TREE TRAVERSAL

- Many different algorithms for manipulating trees exist, and they 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 sub-trees of the root one-by-one in the order given (e.g., from left to right).
- Post-order traversal:
  - Do a postorder traversal of each of the sub-trees 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 sub-tree first, visit the root, and then traverse the right subtree.

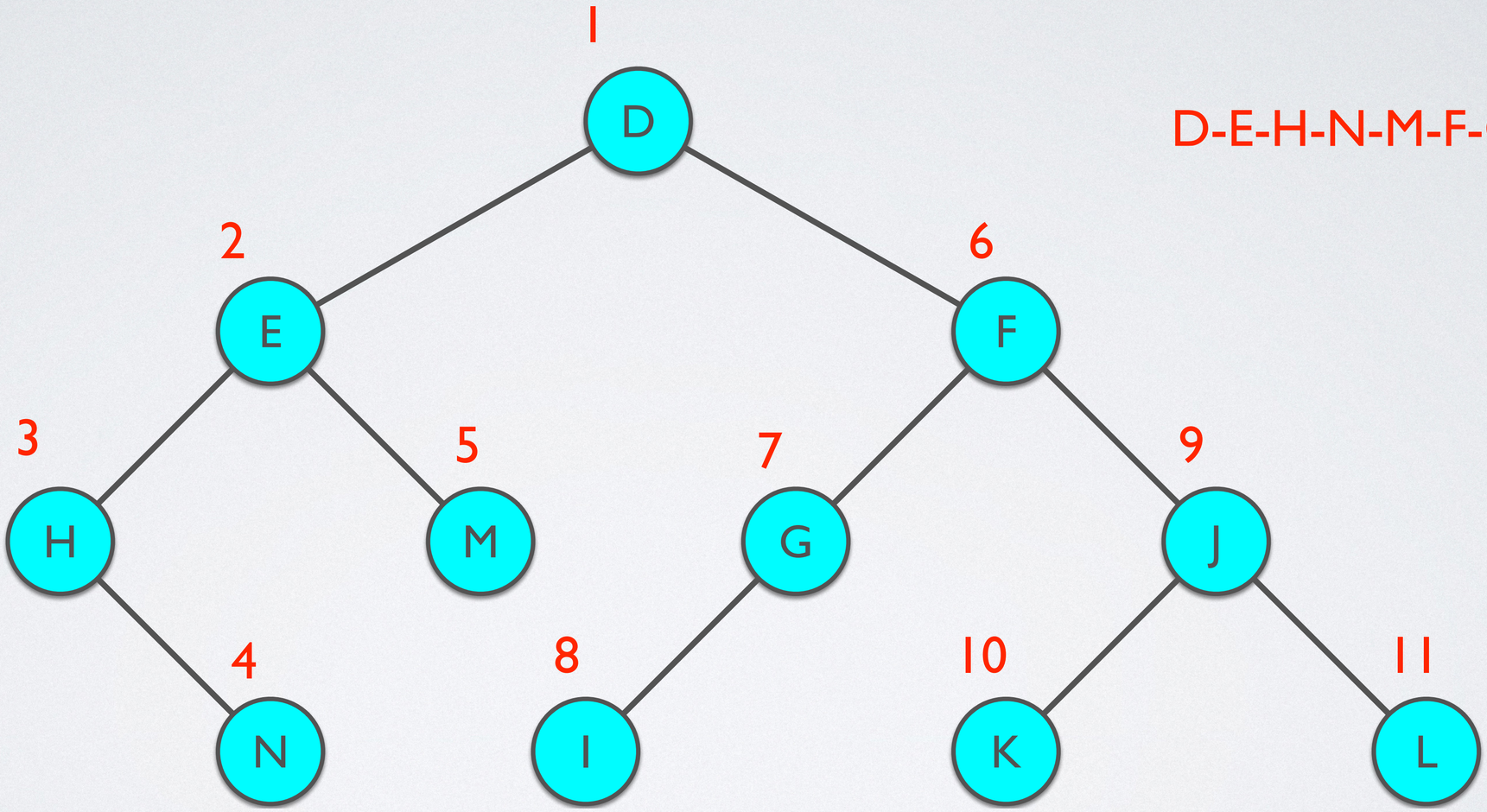
PRE-ORDER TRAVERSAL

Depth = 0:

Depth = 1:

Depth = 2:

Depth = 3:



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

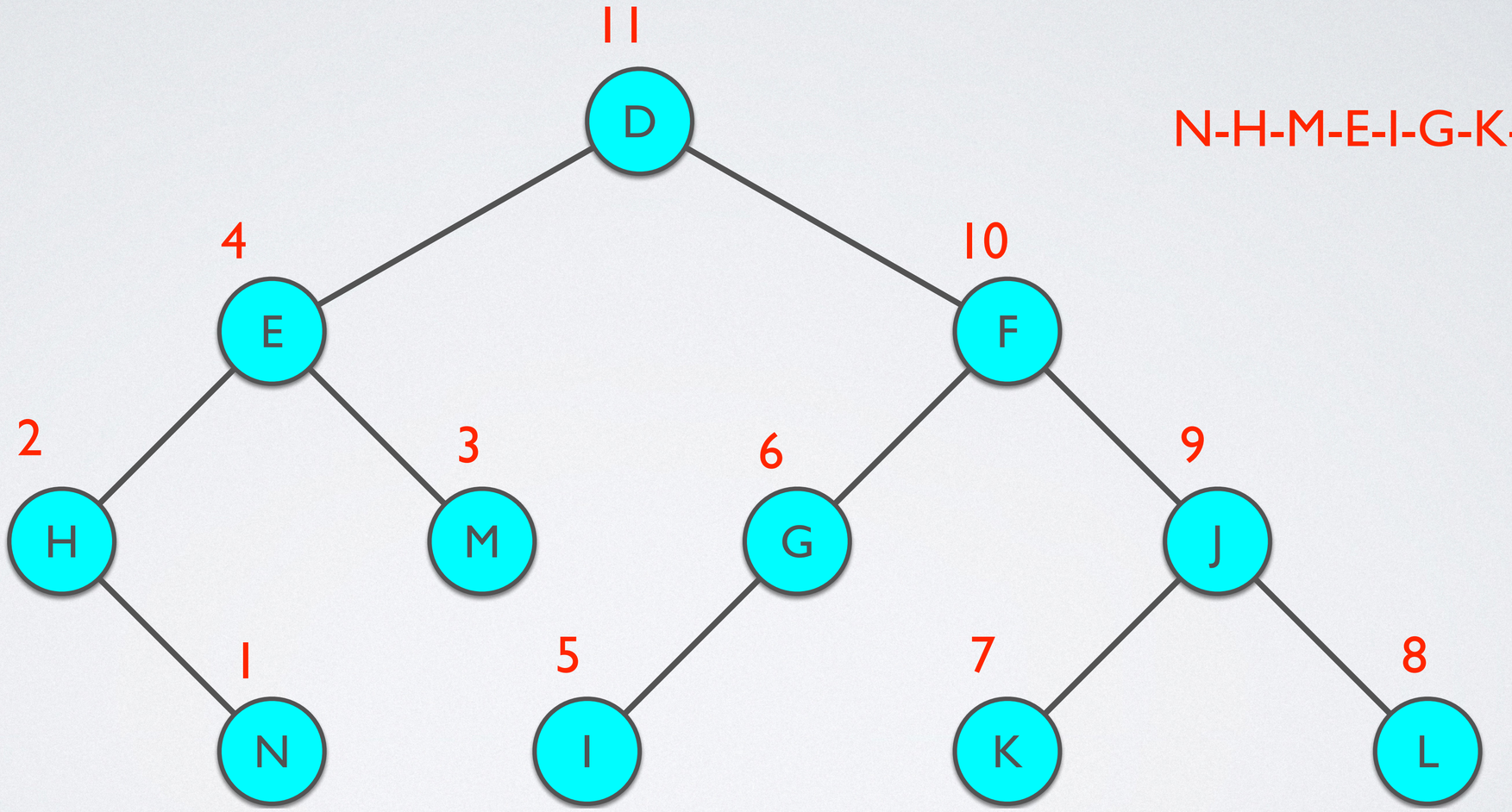
POST-ORDER TRAVERSAL

Depth = 0:

Depth = 1:

Depth = 2:

Depth = 3:



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

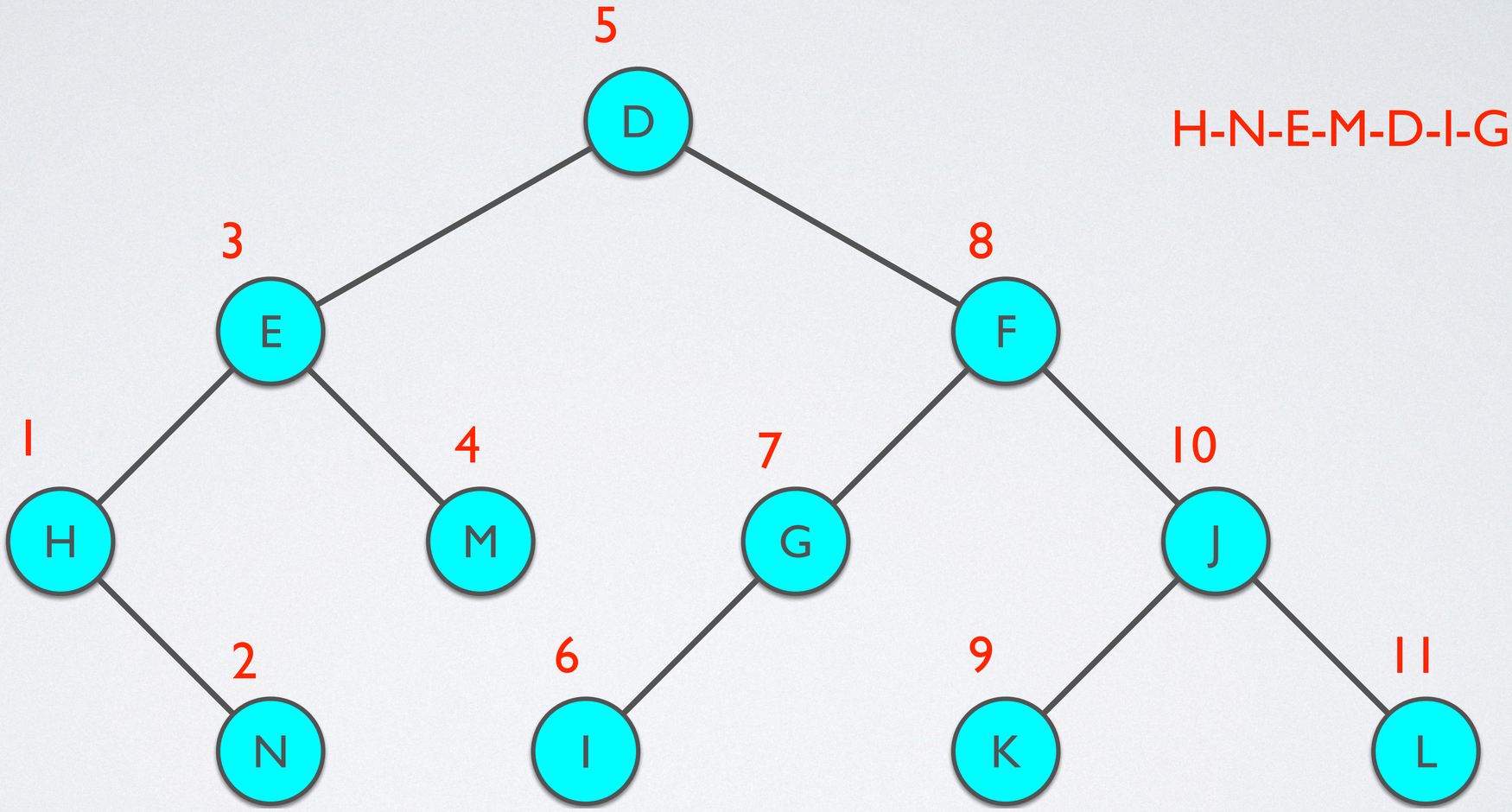
IN-ORDER TRAVERSAL

Depth = 0:

Depth = 1:

Depth = 2:

Depth = 3:



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

BREADTH-FIRST TRAVERSAL

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

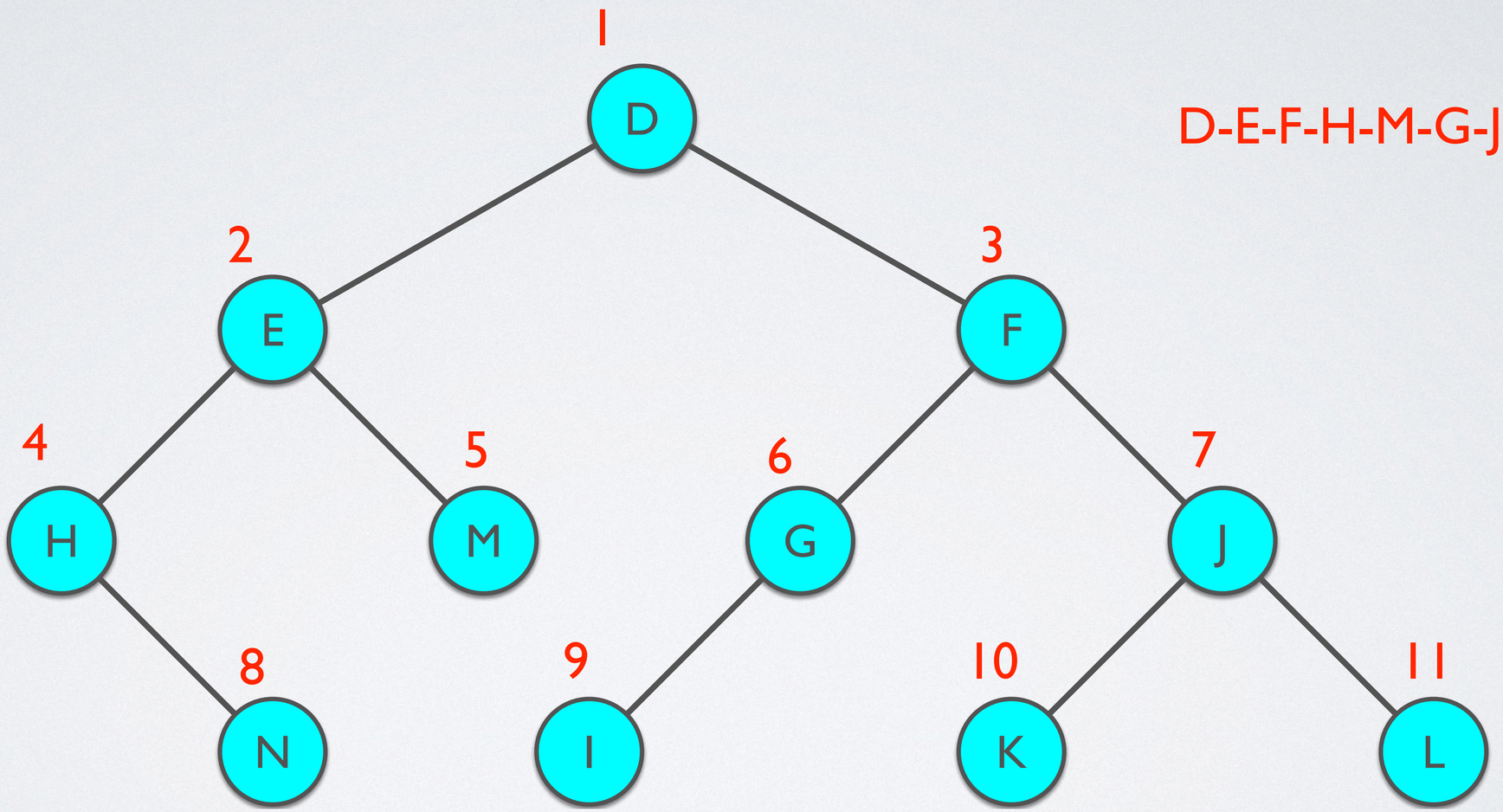
BREADTH-FIRST TRAVERSAL

Depth = 0:

Depth = 1:

Depth = 2:

Depth = 3:



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

TRAVERSAL IMPLEMENTATION

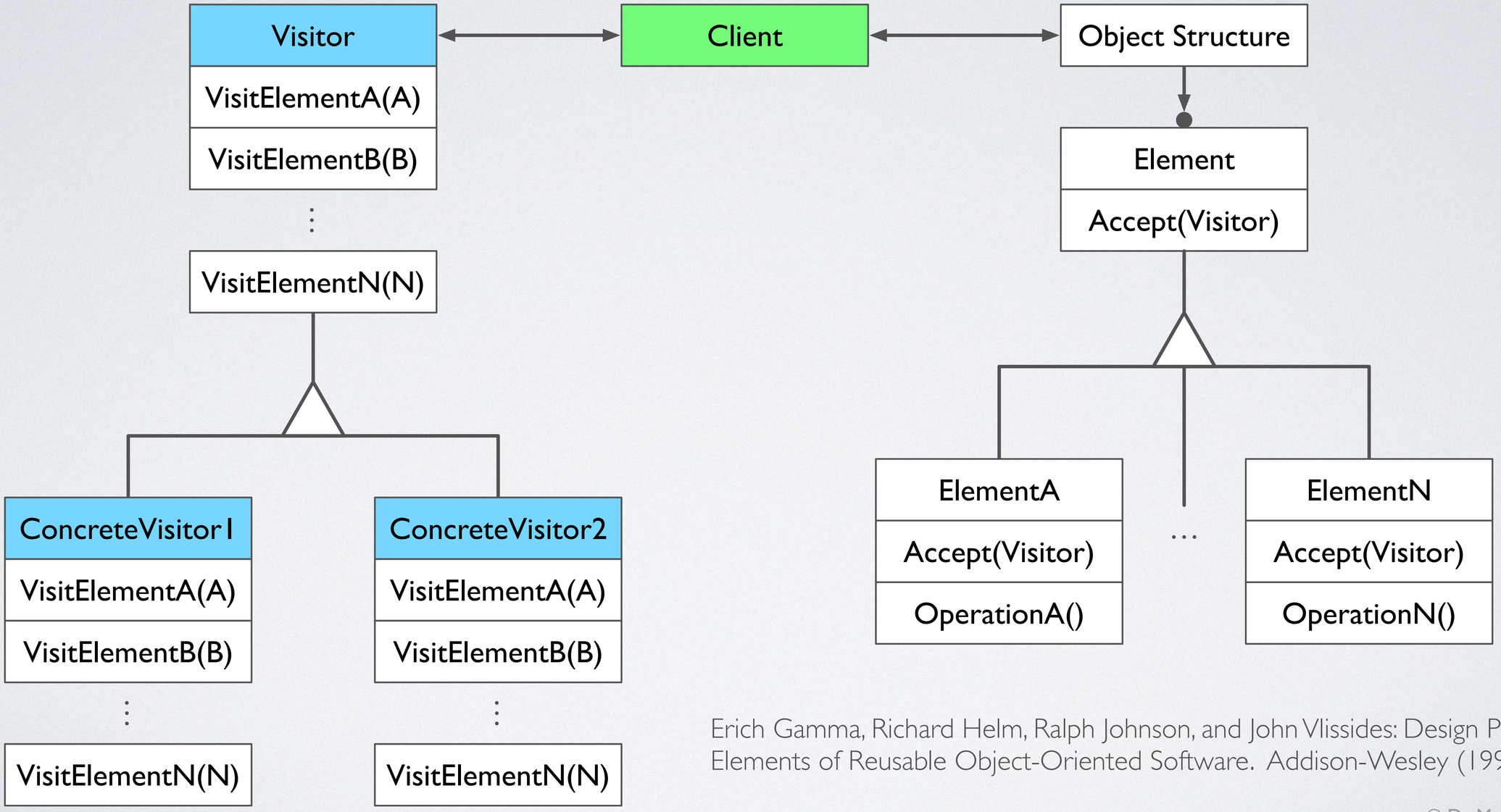
DESIGN CONSIDERATIONS

- Tree traversal requires a systematic approach to visit/access all nodes in a tree (we restrict the analysis to binary trees only).
- It is possible to define a recursive solution to implement tree traversal. A known solution employs the Visitor Pattern.
- Iterators offer an alternative solution. An iterator-based solution maps the traversal to a linear process:
  - Depth-first search traversal:
    - Uses a stack of frontiers nodes to yield the next tree node.
  - Breadth-first search traversal:
    - Uses a queue of nodes to yield the next tree node.

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 corresponding 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 PATTERN



Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)

A TREE VISITOR

```
#include <iostream>

template<typename T>
class TreeVisitor
{
public:
    virtual ~TreeVisitor() {}

    virtual void preVisit( const T& aKey ) const noexcept {}
    virtual void postVisit( const T& aKey ) const noexcept {}
    virtual void inVisit( const T& aKey ) const noexcept {}

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

default depth-first search traversal

default breadth-first search traversal

PREORDER VISITOR

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

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

override default behavior

POSTORDER VISITOR

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

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

override default behavior

INORDER VISITOR

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

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

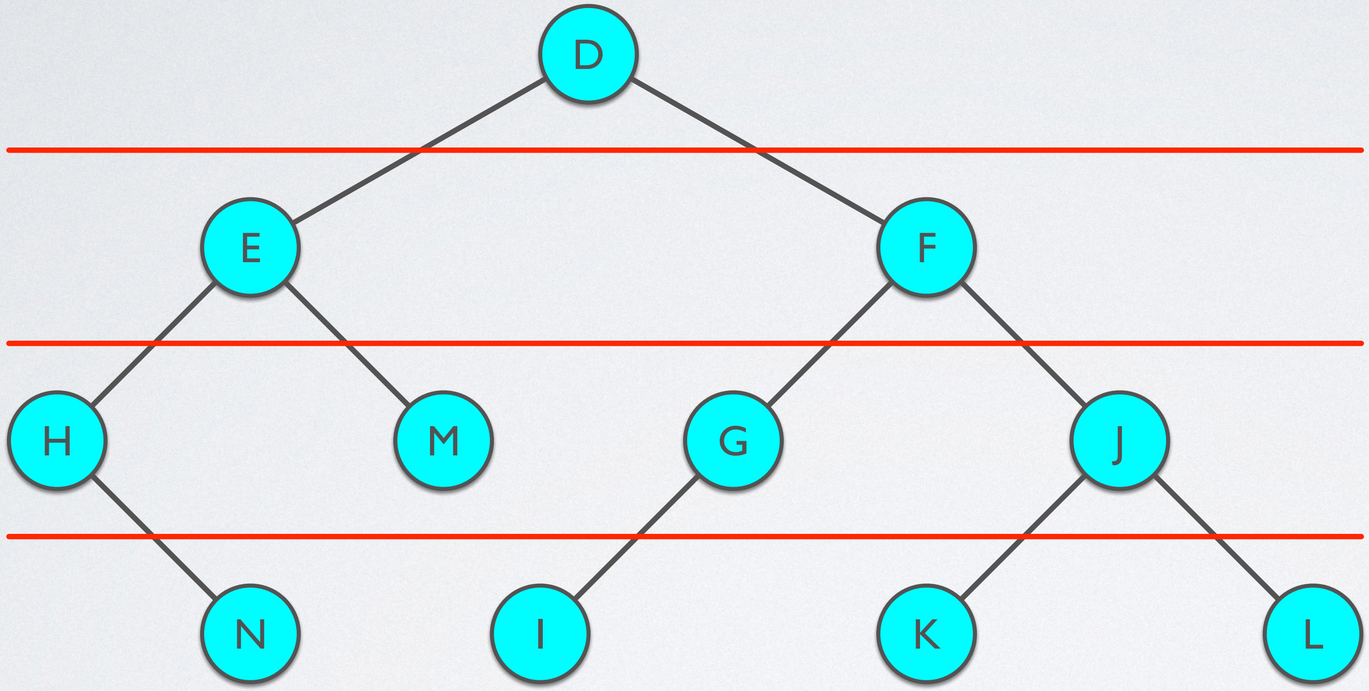
override default behavior

DEPTH-FIRST SEARCH TRAVERSAL FOR BTREE

