CSC 225 - Summer 2019

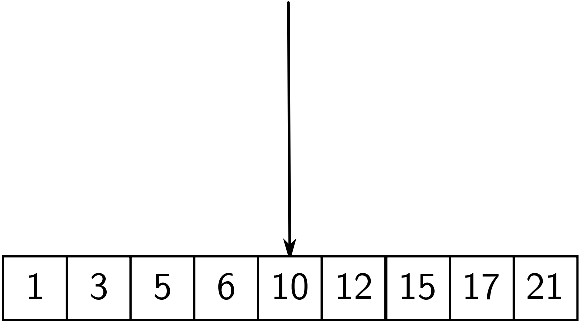
Trees

Bill Bird

Department of Computer Science
University of Victoria

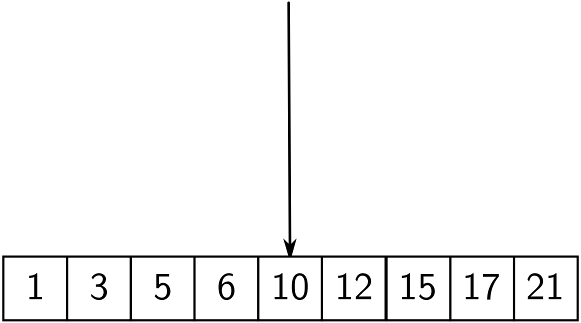
June 12, 2019

Binary Search (1)



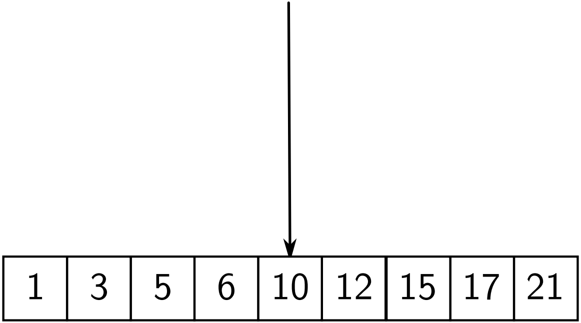
Problem: Describe a data structure which contains a collection of elements and has fast FIND and INSERT operations.

Binary Search (2)



A sorted array is a good choice to optimize the FIND operation, since binary search requires $\Theta(\log n)$ time.

Binary Search (3)



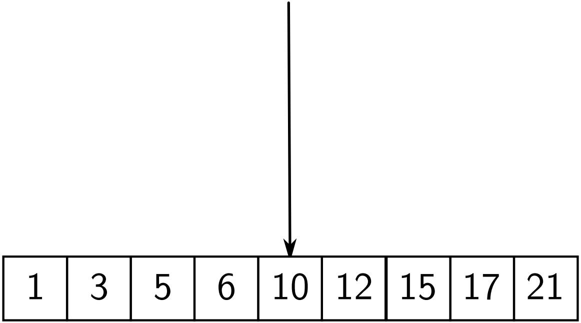
Aside: $\Theta(\log n)$ is optimal for comparison-based searching (proven in CSC 226).

Binary Search (4)

1	3	5	6	10	12	15	17	21

However, insertion into an array requires $\Theta(n)$ time in the worst case.

Binary Search (5)



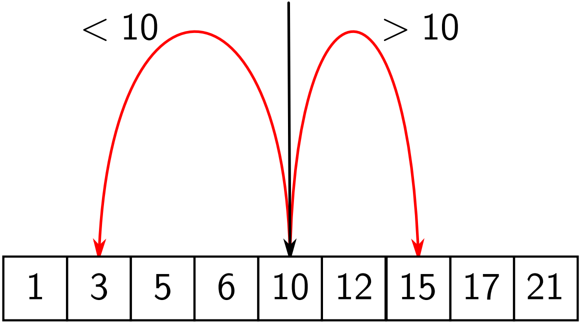
Linked lists support $\Theta(1)$ insertion, but fast binary search is not possible in a linked list.

Binary Search (6)

1	3	5	6	10	12	15	17	21

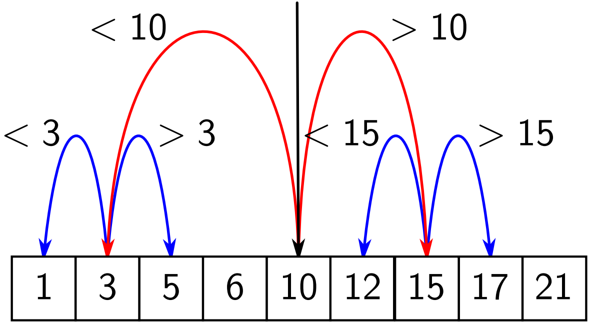
Consider the sequence of values examined by binary search for a value k in the above array.

Binary Search (7)



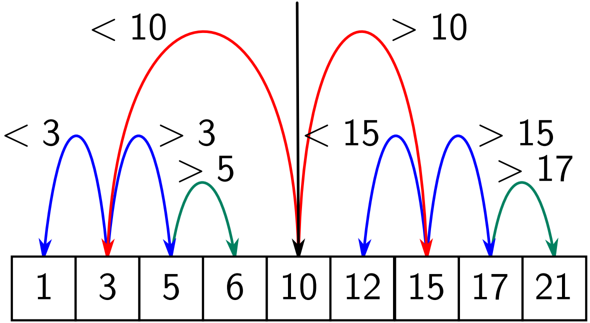
Depending on the value of k, the search follows a different path through the array, but all of the possible paths are essentially symmetric.

