# Trees

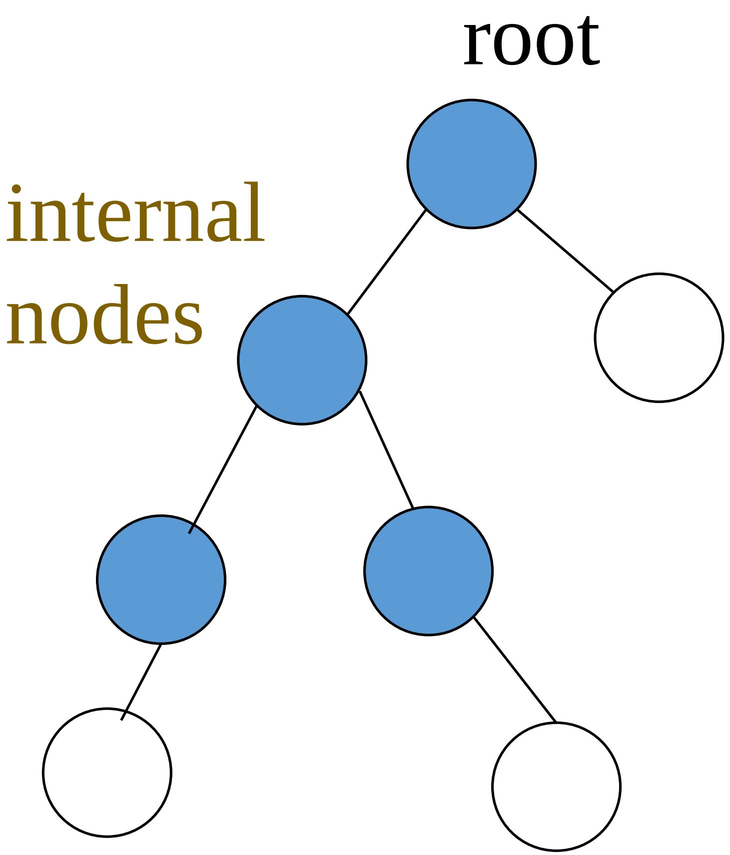
黃世強 (Sai-Keung Wong)

National Yang Ming Chiao Tung University

Taiwan

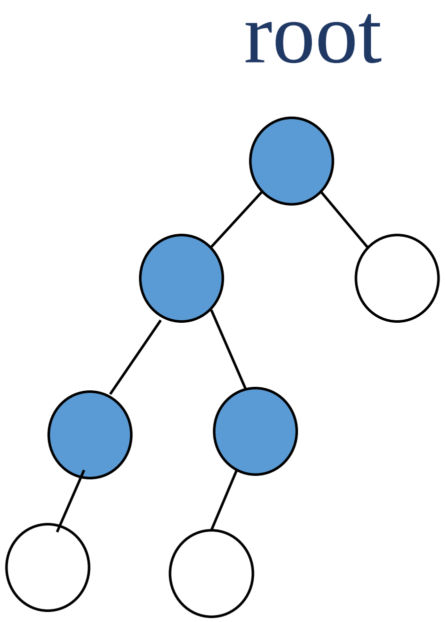
# Tree Structure

- A hierarchical data structure
- One root
- Internal nodes
- External nodes, aka, leaves
- An edge connects two nodes
- No cycle



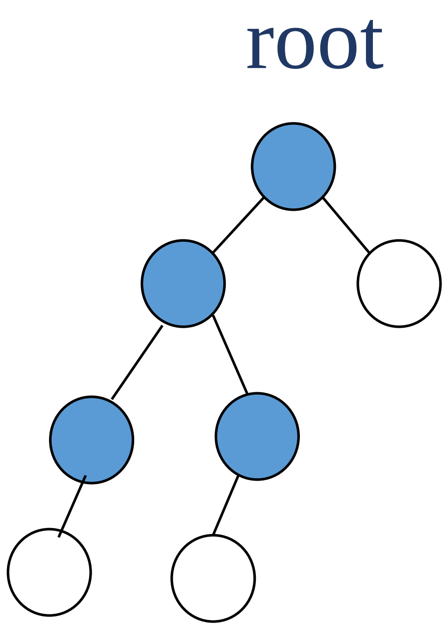
# Tree Structure

- The node at the top of the hierarchy is the root.
- Nodes next in the hierarchy are the children of the root.
- Nodes next to the children of the root are the grandchildren of the root, and so on.
- Nodes that have no children are leaves.



# Tree Definition

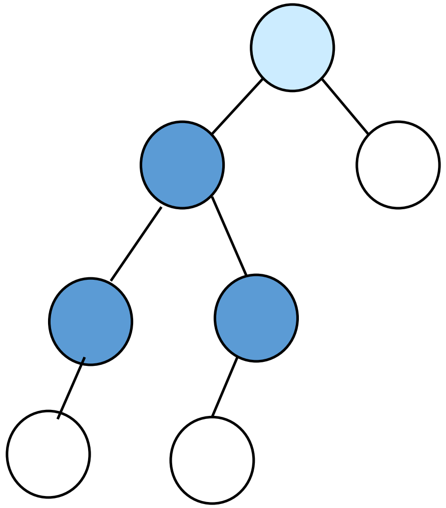
- A tree T is a finite **nonempty** set of nodes.
- One of these nodes is called the root.
- If the root is taken out, the tree is partitioned into trees (if there is/are nodes). The trees are called the subtrees of T.

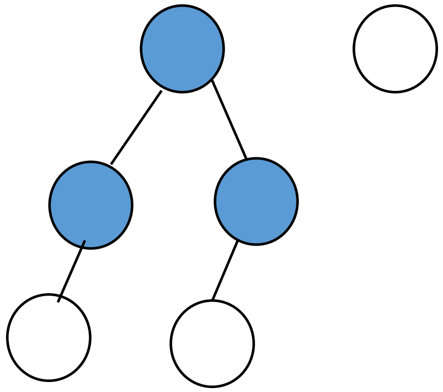


# Tree Definition

- A tree T is a finite **nonempty** set of nodes.
- One of these nodes is called the root.
- If the root is taken out, the tree is partitioned into trees (if there is/are nodes). The trees are called the subtrees of T.

root

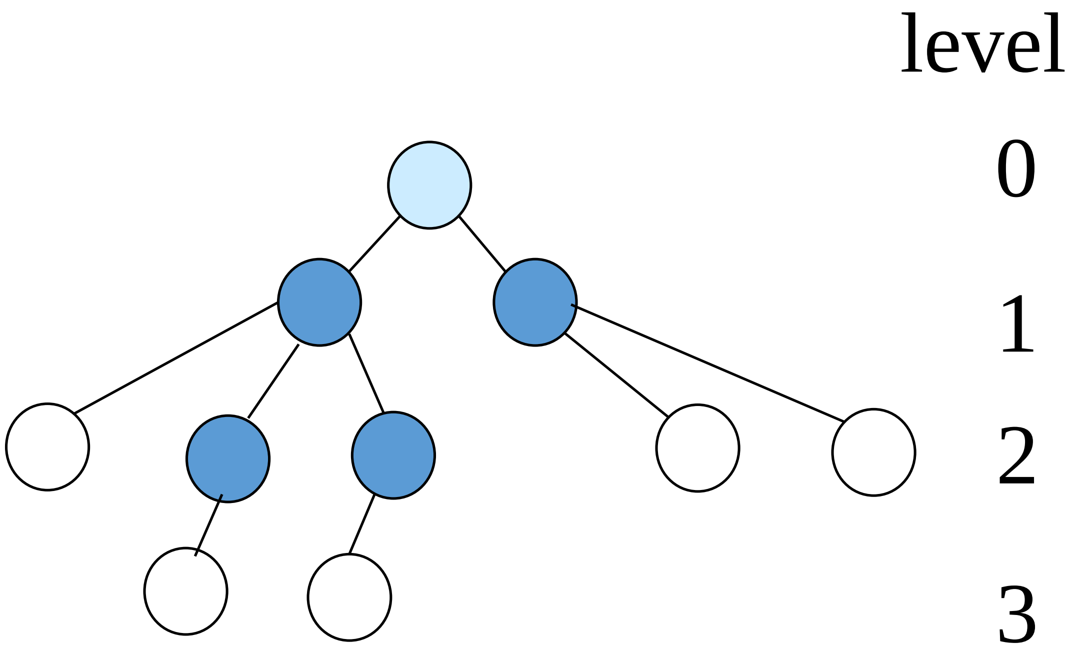




# Terminologies

- Level: The root node is level 0. When we traverse down the hierarchy by one step, the level increases by 1.

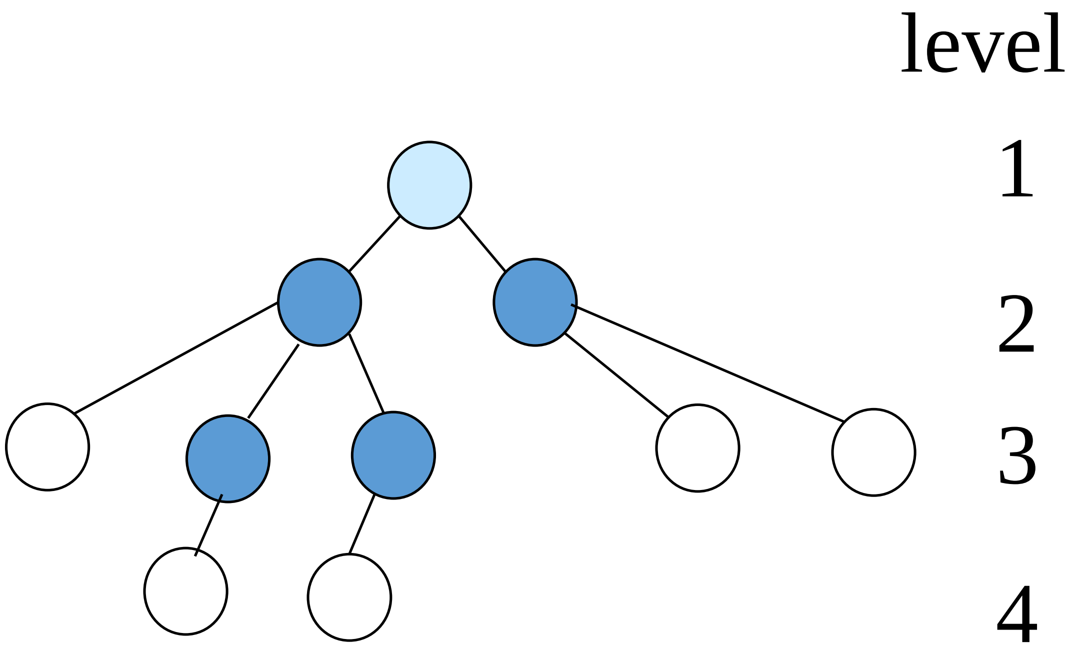
- Parent
- Grandparent
- Siblings
- Ancestors
- Descendants



# Terminologies

- Level: The root node is level 1. When we traverse down the hierarchy by one step, the level increases by 1.

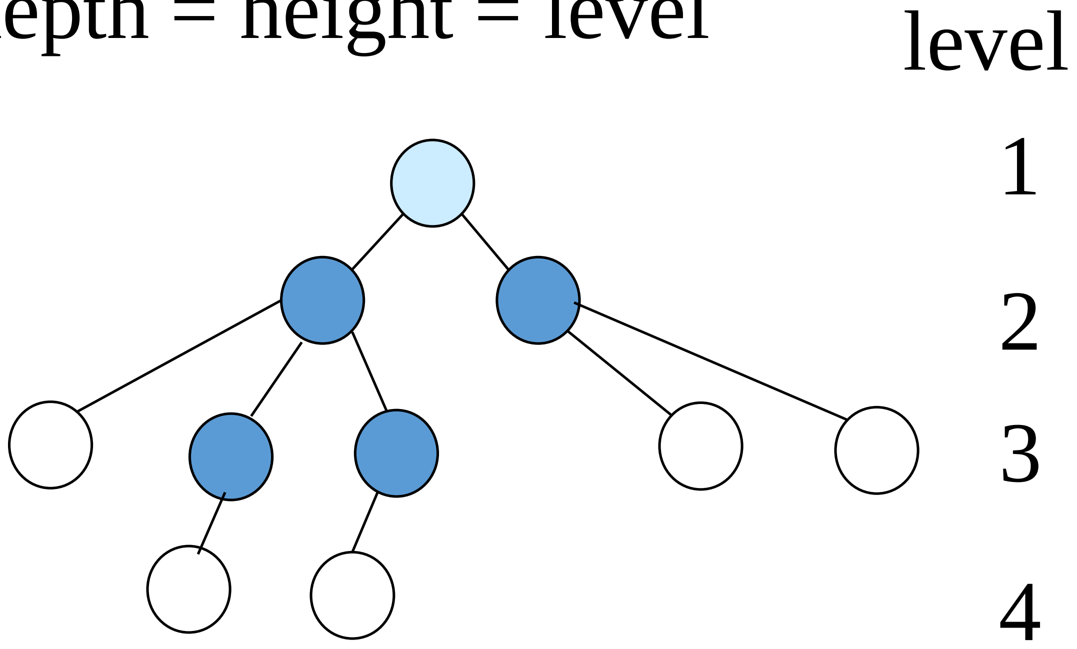
- Parent
- Grandparent
- Siblings
- Ancestors
- Descendants



# Terminologies

- Level: The root node is level 1. When we traverse down the hierarchy by one step, the level increases by 1.

