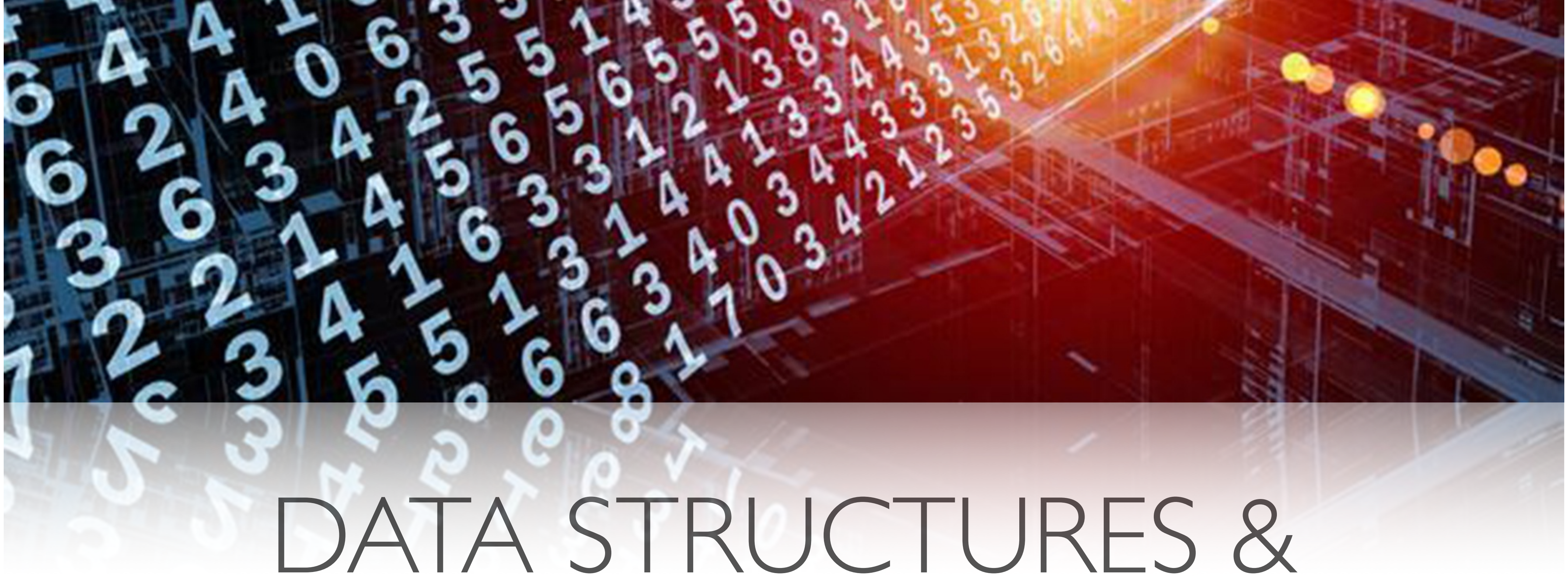


DATA STRUCTURES & ALGORITHMS

COMP H3025

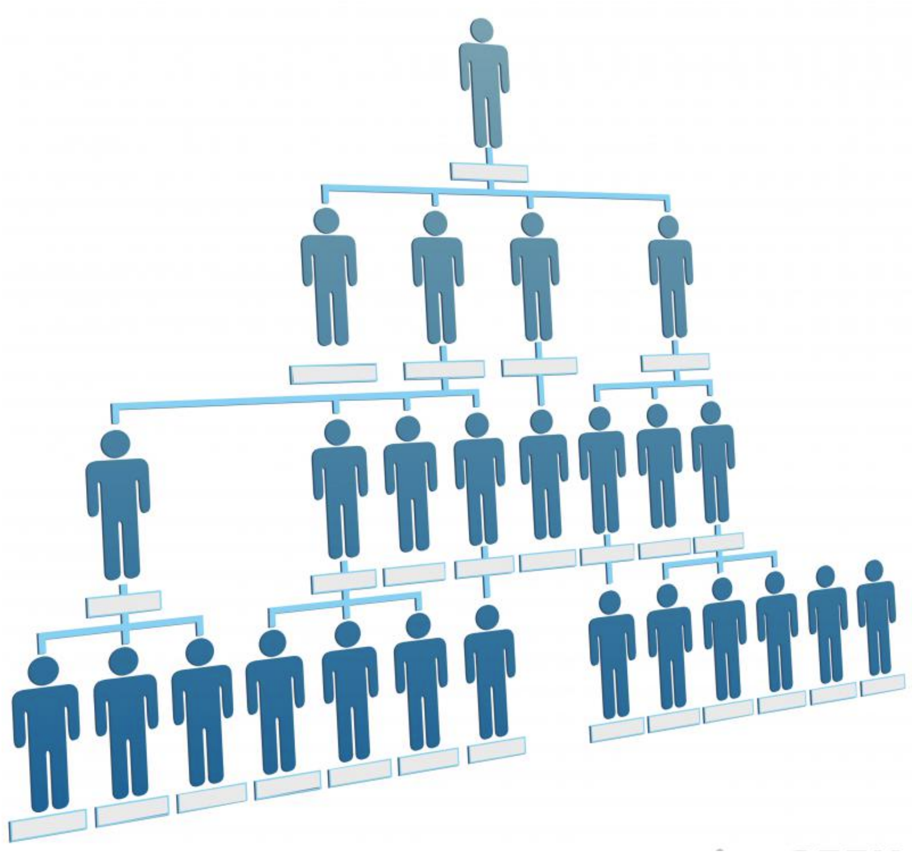
Lecture 6: Trees (Non linear data structures)

Lecturer: Stephen Sheridan

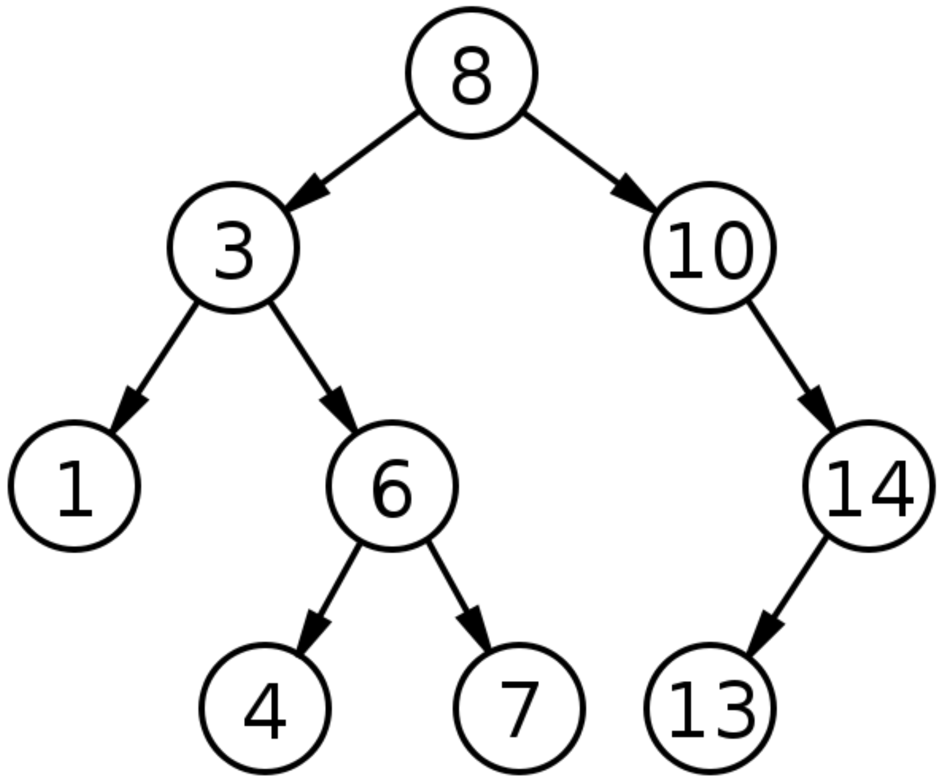
BINARY TREES

- The abstract data types we have studied so far have all been **linear** in nature. That is their data items have all been stored one after the other.
    - Array, Linked list, Stack & Queue
- The abstract data types that we will cover next organise data in a **non-linear**, hierarchical manner, whereby an item can have more than one immediate sucessor.
- In this lecture we will cover one such non-linear data structure called a **Binary Search Tree**.

