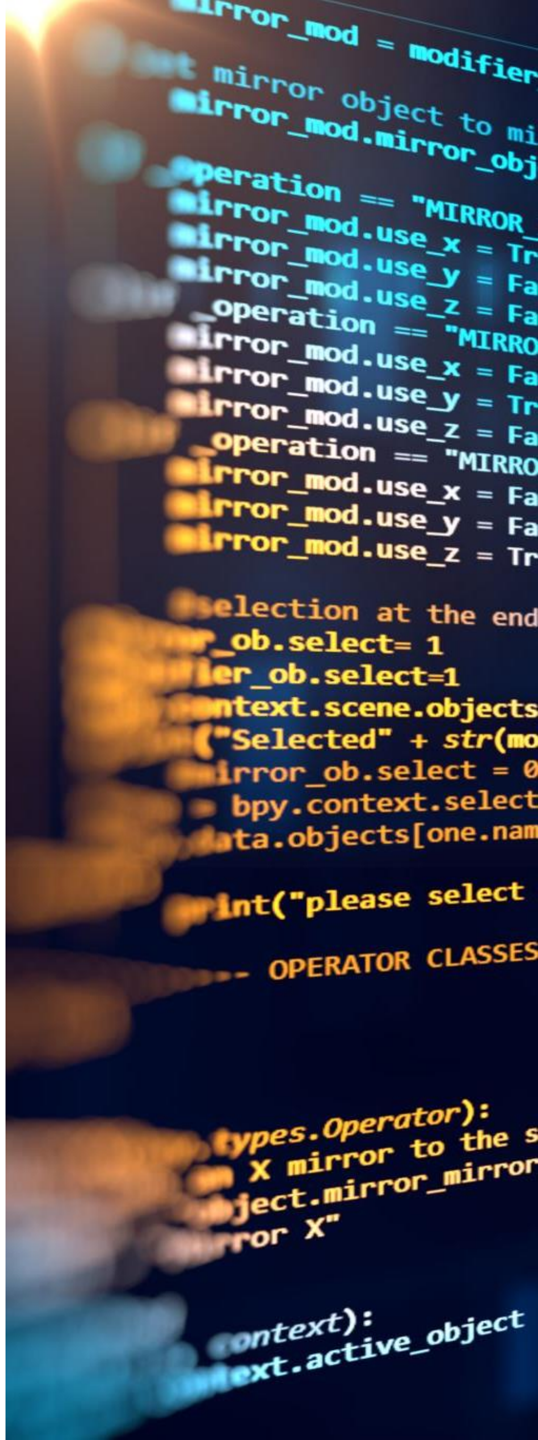
Please use the following QR code to check in and record your attendance.

CS 1037
Fundamentals of Computer Science II

AVL Tree

Self-Balancing Binary Search Tree

Ahmed Ibrahim

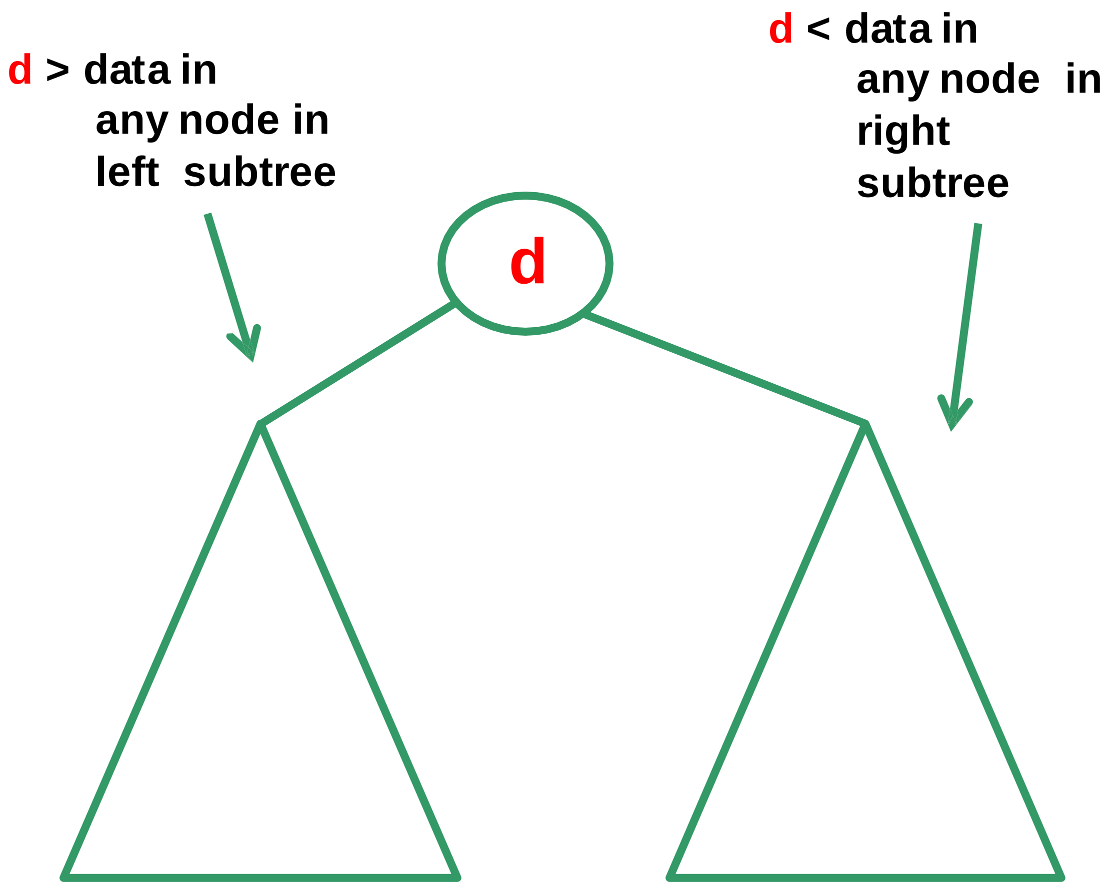


Recall: Binary Search Trees (BST)

- What is a Binary Search tree?

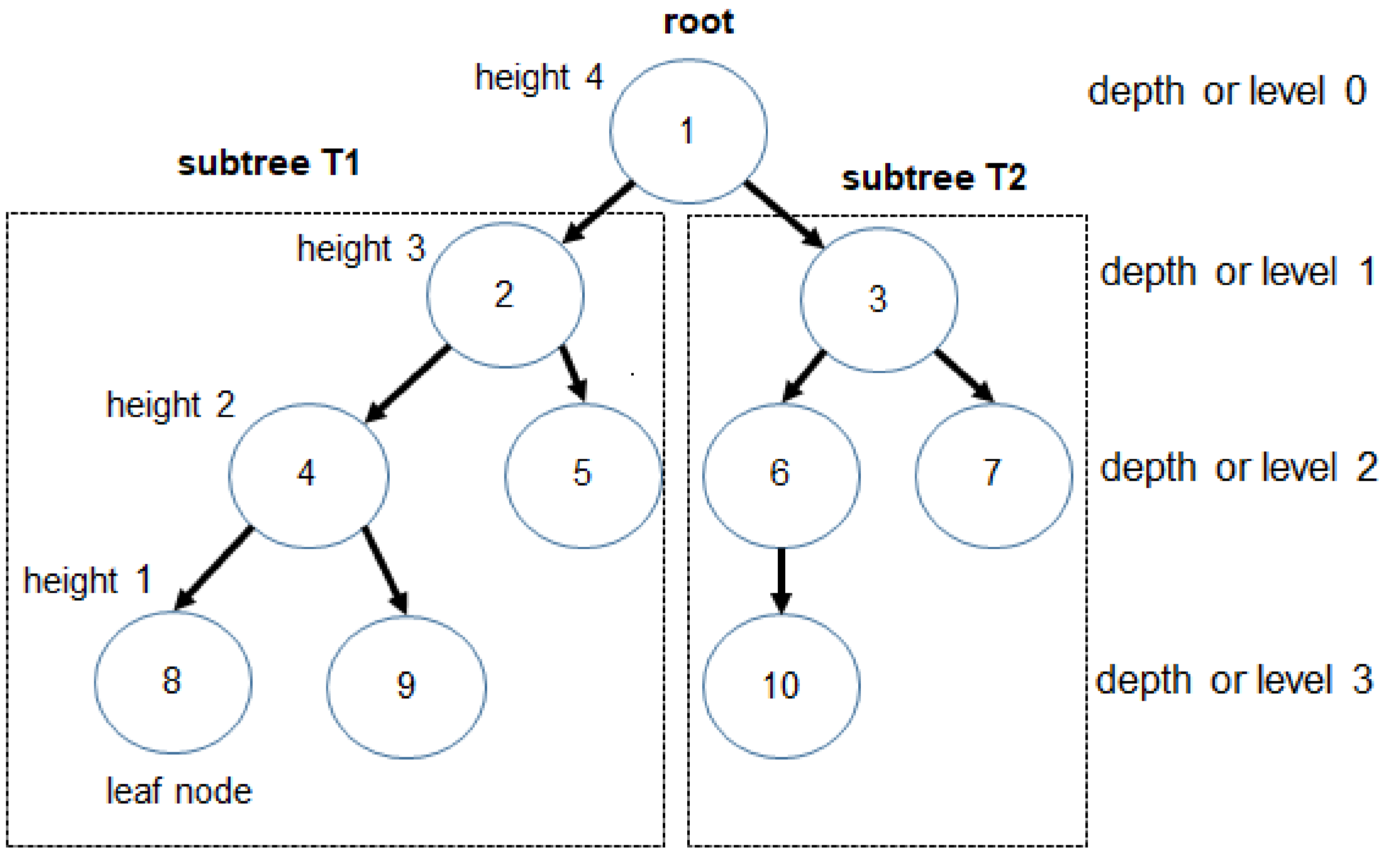
  A binary search tree is a **binary tree** in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

- Every BST is a BT, but every BT must not be a BST.
- There must be no duplicate nodes (in general).


d > data in any node in left subtree
d < data in any node in right subtree
d

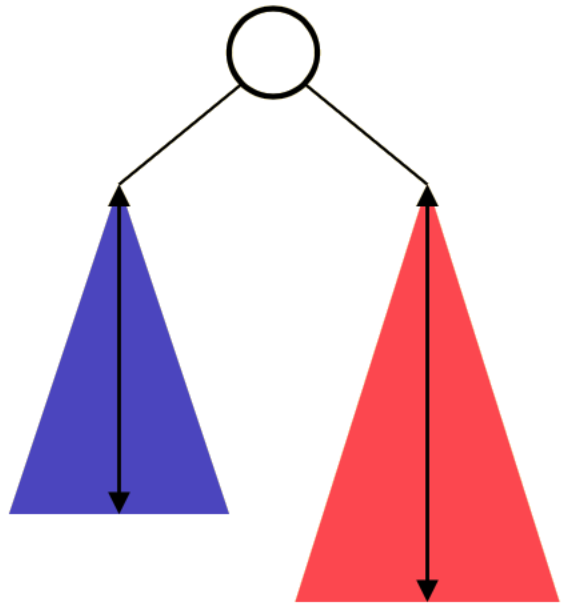
Recall: Tree Terminology

- The **depth** of a node is the number of ancestors
- The **height** of a tree is the maximum depth from the root
  - 3 in this example
- **Descendant** of a node is a child, grandchild, grand-grandchild, etc.
- **Siblings** are two nodes that are children of the same parent



Balanced Tree Motivation

- A tree is considered **balanced** if, for any node, the heights of its left and right subtrees are **not too different**. In other words, the difference in height between the left and right subtrees remains within a specific limit.
- Balance is essential for good Binary Search Tree (BST) performance. A BST can become highly unbalanced if nodes are inserted in sorted order.
- Self-balancing trees (like **AVL trees**) address this issue by maintaining balance through **rotations**, which adds extra complexity but ensures efficient operations.



AVL Tree

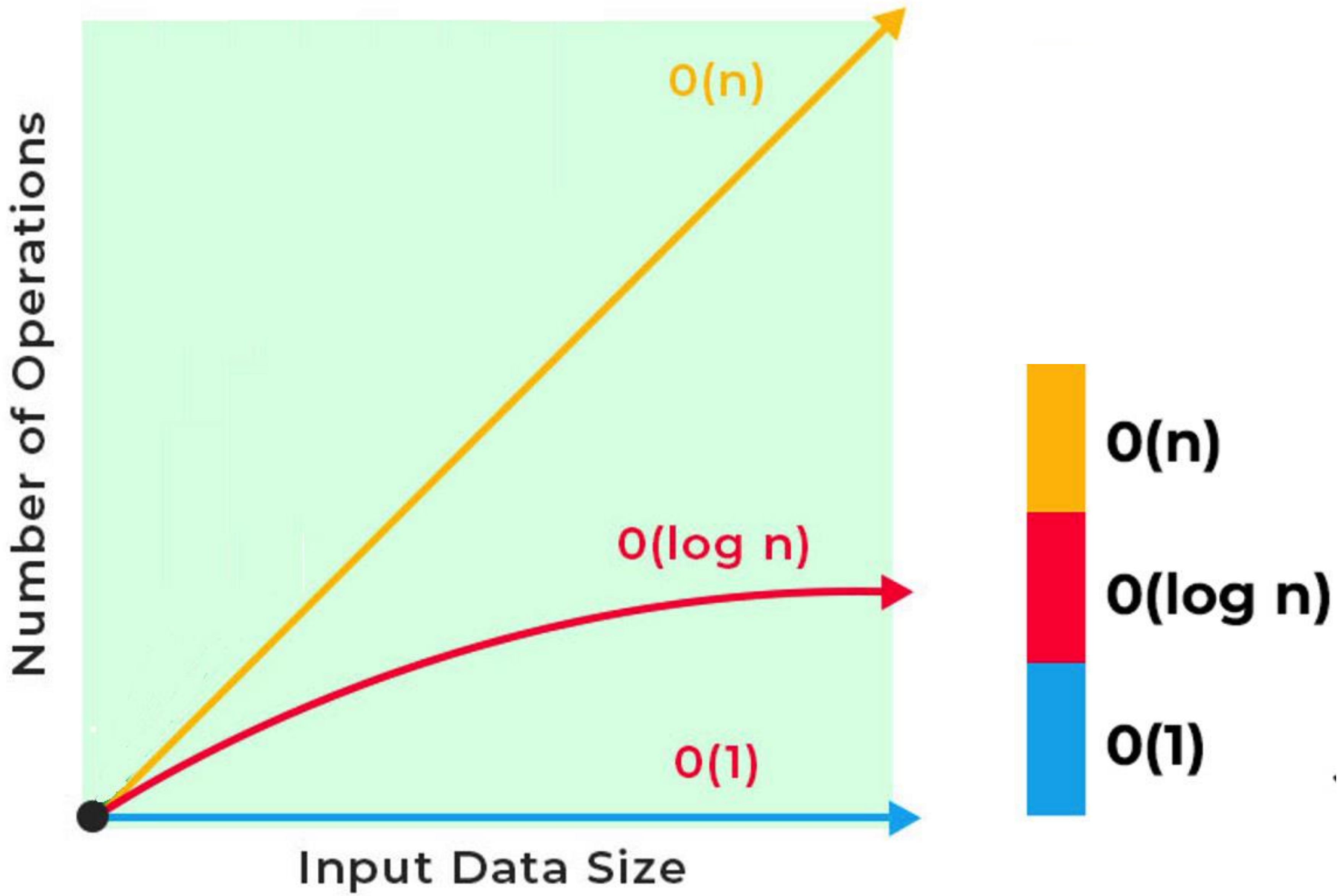
- AVL tree is a **self-balancing binary search tree** in which the **heights** of the two sub-trees of a node may differ by at most one. Because of this property, the AVL tree is also known as a **height-balanced tree**.
- The key advantage of using an AVL tree is that it takes O(***log n***) time to perform search, insertion, and deletion operations in both the average and worst cases**.**
    - *(log n)* the time it takes to complete an operation grows ***logarithmically*** with the input size, n. By "***logarithmically***," we mean that you cut down the number of steps we must look through by half.

What is Big O Notation?

- Big O notation is a way to measure **how long an algorithm takes to run** as the size of the input grows. Think of it as a tool to understand **how "fast" or "slow"** different algorithms are, especially when handling a lot of data.
- Big O notation allows us to predict the behaviour of different algorithms and select the most efficient one, particularly when working with large datasets.

Measuring Time Growth

- Big O notation looks at how the **time** (or number of steps) an algorithm takes grows as the **input size** grows. We're not focusing on exact times; we care about the **general trend** or **pattern**.
- For example, let's say we have a list of $\boldsymbol{n}$ items:
- If each item takes the same time to process, **doubling the list size doubles the time**. We call this **linear time**, or O(n).
- If we can cut the list in half with each step (like in binary search), the time growth of the running algorithm will be much slower. This is called ***logarithmic time***, or O($\log n$).



Source: https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/

Examples

- The following are code snippets of constant time O(1) and linear time O(n):

```
#include <stdio.h>
void constantTimeExample(int arr[], int n) {
printf("First element: %d\n", arr[0]);
}

int main() {
int arr[] = {1, 2, 3, 4, 5};
int n = sizeof(arr) / sizeof(arr[0]);
constantTimeExample(arr, n);
return 0;}
```

The time complexity is O(1)

```
#include <stdio.h>
void linearTimeExample(int arr[], int n) {
int sum = 0;
for (int i = 0; i < n; i++) {sum += arr[i];}
printf("Sum of elements: %d\n", sum);}

int main() {
int arr[] = {1, 2, 3, 4, 5};
int n = sizeof(arr) / sizeof(arr[0]);
linearTimeExample(arr, n);
return 0;}
```

The time complexity is O(n)

Recall: AVL Tree

- AVL tree is a **self-balancing binary search tree** in which the **heights** of the two sub-trees of a node may differ by at most one. Because of this property, the AVL tree is also known as a **height-balanced tree**.
- The key advantage of using an AVL tree is that it takes O(***log n***) time to perform search, insertion, and deletion operations in both the average and worst cases**.**
  - *(log n)* the time it takes to complete an operation grows ***logarithmically*** with the input size, n. By "***logarithmically***," we mean that you cut down the number of steps we must look through by half.

The Structure of An AVL tree

```
typedef struct AVLNode {
int key; // The value or key of the node
int height; // The height of the node for balance calculations
struct AVLNode* left; // Pointer to the left child
struct AVLNode* right; // Pointer to the right child
} AVLNode;
```

- The **structure** of an **AVL tree** is the same as that of a **binary search tree** but with a slight difference**. I**t stores an additional variable called the ***height***.
- We use the ***height*** to calculate the **balance factor** at each node.

Balance Factor (cont.)

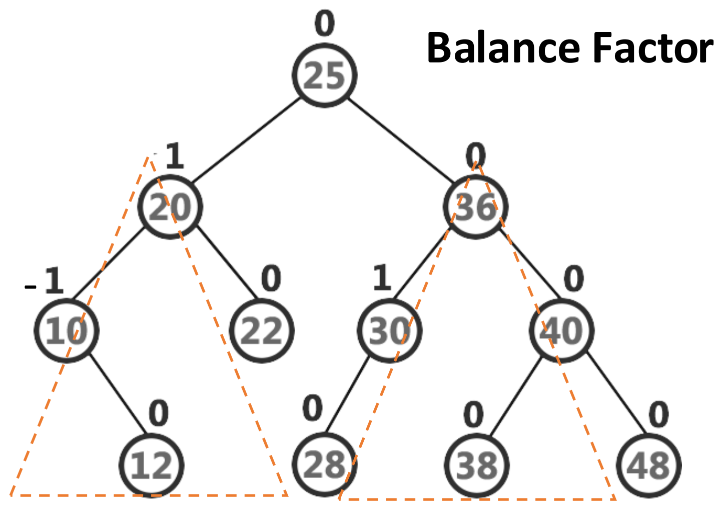
- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.

  Balance factor = Height (left sub-tree) – Height (right sub-tree)

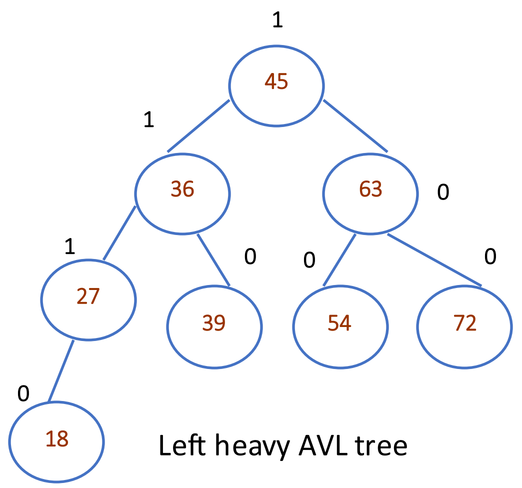
- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is **one level higher** than that of the right sub-tree. Such a tree is called a ***Left-heavy tree***.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree is equal to the height of its right sub-tree.
- If the balance factor of a node is -1, then it means that the left sub-tree of the tree is **one level lower** than that of the right sub-tree. Such a tree is called a **Right-*heavy tree***.

Balance Factor Values on Nodes

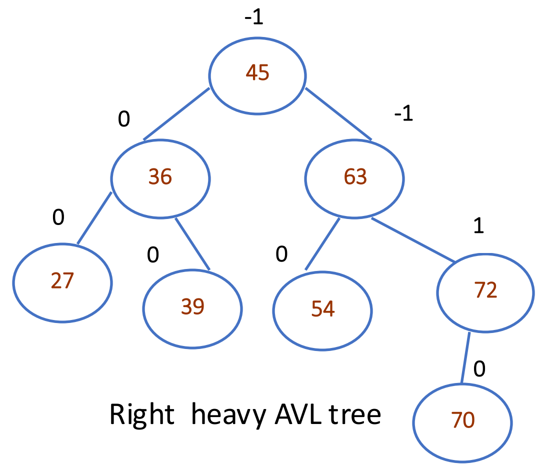
- Example
  - **Node 10**: Left height = 0 (no right child), right height = 1, so balance factor = -1.
  - **Node 20**: Left height = 2 (through 10), right height = 1 (child 22), so balance factor = 1.
  - **Node 25**: Left height = 3 (subtree rooted at 20), right height = 3 (subtree rooted at 36), so balance factor = 0.



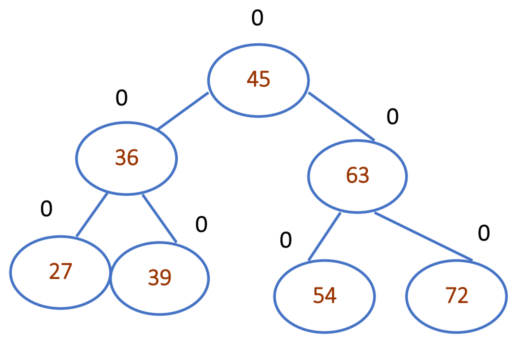
Heavy AVL Types



Left heavy AVL tree



Right heavy AVL tree

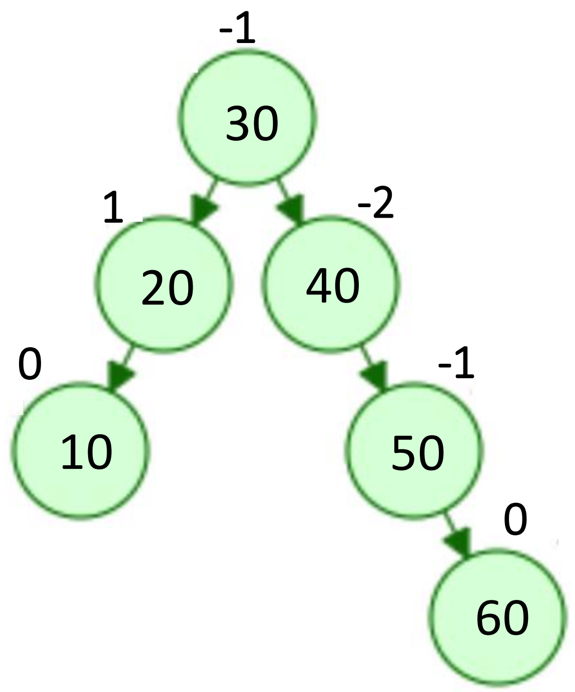


Balanced AVL tree

Question!

- Consider the following AVL tree structure where each node is represented as (N, H, B), where N is the node's key, H is its height, and B is its balance factor:

- Given this structure:
  1. Calculate the height (H) and balance factor (B) for each node in the tree.
  2. Determine whether this tree satisfies the balance conditions of an AVL tree.
  3. If it does not satisfy the AVL conditions, identify the first node that violates the AVL property.



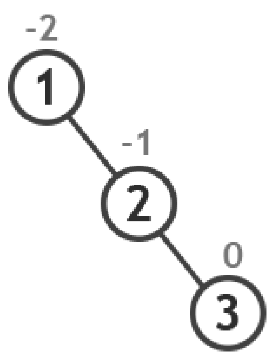
The AVL property is violated at **Node 40** with a balance factor of -2, as AVL trees require balance factors to be between -1 and +1.

Self-Balancing

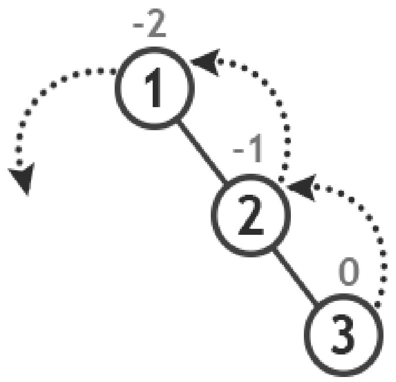
- An AVL tree automatically performs **rotations** to keep its height *logarithmic* relative to the number of nodes ($\log(n)$), ensuring efficient operations.
- This property is crucial for managing large datasets by minimizing the distance from the root to any leaf.
- **Rotations for Balance**
    - **Single Rotation** includes left and right rotations, which are used to balance simple imbalances (LL and RR cases).
        - LL: an element was inserted into the left subtree of its left child
        - RR: an element was inserted into the right subtree of its right child.
    - **Double Rotations**: For complex cases (LR and RL), AVL trees use a combination of left and right rotations to maintain balance.

