Binary Search

Presented by Yasin Ceran

February 12, 2024

Binary Search (Search Array)

Binary search is an efficient way of searching for elements within a sorted array. Typically, we are given an array and an element called the target to search for.

Core Principle

At its core, binary search divides the array in the middle, termed as *mid*, and compares the value at *mid* to the target value.

- If the *mid* value is lower than the target, it eliminates the left half of the array and searches on the right of *mid*.
- If *mid* is higher than the target, the search continues to the left.
- The process repeats until the target is found or determined not to exist in the array.

Variations

- **Search Array:** A sorted array and a target are given; the task is to determine if the target is found in the array.
- **Search Range:** A range of numbers is given without a specific target.

Mechanics of Binary Search

Understanding the operational steps of Binary Search:

- **Objective:** Efficiently search for a target element within a sorted array.

Key Variables

- **L (Low):** The left-most index of the current subarray.
- **R (High):** The right-most index of the current array.
- **Mid:** Calculated as $L + \frac{R-L}{2}$. It's the index dividing the current sub-array into two halves.

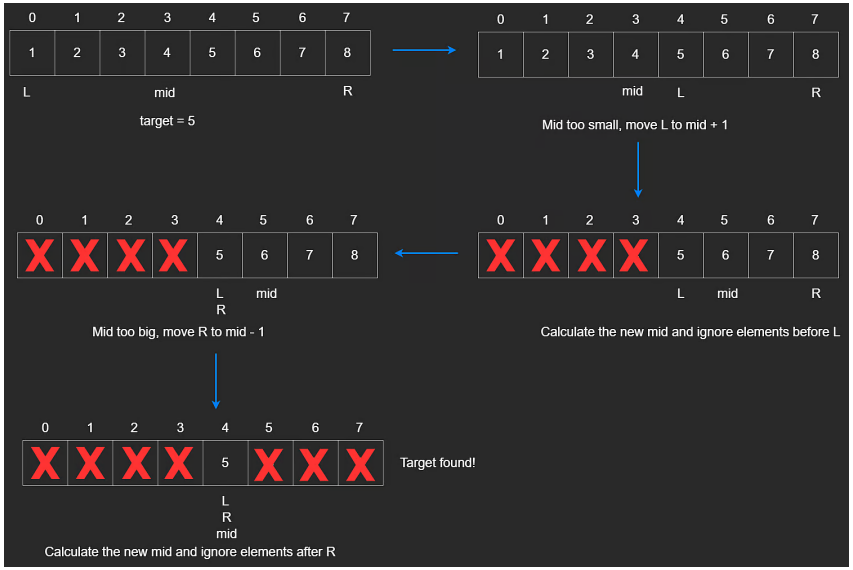
Process

- Begin with the entire array as the initial subarray.
- Calculate *mid* and compare the value at *mid* with the target.
- If *mid* value is less than the target, search the right subarray ($L = mid + 1$).
- If *mid* value is greater, search the left subarray ($R = mid - 1$).
- Repeat until the target is found or L exceeds R, indicating the target is not present.

Binary Search Example: Target Exists in the Array

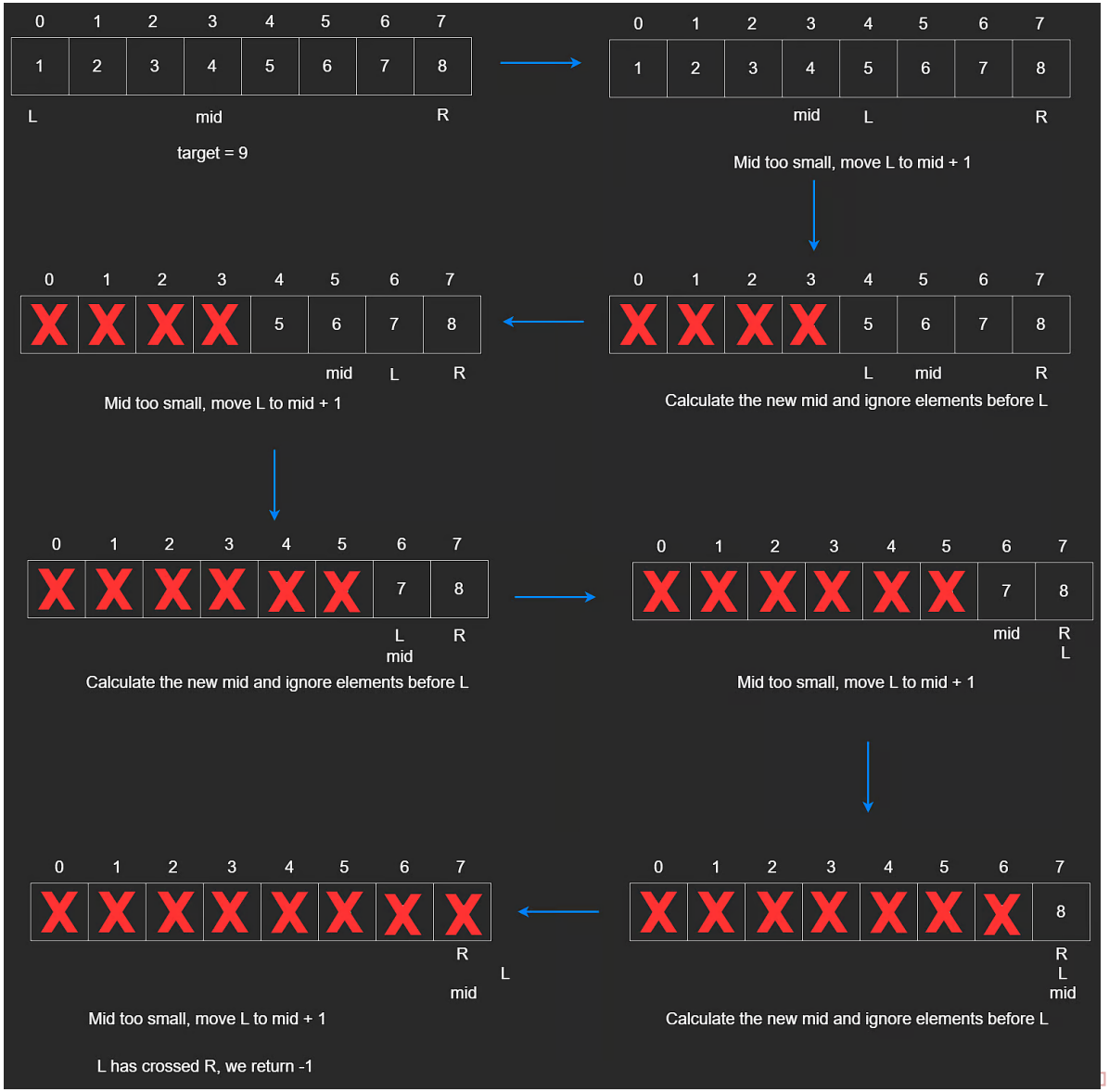
Consider the array arr = [1,2,3,4,5,6,7,8], with the target value 5.

```
arr = [1, 3, 3, 4, 5, 6, 7, 8]

def binarySearch(arr, target):
    L, R = 0, len(arr) - 1

    while L <= R:
        mid = (L + R) // 2

        if target > arr[mid]:
            L = mid + 1
        elif target < arr[mid]:
            R = mid - 1
        else:
            return mid
    return -1
```



Binary Search Example: Target does not exist in the Array

Consider the array `arr` = [1,2,3,4,5,6,7,8], with the target value 9.



Time and Space Complexity of Binary Search

The efficiency of Binary Search lies in its approach to halve the search space with each iteration. This method is similar to the division process used in the Merge Sort algorithm but applied to searching rather than sorting.

Time Complexity

The time complexity of Binary Search is $O(\log n)$. This is because with each comparison, the algorithm reduces the search space by half, leading to a logarithmic number of steps to find the target or conclude its absence.

Space Complexity

Considering the running time of binary search, we observe that a constant number of operations are executed at each recursive call. Hence, the running time is proportional to the number of recursive calls performed. Binary Search is performed in place, requiring no additional storage that scales with the input size. Therefore, its space complexity is $O(1)$, assuming the search is implemented iteratively. Recursive implementations may have a space complexity of $O(\log n)$ due to the call

Concept

Imagine you're given a range from 1 to 100 and asked to guess a number someone is thinking of. There are three possible outcomes for each guess:

1. Your guess is correct.
2. Your guess is too small–you need to guess higher
3. Your guess is too large–you guess lower

Setting Boundaries

To efficiently find the number, we apply the binary search technique:

- Start with the full range as your search space.
- Make a guess in the middle of your current range.
- Use the feedback ("too small" or "too big") to adjust the range.
- Repeat the process until you guess the correct number.