NON-LINEAR HIERARCHICAL STRUCTURES



Organisation Chart, branches are not limited.



Binary Search Tree, each node has at most two branches.

TREE TERMINOLOGY

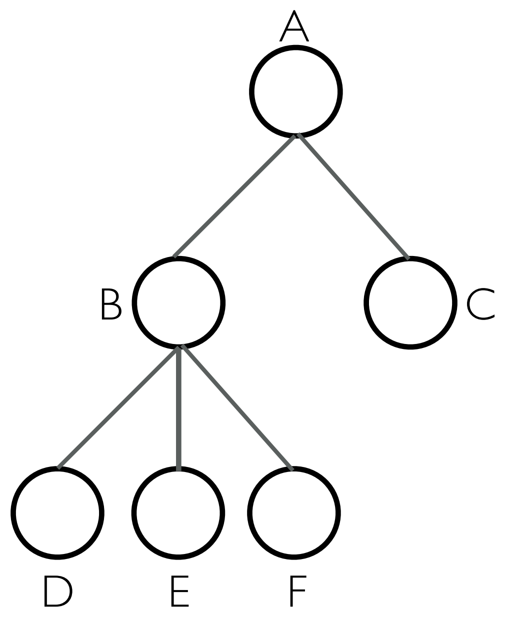
- We use trees to represent relationships. A tree is made up of **nodes** or **vertices**. The lines between the nodes are called **edges**. All trees are hierarchical in nature.

- By **hierarchical** we mean that a **parent-child** relationship exists between nodes in the tree.

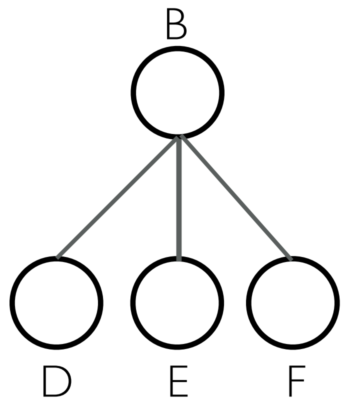
  If an edge is *between* node **n** and node **m**,
  and node **n** is *above* node **m** in the tree
  then **n** is the *parent* of **m** and **m** is the *child* of **n**.

TREE TERMINOLOGY

- In the following diagram, nodes **B** and **C** are **children** of node **A**.
- Children of the same parent (B and C) are called **siblings**.
- Each node in a tree has at most one **parent**, and exactly one node, called the **root** of the tree, has **NO** parent.
- A node that has no children is called a **leaf** of the tree.
- The leaves of the tree in the diagram are **D**, **E**, **F**, and **C**.



A general tree



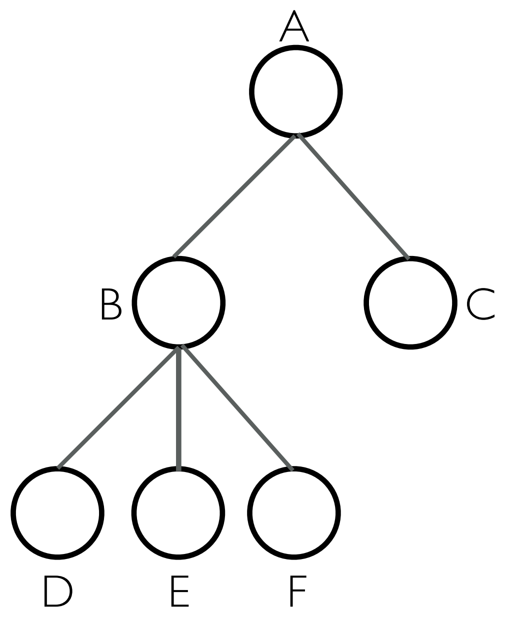
A subtree, is any node and its descendants

TREE TERMINOLOGY

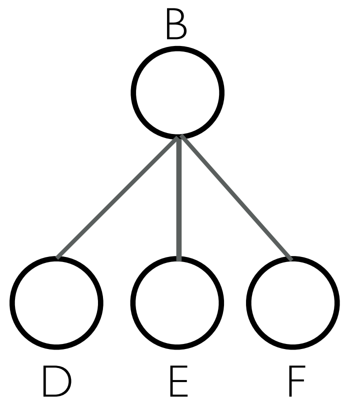
- The parent-child relationship between nodes can be generalised to the relationship **ancestor** and **descendant**.

- In the diagram, **A** is an *ancestor* of **D** and **D** is a *descendant* of **A**.

- Not all nodes are related by the ancestor/descendant relationship. **B** and **C** are not related in this way.



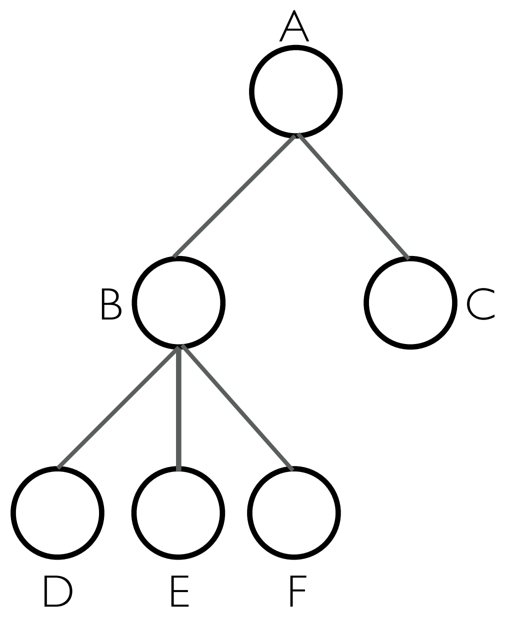
A general tree



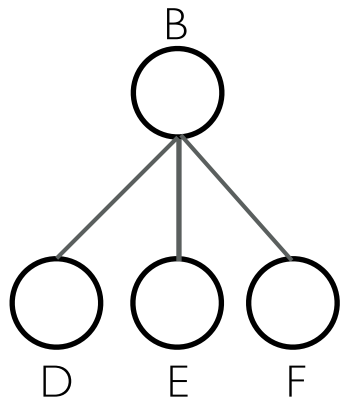
A subtree, is any node and its descendants

TREE TERMINOLOGY

- The **root** of any tree is an ancestor of every node in that tree.
- A **subtree** in a tree is any node in the tree together with all its descendants.
- A subtree of a node **n** is a subtree rooted at a **child** of **n**.
- For example, the diagram shows a subtree of the tree in the earlier diagram. So the second diagram shows a subtree of node **A**.



