Basic Data Structures: Arrays and Linked Lists

Neil Rhodes

Department of Computer Science and Engineering
University of California, San Diego

Data Structures
Data Structures and Algorithms

Outline

1. Arrays
2. Linked Lists

```
long arr[] = new long[5];
```

```
long arr[5];
```

```
arr = [None] * 5
```

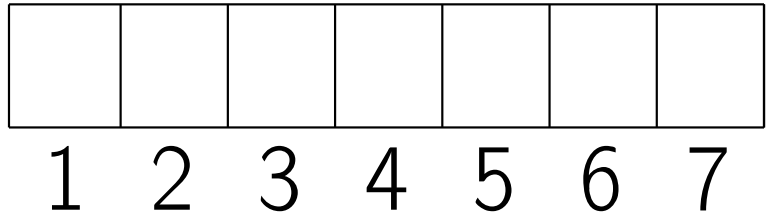
1	5	17	3	25

1	5	17	3	25
8	2	36	5	3

Definition

Array:

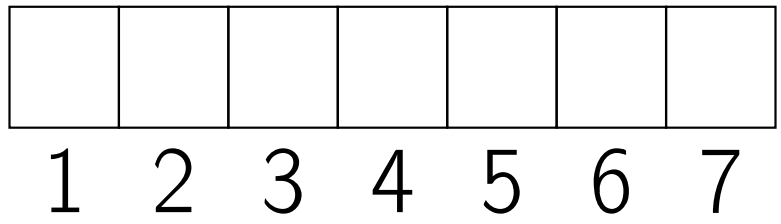
Contiguous area of memory consisting of equal-size elements indexed by contiguous integers.



What's Special About Arrays?

Constant-time access

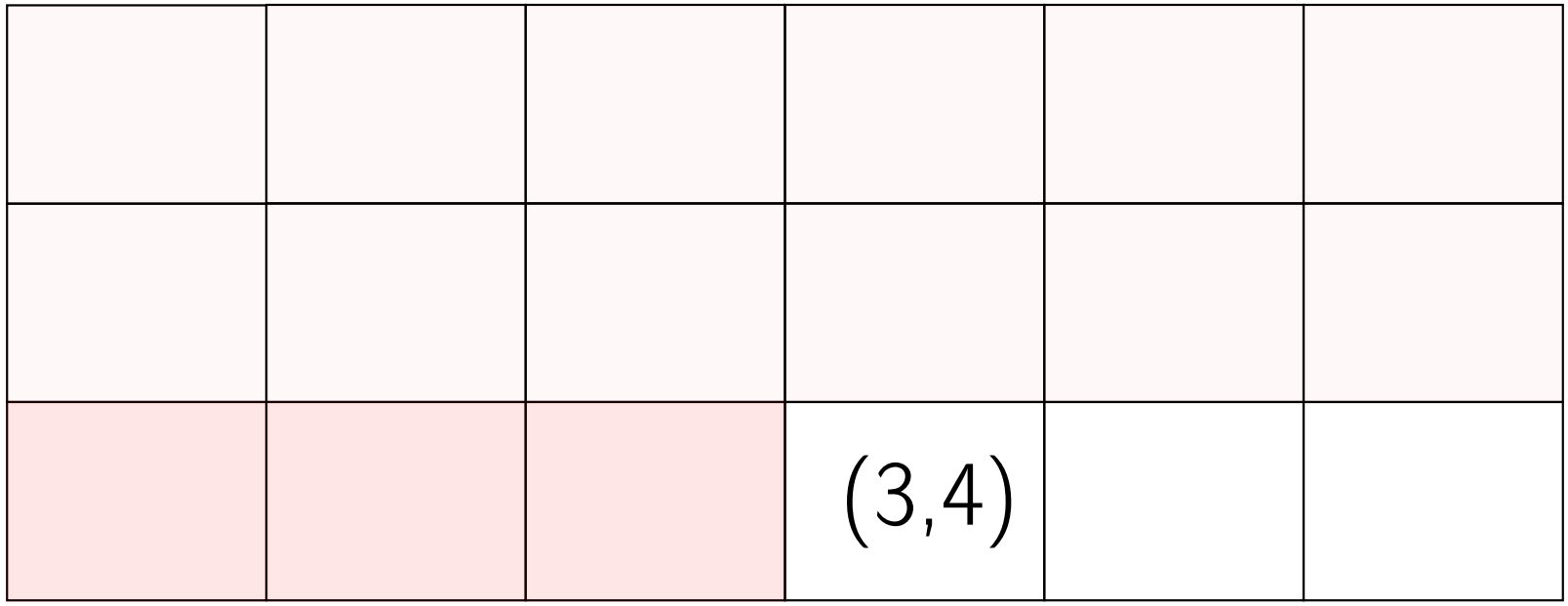
$$\text{array\_addr} + \text{elem\_size} \times (i - \text{first\_index})$$



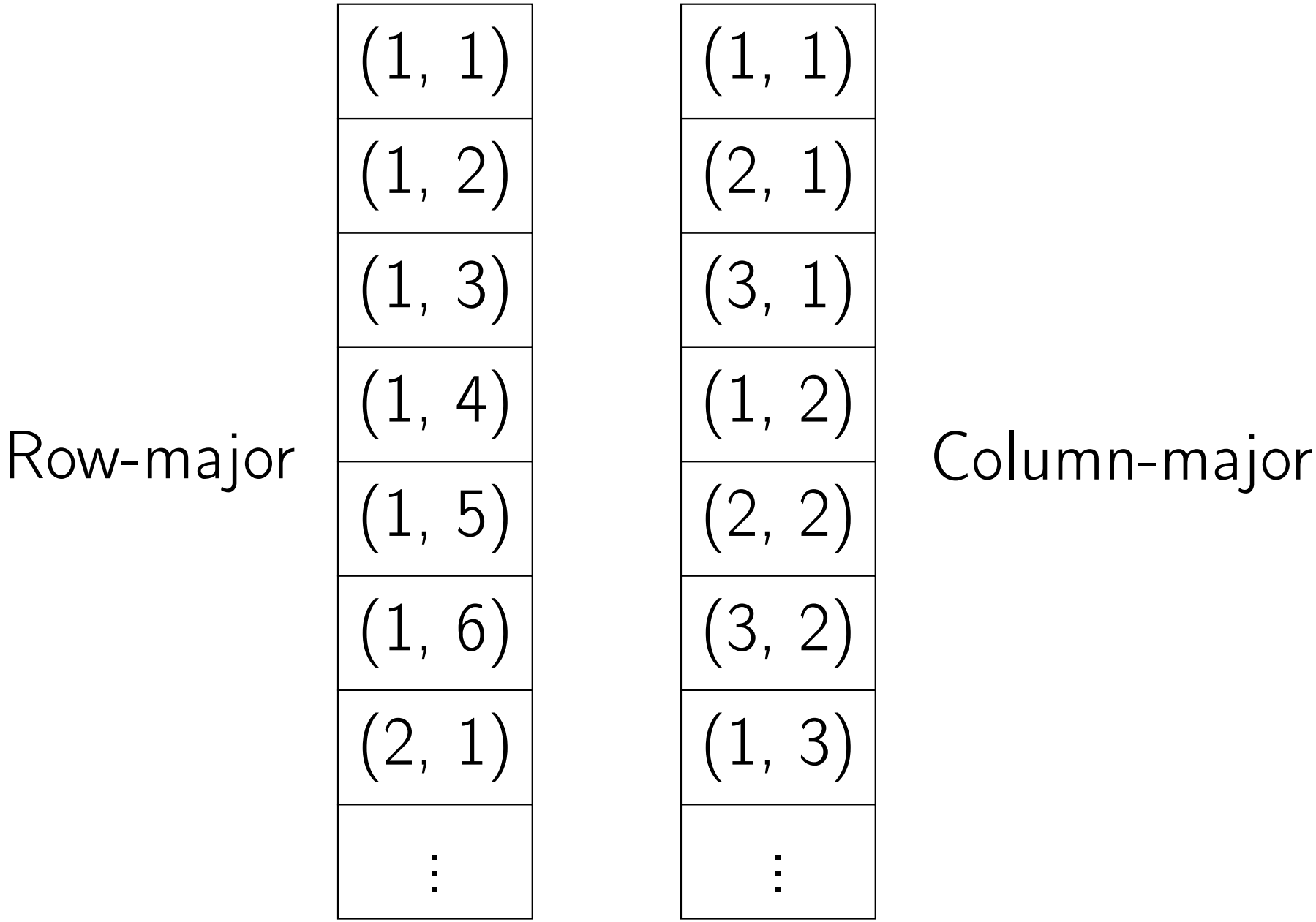
Multi-Dimensional Arrays

(1, 1)					

Multi-Dimensional Arrays

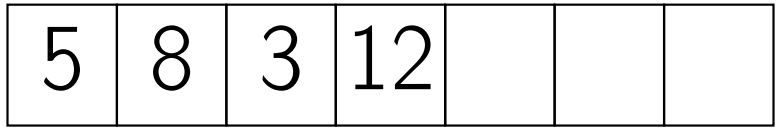


$$\text{array\_addr} +$$
$$\text{elem\_size} \times ((3-1) \times 6 + (4-1))$$


Row-major
(1, 1)
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(2, 1)
⋮
(1, 1)
(2, 1)
(3, 1)
(1, 2)
(2, 2)
(3, 2)
(1, 3)
⋮
Column-major

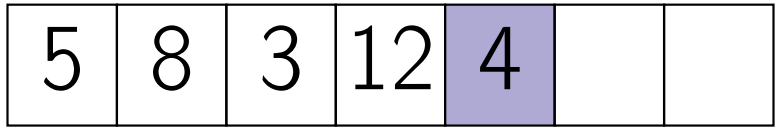
Times for Common Operations

	Add	Remove
Beginning		
End		
Middle		



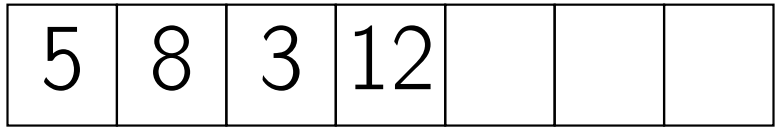
Times for Common Operations

	Add	Remove
Beginning		
End	$O(1)$	
Middle		



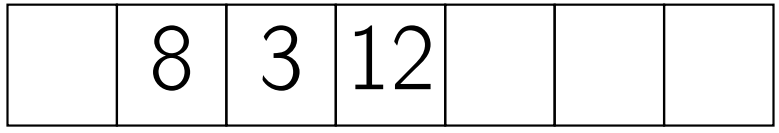
Times for Common Operations

	Add	Remove
Beginning		
End	$O(1)$	$O(1)$
Middle		



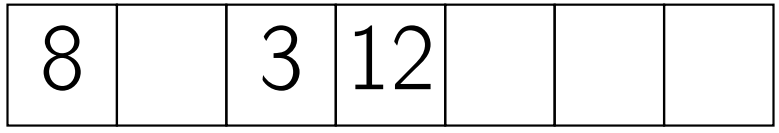
Times for Common Operations

	Add	Remove
Beginning		$O(n)$
End	$O(1)$	$O(1)$
Middle		



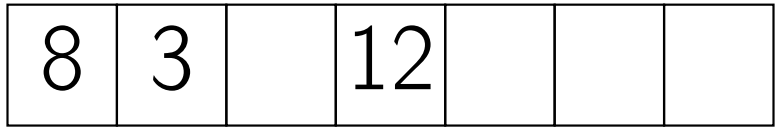
Times for Common Operations

	Add	Remove
Beginning		$O(n)$
End	$O(1)$	$O(1)$
Middle		



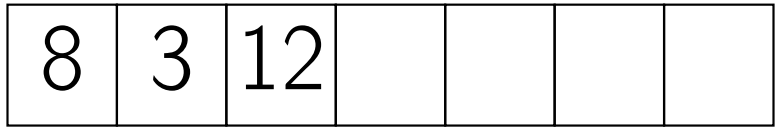
Times for Common Operations

	Add	Remove
Beginning		$O(n)$
End	$O(1)$	$O(1)$
Middle		



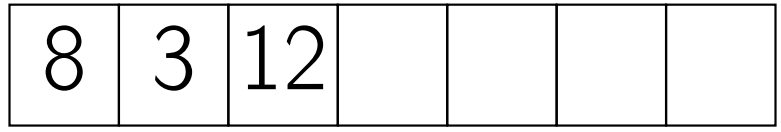
Times for Common Operations

	Add	Remove
Beginning		$O(n)$
End	$O(1)$	$O(1)$
Middle		



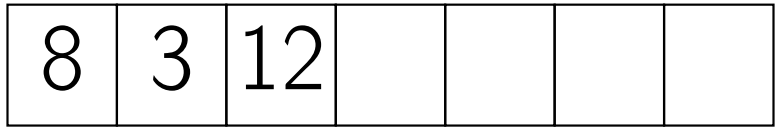
Times for Common Operations

	Add	Remove
Beginning	$O(n)$	$O(n)$
End	$O(1)$	$O(1)$
Middle		



Times for Common Operations

	Add	Remove
Beginning	$O(n)$	$O(n)$
End	$O(1)$	$O(1)$
Middle	$O(n)$	$O(n)$



Summary

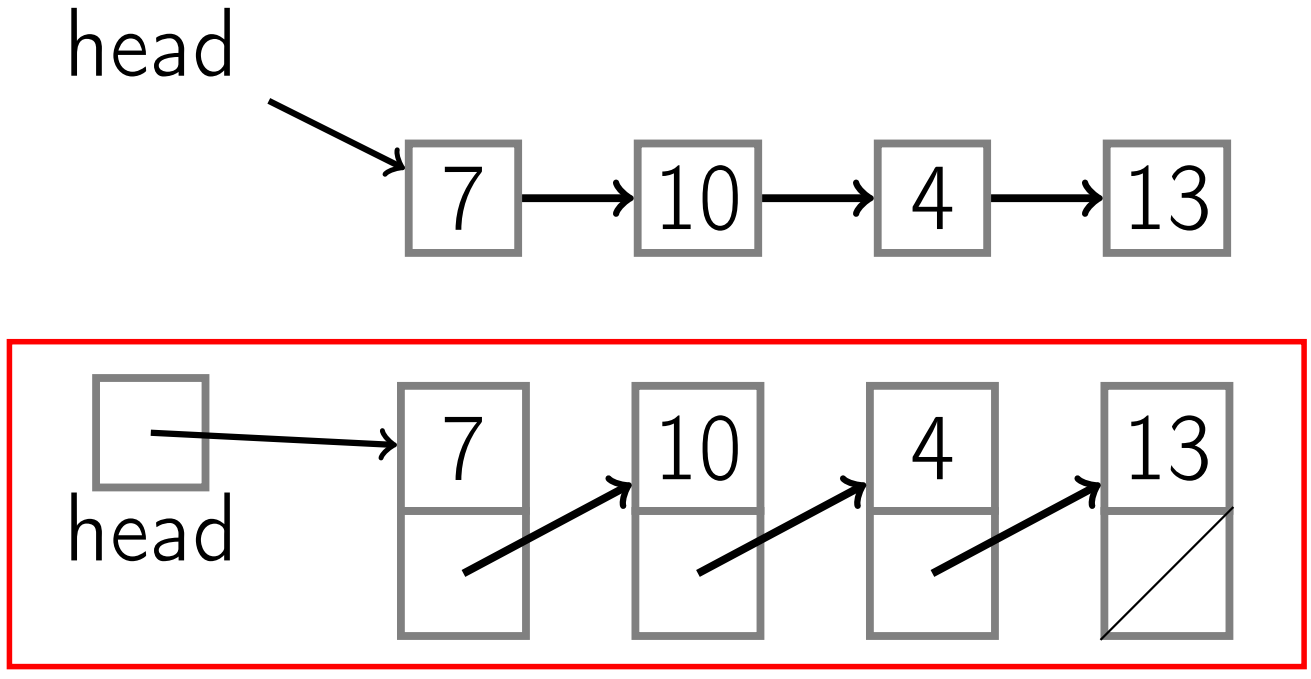
- Array: contiguous area of memory consisting of equal-size elements indexed by contiguous integers.
- Constant-time access to any element.
- Constant time to add/remove at the end.
- Linear time to add/remove at an arbitrary location.

Outline

1 Arrays

2 Linked Lists

Singly-Linked List



Node contains:

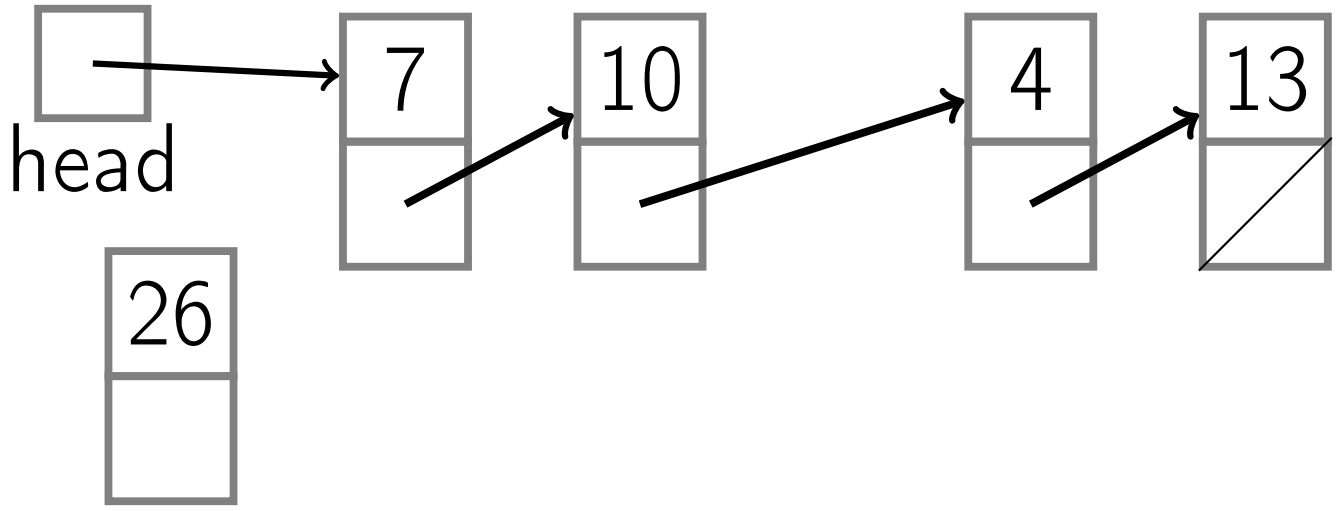
- `key`
- `next pointer`

List API

`PushFront(Key)`	add to front
`Key TopFront()`	return front item
`PopFront()`	remove front item
`PushBack(Key)`	add to back
`Key TopBack()`	return back item
`PopBack()`	remove back item
`Boolean Find(Key)`	is key in list?
`Erase(Key)`	remove key from list
`Boolean Empty()`	empty list?
`AddBefore(Node, Key)`	adds key before node
`AddAfter(Node, Key)`	adds key after node

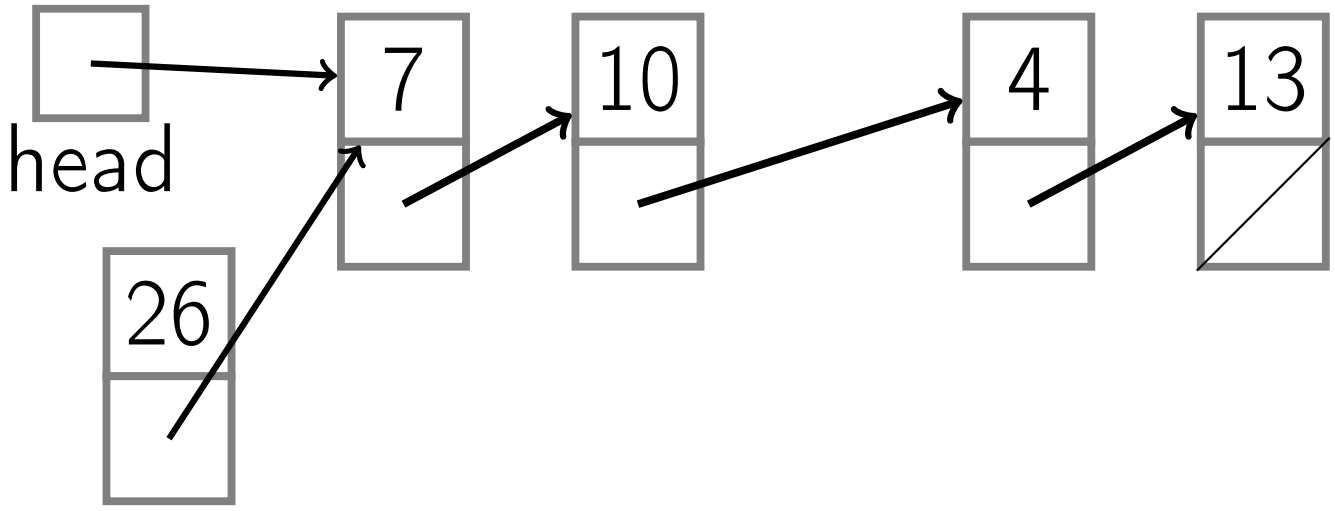
Times for Some Operations

`PushFront`



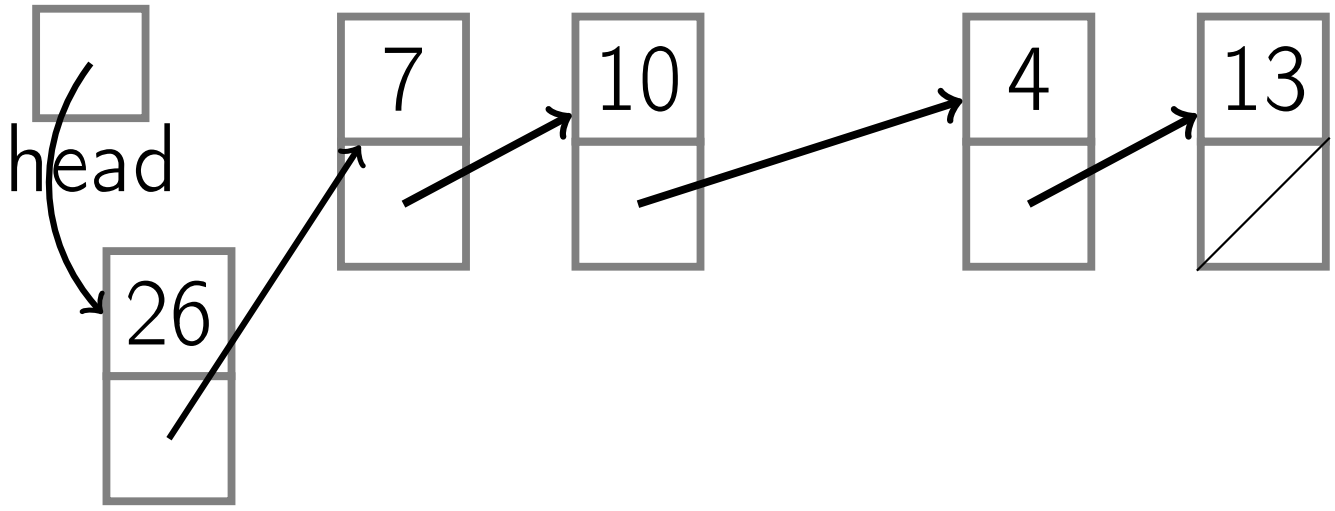
Times for Some Operations

`PushFront`



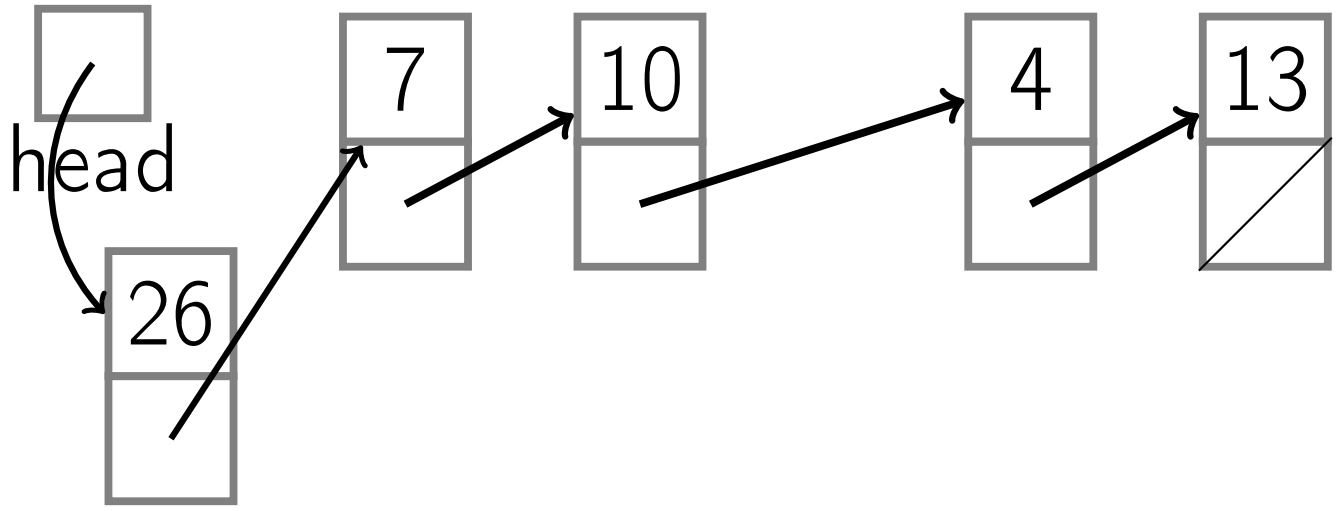
Times for Some Operations

`PushFront` $O(1)$



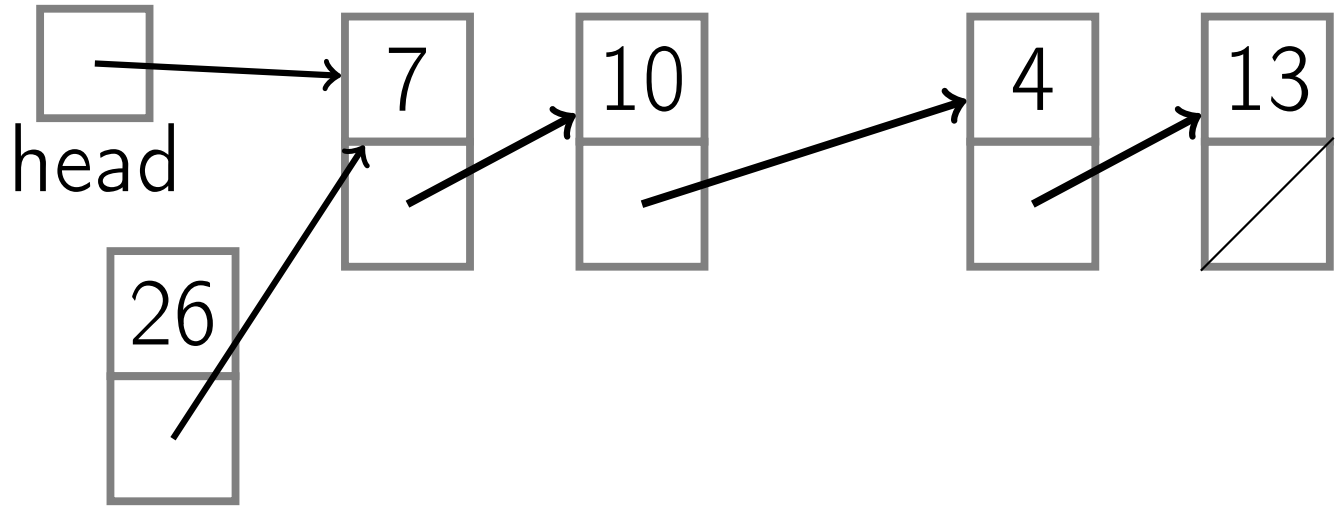
Times for Some Operations

`PopFront`



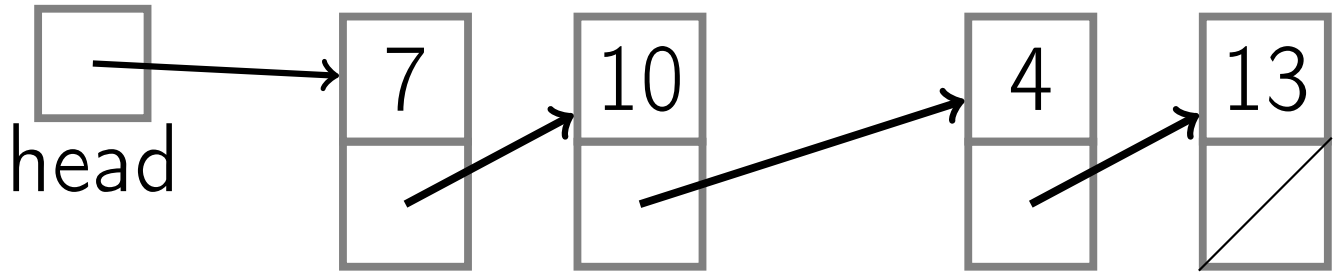
Times for Some Operations

`PopFront`



Times for Some Operations

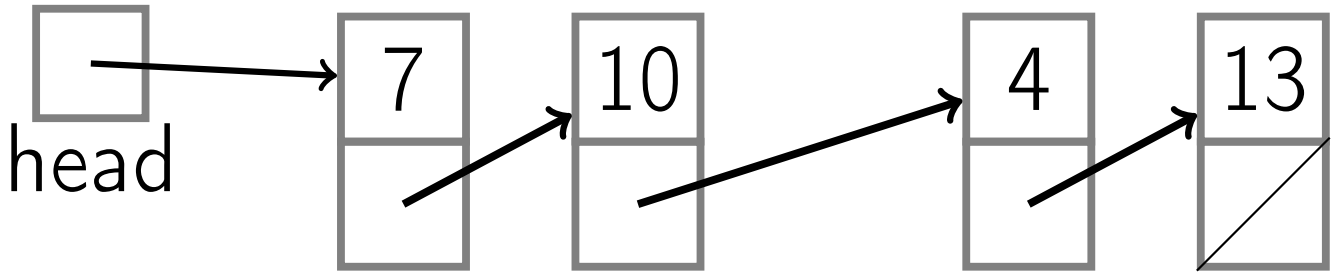
`PopFront` $O(1)$



Times for Some Operations

`PushBack` $O(n)$

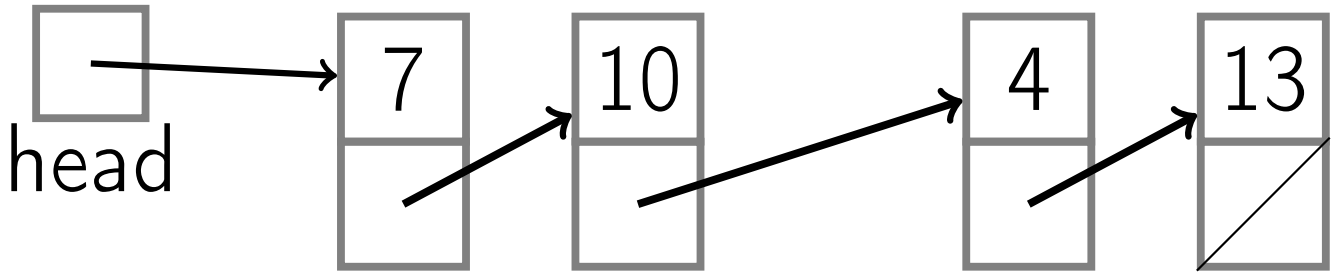
(no tail)



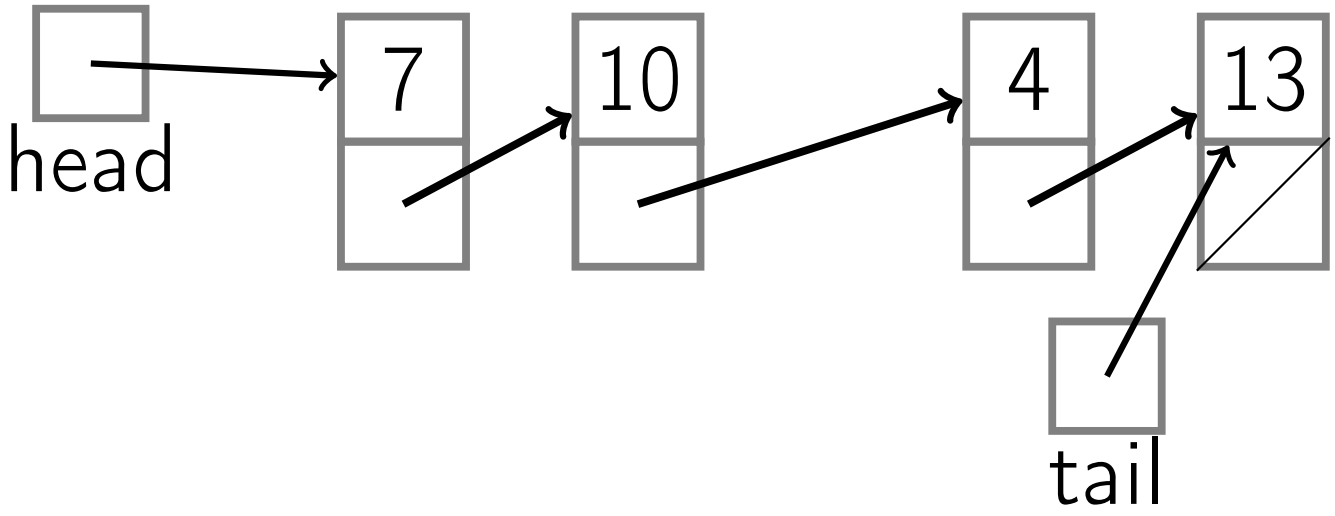
Times for Some Operations

`PopBack` $O(n)$

(no tail)

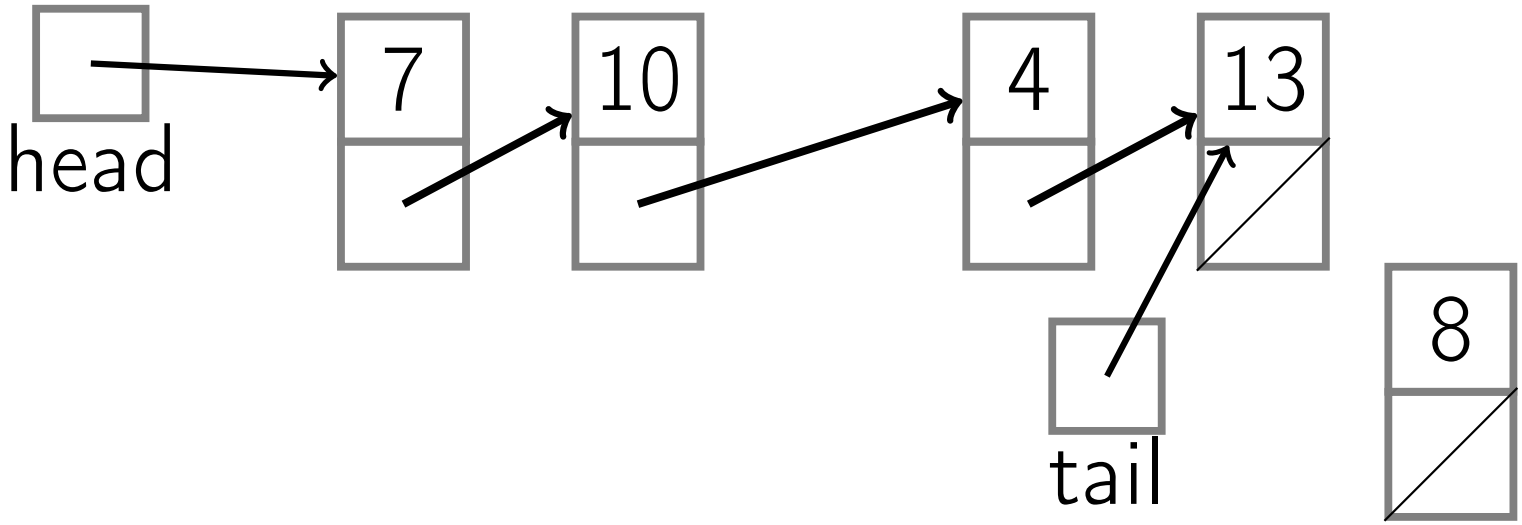


Times for Some Operations



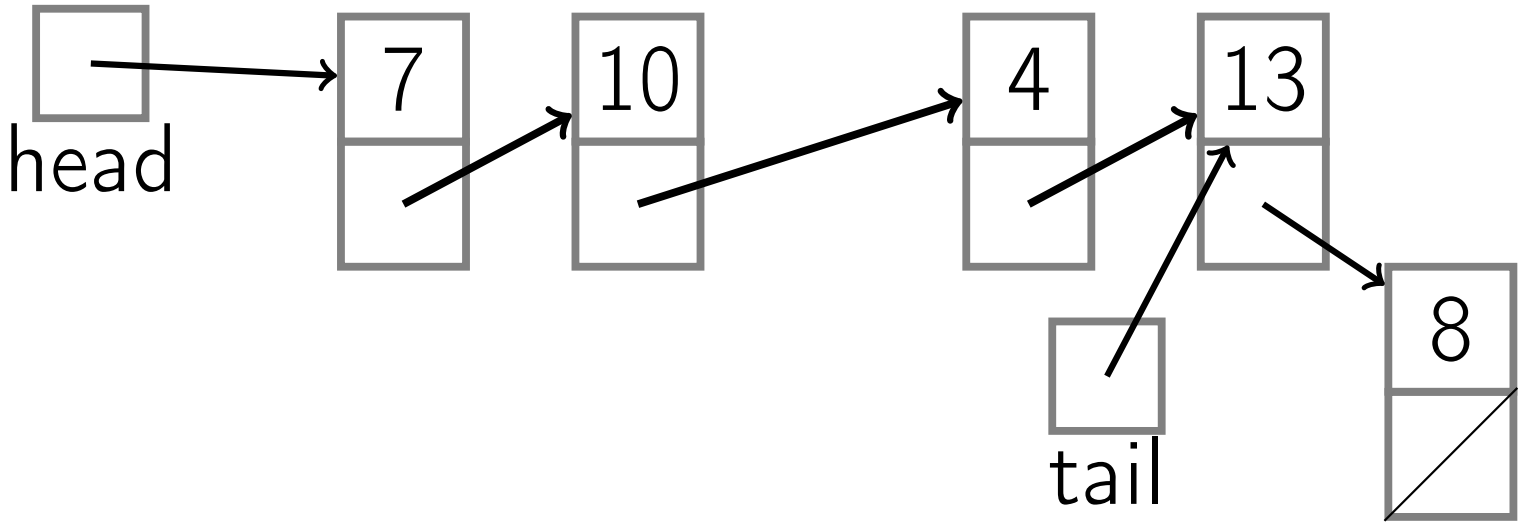
Times for Some Operations

`PushBack`
(with tail)



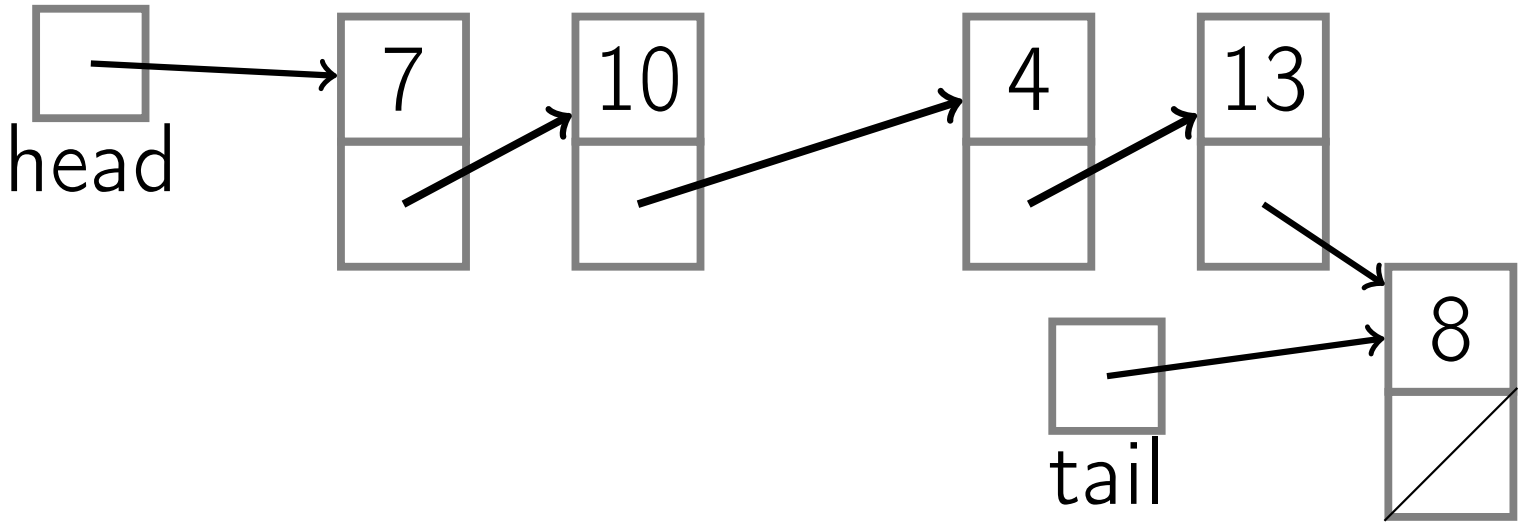
Times for Some Operations

`PushBack`
(with tail)



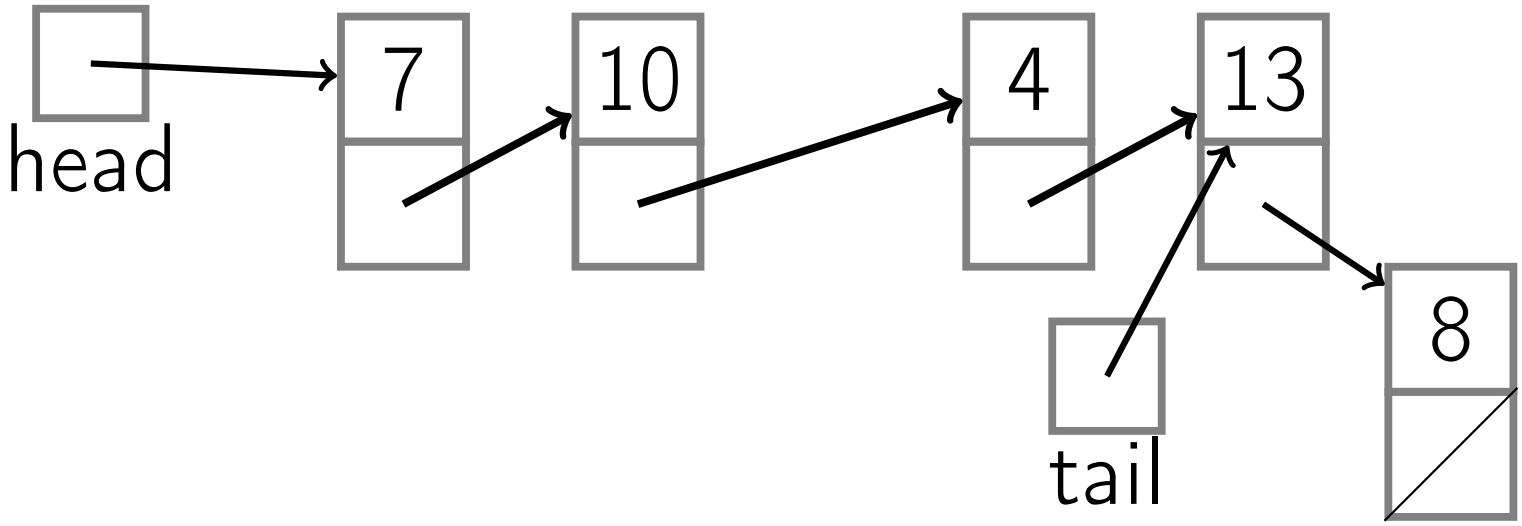
Times for Some Operations

`PushBack` $O(1)$
(with tail)



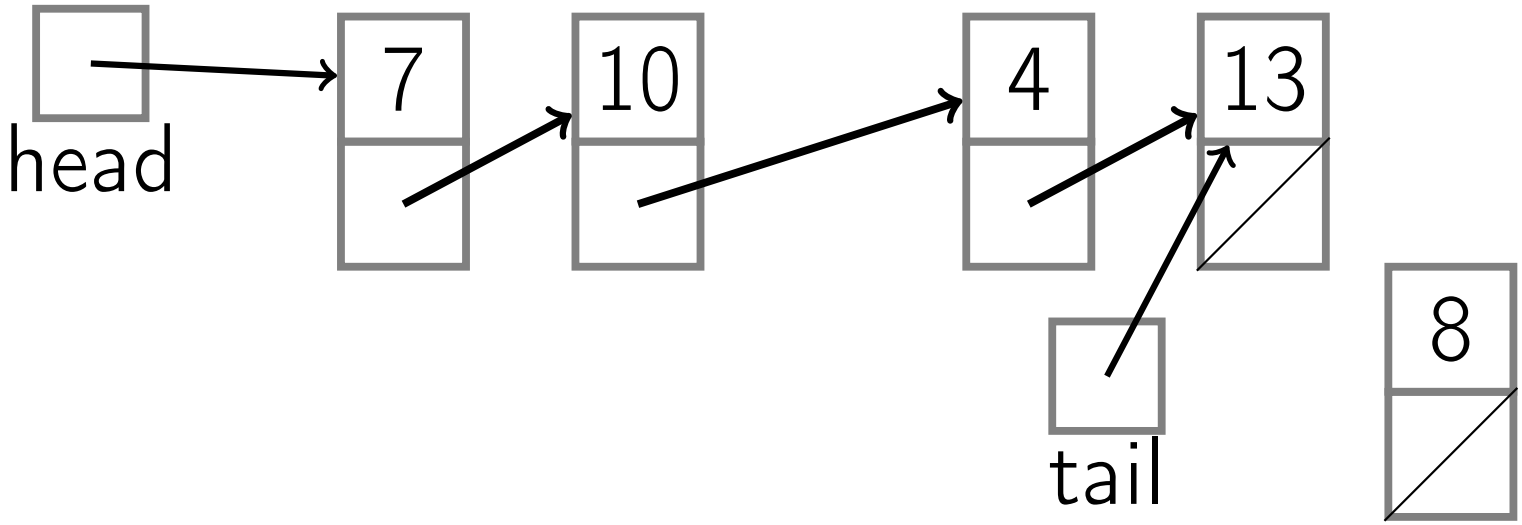
Times for Some Operations

PopBack
(with tail)



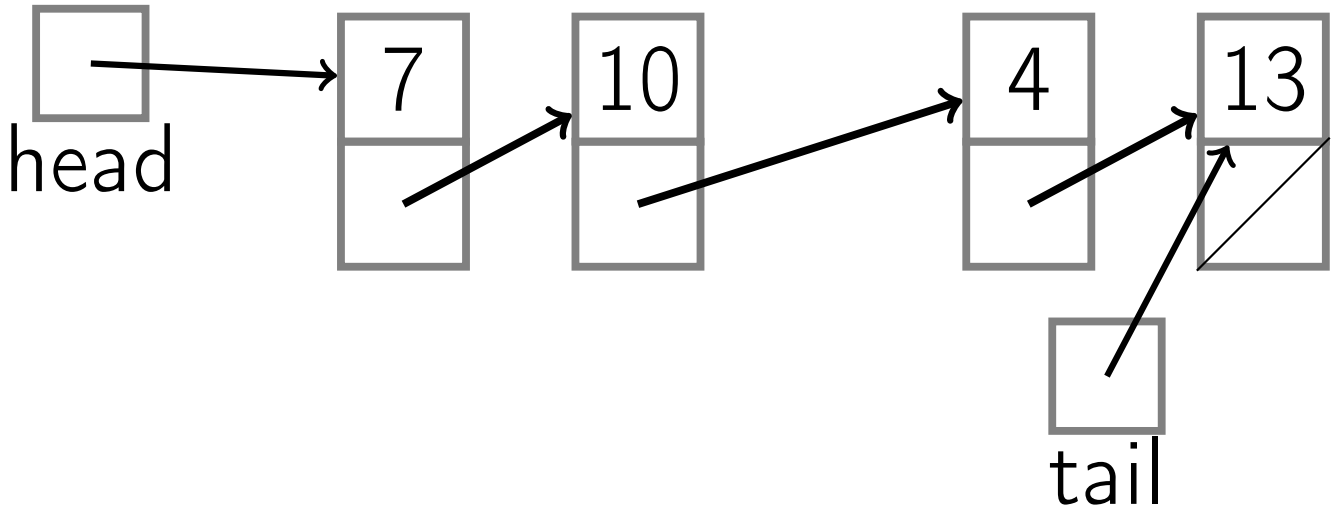
Times for Some Operations

PopBack
(with tail)



Times for Some Operations

`PopBack` $O(n)$
(with tail)



Singly-linked List

PushFront(*key*)

```
node ←new node
node.key ← key
node.next ← head
head ← node
if tail = nil:
    tail ← head
```

Singly-linked List

PopFront()

```
if head = nil:
  ERROR: empty list
head ← head.next
if head = nil:
  tail ← nil
```

Singly-linked List

PushBack(*key*)

```
node ←new node
node.key ← key
node.next =nil
if tail = nil:
    head ← tail ← node
else:
    tail.next ← node
    tail ← node
```

Singly-linked List

PopBack()

```
if head = nil:  ERROR: empty list
if head = tail:
   head ← tail ←nil
else:
   p ← head
   while p.next.next ≠ nil:
      p ← p.next
   p.next ← nil; tail ← p
```

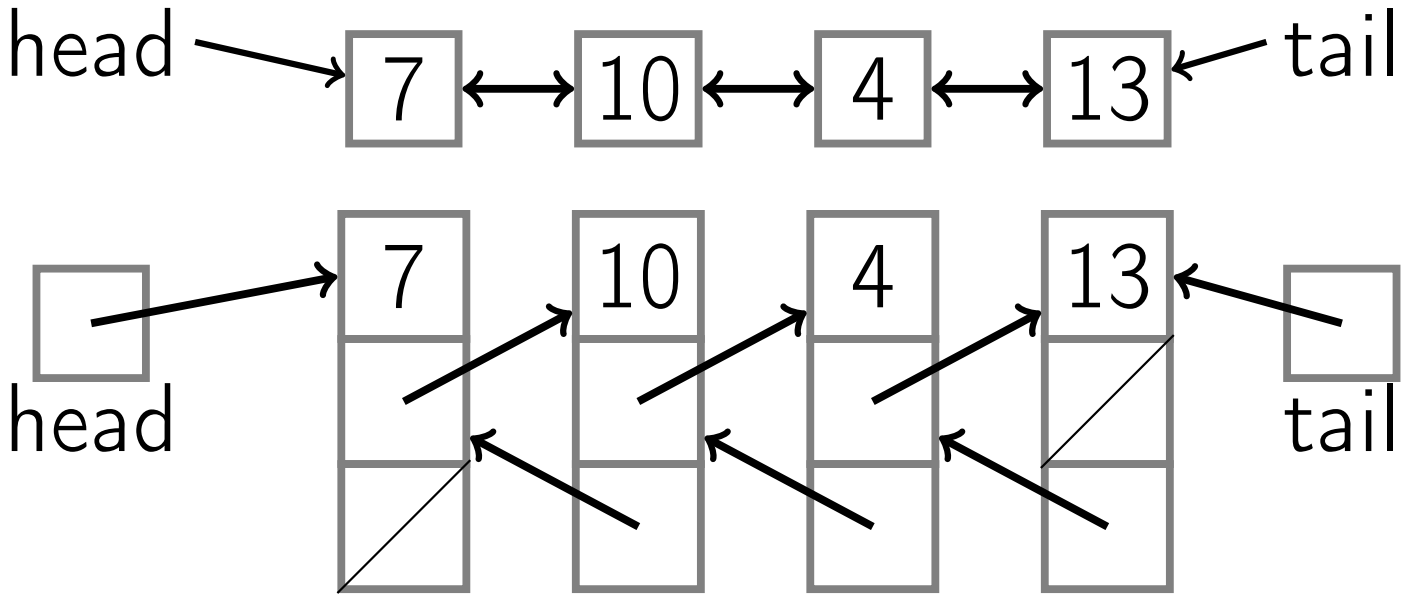
Singly-linked List

AddAfter(*node*, *key*)

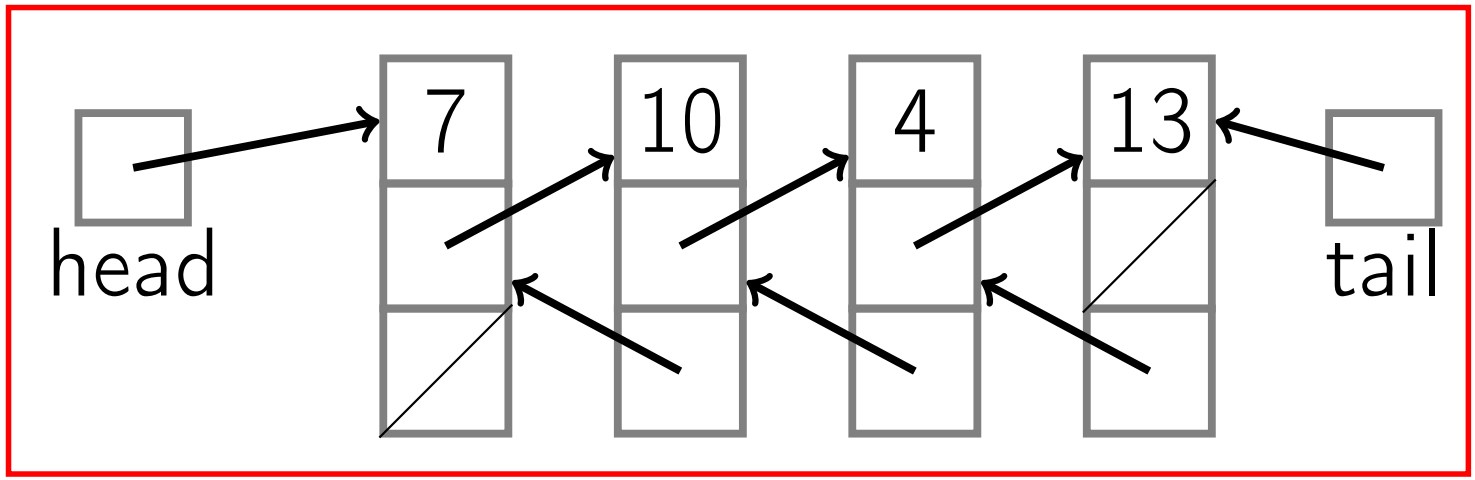
```
node2 ←new node
node2.key ← key
node2.next = node.next
node.next = node2
if tail = node:
    tail ← node2
```

Singly-Linked List	no tail	with tail
`PushFront(Key)`	$O(1)$	
`TopFront()`	$O(1)$	
`PopFront()`	$O(1)$	
`PushBack(Key)`	$O(n)$	$O(1)$
`TopBack()`	$O(n)$	$O(1)$
`PopBack()`	$O(n)$	
`Find(Key)`	$O(n)$	
`Erase(Key)`	$O(n)$	
`Empty()`	$O(1)$	
`AddBefore(Node, Key)`	$O(n)$	
`AddAfter(Node, Key)`	$O(1)$	

Doubly-Linked List



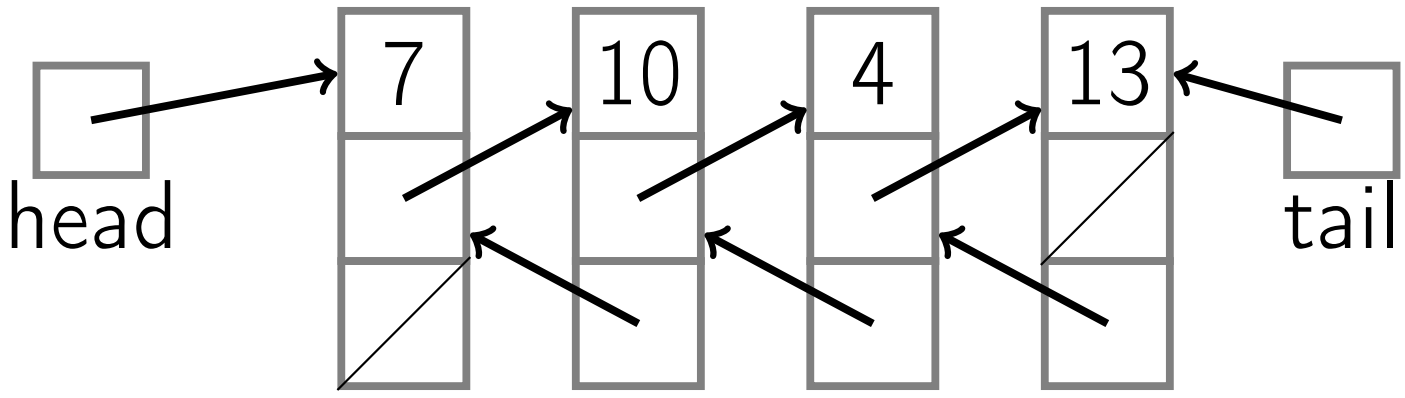
Doubly-Linked List



Node contains:

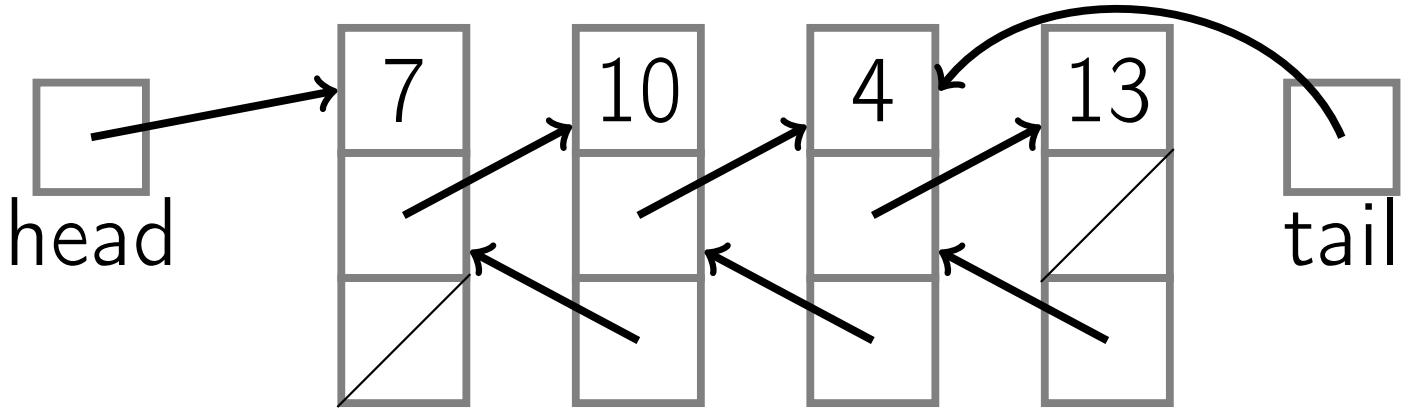
- `key`
- `next pointer`
- `prev pointer`

Doubly-Linked List



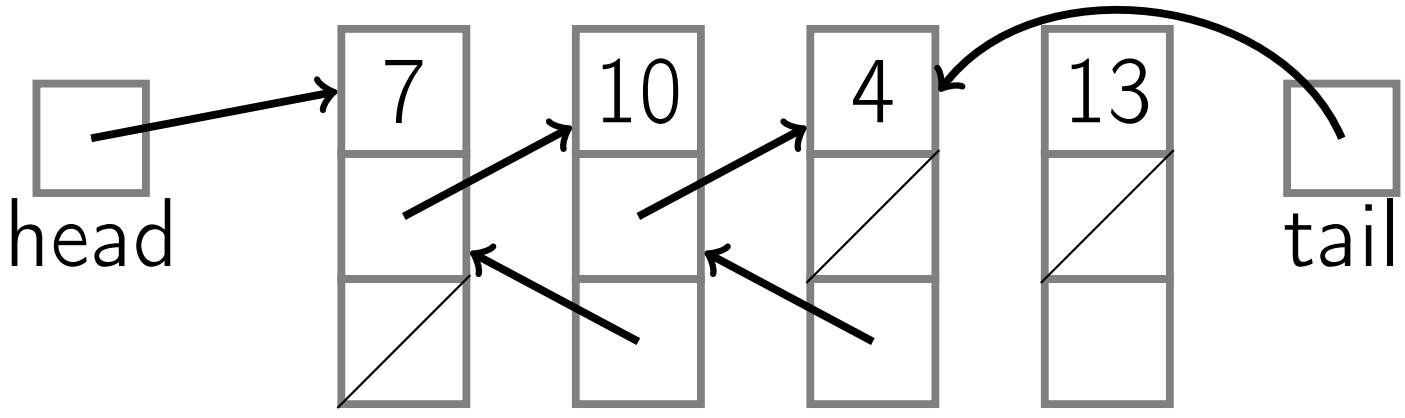
`PopBack`

Doubly-Linked List



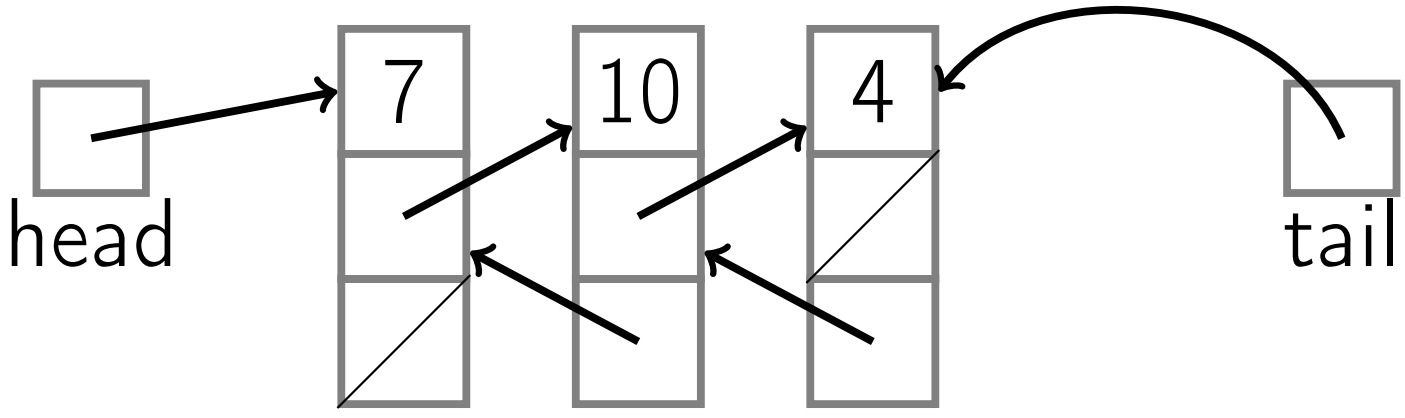
`PopBack`

Doubly-Linked List



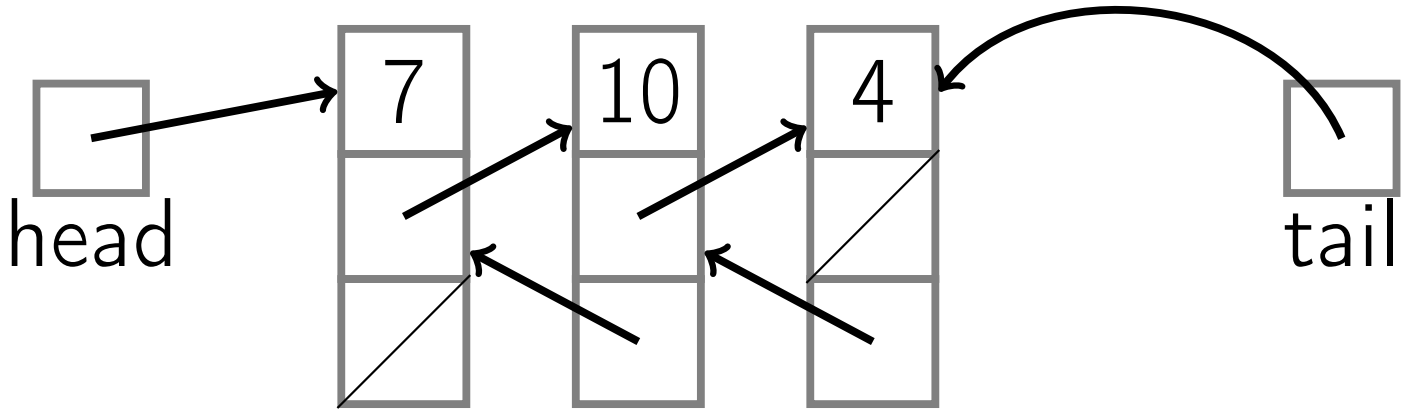
`PopBack`

Doubly-Linked List



`PopBack`

Doubly-Linked List



`PopBack` $O(1)$

Doubly-linked List

PushBack(*key*)

```
node ←new node
node.key ← key ;  node.next =nil
if  tail = nil:
    head ← tail ← node
    node.prev ←nil
else:
    tail.next ← node
    node.prev ← tail
    tail ← node
```

Doubly-linked List

PopBack()

```
if head = nil:  ERROR: empty list
if head = tail:
   head ← tail ←nil
else:
   tail ← tail.prev
   tail.next ←nil
```

Doubly-linked List

AddAfter(*node*, *key*)

```
node2 ←new node
node2.key ← key
node2.next ← node.next
node2.prev ← node
node.next ← node2
if node2.next ≠nil:
    node2.next.prev ← node2
if tail = node:
    tail ← node2
```

Doubly-linked List

AddBefore(*node*, *key*)

```
node2 ←new node
node2.key ← key
node2.next ← node
node2.prev ← node.prev
node.next ← node2
if node2.next ≠nil:
    node2.prev.next ← node2
if head = node:
    head ← node2
```

Doubly-Linked List	no tail	with tail
`PushFront(Key)`	$O(1)$	
`TopFront()`	$O(1)$	
`PopFront()`	$O(1)$	
`PushBack(Key)`	$O(n)$	$O(1)$
`TopBack()`	$O(n)$	$O(1)$
`PopBack()`	~~$O(n)$~~ $O(1)$	
`Find(Key)`	$O(n)$	
`Erase(Key)`	$O(n)$	
`Empty()`	$O(1)$	
`AddBefore(Node, Key)`	~~$O(n)$~~ $O(1)$	
`AddAfter(Node, Key)`	$O(1)$	

Summary

- Constant time to insert at or remove from the front.
- With tail and doubly-linked, constant time to insert at or remove from the back.
- $O(n)$ time to find arbitrary element.
- List elements need not be contiguous.
- With doubly-linked list, constant time to insert between nodes or remove a node.

Basic Data Structures: Stacks and Queues

Neil Rhodes

Department of Computer Science and Engineering
University of California, San Diego

Data Structures
Data Structures and Algorithms

Outline

1 Stacks

2 Queues

Definition

Stack: Abstract data type with the following operations:

- `Push(Key)`: adds key to collection
- `Key Top()`: returns most recently-added key
- `Key Pop()`: removes and returns most recently-added key
- `Boolean Empty()`: are there any elements?

Balanced Brackets

Input: A string ***str*** consisting of '(', ')', '[', ']' characters.

Output: Return whether or not the string's parentheses and square brackets are balanced.

Balanced Brackets

Balanced:

- “([])[]()”,
- “((([([])]))())”

Unbalanced:

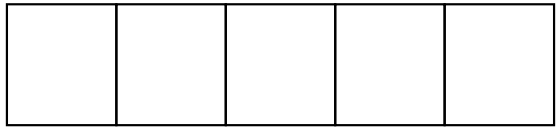
- “([]()”
- “][”

IsBalanced(*str*)

```
Stack stack
for char in str:
  if char in [‘(‘, ‘[‘]:
    stack.Push(char)
  else:
    if stack.Empty():  return False
    top ← stack.Pop()
    if (top = ‘[‘ and char != ‘]’) or
      (top = ‘(‘ and char != ‘)’):
      return False
return stack.Empty()
```

Stack Implementation with Array

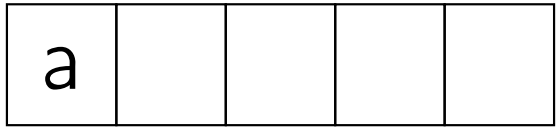
numElements: 0



`Push(a)`

Stack Implementation with Array

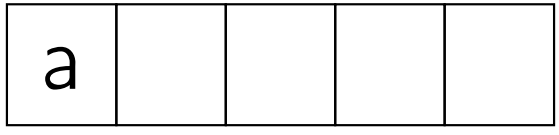
numElements: 1



`Push(a)`

Stack Implementation with Array

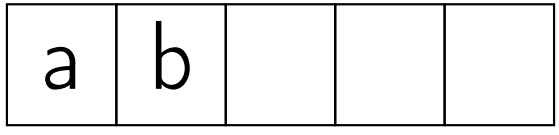
numElements: 1



`Push(b)`

Stack Implementation with Array

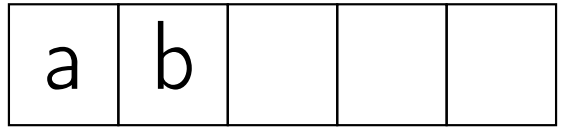
numElements: 2



`Push(b)`

Stack Implementation with Array

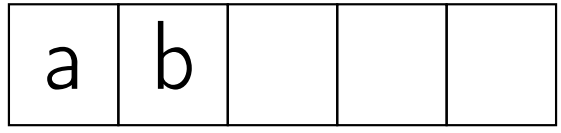
numElements: 2



`Top()`

Stack Implementation with Array

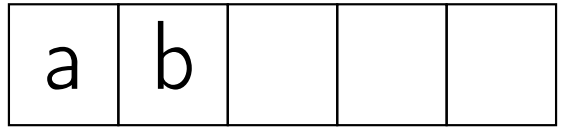
numElements: 2



`Top()` $\rightarrow$ `b`

Stack Implementation with Array

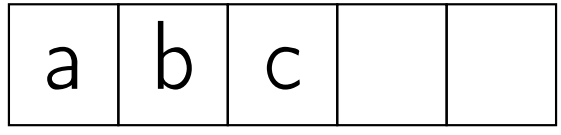
numElements: 2



`Push(c)`

Stack Implementation with Array

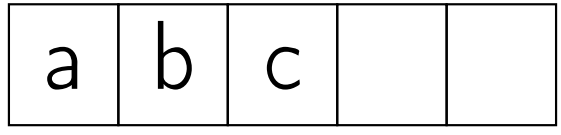
numElements: 3



`Push(c)`

Stack Implementation with Array

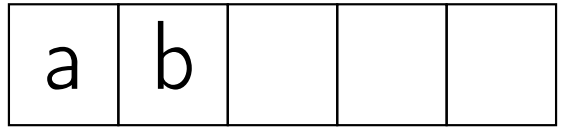
numElements: 3



`Pop()`

Stack Implementation with Array

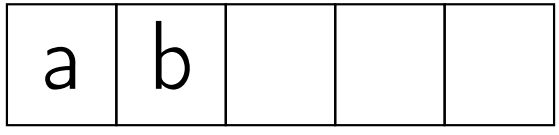
numElements: 2



`Pop()` $\rightarrow$ `c`

Stack Implementation with Array

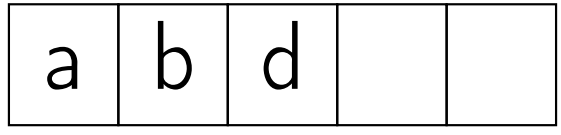
numElements: 2



`Push(d)`

Stack Implementation with Array

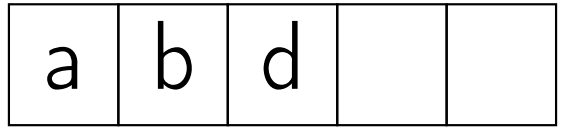
numElements: 3



`Push(d)`

Stack Implementation with Array

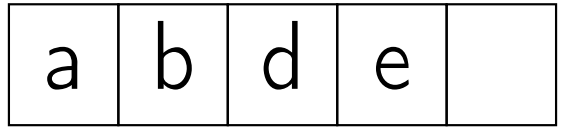
numElements: 3



`Push(e)`

Stack Implementation with Array

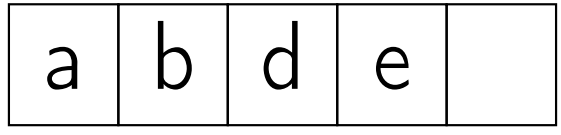
numElements: 4



`Push(e)`

Stack Implementation with Array

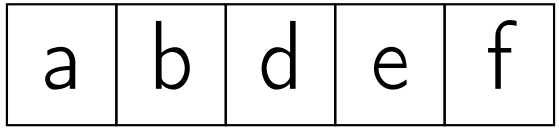
numElements: 4



`Push(f)`

Stack Implementation with Array

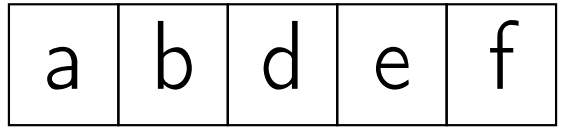
numElements: 5



`Push(f)`

Stack Implementation with Array

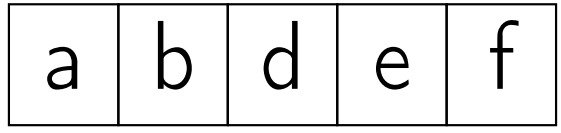
numElements: 5



`Push(g)`

Stack Implementation with Array

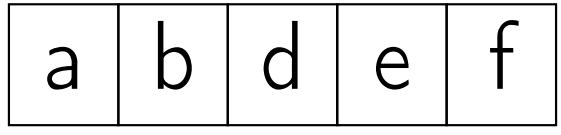
numElements: 5



`Push(g)` $\rightarrow$ `ERROR`

Stack Implementation with Array

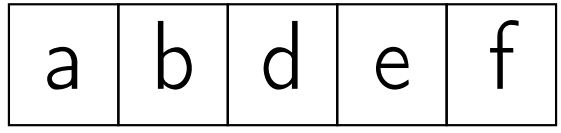
numElements: 5



`Empty()`

Stack Implementation with Array

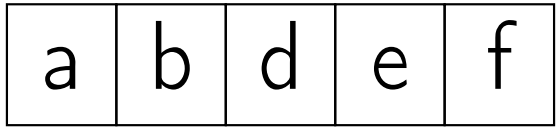
numElements: 5



`Empty()` $\rightarrow$ `False`

Stack Implementation with Array

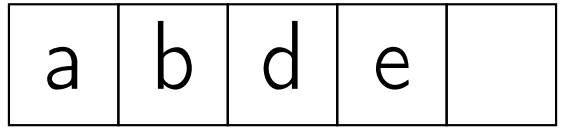
numElements: 5



`Pop()`

Stack Implementation with Array

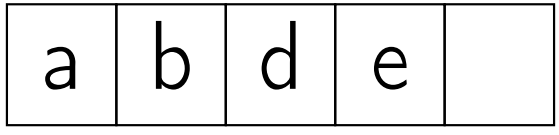
numElements: 4



`Pop()` $\rightarrow$ `f`

Stack Implementation with Array

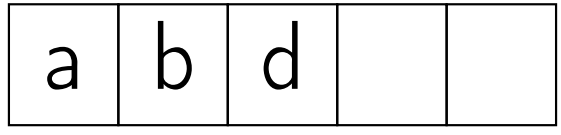
numElements: 4



`Pop()`

Stack Implementation with Array

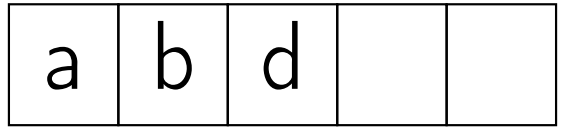
numElements: 3



`Pop()` $\rightarrow$ `e`

Stack Implementation with Array

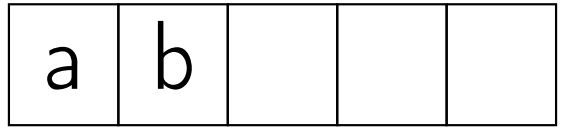
numElements: 3



`Pop()`

Stack Implementation with Array

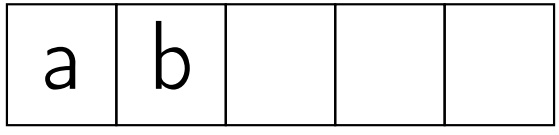
numElements: 2



`Pop()` $\rightarrow$ `d`

Stack Implementation with Array

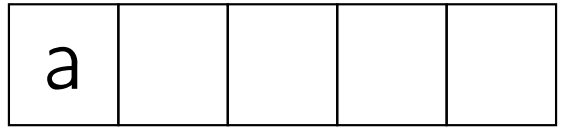
numElements: 2



`Pop()`

Stack Implementation with Array

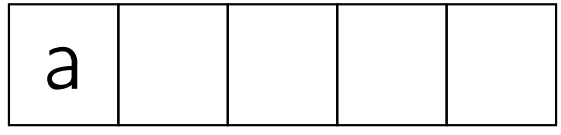
numElements: 1



`Pop()` $\rightarrow$ `b`

Stack Implementation with Array

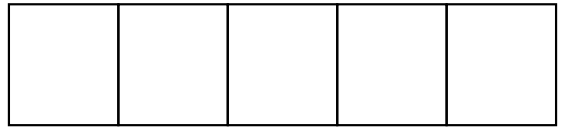
numElements: 1



`Pop()`

Stack Implementation with Array

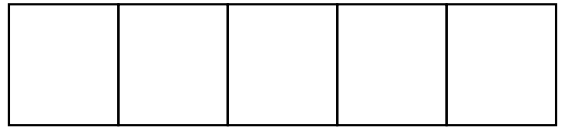
numElements: 0



`Pop()` $\rightarrow$ `a`

Stack Implementation with Array

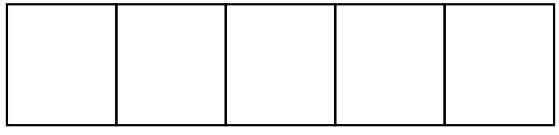
numElements: 0



`Empty()`

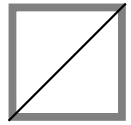
Stack Implementation with Array

numElements: 0



`Empty()` $\rightarrow$ `True`

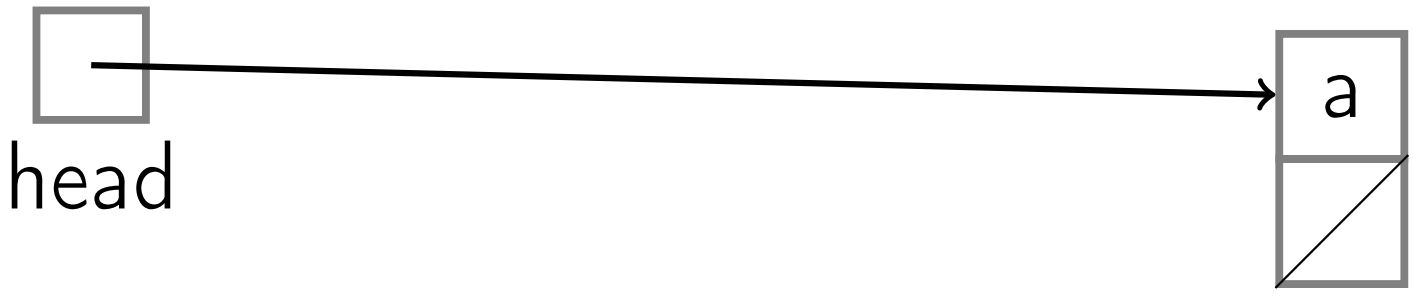
Stack Implementation with Linked List



head

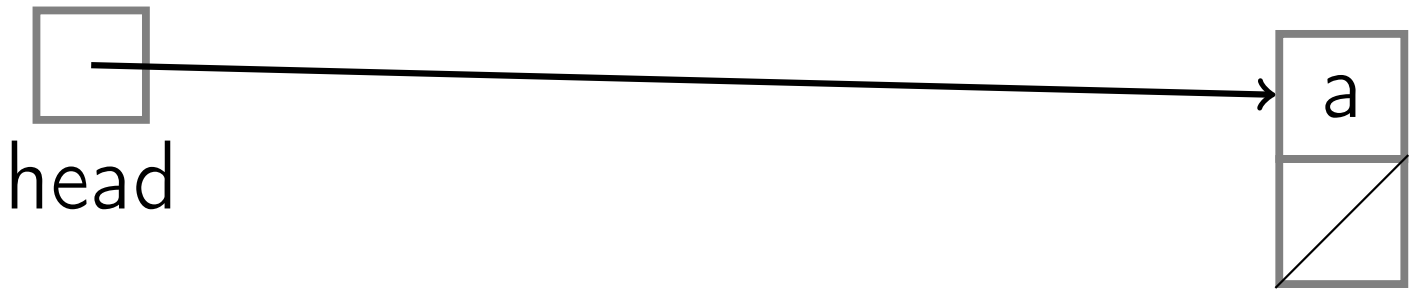
`Push(a)`

Stack Implementation with Linked List



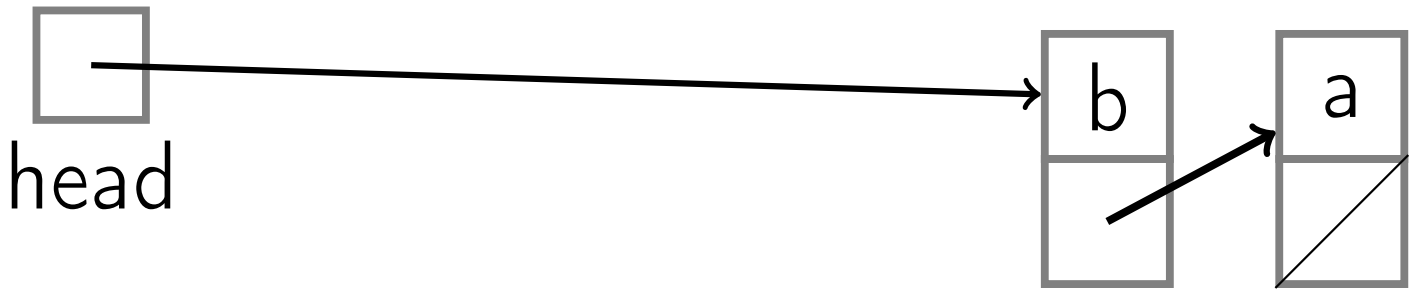
`Push(a)`

Stack Implementation with Linked List



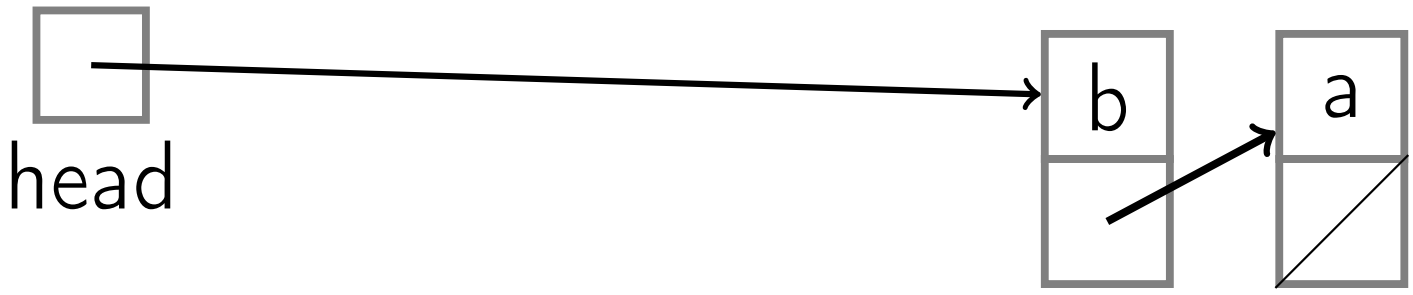
`Push(b)`

Stack Implementation with Linked List



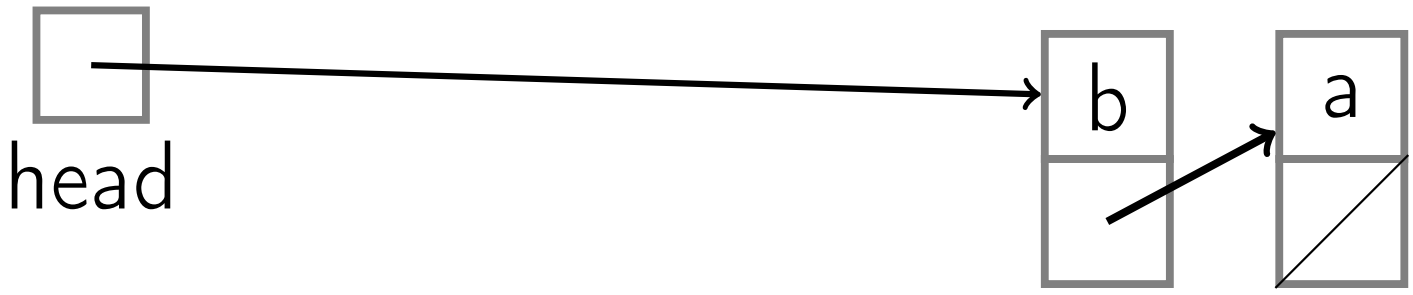
`Push(b)`

Stack Implementation with Linked List



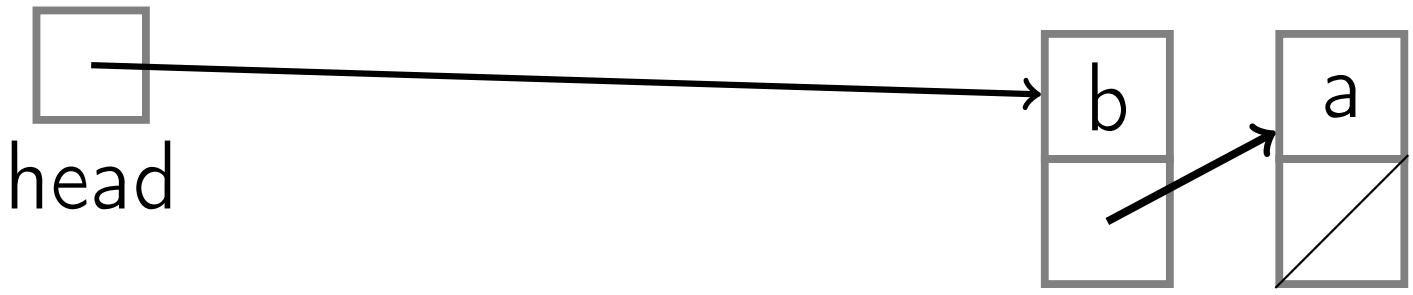
`Top()`

Stack Implementation with Linked List



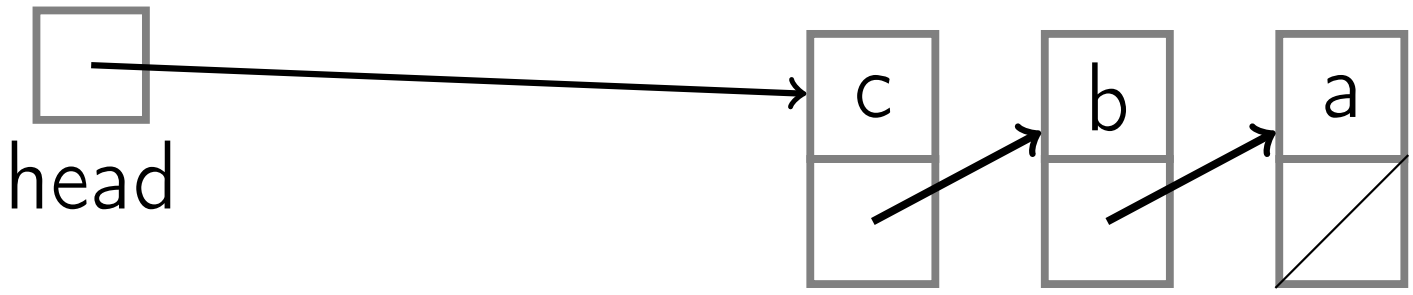
`Top()` $\rightarrow$ `b`

Stack Implementation with Linked List



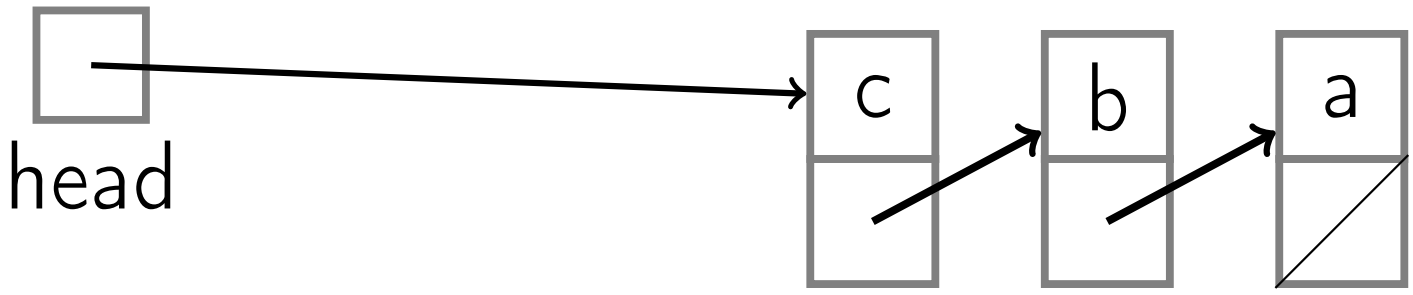
`Push(c)`

Stack Implementation with Linked List



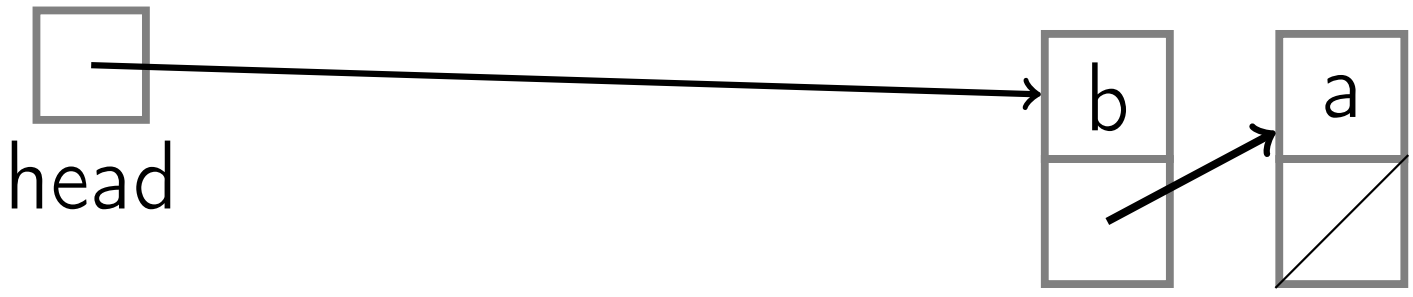
`Push(c)`

Stack Implementation with Linked List



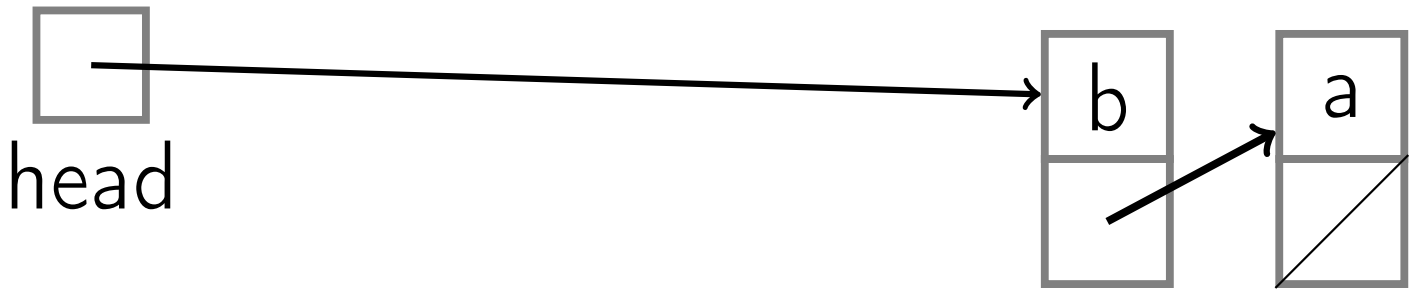
`Pop()`

Stack Implementation with Linked List



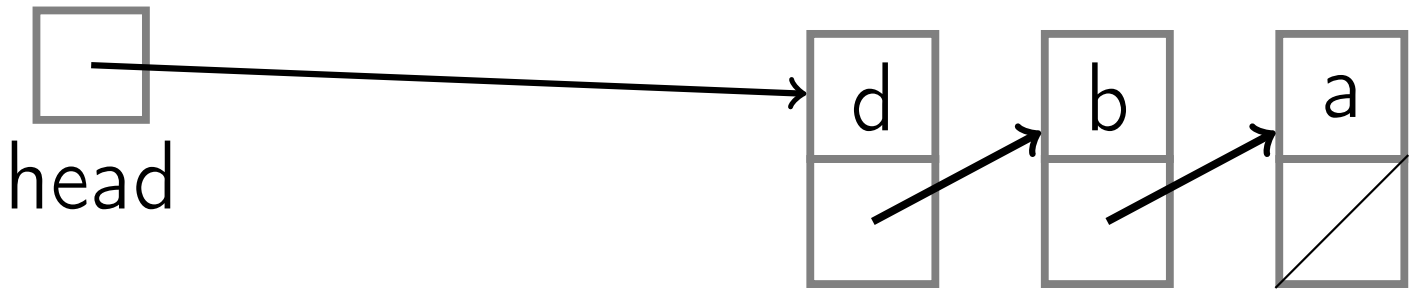
`Pop()` $\rightarrow$ `c`

Stack Implementation with Linked List



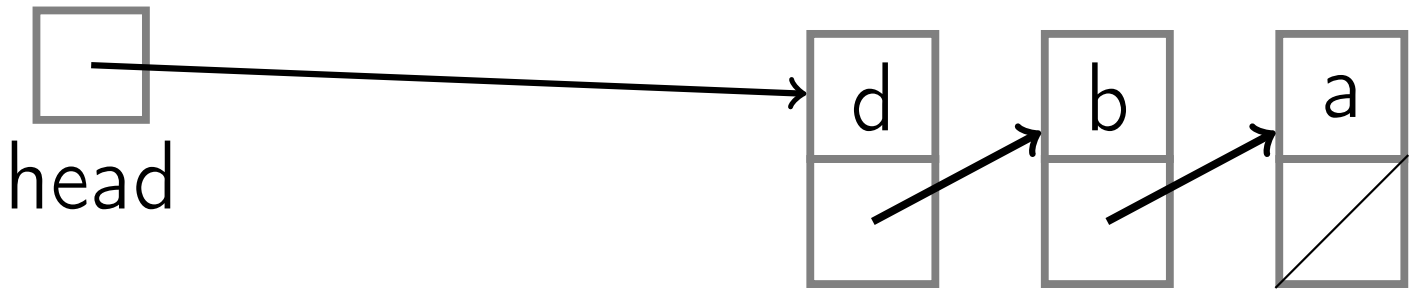
`Push(d)`

Stack Implementation with Linked List



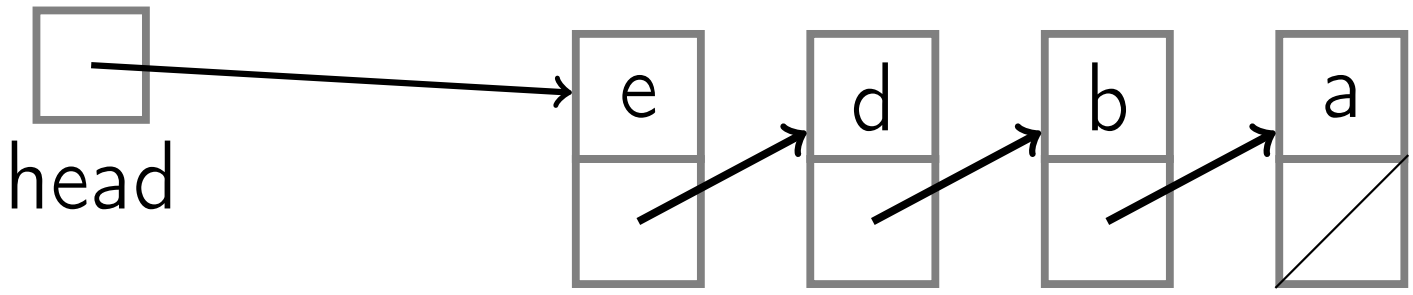
`Push(d)`

Stack Implementation with Linked List



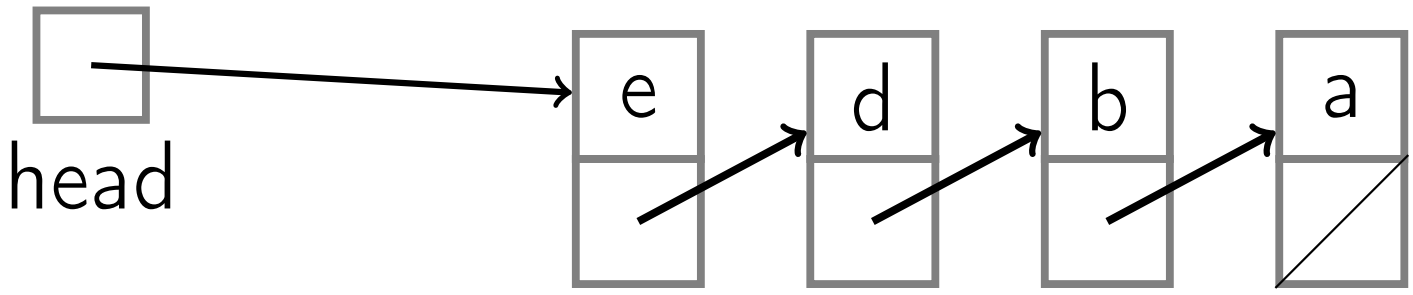
`Push(e)`

Stack Implementation with Linked List



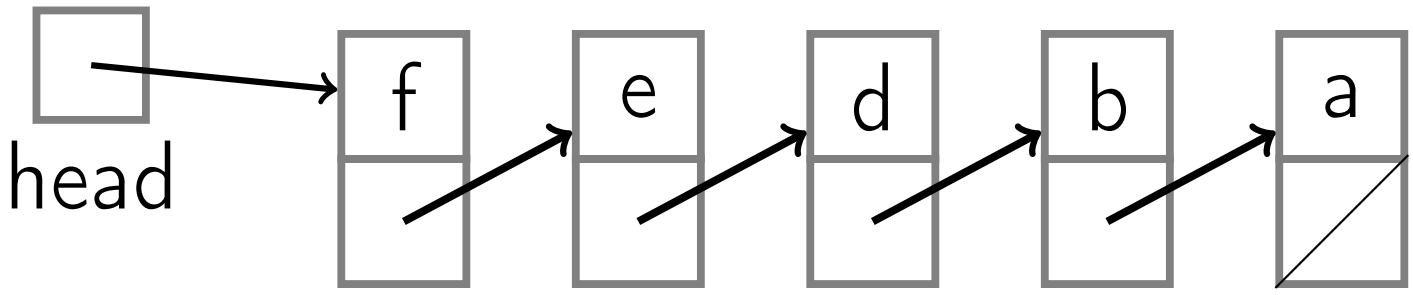
`Push(e)`

Stack Implementation with Linked List



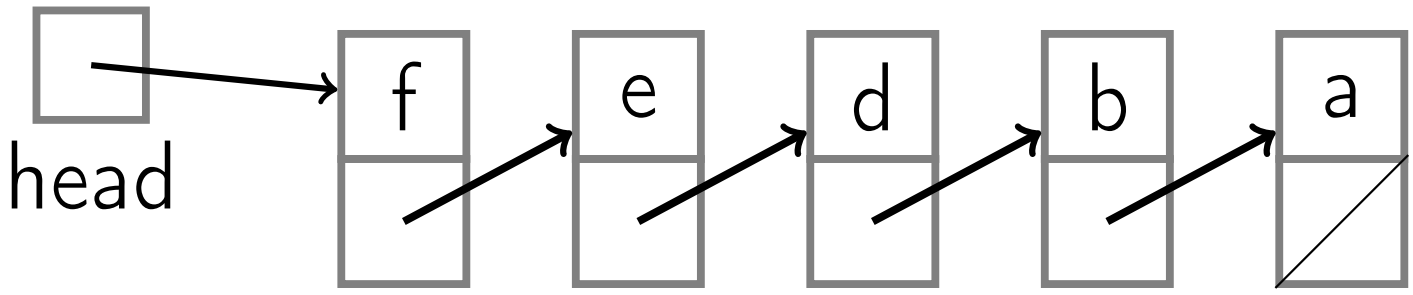
`Push(f)`

Stack Implementation with Linked List



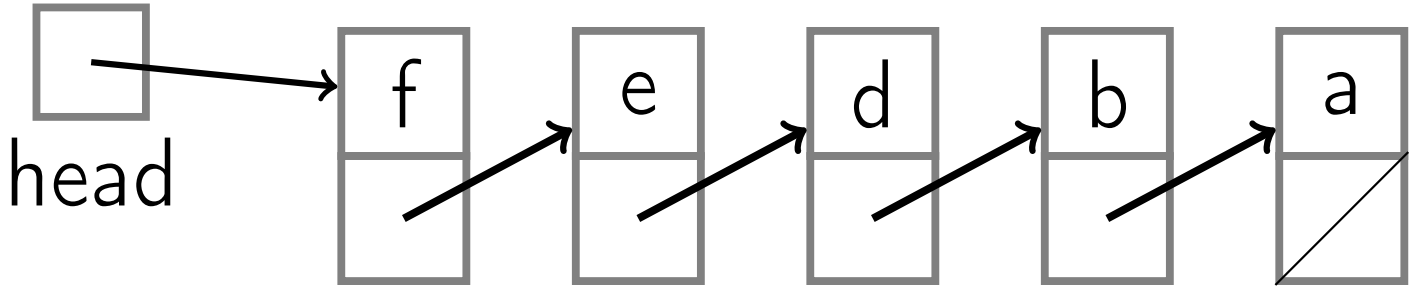
`Push(f)`

Stack Implementation with Linked List



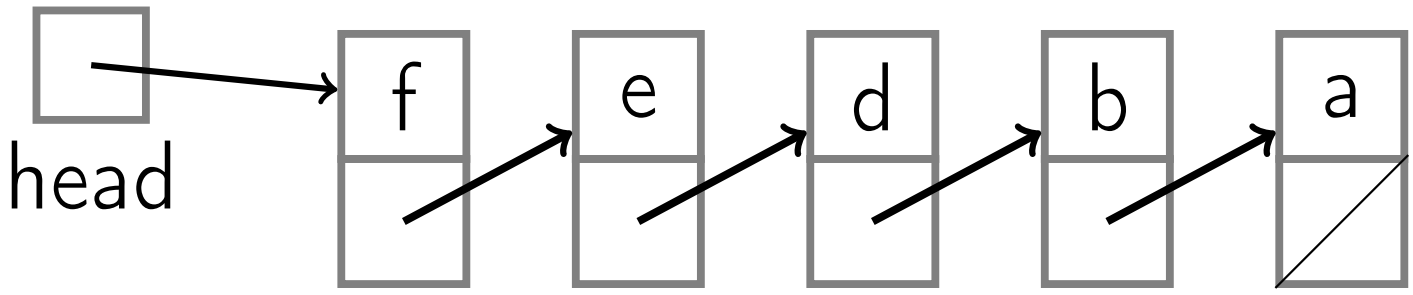
`Empty()`

Stack Implementation with Linked List



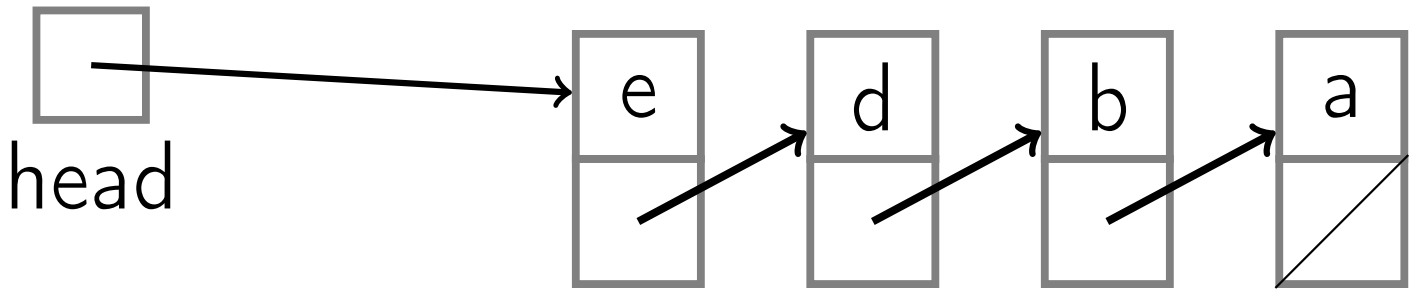
`Empty()` $\rightarrow$ `False`

Stack Implementation with Linked List



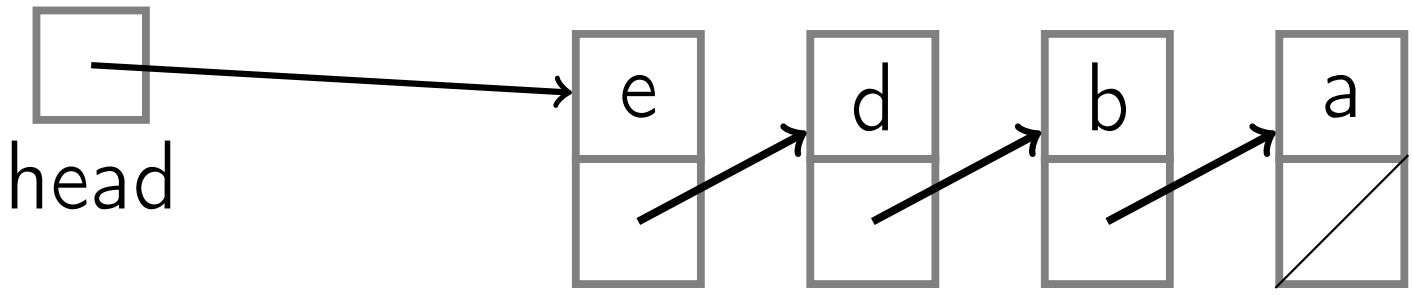
Pop()

Stack Implementation with Linked List



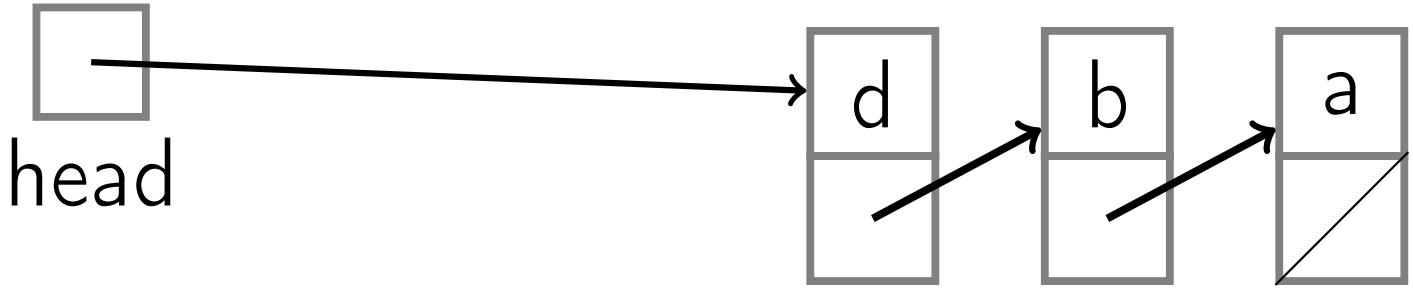
Pop()→ f

Stack Implementation with Linked List



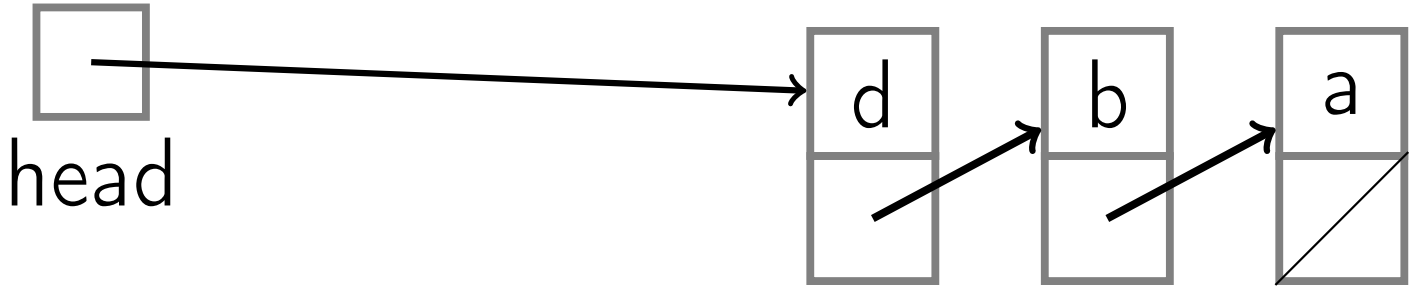
`Pop()`

Stack Implementation with Linked List



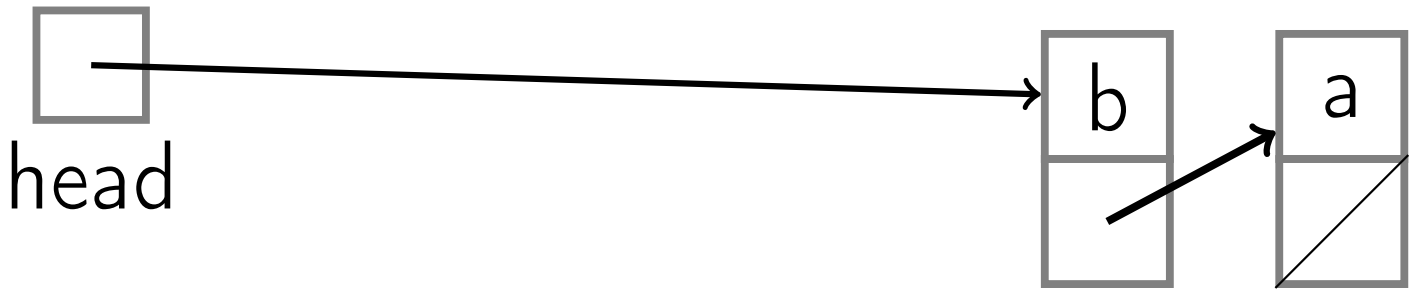
`Pop()` $\rightarrow$ `e`

Stack Implementation with Linked List



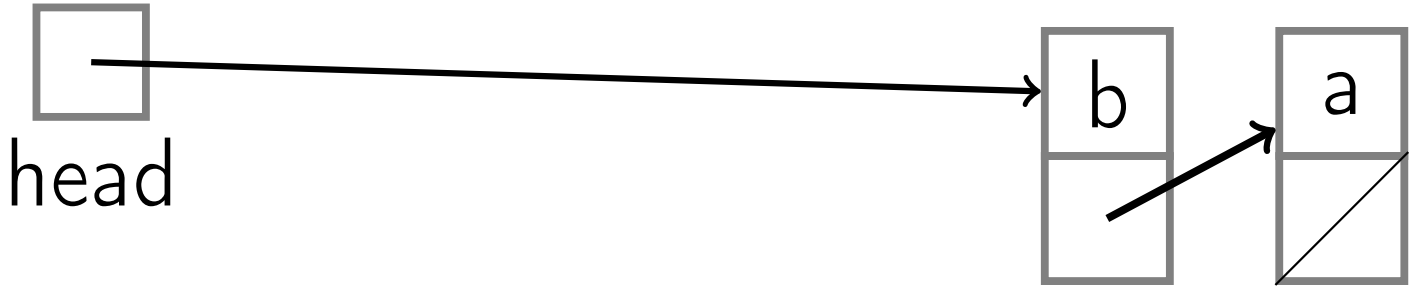
`Pop()`

Stack Implementation with Linked List



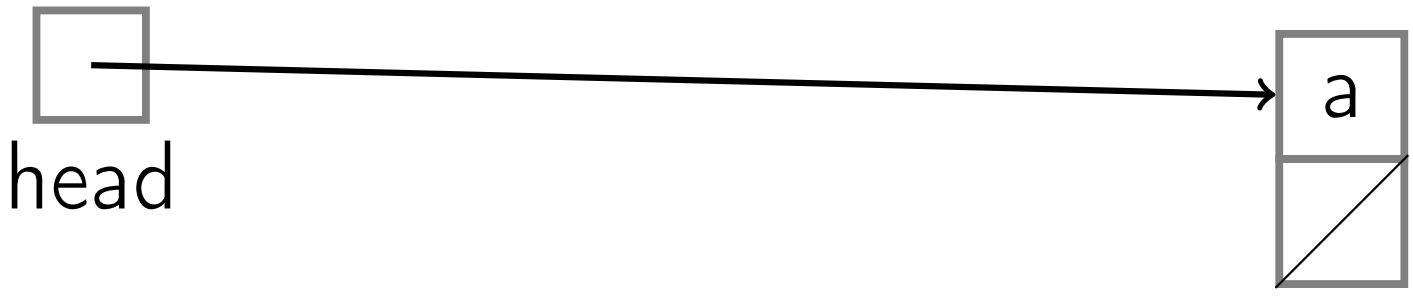
`Pop()` $\rightarrow$ `d`

Stack Implementation with Linked List



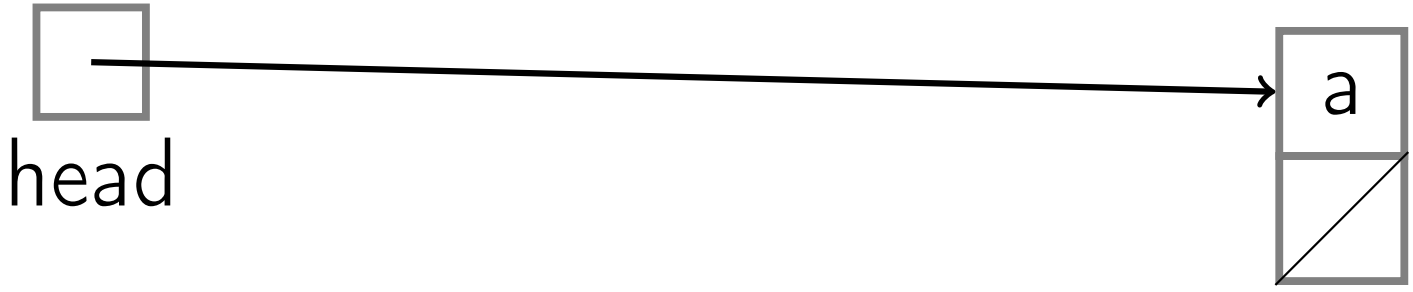
`Pop()`

Stack Implementation with Linked List



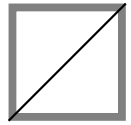
`Pop()` $\rightarrow$ `b`

Stack Implementation with Linked List



`Pop()`

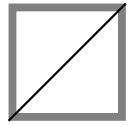
Stack Implementation with Linked List



head

`Pop()` $\rightarrow$ `a`

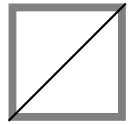
Stack Implementation with Linked List



head

`Empty()`

Stack Implementation with Linked List



head

`Empty()` $\rightarrow$ `True`

Summary

- Stacks can be implemented with either an array or a linked list.
- Each stack operation is $O(1)$: Push, Pop, Top, Empty.
- Stacks are ocassionaly known as LIFO queues.