```
void performDepthFirstSearch( const TreeVisitor<T>& aVisitor ) const noexcept
{
    aVisitor.preVisit( **this );

    if ( hasLeft() )
    {
        left().performDepthFirstSearch( aVisitor );
    }

    aVisitor.inVisit( **this );

    if ( hasRight() )
    {
        right().performDepthFirstSearch( aVisitor );
    }

    aVisitor.postVisit( **this );
}
```

Pre-Order:	**DEHNMFGIJKL**
Post-Order:	**NHMEIGKLJFD**
In-Order:	**HNEMDIGFKJL**

BREADTH-FIRST SEARCH WITH QUEUE



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

BREADTH-FIRST SEARCH TRAVERSAL FOR BTREE

```
void performBreadthFirstSearch( const TreeVisitor<T>& aVisitor ) const noexcept
{
    std::queue<const BTree*> lQueue;    // we use raw pointers here (no ownership control)

    lQueue.push( this );

    while ( !lQueue.empty() )
    {
        const BTree& lFront = *lQueue.front();      // L-value reference to front

        lQueue.pop();
        aVisitor.visit( *lFront );

        if ( lFront.hasLeft() )
        {
            lQueue.push( &lFront.left() );
        }

        if ( lFront.hasRight() )
        {
            lQueue.push( &lFront.right() );
        }
    }
}
```

Order: **DEFHMGJNIKL**

ITERATOR PLANNING

BREATH-FIRST SEARCH ITERATOR

```
template<typename T>
class BreadthFirstTraversal
{
public:
    using pointer = const BTree<T>*;
    using iterator = BreadthFirstTraversal<T>;
    using node = BTree<T>::Node;
    using difference_type = std::ptrdiff_t;          // to satisfy concept weakly_incrementable
    using value_type = T;                            // to satisfy concept indirectly_readable

    BreadthFirstTraversal( const node& aBtree );

    const T& operator*() const noexcept;

    iterator& operator++();
    iterator operator++(int);

    bool operator==( const iterator& aOther ) const noexcept;

    iterator begin() const noexcept;
    iterator end() const noexcept;

private:
    pointer fRoot;
    std::queue<pointer> fQueue;
};
```

type aliases

forward iterator

binary tree root

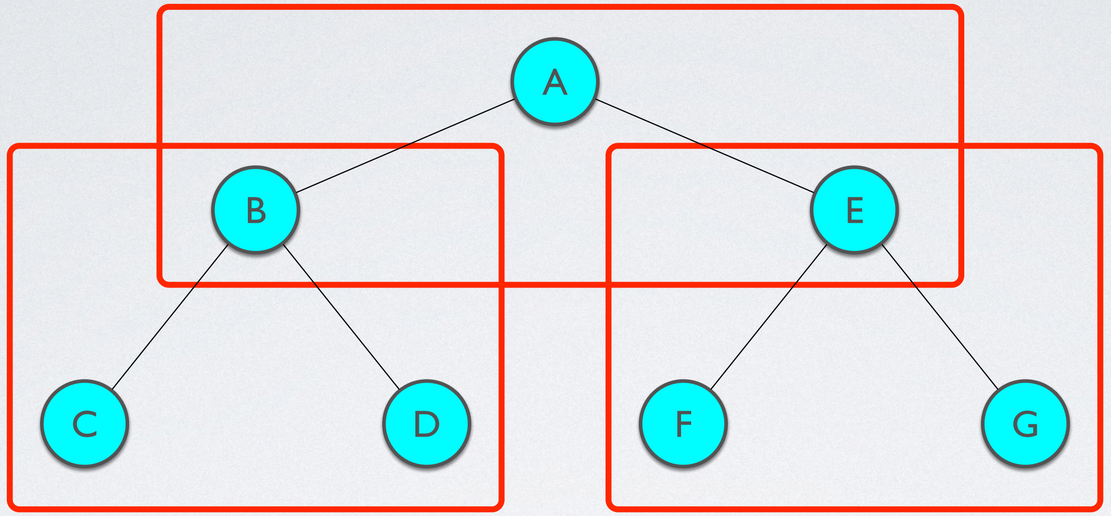
iterator queue

FRONTIER

```
template<typename T>
struct Frontier
{
    bool mustExploreRight;
    T node;

    Frontier( T aNode = nullptr ) :
        mustExploreRight(true),
        node(aNode)
    {}

    bool operator==( const Frontier& aOther ) const noexcept
    {
        return
            mustExploreRight == aOther.mustExploreRight &&
            node == aOther.node;
    }
};
```

- A frontier is an auxiliary type to record traversal progress.
- A frontier abstraction enables traversal tasks to explore multiple paths in a specific sequence.
- A frontier allows the same node to be analyzed as the top node and update this node's traversal state.
- Post-order traversal relies on the existence of a frontier abstraction. It prevents premature removal of nodes from the stack. Once the right sub-tree is marked explored, the top node can be removed from the stack.

PRE-ORDER WITH STACK



has right - left? has right - left? has right - left? has right - left? has right - left? has right - left? has right - left?

Pre-Order: A B C D E F G

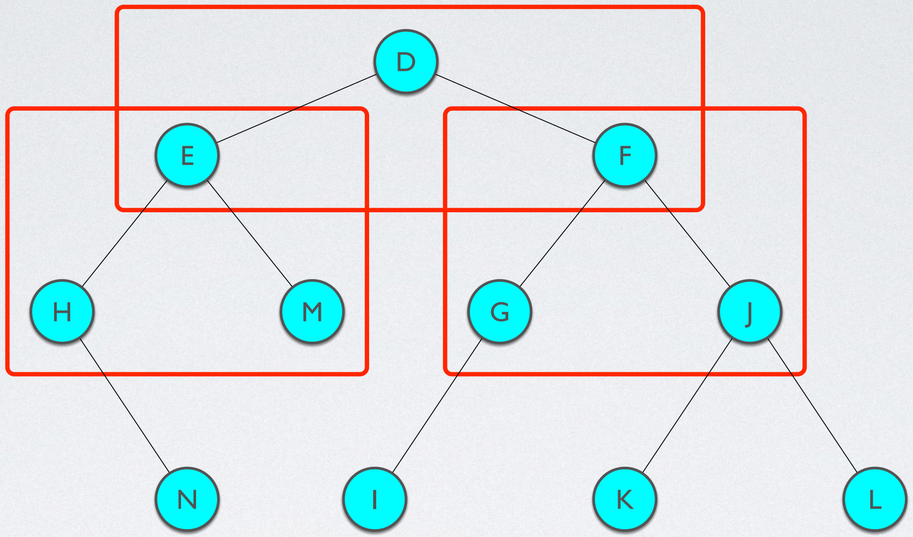
Stack:

A	B	C	D	E	F	G
	E	D	E		G	

ITERATOR STEPS: PRE-ORDER

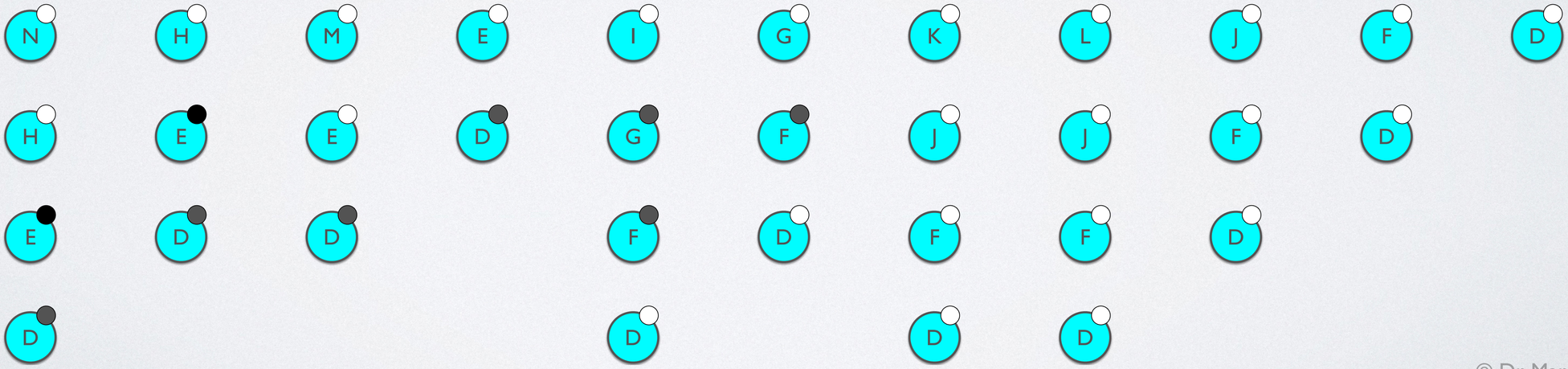
- Initialization:
  - Push the root node onto the stack.
- Element access:
  - Use the top node on the stack.
- Move forward:
  - Hold on to the top node on stack and pop.
  - If the top node has a right node, push it onto the stack.
  - If the new top node has a left node, push it onto the stack.

POST-ORDER WITH STACK (AND FRONTIER)



Post-Order:

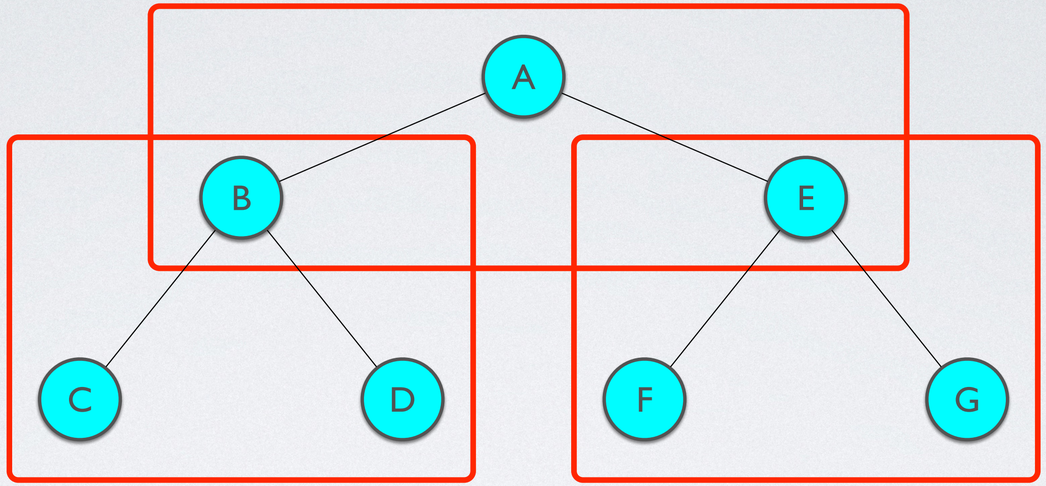
Stack:

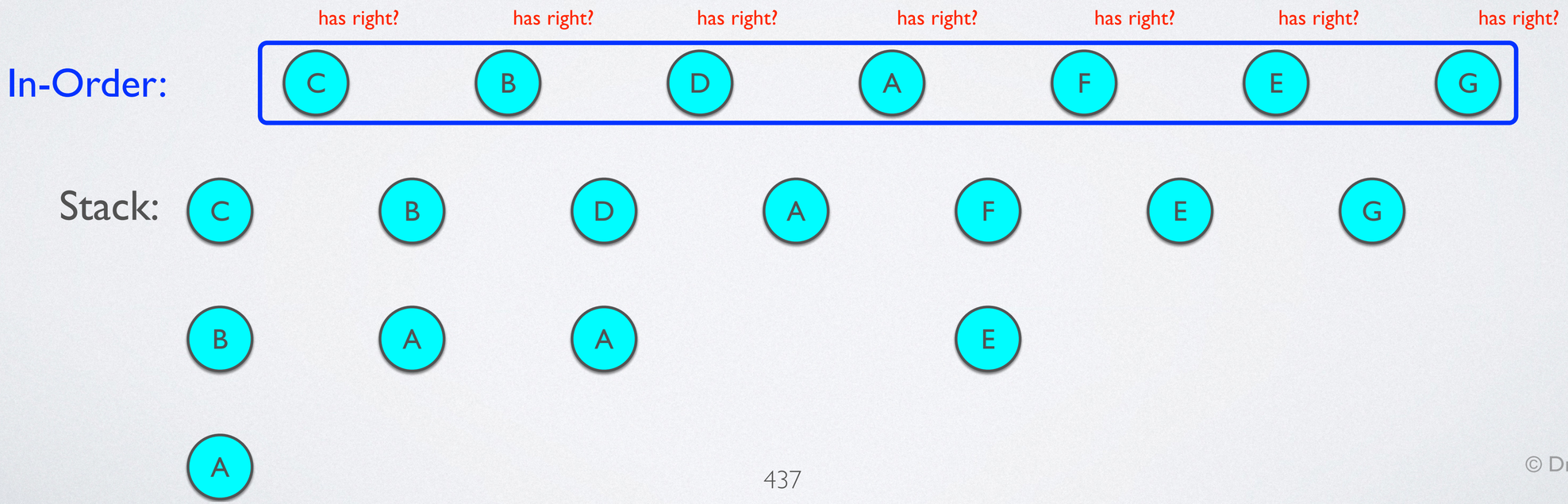


ITERATOR STEPS: POST-ORDER

- Initialization:
    - Push all left nodes, starting at the root node, onto the stack using the frontier tag "must explore right".
    - Clear the top node's frontier tag "must explore right." If the top node has a right node, push it onto the stack using the frontier tag "must explore right".
    - Push all left nodes from the top's right node onto the stack using the frontier tag "must explore right".
- Element access:
    - Use the top node on the stack.
- Move forward:
    - Pop.
    - If the stack is not empty, hold on to the top node.
    - If the top node has the frontier tag "must explore right," then clear the tag and
        - If the top node has the right node, push it onto the stack using the frontier tag "must explore right."
        - Push all left nodes, from the top's right node, onto the stack using the frontier tag "must explore right".

IN-ORDER WITH STACK





ITERATOR STEPS: IN-ORDER

- Initialization:
  - Push all left nodes onto the stack, starting at the root node.
- Element access:
  - Use the top node on the stack.
- Move forward:
  - Hold on to the top node on the stack and pop.
  - If the top node has the right node, push it onto the stack.
  - Push all left nodes from the top's right node onto the stack.

DEPTH-FIRST SEARCH ITERATOR

```
template<typename T>
class DepthFirstTraversal
{
public:
    using pointer = const BTree<T>*;
    using frontier = Frontier<pointer>;
    using iterator = DepthFirstTraversal<T>;
    using node = BTree<T>::node;
    using difference_type = std::ptrdiff_t;          // to satisfy concept weakly_incrementable
    using value_type = T;                            // to satisfy concept indirectly_readable

    enum class Mode { PRE_ORDER, POST_ORDER, IN_ORDER };

    DepthFirstTraversal( const Node& aBtree, Mode aMode = Mode::PRE_ORDER );

    const T& operator*() const noexcept;

    Iterator& operator++();
    Iterator operator++(int);

    bool operator==( const Iterator& aOther ) const noexcept;

    iterator begin() const noexcept;
    iterator end() const noexcept;

private:
    pointer fRoot;
    std::stack<frontier> fStack;
    Mode fMode;

    void pushNode( pointer aNode ) noexcept;
};
```

type aliases

iterator modes

forward iterator

iterator stack

binary tree root

```
void pushNode( pointer aNode ) noexcept
{
    while ( aNode )
    {
        fStack.push( aNode );

        if ( aNode->hasLeft() )
        {
            aNode = &aNode->left();
        }
        else
        {
            if ( fMode == Mode::POST_ORDER )
            {
                fStack.top().mustExploreRight = false;

                if ( aNode->hasRight() )
                {
                    aNode = &aNode->right();

                    continue;
                }
            }

            aNode = nullptr;
        }
    }
}
```

PUSH NODES

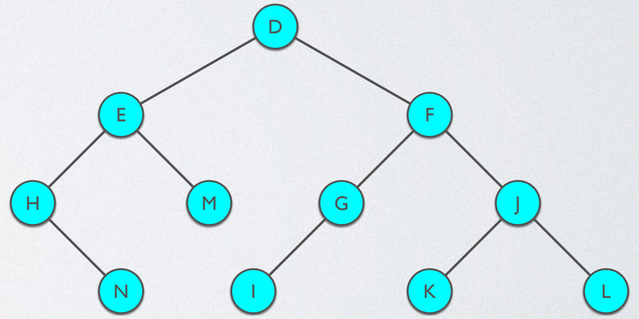
- The method pushNodes() starts by pushing all left nodes onto the stack.

- If the iterator performs post-order traversal, then, when the left nodes are exhausted, we need to inspect the top node on the stack. If it has a right node, we must push it onto the stack with all its left nodes.

BINARY TREE TRAVERSAL WITH ITERATORS

