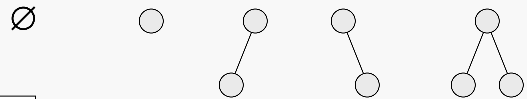
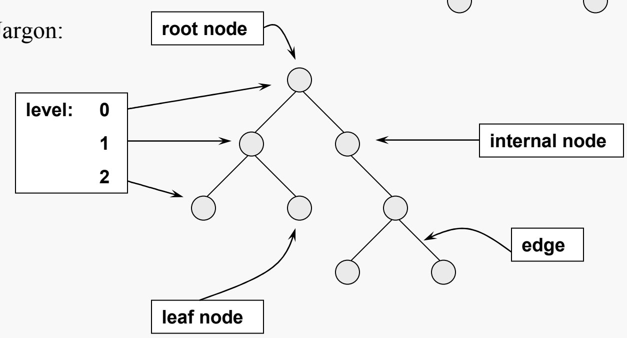
Binary Trees

A binary tree is either empty, or it consists of a node called the root together with two binary trees called the left subtree and the right subtree of the root, which are disjoint from each other and from the root.

For example:



Jargon:

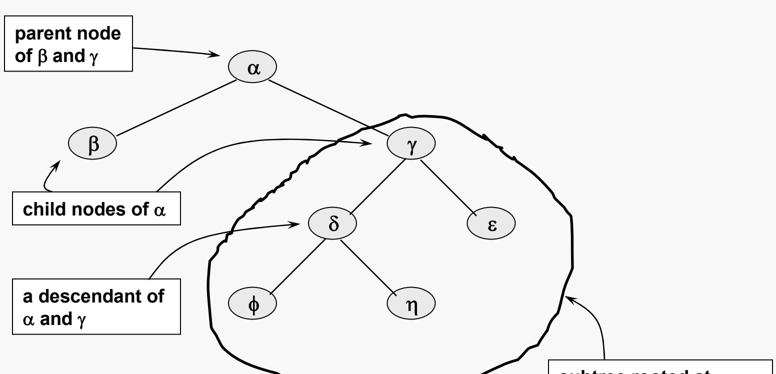


Binary Tree Node Relationships

A binary tree node may have 0, 1 or 2 child nodes.

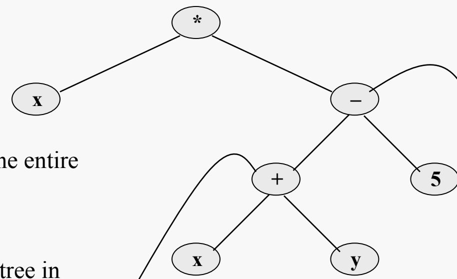
A path is a sequence of adjacent (via the edges) nodes in the tree.

A subtree of a binary tree is either empty, or consists of a node in that tree and all of its descendent nodes.



Quick Application: Expression Trees

A binary tree may be used to represent an algebraic expression:



Each subtree represents a part of the entire expression…

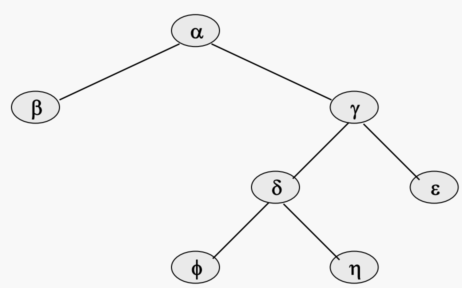
If we visit the nodes of the binary tree in the correct order, we will construct the algebraic expression:

$$x \times \underbrace{\left(\overbrace{(x+y)} - 5\right)}$$

Traversals

A <u>traversal</u> is an algorithm for visiting some or all of the nodes of a binary tree in some defined order.

A traversal that visits every node in a binary tree is called an <u>enumeration</u>.



<u>preorder</u>: visit the node, then the left subtree, then the right subtree

$\boxed{\alpha\ \beta\ \gamma\ \delta\ \phi\ \eta\ \varepsilon}$

<u>postorder</u>: visit the left subtree, then the right subtree, and then the node

$\boxed{\beta\ \phi\ \eta\ \delta\ \varepsilon\ \gamma\ \alpha}$

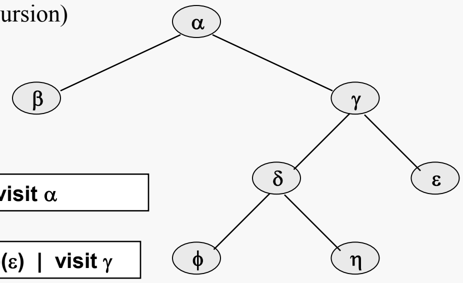
<u>inorder</u>: visit the left subtree, then the node, then the right subtree

$\boxed{\beta\ \alpha\ \phi\ \delta\ \eta\ \gamma\ \varepsilon}$

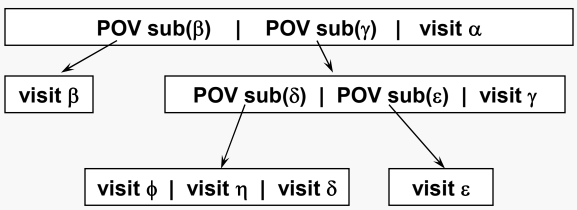
Postorder Traversal Details

Consider the postorder traversal from a recursive perspective:

postorder: postorder visit the left subtree,
postorder visit the right subtree,
then visit the node (no recursion)



If we start at the root:

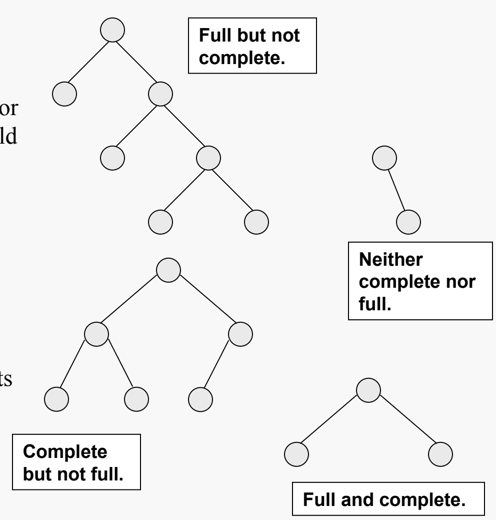


Full and Complete Binary Trees

Here are two important types of binary trees. Note that the definitions, while similar, are logically independent.

Definition: a binary tree T is *full* if each node is either a leaf or possesses exactly two child nodes.

Definition: a binary tree T with n levels is *complete* if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.



Full Binary Tree Theorem

Theorem: Let T be a nonempty, full binary tree Then:

(a) If T has I internal nodes, the number of leaves is $L = I + 1$.

(b) If T has I internal nodes, the total number of nodes is $N = 2I + 1$.

(c) If T has a total of N nodes, the number of internal nodes is $I = (N - 1)/2$.

(d) If T has a total of N nodes, the number of leaves is $L = (N + 1)/2$.

(e) If T has L leaves, the total number of nodes is $N = 2L - 1$.

(f) If T has L leaves, the number of internal nodes is $I = L - 1$.

Basically, this theorem says that the number of nodes N, the number of leaves L, and the number of internal nodes I are related in such a way that if you know any one of them, you can determine the other two.

Proof of Full Binary Tree Theorem

proof of (a): We will use induction on the number of internal nodes, I. Let S be the set of all integers $I \geq 0$ such that if T is a full binary tree with I internal nodes then T has $I + 1$ leaf nodes.

For the base case, if $I = 0$ then the tree must consist only of a root node, having no children because the tree is full. Hence there is 1 leaf node, and so $0 \in S$.

Now suppose that for some integer $K \geq 0$, every I from 0 through K is in S. That is, if T is a nonempty binary tree with I internal nodes, where $0 \leq I \leq K$, then T has $I + 1$ leaf nodes.

Let T be a full binary tree with $K + 1$ internal nodes. Then the root of T has two subtrees L and R; suppose L and R have I_L and I_R internal nodes, respectively. Note that neither L nor R can be empty, and that every internal node in L and R must have been an internal node in T, and T had one additional internal node (the root), and so $K + 1 = I_L + I_R + 1$.

Now, by the induction hypothesis, L must have I_L+1 leaves and R must have I_R+1 leaves. Since every leaf in T must also be a leaf in either L or R, T must have $I_L + I_R + 2$ leaves.

