Module 02 – Data Structure's Effects on Performance

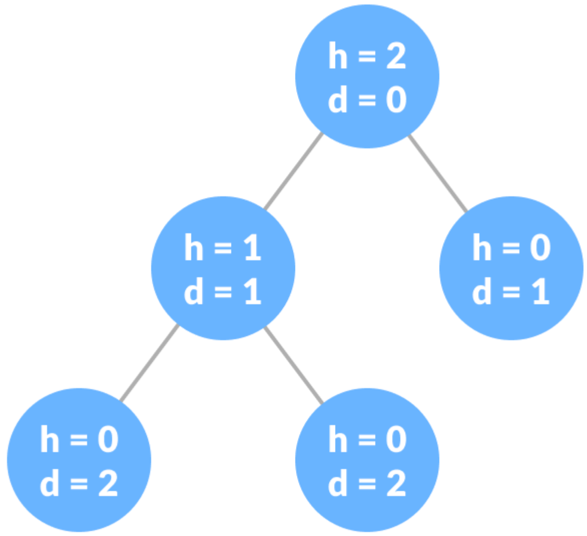
Foundations Slide Deck

- Searching: most common operation performed by a database system
- Baseline for efficiency is Linear Search
    - Start at the beginning of a list and proceed element by element until:
        - You find what you're looking for
        - You get to the last element and haven't found it
- Arrays:
    - Fast for random access BUT slow for random insertions
- Linked list: records are linked through a chain of memory addresses
    - Slow for random access BUT fast for random insertions
    - each node stores the data and the address of the next node.
- Binary search

```
def binary_search(arr, target)
  left, right = 0, len(arr) - 1
  while left <= right:
    mid = (left + right) // 2
    if arr[mid] == target:
      return mid
    elif arr[mid] < target:
      left = mid + 1
    else:
      right = mid - 1
  return -1
```

Binary Search Tree - a binary tree where every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent.

- Node: entity that contains key or value and pointers to child nodes
- Last nodes of each path are called leaf nodes (they do not contain pointer to child node)
- Edge: link between any two nodes
- Root: topmost node of a tree



- Height: # of edges
- Binary trees can have at most two children

<u>Traversal - visit all nodes in a graph</u>

- Inorder: Left → Root → Right   Print after visiting left subtree
- Preorder: Root → Left → Right   Print before visiting subtrees
- Postorder: Left → Right → Root   Print after visiting both subtrees

```
# Binary Tree in Python

class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

    # Traverse preorder
    def traversePreOrder(self):
        print(self.val, end=' ')
        if self.left:
            self.left.traversePreOrder()
        if self.right:
            self.right.traversePreOrder()

    # Traverse inorder
```

```python
    def traverseInOrder(self):
        if self.left:
            self.left.traverseInOrder()
        print(self.val, end=' ')
        if self.right:
            self.right.traverseInOrder()

    # Traverse postorder
    def traversePostOrder(self):
        if self.left:
            self.left.traversePostOrder()
        if self.right:
            self.right.traversePostOrder()
        print(self.val, end=' ')


root = Node(1)

root.left = Node(2)
root.right = Node(3)

root.left.left = Node(4)

print("Pre order Traversal: ", end="")
root.traversePreOrder()
print("\nIn order Traversal: ", end="")
root.traverseInOrder()
print("\nPost order Traversal: ", end="")
root.traversePostOrder()
```

Searching for a node

- ➔ **k** is the key you are searching for.
- ➔ **x** is the starting node (root node) from which the search begins.
- ➔ **y** is a temporary pointer used to traverse the tree.

Explanation of the BST Search Pseudocode:

The **BST-Search** algorithm searches for a key `k` in a binary search tree starting from node `x`. Here's how it works, step by step:

BST-Search(x, k)

1: y ← x

`y ← x` means we initialize the pointer `y` to point to the root node (`x`). From here, the search will begin.

2: while y /= nil do

The loop *`while y ≠ nil do`* ensures that the search continues until the node `y` is `nil`, which means either the key has been found or we've reached a leaf node without finding it.

3: if key[y] = k then return y

If key[y] = k, then the key has been found in the current node y, and we return this node (return y).

4: else if key[y] < k then y ← right[y]

If key[y] < k, the current node's key is smaller than the target key (k), so we move to the right child of y. This follows the property of binary search trees, where the right child contains values greater than the current node.

5: else y ← left[y]

If key[y] > k, the current node's key is larger than the target key (k), so we move to the left child of y. In a binary search tree, the left child contains values smaller than the current node

6: return ("NOT FOUND")

If y becomes null, it means we've reached the end of a branch without finding the key k. In this case, the search concludes by returning "NOT FOUND", indicating that the key is not in the tree.

Explanation of the BST Insertion Pseudocode:

BST-Insert(x, z, k)
x is the starting node (usually the root)
z is the new node to insert
k is the key we are inserting into the tree.

1: if x = nil then return "Error"
If x is nil (meaning the tree is empty), return an error message.
2: y ← x
Initialize y as the current node (start from root, x).
3: while true do{
Start an infinite loop, which will continue until we find the right place for the new node.

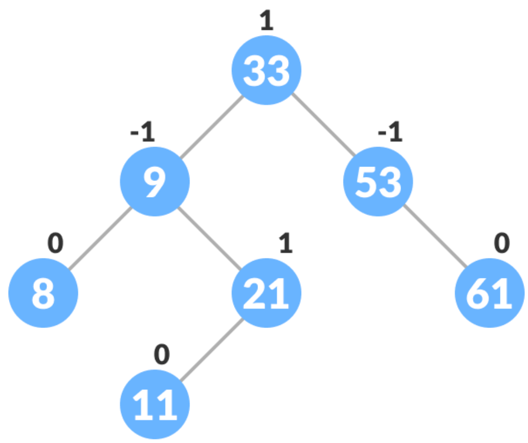
4: if key[y] < k
If the current node's key is smaller than k, move to the right.
5: then z ← left[y]
Set z to be the left child of y.
6: else z ← right[y]
Otherwise, set z to be the right child of y.
7: if z = nil break
If we reach a leaf node (z is nil), break the loop.

8: }
9: if key[y] > k then left[y] ← z
If y's key is greater than k, insert z as the left child of y.
10: else right[p[y]] ← z
Otherwise, insert z as the right child of y.

```
# Insert a node
def insert(node, key):

    # Return a new node if the tree is empty
    if node is None:
        return Node(key)

    # Traverse to the right place and insert the node
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)

    return node
```

AVL Tree Rotations Diagram

- Version of Binary search Tree, Self balancing tree
- Goal: Minimize height of the tree and maintain balance factor at each node
    - Difference must equal -1,0,1 to be balanced



- Balance factor = height |left subtree| - height |right subtree| < 1
    - X > 1 means left is greater than right
- Find node of imbalance and rebalance based off of four cases of imbalance

➔ Four Cases of Imbalance
- Left left insertion
    - Node of imbalance z, three nodes leaning left
- Left Right insertion
    - Node of imbalance z, left then right < (left child node is right heavy)
- Right right insertion
    - Three nodes leaning right
- Right left insertion
    - Node of imbalance z, right then left >

➔ Rebalancing the cases
- LL: use a single right rotation

```
Unbalanced:
      A
     /
    B
   /
  C

def right_rotate(A):
    B = A.left  # B is the left child of A
    T2 = B.right  # T2 is the right subtree of B

    # Perform rotation
    B.right = A
    A.left = T2

    # Update heights (not shown here)
    update_height(A)
    update_height(B)

    return B  # Return the new root of the subtree

Balanced:

    B (balance factor: 0)
   / \
  C   A (balance factor: 0)
```

- LR: use a left rotation on left child node, then right rotation on unbalanced node

Left rotation:

```
  A (balance factor: -2)
     /
    B (balance factor: +1)
     \
      C (balance factor: 0)
```

```
  Right rotation
      A (balance factor: -2)
     /
    C (balance factor: 0)
   /
  B (balance factor: 0)


      C (balance factor: 0)
     / \
    B   A (balance factor: 0)


def left_rotate(A):
    B = A.right  # B is the right child of A
    T2 = B.left  # T2 is the left subtree of B

    # Perform rotation
    B.left = A
    A.right = T2

    # Update heights (not shown here)
    update_height(A)
    update_height(B)

    return B  # Return the new root of the subtree
```

- RR: use a single left rotation

```
Unbalanced:
    A
     \
      B
       \
        C

Balanced:
```

```
      B (balance factor: 0)
     / \
    A   C (balance factor: 0)
```

- RL: use right rotation on right child node, then left rotation on unbalanced node

