Binary Search and binary search trees

Agenda

- Binary Search
- Binary Search Trees

Binary Search

Fast algorithm to find a target value in a **sorted list**.

Instead of checking every element (like linear search), binary search repeatedly **cuts the search space in half**.

The list/array must be **sorted**

Binary Search Algorithm

Algorithm Steps

Set two pointers:
left = 0, right = n − 1

While left ≤ right:

- Compute middle index
  mid = floor(left + right) // 2
- Compare arr[mid] with target:
  - If equal → ✅ **Found**
  - If target < arr[mid] → search **left half**
    right = mid − 1
  - Else → search **right half**
    left = mid + 1

If loop ends → ❌ target not in array.

Binary Search

Target: 9

1	**3**	**4**	**5**	**7**	**9**	**10**	**12**
0 left	1	2	3 **mid**	4	5	6	7 right

Step:

Calculate midpoint of the array search bounds -> floor (0+7)/2 = **3**

Check value at mid point in array.

5<9

Check right half

Binary Search

Target: 9

Discard

1	**3**	**4**	**5**	**7**	**9**	**10**	**12**
0	1	2	3	4 left	5 **mid**	6	7 right

Step:

Calculate midpoint of the array search bounds -> floor (4+7)/2 = **5**

Check value at mid point in array.

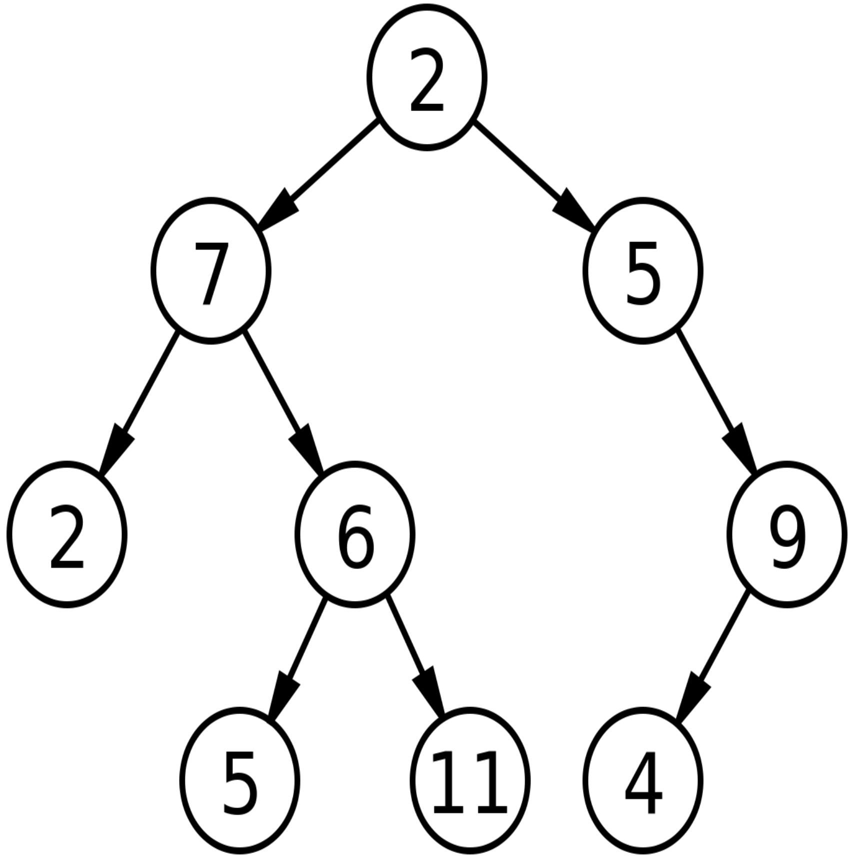
Target found!

Binary Search

- **Time Complexity:**
- Best case: **O(1)**
- Worst case: **O(log n)**
- Much faster than O(n) linear search for large data

Binary Trees

Binary tree A tree in which each node can have two child nodes, a left child node and a right child node. Order of nodes is not a requirement.



