CS201

Data Structures and Algorithms

Revision Session 5

binary search trees

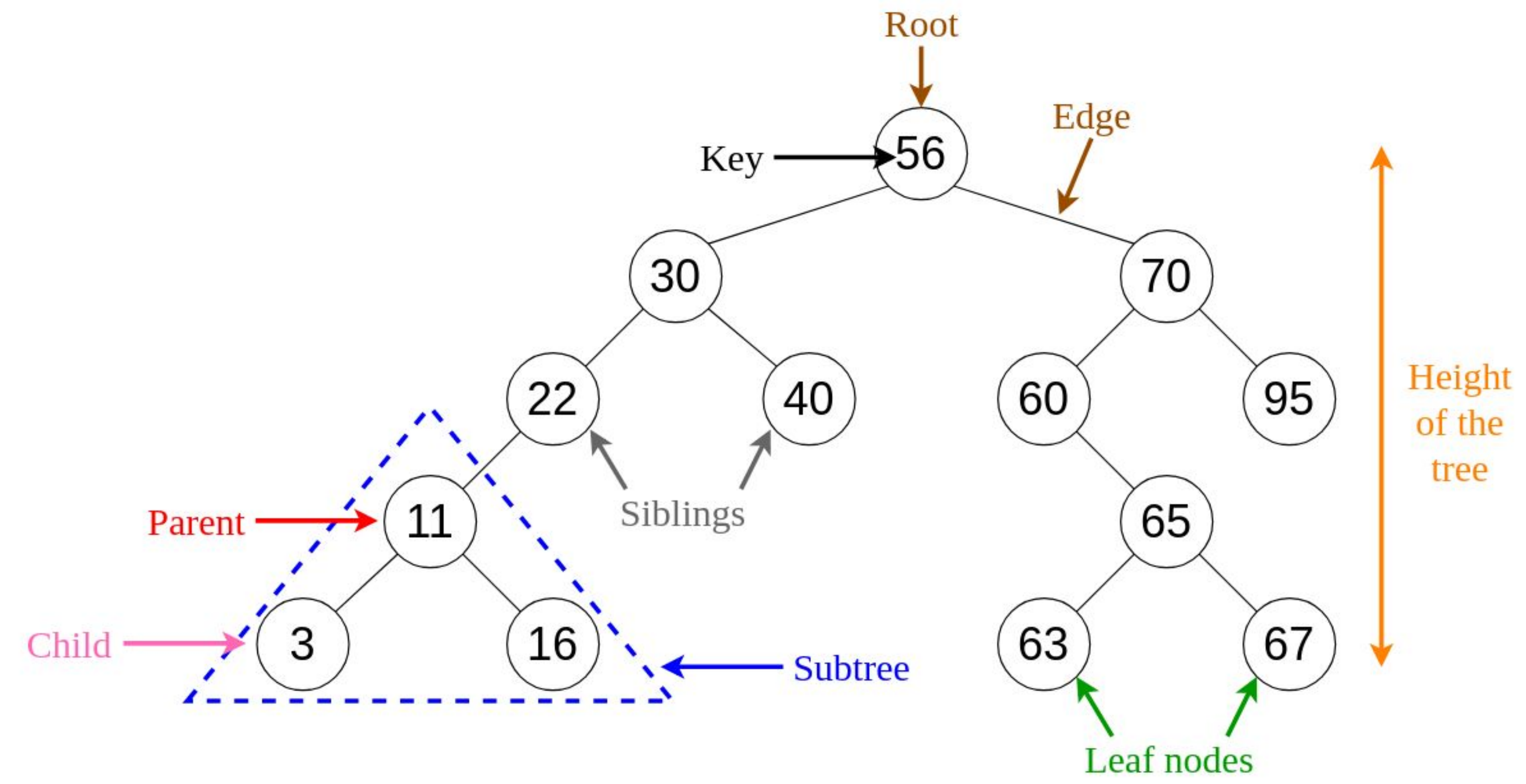
Binary Search Trees

basics:
definition
implementation

operations:
search (iterative and recursive)
minimum maximum search (iterative and recursive)
insertion
deletion

