

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).

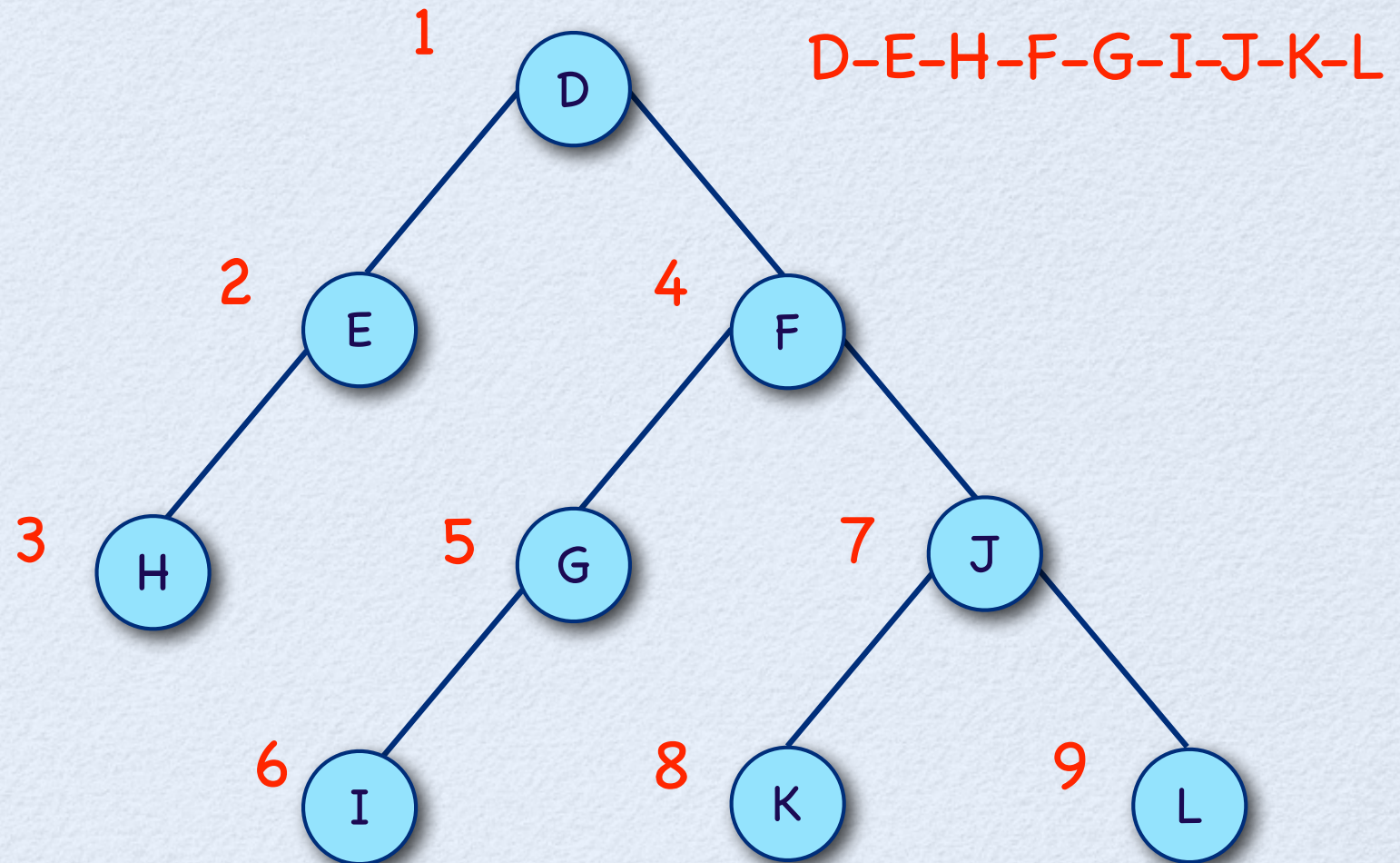
Pre-order Traversal Example

Depth = 0:

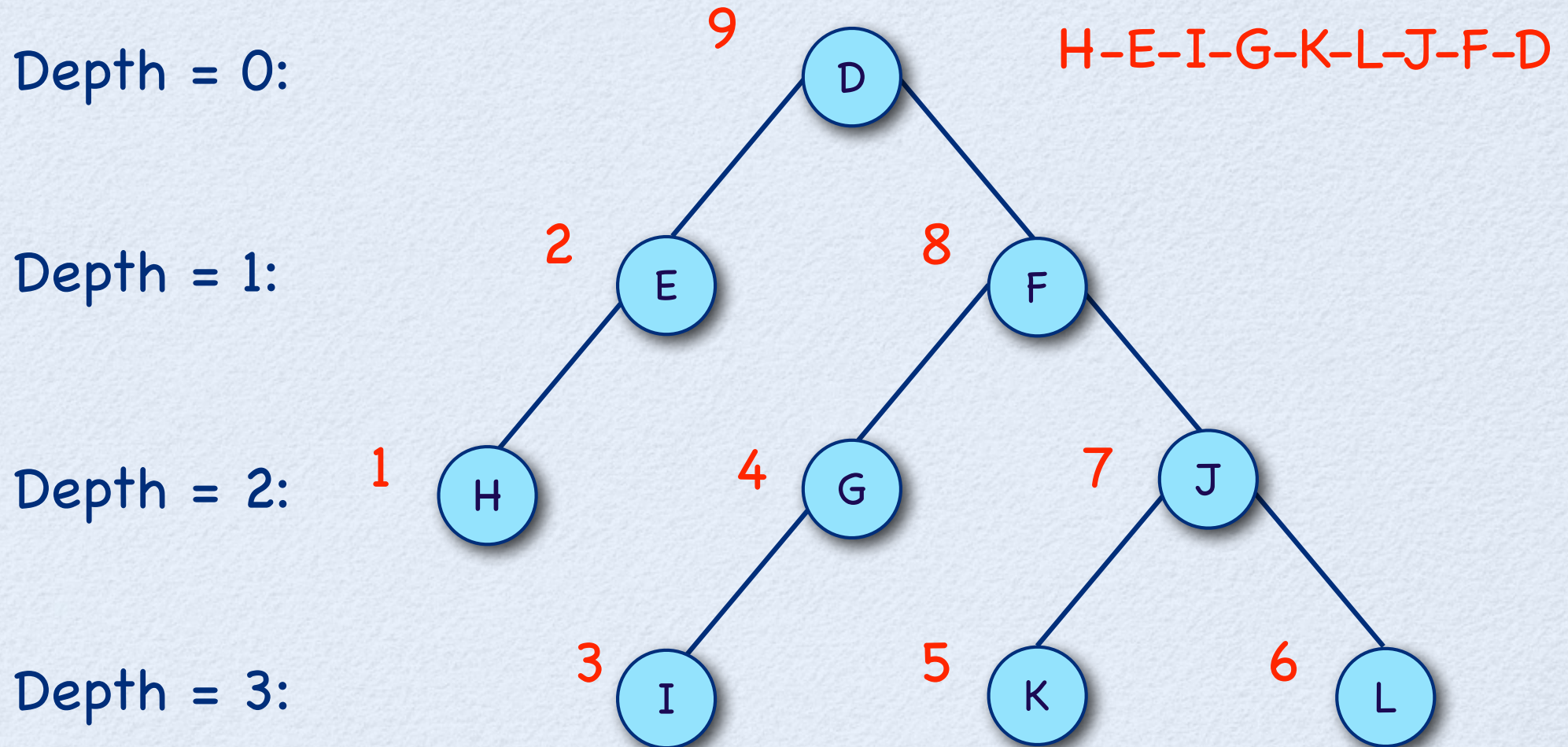
Depth = 1:

Depth = 2:

Depth = 3:



