

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,

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

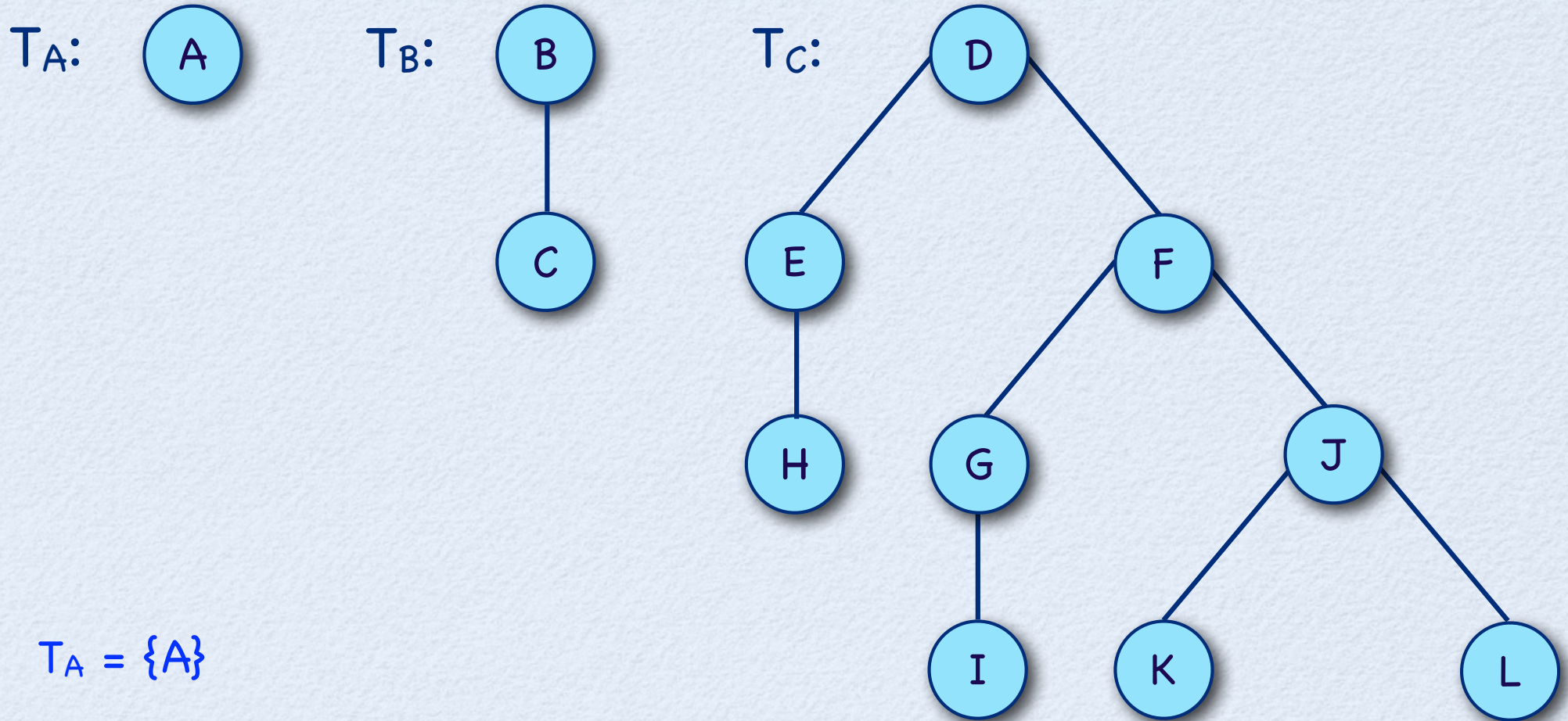
with the following properties:

- 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, \dots, 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



$$T_A = \{A\}$$

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

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

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

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

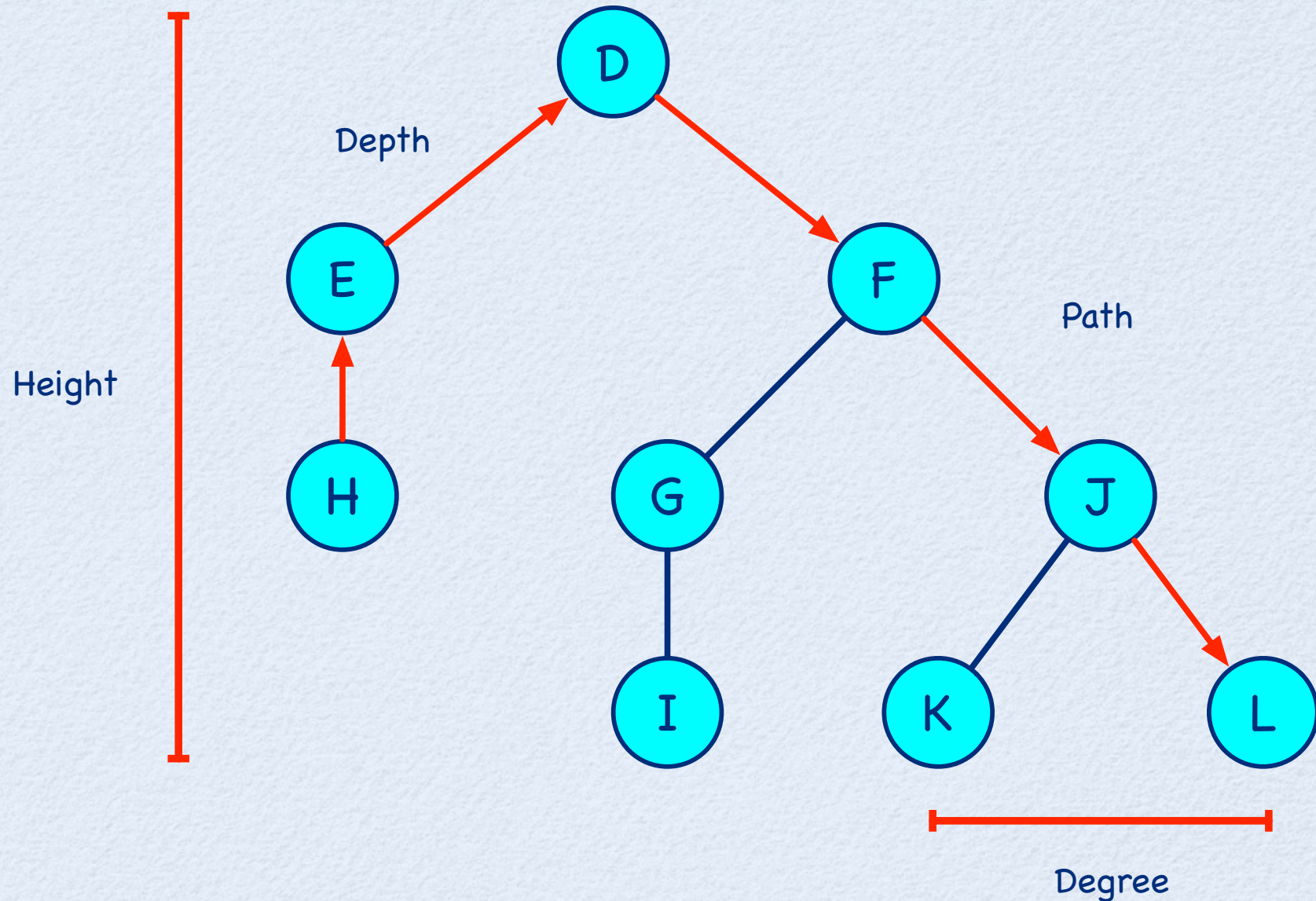
where $r_i \in R$, for $1 \leq i \leq k$ such that the i th node in the sequence, r_i , is the parent of the $(i+1)$ th node in the sequence r_{i+1} .

- The **length** of path P is $k-1$, which corresponds to the distance from the root r_1 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

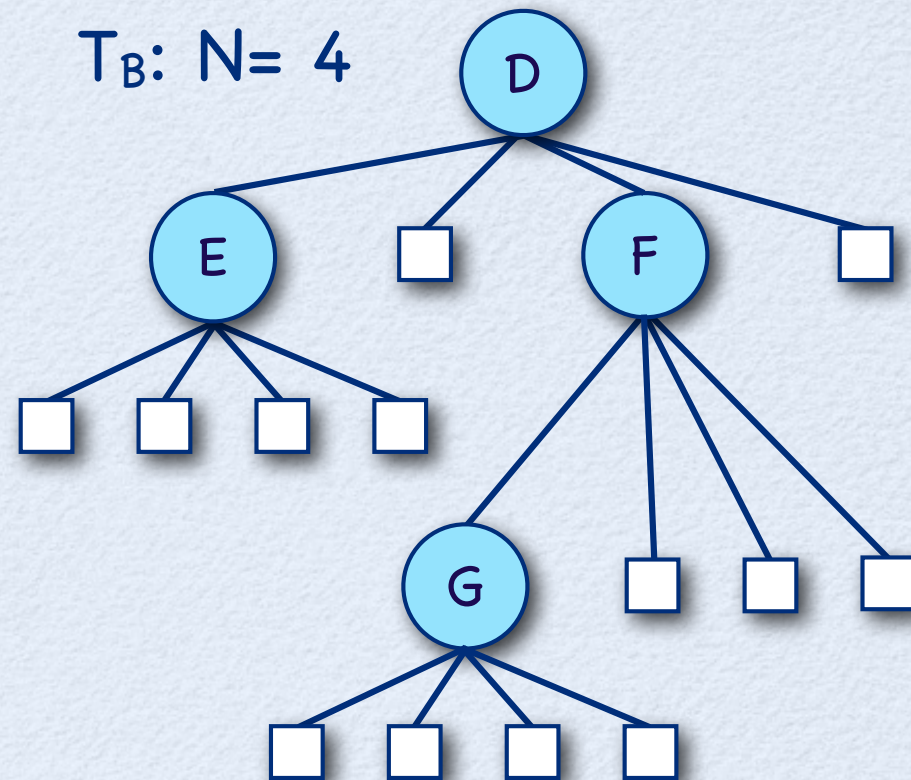
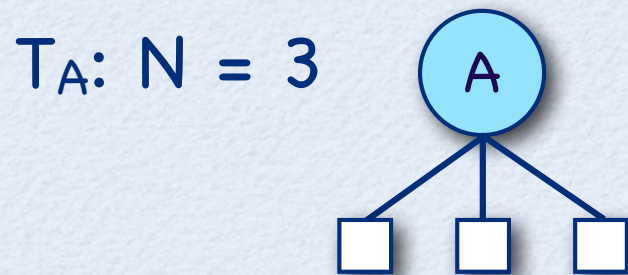
- The general case allows each node in a tree to have a different degree. We now consider a variant of trees in which each node has the same degree.
- Unfortunately, it is not possible to construct a tree that has a finite number of nodes which all have the same degree N , except the trivial case $N = 0$.
- 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, \dots, T_N , each of which is an N-ary tree such that

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

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

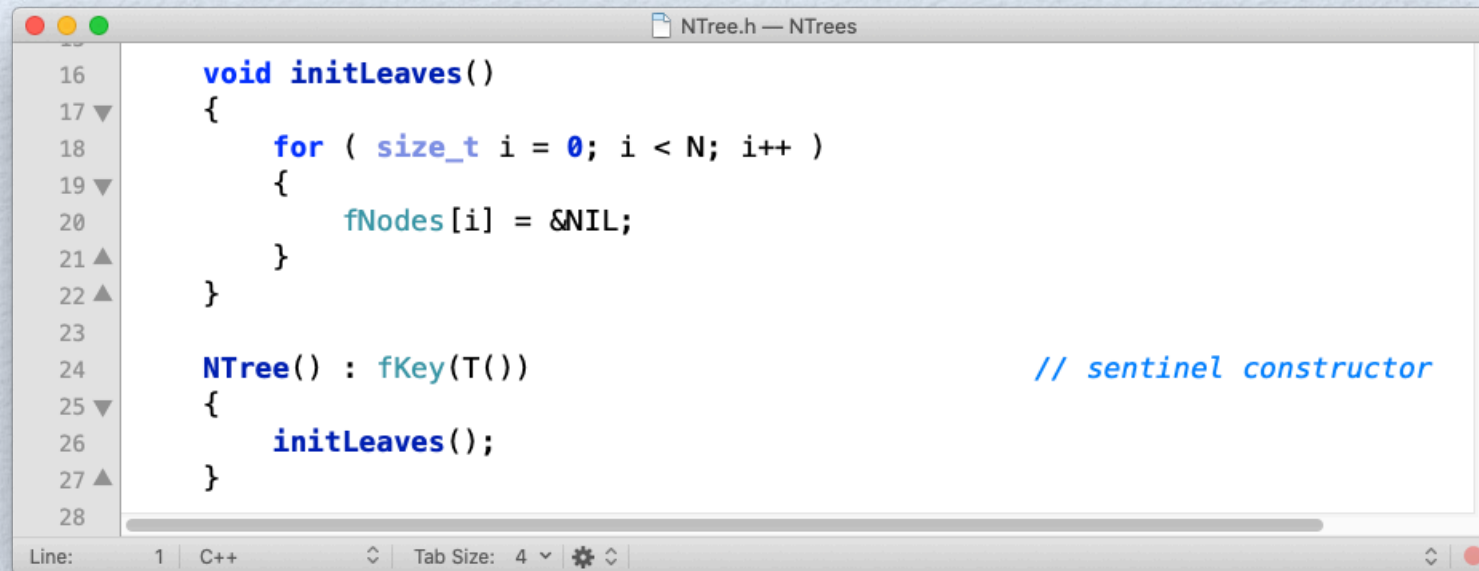
- A sentinel node is a programming idiom used to facilitate tree-based operations.
- A sentinel node in tree structures indicates a node with no children.
- Sentinel nodes behave like null-pointers. However, unlike null-pointers, which refer to nothing, sentinel nodes denote proper, yet empty, subtrees.

Class Template NTree<T,N>

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

```
NTree SPEC.h — NTrees
6  template<typename T, size_t N>
7  class NTree
8  {
9  private:
10     T fKey; // T() for empty NTree
11     NTree<T,N>* fNodes[N]; // N subtrees of degree N
12
13     void initLeaves(); // initialize subtree nodes
14
15     NTree(); // sentinel constructor
16
17 public:
18     static NTree<T,N> NIL; // Empty NTree
19
20     NTree( const T& aKey ); // NTree leaf
21     NTree( T&& aKey ); // NTree leaf
22
23     NTree( const NTree& aOtherNTree ); // copy constructor
24     NTree( NTree&& aOtherNTree ); // move constructor
25
26     virtual ~NTree(); // destructor
27
28     NTree& operator=( const NTree& aOtherNTree ); // copy assignment operator
29     NTree& operator=( NTree&& aOtherNTree ); // move assignment operator
30
31     virtual NTree* clone(); // clone a tree
32
33     bool empty() const; // is tree empty
34     const T& operator*() const; // get key (node value)
35
36     const NTree& operator[]( size_t aIndex ) const; // indexer
37
38     // tree manipulators
39     void attach( size_t aIndex, const NTree<T,N>& aNTree );
40     const NTree& detach( size_t aIndex );
41 };
Line: 1 C++ Tab Size: 4
```


The Private NTree<T,N> Constructor



```
16 void initLeaves()
17 {
18     for ( size_t i = 0; i < N; i++ )
19     {
20         fNodes[i] = &NIL;
21     }
22 }
23
24 NTree() : fKey(T()) // sentinel constructor
25 {
26     initLeaves();
27 }
28
```

- 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

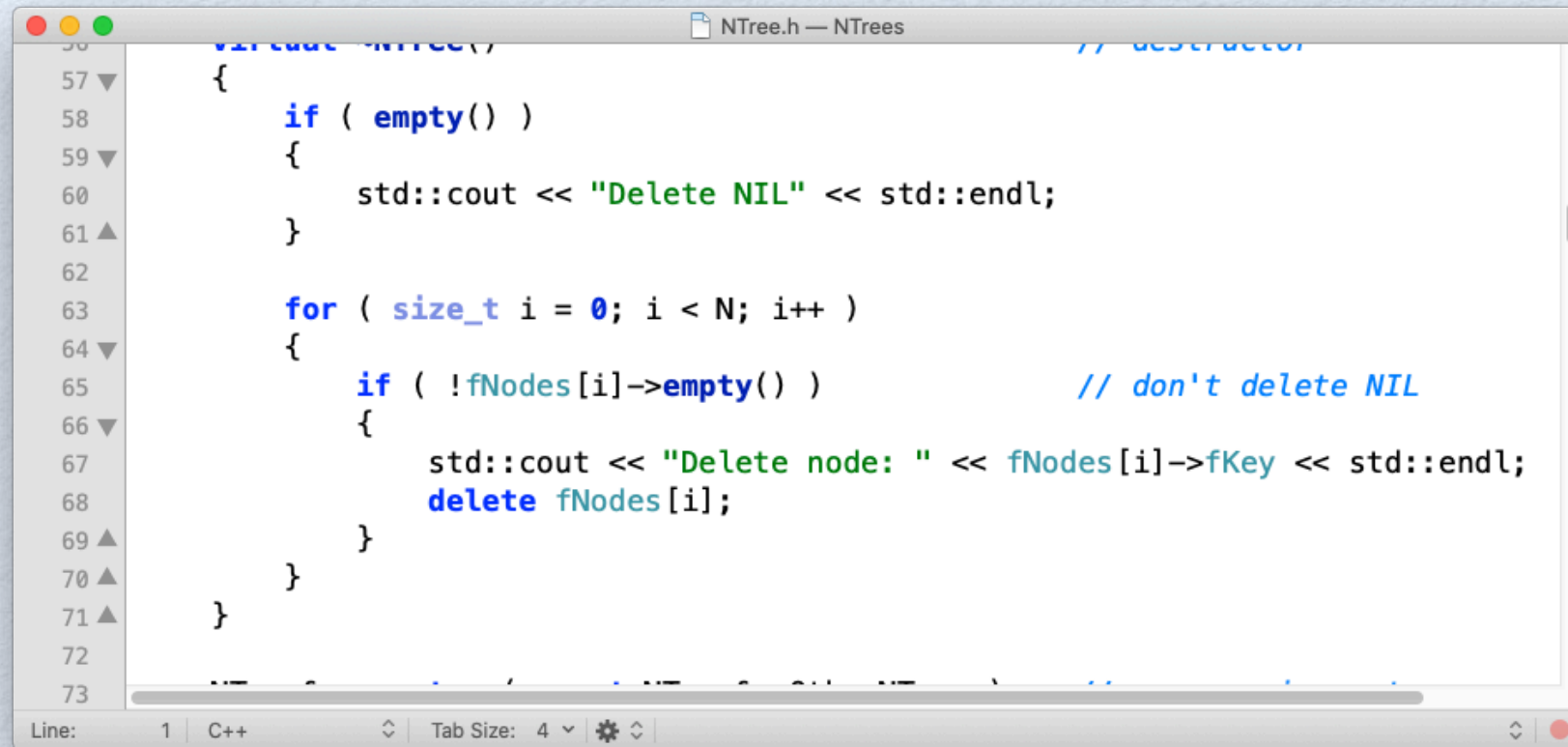


```
31
32 NTree( const T& aKey ) : fKey(aKey)           // NTree leaf
33 {
34     initLeaves();
35 }
36
37 NTree( T&& aKey ) : fKey(std::move(aKey))      // NTree leaf
38 {
39     initLeaves();
40 }
41
```

The screenshot shows a code editor window titled "NTree.h — NTrees". The code defines two public constructors for the NTree class. The first constructor takes a const reference to T (aKey) and initializes fKey with aKey, followed by a call to initLeaves(). The second constructor takes a reference to T (aKey) and initializes fKey with std::move(aKey), followed by a call to initLeaves(). The code is color-coded: NTree is blue, const is blue, T& is black, aKey is black, fKey is cyan, std::move is black, and // NTree leaf is blue. The editor has a line number column on the left (31-41) and a status bar at the bottom showing "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().

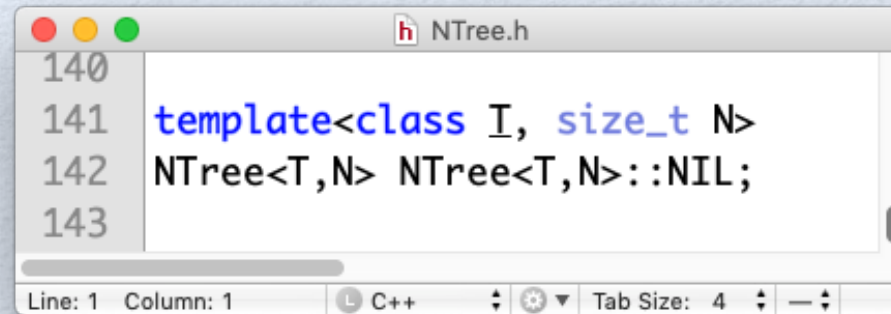
The NTree<T,N> Destructor



```
56 virtual ~NTree() // destructor
57 {
58     if ( empty() )
59     {
60         std::cout << "Delete NIL" << std::endl;
61     }
62
63     for ( size_t i = 0; i < N; i++ )
64     {
65         if ( !fNodes[i]->empty() ) // don't delete NIL
66         {
67             std::cout << "Delete node: " << fNodes[i]->fKey << std::endl;
68             delete fNodes[i];
69         }
70     }
71 }
72
73
```

- 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

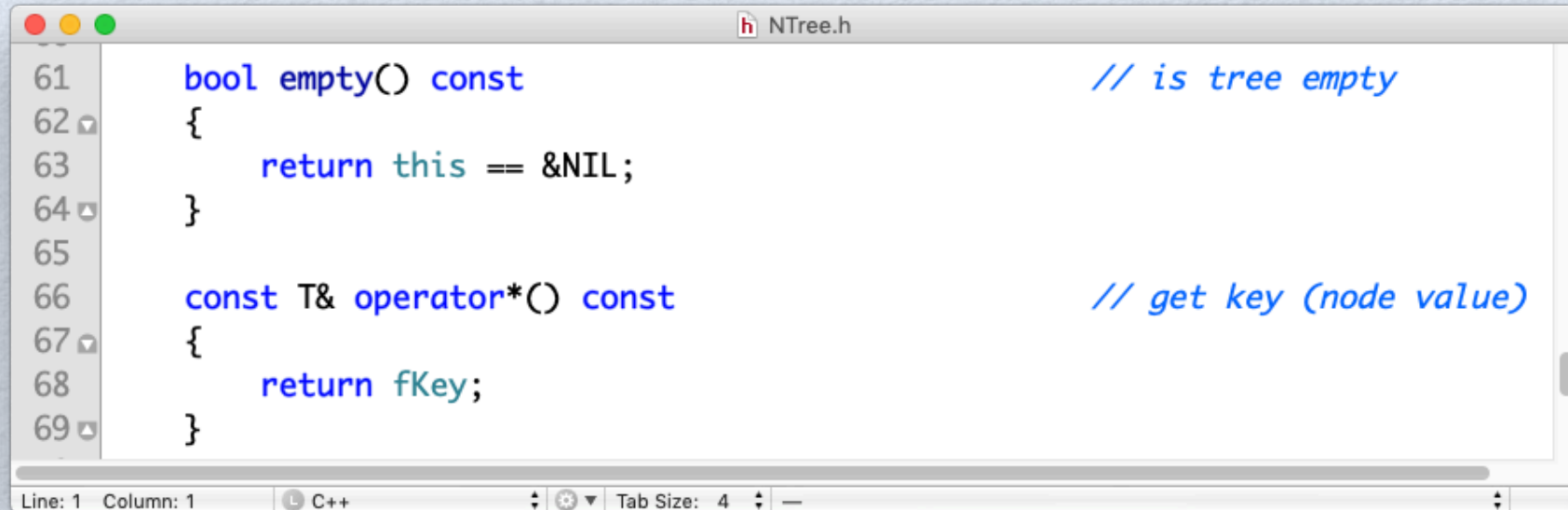


```
140
141 template<class T, size_t N>
142 NTree<T,N> NTree<T,N>::NIL;
143
```

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

- 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.

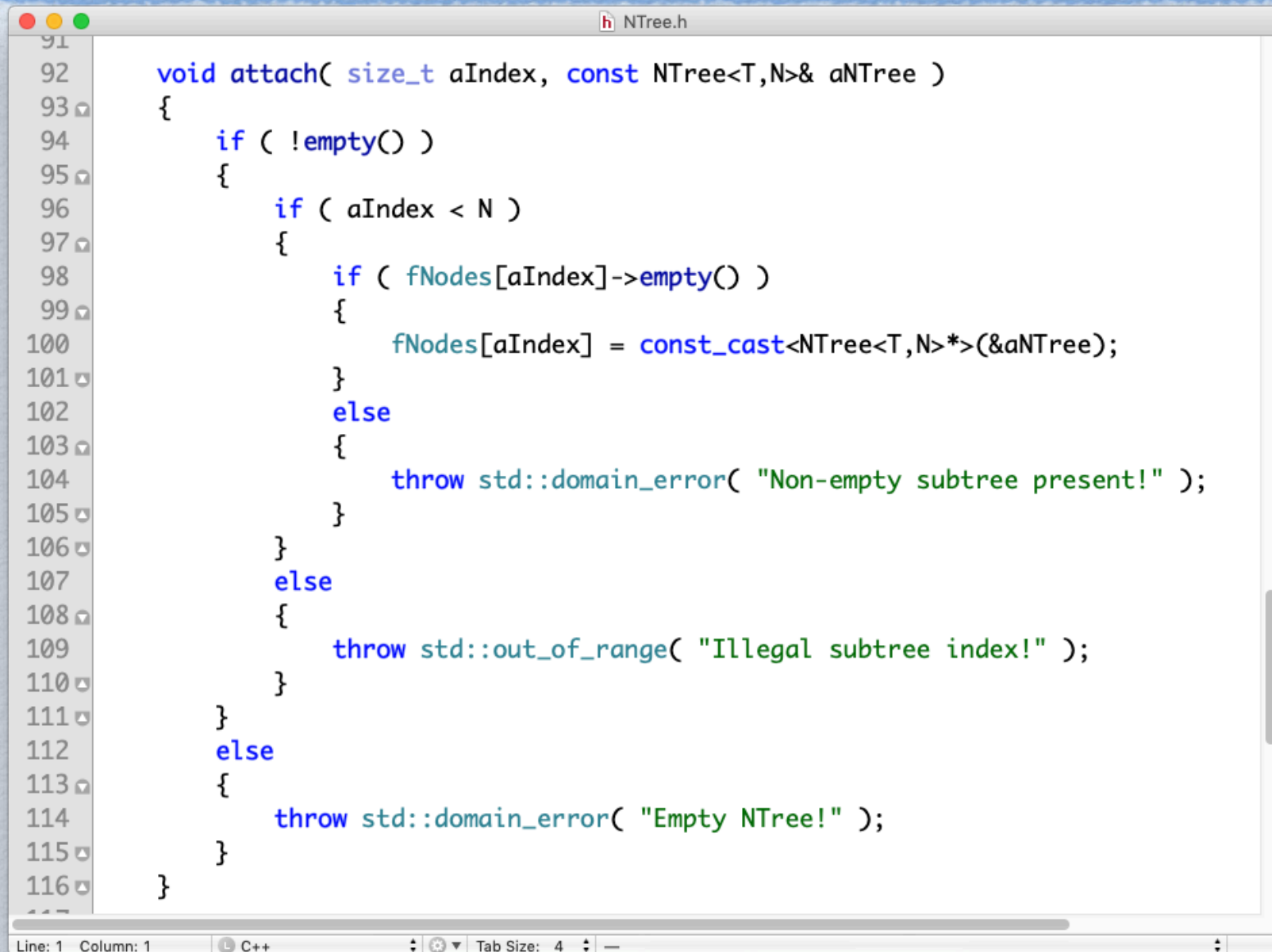
The NTree<T,N> Auxiliaries

A screenshot of a code editor window titled 'NTree.h'. The editor shows two functions: 'empty()' and 'operator*()'. The 'empty()' function is on lines 61-64, and the 'operator*()' function is on lines 66-69. Both functions are marked as 'const'. The 'empty()' function returns 'this == &NIL;'. The 'operator*()' function returns 'fKey;'. The editor has a line number column on the left and a status bar at the bottom showing 'Line: 1 Column: 1' and 'C++'.

```
61     bool empty() const                                // is tree empty
62     {
63         return this == &NIL;
64     }
65
66     const T& operator*() const                          // get key (node value)
67     {
68         return fKey;
69     }
```

- 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.)

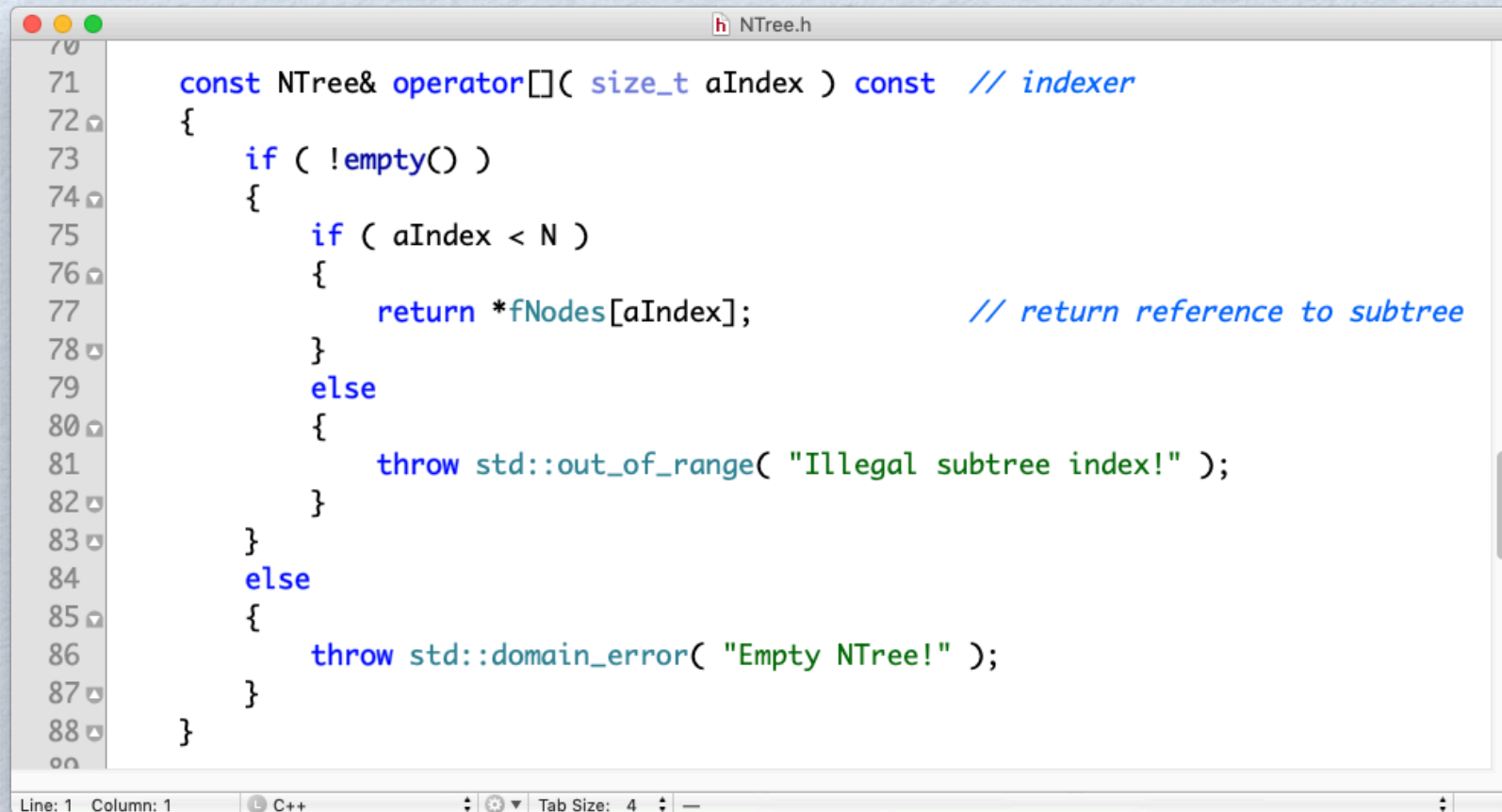
Attaching a New Subtree



```
91
92 void attach( size_t aIndex, const NTree<T,N>& aNTree )
93 {
94     if ( !empty() )
95     {
96         if ( aIndex < N )
97         {
98             if ( fNodes[aIndex]->empty() )
99             {
100                 fNodes[aIndex] = const_cast<NTree<T,N>*>(&aNTree);
101             }
102             else
103             {
104                 throw std::domain_error( "Non-empty subtree present!" );
105             }
106         }
107         else
108         {
109             throw std::out_of_range( "Illegal subtree index!" );
110         }
111     }
112     else
113     {
114         throw std::domain_error( "Empty NTree!" );
115     }
116 }
```

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

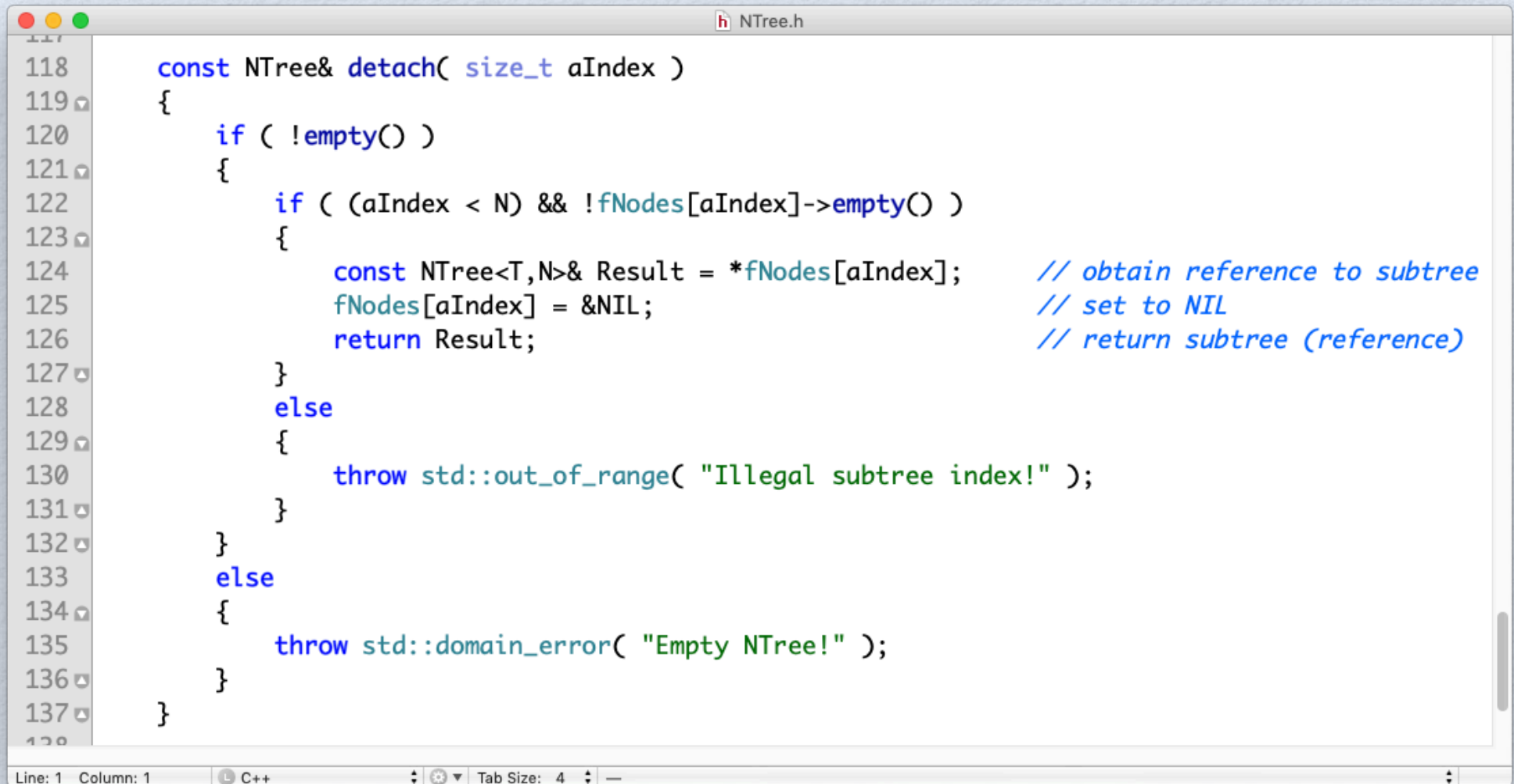
Accessing a Subtree



```
70
71  const NTree& operator[]( size_t aIndex ) const  // indexer
72  {
73      if ( !empty() )
74      {
75          if ( aIndex < N )
76          {
77              return *fNodes[aIndex];           // return reference to subtree
78          }
79          else
80          {
81              throw std::out_of_range( "Illegal subtree index!" );
82          }
83      }
84      else
85      {
86          throw std::domain_error( "Empty NTree!" );
87      }
88  }
```

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

Removing a Subtree

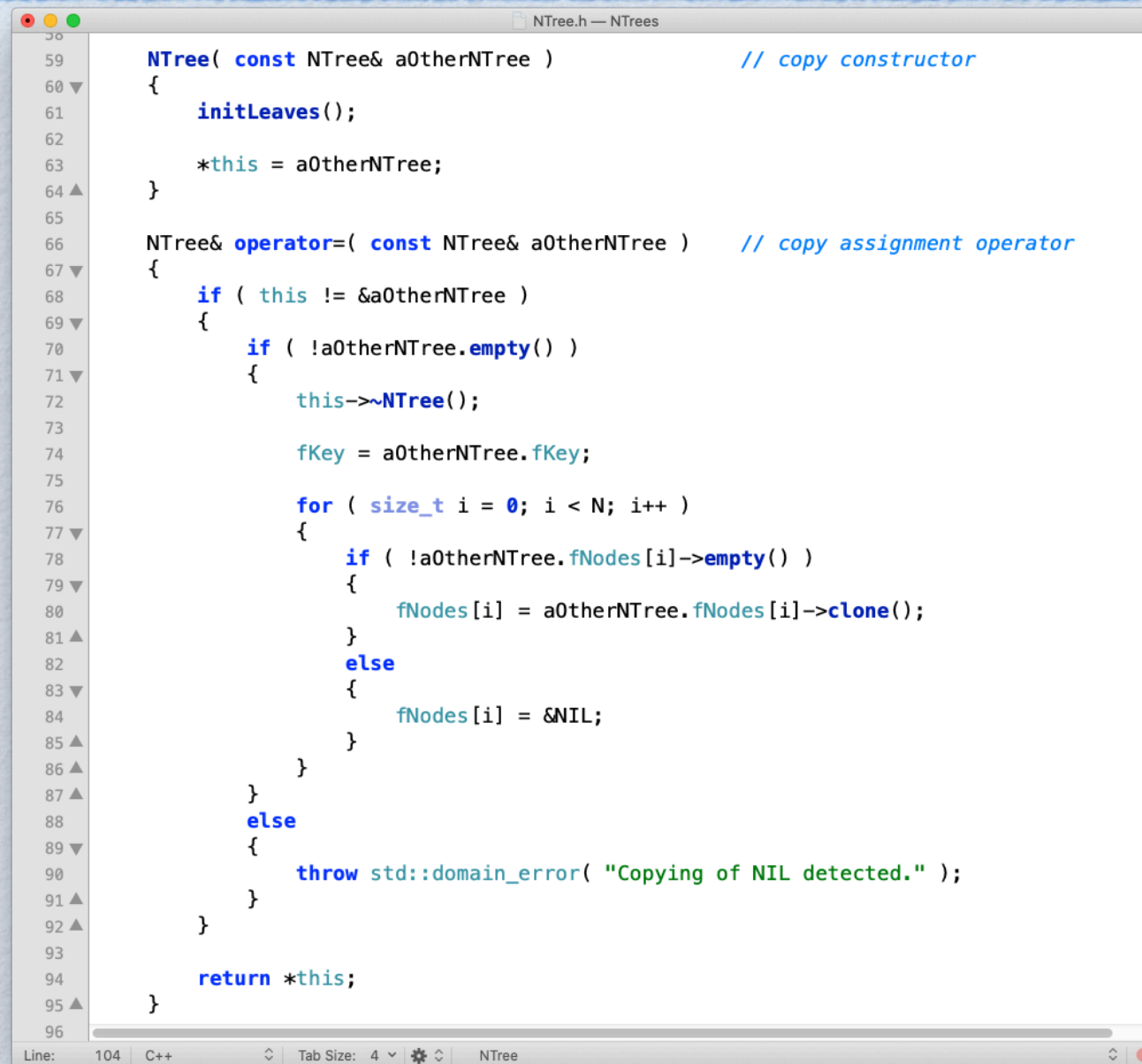


The image shows a code editor window titled "NTree.h" with a line number margin on the left. The code defines a `detach` function for an `NTree` class. The function takes a `size_t aIndex` and returns a `const NTree&`. It checks if the tree is empty. If not, it checks if the index is within bounds and if the node at that index is empty. If the node is empty, it returns the subtree reference. If the node is not empty, it sets the node to `&NIL` and returns the subtree reference. If the index is out of range, it throws a `std::out_of_range` exception. If the tree is empty, it throws a `std::domain_error` exception.

```
118 const NTree& detach( size_t aIndex )
119 {
120     if ( !empty() )
121     {
122         if ( (aIndex < N) && !fNodes[aIndex]->empty() )
123         {
124             const NTree<T,N>& Result = *fNodes[aIndex]; // obtain reference to subtree
125             fNodes[aIndex] = &NIL; // set to NIL
126             return Result; // return subtree (reference)
127         }
128         else
129         {
130             throw std::out_of_range( "Illegal subtree index!" );
131         }
132     }
133     else
134     {
135         throw std::domain_error( "Empty NTree!" );
136     }
137 }
```

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

Copy Semantics



```
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." );
        }
    }

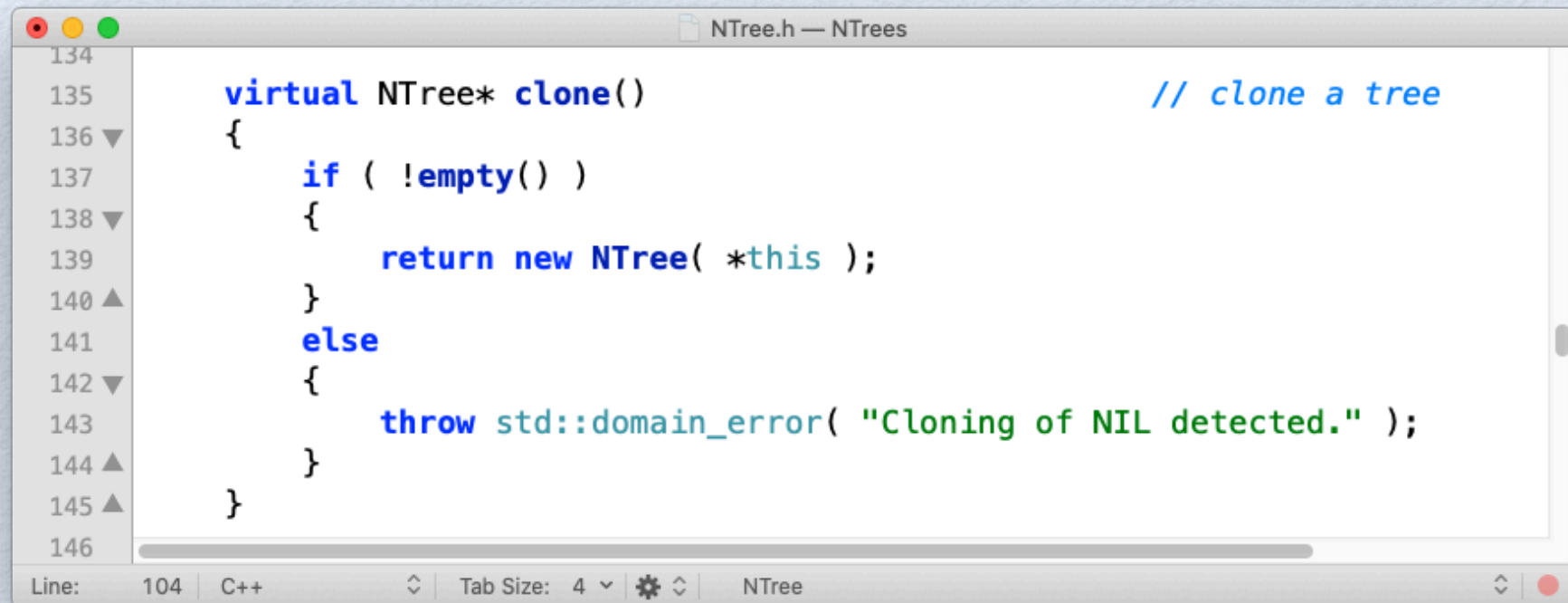
    return *this;
}
```


Move Semantics

```
NTree.h — NTrees
97 NTree( NTree&& a0therNTree ) // move constructor
98 {
99     initLeaves();
100
101     *this = std::move(a0therNTree);
102 }
103
104 NTree& operator=( NTree&& a0therNTree ) // move assignment operator
105 {
106     if ( this != &a0therNTree )
107     {
108         if ( !a0therNTree.empty() )
109         {
110             this->~NTree();
111
112             fKey = std::move(a0therNTree.fKey);
113
114             for ( size_t i = 0; i < N; i++ )
115             {
116                 if ( !a0therNTree.fNodes[i]->empty() )
117                 {
118                     fNodes[i] = const_cast<NTree<T,N>*>(&a0therNTree.detach( i ));
119                 }
120                 else
121                 {
122                     fNodes[i] = &NIL;
123                 }
124             }
125         }
126         else
127         {
128             throw std::domain_error( "Moving of NIL detected." );
129         }
130     }
131
132     return *this;
133 }
```

Steal memory

Clone



The screenshot shows a code editor window titled "NTree.h — NTrees". The editor displays the implementation of the `clone()` method for the `NTree` class. The code is as follows:

```
134 virtual NTree* clone() // clone a tree
135 {
136     if ( !empty() )
137     {
138         return new NTree( *this );
139     }
140     else
141     {
142         throw std::domain_error( "Cloning of NIL detected." );
143     }
144 }
145
146
```

The editor interface includes a line number margin on the left (lines 134 to 146), a status bar at the bottom showing "Line: 104", "C++", "Tab Size: 4", and a search icon. The code uses color-coding: keywords are blue, comments are blue, and error messages are green.

- 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

```
17 void testBasicOperations()
18 {
19     using NS3Tree = NTree<string,3>;
20
21     string s1( "A" );
22     string s2( "B" );
23     string s3( "C" );
24
25     NS3Tree root( "Hello World!" );
26     NS3Tree nodeA( s1 );
27     NS3Tree nodeB( s2 );
28     NS3Tree nodeC( s3 );
29     NS3Tree nodeAB( "AB" );
30
31     root.attach( 0, nodeA );
32     root.attach( 1, nodeB );
33     root.attach( 2, nodeC );
34     const_cast<NS3Tree&>(root[1]).attach( 1, nodeAB );
35
36     cout << "root:      " << *root << endl;
37     cout << "root[0]:    " << *root[0] << endl;
38     cout << "root[1]:    " << *root[1] << endl;
39     cout << "root[2]:    " << *root[2] << endl;
40     cout << "root[1][1]: " << *root[1][1] << endl;
41
42     const_cast<NS3Tree&>(root[1]).detach( 1 );
43     root.detach( 0 );
44     root.detach( 1 );
45     root.detach( 2 );
46 }
```


```
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>

```
8  template<typename T>
9  class BTree
10 {
11 private:
12     T fKey; // T() for empty BTree
13     BTree<T>* fLeft;
14     BTree<T>* fRight;
15
16     BTree(); // sentinel constructor
17
18 public:
19     static BTree<T> NIL; // Empty BTree
20
21     BTree( const T& aKey ); // BTree leaf
22     BTree( T&& aKey ); // BTree leaf
23
24     BTree( const BTree& aOtherBTree ); // copy constructor
25     BTree( BTree&& aOtherBTree ); // move constructor
26
27     virtual ~BTree(); // destructor
28
29     BTree& operator=( const BTree& aOtherBTree ); // copy assignment operator
30     BTree& operator=( BTree&& aOtherBTree ); // move assignment operator
31
32     virtual BTree* clone(); // clone a tree
33
34     bool empty() const; // is tree empty
35     const T& operator*() const; // get key (node value)
36
37     const BTree& left() const;
38     const BTree& right() const;
39
40     // tree manipulators
41     void attachLeft( const BTree<T>& aBTree );
42     void attachRight( const BTree<T>& aBTree );
43     const BTree& detachLeft();
44     const BTree& detachRight();
45 };
```

Line: 2 Column: 11

C++

Tab Size: 4