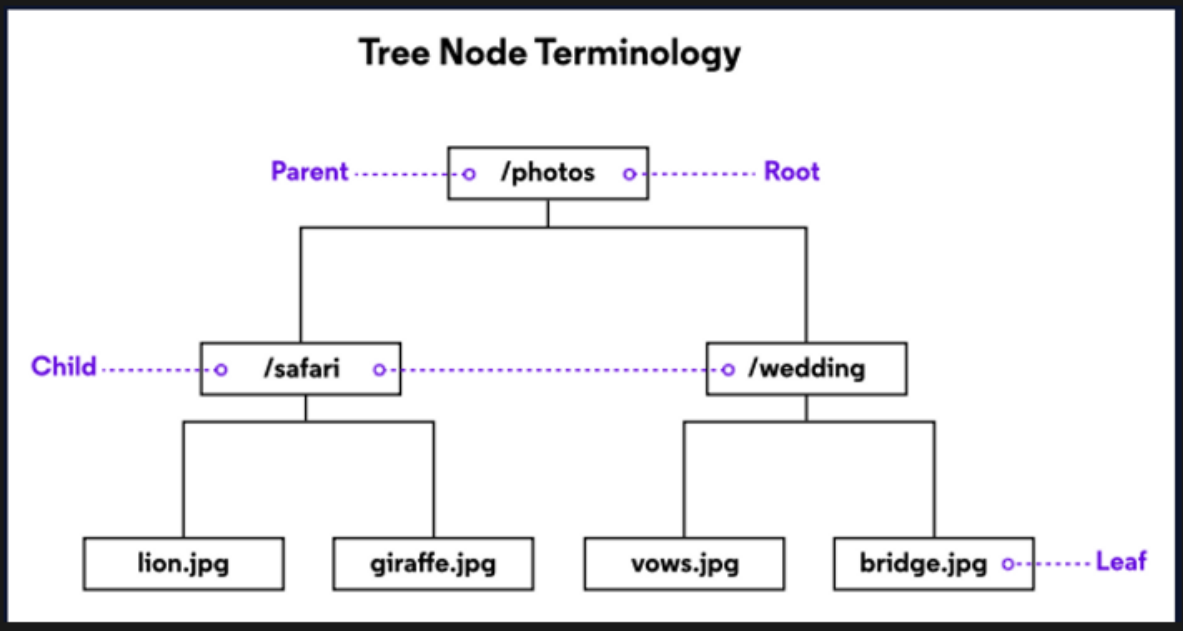
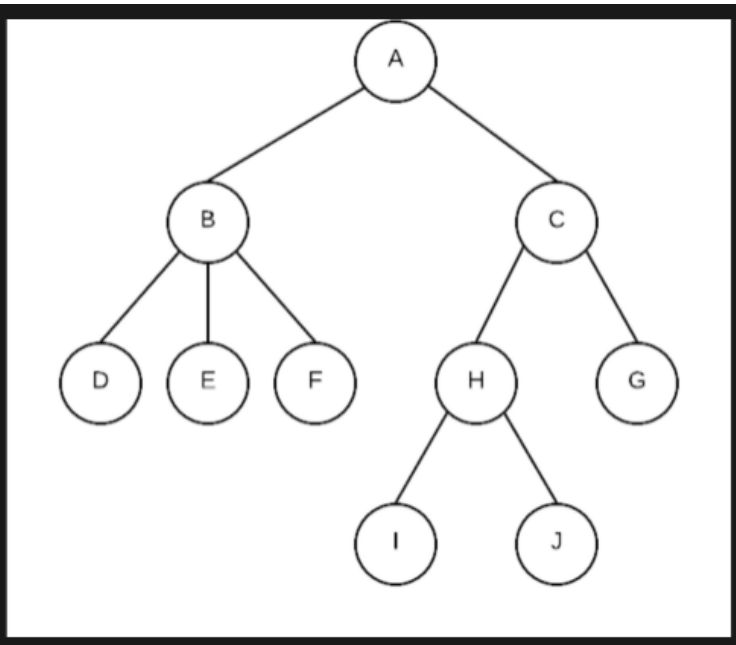
008 - Trees

Trees are an essential data structure for storing hierarchical data with a directed flow. Similar to linked lists and graphs, trees are composed of nodes which hold data. The diagram represents nodes as rectangles and data as text.



Tree Terminology:



Node: The thing that stores data within the tree. (each circle in the above diagram is a node)

Root Node: The top most node from which all other nodes come from. A is the root node of the tree.

Subtree: Some portion of the entire tree, includes a node (the root of the subtree) and every node that goes downwards from there. A is the root of the entire tree. B is the root of the subtree containing B,D,E and F.

Empty trees: A tree with no nodes.

Leaf Node: A node with only empty subtrees (no children) Ex. D,E,F,I,J,and G are all leaf nodes.

Children: The nodes that atr directly 1 link down from a node is that node's child node. Ex. B is the child of A. I is the child of H.

Parent: The node that is directly 1 link up from a node. Ex. A is parent of B. H is the parent of I.

Sibling: All nodes that have the same parent node are siblings Ex. E and F are siblings of D but H is not.

Ancestor: All nodes that can be reached by moving only in an upward direction in the tree. Ex. C, A and H are all ancestors of I but G and B are not.

Descendants or Successors: of a node are nodes that can be reached by only going down in the tree. Ex. Descendants of C are G,H,I and J

Depth: Distance from root node of tree. Root node is at depth 0. B and C are at depth 1. Nodes at depth 2 are D,E,F,G and H. Nodes at depth 3 are I and J

Height: Total number of nodes from root to furthest leaf. Our tree has a height of 4.

Path: Set of branches taken to connect an ancestor of a node to the node. Usually described by the set of nodes encountered along the path.

Binary tree: A binary tree is a tree where every node has 2 subtrees that are also binary trees. The subtrees may be empty. Each node has a left child and a right child. Our tree is NOT a binary tree because B has 3 children.