Therefore, doing a tiny amount of algebra, T must have $K + 2$ leaf nodes and so $K + 1 \in S$. Hence by Mathematical Induction, $S = [0, \infty)$.

QED

Limit on the Number of Leaves

Theorem: Let T be a binary tree with λ levels. Then the number of leaves is at most $2^{\lambda-1}$.

proof: We will use strong induction on the number of levels, λ. Let S be the set of all integers $\lambda \geq 1$ such that if T is a binary tree with λ levels then T has at most $2^{\lambda-1}$ leaf nodes.

For the base case, if $\lambda = 1$ then the tree must have one node (the root) and it must have no child nodes. Hence there is 1 leaf node (which is $2^{\lambda-1}$ if $\lambda = 1$), and so $1 \in S$.

Now suppose that for some integer $K \geq 1$, all the integers 1 through K are in S. That is, whenever a binary tree has M levels with $M \leq K$, it has at most 2^{M-1} leaf nodes.

Let T be a binary tree with K + 1 levels. If T has the maximum number of leaves, T consists of a root node and two nonempty subtrees, say S_1 and S_2. Let S_1 and S_2 have $M_1$and M_2 levels, respectively. Since M_1 and M_2 are between 1 and K, each is in S by the inductive assumption. Hence, the number of leaf nodes in S_1 and S_2 are at most 2^{K-1} and 2^{K-1}, respectively. Since all the leaves of T must be leaves of S_1 or of S_2, the number of leaves in T is at most $2^{K-1} + 2^{K-1}$ which is 2^K. Therefore, K + 1 is in S.

Hence by Mathematical Induction, $S = [1, \infty)$.

QED

More Useful Facts

Theorem: Let T be a binary tree. For every $k \geq 0$, there are no more than 2^k nodes in level k.

Theorem: Let T be a binary tree with λ levels. Then T has no more than $2^\lambda - 1$ nodes.

Theorem: Let T be a binary tree with N nodes. Then the number of levels is at least $\lceil \log (N + 1) \rceil$.

Theorem: Let T be a binary tree with L leaves. Then the number of levels is at least $\lceil \log L \rceil + 1$.

Binary Tree Representation

The natural way to think of a binary tree is that it consists of nodes (objects) connected by edges (pointers). This leads to a design employing two classes:

- binary tree class — to encapsulate the tree and its operations
- binary node class — to encapsulate the data elements, pointers and associated operations.

Each should be a template, for generality.

The node class may handle all direct accesses of the pointers and data element, or allow its client (the tree) free access.

The tree class may maintain a sense of a current location (node) and must provide all the high-level functions, such as searching, insertion and deletion.

Many implementations use a struct type for the nodes. The motivation is generally to make the data elements and pointers public and hence to simplify the code, at the expense of automatic initialization via a constructor.

A Binary Node Interface

Here's a possible interface for a binary tree node:

```
template <typename T> class BinNodeT {
public:
   T           Element;
   BinNodeT<T>* Left;
   BinNodeT<T>* Right;

   BinNodeT();
   BinNodeT(const T& D, BinNodeT<T>* L = NULL,
                        BinNodeT<T>* R = NULL);

   bool isLeaf() const;
   ~BinNodeT();
};
```

Binary tree object can access node data members directly.

Useful for tree navigation.

The design here leaves the data members public to simplify the implementation of the encapsulating binary tree class; due to that encapsulation there is no concern that client code will be able to take advantage of this decision.

The data element is stored by pointer to provide for storing dynamically allocated elements, and elements from an inheritance hierarchy. Converting to direct storage is relatively trivial.