```
std::cout << "Breadth First Search:";

for ( auto& s : BreadthFirstTraversal<std::string>(aBTree) )
{
    std::cout << ' ' << s;
}

std::cout << std::endl << "DFS - PreOrder:";

for ( auto& s : DepthFirstTraversal<std::string>(aBTree) )
{
    std::cout << ' ' << s;
}

std::cout << std::endl << "DFS - PostOrder:";

for ( auto& s : DepthFirstTraversal<std::string>(aBTree, DepthFirstTraversal<std::string>::Mode::POST_ORDER ) )
{
    std::cout << ' ' << s;
}

std::cout << "DFS - InOrder:";

for ( auto& s : DepthFirstTraversal<std::string>(aBTree, DepthFirstTraversal<std::string>::Mode::IN_ORDER ) )
{
    std::cout << ' ' << s;
}

std::cout << std::endl;
```

Breadth First Search: **D E F H M G J N I K L**
DFS - PreOrder: **D E H N M F G I J K L**
DFS - PostOrder: **N H M E I G K L J F D**
DFS - InOrder: **H N E M D I G F K J L**



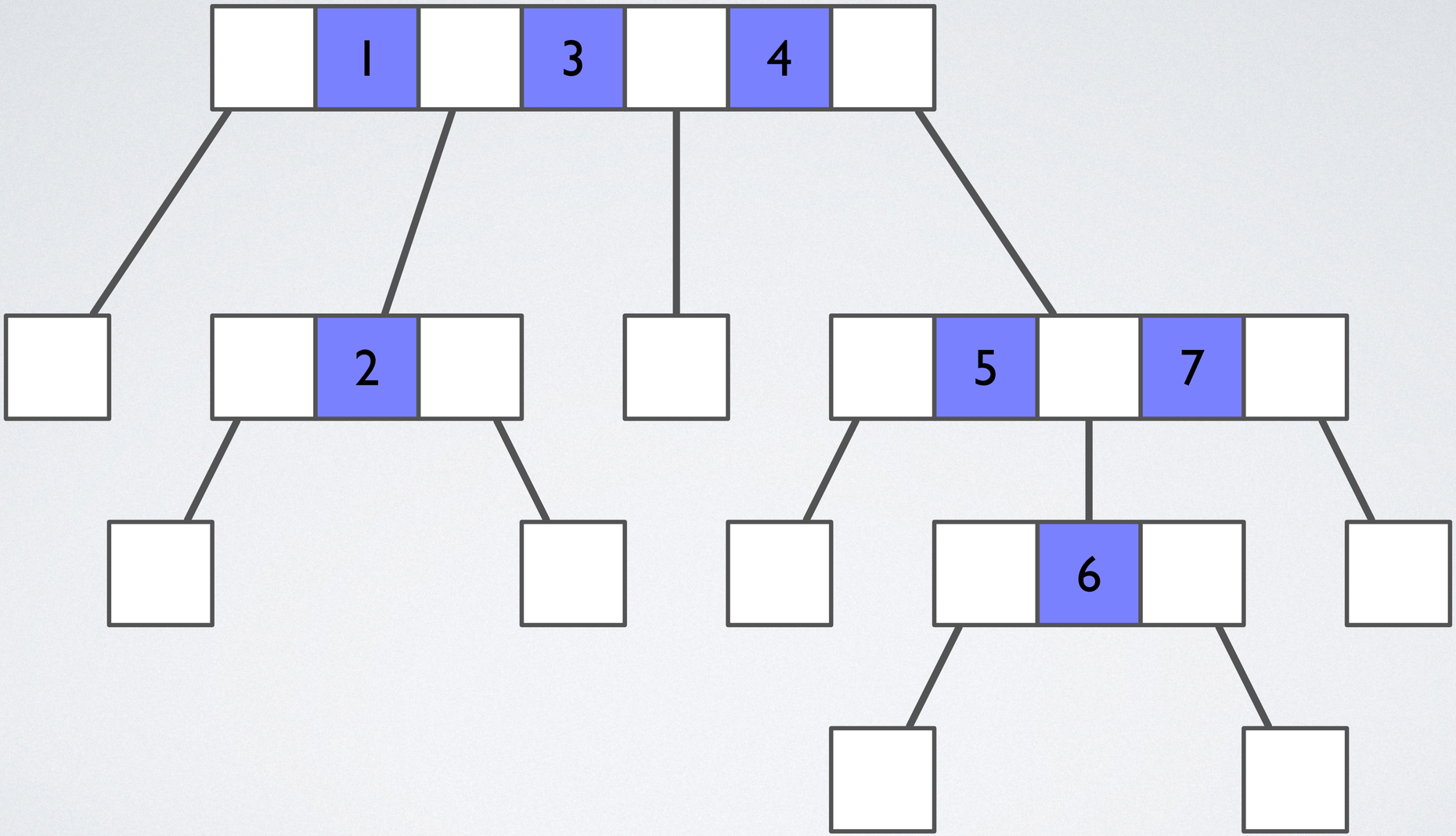