depth = height = level



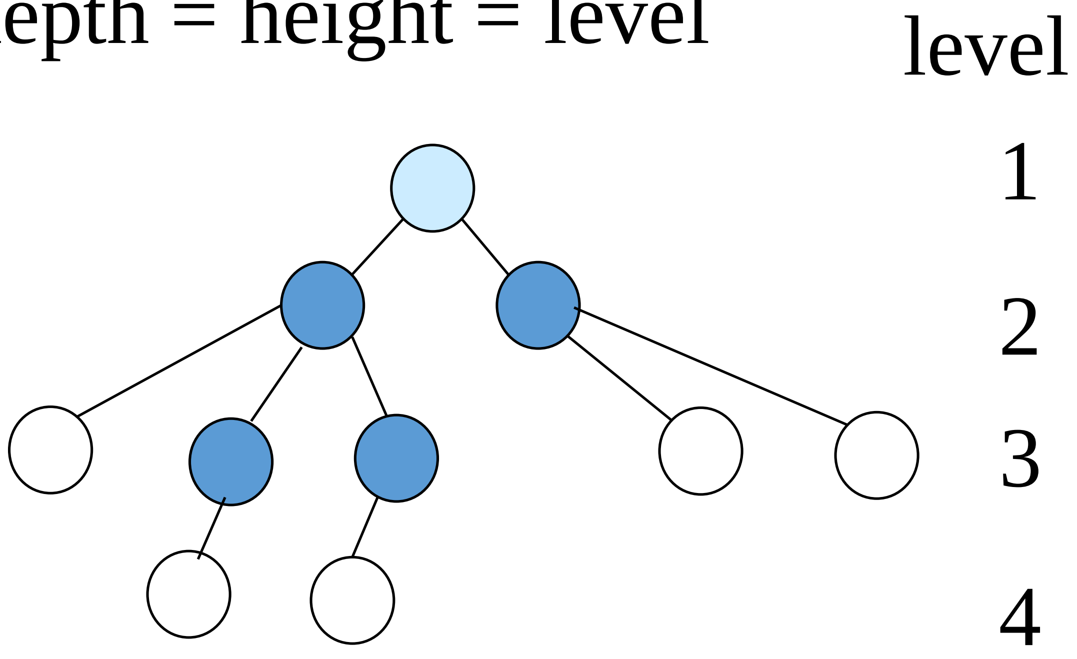

- Parent
- Grandparent
- Siblings
- Ancestors
- Descendants

# Terminologies

- Level: The root node is level 1. When we traverse down the hierarchy by one step, the level increases by 1.

depth = height = level

- Parent
- Grandparent
- Siblings
- Ancestors
- Descendants
- Node degree = number of children

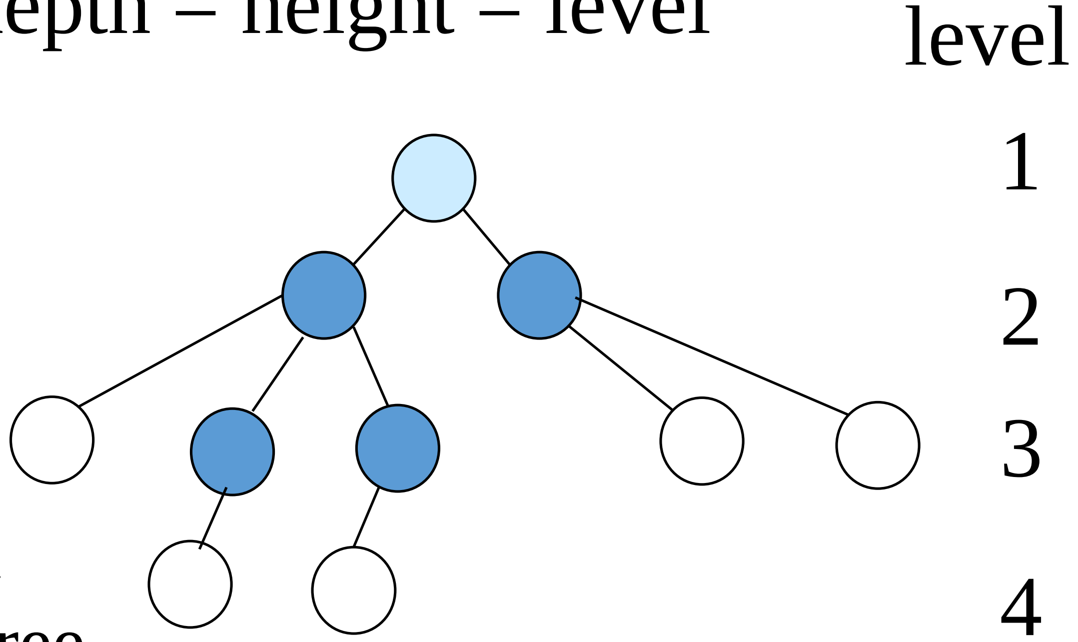


# Terminologies

- Level: The root node is level 1. When we traverse down the hierarchy by one step, the level increases by 1.

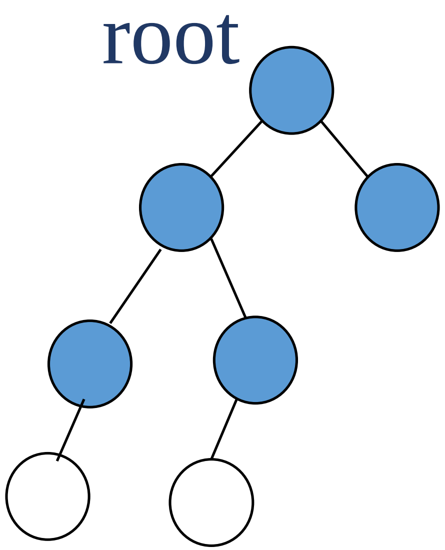
depth = height = level

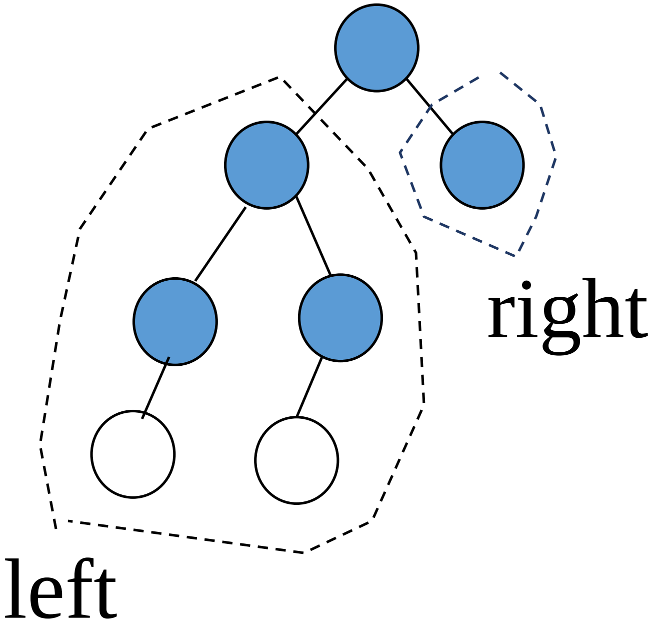
- Parent
- Grandparent
- Siblings
- Ancestors
- Descendants
- Node degree = number of children
- Tree degree = maximum node degree



# Binary Tree

- Finite (possibly empty) collection of elements.
- A nonempty binary tree has a root element.
- The remaining elements (if any) are partitioned into two binary trees.
- These are called the left and right subtrees of the binary tree.



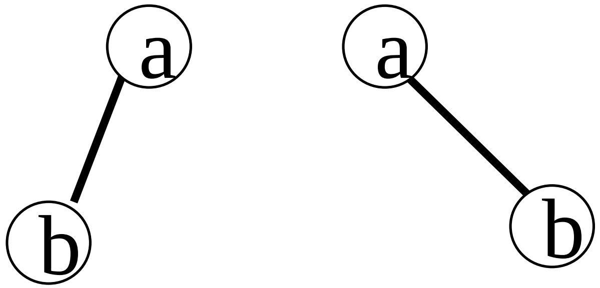
# A Tree and A Binary Tree

Their differences

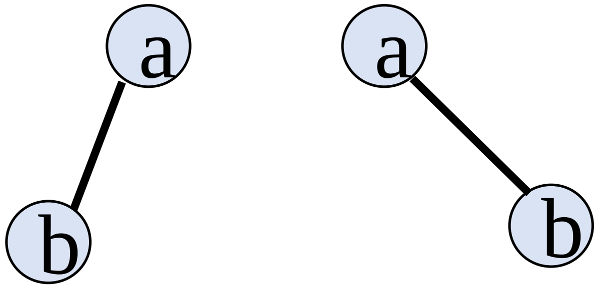
- The degree of a node in a binary tree must be two or fewer.
- There is no limit on the degree of a node in a tree.
- A tree cannot be empty.
- A binary tree may be empty.

# A Tree and A Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered.



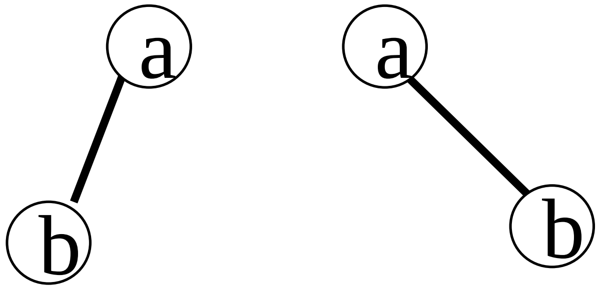

two trees



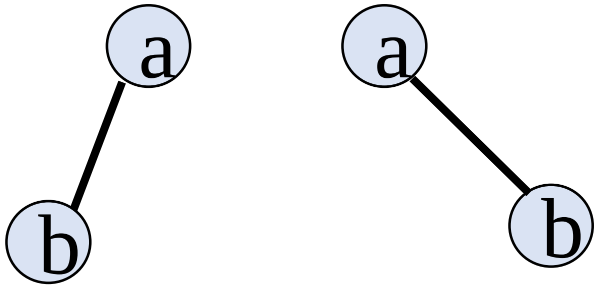

two binary trees

# A Tree and A Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered.




two trees

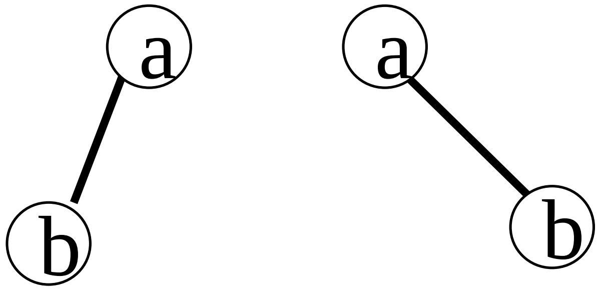



two binary trees

These two trees are the same.

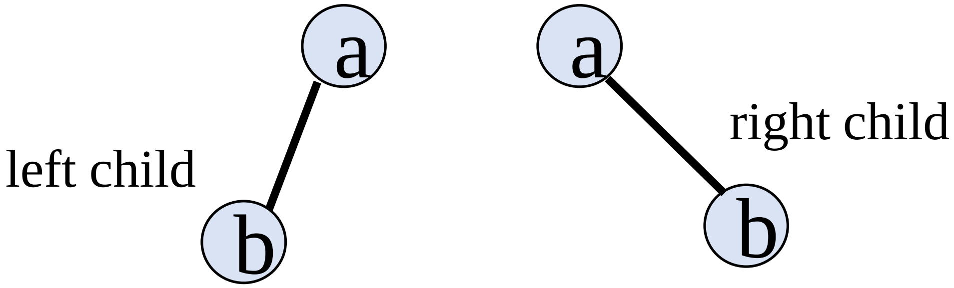
# A Tree and A Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered.




two trees

These two trees are the same.

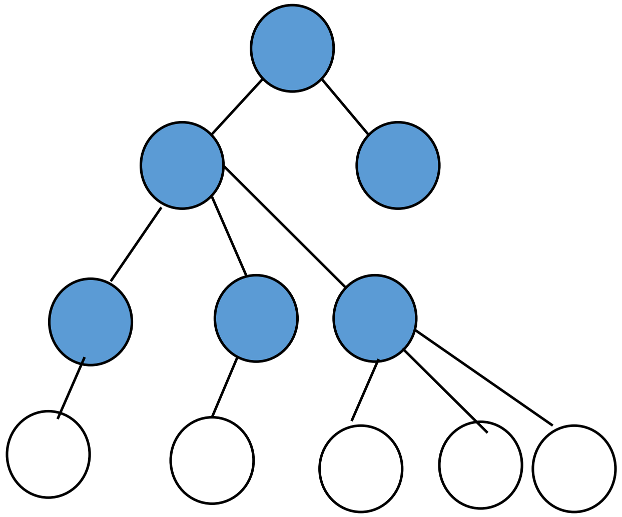



two binary trees

These two binary trees are not the same.

# Implementation: Tree

```
template<typename T> Node {
public:
        T value;
        vector<Node *> children;
};

template<typename T>  class Tree {
public:
        Node *root;
};// any bug(s)?
```



# Implementation: Tree

template<typename T> Node {

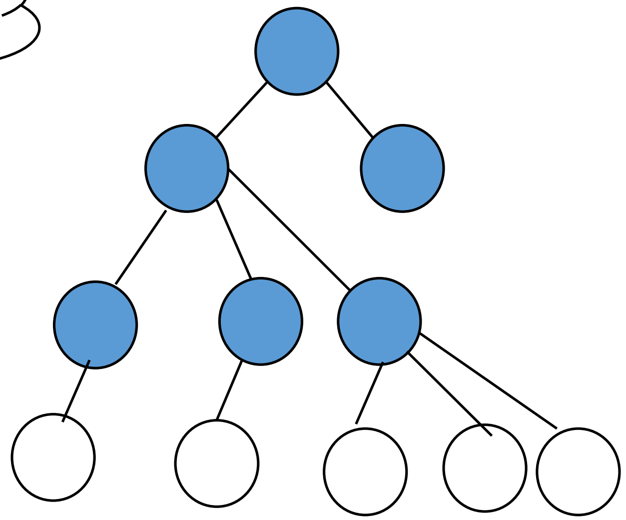
public:

T value;

Don’t require

vector<Node ~~<T>~~ *> children;

};



template<typename T>  class Tree {

public:

Node**<T>** *root;

};// any bug(s)?

# Implementation: Tree

template<typename T> class Node {

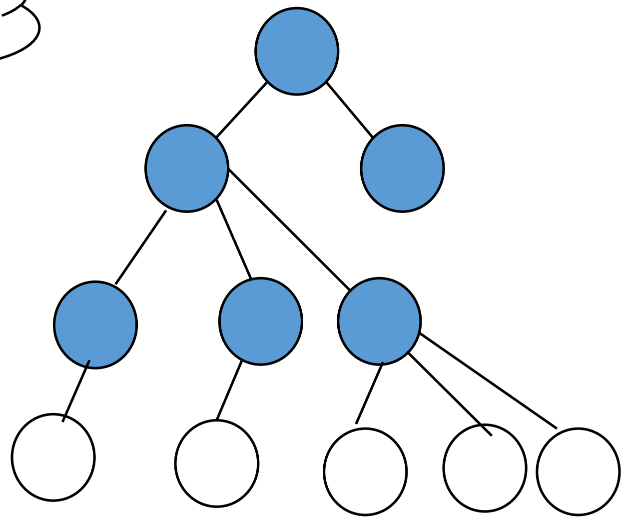
public:

T value;

vector<Node ~~<T>~~ *> children;

};

Don't require



template<typename T> class Tree {

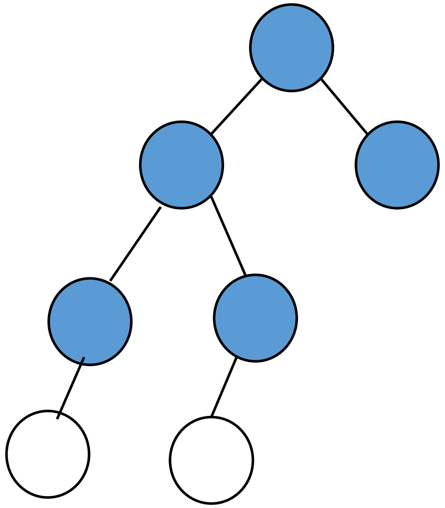
public:

Node**<T>** *root;

};// any bug(s)?

