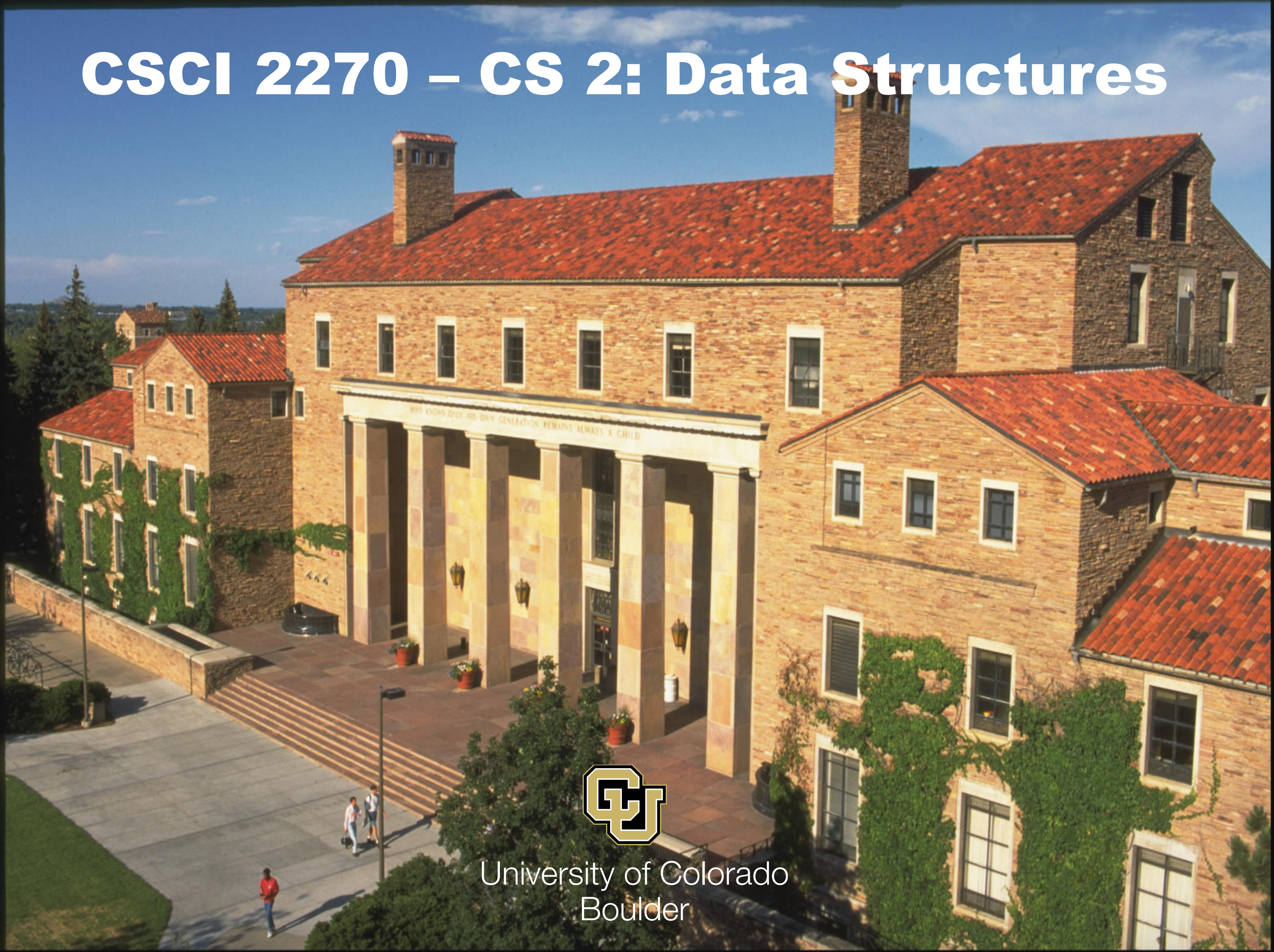
CSCI 2270 – CS 2: Data Structures

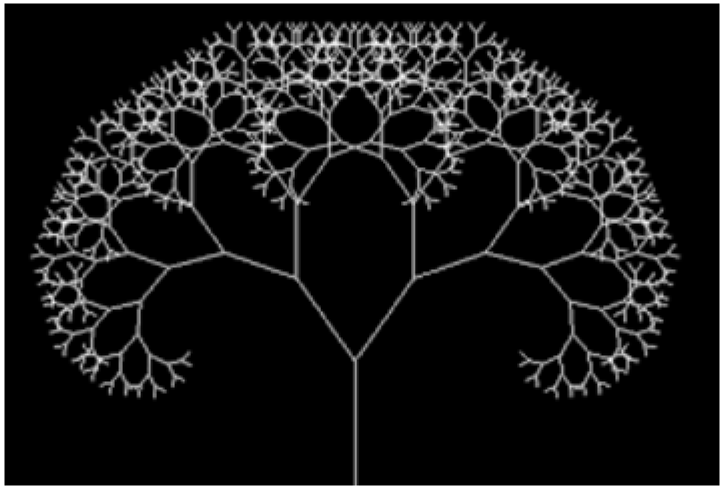


University of Colorado
Boulder

Reminders

Topics

- Trees
- Binary Trees
- Binary Search Trees
- Recursion

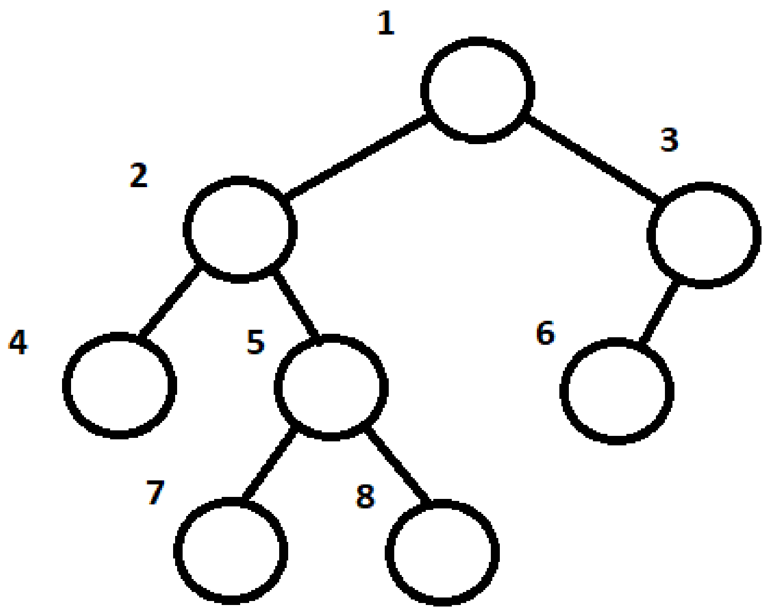


Trees

- A linear data structure is a DS in which data is arranged in a sequential manner.
- A tree is non-linear
  - Hierarchical data structure
  - Can be called a recursive data structure
- Think about an organizational chart.

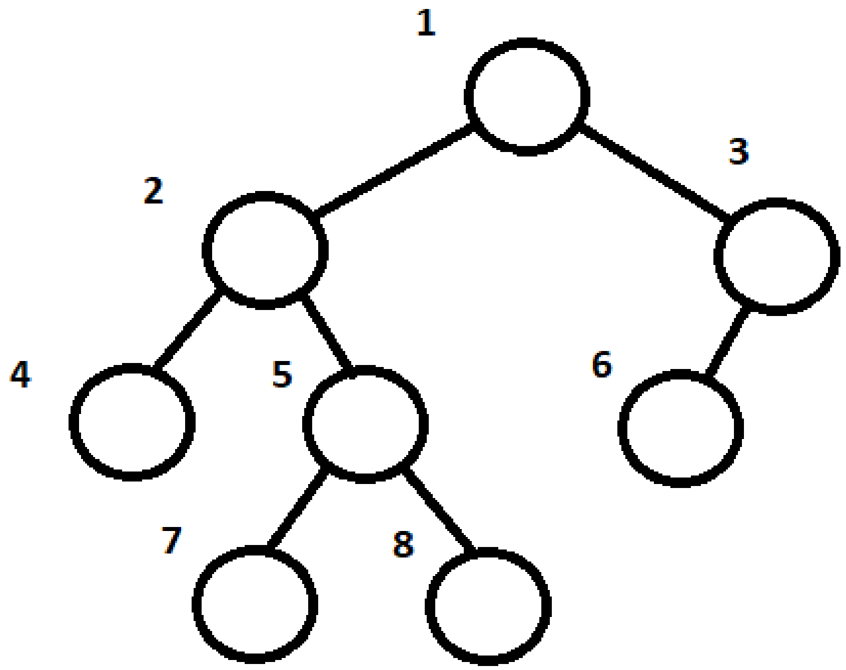
Trees

- Nodes linked together (i.e. linked data structure)
- Top node = root
- Nodes can be of any type
- Connected by edges
- Cannot have disconnected parts
- Parent
  - Not root but has child
- Children
- Sibling
- Leaf
  - Node without children



Trees

- Nodes linked together (i.e. linked data structure)
- Top node = root
- Nodes can be of any type
- Connected by edges
- Cannot have disconnected parts
- Parent
  - Not root but has child
- Children
- Sibling
  - Same parent
- Leaf
  - Node without children
- Sub-tree



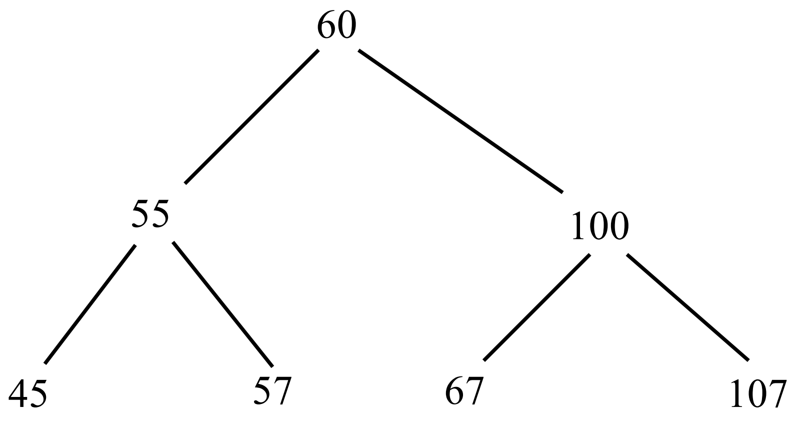
Applications of Trees

- File system in OS
- Network Routing Algorithms

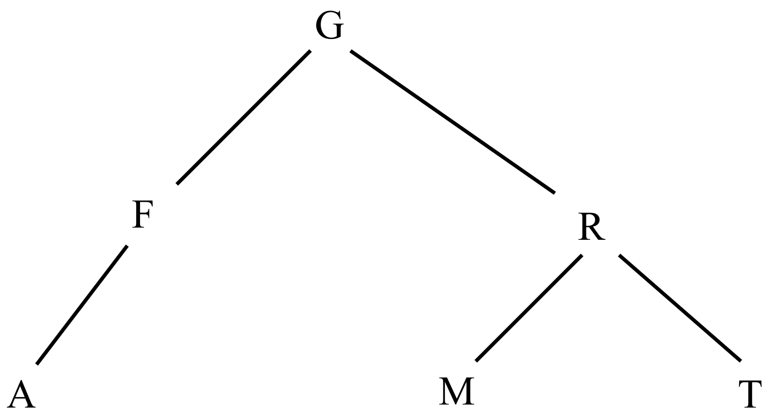
Binary Tree

- Is a classification of tree data structure.
- Linked list is one-dimensional data structure
  - Every node has a pointer to the next node
- Binary tree is made of nodes with two node pointers, called the left and right children
  - Every node has at most two children
  - Root's parent = NULL
  - Parent, left child, right child
  - Dynamically-created nodes linked using pointers or references just as we do for Linked Lists

Binary Tree



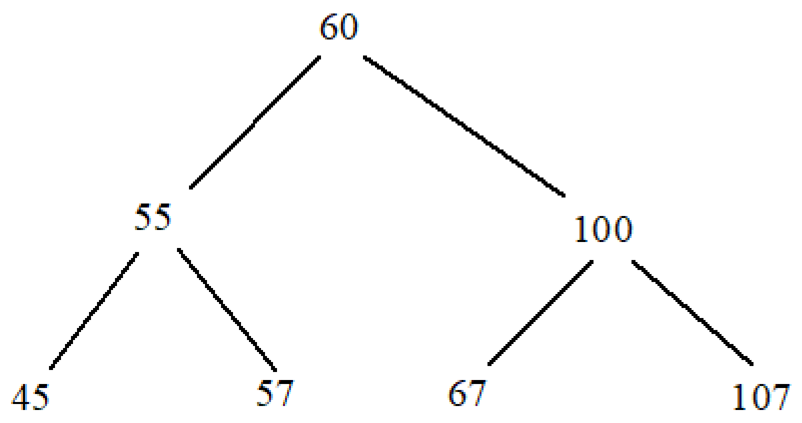
(A)



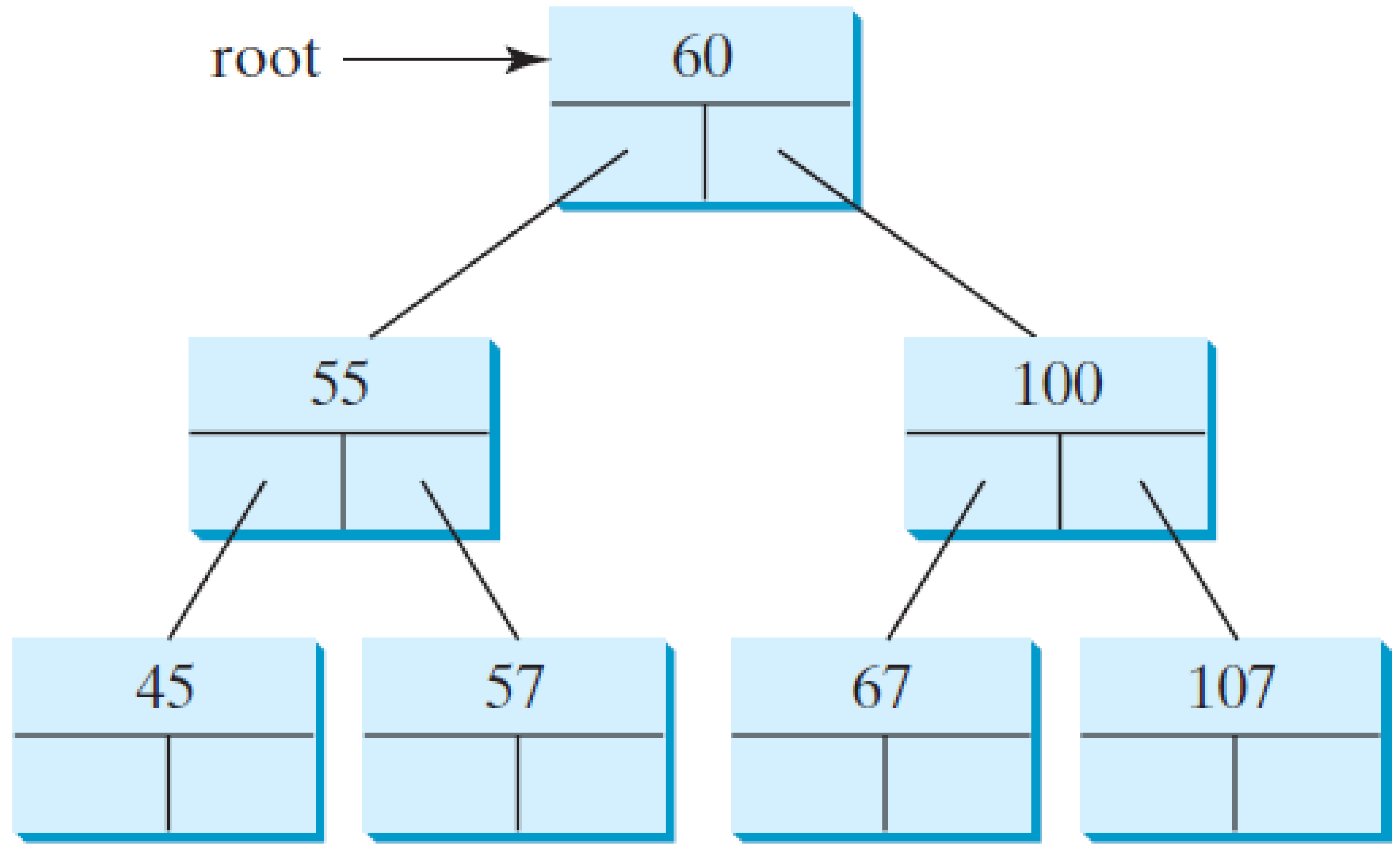
(B)

Binary Tree

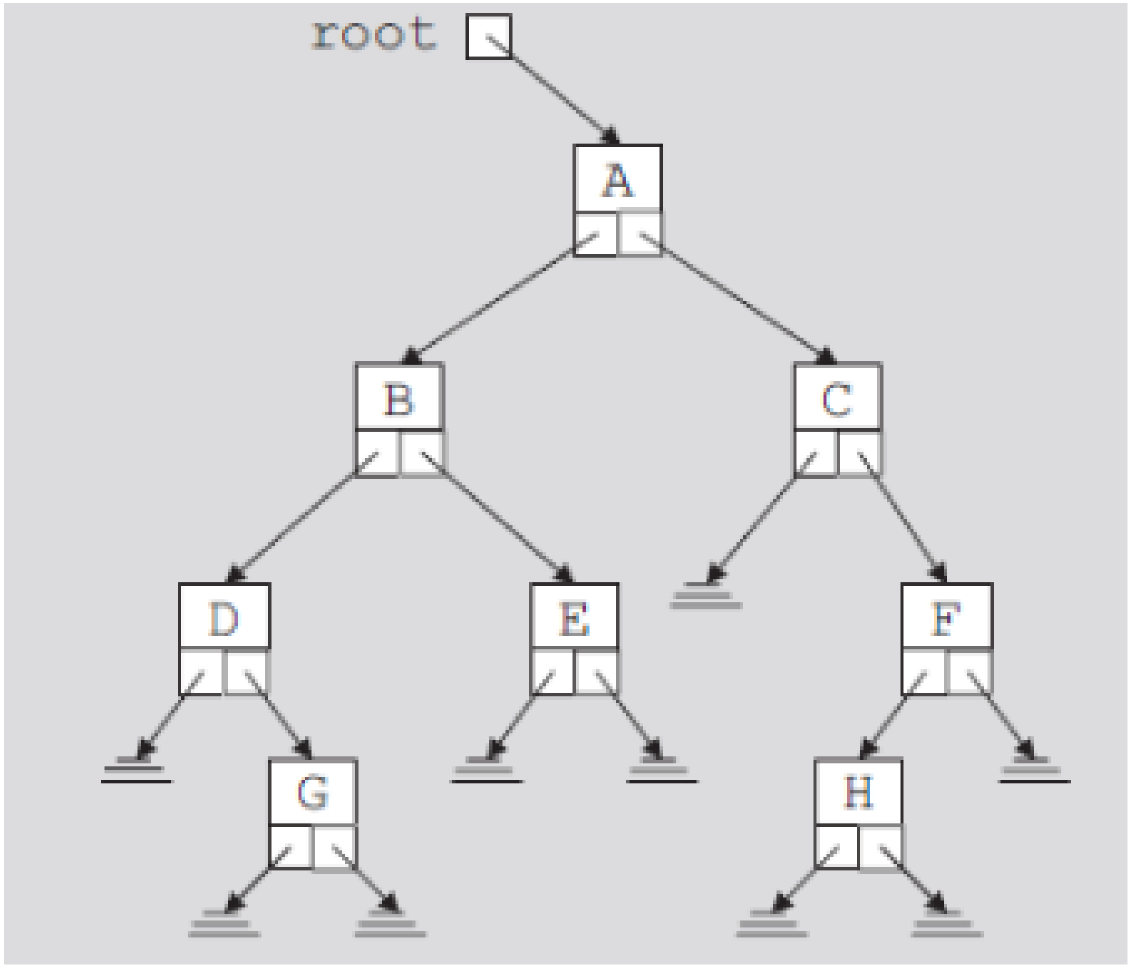
- Tree has *n* nodes and *n-1* edges
- The **length** of a path is the number of the edges in the path.
  - Length of the path from node 60 to 45 is 2
- The **depth** of a node is the length of the path from the root to the node.
  - Depth of node 60 is 0, depth of node 55 is 1, depth of node 45 is 2
- The **height** is the length of the path from the root node to its furthest leaf.
  - Height of the tree is 2
  - Height of 55 is 1
  - Leaf node heights are 0