A Binary Tree Class Interface

Here's a possible interface for a binary tree class. It's not likely to be put to any practical use, just a proof of concept.

```
template <typename T> class BinaryTreeT {
protected:
   BinNodeT<T>* Root;

   unsigned int SizeHelper(BinNodeT<T>* sRoot) const;
   unsigned int HeightHelper(BinNodeT<T>* sRoot) const;
   bool InsertHelper(const T& D, BinNodeT<T>* sRoot);
   bool DeleteHelper(const T& D, BinNodeT<T>* sRoot);
   void TreeCopyHelper(BinNodeT<T>* TargetRoot,
                            BinNodeT<T>* SourceRoot);
   T* const FindHelper(const T& toFind, BinNodeT<T>* sRoot);
   const T* const FindHelper(const T& toFind, BinNodeT<T>* sRoot) const;
   void DisplayHelper(BinNodeT<T>* sRoot, std::ostream& Out,
                                unsigned int Level);
   void ClearHelper(BinNodeT<T>* sRoot);

public:
   BinaryTreeT();
   BinaryTreeT(const T& D);
   BinaryTreeT(const BinaryTreeT<T>& Source);
   BinaryTreeT<T>& operator=(const BinaryTreeT<T>& Source);
// . . . continued . . .
```

Recursive "helper" functions — each has a corresponding public function.

Virtual functions are used to encourage subclasses.

A Binary Tree Class Interface


```
// . . . continued . . .

   bool Insert(const T& D);
   bool Delete(const T& D);
   T* const Find(const T& D);
   const T* const Find(const T& D) const;

   unsigned int Size() const;
   unsigned int Height() const;
   void Display(std::ostream& Out);
   void Clear();

   ~BinaryTreeT();
};
```

Data insertion/search functions.

Reporters, a display function, and a clear function.

The interface is somewhat incomplete since it's not really a serious class… as we will see, specialized binary trees are what we really want.

Still, there are some useful things we can learn from even an incomplete version…

Finding a Data Element

```
template <typename T>
T* const BinaryTreeT<T>::Find(const T& toFind) {

   if (Root == NULL) return NULL;

   return (FindHelper(toFind, Root));
}

template <typename T>
T* const BinaryTreeT<T>::FindHelper(const T& toFind, BinNodeT<T>* sRoot) {

   T* Result;
   if (sRoot == NULL) return NULL;

   if (sRoot->Element == toFind) {
       Result = &(sRoot->Element);
   }
   else {
      Result = FindHelper(toFind, sRoot->Left);
      if (Result == NULL)
         Result = FindHelper(toFind, sRoot->Right);
   }
   return Result;
}
```

Nonrecursive interface function for client…

… uses a recursive protected function to do almost all the work.

Why?

Which traversal is used here?

Why not use a different traversal instead?

Why is `const` used on the return value??

Clearing the Tree

Similar to the class destructor, `Clear()` causes the deallocation of all the tree nodes and the resetting of `Root` and `Current` to indicate an empty tree.

```
template <typename T>
void BinaryTreeT<T>::Clear() {

   ClearHelper(Root);
   Root = NULL;
}

template <typename T>
void BinaryTreeT<T>::ClearHelper(BinNodeT<T>* sRoot) {

   if (sRoot == NULL) return;

   ClearHelper(sRoot->Left);
   ClearHelper(sRoot->Right);
   delete sRoot;
}
```

Which traversal is used here?

Why not use a different traversal instead?

Inorder Printing


```
template <typename T>
void BinaryTreeT<T>::Display(ostream& Out) {

   if (Root == NULL) {
      Out << "tree is empty" << endl;
      return;
   }
   DisplayHelper(Root, Out, 0);
}

template <typename T>
void BinaryTreeT<T>::DisplayHelper(BinNodeT<T>* sRoot, ostream& Out,
                                   unsigned int Level) {

   if (sRoot == NULL) return;

   DisplayHelper(sRoot->Left, Out, Level + 1);

   if ( Level > 0 )  Out << setw(3*Level) << ' ';
   Out << sRoot->Element << endl;

   DisplayHelper(sRoot->Right, Out, Level + 1);
}
```

```
Inorder traversal:
      3            left
   1
      4
0
      5
   2
         7
      6
         8         right
```

QTP: Could we reverse the sides of the printed tree?

Summary of Implementation

The implementation described here is primarily for illustration. The full implementation has been tested, but not thoroughly.

As we will see in the next chapter, general binary trees are not often used in applications. Rather, specialized variants are derived from the notion of a general binary tree, and THOSE are used.

Before proceeding with that idea, we need to establish a few facts regarding binary trees.

Warning: the binary tree classes given in this chapter are intended for instructional purposes. The given implementation contains a number of known flaws, and perhaps some unknown flaws as well. *Caveat emptor.*