Introduction to Algorithms

Module Code: CELEN086

Lecture 8

(28/11/24)

Lecturer & Convenor: Manish Dhyani
Email: manish.dhyani@nottingham.edu.cn

Search in a binary tree

Search in an unsorted list: $O(n)$ (linear search)

10	3	17	11	14	6	2	18

Search in a binary tree: $O(n)$ (Lecture 7)

Comparisons are <u>linearly</u> related to the size of binary tree.

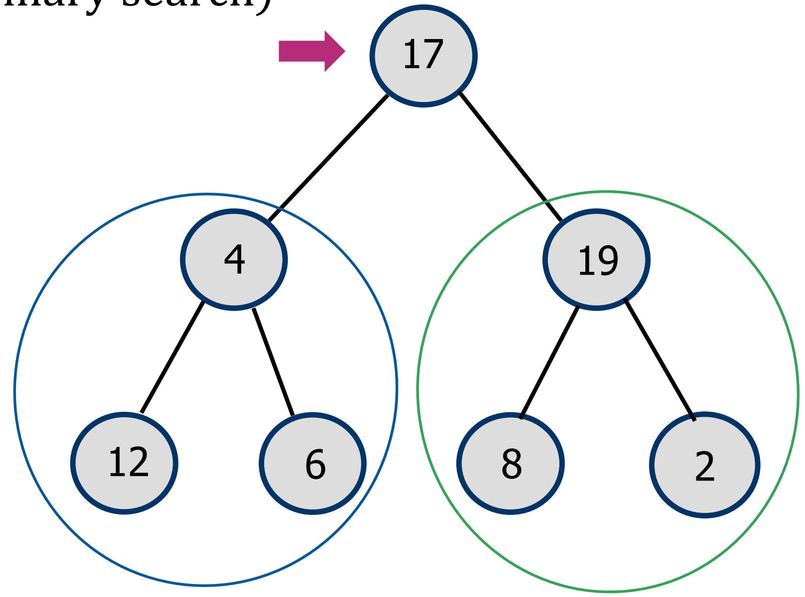
Search in an sorted list: $O(\log_2 n)$ (binary search)

2	3	6	10	11	14	17	18

Can we make better use of the binary structure in a binary tree?

We need a sorted tree:

Binary Search Tree

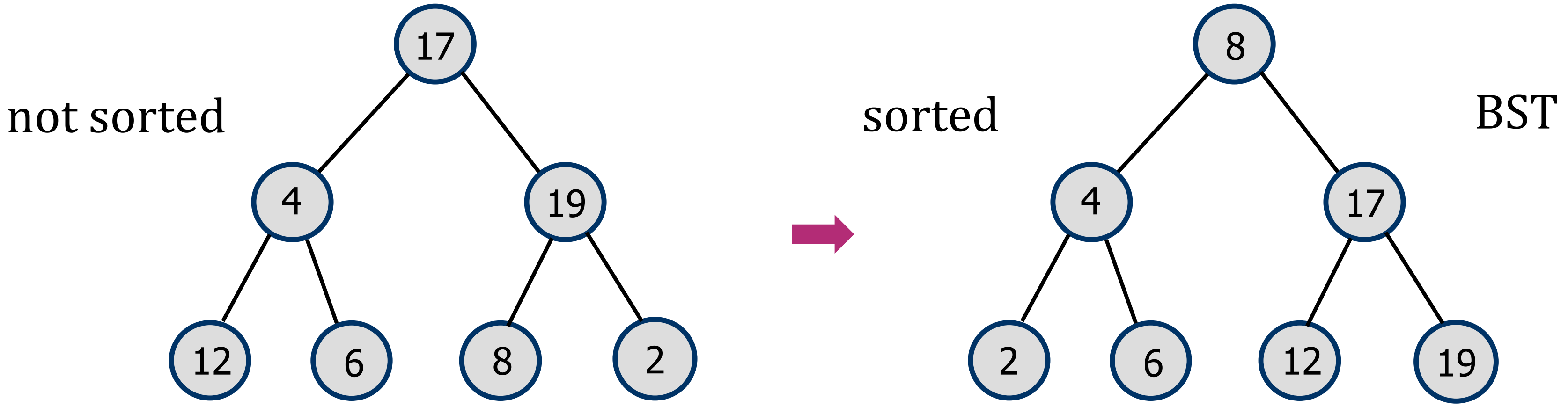


Binary Search Tree

A binary search tree (BST) is a sorted binary tree, such that

for the value stored in each node:

- It is greater than all the values stored its left sub-tree
- It is smaller than all the values stored its right sub-tree



Building a BST from a list

Create a binary search tree (with minimal height/depth) to store elements in the list (Seminar 7)

L = [9, 16, 1, 27, 20, 33, 15, 8, 13, 5]

Step 1: Sort the list

[1, 5, 8, 9, 13, 15, 16, 20, 27, 33]

Step 2: Find the middle element, and store it in root node.

Step 3: For left/right sub-lists, find the middle elements and store them in the left/right child nodes on next level.

Repeat this process until all elements are stored in the tree.

Example

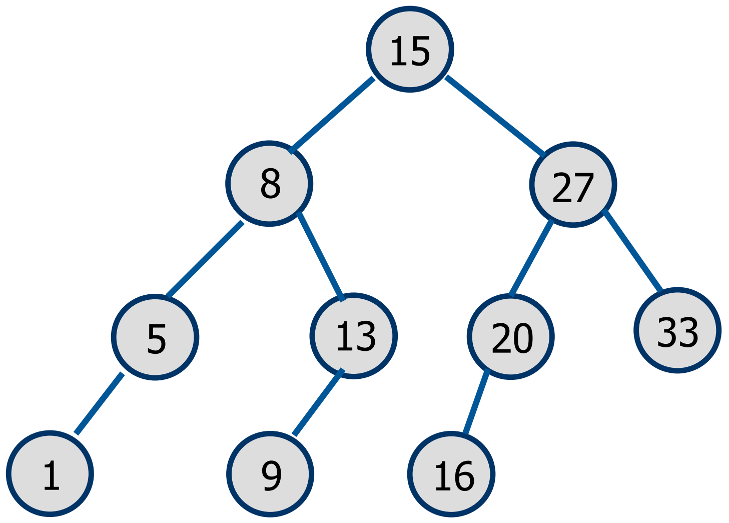
index of middle element: N/2+1

$[1, 5, 8, 9, 13, \boxed{15}, 16, 20, 27, 33]$

$[1, 5, \boxed{8}, 9, 13, \cancel{15}, 16, 20, \boxed{27}, 33]$

$[1, \boxed{5}, \cancel{8}, 9, \boxed{13}, \cancel{15}, 16, \boxed{20}, \cancel{27}, \boxed{33}]$

