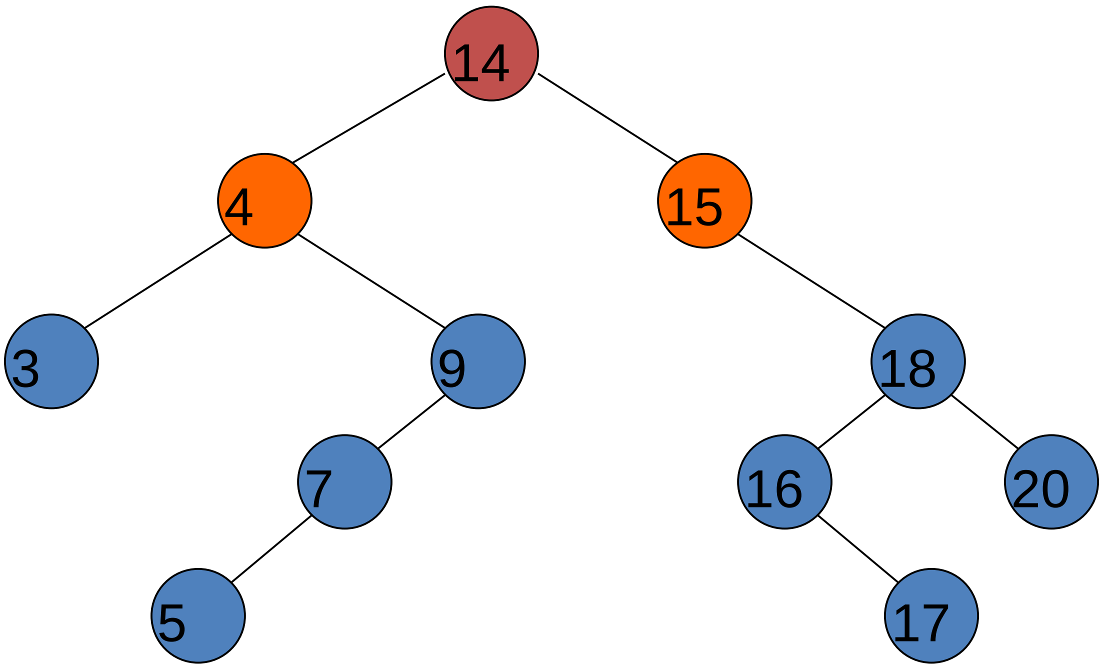
Level-order Traversal



Queue: 14

Output:

