TREES

- **Overview**
  - Trees
  - Search Trees
- **References**
  - Richard F. Gilberg and Behrouz A. Forouzan: Data Structures - A Pseudocode Approach with C. 2nd Edition. Thomson (2005)
  - Russ Miller and Laurence Boxer: Algorithms Sequential & Parallel. 2nd Edition. Charles River Media Inc. (2005)
  - Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms", 4th Edition, The MIT Press (2022)
  - Scott Meyers: Effective Modern C++. O'Reilly (2014)
  - Anthony Williams: C++ Concurrency in Action - Practical Multithreading. Manning Publications Co. (2012)

TREE BASICS

- A tree T is a finite, non-empty set of nodes,

$$T = \{r\} \cup T_1 \cup T_2 \cup \ldots \cup T_n,$$

with the following properties:

  - A designated node of the set, r, is called the root of the tree.
  - The remaining nodes are partitioned into $n \geq 0$ subsets $T_1, T_2, \ldots, T_n$, each of which is a sub-tree.

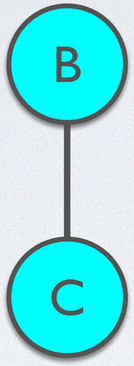
PARENT, CHILD, AND LEAF

- The root node r of tree T is the parent of all the roots r_i of the subtrees T_i, $1 < i \leq n$.
- Each root r_i of subtree T_i of tree T is called a child of r.
- A leaf node is a tree with no subtrees.

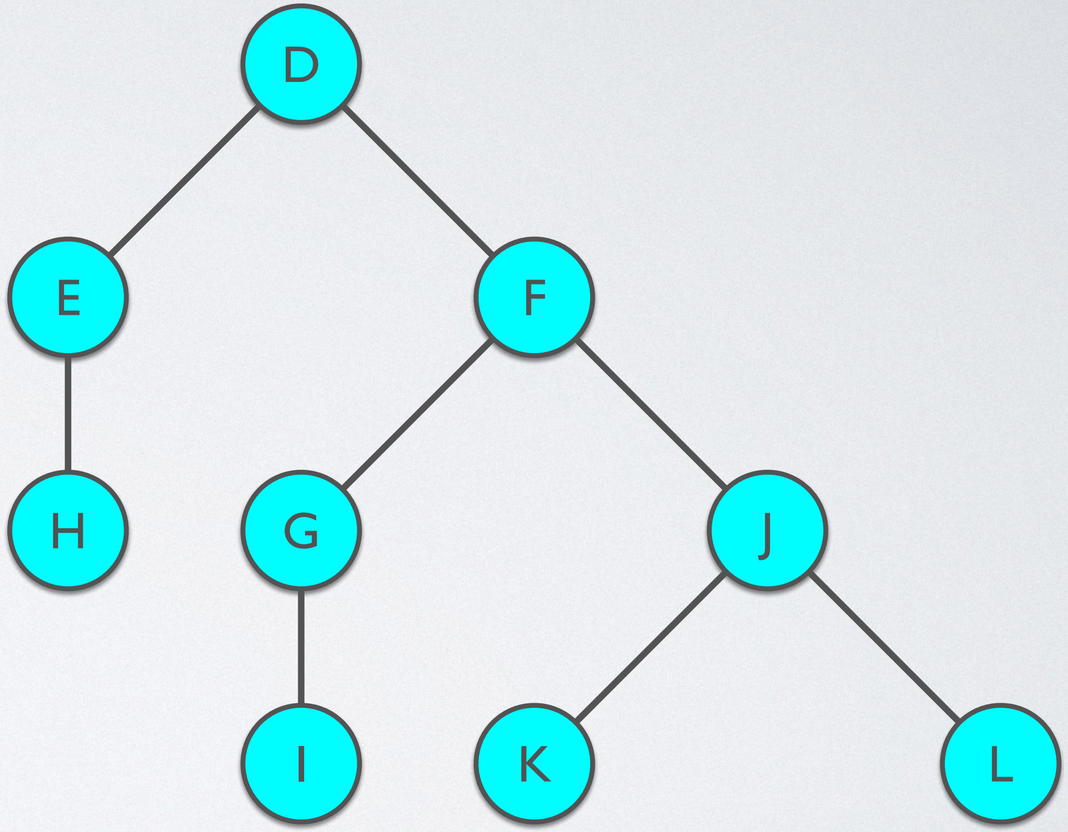
EXAMPLE TREES

Ta:

Tb:



Tc:



Ta = {A}

Tb = {B, {C}}

Tc = {D, {E, {H}}, {F, {G, {I}}, {J, {K}, {L}}}}

DEGREE OF A TREE

- The degree of a node is the number of sub-trees associated with that node. For example, the degree of $Tc = \{D, \{E, \{H\}\}, \{F, \{G,\{I\}\}, \{J, \{K\}, \{L\}\}\}\}$ is 2.

- A node of degree zero has no subtrees. Such a node is called a leaf. For example, the leaves of Tc are H, I, K, L.

- Two roots r_i and r_j, of distinct subtrees T_i and T_j with the same parent in tree T are called siblings. For example, $T_i = \{G, \{I\}\}$ and $T_j = \{J, \{K\}, \{L\}\}$ are siblings in Tc.

PATH AND PATH LENGTH

- Given a tree T containing the set of nodes R, a path p in T is defined as a non-empty sequence of nodes

$$p = \{r_1, r_2, \ldots, r_k\}$$

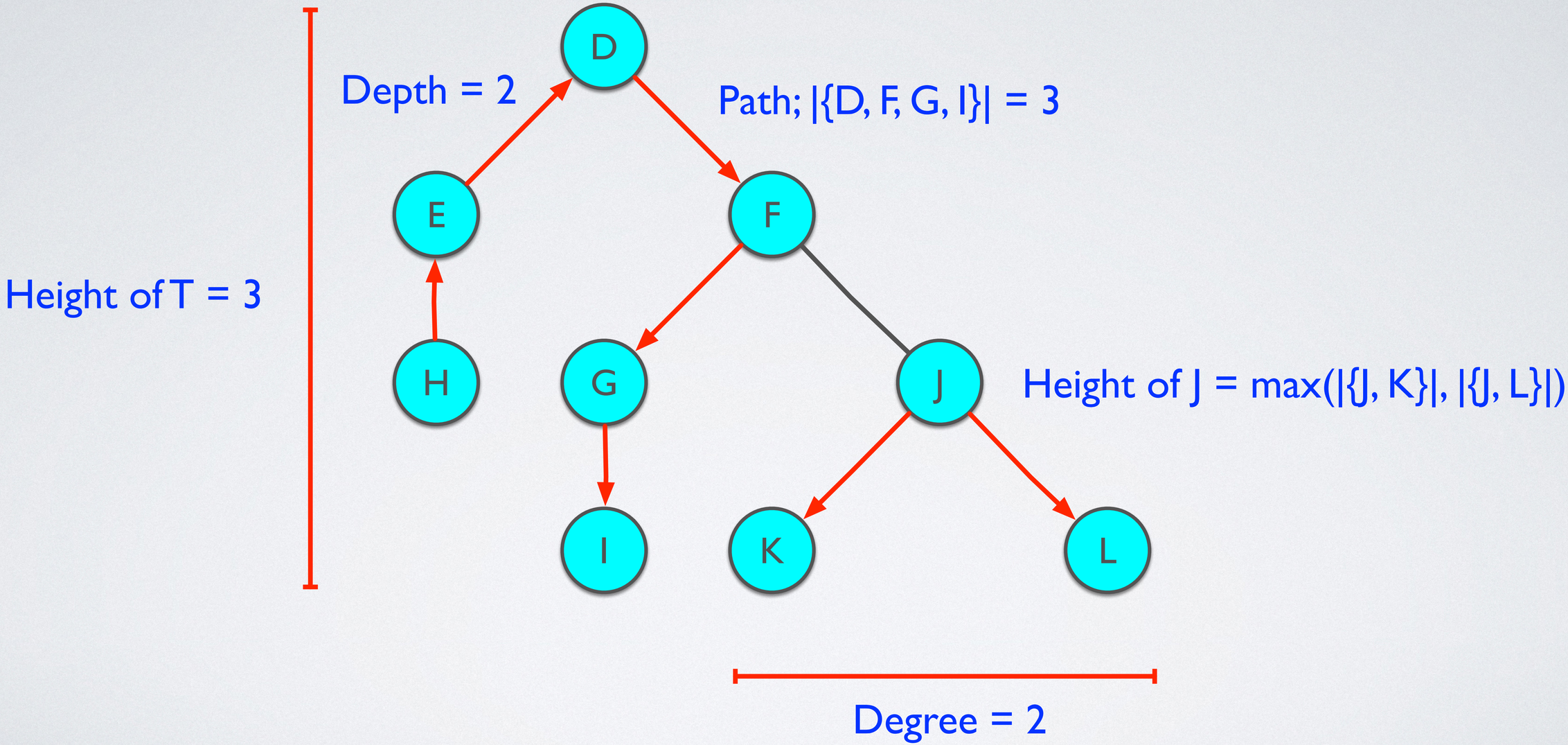
where $r_i \in R$, for $1 \leq i \leq k$ such that the ith node in the sequence, r_i, is the parent of the $(i+1)$th node in the sequence, r_i+1.