Post-order Traversal Example



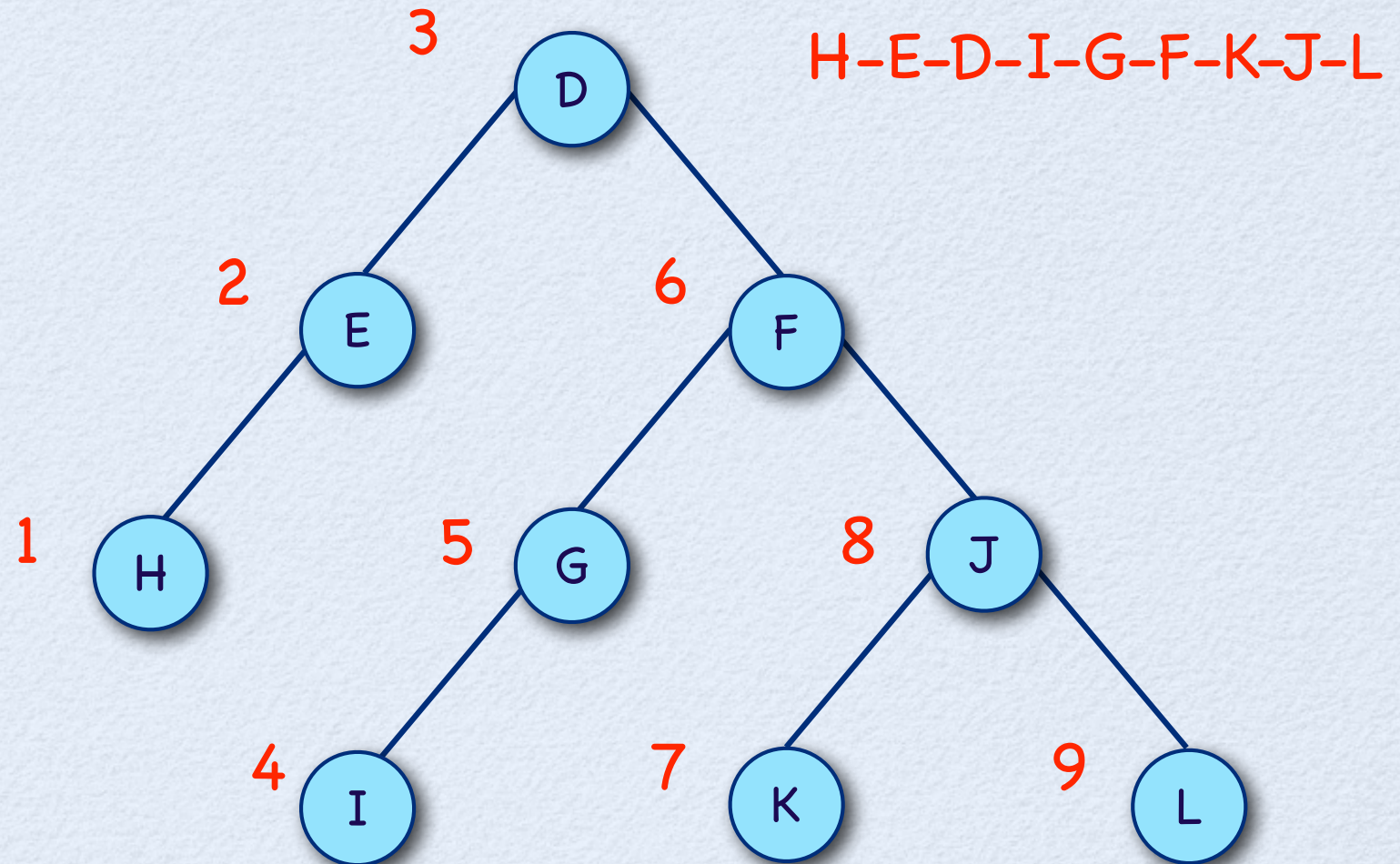
In-order Traversal Example

Depth = 0:

Depth = 1:

Depth = 2:

Depth = 3:



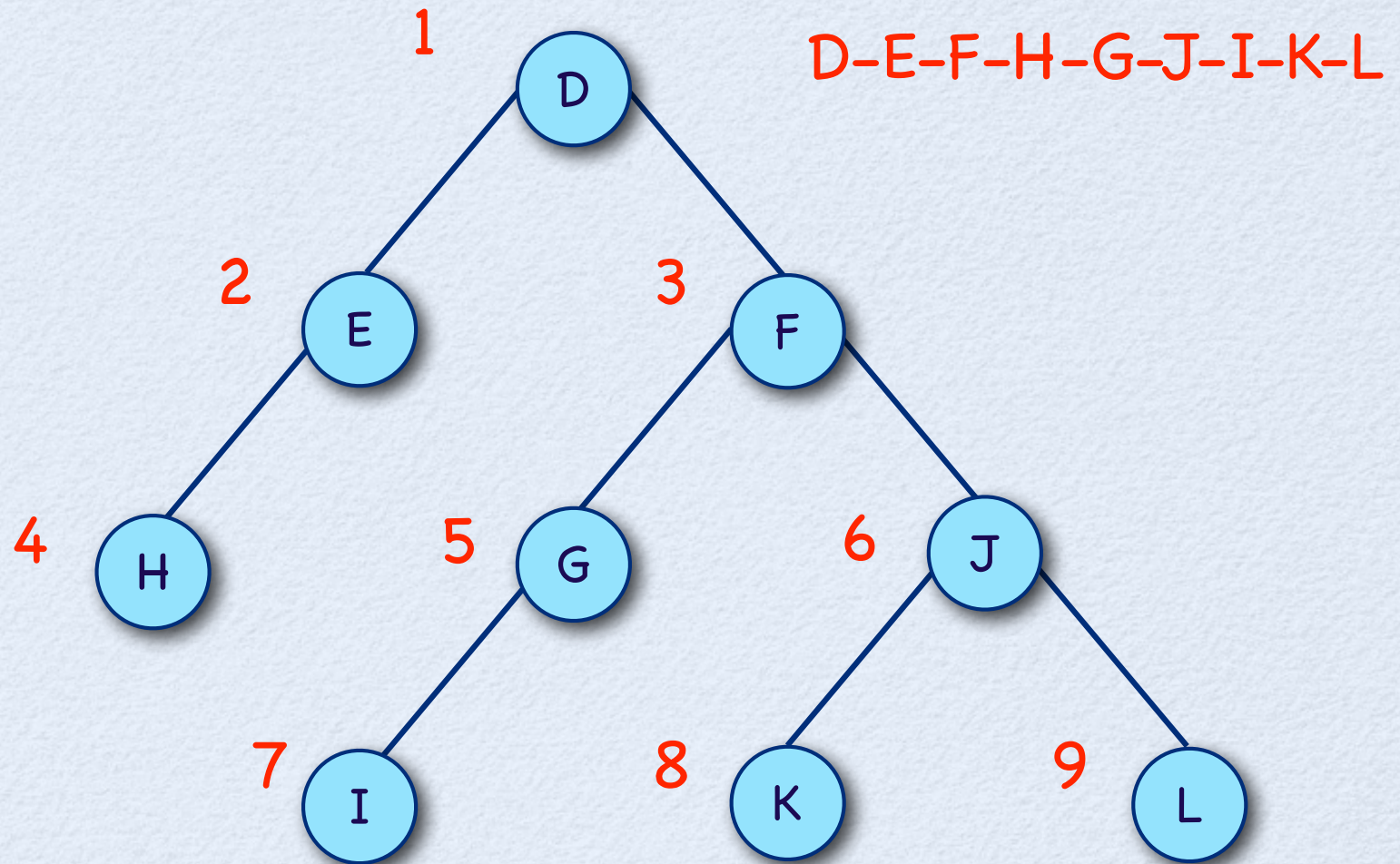
Breadth-first Traversal Example

Depth = 0:

Depth = 1:

Depth = 2:

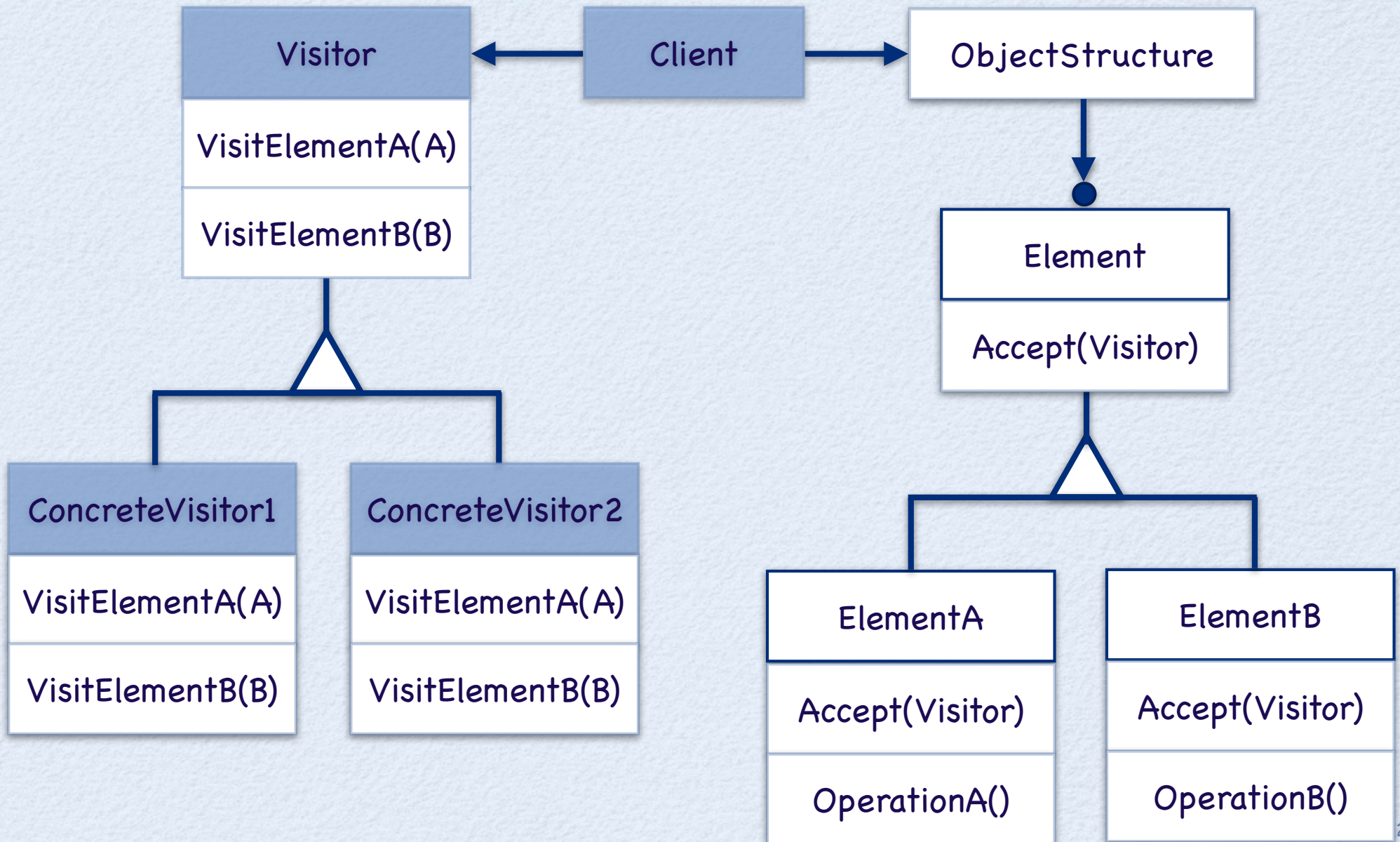
Depth = 3:



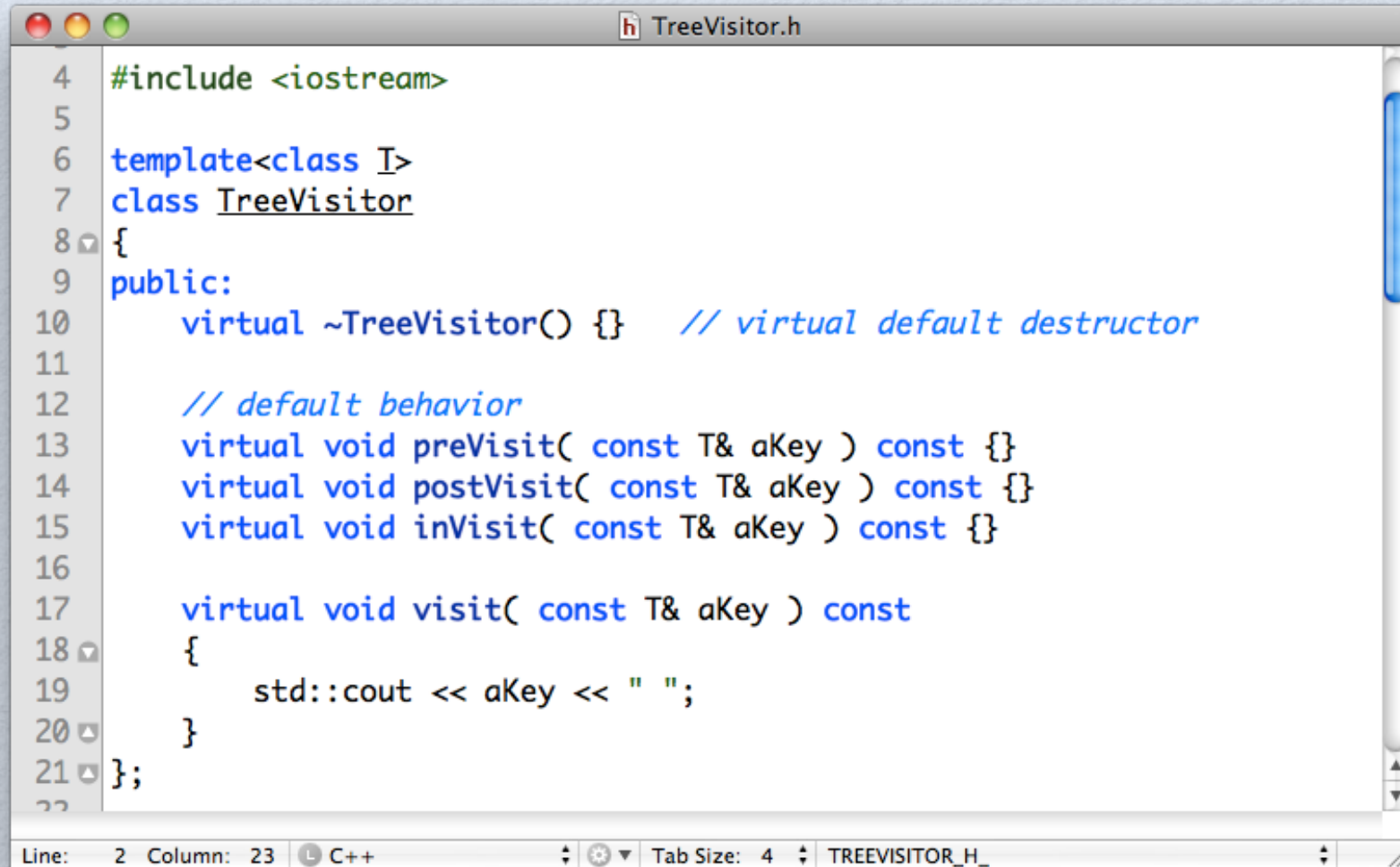
The Visitor Pattern

- 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



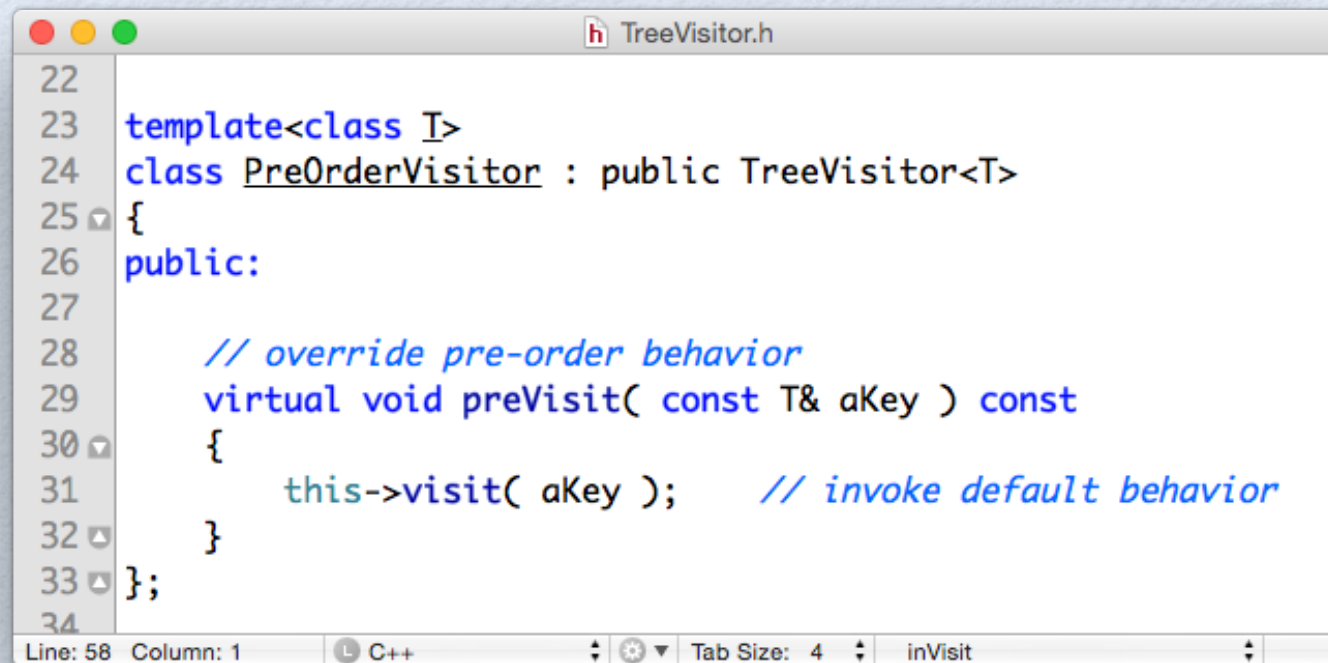
A Tree Visitor



```
4  #include <iostream>
5
6  template<class T>
7  class TreeVisitor
8  {
9  public:
10     virtual ~TreeVisitor() {}    // virtual default destructor
11
12     // default behavior
13     virtual void preVisit( const T& aKey ) const {}
14     virtual void postVisit( const T& aKey ) const {}
15     virtual void inVisit( const T& aKey ) const {}
16
17     virtual void visit( const T& aKey ) const
18     {
19         std::cout << aKey << " ";
20     }
21 };
22
```