A general tree

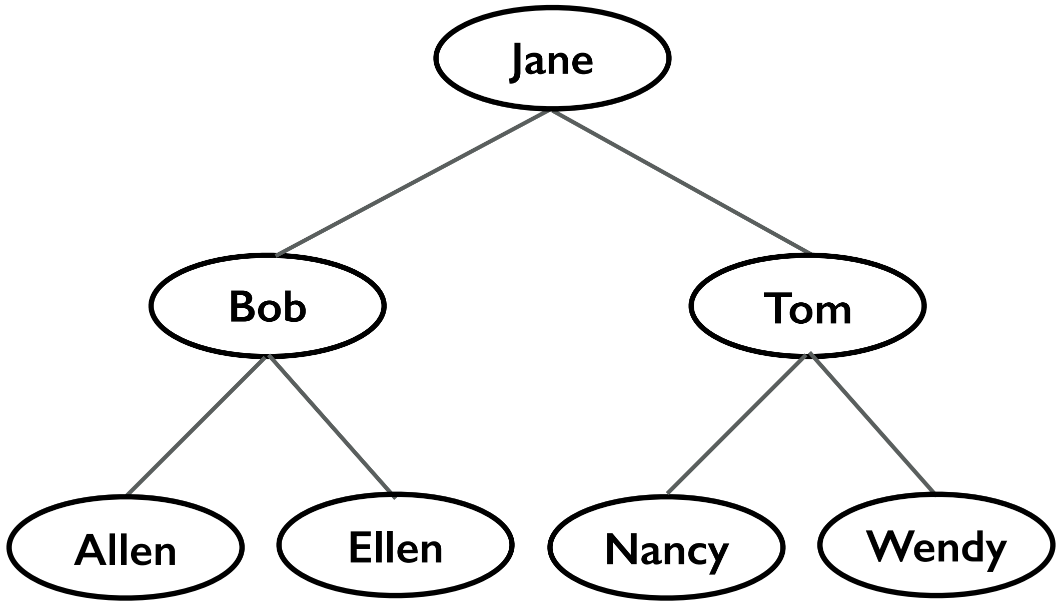


A subtree, is any node and its descendants

BINARY SEARCH TREES

- Because trees are naturally hierarchical, they can be used to represent information that is itself hierarchical.

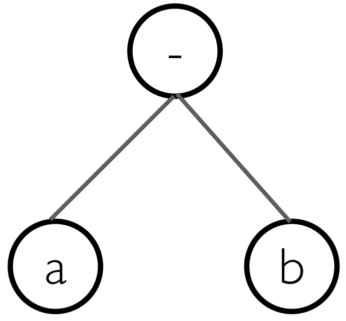
- For example, the diagram below shows a binary search tree of names.



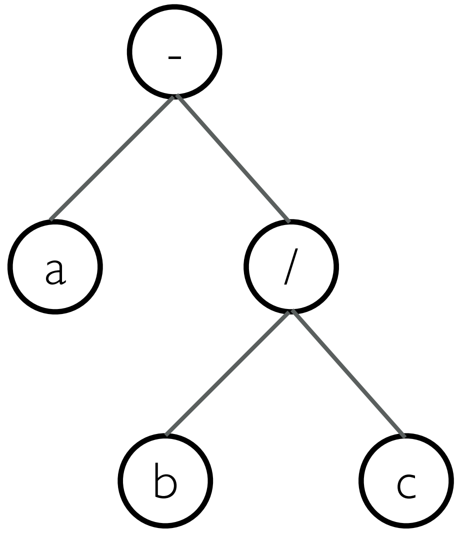
BINARY SEARCH TREES

- Binary trees can represent algebraic expressions that involve the binary operators +, -, * and /.
- To represent an expression such as *a - b,* we place the operator in the root and the operands in the left and right children respectively.
- NOTE: the leaves of these tree contain the operands and other nodes contain the operators. Also notice that brackets are not stored in the tree.

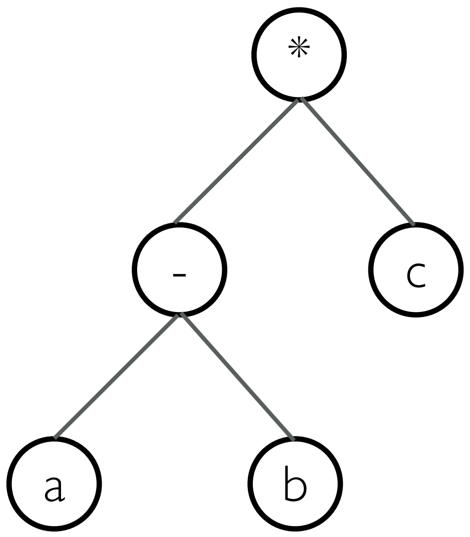
a - b



a - b / c



(a - b) * c



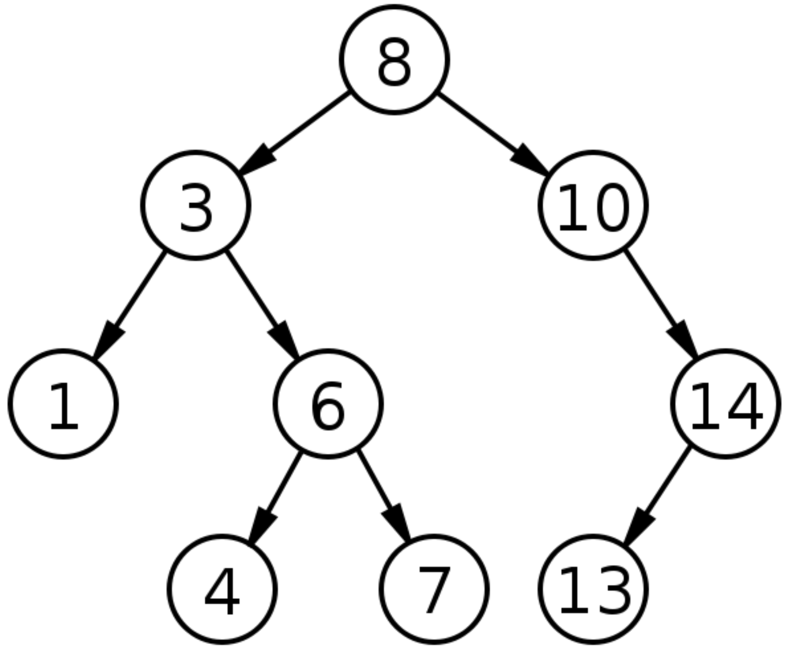
BINARY SEARCH TREES

- Binary search trees are sorted according to the values in its nodes.

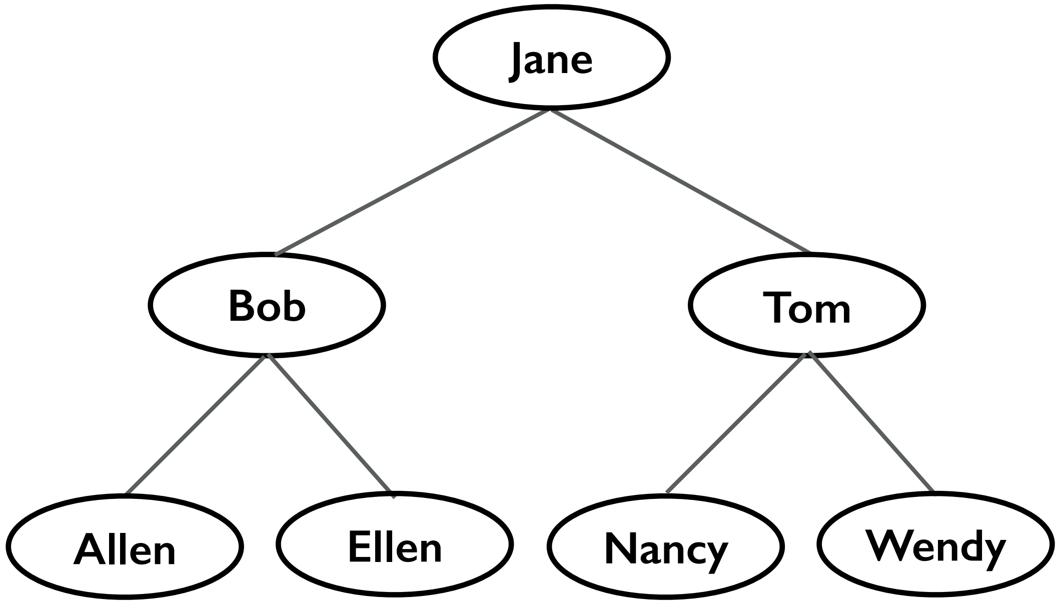
- For each node n, a binary search tree satisfies the following three properties:
  - The value of n is greater than all values in its left subtree $\mathbf{T_L}$
  - The value of n is less than all values in its right subtree $\mathbf{T_R}$
  - Both $\mathbf{T_L}$ and $\mathbf{T_R}$ are binary search trees

BINARY SEARCH TREES - ORDER

- Data in a binary search tree is stored in a way that facilitates searching for a particular item.