$[\boxed{1}, \cancel{5}, \cancel{8}, \boxed{9}, \cancel{13}, \cancel{15}, \boxed{16}, \cancel{20}, \cancel{27}, \cancel{33}]$



Binary Search Tree

BST with minimal height/depth

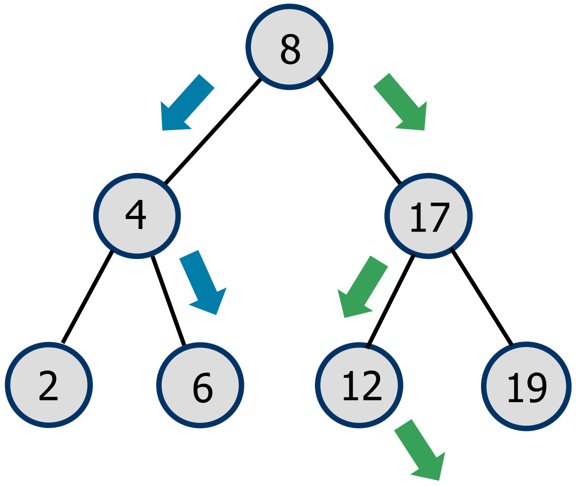
Search in a BST

Search 6 in the BST:

Comparisons	Actions
$6 < 8$	go left
$6 > 4$	go right
$6 == 4$	end

Search 16 in the BST:

$16 > 8$	go right
$16 < 17$	go left
$16 > 12$	go right
$leaf$	end



Time complexity

$O(\log_2 n)$ $or\ O(h)$

n: tree size
h: tree height

Algorithm: search in BST

Design a recursive algorithm that searches for a node value in a binary search tree.

```
Algorithm: search(x, BST)
Requires: a binary search tree BST and an element x
Returns: True if x occurs in BST; False otherwise

1. if  isLeaf(BST)
2.      return  False  // base case
3. elseif  x==root(BST)
4.      return  True // base case
5. elseif  x<root(BST)
6.      return   search(x, left(BST)) // recursive call
7. else  // x>root(BST)
8.      return   search(x, right(BST)) // recursive call
9. endif
```

Example of BST algorithms

Many algorithms on a binary search tree will take operations that are <u>linearly to the height</u> (or <u>logarithmically to the size</u>).

$O(h)$ $O(\log_2 n)$

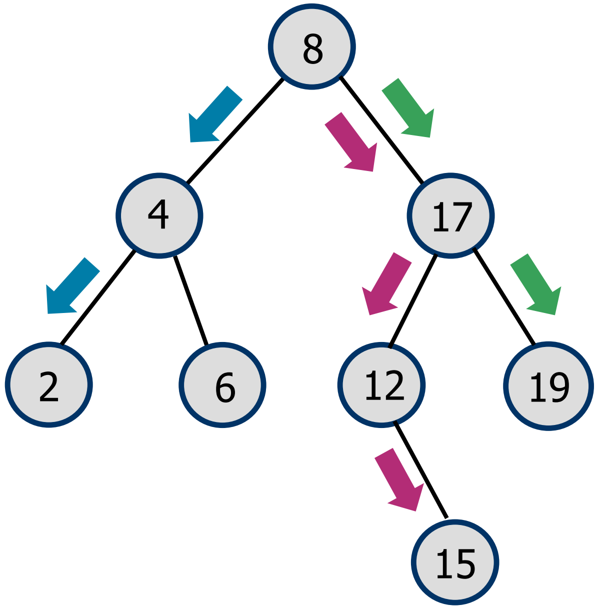
- Find minimum/maximum value.

  Min: 2 Max: 19

- Insert an element to BST.

  e.g., insert 15

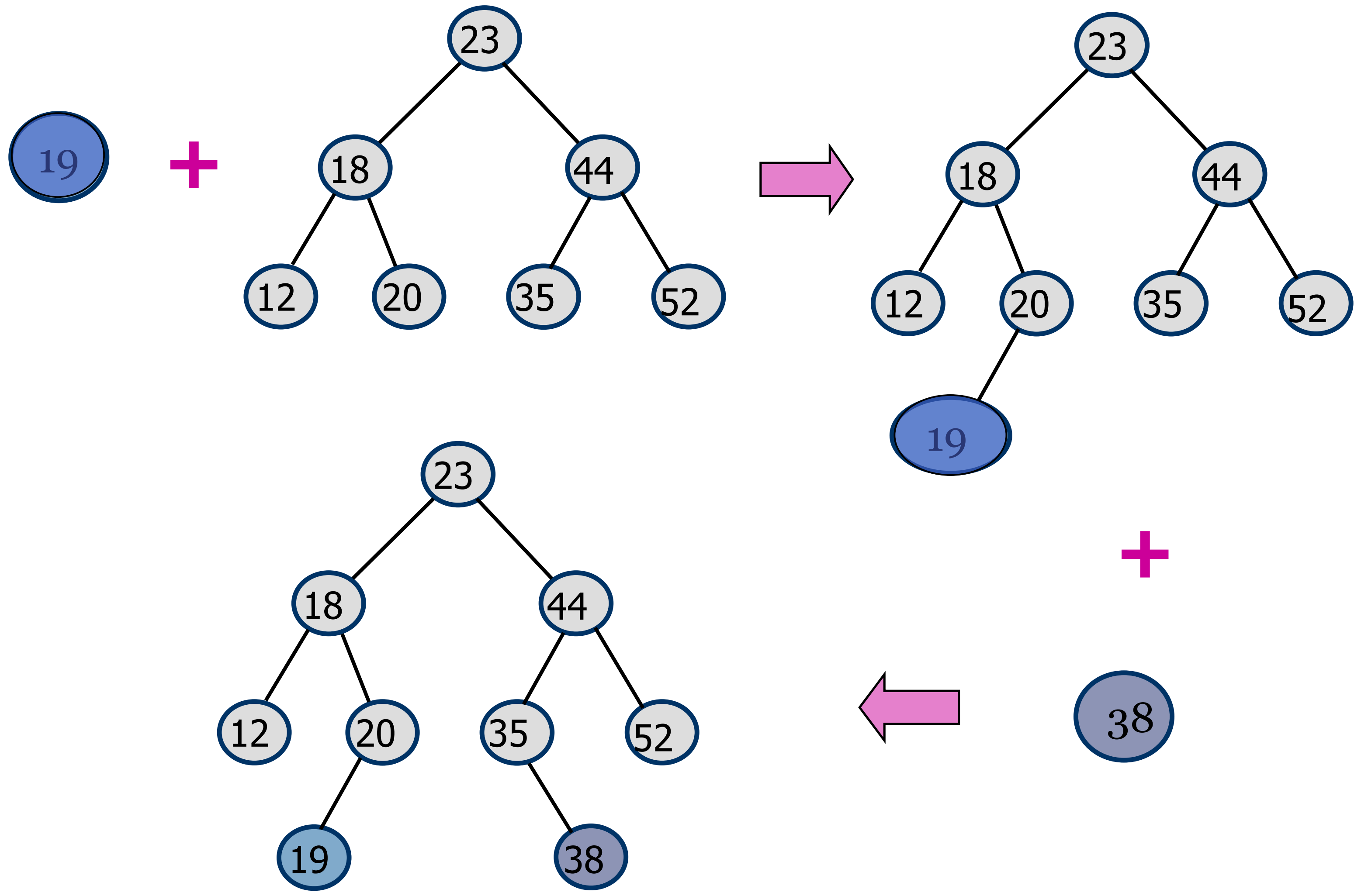
Can you design a recursive algorithm for insert(x, BST)?