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

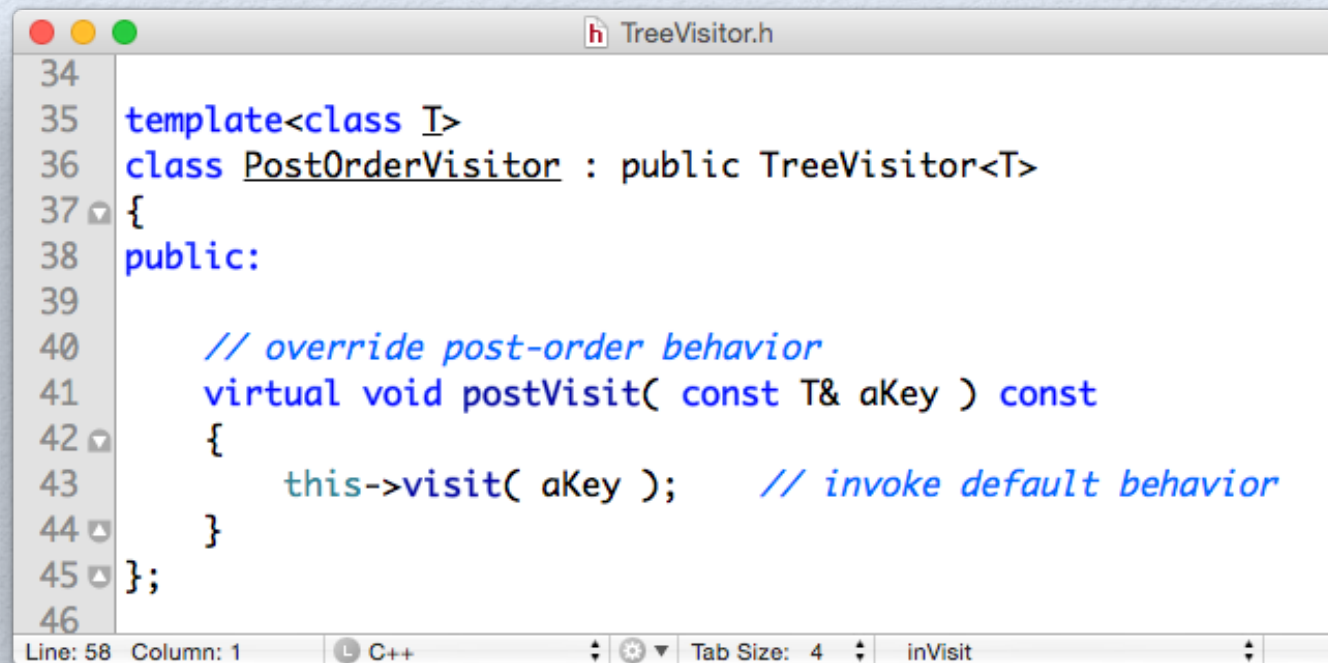
PreOrderVisitor



```
22
23 template<class T>
24 class PreOrderVisitor : public TreeVisitor<T>
25 {
26 public:
27
28     // override pre-order behavior
29     virtual void preVisit( const T& aKey ) const
30     {
31         this->visit( aKey );    // invoke default behavior
32     }
33 };
34
```

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

PostOrderVisitor

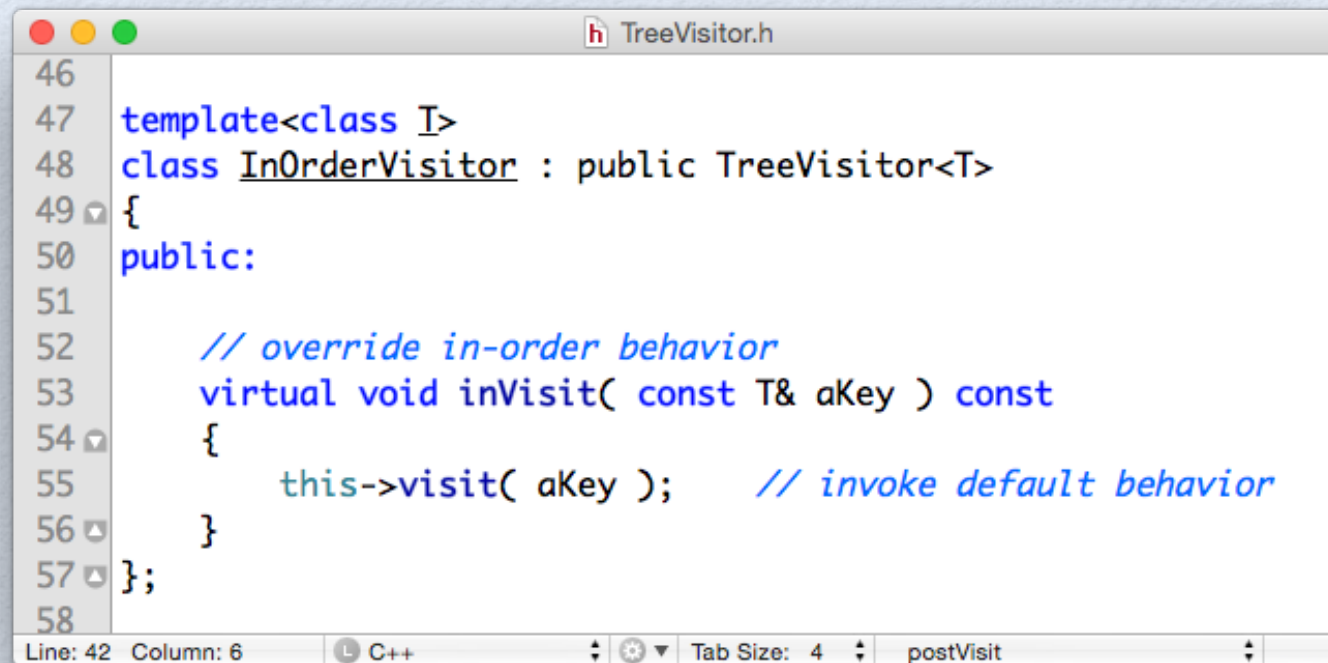


The screenshot shows a code editor window titled "TreeVisitor.h". The code defines a template class `PostOrderVisitor` that inherits from `TreeVisitor<T>`. It implements the `postVisit` method, which calls `visit` on the current node. The code is as follows:

```
34
35 template<class T>
36 class PostOrderVisitor : public TreeVisitor<T>
37 {
38 public:
39
40     // override post-order behavior
41     virtual void postVisit( const T& aKey ) const
42     {
43         this->visit( aKey );    // invoke default behavior
44     }
45 };
46
```

The editor's status bar at the bottom indicates "Line: 58 Column: 1", "C++", "Tab Size: 4", and "inVisit".

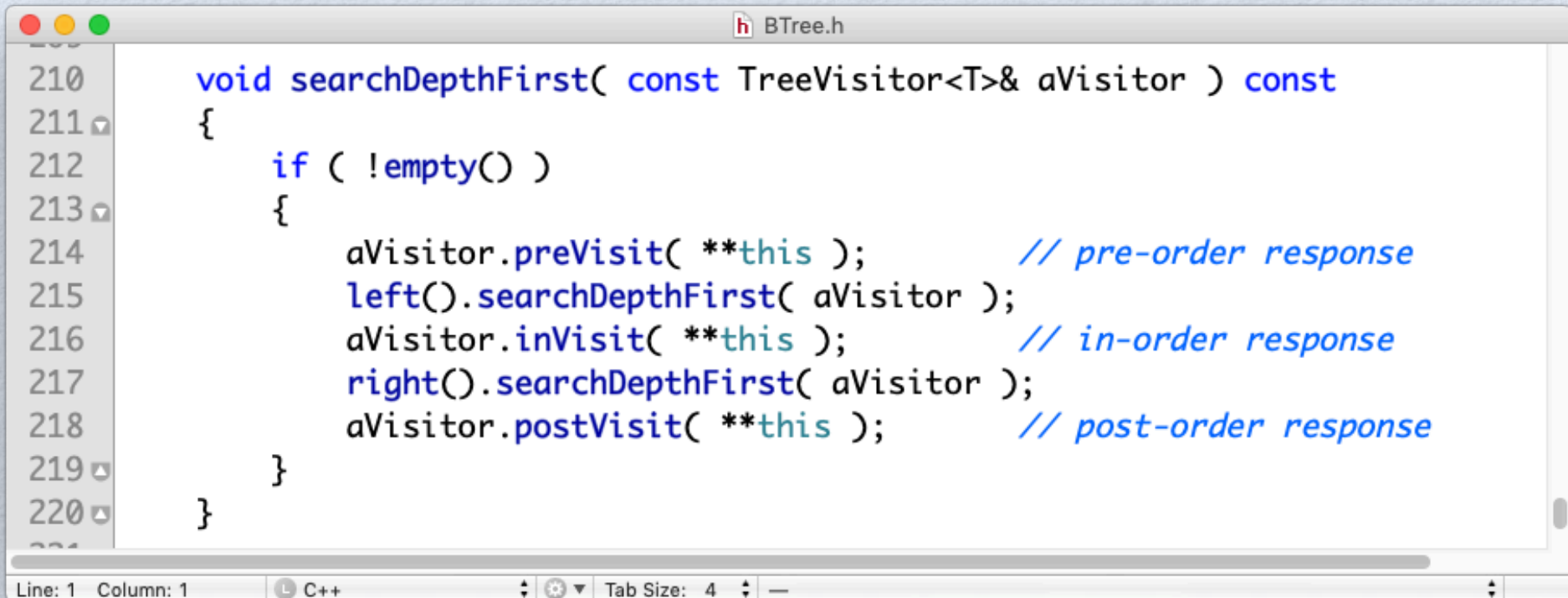
InOrderVisitor



```
46
47 template<class T>
48 class InOrderVisitor : public TreeVisitor<T>
49 {
50 public:
51
52     // override in-order behavior
53     virtual void inVisit( const T& aKey ) const
54     {
55         this->visit( aKey );    // invoke default behavior
56     }
57 };
58
```

