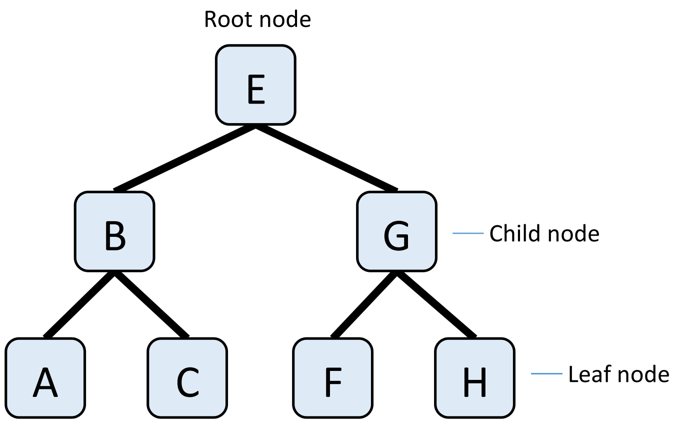
Binary Tree

OCR AS Level	OCR A Level	AQA AS Level	AQA A Level
WJEC AS Level	WJEC A Level	BTEC Nationals	BTEC Highers

A binary tree is a data structure consisting of nodes and pointers. It is a special case of a graph where each node can only have zero, one or two pointers. Each pointer connects to a different node.

The first node is known as the root node. The connected nodes are known as child nodes, with the nodes at the very bottom of the structure being called leaf nodes.



Since a binary tree is essentially just a graph, the nodes can also be referred to as vertices and the pointers, edges. Binary trees can be represented in memory with dictionaries:

```
tree = {"E":["B","G"],"B":["A","C"],"G":["F","H"],"A":[],"C":[],"F":[],"H":[]}
```

See the chapter titled graphs for an alternative implementation with associated breadth and depth first algorithms that can also be applied to binary trees.

It is more common to see binary trees represented as objects consisting of a node, it's data, with a left and right pointer. Typically, when implementing the binary tree, a node has it's left, and/or right pointer set to be nothing or null if there is no child node.

Applications of a binary tree

Binary trees have many uses in computer science. These include database applications where efficient searching and sorting is necessary without moving items which is slow to execute. They can also be found in wireless networking, router tables and scheduling processes in operating systems. Implementations of the A* pathfinding algorithm can use trees. Huffman coding used for compression algorithms such as jpeg and mp3, and cryptography (GGM trees) all make use of binary trees. They are also useful for performing arithmetic using reverse polish notation which negates the need to use brackets to define the order of precedence for operators in an expression.

Operations on a binary tree

Typical operations that can be performed on a binary tree include:

- Add: adds a new node to the tree.
- Delete: removes a node from the tree.
- Binary search: returns the data stored in a node.
- Pre-order traversal: a type of depth first search.
- In-order traversal: a type of depth first search.
- Post-order traversal: a type of depth first search.
- Breadth first search: traverses the tree starting at the root node, visiting each node at the same level before going deeper into the structure.

A **traversal** refers to the process of visiting each node in a binary tree only once. It is also a term commonly used with graphs. A traversal is required to find, update and output the data in a node.

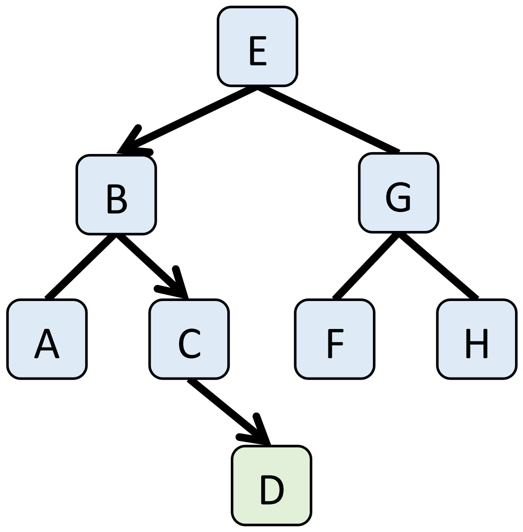
Craig'n'Dave videos

https://www.craigndave.org/algorithm-binary-tree

Adding an item to a binary tree in simple structured English

1. Check there is free memory for a new node. Output an error if not.
2. Create a new node and insert data into it.
3. If the binary tree is empty:
    a. The new node becomes the first item. Create a start pointer to it.
4. If the binary tree is not empty:
    a. Start at the root node.
    b. If the new node should be placed before the current node, follow the left pointer.
    c. If the new node should be placed after the current node, follow the right pointer.
    d. Repeat from step 4b until the leaf node is reached.
    e. If the new node should be placed before the current node, set the left pointer to be the new node.
    f. If the new node should be placed before the current node, set the right pointer to be the new node.

