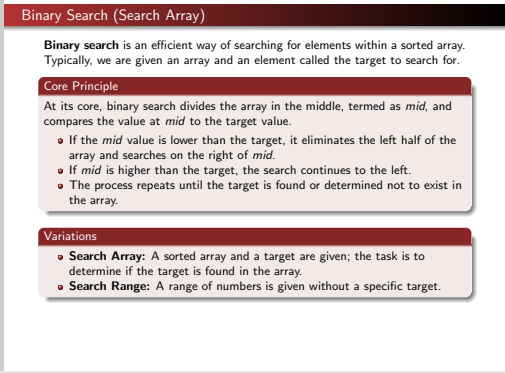
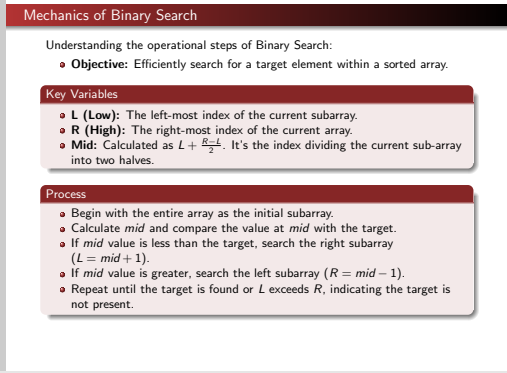


Binary search is an efficient way of searching for elements within a sorted array. Typically, we are given an array, and an element called the target to search for.
At its core, binary search divides the array in the middle, called *mid*, and compares the value at *mid* to the target value. If the *mid* 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 proceeds to the left. We either find the target or determine that the target doesn't exist in the array.
In interviews and algorithmic problems, there are two common variations of binary search problems:

- **Search Array** - A sorted array, and a target is given, and 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.

).



Now that we know the general idea behind binary search, we can determine how it would work logistically. The target value is given as the input, but we need to calculate *mid*. *mid* is initially calculated by adding the left-most index to the right-most index and then dividing the result by 2. This allows us to have two equal sections of the array (recall this is exactly what we did with merge sort, except in this case we are not allocating any new space). In the case of binary search, we will have the following:

- L - the left-most index of the current subarray.
- R - the right-most index of the current array.
- *mid* - $L + \frac{R-L}{2}$, the index at which the current sub-array divides itself into two equal halves.

L and R are sometimes referred to as low and high. It doesn't really matter which one you choose as long as they are understood by the interviewer.
The idea now is that we will keep searching for the target until we either find the target, or our L pointer crosses the R pointer, in which case the target doesn't exist.

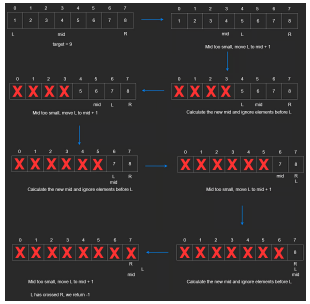
Binary Search Example: Target Exists in the Array

