Tree Traversal

- Many different algorithms for manipulating trees exist, but these algorithms have in common that they systematically visit all the nodes in the tree.
- There are essentially two methods for visiting all nodes in a tree:
    - Depth-first traversal,
    - Breadth-first traversal.

Depth-first Traversal

- Pre-order traversal:
  - Visit the root first; and then
  - Do a preorder traversal of each of the subtrees of the root one-by-one in the order given (from left to right).
- Post-order traversal:
  - Do a postorder traversal of each of the subtrees of the root one-by-one in the order given (from left to right); and then
  - Visit the root.
- In-order traversal:
  - Traverse the left subtree; and then
  - Visit the root; and then
  - Traverse the right subtree.

Breadth-first Traversal

- Breadth-first traversal visits the nodes of a tree in the order of their depth (from left to right).

Pre-order Traversal Example

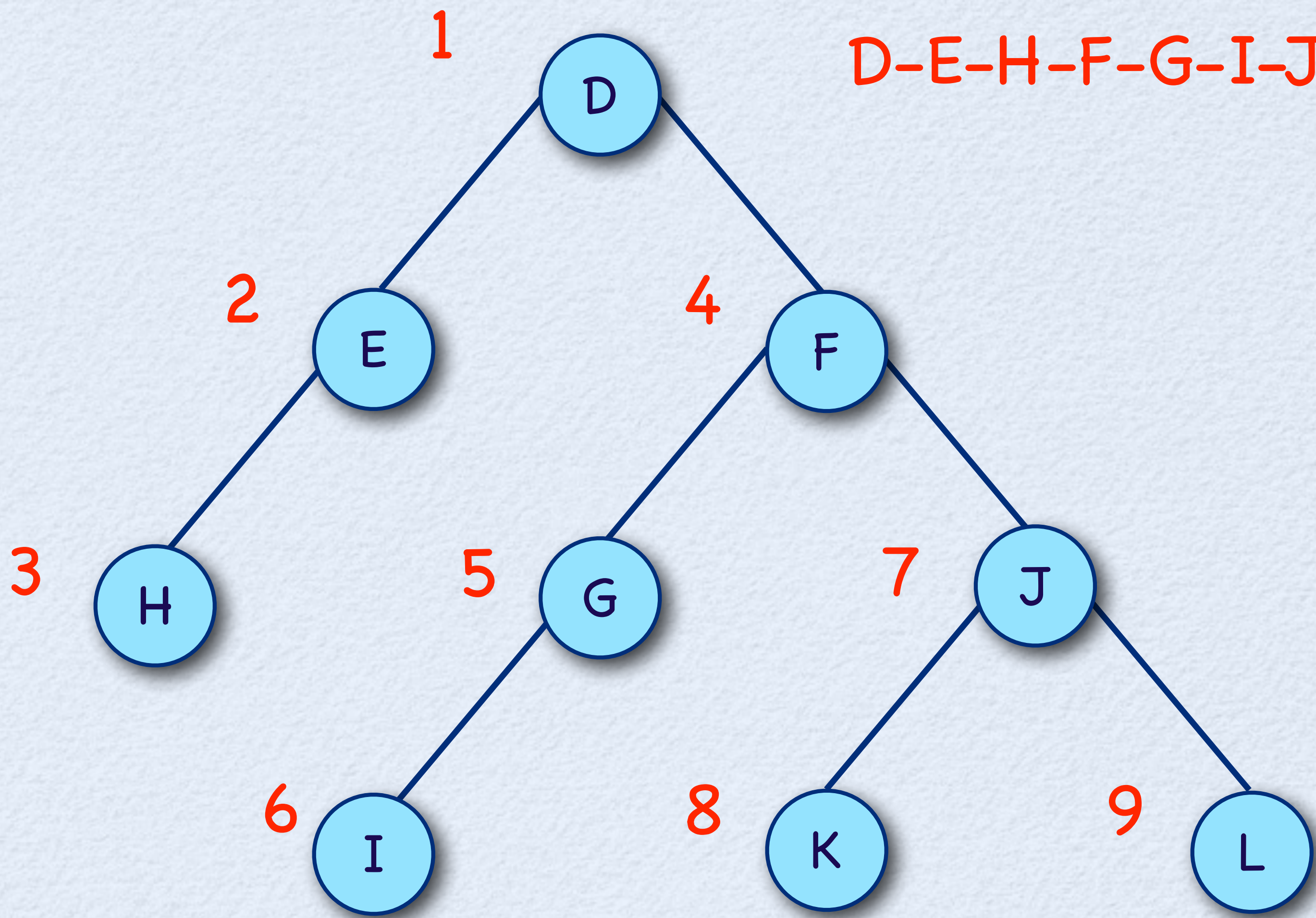
D-E-H-F-G-I-J-K-L

Depth = 0:

Depth = 1:

Depth = 2:

Depth = 3:



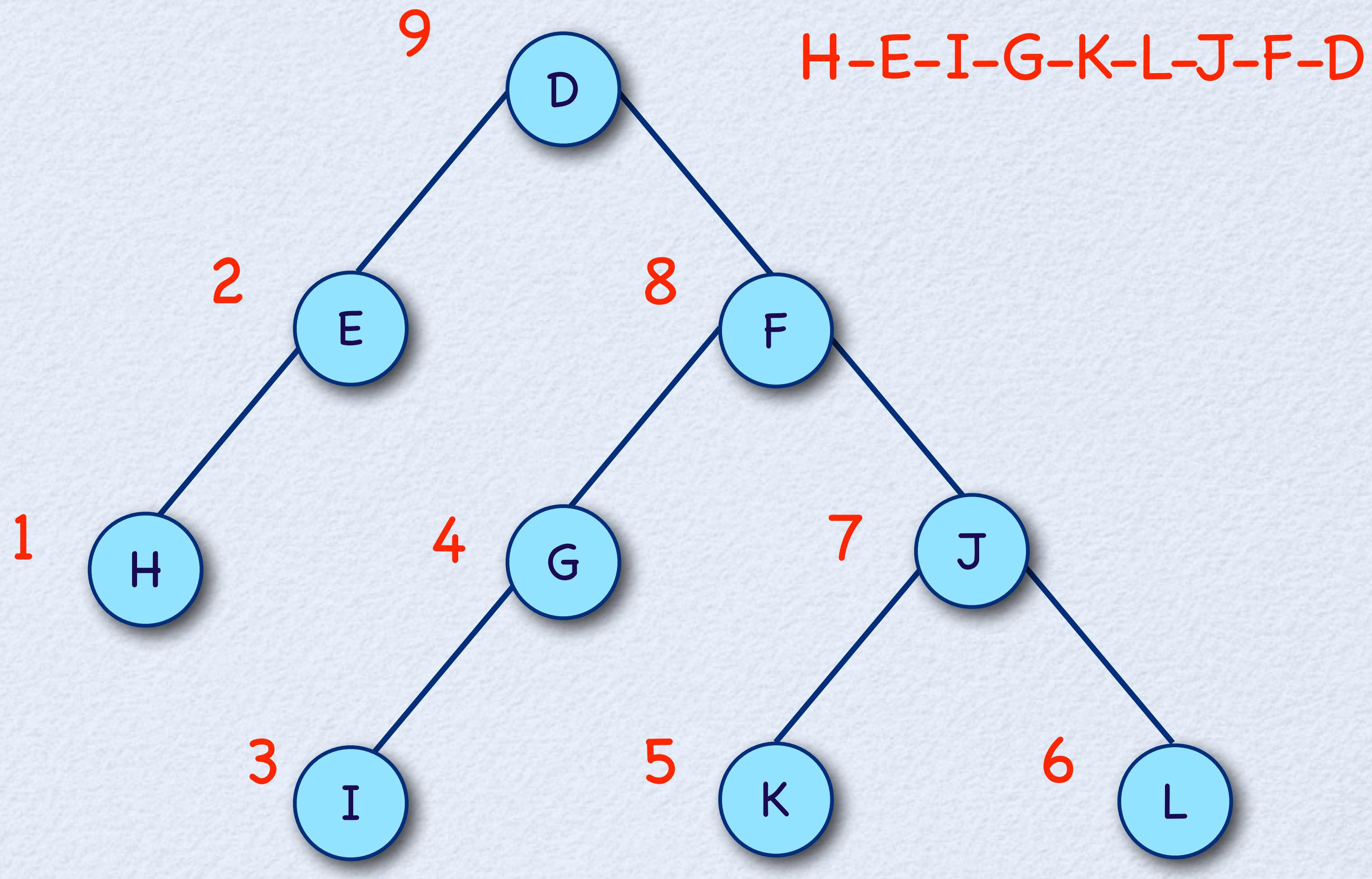
Post-order Traversal Example

Depth = 0:

Depth = 1:

Depth = 2:

Depth = 3:



In-order Traversal Example

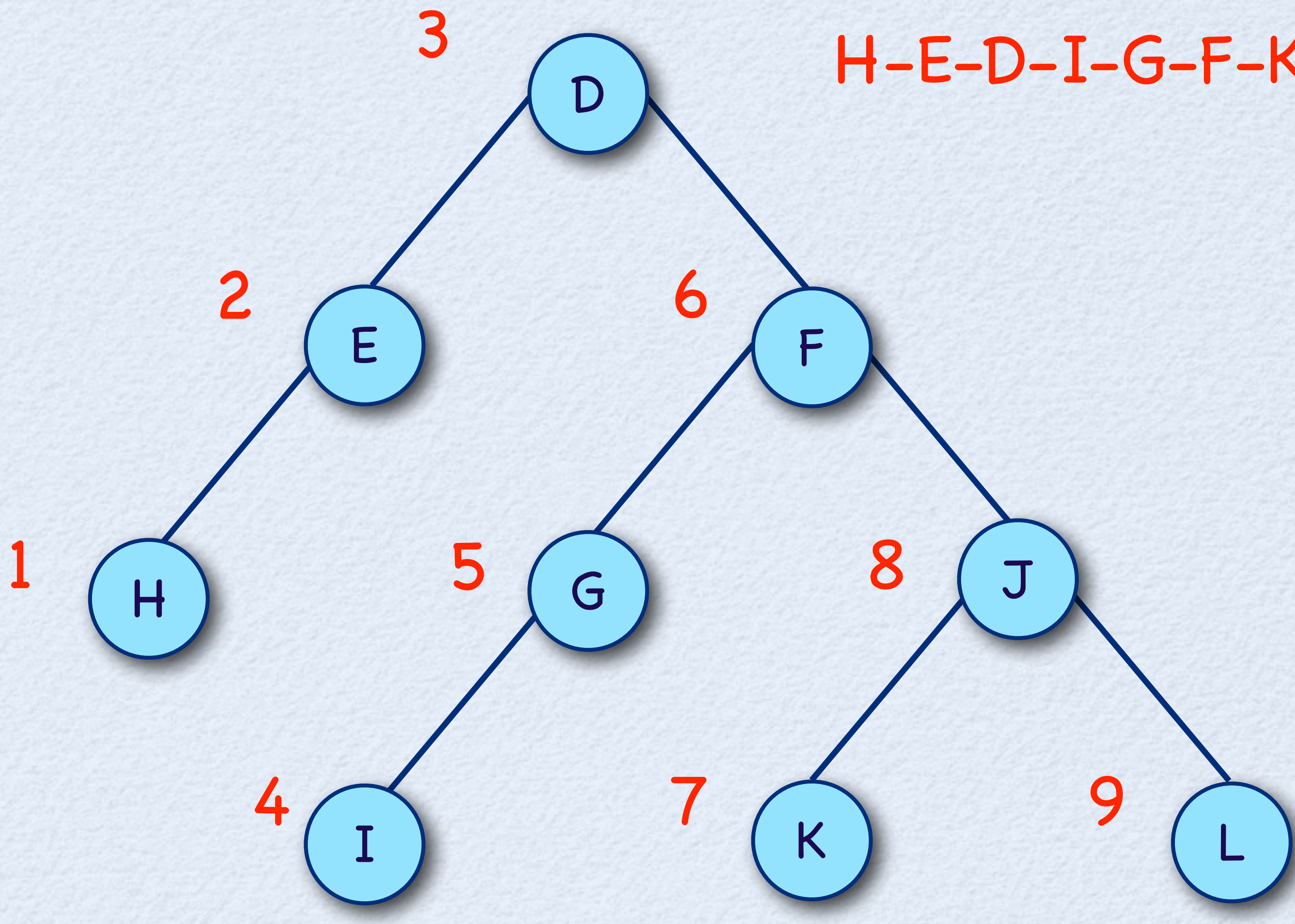
H-E-D-I-G-F-K-J-L

Depth = 0:

Depth = 1:

Depth = 2:

Depth = 3:



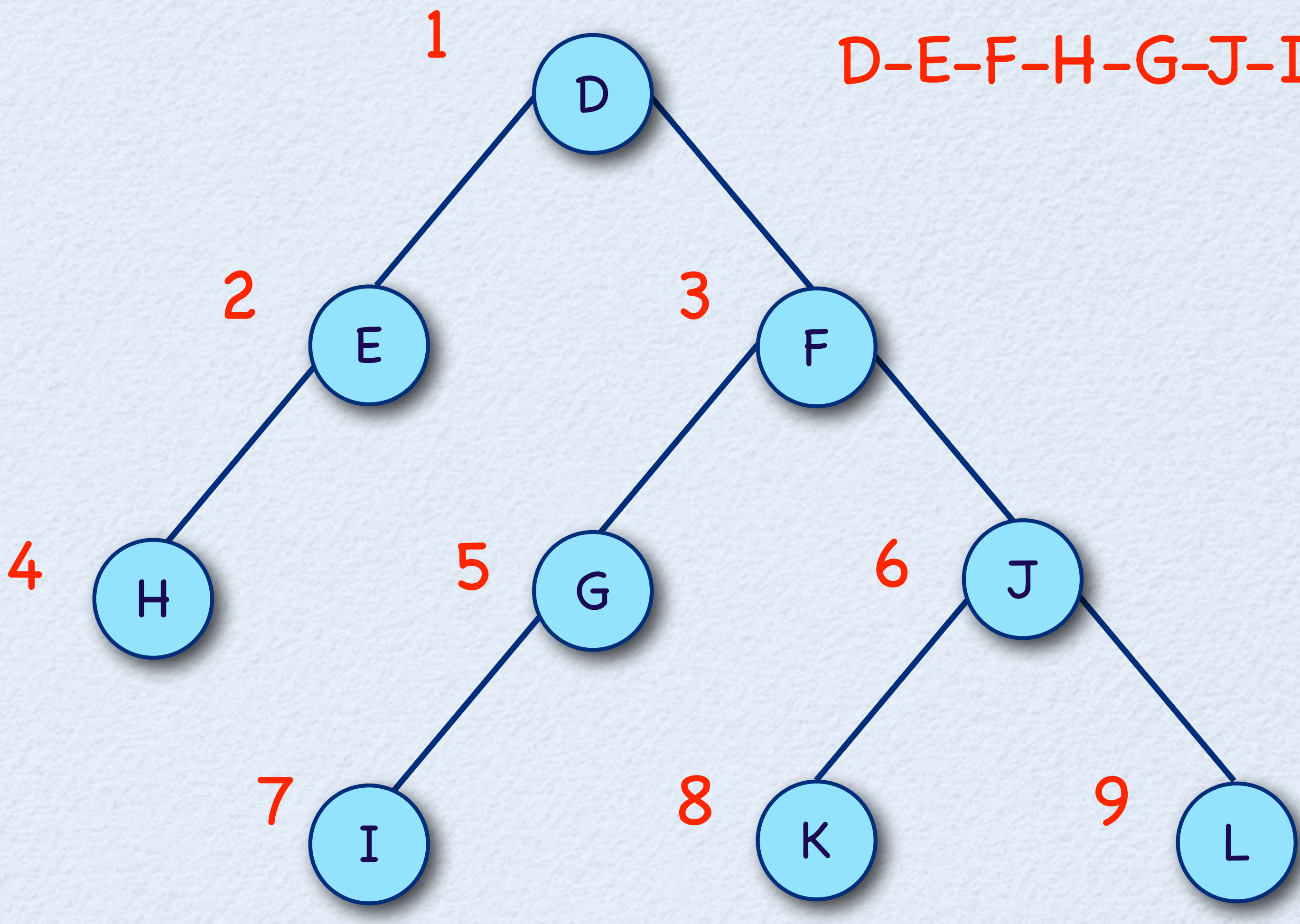
Breadth-first Traversal Example

Depth = 0:

Depth = 1:

Depth = 2:

Depth = 3:

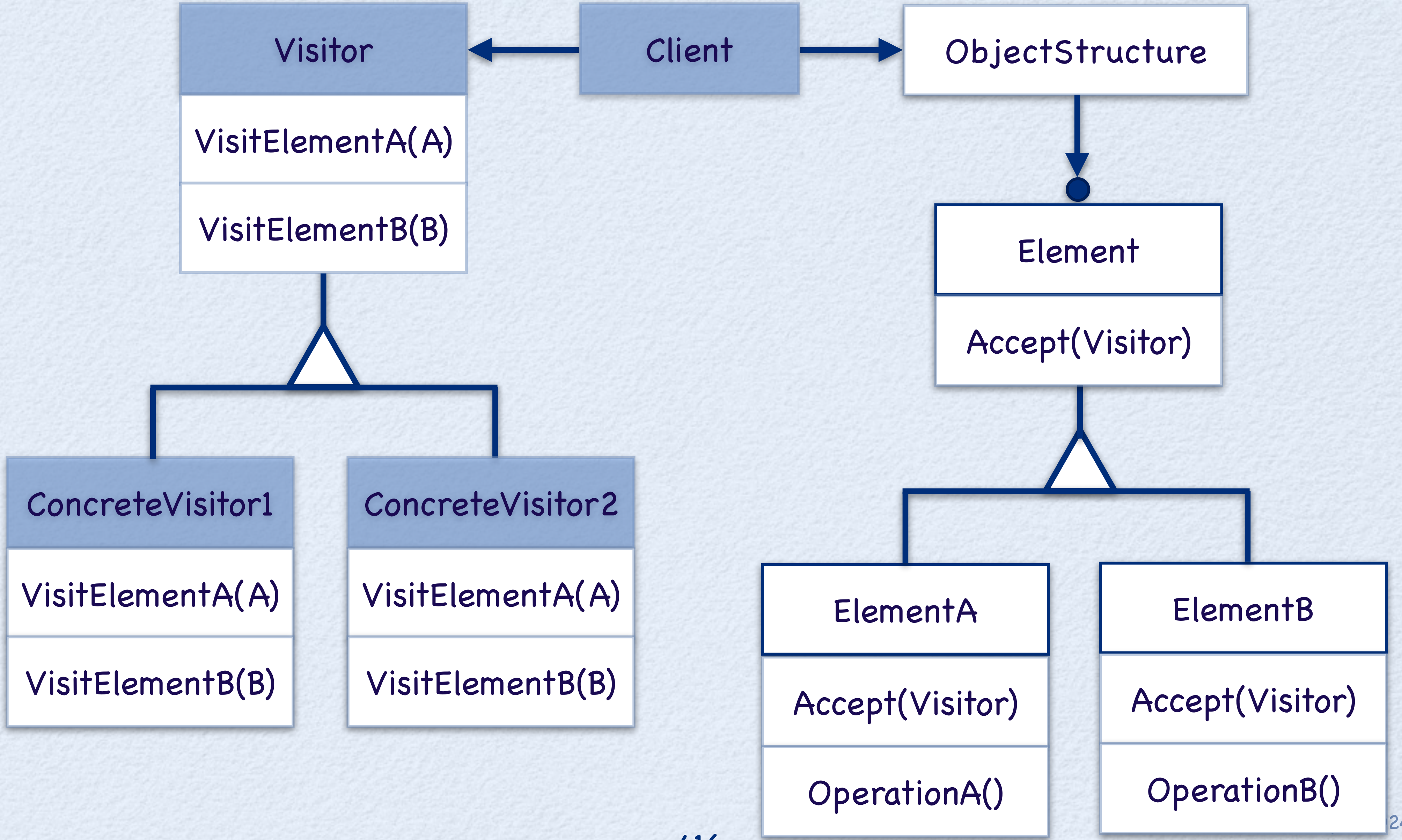


D-E-F-H-G-J-I-K-L

The Visitor Pattern

- Intent:
  - Represent an operation to be performed on the elements of an object structure. Visitor lets one define a new operation without changing the classes of the elements on which it operates.
- Collaborations:
  - A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
  - When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

Structure of Visitor



416

A Tree Visitor

TreeVisitor.h

```cpp
#include <iostream>

template<class T>
class TreeVisitor
{
public:
    virtual ~TreeVisitor() {}   // virtual default destructor

    // default behavior
    virtual void preVisit( const T& aKey ) const {}
    virtual void postVisit( const T& aKey ) const {}
    virtual void inVisit( const T& aKey ) const {}

    virtual void visit( const T& aKey ) const
    {
        std::cout << aKey << " ";
    }
};
```