This method significantly reduces the number of guesses needed by halving the search space with each guess.

Pseudocode

```
# Return 1 if n is too big, -1 if too small, 0 if correct
def isCorrect(n):
    if n > 10:
        return 1
    elif n < 10:
        return -1
    else:
        return 0

# Binary search on some range of values
def binarySearch(low, high):

    while low <= high:
        mid = (low + high) // 2

        if isCorrect(mid) > 0:
            high = mid - 1
        elif isCorrect(mid) < 0:
            low = mid + 1
        else:
            return mid
    return -1
```

The work being done is the same as the previous section, which is standard binary search procedure. Therefore, this procedure is also in $O(\log n)$

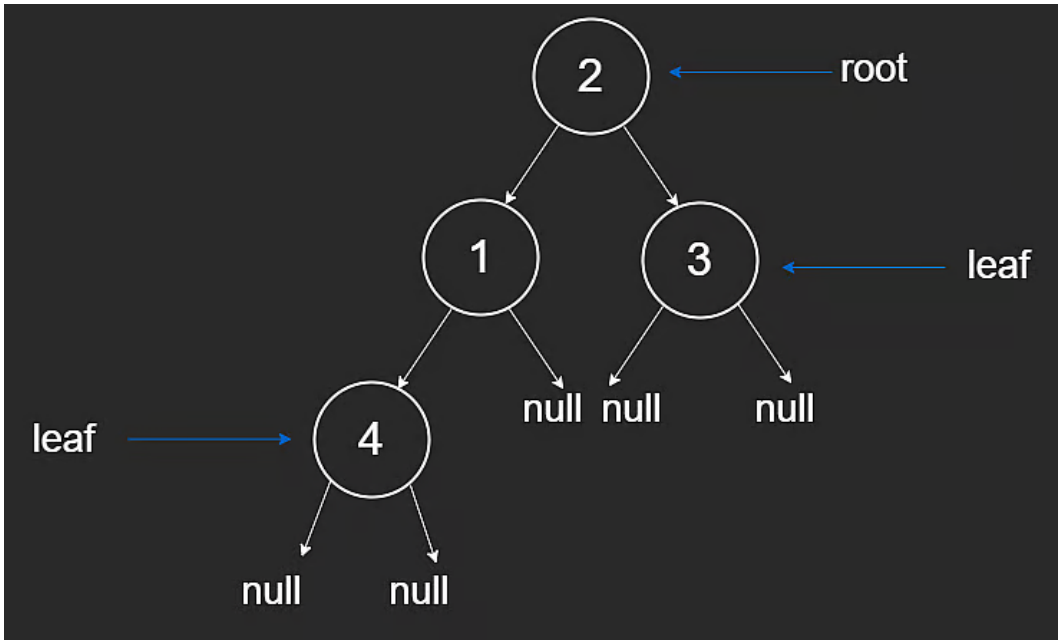
Concept

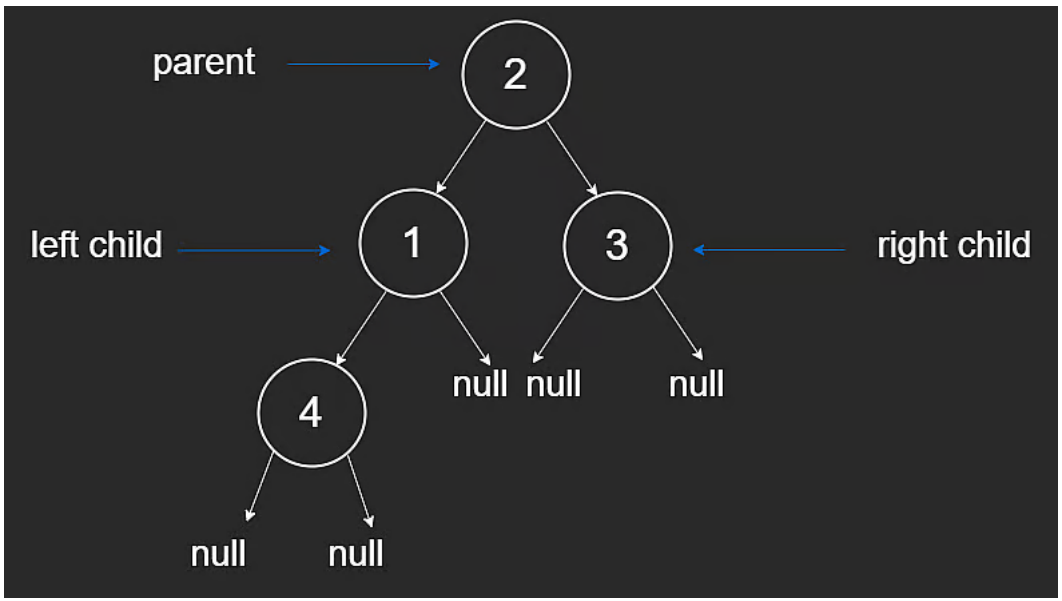
TreeNode Implementation

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
```

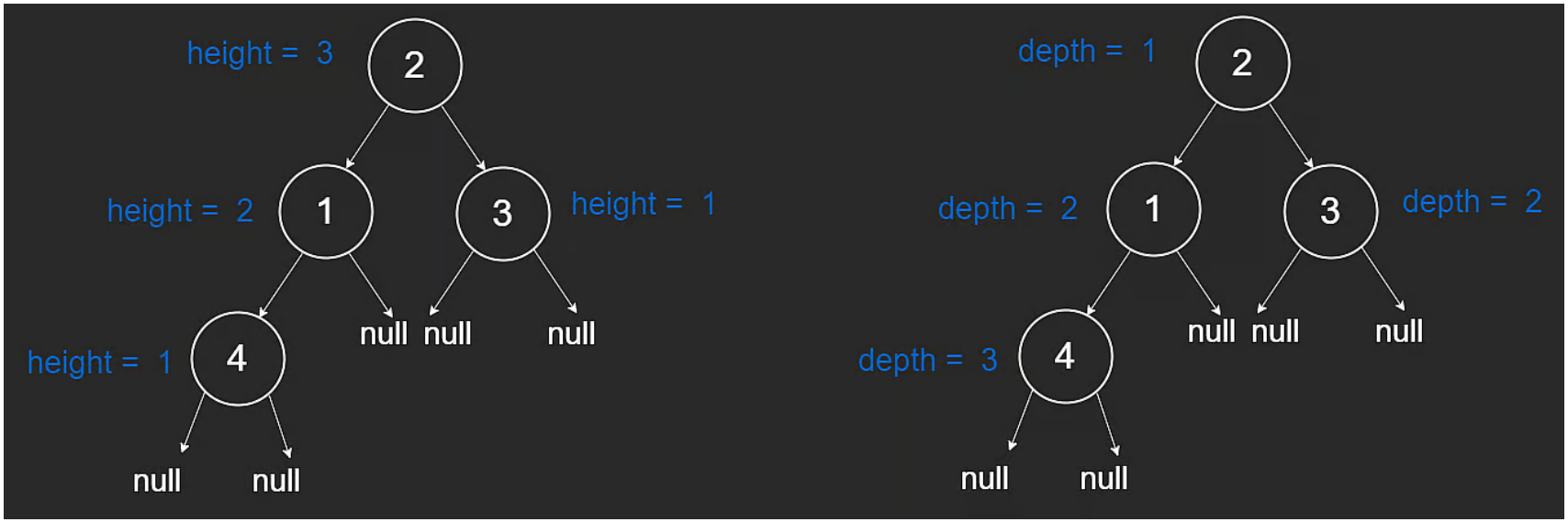
- A node without children is a **leaf node**.
- Nodes with one or two children are **non-leaf nodes**.
- Binary trees are undirected graphs with no cycles, ensuring the existence of leaf nodes.

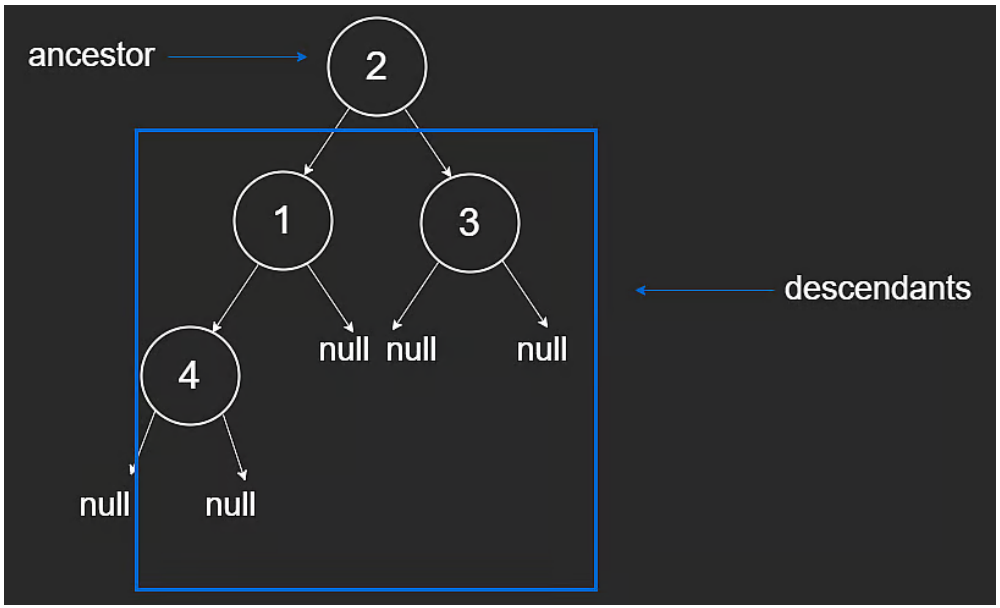
Properties-1





Properties-2





Binary Search Trees

Difference between Binary Trees and Binary Search Trees