# Implementation: Binary Tree

```
template<typename T> Node {
public:
        T value;
        Node *left, right;
};

template<typename T>  class BinaryTree {
public:
        Node *root;
};// any bug(s)?
```



# Implementation: Binary Tree

```
template<typename T> class Node {
public:
        T value;
        Node<T> *left, *right; // using Node is also fine
};

template<typename T>  class BinaryTree {
public:
        Node<T> *root;                    //
};
```

# Implementation

```
template<typename T> class Node {
public:
        Node( ) : left (nullptr), right(nullptr)  { }
        T value;
        Node *left, *right;
};
template<typename T> class BinaryTree {
public:
        BinaryTree() { root = 0; }
        Node<T> *root;
        void insert( const T &a) { }
        void clear() { }
};
```

# Implementation

```
template<typename T> class Node {
public:
        Node( ) : left (nullptr),  right(nullpt), right(nullpt), right(nullpt)
, right(nullpt)                                                   // easier to manage
, right(nullpt)
, right(nullpt)
, right(nullpt)
, right(nullpt)
{ }
        T value;
        Node *left, *right;
};
template<typename T> class BinaryTree {
public:
        BinaryTree() { root = 0; }
        Node<T> *root;
        void insert( const T &a) { }
        void clear() { }
```

# Implementation

```
template<typename T> class Node {
public:
   Node ( ) : left (nullptr),  right(nullpt), right(nullpt), right(nullpt), right(nullpt), right(nullpt), right(nullpt), right(nullpt), right(nullpt)
{ }
        T value;
        Node *left, *right;
};
template<typename T> class BinaryTree {
public:
        BinaryTree() { root = 0; }
        Node<T> *root;
        void insert( const T &a) { }
        void clear() { }
};
```

Difficult to read

# Implementation

```
template<typename T> class Node {
public:
                Node( ) :
left (nullptr)
, right(nullpt)
, right(nullpt)
, right(nullpt)
, right(nullpt)
, right(nullpt)
, right(nullpt)
, right(nullpt)
, right(nullpt)

{ }
                T value;
                Node *left, *right;
};
template<typename T> class BinaryTree {
public:
                BinaryTree() { root = 0; }
                Node<T> *root;
                void insert( const T &a) { }
                void clear() { }
};
```

# Implementation

```
template<typename T> class Node {
public:
        Node( ) : left (nullptr), right(nullpt)  { }
        T value;
        Node *left, *right;
};
template<typename T> class BinaryTree {
public:
        BinaryTree( ): root(0) { … }
        Node <T> *root;
        void insert( const T &a) { … }
        void clear( ) {  … }
};
```

```
BinaryTree<int> bt;
BinaryTree<X> btx;
BinaryTree<vector<int>> bti;
```

# Implementation

```
template<typename T> class Node {
public:
        Node( ) : left ( nullptr ), right( nullptr )  { }
        T value;
        Node *left, *right; // Node<T> *left, *right. Also good
};
template<typename T> class BinaryTree {
public:
        BinaryTree( ) { root = 0; }
        Node < T > *root;
        void insert( const T &a ) { … }
        void clear( ) { … }
};
```

```
BinaryTree<int> bt;
BinaryTree<X> btx;
BinaryTree<vector<int>> bti;
```

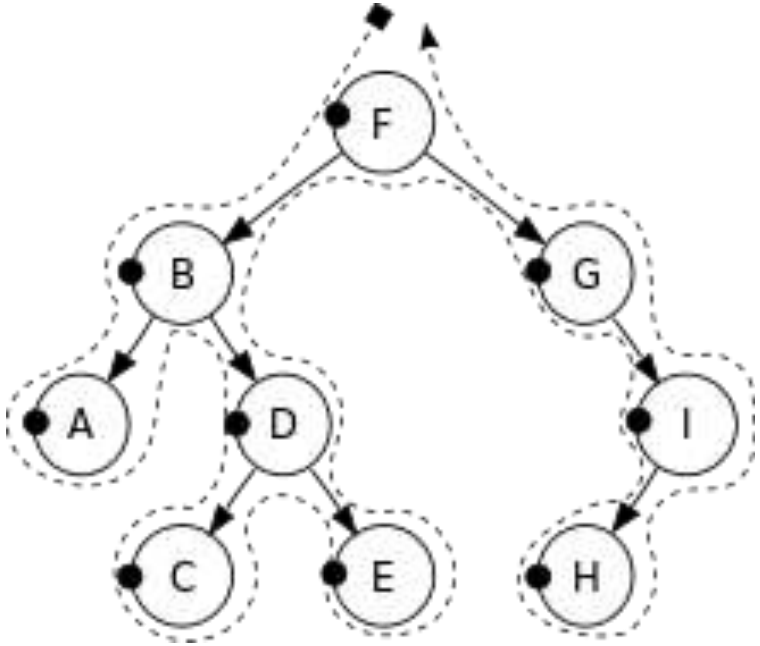
# Tree Traversal
https://en.wikipedia.org/wiki/Tree_traversal

- Unlike linked lists, one-dimensional arrays and other linear data structures, which are canonically traversed in linear order, trees may be traversed in multiple ways. They may be traversed in depth-first or breadth-first order.

- Three common ways to traverse them in depth-first order:
  - in-order, pre-order and post-order.

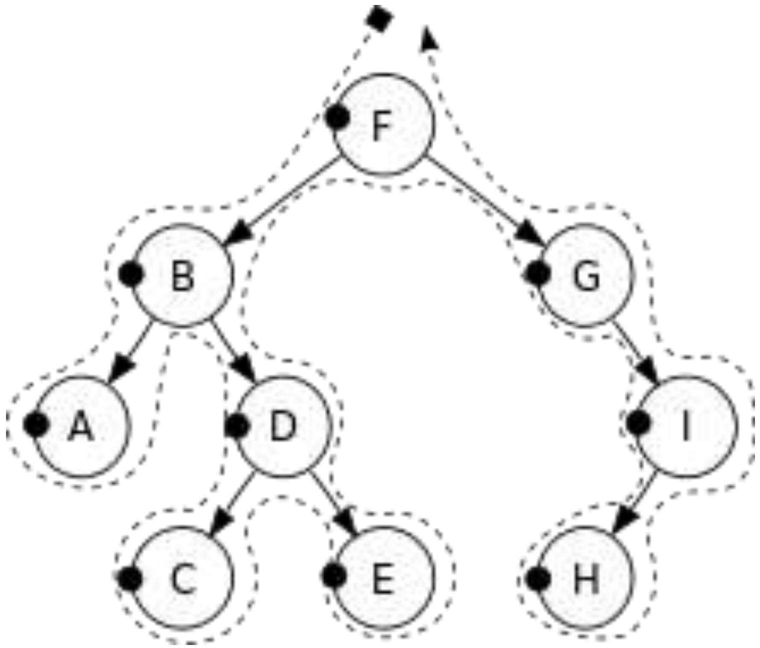
# Pre-Order
https://en.wikipedia.org/wiki/Tree_traversal

- Check if the current node is empty / null.
- Display the data part of the root (or current node).
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.



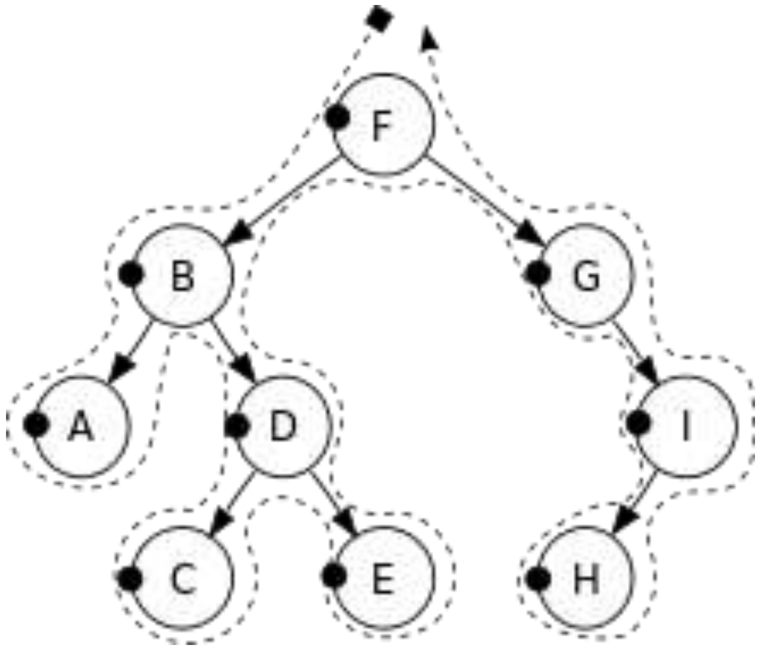
# Implementation: Pre-order

```
template <typename T> void traverse(const BinaryTree<T> *node)
{
        if (node ==0) return;
        cout << node << endl;
        traverse(node->left);
        traverse(node->right);
}
```



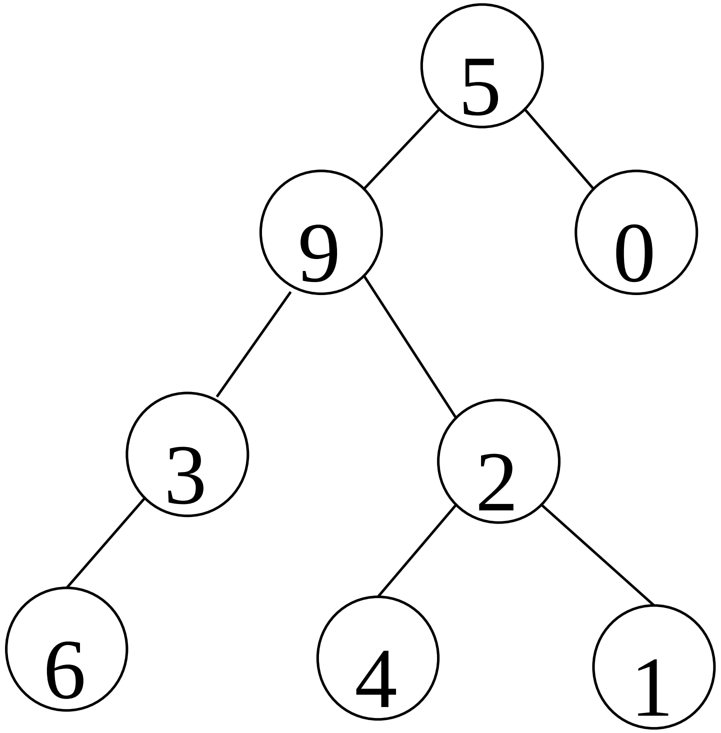
# Implementation: Pre-order

```
template <typename T> void traverse(const BinaryTree<T> *node)
{
        if (node ==0) return;
        cout << node << endl;
        traverse(node->left);
        traverse(node->right);
}
```

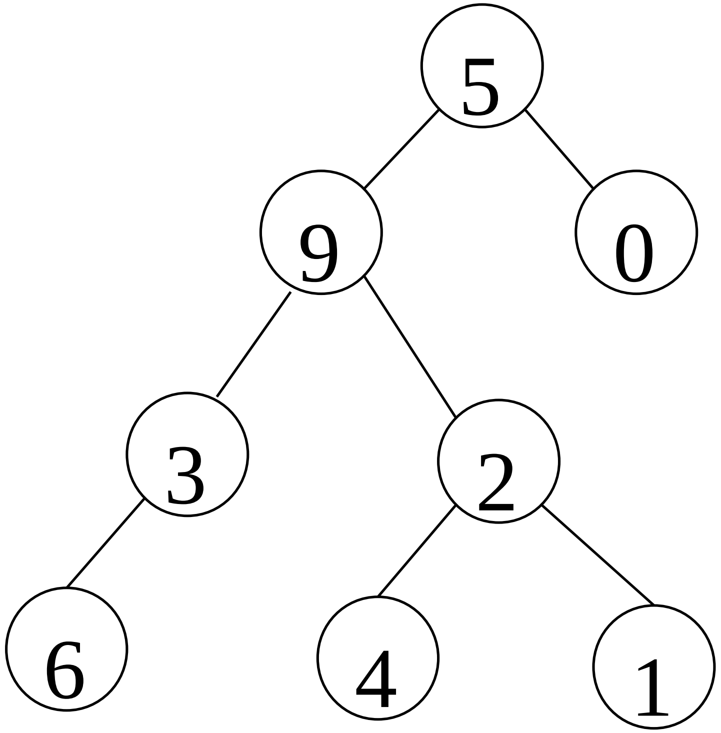



Pre-order: F, B, A, D, C, E, G, I, H.