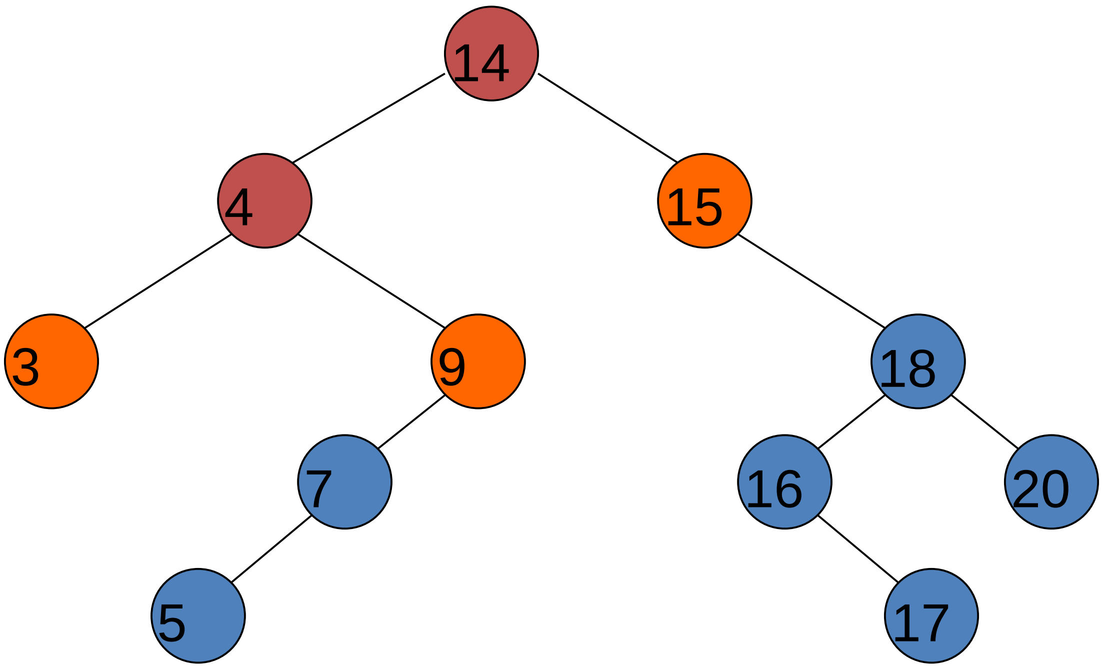
Level-order Traversal



Queue: 4 15

Output: 14

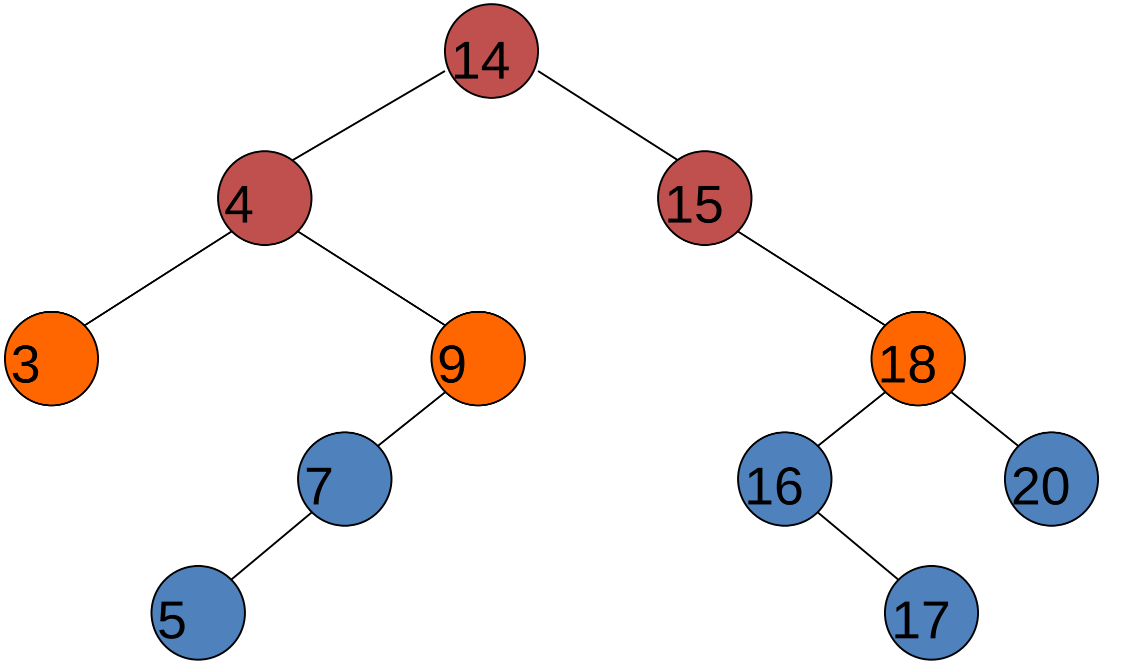
Level-order Traversal



Queue: 15 3 9

Output: 14 4

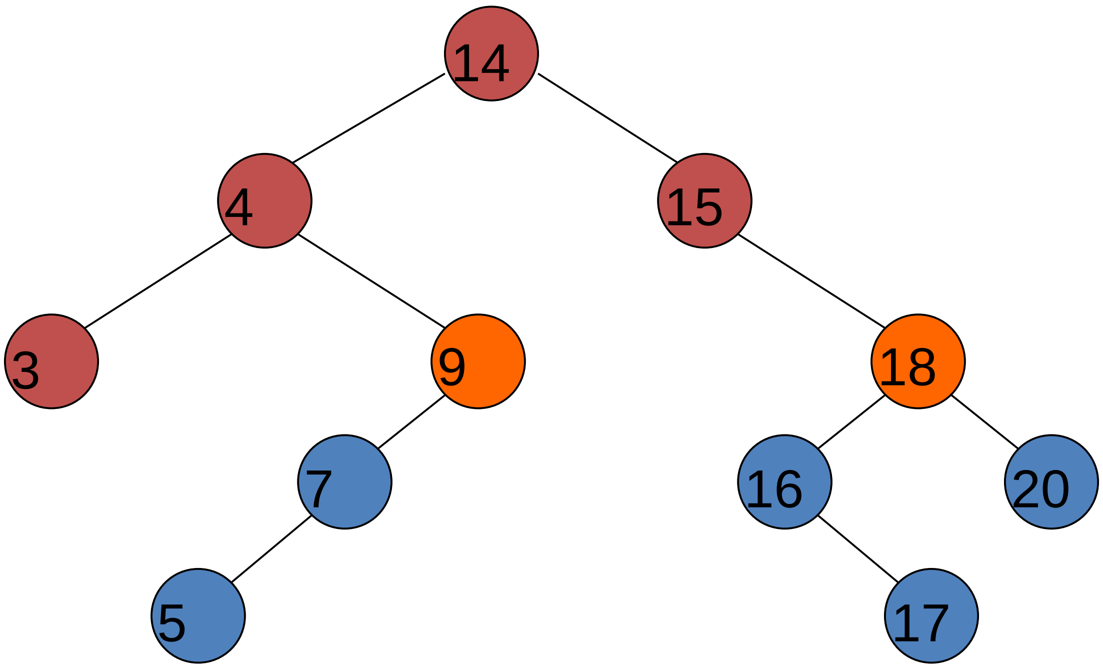
Level-order Traversal



Queue: 3 9 18

Output: 14 4 15

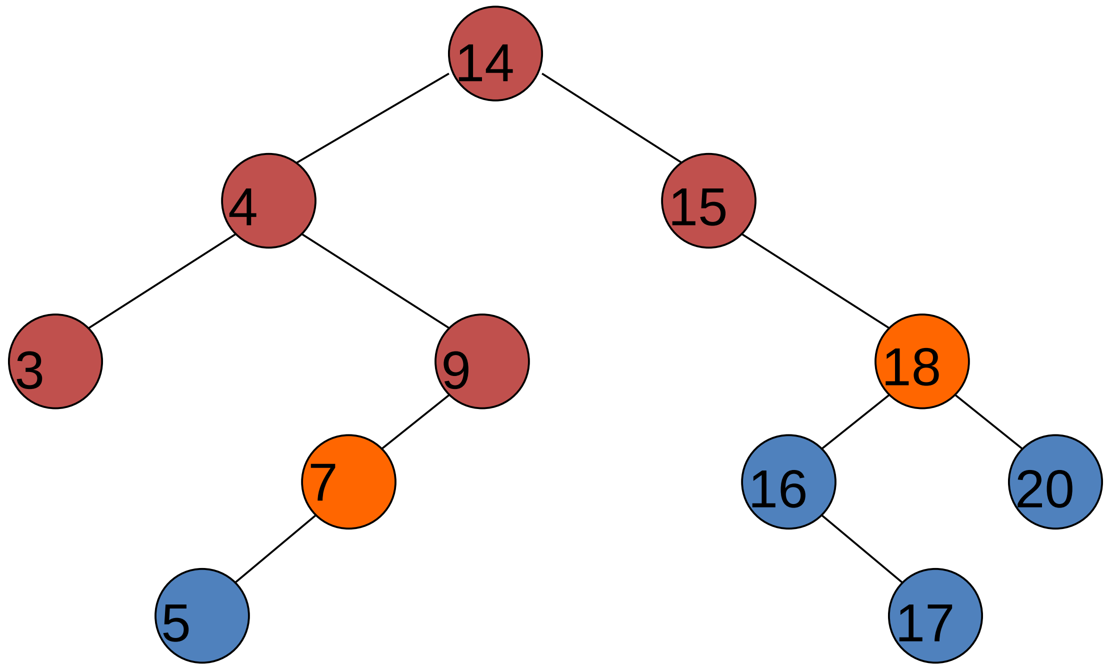
Level-order Traversal



Queue: 9 18

Output: 14 4 15 3

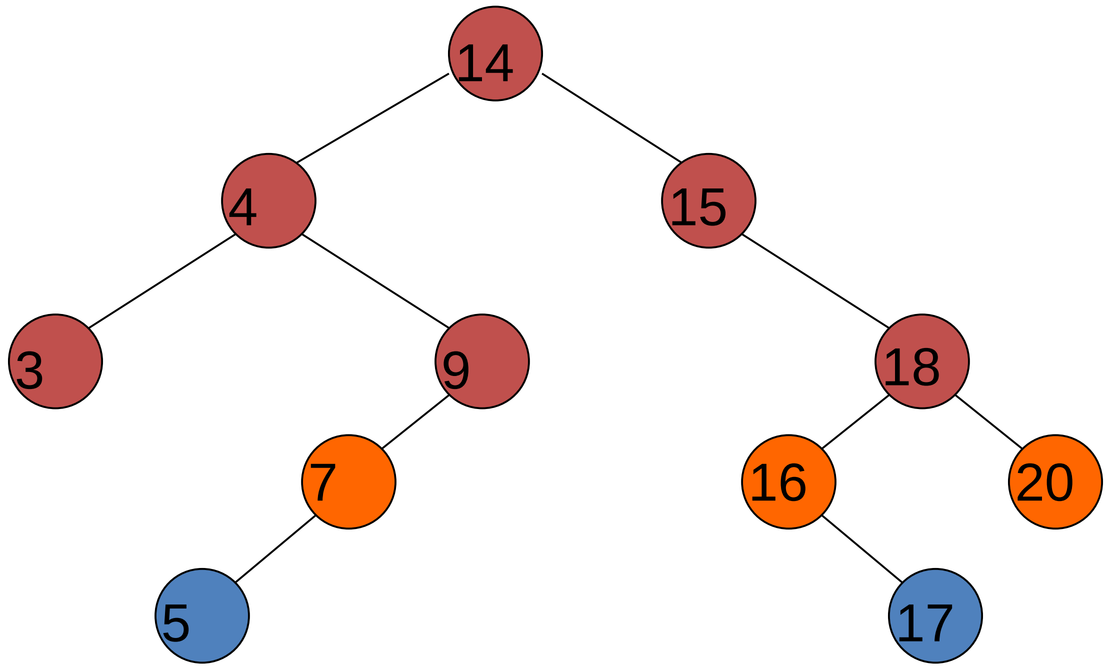
Level-order Traversal



Queue: 18 7

Output: 14 4 15 3 9

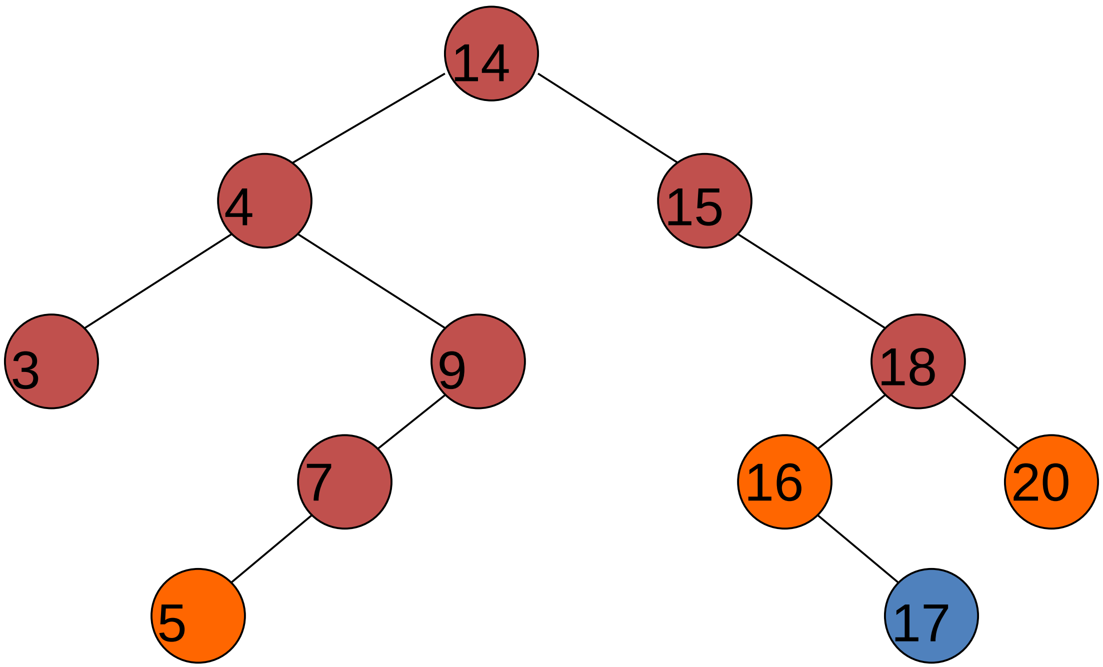
Level-order Traversal



