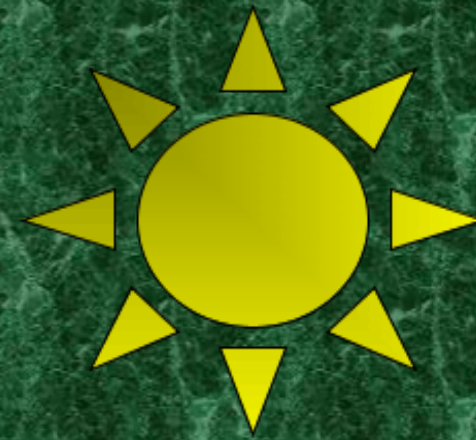
Data Structures and Algorithms

BSc. In Computing

Semester 5 Lecture 5

Lecturer: Dr. Simon McLoughlin

This Week:



Trees

- Terminology

<u>The ADT Binary Tree</u>

- 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 immediatesuccessor.

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 verteces.

The lines between each node are calles**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.

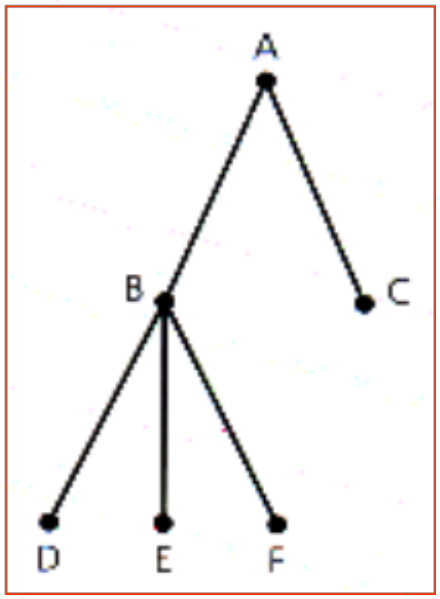
Children of the same parent, B and C are called **siblings**

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.

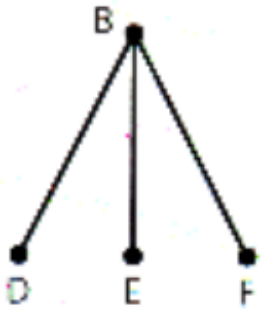
A node that has no children is called a **leaf** of the tree.

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



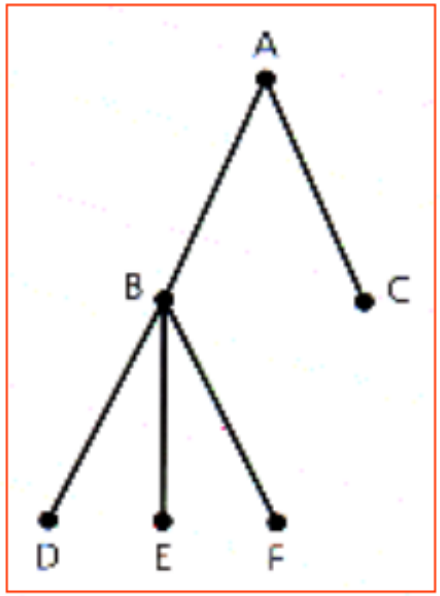
A subtree of the tree

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.

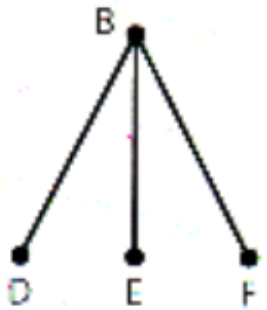
- Not all nodes are related by the ancestor or descendent relationship.

- B and C are not so related



A general tree

A subtree is any node and its descendants



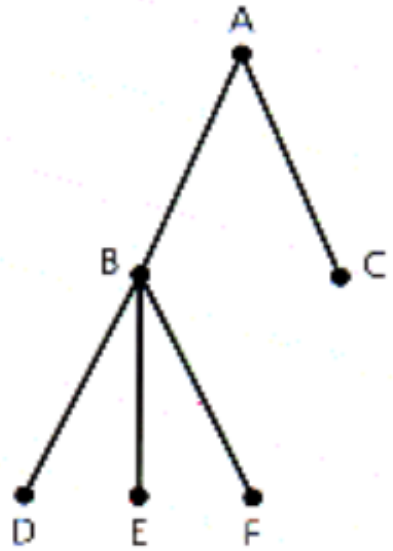
A subtree of the tree

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 descendents.

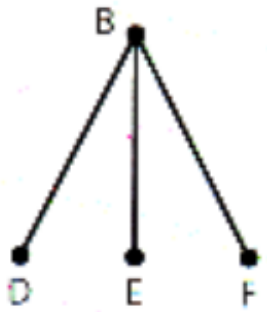
A subtree of a node n is a subtree rooted at a child of n

- 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



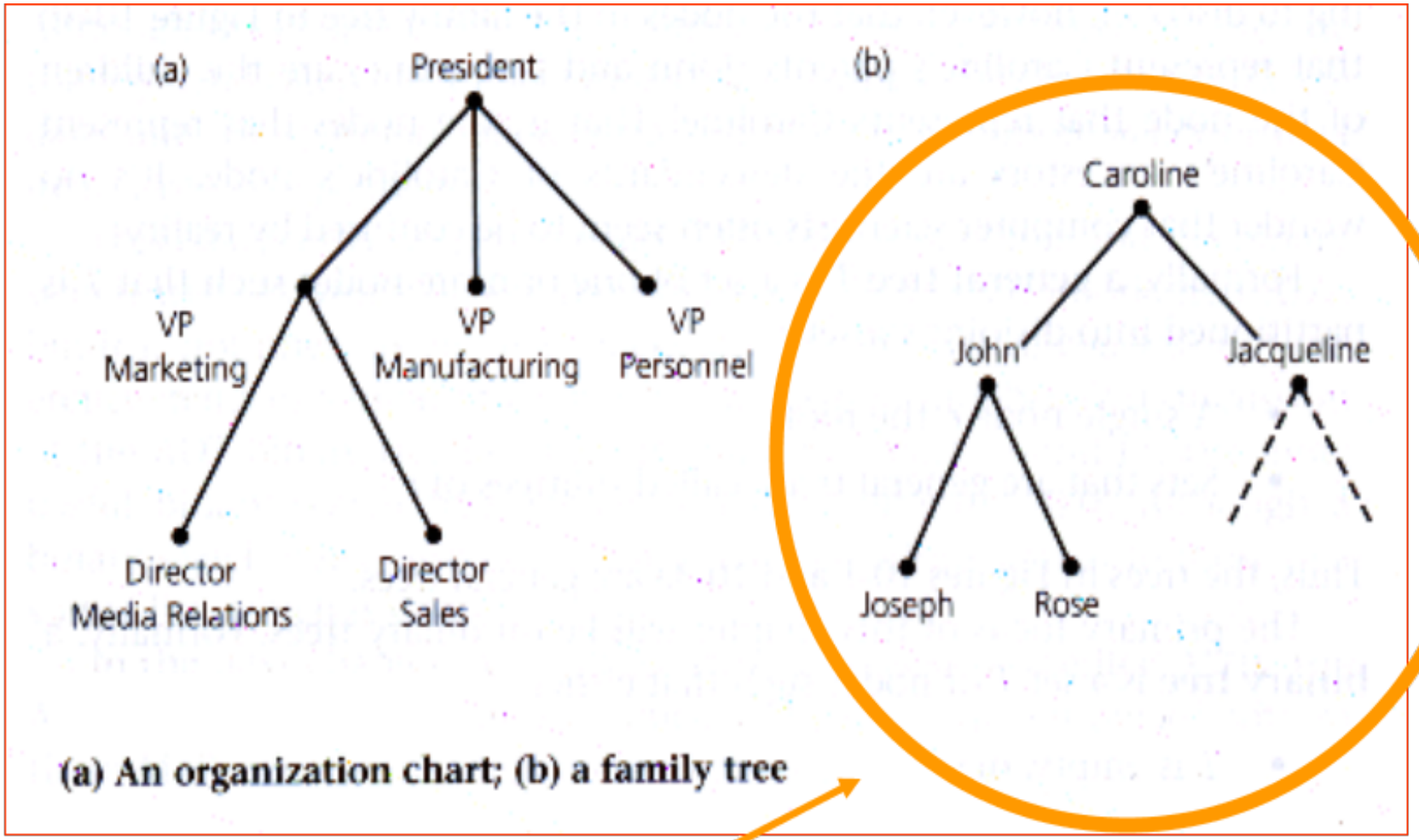
A subtree of the tree

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:

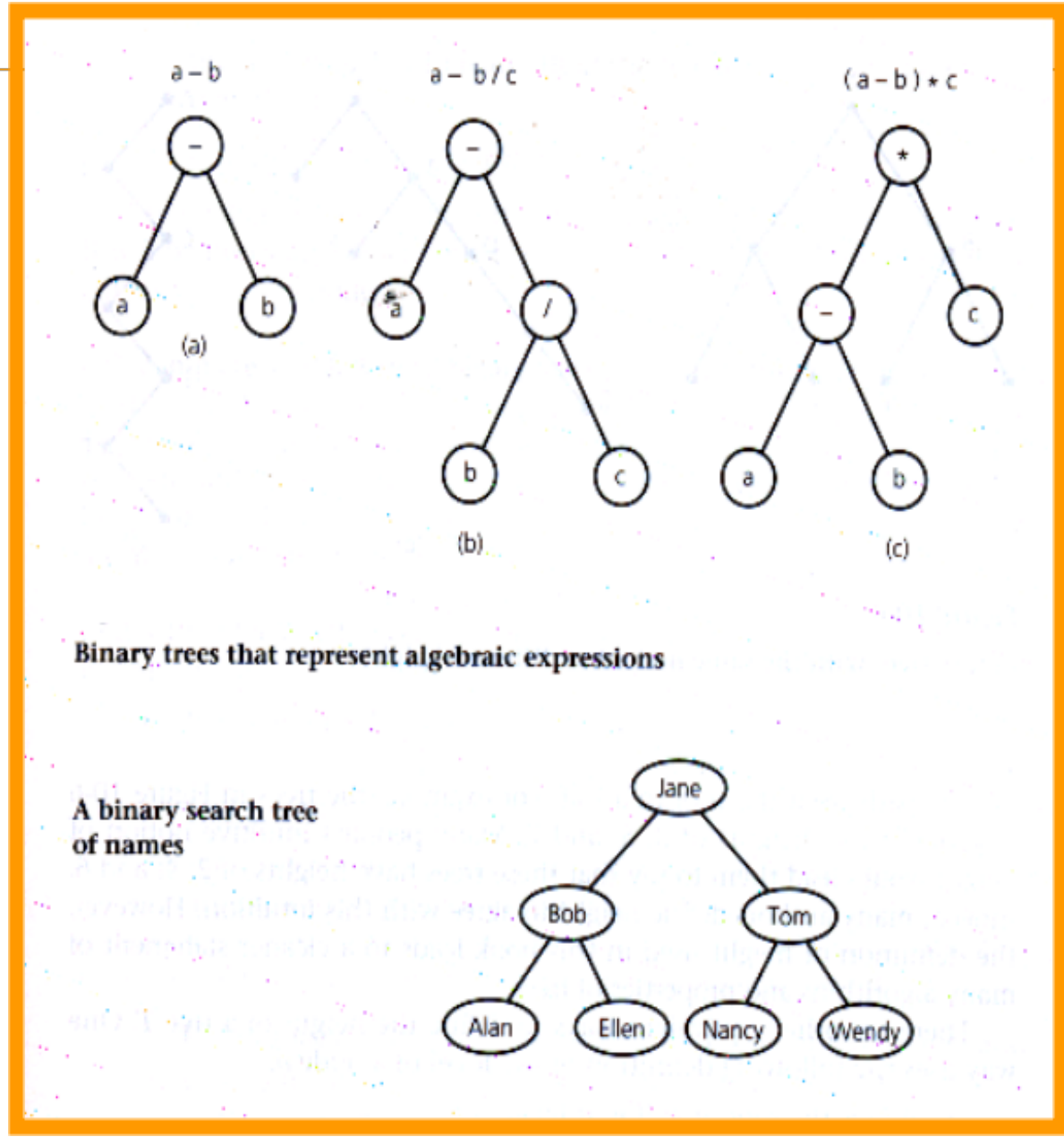
- T is **empty**, or
- T is **partitioned** into three disjoint subsets:
  - A single node **r**, the root
  - Two possibly empty sets that are binary trees, called **left and right subtrees** of **r**



(a) 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 can represent algebraic expressions that involve the binary operators **+, -, *** and **/.**

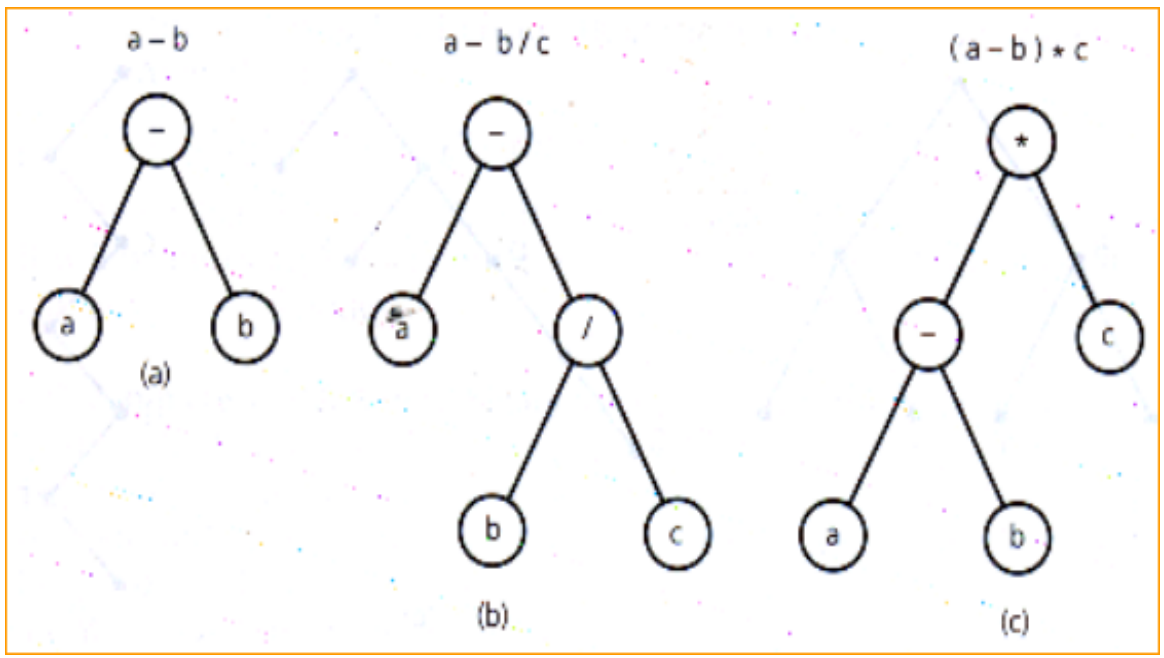
To represent an expression such a – b, we place the operator in the root and the operands in the left and right children of the root, respectively.

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**.

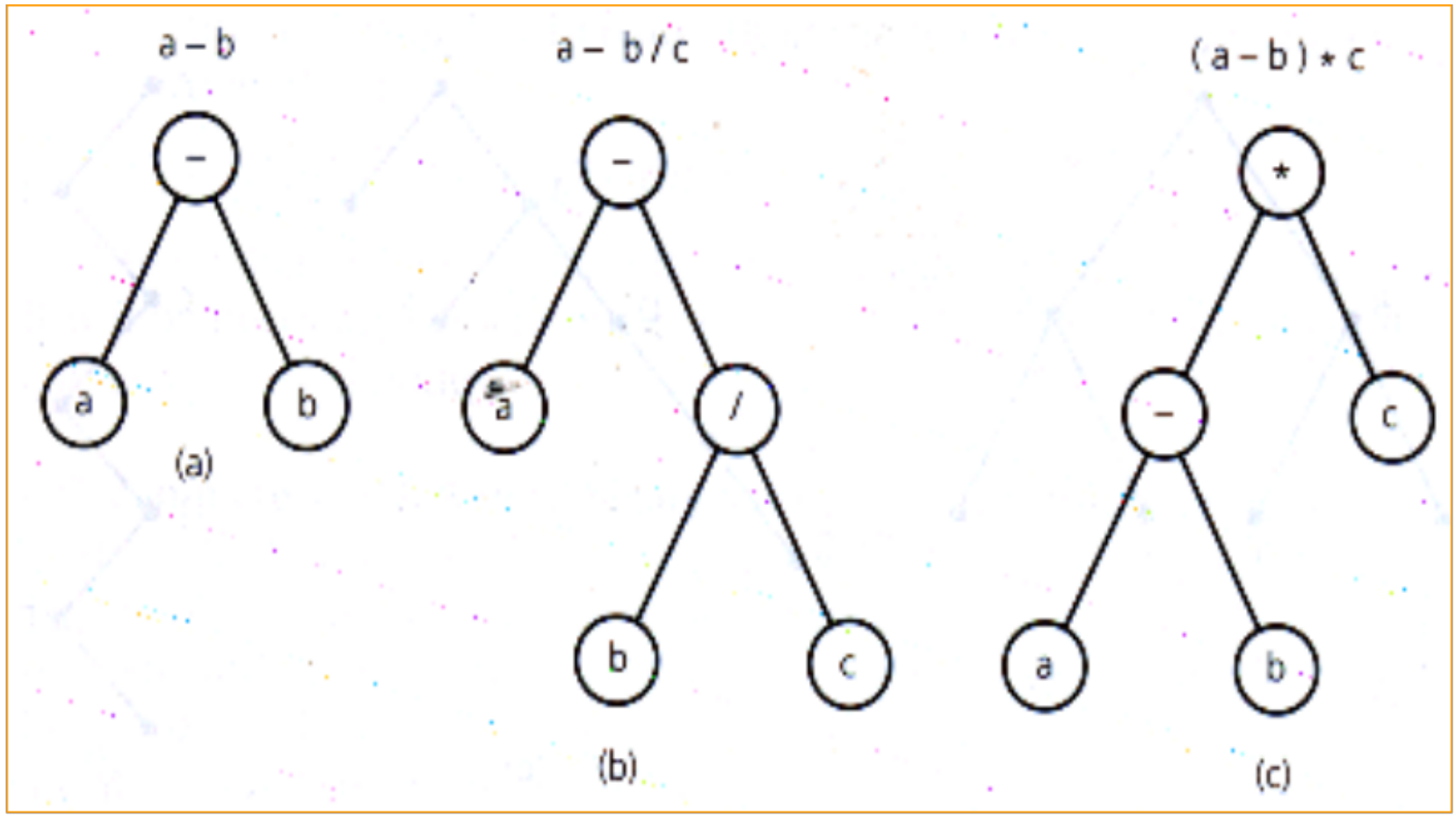
The leaves of these trees contain the expressions' operands, while other tree nodes contain the operators.

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.