# Pre-Order Exercises



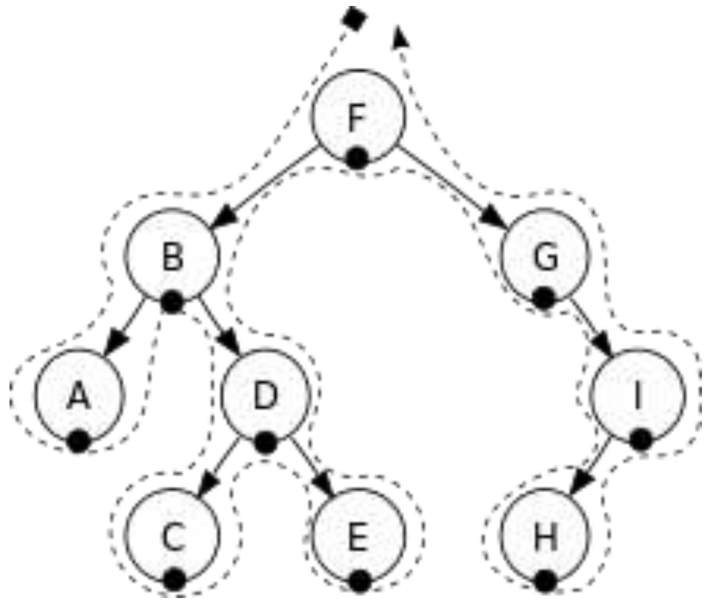
# Pre-Order Exercises




Pre-order: 5, 9, 3, 6, 2, 4, 1, 0

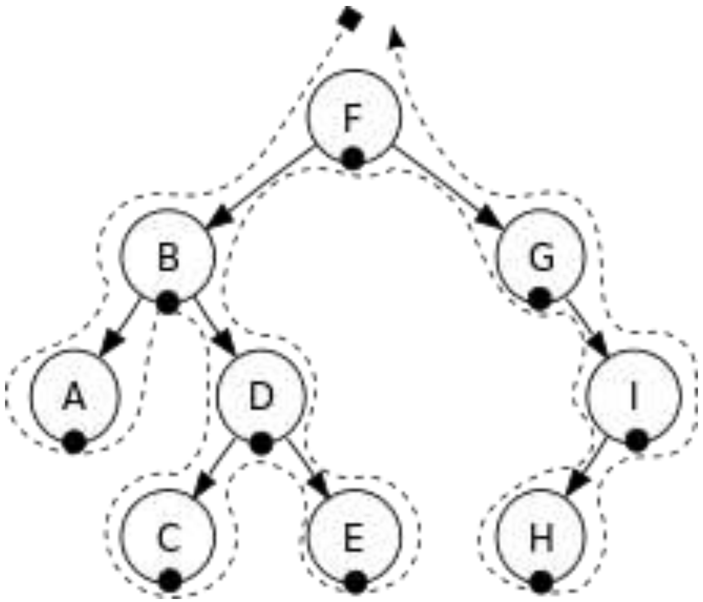
# In-Order
https://en.wikipedia.org/wiki/Tree_traversal

- Check if the current node is empty / null.
- Traverse the left subtree by recursively calling the in-order function.
- Display the data part of the root (or current node).
- Traverse the right subtree by recursively calling the in-order function.



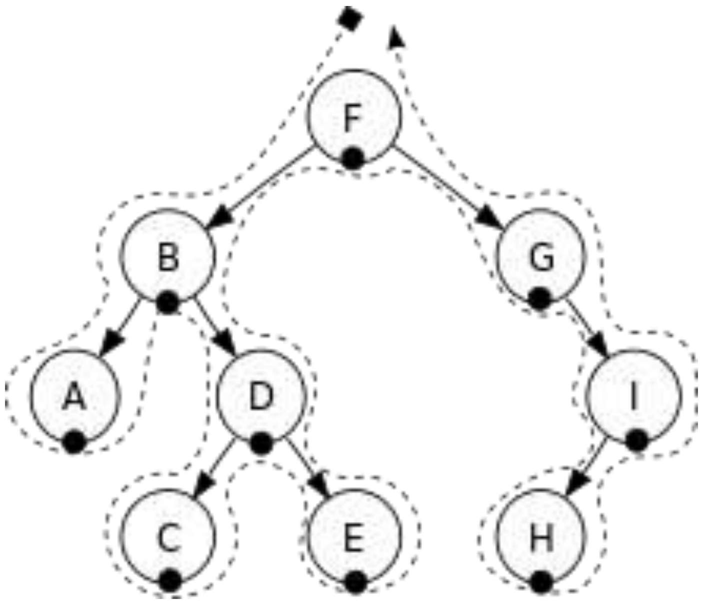
# Implementation: In-order

```
template <typename T> void traverse(const BinaryTree<T> *node)
{
        if (node ==0) return;
        traverse(node->left);
        cout << node << endl;
        traverse(node->right);
}
```



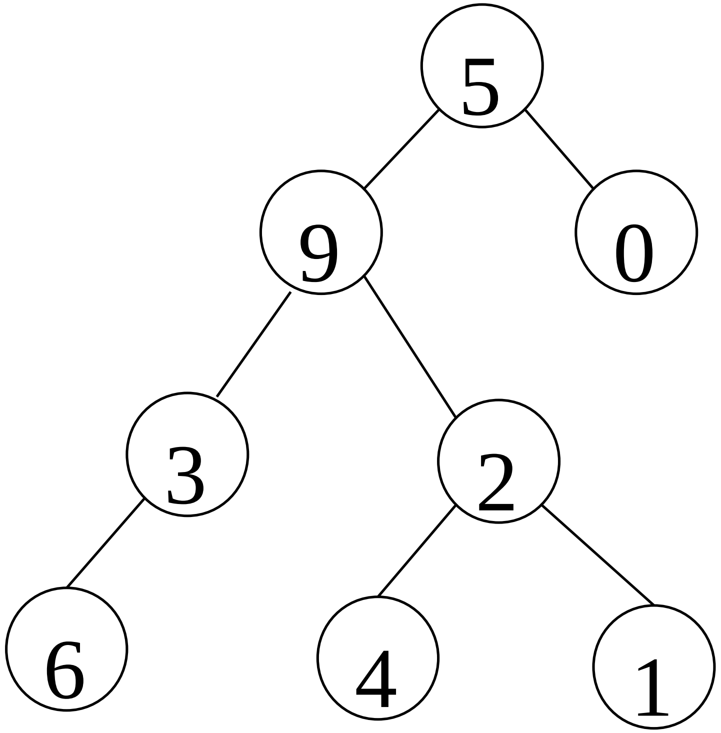
# Implementation: In-order

```
template <typename T> void traverse(const BinaryTree<T> *node)
{
        if (node ==0) return;
        traverse(node->left);
        cout << node << endl;
        traverse(node->right);
}
```

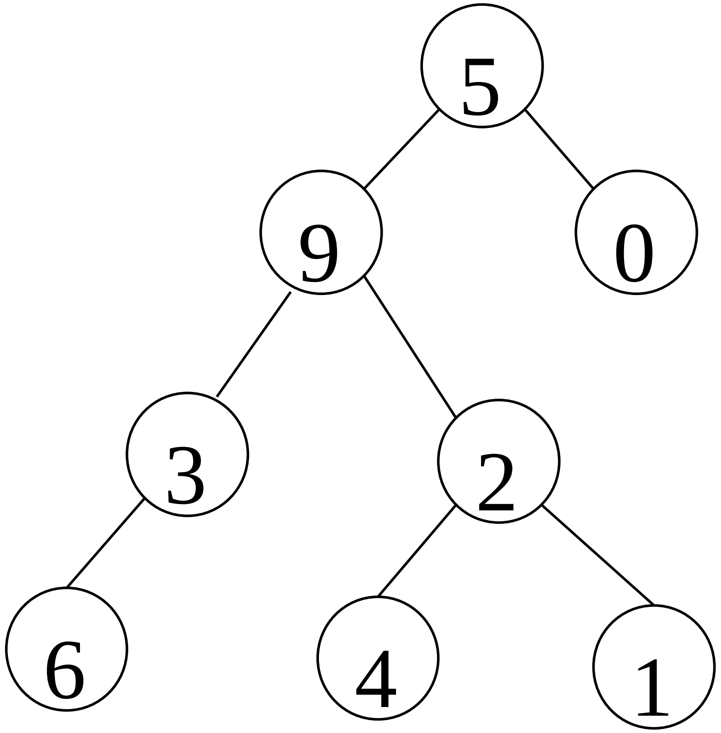



In-order: A, B, C, D, E, F, G, H, I.

# In-Order Exercises



# In-Order Exercises

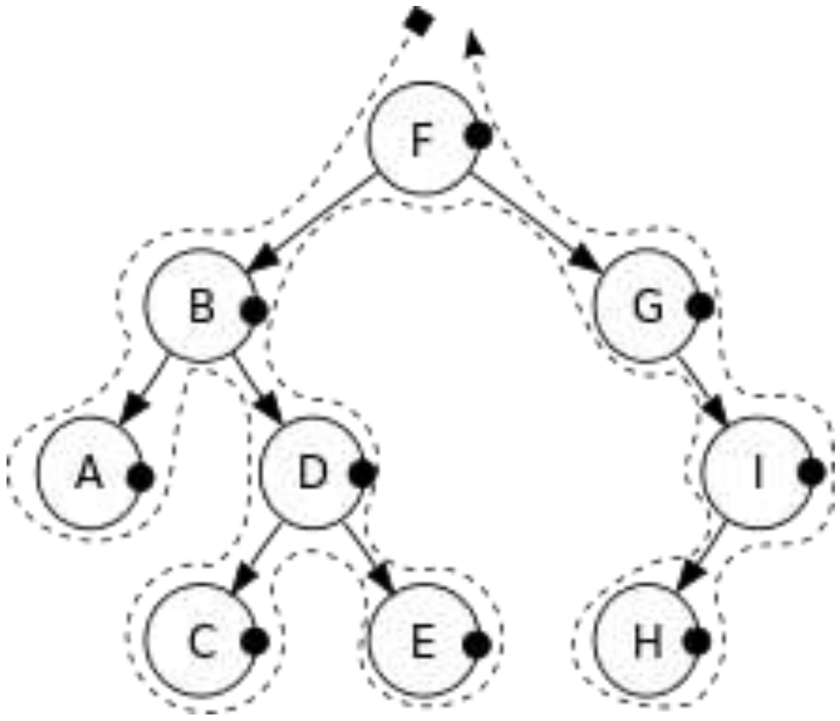



In-order: 6, 3, 9, 4, 2, 1, 5, 0

# Post-Order

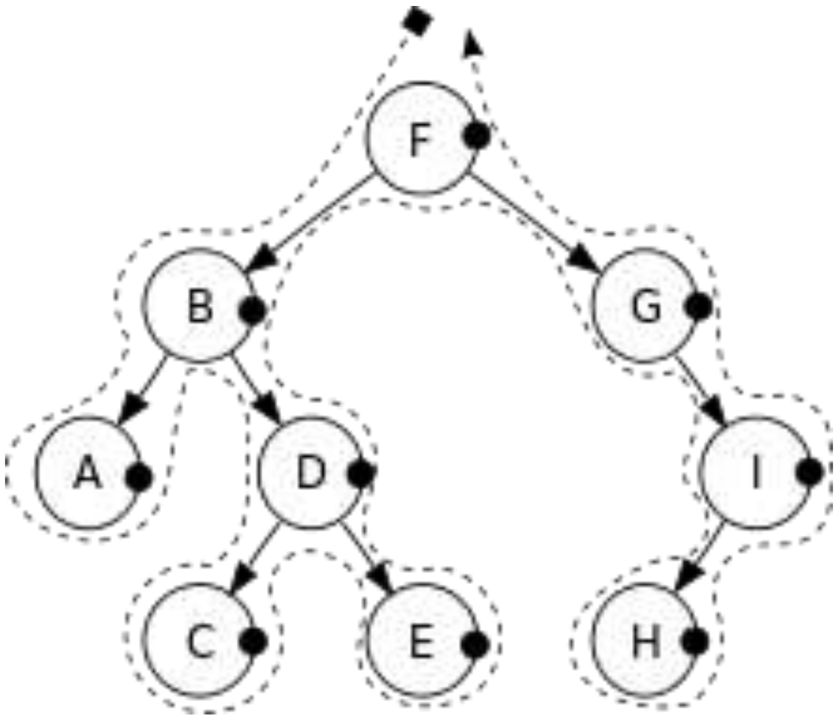
https://en.wikipedia.org/wiki/Tree_traversal

- Check if the current node is empty / null.
- Traverse the left subtree by recursively calling the post-order function.
- Traverse the right subtree by recursively calling the post-order function.
- Display the data part of the root (or current node).



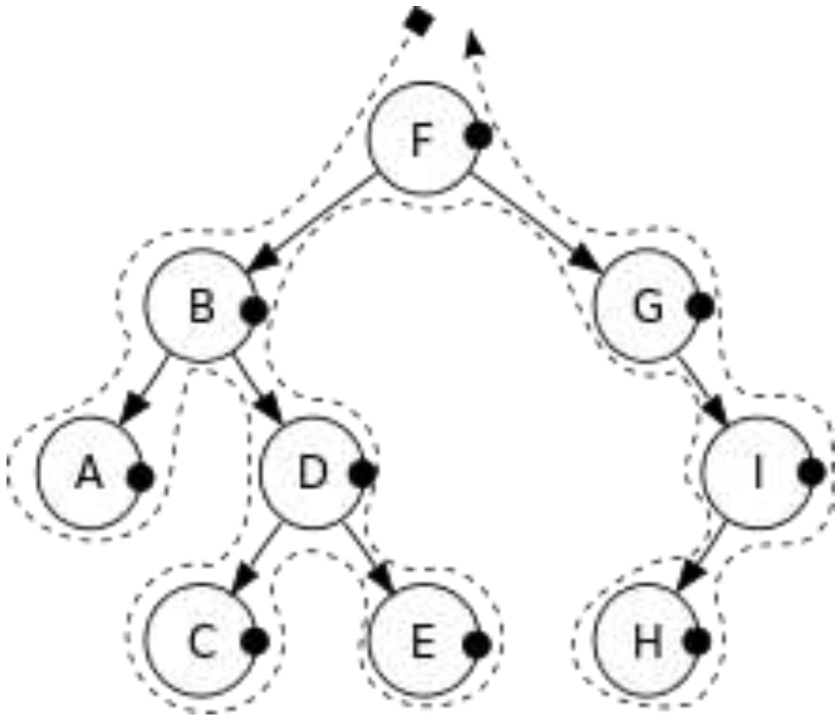
# Implementation: Post-order

```
template <typename T> void traverse(const BinaryTree<T> *node)
{
        if (node ==0) return;
        traverse(node->left);
        traverse(node->right);
        cout << node << endl;
}
```



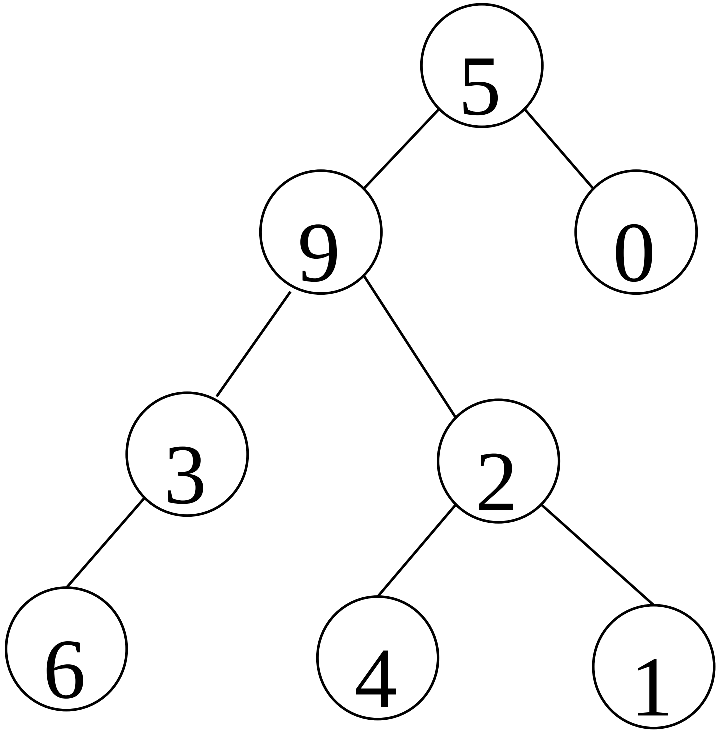
# Implementation: Post-order

```
template <typename T> void traverse(const BinaryTree<T> *node)
{
        if (node ==0) return;
        traverse(node->left);
        traverse(node->right);
        cout << node << endl;
}
```

