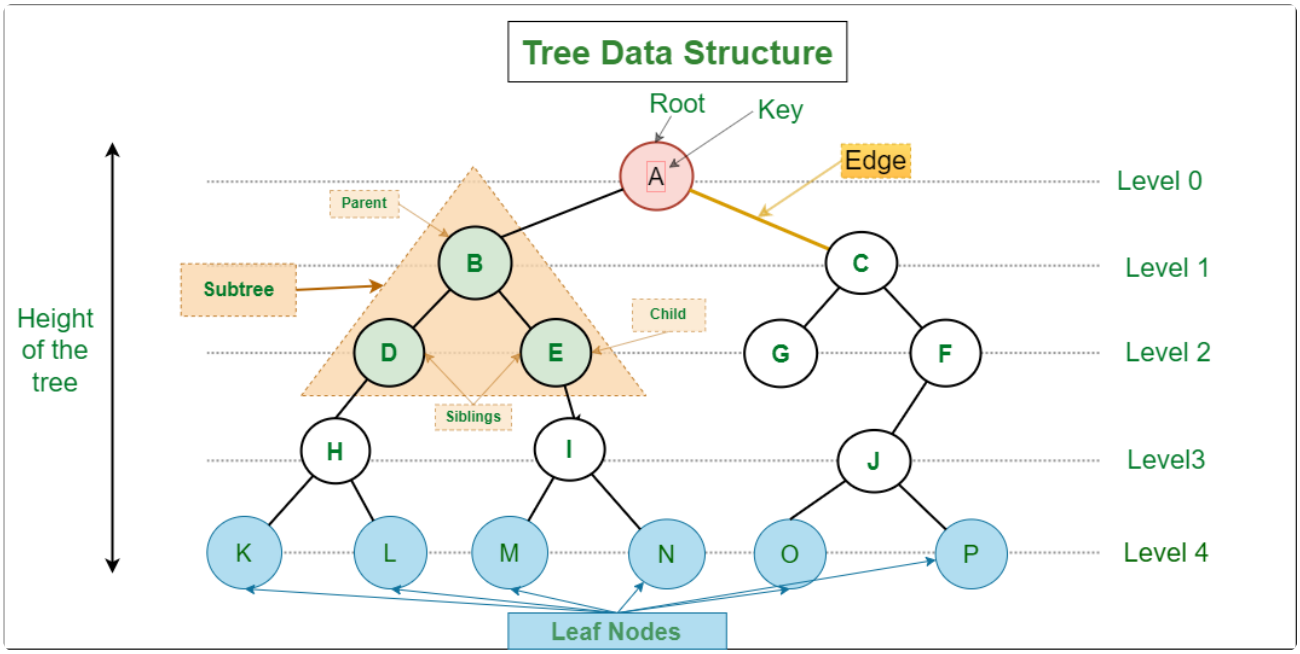
Trees

Trees are specialized graphs that do not have cycles in them, which means they do not loop
Hard drive in a computer is a B* tree

Binary Tree



Every tree has three parts:

- **Exactly ONE** root
- Each node in Binary Tree has **AT MOST** 2 children
- Binary tree always goes Top → Down, Left → Right
- Binary tree has no order.

Nodes that do not have children are called **leaf nodes**
The node above is the **parent**
Nodes on the same level are **sibling nodes**

Node class is similar to DoublyLinked node, except next and prev are now left and right

Binary Node

```
public class BinaryNode {
        private int data;
        private BinaryNode left;
```

```
        private BinaryNode right;

        public BinaryNode(int data, BinaryNode left, BinaryNode right) {
                this.data = data;
                this.left = left;
                this.right = right;
        }
        public BinaryNode(int data) {this(data, null, null);}

        public void setData(int data) {this.data = data;}
        public int getData() {return this.data;}
        public void setLeft(BinaryNode left) {this.left = left;}
        public Node getLeft() {return this.left;}
        public void setRight(BinaryNode right) {this.right = right;}
        public Node getRight() {return this.right;}
}
```

Binary Search Tree(BST)

We want to make the searching of the tree simple and fast.
It still contains:

- Exactly 1 root
- At most 2 children
- Smaller items are in the Left Sub-tree
- Larger items are in the Right Sub-tree

Subtree:

Each node you are looking at has subtrees: left and right

The node all the way on the left is the minimum
The node all the way on the right is the maximum
To find minimum, traverse all the way on left until next is null, traverse right size for maximum

We need to learn a few operations:

- Insertion
- Find
- Delete
- Walk around:
    - Pre-order

- Post-order
- Level-order

```
public class BinarySearchTree {
        private BinaryNode root;
        public BinarySearchTree() {this.root = null;}

        public void insert(int data) {
                BinaryNode newNode = new BinaryNode(data);
                if(root == null) {root = newNode;}
                else{
                        BinaryNode node = root;
                        BinaryNode parent = null;
                        while(node != null) {
                                parent = node;
                                if(data < node.getData) {node =
node.getLeft();}
                                else {node = node.getRight();}
                        }
                        if(data < parent.getData())
{parent.setLeft(newNode);}
                        else {parent.setRight(newNode);}
                }
        }

        public int getMin() throws EmptyTreeException {
                if(root == null) {
                        throw new EmptyTreeException(); //you can use just
Exception as well
                }
                BinaryTree minNode = root;
                while(minNode.getLeft() != null){minNode =
minNode.getLeft();}
                return minNode.getData();
        }

        public int getMax() throws EmptyTreeException {
                if(root == null) {throw new EmptyTreeException();}
                BinaryTree maxNode = root;
                while(maxNode.getRight() != null) {maxNode =
maxNode.getRight();}
                return maxNode.getData();
        }
}
```

Let’s make the deletion method.
There are three cases to delete:

- Deleting leaf node
- Deleting node with one child
- Deleting node with two children (hardest case)
  - Move one to the left, then all the way right. The last node is what you will replace it with
    - Now it will have either one child or none
  - Move right once, then all the way to the left. Swap node with last left
    - Now it will have either one child or none

```
public boolean delete(int data) {
        if(root == null) {return false;}
        BinaryNode node = root;
        BinaryNode parent = null;
        while(node != null && node.getData() != data) {
                parent = node;
                if(data < node.getData()) {node = node.getLeft();}
                else{node = node.getRight();}
        }
        if(isCase1(node)) {
                deleteCase1(parent, node);
        }
        else if(isCase2(node)) {
                deleteCase2(parent, node);
        }
        else {
                parent = node;
                BinaryNode tmp = node.getLeft();
                while(tmp.getRight() != null) {
                parent = tmp;
                tmp = tmp.getRight();
                }
                int nodeData = node.getData();
                node.setData(tmp.getData());
                tmp.setData(nodeData);
                if(case1(tmp)) {deleteCase1(parent, tmp);}
                else{deleteCase2(parent, tmp);}
        }
 }

public void deleteCase1(BinaryNode parent, BinaryNode ndoe) {
        if(parent.getLeft() == node) {parent.setLeft(null);}
        else{parent.setRight(null);}
}
```

```
private void deleteCase2(BinaryNode parent, BinaryNode node) {
        BinaryNode child = node.getLeft() != null ? node.getLeft() :
node.getRight();
        /*
        This is basically saying:
        BinaryNode child = null;
        if(node.getLeft != null) {
                child = node.getLeft();
        }
        else {
                child = node.getRight();
        }
        */
        if(parent.getLeft() == node) {parent.setLeft(child);}
        else {parent.setRight(child);}
}

private boolean isCase1(BinaryNode node) {
        return node.getLeft() == null && node.getRight() == null;
}
private boolean isCase2(BinaryNode node) {
        return (node.getLeft() != null && node.getRight() == null) ||
(node.getLeft() == null && node.getRight != null);
}
```

```
public class EmptyTreeException extends Exception {
        public EmptyTreeException() {
                super();
        }
}
```

```
public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        bst.insert();

        //How to handle exceptions, you must catch what you throw!!
        try {
                int min = bst.getMin();
        }catch(EmptyTreeException e) {
                System.out.println("Tree is empty " + e.toString());
        }
}
```

Pre Order Traversal

Operation, Left, Right

```
public void preOrder() {preOrder(root);}
private void preOrder(BinaryNode node) {
        if(node == null) {return;}
        System.out.println(node.getData());
        preOrder(node.getLeft());
        preOrder(node.getRight());
}
```

In Order Traversal

Left, Operation, Right
Goes through a binary search tree in **sorted order**

```
public void inOrder() {inOrder(root);}
private void inOrder(BinaryNode node) {
        if(node == null) {return;}
        inOrder(node.getLeft());
        System.out.println(node.getData());
        inOrder(node.getRight());
}
```

Post Order Traversal

Left, Right, Operation

```
public void postOrder() {postOrder(root);}
private void postOrder(BinaryNode node) {
        if(node == null) {return;}
        postOrder(node.getLeft());
        postOrder(node.getRight());
        System.out.println(node.getData());
}
```

Height/Level of binary tree

height: Longest path from root to a leaf

```
public int height() {
        return height(root);
}
private int height(BinaryNode node) {
```

```
        if(node == null) {return 0;}
        return max(height(node.getLeft(), height(node.getRight())) + 1;
}
```