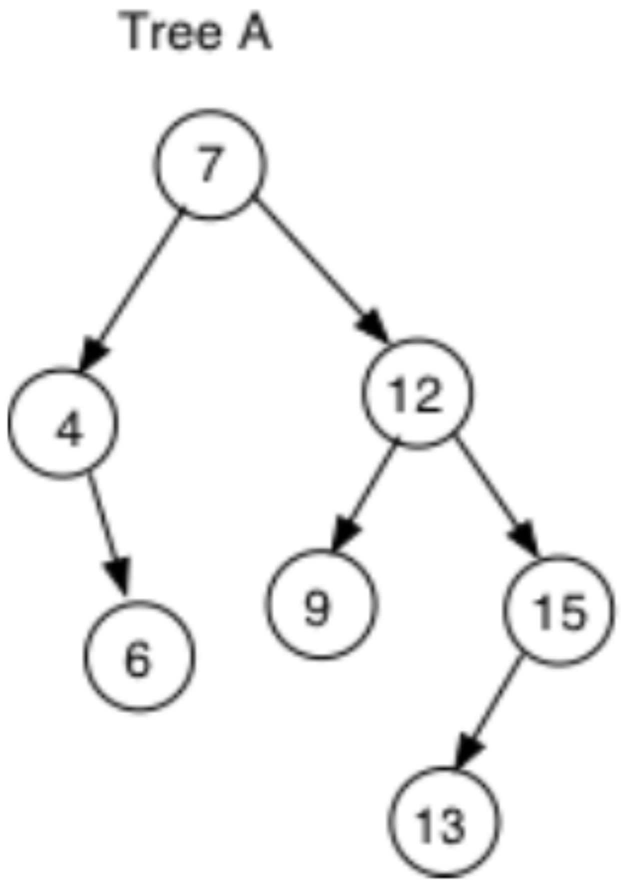


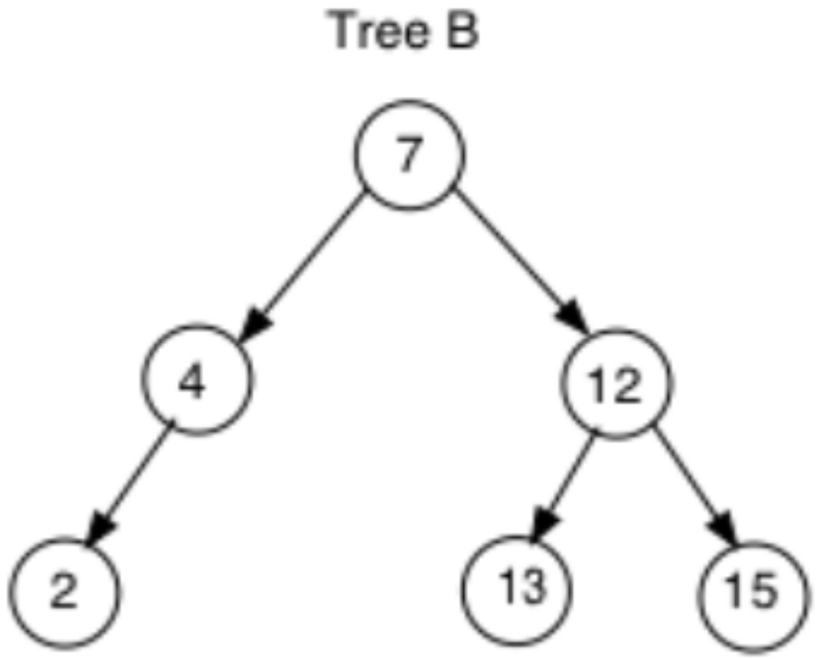
Numerical order



Alphabetical order

BINARY SEARCH TREES - ORDER





Binary search tree

NOT a binary search tree

BINARY SEARCH TREES - ORDER

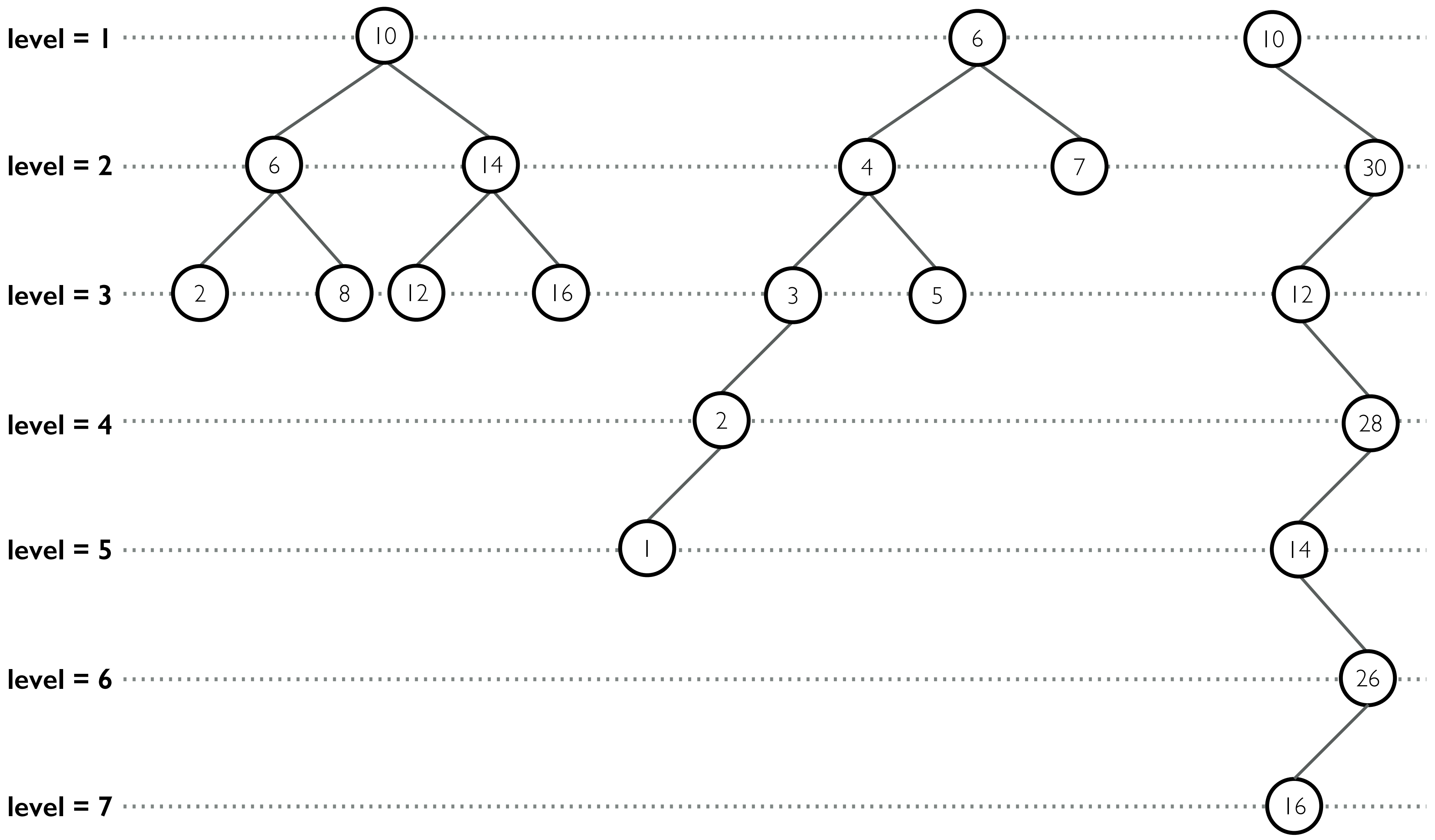
QUESTION: Create a binary search tree from the following list of values with the first value as the root.