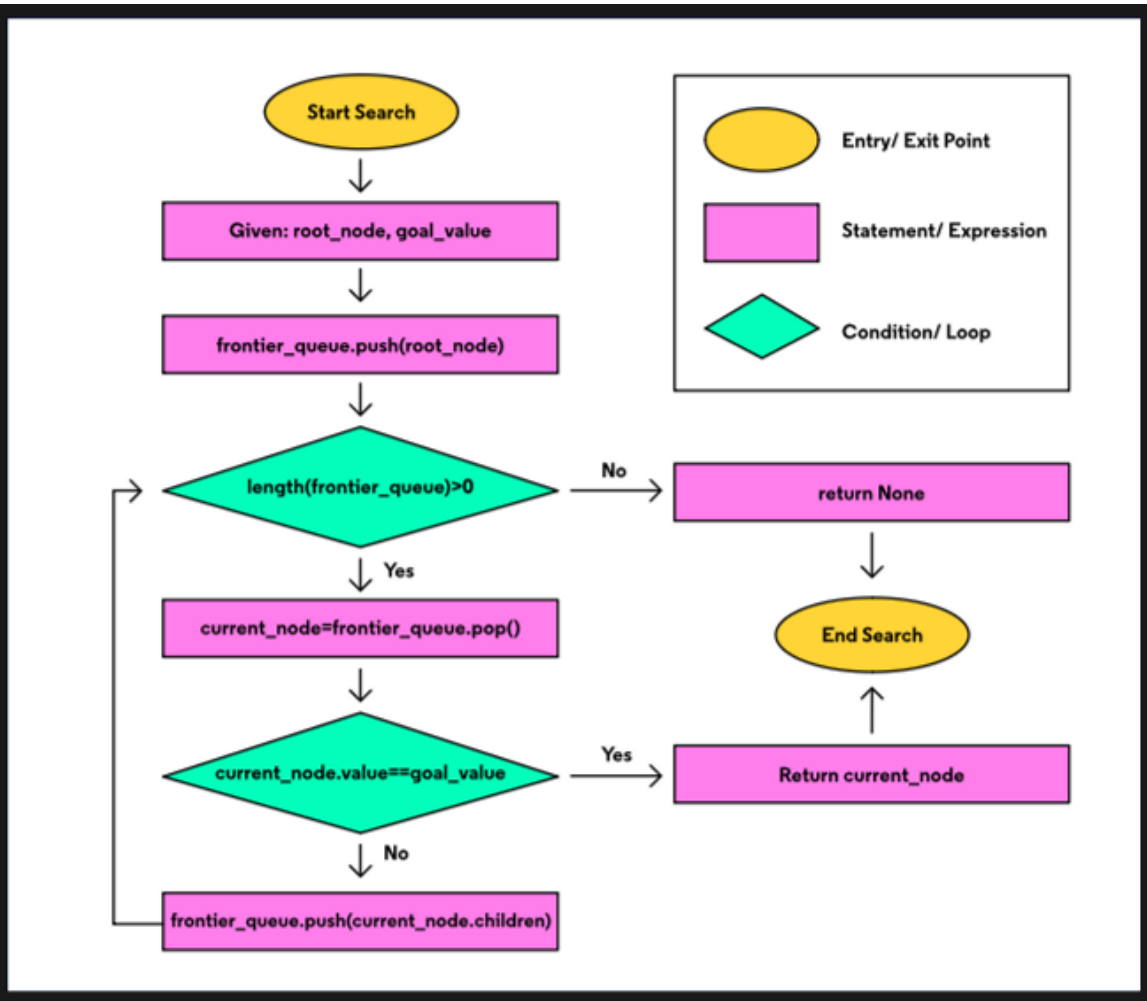
Basic Tree Implementation

```
class TreeNode:
  def __init__(self, value):
    self.value = value # data
    self.children = [] # references to other nodes

  def add_child(self, child_node):
    # creates parent-child relationship
    print("Adding " + child_node.value)
    self.children.append(child_node)

  def remove_child(self, child_node):
    # removes parent-child relationship
    print("Removing " + child_node.value + " from " + self.value)
    self.children = [child for child in self.children
                     if child is not child_node]

  def traverse(self):
    # moves through each node referenced from self downwards
    nodes_to_visit = [self]
    while len(nodes_to_visit) > 0:
      current_node = nodes_to_visit.pop()
      print(current_node.value)
      nodes_to_visit += current_node.children
```

Breadth First Search

A *breadth-first search* is when you inspect every node on a level starting at the top of the tree and then move to the next level.

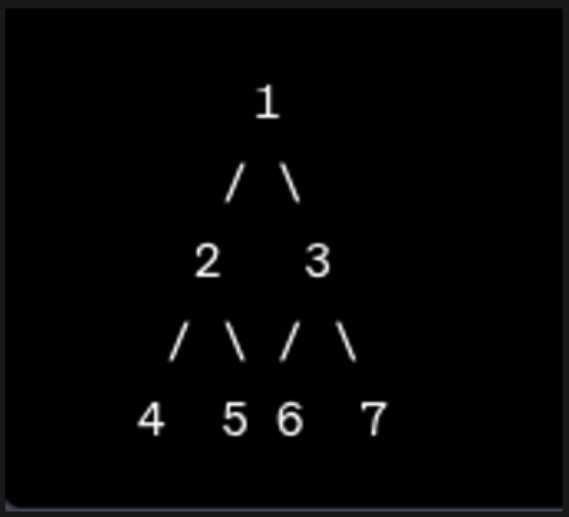
Storing the frontier nodes in a queue creates the level-by-level pattern of a breadth-first search. Child nodes are searched in the order they are added to the frontier. The nodes on the next level are always behind the nodes on the current level. Breadth-first search is known as a complete algorithm since no matter how deep the goal is in the tree it will always be located.



Tracking the path to the goal node:

- Define an initial path with the root node inside a list data type.
- Push the initial path onto the frontier queue.
- Pop the current path off the frontier.
- Retrieve the current node from the end of the current path.
- Define a new path from a copy of the current path.
- Add the child node to the end of the new path.
- Push the new path onto the frontier queue.