Binary Search (8)



Depending on the value of k, the search follows a different path through the array, but all of the possible paths are essentially symmetric.

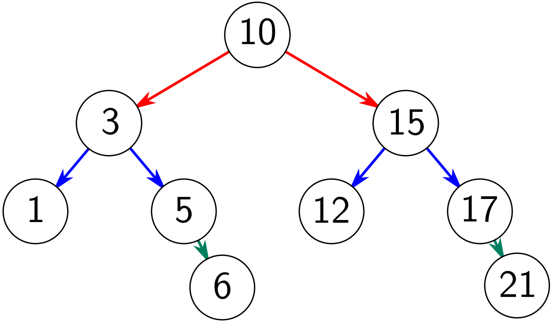
Binary Search (9)



Idea: Represent the set of possible paths of the binary search algorithm with a linked structure.

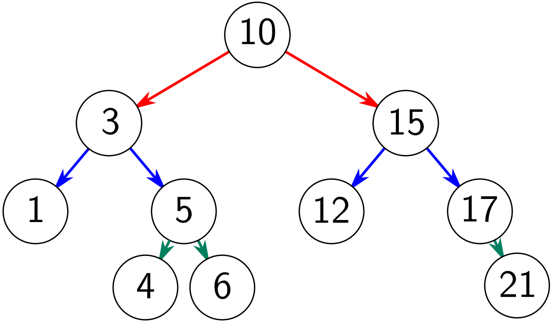
Intuitively, this should preserve the fast running time of binary search and allow fast insertion.

Binary Search (10)



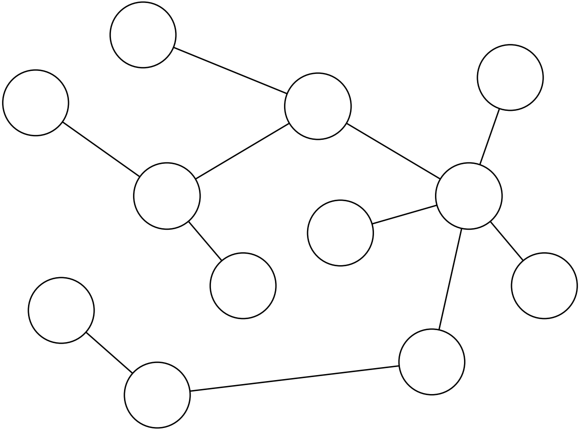
The result is a **binary search tree**.

Binary Search (11)



Inserting an element into a binary search tree is $\Theta(1)$ after the correct location has been found.

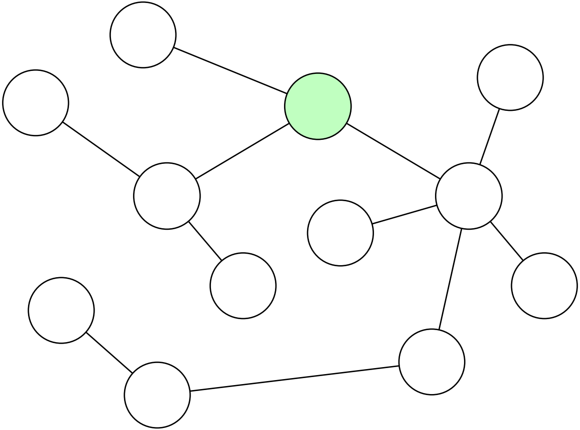
Trees (1)



A **tree** is a **connected, acyclic graph**.

This definition will not be useful until we cover graphs (in July).

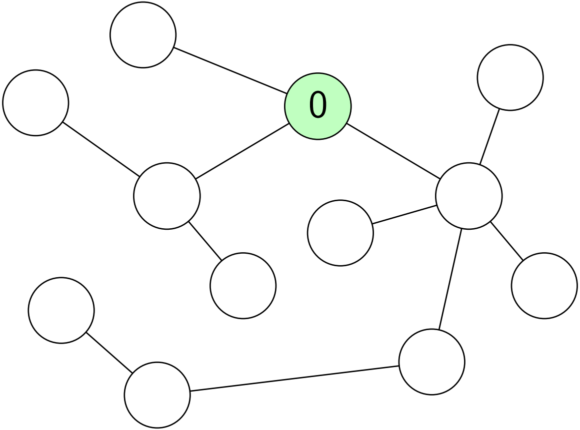
Trees (2)



For now, we will restrict ourselves to rooted trees, which are the most common type encountered in computer science.

A **rooted tree** has a distinguished root node (shaded).

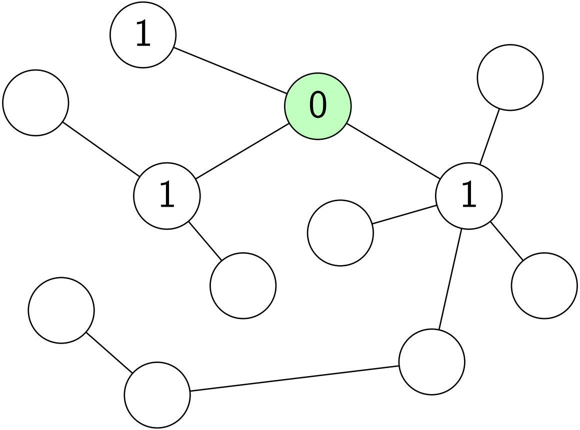
Trees (3)



Since the root is a special node, the every other node can be classified by its distance to the root.