Adding an item to a binary tree illustrated



Start at the root node. D is less than E, follow the left pointer. D is more than B, follow the right pointer. D is more than C, create a right pointer from C to D.

Did you know?

The maximum efficiency with binary trees is achieved when the tree is balanced. That means one branch is not significantly larger than another. A special case of a binary tree called a self-balancing binary tree automatically keeps its height small when items are added and deleted by moving nodes within the structure to maintain the balance.

A self-balancing operation decreases the efficiency of adding and deleteing items, but increases the efficiency of searching the structure. In this case it is a very efficient way to implement a dictionary.

Pseudocode for adding an item to a binary tree

```
If not memoryfull Then
       new_node = New Node
       new_node.left_pointer = Null
       new_node.right_pointer = Null
       current_node = start_pointer
       If current_node == Null Then
              start_pointer = new_node
       Else
              While current_node != Null
                     previous_node = current_node
                     If new_node < current_node Then
                            current_node = current_node.left_pointer
                     Else
                            current_node = current_node.right_pointer
                     End If
              End While
              If current_node < previous_node Then
                     previous_node.left_pointer = new_node
              Else
                     previous_node.rigth_pointer = new_node
              End If

       End If
End If
```

Did you know?

Like all data structures, a binary tree can be represented using an array. A binary tree requires a two dimensional array.

The root node E is stored at the first index because items are entered into the array as they are added. A feature of a binary tree is that items can be traversed in order even though they were not entered in order. A binary tree negates the need for a sorting algorithm.

Node E is connected to B (index 1) and G (index 3).

0	E	1	3
1	B	2	4
2	A		
3	G	5	6
4	C		7
5	F		
6	H		
7	D		

Deleting an item from a binary tree in simple structured English

Firstly, we need to find the node to delete in the structure.

1. Start at the root node
2. Whilst the current node exists, and the current node is not the one to delete:
    a. Set the previous node to be the current node.
    b. If the item to be deleted is less than the current node, follow the left pointer.
    c. If the item to be deleted is greater than the current node, follow the left pointer.

Assuming the node exists and therefore has been found, there are three possibilities that need to be considered when deleting a node from a binary tree:

i. The node is a leaf node and has no children.
ii. The node has one child.
iii. The node has two children.

The node has no children

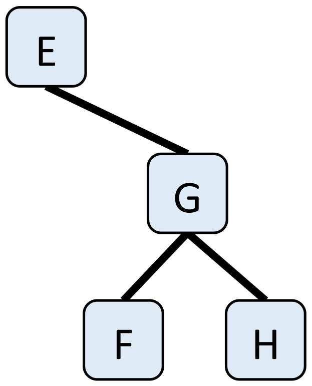
3. If the previous node is greater than the current node then the previous node's left pointer is set to null.
4. If the previous node is less than the current node then the previous node's right pointer is set to null.

The node has one child

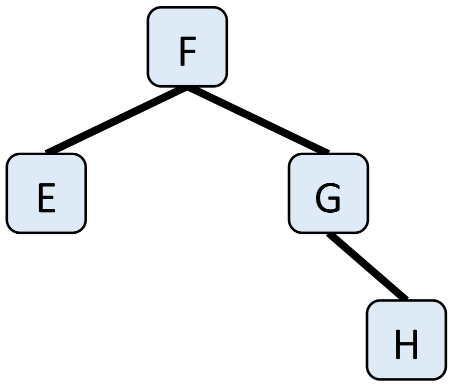
5. If the current node is on the left of the previous node:
    a. Set the previous node's left pointer to the current node's left child.
6. If the current node is on the right of the previous node:
    a. Set the previous node's right pointer to the current node's right child.

The node has two children

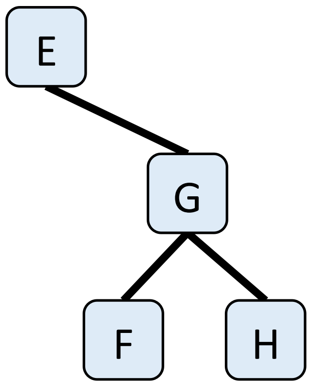
In this situation we can make use of the fact that the data in a binary tree can be represented in a different way. For example:



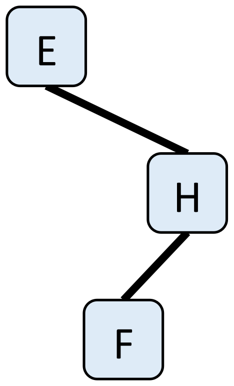
Is the same as…



Therefore, one approach to deleting node G is to find the smallest value in the right sub-tree (known as the successor), and move it to the position occupied by G. The smallest value in the right sub-tree will always be the left-most node in the right-sub tree. The approach is known as the Hibbard deletion algorithm. In the example above, there is a special case because there is no left sub-tree from node H. Therefore, we can move H into the position occupied by G.