Queue: 7 16 20

Output: 14 4 15 3 9 18

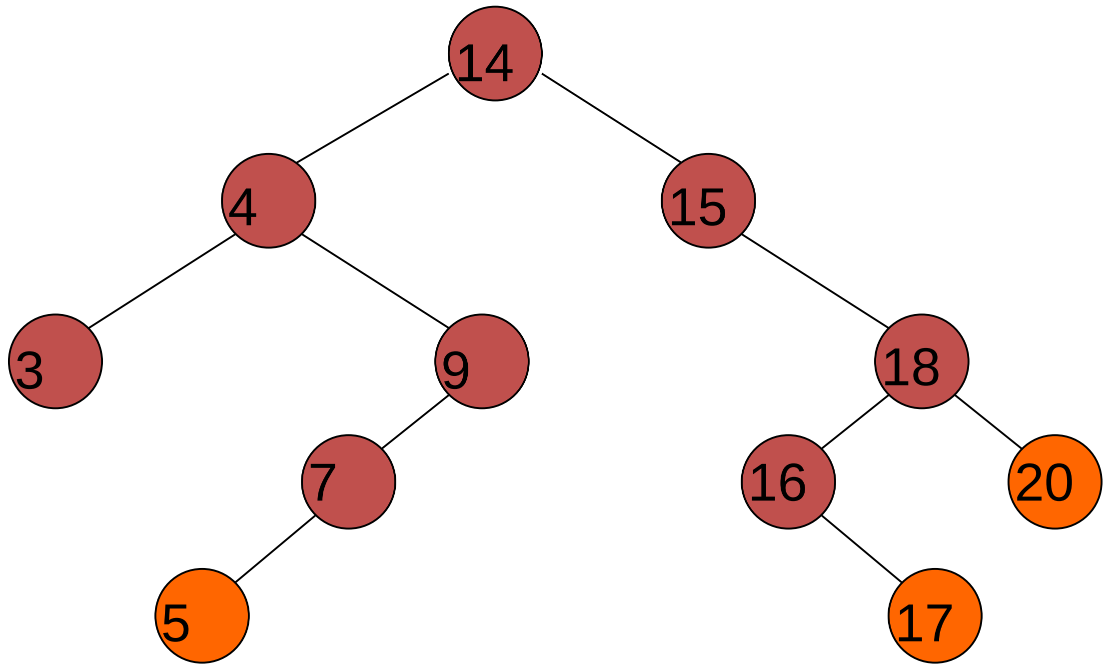
Level-order Traversal



Queue: 16 20 5

Output: 14 4 15 3   9 18   7

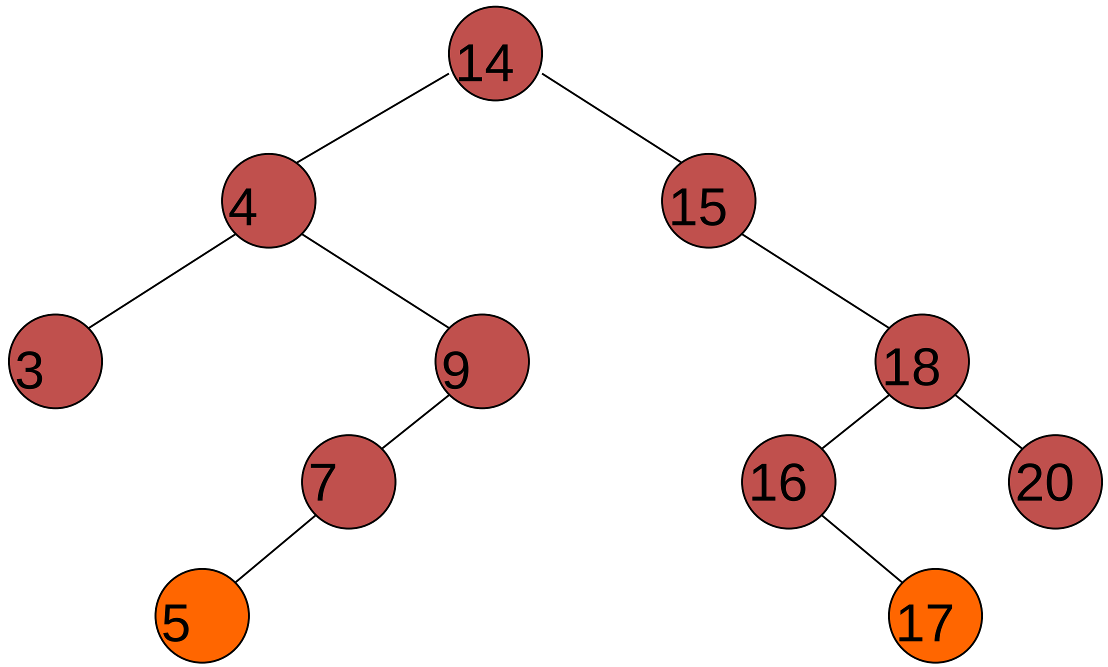
Level-order Traversal



Queue: 20 5 17

Output: 14 4 15 3 9 18 7 16

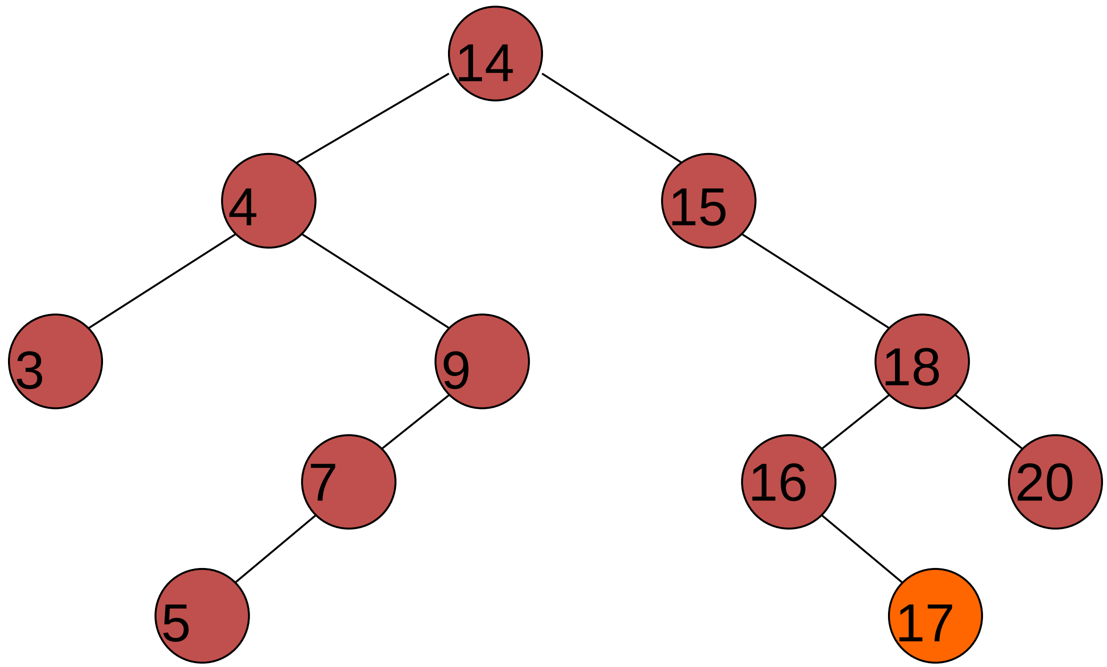
Level-order Traversal



Queue: 5 17

Output: 14 4 15 3 9 18 7 16 20

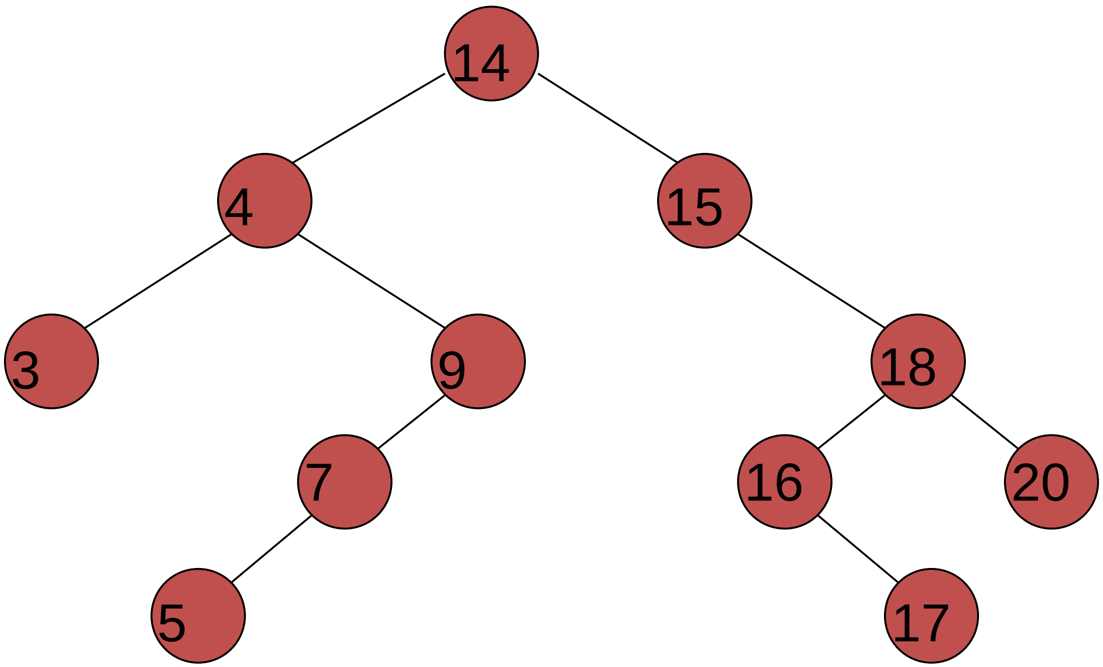
Level-order Traversal



Queue: 17

Output: 14 4 15 3   9 18   7 16 20 5

Level-order Traversal



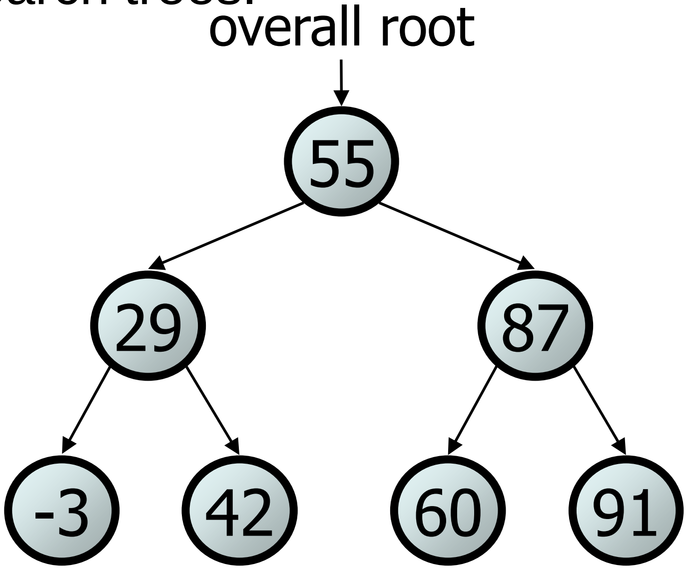
Queue:

Output: 14 4 15 3 9 18 7 16 20 5 17

Binary search trees

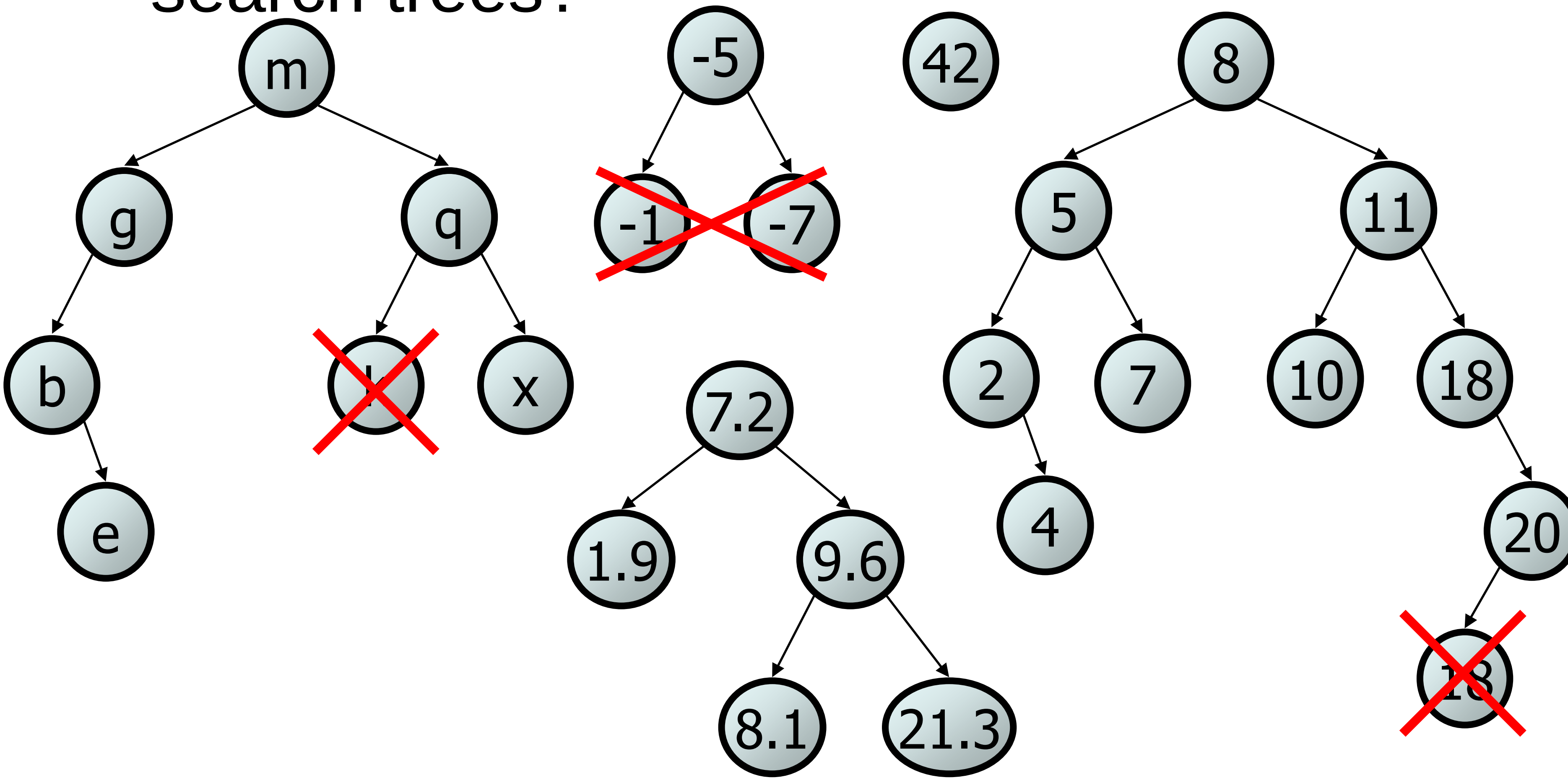
- **binary search tree** ("BST"): a binary tree where each non-empty node R has the following properties:
  - every element of R's left subtree contains data "less than" R's data,
  - every element of R's right subtree contains data "greater than" R's,
  - R's left and right subtrees are also binary search trees.

- BSTs store their elements in sorted order, which is helpful for searching/sorting tasks.

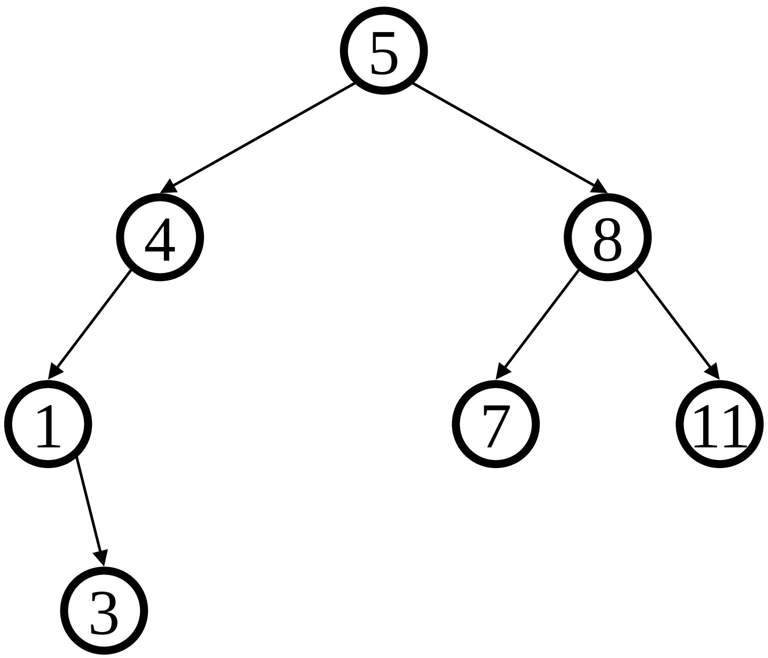


BST examples

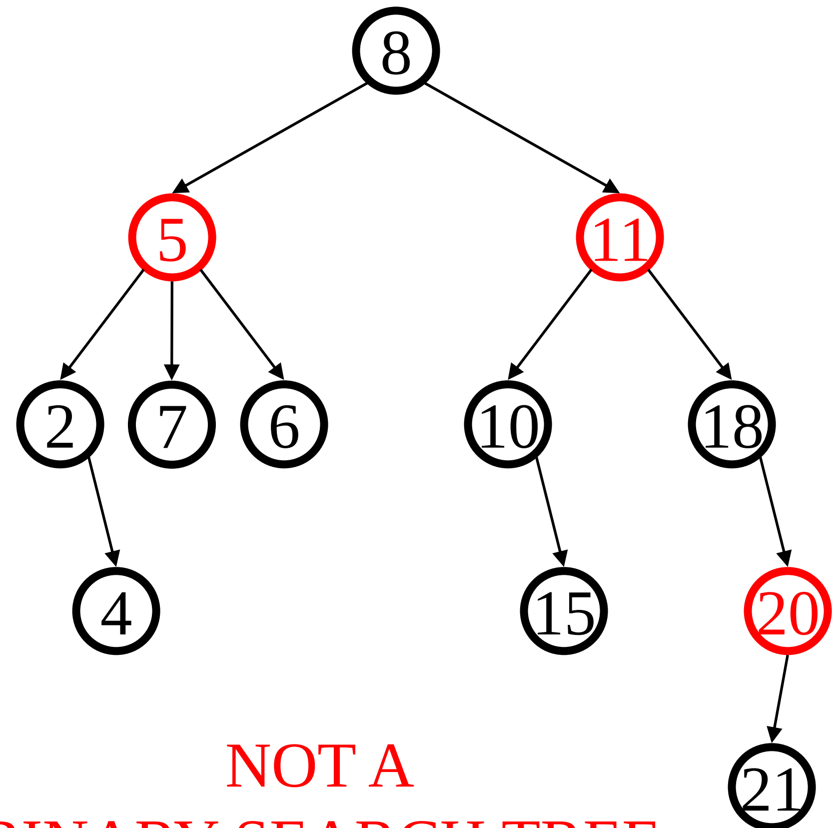
- Which of the trees shown are legal binary search trees?