- Every left child node must be *smaller* than its parent node.
- Every right child node must be *greater* than its parent node.
- BSTs do not allow duplicates.

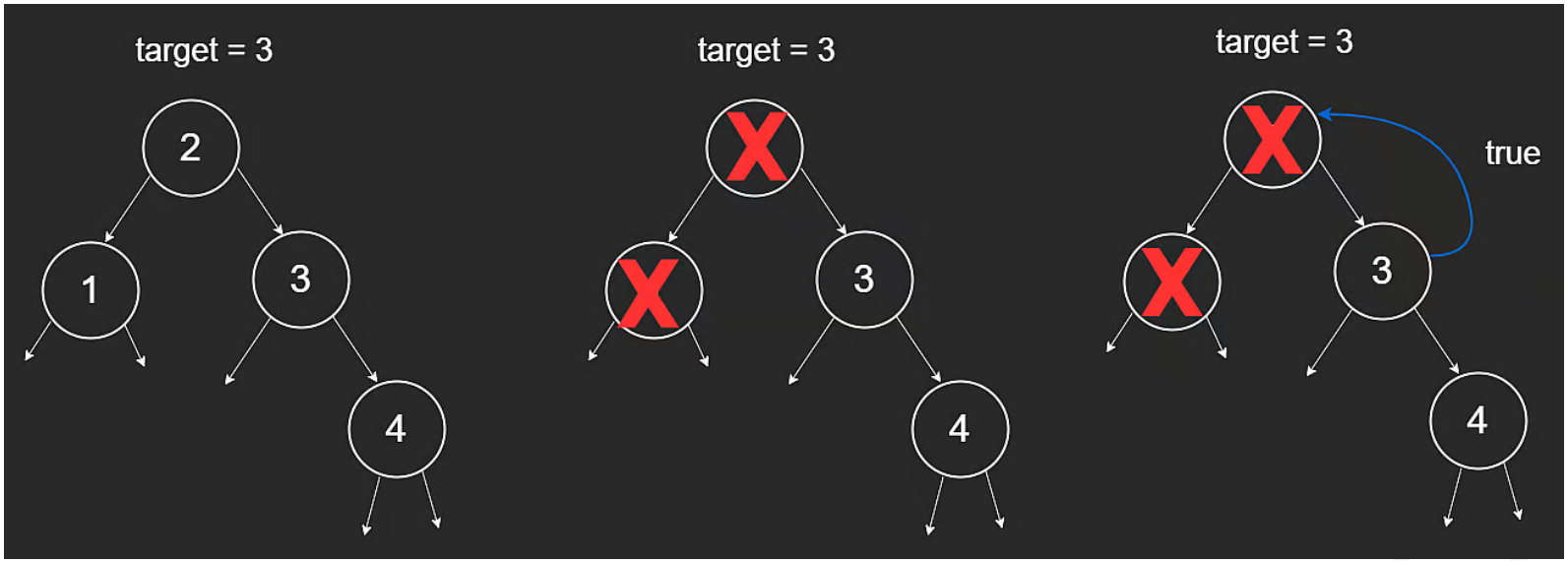
Motivation

- While binary search on sorted arrays and search operations in BST both operate in $O(\log n)$ time, BSTs excel in *insertion* and *deletion* operations.
- Inserting or deleting a value in a BST runs in $O(\log n)$ time, significantly more efficient than the $O(n)$ time required for the same operations in an array.

BST Search

Let's take the tree [2,1,3,null,null,null,4] for example and search for target = 3.