Example:



1. Start with the root node 1 and goal value 6.
2. Initialize the frontier queue with a path containing just the root node: `[1]`.
3. Begin the BFS loop:
   a. Pop the current path `[1]` from the frontier queue.
   b. Retrieve the current node from the end of the current path: `1`.
   c. Since the current node (1) is not the goal value (6), we add its children (2 and 3) to the frontier queue with updated paths:
   - New path for child 2: `[1, 2]`
   - New path for child 3: `[1, 3]`
4. Next iteration of the BFS loop:
   a. Pop the current path `[1, 2]` from the frontier queue.
   b. Retrieve the current node from the end of the current path: `2`.
   c. Node 2 is not the goal value, so we add its children (4 and 5) to the frontier queue with updated paths:
   - New path for child 4: `[1, 2, 4]`
   - New path for child 5: `[1, 2, 5]`
5. Next iteration of the BFS loop:
   a. Pop the current path `[1, 3]` from the frontier queue.
   b. Retrieve the current node from the end of the current path: `3`.
   c. Node 3 is not the goal value, so we add its children (6 and 7) to the frontier queue with updated paths:
   - New path for child 6: `[1, 3, 6]`
   - New path for child 7: `[1, 3, 7]`

6. Next iteration of the BFS loop:
   a. Pop the current path `[1, 2, 4]` from the frontier queue.
   b. Retrieve the current node from the end of the current path: `4`.
   c. Node 4 is the goal value! We have found the goal node. The current path `[1, 2, 4]` represents the path from the root node to the goal node.
7. We can now return the path `[1, 2, 4]` as the path from the root node to the goal node 6.

```
from collections import deque

class TreeNode:
  def __init__(self, value):
   self.value = value
   self.children = []

  def __str__(self):
    stack = deque()
    stack.append([self, 0])
    level_str = "\n"
    while len(stack) > 0:
      node, level = stack.pop()
      if level > 0:
        level_str += "| "*(level-1)+ "|-"
      level_str += str(node.value)
      level_str += "\n"
      level+=1
      for child in reversed(node.children):
        stack.append([child, level])

    return level_str

# Breadth-first search function
def bfs(root_node, goal_value):
  path_queue = deque()
  initial_path = [root_node]
  path_queue.appendleft(initial_path)
  while path_queue:
    current_path = path_queue.pop()
    current_node = current_path[-1]
    print(f"Searching node with value: {current_node.value}")

    if current_node.value == goal_value:
      return current_path

    for child in current_node.children:
      new_path = current_path[:]
      new_path.append(child)
      path_queue.appendleft(new_path)

  return None
```

Depth First Search

A *depth-first search* is where you search deep into a branch and don't move to the next one until you've reached the end.

Frontier nodes stored in a stack create the deep dive of a depth-first search. Nodes added to the frontier early on can expect to remain in the stack while their sibling's children (and their children, and so on) are searched. Depth-first search is not considered a complete algorithm since searching an infinite branch in a tree can go on forever. In this situation, an entire section of the tree would be left un-inspected.

Recursive Implementation

The recursive version of the algorithm works by starting at the root node, and breaking the tree up into subtrees, until it finds the target node, or until every node in the tree has been considered as the root of a subtree. We recursively call the function on all of our root's children, treating each child node as a root of its own subtree. We define a function that accepts a tree node and a target value as input parameters. The recursive DFS algorithm implements the following logic:

- If the input node value matches our target value then return the input node.
- For each child of the input node, recursively call this function and return the first non-null value returned by a recursive call.
- If this root node has no children, or the recursive calls did not return any node, then return null.

To search a tree with this function, we invoke the function with the root node of our tree.

Iterative Implementation

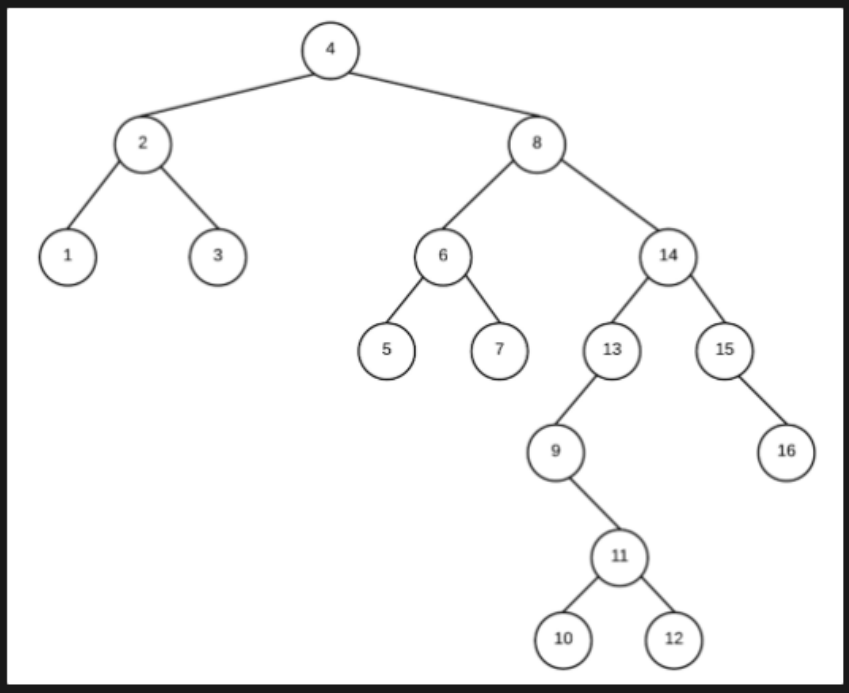
The iterative algorithm does not make use of any recursive calls. Instead, we maintain a stack of references to unexplored siblings of the nodes we have already accessed. The recursive algorithm is effectively doing something very similar, but the program call stack is implicitly used to store the path from the root to the current node. With the iterative algorithm, we need to implement a stack ourselves. These two implementations have the same time and space complexity, so the choice of which to implement is usually a matter of personal preference.

Complexity

Depth-First Search has a time complexity of `O(n)` where n is the number of nodes in the tree. In the worst case, we will examine every node of a tree.

Depth-First Search has a space complexity of `O(n)` where n is the number of nodes in the tree. In the worst case, we will need to store a reference to every node in a stack. Consider an adversarial example of a linked list as a type of tree with no branching. Trees like this could reach the worst-case space complexity if the target node is the leaf node or is absent from the tree altogether.