Example of Left Rotation

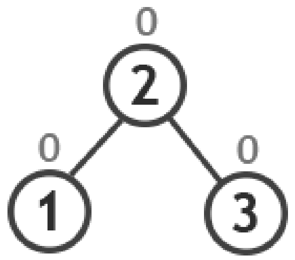
Right heavy AVL tree



a) Tree is imbalanced

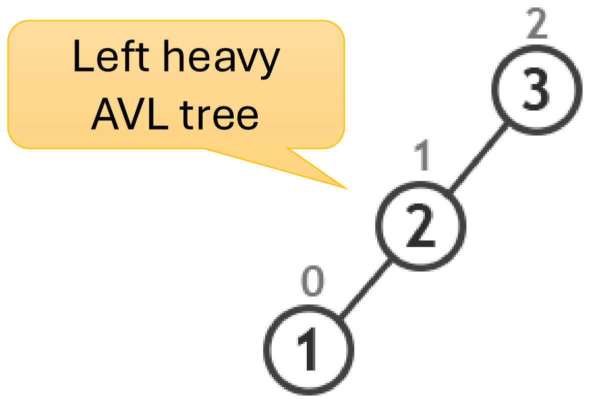


b) Single left rotation

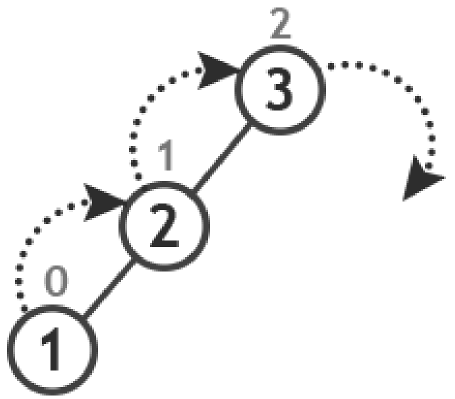


c) Balanced Tree

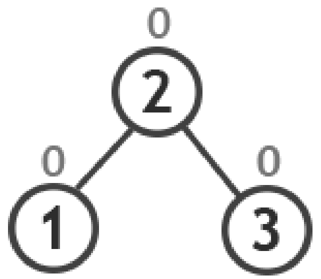
Example of Right Rotation



a) Tree is imbalanced



b) Single right rotation



c) Balanced Tree

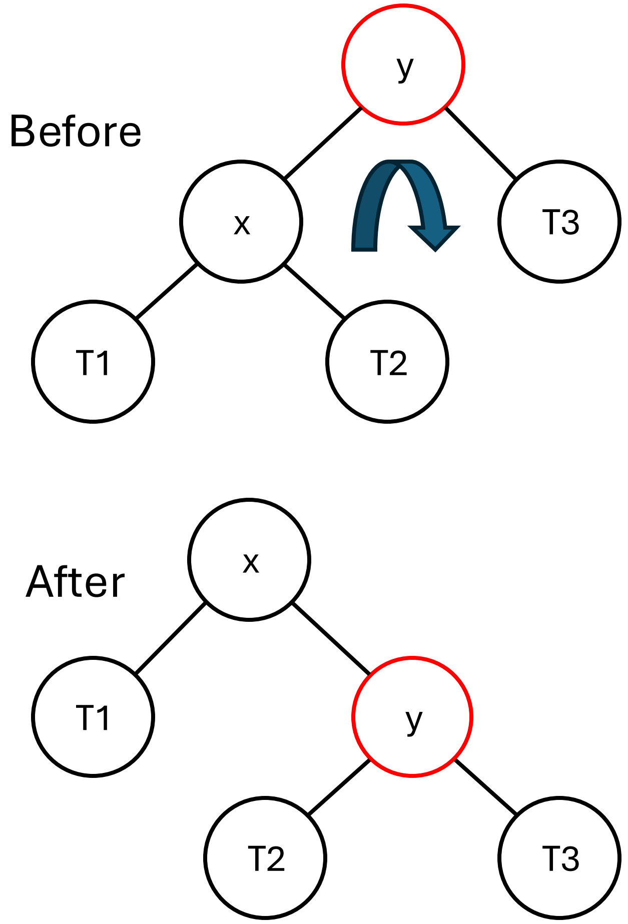
Right Rotation Algorithm

```
// Right rotate subtree rooted with y
AVLNode* rightRotate(AVLNode* y) {
 AVLNode* x = y->left;
 AVLNode* T2 = x->right;

 // Perform rotation
 x->right = y; y->left = T2;

 // Update heights
 y->height = max(height(y->left), height(y-
>right)) + 1;
 x->height = max(height(x->left), height(x-
>right)) + 1;

 return x; // Return new root
}
```



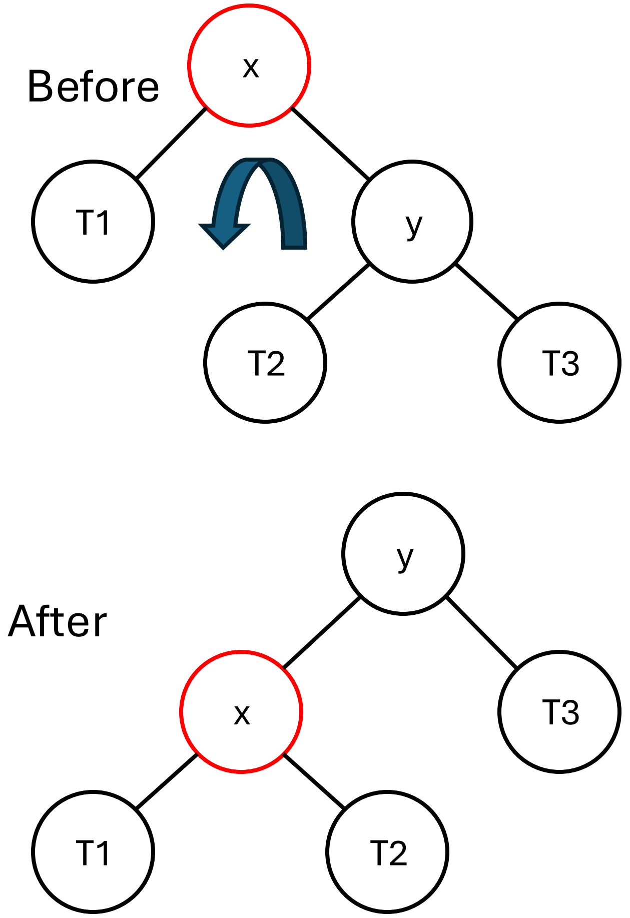
Left Rotation Algorithm

```
// Left rotate subtree rooted with x
AVLNode* leftRotate(AVLNode* x) {
AVLNode* y = x->right;
AVLNode* T2 = y->left;

// Perform rotation
y->left = x;
x->right = T2;

// Update heights
x->height = max(height(x->left), height(x->right)) + 1;
y->height = max(height(y->left), height(y->right)) + 1;

return y; // Return new root
}
```

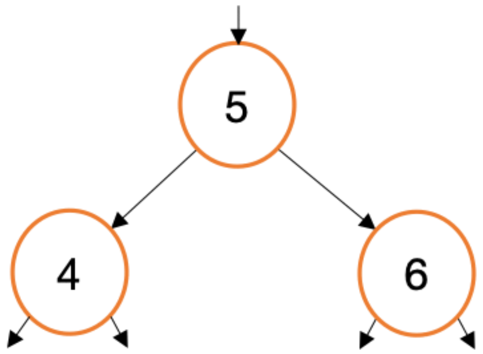


Searching for a Node in an AVL Tree

- **Searching** in an AVL tree is performed the same way as in a **binary search tree**.
- Because of the height-balancing of the tree, the search operation takes *O(log n)* time to complete.
- Since the operation does not modify the structure of the tree, no special provisions need to be taken.

Recall: Insertion in a Binary Tree

- In a binary tree, nodes are added **level** by **level** from **left** to **right**, typically at the first available position.



Add a new value, 3, to existing tree of three nodes

```
Algorithm (Level-order):
1. Start from the root node.
2. Use a queue to traverse the tree level by
level.
For each node:
   4. Check if it has a left child; if not,
   insert the new node here and stop.
   5. If there is a left child, move to the
   right child.
   6. If there is no right child, insert the
   new node here.
   If both left and right children exist, add
   them to the queue to continue the
   traversal.
```

Source: https://www.gormanalysis.com/blog/making-a-binary-search-tree-in-cpp/

Insertion for a Node in an AVL Tree

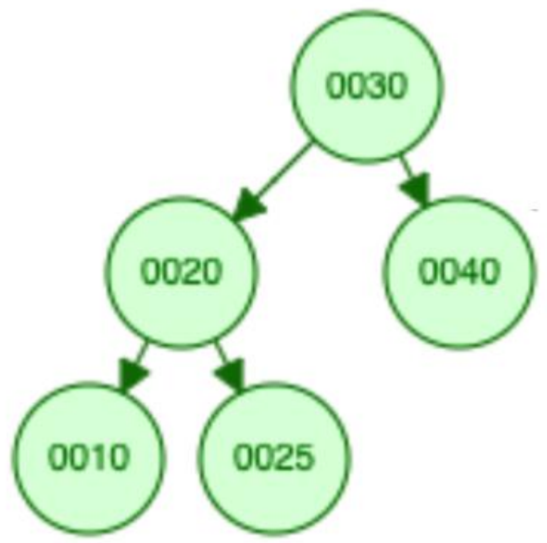
- Like in binary search trees, the new node is always inserted as a leaf node. But the insertion step is usually followed by an additional step to restore the balance.
- **Rotation** is done to restore the balance of the tree.
- However, if insertion of the new node does not disturb the **balance factor**, that is, if the balance factor of every node is still -1, 0 or 1, then rotations are not needed.

Example: Inserting Nodes into an AVL Tree

- Suppose we start with an empty AVL tree and want to insert the following nodes in this order: 30, 20, 40, 10, 25.
- Step-by-Step Insertion and Balancing
  - Insert 30:
    - Since the tree is empty, 30 becomes the root node.
    - No rotation is needed because there's only one node, and the tree is balanced.
  - Insert 20:
    - 20 is less than 30, so it becomes the left child of 30.
    - The balance factor of node 30 is 1 (left height - right height = 1 - 0 = 1).
    - No rotation is needed since all nodes have balance factors within -1, 0, or 1.
  - insert 40:
    - 40 is greater than 30, so it becomes the right child of 30.
    - Now, the balance factor of node 30 is 0 (left height - right height = 1 - 1 = 0).
    - No rotation is needed as the tree is balanced .

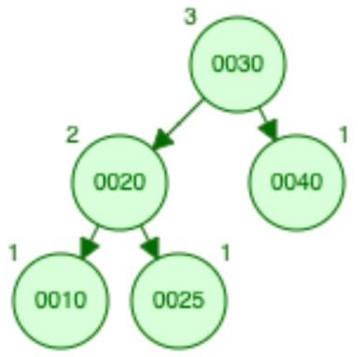
Example: Inserting Nodes into an AVL Tree

- Insert 10:
  - 10 is less than 30 and also less than 20, so it becomes the left child of 20.
  - The balance factors are as follows:
  - Node 10: 0 (no children)
  - Node 20: 1 (left height - right height = 1 - 0 = 1)
  - Node 30: 1 (left height - right height = 2 - 1 = 1)
  - The tree is still balanced, so no rotation is needed.
- Insert 25:
  - 25 is less than 30 but greater than 20, so it becomes the right child of 20.
  - After this insertion, the balance factors are:
  - Node 10: 0 (no children)
  - Node 25: 0 (no children)
  - Node 20: 0 (left height - right height = 1 - 1 = 0)
  - Node 30: 1 (left height - right height = 2 - 1 = 1)
  - The tree is balanced, so no rotation is needed.



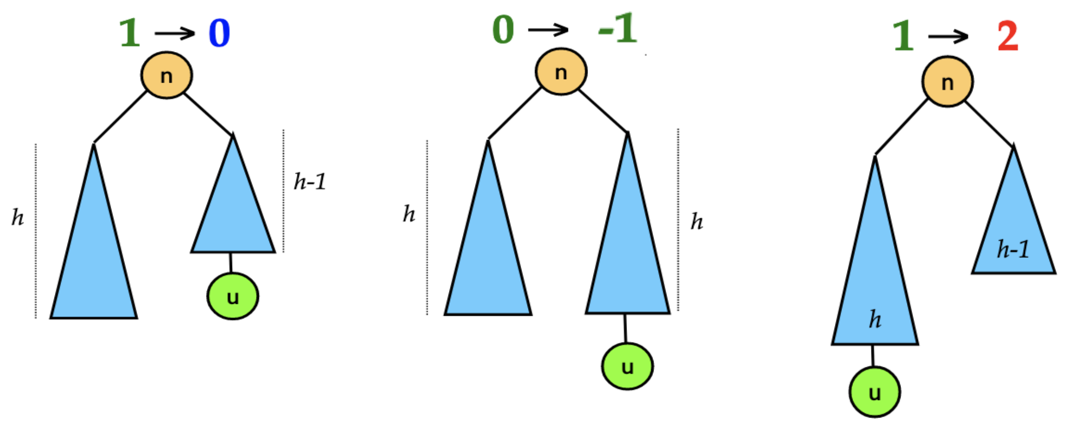
Causing Rotation

- If we now insert 5 into the tree:
  - Insert 5: This node will be the left child of 10.
  - Balance factors:
    - Node 5: 0
    - Node 10: 1
    - Node 20: 2 (left-heavy, causing an imbalance)
    - Since node 20's balance factor is now 2, a right rotation around it would be needed to restore balance.



Possible AVL Tree states with Insertion

- After insertion, the tree has become **balanced**.
- After insertion, the tree has become either **left** or **right-heavy**.
- Initially, the tree was heavy (either left or right), and the new node was inserted in the heavy sub-tree, thereby creating an imbalanced sub-tree.



Algorithm of Inserting a Node in an AVL Tree

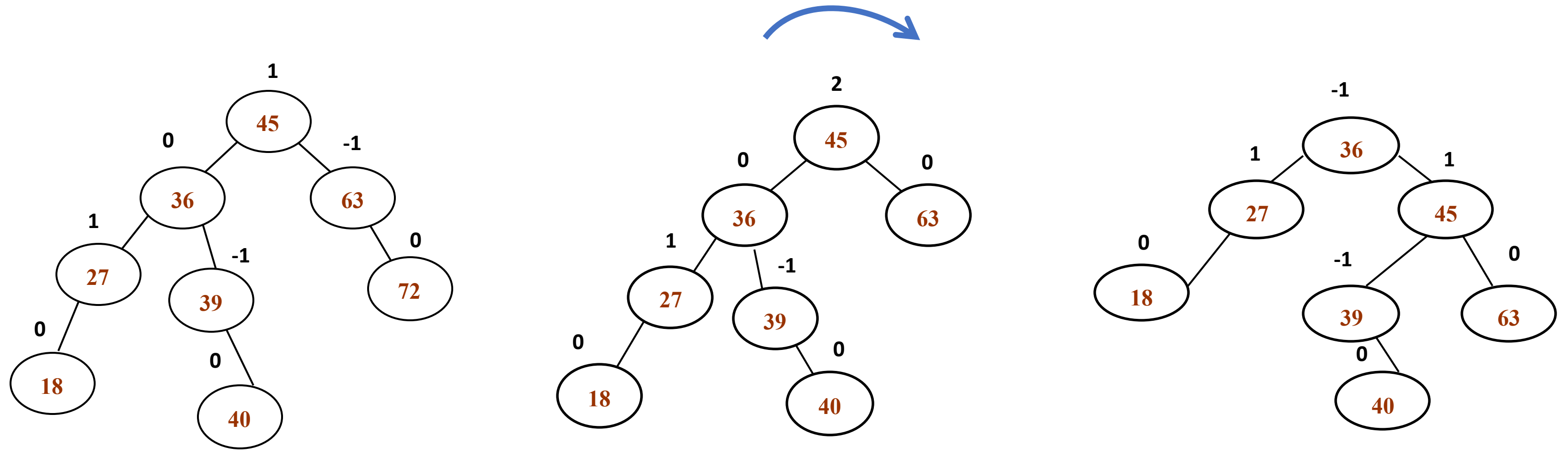
- Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
- Step 2 - After insertion, check the **Balance Factor** of every node.
- Step 3 - If the **Balance Factor** is **0 or 1, or -1,** then go for the next operation.
- Step 4 - If the **Balance Factor** is other than **0 or 1 or -1,** then that tree is said to be imbalanced. In this case, perform a suitable **Rotation** to make it balanced and go for the next operation.

Deleting a Node from an AVL Tree

- Deletion of a node in an AVL tree is **similar** to that of binary search trees.
- But deletion may disturb the AVLness of the tree, so to re-balance the AVL tree, we need to perform rotations.

Deleting a Node from an AVL Tree

- Consider the AVL tree given below and delete 72 from it.



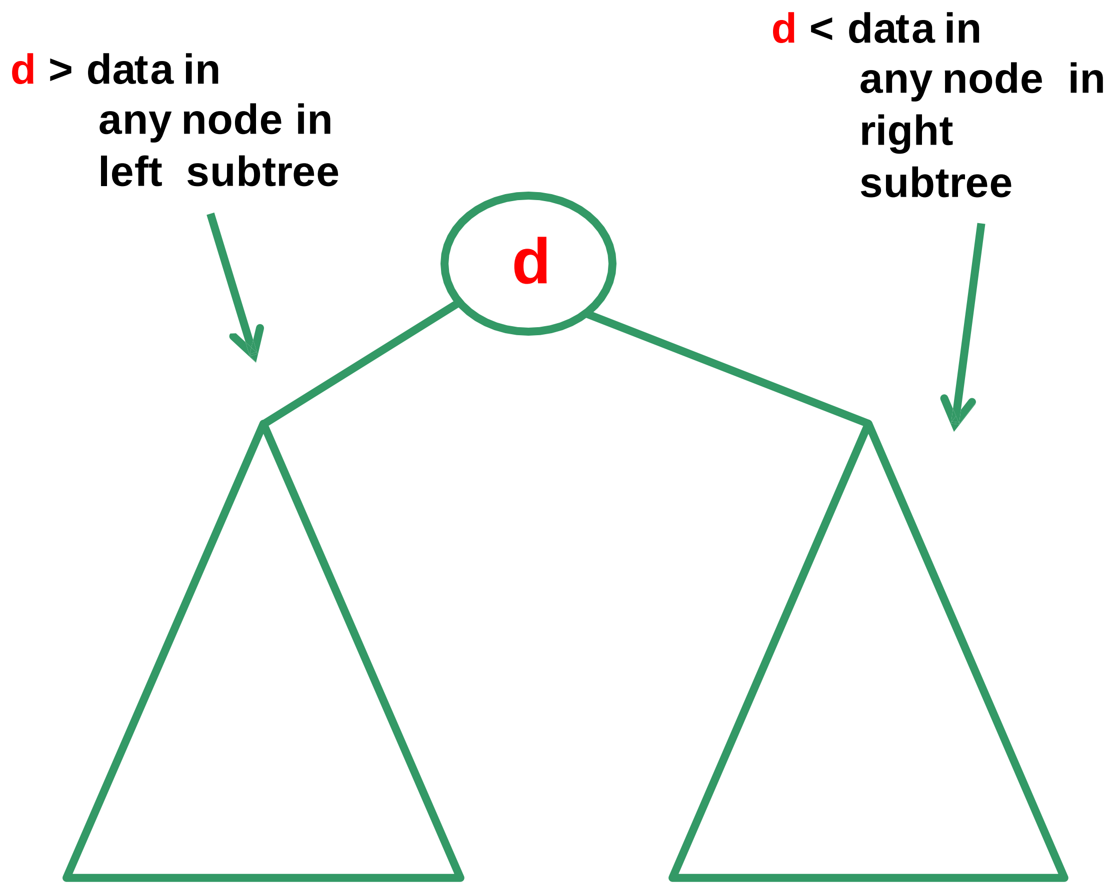
Multi-way Search Trees

Recall: Binary Search Trees (BST)

- What is a Binary Search tree?

  A binary search tree is a **binary tree** in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

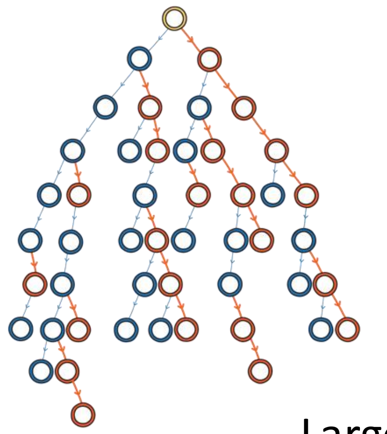
- Every BST is a BT, but every BT must not be a BST.
- There must be no duplicate nodes (in general).

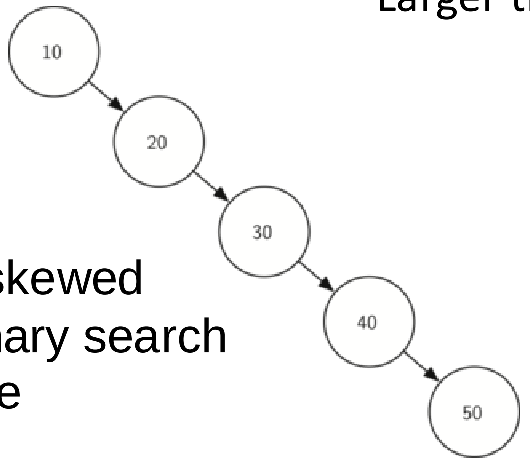

d > data in any node in left subtree
d
d < data in any node in right subtree

BTs Drawbacks

- In a BST, each node can have **only two children**, so the tree's height grows quickly as more nodes are added. For large datasets, this height increase makes searches slower, as it takes more steps to reach a **leaf** from the **root**.

- **Balance** is essential for good performance in a BST. If nodes are inserted in a sorted order (like ascending or descending), the tree can easily become unbalanced. Self-balancing trees (like AVL) solve this issue but add extra complexity and require rebalancing.


Larger tree

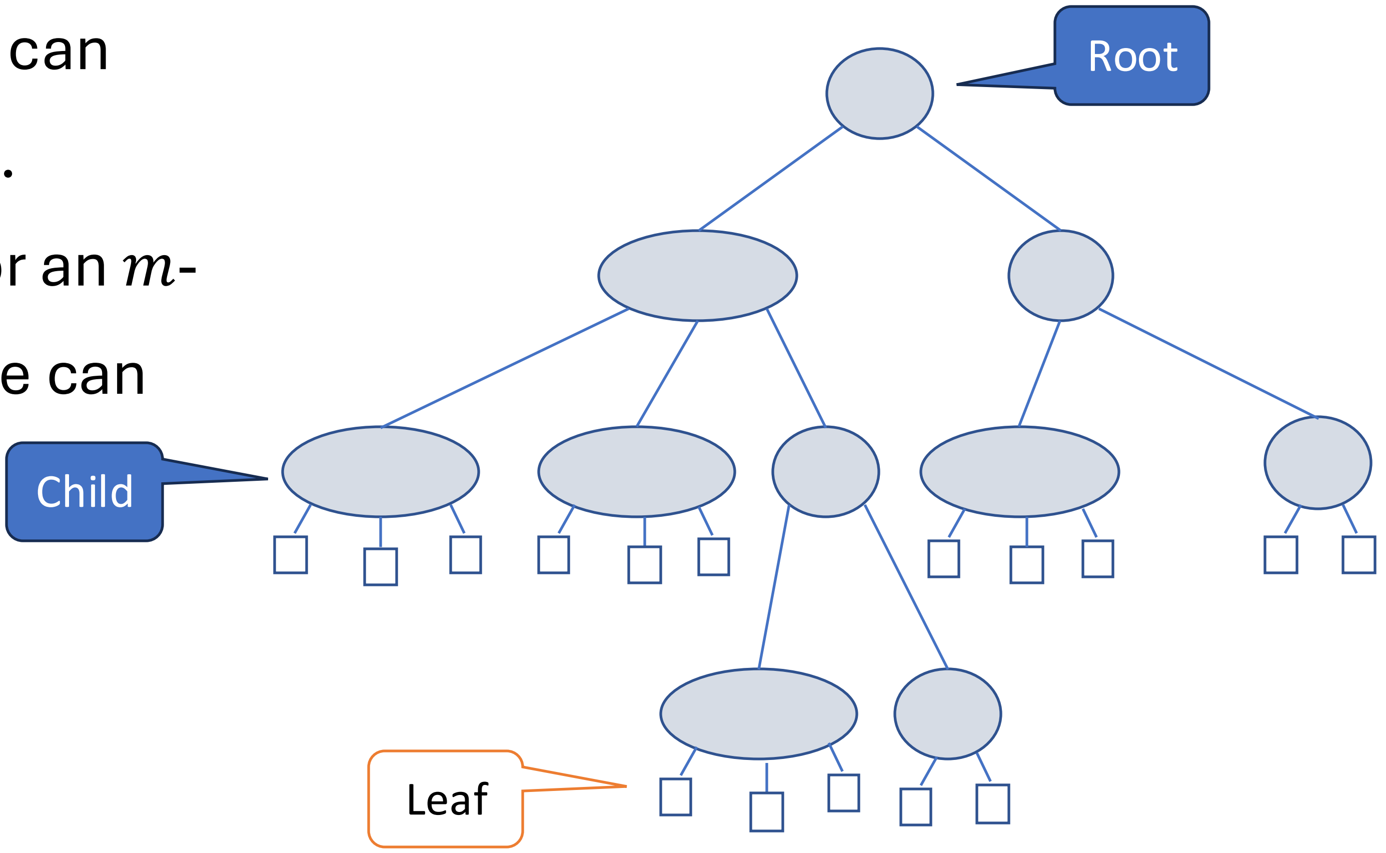


A skewed binary search tree

The concept of Multi-way Search Trees

- A multi-way tree is a tree that can have more than **two children**.
- A multi-way tree of order m (or an m-way tree) is one in which a tree can have m children.
- Legend:

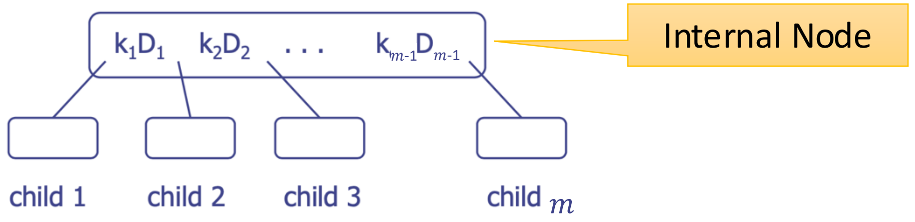
Internal Node	▨
External Node	□

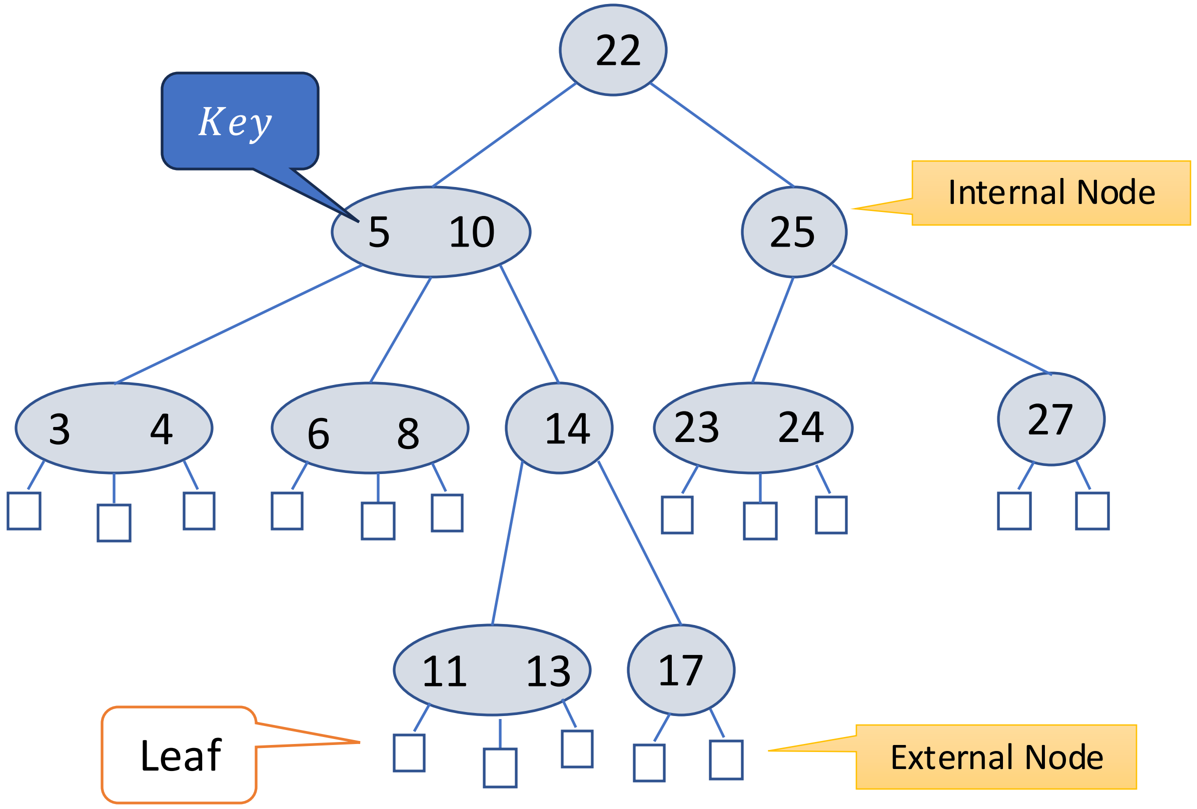


Example of 3-way tree

Multi-way Search Tree

- As with the other studied trees, the nodes in an m-way tree consist of $m-1$ ***entries*** and pointers to children.
- The **entries** are in the form of pairs (k, D) where k is the *key* and D is the value (*data*) associated with the *key*.
- The external nodes of a m-way search tree do not store any *entries* and are "*dummy*" nodes.





Example of 3-way tree

The Structure of Multi-Way Search Trees

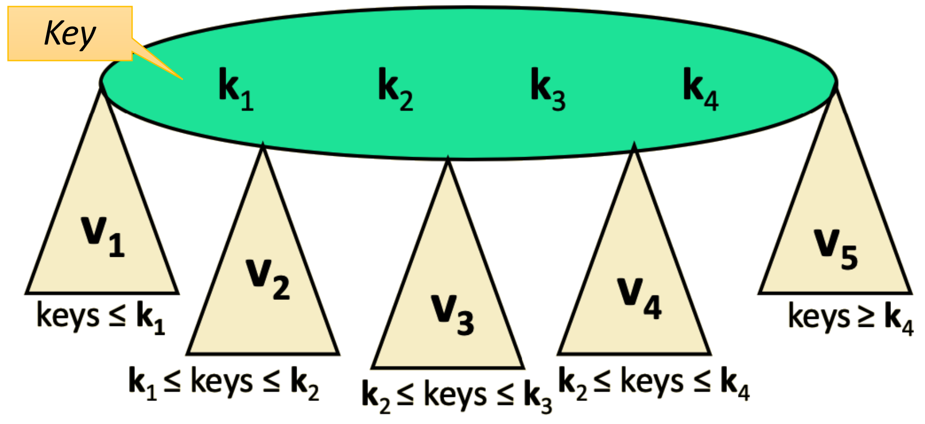
The following structure defines a simple m-way node structure, where m (>1) is a predefined constant.

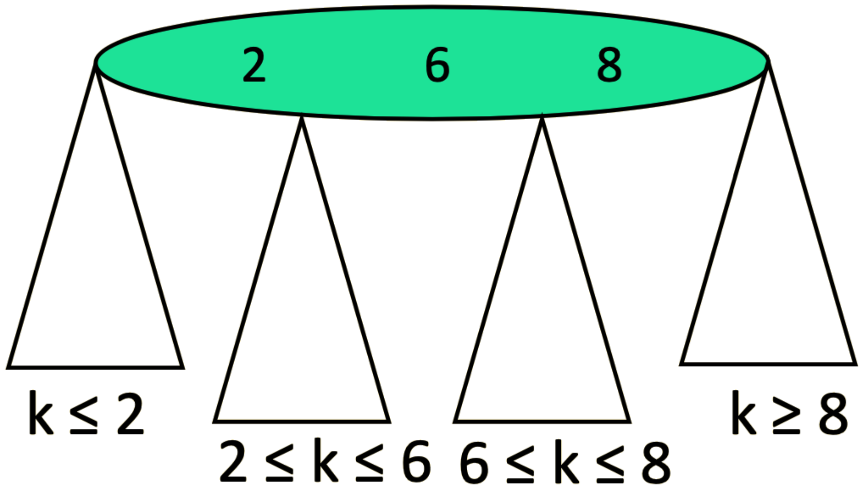
```
typedef struct node {
int count; // number of key values
int key[m-1]; // key arrays
struct node *child[m]; // sub-tree pointer array
} TNODE;
```

Properties of Multi-Way Search Trees

- A multi-way search tree is an ordered tree such that
  - Each **internal node** has at least two and at most $\boldsymbol{m}$ children and stores $\boldsymbol{m}$***-1*** data items
  - **External nodes** have zero data items
  - Number of children = 1 + number of data items in a node
    - A node with three children is called a 3-node.

An Internal Node

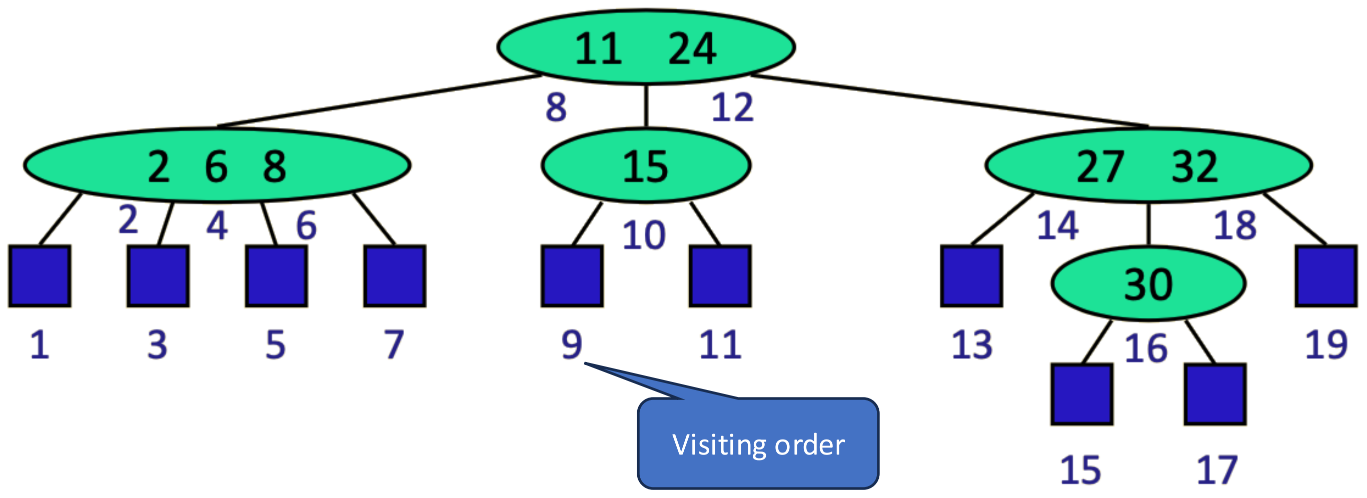


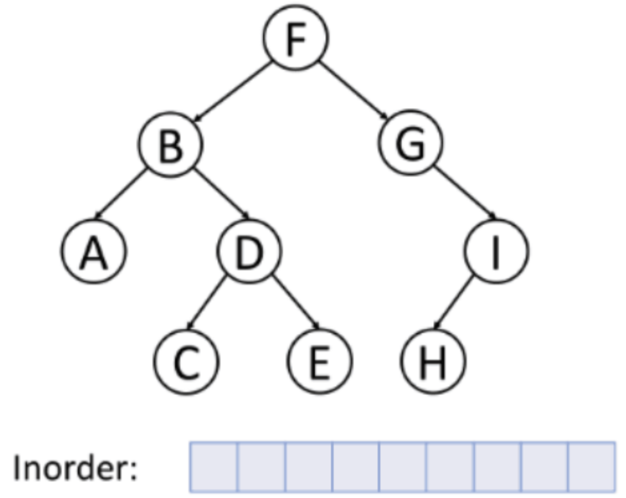


Numerical Example

- Each internal node has $m \geq 2$ children and stores m-1 entries.

Multi-Way Tree Traversal



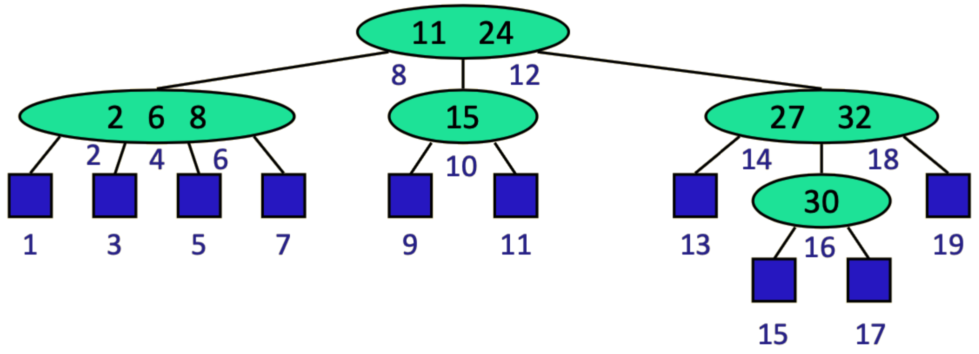


- In a multi-way search tree, in-order traversal involves visiting all keys in sorted order by **recursively** traversing child nodes and printing keys. Unlike binary trees, multi-way search trees may have more than two children per node, so the traversal is slightly different.
- An **in-order** traversal of a multi-way search tree visits the keys in **increasing order**.

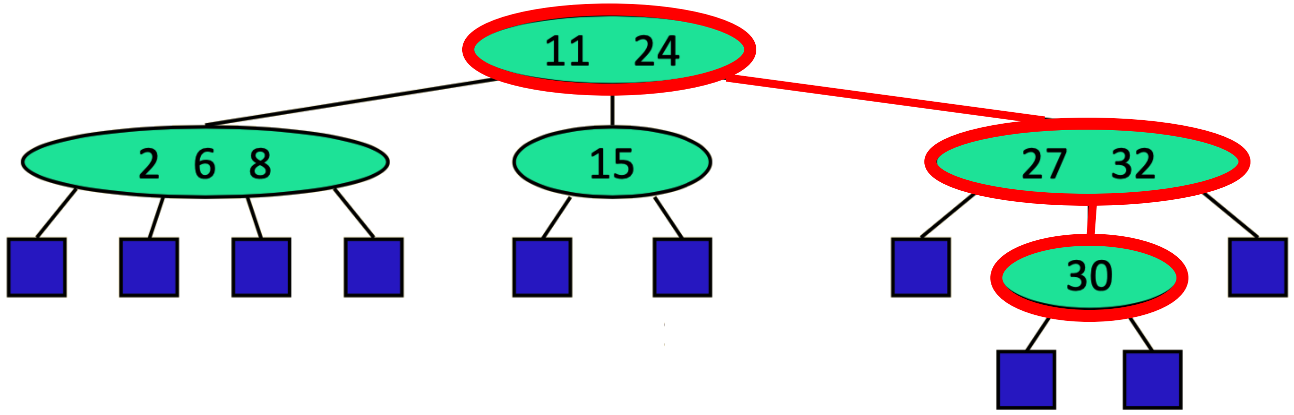
Multi-Way Tree Traversal Algorithm

- The following program is an example of in-order traversal with printing key values.

```
/* in-order traversal of m-way tree*/
void print_inorder(TNODE *root) {
  if (root != NULL) {
  // Traverse the first child subtree
    print_inorder(child[0]);
    int i;
    for (i=0; i < root->count; i++) // Traverse through each key in the node
    {
      printf("%d ", root->key[i]; // Print the current key
      print_inorder(child[i+1]);} // Traverse the next child subtree
     }
}
```

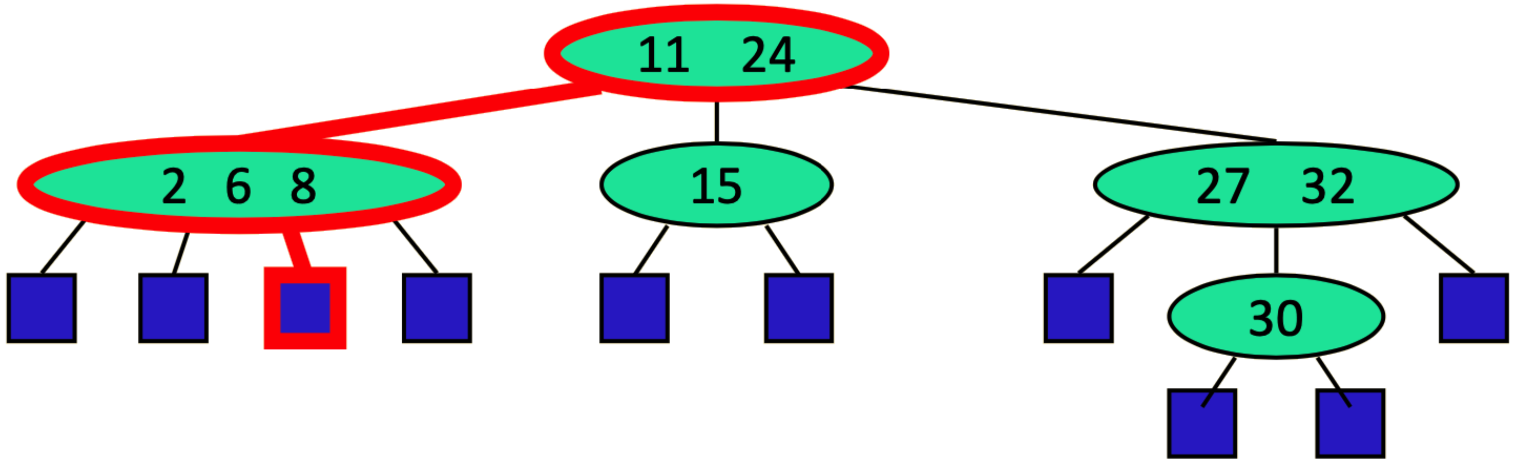


Multi-Way Search



- Assuming $\boldsymbol{m}$ is a constant independent of a number of nodes, examining each node takes constant time (*O(1)*); thus, the time to search is **proportional** to the tree height.
- Imagine that we are searching for *k* = 30

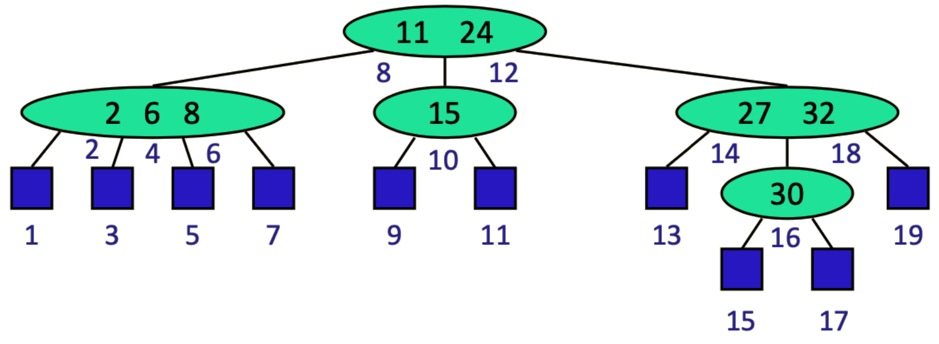
Another Example: Multi-Way Searching



- Search for key 7
  - Search terminates at a leaf child, which implies there is no entry with key 7

Multi-Way Searching Pseudocode

```
Algorithm get(r,k)
In: Root r of a multiway search tree, key k
Out: data for key k or null if k not in tree
if r is a leaf then return null
else {
    Use binary search to find the index i such that either
        • r.keys[i] = k, or
        • r.keys[i] < k < r.keys[i+1]
    if k = r.keys[i] then return r.data[i]
    else return get(r.child[i],k)
}
```



Thank you