M-WAY SEARCH TREE

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

  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 in sub-tree 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 sub-tree T_{i-1} is called the left sub-tree of k_i.
  - All the keys in sub-tree 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 sub-tree T_i is called the right sub-tree of k_i.

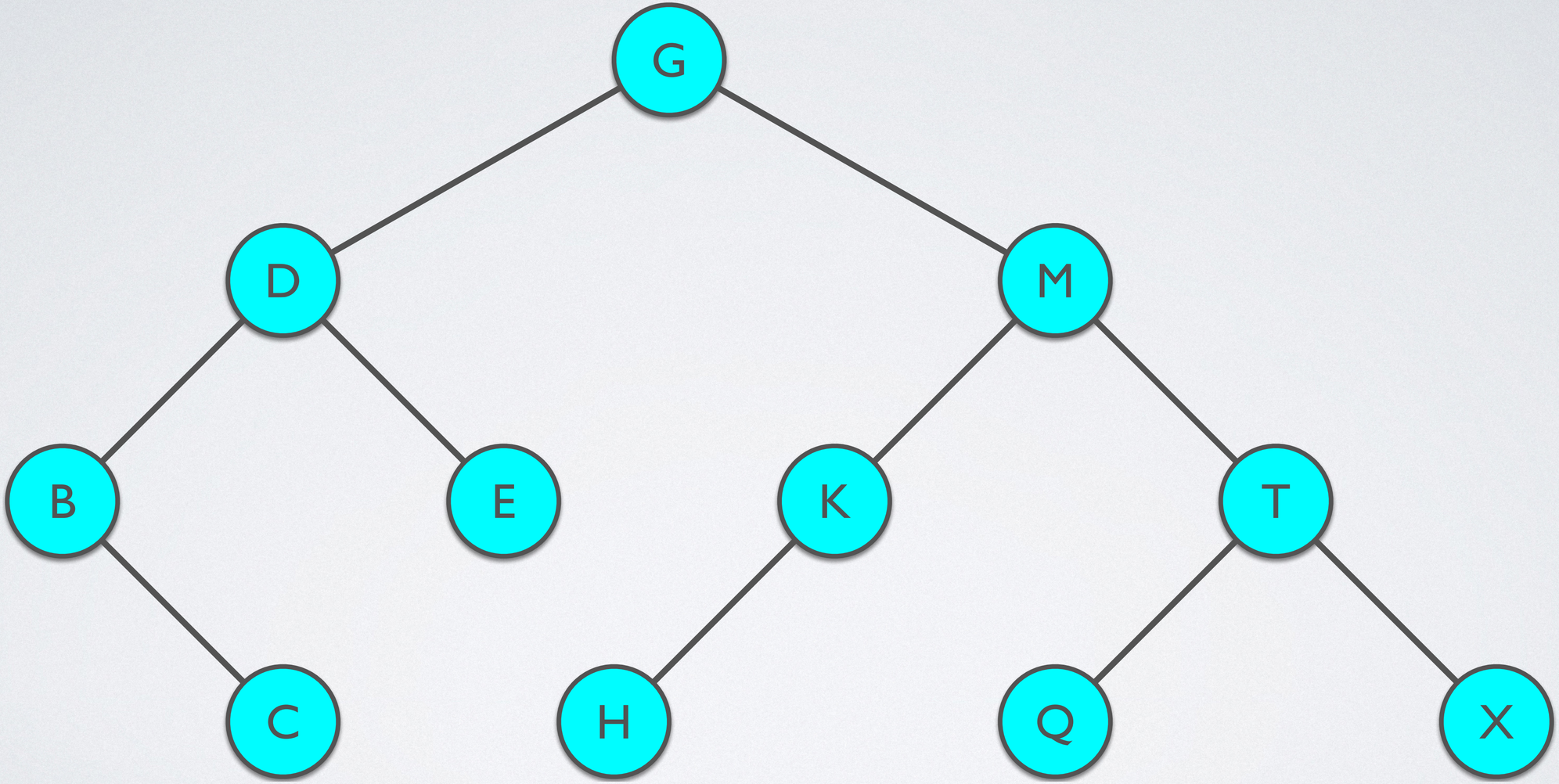
A 4-WAY SEARCH TREE



2-WAY SEARCH TREE

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

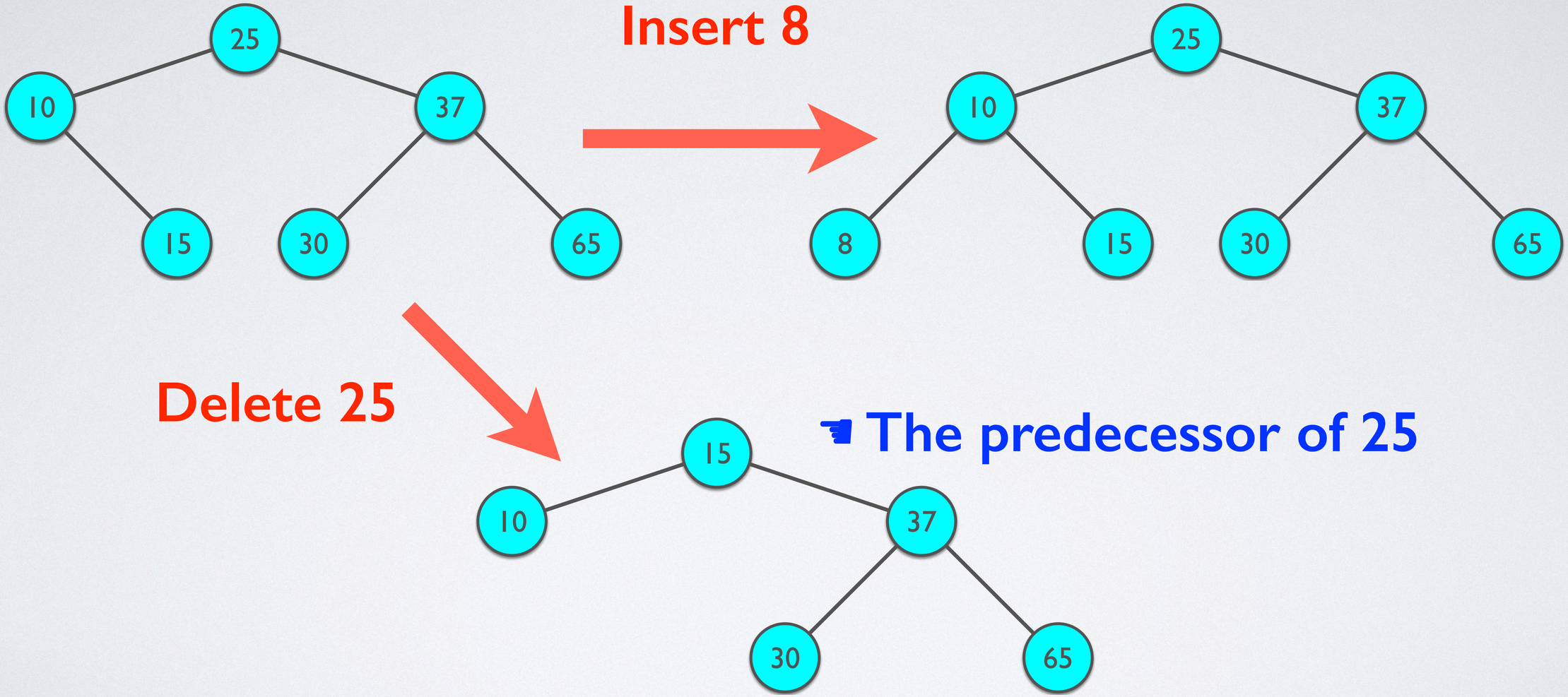
BINARY SEARCH TREE EXAMPLE



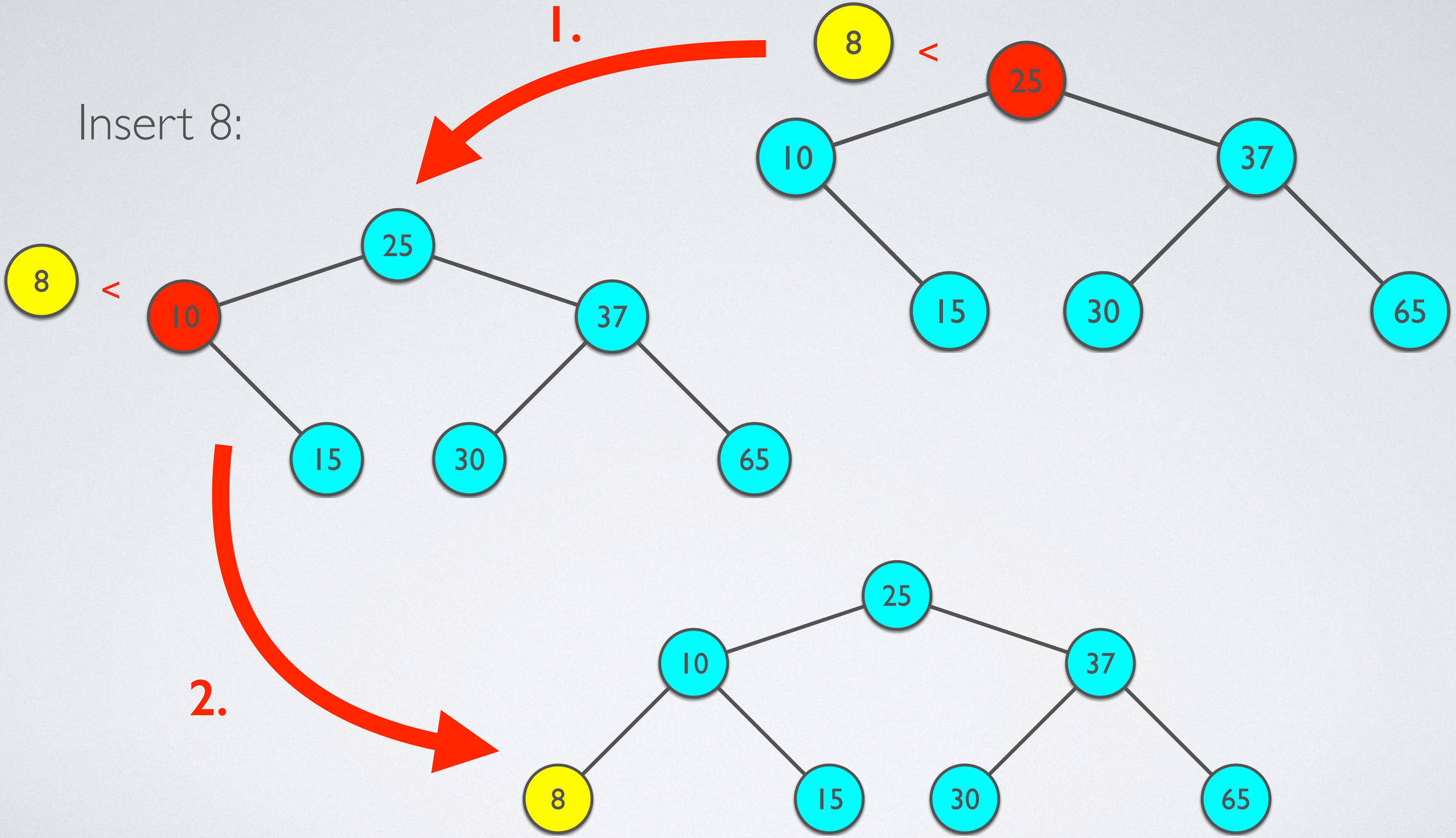
TRAVERSAL OF A BINARY SEARCH TREE

- 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.

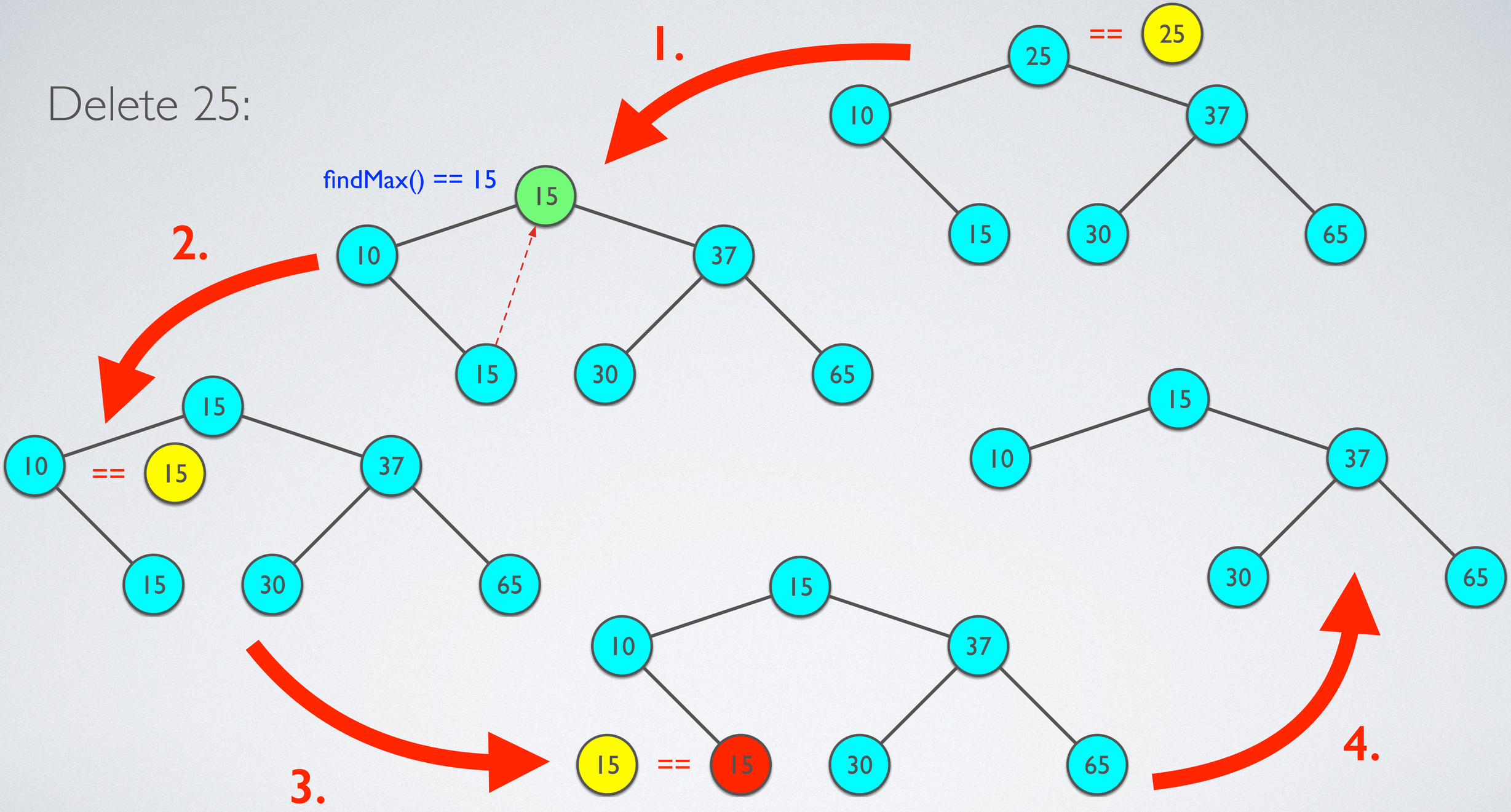
BINARY SEARCH TREE OPERATIONS



Insert 8:



Delete 25:



BINARY SEARCH TREE FEATURES

```
template<typename T>
class BTree
{
public:
    using node = std::unique_ptr<BTree>;

    ...

    const T& findMax() const noexcept;
    const T& findMin() const noexcept;

    bool insert( const T& aKey ) noexcept;
    bool remove( const T& aKey, BTree* aParent = nullptr ) noexcept;

private:
     T fKey;
     node fLeft;
     node fRight;
};
```

FINDMAX

```
const T& findMax() const noexcept
{
    if ( hasRight() )
    {
        return right().findMax();
    }

    return **this;
}
```

return payload (key)

FINDMIN

```
const T& findMin() const noexcept
{
    if ( hasLeft() )
    {
        return left().findMin();
    }

    return **this;
}
```

return payload (key)

INSERT

```
bool insert( const T& aKey ) noexcept
{
    BTree* x = this;
    BTree* y = nullptr;

    while ( x )
    {
        y = x;

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

        x = (aKey < **x ? x->fLeft.get() : x->fRight.get());
    }

    if ( !y ) { return false; }                     // insert failed (NIL)
    else
    {
        node z = BTree::makeNode( aKey );

        if ( aKey < **y ) { y->fLeft = std::move( z ); }
        else { y->fRight = std::move( z ); }
    }

    return true;                                    // insert succeeded
}
```

find insertion point

y is insertion root node

move in position

REMOVE

```
bool remove( const T& aKey, BTree* aParent = nullptr ) noexcept
{
   BTree* x = this;
   BTree* y = aParent;

   while ( x )
   {
      if ( aKey == **x ) { break; }

      y = x;                                    // new parent

      x = (aKey < **x ? x->fLeft.get() : x->fRight.get());
   }

   if ( !x ) { return false; }                  // delete failed

   if ( x->hasLeft() )
   {
      x->fKey = x->left().findMax();            // find max to left
      x->left().remove( **x, x );
   }
   else
   {
      if ( x->hasRight() )
      {
         x->fKey = x->right().findMin();        // find min to right
         x->right().remove( **x, x );
      }
      else
      {
         if ( y->fLeft.get() == x ) { y->fLeft = nullptr; }
         else { y->fRight = nullptr; }
      }
   }

   return true;
}
```

remove is a recursive procedure

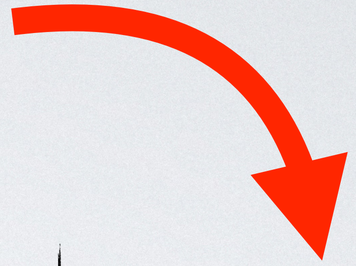
find deletion root node

node to be removed is an interior node

delete leaf node

BINARY SEARCH TREE TEST

```
std::cout << "The tree (BFS):";
for ( auto& s : BreadthFirstTraversal<size_t>(aBTree) ) { std::cout << ' ' << s; }
std::cout << std::endl;

if ( n25->insert( 8 ) ) { std::cout << "Insert 8 successful." << std::endl; }
else { std::cout << "Insert 8 failed." << std::endl; }

std::cout << "The tree (BFS):";
for ( auto& s : BreadthFirstTraversal<size_t>(aBTree) ) { std::cout << ' ' << s; }
std::cout << std::endl;

if ( n25->remove( 8 ) ) { std::cout << "Remove 8 successful." << std::endl; }
else { std::cout << "Remove 8 failed." << std::endl; }

if ( n25->remove( 25 ) ) { std::cout << "Remove 25 successful." << std::endl; }
else { std::cout << "Remove 25 failed." << std::endl; }

std::cout << "The tree (BFS):";
for ( auto& s : BreadthFirstTraversal<size_t>(aBTree) ) { std::cout << ' ' << s; }
std::cout << std::endl;
```



```
The tree (BFS): 25 10 37 15 30 65
Insert 8 successful.
The tree (BFS): 25 10 37 8 15 30 65
Delete 8
Remove 8 successful.
Delete 15
Remove 25 successful.
The tree (BFS): 15 10 37 30 65
Delete 15
Delete 37
Delete 65
Delete 30
Delete 10
```

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 binary trees)

AVL VS RED-BLACK TREES

- Both AVL trees and Red-Black trees are self-balancing binary search trees. However, the balance operations 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, resulting in slower insertion and removal but faster retrieval.