With depth-first traversal, we always traverse down each left-side branch of a tree fully before proceeding down the right branch. However, there are three traversal options:



- *Preorder* is when we perform an action on the current node first, followed by its left child node and its right child node

> 4, 2, 1, 3, 8, 6, 5, 7, 14, 13, 9, 11, 10, 12, 15, 16

- *Inorder* is when we perform an action on the left child node first, followed by the current node and the right child node. Notice that this type of traversal results in values being listed in its sorted order

> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

- *Postorder* is when we perform an action on the left child node first, followed by the right child node and then the current node. This is the type of traversal you would use to destroy a tree.

> 1, 3, 2, 5, 7, 6, 10, 12, 11, 9, 13, 16, 15, 14, 8, 4

```
from TreeNode import TreeNode, sample_root_node, print_path, print_tree

print_tree(sample_root_node)

def dfs(root, target, path=()):
  path = path + (root,)

  if root.value == target:
    return path

  for child in root.children:
    path_found = dfs(child, target, path)

    if path_found is not None:
      return path_found

  return None

node = dfs(sample_root_node, "F")
print(node)
```

Divide and Conquer Algorithms

A divide-and-conquer algorithm solves a large problem by recursively breaking it down into smaller subproblems until they become simple enough to be solved directly.

The divide-and-conquer strategy is a three-step process:

1. Divide: Recursively divide a big problem into smaller subproblems.
2. Conquer: Solve the subproblems after they become small enough.
3. Combine: Merge the subproblems to get the desired solution.

Time & Space Complexities

There is no single time or space complexity that can describe all divide-and-conquer algorithms. However, we can follow these general rules to determine the big-O runtime of a specific algorithm:

- Because the problem is divided into at least 2 subproblems at each step, the number of steps needed to divide the problem is **log(n)**.
- The cost of solving and merging the subproblems is a factor as well. This is often multiplied by the cost of division to get the overall runtime.

For merge sort, division takes **O(log(n))** time, and combining the sorted sublists takes **O(n)** time. Therefore, merge sort has a big-O runtime of **O(nlog(n))**.

The space complexity also varies from algorithm to algorithm. Divide-and-conquer usually requires more space than other paradigms due to the overhead of the recursion stack.

Pros and Cons

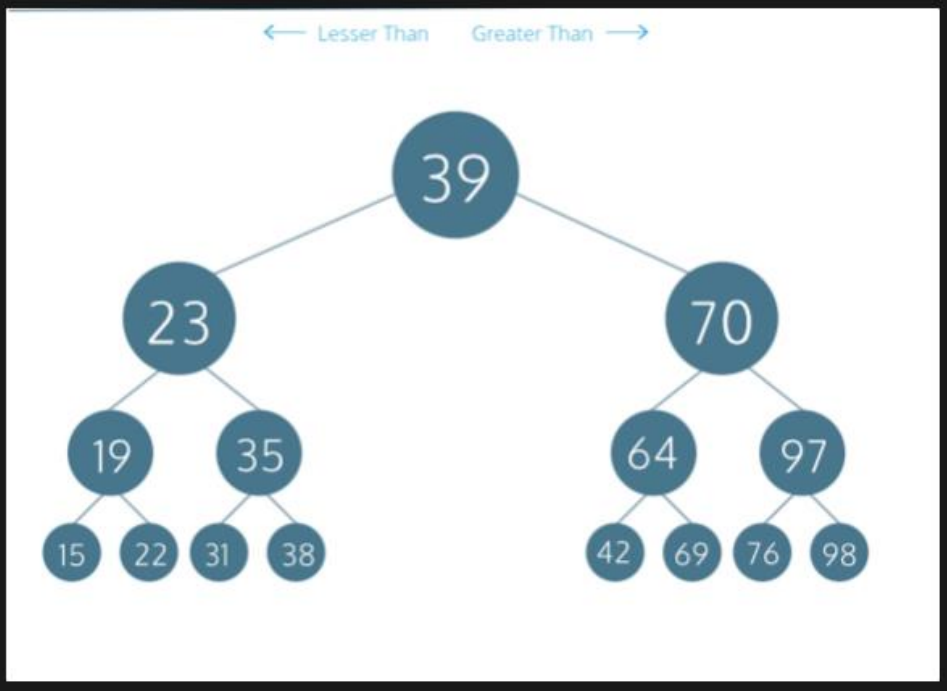
Divide-and-conquer has several advantages:

1. It lets us solve conceptually complex problems by breaking them into smaller subproblems.
2. It has a better asymptotic cost than brute force approaches and is used in many efficient algorithms such as merge sort and quicksort.
3. It makes efficient use of memory caches, which are the fastest type of memory units in a computer.

However, it does have some issues as well:

1. Since divide-and-conquer algorithms are usually implemented recursively, they require additional memory allocation on the stack. If a divide-and-conquer algorithm is executed without sufficient memory, a *stack overflow*error will occur.
2. Divide-and-conquer cannot avoid evaluating the same subproblem repeatedly, making it a bad fit for problems that have overlapping subproblems.

