SWINBURNE UNIVERSITY OF TECHNOLOGY

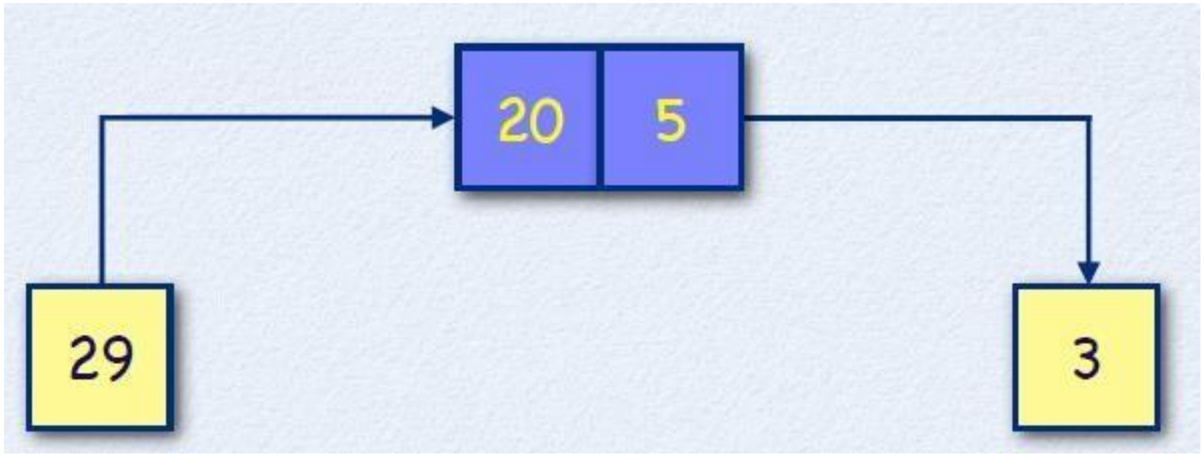
Data Structures and Patterns
Revision 2

SWINBURNE UNIVERSITY OF TECHNOLOGY

Lecture 7

Queue

- Queue is another important basic data structure.
- A queue is a special version of a linear sequence where access to element is only possible at its front and end.



Queue Behaviour

- Queues manage elements in first-in, first-out (FIFO) manner, just like a real queue.
- A queue underflow happens when one tries to dequeue on an empty queue.
- A queue overflow happens when one tries to enqueue on a full queue.

Requirements for a Priority Queue

- The underlying data structure for a priority queue must be sorted.
- Elements are queued using an integer to specify priority. We can use a pair<Key, T> to store elements with their associated priority.
- We need to provide a matching operator < on key values to sort elements in the priority queue.

Static variables

- C++ allows for two forms of global variables:
  - Static non-class variables,
  - Static class variables.
- Static variables are mapped to the global memory. Access to them depends on the visibility specified.
- Generally, local variables and function arguments are stored on the stack, while global and static variables are stored on the heap.
  - Static variables get created only once no matter how many times the function is called or how many class instances are created.

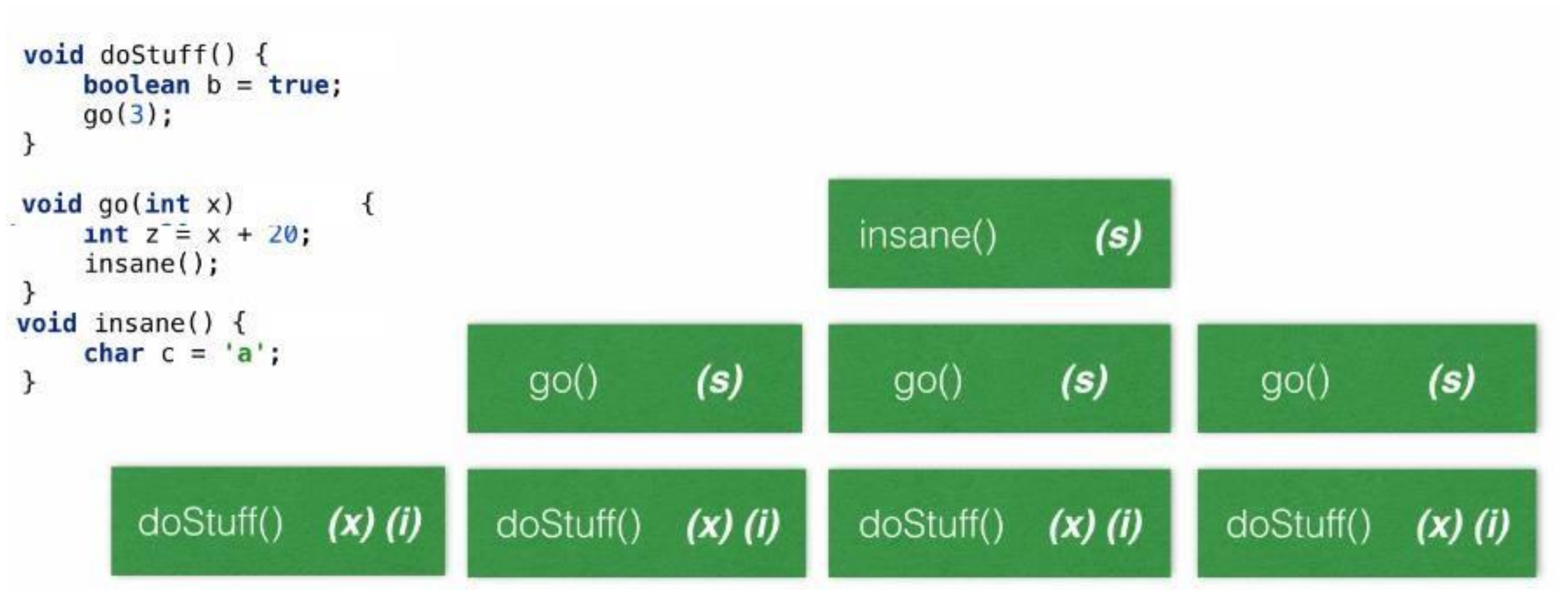
The Keyword static

- The keyword static can be used to
  - mark the linkage of a variable or function internal,
  - retain the value of a local variable between function calls,
  - declare a class instance variable,
  - define a class method.

Program Memory: Stack

- All value-based objects are stored in the program's stack.
- The program stack is automatically allocated and freed.
- References to stack locations are only valid when passed to a callee (a function called by another).
- References to stack locations cannot be returned from a function.

Example of Stack



```
void doStuff() {
    boolean b = true;
    go(3);
}

void go(int x)       {
    int z = x + 20;
    insane();
}
void insane() {
    char c = 'a';
}
```

More readings on stack:
http://cryptroix.com/2016/10/16/journey-to-the-stack/

Program Memory: Heap

- Every program maintains a heap for dynamically allocated objects.
- Each heap object is accessed through a pointer.
- Heap objects are not automatically freed when pointer variables become inaccessible (i.e., go out of scope).
- Memory management becomes essential in C++ to reclaim memory and to prevent the occurrences of so-called memory leaks.
  - a memory leak is a type of resource leak that occurs when a computer program incorrectly manages memory allocations in such a way that memory which is no longer needed is not released

The Copy Constructor

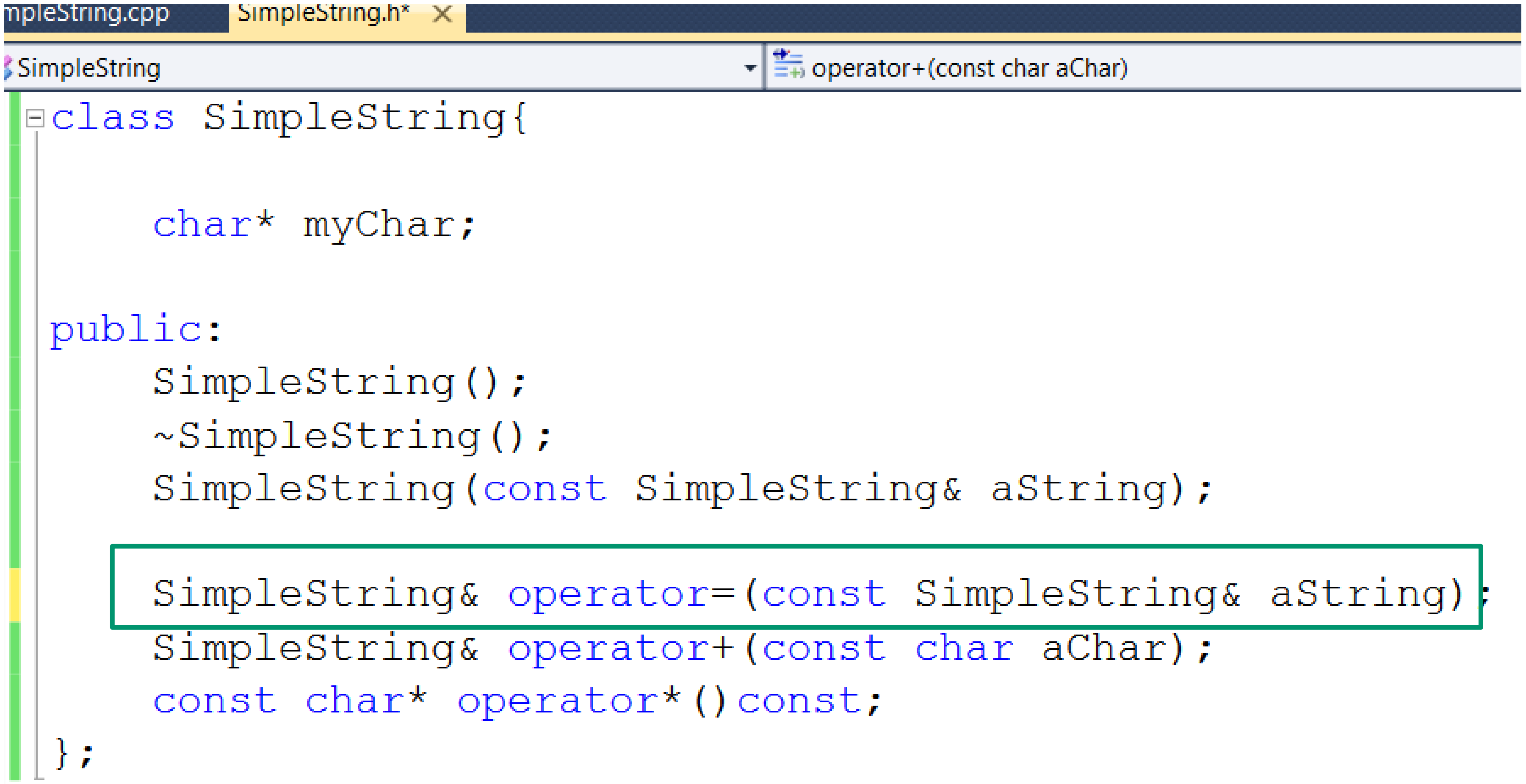
- The **copy constructor** is a **constructor** which creates an object by initializing it with an object of the same class, which has been created previously. The **copy constructor** is used to: Initialize one object from another of the same type.
- Whenever one defines a new type, one needs to specify implicitly or explicitly what has to happen when objects of that that type are copied, assigned, and destroyed.
  - The copy constructor is a special member, taking just a single parameter that is a const reference to an object of the class itself.

Rule Of Thumb

- Copy control in C++ requires three elements:
  - a copy constructor
  - an assignment operator (=)
  - a destructor
- Whenever one defines a copy constructor, one must also define an assignment operator and a destructor.

What if I want to use "="

- Overload the operator "="



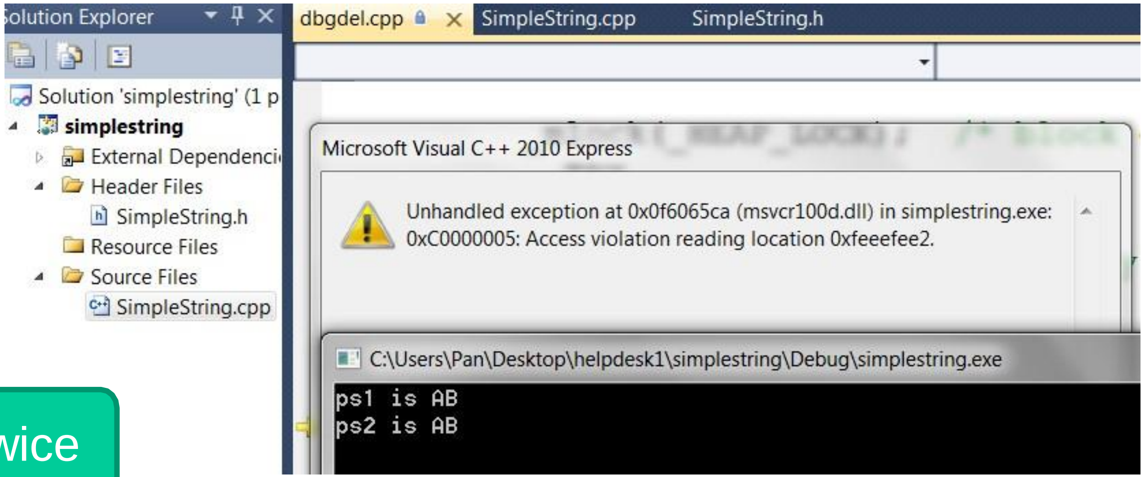
```
class SimpleString{

    char* myChar;

public:
    SimpleString();
    ~SimpleString();
    SimpleString(const SimpleString& aString);

    SimpleString& operator=(const SimpleString& aString);
    SimpleString& operator+(const char aChar);
    const char* operator*()const;
};
```

Copying Pointers

```
int main(){


    SimpleString* ps1= new SimpleString();
    *ps1+'A';

    SimpleString* ps2=ps1;
    *ps2+'B';


    cout<<"ps1 is "<<**ps1<<endl;
    cout<<"ps2 is "<<**ps2<<endl;

    delete ps1;
    delete ps2;
```

Shallow Copy

Free twice



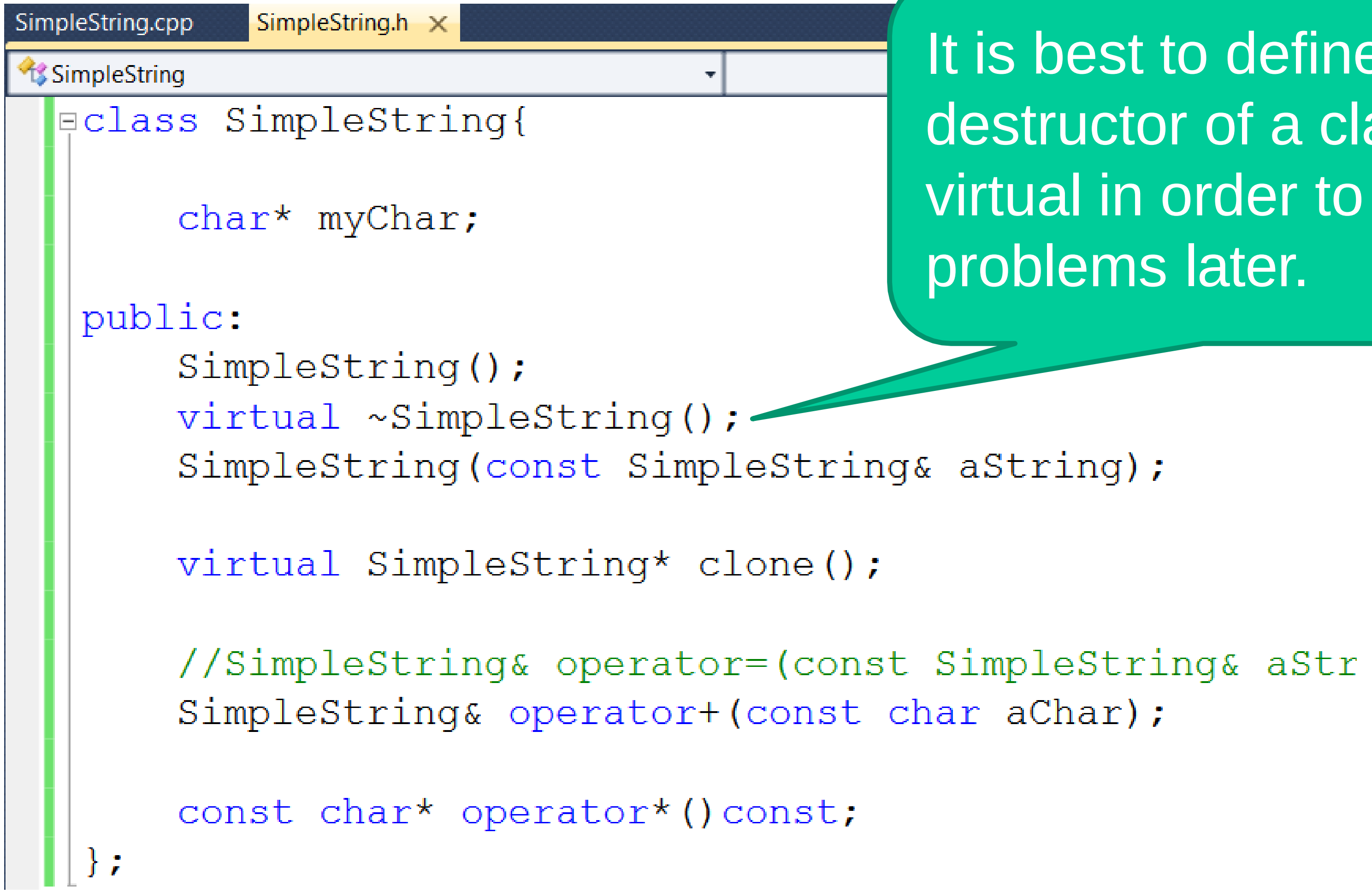
Microsoft Visual C++ 2010 Express

Unhandled exception at 0x0f6065ca (msvcr100d.dll) in simplestring.exe: 0xC0000005: Access violation reading location 0xfeeefee2.

C:\Users\Pan\Desktop\helpdesk1\simplestring\Debug\simplestring.exe

```
ps1 is AB
ps2 is AB
```

Solution: A clone() function



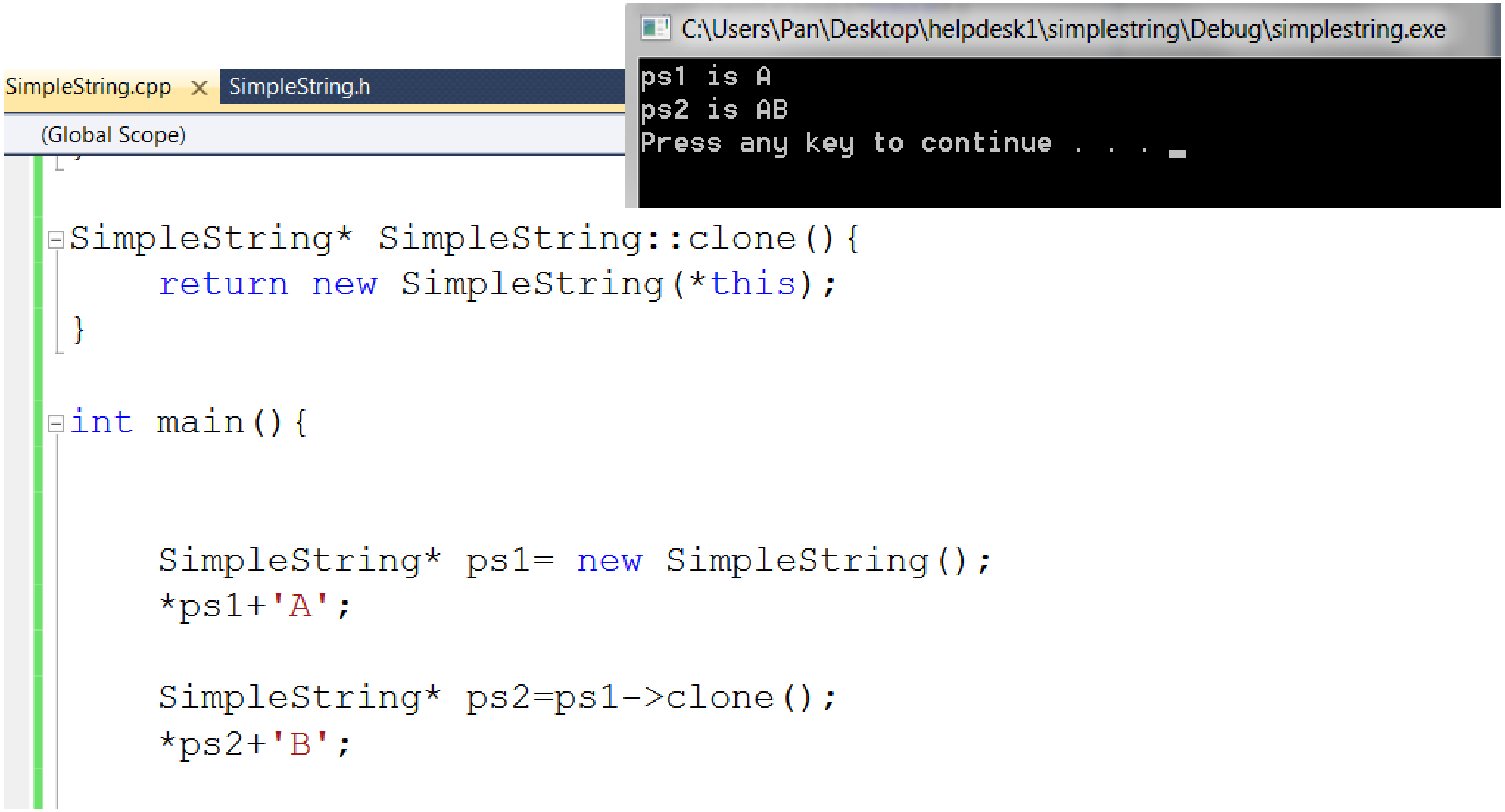
SimpleString.cpp | SimpleString.h ✕

SimpleString

```
class SimpleString{

    char* myChar;

public:
    SimpleString();
    virtual ~SimpleString();
    SimpleString(const SimpleString& aString);

    virtual SimpleString* clone();

    //SimpleString& operator=(const SimpleString& aStr
    SimpleString& operator+(const char aChar);

    const char* operator*()const;
};
```

It is best to define the destructor of a class as virtual in order to avoid problems later.