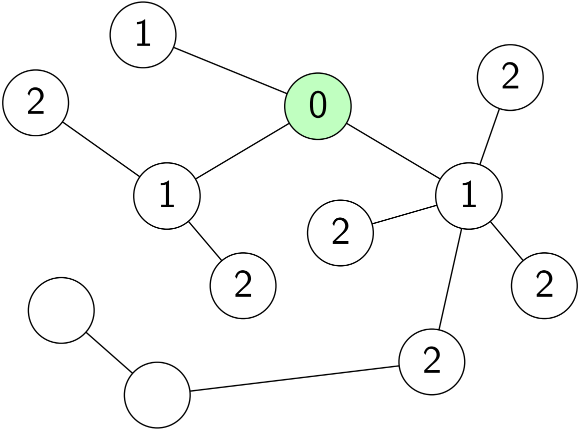
The root is at distance 0.

Trees (4)



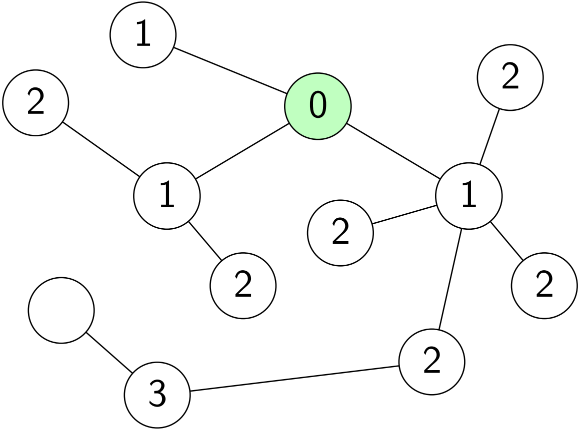
The neighbours of the root are at distance 1.

Trees (5)



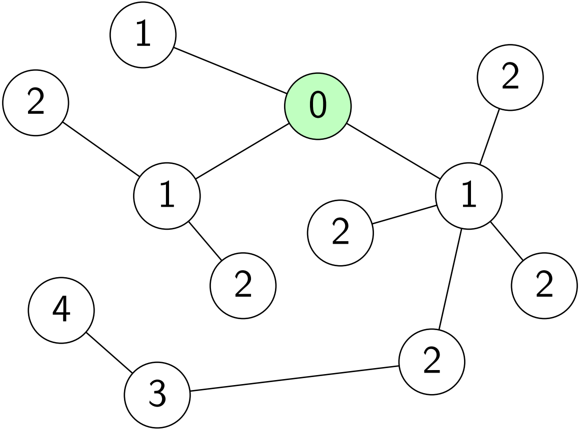
The neighbours of those nodes have distance 2.

Trees (6)



The distance of a node from the root is also called its **depth**.

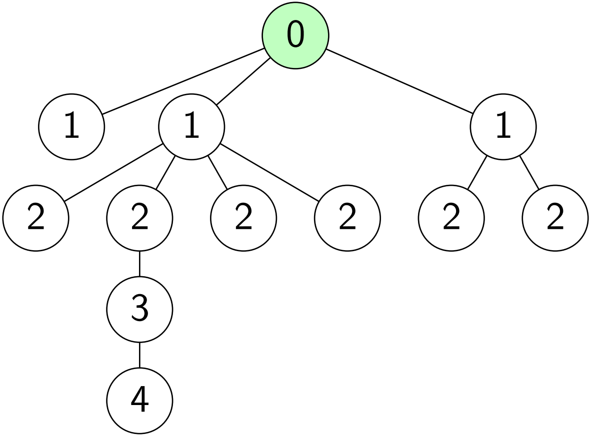
Trees (7)



The maximum depth of a node in the tree is called the **height** of the tree.

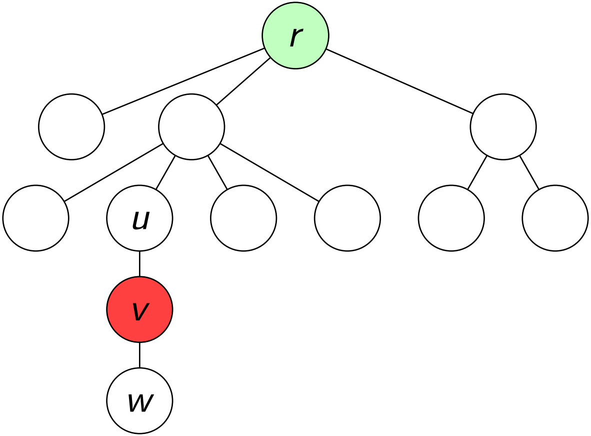
The tree above has height 4.

Trees (8)



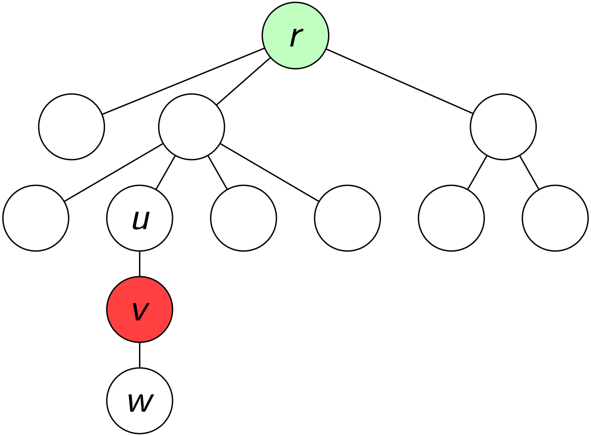
Since the nodes can be classified by distance, rooted trees are usually drawn with a hierarchical diagram like the one above.