Example and Counter-Example



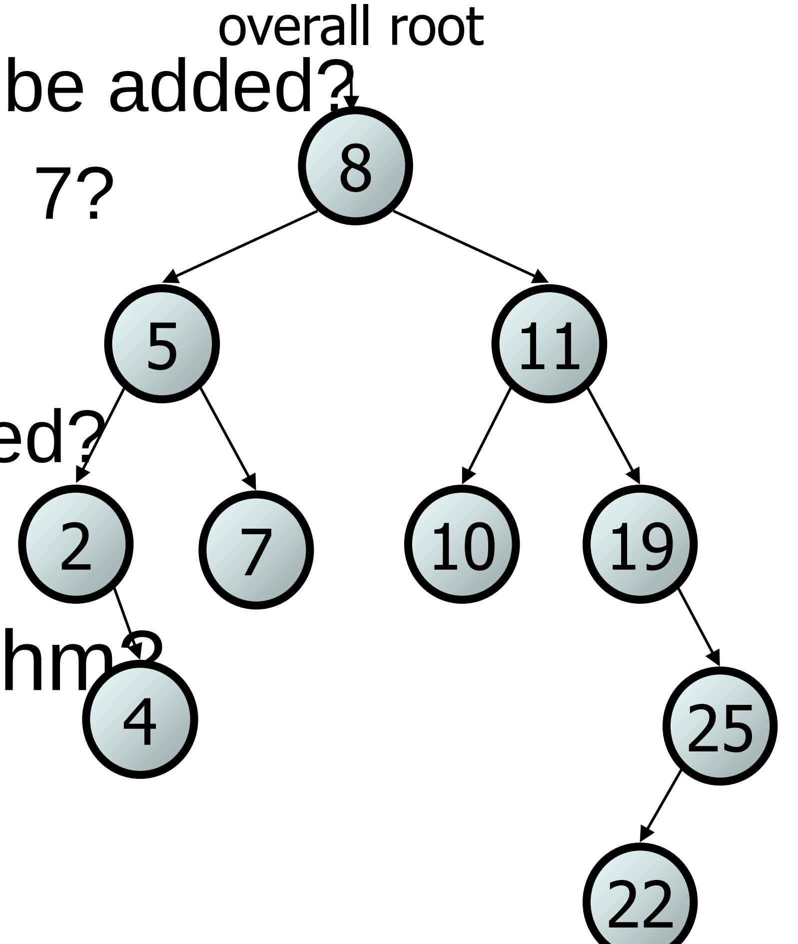
BINARY SEARCH TREE



NOT A
BINARY SEARCH TREE

Adding to a BST

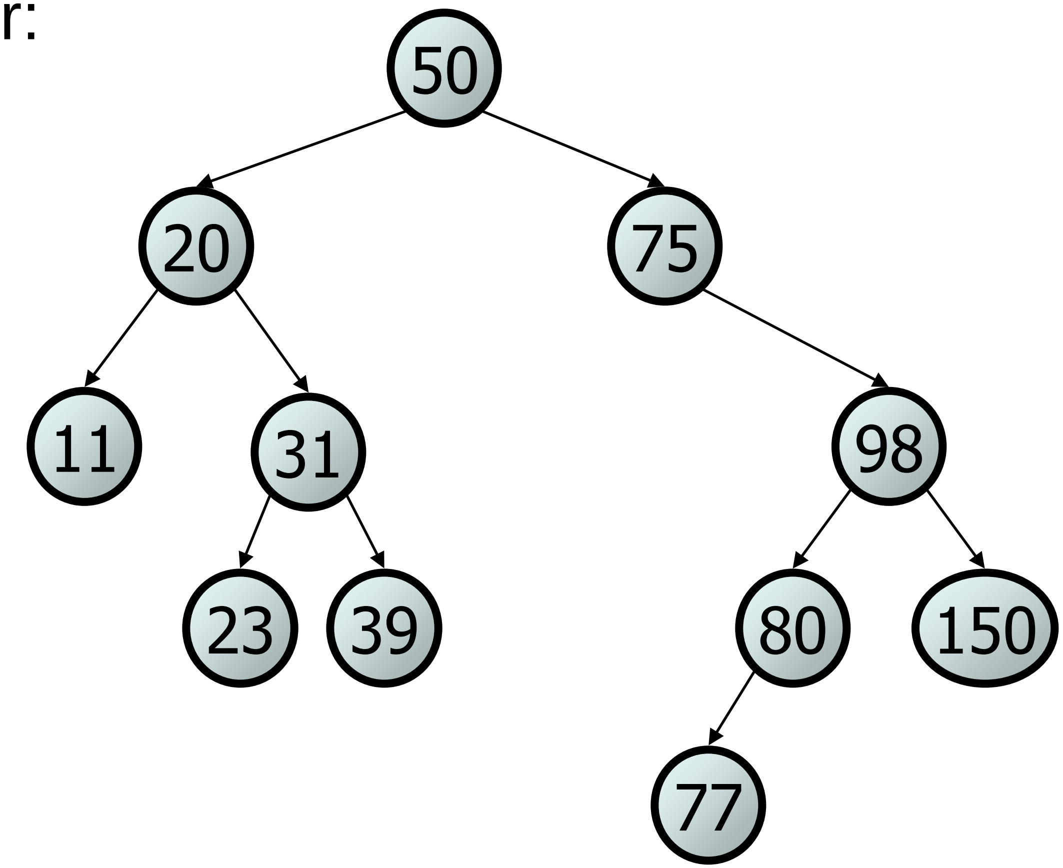
- Suppose we want to add new values to the BST below.
  - Where should the value 14 be added?
  - Where should 3 be added? 7?
  - If the tree is empty, where should a new value be added?
- What is the general algorithm?



Adding exercise

- Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

50
20
75
98
80
31
150
39
23
11
77

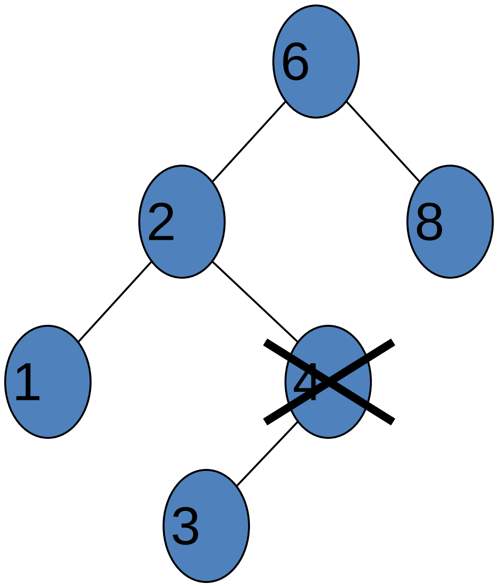


Deleting a node in BST

- As is common with many data structures, the hardest operation is deletion.
- Once we have found the node to be deleted, we need to consider several possibilities.
- If the node is a *leaf*, it can be deleted immediately.

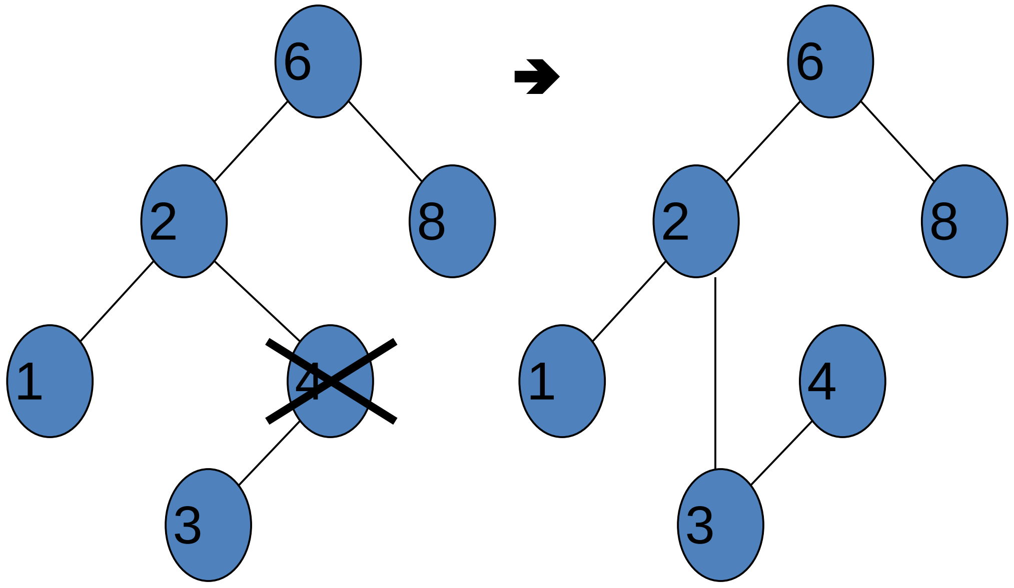
Deleting a node in BST

- If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node and connect to inorder successor.



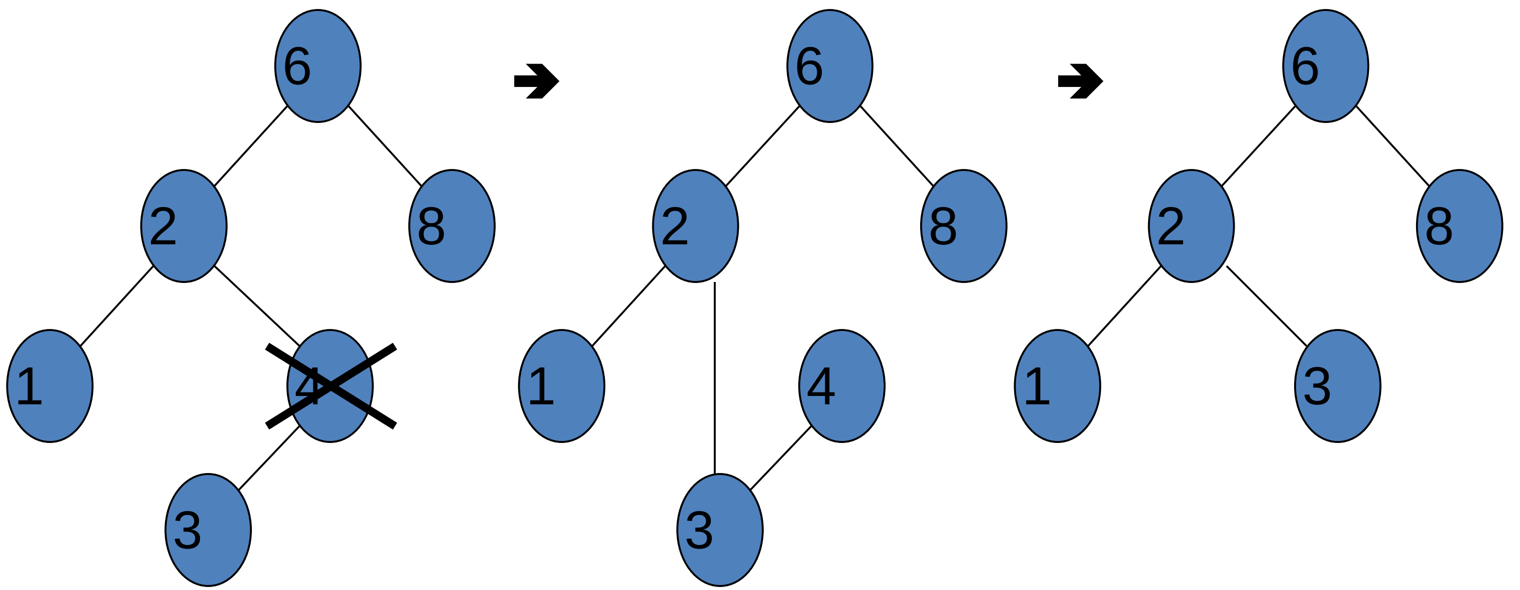
Deleting a node in BST

- The inorder traversal order has to be maintained after the delete.



Deleting a node in BST

- The inorder traversal order has to be maintained after the delete.

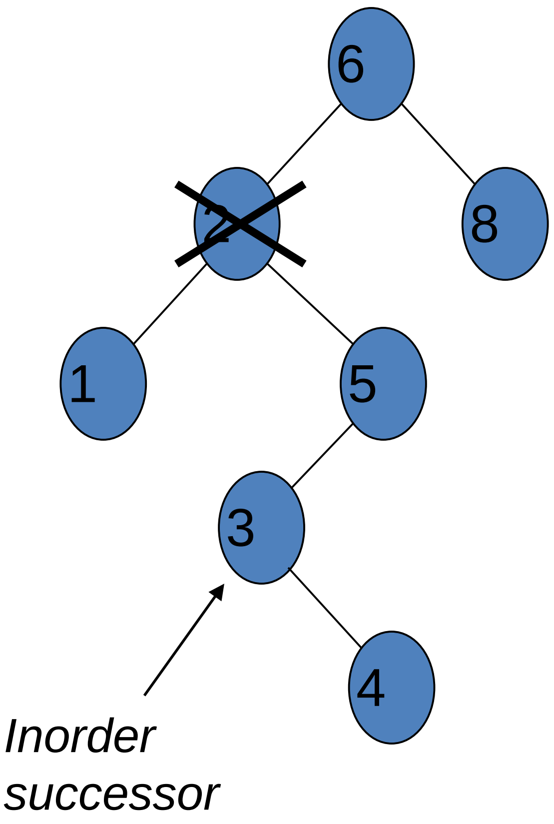


Deleting a node in BST

- The complicated case is when the node to be deleted has both left and right subtrees.
- The strategy is to replace the data of this node with the smallest data of the right subtree and recursively delete that node.

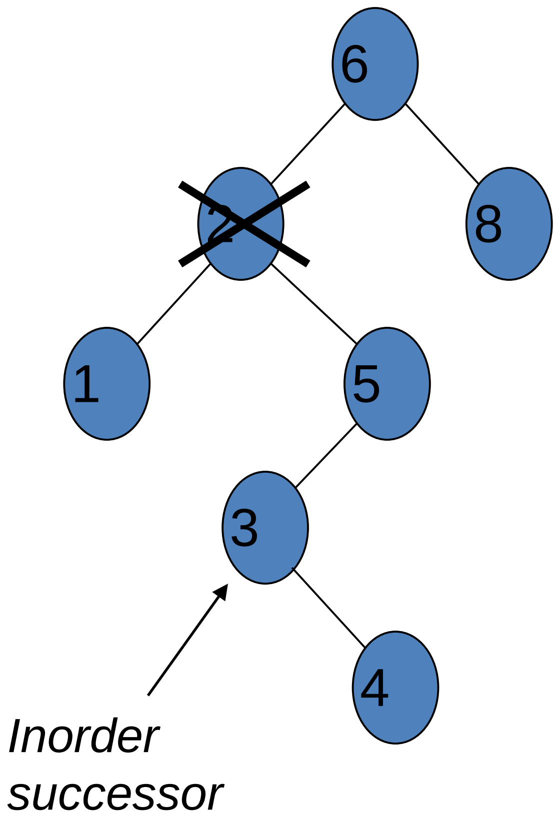
Deleting a node in BST

Delete(2): locate inorder successor



Deleting a node in BST

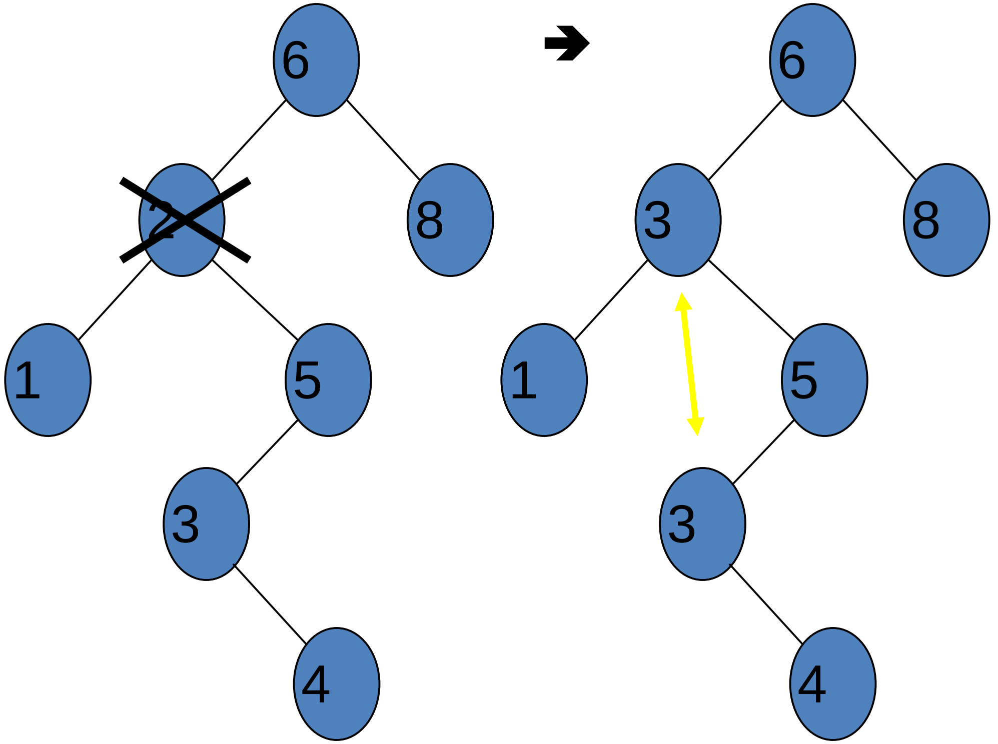
Delete(2): locate inorder successor



- Inorder successor will be the left-most node in the right subtree of 2.
- The inorder successor will not have a left child because if it did, that child would be the left-most node.

