

Data Structures and Algorithms

# **BINARY SEARCH TREES**

Nguyễn Ngọc Thảo  
nnthao@fit.hcmus.edu.vn

Ho Chi Minh City, 05/2022

# Outline

---

- Tree terminology
- Binary tree
- Binary search tree
- Implement a Binary search tree

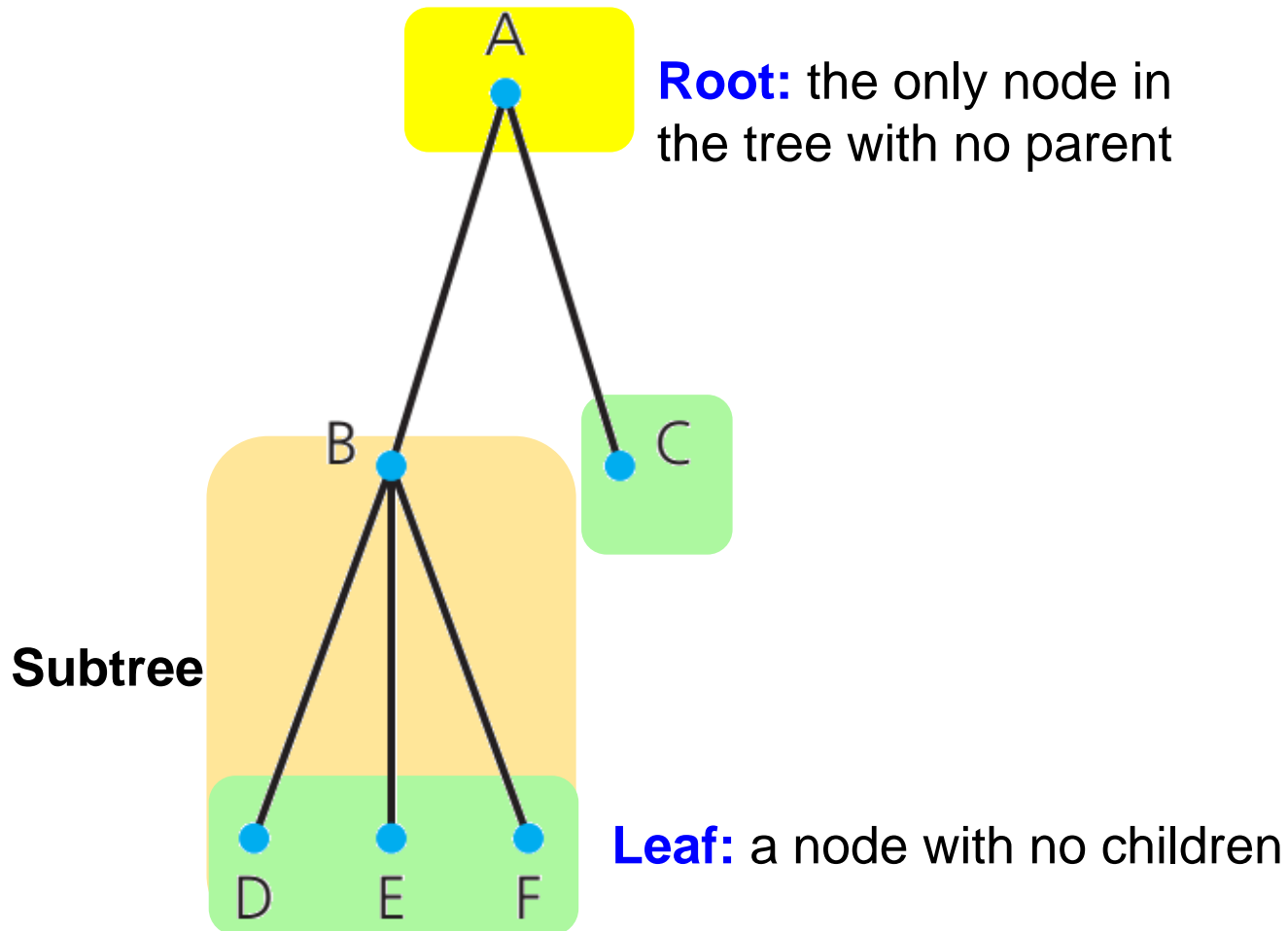
# Tree terminology



- *Kind of trees*
- *The height of trees*
- *Full, Complete, and Balanced binary trees*
- *The maximum and minimum heights of a binary tree*

# General tree and its components

- A **general tree** is a set of **one or more nodes**, partitioned into a **root node and subsets** that are general subtrees of the root

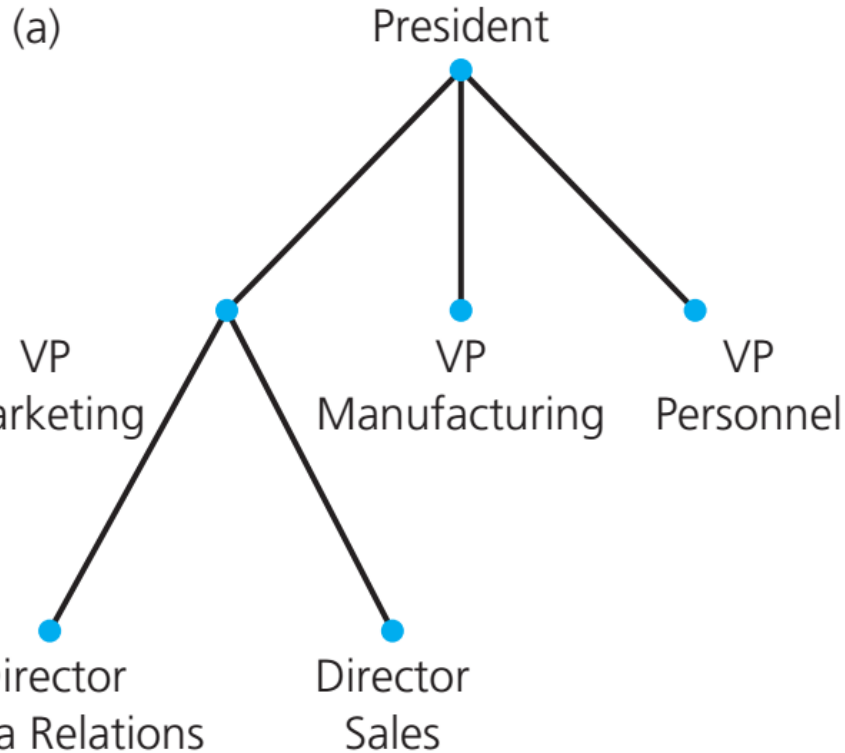


# The parent-child relationship

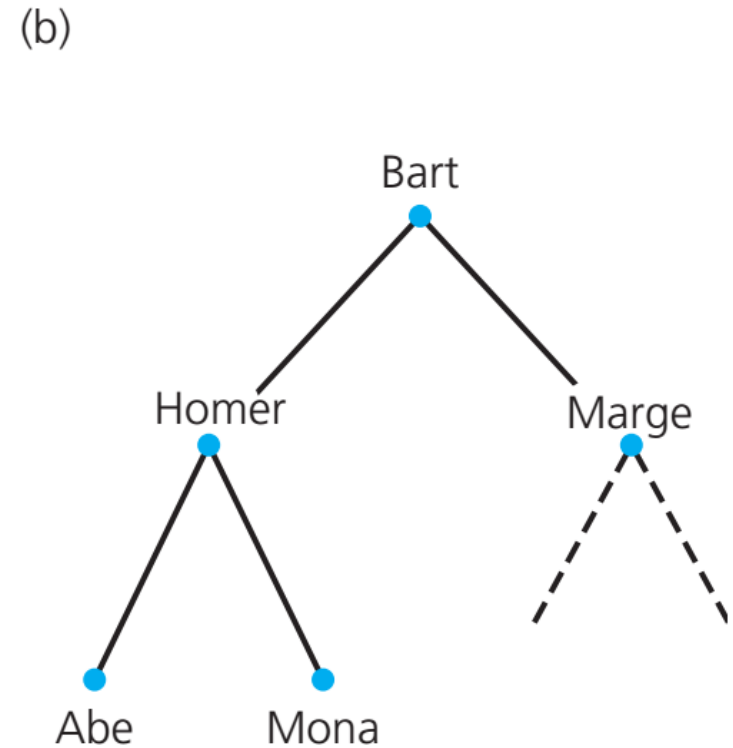
---

- All trees are **hierarchical** in nature.
  - There exists a “parent-child” relationship between nodes in the tree.
- $n$  is the **parent** of  $m \Leftrightarrow m$  is a **child** of  $n$ 
  - An edge is between  $n$  and  $m$  and  $n$  is above  $m$  and in the tree.
  - E.g., nodes B and C are children of node A
- Each node in a tree has at most one parent, **except the root**.
- Each leaf has no children.
  - E.g., nodes C, D, E, and F are leaves.
- An instance of the relationship: **ancestor** and **descendant**.
- **Siblings** are children of the same parent.
  - E.g., nodes B and C of node A; nodes D, E, and F of node B