list = {9, 6, 14, 8, 11, 17, 15}

BINARY SEARCH TREES - HEIGHT

- The binary search trees in the following diagrams all have the same number of nodes but have different structures.
- Although they have the same number of nodes some are **"taller"** than others.
- The height of any tree is the number of nodes on the longest path from the root to a leaf.

BINARY SEARCH TREES

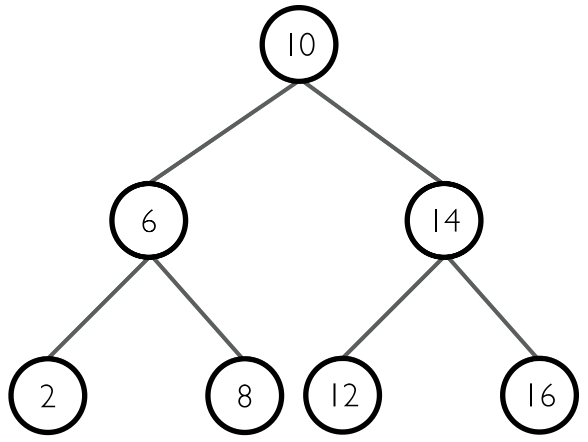


BINARY SEARCH TREES - FULL TREE

- In a **full** binary tree of height **h**, all nodes that are at a level less than **h** have **two children each**.

- Each node in a full binary tree has **left** and **right subtrees** of the same **height**.

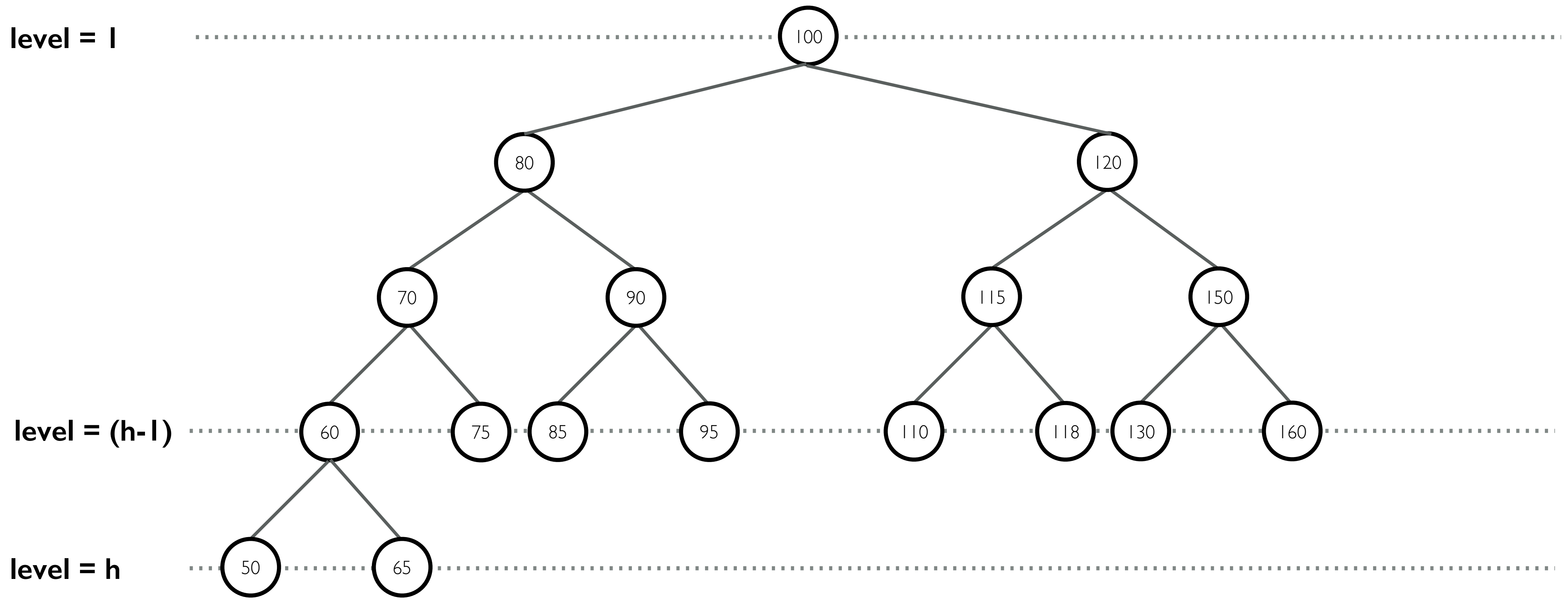
- Among any binary trees of height h, a full binary tree has as many leaves as possible. All leaves are at level h.



A full binary tree of height 3

BINARY SEARCH TREES - COMPLETE TREE

- A complete binary tree of height h is a binary tree that is
    - full down to level (h-1)
    - with level h filled in from left to right

