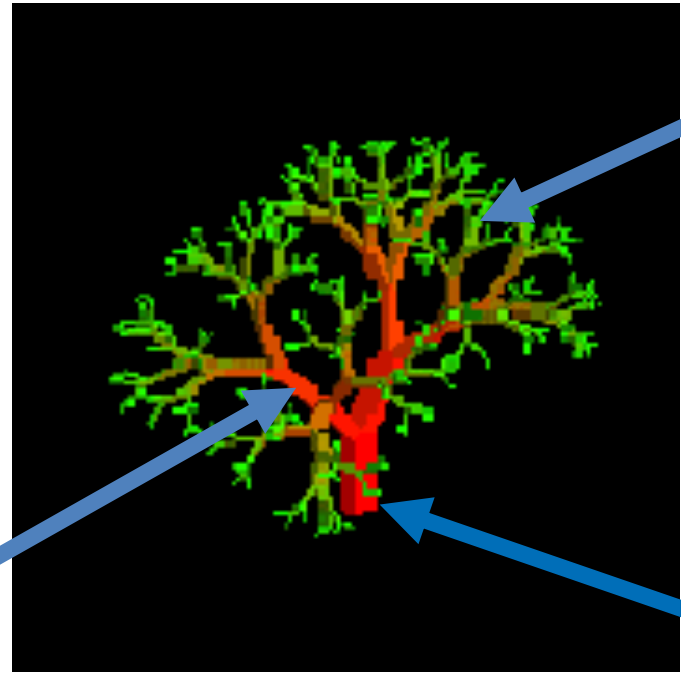
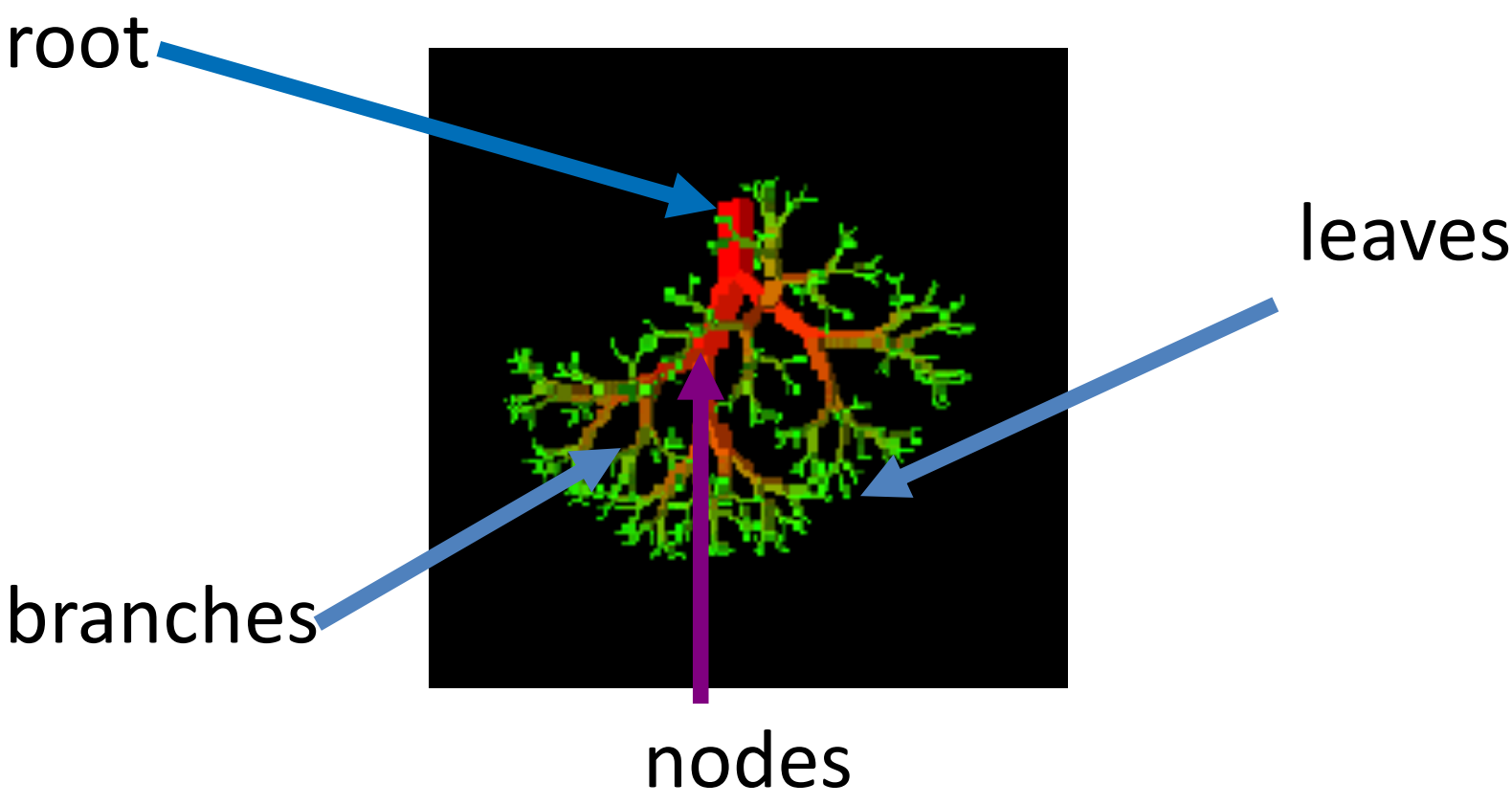
Tree

IIITS

Nature View of a Tree



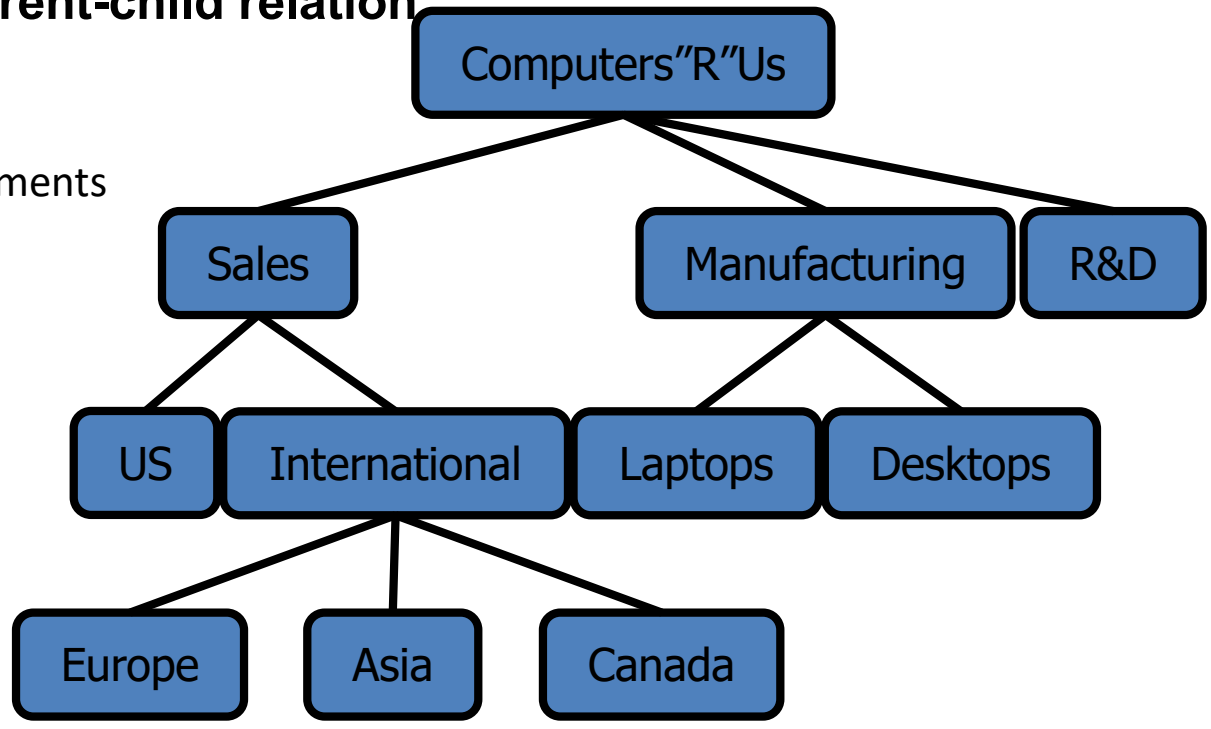
Computer Scientist's View



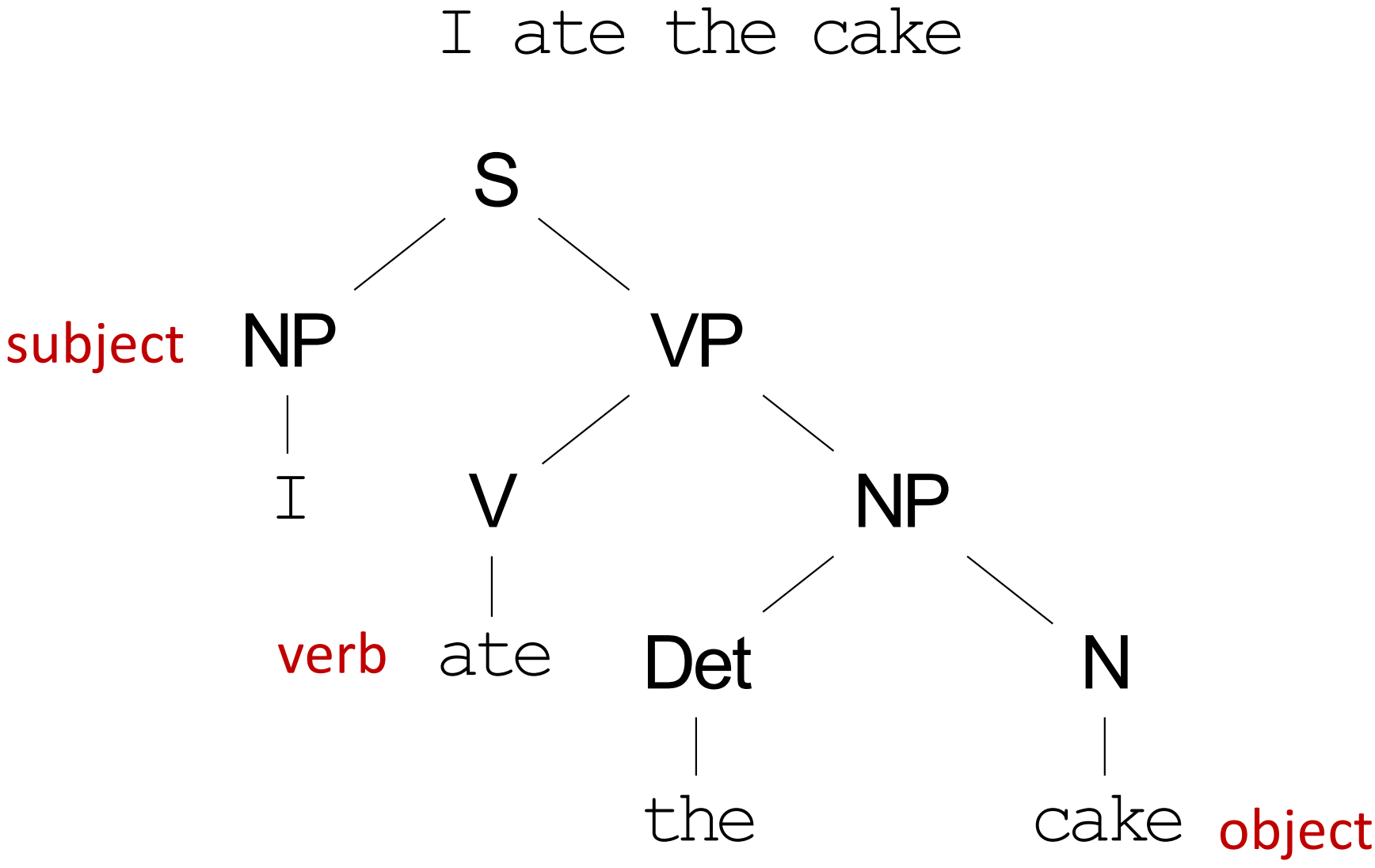
What is a Tree?

A tree is a finite nonempty set of elements.
It is an abstract model of a **hierarchical structure**.
consists of nodes with a **parent-child relation**.
Applications:

- Organization charts
- File systems
- Programming environments

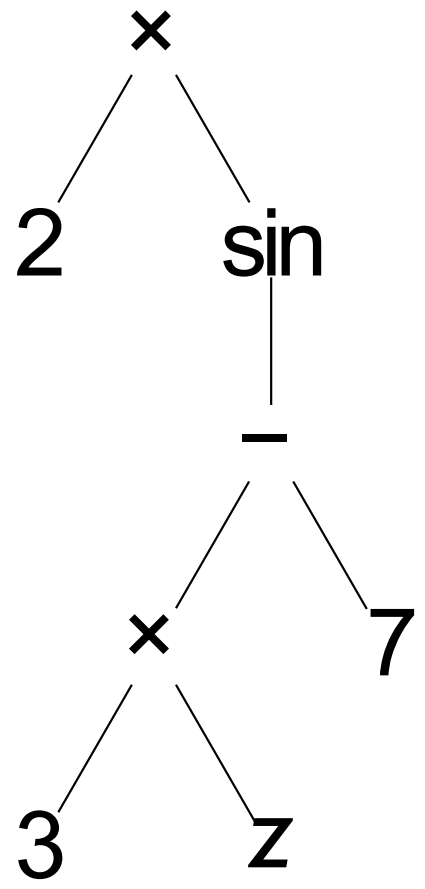


Syntax Tree for a Sentence

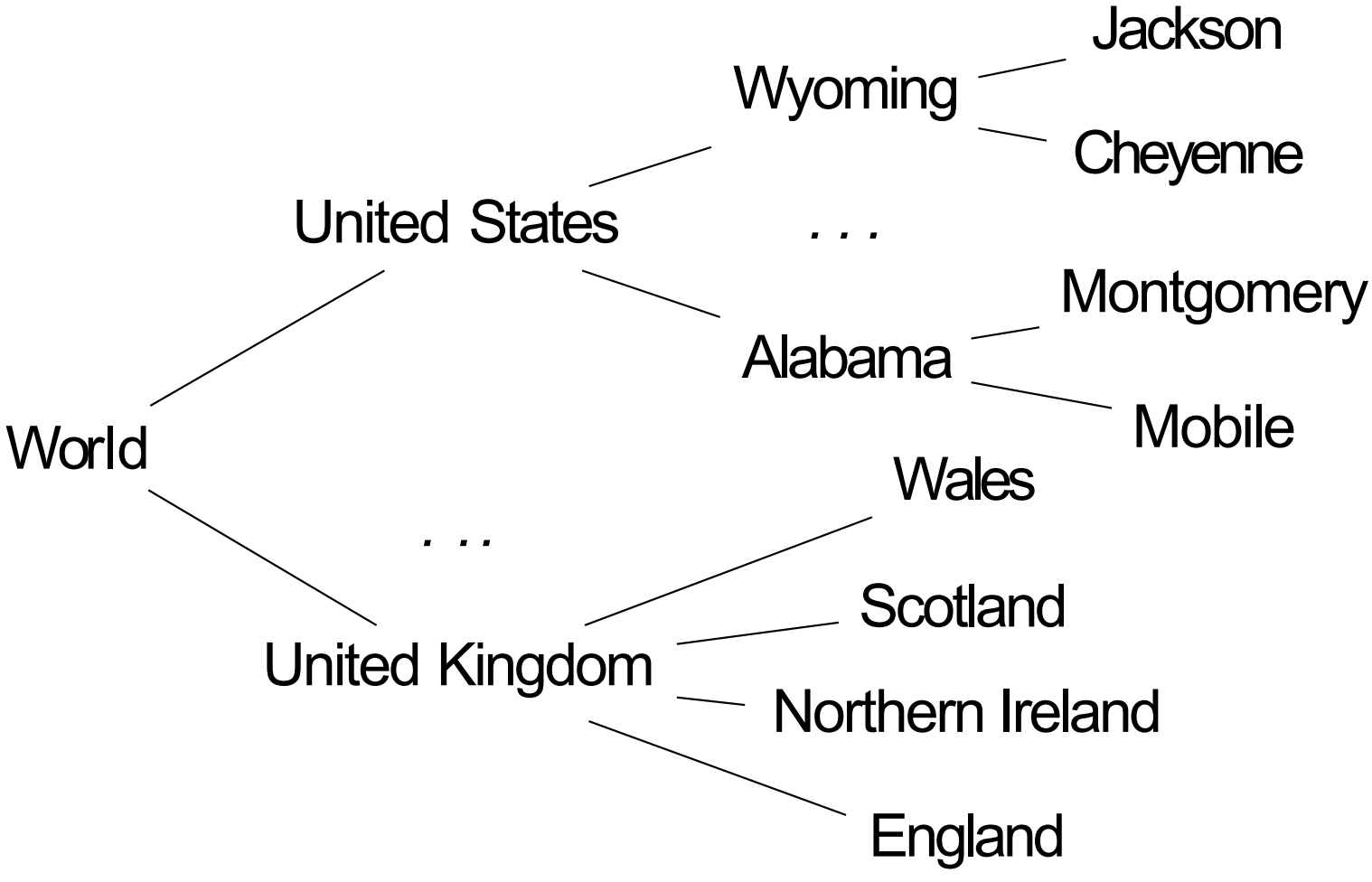


Syntax tree for an Expression

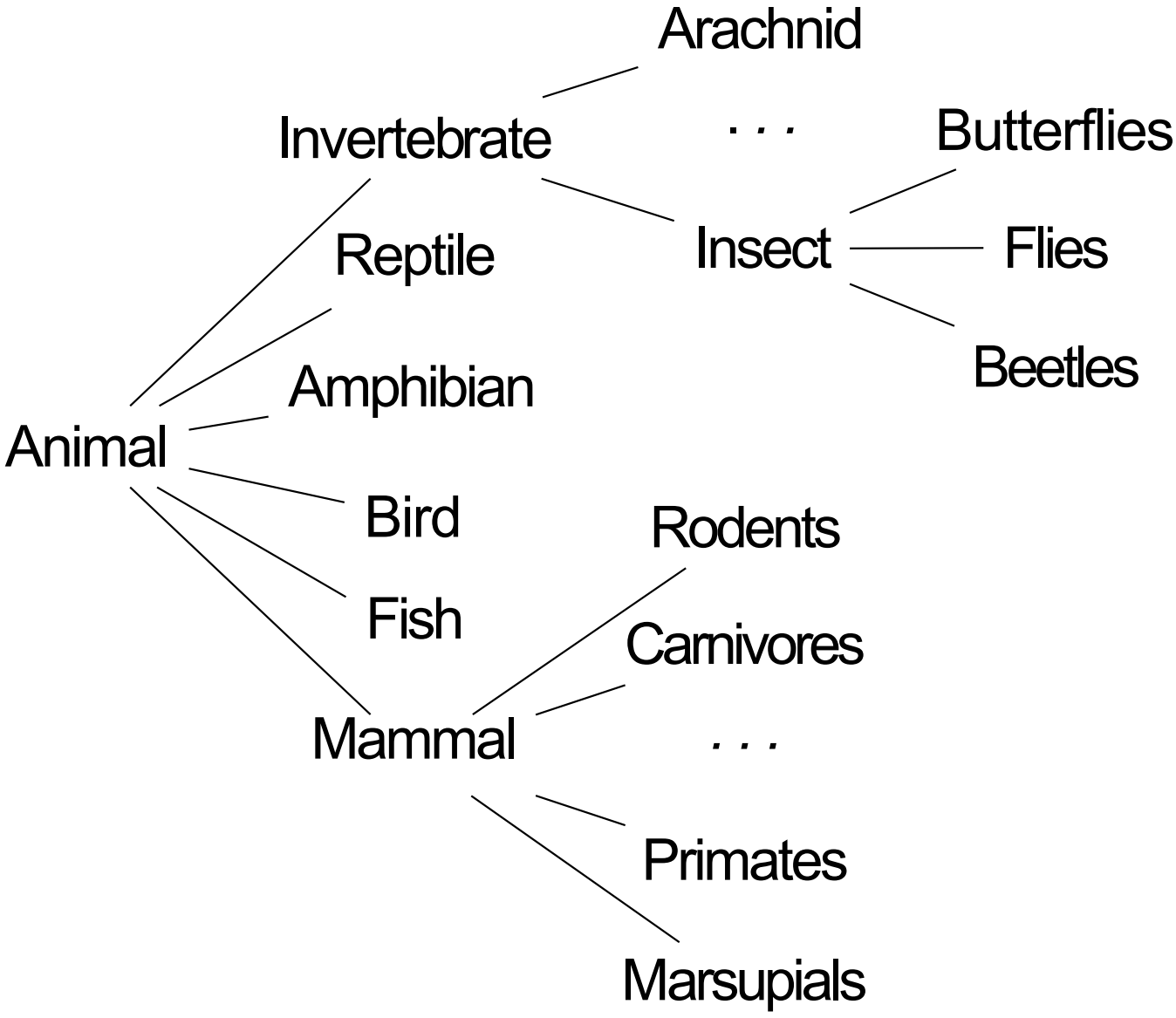
$$2\sin(3z - 7)$$



Geography Hierarchy

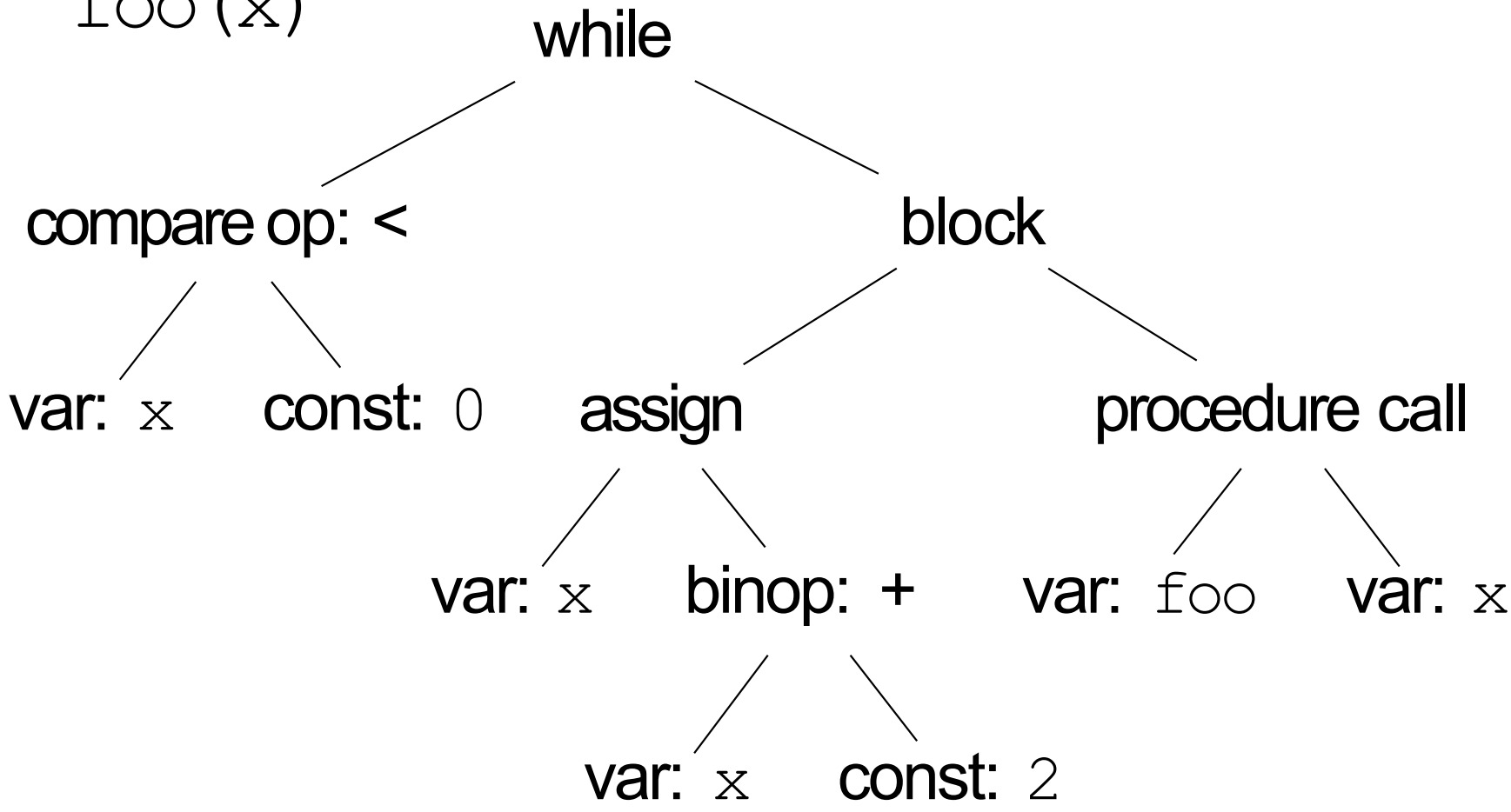


Animal Kingdom



Abstract Syntax Tree for Code

```
while x < 0:
  x = x + 2
  foo(x)
```



Definition – A recursive definition

A Tree is:

- empty, or
- a node with:
  - a key, and
  - a list of child trees.

Simple Tree

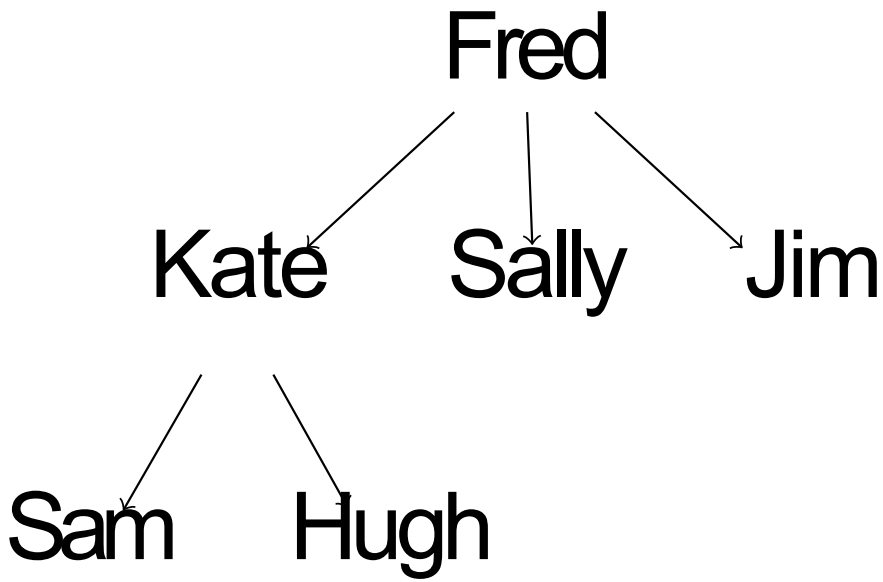
Empty tree:

Tree with one node:

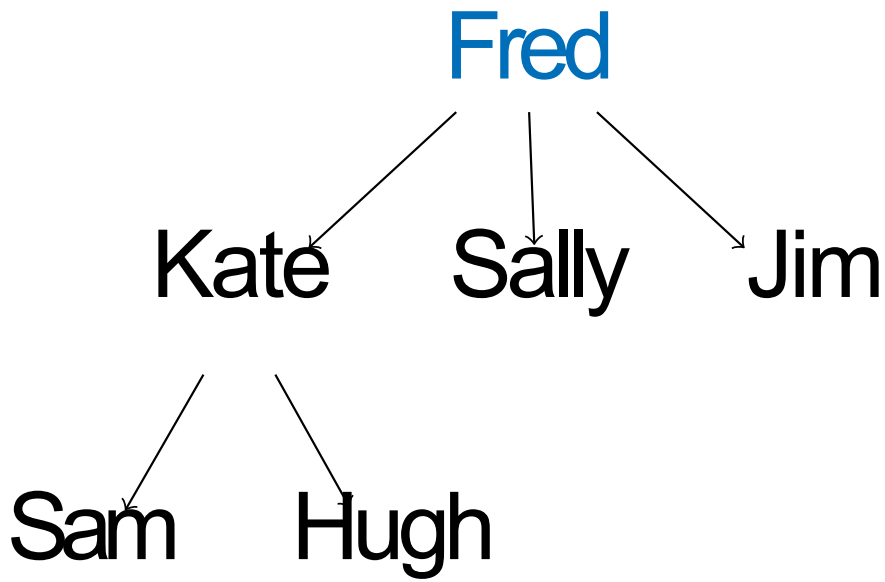
Fred

Tree with two nodes:

Terminology



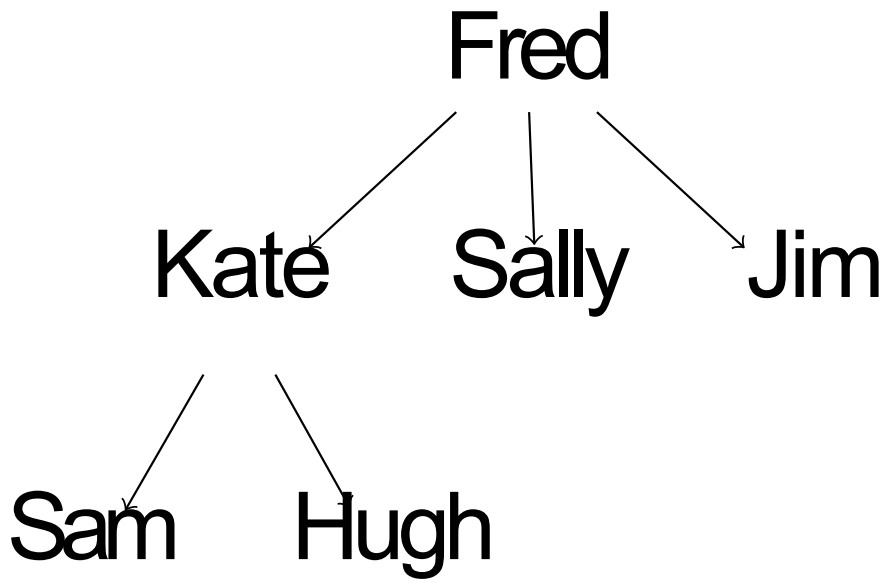
Terminology



***Root*:**

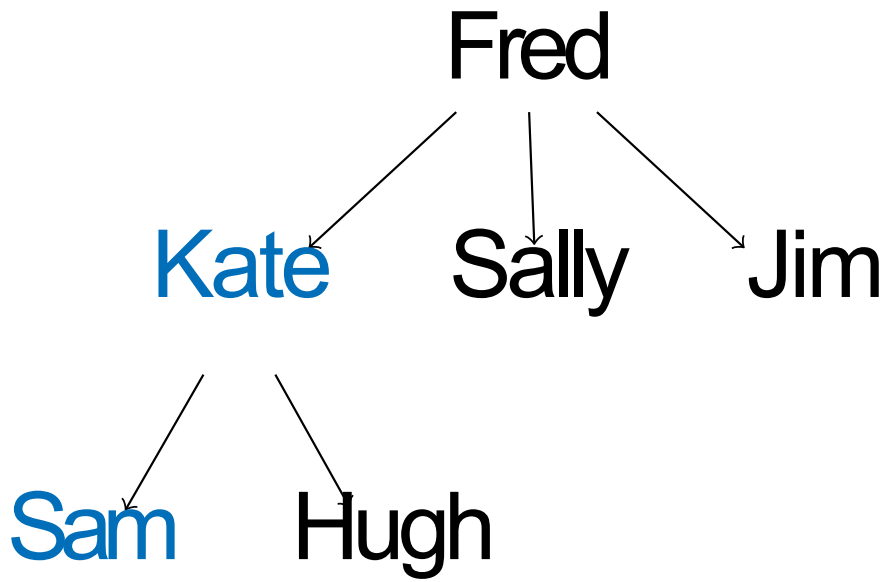
top node in the tree

Terminology



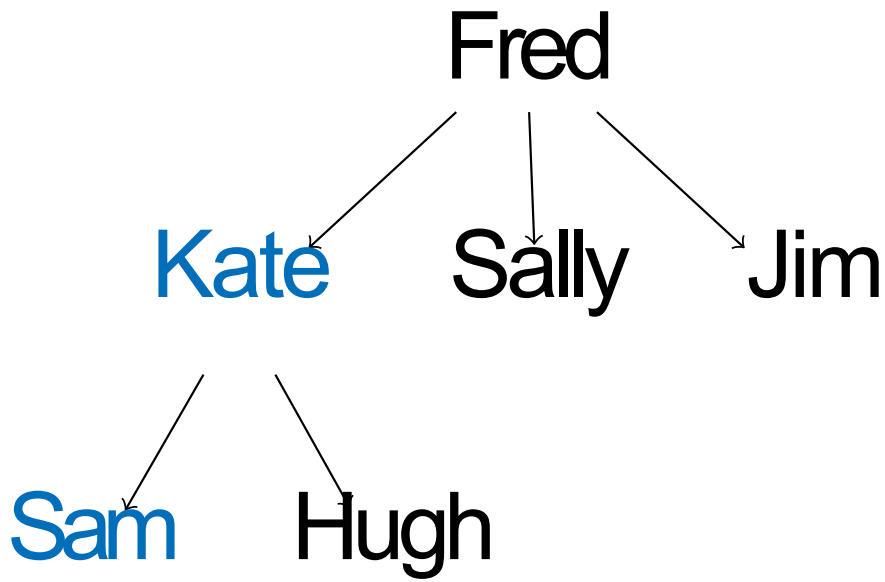
A *child* has a line down directly from a *parent*

Terminology



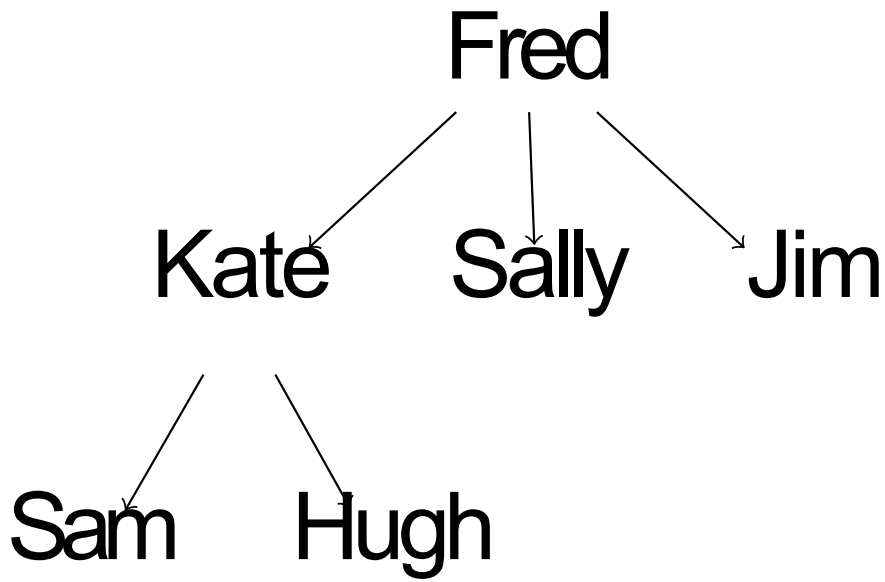
Kate is a *parent* of Sam

Terminology



Sam is a *child* of Kate

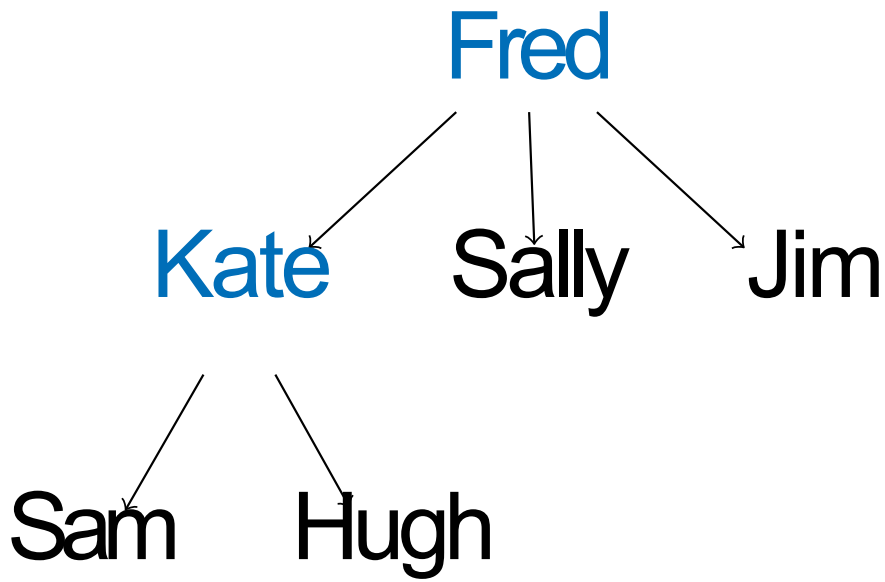
Terminology



***Ancestor*:**

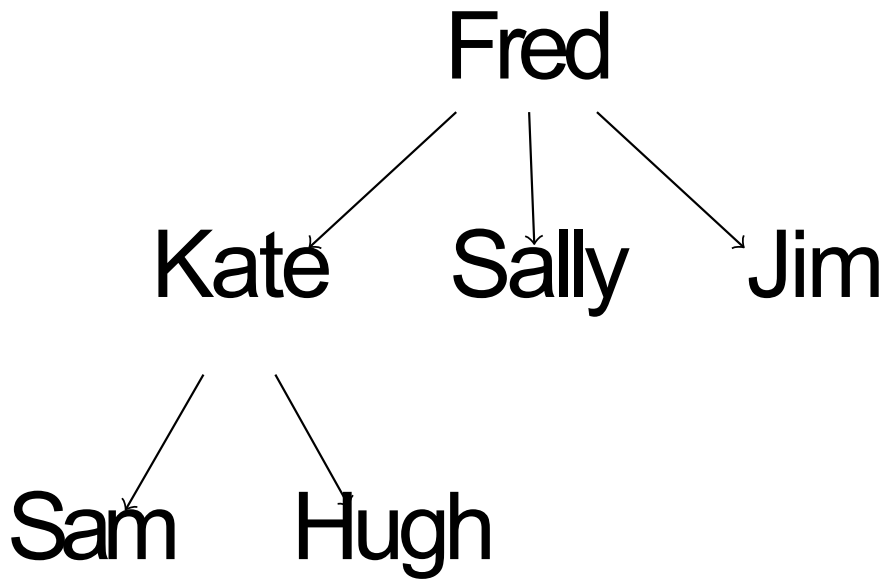
parent, or parent of parent, etc.

Terminology



***Ancestors* of Sam**

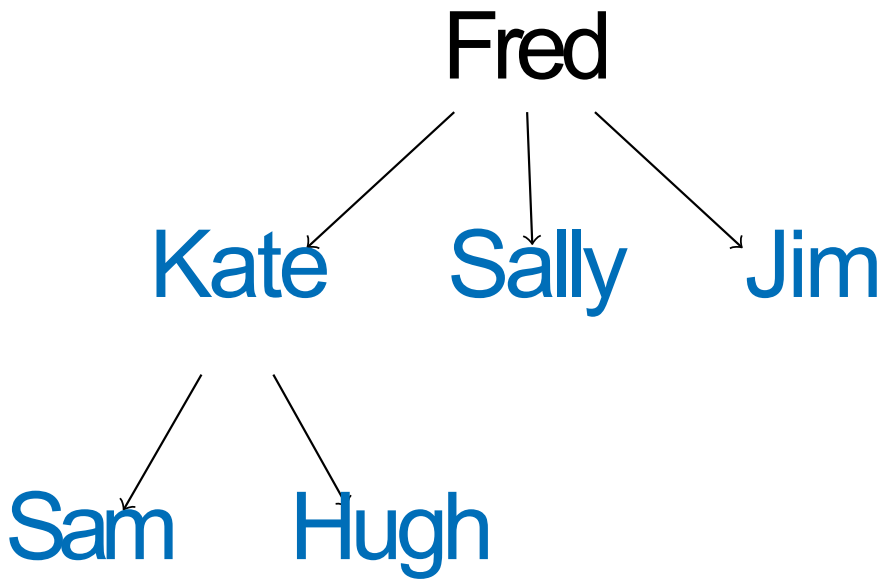
Terminology



***Descendant*:**

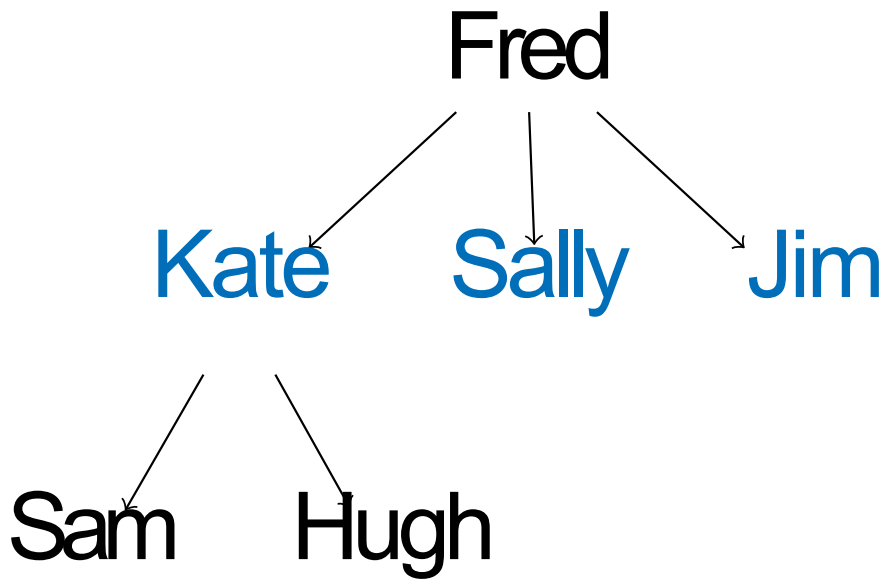
child, or child of child, etc.

Terminology



***Descendants* of Fred**

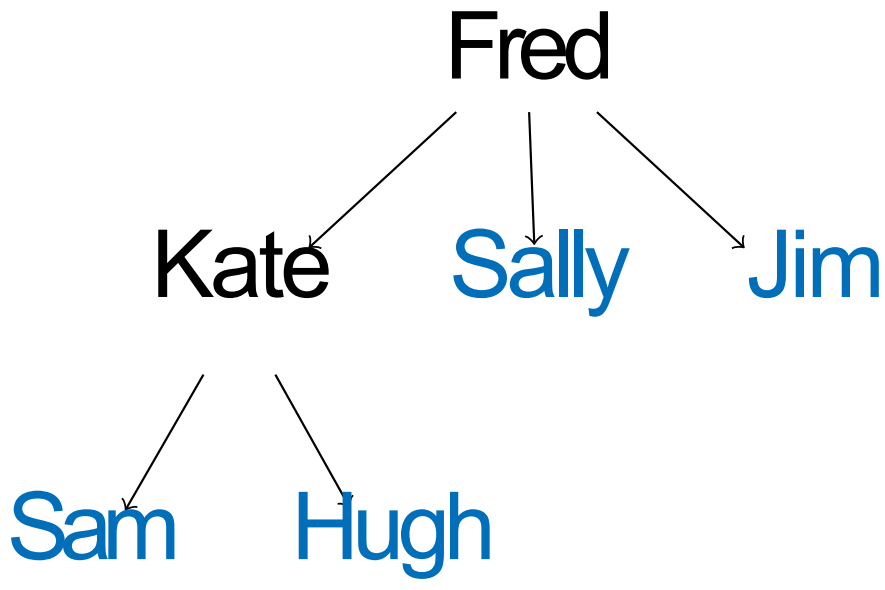
Terminology



Sibling **:**

sharing the same parent

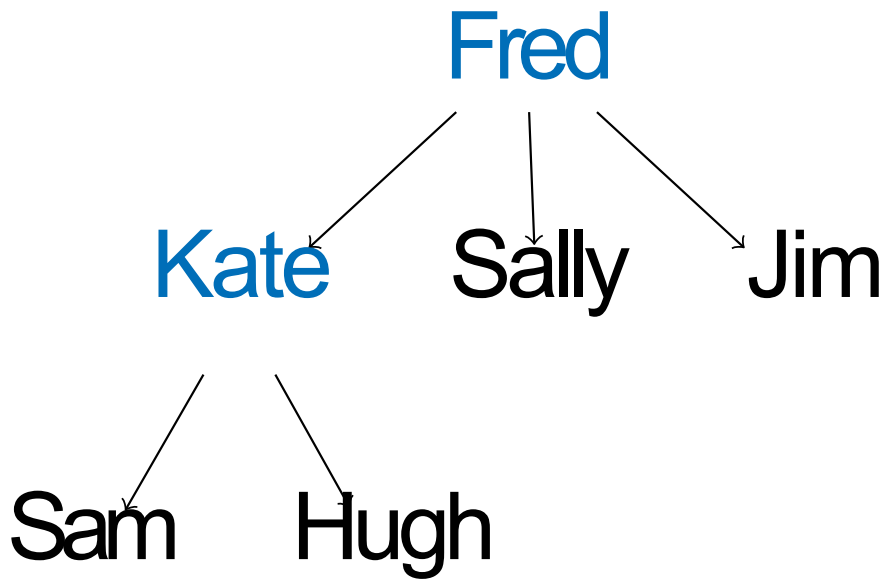
Terminology



Leaf **:**

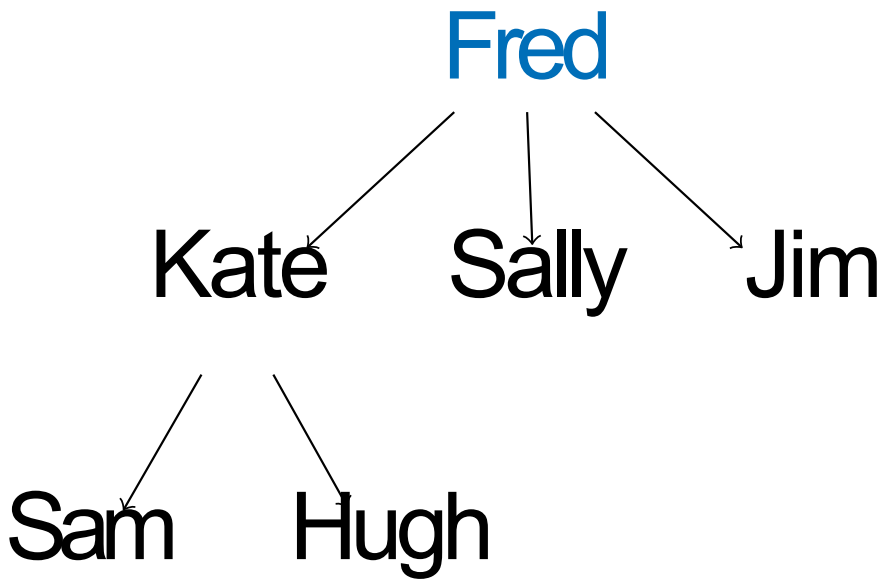
node with no children

Terminology



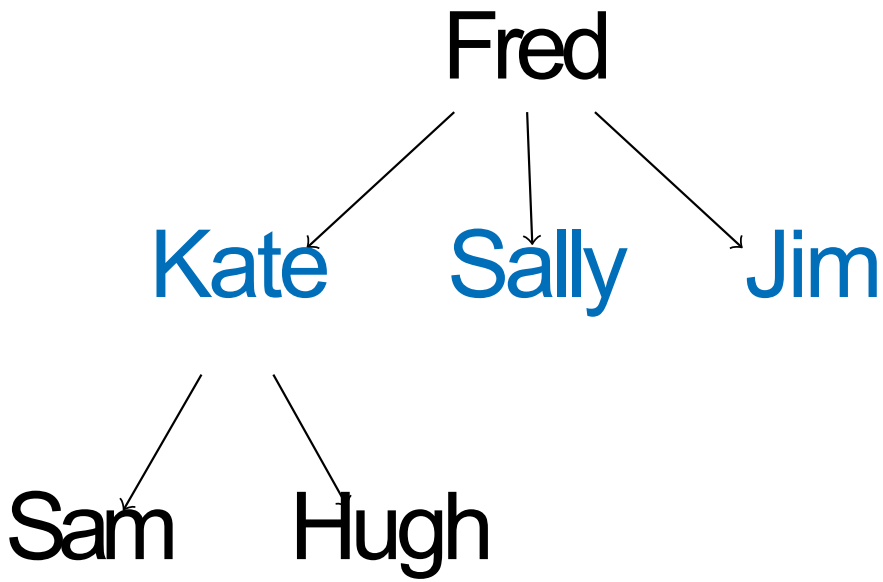
Interior node (non-leaf)

Terminology



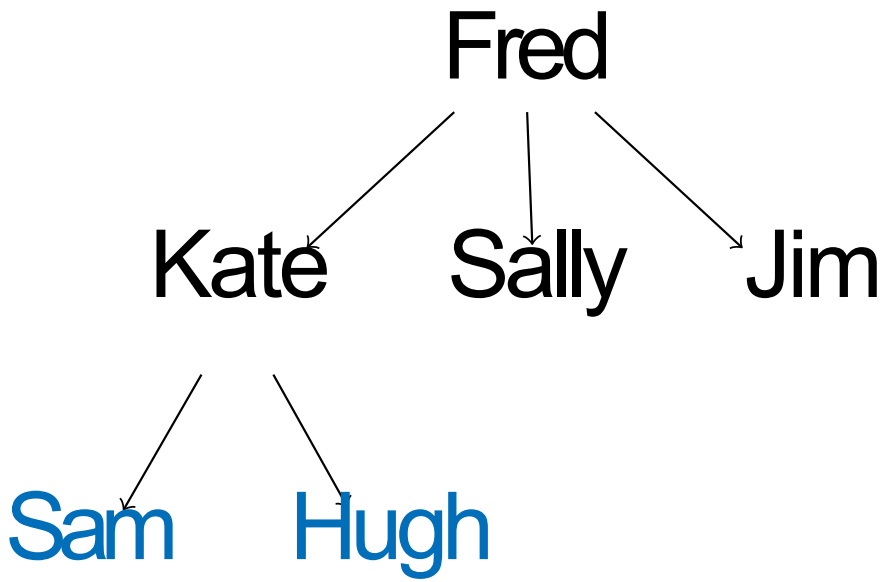
Level 1

Terminology



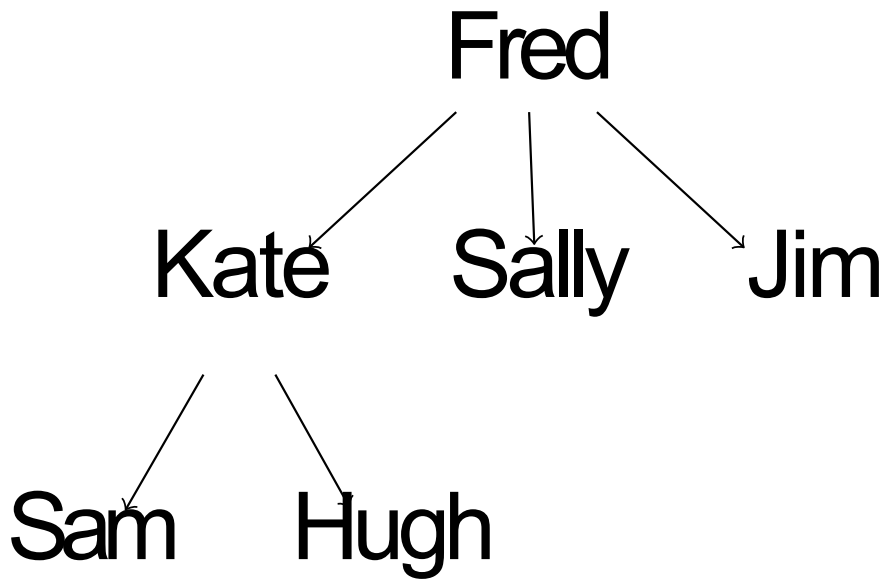
Level 2

Terminology



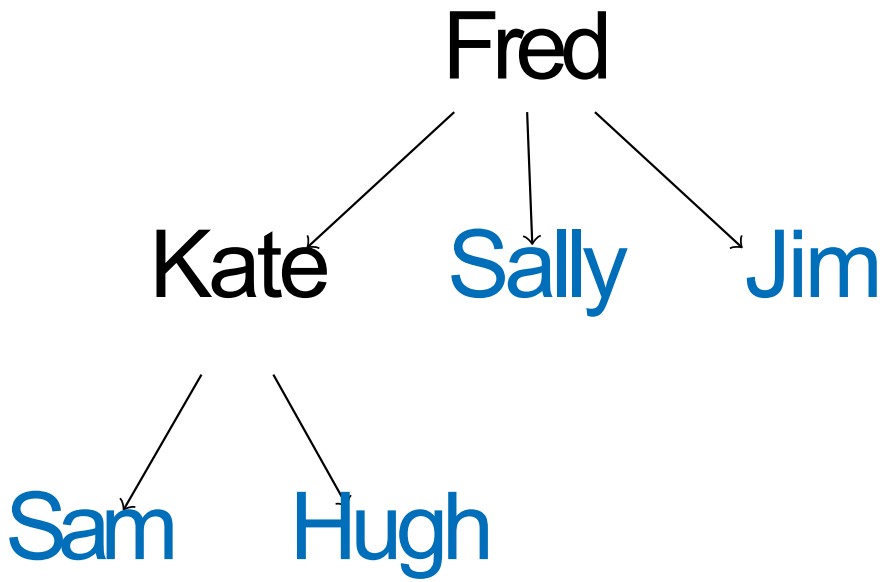
Level 3

Terminology



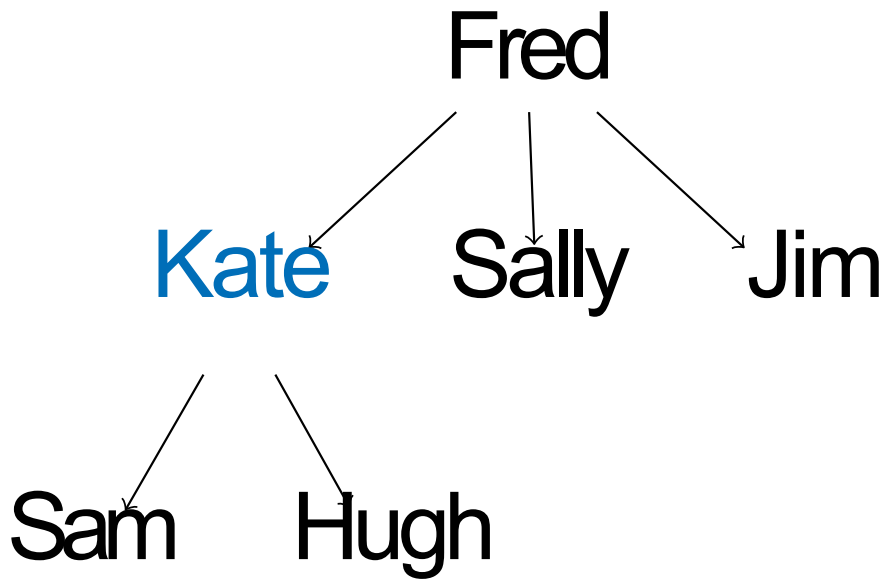
***Height*: maximum depth of subtree node and farthest leaf**

Terminology



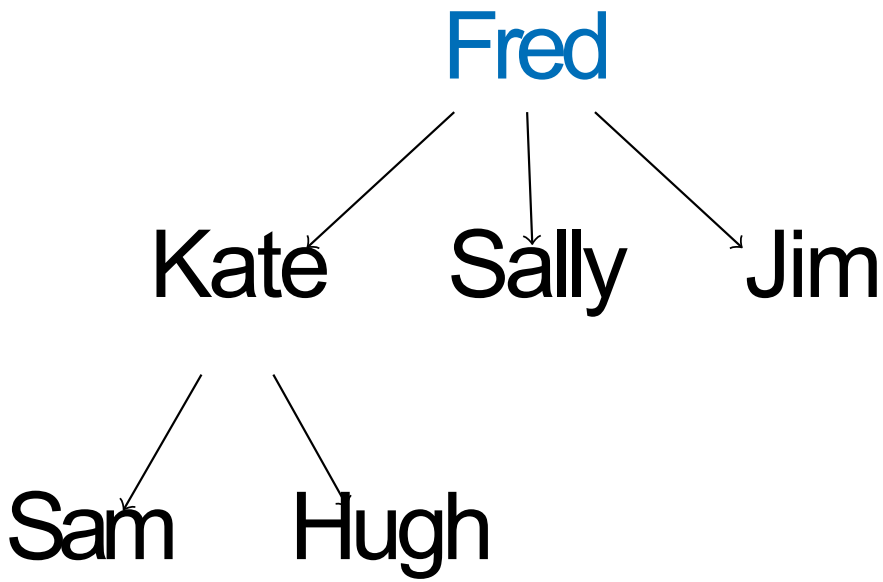
Height 1

Terminology



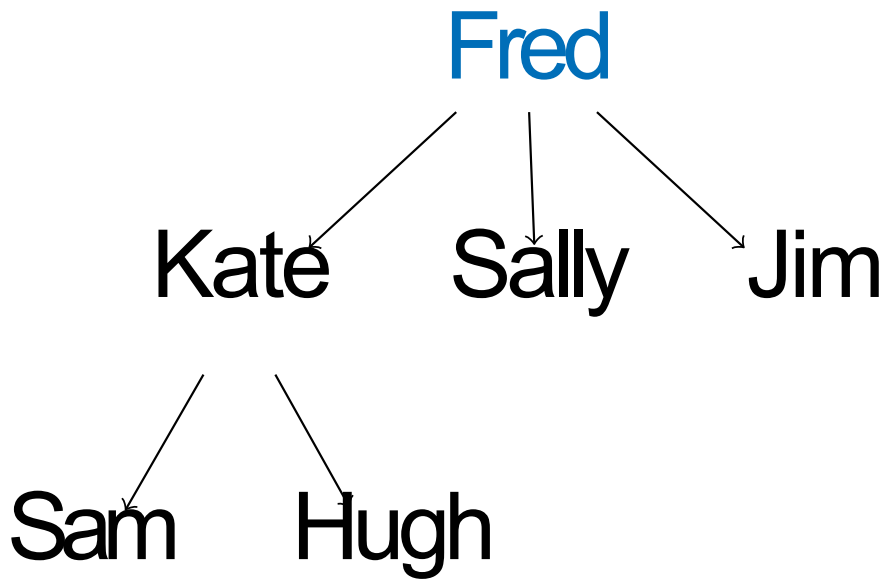
Height 2

Terminology



Height 3

Terminology



Height of the tree is 3

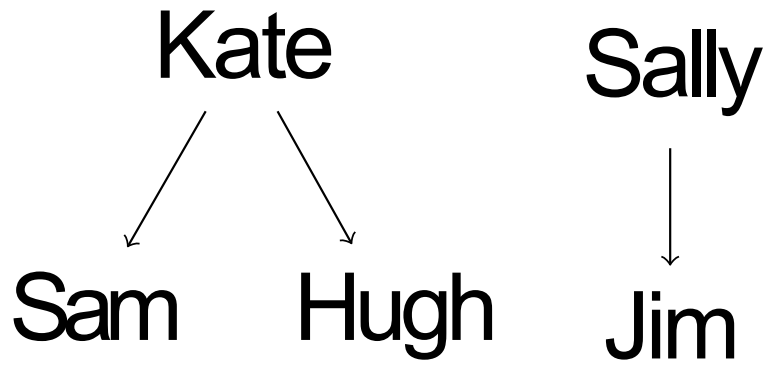
Height(*tree*)

```
if tree = nil:
   return 0
return 1 + Max(Height(tree.left),
                Height(tree.right))
```

Size(*tree*)

```
if tree = nil
   return 0
return 1 + Size(tree.left) +
           Size(tree.right)
```

Terminology

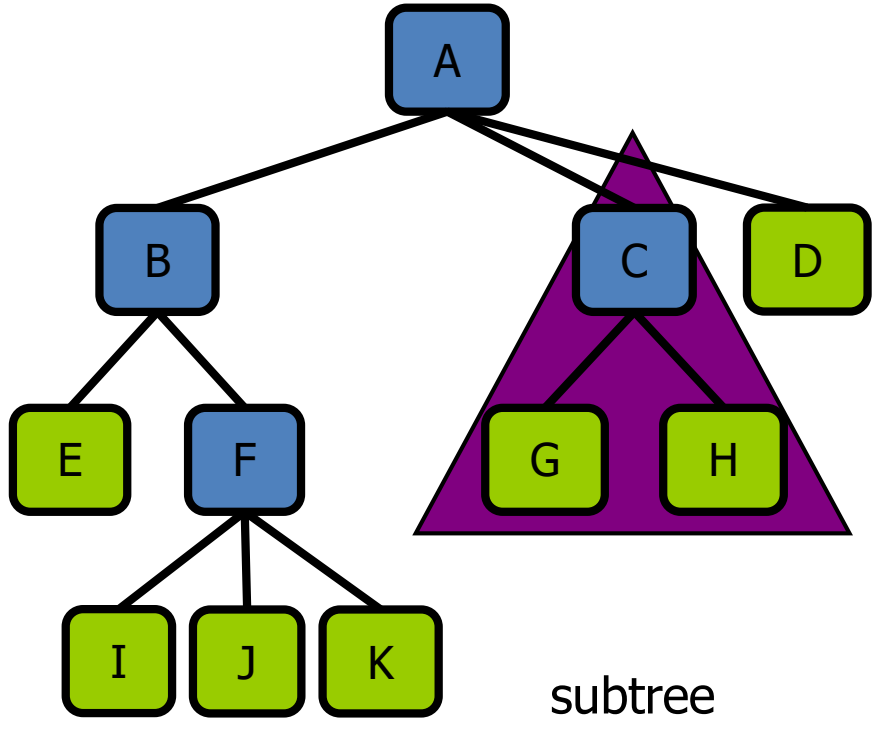


***Forest*:**

collection of trees

Tree Terminologies - Summary

- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (3)
- **Degree** of a node: the number of its children
- **Subtree**: tree consisting of a node and its descendants



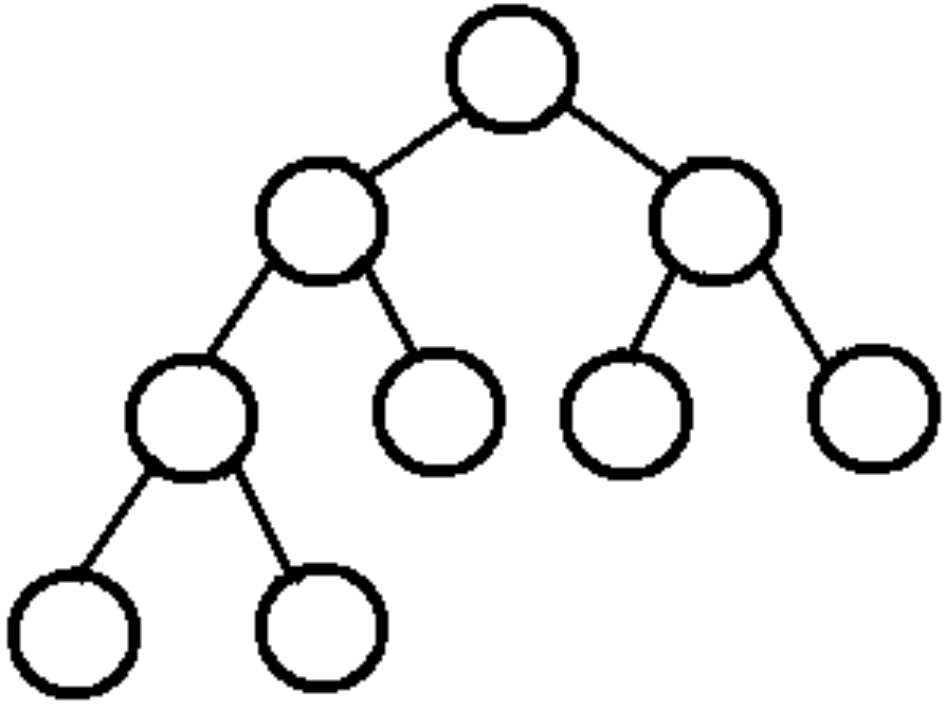
Binary Tree

For binary tree, node contains:

- `key`
- `left`
- `right`
- (optional) `parent`

Complete Binary Tree

A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

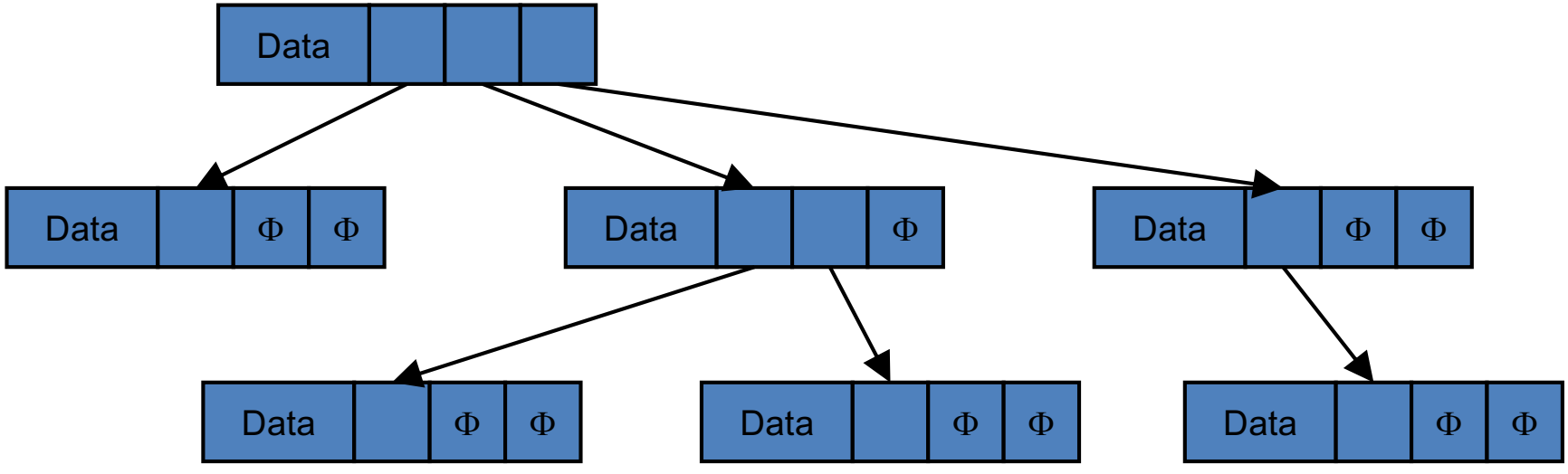


Trees - ADT

Every tree node:

object – useful information or data

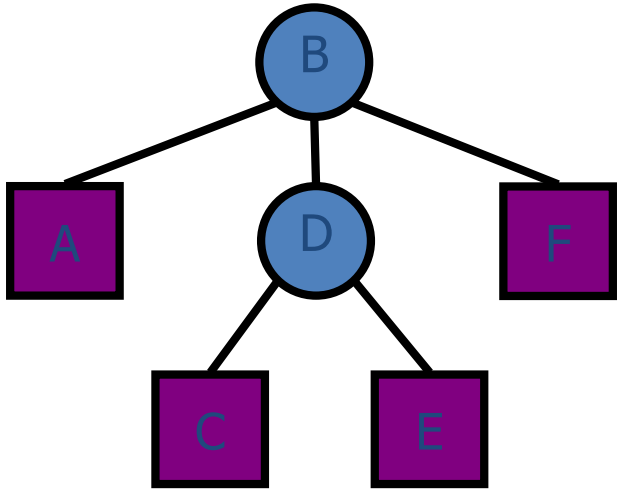
children – pointers to its children

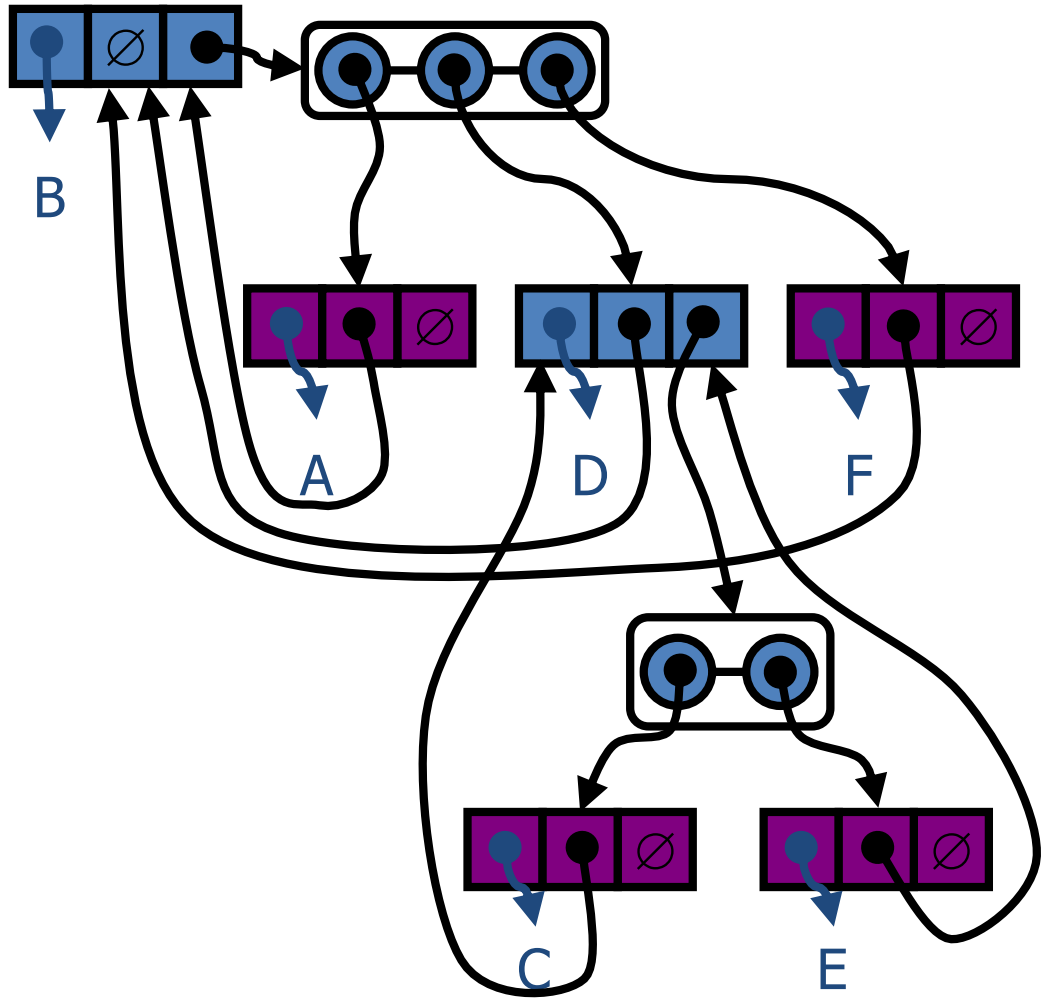


A Tree Representation

A node is represented by an object storing

- Element
- Parent node
- Sequence of children nodes





Assignment

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* newNode() allocates a new node with the given data and NULL left and
   right pointers. */
struct node* newNode(int data)
{
  // Allocate memory for new node
  struct node* node = (struct node*)malloc(sizeof(struct node));

  // Assign data to this node
  node->data = data;

  // Initialize left and right children as NULL
  node->left = NULL;
  node->right = NULL;
  return(node);
}
```

Assignment

```
int main()
{
  /*create root*/
  struct node *root = newNode(1);
  /* following is the tree after above statement

        1
      /   \
    NULL  NULL
  */


  root->left        = newNode(2);
  root->right       = newNode(3);
  /* 2 and 3 become left and right children of 1
           1
         /   \
        2     3
      /   \   /  \
    NULL NULL NULL NULL
  */


  root->left->left  = newNode(4);
  /* 4 becomes left child of 2
           1
         /       \
        2         3
      /   \      /  \
     4    NULL  NULL  NULL
   /  \
NULL NULL
*/

