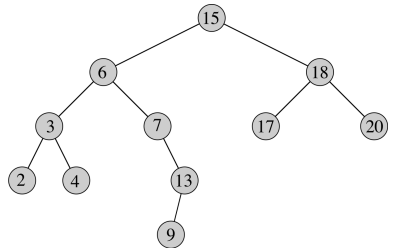


Binary Search Trees 2
Lecture 13

COSC 242 – Algorithms and Data Structures

Practical test announcement

Practical test is this Tuesday, August 18th.

A resit is also scheduled for this Friday, August 21st.

If you feel uncomfortable about at attending face-to-face while maintaining social distancing, students may optionally attend the 8am Tuesday stream.

If you choose to attend Tuesday 8am, and this is not your regularly scheduled lab time, please email your intent to Iain today.

Today’s outline

1. More about BSTs
2. Search
3. BST vs Binary search
4. Depth First Search
5. Inorder traversal
6. Preorder traversal
7. Postorder traversal

Today’s outline

1. More about BSTs
2. Search
3. BST vs Binary search
4. Depth First Search
5. Inorder traversal
6. Preorder traversal
7. Postorder traversal

More about BSTs

Like linked lists, BSTs are dynamic data structures that can easily grow and shrink.

Unlike linked lists, BSTs can be efficient to search, insert, and delete, as long as they remain balanced.

More about BSTs

BSTs support quite a few useful operations:

- Insert, search, and delete
- Can sort data
- Can traverse the data in various orders in O(*n*)

We've already seen insert. In this lecture and the next, we'll look at the other operations.

Today’s outline

1. More about BSTs
2. Search
3. BST vs Binary search
4. Depth First Search
5. Inorder traversal
6. Preorder traversal
7. Postorder traversal

Search


```
1:   function BST_Search(BST T, KeyType key)
2:       if T == NIL then
3:           return Not Found
4:       else if key == T -> key then
5:           return T
6:       else if key < T -> key then
7:           return BST_Search(T->left, key)
8:       else
9:           return BST_Search(T->right, key)
10:      end if
11:  end function
```

Search complexity

What’s the complexity of BST search?

Search complexity

What's the complexity of BST search?

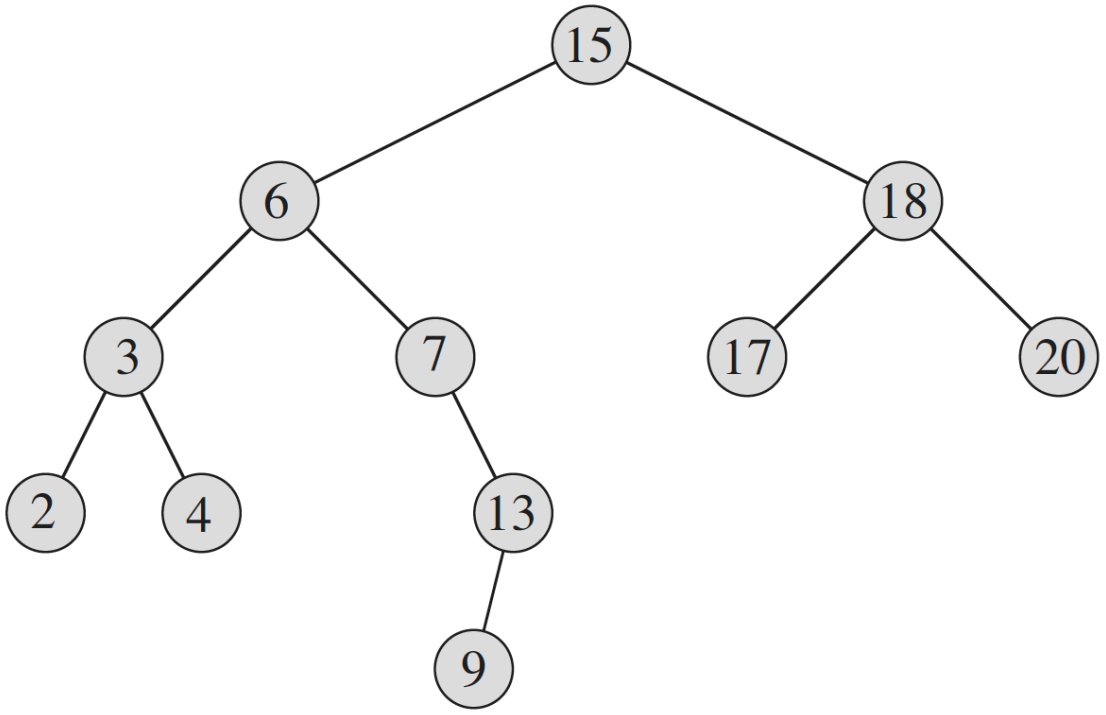
O(h), where h is the height of the tree.

We saw in L12 that for a **complete tree**, $h = \lfloor \log_2 n \rfloor$.

- Therefore, in best case, search should take $\boldsymbol{\Theta}(\log n)$.
- In worst case, where the tree is a linear chain of n nodes, search should take $\boldsymbol{\Theta}(n)$.

Search example 1

Trace the search path for Key = 13, indicating branch points in the pseudocode.




```
function BST_Search(BST T, KeyType key)
    if T == NIL then
        return Not Found
    else if key == T -> key then
        return T
    else if key < T -> key then
        return BST_Search(T->left, key)
    else
        return BST_Search(T->right, key)
    end if
end function
```

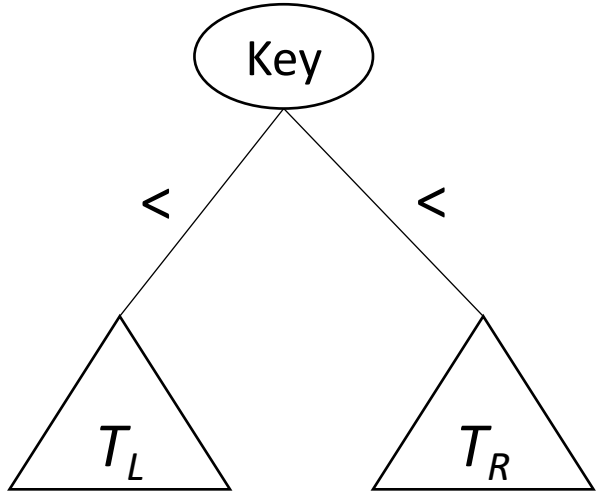
Today’s outline

1. More about BSTs
2. Search
3. BST vs Binary search
4. Depth First Search
5. Inorder traversal
6. Preorder traversal
7. Postorder traversal

BST vs Binary search

In L06 we looked at the binary search algorithm. Lets explore how the BST and Binary search algorithm are related.

Refresh: A BST is a **data structure** that consists of a root node containing a key field and data fields, a left subtree T_L, and a right subtree T_R.



BST vs Binary search

Refresh: Binary search is an efficient **algorithm** for finding an item in a sorted list.

Given an array A[0..n-1] of sorted keys, to locate a target value x, find index $m = \lfloor (n-1)/2 \rfloor$ of middle element, then compare x with A[m]. Based on that value, we then divide the array in half, and conquer.

Binary search

Notice how the binary search algorithm works. At each recursion, we remove half the search space. This is the same behavior that we saw for BST_Search.

This is because binary search works on an *ordered collection*. That collection can be a sorted array, or a BST, which is also sorted.

Comparing BST and BS

```
Binary_search(A, x, low, high):
1.    if low > high then
2.       report failure and stop
3.    else
4.       mid ← (low + high) / 2
5.       if x = A[mid] then
6.          report success and return mid
7.       else if x < A[mid] then
8.          return Binary_search(A, x, low, mid – 1)
9.       else if x > A[mid] then
10.         return Binary_search(A, x, mid+1, high)
```

```
1:    function BST_Search(BST T, KeyType key)
2:       if T == NIL then
3:          return Not Found
4:       else if key == T -> key then
5:          return T
6:       else if key < T -> key then
7:          return BST_Search(T->left, key)
8:       else
9:          return BST_Search(T->right, key)
10:      end if
11:   end function
```

Binary search

Binary search was first presented in the context of an ordered array.

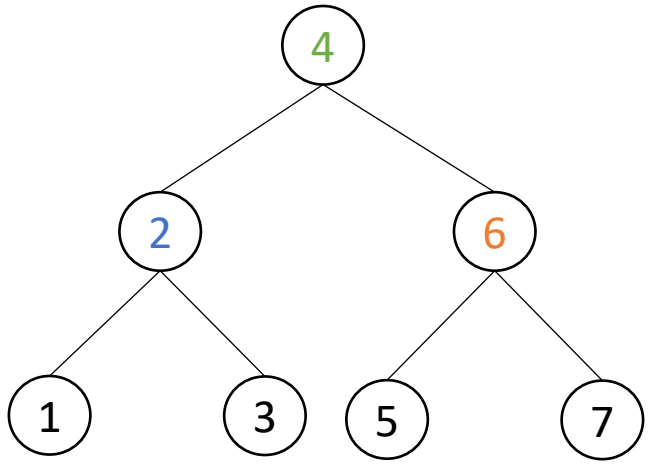
But more generally, Binary Search operates on an *ordered collection*. That collection can be a sorted array, or a BST, which is also sorted.