Trees (9)



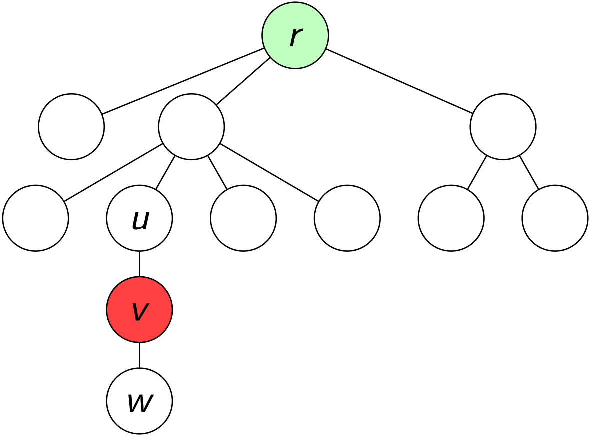
- Every node besides the root has a **parent**, which is the unique neighbouring node which is closer to the root.
- The parent of node v is node u.
- The parent of node w is node v.

Trees (10)



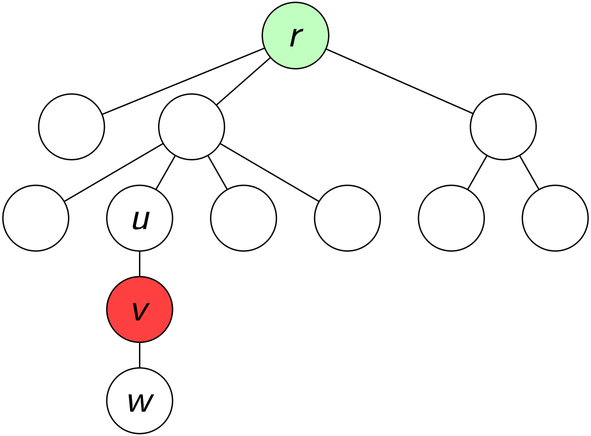
- Since node u is the parent of node v, the node v is a **child** of u.
- Similarly, w is a child of v.

Trees (11)



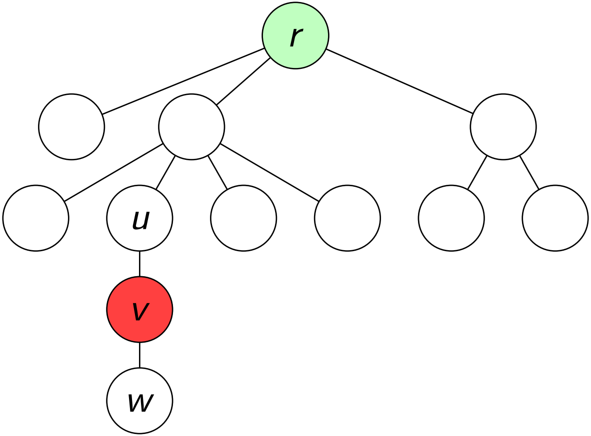
- The parent/child relationship can be extended to multiple generations.
- u is a **grandparent** of w.
- w is a **grandchild** of u.

Trees (12)



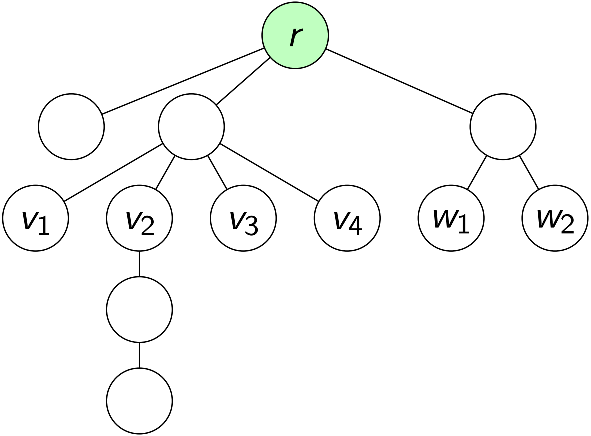
- In general, a node q is an **ancestor** of all the nodes accessible through the children of q.
- In the diagram above, u is an ancestor of v and w.

Trees (13)



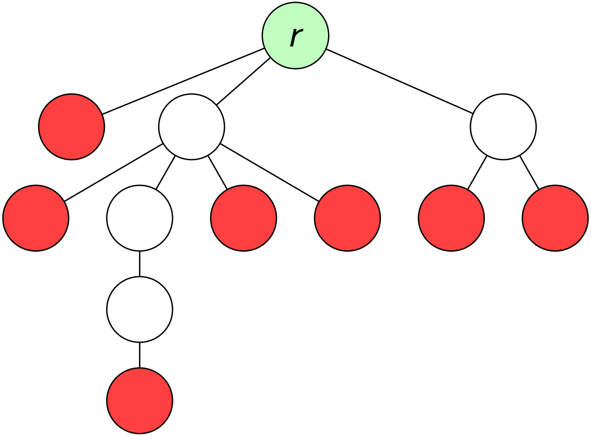
- A node q is a **descendant** of each of its ancestors.
- The node w in the diagram is a descendant of u and v.