- The length of path p, $|p|$, is $k-1$, which corresponds to the distance from the root r_i to the leaf r_k in tree T.

DEPTH AND HEIGHT

- The depth of a node $r_i \in R$ in a tree T is the length of the unique path p in T from its root to the node r_i.
- The height of a node $r_i \in R$ in a tree T is the length of the longest path from node r_i to a leaf. Consequently, all leaves have a height zero.
- The height of a tree T is the height of its root node r.

DEGREE, PATH, DEPTH, AND HEIGHT



TREE NODES WITH THE SAME DEGREE

- The general case allows each node in a tree to have a different degree.
- We now consider a variant of trees in which each node has the same degree.
- Unfortunately, it is impossible to construct a tree with a finite number of nodes that all have the same degree N, except the trivial case N = 0.
- We need a special notation, called empty tree, to realize trees in which all nodes have the same degree.

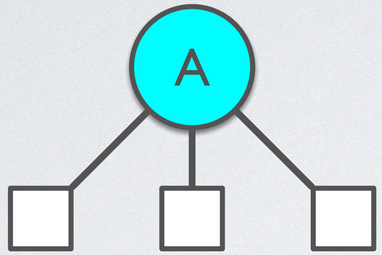
N-ARY TREES

- An N-ary tree T, $N \geq 1$, is a finite set of nodes with one of the following properties:
  - Either the set is empty, $T = \varnothing$, or
  - The set consists of a root, R, and exactly N distinct N-ary trees. That is, the remaining nodes are partitioned into $N \geq 1$ subsets, $T_1, T_2, \ldots, T_N$, each of which is an N-ary tree such that

