

Data Structures and Algorithms

BSc. In Computing

Semester 5 Lecture 5

Lecturer: Dr. Simon
McLoughlin

This Week:

Trees



- Terminology

The ADT Binary Tree

- Basic operations of the ADT Binary Tree
- General Operations of the ADT Binary Tree
- Traversals of a Binary Tree
- Possible Representations of a Binary Tree
- A Reference -based Implementation of the ADT Binary Tree
- Tree Traversals Using an Iterator

The data organisations presented in previous lectures are linear in that the items are one after the other.

- Linked List
- Stack
- Queue

The ADTs in this lecture organise data in a non-linear, hierarchical form, whereby an item can have more than one immediate successor.

In particular, this lecture and the next discusses the specifications, implementations, and relative efficiency of the **ADT Binary Tree** ...

Terminology

We use trees to represent relationships.

A tree is made up of boxes, **nodes** or vertices.

The lines between each node are called **edges**.

All trees are **hierarchical** in nature.

Intuitively, hierarchical means that a “parent -child” relationship exists between the nodes in a tree.

If an **edge** is between **node n** and **node m**,
and **node n** is above **node m** in the tree,
Then n is the **parent** of m,
m is the **child** of n.

In the diagram following, node B and C are children of node A.

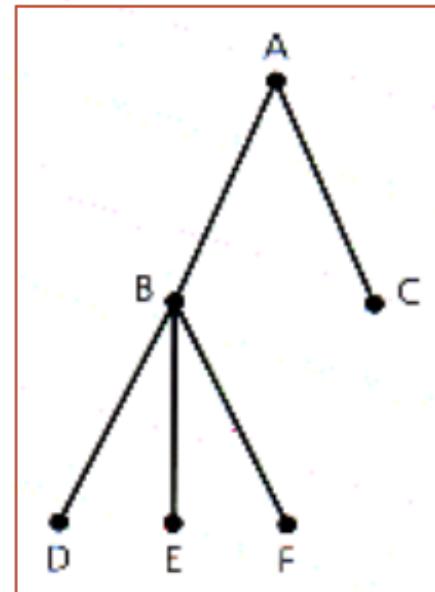
Children of the same parent, B and C are called **siblings**

Each node in a tree has at most one **parent**, and exactly one node, called a **root** of the tree, has no parent.

Node A is the root of the tree in the diagram.

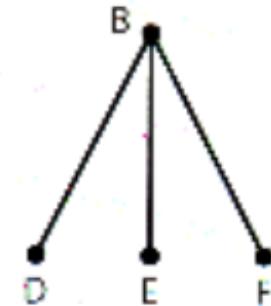
A node that has no children is called a **leaf** of the tree.

The leaves of the tree in the diagram are C, D, E, and F.



A general tree

A subtree is any node and its descendants



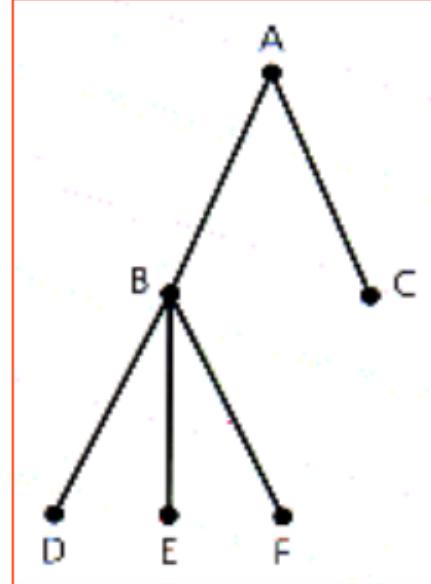
A subtree of the tree

The parent-child relationship between the nodes is generalised to the relationships **ancestor** and **descendant**

In the diagram, ...

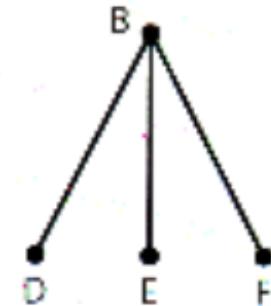
A is an ancestor of D, and D is a descendent of A.

- Not all nodes are related by the ancestor or descendent relationship.
- B and C are not so related



A general tree

A subtree is any node and its descendants



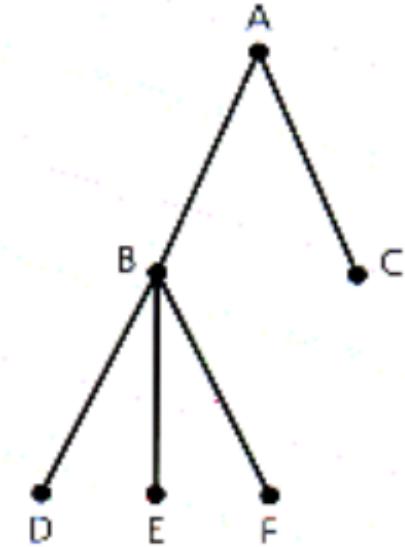
A subtree of the tree

The root of any tree is an ancestor of every node in that tree.

A subtree in a tree is any node in the tree together with all of its descendants.

A **subtree** of a node n is a **subtree rooted at a child of n**

- For example, the diagram shows a subtree of the tree in the earlier diagram.
- This has B as its root and is a subtree of the node A.



A general tree

A subtree is any node and its descendants



A subtree of the tree

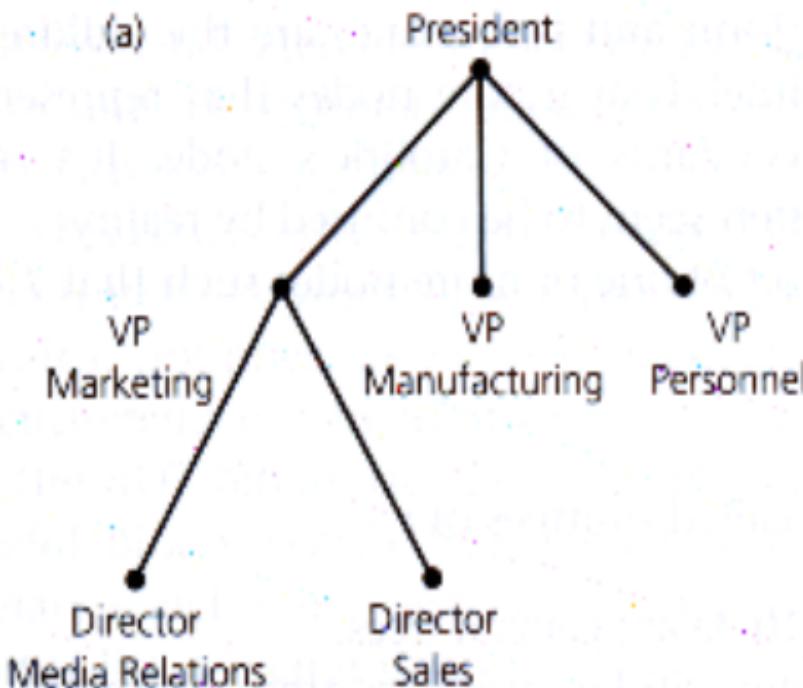
Because trees are naturally hierarchical, they can be used to represent information that is itself hierarchical.

- Organisation charts,
- Family trees

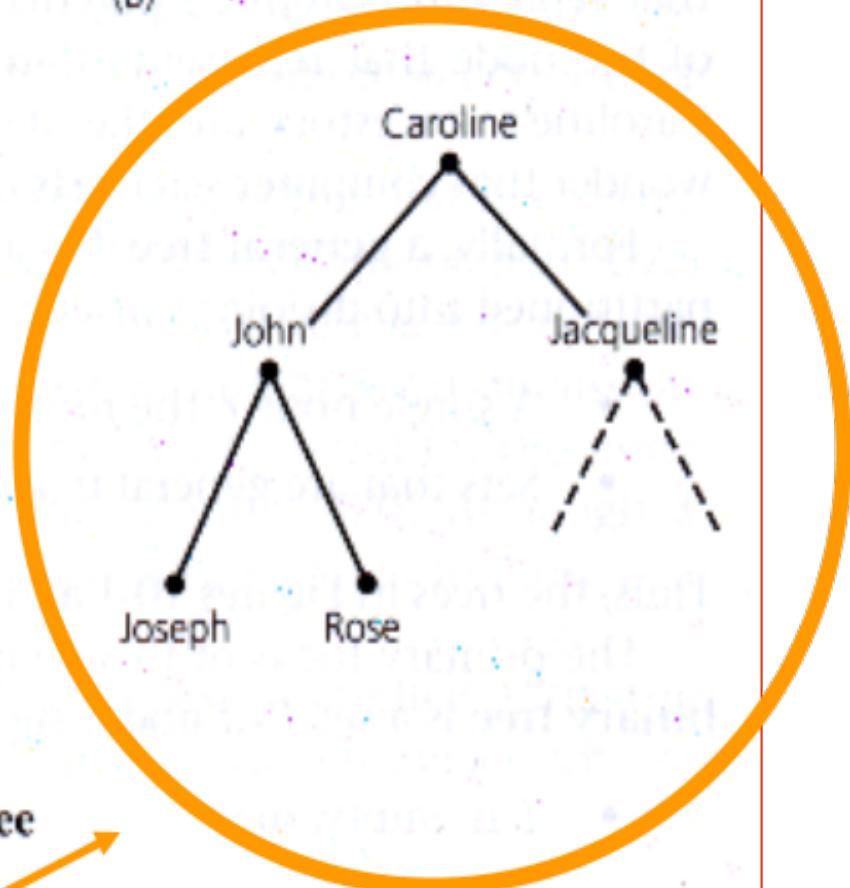
Formally a binary tree is a set T of nodes such that either:

- T is **empty**, or
 - T is **partitioned** into three disjoint subsets:
 - A single node **r**, the root
 - Two possibly empty sets that are binary trees,
called **left and right subtrees** of **r**
-

(a)



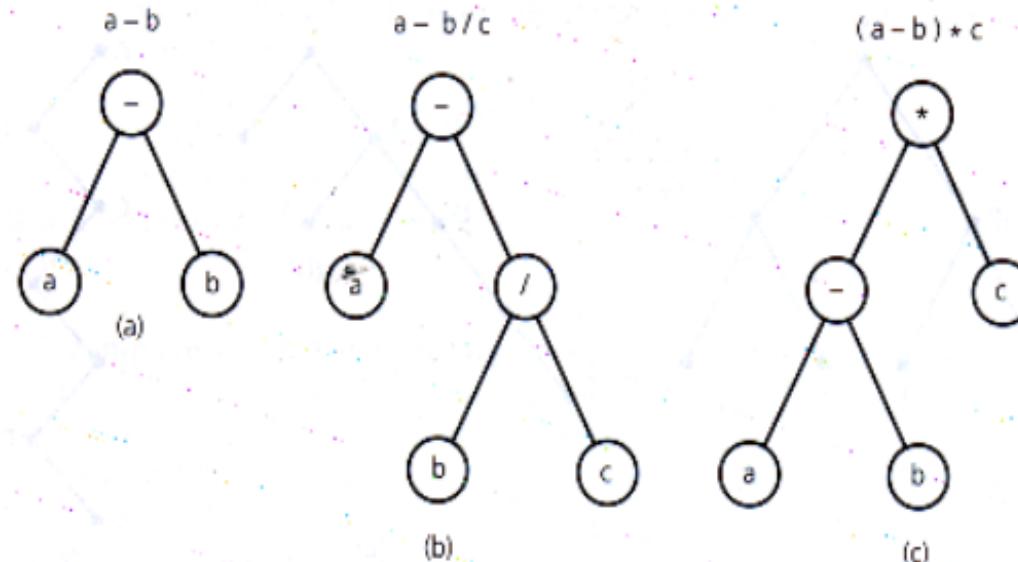
(b)



(a) An organization chart; (b) a family tree

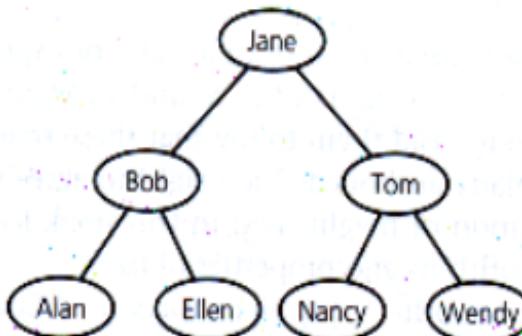
Each node in a **binary tree** has no more than two children.

An example of how we can represent data in trees, in hierarchical form.



Binary trees that represent algebraic expressions

**A binary search tree
of names**



Binary trees can represent algebraic expressions that involve the binary operators +, -, * and /.

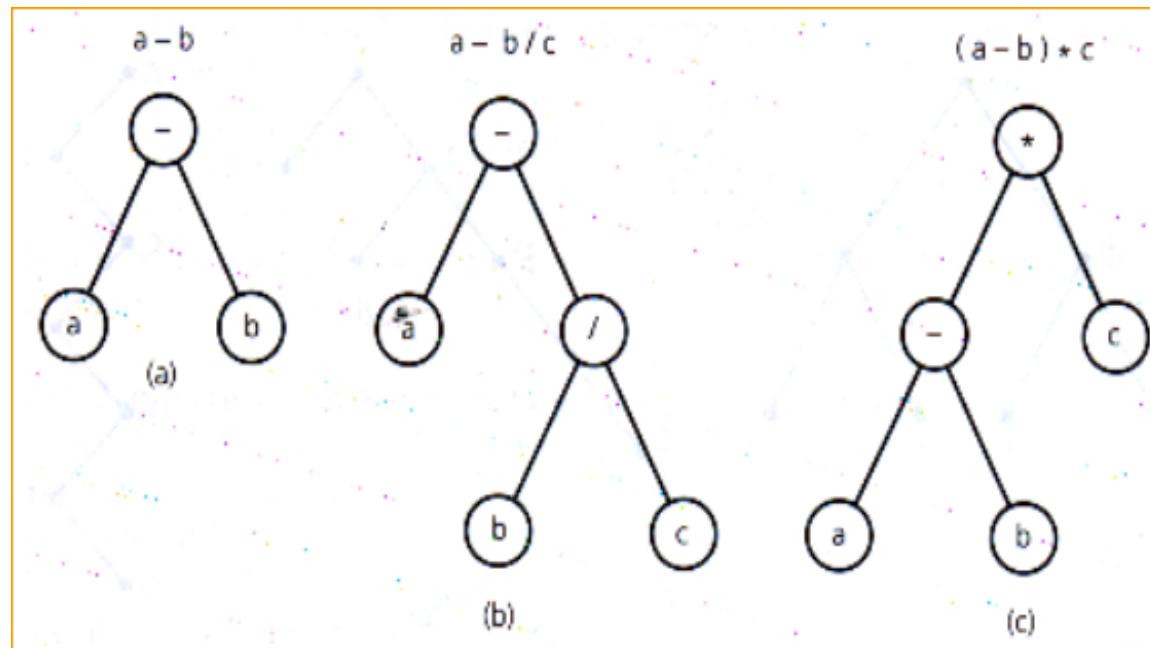
To represent an expression such as $a - b$, we place the operator in the root and the operands in the left and right children of the root, respectively.

The (b) figure represents the expression $a - b/c$ and a subtree represents the expression b/c .

A similar situation exists in (c), which represents $(a - b) * c$.

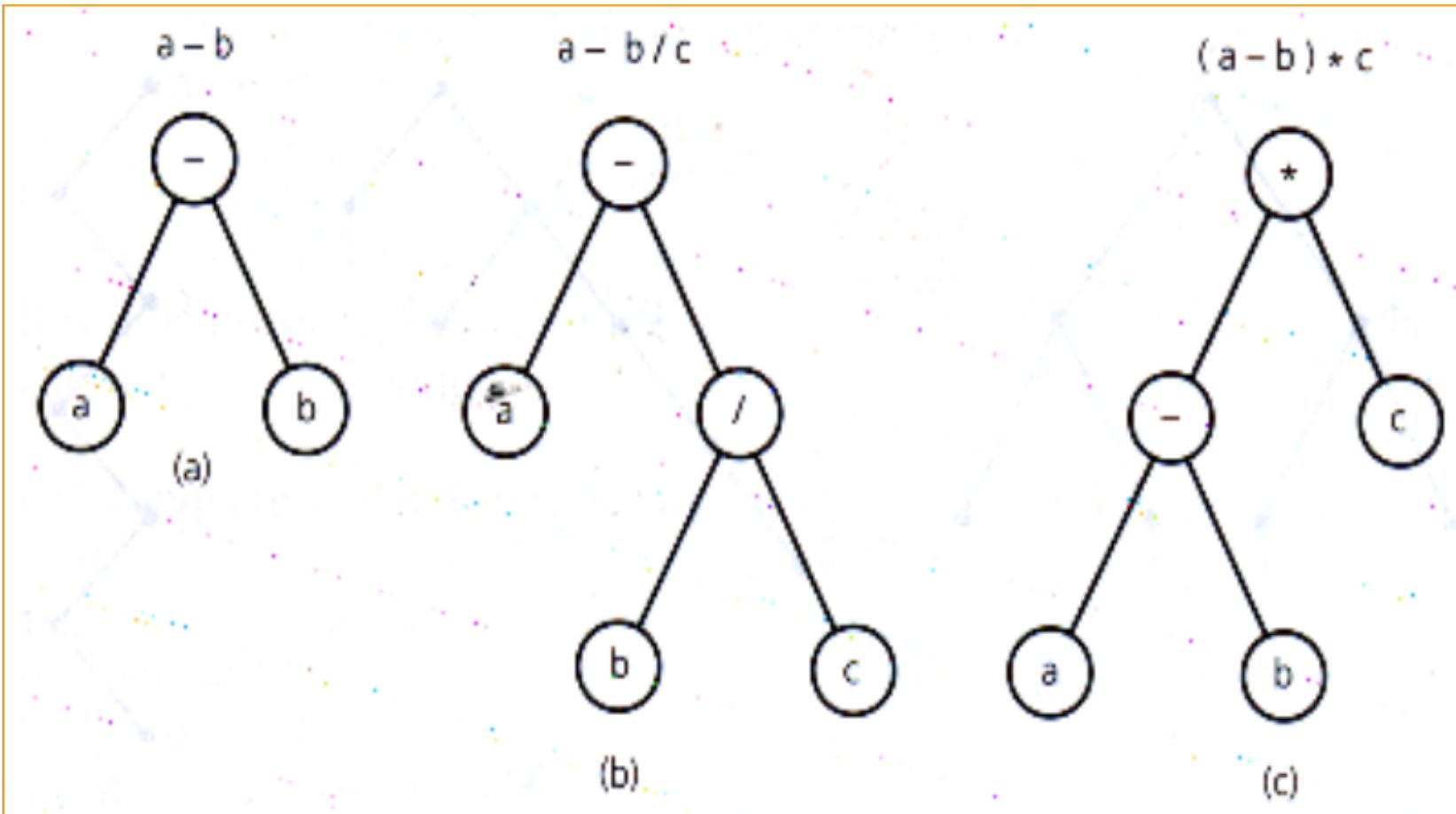
The leaves of these trees contain the expressions' operands, while other tree nodes contain the operators.