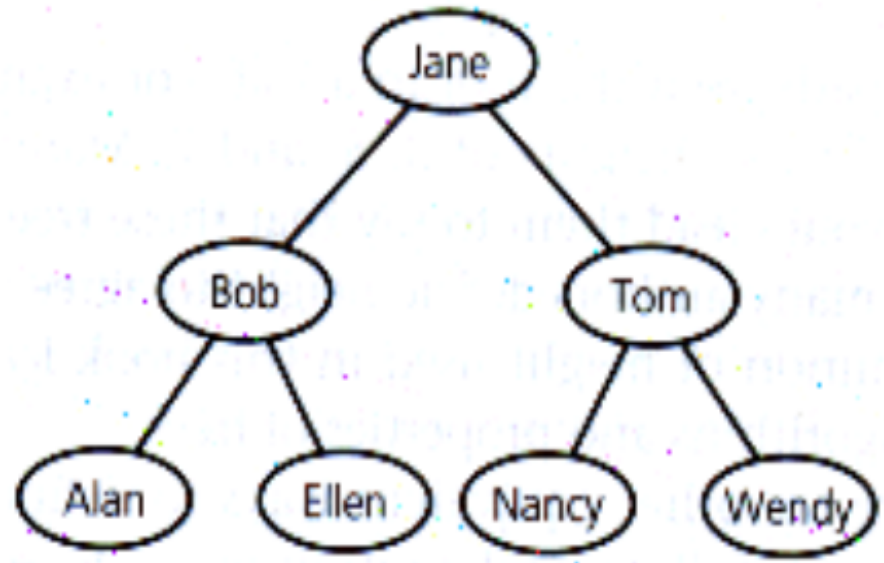
The nodes of a tree typically contain values.

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 $\mathbf{T}_L$
- The value of **n** is less than all values in its right subtree $\mathbf{T}_R$
- Both $\mathbf{T}_L$ and $\mathbf{T}_R$ 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.

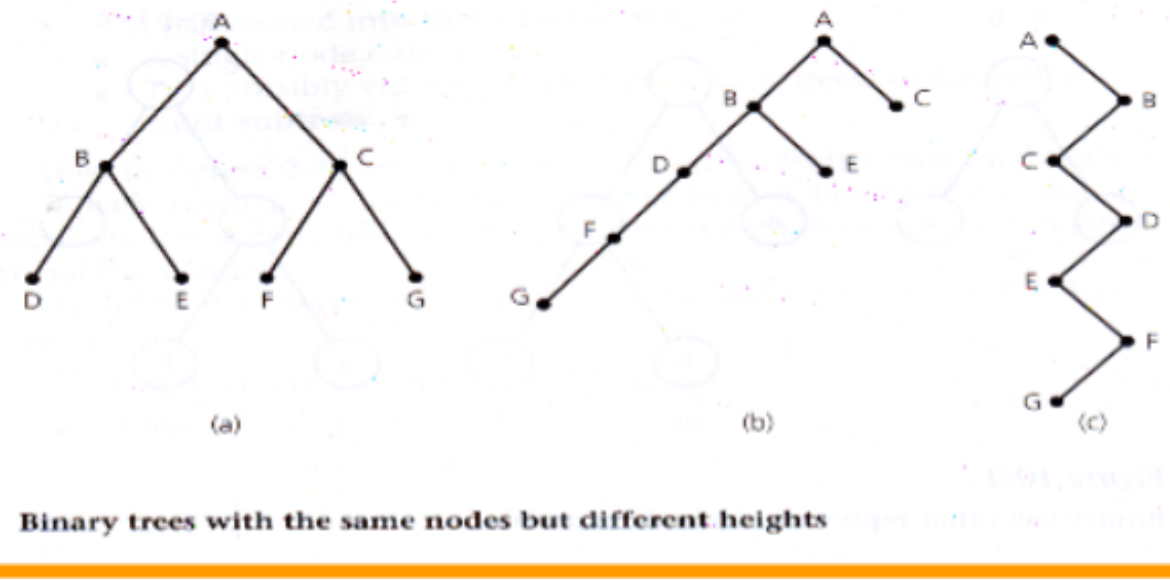
The Height of Trees

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.



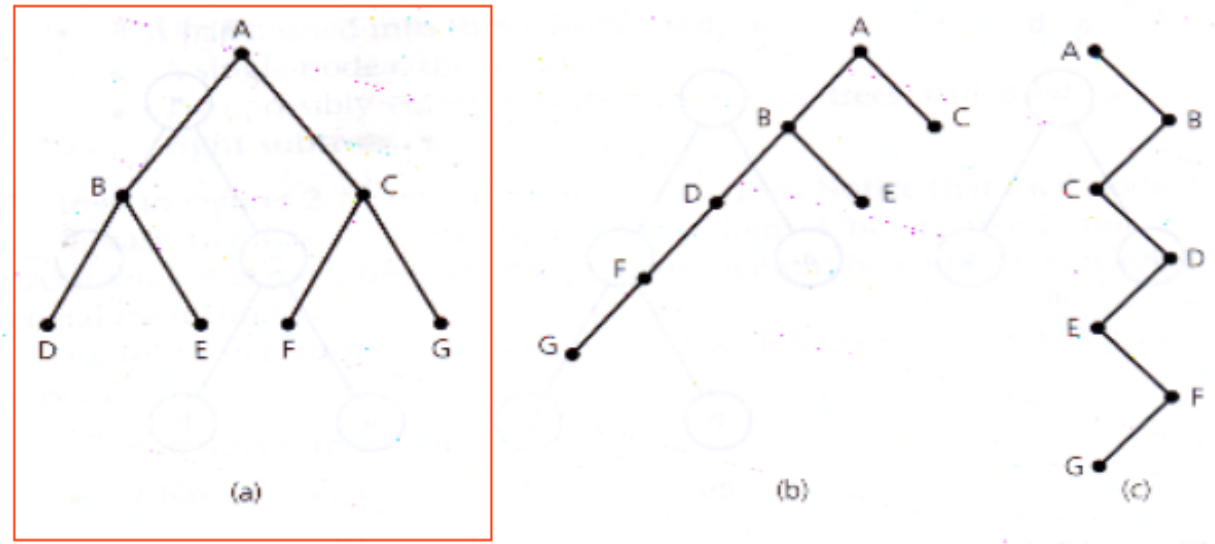
Binary trees with the same nodes but different heights

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

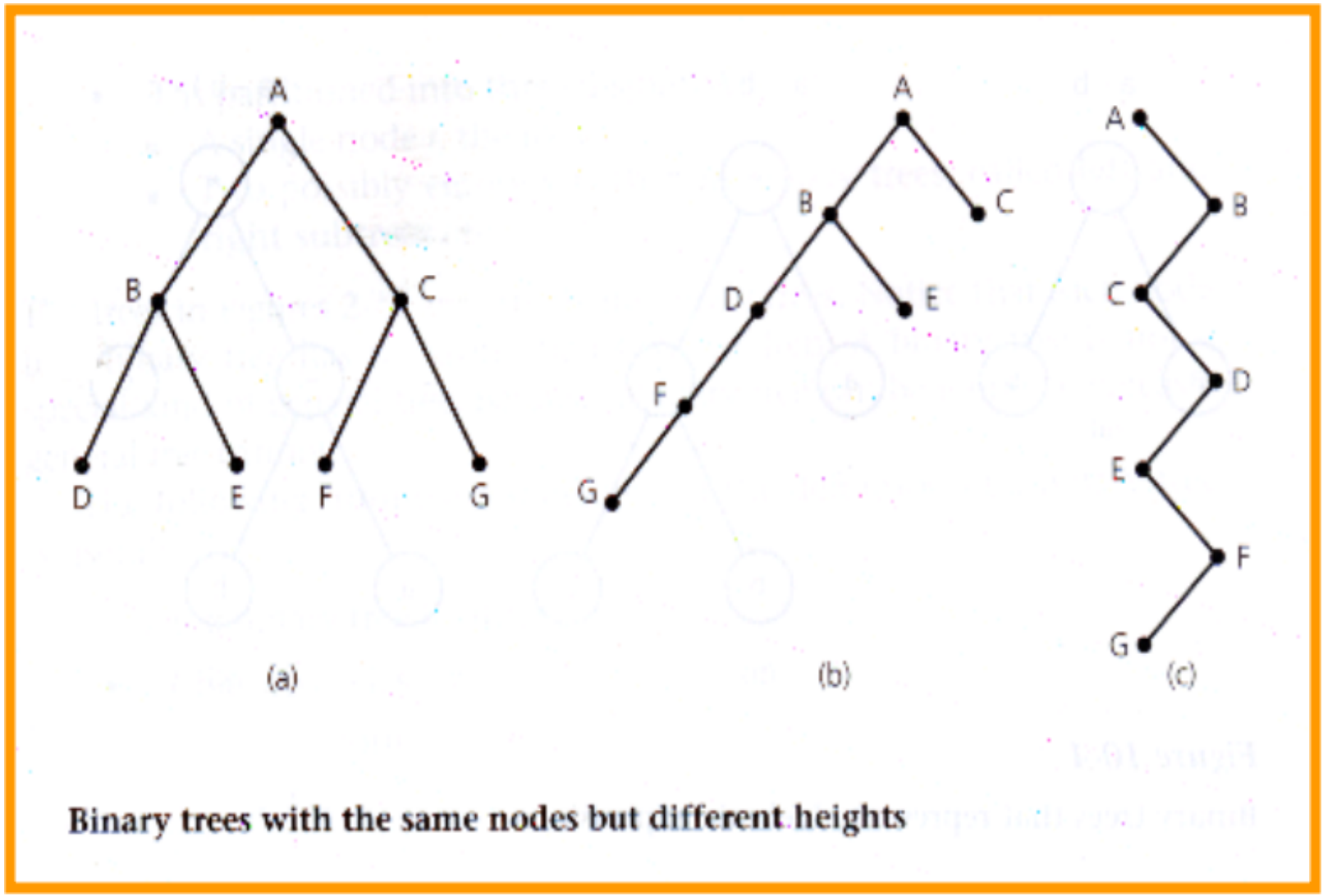
The level of a node in calculating the height

- 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



Binary trees with the same nodes but different heights

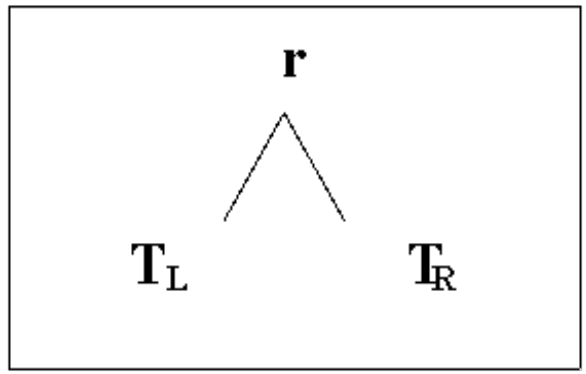
If T is empty, its height is zero

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,

$$height(\mathbf{T}) = 1 + \max\{ height(\mathbf{T}_L), height(\mathbf{T}_R) \}$$

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.

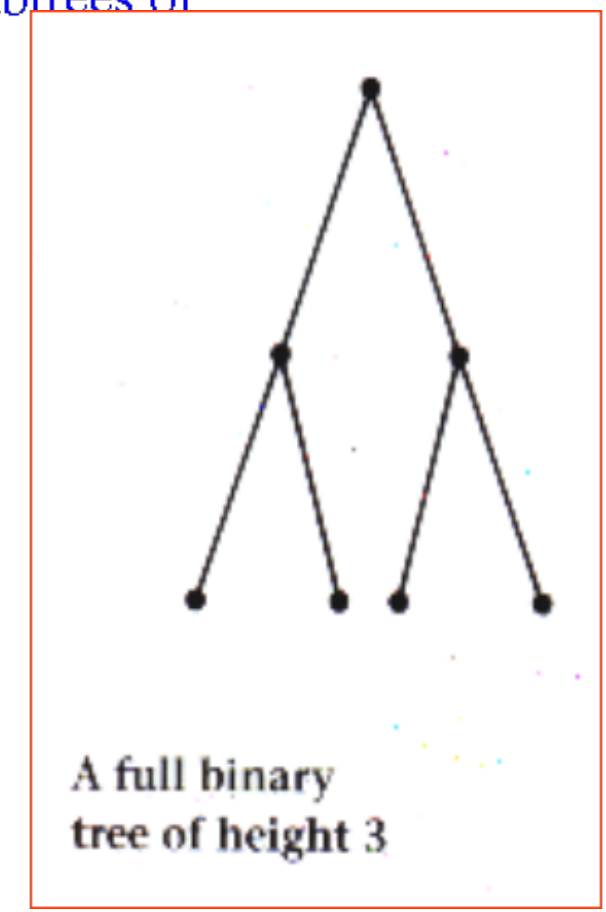
The diagram depicts a full binary tree of height 3.

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.

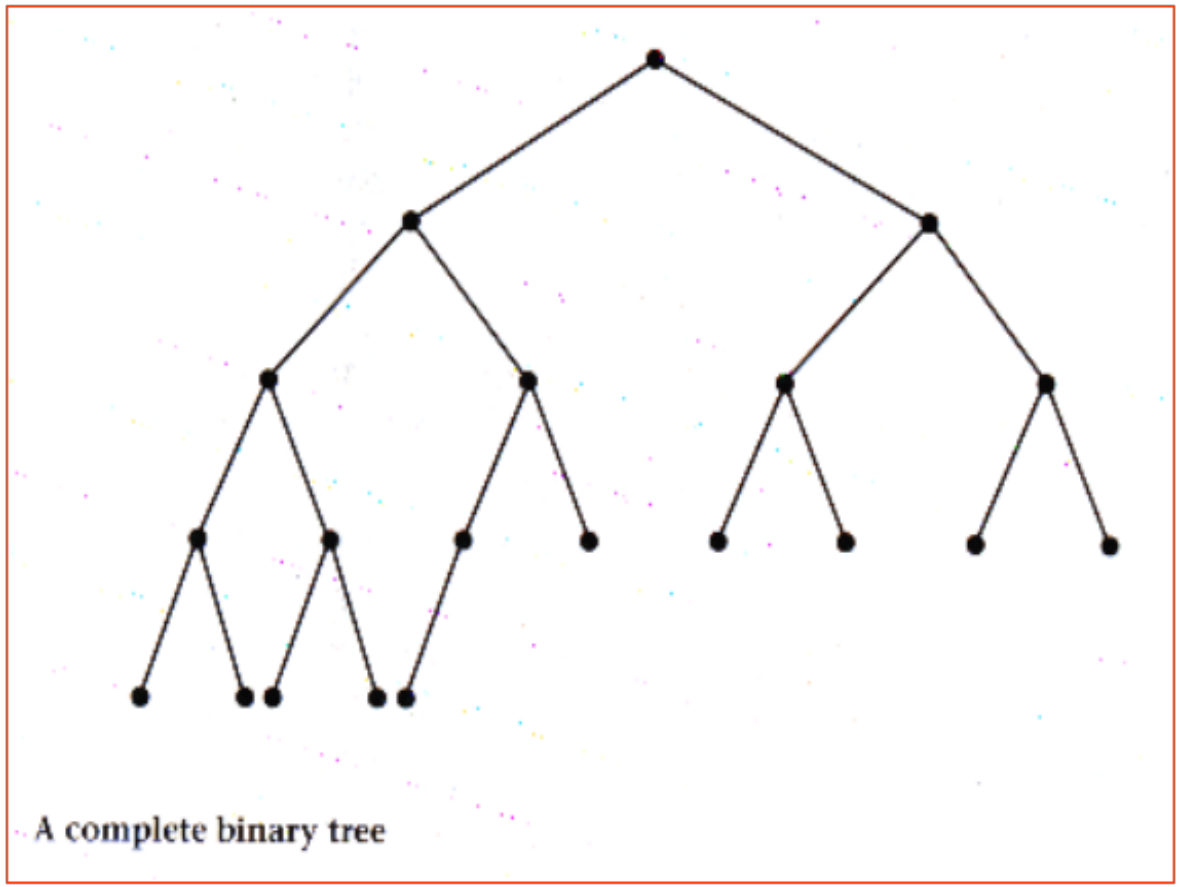


A full binary tree of height 3

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



A complete binary tree

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

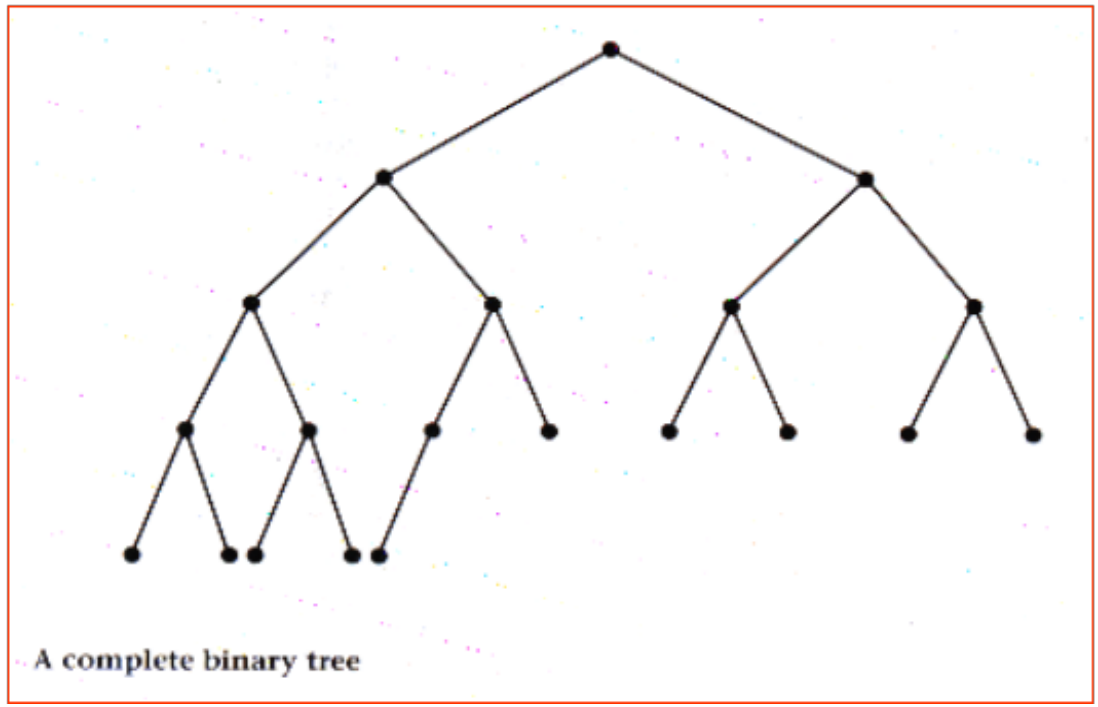
1. All nodes at level h – 2 and above **have two children each** and

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.



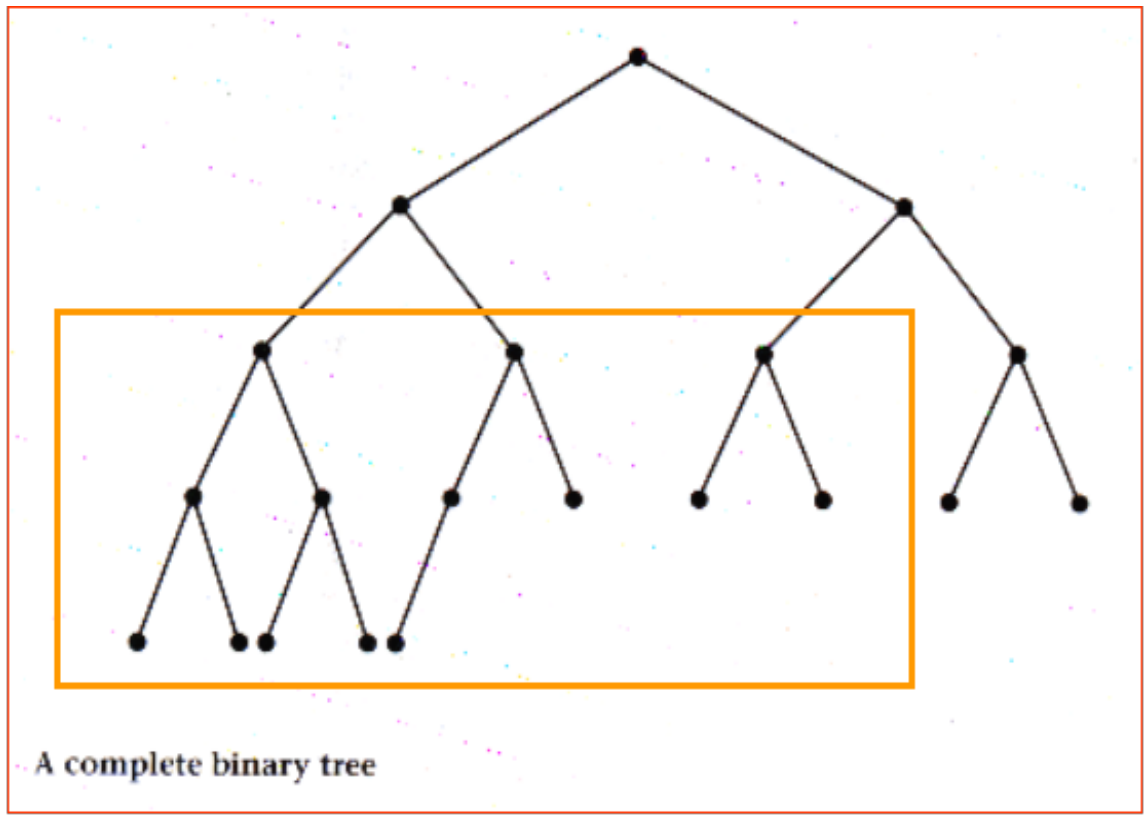
A complete binary tree

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.