Trees (14)



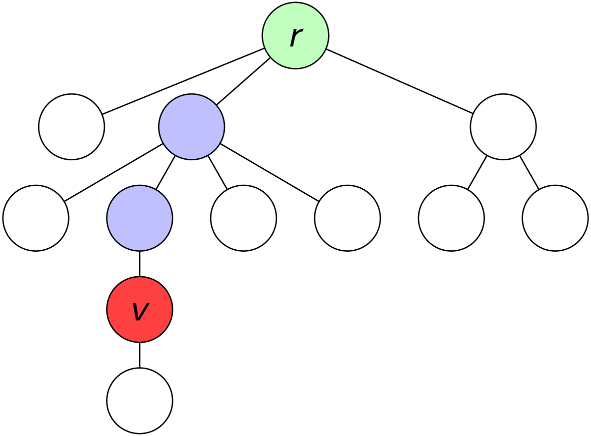
- Nodes which are children of the same parent are **siblings**.
- v_1, v_2, v_3 and v_4 are siblings.
- w_1 and w_2 are siblings.

Trees (15)



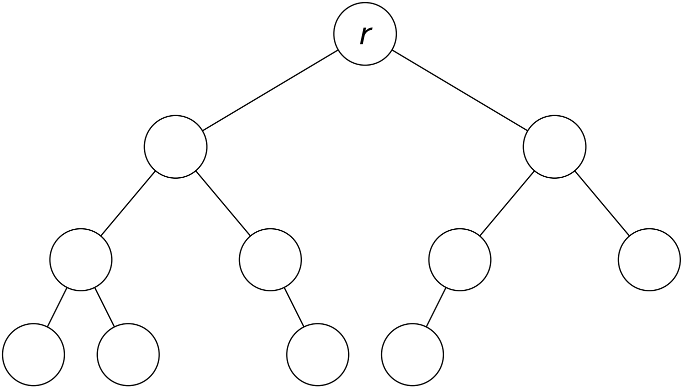
- **Leaves** of a rooted tree are nodes with no children.
- Nodes which are not leaves are called **internal nodes**

Trees (16)



For any node in a rooted tree, there is a **unique** path from that node to the root, obtained by walking up from the node through its successive ancestors until reaching the root.

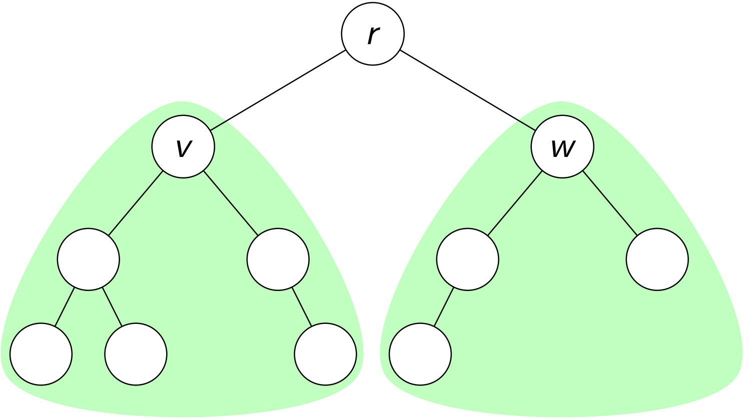
Binary Trees (1)



A **binary tree** is a tree in which each node has 0, 1 or 2 children.

Note that this definition implies that binary trees are always rooted.

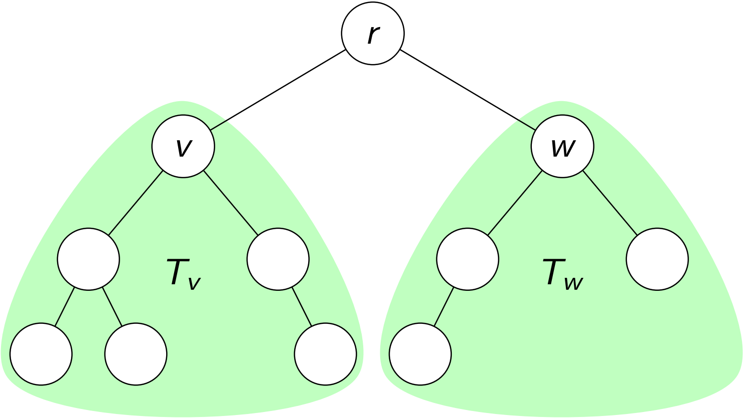
Binary Trees (2)



Every node in a binary tree can be considered to be the root of a smaller binary tree.

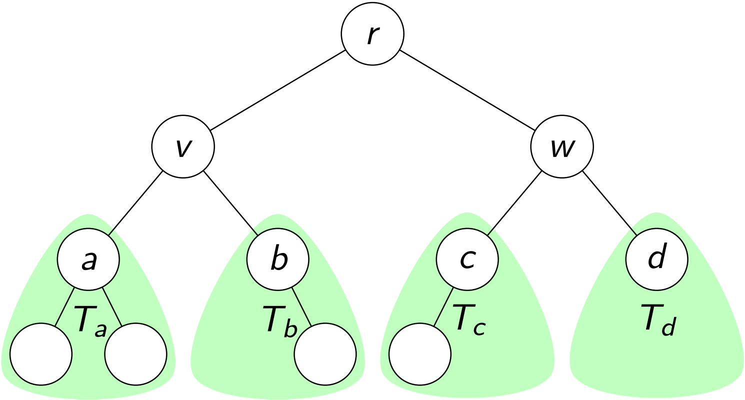
A **subtree** is the binary tree rooted at a particular node.