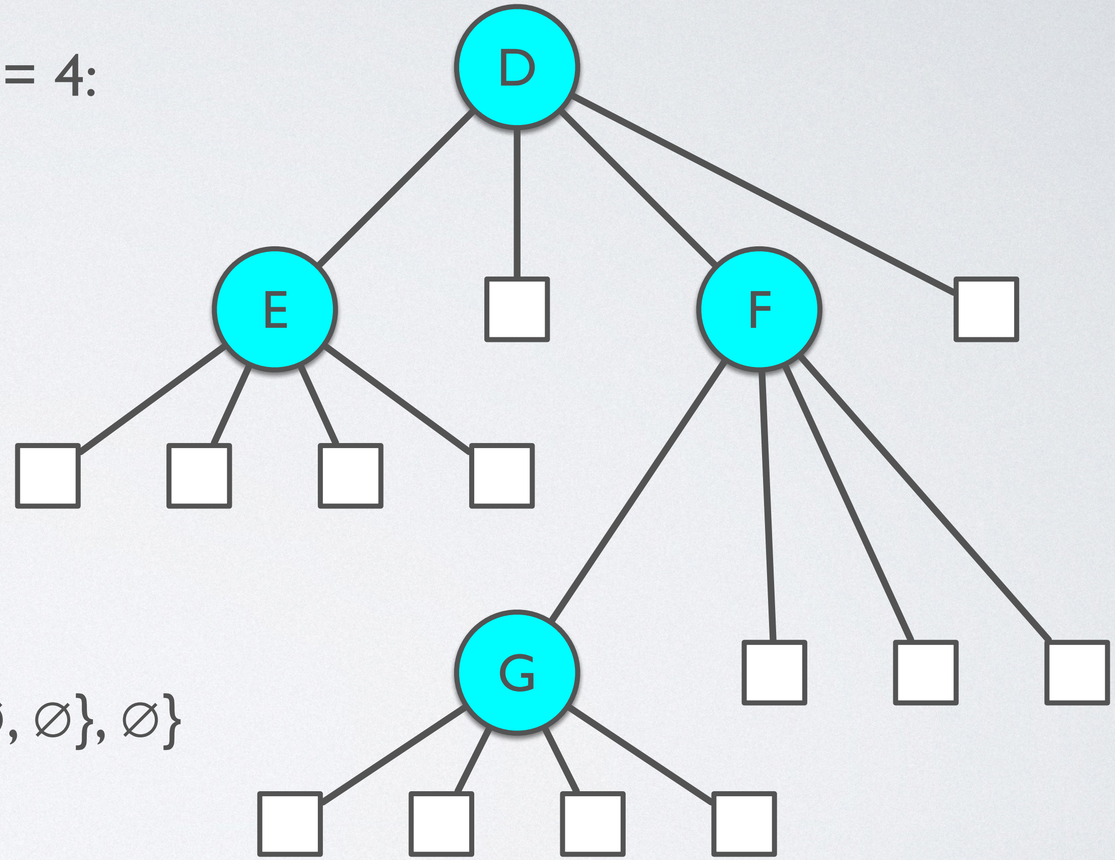
$$T = \{R, T_1, T_2, \ldots, T_N\}.$$

EXAMPLE N-ARY TREES

Ta; N = 3:



Tb; N = 4:



Ta = {A, ∅, ∅, ∅}

Tb = {D, {E, ∅, ∅, ∅, ∅}, ∅, {F, {G, ∅, ∅, ∅, ∅}, ∅, ∅, ∅}, ∅}

THE EMPTY TREE

- The empty tree, $T = \varnothing$, is a tree.
- From the modeling point of view, an empty N-ary tree:
  - has no key, and
  - has to have the same type as a non-empty N-ary tree.
- Using null (i.e., **nullptr**) to denote an empty N-ary tree may be unsuitable, as null refers to nothing!