Binary Tree



Binary Tree



Binary Tree

```
struct Node {
   int data;                    // aka key
   Node *left;
   Node *right;
}

class BST {
   private:
      Node *root;
      …
}
```

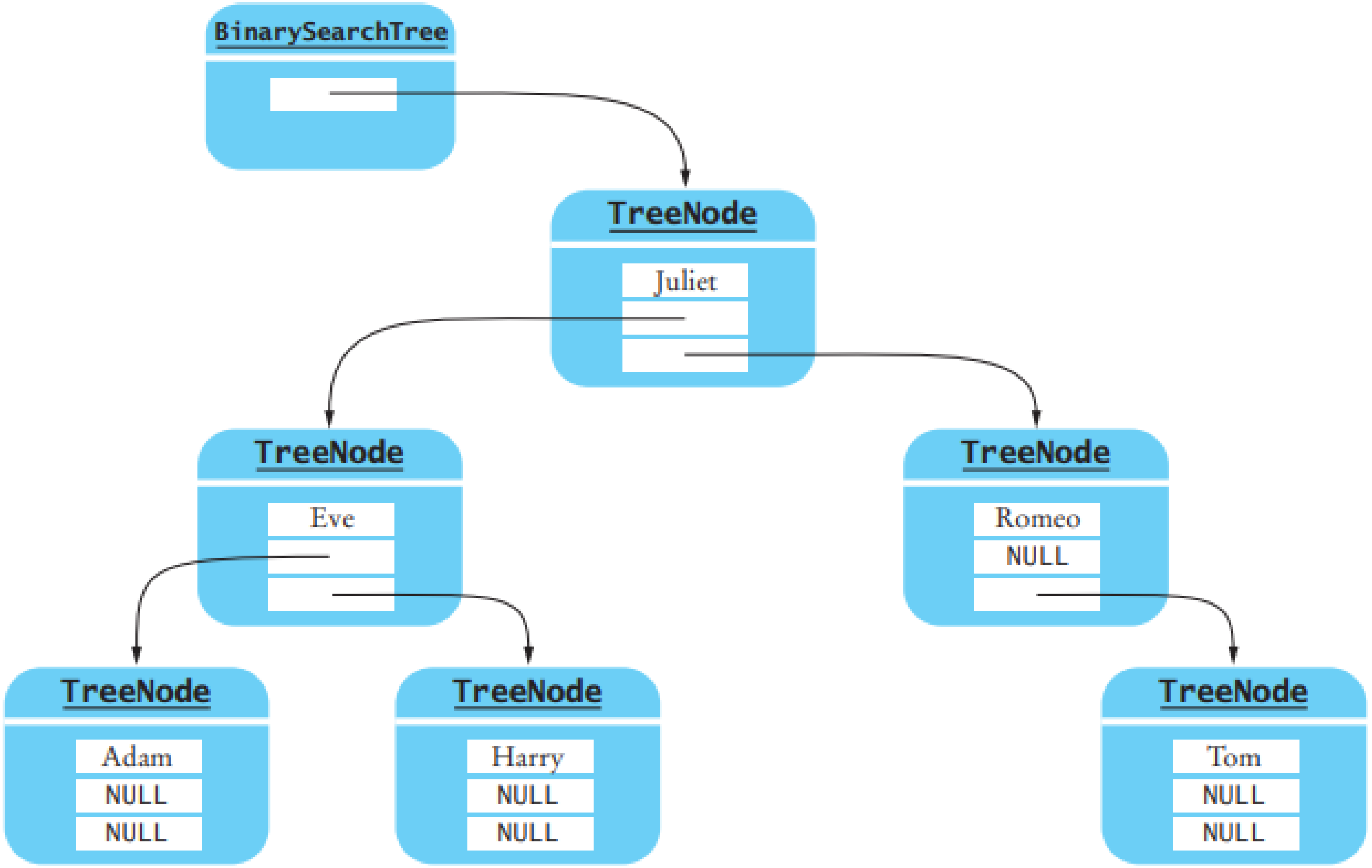
Binary Search Tree

- Special kind of tree in which data is ordered.
- Really efficient structure for storing ordered data
  - O(log n)
    - ***More on this to come***
- Every node can have 0 to 2 children.
- The data values of all descendants to the left of any node are less than the data value stored in that node, and all descendants to the right have greater data values.
  - Thus, by definition BST's cannot have distinct keys (i.e. can't have more than one of the same value)

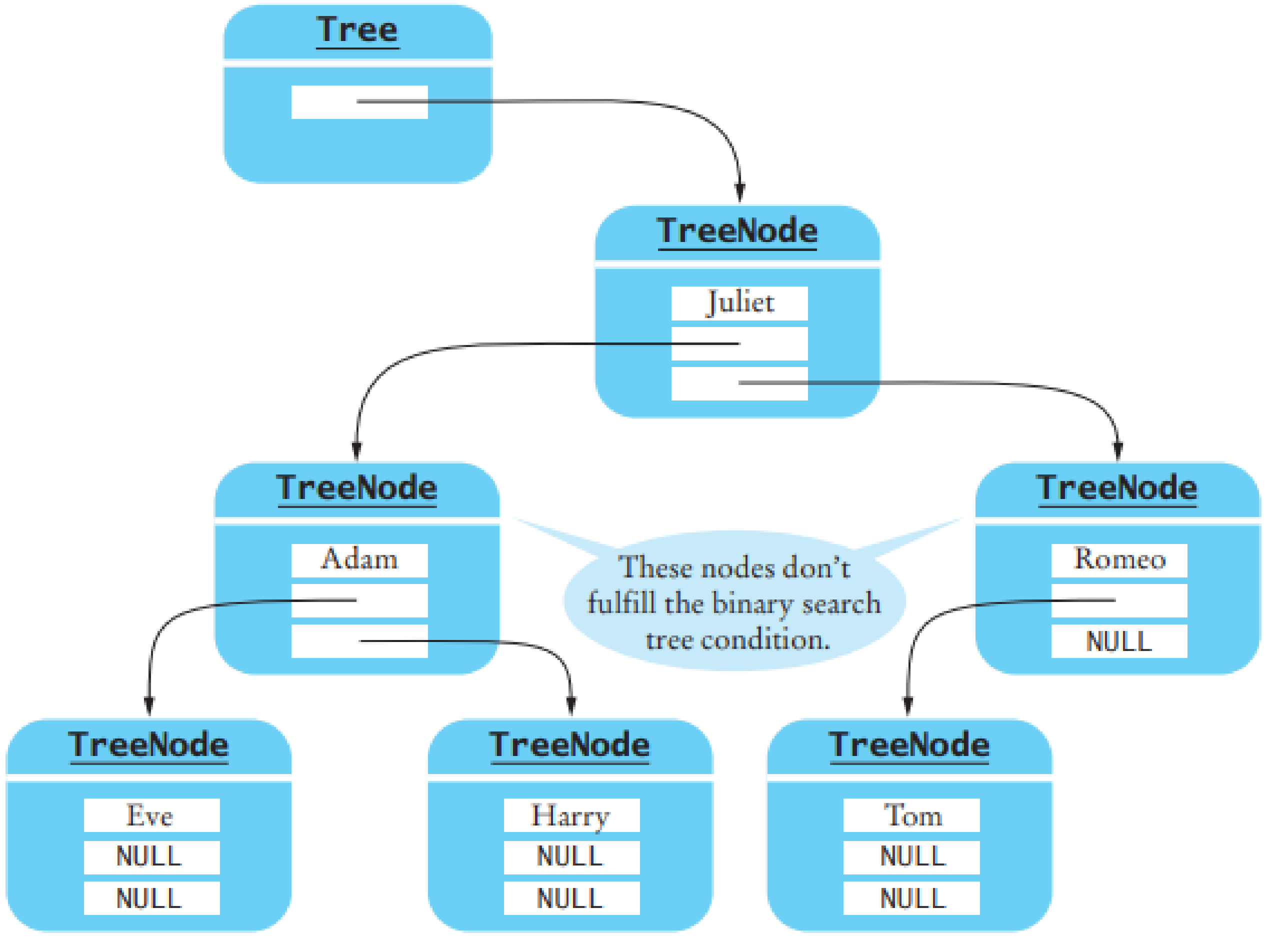
Binary Search Tree

- One more time…
  - Nodes in the left subtree have key values less than the node value
  - Nodes in the right subtree have key values greater than or equal to the node value

Binary Search Tree



Binary Search Tree



Binary Search Tree - ADT

- **Operations typically performed on a binary tree**
  - Determine if binary tree is empty
  - Search binary tree for a particular item
  - Insert an item in the binary tree
  - Delete an item from the binary tree
  - Find the height of the binary tree
  - Find the number of nodes in the binary tree
  - Find the number of leaves in the binary tree
  - Traverse the binary tree
  - Copy the binary tree

Recursion

- When a program calls another instance of itself
- Recursive definition
  - A definition in which something is defined in terms of a smaller version of itself
- Recursive algorithm
  - Finds problem solution by reducing problem to smaller versions of itself
- Recursive function
  - Function that calls itself
- Base case
  - Case for which the solution is obtained directly
- General case
  - Case for which the solution is obtained indirectly using recursion
- For a proper recursive algorithm, the algorithm needs to **eventually reach a base case**, so it can terminate.