The Use of clone()



C:\Users\Pan\Desktop\helpdesk1\simplestring\Debug\simplestring.exe

```
ps1 is A
ps2 is AB
Press any key to continue . . .
```

```
SimpleString* SimpleString::clone(){
    return new SimpleString(*this);
}

int main(){


    SimpleString* ps1= new SimpleString();
    *ps1+'A';

    SimpleString* ps2=ps1->clone();
    *ps2+'B';
```

Note with clone()

- if a class has *any* virtual function, it should have a virtual destructor,
- The member function clone() must be defined virtual to allow for proper redefinition in subtypes.
- The member function clone() can only return one type. When a subtype redefines clone(), only the super type can be returned

Reference Counting

- A simple technique to record the number of active uses of an object is **reference counting**.
- Each time a heap-based object is assigned to a variable, the object’s reference count is incremented and the reference count of what the variable previously pointed to is decremented.
- Some compilers emit the necessary code, but in case of C++ reference counting must be defined (semi-)manually.

Smart Pointers in C++

- There are several smart pointers available:
  - std::unique_ptr
  - std::shared_ptr
  - std::weak_ptr
- Smart pointers are used to guarantee automatic deletion of heap-allocated class instances.
- With the unique and shared smart pointers, you do not have to call delete yourself for an object, as it will be done automatically.

Smart Pointers in C++

- std::unique_ptr
  - The unique pointer is a scoped pointer. When this pointer goes out of scope, it will get destroyed and call delete on the object. It cannot be copied.
- std::shared_ptr
  - The shared pointer keeps track of how many references you have to a pointer. Reference count will increase each time the pointer is shared. The object will get deleted when the reference count gets to zero.
- std::weak_ptr
  - The weak pointer will not increase the reference count. It is used only for storing a reference to an object to check if it is still valid or has expired.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Lecture 8

Basics

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

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

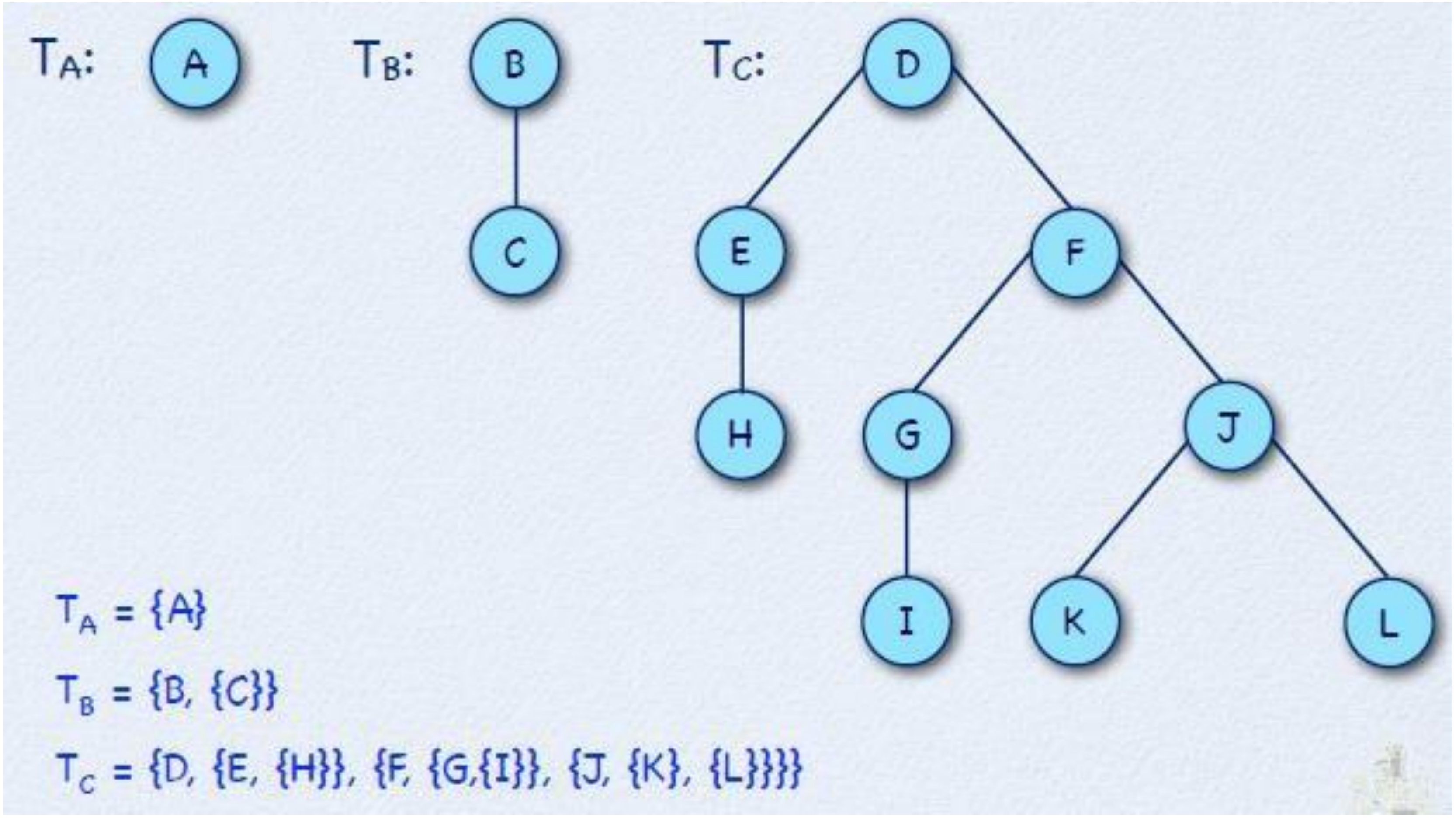
- with the following properties:
  - □A designated node of the singleton, r, is called the root of the tree.
  - □The remaining nodes are partitioned into $n \geq 0$ subsets T1, T2, …, Tn, each of which is a tree.

Definition

- Tree
  - undirected graph
  - any two nodes are connected by exactly one edge
  - no loop

Why are trees in computer science generally drawn upside down?
https://www.quora.com/Why-are-trees-in-computer-science-generally-drawn-upside-down-from-how-trees-are-in-real-life

Tree Examples



T_A = {A}

T_B = {B, {C}}

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

Structure of a Tree

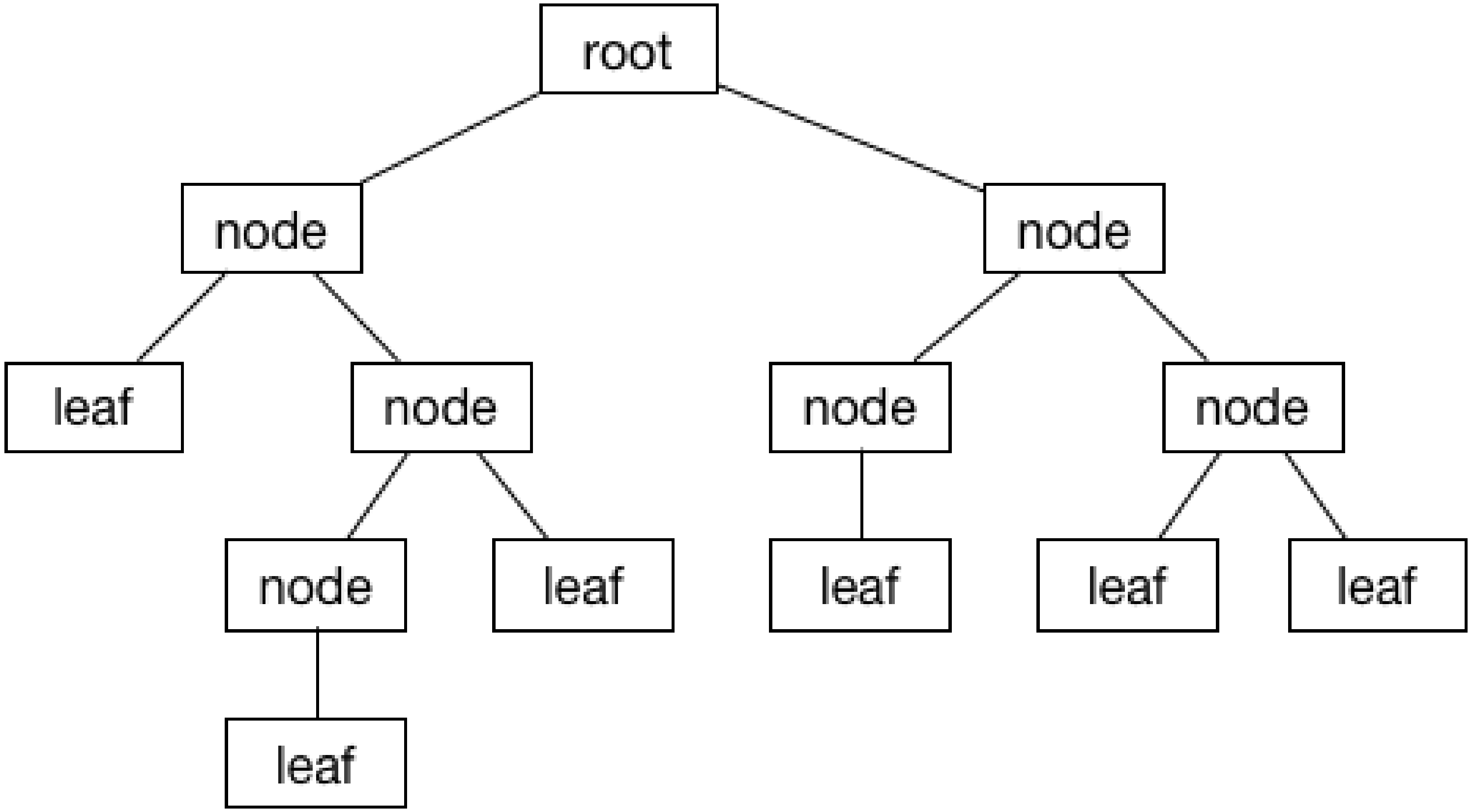


Figure from: https://www.raywenderlich.com/990-swift-algorithm-club-swift-binary-search-tree-data-structure

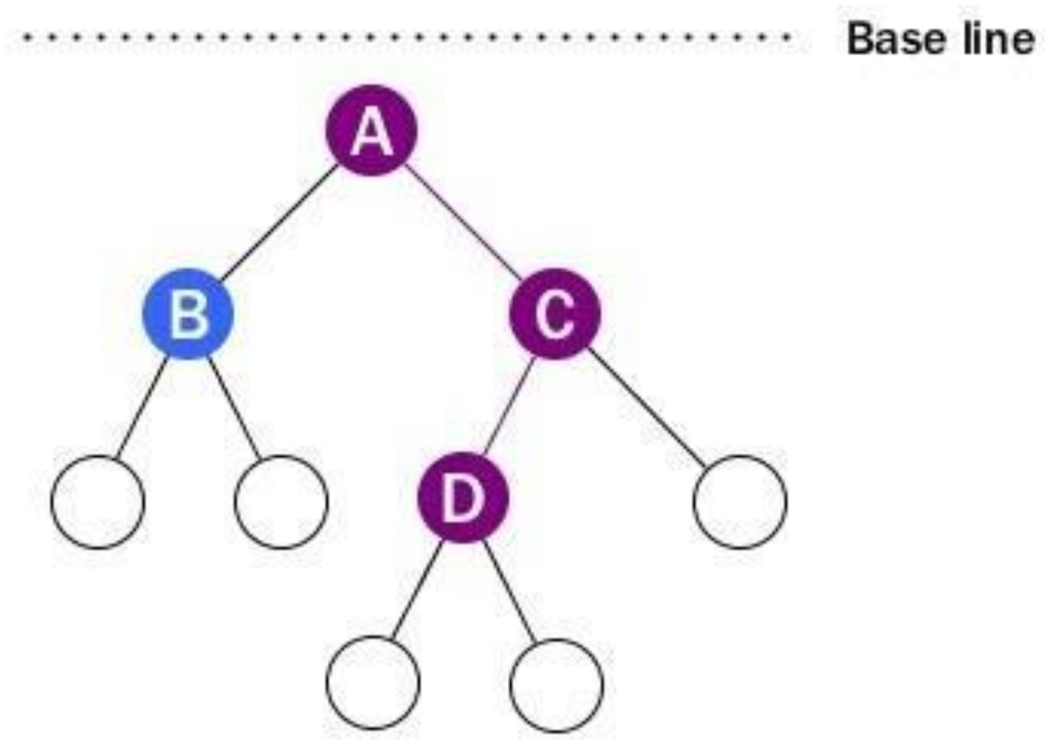
Depth, Height and Level

- The **depth** of a node is the number of edges from the node to the **tree's** root node.
  - A root node will have a **depth** of 0.
- The **height** of a node is the number of edges on the longest path from the node to a leaf.
  - *Height of tree –The height of a tree is the number of edges on the longest downward path between the root and a leaf.*
  - A leaf node will have a **height** of 0.
- The **level** of a node is defined by 1 + the number of connections between the node and the root.

i.e. level = depth+1

Example

- The height of the tree is 3
- The height of Node C is 2 (count from bottom)
- The depth of Node D is 2 (count from top)



Nodes With the Same Degree

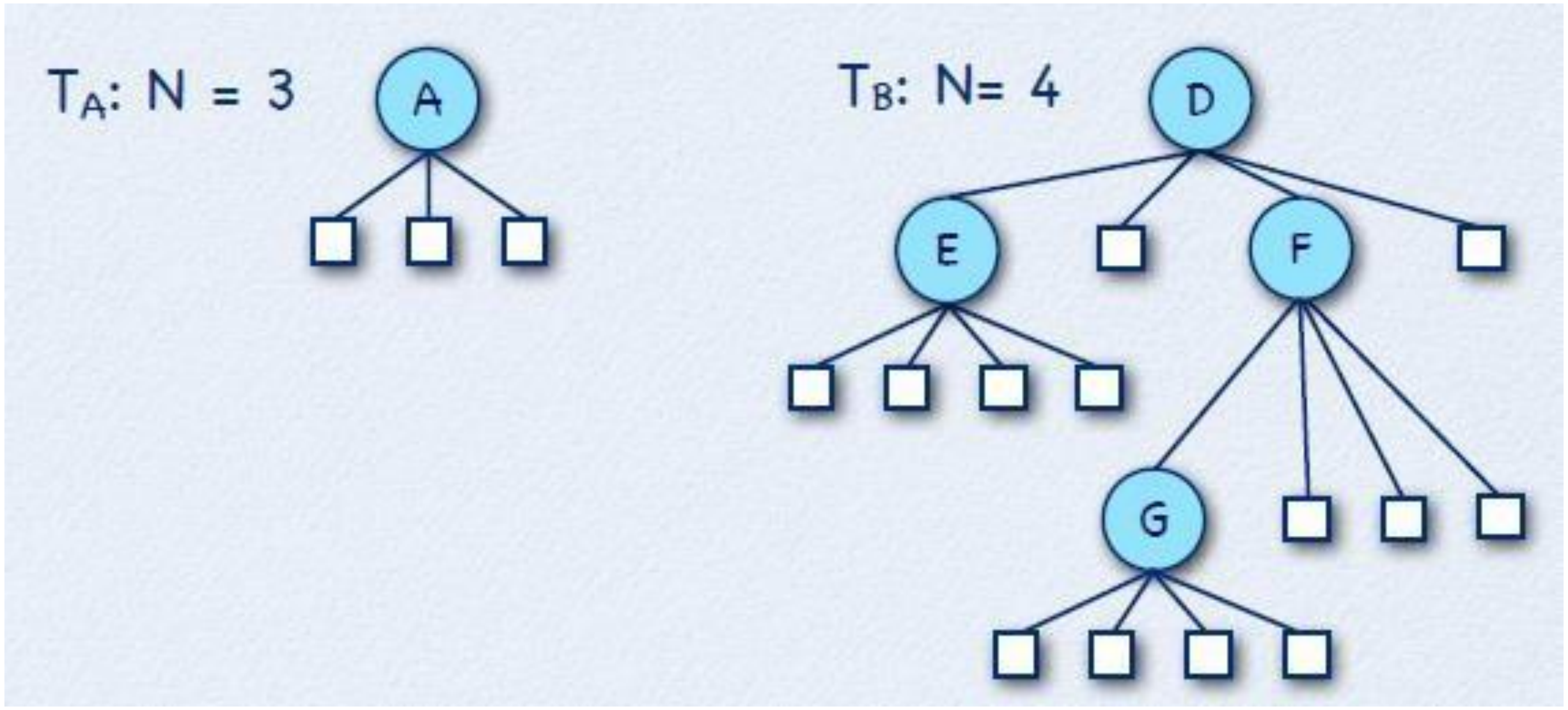
- The general case allows each node in a tree to have a different degree (The number of subtrees of a node is called the **degree** of the node). We now consider a variant of trees in which each node has the same degree.
- Unfortunately, it is not possible to construct a tree that has a finite number of nodes, which all have the same degree N, except the trivial case N = 0.
- We need a special notion, called empty tree, to realize trees in which all nodes have the same degree.

N-ary tree

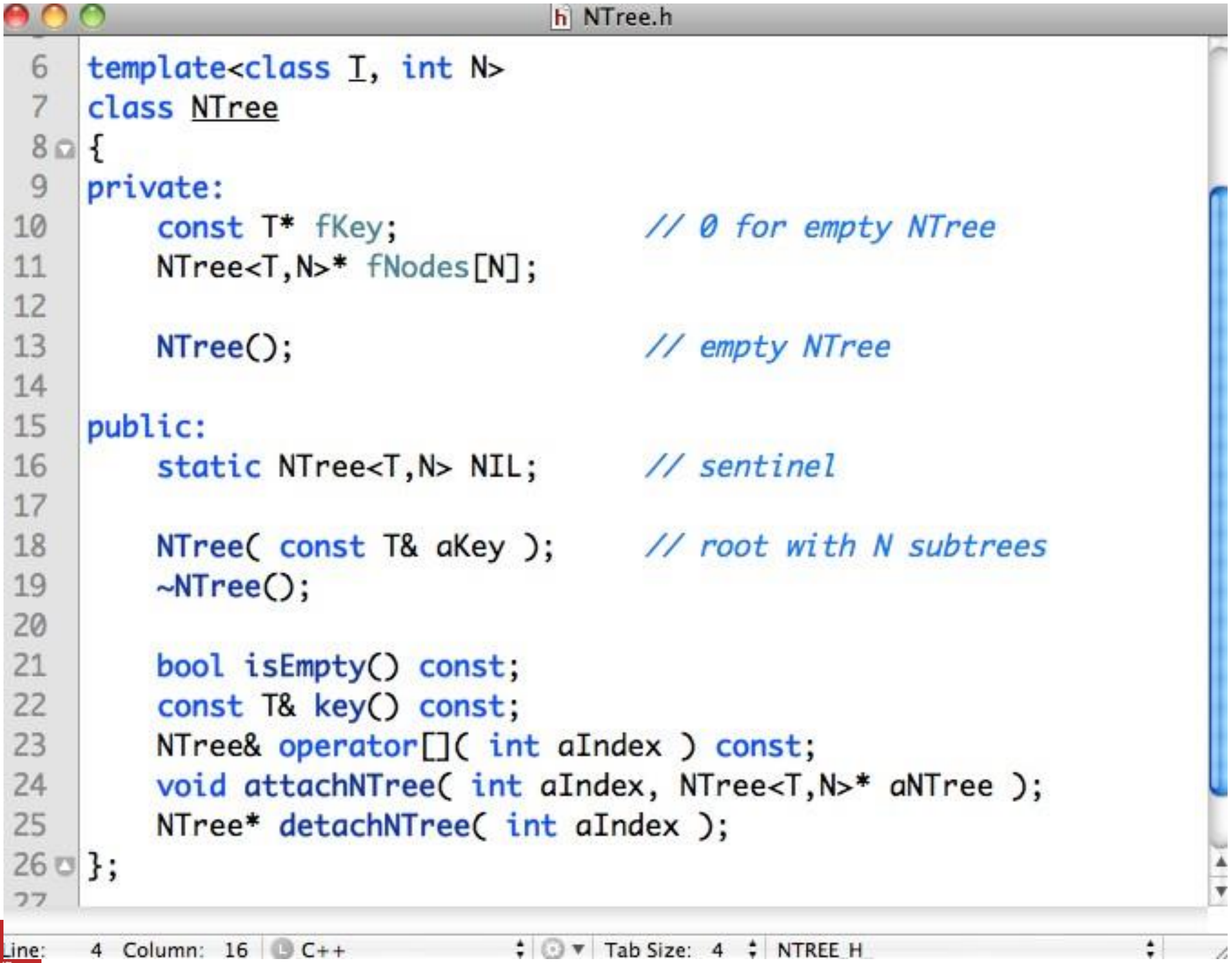
- An N-ary tree T, N ≥ 1, is a finite set of nodes with one of the following properties:
  - Either the set is empty, T = ∅, or
  - The set consists of a root, R, and exactly N distinct
- N-ary trees, That is, the remaining nodes are partitioned into N ≥ 1 subsets, T1, T2, …, TN, each of which is an N-ary tree such that

  T = {R, T1, T2, …, TN}.