Becomes…



Notice the tree has become unbalanced when this happens since F could now be the root node, with E to the left and H to the right. Therefore, there is an impact on the efficiency of algorithms on binary trees when nodes are added and deleted over time.

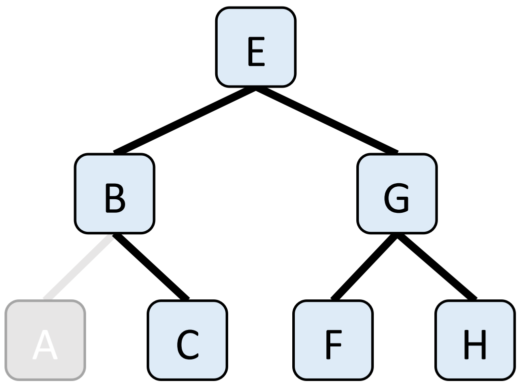
As with all algorithms, there are alternative approaches depending on how the structure is implemented by the programmer. One alternative could be to use the predecessor (right-most node in the left sub-tree) instead of the successor node. A simple alternative would be to introduce another attribute to each node to simply flag whether the node is deleted or not and skip deleted nodes when traversing the tree. This approach does increase the space complexity as nodes are added and deleted though.

Hibbard deletion

7. If a right node exists find the smallest leaf node in the right sub-tree
    a. Change the current node to the smallest leaf node.
    b. Remove the smallest leaf node.
8. If there is no left sub-tree from the right node then
    a. Set the current node to be the current node's right pointer.
    b. Set the current node's right pointer to be null.

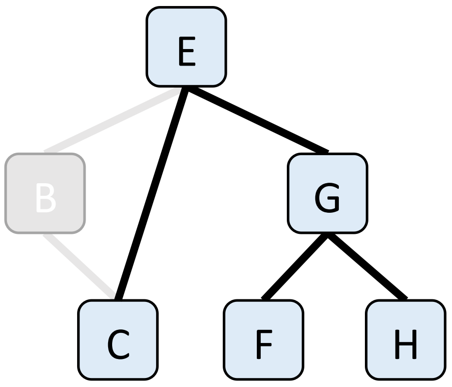
Deleting an item to a binary tree illustrated

Node has no children:



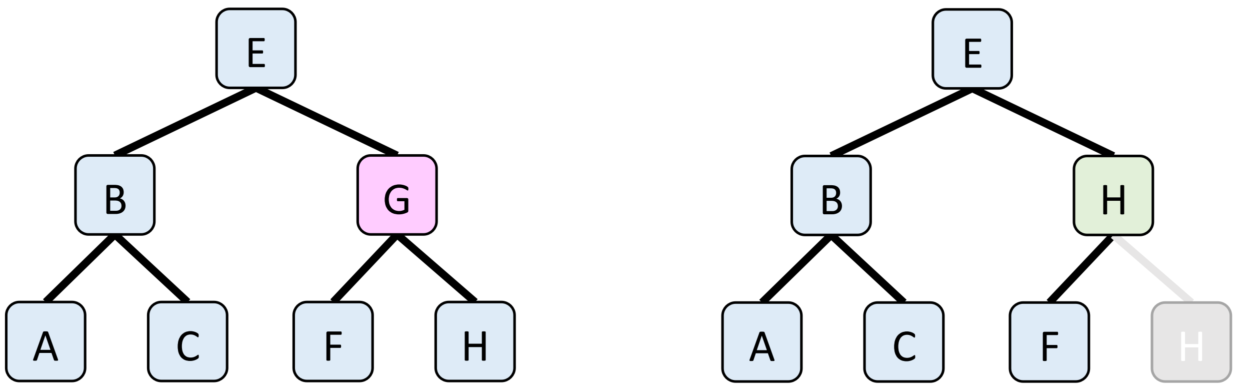
The node can simply be removed, and the previous node's left or right pointer accordingly is set to null.

Node has one child:



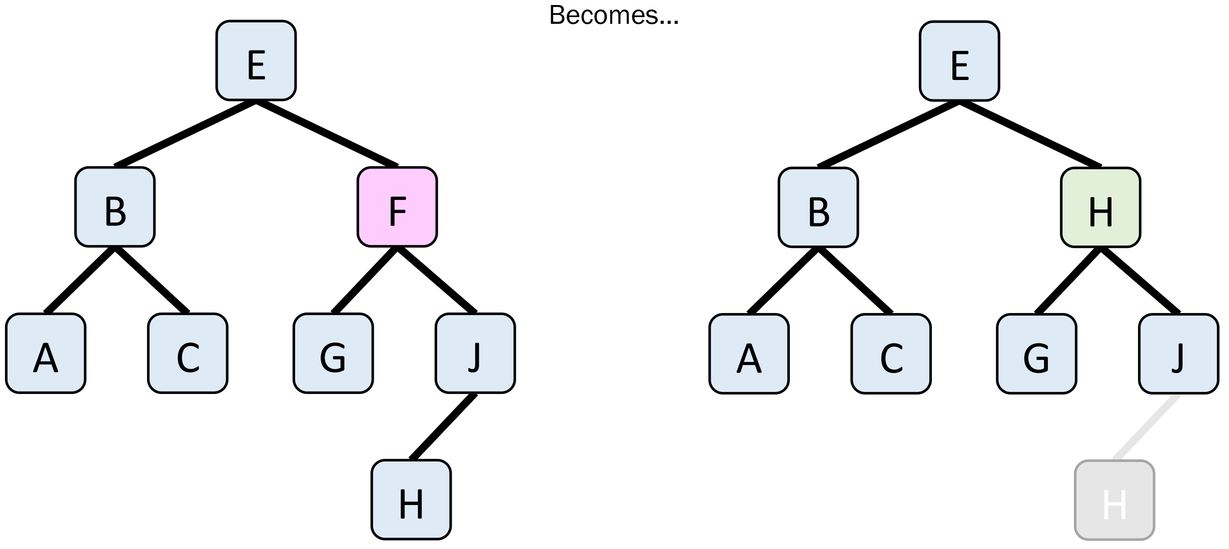
Here B is illustrated as being subsequently deleted. The previous node's left or right pointer is set to the left or right pointer of the node being deleted. E's left pointer is now C.

G is to be deleted. It has two children:



In this special case, there is no left sub-tree from H, so G is replaced with H and its right pointer is set to null. An example with a left sub-tree is shown below.

F is to be deleted. It has two children:



Here F is illustrated as being deleted. The lowest node in the right sub-tree is H. F is replaced with H, and J's left pointer is set to null. A mistake that can be made here is assuming it is always the first left node from the right sub-tree that is found. That is not the case. It is the left-most leaf node that is swapped. Therefore, if H had a left pointer, that would be followed until the leaf node is found.

Pseudocode for deleting an item from a binary tree

```
current_node = root node
while current_node != null and current_node != item
    previous_node = current_node
    if item < current_node.data Then
        current_node = current_node.left_pointer
    Else
        current_node = current_node.right_pointer
    End If
If current_node != null then
    If current_node.left_pointer == null and current_node.right_pointer == null Then
        If previous_node.data > current_node.data Then
            previous_node.left_pointer = null
        Else
             previous_node.right_pointer = null
        End If
        Elseif current_node.right_pointer == null Then
            If previous_node.data > current_node.data Then
                previous_node.left_pointer = current_node.left_pointer
            Else
                previous_node.right_pointer = current_node.left_pointer
            End If
            If previous_node.data < current_node.data Then
                previous_node.left_pointer = current_node.right_pointer
            Else
                previous_node.right_pointer = current_node.right_pointer
            End If
        Else
            right_node = current_node.right_pointer
            If right_node.left_pointer != null Then
                smallest_node = right_node
                While smallest_node.left_pointer != null
                    previous_node = smallest_node
                    smallest_node = smallest_node.right_pointer
                End While
                current_node.data = smallest_node.data
                previous_node.left_pointer = null
            End If
        End If
End If
```

Pre-order traversal (depth first search) in simple structured English

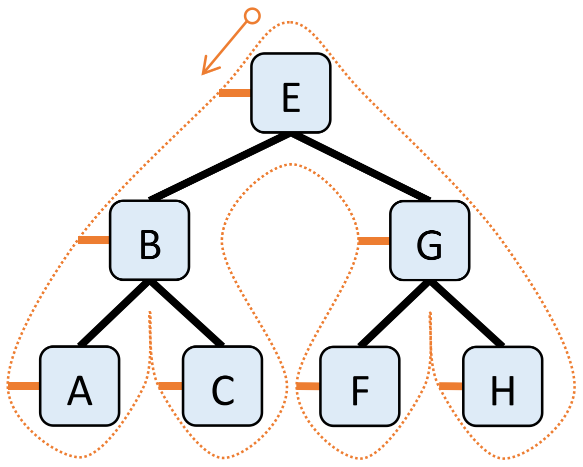
A pre-order traversal, which is a variation of a depth first search is used to create a copy of a binary tree, or to return prefix expressions in Polish notation which can be used by programming language interpreters to evaluate syntax. It is worth remembering that traversals can also be performed on graphs.

The algorithm can be described as: Node-Left-Right:

1. Start at the root node.
2. Output the node.
3. Follow the left pointer and repeat from step 2 recursively until there is no pointer to follow.
4. Follow the right pointer and repeat from step 2 recursively until there is no pointer to follow.