A complete binary tree

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 isnewItem 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.

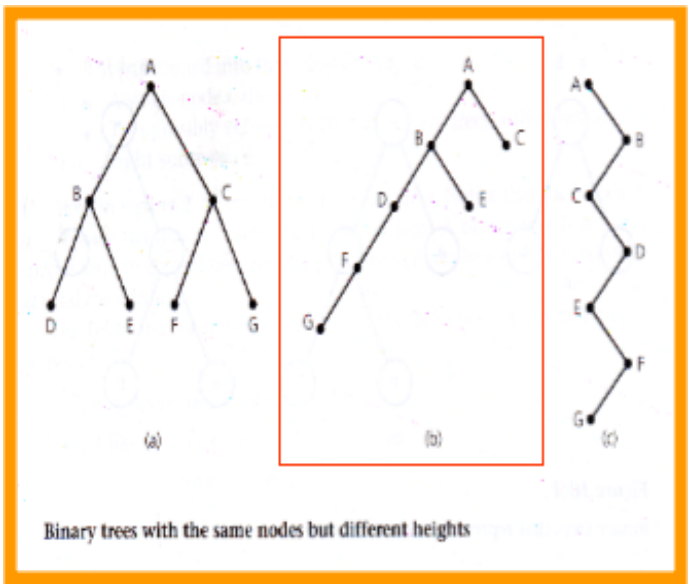
The following pseudocode constructs the tree from the…

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

Initially …

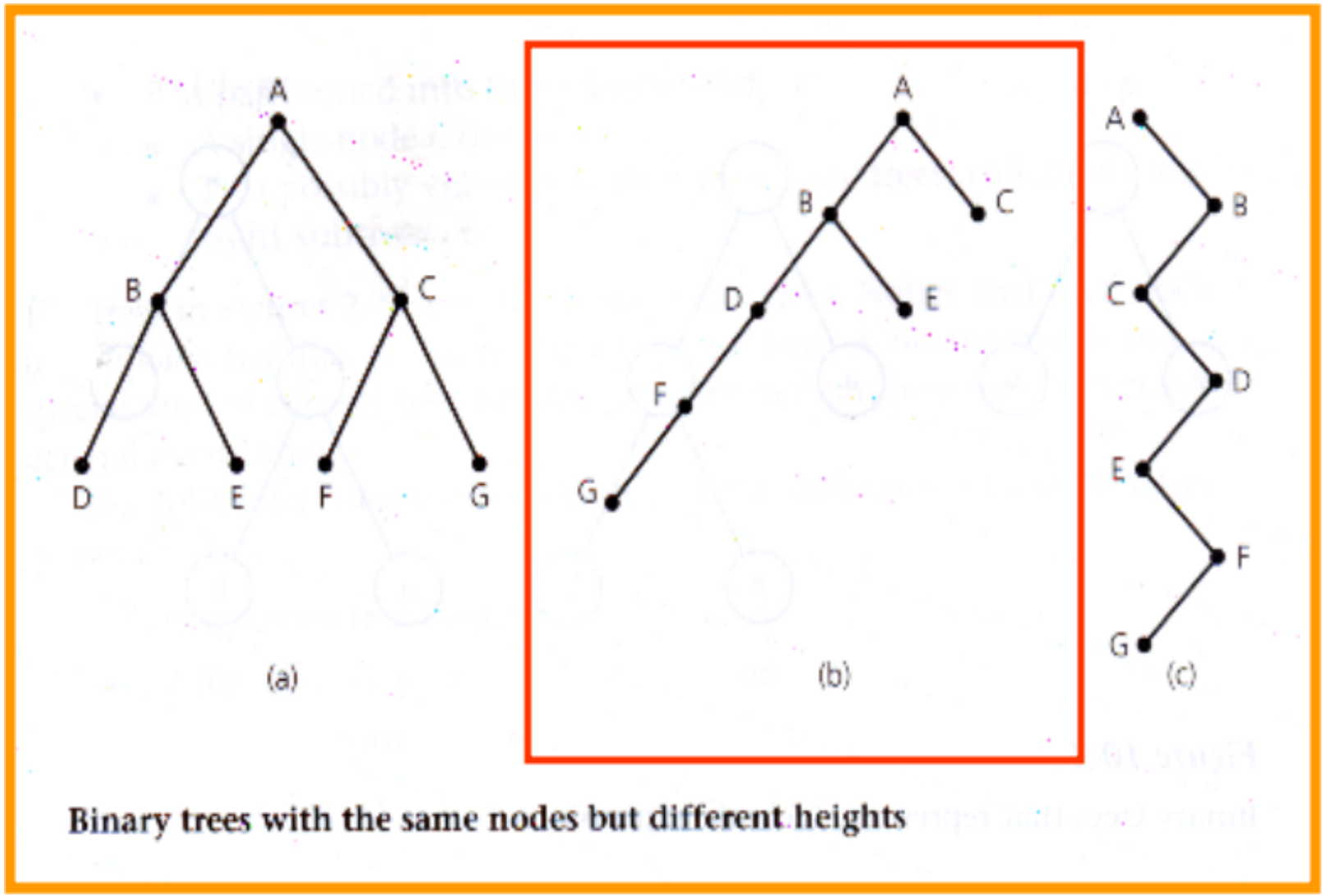
- **subtrees** exist but are **empty**

```
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)
```



Binary trees with the same nodes but different heights

The traversal details are considered next.



Binary trees with the same nodes but different heights

Traversals of a Binary Tree

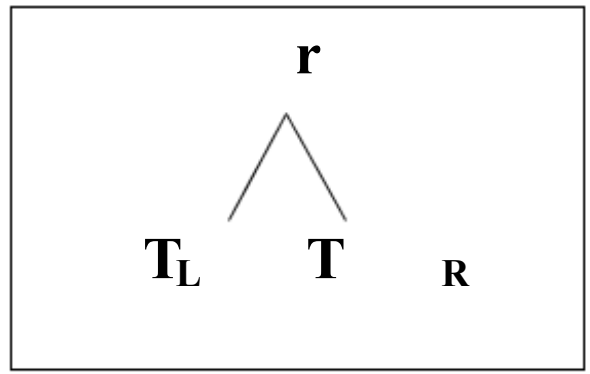
A traversal algorithm for a binary tree visits each node in the tree.

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

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

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

A binary tree is either empty or is of the 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 $\mathbf{T_L}$ and $\mathbf{T_R}$
  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
```

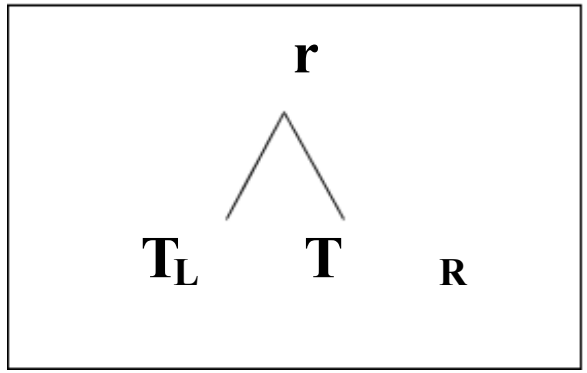
The above algorithm is missing the instruction to display the data in the root.

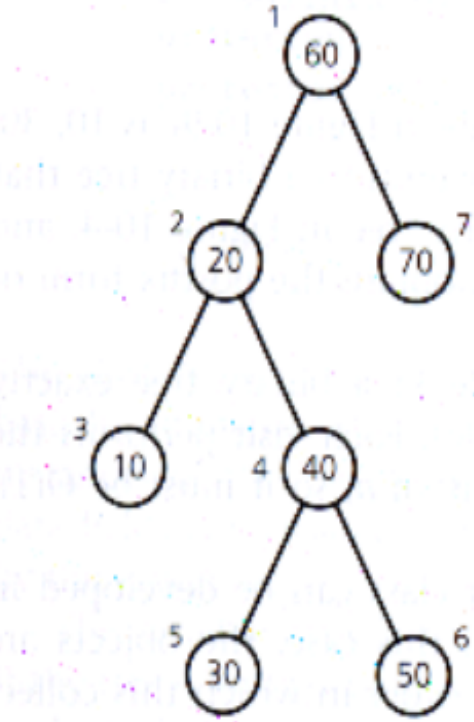
When **traversing** any binary tree, the algorithm has three choices of when to visit the root r.

- 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.

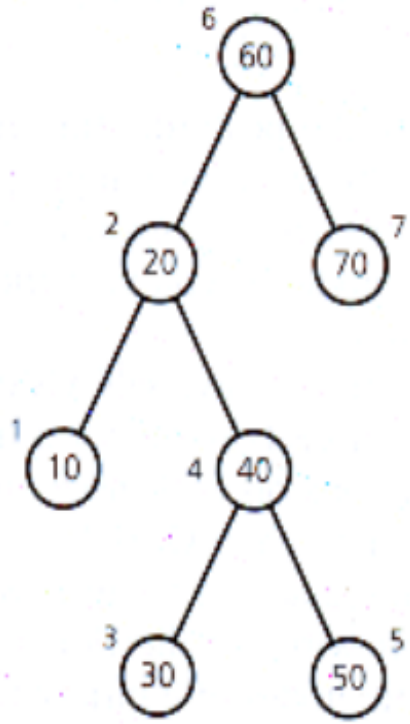
These traversals are called **preorder**, **inorder** and **postorder** respectively.

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

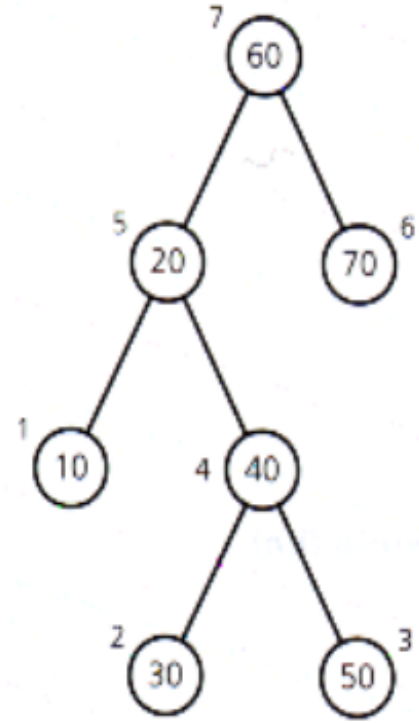




(a) Preorder: 60, 20, 10, 40, 30, 50, 70



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



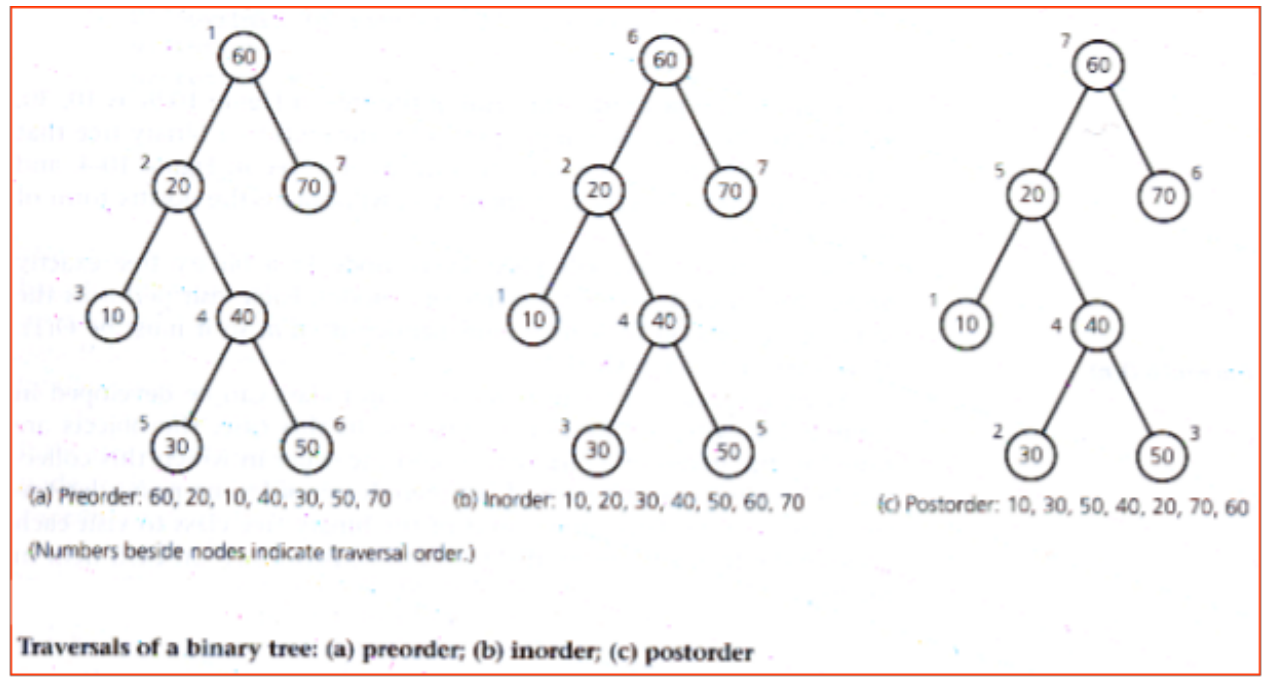
(c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

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

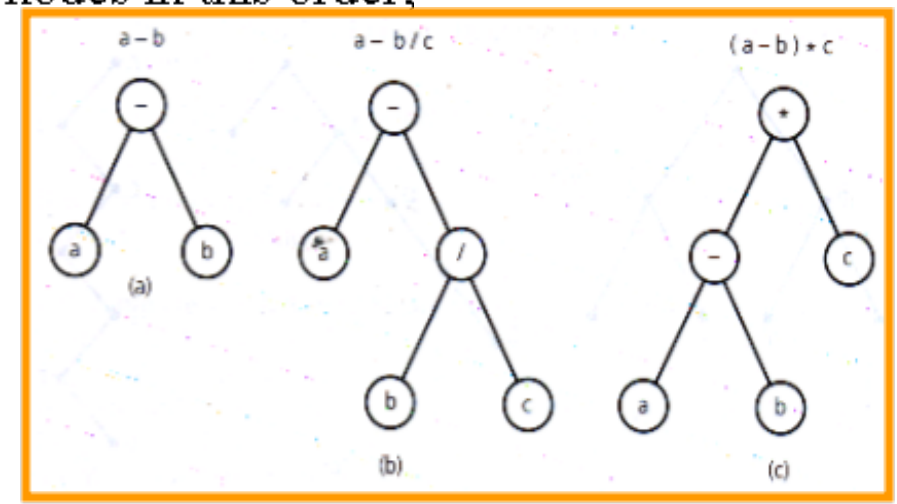
The **preorder traversal algorithm** is as follows:

```
Preorder(binTree)
// traverse the binary treebinTree 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 ofbinTree
      preorder(Left subtree of binTree's root)
      preorder(Right subtree ofbinTree's root)
} //end if
```



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

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



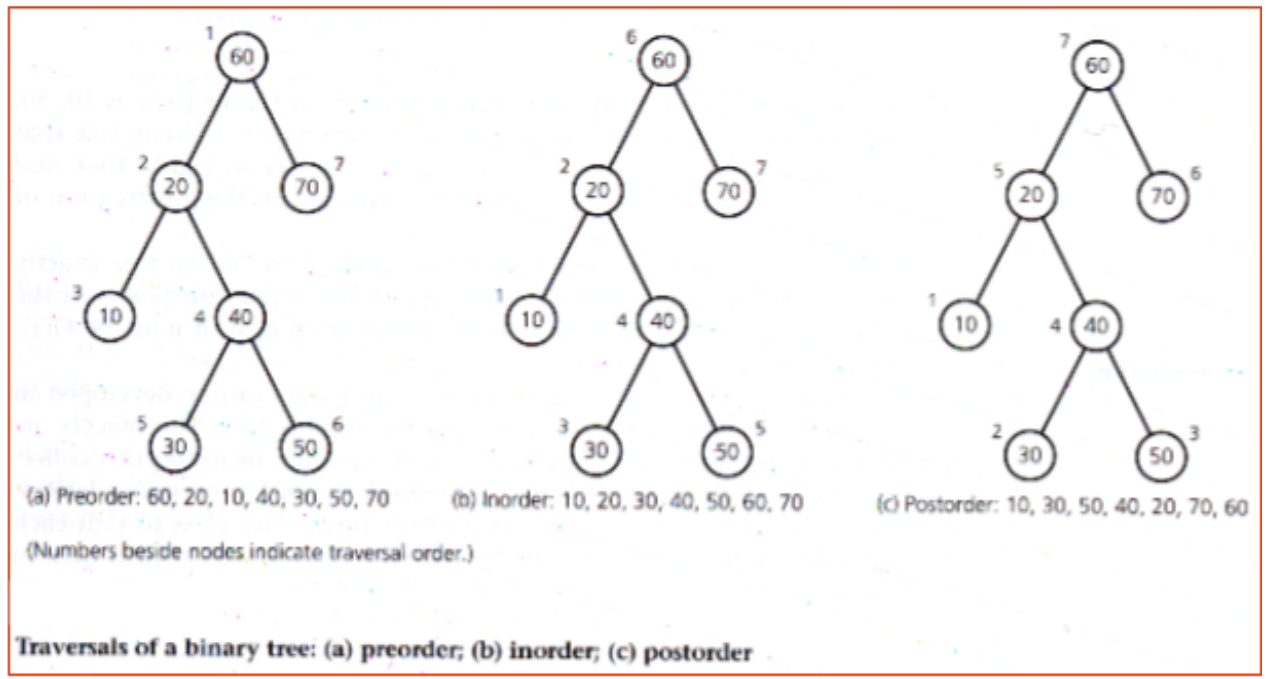
If we apply …

- **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)

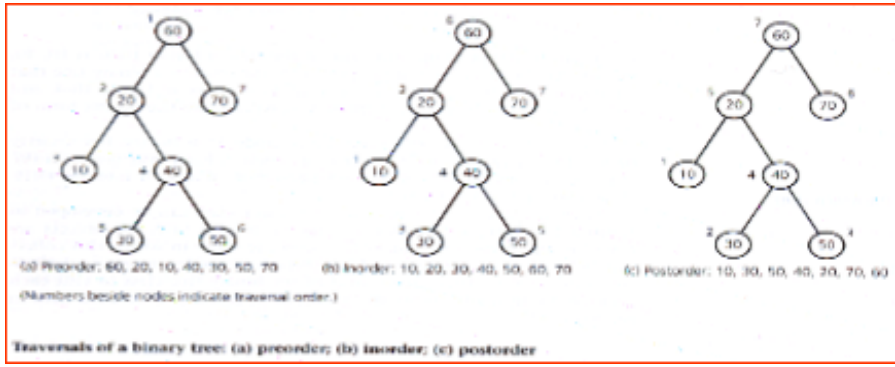
That is, the **prefix** expressions are:

(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
```