# Represent hierarchical information

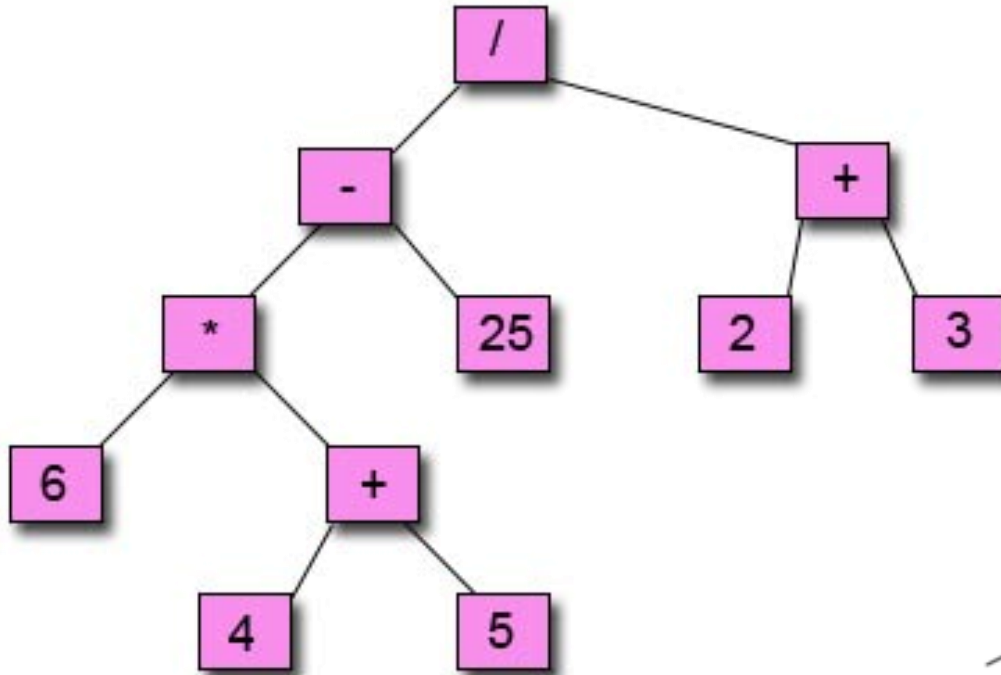


An organization chart

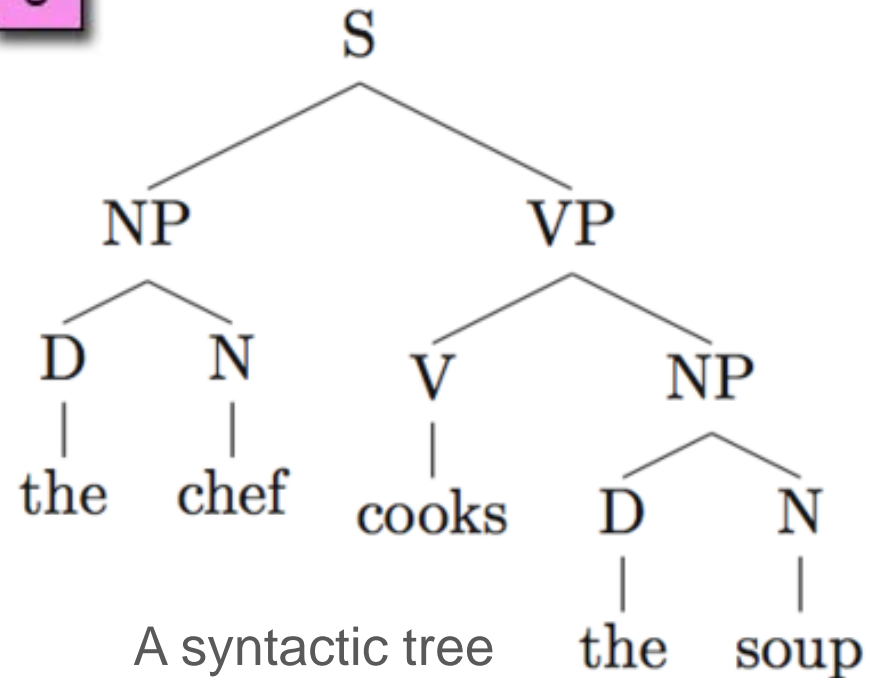


A family tree

# Represent hierarchical information



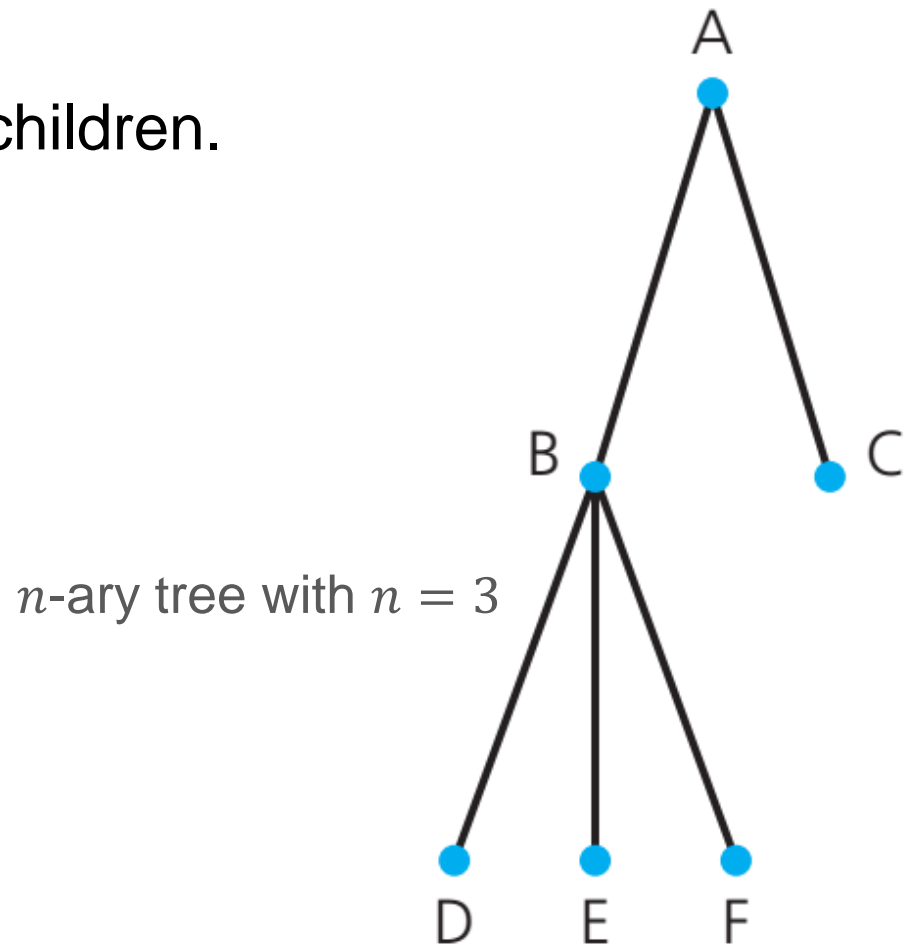
Algebraic expressions



A syntactic tree

# $n$ -ary trees

- A  **$n$ -ary tree** is a set of nodes that is either empty or partitioned into a root node and at most  $n$  subsets that are  $n$ -ary subtrees of the root.
- Each node has at most  $n$  children.

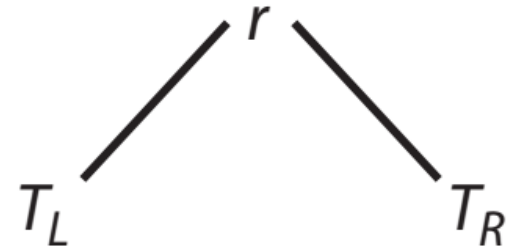




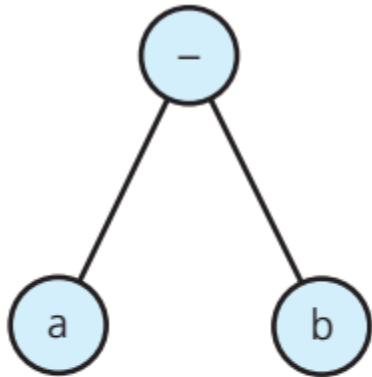
# Binary trees

---

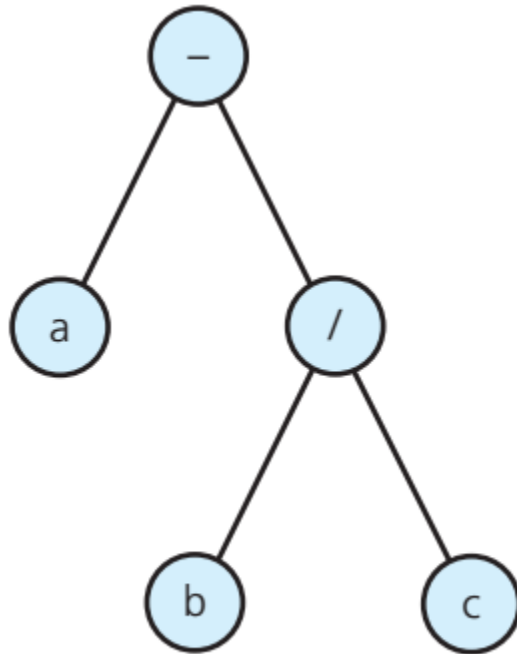
- A **binary tree** is a set of nodes that is either empty or partitioned into a root node and one or two subsets that are binary subtrees of the root.
- This is a  $n$ -ary tree with  $n = 2$ .
- Each node has at most two children, the left child and the right child.



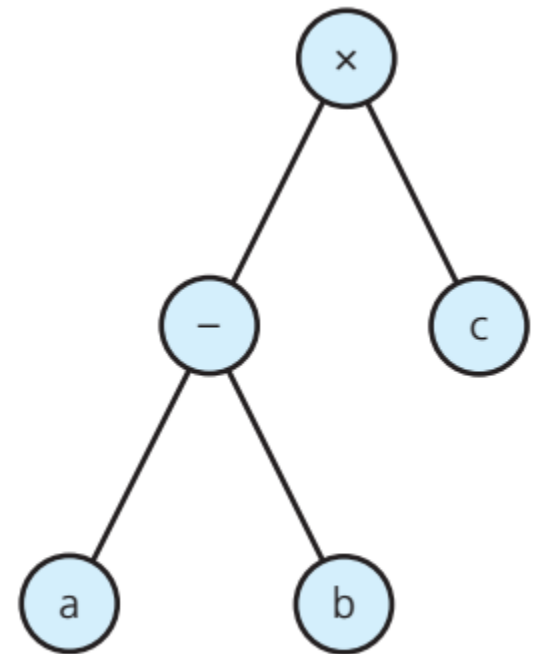
# Binary trees: Applications



(a)



(b)



(c)

Binary trees that represent algebraic expressions.

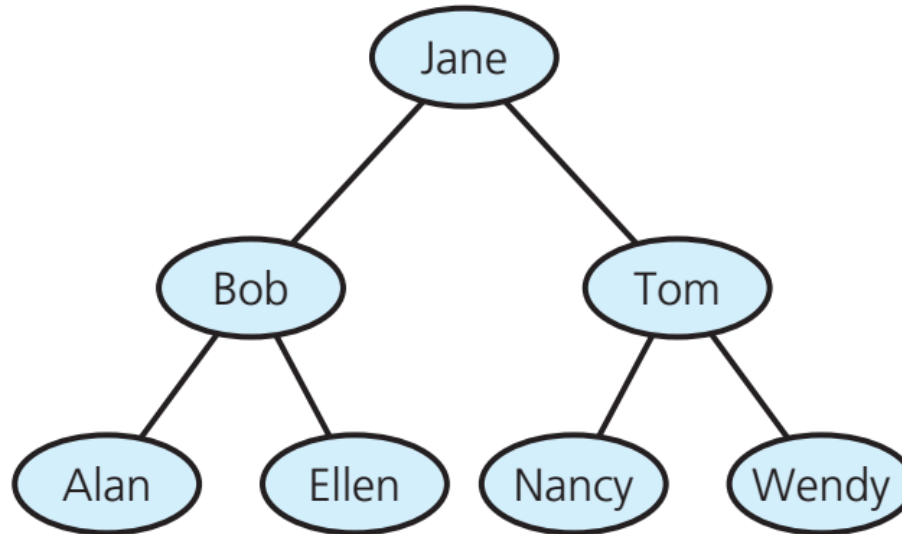
Leaves contain the expression's operands; other nodes contain the operators.

Unambiguous order: Operators lower in the tree are evaluated first.

# Binary search trees (BST)

---

- A **binary search tree** is a binary tree in which the value in any node  $n$  is **greater than** the value in every node in  $n$ 's **left subtree** but **less than** that in every node in  $n$ 's **right subtree**.
- Data is organized in a way that facilitates searching.



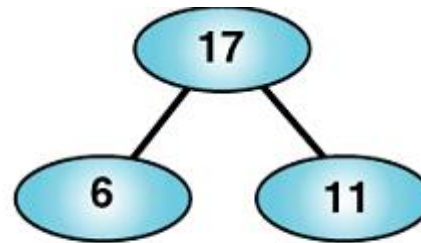
A binary search tree of names

# Checkpoint 01a: Trees and Binary Search Trees

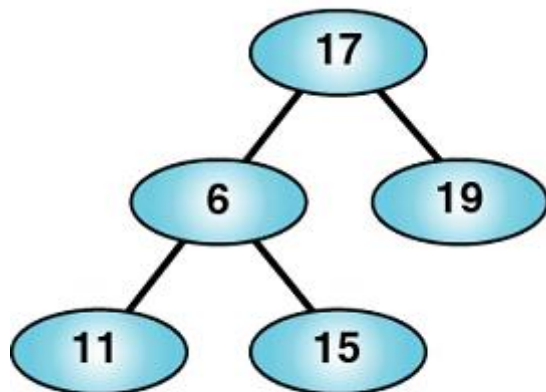
Which of the following trees are binary search trees?



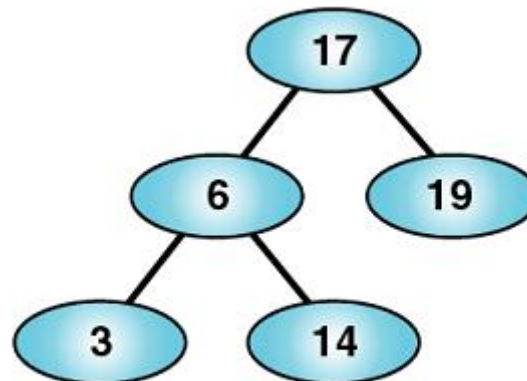
(a)



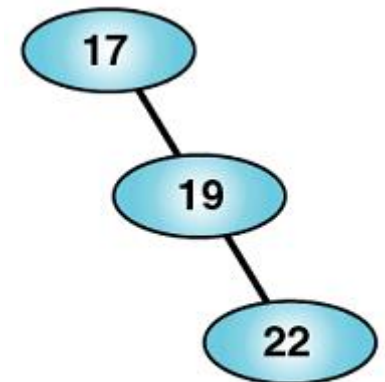
(b)



(c)



(d)

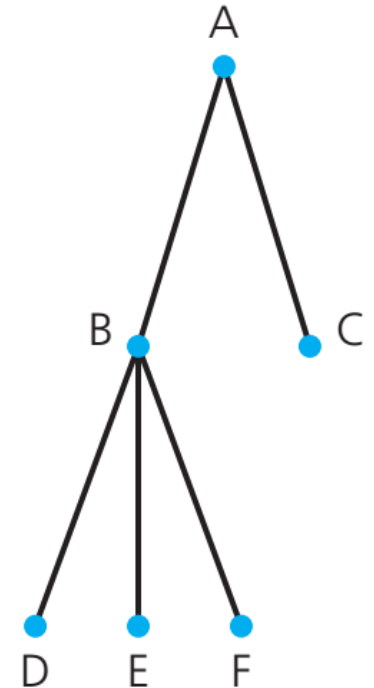
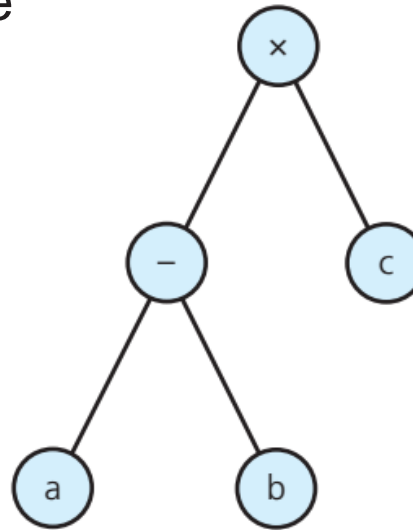
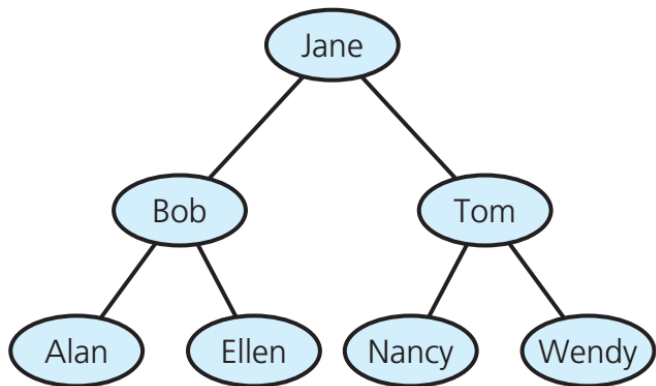


(e)

# Checkpoint 01b: Trees and their components

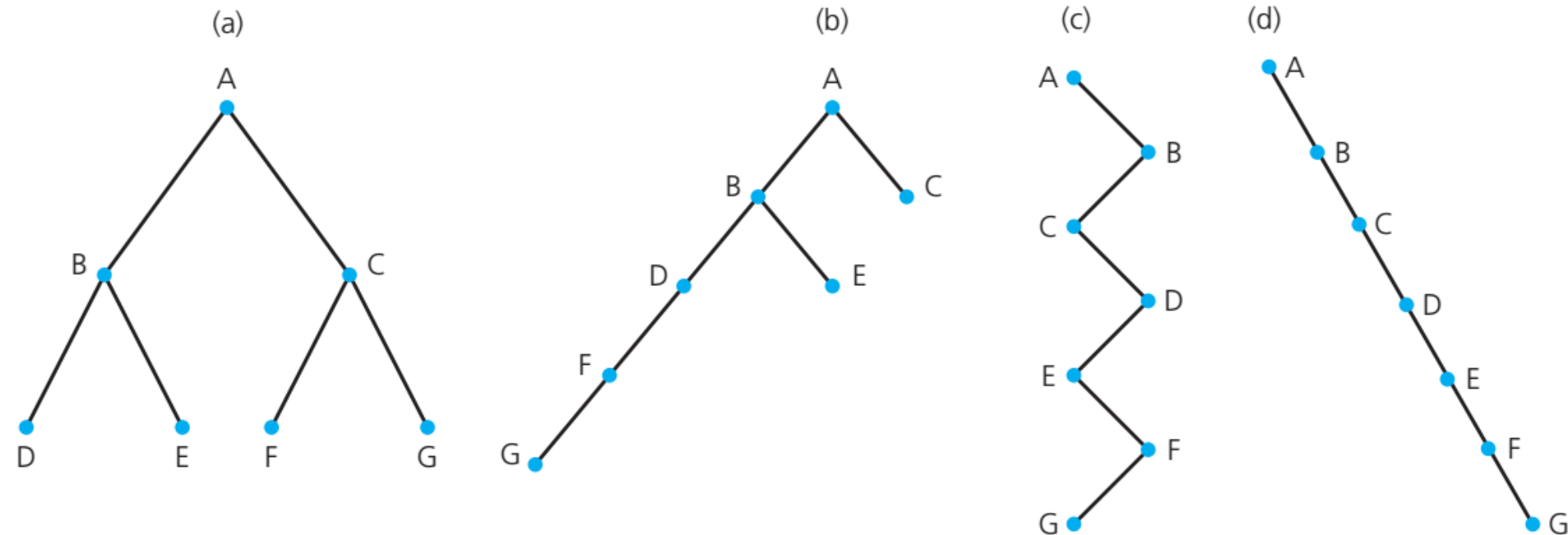
For each of the following trees, answer the questions.