Insert an element into BST

- Where is the right place to insert?
- Insert algorithm is a lot like search algorithm,
    - but it keeps going until it gets to a leaf,
    - and then adds a new left or right child, as appropriate.

BST insertion



Algorithm : insertBST$(x\ ,t\ )$
Requires : a BST t and an integer x
Returns : a BST with node value x in t

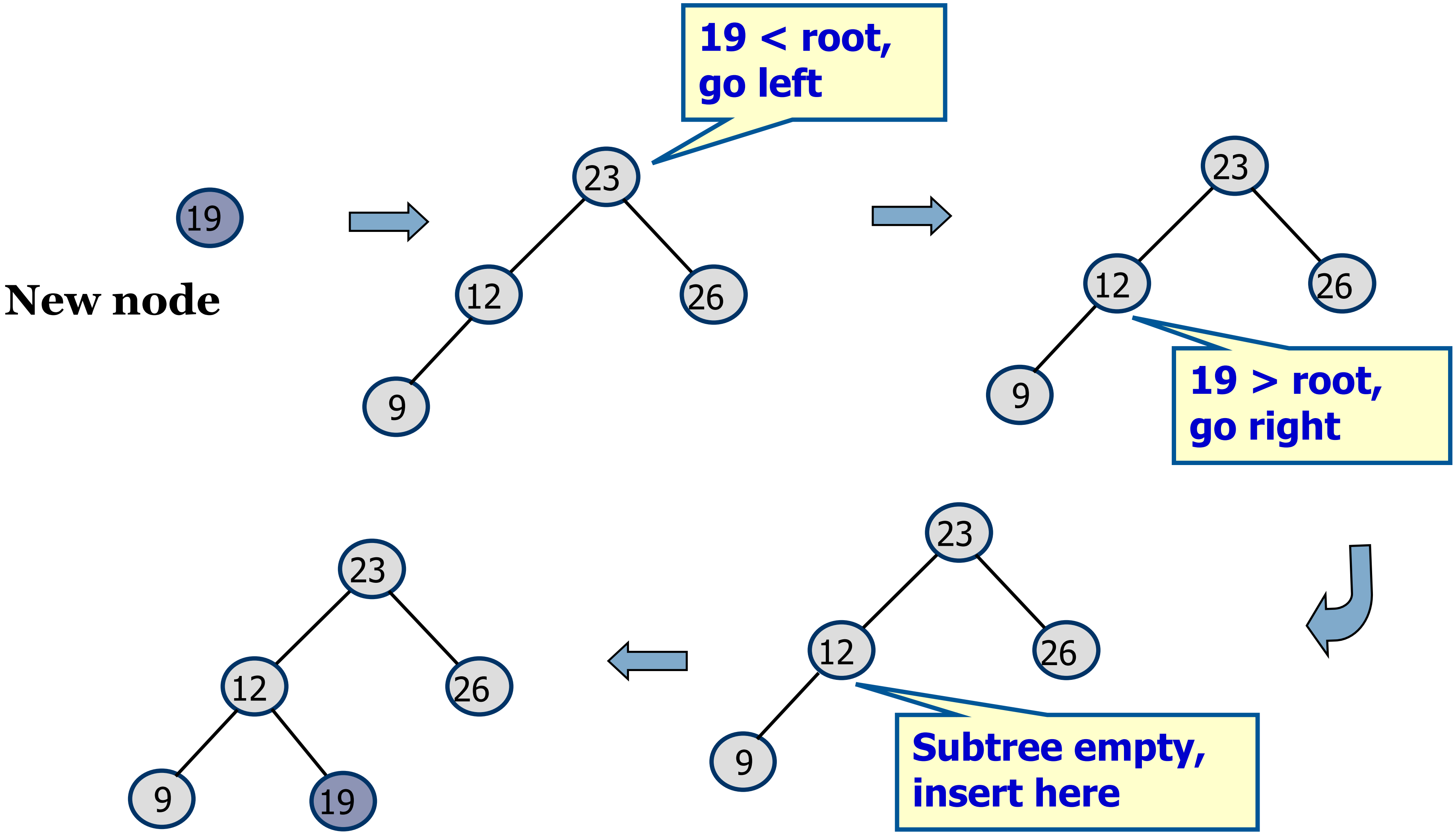
```
1:  if isLeaf($t$) then
2:      Node(leaf, $x$, leaf)
3:      return True
4:  else if $root(t) == x$ then
5:      return False
6:  else if $root(t) < x$ then
7:      return insertBST($x, right(t)$)
8:  else
9:      return insertBST($x, left(t)$)
10: endif
```

this algorithm has a fault – it does not attach the value to the tree. Revise version on the next slide