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
```

1. Initially, $L = 0$ (0th index) and $R = 7$ (7th index, `arr.length - 1`).
2. Calculate *mid* by $(7 + 0)/2 = 3$. The value at index 3 is 4.
3. Since 4 is less than 5, we search the right half by moving L to $mid + 1$.
4. Next, *mid* is recalculated and gives us 5. The value at index 5 is 6, which is greater than 5.
5. We adjust R to *mid* - 1, focusing our search on a smaller element. Now, L and R both point to index 4.
6. The new *mid* is 4, and we find our target at index 4. Thus, we return *mid*.

Binary Search Example: Target does not exist in the Array



1. Initially, $L = 0$ (0th index) and $R = 7$ (7th index, `arr.length` - 1).
2. Calculate *mid* by $(7 + 0)/2 = 3$. The value at index 3 is 4.
3. Since 4 is less than 5, we search the right half by moving L to $mid + 1$.
4. Next, *mid* is recalculated and gives us 5. The value at index 5 is 6, which is greater than 5.
5. We adjust R to *mid* - 1, focusing our search on a smaller element. Now, L and R both point to index 4.
6. The new *mid* is 4, and we find our target at index 4. Thus, we return *mid*.

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:

- Your guess is correct.
- Your guess is too small–you need to guess higher
- 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.

Imagine you're given a range from 1 to 100 and asked to guess a number someone is thinking of. If your guess is too small, you need to guess higher; if it's too big, you guess lower. Unlike traditional binary search in an array, you don't know the target value here. You rely on feedback after each guess to adjust your next guess accordingly. This strategy mirrors binary search's logic but is adapted to guessing a number within a range based on feedback, rather than searching for a specific value in an array.

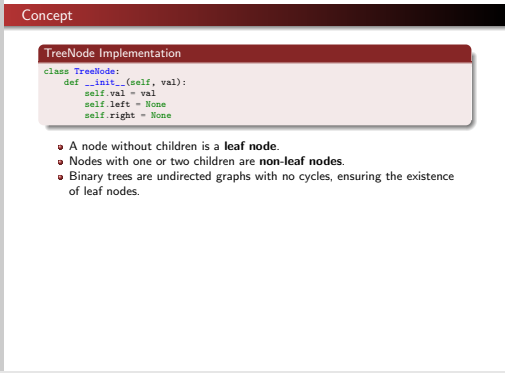
Pseudocode

```python
# 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 $O(\log n)$

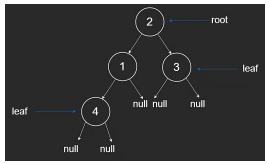
As observed, based on the return value we receive from the `isCorrect` function, we can choose to adjust our high and low accordingly, which, as stated in the previous chapter, is just another way of representing the L and R pointers.

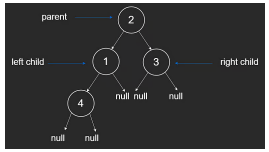
This technique can be confusing and may come across as very subtle because you have to figure out how to modify the typical implementation of binary search. For questions like these, a predefined method API is given, in this case, `isCorrect`, and you are required to treat the function as a black-box and use it within your own binary search method.



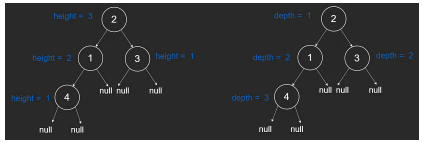
The Binary Tree is a fundamental data structure composed of nodes connected by pointers. Unlike linear connections in linked lists, binary trees have nodes with at most two pointers named the left child and the right child. The initial node is called the root node, with connections branching downwards. If a node does not have any children, it is classified as a leaf node. If a node has even a single child, either left or right, it would be classified as a non-leaf node.

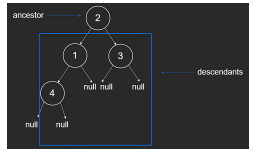
Unlike linked lists, binary tree node pointers can only point in one direction. As such, cycles are not allowed in binary trees. Mathematically speaking, a binary tree is an undirected graph with no cycles. This means that a leaf node is always guaranteed to exist. The same applies to the decision trees that are used in recursion.





Root node is the highest node in the tree and has no parent node. All of the nodes in the tree can be reached by the root node. Leaf nodes are nodes with no children. The nodes at the last level of the tree are guarenteed to be leaf nodes but they can also be found on other levels.





The height of a binary tree is measured from the root node all to way to the lowest leaf node, just like the height of anything in real life. The height of a single node tree is just 1, if the node itself is counted, or 0 if not.
Sometimes, the height is counted by the number of edges that are in between the nodes instead of the nodes themselves. Using this method, the height will be n-1 where n is the number of nodes, in the path from the root to the lowest leaf.
The maximum height of the given binary tree in the visual below is 3. Alternatively, if we were counting by edges, instead of nodes, it would be 2. The number of edges in a tree are n1, where n is the number of nodes. Depth of a binary tree node is measured from itself all the way up to the root. As observed in the visual below, the depth at the root node is 1, with it increasing as we go down. Measure depth at a given node by looking at how many nodes are above it, including the node itself.

A node connected to all of the nodes below it is considered an ancestor to those nodes.