Notice that brackets/parentheses do not appear in the trees.



The binary tree provides a hierarchy for the operations.

- That is, the tree structure provides a clear, unambiguous order for evaluating an expression.



The nodes of a tree typically contain values.

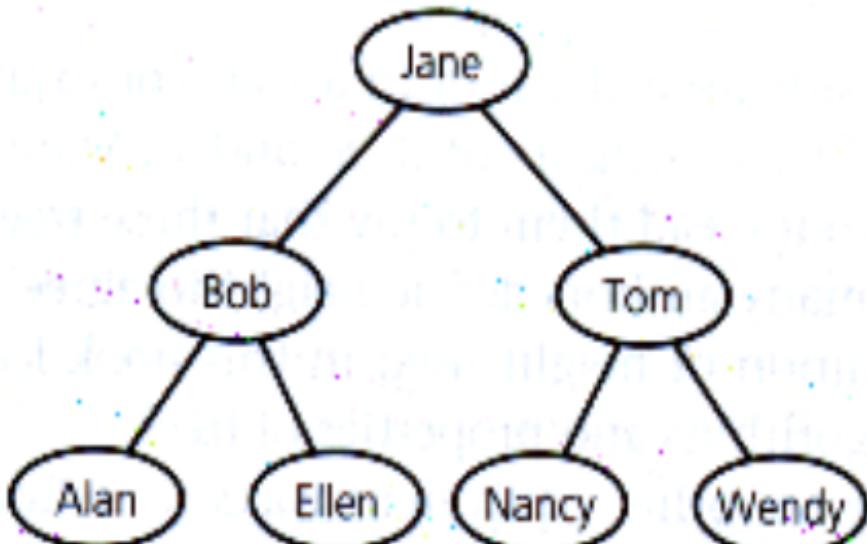
A binary search tree is a binary tree that is sorted according to the values in its nodes.

For each node **n**, ...

a binary search tree satisfies the following three properties :

- The value of **n** is greater than all values in its left subtree T_L
- The value of **n** is less than all values in its right subtree T_R
- Both T_L and T_R are binary search trees

A binary search tree of names



The diagram illustrates an example of a binary search tree.

The data is organised in a way that facilitates searching it for a particular data item.

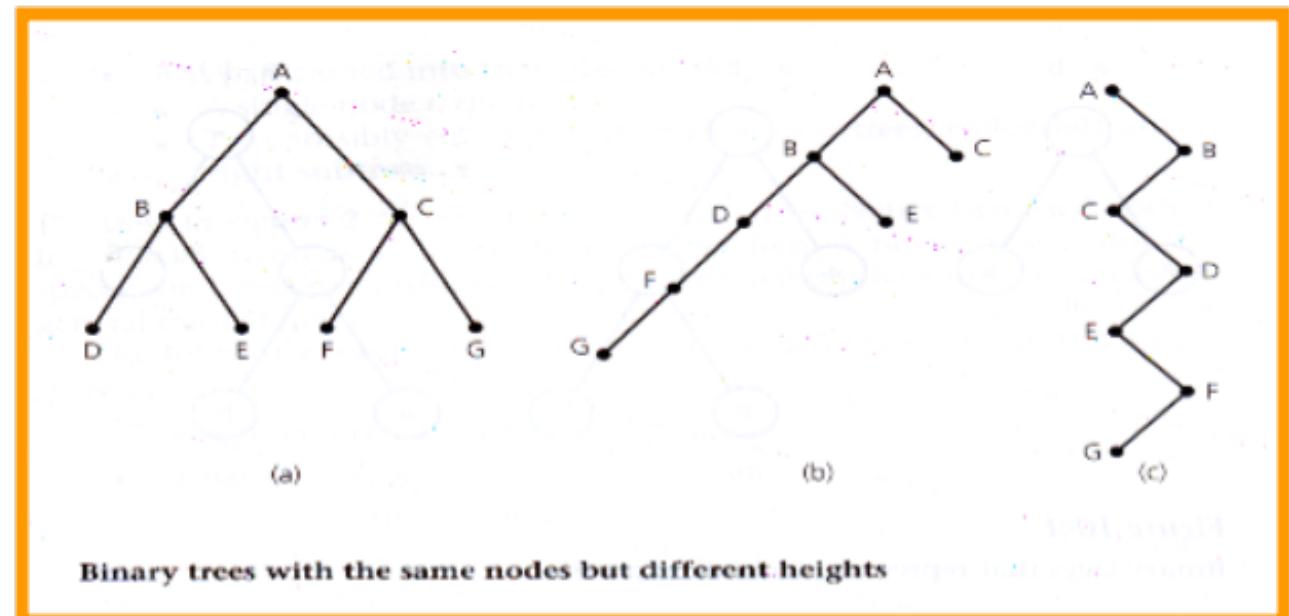
The Height of Trees

Trees come in many shapes.

Although the binary trees in the diagram following all contain the same nodes, their structures are quite different.

Although each of these trees contains seven nodes, some are “taller” than others.

The height of any tree is the number of nodes on the longest path from the root to a leaf.

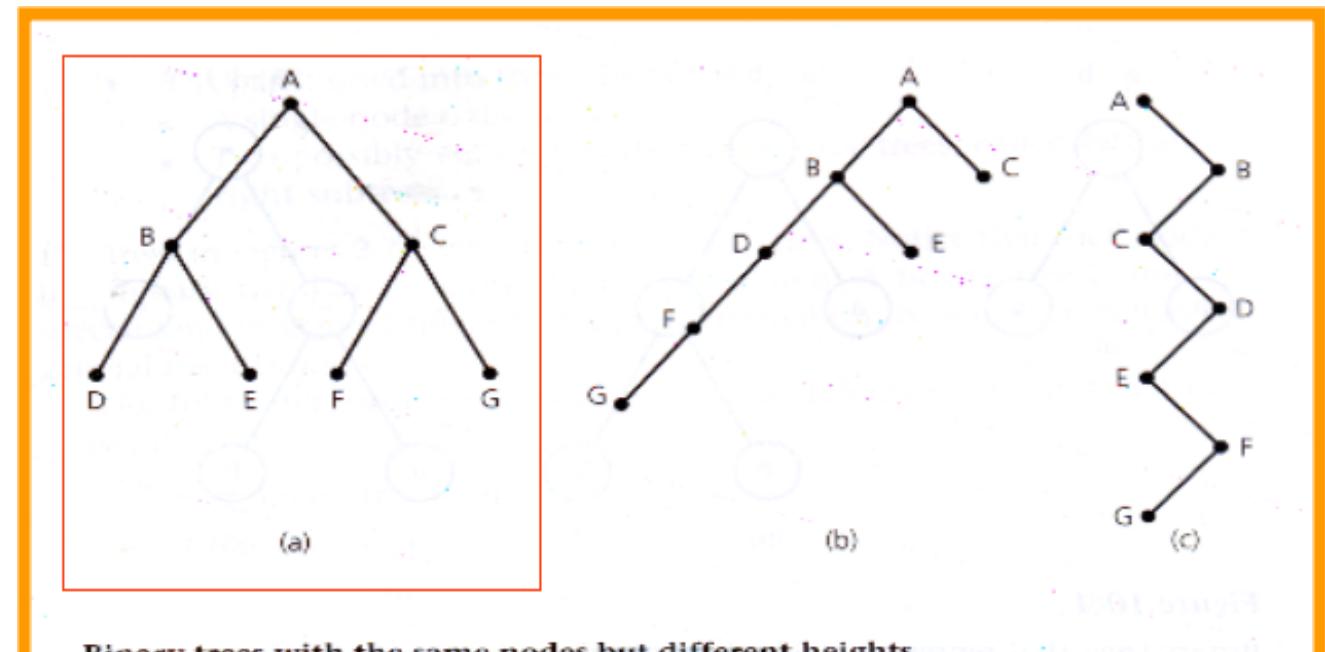


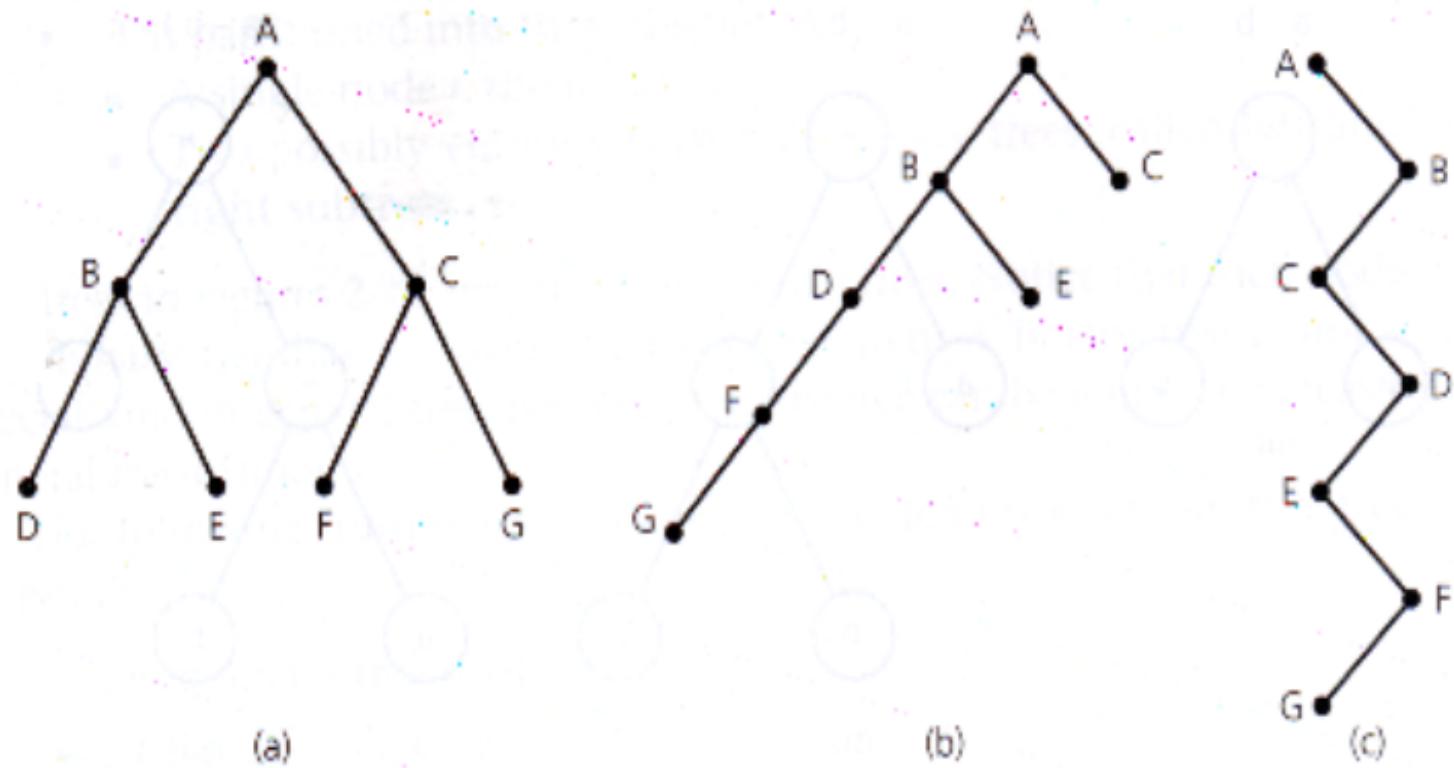
For example, the trees in the diagram have respective heights of 3, 5, and 7.

The level of a node in calculating the height

- If n is the root of T , it is at level 1.
- If n is not the root of T , its level is 1 greater than the level of its parent.

In the (a) figure, node A is at level 1, node B is at level 2 and node D is at level 3.





Binary trees with the same nodes but different heights

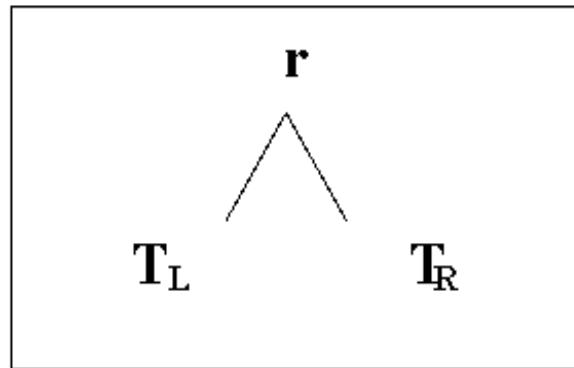
If T is empty, its height is zero

If T is not empty, its height is equal to the maximum level of its nodes.

A recursive definition of height:

If T is empty, its height is zero

If T is a nonempty binary tree, than because T is of the form



The height of T is 1 greater than the height of its root's taller subtree, that is,

$$\text{height}(T) = 1 + \max\{\text{height}(T_L), \text{height}(T_R)\}$$

Full, Complete, and Balanced Binary Trees

In a **full binary tree** of height h , all nodes that are at a level less than h have **two children each**.

The diagram depicts a full binary tree of height 3.

Each **node** in a full binary tree has **left** and **right** subtrees of the **same height**.

Among binary trees of height h , ...

- a full binary tree has as many leaves as possible.

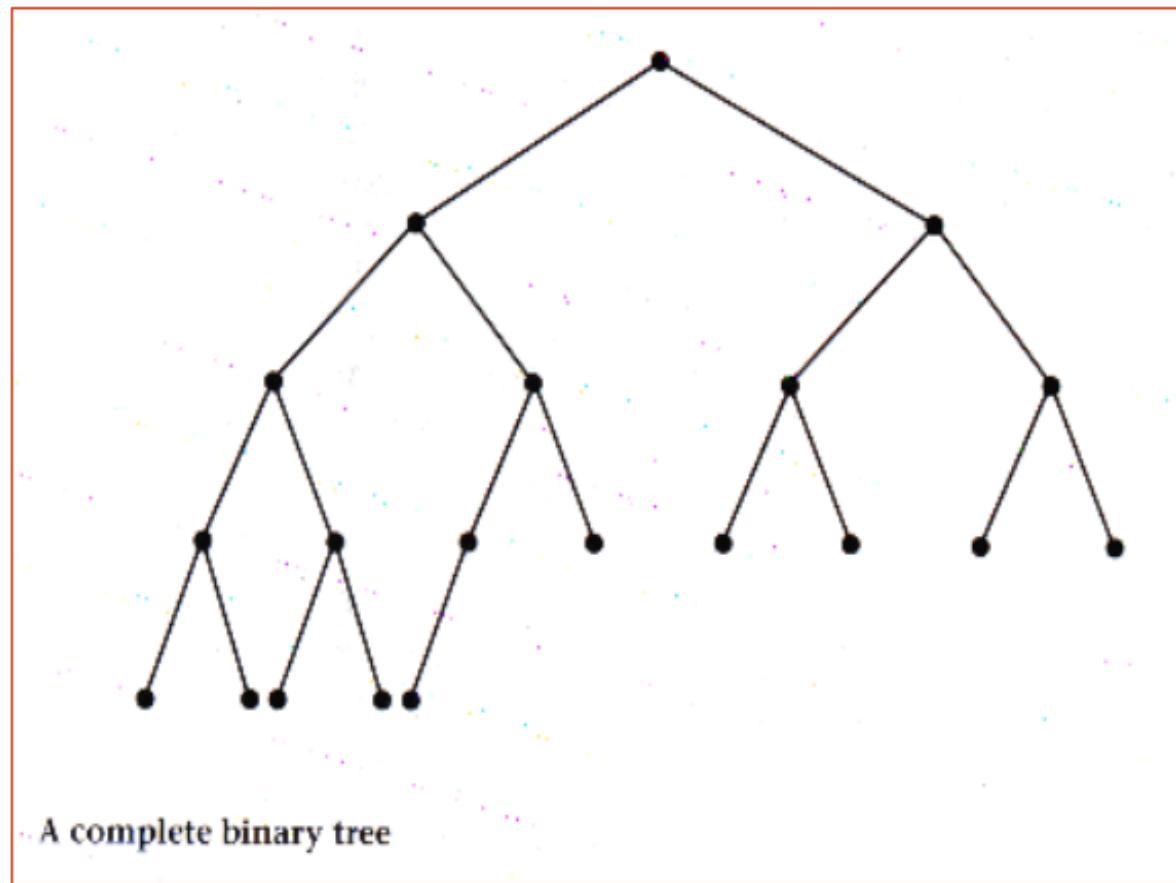
They are all at level h .