Outline

1 Stacks

2 Queues

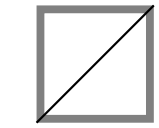
Definition

Queue: Abstract data type with the following operations:

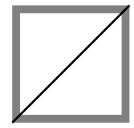
- `Enqueue(Key)`: adds key to collection
- `Key Dequeue()`: removes and returns least recently-added key
- `Boolean Empty()`: are there any elements?

FIFO: First-In, First-Out

Queue Implementation with Linked List



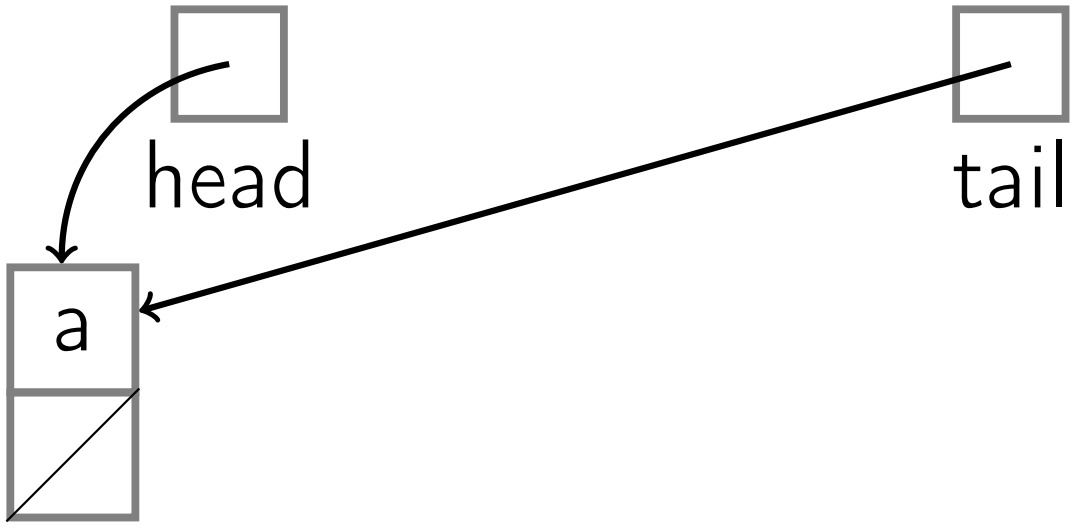
head



tail

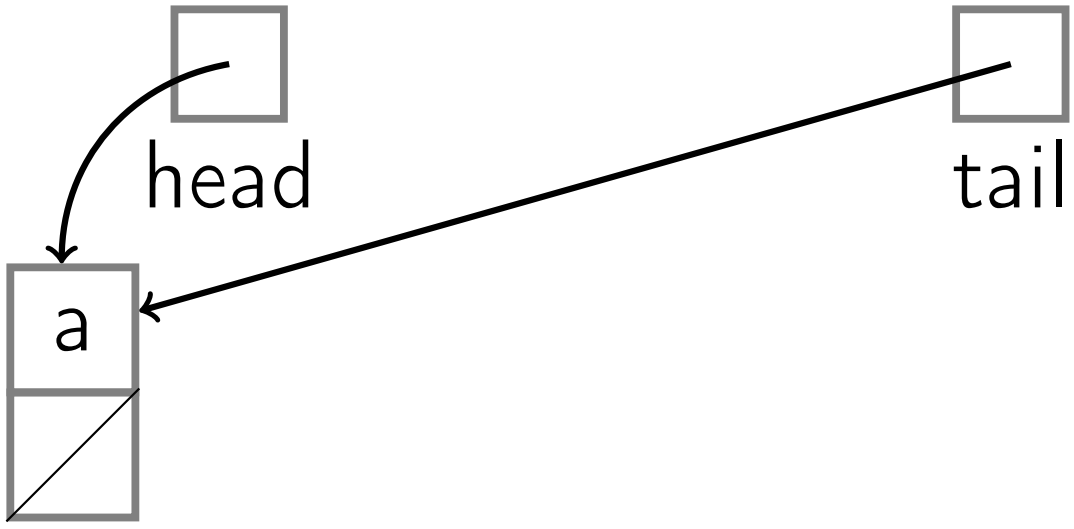
`Enqueue(a)`

Queue Implementation with Linked List



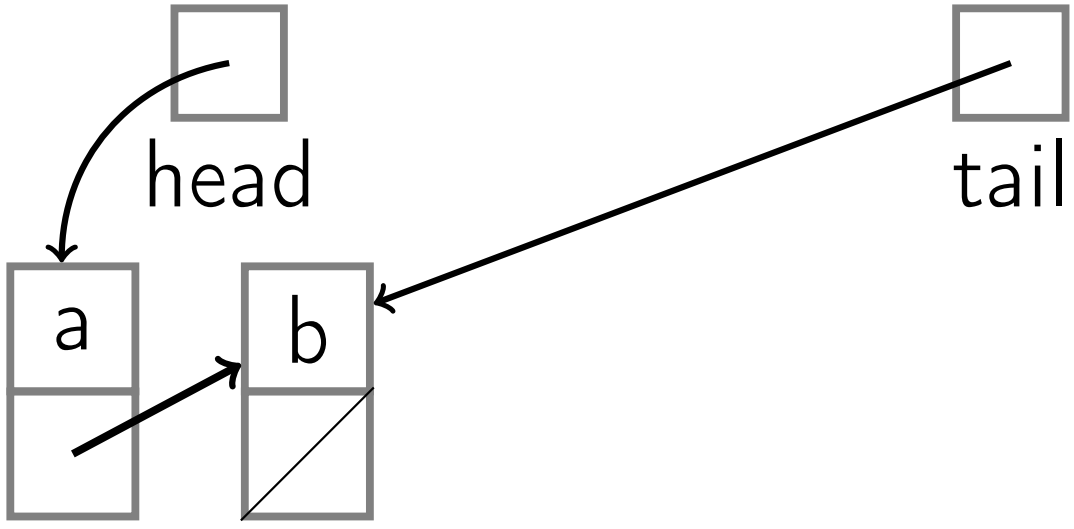
`Enqueue(a)`

Queue Implementation with Linked List



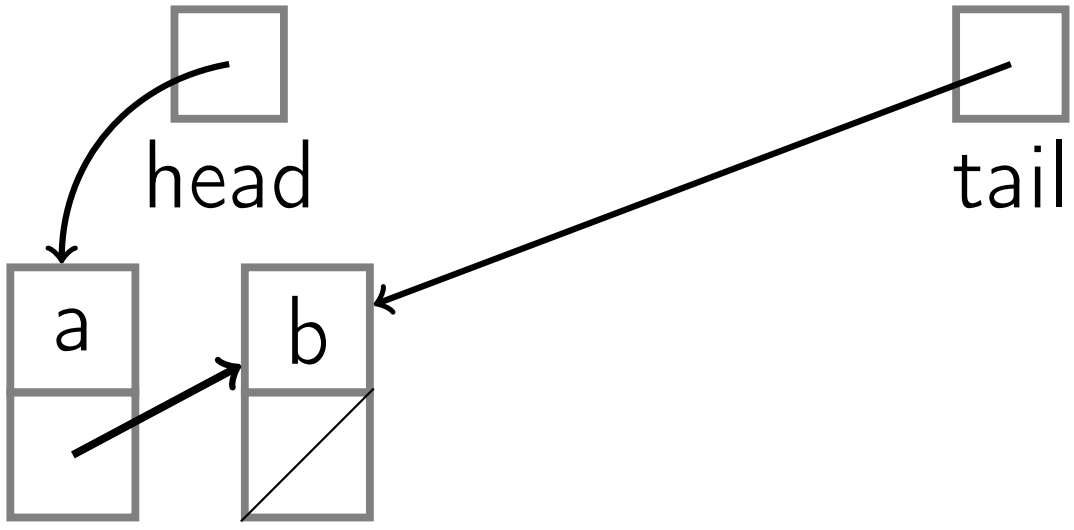
`Enqueue(b)`

Queue Implementation with Linked List



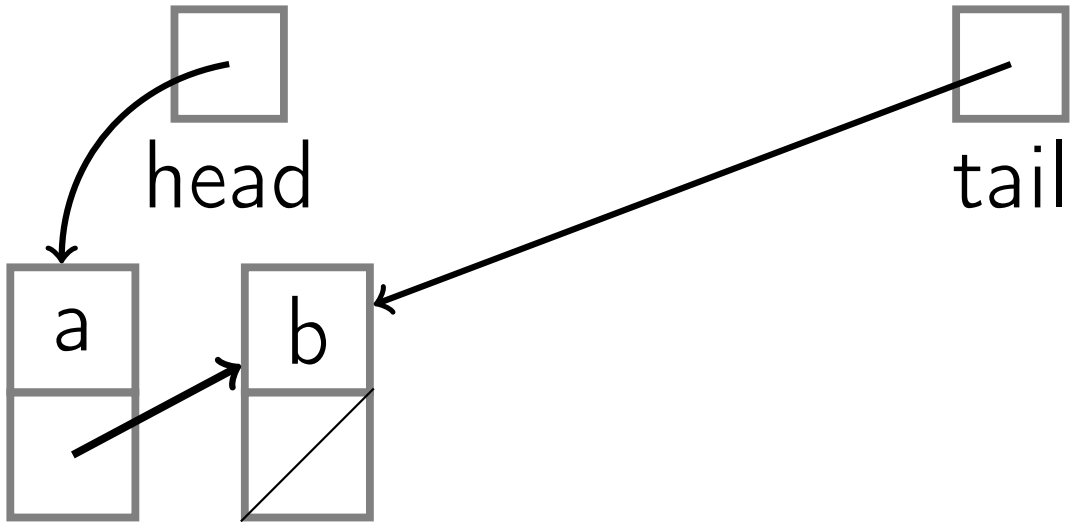
`Enqueue(b)`

Queue Implementation with Linked List



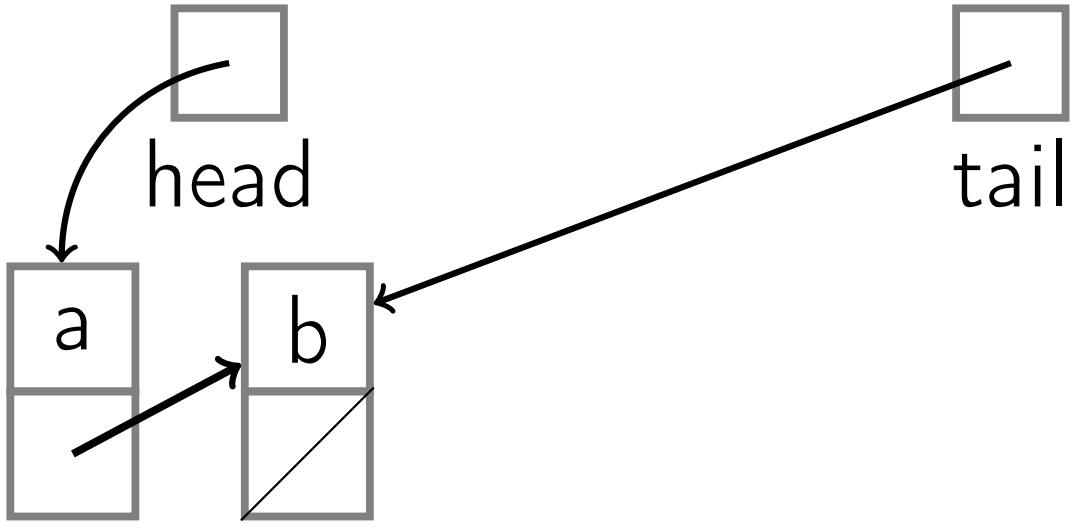
`Empty()`

Queue Implementation with Linked List



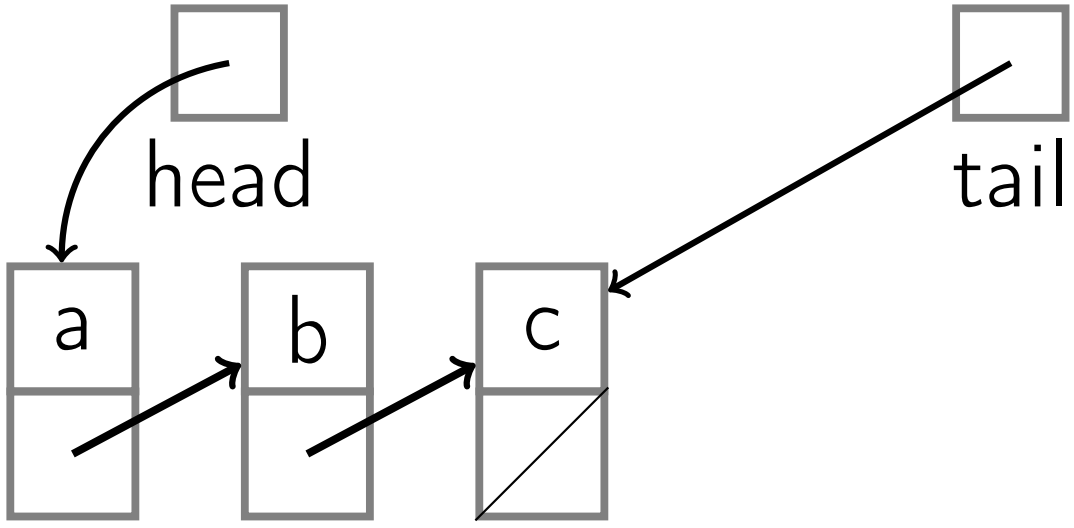
`Empty()` $\rightarrow$ `False`

Queue Implementation with Linked List



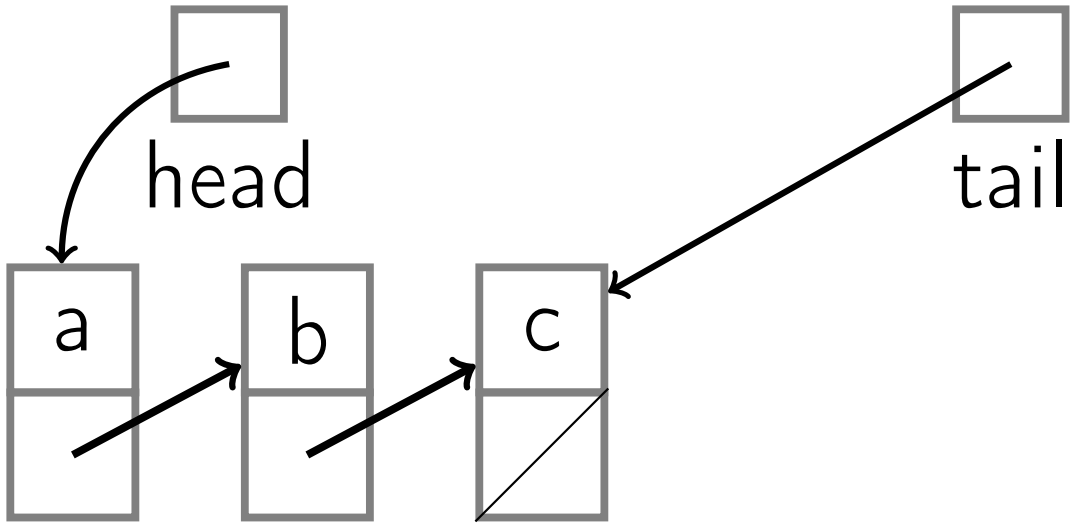
`Enqueue(c)`

Queue Implementation with Linked List



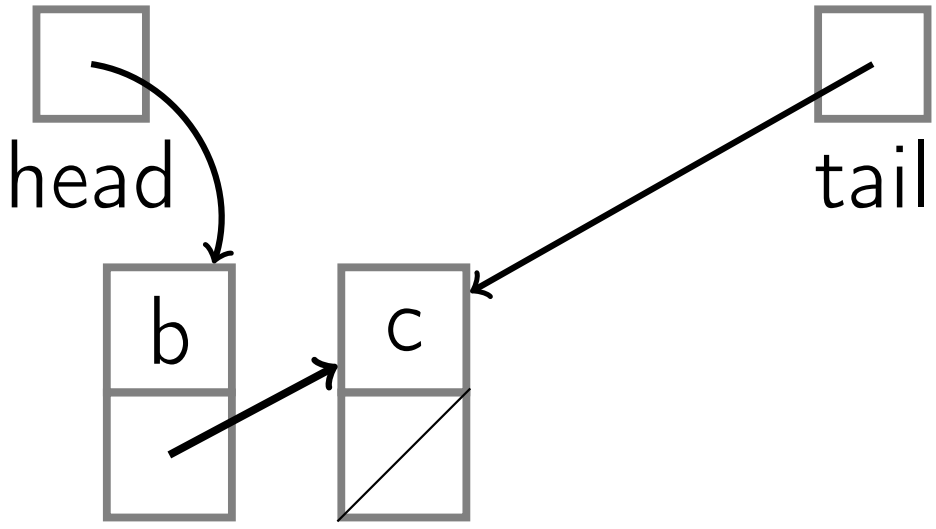
`Enqueue(c)`

Queue Implementation with Linked List



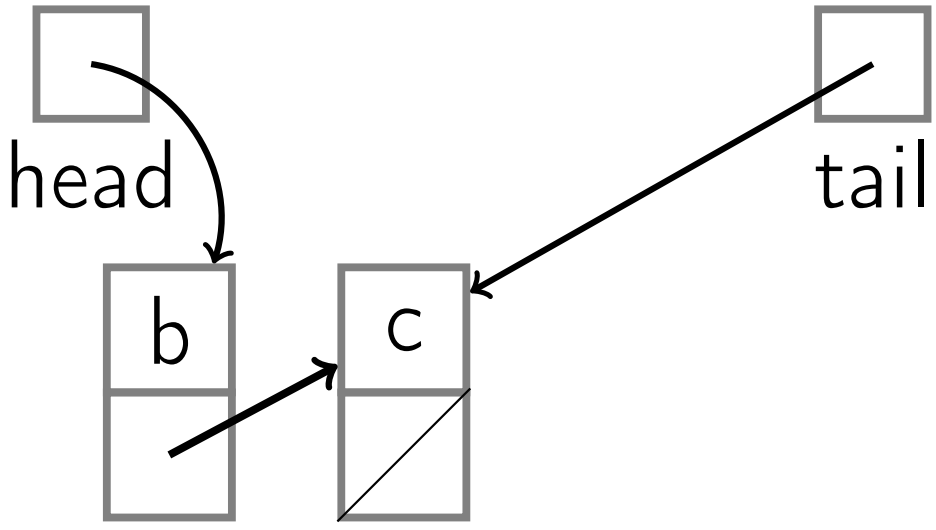
`Dequeue()`

Queue Implementation with Linked List



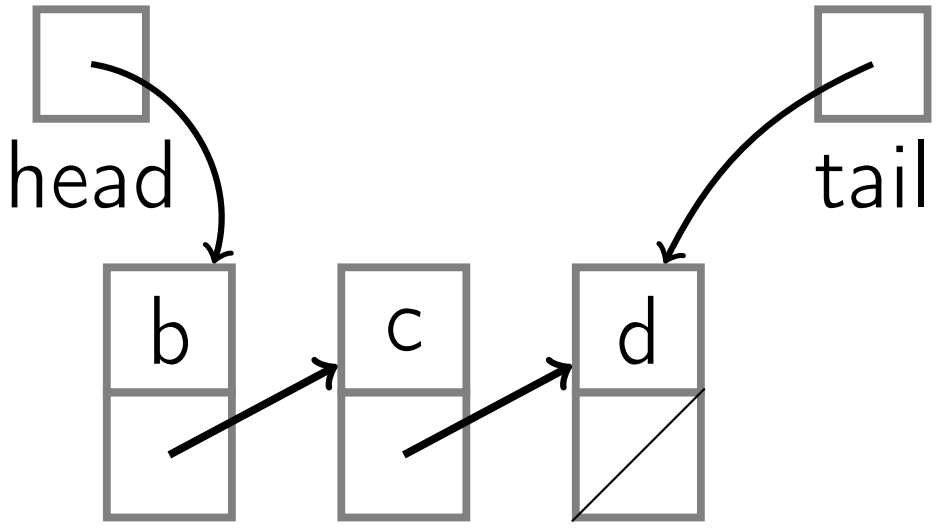
`Dequeue()` $\rightarrow$ `a`

Queue Implementation with Linked List



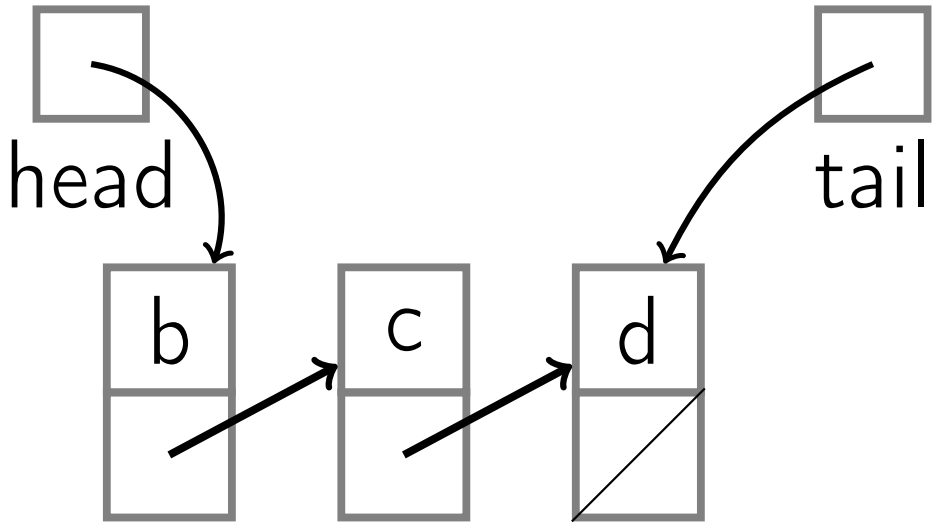
`Enqueue(d)`

Queue Implementation with Linked List



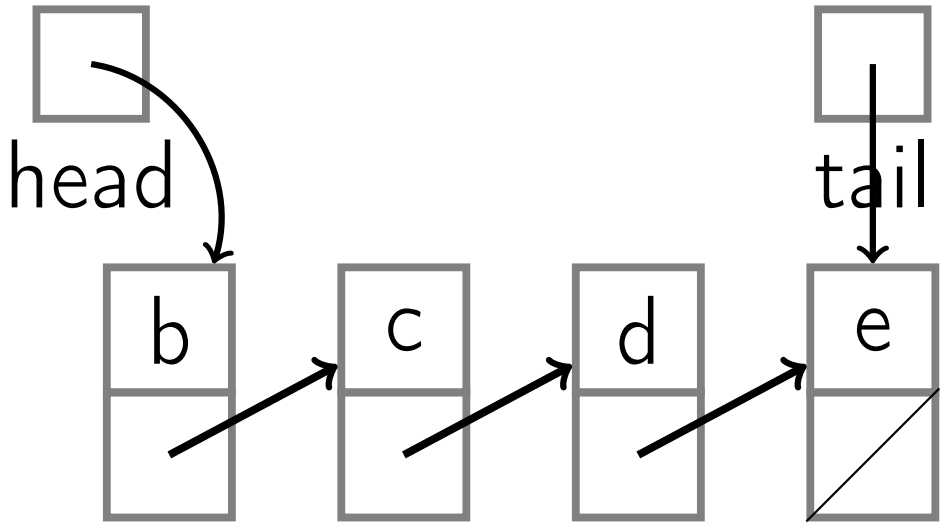
`Enqueue(d)`

Queue Implementation with Linked List



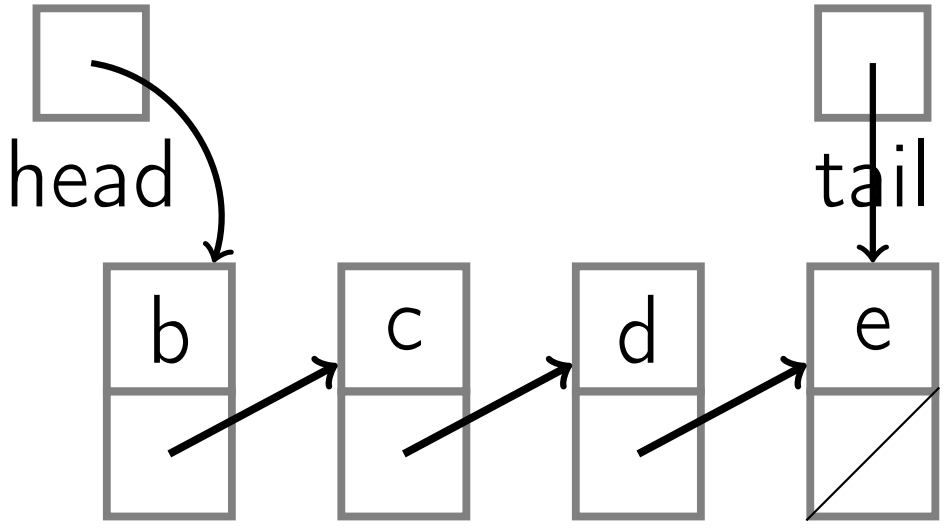
Enqueue(e)

Queue Implementation with Linked List



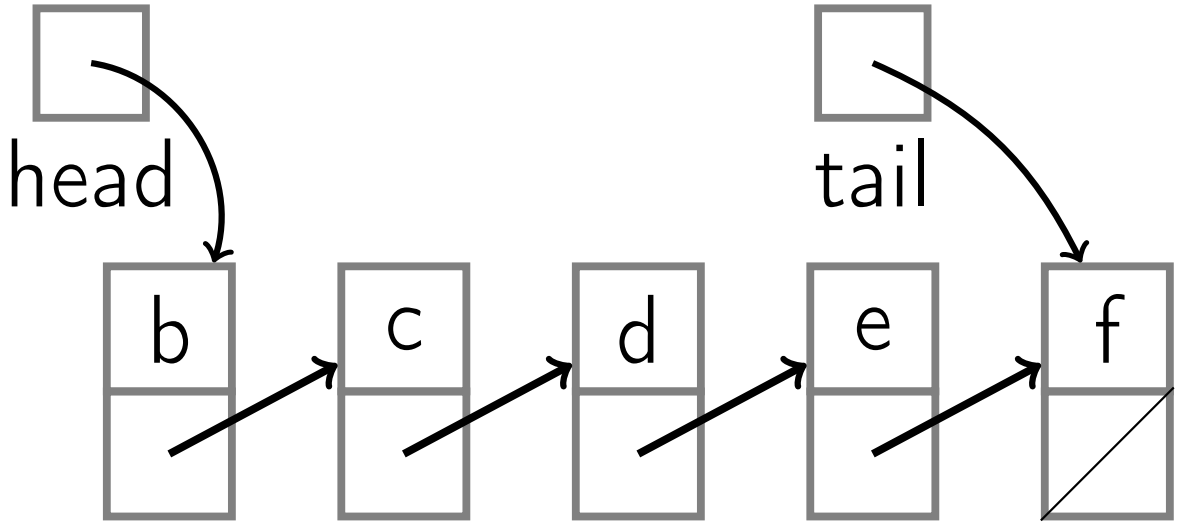
Enqueue(e)

Queue Implementation with Linked List



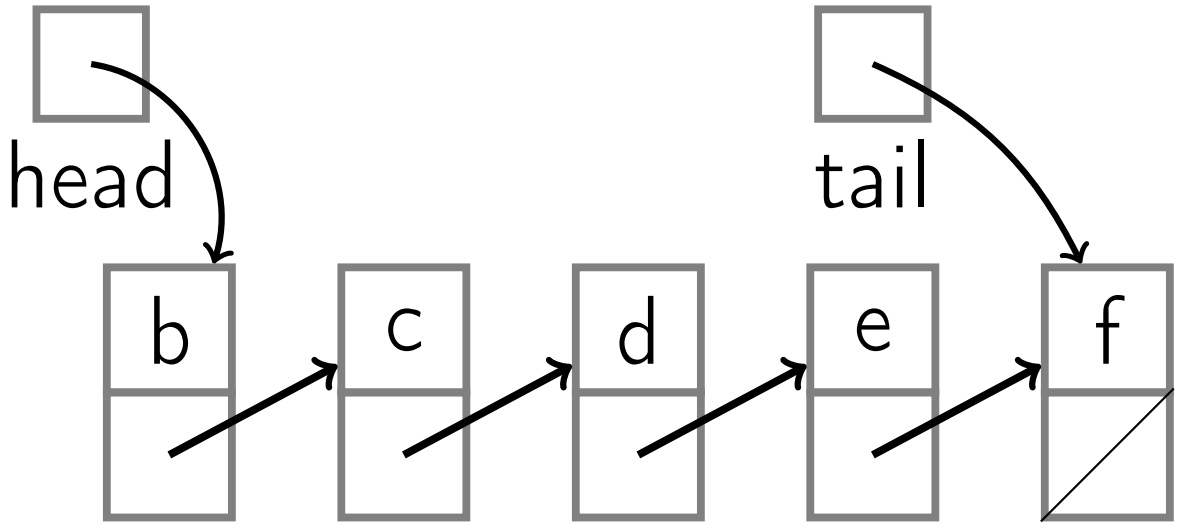
`Enqueue(f)`

Queue Implementation with Linked List



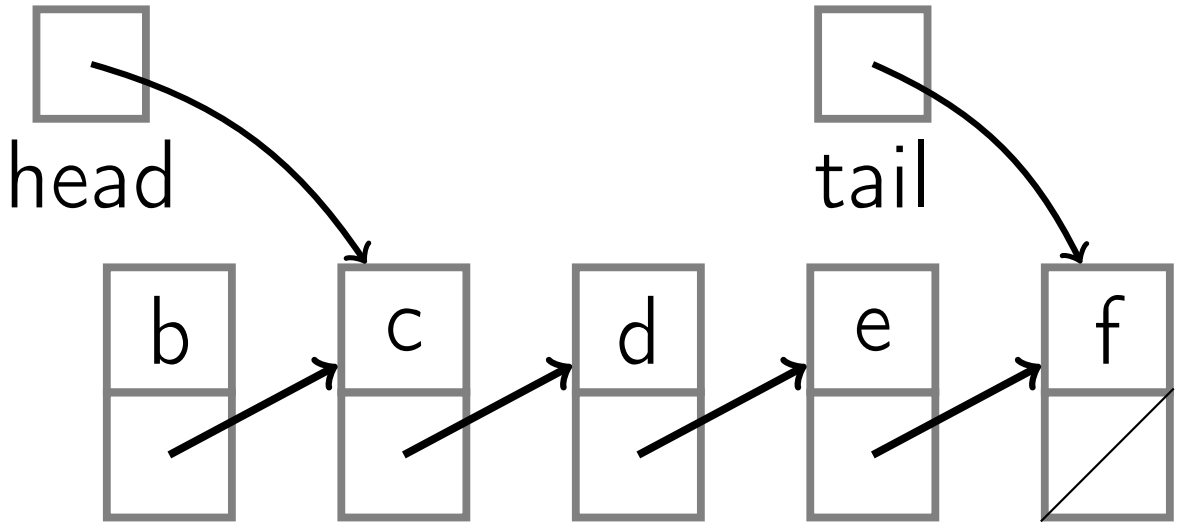
`Enqueue(f)`

Queue Implementation with Linked List



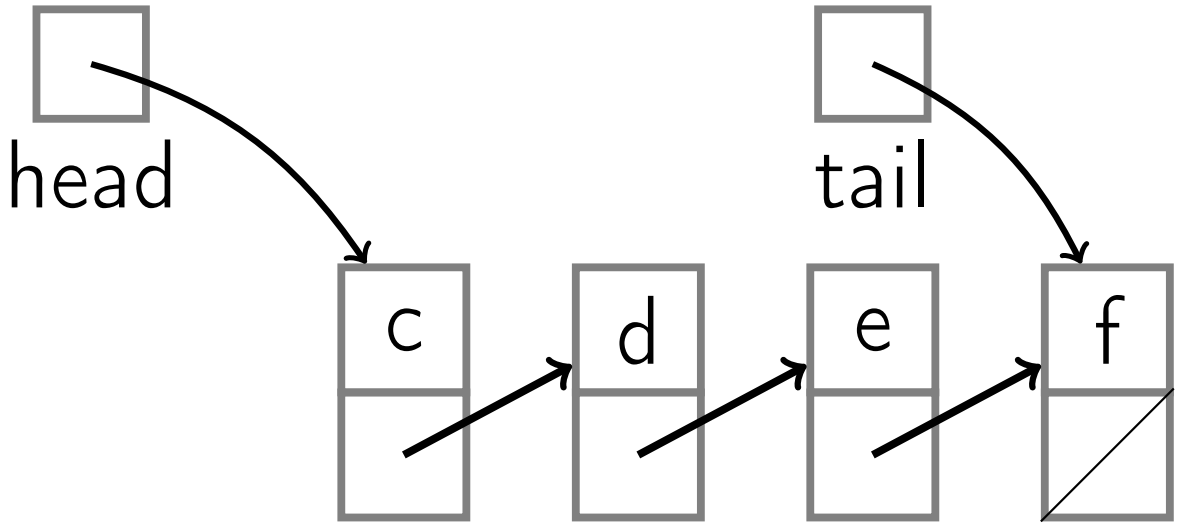
Dequeue()

Queue Implementation with Linked List

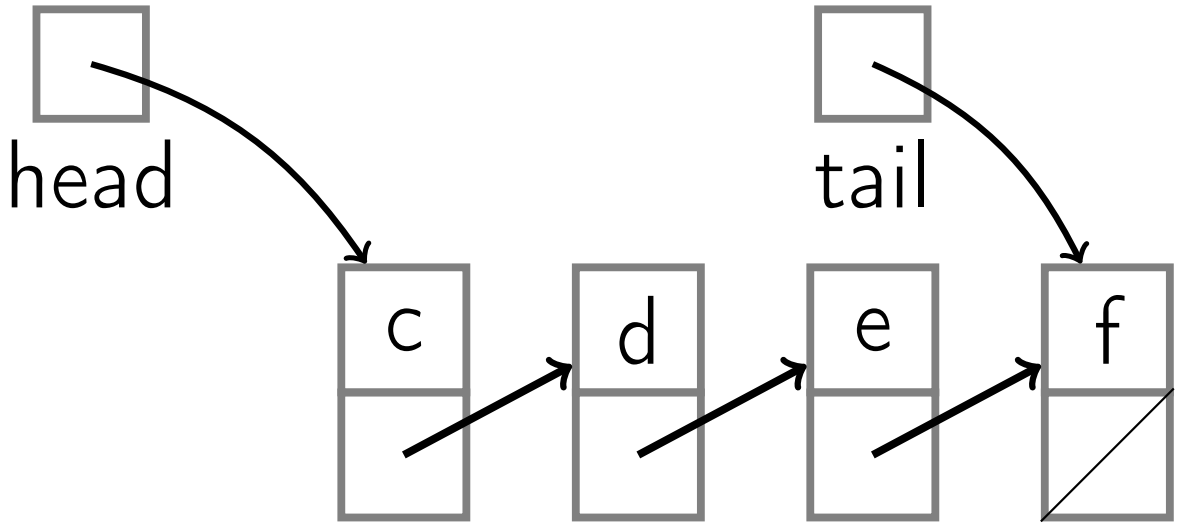


Dequeue()$\rightarrow$ b

Queue Implementation with Linked List

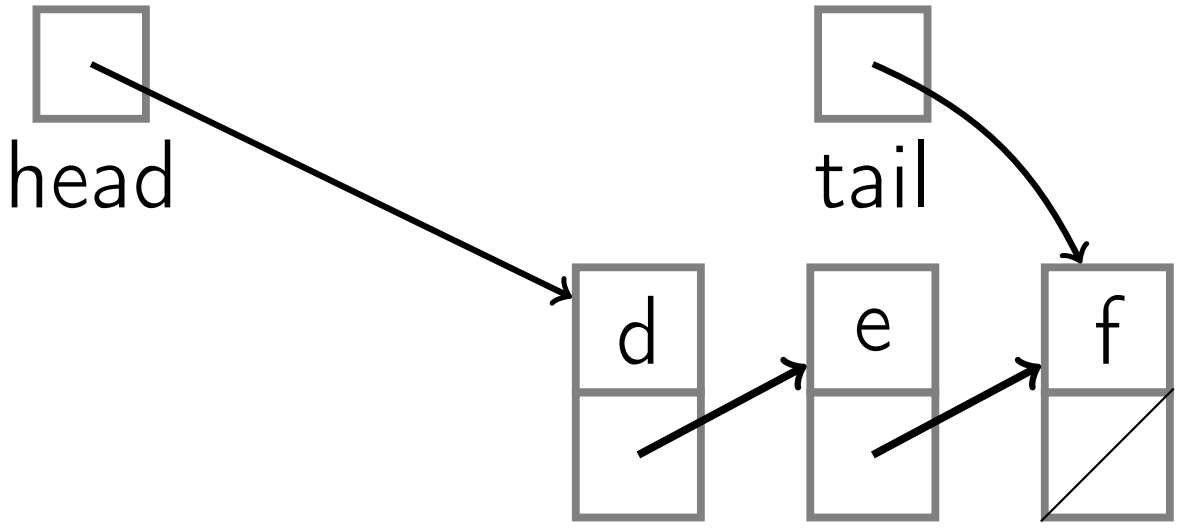


Queue Implementation with Linked List



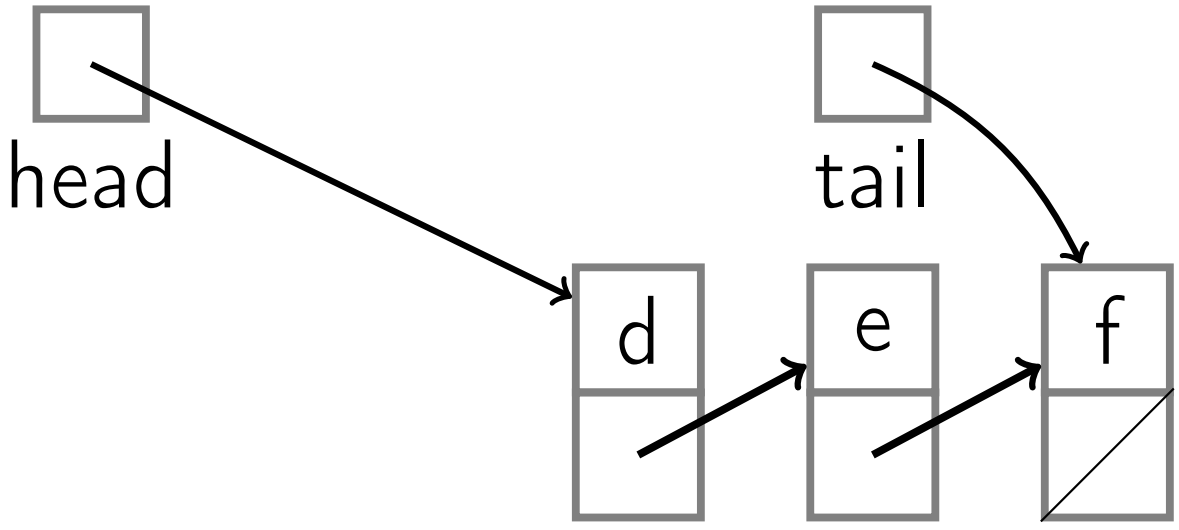
`Dequeue()`

Queue Implementation with Linked List



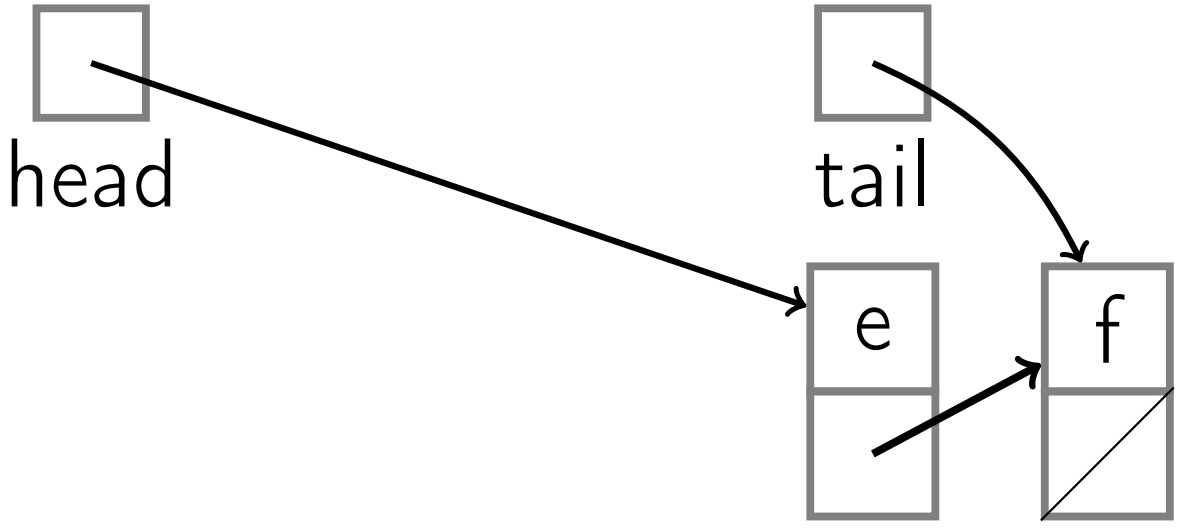
`Dequeue()` $\rightarrow$ `c`

Queue Implementation with Linked List



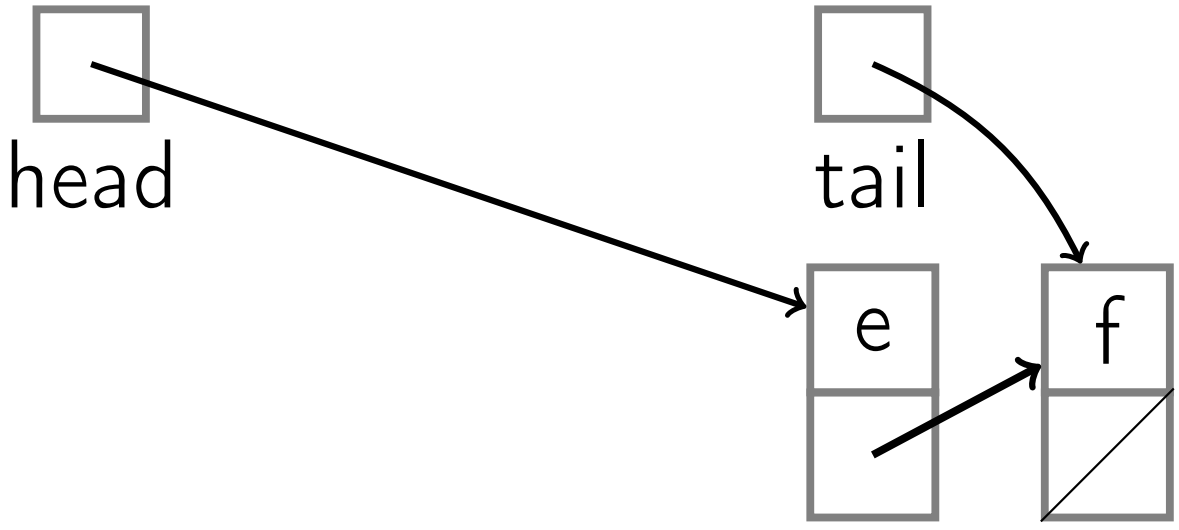
`Dequeue()`

Queue Implementation with Linked List



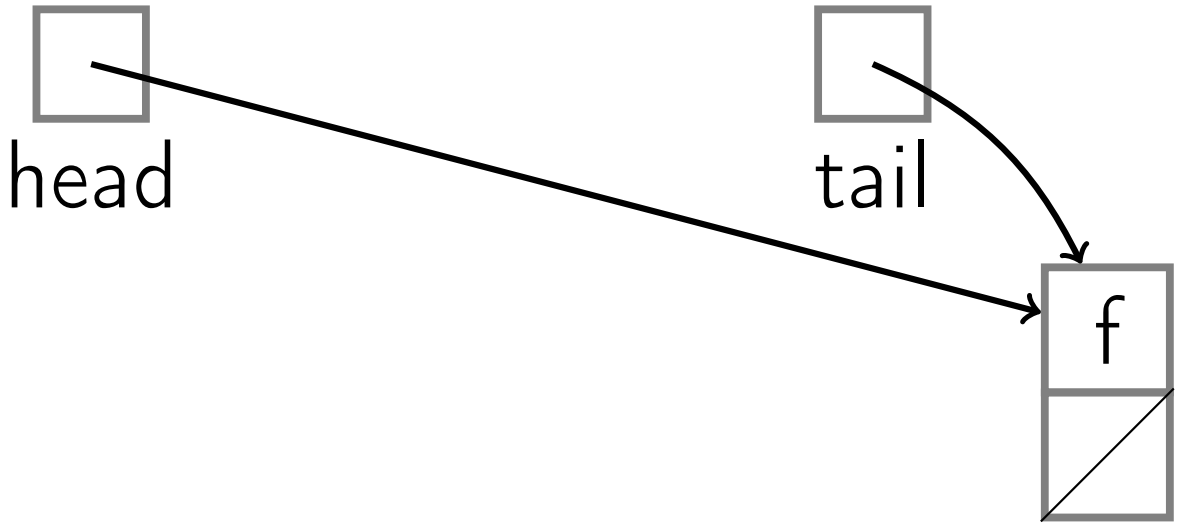
`Dequeue()` $\rightarrow$ `d`

Queue Implementation with Linked List



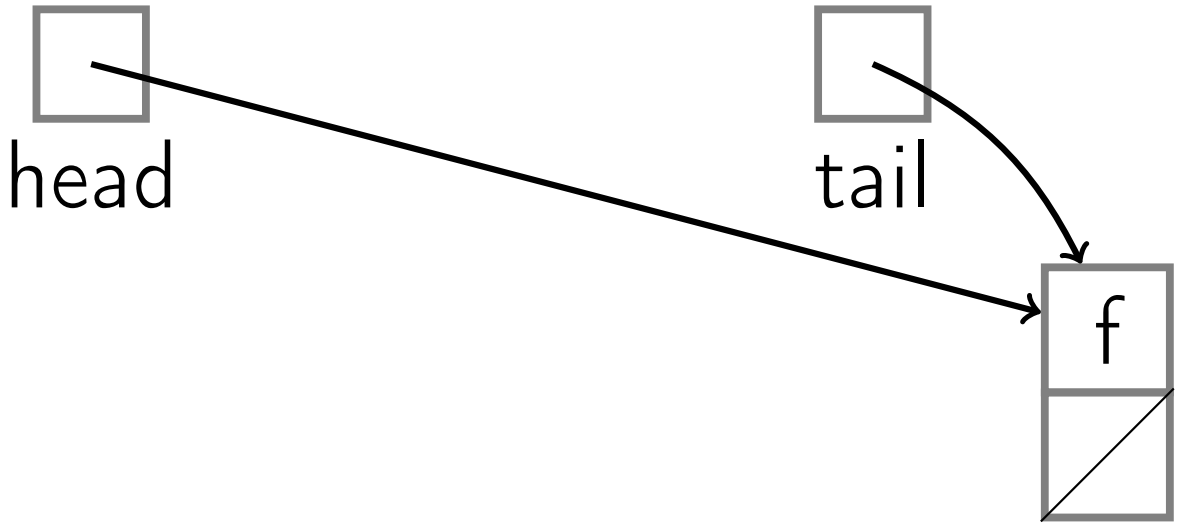
`Dequeue()`

Queue Implementation with Linked List



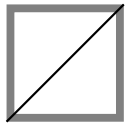
`Dequeue()` $\rightarrow$ `e`

Queue Implementation with Linked List

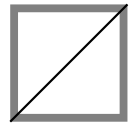


`Dequeue()`

Queue Implementation with Linked List



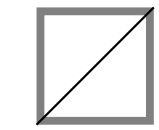
head



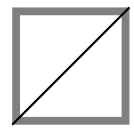
tail

`Dequeue()` $\rightarrow$ `f`

Queue Implementation with Linked List



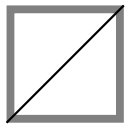
head



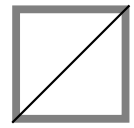
tail

`Empty()`

Queue Implementation with Linked List



head



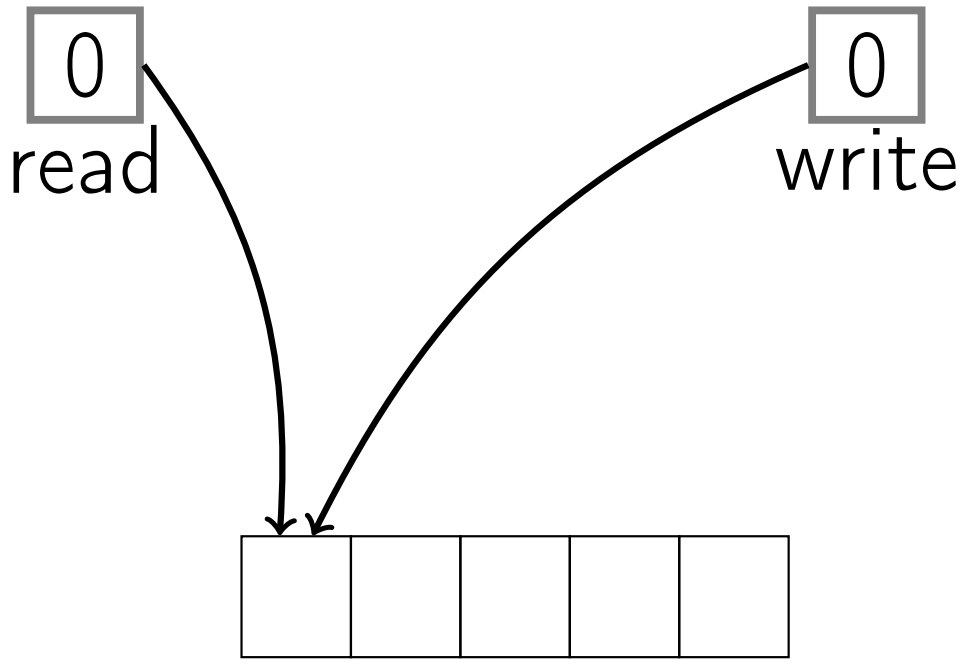
tail

`Empty()` $\rightarrow$ `True`

Queue Implementation with Linked List

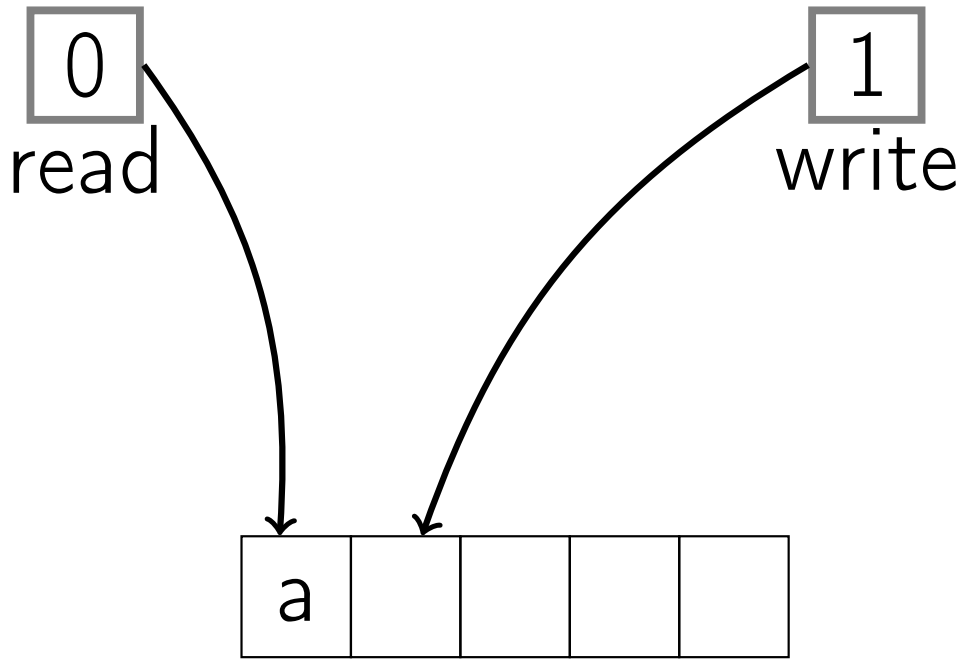
- `Enqueue: use List.PushBack`
- `Dequeue: use List.TopFront and List.PopFront`
- `Empty: use List.Empty`

Queue Implementation with Array



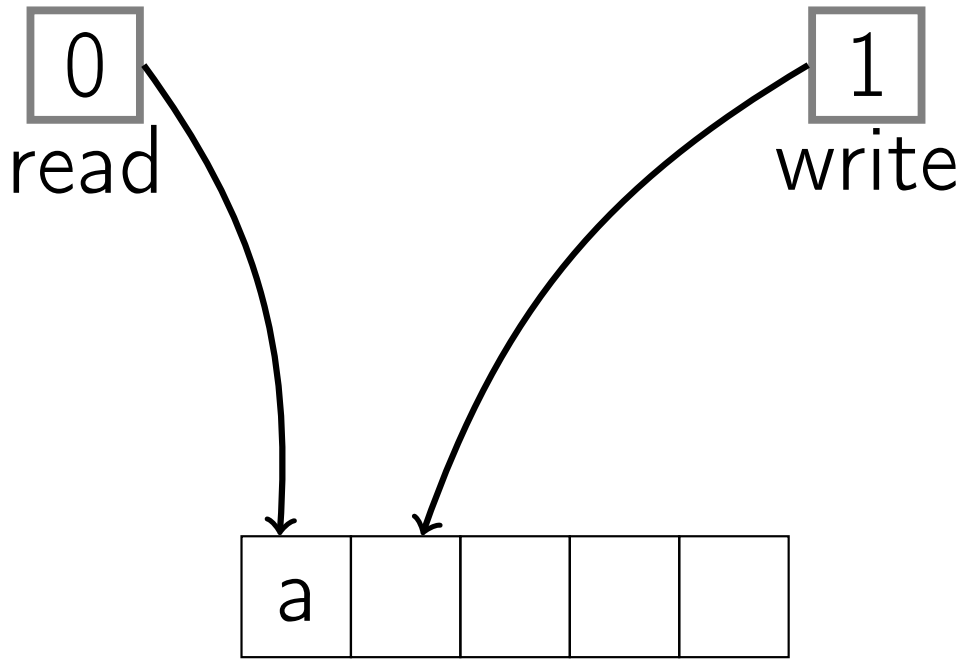
Enqueue(a)

Queue Implementation with Array



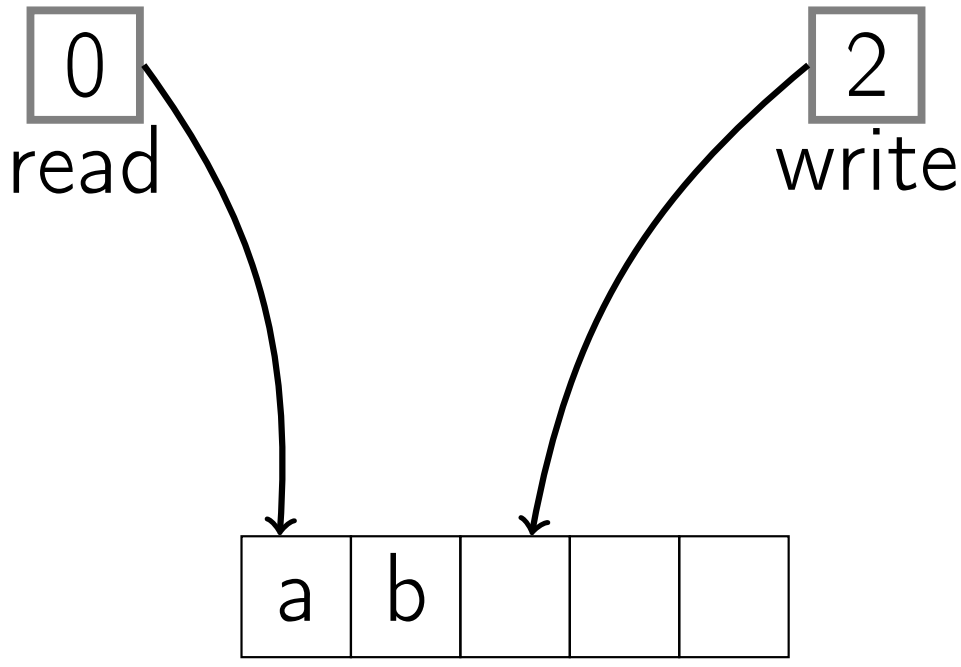
Enqueue(a)