Recursion

- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

Recursive Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4)

Recursive Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

Recursive Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

Recursive Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

```
factorial(4) = 4 * factorial(3)
             = 4 * 3 * factorial(2)
             = 4 * 3 * (2 * factorial(1))
```

Recursive Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

= 4 * 3 * (2 * (1 * factorial(0)))

Recursive Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

= 4 * 3 * (2 * (1 * factorial(0)))

= 4 * 3 * (2 * (1 * 1)))

Recursive Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

```
factorial(4) = 4 * factorial(3)
             = 4 * 3 * factorial(2)
             = 4 * 3 * (2 * factorial(1))
             = 4 * 3 * ( 2 * (1 * factorial(0)))
             = 4 * 3 * ( 2 * ( 1 * 1)))
             = 4 * 3 * ( 2 * 1)
```

Recursive Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

```
factorial(4) = 4 * factorial(3)
             = 4 * 3 * factorial(2)
             = 4 * 3 * (2 * factorial(1))
             = 4 * 3 * ( 2 * (1 * factorial(0)))
             = 4 * 3 * ( 2 * ( 1 * 1)))
             = 4 * 3 * ( 2 * 1)
             = 4 * 3 * 2
```

Recursive Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

```
factorial(4) = 4 * factorial(3)
             = 4 * (3 * factorial(2))
             = 4 * (3 * (2 * factorial(1)))
             = 4 * (3 * ( 2 * (1 * factorial(0))))
             = 4 * (3 * ( 2 * ( 1 * 1))))
             = 4 * (3 * ( 2 * 1))
             = 4 * (3 * 2)
             = 4 * (6)
```

Recursive Factorial

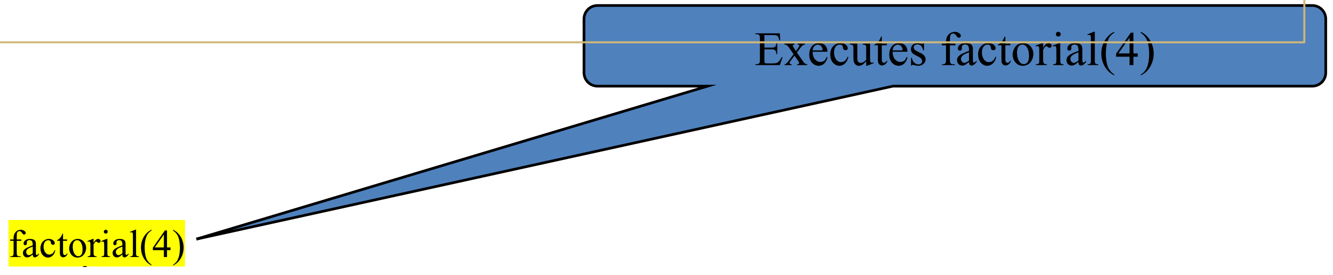
factorial(0) = 1;

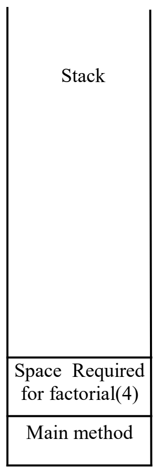
factorial(n) = n*factorial(n-1);

```
factorial(4) = 4 * factorial(3)
             = 4 * (3 * factorial(2))
             = 4 * (3 * (2 * factorial(1)))
             = 4 * (3 * ( 2 * (1 * factorial(0))))
             = 4 * (3 * ( 2 * ( 1 * 1))))
             = 4 * (3 * ( 2 * 1))
             = 4 * (3 * 2)
             = 4 * (6)
             = 24
```