  getchar();
  return 0;
}
```

Walking a Tree

Often we want to visit the nodes of a tree in a particular order.

For example, print the nodes of the tree.

Walking a Tree

Often we want to visit the nodes of a tree in a particular order.

For example, print the nodes of the tree.

- Depth-first: We completely traverse one sub-tree before exploring a sibling sub-tree.

Walking a Tree

Often we want to visit the nodes of a tree in a particular order.

For example, print the nodes of the tree.

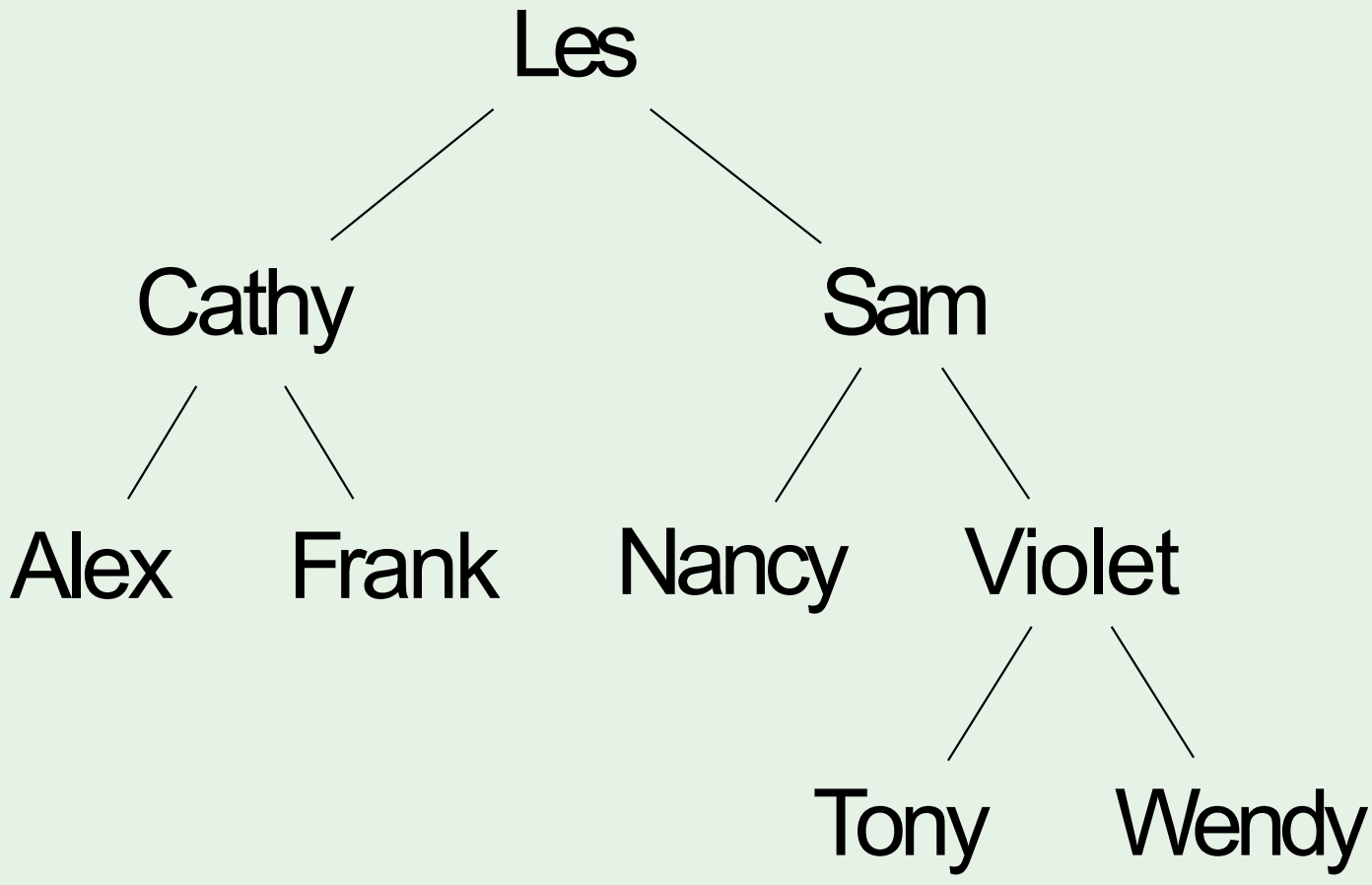
- Depth-first: We completely traverse one sub-tree before exploring a sibling sub-tree.
- Breadth-first: We traverse all nodes at one level before progressing to the next level.

Depth-first

InOrderTraversal(*tree*)

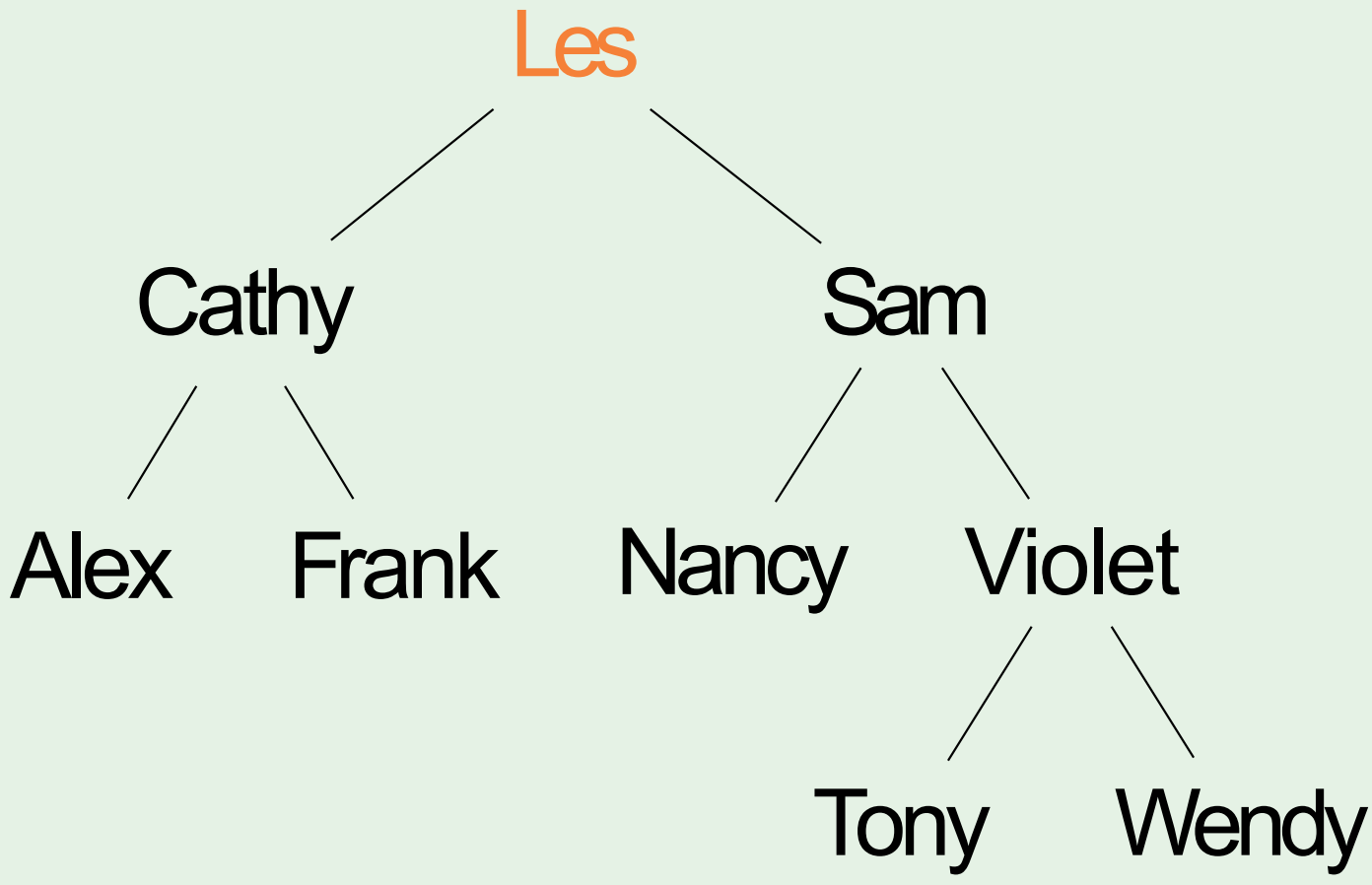
```
if tree = nil:
   return
InOrderTraversal(tree.left)
Print(tree.key)
InOrderTraversal(tree.right)
```

InOrderTraversal



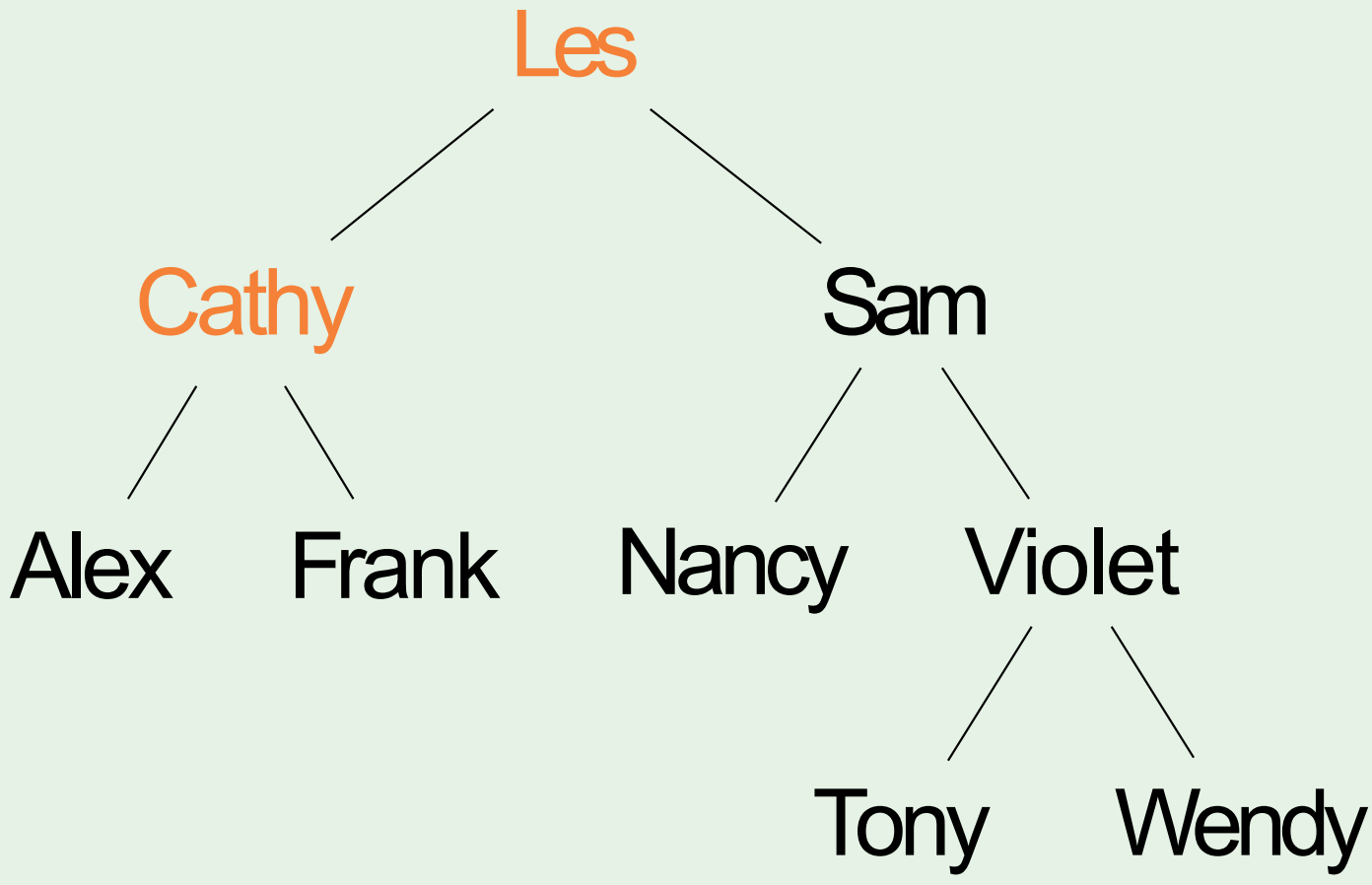
Output:

InOrderTraversal



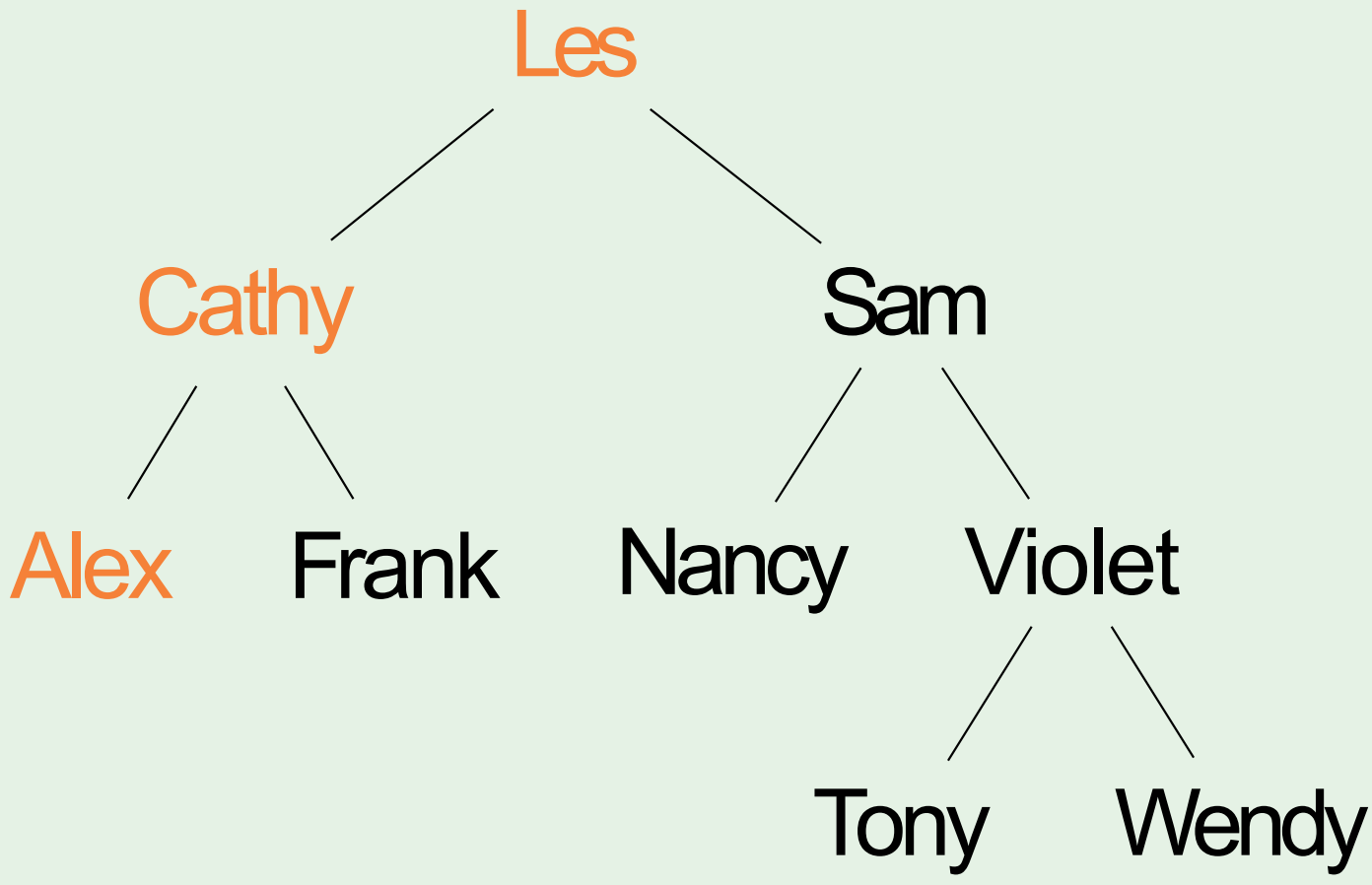
Output:

InOrderTraversal



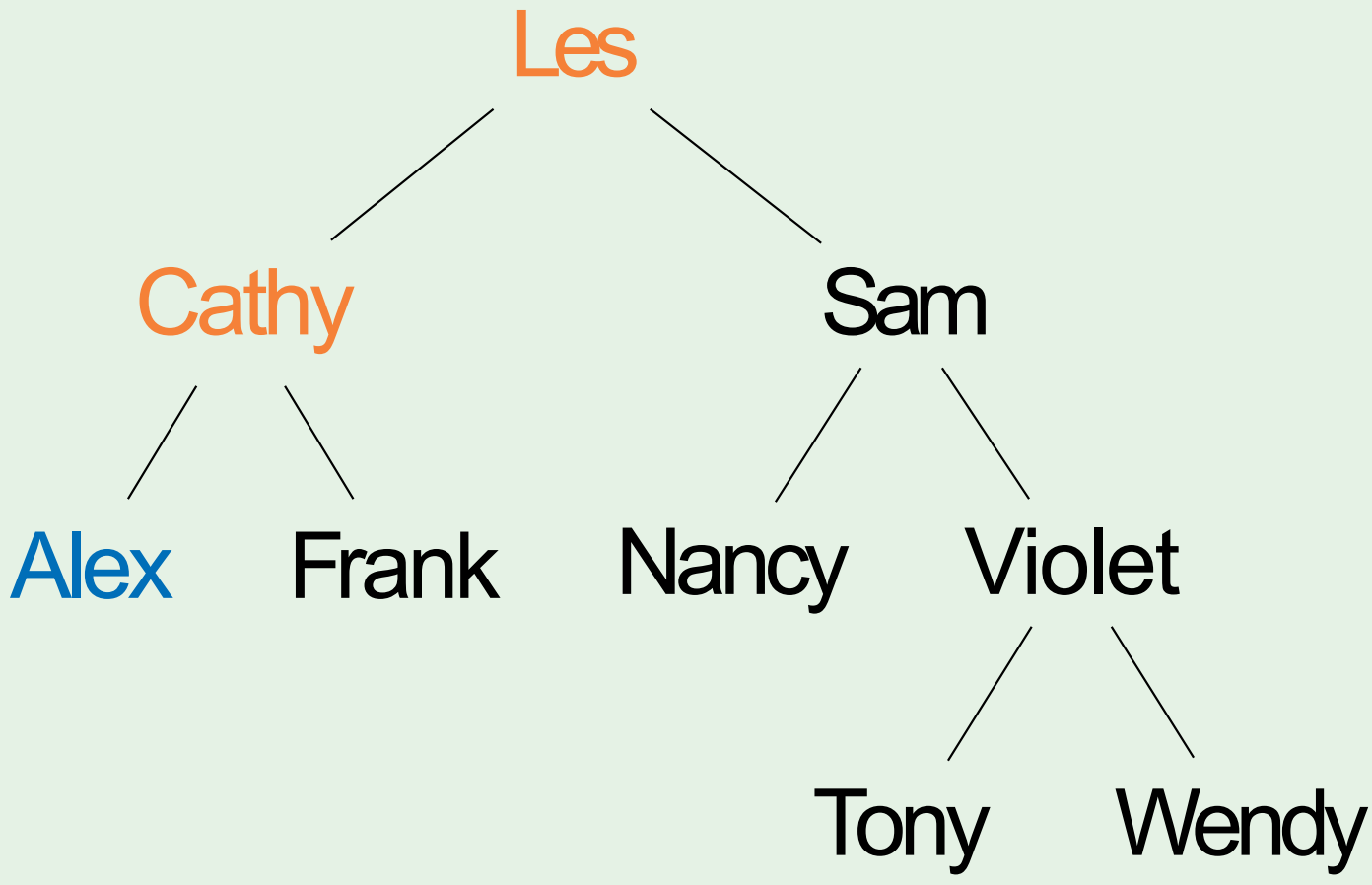
Output:

InOrderTraversal



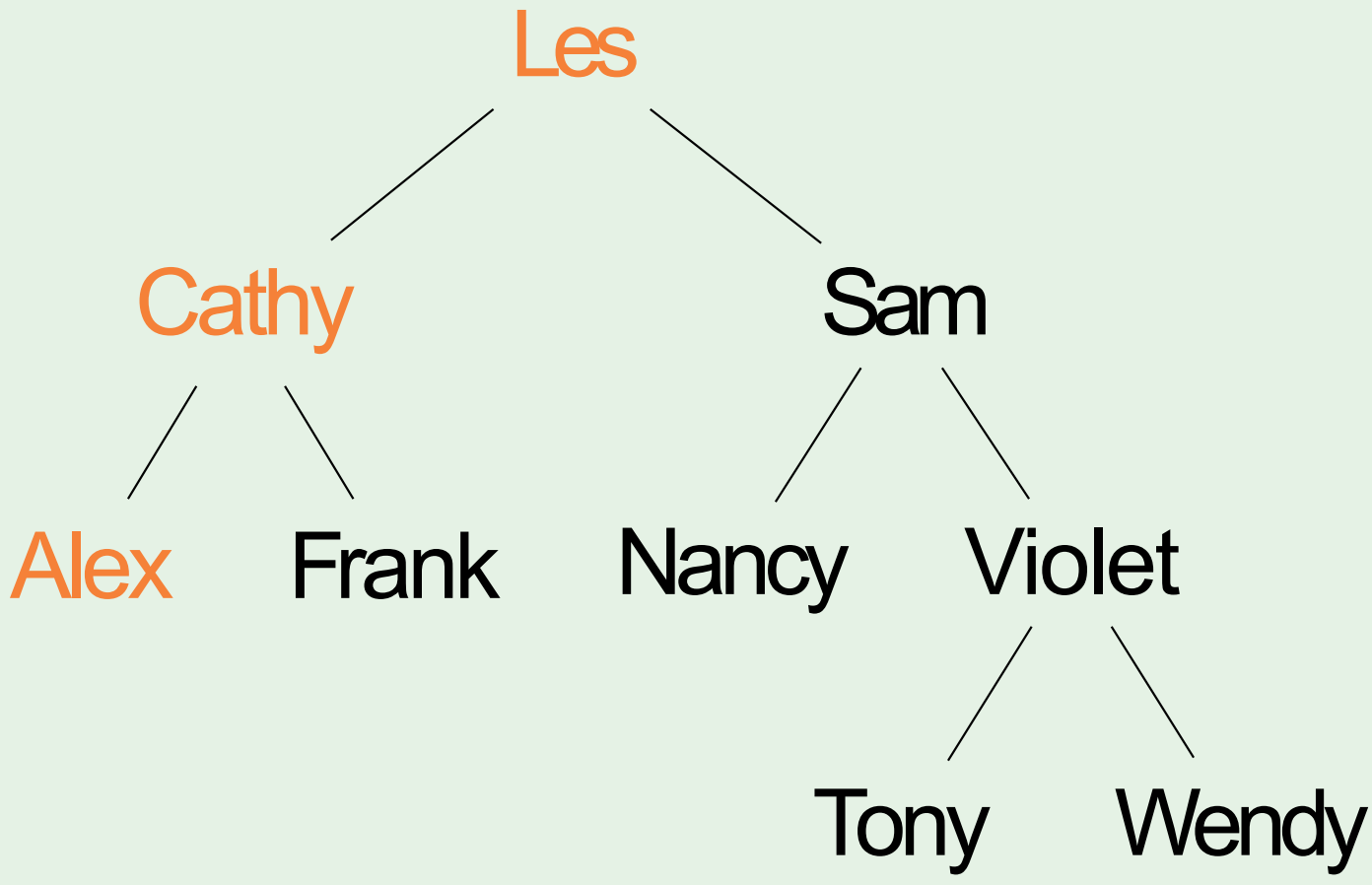
Output:

InOrderTraversal



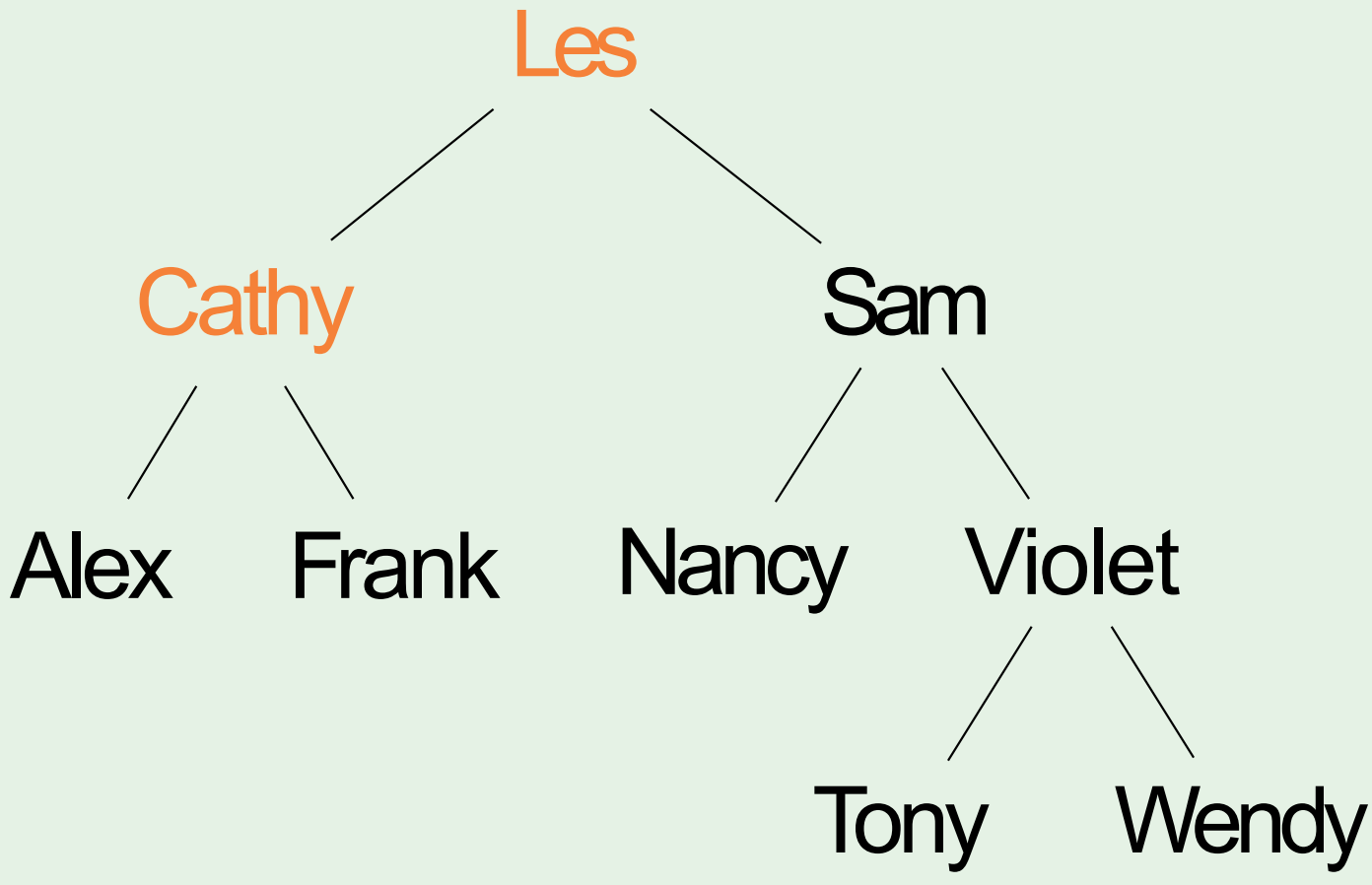
Output: Alex

InOrderTraversal



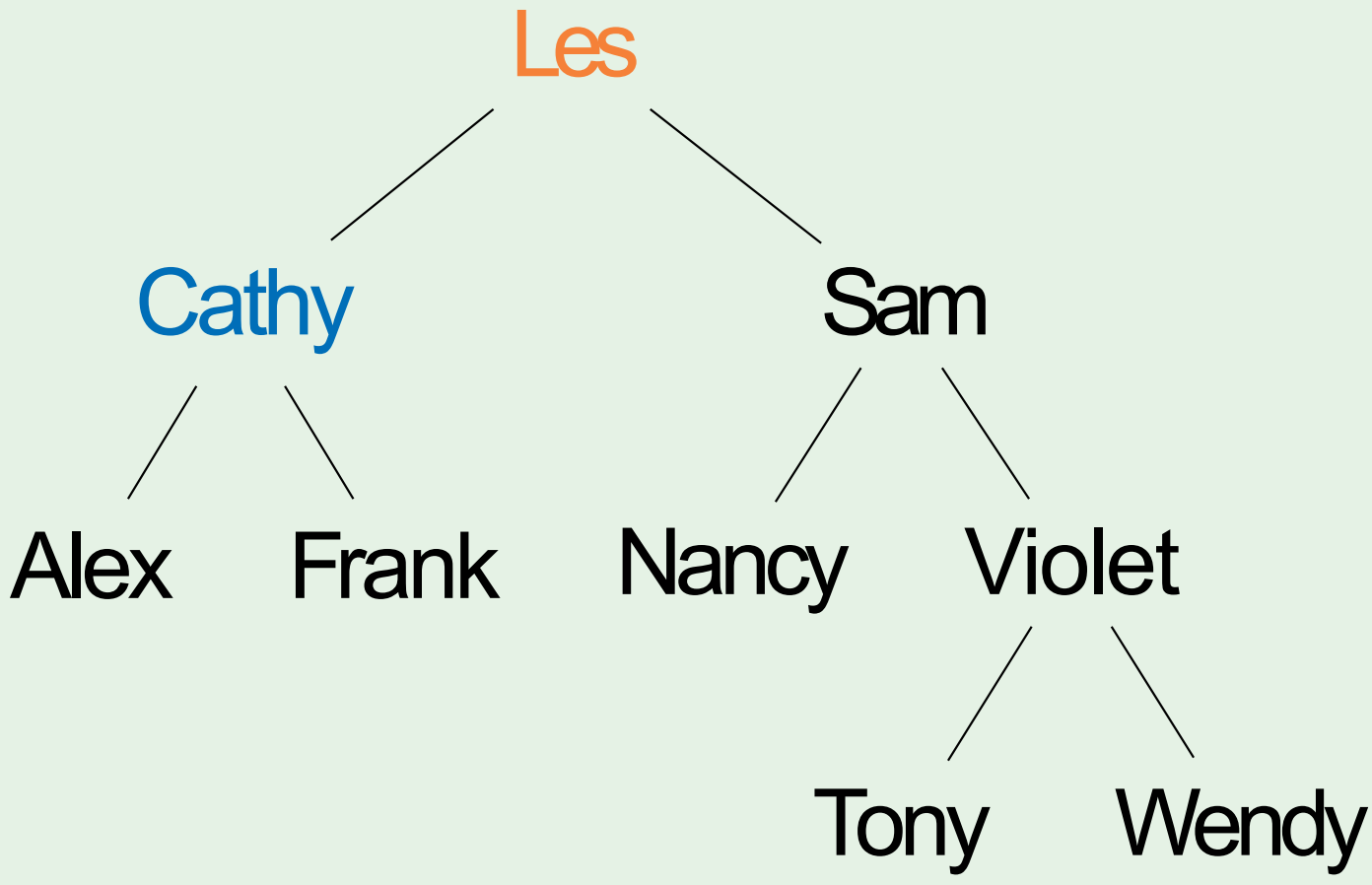
Output: Alex

InOrderTraversal



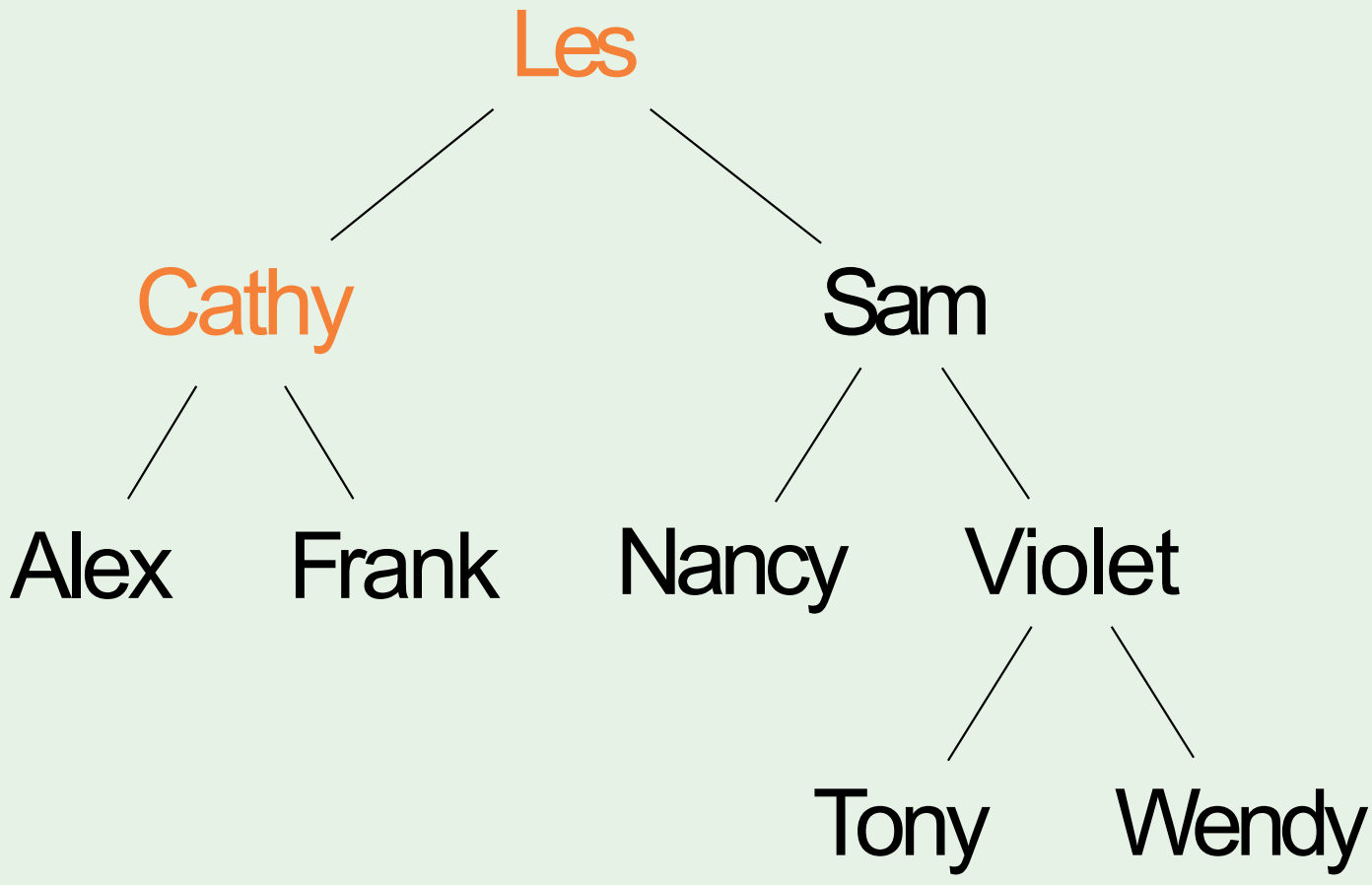
Output: Alex

InOrderTraversal



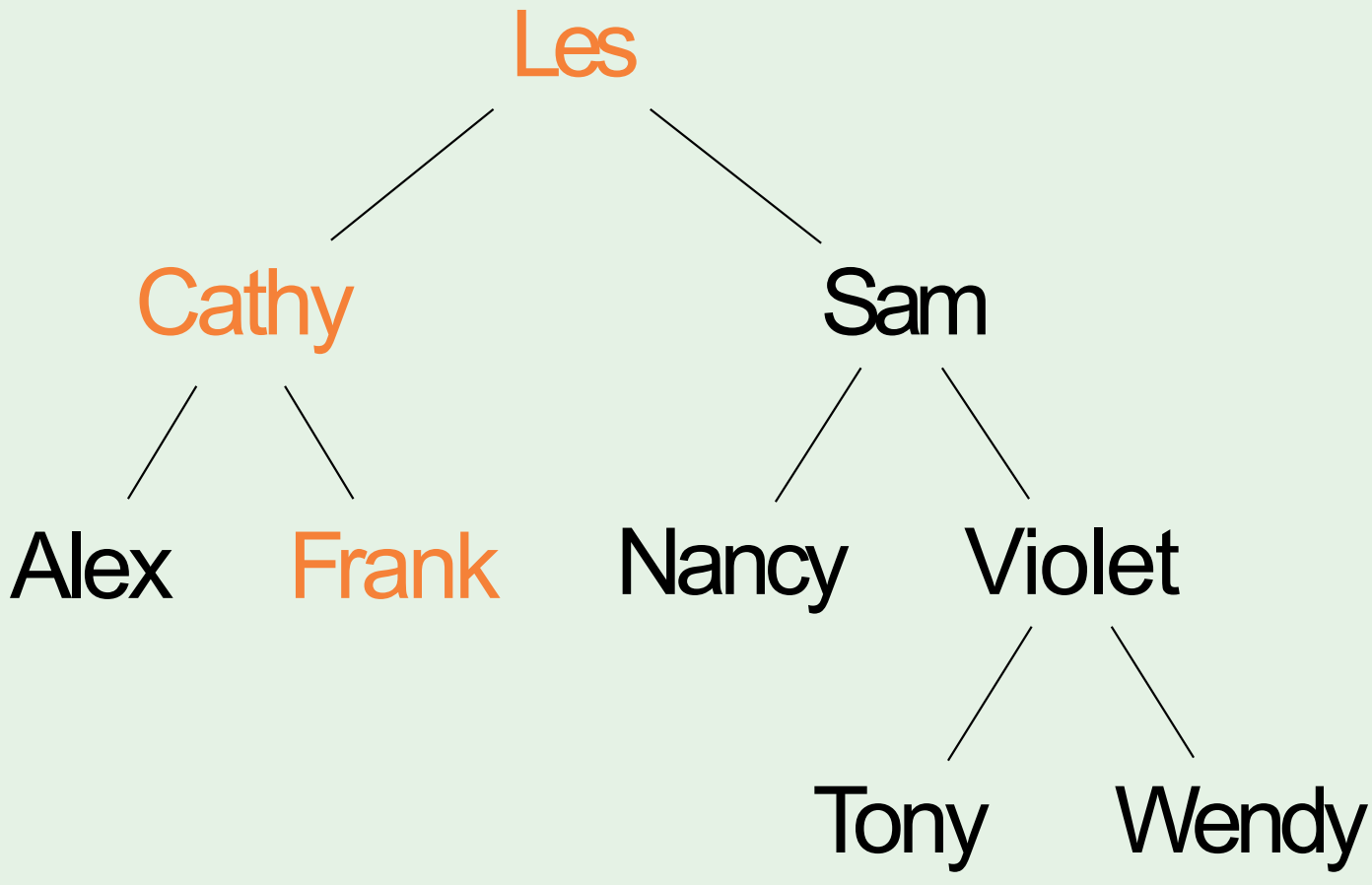
Output: Alex Cathy

InOrderTraversal



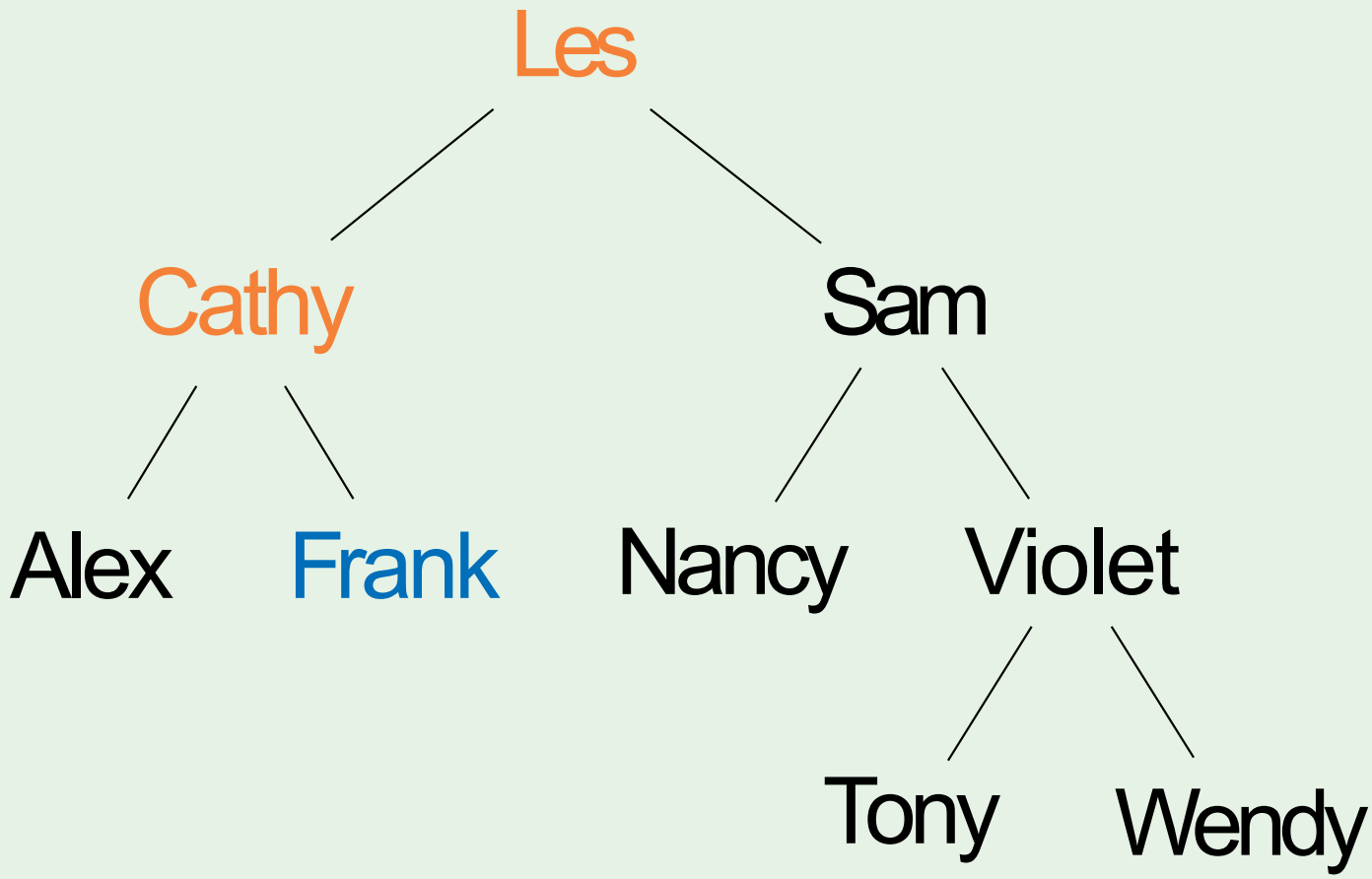
Output: Alex Cathy

InOrderTraversal



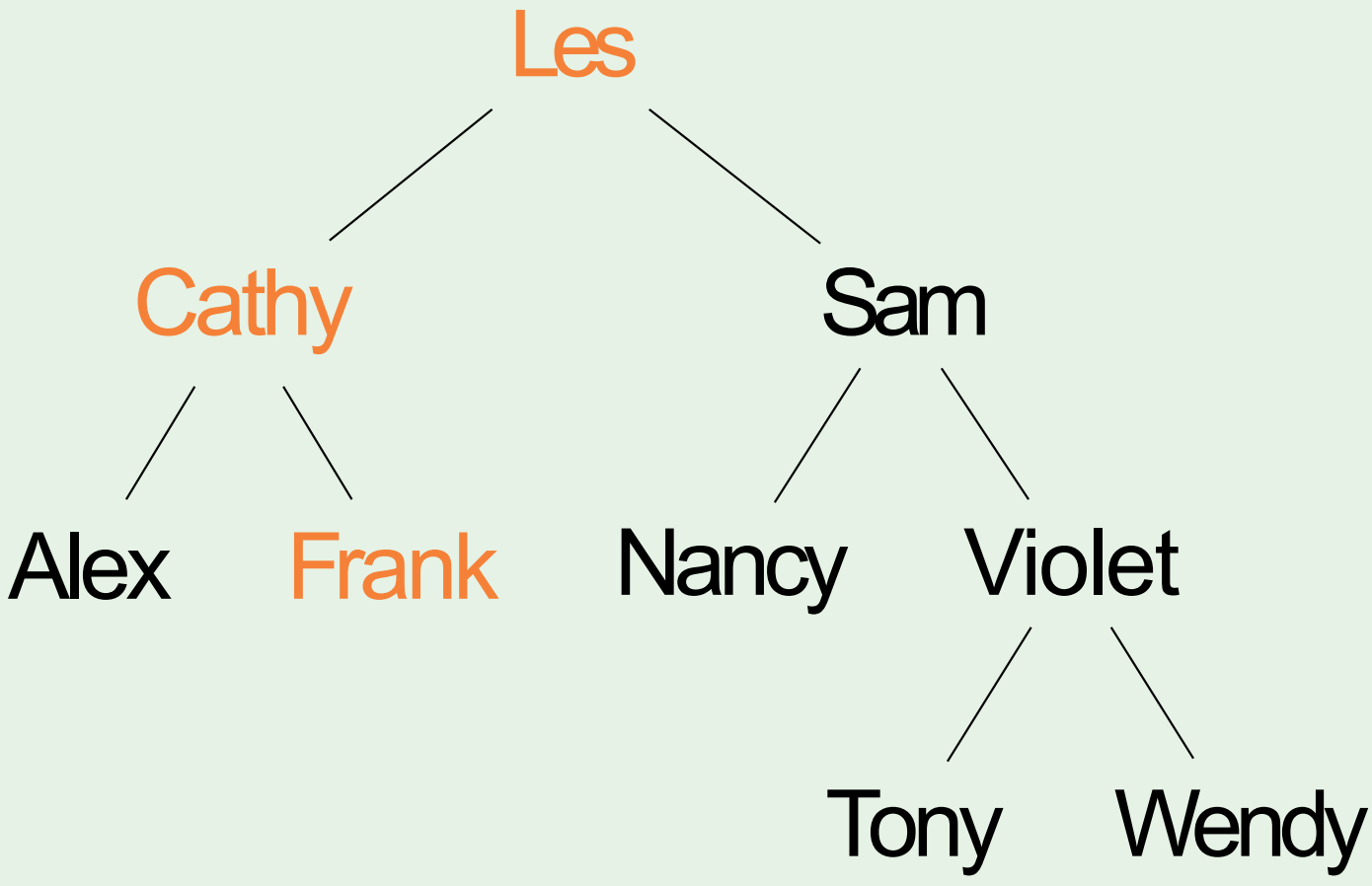
Output: AlexCathy

InOrderTraversal



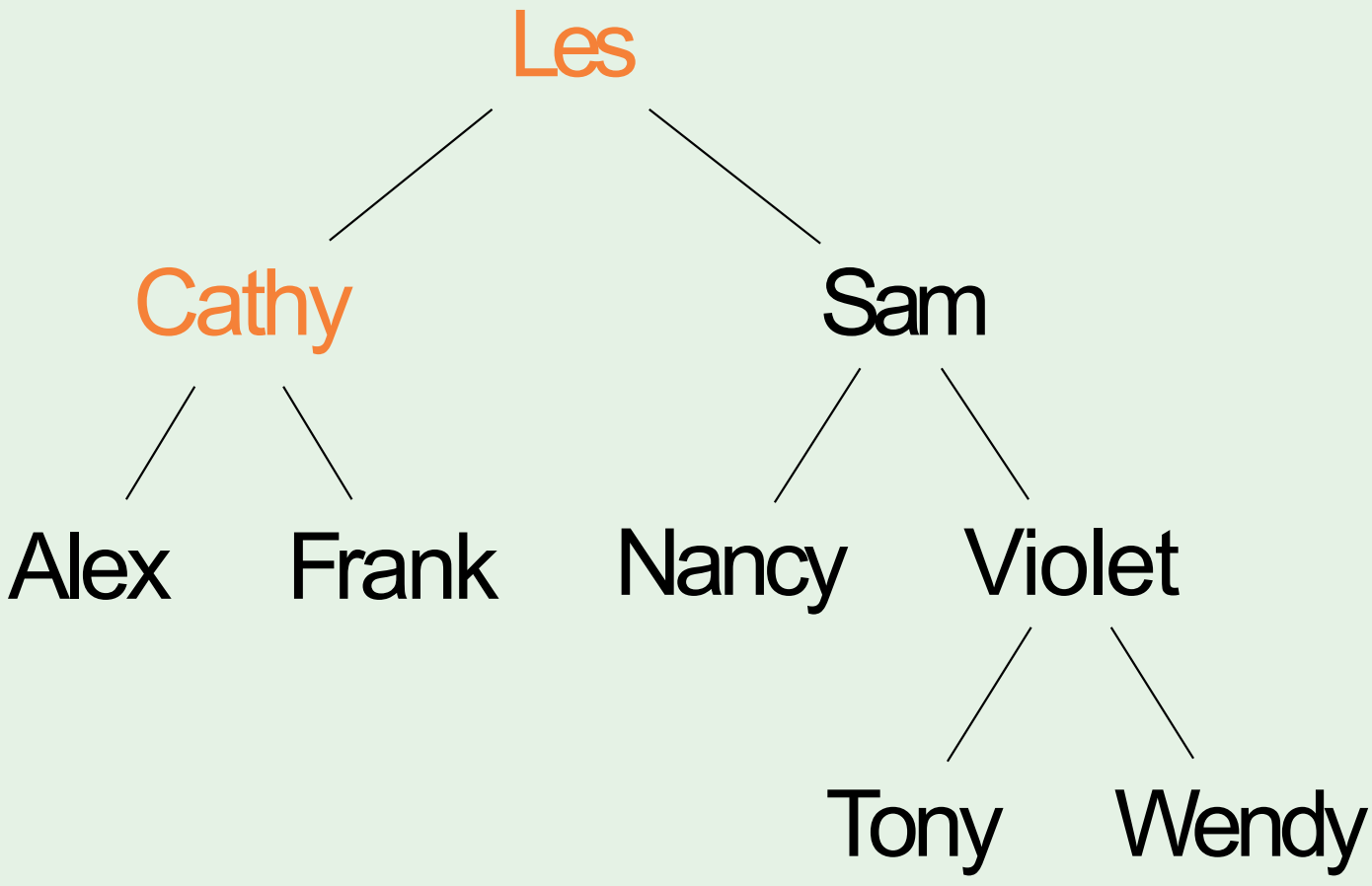
Output: Alex Cathy Frank

InOrderTraversal



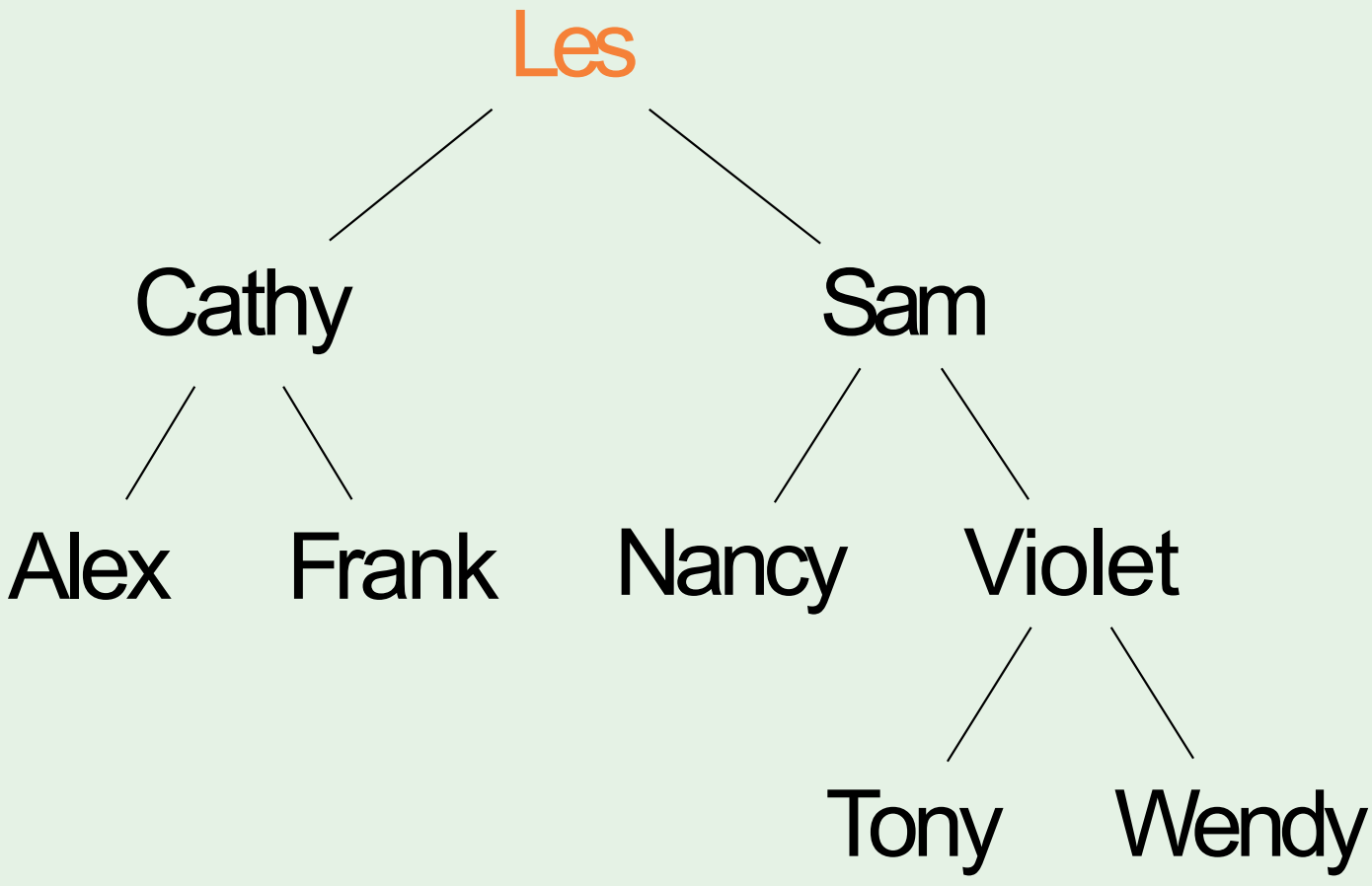
Output: Alex Cathy Frank

InOrderTraversal



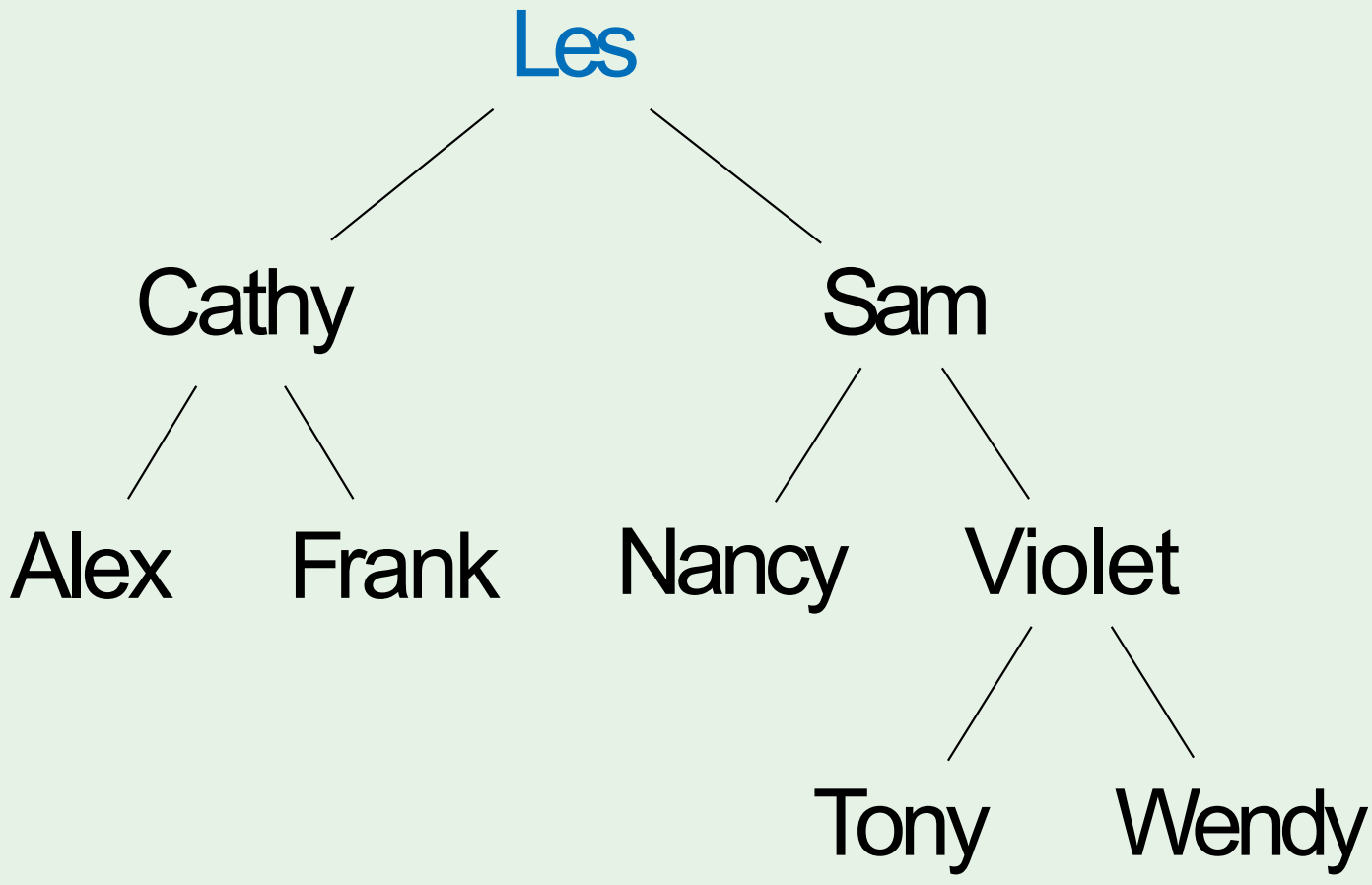
Output: Alex Cathy Frank

InOrderTraversal



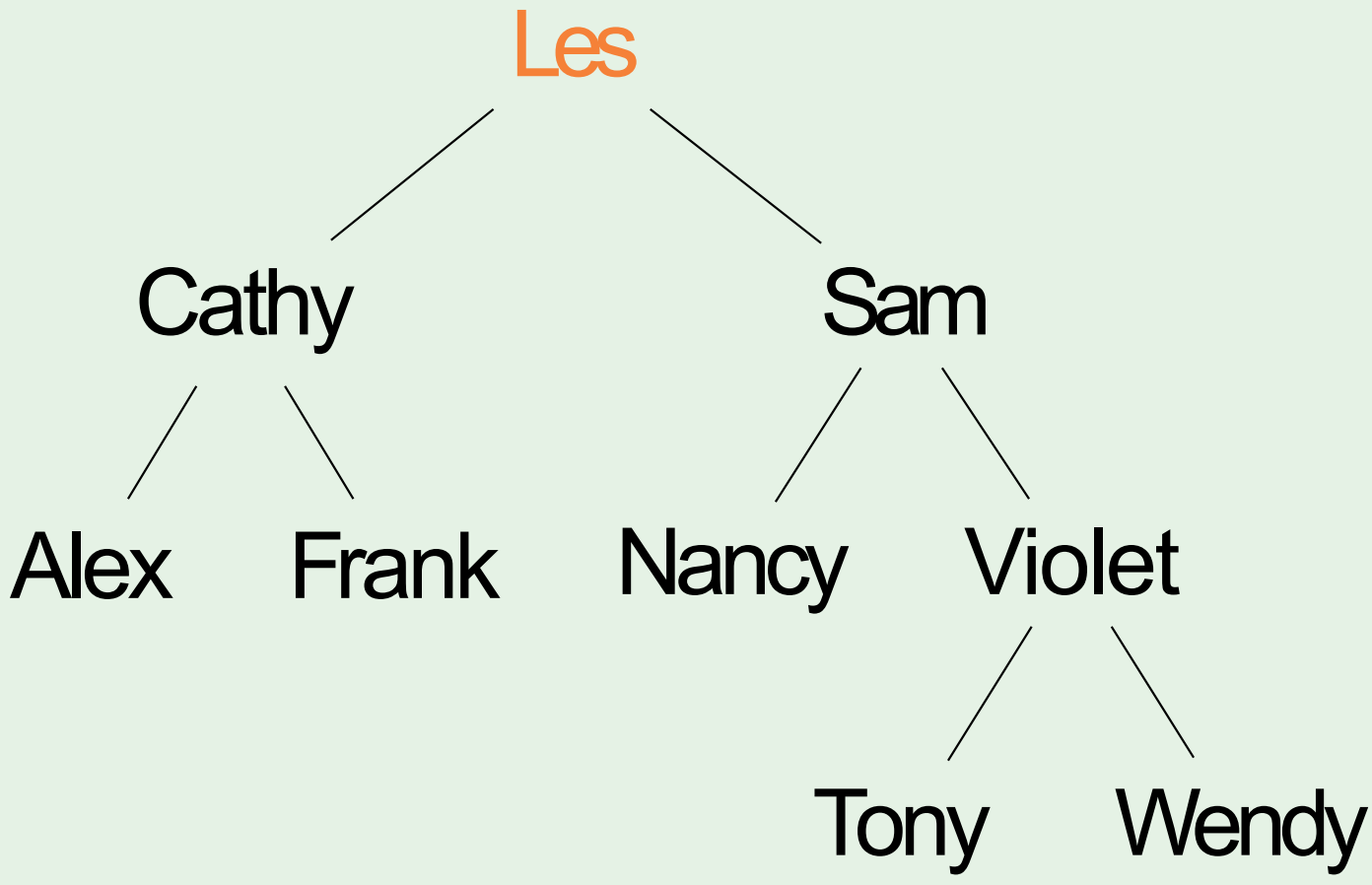
Output: Alex Cathy Frank

InOrderTraversal



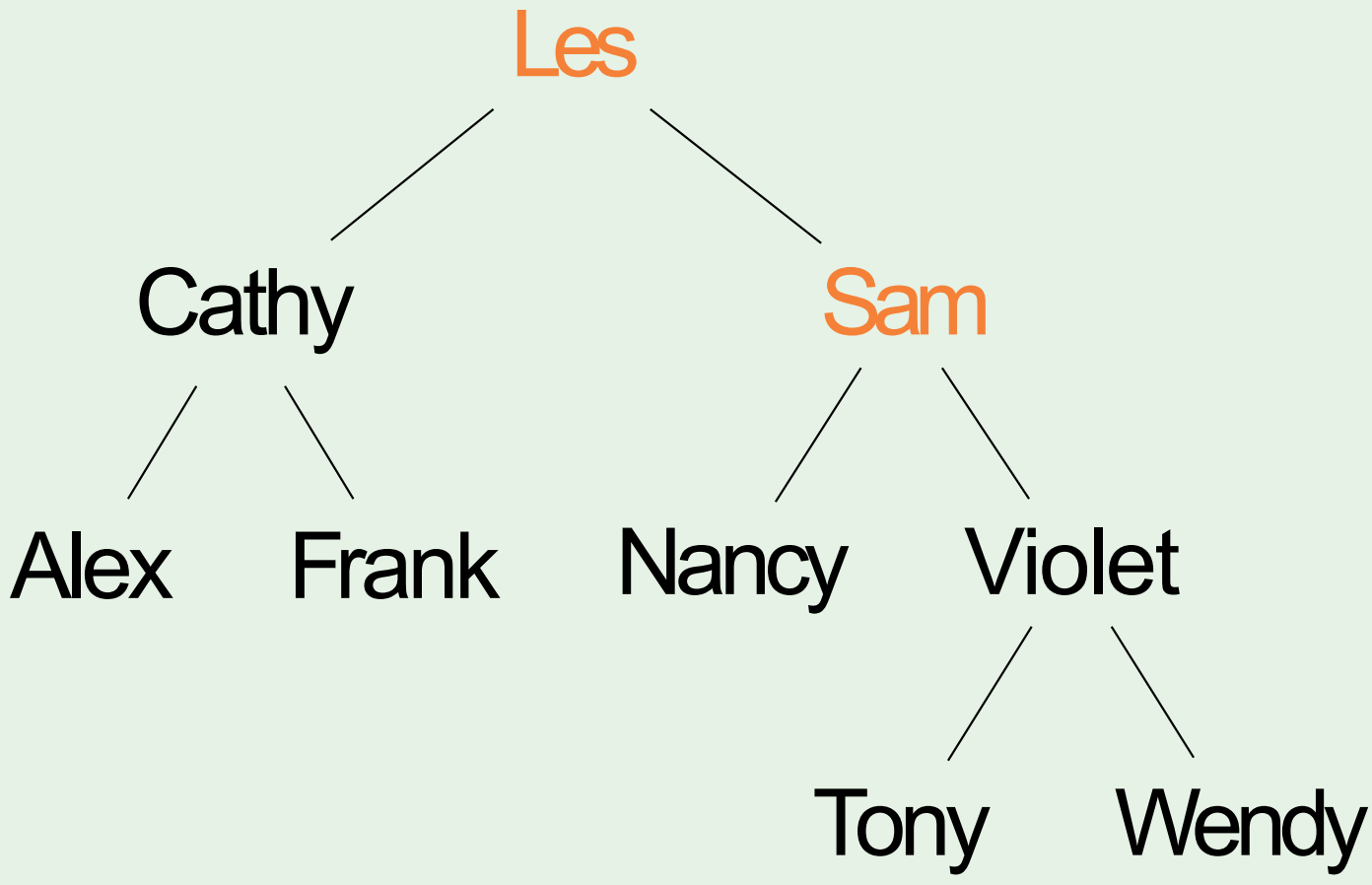
Output: Alex Cathy Frank Les

InOrderTraversal



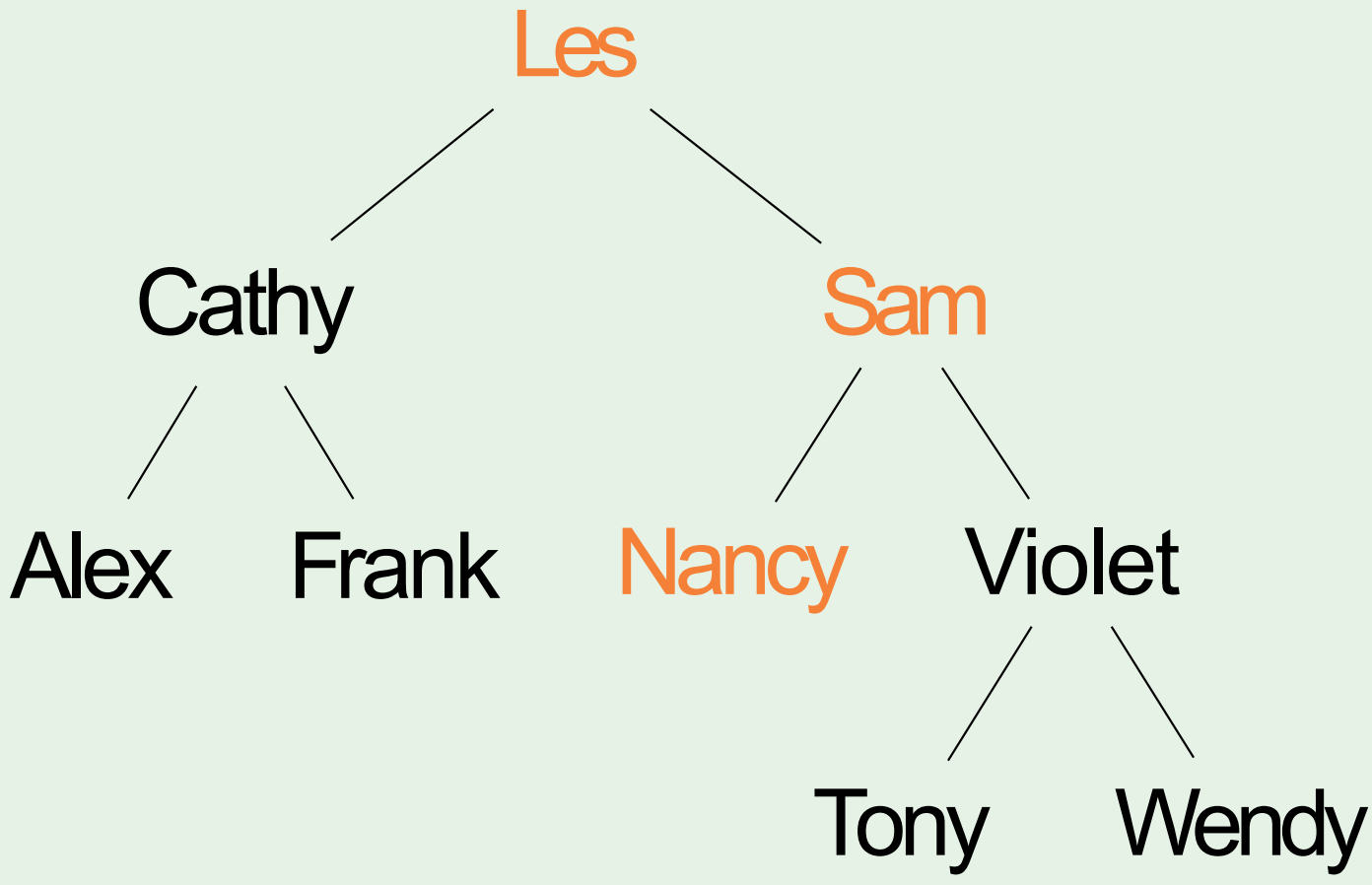
Output: Alex Cathy Frank Les

InOrderTraversal



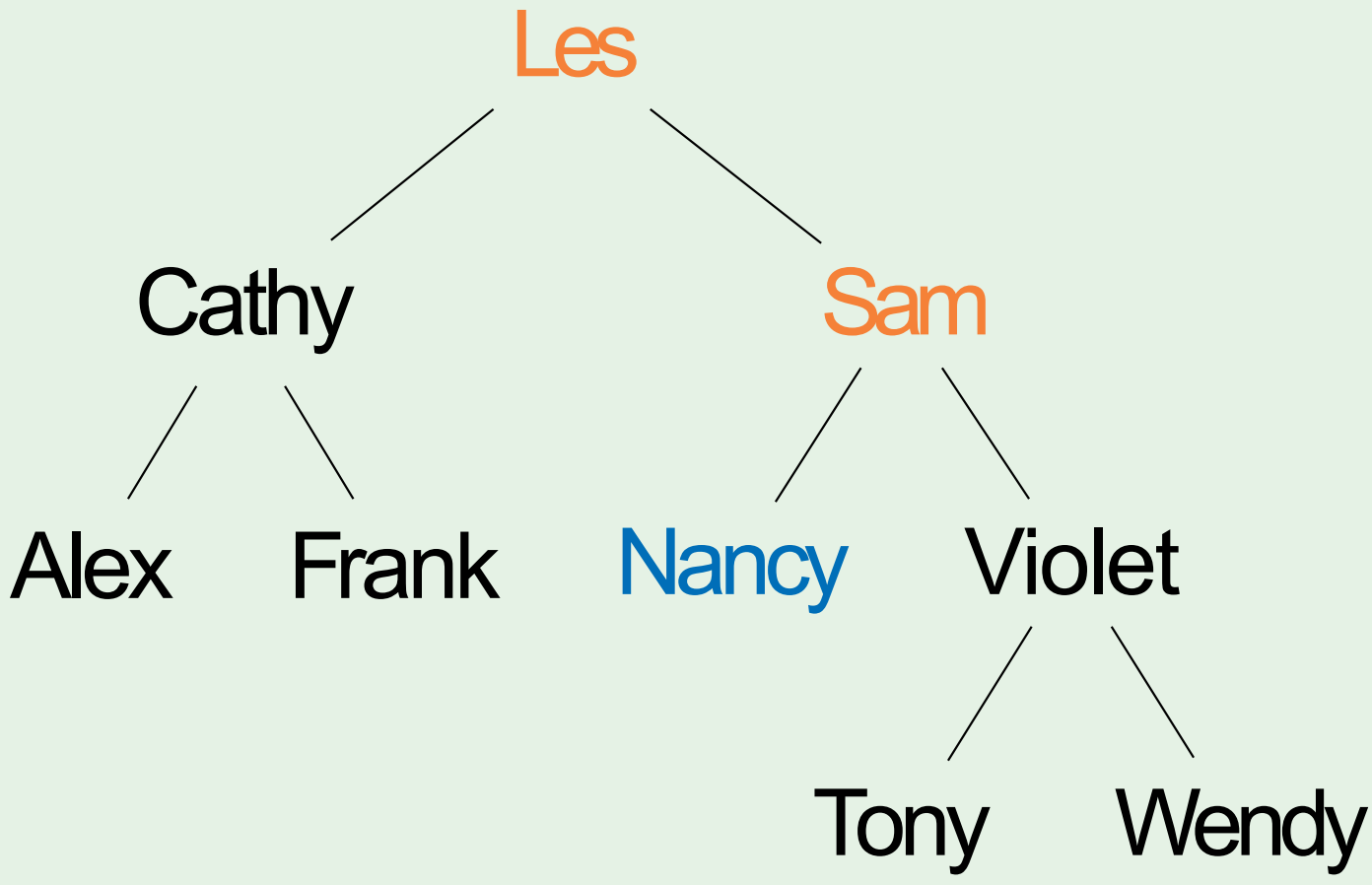
Output: Alex Cathy Frank Les

InOrderTraversal



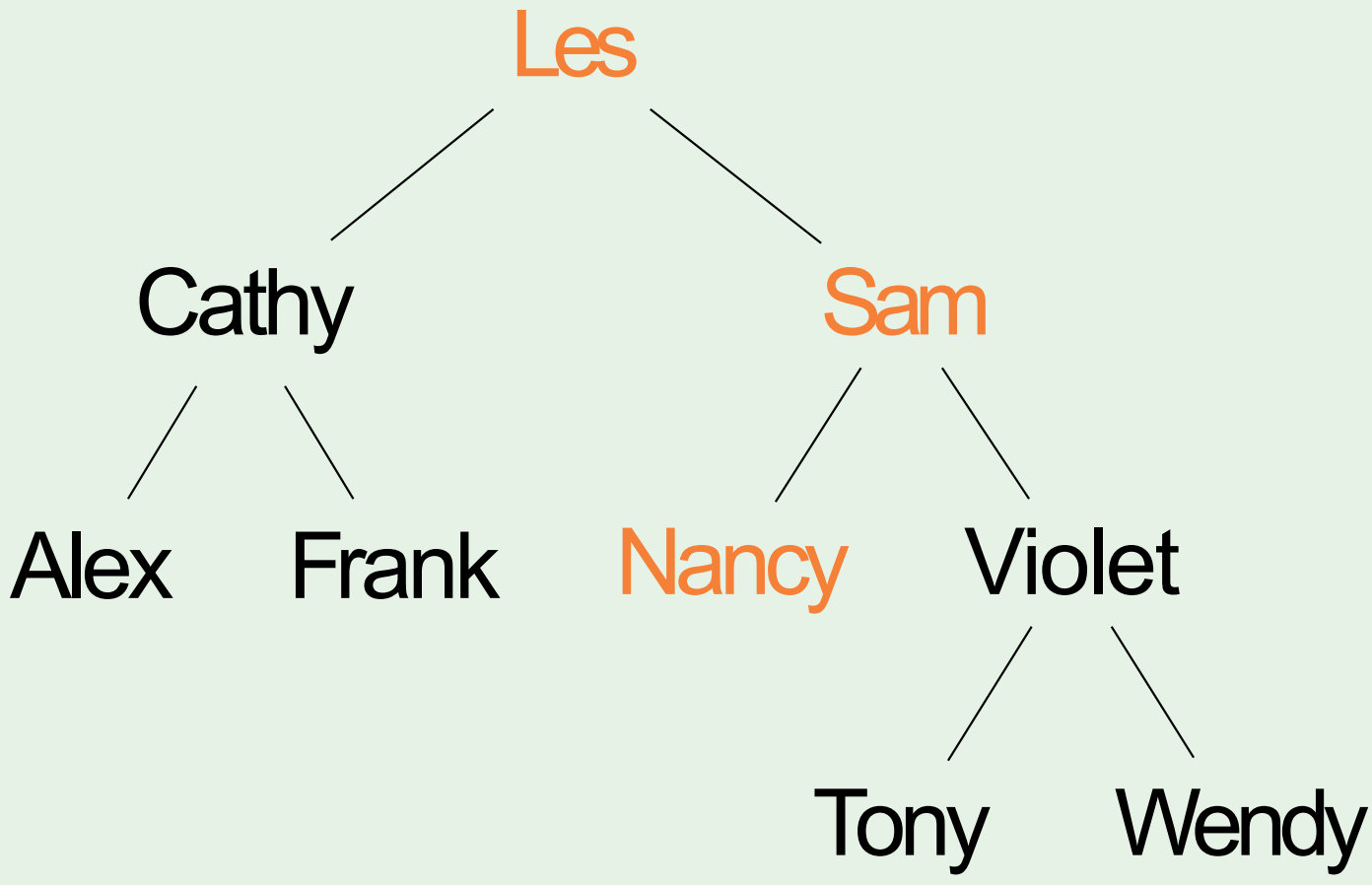
Output: Alex Cathy Frank Les

InOrderTraversal



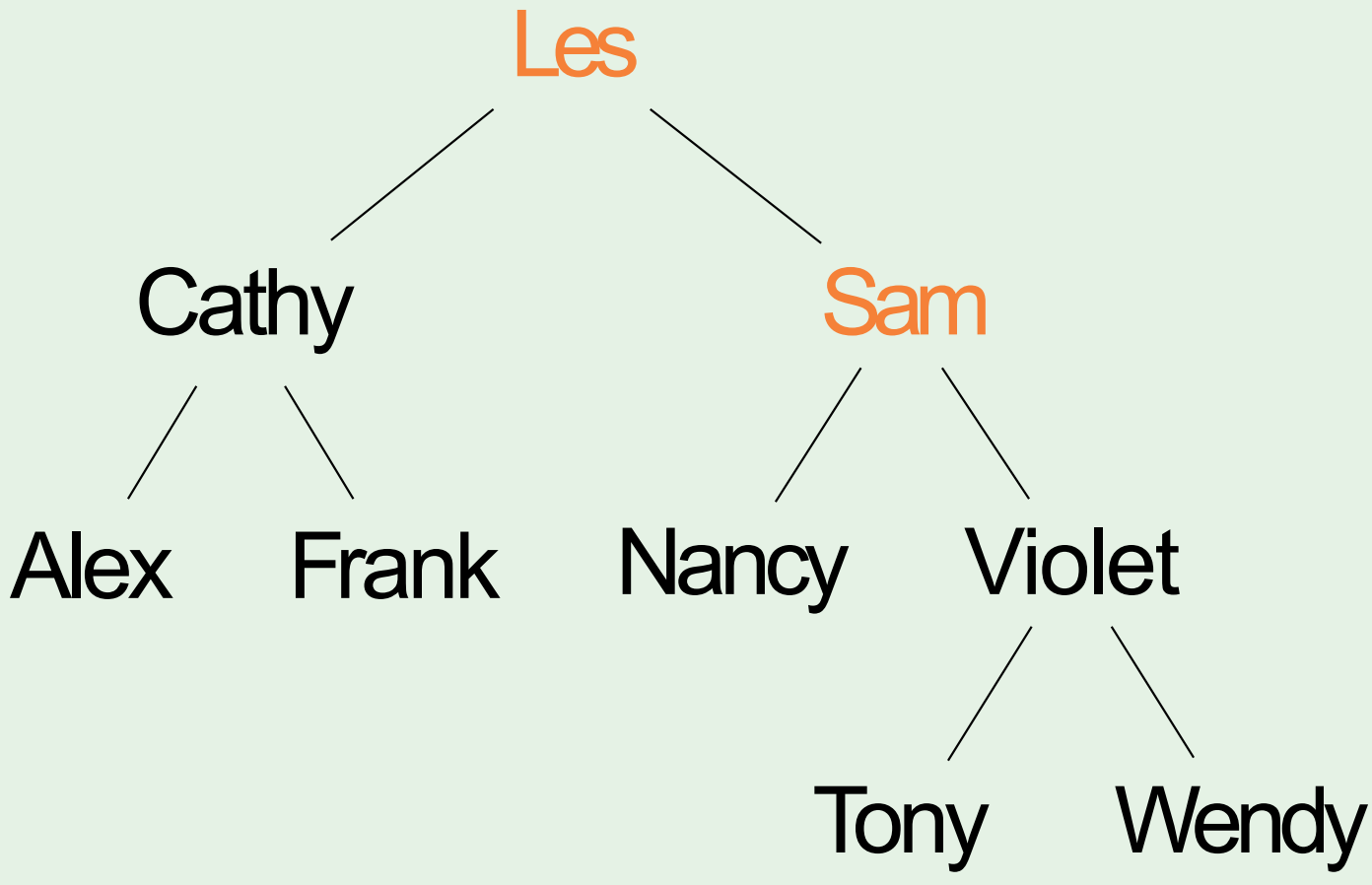
Output: Alex Cathy Frank Les Nancy

InOrderTraversal



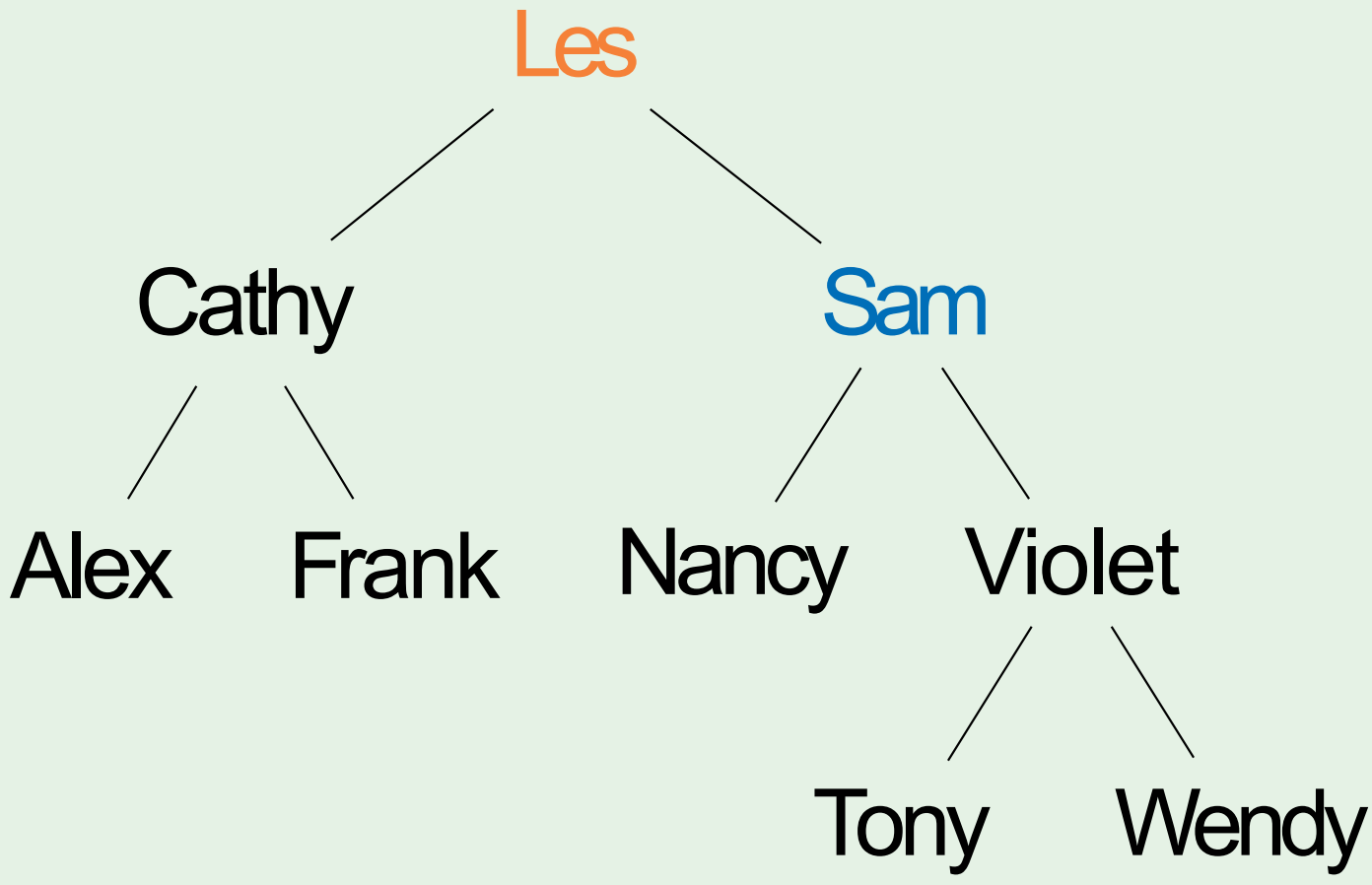
Output: Alex Cathy Frank Les Nancy

InOrderTraversal



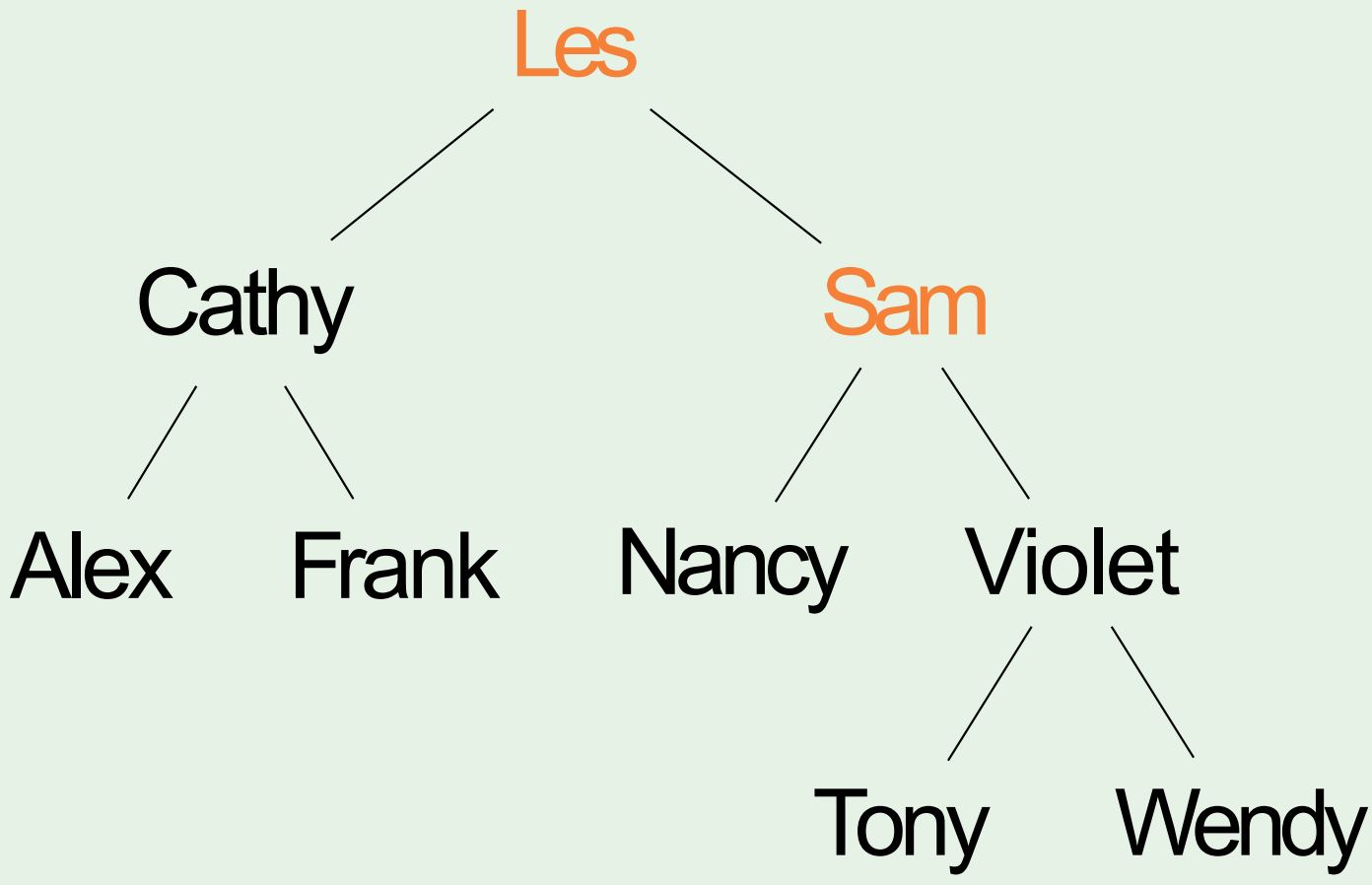