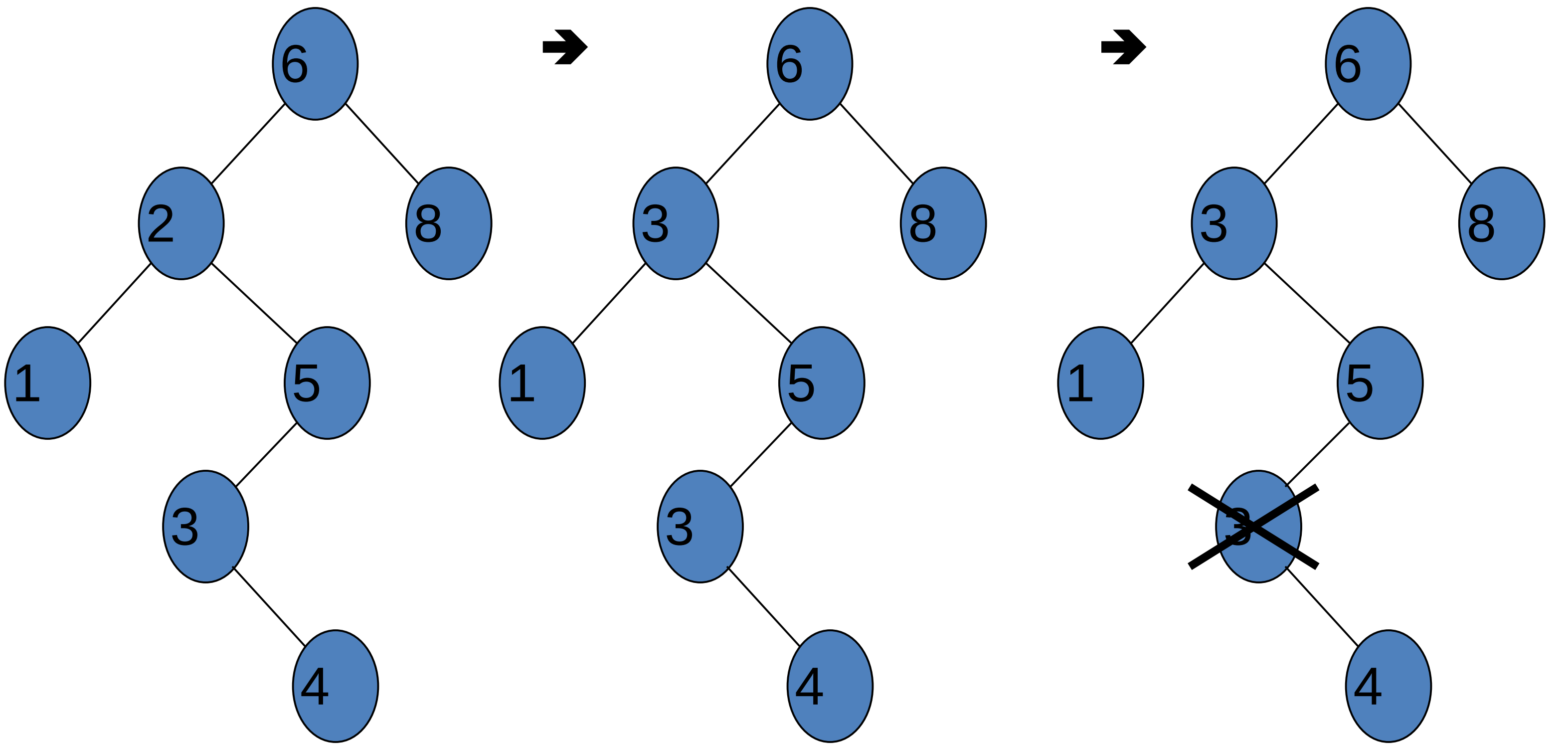
Deleting a node in BST

Delete(2): copy data from inorder successor



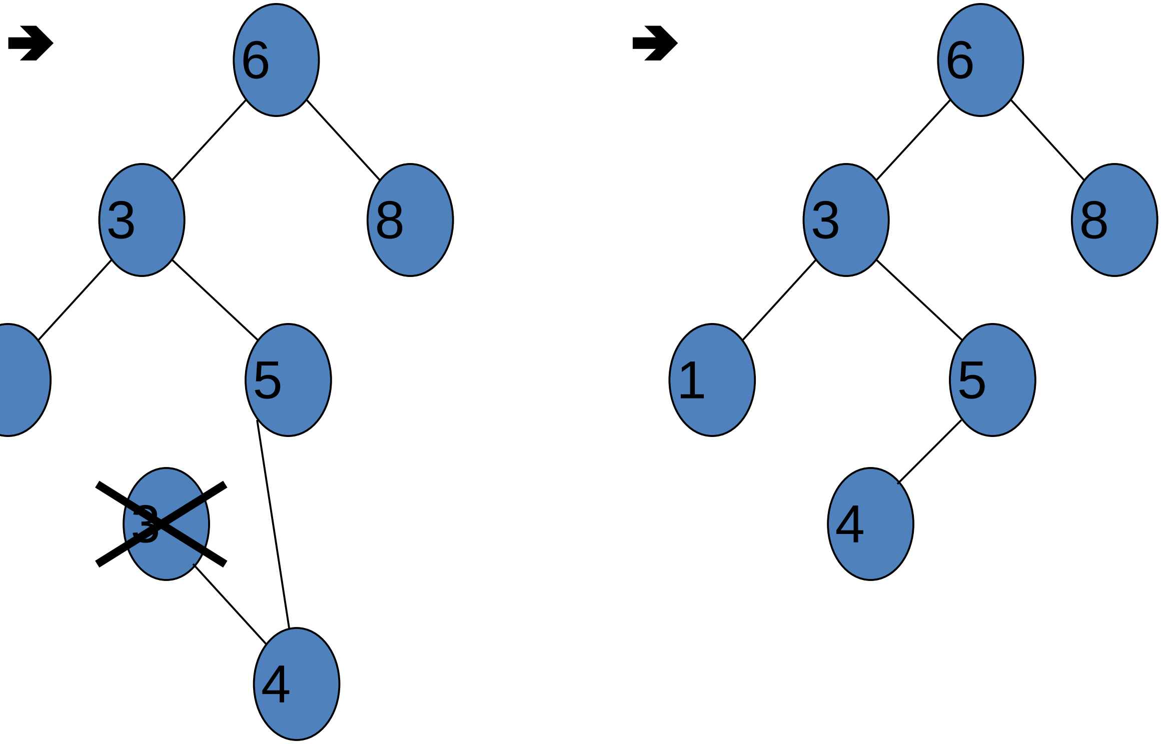
Deleting a node in BST

Delete(2): remove the inorder successor



Deleting a node in BST

Delete(2)



delete

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  – This has to be done such that the property of the search tree is maintained.

delete

Three cases:

(1) the node is a leaf

- Delete it immediately

(2) the node has one child

- Adjust a pointer from the parent to bypass that node

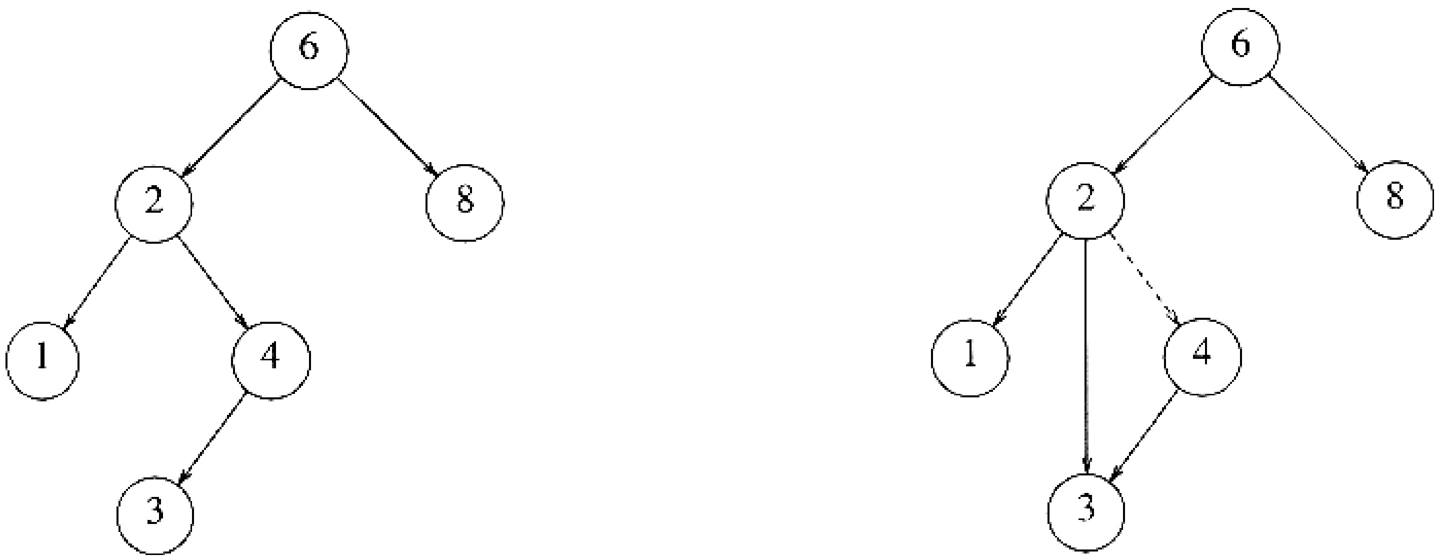


Figure 4.24 Deletion of a node (4) with one child, before and after

delete

(3) the node has 2 children

- replace the key of that node with the minimum element at the right subtree
- delete the minimum element
  - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

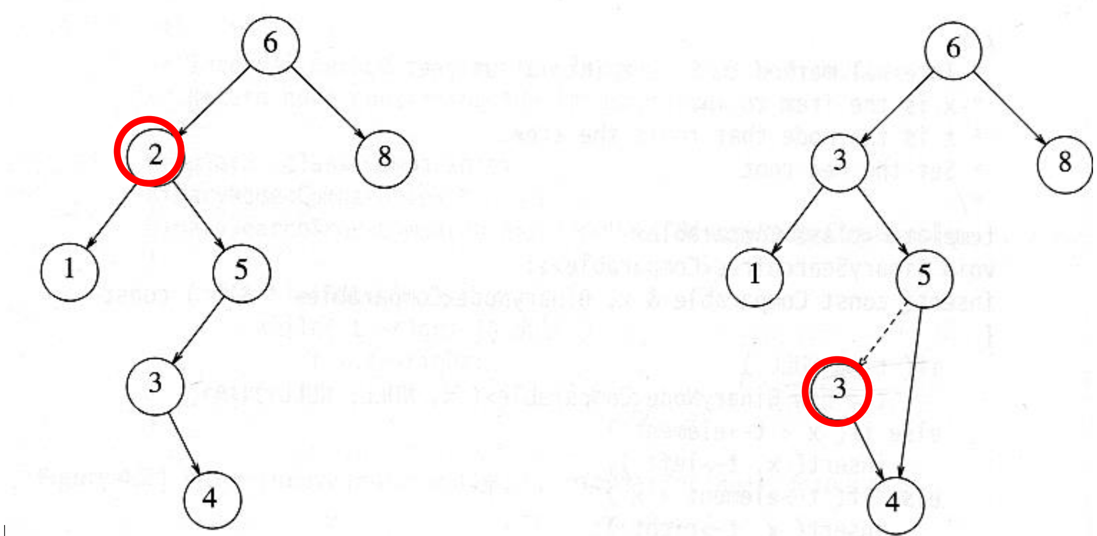
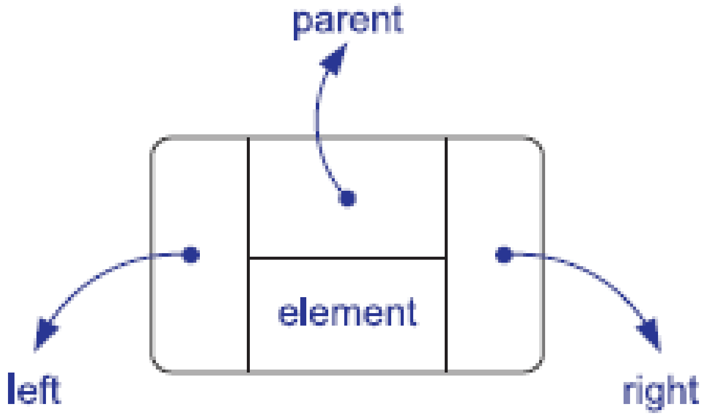


