A tree is a linked node structure such that

- Up to 2 children per node
- Root exists, top parent
- Each node except root has exactly one parent (not a DAG)
- Can't be pointed to > 1
- No cycles

The sorted property says:

1. Child values < parent.value < rchild values

The heap property says:

1. All child values < parent.value < equality
2. All rchild values > parent.value
3. No duplicate values
4. 1 ≤ p < r or 1 < p ≤ r
B. The two properties

In the sorted property, all the left child subtree’s keys are less than the root, and all the right child subtree’s keys are greater than the root.

```
      value
    cat
     /   \
  ape    zebra
 /   \    /   
ant  asp  dog
```

This is wrong, not a BST, because dog > cat

The Zybooks talks about the left child’s keys ≤ the root, and all the right child’s keys ≥ the root’s value. However, if they both include the = clause, we could run into trouble. So we usually choose one side or the other to be equal.

```
      value
    cat
     /   \
  ape    dog
 /   \    /   
ant  cat  dog
   \   /   
    cat
```

This is technically a BST, but since there is not just one node with CAT or one node with DOG, it makes it hard to search. So in practice, this won’t happen, or we would confine the equals version to just, say, the right subtree.

$$1 \leq p \leq r$$

Most BST do not have duplicates (key = value) payload.

Heap property is a weaker form of the sorted property. However, it has some amazing uses.

```
      cat
     /   
  ape   dog
```

BST, but violates the heap property

```
      dog
     /   
  ape   cat
```

Heap property OK, one version

```
      dog
     /   
  cat   ape
```

Heap property OK, second version

$$\Theta(n \log n)$$ -> heapsort
$$\Theta(n^2)$$ -> selection
$$\Theta(n \log n)$$ -> quicksort
$$\Theta(n \log n)$$ -> avg
C. Let's write code:

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.lchild = None
        self.rchild = None

class Tree:
    def __init__(self):
        self.root = None
```

Tree code tends to be recursive because it is very easy to talk about it.
- A tree can be None
- A tree can be one node
- Every node of a tree points to either
  - Nothing
  - another tree (whose nodes are distinct from all other parts of the tree, i.e. the root node is not included in this subtree) – called the left child
  - another tree – call the right child

Both left child and right child are trees. We sometimes call them subtrees.

Let's write some methods:
```python
class Tree:
    def add2(self, value):
        """Start at the root, that is the current node""
        self.add2_aux(self.root, value)

    def add2_aux(self, nodeptr, value):
        if value < nodeptr.value:
            if nodeptr.lchild is None:
                nodeptr.lchild = Node(value)
            else:
                self.add2_tree_aux(nodeptr.lchild, value)
        else:
            if nodeptr.rchild is None:
                nodeptr.rchild = Node(value)
            else:
                self.add2_tree_aux(nodeptr.rchild, value)
```

API

private

--- insert here as

aux = auxilium = help

if value < nodeptr.value:
    if nodeptr.lchild is None:
        nodeptr.lchild = Node(value)
    else:
        self.add2_tree_aux(nodeptr.lchild, value)
else:
    if nodeptr.rchild is None:
        nodeptr.rchild = Node(value)
    else:
        self.add2_tree_aux(nodeptr.rchild, value)
Now let's write `find(self, value)`:
It will also be recursive.

```python
class Tree:
    def find(self, value):  # value is the target
        return self._find_aux(self, root, value)

    def _find_aux(self, self, node_ptr, value):
        if node_ptr.value == value:
            return True
        elif node_ptr.value > value:
            return self._find_aux(node_ptr.lchild, value)
        else:
            return self._find_aux(node_ptr.rchild, value)

        if node_ptr == None:
            return False
```

We could also write delete, but let's just sketch out the idea of the algorithm.

Cases:
- You want to delete the root
- You want to delete a leaf (super easy)
- You want to delete an interior node
D. Types of trees based on children

**Complete binary tree:** every node has 0 or 2 children:

An imbalanced tree violates this property somewhere. You can also have ternary trees.

**Balanced tree:** There are \( k \) levels. Every level 0, 1, 2, \ldots, \( k-1 \) are complete. The \( k^{th} \) level might be incomplete. So every complete binary tree is balanced.

A fascinating way to implement a balanced tree is with an array (a list):

Here's a binary tree represented in memory, and pictorially:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
36 & 49 & 12 & 99 & 62 & 50 & 84 & 71 & 29 & 107 \\
\end{array}
\]

The root is at position 1. The left child of a node at position \( p \) is \( 2p \). The right child of a node at position \( p \) is \( 2p+1 \).

This assumes there are no "holes" in the tree. If there were, we would just use a special value to designate them. You can insert at the end only, but surprisingly you can still conceptually insert anywhere in the tree! It is just that no leaf in the \( k^{th} \) level can have a child until all the \( k-1 \) nodes have a full quota of 2 children. So you could not give 107 a child yet. Not until 50, 84, 71 and 29 are full.
E. **Ternary and higher trees**

What would a ternary search tree look like? It would have three children. Here's a 4-way tree. It has 3 keys and 4 children. So a ternary tree would have 2 keys.

![Ternary Search Tree Diagram](image)

Applications in search engines like Google.

Applications in databases.
public API — visible to users

private implementation
details < no user burden
proprietary