Line: 2 Column: 23 C++ Tab Size: 4 TREEVISITOR_H_

PreOrderVisitor

TreeVisitor.h

```cpp

template<class T>
class PreOrderVisitor : public TreeVisitor<T>
{
public:

    // override pre-order behavior
    virtual void preVisit( const T& aKey ) const
    {
        this->visit( aKey );    // invoke default behavior
    }
};
```

Line: 58 Column: 1 C++ Tab Size: 4 inVisit

PostOrderVisitor

TreeVisitor.h

```cpp

template<class T>
class PostOrderVisitor : public TreeVisitor<T>
{
public:

    // override post-order behavior
    virtual void postVisit( const T& aKey ) const
    {
        this->visit( aKey );    // invoke default behavior
    }
};

```

Line: 58 Column: 1 C++ Tab Size: 4 inVisit

InOrderVisitor

TreeVisitor.h

```cpp

template<class T>
class InOrderVisitor : public TreeVisitor<T>
{
public:

    // override in-order behavior
    virtual void inVisit( const T& aKey ) const
    {
        this->visit( aKey );    // invoke default behavior
    }
};

```

Line: 42 Column: 6 C++ Tab Size: 4 postVisit

Depth-first Traversal for BTree

BTree.h

```cpp
    void searchDepthFirst( const TreeVisitor<T>& aVisitor ) const
    {
        if ( !empty() )
        {
            aVisitor.preVisit( **this );        // pre-order response
            left().searchDepthFirst( aVisitor );
            aVisitor.inVisit( **this );         // in-order response
            right().searchDepthFirst( aVisitor );
            aVisitor.postVisit( **this );       // post-order response
        }
    }
```

Line: 1 Column: 1 C++ Tab Size: 4

Main.cpp

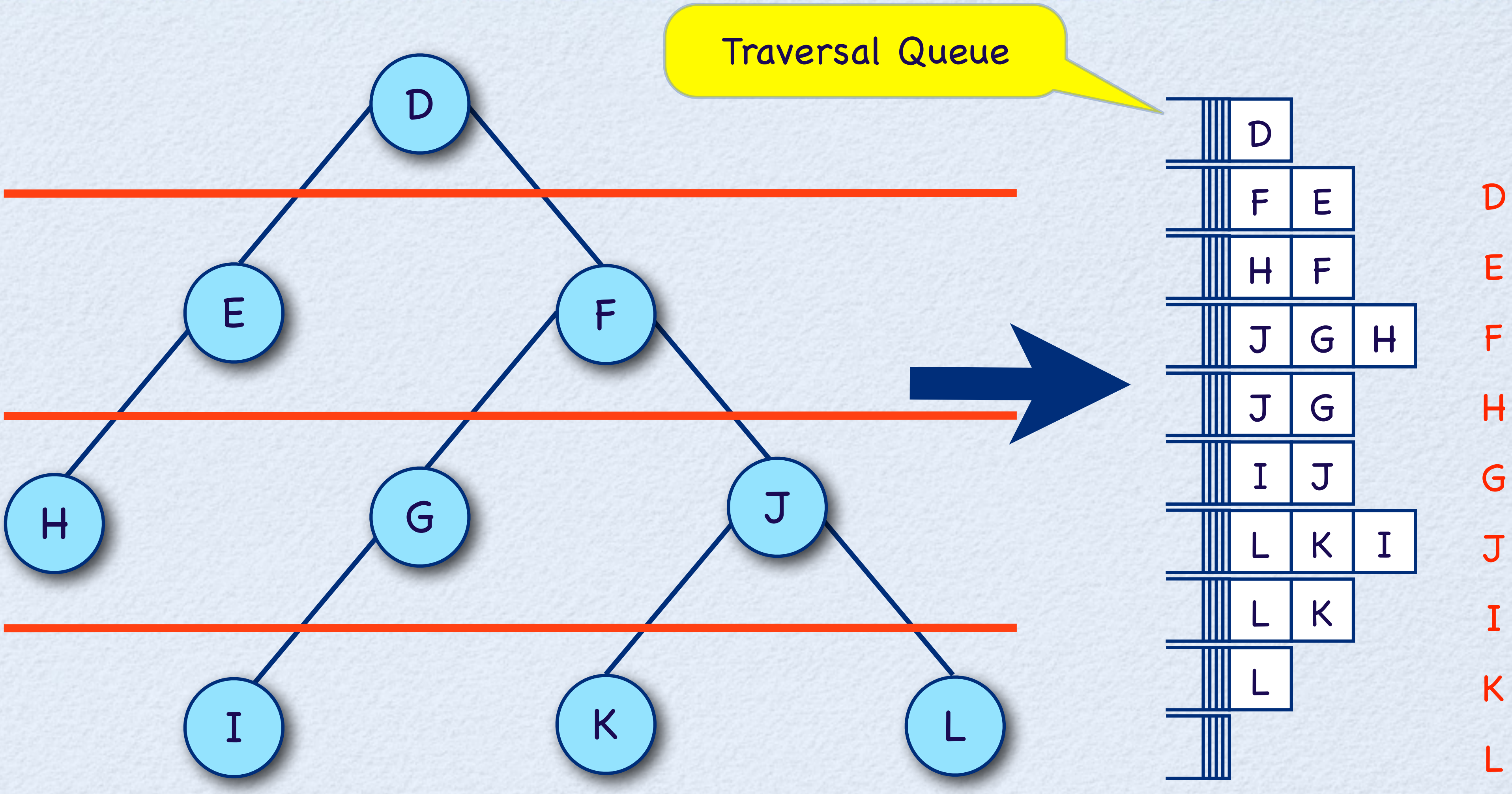
```cpp
#include "BTree.h"

void testDFS()
{
    cout << "Test DFS." << endl;

    using StringBTree = BTree<string>;

    string s1( "A" );
    string s2( "B" );
    string s3( "C" );

    StringBTree root( "Hello World!" );
    StringBTree nodeA( s1 );
    StringBTree nodeB( s2 );
    StringBTree nodeAA( s3 );
    StringBTree nodeBB( "D" );

    root.attachLeft( nodeA );
    root.attachRight( nodeB );
    const_cast<StringBTree&>(root.left()).attachLeft( nodeAA );
    const_cast<StringBTree&>(root.right()).attachRight( nodeBB );

    cout << "root:       " << *root << endl;
    cout << "root->L:    " << *root.left() << endl;
    cout << "root->R:    " << *root.right() << endl;
    cout << "root->L->L: " << *root.left().left() << endl;
    cout << "root->R->R: " << *root.right().right() << endl;

    root.searchDepthFirst( PreOrderVisitor<string>() );
    cout << endl;

    const_cast<StringBTree&>(root.right()).detachRight();
    const_cast<StringBTree&>(root.left()).detachLeft();
    root.detachRight();
    root.detachLeft();

    cout << "All trees are going to be deleted now!" << endl;
}
```

Line: 1 Column: 1 C++ Tab Size: 4

BTree

```
Kamala:BTree Markus$ ./BTreeTest
Test DFS.
root:       Hello World!
root->L:    A
root->R:    B
root->L->L: C
root->R->R: D
Hello World! A C B D
All trees are going to be deleted now!
```