Binary Trees (3)



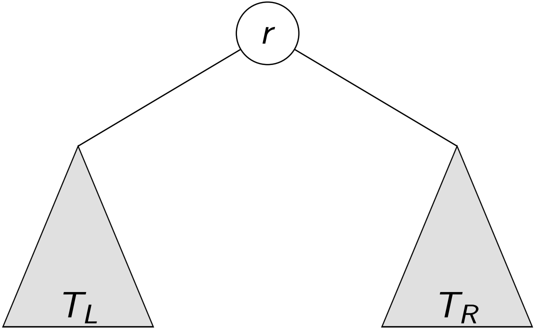
Normally, the subtree of T rooted at a node v is denoted T_v.

Binary Trees (4)



Subtrees can be rooted at any node (even a node without children).

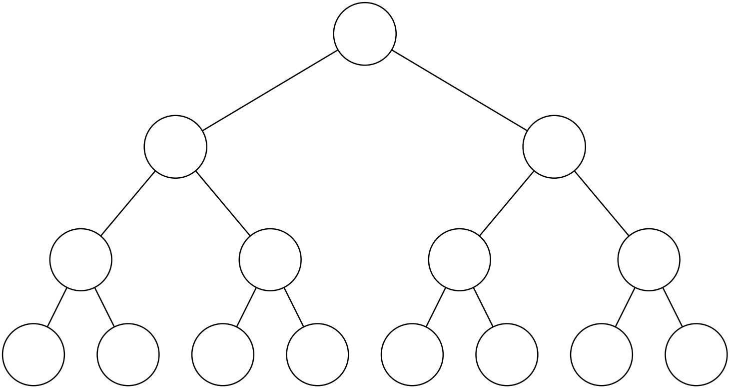
Binary Trees (5)



Often, a node's children are drawn as triangles to denote their respective subtrees without explicitly drawing them.

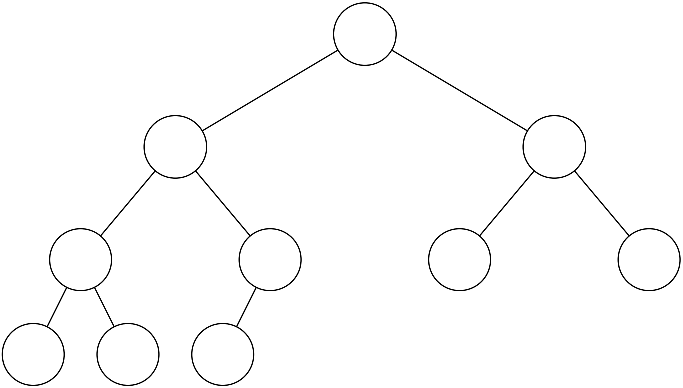
In the event that a node has no left (or right) child, the corresponding subtree is an empty tree.

Full Binary Trees



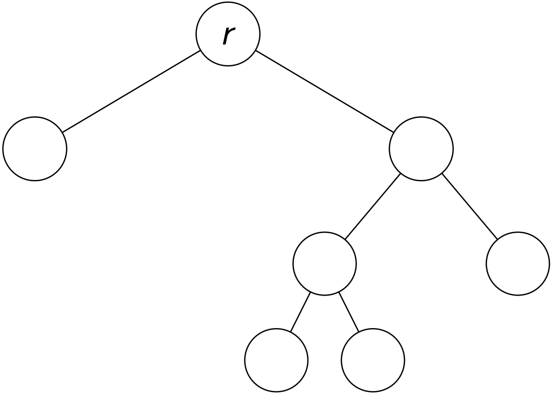
A **full binary tree** is a binary tree in which every level contains the maximum number of nodes (or is empty).

Complete Binary Trees



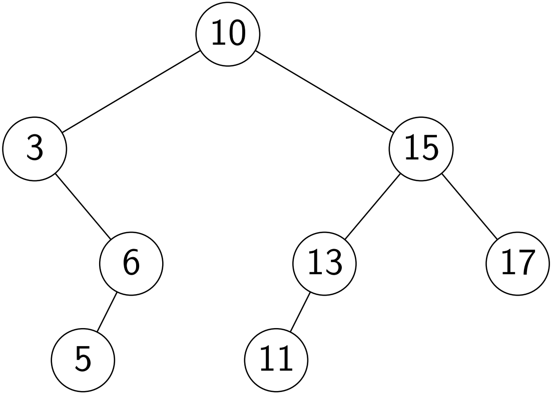
A **complete binary tree** is a binary tree in which every level but the last level is full, and the last level is filled in from left-to-right.

Proper Binary Trees (1)



A **proper binary tree** is a tree in which each node has 0 or 2 children.

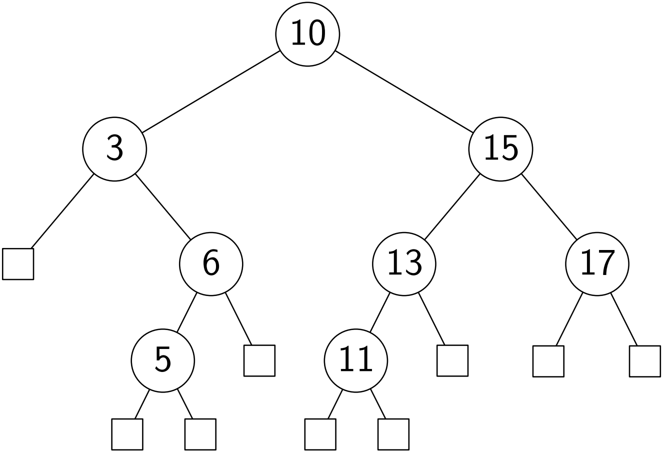
Proper Binary Trees (2)



The tree above is not a proper binary tree.