- What kind of tree is the tree?
- List all pairs of parent – child
- List all pairs of siblings
- The root and leaves of the tree



# The height of trees

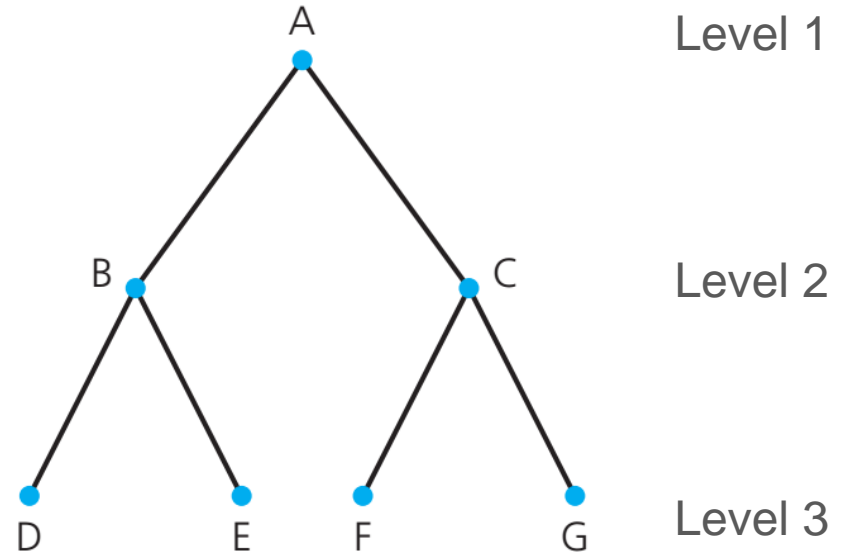
- The **height of a tree** is the number of nodes on the longest path from the root to a leaf.



Binary trees with the same nodes but different heights, 3, 5, 7, and 7

# The height of trees

- The **level of a node  $n$** :
  - If  $n$  is the root of the tree  $T$ , it is at level 1.
  - If not, its level is 1 greater than the level of its parent.

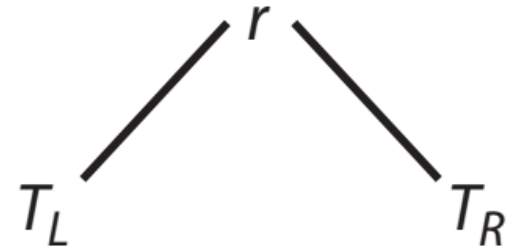


- The **height of a tree  $T$**  can be defined in terms of the levels of its nodes.
  - If  $T$  is empty, its height is 0.
  - If not, its height is equal to the **maximum level** of its nodes.

# The height of binary trees

---

- The **height of a binary tree  $T$**  uses an equivalent recursive definition of height.
  - If  $T$  is empty, its height is 0.
  - If not,  $height(T) = 1 + \max\{height(T_L), height(T_R)\}$





## Checkpoint 02: The height of trees

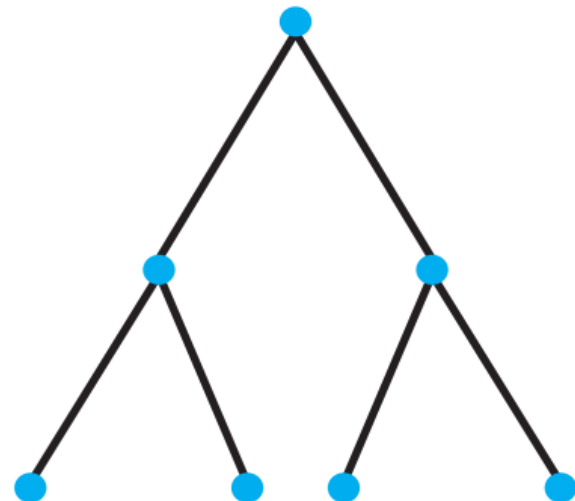
Draw all binary trees having 1 node, 2 nodes and 3 nodes.

# Full binary trees

---

- A **full binary tree** is a binary tree of height  $h$  with no missing nodes, in which all **leaves are at level  $h$**  and other nodes **each have two children**.
- A recursive definition of a full binary tree
  - If  $T$  is empty,  $T$  is a full binary tree of height 0.
  - If not (height  $h > 0$ ),  $T$  is a full binary tree if its root's subtrees are both full binary trees of height  $h - 1$ .

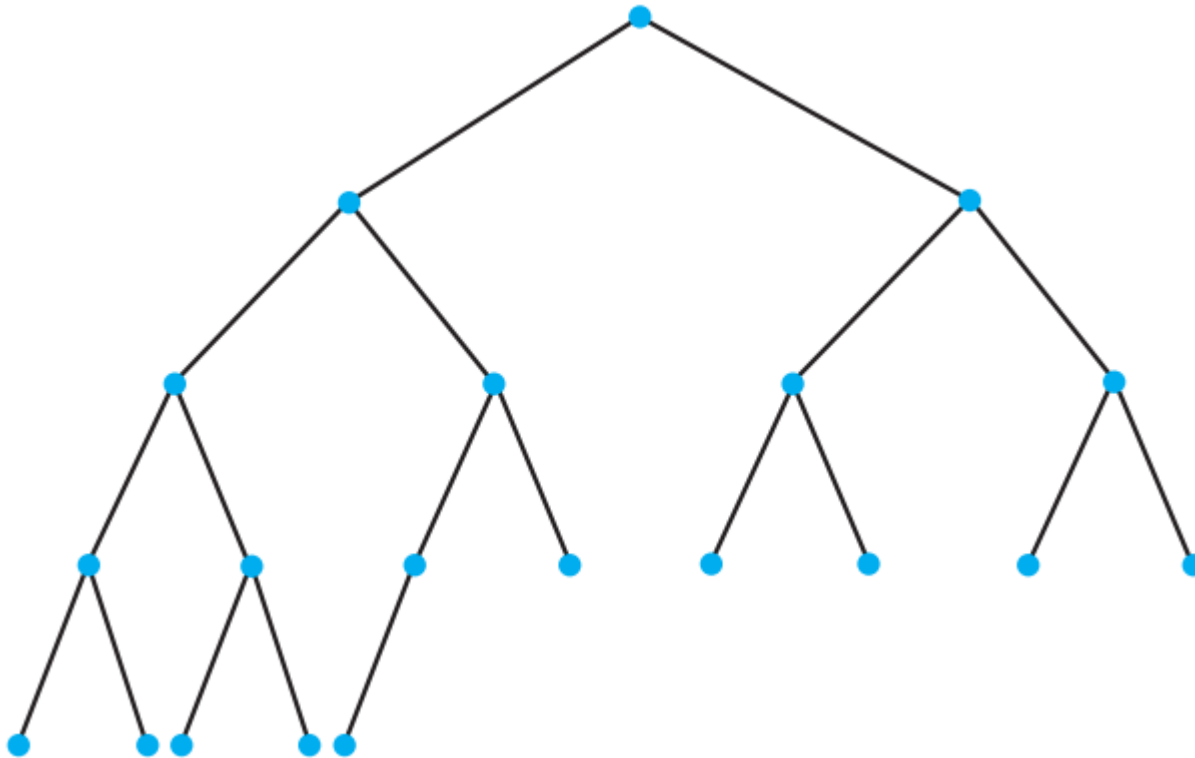
A full binary tree of height 3



# Complete binary trees

---

- A **complete binary tree** is a binary tree of height  $h$  that is full to level  $h - 1$  and has level  $h$  filled in from left to right.



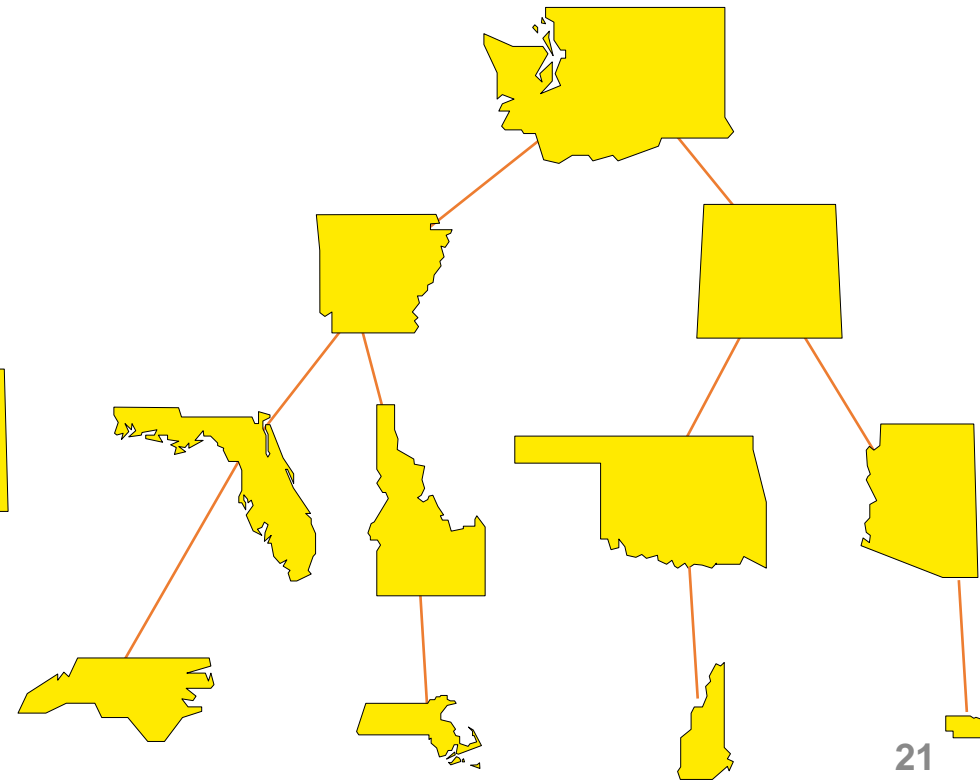
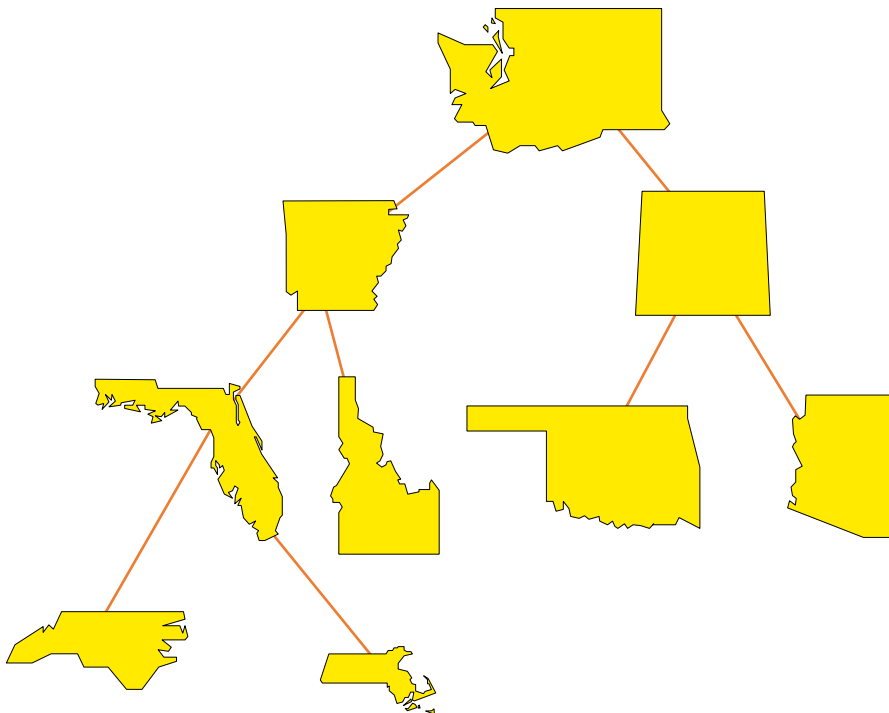
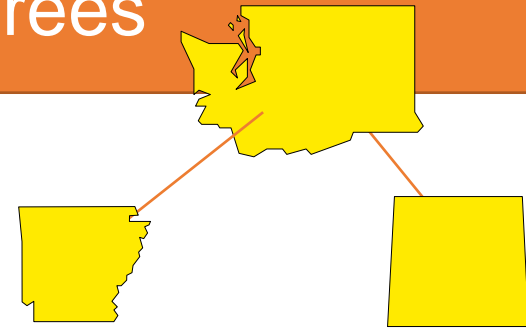
# Complete binary trees