Revised insertBST

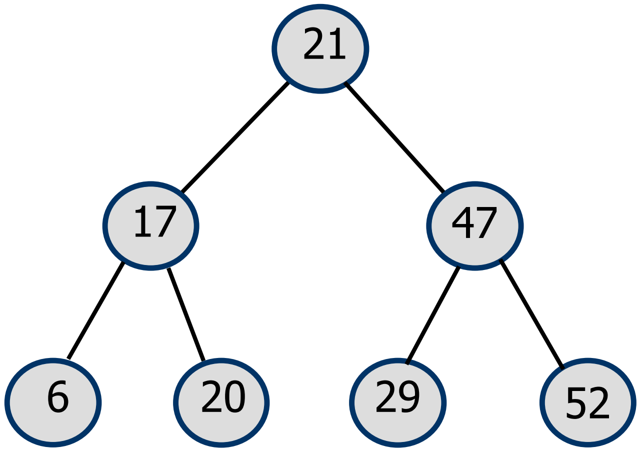
Algorithm: insertBST(x, t)
Requires : a number x and a Binary Search Tree t
Returns : the tree with x inserted, if the number is not already in the tree; otherwise returns the original tree

```
1. if isLeaf( $t$ )   then
2.        return node( leaf, $x$ , leaf )
3. else if root( $t$ )  ==  $x$ then
4.        return $t$
5. else if $x < root(t)$ then
6.     return node( insertBST($x$, $left(t)$) ,root( $t$ ), $right(t)$ )
7. else
8.     return node($left(t)$,root( $t$ ), insertBST( $x$, $right(t)$))
9. end if
```


19
New node
23
12
26
9
19 < root, go left
19 > root, go right
Subtree empty, insert here
19

Practice

- Insert a number 48 into the BST
- show how the algorithm is executed recursively



Traversal schemes

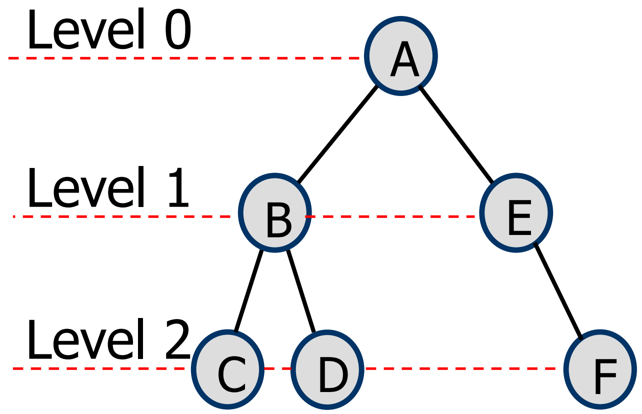
By traversing a tree we visit all the nodes in a particular order and create a list of node values stored in the tree.

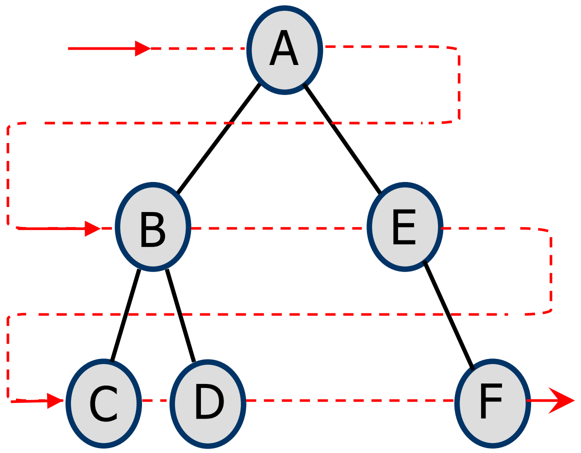
Traversal schemes can be classified as:

- Breadth first traversal scheme
- Depth first traversal schemes

Breadth first

- Each level is completed visited before the next level is started.
- In each level we list the node values from left to right.





[A B E C D F]

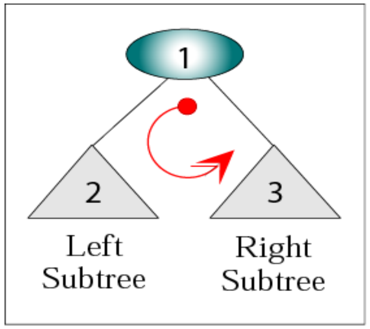
Depth first

There are three depth first traversal schemes on a binary tree, depending on the orders in which we recursively visit

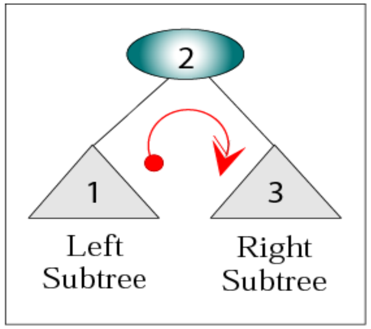
Node

Left subtree

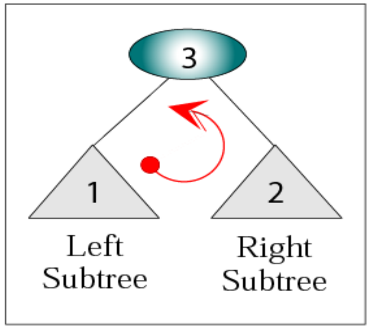
Right subtree



a. Preorder Traversal



b. Inorder Traversal



c. Postorder Traversal

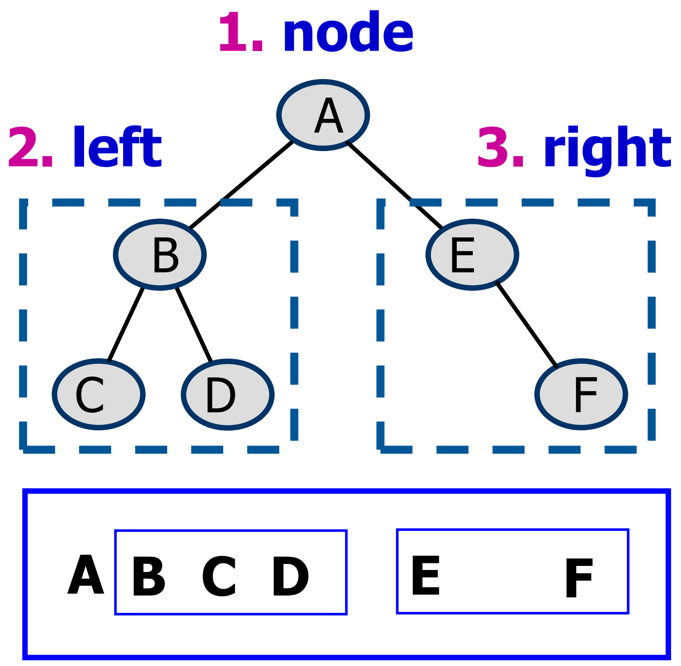
(NLR scheme)

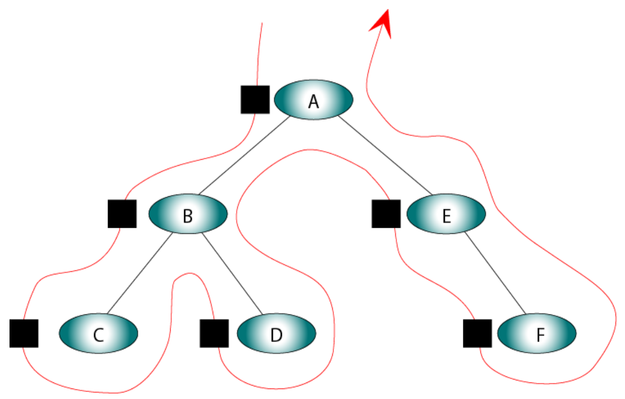
(LNR scheme)