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

Depth-first Traversal for BTree



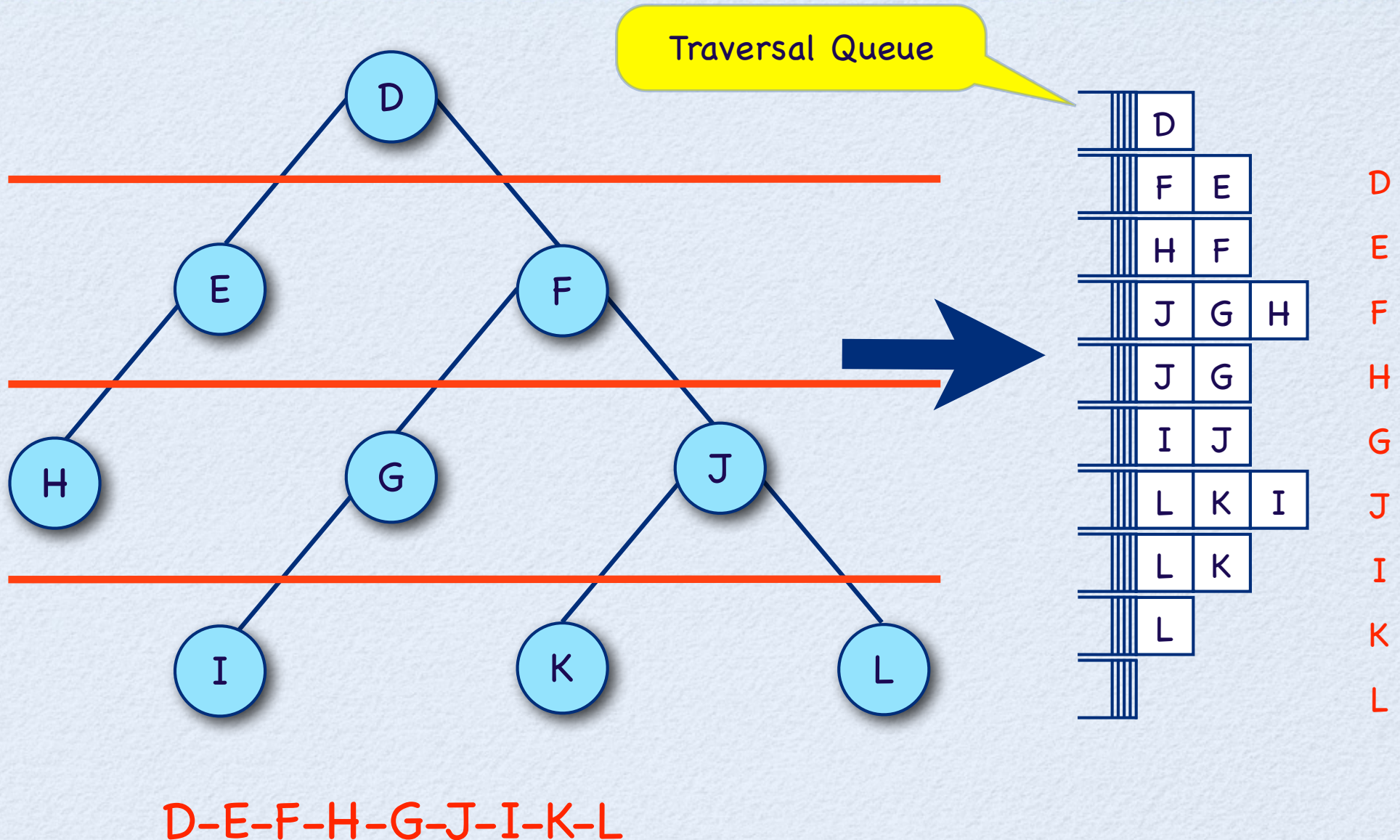
```
210 void searchDepthFirst( const TreeVisitor<T>& aVisitor ) const
211 {
212     if ( !empty() )
213     {
214         aVisitor.preVisit( **this );           // pre-order response
215         left().searchDepthFirst( aVisitor );
216         aVisitor.inVisit( **this );           // in-order response
217         right().searchDepthFirst( aVisitor );
218         aVisitor.postVisit( **this );         // post-order response
219     }
220 }
```

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

```
219 #include "BTree.h"
220
221 void testDFS()
222 {
223     cout << "Test DFS." << endl;
224
225     using StringBTree = BTree<string>;
226
227     string s1( "A" );
228     string s2( "B" );
229     string s3( "C" );
230
231     StringBTree root( "Hello World!" );
232     StringBTree nodeA( s1 );
233     StringBTree nodeB( s2 );
234     StringBTree nodeAA( s3 );
235     StringBTree nodeBB( "D" );
236
237     root.attachLeft( nodeA );
238     root.attachRight( nodeB );
239     const_cast<StringBTree&>(root.left()).attachLeft( nodeAA );
240     const_cast<StringBTree&>(root.right()).attachRight( nodeBB );
241
242     cout << "root:      " << *root << endl;
243     cout << "root->L:    " << *root.left() << endl;
244     cout << "root->R:    " << *root.right() << endl;
245     cout << "root->L->L: " << *root.left().left() << endl;
246     cout << "root->R->R: " << *root.right().right() << endl;
247
248     root.searchDepthFirst( PreOrderVisitor<string>() );
249     cout << endl;
250
251     const_cast<StringBTree&>(root.right()).detachRight();
252     const_cast<StringBTree&>(root.left()).detachLeft();
253     root.detachRight();
254     root.detachLeft();
255
256     cout << "All trees are going to be deleted now!" << endl;
257 }
```

```
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



Breadth-first Traversal for BTree

```
222
223 void searchBreadthFirst( const TreeVisitor<T>& aVisitor ) const
224 {
225     std::queue<const BTree<T>*> lQueue;    // traversal queue (pointer required)
226
227     lQueue.push( this );                  // start with root node
228
229     while ( !lQueue.empty() )
230     {
231         const BTree<T>* lFront = lQueue.front();
232
233         if ( !lFront->empty() )
234         {
235             aVisitor.visit( **lFront );
236             lQueue.push( &lFront->left() );
237             lQueue.push( &lFront->right() );
238         }
239
240         lQueue.pop();
241     }
242 }
```

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


```
265 #include "BTree.h"
266
267 void testBFS()
268 {
269     cout << "Test BFS." << endl;
270
271     using StringBTree = BTree<string>;
272
273     string s1( "A" );
274     string s2( "B" );
275     string s3( "C" );
276
277     StringBTree root( "Hello World!" );
278     StringBTree nodeA( s1 );
279     StringBTree nodeB( s2 );
280     StringBTree nodeAA( s3 );
281     StringBTree nodeBB( "D" );
282
283     root.attachLeft( nodeA );
284     root.attachRight( nodeB );
285     const_cast<StringBTree&>(root.left()).attachLeft( nodeAA );
286     const_cast<StringBTree&>(root.right()).attachRight( nodeBB );
287
288     cout << "root:      " << *root << endl;
289     cout << "root->L:    " << *root.left() << endl;
290     cout << "root->R:    " << *root.right() << endl;
291     cout << "root->L->L: " << *root.left().left() << endl;
292     cout << "root->R->R: " << *root.right().right() << endl;
293
294     root.searchBreadthFirst( TreeVisitor<string>() );
295     cout << endl;
296
297     const_cast<StringBTree&>(root.right()).detachRight();
298     const_cast<StringBTree&>(root.left()).detachLeft();
299     root.detachRight();
300     root.detachLeft();
301
302     cout << "All trees are going to be deleted now!" << endl;
303 }
```