---

- A **binary tree**  $T$  of height  $h$  is **complete** if
  1. All nodes at level  $h - 2$  and above have two children each, and
  2. When a node at level  $h - 1$  has children, all nodes to its left at the same level have two children each, and
  3. When a node at level  $h - 1$  has one child, it is a left child

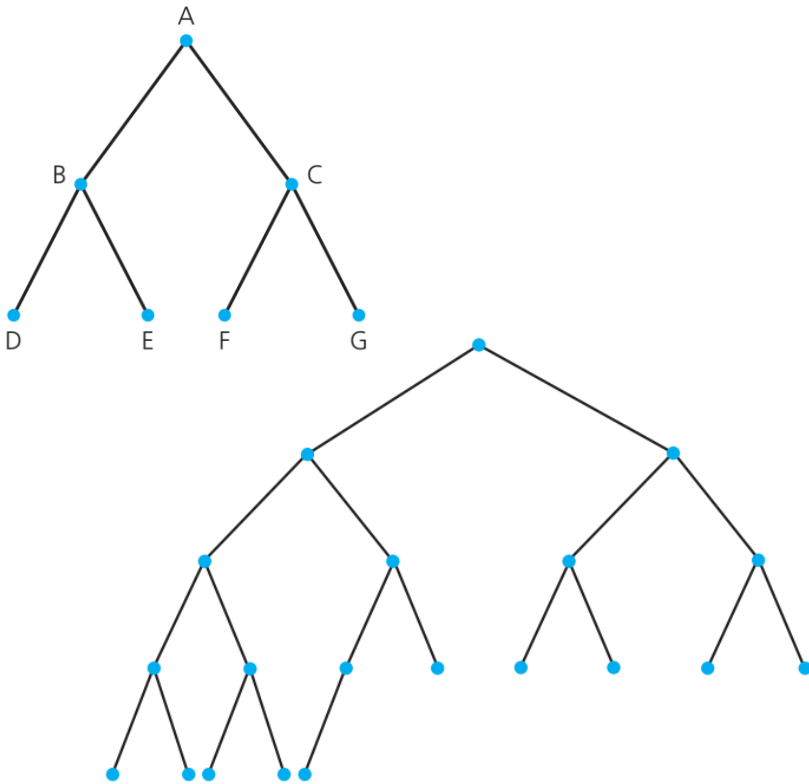
## Checkpoint 03: Full / Complete binary trees

Which of the following trees are full binary tree? Complete binary tree?

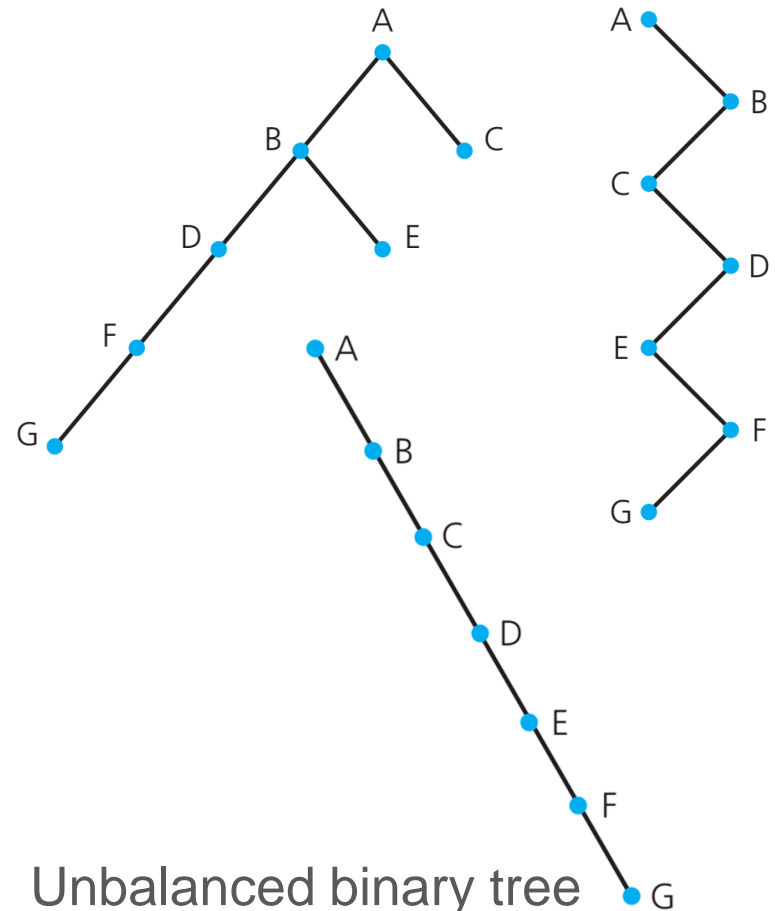


# Balanced binary trees

- A **balanced binary tree** is a binary tree in which the **left and right subtrees** of any node have **heights that differ at most 1**.



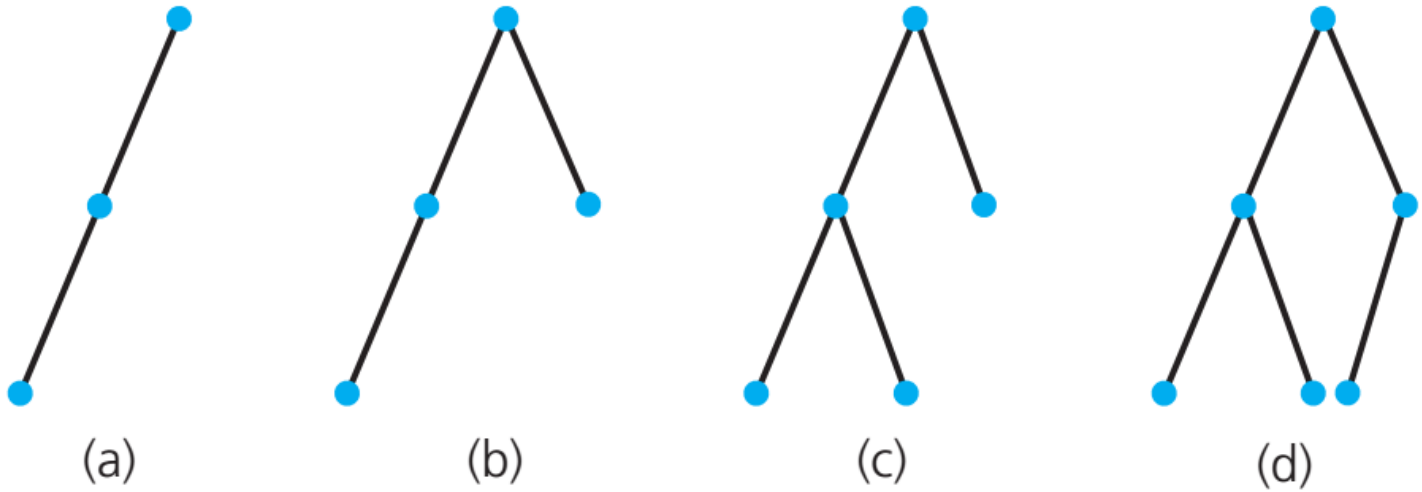
## Balanced binary tree



## Unbalanced binary tree G

# Maximum and Minimum heights

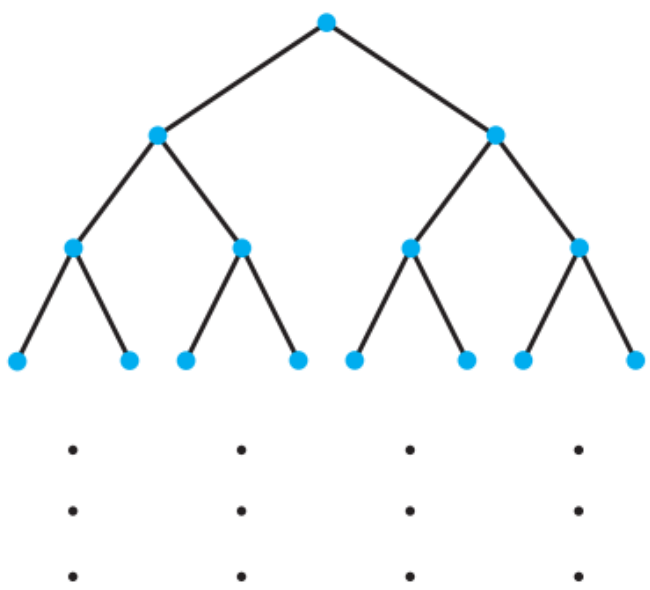
- The **maximum height** of a  $n$ -node binary tree is  $n$ .
  - This resembles a chain of linked nodes.
- The **minimum height** of a  $n$ -node binary tree is  $\lceil \log_2(n + 1) \rceil$ .
  - Complete trees and full trees with  $n$  nodes have heights of  $\lceil \log_2(n + 1) \rceil$ , which, is the theoretical minimum.



Binary trees of height 3

# Maximum and Minimum heights

- The **maximum number of nodes** that a binary tree of height  $h$  can have is  $2^h - 1$ .

|  | Level | Number of nodes<br>at this level | Total number of nodes at this<br>level and all previous levels |
|--|-------|----------------------------------|--|
|  | 1     | $1 = 2^0$                        | $1 = 2^1 - 1$  |
|  | 2     | $2 = 2^1$                        | $3 = 2^2 - 1$  |
|  | 3     | $4 = 2^2$                        | $7 = 2^3 - 1$  |
|  | 4     | $8 = 2^3$                        | $15 = 2^4 - 1$   |
|  | $h$   | $2^{h-1}$                        | $2^h - 1$  |

Counting the nodes in a full binary tree of height  $h$



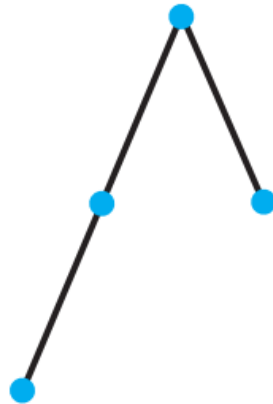
## Checkpoint 04: The properties of binary trees

For each of the following binary trees, answer the questions

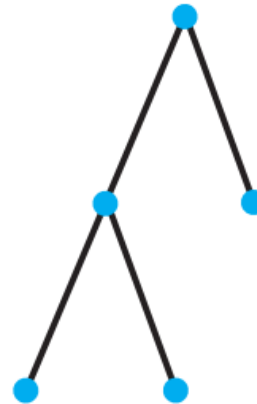
- Which are complete?
- Which are full?
- Which are balanced?
- Which have minimum height?
- Which have maximum height?



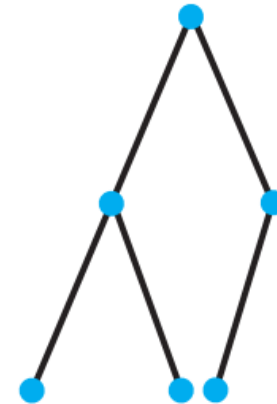
(a)



(b)

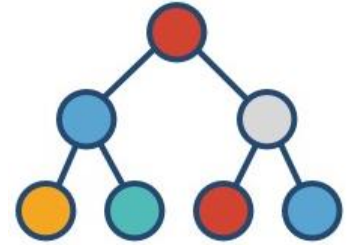


(c)



(d)

# ADT Binary tree



- *Traversals of a binary tree*
- *Binary tree operations*

# Traversals of a binary tree

- A traversal algorithm for a binary tree visits each node in the tree.
- Every node in the tree is visited exactly once.
- It can visit the root  $r$ 
  - **Pre-order traversal:** Before it traverses both of  $r$ 's subtrees
  - **In-order traversal:** After it has traversed  $r$ 's left subtree  $T_L$  but before it traverses  $r$ 's right subtree  $T_R$
  - **Post-order traversal:** After it has traversed both of  $r$ 's subtrees

```
if (T is not empty)
{
    Display the data in T's root
    Traverse T's left subtree
    Traverse T's right subtree
}
```

# Pre-order traversal

*// Traverses the given binary tree in pre-order .*

*// Assumes that “visit a node” means to process the node’s data item*

```
preorder(binTree: BinaryTree): void
```

```
if (binTree is not empty)
```

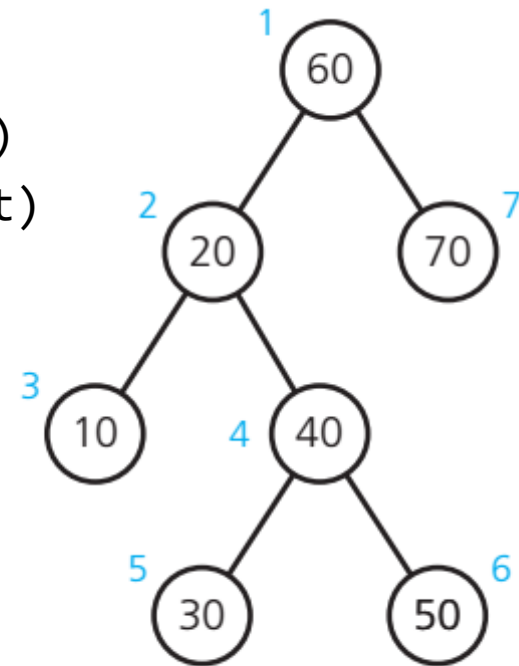
```
{
```

```
    Visit the root of binTree
```

```
    preorder(Left subtree of binTree’s root)
```

```
    preorder(Right subtree of binTree’s root)
```

```
}
```



(a) Preorder: 60, 20, 10, 40, 30, 50, 70

# In-order traversal

*// Traverses the given binary tree in in-order .*

*// Assumes that “visit a node” means to process the node’s data item*

```
inorder(binTree: BinaryTree): void
```

```
if (binTree is not empty)
```