(LRN scheme)

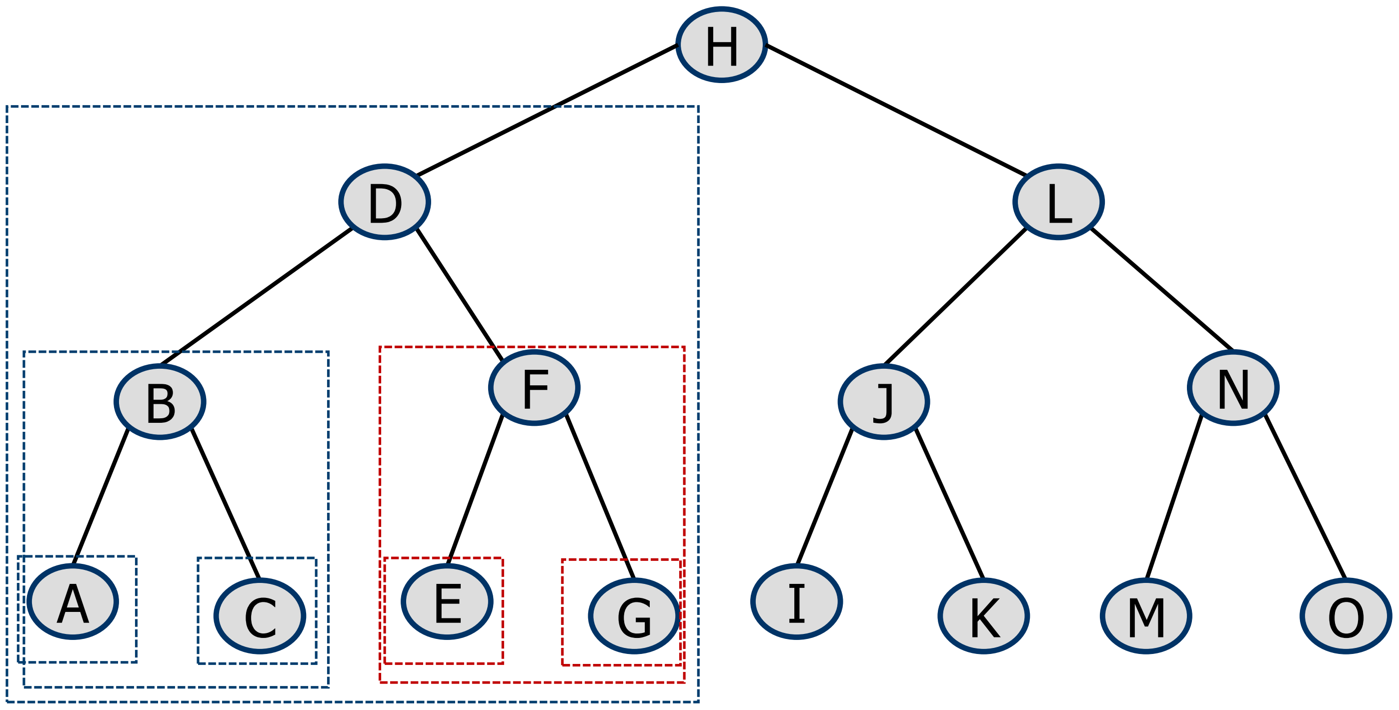
Preorder traversal (NLR)