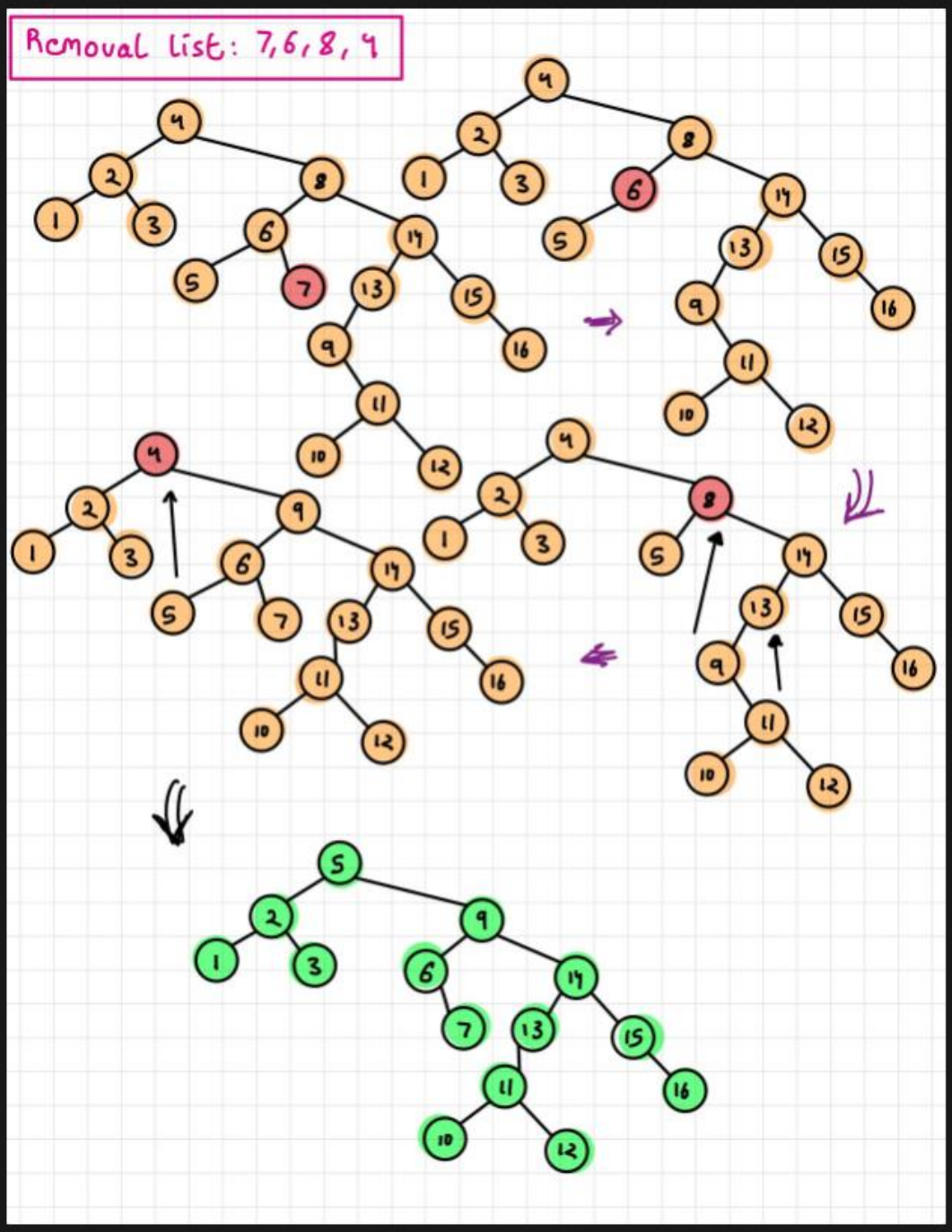
Binary Search Trees



Code for Recursive BST:

```
class BinarySearchTree:
  def __init__(self, value, depth=1):
    self.value = value
    self.depth = depth
    self.left = None
    self.right = None

  def insert(self, value):
    if (value < self.value):
      if (self.left is None):
        self.left = BinarySearchTree(value, self.depth + 1)
        print(f'Tree node {value} added to the left of {self.value} at depth {se
lf.depth + 1}')
      else:
        self.left.insert(value)
    else:
      if (self.right is None):
        self.right = BinarySearchTree(value, self.depth + 1)
        print(f'Tree node {value} added to the right of {self.value} at depth {s
elf.depth + 1}')
      else:
        self.right.insert(value)

  def get_node_by_value(self, value):
    if (self.value == value):
      return self
    elif ((self.left is not None) and (value < self.value)):
      return self.left.get_node_by_value(value)
    elif ((self.right is not None) and (value >= self.value)):
      return self.right.get_node_by_value(value)
    else:
      return None

  # Define .depth_first_traversal() below:
  # In order traversal
  def depth_first_traversal(self):
    if (self.left is not None):
      self.left.depth_first_traversal()
    print(f'Depth={self.depth}, Value={self.value}')
    if (self.right is not None):
      self.right.depth_first_traversal()
```

Binary Search Tree Deletion



Binary Search Tree Implementation

```
import queue

class BST:
    class Node:
        def __init__(self, data=None, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    def __init__(self):
        self.root = None

    def insert(self, data):
        if self.root is None:
            self.root = self.Node(data)
        else:
            curr = self.root
            inserted = False

            while not inserted:
                if data < curr.data:
                    if curr.left is not None:
                        curr = curr.left
                    else:
                        curr.left = self.Node(data)
                        inserted = True
                else:
                    if curr.right is not None:
                        curr = curr.right
                    else:
                        curr.right = self.Node(data)
                        inserted = True

    def search(self, data):
        curr = self.root
        while curr is not None:
            if data < curr.data:
                curr = curr.left
            elif data > curr.data:
                curr = curr.right
            else:
                return curr
        return None
```

```
    def r_search(self, data, subtree):
        if subtree is None:
            return None
        else:
            if data < subtree.data:
                return self.r_search(data, subtree.left)
            elif data > subtree.data:
                return self.r_search(data, subtree.right)
            else:
                return subtree

    def search_recursive(self, data):
        return self.r_search(data, self.root)

    def r_insert(self, data, subtree):
        if subtree is None:
            return self.Node(data)
        else:
            if data < subtree.data:
                subtree.left = self.r_insert(data, subtree.left)
                return subtree
            else:
                subtree.right = self.r_insert(data, subtree.right)
                return subtree

    def insert_recursive(self, data):
        self.r_insert(data, self.root)

    def inorder_print(self, subtree):
        if subtree is not None:
            self.inorder_print(subtree.left)
            print(subtree.data, end=" ")
            self.inorder_print(subtree.right)

    def pre_order_print(self, subtree):
        if subtree is not None:
            print(subtree.data, end=" ")
            self.pre_order_print(subtree.left)
            self.pre_order_print(subtree.right)

    def post_order_print(self, subtree):
        if subtree is not None:
            self.post_order_print(subtree.left)
            self.post_order_print(subtree.right)
            print(subtree.data, end=" ")
```

```
    def iterative_in_order_print(self):
        stack = []
        curr = self.root
        while stack or curr:
            while curr:
                stack.append(curr)
                curr = curr.left
            curr = stack.pop()
            print(curr.data, end=" ")
            curr = curr.right

    def iterative_pre_order_print(self):
        stack = [self.root]
        while stack:
            curr = stack.pop()
            if curr:
                print(curr.data, end=" ")
                stack.append(curr.right)
                stack.append(curr.left)

    def iterative_post_order_print(self):
        stack1 = [self.root]
        stack2 = []
        while stack1:
            curr = stack1.pop()
            if curr:
                stack2.append(curr)
                stack1.append(curr.left)
                stack1.append(curr.right)
        while stack2:
            print(stack2.pop().data, end=" ")

    def breadthfirst_print(self):
        if self.root is None:
            print([])
            return
        nodes_to_visit = queue.Queue()
        nodes_to_visit.put(self.root)
        while not nodes_to_visit.empty():
            curr_node = nodes_to_visit.get()
            if curr_node.left:
                nodes_to_visit.put(curr_node.left)
            if curr_node.right:
                nodes_to_visit.put(curr_node.right)
            print(curr_node.data, end=" ")
```

```
    def delete(self, data):
        curr = self.root
        parent = None
        while curr is not None:
            if data < curr.data:
                parent = curr
                curr = curr.left
            elif data > curr.data:
                parent = curr
                curr = curr.right
            else:
                if curr.left is None:
                    if parent is None:
                        self.root = curr.right
                    elif parent.left == curr:
                        parent.left = curr.right
                    else:
                        parent.right = curr.right
                elif curr.right is None:
                    if parent is None:
                        self.root = curr.left
                    elif parent.left == curr:
                        parent.left = curr.left
                    else:
                        parent.right = curr.left
                else:
                    successor_parent = curr
                    successor = curr.right
                    while successor.left is not None:
                        successor_parent = successor
                        successor = successor.left
                    curr.data = successor.data
                    if successor_parent.left == successor:
                        successor_parent.left = successor.right
                    else:
                        successor_parent.right = successor.right
                return
        return
```