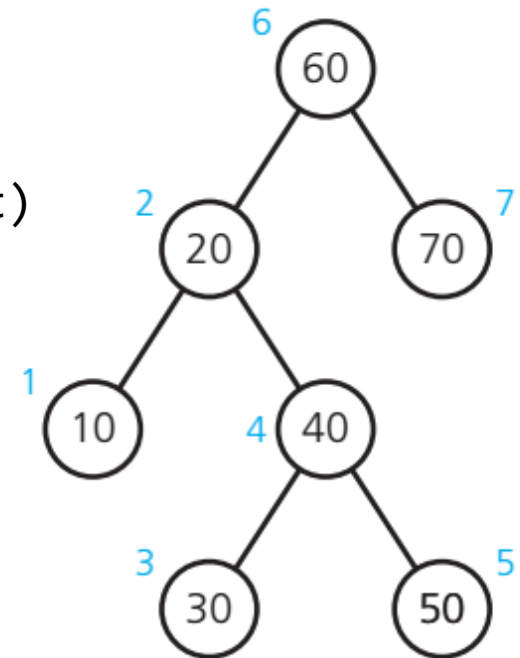
```
{
```

```
    inorder(Left subtree of binTree’s root)
```

```
    Visit the root of binTree
```

```
    inorder(Right subtree of binTree’s root)
```

```
}
```



(b) Inorder: 10, 20, 30, 40, 50, 60, 70

# Post-order traversal

*// Traverses the given binary tree in post-order*

*// Assumes that “visit a node” means to process the node’s data item*

**postorder**(binTree: BinaryTree): void

**if** (binTree is not empty)

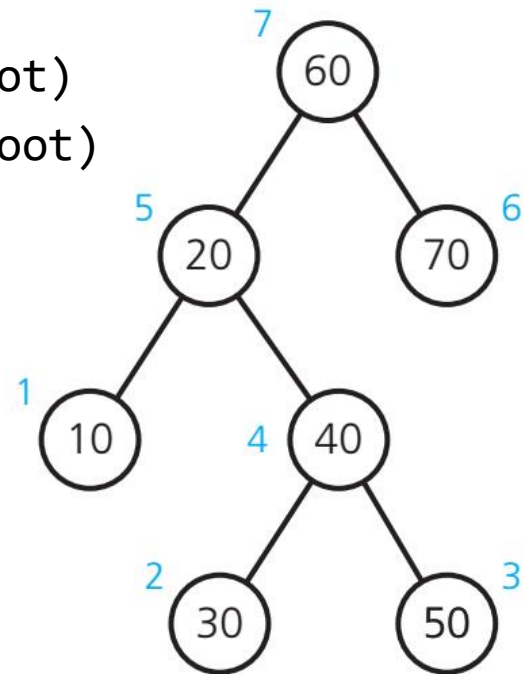
{

**postorder**(Left subtree of binTree’s root)

**postorder**(Right subtree of binTree’s root)

    Visit the root of binTree

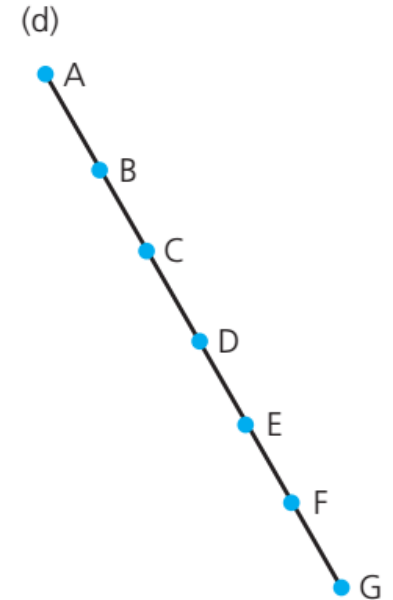
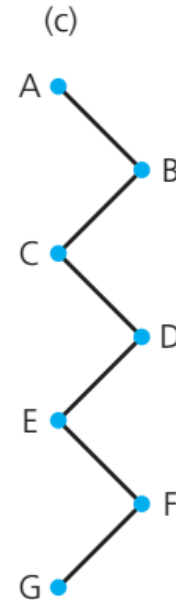
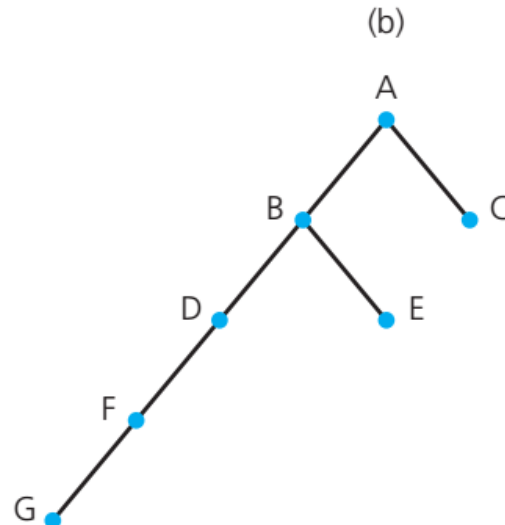
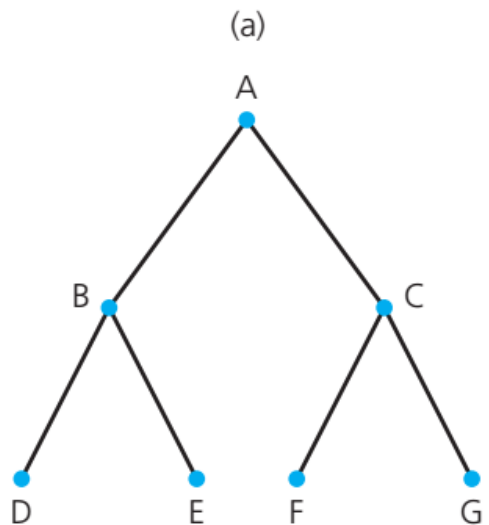
}



(c) Postorder: 10, 30, 50, 40, 20, 70, 60

## Checkpoint 05: Traversals of a binary tree

What are the pre-order, in-order, and post-order traversals of the binary trees shown below?



# ADT Binary tree operations

## BinaryTree

```
+isEmpty(): boolean  
+getHeight(): integer  
+getNumberOfNodes(): integer  
+getRootData(): ItemType  
+setRootData(newData: ItemType): void  
+add(newData: ItemType): boolean  
+remove(data: ItemType): boolean  
+clear(): void  
+getEntry(anEntry: ItemType): ItemType  
+contains(data: ItemType): boolean  
+preorderTraverse(visit(item: ItemType): void): void  
+inorderTraverse(visit(item: ItemType): void): void  
+postorderTraverse(visit(item: ItemType): void): void
```

UML diagram for the class BinaryTree

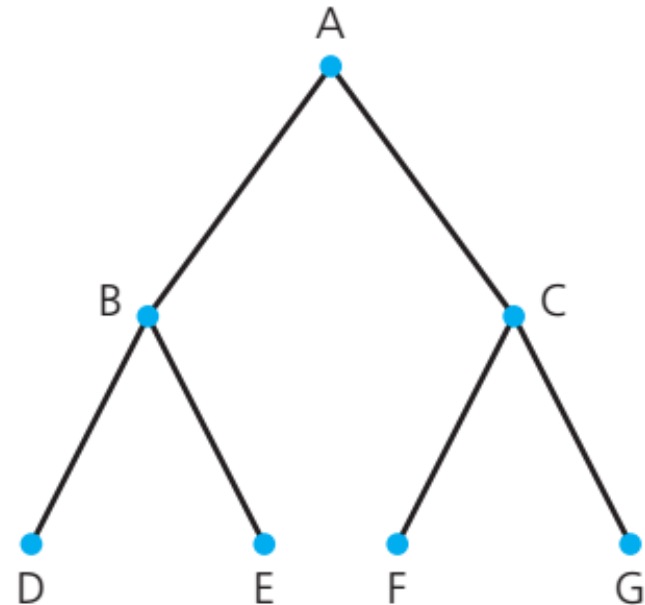


# ADT Binary tree operations

---

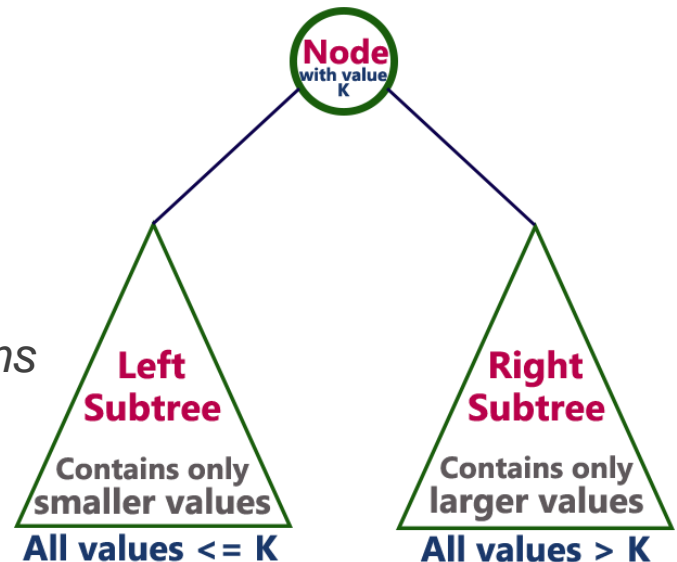
- For example, use the **add** operation to build the binary tree, where the node labels represent character data

```
tree = a new empty binary tree  
tree.add('A')  
tree.add('B')  
tree.add('C')  
tree.add('D')  
tree.add('E')  
tree.add('F')  
tree.add('G')
```



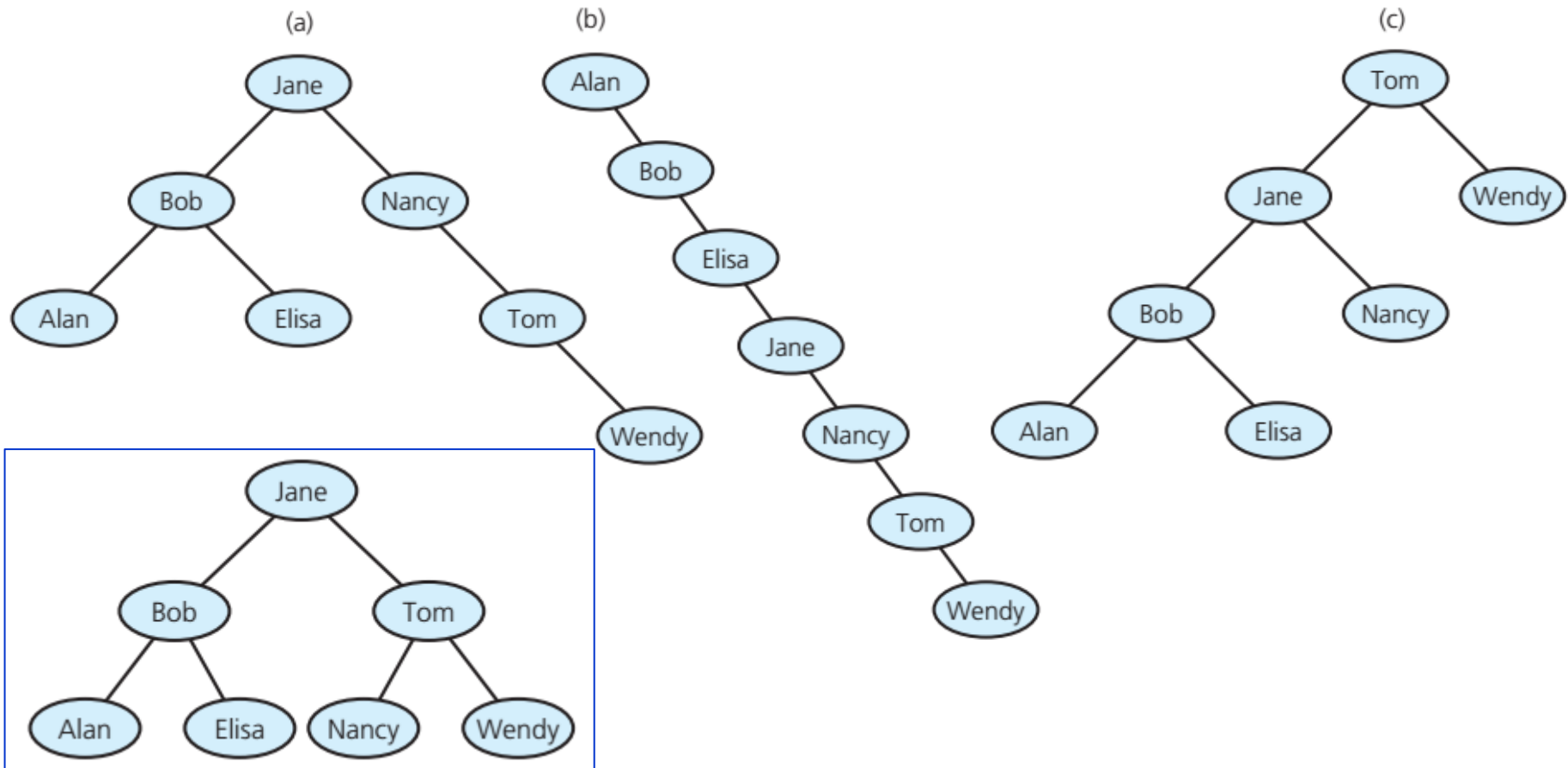
# ADT Binary search tree

- *Binary Search Tree operations*
- *Searching a Binary Search Tree*
- *Traversals of a Binary Search Tree*
- *Insertion/Removal on a Binary Search Tree*
- *The Efficiency of Binary Search Tree operations*



# Binary search tree

- Searching for a particular item is one operation for which the binary tree is ill suited → corrected by a **binary search tree**.