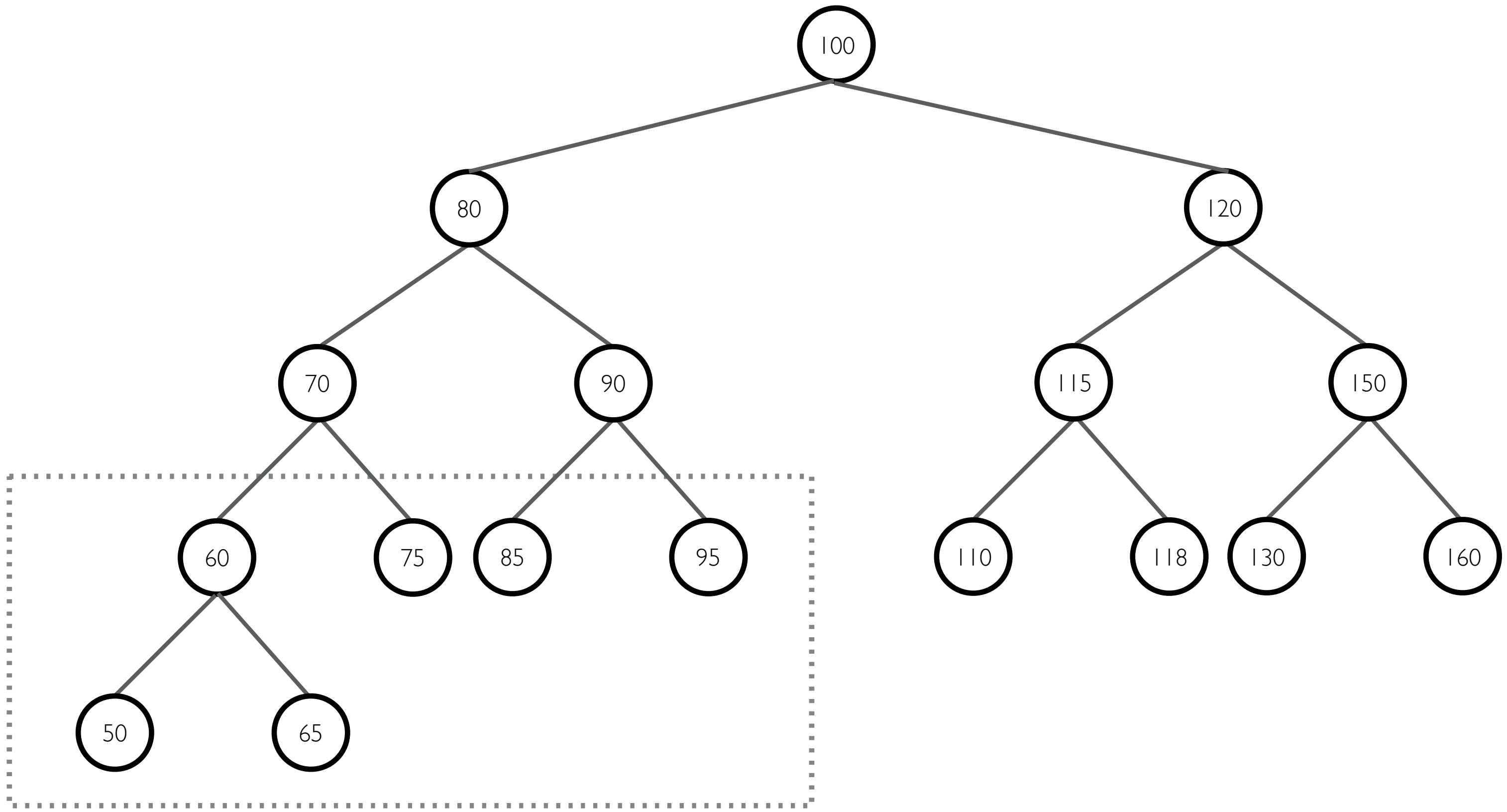


BINARY SEARCH TREES - COMPLETE TREE

- More formally, a binary tree **T** of height **h** is complete if:
    1. All nodes at level **(h-2)** and above have two children
    2. When a node at level **(h-1)** has children, all nodes to its left at the same height have two children
    3. When a node at level **h** has one child, its a left child
- Parts 2 and 3 of this definition formalise the requirement that level h be filled form left to right.
- **NOTE**: a full binary tree is **complete!**

BINARY SEARCH TREES - BALANCED TREE

- A binary tree is **height balanced** if the height of any node's **right subtree** differs from the height of the node's **left subtree** by no more than **1**.



ADT BINARY TREE

- As an abstract data type, the binary tree has operations that add and remove nodes and subtrees. These operations can be used to build a tree.

- Other operations set or retrieve the data of the tree and determine whether the tree is empty.

- Traversal operations that visit every node in a binary tree are also typical.

BINARY TREE - TRAVERSAL

- We have already seen traversal with a linked list. Beginning with the lists first node we visit each node sequentially until the end of the list is reached.

- Traversal of a binary tree visits the tree's nodes in one of several different orders.

- The three standard orders are:
  - **Pre-order**
  - **In-order**
  - **Post-order**

BINARY TREE - TRAVERSAL

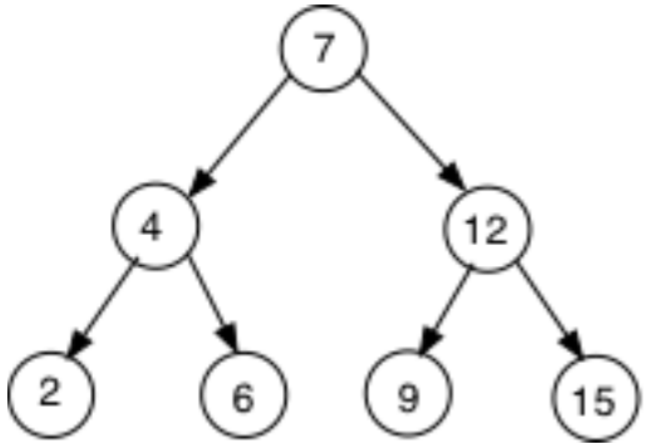
- These are three natural ways in which we can visit each node in a binary tree. These are based on the recursive nature of the tree structure. The order of possible visitations is:

(1) Preorder
process root;
process left sub-tree;
process right sub-tree
	(2) Inorder
	process left sub-tree;
	process root;
	process right sub-tree

(3) Postorder
process left sub-tree;
process right sub-tree;
process root	

BINARY TREE - TRAVERSAL

- Consider the following binary tree:



- Traversal using
  - **preorder** is: 7, 4, 2, 6, 12, 9, 15
  - **inorder** is: 2, 4, 6, 7, 9, 12, 15
  - **postorder** is: 2, 6, 4, 9, 15, 12, 7
- Observe that an **inorder** traversal of the tree gives a **sorted** list of values.

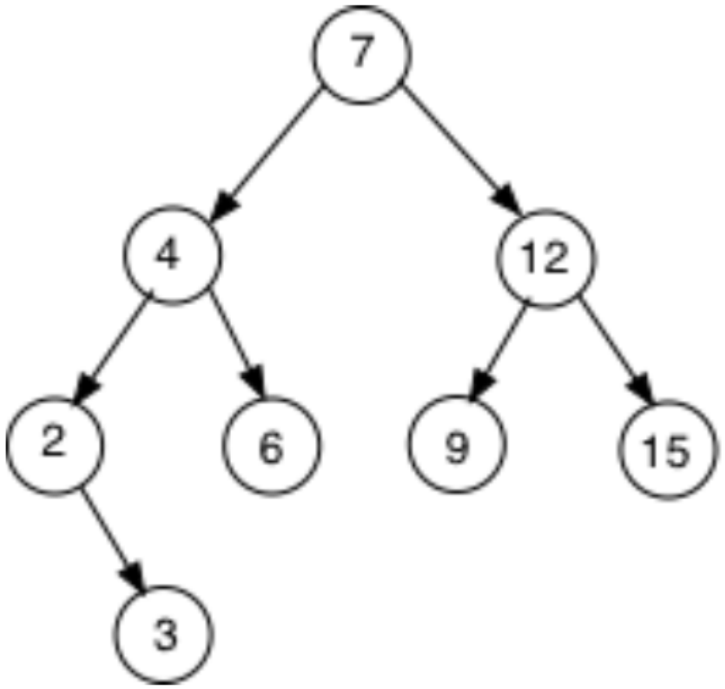
BINARY TREE - ADDING A NEW VALUE

- When adding a new value we must ensure that the value in every node must be greater than all values in its left sub-tree, and less than all values in its right sub-tree is invariant.

- A new value is added as a leaf node in such a way that this rule is preserved. To insert the value 3 in the tree given above search the tree until we find the correct node to attach it to.