SENTINEL NODE

- A sentinel node is a programming idiom used to facilitate tree-based operations.
- A sentinel node in tree structures indicates a node with no children.
- Sentinel nodes behave like null-pointers. However, unlike null-pointers, which refer to nothing, sentinel nodes denote proper, yet empty, subtrees.
- Smart pointers allow us to represent sentinel nodes and non-empty trees equally. In particular, for an N-ary tree, we employ unique smart pointers (i.e., std::unique_ptr) to represent N-ary tree nodes. In the case of the sentinel node, the smart pointer wraps **nullptr**, whereas for non-empty N-ary trees, it wraps a unique, ownership-managed heap location.

```
template<typename T, size_t N>
class NTree
{
public:
   using node = std::unique_ptr<NTree>;

   NTree( const T& aKey = T{} ) noexcept;
   NTree( T&& aKey ) noexcept;

   template<typename... Args>
   static Node makeNode( Args&&... args );

   NTree( const NTree& aOther );
   NTree& operator=( const NTree& aOther );

   NTree( NTree&& aOther ) noexcept;
   NTree& operator=( NTree&& aOther ) noexcept;

   void swap( NTree& aOther ) noexcept;

   const T& operator*() const noexcept;
   NTree& operator[]( size_t aIndex ) const;

   void attach( size_t aIndex, node& aNode );
   node detach( size_t aIndex );

   bool leaf() const noexcept;
   size_t height() const noexcept;

private:
   T fKey;
   node fNodes[N];
};
```

NTREE<T, N>

- Class NTree is parameterized over the key type T and the degree of the tree N.
- Objects of class NTree do not support shared ownership. Nodes take exclusive ownership.
- Attaching and detaching sub-trees transfers ownership.
- The indexer returns a reference to the corresponding node. The ownership of a node remains with the tree, but we can obtain the address of the returned node.
- NTree defines proper copy and move semantics. The synthesized destructor works as expected.
- The auxiliary methods leaf() and height() test a node for the leaf property and return the height, respectively.

NTREE<T,N> CONSTRUCTORS

```
NTree( const T& aKey = T{} ) noexcept :
        fKey(aKey)
{}

NTree( T&& aKey ) noexcept :
    fKey(std::move(aKey))
{}
```

- We can construct an N-ary tree node by copying the key value or by moving the key value.
- In the lvalue constructor, we can initialize an N-ary tree node with a default key, T{}. Hence, this constructor can be used as the default constructor.
- The constructors create leaf nodes. All sub-trees are initialized as sentinel nodes through the default initialization of std::unique_ptr.

MAKENODE METHOD

```
template<typename... Args>
static Node makeNode( Args&&... args )
{
    return std::make_unique<NTree>( std::forward<Args>(args)... );
}
```

- The static method makeNode() creates N-ary tree objects with unique ownership semantics.

- We use a variadic template method and perfect forwarding to allow for different ways to create N-ary tree nodes. In particular, makeNode() can create new N-ary tree nodes from keys and other N-ary tree nodes using copy and move semantics.

KEY AND SUB-TREE ACCESS

```
const T& operator*() const noexcept
{
    return fKey;
}

NTree& operator[]( size_t aIndex ) const
{
    // valid index and node
    assert( aIndex < N && fNodes[aIndex] );

    return *fNodes[aIndex];
}
```

- The dereference operator returns the key value as a constant reference.
- The node indexer returns a reference to the indexed sub-tree. Ownership of this sub-tree remains unaffected.
- The indexer checks that we select a valid tree node (i.e., within range of N and not a sentinel node).
- **Note**: It is impossible to obtain a reference to **nullptr**. Hence, the indexer cannot return a reference to a sentinel node. But this has the desired effect.

ATTACHING A SUB-TREE

```
void attach( size_t aIndex, Node& aNode )
{
    // transfers ownership from aNode to fNodes[aIndex]

    assert( aIndex < N&& !fNodes[aIndex] );

    fNodes[aIndex] = std::move(aNode);
}
```

- To attach an N-ary tree node to a given target node, we require a valid sub-tree index (i.e., an index occupied by a sentinel node) and transfer ownership of the sub-tree to the target node.

- Unique smart pointers can only be moved, not copied. Ownership of the managed heap space is transferred (i.e., after method attach() finishes, aNode is an empty N-ary tree node).