AVL Trees

Binary tees do not always give you a O(log n) time complexity. This is because, sometimes the number might be arranged in a way that the tree looks more like a linked list. In these cases, the time complexity would be O(n). To avoid this, we use augmented data structures.

Height of a tree: Length of the longest path from root to leaf.

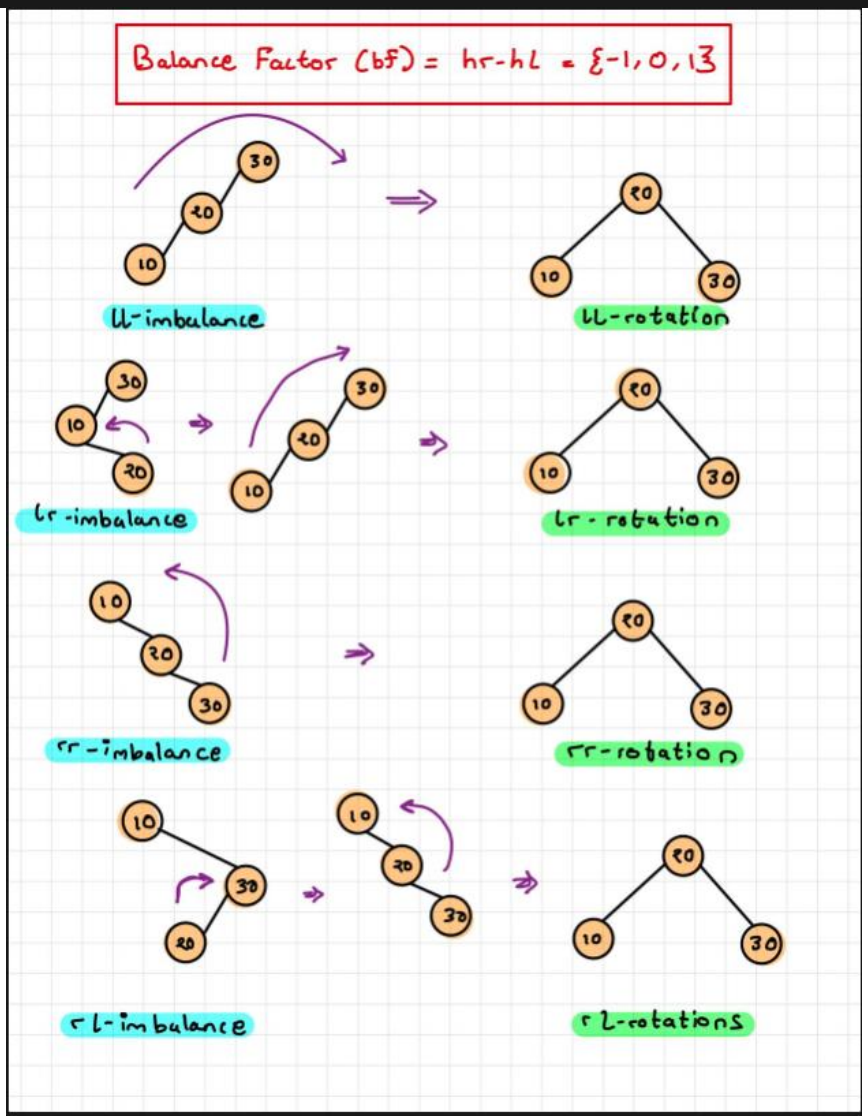
Balanced Tree: The height of a balanced tree is O(log n)

Height of a node: Length of the longest path from the node to a leaf. max(height(left and right)) + 1

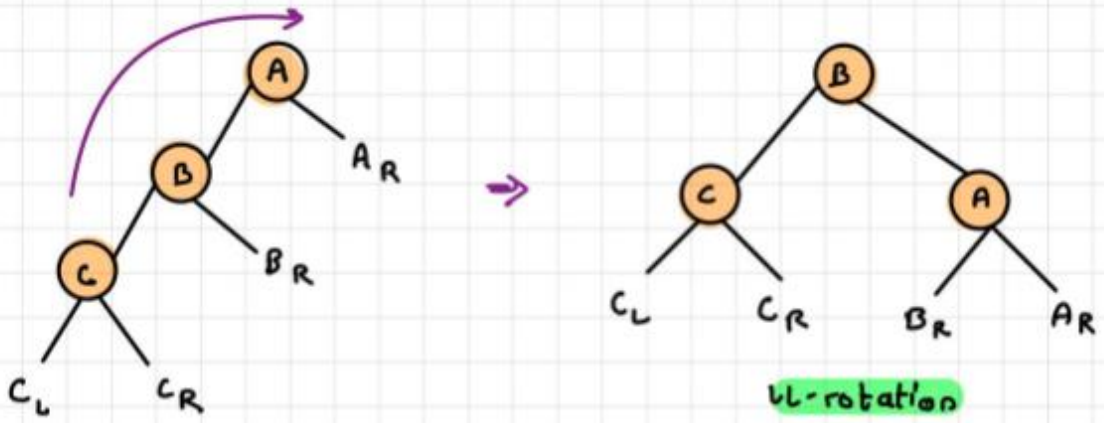
Node balance = Height of right subtree - Height of left subtree

