COS30008
Data Structures and Patterns

Trees

Trees

Overview

- Trees
- Search Trees

References

- Bruno R. Preiss: Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc. (1999)
- Richard F. Gilberg and Behrouz A. Forouzan: Data Structures - A Pseudocode Approach with C. 2nd Edition. Thomson (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, and Clifford Stein: Introduction to Algorithms. 2nd Edition. The MIT Press (2001)

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, …, T_n, each of which is a tree.

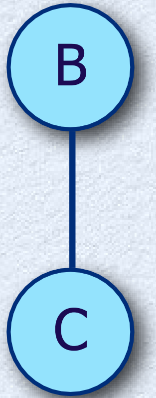
Parent, Children, and Leaf

- 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.

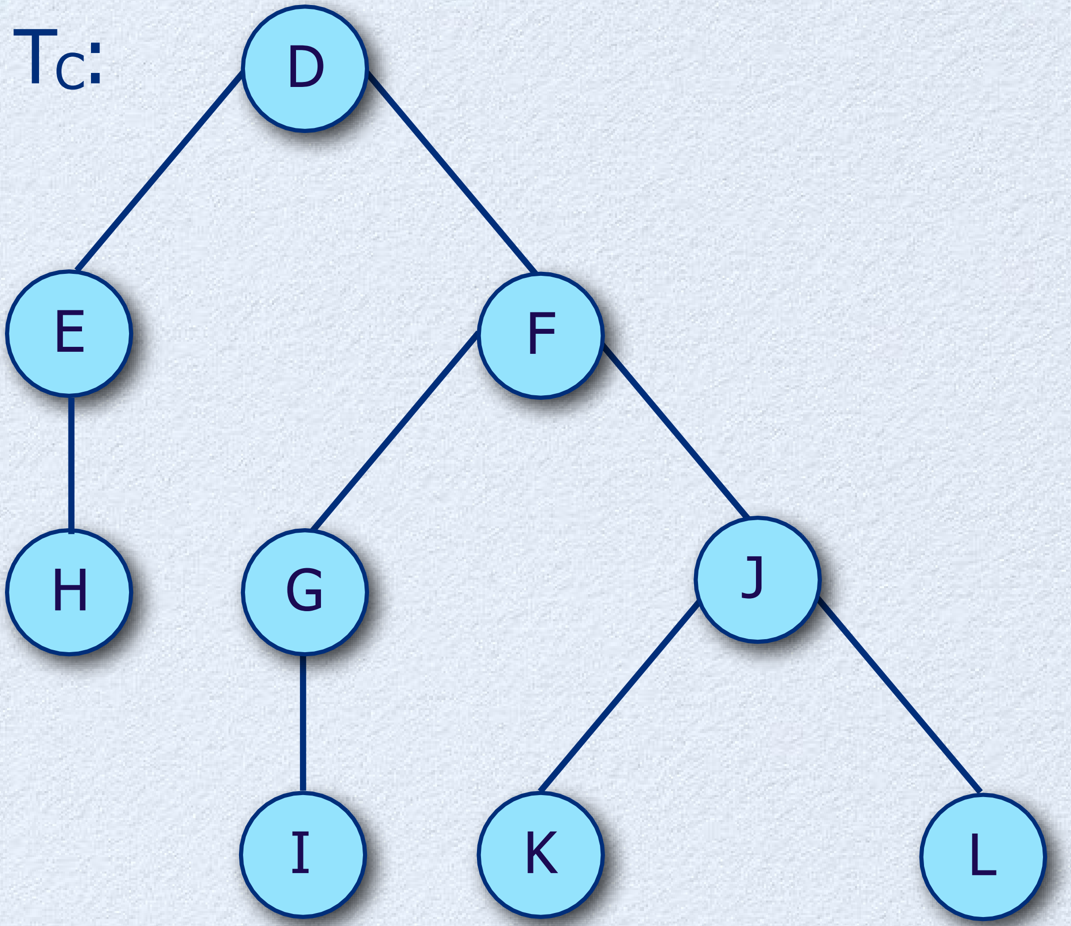
Tree Examples

T_A:

T_B:



T_C:



$T_A = \{A\}$

$T_B = \{B, \{C\}\}$

$T_C = \{D, \{E, \{H\}\}, \{F, \{G, \{I\}\}, \{J, \{K\}, \{L\}\}\}\}$

Degree

- The degree of a node is the number of subtrees associated with that node. For example, the degree of T_C = {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 T_C 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 T_C.