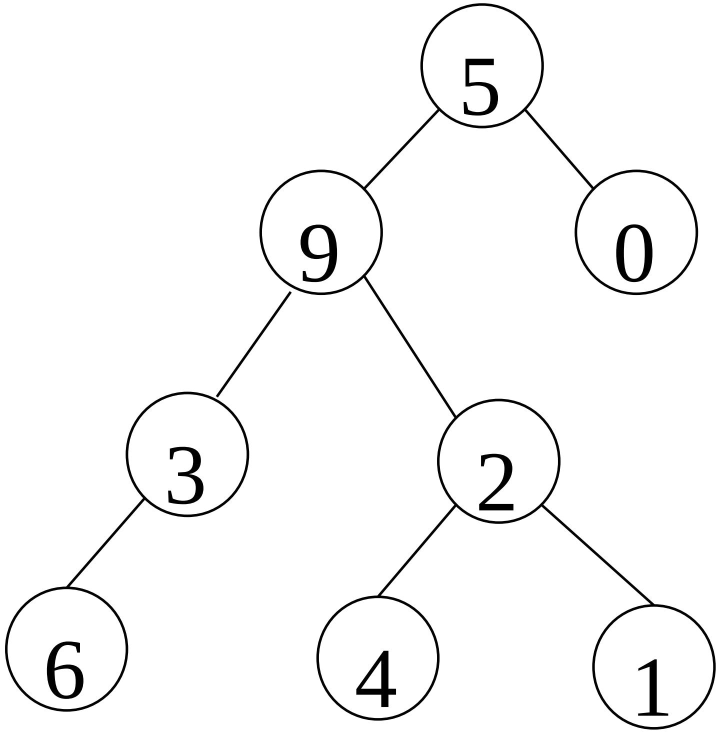



Post-order: A, C, E, D, B, H, I, G, F.

# Post-Order Exercises



# Post-Order Exercises

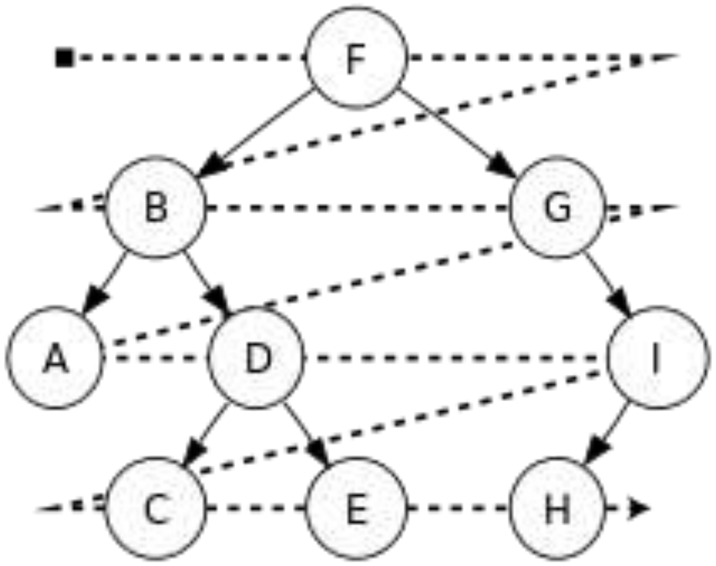



Post-order: 6, 3, 4, 1, 2, 9, 0, 5

# Breadth-First Order

https://en.wikipedia.org/wiki/Tree_traversal

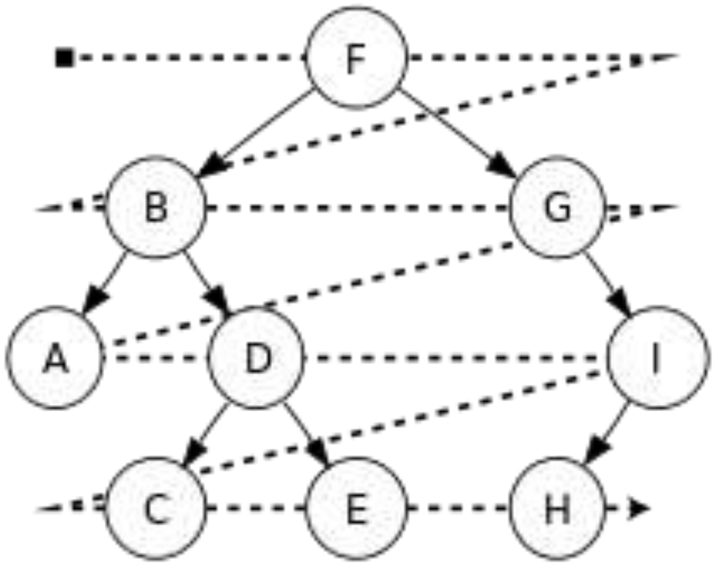
- Trees can also be traversed in *level-order*, where we visit every node on a level before going to a lower level. This search is referred to as *breadth-first search* (BFS), as the search tree is broadened as much as possible on each depth before going to the next depth.



# Implementation: Breadth-First Order

https://en.wikipedia.org/wiki/Tree_traversal

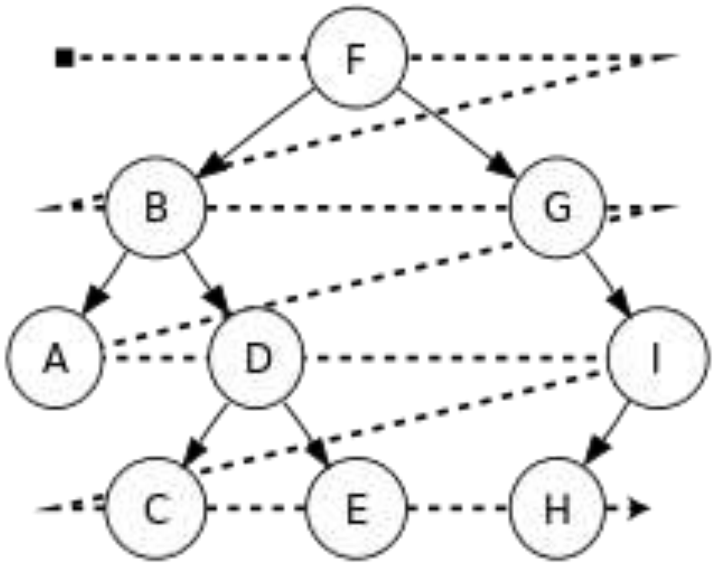
```
levelorder(root)
  q ← empty queue
  q.enqueue(root)
  while (not q.isEmpty())
    node ← q.dequeue()
    visit(node)
    if (node.left ≠ null)
      q.enqueue(node.left)
    if (node.right ≠ null)
      q.enqueue(node.right)
```



# Implementation: Breadth-First Order

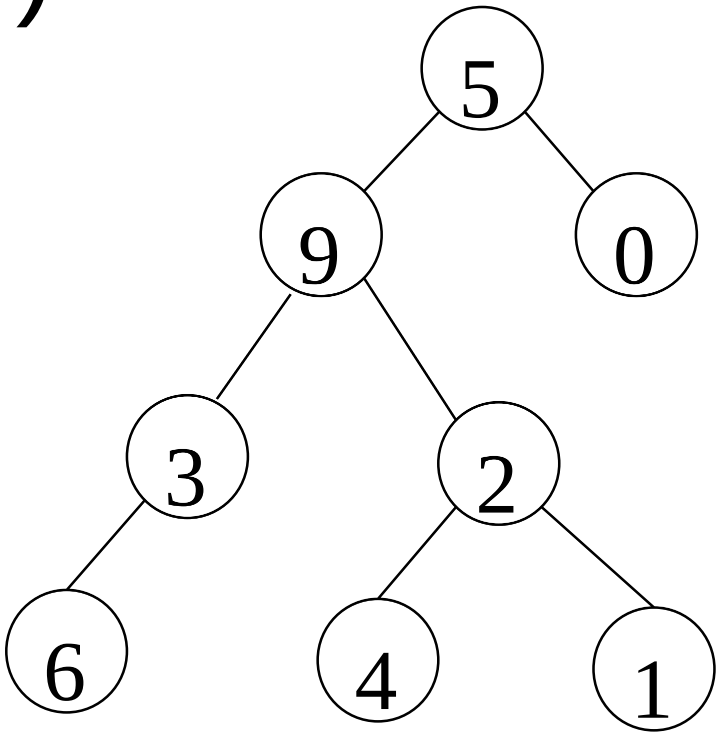
https://en.wikipedia.org/wiki/Tree_traversal

```
levelorder(root)
  q ← empty queue
  q.enqueue(root)
  while (not q.isEmpty())
    node ← q.dequeue()
    visit(node)
    if (node.left ≠ null)
      q.enqueue(node.left)
    if (node.right ≠ null)
      q.enqueue(node.right)
```

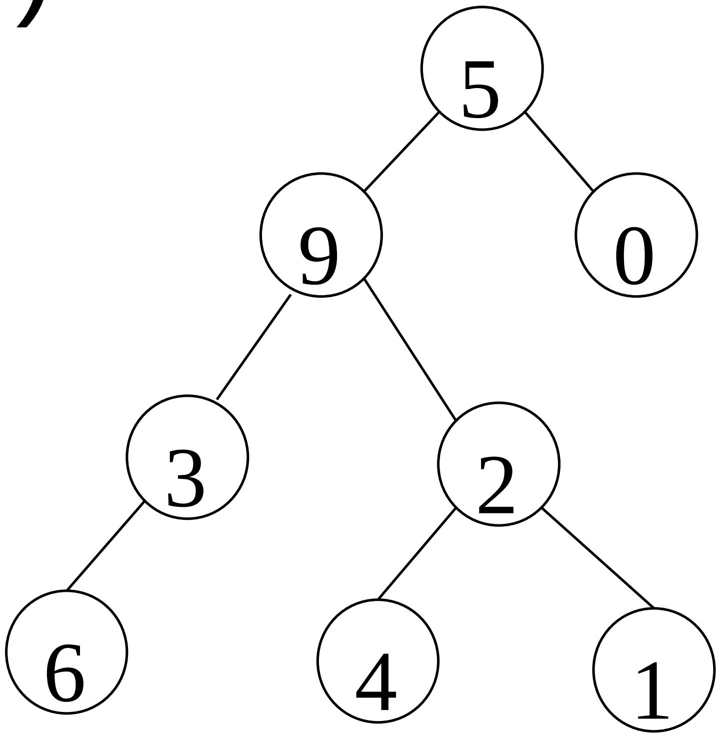



Level-order: F, B, G, A, D, I, C, E, H.

# Breadth-First Order Exercises (Level-Order)



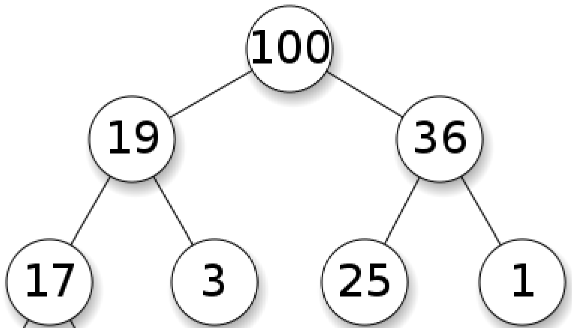
# Breadth-First Order Exercises (Level-Order)



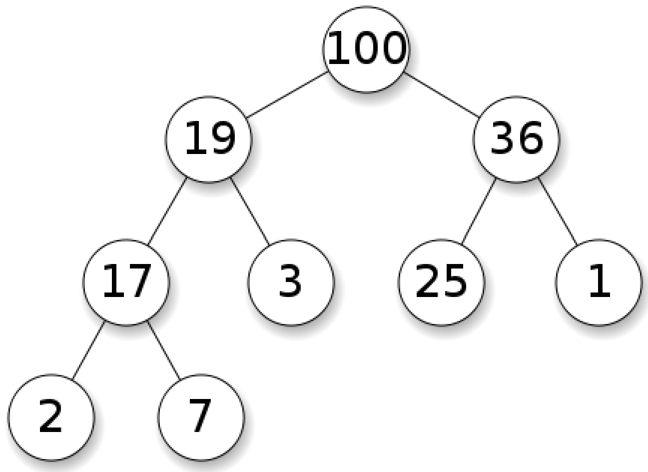

Breadth-First: 5, 9, 0, 3, 2, 6, 4, 1

# Perfect binary trees

A **perfect binary tree**: All internal nodes have two children and all leaves have the same depth or same level.



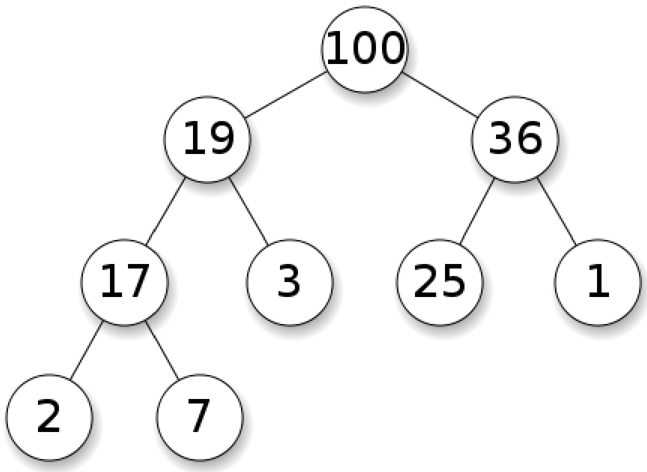

**A perfect binary tree**



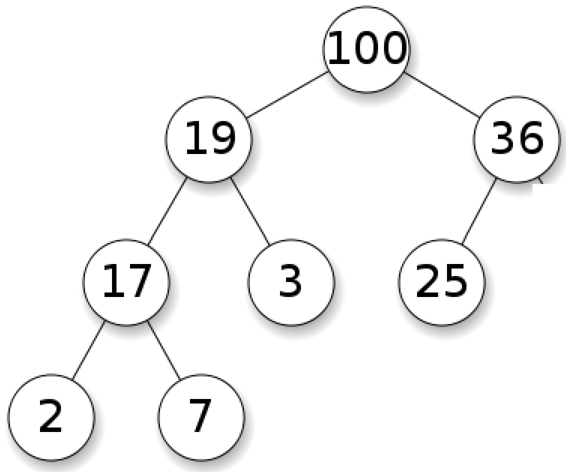

**Not a perfect binary tree**

# Complete binary trees

A **complete binary tree**: Every level, except possibly the last, is completely filled.