Breadth-first Traversal Implementation



D-E-F-H-G-J-I-K-L

Breadth-first Traversal for BTree

BTree.h

```
    void searchBreadthFirst( const TreeVisitor<T>& aVisitor ) const
    {
        std::queue<const BTree<T>*> lQueue;     // traversal queue (pointer required)

        lQueue.push( this );                    // start with root node

        while ( !lQueue.empty() )
        {
            const BTree<T>* lFront = lQueue.front();

            if ( !lFront->empty() )
            {
                aVisitor.visit( **lFront );
                lQueue.push( &lFront->left() );
                lQueue.push( &lFront->right() );
            }

            lQueue.pop();
        }
    }
```

Line: 1 Column: 1 C++ Tab Size: 4 —

Main.cpp

```cpp
#include "BTree.h"

void testBFS()
{
    cout << "Test BFS." << endl;

    using StringBTree = BTree<string>;

    string s1( "A" );
    string s2( "B" );
    string s3( "C" );

    StringBTree root( "Hello World!" );
    StringBTree nodeA( s1 );
    StringBTree nodeB( s2 );
    StringBTree nodeAA( s3 );
    StringBTree nodeBB( "D" );

    root.attachLeft( nodeA );
    root.attachRight( nodeB );
    const_cast<StringBTree&>(root.left()).attachLeft( nodeAA );
    const_cast<StringBTree&>(root.right()).attachRight( nodeBB );

    cout << "root:       " << *root << endl;
    cout << "root->L:    " << *root.left() << endl;
    cout << "root->R:    " << *root.right() << endl;
    cout << "root->L->L: " << *root.left().left() << endl;
    cout << "root->R->R: " << *root.right().right() << endl;

    root.searchBreadthFirst( TreeVisitor<string>() );
    cout << endl;

    const_cast<StringBTree&>(root.right()).detachRight();
    const_cast<StringBTree&>(root.left()).detachLeft();
    root.detachRight();
    root.detachLeft();

    cout << "All trees are going to be deleted now!" << endl;
}
```

Line: 1 Column: 1 C++ Tab Size: 4

BTree

```
Kamala:BTree Markus$ ./BTreeTest
Test BFS.
root:       Hello World!
root->L:    A
root->R:    B
root->L->L: C
root->R->R: D
Hello World! A B C D
All trees are going to be deleted now!
```

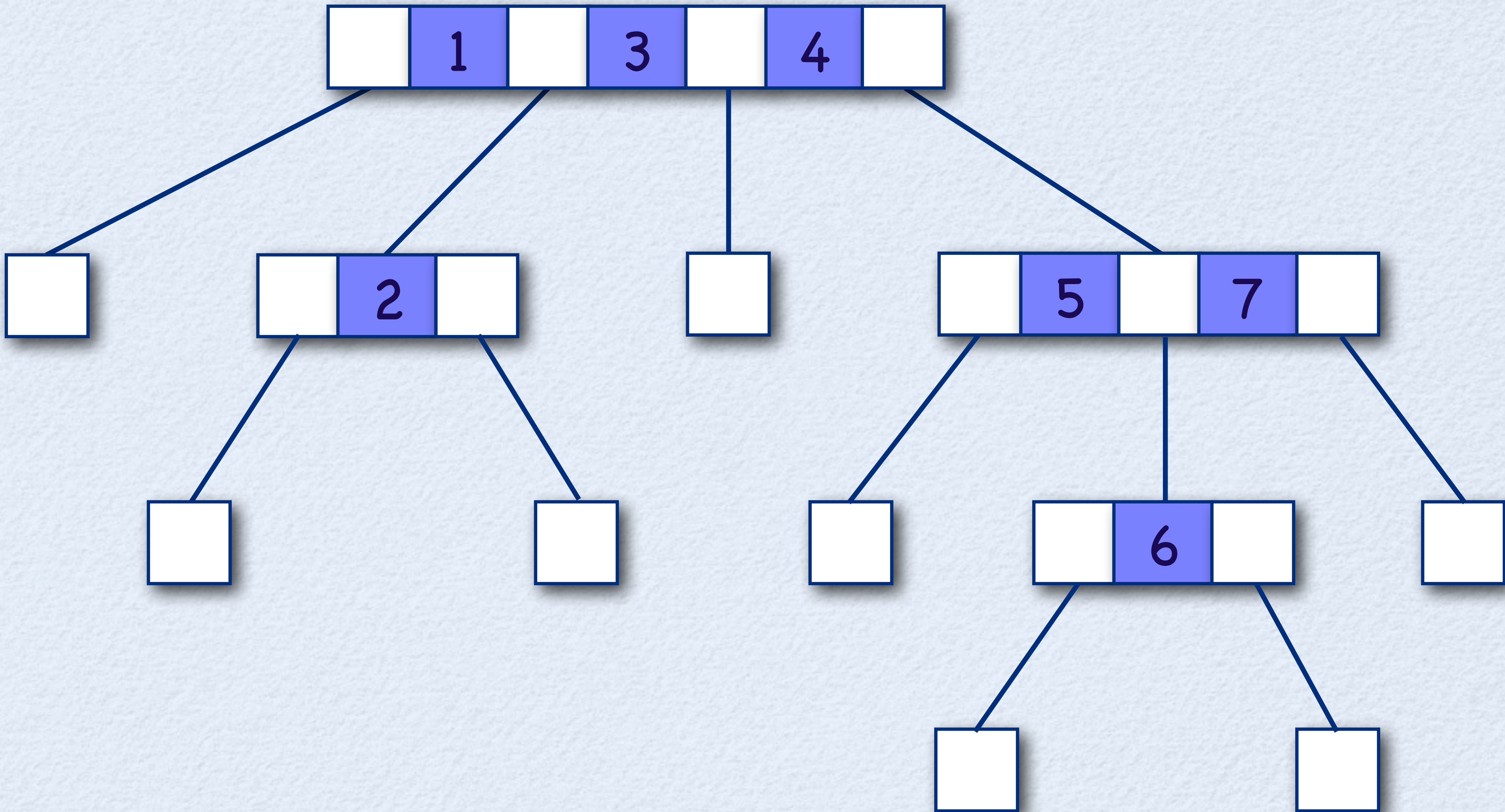
M-way Search Tree

- An M-ary search tree T is a finite set of nodes with one of the following properties:
  - either the set is empty, $T = \emptyset$, or
  - for $2 \leq n \leq M$, the set consists of n M-ary subtrees $T_1, T_2, \ldots, T_{n-1}, T_n$ and $n-1$ keys $k_1, k_2, \ldots, k_{n-1}$.

and the keys and nodes satisfy the data ordering properties:

- The keys in each node are distinct and ordered, i.e., $k_i < k_{i+1}$, for $1 \leq i \leq n-1$.
- All the keys contained in subtree T_{i-1} are less than k_i, i.e., $\forall k \in T_{i-1}: k < k_i$, for $1 \leq i \leq n-1$. The tree T_{i-1} is called left subtree with respect the key k_i.
- All the keys contained in subtree T_i are greater than k_i, i.e., $\forall k \in T_i: k > k_i$, for $1 \leq i \leq n-1$. The tree T_i is called right subtree with respect the key k_i.