Visit root node → Traverse left subtree → Traverse right subtree



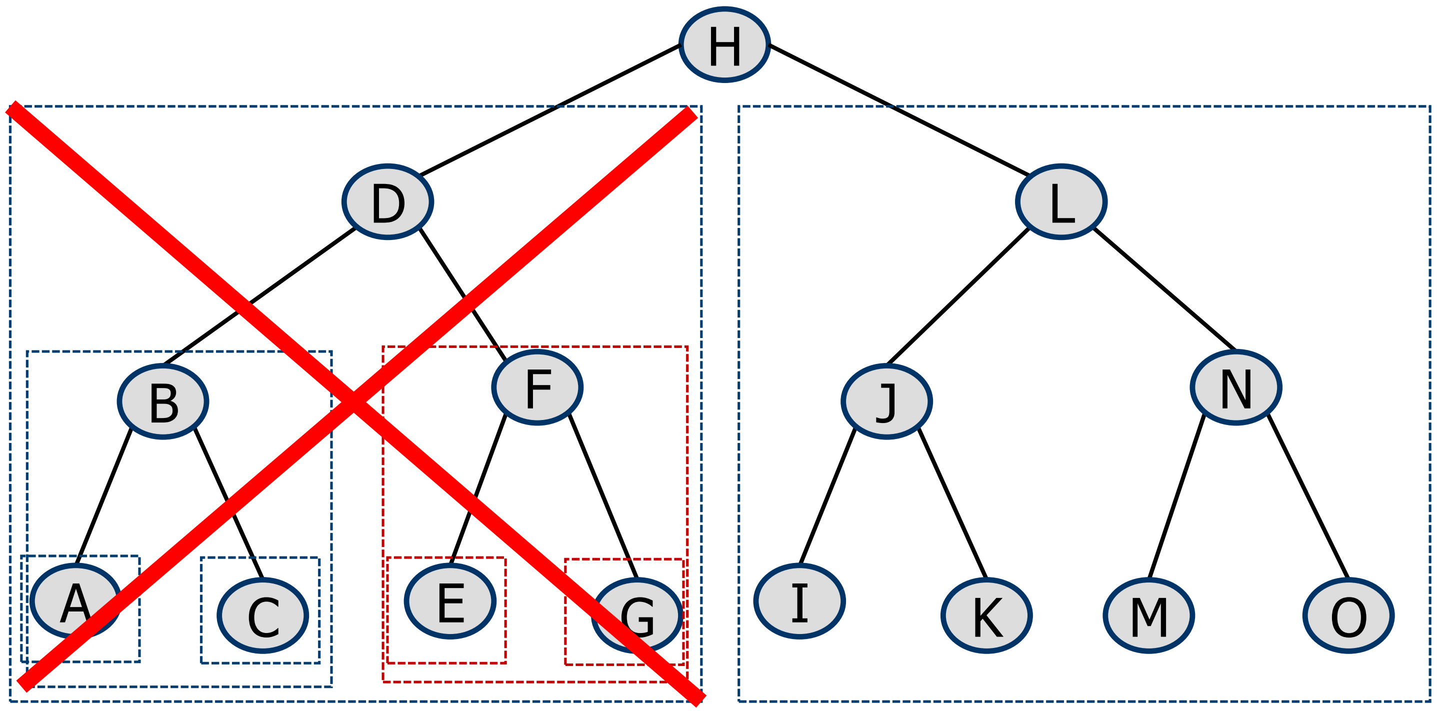


Preorder (NLR) Example



HDBACFEG

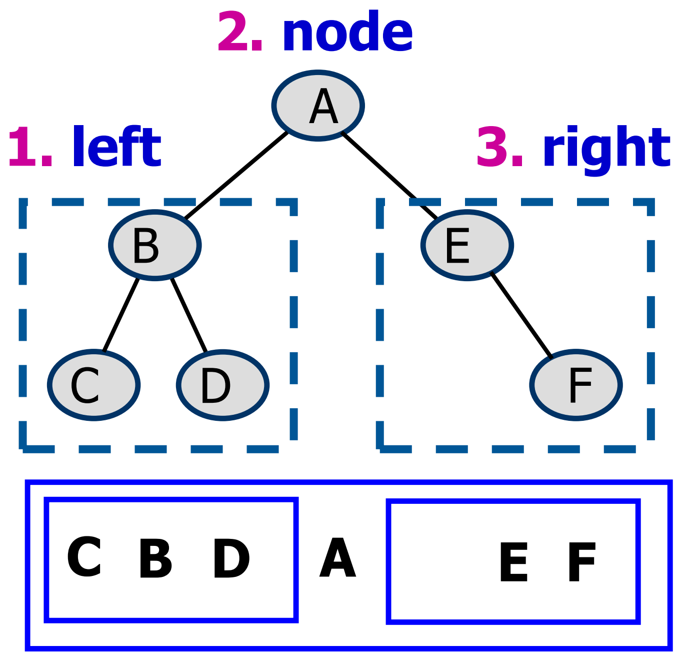
Preorder (NLR) Example

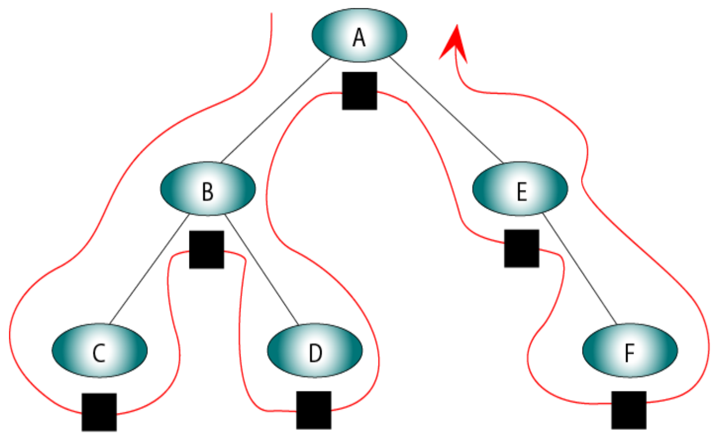


HDBACFEGLJIKNMO

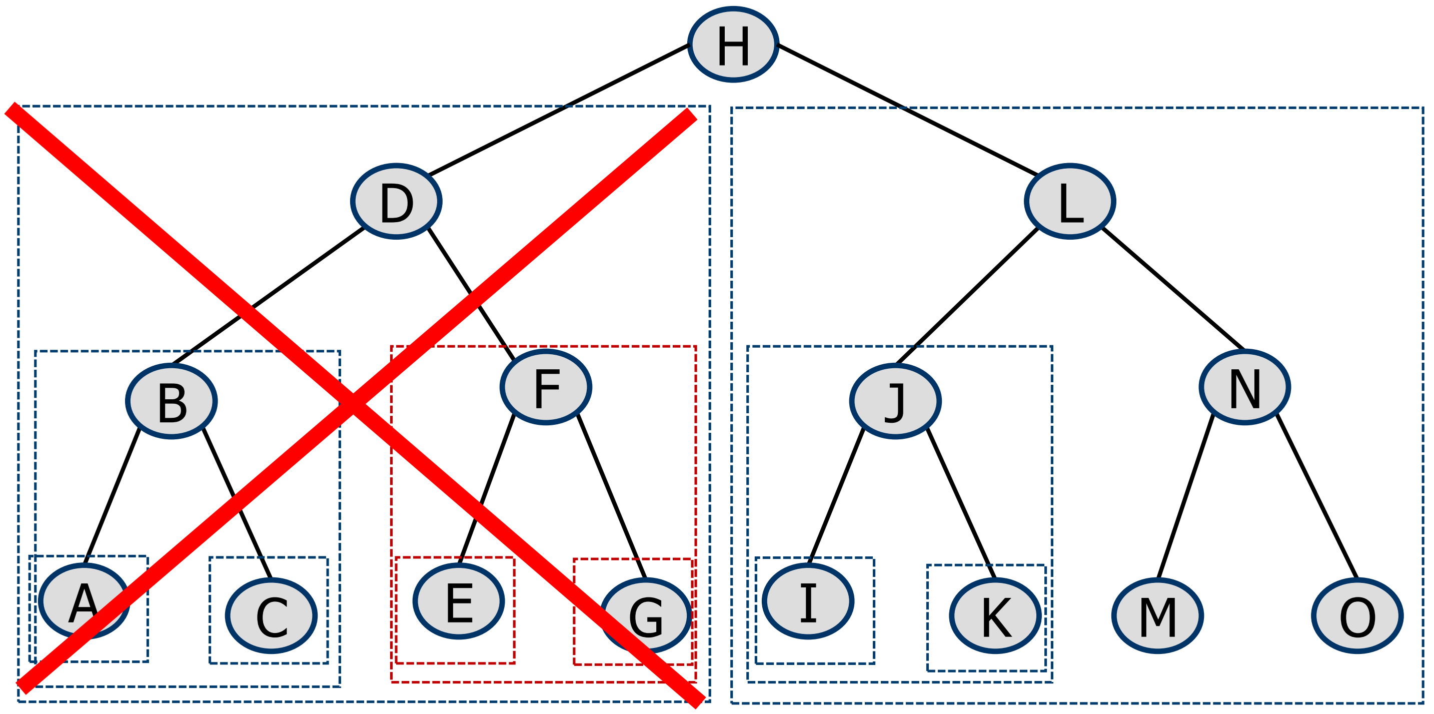
Inorder traversal (LNR)