A **complete binary tree** of height h

- is a **binary tree** that is
- **full** down to level $- h-1$,
- with level h **filled in from left to right**,

as the diagram shows

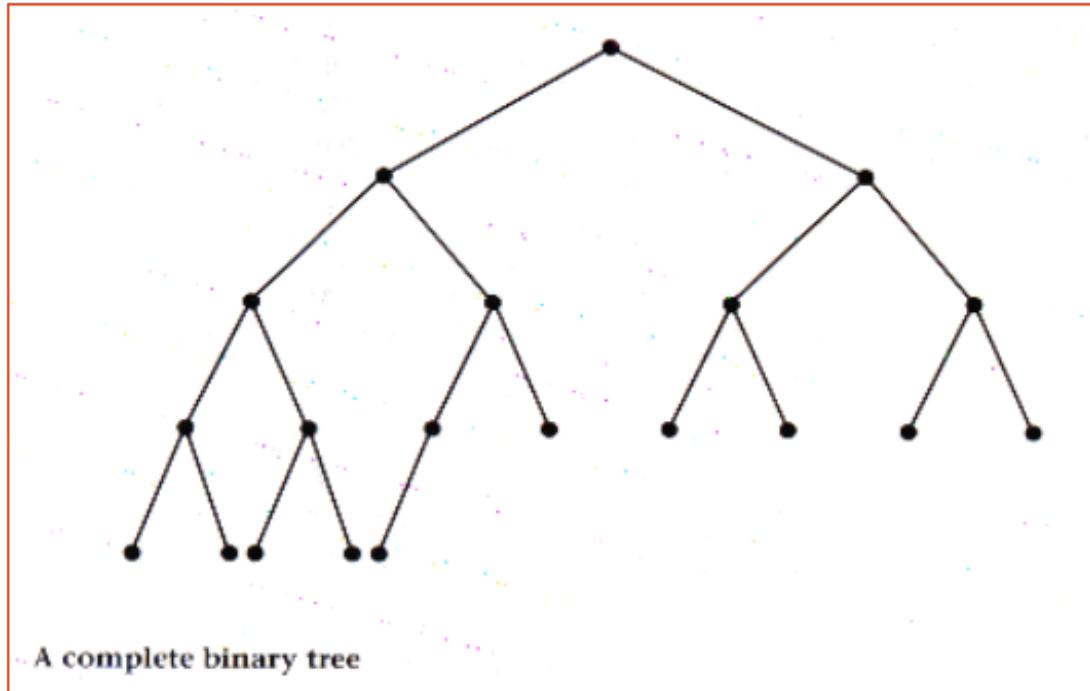


More formally, a binary tree T of height h is **complete** if:

1. All nodes at level $h - 2$ and above have two children each and
2. When a node at level $h - 1$ has children, all nodes to its left at the same height have two children each and
3. When a node at level $h - 1$ has one child, it is a **left** child.

Parts 2 and 3 of this definition formalise the requirement that level h be filled in from left to right.

A full binary tree is complete.

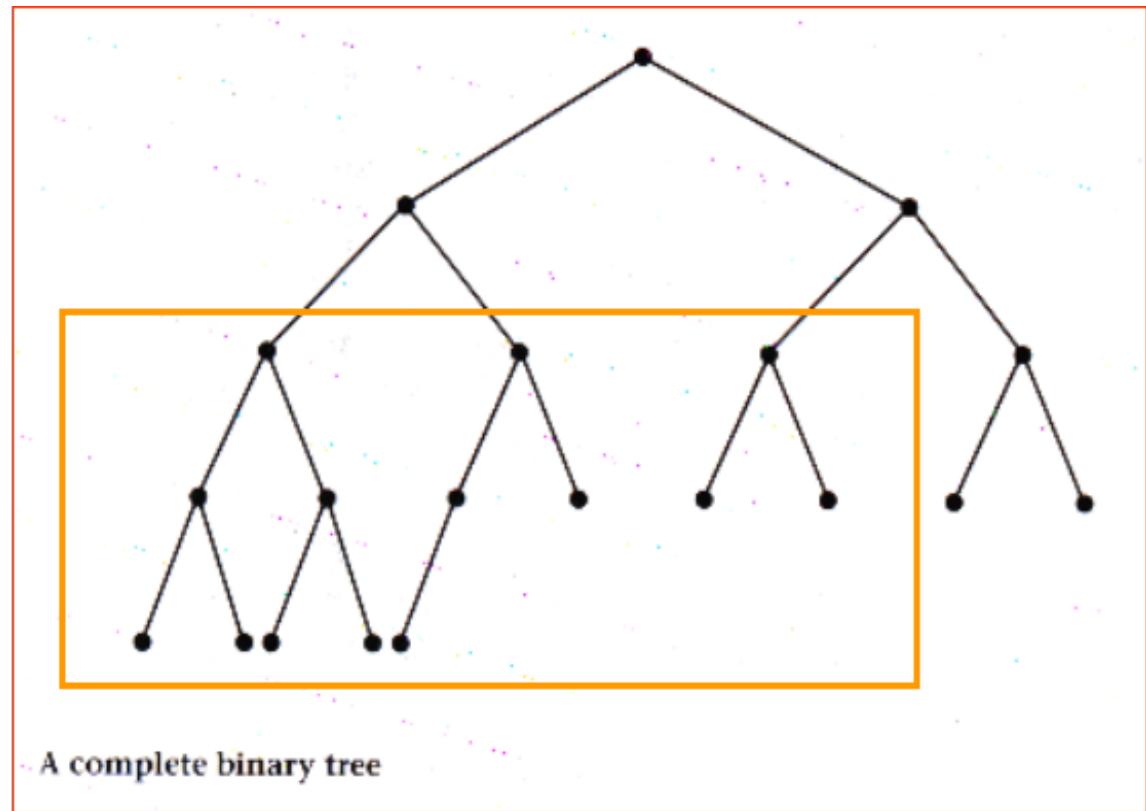


A binary tree is height balanced, or **balanced**, ...

if the height of any node's right subtree differs from the height of the node's left subtree by **no more than 1**.

The binary tree in the diagram is balanced

A complete binary tree is balanced.



Summary of Tree Terminology

General tree	A set of one or more nodes, partitioned into a root node and subsets that are general subtrees of the root.
Parent of node n	The node directly above node n in the tree.
Child of node n	A node directly below node n in the tree.
Root	The only node in the tree with no parent.
Leaf	A node with no children.
Siblings	Nodes with a common parent.
Ancestor of node n	A node on the path from the root to n .
Descendant of node n	A node on a path from n to a leaf.
Subtree of node n	A tree that consists of a child (if any) of n and the child's descendants.
Height	The number of nodes on the longest path from the root to a leaf.
Binary tree	A set of nodes that is either empty or partitioned into a root node and one or two subsets that are binary subtrees of the root. Each node has at most two children, the left child and the right child.
Left (right) child of node n	A node directly below and to the left (right) of node n in a binary tree.
Left (right) subtree of node n	In a binary tree, the left (right) child (if any) of node n plus its descendants.
Binary search tree	A binary tree where the value in any node n is greater than the value in every node in n 's left subtree, but less than the value of every node in n 's right subtree.
Empty binary tree	A binary tree with no nodes.
Full binary tree	A binary tree of height h with no missing nodes. All leaves are at level h and all other nodes each have two children.
Complete binary tree	A binary tree of height h that is full to level $h - 1$ and has level h filled in from left to right.
Balanced binary tree	A binary tree in which the left and right subtrees of any node have heights that differ by at most 1.

The ADT Binary Tree

As an abstract data type, the binary tree has operations that **add** and **remove** nodes and subtrees.

- By using these basic operations, we can build any binary tree.

Other operations **set** or **retrieve** the data of the tree and determine whether the tree **is empty**.

Traversal operations that visit every node in a binary tree are also typical.

Visiting a node means “**doing something with or to**” the node.

We have already seen traversal with a linked list.

Beginning with the list's first node we visit each node sequentially until the end of the list is reached.

Traversal of a binary tree visits the tree's nodes in one of several different orders.

The three standard orders are:

- **preorder**,
- **inorder** and
- **postorder**.

These are important and we will discuss them shortly.

Basic Operations of the ADT Binary Tree

1. Create an empty binary tree
2. Create a one-node binary tree, given an item
3. Remove all nodes from a binary tree, leaving it empty
4. Determine whether a binary tree is empty
5. Determine what data is the binary tree's root

Notice that we did **not** include in this basic set of operations an operation that **changes** the item in the root.

For some binary trees, such an operation would not be desirable.

The Pseudocode for the Basic Operations of the ADT Binary Tree

createBinaryTree()

//creates an empty binary tree

createBinaryTree(rootItem)

//creates a one-node binary tree whose root contains rootItem

makeEmpty()

//removes all of the nodes from a binary tree, leaving an empty tree.

isEmpty()

// determines whether a binary tree is empty

getRootItem() throws TreeException

// retrieves the data item in the root of a nonempty binary tree.

// throws TreeException if the tree is empty

We must still specify other operations for building the tree.

One possible set of operations is presented next.

General Operations of the ADT Binary Tree

The particular **operations** provided for an ADT binary tree **depend** on the kind of binary tree we are designing.

We look at some of the general operations for a binary tree ,
...

with the **assumption** that ...

- we are adding these operations to the basic operations of the ADT binary tree specified earlier.

Pseudocode for the General Operations of the ADT Binary Tree

createBinaryTree(rootItem, leftTree, rightTree)

// creates a binary tree whose root containsrootItem and

// has leftTree and rightTree, respectively, as its left and right subtrees.

setRootItem(newItem)

// replaces the data item in the root of a binary tree

// with newItem, if the tree is not empty.

// if the tree is empty, creates a root node whose data item is newItem and

// inserts the new node into the tree.

attachLeft(newItem) throws TreeException

// attaches a left child containing newItem to the root of
// a binary tree. Throws Tree Exception if the binary tree is empty
// (no root node to attach to) or a left subtree already exists
// (should explicitly detach it first)

attachRight(newItem) throws TreeException

// attaches a right child containing newItem to the root of
// a binary tree. Throws Tree Exception if the binary tree is empty
// (no root node to attach to) or a right subtree already exists
// (should explicitly detach it first)

attachLeftSubtree(leftTree) throws TreeException

// attaches a leftTree as the left subtree of the root of a binary tree
// and makes leftTree empty
// so that it cannot be used as a reference into this tree.
// Throws TreeException if the binary tree is empty
// (no root node to attach to) or a left subtree already exists
// (should explicitly detach it first)

attachRightSubtree(rightTree) throws TreeException

// attaches a rightTree as the right subtree of the root of a binary tree
// and makes rightTree empty
// so that it cannot be used as a reference into this tree.
// Throws TreeException if the binary tree is empty
// (no root node to attach to) or a right subtree already exists
// (should explicitly detach it first)

detachLeftSubtree(leftTree) throws TreeException

// detaches and returns the left subtree of a binary tree's root.
// Throws TreeException if the binary tree is empty
// (no root node to attach to)

detachRightSubtree(rightTree) throws TreeException

// detaches and returns the right subtree of a binary tree's root.
// Throws TreeException if the binary tree is empty
// (no root node to attach to)

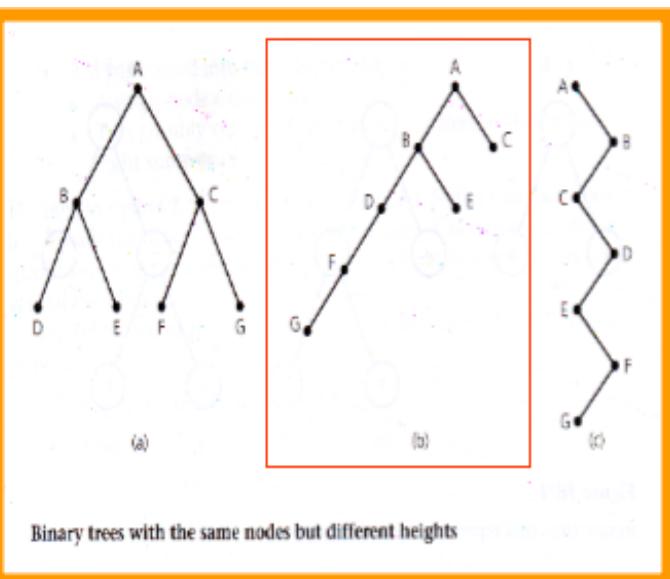
We can use these operations to build the binary tree in the diagram below in which the node labels represent character data.

The following pseudocode constructs the tree from the...

- subtree **tree1** rooted at “F”,
- subtree **tree2** rooted at “D”,
- subtree **tree3** rooted at “B”,
- subtree **tree4** rooted at “C”.

Initially ...

- **subtrees** exist but are **empty**



tree1.setRootItem("F")

tree1.attachLeft("G")

tree2.setRootItem("D")

tree2.attachLeftSubtree(tree1)

tree3.setRootItem("B")

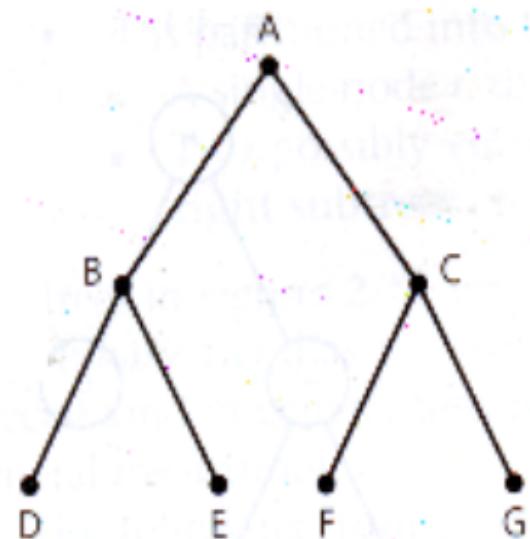
tree3.attachLeftSubtree(tree2)

tree3.attachRight("E")

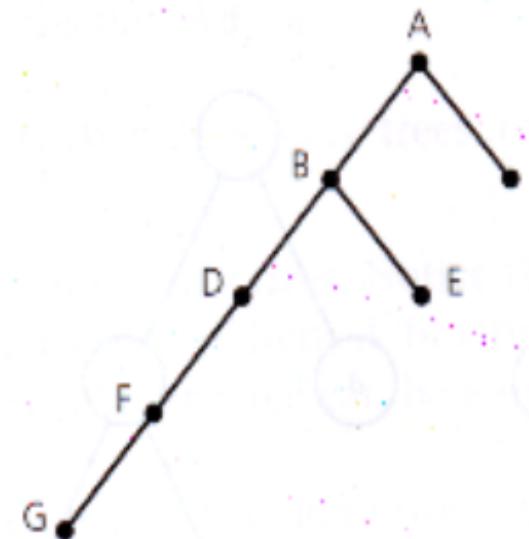
tree4.setRootItem("C")

BinTree.createBinaryTree ("A", tree3, tree4)

The traversal details are considered next.



(a)



(b)



(c)

Binary trees with the same nodes but different heights

Traversals of a Binary Tree

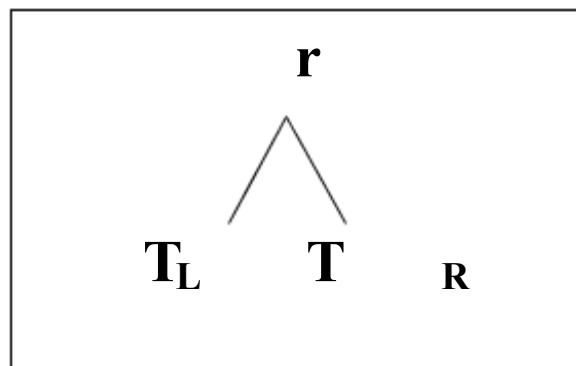
A traversal algorithm for a binary tree visits each node in the tree.

While visiting a node, we do something with or to the node.

- For the purposes of this discussion, we assume visiting a means displaying the data portion of the node.

With the recursive definition of a binary tree, we can recursive traversal algorithms as follows.

A binary tree is empty or is of the either form



If T is empty,
the traversal algorithm takes no action
an empty tree is the base case.

If T is not empty, the traversal algorithm must perform three tasks.

It must:

- Display the data in the root.
- Traverse the two subtrees T_L and T_R
each of which is a binary tree smaller than T .

Therefore, the general form of the recursive traversal algorithm is:

```
traverse(binTree)
// traverse the binary treebinTree

if (binTree is not empty)
{
    traverse (Left subtree ofbinTree's root)
    traverse(Right subtree ofbinTree's root)
} //end if
```

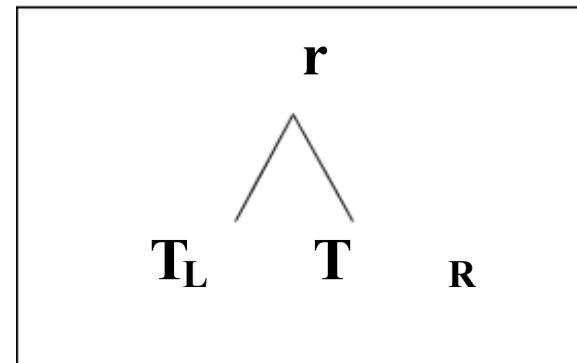
The above algorithm is missing the instruction to display the data in the root.