# ADT Binary search tree operations

---

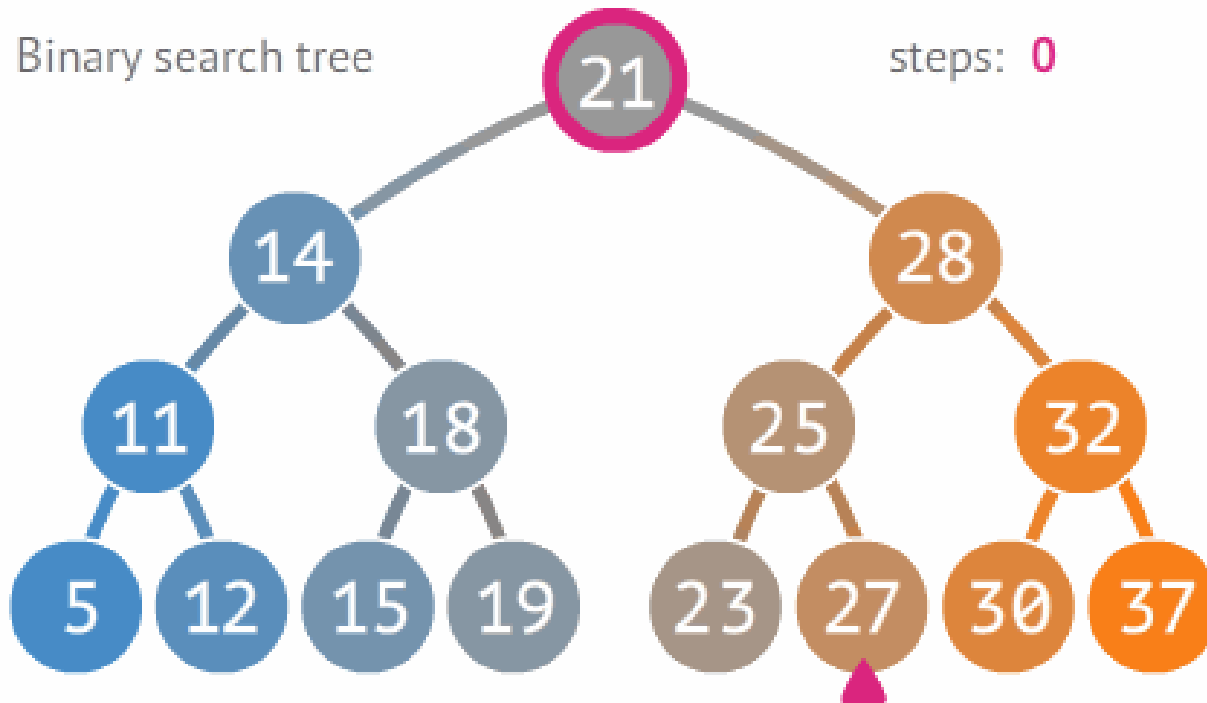
- Most of the methods have the same specifications as for a binary tree.
  - isEmpty, getHeight, getNumberOfNodes, getRootData, clear, getEntry, contains, preorderTraverse, inorderTraverse, and postorderTraverse
- We must write the **add** and **remove** operations such that the properties of a binary search tree are maintained.

# Searching a binary search tree

- The shape of the tree in no way affects the validity of the search algorithm but the efficiency of its operations.

```
// Search the BST for a given target value
search(bstTree: BinarySearchTree, target: ItemType)
if (bstTree is empty)
    The desired item is not found
else if (target == data item in the root of bstTree)
    The desired item is found
else if (target < data item in the root of bstTree)
    search(Left subtree of bstTree, target)
else
    search(Right subtree of bstTree, target)
```

# Searching a binary search tree

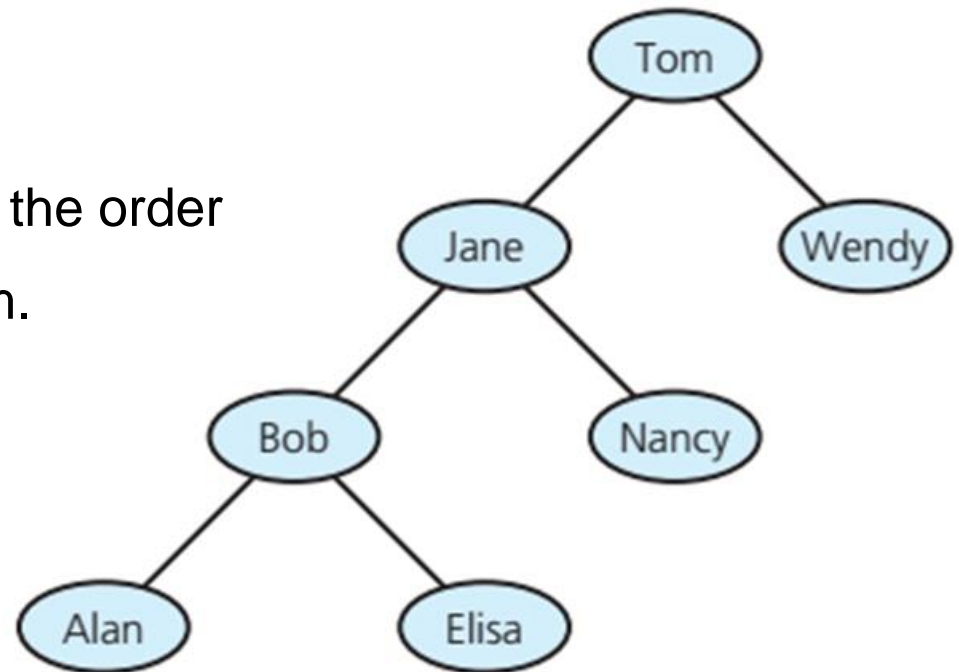


## Checkpoint 06: Search a binary search tree

Trace the algorithm that searches the following binary search tree for

- Elisa
- Kyle

In each case, list the nodes in the order in which the search visits them.



# Traversals of a binary search tree

---

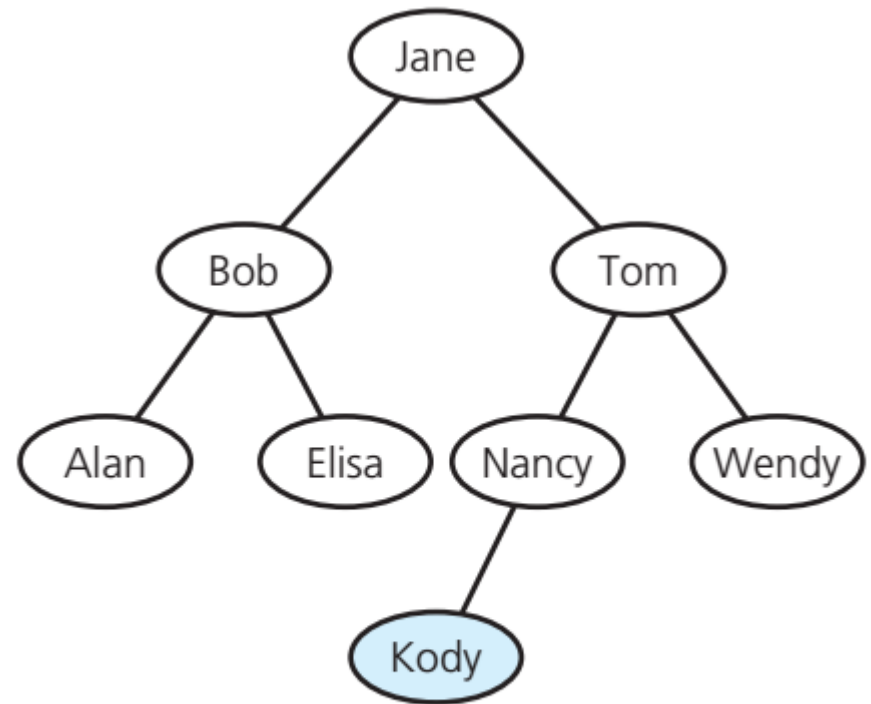
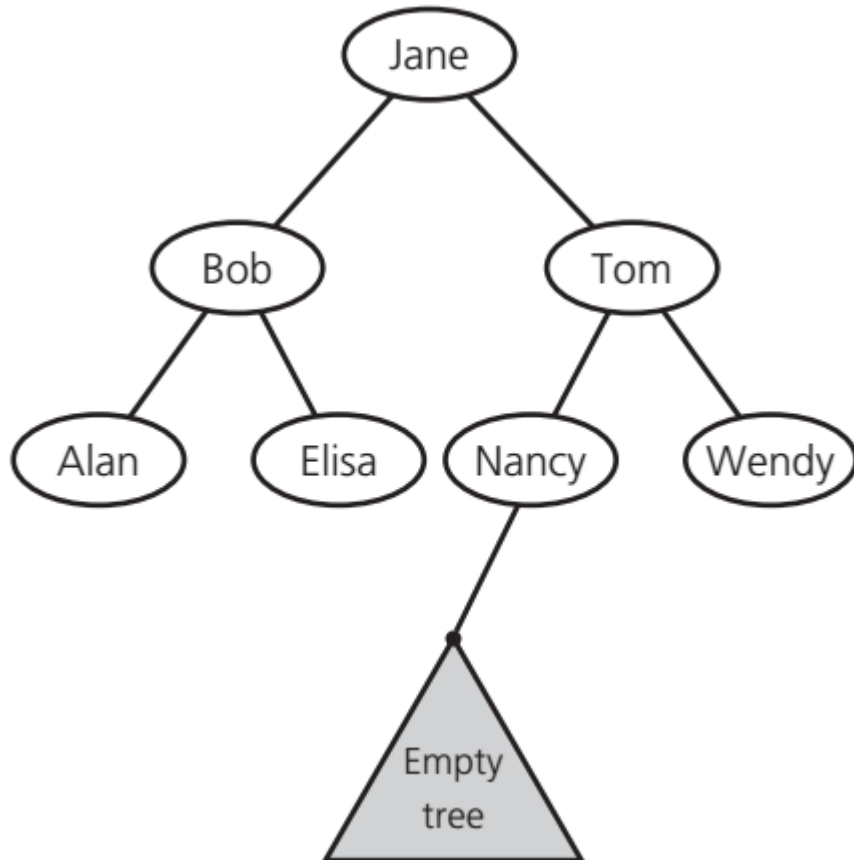
- The **inorder traversal** of a binary search tree visits the tree's nodes in **sorted search-key order**.





# Insertion on a binary search tree

- Use **search** to determine where to insert a new node, as a **new leaf** always



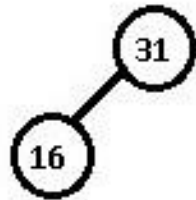
# Insertion on a binary search tree

---

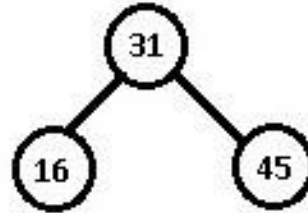
# Insertion on a BST: An example



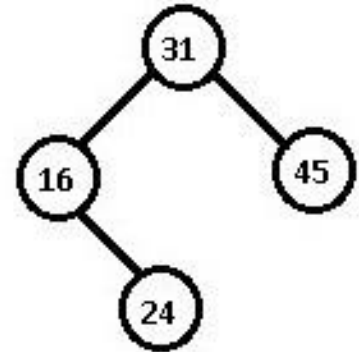
**Insert 31**



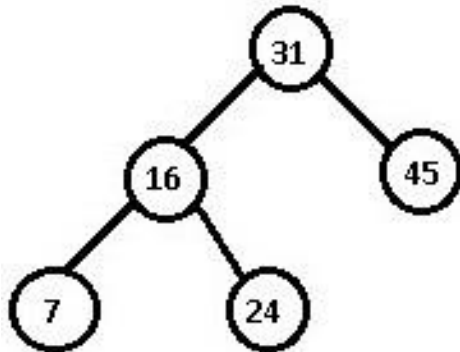
**Insert 16**



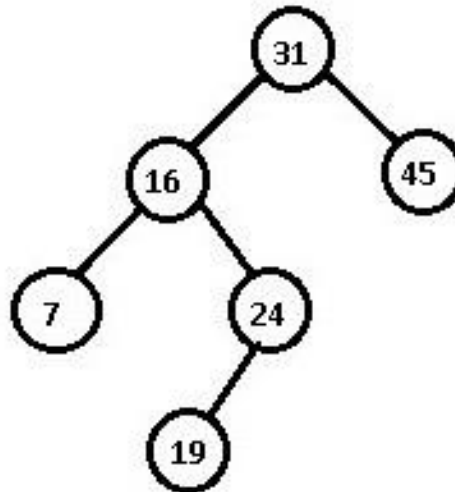
**Insert 45**



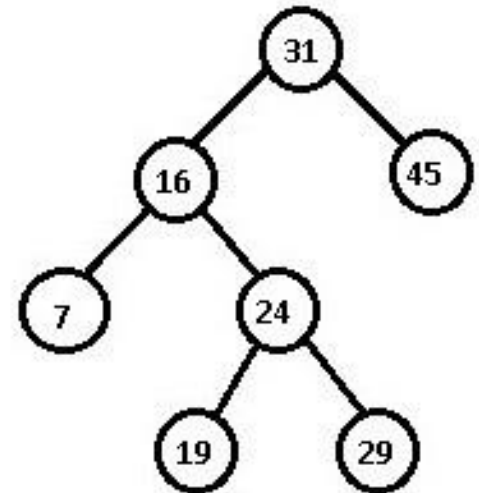
**Insert 24**



**Insert 7**



**Insert 19**



**Insert 29**

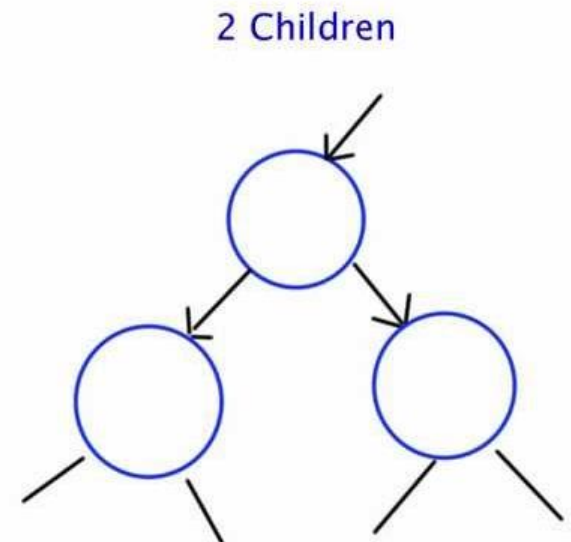
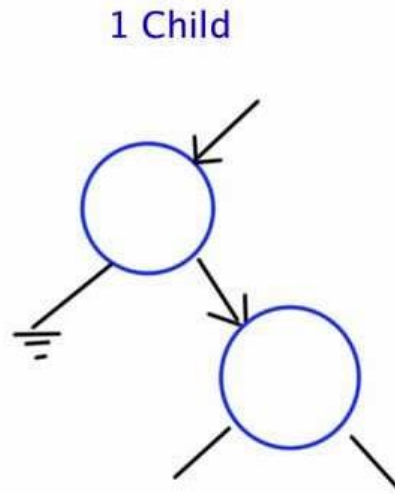
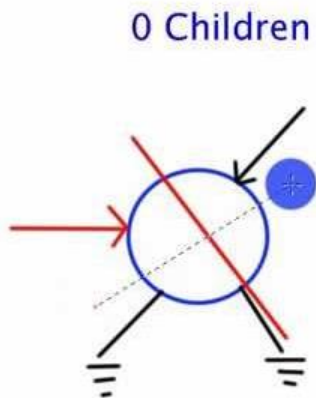
## Checkpoint 07: Create a binary search tree

Beginning with an empty binary search tree, what binary search tree is formed when you insert the following letters in the order given? J, N, B, A, W, E, T

Arrange nodes that contain the letters A, C, E, F, L, V, and Z into two binary search trees: one that has maximum height and one that has minimum height.