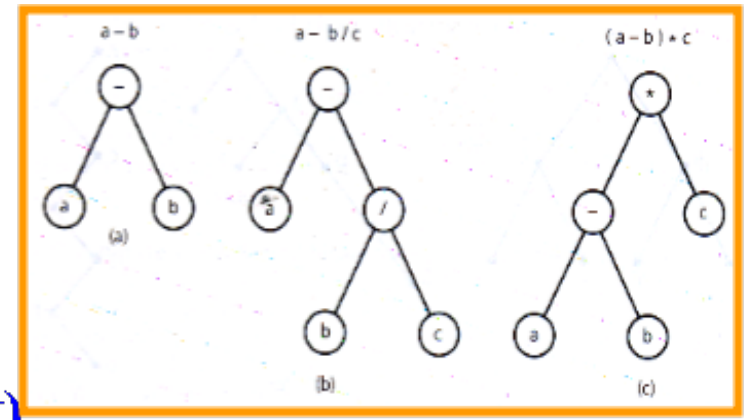
Traversals of a binary tree: (a) preorder; (b) inorder; (c) postorder

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

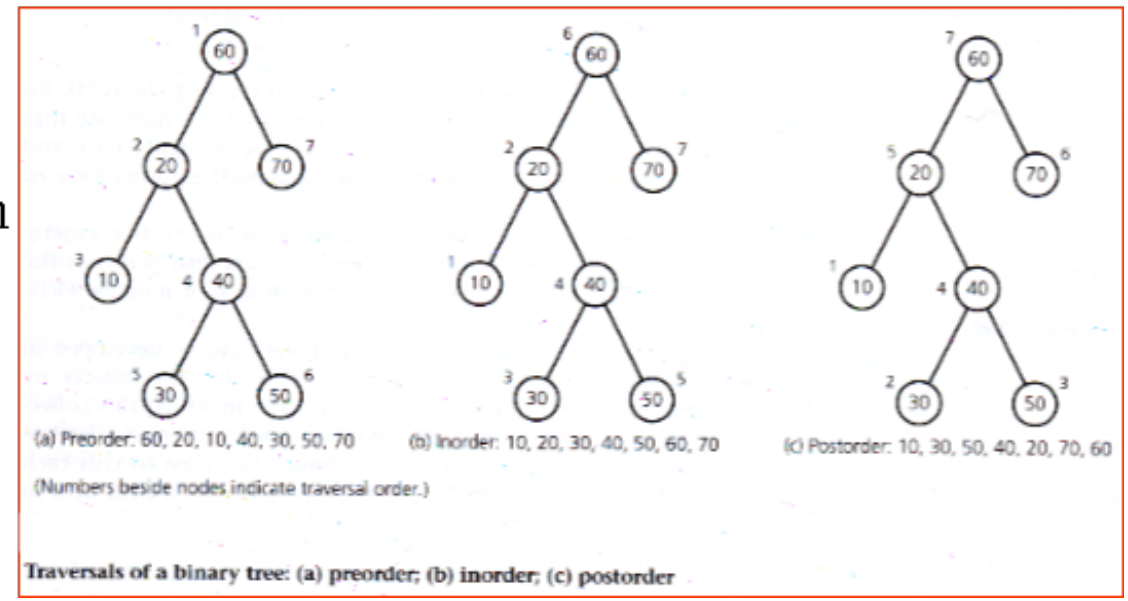
If you apply the **inorder** traversal to a binary search tree, you will visit the nodes in order according to their data values .

The postorder traversal algorithm

```
postorder(binTree)
// traverses the binary treebinTree in postOrder
// assumes that visit a node means
// display the nodes data item.

if (binTree is not empty)
{
      postorder(Left subtree ofbinTree's root)
      postorder(Right subtree ofbinTree's root)
      Display the data in the root ofbinTree
}  //end if
```



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



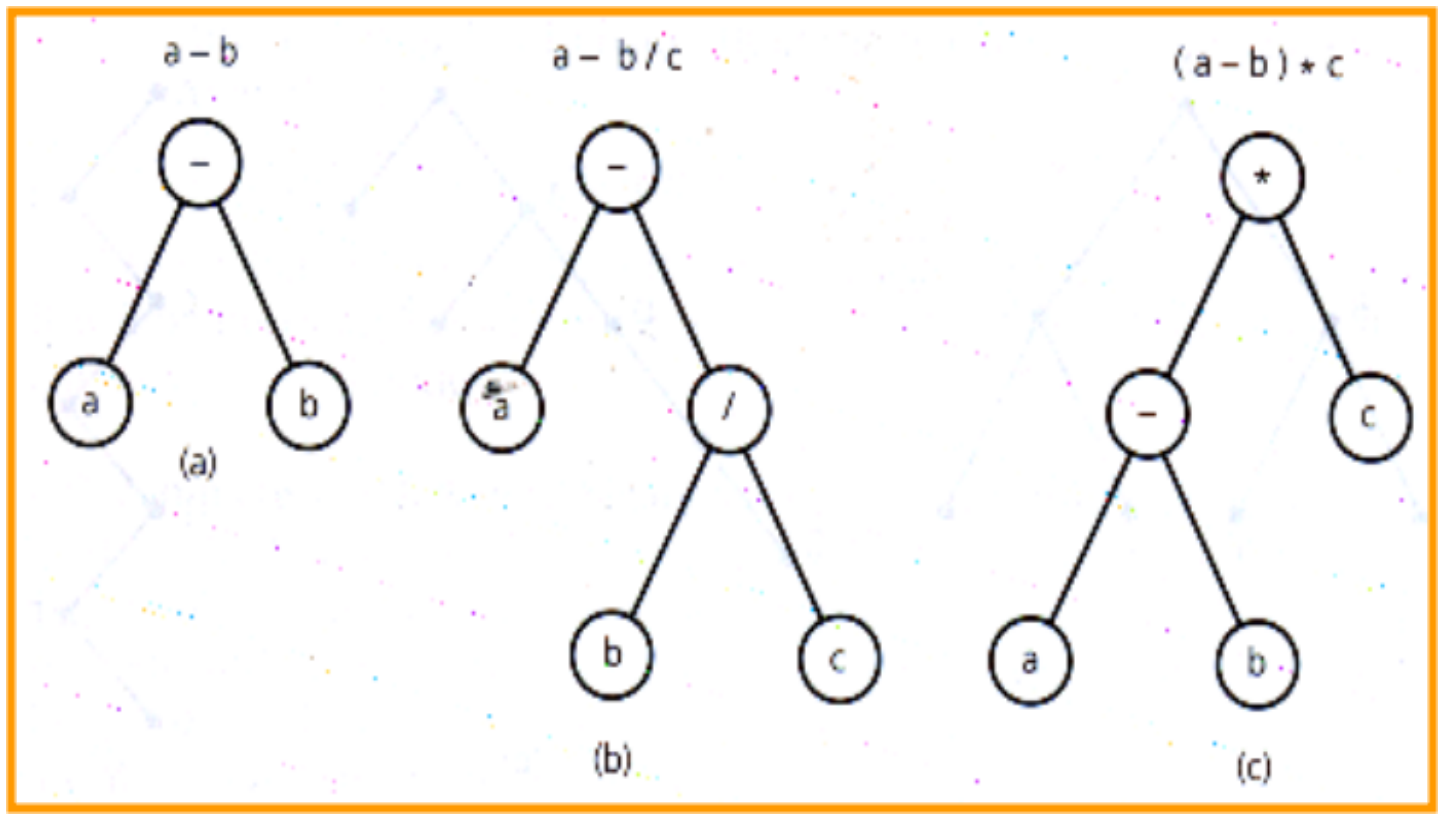
Traversals of a binary tree: (a) preorder; (b) inorder; (c) postorder

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:

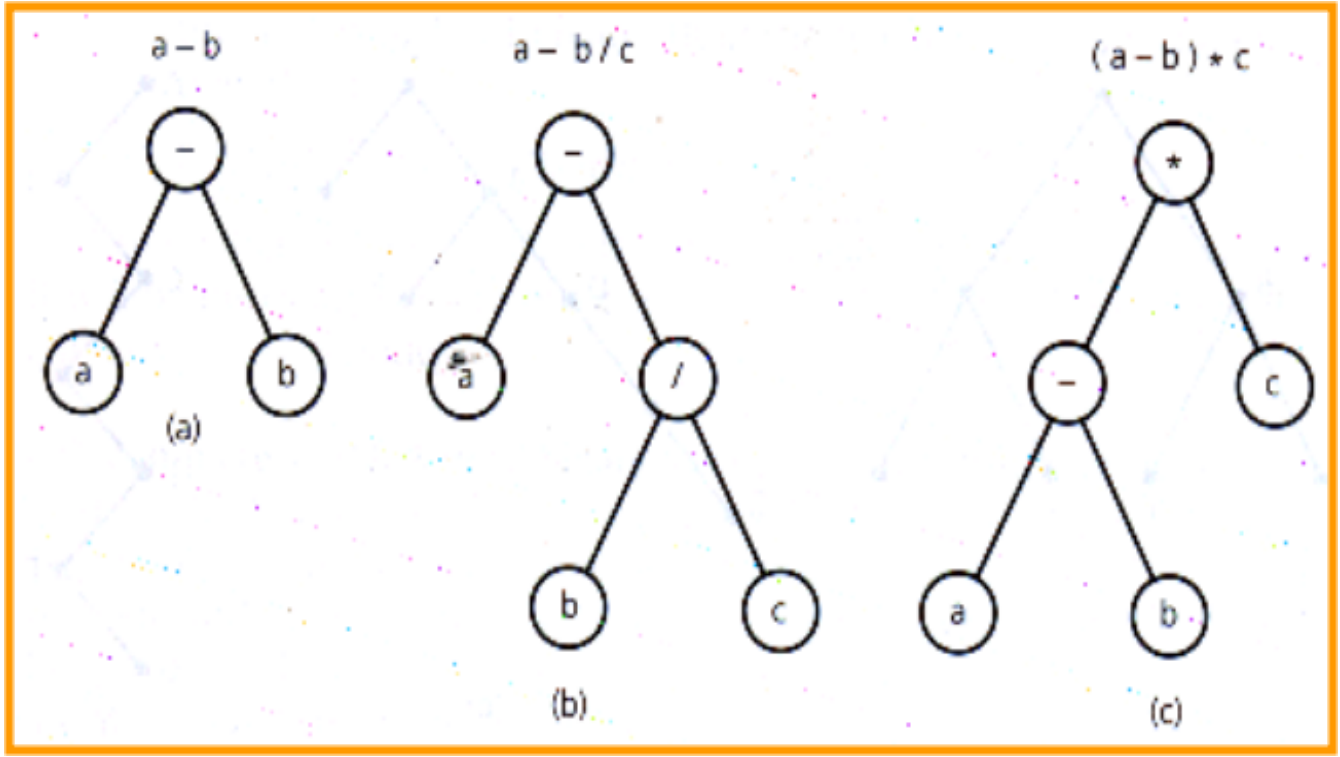
(a) ab-
(b) abc/-
(c) ab-c*



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

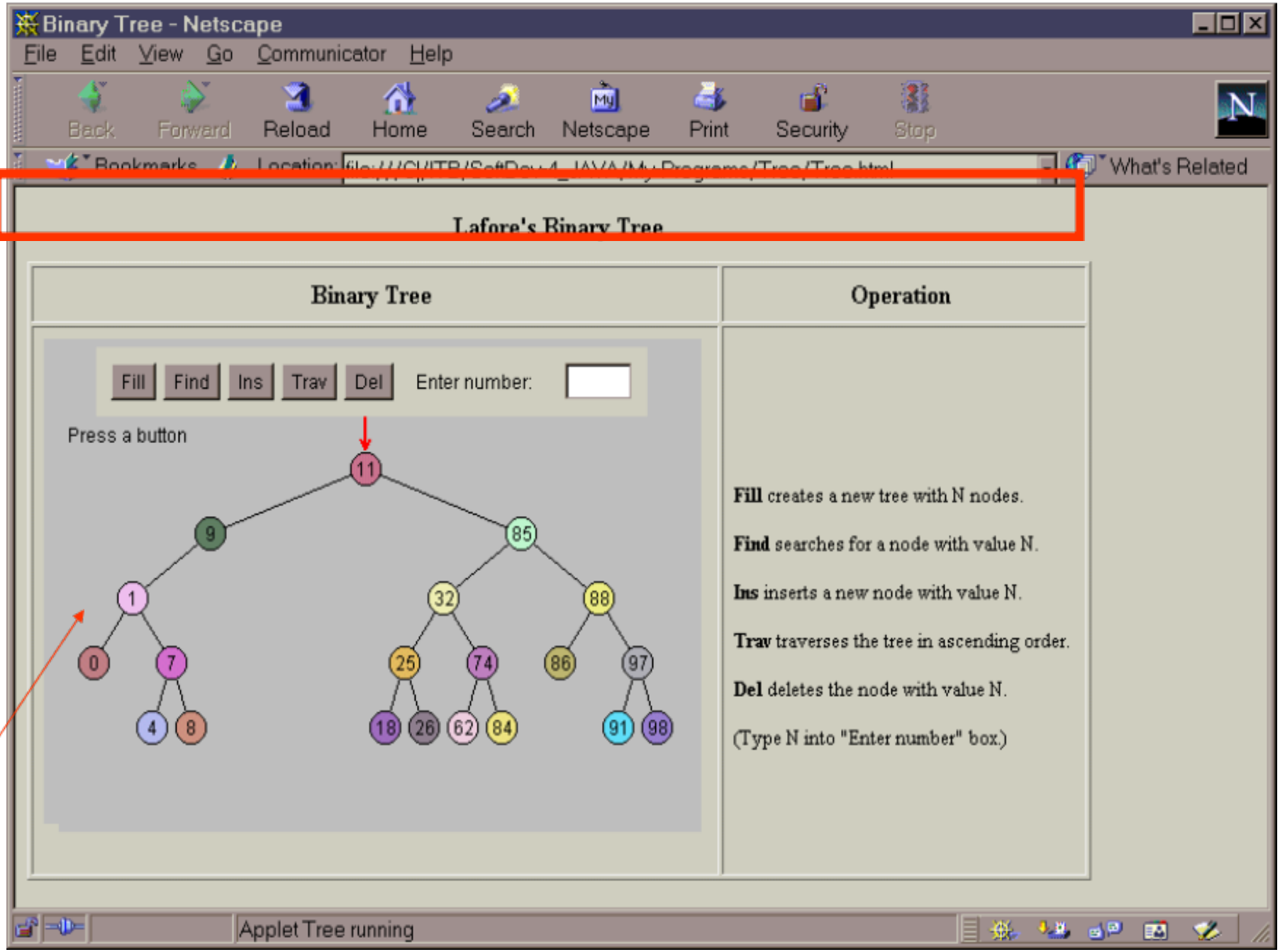
- Therefore, **n** visits occur for a tree of **n** nodes.
- Each visit performs the same operations of each node, independently of **n**.

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!



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**.

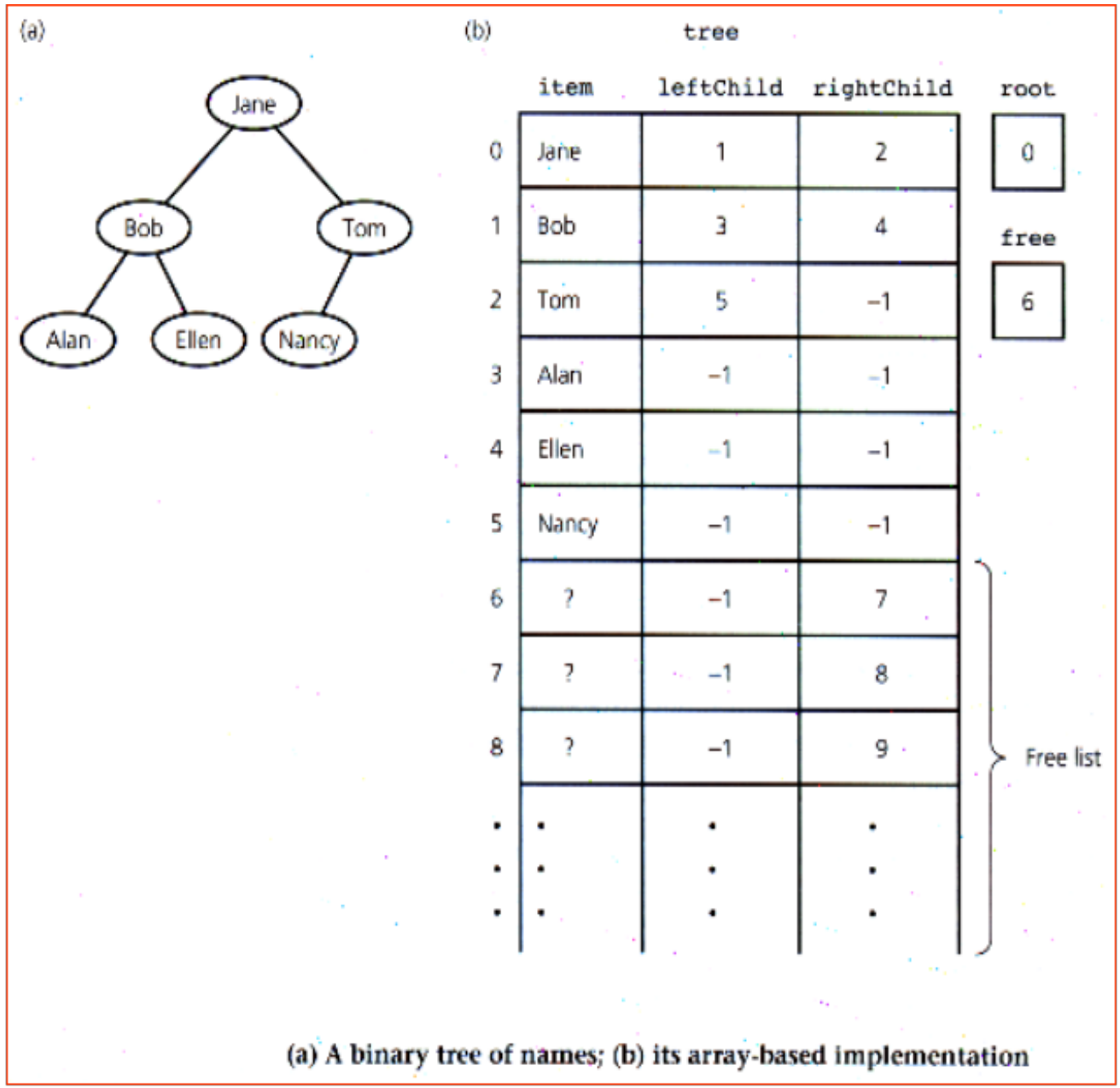
If the tree is **empty**, root is **–1**.

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) 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.