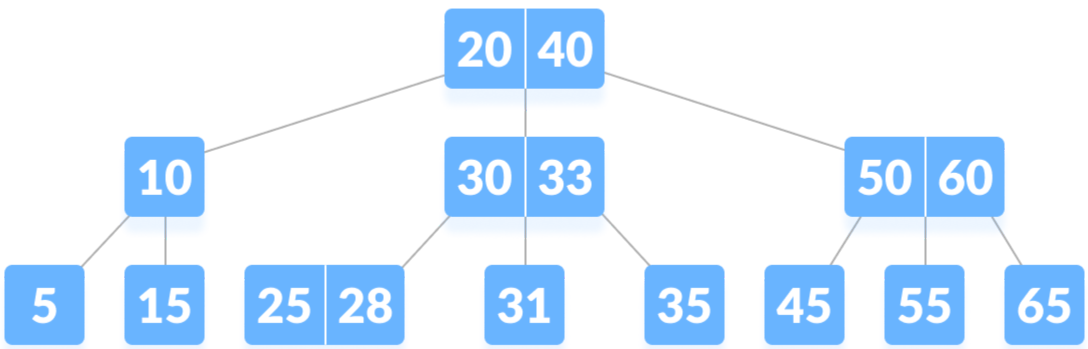
```
  Unbalanced
      A (balance factor: +2)
       \
        B (balance factor: -1)
       /
      C (balance factor: 0)
Right Rotation
      A (balance factor: +2)
       \
        C (balance factor: 0)
         \
          B (balance factor: 0)

Left Rotation
      C (balance factor: 0)
     / \
    A   B (balance factor: 0)

def right_left_rotate(A):
    # Step 1: Right rotation on the right child of A
    A.right = right_rotate(A.right)

    # Step 2: Left rotation on A
    return left_rotate(A)



AVL Node Insertion:
```

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1  # Height of the node (leaf nodes have height
1)

def get_height(node):
    if not node:
        return 0
    return node.height

def get_balance_factor(node):
    if not node:
        return 0
    return get_height(node.left) - get_height(node.right)

def update_height(node):
    if not node:
        return
    node.height = 1 + max(get_height(node.left),
get_height(node.right))

def left_rotate(z):
    y = z.right
    T2 = y.left

    # Perform rotation
    y.left = z
    z.right = T2

    # Update heights
    update_height(z)
    update_height(y)

    return y  # New root

def right_rotate(z):
```

```python
    y = z.left
    T3 = y.right

    # Perform rotation
    y.right = z
    z.left = T3

    # Update heights
    update_height(z)
    update_height(y)

    return y  # New root

def insert(root, key):
    # Step 1: Perform standard BST insertion
    if not root:
        return TreeNode(key)

    if key < root.key:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)

    # Step 2: Update the height of the current node
    update_height(root)

    # Step 3: Check the balance factor and rebalance if necessary
    balance = get_balance_factor(root)

    # Left Heavy (balance factor > 1)
    if balance > 1:
        if key < root.left.key:  # Left-Left Case
            return right_rotate(root)
        else:  # Left-Right Case
            root.left = left_rotate(root.left)
            return right_rotate(root)

    # Right Heavy (balance factor < -1)
    if balance < -1:
```

```
        if key > root.right.key:  # Right-Right Case
            return left_rotate(root)
        else:  # Right-Left Case
            root.right = right_rotate(root.right)
            return left_rotate(root)

    return root  # Return the (possibly new) root of the subtree
```

B Tree



- Root node has two keys: 20,40
  - Root node has three children - left child contains keys less than 10, middle child contains keys between 20 and 40, right child contains keys greater than 40
- Binary Trees can store many keys in a single node and have multiple child nodes -> decreases height to allow for faster disk access

B Plus Trees

- Minimize disk access, faster than a disk read,
- M = Maximum # of **keys** in each node (3)
- M + 1 = Maximum # of **children** of each node (4)
- Insertion: 04-B+Tree Walkthrough

Difference between B vs B plus Tree

- B Tree:
  - Nodes store keys **and** values
- B Plus Tree:

        - Internal nodes only store keys - no values
        - Leaf nodes store keys and values, and they are all link together in a linked list (more effective for range queries)
        - Disk-based storage (like databases)

Hash Map

- Hash maps store key-value pairs
- Uses **hash function** to map key to an index in an array
    - hash(hello) = 0 -> put value at index 0
- Collision: when two keys map to same bucket
    - Resize hashtable to avoid collisions BUT uses a lot more memory
- Collision handling = chaining (storing multiple key-value pairs in the same bucket using a linked list)