```
def search(root, target):
    if not root:
        return False

    if target > root.val:
        return search(root.right, target)
    elif target < root.val:
        return search(root.left, target)
    else:
        return True
```



Balanced vs. Skewed Binary Trees

Balanced Binary Tree:

- In a balanced binary tree, the height of the left and right subtrees is either equal or has a difference of 1.
- This balance allows for the elimination of half of the nodes at each step, akin to binary search in an array.
- **Time Complexity:** $O(\log n)$ - Efficient search time due to the structured reduction of the search space.

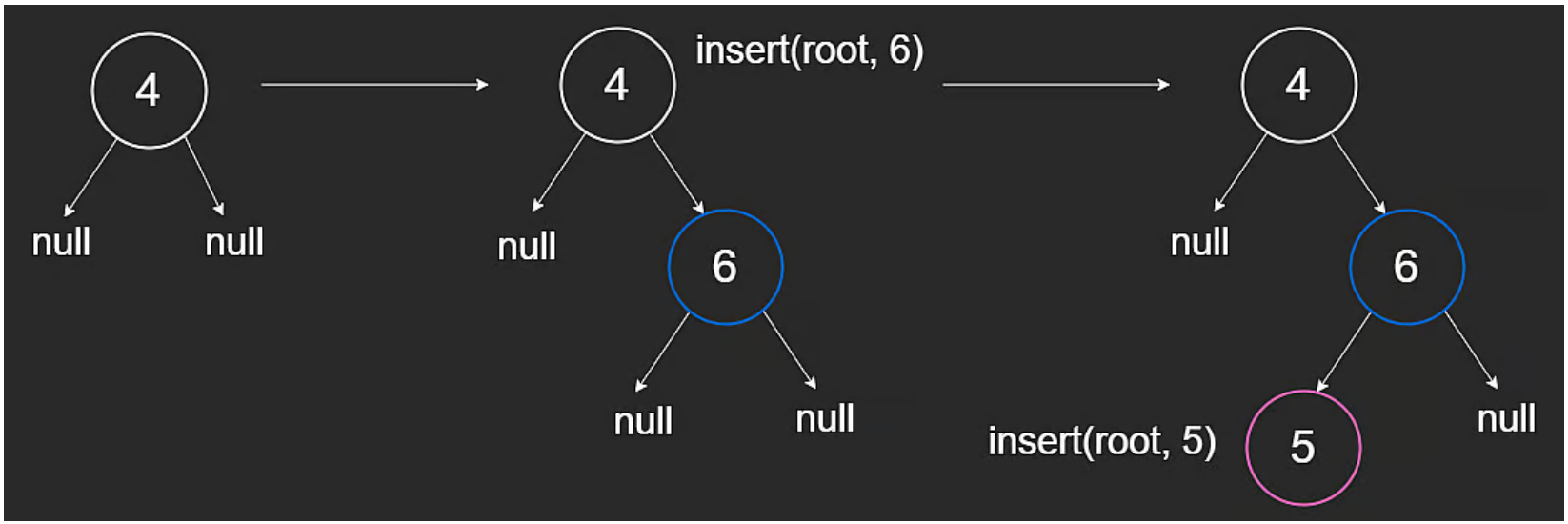
Skewed Binary Tree:

- A skewed binary tree occurs when all the nodes are on one side, either left or right, making it resemble a linked list.
- In such cases, each node needs to be examined, leading to linear search time.
- **Time Complexity:** $O(n)$ - Represents the worst-case scenario for search operations.

Insertion to a BST

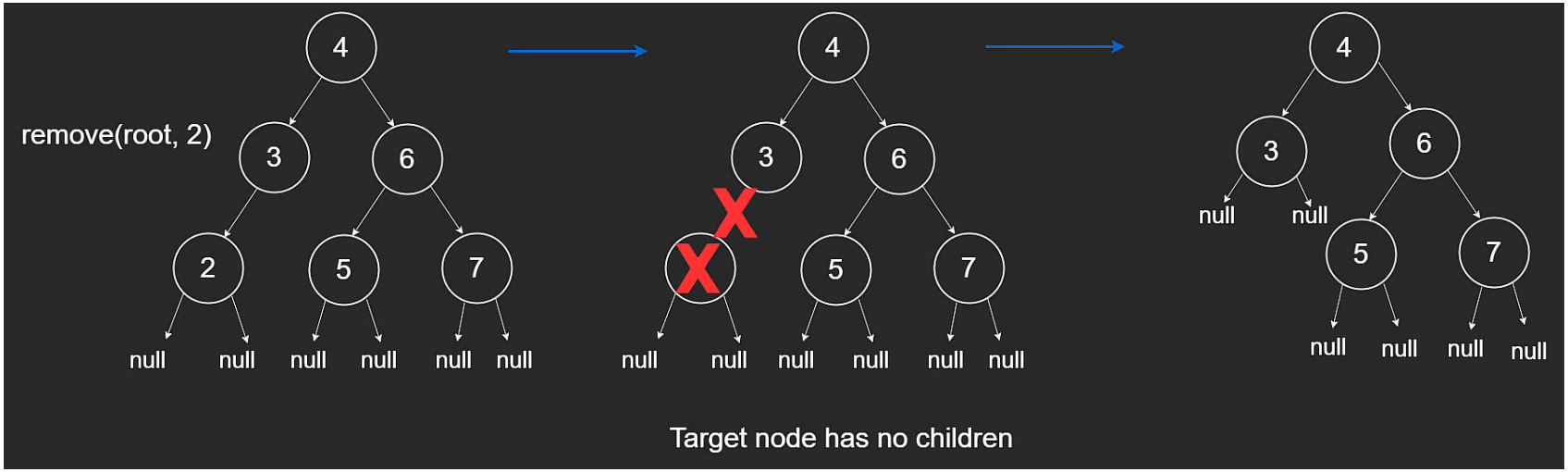
Let's take the tree [2,1,3,null,null,null,4] for example and search for target = 3.

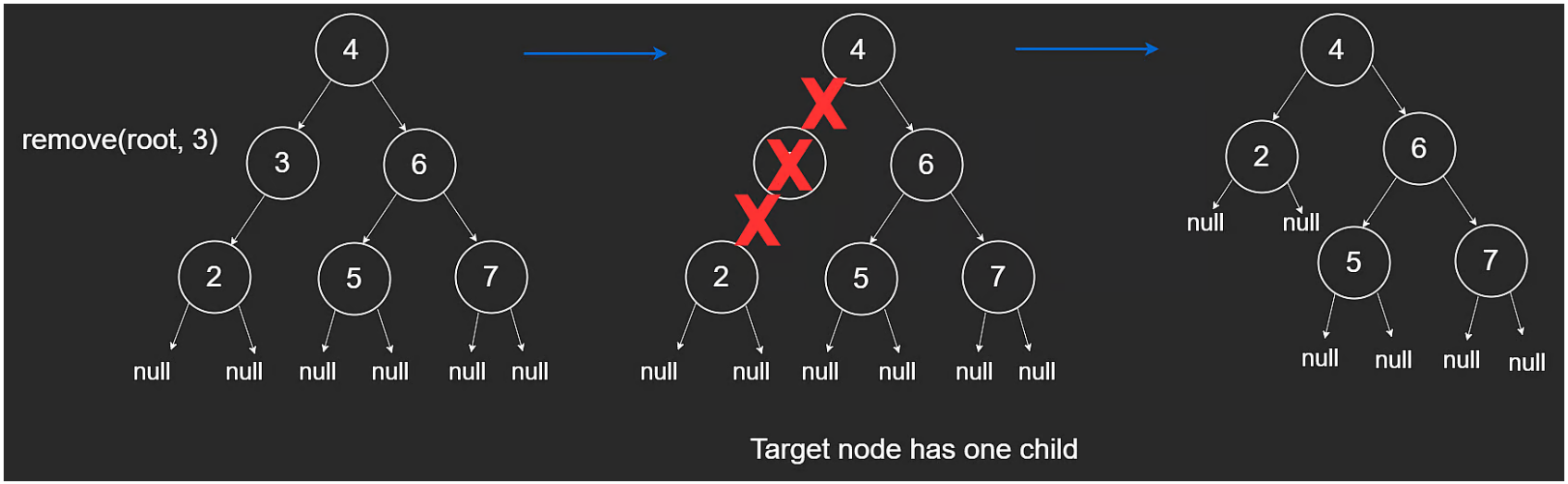
```
# Insert a new node and return the root of the BST.
def insert(root, val):
    if not root:
        return TreeNode(val)