traversals:
iterative (preorder, inorder, postorder)
recursive (counting nodes with stack/queue)

definition


Root
Key
56
Edge
30
70
22
40
60
95
Height of the tree
Parent
11
Siblings
65
Child
3
16
Subtree
63
67
Leaf nodes

non-linear

tree

degree : number of children

ascendants vs. descendants

depth of a tree = number of nodes traversed to reach a leaf node

binary tree

binary search tree

balanced vs unbalanced

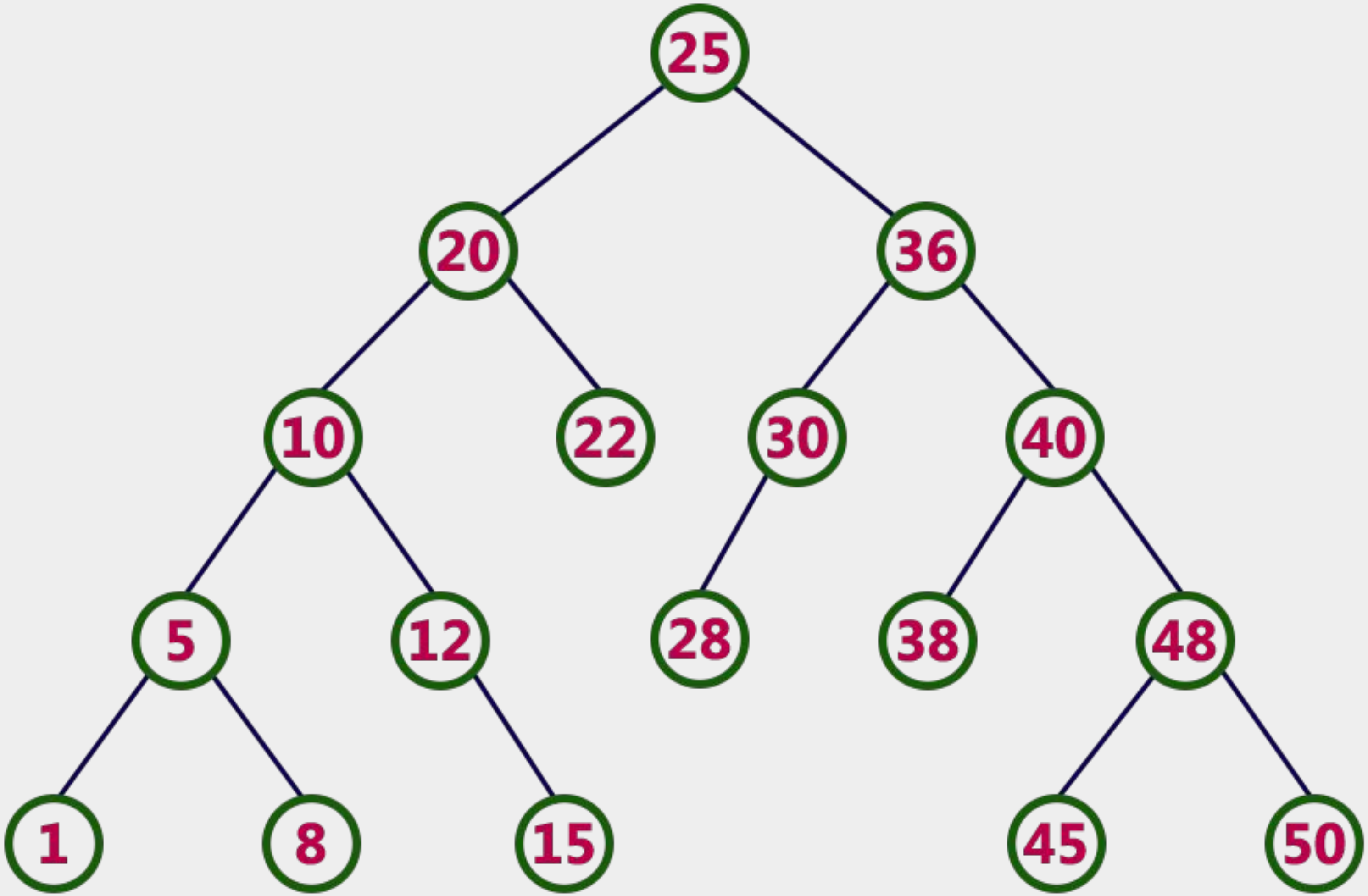
implementation

```java
public class TreeNode {
    3 usages
    int data;
    3 usages
    TreeNode left;
    3 usages
    TreeNode right;

    no usages   neslihancesurr
    public TreeNode(int data){
        this.data = data;
        left = null;
        right = null;
    }
```

```java
public class BinaryTree {
    4 usages
    TreeNode root;

    1 usage   neslihancesurr
    public BinaryTree(){
        root = null;
    }
```