Traverse left subtree → Visit root node → Traverse right subtree





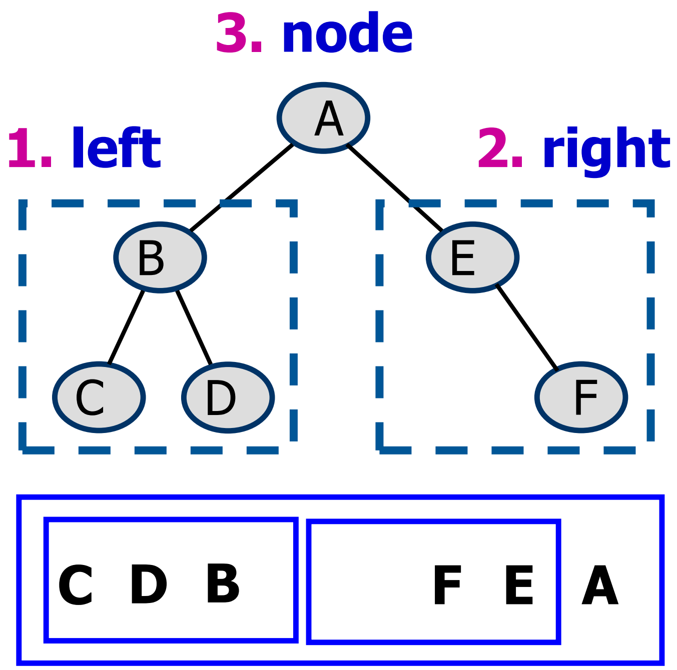
Inorder (LNR) Example

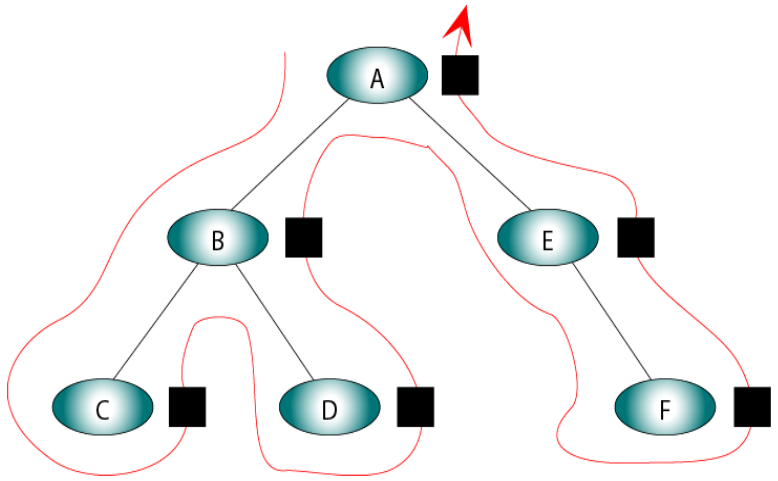


ABCDEFGHIJKLMNO

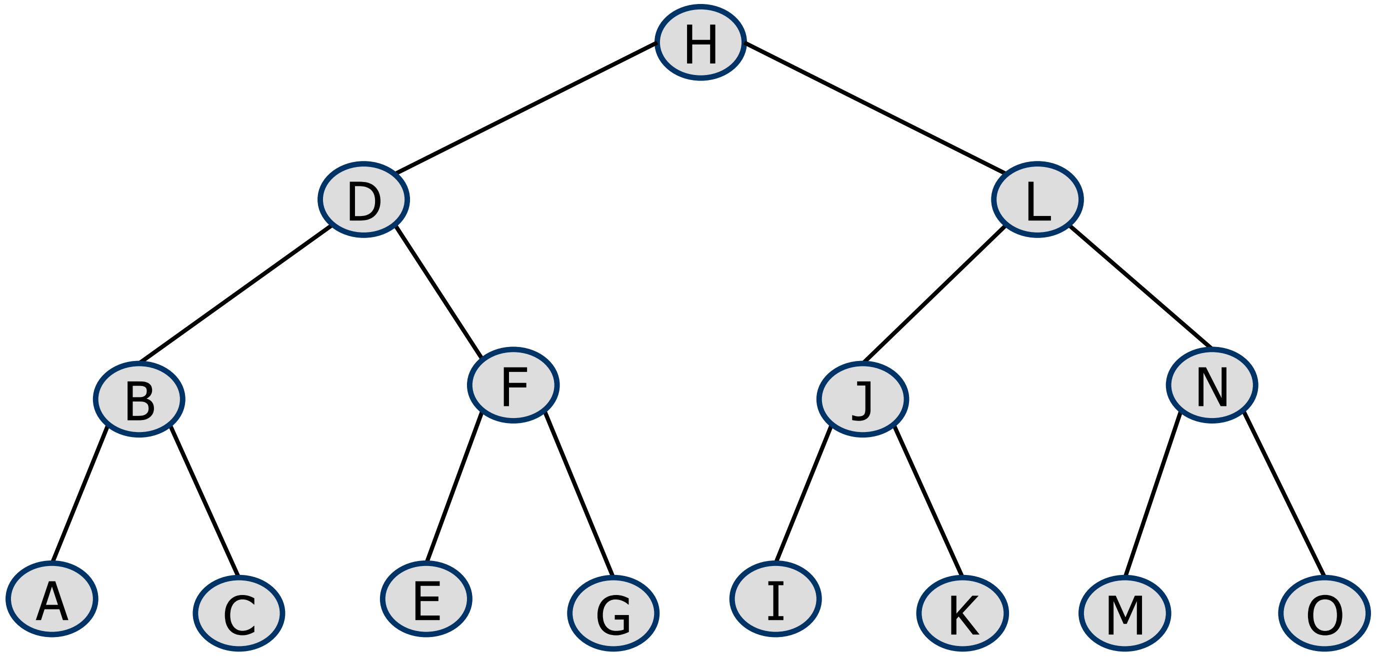
Postorder traversal (LRN)

Traverse left subtree → Traverse right subtree → Visit root node





Postorder (LRN) Example



Answer: [A C B E G F D I K J M O N L H]

Practice

Find the lists obtained by traversing the following binary search tree using:

NLR scheme (preorder):