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time com

IMPLEMENTATION


parent
element
left
right

Binary Search Tree Implementation

- We use two classes:
  - **BSTNode**
  - **BST**
- Declare `BSTNode` class for the nodes
  - `info`: `int`-type data in this example

```
class BSTNode {
private:
        int             info;        // data
        BSTNode*        left;        // pointer to Left
        BSTNode*        right;       // pointer to Right
        BSTNode*        parent;      // pointer to Parent



};
```

BSTNode Implementation

```
class BSTNode{
public:
        //constructors
        BSTNode();
        BSTNode( int info);
        int getInfo();
        void setInfo(int info);
        BSTNode* getLeft();
        void setLeft(BSTNode *left);
        BSTNode *getRight();
        void setRight(BSTNode *right);
        int isLeaf( );
        BSTNode* getParent();
        void setParent(BSTNode *parent);


private:
        int     info;           // data
        BSTNode*        left;           // pointer to Left
        BSTNode*        right;          // pointer to Right
        BSTNode*        parent;         // pointer to Parent


};
```

BSTNode Constructors

```
BSTNode::BSTNode()
{
        this->info= -1;
         this->left = this->right = NULL;
        this->parent = NULL;

}
BSTNode:: BSTNode( int info)
{
        this->info= info;
         this->left = this->right = NULL;
        this->parent = NULL;

}
```

BSTNode Methods

```
int BSTNode::getInfo()
{
        return this->info;
}
void BSTNode::setInfo(int info)
{
        this->info = info;

}
BSTNode* BSTNode::getLeft()
{
        return this->left;

}
void BSTNode::setLeft (BSTNode *left)
{
        this->left = left;
}
```

BSTNode Methods

```
BSTNode* BSTNode::getRight()
{
        return this->right;

}
void BSTNode::setRight(BSTNode *right)
{

    this->right = right;

}
int BSTNode::isLeaf( )
{

        if( this->left == NULL && this->right == NULL )
                   return 1;
        return 0;
}
BSTNode* BSTNode::getParent()
{
        return this->parent;

}
void BSTNode::setParent(BSTNode *parent)
{
        this->parent = parent;


}
```

BST

```
class BST{
public:
     void insert(int info);
     bool search(int key);
     BSTNode* remove(BSTNode* tree,int key);
     BSTNode* findMin(BSTNode* Node);
    BSTNode* findMax(BSTNode* Node);
     int height();
     void preorder(BSTNode* root);
    void postorder(BSTNode* root);
    void inorder(BSTNode* root);


public:

      BSTNode*                    root;


};
```

Insert

```
void BST::insert(int value)
{
	BSTNode  *p = root, *prev = 0;
	// find a place for inserting new node;
	while (p != 0)
	{

		prev = p;
		if (value < p->getInfo())
			p = p->left;
		else
			p = p->right;
	}
	if (root == 0) // tree is empty;
		root = new BSTNode(value);
	else if (value < prev->getInfo()){
		BSTNode temp= new BSTNode(value);
		temp->setParent (prev);
		prev->setLeft (temp);

	}else{
		BSTNode temp = new BSTNode(value);
		temp->setParent ( prev);
		prev->setright (temp);

		}
}
```

C++ code for delete

```
BSTNode* BST::remove(BSTNode* tree, int info)
{
    BSTNode* t;
    int cmp = info - tree->getInfo();
    if( cmp < 0 ){
        t = remove(tree->getLeft(), info);
        tree->setLeft( t );
    }
    else if( cmp > 0 ){
        t = remove(tree->getRight(), info);
        tree->setRight( t );
    }
```

C++ code for delete

```
BSTNode* BST::remove(BSTNode* tree, int info)
{
    BSTNode* t;
    int cmp = info - tree->getInfo();
    if( cmp < 0 ){
        t = remove(tree->getLeft(), info);
        tree->setLeft( t );
    }
    else if( cmp > 0 ){
        t = remove(tree->getRight(), info);
        tree->setRight( t );
    }
```

C++ code for delete

```
BSTNode* BST::remove(BSTNode* tree, int info)
{
    BSTNode* t;
    int cmp = info - tree->getInfo();
    if( cmp < 0 ){
        t = remove(tree->getLeft(), info);
        tree->setLeft( t );
    }
    else if( cmp > 0 ){
        t = remove(tree->getRight(), info);
        tree->setRight( t );
    }
```

C++ code for delete

```
BSTNode* BST::remove (BSTNode* tree, int info)
{
    BSTNode* t;

    int cmp = info - tree->getInfo();
    if( cmp < 0 ){
        t = remove(tree->getLeft(), info);
        tree->setLeft( t );
    }
    else if( cmp > 0 ){
        t = remove(tree->getRight(), info);
        tree->setRight( t );
    }
```

C++ code for delete

```
//two children, replace with inorder successor
else if(tree->getLeft() != NULL
        && tree->getRight() != NULL ){
    BSTNode* minNode;
    minNode = findMin(tree->getRight());
    tree->setInfo( minNode->getInfo() );
    t = remove(tree->getRight(),
               minNode->getInfo());
    tree->setRight( t );
}
```

C++ code for delete

```
//two children, replace with inorder successor
else if(tree->getLeft() != NULL
        && tree->getRight() != NULL ){
    BSTNode* minNode;
    minNode = findMin(tree->getRight());
    tree->setInfo( minNode->getInfo() );
    t = remove(tree->getRight(),
                    minNode->getInfo());
    tree->setRight( t );
}
```

C++ code for delete

```
//two children, replace with inorder successor
else if(tree->getLeft() != NULL
        && tree->getRight() != NULL ){
    BSTNode* minNode;
    minNode = findMin(tree->getRight());
    tree->setInfo( minNode->getInfo() );
    t = remove(tree->getRight(),
               minNode->getInfo());
    tree->setRight( t );
}
```

C++ code for delete

```
//two children, replace with inorder successor
else if(tree->getLeft() != NULL
        && tree->getRight() != NULL ){
    BSTNode* minNode;
    minNode = findMin(tree->getRight());
    tree->setInfo( minNode->getInfo() );
    t = remove(tree->getRight(),
                minNode->getInfo());
    tree->setRight( t );
}
```

C++ code for delete

```
//two children, replace with inorder successor
else if(tree->getLeft() != NULL
        && tree->getRight() != NULL ){
    BSTNode* minNode;
    minNode = findMin(tree->getRight());
    tree->setInfo( minNode->getInfo() );
    t = remove(tree->getRight(),
                  minNode->getInfo());
    tree->setRight( t );
}
```

C++ code for delete

```
    else {  // case 1
        BSTNode* nodeToDelete = tree;
        if( tree->getLeft() == NULL ) //will handle 0 children
            tree = tree->getRight();
        else if( tree->getRight() == NULL )
            tree = tree->getLeft();
        else tree = NULL;

        delete nodeToDelete;
    }
    return tree;
}
```

C++ code for delete

```
else {  // case 1
     BSTNode<int>* nodeToDelete = tree;
     if( tree->getLeft() == NULL ) //will handle 0 children
          tree = tree->getRight();
     else if( tree->getRight() == NULL )
          tree = tree->getLeft();
     else tree = NULL;

     delete nodeToDelete;
}
return tree;
}
```

C++ code for delete

```
else {  // case 1
        BSTNode<int>* nodeToDelete = tree;
        if( tree->getLeft() == NULL ) //will handle 0 children
            tree = tree->getRight();
        else if( tree->getRight() == NULL )
            tree = tree->getLeft();
        else tree = NULL;

        delete nodeToDelete;
    }
    return tree;
}
```

FindMin

```
BSTNode* findMin(BSTNode* Node)
{
    if( Node== NULL )
        return NULL;
    if( Node->getLeft() == NULL )
        return Node; // this is it.
    return findMin( Node->getLeft() );
}
```

FindMin

```
BSTNode* findMin(BSTNode* Node)
{
    if( Node== NULL )
        return NULL;
    if( Node->getLeft() == NULL )
        return Node; // this is it.
    return findMin( Node->getLeft() );
}
```

FindMin

```
BSTNode* findMin(BSTNode* Node)
{
    if( Node== NULL )
        return NULL;
    if( Node->getLeft() == NULL )
        return Node; // this is it.
    return findMin( Node->getLeft() );
}
```

FindMax

```
BSTNode* findMax(BSTNode* Node)
{
    if( Node== NULL )
        return NULL;
    if( Node->getRight() == NULL )
        return Node; // this is it.
     return findMax( Node->getRight() );
}
```

Search

```
bool search(int value)
{
      BSTNode* p = root;
      while (p != 0)
           {
                if (value== p->getInfo())
                   return true;
                else if (value < p->getInfo())
                   p = p->getLeft();
                else
                   p = p->getRight();
           }
           return 0;
}
```

Lecture content adapted from Michael T. Goodrich textbook, chapters 7 and 10.