In a binary search we need to do three things: get an element, which if it's an array is the "middle" element; return the left object/subarray; or return the right object/subarray.

Binary search

In fact, an array can also be written in tree form.

The array [1 2 3 4 5 6 7] corresponds to the tree:



If we performed a binary search on our sorted array, the coloured “middle” elements correspond to the accordingly coloured nodes in our binary search tree.

Today’s outline

1. More about BSTs
2. Search
3. BST vs Binary search
4. Depth First Search
5. Inorder traversal
6. Preorder traversal
7. Postorder traversal

Traversal

Suppose we want to print the items in a BST in sorted order by key value.

We need to **traverse** (walk over) the tree, pausing at the right moment to print a node, so that we print the nodes in the right order (with increasing key values).

On each node we'll perform an operation, which we'll call *Process*. In reality, the process operation could be anything, like print or update.

Traversal

We will three examine three common forms of traversal:

- Pre-order
- In-order
- Post-order

The prefix term (pre-, in-, post-) is in reference to when the root node is processed relative to the left and right subtrees.

Depth first search

These three traversal algorithms form part of a more general search strategy known as **depth-first search (DFS)**.

In a DFS, the tree is deepened as much as possible on each child before going to the next sibling.

Depth first search

The general recursive pattern of DFS is:

Go down one level to *T*, the recursive argument. If *T* exists (non-empty), then execute the following operations in a specified order:

(L) Recursively traverse T's left subtree

(R) Recursively traverse T's right subtree

(T) Process the current node *T*

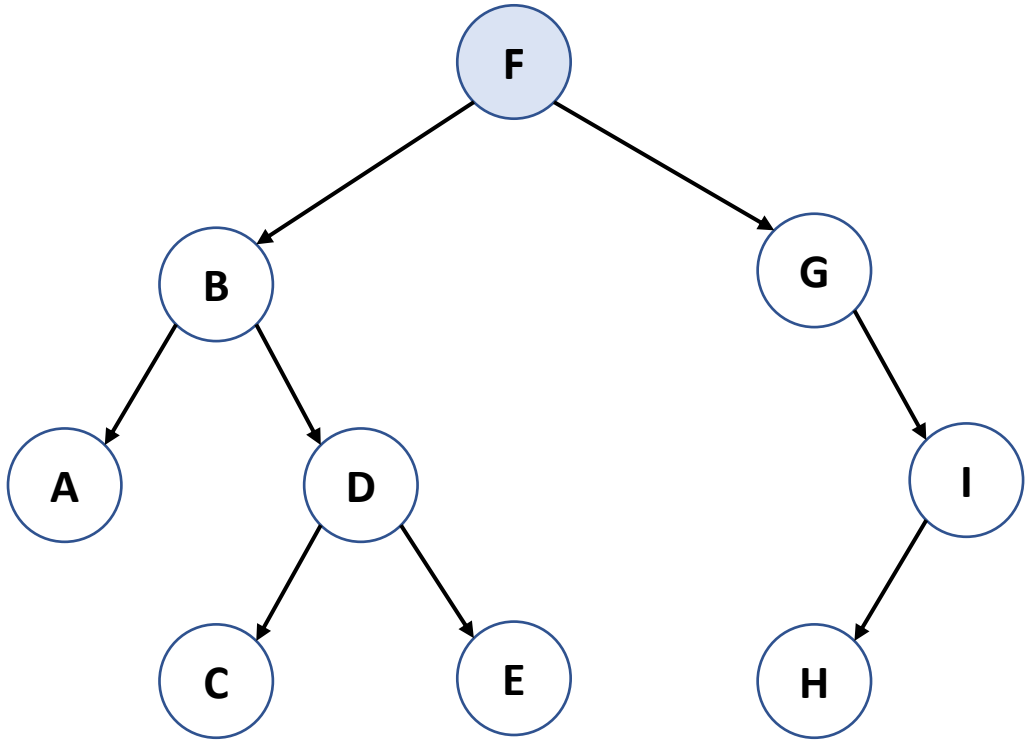
Return by going up one level, arriving at the parent node of *T*.

Today’s outline

1. More about BSTs
2. Search
3. BST vs Binary search
4. Depth First Search
5. Inorder traversal
6. Preorder traversal
7. Postorder traversal

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

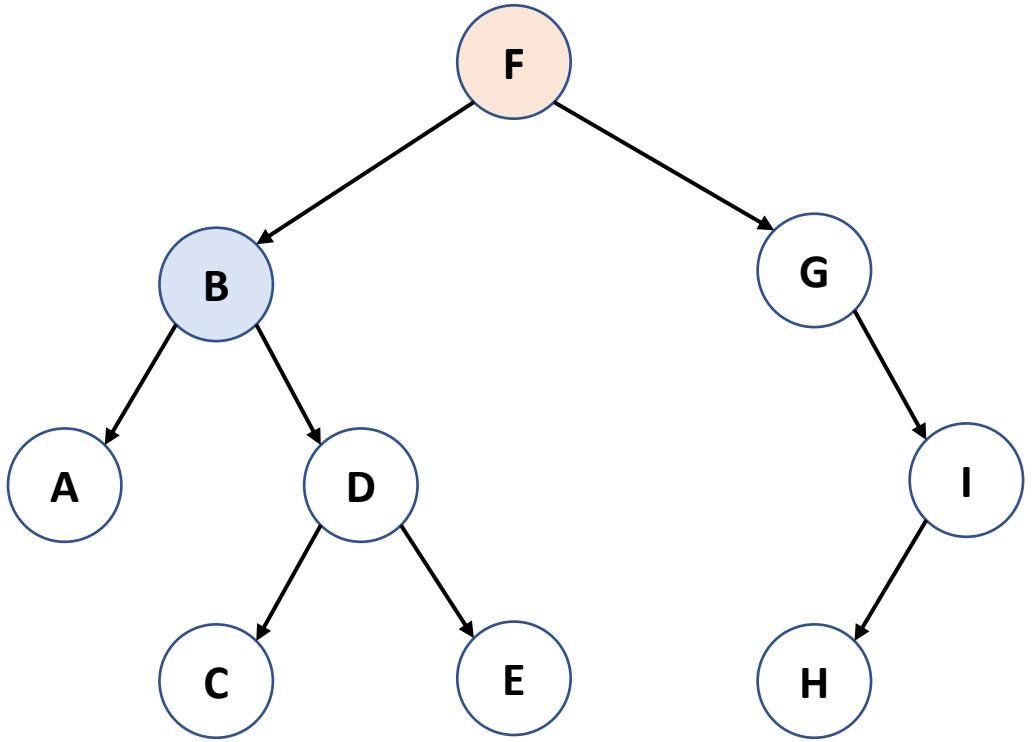
= completed node

✓ = Element processed

Process output:

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

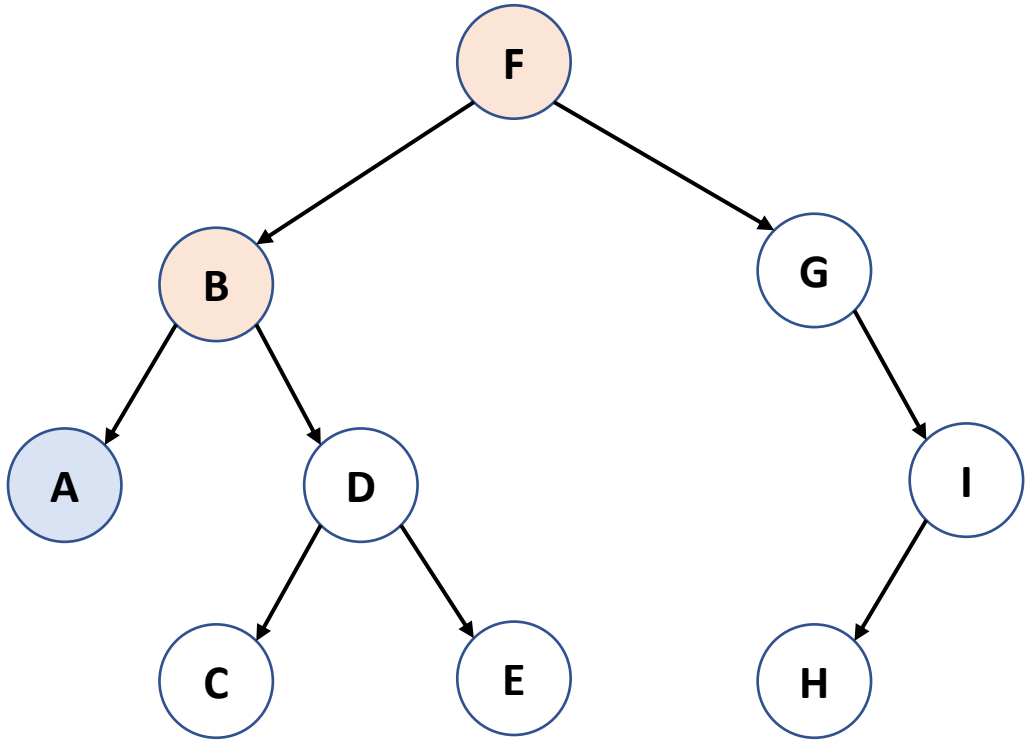
= completed node

✓ = Element processed

Process output:

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

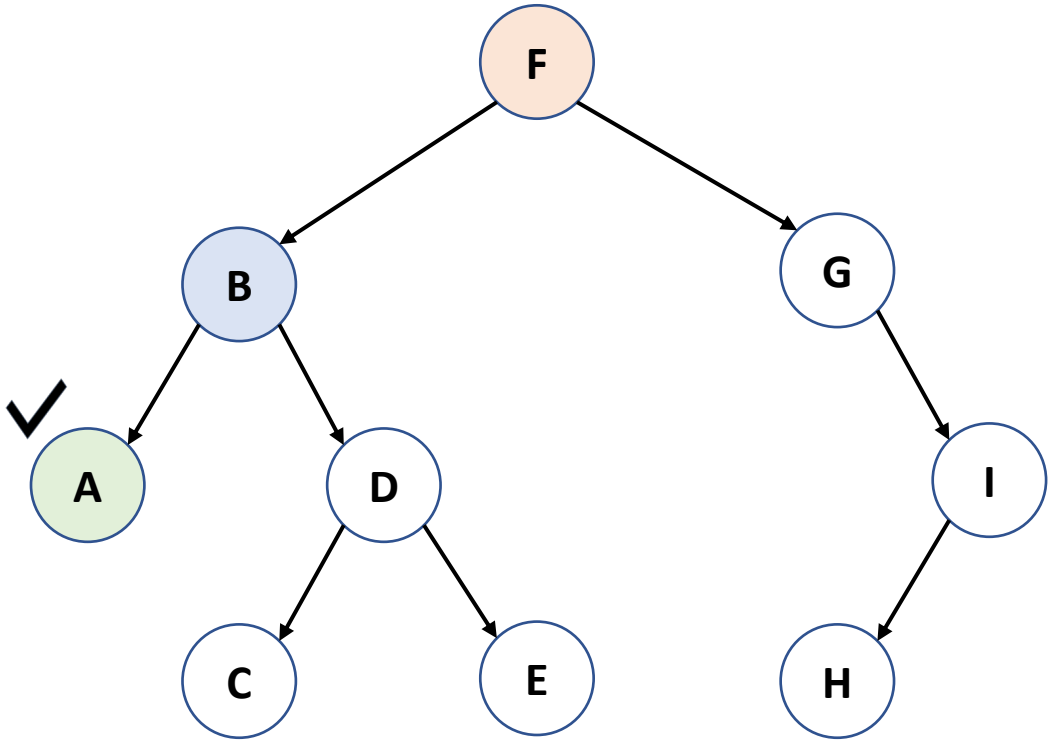
= completed node

✓ = Element processed

Process output:

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

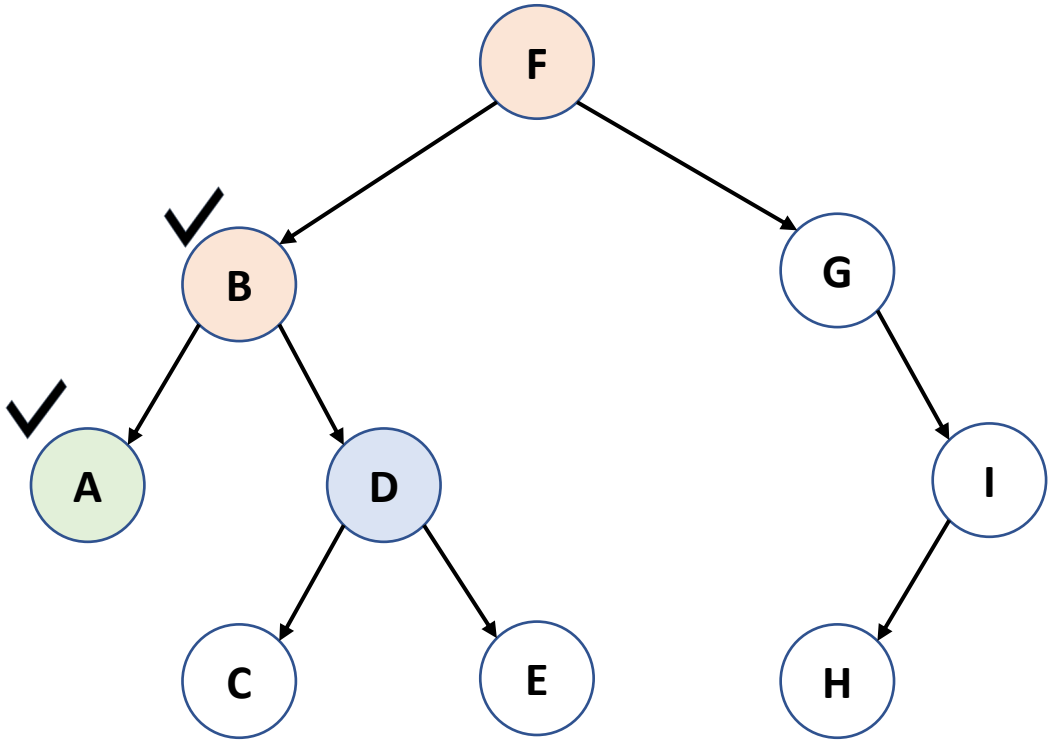
= completed node

= Element processed

Process output: A

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

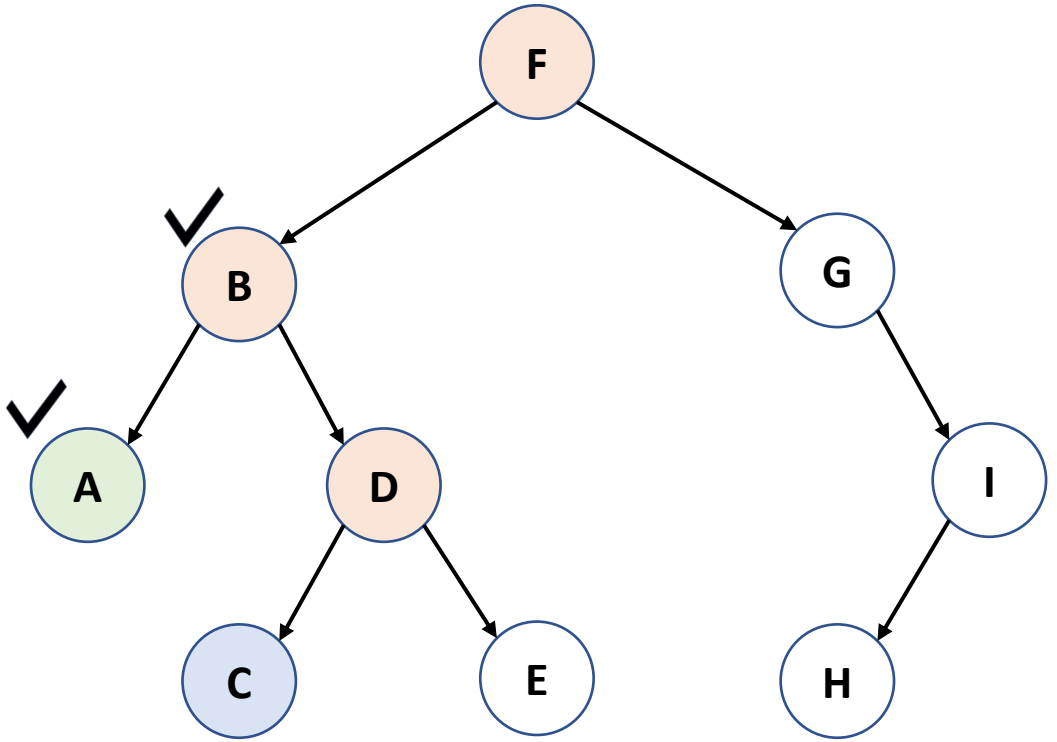
= completed node

= Element processed

Process output: A B

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

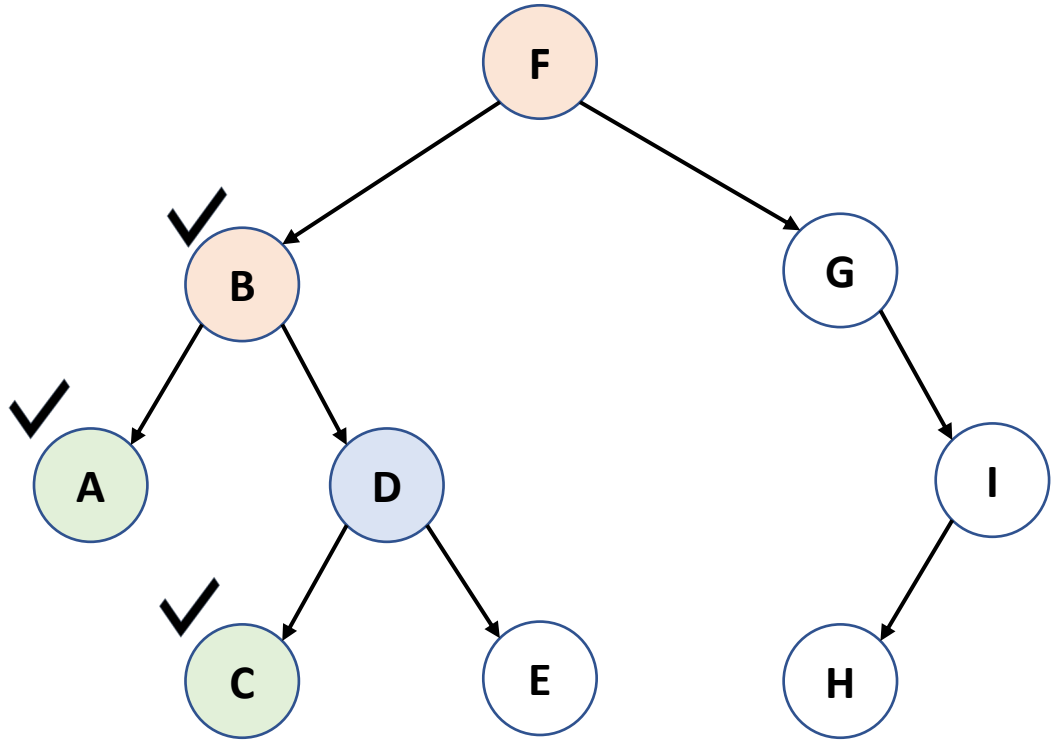
= completed node

= Element processed

Process output: A B

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

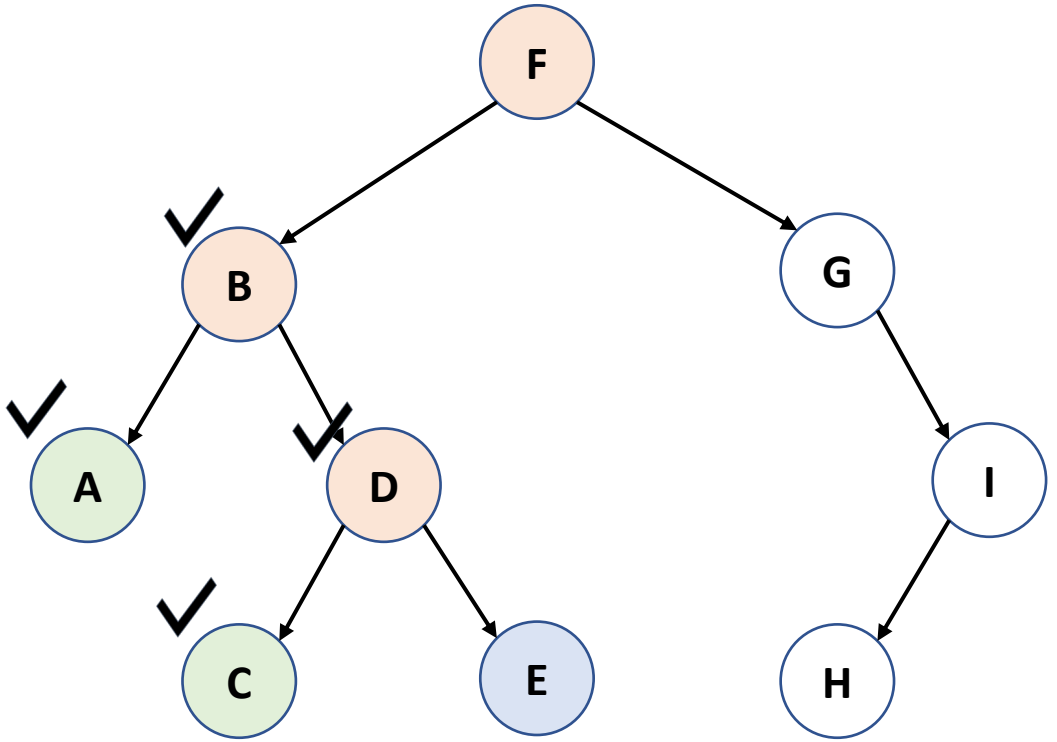
= completed node

= Element processed

Process output: A B C

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