Pre-order traversal illustrated

If illustrations help you remember, this is how you can picture a pre-order traversal on a binary tree:



Note the markers on the left side of each node. As you traverse the tree, starting from the root, the nodes are only output when you pass the marker: E, B, A, C, G, F, H. You can illustrate like this in exams to demonstrate your understanding of the algorithm.

Pseudocode for a pre-order traversal

```
Procedure preorder(current_node)
       If current_node != null Then
              Print(current_node.data)
              If current_node.left_pointer != null Then
                     preorder(current_node.left_pointer)
              End If
              If current_node.right_pointer != null Then
                     preorder(current_node.right_pointer)
              End If
       End If
End procedure
```

In-order traversal (depth first search) in simple structured English

An in-order traversal, which is a variation of a depth first search is used to output the contents of the binary tree in order. One of the significant advantages of the binary tree is that it automatically sorts the contents of the structure without moving data, irrespective of the order in which the data arrived. This negates the need for an insertion sort for example. It is worth remembering that traversals can also be performed on graphs.

The algorithm can be described as: Left-Node-Right:

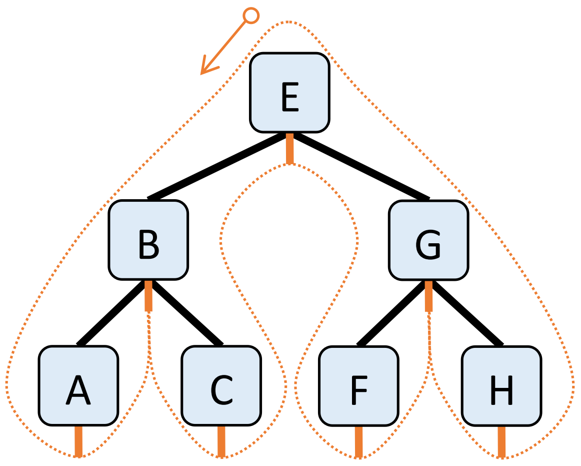
1. Start at the root node.
2. Follow the left pointer and repeat from step 2 recursively until there is no pointer to follow.
3. Output the node.
4. Follow the right pointer and repeat from step 2 recursively until there is no pointer to follow.

Did you know?

A common mistake is to forget that all traversals including pre-order, in-order and post-order always follow the left path first unless you want to output the items in reverse order.

In-order traversal illustrated

If illustrations help you remember, this is how you can picture an in-order traversal on a binary tree:



Note the markers on the bottom of each node. As you traverse the tree, starting from the root, the nodes are only output when you pass the marker: A, B, C, E, F, G, H. You can illustrate like this in exams to demonstrate your understanding of the algorithm.

To output the nodes in reverse order, you simply reverse the algorithm by following the right pointers before outputting the node, and then following the left pointers.

Pseudocode for an in-order traversal

```
Procedure preorder(current_node)
       If current_node != null Then
              If current_node.left_pointer != null Then
                     preorder(current_node.left_pointer)
              End If
              Print(current_node.data)
              If current_node.right_pointer != null Then
                     preorder(current_node.right_pointer)
              End If
       End If
End procedure
```

Post-order traversal (depth first search) in simple structured English

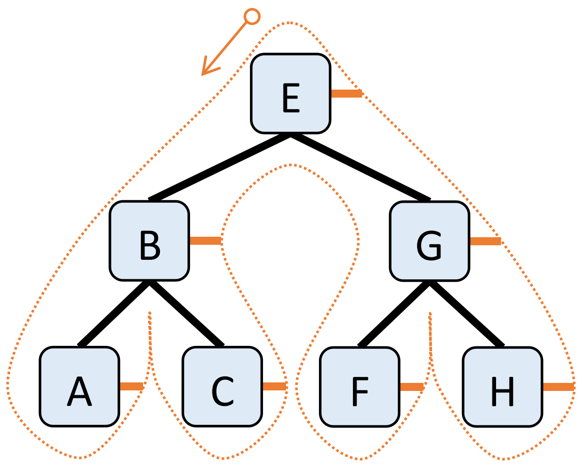
An in-order traversal, which is a variation of a depth first search is used to delete a binary tree. It is also useful to output post-fix expressions that can be used to evaluate mathematical expressions without the need for brackets. This is how arithmetic logic units work in stack machine computers and was popular in pocket calculators until the early 2010s. It is worth remembering that traversals can also be performed on graphs.

The algorithm can be described as: Right-Node-Left:

1. Start at the root node.
2. Follow the left pointer and repeat from step 2 recursively until there is no pointer to follow.
3. Follow the right pointer and repeat from step 2 recursively until there is no pointer to follow
4. Output the node.