N-ary Tree Examples



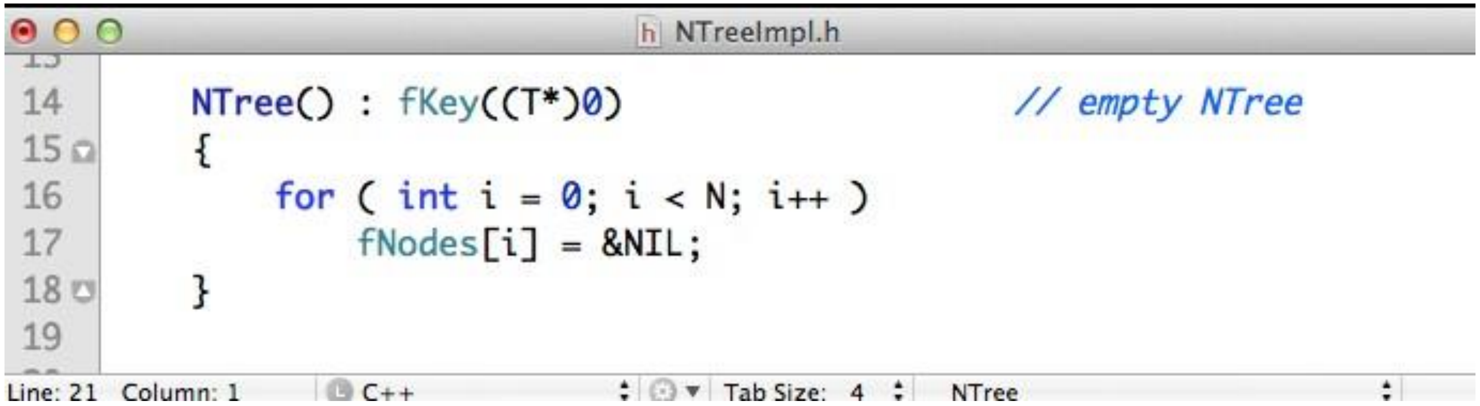
Ntree Class


NTree.h

```
template<class T, int N>
class NTree
{
private:
    const T* fKey;                  // 0 for empty NTree
    NTree<T,N>* fNodes[N];

    NTree();                        // empty NTree

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

    NTree( const T& aKey );         // root with N subtrees
    ~NTree();

    bool isEmpty() const;
    const T& key() const;
    NTree& operator[]( int aIndex ) const;
    void attachNTree( int aIndex, NTree<T,N>* aNTree );
    NTree* detachNTree( int aIndex );
};
```

Line: 4 Column: 16 C++ Tab Size: 4 NTREE_H

The Private Empty NTree Constructor

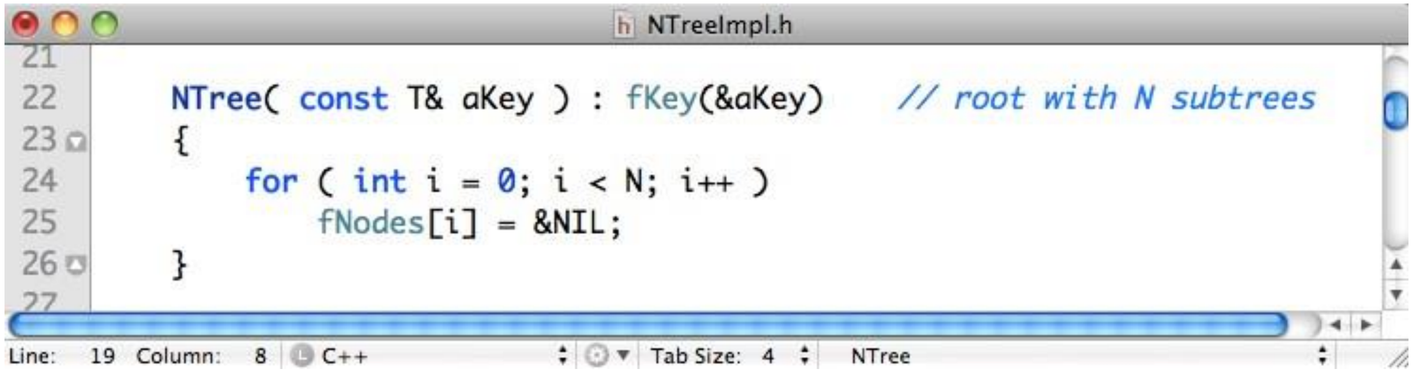

NTreeImpl.h

```
    NTree() : fKey((T*)0)                       // empty NTree
    {
        for ( int i = 0; i < N; i++ )
            fNodes[i] = &NIL;
    }
```

Line: 21   Column: 1   C++   Tab Size: 4   NTree

- We use (T*)0), the null pointer, to initialize the fKey with a suitable value of type pointer to T.
- Each subtree-node is set to the location of NIL, the sentinel node for NTree.
- This constructor is only used to set up the sentinel for NTree.

The Public NTree Constructor



NTreeImpl.h

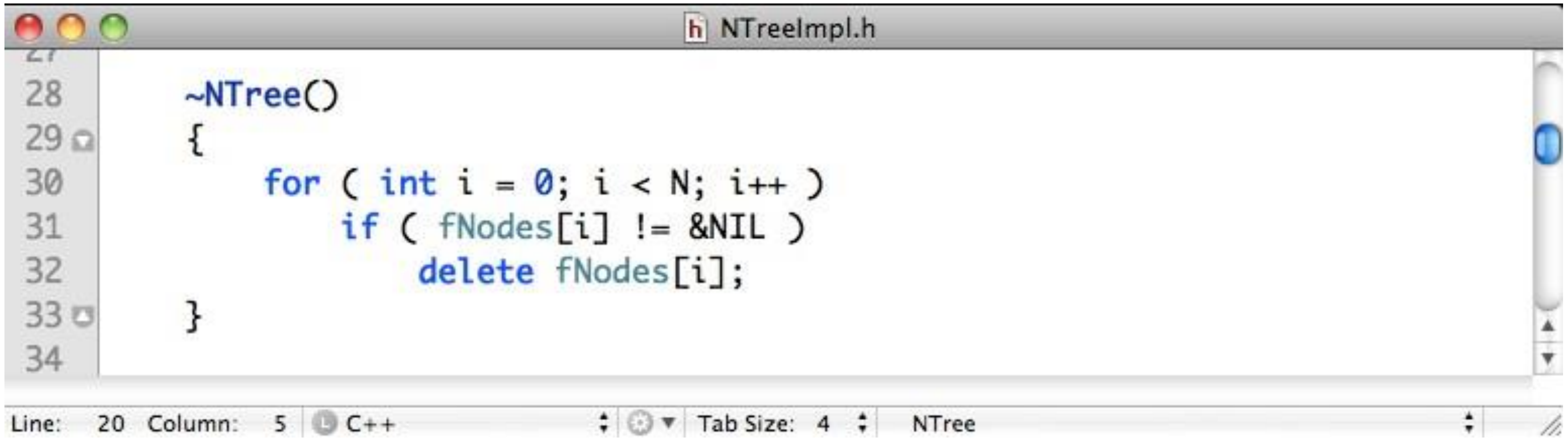
```
    NTree( const T& aKey ) : fKey(&aKey)    // root with N subtrees
    {
        for ( int i = 0; i < N; i++ )
            fNodes[i] = &NIL;
    }
```

Line: 19 Column: 8 C++ Tab Size: 4 NTree

- We store the address of the reference aKey in fKey.
- Each child node in a non-empty NTree leaf node is set to the location of NIL, the sentinel node for NTree.

In computer programming, a **sentinel node** is a specifically designated **node** used with linked lists and **trees** as a traversal path terminator. This type of **node** does not hold or reference any data managed by the data structure.

The NTree Destructor

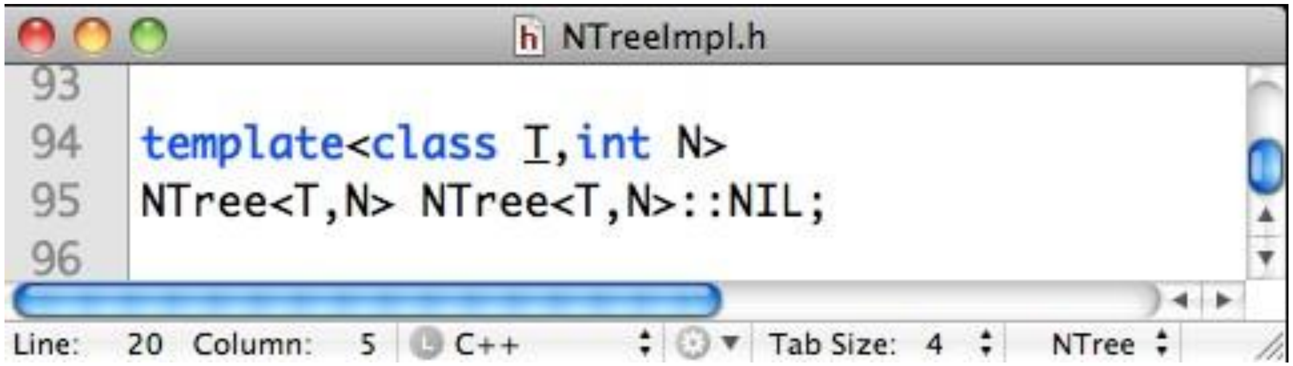

NTreeImpl.h

```
    ~NTree()
    {
        for ( int i = 0; i < N; i++ )
            if ( fNodes[i] != &NIL )
                delete fNodes[i];
    }
```

Line: 20 Column: 5 C++ Tab Size: 4 NTree

- In the destructor of NTree only non-sentinel nodes are destroyed.

The NTree Sentinel

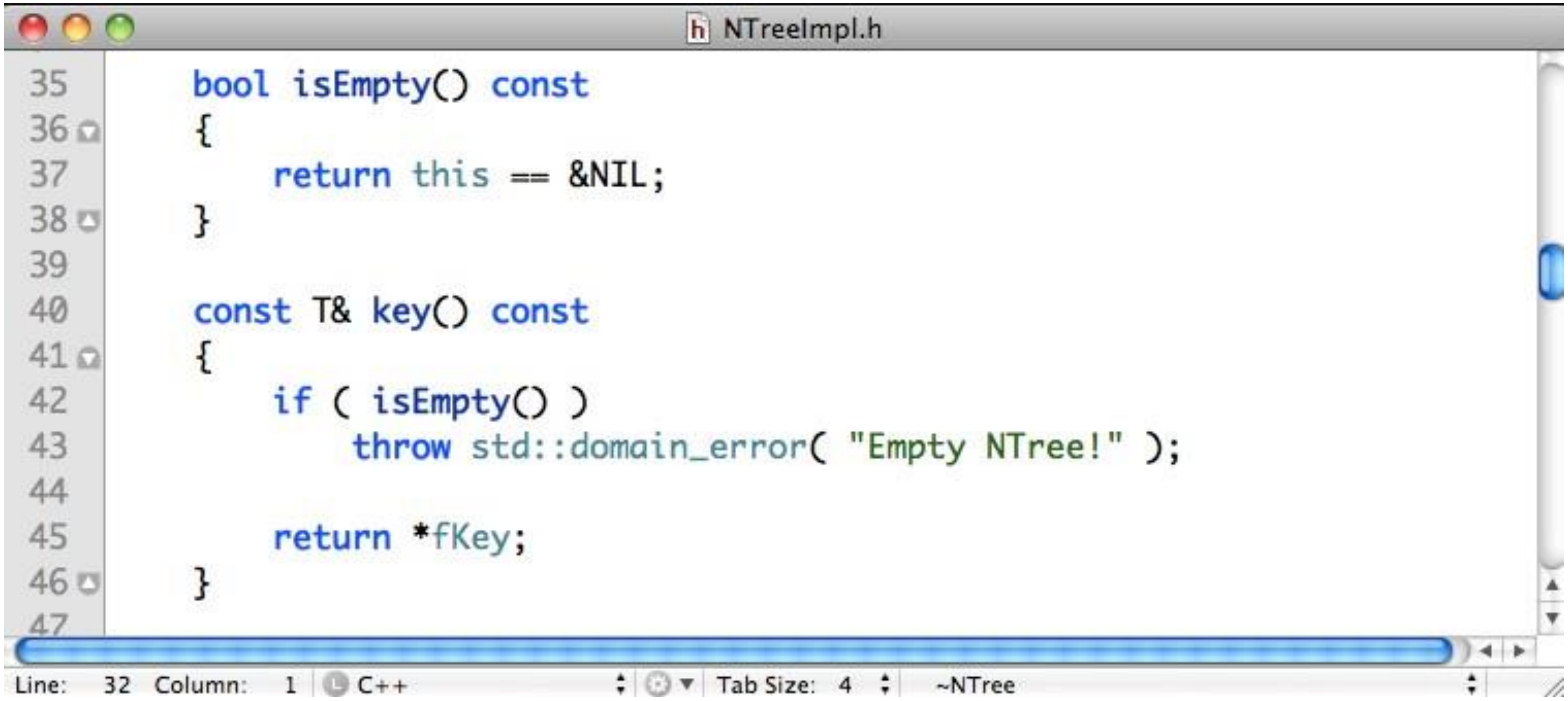

NTreeImpl.h

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

Line: 20 Column: 5 C++ Tab Size: 4 NTree

- Static instance variables, like the NTree 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. It means that all members of Ntree are available, including the private constructor to initialize NIL.

The NTree Auxiliaries

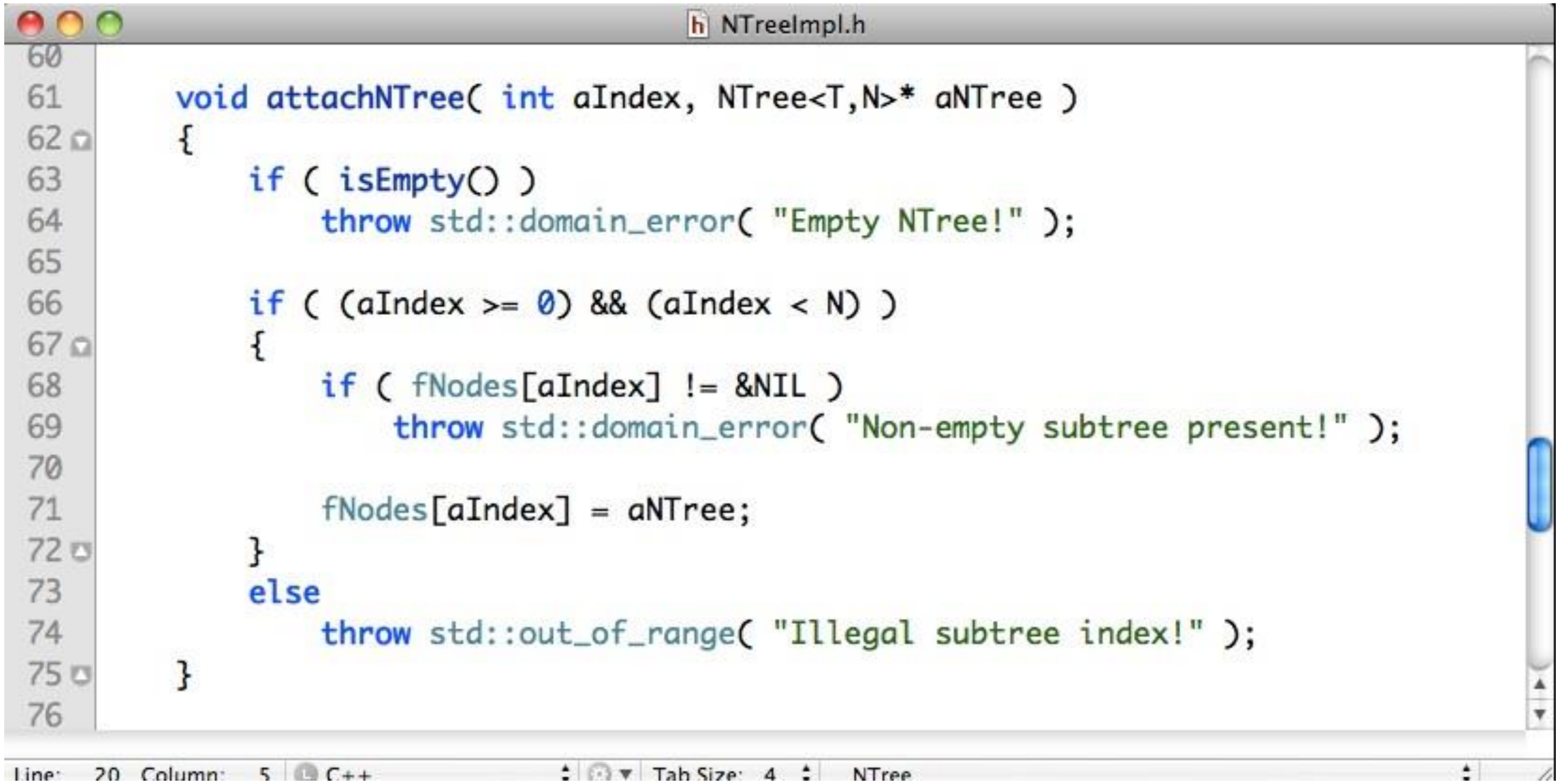


NTreeImpl.h

```
    bool isEmpty() const
    {
        return this == &NIL;
    }

    const T& key() const
    {
        if ( isEmpty() )
            throw std::domain_error( "Empty NTree!" );

        return *fKey;
    }
```

Line: 32 Column: 1 C++ Tab Size: 4 ~NTree

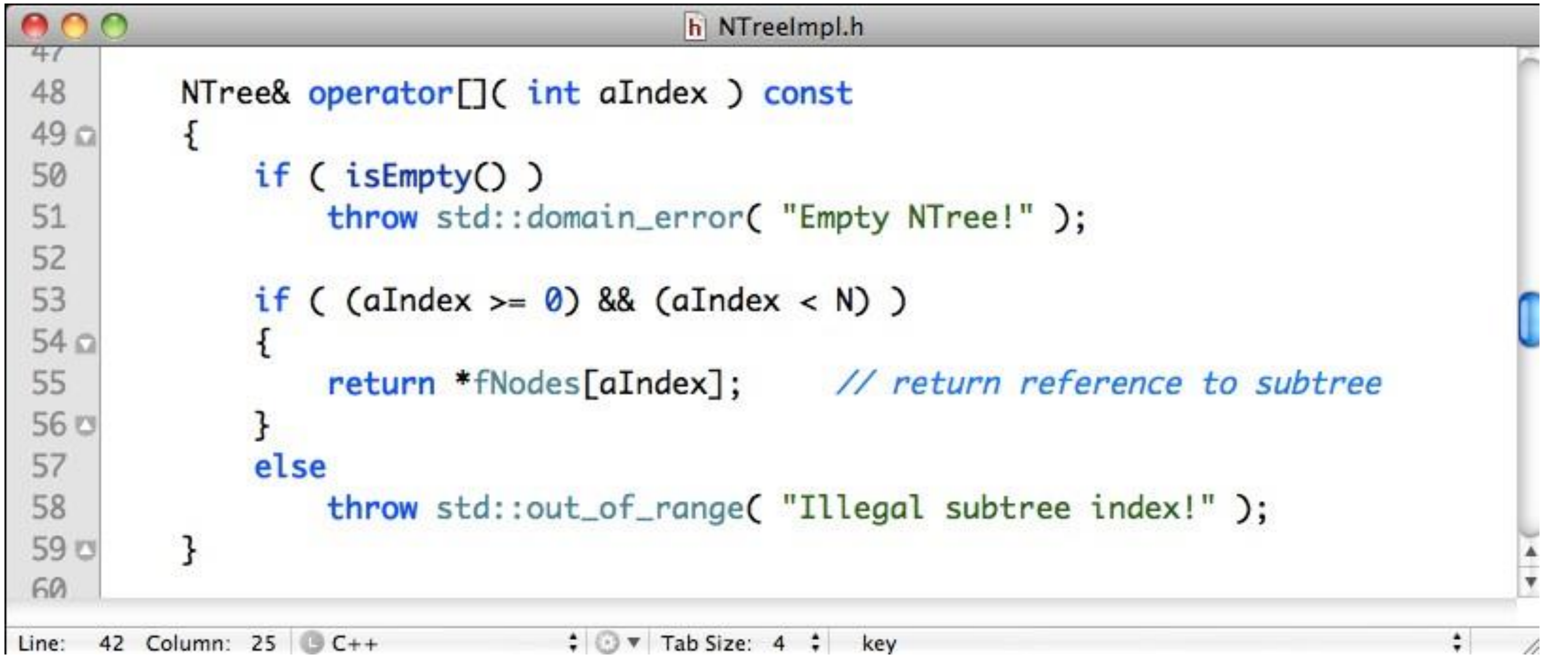
Attaching a New Subtree


NTreeImpl.h

```
void attachNTree( int aIndex, NTree<T,N>* aNTree )
{
    if ( isEmpty() )
        throw std::domain_error( "Empty NTree!" );

    if ( (aIndex >= 0) && (aIndex < N) )
    {
        if ( fNodes[aIndex] != &NIL )
            throw std::domain_error( "Non-empty subtree present!" );

        fNodes[aIndex] = aNTree;
    }
    else
        throw std::out_of_range( "Illegal subtree index!" );
}
```

Line: 20 Column: 5 C++ Tab Size: 4 NTree

Accessing a Subtree (Operator [])



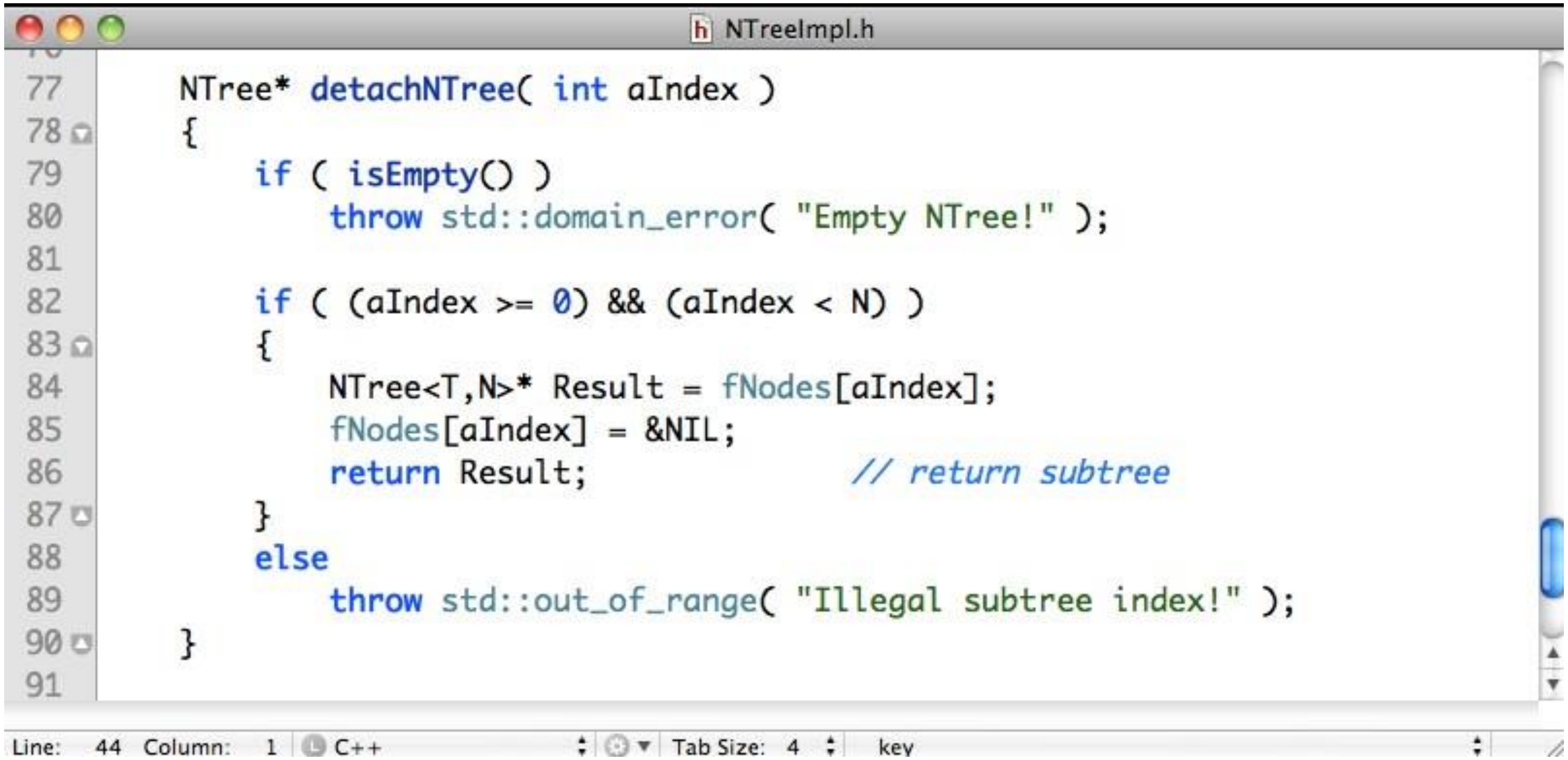
NTreeImpl.h

```
    NTree& operator[]( int aIndex ) const
    {
        if ( isEmpty() )
            throw std::domain_error( "Empty NTree!" );

        if ( (aIndex >= 0) && (aIndex < N) )
        {
            return *fNodes[aIndex];     // return reference to subtree
        }
        else
            throw std::out_of_range( "Illegal subtree index!" );
    }
```

Line: 42 Column: 25 C++ Tab Size: 4 key

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

Removing a Subtree

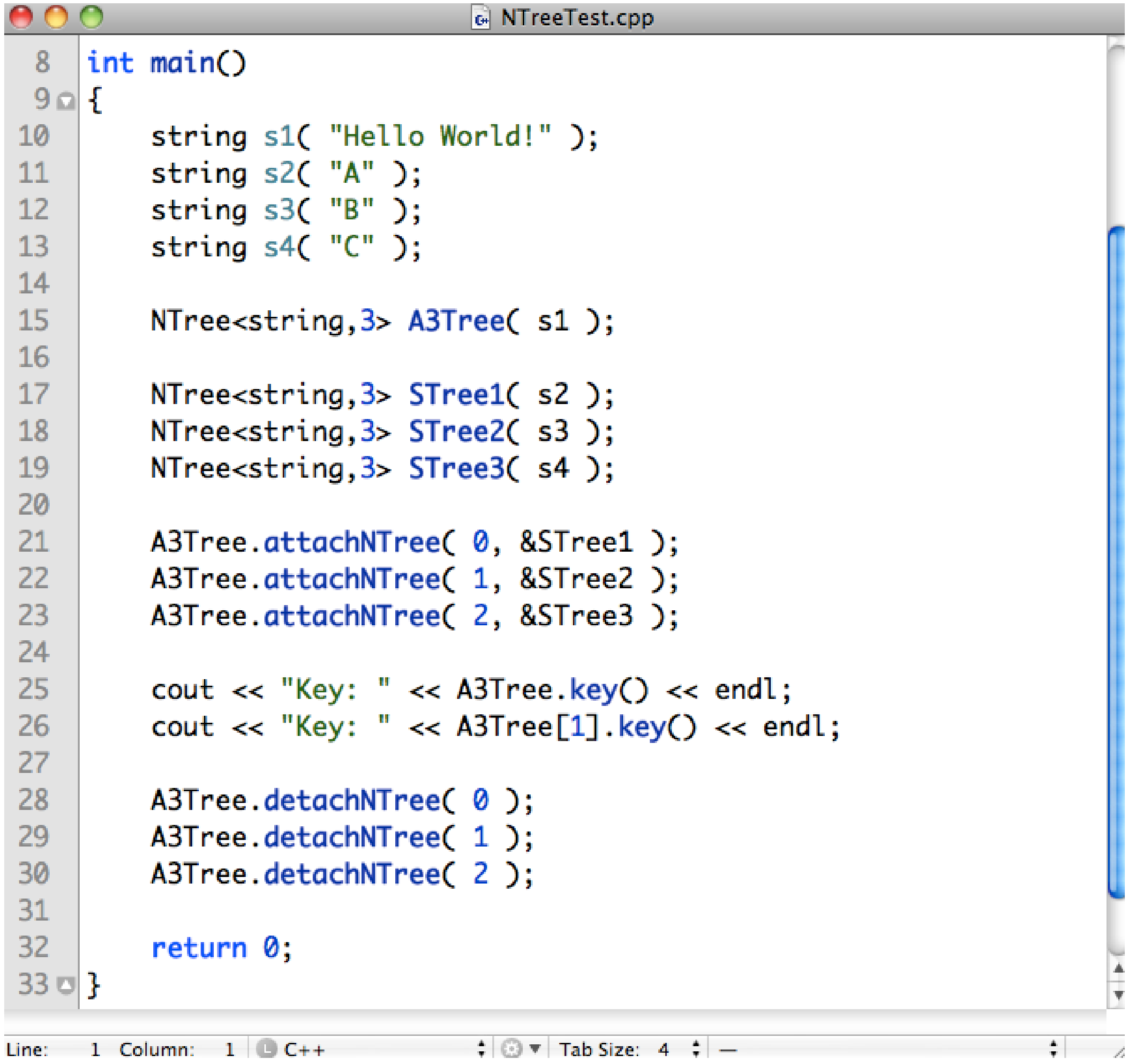


NTreeImpl.h

```
NTree* detachNTree( int aIndex )
{
    if ( isEmpty() )
        throw std::domain_error( "Empty NTree!" );

    if ( (aIndex >= 0) && (aIndex < N) )
    {
        NTree<T,N>* Result = fNodes[aIndex];
        fNodes[aIndex] = &NIL;
        return Result;                    // return subtree
    }
    else
        throw std::out_of_range( "Illegal subtree index!" );
}
```

Line: 44 Column: 1 C++ Tab Size: 4 key

A NTree Example



NTreeTest.cpp

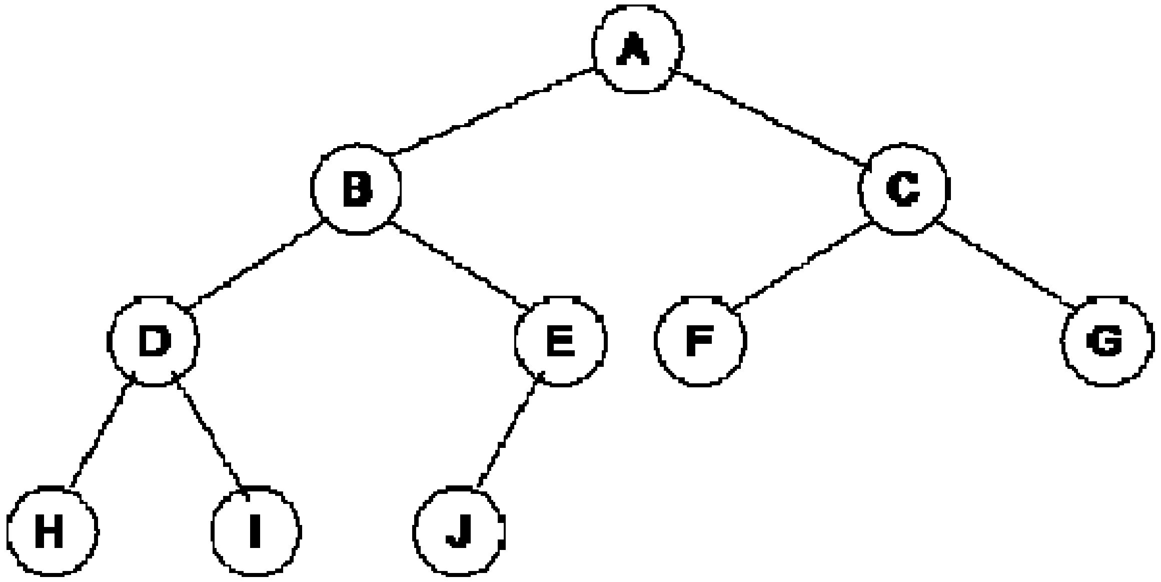
```cpp
int main()
{
    string s1( "Hello World!" );
    string s2( "A" );
    string s3( "B" );
    string s4( "C" );

    NTree<string,3> A3Tree( s1 );

    NTree<string,3> STree1( s2 );
    NTree<string,3> STree2( s3 );
    NTree<string,3> STree3( s4 );

    A3Tree.attachNTree( 0, &STree1 );
    A3Tree.attachNTree( 1, &STree2 );
    A3Tree.attachNTree( 2, &STree3 );

    cout << "Key: " << A3Tree.key() << endl;
    cout << "Key: " << A3Tree[1].key() << endl;

    A3Tree.detachNTree( 0 );
    A3Tree.detachNTree( 1 );
    A3Tree.detachNTree( 2 );

    return 0;
}
```

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

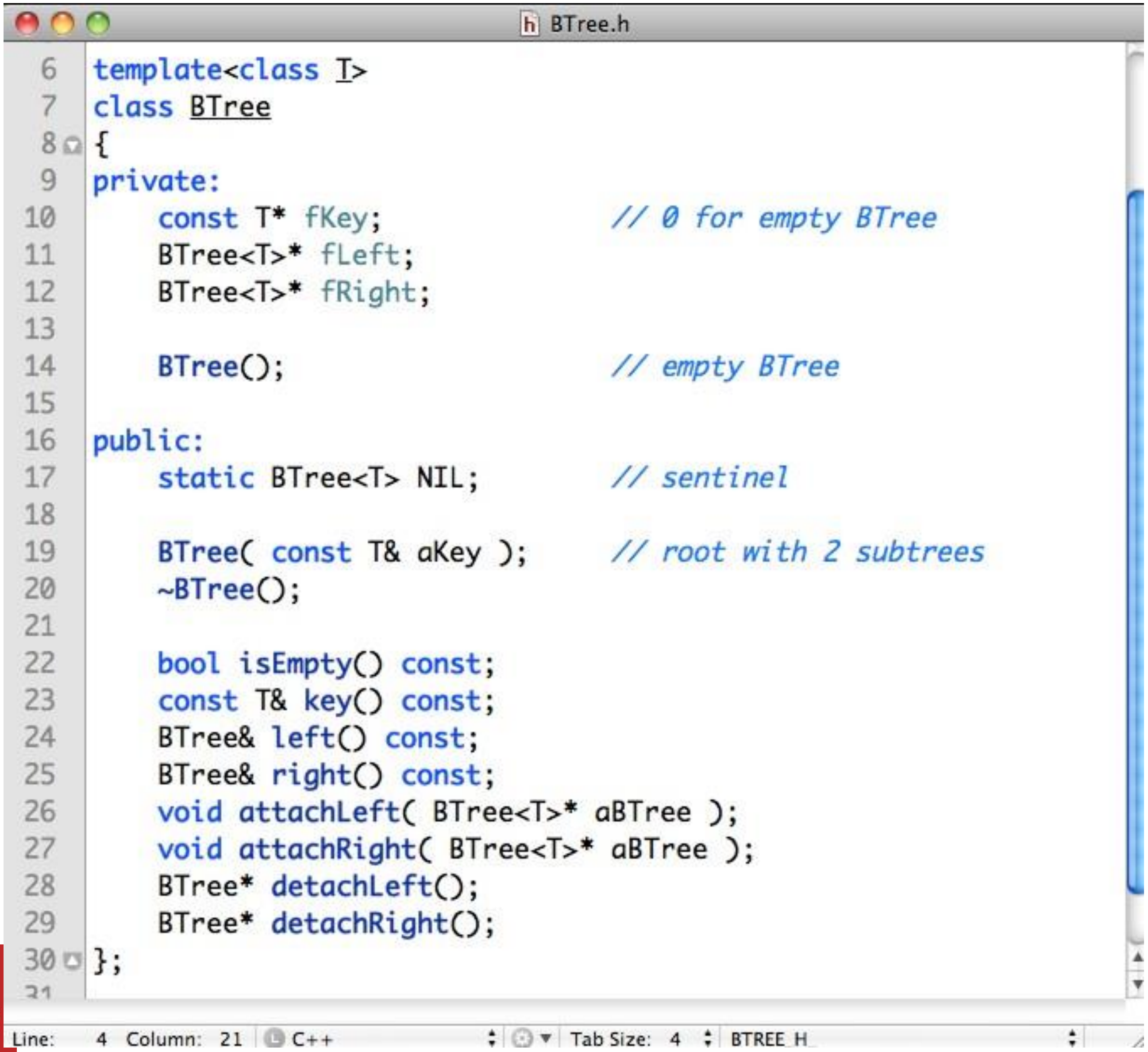
Binary Trees (2-ary Trees)

- Definition: A Binary Tree is a Tree in which each node has a degree of at most 2.
- A binary tree T is a finite set of nodes with one of the following properties:
  - Either the set is empty, T = Ø, or
  - The set consists of a root, R, and exactly 2 distinct binary trees TL and TR, T = {R, TL, TR}.
- The tree T**L** is called the **left** subtree of T and the tree T**R** is called the **right** subtree of T.

Example : Binary Tree



BTree Class

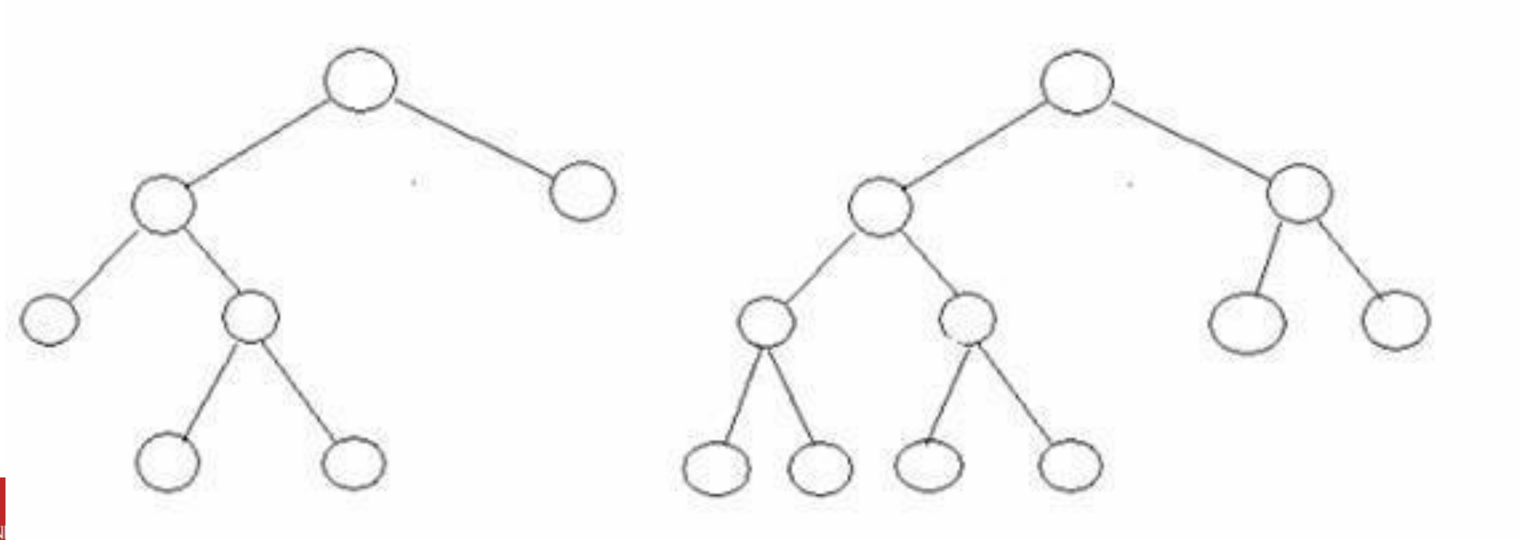

BTree.h

```
template<class T>
class BTree
{
private:
    const T* fKey;               // 0 for empty BTree
    BTree<T>* fLeft;
    BTree<T>* fRight;

    BTree();                     // empty BTree

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

    BTree( const T& aKey );      // root with 2 subtrees
    ~BTree();

    bool isEmpty() const;
    const T& key() const;
    BTree& left() const;
    BTree& right() const;
    void attachLeft( BTree<T>* aBTree );
    void attachRight( BTree<T>* aBTree );
    BTree* detachLeft();
    BTree* detachRight();
};
```

Line: 4 Column: 21 C++ Tab Size: 4 BTREE_H

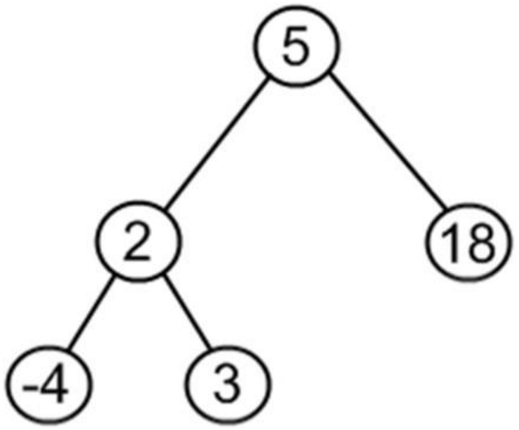
More on binary tree

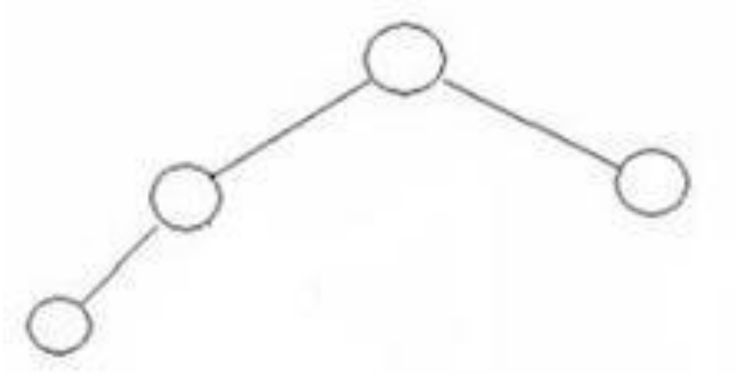
- **Full Binary tree:**
  - Every node should have exactly 2 nodes except the leaves. It is also called as **Strict Binary Tree** or **Proper Binary Tree.**
  - Every node must have either 0 or 2 children.



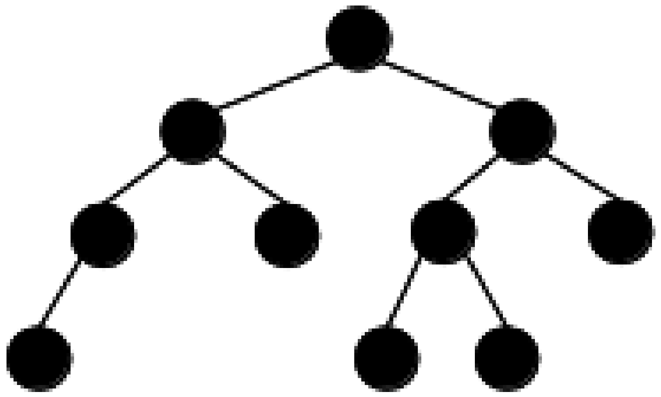
More on binary tree

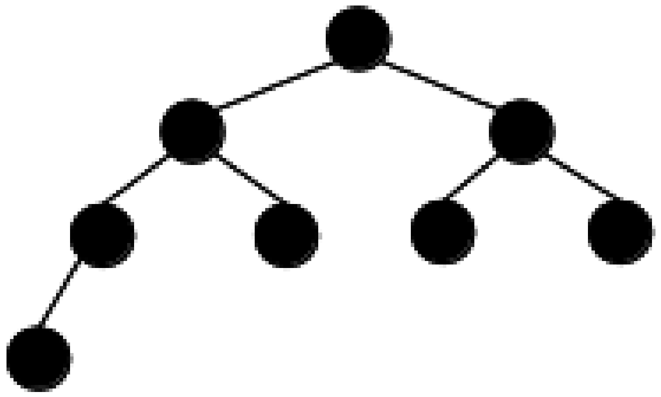
- **Complete Binary Tree:**
  - It is a binary tree in which every level (except possibly the last) is completely filled, and all nodes are as far left as possible.

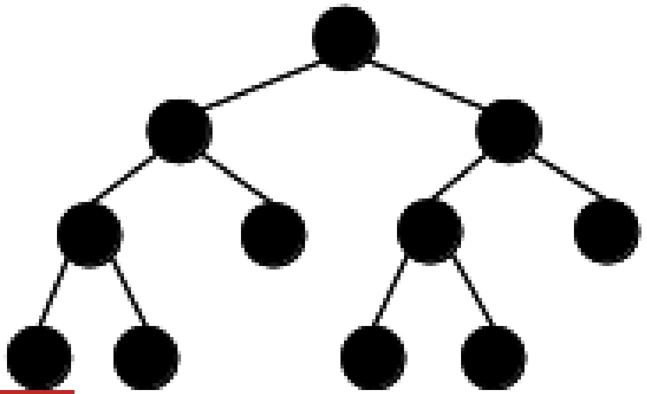


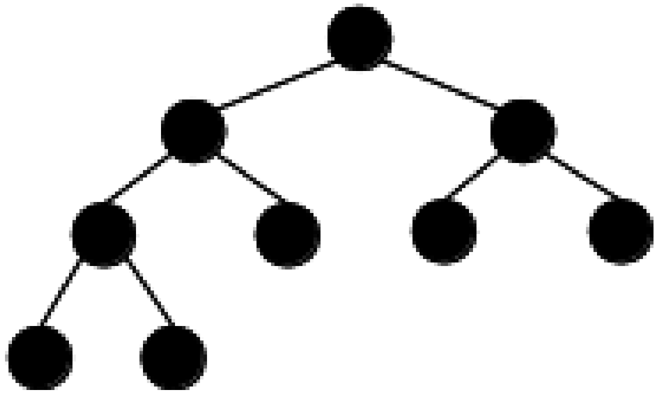


More on binary tree

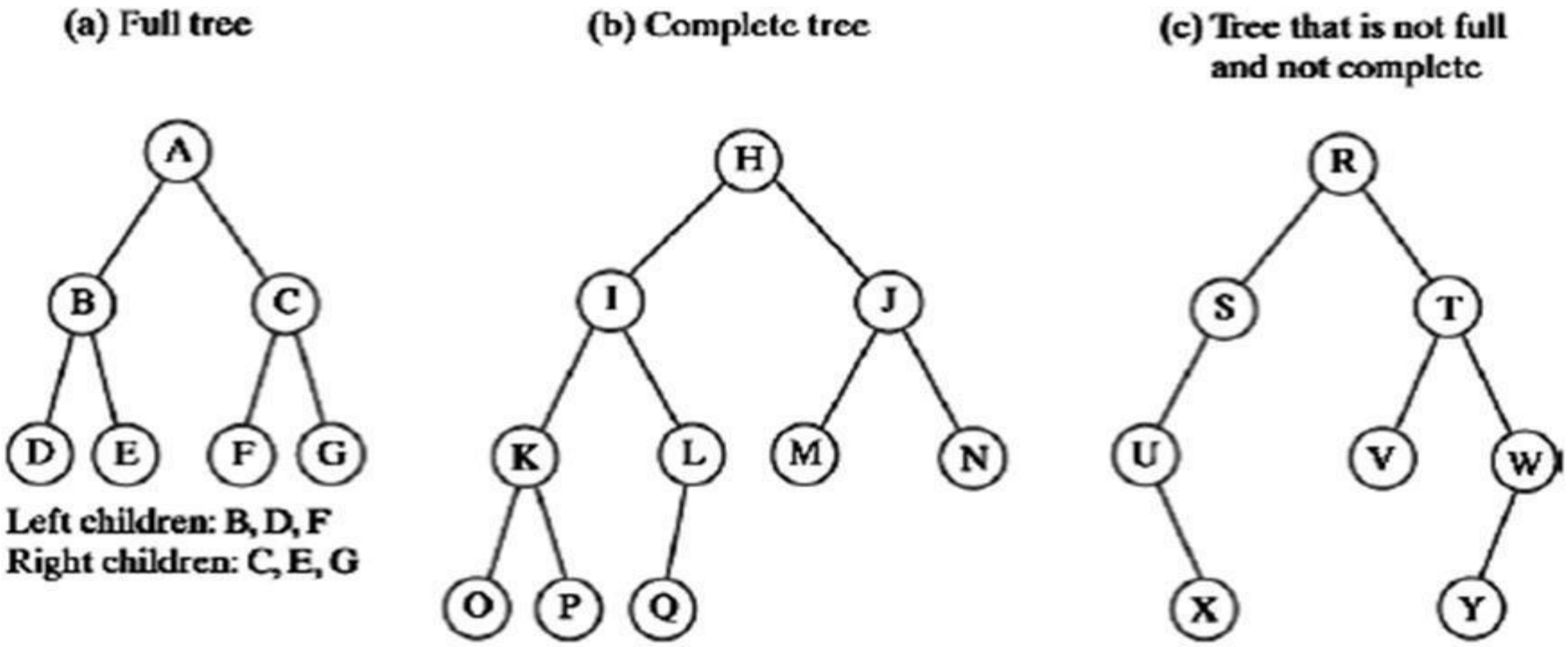

Neither complete nor full


Complete but not full


Full but not complete


Complete and full

More on binary tree



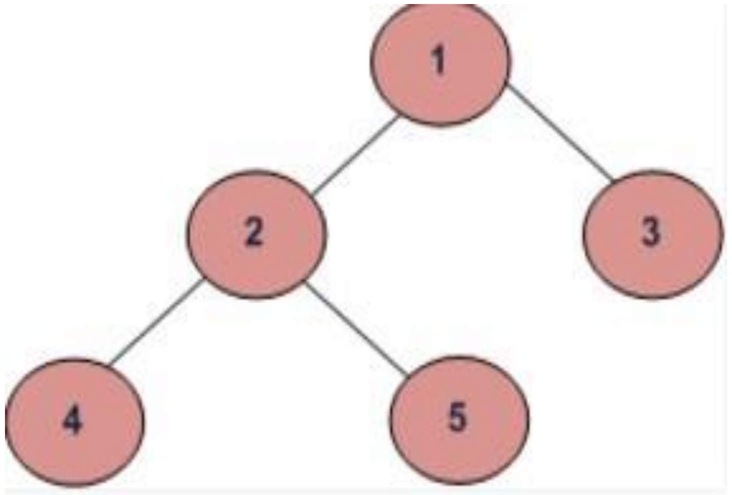
Tree Traversal

- 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 (Depth First Search, DFS),
- Breadth-first traversal (Breadth First Search, BFS)

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.

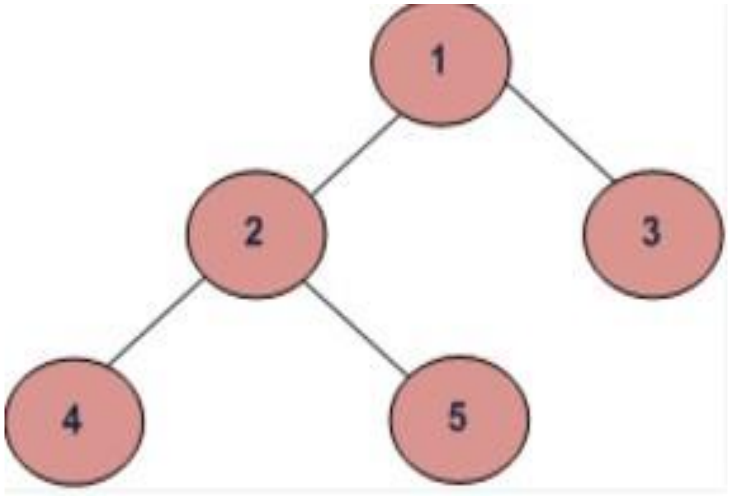
Example of Depth-first Traversal



- (a) Preorder (Root, Left, Right) : 1 2 4 5 3
- (b) Postorder (Left, Right, Root) : 4 5 2 3 1
- (c) Inorder (Left, Root, Right) : 4 2 5 1 3

Breadth-first Traversal

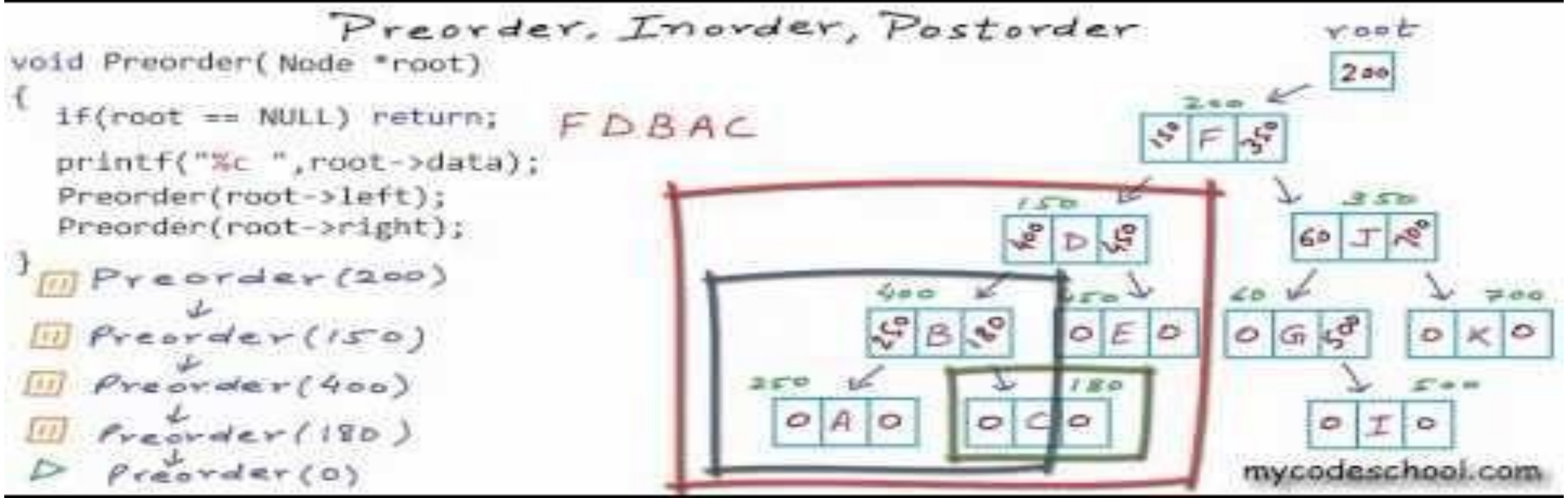
- Breadth-first traversal visits the nodes of a tree in the order of their depth (from left to right).



- BFS: 1 2 3 4 5

DFS: Preorder, Inorder and Postorder

Note: Slight mistake for Inorder example, there is no H in the actual answer.



```
void Preorder( Node *root)
{
    if(root == NULL) return;
    printf("%c ",root->data);
    Preorder(root->left);
    Preorder(root->right);
}
```

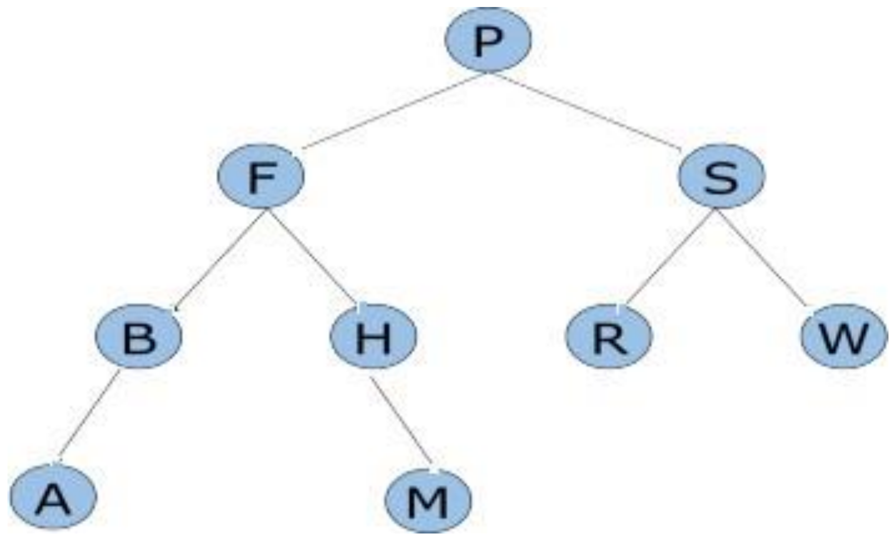
```
Preorder(200)
Preorder(150)
Preorder(400)
Preorder(180)
Preorder(0)
```

F D B A C

URL: https://www.youtube.com/watch?v=gm8DUJJhmY4

Exercise

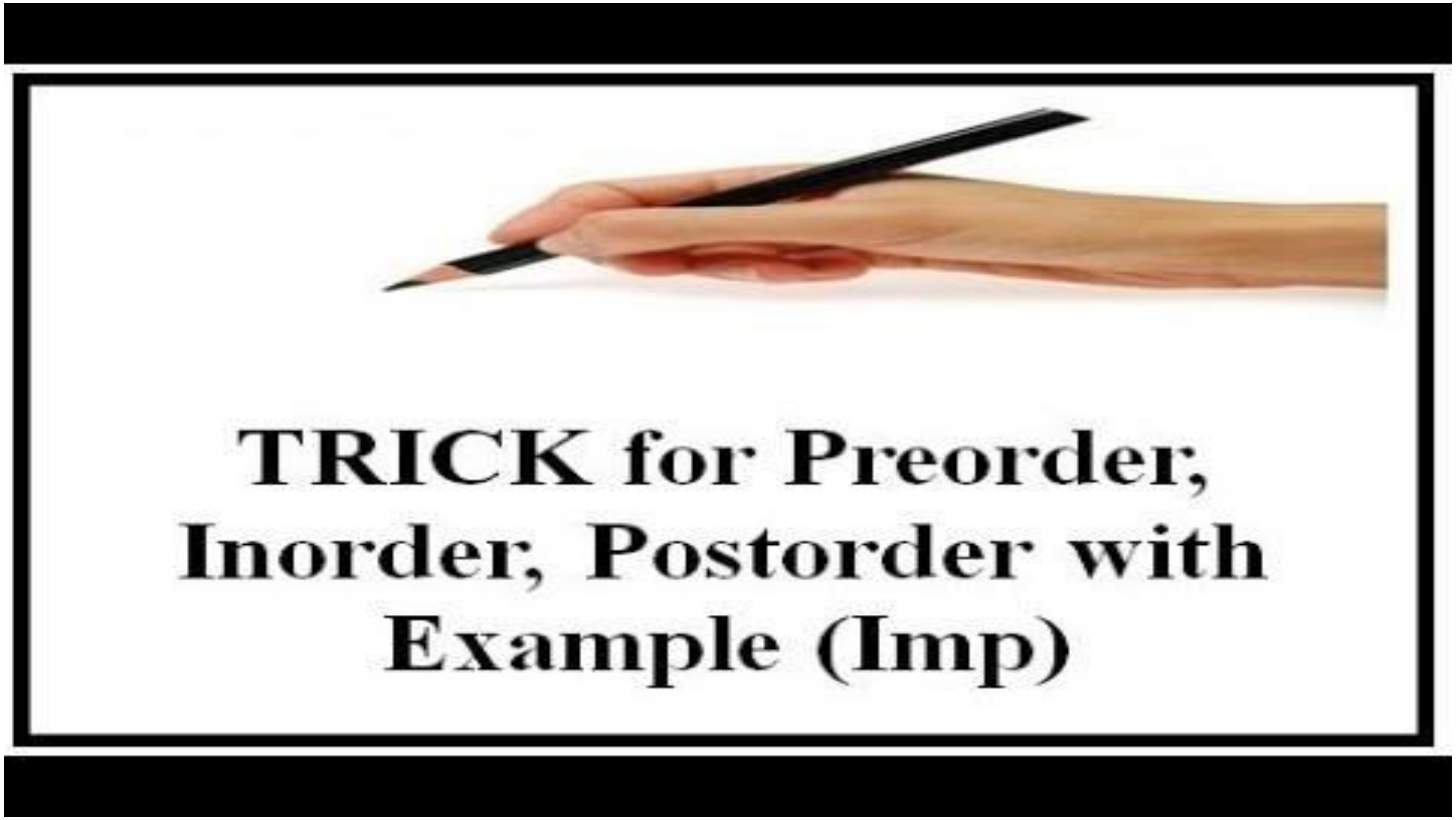
- Write the pre-order, in-order, post-order traversal of the following binary tree:



- Is the binary tree complete or full?