search

iterative search (tree based)


25
20
36
10
22
30
40
5
12
28
38
48
1
8
15
45
50

Create a pointer for traversal

Start a loop to run until the leaf nodes.

If the data matches the value, return the node.

If the data is smaller than the value, move to the right (greater) node.

If the data is greater than the value,

move to the left (smaller) node.

```java
public TreeNode iterativeSearch(int value) {
    TreeNode tmp = root;

    while (tmp != null) {
        if (tmp.data == value) {
            return tmp;
        } else if (tmp.data < value) {
            tmp = tmp.right;
        } else {
            tmp = tmp.left;
        }
    }
    return null;
}
```

recursive search (node based)

Base case : when value equals the data, return the node

if value is greater and if right is not null, recall function on right side

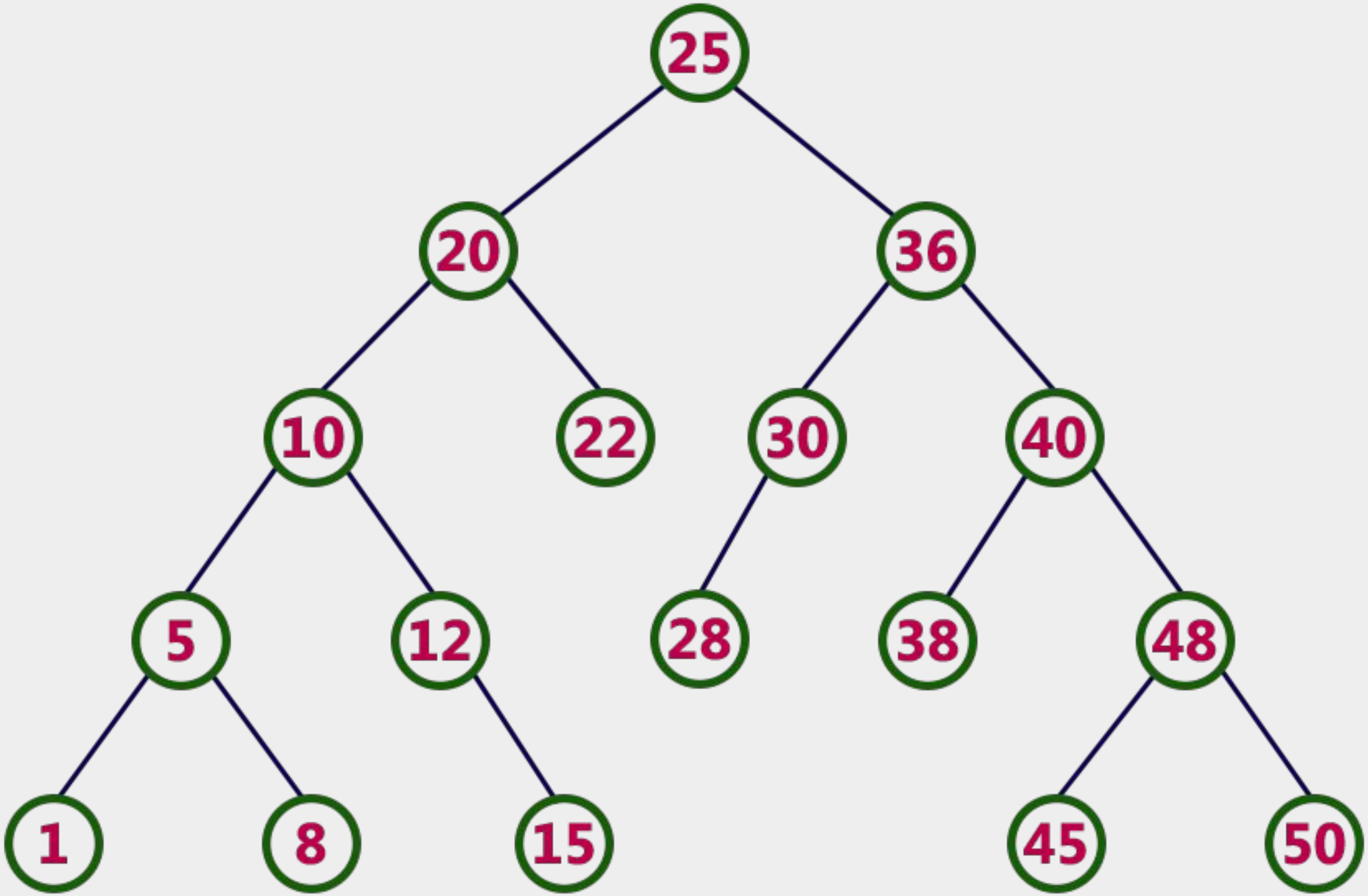
if there is no right side, return null

if value is smaller and left is not null, recall function on left side

if there is no left side, return null

```java
public TreeNode recursiveSearch(int value) {
    if (data == value) {
        return this;

    } else if (data < value) {
        if (right != null)
            return right.recursiveSearch(value);
        else
            return null;

    } else if (left != null)
        return left.recursiveSearch(value);
    else
        return null;
}
```

iterative min / max search (node class)


25
20
36
10
22
30
40
5
12
28
38
48
1
8
15
45
50

Using a pointer, move down to the left side until pointer has no left node. Return the pointer.

```java
public TreeNode iterativeMinSearch(){
    TreeNode tmp = this;

    while (tmp.left != null){
        tmp = tmp.left;
    }

    return tmp;
}
```

Using a pointer, move down to the right side until pointer has no right node. Return the pointer.

```java
no usages   new *
public TreeNode iterativeMaxSearch(){
    TreeNode tmp = this;

    while (tmp.right != null){
        tmp = tmp.right;
    }
    return tmp;
}
```

recursive min / max search (node class)

Base case : when left branch of node is null, return that node.