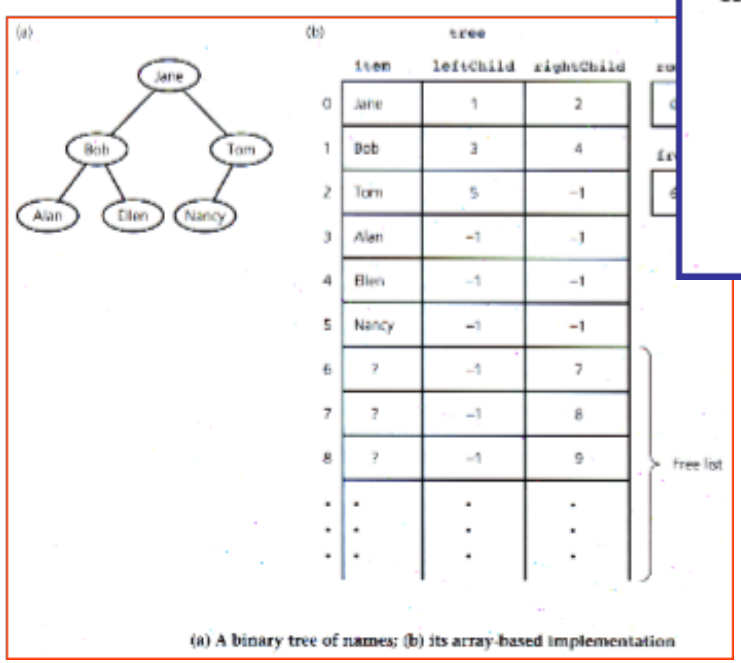
The variable **free** is the index to the first node in the free list and, ***arbitrarily*** the **rightChild** field of each node in the **free** list is the index of the next node in the free list

An Array-based representation of a complete tree.

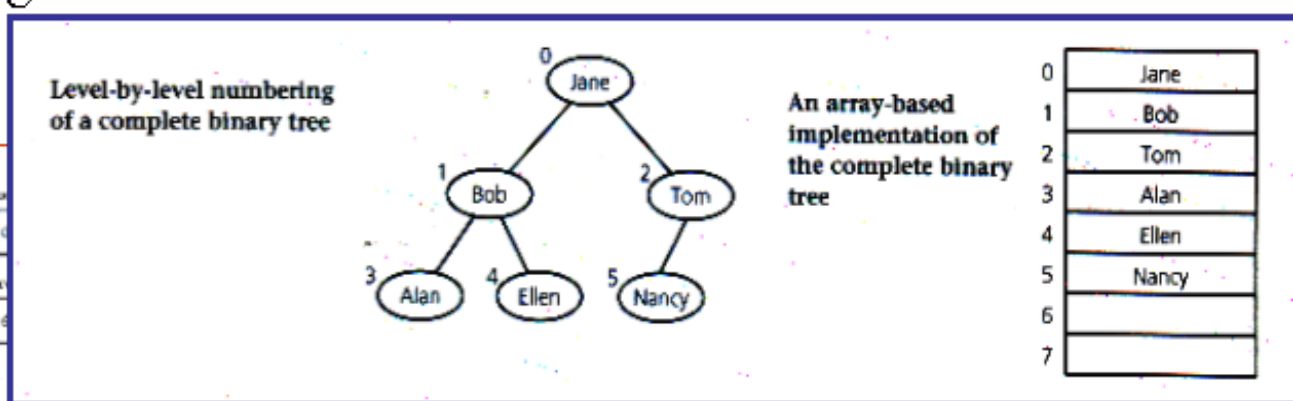
The previous implementation works with any binary tree, even though the tree in the previous diagram is complete.

- 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 **h** is full to level **h – 1** and has level **h** filled from left to right.



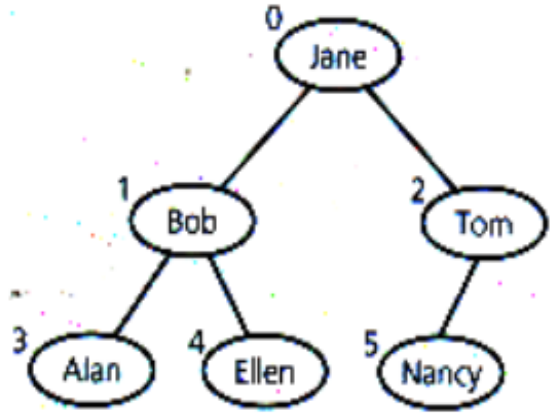
(a) A binary tree of names; (b) its array-based implementation



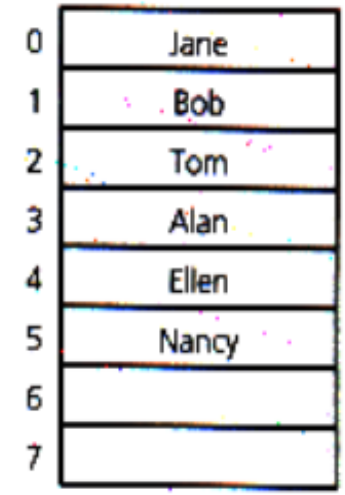
vs

•

Level-by-level numbering of a complete binary tree



An array-based implementation of the complete binary tree



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]**.

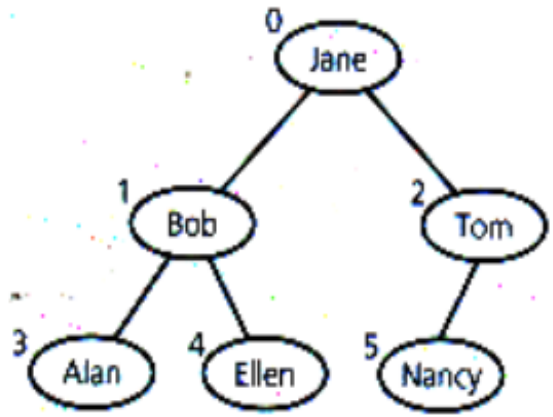
- We can match the tree to the array in the example in the diagrams

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

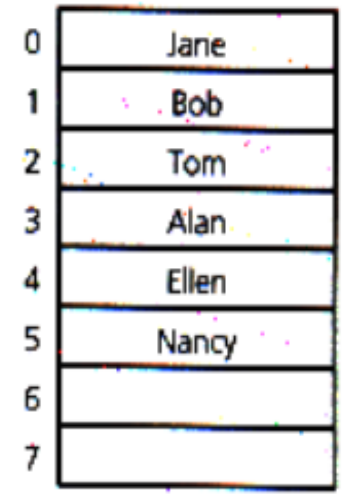
- Its left child (if its exists) is **tree[2*k+1]**
- Its right child (if its exists) is **tree[2*k+2]**
- Its parent (if tree[k] is not root) is **tree[(k – 1) / 2]**

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

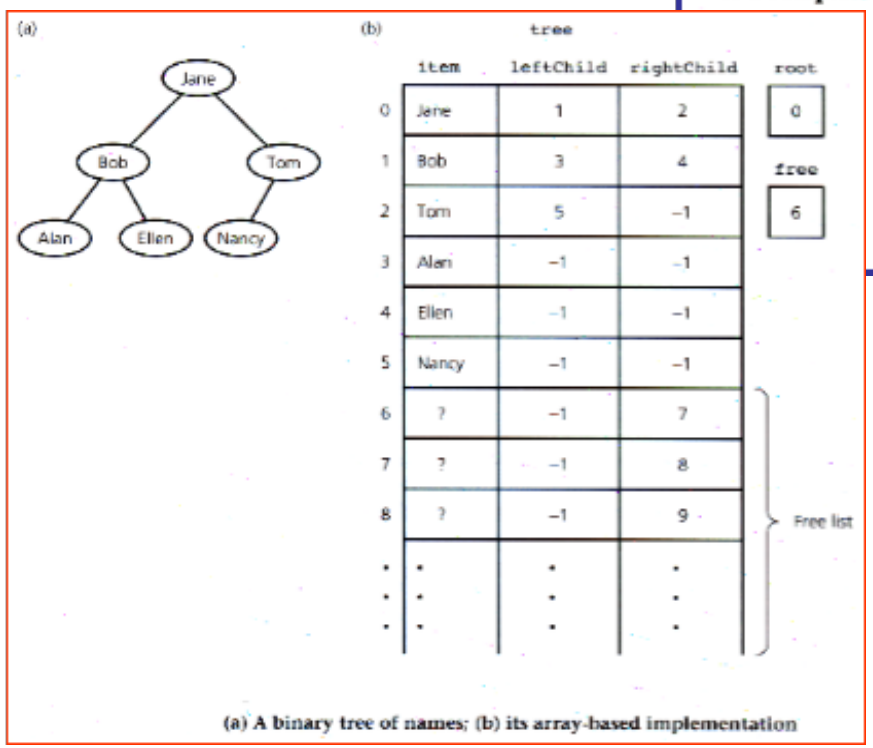
Level-by-level numbering of a complete binary tree



An array-based implementation of the complete binary tree

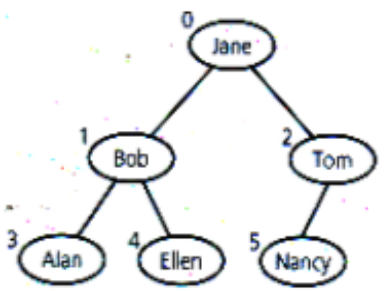


- The array based implementation requires a complete tree.
- If nodes were missing from the middle of the tree , the numbering scheme would be thrown off, and the parent-child relationship would be ambiguous.
- This requirement implies that any changes to the tree must maintain its completeness.

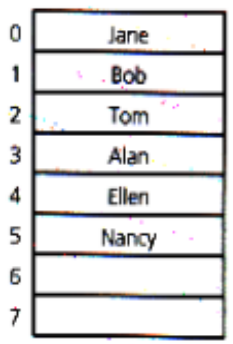


(a) A binary tree of names; (b) its array-based implementation

Level-by-level numbering of a complete binary tree

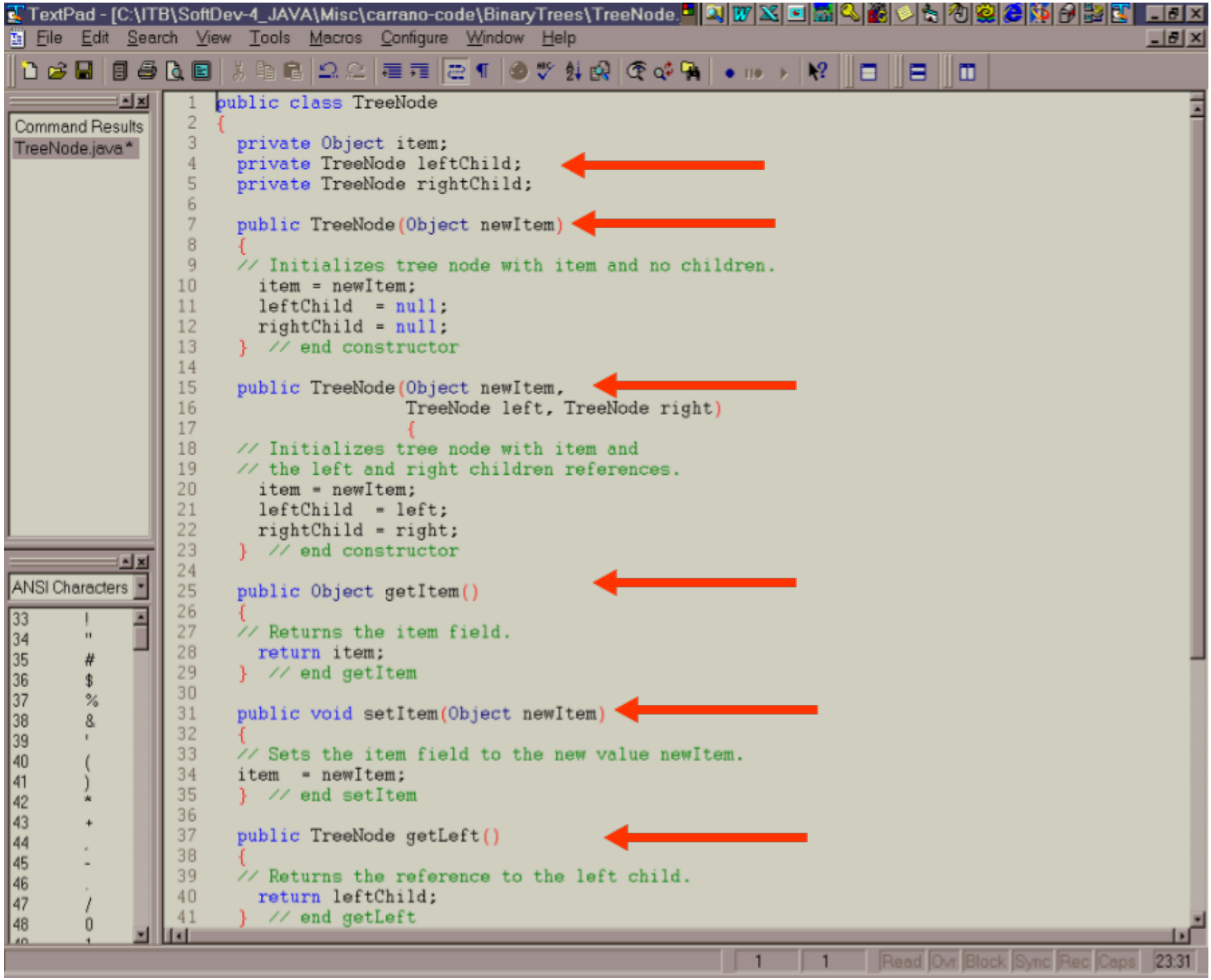


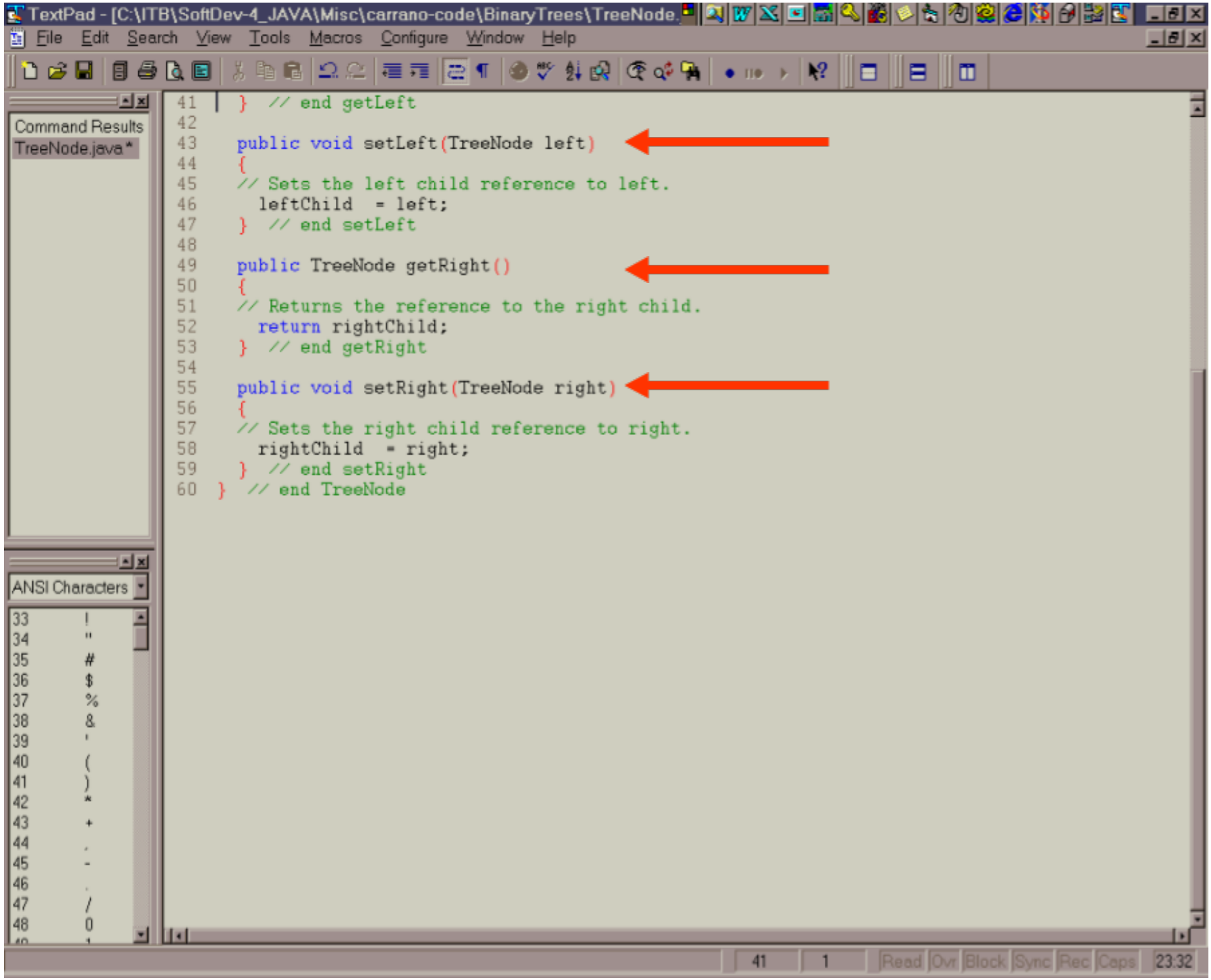
An array-based implementation of the complete binary tree

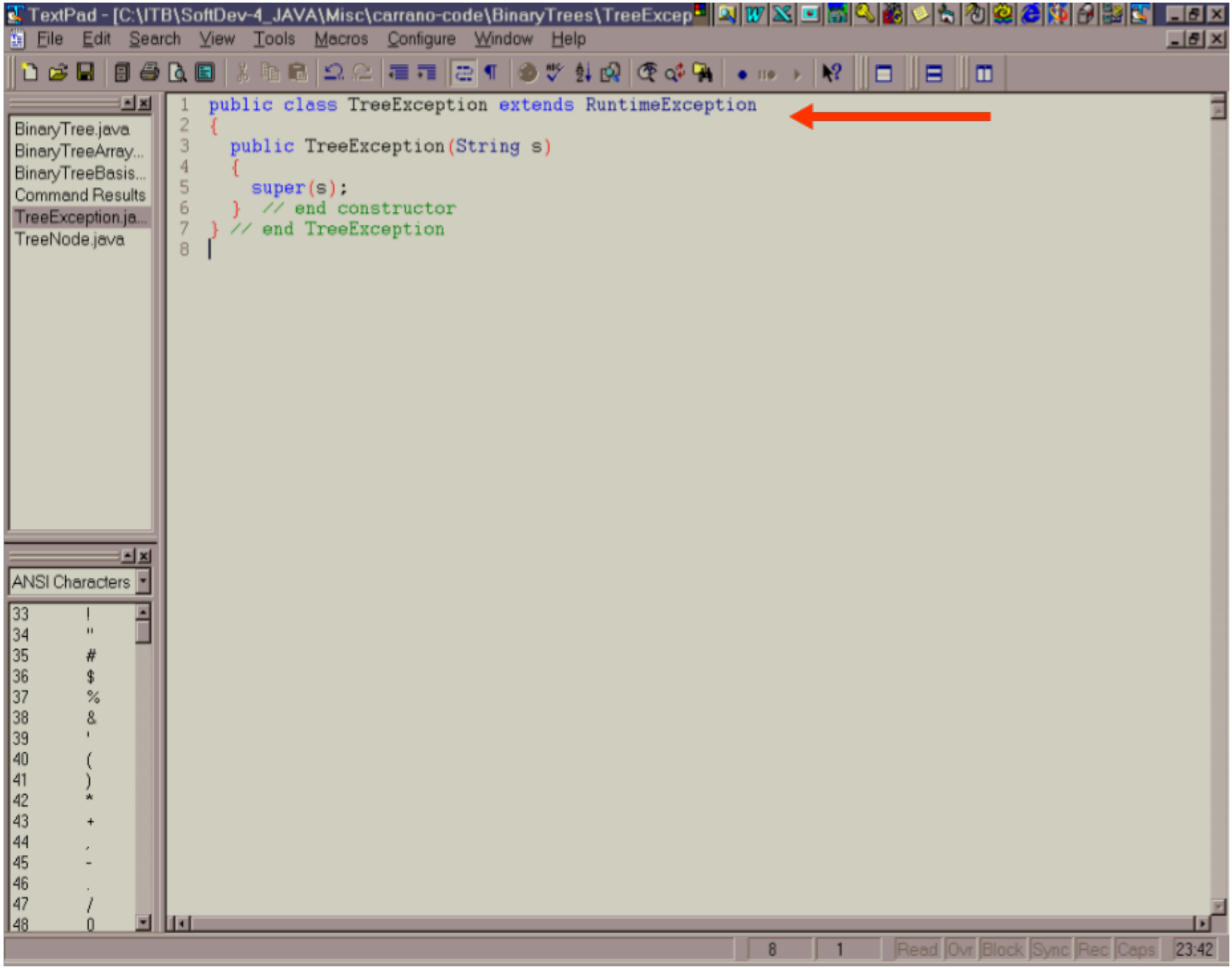


A Reference-Based Implementation of the ADT Binary Tree

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

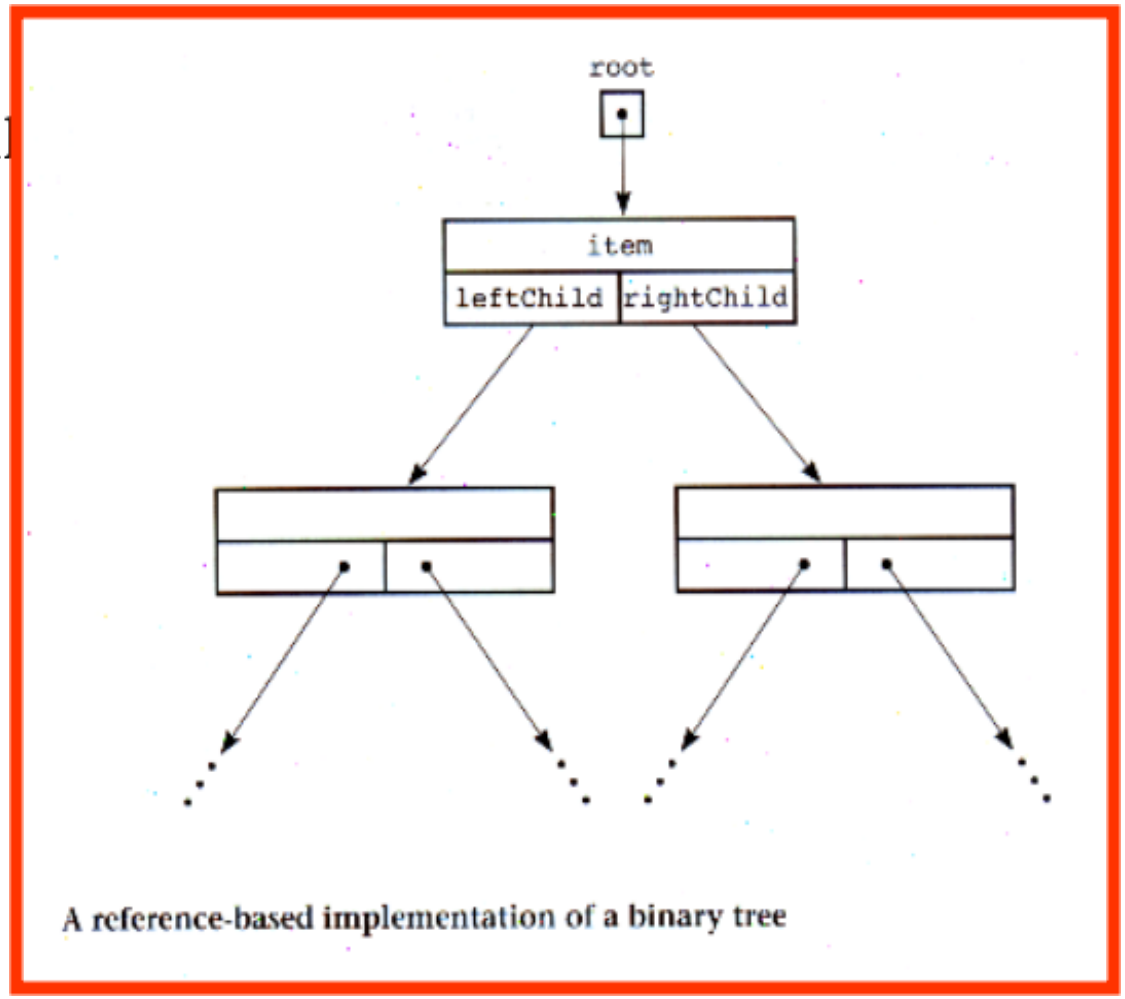

TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeNode.
File Edit Search View Tools Macros Configure Window Help
Command Results
TreeNode.java*
ANSI Characters
public class TreeNode
{
private Object item;
private TreeNode leftChild;
private TreeNode rightChild;
public TreeNode(Object newItem)
{
// Initializes tree node with item and no children.
item = newItem;
leftChild = null;
rightChild = null;
} // end constructor
public TreeNode(Object newItem,
TreeNode left, TreeNode right)
{
// Initializes tree node with item and
// the left and right children references.
item = newItem;
leftChild = left;
rightChild = right;
} // end constructor
public Object getItem()
{
// Returns the item field.
return item;
} // end getItem
public void setItem(Object newItem)
{
// Sets the item field to the new value newItem.
item = newItem;
} // end setItem
public TreeNode getLeft()
{
// Returns the reference to the left child.
return leftChild;
} // end getLeft
Read Ovr Block Sync Rec Caps
23:31


TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeNode.
File Edit Search View Tools Macros Configure Window Help
Command Results
TreeNode.java*
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
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
41
1
Read Ovr Block Sync Rec Caps
23:32


TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeExcep
File Edit Search View Tools Macros Configure Window Help
BinaryTree.java
BinaryTreeArray...
BinaryTreeBasis...
Command Results
TreeException.ja...
TreeNode.java
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
1 public class TreeException extends RuntimeException
2 {
3 public TreeException(String s)
4 {
5 super(s);
6 } // end constructor
7 } // end TreeException
8
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



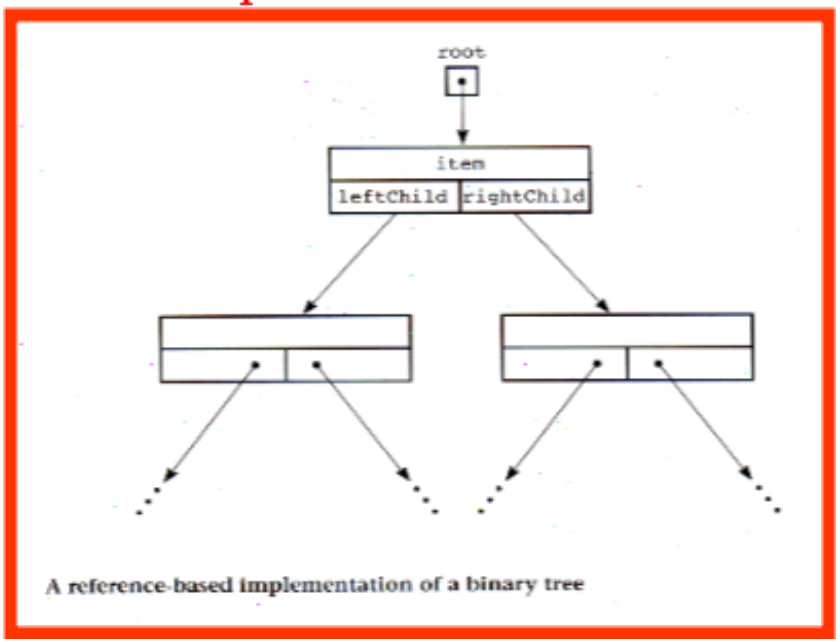
A reference-based implementation of a binary tree

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

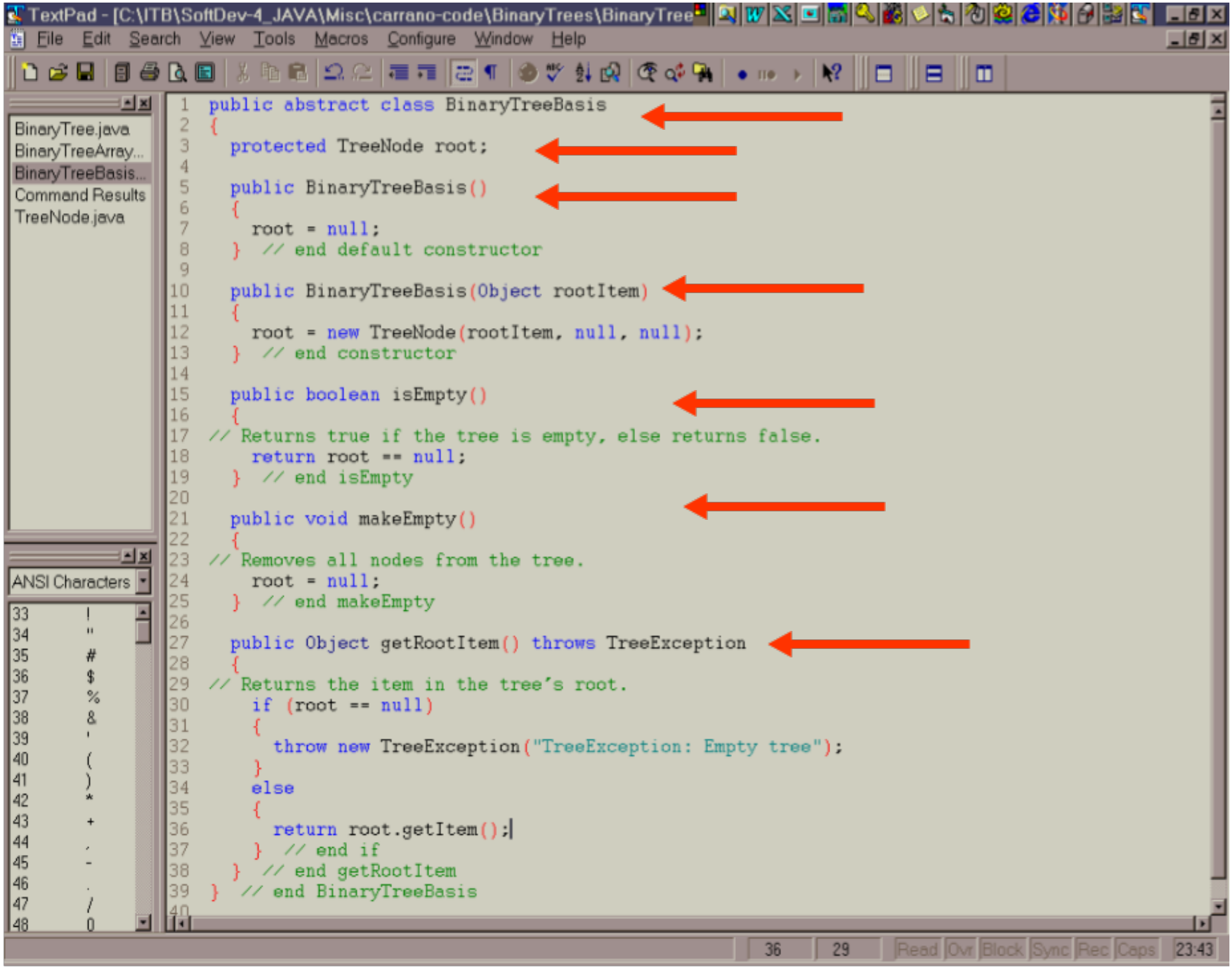
In a reference-based implementation, …

- **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**.



A reference-based implementation of a binary tree

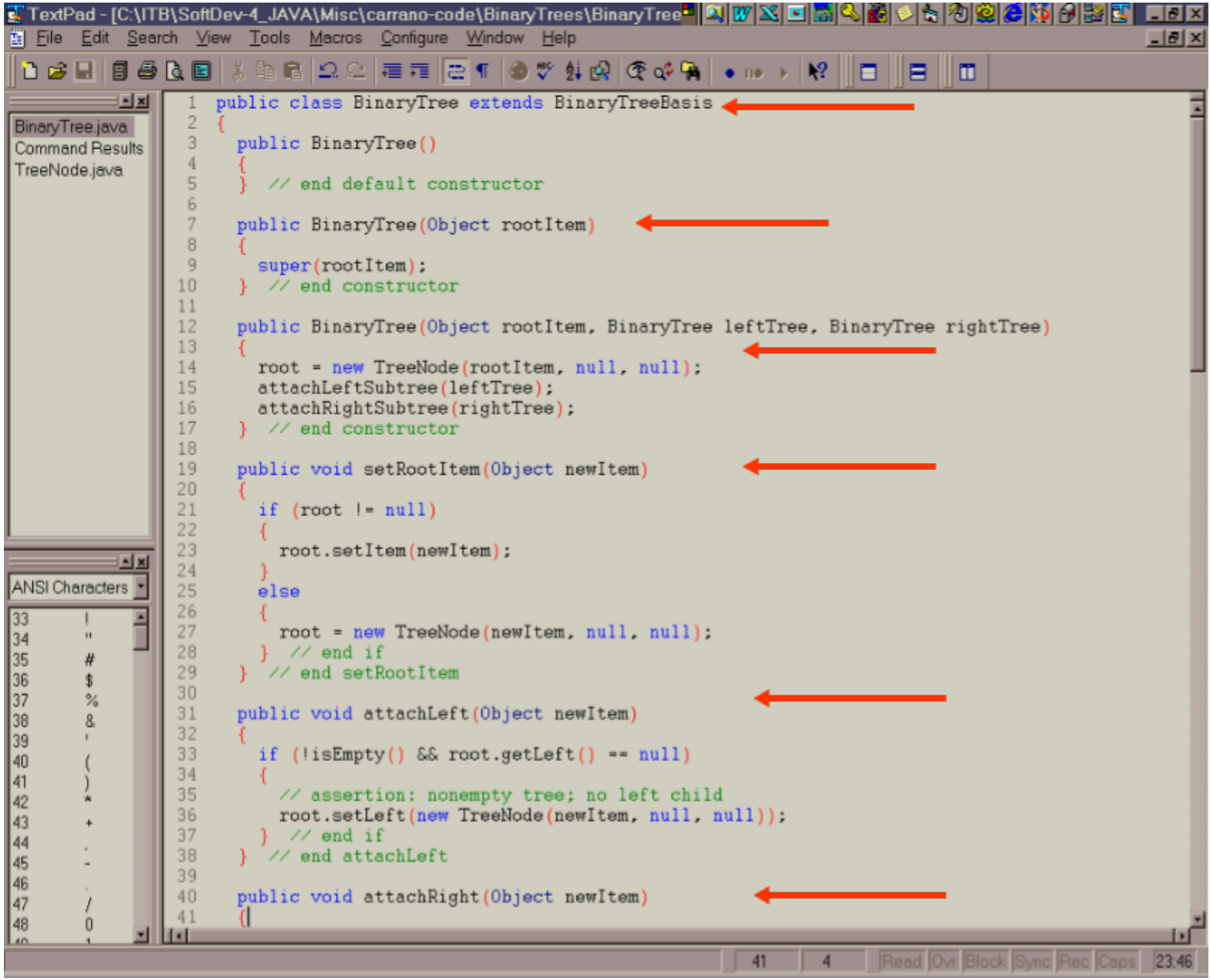

TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\BinaryTree
File Edit Search View Tools Macros Configure Window Help
BinaryTree.java
BinaryTreeArray...
BinaryTreeBasis...
Command Results
TreeNode.java
ANSI Characters
1 public abstract class BinaryTreeBasis
2 {
3 protected TreeNode root;
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
40
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
36
29
Read Ovr Block Sync Rec Caps
23:43

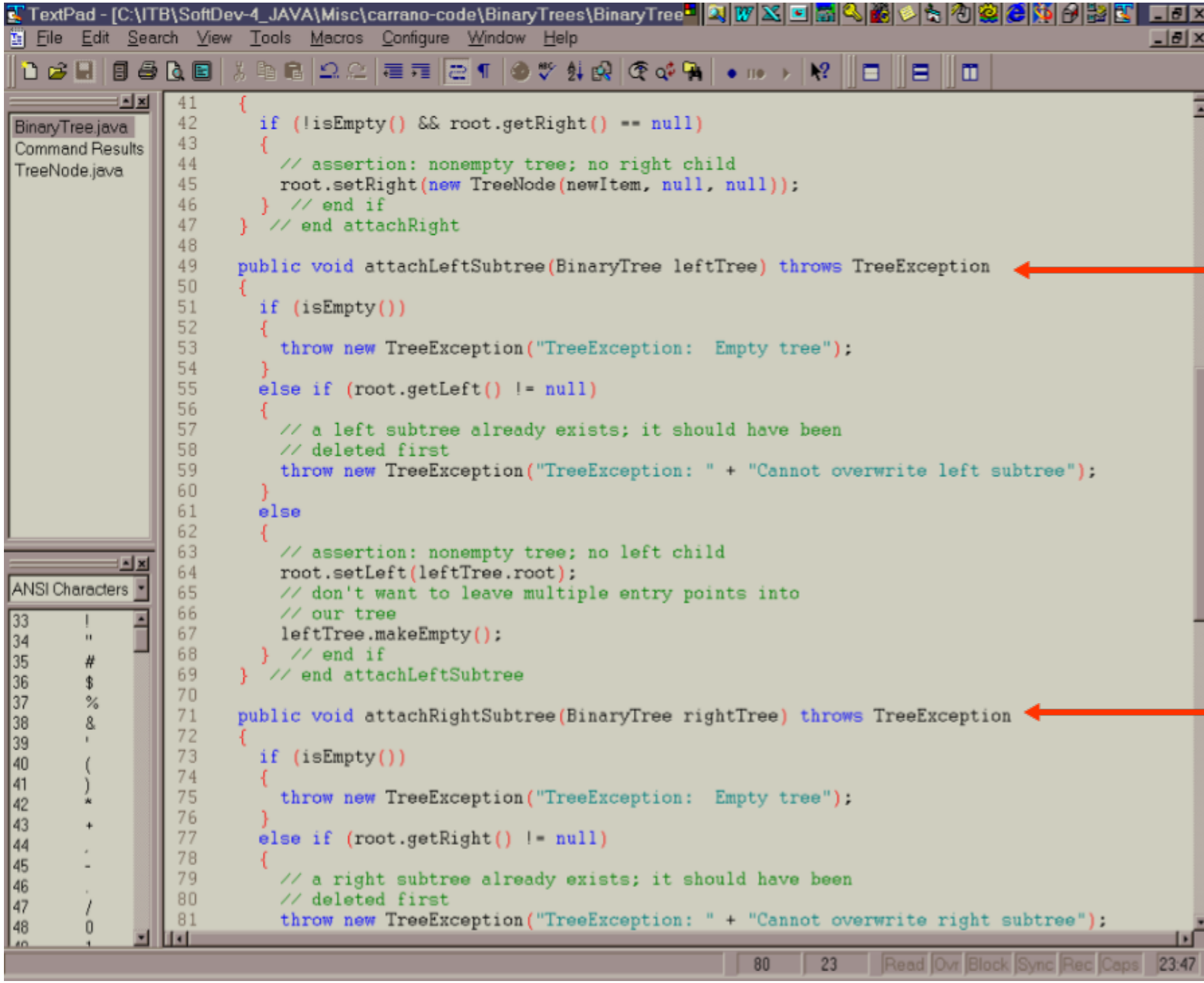
BinaryTreeBasis will be used as the base for the implementation of particular binary trees.