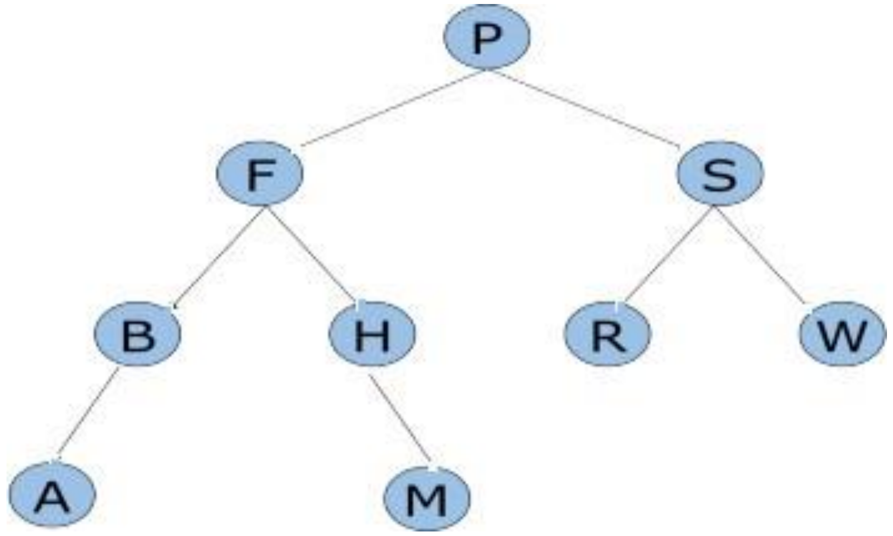
Tricks for Preorder, Inorder and Postorder

Note: The video is in Hindi but the tricks for solving are great.



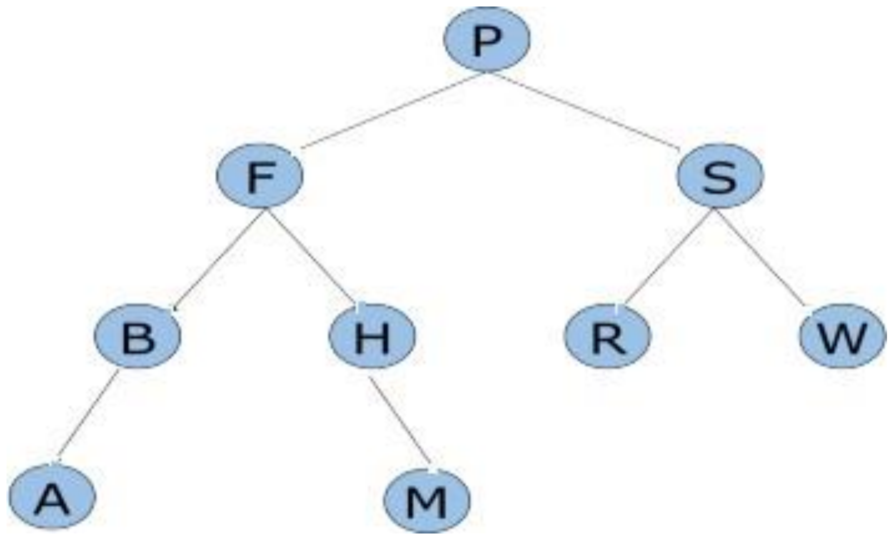
URL: https://www.youtube.com/watch?v=gm8DUJJhmY4

Exercise (Answers)



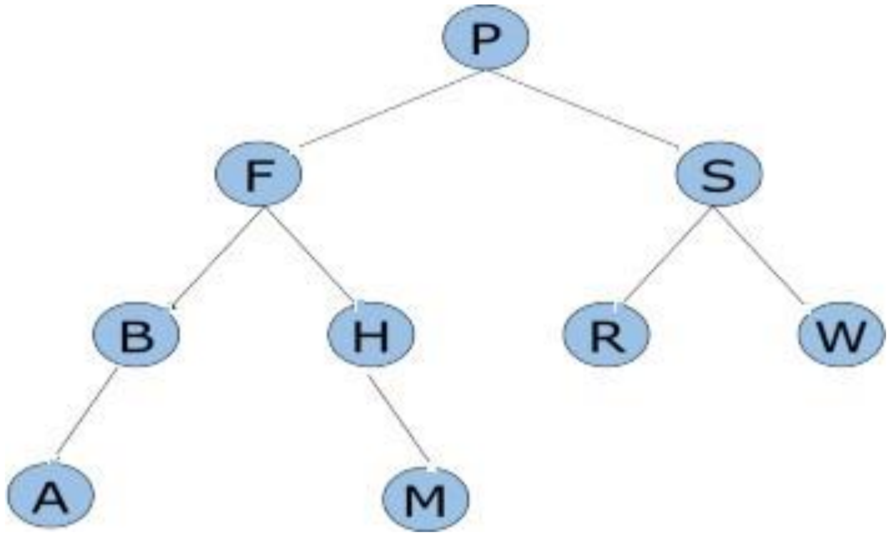
- Pre-order Traversal (Root, Left, Right):
  - P, F, B, A, H, M, S, R, W

Exercise (Answers)



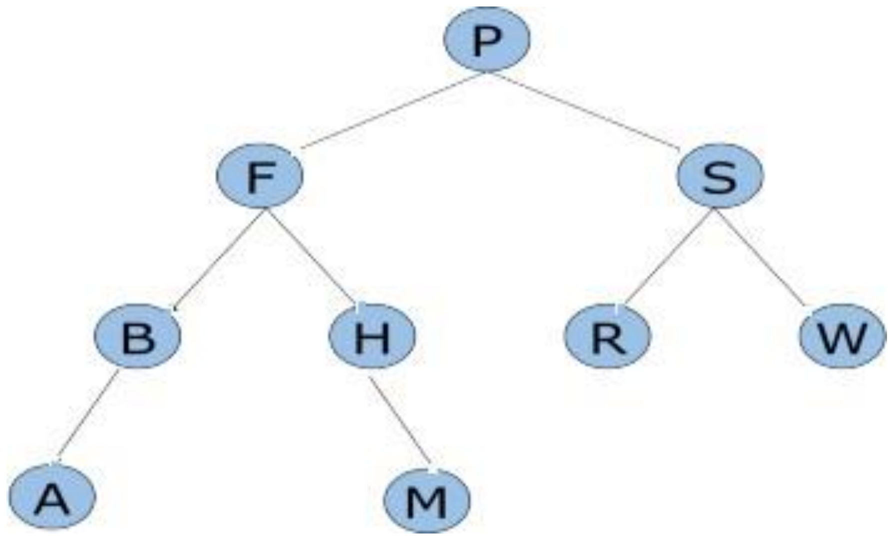
- In-order Traversal (Left, Root, Right):
  - A, B, F, H, M, P, R, S, W

Exercise (Answers)



- Post-order Traversal (Left, Right, Root):
  - A, B, M, H, F, R, W, S, P

Exercise (Answers)



- The binary tree is not complete and not full.
- It is not complete as some of the nodes have 1 leaf. In a complete binary tree, each nodes much have 0 or 2 children.
  - It is not full as not all nodes are as far left as possible.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Lecture 9

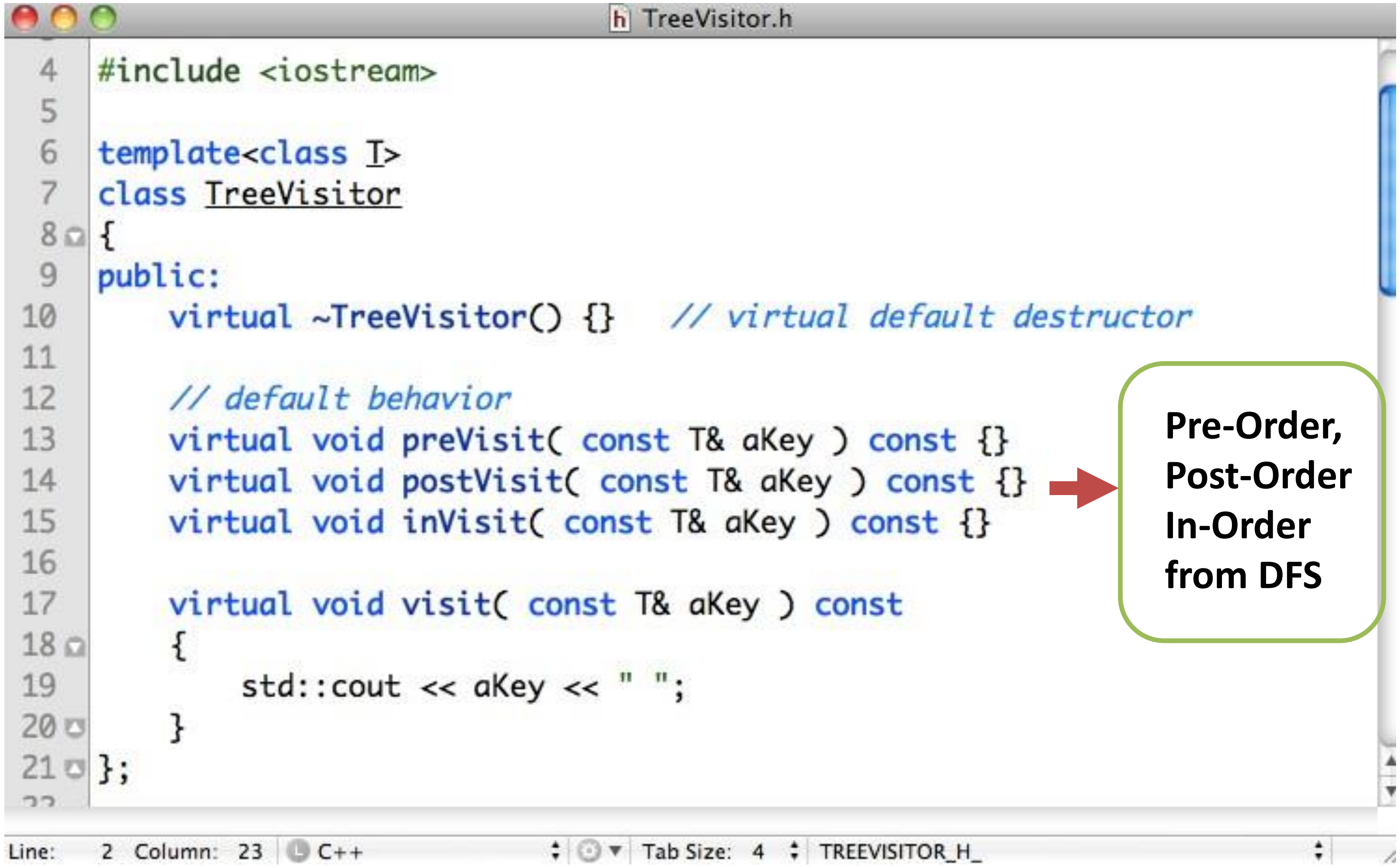
The Visitor Pattern

- The Visitor Pattern falls under the category of behavioral design patterns
- **Intent**
  - ☐ Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
  - ☐ The classic technique for recovering lost type information.
  - ☐ Do the right thing based on the type of two objects.
    - ☐ **Double Dispatch** is used to invoke an overloaded method where the parameters vary among an inheritance hierarchy

The Visitor Pattern

- 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.

A Tree Visitor



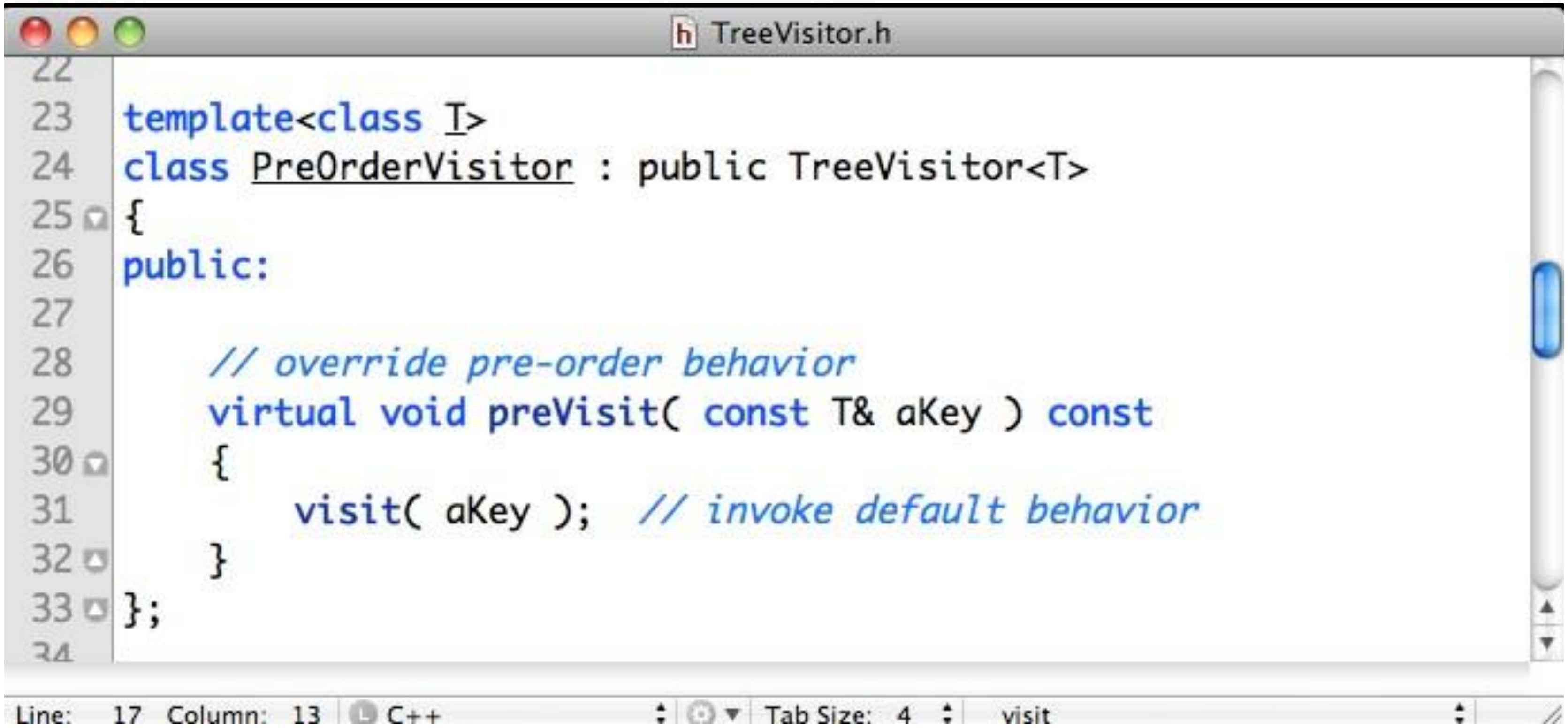
TreeVisitor.h

```
#include <iostream>

template<class T>
class TreeVisitor
{
public:
    virtual ~TreeVisitor() {}   // virtual default destructor

    // default behavior
    virtual void preVisit( const T& aKey ) const {}
    virtual void postVisit( const T& aKey ) const {}
    virtual void inVisit( const T& aKey ) const {}

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

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

Pre-Order,
Post-Order
In-Order
from DFS

PreOrderVisitor

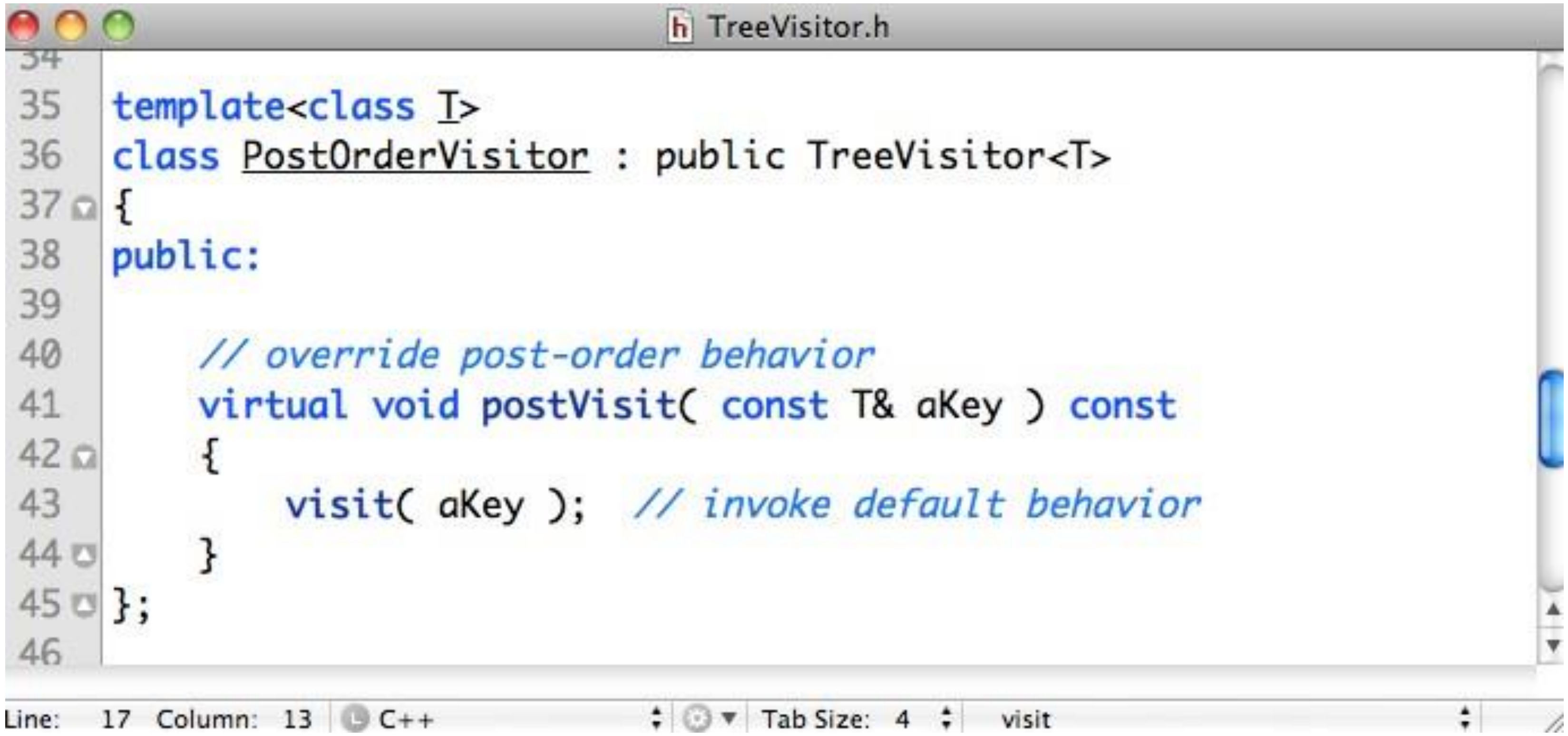


TreeVisitor.h

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

    // override pre-order behavior
    virtual void preVisit( const T& aKey ) const
    {
        visit( aKey );  // invoke default behavior
    }
};
```