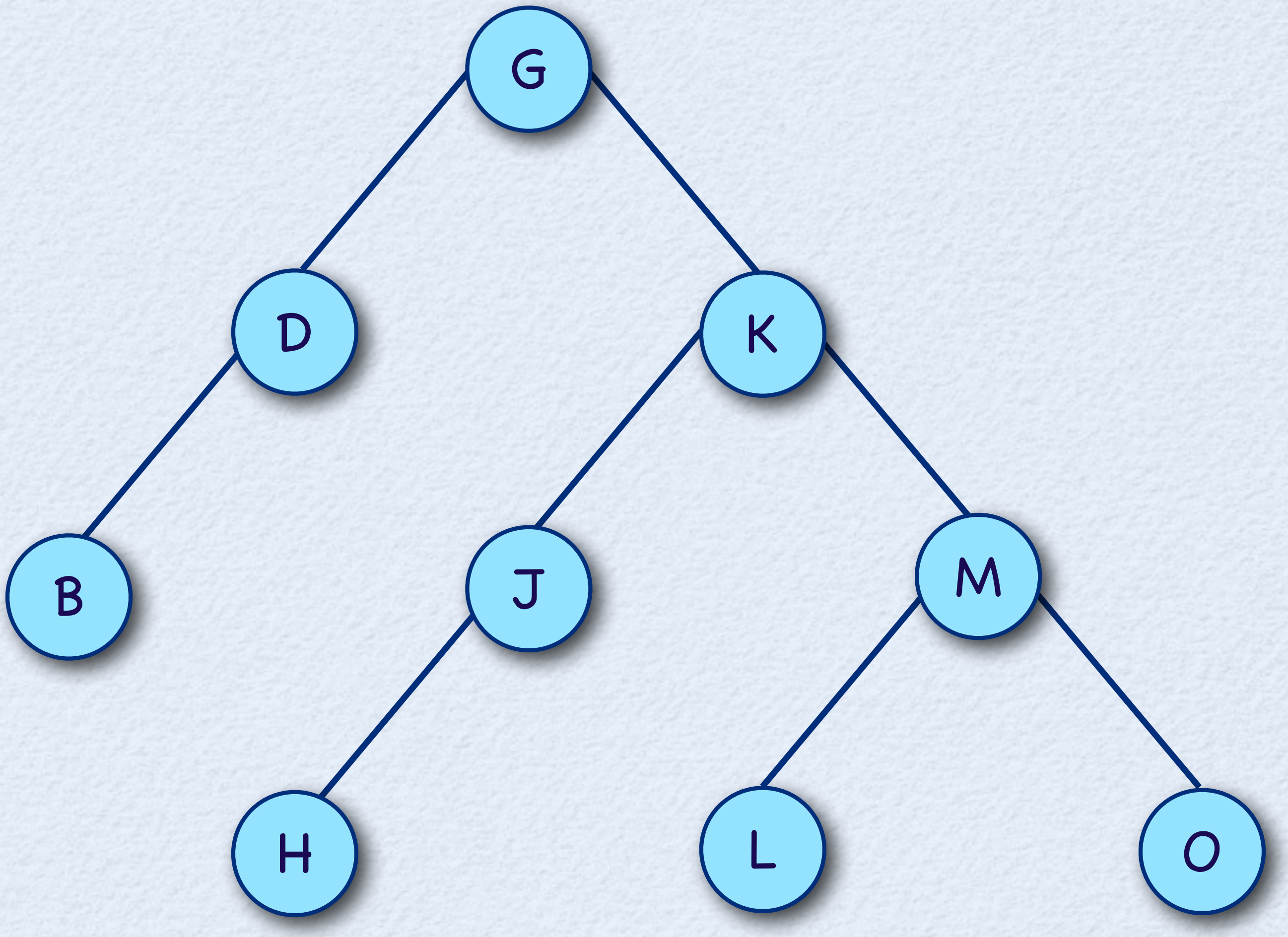
A 4-way Search Tree



2-way Search Tree

- A 2-ary (binary) search tree T is a finite set of nodes with one of the following properties:
  - either the set is empty, $T = \emptyset$, or
  - the set consists of one key, r, and exactly 2 binary subtrees T_L and T_R such that following properties are satisfied:
    - All the keys in the left subtree, T_L, are less than r, i.e., $\forall k \in T_L: k < r$.
    - All the keys contained in the right subtree, T_R, are greater than r, i.e., $\forall k \in T_R: k > r$.