BINARY TREE - ADDING A NEW VALUE

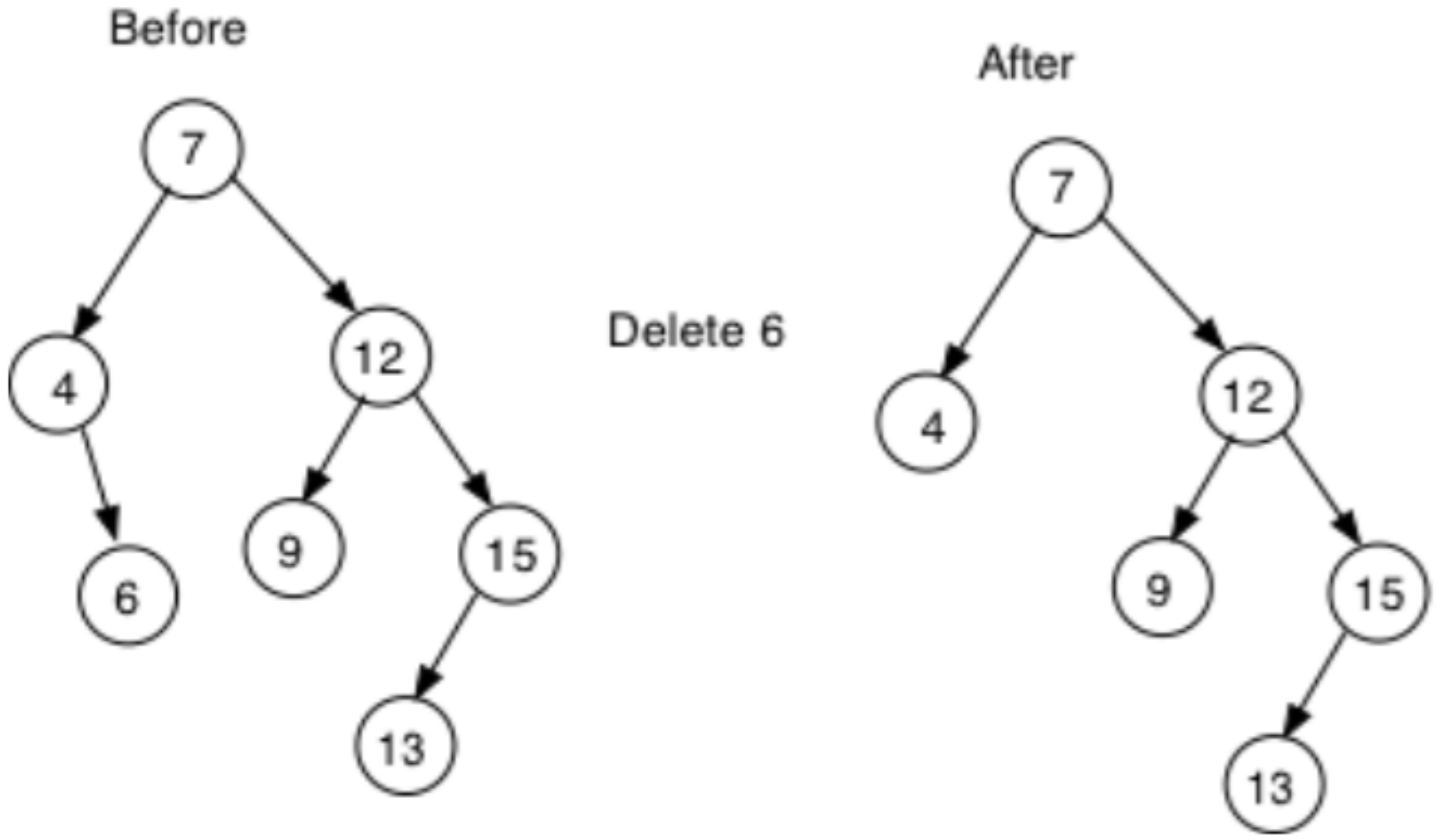
- In this tree the only possible position is the right sub-tree of node 2. This gives:



- To construct a binary search tree from a given list of values the first item on the list becomes the root node.

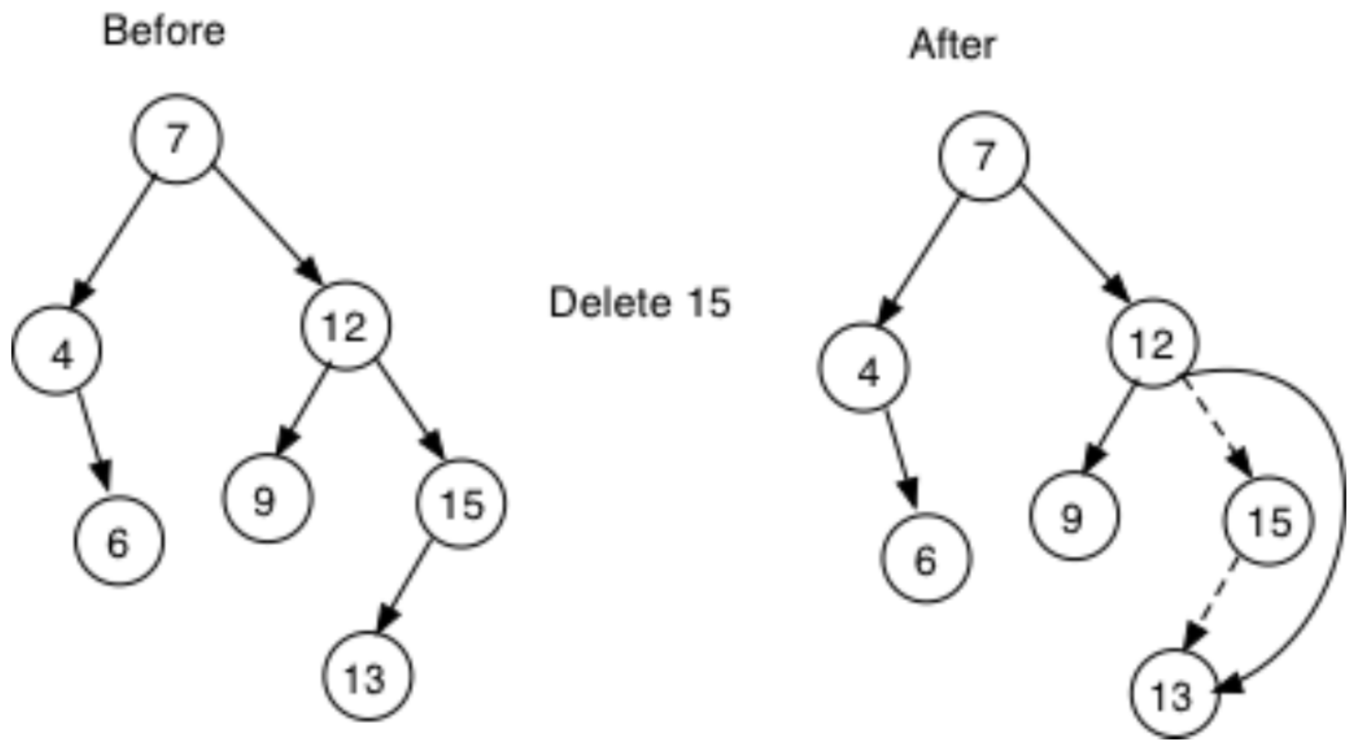
BINARY TREE - DELETING A VALUE

- Deleting an element is relatively simple if the element is a leaf node or a node with a single descendent. The diagram below shows a binary search tree prior to deleting the leaf node 6.



BINARY TREE - DELETING A VALUE

- If the node to be deleted has a single descendent then the parent node simply points to the descendent of the node to be deleted. The diagram below illustrates the deletion of the node containing the value 15. It has a single descendent and its parent node points to it after the delete operation.



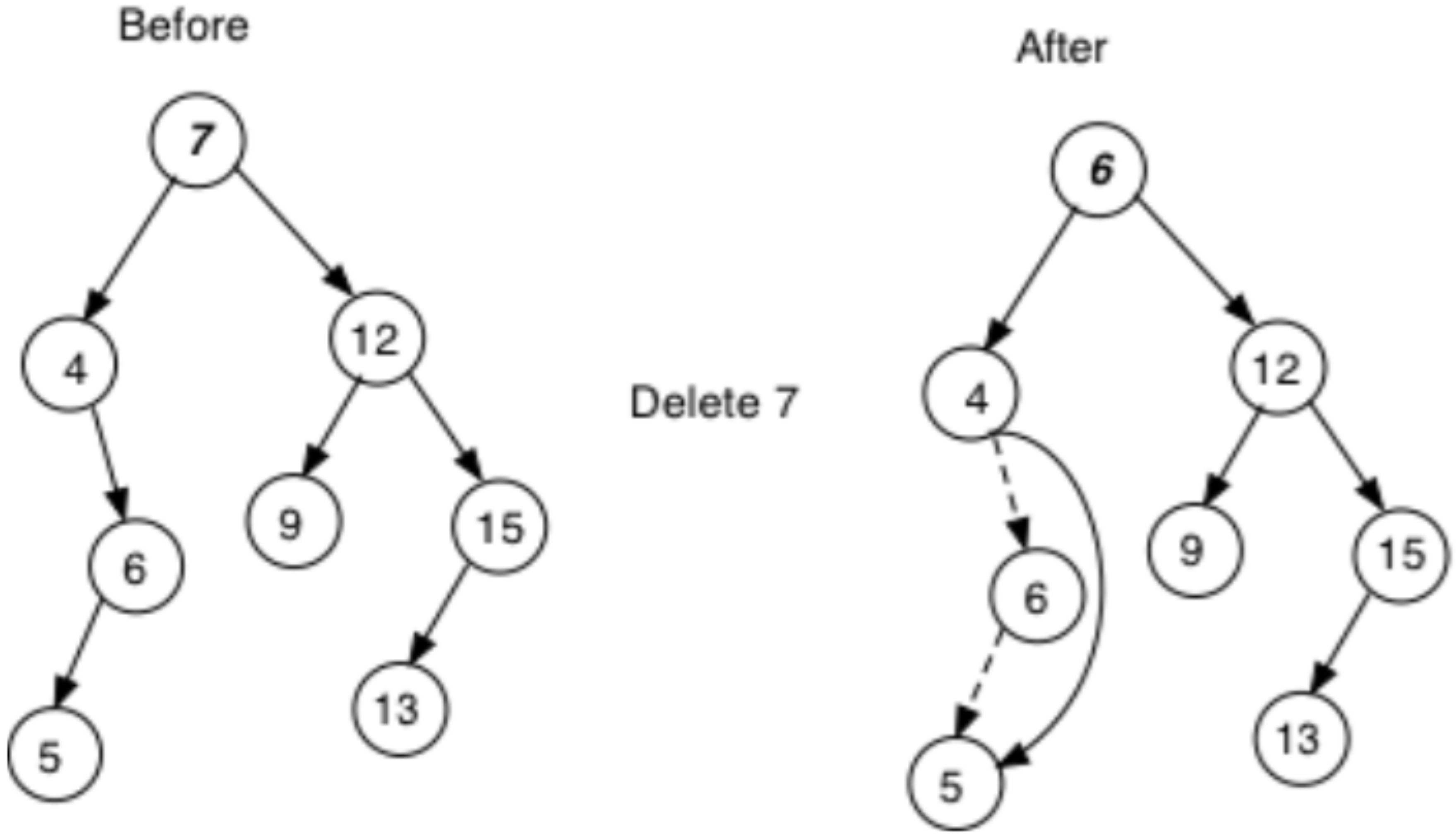
BINARY TREE - DELETING A VALUE

- However, removing a node with two descendent sub-trees proves a problem. To remove such a node we cannot point its parent node in two directions at once. Hence, we must replace the data item with some other element in the tree.

- One possible solution is to replace the value in the node to be deleted by the rightmost element in its left sub-tree and then delete this node.

- The diagram on the next slide shows that state of the tree before the root node containing 7 is removed. The value 6 is the rightmost element of its left sub-tree and it is chosen as the candidate to replace it. This node has a single descendent and is removed by pointing its parent to this descendent node.

BINARY TREE - DELETING A VALUE



INTEGER BINARY TREE NODE

- A binary search tree has to maintain a sorted list of data elements. Each node has degree 2 and, hence, has to have a left and right reference to possible descendants.

- A new instance of **BNode** always initialises both descendants to null.

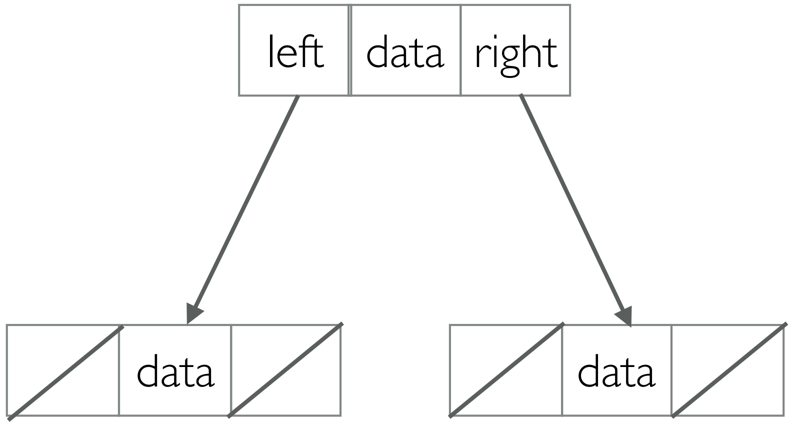
- The constructor takes a single instance of type **int** as an argument and initialises the data attribute.

- This value may also be modified using **set** and retrieved with the **data** method. The methods **setLeft** and **setRight** are used to connect child nodes.

INTEGER BINARY TREE NODE

```
class BNode{
    private int data;
    private BNode left;
    private BNode right;

    public BNode(int d){
        data = d;
        left = null;
        right = null;
    }
    public int data(){
        return data;
    }
    public void set(int d){
        data = d;
    }
    public BNode left(){
         return left;
    }
    public BNode right(){
         return right;
    }
    public void setLeft(BNode k){
         left = k;
    }
    public void setRight(BNode k){
         right =k;
    }
}
```



INTEGER BINARY TREE INTERFACE

- A binary search tree has to maintain a sorted list of data elements. Each node has degree 2 and, hence, has to have a left and right reference to possible descendants.

- A new instance of **BNode** always initialises both descendants to null.

- The constructor takes a single instance of type **int** as an argument and initialises the data attribute.

- This value may also be modified using **set** and retrieved with the **data** method. The methods **setLeft** and **setRight** are used to connect child nodes.

INTEGER BINARY TREE OPERATIONS

```
public void add(int d);
// Traverse the tree and find the correct
// position to insert the value d
public boolean contains(int d);
// Traverse the tree and return true if
// the value d is found
public ArrayList inOrder();
// Traverse the tree and return an ArrayList
// populated with tree values in inOrder
public ArrayList preOrder();
// Traverse the tree and return an ArrayList
// populated with tree values in preOrder
public ArrayList postOrder();
// Traverse the tree and return an ArrayList
// populated with tree values in postOrder
public void remove(int d);
// Search for the data value d in the tree
// and remove if it exists
public int height();
// Return the length of the longest leaf path
```

TODO - WEEK 6

- Implement an integer based Binary Search Tree Node.
- Implement an Integer Binary Search Tree ADT based on the IntegerBSTInterface
- Create a simple test program that verifies all operations are working.