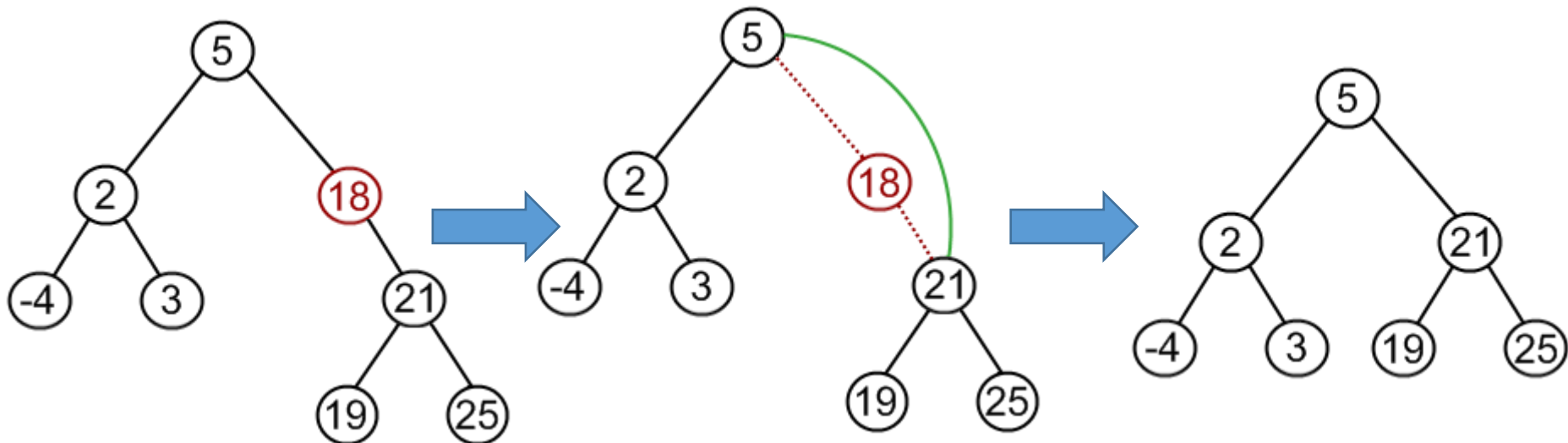
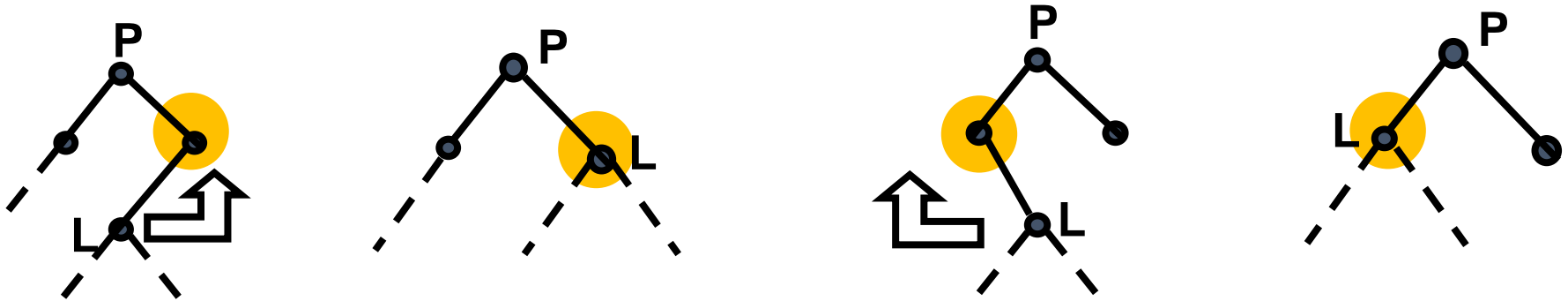
# Removal on a binary search tree

- Use **search** to determine where in the tree located the node to be removed
- The node being removed may have **no child** or **a single child** (left child or right child) or **both two children**.



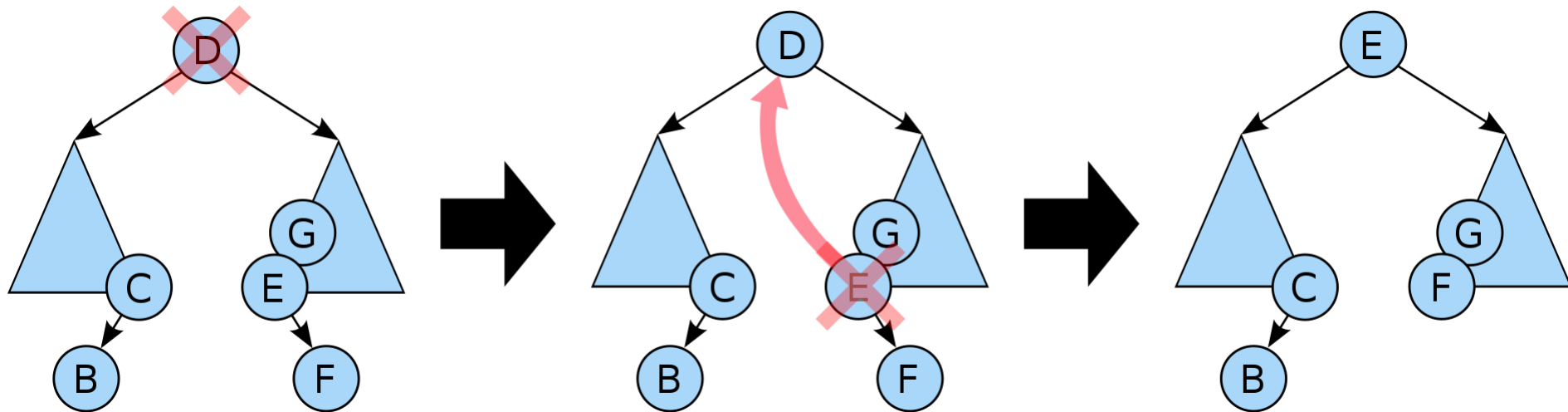
# Removal a node with single child

- The node to be removed has **either a left child or a right child**.



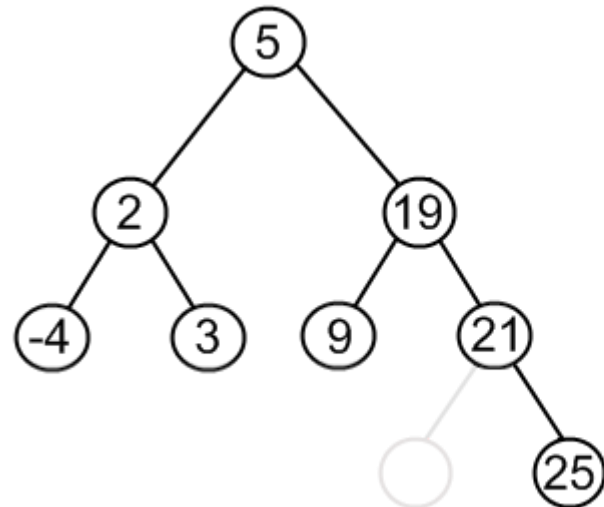
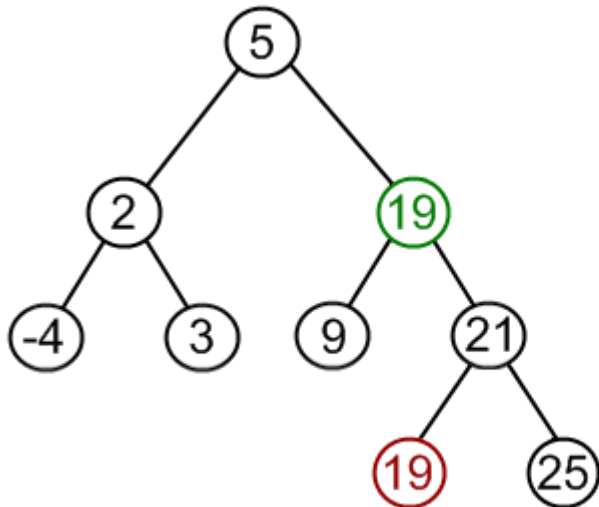
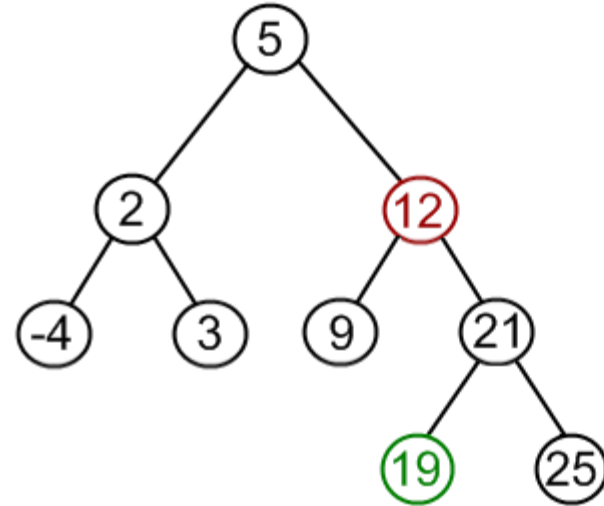
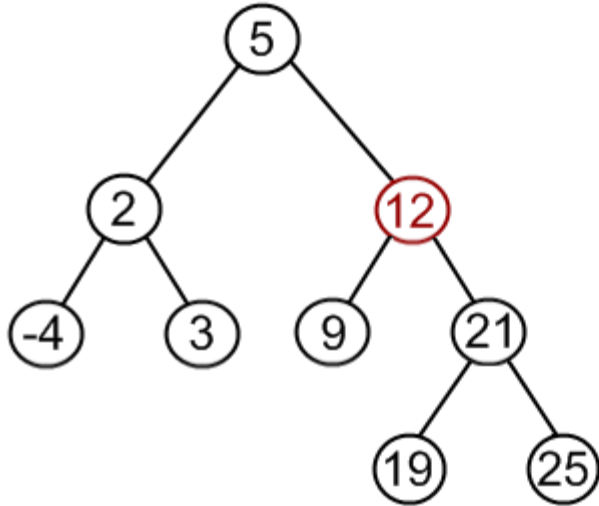
# Removal on a binary search tree

- The node to be removed has both left child and right child.
  - Find the **leftmost node** in the **right subtree**
  - Copy the found node's value to the node being deleted
  - Delete the found node



- The same method works symmetrically using **rightmost node** in the **left subtree**.

# Remove a node with both children



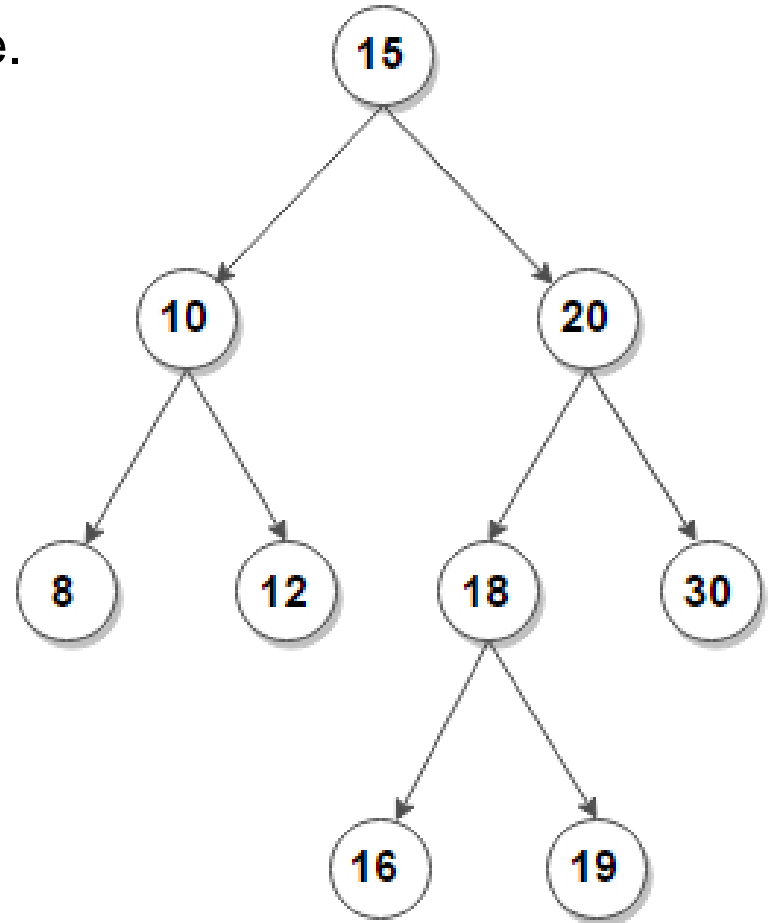


## Checkpoint 08: Delete a node from a BST

Consider the following binary search tree.