DETACHING A SUB-TREE

```
Node detach( size_t aIndex )
{
    // transfers ownership from fNodes[aIndex] to result

    assert( aIndex < N && fNodes[aIndex] );

    return std::move(fNodes[aIndex]);
}
```

- To detach an N-ary tree node from a given source node, we require a valid sub-tree index (i.e., an index occupied by an N-ary tree node) and transfer ownership of the sub-tree to the caller.
- After method detach() finishes, the N-ary sub-tree at index aIndex is an empty N-ary tree node (i.e., a sentinel node).

```
bool leaf() const noexcept
{
    for ( size_t i = 0; i < N; i++ )
    {
        if ( fNodes[i] )
        {
            return false;
        }
    }

    return true;
}

size_t height() const noexcept
{
    size_t Result = 0;

    if ( !leaf() )
    {
        for ( size_t i = 0; i < N; i++ )
        {
            if ( fNodes[i] )
            {
                Result = std::max( Result, fNodes[i]->height() + 1 );
            }
        }
    }

    return Result;
}
```

LEAF() AND HEIGHT()

- The method leaf() returns true if all sub-trees of the current node are sentinels. The running time is O(n).
- The method height() returns the height of the current node, which is the longest path from the current node to a leaf. Leaf nodes have height zero. The running time is O(n).

```
NTree( const NTree& aOther ) :
          fKey(aOther.fKey)
{
   for ( size_t i = 0; i < N; i++ )
   {
      if ( aOther.fNodes[i] )
      {
         // copy non-empty subtree
         fNodes[i] = std::move(makeNode( *aOther.fNodes[i] ));
      }
   }
}

NTree& operator=( const NTree& aOther )
{
   if ( this != &aOther )
   {
      this->~NTree();

      new (this) NTree( aOther );
   }

   return *this;
}
```

COPY SEMANTICS

- The copy constructor for N-ary trees first copies the root key and then recursively copies all non-empty sub-trees. The ownership of the copied sub-trees is transferred to the newly created N-ary tree node.
- The copy assignment operator is defined in the standard way, first releasing all resources and then performing an in-place copy construction.

MOVE SEMANTICS

- The move operations are defined in the standard way and rely on the existence of a suitable swap() method.
- Once a move operation has been completed, the source node, aOther, is an empty tree.

```
NTree( NTree&& aOther ) noexcept :
    NTree()
{
    swap( aOther );
}

NTree& operator=( NTree&& aOther ) noexcept
{
    if ( this != &aOther )
    {
        swap( aOther );
    }

    return *this;
}
```

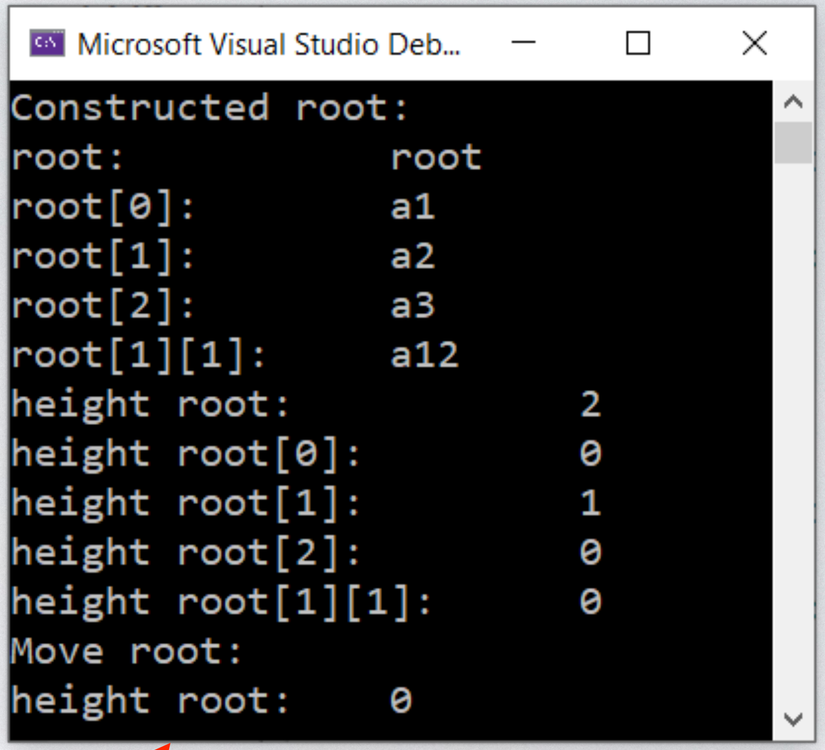
SWAP FOR N-ARY TREES

```
void swap( NTree& aOther ) noexcept
{
    std::swap( fKey, aOther.fKey );

    for ( size_t i = 0; i < N; i++ )
    {
        std::swap( fNodes[i], aOther.fNodes[i] );
     }
}
```