Line: 17 Column: 13 C++ Tab Size: 4 visit

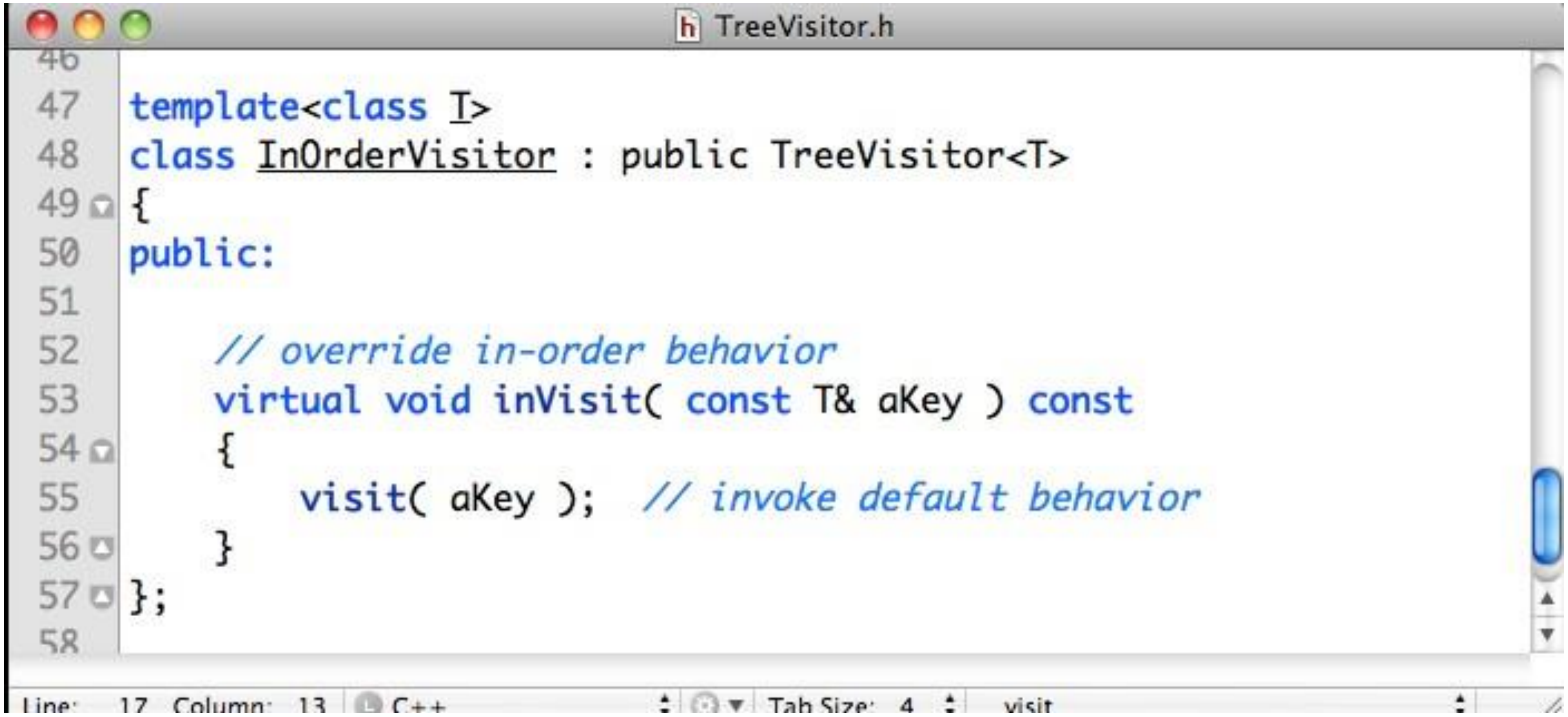
PostOrderVisitor



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

    // override post-order behavior
    virtual void postVisit( const T& aKey ) const
    {
        visit( aKey );  // invoke default behavior
    }
};

```

Line: 17 Column: 13 C++ Tab Size: 4 visit

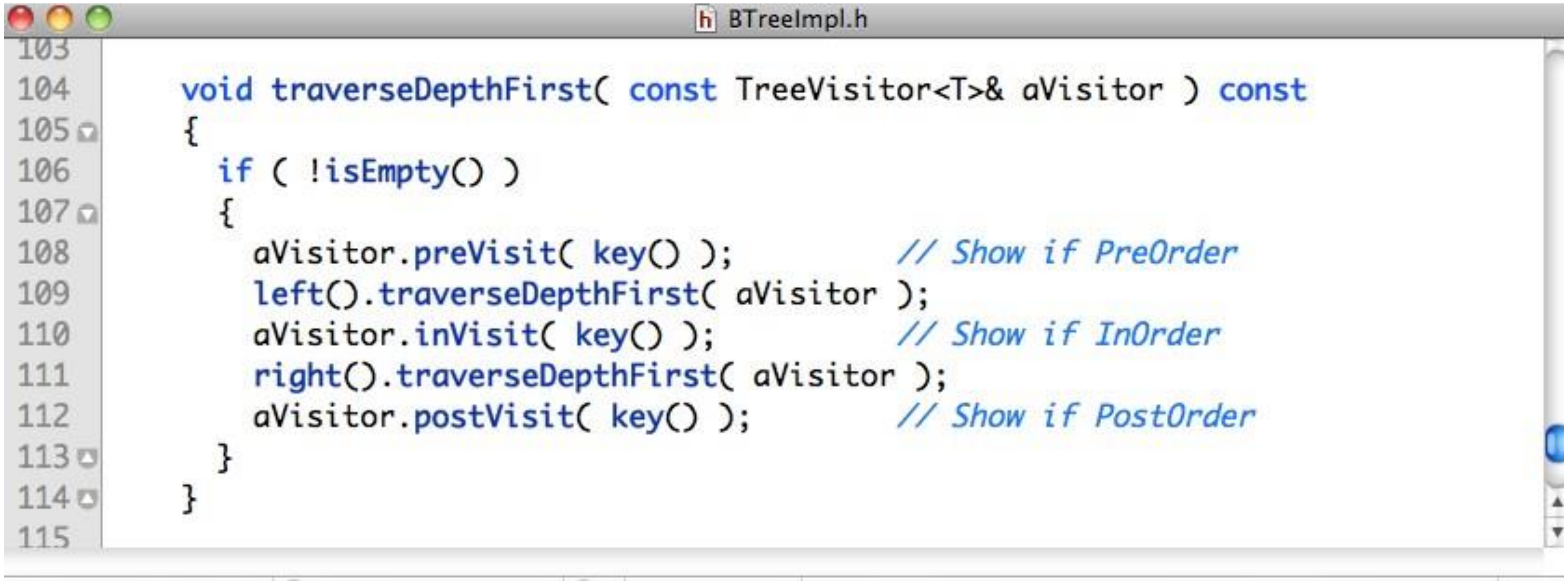
InOrderVisitor


TreeVisitor.h

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

    // override in-order behavior
    virtual void inVisit( const T& aKey ) const
    {
        visit( aKey );   // invoke default behavior
    }
};
```

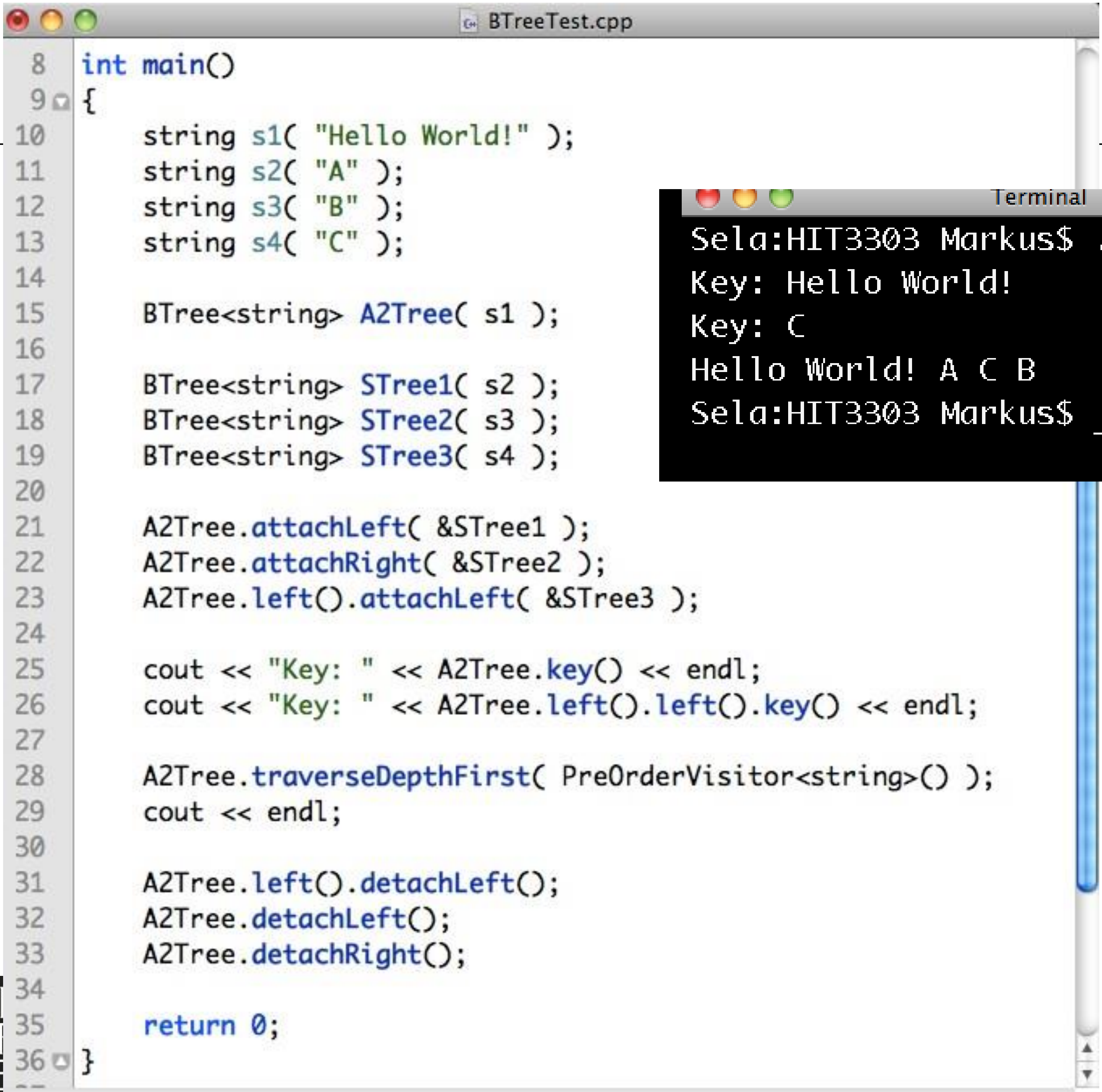
Line: 17 Column: 13 C++ Tab Size: 4 visit

Depth-first Traversal for BTree



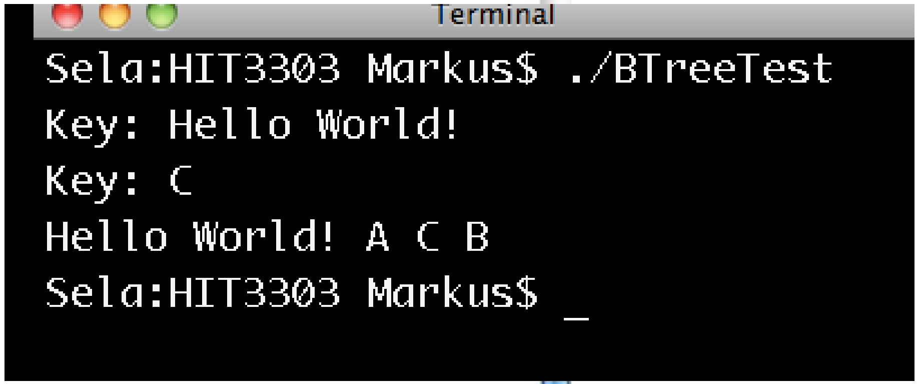
BTreeImpl.h

```
    void traverseDepthFirst( const TreeVisitor<T>& aVisitor ) const
    {
      if ( !isEmpty() )
      {
        aVisitor.preVisit( key() );           // Show if PreOrder
        left().traverseDepthFirst( aVisitor );
        aVisitor.inVisit( key() );            // Show if InOrder
        right().traverseDepthFirst( aVisitor );
        aVisitor.postVisit( key() );          // Show if PostOrder
      }
    }
```



BTreeTest.cpp

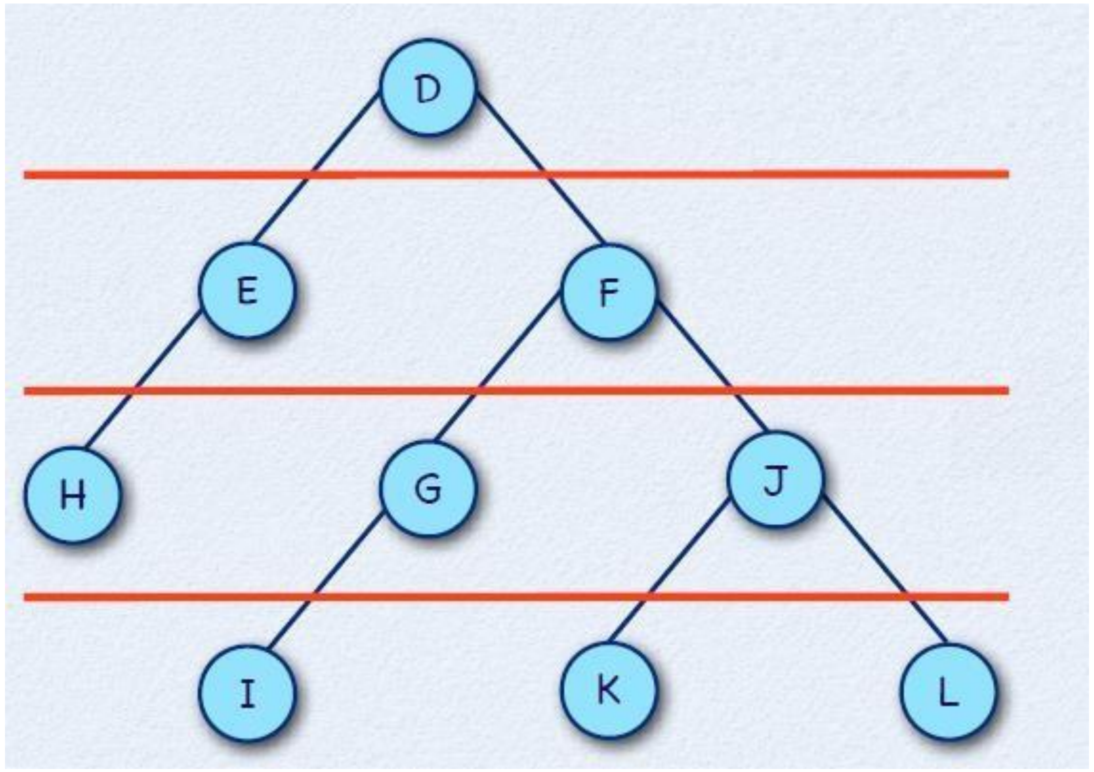
```cpp
int main()
{
    string s1( "Hello World!" );
    string s2( "A" );
    string s3( "B" );
    string s4( "C" );

    BTree<string> A2Tree( s1 );

    BTree<string> STree1( s2 );
    BTree<string> STree2( s3 );
    BTree<string> STree3( s4 );

    A2Tree.attachLeft( &STree1 );
    A2Tree.attachRight( &STree2 );
    A2Tree.left().attachLeft( &STree3 );

    cout << "Key: " << A2Tree.key() << endl;
    cout << "Key: " << A2Tree.left().left().key() << endl;

    A2Tree.traverseDepthFirst( PreOrderVisitor<string>() );
    cout << endl;

    A2Tree.left().detachLeft();
    A2Tree.detachLeft();
    A2Tree.detachRight();

    return 0;
}
```



Terminal

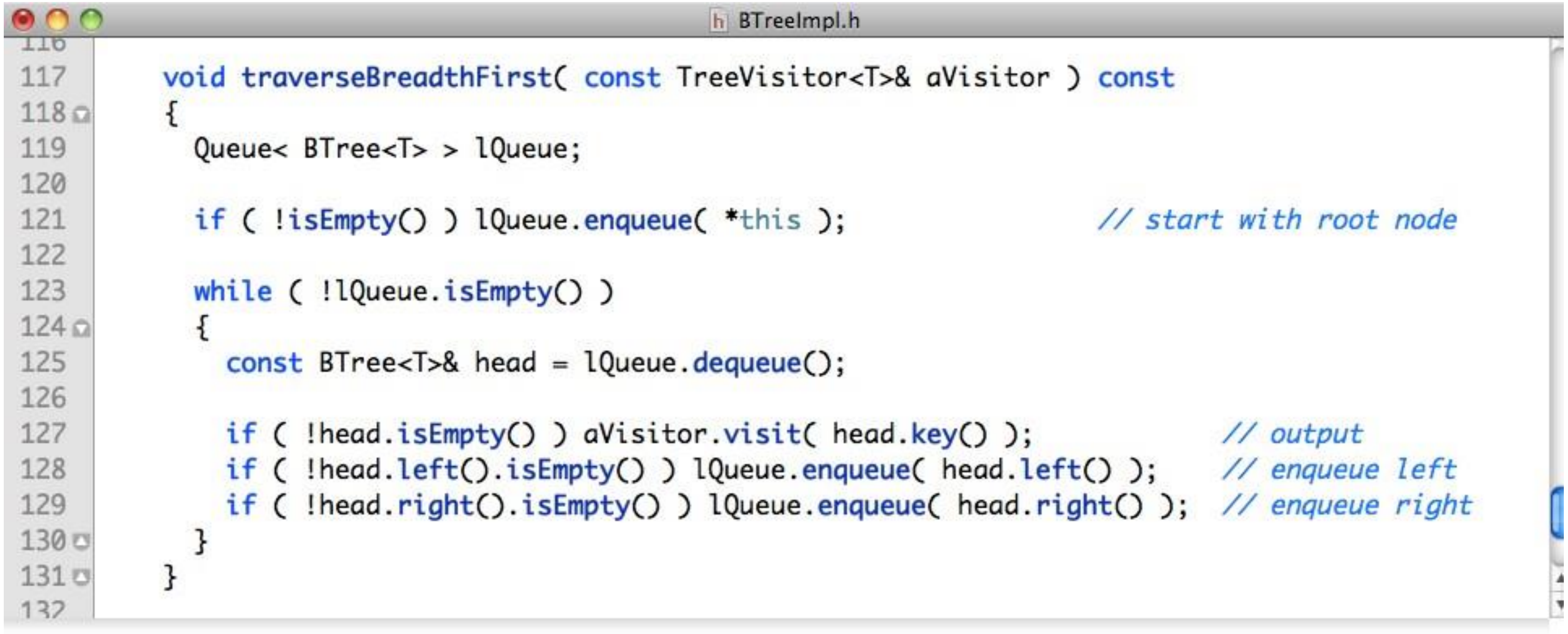
```
Sela:HIT3303 Markus$ ./BTreeTest
Key: Hello World!
Key: C
Hello World! A C B
Sela:HIT3303 Markus$ _
```

Breadth-first Traversal Implementation



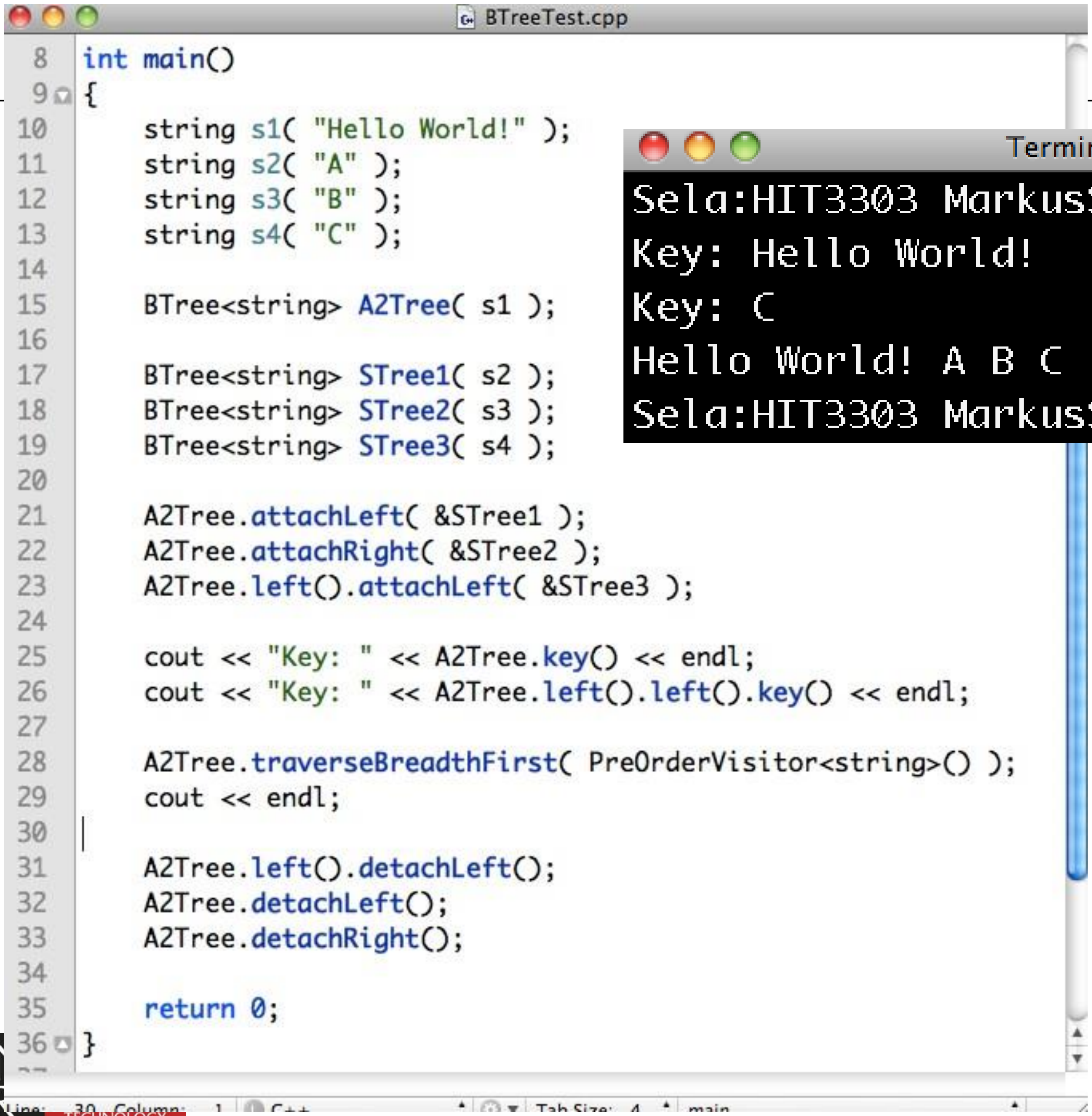
- Traversal : D-E-F-H-G-J-I-K-L

Breadth-first Traversal for BTree



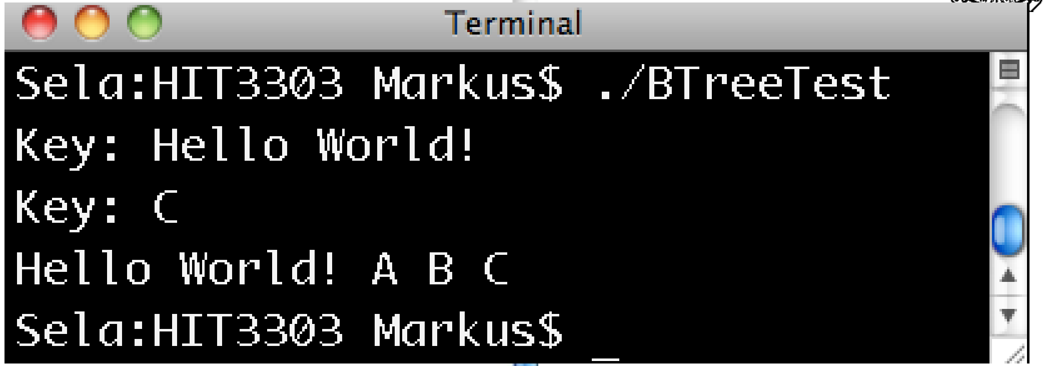
BTreeImpl.h

```
    void traverseBreadthFirst( const TreeVisitor<T>& aVisitor ) const
    {
      Queue< BTree<T> > lQueue;

      if ( !isEmpty() ) lQueue.enqueue( *this );              // start with root node

      while ( !lQueue.isEmpty() )
      {
        const BTree<T>& head = lQueue.dequeue();

        if ( !head.isEmpty() ) aVisitor.visit( head.key() );          // output
        if ( !head.left().isEmpty() ) lQueue.enqueue( head.left() );   // enqueue left
        if ( !head.right().isEmpty() ) lQueue.enqueue( head.right() ); // enqueue right
      }
    }
```