Output: Alex Cathy Frank Les Nancy

InOrderTraversal



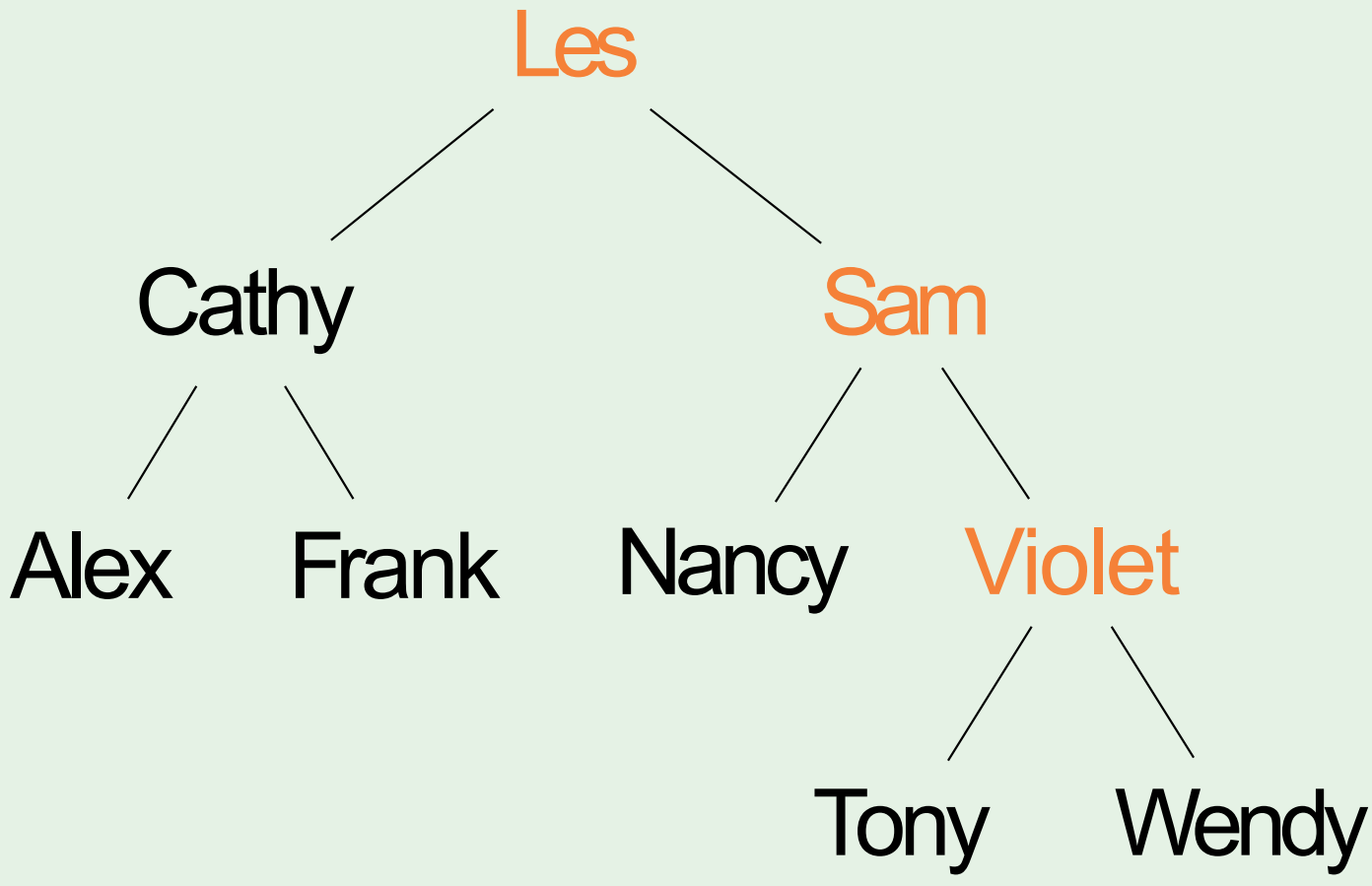
Output: Alex Cathy Frank Les Nancy Sam

InOrderTraversal



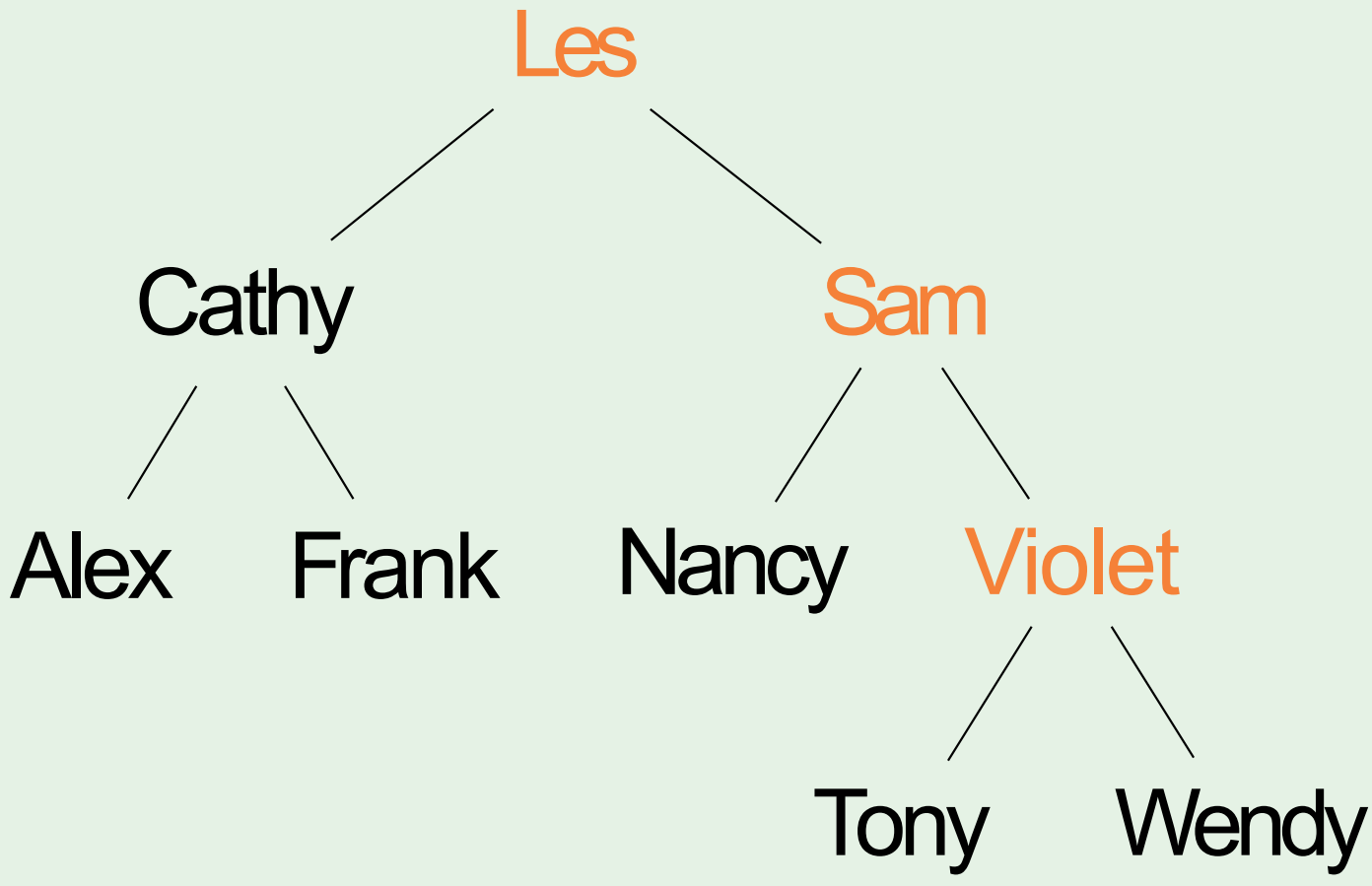
Output: Alex Cathy Frank Les Nancy Sam

InOrderTraversal



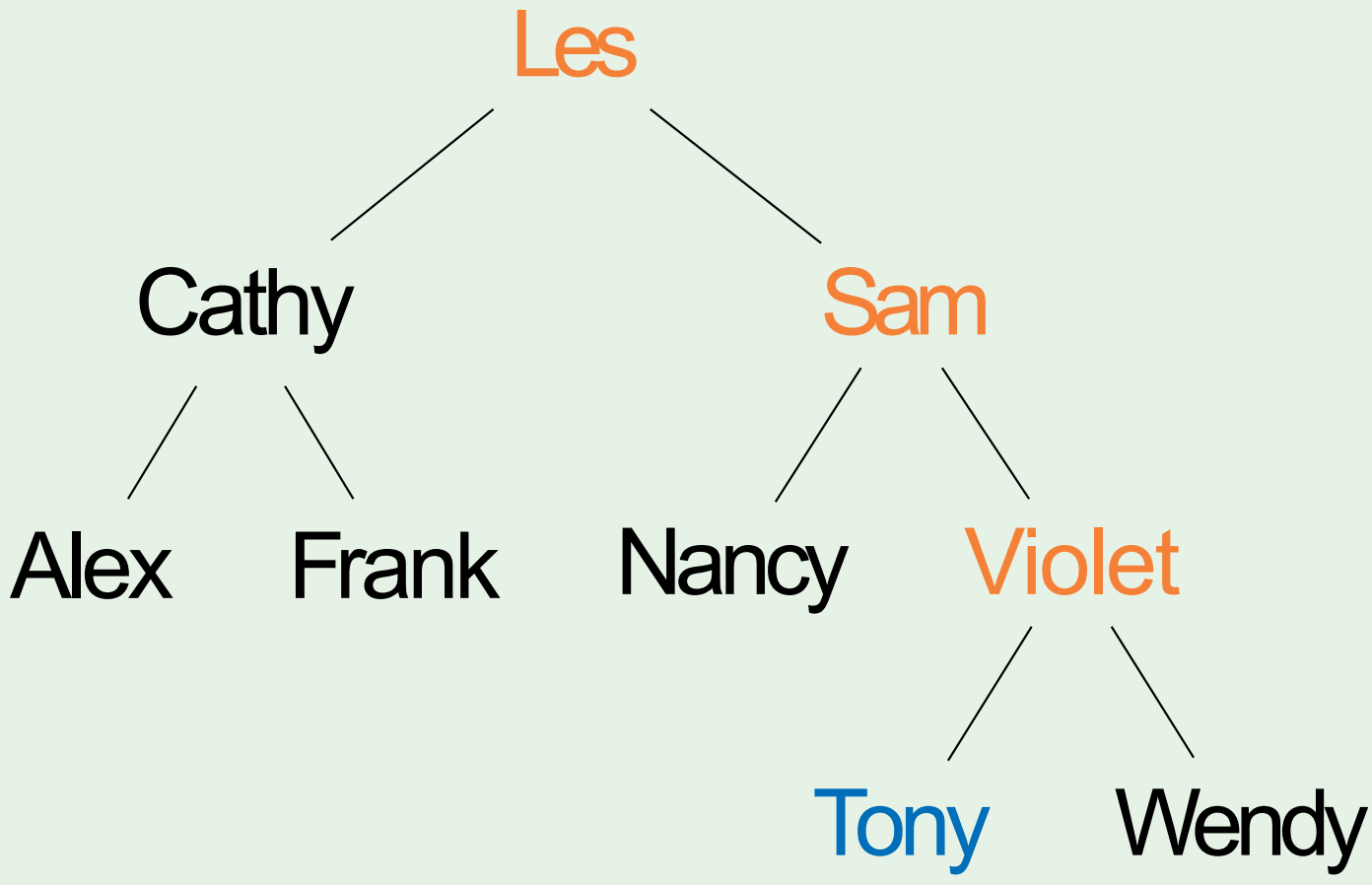
Output: Alex Cathy Frank Les Nancy Sam

InOrderTraversal



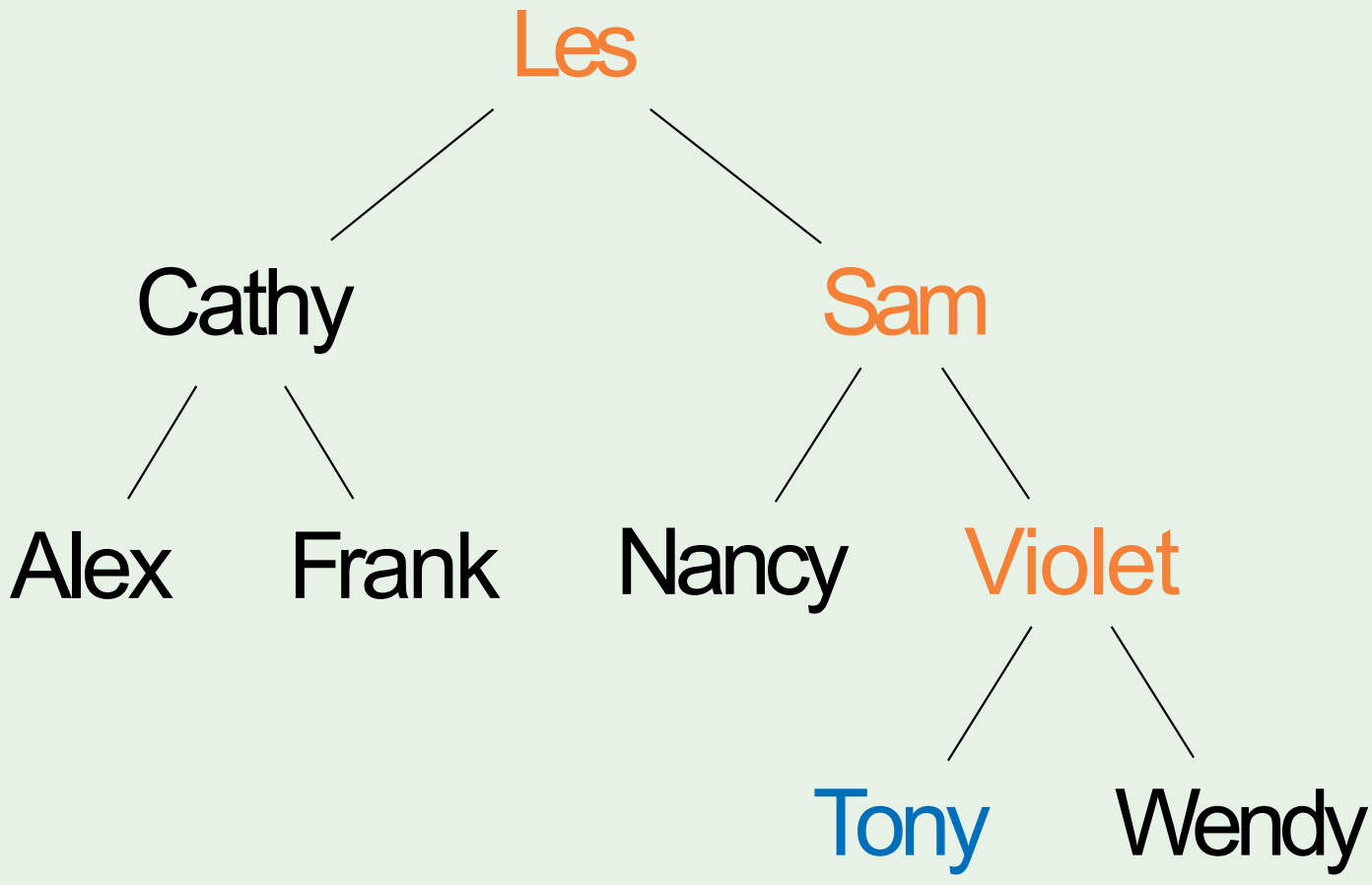
Output: Alex Cathy Frank Les Nancy Sam

InOrderTraversal



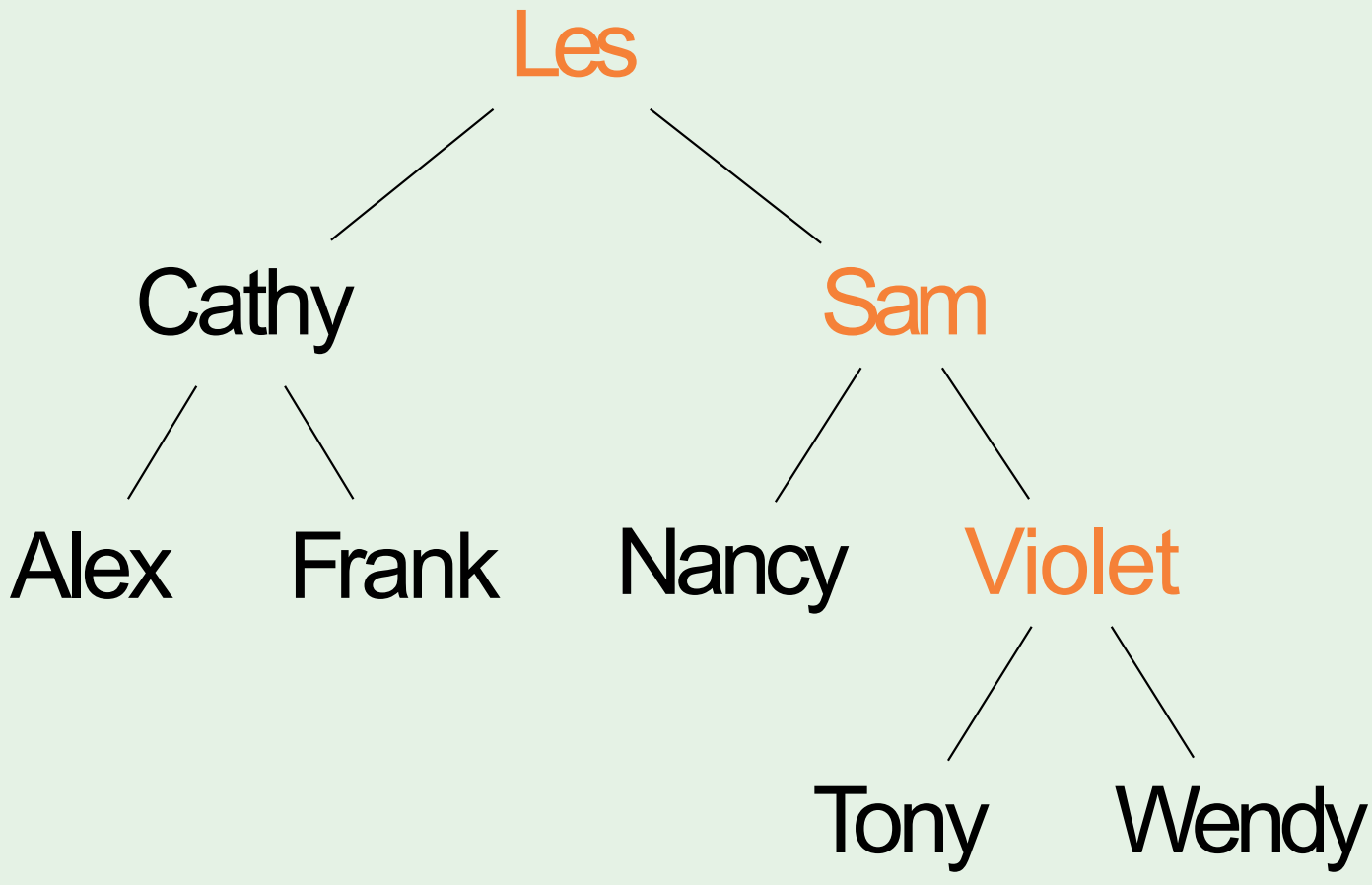
Output: Alex Cathy Frank Les Nancy Sam Tony

InOrderTraversal



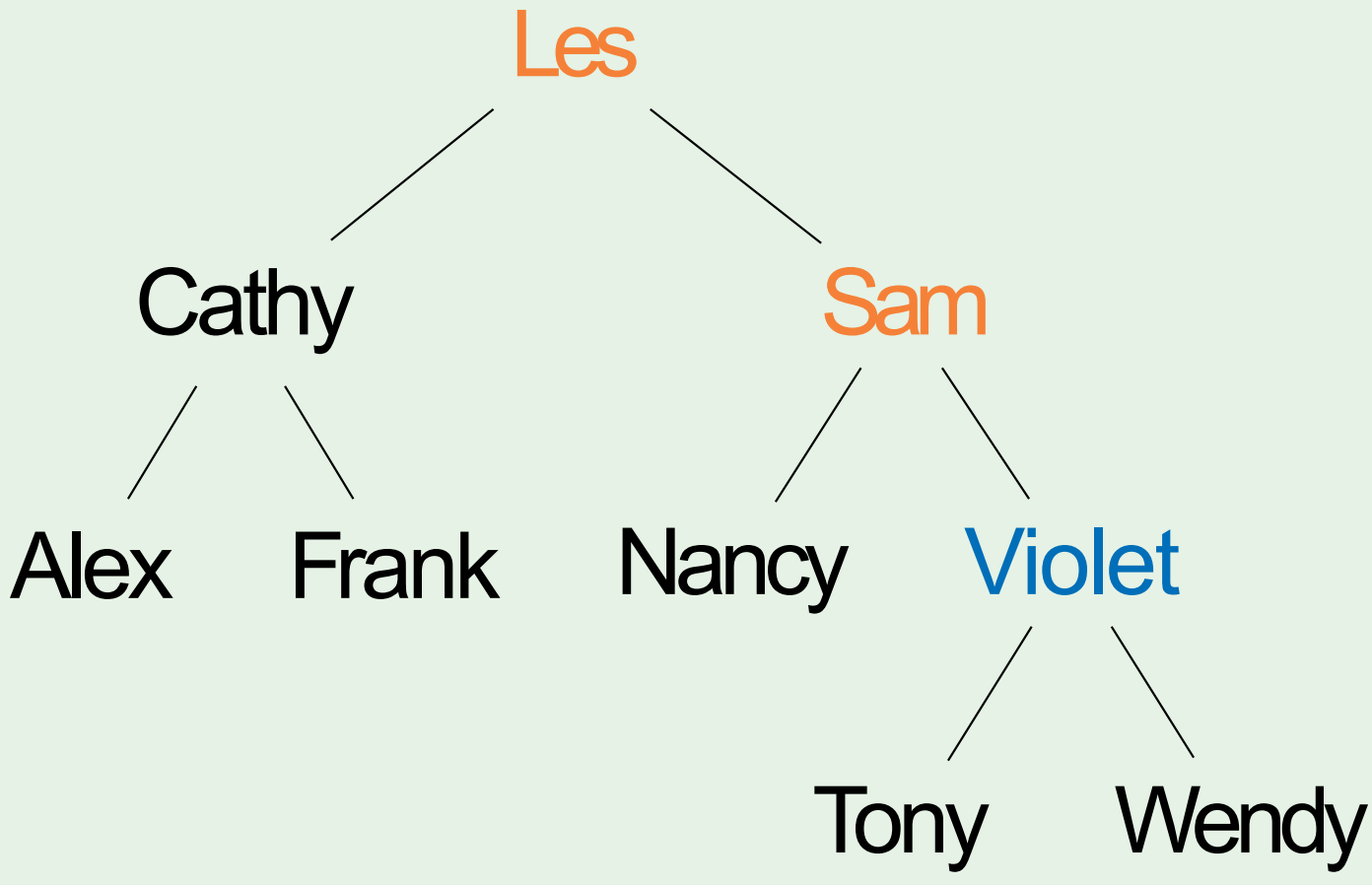
Output: Alex Cathy Frank Les Nancy Sam Tony

InOrderTraversal



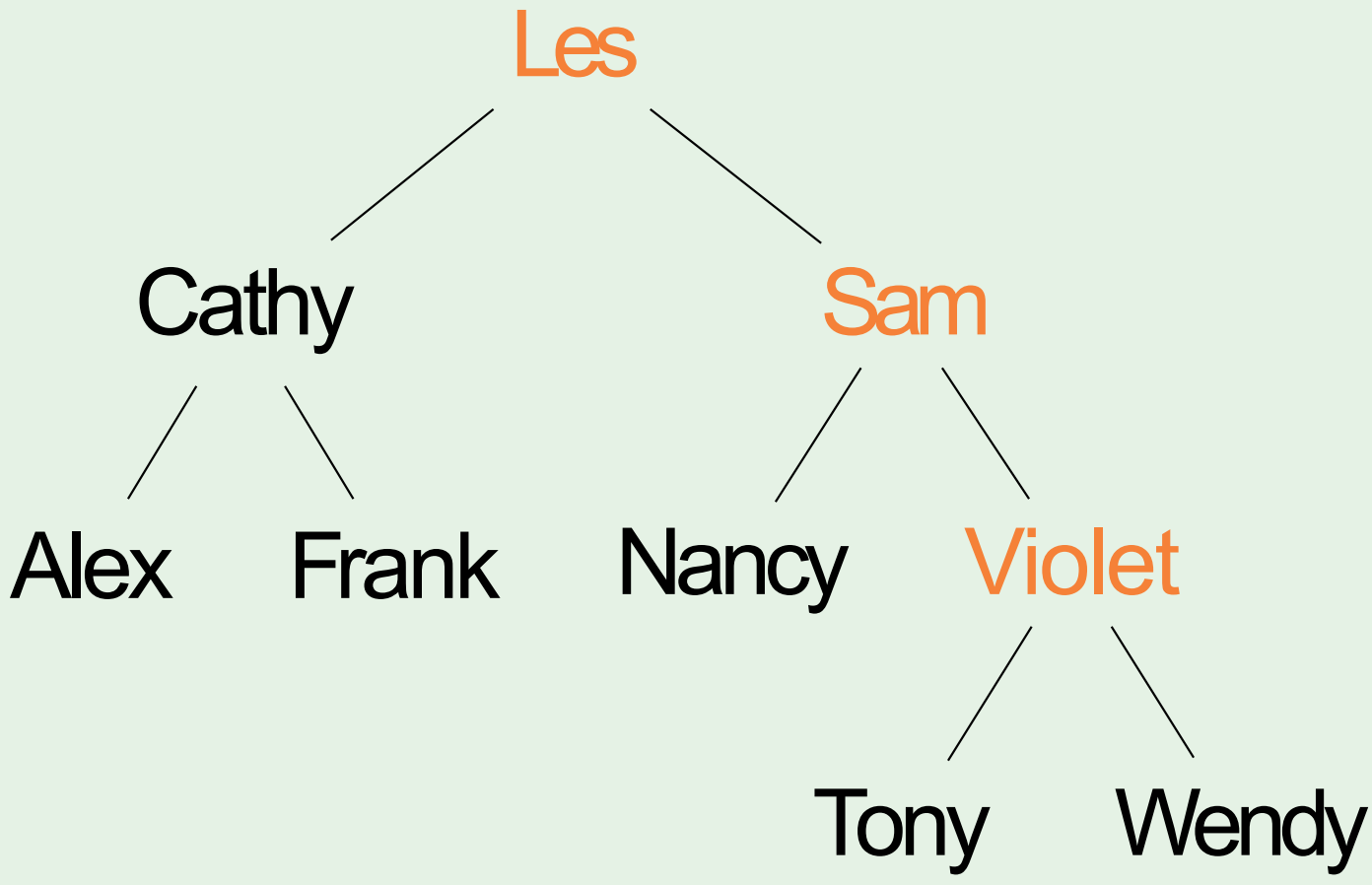
Output: Alex Cathy Frank Les Nancy Sam Tony

InOrderTraversal



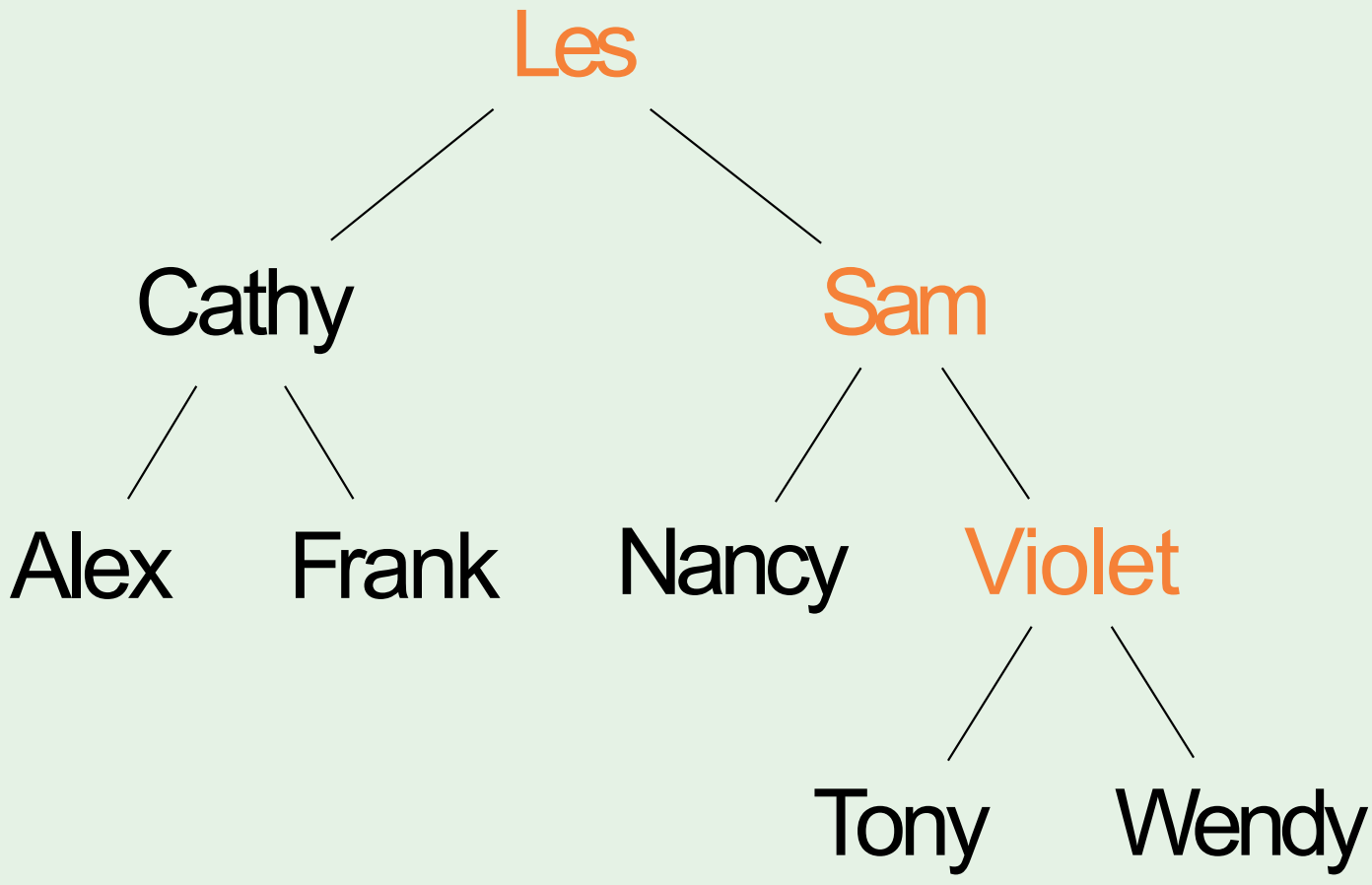
Output: Alex Cathy Frank Les Nancy Sam Tony Violet

InOrderTraversal



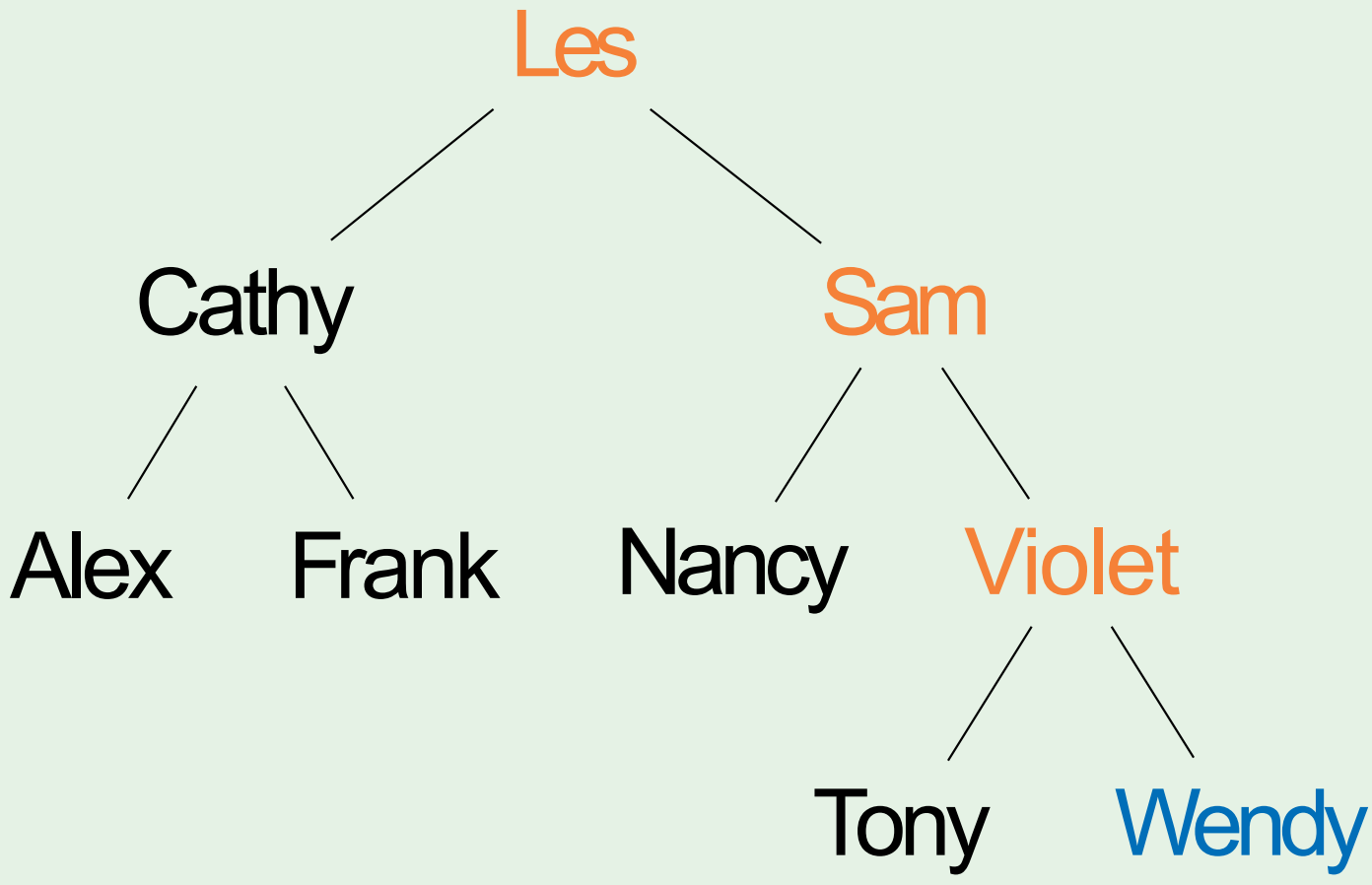
Output: Alex Cathy Frank Les Nancy Sam Tony Violet

InOrderTraversal



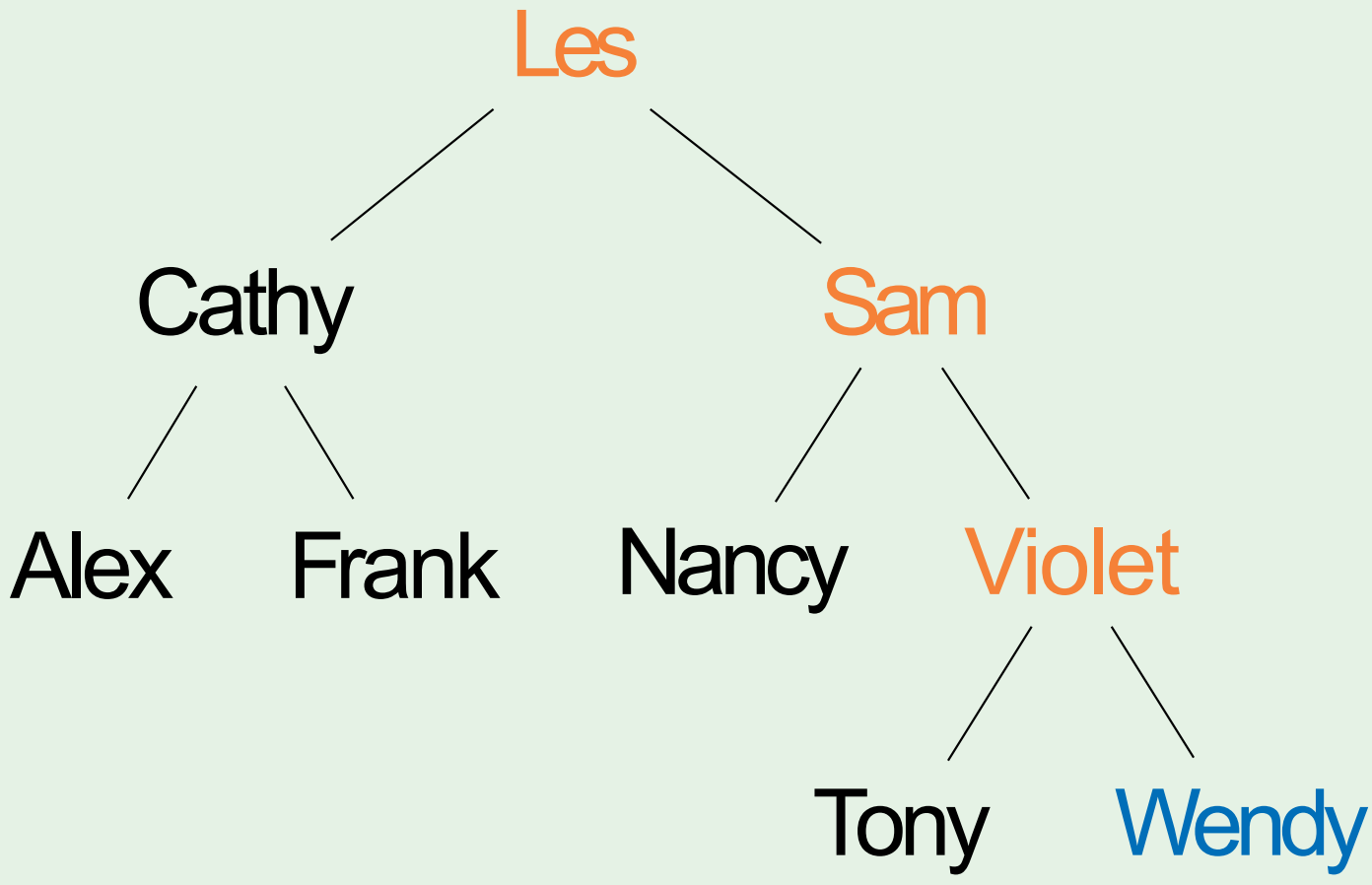
Output: Alex Cathy Frank Les Nancy Sam Tony Violet

InOrderTraversal



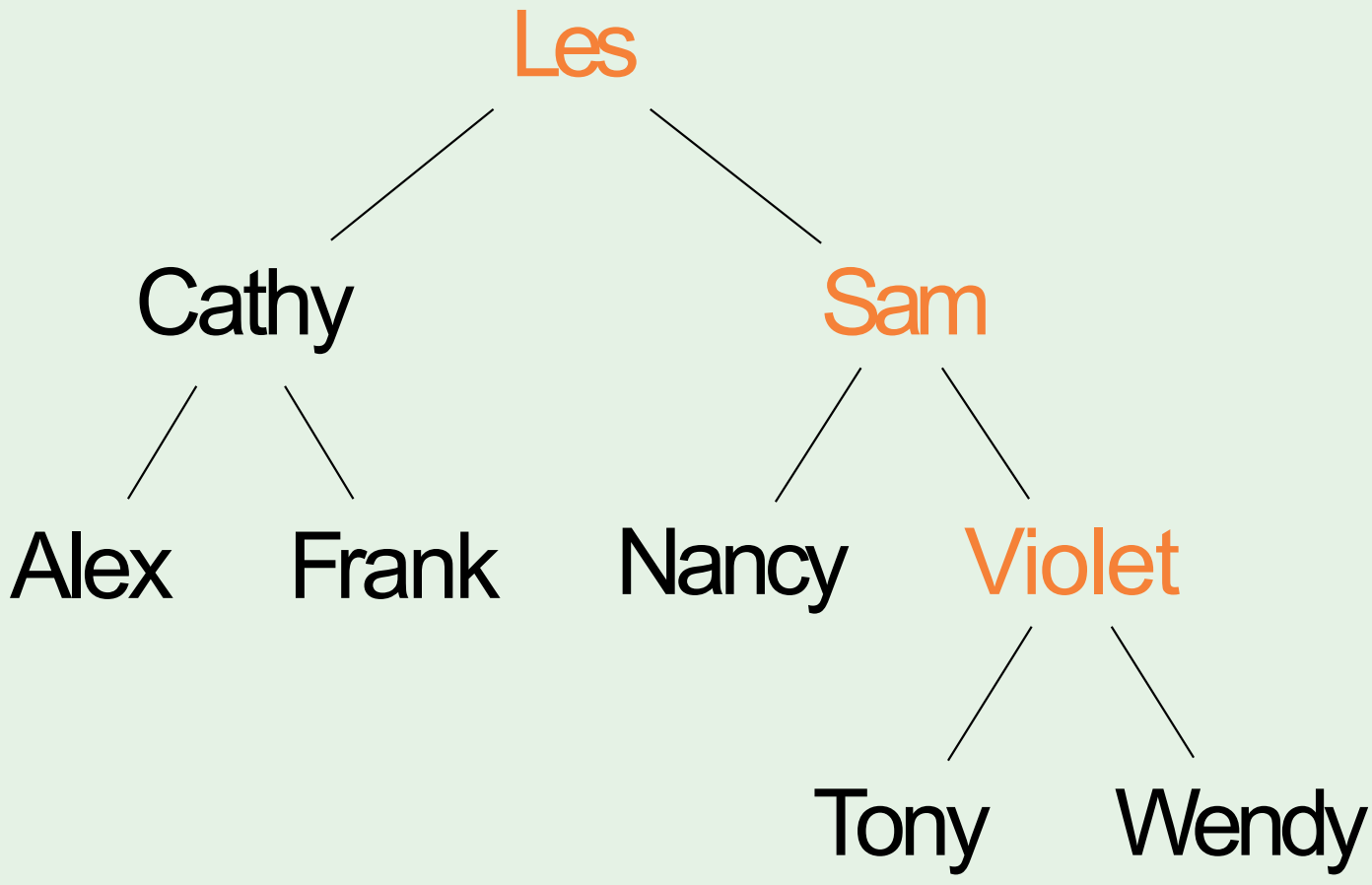
Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

InOrderTraversal



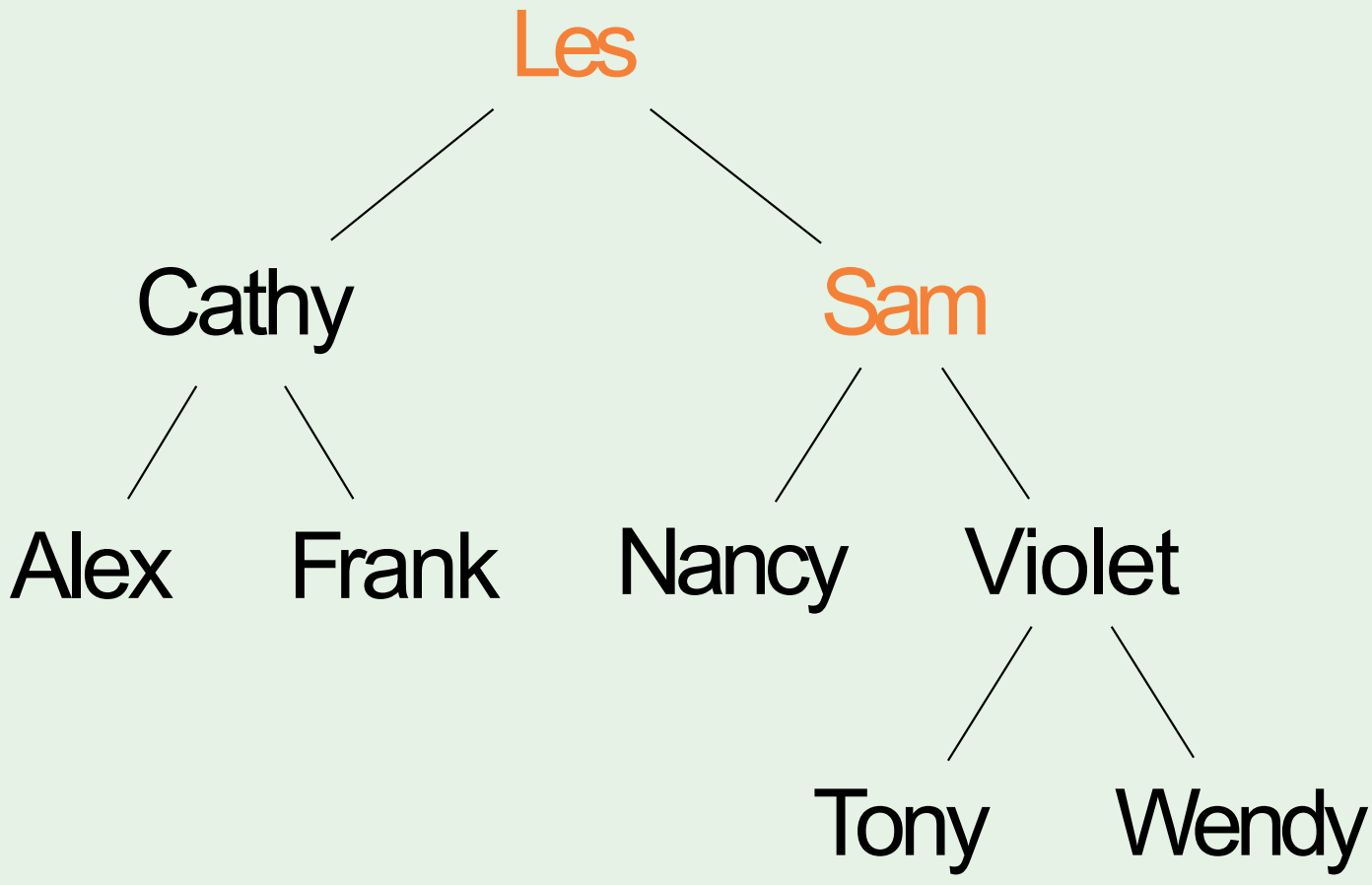
Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

InOrderTraversal



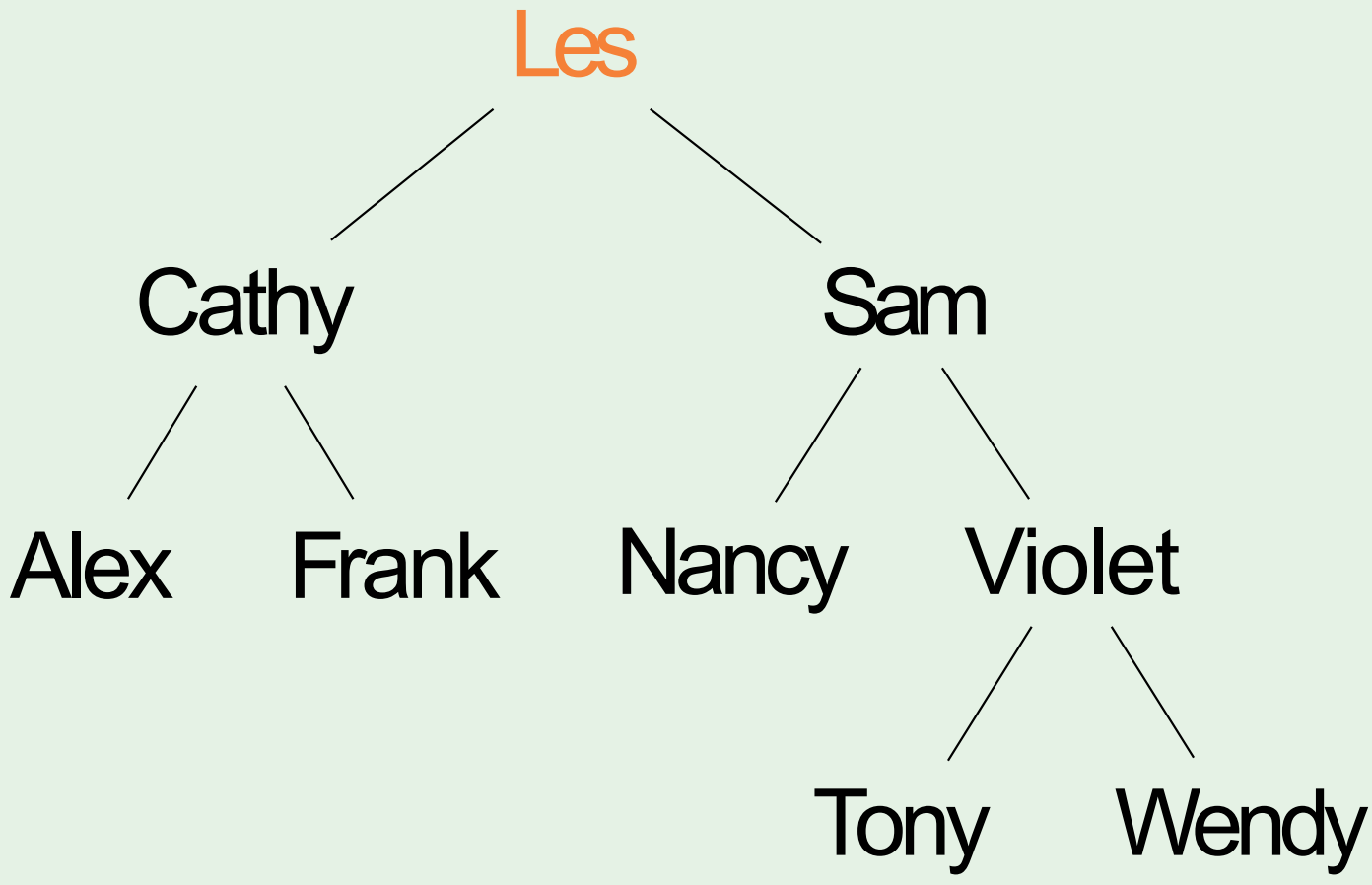
Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

InOrderTraversal



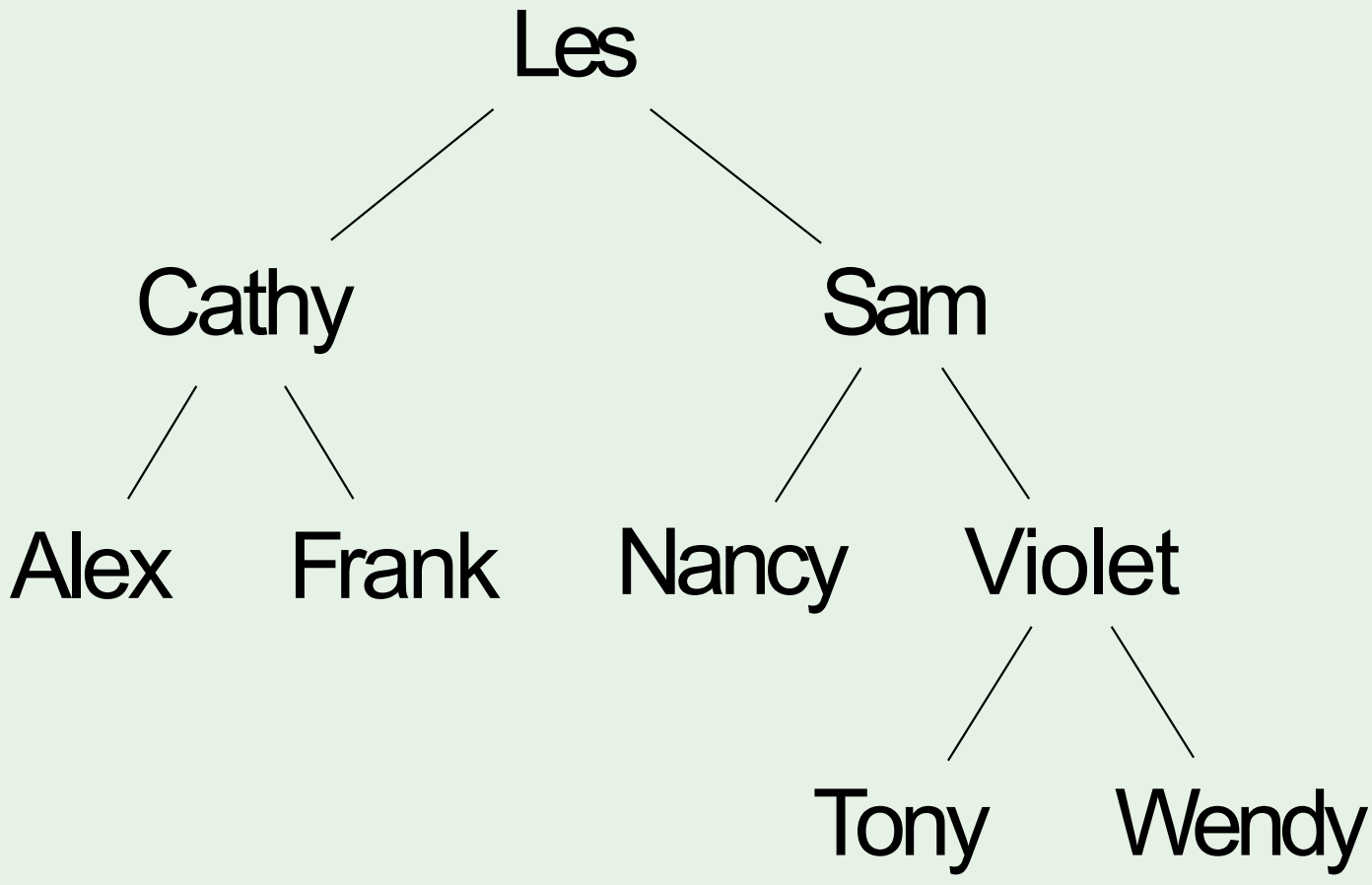
Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

InOrderTraversal



Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

InOrderTraversal



Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

Assignment

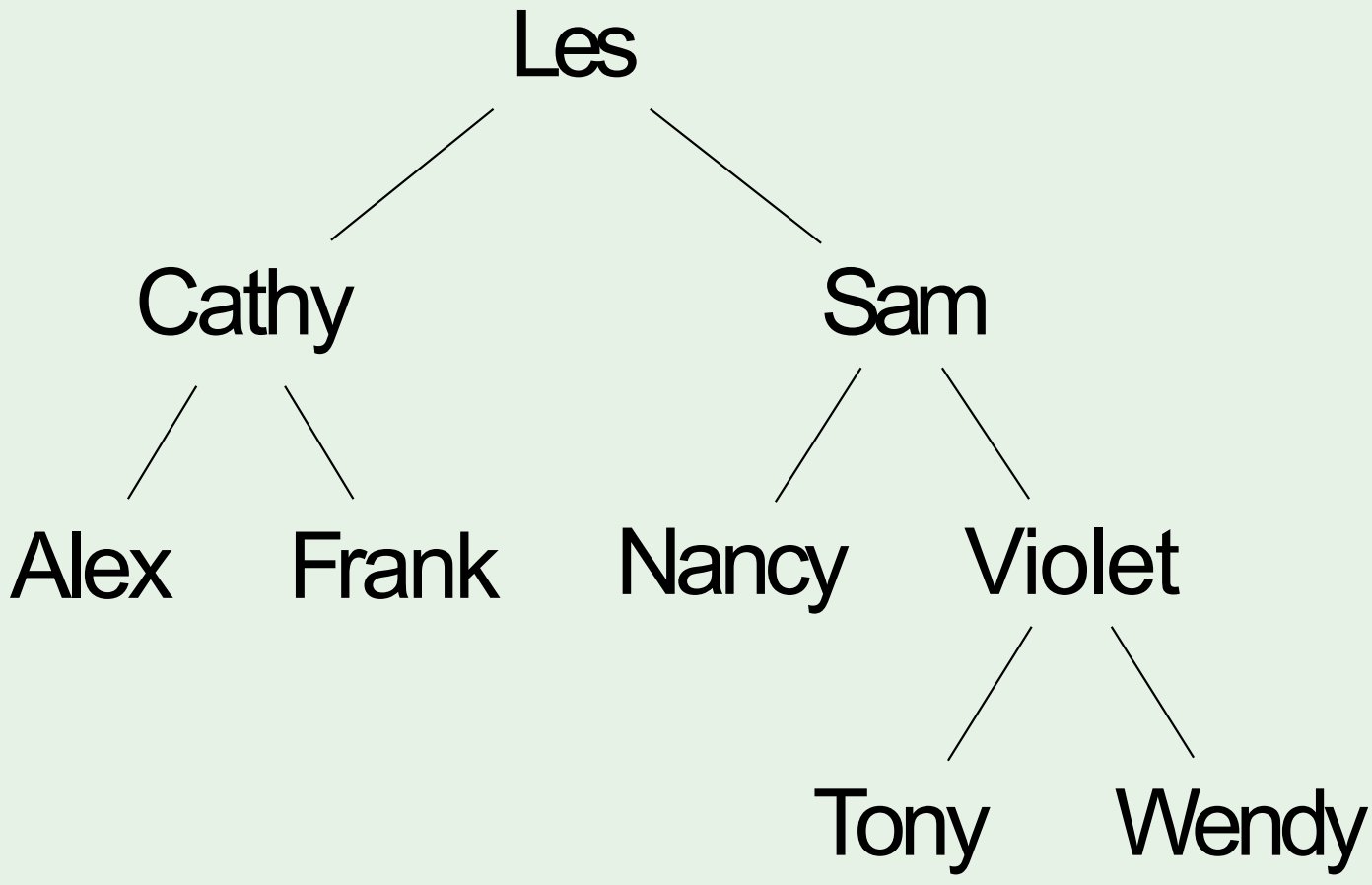
- Take input from user in InOrder and create the binary tree.
- Draw the Tree

Depth-first

PreOrderTraversal(*tree*)

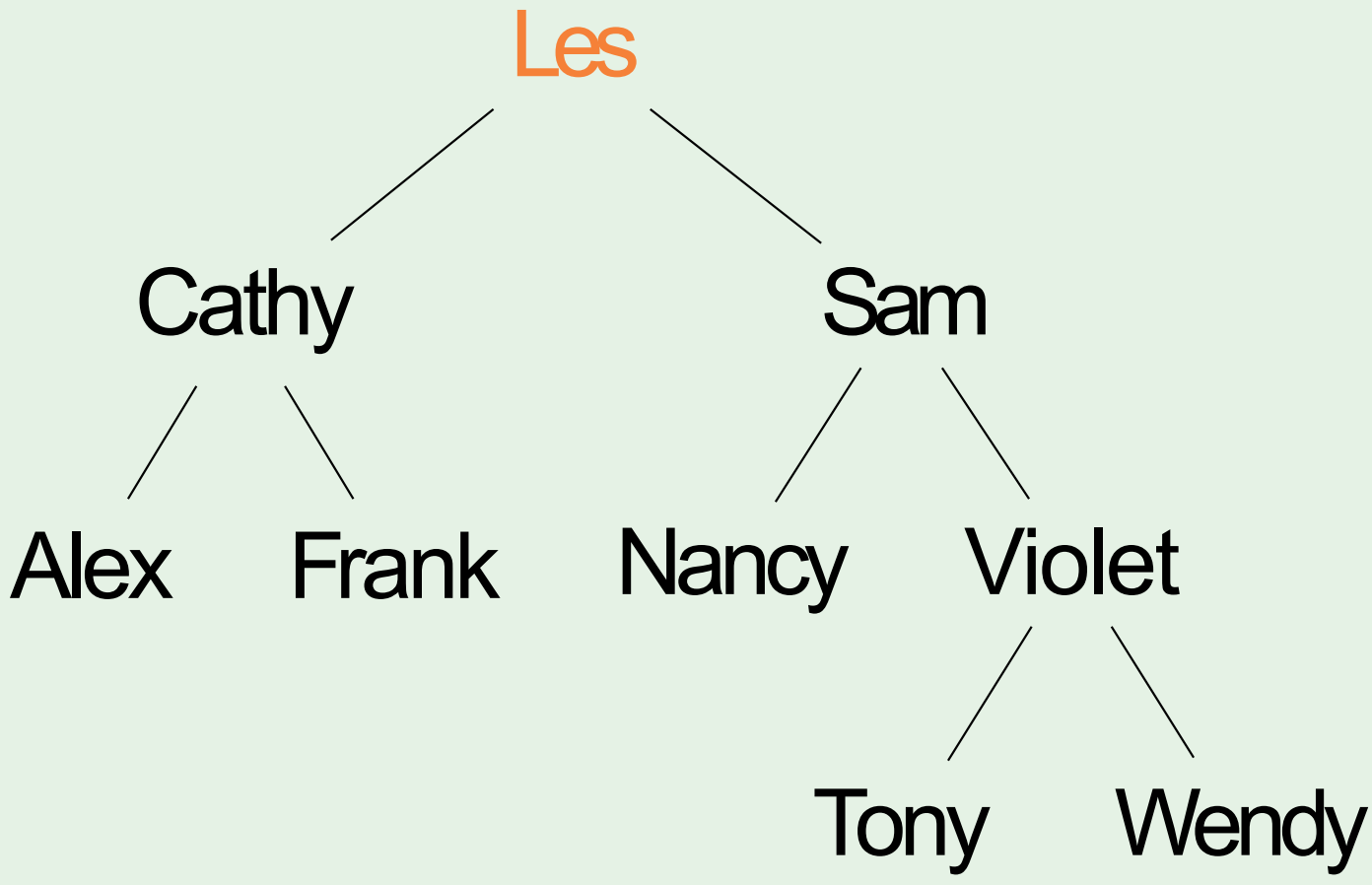
```
if tree = nil :
   return
Print(tree.key )
PreOrderTraversal(tree.left)
PreOrderTraversal(tree.right)
```

PreOrderTraversal



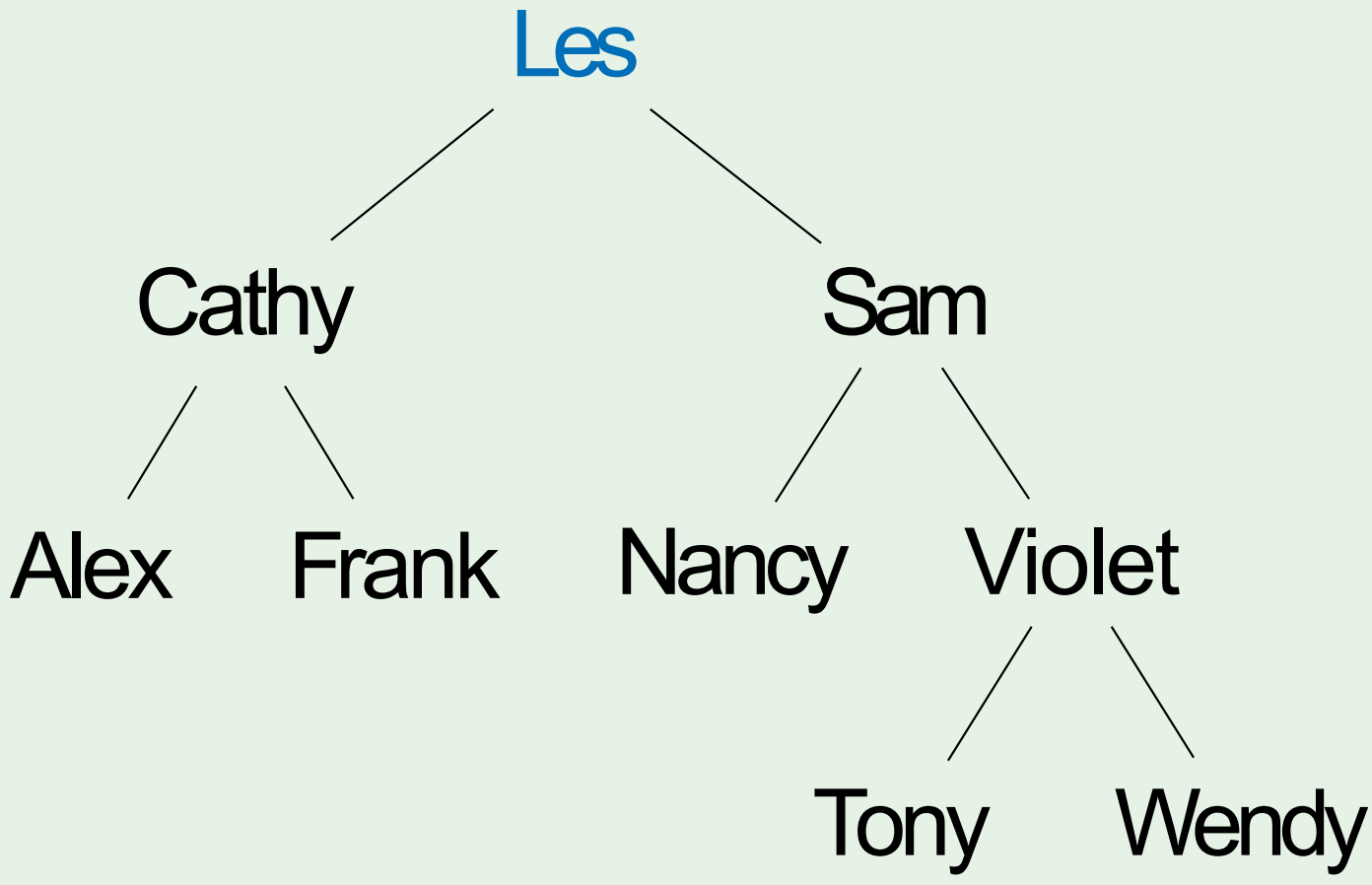
Output:

PreOrderTraversal



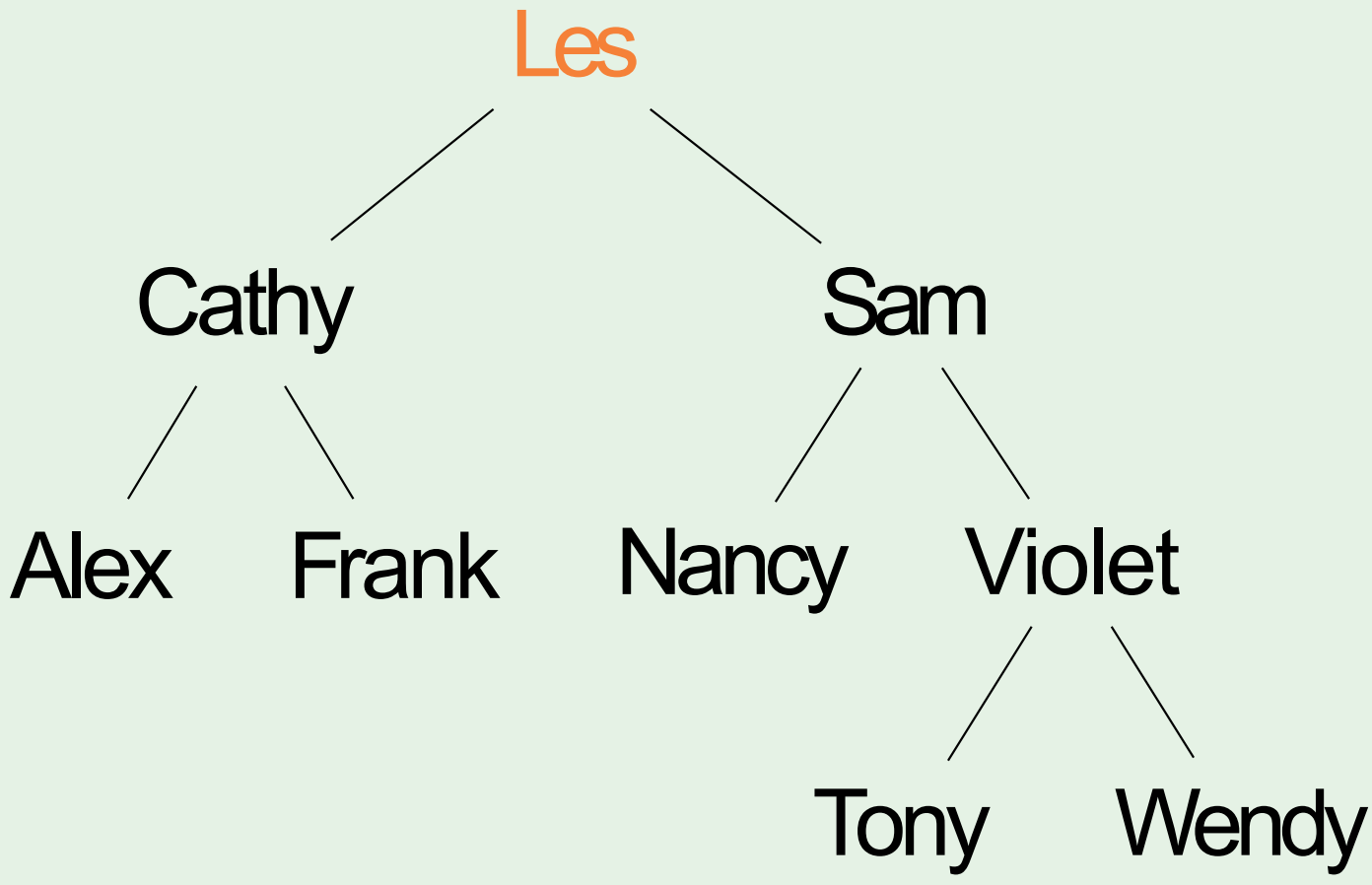
Output:

PreOrderTraversal



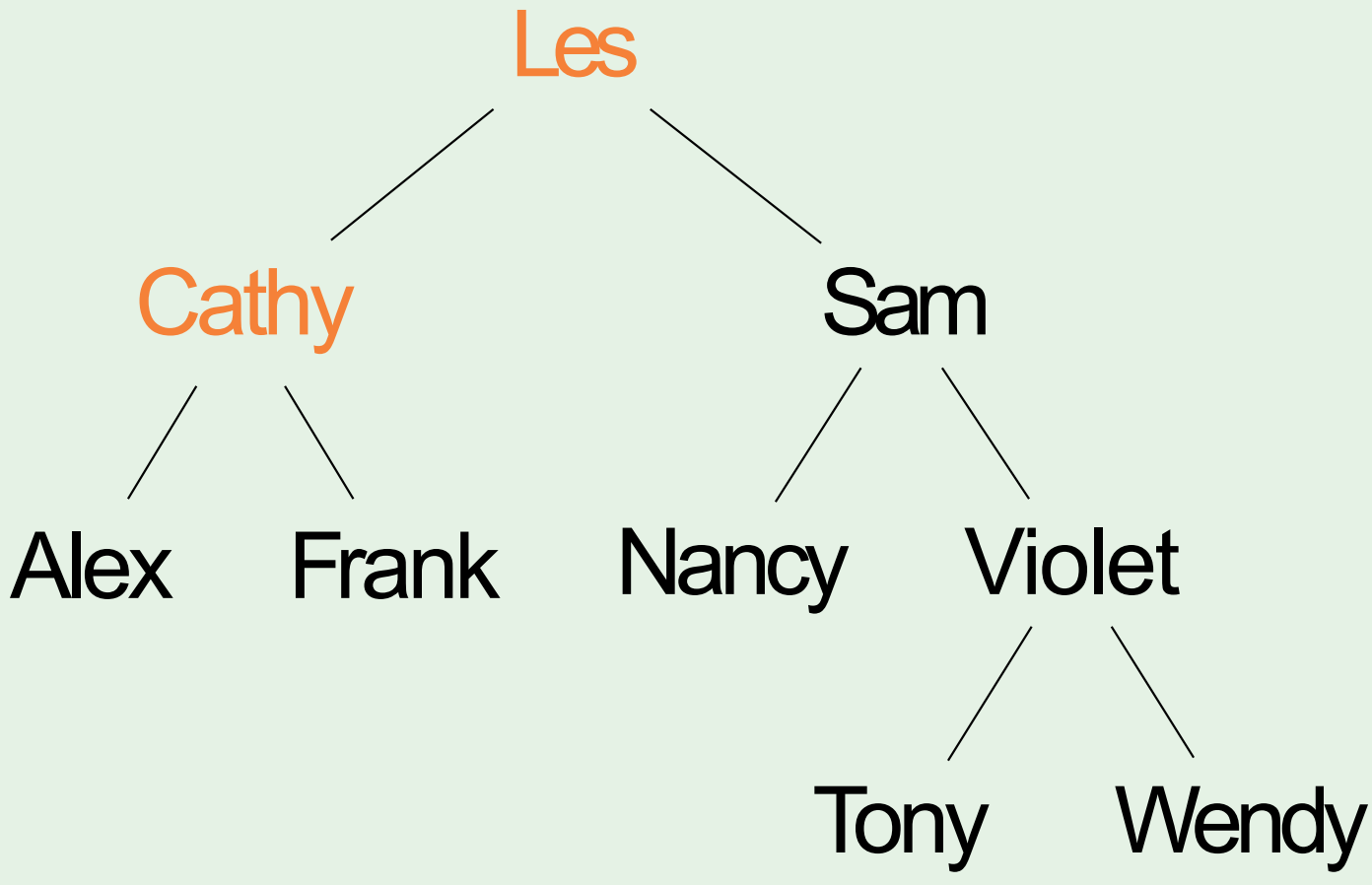
Output: Les

PreOrderTraversal



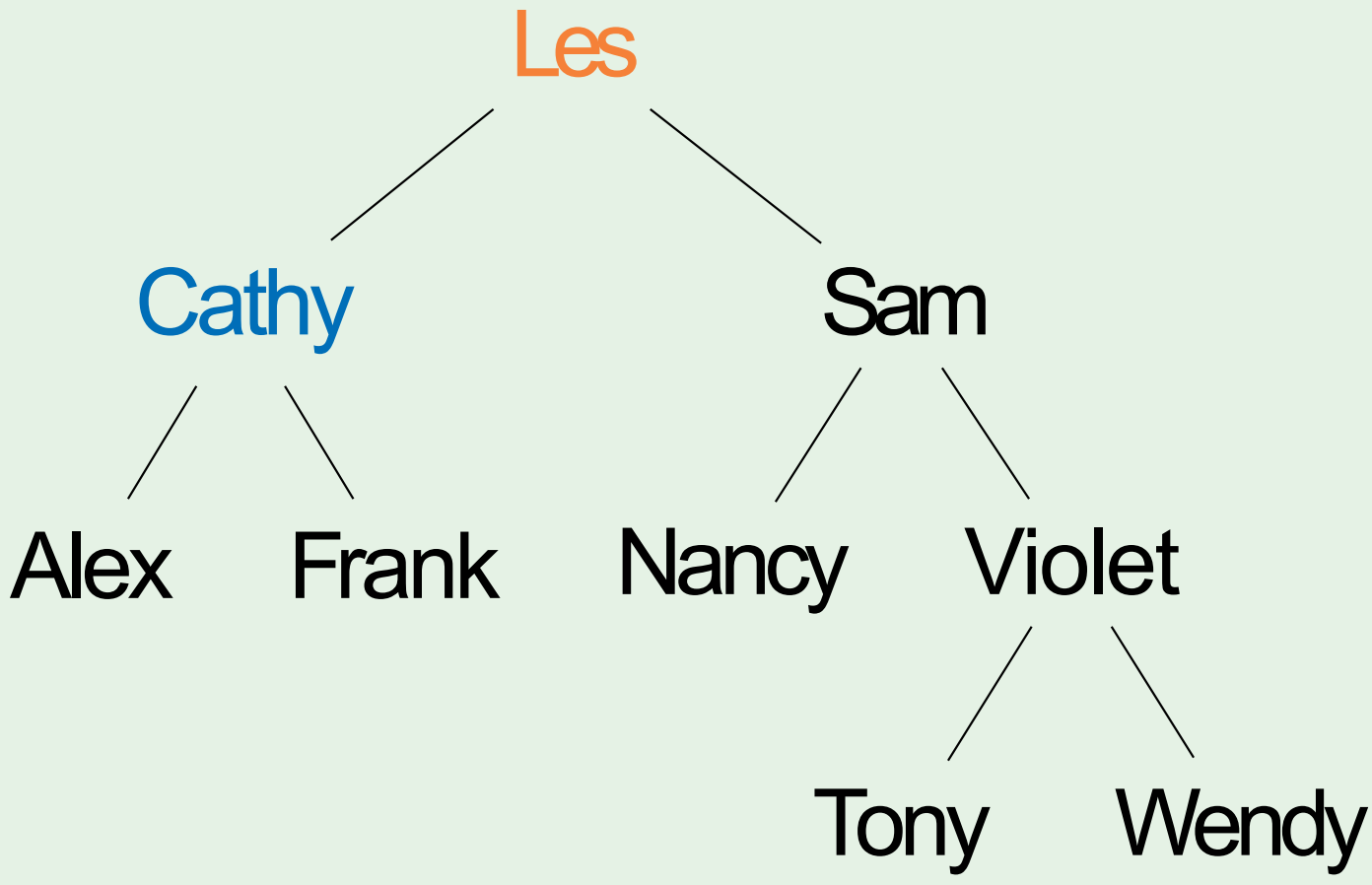
Output: Les

PreOrderTraversal



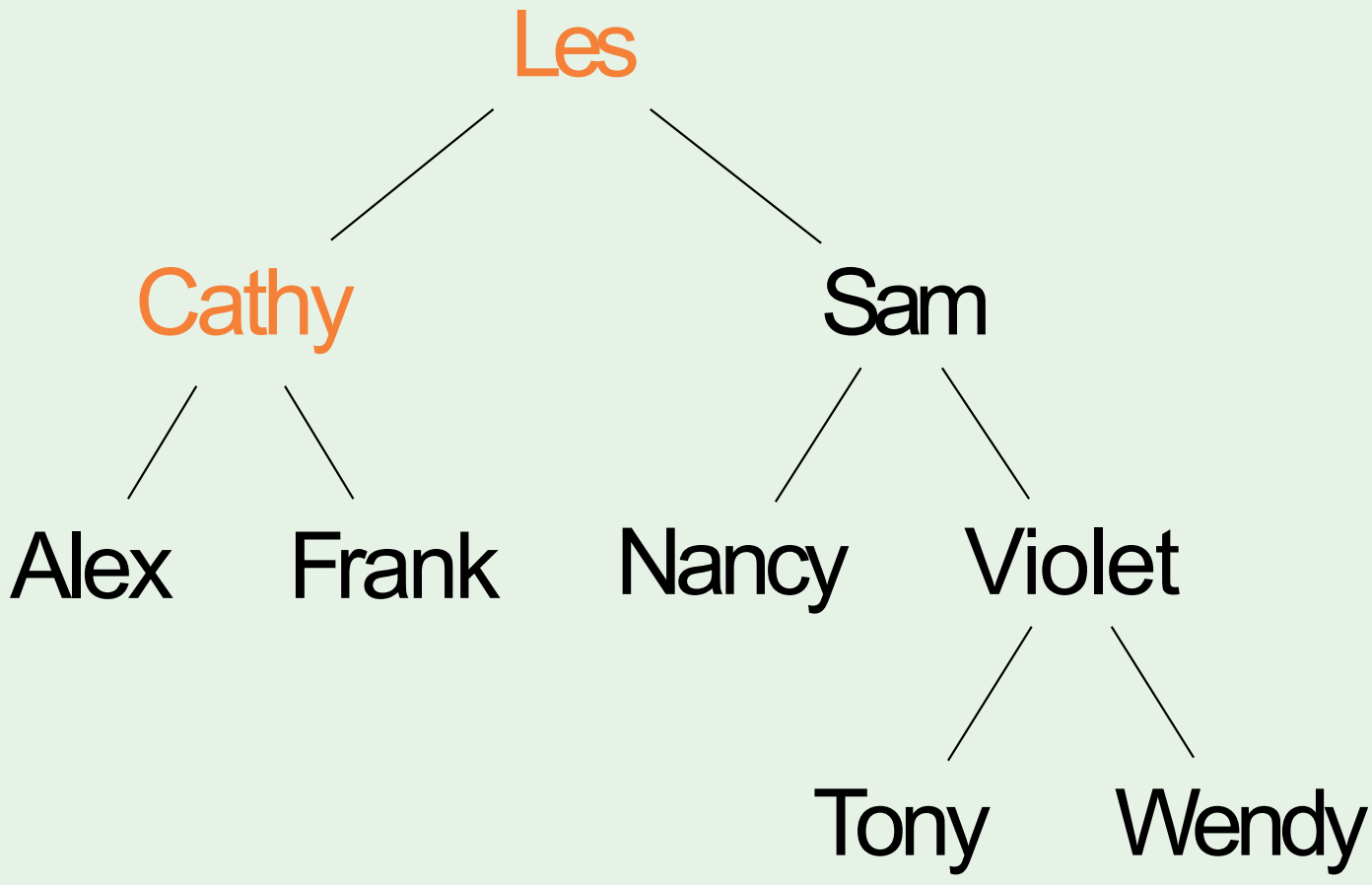
Output: Les

PreOrderTraversal



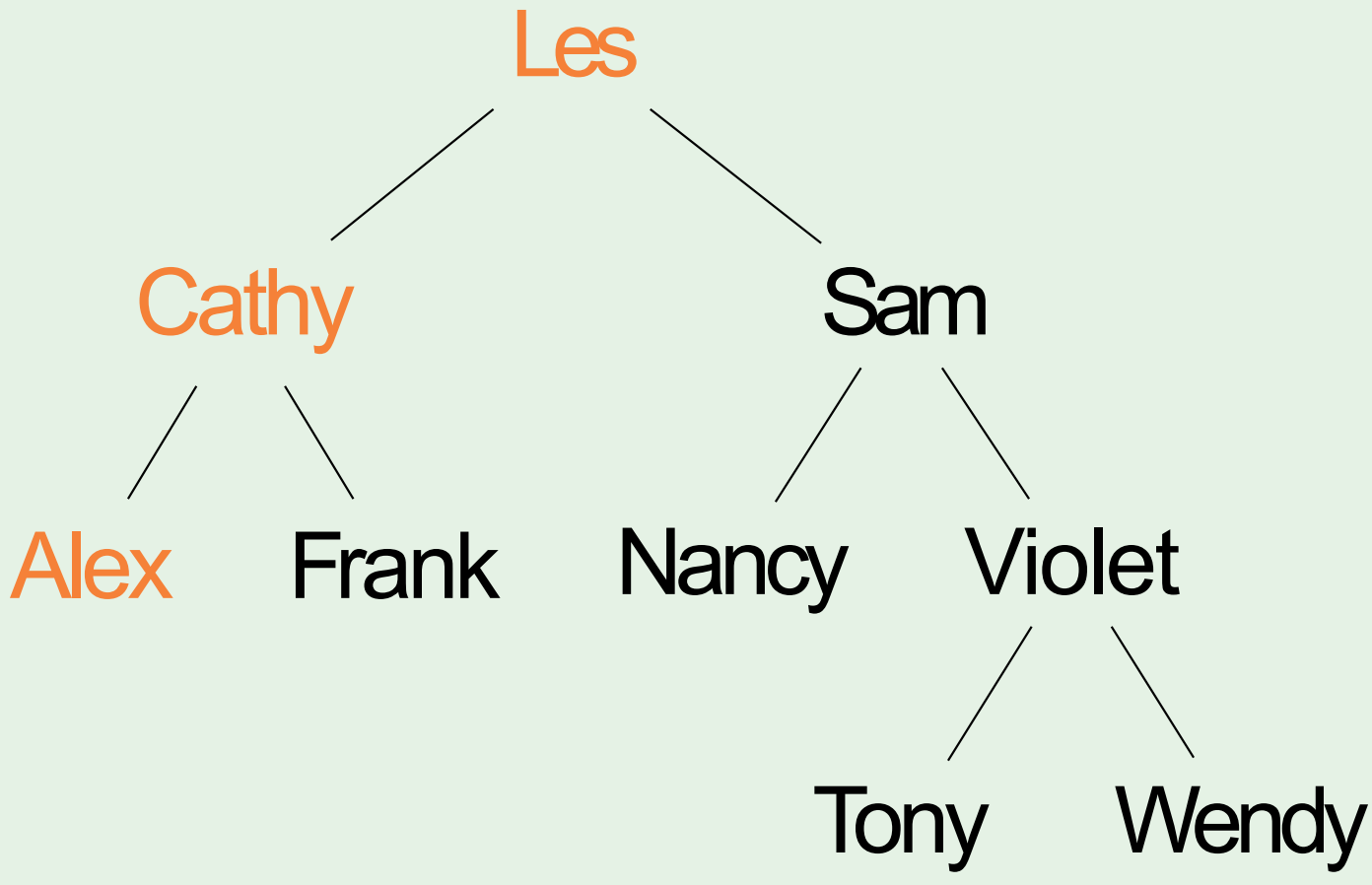
Output: Les Cathy

PreOrderTraversal



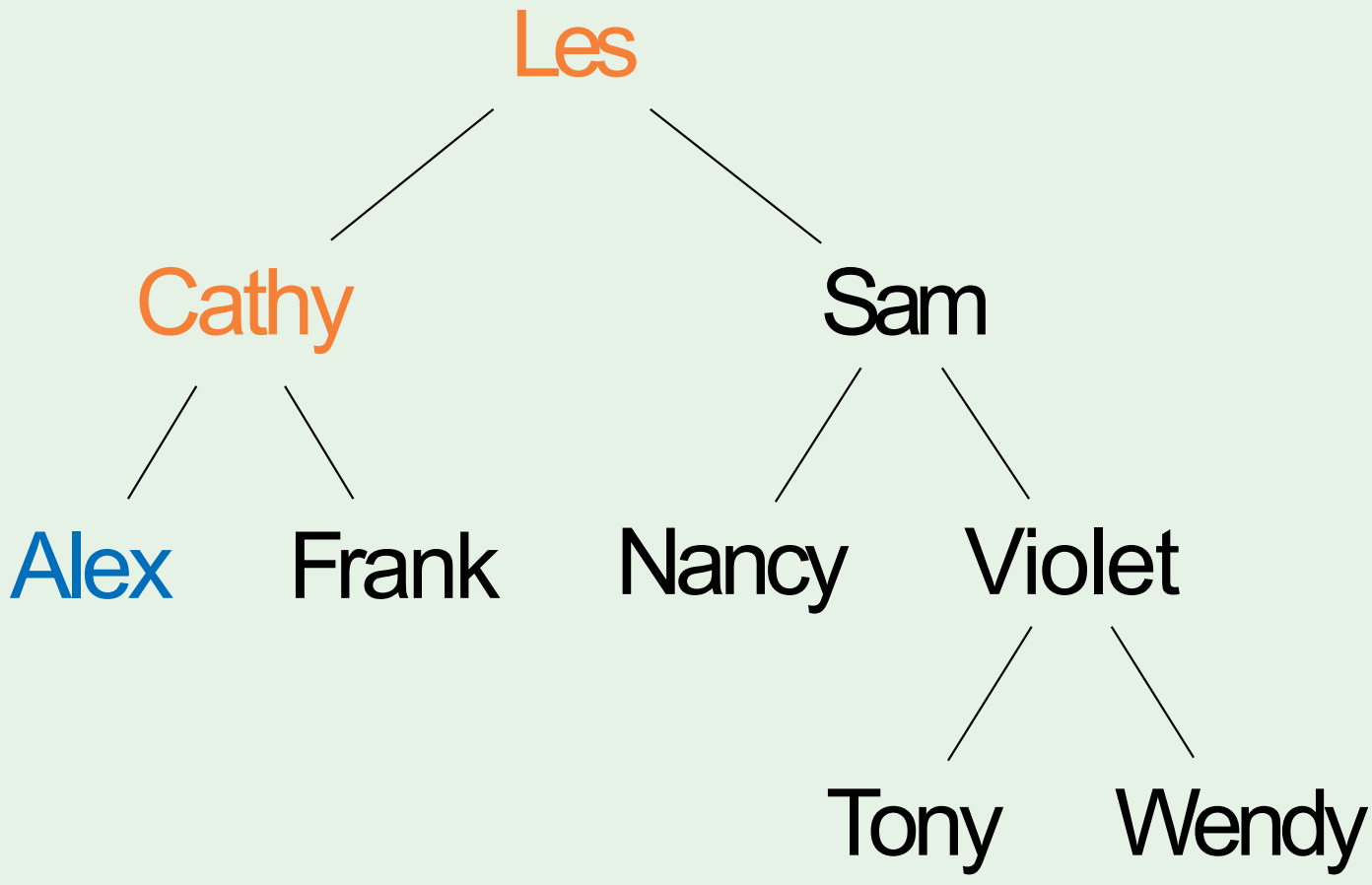
Output: Les Cathy

PreOrderTraversal



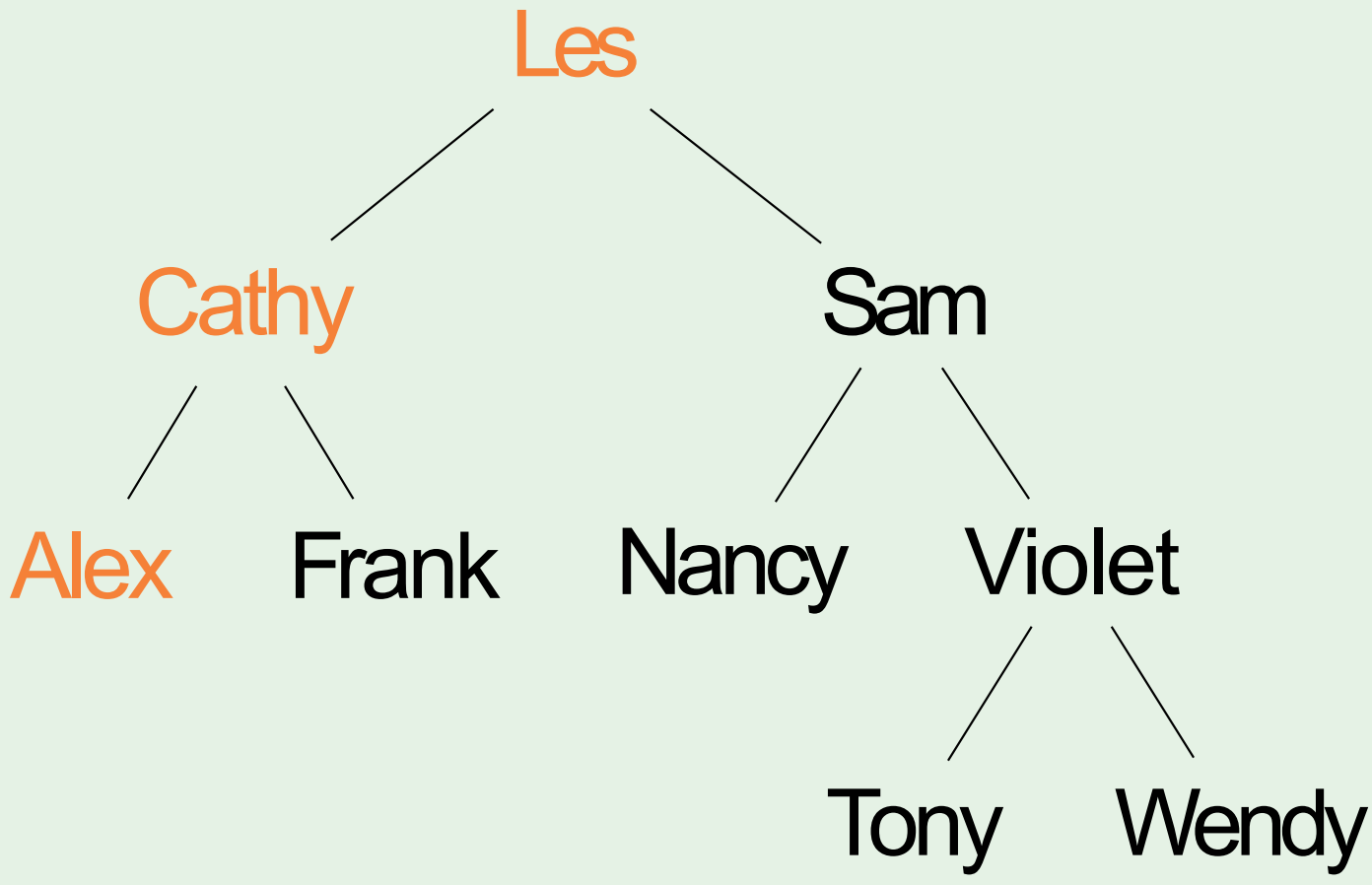
Output: LesCathy

PreOrderTraversal



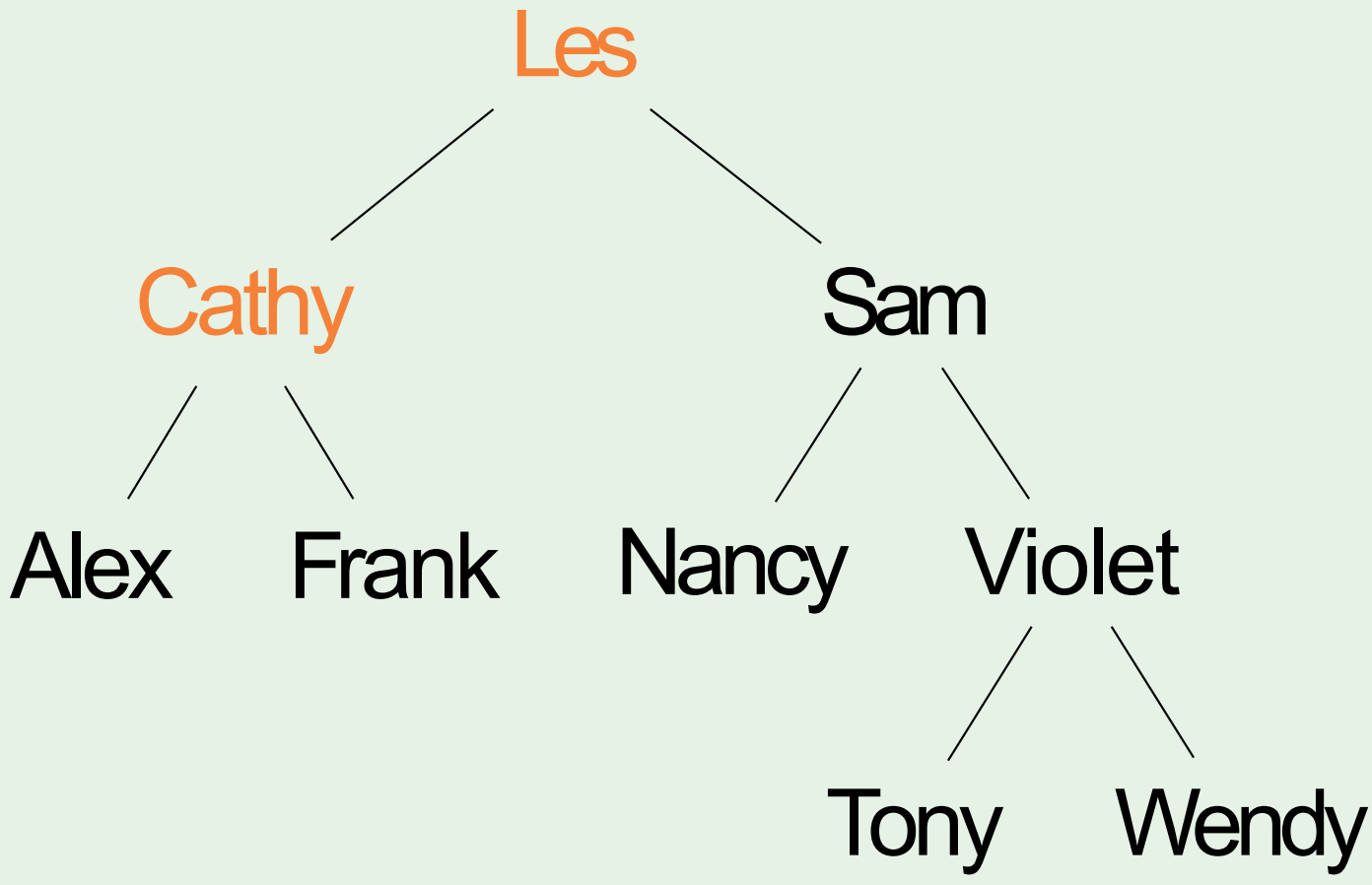
Output: Les Cathy Alex

PreOrderTraversal



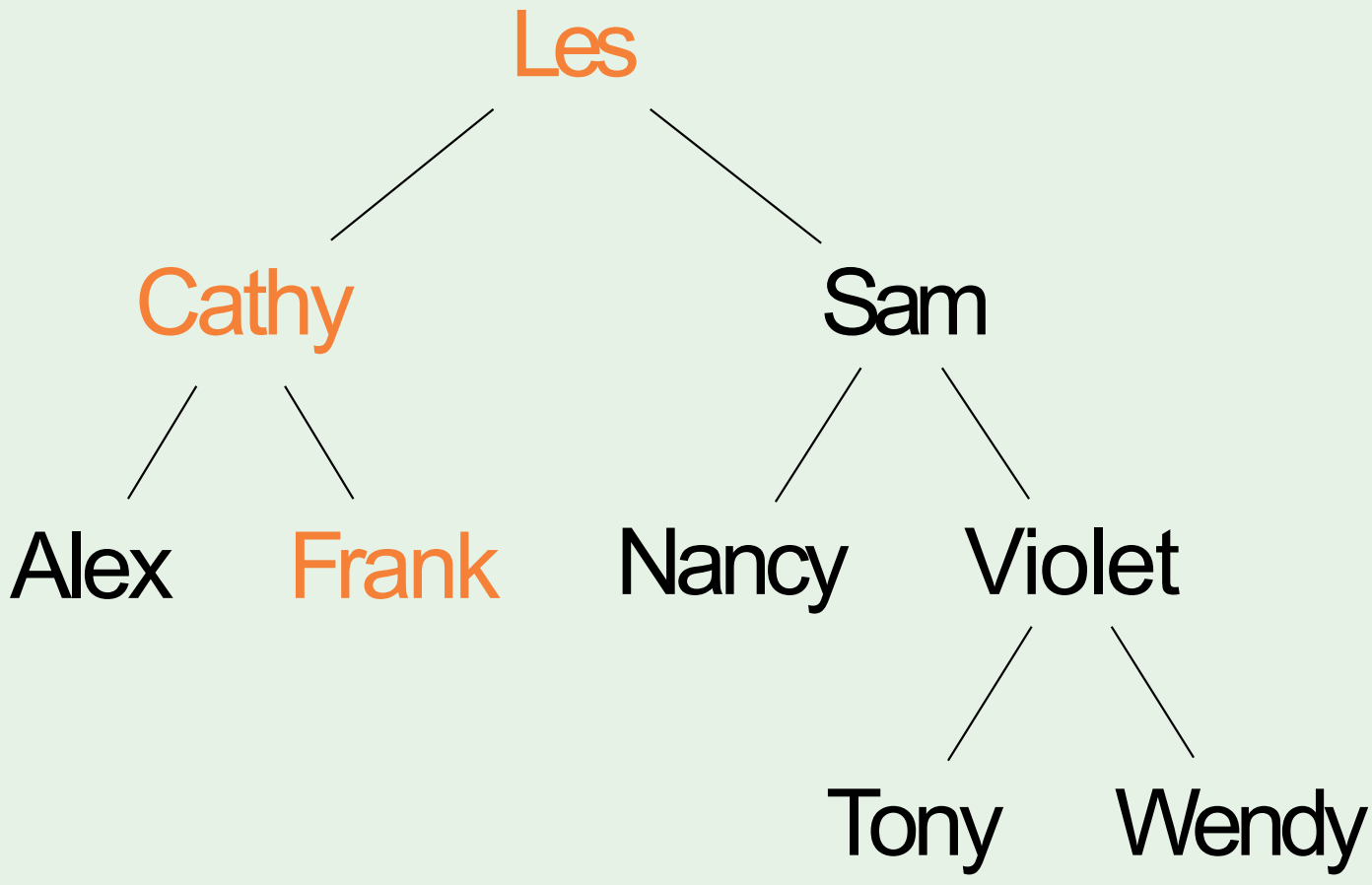
Output: Les Cathy Alex

PreOrderTraversal



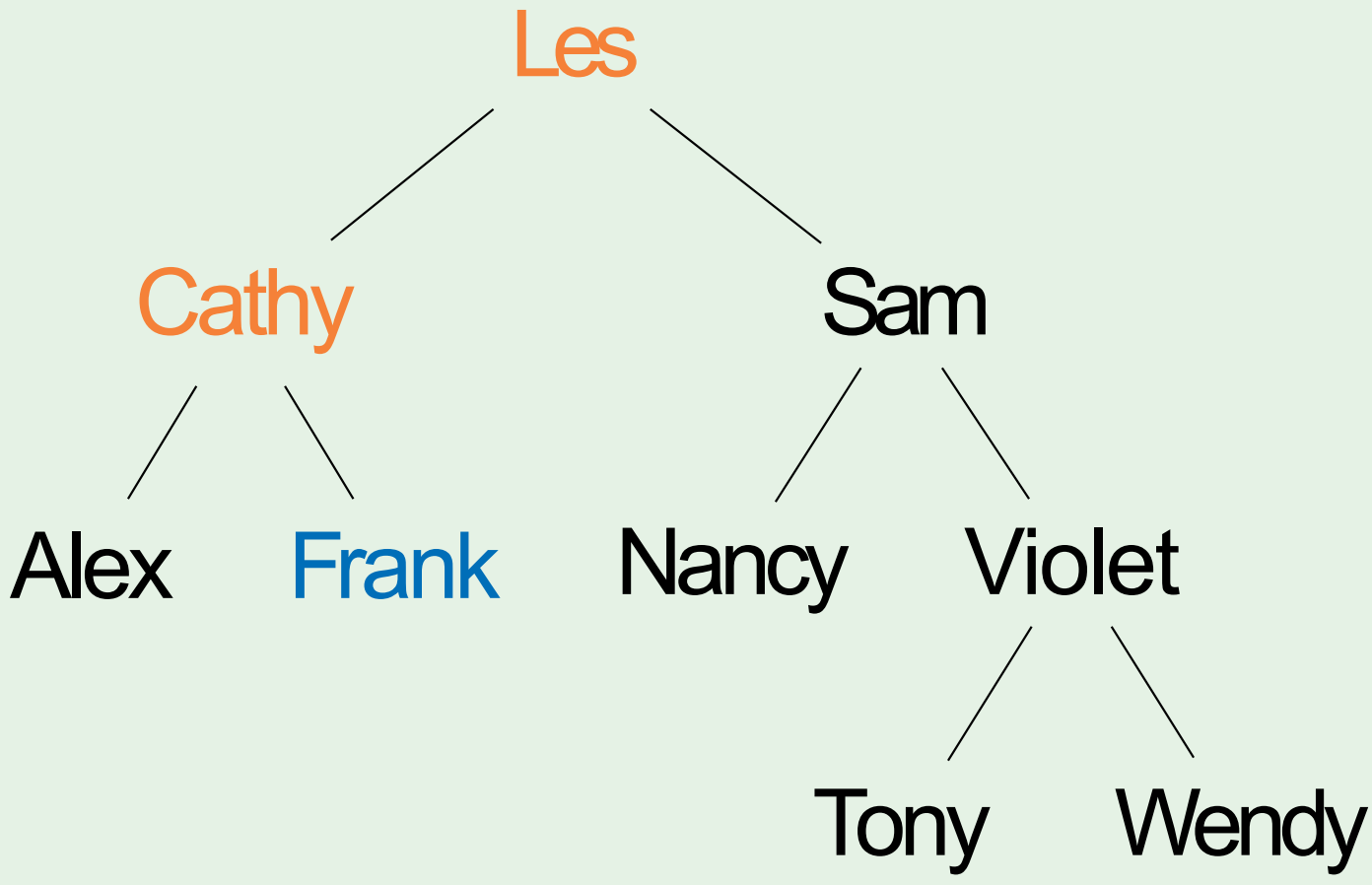
Output: Les Cathy Alex

PreOrderTraversal



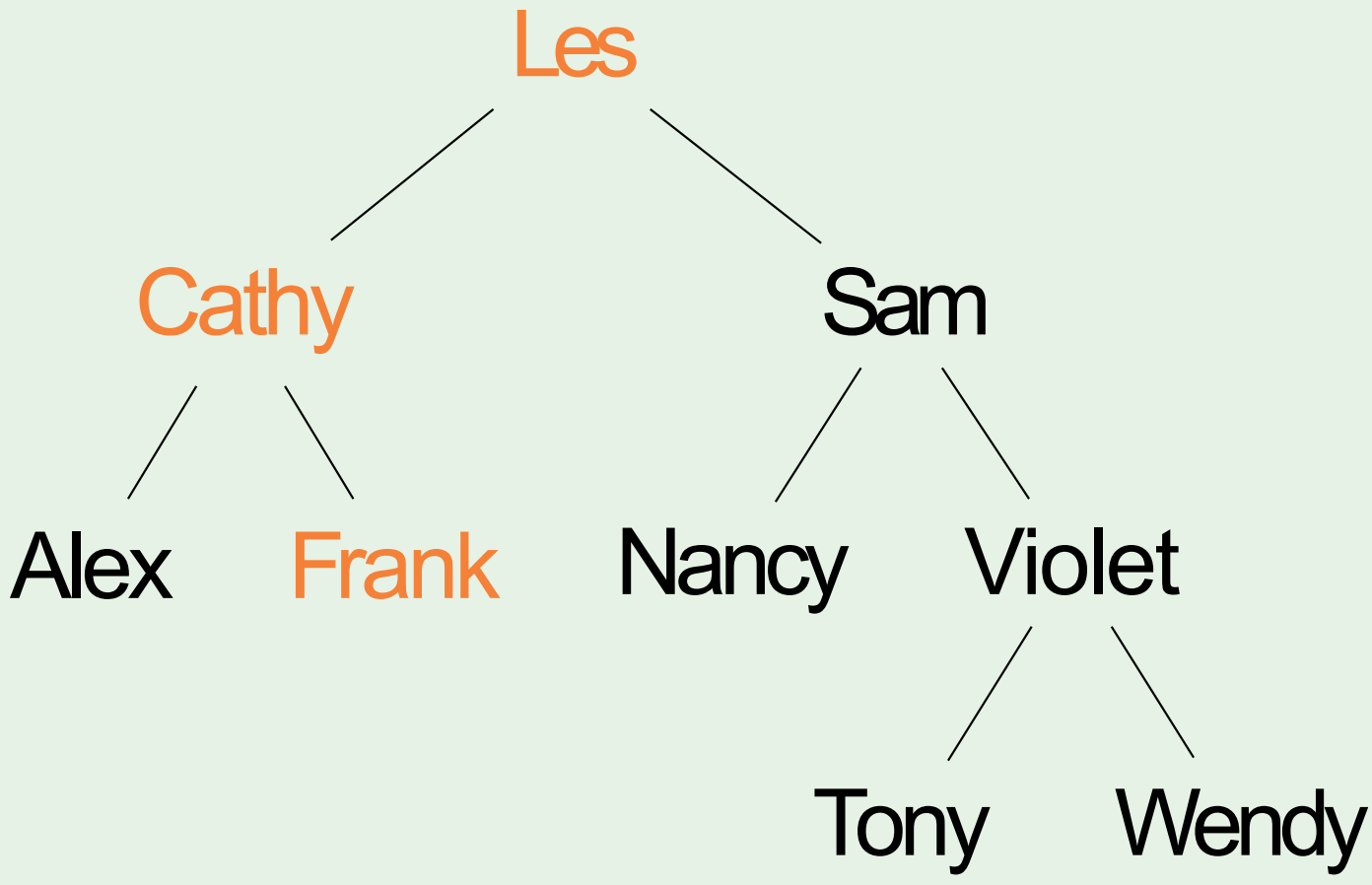
Output: Les Cathy Alex

PreOrderTraversal



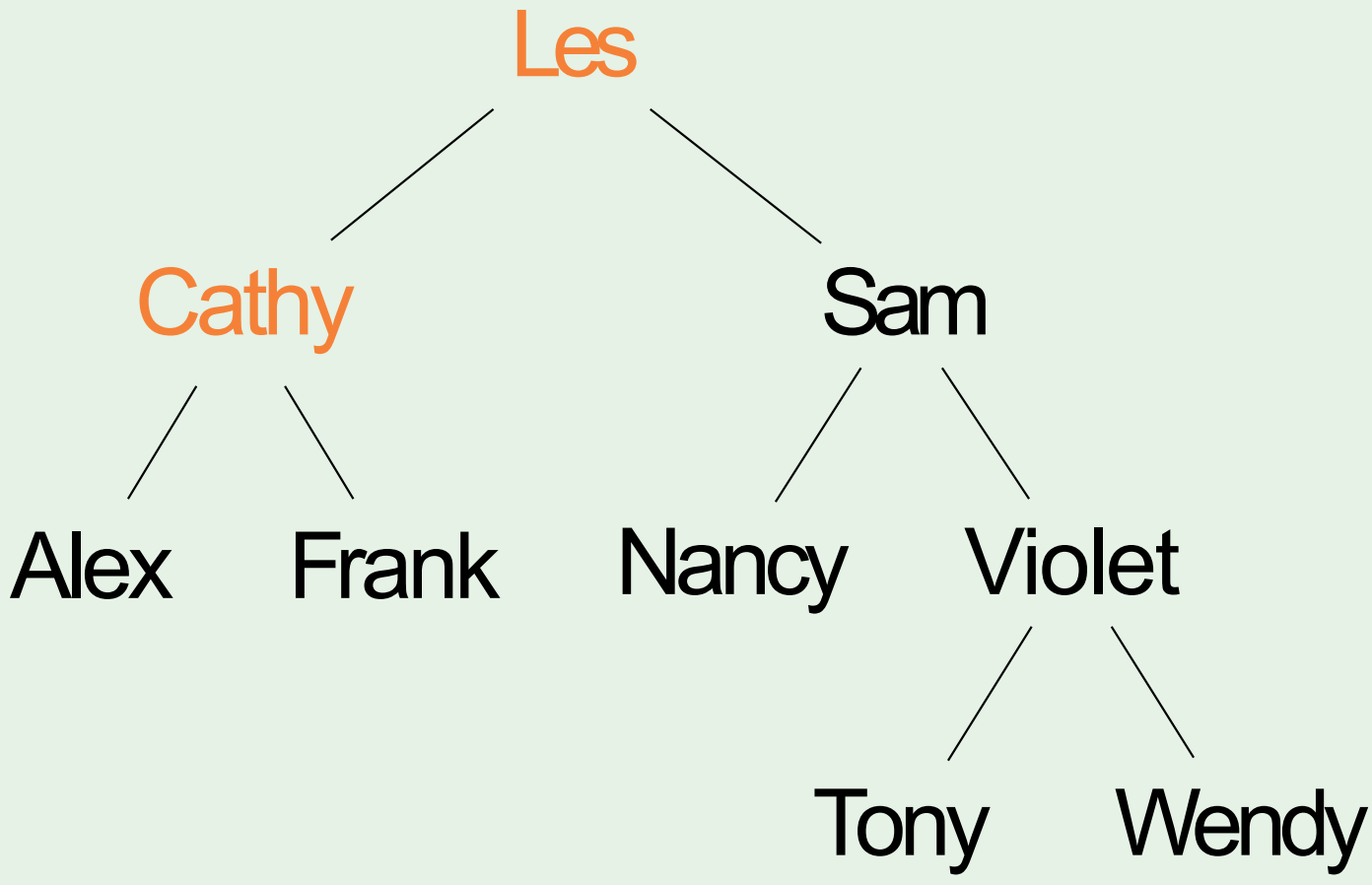
Output: Les Cathy Alex Frank

PreOrderTraversal



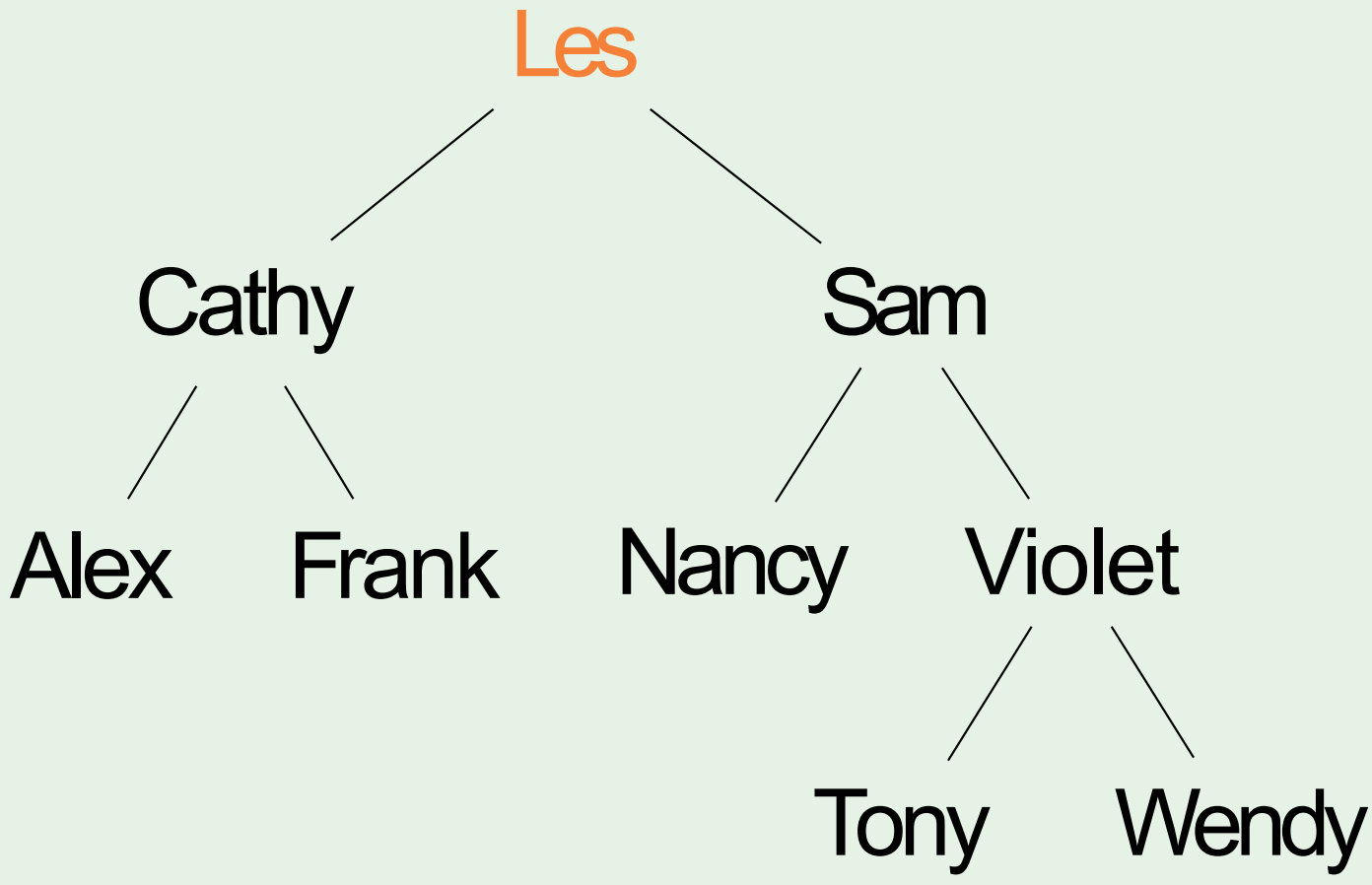
Output: Les Cathy Alex Frank

PreOrderTraversal



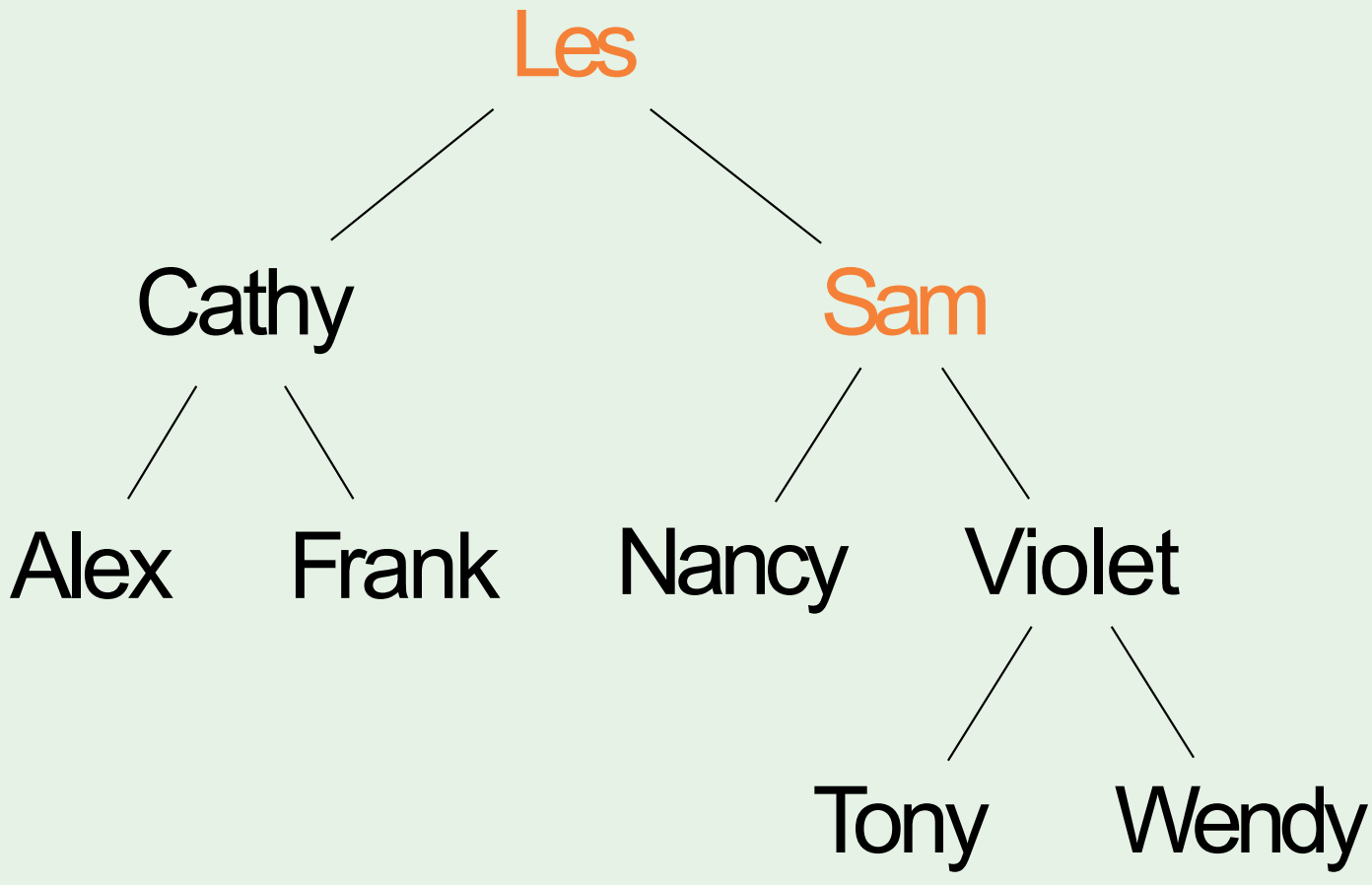
Output: Les Cathy Alex Frank

PreOrderTraversal



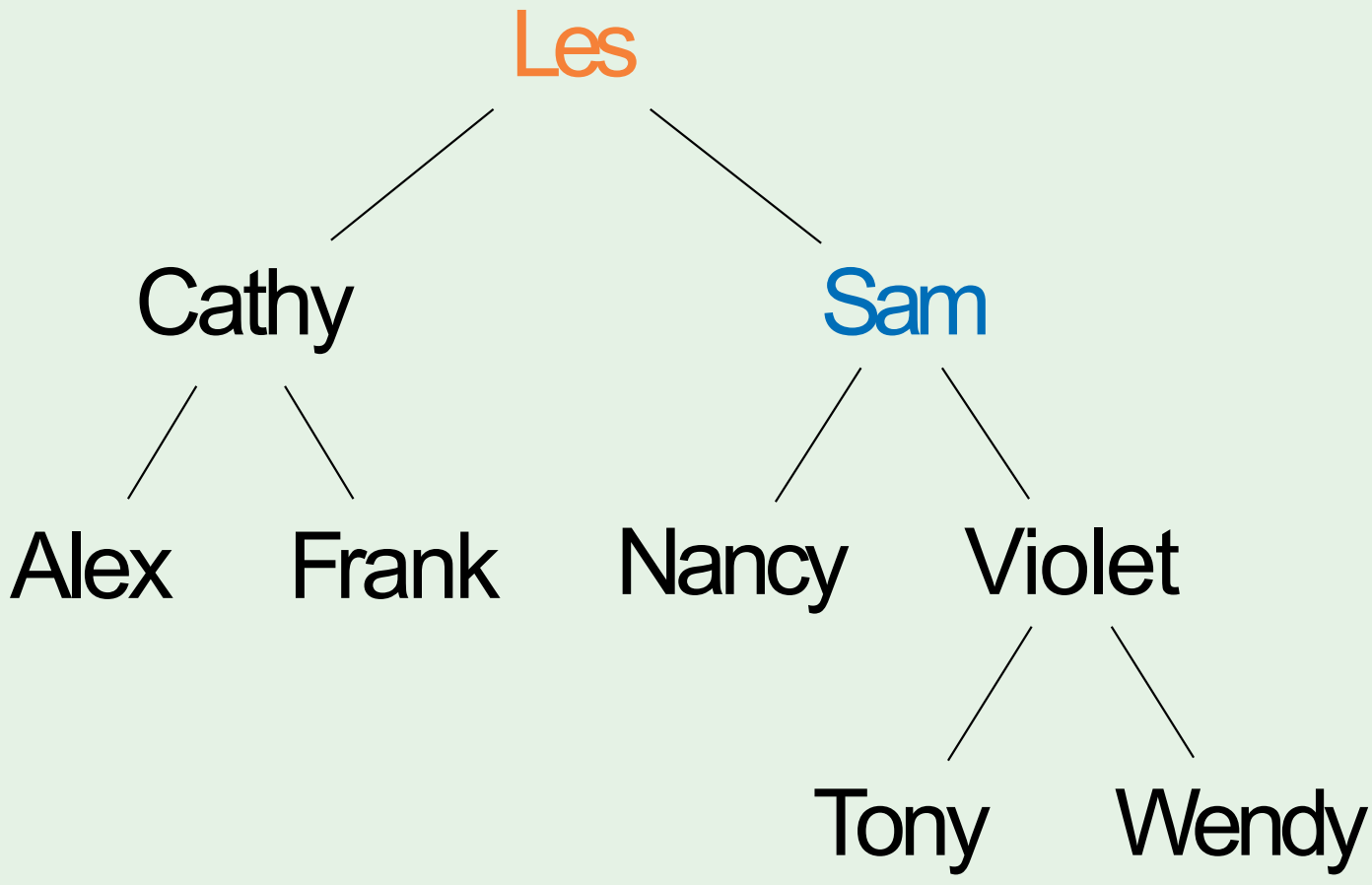
Output: Les Cathy Alex Frank

PreOrderTraversal



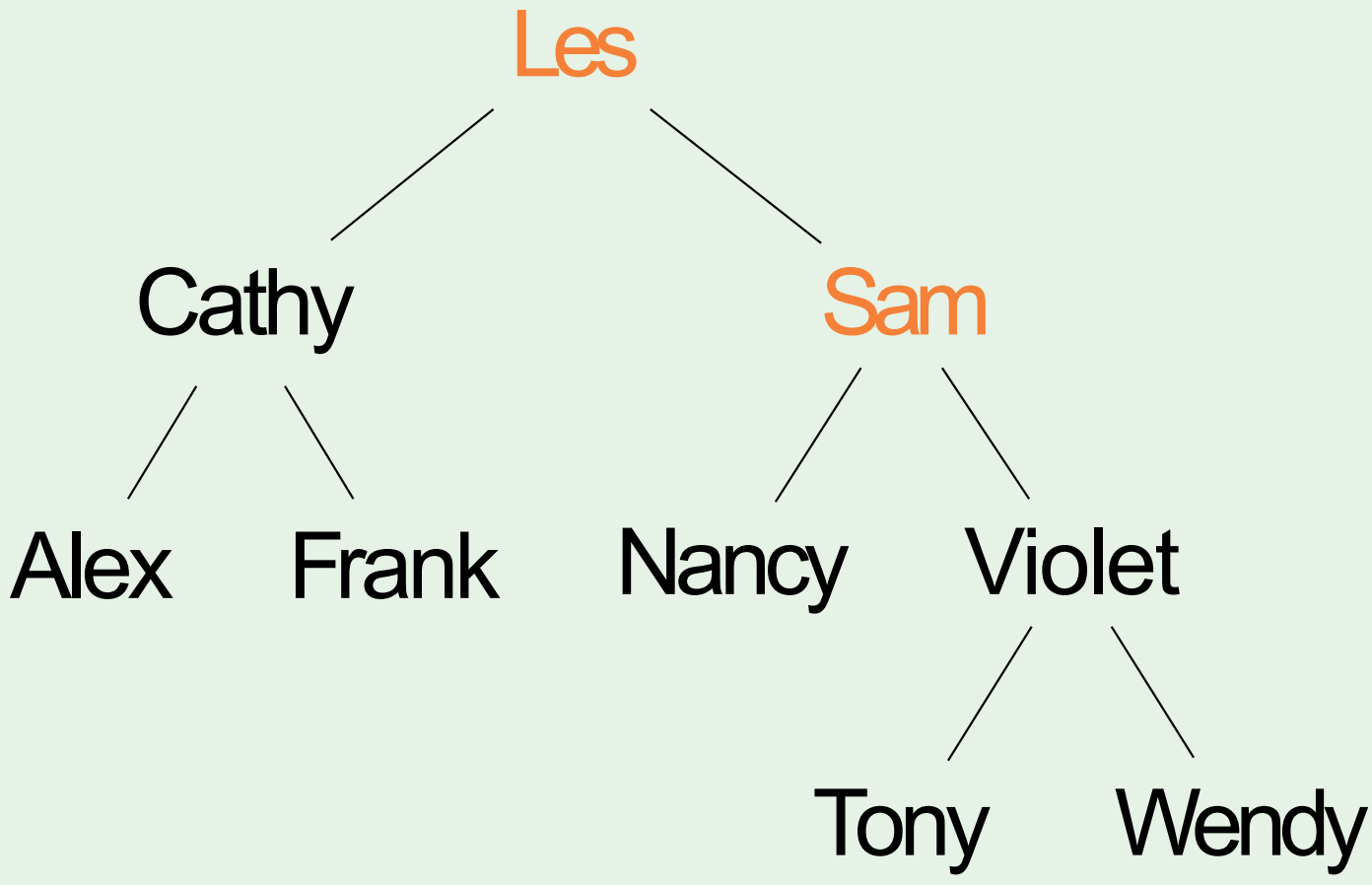
Output: Les Cathy Alex Frank

PreOrderTraversal



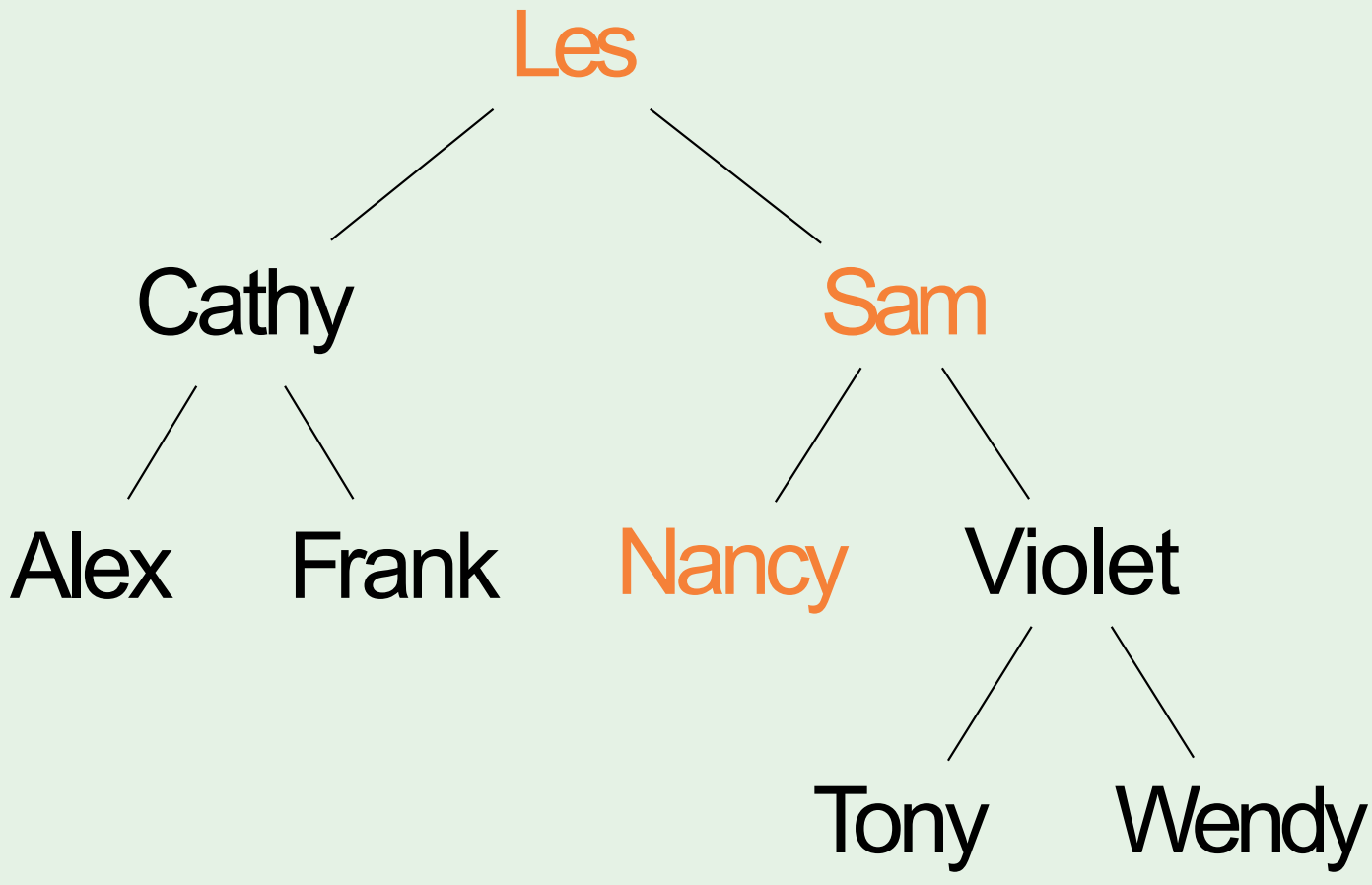
Output: Les Cathy Alex Frank Sam

PreOrderTraversal



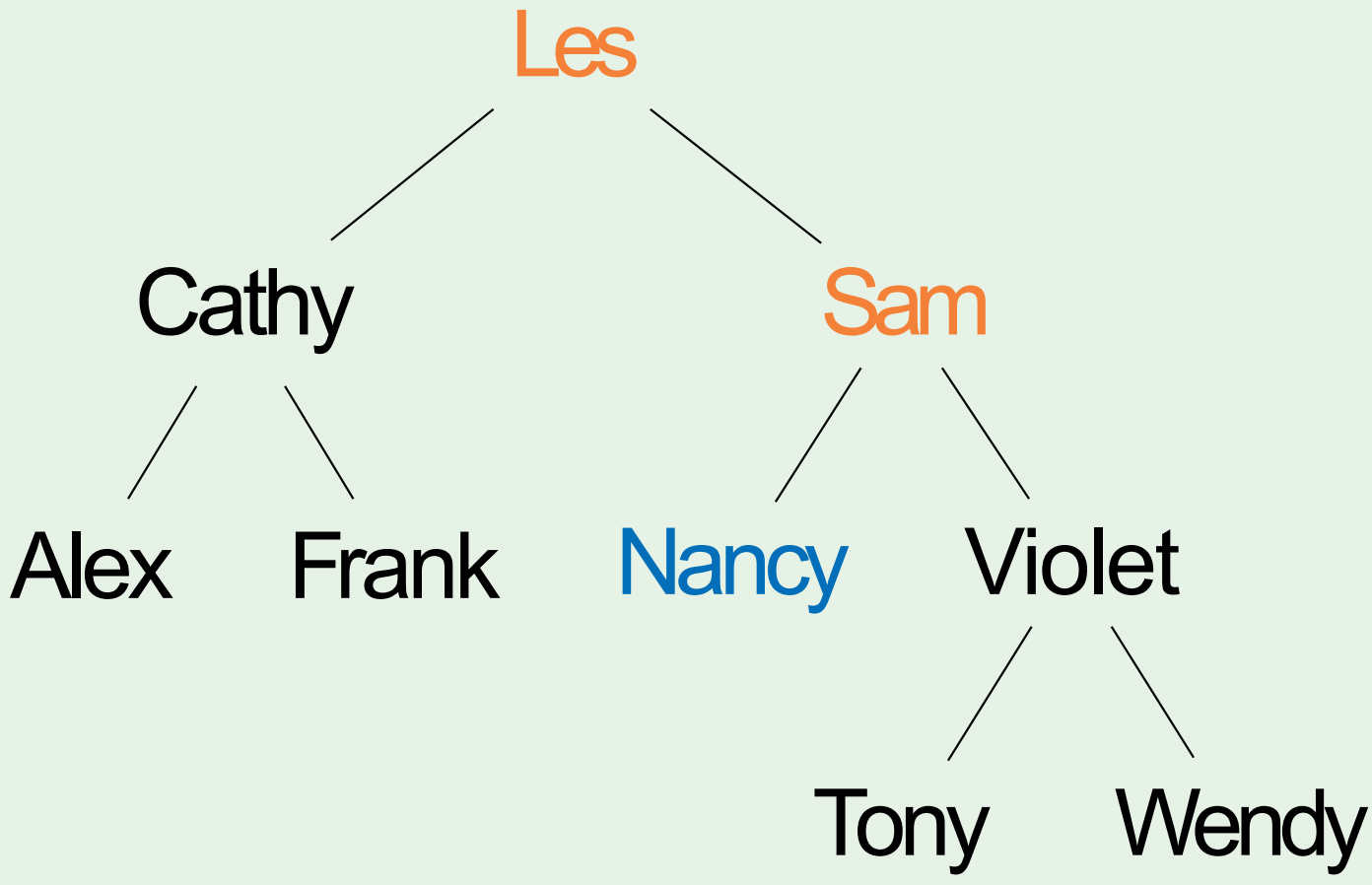
Output: Les Cathy Alex Frank Sam

PreOrderTraversal



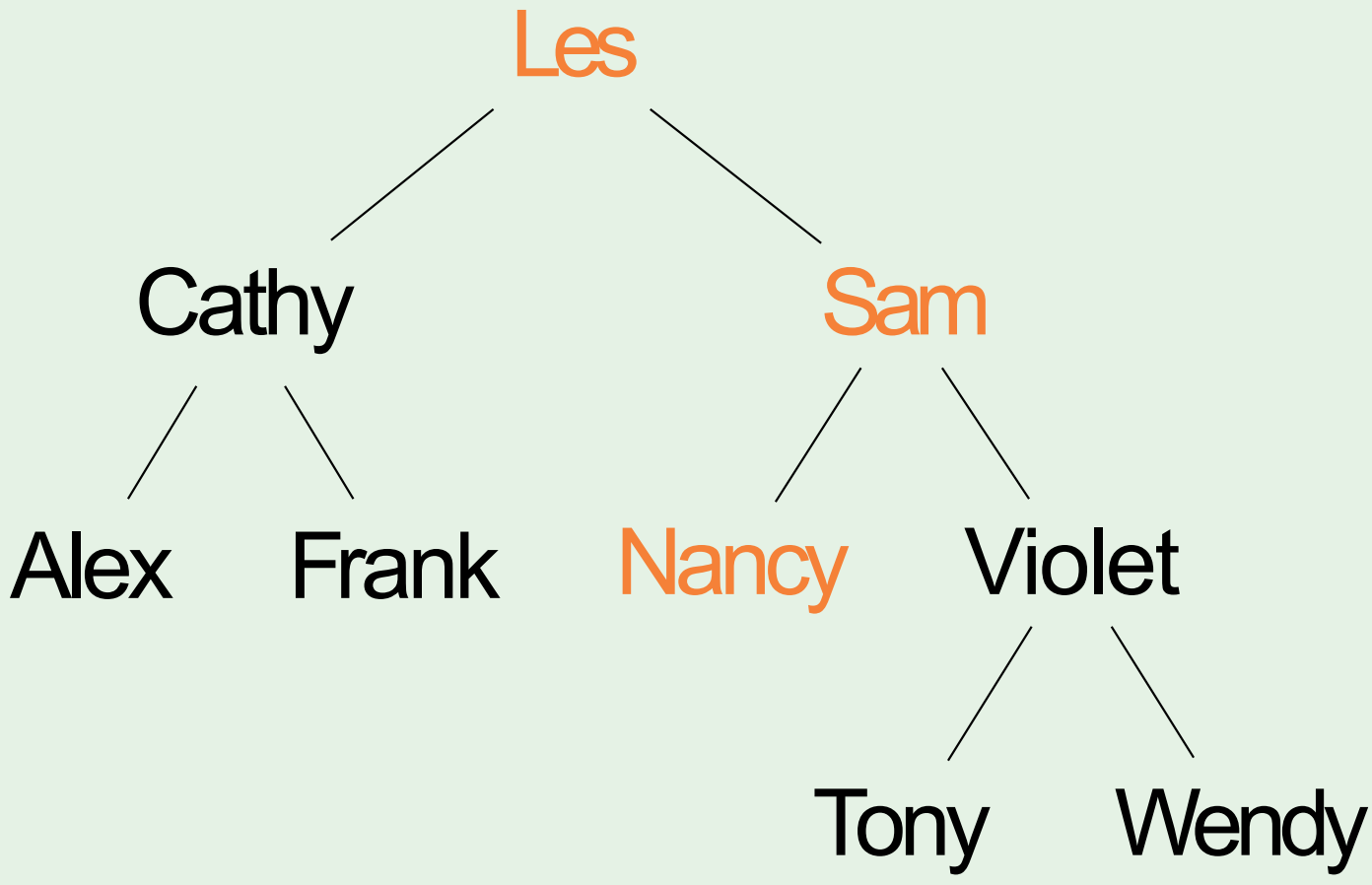
Output: Les Cathy Alex Frank Sam

PreOrderTraversal



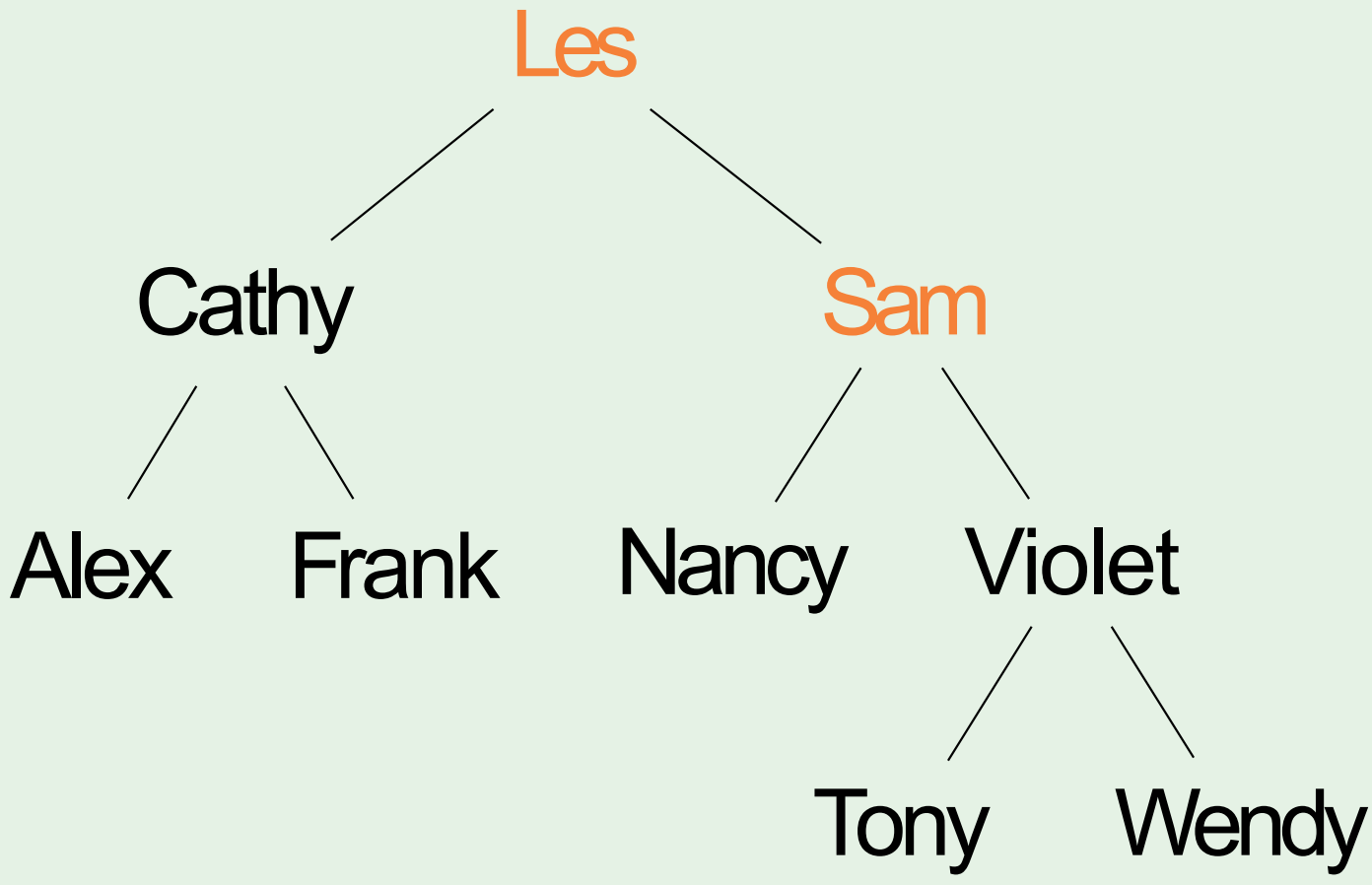
Output: Les Cathy Alex Frank Sam Nancy

PreOrderTraversal



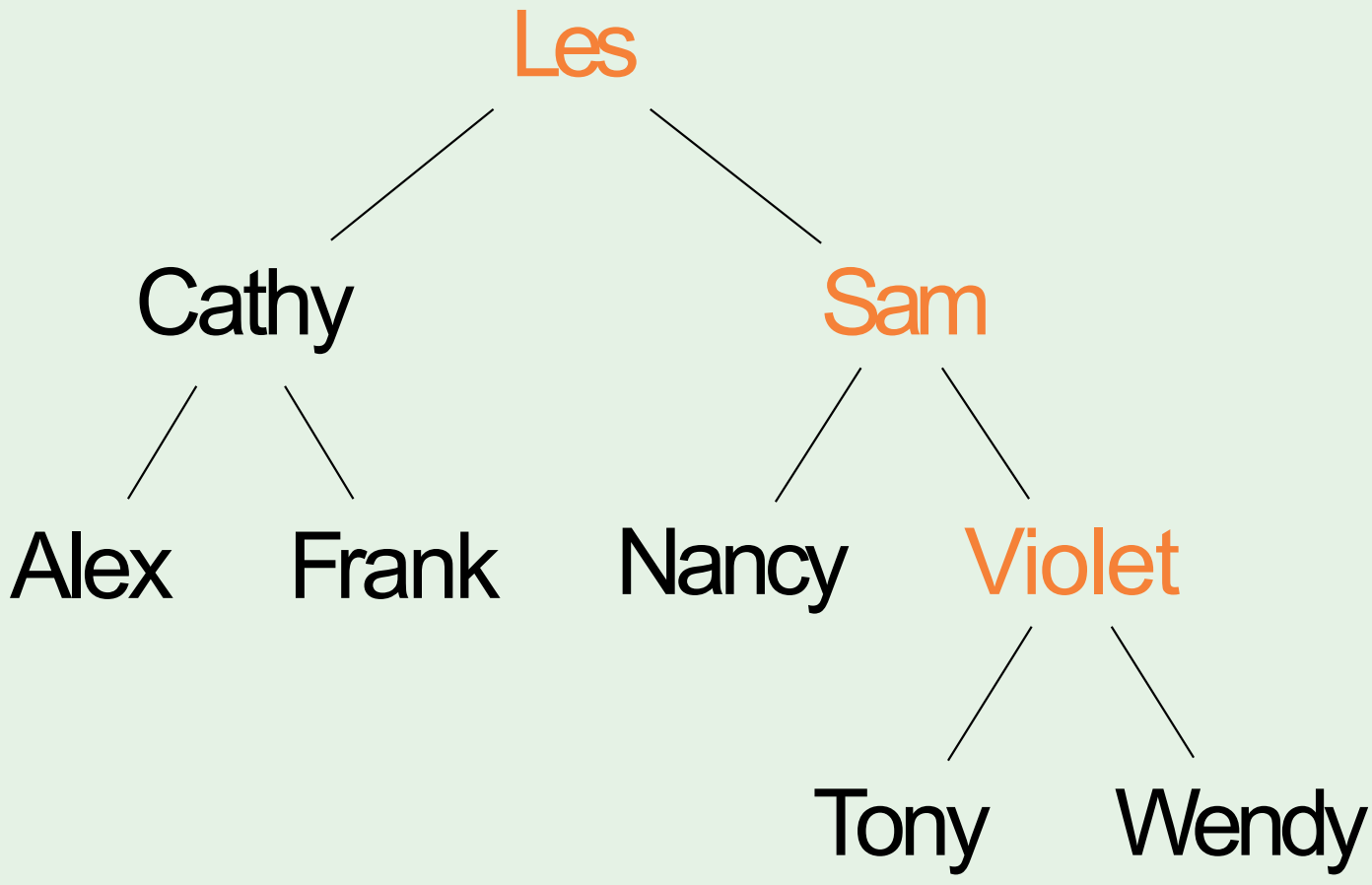
Output: Les Cathy Alex Frank Sam Nancy

PreOrderTraversal



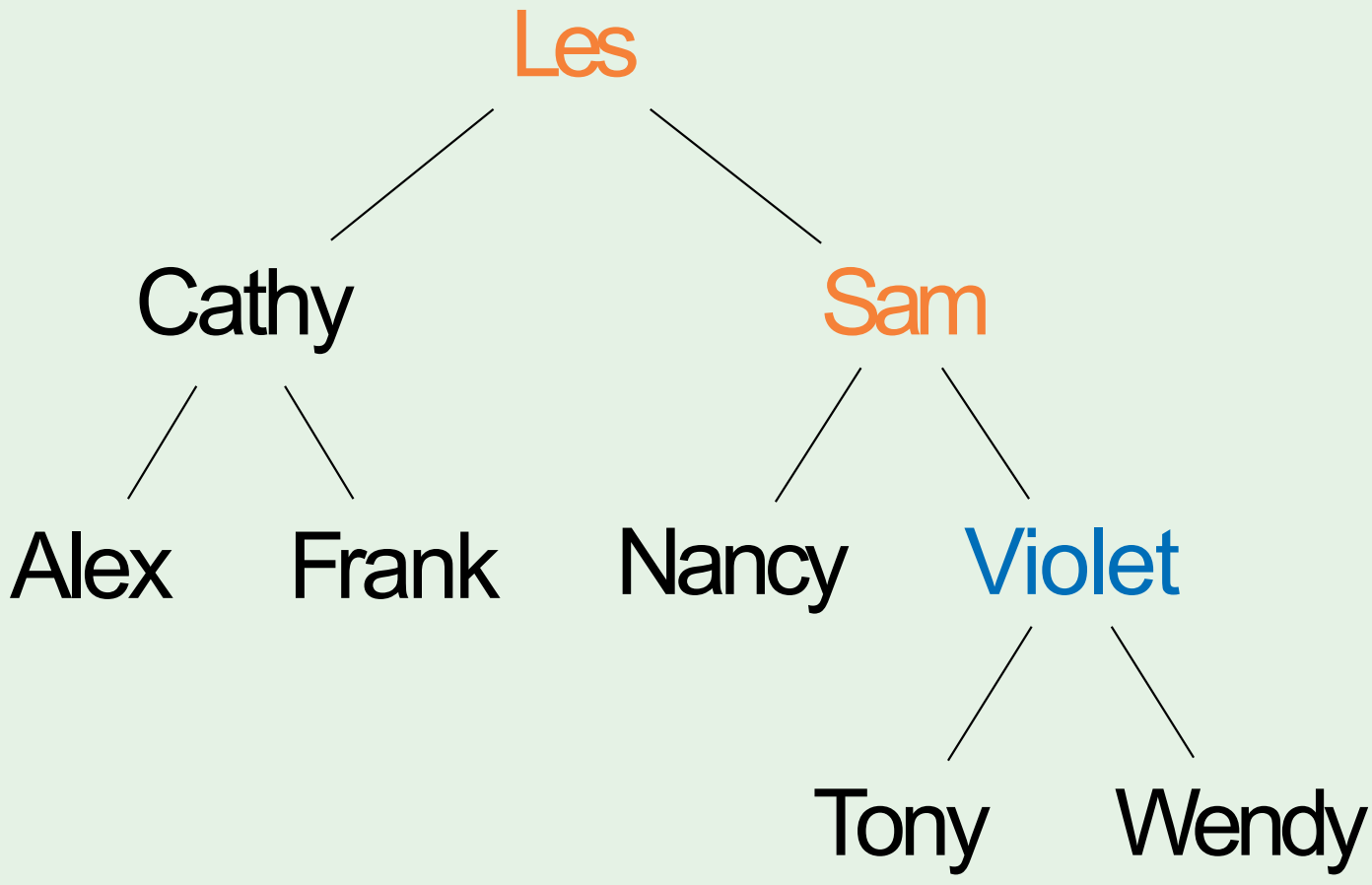
Output: Les Cathy Alex Frank Sam Nancy

PreOrderTraversal



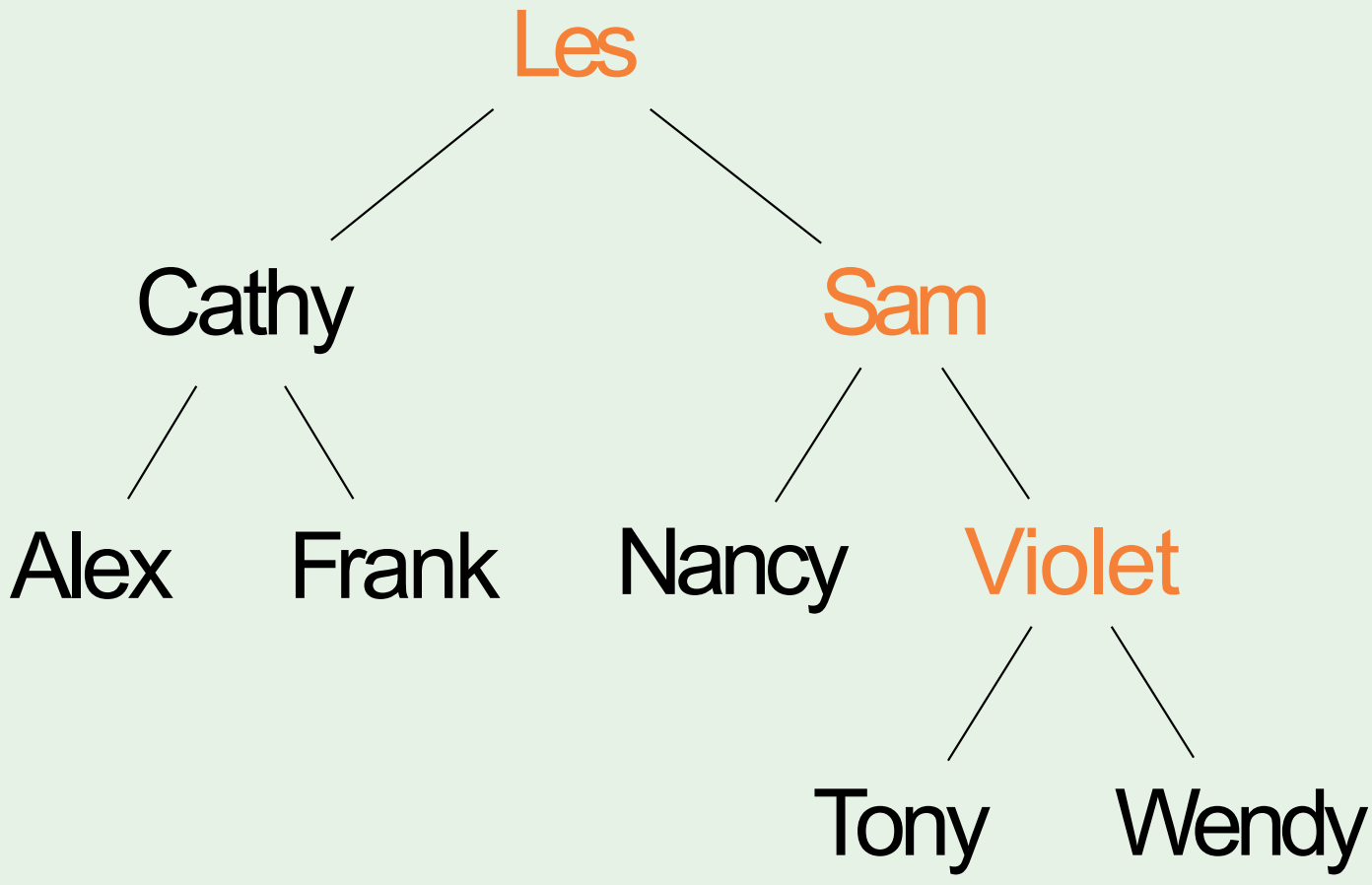
Output: Les Cathy Alex Frank Sam Nancy

PreOrderTraversal



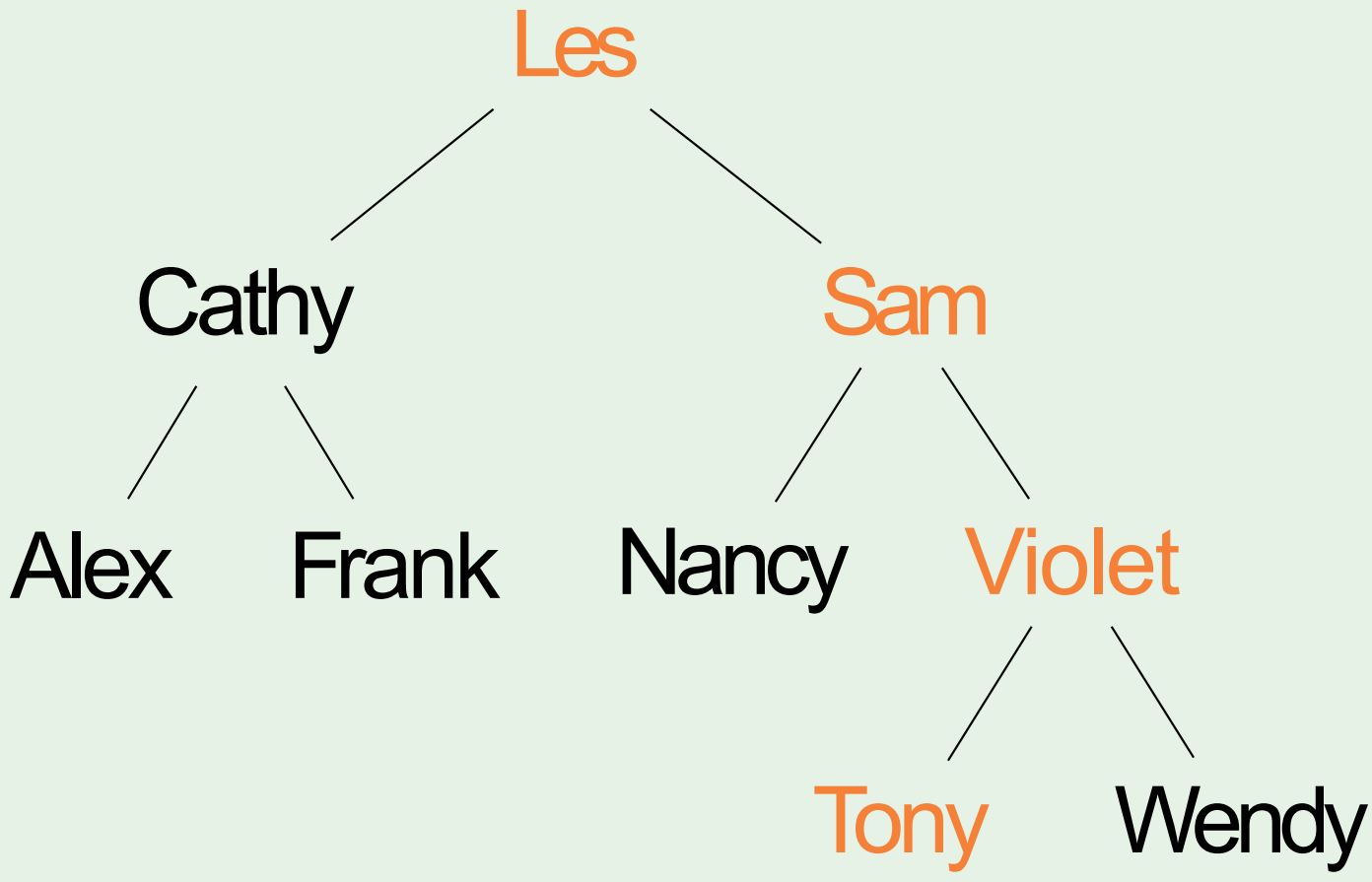
Output: Les Cathy Alex Frank Sam Nancy Violet

PreOrderTraversal



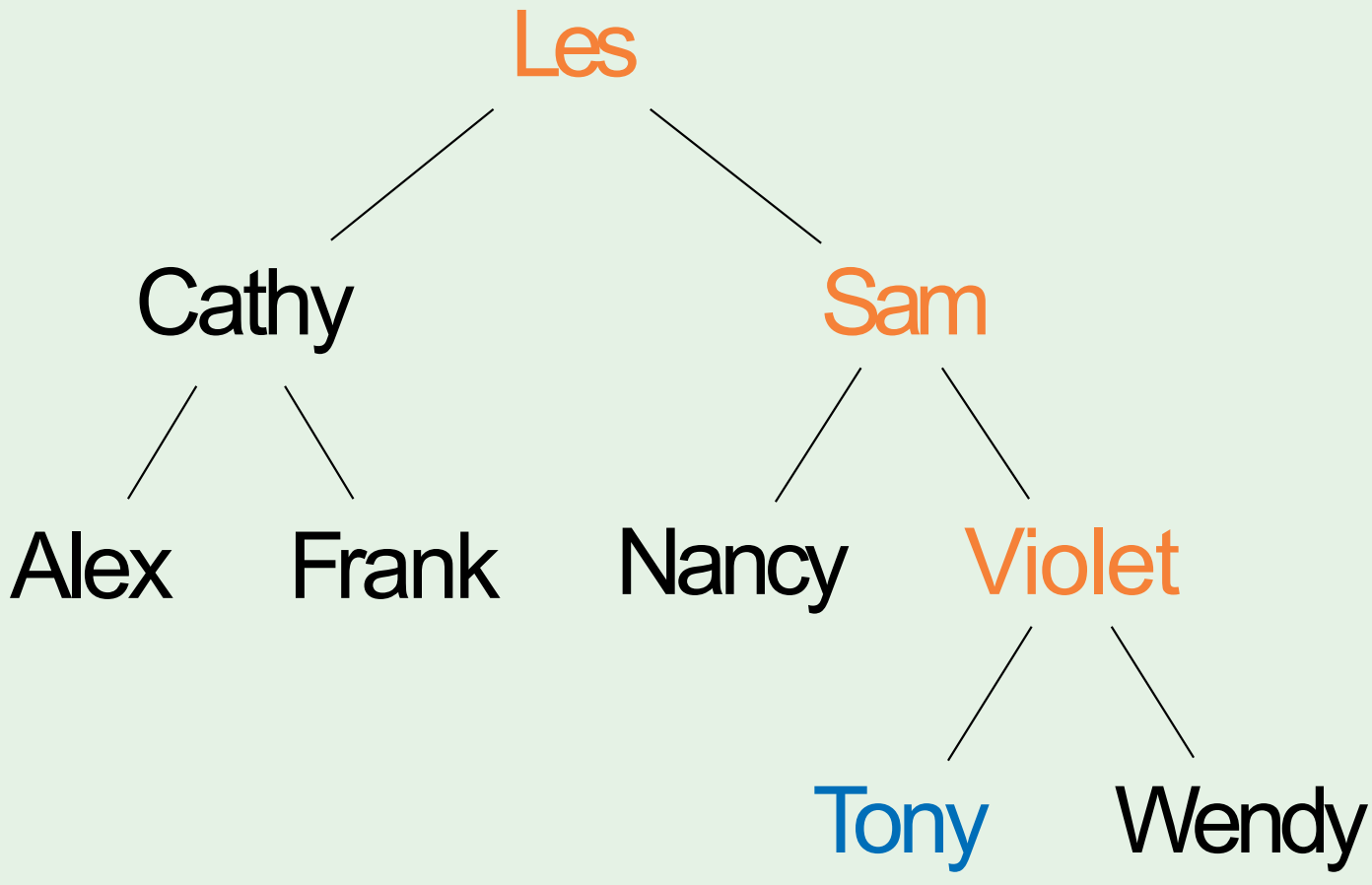
Output: Les Cathy Alex Frank Sam Nancy Violet

PreOrderTraversal



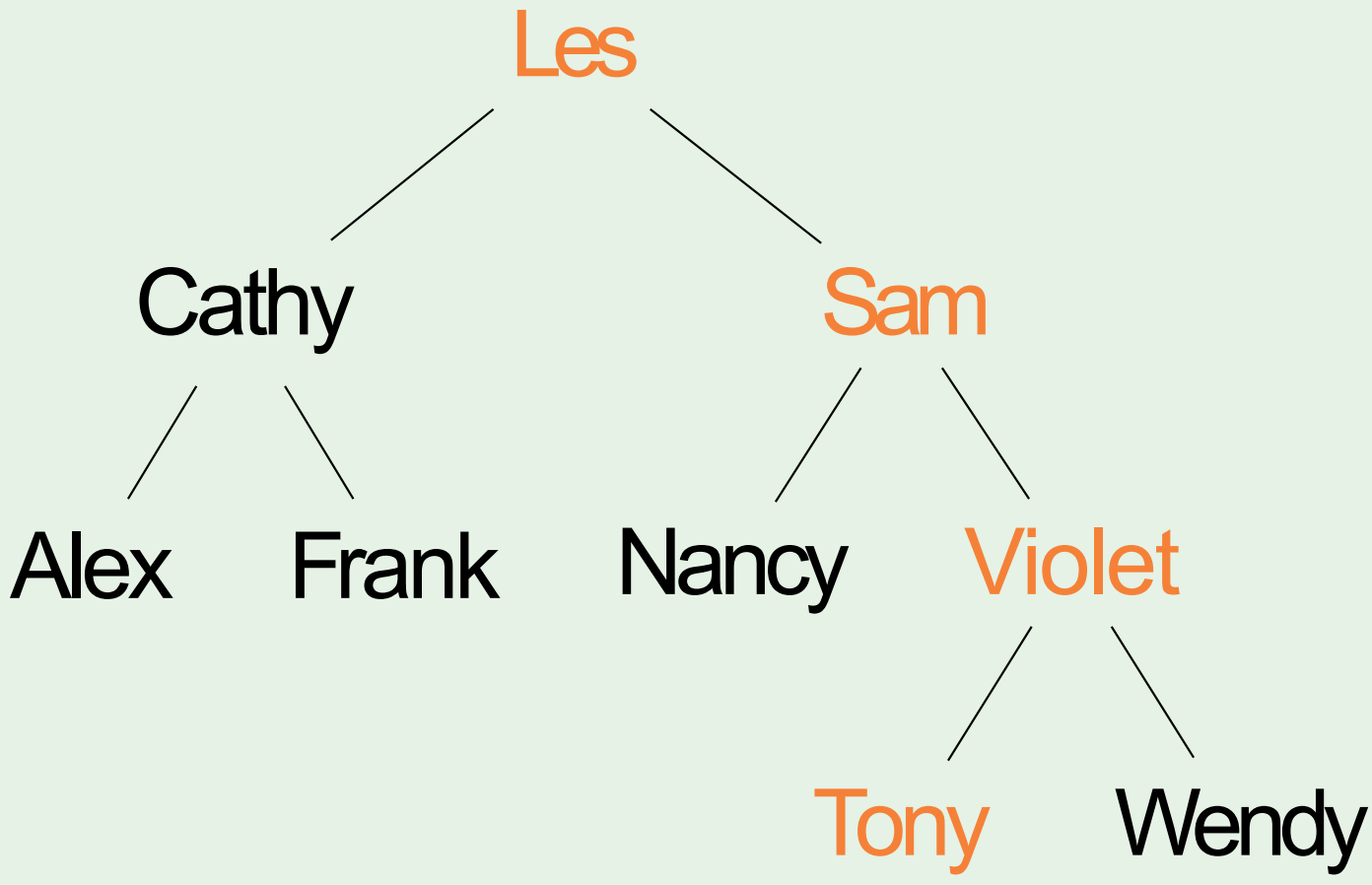
Output: Les Cathy Alex Frank Sam Nancy Violet

PreOrderTraversal



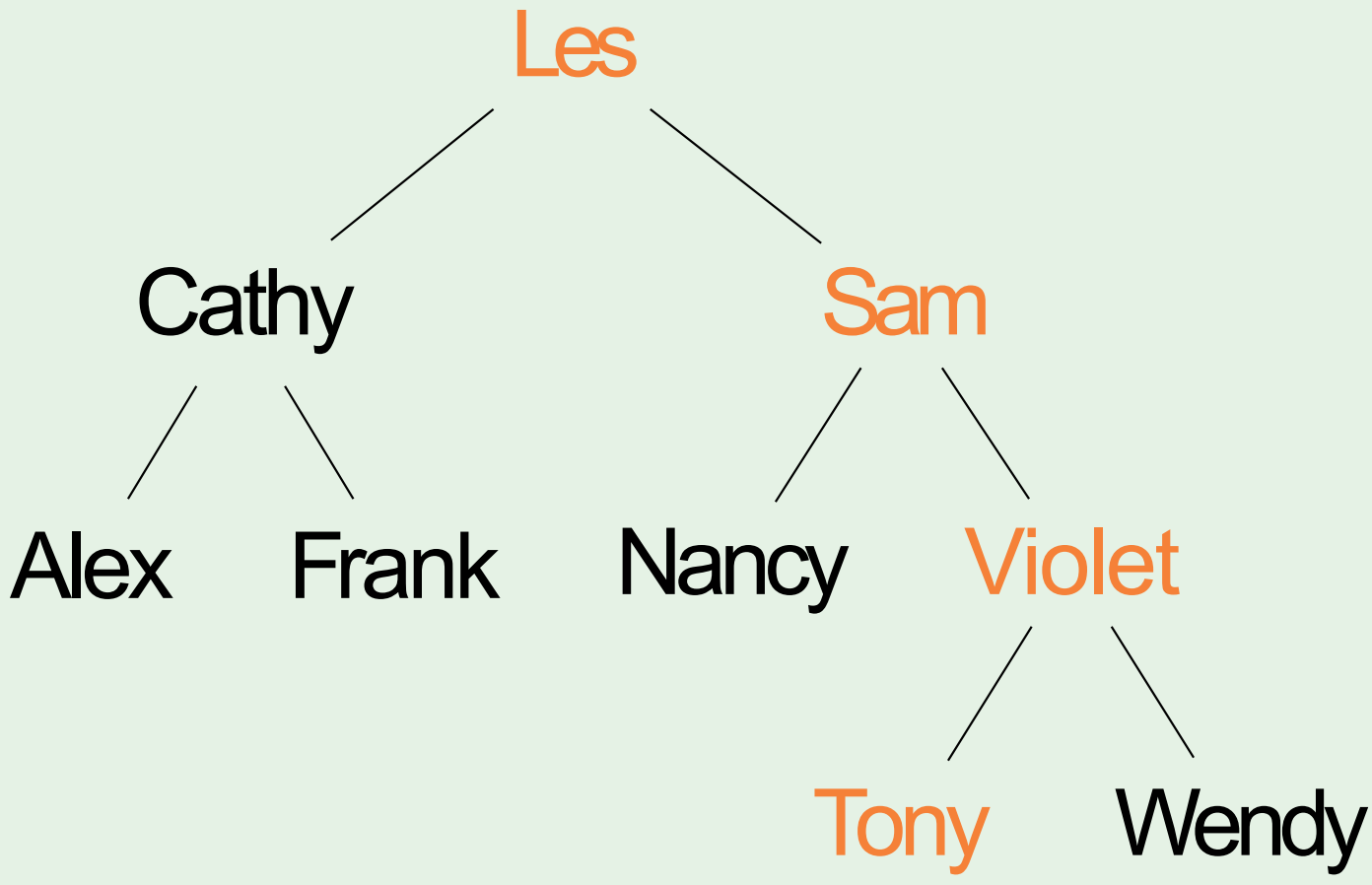
Output: Les Cathy Alex Frank Sam Nancy Violet Tony

PreOrderTraversal



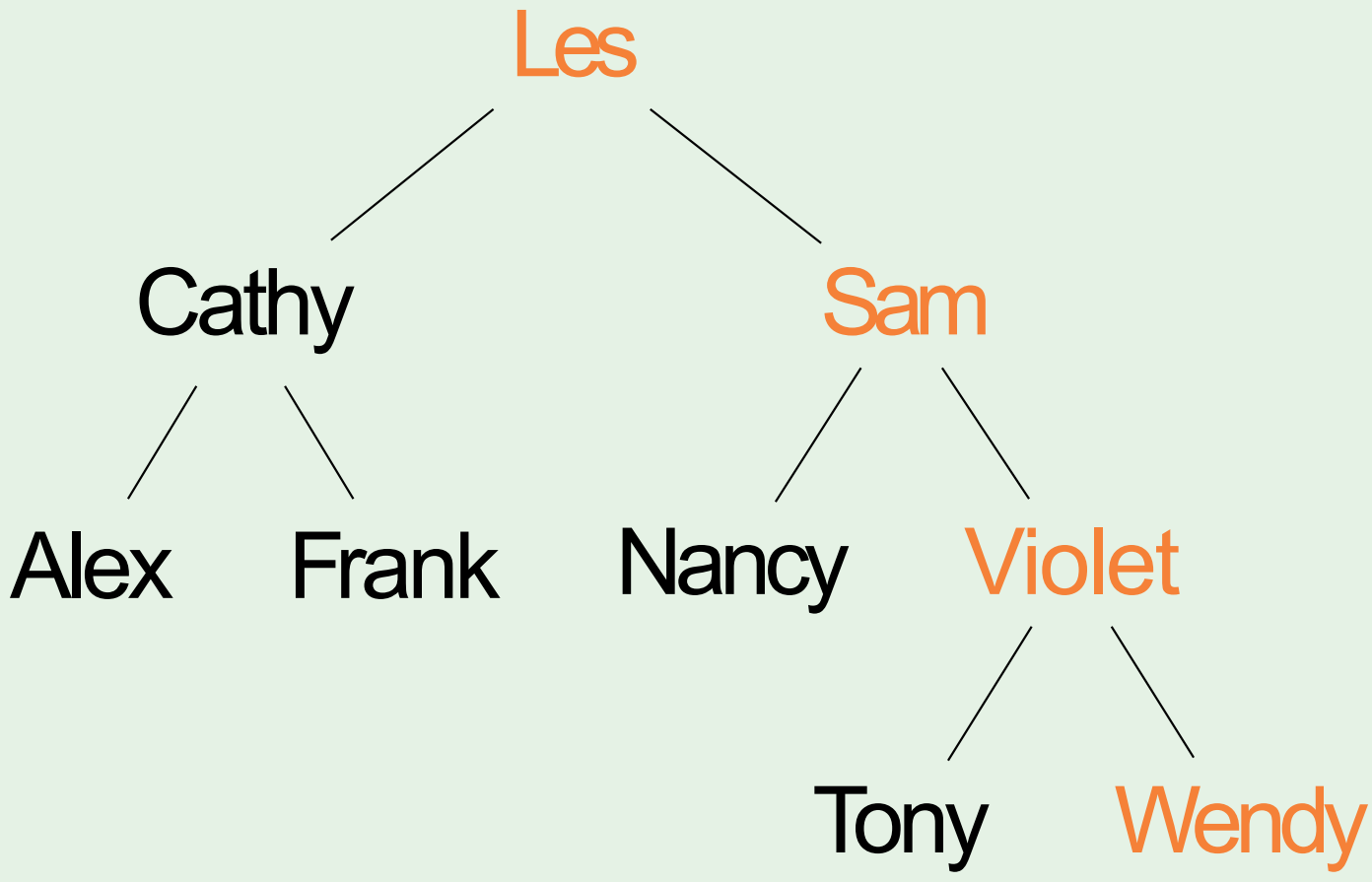
Output: Les Cathy Alex Frank Sam Nancy Violet Tony

PreOrderTraversal



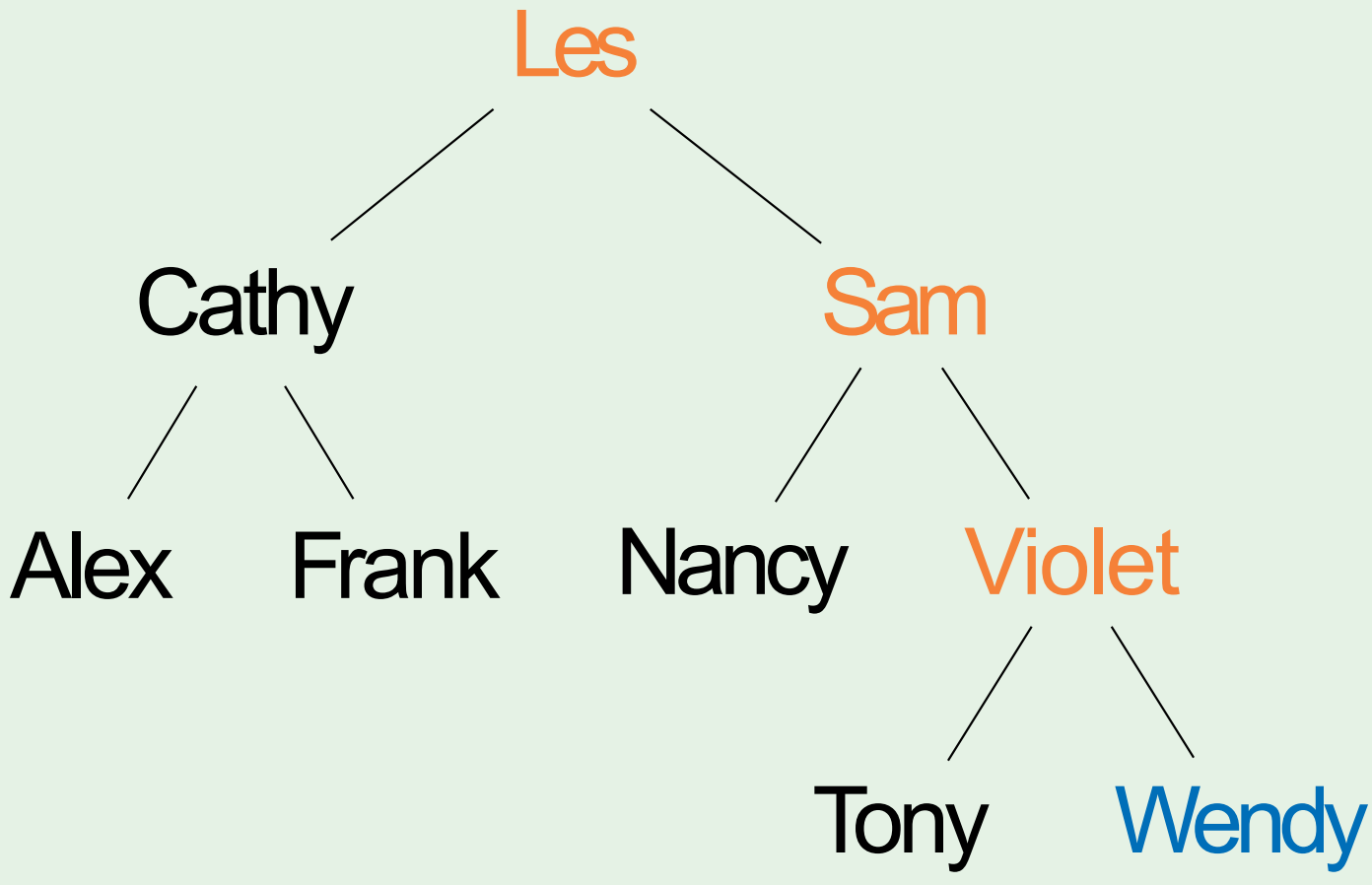
Output: Les Cathy Alex Frank Sam Nancy Violet Tony

PreOrderTraversal



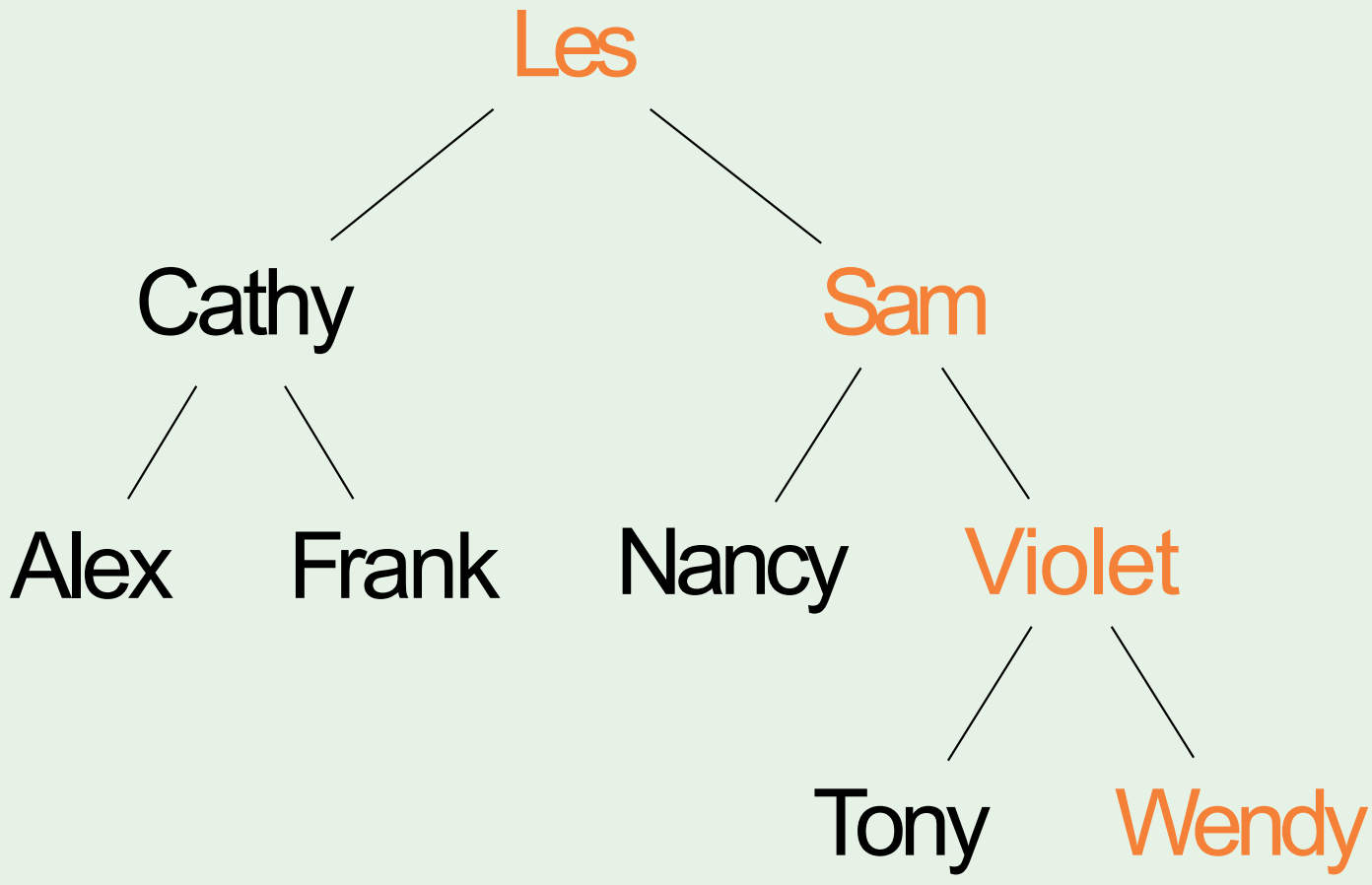
Output: Les Cathy Alex Frank Sam Nancy Violet Tony

PreOrderTraversal



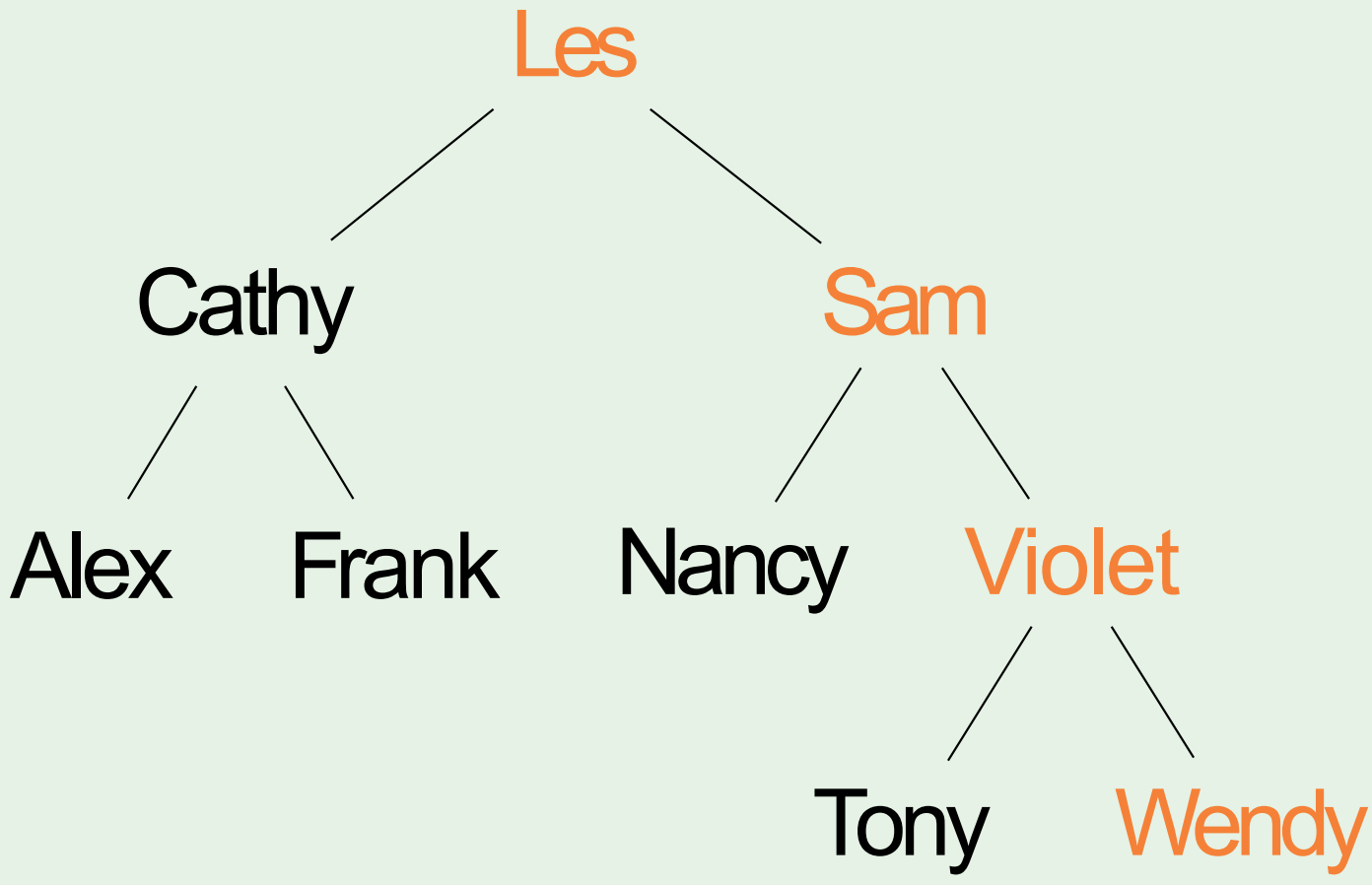
Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

PreOrderTraversal



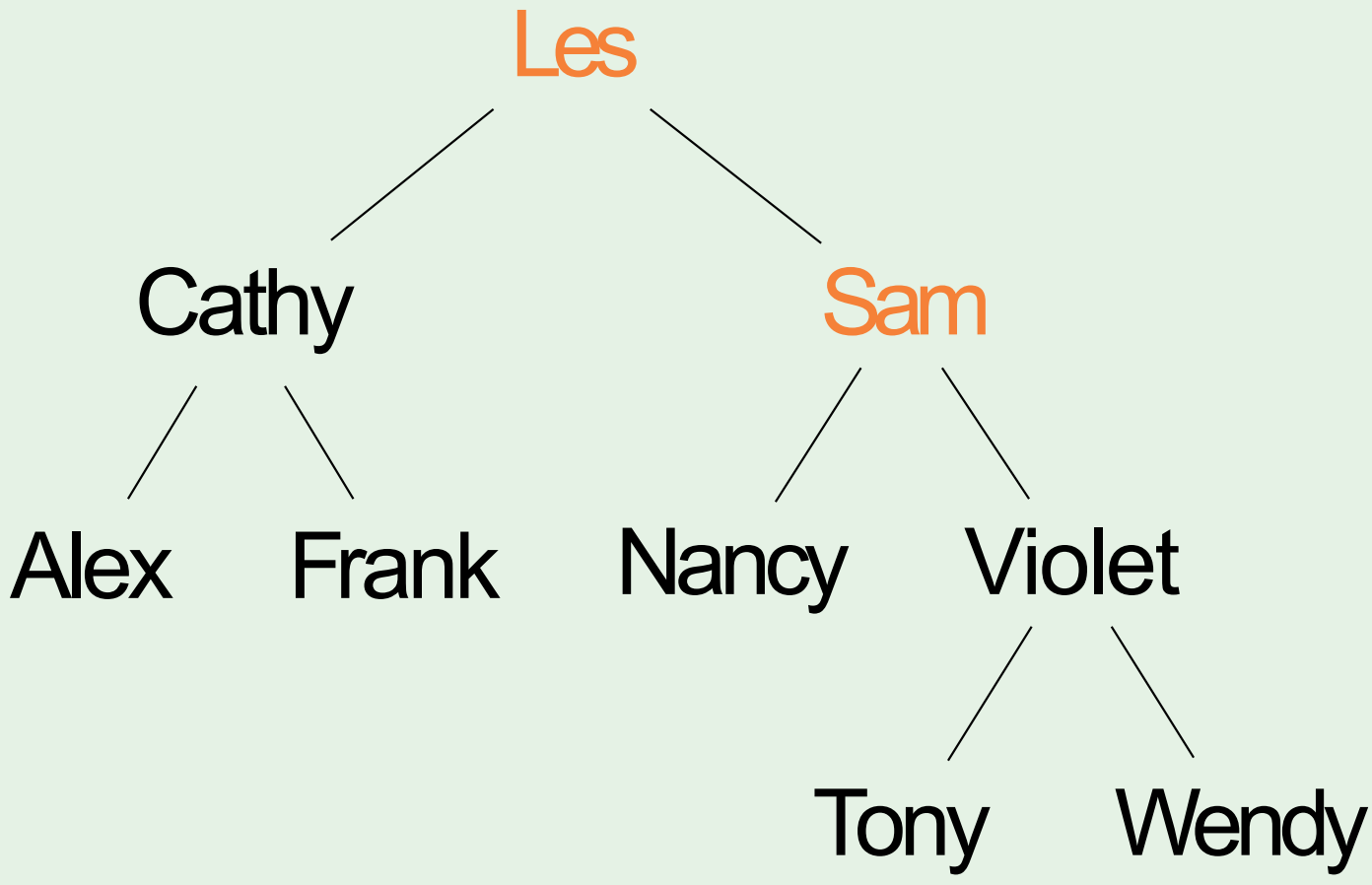
Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

PreOrderTraversal



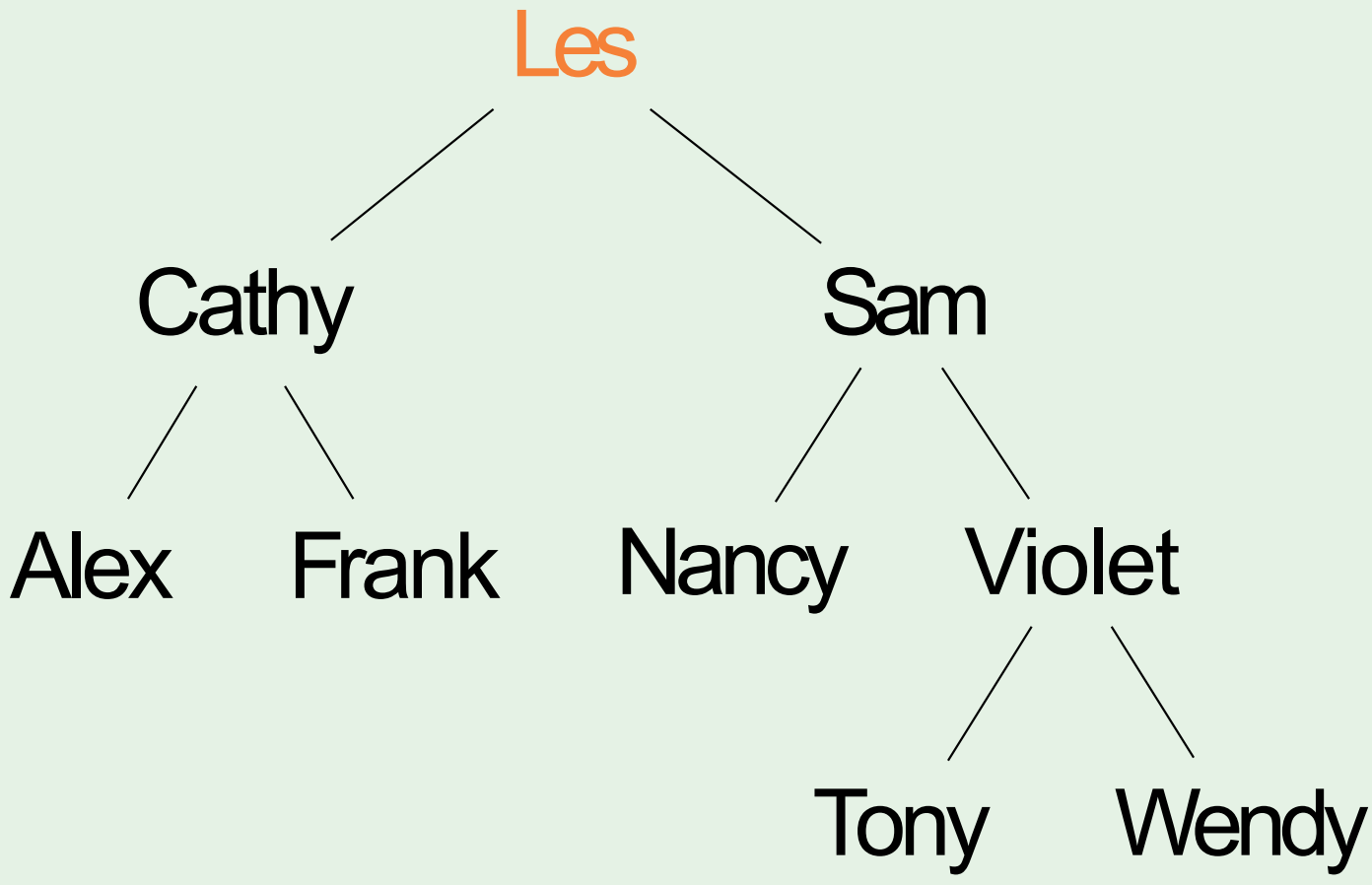
Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

PreOrderTraversal



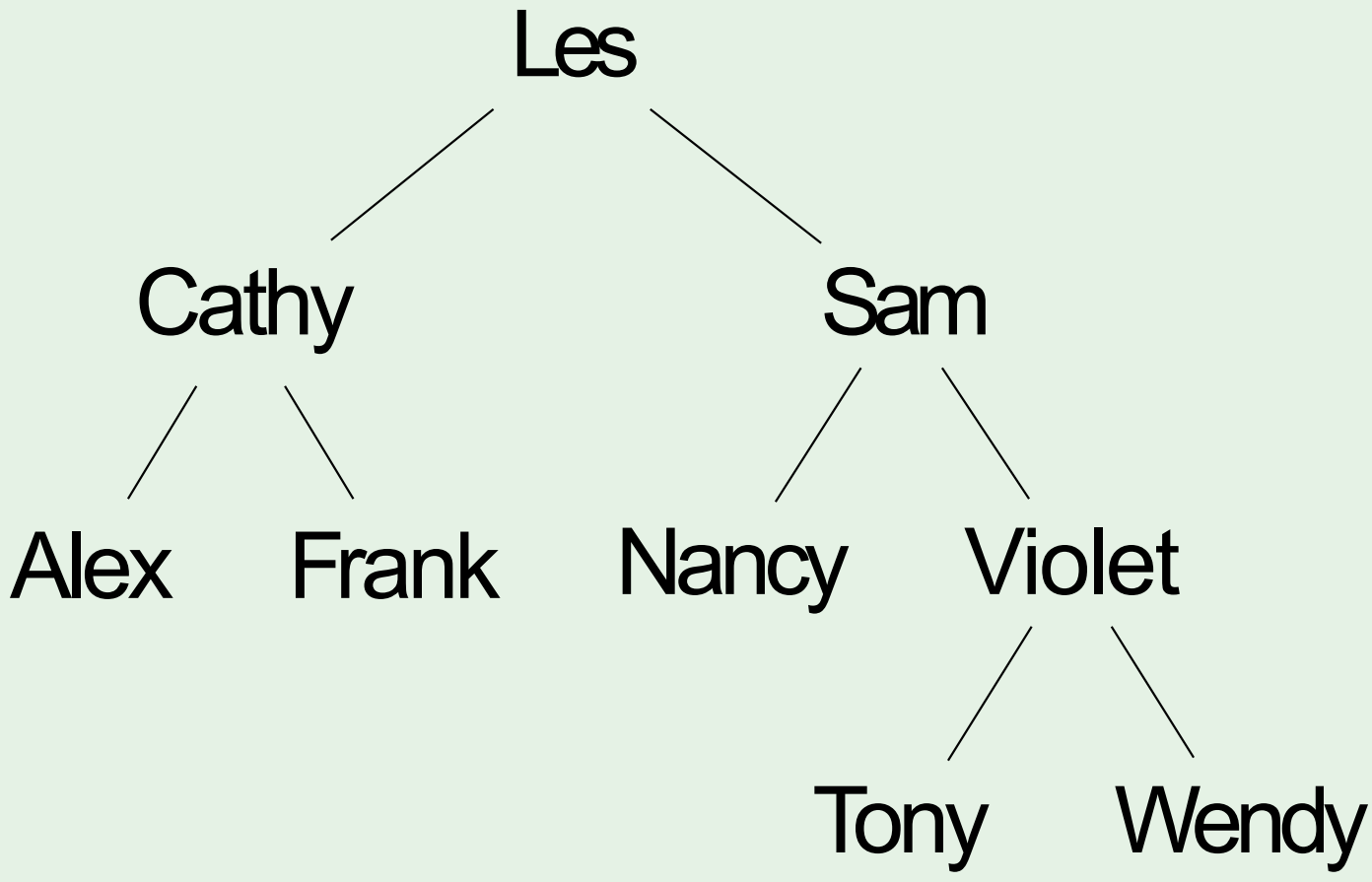
Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

PreOrderTraversal



Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

PreOrderTraversal



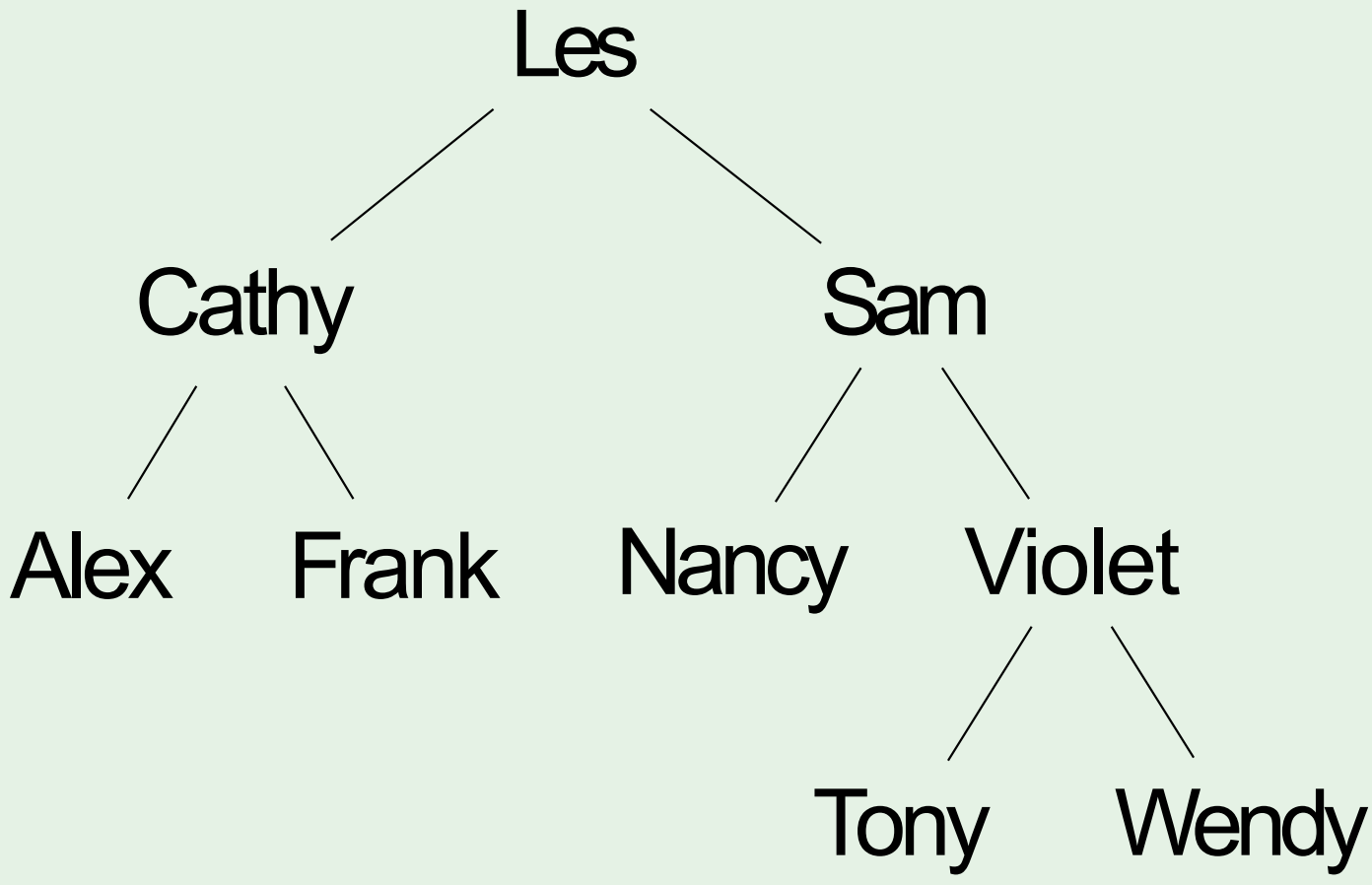
Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

Depth-first

PostOrderTraversal(*tree*)

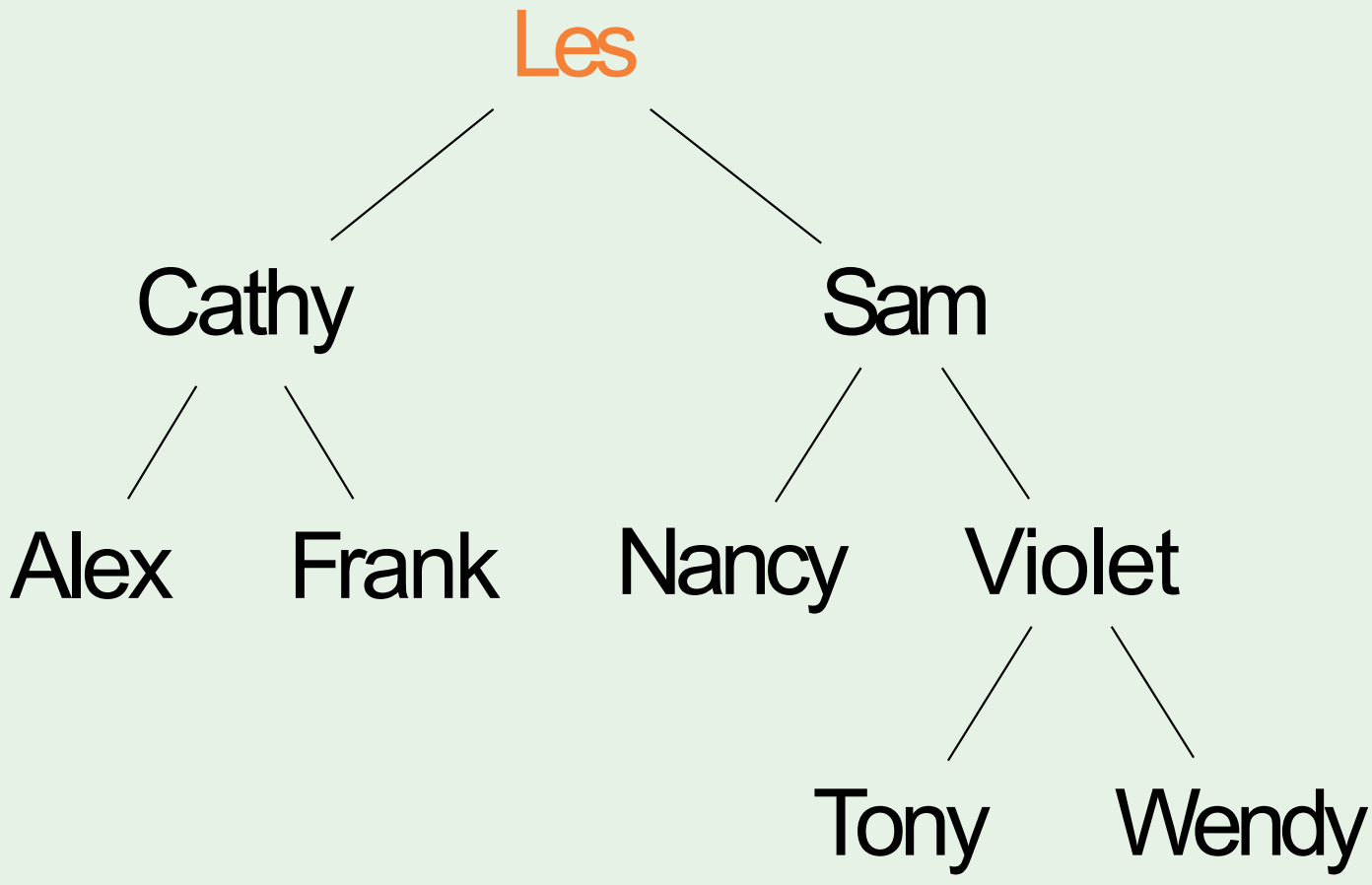
```
if tree = nil:
   return
PostOrderTraversal(tree.left)
PostOrderTraversal(tree.right)
Print(tree.key)
```

PostOrderTraversal



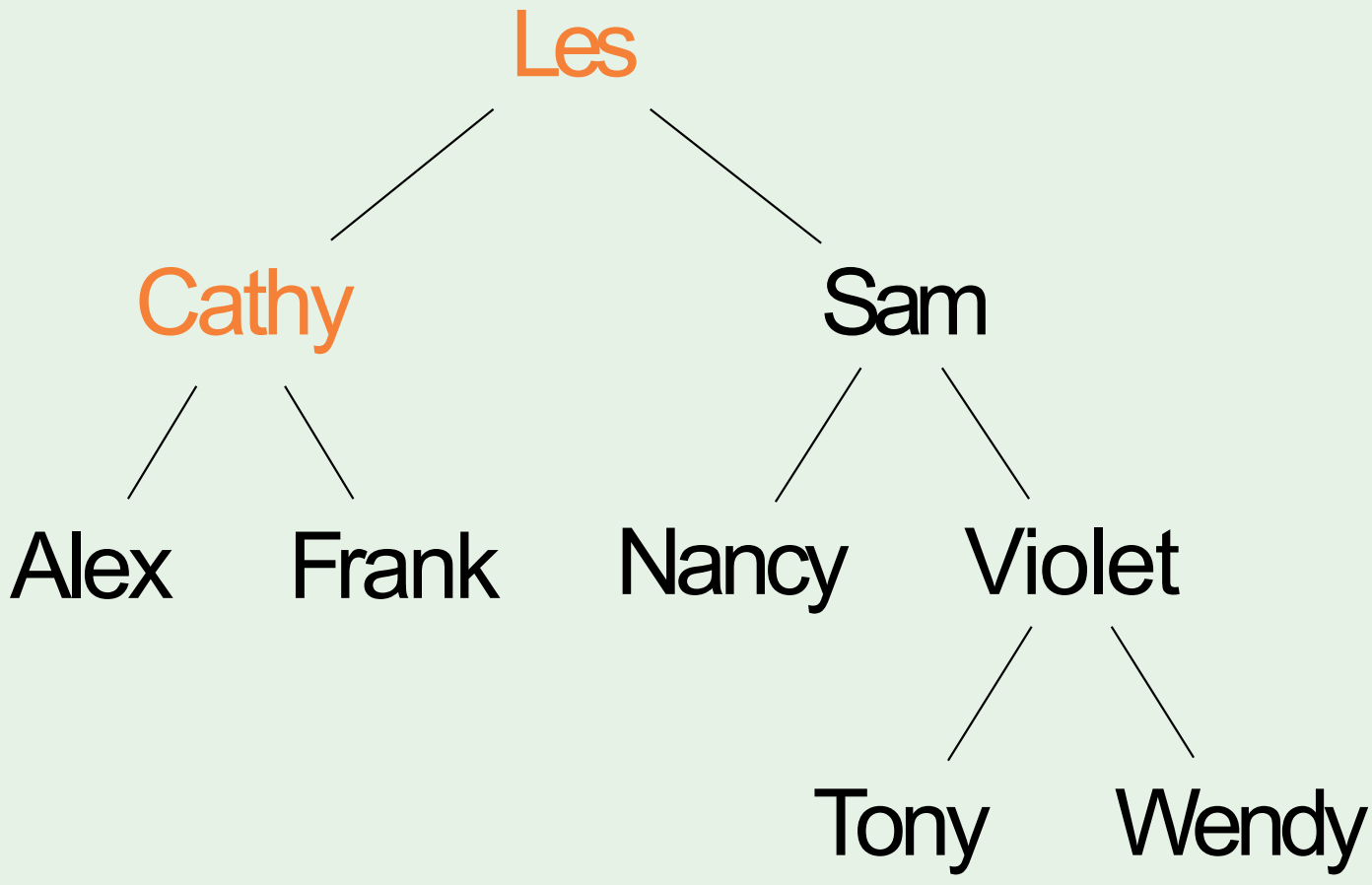
Output:

PostOrderTraversal



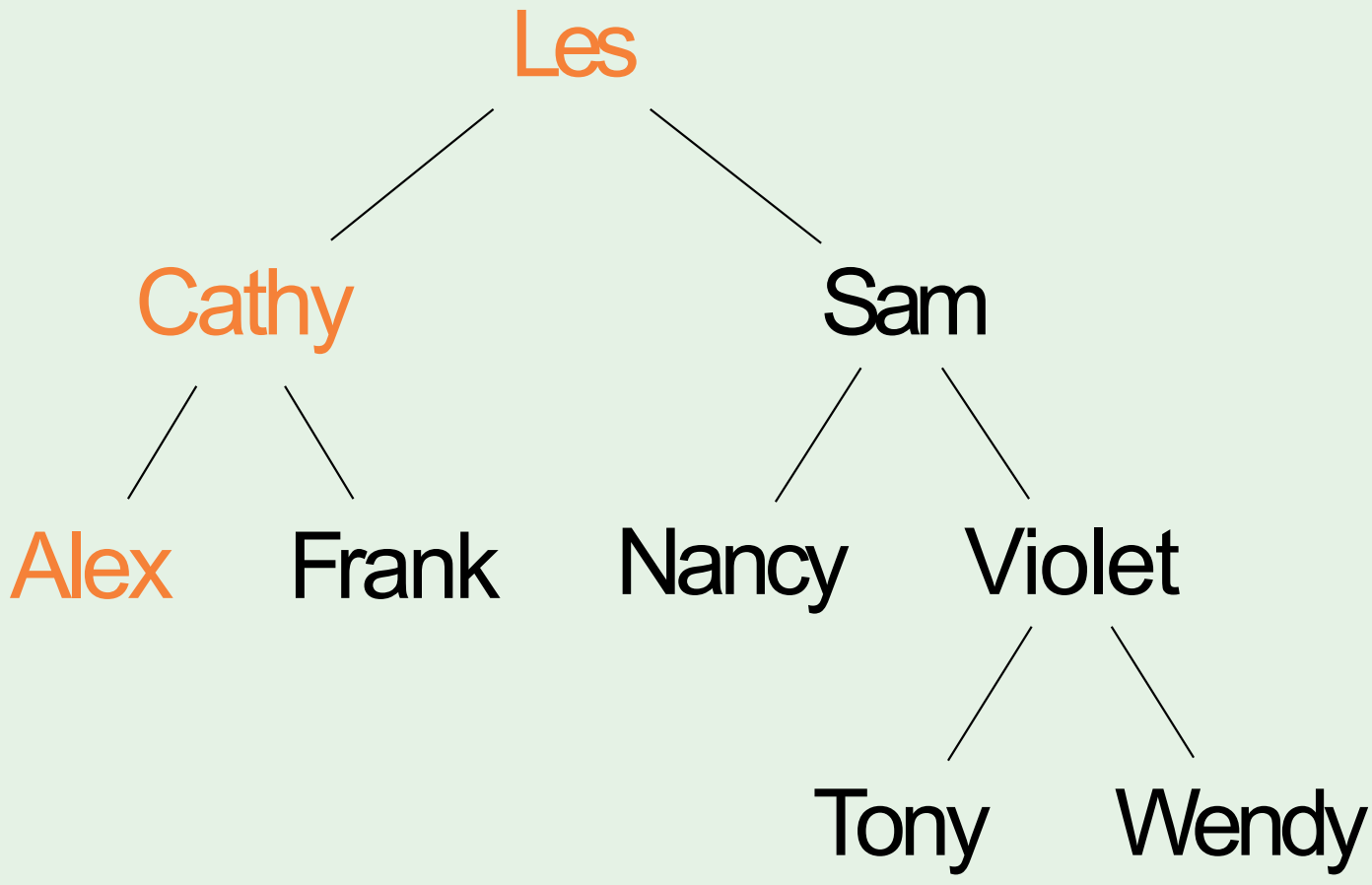
Output:

PostOrderTraversal



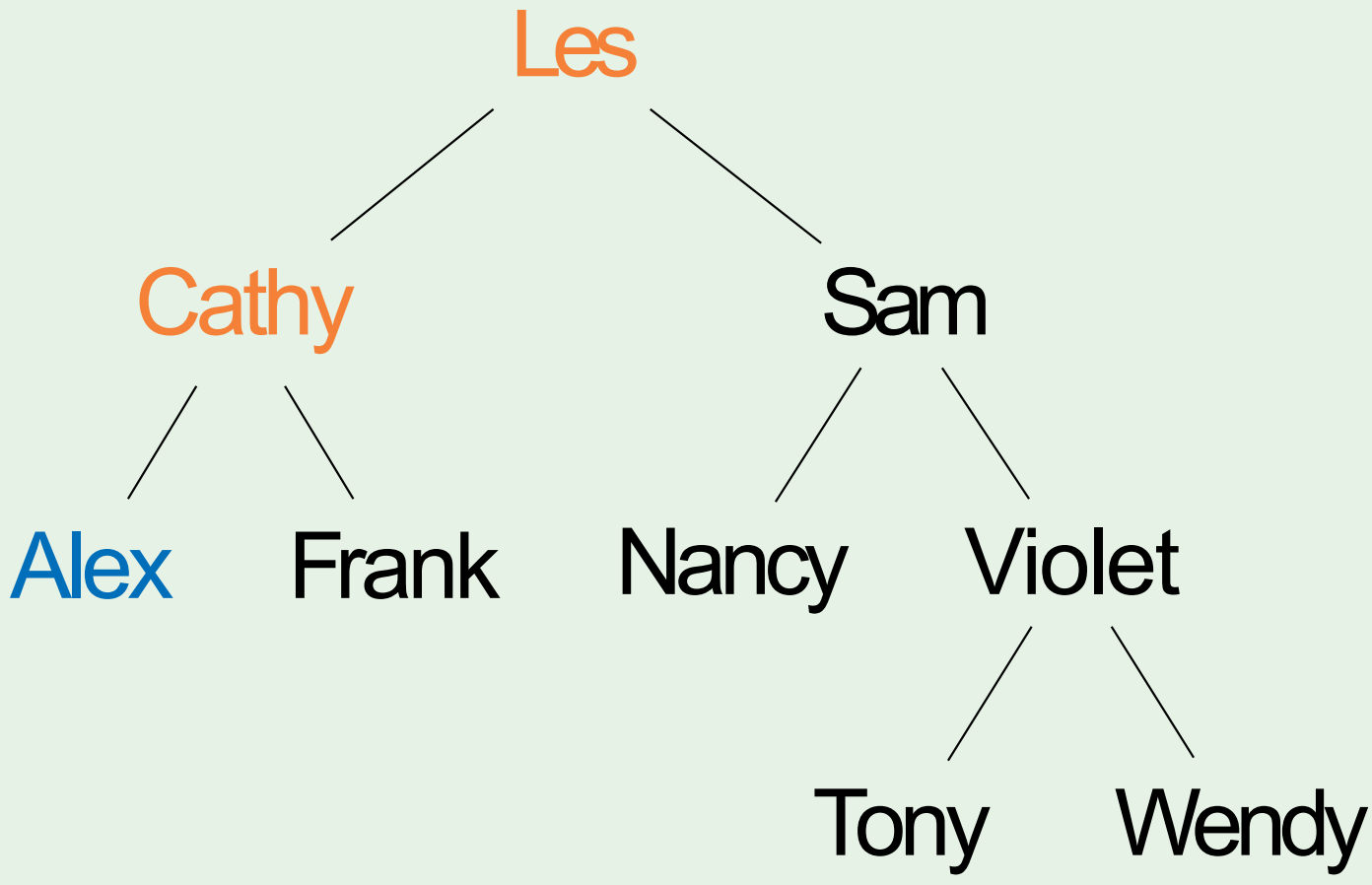
Output:

PostOrderTraversal



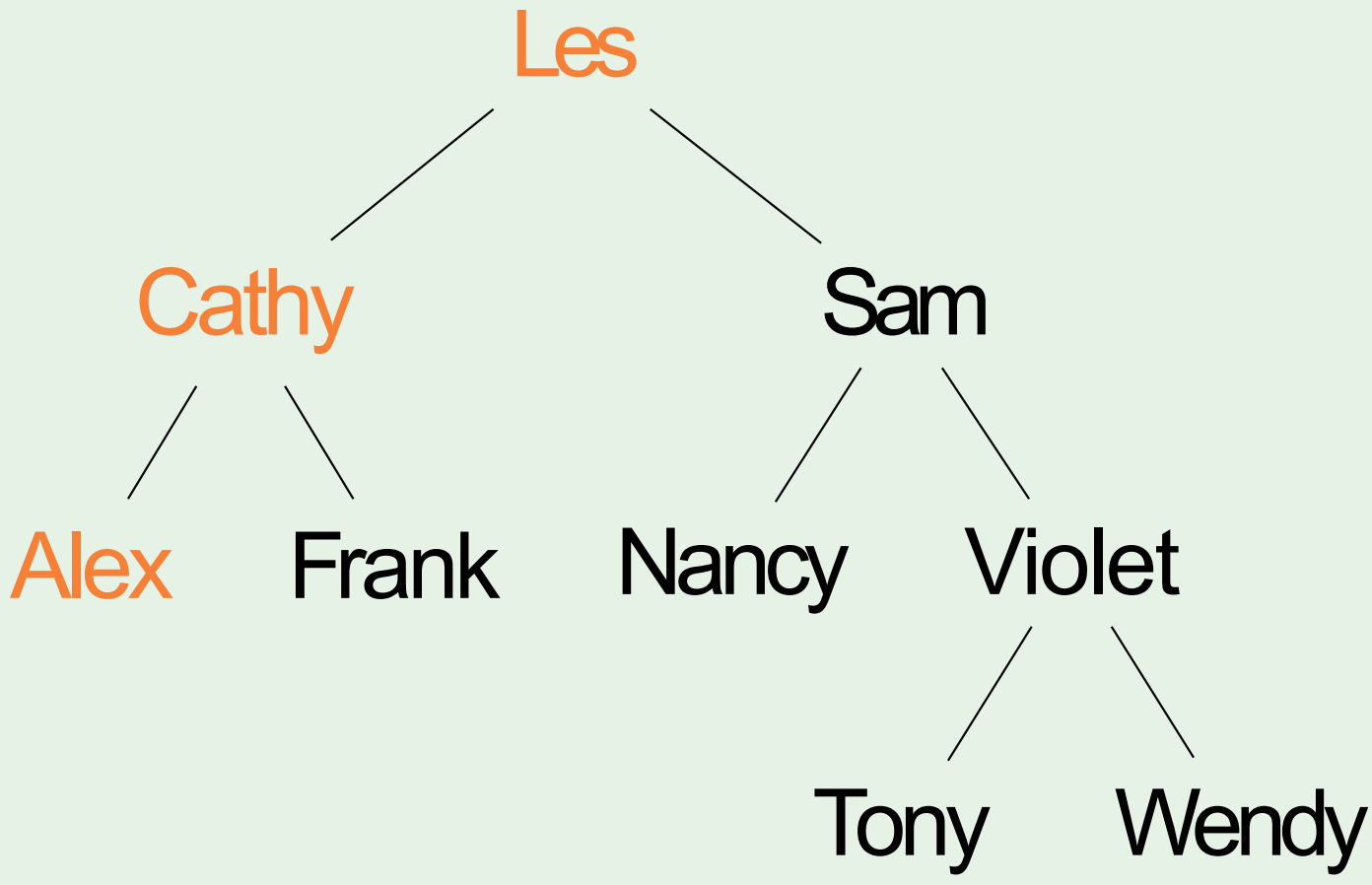
Output:

PostOrderTraversal



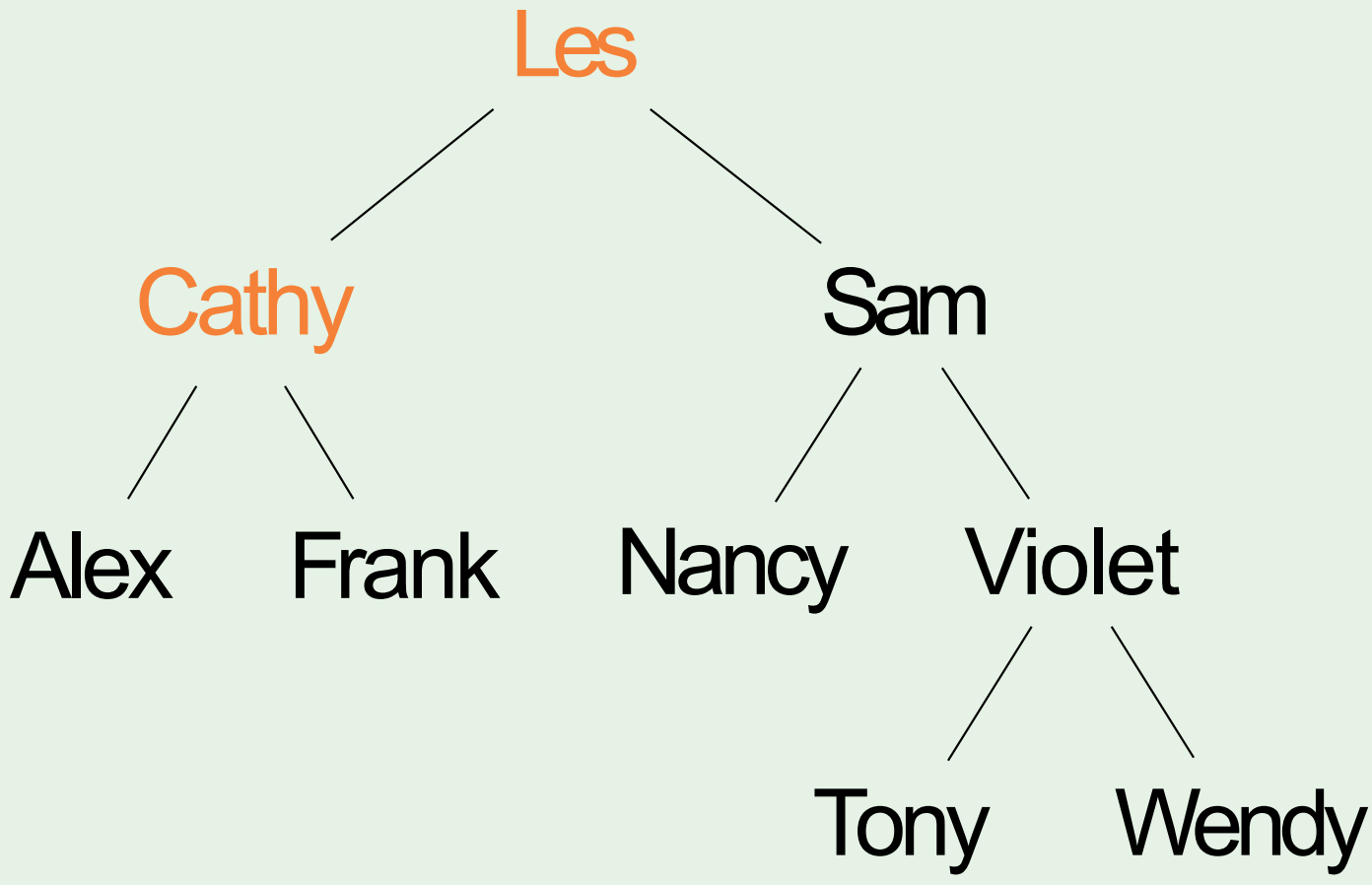
Output: Alex

PostOrderTraversal



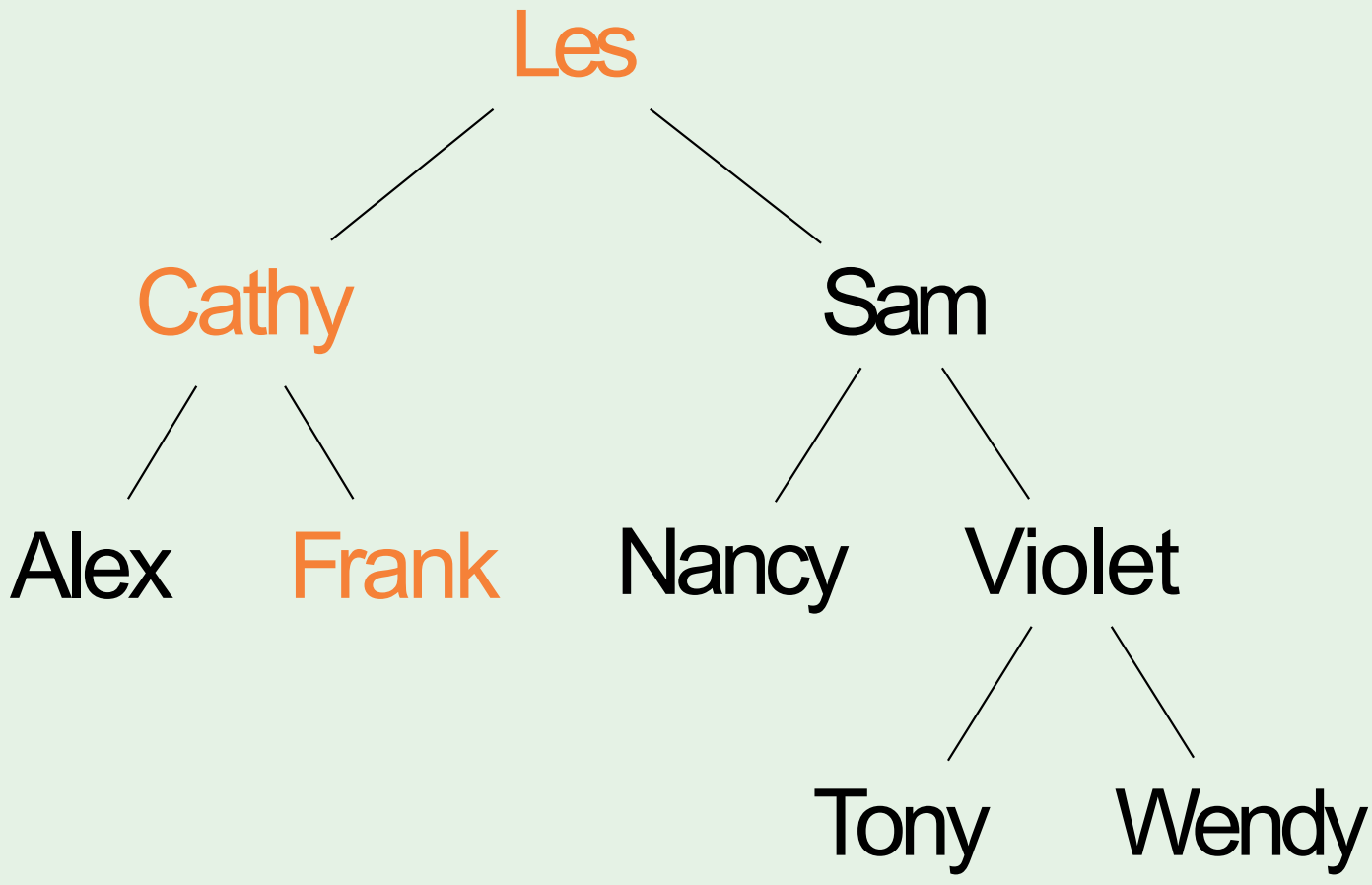
Output: Alex

PostOrderTraversal



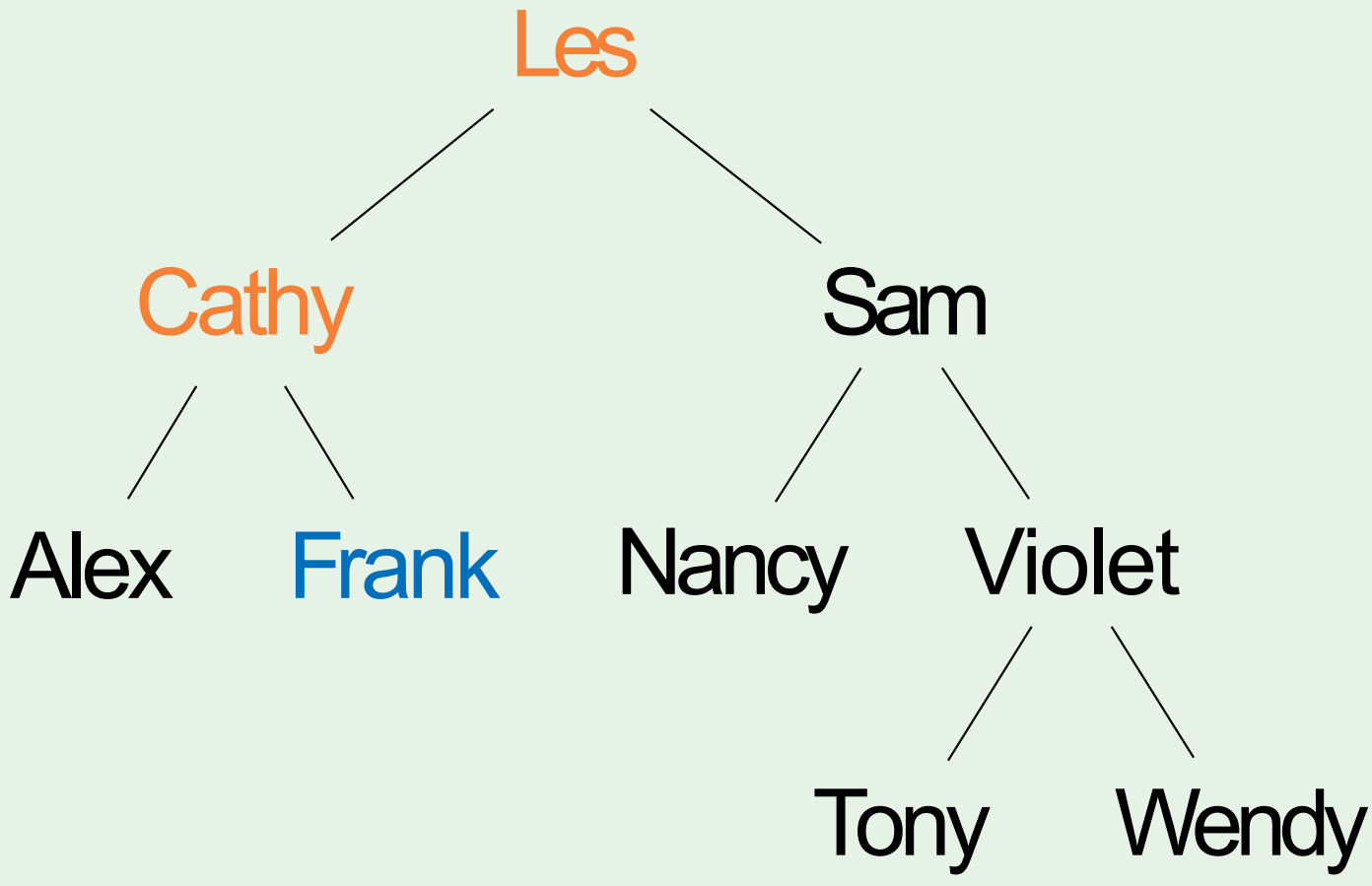
Output: Alex

PostOrderTraversal



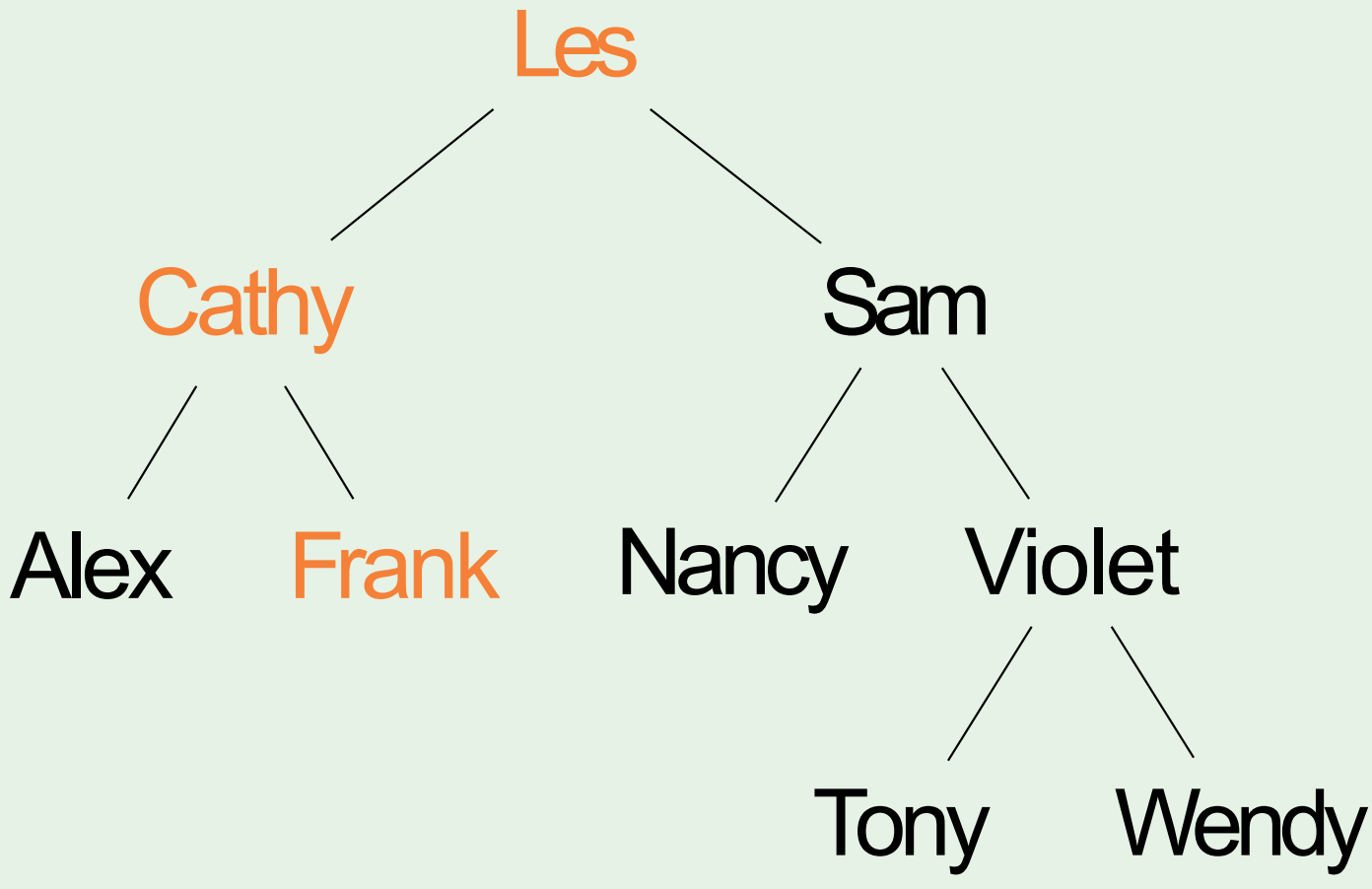
Output: Alex

PostOrderTraversal



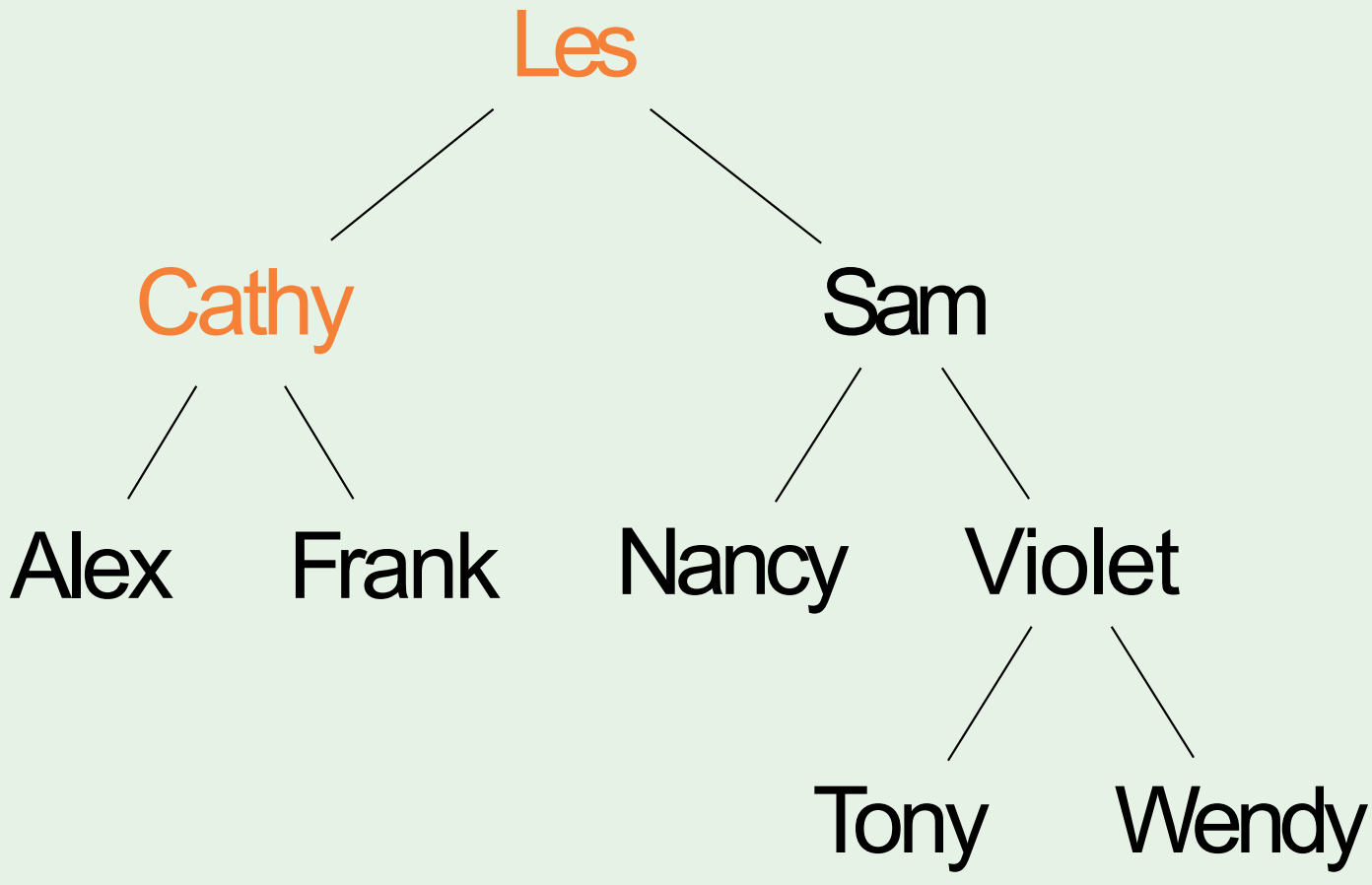
Output: Alex Frank

PostOrderTraversal



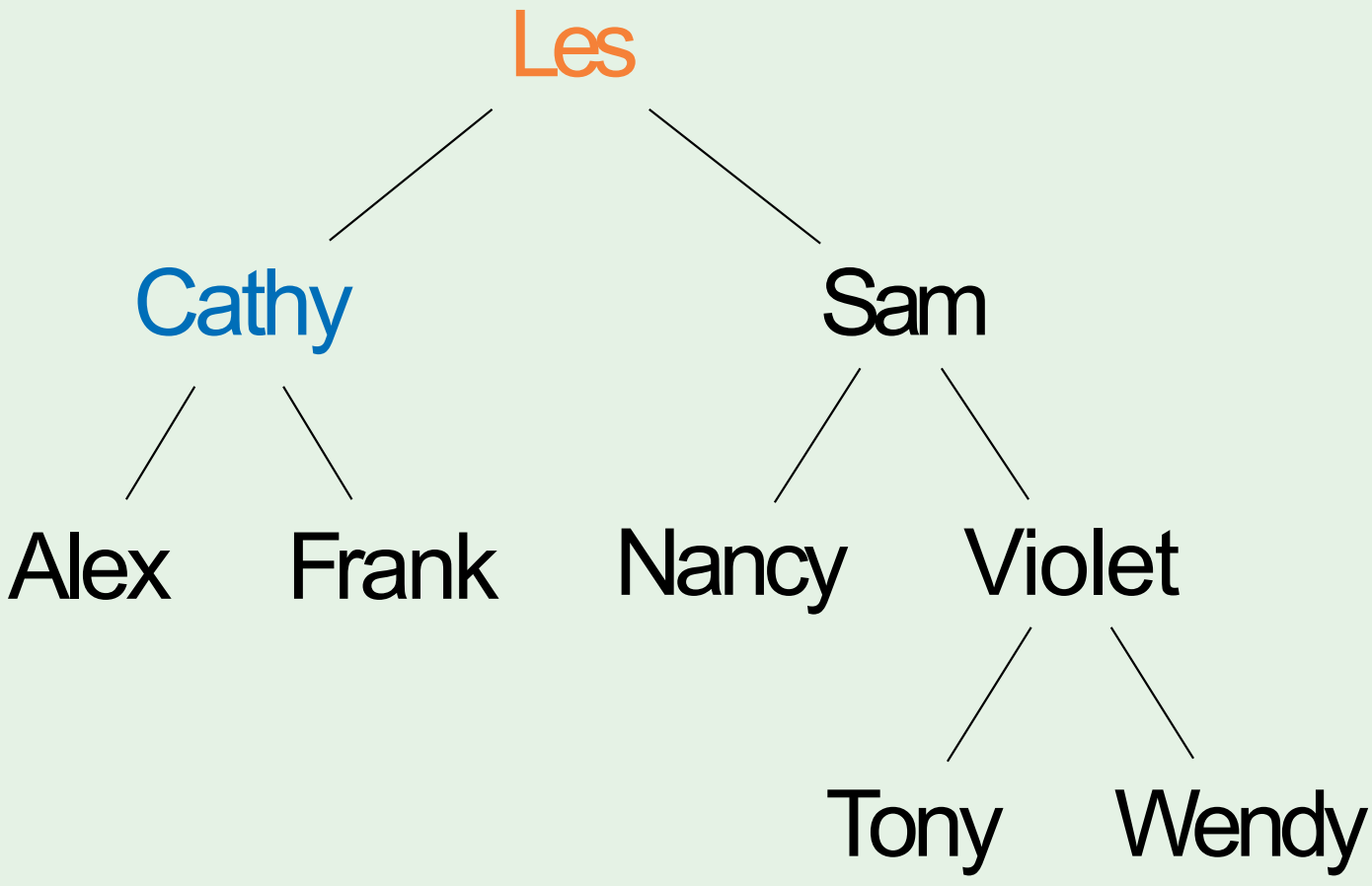
Output: Alex Frank

PostOrderTraversal



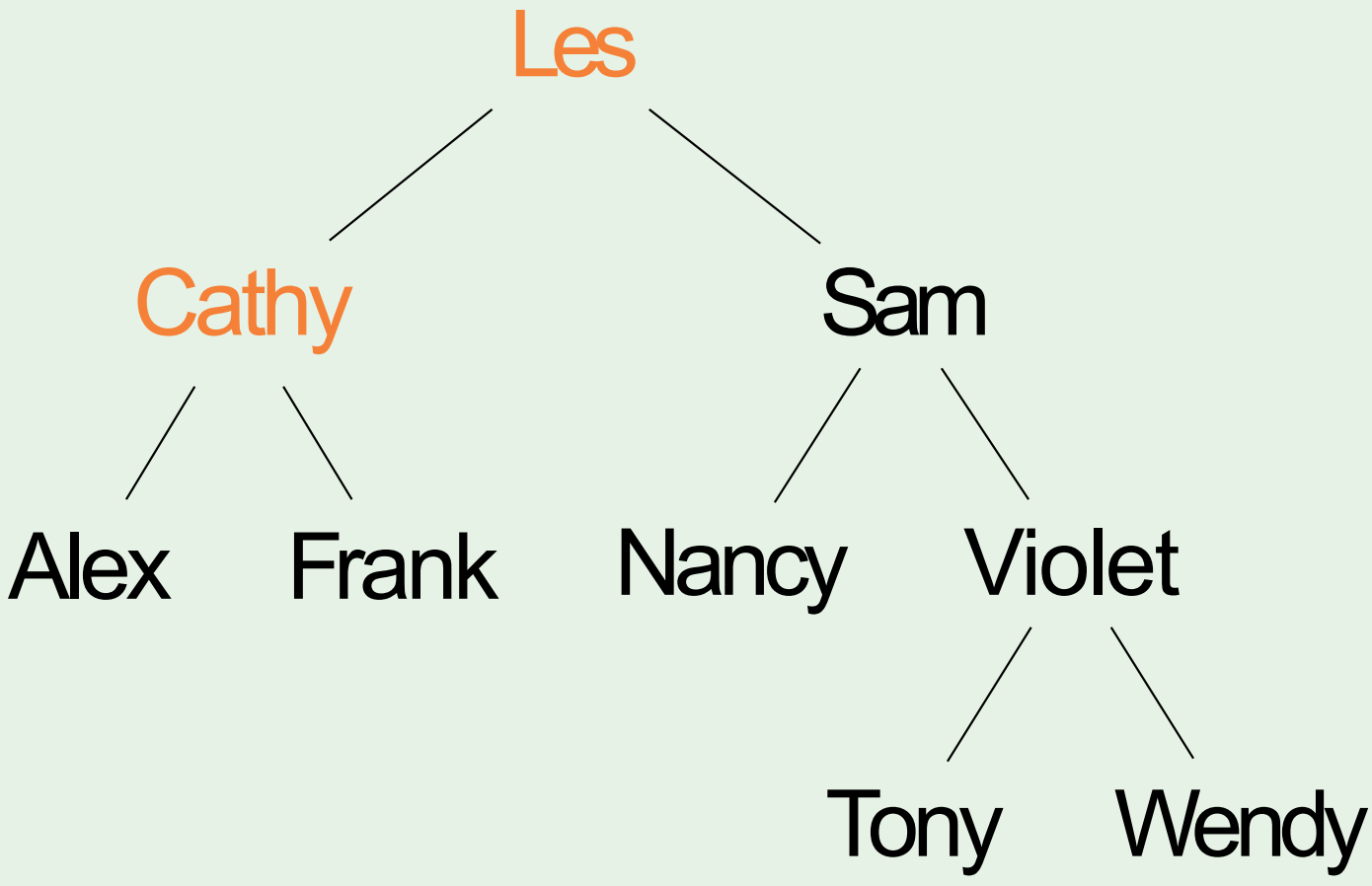
Output: AlexFrank

PostOrderTraversal



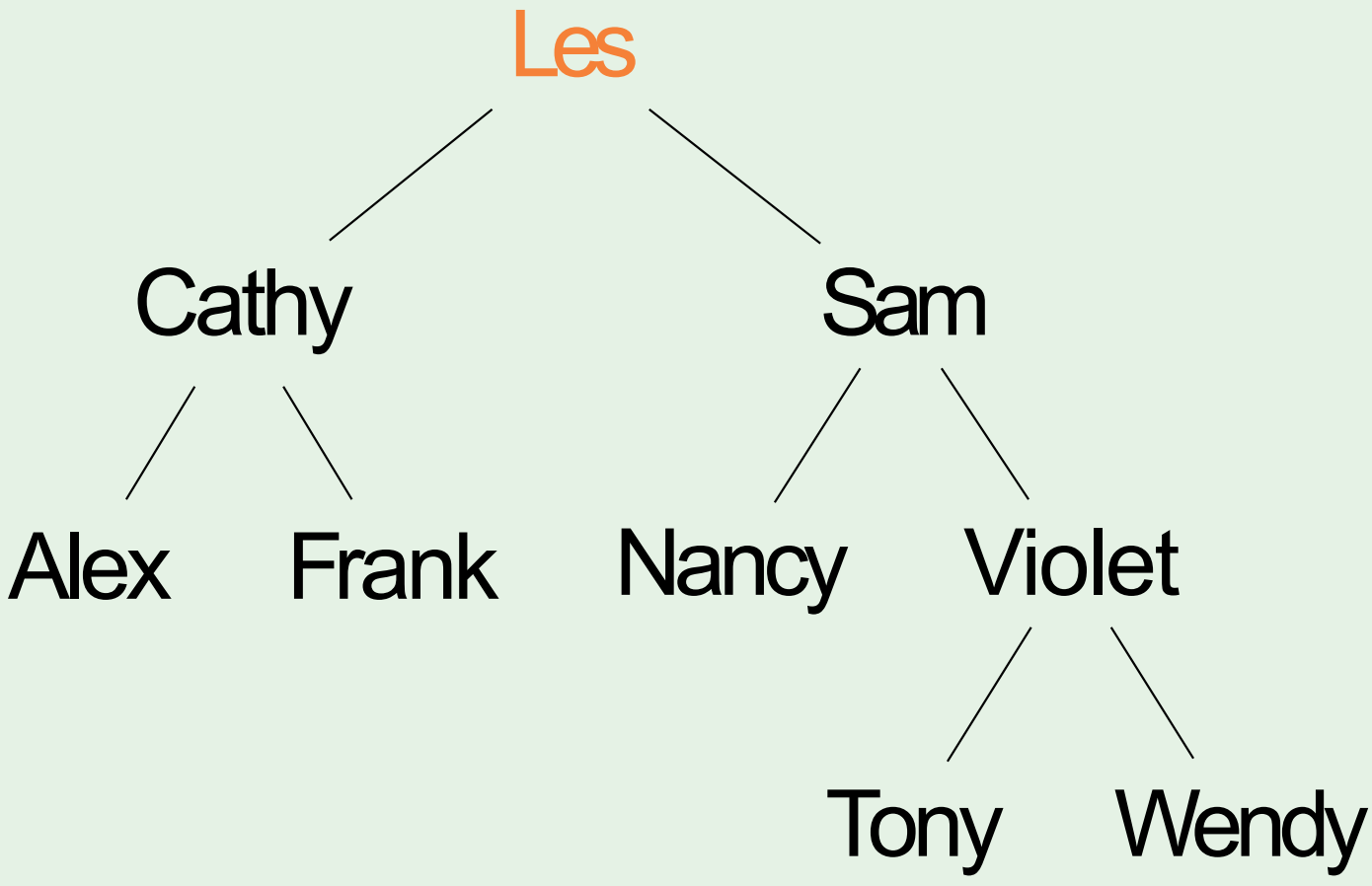
Output: Alex Frank Cathy

PostOrderTraversal



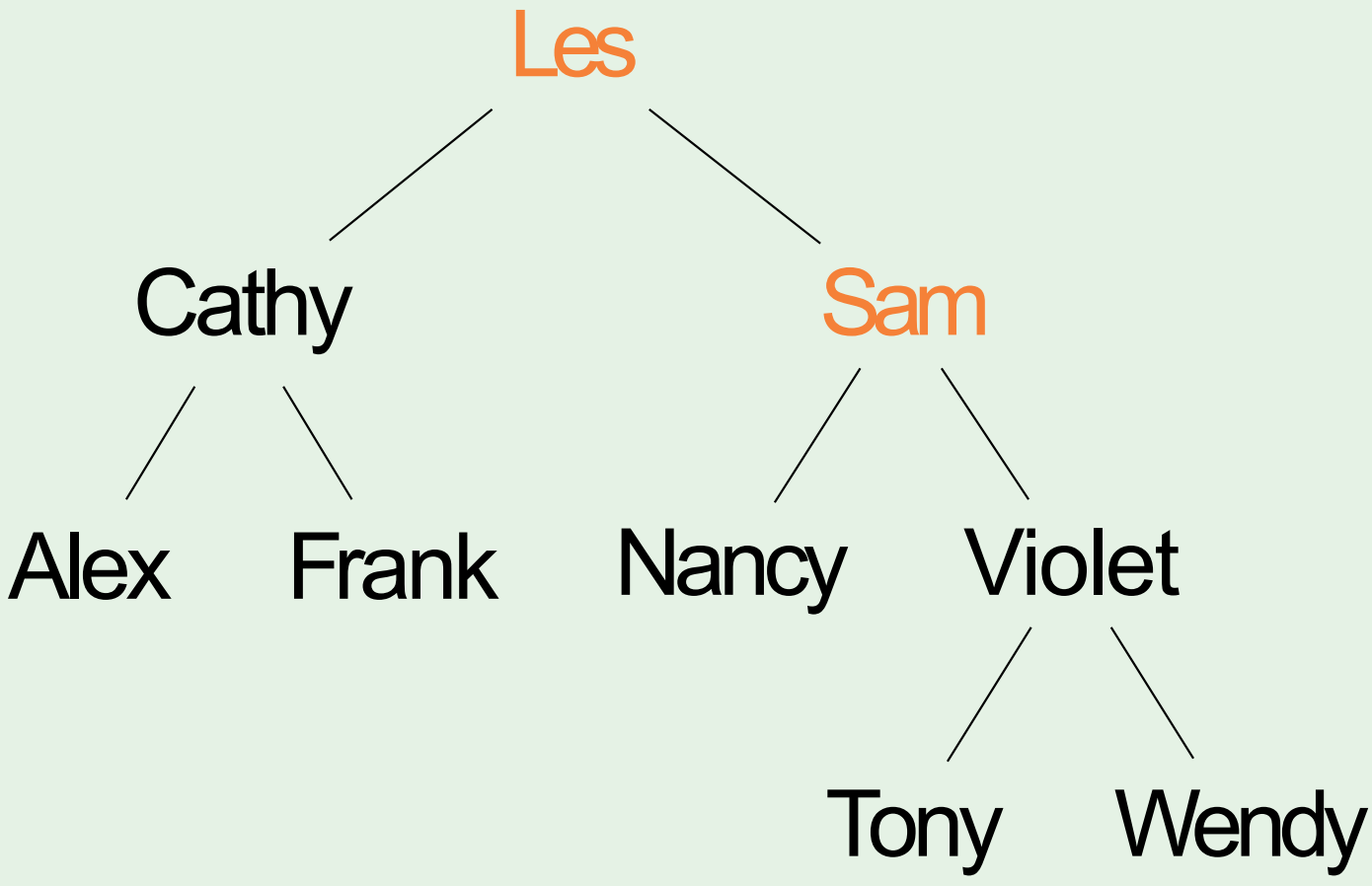
Output: Alex Frank Cathy

PostOrderTraversal



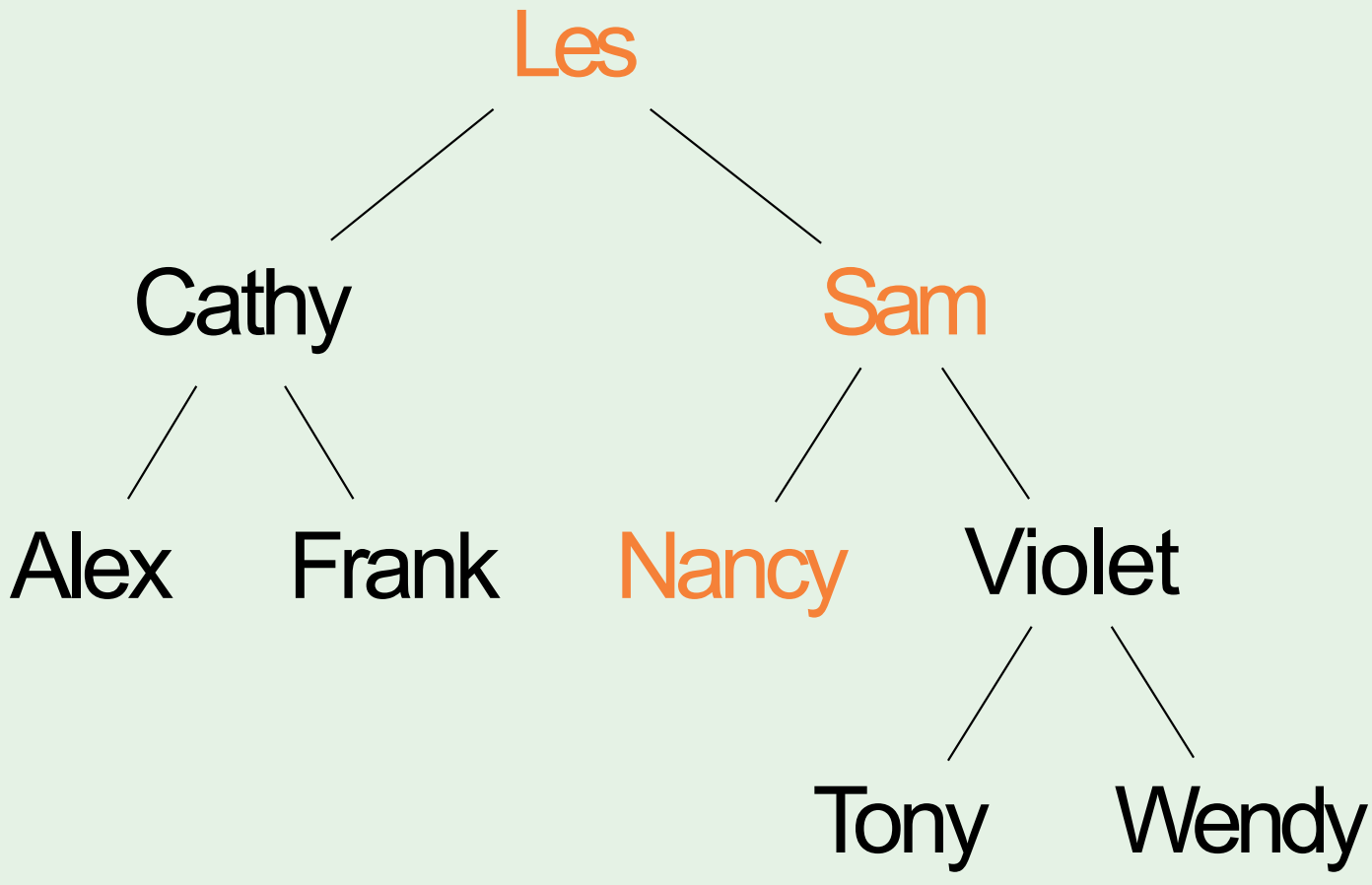
Output: Alex Frank Cathy

PostOrderTraversal



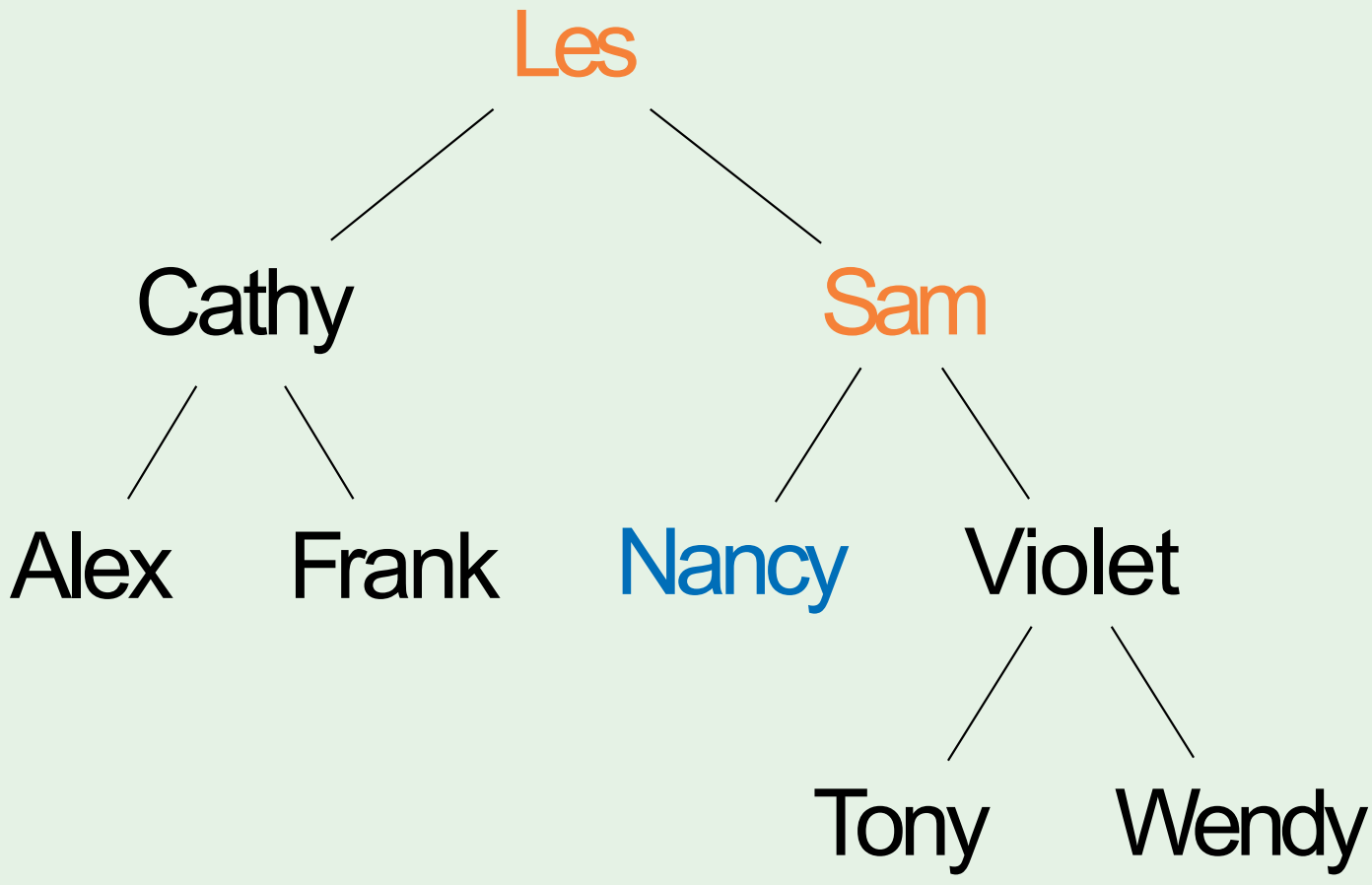
Output: Alex Frank Cathy

PostOrderTraversal



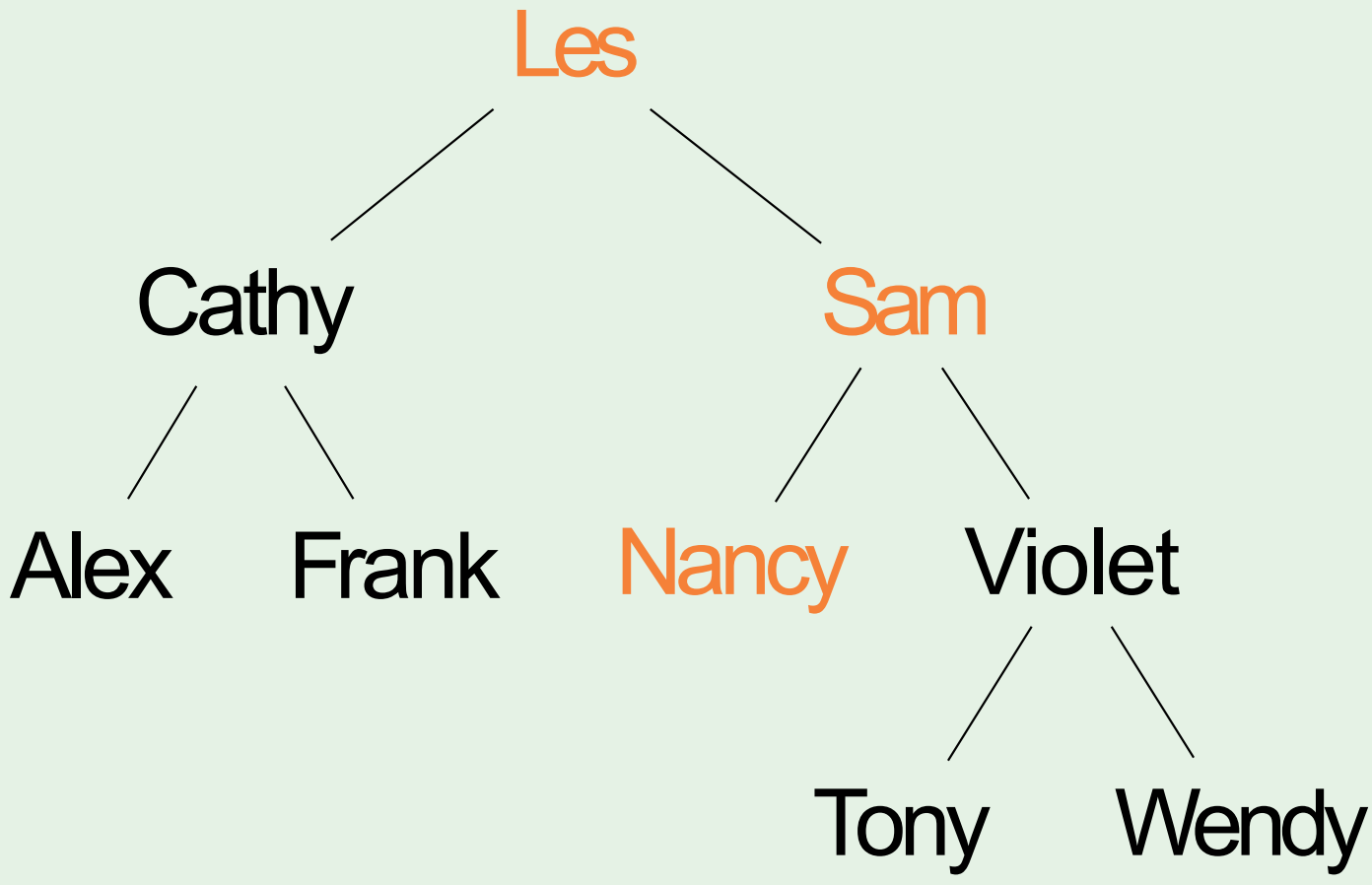
Output: Alex Frank Cathy

PostOrderTraversal



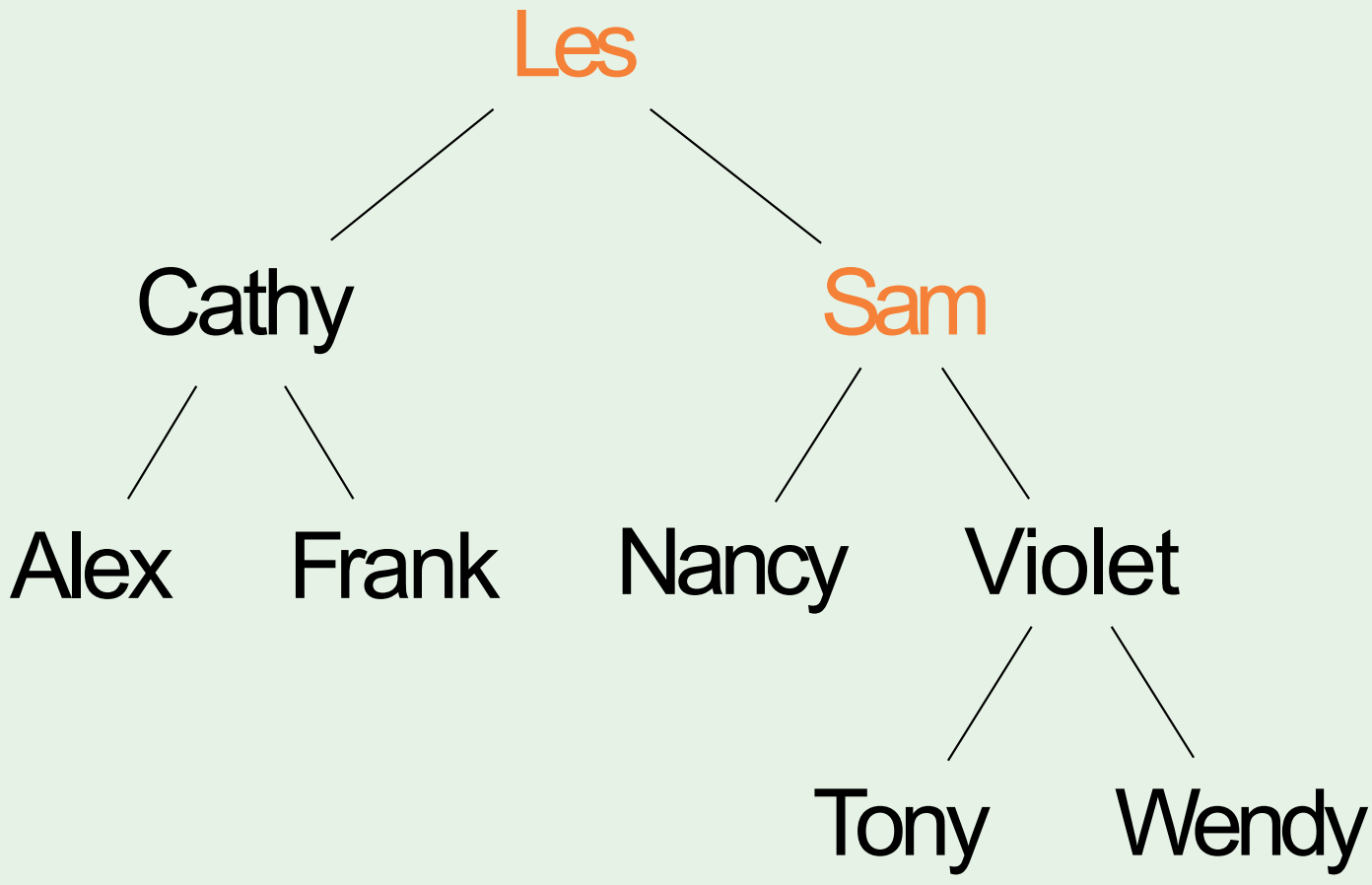
Output: Alex Frank Cathy Nancy

PostOrderTraversal



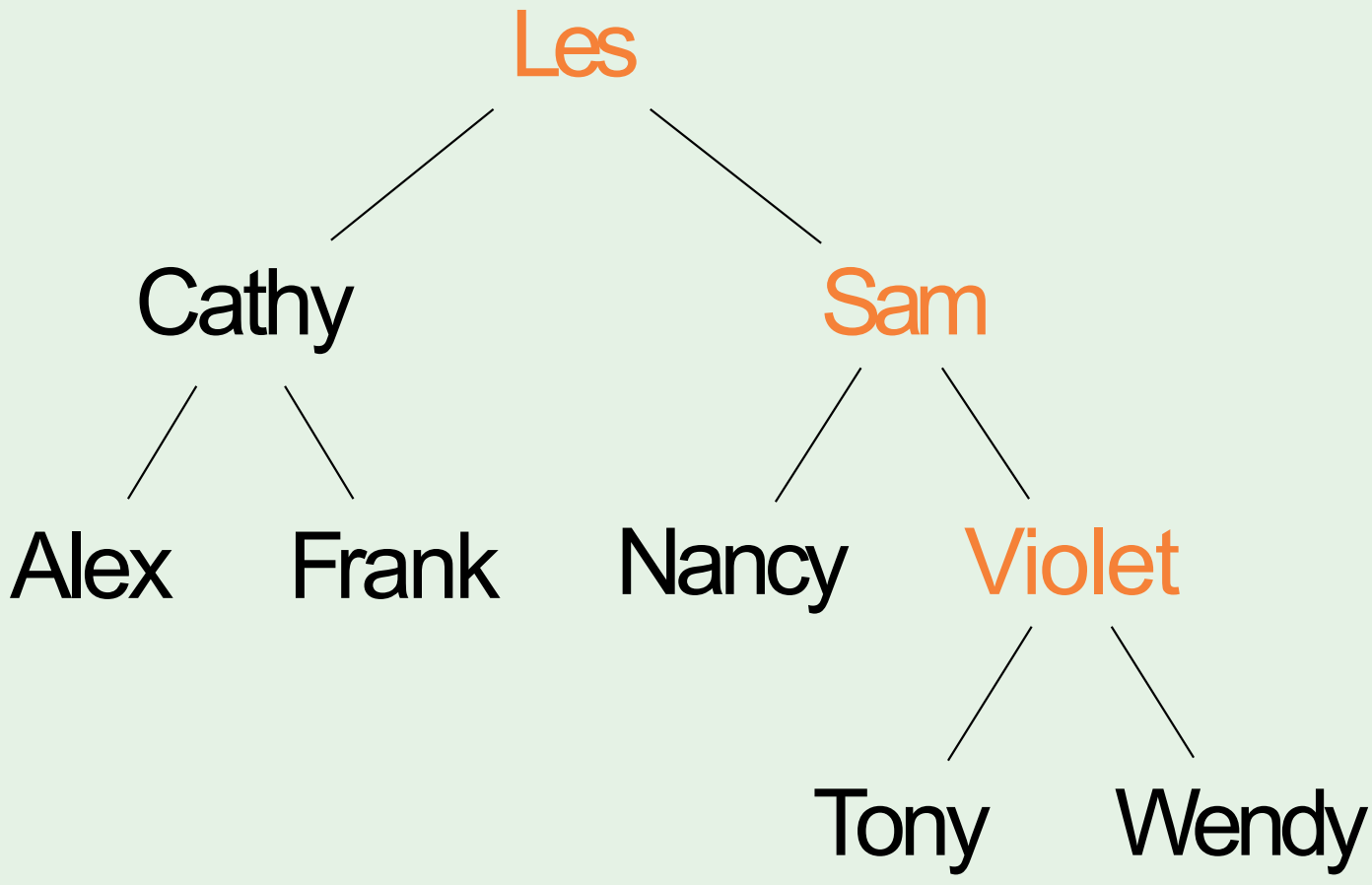
Output: Alex Frank Cathy Nancy

PostOrderTraversal



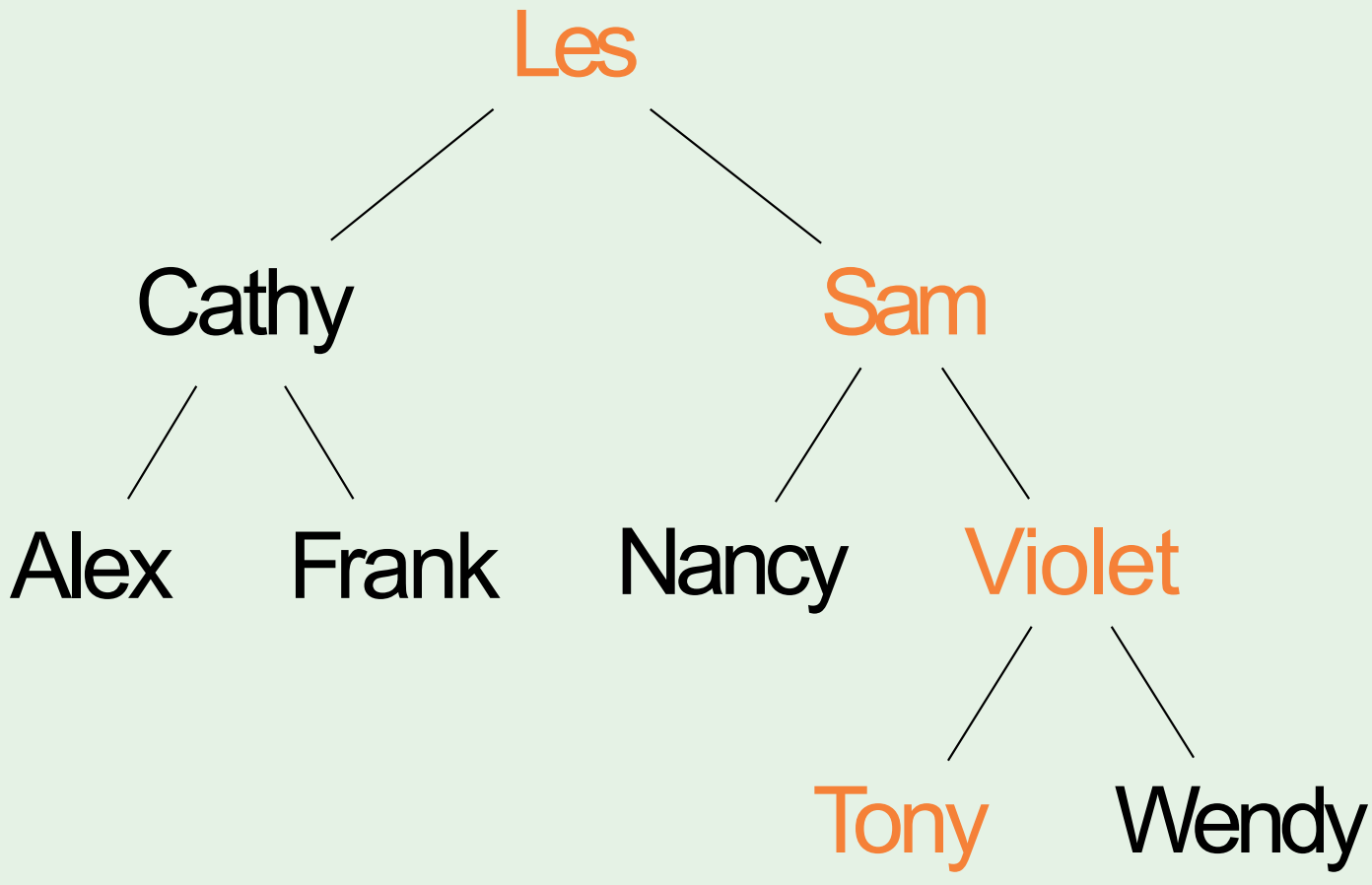
Output: Alex Frank Cathy Nancy

PostOrderTraversal



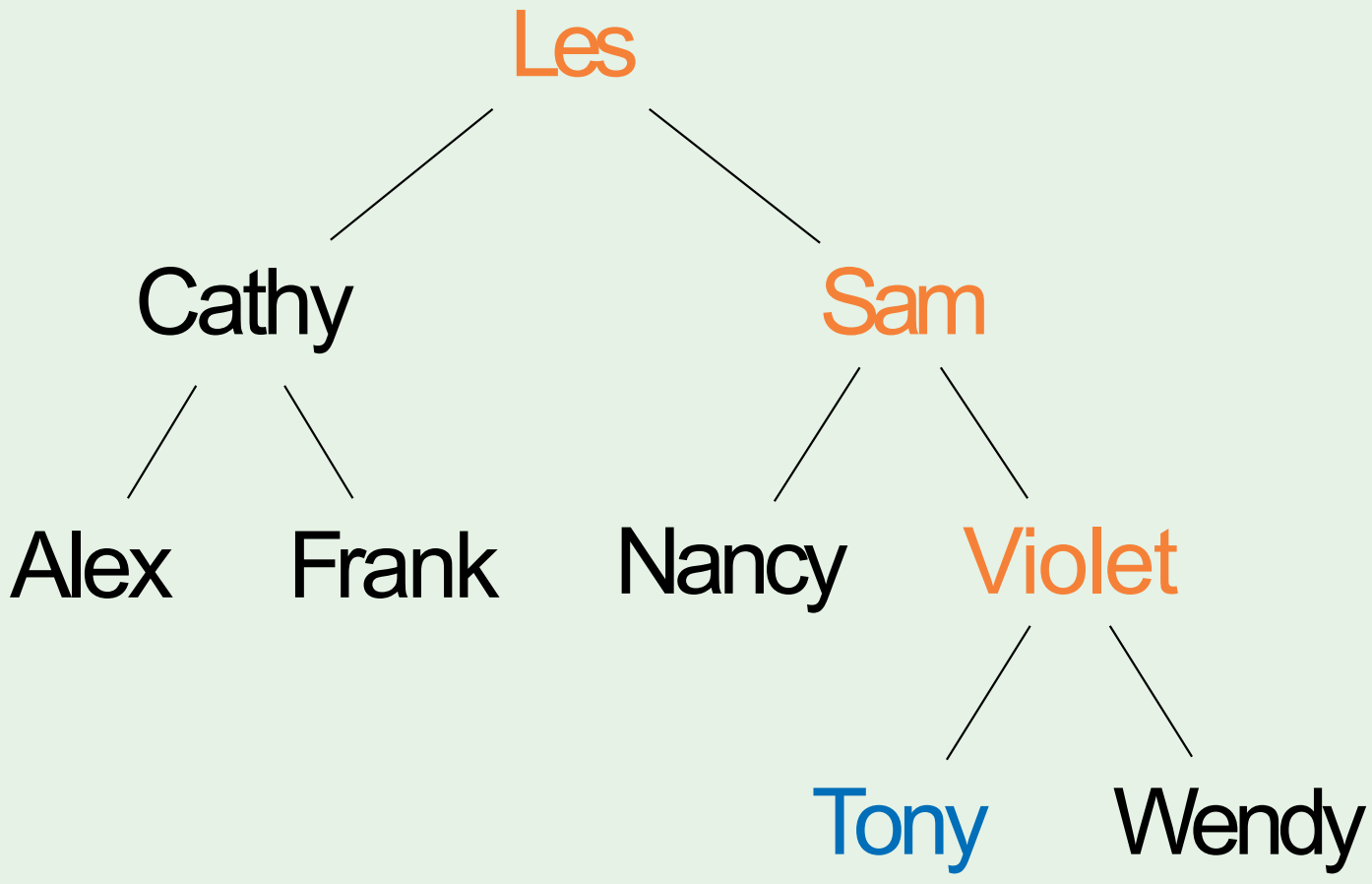
Output: Alex Frank Cathy Nancy

PostOrderTraversal



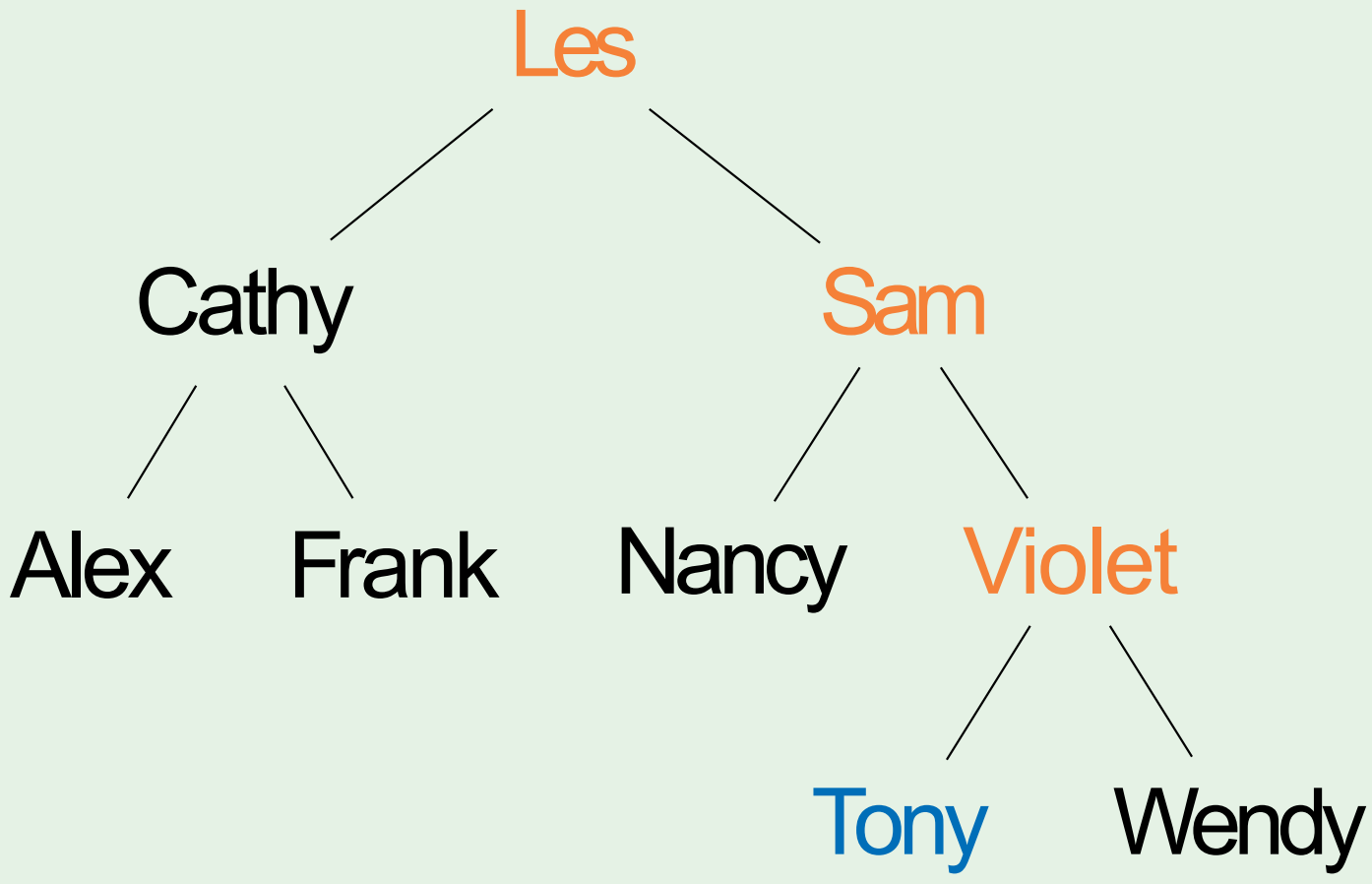
Output: Alex Frank Cathy Nancy

PostOrderTraversal



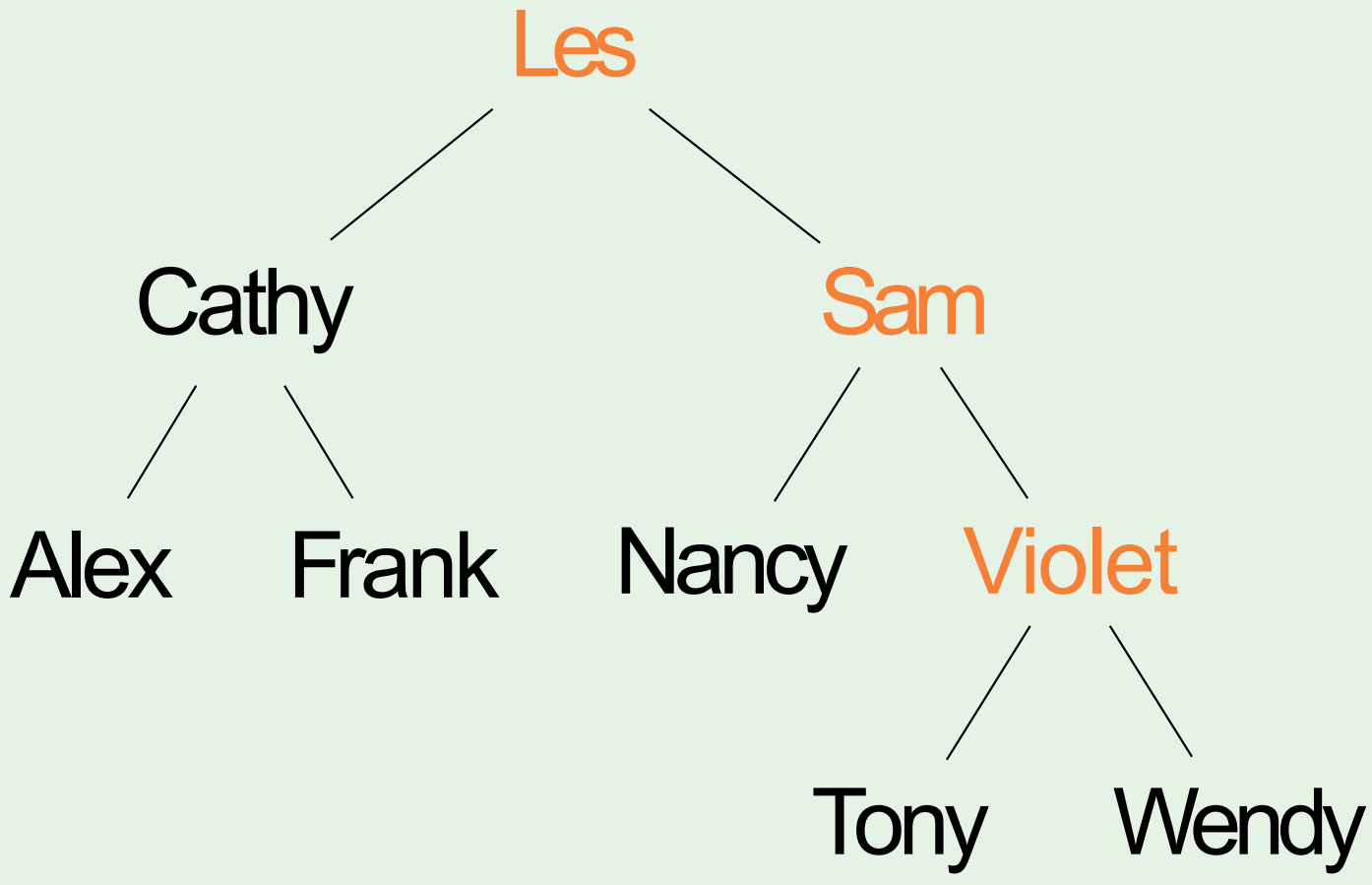
Output: Alex Frank Cathy Nancy Tony

PostOrderTraversal



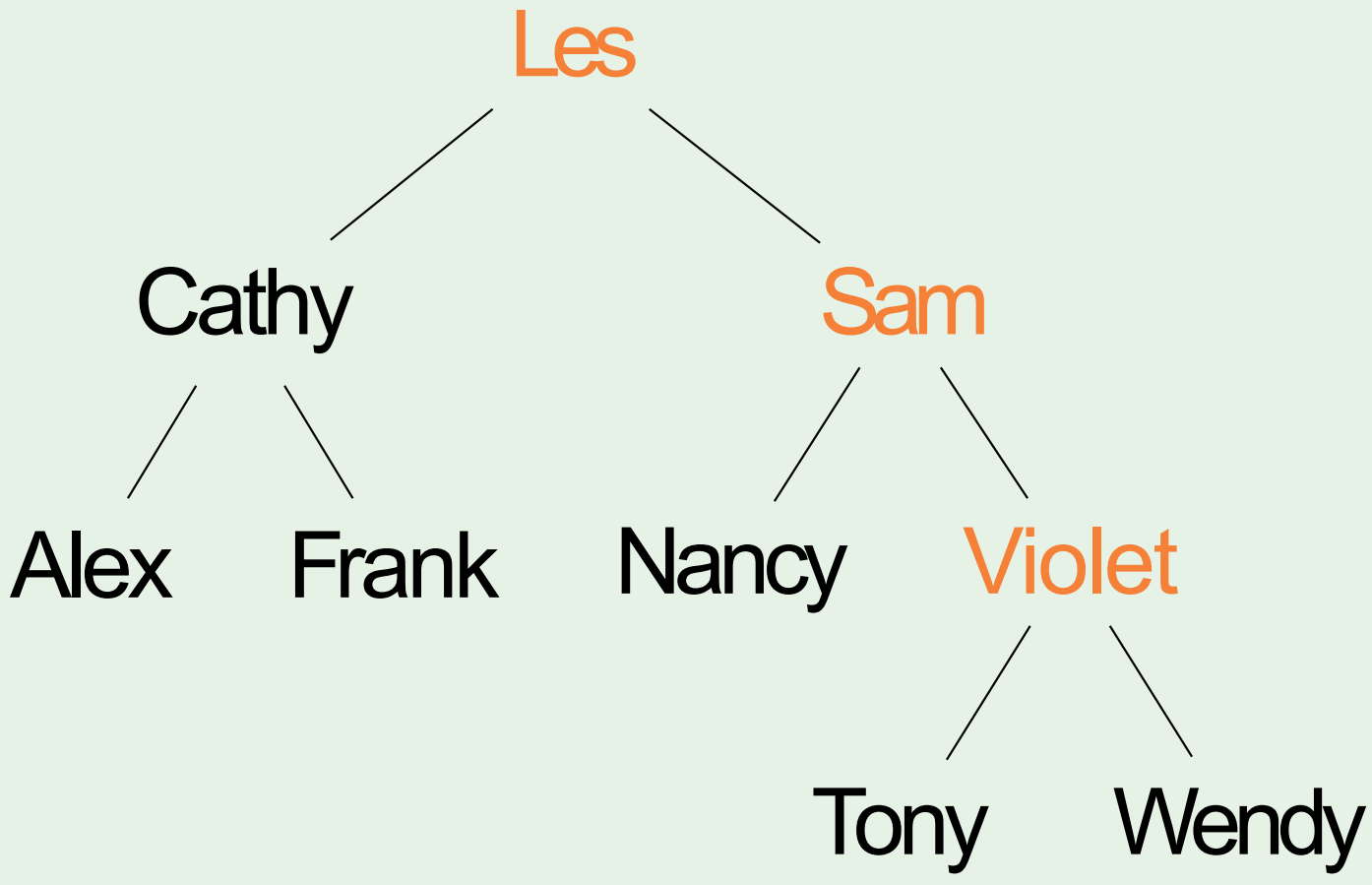
Output: Alex Frank Cathy Nancy Tony

PostOrderTraversal



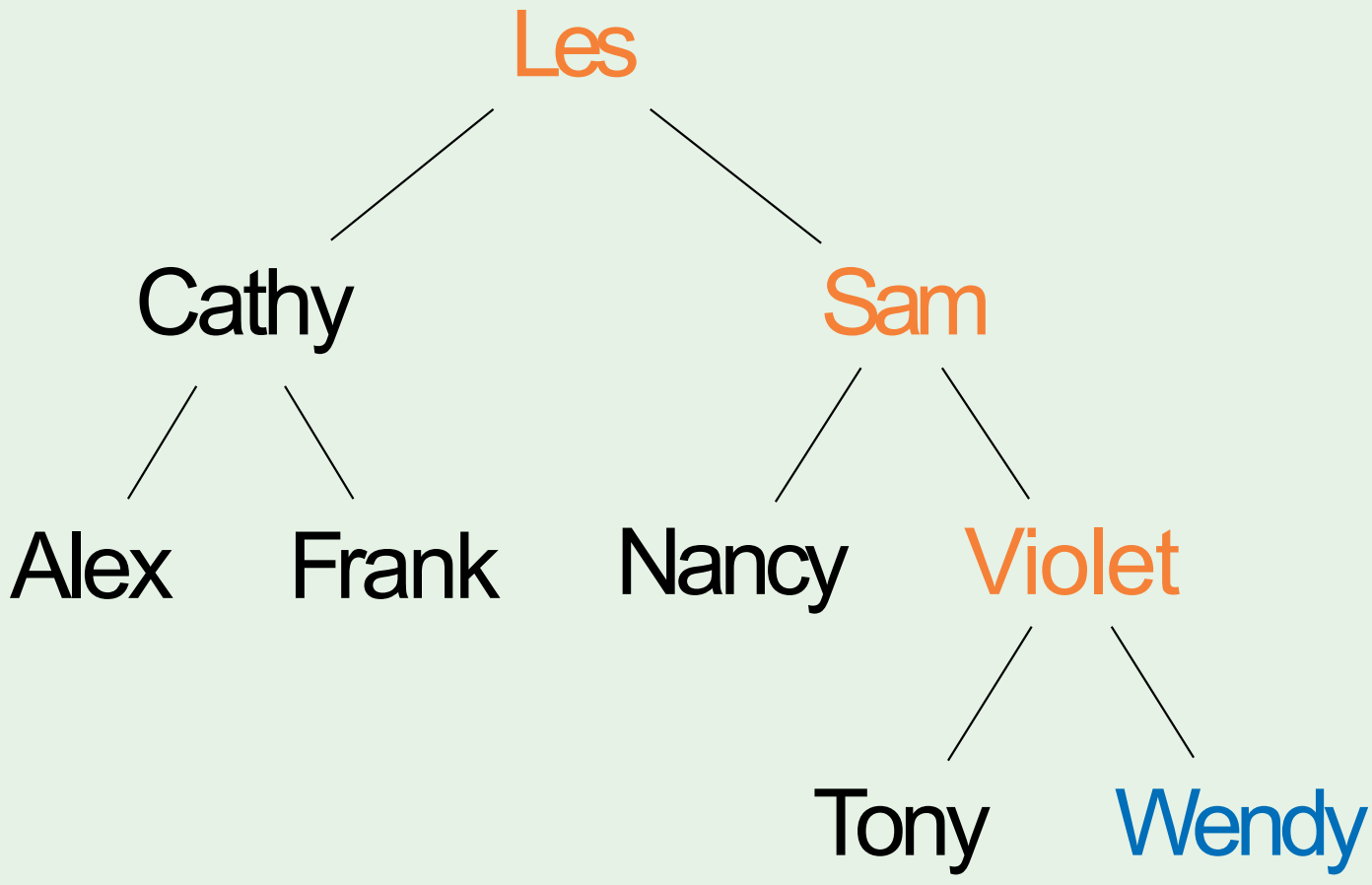
Output: Alex Frank Cathy Nancy Tony