- The method swap exchanges the root keys and the sub-trees.
- The ownership of the sub-trees is mutually transferred.
- The method std::swap() internally uses move semantics to and from a temporary variable. This guarantees that unique smart pointers can be swapped, that is, their heap locations and the ownership can be mutually exchanged.

NTREE<T,N> EXAMPLE

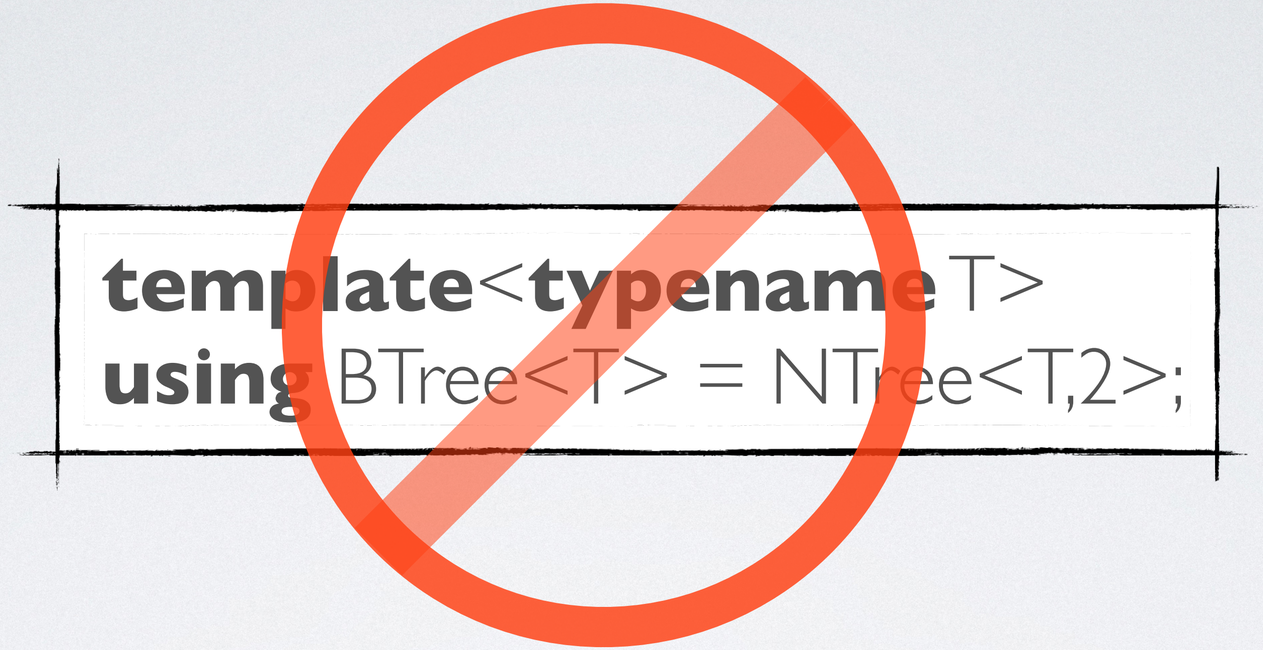
```
std::string a( "root" );
std::string a1( "a1" );
std::string a2( "a2" );
std::string a3( "a3" );
std::string a12( "a12" );

using NS3Tree = NTree<std::string, 3u>;
using NS3TreeNode = NS3Tree::Node;

NS3TreeNode root = NS3Tree::makeNode( a );
NS3TreeNode n1 = NS3Tree::makeNode( a1 );
NS3TreeNode n2 = NS3Tree::makeNode( a2 );
NS3TreeNode n3 = NS3Tree::makeNode( a3 );
NS3TreeNode n12 = NS3Tree::makeNode( a12 );

root->attach( 0u, n1 );
root->attach( 1u, n2 );
root->attach( 2u, n3 );
(*root)[1].attach( 1u, n12 );

std::cout << "Constructed root:" << std::endl;

std::cout << "root:\t\t" << **root << std::endl;
std::cout << "root[0]:\t" << *(*root)[0u] << std::endl;
std::cout << "root[1]:\t" << *(*root)[1u] << std::endl;
std::cout << "root[2]:\t" << *(*root)[2u] << std::endl;
std::cout << "root[1][1]:\t" << *(*root)[1u][1u] << std::endl;

std::cout << "height root:\t\t" << root->height() << std::endl;
std::cout << "height root[0]:\t\t" << (*root)[0u].height() << std::endl;
std::cout << "height root[1]:\t\t" << (*root)[1u].height() << std::endl;
std::cout << "height root[2]:\t\t" << (*root)[2u].height() << std::endl;
std::cout << "height root[1][1]:\t" << (*root)[1u][1u].height() << std::endl;

std::cout << "Move root:" << std::endl;

NS3TreeNode move = NS3Tree::makeNode( std::move(*root) );

std::cout << "height root:\t" << root->height() << std::endl;
```