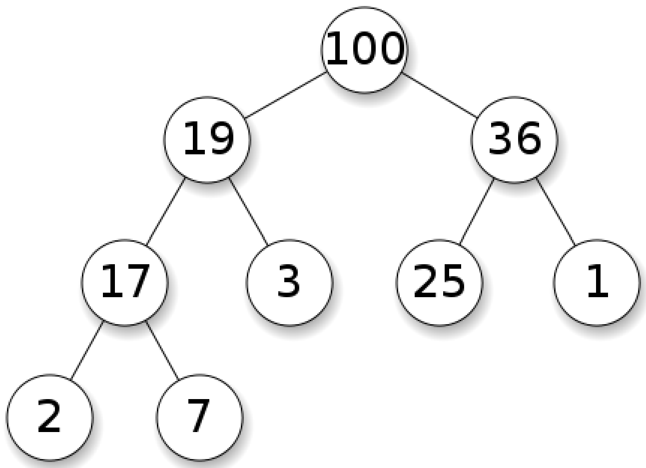

**A complete binary tree**



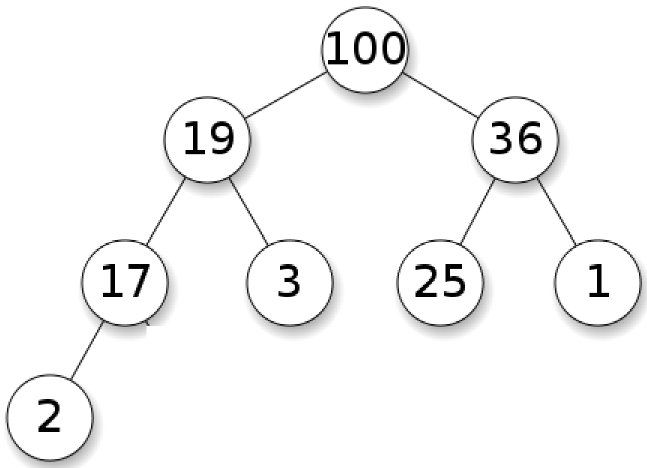

**Not a complete binary tree**

# Full binary trees

A **full binary tree**: every node has either 0 or 2 children.




**A full binary tree**




**Not a full binary tree**

# An Example

- Processing arithmetic expressions

# Arithmetic Expressions

An expression:

(a + b) * (c + d) + f – x*y/z + 7

- Expressions comprise three kinds of entities.

➢Operators: +, -, /, *

➢Operands: a, b, c, d, f, x, y, z, 7, (a + b), (c + d).

➢Delimiters: (,  )

# Operator Types

- Binary operator: need two operands.

  a + b

  x/z

  f-x

  b*c

- Unary operator: need one operand.

  +x

  -y

# Infix Form

- Binary operators are in between their left and right operands.

  a + b

  x*y/z

  (a + b) * (c + d)

  (a + b) * (c + d) + f – x*y/z + 7

# Operator Priorities

- How do we figure out the operands of an operator?

    a + b

    x/y – z*w

- Use operator priorities to resolve the issue.

    **^** > ***** = **/** > **+** = **-**

- ^ is the power operator ( it is the bitwise XOR op in C++)
- An operand is associated with an adjacent operator which has higher priority.

# Left Associativity

- The operators are left associative.
- An operand is associated with the operator on the left if the operator on the right has the same priority.

    x + y - z

    z / x * y / w

# Right Associativity

- The operators are right associative.
- An operand is associated with the operator on the right if the operator on the left has the same priority.

x^y^z : ^ is the power operator

z^y^x^w = z^(y^(x^w))

# Delimiters

- A subexpression enclosed by a pair of delimiters is a single operand
- It is independent from the remainder of the expression.

(a + b) * (c + d) + f – ( x*(y/z) + 7)

# Postfix Form

- The order of operands is the same as in the infix form.

- Operators come at once after the postfix form of their operands.

Infix = x + y

Postfix = xy+

# Examples

Infix = x + y
Postfix = xy+

Infix = (a + b) * (c + d)
Postfix = ab+cd+*

Infix = f – ( x*(y/z) + 7)
Postfix = fxyz/*7+ –

Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)
Postfix = ab+cd+*f+xyz/*7+ –

# Unary Operators

- The symbols of the unary operators + and – are the same as the binary operators + and –.
- Replace with new symbols.

  + a => a @

  + a + b => a @ b +

  - a => a $

  - a-b => a $ b -

@ and $ are new symbols that are used to denote the unary + and – operators, respectively.

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)
Postfix = ?

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)
Postfix = ab+cd+*

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)
Postfix = ab+cd+*

ab+cd+*

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)

Postfix = ab+cd+*

`ab+cd+*`

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)

Postfix = ab+cd+*

`ab+cd+*`   ab**+**cd+*

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)

Postfix = ab+cd+*

`ab+cd+*` **`ab+`**`cd+*`

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)
Postfix = ab+cd+*

`r = a+b`

`ab+cd+*` **`r`**`cd+*`

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)

Postfix = ab+cd+*

`r = a+b`

`ab+cd+*   rcd`**`+`**`*`

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)

Postfix = ab+cd+*

`r = a+b`

`ab+cd+*` `r`**`cd+`**`*`

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)
Postfix = ab+cd+*

`ab+cd+*`   `r`**`s`**`*`

`r = a+b`

`s = c+d`

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)
Postfix = ab+cd+*

`ab+cd+*   rs*`

`r = a+b`

`s = c+d`

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)

Postfix = ab+cd+*

`ab+cd+*` **`rs*`**

`r = a+b`

`s = c+d`

# Postfix Evaluation

- We scan the postfix expression from left to right and push operands on to a stack.
- When an operator is encountered, pop operands that are required for the operator.
- Evaluate the operator and push the result on to the stack.
- Repeat the process until all operators are evaluated.

Infix = (a + b) * (c + d)
Postfix = ab+cd+*

`ab+cd+*`   **`t`**

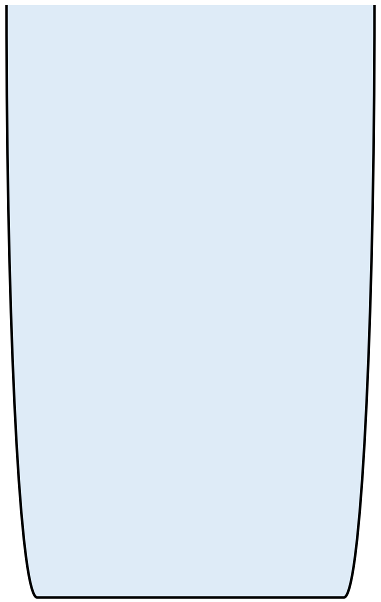
`r = a+b`

`s = c+d`

`t = r*s`

# Postfix Evaluation

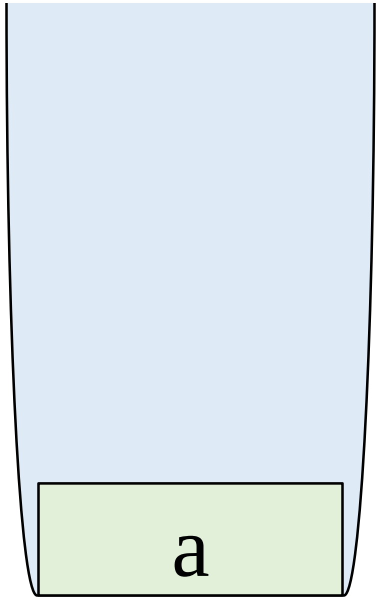
```
Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)

Postfix = ab+cd+*f+xyz/*7+ –
```



# Postfix Evaluation

```
Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)
```

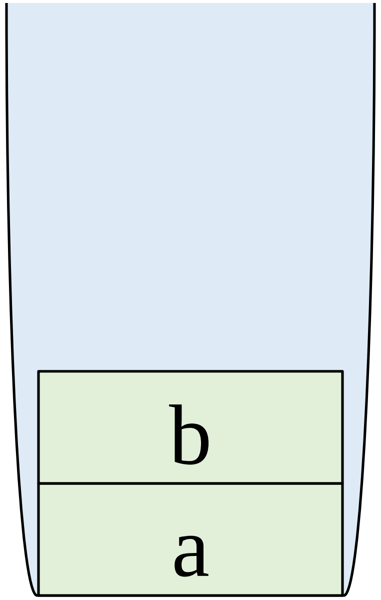
Postfix = **a**b+cd+*f+xyz/*7+ −



# Postfix Evaluation

```
Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)
```

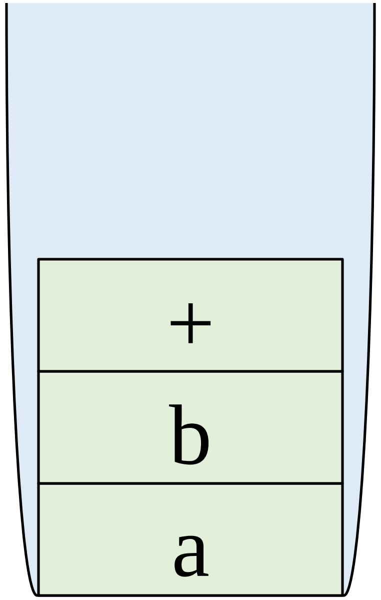
Postfix = **ab**+cd+*f+xyz/*7+ −



# Postfix Evaluation

```
Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)
```

Postfix = **ab+**cd+*f+xyz/*7+ −



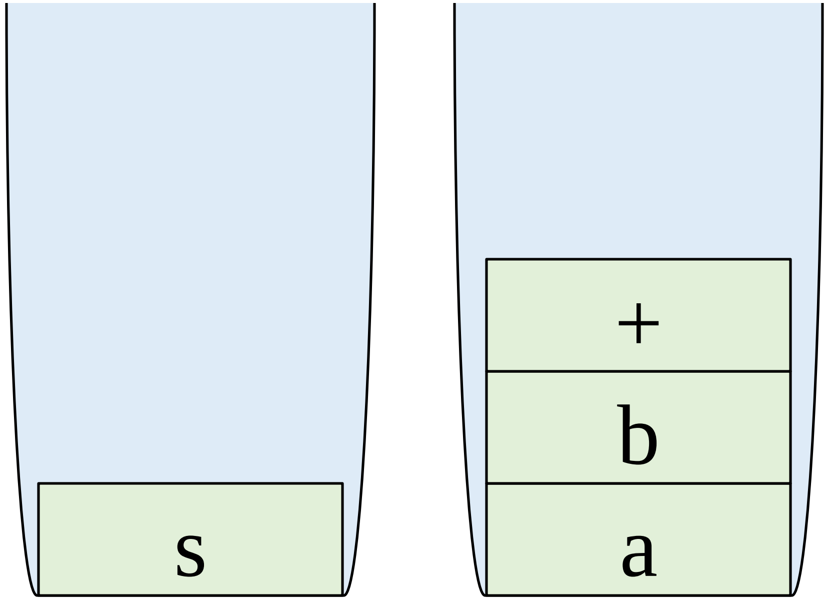
# Postfix Evaluation

```
Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)
```

Postfix = **ab+**cd+*f+xyz/*7+ –

Evaluate a+b

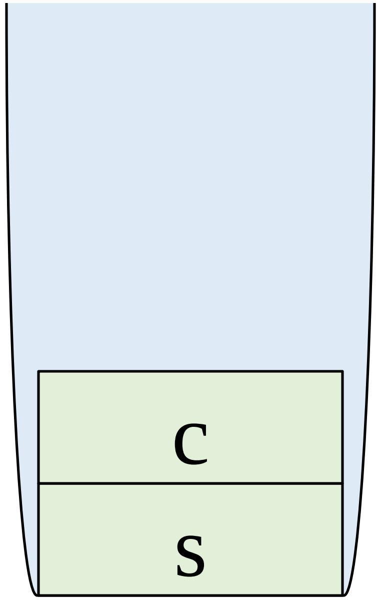
Assign the result to s



# Postfix Evaluation

```
Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)
```

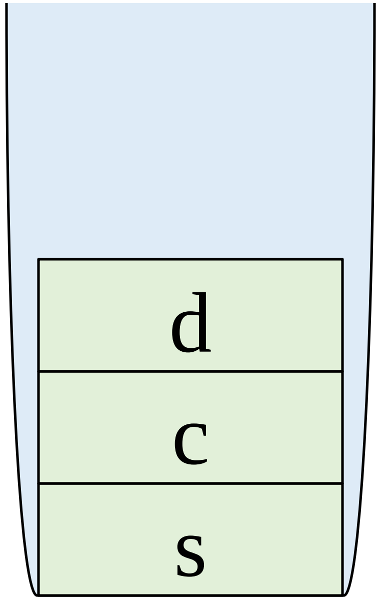
Postfix = **ab+c**d+*f+xyz/*7+ −



# Postfix Evaluation

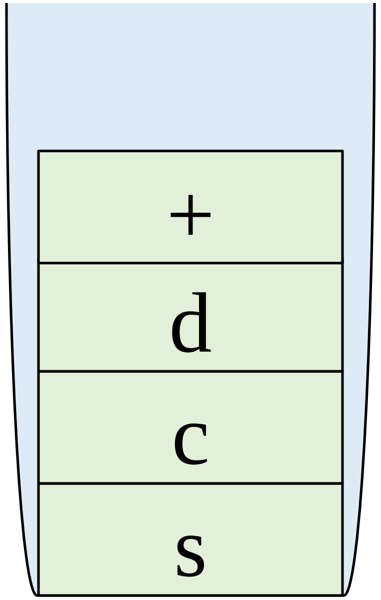
```
Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)
```