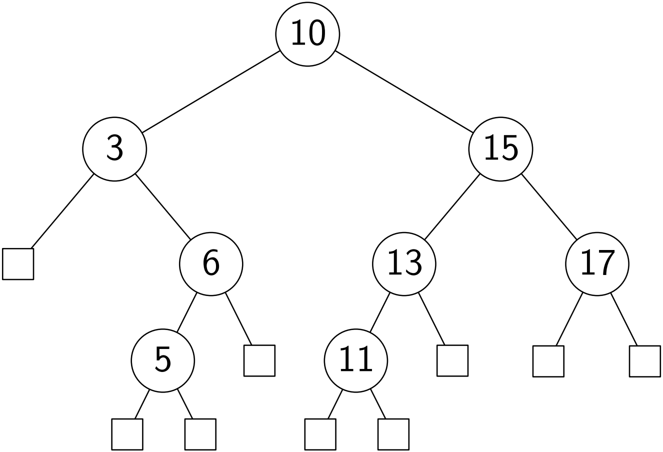
However, there is a correspondence between improper and proper binary trees.

Proper Binary Trees (3)



To convert an improper binary tree to a proper binary tree, attach vestigial leaf nodes all nodes with fewer than two children.

Proper Binary Trees (4)



The added leaf nodes can be thought of as a representation of `null` references in a tree implementation.

Proper Binary Trees (5)

A proper binary tree can be defined with an **inductive definition**:

- A proper binary tree of height 0 is a single node.
- A proper binary tree of height 1 or greater is a node with two children, each of which is the root of a proper binary tree.

Proper Binary Trees (6)

Exercises:

1. Prove that every non-empty proper binary tree with n internal nodes has exactly $n + 1$ leaves.
2. Prove that a proper binary tree with height h has at most 2^h leaves.

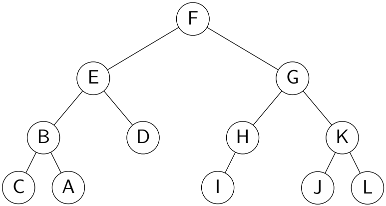
Recursion and Binary Trees

```
procedure CountNodes(v)
    if v = null then
        return 0
    end if
    return 1+ CountNodes(v.left) + CountNodes(v.right)
end procedure
```

- Similar to linked lists, binary trees are naturally suited to recursive algorithms.
- The algorithm CountNodes above returns the total number of nodes in the subtree rooted at the provided node (including the node itself).
- CountNodes is an example of a general class of algorithms called **traversals**.

Binary Tree Traversals (1)

```
procedure PreOrder(v)
    if v = null then
        return
    end if
    Visit v
    PreOrder(v.left)
    PreOrder(v.right)
end procedure
```

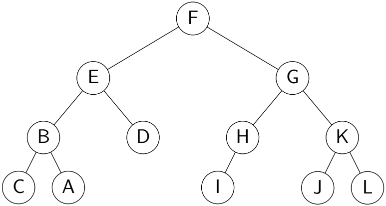


Pre-Order Traversal:
F, E, B, C, A, D, G, H, I, K, J, L

A **traversal** of a binary tree visits every node in the tree in a specific order.

Binary Tree Traversals (2)

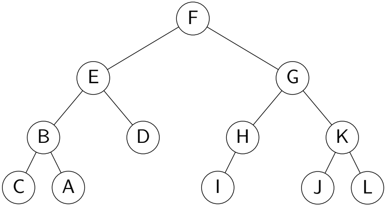
```
procedure PreOrder(v)
    if v = null then
        return
    end if
    Visit v
    PreOrder(v.left)
    PreOrder(v.right)
end procedure
```



Pre-Order Traversal:
F, E, B, C, A, D, G, H, I, K, J, L

In a **pre-order traversal**, each node v is visited before recursively traversing both of its subtrees.

```
procedure InOrder(v)
    if v = null then
        return
    end if
    InOrder(v.left)
    Visit v
    InOrder(v.right)
end procedure
```

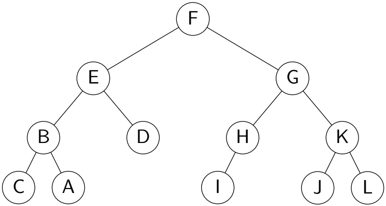


In-Order Traversal:
C, B, A, E, D, F, I, H, G, J, K, L

In an **in-order traversal**, the node v is visited after recursively traversing the left subtree and before recursively traversing the right subtree.

Binary Tree Traversals (4)

```
procedure PostOrder(v)
    if v = null then
        return
    end if
    PostOrder(v.left)
    PostOrder(v.right)
    Visit v
end procedure
```



Post-Order Traversal:
C, A, B, D, E, I, H, J, L, K, G, F

In a **post-order traversal**, each node v is visited after recursively traversing both of its subtrees.

Binary Tree Traversals (5)

```
procedure PreOrder(v)
    if v = null then
        return
    end if
    Visit v
    PreOrder(v.left)
    PreOrder(v.right)
end procedure
```

```
procedure PreOrderIterative(root)
    if root = null then
        return
    end if
    S ← Empty stack
    Push root onto S
    while S is non-empty do
        v ← Pop(S)
        Visit v
        if v.right ≠ null then
            Push v.right onto S
        end if
        if v.left ≠ null then
            Push v.left onto S
        end if
    end while
end procedure
```

Traversals can also be implemented iteratively with a stack.

Stack-based traversals (including pre-order, in-order and post-order) are **depth first** traversals.