What is the resulting tree after deleting

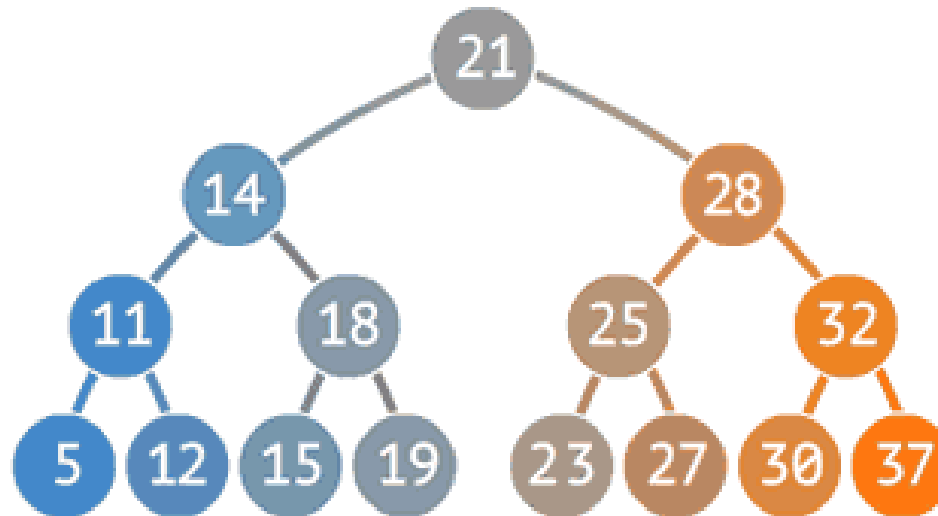
- The node 19
- The node 18
- The node 15



# Binary search tree degradation

---

Binary search tree degradation



# The efficiency of BST operations

- If the height of the BST is  $\lceil \log_2(n + 1) \rceil$ , the efficiency of its operations is  $O(\log n)$ .
  - The height of a BST depends on the order in which you perform **insertion** and **removal** operations on the tree.
- Variations of the basic BST are guaranteed always to remain balanced and therefore be of minimum height.
  - E.g., AVL tree, Red-Black tree, AA tree, etc.

| <u>Operation</u> | <u>Average case</u> | <u>Worst case</u> |
|------------------|---------------------|-------------------|
| Retrieval        | $O(\log n)$         | $O(n)$            |
| Insertion        | $O(\log n)$         | $O(n)$            |
| Removal          | $O(\log n)$         | $O(n)$            |
| Traversal        | $O(n)$              | $O(n)$            |

---

# BST Implementation

---

- *Array-based implementation*
- *Link-based implementation*

# The nodes in a binary tree

---

- The first step in implementing a tree is to choose a data structure to represent its nodes.
- Each node must contain both **data** and **“pointers” to the node’s children**.
  - If we place these nodes in an array, the “pointers” in the nodes are array indices;
  - If the nodes form a linked chain, we use pointers to link them.

# An array-based representation

- An array-based tree implementation uses an array of nodes.

```
TreeNode tree[MAX_NODES];           // Array of tree nodes
int root;                           // Index of root
int free;                           // Index of free list
```

- The variable **root** is an index to the tree's root node within the array tree.
  - If the tree is empty, root is  $-1$ .
- The variable **free** is the index to the first node in the **free list**, which is a collection of available nodes.

# An array-based representation

---

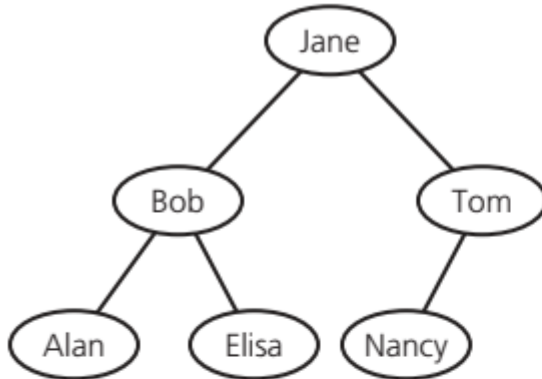
- The class `TreeNode` can be implemented with the following data members.

```
class TreeNode {  
    ItemType item;           // Data portion  
    int leftChild;           // Index to left child  
    int rightChild;          // Index to right child  
}
```

- Each node has a data item and two array indices, one to each child.
  - Both `leftChild` and `rightChild` within a node are indices to the children of that node.
  - If a node has no left child, `leftChild` is -1; similarly for `rightChild`.

# An array-based representation

(a)



(a) A binary tree of names;

(b) Its implementation using the array tree

(b)

| tree |           |            |    | root      |
|------|-----------|------------|----|-----------|
| item | leftChild | rightChild |    |           |
| 0    | Jane      | 1          | 2  | 0         |
| 1    | Bob       | 3          | 4  | free      |
| 2    | Tom       | 5          | -1 | 6         |
| 3    | Alan      | -1         | -1 |           |
| 4    | Elisa     | -1         | -1 |           |
| 5    | Nancy     | -1         | -1 |           |
| 6    | ?         | -1         | 7  | Free list |
| 7    | ?         | -1         | 8  |           |
| 8    | ?         | -1         | 9  |           |
| •    | •         | •          | •  |           |
| •    | •         | •          | •  |           |
| •    | •         | •          | •  |           |



# Managing nodes with the free list

---

- The nodes may not be in contiguous elements of the array.
  - Since the tree changes due to insertions and removals, it is necessary to maintain the variable `freelist`.
- To insert a new node into the tree, you first obtain an available node from the free list.
- To remove a node from the tree, you place it into the free list so that it can be used later.

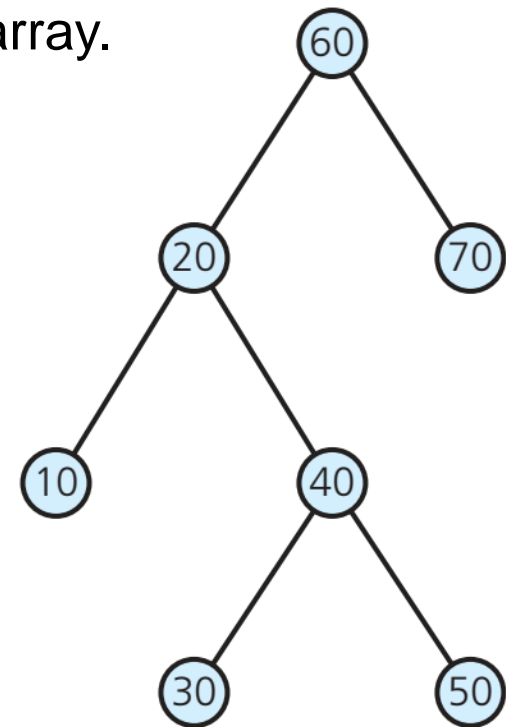
# Managing nodes with the free list

---

- The next available node is not necessarily at index `free+1`.
  - When a node is removed from a tree and returned to the free list, it could be anywhere in the array.
- The available nodes are “linked” together by arbitrarily making the `rightChild` of each node be the index of the next node in the free list.
- While the free list is a linear data structure, the tree is not.

## Checkpoint 09: Represent a BST with an array

Represent the following binary tree with an array.



# A link-based representation

---

- The most common way of implementing a tree.

```
class BinaryNode {  
    ItemType item;           // Data portion  
    BinaryNode* leftChild;   // Pointer to left child  
    BinaryNode* rightChild;  // Pointer to right child  
}
```

- The binary tree is then represented by a single variable **rootPtr**, which points to the tree's root node.
  - If the tree is empty, rootPtr contains nullptr.
  - If either of these subtrees is empty, the pointer to it would be nullptr.

# Acknowledgements

---

- This part of the lecture is adapted from

[1] Frank M. Carrano, Robert Veroff, Paul Helman (2014) “*Data Abstraction and Problem Solving with C++: Walls and Mirrors*” Sixth Edition, Addison-Wesley. **Chapter 15, 16.**

---

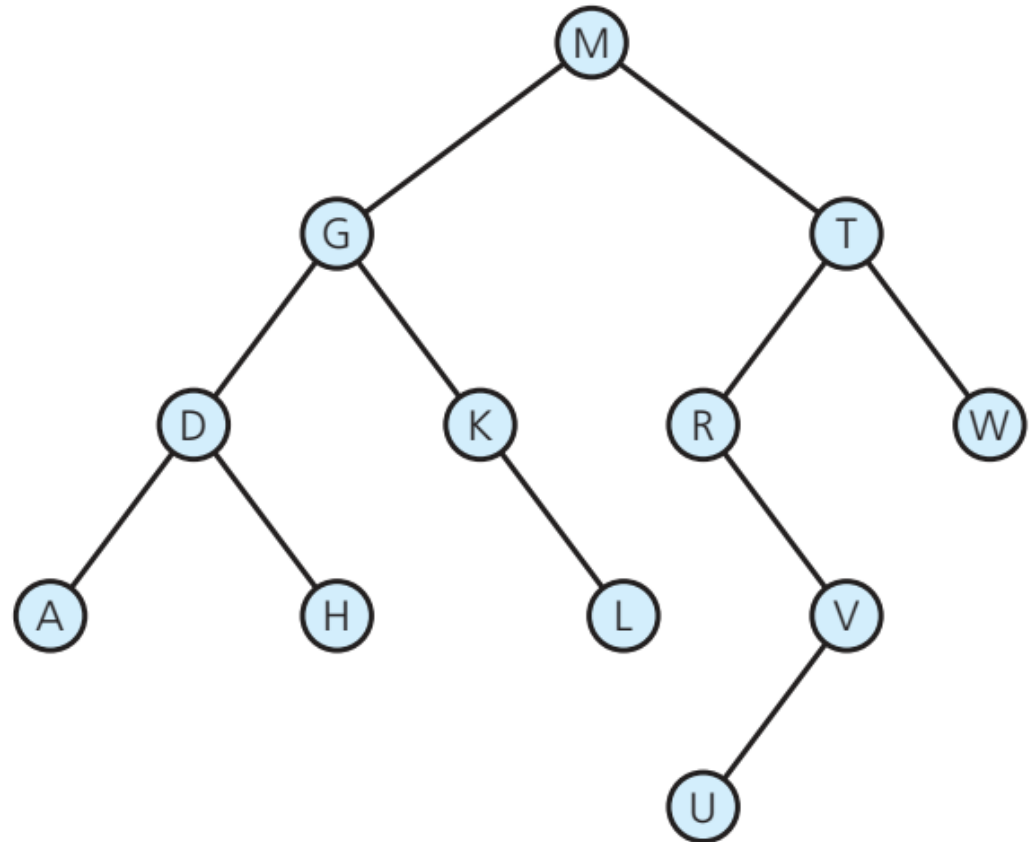
# Exercises

---



# 01. Binary tree traversals

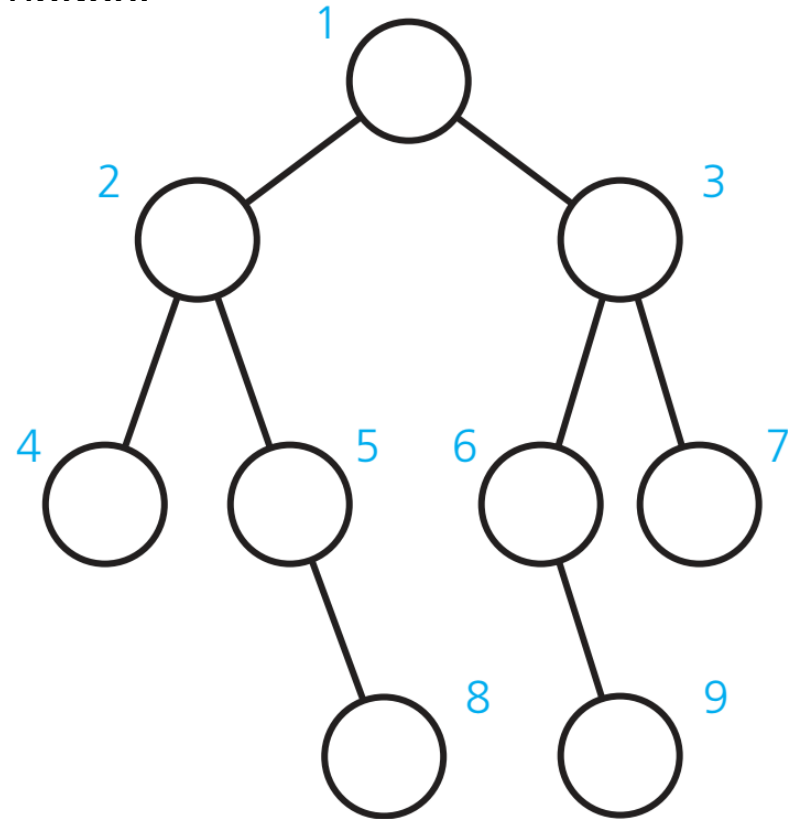
- What are the pre-order, in-order, and post-order traversals of the binary trees shown below?



- Is the given binary tree a binary search tree?

# 02. Binary search tree traversals

- In the below BST, the numbers simply label the nodes so that you can reference them; they do not indicate the contents of the nodes.
  - a) Without performing an inorder traversal, which node must contain the value that comes immediately after the value in the root? Explain.
  - b) In what order will an inorder traversal visit the nodes of this tree? Indicate this order by listing the labels of the nodes in the order that they are visited





# 03. Operations on a BST tree

---

- Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given?
  - a) W, T, N, J, E, B, A
  - b) W, T, N, A, B, E, J
  - c) A, B, W, J, N, T, E
  - d) B, T, E, A, N, W, J
- Choose any of the above binary search trees and repeatedly delete the root node until the tree is empty



**THE END**