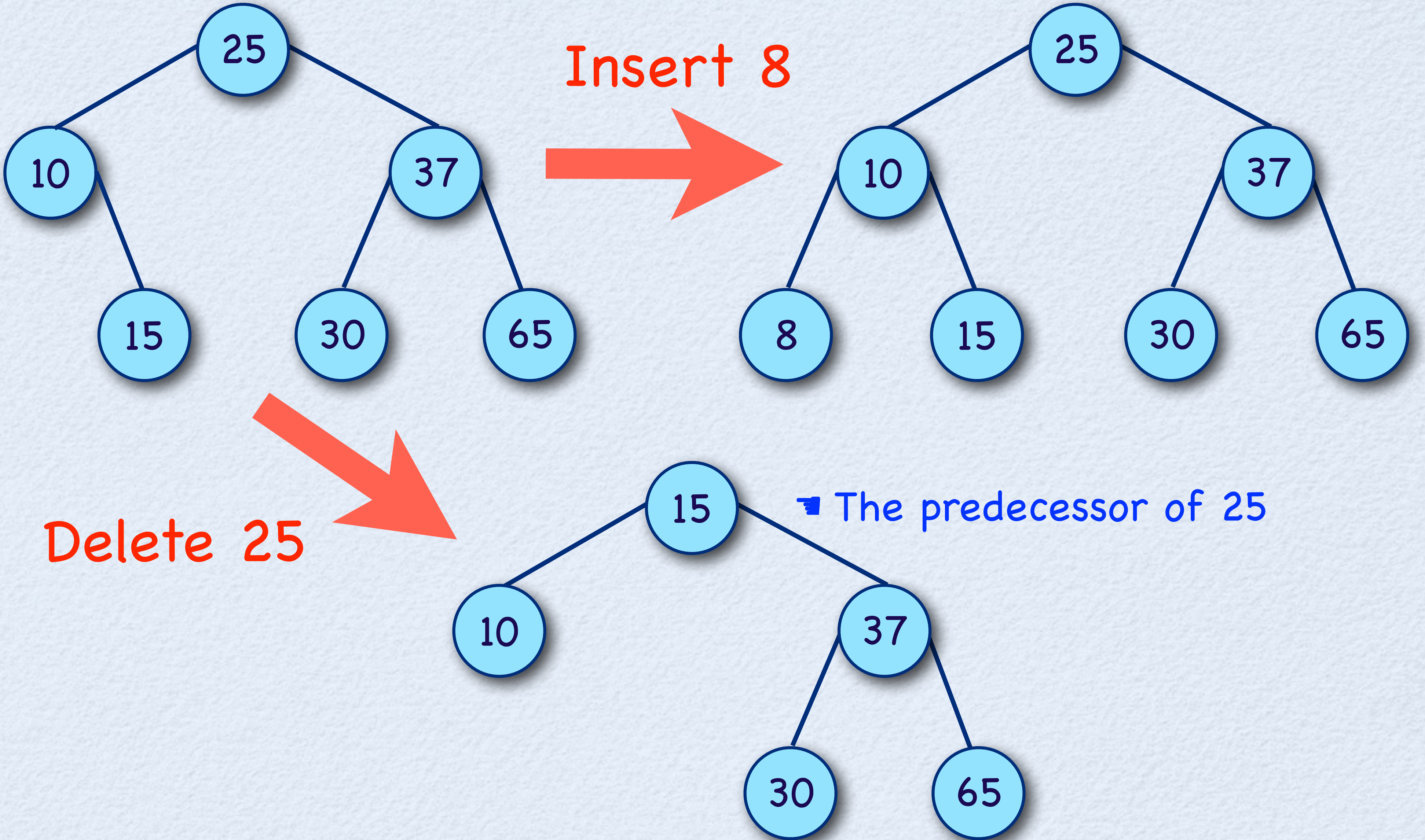
A Binary Search Tree Example



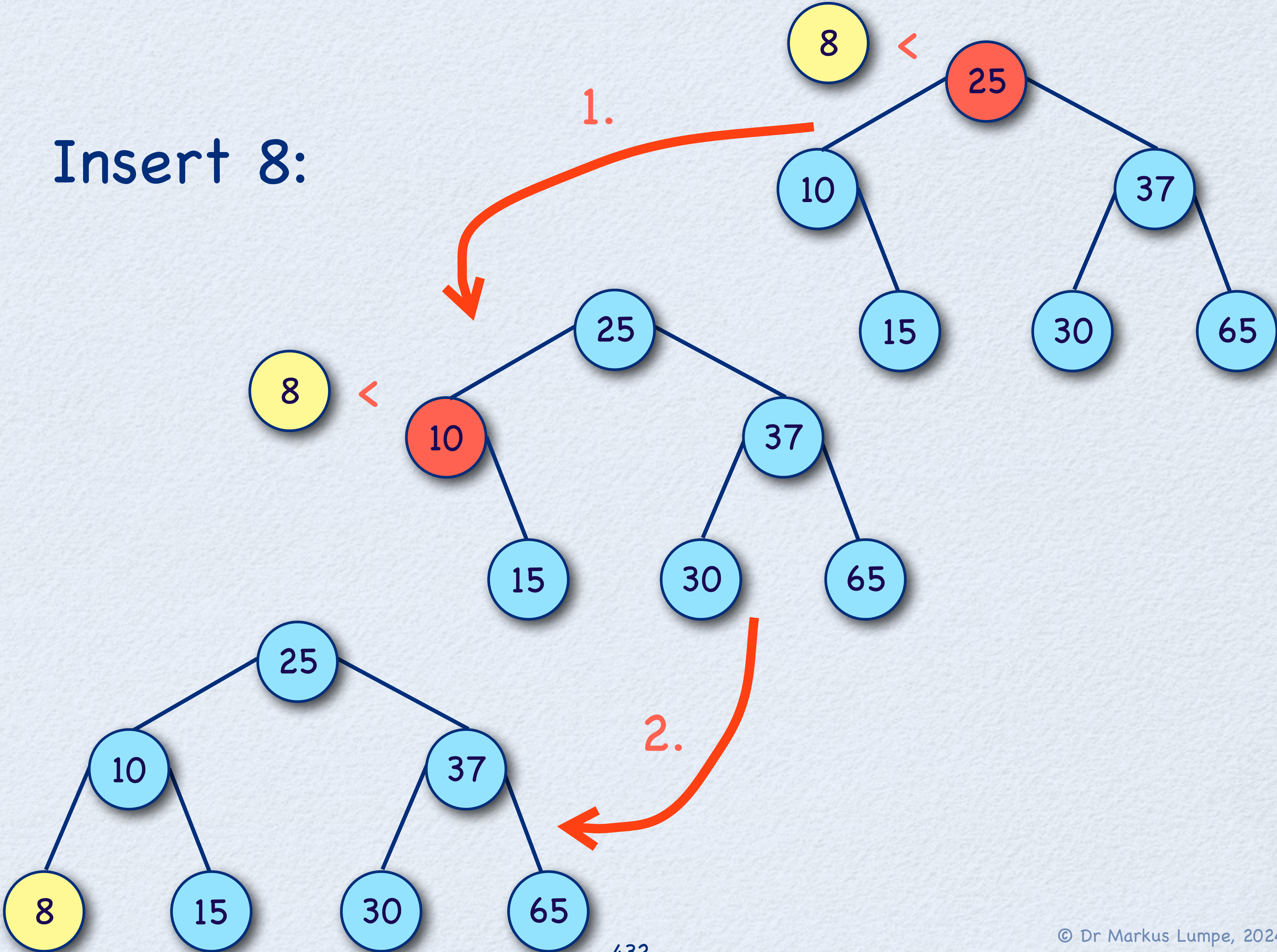
Traversing a Binary Search Tree

- Binary Tree Search:
  - Traverse the left subtree, and then
  - Visit the root, and then
  - Traverse the right subtree.
- We use in-order traversal to search for a given key in an M-ary search tree.