Path and Path Length

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

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

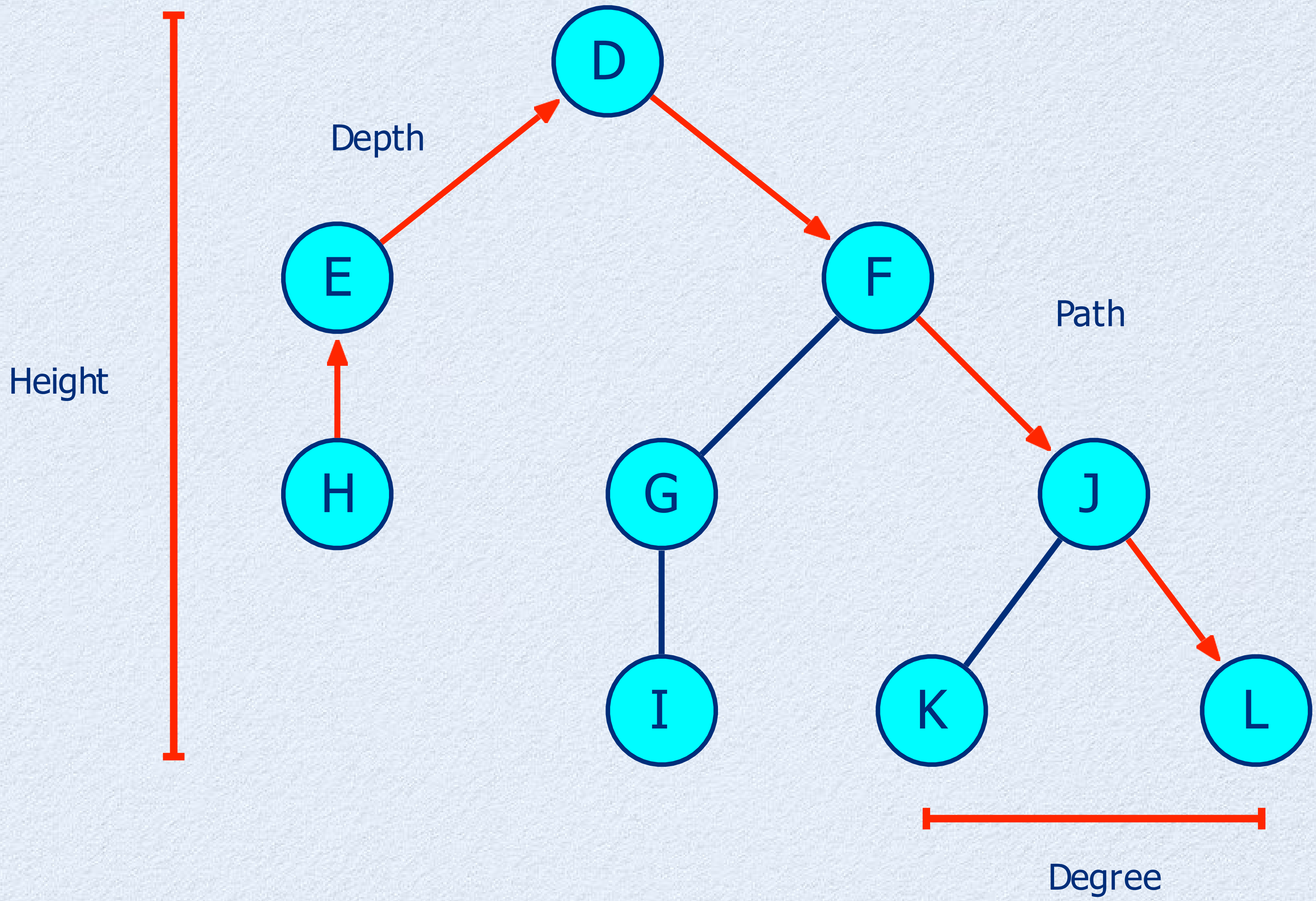
where $r_i \in R$, for $1 \leq i \leq k$ such that the ith node in the sequence, r_i, is the parent of the (i+1)th node in the sequence r_{i+1}.