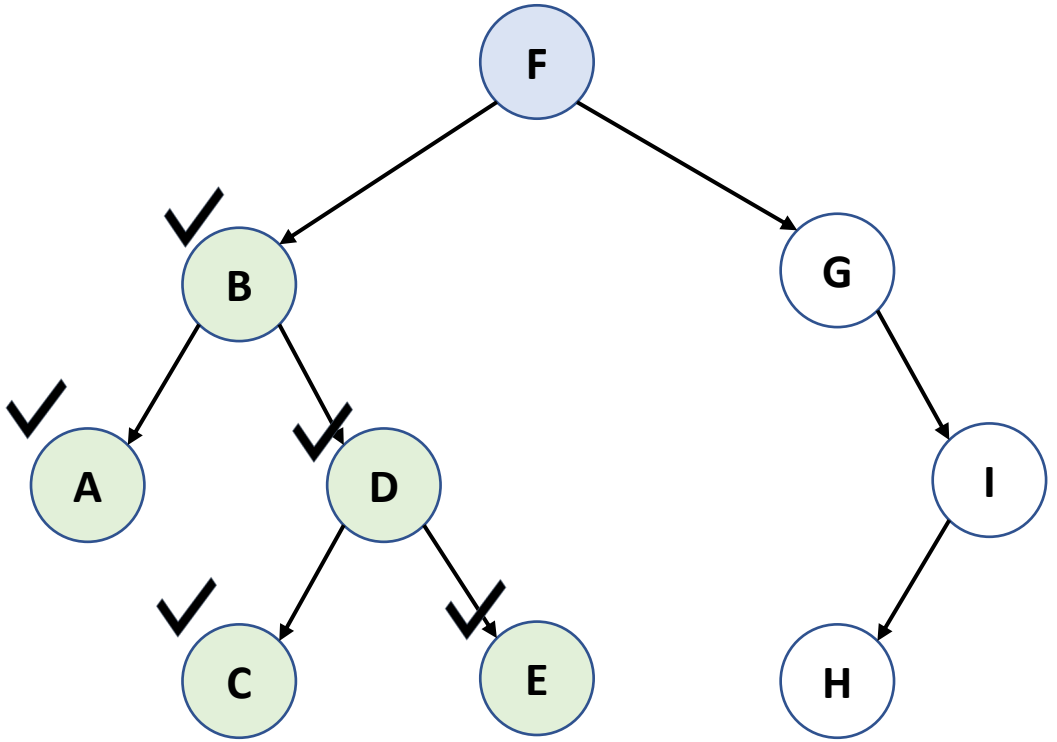
= completed node

= Element processed

Process output: A B C D

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

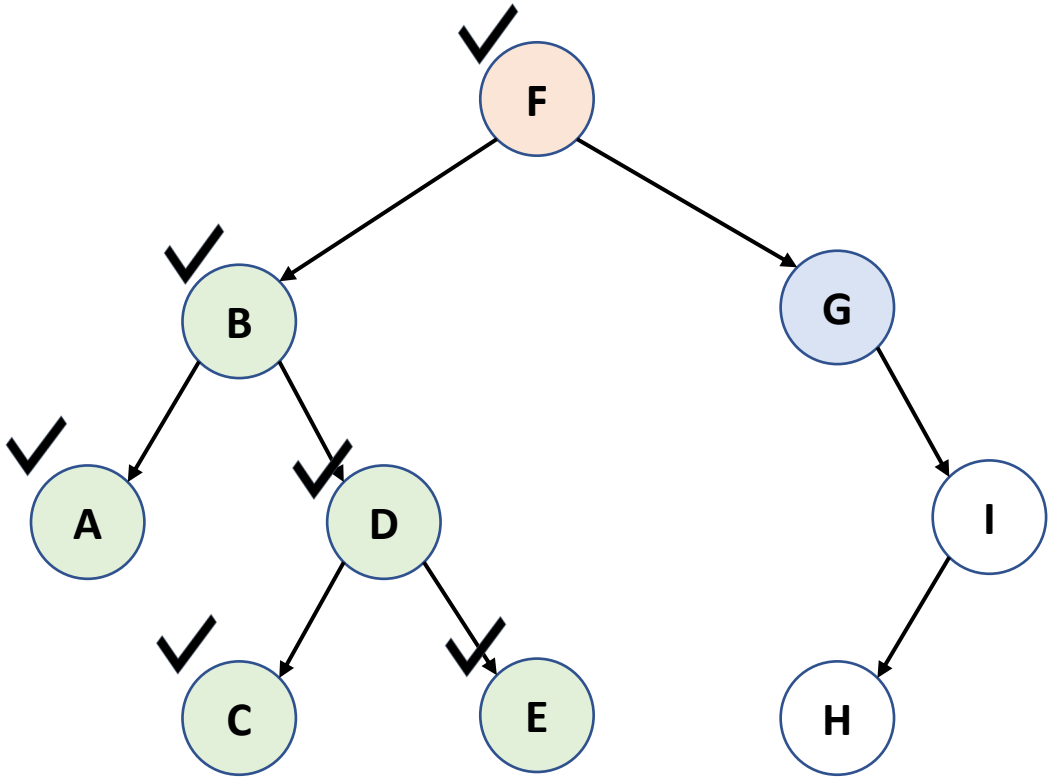
= completed node

= Element processed

Process output: A B C D E

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

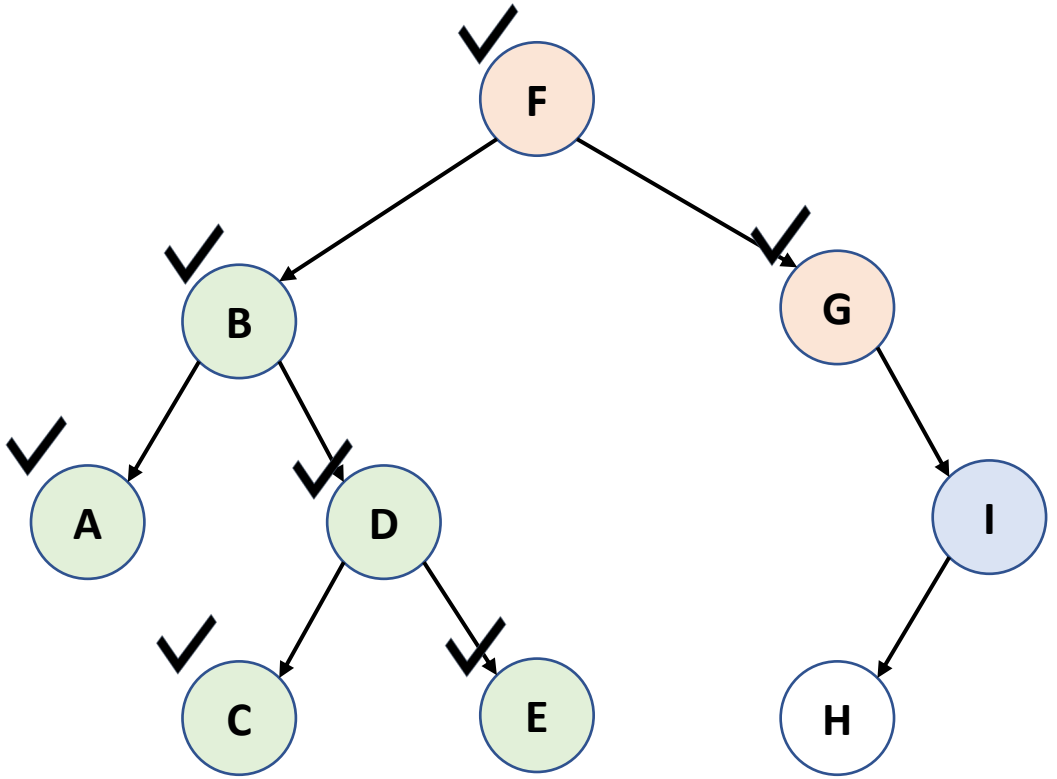
= completed node

= Element processed

Process output: A B C D E F

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

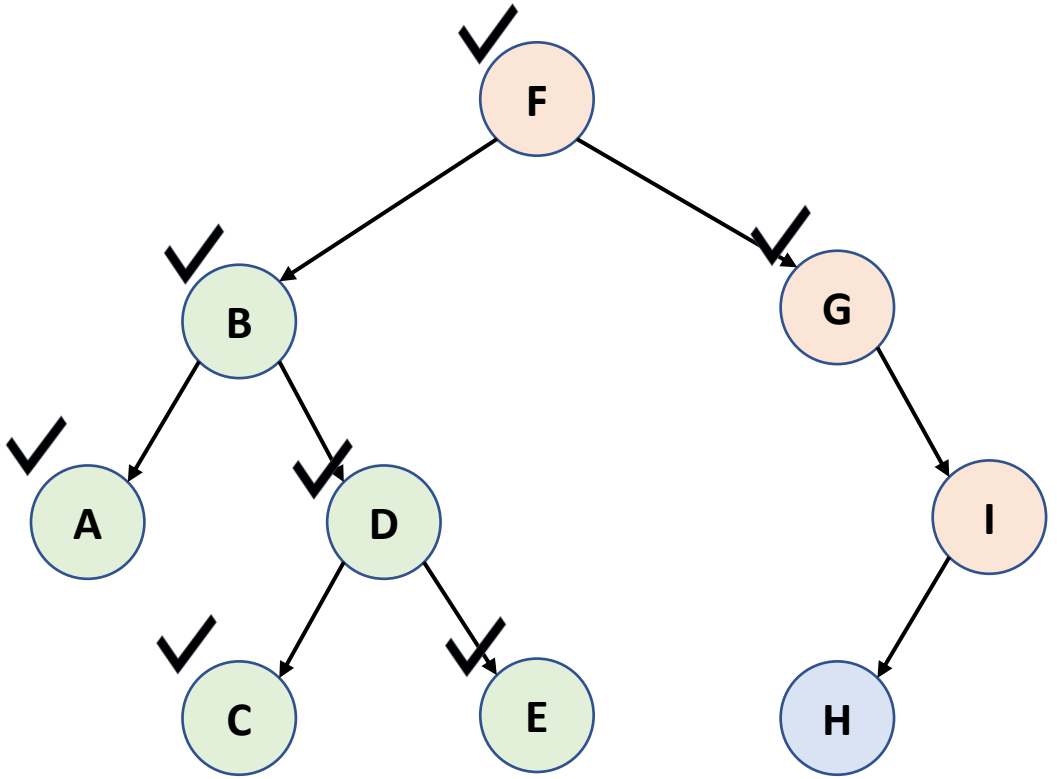
= completed node

= Element processed

Process output: A B C D E F G

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```



= current node

= processing (on call stack)