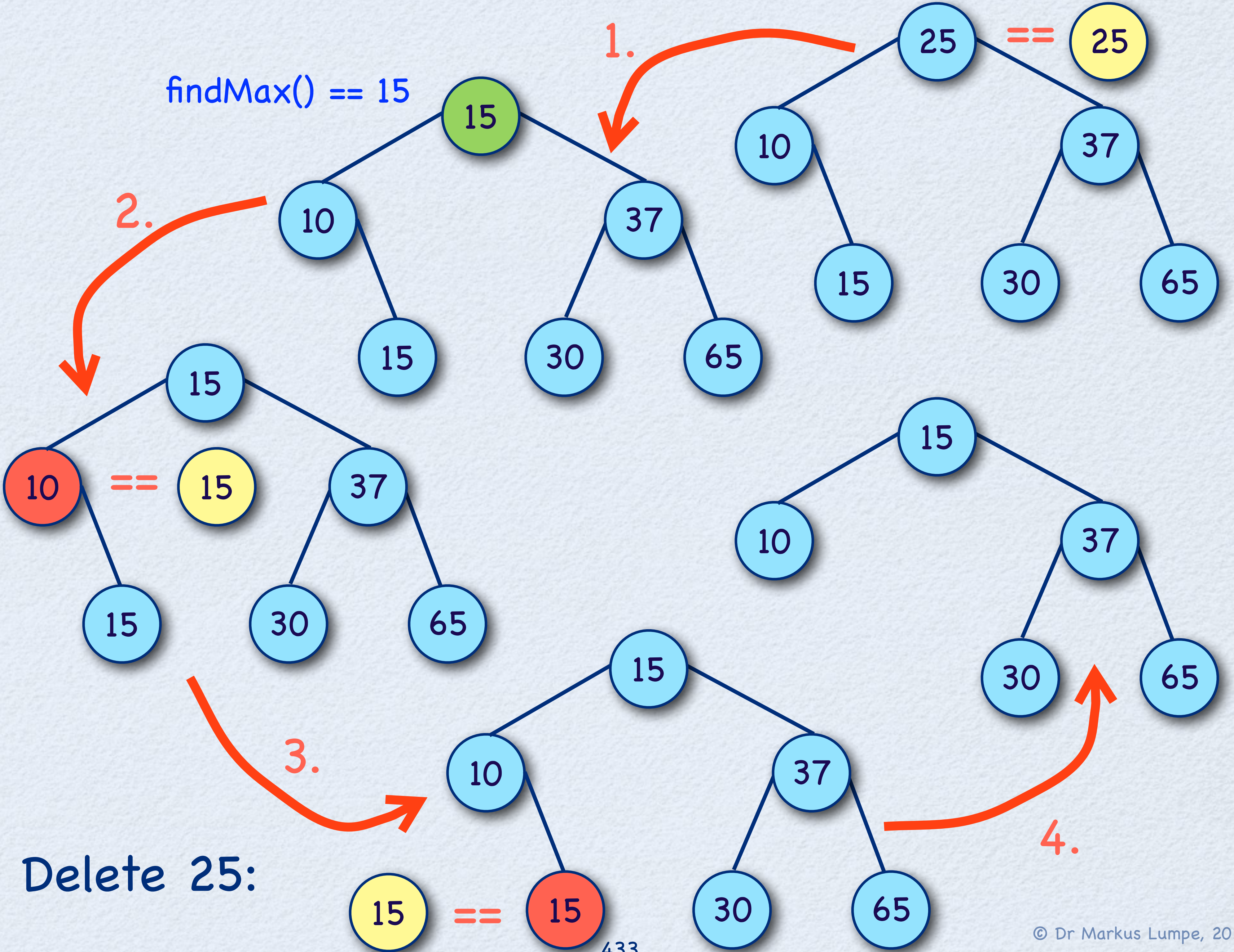
Binary Search Tree Operations



Insert 8:



Delete 25:



We define the representation of a binary search tree in class BNode.

BNode.h

```
#include <stdexcept>

template<typename S>
struct BNode
{
    S key;
    BNode<S>* left;
    BNode<S>* right;

    static BNode<S> NIL;

    ...
};

template<typename S>
BNode<S> BNode<S>::NIL;
```

Line: 3 Column: 1 C++ Tab Size: 4

BinarySearchTree.h

```
#include "BNode.h"
#include "TreeVisitor.h"

template<typename T>
class BinarySearchTree
{
private:
    BNode<T>* fRoot;

public:

    BinarySearchTree();
    ~BinarySearchTree();

    bool empty() const;
    size_t height() const;

    bool insert( const T& aKey );
    bool remove( const T& aKey );

    void traverseDepthFirst( const TreeVisitor<T>& aVisitor ) const;
};
```

Line: 3 Column: 1 C++ Tab Size: 4

BNode.h

```cpp
    bool insert( const S& aKey )
    {
        BNode<S>* x = this;
        BNode<S>* y = &BNode<S>::NIL;

        while ( !x->empty() )
        {
            y = x;

            if ( aKey == x->key )
            {
                return false;           // duplicate key - error
            }

            x = aKey < x->key ? x->left : x->right;
        }

        BNode<S>* z = new BNode<S>( aKey );

        if ( y->empty() )
        {
            return false;           // insert failed (NIL)
        }
        else
        {
            if ( aKey < y->key )
            {
                y->left = z;
            }
            else
            {
                y->right = z;
            }
        }

        return true;                // insert succeeded
    }
```

Line: 129 Column: 1 C++ Tab Size: 4 remove

Remove and Depth-first Traversal

BinarySearchTree.h — BTree

```cpp
    bool remove( const T& aKey )
    {
        bool lLeaf = fRoot->leaf();

        if ( fRoot->remove( aKey, &BNode<T>::NIL ) )
        {
            if ( lLeaf )
            {
                fRoot = &BNode<T>::NIL;
            }

            return true;
        }

        return false;
    }

    void traverseDepthFirst( const TreeVisitor<T>& aVisitor ) const
    {
        fRoot->traverseDepthFirst( aVisitor );
    }
```

Line: 60:10 C++ Tab Size: 4 insert

- The class BSTree defines an adapter for BSTNode.
- The operation insert can be defined as a simple while-loop over BSTNodes from the root node.
- The operations remove and traverseDepthFirst use recursion to explore BSTNodes. In the beginning, both start with the root node.

BNode.h

```cpp
    bool remove( const S& aKey, BNode<S>* aParent )
    {
        BNode<S>* x = this;
        BNode<S>* y = aParent;

        while ( !x->empty() )
        {
            if ( aKey == x->key )
                break;

            y = x;                                          // new parent
            x = aKey < x->key ? x->left : x->right;
        }

        if ( x->empty() )
            return false;                                   // delete failed

        if ( !x->left->empty() )
        {
            const S& lKey = x->left->findMax();             // find max to left
            x->key = lKey;
            x->left->remove( lKey, x );
        }
        else
        {
            if ( !x->right->empty() )
            {
                const S& lKey = x->right->findMin();    // find min to right
                x->key = lKey;
                x->right->remove( lKey, x );
            }
            else
            {
                if ( y->left == x )
                    y->left = &NIL;
                else
                    y->right = &NIL;

                delete x;                                   // free deleted node
            }
        }

        return true;
    }
```

Line: 129 Column: 1 C++ Tab Size: 4 remove

Binary Search Tree Test

Main.cpp

```cpp
#include "BinarySearchTree.h"

void testVisitor()
{
    BinarySearchTree<int> lTree;

    lTree.insert( 25 );
    lTree.insert( 10 );
    lTree.insert( 15 );
    lTree.insert( 37 );
    lTree.insert( 30 );
    lTree.insert( 65 );

    lTree.traverseDepthFirst( PreOrderVisitor<int>() );
    cout << endl;

    lTree.insert( 8 );

    lTree.traverseDepthFirst( PreOrderVisitor<int>() );
    cout << endl;

    lTree.remove( 25 );

    lTree.traverseDepthFirst( PreOrderVisitor<int>() );
    cout << endl;
}
```

Line: 1 Column: 1 C++ Tab Size: 4 —

COS30008

```
Kamala:COS30008 Markus$ ./BSTreeTest
25 10 15 37 30 65
25 10 8 15 37 30 65
15 10 8 37 30 65
Kamala:COS30008 Markus$
```

Other Tree Variants

- Rose Trees (directories)
- Expression Trees (internal program representation)
- Multi-rooted trees (C++: multiple inheritance)
- Red-Black Trees (directories in compound documents, java.util.TreeMap)
- AVL Trees (Adelson-Velskii & Landis balanced BTrees)

AVL vs. Red-Black Trees

- Both AVL trees and Red-Black trees are self-balancing binary search trees. However, the operations to balance the trees are different.
- AVL and Red-Black trees have different height limits. For a tree of size n:
    - An AVL tree's height is limited to $1.44\log_2(n)$.
    - A Red-Black tree's height is limited to $2\log_2(n)$.
- The AVL tree is more rigidly balanced than Red-Black trees, leading to slower insertion and removal but faster retrieval.