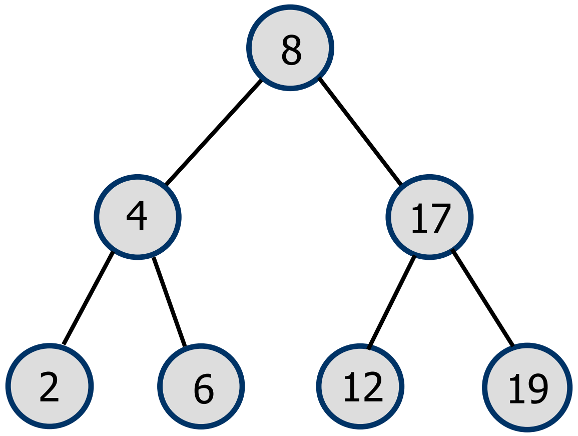
[8, 4, 2, 6, 17, 12, 19]

LNR scheme (inorder):

[2, 4, 6, 8, 12, 17, 19]

LRN scheme (postorder):

[2, 6, 4, 12, 19, 17, 8]



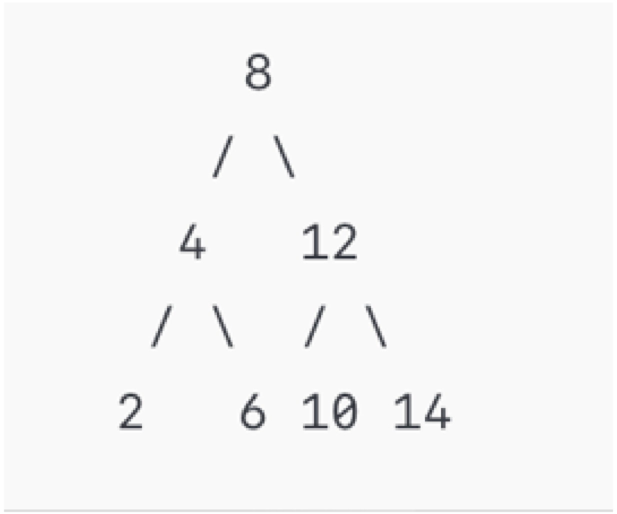
Conclusion:
LNR scheme (inorder traversal) returns a sorted list from a BST.

Balanced Binary Search Tree

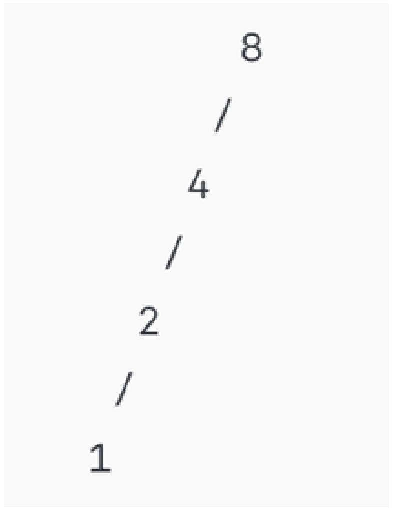
A binary tree satisfy condition of the Binary search tree and at each node, verify that :

Absolutedifference(Height(left(T))−Height(right(T)))≤ 1.

A balanced BST is a binary search tree where the height difference between the left and right subtrees of any node is at most 1.

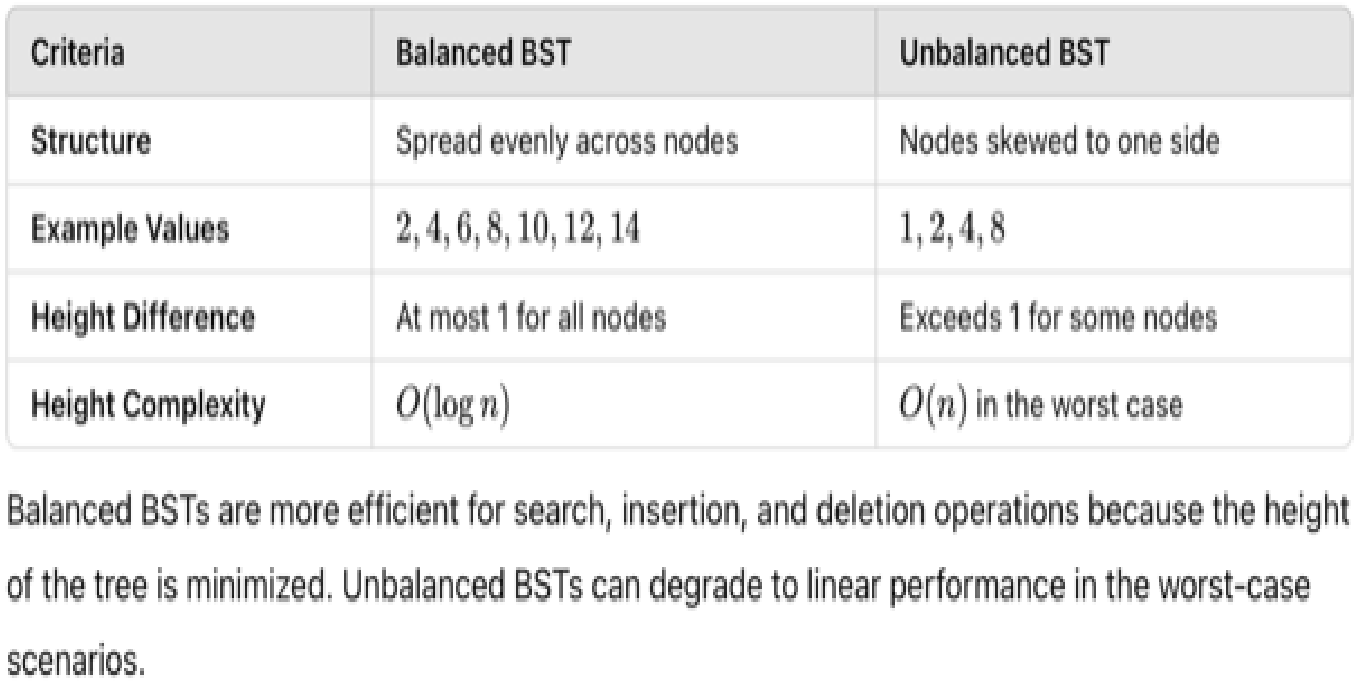


Balanced BST



Unbalanced BST

Comparison: Balanced BST and Unbalanced BST



Criteria	Balanced BST	Unbalanced BST
Structure	Spread evenly across nodes	Nodes skewed to one side
Example Values	$2, 4, 6, 8, 10, 12, 14$	$1, 2, 4, 8$
Height Difference	At most 1 for all nodes	Exceeds 1 for some nodes
Height Complexity	$O(\log n)$	$O(n)$ in the worst case

Balanced BSTs are more efficient for search, insertion, and deletion operations because the height of the tree is minimized. Unbalanced BSTs can degrade to linear performance in the worst-case scenarios.

Binary Search Tree Review

Binary Search Tree