- The length of path P is k-1, which corresponds to the distance from the root r_i to the leaf r_k.

Depth and Height

- The depth of a node $r_i \in R$ in a tree T is the length of the unique path 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. Therefore, the leaves are all at height zero.
- The height of a tree T is the height of its root node r.

Path, Depth, and Height



Nodes With the Same Degree

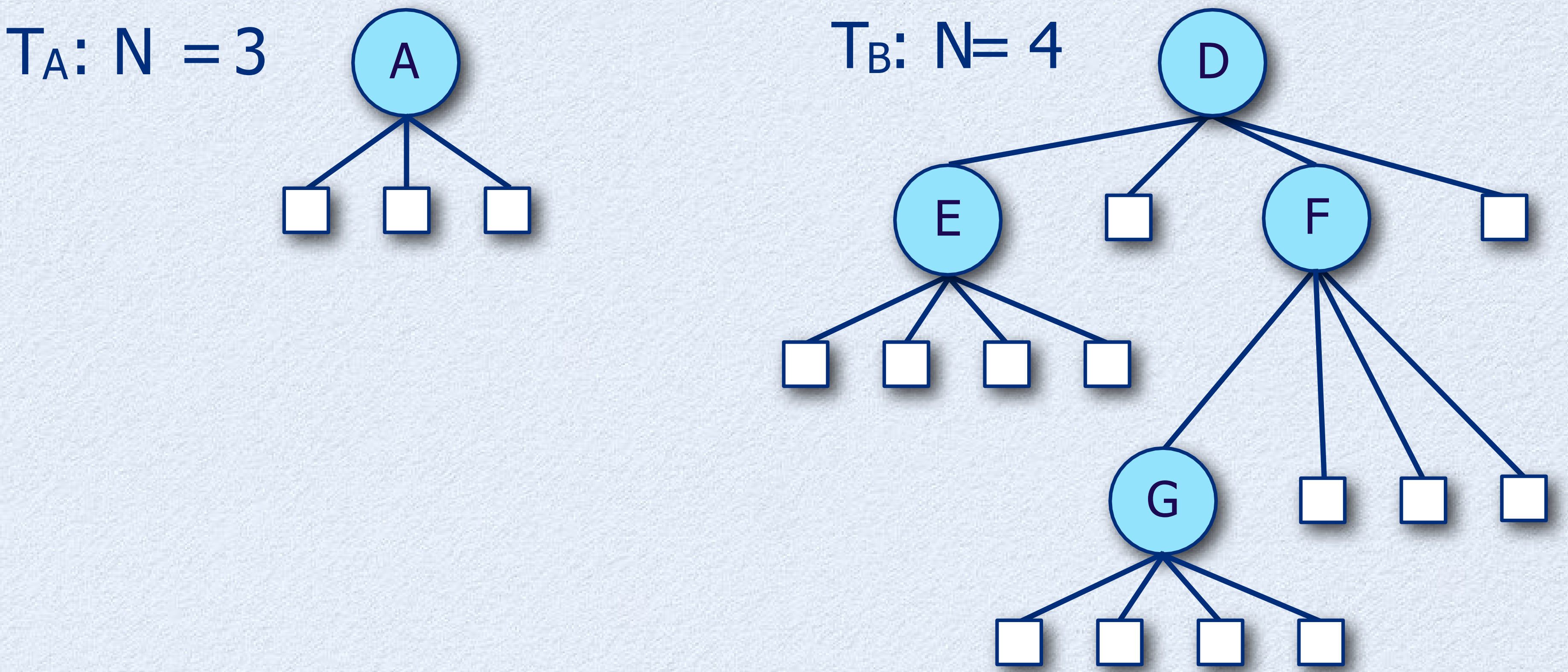
- The general case allows each node in a tree to have a different degree. We now consider a variant of trees in which each node has the same degree.
- Unfortunately, it is not possible to construct a tree that has a finite number of nodes which all have the same degree N, except the trivial case N = 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, $T = \emptyset$, 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, …, T_N, each of which is an N-ary tree such that