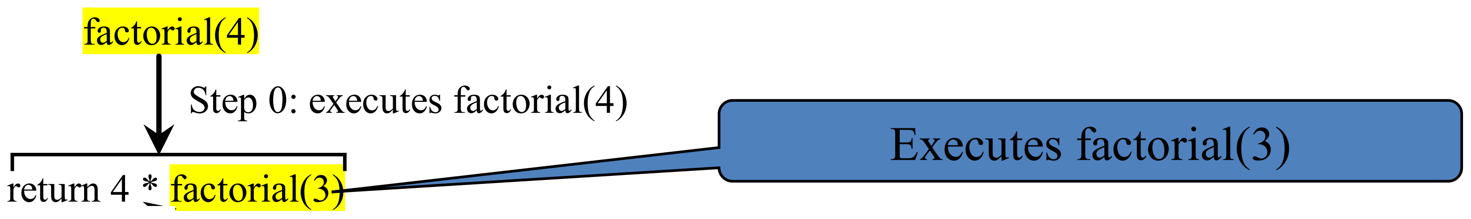
animation

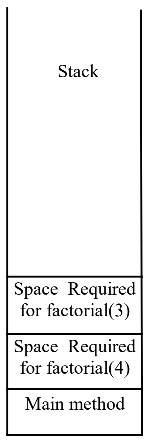
Recursive Factorial



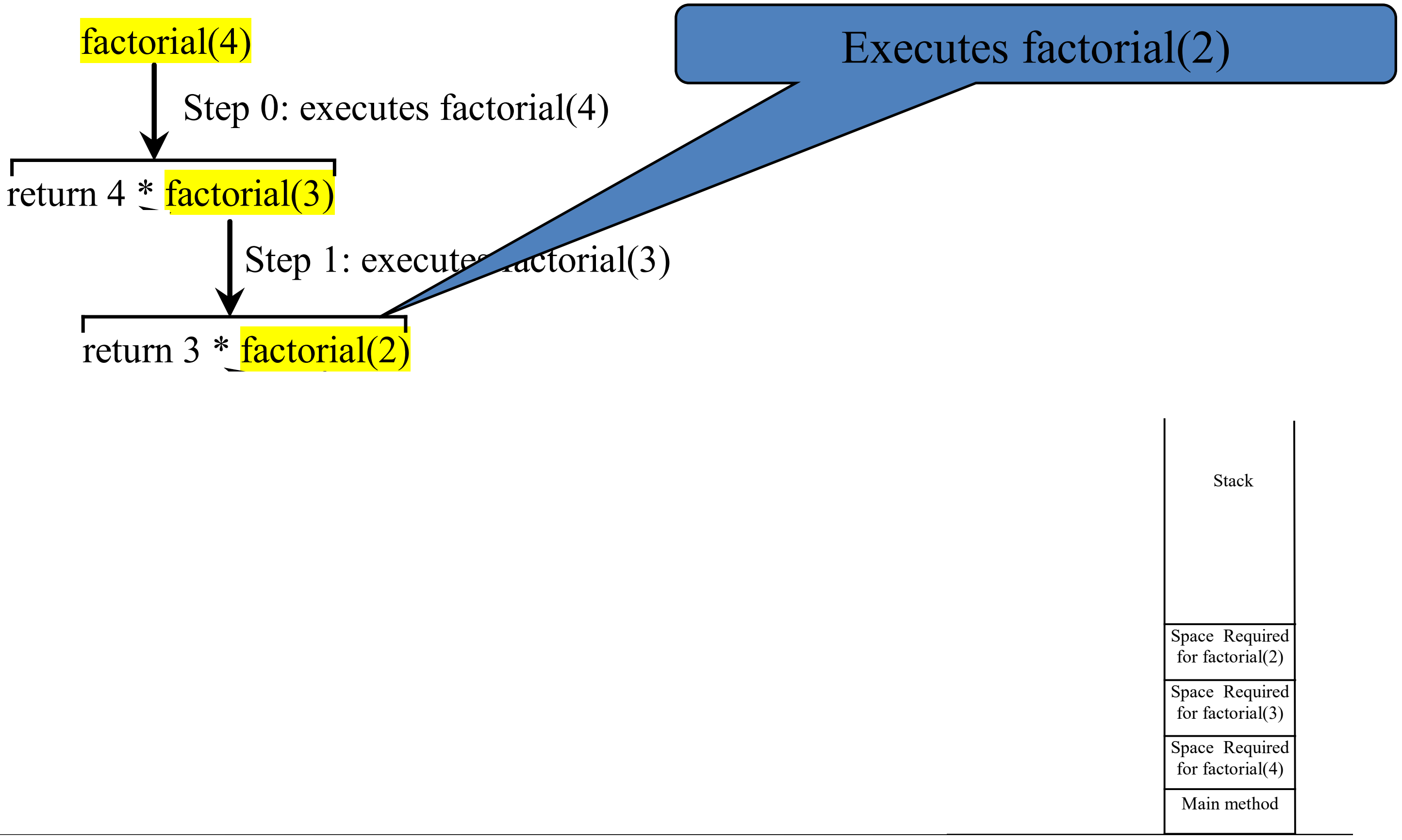


Recursive Factorial

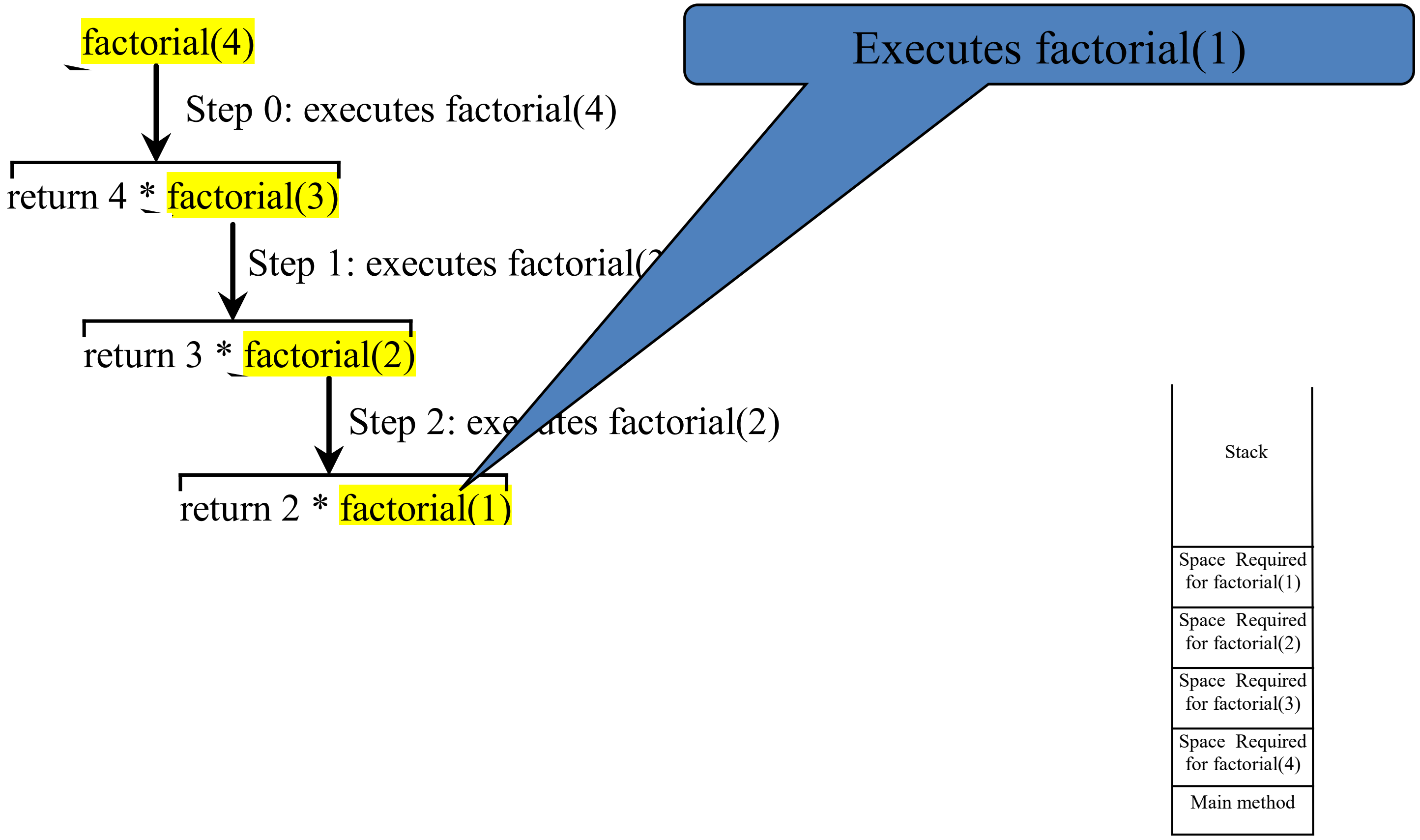



Stack
Space Required for factorial(3)
Space Required for factorial(4)
Main method

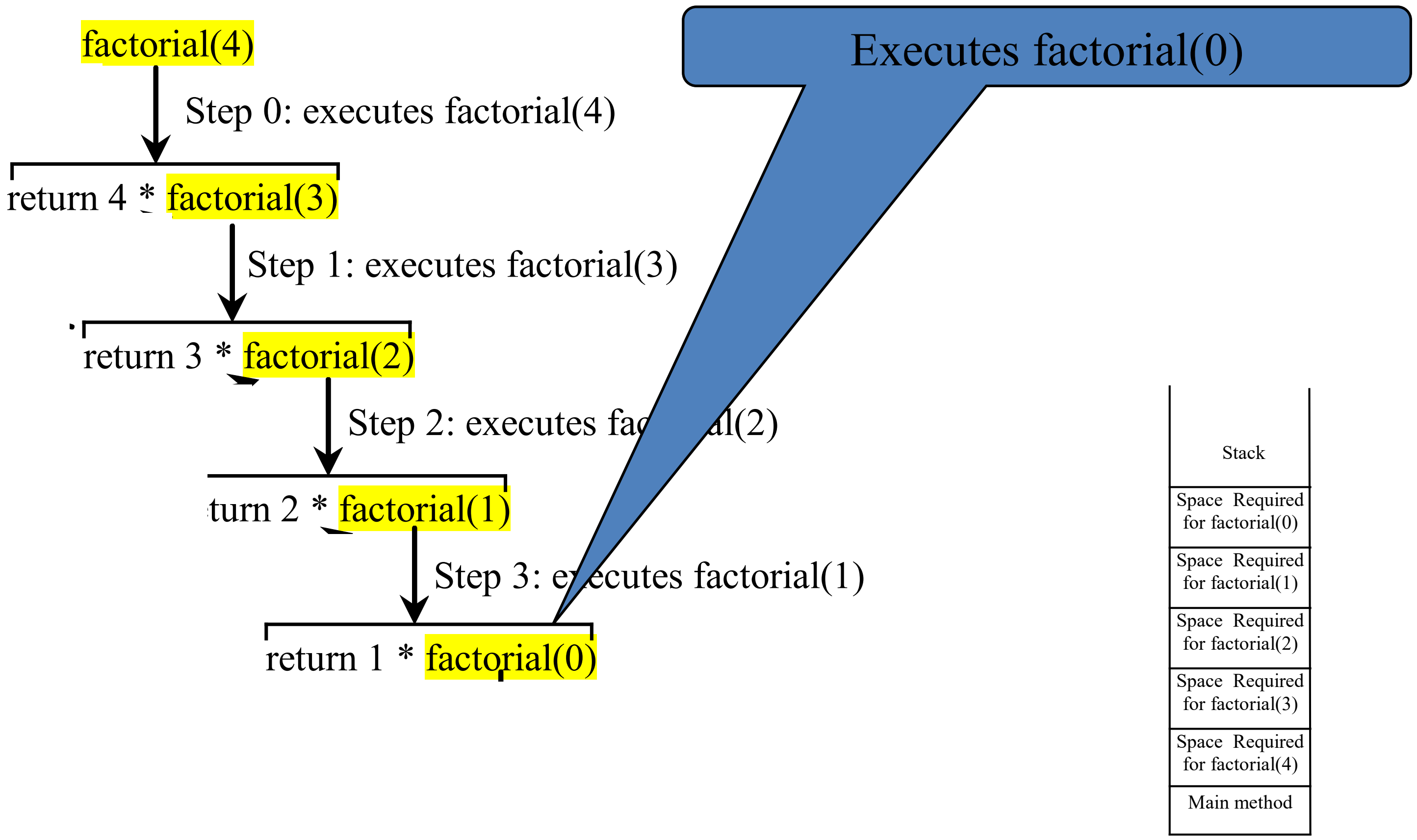
Recursive Factorial


factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Executes factorial(2)
Stack
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

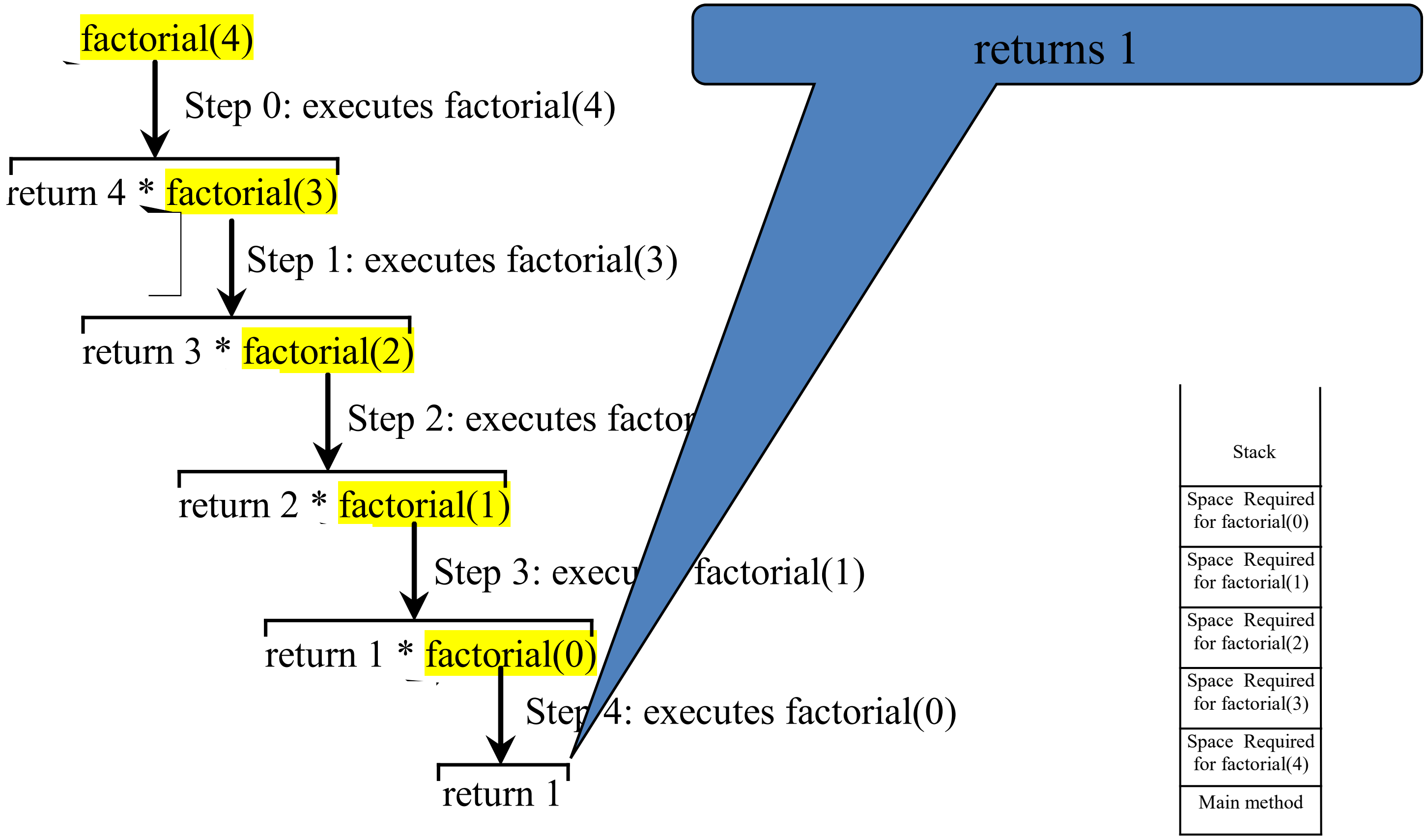
Recursive Factorial


factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factorial(2)
return 2 * factorial(1)
Executes factorial(1)
Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

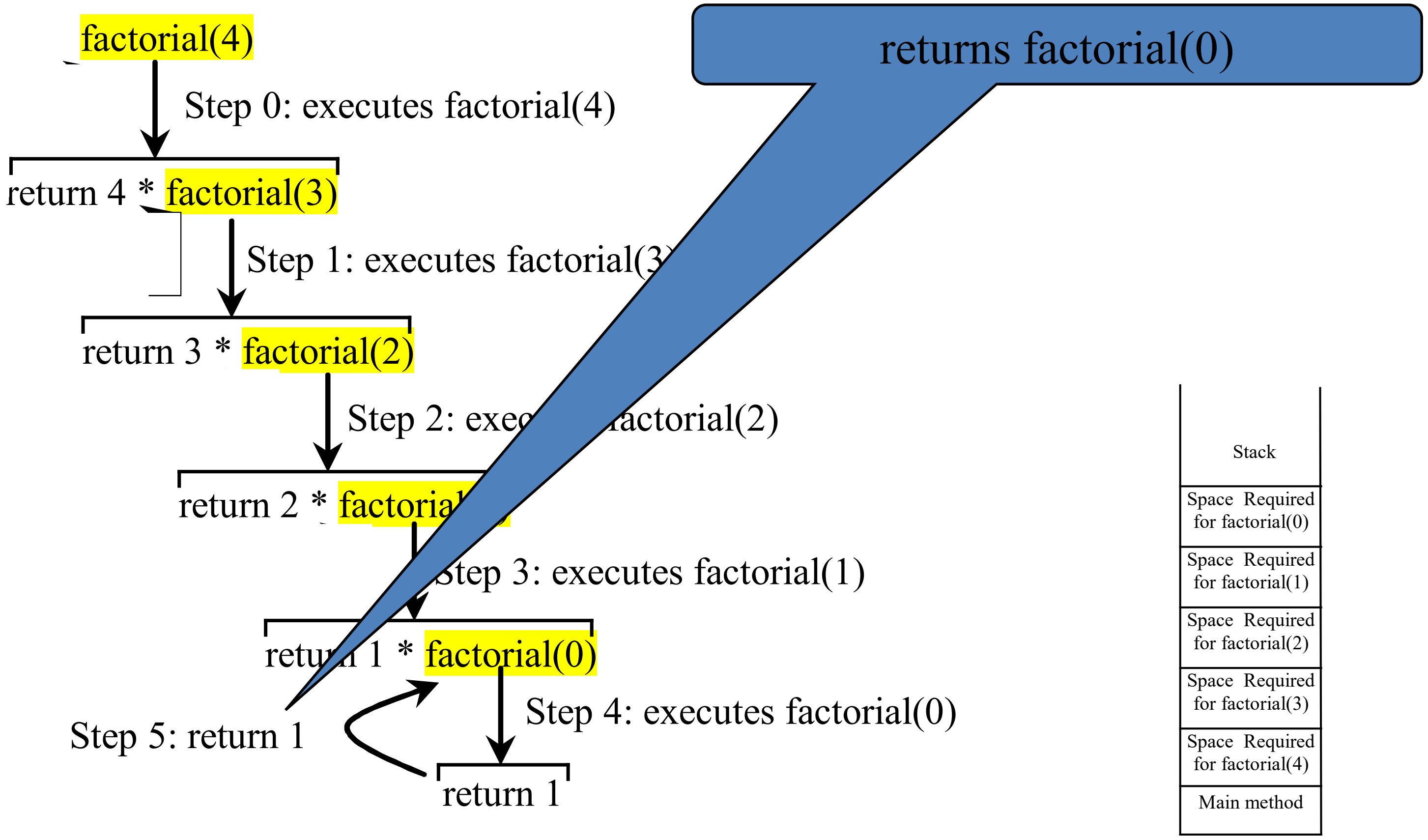
Recursive Factorial


Executes factorial(0)
factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factorial(2)
turn 2 * factorial(1)
Step 3: executes factorial(1)
return 1 * factorial(0)
Stack
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Recursive Factorial

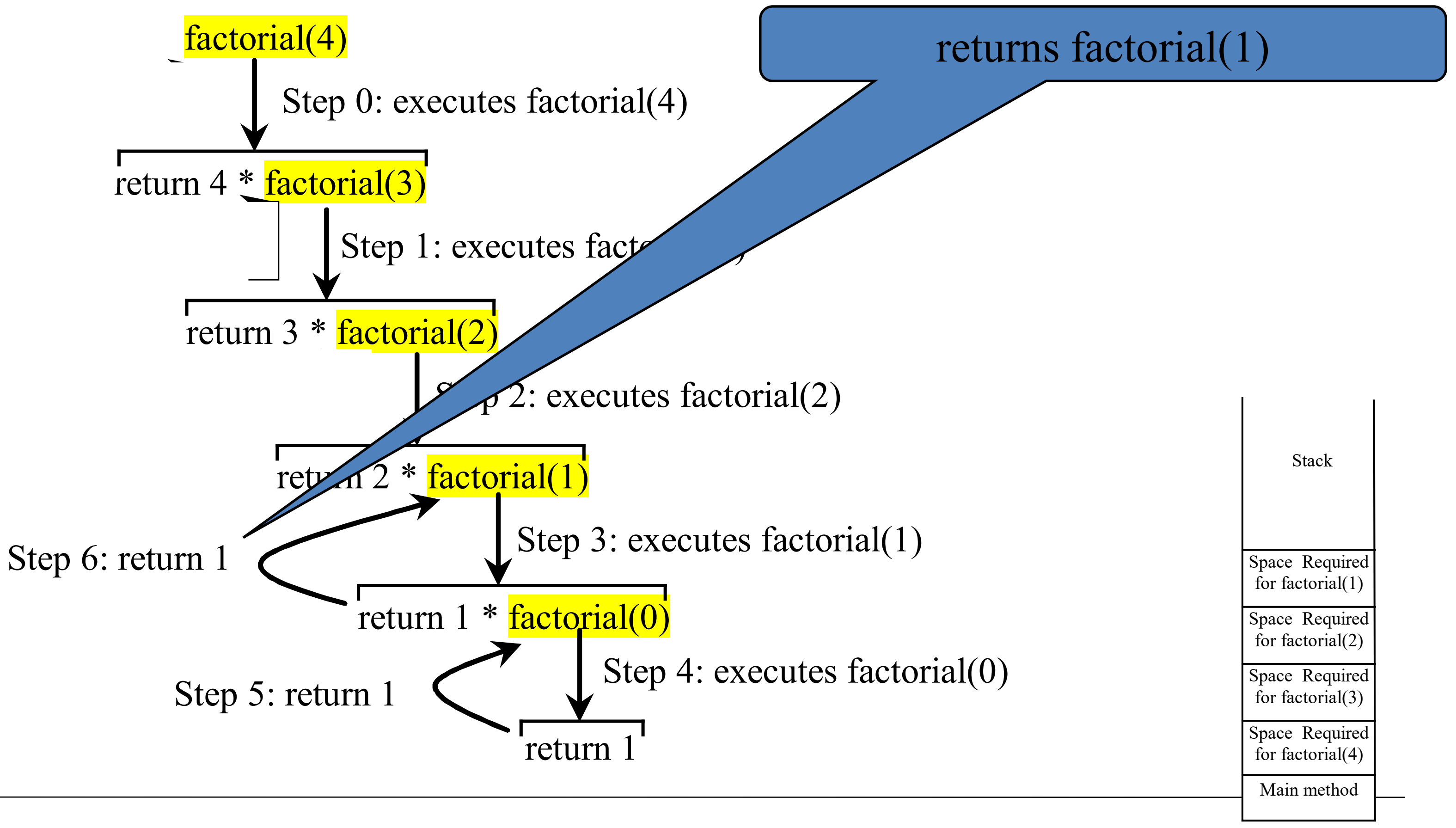

returns 1
factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factor
return 2 * factorial(1)
Step 3: execu factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
return 1
Stack
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Recursive Factorial

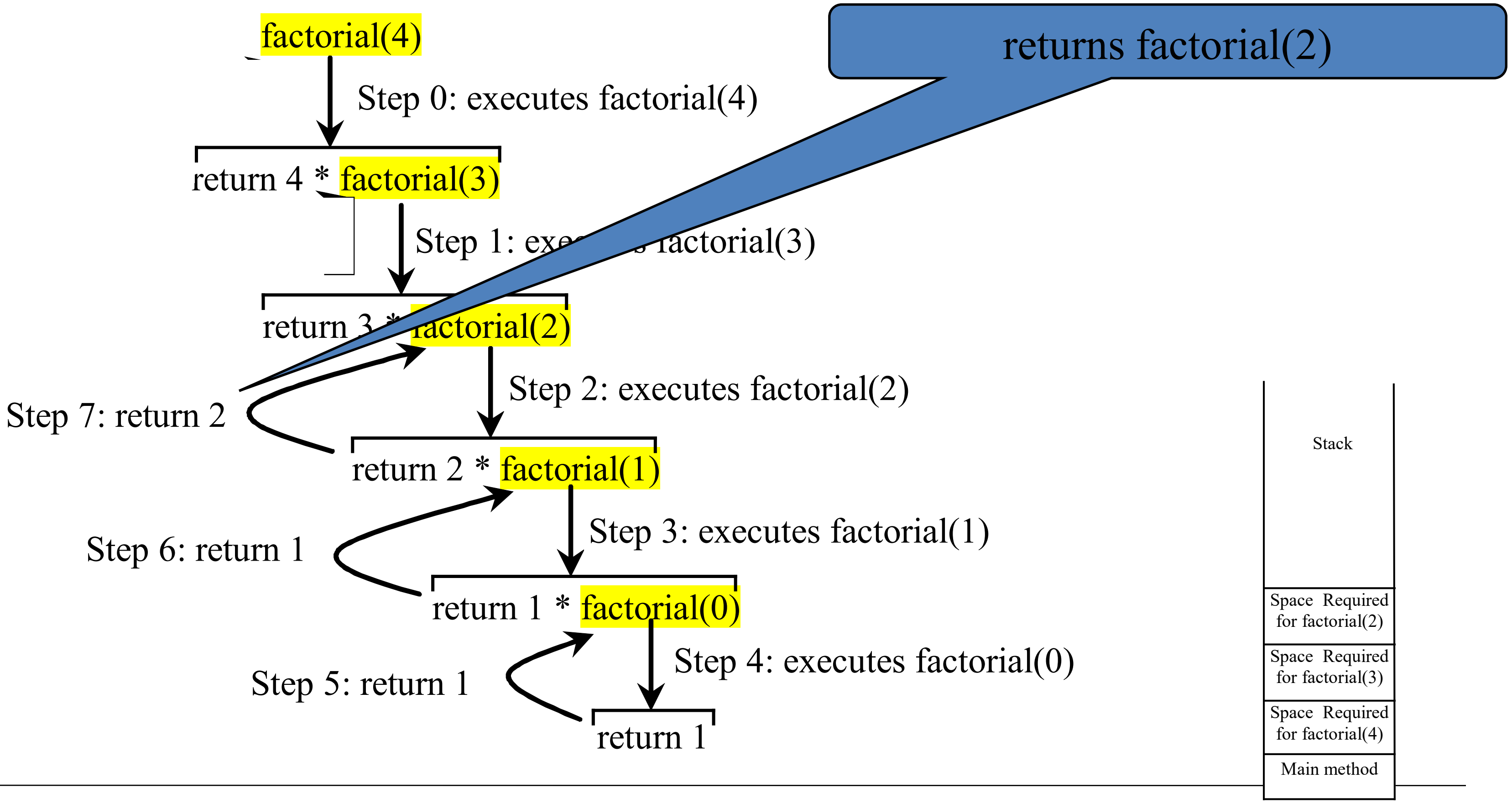


returns factorial(0)
factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factorial(2)
return 2 * factorial(1)
Step 3: executes factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
return 1
Step 5: return 1
Stack
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

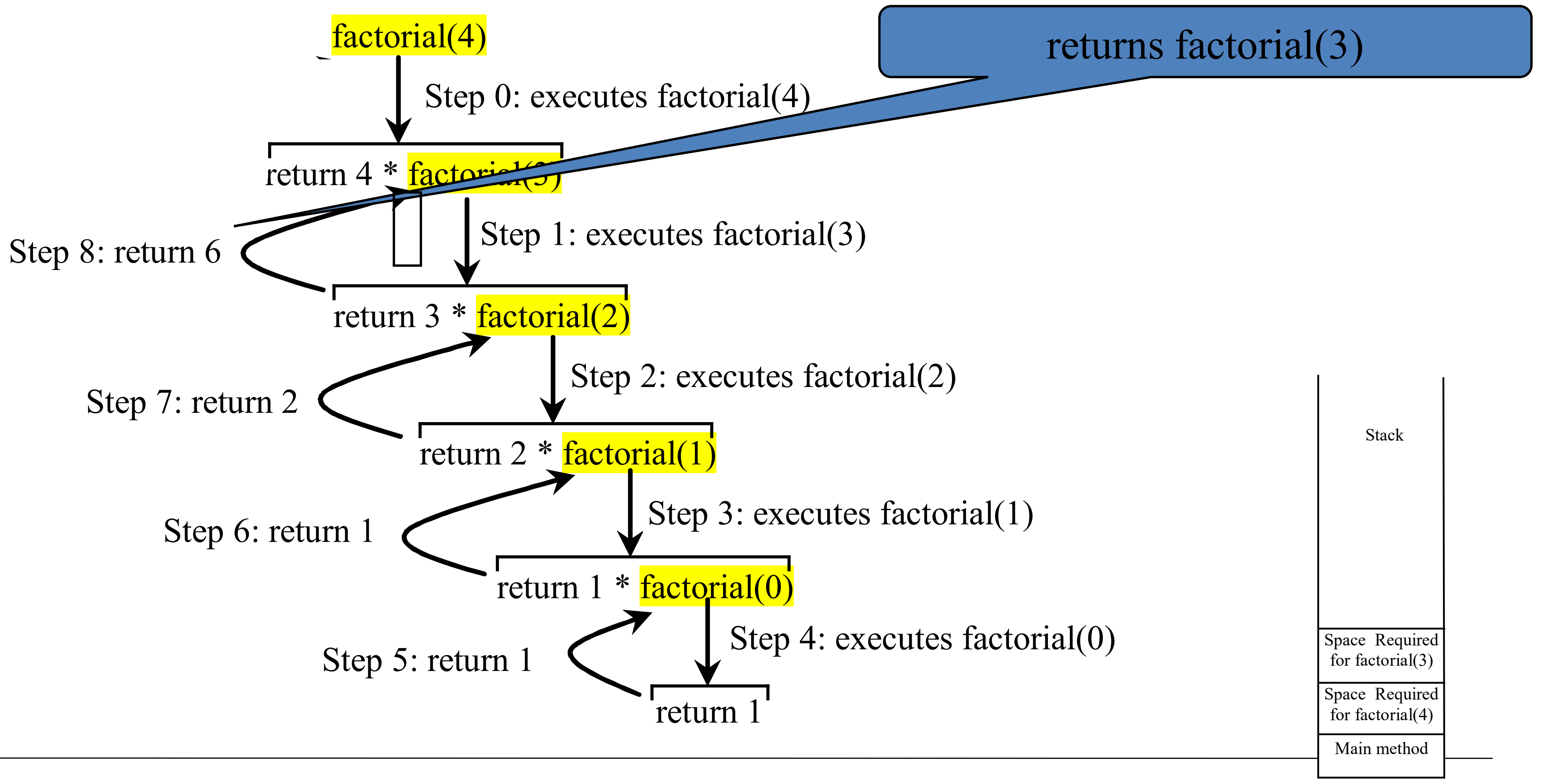
Recursive Factorial


factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes fact
return 3 * factorial(2)
p 2: executes factorial(2)
ret n 2 * factorial(1)
Step 3: executes factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
return 1
Step 5: return 1
Step 6: return 1
returns factorial(1)
Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

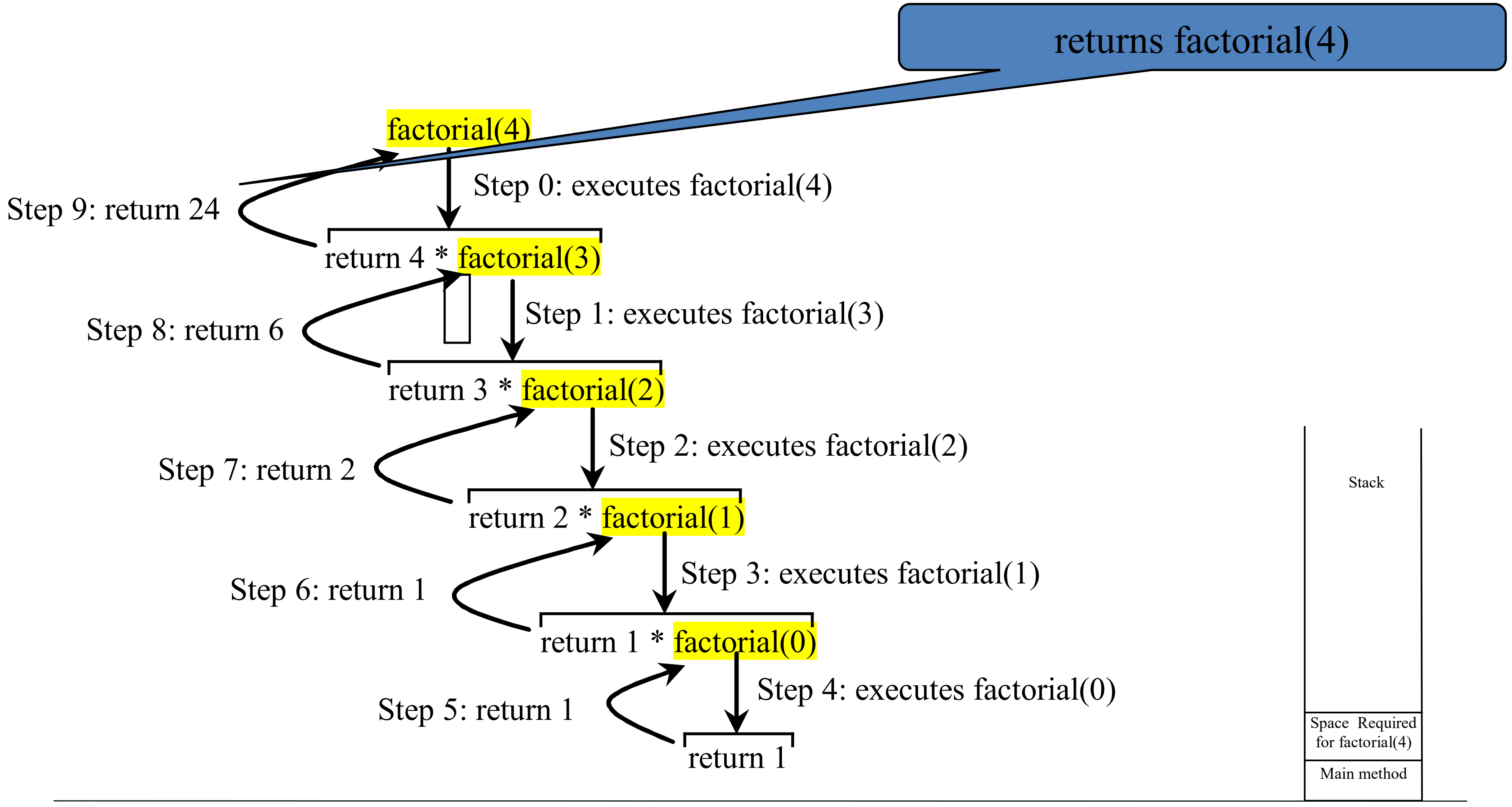
Recursive Factorial


factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factorial(2)
return 2 * factorial(1)
Step 3: executes factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
return 1
Step 5: return 1
Step 6: return 1
Step 7: return 2
returns factorial(2)
Stack
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

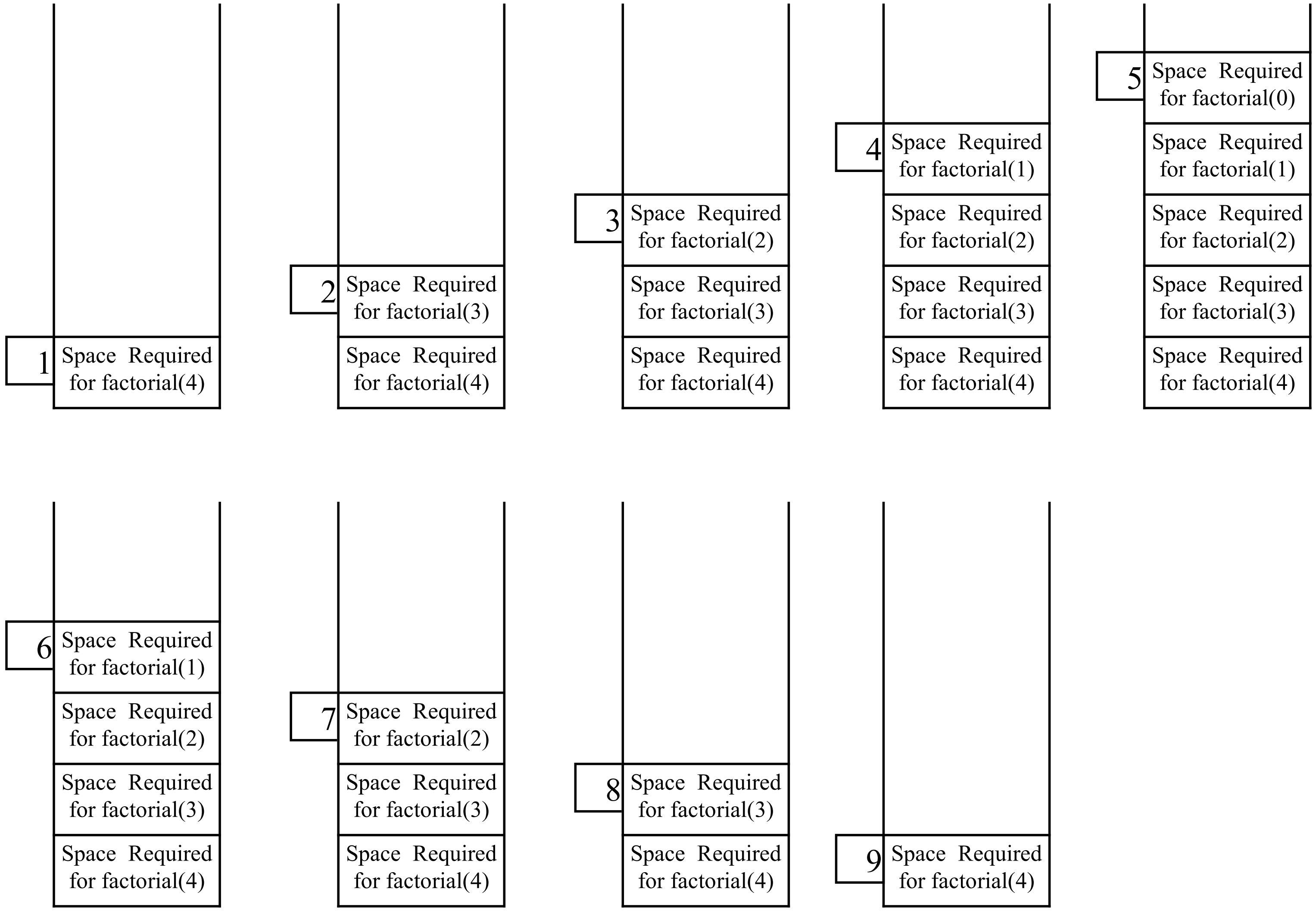
Recursive Factorial



Recursive Factorial



Recursive Factorial - Stack


1
Space Required for factorial(4)
2
Space Required for factorial(3)
Space Required for factorial(4)
3
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
4
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
5
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
6
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
7
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
8
Space Required for factorial(3)
Space Required for factorial(4)
9
Space Required for factorial(4)

Recursive Fibonacci

Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89…

indices: 0 1 2 3 4 5 6 7   8   9 10 11

fib(0) = 0;

fib(1) = 1;

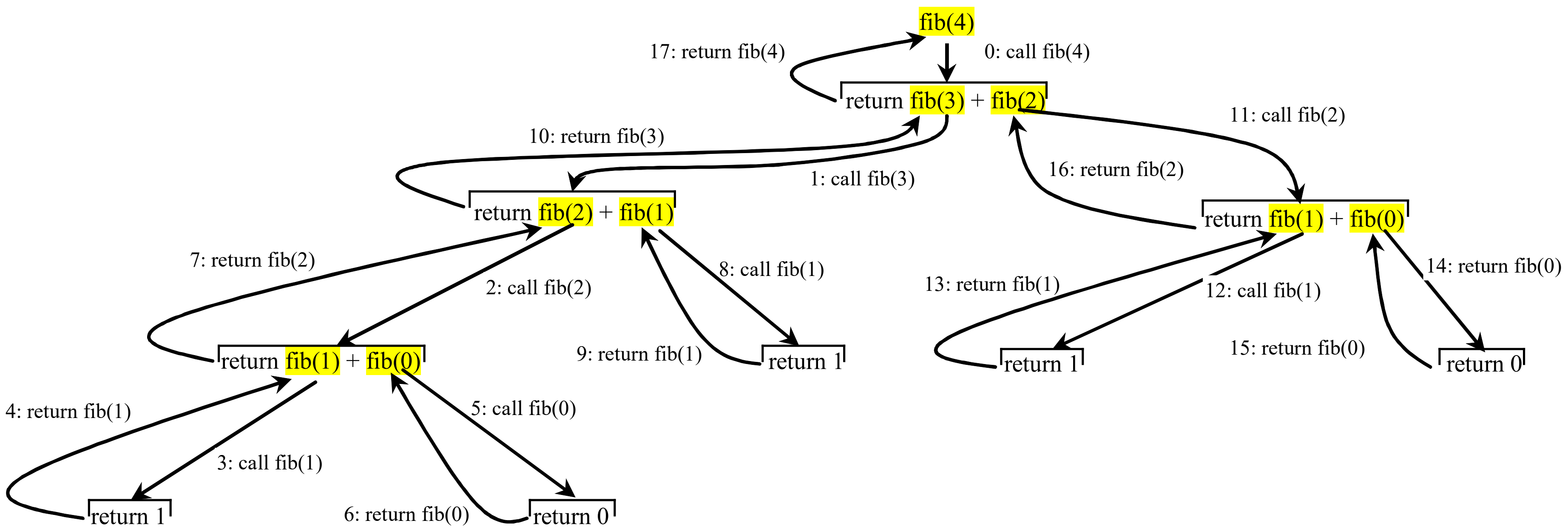
fib(index) = fib(index -1) + fib(index -2); index >=2

e.g. fib(3) = fib(2) + fib(1)

= etc…

= 2

Recursive Fibonacci



Questions?