
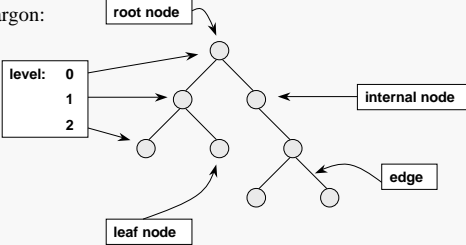


Binary Trees General Binary Trees 1

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.

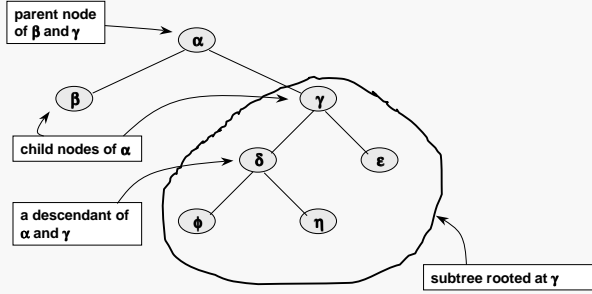
For example: \emptyset 

Jargon: 

Computer Science Dept. VU Tech. July 2000 Data Structures & File Management ©2000 McQuain WD

Binary Tree Node Relationships General Binary Trees 2

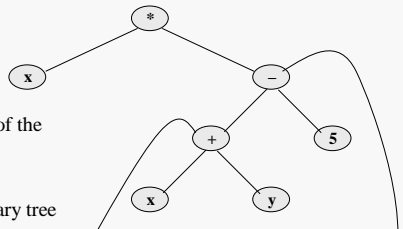
A binary tree node may have 0, 1 or 2 **child** nodes.
 A **path** is a sequence of adjacent (via the edges) nodes in the tree.
 A **subtree** of a binary tree is either empty, or consists of a node in that tree and all of its descendent nodes.



Computer Science Dept. VU Tech. July 2000 Data Structures & File Management ©2000 McQuain WD

Quick Application: Expression Trees General Binary Trees 3

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



Each subtree represents a part of the entire expression...

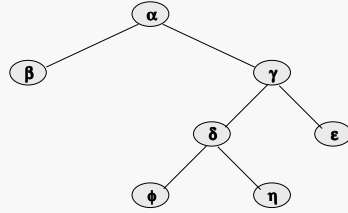
If we visit the nodes of the binary tree in the correct order, we will construct the algebraic expression:

$$x \times ((x + y) - 5)$$

Computer Science Dept. VU Tech. July 2000 Data Structures & File Management ©2000 McQuain WD

Traversals General Binary Trees 4

A **traversal** 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 **enumeration**.



preorder: visit the node, then the left subtree, then the right subtree
 $\alpha \beta \gamma \delta \phi \eta \epsilon$

postorder: visit the left subtree, then the right subtree, and then the node
 $\beta \phi \eta \delta \epsilon \gamma \alpha$

inorder: visit the left subtree, then the node, then the right subtree
 $\beta \alpha \phi \delta \eta \gamma \epsilon$

Computer Science Dept. VU Tech. July 2000 Data Structures & File Management ©2000 McQuain WD

Postorder Traversal Details General Binary Trees 5

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:

```

POV sub(beta) | POV sub(gamma) | visit alpha
  |
  |
visit beta | POV sub(delta) | POV sub(epsilon) | visit gamma
  |
  |
visit phi | visit eta | visit delta | visit epsilon
    
```

Computer Science Dept. Va Tech, July 2000 Data Structures & File Management ©2000 McQuain WD

Binary Tree Representation General Binary Trees 6

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 should handle all direct accesses of the pointers.

The tree class must maintain a sense of a current location (node) and 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 access restriction. Kruse/Ryba take this approach... the code to be presented here does not.

Computer Science Dept. Va Tech, July 2000 Data Structures & File Management ©2000 McQuain WD

A Binary Node Class Interface General Binary Trees 7

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

```

template <class Data> class BinNodeT {
private:
    Data      Element;
    BinNodeT<Data>* Left;
    BinNodeT<Data>* Right;
public:
    BinNodeT();
    BinNodeT(Data newData, BinNodeT<Data>* newLeft = NULL,
              BinNodeT<Data>* newRight = NULL);
    void setData(Data newData);
    Data getData() const;
    void setLeft(BinNodeT<Data>* newLeft);
    void setRight(BinNodeT<Data>* newRight);
    BinNodeT<Data>* getLeft() const;
    BinNodeT<Data>* getRight() const;
    bool isLeaf() const;
    ~BinNodeT();
};
    
```

Client can retrieve a copy of data, modify it, and reinsert it, but cannot edit it *in situ*.

Needed for tree navigation.

Computer Science Dept. Va Tech, July 2000 Data Structures & File Management ©2000 McQuain WD

A Binary Tree Class Interface General Binary Trees 8

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 <class Data> class BinaryTreeT {
protected:
    BinNodeT<Data>* Root;
    BinNodeT<Data>* Current;
    int getSizeHelper(BinNodeT<Data>* sRoot) const;
    int getHeightHelper(BinNodeT<Data>* sRoot) const;
    bool InsertHelper(Data D, BinNodeT<Data>* sRoot);
    bool DeleteHelper(Data D, BinNodeT<Data>* sRoot);
    void TreeCopyHelper(BinNodeT<Data>* TargetRoot,
                       BinNodeT<Data>* SourceRoot);
    void inOrderHelper(BinNodeT<Data>* sRoot, void (*visit)(Data D));
    bool moveToDataHelper(Data toFind, BinNodeT<Data>* sRoot);
    void InOrderPrintHelper(BinNodeT<Data>* sRoot, ostream& Out, int Level);
    void ClearHelper(BinNodeT<Data>* sRoot);
    // ... continues ...
};
    
```

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

Computer Science Dept. Va Tech, July 2000 Data Structures & File Management ©2000 McQuain WD

A Binary Tree Class Interface General Binary Trees 9

```

// . . . continued
public:
    BinaryTreeT();
    BinaryTreeT(Data newData);
    BinaryTreeT(const BinaryTreeT<Data>& Source);
    BinaryTreeT<Data>& operator=(const BinaryTreeT<Data>& Source);

    int  getSize() const;
    int  getHeight() const;

    Data getCurrentData() const;
    bool setCurrentData(Data D);

    bool moveToData(Data toFind);
    bool atLeaf() const;

    bool Insert(Data D);
    bool Delete(Data D);

    void InOrderPrint(ostream& Out);
    void InOrder(void (*visit)(Data D));

    void Clear();
    ~BinaryTreeT();
};
    
```

Data access functions.

Navigation functions.

Data insertion/deletion functions.

Auxiliary functions.

"Cleanup" functions.

Computer Science Dept VU Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Finding a Data Element General Binary Trees 10

```

template <class Data>
bool BinaryTreeT<Data>::moveToData(Data toFind) {
    if (Root == NULL) return false;

    return (moveToDataHelper(toFind, Root));
}

template <class Data>
bool BinaryTreeT<Data>::moveToDataHelper(Data toFind,
                                        BinNodeT<Data>* sRoot) {
    if (sRoot == NULL) return false;

    if (sRoot->getData() == toFind) {
        Current = sRoot;
        return true;
    }

    return ( moveToDataHelper(toFind, sRoot->getLeft()) ||
            moveToDataHelper(toFind, sRoot->getRight()) );
}
    
```

Nonrecursive interface function for client...

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

Here a preorder traversal is used. Why not use postorder instead?

Note we must allow for checking BOTH subtrees; however, short-circuiting will eliminate the second call if it's unnecessary.

Computer Science Dept VU Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Clearing the Tree General Binary Trees 11

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 <class Data>
void BinaryTreeT<Data>::Clear() {
    ClearHelper(Root);
    Root = Current = NULL;
}

template <class Data>
void BinaryTreeT<Data>::ClearHelper(BinNodeT<Data>* sRoot) {
    if (sRoot == NULL) return;

    ClearHelper(sRoot->getLeft());
    ClearHelper(sRoot->getRight());
    delete sRoot;
}
    
```

Applies a postorder traversal so subtrees are deleted before their root nodes.

Computer Science Dept VU Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Inorder Printing General Binary Trees 12

```

template <class Data>
void BinaryTreeT<Data>::InOrderPrint(ostream& Out) {
    if (Root == NULL) {
        Out << "tree is empty" << endl;
        return;
    }
    InOrderPrintHelper(Root, Out, 0);
}

template <class Data>
void BinaryTreeT<Data>::InOrderPrintHelper(BinNodeT<Data>* sRoot,
                                           ostream& Out, int Level) {
    if (sRoot == NULL) return;

    InOrderPrintHelper(sRoot->getLeft(), Out, Level + 1);

    for (int L = 0; L < Level; L++)
        Out << "  ";
    Out << sRoot->getData() << endl;

    InOrderPrintHelper(sRoot->getRight(), 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?

Computer Science Dept VU Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Inserting a Data Element General Binary Trees 13

Insertion poses a question: how do we decide where to put the new element. Since there's no defined data ordering for a (general) binary tree, we can do whatever we like.

The implementation given here inserts the data element into the shorter subtree of the root (recursively of course).

```

template <class Data>
bool BinaryTree<Data>::Insert(Data D) {
    if (Root == NULL) {
        BinNodeT<Data>* Temp = new BinNodeT<Data>(D);
        if (Temp == NULL) return false;
        Root = Current = Temp;
        return true;
    }
    return InsertHelper(D, Root);
}
    
```

The public function handles insertion at the root (if the tree is empty).

The subtlety here is that `InsertHelper()` can't modify the value of `Root`, so the interface function `Insert()` must handle that case. Why...?

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Insert Helper Function General Binary Trees 14

`InsertHelper()` must pass itself a pointer to the root node of the correct subtree. Since that's a private node data member, only a copy of it can be passed — be clear on this point, it would be worse than useless to pass the second parameter by reference.

```

template <class Data>
bool BinaryTree<Data>::InsertHelper(Data D, BinNodeT<Data>* sRoot) {
    if (sRoot->getLeft() == NULL) {
        BinNodeT<Data>* Temp = new BinNodeT<Data>(D);
        if (Temp == NULL) return false;
        sRoot->setLeft(Temp);
        return true;
    }
    else if (sRoot->getRight() == NULL) {
        BinNodeT<Data>* Temp = new BinNodeT<Data>(D);
        if (Temp == NULL) return false;
        sRoot->setRight(Temp);
        return true;
    }
    // . . . continues . . .
}
    
```

If the subtree root has an "empty" child, we'll just put the new value there.

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Insert Helper Function General Binary Trees 15

```

// . . . continued . . .
int lHeight = getHeightHelper(sRoot->getLeft());
int rHeight = getHeightHelper(sRoot->getRight());

if (lHeight <= rHeight) {
    return (InsertHelper(D, sRoot->getLeft()));
}
else {
    return (InsertHelper(D, sRoot->getRight()));
}
    
```

If the subtree root has an two children, we'll just put the new value down in the shorter sub-subtree.

There's no good reason to define insertion this way, it's just for an illustration of the logic involved in a typical binary tree manipulation.

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Deleting a Data Element General Binary Trees 16

Deletion involves cases.

- Removing a leaf node is trivial, just set the relevant child pointer in the parent node to NULL.
- Removing an internal node poses a problem: what do we replace that node with?

The question hinges on how many children the targeted node has.

If only one, we can just make the parent of the targeted node point to that child node, and then delete the targeted node.

What if the targeted node has two children?

We resolve the question (somewhat arbitrarily) by swapping with the child whose subtree is taller, and then (recursively) deleting the targeted value from that subtree.

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Sliding the Target Value Down

General Binary Trees 17

For example, suppose we want to delete the value γ from the tree below:

After searching for the value γ , we will swap it with the data element in the root of the shorter subtree...

...then we target the node that now contains γ for deletion. Eventually, γ MUST "slide down" to a leaf node, which can be trivially deleted.

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Delete Implementation

General Binary Trees 18

So, how do we accomplish this?

The public function handles deletion at the root.

```

template <class Data>
bool BinaryTreeT<Data>::Delete(Data D) {

    if (Root == NULL) return false;

    if (Root->getData() == D) { // deletion of tree root node
        BinNodeT<Data>* Temp = Root;
        if (Root->getLeft() == NULL) { // no left child
            Root = Root->getRight(); // right child, if any, is new root
            delete Temp;
            return true;
        }
        else if (Root->getRight() == NULL) { // left child, no right child
            Root = Root->getLeft(); // left child, is new root
            delete Temp;
            return true;
        }
    }
    // . . . continues for Root node with two children . . .
    
```

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Delete Implementation

General Binary Trees 19

```

// . . . continued . . .

else { // both left and right children
    // find taller subtree
    int lHeight = getHeightHelper(Root->getLeft());
    int rHeight = getHeightHelper(Root->getRight());
    if (lHeight >= rHeight) {
        Root->setData(Root->getLeft()->getData()); // swap data
        Root->getLeft()->setData(D);
        return DeleteHelper(D, Root); // delete left subtree root
    }
    else {
        Root->setData(Root->getRight()->getData()); // swap data
        Root->getRight()->setData(D);
        return DeleteHelper(D, Root); // delete right subtree root
    }
}

return DeleteHelper(D, Root);
}
    
```

If the target value isn't in the root node... call the helper function.

Ugly...

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Delete Helper Function

General Binary Trees 20

Note: if the helper function is called, we know that the targeted value is NOT in *sRoot.

```

template <class Data>
bool BinaryTreeT<Data>::DeleteHelper(Data D, BinNodeT<Data>* sRoot) {

    if (sRoot == NULL) return false;

    BinNodeT<Data>* TargetChild = NULL;
    bool Left = false;
    bool Right = false;

    // see if target value is in a child of sRoot

    if ( (sRoot->getLeft() != NULL) &&
        (sRoot->getLeft()->getData() == D) ) { // need to delete left child
        TargetChild = sRoot->getLeft();
        Left = true;
    }
    else if ( (sRoot->getRight() != NULL) &&
             (sRoot->getRight()->getData() == D) ) { // need to delete right child
        TargetChild = sRoot->getRight();
        Right = true;
    }

    // . . . continues . . .
    
```

Determine if either child of the subtree root holds the targeted value.

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Delete Helper Function General Binary Trees 21

```
// ... continued ...
if (TargetChild == NULL) { // Haven't found data value yet, so check
    // the subtrees of sRoot recursively.
    return ( DeleteHelper(D, sRoot->getLeft()) ||
            DeleteHelper(D, sRoot->getRight()) );
}

If the target value ISN'T in a child of the subtree root node... recurse.

If the target value IS in a child of the subtree root node, determine the case...
if (TargetChild->getLeft() == NULL) {
    if (Left)
        sRoot->setLeft(TargetChild->getRight());
    else
        sRoot->setRight(TargetChild->getRight());
    delete TargetChild;
    return true;
}
else if (TargetChild->getRight() == NULL) {
    if (Left)
        sRoot->setLeft(TargetChild->getLeft());
    else
        sRoot->setRight(TargetChild->getLeft());
    delete TargetChild;
    return true;
}

No left child, so just let the parent node point to the right child, if any.

Left child, but no right child, so just let the parent node point to the left child.

```

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Delete Helper Function General Binary Trees 22

```
// ... continued ...
else {
    int lHeight = getHeightHelper(TargetChild->getLeft());
    int rHeight = getHeightHelper(TargetChild->getRight());

    if (lHeight >= rHeight) {
        TargetChild->setData(TargetChild->getLeft()->getData());
        TargetChild->getLeft()->setData(D);
        return DeleteHelper(D, TargetChild);
    }
    else {
        TargetChild->setData(TargetChild->getRight()->getData());
        TargetChild->getRight()->setData(D);
        return DeleteHelper(D, TargetChild);
    }
}

Targeted node has two children. First determine which is the root of the taller subtree...

...then swap the targeted value into that child, and recurse on that subtree.

... now that was fun ... fortunately, it doesn't get any worse than this ...

```

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Summary of Implementation General Binary Trees 23

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.*

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

Full and Complete Binary Trees General Binary Trees 24

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 but not complete.

Neither complete nor full.

Complete but not full.

Full and complete.

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

General Binary Trees 25

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.

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

General Binary Trees 26

Full Binary Tree Theorem Proof

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

For the base case, if $N = 1$ then the tree must have one internal node (the root) and it must have two child nodes because the tree is full. Hence there are $N + 1$ leaf nodes, and so $1 \in S$.

Now suppose that some integer $K \geq 1$ is in S. That is, whenever a nonempty full binary tree has K internal nodes it has $K + 1$ leaf nodes.

Let T be a full binary tree with $K + 1$ internal nodes. Pick an internal node of T whose child nodes are both leaves (how do we know this is possible?), and delete both its children; call the resulting tree T'. Then T' is a nonempty full binary tree, and T' has K internal nodes; so by the inductive assumption, T' must have $K + 1$ leaf nodes. But the number of leaf nodes in T is one less than the number of leaf nodes in T' (deleting the two child nodes turned the former internal node into a leaf). Therefore, T must have $K + 2$ leaf nodes and so $K + 1 \in S$.

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

QED

The remaining parts are easily derived algebraically from (a).

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

General Binary Trees 27

Limit on the Number of Leaves

Theorem: Let T be a binary tree of 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^{M_1-1} and 2^{M_2-1} , 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

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD

General Binary Trees 28

Limit on the Number of Levels

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

proof: From the previous theorem, if T has λ levels then the number of leaves is at most $2^{\lambda-1}$. That is,

$$L \leq 2^{\lambda-1}$$

Taking logarithms of both sides yields:

$$\log L \leq \lambda - 1$$

Since λ is an integer, we may apply the ceiling function to the left side, to obtain:

$$\log L \leq \lambda - 1$$

and the final result follows immediately.

QED

Computer Science Dept Va Tech July 2000 Data Structures & File Management ©2000 McQuain WD