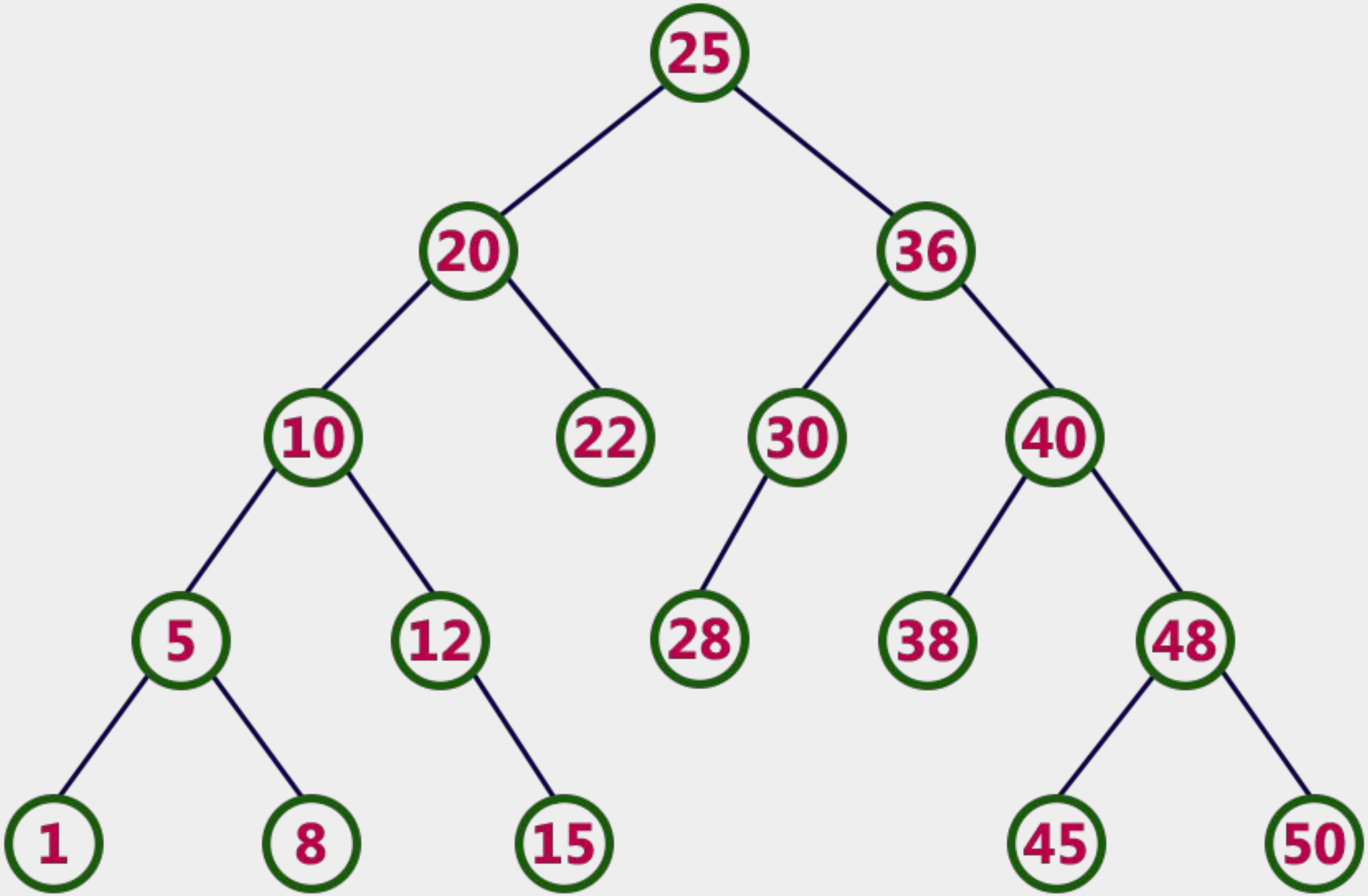
If there is a left branch, recall function on that subtree.

```java
public TreeNode recursiveMinSearch(){

    if (left == null){
        return this;
    } else {
        return left.recursiveMinSearch();
    }
}

1 usage  new *
public TreeNode recursiveMaxSearch(){

    if (right == null){
        return this;
    } else {
        return right.recursiveMaxSearch();
    }
}
```

Base case : when right branch of node is null, return that node.

If there is a right branch, recall function on that subtree.

insertion


25
20
36
10
22
30
40
5
12
28
38
48
1
8
15
45
50

This code inserts at the leaves.

Create two pointers: one for iteration and one to hold the parent information.

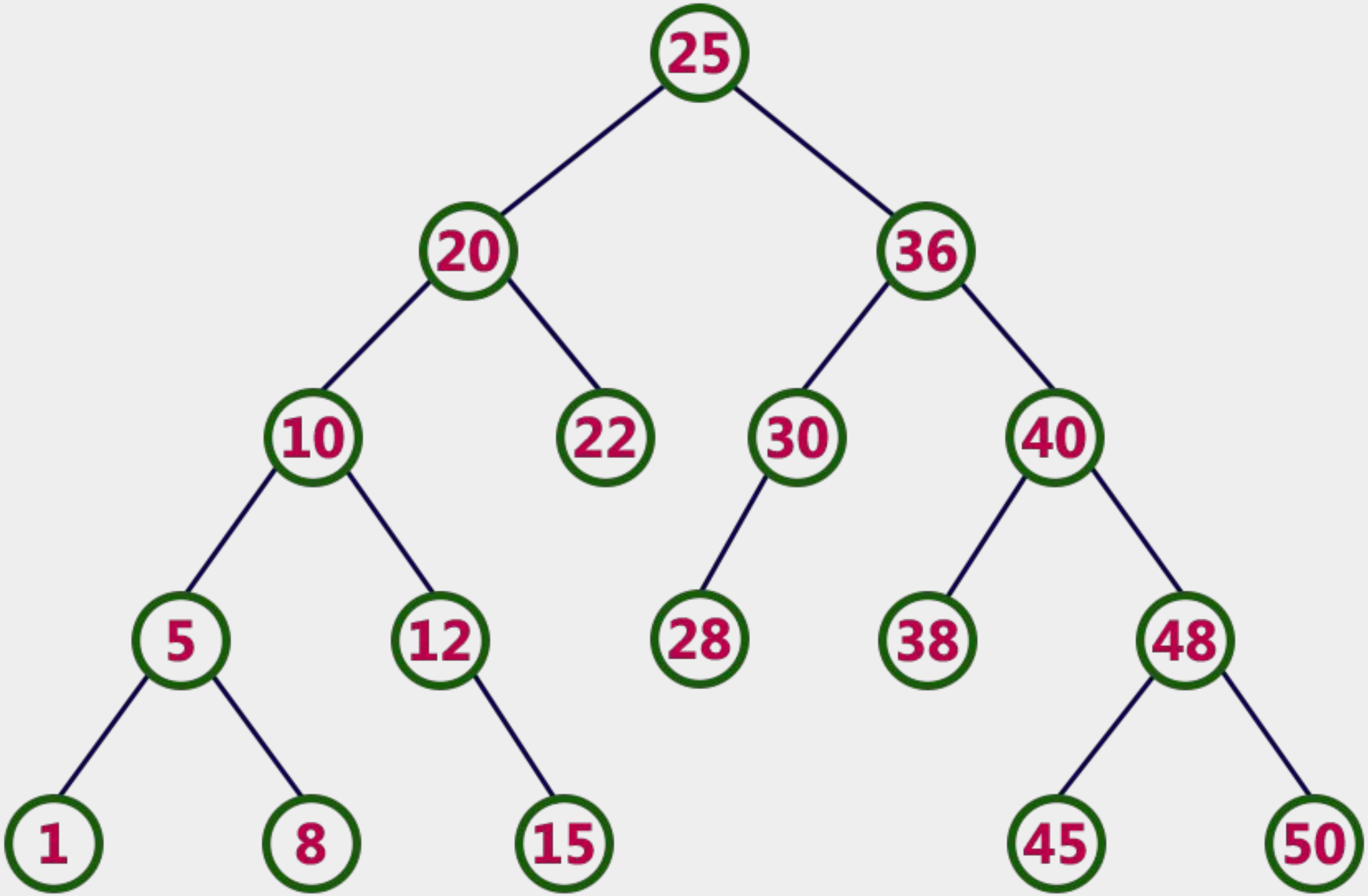
Until the pointer reaches null, shift it to right or left according to the data.

If BST is empty (never entered loop), newNode is the root.

If newNode is greater than parent, put it to the right, else put it to the left.

```java
public void insert(TreeNode newNode){
    TreeNode parent = null;
    TreeNode tmp = root;


    while (tmp != null){
        parent = tmp;

        if (newNode.data > tmp.data){
            tmp = tmp.right;
        } else {
            tmp = tmp.left;
        }
    }

    if (parent == null){
        root = newNode;
    } else if (newNode.data > parent.data){
        parent.right = newNode;
    } else {
        parent.left = newNode;
    }
}
```

deletion


25
20
36
10
22
30
40
5
12
28
38
48
1
8
15
45
50

Find node to be deleted and its parent

Create two pointers, deleted node will be at current. Its parent will be at variable called *parent*.

Until current hits null or searched value is found, traverse the list iteratively according to the data vs. value comparison.

If current is not found (is null), return and print that the value is not found.

```java
public void delete(int value) {
    TreeNode parent = null;
    TreeNode current = root;

    // Find the node to be deleted and its parent
    while (current != null && current.data != value) {
        parent = current;
        if (value < current.data) {
            current = current.left;
        } else {
            current = current.right;
        }
    }

    // If the node was not found, return (value not in the tree)
    if (current == null) {
        System.out.println("Value not found in the tree.");
        return;
    }
```

Case 1: Removing leaf nodes

If current (deleted) does not have a branch on either side (right and left are null), do the following:

- If current is the root (the only node), make it null.
- Else if current is on the left branch of its parent, make the parent’s left branch null.

```
// Case 1: Node to be deleted has no children (leaf node)
if (current.left == null && current.right == null) {
    if (current == root) {
        root = null;
    } else if (parent.left == current) {
        parent.left = null;
    } else {
        parent.right = null;
    }
}
```

- Else if current is on the right branch of its parent make the parent’s right branch null.

Case 2 : Removing node with one branch/subtree

If one branch of current in null, find the child branch and create a pointer for it.

If current is the root, make its only child the root.

Else if current is the left child of its parent, link the left of parent to current’s only child.

Else if current is the right child of its parent, link the right to the parent to current’s only child.

```java
// Case 2: Node to be deleted has only one child
else if (current.left == null || current.right == null) {
    // get the child of the deleted node
    TreeNode child;
    if (current.left != null) {
        child = current.left;
    } else {
        child = current.right;
    }

    // if the deleted node is the root, make the only child the root
    // else find on which side and delete the child
    if (current == root) {
        root = child;
    } else if (parent.left == current) {
        parent.left = child;
    } else {
        parent.right = child;
    }
}
```

Case 3: Removing node with two branches/subtrees

If current has two branches, create two pointers to find the its smallest successor (smallest node of its right branch).

Move both pointers until *successor* shows the replacement and *successorParent* shows its parent.

Change the data of the deleted node.

If successor is found on the left of its parent, link the children to its left. Else, link them to its right.

```java
    // Case 3: Node to be deleted has two children
    else {
        TreeNode successorParent = current;
        TreeNode successor = current.right;

        // Find the in-order successor (smallest node in the right subtree)
        while (successor.left != null) {
            successorParent = successor;
            successor = successor.left;
        }

        // Replace current node's data with successor's data
        current.data = successor.data;

        // Delete the successor node
        if (successorParent.left == successor) {
            successorParent.left = successor.right;
        } else {
            successorParent.right = successor.right;
        }
    }
}
```

traversals

recursive traversals

preorder (*parent, left child, right child*)

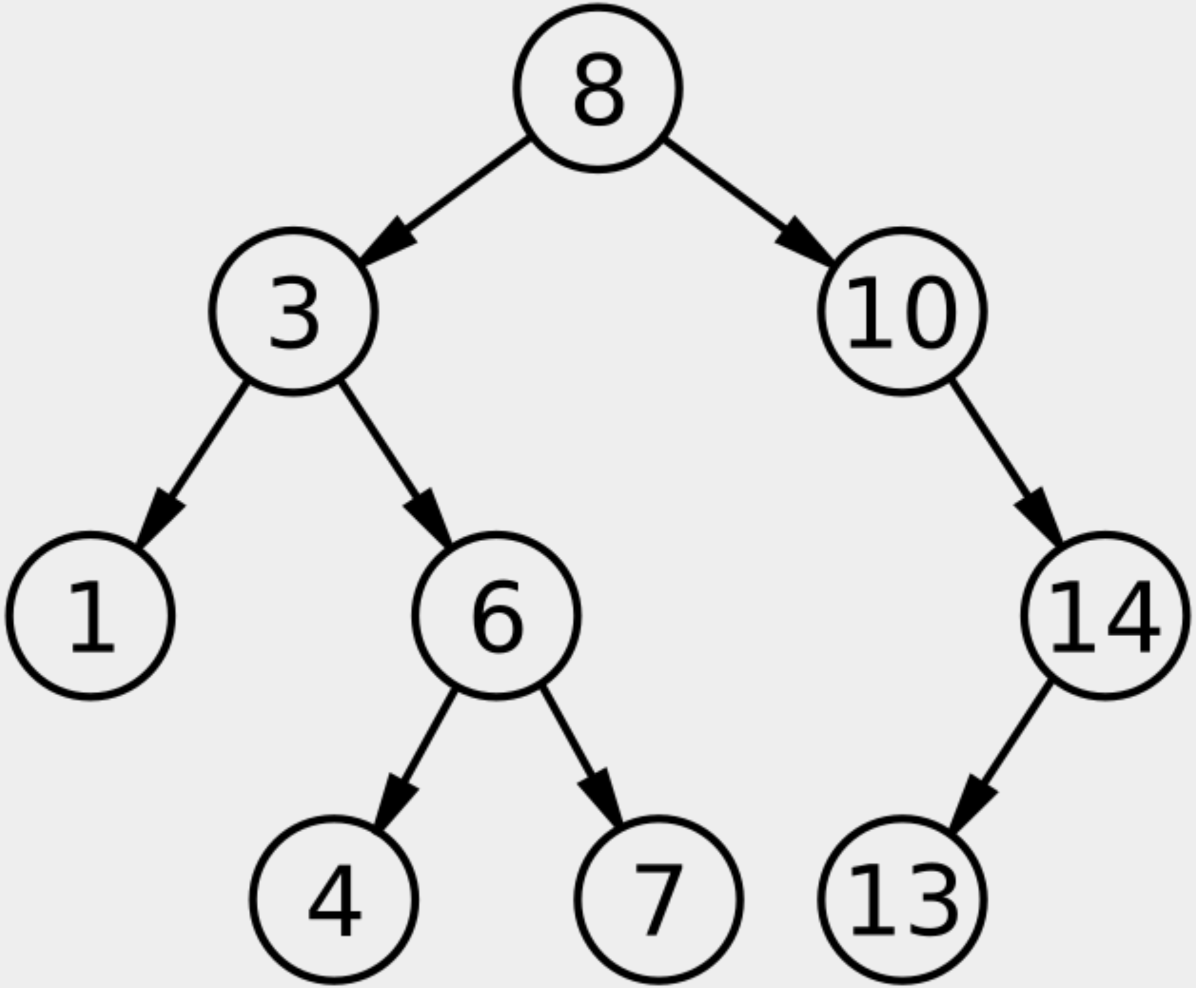
inorder (*left child, parent, right child*)

postorder (*left child, right child, parent*)

```java
public void preorder(){
    System.out.println(data);

    if (left != null){
        left.preorder();
    }
    if (right != null){
        right.preorder();
    }
}
```

```java
public void postorder(){
    if (left != null){
        left.postorder();
    }
    if (right != null){
        right.postorder();
    }
    System.out.println(data);
}
```

```java
public void inorder(){
    if (left != null){
        left.inorder();
    }
    System.out.println(data);
    if (right != null){
        right.inorder();
    }
}
```


8
3
10
1
6
14
4
7
13

non-recursive traversals

counting nodes with stack

Create external stack and push the root on the stack

Until the stack is empty, pop the added node and push its immediate children in the stack while counting every element that you pop

Return the count

```java
public int nodeCountStack(){
    if (root == null){
        return 0;
    }

    Stack<TreeNode> externalStack = new Stack<>();
    externalStack.push(root);

    int count = 0;
    while (!externalStack.isEmpty()){
        TreeNode node = externalStack.pop();
        count++;

        if (node.left != null){
            externalStack.push(node.left);
        }
        if (node.right != null){
            externalStack.push(node.right);
        }
    }
    return count;
}
```

counting nodes with queue

Create external queue as linked list implementation and offer/enqueue the root

Until the queue is empty, poll/dequeue the added node and offer/enqueue its immediate children while counting every element that you dequeue

Return the count

```java
public int nodeCountQueue(){
    if (root == null){
        return 0;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int count = 0;

    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();
        count++;

        if (node.left != null) {
            queue.offer(node.left);
        }
        if (node.right != null) {
            queue.offer(node.right);
        }
    }
    return count;
}
```