Post-order traversal illustrated

If illustrations help you remember, this is how you can picture a post-order traversal on a binary tree:



Note the markers on the right side of each node. As you traverse the tree, starting from the root, the nodes are only output when you pass the marker: A, C, B, F, H, G, E. You can illustrate like this in exams to demonstrate your understanding of the algorithm.

Pseudocode for a post-order traversal

```
Procedure preorder(current_node)
       If current_node != null Then
              If current_node.left_pointer != null Then
                     preorder(current_node.left_pointer)
              End If
              If current_node.right_pointer != null Then
                     preorder(current_node.right_pointer)
              End If
              Print(current_node.data)
       End If
End procedure
```

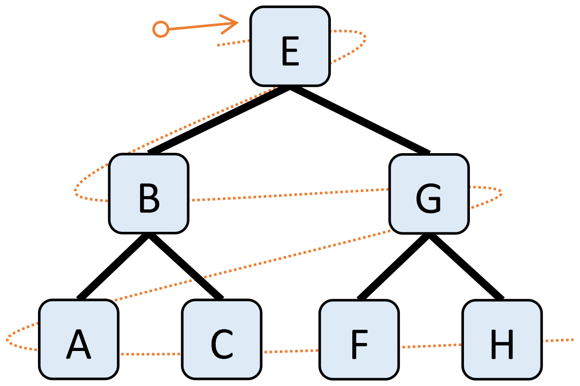
Breadth first search using a binary tree in simple structured English

Visiting each node on the same level of a tree before going deeper is an example of a breadth first search. Although commonly associated with graphs, a binary tree can also be traversed in this way, but it does require a queue structure to implement it.

1. Start at the root node.
2. Whilst the current node exists:
   a. Output the current node.
   b. If the current node has a left child, enqueue the left node.
   c. If the current node has a right child, enqueue the right node.
   d. Dequeue and set the current node to the dequeued node.

Breadth first search on a binary tree illustrated

If illustrations help you remember, this is how you can picture a post-order traversal on a binary tree:



Pseudocode for a breadth first search on a binary tree

```
current_node = root_node
While current_node != null
       Print(current_node)
       If current_node.left_pointer Then enqueue(current_node.left_pointer)
       If current_node.right_pointer Then enqueue(current_node.right_pointer)
       current_node = dequeue()
```

Binary tree coded in Python

```
class binary_tree:

    class node:
        data = None
        left_pointer = None
        right_pointer = None

    start = None

    def add(self,item):
        #Check memory overflow
        try:
            new_node = binary_tree.node()
            new_node.data = item
            current_node = self.start
            new_node.left_pointer = None
            new_node.right_pointer = None
            #Tree is empty
            if current_node == None:
                self.start = new_node
            else:
                #Find correct position in the tree
                while current_node != None:
                    previous = current_node
                    if item < current_node.data:
                        current_node = current_node.left_pointer
                    else:
                        current_node = current_node.right_pointer
                if item < previous.data:
                    previous.left_pointer = new_node
                else:
                    previous.right_pointer = new_node
                return True
        except:
            return False

    def delete(self,item):
        #Using Hibbard's algorithm (leftmost node of right sub-tree is the successor)
        #Find the node to delete
        current_node = self.start
        while current_node != None and current_node.data != item:
            previous = current_node
            if item < current_node.data:
                current_node = current_node.left_pointer
            else:
                current_node = current_node.right_pointer

        #Handle 3 cases depending on the number of child nodes
        if current_node != None:
            if current_node.left_pointer == None and current_node.right_pointer == None:
                #Node has no children
                if previous.data > current_node.data:
                    previous.left_pointer = None
                else:
                    previous.right_pointer = None
            elif current_node.right_pointer == None:
                #Node has one left child
                if previous.data > current_node.data:
                    previous.left_pointer = current_node.left_pointer
                else:
                    previous.right_pointer = current_node.left_pointer
            elif current_node.left_pointer == None:
                #Node has one right child
                if previous.data < current_node.data:
                    previous.left_pointer = current_node.right_pointer
                else:
                    previous.right_pointer = current_node.right_pointer
            else:
                #Node has two children
                right_node = current_node.right_pointer
                if right_node.left_pointer != None:
                    #Find the smallest value in the right sub-tree (successor node)
                    smallest = right_node
                    while smallest.left_pointer != None:
                        previous = smallest
                        smallest = smallest.left_pointer
                    #Change the deleted node value to the smallest value
                    current_node.data = smallest.data
                    #Remove the successor node
                    previous.left_pointer = None
                else:
                    #Handle special case of no left sub-tree from right node
                    current_node.data = right_node.data
                    current_node.right_pointer = None


    def preorder(self,current_node):
        if current_node != None:
            #Visit each node: NLR
            print(current_node.data)
            if current_node.left_pointer != None:
                self.preorder(current_node.left_pointer)
            if current_node.right_pointer != None:
                self.preorder(current_node.right_pointer)

    def inorder(self,current_node):
            if current_node != None:
                #Visit each node: LNR
                if current_node.left_pointer != None:
                    self.inorder(current_node.left_pointer)
                print(current_node.data)
                if current_node.right_pointer != None:
                    self.inorder(current_node.right_pointer)

    def postorder(self,current_node):
            if current_node != None:
                #Visit each node: LRN
                if current_node.left_pointer != None:
                    self.postorder(current_node.left_pointer)
                if current_node.right_pointer != None:
                    self.postorder(current_node.right_pointer)
                print(current_node.data)
```