The descendent of a node is either child of the node or child of some other descendent of the node.

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.

Binary Search Trees (BST) are a specialized form of binary trees with a unique sorted property:

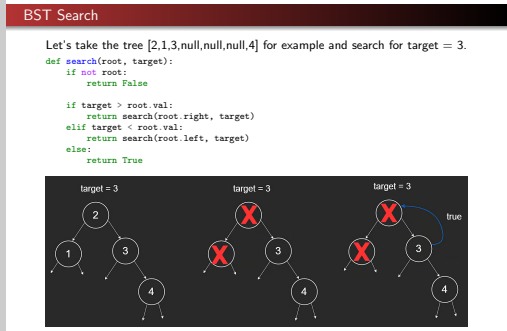
- Every left child node must be *smaller* than its parent node.
- Every right child node must be *greater* than its parent node.
- BSTs do not allow duplicates.

This sorting property extends to all subtrees within the BST, ensuring organized data that allows for efficient search, insertion, and deletion operations.

Why use Binary Search Trees over sorted arrays?

- 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.

BSTs offer a balanced approach, providing efficient search capabilities like sorted arrays, while also enabling quicker modifications to the data structure.



Trees are best traversed using recursion. You could traverse iteratively, however, that requires maintaining a stack, which is a lot more complicated. For recursion, as discussed before, we need a base case and the function calling itself.
Let's take the tree [2,1,3,null,null,null,4] for example and search for target $= 3$.
In binary search, if the current element was greater than the target, we went left and if the current element was smaller than the target, we went right. A similar approach can be taken here. We know all nodes to the left are smaller than our current node and all nodes to the right are greater than our current node. Knowing this, we can go right if our current node is smaller than the target and go left if the current node is greater than the target.
If the target exists in the tree, we will return true. Otherwise, we return false.
In the case of the example, we first start by comparing the root value against the target. 2 is too small, so our target must be on the right, meaning we can eliminate the left-subtree. When we go right, the first node is 3, which equals target, so we return true from the recursive call, meaning our target does exist in the tree.

Balanced vs. Skewed Binary Trees

Balanced Binary Tree:

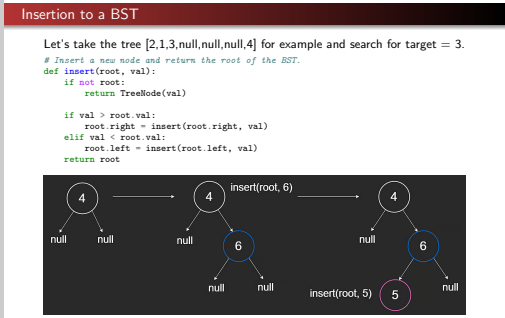
- In a balanced binary tree, the height of the left and right subtrees is either equal or has a difference of 1.
- This balance allows for the elimination of half of the nodes at each step, akin to binary search in an array.
- **Time Complexity:** $O(\log n)$ - Efficient search time due to the structured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

Understanding the structure of the tree is crucial for optimizing search operations and achieving efficient time complexity.

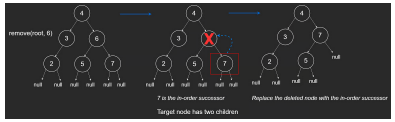
If we wish to delete node 2, which has no children, the left_child pointer of 3 now points to null. If we wish to delete node 3, which has one child, the left_child pointer of the root node will point to 2 instead of 3.



If we wish to insert a new node into the BST, we first have to traverse the BST to find the right position, and then insert this node. If we have a BST [4], and wish to insert 6, we could either end up with [4,null,6] or [6,4,null]. Both of these would be valid BSTs. In the first example, we added the 6 as a leaf node, which is an easier process than the second example. Let's add 5 to previously resulting tree [4,null,6], which results in [4,null,6,5,null].

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
```



If we wanted to delete a node with two children, say, 6, we replace the node with its in-order successor.

The in-order successor is the left-most node in the right subtree of the target node. Another way of looking at it is that it is the smallest node among all the nodes that are greater than the target node. This will ensure that the resulting tree is still a valid binary search tree.