Postfix = **ab+cd**+*f+xyz/*7+ −



# Postfix Evaluation

```
Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)

Postfix = ab+cd+*f+xyz/*7+ −
```



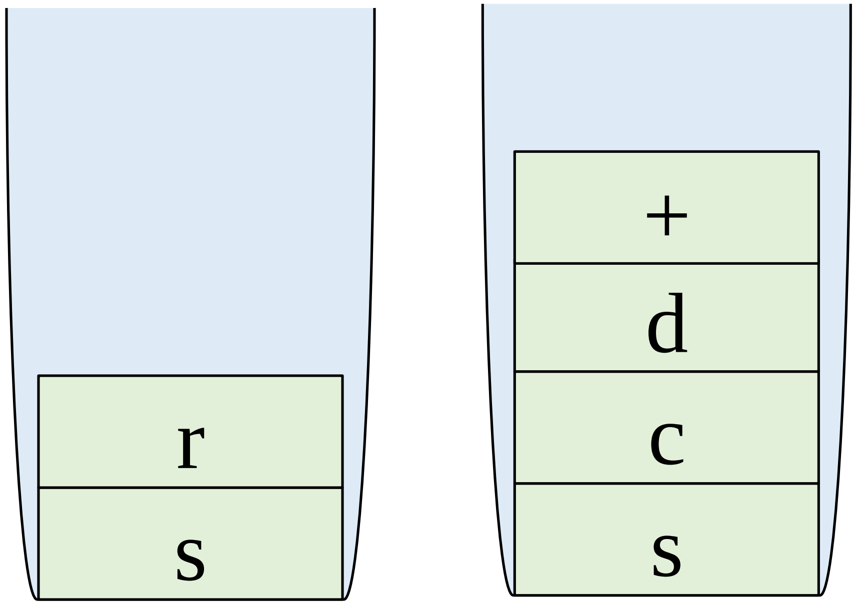
# Postfix Evaluation

Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)

Postfix = **ab+cd+***f+xyz/*7+ –

Evaluate
c+d

Assign the
result to r

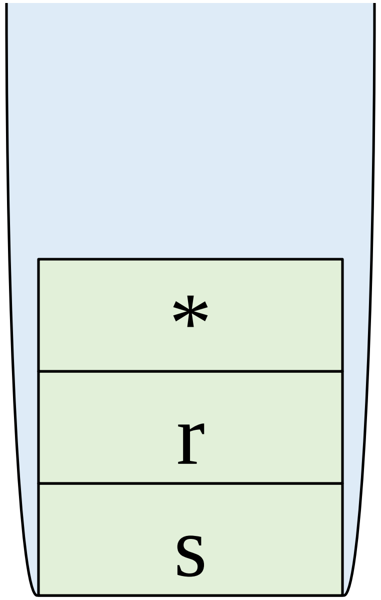


# Postfix Evaluation

```
Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)

Postfix = ab+cd+*f+xyz/*7+ −
```

Evaluate
c+d

Assign the
result to r



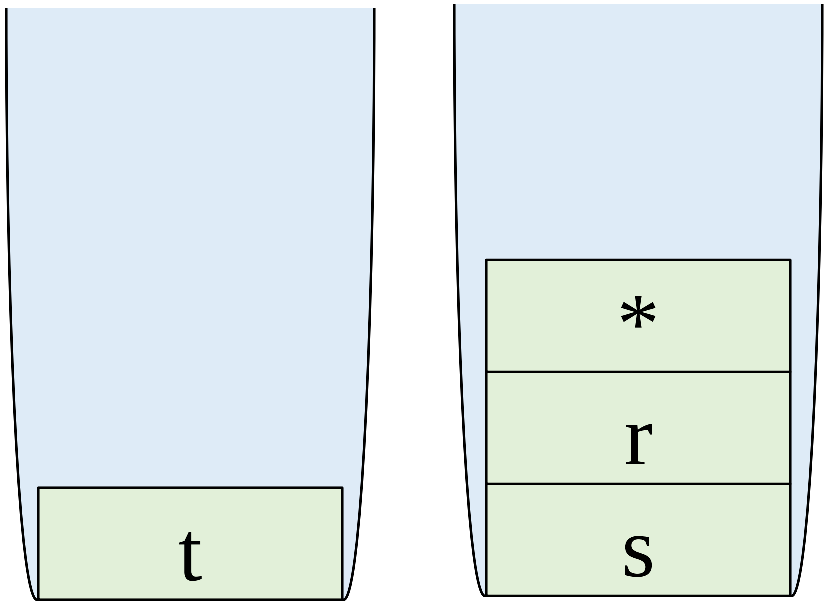
# Postfix Evaluation

Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)

Postfix = **ab+cd+***f+xyz/*7+ −

Evaluate
r+s

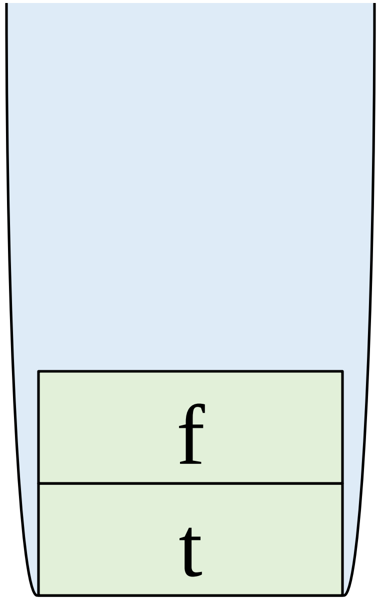
Assign the
result to t



# Postfix Evaluation

`Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)`

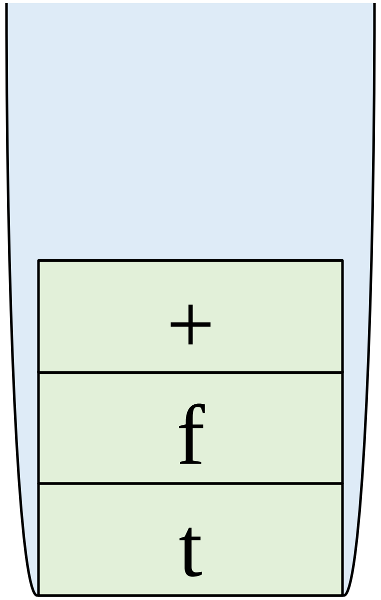
Postfix = **ab+cd+*f**+xyz/*7+ –



# Postfix Evaluation

```
Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)
```

Postfix = **ab+cd+*f+**xyz/*7+ −

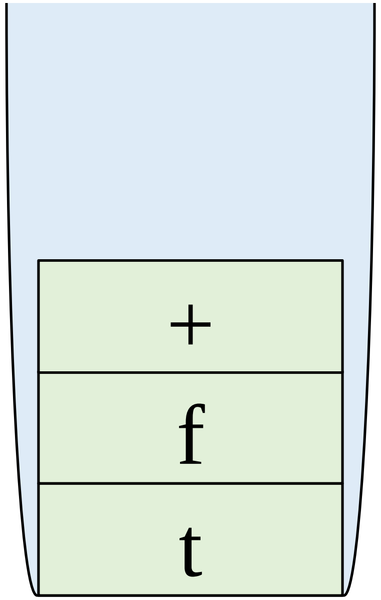


# Postfix Evaluation

```
Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)
```

Postfix = **ab+cd+*f+**xyz/*7+ −

Repeat the process until all the operators are evaluated.



# Prefix Form

- The order of operands is the same as in the infix and post-fix forms.
- Operators come at once before the prefix form of their operands.

Infix = a + b

Postfix = ab+

Prefix = +ab

# Examples

```
Infix = x + y
Prefix = +xy

Infix = (a + b) * (c + d)
Prefix = *+ab+cd

Infix = f − ( x*(y/z) + 7)
Prefix = −f+*x/yz7

Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)
Prefix = −+*+ab+cdf+*x/yz7
```

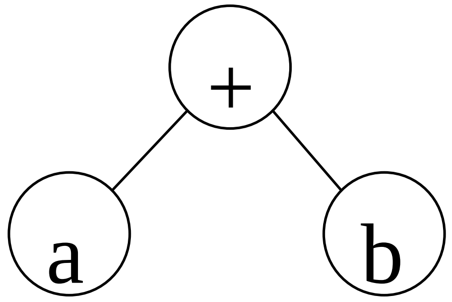
# Binary Tree Form

- Each leaf represents a variable or constant.

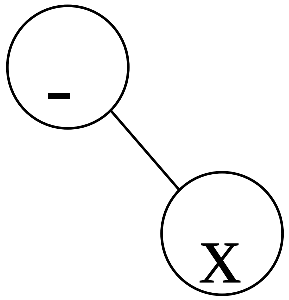
- Each nonleaf represents an operator.

- The left operand of an operator is represented by the left subtree.
- The right subtree represents the right operand of the operator.

a + b

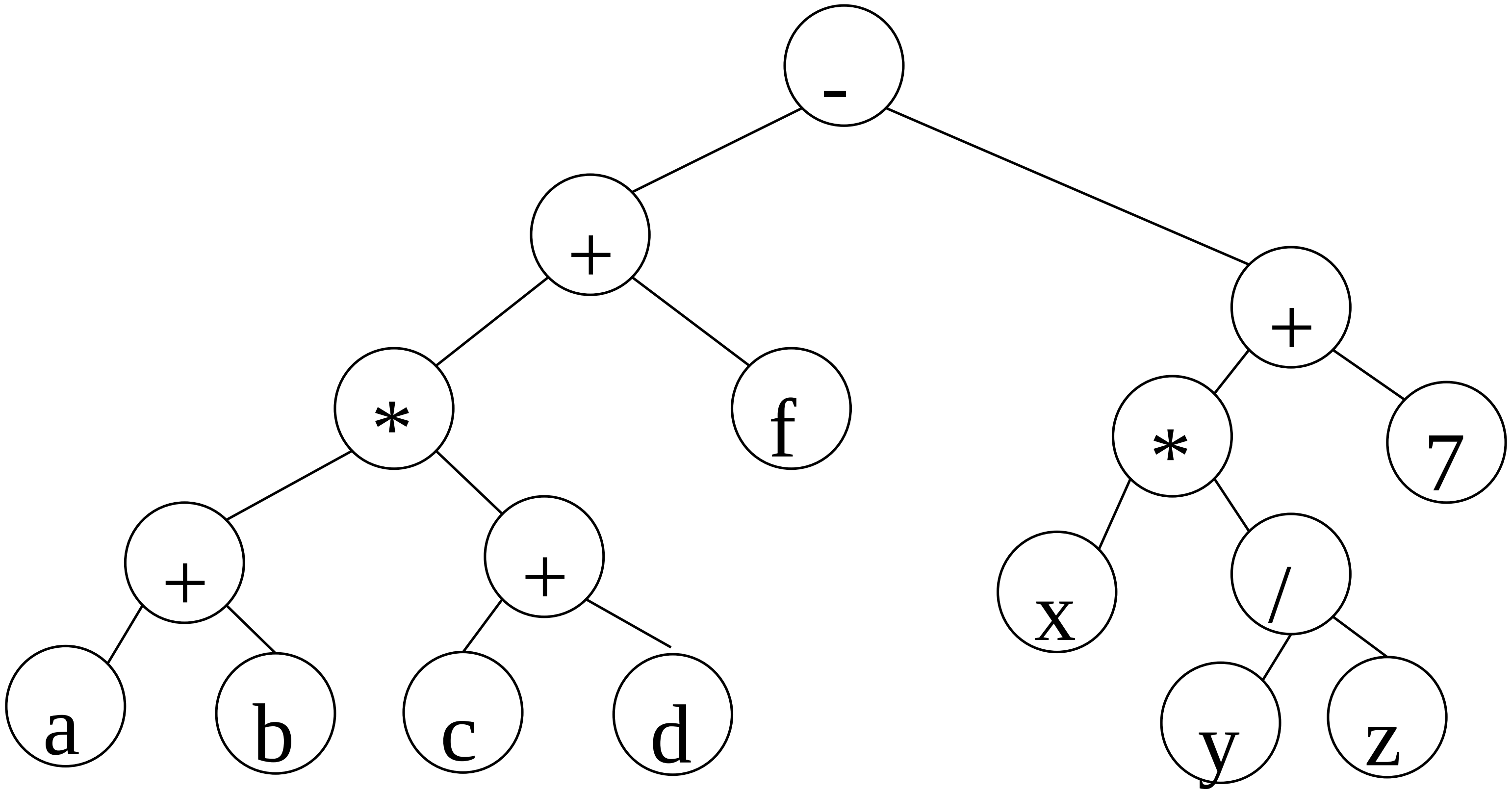



- x



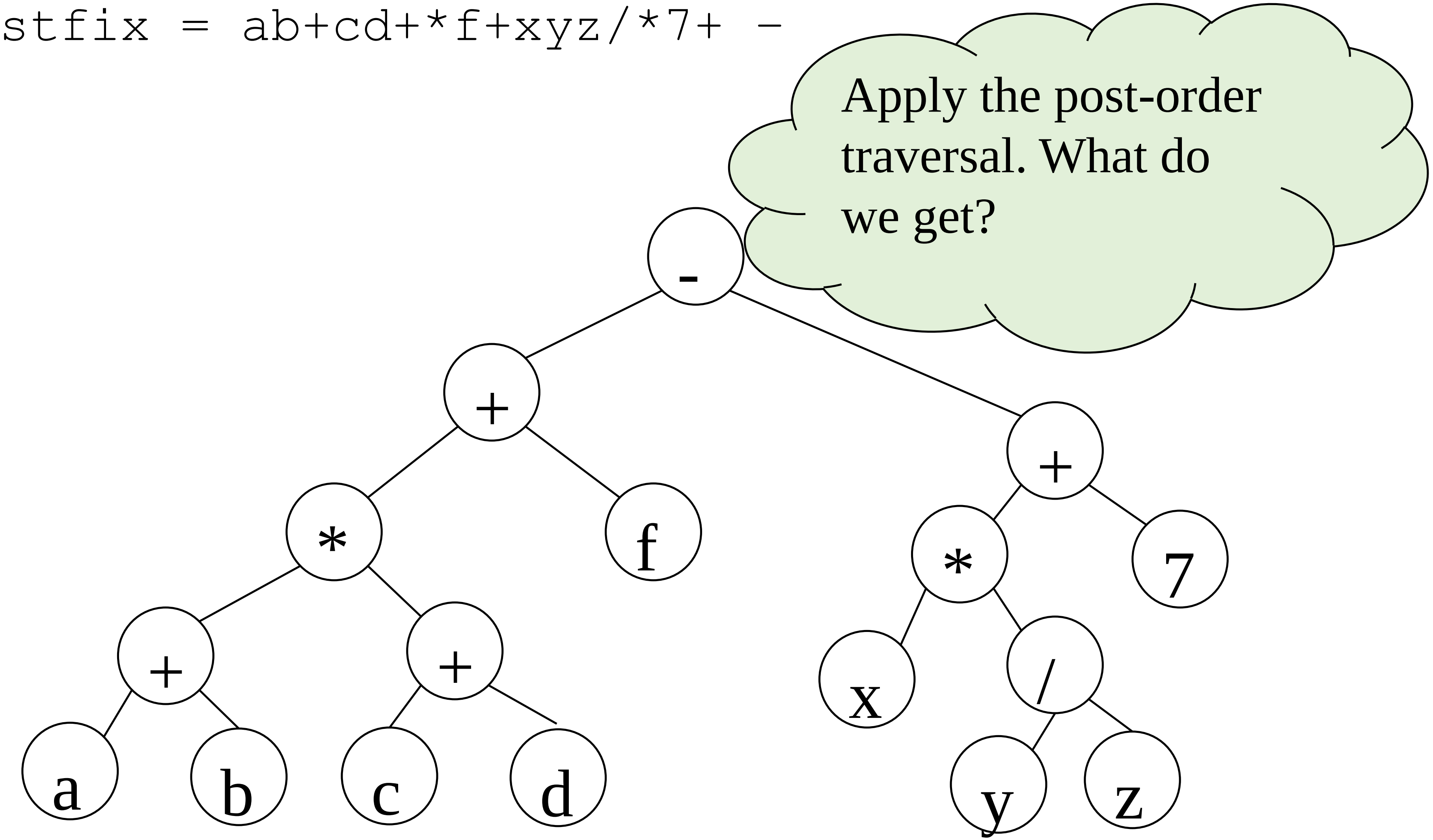
# Binary Tree Form

```
Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)

Postfix = ab+cd+*f+xyz/*7+ –
```