PostOrderTraversal



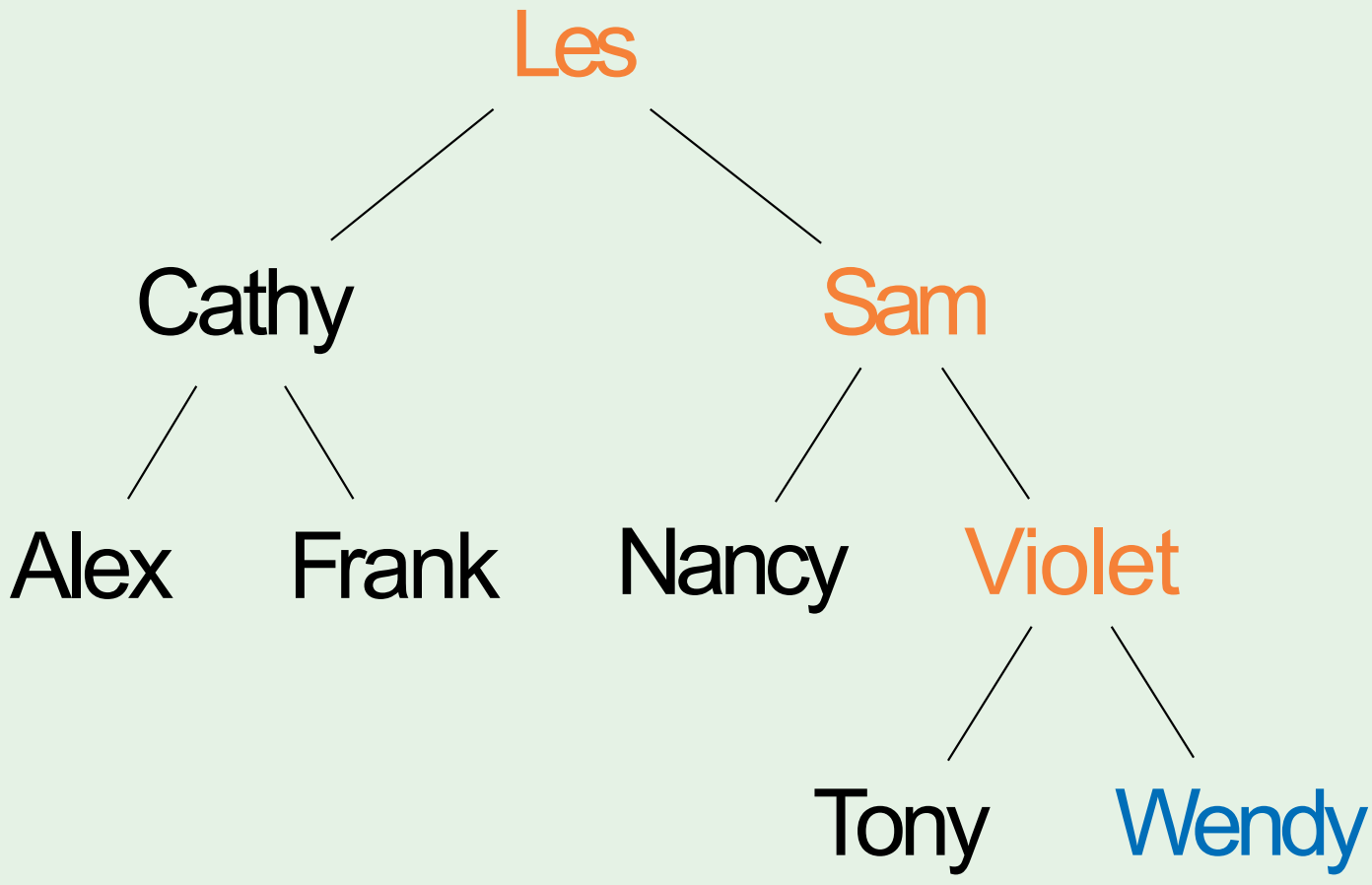
Output: Alex Frank Cathy Nancy Tony

PostOrderTraversal



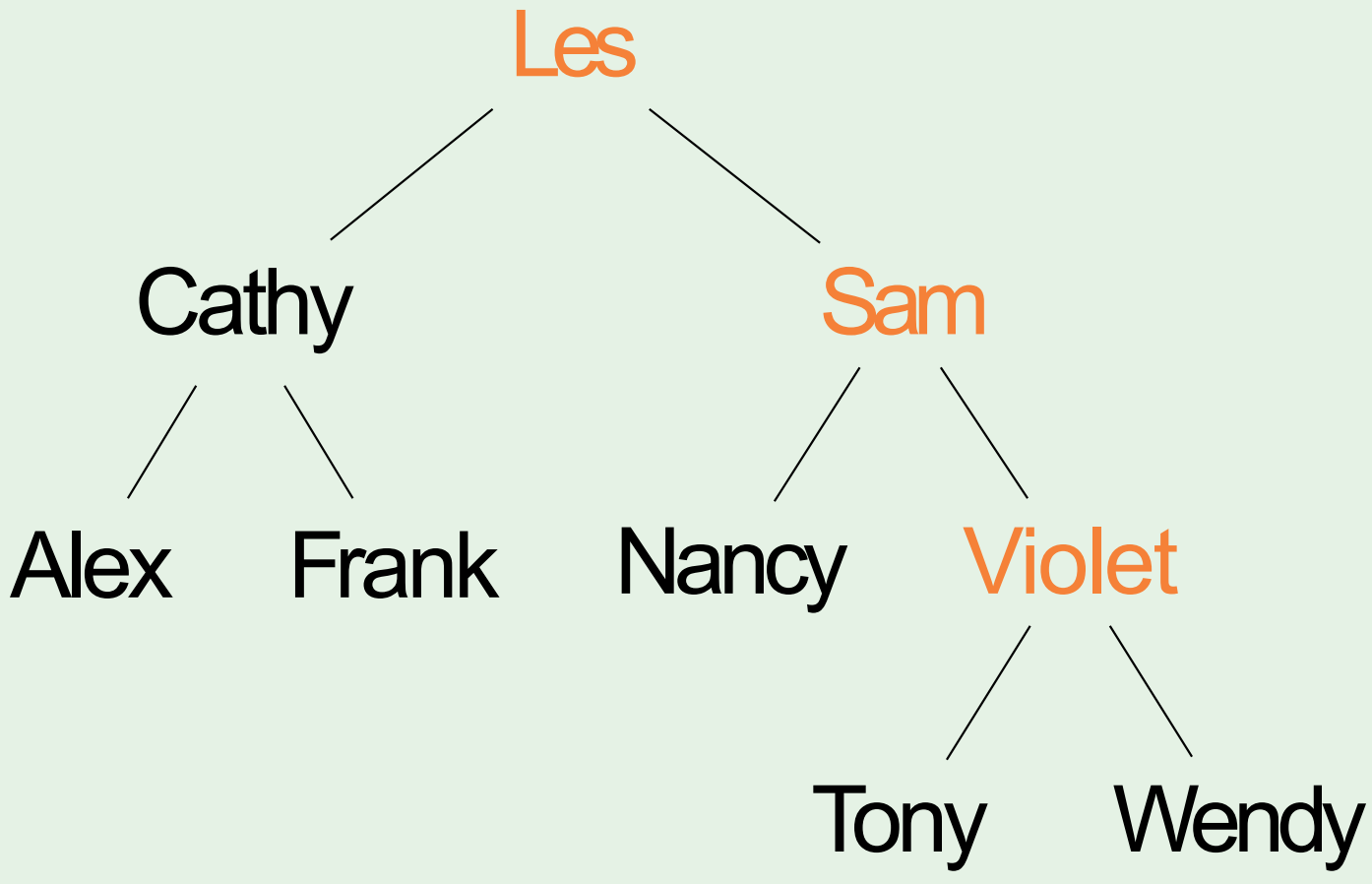
Output: Alex Frank Cathy Nancy Tony Wendy

PostOrderTraversal



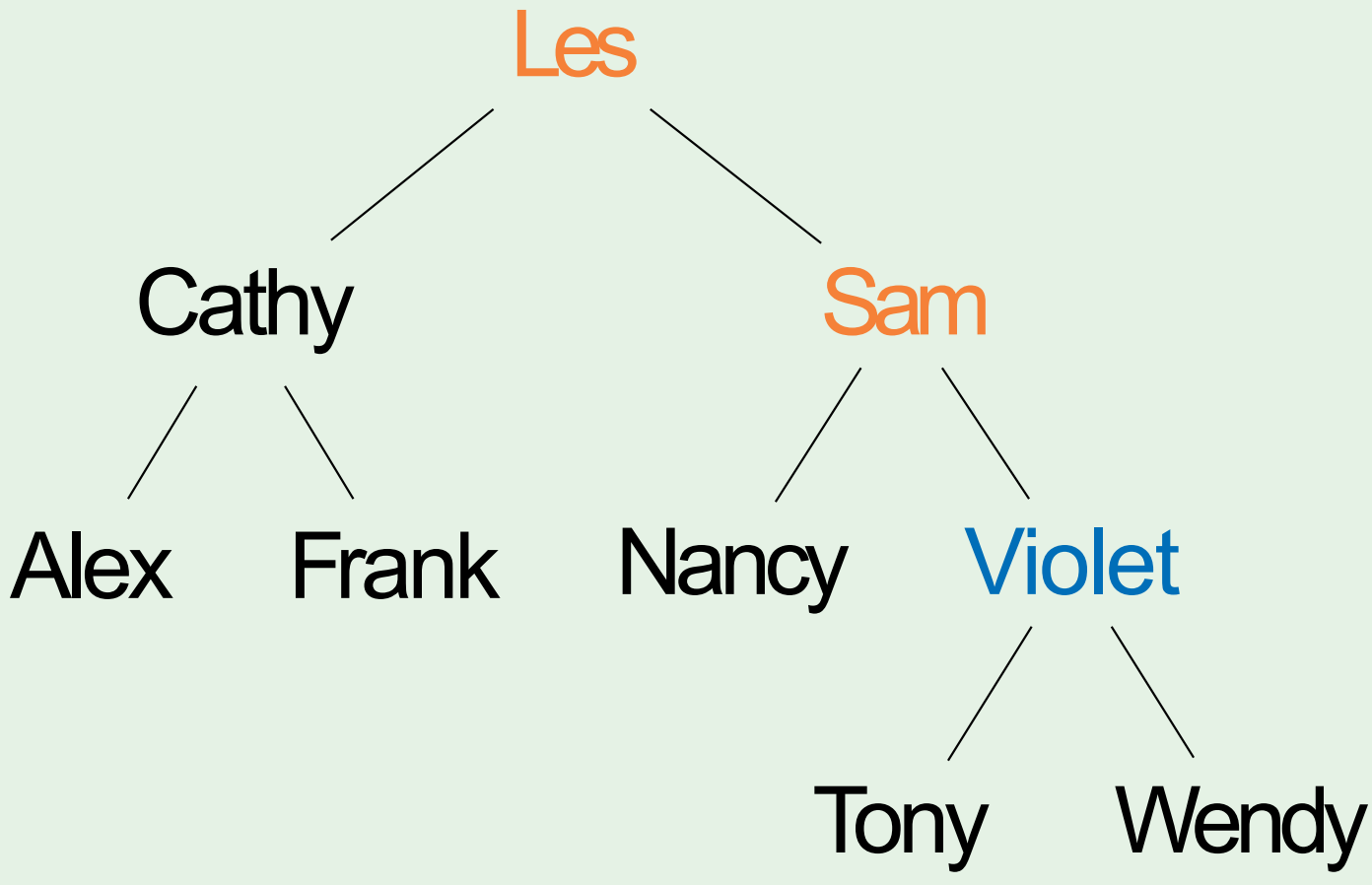
Output: Alex Frank Cathy Nancy Tony Wendy

PostOrderTraversal



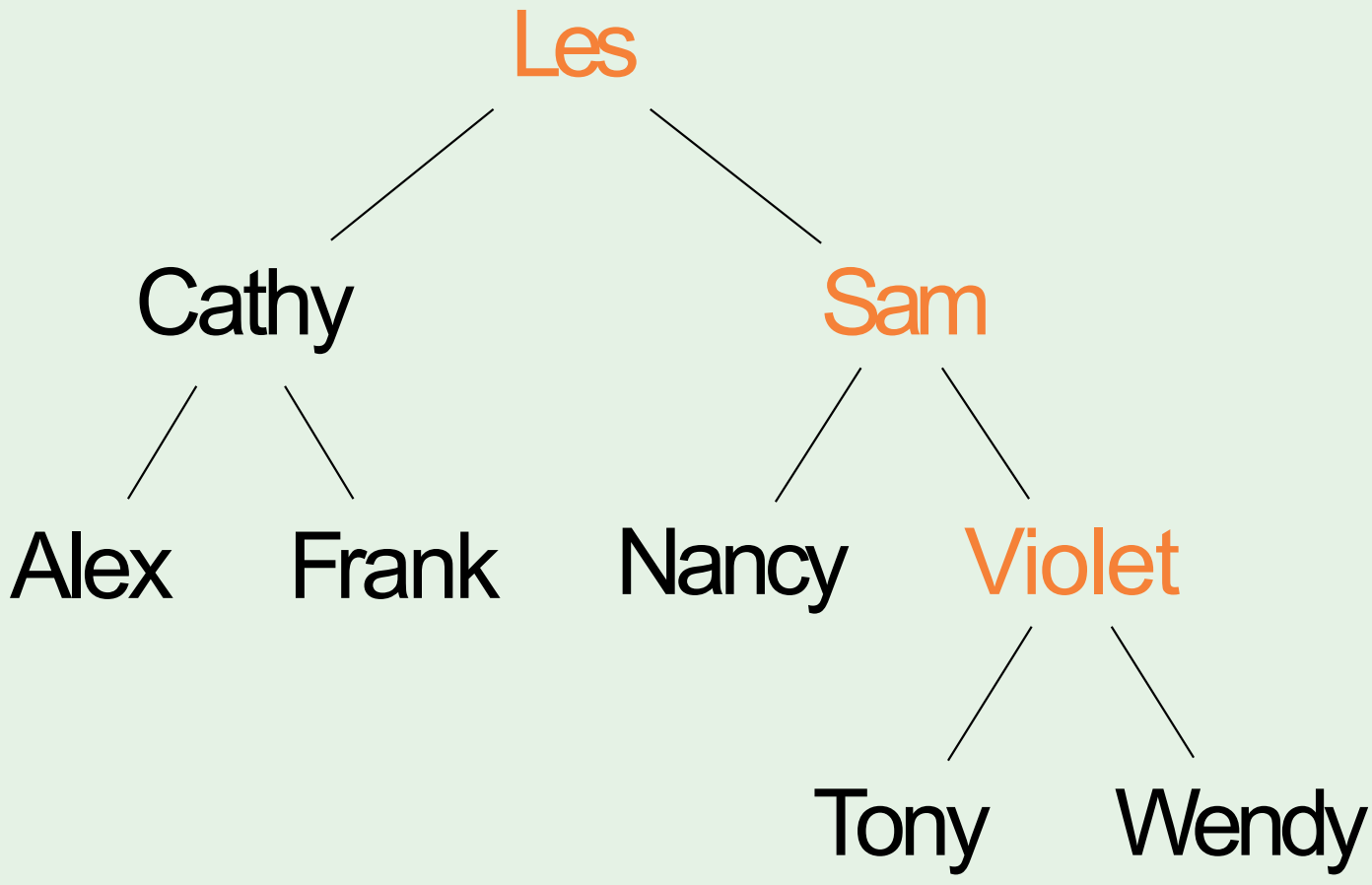
Output: Alex Frank Cathy Nancy Tony Wendy

PostOrderTraversal



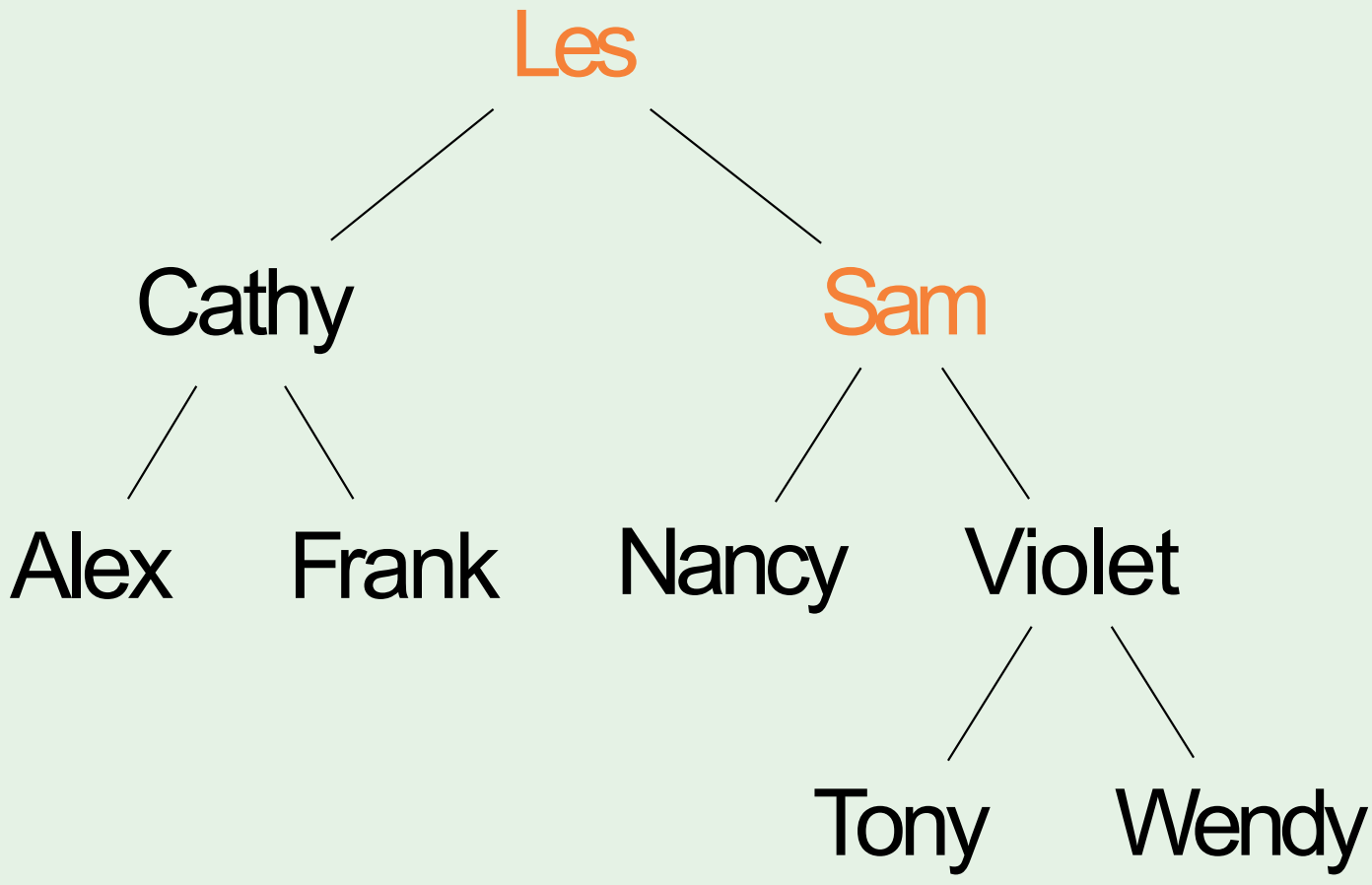
Output: Alex Frank Cathy Nancy Tony Wendy Violet

PostOrderTraversal



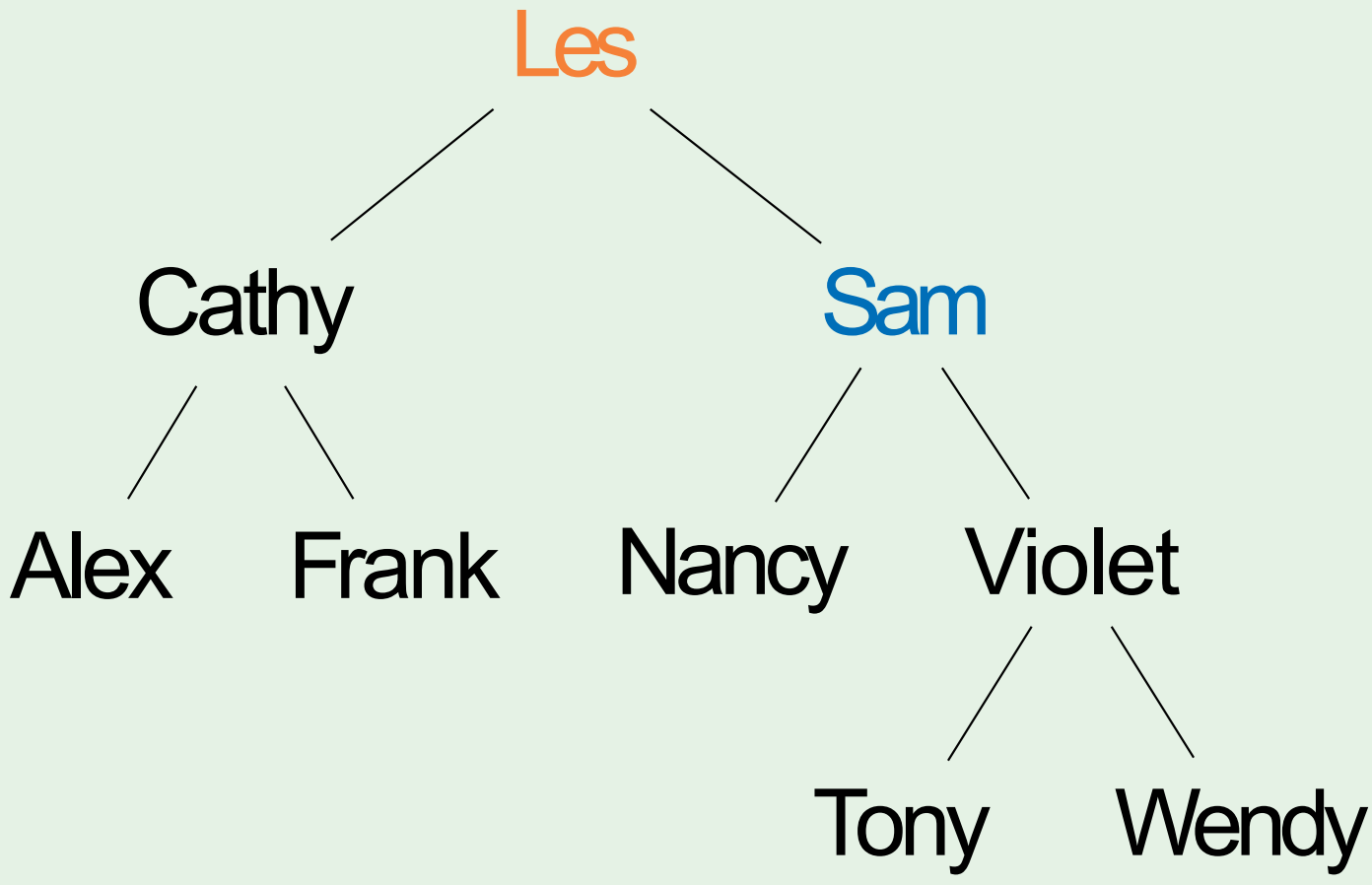
Output: Alex Frank Cathy Nancy Tony Wendy Violet

PostOrderTraversal



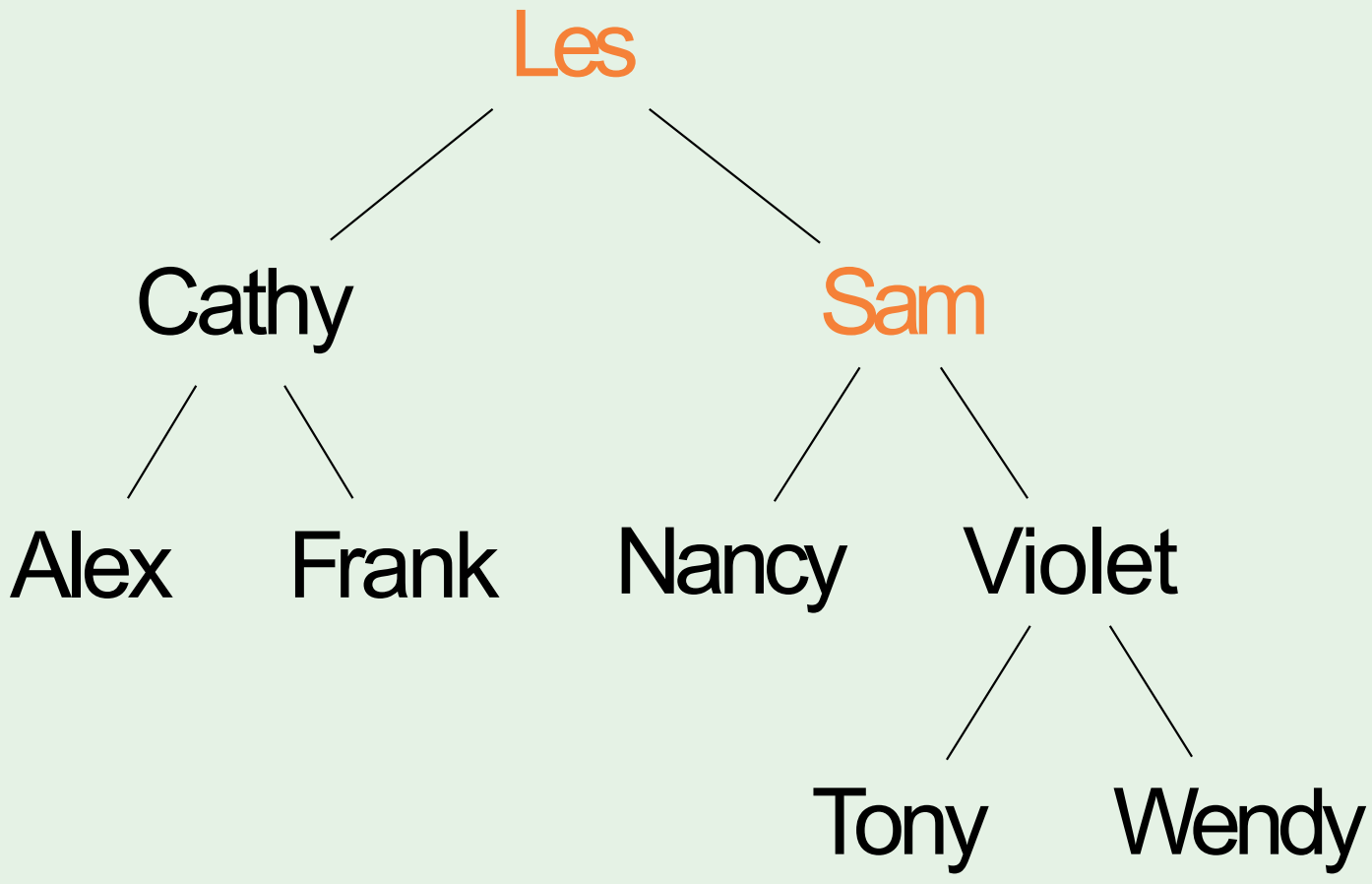
Output: Alex Frank Cathy Nancy Tony Wendy Violet

PostOrderTraversal



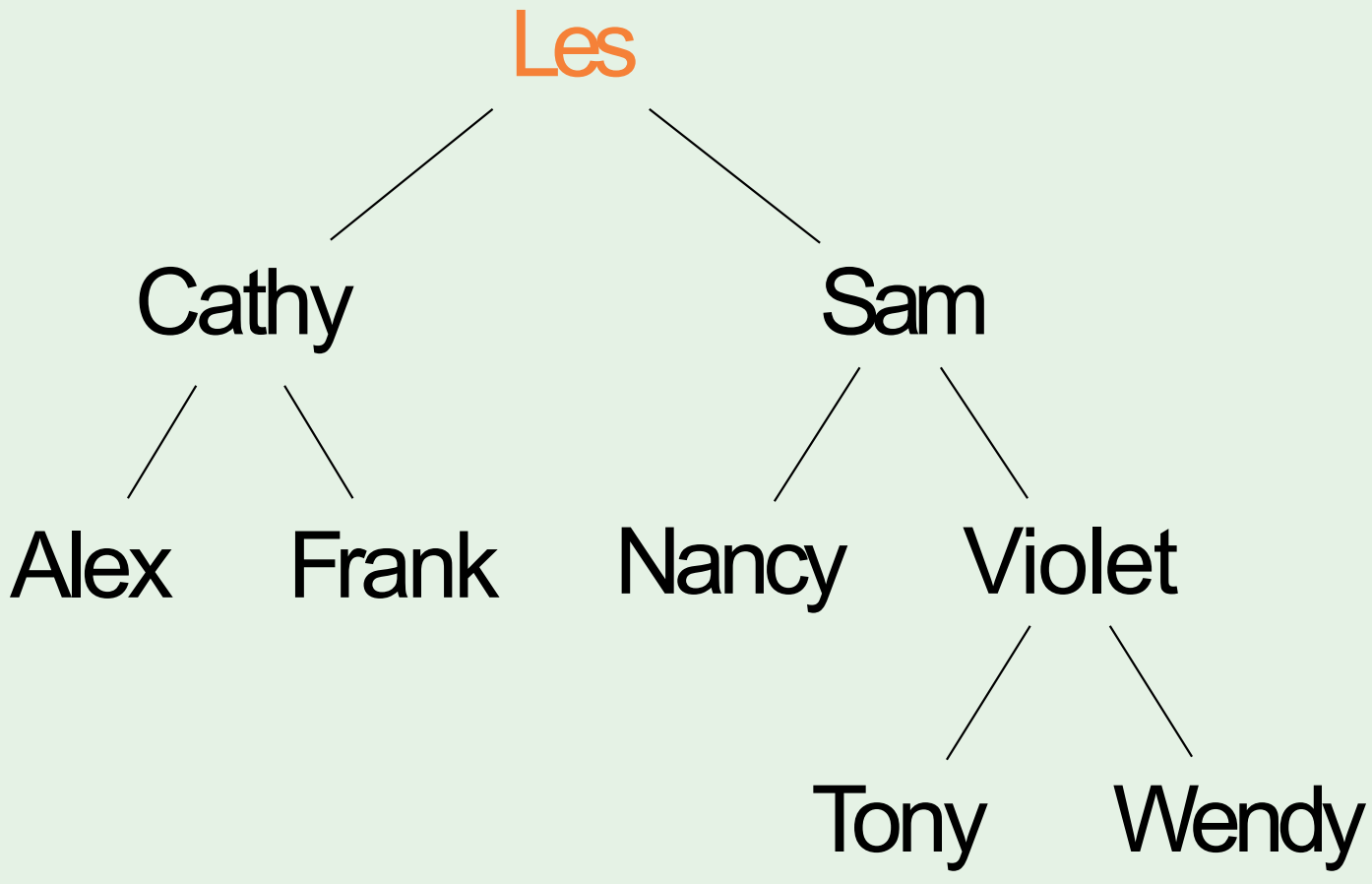
Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam

PostOrderTraversal



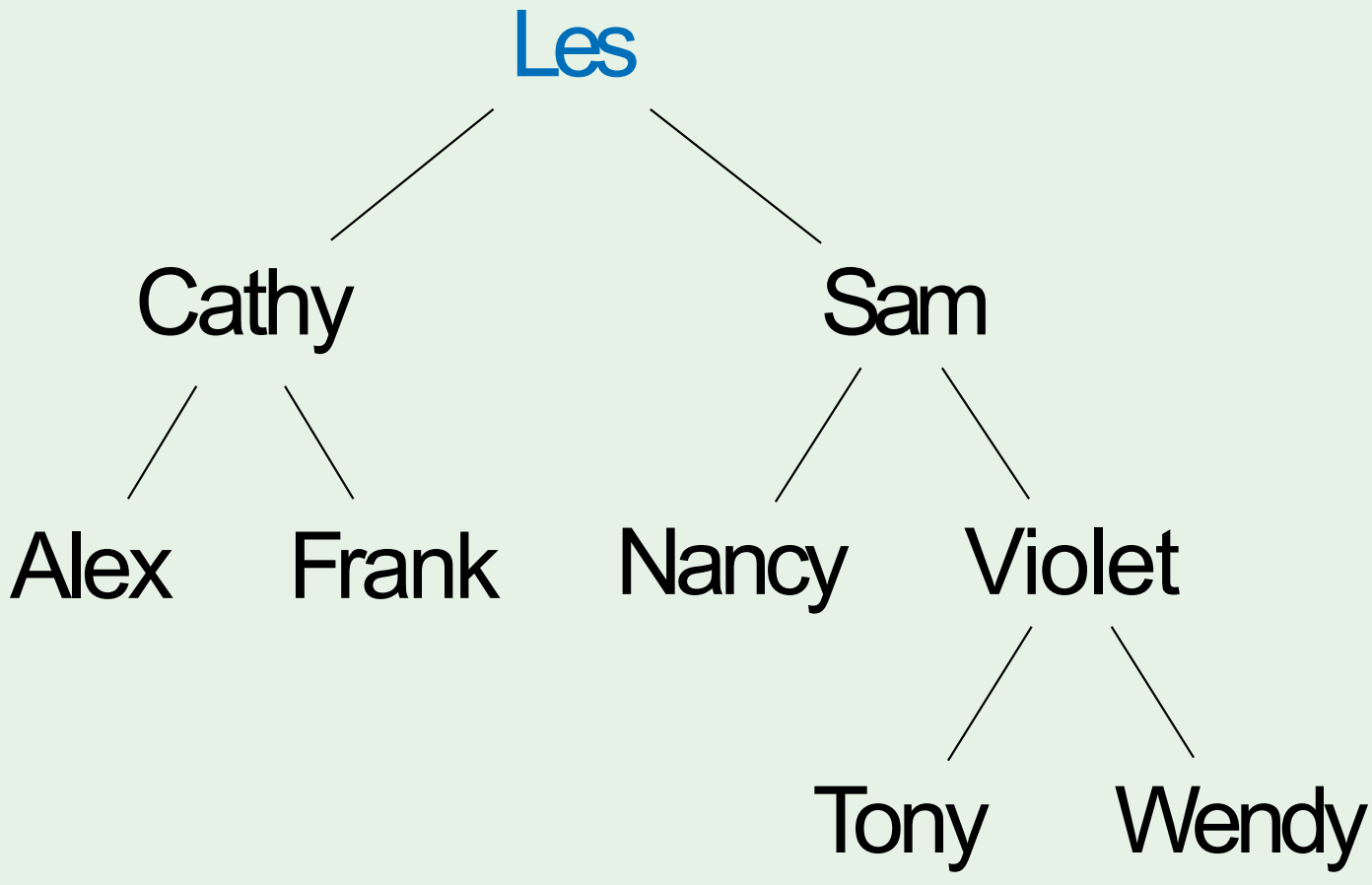
Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam

PostOrderTraversal



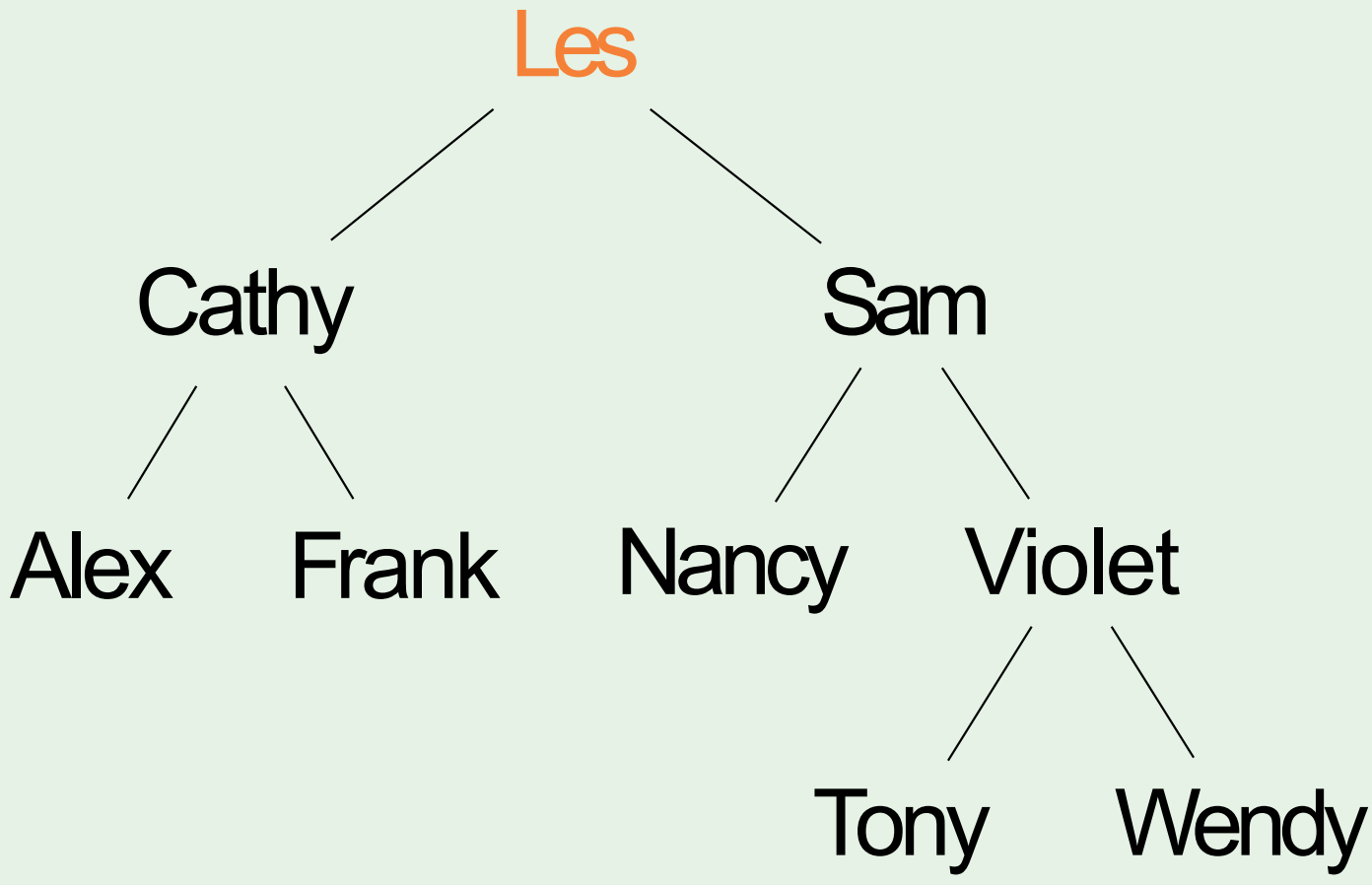
Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam

PostOrderTraversal



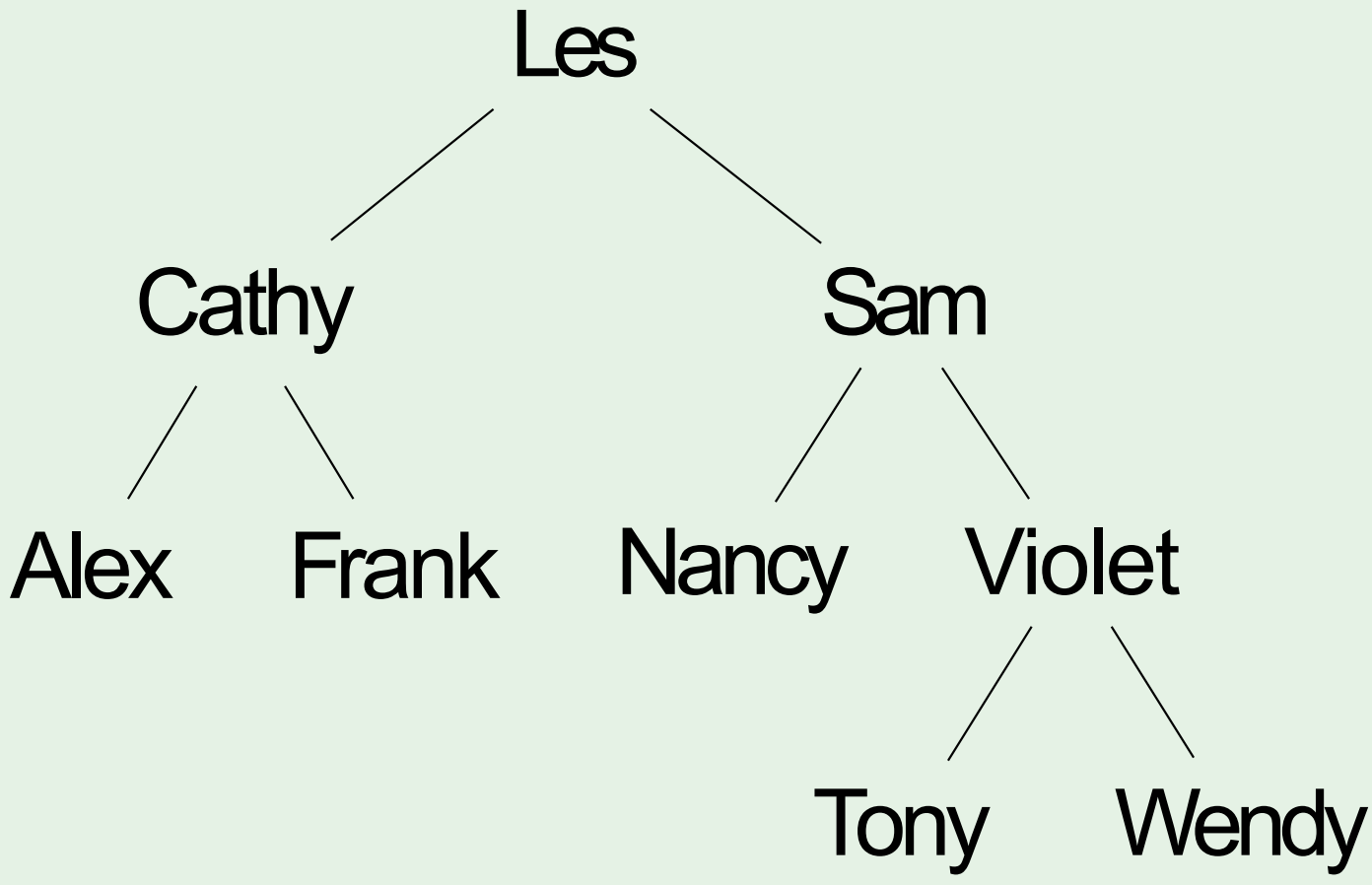
Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam Les

PostOrderTraversal



Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam Les

PostOrderTraversal



Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam Les

Construct a Tree

Inorder sequence: D B E A F C

Can we construct a tree?

Which one is the root node?

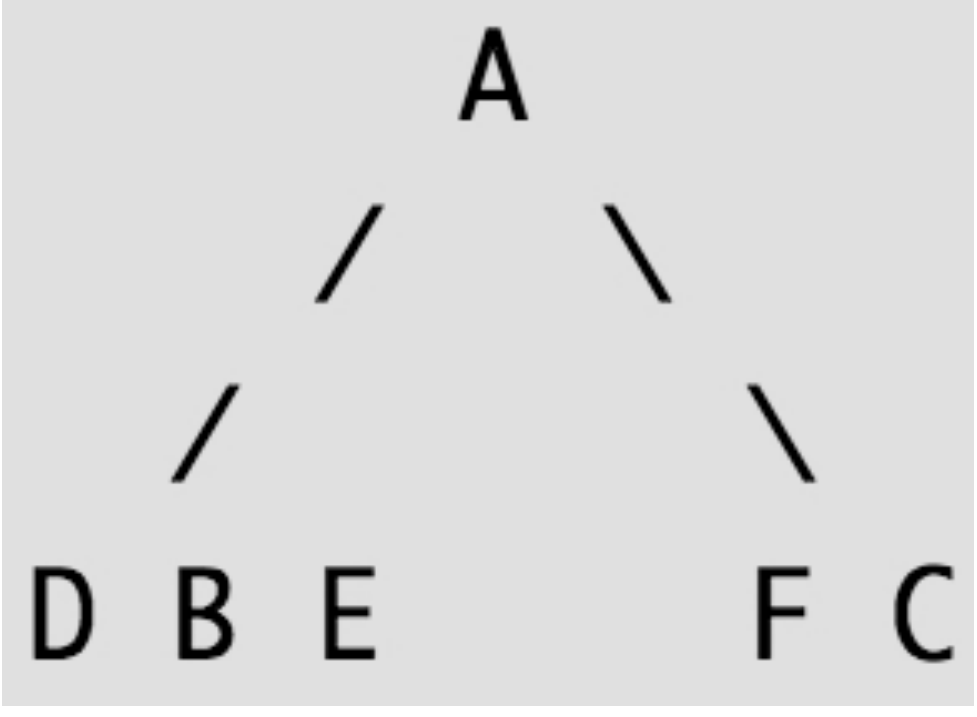
Construct a Tree

Inorder sequence: D B E A F C
Preorder sequence: A B D E C F

- In a Preorder sequence, leftmost element is the root of the tree.
- So we know ‘A’ is root for given sequences.

Construct a Tree

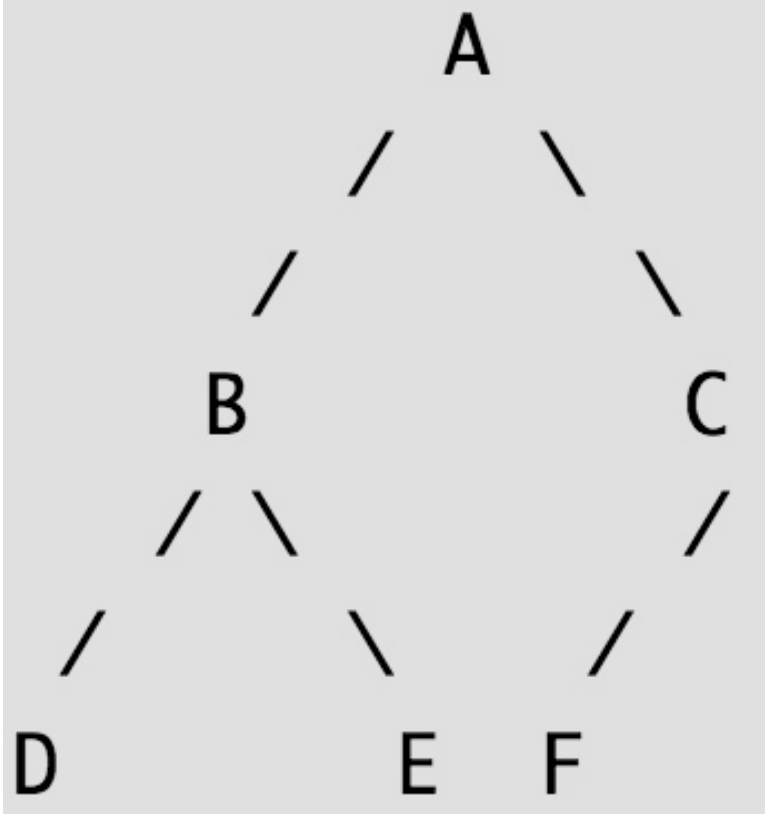
Inorder sequence: D B E A F C
Preorder sequence: A B D E C F



Construct a Tree

Inorder sequence: D B E A F C

Preorder sequence: A B D E C F



Construct a Tree

Inorder sequence: D B E A F C
Preorder sequence: A B D E C F

Algorithm: buildTree()

1. First element in preorder[] will be the root of the tree, here its A.
2. Now the search element A in inorder[], say you find it at position i, once you find it, make note of elements which are left to *i* (this will construct the left-subtree) and elements which are right to *i* (this will construct the right-subtree).
3. See this step above and recursively construct left subtree and link it root.left and recursively construct right subtree and link it root.right.

Construct a Tree

Inorder sequence: D B E A F C
Preorder sequence: A B D E C F