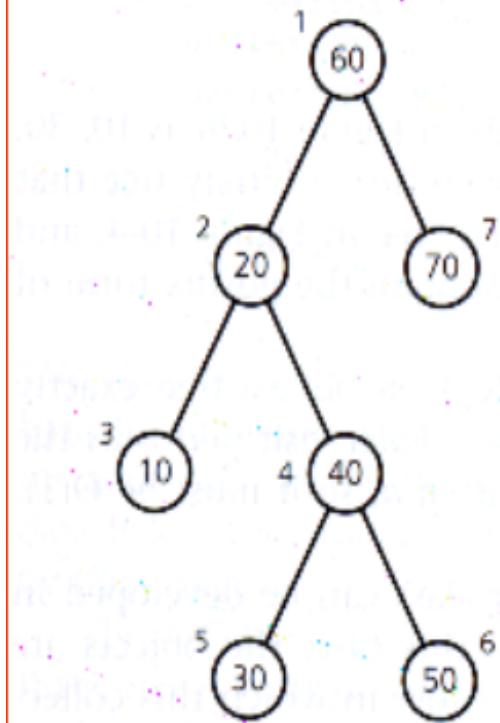
When **traversing** any binary tree, the algorithm has three choices of when to visit the root r .

- It can visit r **before** it traverses both of r 's subtrees.
- It can visit r **after** it has traversed r 's left subtree T_L , but **before** it traverses r 's right subtree T_R .
- It can visit r **after** it has traversed **both** of r 's subtrees.

These traversals are called **preorder**, **inorder** and **postorder** respectively.

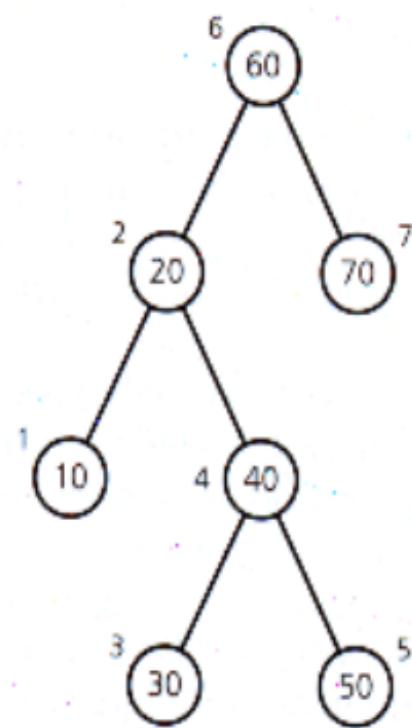
The diagram shows the results of these traversals for a given binary tree.



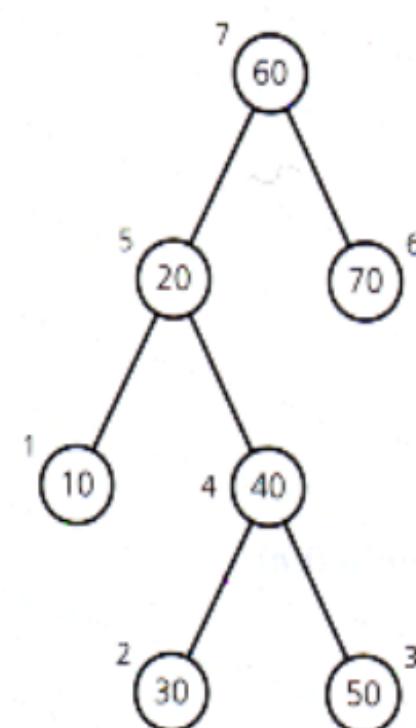


(a) Preorder: 60, 20, 10, 40, 30, 50, 70

(Numbers beside nodes indicate traversal order.)



(b) Inorder: 10, 20, 30, 40, 50, 60, 70



(c) Postorder: 10, 30, 50, 40, 20, 70, 60

Traversals of a binary tree: (a) preorder; (b) inorder; (c) postorder

The **preorder traversal algorithm** is as follows:

Preorder(binTree)

// traverse the binary tree binTree in preorder

// assume that visit a node means to display the node's data item

if (binTree is not empty)

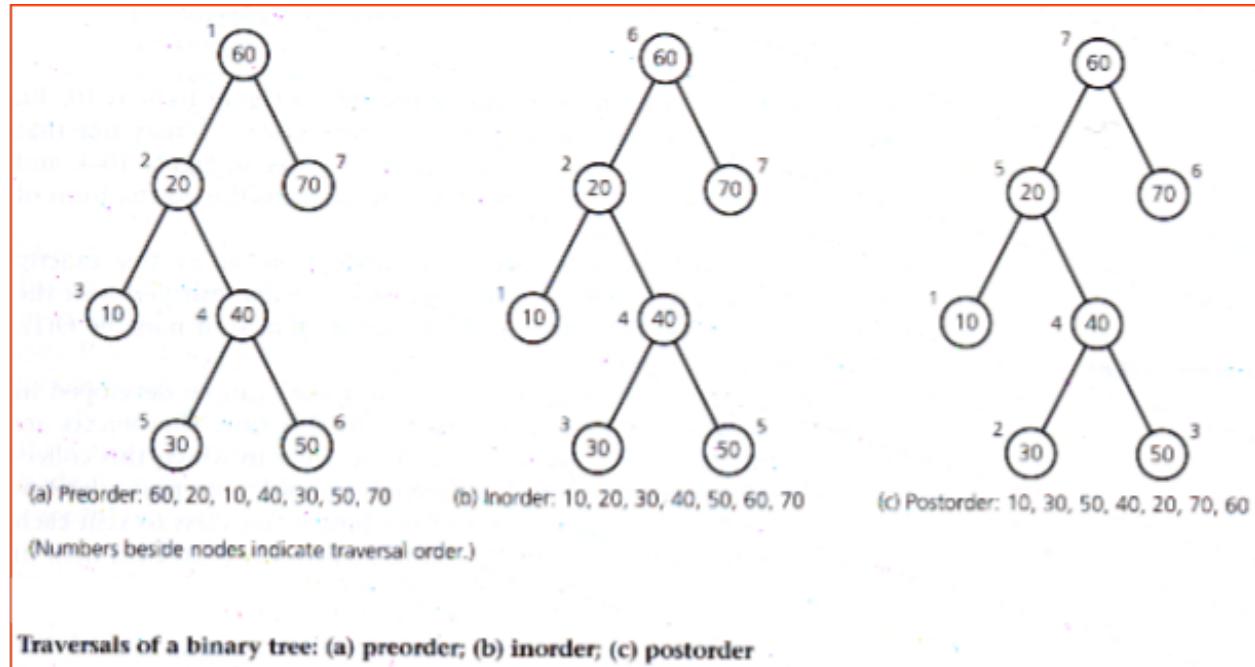
{

 display the data in the root of binTree

 preorder(Left subtree of binTree's root)

 preorder(Right subtree of binTree's root)

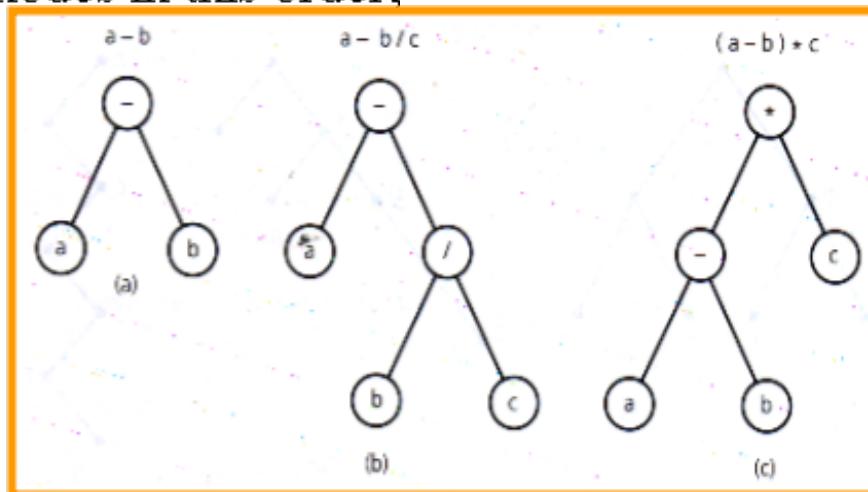
} //end if



The preorder traversal of the tree visits the nodes in this order:
60, 20, 10, 40, 30, 50, 70.

If we apply ...

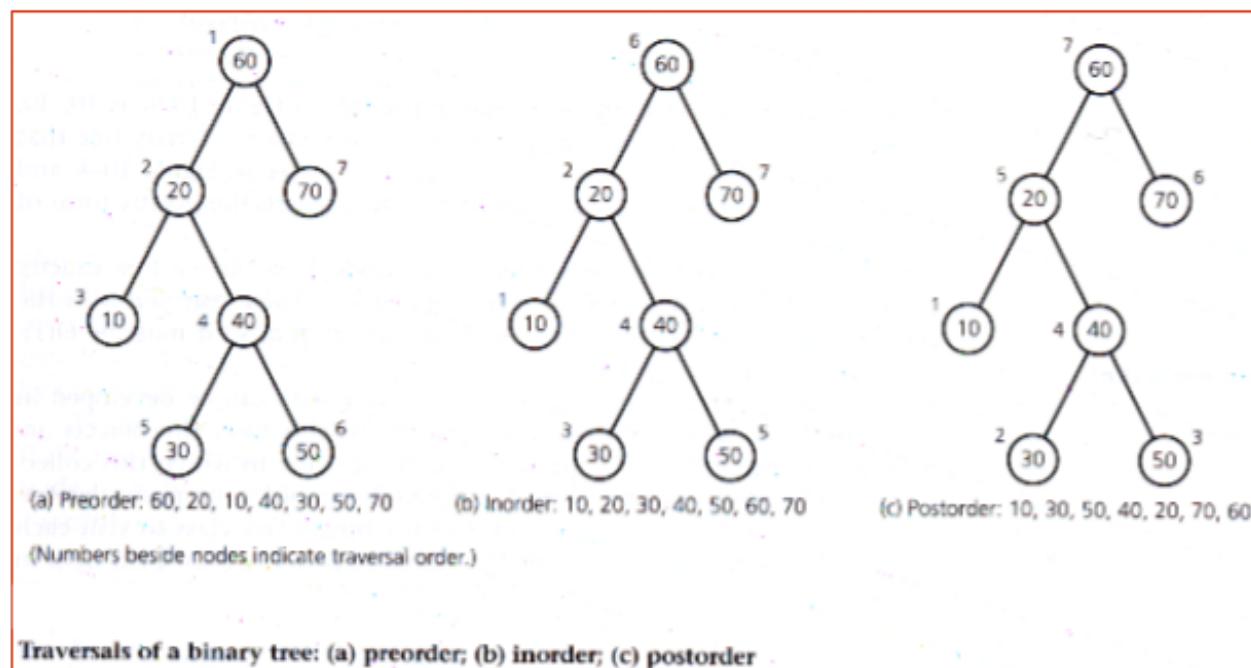
- **preorder** traversal to a binary tree that
- represents an algebraic expression and
- **display** the nodes as you visit them,



we will obtain the **prefix** form of the expression
(and not infix or postfix)

That is, the **prefix** expressions are:

- (a) **-ab**
- (b) **-a/bc**
- (c) ***-abc**



The **inorder traversal algorithm** is as follows:

inorder(binTree)

```
// traverses the binary tree binTree in inorder.  
// assumes that visit a node means to display  
// the node's data item
```

```
if (binTree is not empty)
```

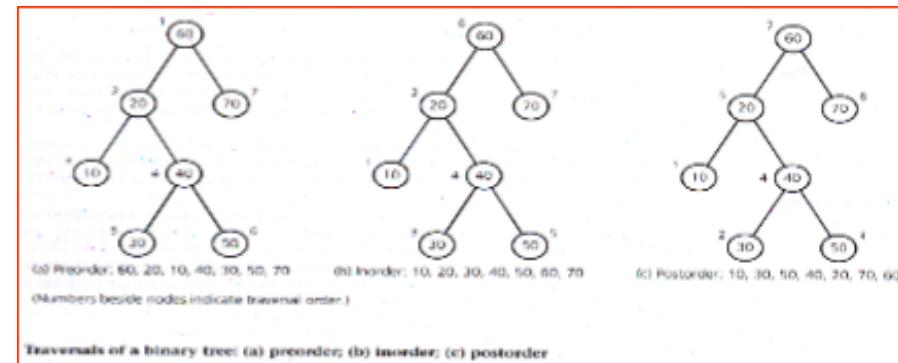
```
{
```

```
    inorder(Left subtree of binTree's root)
```

```
    Display the data in the root of binTree
```

```
    Inorder(Right subtree of binTree's root)
```

```
} //end if
```



The result of the **inorder** traversal of the tree in the diagram is 10, 20, 30, 40, 50, 60, 70.

If you apply the **inorder** traversal to a binary search tree, you will visit the nodes in order according to their data values .

The postorder traversal algorithm

postorder(binTree)

// traverses the binary tree binTree in postOrder

// assumes that visit a node means

// display the nodes data item.

if (binTree is not empty)

{

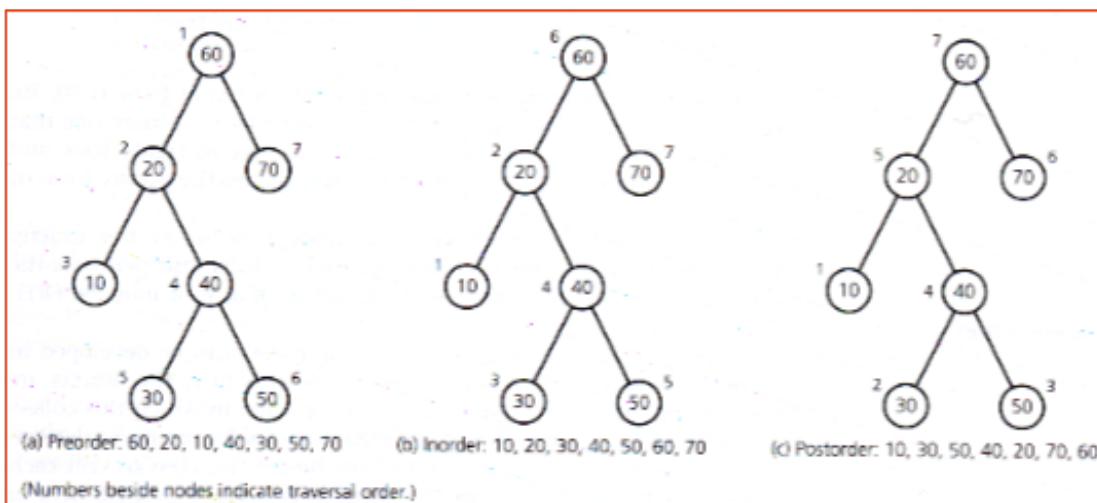
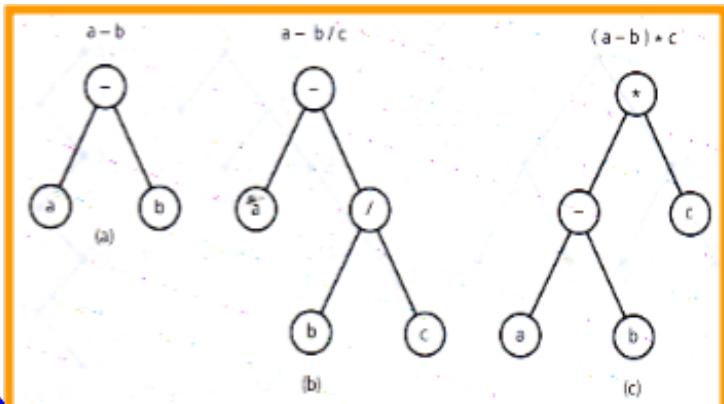
postorder(Left subtree of binTree's root)

postorder(Right subtree of binTree's root)

Display the data in the root of binTree

} //end if

The result of the **postorder** traversal of the tree in the diagram is **10, 30, 50, 40, 20, 70, 60**



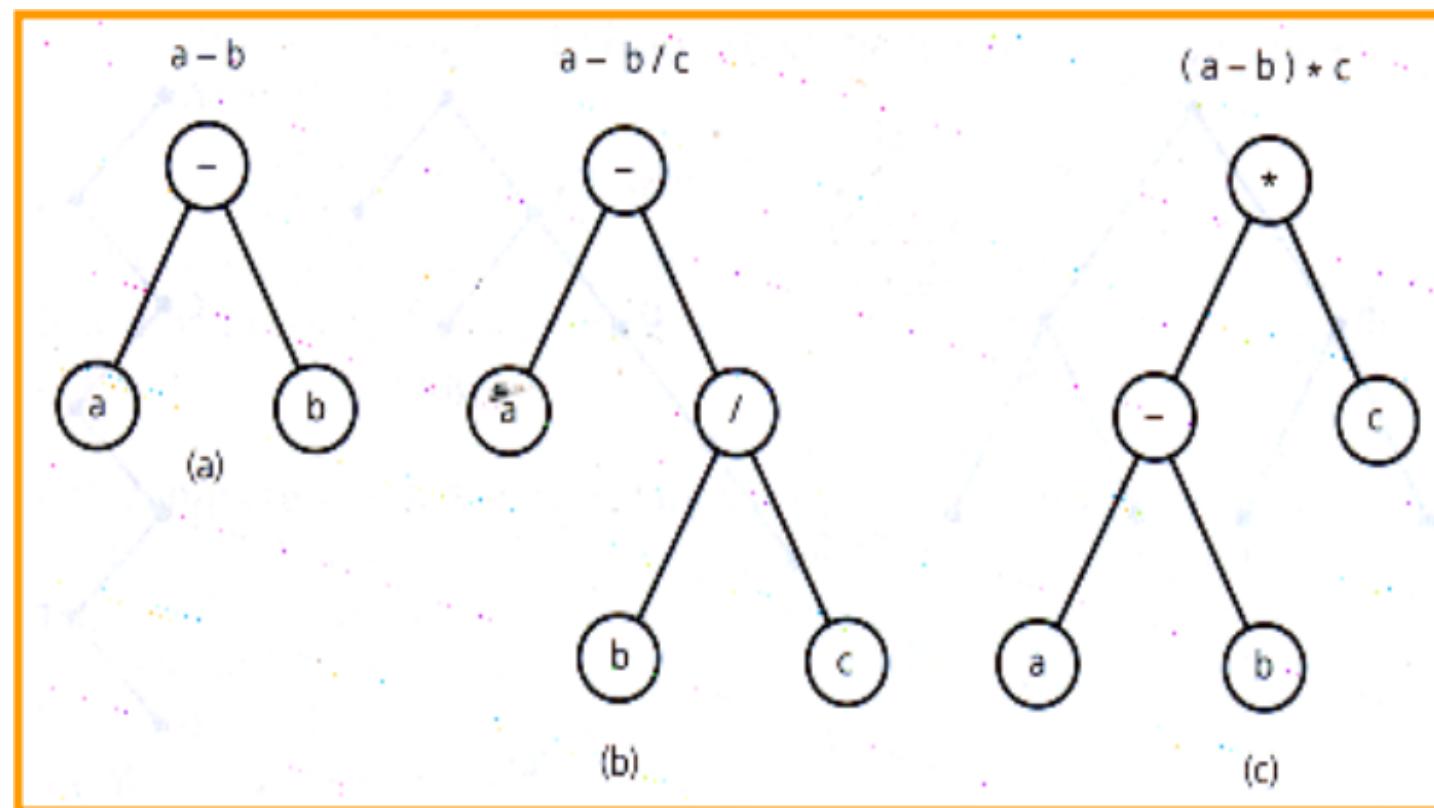
Traversals of a binary tree: (a) preorder; (b) inorder; (c) postorder

If you apply **postorder** traversal to a binary tree that...