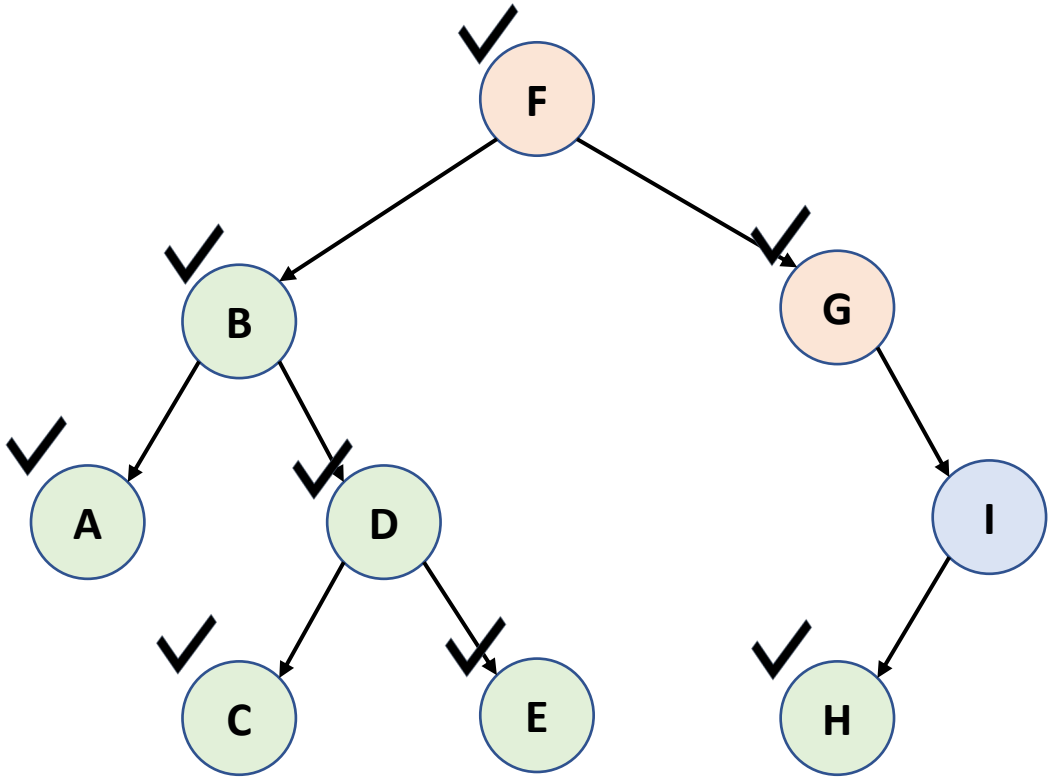
= completed node

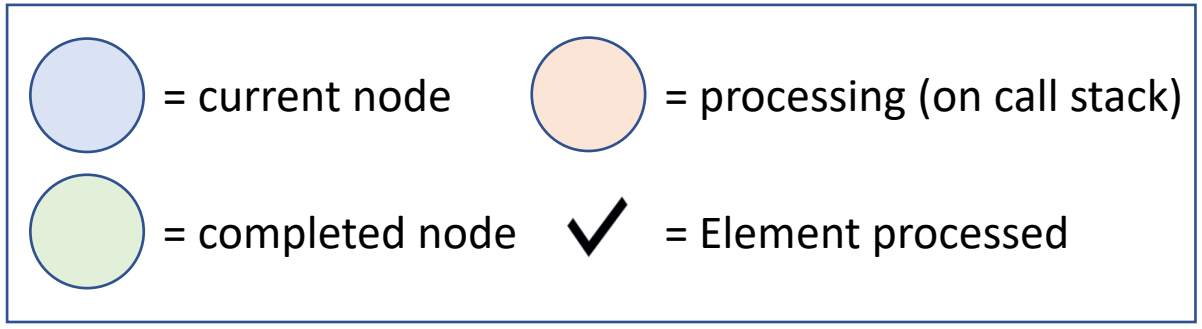
= Element processed

Process output: A B C D E F G

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```

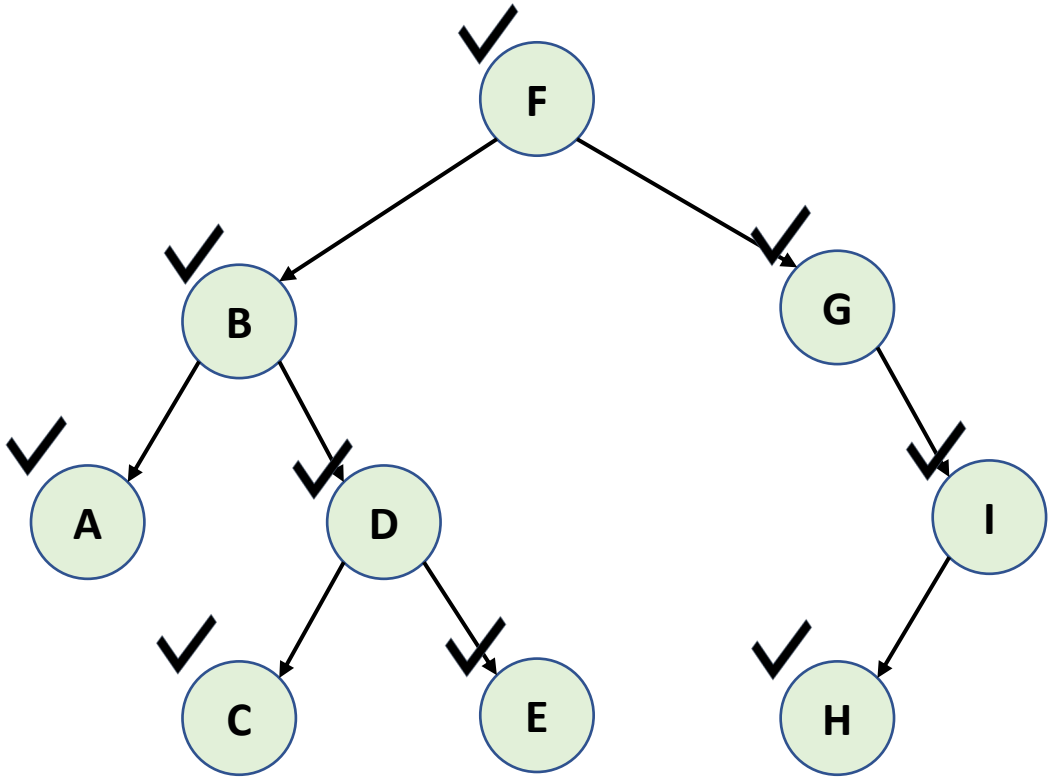


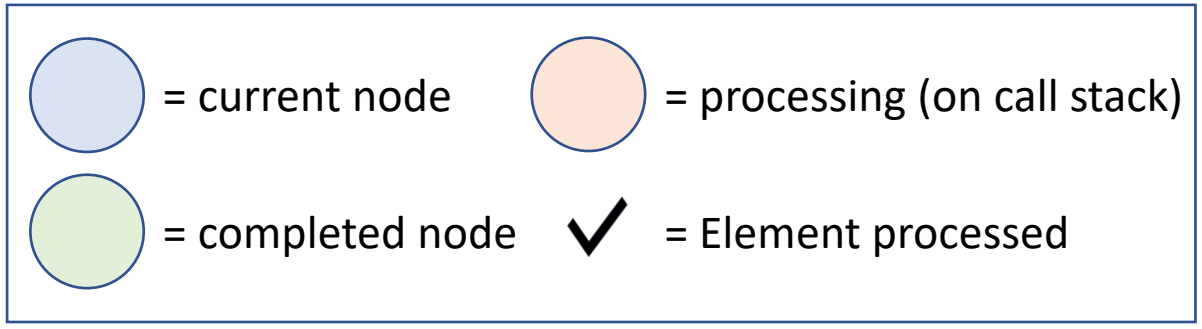


Process output: A B C D E F G H

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T->left)
4:          process(T)
5:          Inorder_traversal(T->right)
6:      end if
7:  end procedure
```





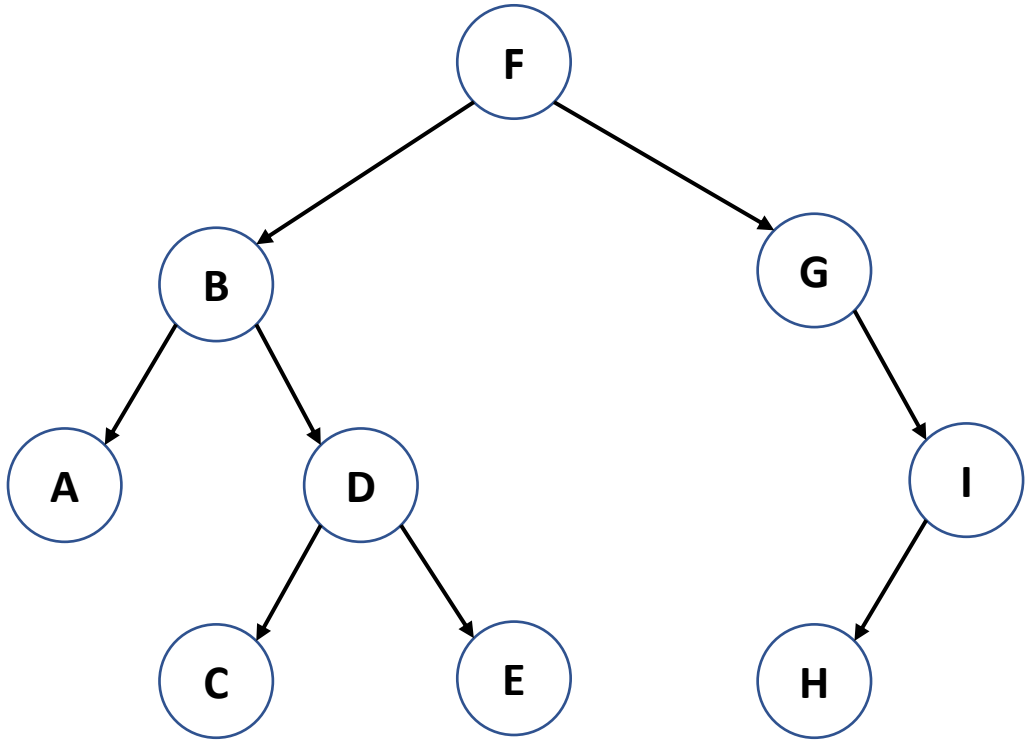
Process output: A B C D E F G H I

Today’s outline

1. More about BSTs
2. Search
3. BST vs Binary search
4. Depth First Search
5. Inorder traversal
6. Preorder traversal
7. Postorder traversal

Preorder traversal

```
1:  function Preorder_traversal(BST T)
2:      if T ≠ NIL then
3:          process(T)
4:          Preorder_traversal(T->left)
5:          Preorder_traversal(T->right)
6:      end if
7:  end procedure
```

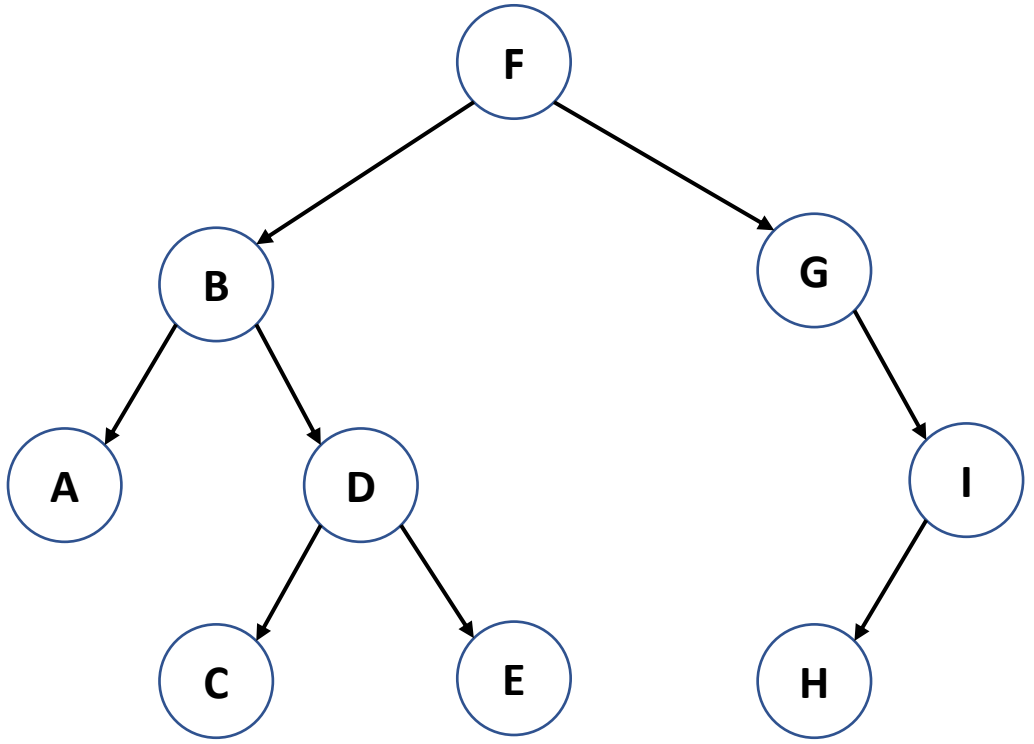


What's the output?

Process output:

Preorder traversal

```
1:  function Preorder_traversal(BST T)
2:      if T ≠ NIL then
3:          process(T)
4:          Preorder_traversal(T->left)
5:          Preorder_traversal(T->right)
6:      end if
7:  end procedure
```



What's the output?

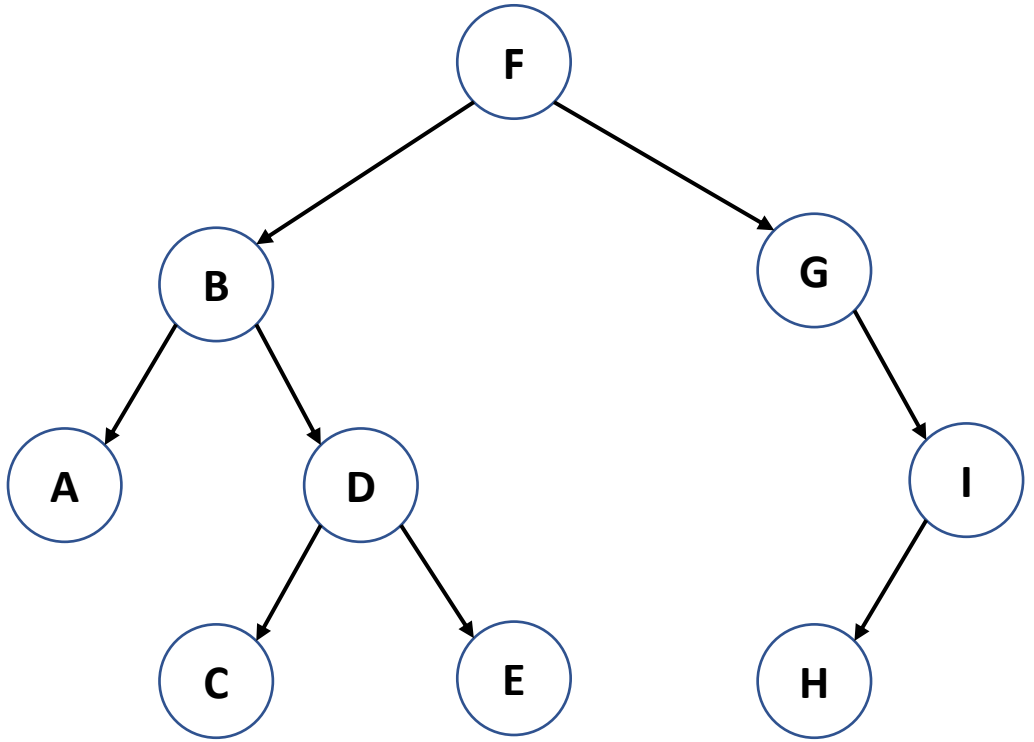
Process output: F B A D C E G I H

Today’s outline

1. More about BSTs
2. Search
3. BST vs Binary search
4. Depth First Search
5. Inorder traversal
6. Preorder traversal
7. Postorder traversal

Postorder traversal

```
1:  function Postorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Postorder_traversal(T->left)
4:          Postorder_traversal(T->right)
5:          process(T)
6:      end if
7:  end procedure
```

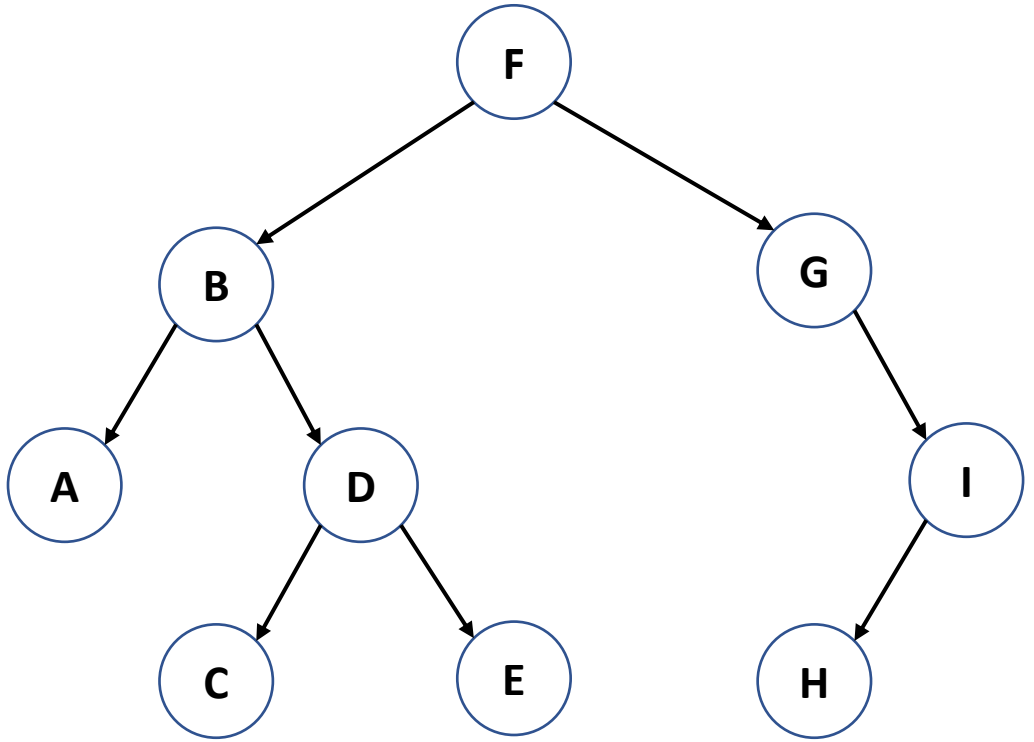


What's the output?

Process output:

Postorder traversal

```
1:  function Postorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Postorder_traversal(T->left)
4:          Postorder_traversal(T->right)
5:          process(T)
6:      end if
7:  end procedure
```



What's the output?

Process output: A C E D B H I G F

Suggested reading

Search is discussed in Section 12.2

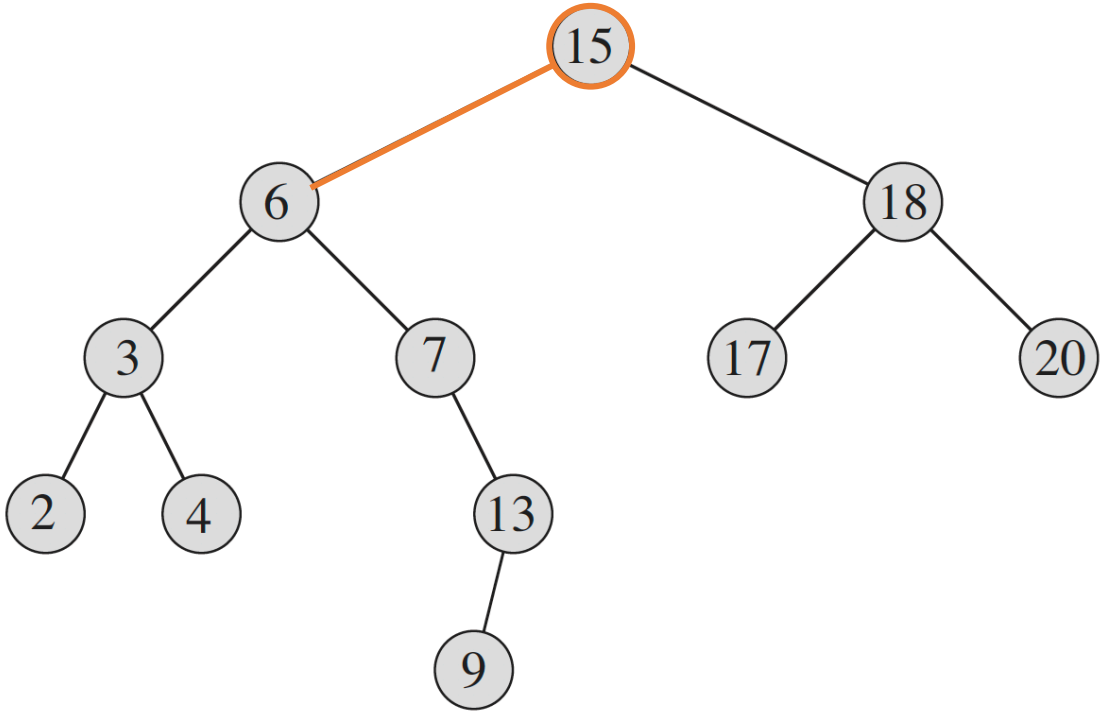
Preorder traversal is discussed in Section 12.1 (except they call it Preorder walk).

Binary search is covered in L06 and Section 2.3.

Solutions

Search example 1

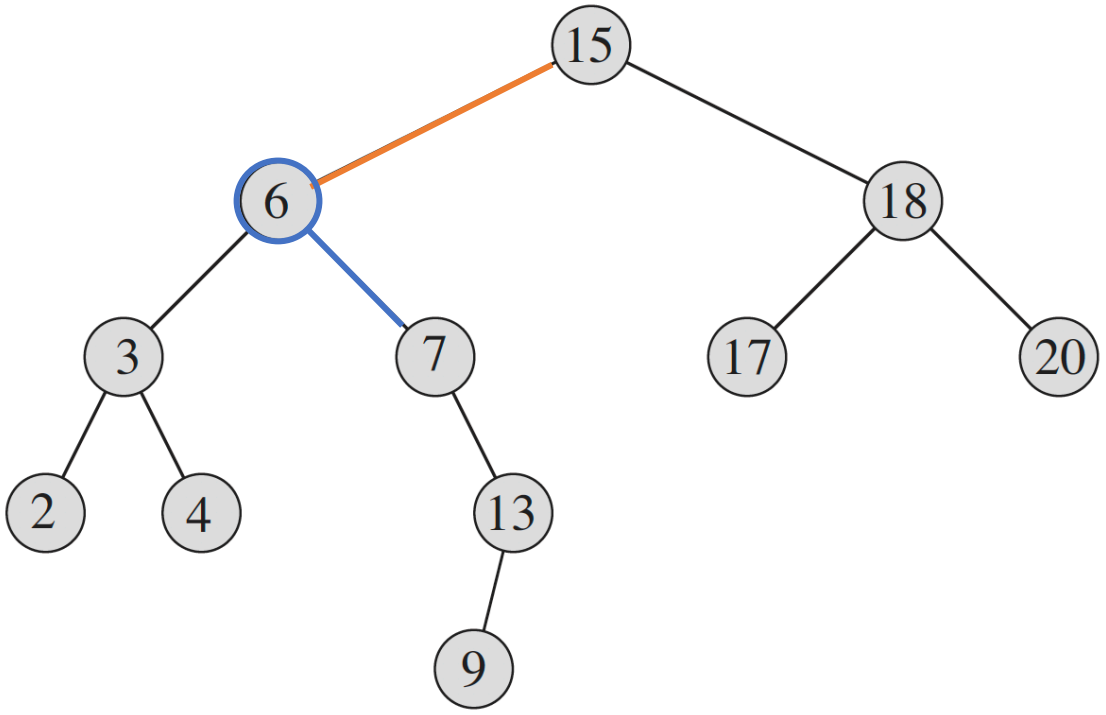
Trace the search path for Key = 13, indicating branch points in the pseudocode.