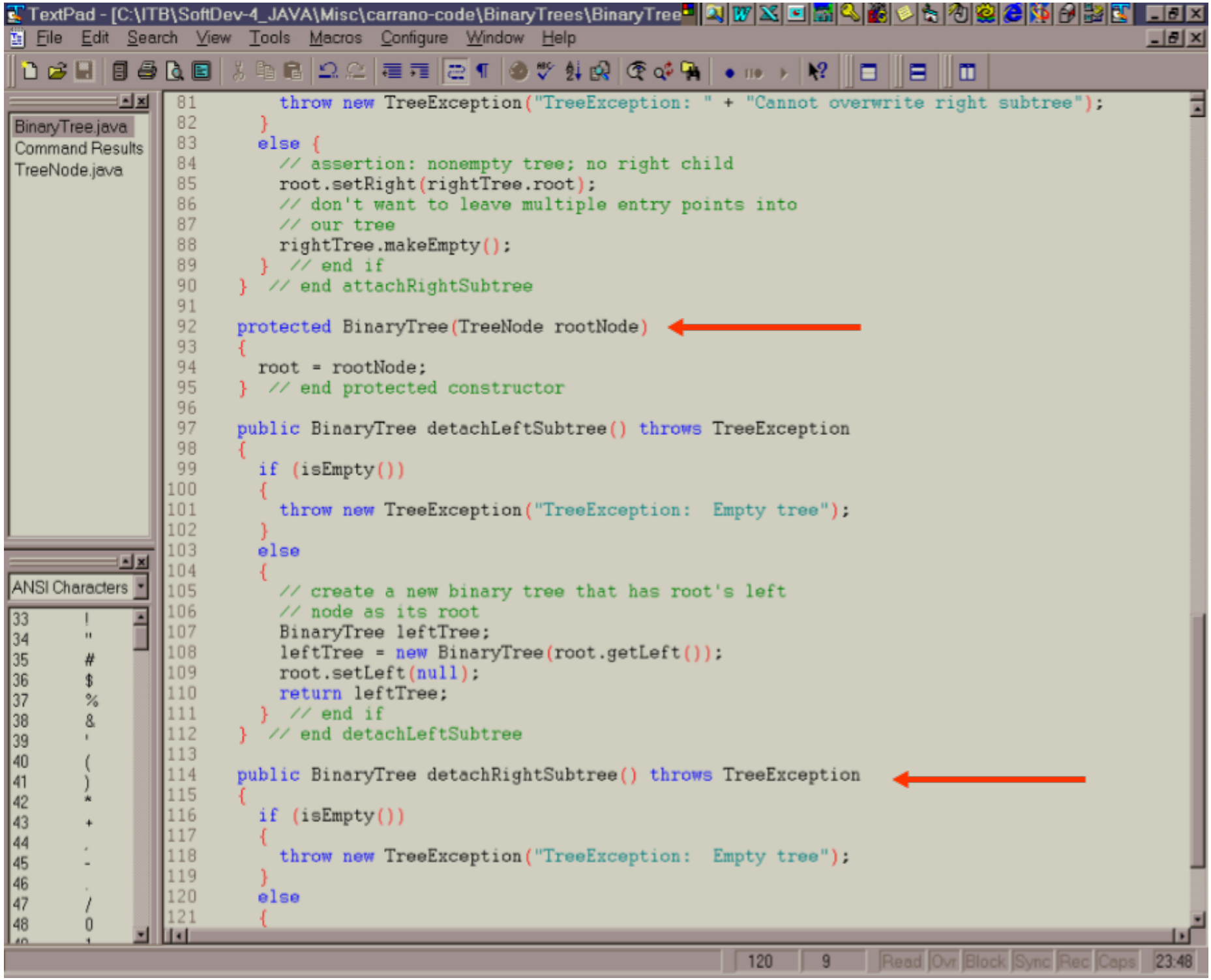
- It is declared as an **abstract** class because it is used for inheritance purposes only. There can be no direct instances of this class.
- **BinaryTreeBasis** declares the root of the tree as a protected item so that the subclasses will have direct access to the root of the tree.
- It also provides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

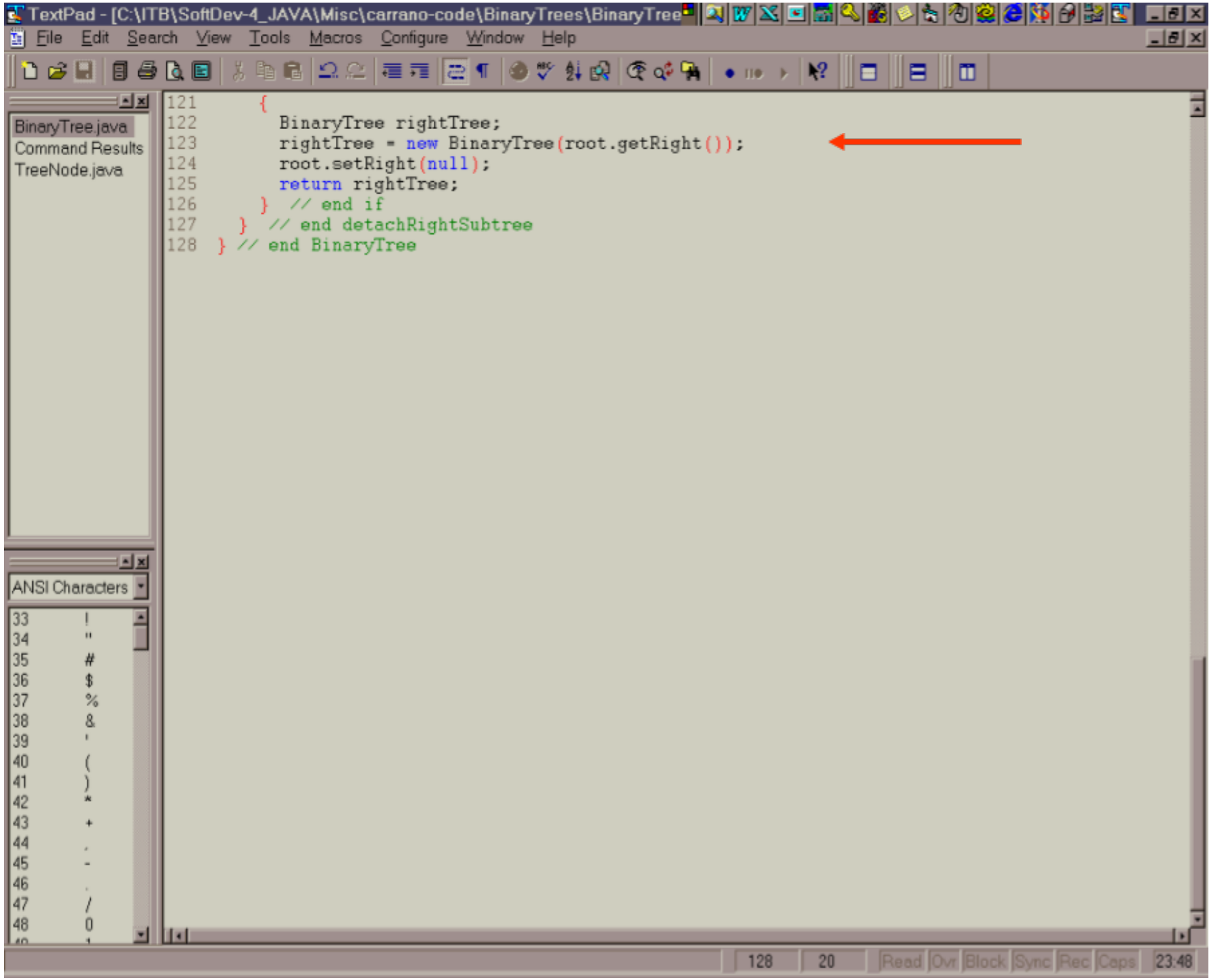
Setting the contents of the root node is left to a subclass, since different kinds of binary trees may have varying requirements regarding the value in the **root** node of the tree.

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
ANSI Characters
public class BinaryTree extends BinaryTreeBasis
{
public BinaryTree()
{
} // end default constructor
public BinaryTree(Object rootItem)
{
super(rootItem);
} // end constructor
public BinaryTree(Object rootItem, BinaryTree leftTree, BinaryTree rightTree)
{
root = new TreeNode(rootItem, null, null);
attachLeftSubtree(leftTree);
attachRightSubtree(rightTree);
} // end constructor
public void setRootItem(Object newItem)
{
if (root != null)
{
root.setItem(newItem);
}
else
{
root = new TreeNode(newItem, null, null);
} // end if
} // end setRootItem
public void attachLeft(Object newItem)
{
if (!isEmpty() && root.getLeft() == null)
{
// assertion: nonempty tree; no left child
root.setLeft(new TreeNode(newItem, null, null));
} // end if
} // end attachLeft
public void attachRight(Object newItem)
{
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
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
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");
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
ANSI Characters
throw new TreeException("TreeException: " + "Cannot overwrite right subtree");
}
else {
// assertion: nonempty tree; no right child
root.setRight(rightTree.root);
// don't want to leave multiple entry points into
// our tree
rightTree.makeEmpty();
} // end if
} // end attachRightSubtree
protected BinaryTree(TreeNode rootNode)
{
root = rootNode;
} // end protected constructor
public BinaryTree detachLeftSubtree() throws TreeException
{
if (isEmpty())
{
throw new TreeException("TreeException: Empty tree");
}
else
{
// create a new binary tree that has root's left
// node as its root
BinaryTree leftTree;
leftTree = new BinaryTree(root.getLeft());
root.setLeft(null);
return leftTree;
} // end if
} // end detachLeftSubtree
public BinaryTree detachRightSubtree() throws TreeException
{
if (isEmpty())
{
throw new TreeException("TreeException: Empty tree");
}
else
{
120 9 Read Ovr Block Sync Rec Caps 23:48


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
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
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
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 **root** 2 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:

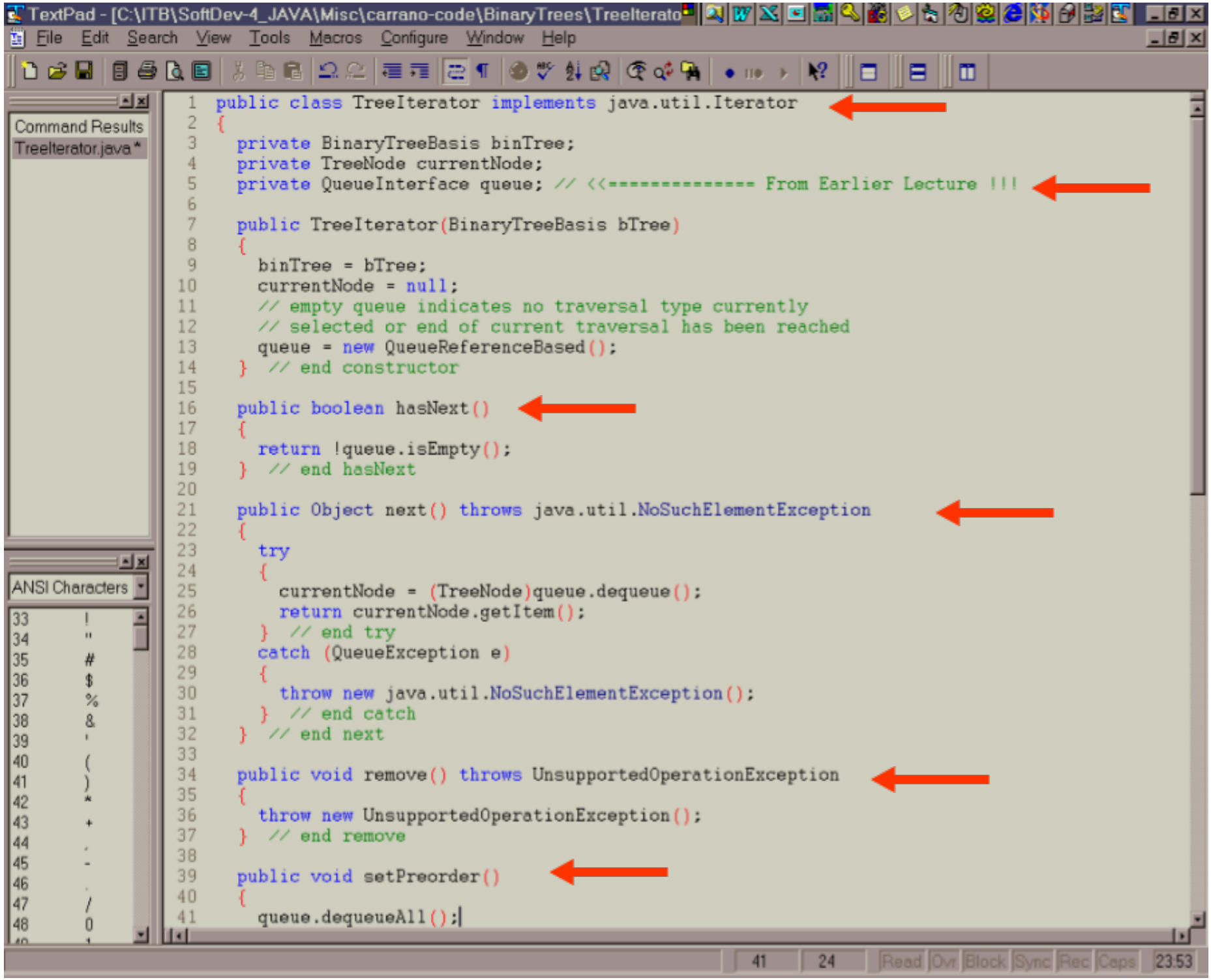
- **next()**
- **hasNext()**
- **remove()**

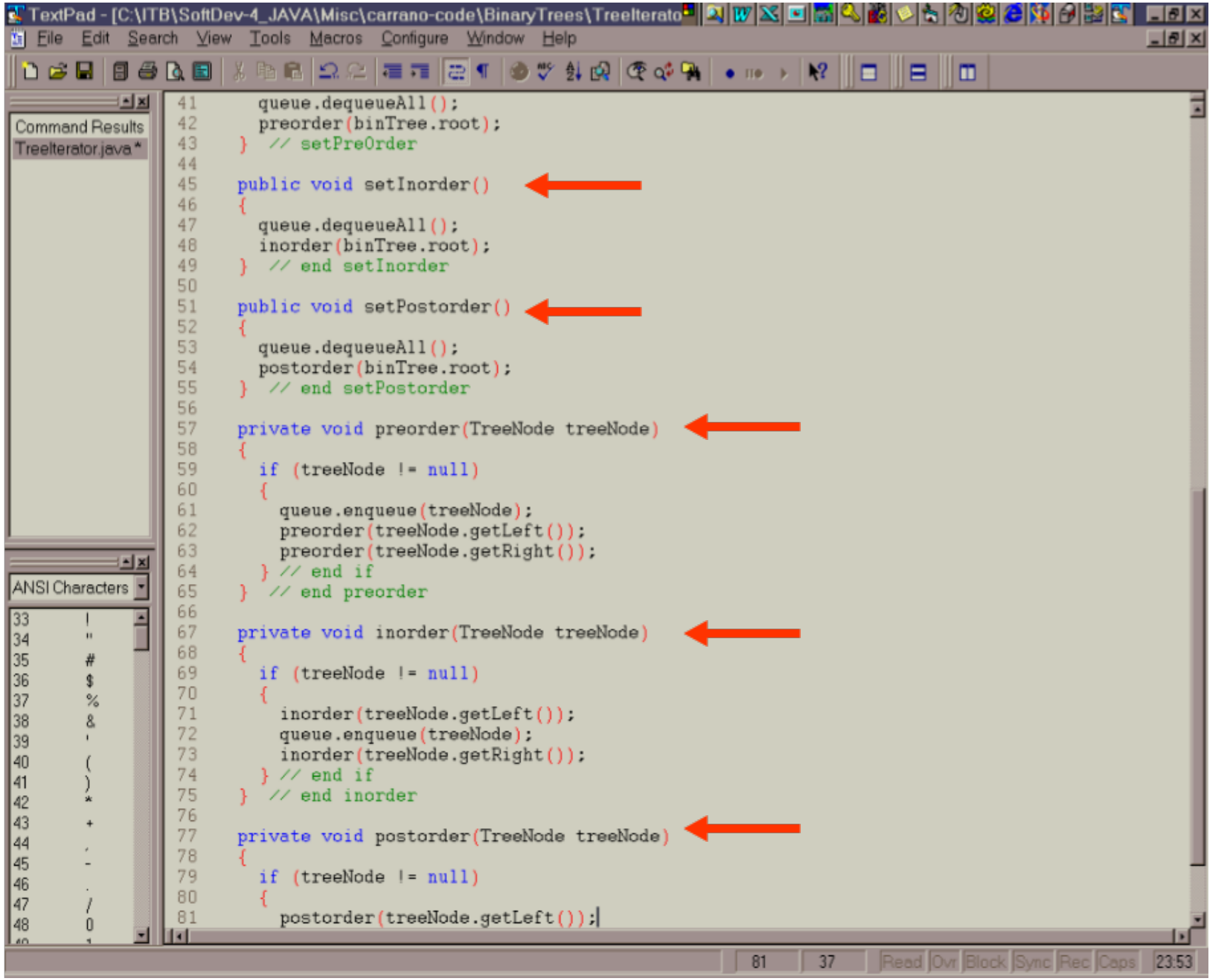
We will not implement the **remove**() method in this version of the **iterator** because:

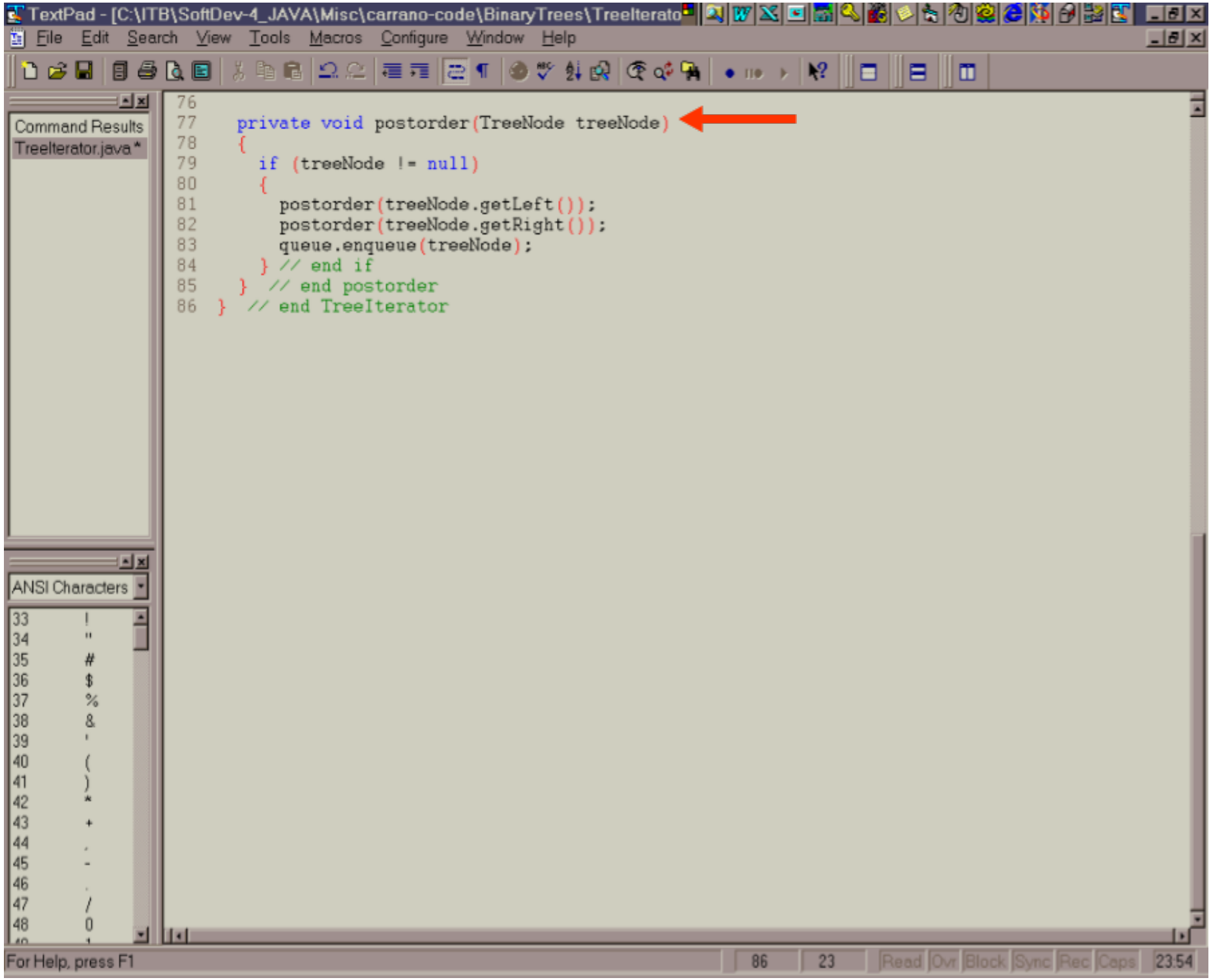
- The semantics of removing a node from a tree may depend on the type of tree we are working with(i.e. a binary search tree)
- The class BinaryTree itself does not provide a method for removing nodes from the tree.

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 itera **t**or will not provide any items from the tree (**hasNext** returns false)


TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeIterato
File Edit Search View Tools Macros Configure Window Help
Command Results
TreeIterator.java *
ANSI Characters
public class TreeIterator implements java.util.Iterator
{
private BinaryTreeBasis binTree;
private TreeNode currentNode;
private QueueInterface queue; // <<============== From Earlier Lecture !!!
public TreeIterator(BinaryTreeBasis bTree)
{
binTree = bTree;
currentNode = null;
// empty queue indicates no traversal type currently
// selected or end of current traversal has been reached
queue = new QueueReferenceBased();
} // end constructor
public boolean hasNext()
{
return !queue.isEmpty();
} // end hasNext
public Object next() throws java.util.NoSuchElementException
{
try
{
currentNode = (TreeNode)queue.dequeue();
return currentNode.getItem();
} // end try
catch (QueueException e)
{
throw new java.util.NoSuchElementException();
} // end catch
} // end next
public void remove() throws UnsupportedOperationException
{
throw new UnsupportedOperationException();
} // end remove
public void setPreorder()
{
queue.dequeueAll();
41 24 Read Ovr Block Sync Rec Caps 23:53


TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeIterato
File Edit Search View Tools Macros Configure Window Help
Command Results
TreeIterator.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());
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
81 37
Read Ovr Block Sync Rec Caps
23:53


TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeIterato
File Edit Search View Tools Macros Configure Window Help
Command Results
TreeIterator.java *
ANSI Characters
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
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.

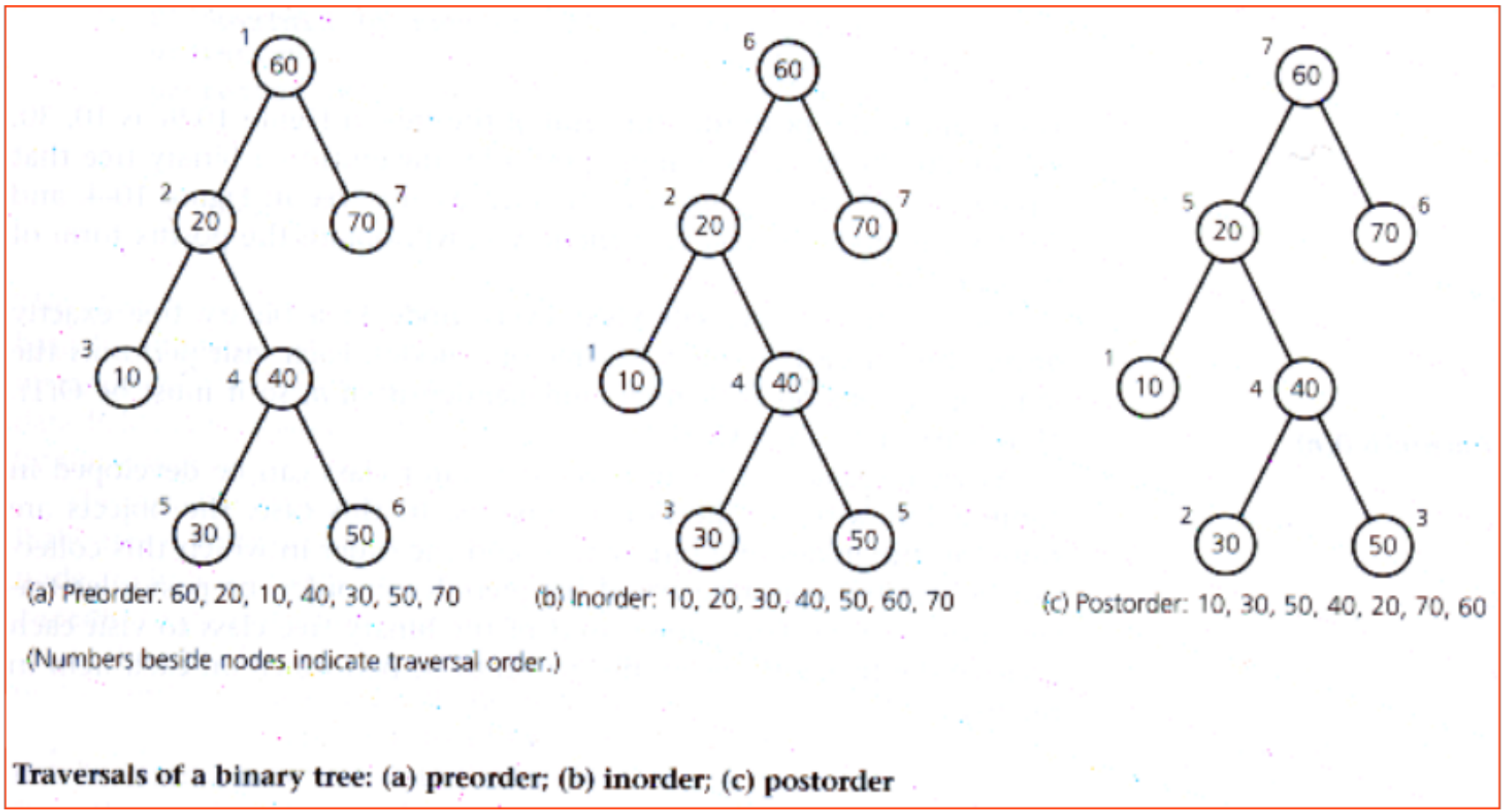
The following statements create an iterator that will perform a preorder traversal of a tree **tree4**:

```
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.j

File Edit Search View Tools Macros Configure Window Help

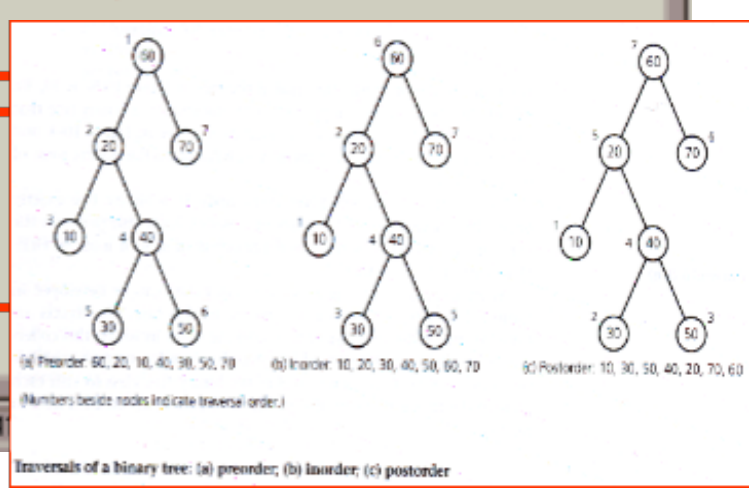
Command Results
QueueTest.java
TreeIterator.java *
TreeTest.java

ANSI Characters

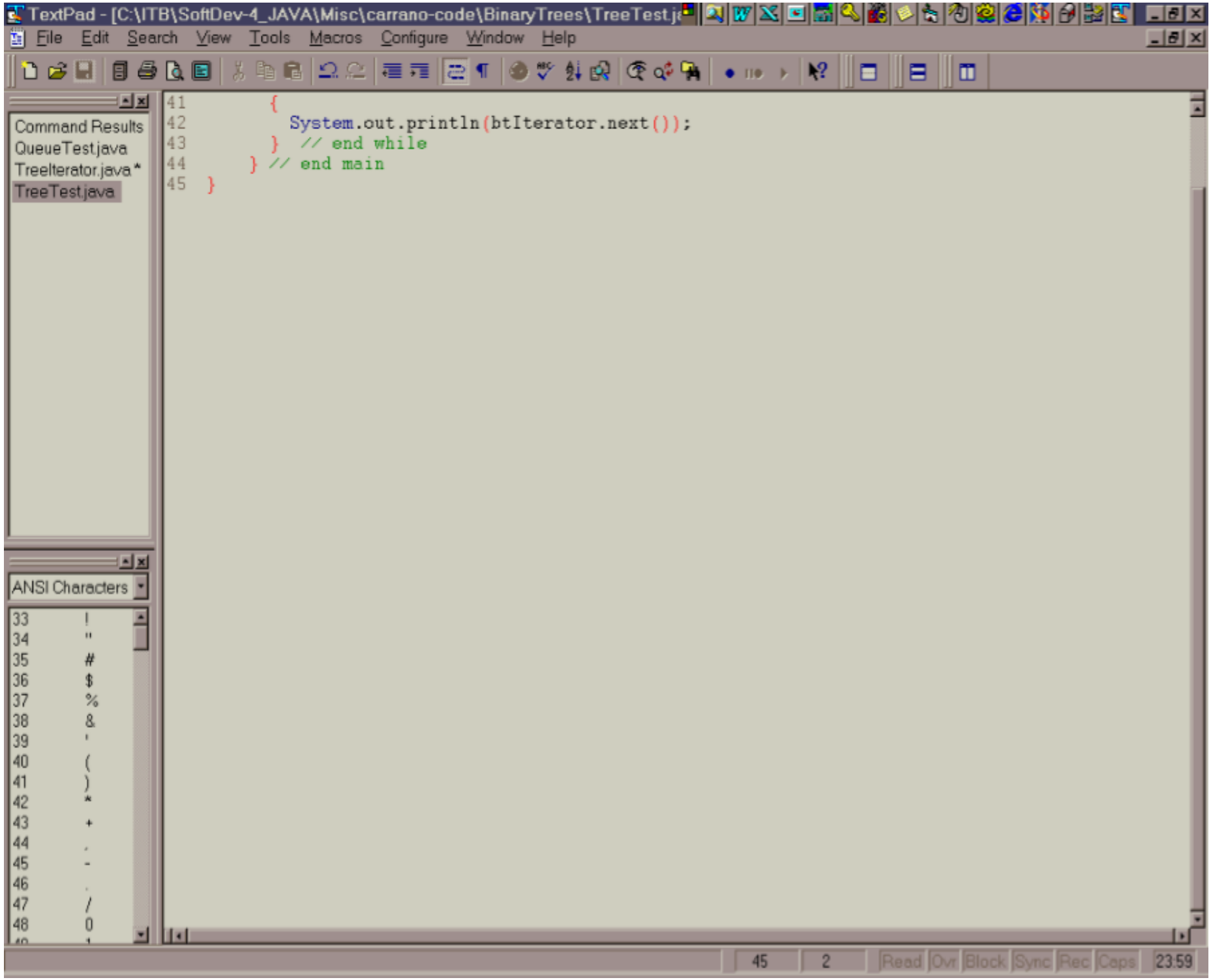
```
33  !
34  "
35  #
36  $
37  %
38  &
39  '
40  (
41  )
42  *
43  +
44  ,
45  -
46  .
47  /
48  0
```

```
1  public class TreeTest
2  {
3      public static void main(String[] args)
4      {
5        BinaryTree tree3 = new BinaryTree(new Integer(70));
6
7        // build the tree in diagram
8        BinaryTree tree1 = new BinaryTree();
9        tree1.setRootItem(new Integer(40));
10       tree1.attachLeft(new Integer(30));
11       tree1.attachRight(new Integer(50));
12
13       BinaryTree tree2 = new BinaryTree();
14       tree2.setRootItem(new Integer(20));
15       tree2.attachLeft(new Integer(10));
16       tree2.attachRightSubtree(tree1);
17
18       BinaryTree binTree = new BinaryTree(new Integer(60), tree2, tree3);// tree in Figure
19
20       TreeIterator btIterator = new TreeIterator(binTree);
21       btIterator.setInorder();
22
23       while (btIterator.hasNext())
24       {
25         System.out.println(btIterator.next());
26       } // end while
27
28       BinaryTree leftTree = binTree.detachLeftSubtree();
29       TreeIterator leftIterator = new TreeIterator(leftTree);
30
31       // iterate through the left subtree
32       leftIterator.setInorder();
33       while (leftIterator.hasNext())
34       {
35         System.out.println(leftIterator.next());
36       } // end while
37
38       // iterate through binTree minus left subtree
39       btIterator.setInorder();
40       while (btIterator.hasNext())
41       {
```

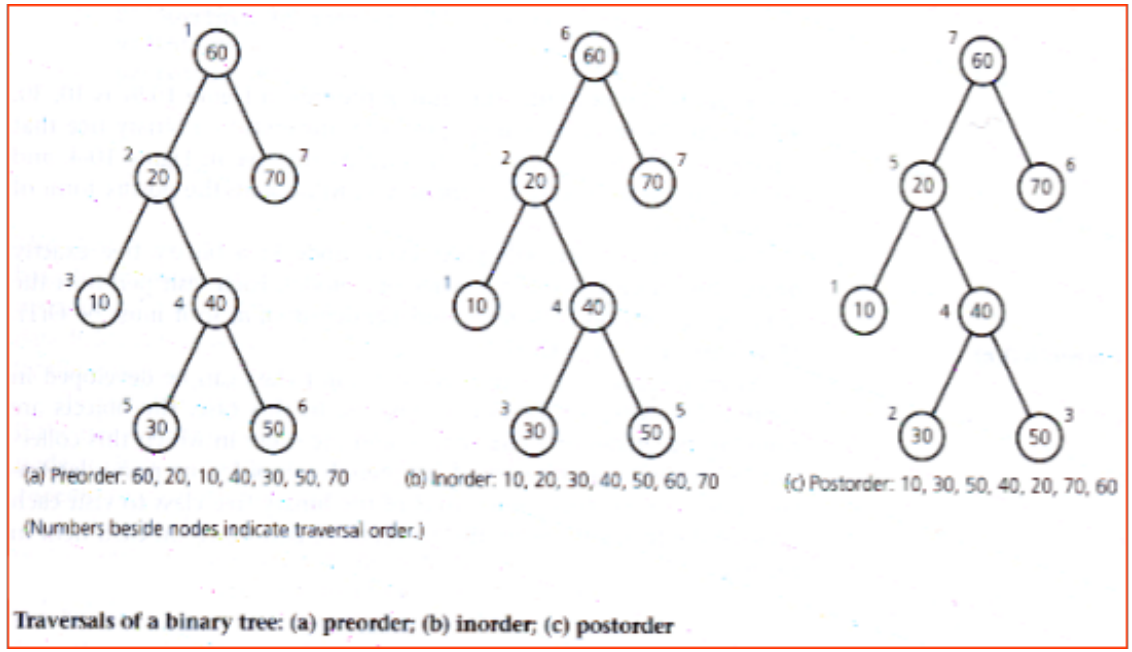
A sample program



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


TextPad - [C:\ITB\SoftDev-4_JAVA\Misc\carrano-code\BinaryTrees\TreeTest.j
File Edit Search View Tools Macros Configure Window Help
Command Results
QueueTest.java
TreeIterator.java *
TreeTest.java
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
{
System.out.println(btIterator.next());
} // end while
} // end main
}
45
2
Read
Ovr
Block
Sync
Rec
Caps
23:59

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



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

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) is10, 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.

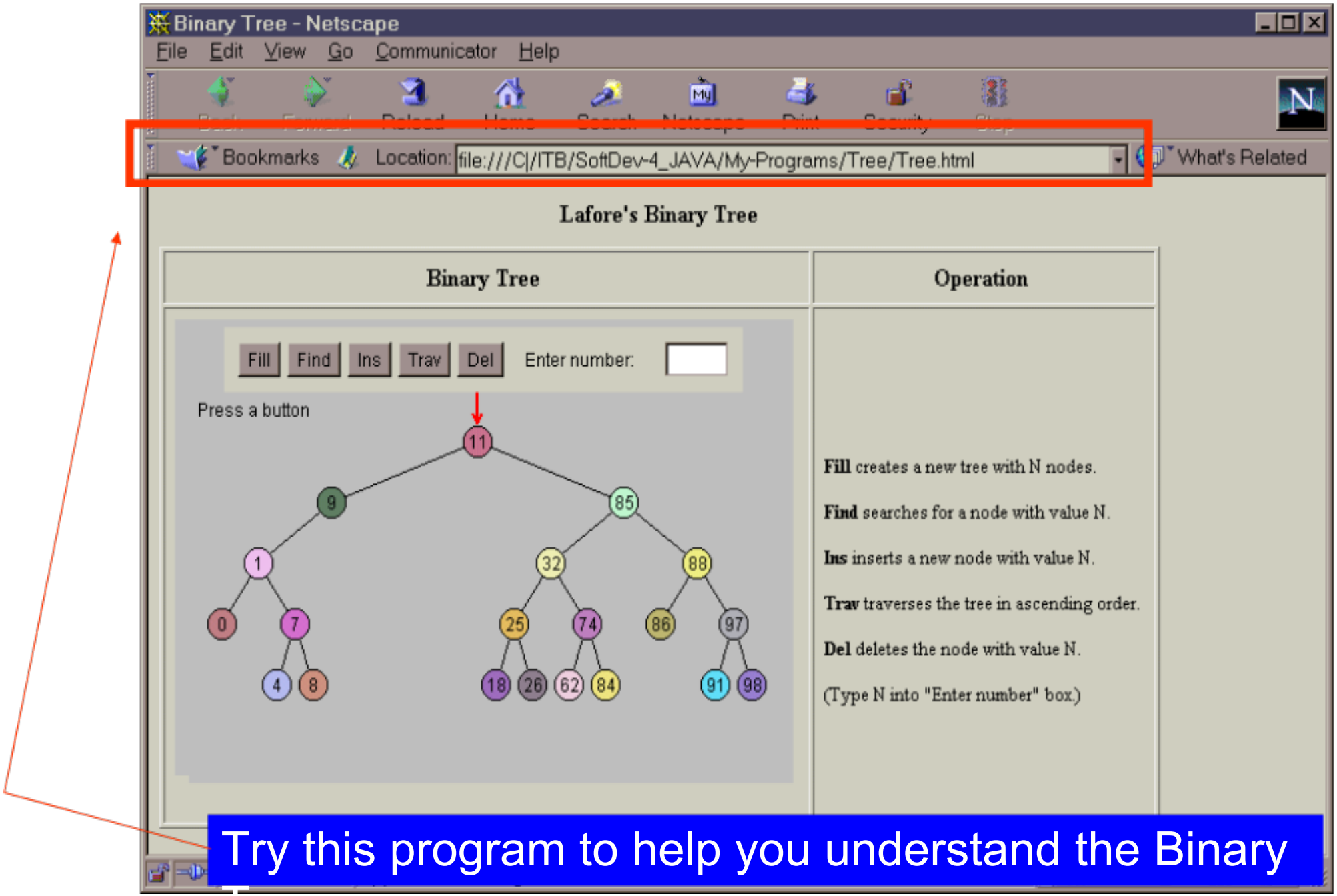
A **disadvantage of this implementation** of the traversal is that it performs a lot of computations that may never be used.

- 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
Operation
Fill Find Ins Trav Del Enter number:
Press a button
11
9 85
1 32 88
0 7 25 74 86 97
4 8 18 26 62 84 91 98
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