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

N-ary Tree Examples



$T_A = \{A, \emptyset, \emptyset, \emptyset\}$

$T_B = \{D, \{E, \emptyset, \emptyset, \emptyset, \emptyset\}, \emptyset, \{F, \{G, \emptyset, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset, \emptyset\}, \emptyset\}$

The Empty Tree

- The empty tree, $T = \emptyset$, 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.
- To use null (i.e., nullptr) to denote an empty N-ary tree is inappropriate, as null refers to nothing at all!

Sentinel Node: NIL

- 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.

Class Template NTree<T,N>

NTree SPEC.h — NTrees

```cpp
template<typename T, size_t N>
class NTree
{
private:
    T fKey;                                              // T() for empty NTree
    NTree<T,N>* fNodes[N];                               // N subtrees of degree N

    void initLeaves();                                   // initialize subtree nodes

    NTree();                                             // sentinel constructor

public:
    static NTree<T,N> NIL;                               // Empty NTree

    NTree( const T& aKey );                              // NTree leaf
    NTree( T&& aKey );                                   // NTree leaf

    NTree( const NTree& aOtherNTree );                   // copy constructor
    NTree( NTree&& aOtherNTree );                        // move constructor

    virtual ~NTree();                                    // destructor

    NTree& operator=( const NTree& aOtherNTree );        // copy assignment operator
    NTree& operator=( NTree&& aOtherNTree );             // move assignment operator

    virtual NTree* clone();                              // clone a tree

    bool empty() const;                                  // is tree empty
    const T& operator*() const;                          // get key (node value)

    const NTree& operator[]( size_t aIndex ) const;      // indexer

    // tree manipulators
    void attach( size_t aIndex, const NTree<T,N>& aNTree );
    const NTree& detach( size_t aIndex );
};
```

Line: 1 C++ Tab Size: 4

We do not wish to allow clients to create empty NTrees.

The Private NTree<T,N> Constructor

NTree.h — NTrees

```cpp
    void initLeaves()
    {
        for ( size_t i = 0; i < N; i++ )
        {
            fNodes[i] = &NIL;
        }
    }

    NTree() : fKey(T())                                     // sentinel constructor
    {
        initLeaves();
    }
```

- We use T(), the default constructor for type T, to initialize the fKey.
- Each subtree-node is set to to the location of NIL, the sentinel node for NTree<T,N> using initLeaves().
- This constructor is solely being used to set up the sentinel for NTree<T,N>. Clients should and cannot use the default constructor.

The Public NTree<T,N> Constructors

NTree.h — NTrees

```
    NTree( const T& aKey ) : fKey(aKey)                // NTree leaf
    {
        initLeaves();
    }

    NTree( T&& aKey ) : fKey(std::move(aKey))          // NTree leaf
    {
        initLeaves();
    }

```

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

- We copy (or move) aKey into fKey.
- Each child node in a non-empty NTree<T,N> leaf node is set to the location of NIL, the sentinel node for NTree<T,N> using initLeaves().

The NTree<T,N> Destructor

```cpp
    virtual ~NTree()                                        // destructor
    {
        if ( empty() )
        {
            std::cout << "Delete NIL" << std::endl;
        }

        for ( size_t i = 0; i < N; i++ )
        {
            if ( !fNodes[i]->empty() )                      // don't delete NIL
            {
                std::cout << "Delete node: " << fNodes[i]->fKey << std::endl;
                delete fNodes[i];
            }
        }
    }
```

- In the destructor of NTree<T,N> only non-sentinel nodes are destroyed.
- The output is for debugging purposes only.

The NTree<T,N> Sentinel

```cpp
template<class T, size_t N>
NTree<T,N> NTree<T,N>::NIL;
```

- Static instance variables, like the NTree<T,N> sentinel NIL, need to be initialized outside the class definition.
- Here, NIL is initialized using the private default constructor.
- The scope of NIL is NTree<T,N>, which means that all members of NTree<T,N> are available, including the private constructor to initialize NIL.

The NTree<T,N> Auxiliaries

```cpp
    bool empty() const                              // is tree empty
    {
        return this == &NIL;
    }

    const T& operator*() const                      // get key (node value)
    {
        return fKey;
    }
```

- A tree of type NTree<T,N> is empty if it is equal to the sentinel NIL.
- The dereference operator returns the payload (i.e., the root) of a NTree<T,N> tree. (We can use NIL as temporary storage.)

Attaching a New Subtree

NTree.h

```
void attach( size_t aIndex, const NTree<T,N>& aNTree )
{
    if ( !empty() )
    {
        if ( aIndex < N )
        {
            if ( fNodes[aIndex]->empty() )
            {
                fNodes[aIndex] = const_cast<NTree<T,N>*>(&aNTree);
            }
            else
            {
                throw std::domain_error( "Non-empty subtree present!" );
            }
        }
        else
        {
            throw std::out_of_range( "Illegal subtree index!" );
        }
    }
    else
    {
        throw std::domain_error( "Empty NTree!" );
    }
}
```

Line: 1 Column: 1 C++ Tab Size: 4 —

Accessing a Subtree

NTree.h

```cpp
    const NTree& operator[]( size_t aIndex ) const  // indexer
    {
        if ( !empty() )
        {
            if ( aIndex < N )
            {
                return *fNodes[aIndex];             // return reference to subtree
            }
            else
            {
                throw std::out_of_range( "Illegal subtree index!" );
            }
        }
        else
        {
            throw std::domain_error( "Empty NTree!" );
        }
    }
```

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

- We return a reference to the subtree rather than a pointer. This way, we prevent accidental manipulations outside the tree structure.

Removing a Subtree

NTree.h

```
    const NTree& detach( size_t aIndex )
    {
        if ( !empty() )
        {
            if ( (aIndex < N) && !fNodes[aIndex]->empty() )
            {
                const NTree<T,N>& Result = *fNodes[aIndex];     // obtain reference to subtree
                fNodes[aIndex] = &NIL;                          // set to NIL
                return Result;                                  // return subtree (reference)
            }
            else
            {
                throw std::out_of_range( "Illegal subtree index!" );
            }
        }
        else
        {
            throw std::domain_error( "Empty NTree!" );
        }
    }
```

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

Copy Semantics

```cpp
NTree( const NTree& aOtherNTree )              // copy constructor
{
    initLeaves();

    *this = aOtherNTree;
}

NTree& operator=( const NTree& aOtherNTree )    // copy assignment operator
{
    if ( this != &aOtherNTree )
    {
        if ( !aOtherNTree.empty() )
        {
            this->~NTree();

            fKey = aOtherNTree.fKey;

            for ( size_t i = 0; i < N; i++ )
            {
                if ( !aOtherNTree.fNodes[i]->empty() )
                {
                    fNodes[i] = aOtherNTree.fNodes[i]->clone();
                }
                else
                {
                    fNodes[i] = &NIL;
                }
            }
        }
        else
        {
            throw std::domain_error( "Copying of NIL detected." );
        }
    }

    return *this;
}
```

NTree.h — NTrees

Line: 104 C++ Tab Size: 4 NTree

Move Semantics

NTree.h — NTrees

```cpp
    NTree( NTree&& aOtherNTree )                        // move constructor
    {
        initLeaves();

        *this = std::move(aOtherNTree);
    }

    NTree& operator=( NTree&& aOtherNTree )             // move assignment operator
    {
        if ( this != &aOtherNTree )
        {
            if ( !aOtherNTree.empty() )
            {
                this->~NTree();

                fKey = std::move(aOtherNTree.fKey);

                for ( size_t i = 0; i < N; i++ )
                {
                    if ( !aOtherNTree.fNodes[i]->empty() )
                    {
                        fNodes[i] = const_cast<NTree<T,N>*>(&aOtherNTree.detach( i ));
                    }
                    else
                    {
                        fNodes[i] = &NIL;
                    }
                }
            }
            else
            {
                throw std::domain_error( "Moving of NIL detected." );
            }
        }

        return *this;
    }
```

Line: 104 C++ Tab Size: 4 NTree

Steal memory

Clone

NTree.h — NTrees

```cpp
    virtual NTree* clone()                          // clone a tree
    {
        if ( !empty() )
        {
            return new NTree( *this );
        }
        else
        {
            throw std::domain_error( "Cloning of NIL detected." );
        }
    }
```

Line: 104 C++ Tab Size: 4 NTree

- The method clone() must not duplicate NIL.
- The sentinel NIL is a unique instance of NTree<N,T> for every instantiation of N and T.

A NTree<T,N> Example

Main.cpp

```cpp
void testBasicOperations()
{
    using NS3Tree = NTree<string,3>;

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

    NS3Tree root( "Hello World!" );
    NS3Tree nodeA( s1 );
    NS3Tree nodeB( s2 );
    NS3Tree nodeC( s3 );
    NS3Tree nodeAB( "AB" );

    root.attach( 0, nodeA );
    root.attach( 1, nodeB );
    root.attach( 2, nodeC );
    const_cast<NS3Tree&>(root[1]).attach( 1, nodeAB );

    cout << "root:        " << *root << endl;
    cout << "root[0]:     " << *root[0] << endl;
    cout << "root[1]:     " << *root[1] << endl;
    cout << "root[2]:     " << *root[2] << endl;
    cout << "root[1][1]: " << *root[1][1] << endl;

    const_cast<NS3Tree&>(root[1]).detach( 1 );
    root.detach( 0 );
    root.detach( 1 );
    root.detach( 2 );
}
```

Line: 1 Column: 1 C++ Tab Size: 4 —

NTrees

```
Kamala:NTrees Markus$ ./NTreeTest
root:       Hello World!
root[0]:    A
root[1]:    B
root[2]:    C
root[1][1]: AB
Kamala:NTrees Markus$
```

See full test on Canvas

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, $T = \emptyset$, 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.

We cannot just create a top-level type alias!

```
template<class T>
using BTree<T> = NTree<T,2>;
```

BTree<T>

BTree SPEC.h

```cpp
template<typename T>
class BTree
{
private:
    T fKey;                                            // T() for empty BTree
    BTree<T>* fLeft;
    BTree<T>* fRight;

    BTree();                                           // sentinel constructor

public:
    static BTree<T> NIL;                               // Empty BTree

    BTree( const T& aKey );                            // BTree leaf
    BTree( T&& aKey );                                 // BTree leaf

    BTree( const BTree& aOtherBTree );                 // copy constructor
    BTree( BTree&& aOtherBTree );                      // move constructor

    virtual ~BTree();                                  // destructor

    BTree& operator=( const BTree& aOtherBTree );      // copy assignment operator
    BTree& operator=( BTree&& aOtherBTree );           // move assignment operator

    virtual BTree* clone();                            // clone a tree

    bool empty() const;                                // is tree empty
    const T& operator*() const;                        // get key (node value)

    const BTree& left() const;
    const BTree& right() const;

    // tree manipulators
    void attachLeft( const BTree<T>& aBTree );
    void attachRight( const BTree<T>& aBTree );
    const BTree& detachLeft();
    const BTree& detachRight();
};
```

Line: 2 Column: 11 C++ Tab Size: 4