```
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!]
```

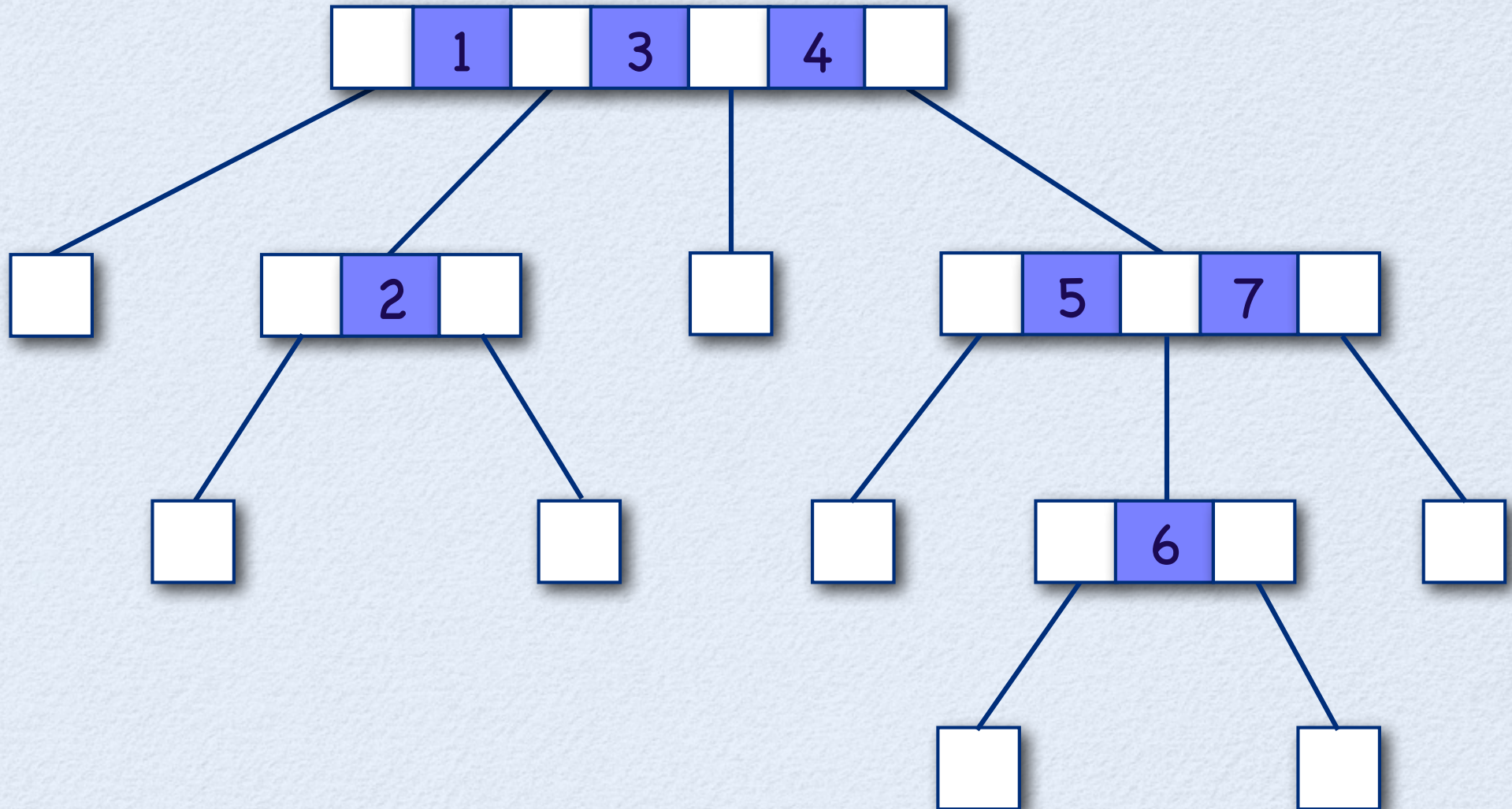
M-way Search Tree

- 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 $T_1, T_2, \dots, T_{n-1}, T_n$ and $n-1$ keys k_1, k_2, \dots, 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 .

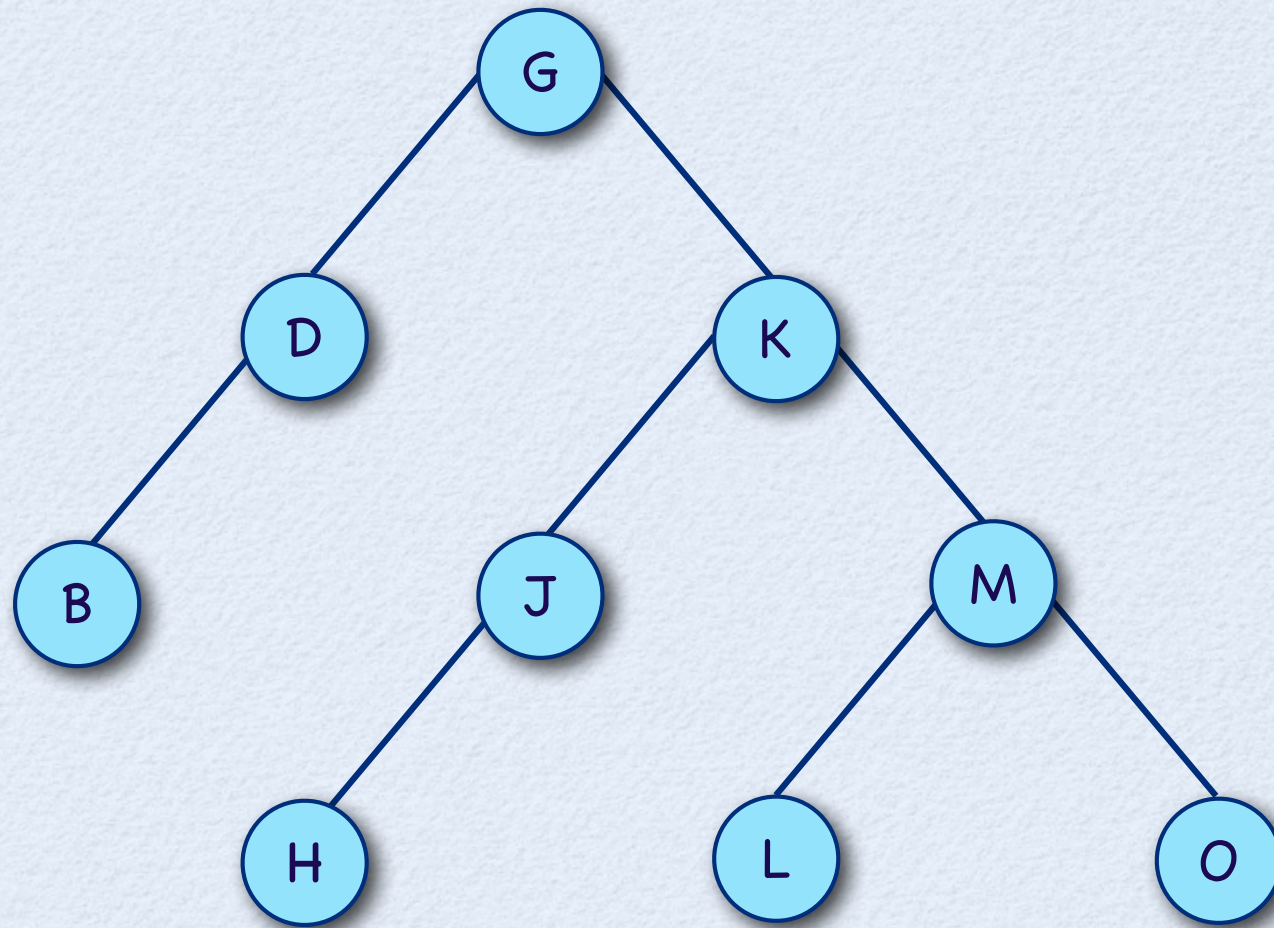
A 4-way Search Tree



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., $\forall k \in T_R: k > r$.

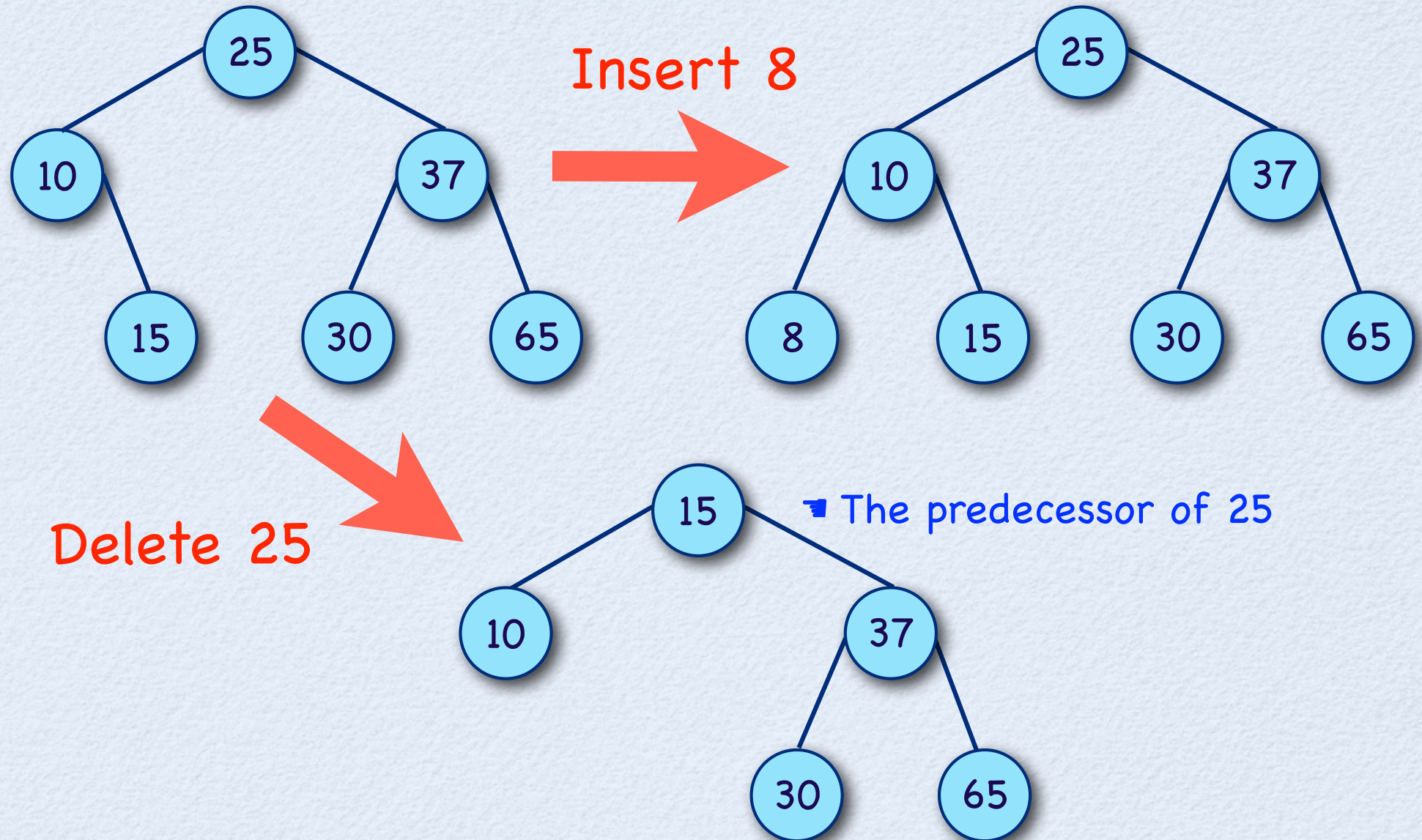
A Binary Search Tree Example



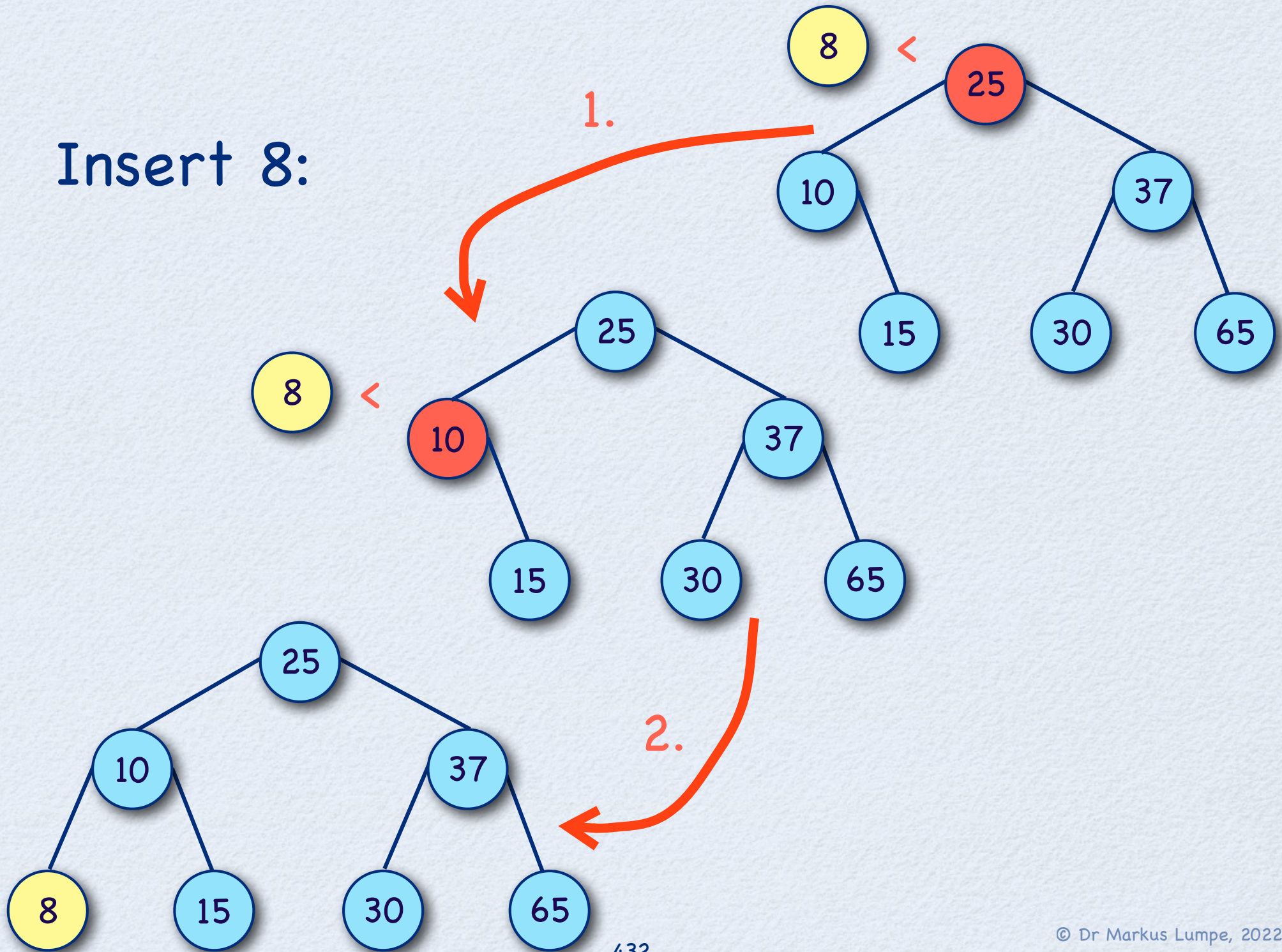
Traversing a Binary Search Tree

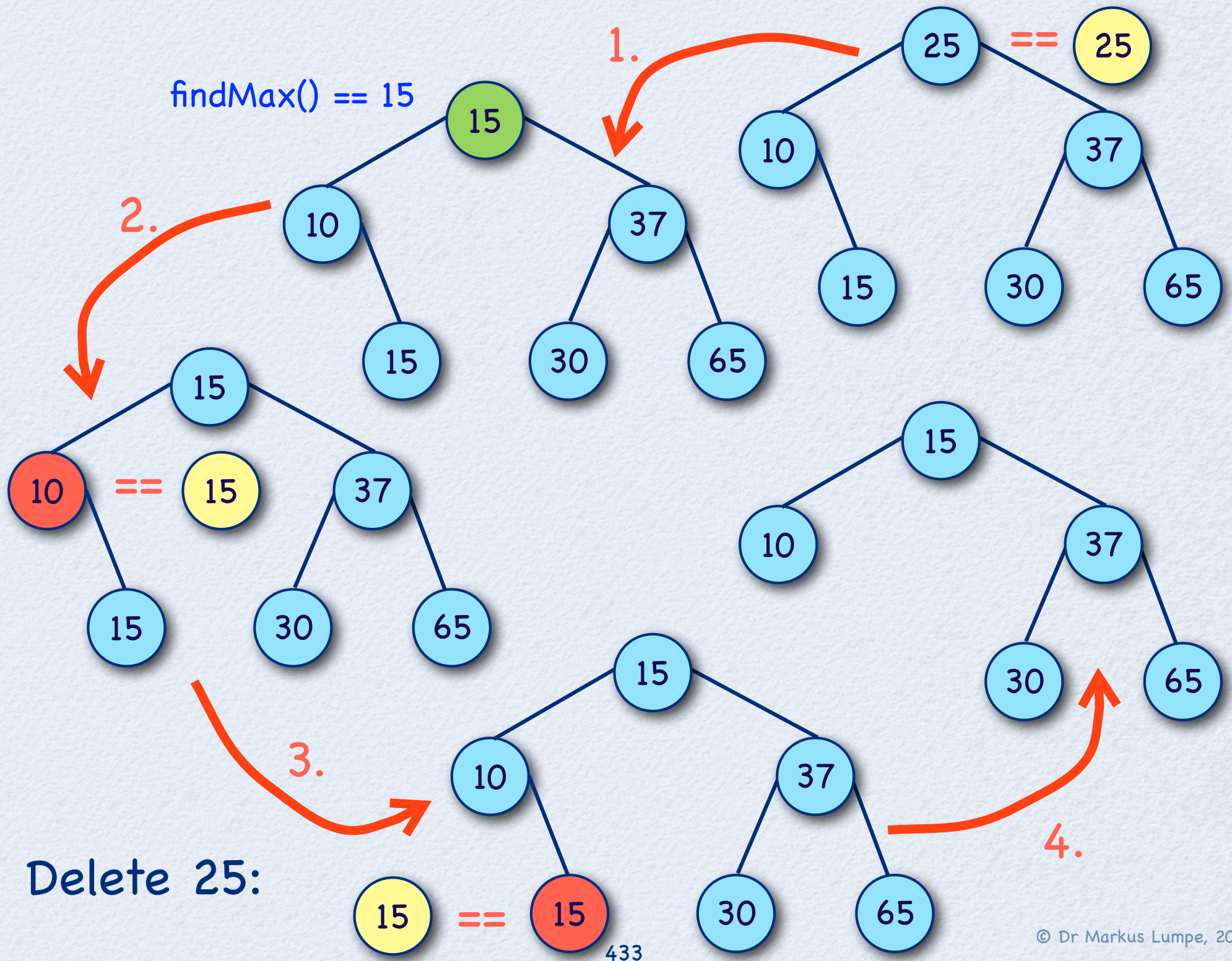
- Binary Tree Search:
 - Traverse the left subtree, and then
 - Visit the root, and then