Binary Tree Traversals (6)

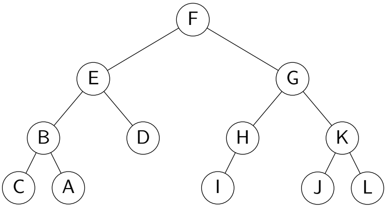
```
procedure PreOrder(v)
    if v = null then
        return
    end if
    Visit v
    PreOrder(v.left)
    PreOrder(v.right)
end procedure
```

```
procedure PreOrderIterative(root)
    if root = null then
        return
    end if
    S ← Empty stack
    Push root onto S
    while S is non-empty do
        v ← Pop(S)
        Visit v
        if v.right ≠ null then
            Push v.right onto S
        end if
        if v.left ≠ null then
            Push v.left onto S
        end if
    end while
end procedure
```

Question: What if the stack is replaced by a queue?

Binary Tree Traversals (7)

```
procedure LevelOrder(root)
    if root = null then
        return
    end if
    Q ← Empty queue
    Enqueue root in Q
    while Q is non-empty do
        v ← Dequeue(S)
        Visit v
        if v.left ≠ null then
            Enqueue v.left in Q
        end if
        if v.right ≠ null then
            Enqueue v.right in Q
        end if
    end while
end procedure
```



Level-Order Traversal:
F, E, G, B, D, H, K, C, A, I, J, L

The result is a **level order traversal**, which visits the tree from the top down.

Queue-based traversals are **breadth first** traversals.

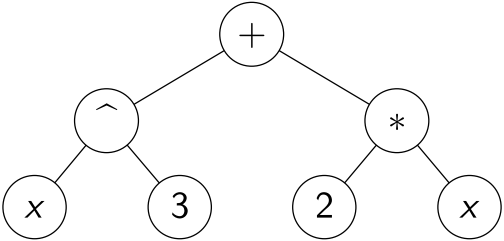
Binary Tree Traversals (8)

Theorem: All four of the preceding binary tree traversals require

$$\Theta(n)$$

operations to traverse a tree T containing n nodes.

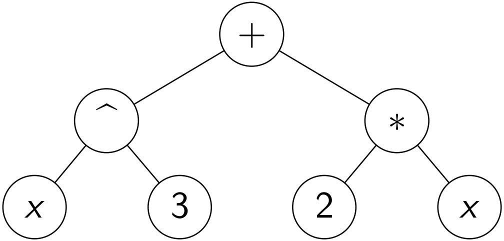
Example: Expression Trees (1)



$$f(x) = x^3 + 2x$$

An **expression tree** or **parse tree** can be used to represent an in-fix arithmetic expression.

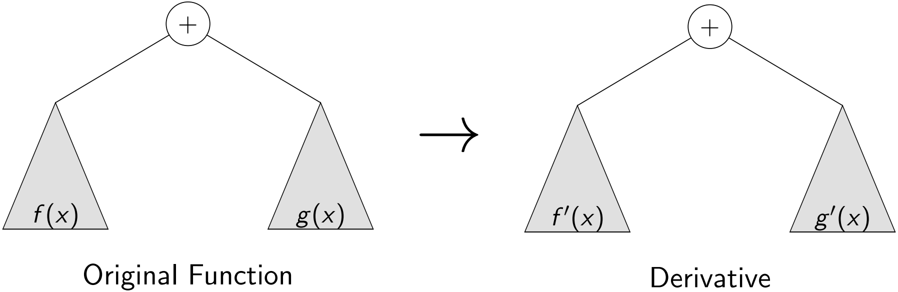
Example: Expression Trees (2)



$$f(x) = x^3 + 2x$$

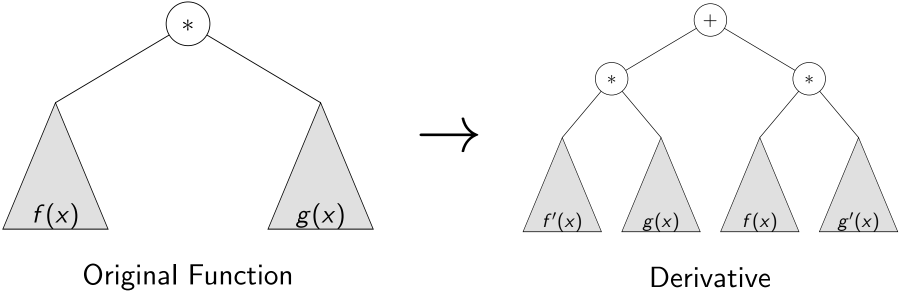
A recursive traversal algorithm can be used to evaluate the function on a given input x.

Example: Expression Trees (3)



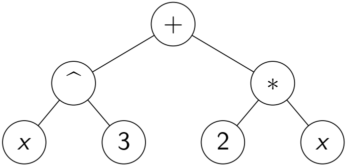
Given an expression tree for a polynomial $f(x)$, a new tree corresponding to the derivative $f'(x)$ can be constructed with simple recursive manipulations.

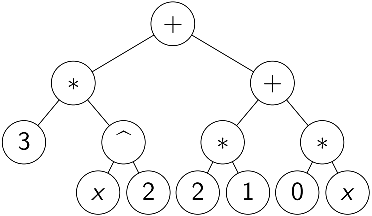
Example: Expression Trees (4)



Given an expression tree for a polynomial $f(x)$, a new tree corresponding to the derivative $f'(x)$ can be constructed with simple recursive manipulations.

Example: Expression Trees (5)





Original Function

Derivative

Given an expression tree for a polynomial $f(x)$, a new tree corresponding to the derivative $f'(x)$ can be constructed with simple recursive manipulations.