Queue Implementation with Array



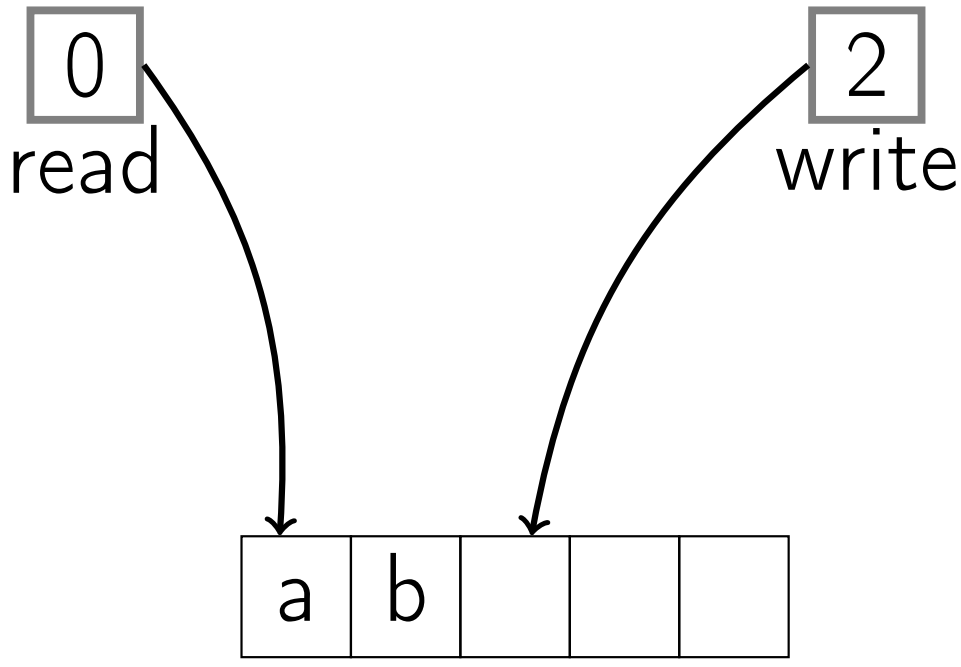
`Enqueue(b)`

Queue Implementation with Array



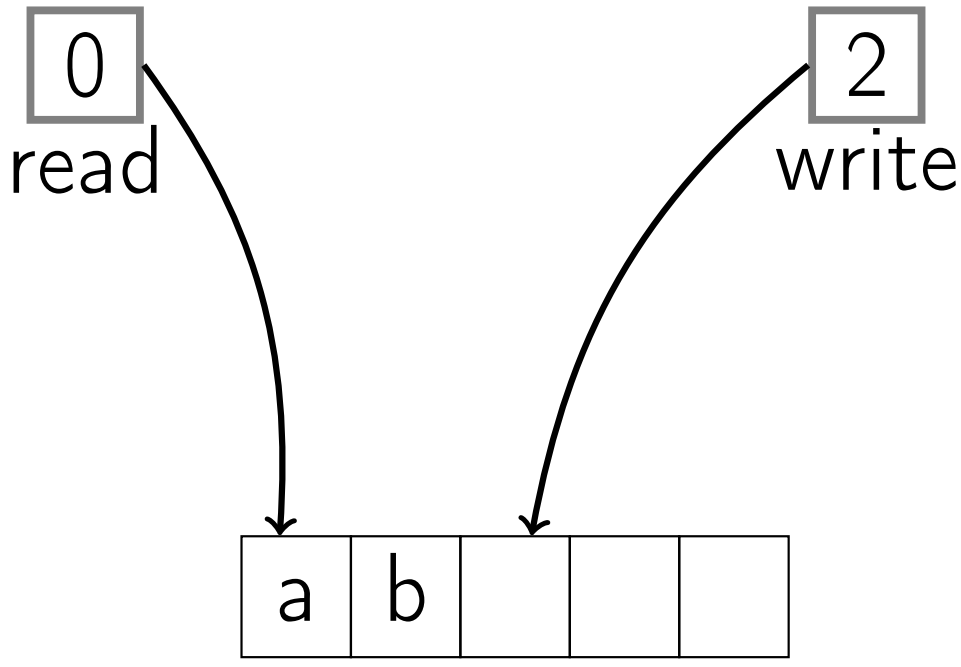
`Enqueue(b)`

Queue Implementation with Array



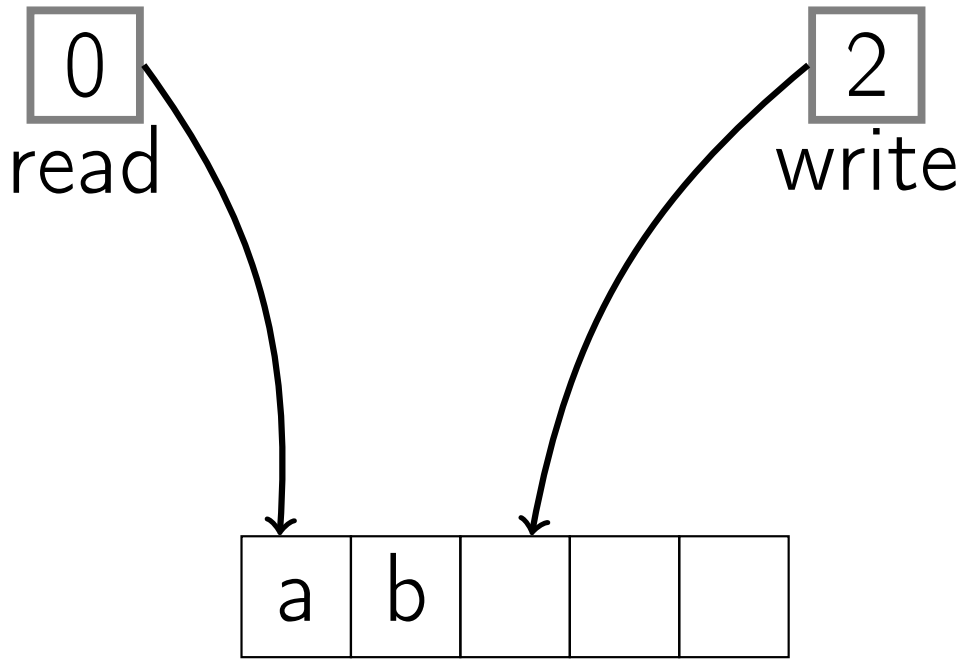

```
Empty()
```

Queue Implementation with Array



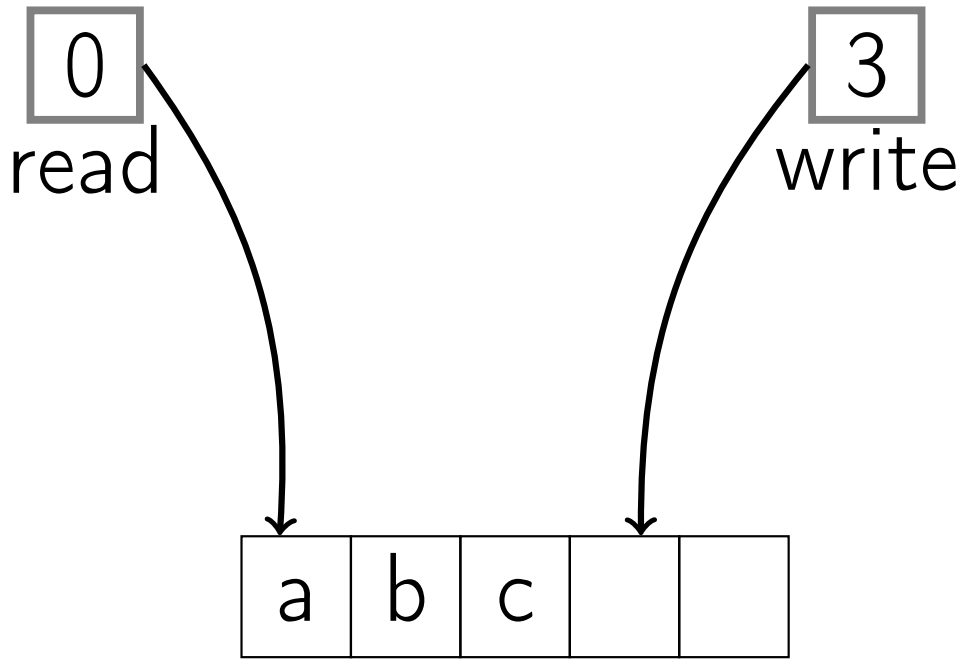
`Empty()` $\rightarrow$ `False`

Queue Implementation with Array



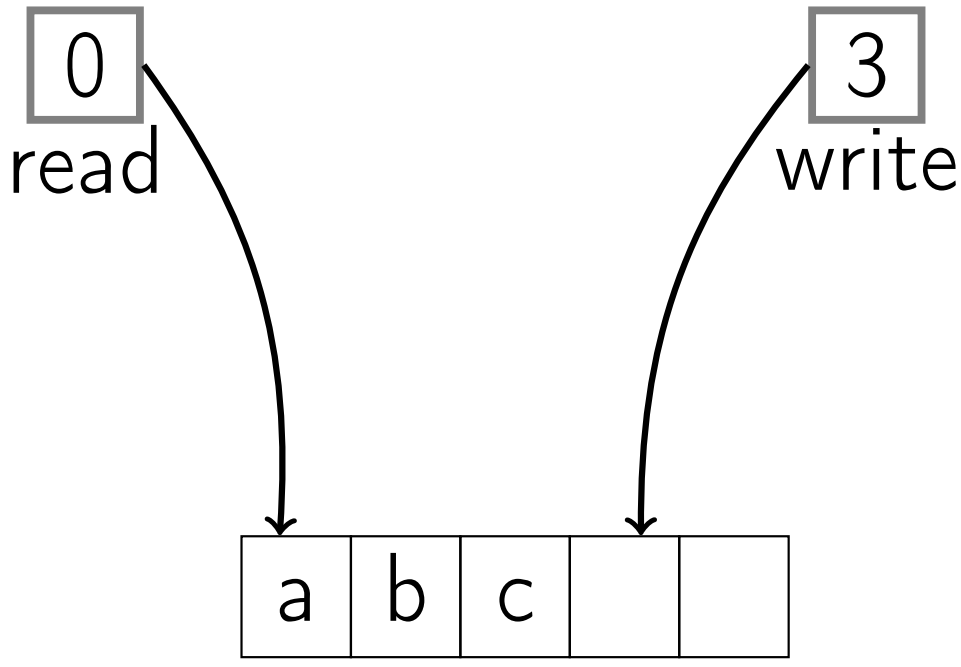
`Enqueue(c)`

Queue Implementation with Array



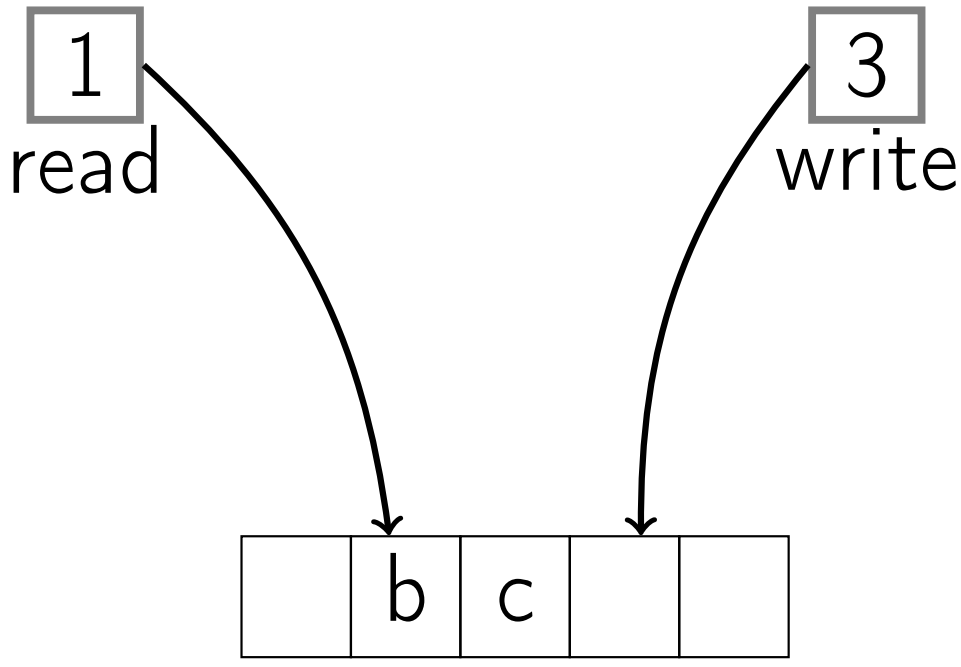
`Enqueue(c)`

Queue Implementation with Array



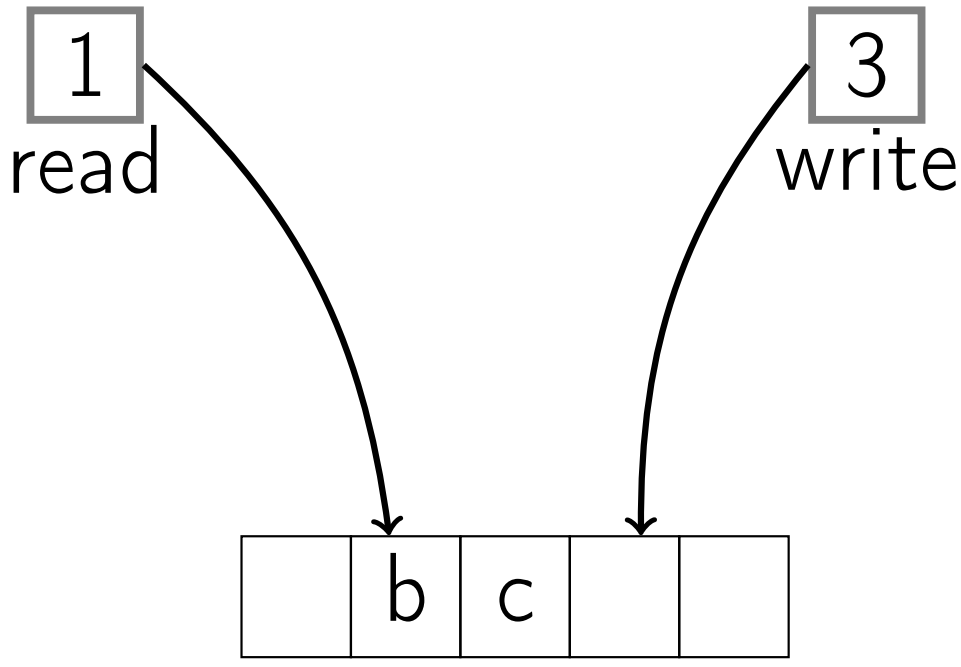
`Dequeue()`

Queue Implementation with Array



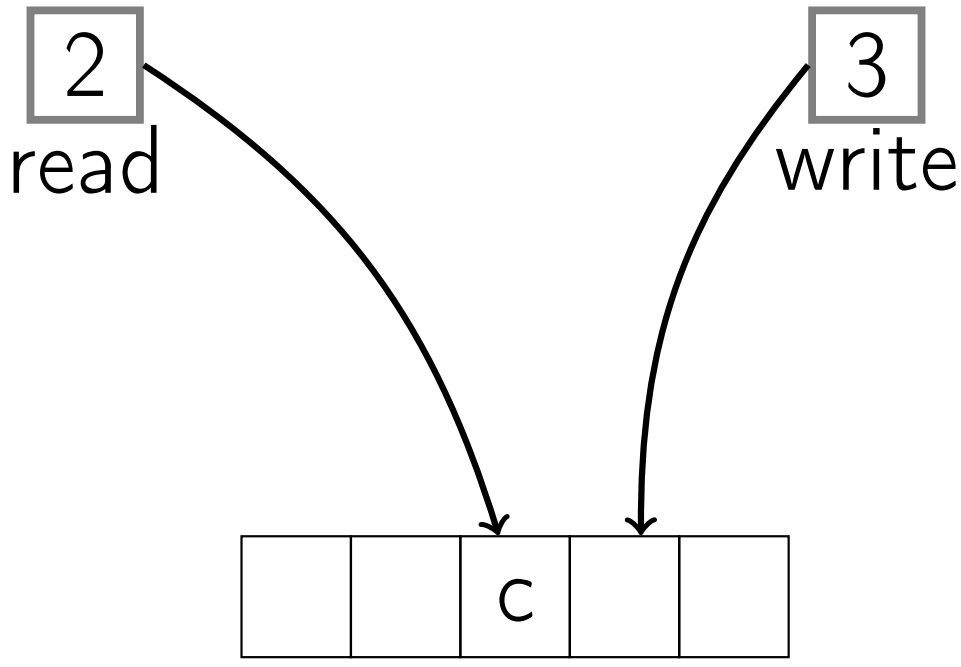
`Dequeue()` $\rightarrow$ `a`

Queue Implementation with Array



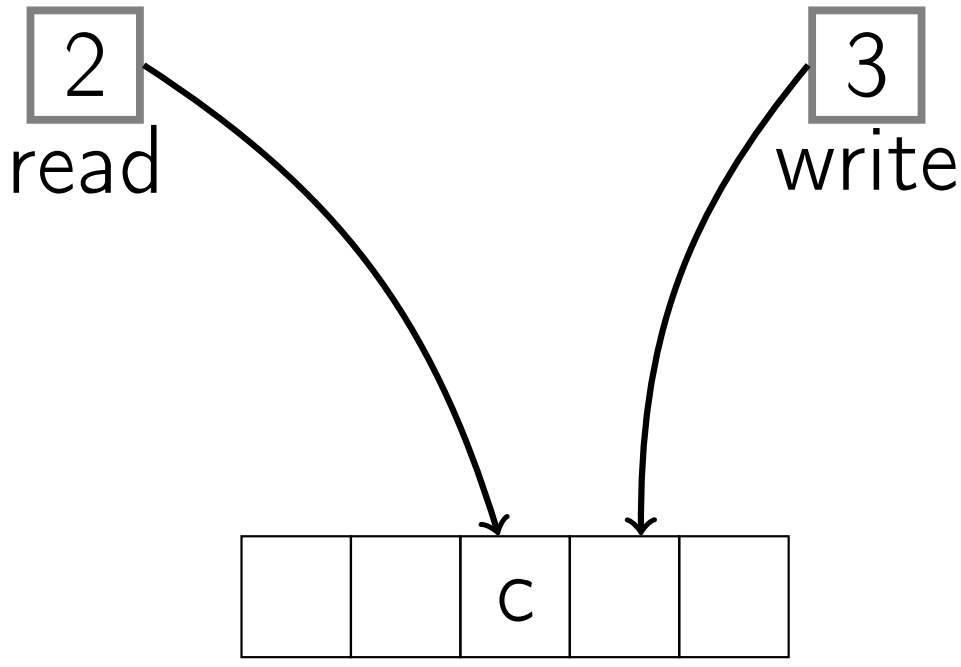
`Dequeue()`

Queue Implementation with Array



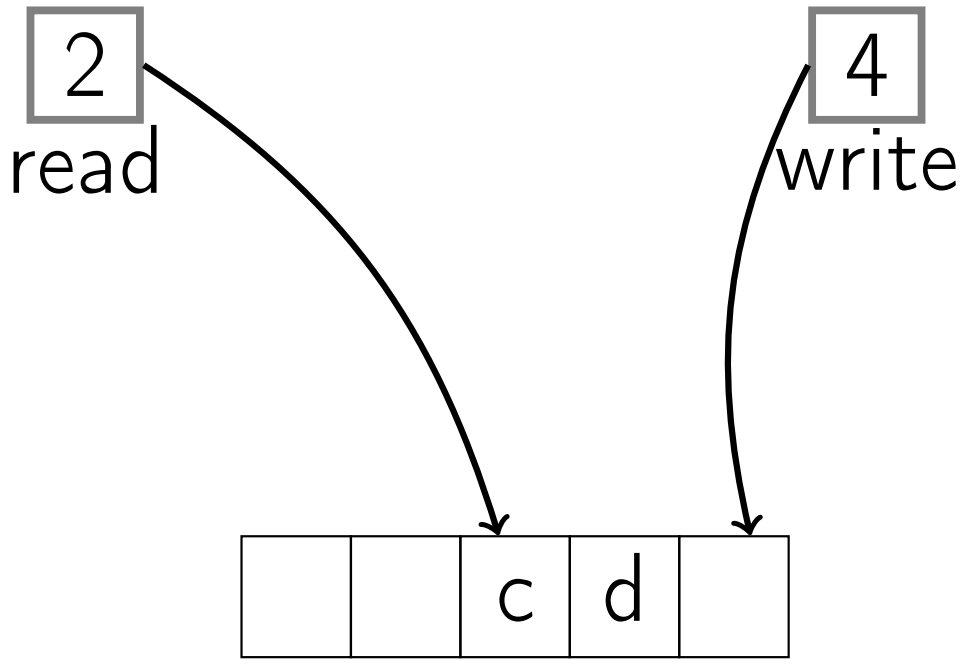
`Dequeue()` $\rightarrow$ `b`

Queue Implementation with Array



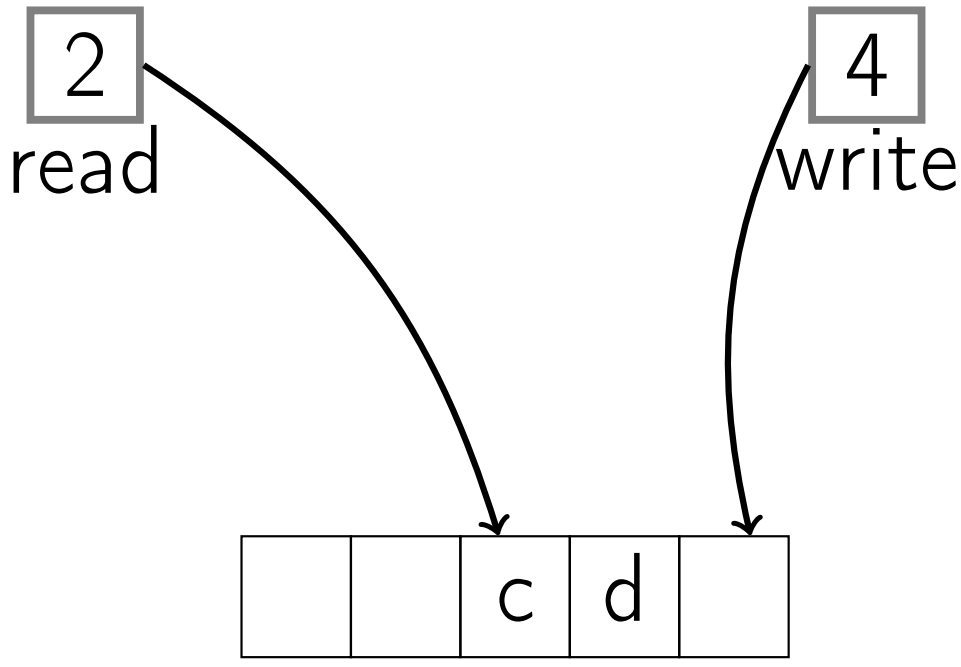
`Enqueue(d)`

Queue Implementation with Array



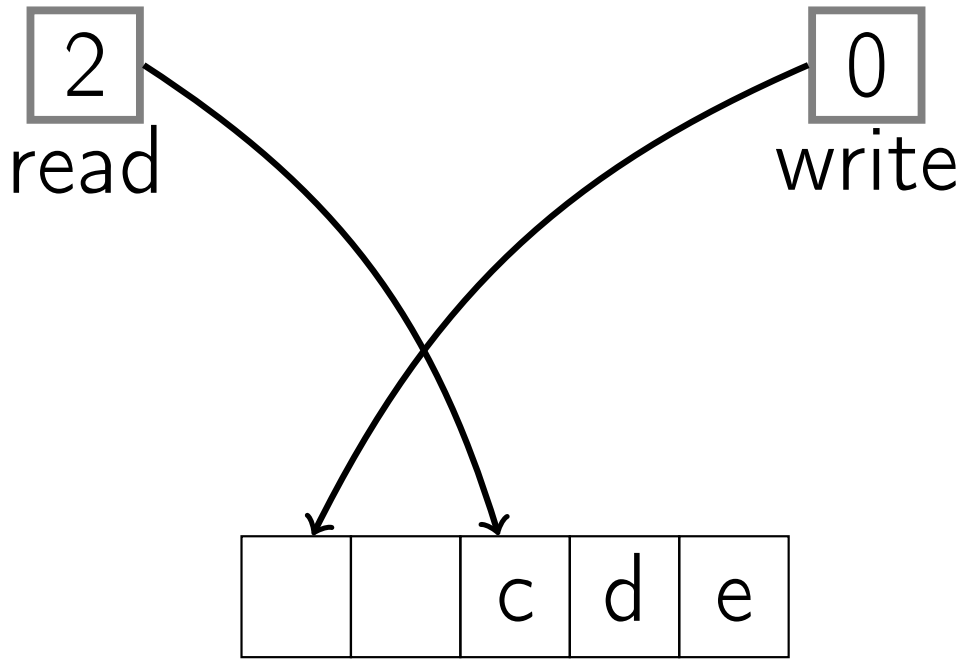
`Enqueue(d)`

Queue Implementation with Array



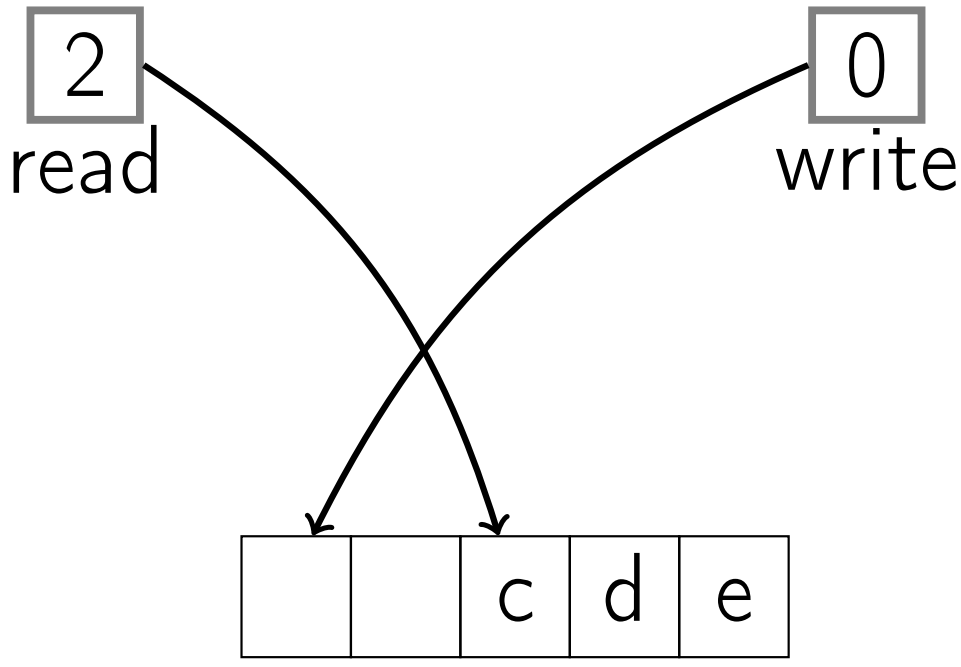
`Enqueue(e)`

Queue Implementation with Array



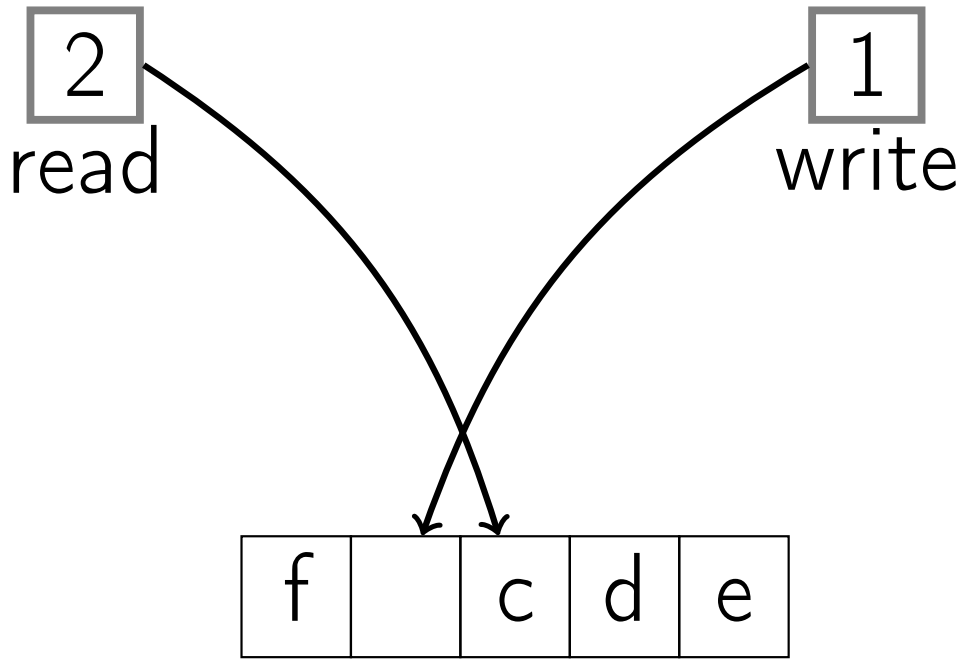
`Enqueue(e)`

Queue Implementation with Array



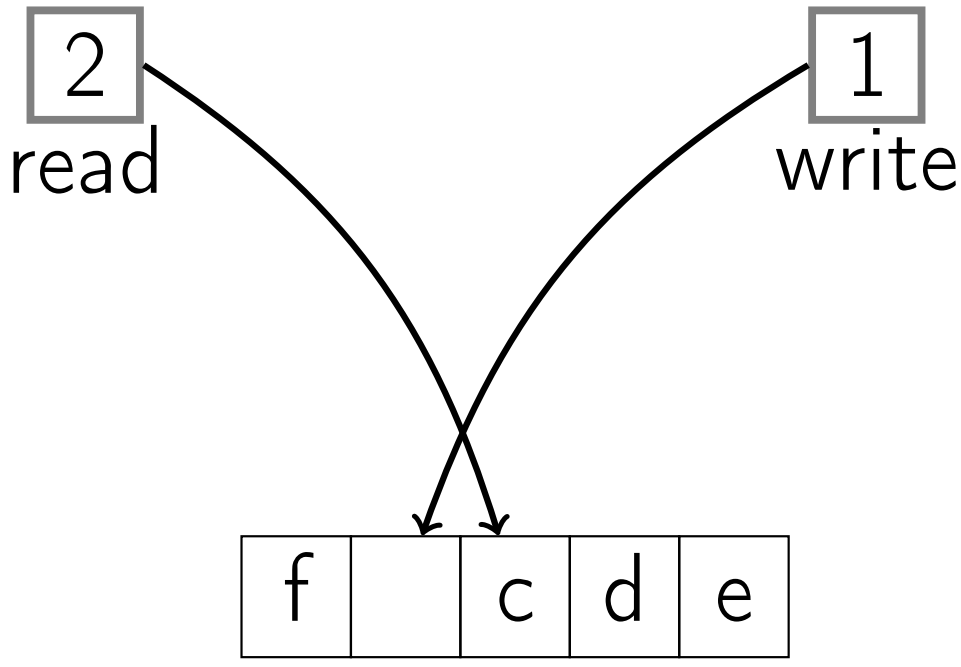
`Enqueue(f)`

Queue Implementation with Array



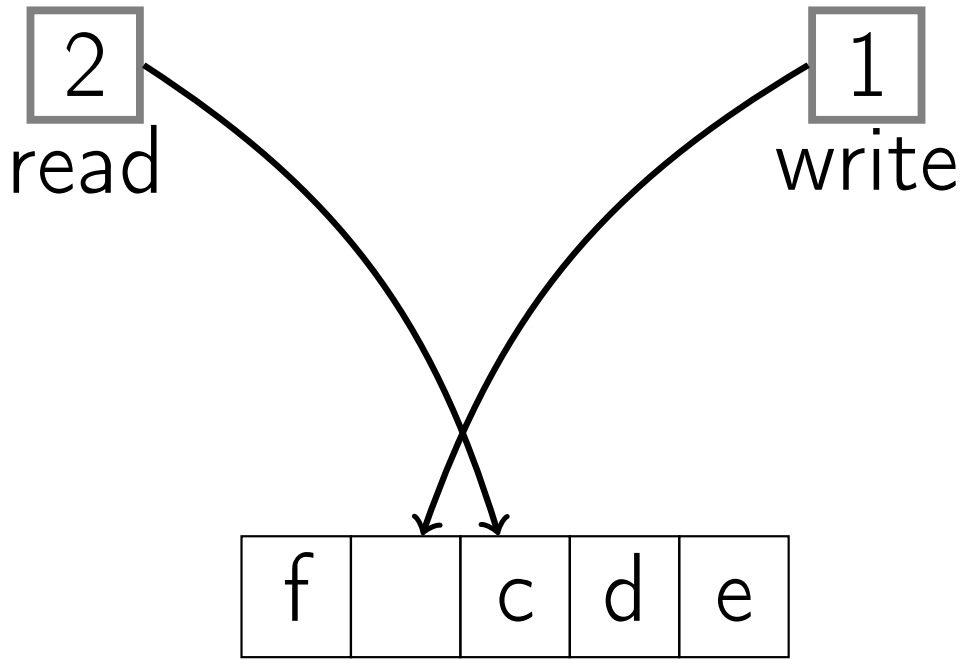
`Enqueue(f)`

Queue Implementation with Array



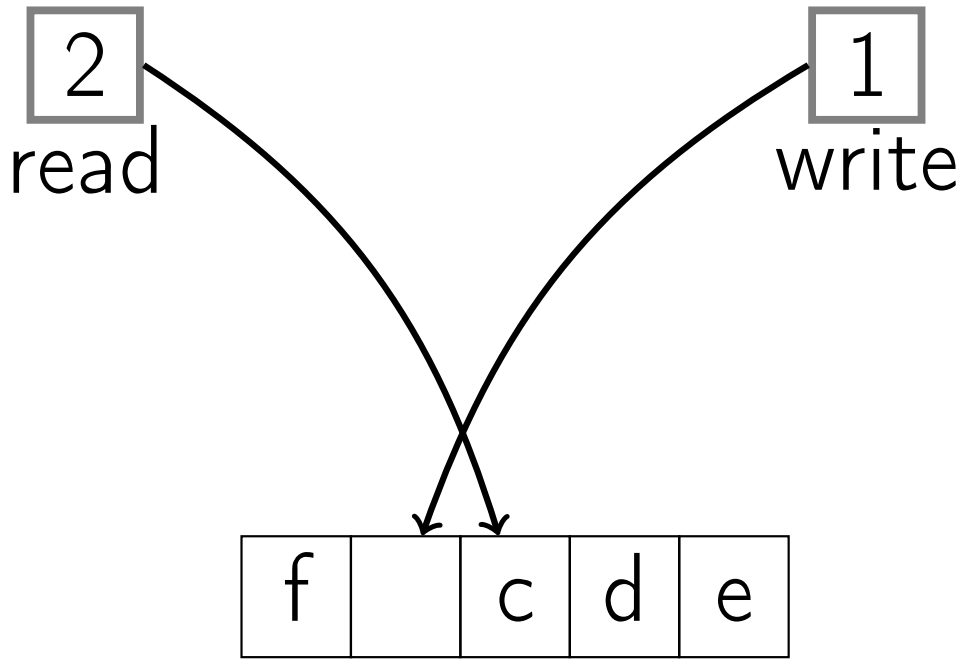
`Enqueue(g)`

Queue Implementation with Array



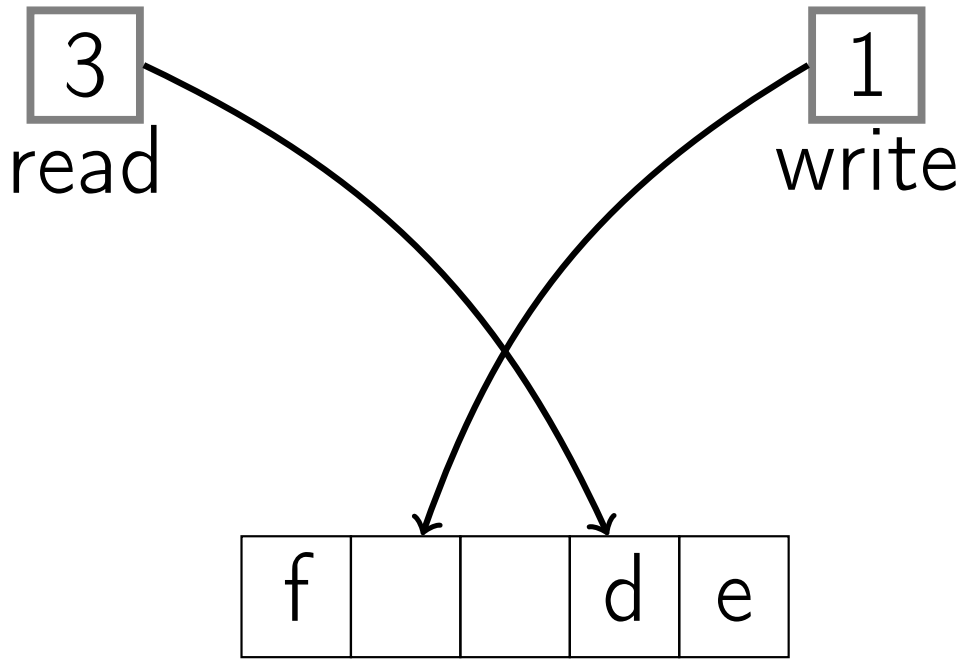
`Enqueue(g)` $\rightarrow$ `ERROR`

Queue Implementation with Array



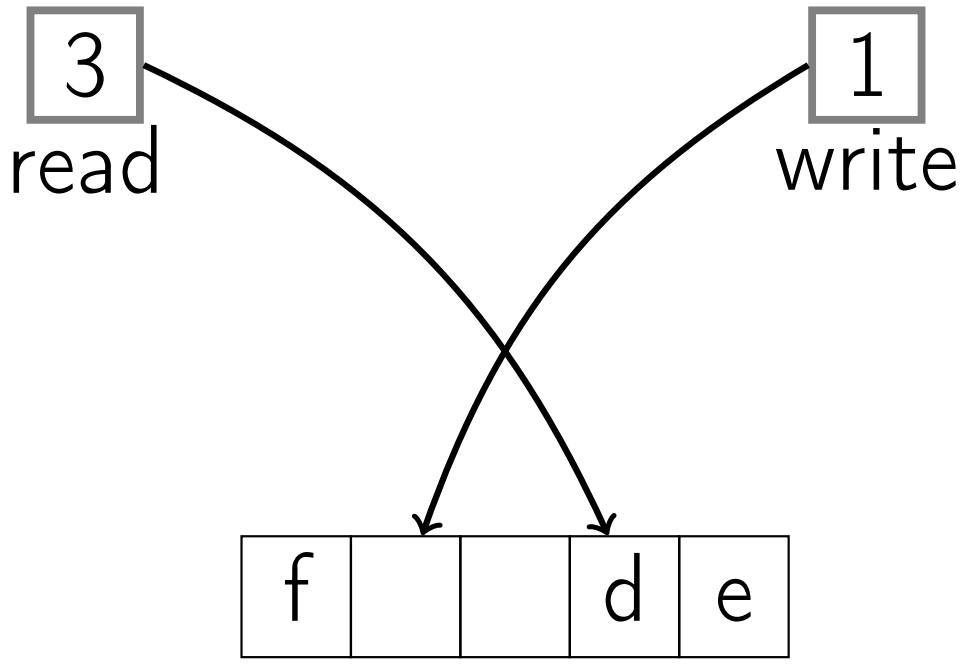
`Dequeue()`

Queue Implementation with Array



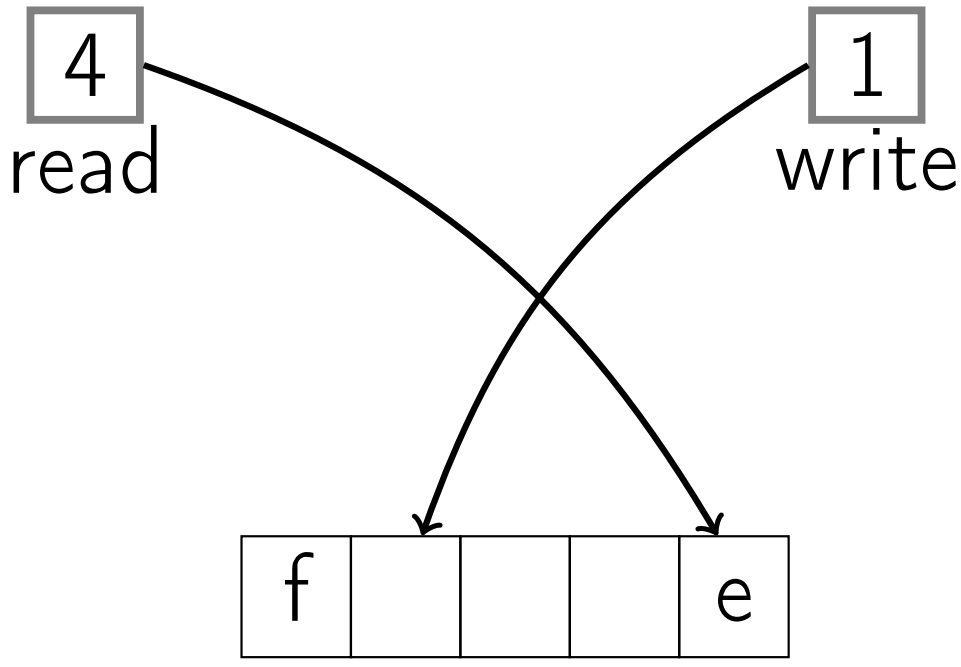
`Dequeue()` $\rightarrow$ `c`

Queue Implementation with Array



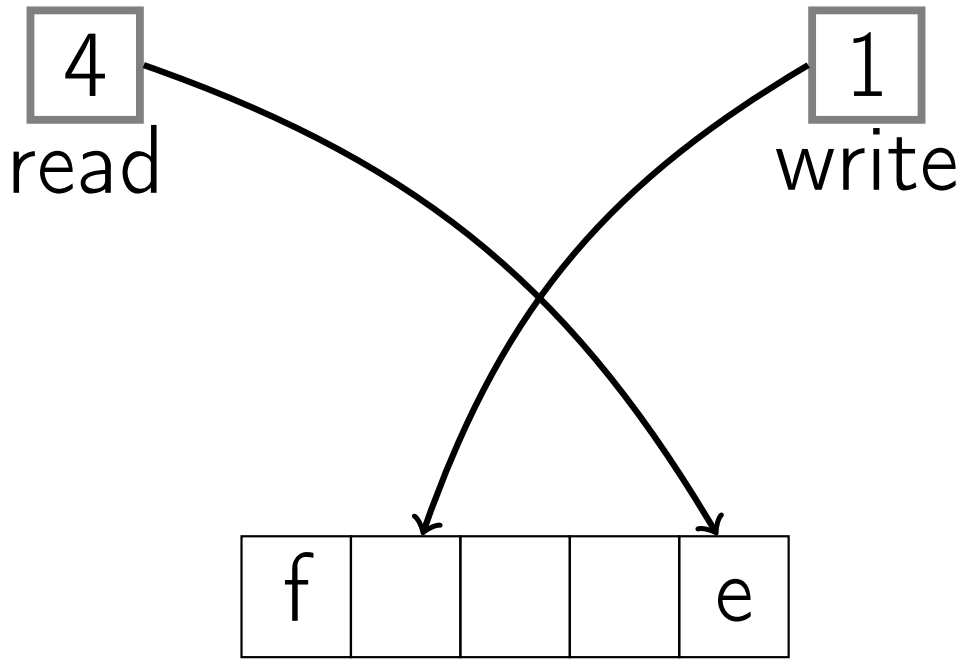
`Dequeue()`

Queue Implementation with Array



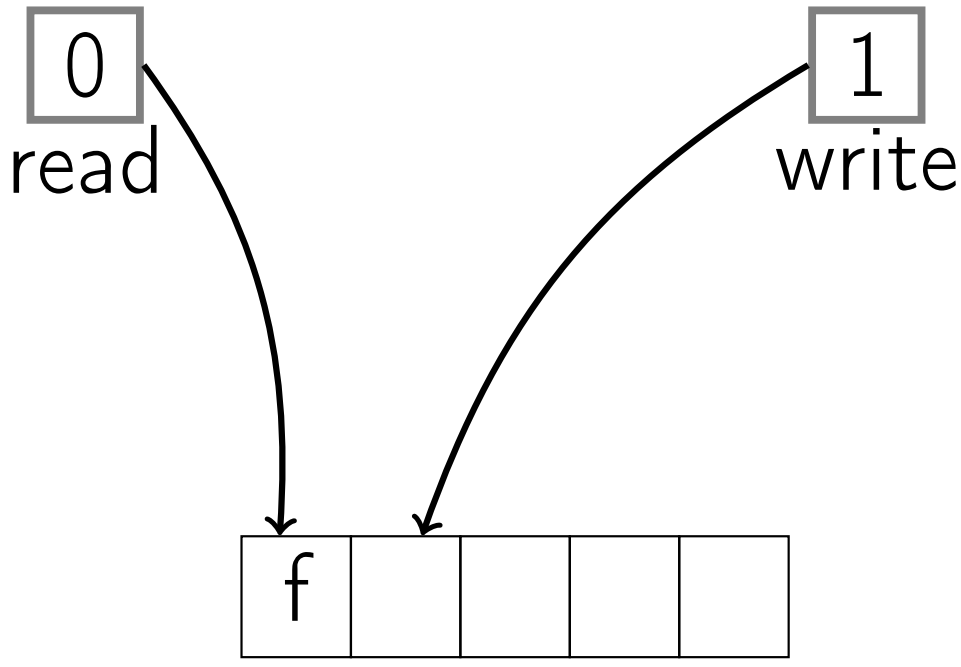
`Dequeue()` $\rightarrow$ `d`

Queue Implementation with Array



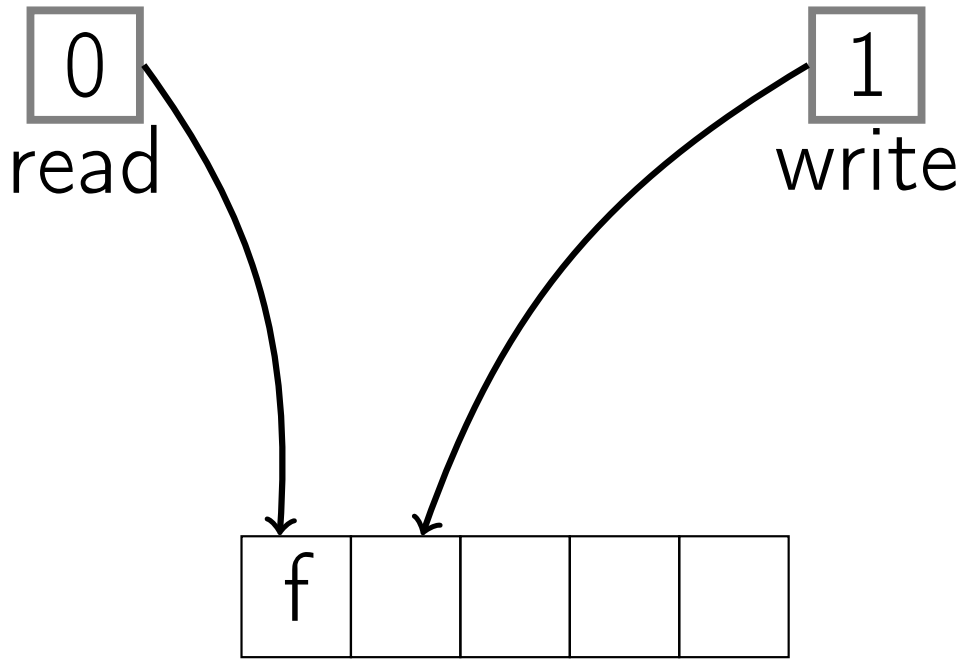
`Dequeue()`

Queue Implementation with Array



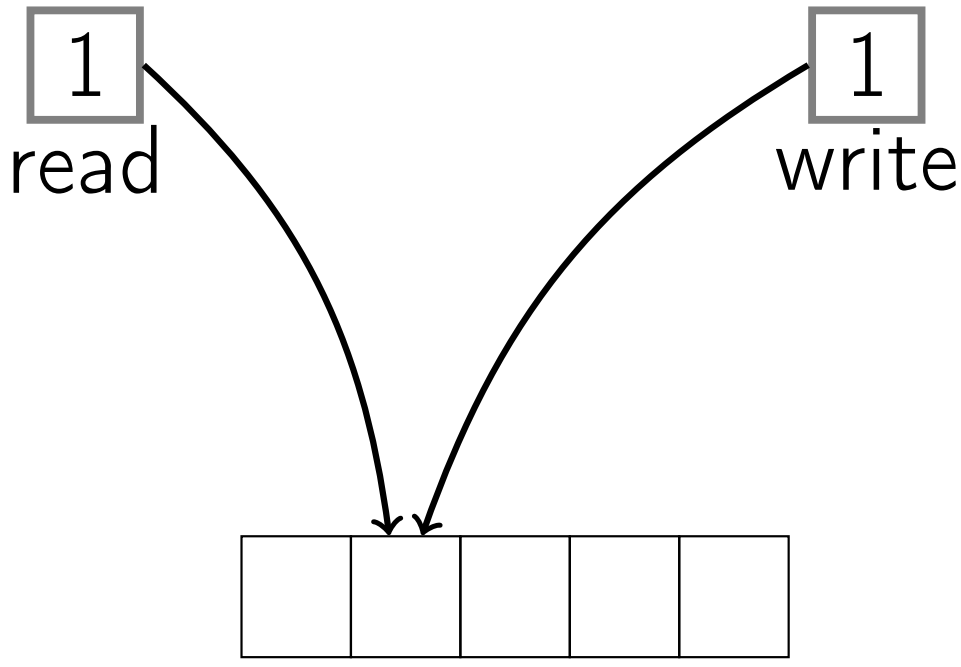
`Dequeue()` $\rightarrow$ `e`

Queue Implementation with Array



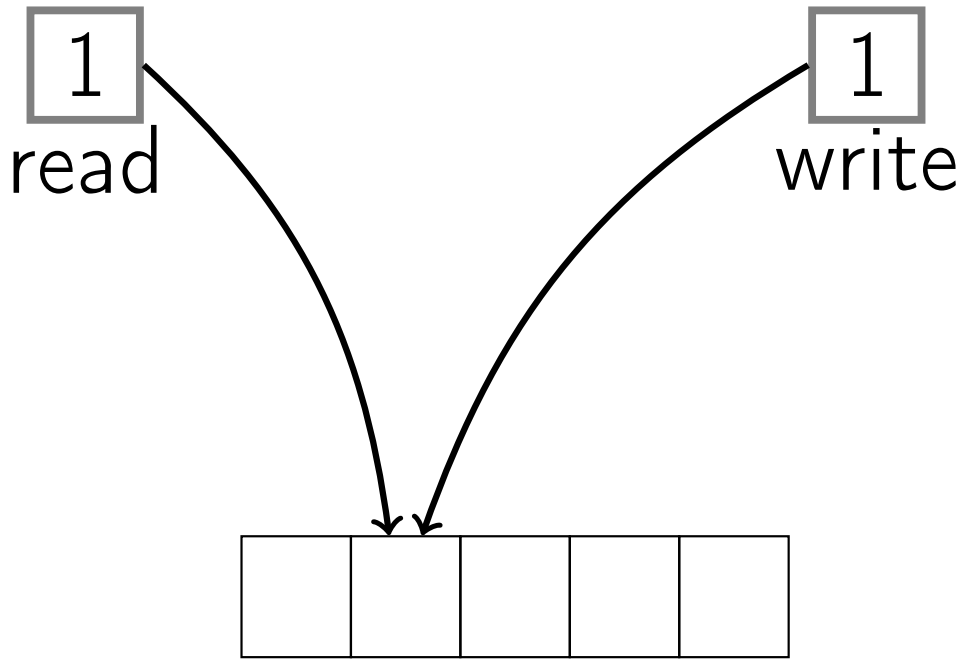
`Dequeue()`

Queue Implementation with Array



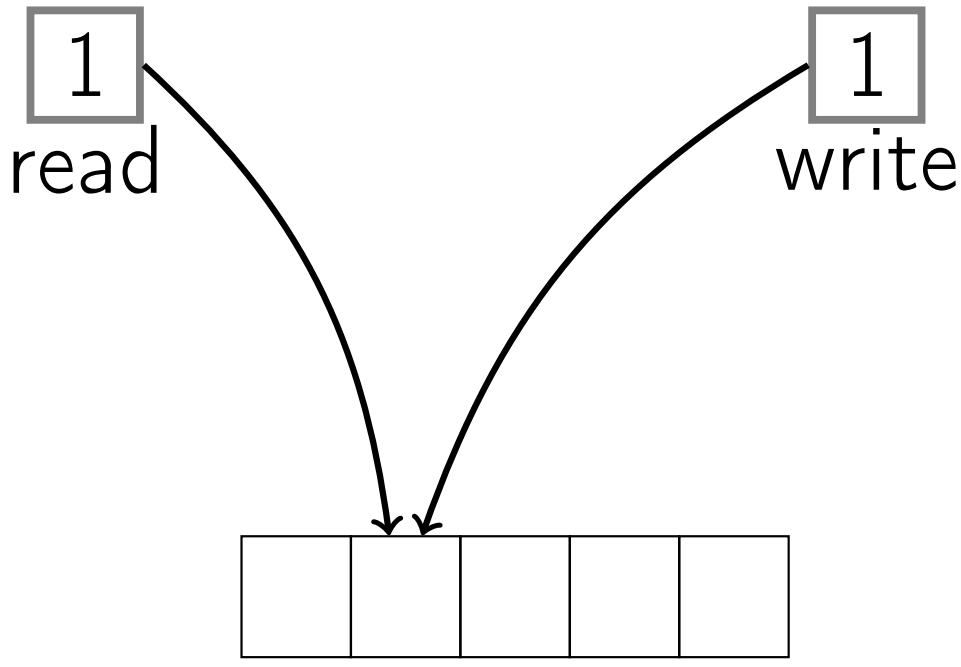
`Dequeue()` $\rightarrow$ `f`

Queue Implementation with Array



`Empty()`

Queue Implementation with Array



`Empty()` $\rightarrow$ `True`

Summary

- Queues can be implemented with either a linked list (with tail pointer) or an array.
- Each queue operation is $O(1)$: Enqueue, Dequeue, Empty.

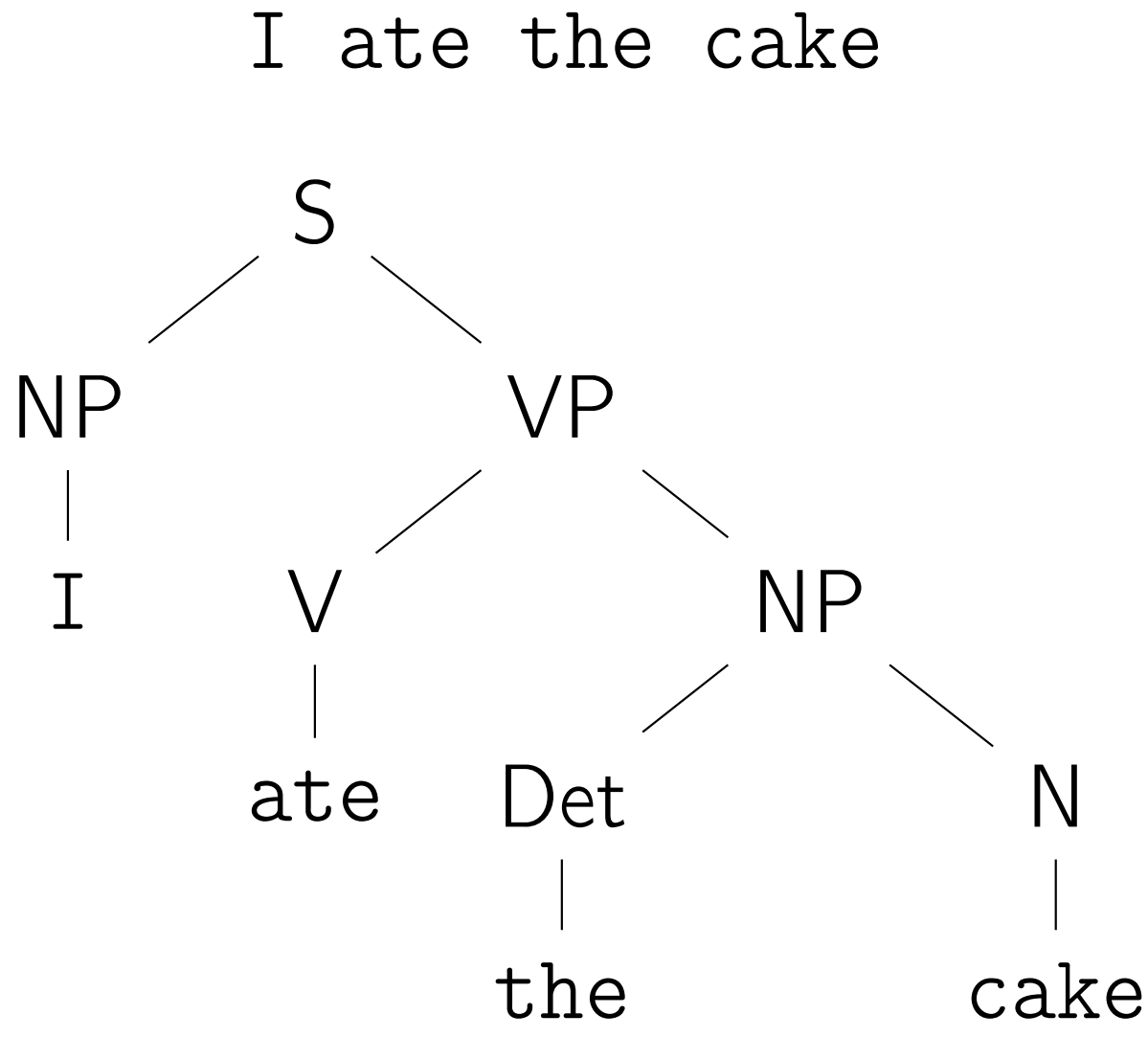
Basic Data Structures: Trees

Neil Rhodes

Department of Computer Science and Engineering
University of California, San Diego

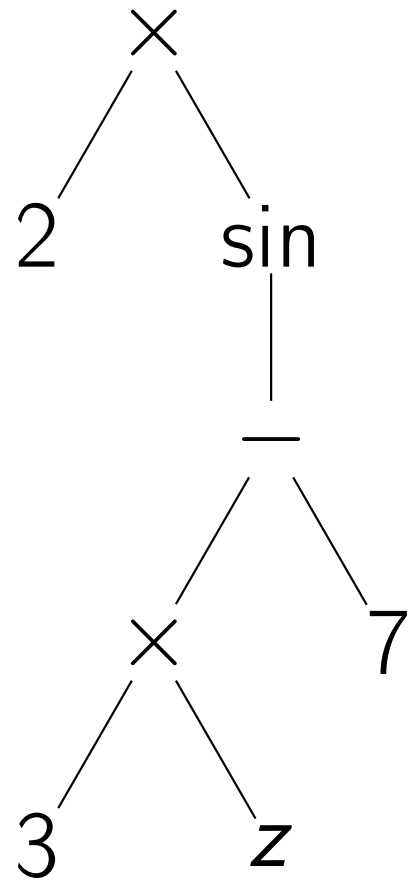
Data Structures
Data Structures and Algorithms

Syntax Tree for a Sentence

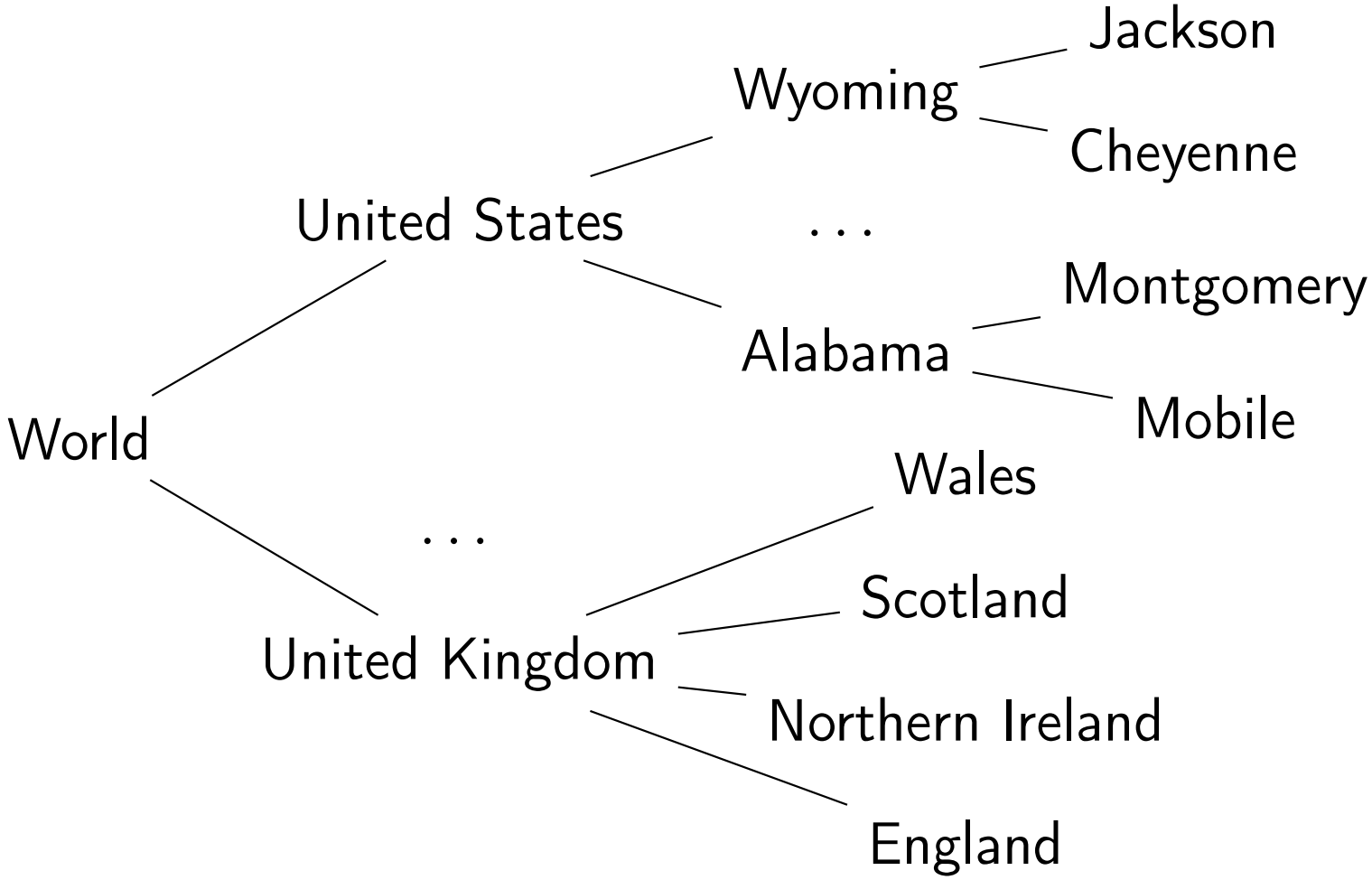


Syntax tree for an Expression

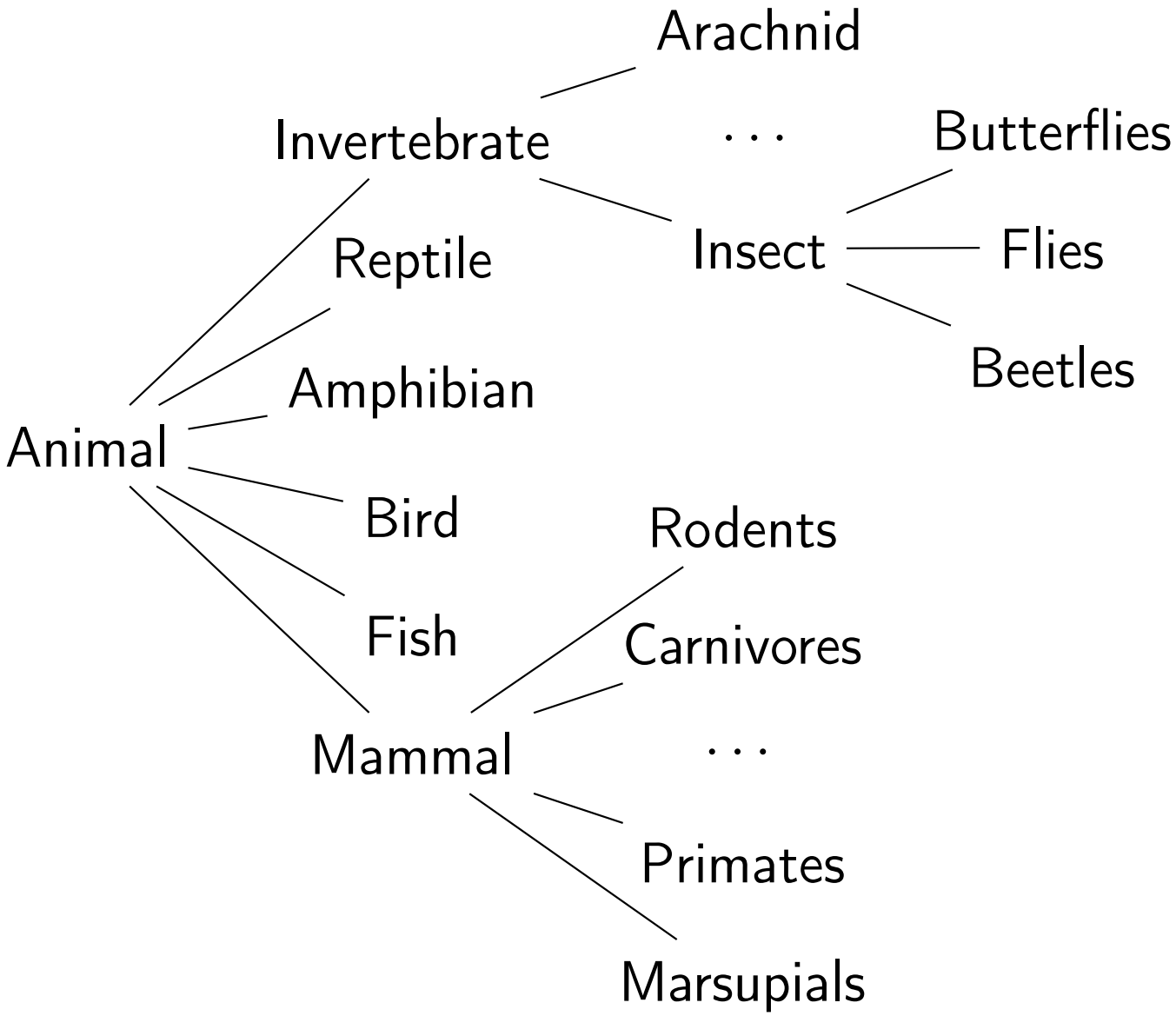
$$2\sin(3z - 7)$$



Geography Hierarchy

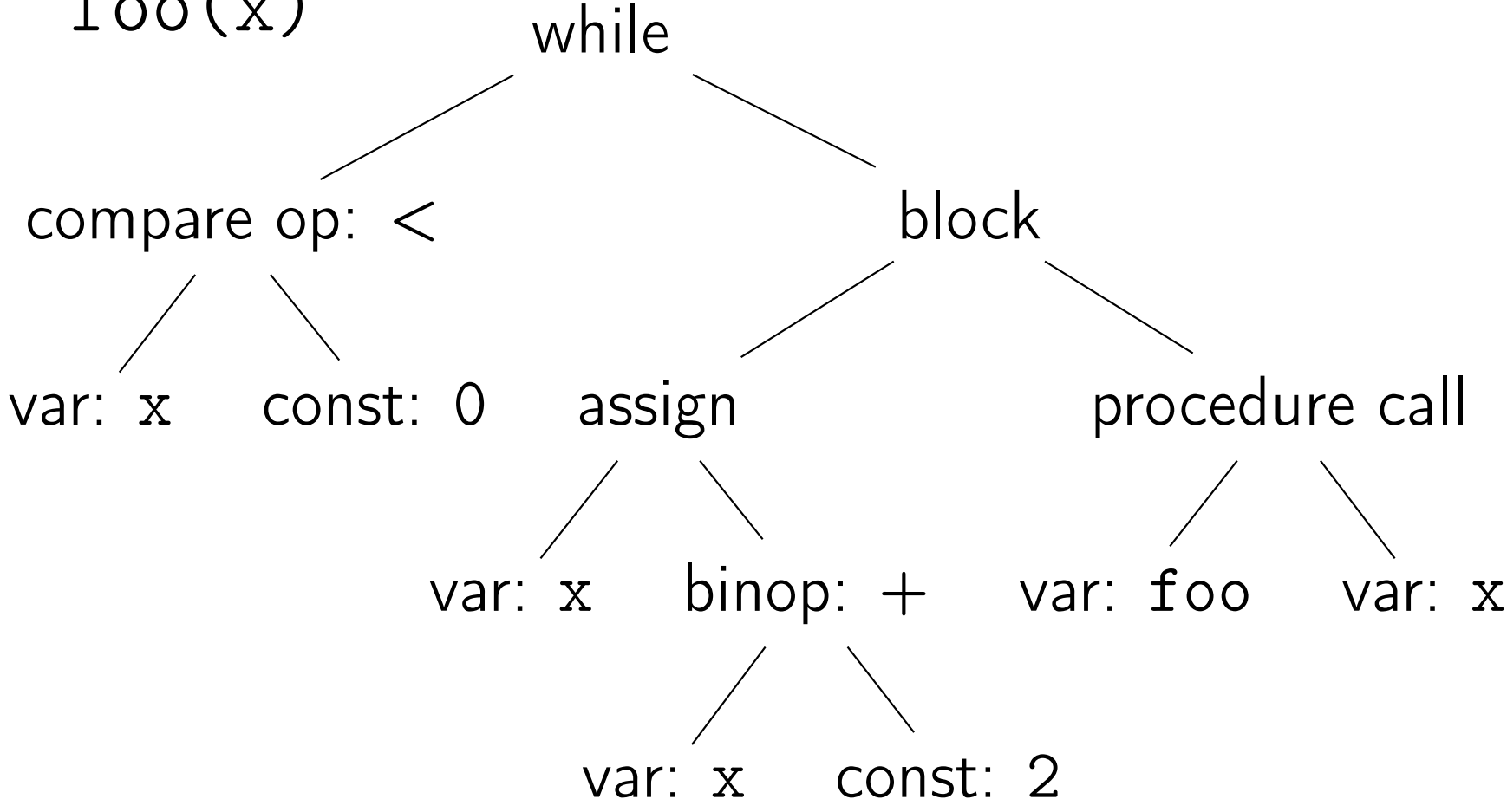


Animal Kingdom (partial)

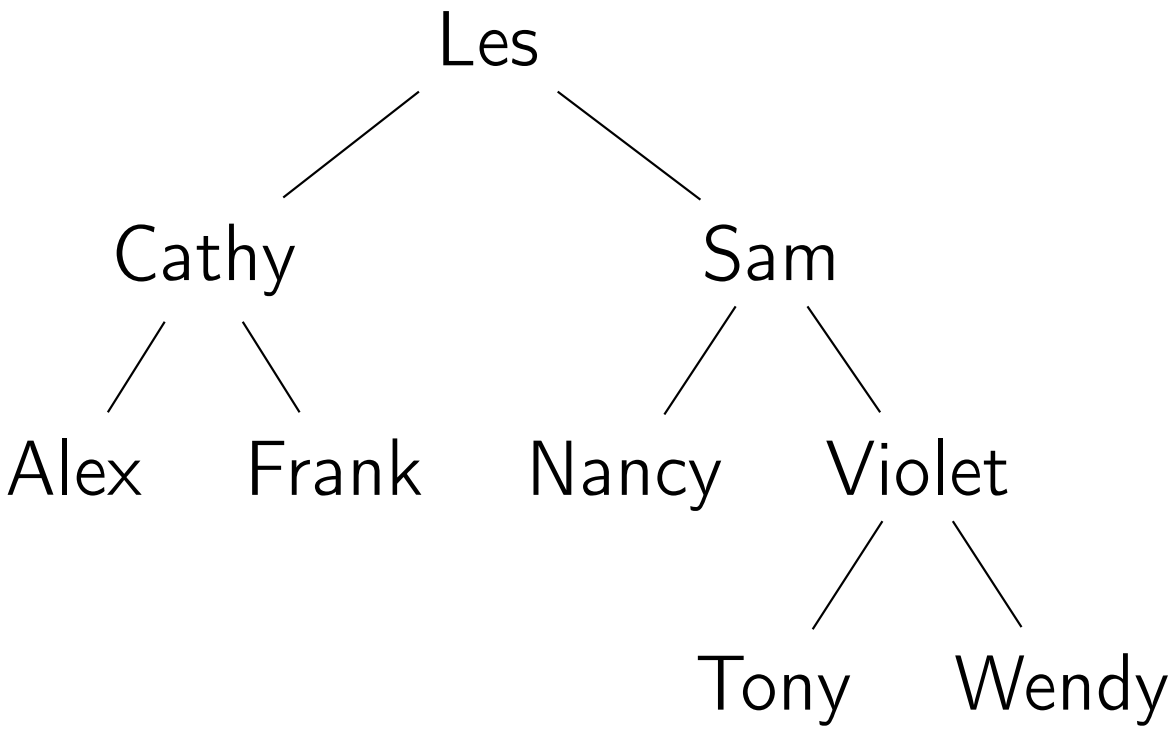


Abstract Syntax Tree for Code

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



Binary Search Tree



Definition

A Tree is:

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

Simple Tree

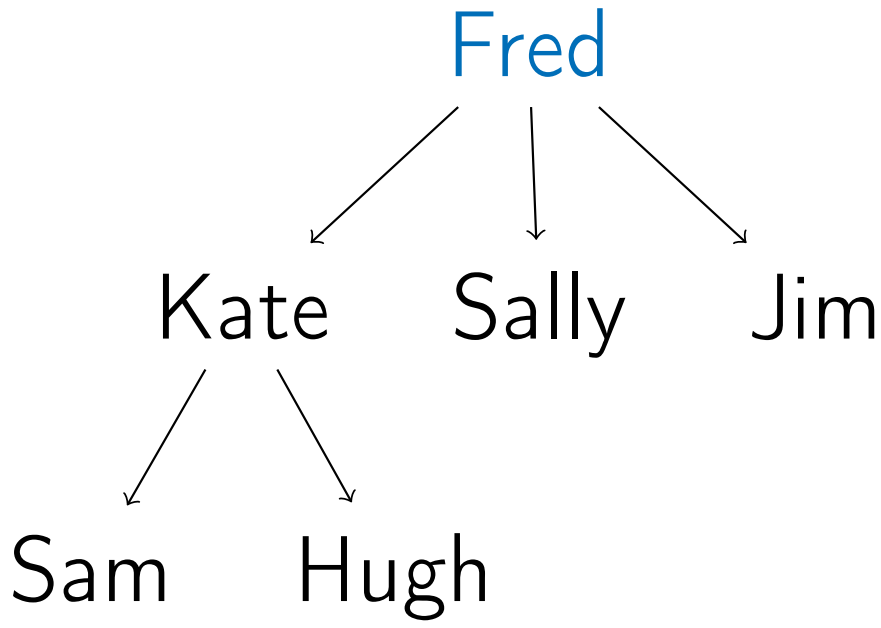
Empty tree:

Tree with one node:

Fred

Tree with two nodes:

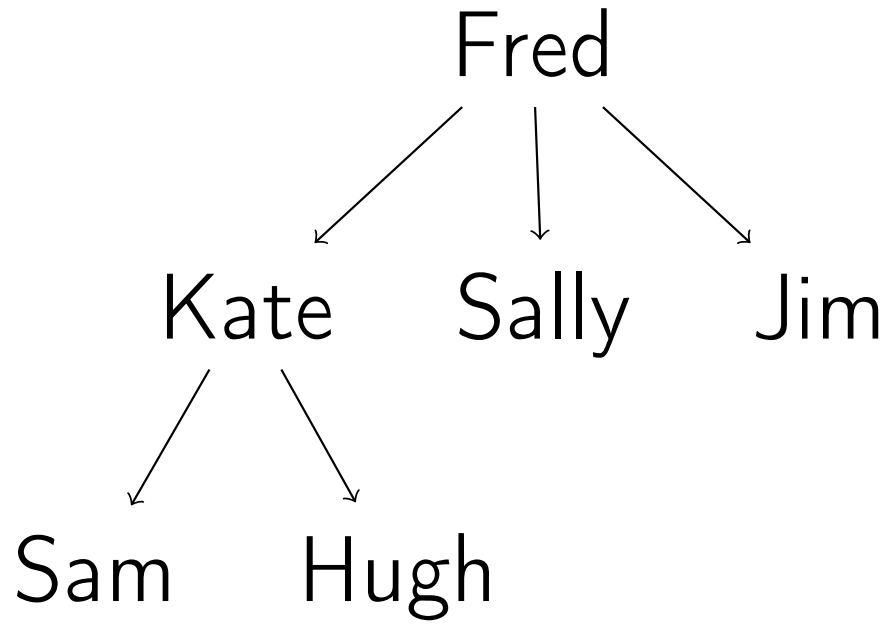
Terminology



***Root*:**

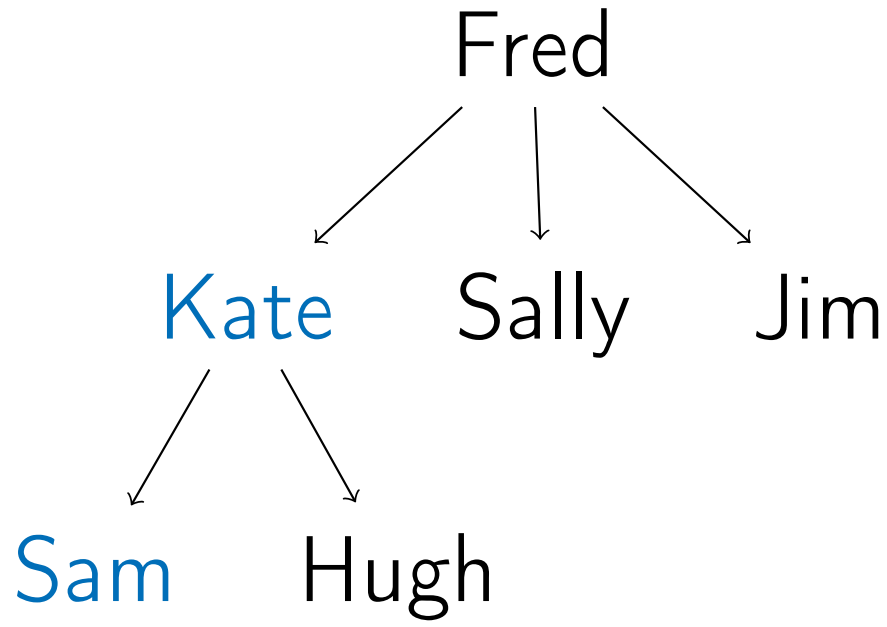
top node in the tree

Terminology



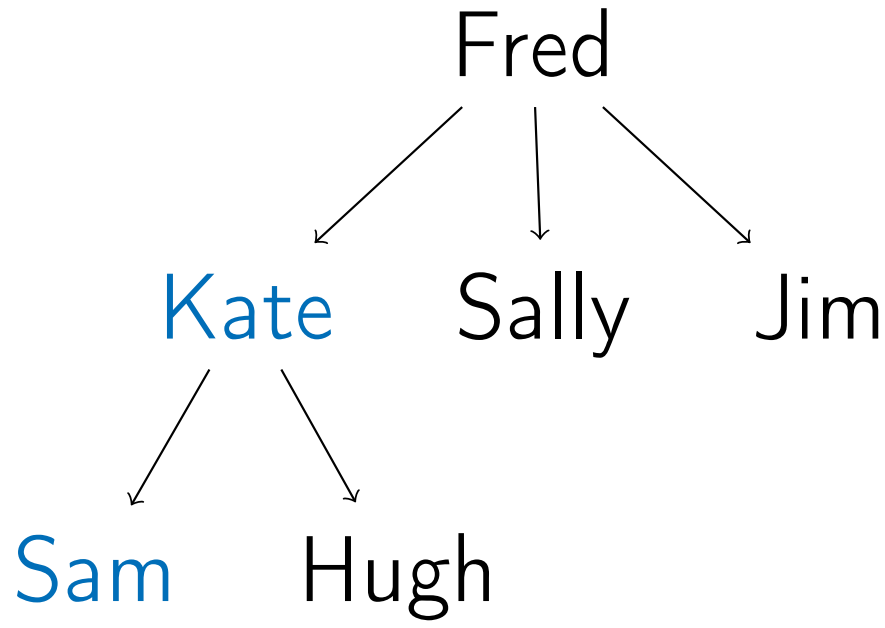
A *child* has a line down directly from a *parent*

Terminology



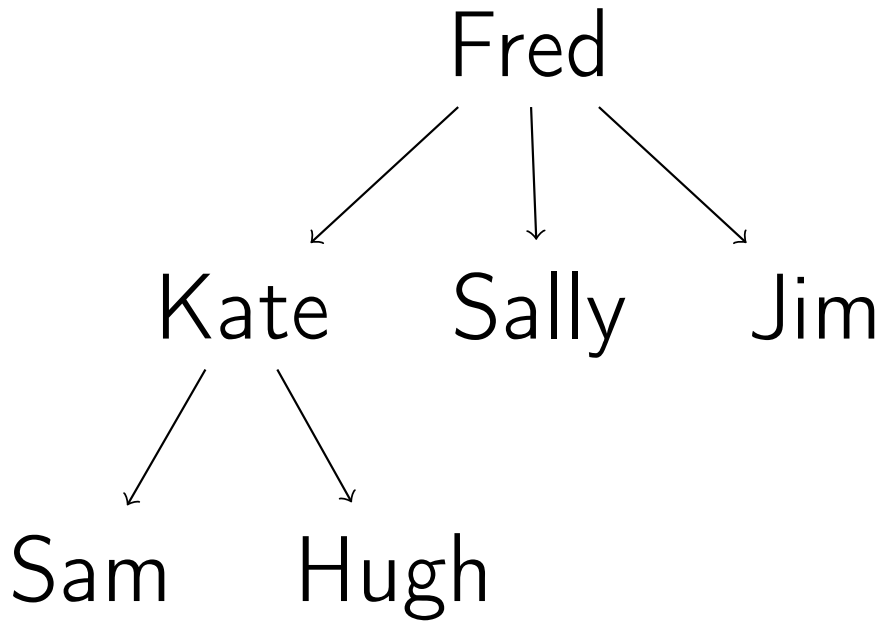
Kate is a *parent* of Sam

Terminology



Sam is a *child* of Kate

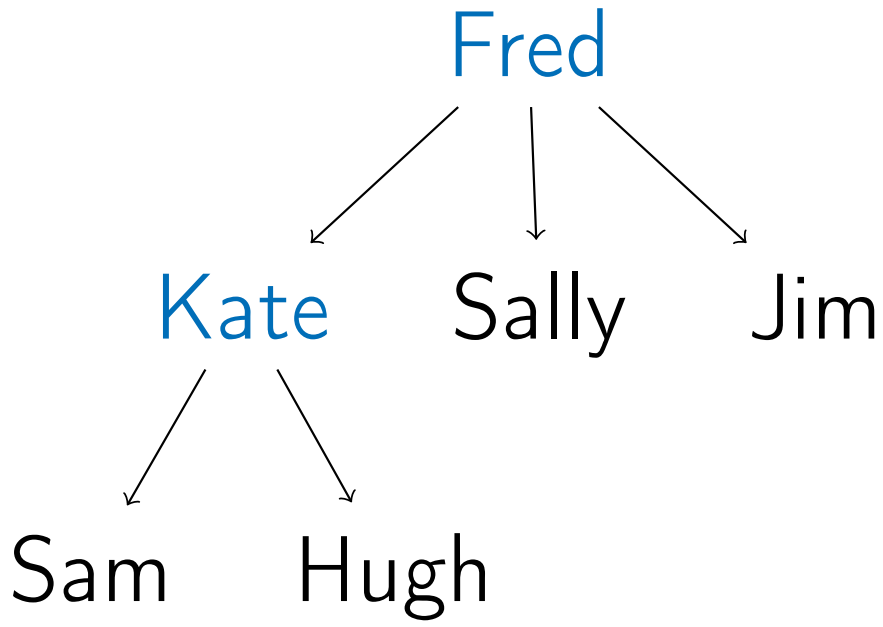
Terminology



Ancestor:

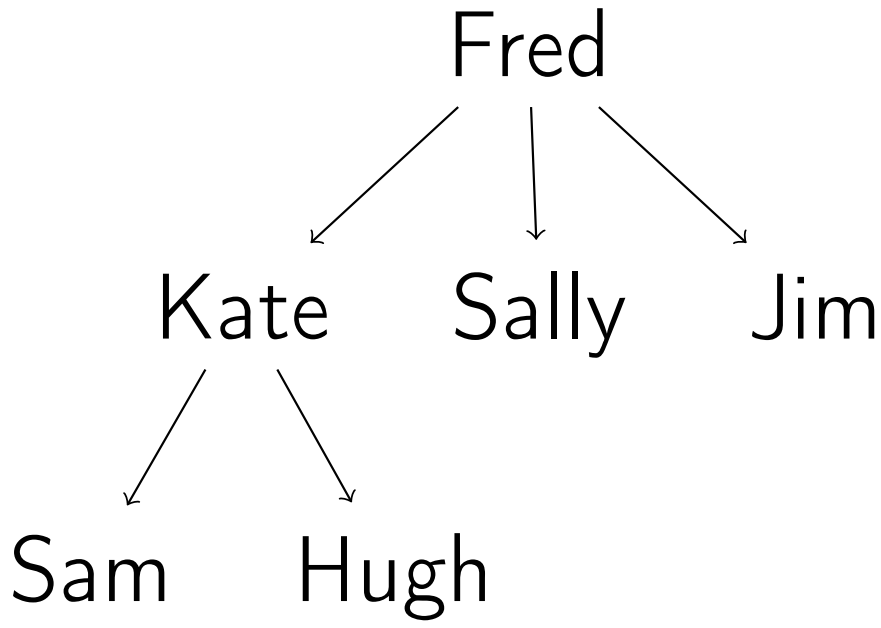
parent, or parent of parent, etc.

Terminology



***Ancestors* of Sam**

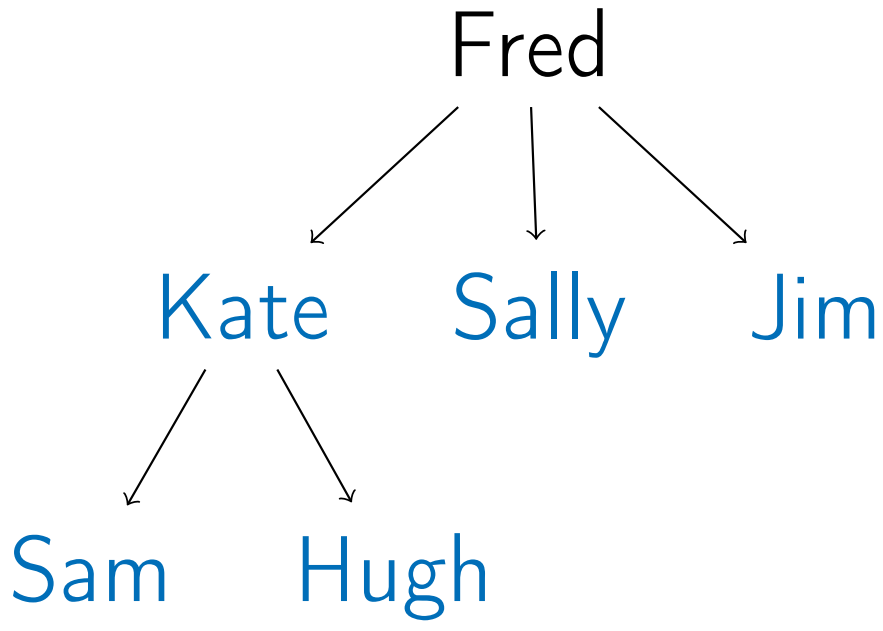
Terminology



Descendant:

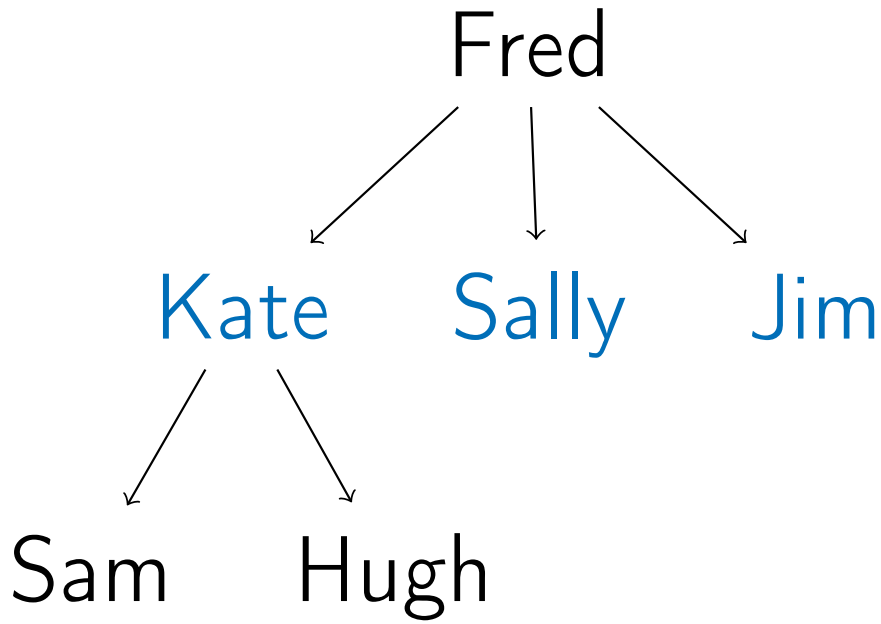
child, or child of child, etc.

Terminology



***Descendants* of Fred**

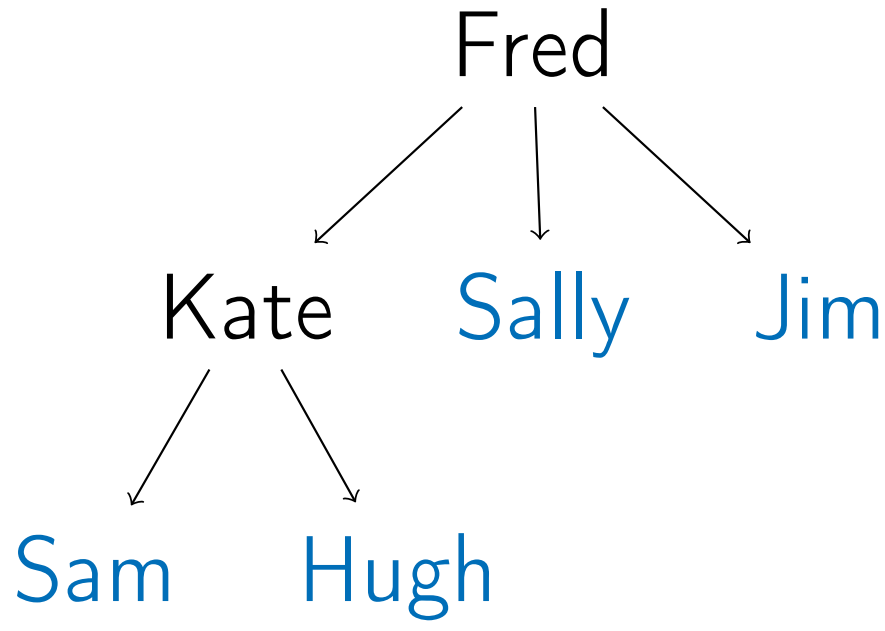
Terminology



***Sibling*:**

sharing the same parent

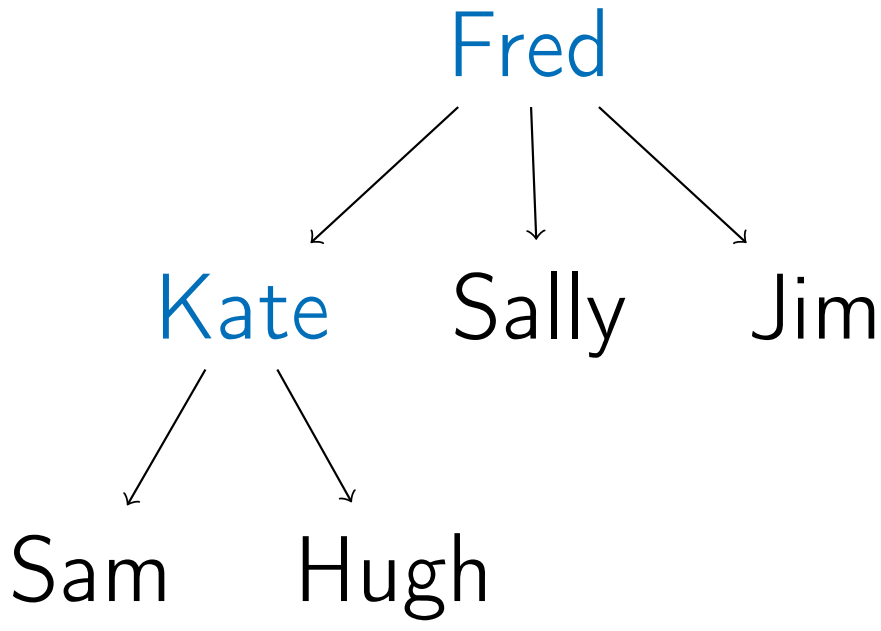
Terminology



***Leaf*:**

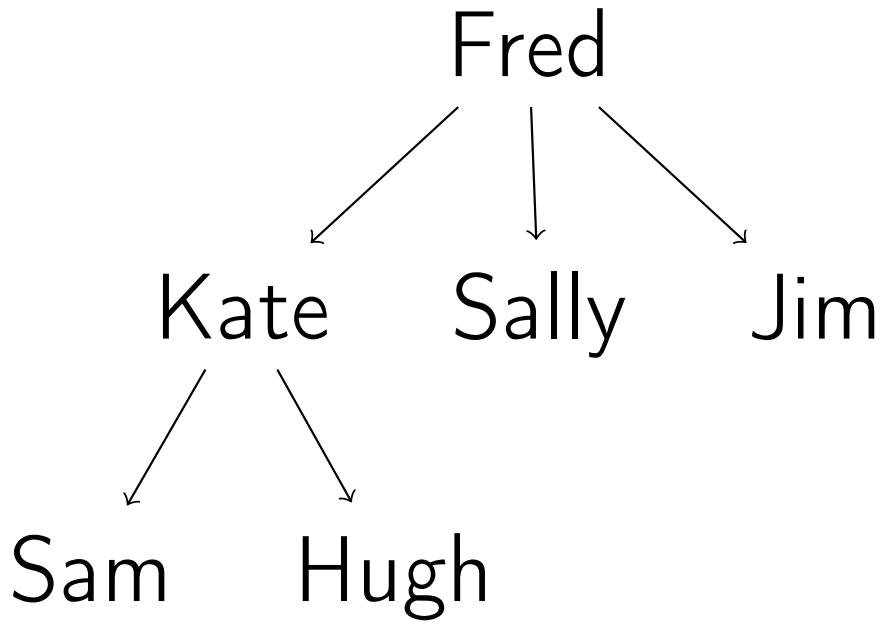
node with no children

Terminology



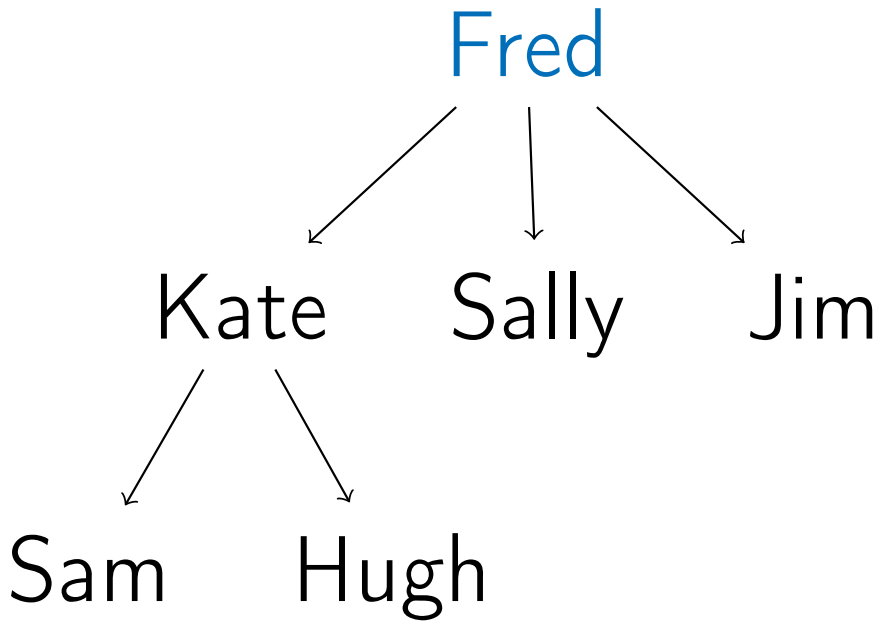
Interior node
(non-leaf)

Terminology



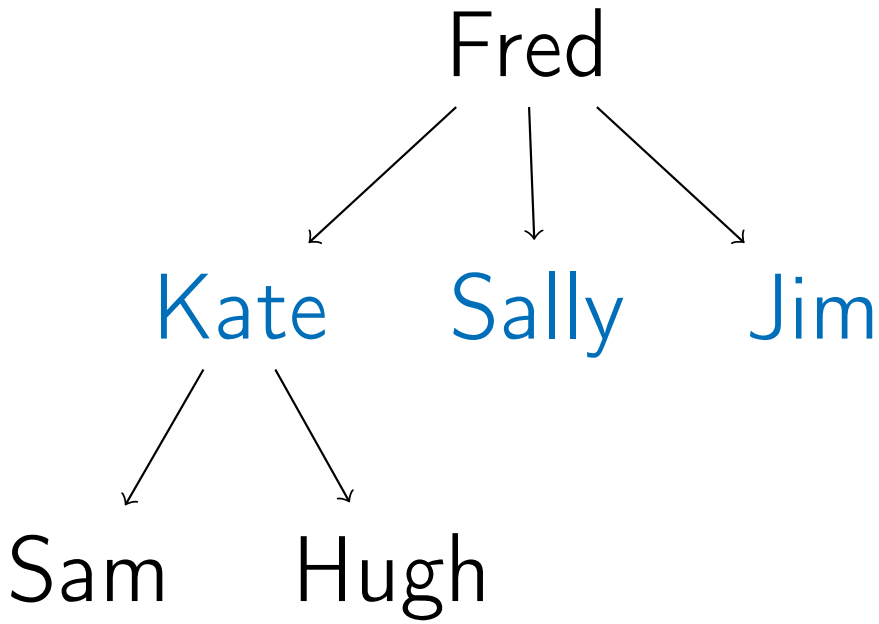
***Level*: 1+ num edges between root and node**

Terminology



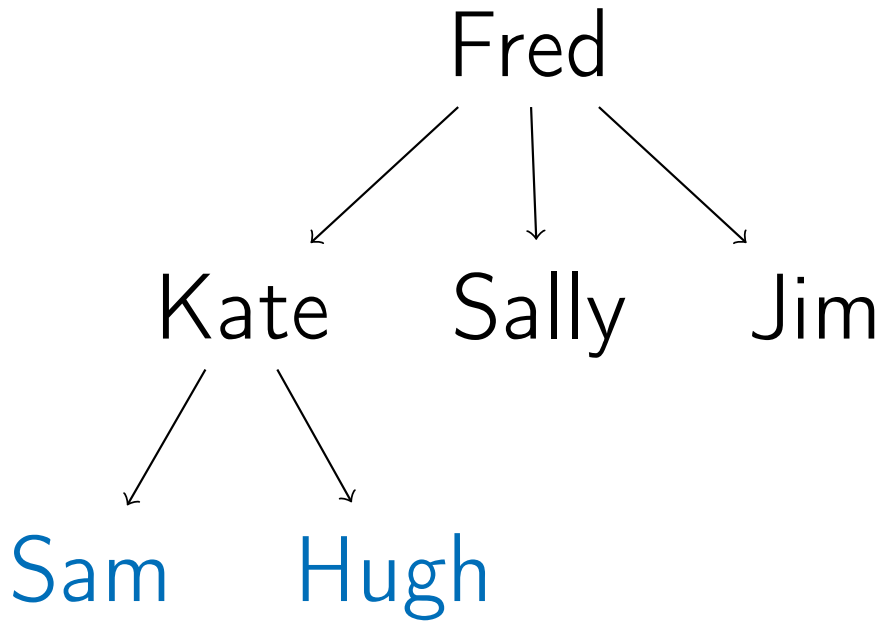
Level 1

Terminology



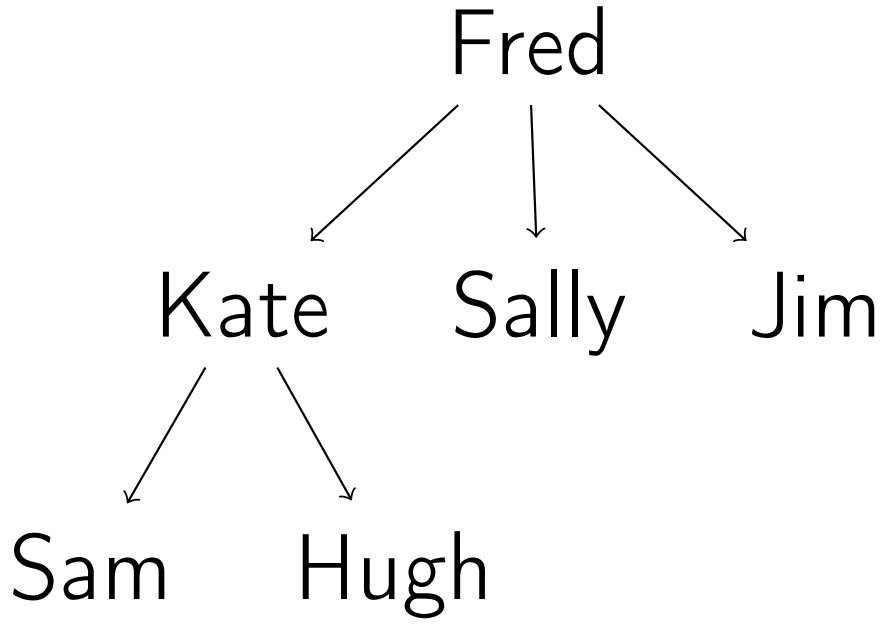
Level 2

Terminology



Level 3

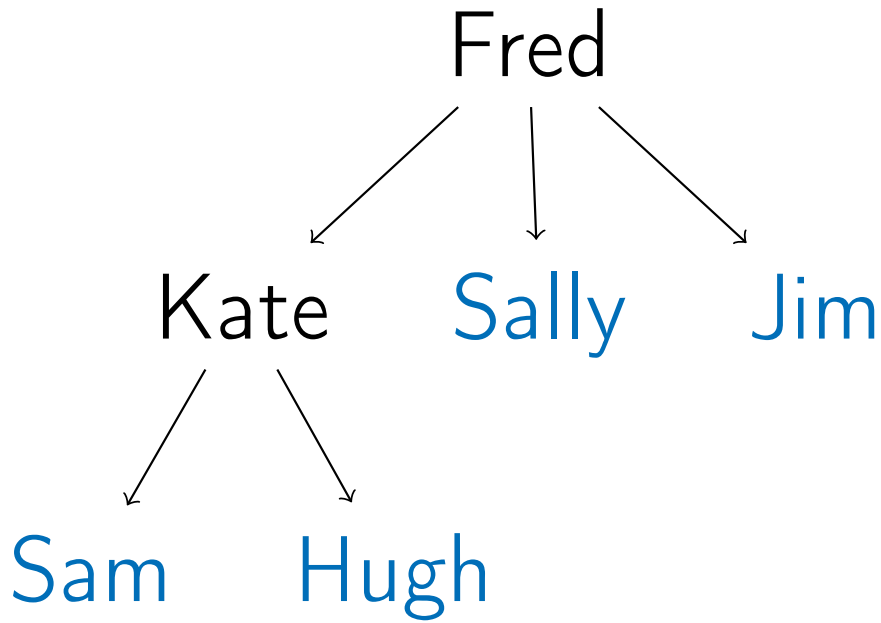
Terminology



***Height*: maximum depth of subtree**

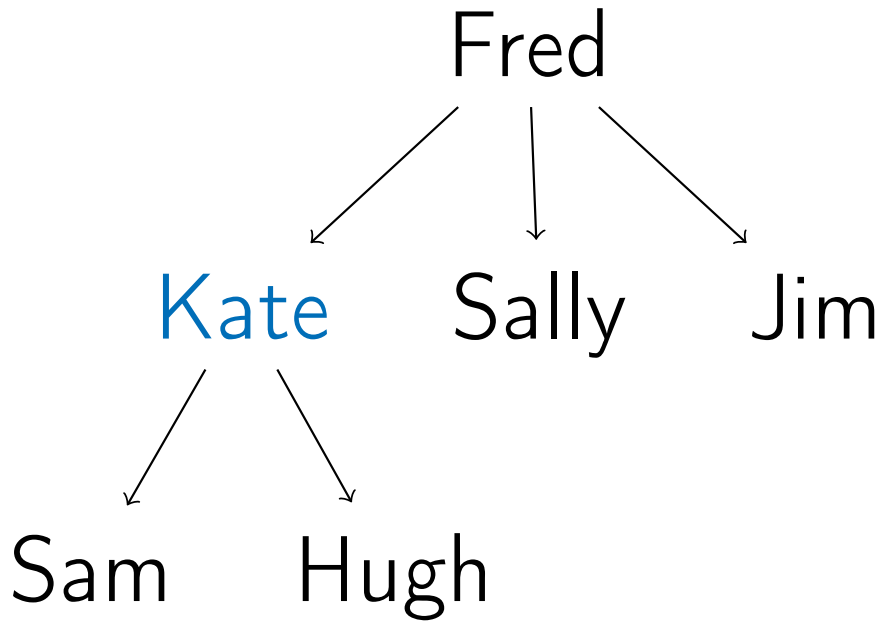
node and farthest leaf

Terminology



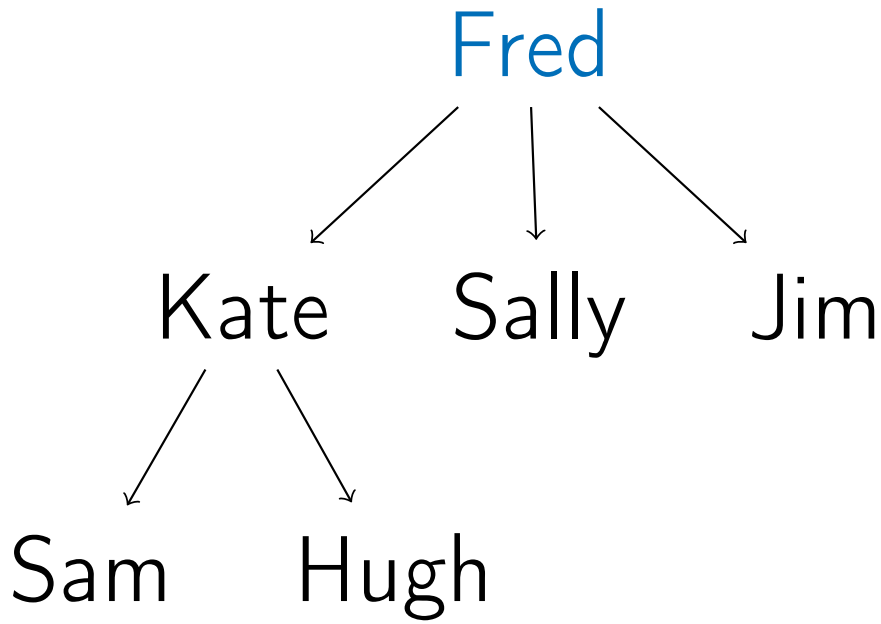
Height 1

Terminology



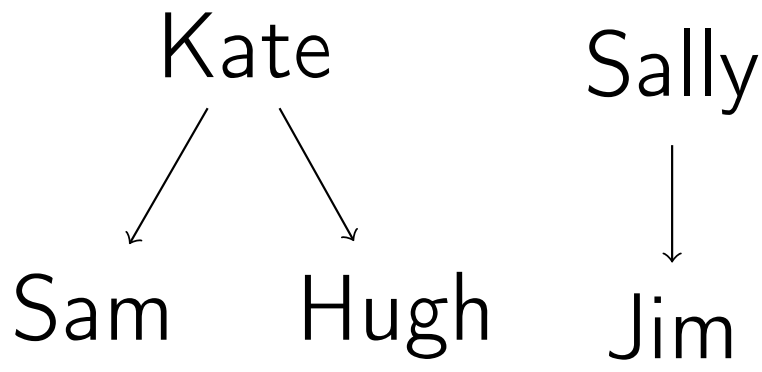
Height 2

Terminology



Height 3

Terminology



***Forest*:**
collection of trees

Node contains:

- `key`
- `children`: list of children nodes
- (optional) `parent`

For binary tree, node contains:

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

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)
```

Walking a Tree

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

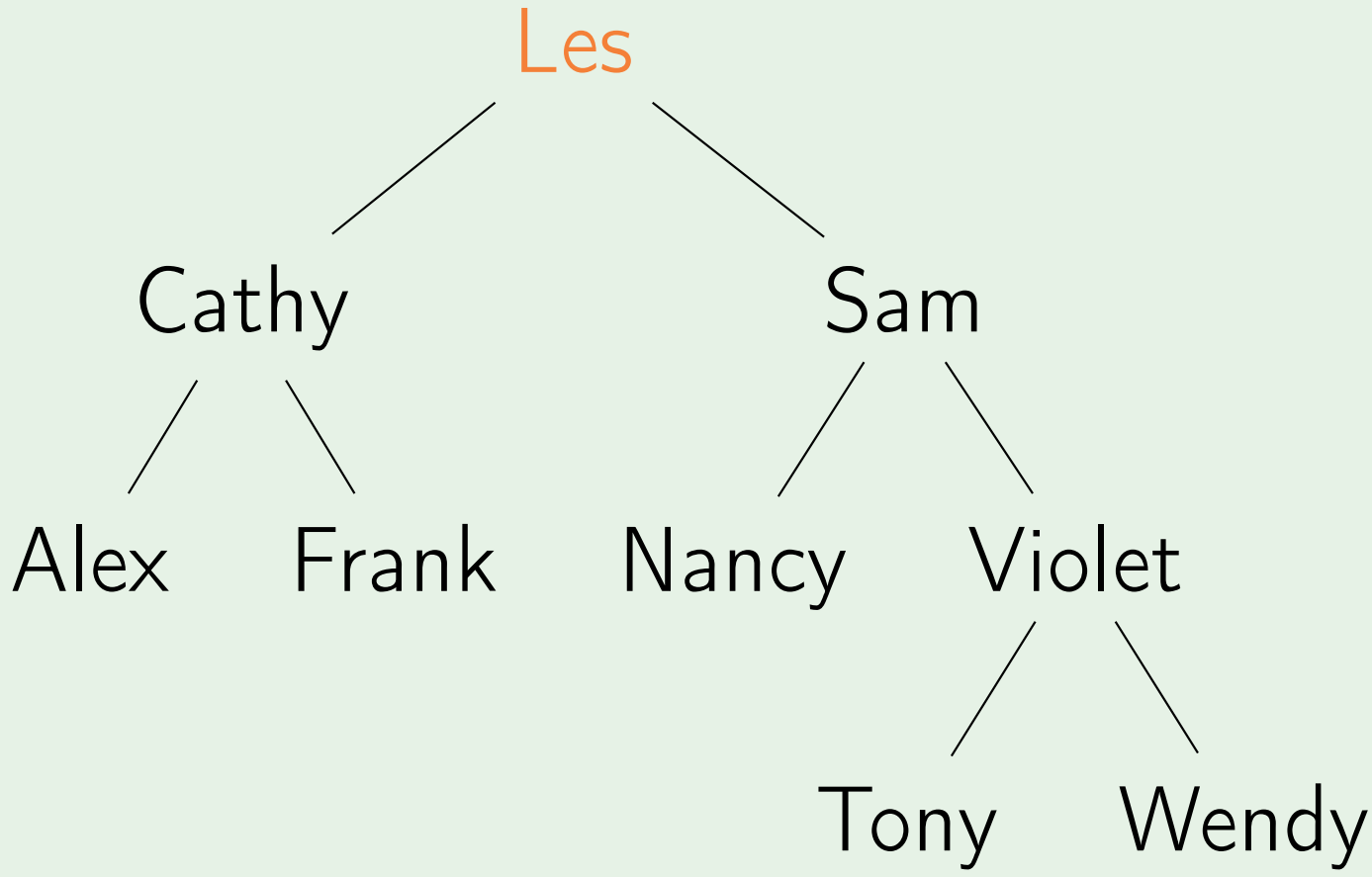
For example, print the nodes of the tree.

- 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

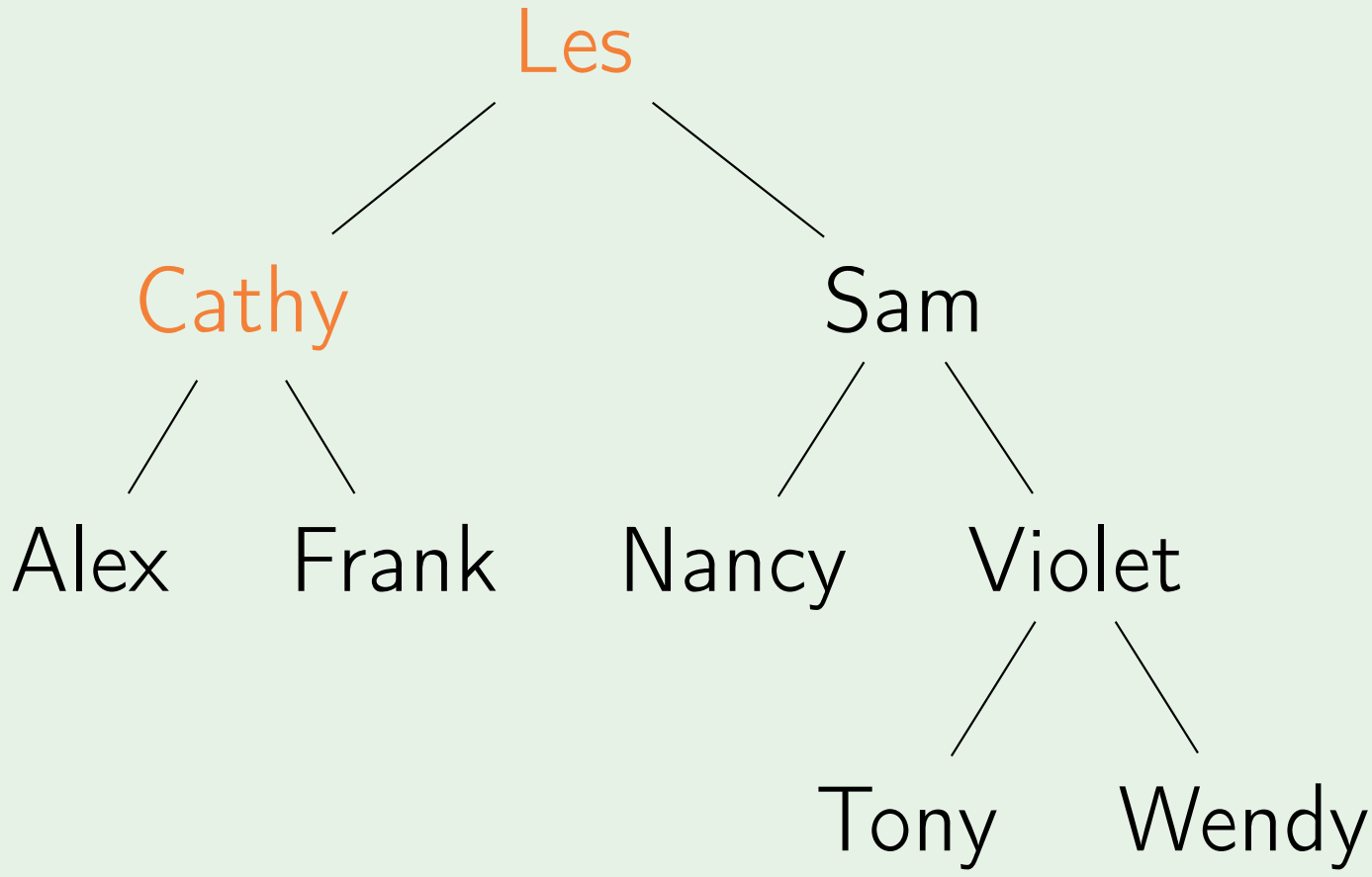
```
InOrderTraversal(tree)
if tree = nil:
    return
InOrderTraversal(tree.left)
Print(tree.key)
InOrderTraversal(tree.right)
```

InOrderTraversal



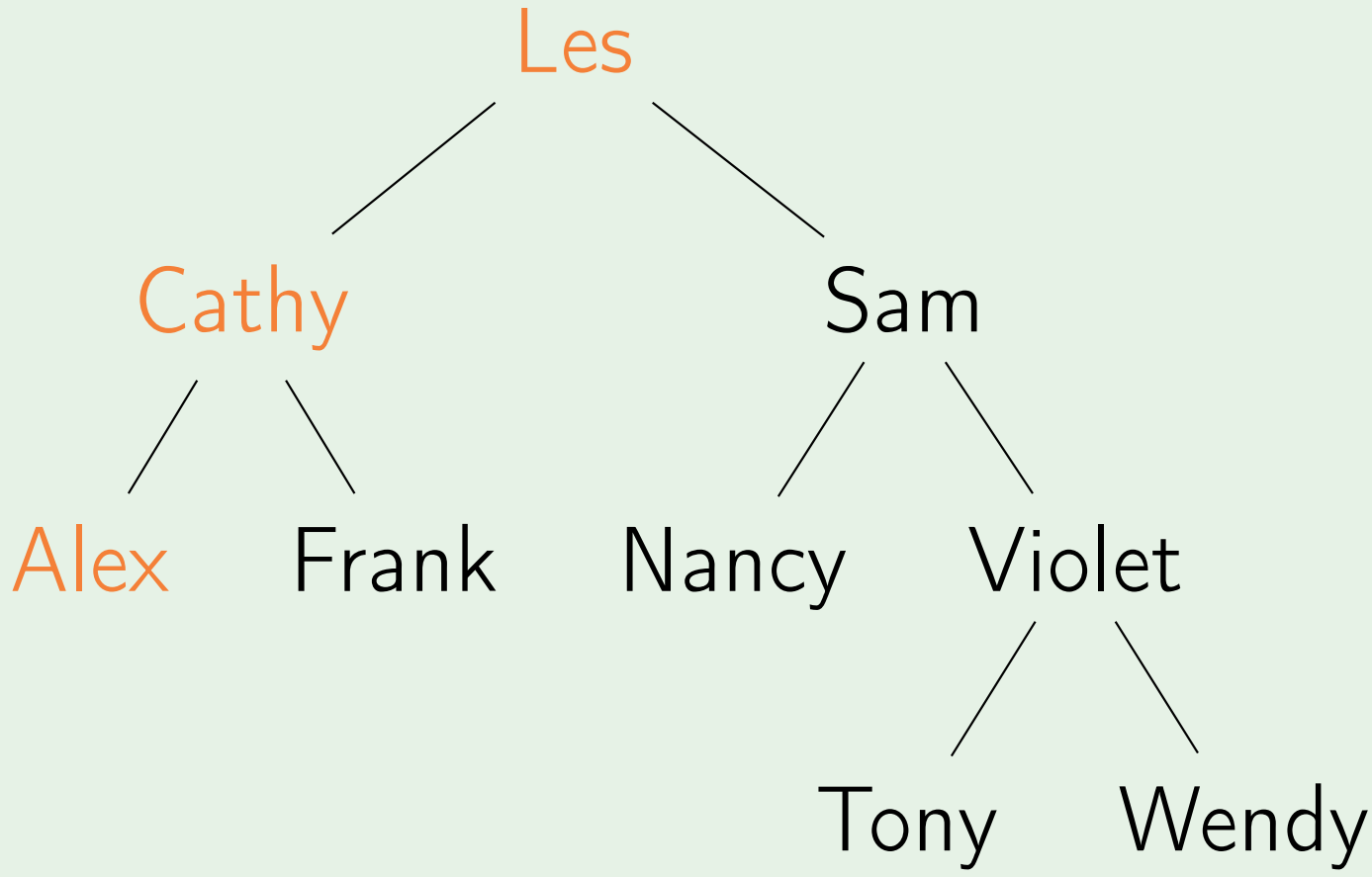
Output:

InOrderTraversal



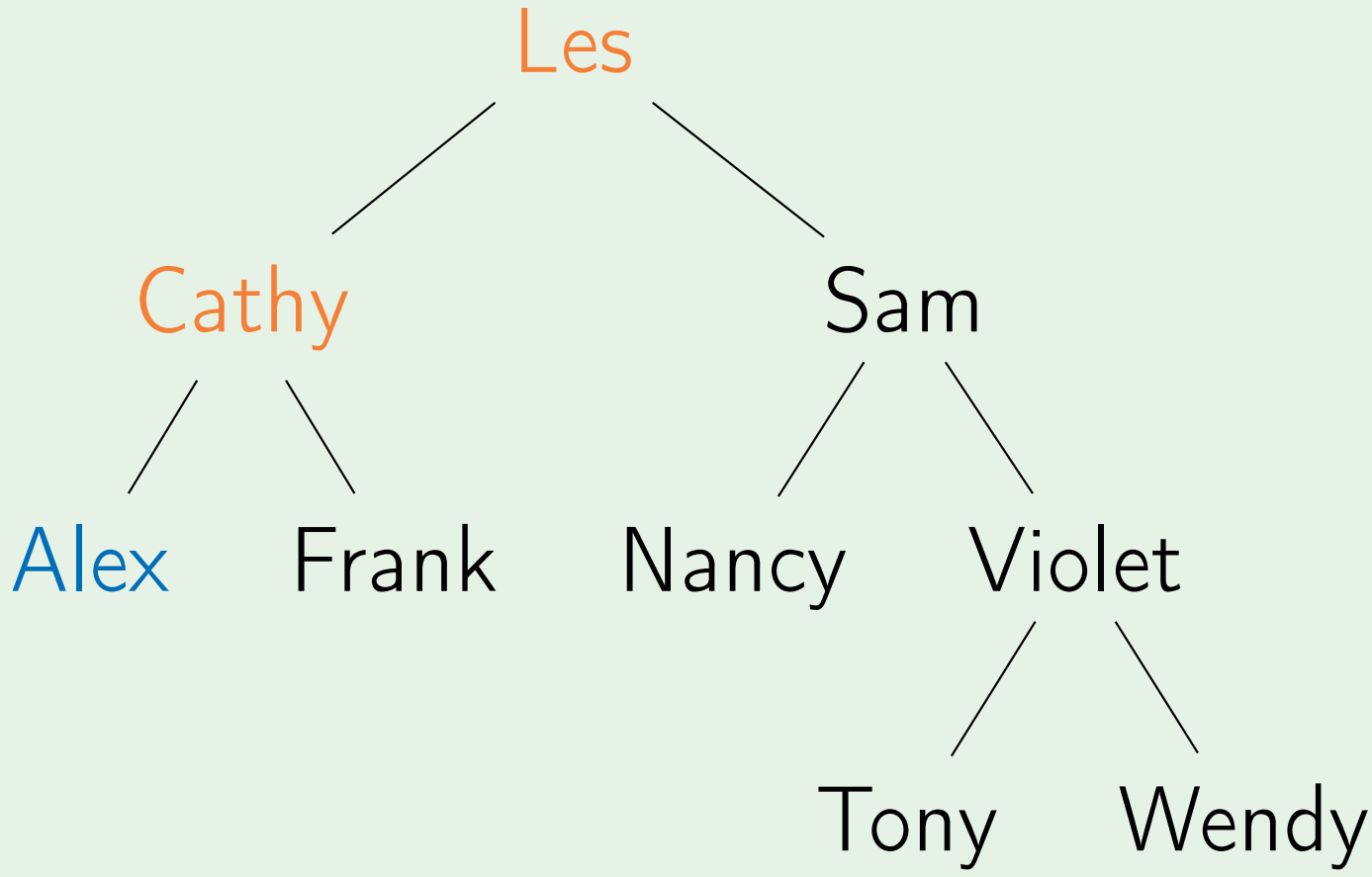
Output:

InOrderTraversal



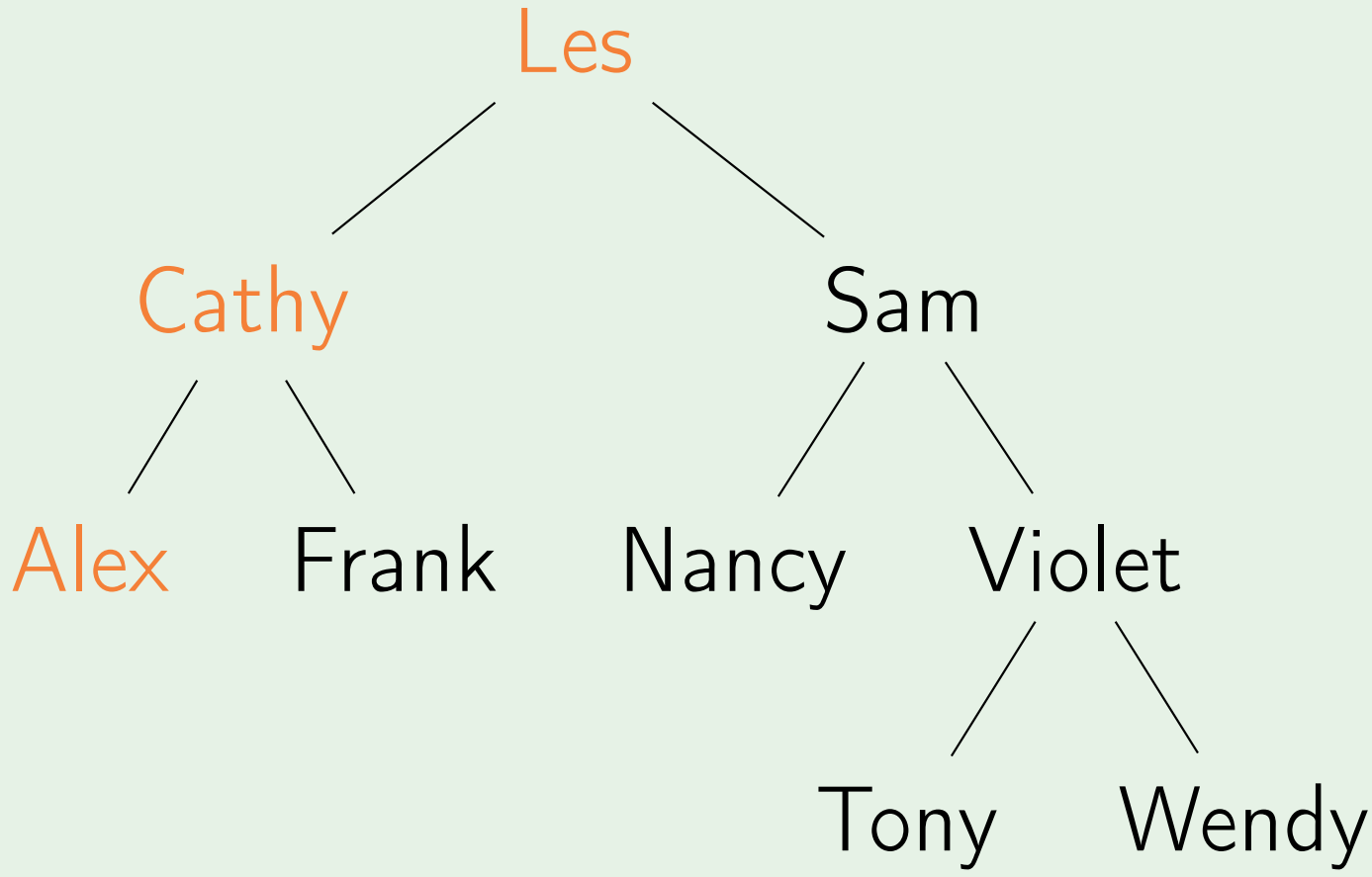
Output:

InOrderTraversal



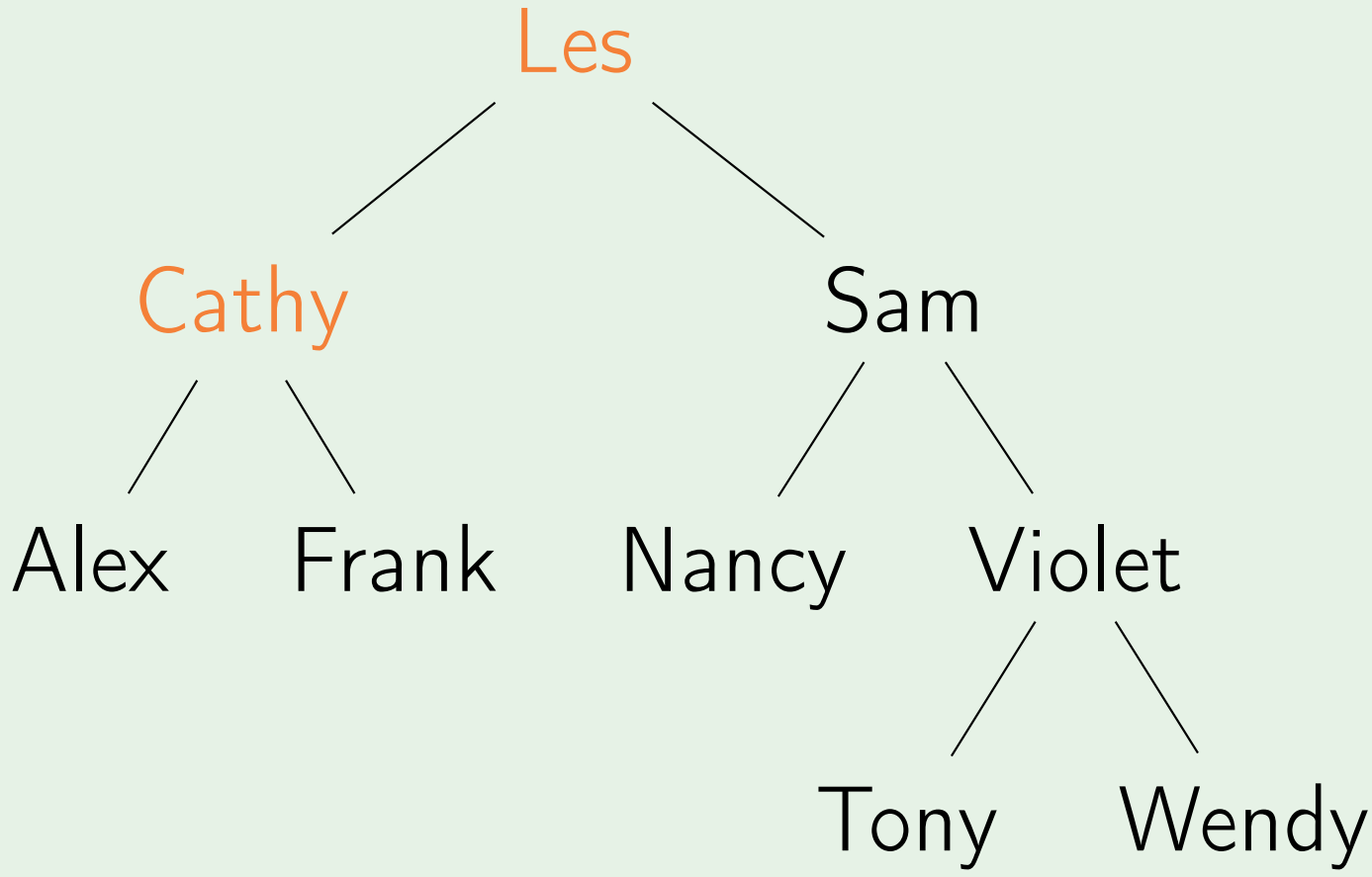
Output: Alex

InOrderTraversal



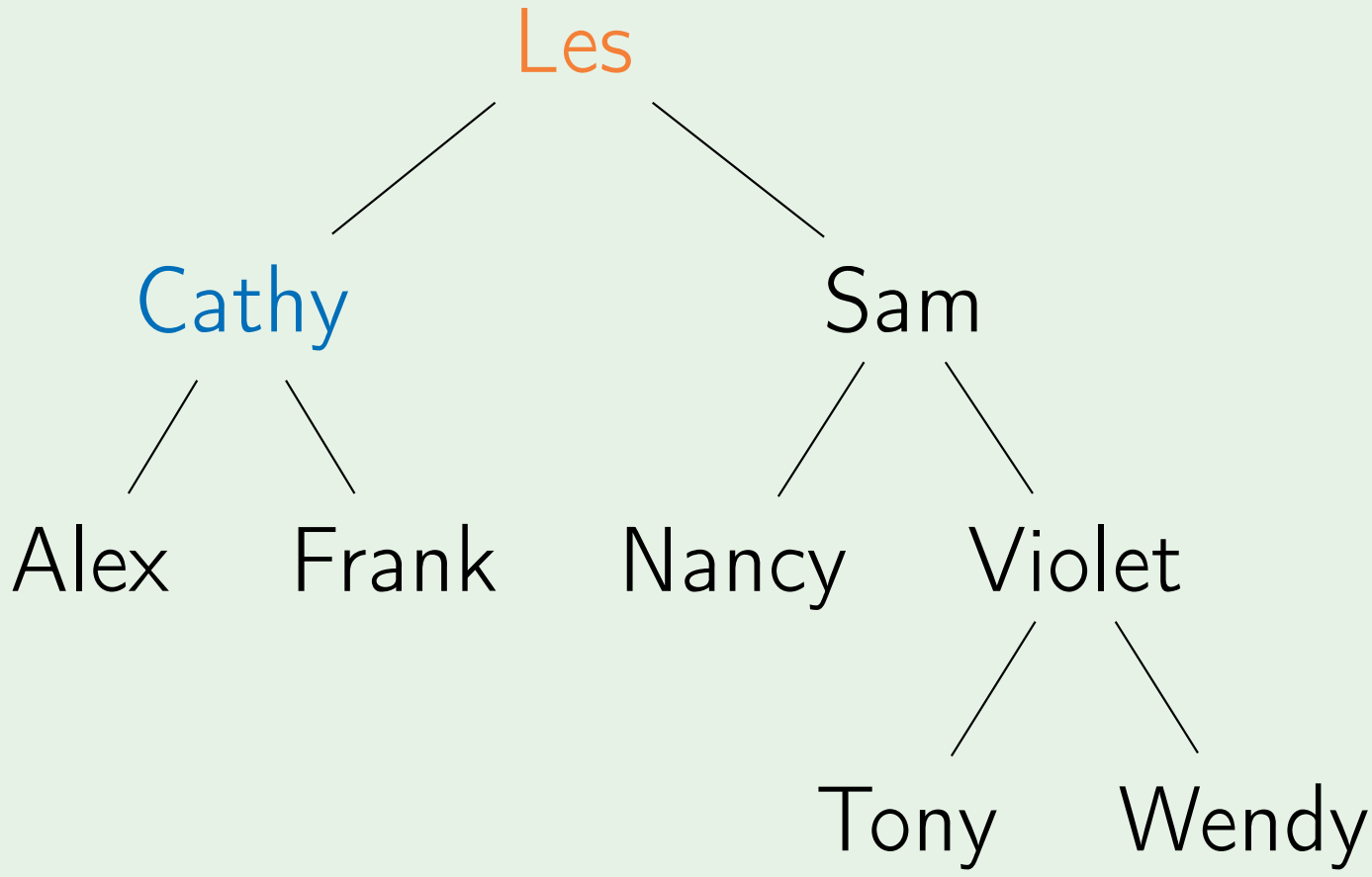
Output: Alex

InOrderTraversal



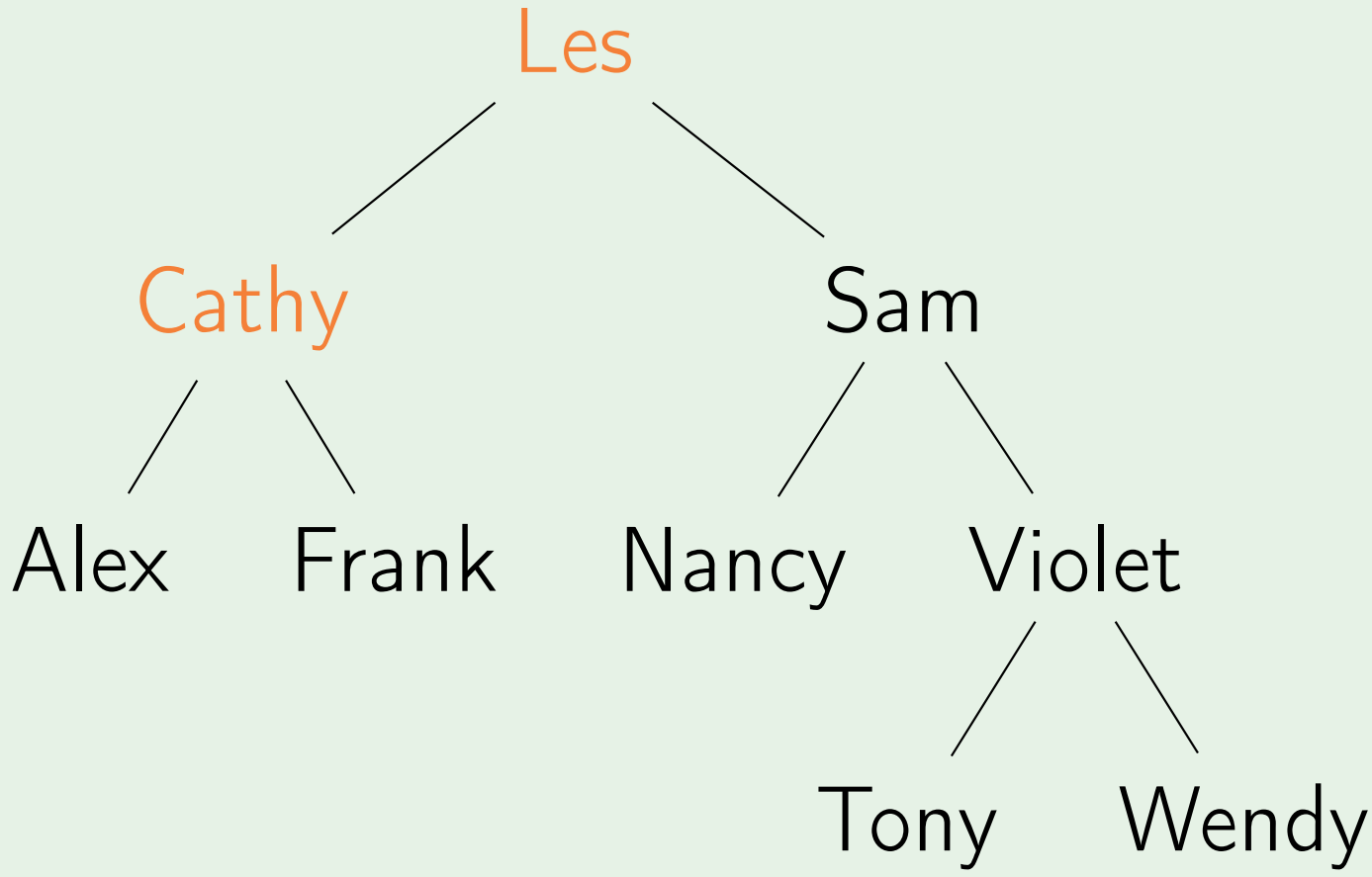
Output: Alex

InOrderTraversal



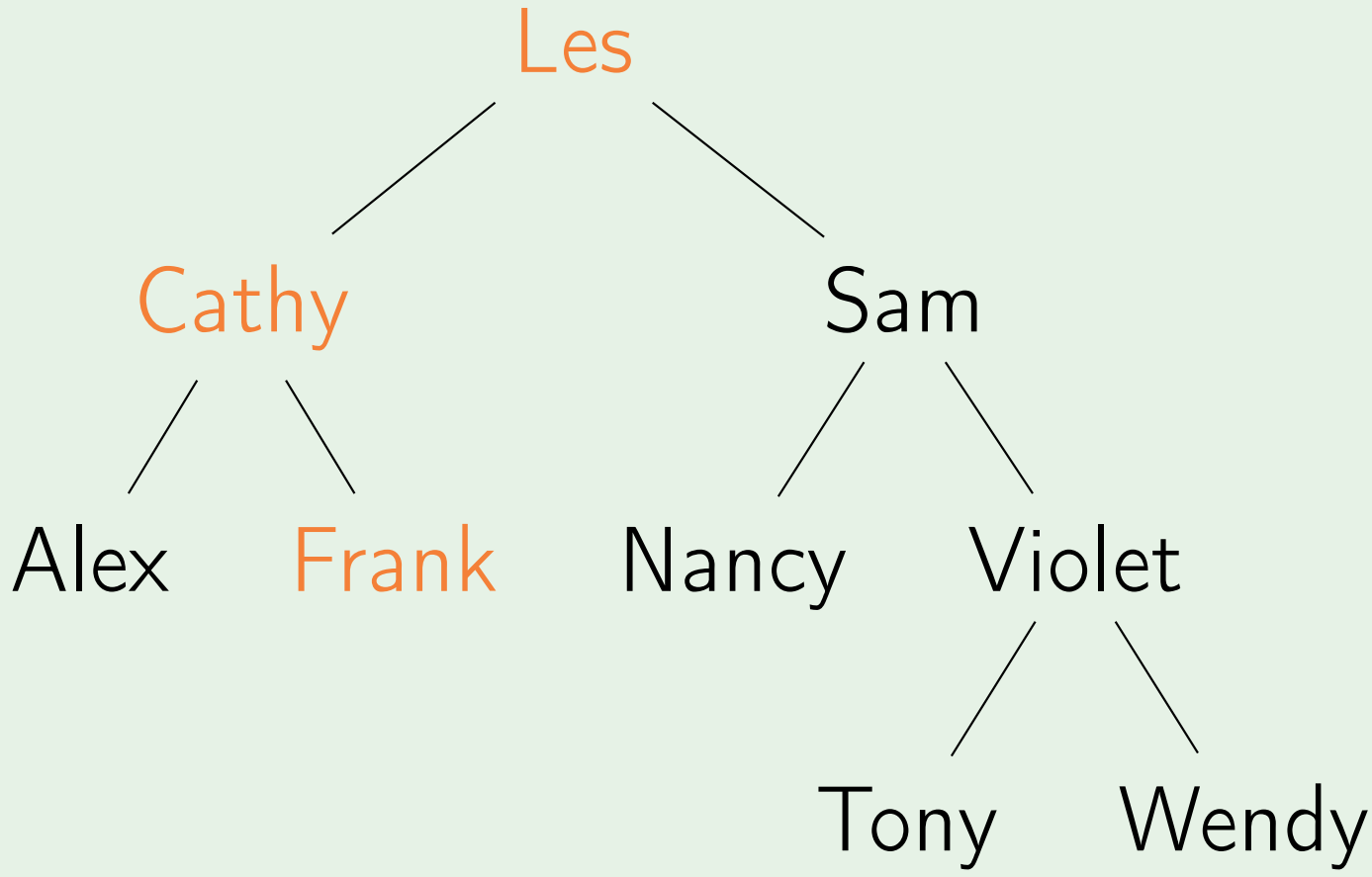
Output: Alex Cathy

InOrderTraversal



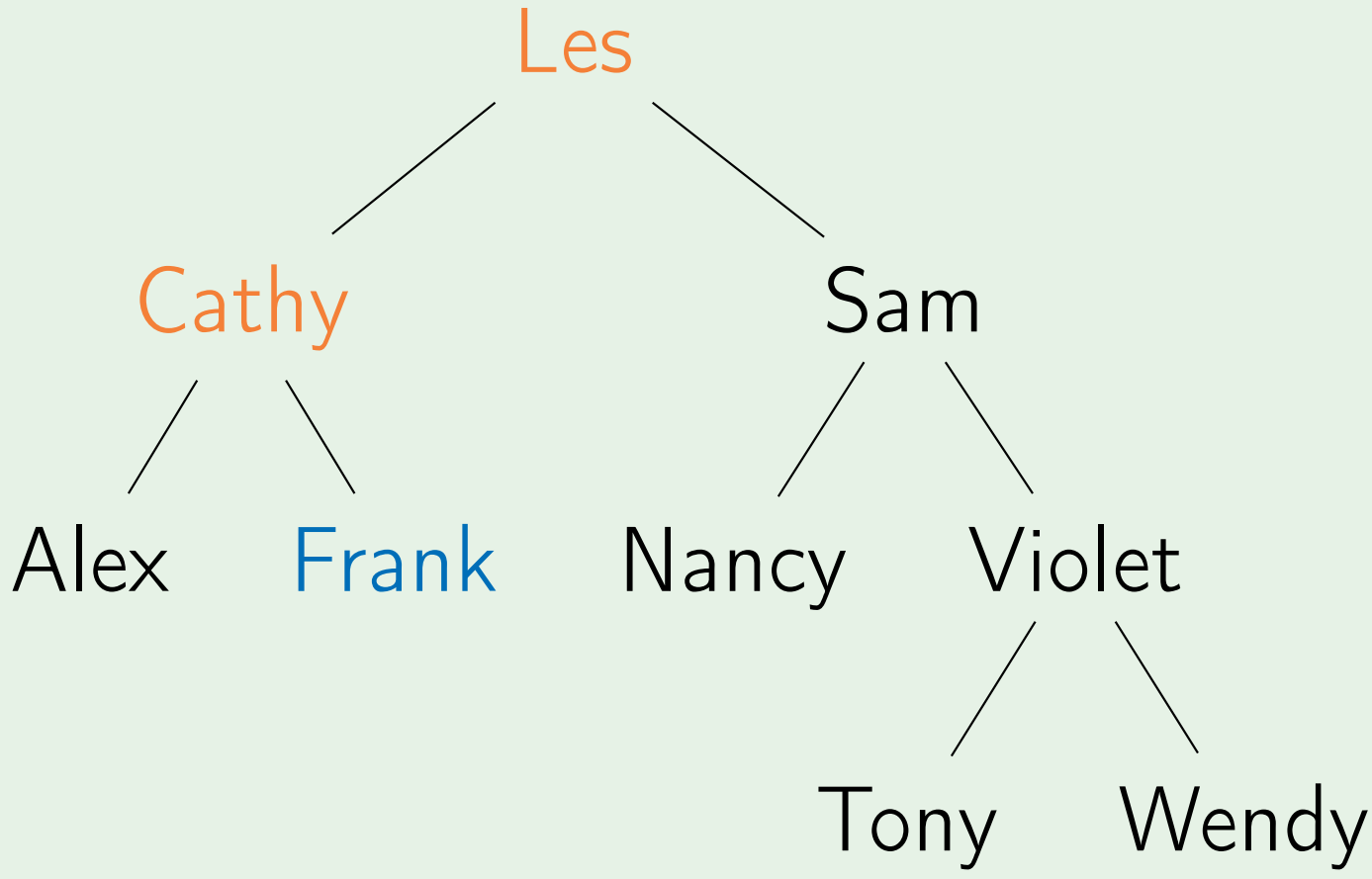
Output: Alex Cathy

InOrderTraversal



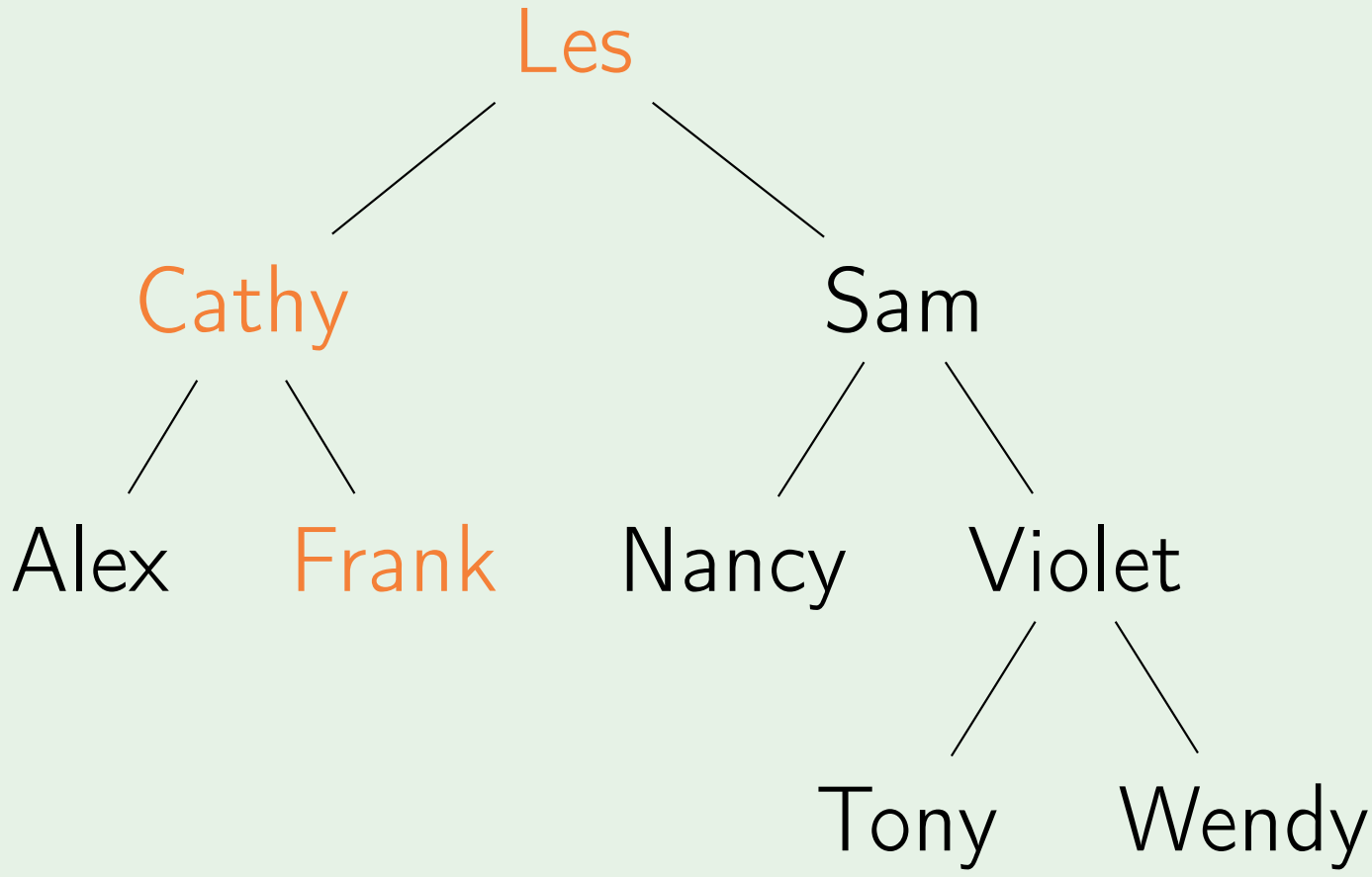
Output: Alex Cathy

InOrderTraversal



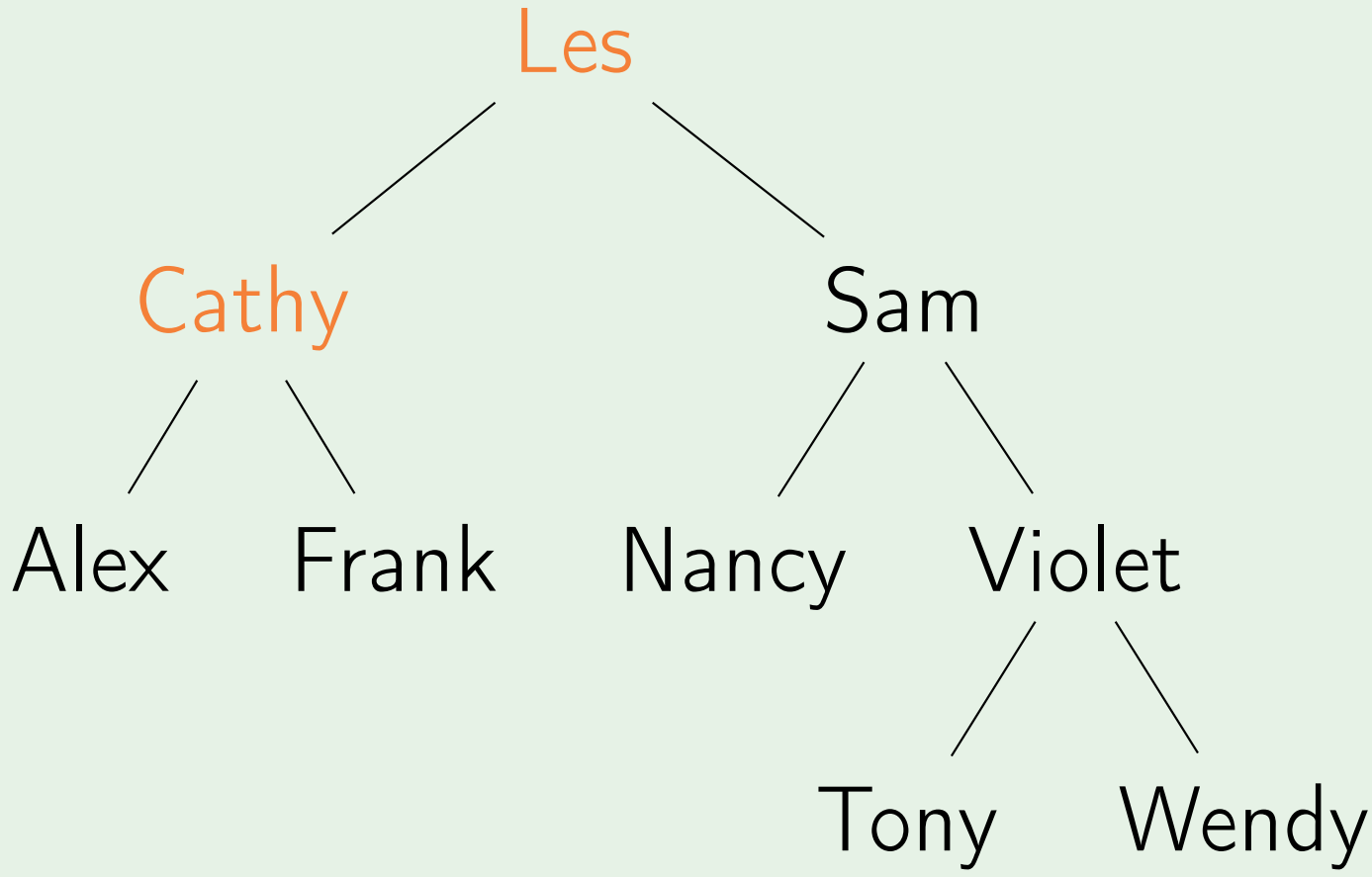
Output: Alex Cathy Frank

InOrderTraversal



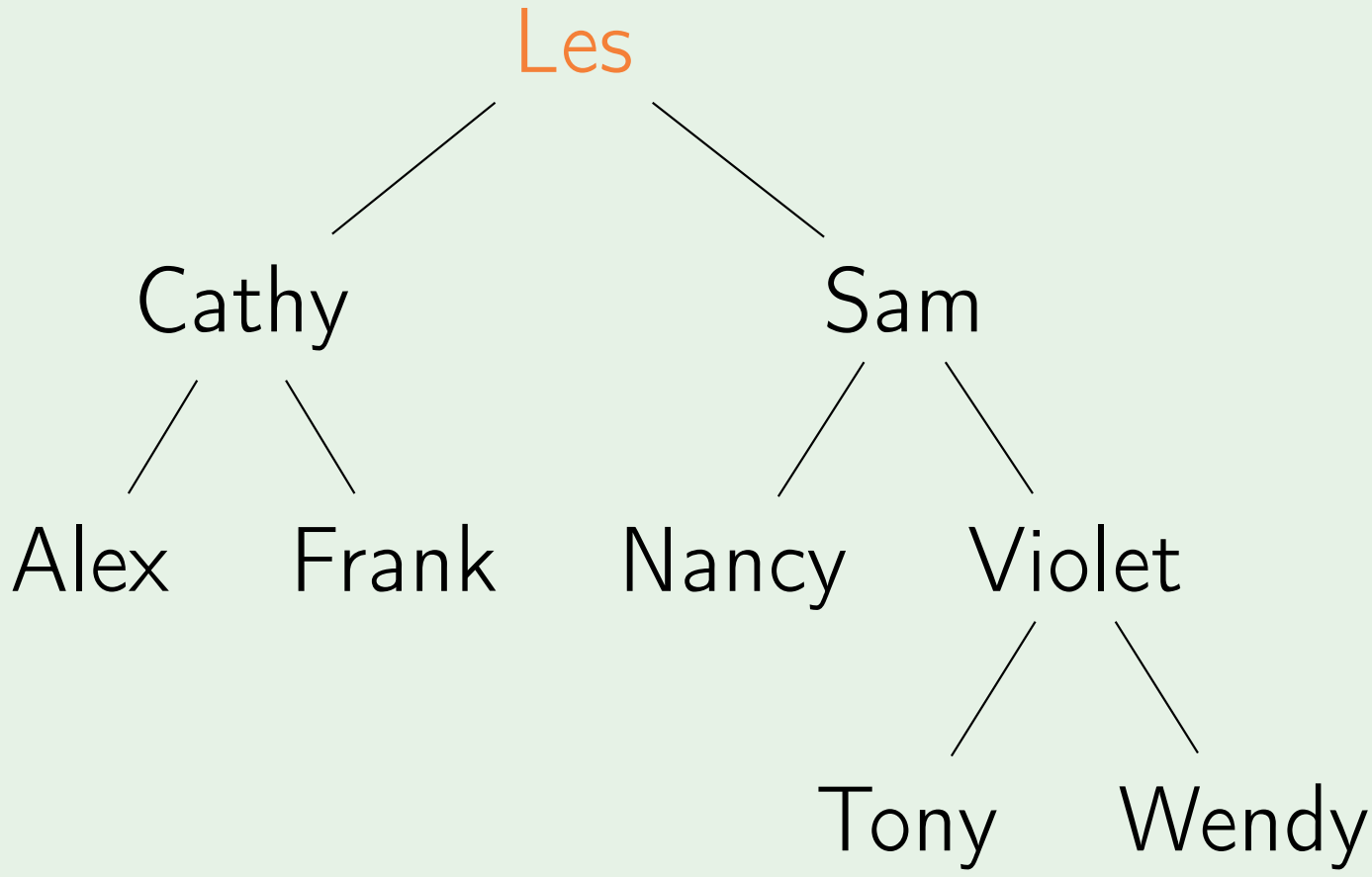
Output: Alex Cathy Frank

InOrderTraversal



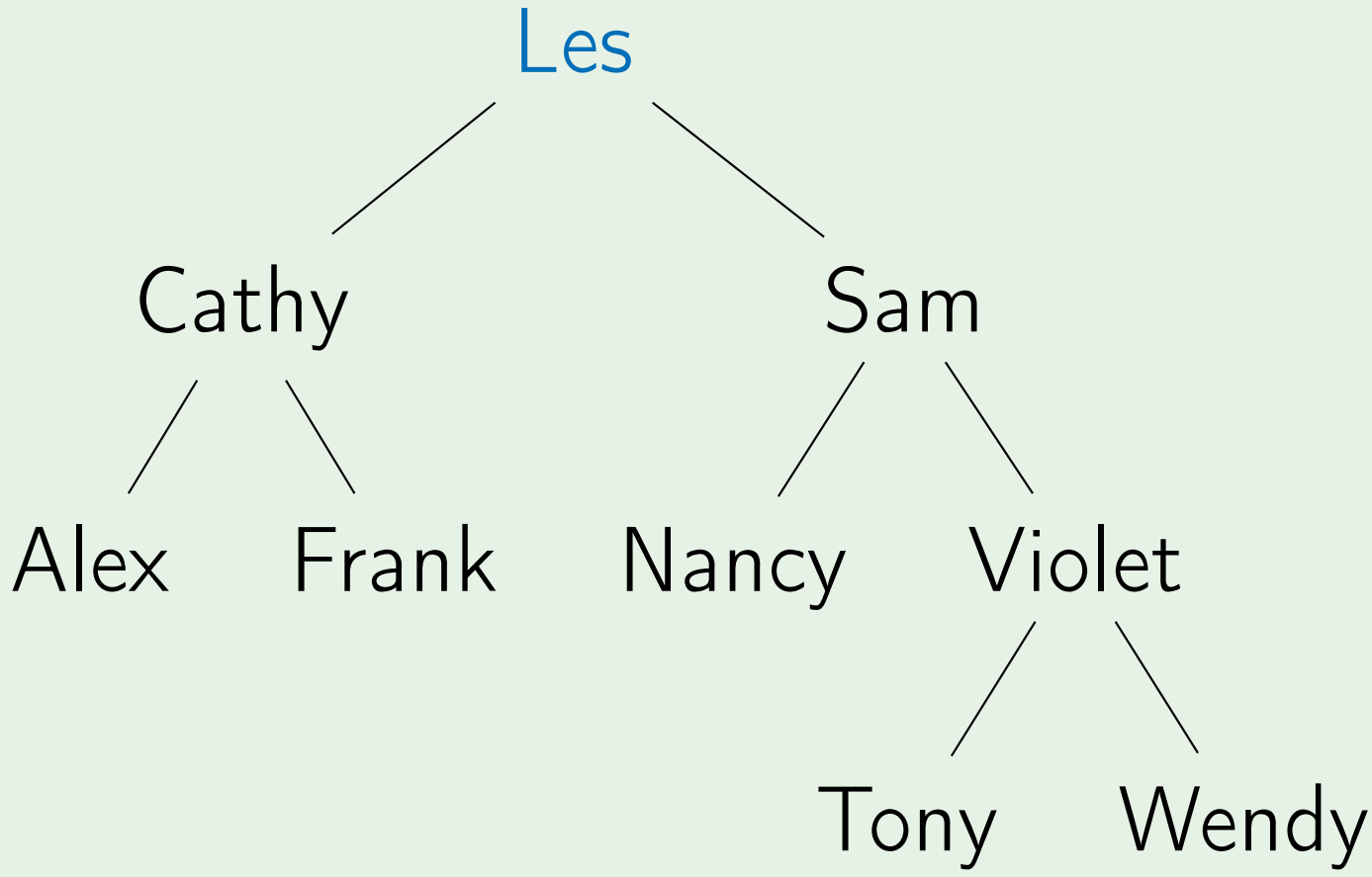
Output: Alex Cathy Frank

InOrderTraversal



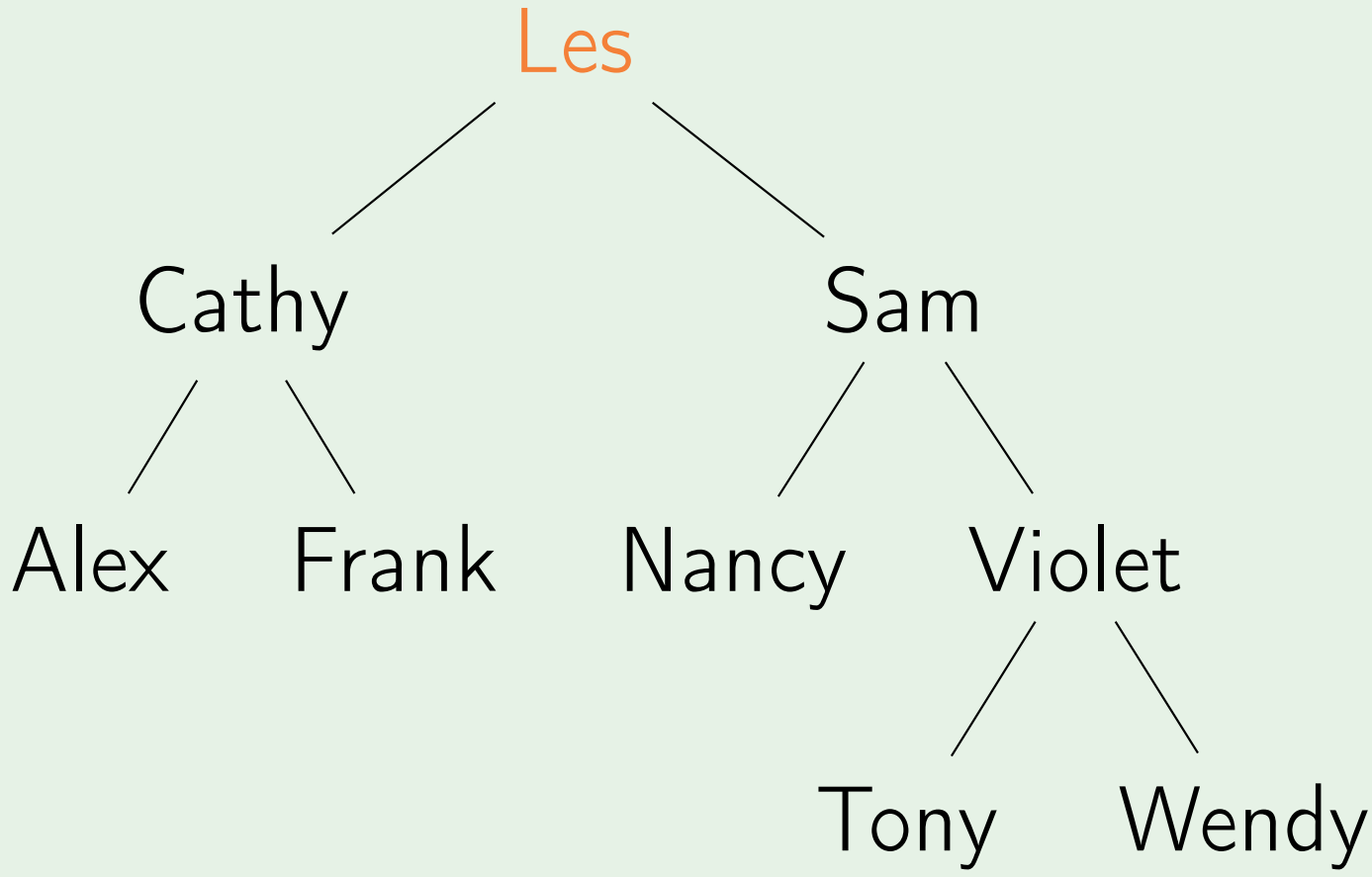
Output: Alex Cathy Frank

InOrderTraversal



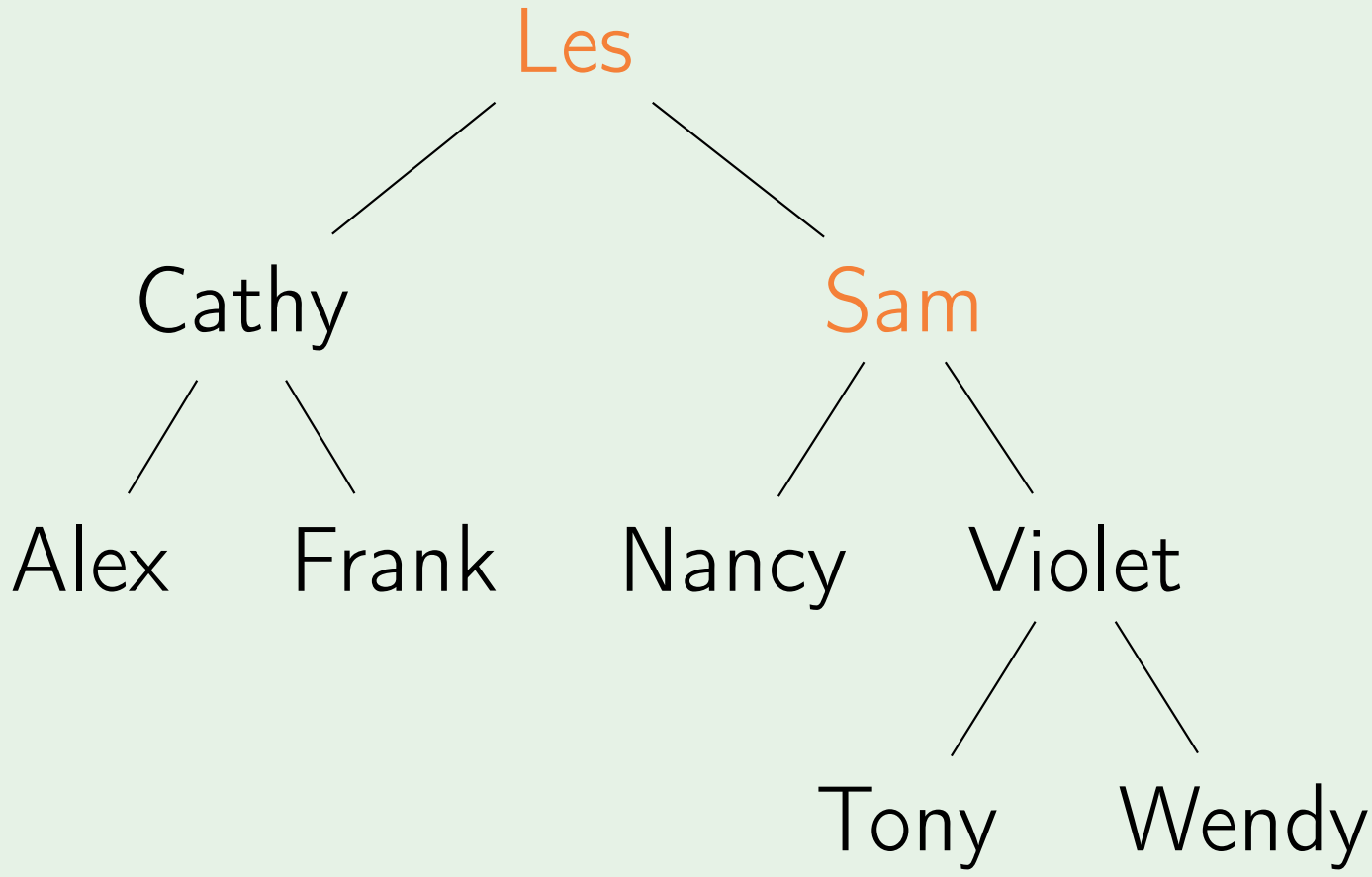
Output: Alex Cathy Frank Les

InOrderTraversal



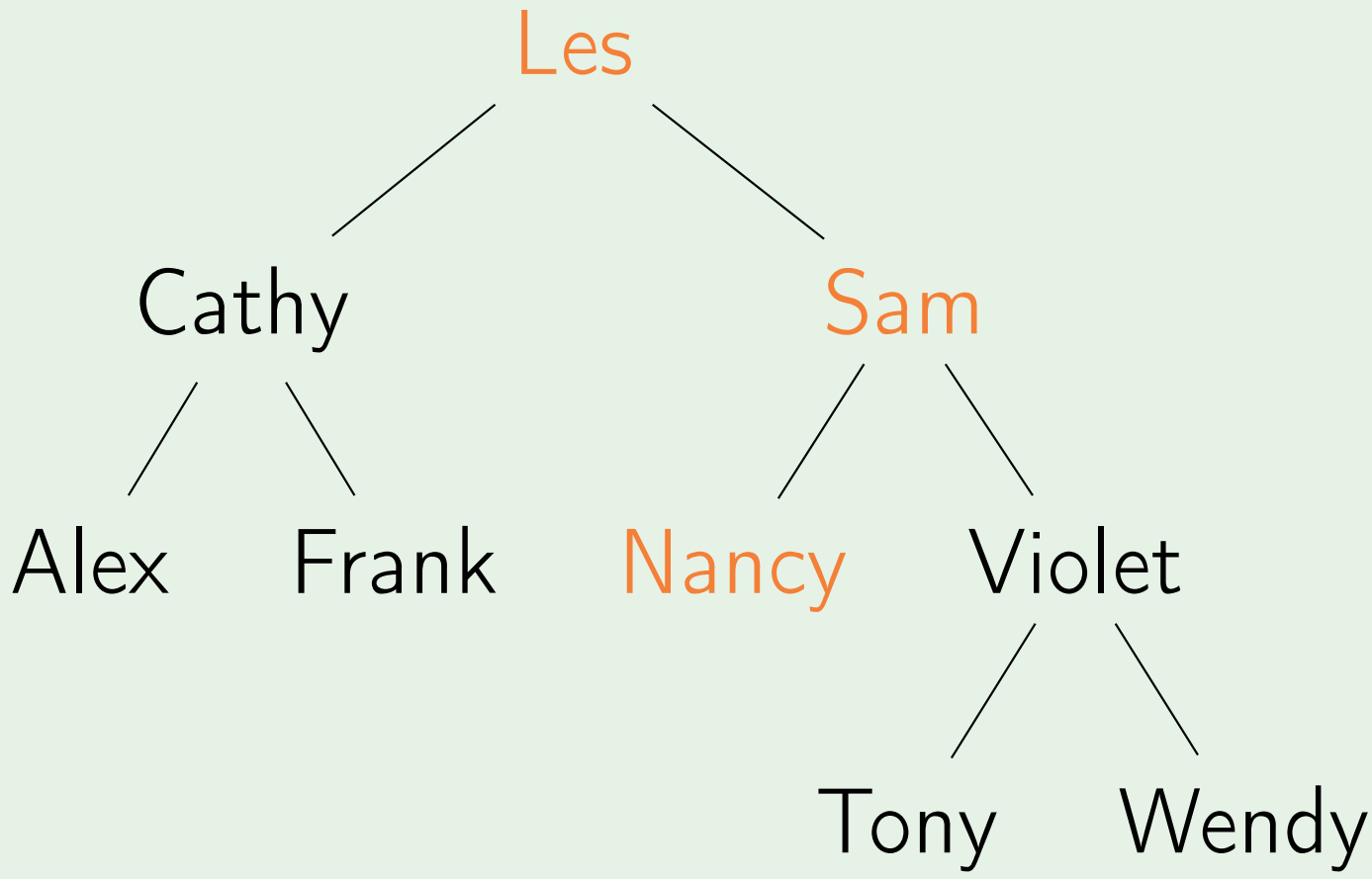
Output: Alex Cathy Frank Les

InOrderTraversal



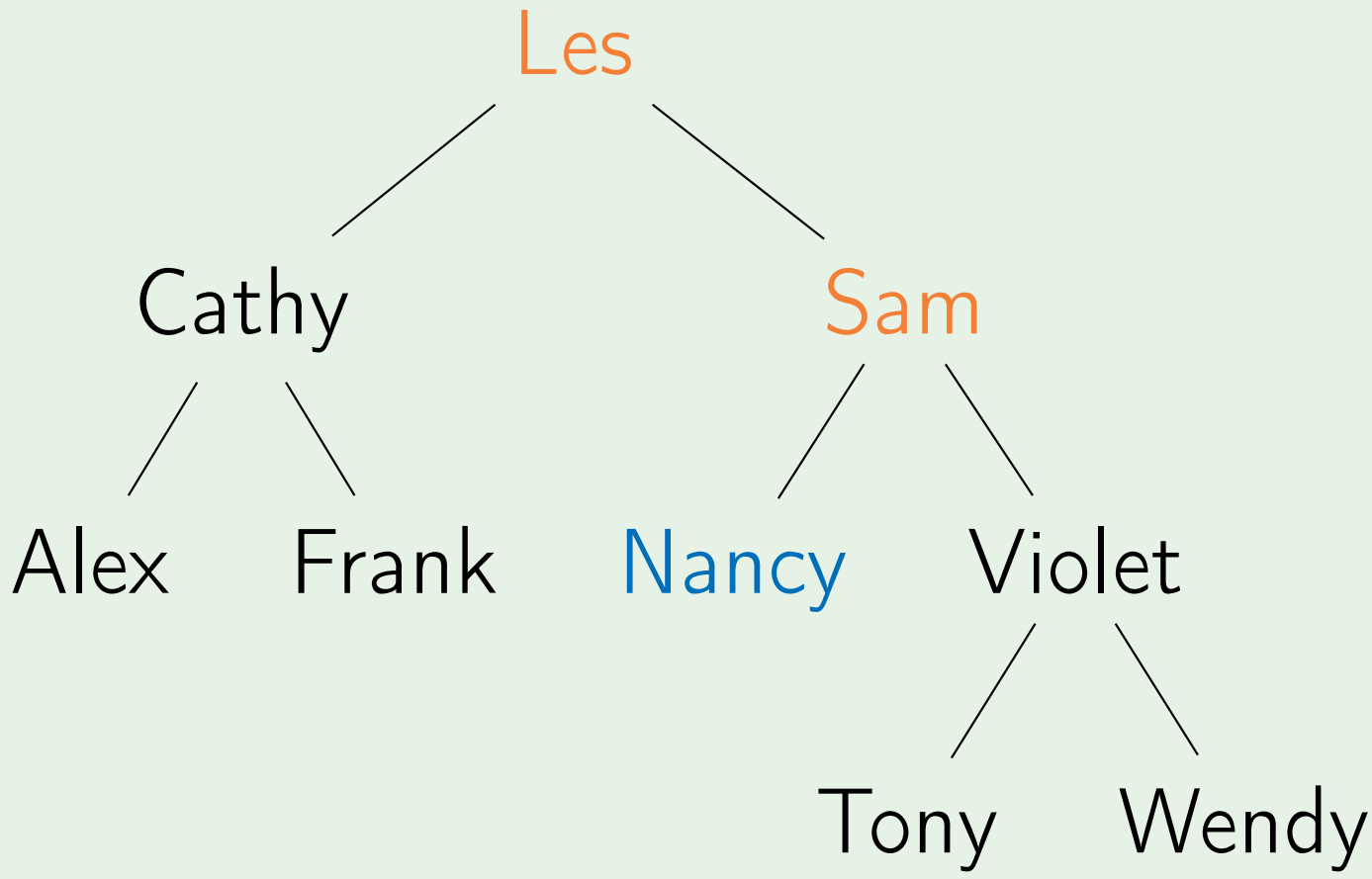
Output: Alex Cathy Frank Les

InOrderTraversal



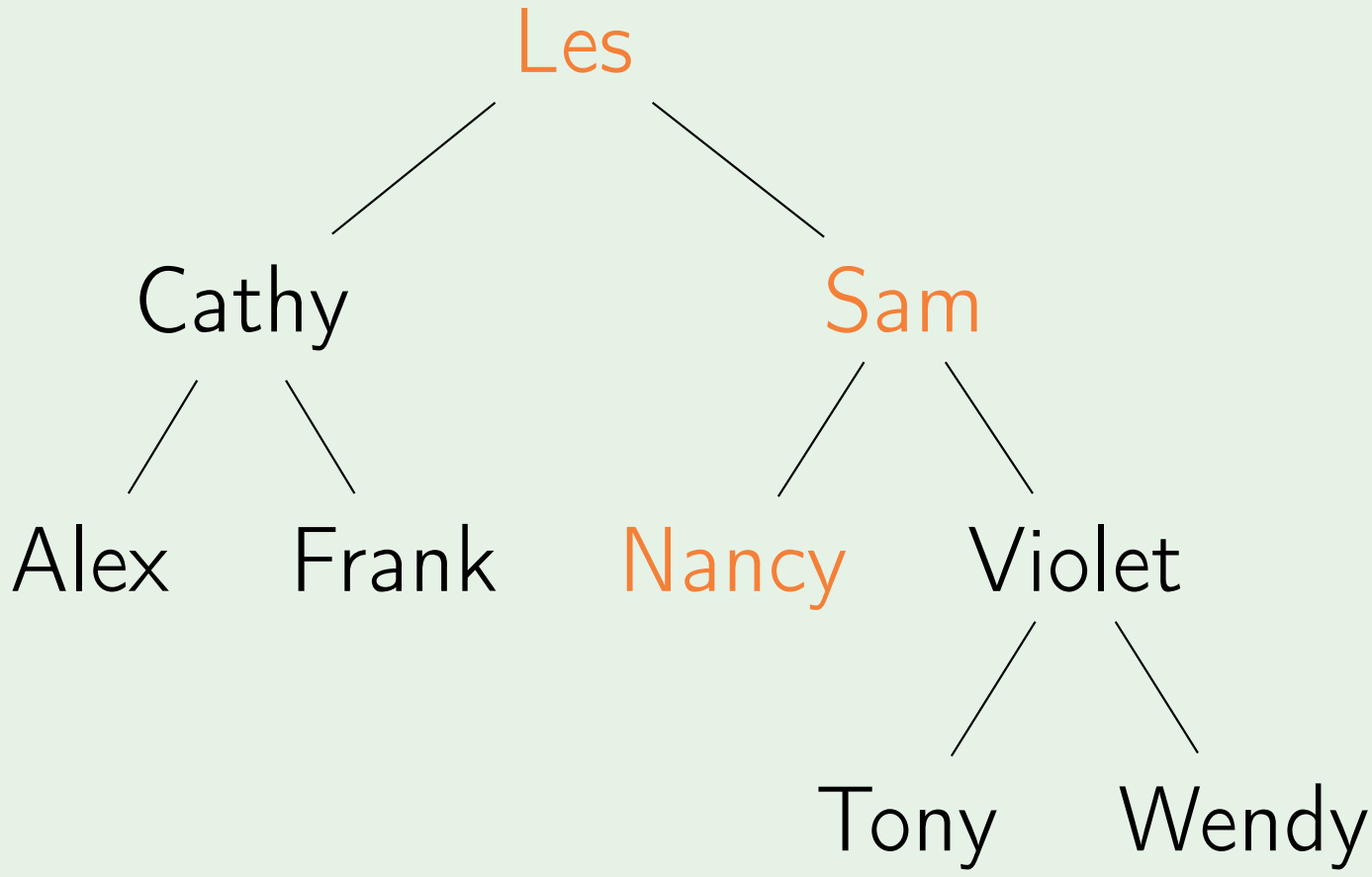
Output: Alex Cathy Frank Les

InOrderTraversal



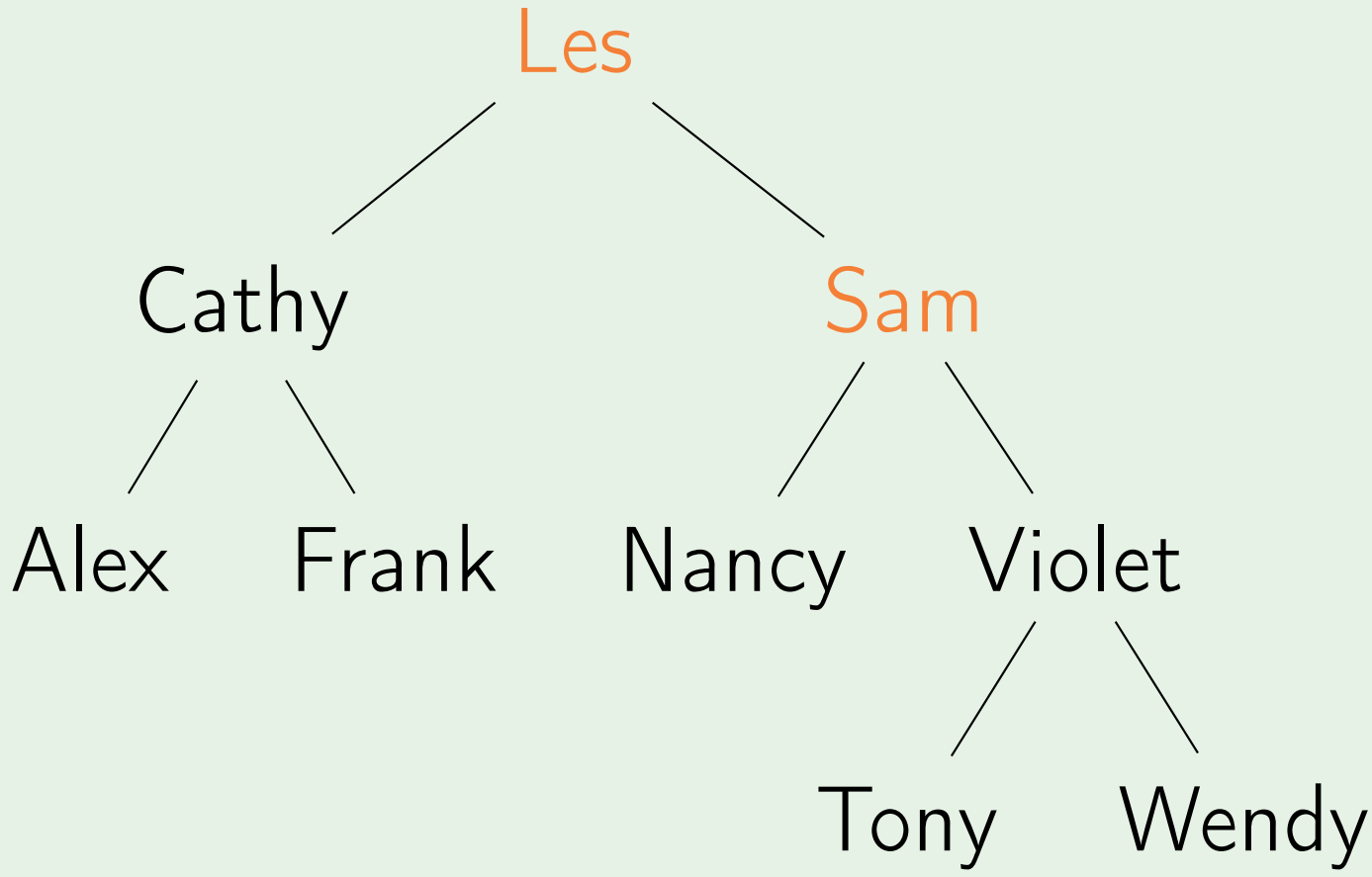
Output: Alex Cathy Frank Les Nancy

InOrderTraversal



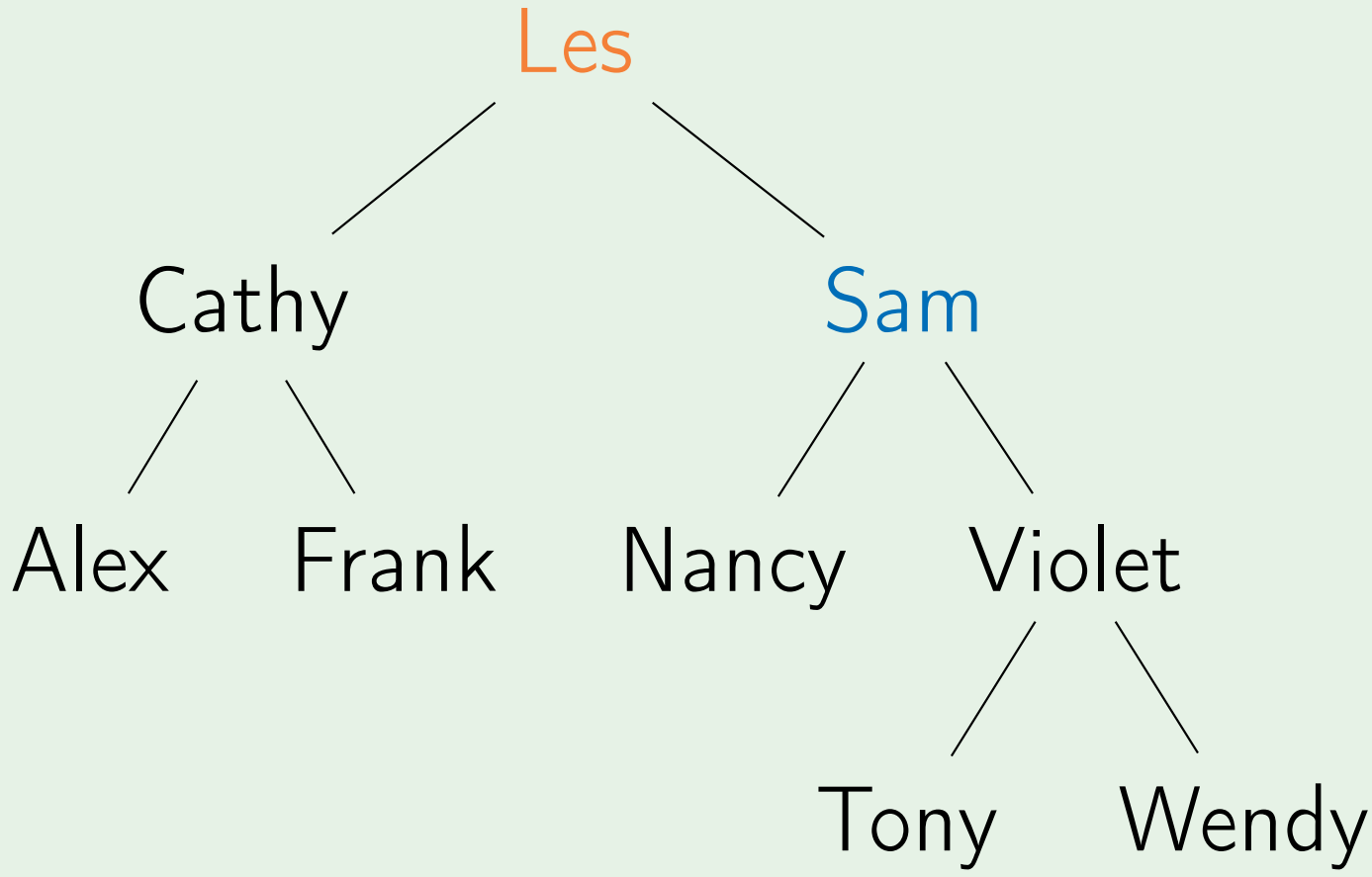
Output: Alex Cathy Frank Les Nancy

InOrderTraversal



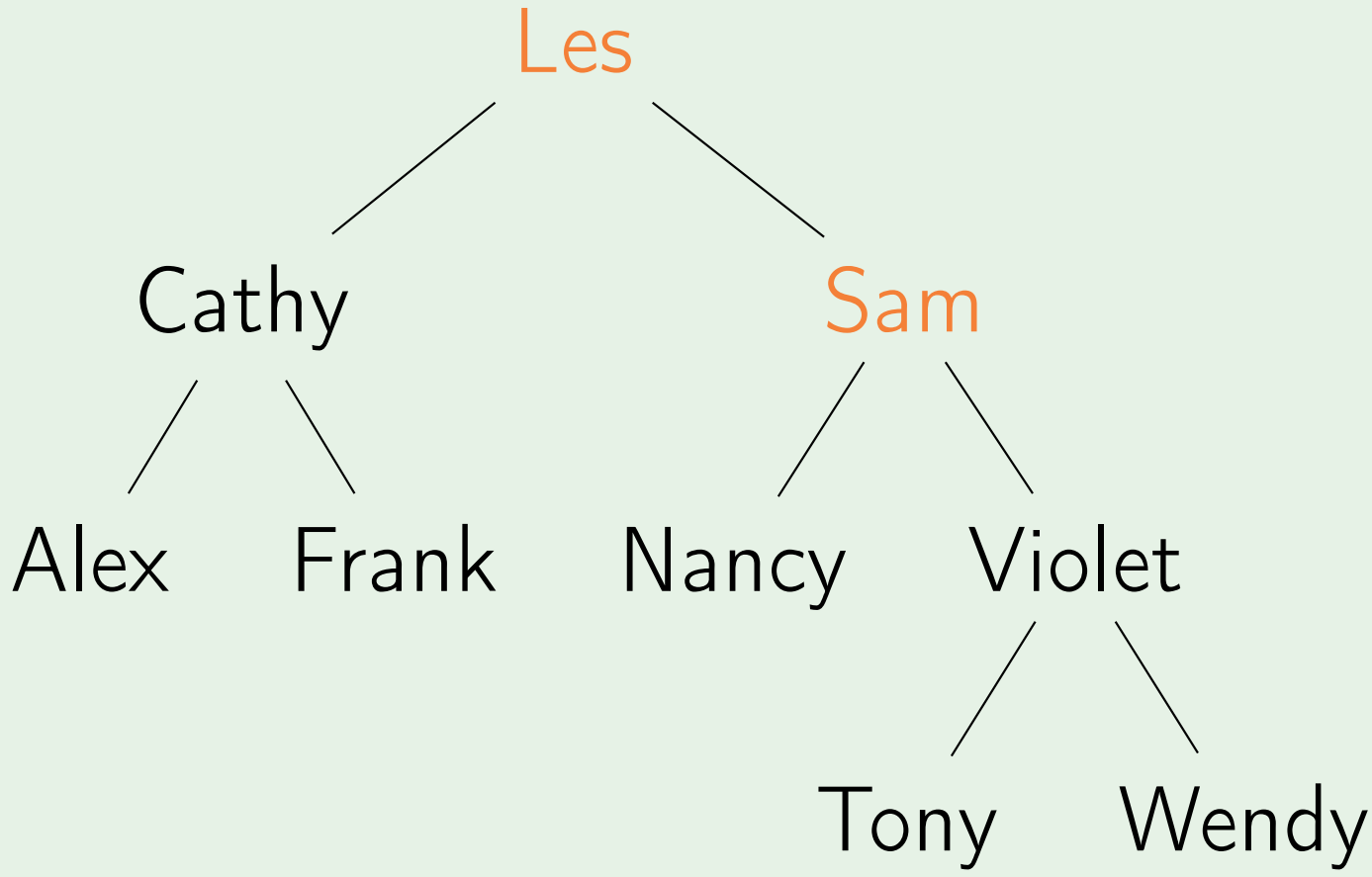
Output: Alex Cathy Frank Les Nancy

InOrderTraversal



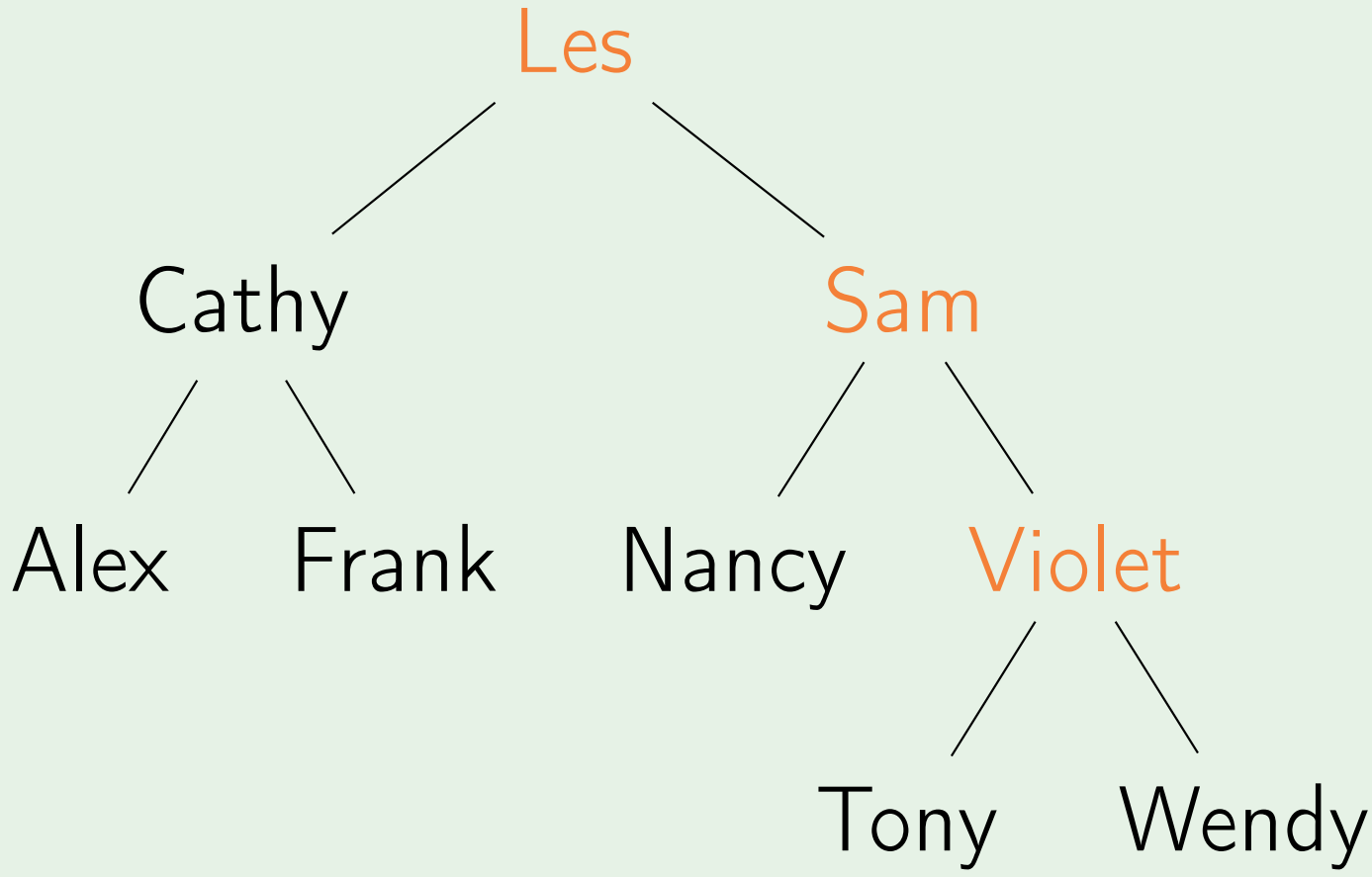
Output: Alex Cathy Frank Les Nancy Sam

InOrderTraversal



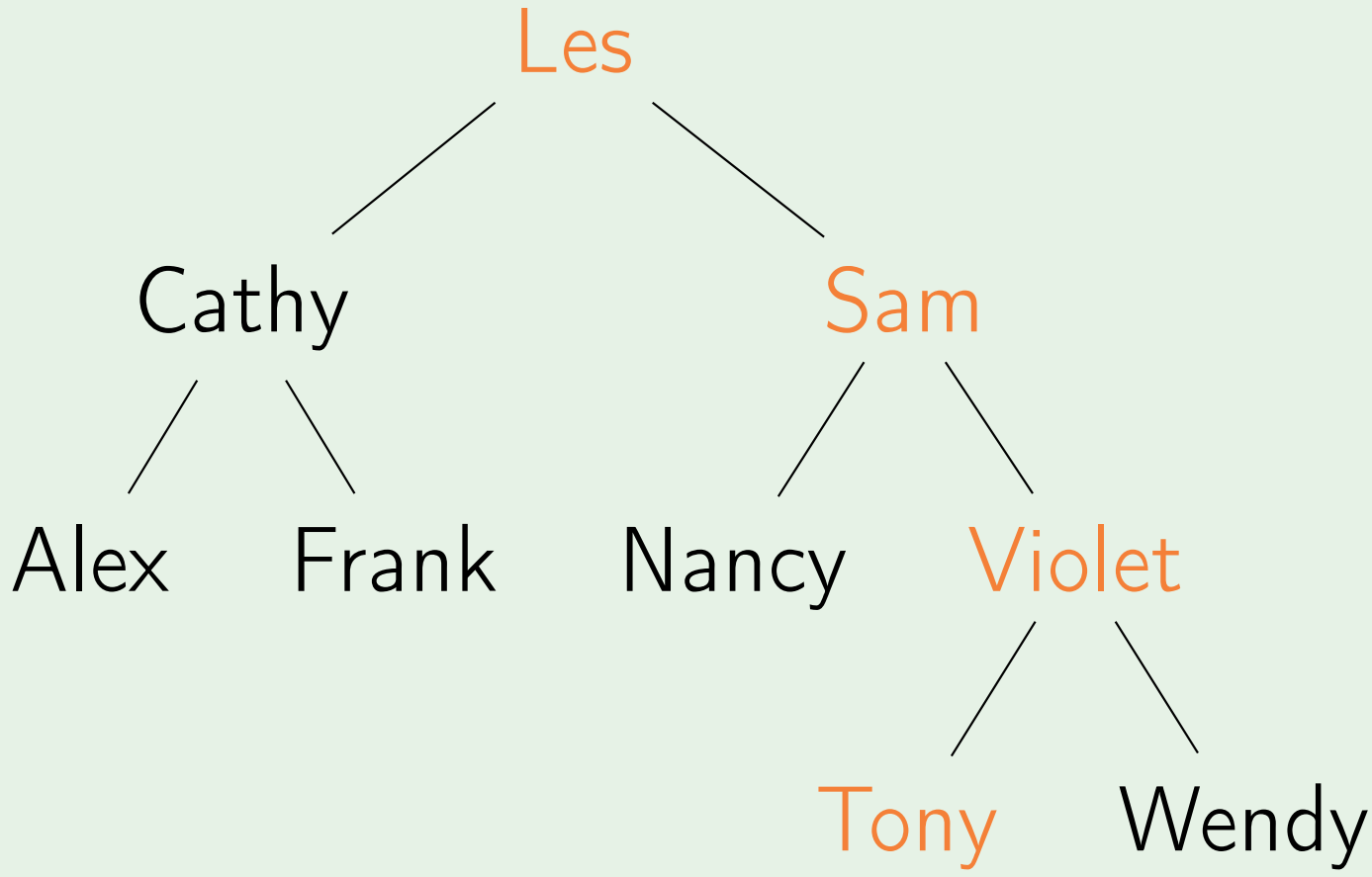
Output: Alex Cathy Frank Les Nancy Sam

InOrderTraversal



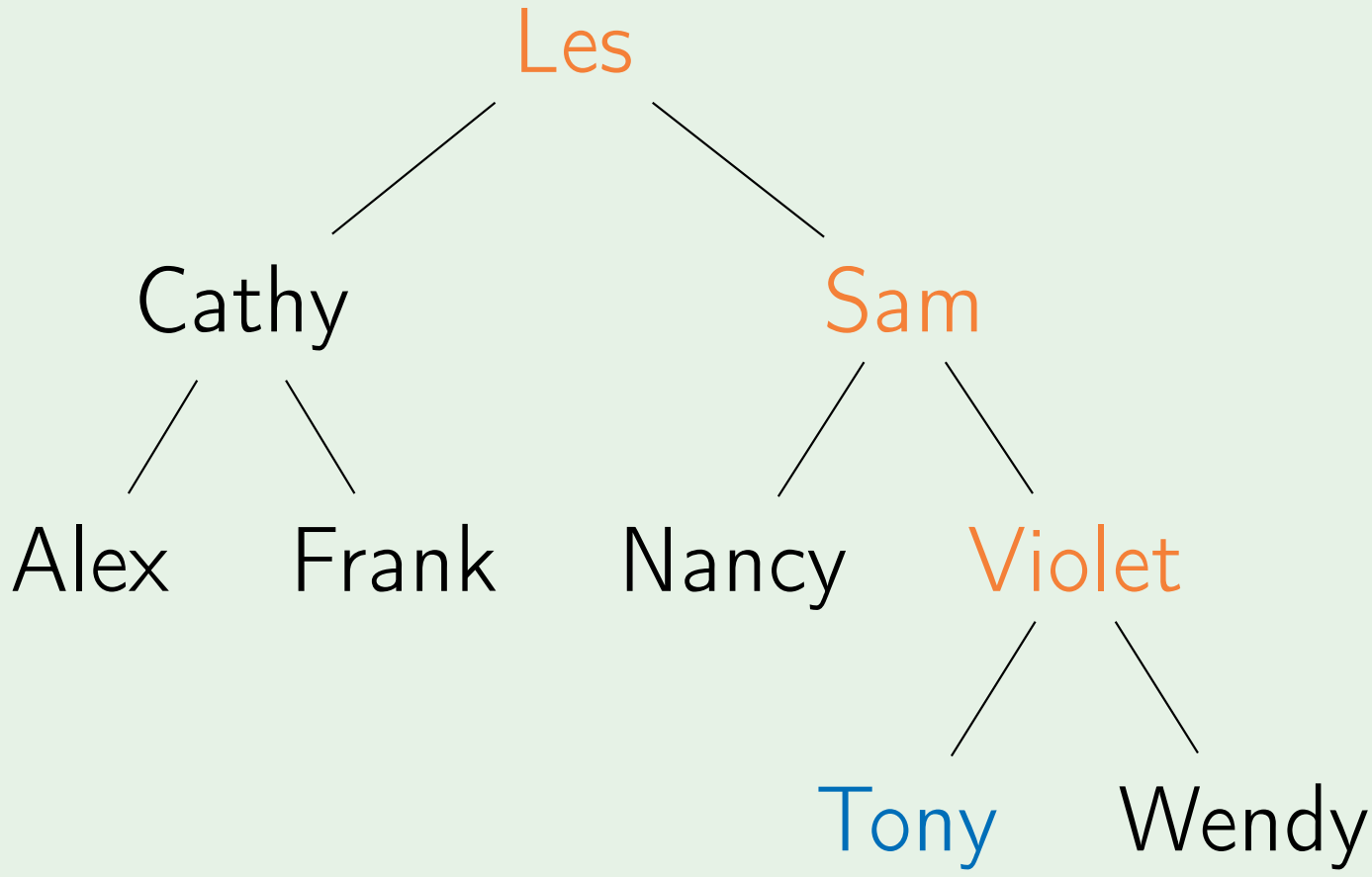
Output: Alex Cathy Frank Les Nancy Sam

InOrderTraversal



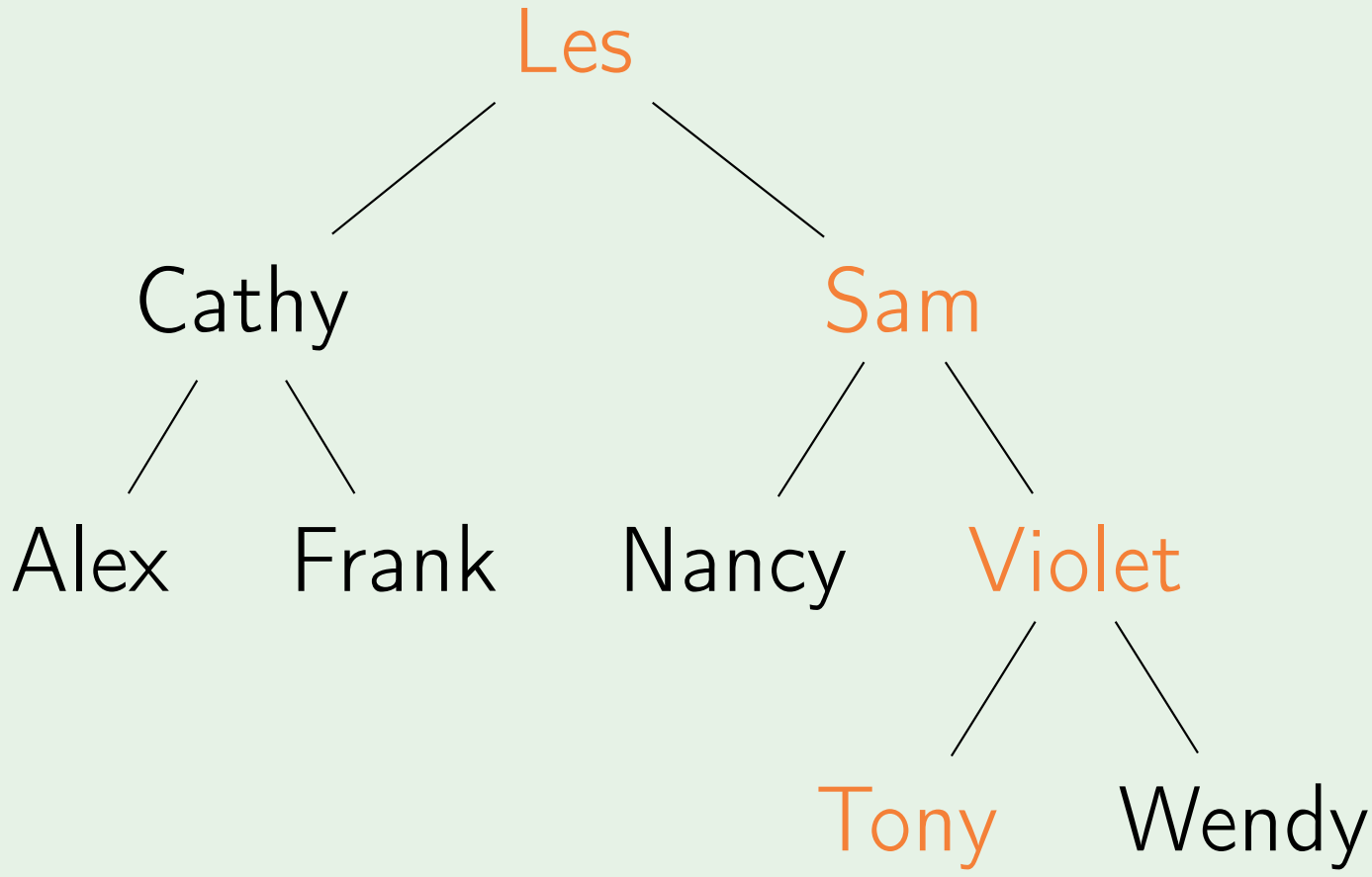
Output: Alex Cathy Frank Les Nancy Sam

InOrderTraversal



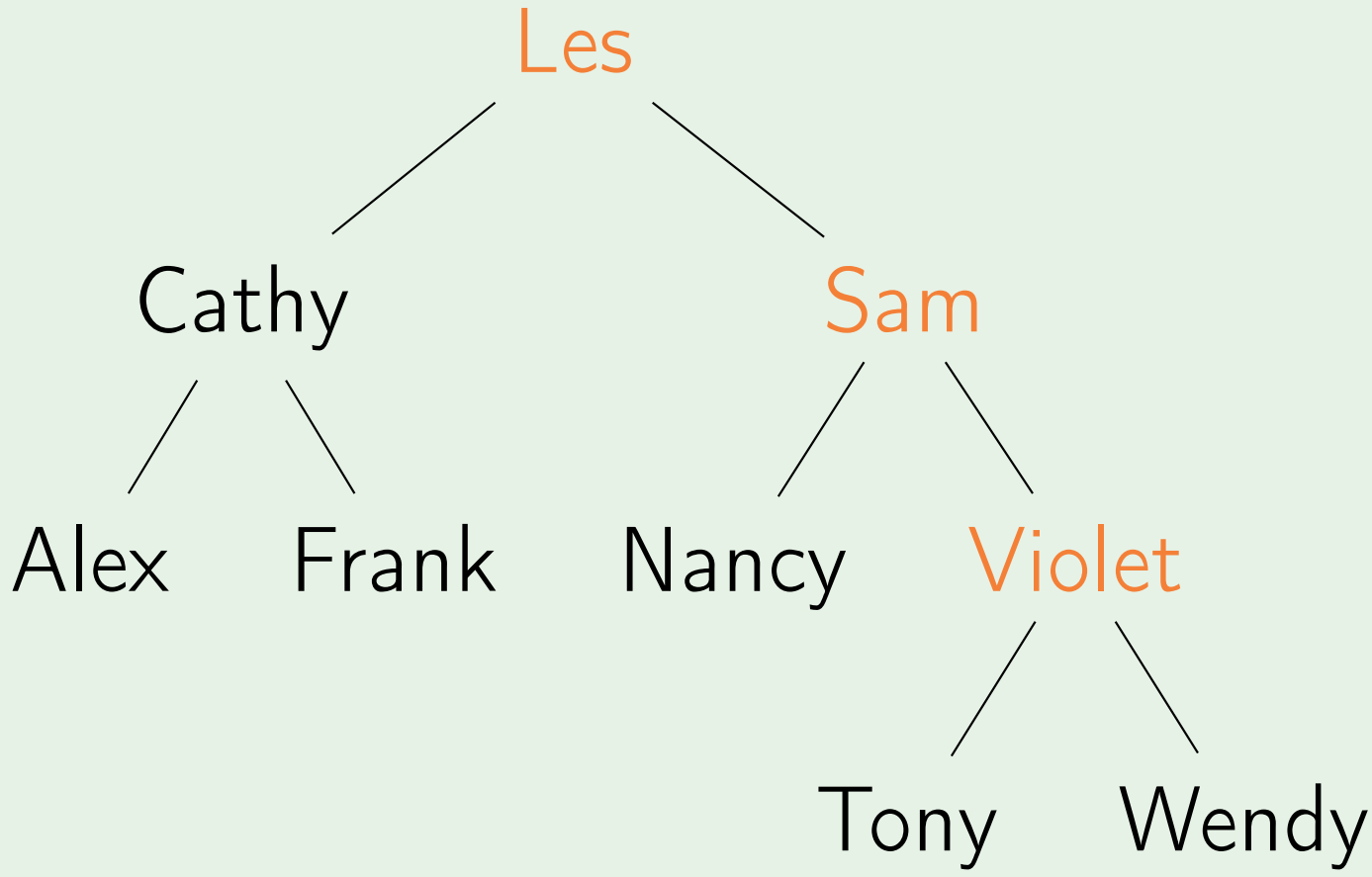
Output: Alex Cathy Frank Les Nancy Sam Tony

InOrderTraversal



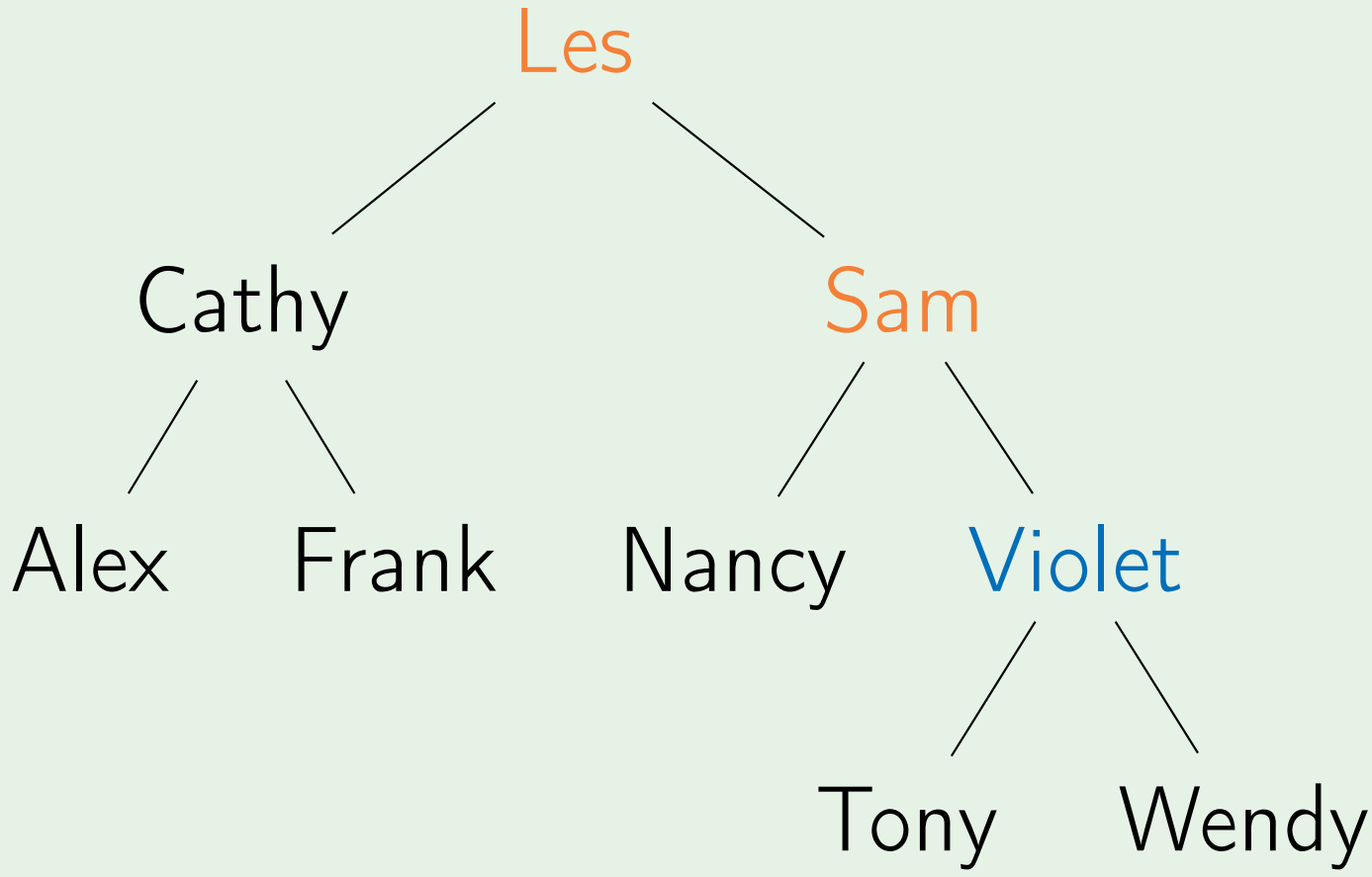
Output: Alex Cathy Frank Les Nancy Sam Tony

InOrderTraversal



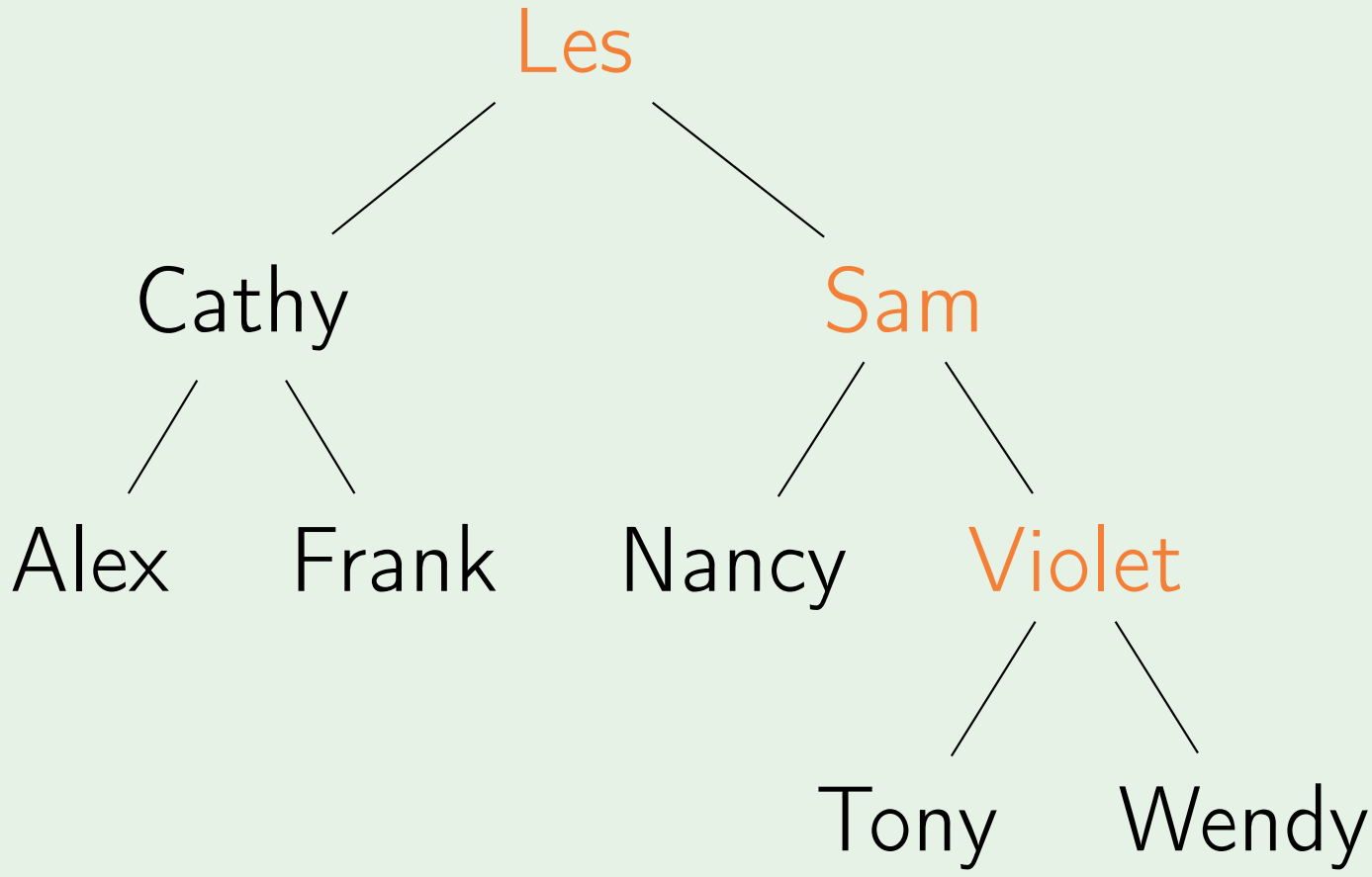
Output: Alex Cathy Frank Les Nancy Sam Tony

InOrderTraversal



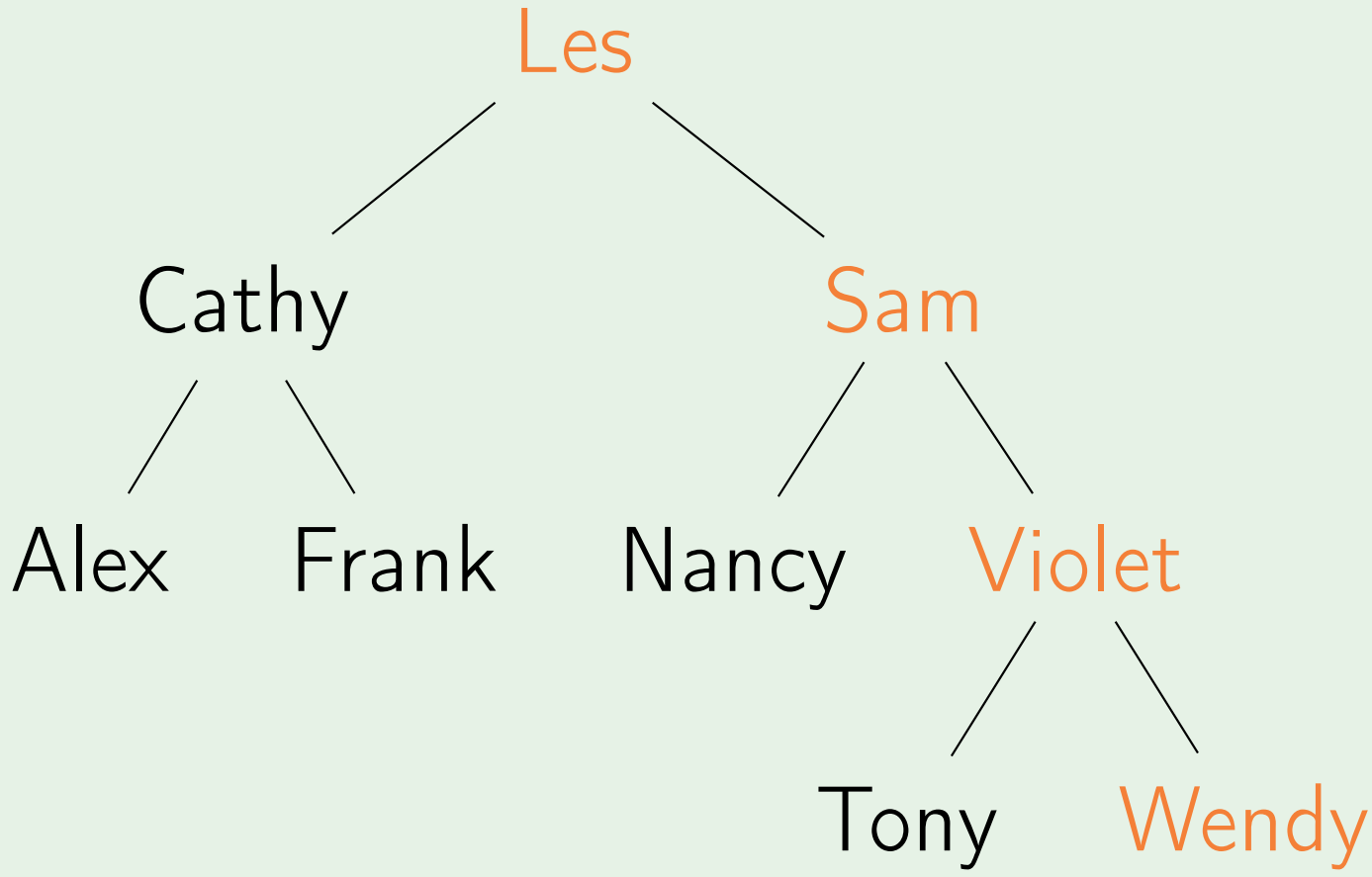
Output: Alex Cathy Frank Les Nancy Sam Tony Violet

InOrderTraversal



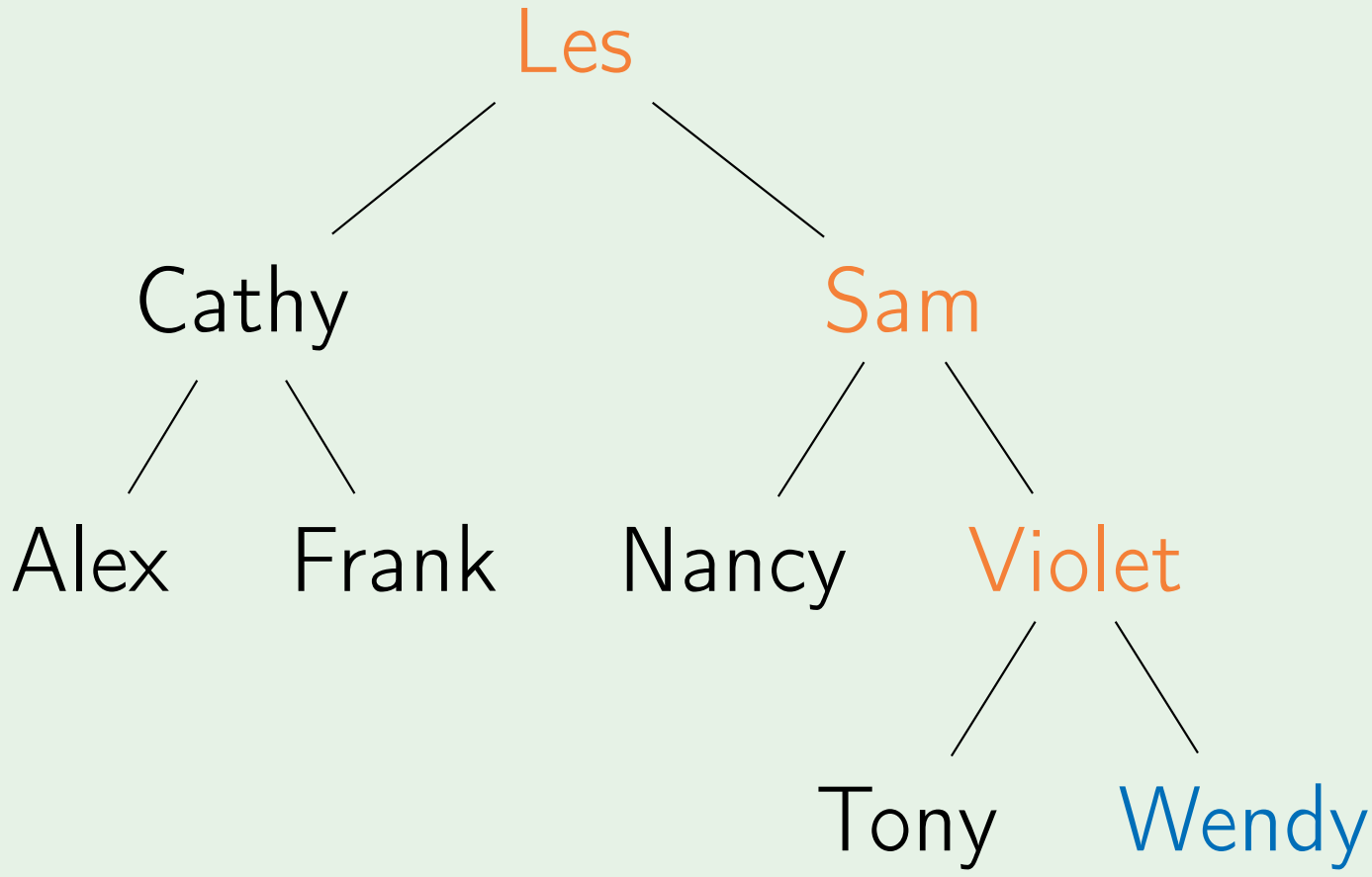
Output: Alex Cathy Frank Les Nancy Sam Tony Violet

InOrderTraversal



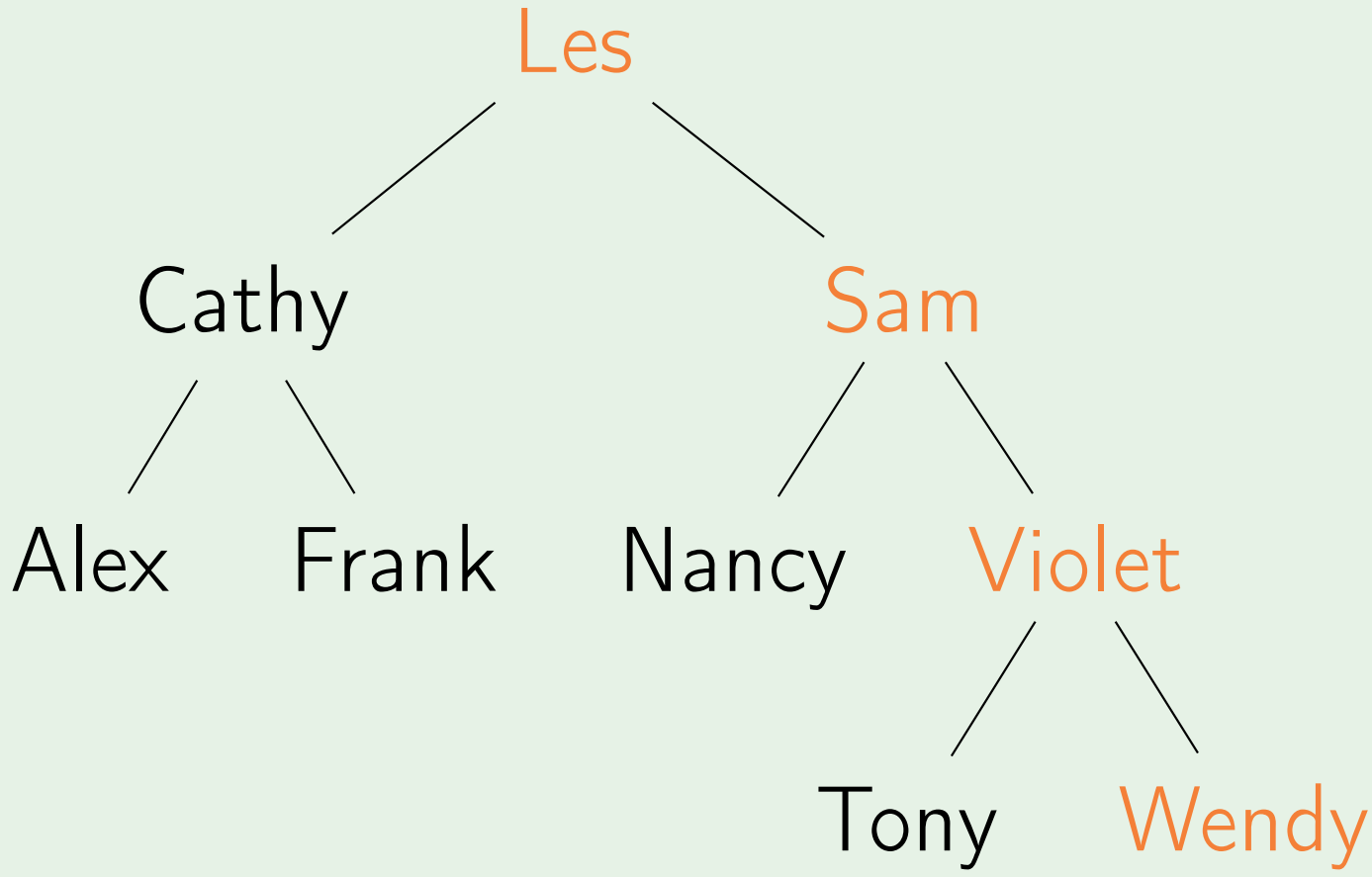
Output: Alex Cathy Frank Les Nancy Sam Tony Violet

InOrderTraversal



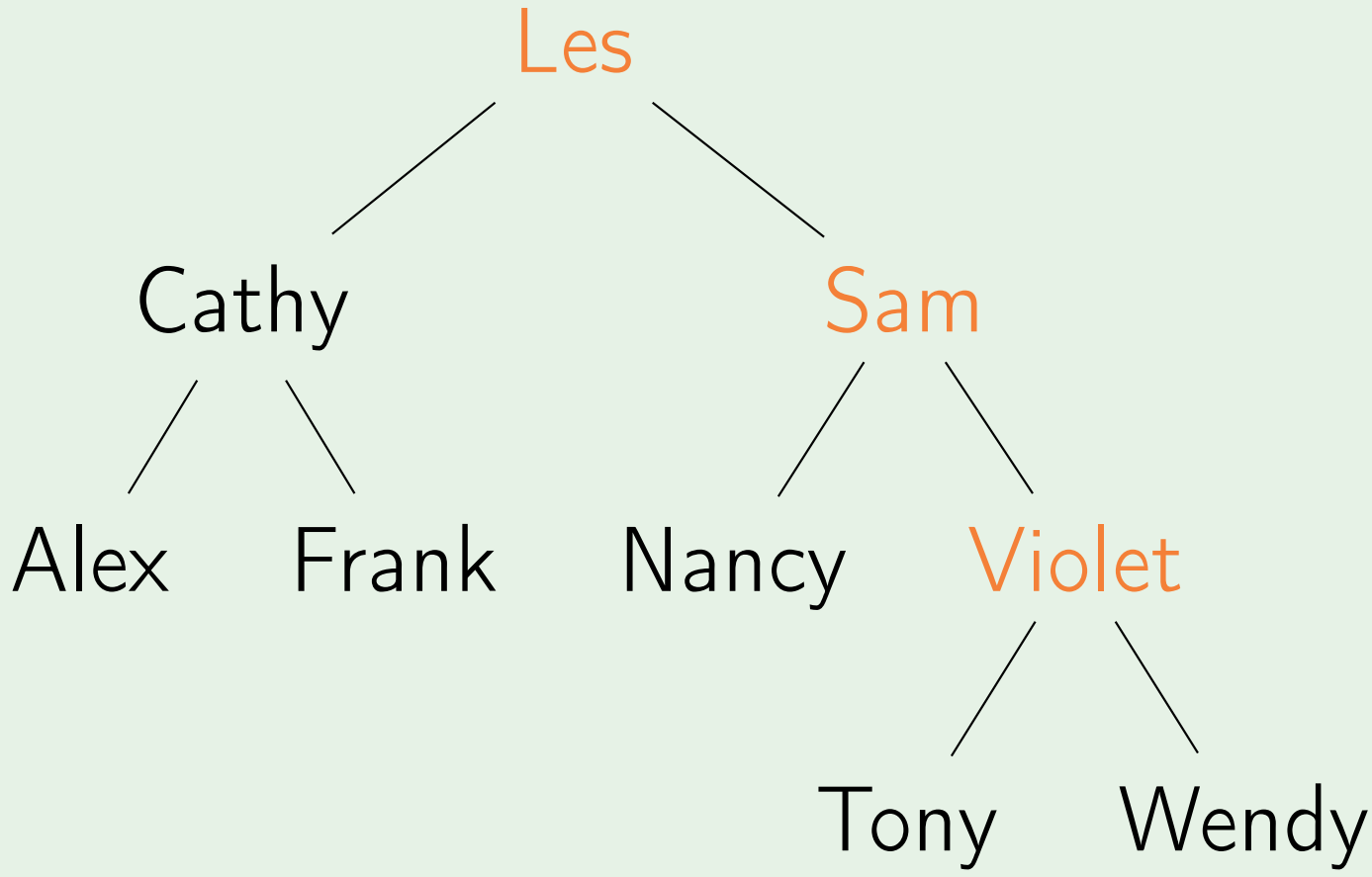
Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

InOrderTraversal



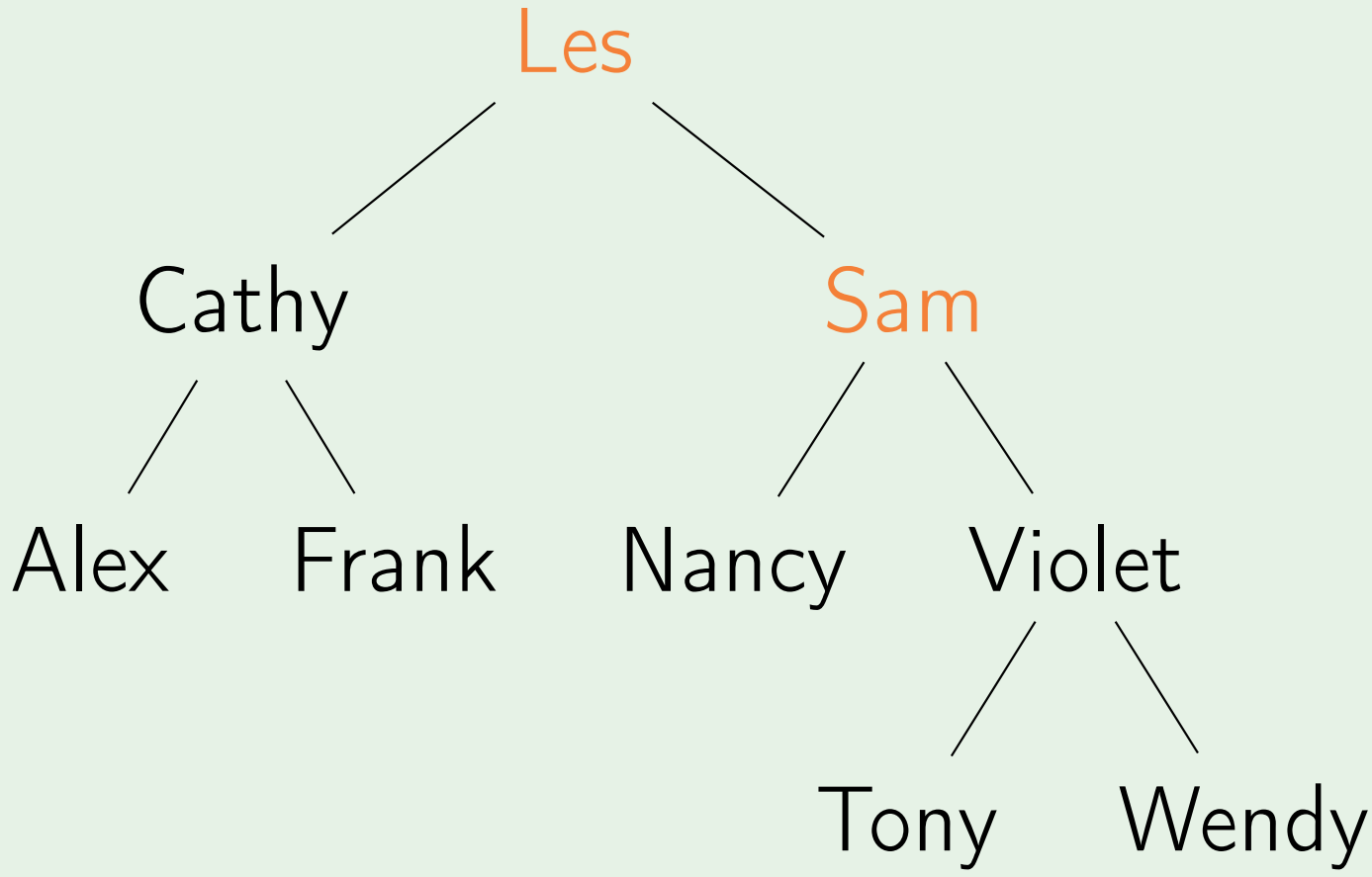
Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

InOrderTraversal



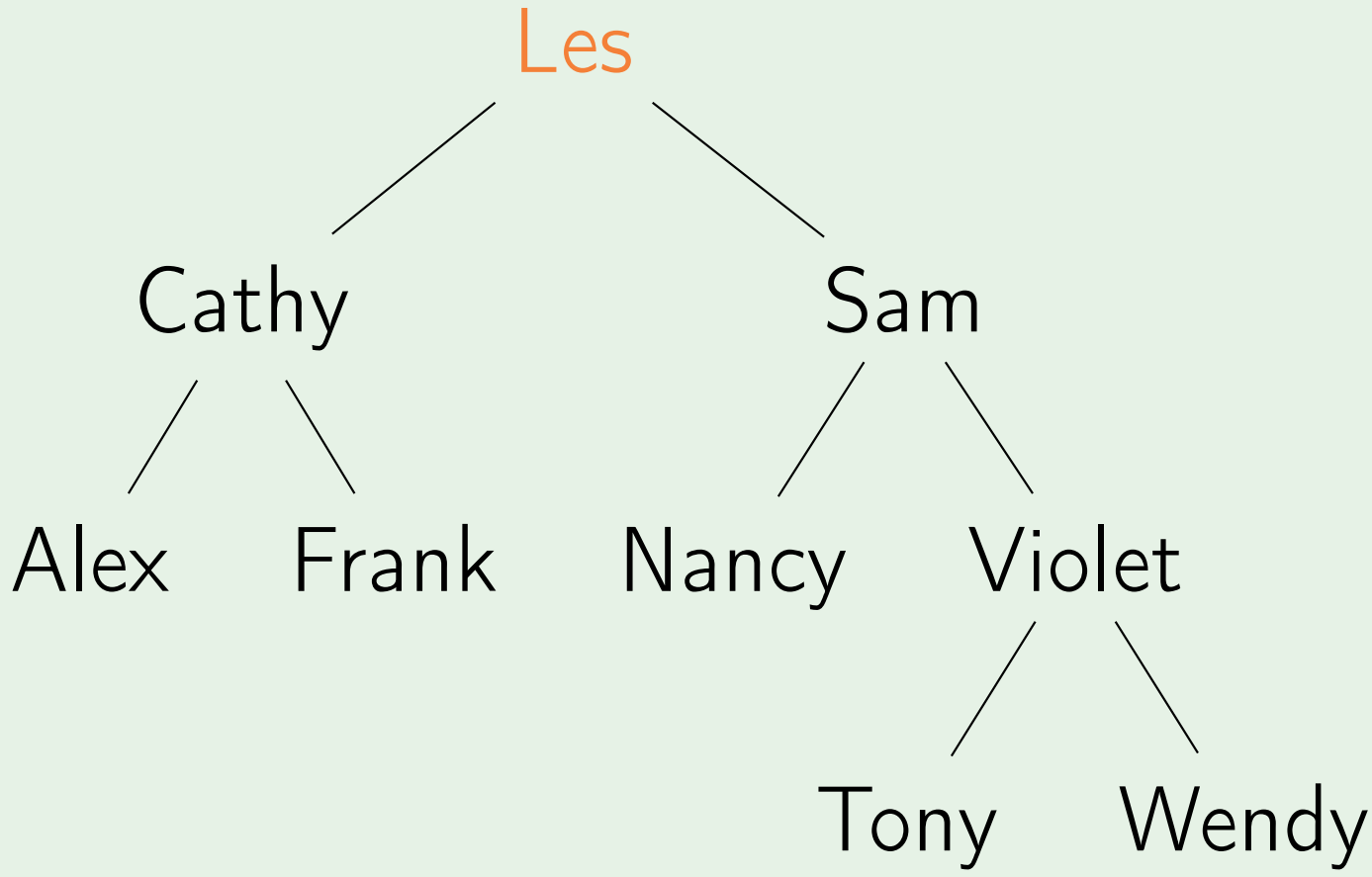
Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

InOrderTraversal



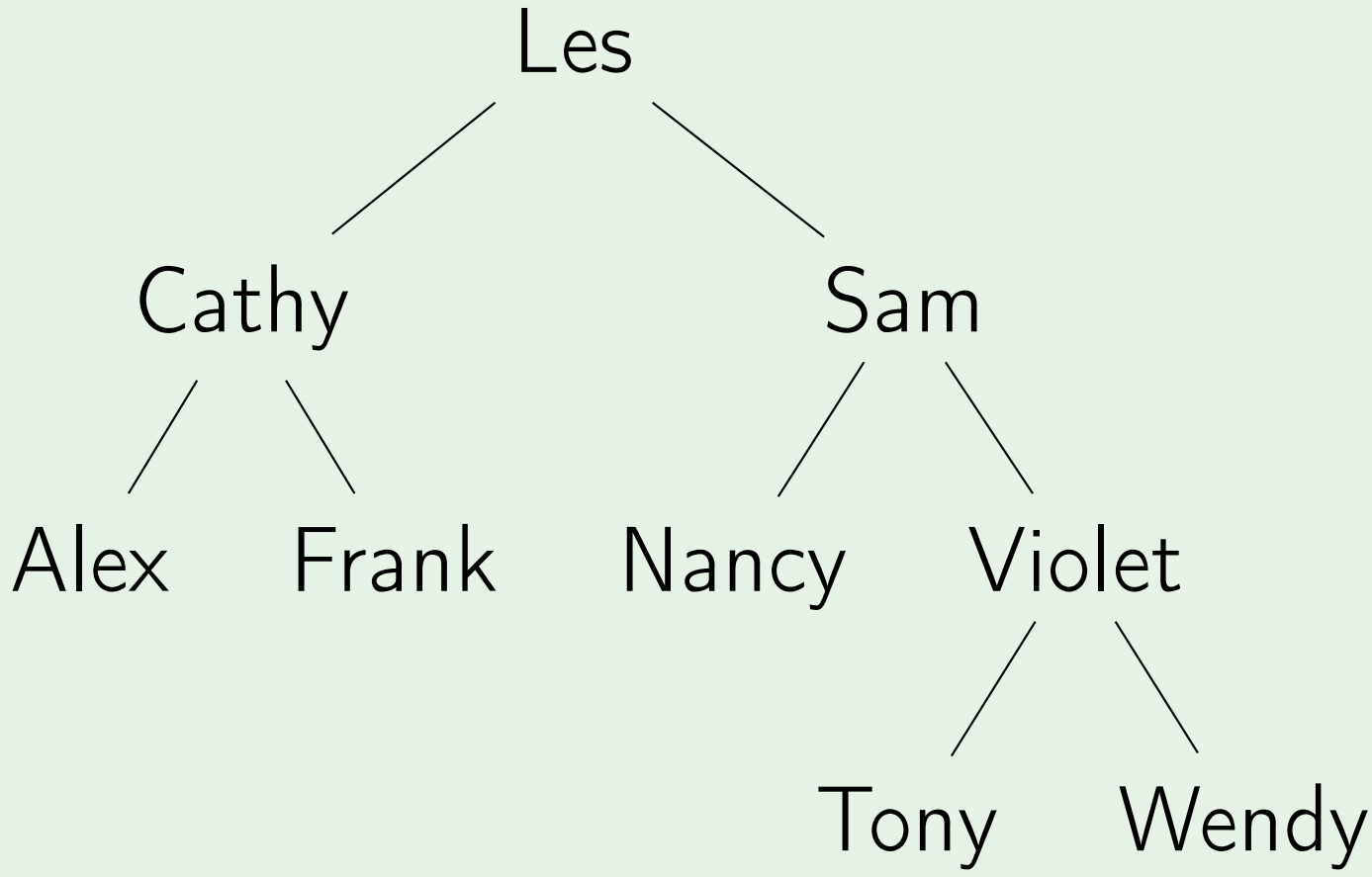
Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

InOrderTraversal



Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

InOrderTraversal

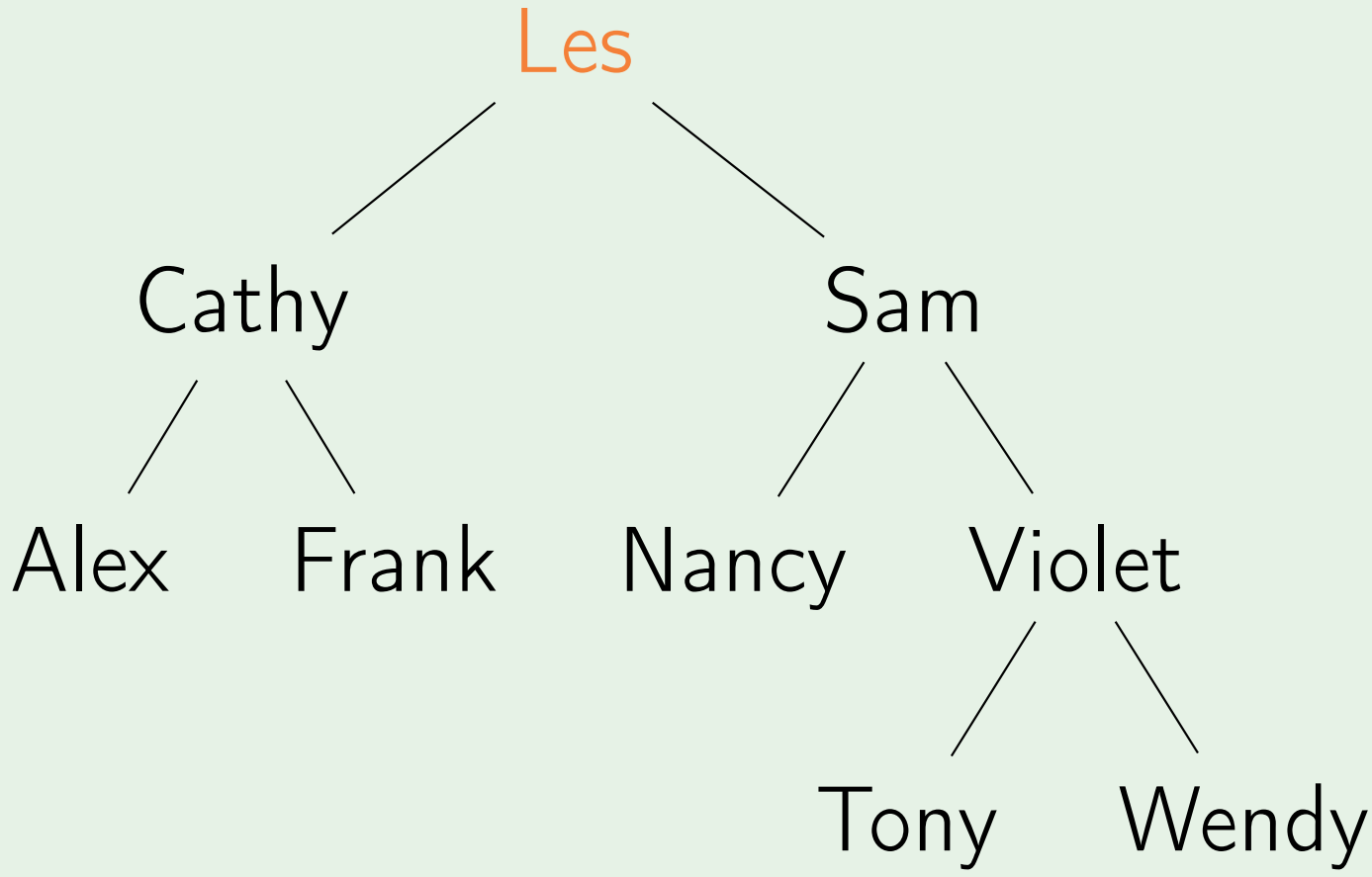


Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

Depth-first

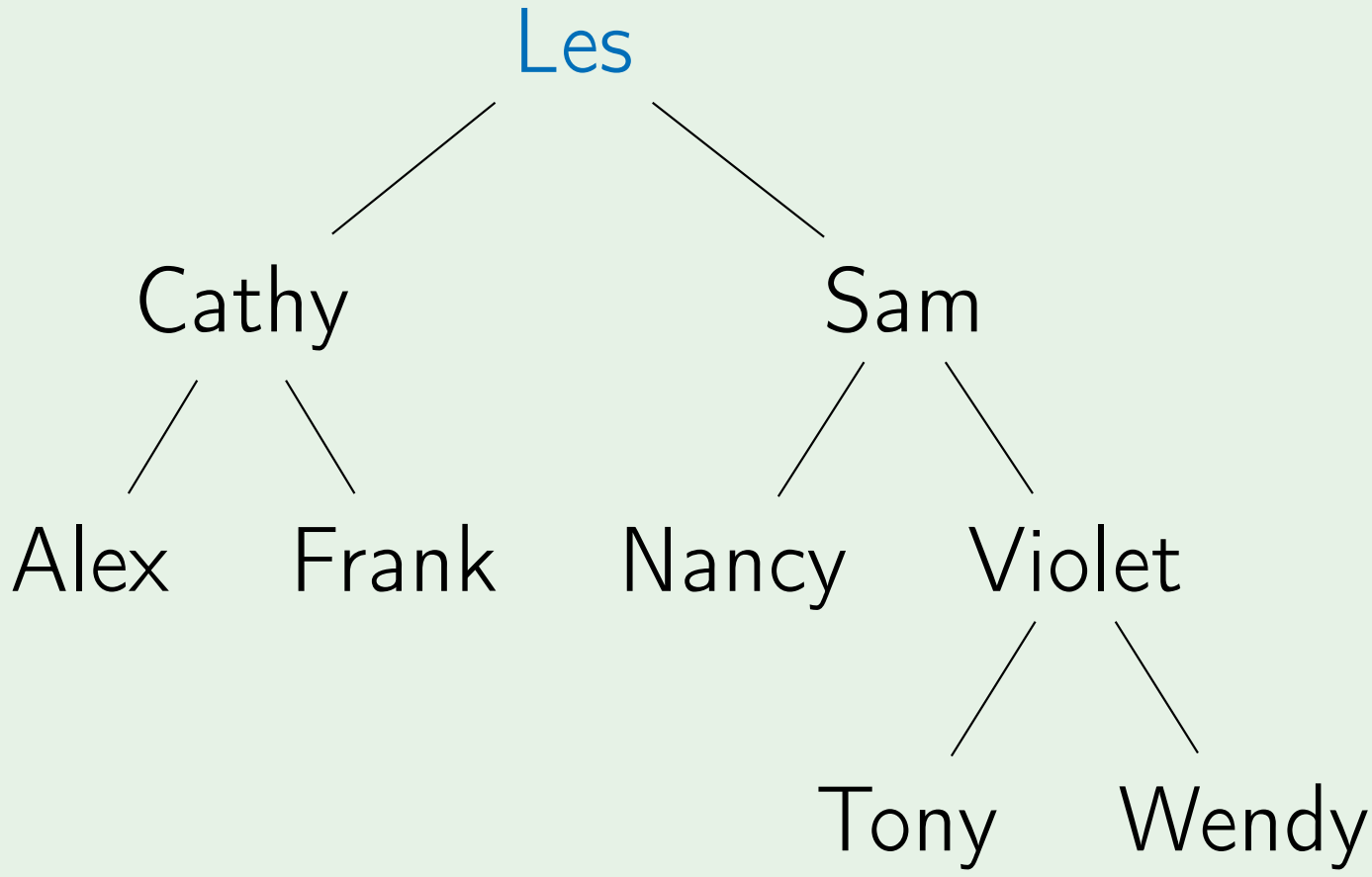
```
PreOrderTraversal(tree)
if tree = nil:
    return
Print(tree.key)
PreOrderTraversal(tree.left)
PreOrderTraversal(tree.right)
```

PreOrderTraversal



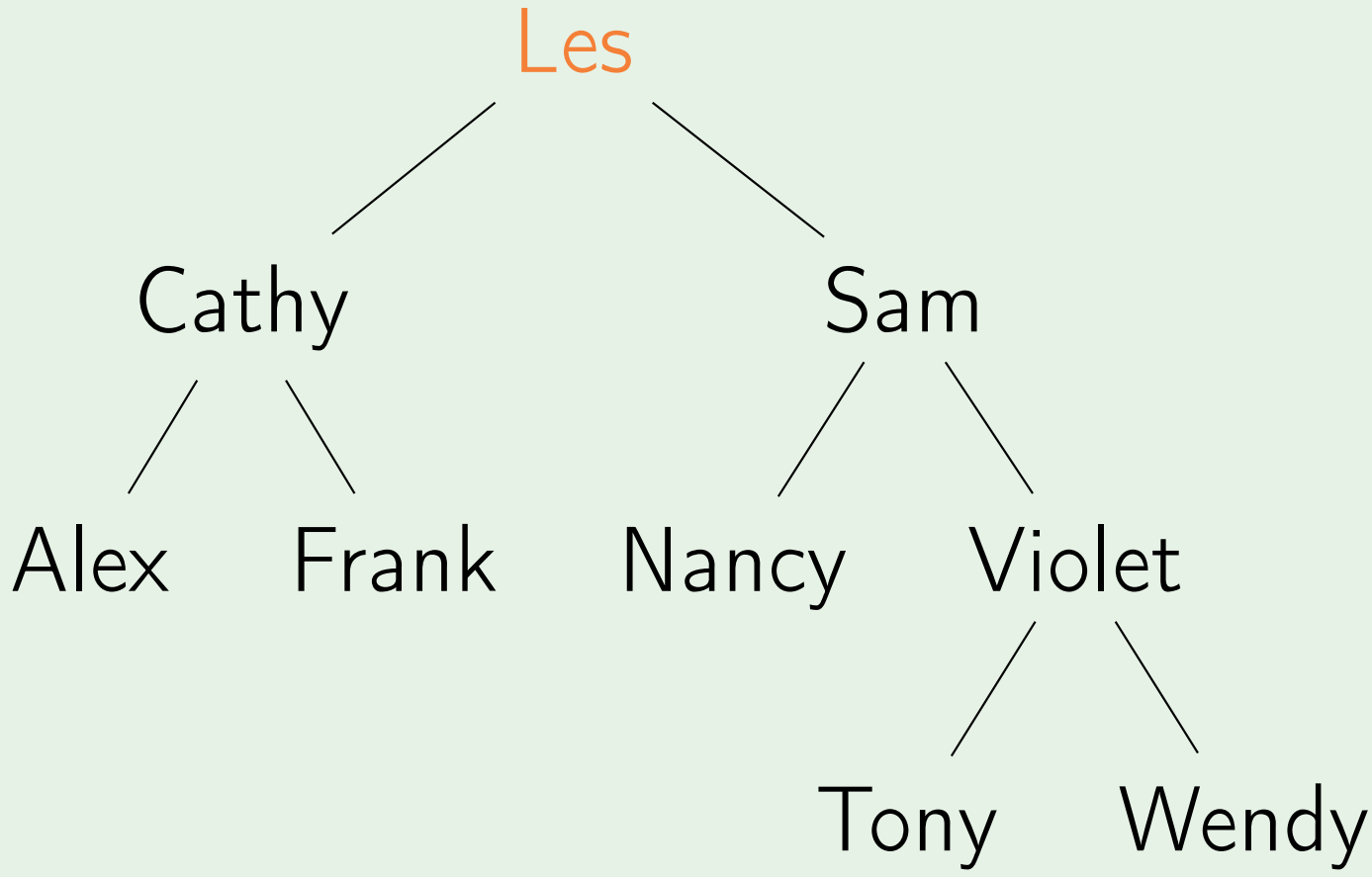
Output:

PreOrderTraversal



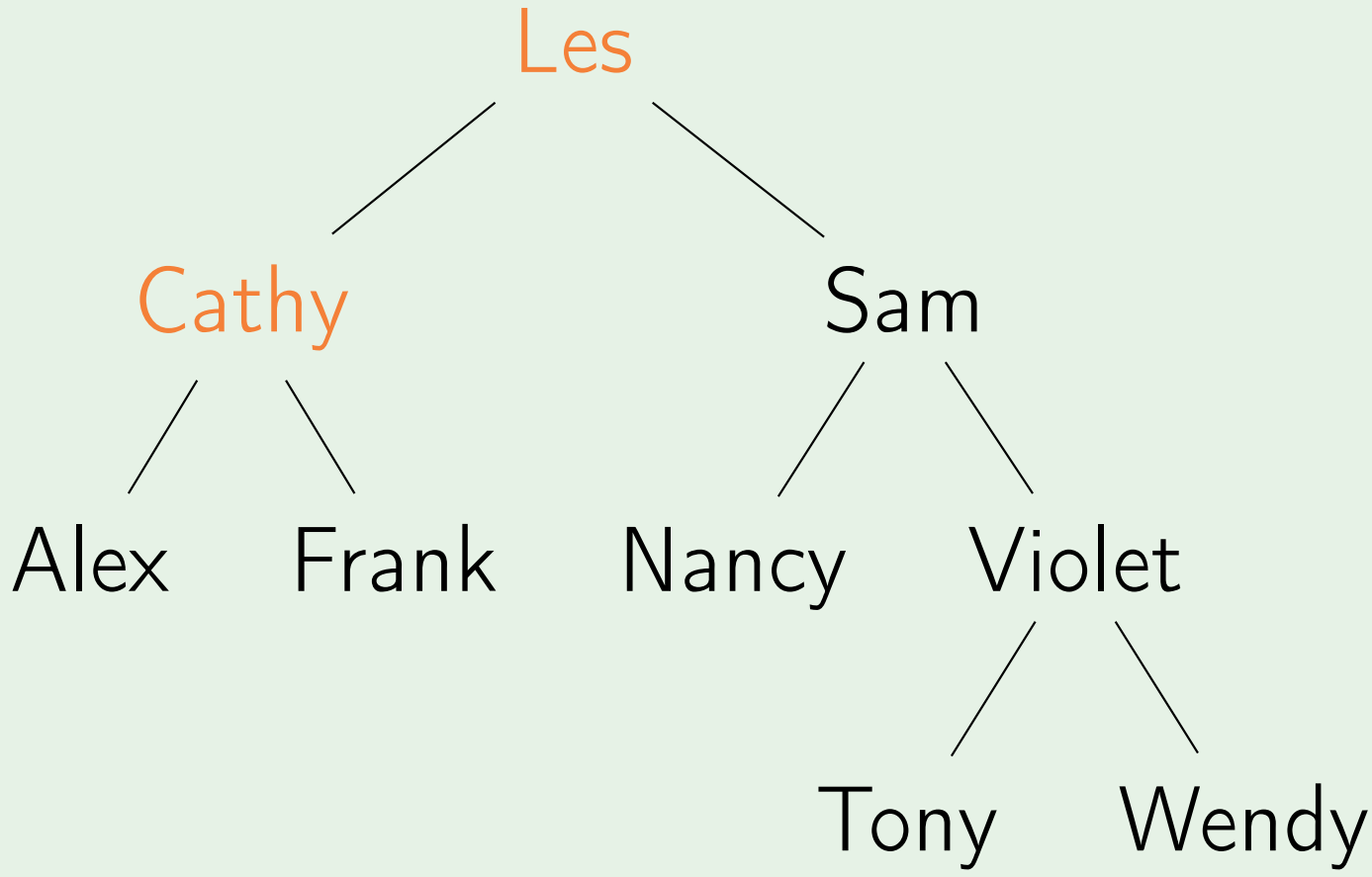
Output: Les

PreOrderTraversal



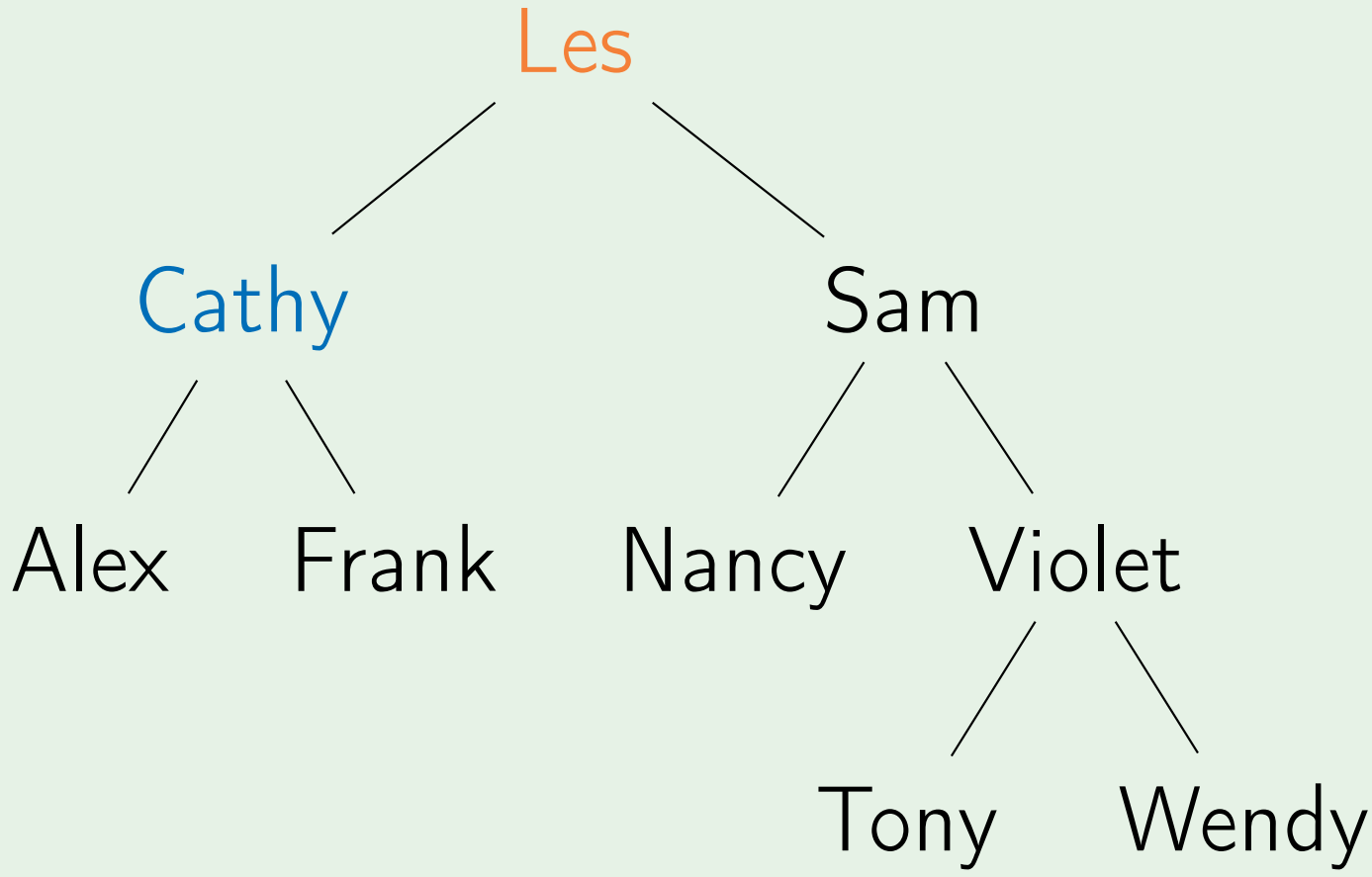
Output: Les

PreOrderTraversal



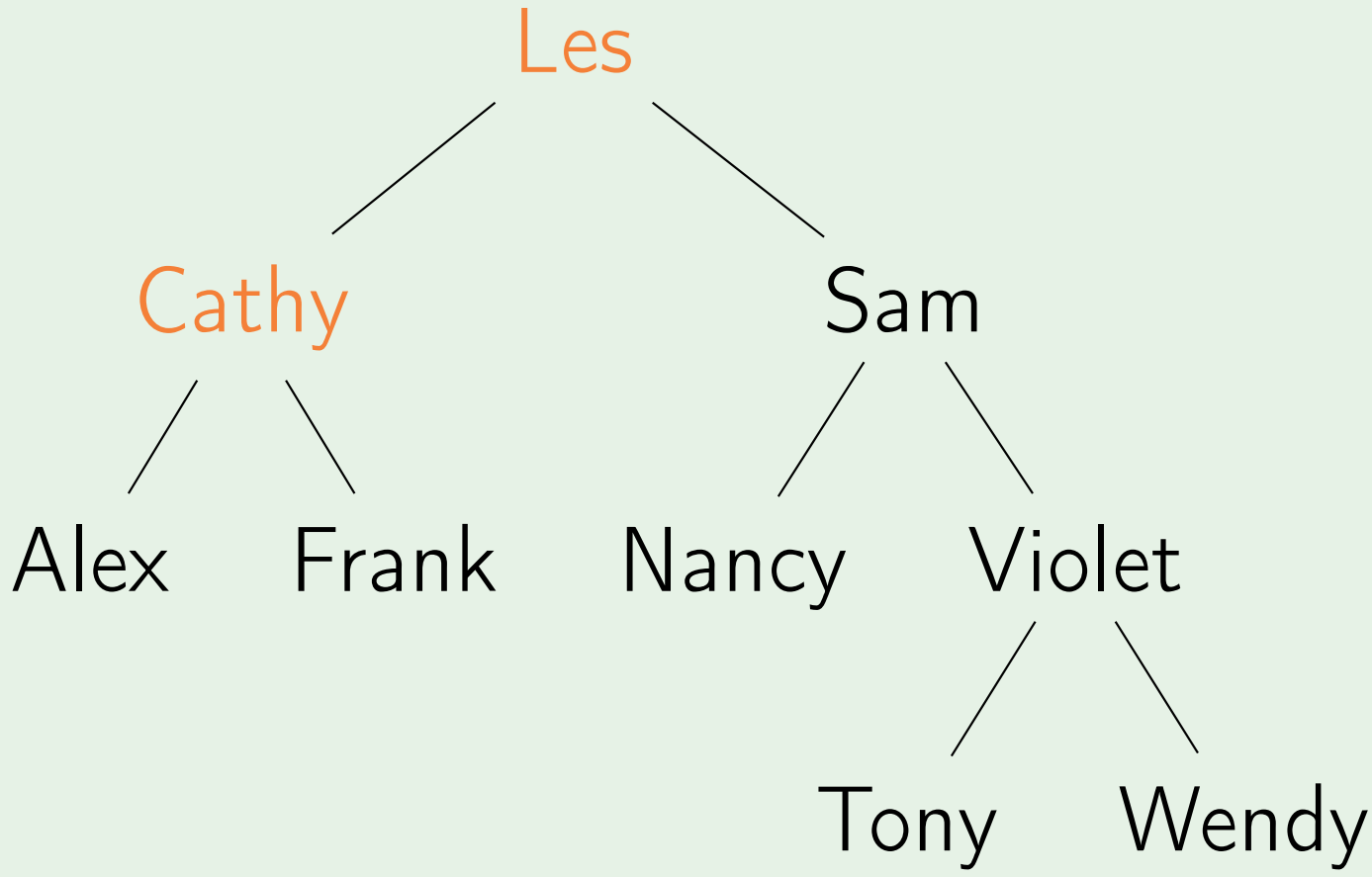
Output: Les

PreOrderTraversal



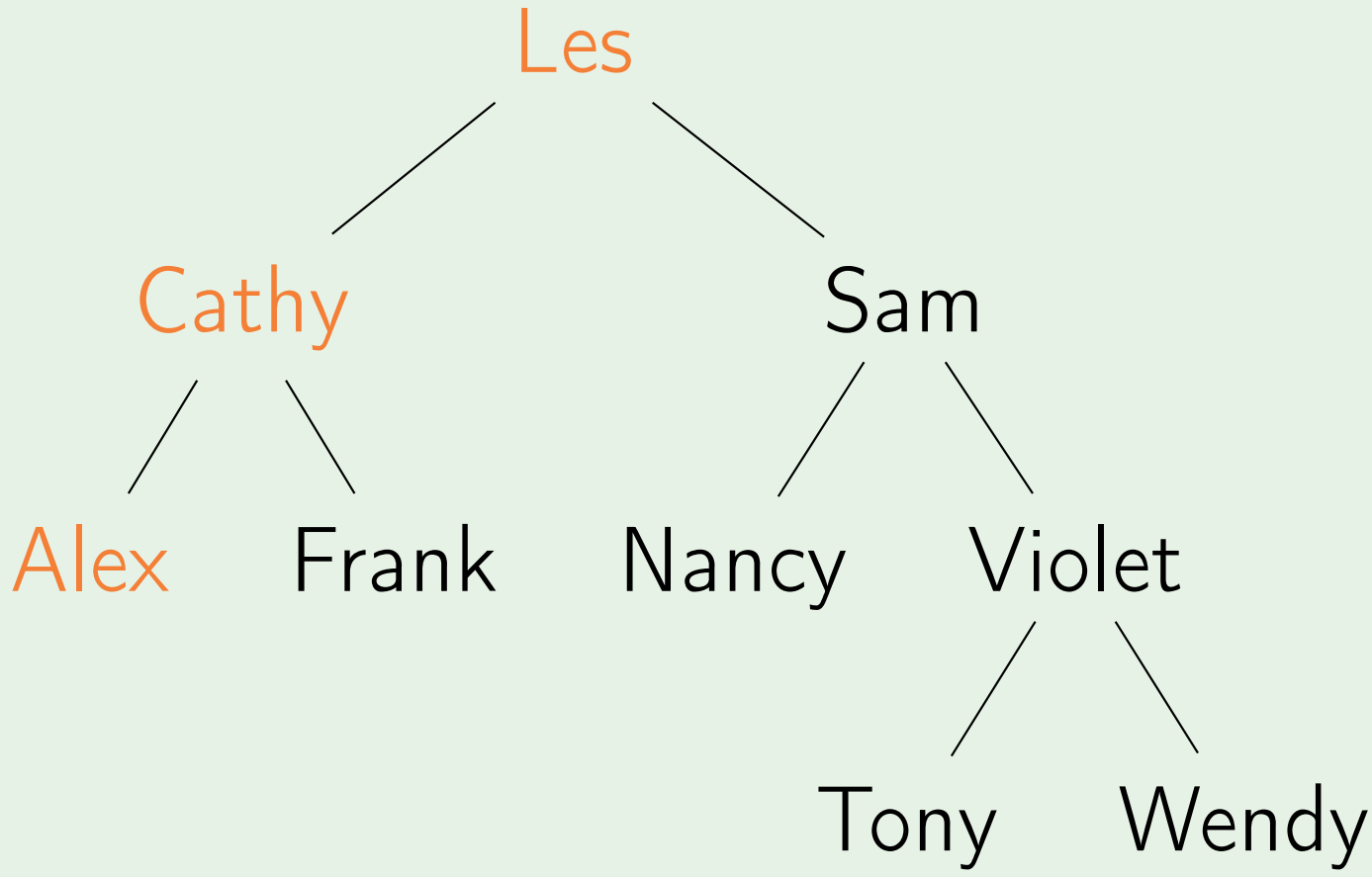
Output: Les Cathy

PreOrderTraversal



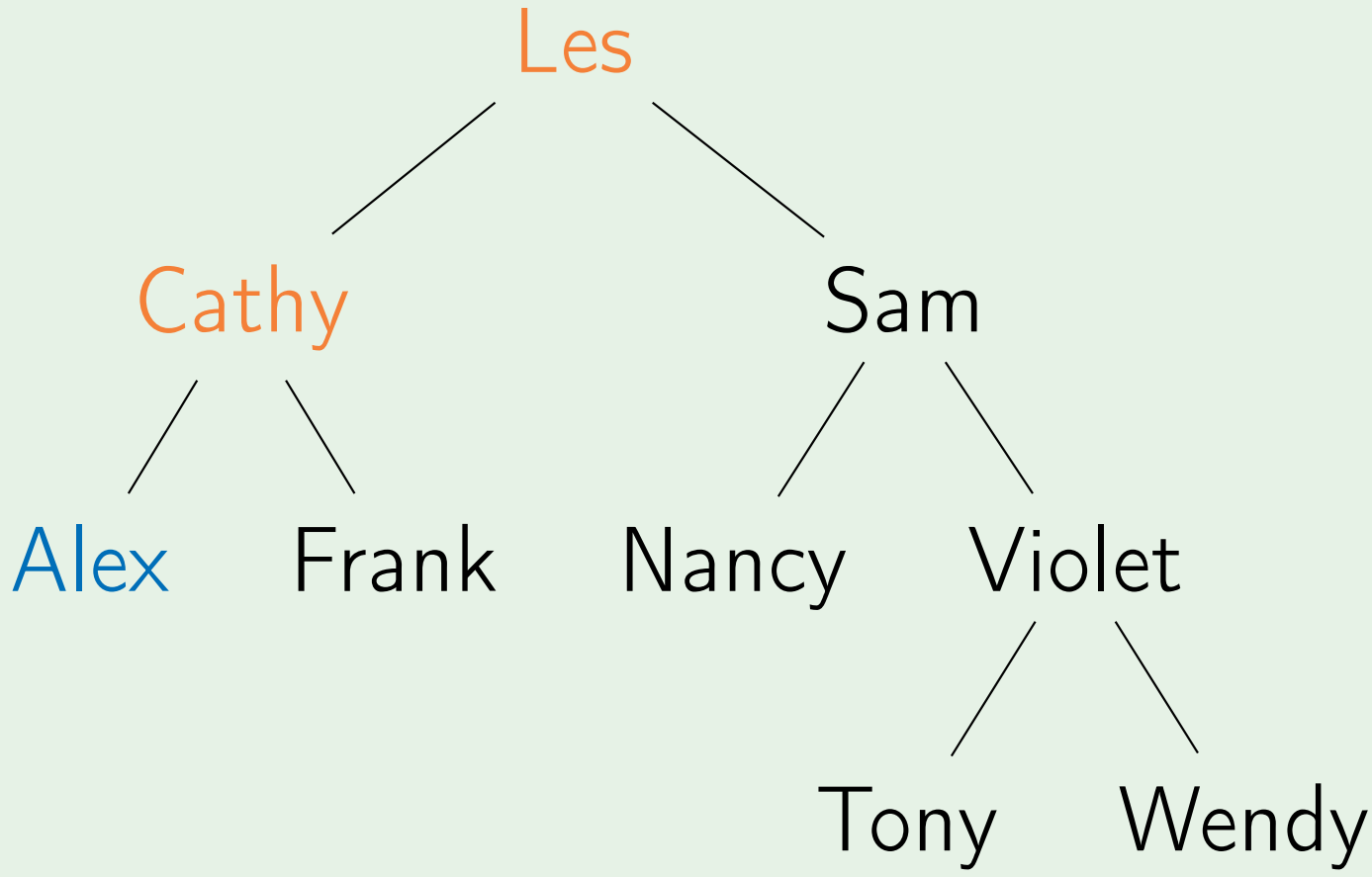
Output: Les Cathy

PreOrderTraversal



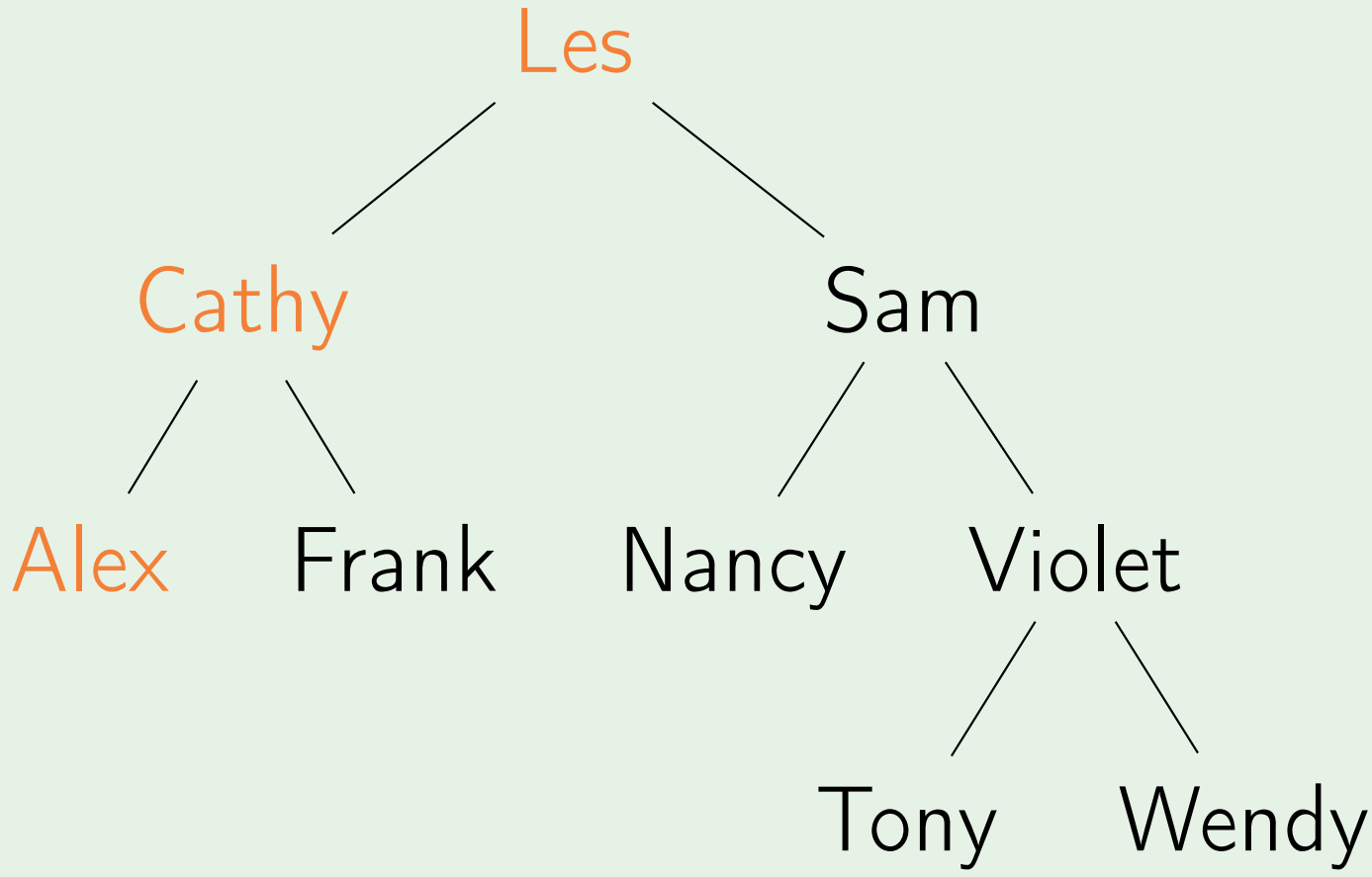
Output: Les Cathy

PreOrderTraversal



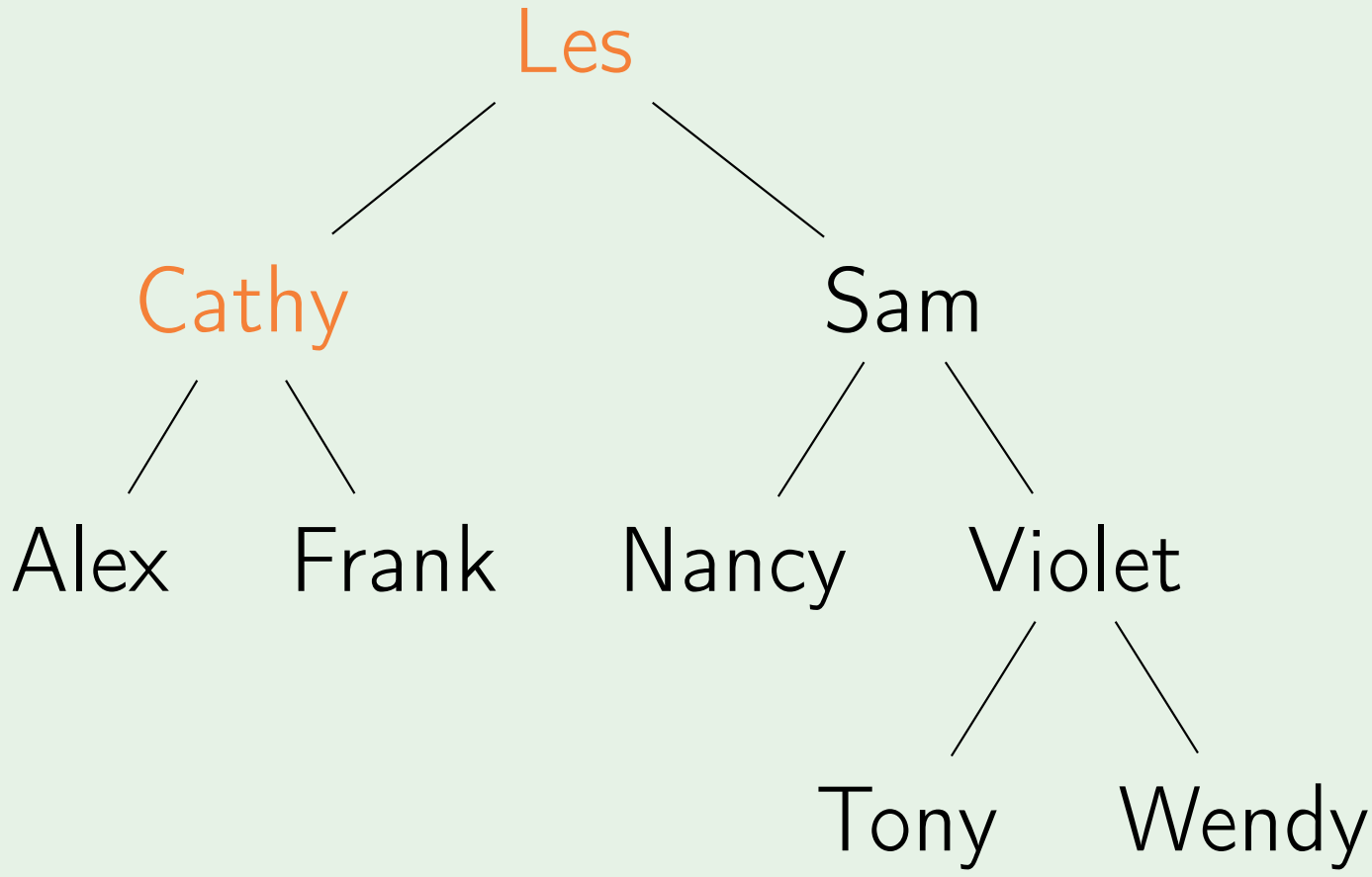
Output: Les Cathy Alex

PreOrderTraversal



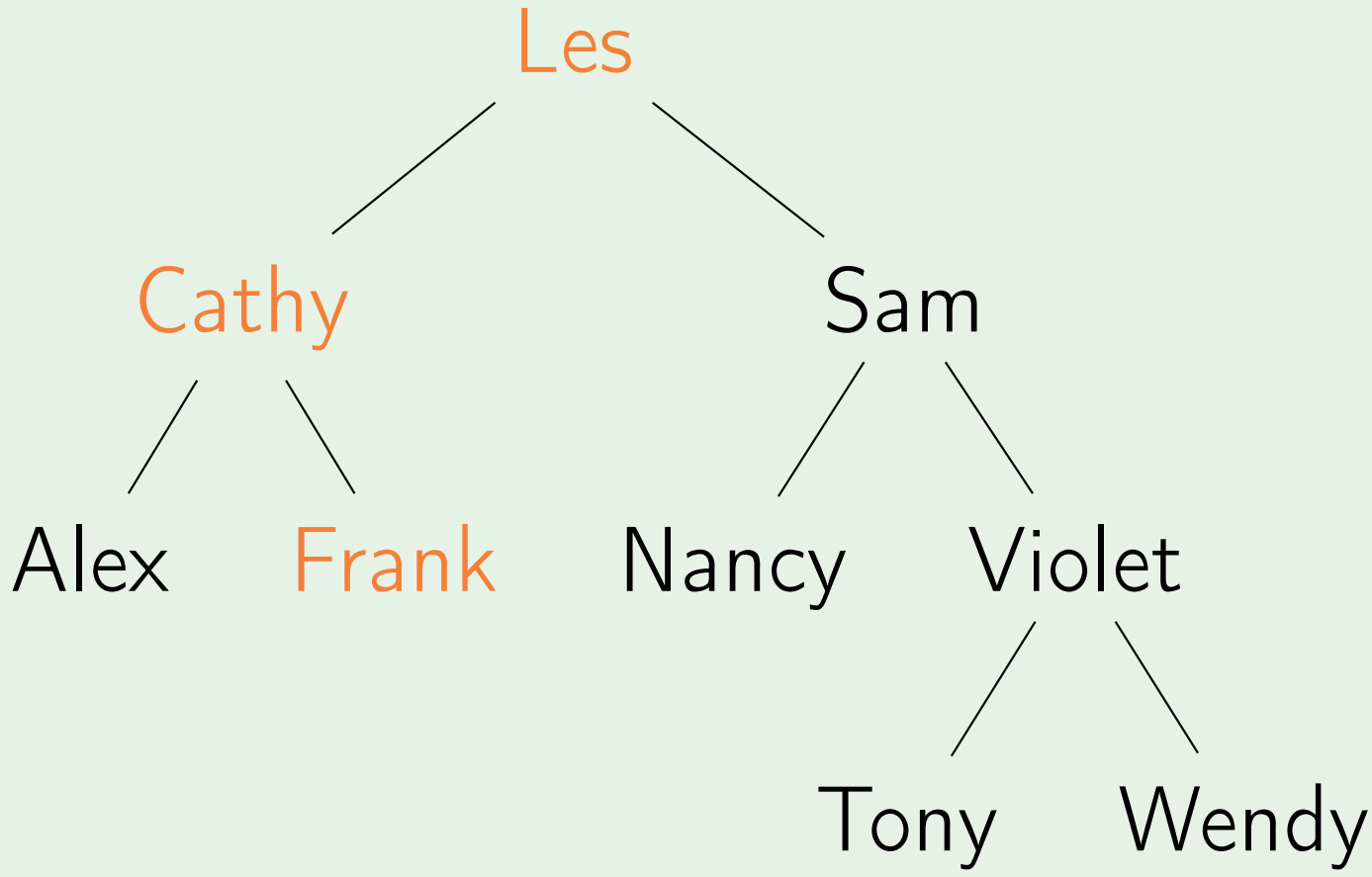
Output: Les Cathy Alex

PreOrderTraversal



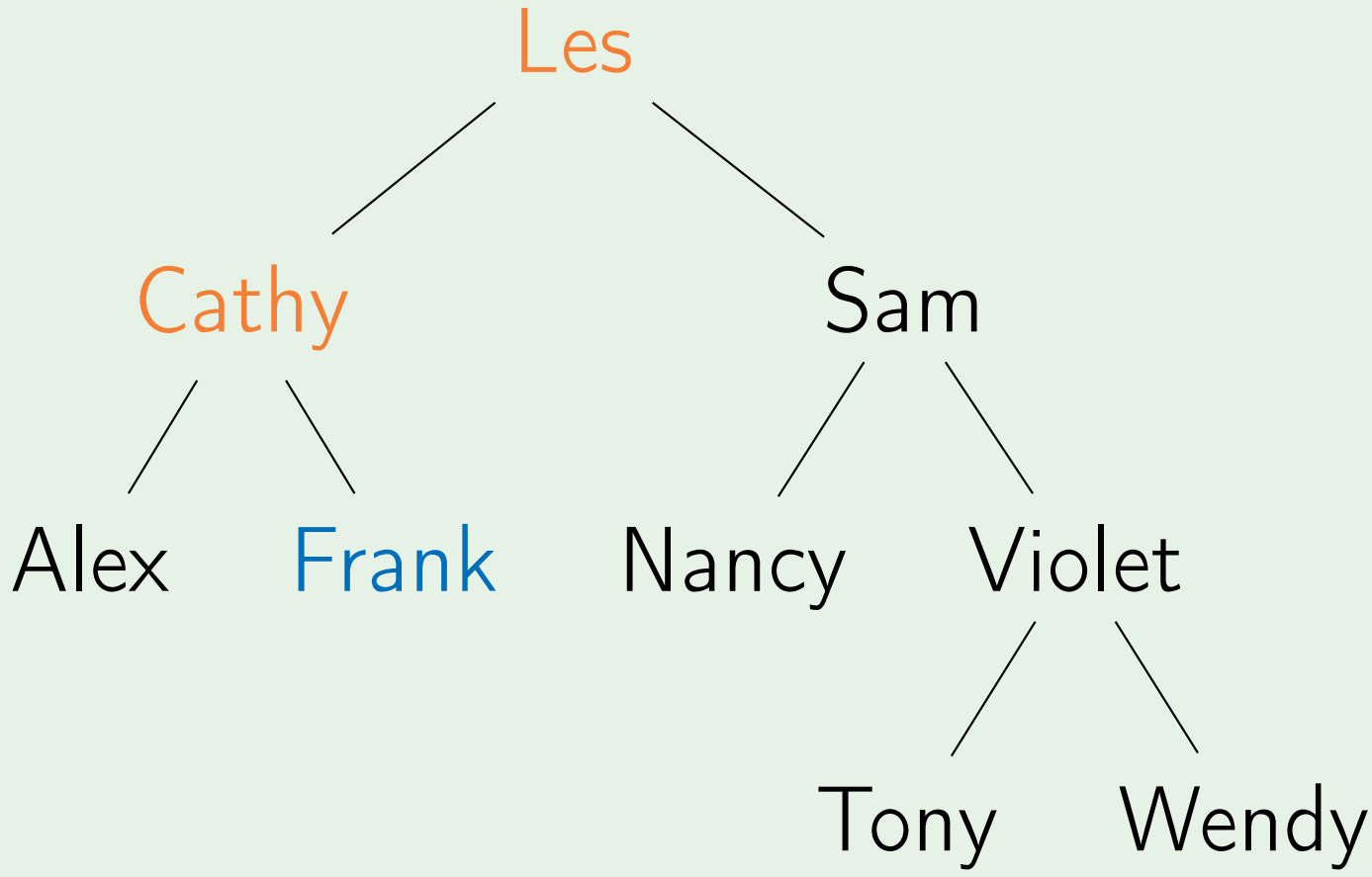
Output: Les Cathy Alex

PreOrderTraversal



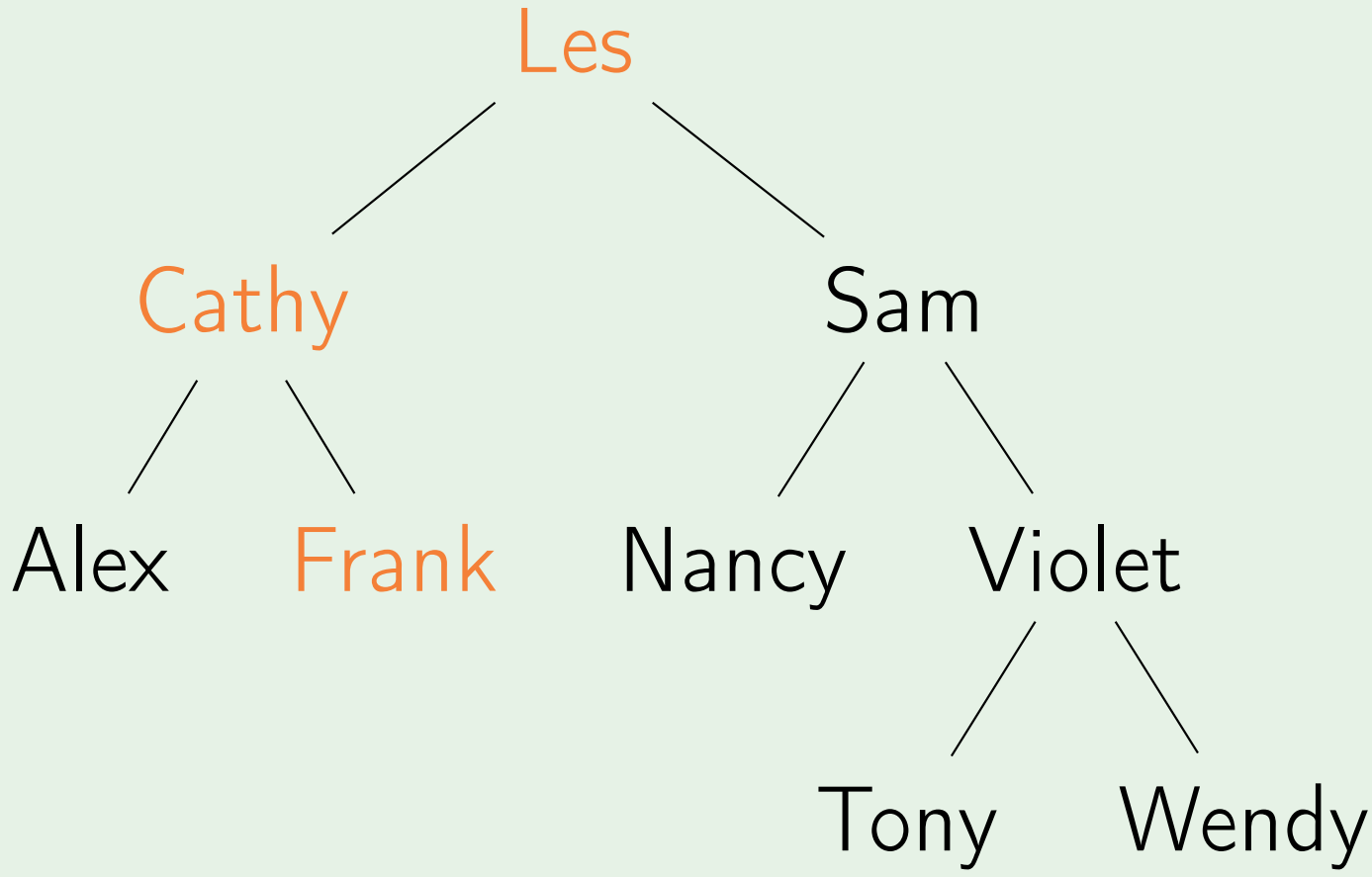
Output: Les Cathy Alex

PreOrderTraversal



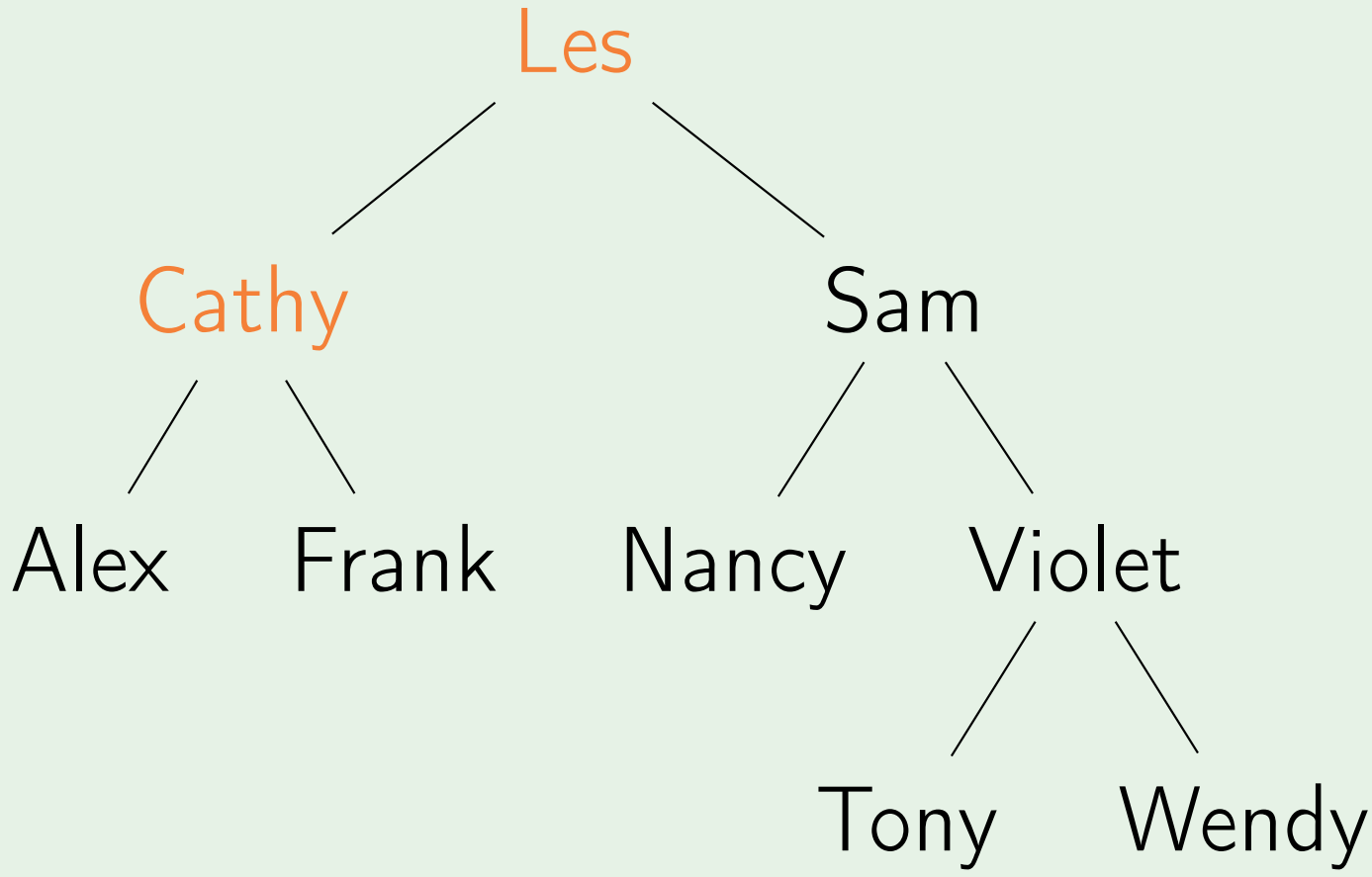
Output: Les Cathy Alex Frank

PreOrderTraversal



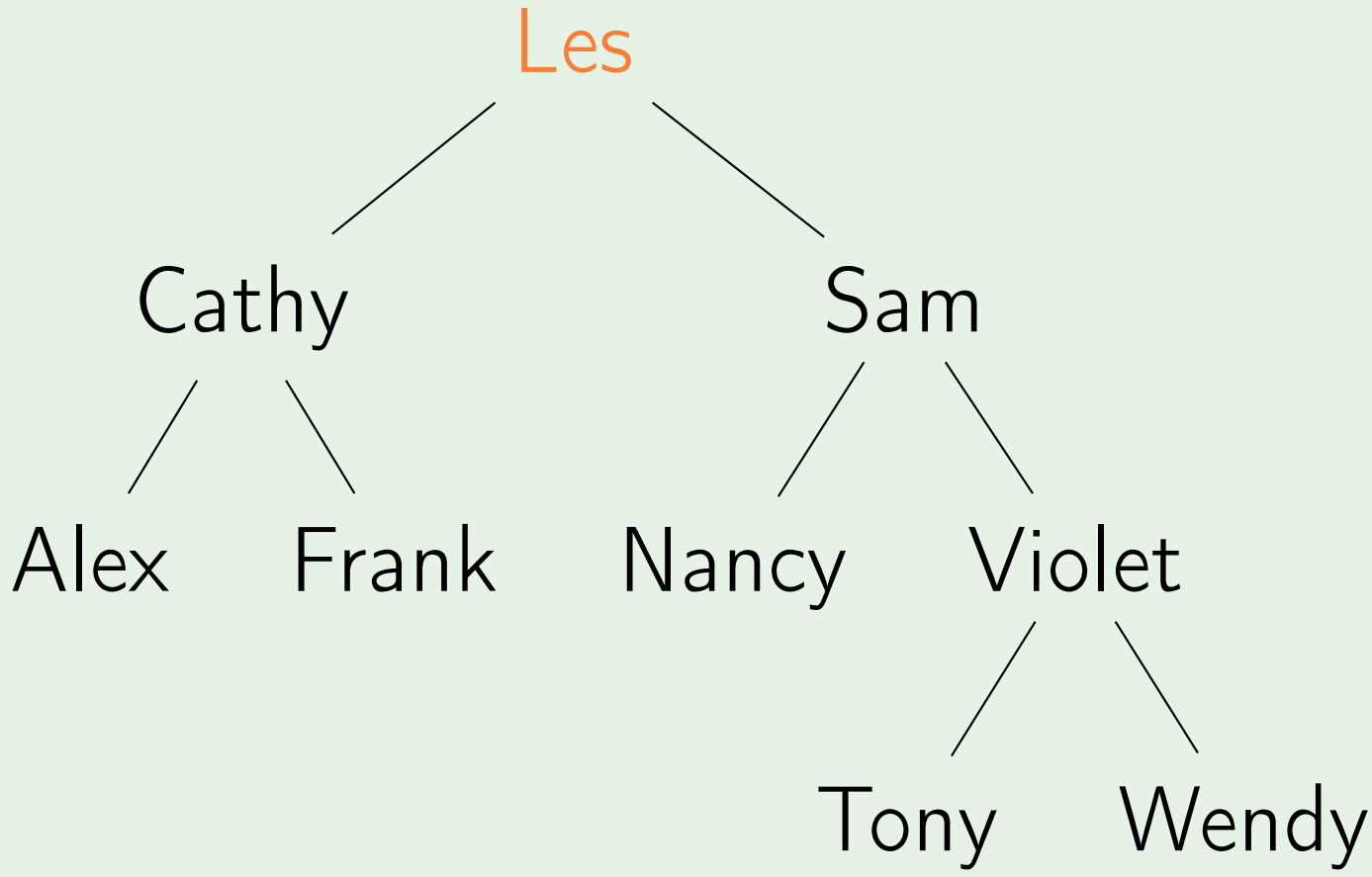
Output: Les Cathy Alex Frank

PreOrderTraversal



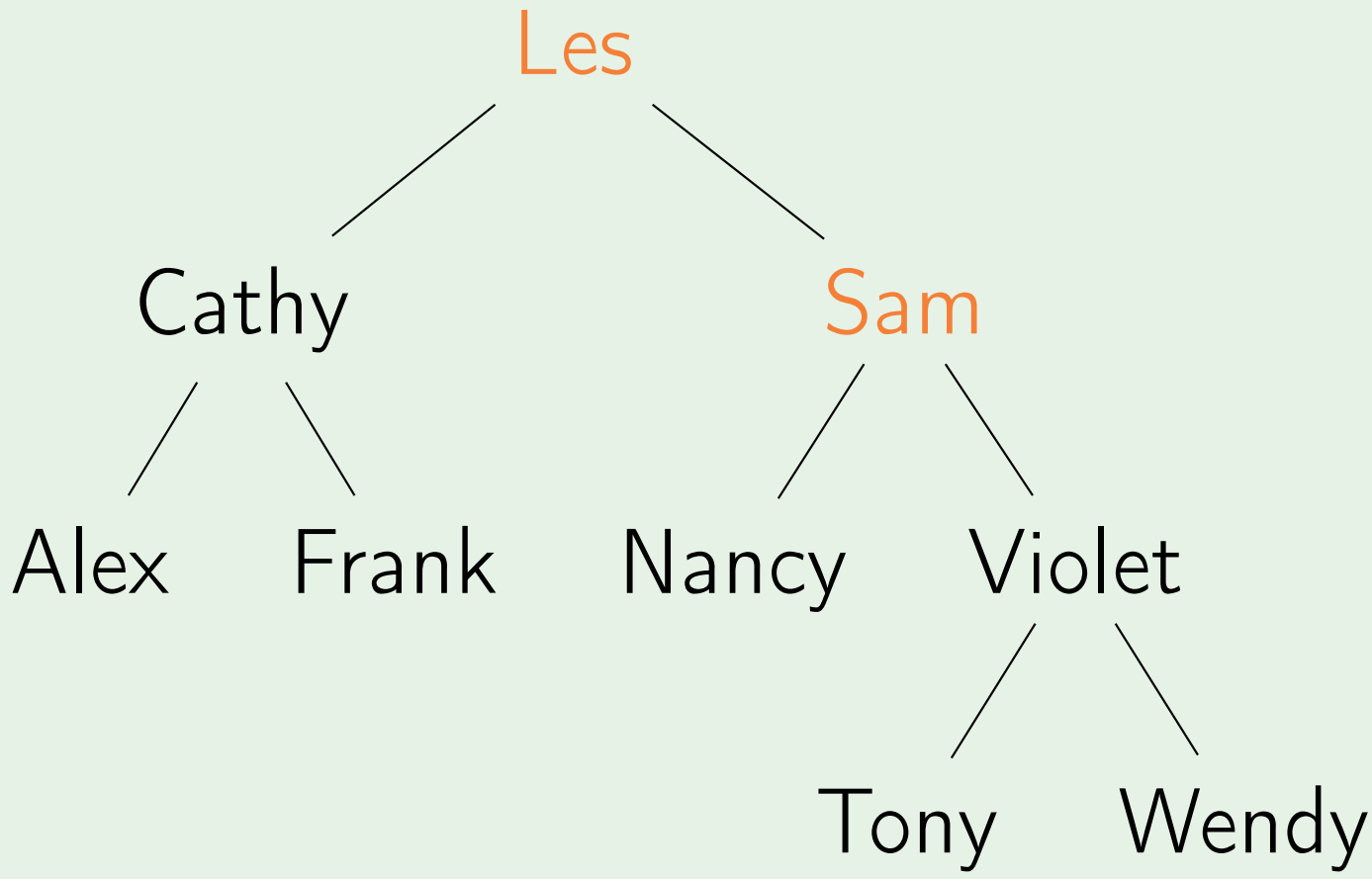
Output: Les Cathy Alex Frank

PreOrderTraversal



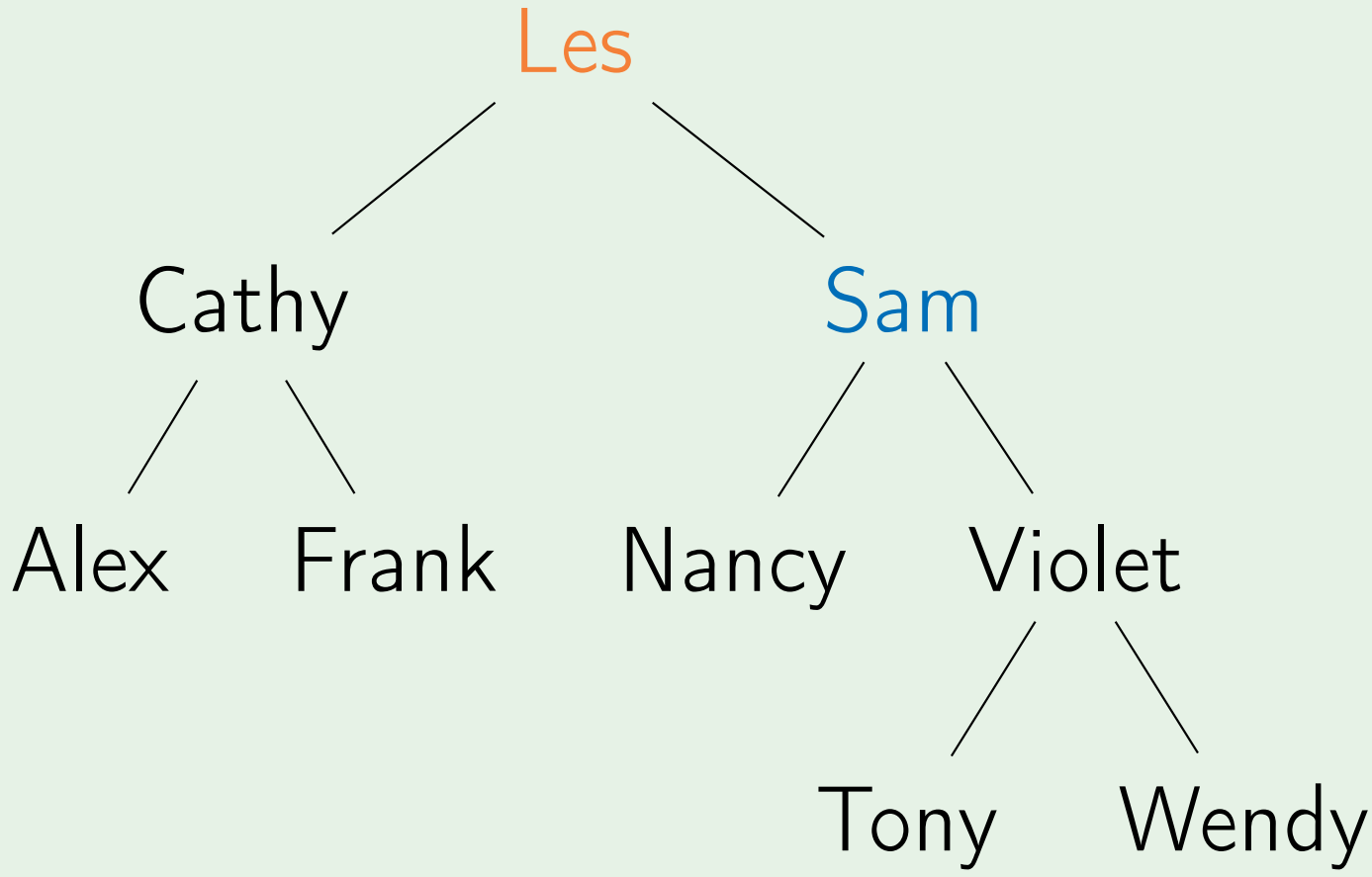
Output: Les Cathy Alex Frank

PreOrderTraversal



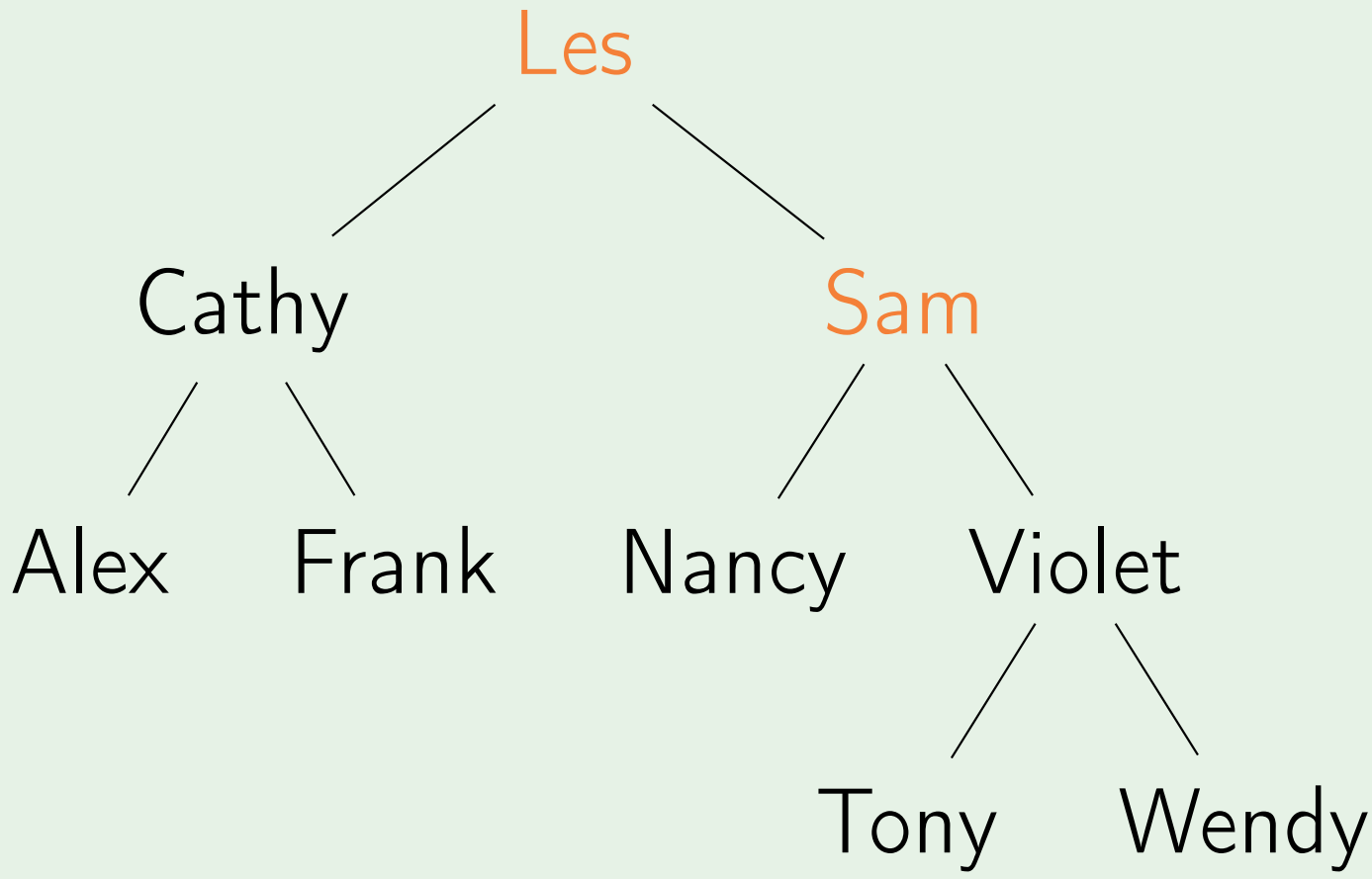
Output: Les Cathy Alex Frank

PreOrderTraversal



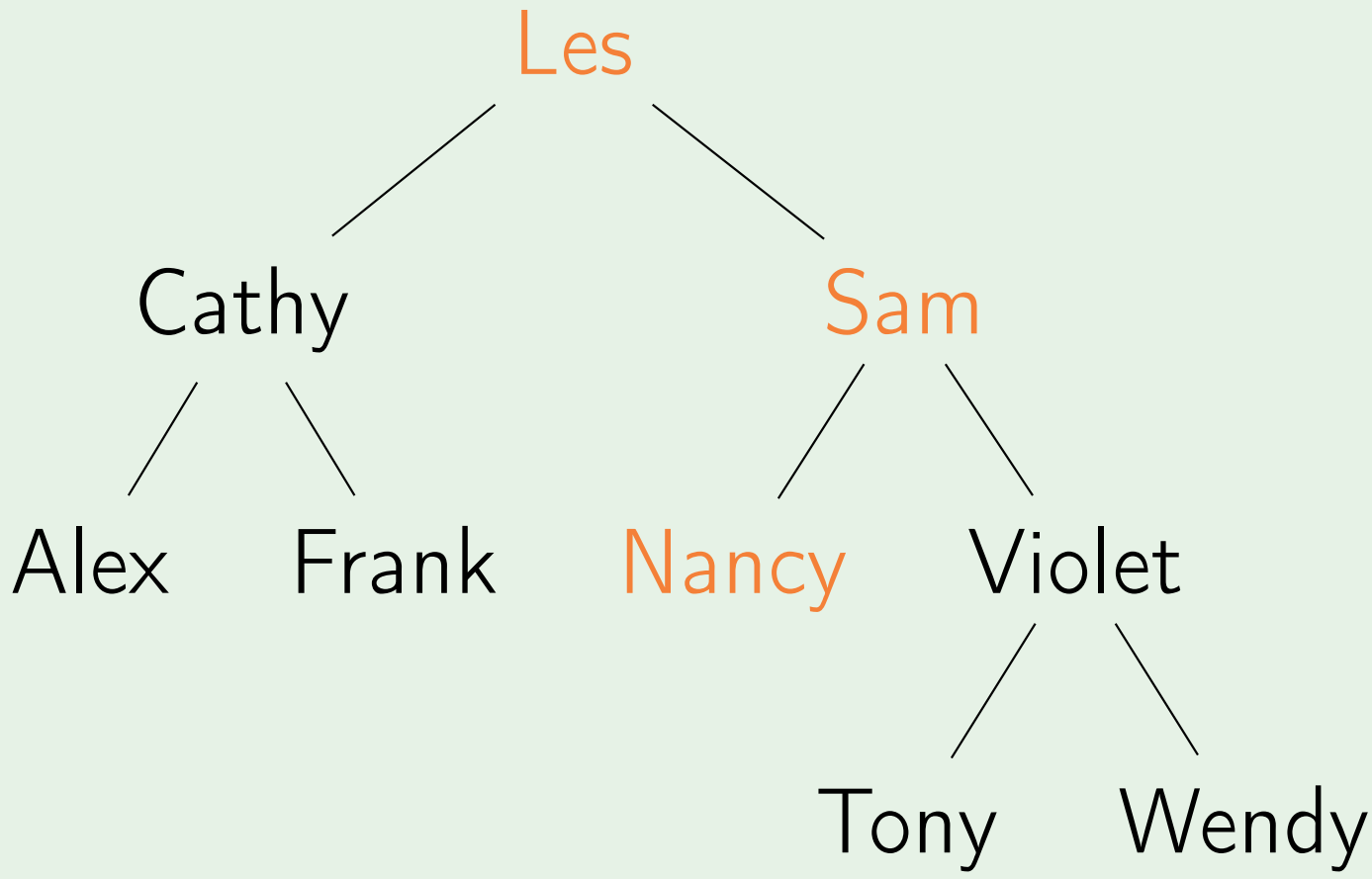
Output: Les Cathy Alex Frank Sam

PreOrderTraversal



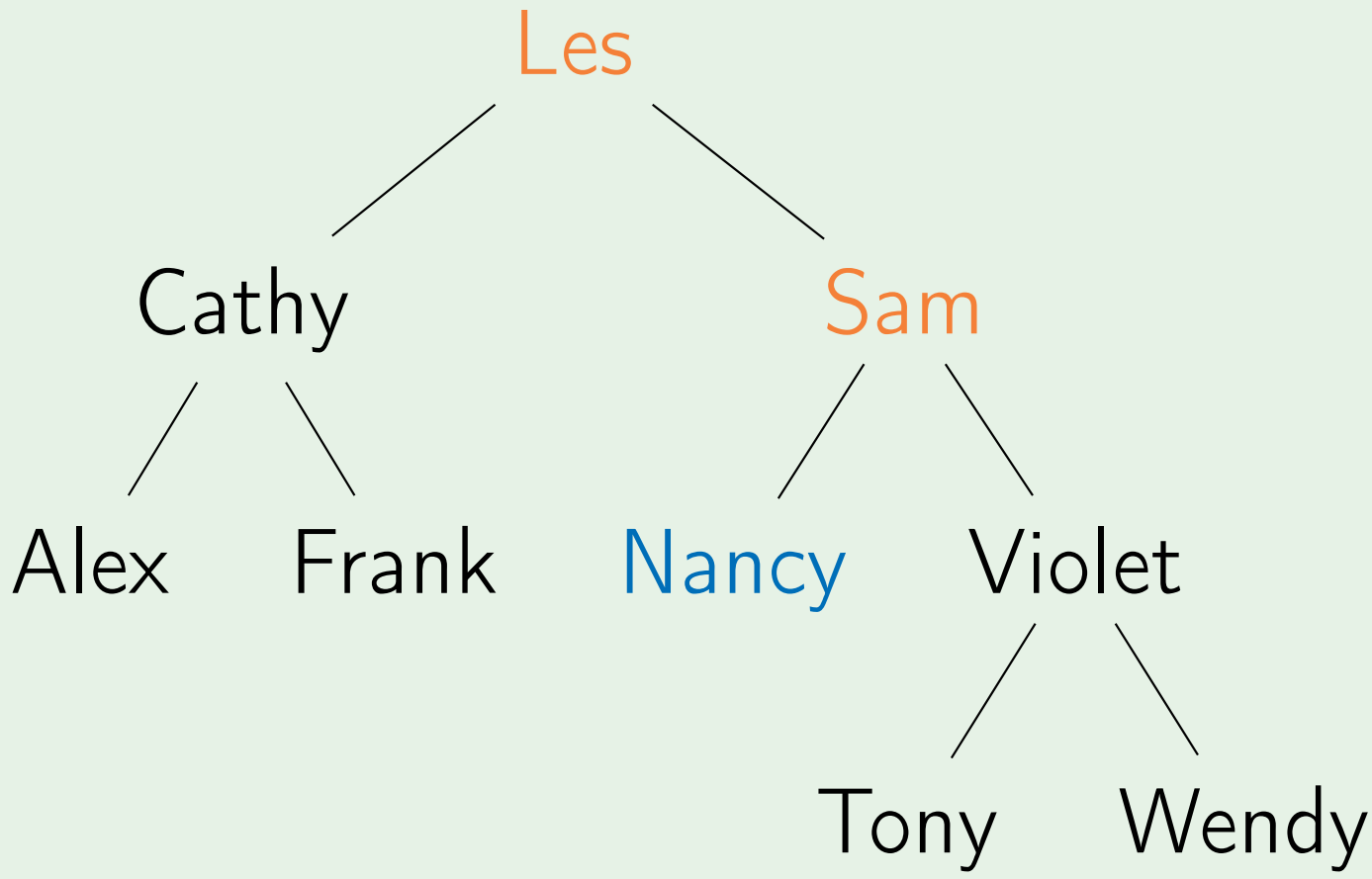
Output: Les Cathy Alex Frank Sam

PreOrderTraversal



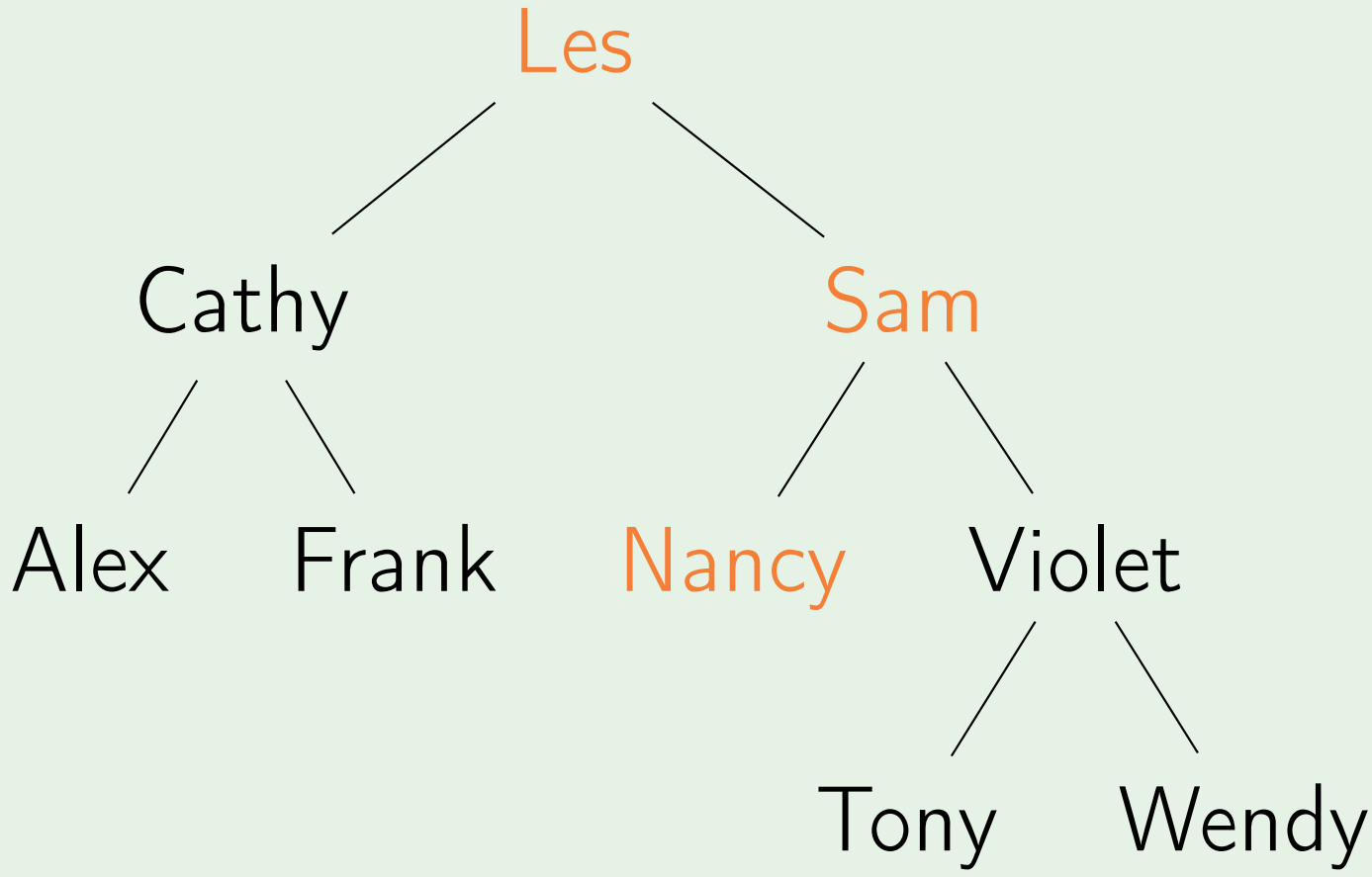
Output: Les Cathy Alex Frank Sam

PreOrderTraversal



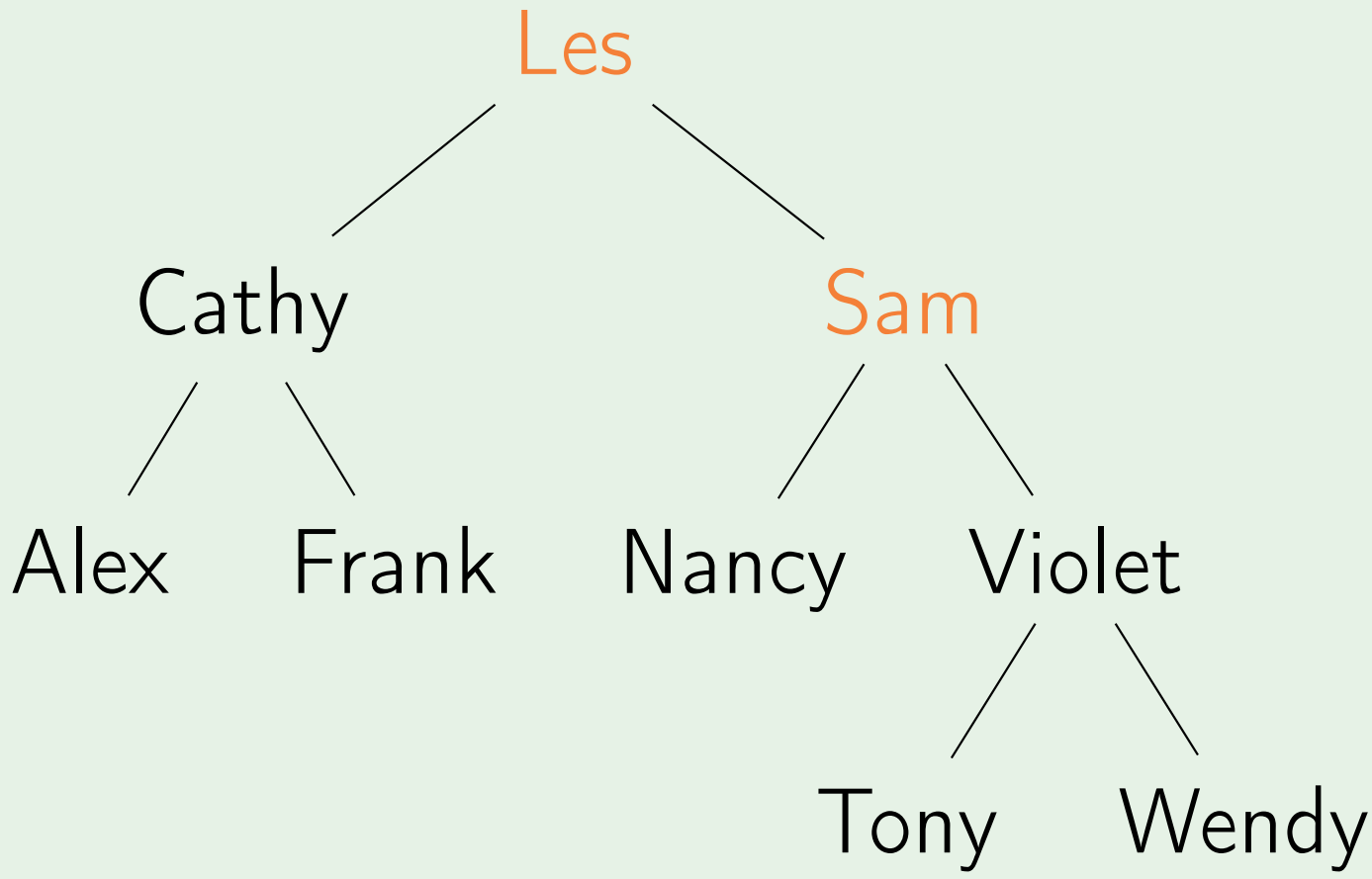
Output: Les Cathy Alex Frank Sam Nancy

PreOrderTraversal



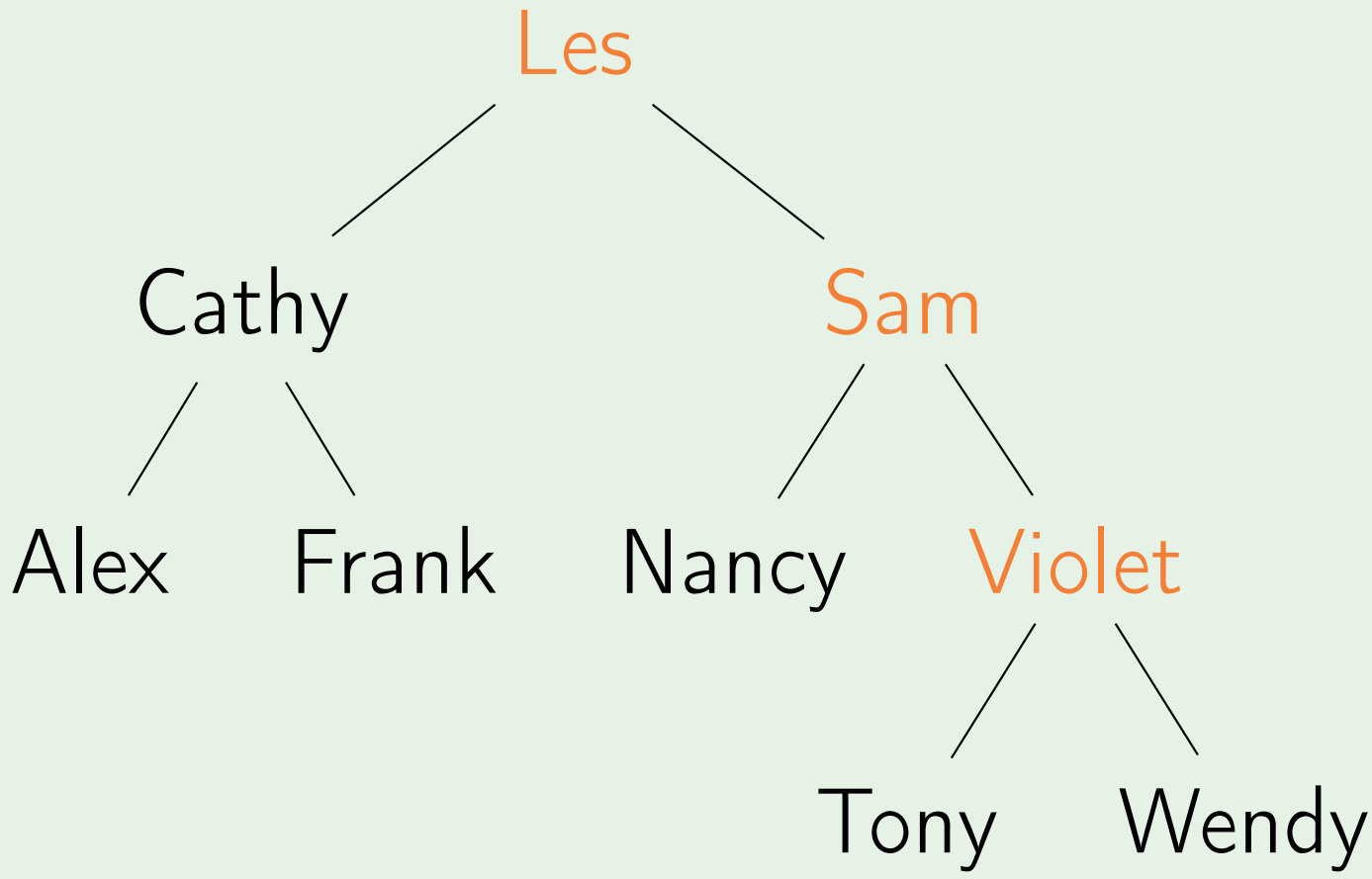
Output: Les Cathy Alex Frank Sam Nancy

PreOrderTraversal



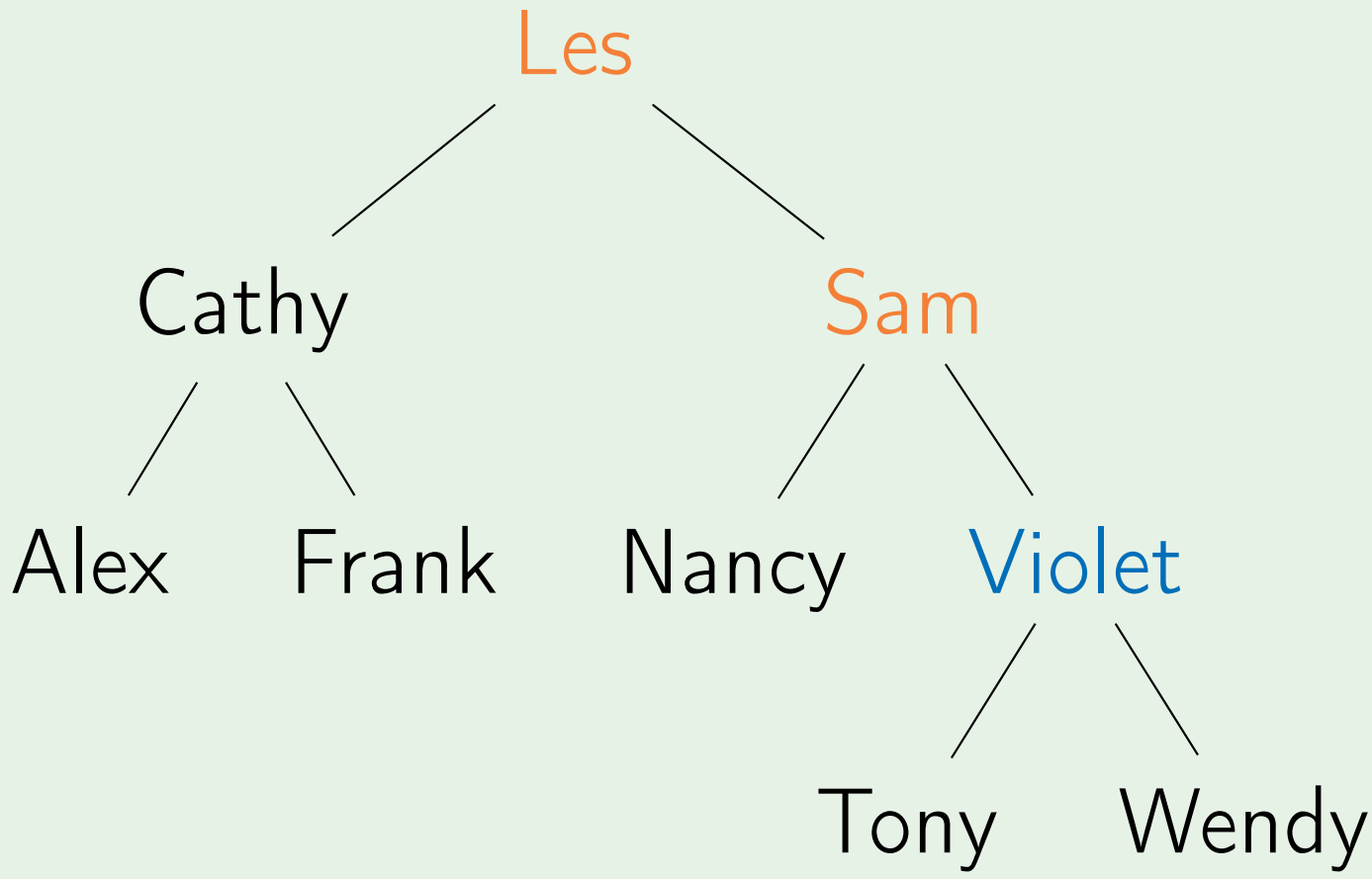
Output: Les Cathy Alex Frank Sam Nancy

PreOrderTraversal



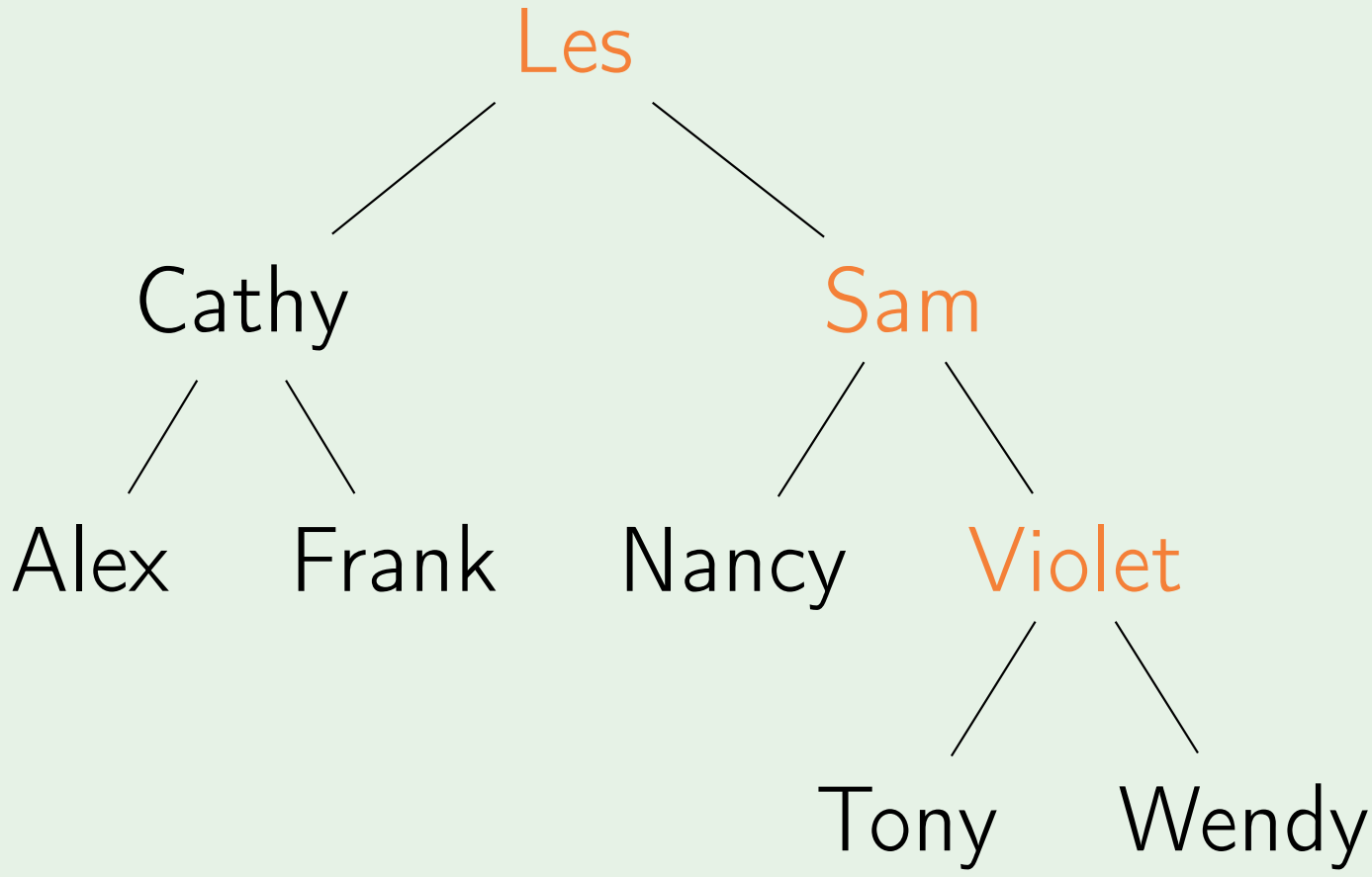
Output: Les Cathy Alex Frank Sam Nancy

PreOrderTraversal



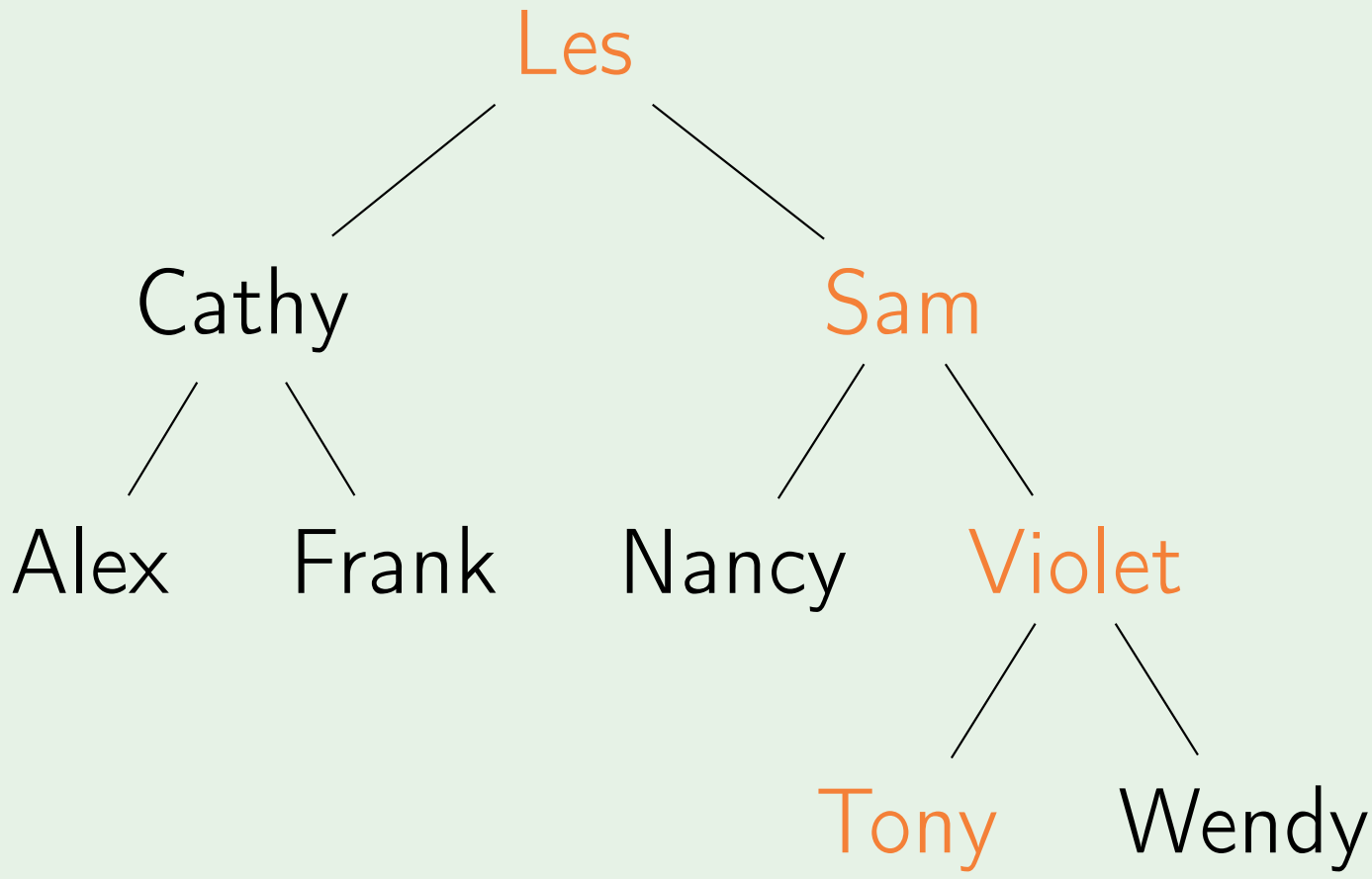
Output: Les Cathy Alex Frank Sam Nancy Violet

PreOrderTraversal



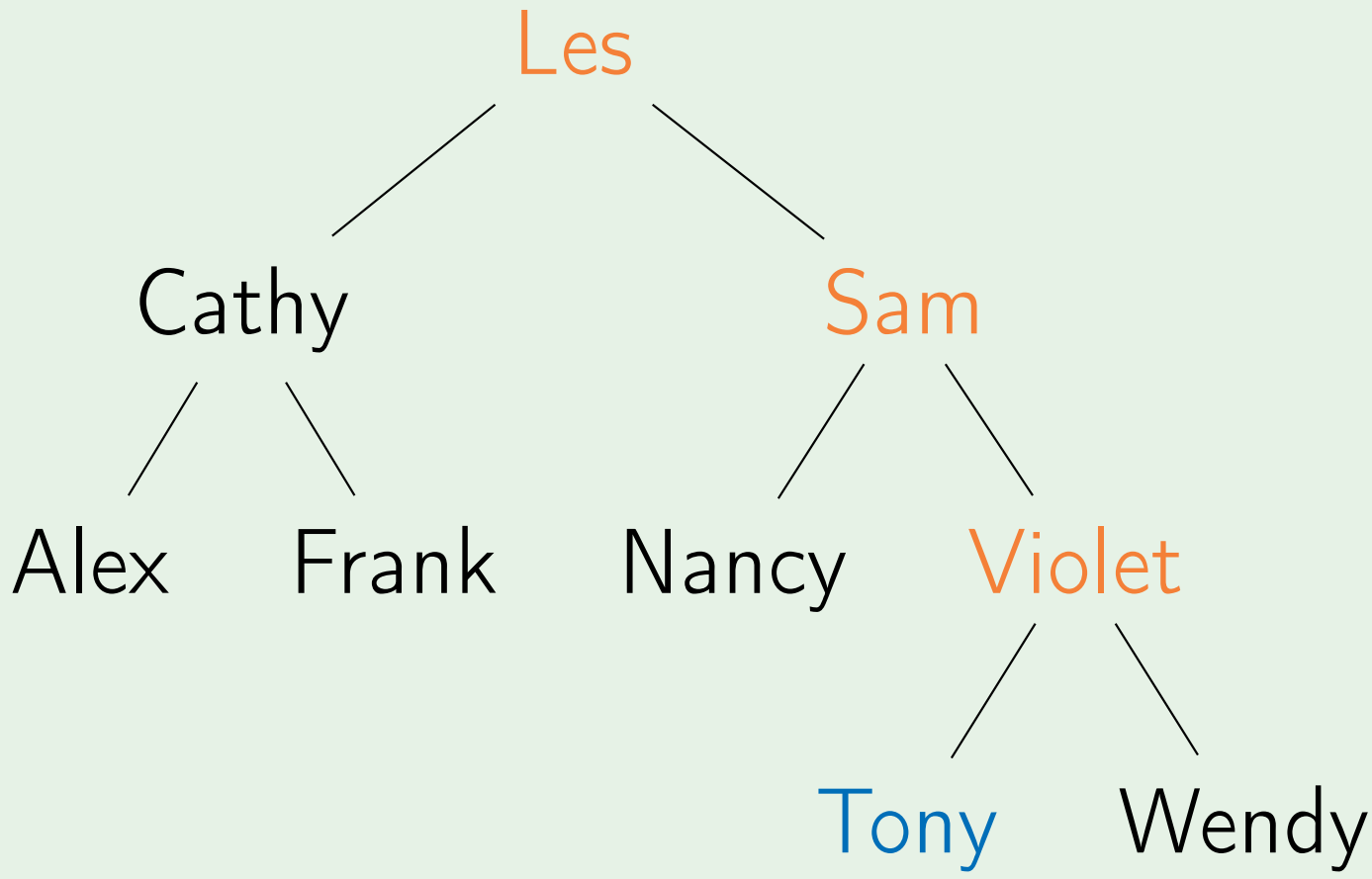
Output: Les Cathy Alex Frank Sam Nancy Violet

PreOrderTraversal



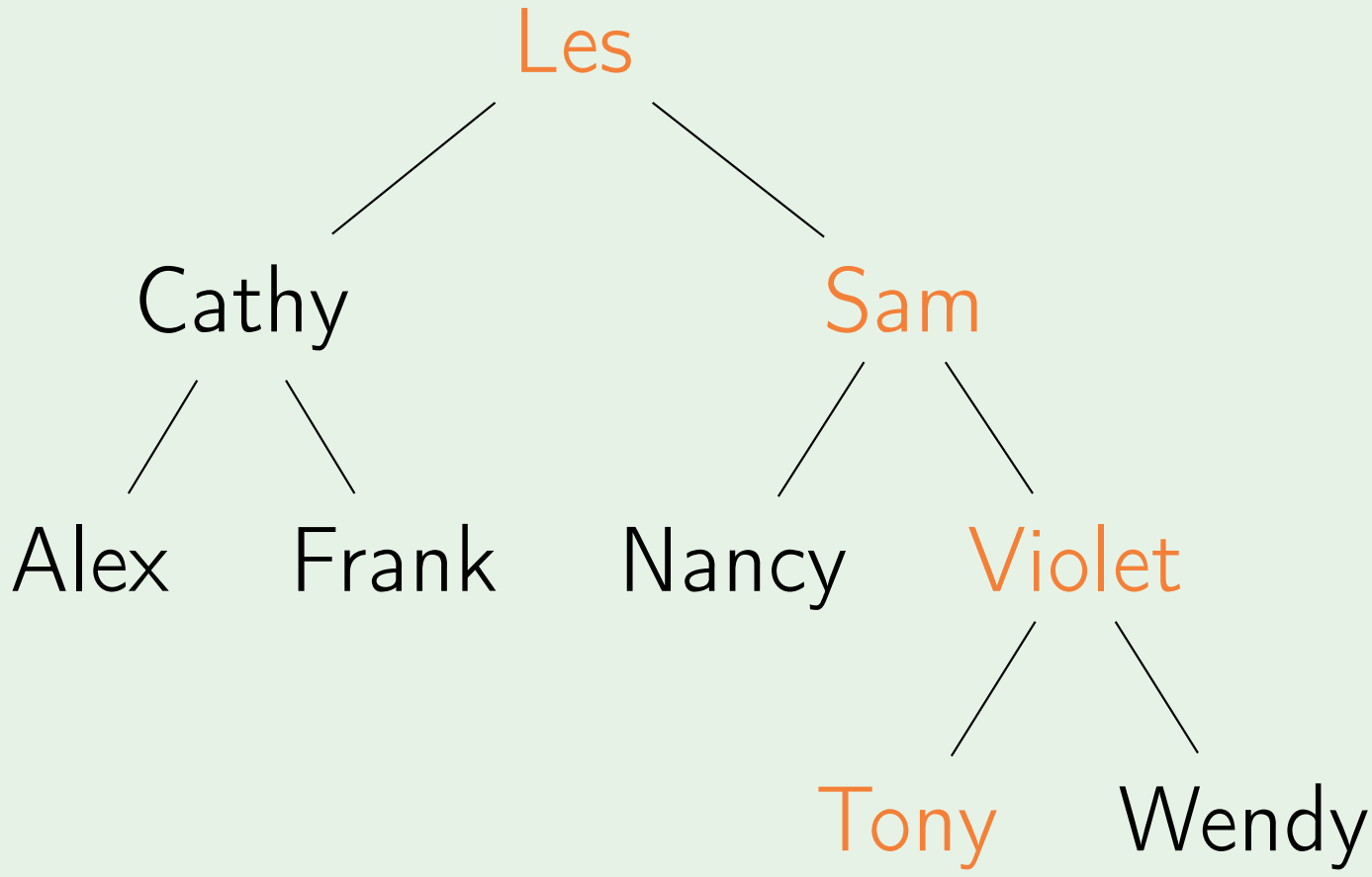
Output: Les Cathy Alex Frank Sam Nancy Violet

PreOrderTraversal



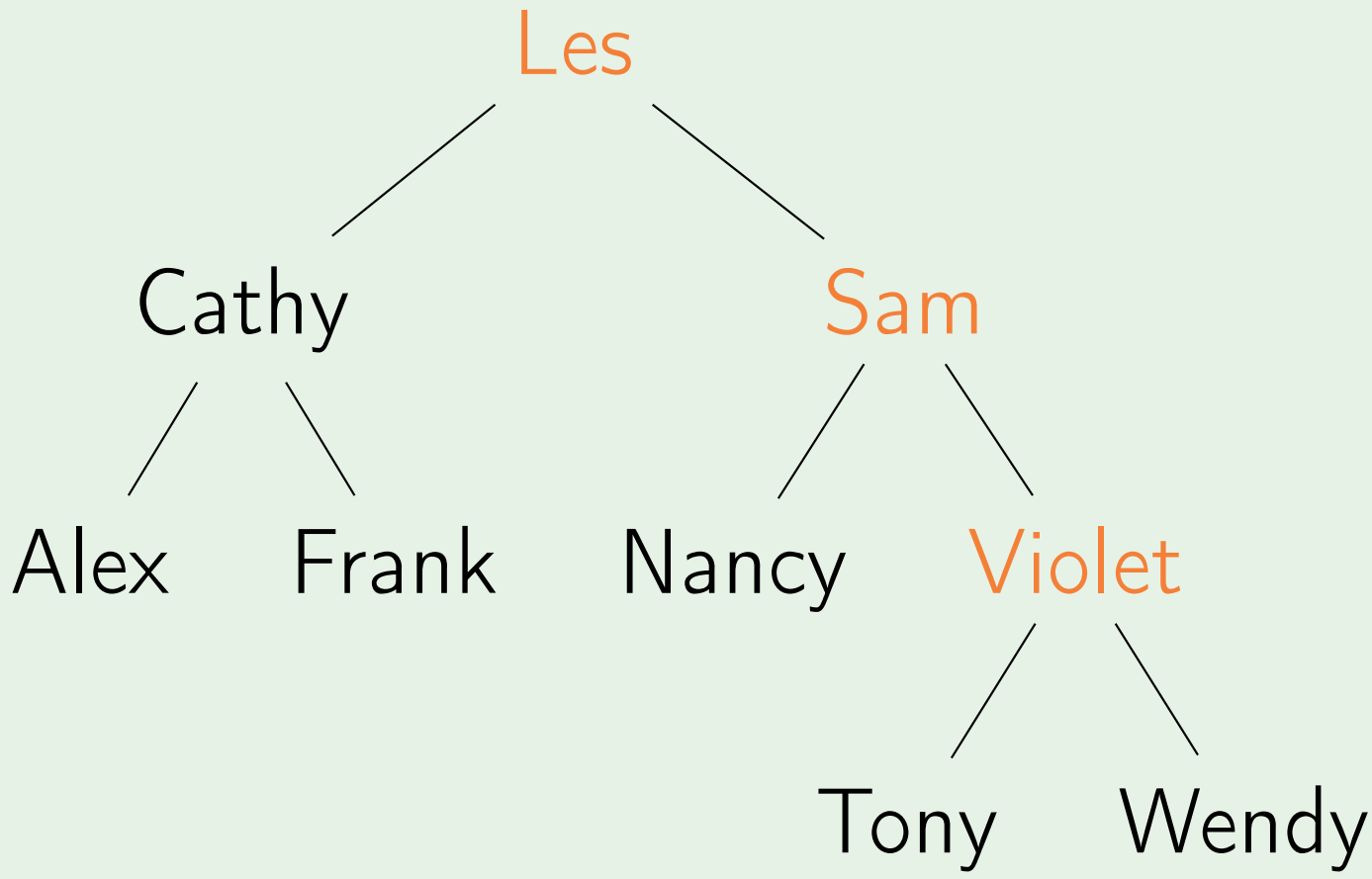
Output: Les Cathy Alex Frank Sam Nancy Violet Tony

PreOrderTraversal



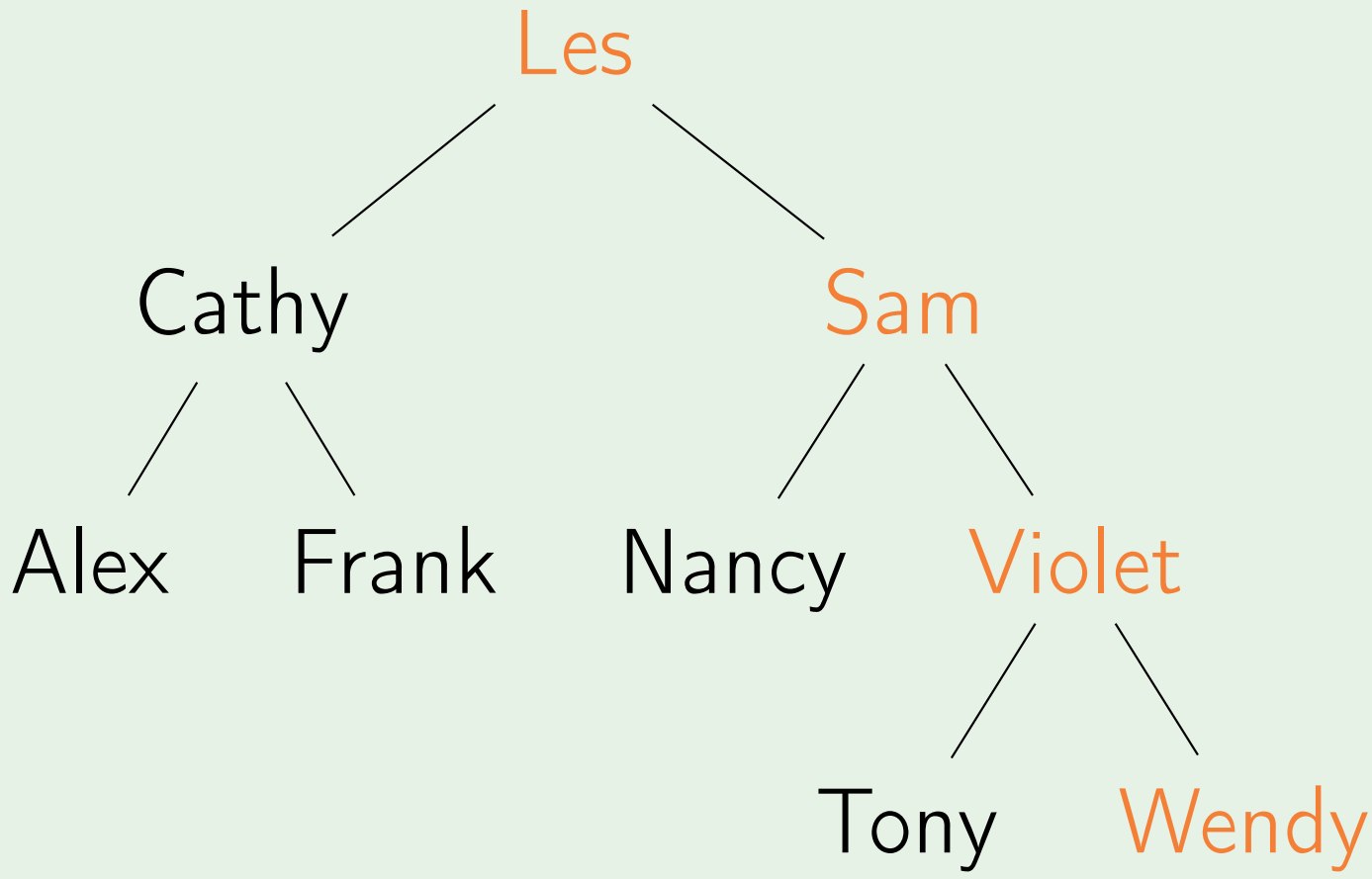
Output: Les Cathy Alex Frank Sam Nancy Violet Tony

PreOrderTraversal



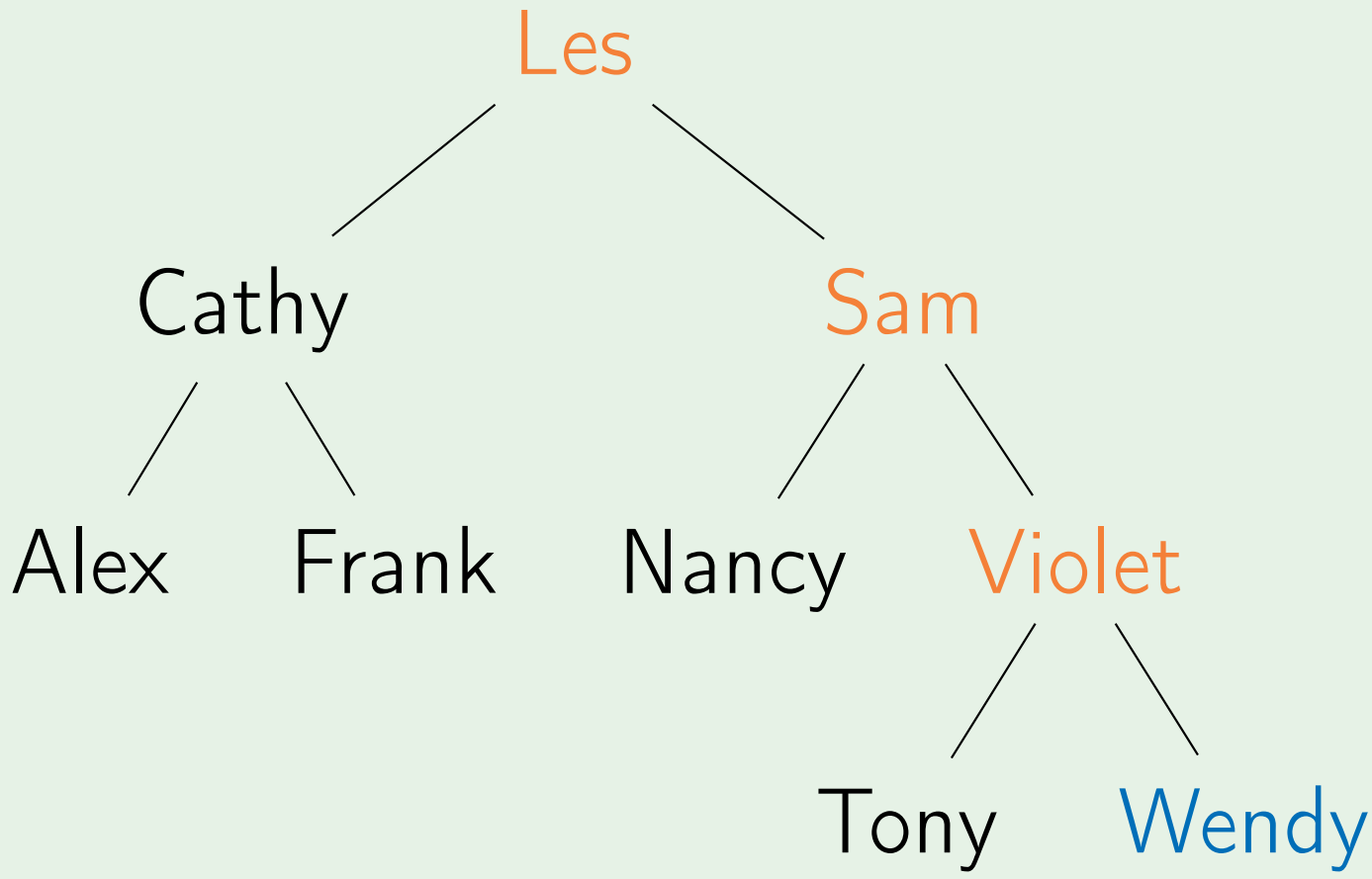
Output: Les Cathy Alex Frank Sam Nancy Violet Tony

PreOrderTraversal



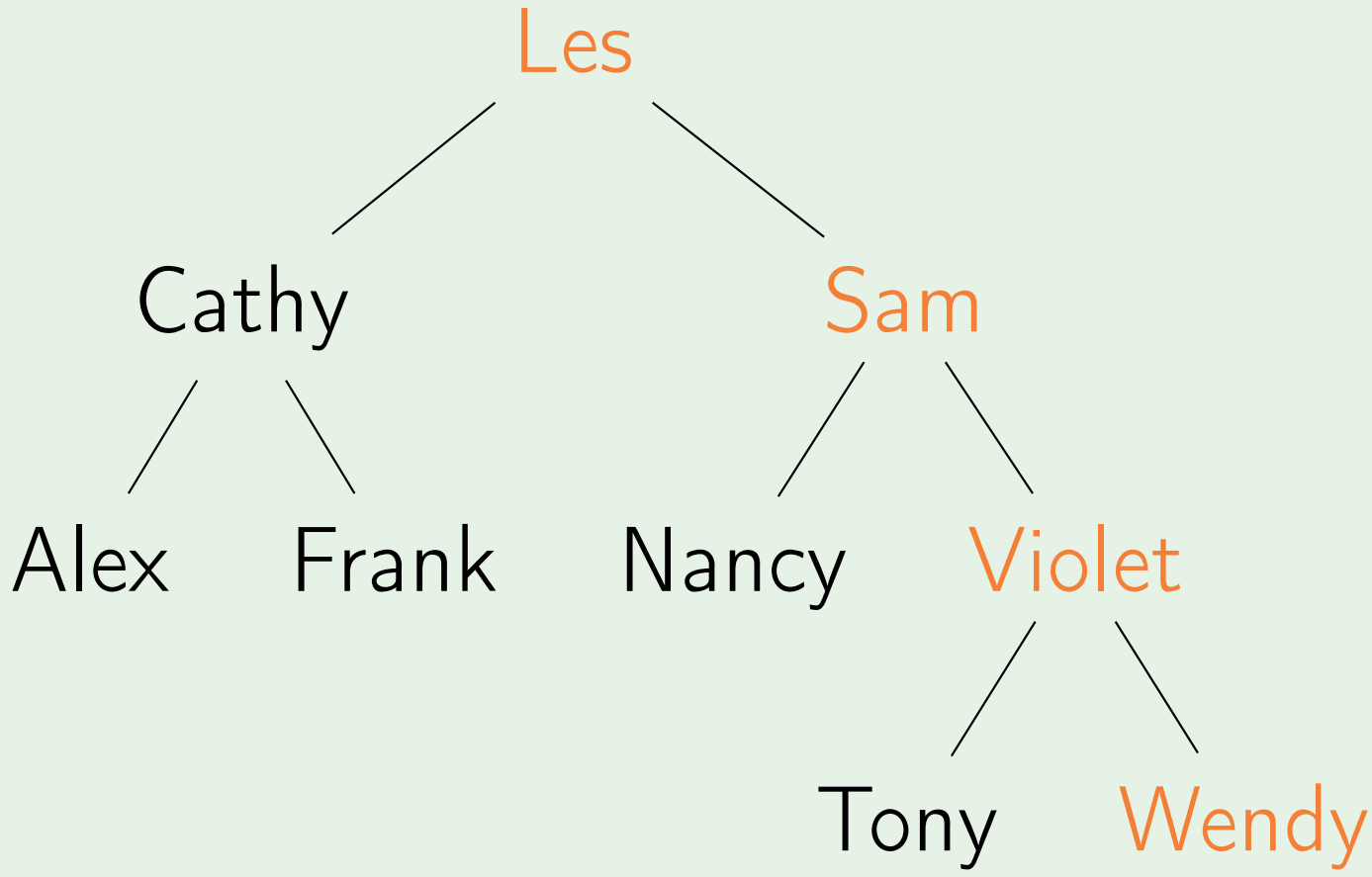
Output: Les Cathy Alex Frank Sam Nancy Violet Tony

PreOrderTraversal



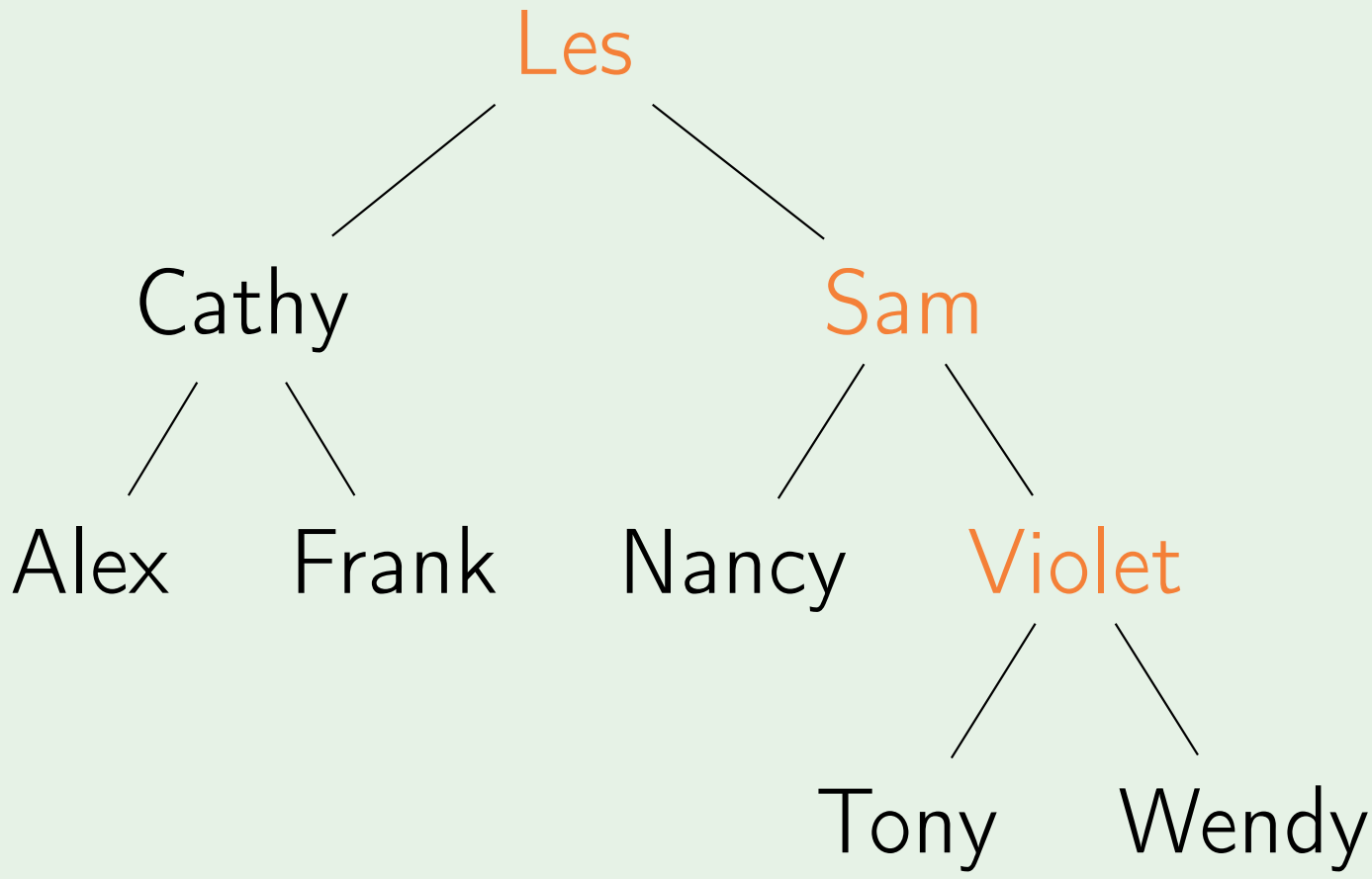
Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

PreOrderTraversal



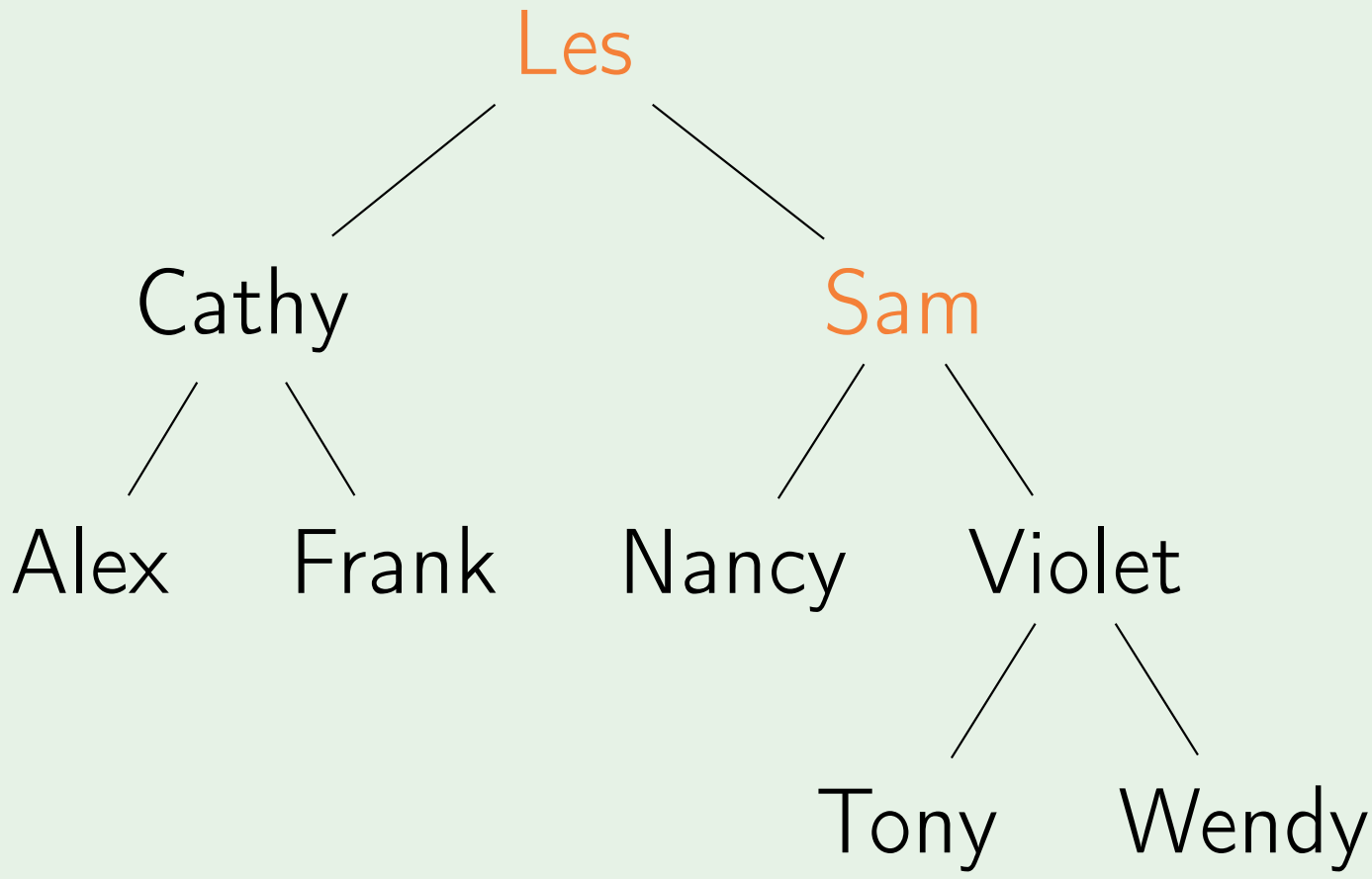
Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

PreOrderTraversal



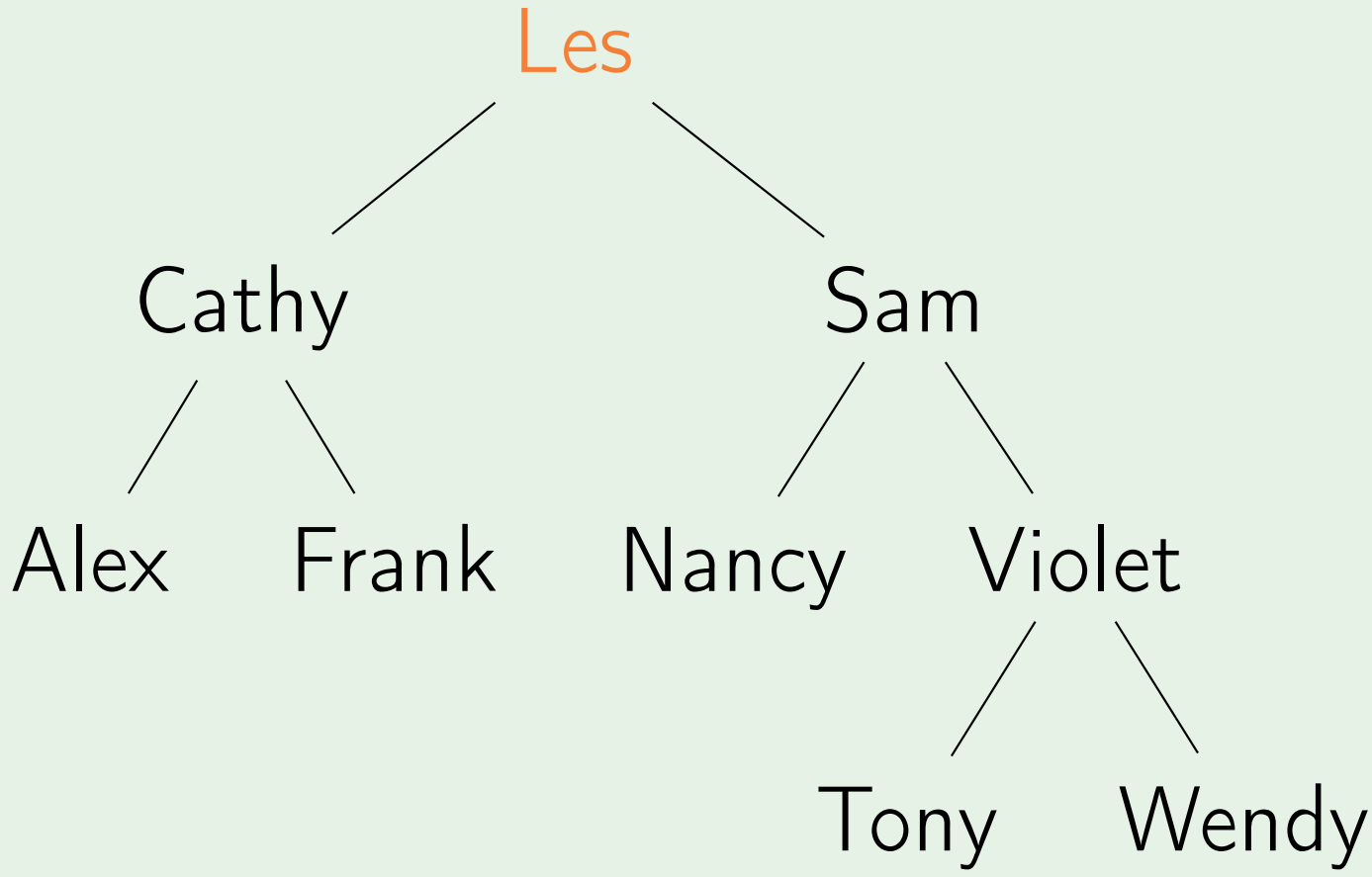
Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

PreOrderTraversal



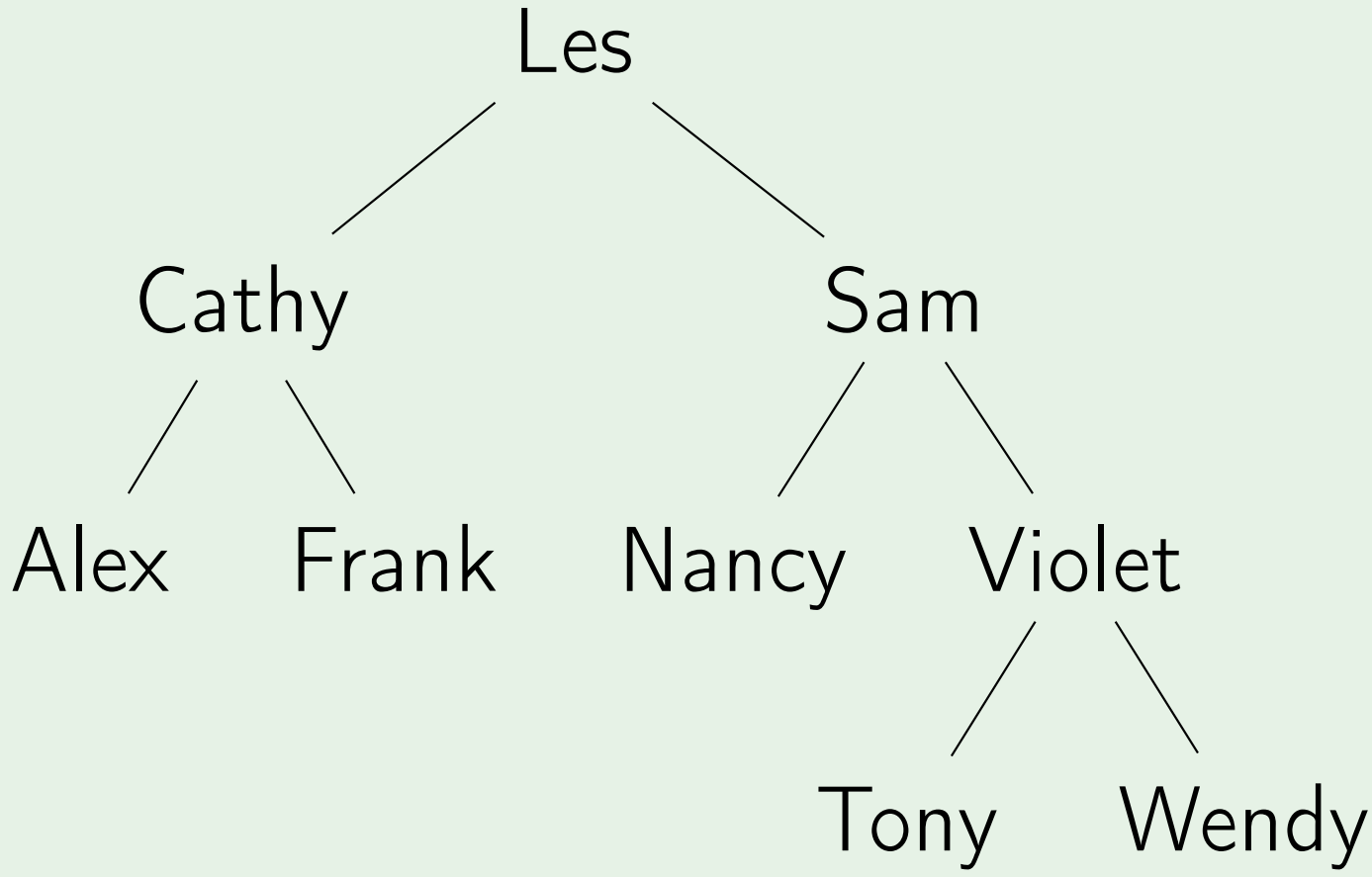
Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

PreOrderTraversal



Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

PreOrderTraversal

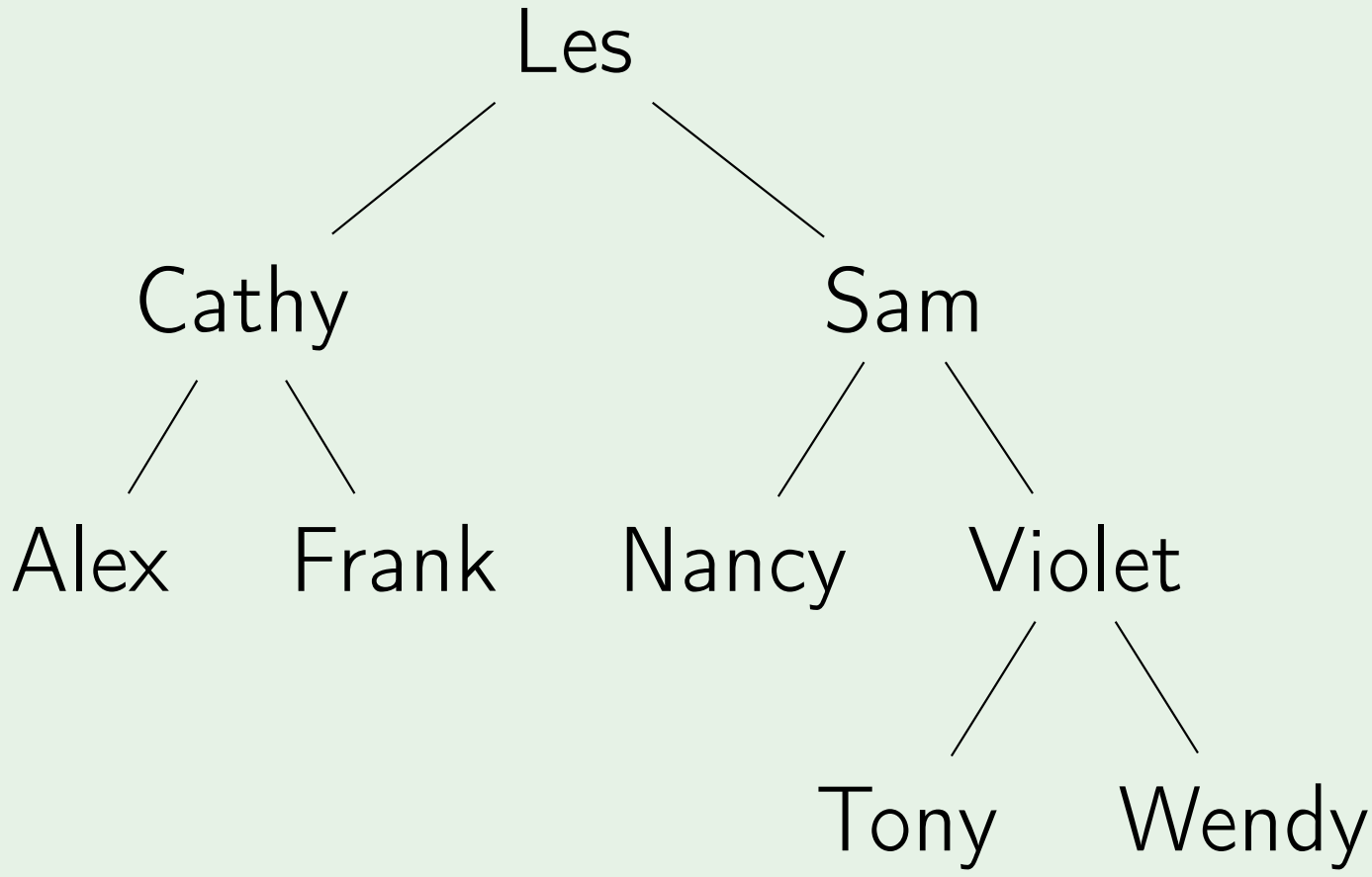


Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

Depth-first

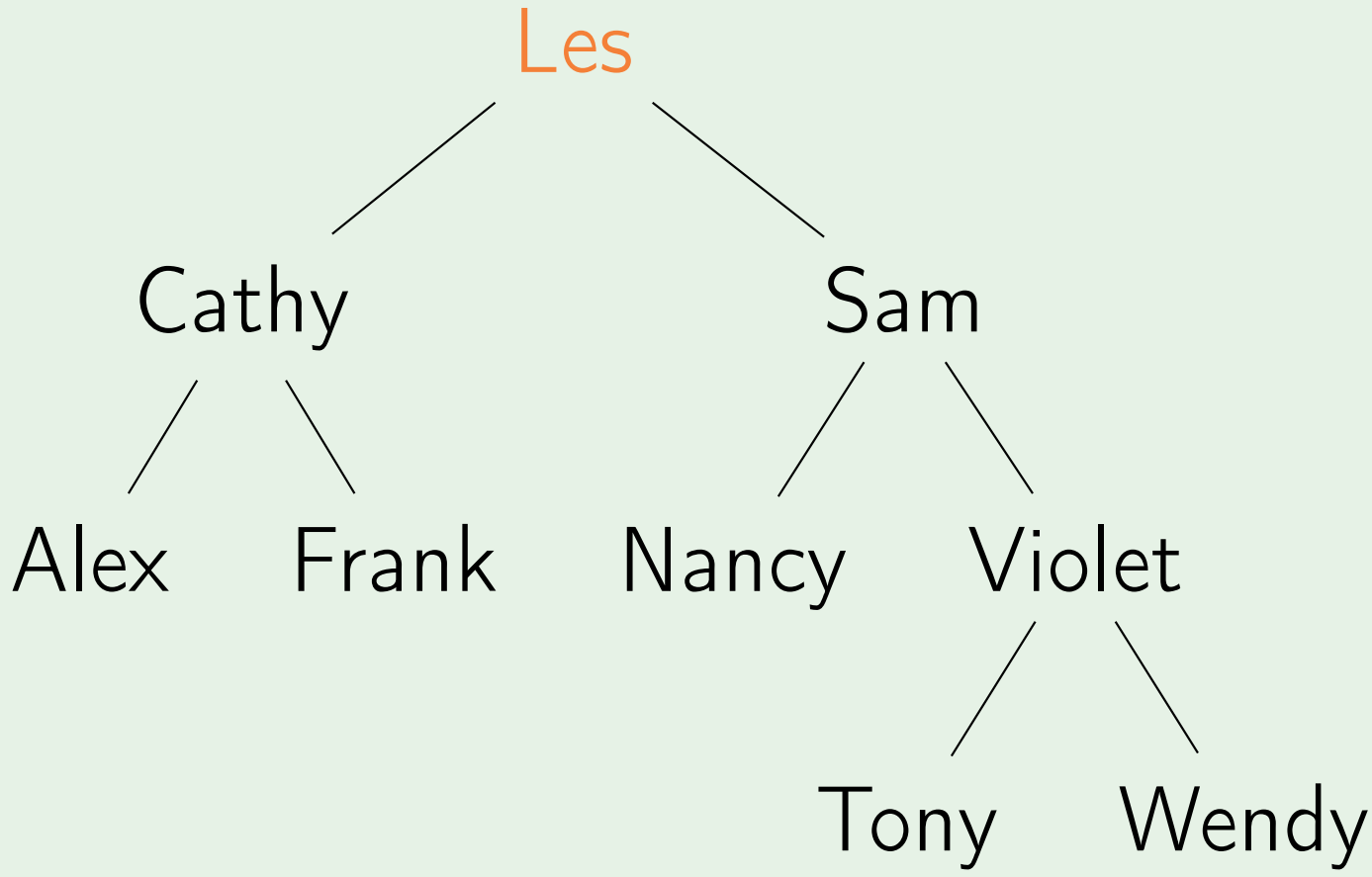
```
PostOrderTraversal(tree)
if tree = nil:
    return
PostOrderTraversal(tree.left)
PostOrderTraversal(tree.right)
Print(tree.key)
```

PostOrderTraversal



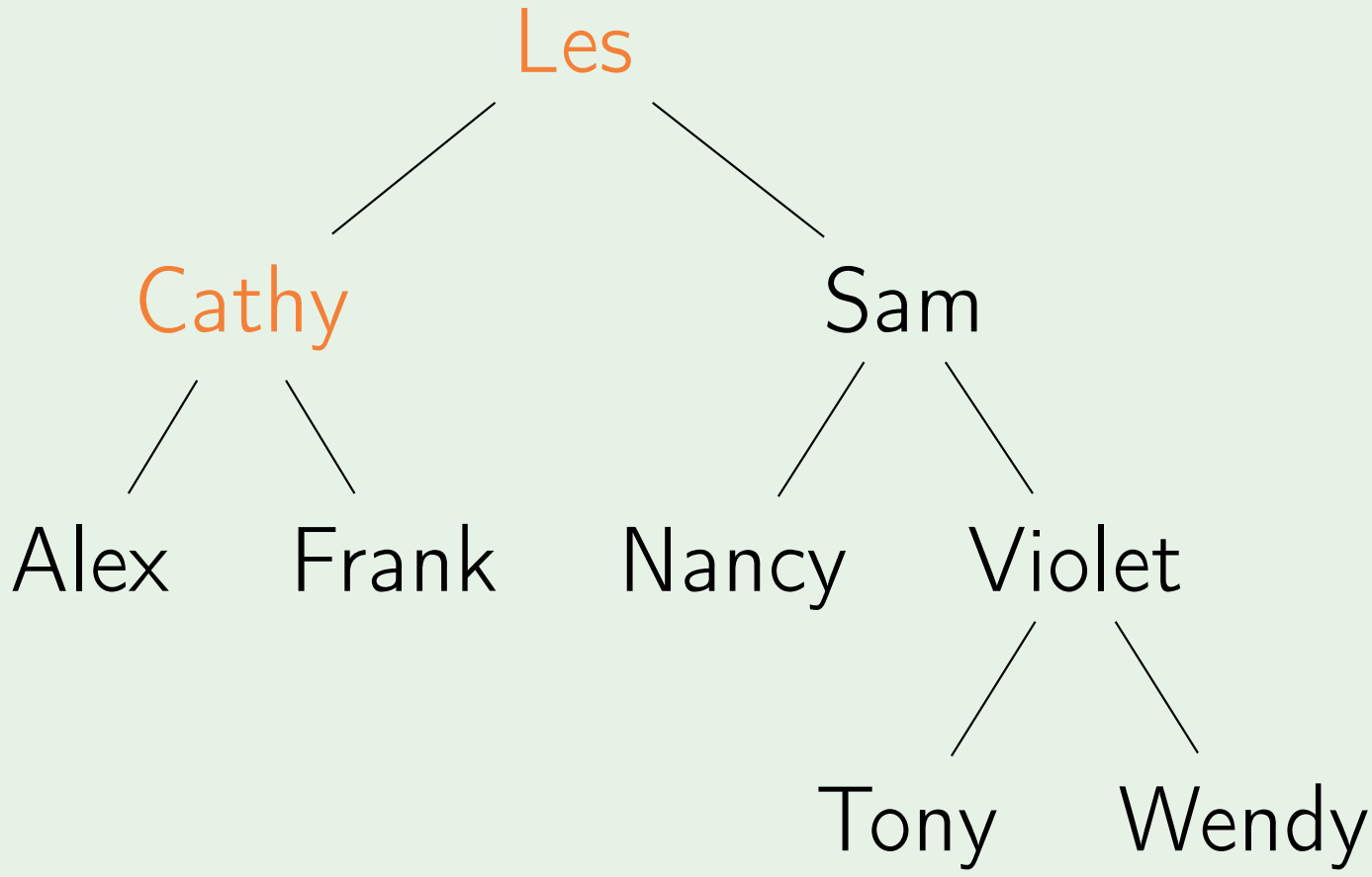
Output:

PostOrderTraversal



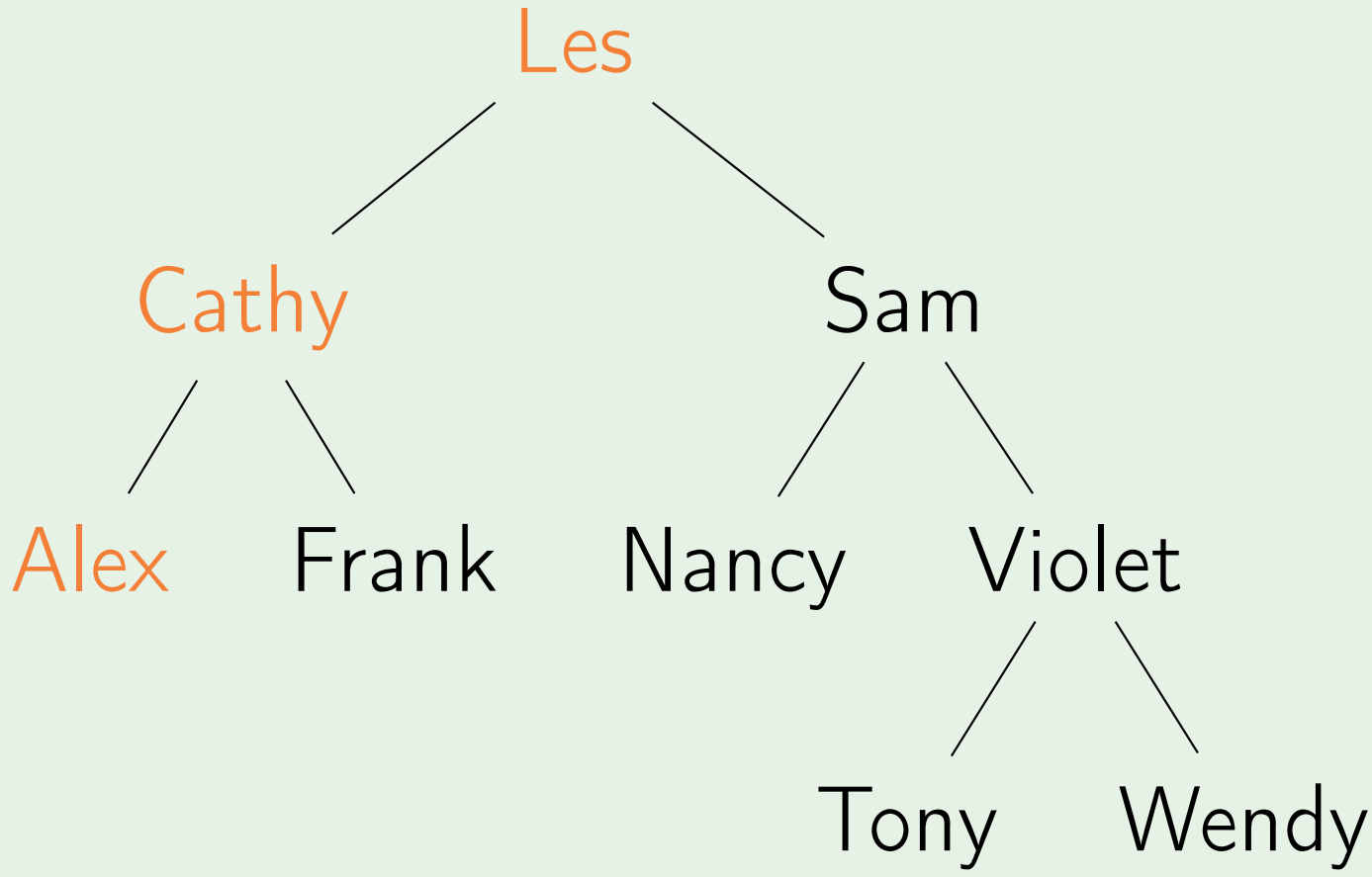
Output:

PostOrderTraversal



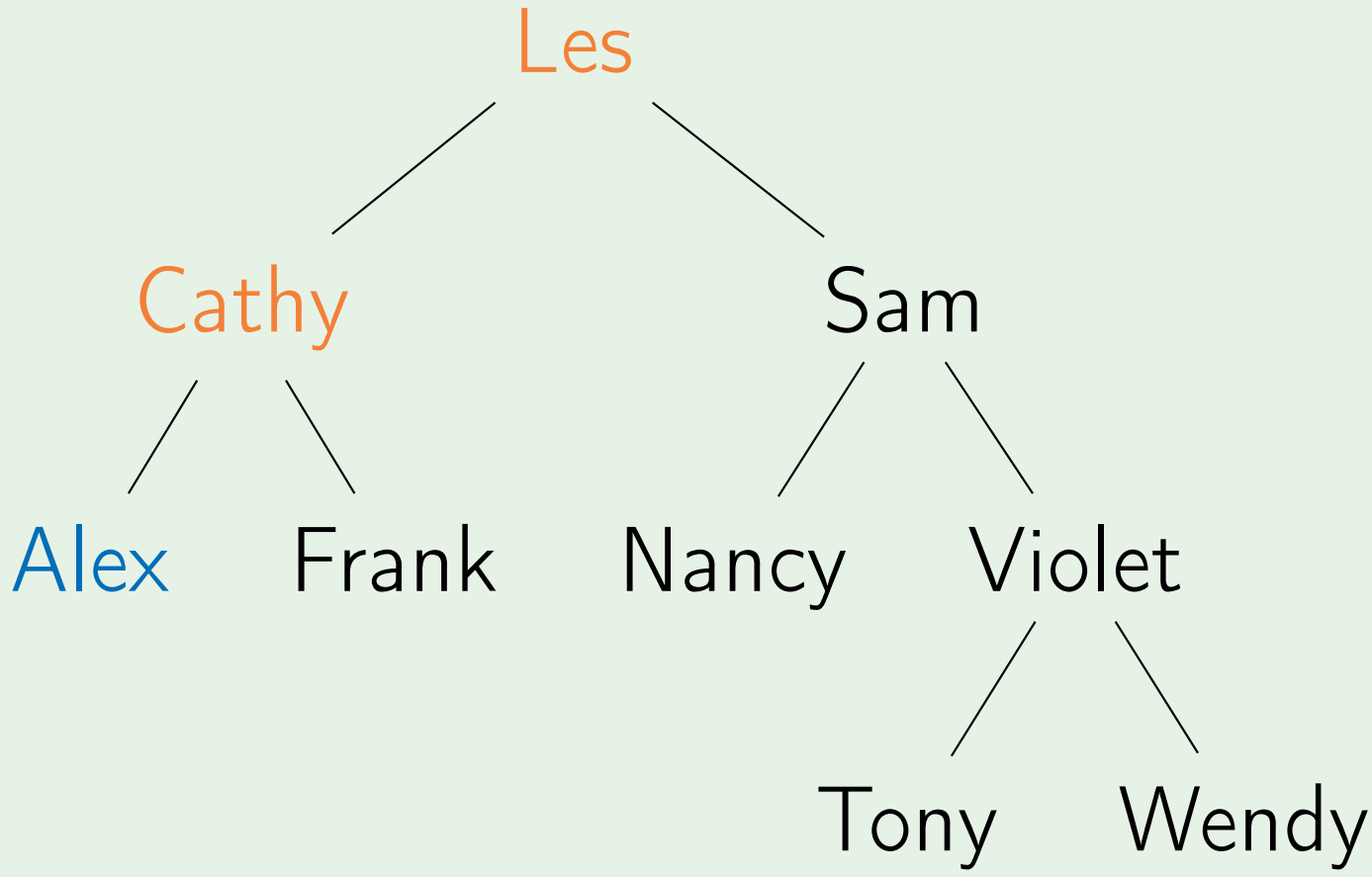
Output:

PostOrderTraversal



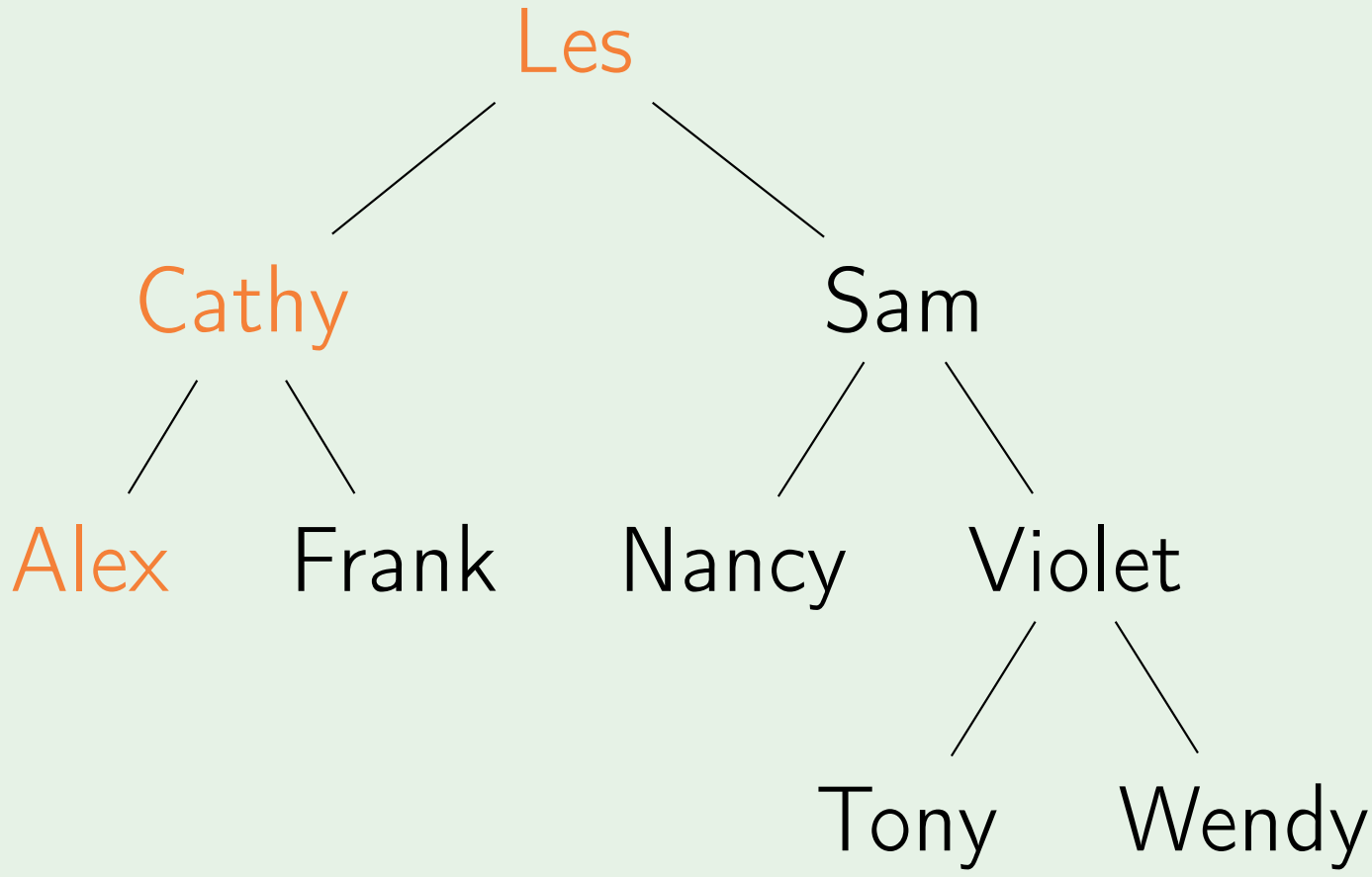
Output:

PostOrderTraversal



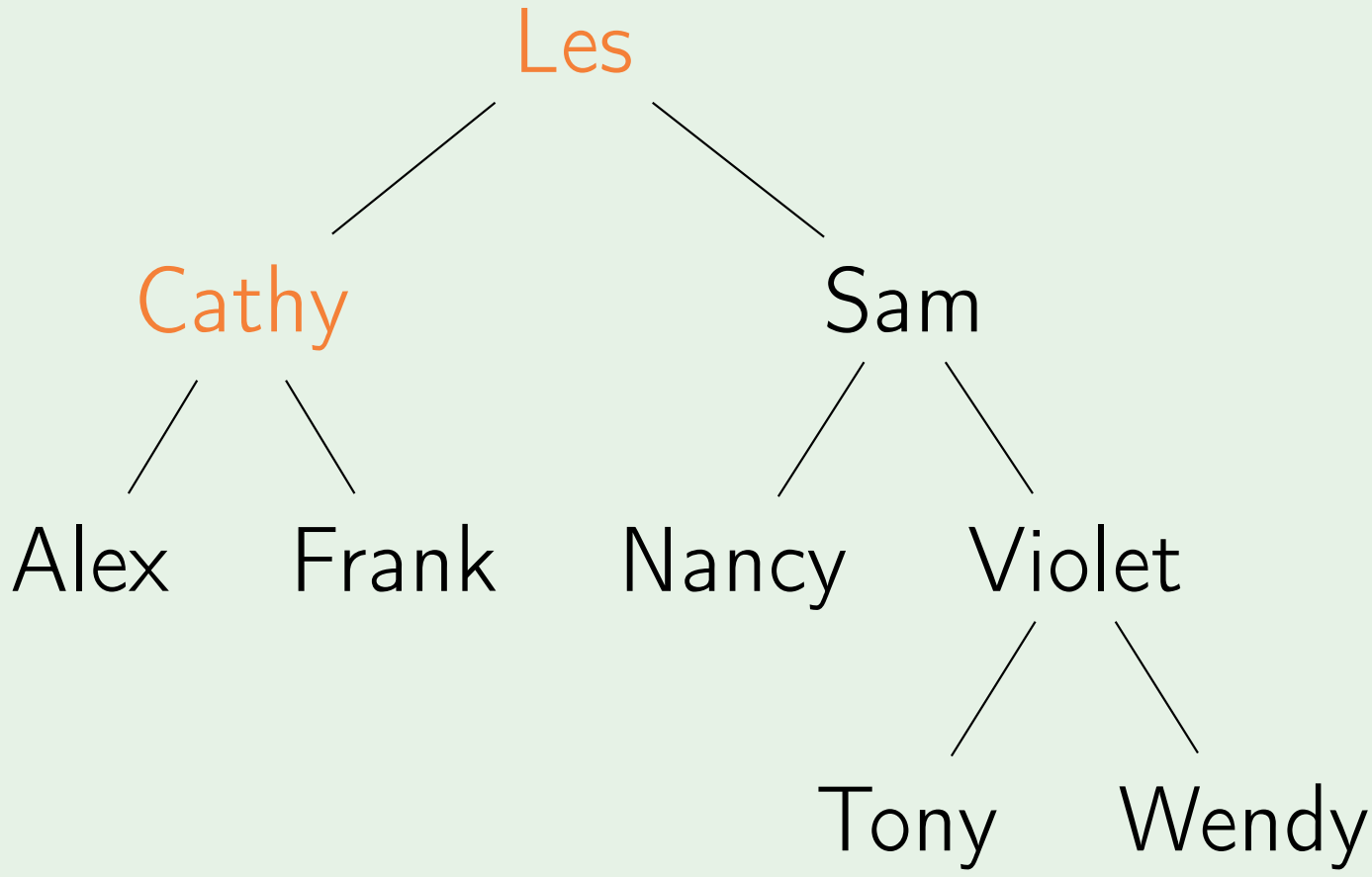
Output: Alex

PostOrderTraversal



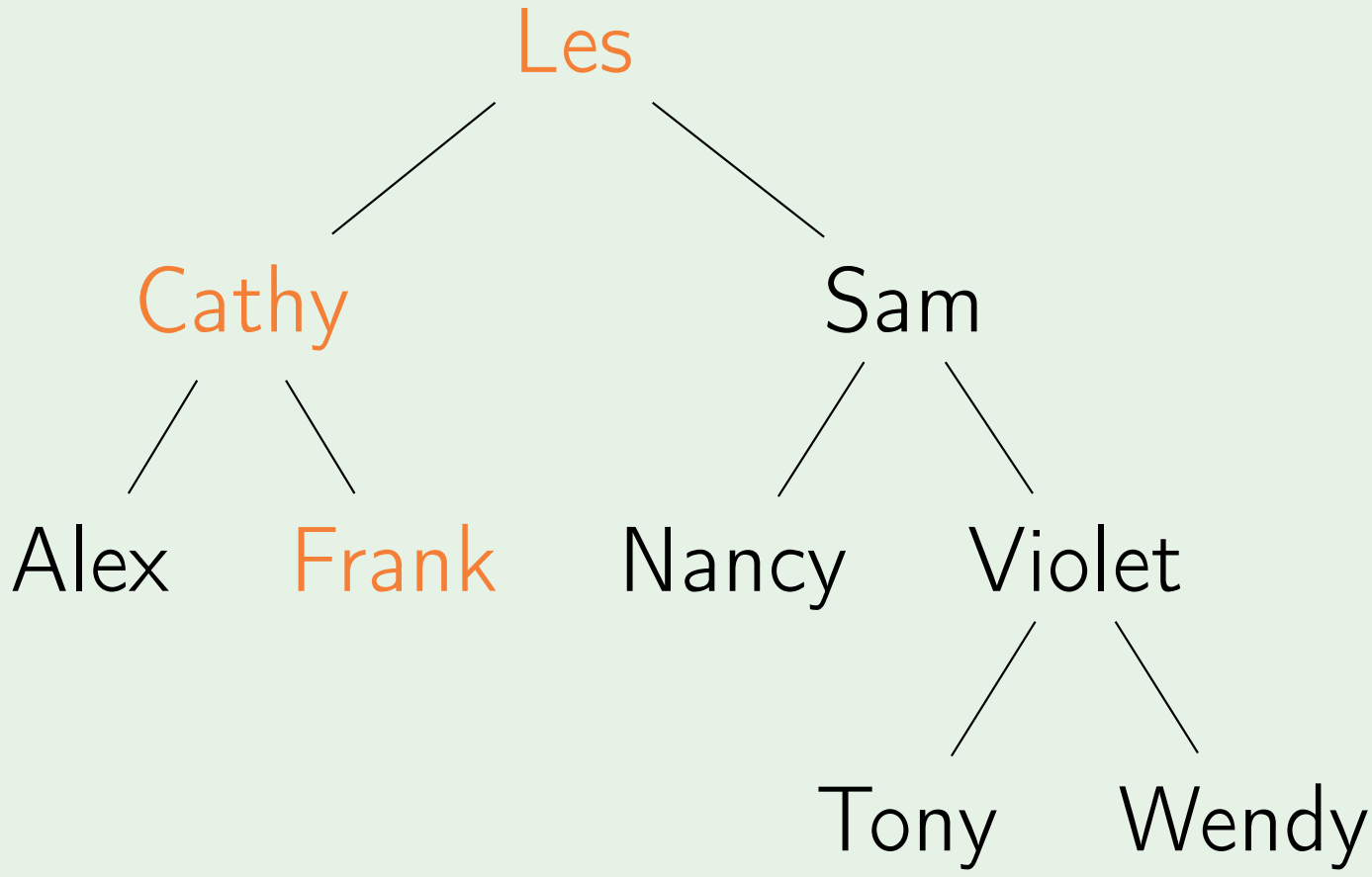
Output: Alex

PostOrderTraversal



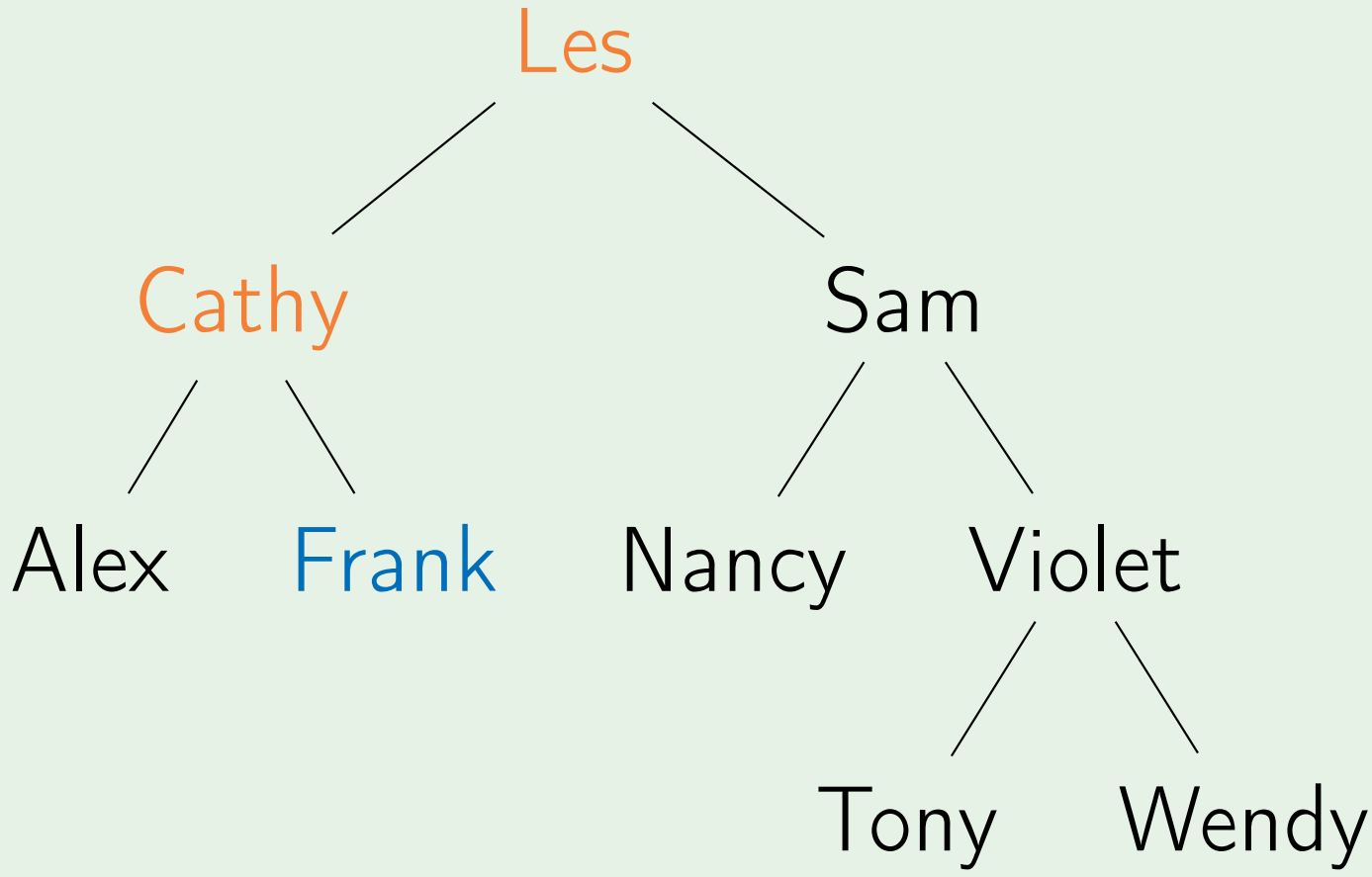
Output: Alex

PostOrderTraversal



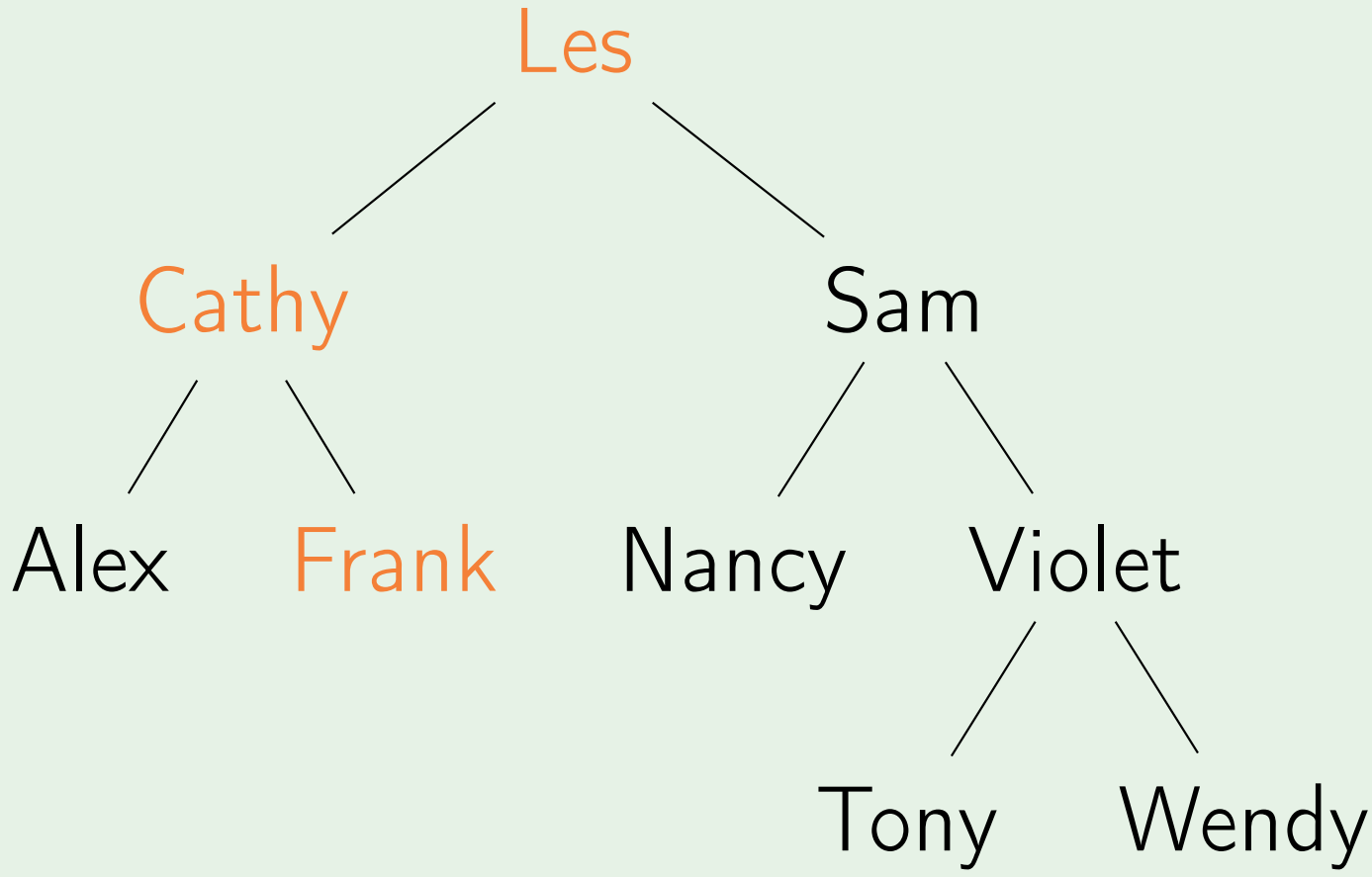
Output: Alex

PostOrderTraversal



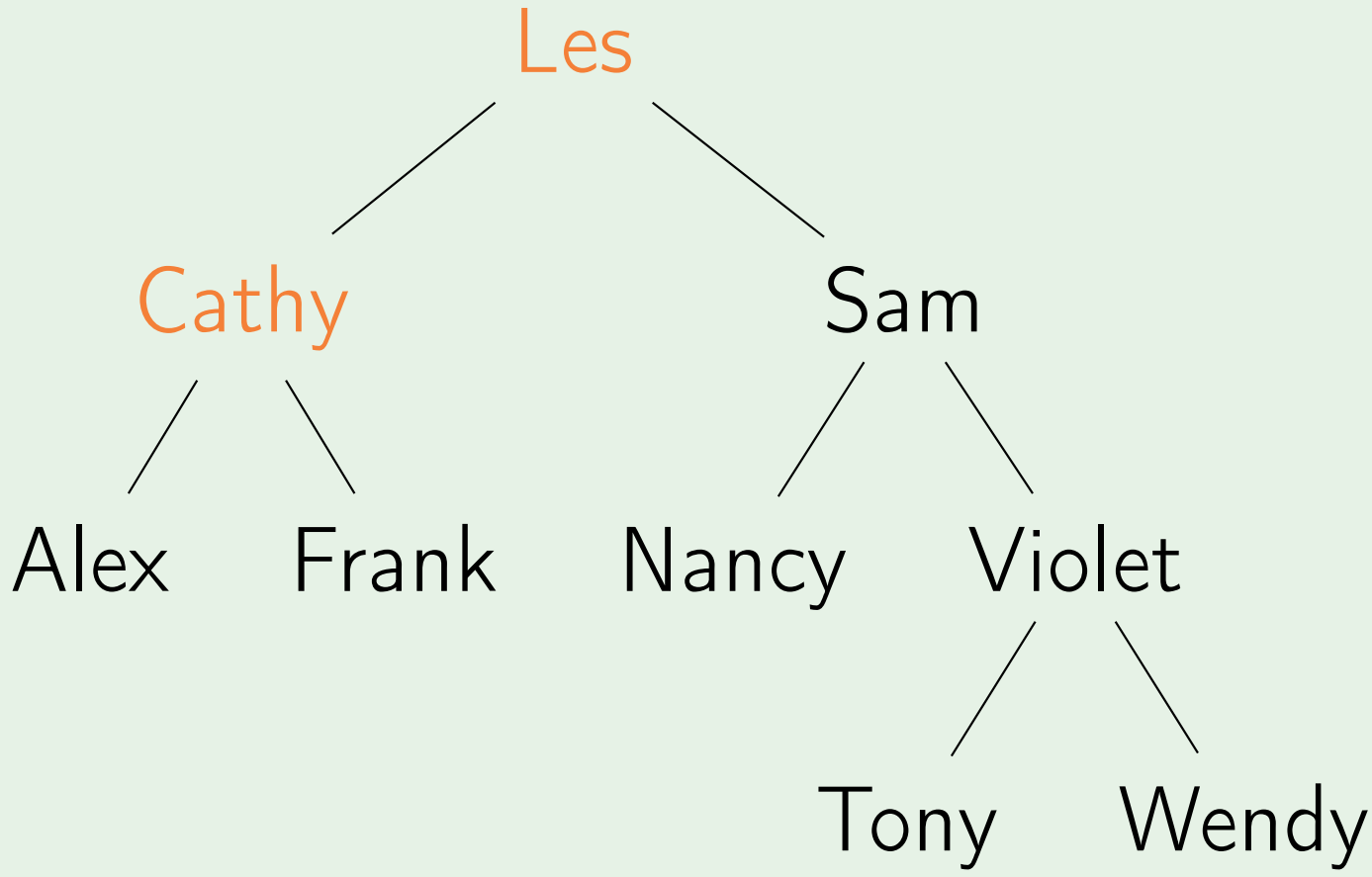
Output: Alex Frank

PostOrderTraversal



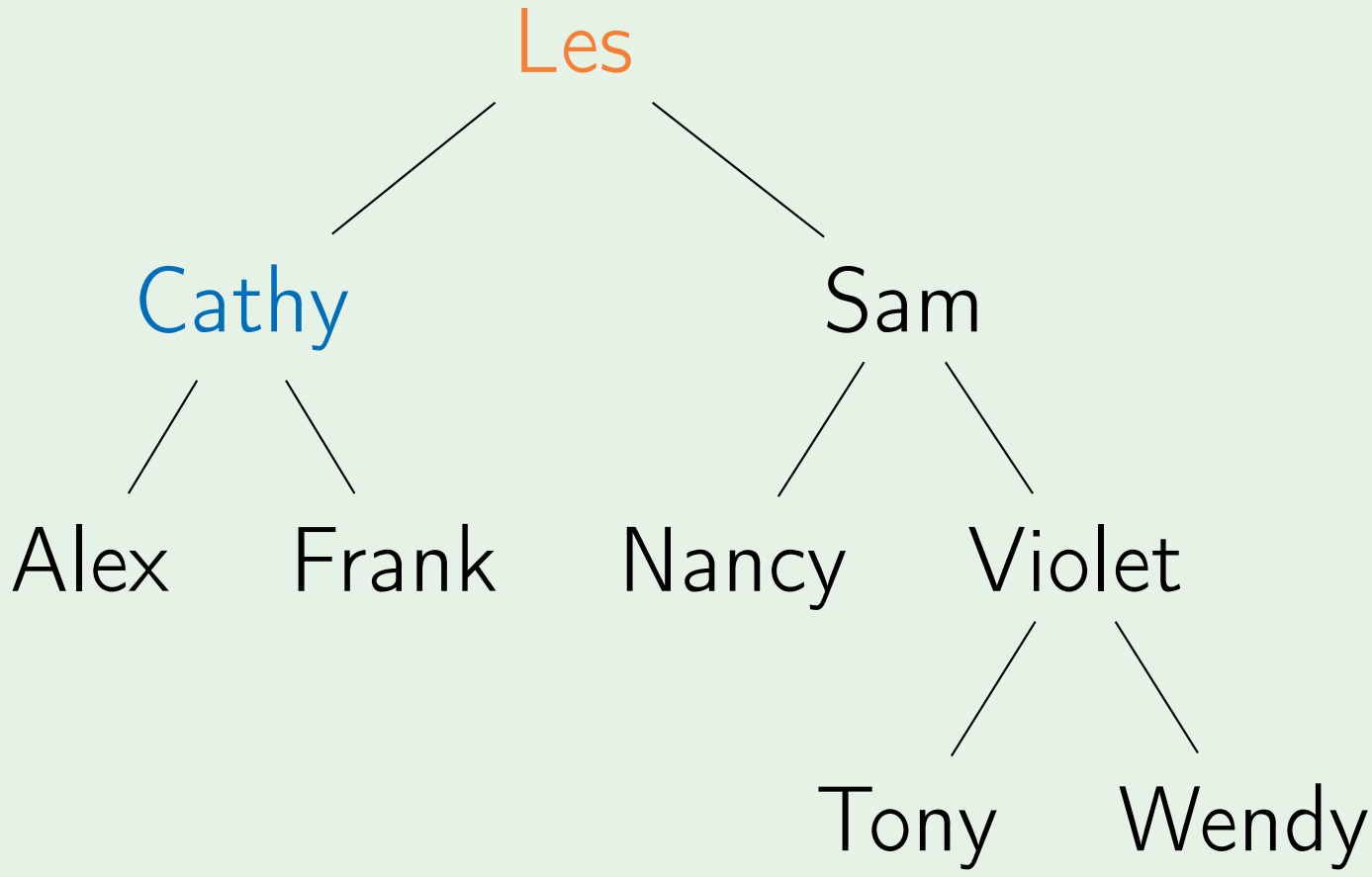
Output: Alex Frank

PostOrderTraversal



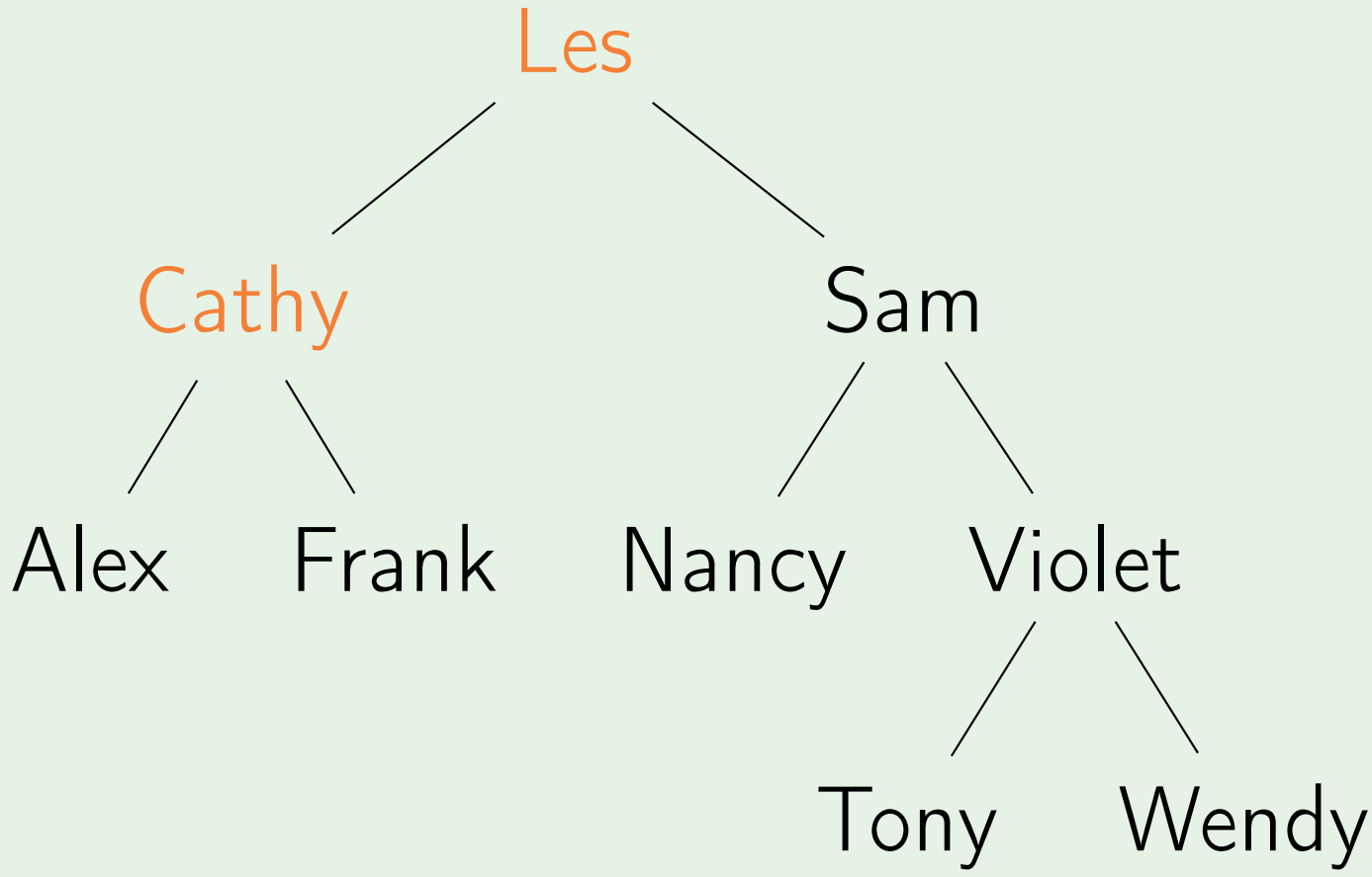
Output: Alex Frank

PostOrderTraversal



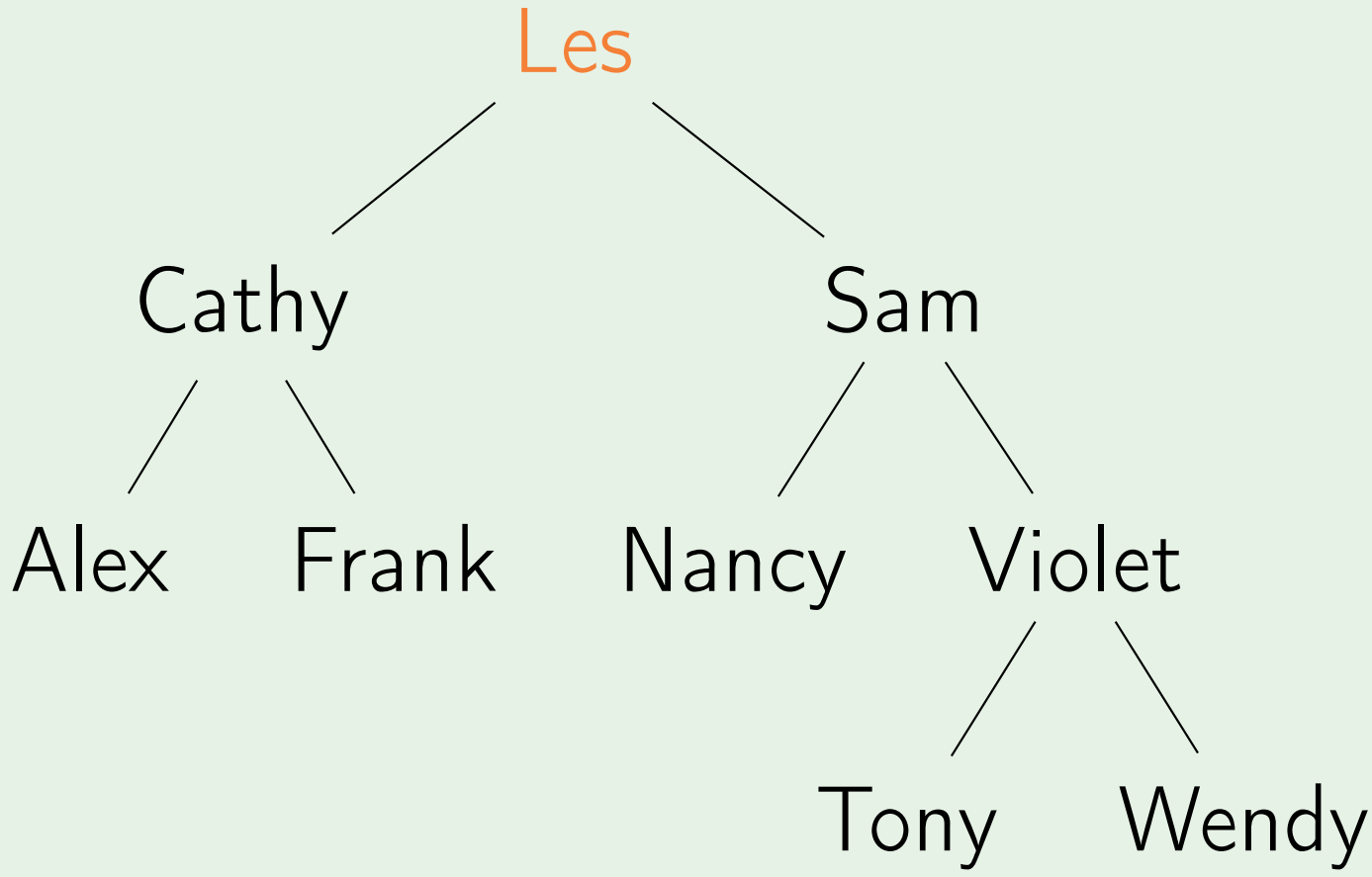
Output: Alex Frank Cathy

PostOrderTraversal



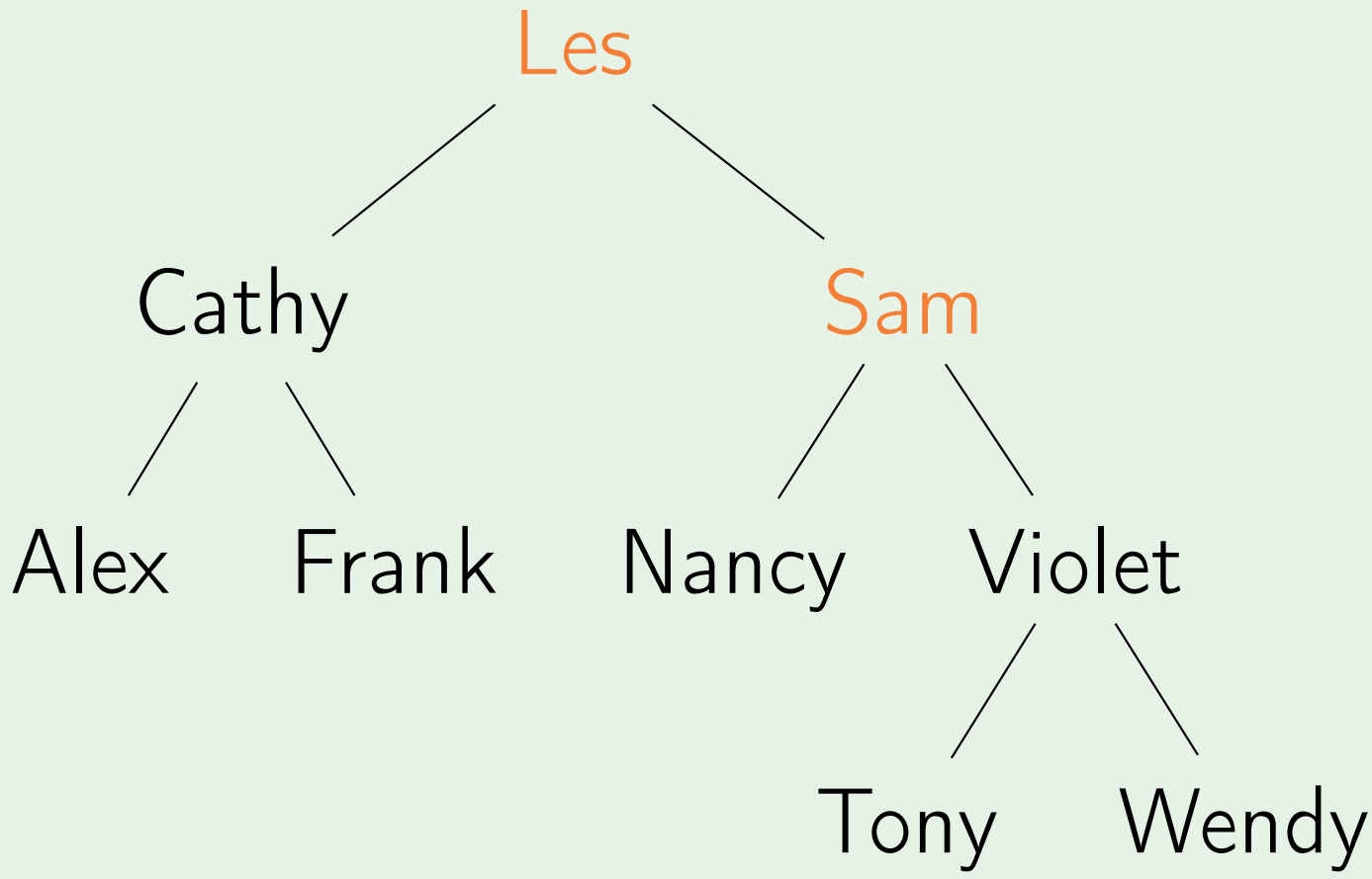
Output: Alex Frank Cathy

PostOrderTraversal



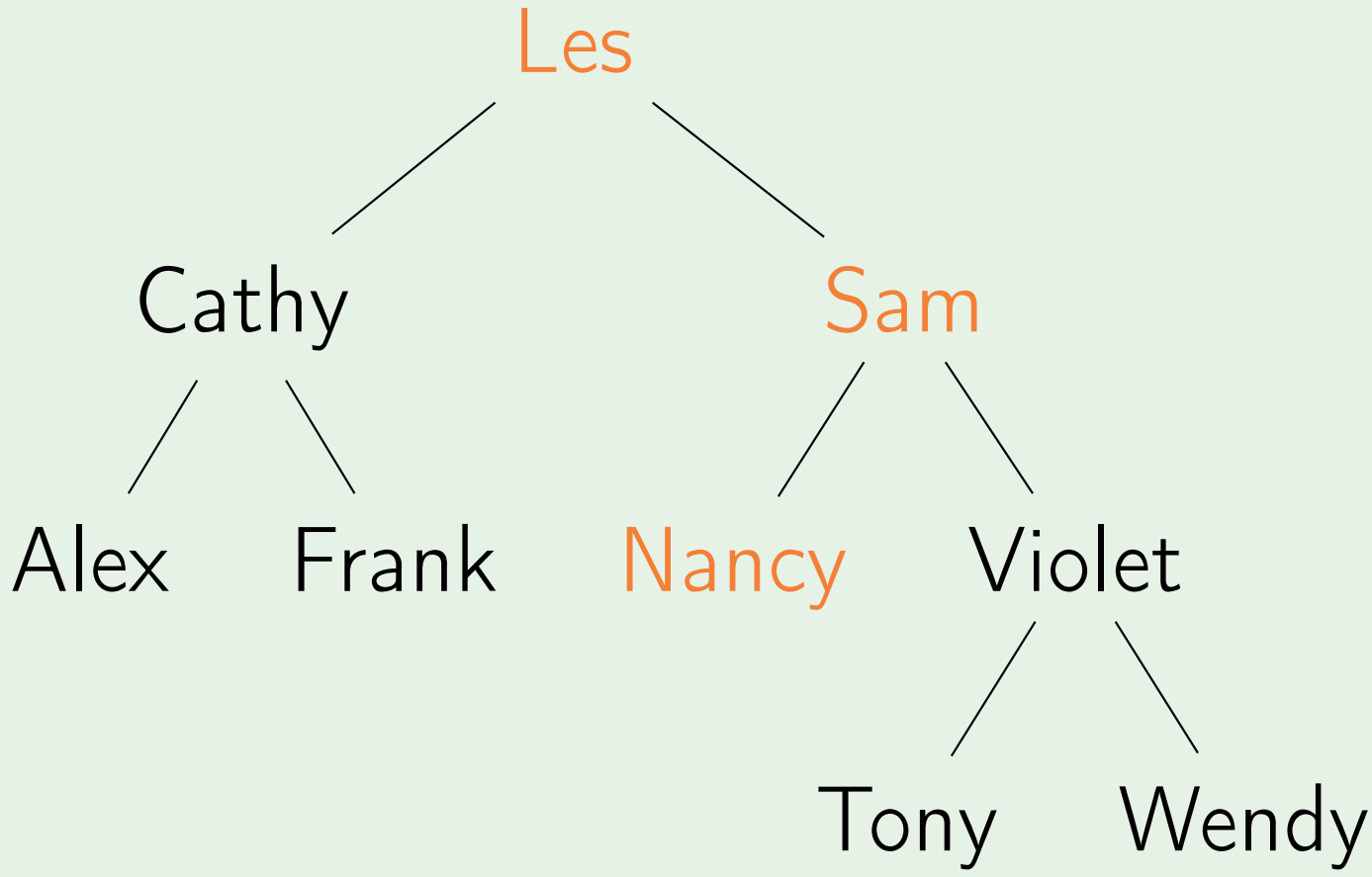
Output: Alex Frank Cathy

PostOrderTraversal



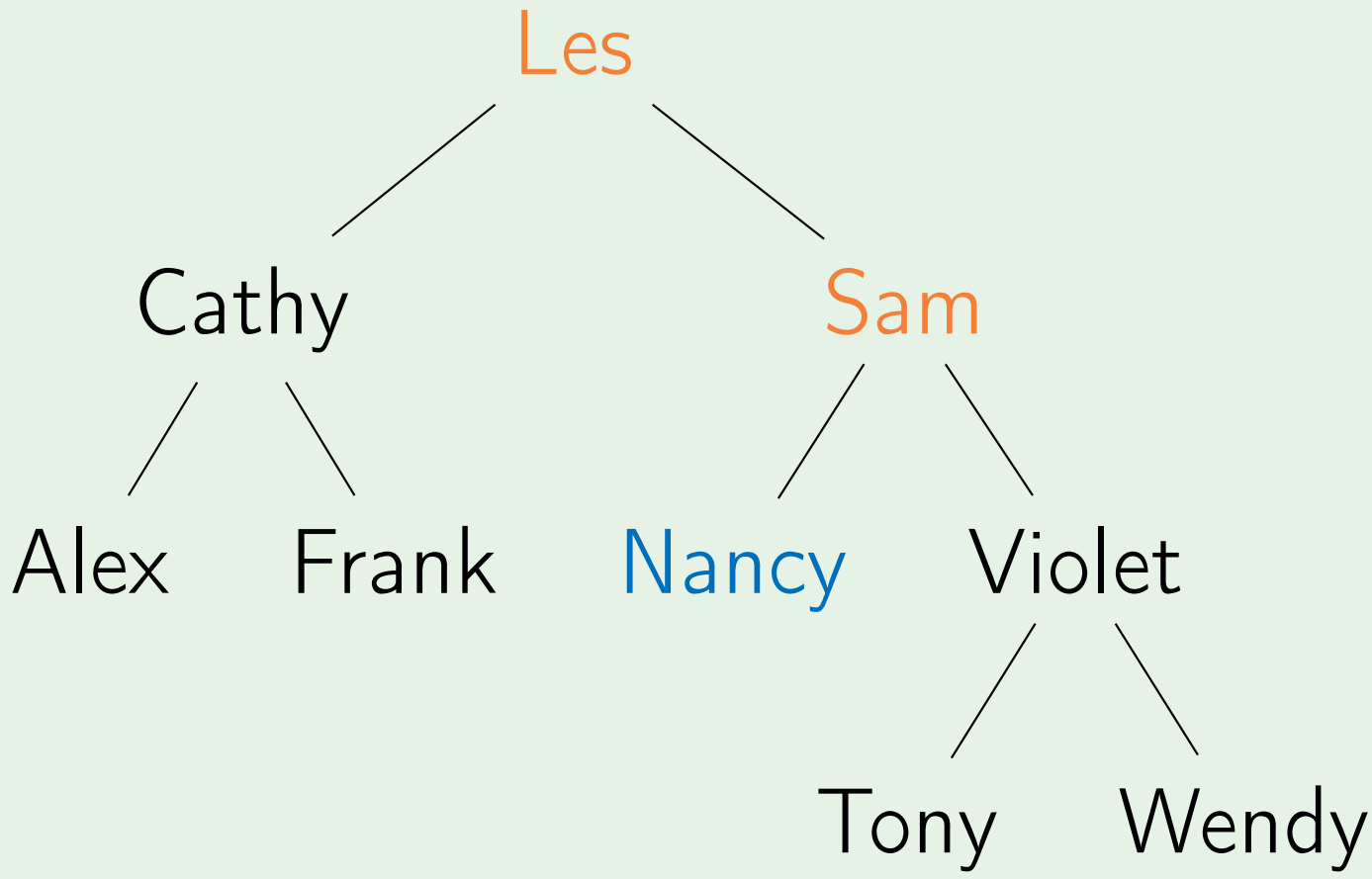
Output: Alex Frank Cathy

PostOrderTraversal



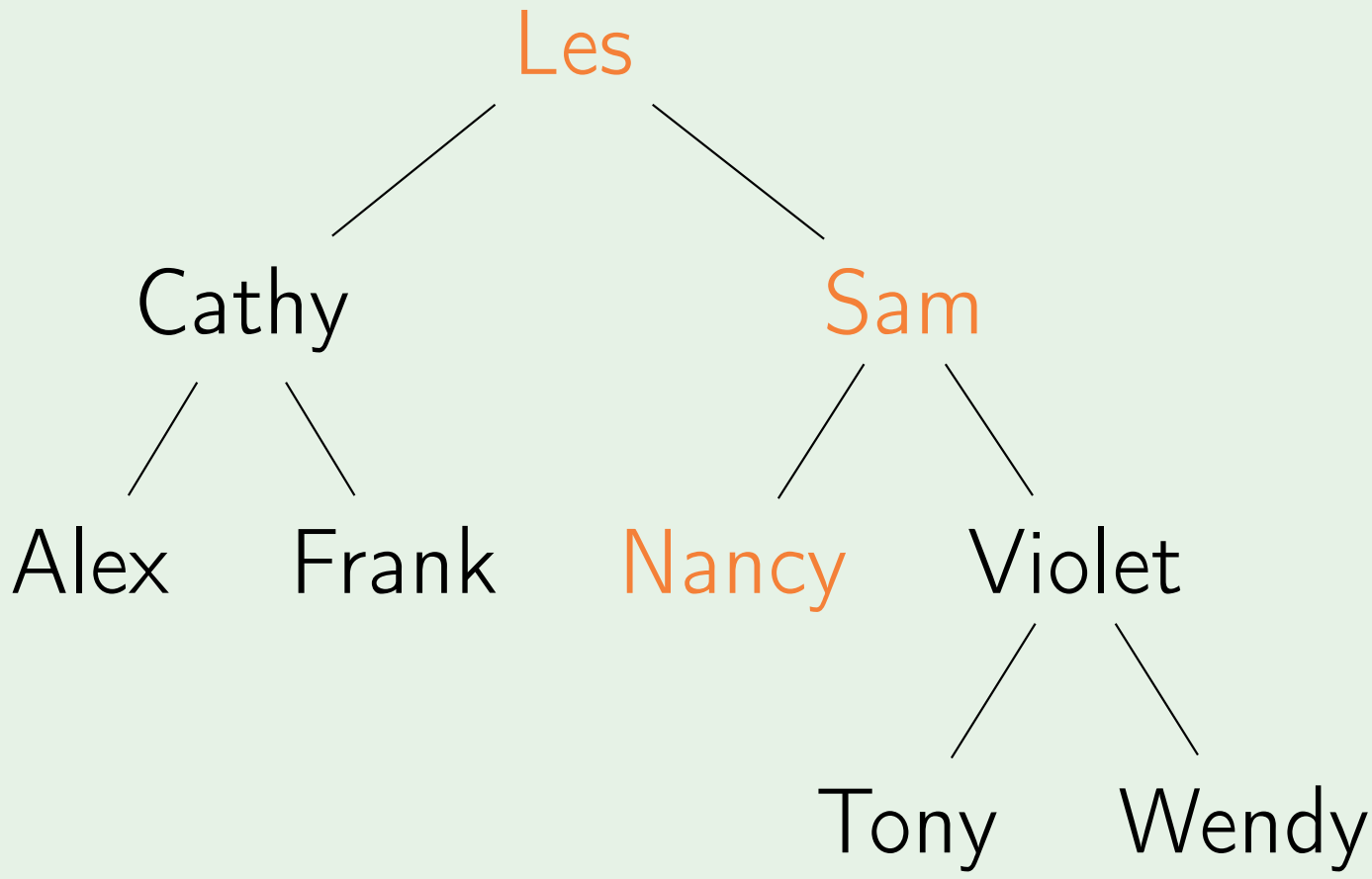
Output: Alex Frank Cathy

PostOrderTraversal



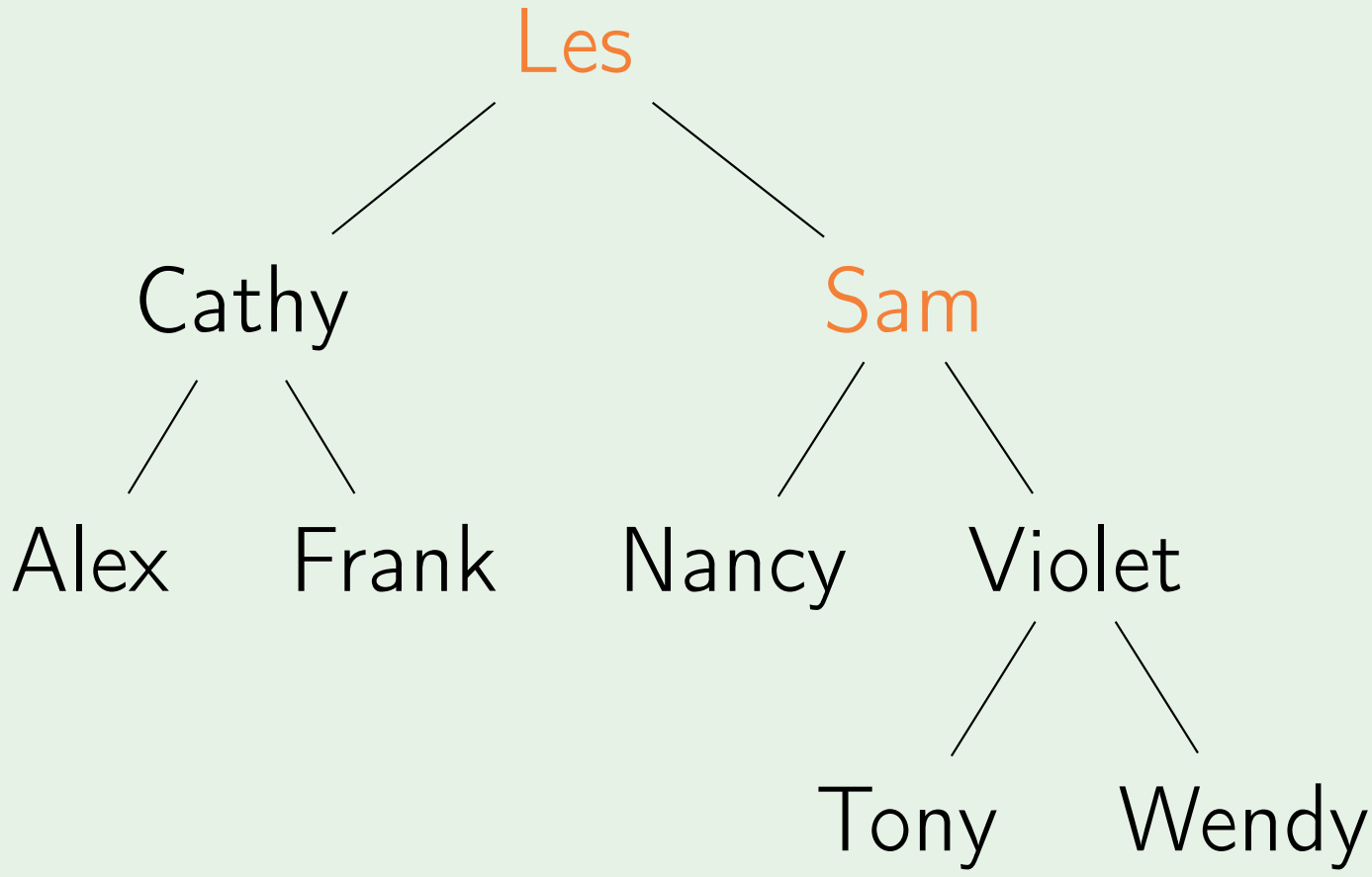
Output: Alex Frank Cathy Nancy

PostOrderTraversal



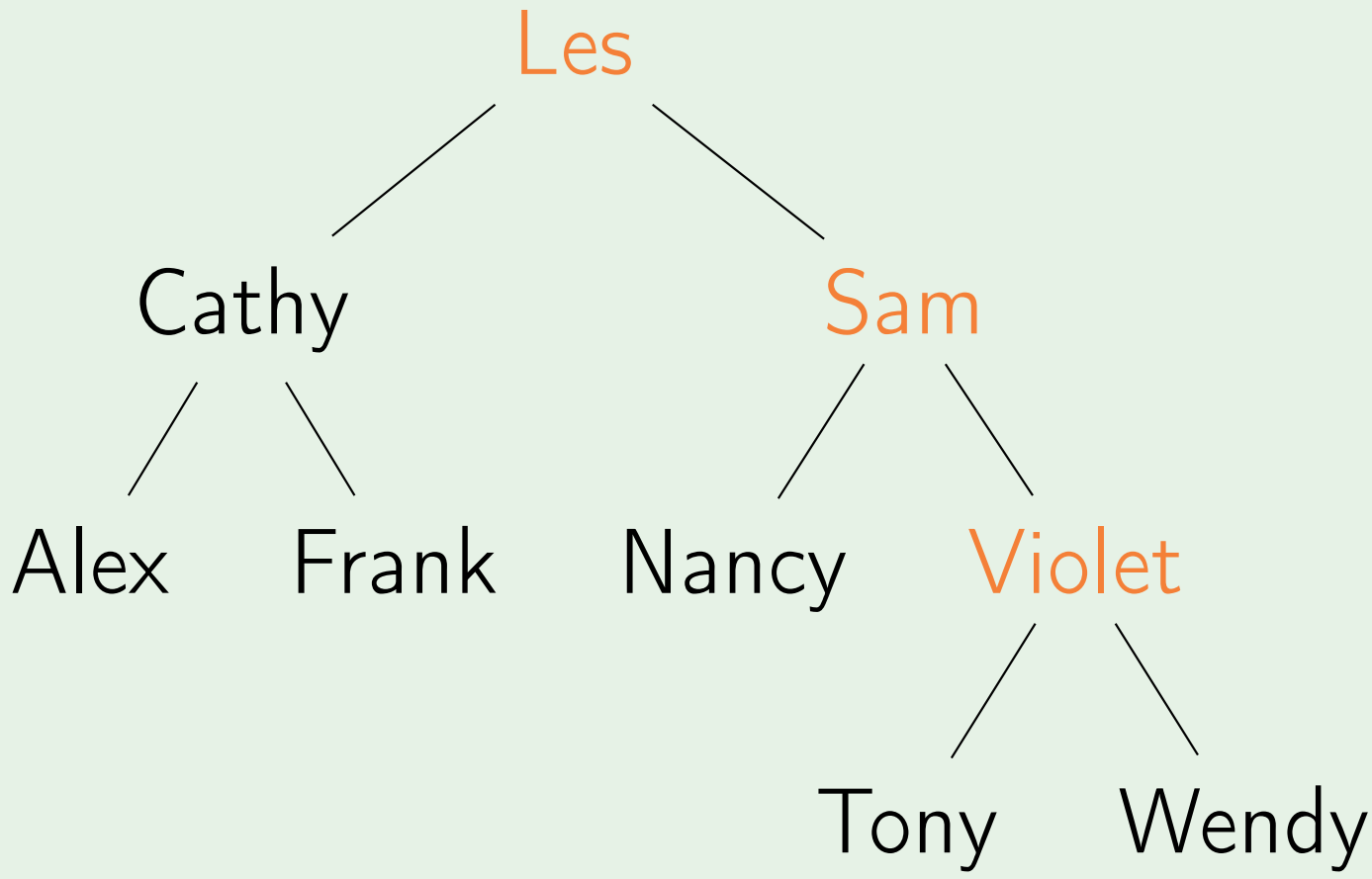
Output: Alex Frank Cathy Nancy

PostOrderTraversal



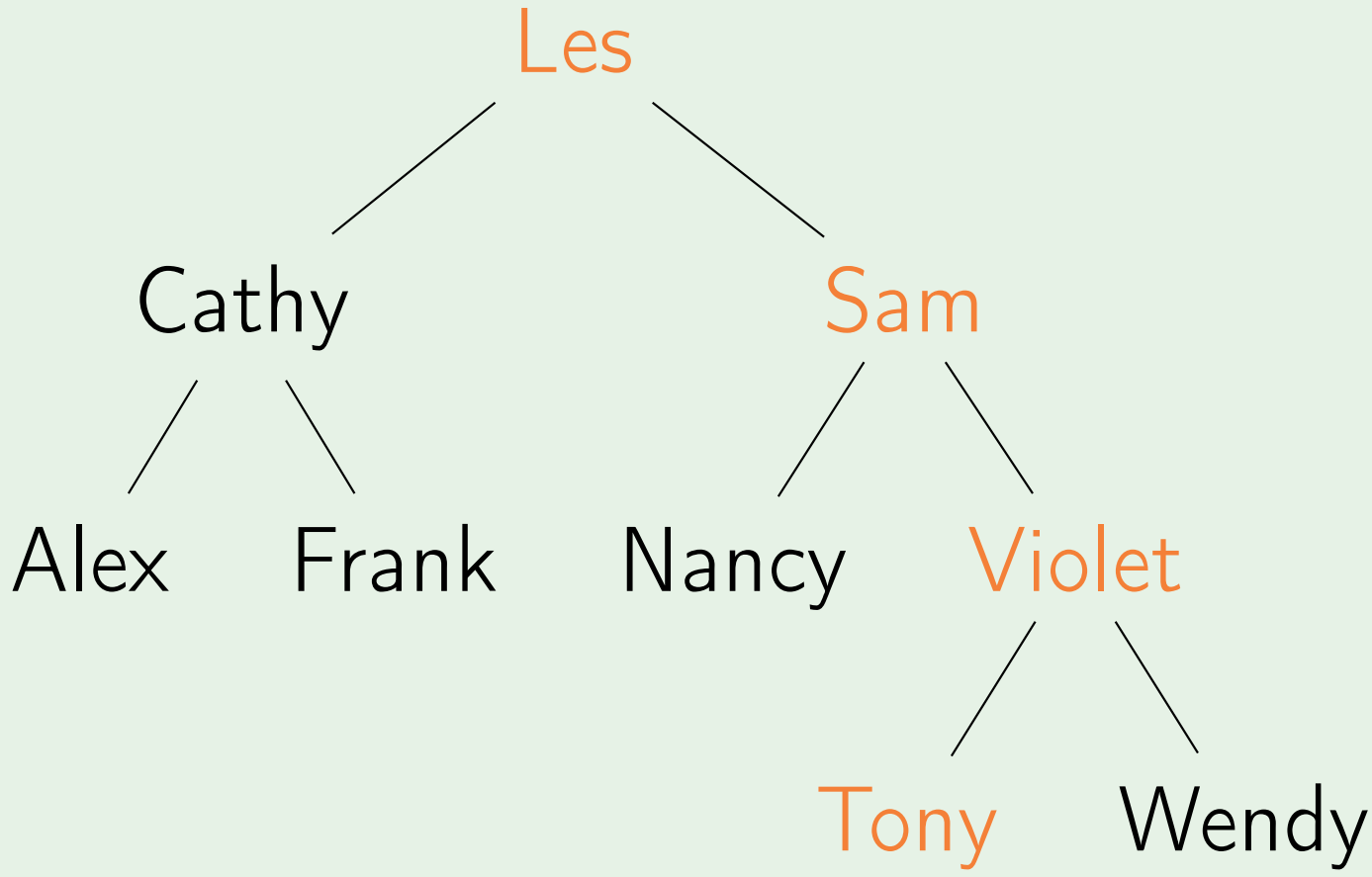
Output: Alex Frank Cathy Nancy

PostOrderTraversal



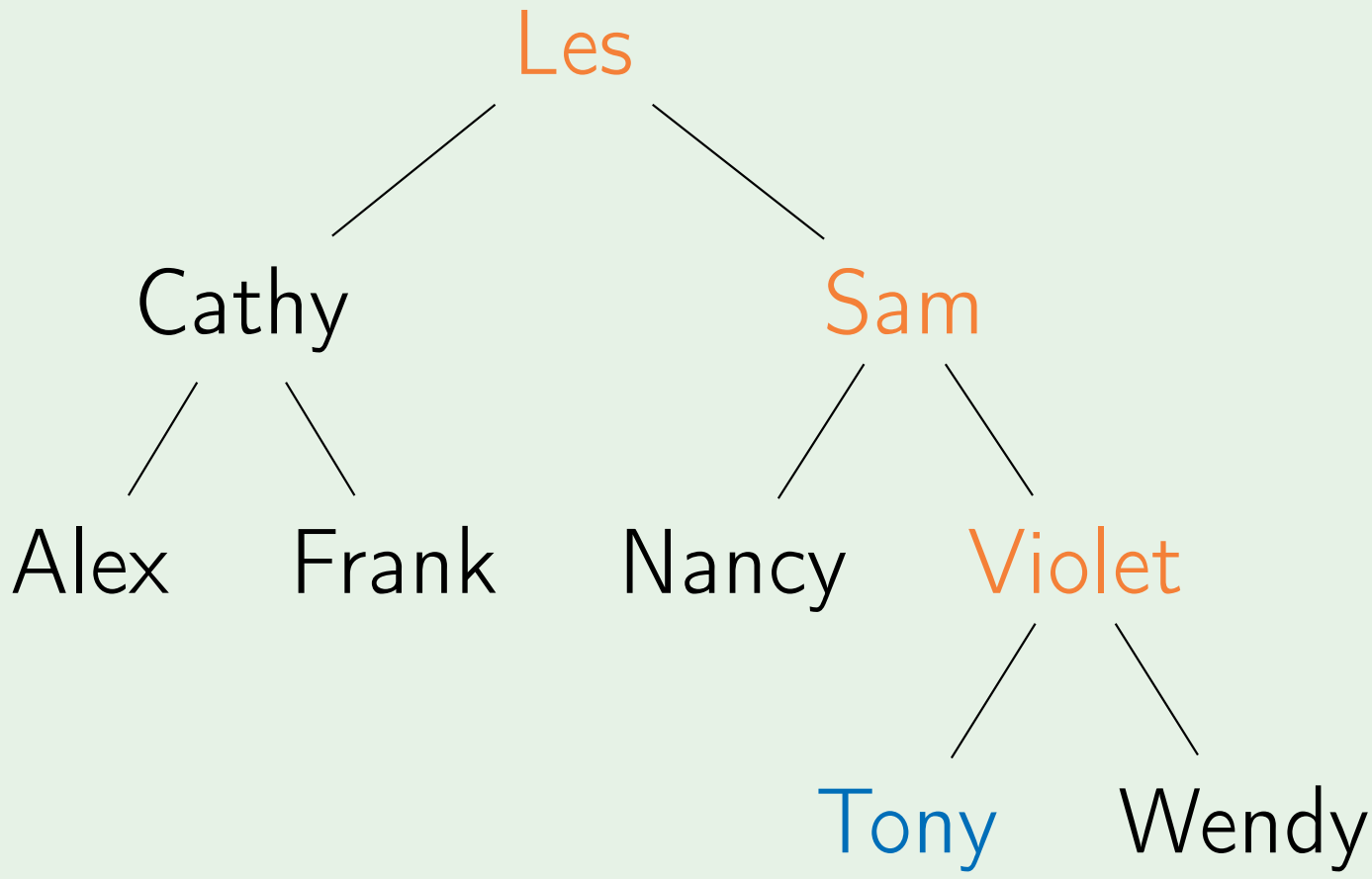
Output: Alex Frank Cathy Nancy

PostOrderTraversal



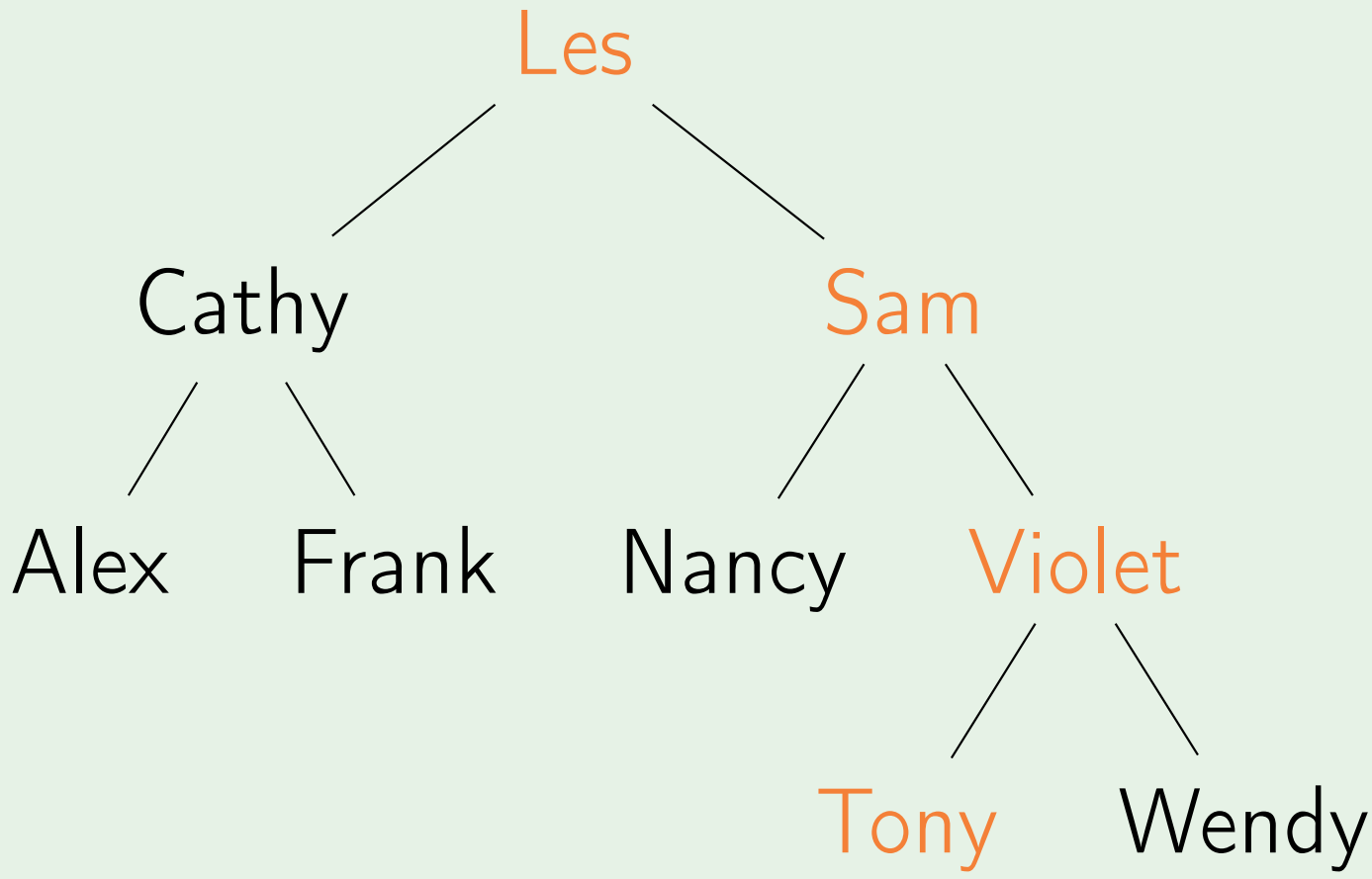
Output: Alex Frank Cathy Nancy

PostOrderTraversal



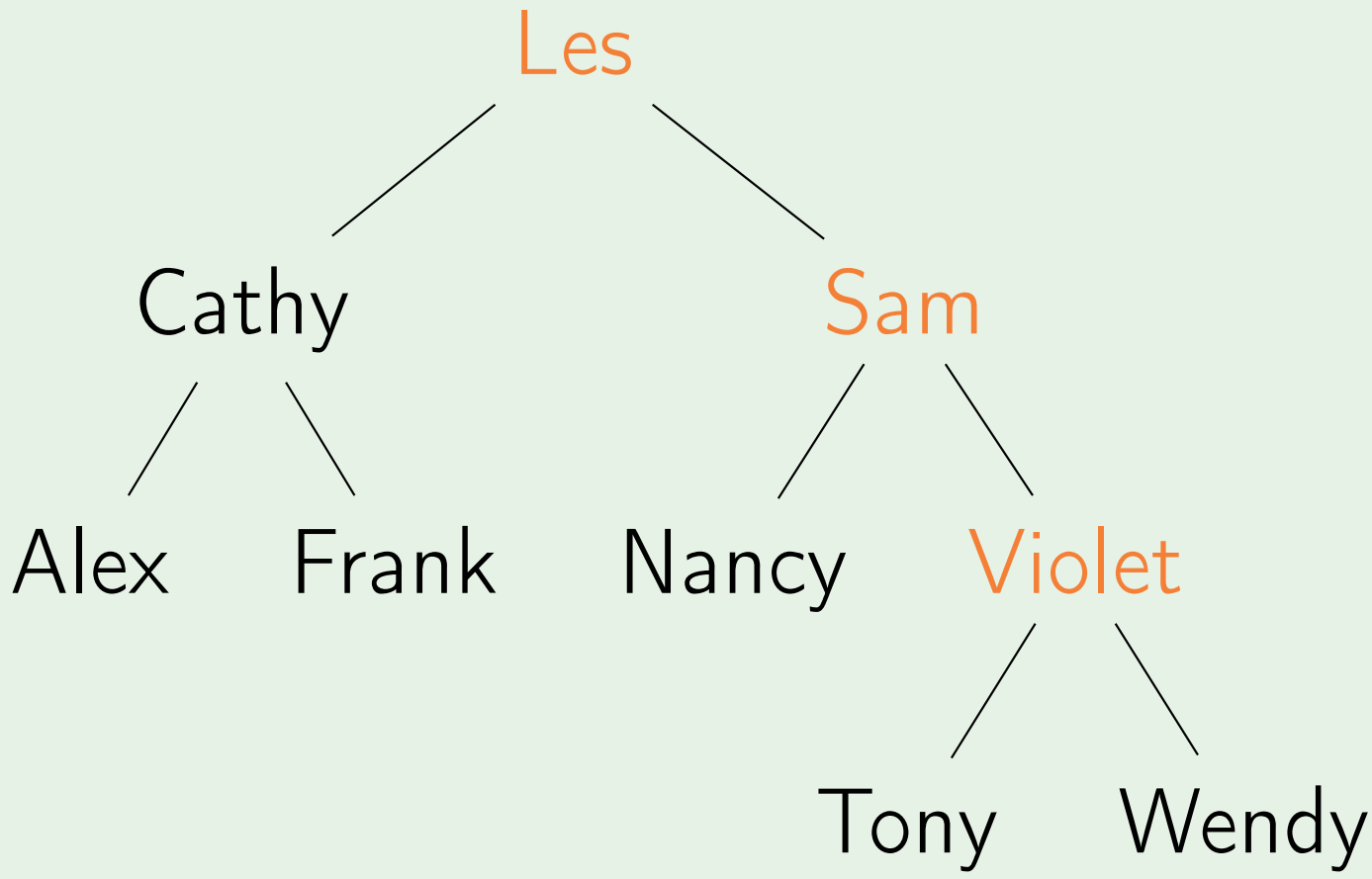
Output: Alex Frank Cathy Nancy Tony

PostOrderTraversal



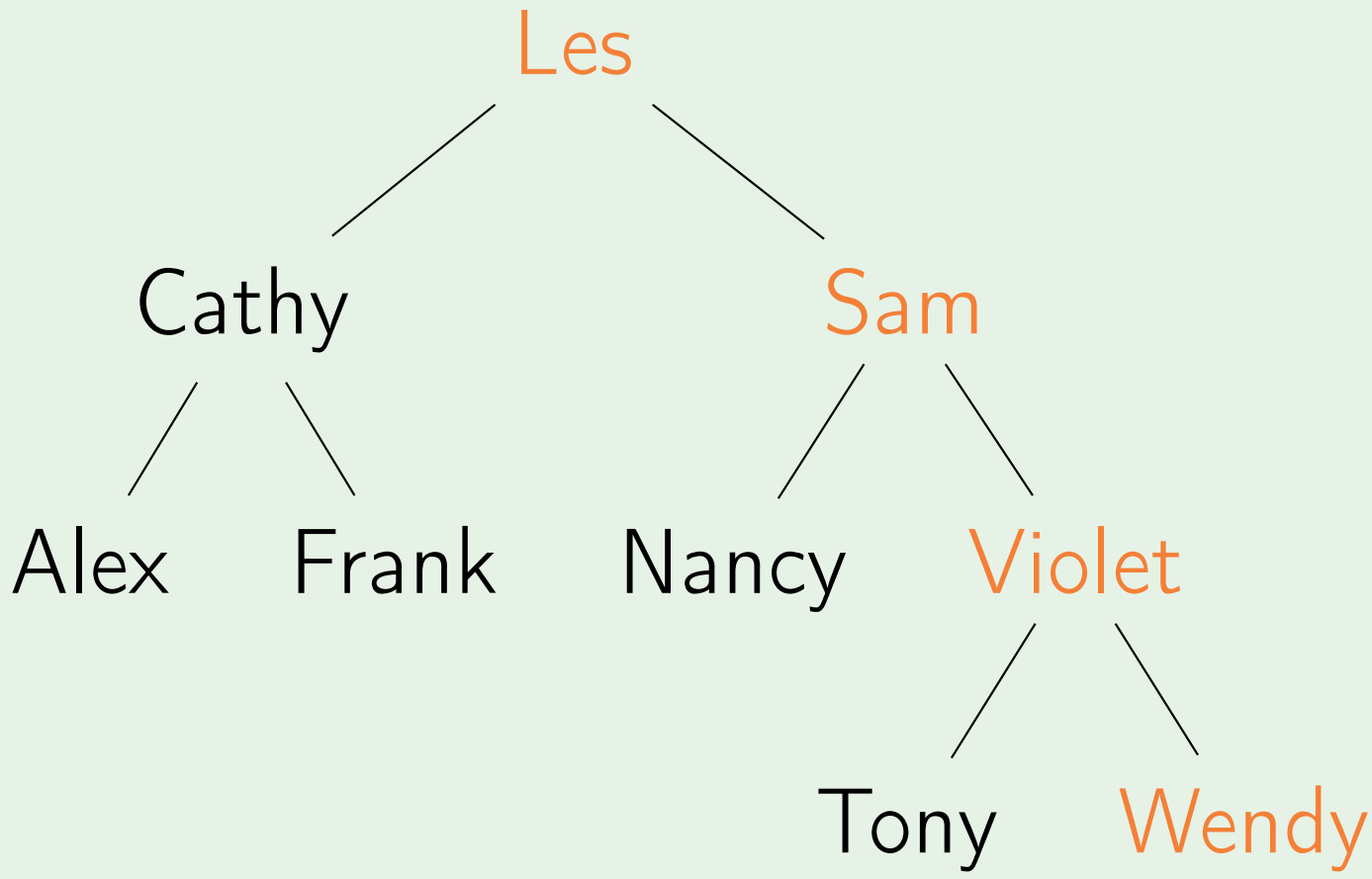
Output: Alex Frank Cathy Nancy Tony

PostOrderTraversal



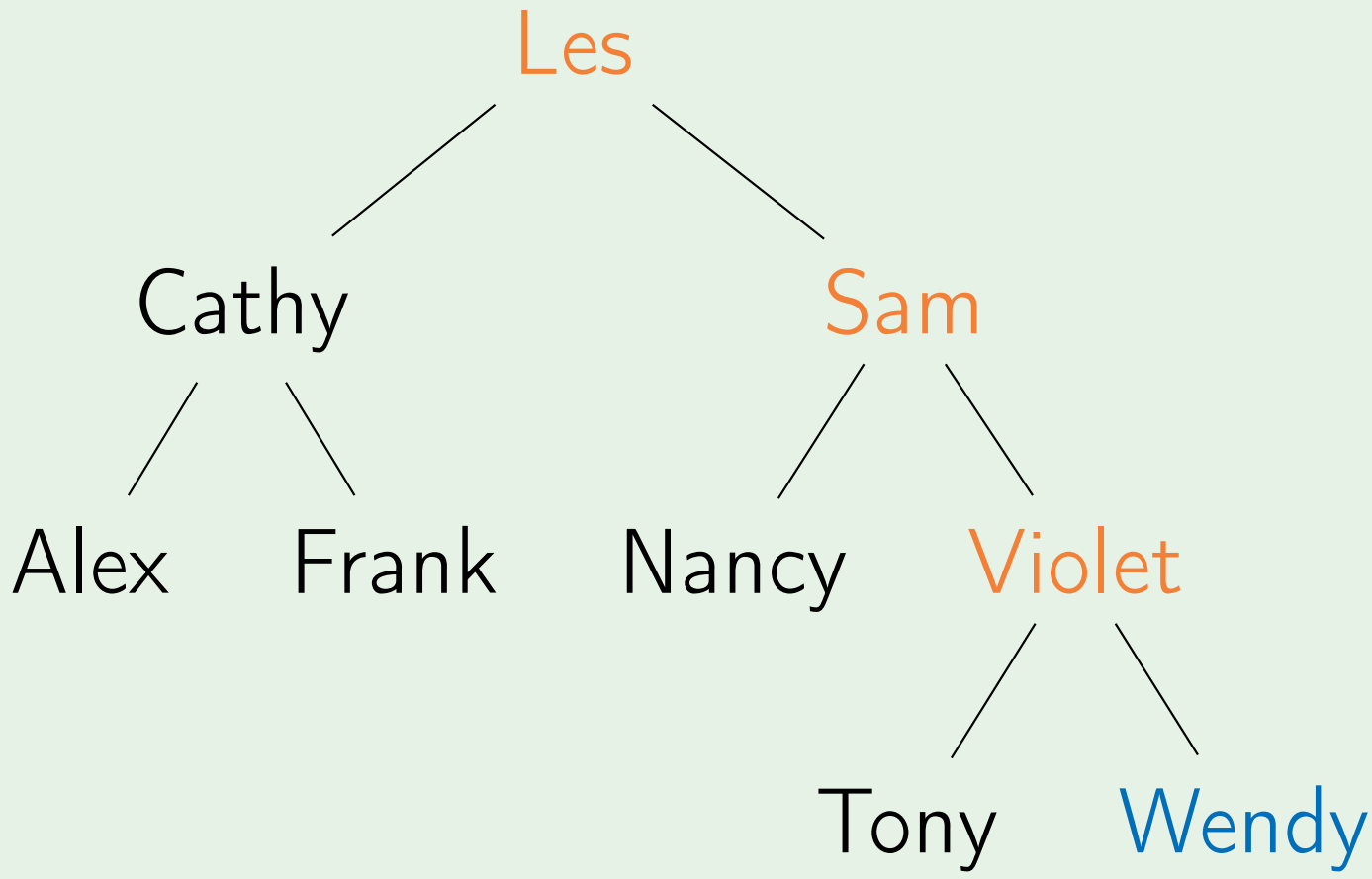
Output: Alex Frank Cathy Nancy Tony

PostOrderTraversal



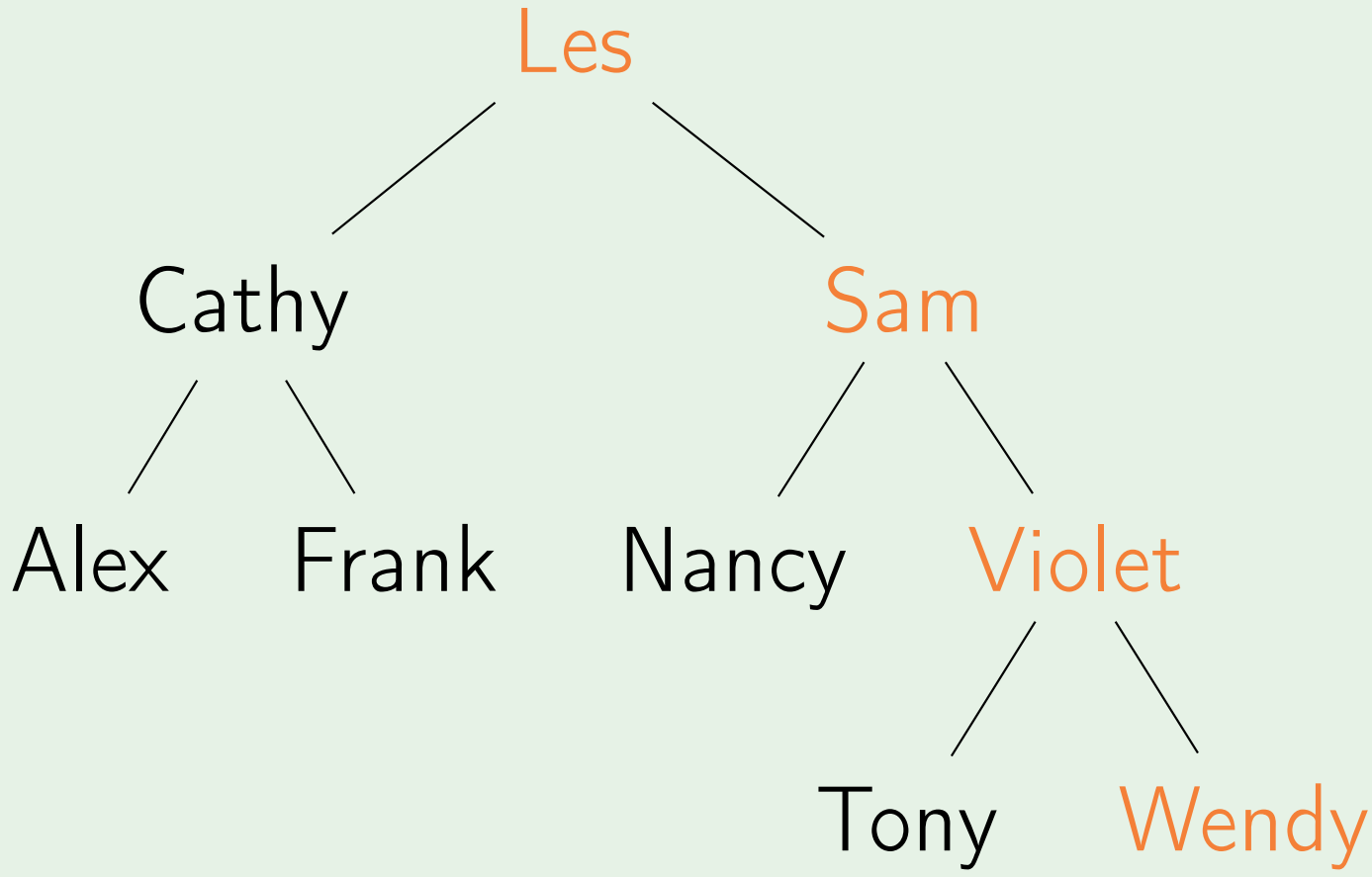
Output: Alex Frank Cathy Nancy Tony

PostOrderTraversal



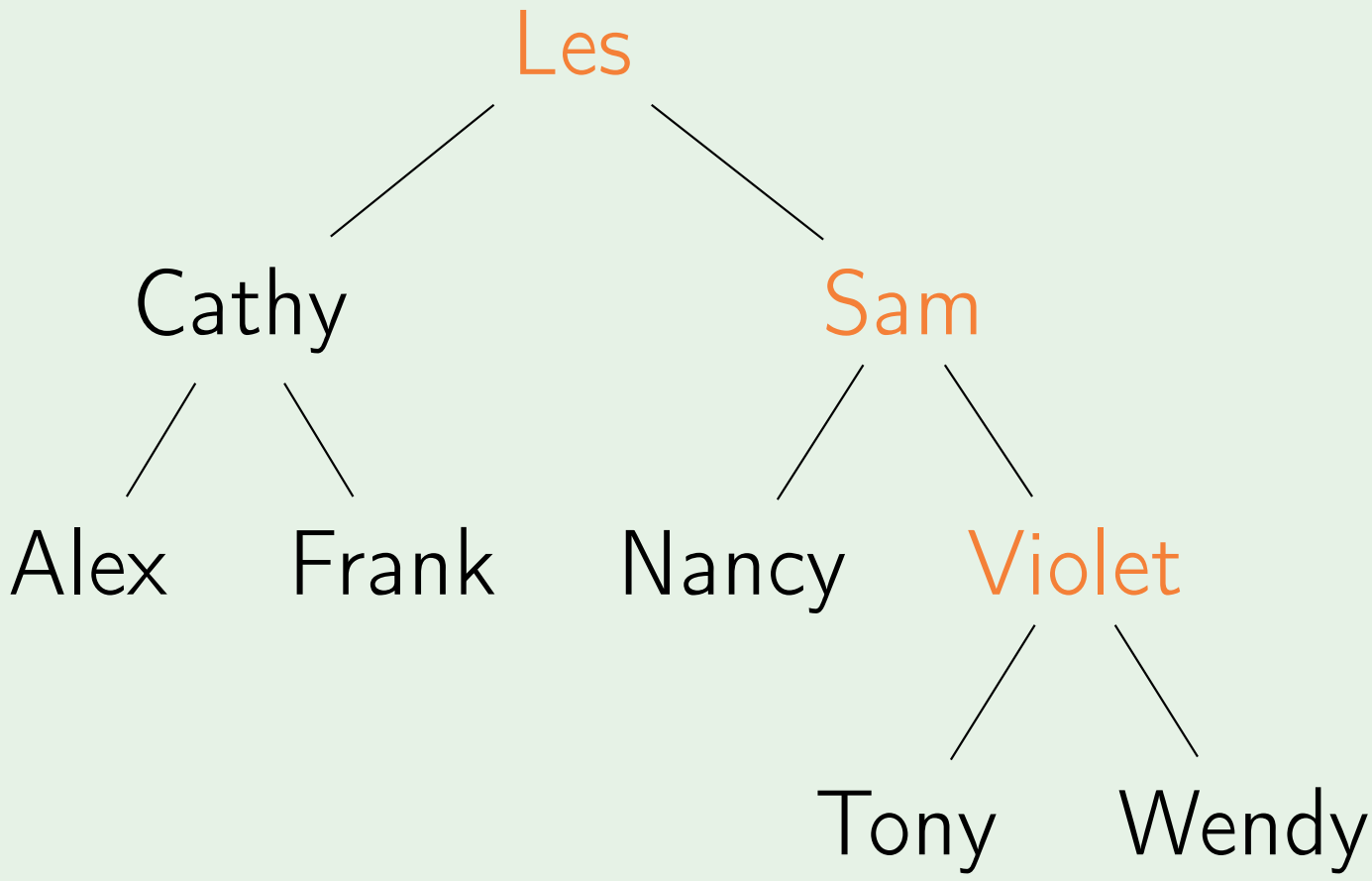
Output: Alex Frank Cathy Nancy Tony Wendy

PostOrderTraversal



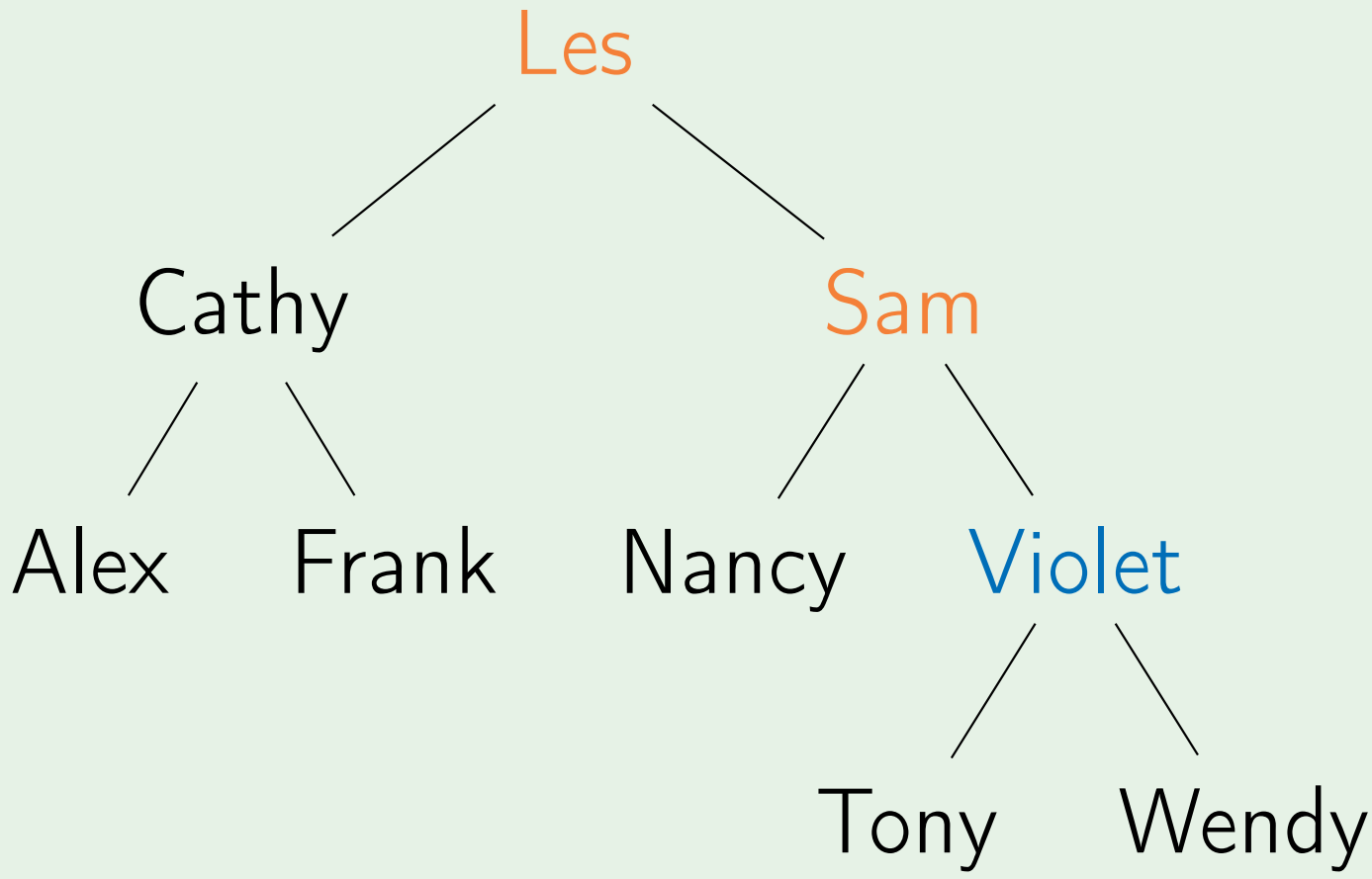
Output: Alex Frank Cathy Nancy Tony Wendy

PostOrderTraversal



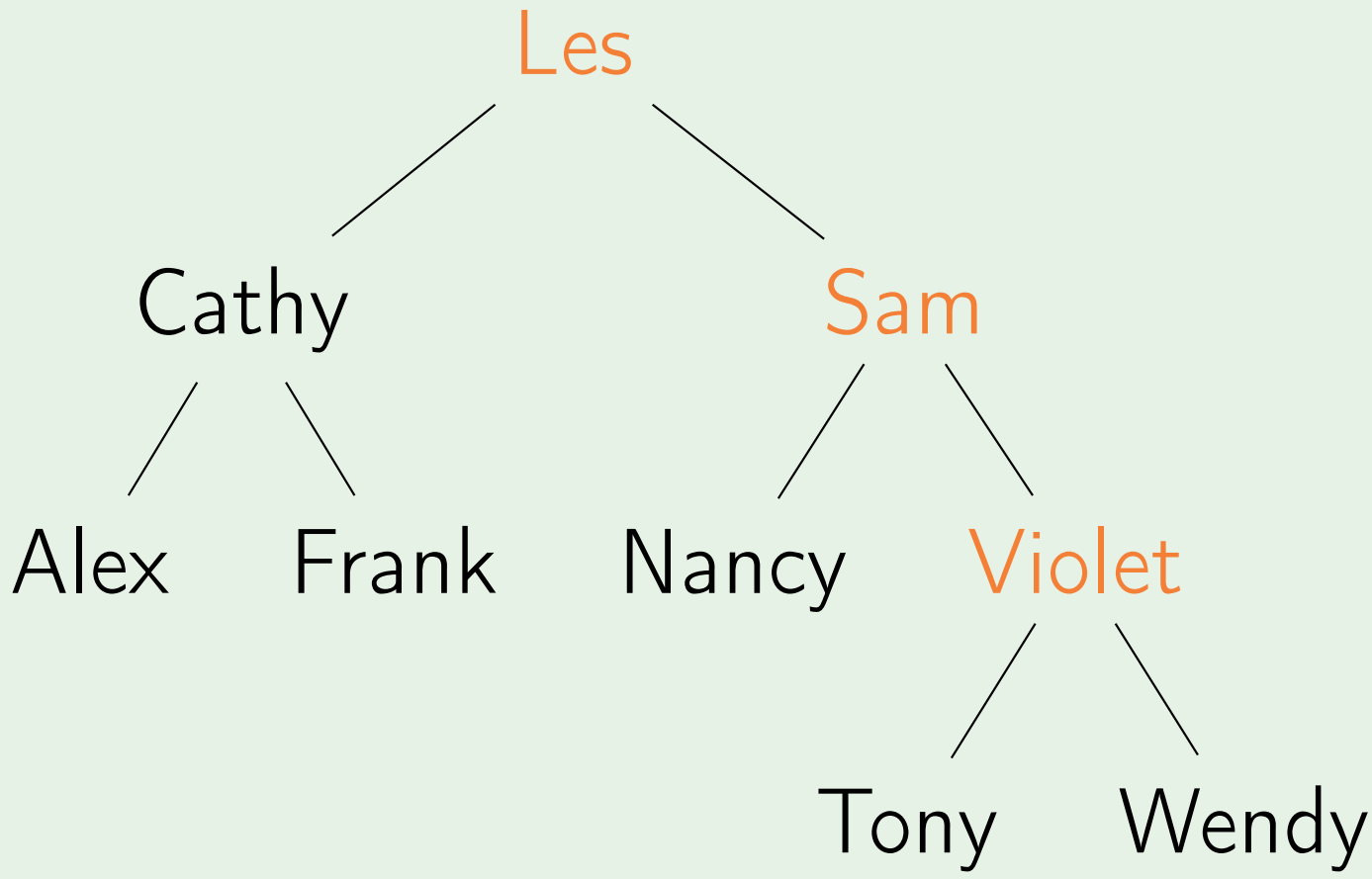
Output: Alex Frank Cathy Nancy Tony Wendy

PostOrderTraversal



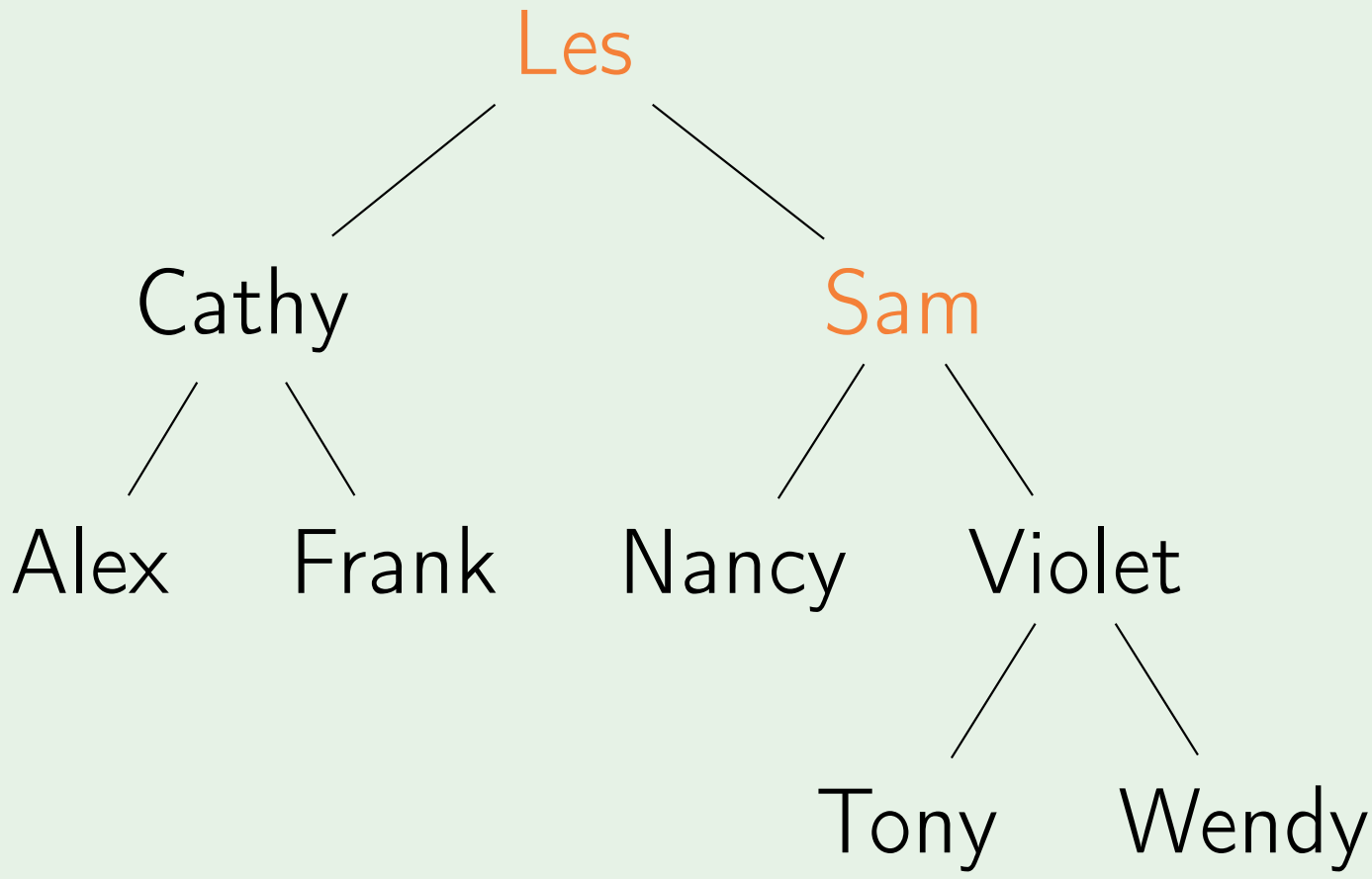
Output: Alex Frank Cathy Nancy Tony Wendy Violet

PostOrderTraversal



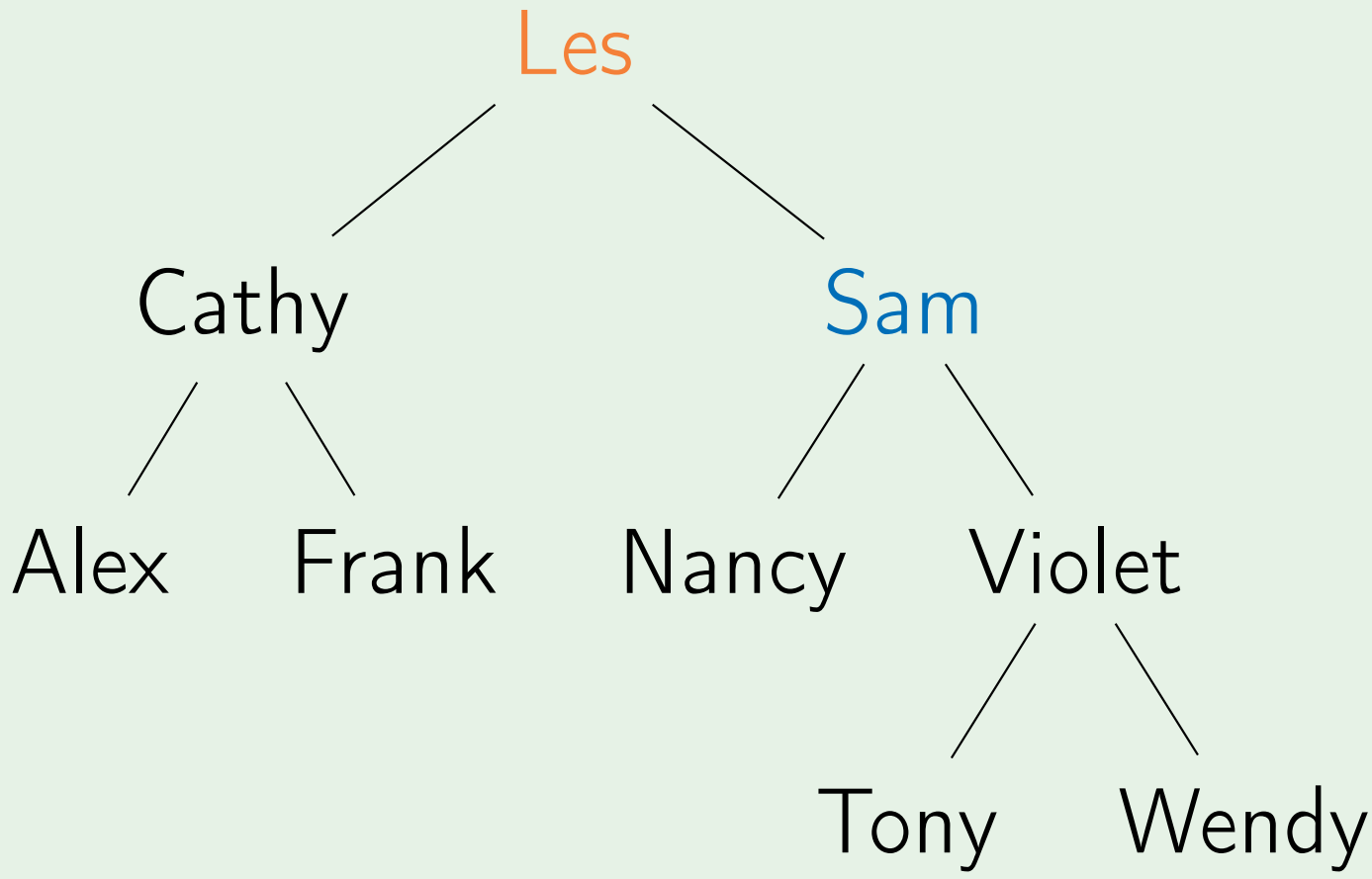
Output: Alex Frank Cathy Nancy Tony Wendy Violet

PostOrderTraversal



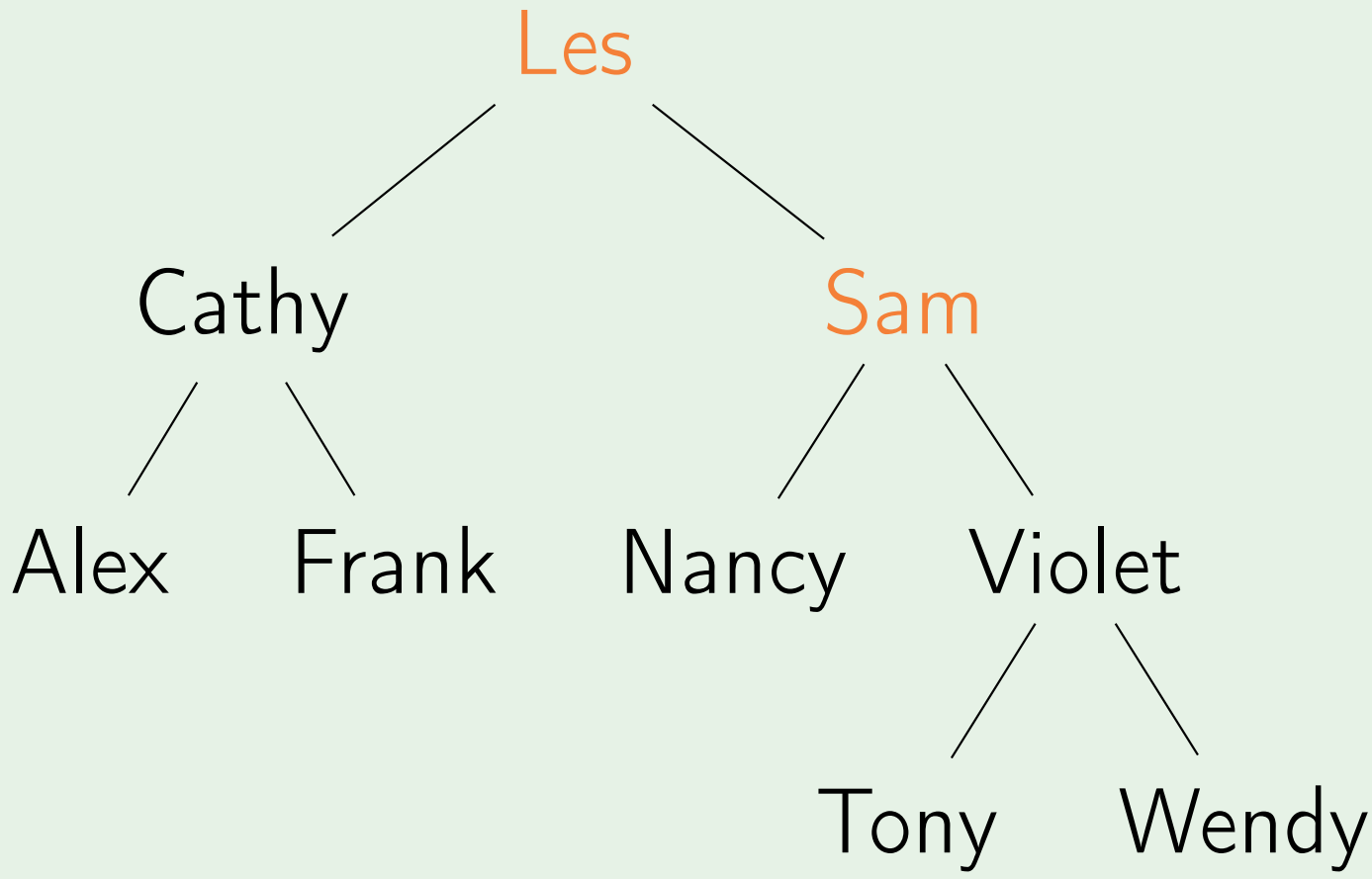
Output: Alex Frank Cathy Nancy Tony Wendy Violet

PostOrderTraversal



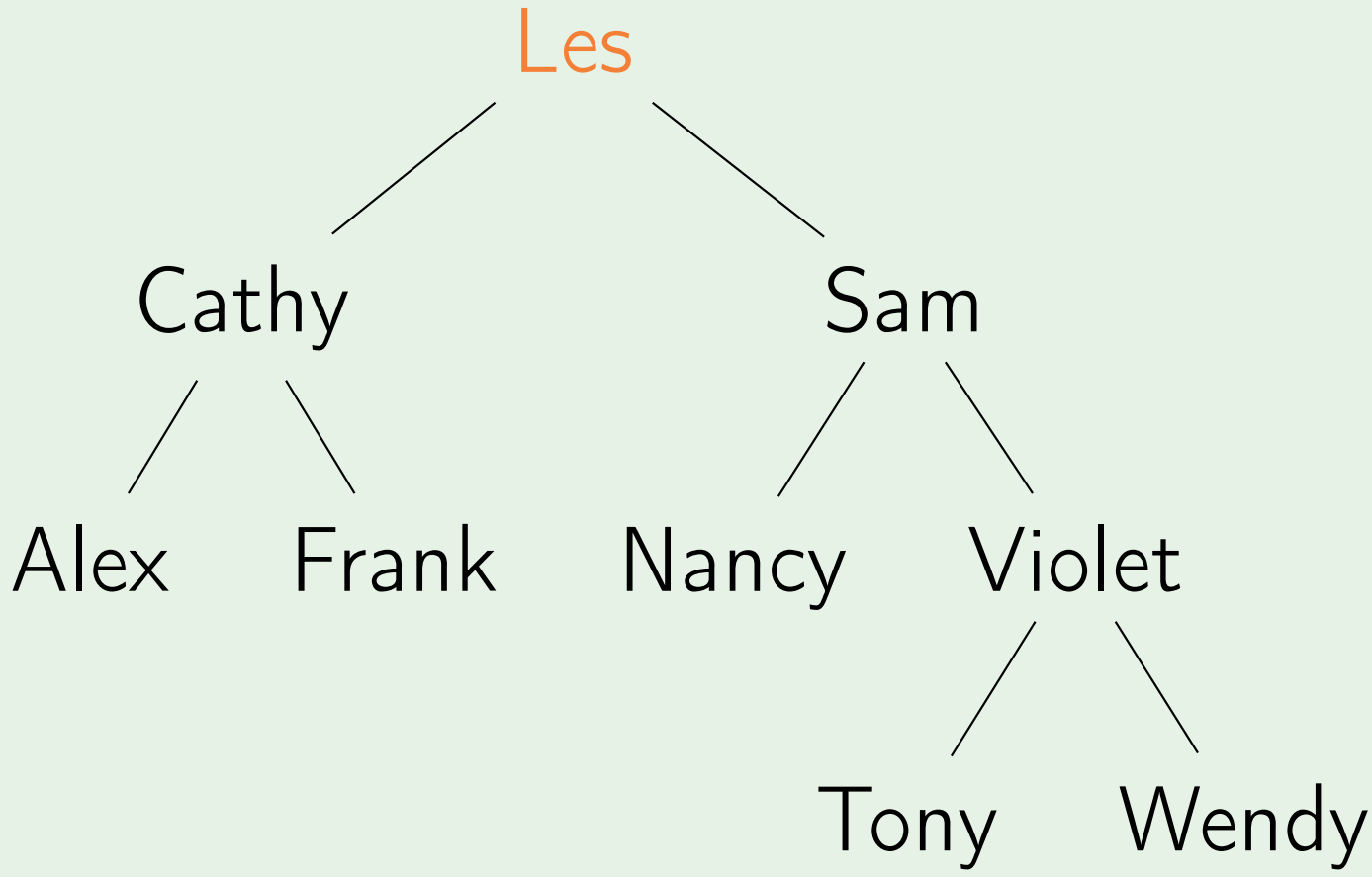
Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam

PostOrderTraversal



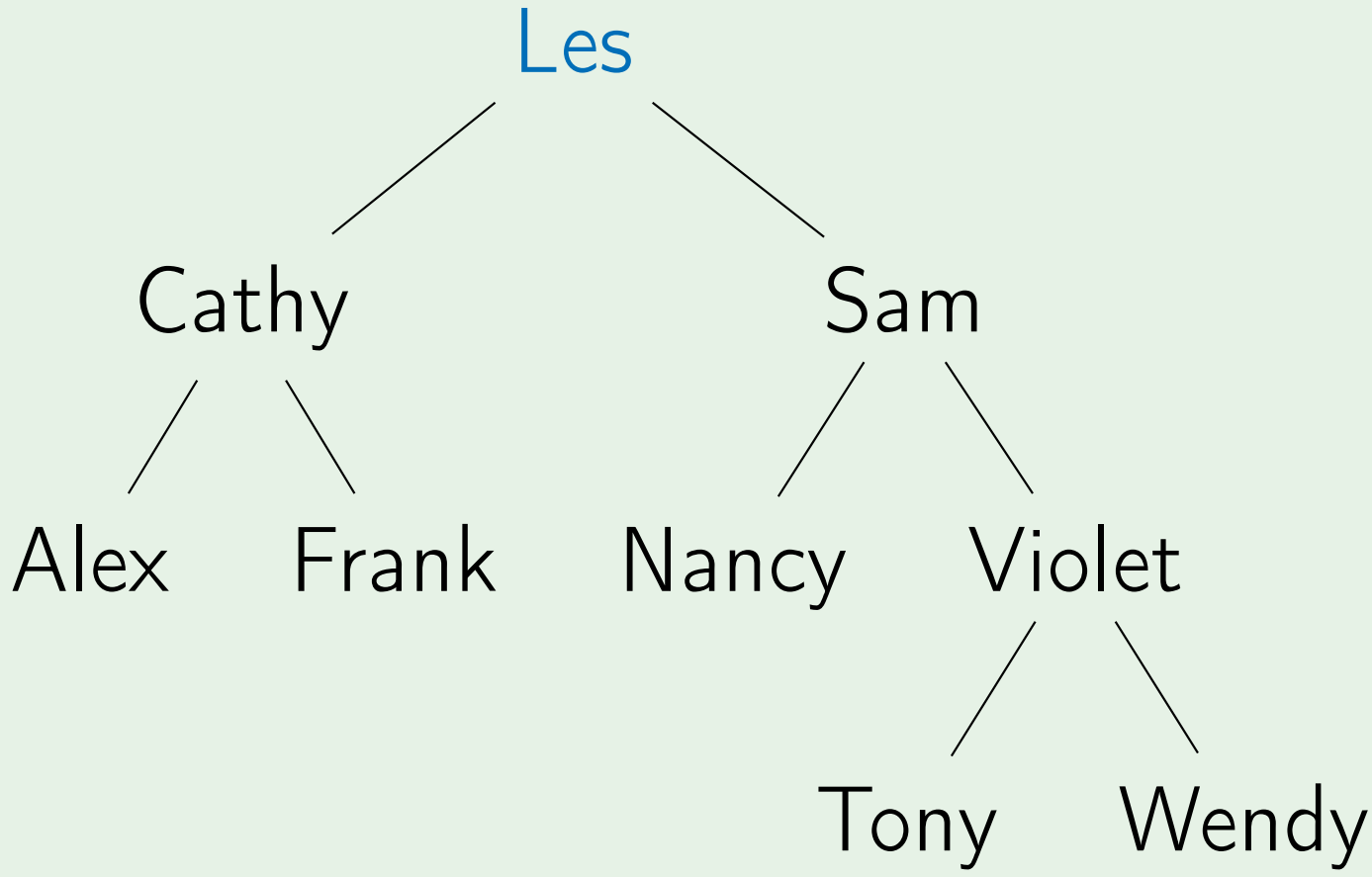
Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam

PostOrderTraversal



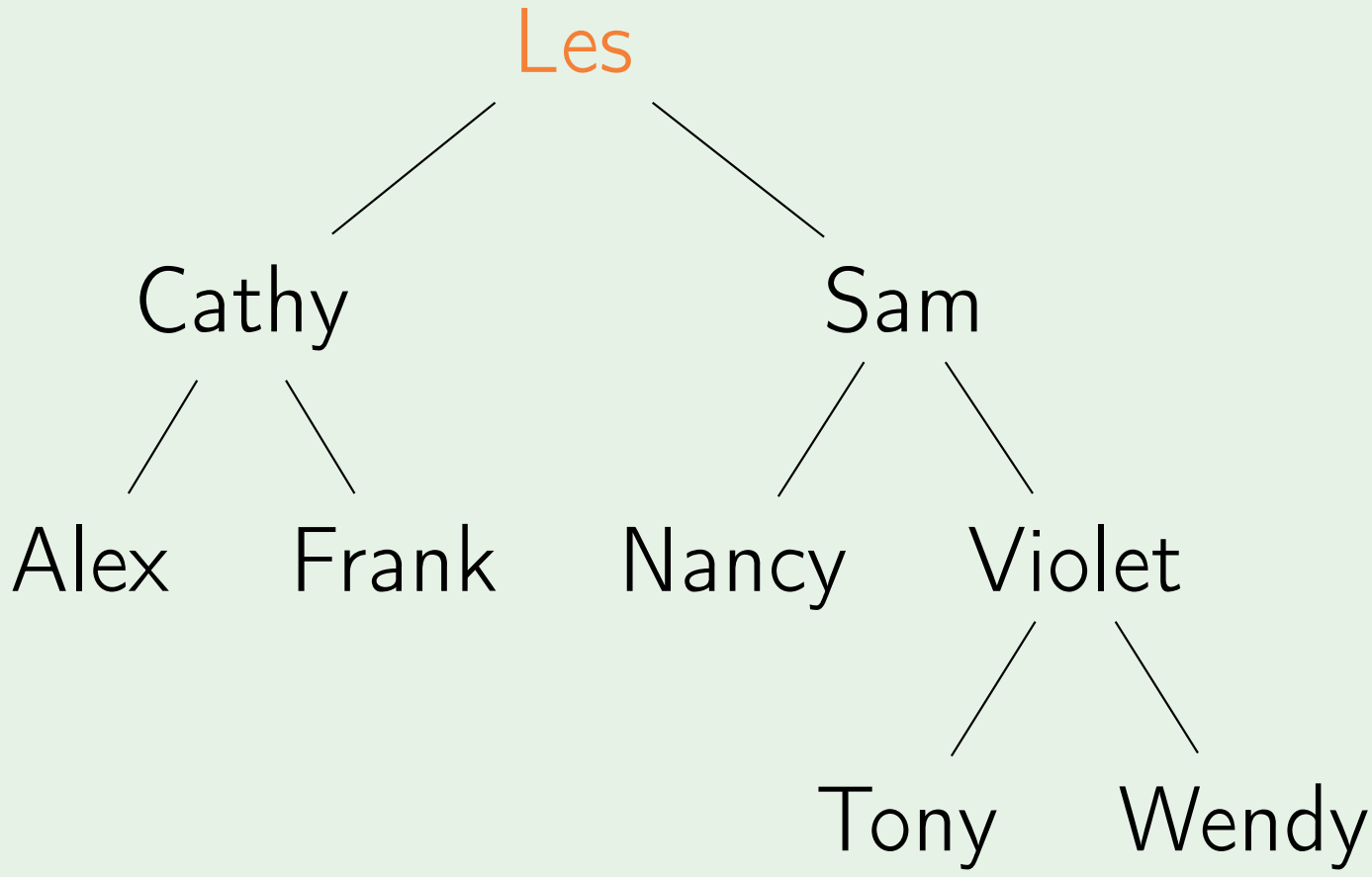
Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam

PostOrderTraversal



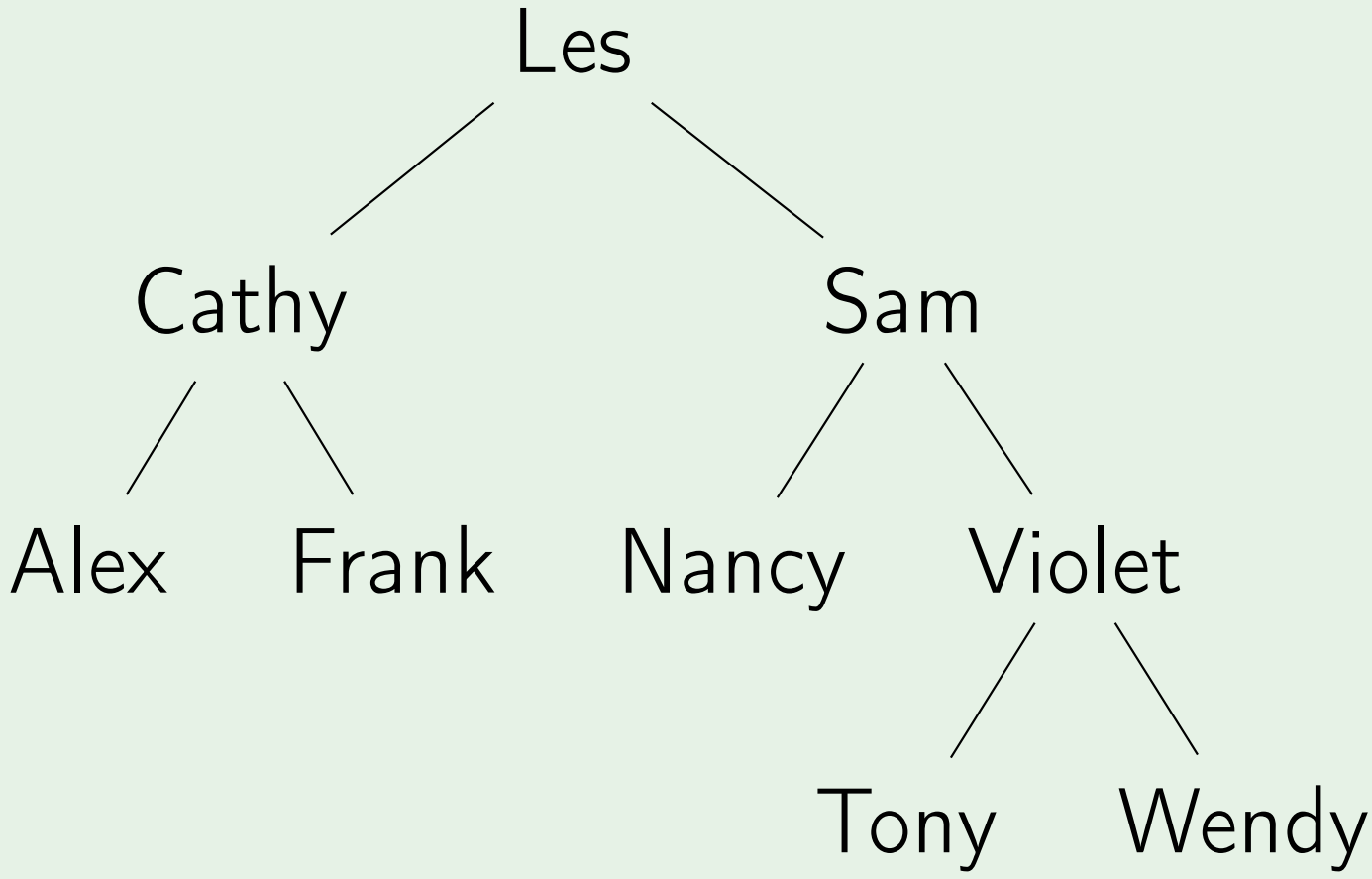
Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam Les

PostOrderTraversal



Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam Les

PostOrderTraversal



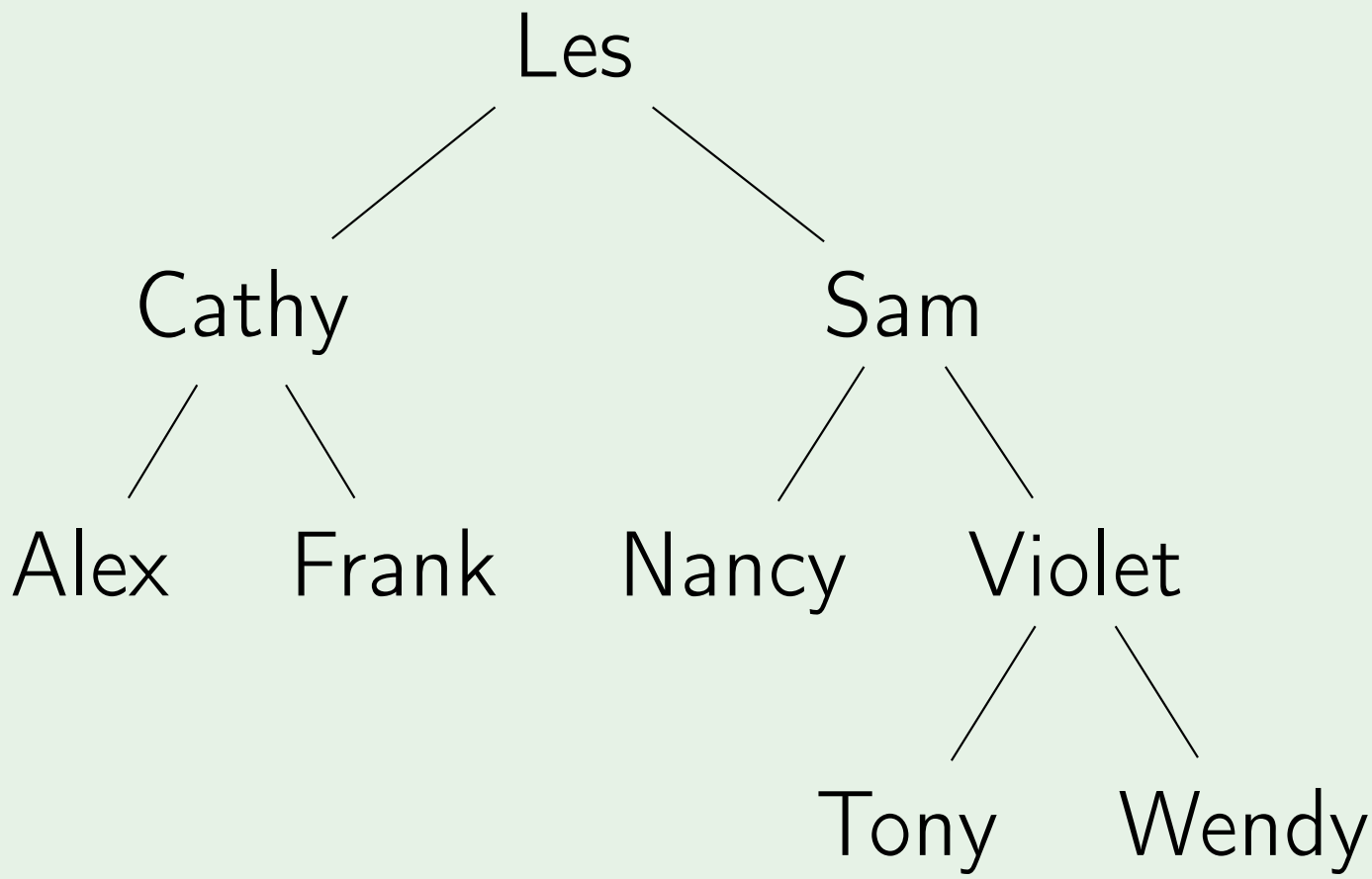
Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam Les

Breadth-first

```
LevelTraversal(tree)
if tree = nil:  return

Queue q
q.Enqueue(tree)
while not q.Empty():
    node ← q.Dequeue()
    Print(node)
    if node.left ≠ nil:
        q.Enqueue(node.left)
    if node.right ≠ nil:
        q.Enqueue(node.right)
```

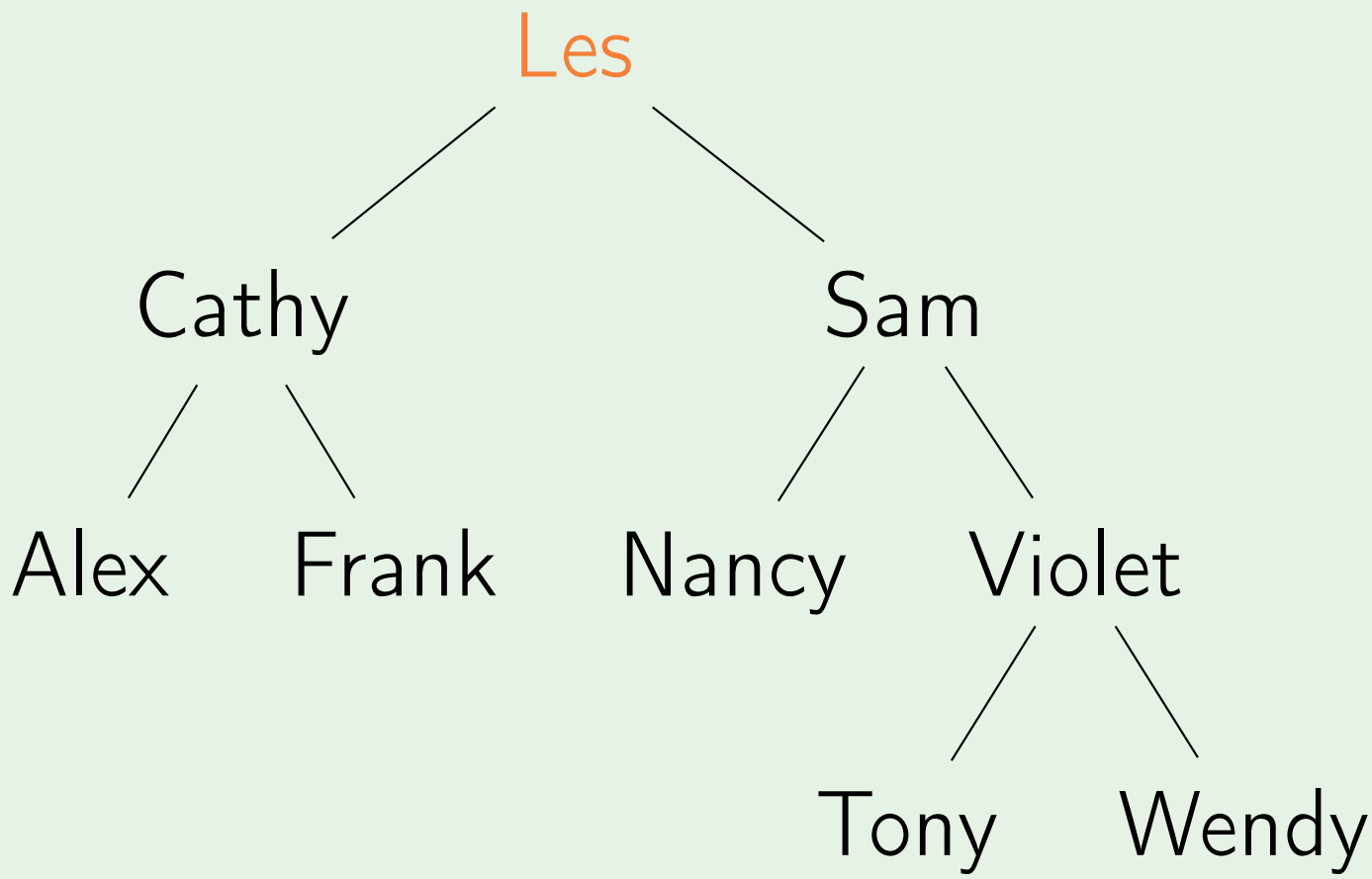
LevelTraversal



Output:

Queue: Les

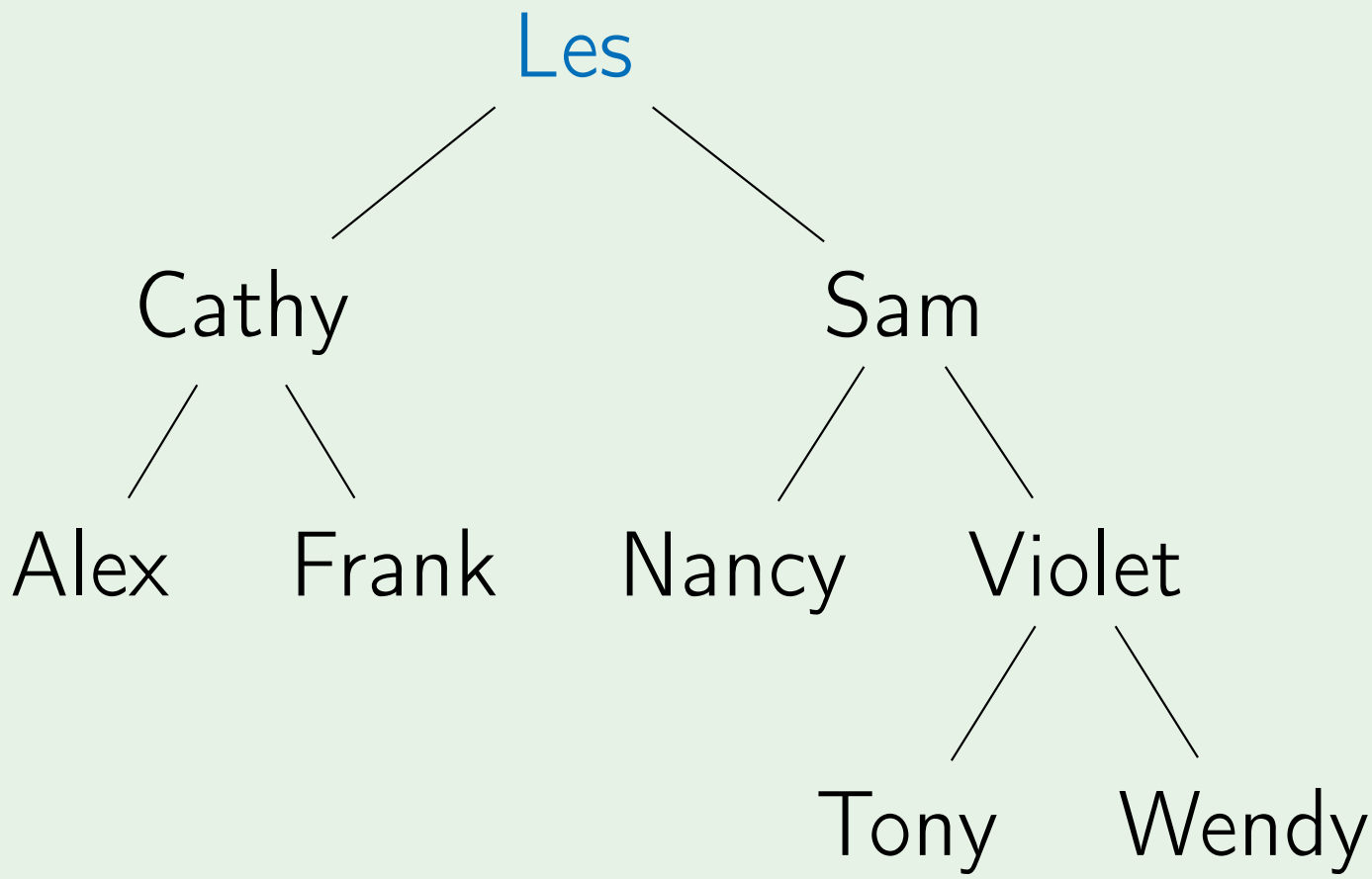
LevelTraversal



Output:

Queue:

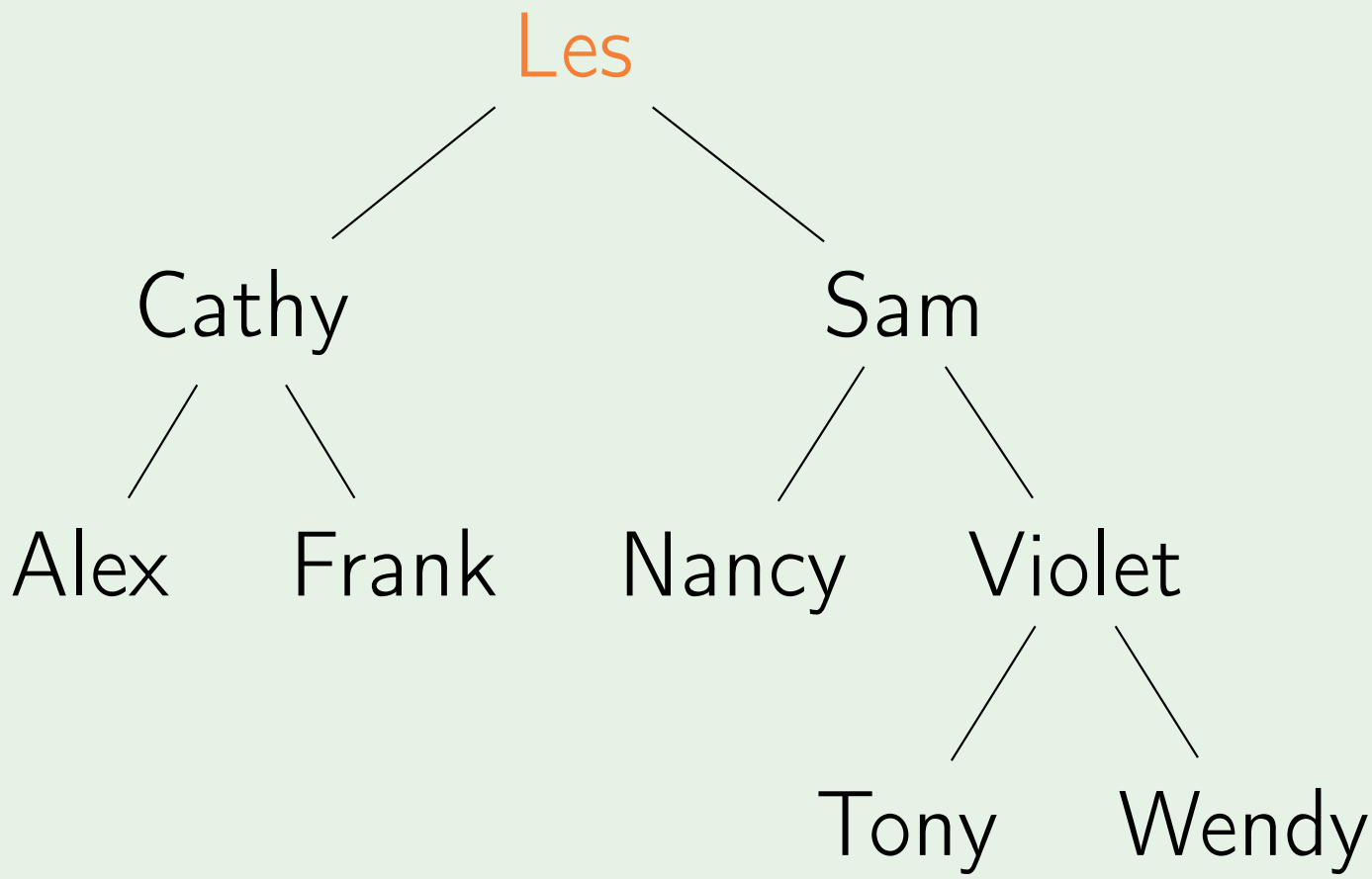
LevelTraversal



Output: Les

Queue:

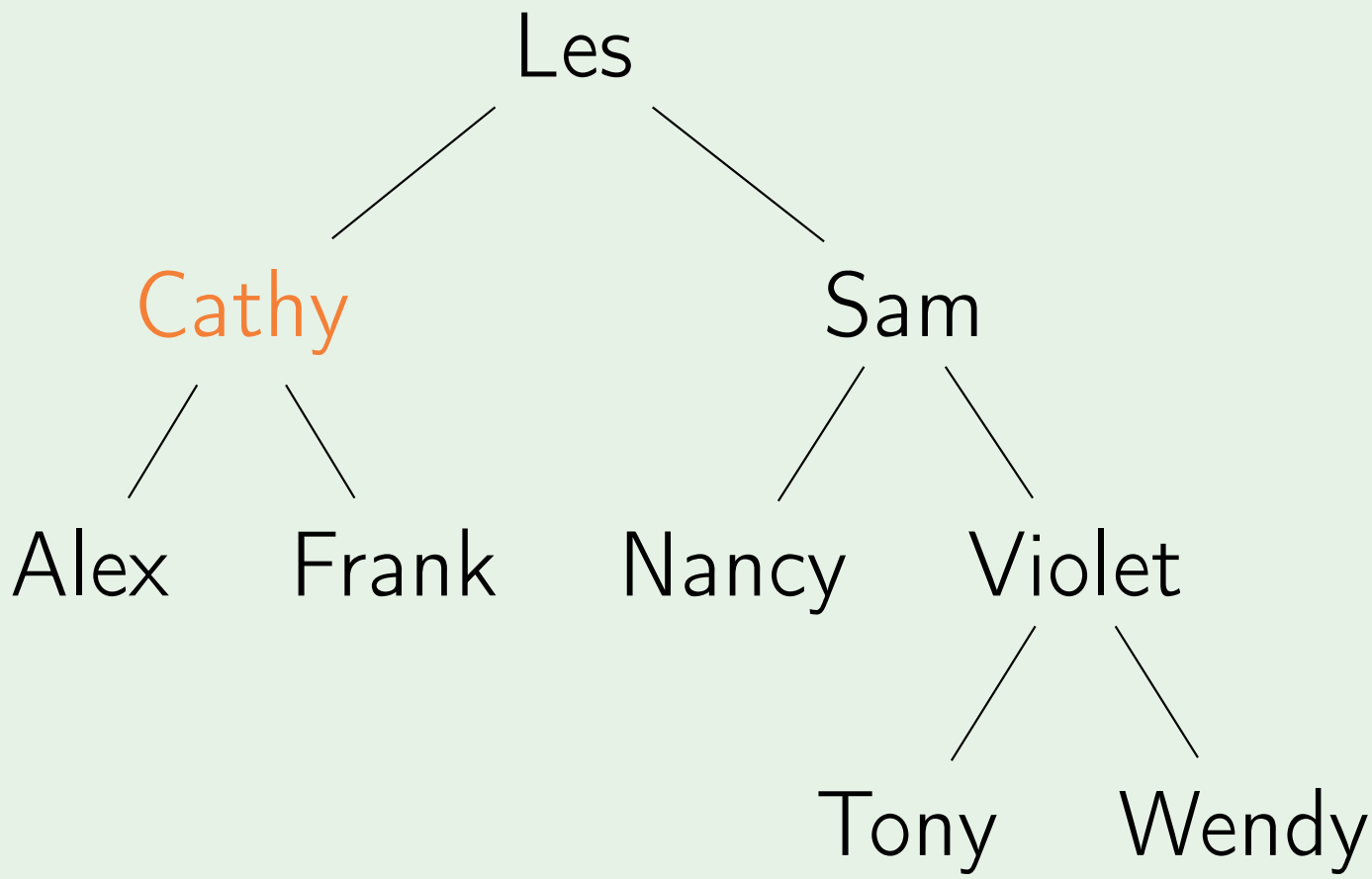
LevelTraversal



Output: Les

Queue: Cathy, Sam

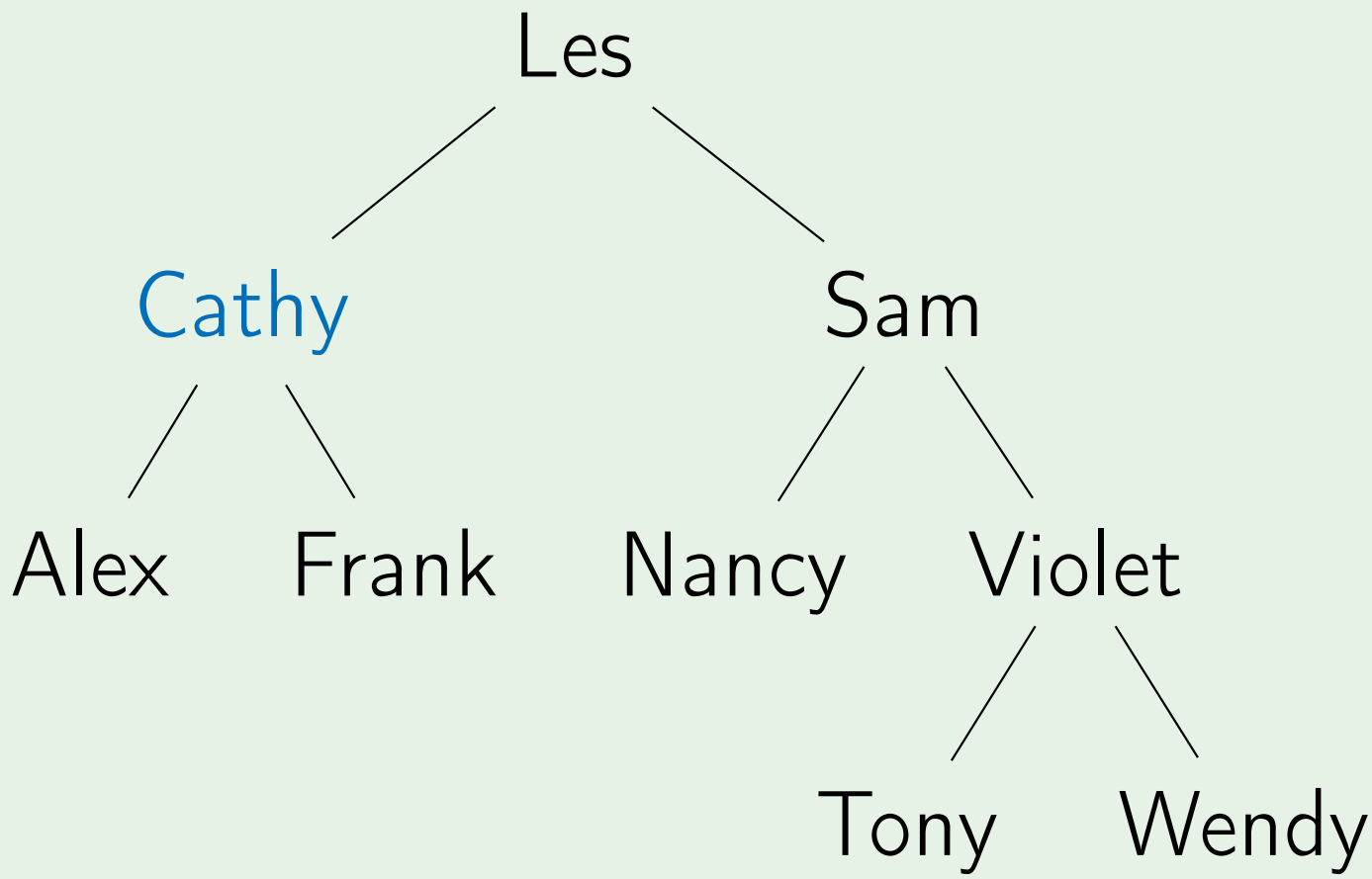
LevelTraversal



Output: Les

Queue: Sam

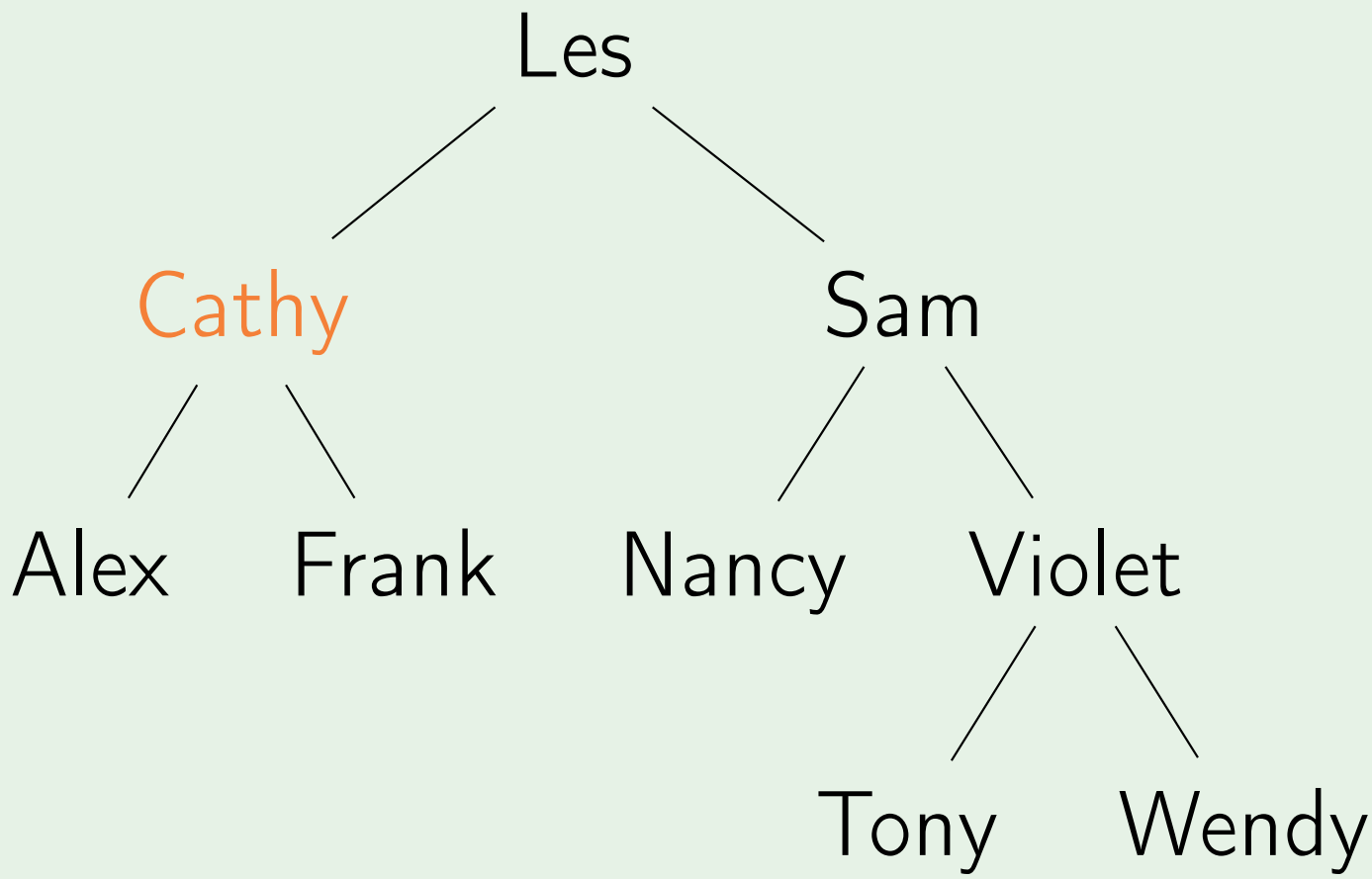
LevelTraversal



Output: Les Cathy

Queue: Sam

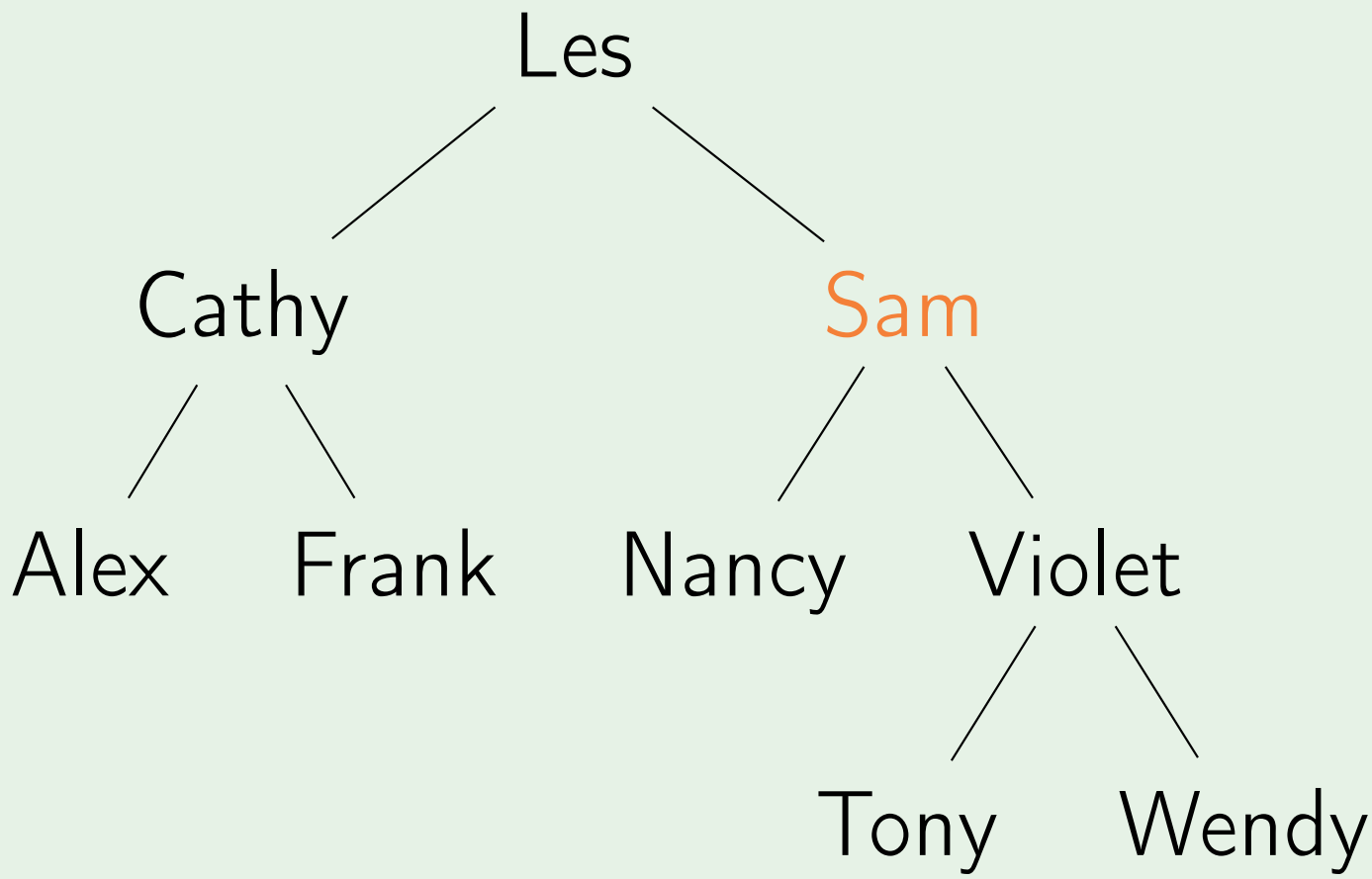
LevelTraversal



Output: Les Cathy

Queue: Sam,Alex, Frank

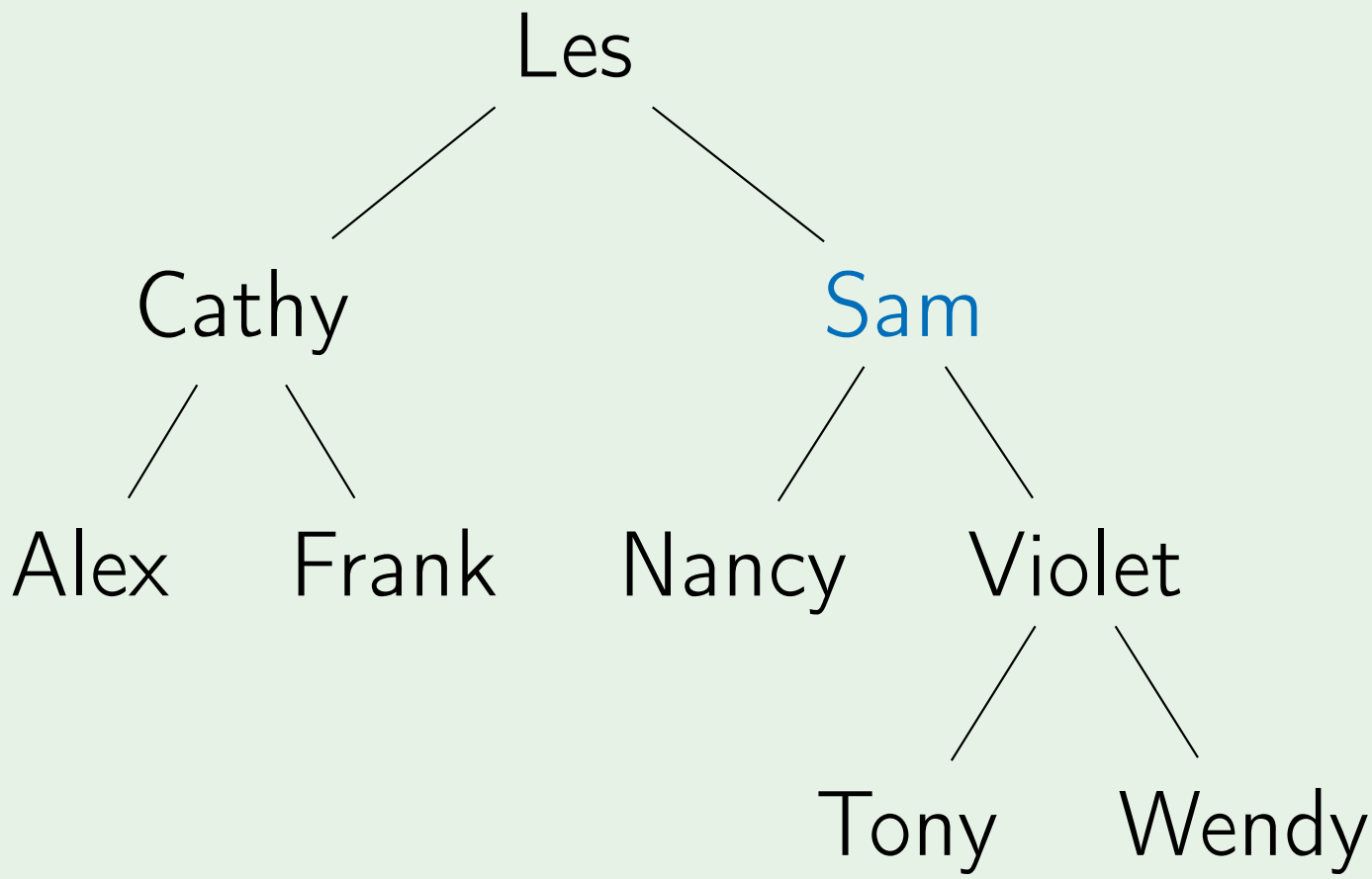
LevelTraversal



Output: Les Cathy

Queue: Alex, Frank

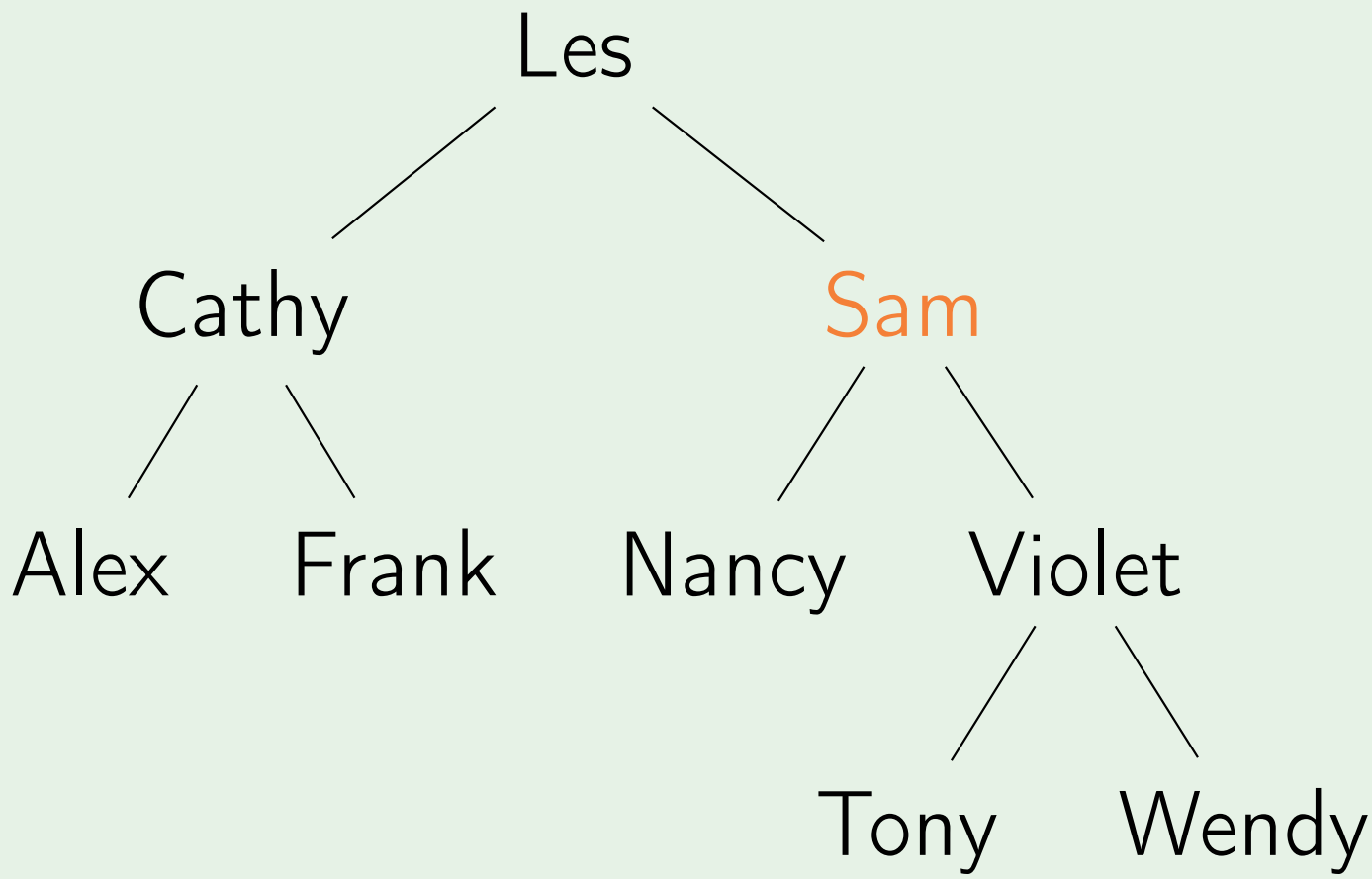
LevelTraversal



Output: Les Cathy Sam

Queue: Alex, Frank

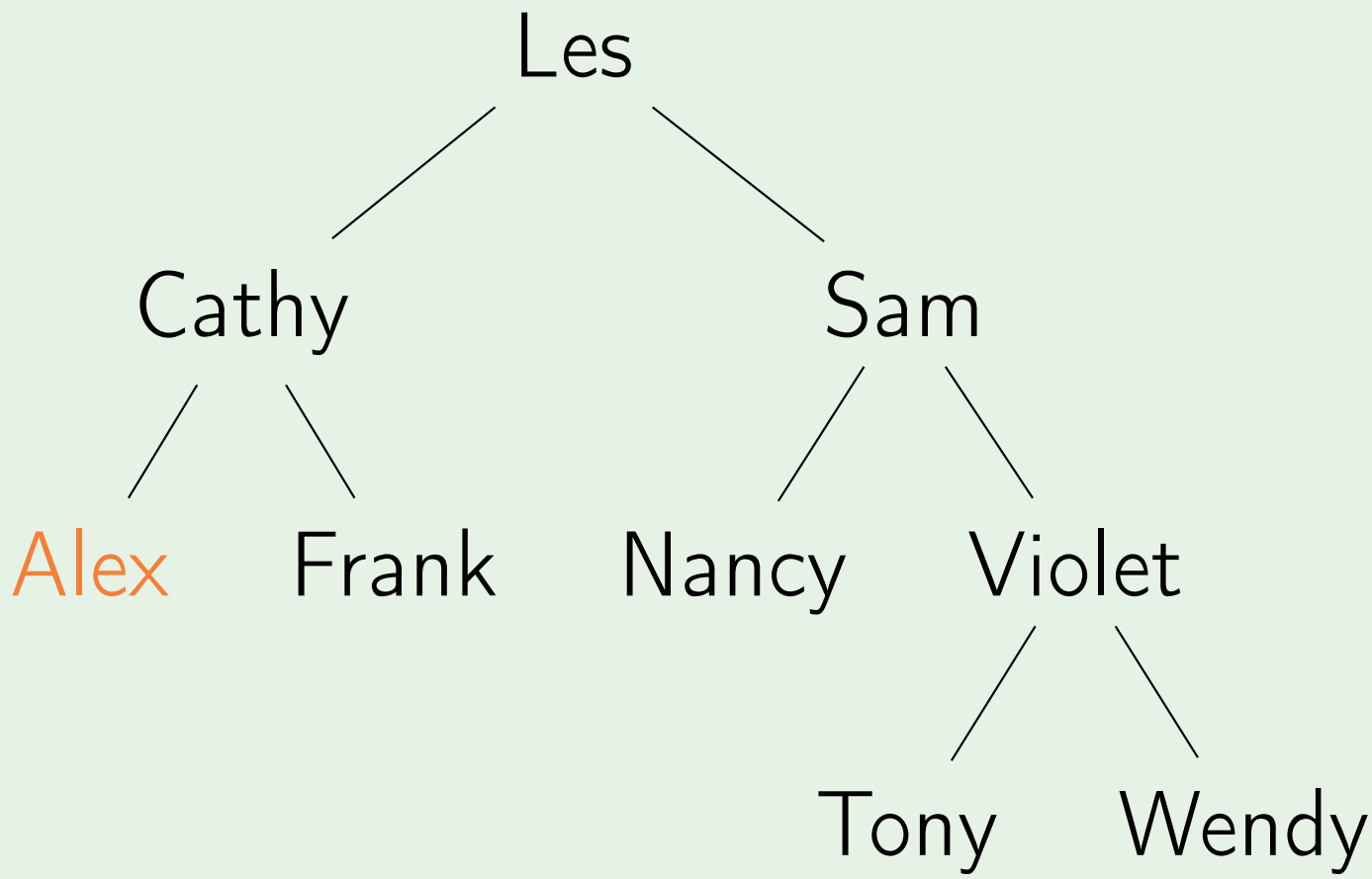
LevelTraversal



Output: Les Cathy Sam

Queue: Alex, Frank, Nancy, Violet

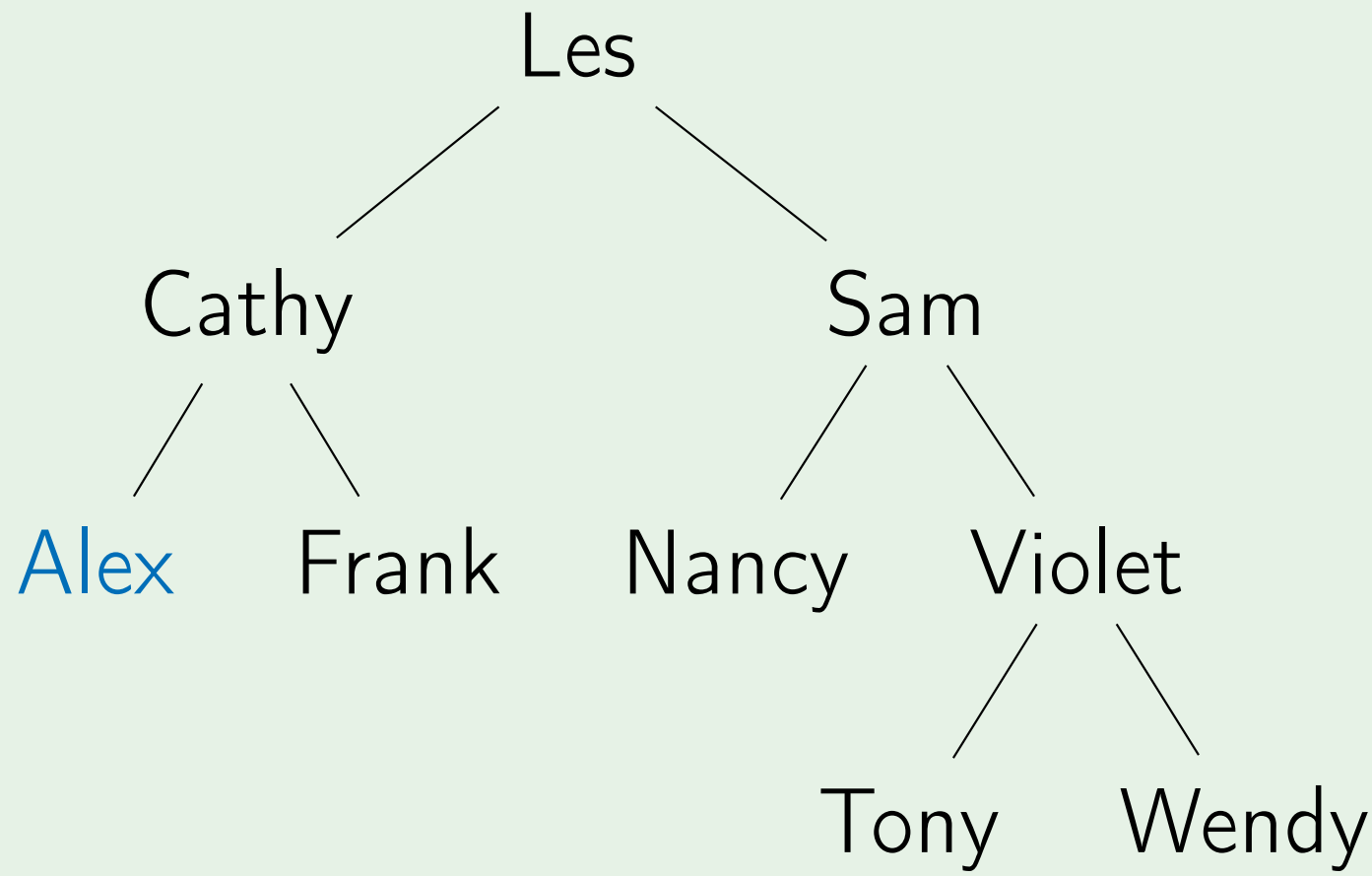
LevelTraversal



Output: Les Cathy Sam

Queue: Frank, Nancy, Violet

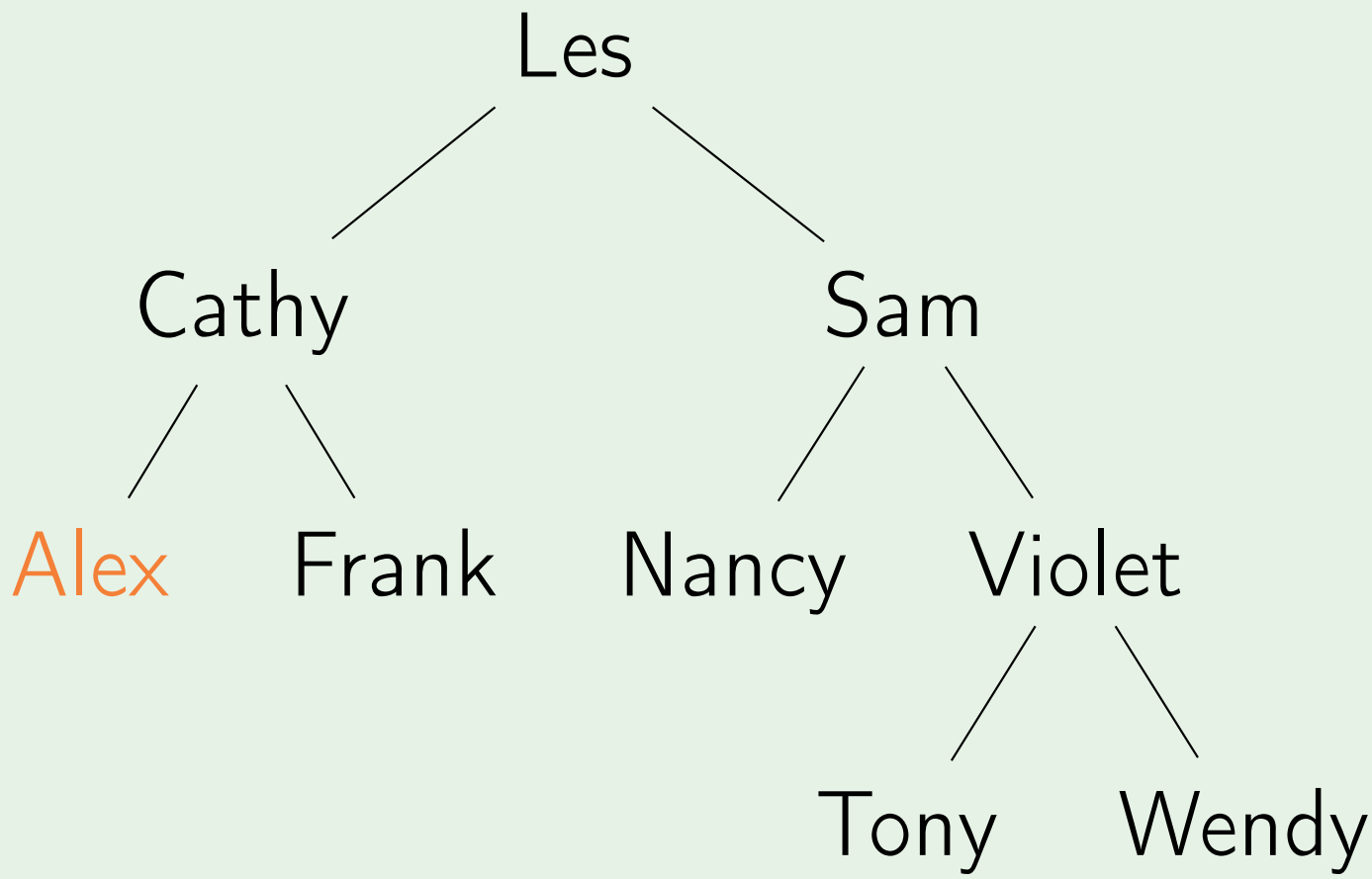
LevelTraversal



Output: Les Cathy Sam Alex

Queue: Frank, Nancy, Violet

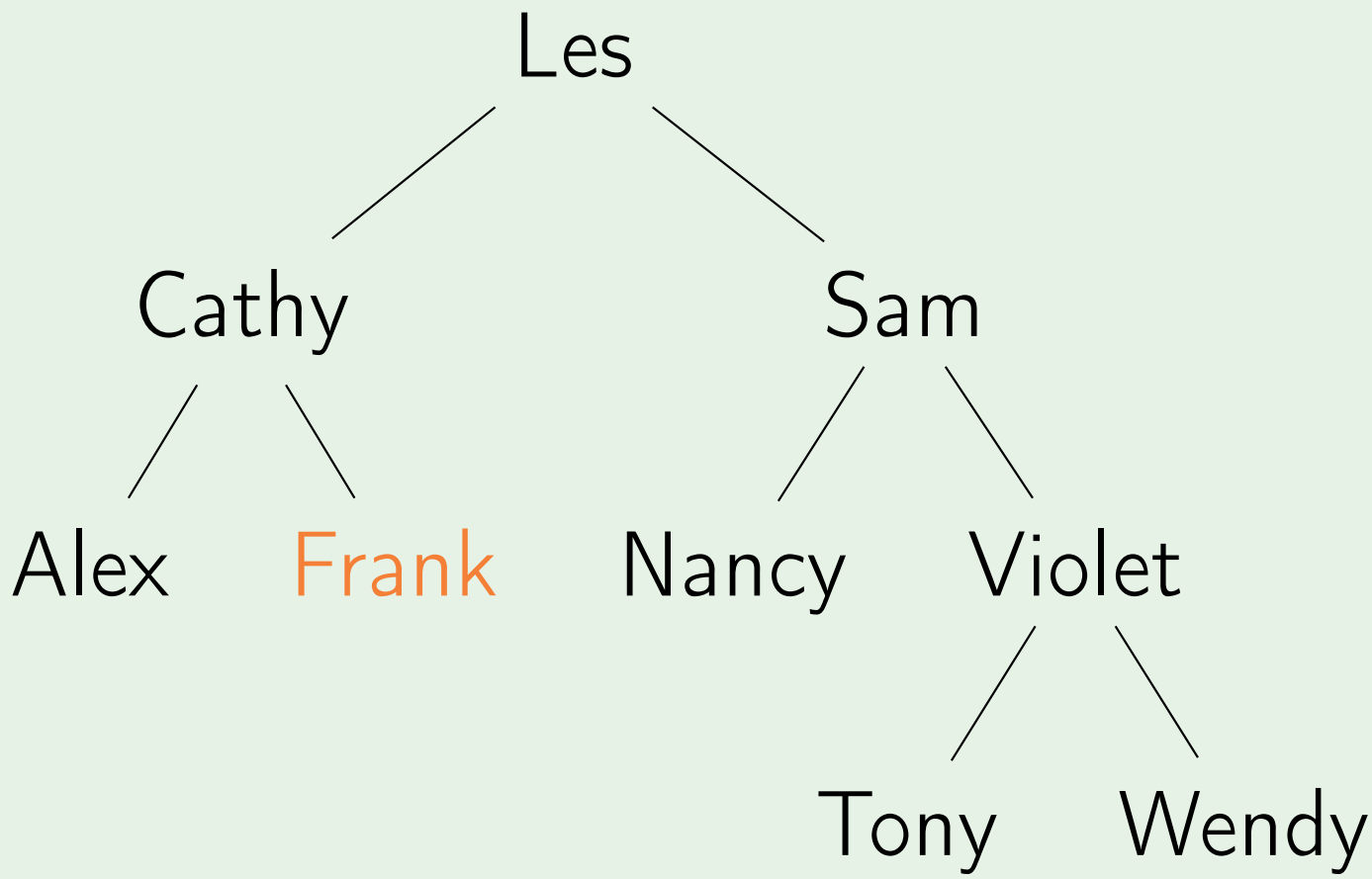
LevelTraversal



Output: Les Cathy Sam Alex

Queue: Frank, Nancy, Violet

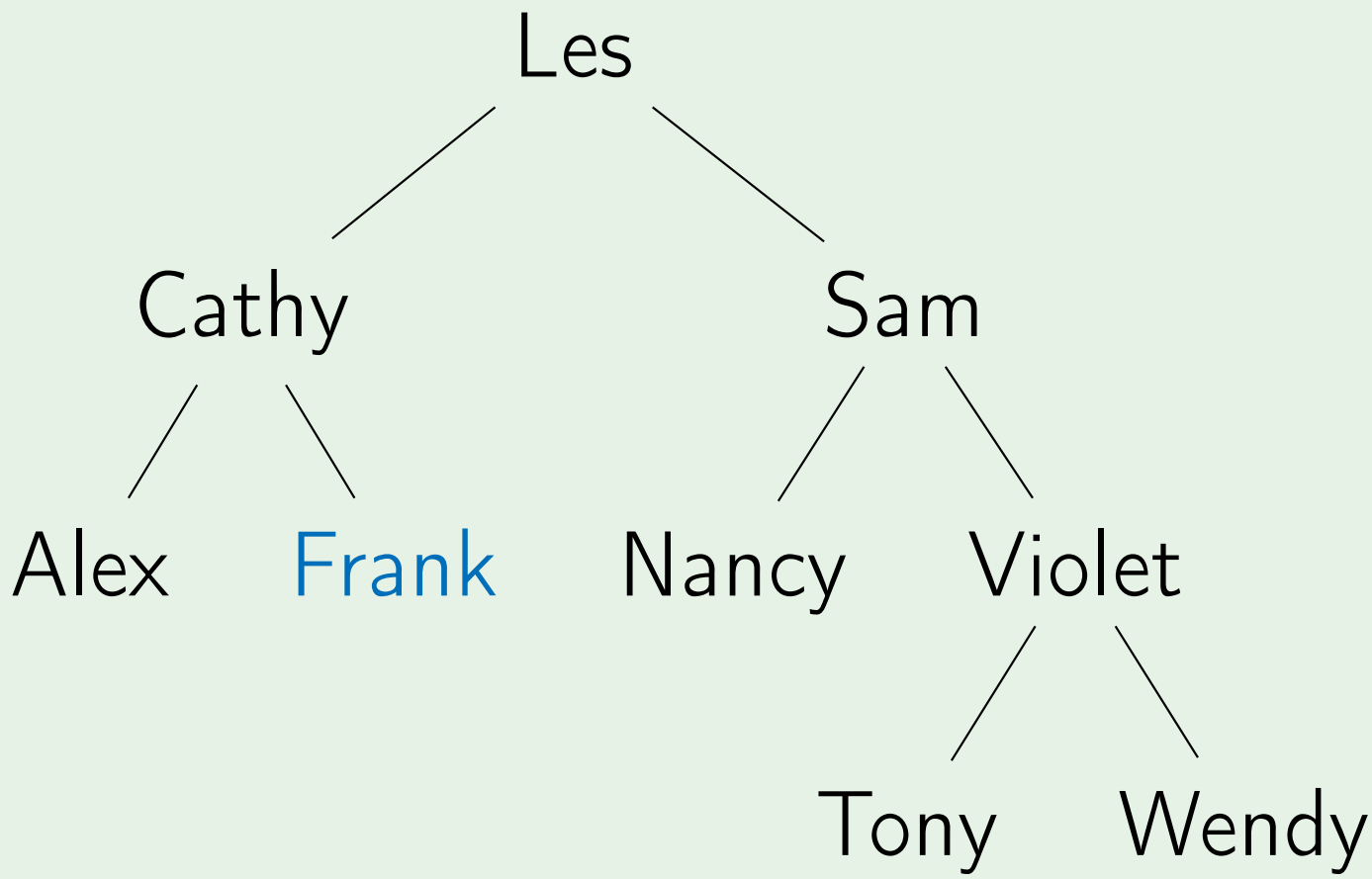
LevelTraversal



Output: Les Cathy Sam Alex

Queue: Nancy, Violet

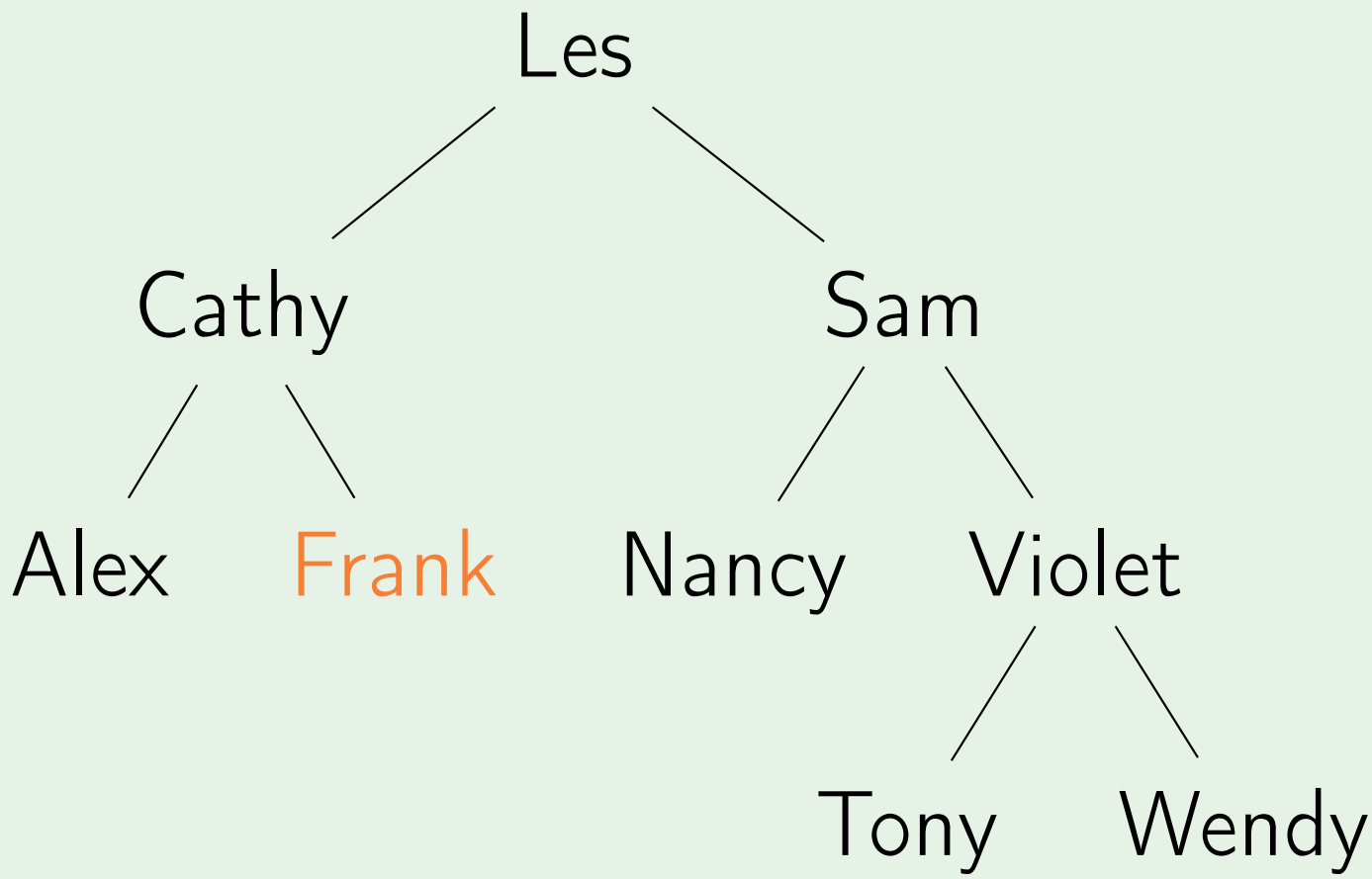
LevelTraversal



Output: Les Cathy Sam Alex Frank

Queue: Nancy, Violet

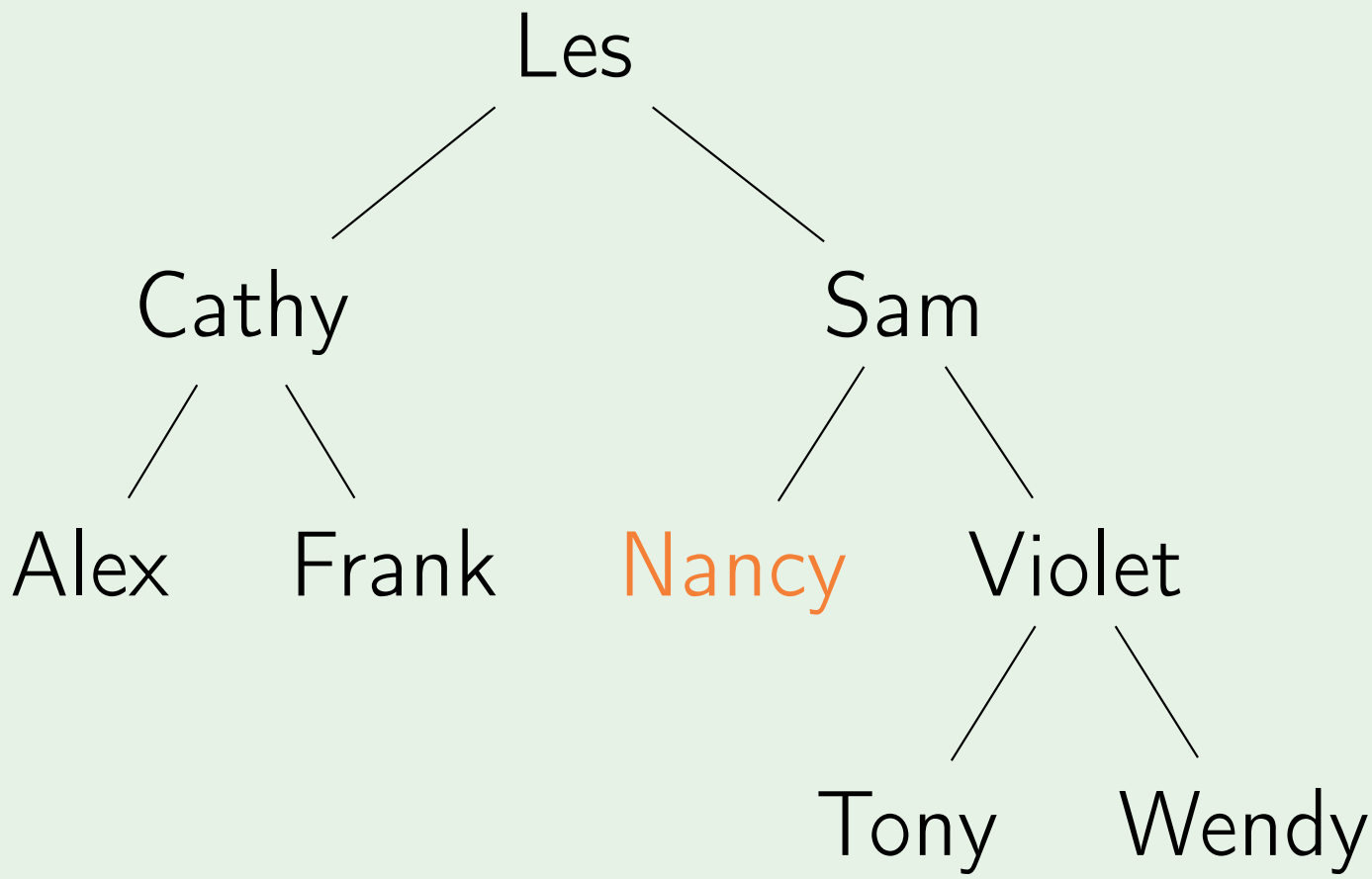
LevelTraversal



Output: Les Cathy Sam Alex Frank

Queue: Nancy, Violet

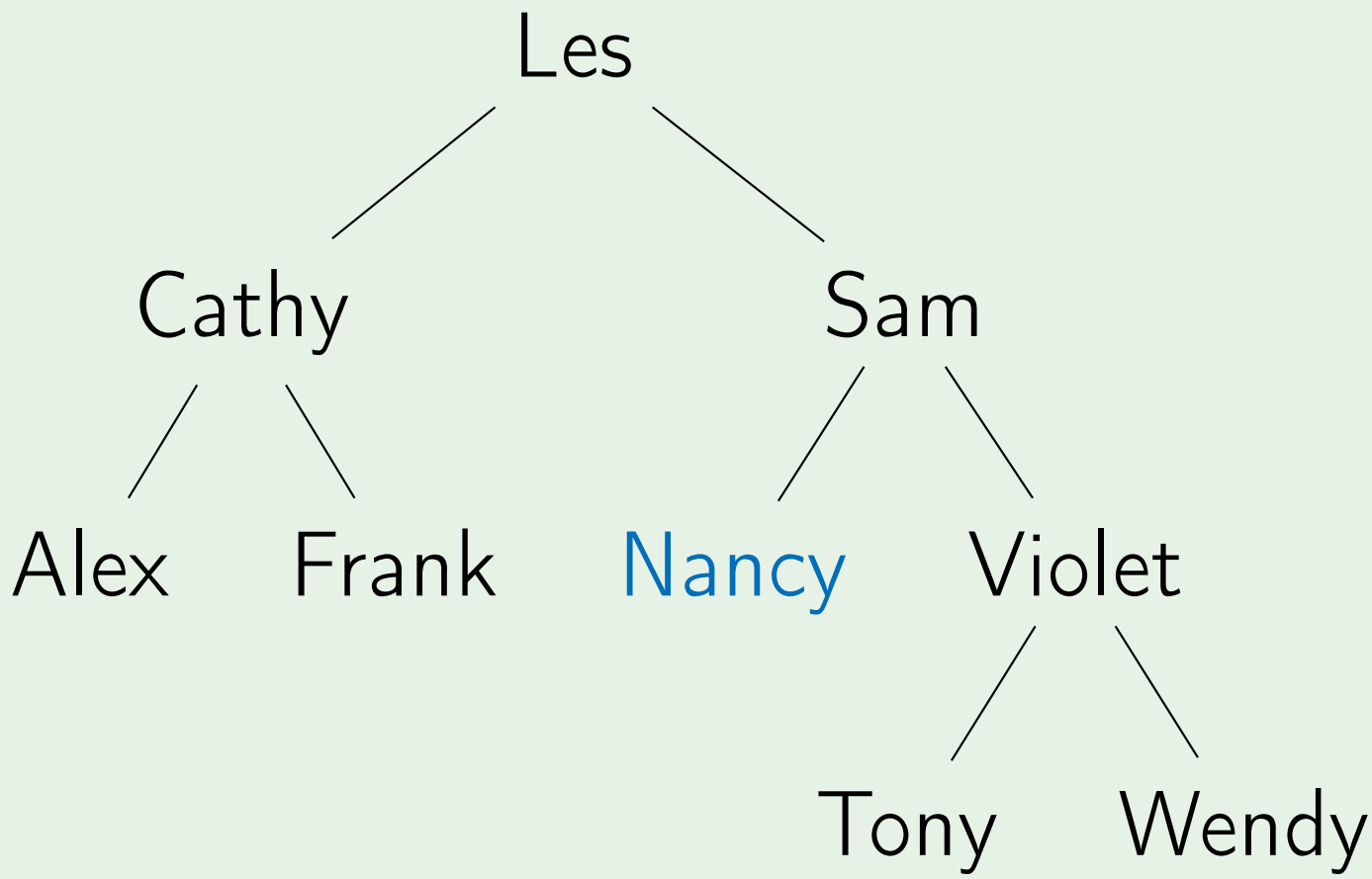
LevelTraversal



Output: Les Cathy Sam Alex Frank

Queue: Violet

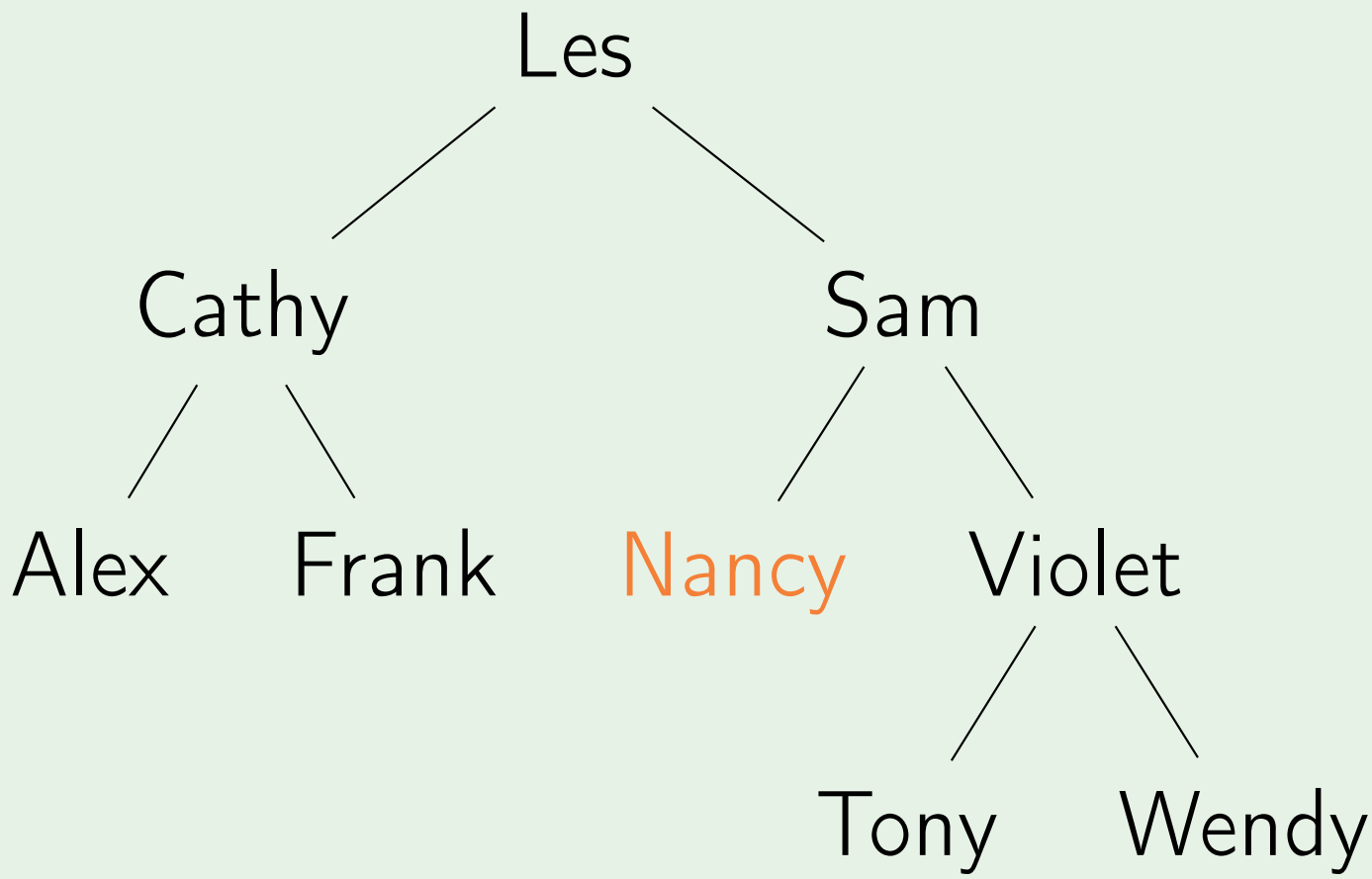
LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy

Queue: Violet

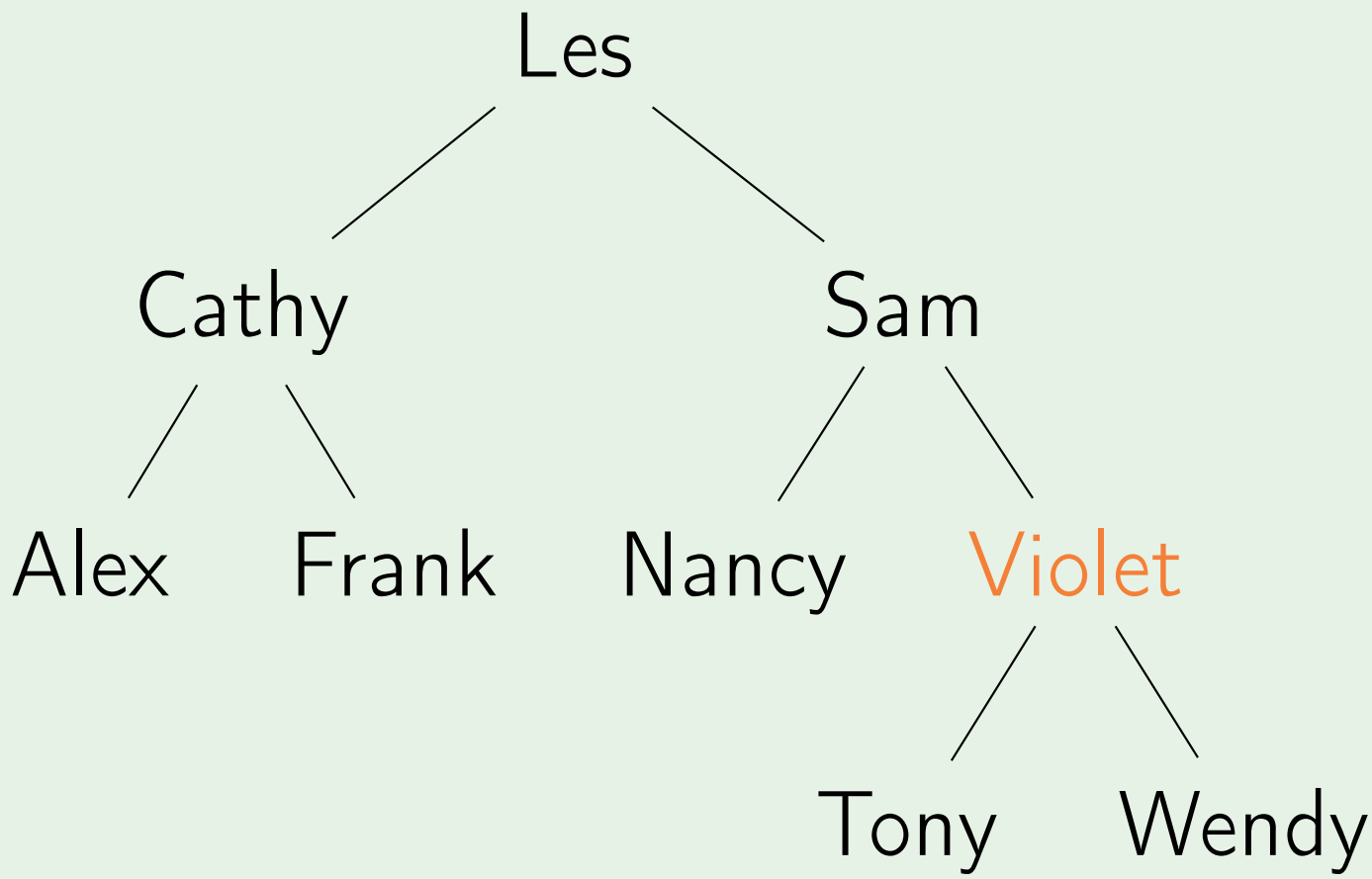
LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy

Queue: Violet

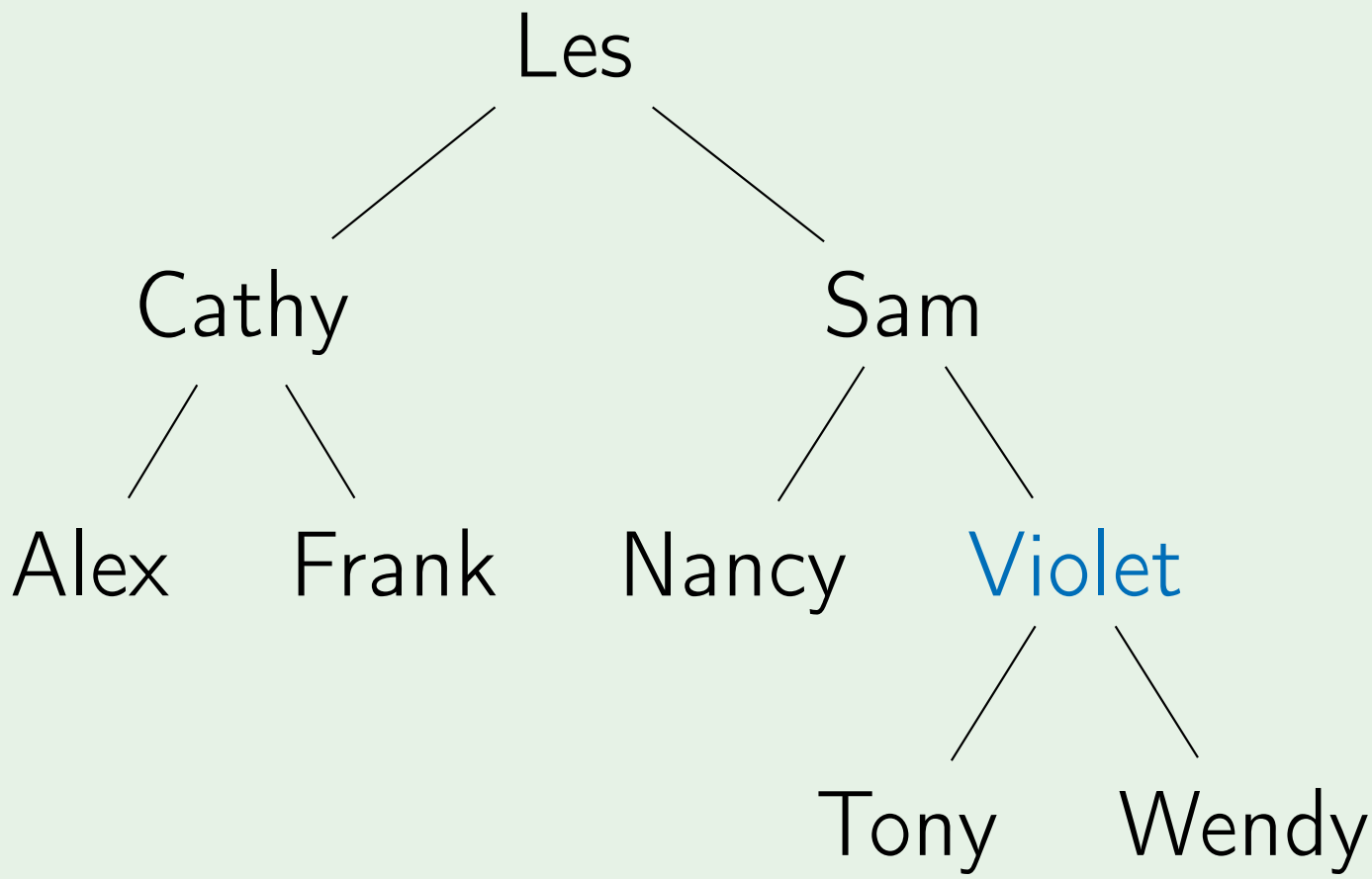
LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy

Queue:

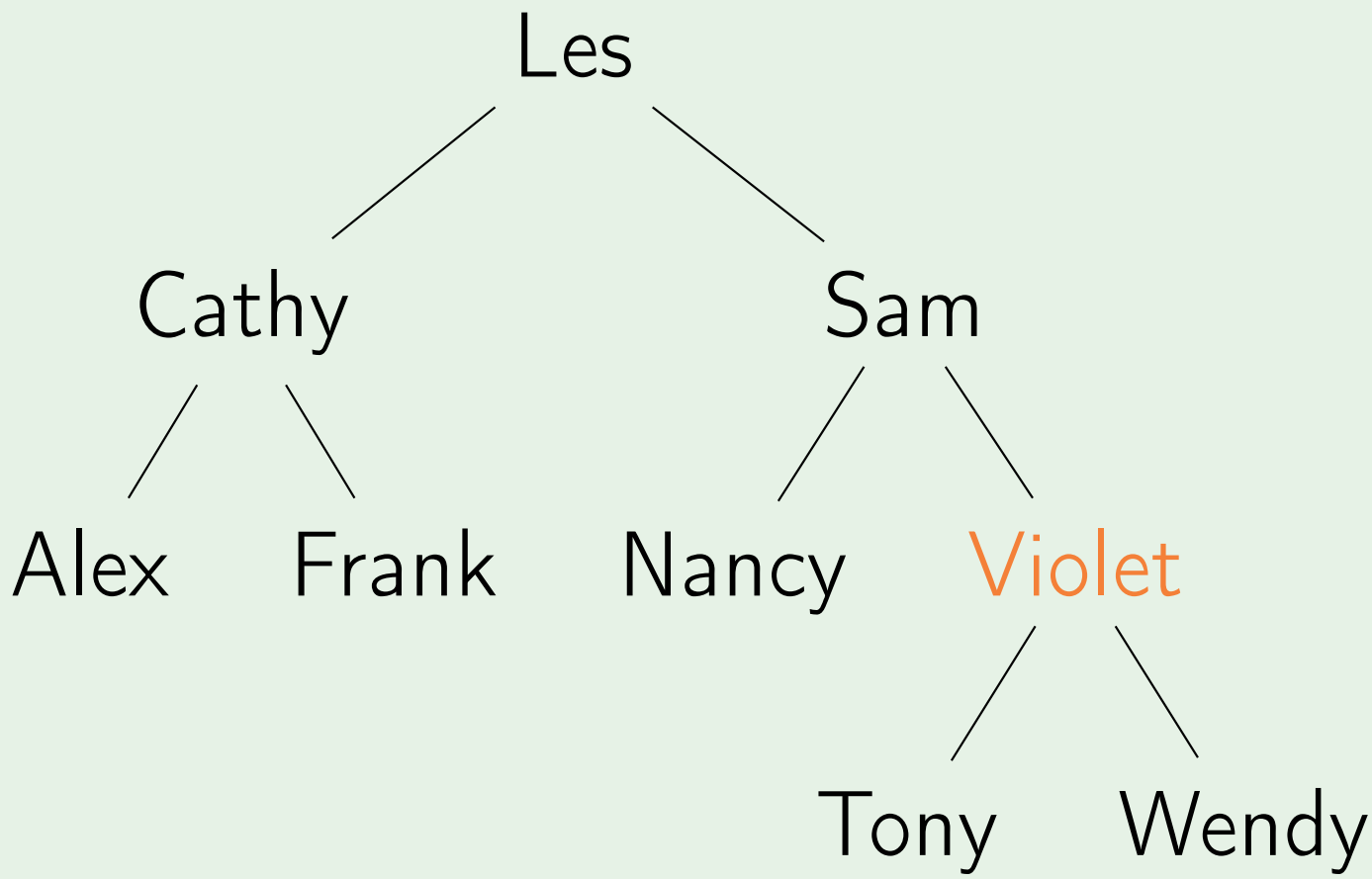
LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy Violet

Queue:

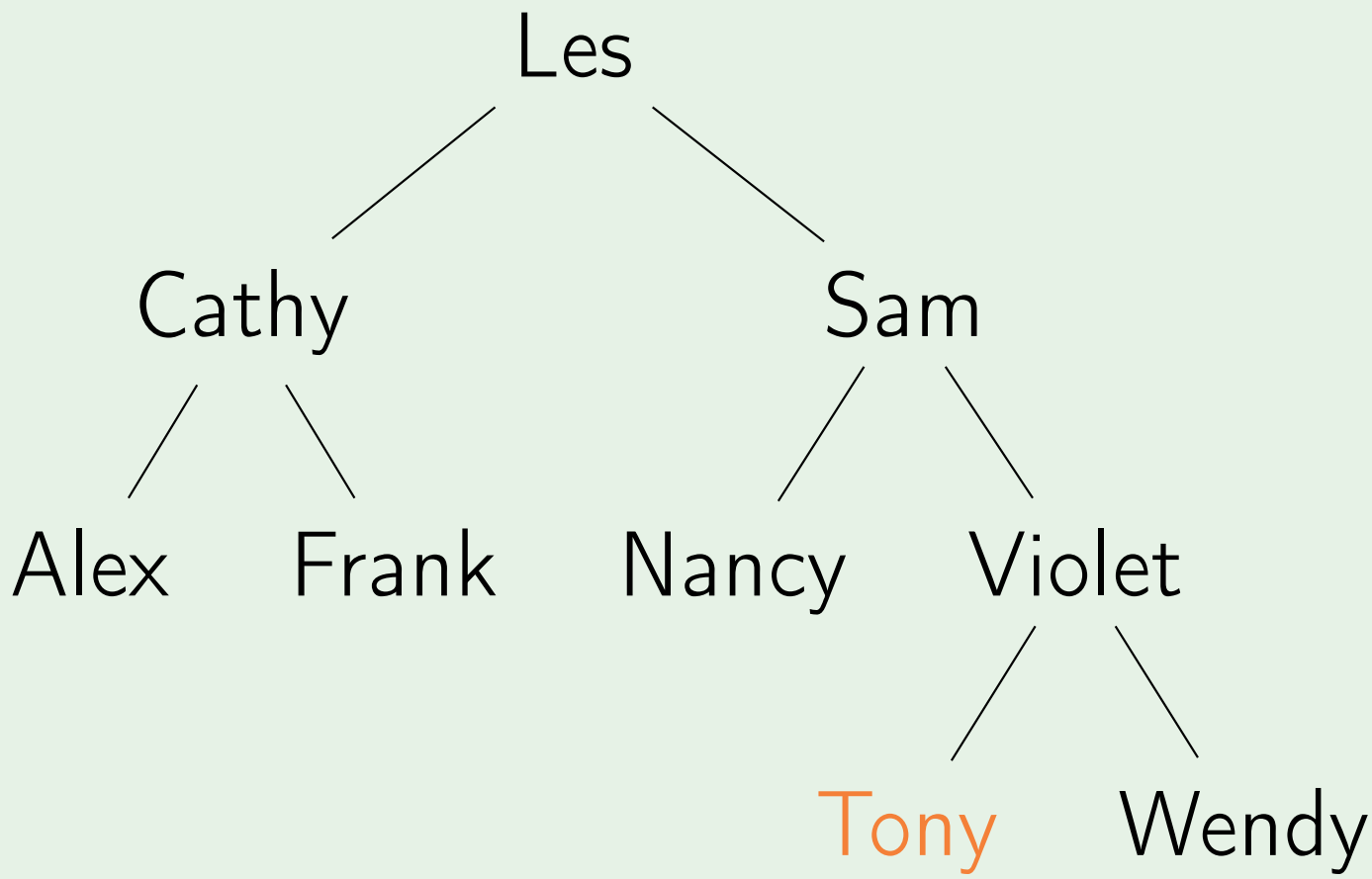
LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy Violet

Queue: Tony Wendy

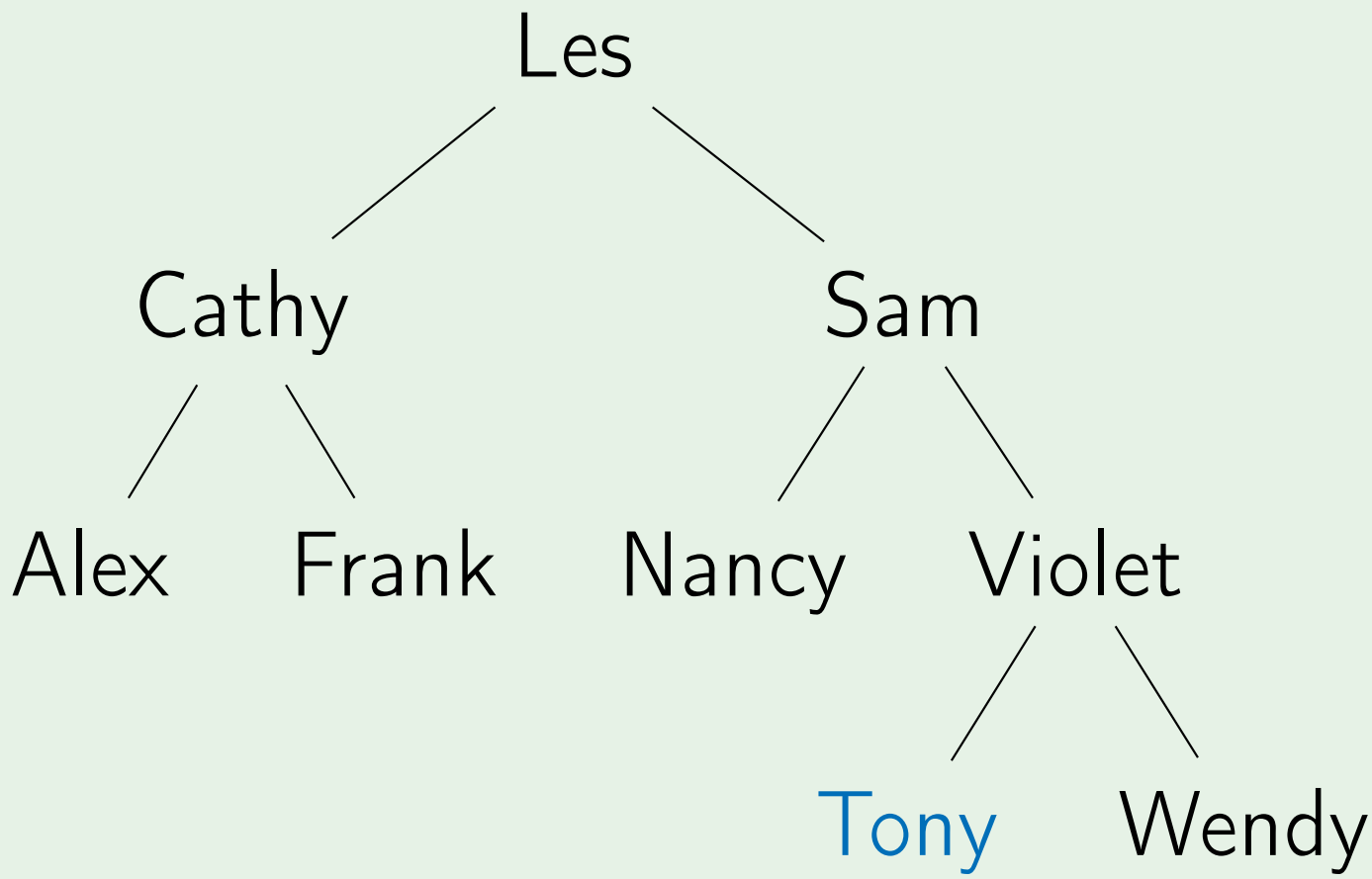
LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy Violet

Queue: Wendy

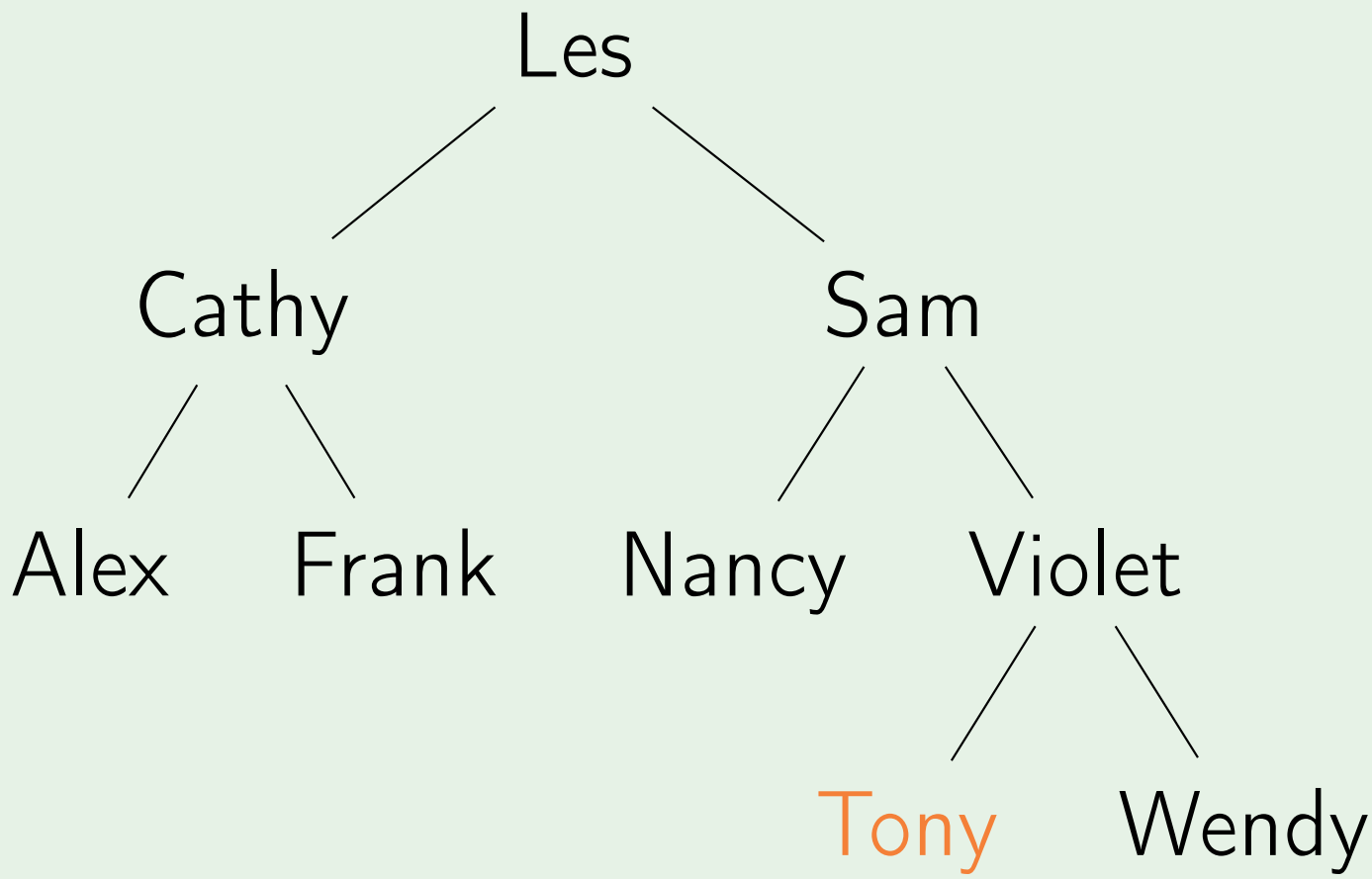
LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy Violet Tony

Queue: Wendy

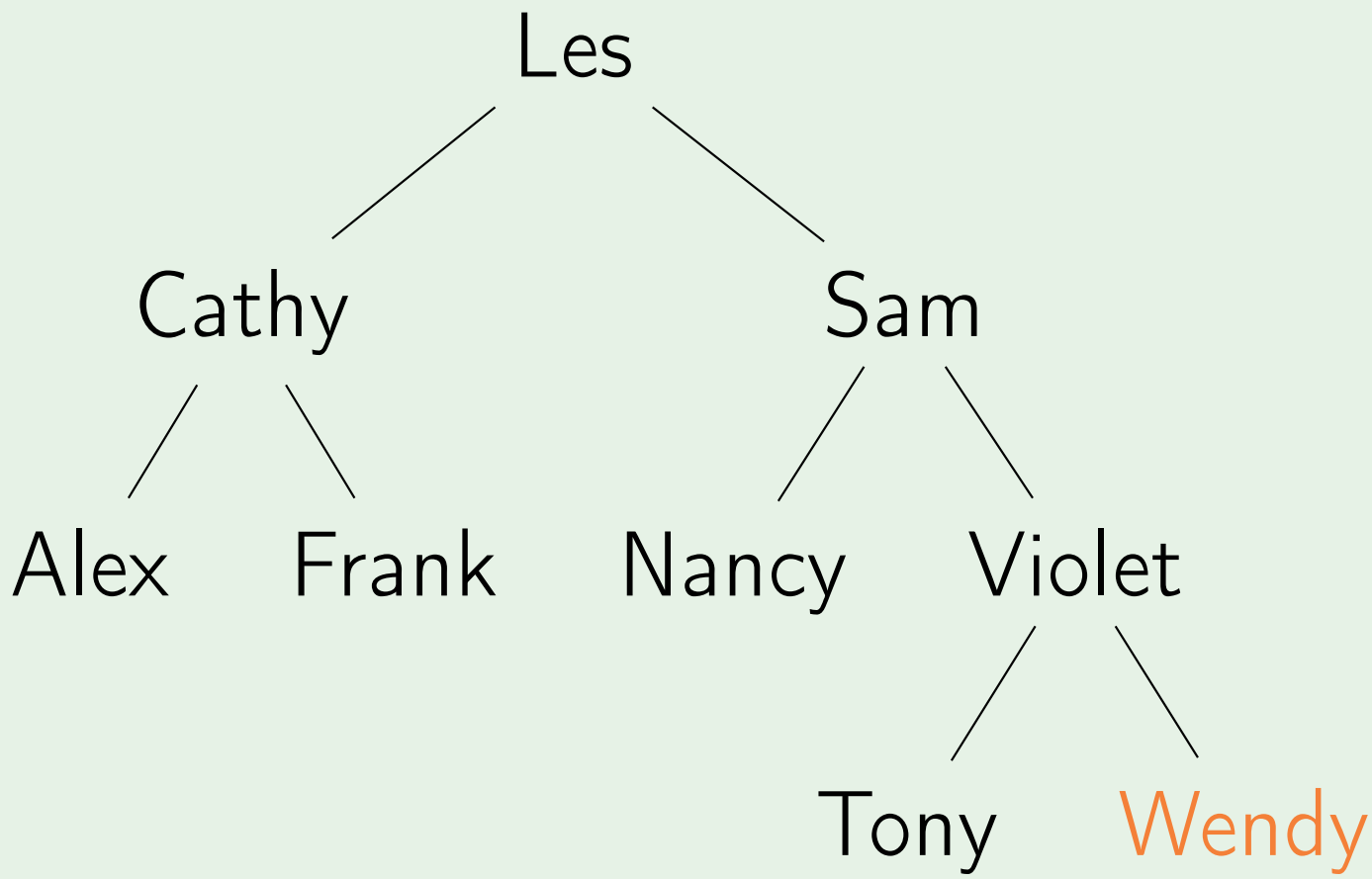
LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy Violet Tony

Queue: Wendy

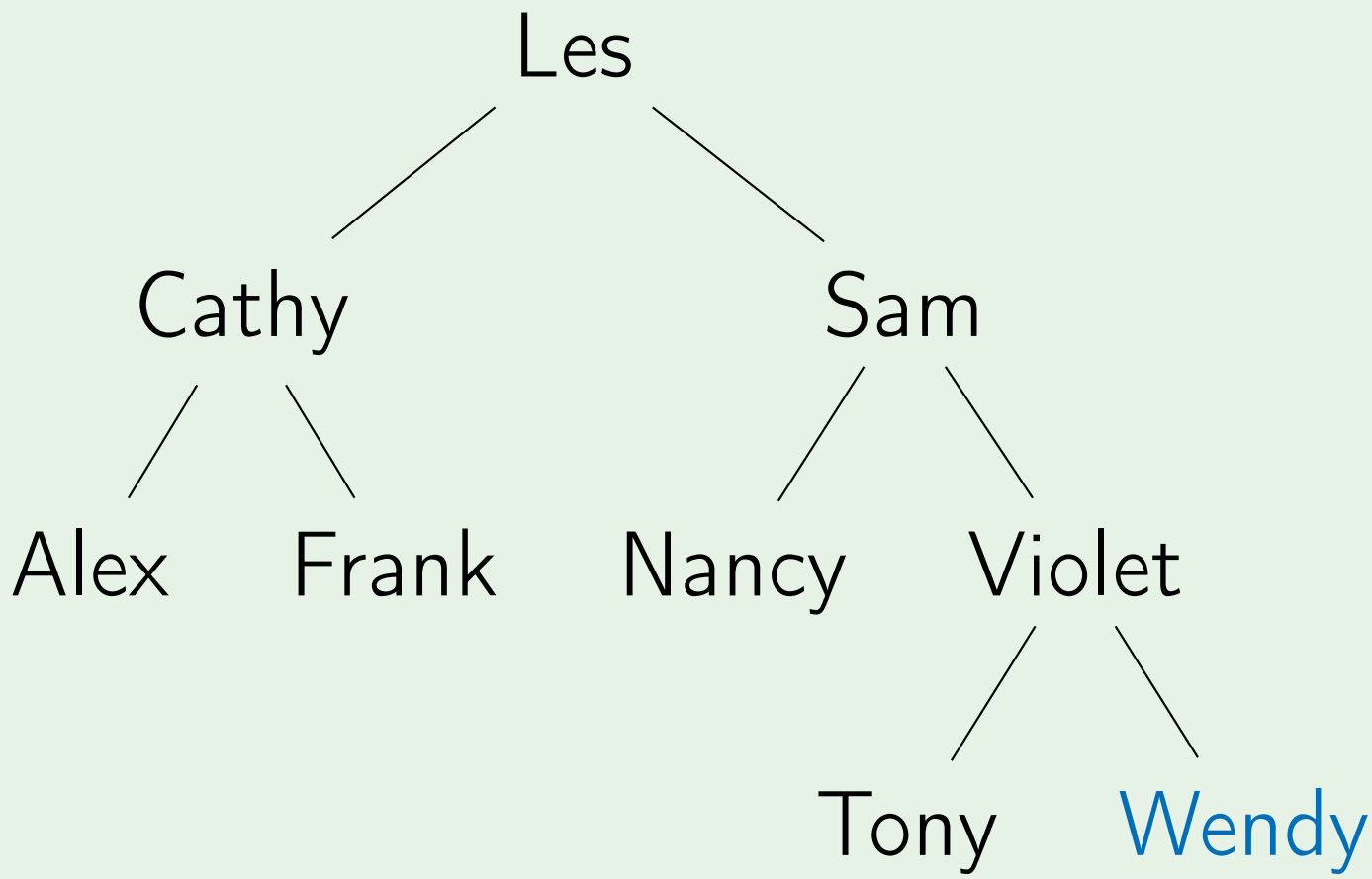
LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy Violet Tony

Queue:

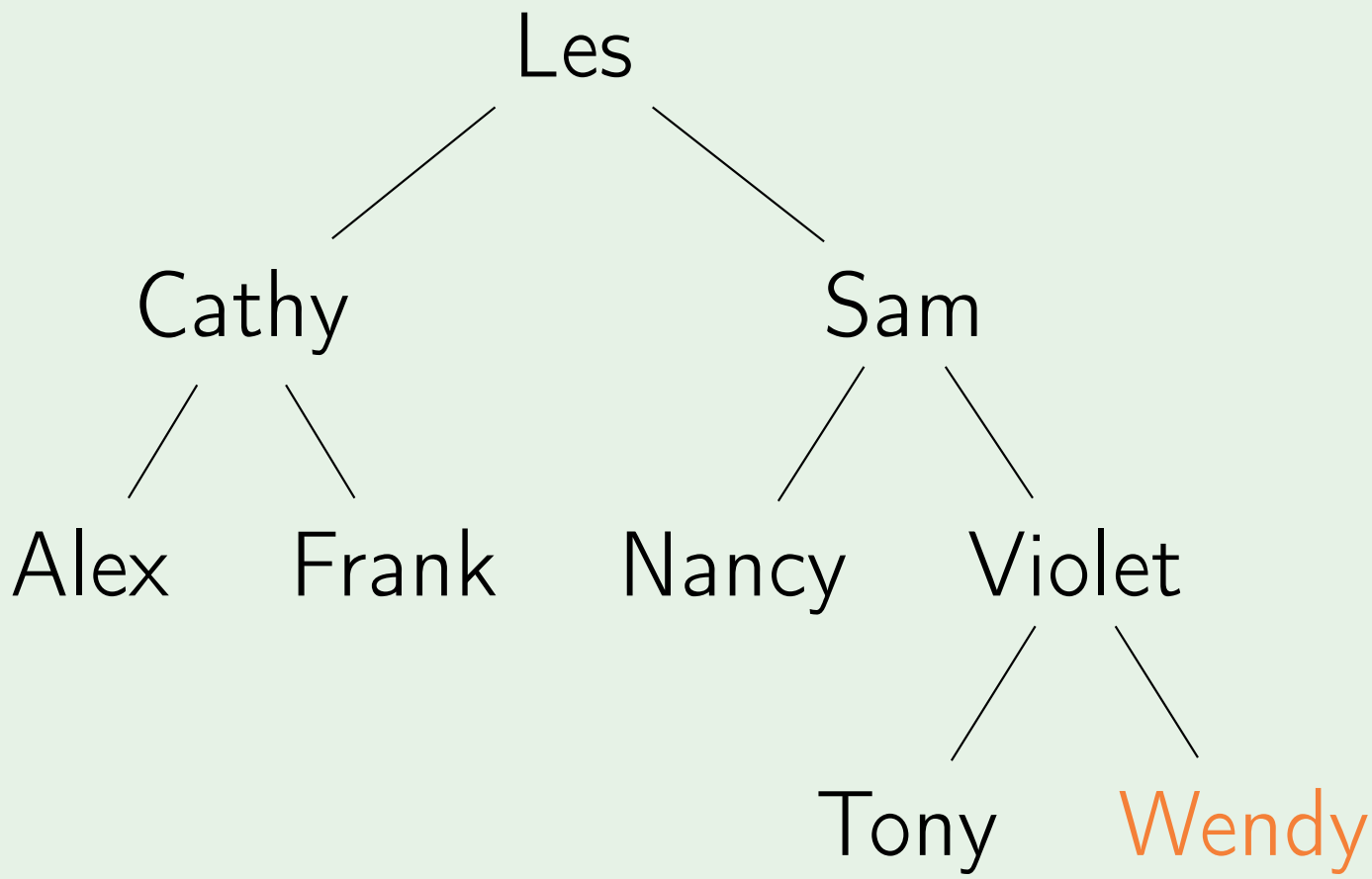
LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy Violet Tony Wendy

Queue:

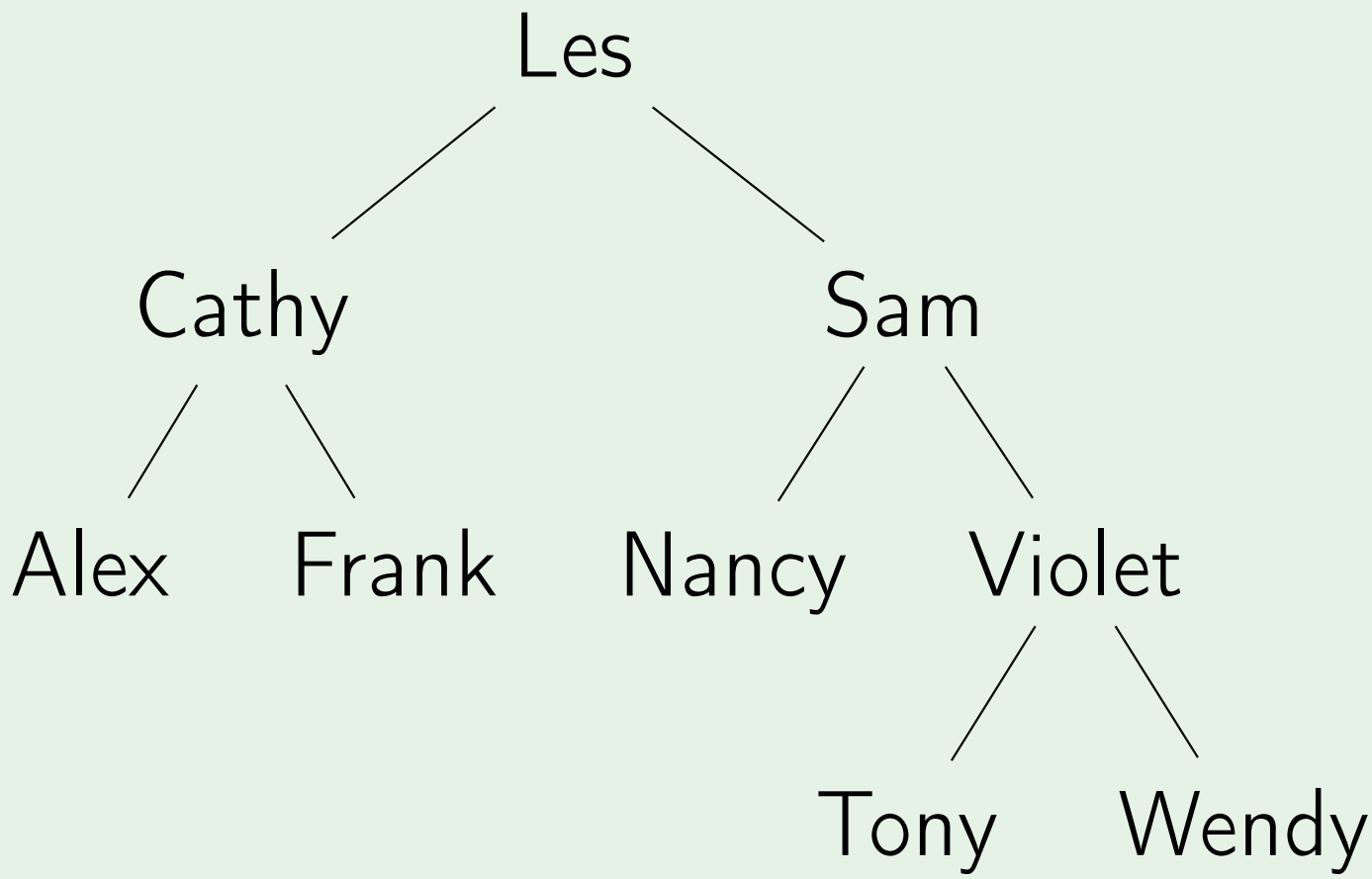
LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy Violet Tony Wendy

Queue:

LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy Violet Tony Wendy

Queue:

Summary

- Trees are used for lots of different things.
- Trees have a key and children.
- Tree walks: DFS (pre-order, in-order, post-order) and BFS.
- When working with a tree, recursive algorithms are common.
- In Computer Science, trees grow down!