- represents an algebraic expression and
- display the nodes as you visit them,
- you will obtain the **postfix** form of the expression

The **postfix** expressions are:

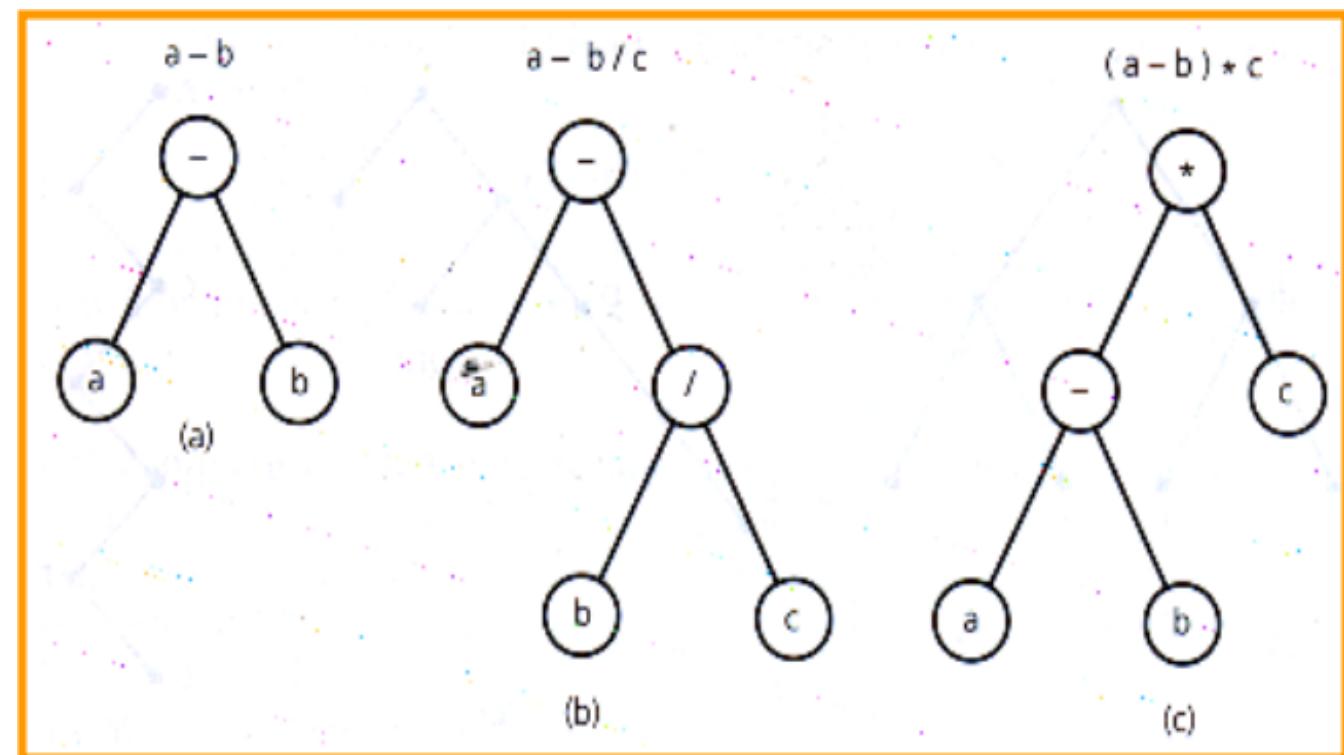
- (a) ab-
- (b) abc/-
- (c) ab-c*



Each of these traversals visits every node in a binary tree exactly once.

- Therefore, **n** visits occur for a tree of **n** nodes.
- Each visit performs the same operations of each node, independently of **n**.

Traversal performance is therefore a function of the size of **n**, and directly proportional to **n**.



Programs to do this week!

- See the next slide for details of the labwork!

Binary Tree - Netscape

File Edit View Go Communicator Help

Back Forward Reload Home Search Netscape Print Security Stop

Bookmarks Location: file:///C:/ITP/SoftDev/4_JAVA/My%20Programs/Tree/Tree.html What's Related

Tafore's Binary Tree

Binary Tree	Operation
<p>Fill Find Ins Trav Del Enter number: <input type="text"/></p> <p>Press a button</p> <pre>graph TD; 11((11)) --> 9((9)); 11 --> 85((85)); 9 --> 1((1)); 9 --> 0((0)); 1 --> 7((7)); 1 --> 4((4)); 7 --> 8((8)); 85 --> 32((32)); 85 --> 88((88)); 32 --> 25((25)); 32 --> 74((74)); 25 --> 18((18)); 25 --> 26((26)); 74 --> 62((62)); 74 --> 84((84)); 88 --> 86((86)); 88 --> 97((97)); 97 --> 91((91)); 97 --> 98((98))</pre>	<p>Fill creates a new tree with N nodes.</p> <p>Find searches for a node with value N.</p> <p>Ins inserts a new node with value N.</p> <p>Trav traverses the tree in ascending order.</p> <p>Del deletes the node with value N.</p> <p>(Type N into "Enter number" box.)</p>

Applet Tree running

Write out the inorder, preorder and post order output for above

An Array-based Representation.

If we use a Java class to define a node in the tree, we can represent the entire binary tree by using an **array of tree nodes**.

Each tree node contains a `data` portion, a name in this case, and two `indexes`, one for each of the node's children.

The variable **root** is an index to the tree's root within the array **tree**.

If the tree is **empty**, root is **-1**.

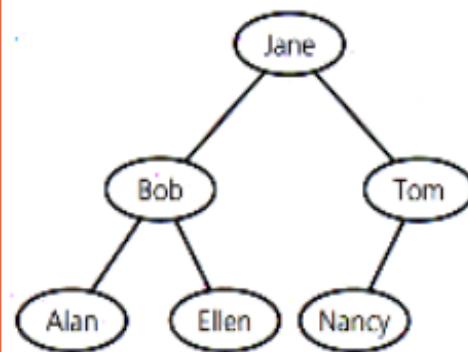
Both **leftChild** and **rightChild** within a node are indexes to the children of that node.

If a node has **no left child**, **leftChild** is **-1**.

If a node has **no right child**, **rightChild** is **-1**.

The diagram contains a binary tree and the data fields for its array-based implementation.

(a)



(b)

item	leftChild	rightChild	root
Jane	1	2	0
Bob	3	4	free
Tom	5	-1	6
Alan	-1	-1	
Ellen	-1	-1	
Nancy	-1	-1	
?	-1	7	
?	-1	8	
?	-1	9	
.	.	.	
.	.	.	
.	.	.	

(a) A binary tree of names; (b) its array-based implementation

As the tree changes due to insertions and deletions, its **nodes** may not be in contiguous elements of the array.

Therefore, this implementation requires you to establish a **list of available nodes** which is called a **free list**.

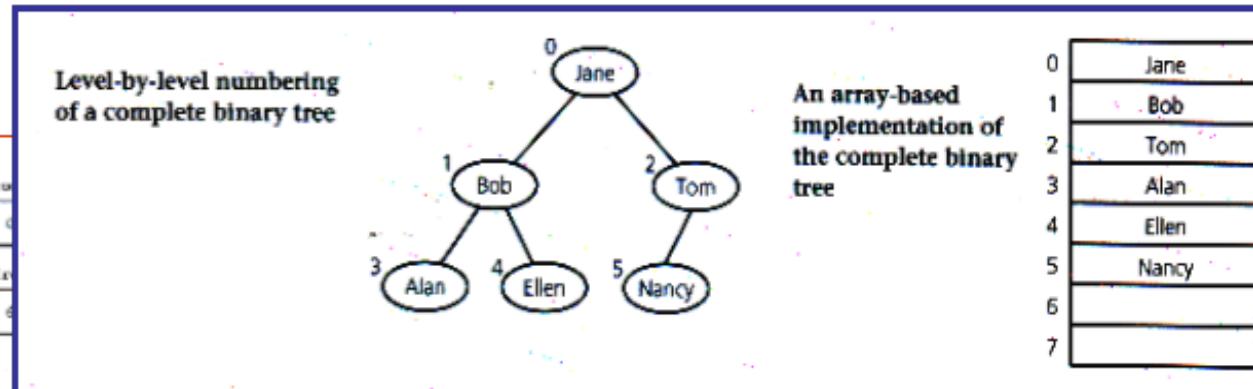
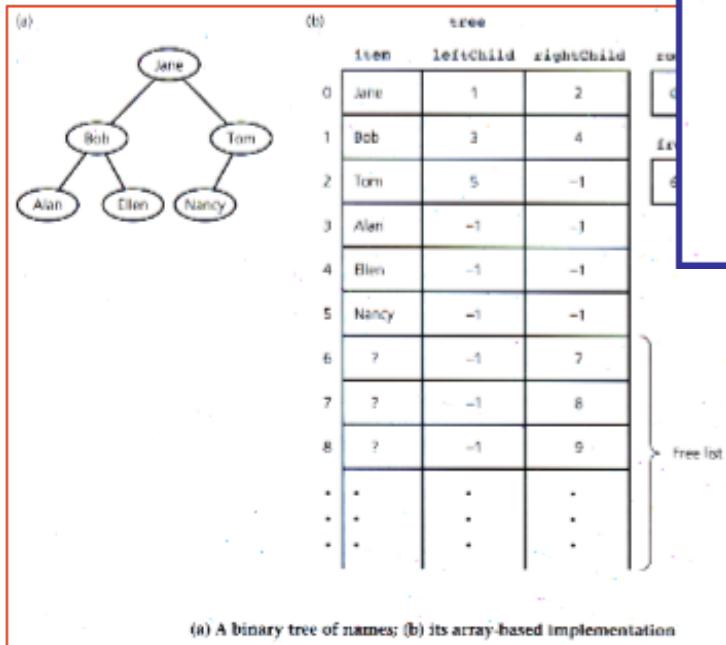
- To **insert** a new node into the tree, we must first **obtain an available node from the free list**
- If you **delete** a node from the tree, we **place it into the free list** so that the node can be reused later.

The variable **free** is the **index to the first node in the free list** and, **arbitrarily** the **rightChild** field of each node in the **free** list is the **index of the next node in the free list**

An Array-based representation of a complete tree.

The previous implementation works with any binary tree, even though the tree in the previous diagram is complete.

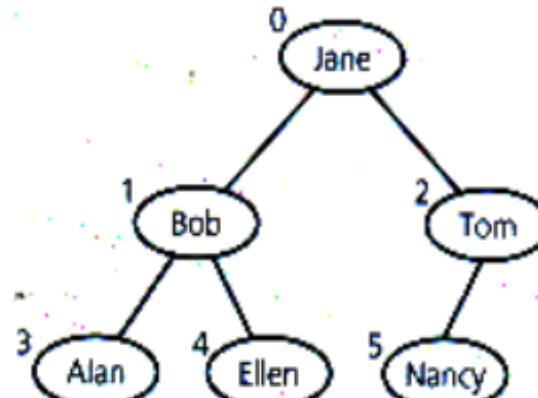
- If you know that your binary tree is complete, you can use a simpler array-based implementation that saves memory.
- A **complete** tree of height \mathbf{h} is full to level $\mathbf{h - 1}$ and has level \mathbf{h} filled from **left to right**.



VS

•

Level-by-level numbering
of a complete binary tree



An array-based
implementation of
the complete binary
tree

0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Nancy
6	
7	

The diagram shows the complete binary tree with its nodes numbered according to a standard level-by-level scheme.

The **root** is numbered **0**, and the **children** of the root (the next level of the tree) are numbered, left to right, **1** and **2**, and so on...

We place these nodes into the array tree in numeric order.

The node numbered **k** is contained in **tree[k]**.

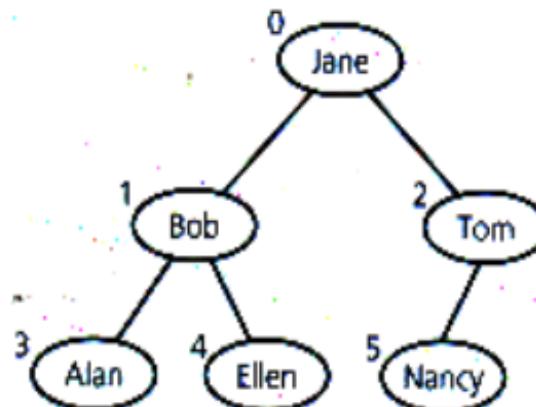
- We can match the tree to the array in the example in the diagrams

Given any node **tree[k]**, we can easily locate **both of its children** and its **parent**

- Its left child (if it exists) is **tree[2*k+1]**
- Its right child (if it exists) is **tree[2*k+2]**
- Its parent (if tree[k] is not root) is **tree[(k - 1) / 2]**

We can easily prove this with the assistance of the diagram by walking the tree.

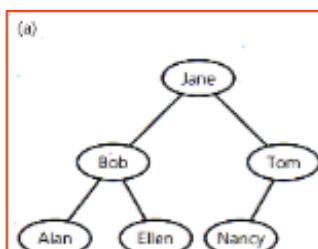
Level-by-level numbering
of a complete binary tree



An array-based
implementation of
the complete binary
tree

0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Nancy
6	
7	

- The array based implementation requires a complete tree.
- If nodes were missing from the middle of the tree , the numbering scheme would be thrown off, and the parent-child relationship would be ambiguous.
- This requirement implies that any changes to the tree must maintain its completeness.



(b)

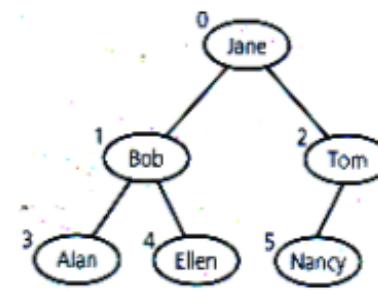
tree

item	leftChild	rightChild	root
0 Jane	1	2	0
1 Bob	3	4	free
2 Tom	5	-1	6
3 Alan	-1	-1	
4 Ellen	-1	-1	
5 Nancy	-1	-1	
6 ?	-1	7	
7 ?	-1	8	
8 ?	-1	9	
· · ·	· · ·	· · ·	

Free list

(a) A binary tree of names; (b) its array-based implementation

Level-by-level numbering
of a complete binary tree



An array-based
implementation of
the complete binary
tree

0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Nancy
6	
7	

A Reference-Based Implementation of the ADT Binary Tree

The following classes provide a reference-based implementation for the ADT Binary Tree.

TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeNode.java]

File Edit Search View Tools Macros Configure Window Help

Command Results
TreeNode.java *

```
1 public class TreeNode
2 {
3     private Object item;
4     private TreeNode leftChild;
5     private TreeNode rightChild;
6
7     public TreeNode(Object newItem)
8     {
9         // Initializes tree node with item and no children.
10        item = newItem;
11        leftChild = null;
12        rightChild = null;
13    } // end constructor
14
15    public TreeNode(Object newItem,
16                    TreeNode left, TreeNode right)
17    {
18        // Initializes tree node with item and
19        // the left and right children references.
20        item = newItem;
21        leftChild = left;
22        rightChild = right;
23    } // end constructor
24
25    public Object getItem()
26    {
27        // Returns the item field.
28        return item;
29    } // end getItem
30
31    public void setItem(Object newItem)
32    {
33        // Sets the item field to the new value newItem.
34        item = newItem;
35    } // end setItem
36
37    public TreeNode getLeft()
38    {
39        // Returns the reference to the left child.
40        return leftChild;
41    } // end getLeft
42
43    public void setLeft(TreeNode newLeft)
44    {
45        leftChild = newLeft;
46    } // end setLeft
47
48    public TreeNode getRight()
49    {
50        // Returns the reference to the right child.
51        return rightChild;
52    } // end getRight
53
54    public void setRight(TreeNode newRight)
55    {
56        rightChild = newRight;
57    } // end setRight
58
59    public String toString()
60    {
61        return item.toString();
62    }
63}
```

ANSI Characters

33	!
34	"
35	#
36	\$
37	%
38	&
39	'
40	(
41)
42	*
43	+
44	-
45	.
46	,
47	/
48	0

1 1 Read Ovr Block Sync Rec Caps 23:31

TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeNode.java]

File Edit Search View Tools Macros Configure Window Help

Command Results
TreeNode.java*

```
41 } // end getLeft
42
43 public void setLeft(TreeNode left) ←
44 {
45 // Sets the left child reference to left.
46 leftChild = left;
47 } // end setLeft
48
49 public TreeNode getRight() ←
50 {
51 // Returns the reference to the right child.
52 return rightChild;
53 } // end getRight
54
55 public void setRight(TreeNode right) ←
56 {
57 // Sets the right child reference to right.
58 rightChild = right;
59 } // end setRight
60 } // end TreeNode
```

ANSI Characters

33	!
34	"
35	#
36	\$
37	%
38	&
39	'
40	(
41)
42	*
43	+
44	-
45	.
46	,
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	0

41 1 Read Ovr Block Sync Rec Caps 23:32

TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeException.java]

File Edit Search View Tools Macros Configure Window Help

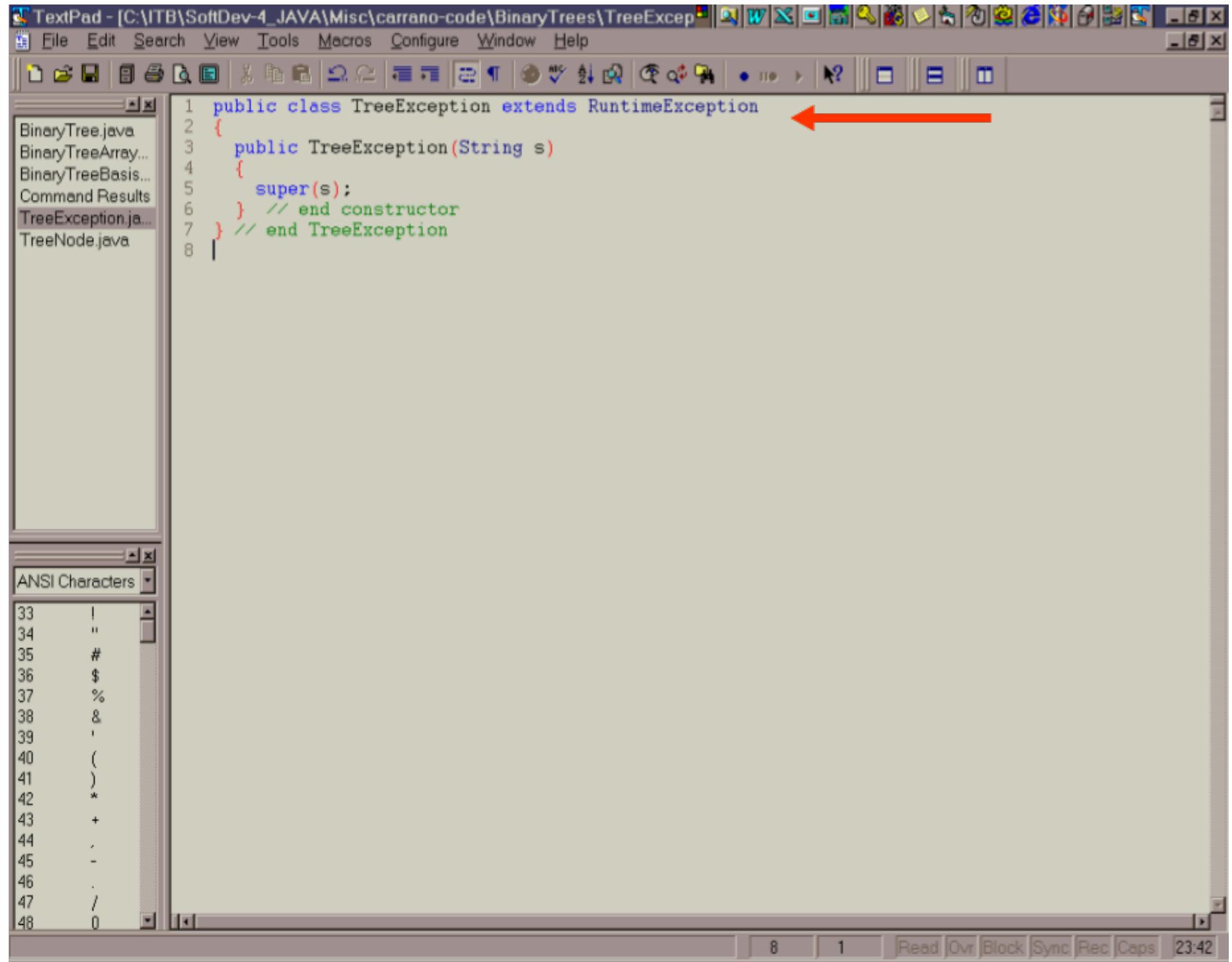
BinaryTree.java
BinaryTreeArray...
BinaryTreeBasis...
Command Results
TreeException.ja...
TreeNode.java

```
1 public class TreeException extends RuntimeException
2 {
3     public TreeException(String s)
4     {
5         super(s);
6     } // end constructor
7 } // end TreeException
8
```

ANSI Characters

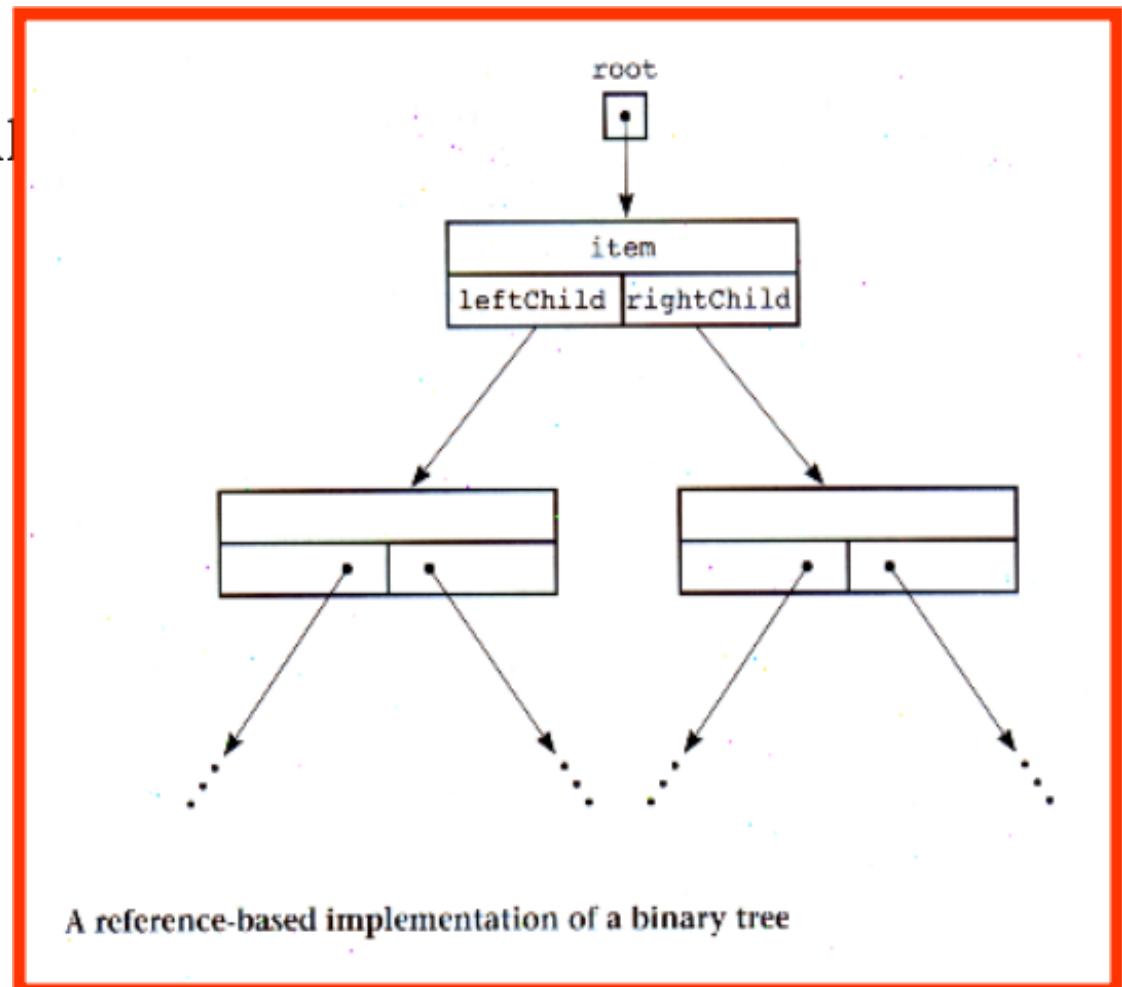
33	!
34	"
35	#
36	\$
37	%
38	&
39	'
40	(
41)
42	*
43	+
44	,
45	-
46	.
47	/
48	0

8 1 Read Ovr Block Sync Rec Caps 23:42



The class **treeNode** is analogous to the class **Node** that we used for a linked list.

- Within the class **BinaryTreeBasis**, the external reference **root** references the tree's root.
- If the tree is empty, root is null

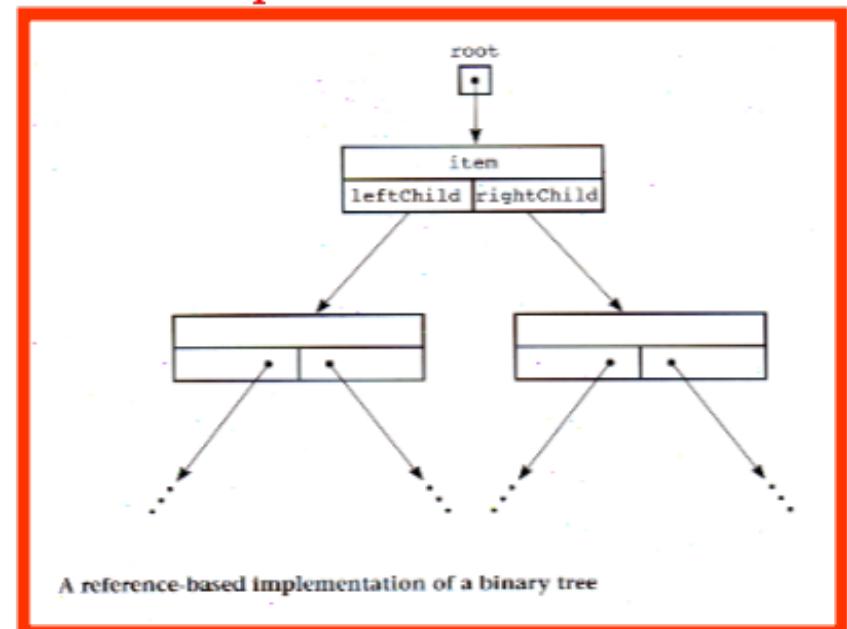


The **root** of a nonempty binary tree **has a left subtree and a right subtree**, each of which is a binary tree.

In a reference-based implementation, ...

- **root** references the **root** r of a binary tree.
- **root.getLeft()** references the root of the **left** subtree of r, and
- **root.getRight()** references the root of the **right** subtree of r.

We now provide the details for a reference-based implementation of the **ADT Binary Tree**.



TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\BinaryTree]

File Edit Search View Tools Macros Configure Window Help

BinaryTree.java
BinaryTreeArray...
BinaryTreeBasis...
Command Results
TreeNode.java

```
1 public abstract class BinaryTreeBasis
2 {
3     protected TreeNode root;
4
5     public BinaryTreeBasis()
6     {
7         root = null;
8     } // end default constructor
9
10    public BinaryTreeBasis(Object rootItem)
11    {
12        root = new TreeNode(rootItem, null, null);
13    } // end constructor
14
15    public boolean isEmpty()
16    {
17        // Returns true if the tree is empty, else returns false.
18        return root == null;
19    } // end isEmpty
20
21    public void makeEmpty()
22    {
23        // Removes all nodes from the tree.
24        root = null;
25    } // end makeEmpty
26
27    public Object getRootItem() throws TreeException
28    {
29        // Returns the item in the tree's root.
30        if (root == null)
31        {
32            throw new TreeException("TreeException: Empty tree");
33        }
34        else
35        {
36            return root.getItem();
37        } // end if
38    } // end getRootItem
39} // end BinaryTreeBasis
```

ANSI Characters

! " # \$ % & ' () * + - . / 0

BinaryTreeBasis will be used as the base for the implementation of particular binary trees.

- It is declared as an **abstract** class because it is used for inheritance purposes only. There can be no direct instances of this class.
- **BinaryTreeBasis** declares the root of the tree as a protected item so that the subclasses will have direct access to the root of the tree.
- It also provides methods to check for an empty tree, to make the tree empty, and to retrieve the contents of the root node.

Setting the contents of the root node is left to a subclass, since different kinds of binary trees may have varying requirements regarding the value in the **root** node of the tree.

The following class implements the **general operation** of a binary tree and is **derived** from [BinaryTreeBasis](#)

TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\BinaryTree]

File Edit Search View Tools Macros Configure Window Help

BinaryTree.java Command Results TreeNode.java

```
1 public class BinaryTree extends BinaryTreeBasis ←
2 {
3     public BinaryTree()
4     {
5         // end default constructor
6
7     public BinaryTree(Object rootItem) ←
8     {
9         super(rootItem);
10    } // end constructor
11
12    public BinaryTree(Object rootItem, BinaryTree leftTree, BinaryTree rightTree)
13    {
14        root = new TreeNode(rootItem, null, null);
15        attachLeftSubtree(leftTree);
16        attachRightSubtree(rightTree);
17    } // end constructor
18
19    public void setRootItem(Object newItem) ←
20    {
21        if (root != null)
22        {
23            root.setItem(newItem);
24        }
25        else
26        {
27            root = new TreeNode(newItem, null, null);
28        } // end if
29    } // end setRootItem
30
31    public void attachLeft(Object newItem) ←
32    {
33        if (!isEmpty() && root.getLeft() == null)
34        {
35            // assertion: nonempty tree; no left child
36            root.setLeft(new TreeNode(newItem, null, null));
37        } // end if
38    } // end attachLeft
39
40    public void attachRight(Object newItem) ←
41    {
```

ANSI Characters

33	!
34	"
35	#
36	\$
37	%
38	&
39	'
40	(
41)
42	*
43	+
44	-
45	.
46	,
47	/
48	0

41 4 Read Ovr Block Sync Rec Caps 23:46

TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\BinaryTree]

File Edit Search View Tools Macros Configure Window Help

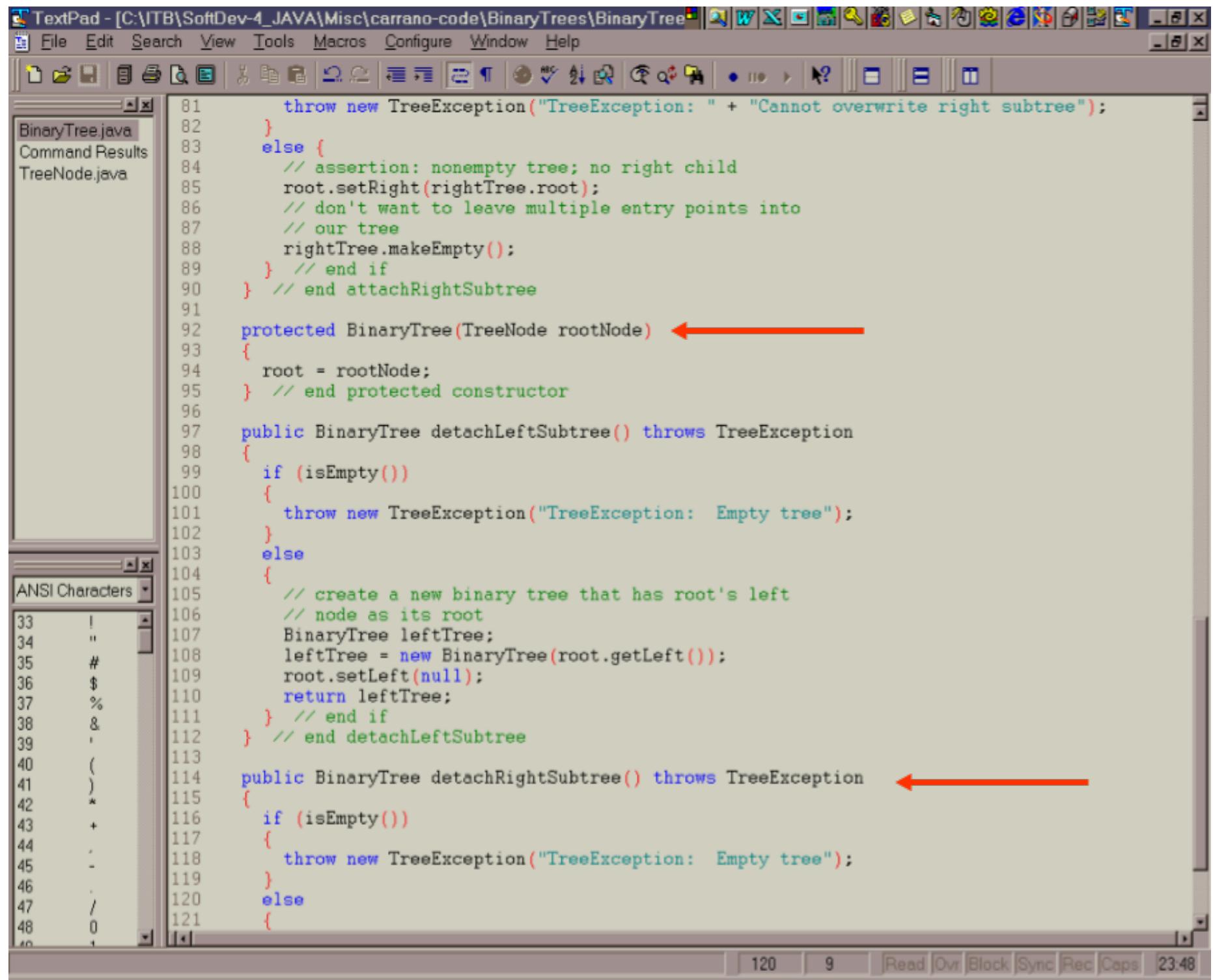
BinaryTree.java Command Results TreeNode.java

```
41  {
42      if (!isEmpty() && root.getRight() == null)
43      {
44          // assertion: nonempty tree; no right child
45          root.setRight(new TreeNode(newItem, null, null));
46      } // end if
47  } // end attachRight
48
49  public void attachLeftSubtree(BinaryTree leftTree) throws TreeException ←
50  {
51      if (isEmpty())
52      {
53          throw new TreeException("TreeException: Empty tree");
54      }
55      else if (root.getLeft() != null)
56      {
57          // a left subtree already exists; it should have been
58          // deleted first
59          throw new TreeException("TreeException: " + "Cannot overwrite left subtree");
60      }
61      else
62      {
63          // assertion: nonempty tree; no left child
64          root.setLeft(leftTree.root);
65          // don't want to leave multiple entry points into
66          // our tree
67          leftTree.makeEmpty();
68      } // end if
69  } // end attachLeftSubtree
70
71  public void attachRightSubtree(BinaryTree rightTree) throws TreeException ←
72  {
73      if (isEmpty())
74      {
75          throw new TreeException("TreeException: Empty tree");
76      }
77      else if (root.getRight() != null)
78      {
79          // a right subtree already exists; it should have been
80          // deleted first
81          throw new TreeException("TreeException: " + "Cannot overwrite right subtree");
82      }
83  } // end attachRightSubtree
84
85  public void makeEmpty()
86  {
87      if (root != null)
88      {
89          root.setLeft(null);
90          root.setRight(null);
91          root.setItem(null);
92      }
93  } // end makeEmpty
94
95  public void printInorder()
96  {
97      if (root != null)
98      {
99          root.printInorder();
100     }
101  } // end printInorder
102 }
```