# Binary Tree Form

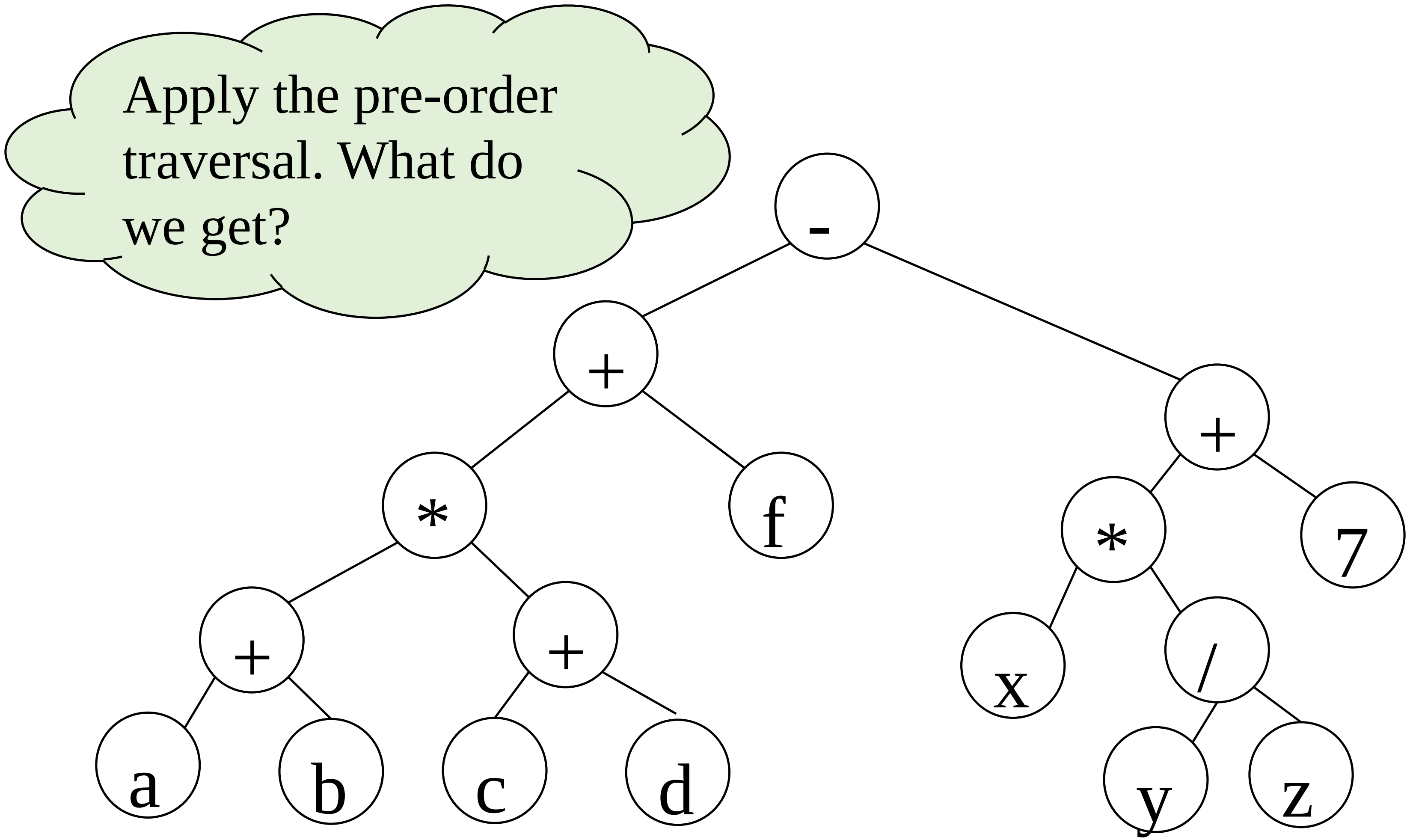
```
Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)
Postfix = ab+cd+*f+xyz/*7+ –
```



# Binary Tree Form

Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)

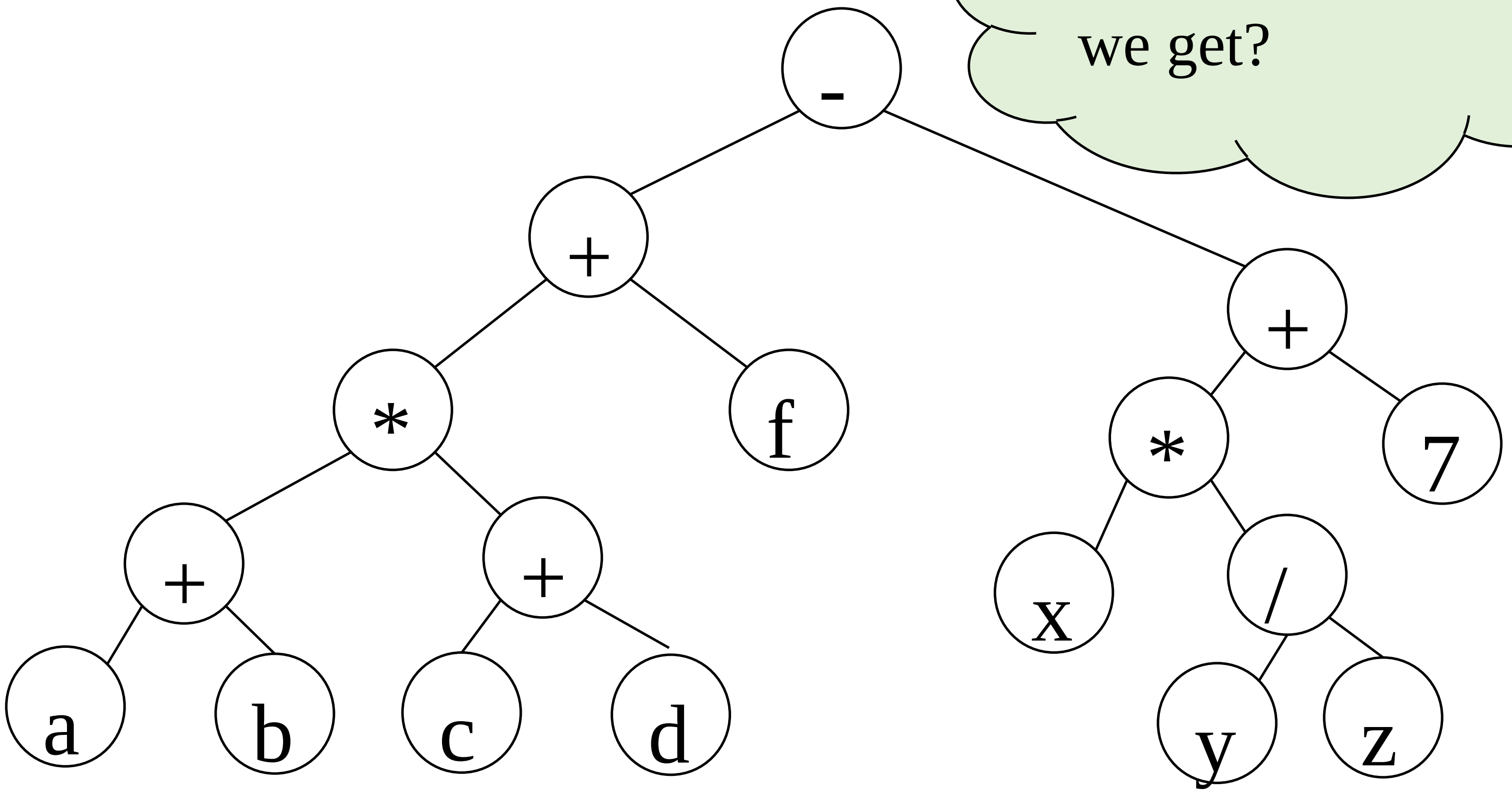
Postfix = ab+cd+*f+xyz/*7+ −



# Binary Tree Form

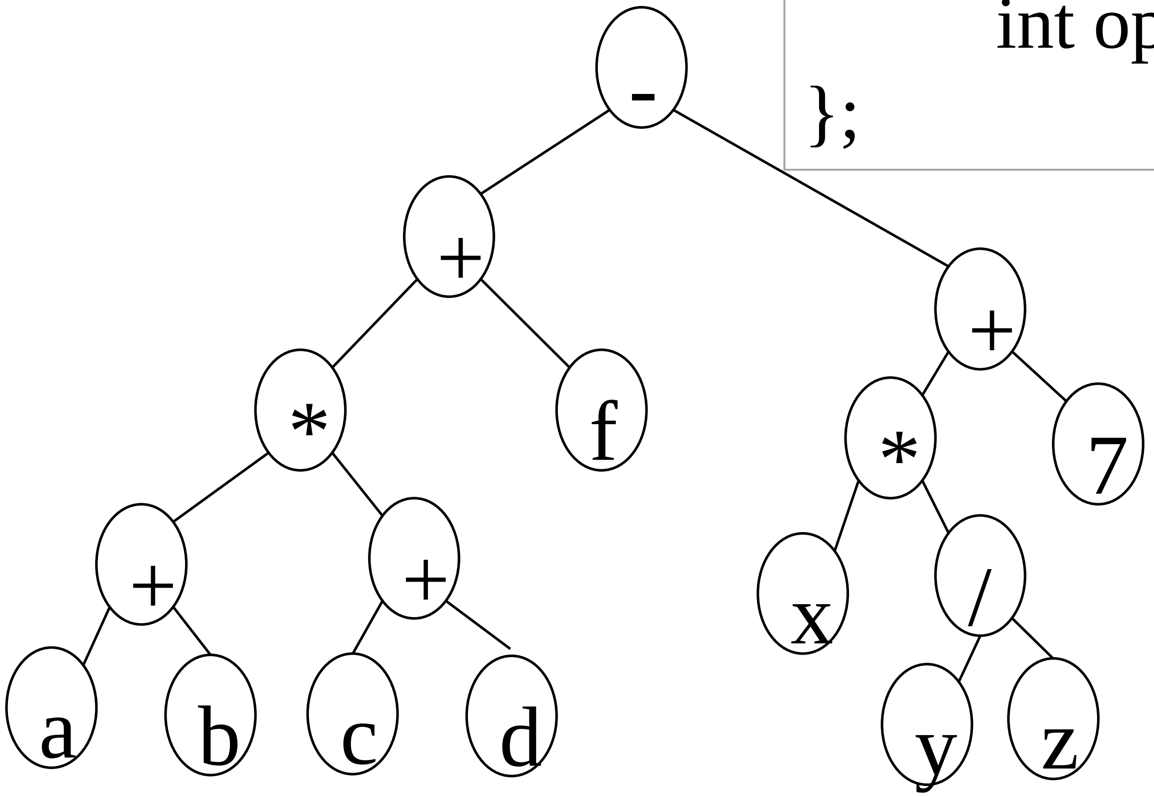
```
Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)
Postfix = ab+cd+*f+xyz/*7+ –
```

Apply the in-order traversal. What do we get?



# Implementation

```
class Anode {
public: int nodeType;
        double value;
        char vname; //variable name
        int opType;
};
```



# Implementation

```
template<typename T> class Node {
public:
        Node( ) : left ( nullptr ), right( nullptr )  { }
        T value;
        Node *left, *right; // Node<T> *left, *right. Also good
};
template<typename T>
class BinaryTree {
public:
        BinaryTree( ) { root = 0; }
        Node < T > *root;
        void insert( const T &a ) {  }
        void clear( ) {  }
};
```

```
class Anode {
public: int nodeType;
        double value;
        char vname;
        int opType;
};

BinaryTree<ANode> bt;
```

# Implementation

- nodeType: number, operator, or variable
- value: the number value
- vname: the variable name
- opType: the operator type

```
class Anode {
public: int nodeType;
        double value;
        char vname;
        int opType;
};

BinaryTree<ANode> bt;
```

# Implementation

- nodeType: number, operator, or variable
- value: the number value
- vname: the variable name
- opType: the operator type

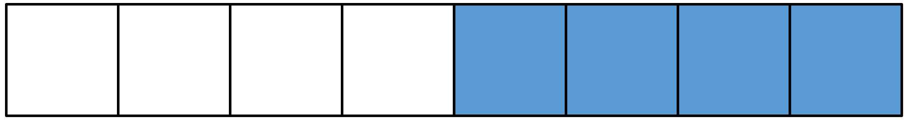
```
class Anode {
public: int nodeType;
        union {
          double value;
          char vname;
          int opType;
        };
};
```

In [computer science](), a **union** is a [value]() that may have any of several representations or formats within the same position in [memory](); or it is a [data structure]() that consists of a [variable]() that may hold such a value.
https://en.wikipedia.org/wiki/Union_type

# Implementation

- nodeType: number, operator, or variable
- value: the number value
- vname: the variable name
- opType: the operator type

```
class Anode {
public: int nodeType;
        union {
          double value;
          char vname;
          int opType;
        };
};
```



In [computer science](), a **union** is a [value]() that may have any of several representations or formats within the same position in [memory](); or it is a [data structure]() that consists of a [variable]() that may hold such a value.

https://en.wikipedia.org/wiki/Union_type