2-ARY TREES: BINARY TREES

- A binary tree T is a finite set of nodes with one of the following properties:
  - Either the set is empty, $T = \varnothing$, or
  - The set consists of a root, r, and exactly 2 distinct binary trees T_L and T_R, $T = \{r, T_L, T_R\}$.
- The tree T_L is called the left subtree of T and the tree T_R is called the right subtree of T.

SEMANTIC MISMATCH



- Representing a binary tree simply as a type alias to NTree<T,2> works. Unfortunately, this type alias does not convey the intended semantics of a binary tree, namely that a binary tree has a dedicated left and right sub-tree.

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

    BTree( const T& aKey = T{} ) noexcept;
    BTree( T&& aKey ) noexcept;

    template<typename... Args>
    static node makeNode( Args&&... args );

    BTree( const BTree& aOther );
    BTree& operator=( const BTree& aOther );

    BTree( BTree&& aOther ) noexcept;
    BTree& operator=( BTree&& aOther ) noexcept;

    void swap( BTree& aOther ) noexcept;

    const T& operator*() const noexcept;
    bool hasLeft() const noexcept;
    BTree& left() const;
    bool hasRight() const noexcept;
    BTree& right() const;

    void attachLeft( node& aNode );
    void attachRight( node& aNode );

    node detachLeft();
    node detachRight();

    bool leaf() const noexcept;
    size_t height() const noexcept;

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

BTREE<T>

- The template class BTree<T> defines the semantics of a binary tree.
- BTree<T> provides dedicated access to the left and right sub-tree of a given binary tree node.
- The implementation follows that for N-ary tree with the exception that BTree<T> defines two predicates: hasLeft() and hasRight(), which return **true** if the corresponding sub-tree is not an empty tree. These predicates are required for tree traversal.
- We cannot simply use subclassing and define BTree<T> as a subtype of NTree<T, 2>. The methods hasLeft() and hasRight() require access to the sub-tree nodes, which are private in NTree. Exposing the sub-trees to sub-classes is not safe.