ANSI Characters

33	!
34	"
35	#
36	\$
37	%
38	&
39	'
40	(
41)
42	*
43	+
44	-
45	.
46	,
47	/
48	0

80 23 Read Ovr Block Sync Rec Caps 23:47



TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\BinaryTree]

File Edit Search View Tools Macros Configure Window Help

BinaryTree.java Command Results TreeNode.java

```
121  {
122      BinaryTree rightTree;
123      rightTree = new BinaryTree(root.getRight());
124      root.setRight(null);
125      return rightTree;
126  } // end if
127 } // end detachRightSubtree
128 } // end BinaryTree
```

ANSI Characters

33	!
34	"
35	#
36	\$
37	%
38	&
39	'
40	(
41)
42	*
43	+
44	-
45	-
46	.
47	/
48	0

128 | 20 | Read | Ovr | Block | Sync | Rec | Caps | 23:48

The class **BinaryTree** has more constructors than previous classes we have seen.

- They allow us to define binary trees in a variety of circumstances.
- Two of these constructors refer back to the abstract class **BinaryTreeBasis** with a third declared within a **BinaryTree**.

With these constructors we can construct a binary tree:

- That is **empty**
- From **data** for its **root**, which is its only node
- From **data** for its **root** and the **root's two subtrees**

For example, the following statements invoke the three constructors:

```
BinaryTree tree1 = new BinaryTree();  
BinaryTree tree2 = new BinaryTree(root2);  
BinaryTree tree3 = new BinaryTree(root3);  
BinaryTree tree4 = new BinaryTree(root4, tree2, tree3);
```

In these statements, ...

- tree1 is an empty binary tree,
- tree2 and tree3 have only root nodes,
whose data is **root2** and **root3** respectively
- tree4 is a binary tree
whose **root** contains **root4** and has **subtrees tree2 and tree3**.

The class also contains a **protected constructor**, which creates a tree from a reference to a root node.

For example,

```
leftTree = new BinaryTree(root.getLeft())
```

This constructs a tree **leftTree** whose root is the node that **root.getLeft()** references.

- Although the methods **detachLeftSubtree** and **detactRightSubTree** use this constructor, it should **not** be available to clients of the class, because they do not have access to node references.
- (“**The Wall surrounding the ADT**”)

Therefore, this constructor is not public.

Note that **attachLeftSubTree** and **attachRightSubTree** call **makeEmpty()** with the subtree as an argument **after** the subtree has been attached to the invoking tree.

- This causes the root of the subtree as it existed in the client to be set to null.
- Therefore, the client will not be able to access and manipulate the subtree directly, a violation of the **abstraction of the tree**

Tree Treversal Using an Iterator

We will use the tree traversals to determine the order in which an **iterator** will visit the nodes of a tree.

The tree **iterator** will implement the Java **Iterator** interface and will provide methods to set the**iterator** to the type of traversal desired.

Since the abstract class **BinaryTreeBasis** has sufficient information to perform a traversal, we will define the**iterator** using this class.

This will allow the **iterator** to be used by any subclass of the **BinaryTreeBasis**

A class that implements the **Iterator** interface of Java must provide three methods:

- **next()**
- **hasNext()**
- **remove()**

We will not implement the `remove()` method in this version of the **iterator** because:

- The semantics of removing a node from a tree may depend on the type of tree we are working with(i.e. a binary search tree)
- The class `BinaryTree` itself does not provide a method for removing nodes from the tree.

The implementation presented here for the **iterator** of a binary tree **assumes** that the iterator order will be set by calling `setPreorder`, `setInorder`, or `setPostorder`.

- Until one of these methods are called, the iterator will not provide any items from the tree (`hasNext` returns false)

TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeIterator.java]

File Edit Search View Tools Macros Configure Window Help

Command Results
TreeIterator.java *

```
1 public class TreeIterator implements java.util.Iterator
2 {
3     private BinaryTreeBasis binTree;
4     private TreeNode currentNode;
5     private QueueInterface queue; // <<===== From Earlier Lecture !!!
6
7     public TreeIterator(BinaryTreeBasis bTree)
8     {
9         binTree = bTree;
10        currentNode = null;
11        // empty queue indicates no traversal type currently
12        // selected or end of current traversal has been reached
13        queue = new QueueReferenceBased();
14    } // end constructor
15
16    public boolean hasNext()
17    {
18        return !queue.isEmpty();
19    } // end hasNext
20
21    public Object next() throws java.util.NoSuchElementException
22    {
23        try
24        {
25            currentNode = (TreeNode)queue.dequeue();
26            return currentNode.getItem();
27        } // end try
28        catch (QueueException e)
29        {
30            throw new java.util.NoSuchElementException();
31        } // end catch
32    } // end next
33
34    public void remove() throws UnsupportedOperationException
35    {
36        throw new UnsupportedOperationException();
37    } // end remove
38
39    public void setPreorder()
40    {
41        queue.dequeueAll();
42    }
43
44    public void setInorder()
45    {
46        queue.enqueueAll();
47    }
48
49    public void setPostorder()
50    {
51        queue.enqueueAll();
52    }
53
54    public void setLevelOrder()
55    {
56        queue.enqueueAll();
57    }
58
59    public void setDepthFirst()
60    {
61        queue.enqueueAll();
62    }
63
64    public void setBFS()
65    {
66        queue.enqueueAll();
67    }
68
69    public void setDFS()
70    {
71        queue.enqueueAll();
72    }
73
74    public void setLRU()
75    {
76        queue.enqueueAll();
77    }
78
79    public void setLRU()
80    {
81        queue.enqueueAll();
82    }
83
84    public void setLRU()
85    {
86        queue.enqueueAll();
87    }
88
89    public void setLRU()
90    {
91        queue.enqueueAll();
92    }
93
94    public void setLRU()
95    {
96        queue.enqueueAll();
97    }
98
99    public void setLRU()
100   {
101      queue.enqueueAll();
102  }
103
104  public void setLRU()
105  {
106      queue.enqueueAll();
107  }
108
109  public void setLRU()
110  {
111      queue.enqueueAll();
112  }
113
114  public void setLRU()
115  {
116      queue.enqueueAll();
117  }
118
119  public void setLRU()
120  {
121      queue.enqueueAll();
122  }
123
124  public void setLRU()
125  {
126      queue.enqueueAll();
127  }
128
129  public void setLRU()
130  {
131      queue.enqueueAll();
132  }
133
134  public void setLRU()
135  {
136      queue.enqueueAll();
137  }
138
139  public void setLRU()
140  {
141      queue.enqueueAll();
142  }
143
144  public void setLRU()
145  {
146      queue.enqueueAll();
147  }
148
149  public void setLRU()
150  {
151      queue.enqueueAll();
152  }
153
154  public void setLRU()
155  {
156      queue.enqueueAll();
157  }
158
159  public void setLRU()
160  {
161      queue.enqueueAll();
162  }
163
164  public void setLRU()
165  {
166      queue.enqueueAll();
167  }
168
169  public void setLRU()
170  {
171      queue.enqueueAll();
172  }
173
174  public void setLRU()
175  {
176      queue.enqueueAll();
177  }
178
179  public void setLRU()
180  {
181      queue.enqueueAll();
182  }
183
184  public void setLRU()
185  {
186      queue.enqueueAll();
187  }
188
189  public void setLRU()
190  {
191      queue.enqueueAll();
192  }
193
194  public void setLRU()
195  {
196      queue.enqueueAll();
197  }
198
199  public void setLRU()
200  {
201      queue.enqueueAll();
202  }
203
204  public void setLRU()
205  {
206      queue.enqueueAll();
207  }
208
209  public void setLRU()
210  {
211      queue.enqueueAll();
212  }
213
214  public void setLRU()
215  {
216      queue.enqueueAll();
217  }
218
219  public void setLRU()
220  {
221      queue.enqueueAll();
222  }
223
224  public void setLRU()
225  {
226      queue.enqueueAll();
227  }
228
229  public void setLRU()
230  {
231      queue.enqueueAll();
232  }
233
234  public void setLRU()
235  {
236      queue.enqueueAll();
237  }
238
239  public void setLRU()
240  {
241      queue.enqueueAll();
242  }
243
244  public void setLRU()
245  {
246      queue.enqueueAll();
247  }
248
249  public void setLRU()
250  {
251      queue.enqueueAll();
252  }
253
254  public void setLRU()
255  {
256      queue.enqueueAll();
257  }
258
259  public void setLRU()
260  {
261      queue.enqueueAll();
262  }
263
264  public void setLRU()
265  {
266      queue.enqueueAll();
267  }
268
269  public void setLRU()
270  {
271      queue.enqueueAll();
272  }
273
274  public void setLRU()
275  {
276      queue.enqueueAll();
277  }
278
279  public void setLRU()
280  {
281      queue.enqueueAll();
282  }
283
284  public void setLRU()
285  {
286      queue.enqueueAll();
287  }
288
289  public void setLRU()
290  {
291      queue.enqueueAll();
292  }
293
294  public void setLRU()
295  {
296      queue.enqueueAll();
297  }
298
299  public void setLRU()
300  {
301      queue.enqueueAll();
302  }
303
304  public void setLRU()
305  {
306      queue.enqueueAll();
307  }
308
309  public void setLRU()
310  {
311      queue.enqueueAll();
312  }
313
314  public void setLRU()
315  {
316      queue.enqueueAll();
317  }
318
319  public void setLRU()
320  {
321      queue.enqueueAll();
322  }
323
324  public void setLRU()
325  {
326      queue.enqueueAll();
327  }
328
329  public void setLRU()
330  {
331      queue.enqueueAll();
332  }
333
334  public void setLRU()
335  {
336      queue.enqueueAll();
337  }
338
339  public void setLRU()
340  {
341      queue.enqueueAll();
342  }
343
344  public void setLRU()
345  {
346      queue.enqueueAll();
347  }
348
349  public void setLRU()
350  {
351      queue.enqueueAll();
352  }
353
354  public void setLRU()
355  {
356      queue.enqueueAll();
357  }
358
359  public void setLRU()
360  {
361      queue.enqueueAll();
362  }
363
364  public void setLRU()
365  {
366      queue.enqueueAll();
367  }
368
369  public void setLRU()
370  {
371      queue.enqueueAll();
372  }
373
374  public void setLRU()
375  {
376      queue.enqueueAll();
377  }
378
379  public void setLRU()
380  {
381      queue.enqueueAll();
382  }
383
384  public void setLRU()
385  {
386      queue.enqueueAll();
387  }
388
389  public void setLRU()
390  {
391      queue.enqueueAll();
392  }
393
394  public void setLRU()
395  {
396      queue.enqueueAll();
397  }
398
399  public void setLRU()
400  {
401      queue.enqueueAll();
402  }
403
404  public void setLRU()
405  {
406      queue.enqueueAll();
407  }
408
409  public void setLRU()
410  {
411      queue.enqueueAll();
412  }
413
414  public void setLRU()
415  {
416      queue.enqueueAll();
417  }
418
419  public void setLRU()
420  {
421      queue.enqueueAll();
422  }
423
424  public void setLRU()
425  {
426      queue.enqueueAll();
427  }
428
429  public void setLRU()
430  {
431      queue.enqueueAll();
432  }
433
434  public void setLRU()
435  {
436      queue.enqueueAll();
437  }
438
439  public void setLRU()
440  {
441      queue.enqueueAll();
442  }
443
444  public void setLRU()
445  {
446      queue.enqueueAll();
447  }
448
449  public void setLRU()
450  {
451      queue.enqueueAll();
452  }
453
454  public void setLRU()
455  {
456      queue.enqueueAll();
457  }
458
459  public void setLRU()
460  {
461      queue.enqueueAll();
462  }
463
464  public void setLRU()
465  {
466      queue.enqueueAll();
467  }
468
469  public void setLRU()
470  {
471      queue.enqueueAll();
472  }
473
474  public void setLRU()
475  {
476      queue.enqueueAll();
477  }
478
479  public void setLRU()
480  {
481      queue.enqueueAll();
482  }
483
484  public void setLRU()
485  {
486      queue.enqueueAll();
487  }
488
489  public void setLRU()
490  {
491      queue.enqueueAll();
492  }
493
494  public void setLRU()
495  {
496      queue.enqueueAll();
497  }
498
499  public void setLRU()
500  {
501      queue.enqueueAll();
502  }
503
504  public void setLRU()
505  {
506      queue.enqueueAll();
507  }
508
509  public void setLRU()
510  {
511      queue.enqueueAll();
512  }
513
514  public void setLRU()
515  {
516      queue.enqueueAll();
517  }
518
519  public void setLRU()
520  {
521      queue.enqueueAll();
522  }
523
524  public void setLRU()
525  {
526      queue.enqueueAll();
527  }
528
529  public void setLRU()
530  {
531      queue.enqueueAll();
532  }
533
534  public void setLRU()
535  {
536      queue.enqueueAll();
537  }
538
539  public void setLRU()
540  {
541      queue.enqueueAll();
542  }
543
544  public void setLRU()
545  {
546      queue.enqueueAll();
547  }
548
549  public void setLRU()
550  {
551      queue.enqueueAll();
552  }
553
554  public void setLRU()
555  {
556      queue.enqueueAll();
557  }
558
559  public void setLRU()
560  {
561      queue.enqueueAll();
562  }
563
564  public void setLRU()
565  {
566      queue.enqueueAll();
567  }
568
569  public void setLRU()
570  {
571      queue.enqueueAll();
572  }
573
574  public void setLRU()
575  {
576      queue.enqueueAll();
577  }
578
579  public void setLRU()
580  {
581      queue.enqueueAll();
582  }
583
584  public void setLRU()
585  {
586      queue.enqueueAll();
587  }
588
589  public void setLRU()
590  {
591      queue.enqueueAll();
592  }
593
594  public void setLRU()
595  {
596      queue.enqueueAll();
597  }
598
599  public void setLRU()
600  {
601      queue.enqueueAll();
602  }
603
604  public void setLRU()
605  {
606      queue.enqueueAll();
607  }
608
609  public void setLRU()
610  {
611      queue.enqueueAll();
612  }
613
614  public void setLRU()
615  {
616      queue.enqueueAll();
617  }
618
619  public void setLRU()
620  {
621      queue.enqueueAll();
622  }
623
624  public void setLRU()
625  {
626      queue.enqueueAll();
627  }
628
629  public void setLRU()
630  {
631      queue.enqueueAll();
632  }
633
634  public void setLRU()
635  {
636      queue.enqueueAll();
637  }
638
639  public void setLRU()
640  {
641      queue.enqueueAll();
642  }
643
644  public void setLRU()
645  {
646      queue.enqueueAll();
647  }
648
649  public void setLRU()
650  {
651      queue.enqueueAll();
652  }
653
654  public void setLRU()
655  {
656      queue.enqueueAll();
657  }
658
659  public void setLRU()
660  {
661      queue.enqueueAll();
662  }
663
664  public void setLRU()
665  {
666      queue.enqueueAll();
667  }
668
669  public void setLRU()
670  {
671      queue.enqueueAll();
672  }
673
674  public void setLRU()
675  {
676      queue.enqueueAll();
677  }
678
679  public void setLRU()
680  {
681      queue.enqueueAll();
682  }
683
684  public void setLRU()
685  {
686      queue.enqueueAll();
687  }
688
689  public void setLRU()
690  {
691      queue.enqueueAll();
692  }
693
694  public void setLRU()
695  {
696      queue.enqueueAll();
697  }
698
699  public void setLRU()
700  {
701      queue.enqueueAll();
702  }
703
704  public void setLRU()
705  {
706      queue.enqueueAll();
707  }
708
709  public void setLRU()
710  {
711      queue.enqueueAll();
712  }
713
714  public void setLRU()
715  {
716      queue.enqueueAll();
717  }
718
719  public void setLRU()
720  {
721      queue.enqueueAll();
722  }
723
724  public void setLRU()
725  {
726      queue.enqueueAll();
727  }
728
729  public void setLRU()
730  {
731      queue.enqueueAll();
732  }
733
734  public void setLRU()
735  {
736      queue.enqueueAll();
737  }
738
739  public void setLRU()
740  {
741      queue.enqueueAll();
742  }
743
744  public void setLRU()
745  {
746      queue.enqueueAll();
747  }
748
749  public void setLRU()
750  {
751      queue.enqueueAll();
752  }
753
754  public void setLRU()
755  {
756      queue.enqueueAll();
757  }
758
759  public void setLRU()
760  {
761      queue.enqueueAll();
762  }
763
764  public void setLRU()
765  {
766      queue.enqueueAll();
767  }
768
769  public void setLRU()
770  {
771      queue.enqueueAll();
772  }
773
774  public void setLRU()
775  {
776      queue.enqueueAll();
777  }
778
779  public void setLRU()
780  {
781      queue.enqueueAll();
782  }
783
784  public void setLRU()
785  {
786      queue.enqueueAll();
787  }
788
789  public void setLRU()
790  {
791      queue.enqueueAll();
792  }
793
794  public void setLRU()
795  {
796      queue.enqueueAll();
797  }
798
799  public void setLRU()
800  {
801      queue.enqueueAll();
802  }
803
804  public void setLRU()
805  {
806      queue.enqueueAll();
807  }
808
809  public void setLRU()
810  {
811      queue.enqueueAll();
812  }
813
814  public void setLRU()
815  {
816      queue.enqueueAll();
817  }
818
819  public void setLRU()
820  {
821      queue.enqueueAll();
822  }
823
824  public void setLRU()
825  {
826      queue.enqueueAll();
827  }
828
829  public void setLRU()
830  {
831      queue.enqueueAll();
832  }
833
834  public void setLRU()
835  {
836      queue.enqueueAll();
837  }
838
839  public void setLRU()
840  {
841      queue.enqueueAll();
842  }
843
844  public void setLRU()
845  {
846      queue.enqueueAll();
847  }
848
849  public void setLRU()
850  {
851      queue.enqueueAll();
852  }
853
854  public void setLRU()
855  {
856      queue.enqueueAll();
857  }
858
859  public void setLRU()
860  {
861      queue.enqueueAll();
862  }
863
864  public void setLRU()
865  {
866      queue.enqueueAll();
867  }
868
869  public void setLRU()
870  {
871      queue.enqueueAll();
872  }
873
874  public void setLRU()
875  {
876      queue.enqueueAll();
877  }
878
879  public void setLRU()
880  {
881      queue.enqueueAll();
882  }
883
884  public void setLRU()
885  {
886      queue.enqueueAll();
887  }
888
889  public void setLRU()
890  {
891      queue.enqueueAll();
892  }
893
894  public void setLRU()
895  {
896      queue.enqueueAll();
897  }
898
899  public void setLRU()
900  {
901      queue.enqueueAll();
902  }
903
904  public void setLRU()
905  {
906      queue.enqueueAll();
907  }
908
909  public void setLRU()
910  {
911      queue.enqueueAll();
912  }
913
914  public void setLRU()
915  {
916      queue.enqueueAll();
917  }
918
919  public void setLRU()
920  {
921      queue.enqueueAll();
922  }
923
924  public void setLRU()
925  {
926      queue.enqueueAll();
927  }
928
929  public void setLRU()
930  {
931      queue.enqueueAll();
932  }
933
934  public void setLRU()
935  {
936      queue.enqueueAll();
937  }
938
939  public void setLRU()
940  {
941      queue.enqueueAll();
942  }
943
944  public void setLRU()
945  {
946      queue.enqueueAll();
947  }
948
949  public void setLRU()
950  {
951      queue.enqueueAll();
952  }
953
954  public void setLRU()
955  {
956      queue.enqueueAll();
957  }
958
959  public void setLRU()
960  {
961      queue.enqueueAll();
962  }
963
964  public void setLRU()
965  {
966      queue.enqueueAll();
967  }
968
969  public void setLRU()
970  {
971      queue.enqueueAll();
972  }
973
974  public void setLRU()
975  {
976      queue.enqueueAll();
977  }
978
979  public void setLRU()
980  {
981      queue.enqueueAll();
982  }
983
984  public void setLRU()
985  {
986      queue.enqueueAll();
987  }
988
989  public void setLRU()
990  {
991      queue.enqueueAll();
992  }
993
994  public void setLRU()
995  {
996      queue.enqueueAll();
997  }
998
999  public void setLRU()
1000 {
1001     queue.enqueueAll();
1002 }
```

TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\Treeliterator.java]

File Edit Search View Tools Macros Configure Window Help

Command Results Treeliterator.java *

ANSI Characters

```
41     queue.dequeueAll();
42     preorder(binTree.root);
43 } // setPreOrder
44
45 public void setInorder()
46 {
47     queue.dequeueAll();
48     inorder(binTree.root);
49 } // end setInorder
50
51 public void setPostorder()
52 {
53     queue.dequeueAll();
54     postorder(binTree.root);
55 } // end setPostorder
56
57 private void preorder(TreeNode treeNode)
58 {
59     if (treeNode != null)
60     {
61         queue.enqueue(treeNode);
62         preorder(treeNode.getLeft());
63         preorder(treeNode.getRight());
64     } // end if
65 } // end preorder
66
67 private void inorder(TreeNode treeNode)
68 {
69     if (treeNode != null)
70     {
71         inorder(treeNode.getLeft());
72         queue.enqueue(treeNode);
73         inorder(treeNode.getRight());
74     } // end if
75 } // end inorder
76
77 private void postorder(TreeNode treeNode)
78 {
79     if (treeNode != null)
80     {
81         postorder(treeNode.getLeft());
```

81 37 Read Ovr Block Sync Rec Caps 23:53

TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeIterator.java]

File Edit Search View Tools Macros Configure Window Help

Command Results

TreeIterator.java *

```
76
77     private void postorder(TreeNode treeNode) ←
78     {
79         if (treeNode != null)
80         {
81             postorder(treeNode.getLeft());
82             postorder(treeNode.getRight());
83             queue.enqueue(treeNode);
84         } // end if
85     } // end postorder
86 } // end TreeIterator
```

ANSI Characters

33	!
34	"
35	#
36	\$
37	%
38	&
39	'
40	(
41)
42	*
43	+
44	-
45	-
46	.
47	/
48	0

For Help, press F1

86 23 Read Ovr Block Sync Rec Caps 23:54

The class **TreeIterator** uses a **queue** to maintain the current traversal of the nodes in the tree.

- This traversal order is placed in a queue when the client selects the **desired traversal method**
- If a new traversal is set in the middle of the iteration, the queue is cleared first, and then the new traversal is generated.

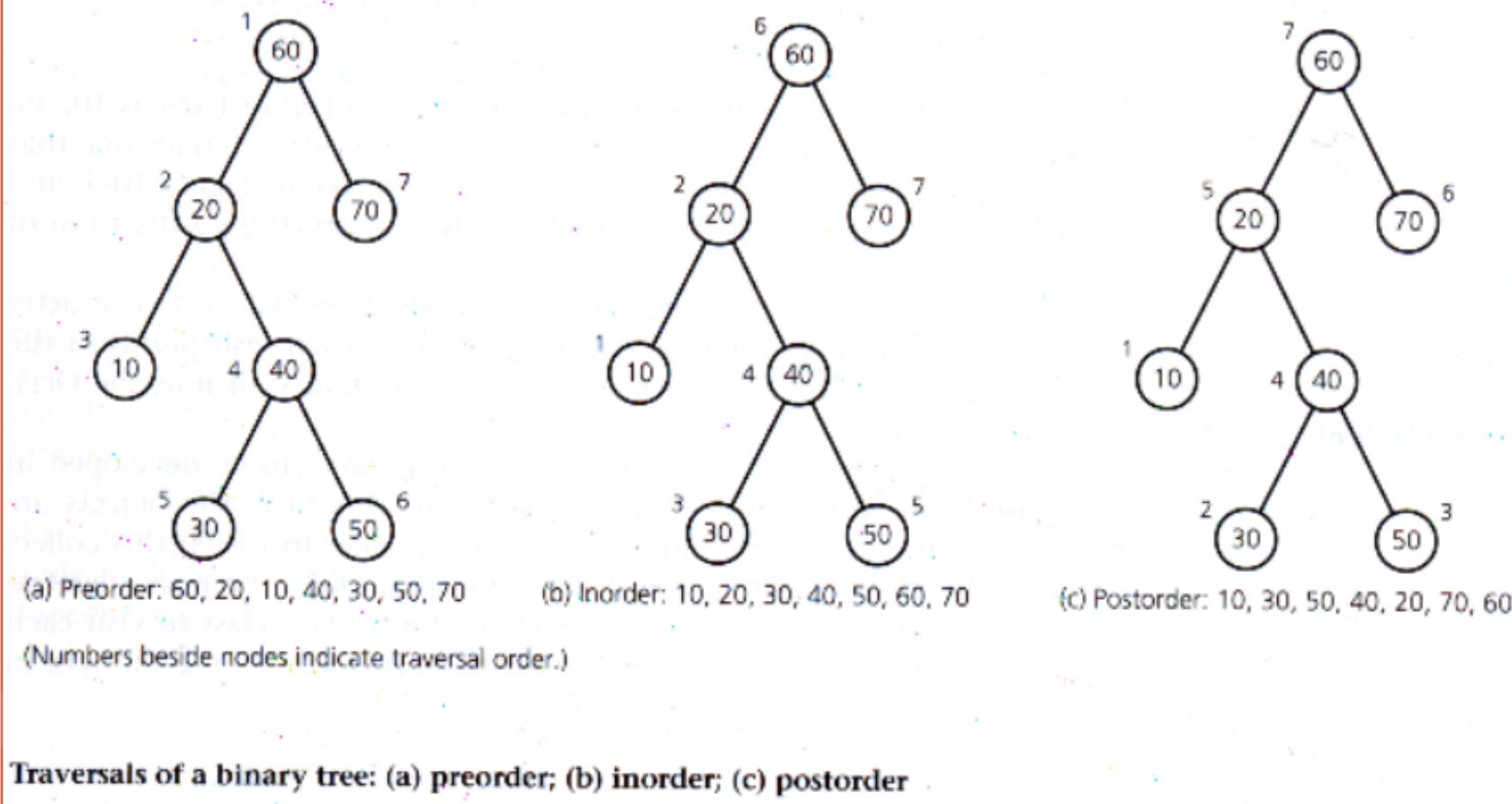
The following statements create an iterator that will perform a preorder traversal of a tree **tree4**:

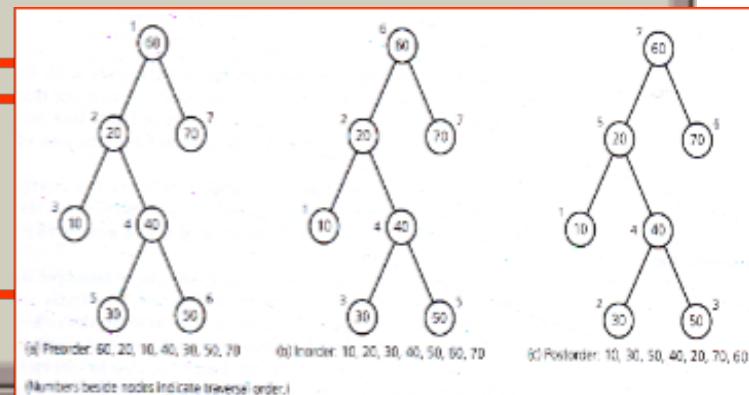
```
TreeIterator treeIterator = new TreeIterator(tree4);
treeIterator.setPreorder();
```

Here is an example that uses the iterator to print out the nodes of the tree using the **preorder** traversal.

```
System.out.println("preorder traversal: ");
while (treeIterator.hasNext())
{
    System.out.println(treeIterator.next());
} //end while
```

To demonstrate how to use **BinaryTree** and **TreeIterator**, we build and then traverse the binary tree in the diagram





Traversals of a binary tree: (a) preorder; (b) inorder; (c) postorder

TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeTest.java]

File Edit Search View Tools Macros Configure Window Help

Command Results

QueueTest.java

Treeliterator.java*

TreeTest.java

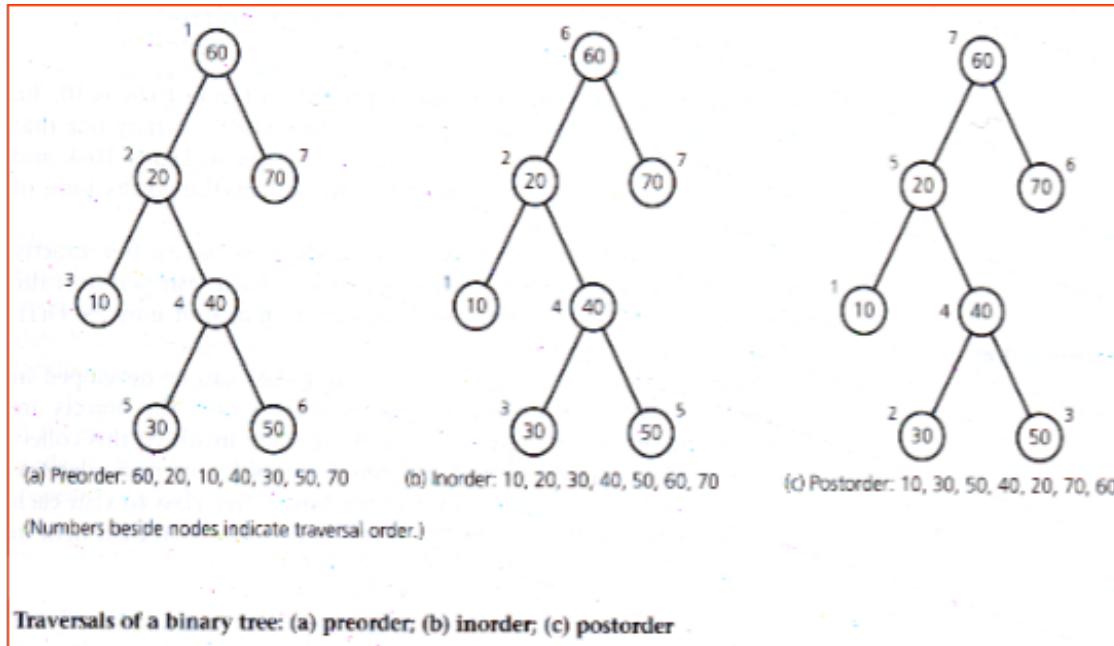
```
41     {
42         System.out.println(btIterator.next());
43     } // end while
44 } // end main
45 }
```

ANSI Characters

33	!
34	"
35	#
36	\$
37	%
38	&
39	'
40	(
41)
42	*
43	+
44	-
45	-
46	.
47	/
48	0

45 2 Read Ovr Block Sync Rec Caps 23:59

Here, `binTree` is the tree in the diagram.



Its **inorder** traversal is 10, 20, 30, 40, 50, 60, 70

The **inorder** traversal of the left subtree of `binTree`'s root (the subtree rooted at 20) is 10, 20, 30, 40.

The **inorder** traversal of `leftTree` produces the same result.

Since `leftTree` is actually detached from `binTree`, the final traversal of `binTree` is 60, 70.

A **disadvantage of this implementation** of the traversal is that it performs a lot of computations that may never be used.

- Not only is a queue of node references created, but also the recursion stores activation records on an implicit stack.
- Besides the time requirement, n additional space is used by the recursion for a tree with n nodes.
- In contrast, the space requirement for a nonrecursive traversal of the tree would be only a function of the **height of the tree**.

Programs to do this week:

1 Create a reference based **ADT Binary Tree**

- List the operations for the ADT **ADT Binary Tree**
- Implement the reference based **ADT Binary Tree** as a Java class
- This class must implement the ADT operations as public methods of the class
- Write a small driver program that demonstrates the **ADT Binary Tree**
- This program should print results to the screen in some simple manner.
Ideally this should be done via a GUI display using a graphical metaphor of the Tree
- Exercise all of the methods to prove that the data is processed correctly
- Show results for **preorder**, **inorder** and **postorder** traversal

Binary Tree - Netscape

File Edit View Go Communicator Help

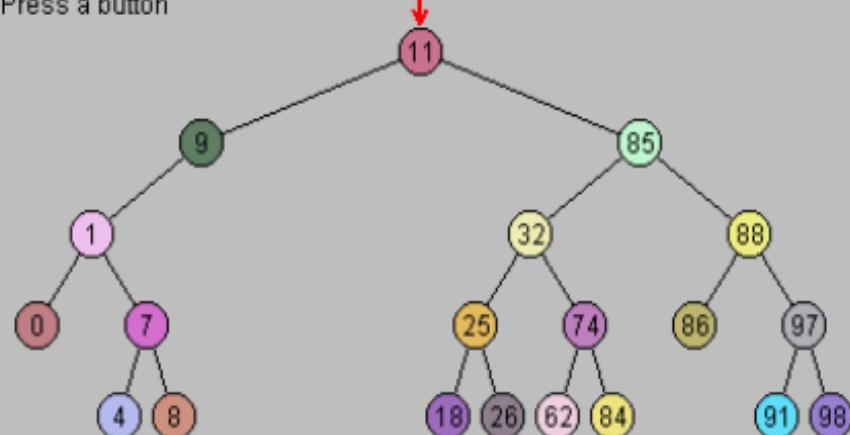


Bookmarks Location: file:///C/ITB/SoftDev-4_JAVA/My-Programs/Tree/Tree.html What's Related

Lafore's Binary Tree

Binary Tree

Press a button



Fill

Find

Ins

Trav

Del

Enter number:

Operation

Fill creates a new tree with N nodes.

Find searches for a node with value N.

Ins inserts a new node with value N.

Trav traverses the tree in ascending order.

Del deletes the node with value N.

(Type N into "Enter number" box.)

Try this program to help you understand the Binary