    if val > root.val:
        root.right = insert(root.right, val)
    elif val < root.val:
        root.left = insert(root.left, val)
    return root
```



Removal from a BST-1

Case 1 - The target node has one child or no children

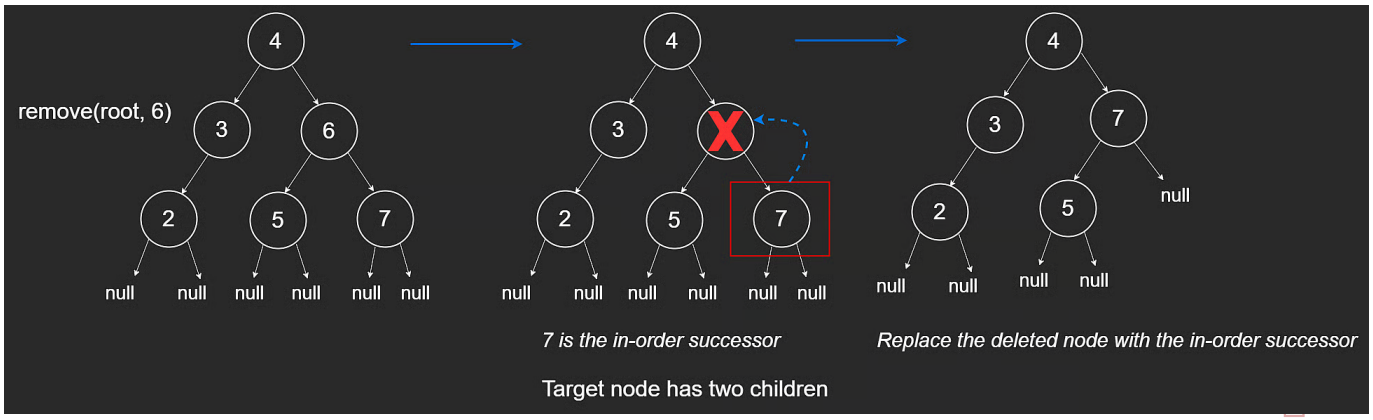




Removal from a BST-2

Case 2 - The target node has two children

```
# Return the minimum value node of the BST.
def minValueNode(root):
    curr = root
    while curr and curr.left:
        curr = curr.left
    return curr

# Remove a node and return the root of the BST.
def remove(root, val):
    if not root:
        return None

    if val > root.val:
        root.right = remove(root.right, val)
    elif val < root.val:
        root.left = remove(root.left, val)
    else:
        if not root.left:
            return root.right
        elif not root.right:
            return root.left
        else:
            minNode = minValueNode(root.right)
            root.val = minNode.val
            root.right = remove(root.right, minNode.val)
    return root
```



Balanced vs. Skewed Trees

- **Balanced Binary Tree:**
  - In a balanced binary tree, the height is $\log n$, ensuring efficient operations.
  - This efficiency stems from the ability to divide the tree into roughly equal halves at each level, similar to the concept used in merge sort.
  - **Time Complexity:** For search, insert, and delete operations - $O(\log n)$.
- **Skewed Binary Tree:**
  - In the worst case, a binary tree may be left-skewed or right-skewed, resembling a linked list.
  - The height of such a tree is $O(n)$, significantly impacting the efficiency of operations.
  - **Time Complexity:** For all operations described - $O(n)$.

Understanding the structure of the binary tree is crucial for assessing the efficiency of operations performed on it.

Summary:

- **Search Array:** Utilizes binary search to find a target value in $O(\log n)$ time within a sorted array.
- **Search Range:** Adapts binary search to guess a number within a given range based on feedback, efficiently narrowing down the possibilities.
- **Binary Tree:** A hierarchical data structure with nodes connected by edges, where each node has at most two children (left and right).
- **Binary Search Tree (BST):** A special type of binary tree that maintains a sorted order, where each node's left child is less than the parent, and the right child is greater.
- **BST Insert and Removal:**
  - *Insert:* Adds a new node while preserving the BST properties, typically in $O(\log n)$ time for balanced trees.
  - *Removal:* Deletes a node and restructures the tree to maintain the BST order, also aiming for $O(\log n)$ efficiency.