Adding items to a binary tree object in Python

```
items = ["E","B","G","A","C","F","H"]
bt = binary_tree()
for index in range(0,len(items)):
    bt.add(items[index])
```

Deleting items from a binary tree object in Python

```
bt.delete("G")
```

Outputting items from a binary tree object in Python

```
bt.preorder(bt.start)

bt.inorder(bt.start)
bt.postorder(bt.start)
```

Binary tree coded in Visual Basic Console

```
Module Module1
    Public Class binarytree

        Public Class node
            Public data As String
            Public left_pointer As node
            Public right_pointer As node
        End Class

        Public start As node

        Function add(item As String)
            'Check memory overflow
            Try
                Dim new_node As New node
                new_node.data = item
                Dim current_node As node = start
                new_node.left_pointer = Nothing
                new_node.right_pointer = Nothing
                'Tree is empty
                If IsNothing(current_node) Then
                    start = new_node
                Else
                    'Find correct position in the tree
                    Dim previous As node
                    While Not IsNothing(current_node)
                        previous = current_node
                        If item < current_node.data Then
                            current_node = current_node.left_pointer
                        Else
                            current_node = current_node.right_pointer
                        End If
                    End While
                    If item < previous.data Then
                        previous.left_pointer = new_node
                    Else
                        previous.right_pointer = new_node
                    End If
                    Return True
                End If
            Catch
                Return False
            End Try
        End Function

        Sub delete(item As String)
            'Using Hibbard's algorithm (leftmost node of right sub-tree is the successor)
            'Find the node
            Dim current_node As node = start
            Dim previous As node
            While Not IsNothing(current_node) And current_node.data <> item
                previous = current_node
                If item < current_node.data Then
                    current_node = current_node.left_pointer
                Else
                    current_node = current_node.right_pointer
                End If
            End While

            'Handle 3 cases depending on the number of child nodes
            If Not IsNothing(current_node) Then
                If IsNothing(current_node.left_pointer) And IsNothing(current_node.right_pointer)
Then
                    'Node has no children
                    If previous.data > current_node.data Then
                        previous.left_pointer = Nothing
                    Else
                        previous.right_pointer = Nothing
                    End If
                ElseIf IsNothing(current_node.right_pointer)
                    'Node has one left child
                    If previous.data > current_node.data Then
                        previous.left_pointer = current_node.left_pointer
                    Else
                        previous.right_pointer = current_node.left_pointer
                    End If
                ElseIf IsNothing(current_node.left_pointer)
                    'Node has one right child
                    If previous.data < current_node.data Then
                        previous.left_pointer = current_node.right_pointer
                    Else
                        previous.right_pointer = current_node.right_pointer
                    End If
                Else
                    'Node has two children
                    Dim right_node As node = current_node.right_pointer
                    If Not IsNothing(right_node.left_pointer) Then
                        'Find the smallest value in the right sub-tree
                        Dim smallest As node = current_node.right_pointer
                        While Not IsNothing(smallest.left_pointer)
                            previous = smallest
                            smallest = smallest.left_pointer
                        End While
                        'Change the deleted node value to the smallest value
                        current_node.data = smallest.data
                        'Remove the successor node
                        previous.left_pointer = Nothing
                Else
                    'Handle special case of no left sub-tree from right node
                    current_node.data = right_node.data
                    current_node.right_pointer = Nothing
                End If
            End If
        End If
    End Sub

    Sub preorder(current_node As node)
        If Not IsNothing(current_node) Then
            'Visit each node: NLR
            Console.WriteLine(current_node.data)
            If Not IsNothing(current_node.left_pointer) Then
                preorder(current_node.left_pointer)
            End If
            If Not IsNothing(current_node.right_pointer) Then
                preorder(current_node.right_pointer)
            End If
        End If
    End Sub

    Sub inorder(current_node As node)
        If Not IsNothing(current_node) Then
            'Visit each node: LNR
            If Not IsNothing(current_node.left_pointer) Then
                inorder(current_node.left_pointer)
            End If
            Console.WriteLine(current_node.data)
            If Not IsNothing(current_node.right_pointer) Then
                inorder(current_node.right_pointer)
            End If
        End If
    End Sub

    Sub postorder(current_node As node)
        If Not IsNothing(current_node) Then
            'Visit each node: LRN
            If Not IsNothing(current_node.left_pointer) Then
                postorder(current_node.left_pointer)
            End If
            If Not IsNothing(current_node.right_pointer) Then
                postorder(current_node.right_pointer)
            End If
            Console.WriteLine(current_node.data)
        End If
    End Sub

End Class
```

```
    'Main program starts here
    Sub Main()

        bt.delete("Delaware")
        bt.inorder(bt.start)
        Console.ReadLine()
    End Sub
End Module
```