 - Traverse the right subtree.
- We use in-order traversal to search for a given key in an M-ary search tree.

Binary Search Tree Operations



Insert 8:





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

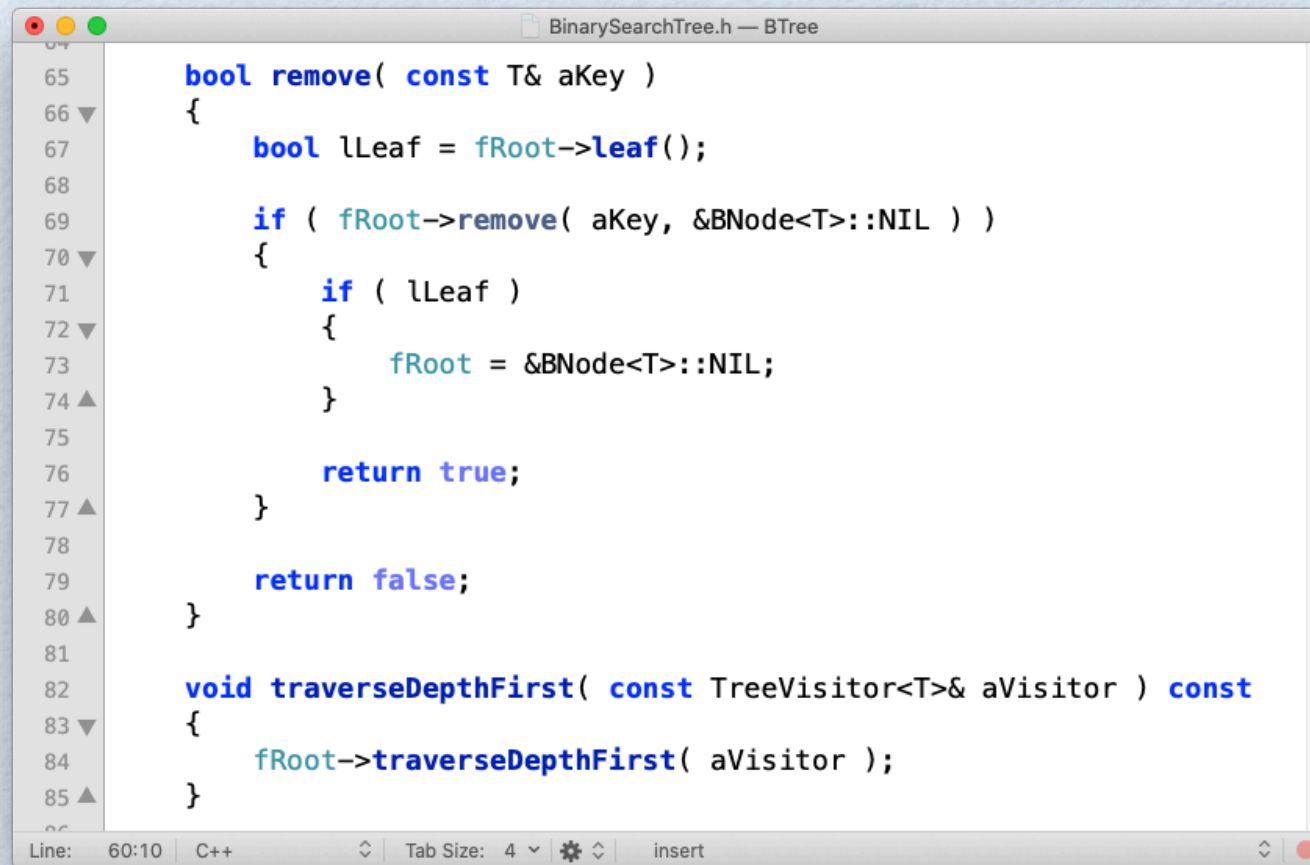
```
BinarySearchTree.h
9  #include "BNode.h"
10 #include "TreeVisitor.h"
11
12 template<typename T>
13 class BinarySearchTree
14 {
15 private:
16     BNode<T>* fRoot;
17
18 public:
19
20     BinarySearchTree();
21     ~BinarySearchTree();
22
23     bool empty() const;
24     size_t height() const;
25
26     bool insert( const T& aKey );
27     bool remove( const T& aKey );
28
29     void traverseDepthFirst( const TreeVisitor<T>& aVisitor ) const;
30 };
31
```