> **AVL trees require heights of left and right children of every node to differ by at most + or - 1.**

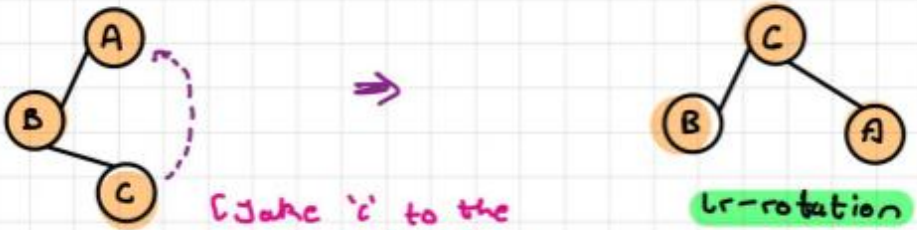
Important points to note is that rotations can only be made with 3 nodes.

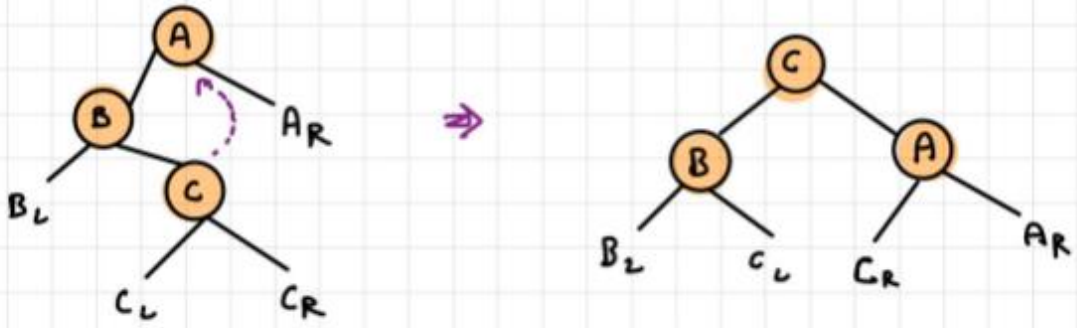


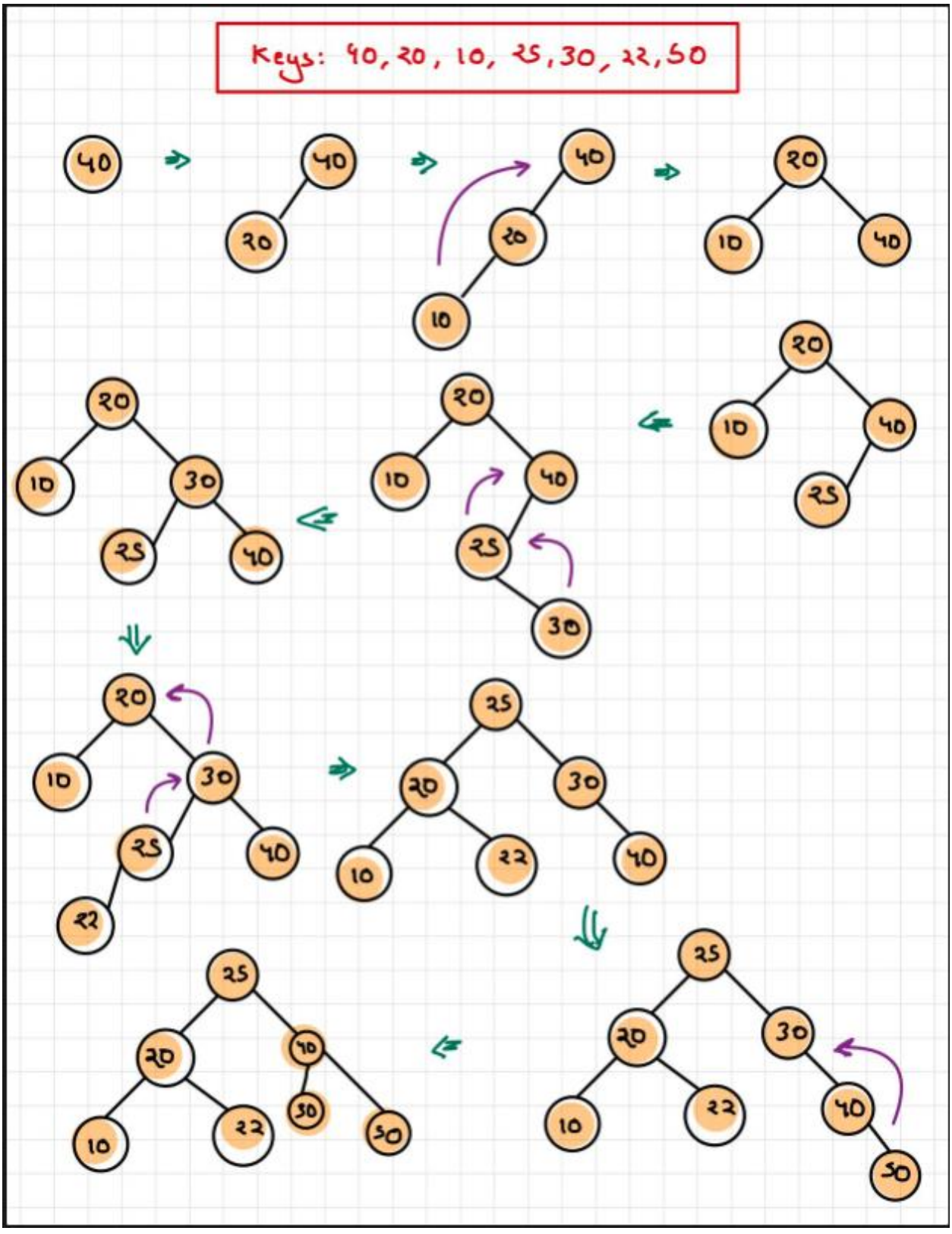
➔ SPECIAL CASES:



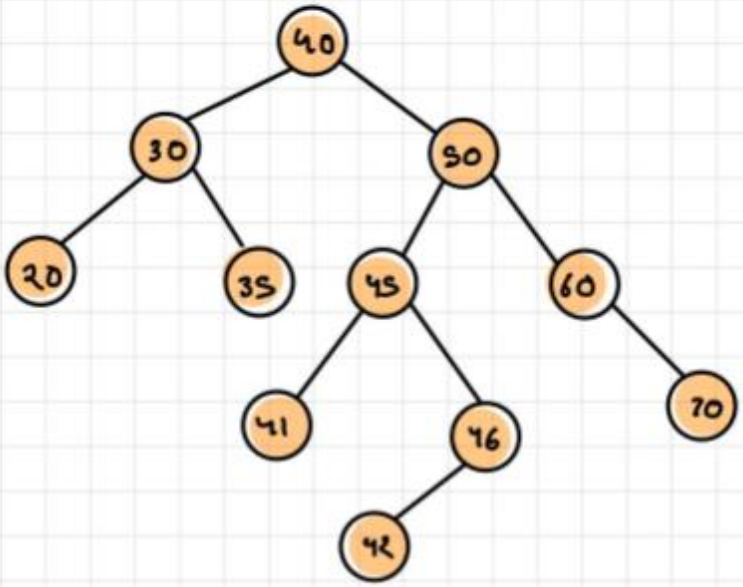
➔ LR Rotation in one step:



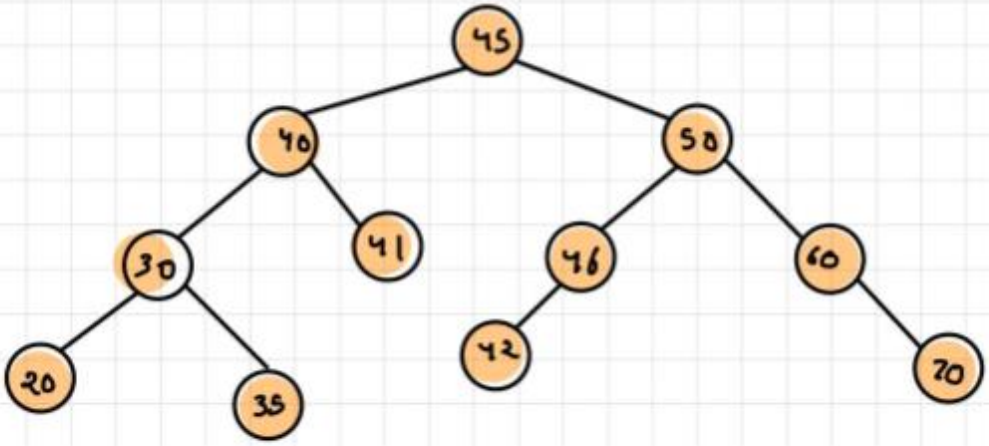



Keys: 40, 20, 10, 25, 30, 22, 50

COMPLEX EXAMPLE :

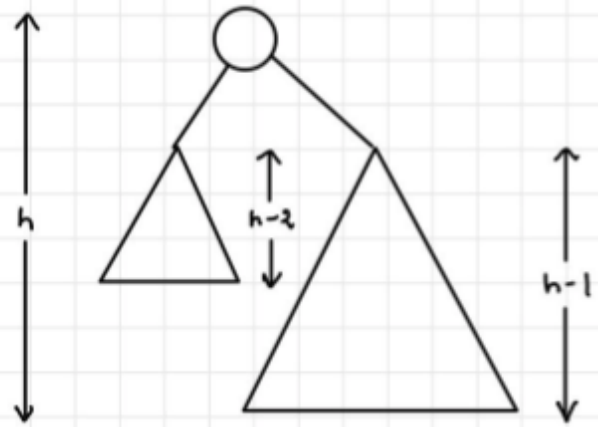


After RL-rotation



⇒ Maximum height of AVL tree:

- Lets consider a scenario where the balance of every node is +1. This scenario is our worse case scenario.



$$N_h = 1 + N_{h-1} + N_{h-2}$$

$N_h > F_h$ (Fibonacci)

$N_h > \frac{\phi^h}{\sqrt{5}}$ → Golden ratio

$\Downarrow$

n

$$\frac{\phi^h}{\sqrt{5}} < n$$

⇒ $\log_\phi$ on both sides

$h - (a \text{ small value}) < \log_\phi n$

$\therefore$ $h < \simeq 1.44 \log_2 n$

OR

$$N_h = 1 + N_{h-1} + N_{h-2}$$

⇒ N_{h-2} is the right way to go:

$$N_h > 1 + 2N_{h-2}$$
$$> 2N_{h-2}$$

⇒ $h < 2\log n$

Red Black Trees

This is also a type of augmented and balanced binary search tree. Some specifications are as follows:

1. A node is either red or black.
2. The root and leaves (NIL) are black.
3. If a node is red, then its children are black.
4. All paths from a node to its NIL descendants conatain the same number of black nodes.
5. There are no consecutive reds.

Notes:

1. Nodes require one storage bit to keep track of color.
2. The longest path (root to farthest NIL) is no more than twice the length of the shortest path.
    a. Shortest Path: All Black Nodes
    b. Longest Path: Alternating red and black nodes
3. The time complexity for search, insert and remove is O(log n).
4. Space complexity is O(log n) too.

Red Black Tree Insertion Procedure:

1. Insert a node and color it red
2. Recolor and rotate the nodes to fix violations
    a. Assume that the node we are inserting is called Z. G = Grandparent, P = Parent, C = Child, PS = Parent Sibling
    b. **If Z is the root** ⇒ Change it to black
    c. **If Z is the leaf node and there are two red nodes in a row:**
        i. **If PS is Black:**
            1. **If G → P → C form a straight line:** zig zig (single) rotation
            2. **If there is a bend:** zig zag rotation (double rotation)
            3. Exchange G's color with the node that took G's spot
        ii. **If PS is Red:** Exchange the colour of the grand parent with its two children.

Red Black Tree Insertion Walkthrough: