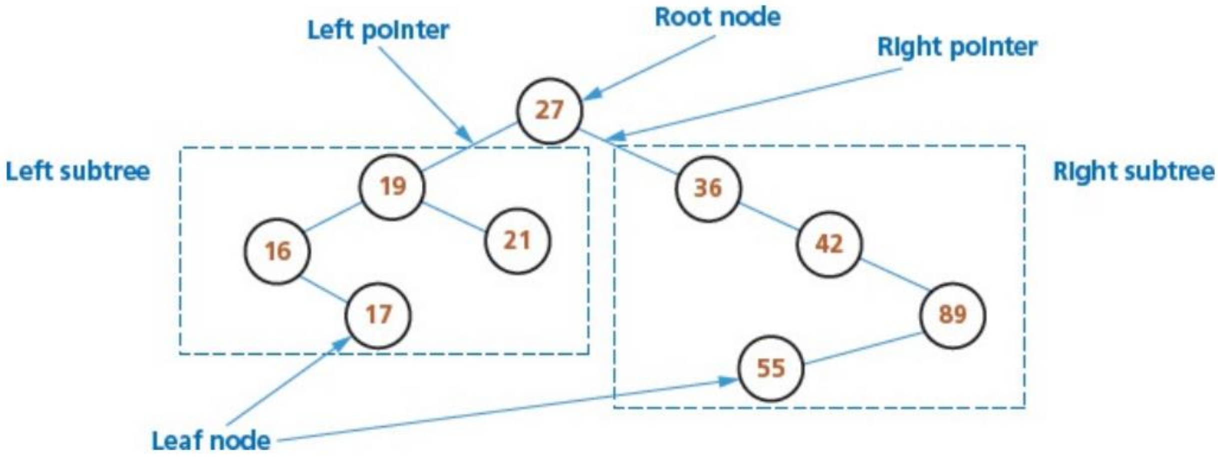
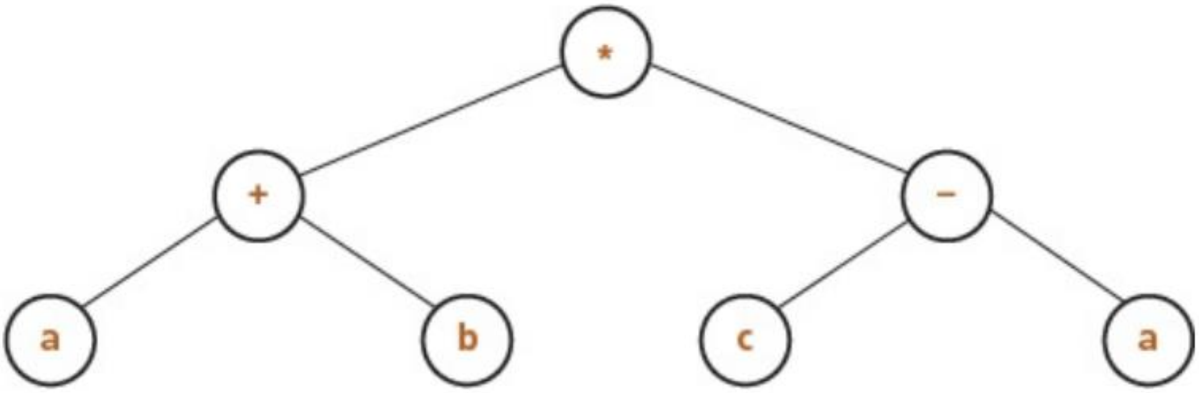
Chapter 19/20: Binary Trees

1. Binary Tree

- A **binary tree** is another frequently used **ADT**.
- It is a **hierarchical data structure** in which **each parent node** can have a maximum of **two child nodes**.
- There are many uses for binary trees; for example, they are used in
  - **syntax analysis**
  - **compression algorithms**
  - **3D video games**
- The binary tree for the data stored is **sorted in ascending order**.
- Each item is stored as a **node**.
- Each node can have up to **two branches**.
- If the **value** to be added is **less than the current node branch left**.
- If the **value** to be added is **greater than or equal to the current node branch right**.

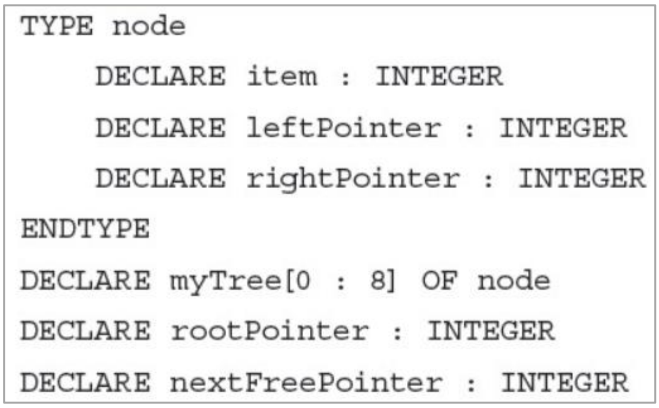


- A binary tree can also be used to represent an **arithmetic expression**.
- Consider **(a + b) * (c – a)**



ACTIVITY 19J
Draw the binary tree for the expression (x – y) / (x * y + z)

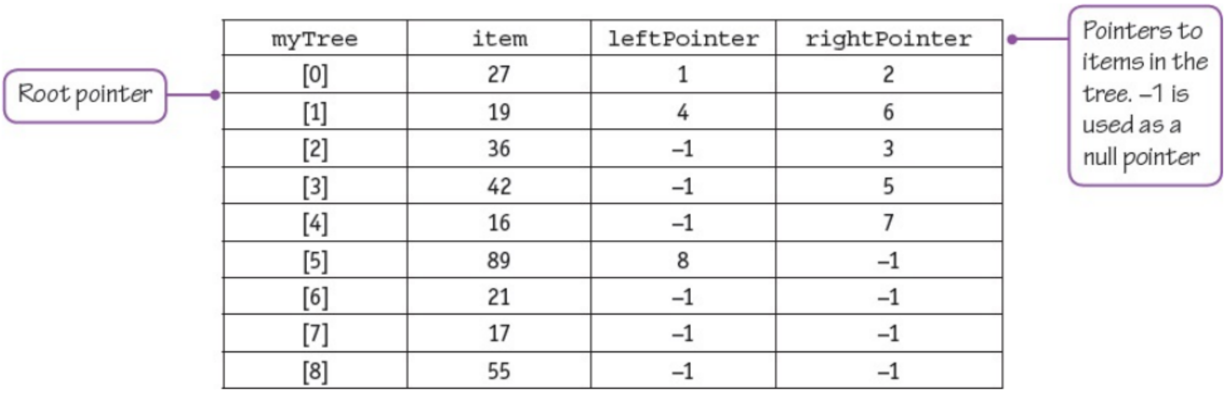
- The data structure for an **ordered binary tree** can be created in **pseudocode** as follows:



```
TYPE node
    DECLARE item : INTEGER
    DECLARE leftPointer : INTEGER
    DECLARE rightPointer : INTEGER
ENDTYPE
DECLARE myTree[0 : 8] OF node
DECLARE rootPointer : INTEGER
DECLARE nextFreePointer : INTEGER
```

ACTIVITY 19K
Create the data structure in pseudocode for a binary tree to store a list of names. Your list must be able to store at least 50 names.

- The populated contents of the data structure **myTree** is shown below:



myTree	item	leftPointer	rightPointer
[0]	27	1	2
[1]	19	4	6
[2]	36	–1	3
[3]	42	–1	5
[4]	16	–1	7
[5]	89	8	–1
[6]	21	–1	–1
[7]	17	–1	–1
[8]	55	–1	–1

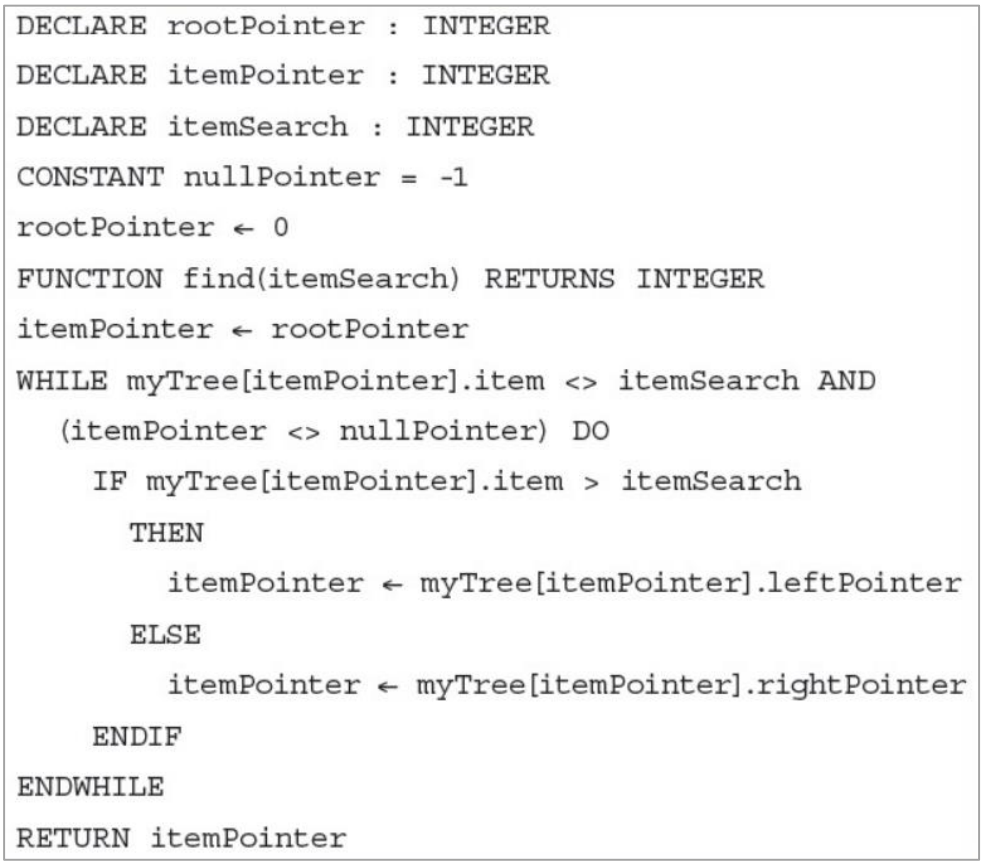
Root pointer

Pointers to items in the tree. –1 is used as a null pointer

- The **root pointer** points to the **first node** in a binary tree.
- A **null pointer** is a **value** stored in the **left** or **right pointer** in a binary tree to indicate that there are **no nodes below this node** on the **left** or **right**.

2. Finding an Item in a Binary Tree

- The algorithm to **find if an item is in the binary tree** myTree and return **the pointer to its node if found** or a **null pointer if not found**, could be written as a function in pseudocode:



```
DECLARE rootPointer : INTEGER
DECLARE itemPointer : INTEGER
DECLARE itemSearch : INTEGER
CONSTANT nullPointer = -1
rootPointer ← 0
FUNCTION find(itemSearch) RETURNS INTEGER
itemPointer ← rootPointer
WHILE myTree[itemPointer].item <> itemSearch AND
   (itemPointer <> nullPointer) DO
     IF myTree[itemPointer].item > itemSearch
       THEN
         itemPointer ← myTree[itemPointer].leftPointer
       ELSE
         itemPointer ← myTree[itemPointer].rightPointer
     ENDIF
ENDWHILE
RETURN itemPointer
```

- Here is the **identifier table** for the **binary tree search** algorithm shown above.

Identifier	**Description**
myTree	Tree to be searched
node	ADT for tree
rootPointer	Pointer to the start of the tree
leftPointer	Pointer to the left branch
rightPointer	Pointer to the right branch
nullPointer	Null pointer set to −1
itemPointer	Pointer to current item
itemSearch	Item being searched for

- The **trace table** below shows the algorithm being used to **search for 42** in myTree.

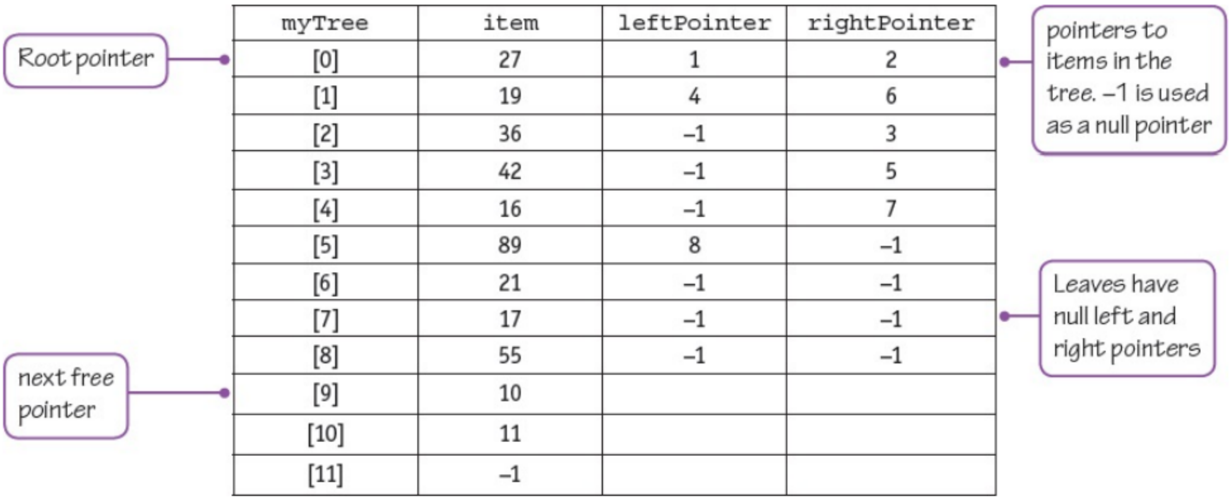
rootPointer	itemPointer	itemSearch
0	0	42
	2	
	3	

ACTIVITY 19L

Use the algorithm to search for 55 and 75 in myTree. Show the results of each search in a trace table.

3. Inserting items into a Binary Tree

- The binary tree. **needs free nodes to add new items**.
- For example, **myTree** below, now has room for **12 items**.
- The **last three nodes** have **not been filled** yet.
- There is a **pointer** to the **next free node** and the free nodes are set up like a **heap** in a **linked list**, using the **left pointer**.



Root pointer

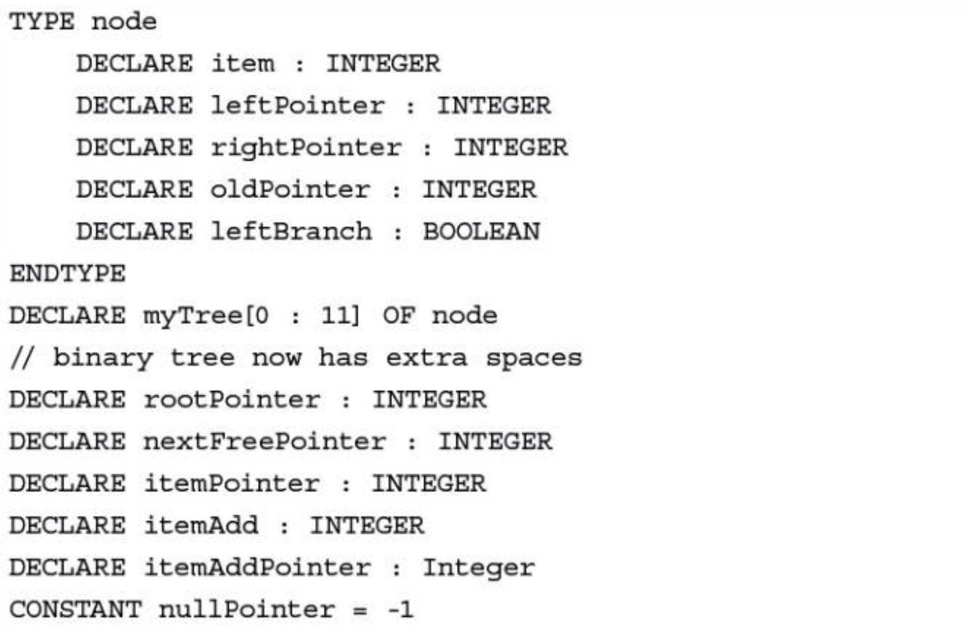
next free pointer

myTree	item	leftPointer	rightPointer
[0]	27	1	2
[1]	19	4	6
[2]	36	–1	3
[3]	42	–1	5
[4]	16	–1	7
[5]	89	8	–1
[6]	21	–1	–1
[7]	17	–1	–1
[8]	55	–1	–1
[9]	10		
[10]	11		
[11]	–1		

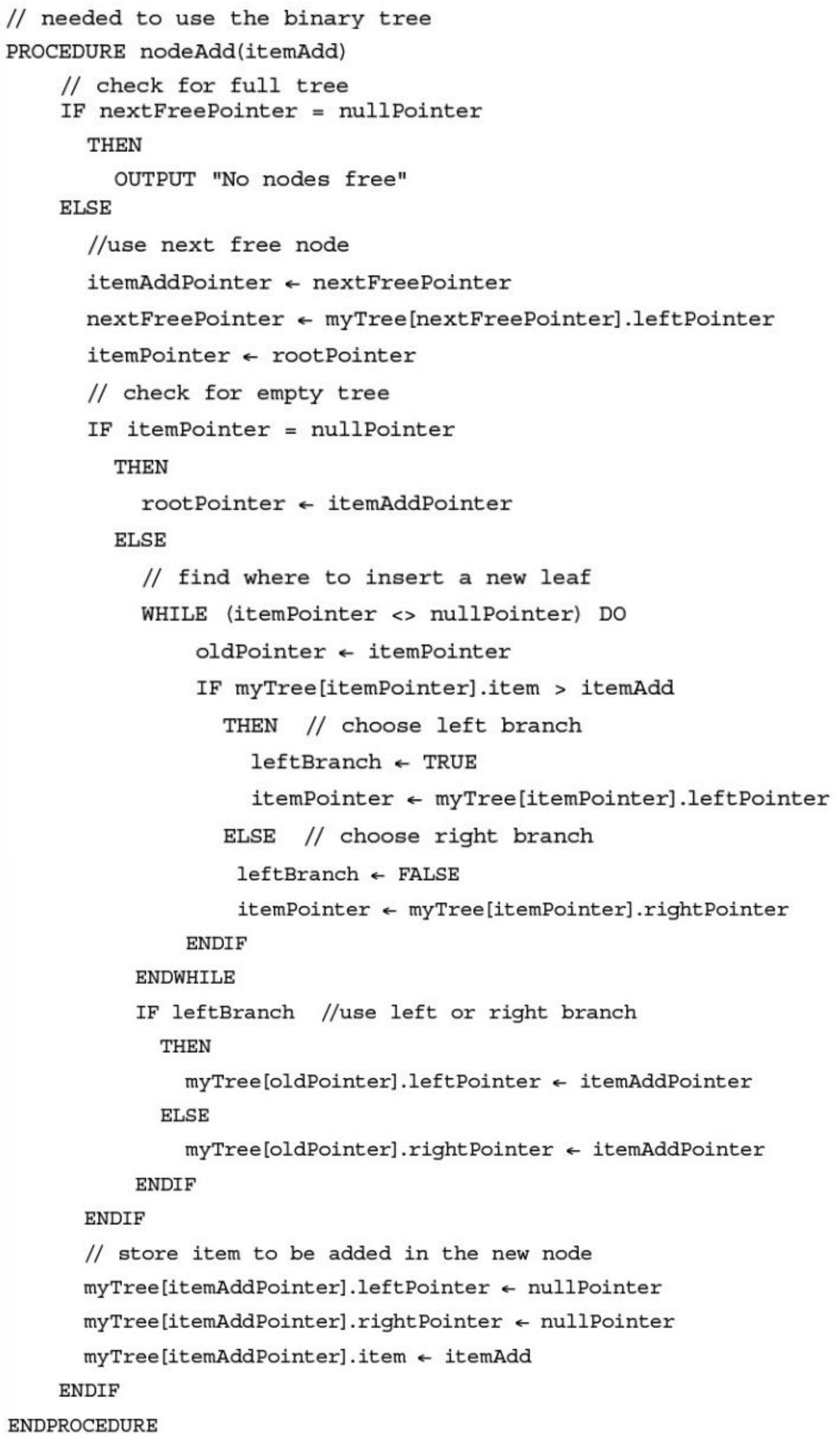
pointers to items in the tree. –1 is used as a null pointer

Leaves have null left and right pointers

- The algorithm to **insert an item at a new node** in the binary tree **myTree** could be written as a **procedure** in **pseudocode** as shown below:



```
TYPE node
    DECLARE item : INTEGER
    DECLARE leftPointer : INTEGER
    DECLARE rightPointer : INTEGER
    DECLARE oldPointer : INTEGER
    DECLARE leftBranch : BOOLEAN
ENDTYPE
DECLARE myTree[0 : 11] OF node
// binary tree now has extra spaces
DECLARE rootPointer : INTEGER
DECLARE nextFreePointer : INTEGER
DECLARE itemPointer : INTEGER
DECLARE itemAdd : INTEGER
DECLARE itemAddPointer : Integer
CONSTANT nullPointer = -1
```



```
// needed to use the binary tree
PROCEDURE nodeAdd(itemAdd)
    // check for full tree
    IF nextFreePointer = nullPointer
      THEN
        OUTPUT "No nodes free"
    ELSE
      //use next free node
      itemAddPointer ← nextFreePointer
      nextFreePointer ← myTree[nextFreePointer].leftPointer
      itemPointer ← rootPointer
      // check for empty tree
      IF itemPointer = nullPointer
        THEN
          rootPointer ← itemAddPointer
        ELSE
          // find where to insert a new leaf
          WHILE (itemPointer <> nullPointer) DO
              oldPointer ← itemPointer
              IF myTree[itemPointer].item > itemAdd
                THEN  // choose left branch
                  leftBranch ← TRUE
                  itemPointer ← myTree[itemPointer].leftPointer
                ELSE  // choose right branch
                 leftBranch ← FALSE
                 itemPointer ← myTree[itemPointer].rightPointer
             ENDIF
          ENDWHILE
          IF leftBranch  //use left or right branch
            THEN
              myTree[oldPointer].leftPointer ← itemAddPointer
            ELSE
              myTree[oldPointer].rightPointer ← itemAddPointer
          ENDIF
      ENDIF
      // store item to be added in the new node
      myTree[itemAddPointer].leftPointer ← nullPointer
      myTree[itemAddPointer].rightPointer ← nullPointer
      myTree[itemAddPointer].item ← itemAdd
    ENDIF
ENDPROCEDURE
```

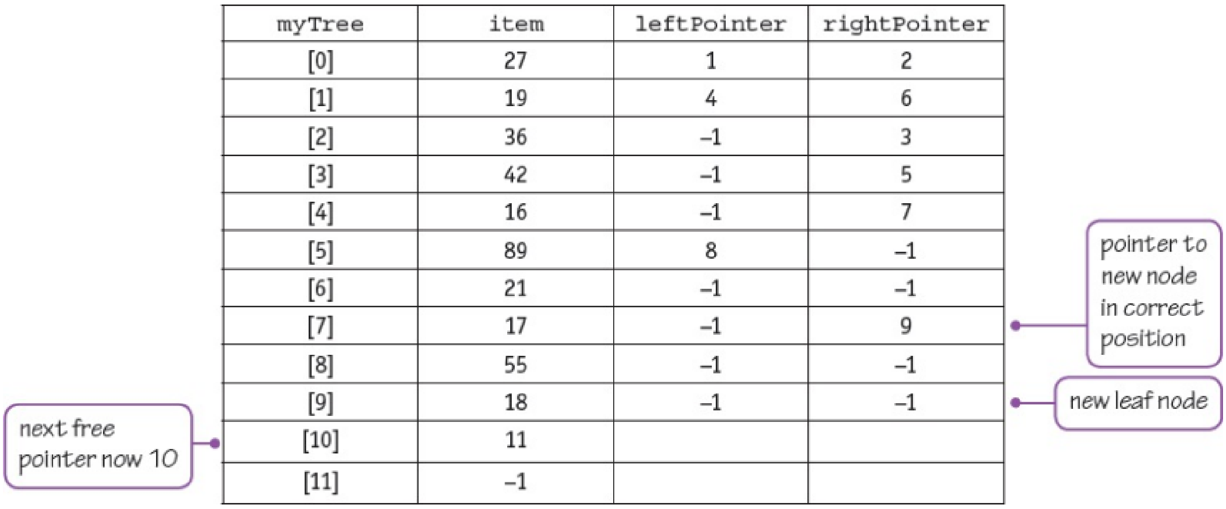
- Here is the **identifier table**:

Identifier	Description
myTree	Tree to be searched
node	ADT for tree
rootPointer	Pointer to the start of the tree
leftPointer	Pointer to the left branch
rightPointer	Pointer to the right branch
nullPointer	Null pointer set to −1
itemPointer	Pointer to current item
itemAdd	Item to add to tree
nextFreePointer	Pointer to next free node
itemAddPointer	Pointer to position in tree to store item to be added
oldPointer	Pointer to leaf node that is going to point to item added
leftBranch	Flag to identify whether to go down the left branch or the right branch

- The **trace table** below shows the algorithm being used to **add 18** to **myTree**.

leftBranch	nextFreePointer	itemAddPointer	rootPointer	itemAdd	itemPointer	oldPointer
	Already set to 9	9	Already set to 0	18		
	10				0	0
TRUE					1	1
TRUE					4	4
FALSE					7	7
					-1	

- The tree, **myTree** will now be as shown below:



myTree	item	leftPointer	rightPointer
[0]	27	1	2
[1]	19	4	6
[2]	36	–1	3
[3]	42	–1	5
[4]	16	–1	7
[5]	89	8	–1
[6]	21	–1	–1
[7]	17	–1	9
[8]	55	–1	–1
[9]	18	–1	–1
[10]	11		
[11]	–1		

pointer to new node in correct position

new leaf node

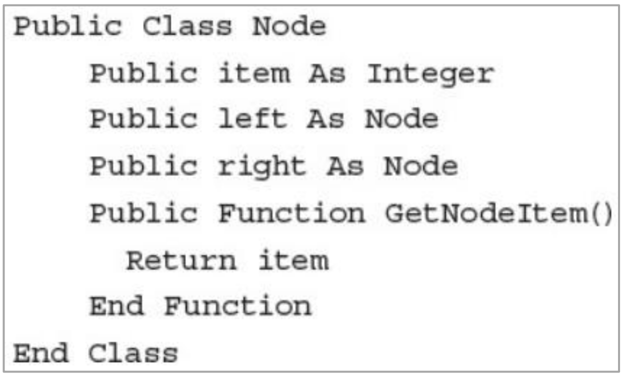
next free pointer now 10

ACTIVITY 19M
Use the algorithm to add 25 to myTree. Show this in a trace table and show myTree once 25 has been added.

4. Writing a program for a Binary Tree

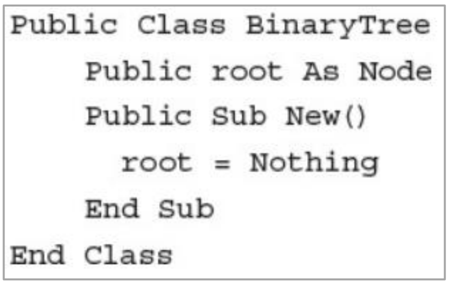
- **Binary trees** are best implemented using:
  - **objects -** tree and node
  - **constructors -** adding a new node to the tree
  - **containment -** the tree contains nodes
  - **functions -** search the binary tree for an item
  - **procedures -** insert a new item in the binary tree
  - **recursion**

- **Binary tree data structure – Class node**
  - VB with a **recursive** definition of **node** to allow for a **tree of any size**



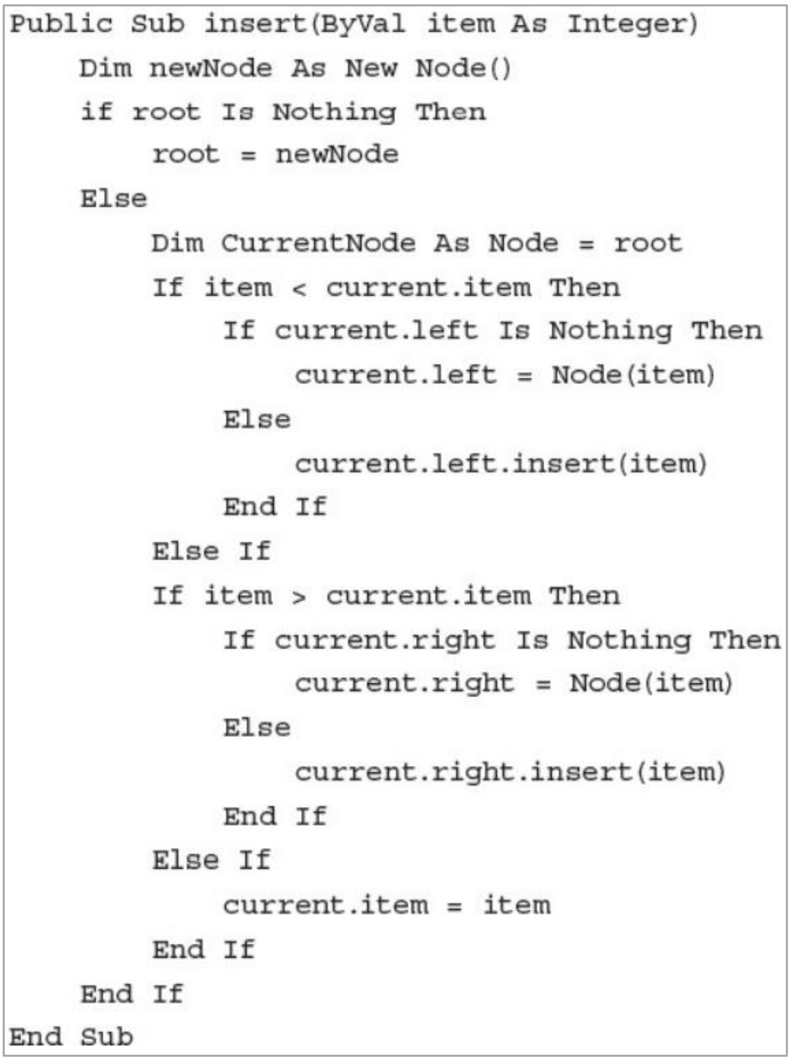
```
Public Class Node
    Public item As Integer
    Public left As Node
    Public right As Node
    Public Function GetNodeItem()
      Return item
    End Function
End Class
```

- **Binary tree data structure – Class tree**
  - VB uses **Nothing** for **null pointers**



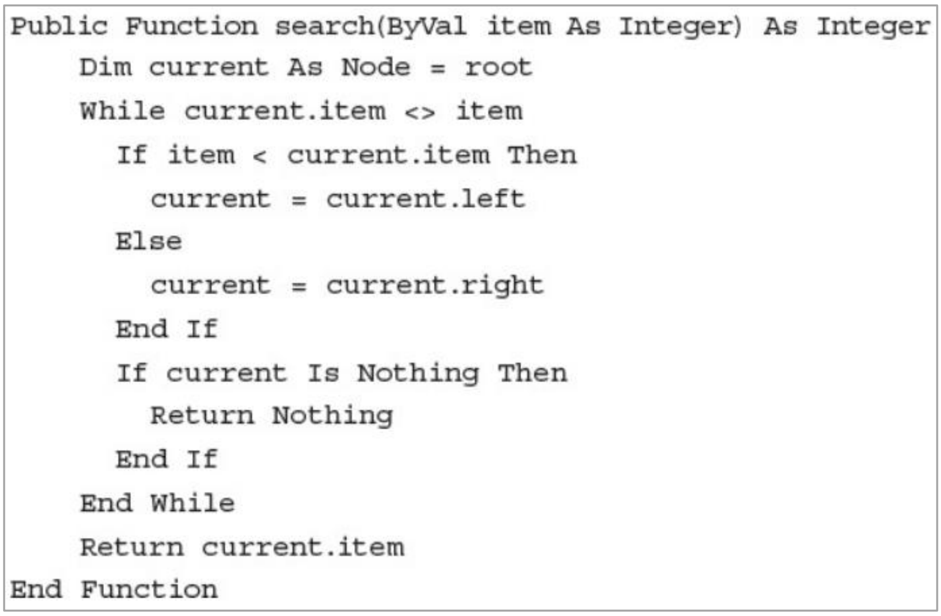
```
Public Class BinaryTree
    Public root As Node
    Public Sub New()
      root = Nothing
    End Sub
End Class
```

- **Add integer to binary tree**
    - VB showing a **recursive** procedure to **insert a new node**



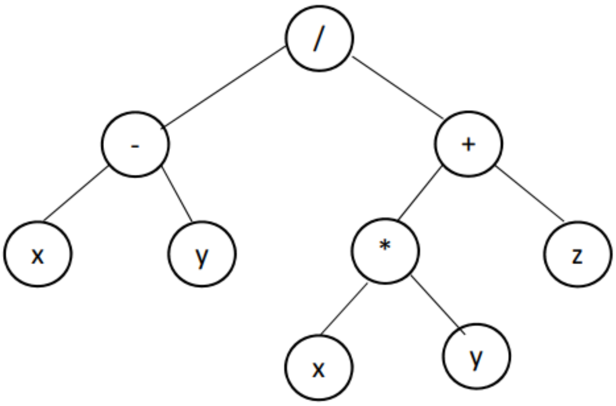
```
Public Sub insert(ByVal item As Integer)
    Dim newNode As New Node()
    if root Is Nothing Then
        root = newNode
    Else
        Dim CurrentNode As Node = root
        If item < current.item Then
            If current.left Is Nothing Then
                current.left = Node(item)
            Else
                current.left.insert(item)
            End If
        Else If
        If item > current.item Then
            If current.right Is Nothing Then
                current.right = Node(item)
            Else
                current.right.insert(item)
            End If
        Else If
            current.item = item
        End If
    End If
End Sub
```

- **Search for integer in binary tree**
    - VB – the function returns **the value searched** for if it is **found**, otherwise it returns **Nothing**

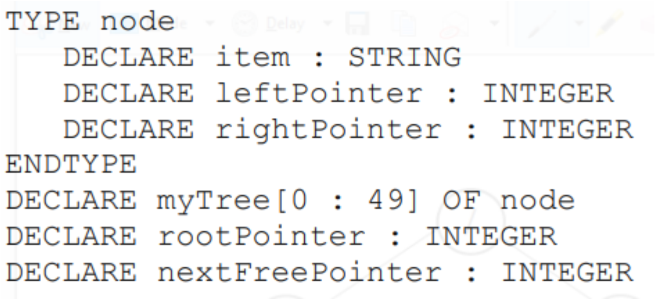


```
Public Function search(ByVal item As Integer) As Integer
    Dim current As Node = root
    While current.item <> item
      If item < current.item Then
        current = current.left
      Else
        current = current.right
      End If
      If current Is Nothing Then
        Return Nothing
      End If
    End While
    Return current.item
End Function
```

ACTIVITY 19J - ANSWER



ACTIVITY 19K - ANSWER



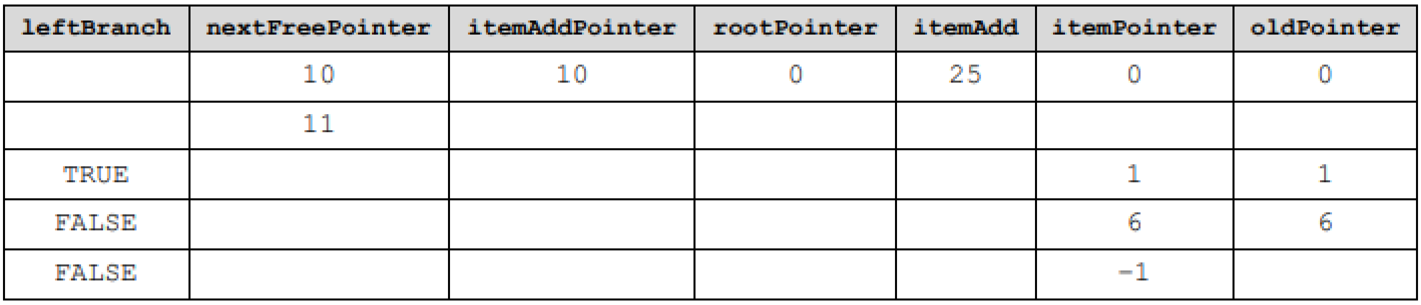
```
TYPE node
   DECLARE item : STRING
   DECLARE leftPointer : INTEGER
   DECLARE rightPointer : INTEGER
ENDTYPE
DECLARE myTree[0 : 49] OF node
DECLARE rootPointer : INTEGER
DECLARE nextFreePointer : INTEGER
```

ACTIVITY 19L - ANSWER

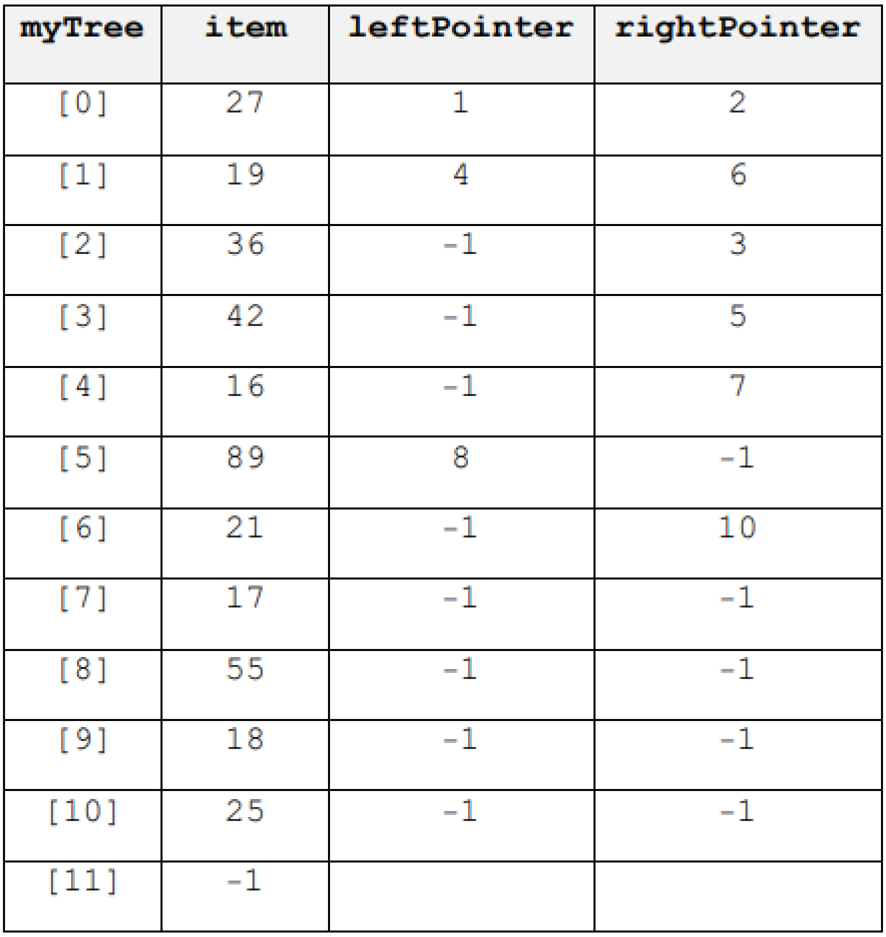
rootPointer	itemPointer	itemSearch
0	0	55
	2	
	3	
	5	
	8	

rootPointer	itemPointer	itemSearch
0	0	75
	2	
	3	
	5	
	-1	

ACTIVITY 19M - ANSWER



leftBranch	nextFreePointer	itemAddPointer	rootPointer	itemAdd	itemPointer	oldPointer
	10	10	0	25	0	0
	11					
TRUE					1	1
FALSE					6	6
FALSE					−1	



myTree	item	leftPointer	rightPointer
[0]	27	1	2
[1]	19	4	6
[2]	36	-1	3
[3]	42	-1	5
[4]	16	-1	7
[5]	89	8	-1
[6]	21	-1	10
[7]	17	-1	-1
[8]	55	-1	-1
[9]	18	-1	-1
[10]	25	-1	-1
[11]	-1		