```
/* Recursive function to construct binary of size len from
   Inorder traversal in[] and Preorder traversal pre[].  Initial values
   of inStrt and inEnd should be 0 and len -1.  The function doesn't
   do any error checking for cases where inorder and preorder
   do not form a tree */
struct node* buildTree(char in[], char pre[], int inStrt, int inEnd)
{
  static int preIndex = 0;

  if(inStrt > inEnd)
     return NULL;

  /* Pick current node from Preorder traversal using preIndex
    and increment preIndex */
  struct node *tNode = newNode(pre[preIndex++]);

  /* If this node has no children then return */
  if(inStrt == inEnd)
    return tNode;

  /* Else find the index of this node in Inorder traversal */
  int inIndex = search(in, inStrt, inEnd, tNode->data);

  /* Using index in Inorder traversal, construct left and
     right subtress */
  tNode->left = buildTree(in, pre, inStrt, inIndex-1);
  tNode->right = buildTree(in, pre, inIndex+1, inEnd);

  return tNode;
}
```

Construct a Tree

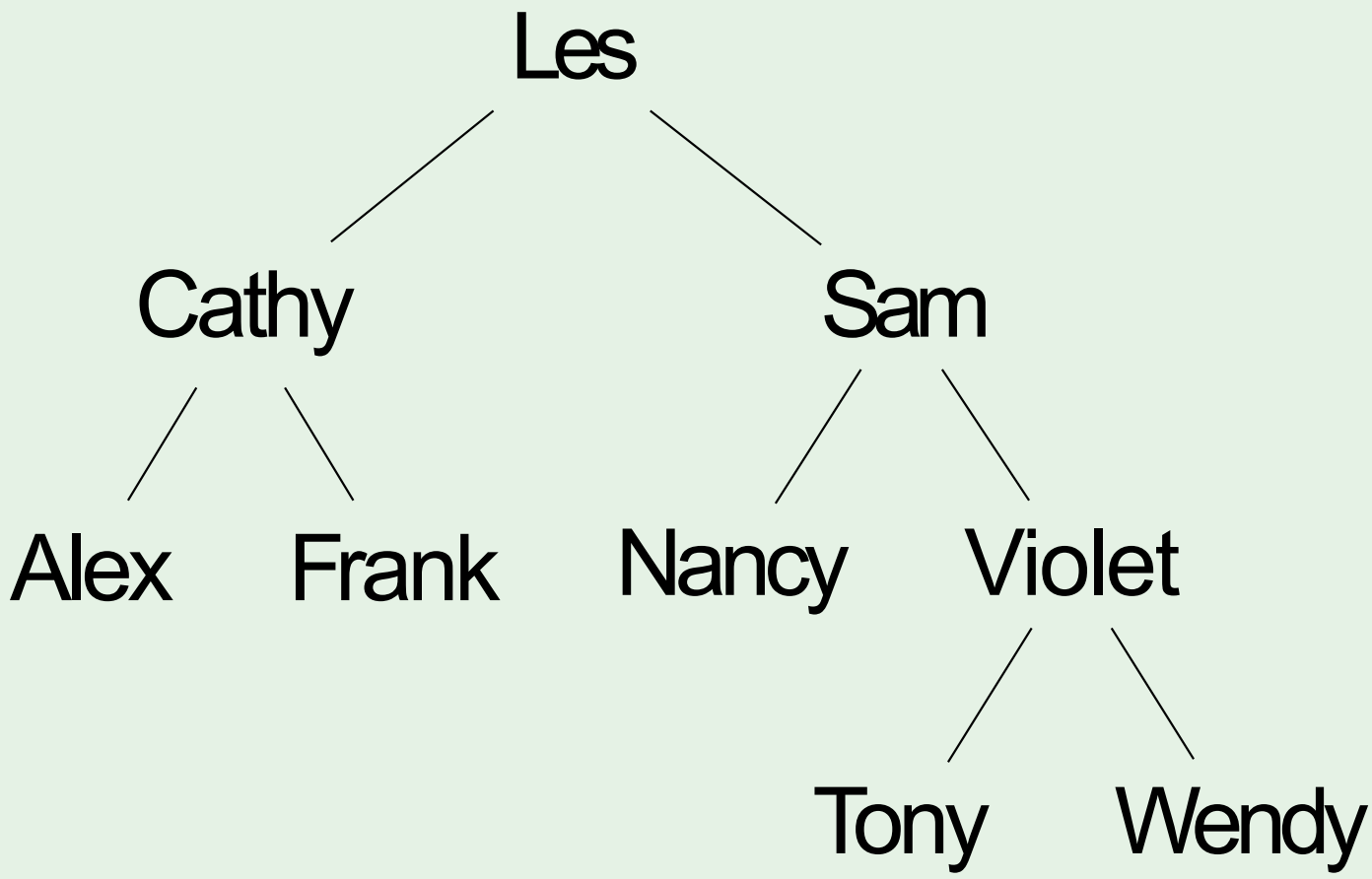
Inorder sequence: D B E A F C
Preorder sequence: A B D E C F

```
/* UTILITY FUNCTIONS */
/* Function to find index of value in arr[start...end]
   The function assumes that value is present in in[] */
int search(char arr[], int strt, int end, char value)
{
  int i;
  for(i = strt; i <= end; i++)
  {
    if(arr[i] == value)
      return i;
  }
}
```

Assignment

- Can you do it for InOrder and PostOrder?
- Validity check - InOrder and PreOrder are given, but tree is not valid!
- Validity check – InOrder, and PostOrder are given, but tree is not valid!

LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy Violet Tony Wendy

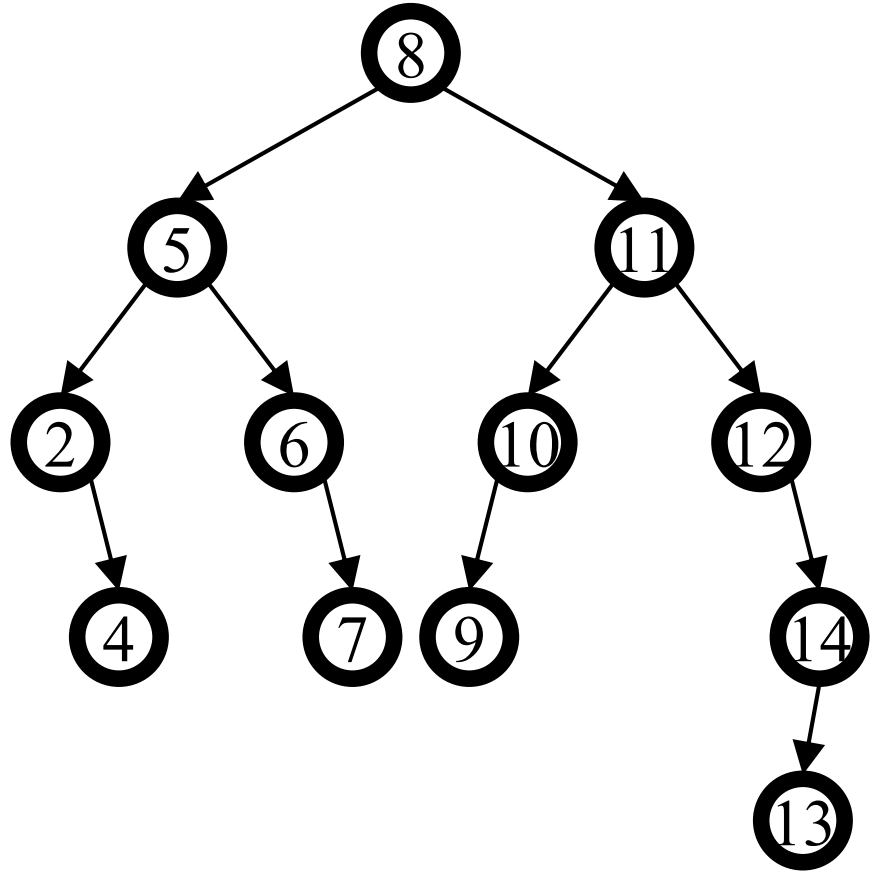
Assignment

- Can you do it for InOrder and PostOrder?
- Validity check - InOrder and PreOrder are given, but tree is not valid!
- Validity check – InOrder, and PostOrder are given, but tree is not valid!
- Implement Breadth-First traversal

Binary Search Tree

Binary Search tree property

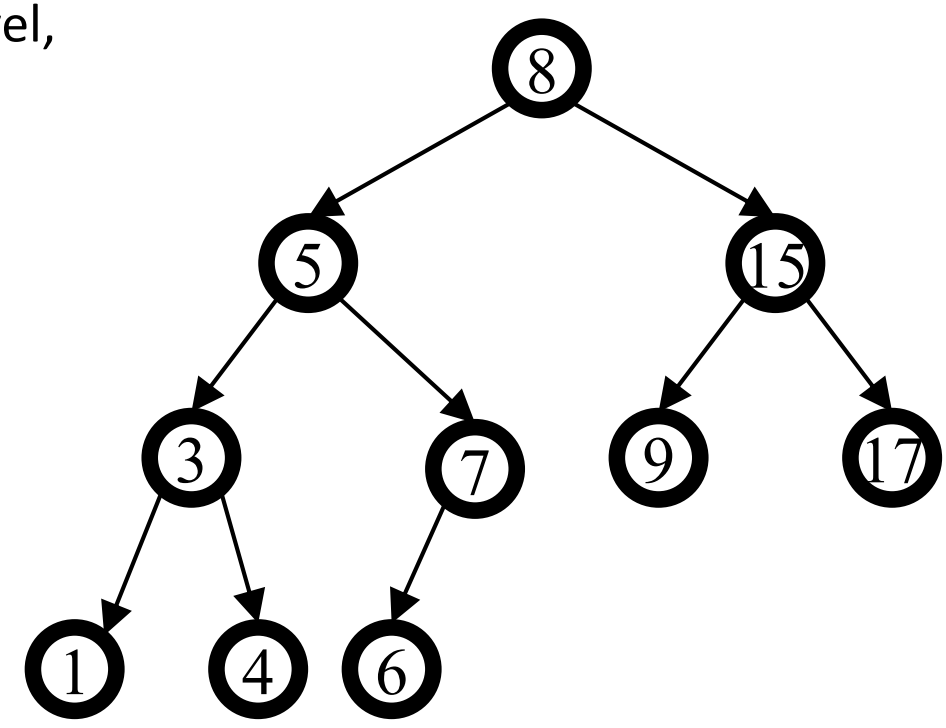
- all keys in left subtree smaller than root's key
- all keys in right subtree larger than root's key
- result:
  - easy to find any given key
  - Insert/delete by changing links



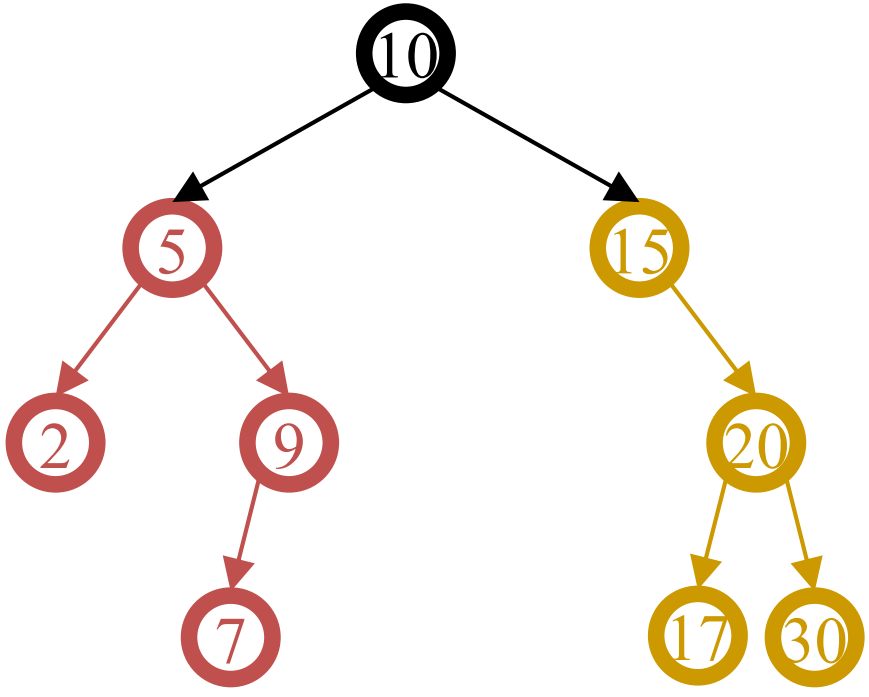
Complete Binary Search Tree

Complete binary search tree:

Links are completely filled,
except possibly bottom level,
which is filled left-to-right.



In-Order Traversal

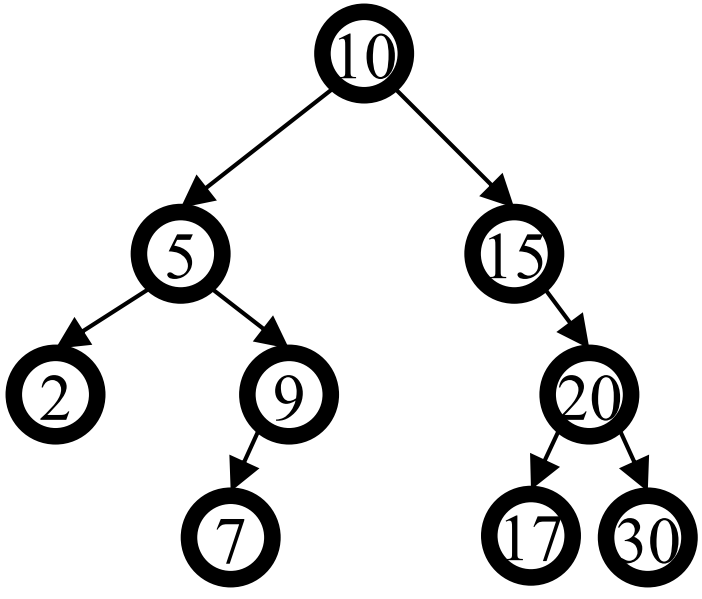


What does this guarantee with a BST?

In order listing:

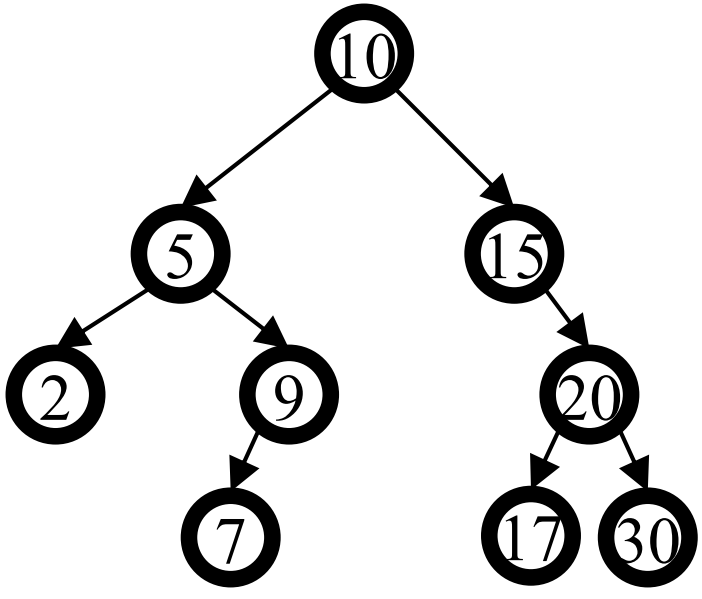
2→5→7→9→10→15→17→20→30

Recursive Find




```
Node *
find(Comparable key, Node * t)
{
  if (t == NULL) return t;
  else if (key < t->key)
    return find(key, t->left);
  else if (key > t->key)
    return find(key, t->right);
  else
    return t;
}
```

Iterative Find




```
Node *
find(Comparable key, Node * t)
{
  while (t != NULL && t->key != key)
  {
    if (key < t->key)
      t = t->left;
    else
      t = t->right;
  }

  return t;
}
```

Assignment

- Can you do it for InOrder and PostOrder?
- Validity check - InOrder and PreOrder are given, but tree is not valid!
- Validity check – InOrder, PreOrder, and PostOrder are given, but tree is not valid!
- Implement Breadth-First traversal.
- Given a sorted list – create a BST. Implement recursive and iterative search. – tell me the complexity?
- Given a tree can you check whether it's a BST?

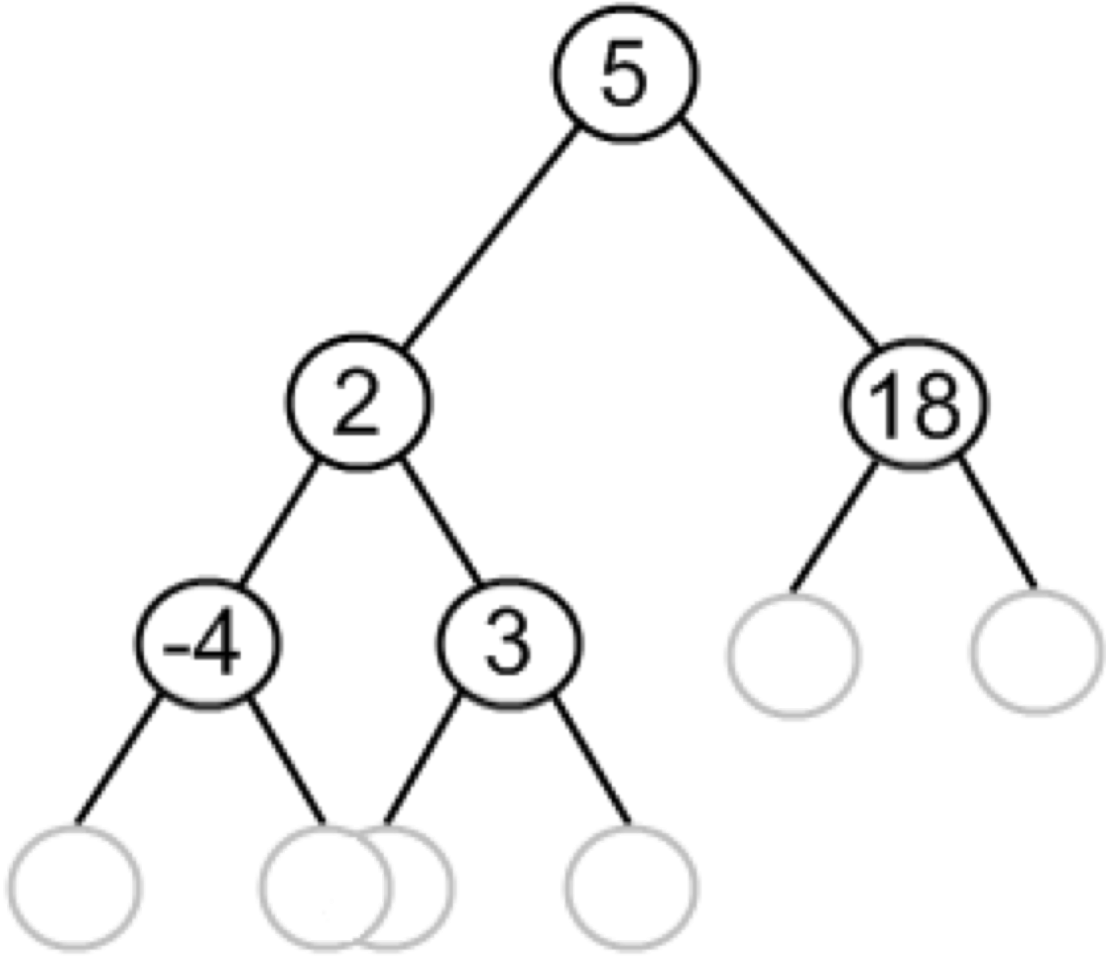
Insert Node - BST

Adding a value to BST can be divided into two stages:

- search for a place to put a new element;
- insert the new element to this place.

Let us see these stages in more detail.

Search for the Place



Search for the Place - Animation

Paly with it –

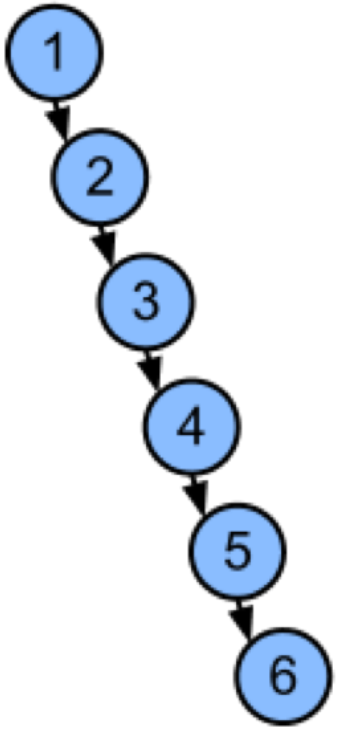
https://visualgo.net/bn/bst

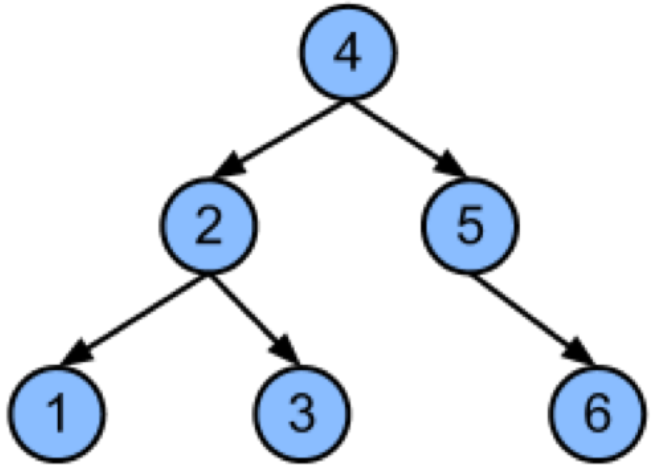
Insert Node - BST

```
/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

Insert Node – BST - Complexity

Time Complexity: The worst case time complexity of search and insert operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).





Delete Node - BST

Remove operation on binary search tree is more **complicated**, than add and search. Basically, in can be divided into two stages:

- search for a node to remove;
- if the node is found, run remove algorithm.

Easy right!

Condition – the tree has to be BST after deletion!

Delete Node – BST

1. *Node to be deleted is leaf:*
2. *Node to be deleted has only one child:*
3. *Node to be deleted has two children:*

Delete Node - BST

1. *Node to be deleted is leaf or no child*

   Simply remove from the tree.

Delete Node - BST

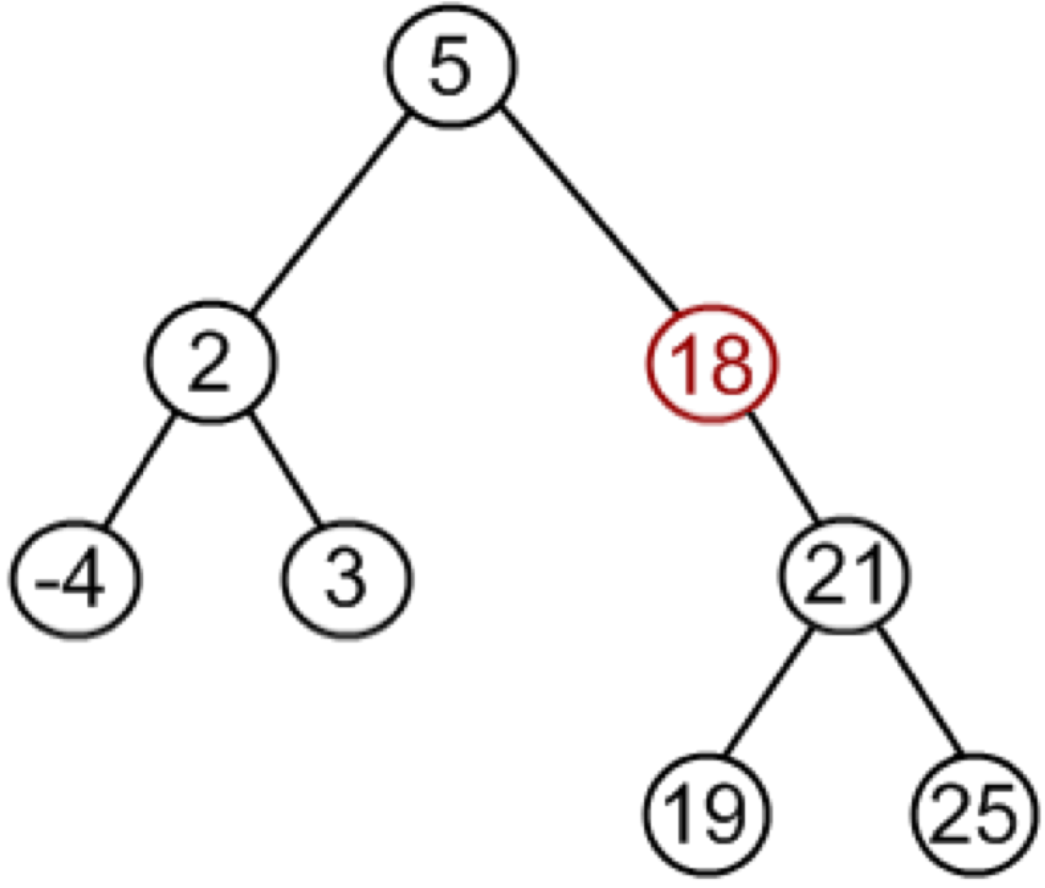
2. Node to be deleted has only one child:

Copy the child to the node and delete the child

Delete Node - BST

2. Node to be deleted has only one child:

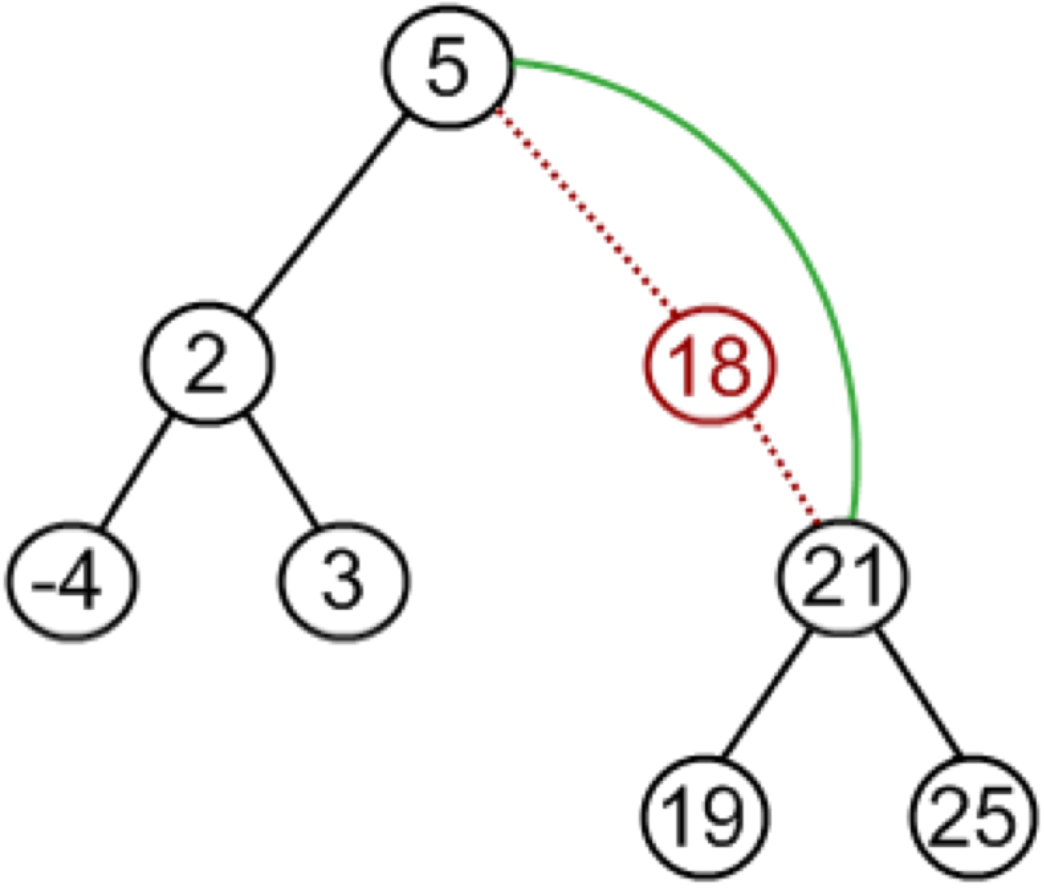
Copy the child to the node and delete the child



Delete Node - BST

2. Node to be deleted has only one child:

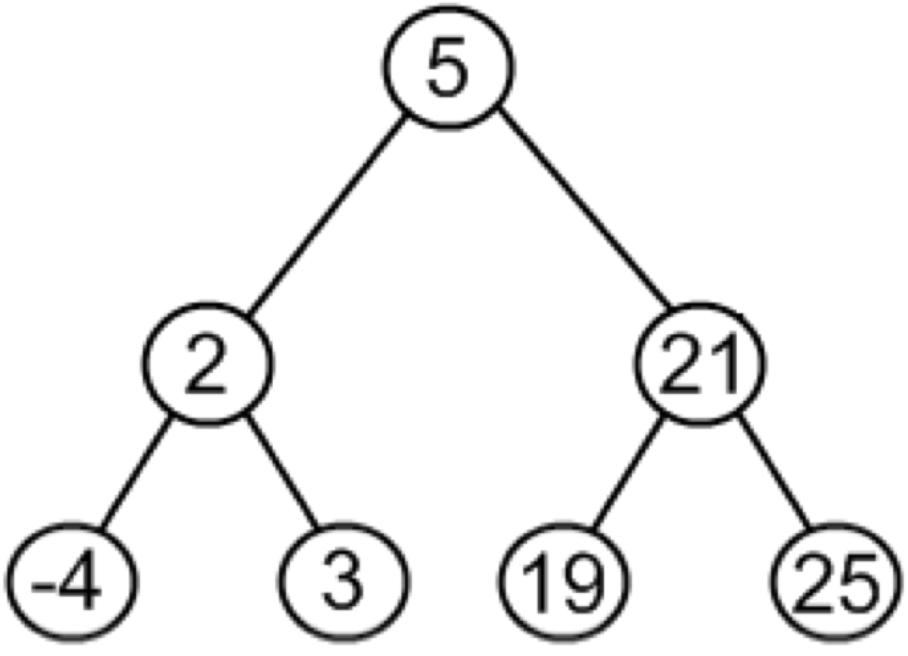
Copy the child to the node and delete the child



Delete Node - BST

2. Node to be deleted has only one child:

Copy the child to the node and delete the child



Delete Node - BST

3. Node to be deleted has two children:

Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

Delete Node – BST

1. *Node to be deleted is leaf:*
2. *Node to be deleted has only one child:*
3. *Node to be deleted has two children:*

The algorithm to delete a node from BST is called **Hibbard deletion algorithm** (named after its founder Hibbard who first described the algorithm in 1962)

Delete Node - BST

```
/* Given a binary search tree and a key, this function deletes the key
   and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
```

Delete Node - BST

```
// if key is same as root's key, then This is the node
// to be deleted
else
{
    // node with only one child or no child
    if (root->left == NULL)
    {
        struct node *temp = root->right;
        free(root);
        return temp;
    }
    else if (root->right == NULL)
    {
        struct node *temp = root->left;
        free(root);
        return temp;
    }
```

Delete Node - BST

```
        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

Delete Node - BST

```
/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}
```

https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/

Binary Search Tree

Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

Binary Search Tree

Applications of binary trees

- Binary Search Tree - Used in *many* search applications where data is constantly entering/leaving, such as the `map` and `set` objects in many languages' libraries.
- Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered.
- Binary Tries - Used in almost every high-bandwidth router for storing router-tables.
- Hash Trees - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.
- Heaps - Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, Quality-of-Service in routers, and A* *(path-finding algorithm used in AI applications, including robotics and video games)*. Also used in heap-sort.
- Huffman Coding Tree (Chip Uni) - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.
- GGM Trees - Used in cryptographic applications to generate a tree of pseudo-random numbers.
- Syntax Tree - Constructed by compilers and (implicitly) calculators to parse expressions.
- Treap - Randomized data structure used in wireless networking and memory allocation.
- T-tree - Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so.

https://stackoverflow.com/questions/2130416/what-are-the-applications-of-binary-trees

Assignment

- Insert in BST
- Delete in BST

As mentioned above, the order in which values are inserted determines what BST is built (inserting the same values in different orders can result in different final BSTs). Draw the BST that results from inserting the values 1 to 7 in each of the following orders (reading from left to right):

- 5 3 7 6 2 1 4
- 1 2 3 4 5 6 7
- 4 3 5 2 6 1 7

Open Google - Assignment

Spiral Order Traversal:

(1,2,3,7,6,5,4)
(1,3,2,4,5,6,7)