```
BNode.h
8  #include <stdexcept>
9
10 template<typename S>
11 struct BNode
12 {
13     S key;
14     BNode<S>* left;
15     BNode<S>* right;
16
17     static BNode<S> NIL;
18
19     ...
20 };
21
22 template<typename S>
23 BNode<S> BNode<S>::NIL;
24
```



```
63 bool insert( const S& aKey )
64 {
65     BNode<S>* x = this;
66     BNode<S>* y = &BNode<S>::NIL;
67
68     while ( !x->empty() )
69     {
70         y = x;
71
72         if ( aKey == x->key )
73         {
74             return false;           // duplicate key - error
75         }
76
77         x = aKey < x->key ? x->left : x->right;
78     }
79
80     BNode<S>* z = new BNode<S>( aKey );
81
82     if ( y->empty() )
83     {
84         return false;           // insert failed (NIL)
85     }
86     else
87     {
88         if ( aKey < y->key )
89         {
90             y->left = z;
91         }
92         else
93         {
94             y->right = z;
95         }
96     }
97
98     return true;               // insert succeeded
99 }
100
```

Remove and Depth-first Traversal



```
BinarySearchTree.h — BTree
65  bool remove( const T& aKey )
66  {
67      bool lLeaf = fRoot->leaf();
68
69      if ( fRoot->remove( aKey, &BNode<T>::NIL ) )
70      {
71          if ( lLeaf )
72          {
73              fRoot = &BNode<T>::NIL;
74          }
75
76          return true;
77      }
78
79      return false;
80  }
81
82  void traverseDepthFirst( const TreeVisitor<T>& aVisitor ) const
83  {
84      fRoot->traverseDepthFirst( aVisitor );
85  }
```

- 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.


```
121 bool remove( const S& aKey, BNode<S>* aParent )
122 {
123     BNode<S>* x = this;
124     BNode<S>* y = aParent;
125
126     while ( !x->empty() )
127     {
128         if ( aKey == x->key )
129             break;
130
131         y = x;
132         x = aKey < x->key ? x->left : x->right; // new parent
133     }
134
135     if ( x->empty() )
136         return false; // delete failed
137
138     if ( !x->left->empty() )
139     {
140         const S& lKey = x->left->findMax(); // find max to left
141         x->key = lKey;
142         x->left->remove( lKey, x );
143     }
144     else
145     {
146         if ( !x->right->empty() )
147         {
148             const S& lKey = x->right->findMin(); // find min to right
149             x->key = lKey;
150             x->right->remove( lKey, x );
151         }
152         else
153         {
154             if ( y->left == x )
155                 y->left = &NIL;
156             else
157                 y->right = &NIL;
158
159             delete x; // free deleted node
160         }
161     }
162
163     return true;
164 }
```

Line: 129 Column: 1

C++

Tab Size: 4

remove

Binary Search Tree Test

```
138 #include "BinarySearchTree.h"
139
140 void testVisitor()
141 {
142     BinarySearchTree<int> lTree;
143
144     lTree.insert( 25 );
145     lTree.insert( 10 );
146     lTree.insert( 15 );
147     lTree.insert( 37 );
148     lTree.insert( 30 );
149     lTree.insert( 65 );
150
151     lTree.traverseDepthFirst( PreOrderVisitor<int>() );
152     cout << endl;
153
154     lTree.insert( 8 );
155
156     lTree.traverseDepthFirst( PreOrderVisitor<int>() );
157     cout << endl;
158
159     lTree.remove( 25 );
160
161     lTree.traverseDepthFirst( PreOrderVisitor<int>() );
162     cout << endl;
163 }
```

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

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


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.