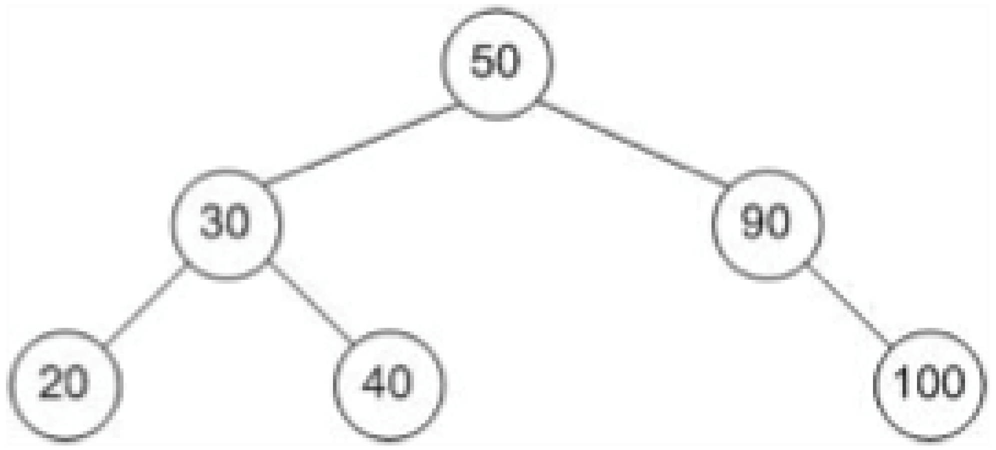
A Binary Tree

Binary Search Trees

Binary: each node has two children

Order: the key in any node is larger than all keys in the left subtree, and smaller than all keys in the right subtree

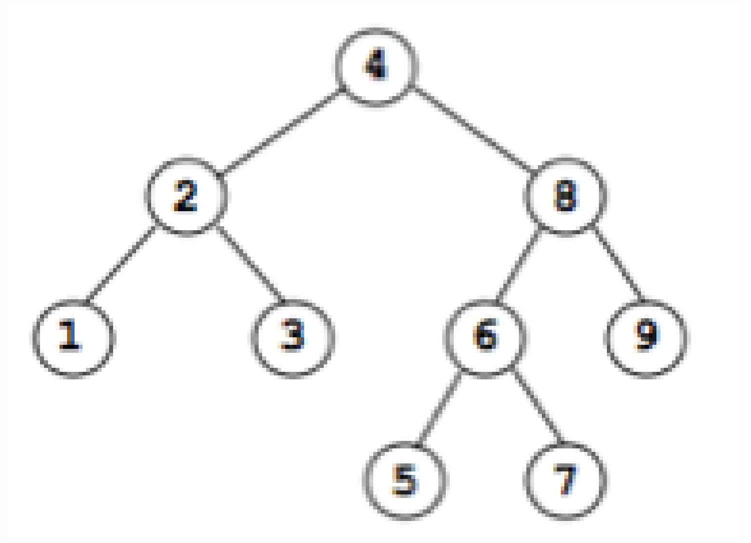
Search: the ordering facilitates search



Classic Tree Traversals

Use tree traversals to visit the nodes in a tree

- *Preorder*: root, left, right
- *Inorder*: left, root, right
- *Postorder*: left, right, root



preOrder and postorder traversals

```
ALGORITHM **preOrdert**($T$)
// Input: binary tree $T$
// Output: a Tree with the nodes visited in pre order (DFS)
if $T$ != ∅ then
    visit(T)
    preOrder($T_{\text{left}}$)
    preOrder($T_{right}$)


ALGORITHM **postOrdert**($T$)
// Input: binary tree $T$
// Output: a Tree with the nodes visited in post order (DFS)
if $T$ != ∅ then
    postOrder($T_{\text{left}}$)
    postOrder($T_{right}$)
    visit(T)
```

InOrder traversal

```
ALGORITHM **inOrdert**($T$)
// Input: binary tree $T$
// Output: a Tree with the nodes visited in order
**if** $T$ != ∅ **then**
    inOrder($T_{\text{left,}}$)
    visit(T)
    inOrder($T_{right}$)
```