BTreeTest.cpp

```cpp
int main()
{
	string s1( "Hello World!" );
	string s2( "A" );
	string s3( "B" );
	string s4( "C" );

	BTree<string> A2Tree( s1 );

	BTree<string> STree1( s2 );
	BTree<string> STree2( s3 );
	BTree<string> STree3( s4 );

	A2Tree.attachLeft( &STree1 );
	A2Tree.attachRight( &STree2 );
	A2Tree.left().attachLeft( &STree3 );

	cout << "Key: " << A2Tree.key() << endl;
	cout << "Key: " << A2Tree.left().left().key() << endl;

	A2Tree.traverseBreadthFirst( PreOrderVisitor<string>() );
	cout << endl;

	A2Tree.left().detachLeft();
	A2Tree.detachLeft();
	A2Tree.detachRight();

	return 0;
}
```



Terminal

```
Sela:HIT3303 Markus$ ./BTreeTest
Key: Hello World!
Key: C
Hello World! A B C
Sela:HIT3303 Markus$ _
```

AVL Tree

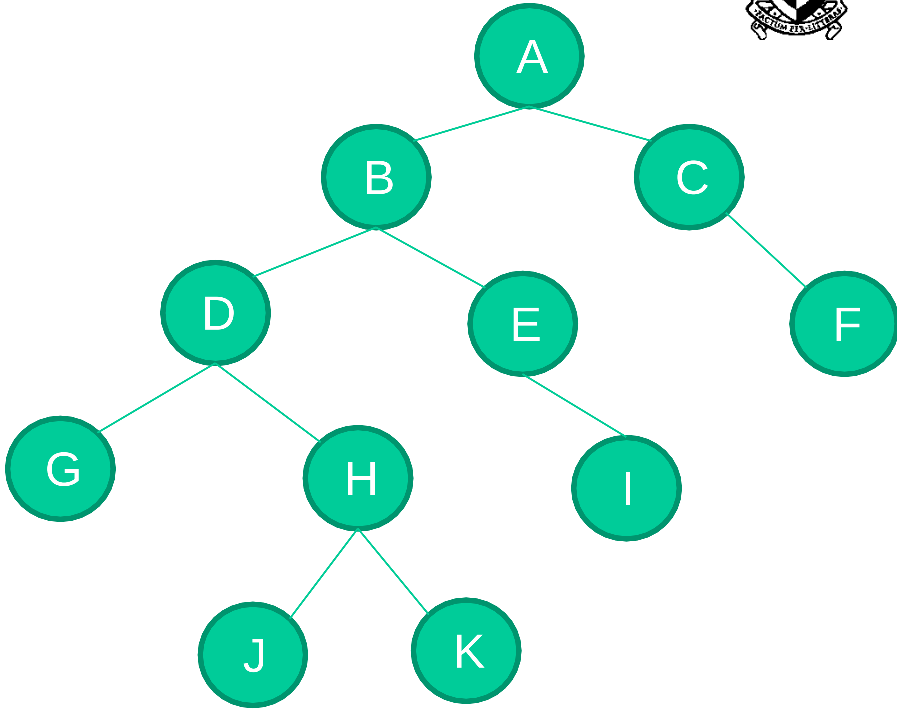
- AVL trees are height-balanced **binary** search trees
  - A tree where no leaf is much farther away from the root than any other leaf.
  - 1) Left subtree of Tree is balanced
    2) Right subtree of Tree is balanced
    3) The difference between heights of left subtree and right subtree is
    not more than 1.
- Balance factor of a node
  - height(left subtree) - height(right subtree)

AVL Tree

- An AVL tree has balance factor calculated at every node
- For every node, heights of left and right subtree can differ by no more than 1
- Store current heights in each node

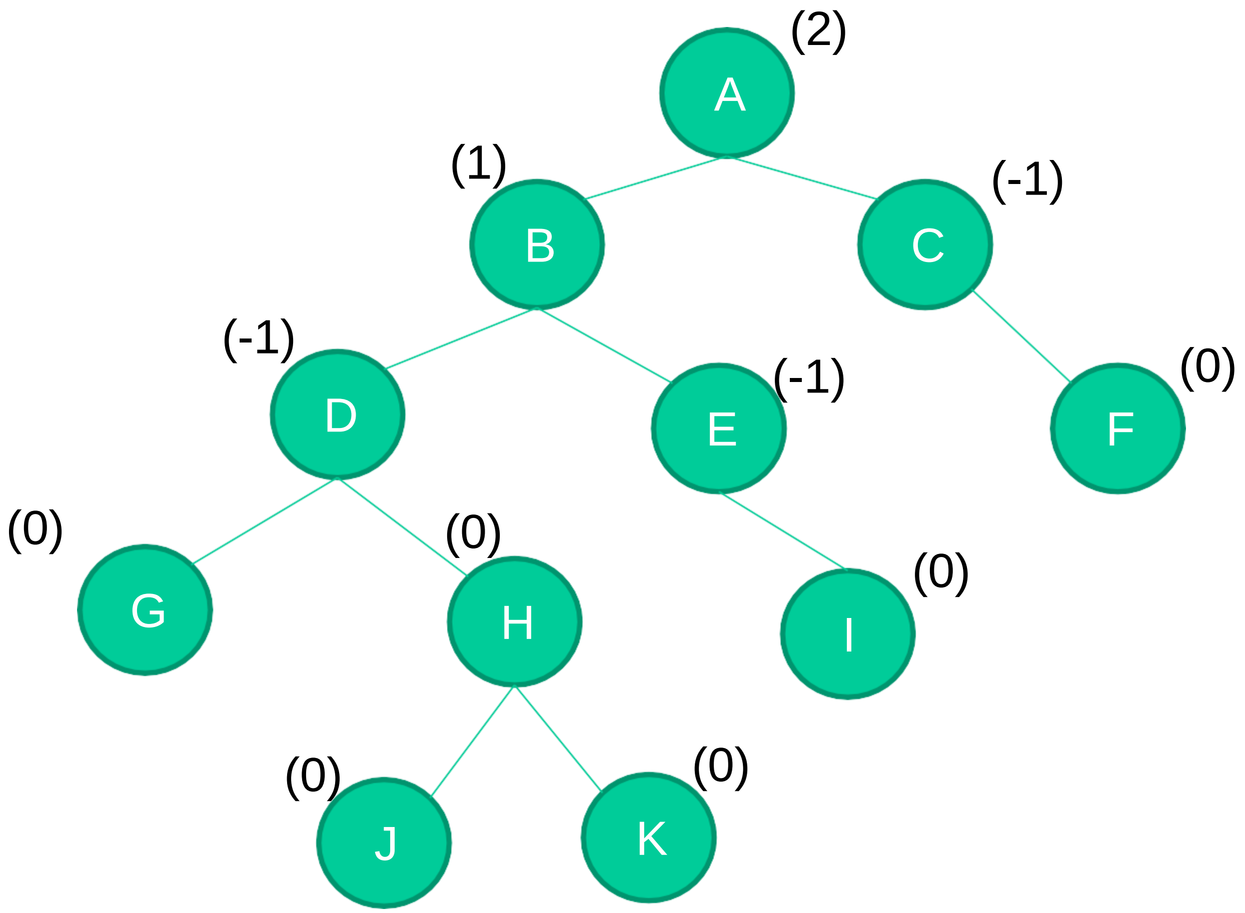
Is this a AVL tree?

- The height of the tree is 4, meaning the length of the longest path from the root to a leaf node.
- The height of the left subtree of the root is 3, meaning that the length of the longest path from the node B to one of the leaf nodes (G, J K or I).
- For finding the balancing factor of the **root** we subtract the height of the right subtree and the left subtree : 3-1 = 2.
- The balancing factor of the node with the key I is very easy to determine. We notice that the node has no children so the balancing factor is 0



- For finding the balancing factor of the node with key D we substract the height of the right subtree from the height of the left subtree: 0-1= -1.

Balance factor



The binary tree is **balanced** when all the balancing factors of all the nodes are -1,0,+1.

This binary tree is not balanced at the **Root**.

Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
- only nodes on the path from insertion point to root node have possibly changed in height
- So after the Insert, go back up to the root node by node, updating heights
- If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Insertions in AVL Trees

- Let the node that needs rebalancing be α.
- There are 4 cases:
- Outside Cases (require single rotation) :
  1. Insertion into left subtree of left child of α.
  2. Insertion into right subtree of right child of α.
- Inside Cases (require double rotation) :
  3. Insertion into right subtree of left child of α.
  4. Insertion into left subtree of right child of α.
- The rebalancing is performed through four separate rotation algorithms, left-left rotation, right-right rotation, right-left rotation and left-right rotation.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Lecture 10

Big-O notation

- Wiki : *Big O notation* is a mathematical *notation* that describes the limiting behaviour of a function when the argument tends towards a particular value or infinity
- It is used to describe the worse case, or ceiling of growth for an algorithm or a function.
- It is a simplified analysis of an algorithm’s efficiency/ complexity.
- **Asymptotic** analysis
- It focuses on basic computing steps

Big-O notation : General rules

- Ignores the constant coefficient
  - n, 2n, 5n, ... O(n)
- Functions/algorithms with “higher growth rate” dominate others
  - $O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(2^n)<O(n!)$

Big-O notation : General rules

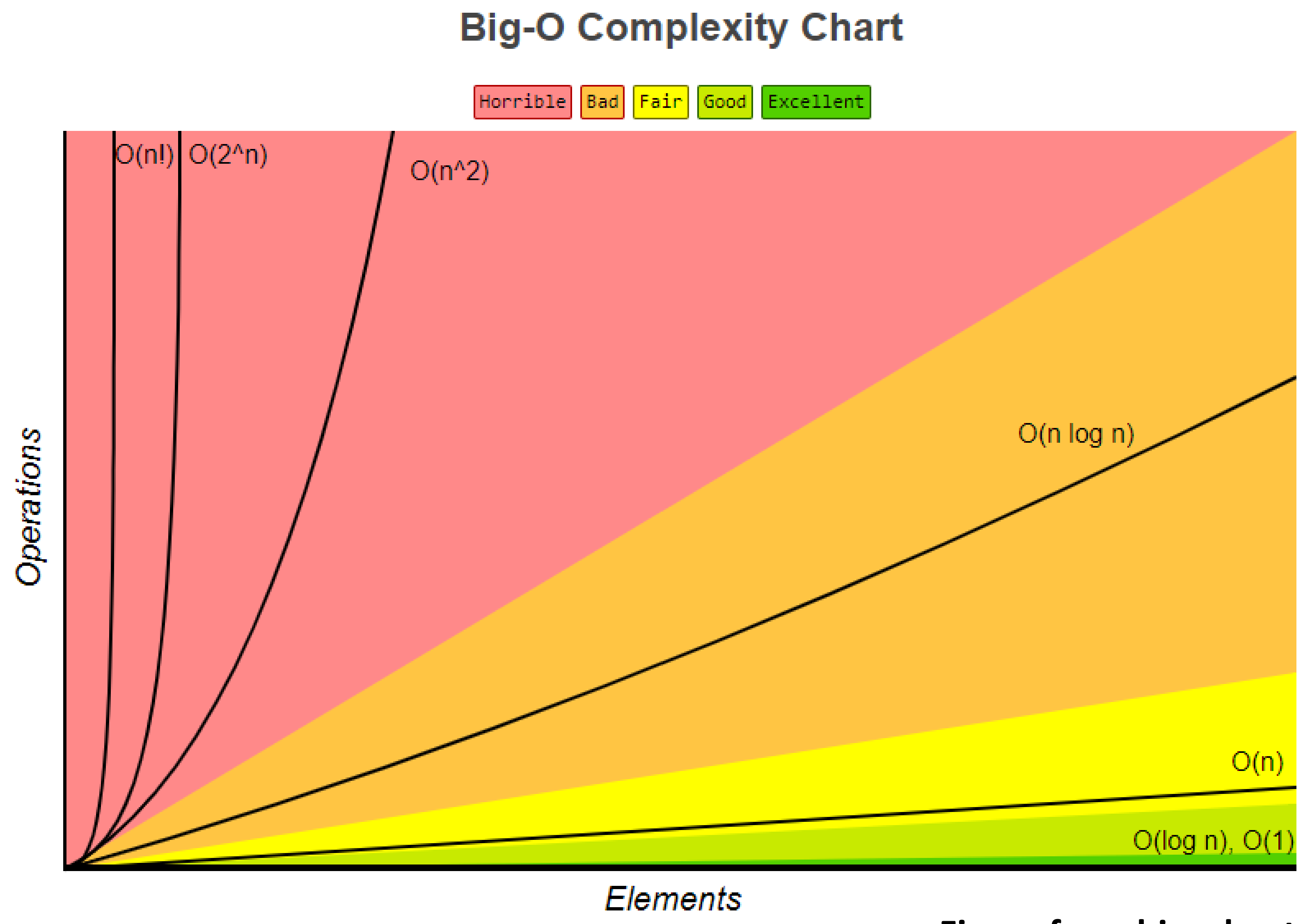


Figure from bigocheatsheet.com

Big-O : Constant time, O(1)

- Notation: **O(1)**
- `x = 3*(5+12)`
- x is independent from any input or additional variables
- We call such complexity, constant time

```
x=3*(5+12);

y=15-2;

print x-y;
```

  - What is big-O for this code?

    Total = O(1)+O(1)+O(1) = 3*O(1) => O(1)

    Since constants are ignored.

Big-O : Linear time, O(n)

- Notation: **O(n)**

```
for(int i =0; i<n; ++i)
    cout<<i<<endl;
```

- O(1) statement repeat n times. n is not a constant coefficient but it is possibly a variable.
- So it is n*O(1) = O(n)

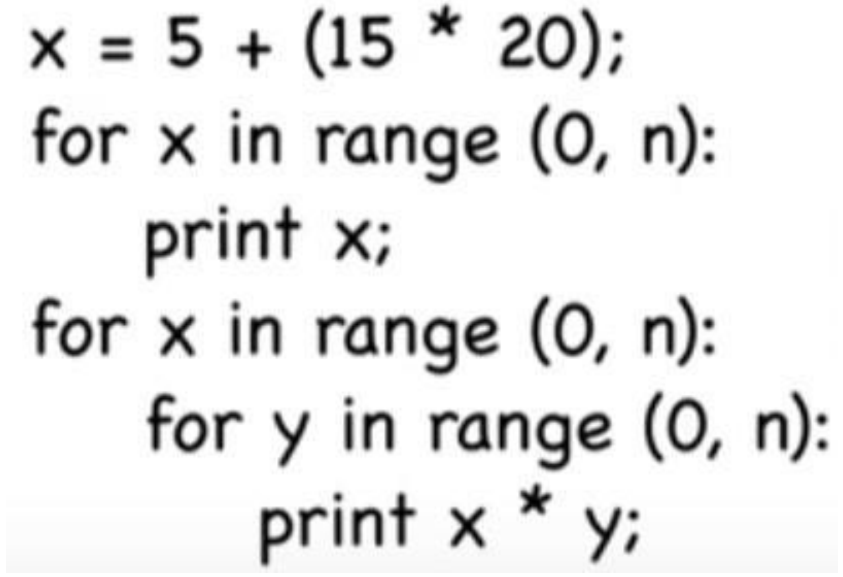
Big-O: Quadratic time, $O(n^2)$

- Two for loops

```
for(i =0; i<n; ++i)
    for(j =0; j<n; ++j)
        cout<<i*j<<endl;
```

Big-O notation: exercise 1

- What is the Big-O for this?



```
x = 5 + (15 * 20);
for x in range (0, n):
    print x;
for x in range (0, n):
    for y in range (0, n):
        print x * y;
```

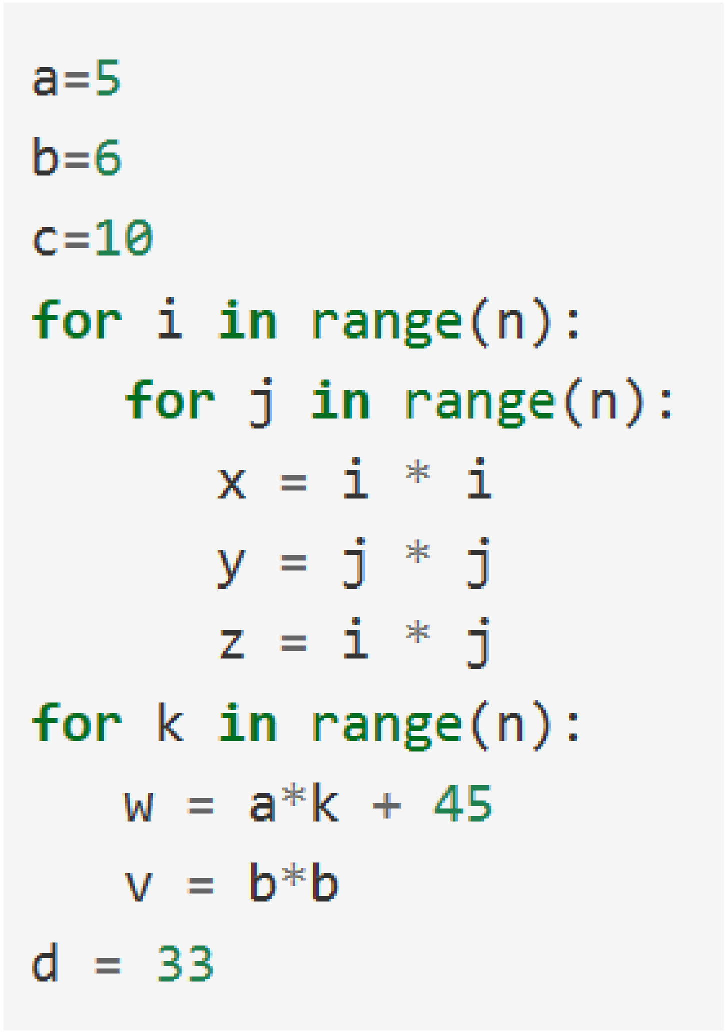
O(1)

O(n)

O(n^2)

The largest run time will be the Big-O notation: O(n^2)

Big-O notation: exercise 2



```
a=5
b=6
c=10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b
d = 33
```

$O(n^2)$

$O(n)$

The largest run time will be the Big-O notation: $O(n^2)$

SWINBURNE UNIVERSITY OF TECHNOLOGY

What is computable?

- Computation is usually modeled as a mapping from inputs to outputs, carried out by a "formal machine", or program, which processes its input in a sequence of steps.
- An "effectively computable" function is one that can be computed in a finite amount of time using finite resources.

Halting Problem

- A problem that cannot be solved by any machine in finite time (or any equivalent formalism) is called uncomputable.
- An uncomputable problem cannot be solved by any real computer.
- **The Halting Problem:**
  - Given a program and its input, determine whether the program will complete or run forever.
  - The Halting Problem is provably uncomputable – which means that it cannot be solved in practice.

```
green = ON
red = amber = OFF
while(true){
   amber = ON; green =
   OFF;
   wait 10 seconds;
   red = ON; amber = OFF;
   wait 40 seconds;
   green = ON; red = OFF;
}
```

Algorithmic Patterns

- Direct solution strategies:
  - Brute force and greedy algorithms
- Backtracking strategies:
  - Simple backtracking and branch-and-bound algorithms
- Top-down solution strategies:
  - Divide-and-conquer algorithms
- Bottom-up solution strategies:
  - Dynamic programming
- Randomized strategies:
  - Monte Carlo algorithms

End of Revision 2

All the best for your finals! :D