Adding items to a binary tree object in Console Visual Basic

```
Dim items() As String = {"E", "B", "G", "A", "C", "F", "H"}
        Dim bt As New binarytree
        For index = 0 To items.Length - 1
            bt.add(items(index))
        Next
```

Deleting items from a binary tree object in Console Visual Basic

```
bt.delete("G")
```

Outputting items from a binary tree object in Console Visual Basic

```
bt.preorder(bt.start)
bt.inorder(bt.start)
bt.postorder(bt.start)
```

Did you know?

A tree traversal is also known as a tree search. Whilst a linked list can only be traversed forwards or backwards, a tree can be traversed in many different ways.

Efficiency of a binary tree

Space complexity of a binary tree

Object implementation	Array implementation
O(n) Linear	O(1) Constant

A binary tree is a dynamic data structure. This means its memory footprint grows and shrinks as data is added and deleted from the structure when implemented using object-oriented techniques. Using only as much memory as is needed is the most efficient way of implementing a binary tree. Alternatively, you could implement a binary tree using a dictionary or an array. With an array you would be using a static data structure and therefore the memory footprint would be static, but the size of the structure would also be limited.

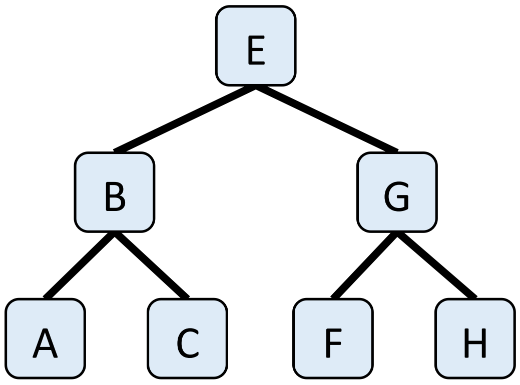
Time complexity of operations on a binary tree

Operation	Best case	Average case	Worst case
Storing the binary tree	O(n) Linear	O(n) Linear	O(n) Linear
Add a node	O(log n) Logarithmic	O(log n) Logarithmic	O(n) Linear
Delete a node	O(log n) Logarithmic	O(log n) Logarithmic	O(n) Linear
Binary search	O(1) Constant	O(log n) Logarithmic	O(V) Linear
Traversal	O(V) Linear	O(V) Linear	O(V) Linear

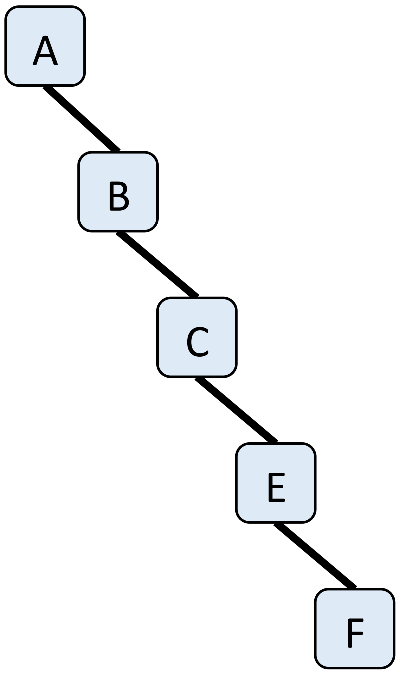
Assuming the binary tree is not pre-populated, to establish the tree from a list of data items will require each item to be added sequentially, although the input order of the items does not matter, therefore the number of operations depends on the number of data items, O(n).

Adding, deleting and searching nodes on a binary tree uses a technique called **divide and conquer** where, with a balanced tree, the number of items to consider is halved each time a node is visited, providing logarithmic complexity, O(log n). If the tree is unbalanced, it becomes the same as a linear search to determine the location of an item to add or delete. Therefore, degrading to linear complexity, O(n).

Balanced tree where each level is complete:



Unbalanced tree holding the same data:



In the best case scenario, the node to be found is the root node, and therefore it can always be found first, O(1).

The traversal algorithms all require each node to be visited, and therefore as the number of nodes in the tree increases, so too does the execution time. Therefore, traversals are linear complexity, O(n).