```
function BST_Search(BST T, KeyType key)
    if T == NIL then
        return Not Found
    else if key == T -> key then
        return T
    else if key < T -> key then
        return BST_Search(T->left, key)
    else
        return BST_Search(T->right, key)
    end if
end function
```

Search example 1

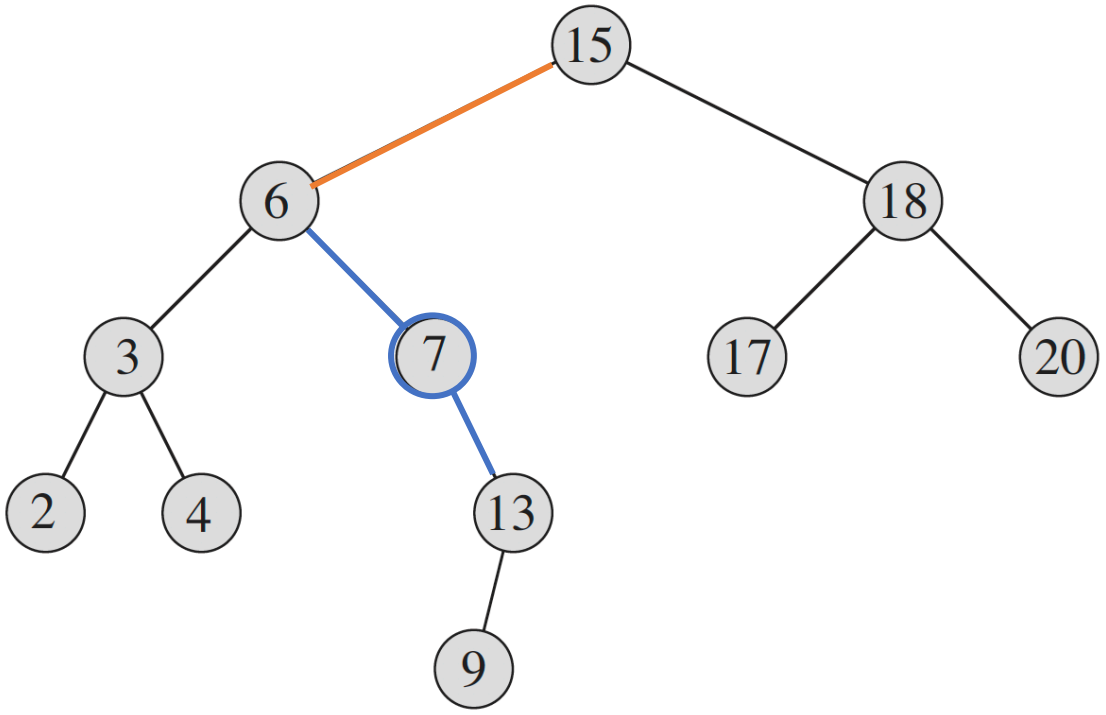
Trace the search path for Key = 13, indicating branch points in the pseudocode.




```
function BST_Search(BST T, KeyType key)
    if T == NIL then
        return Not Found
    else if key == T -> key then
        return T
    else if key < T -> key then
        return BST_Search(T->left, key)
    else
        return BST_Search(T->right, key)
    end if
end function
```

Search example 1

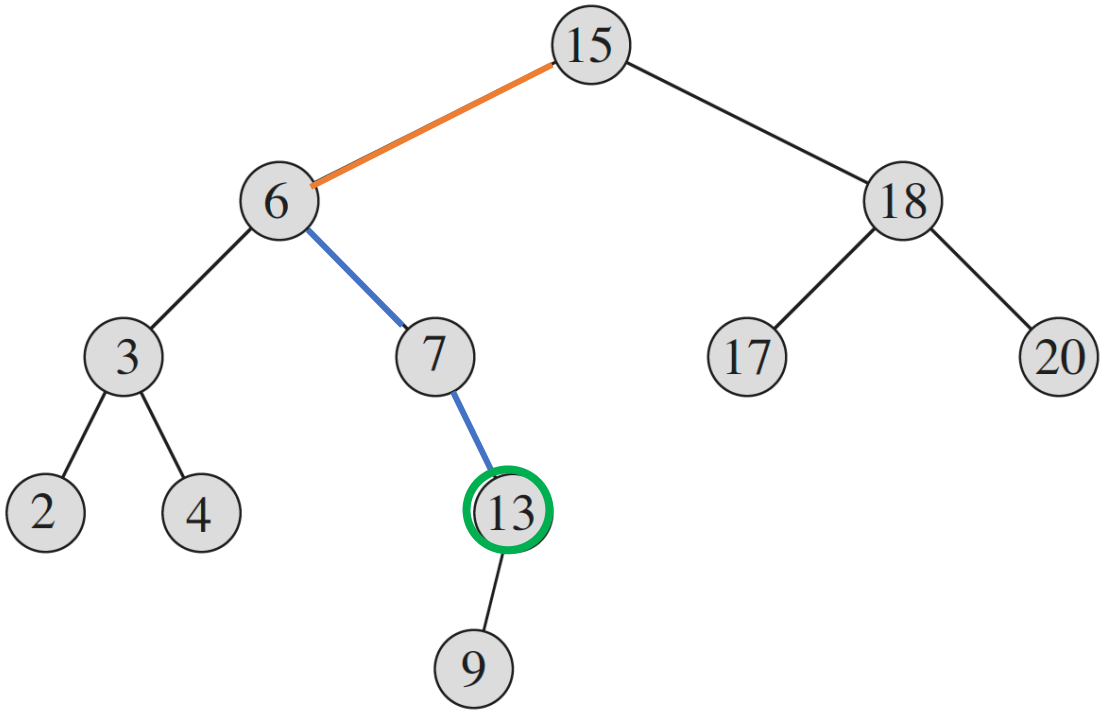
Trace the search path for Key = 13, indicating branch points in the pseudocode.




```
function BST_Search(BST T, KeyType key)
    if T == NIL then
        return Not Found
    else if key == T -> key then
        return T
    else if key < T -> key then
        return BST_Search(T->left, key)
    else
        return BST_Search(T->right, key)
    end if
end function
```

Search example 1

Trace the search path for Key = 13, indicating branch points in the pseudocode.




```
function BST_Search(BST T, KeyType key)
    if T == NIL then
        return Not Found
    else if key == T -> key then
        return T
    else if key < T -> key then
        return BST_Search(T->left, key)
    else
        return BST_Search(T->right, key)
    end if
end function
```