SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING-IoT Including CS & BCT

COURSE NAME :23ITB201-DATA STRUCTURES & ALGORITHMS

II YEAR / III SEMESTER

Unit IV- **TREE ADT**

Topic :Multi- Way Search Trees

The **m-way** search trees are multi-way trees which are generalised versions of binary trees where each node contains multiple elements.

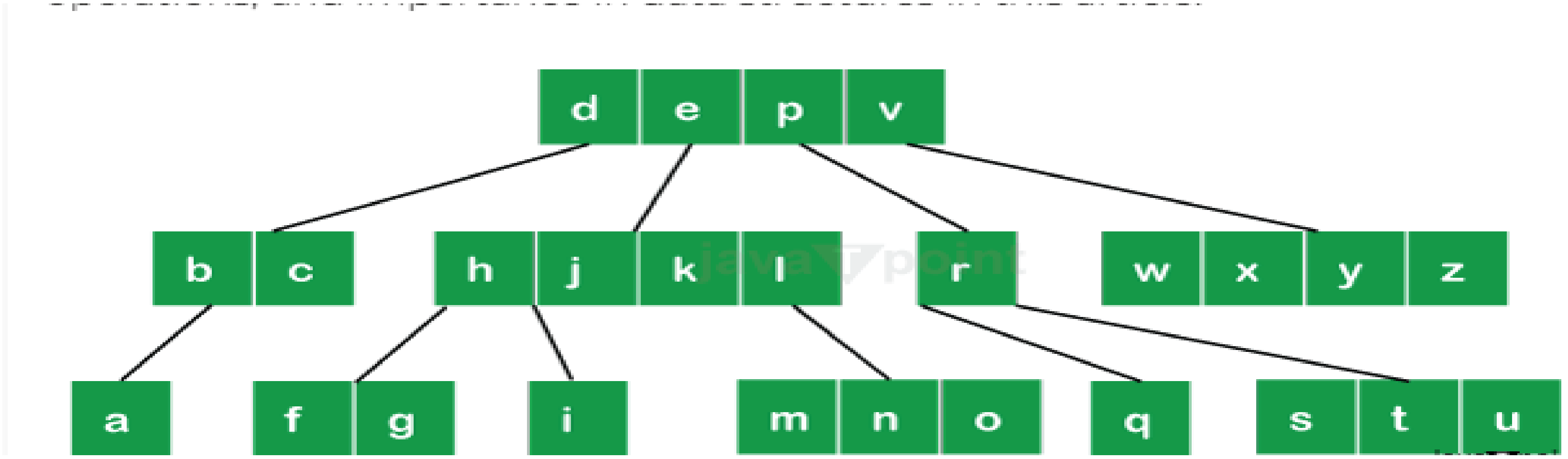
In an m-Way tree of order **m**, each node contains a maximum of **m – 1** elements and m children.

The goal of m-Way search tree of height h calls for O(h) no. of accesses for an insert/delete/retrieval operation.

Hence, it ensures that the height **h** is close to **log_m(n + 1)**.

The number of elements in an m-Way search tree of height h ranges from a minimum of h to a maximum of mh-1

An m-Way search tree of n elements ranges from a minimum height of **log_m(n+1)** to a maximum of **n**



Characteristics of Multiway Trees

A tree data structure called a **multiway tree allows each node to have several offspring.**

Multiway trees can have a **variety of child nodes**, as opposed to binary trees, where each node can only have a maximum of two children.

Multiway trees key characteristics include:

Node Structure: Each node in a multiway tree has numerous pointers pointing to the nodes that are its children. From node to node, the number of pointers may differ.

Root Node: The root node, from which all other nodes can be accessed, is the node that acts as the tree's origin.

Leaf Nodes: Nodes with no children are known as leaf nodes. Leaf nodes in a multiway tree can have any number of children, even zero.

Height and Depth: Multiway trees have height (the maximum depth of the tree) and depth (the distance from the root to a node). The depth and height of the tree might vary substantially due to the different number of children.

Types of Multi-way Search Trees

There are several types of multi-way search trees, each with unique features and use cases. The two common types are

1. 2-3

2. B-tree.

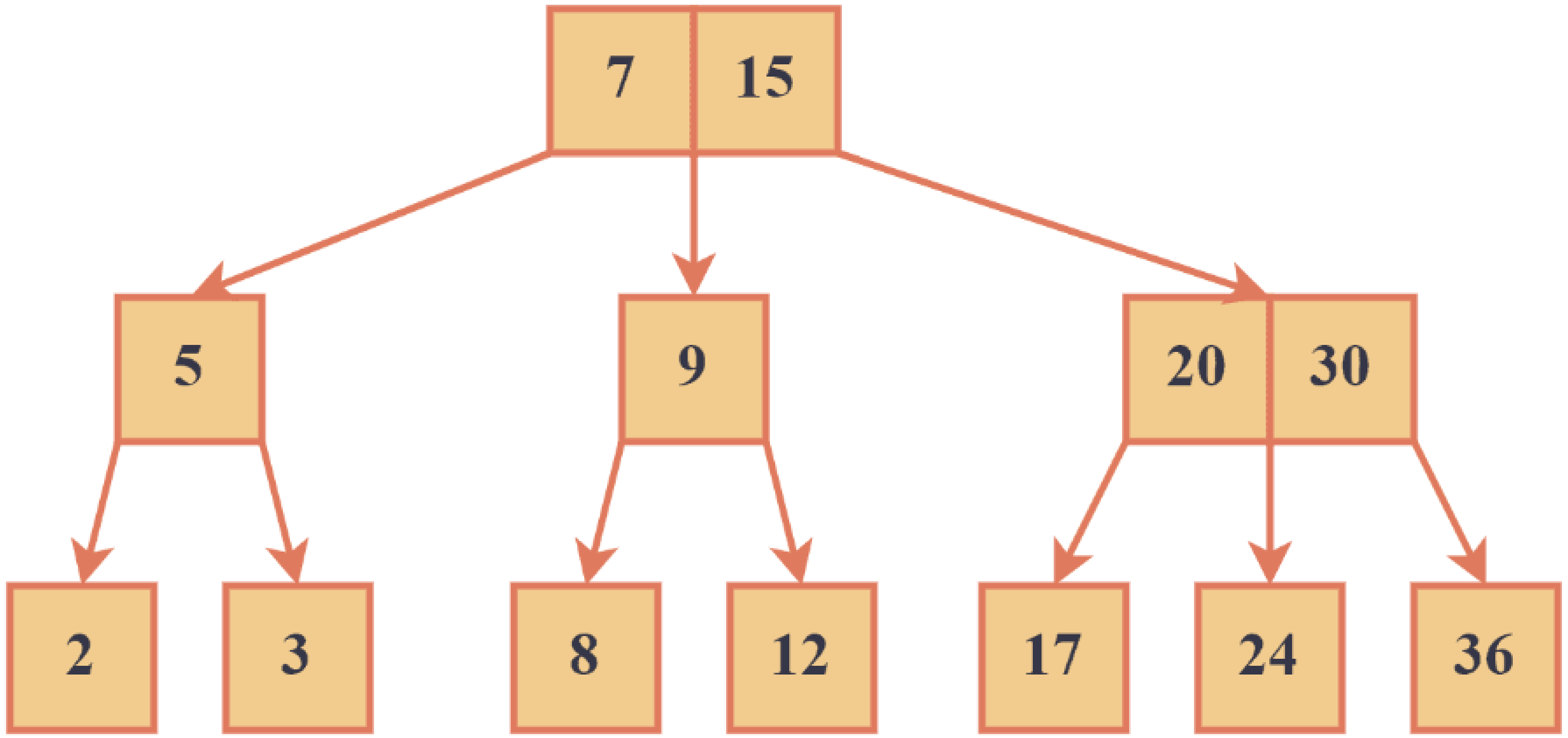
2-3 Tree

2-3 trees are multi-way search trees with two or three children per node.

The nodes in a 2-3 tree are sorted so that the smallest key is always in the leftmost child and the largest key is always in the rightmost child.

They’re often used in applications that require moderate amounts of data, such as compilers and interpreters.

Let’s look at an example of a 2-3 tree:



We can see that each node in the above example is either a 2-node or a 3-node.

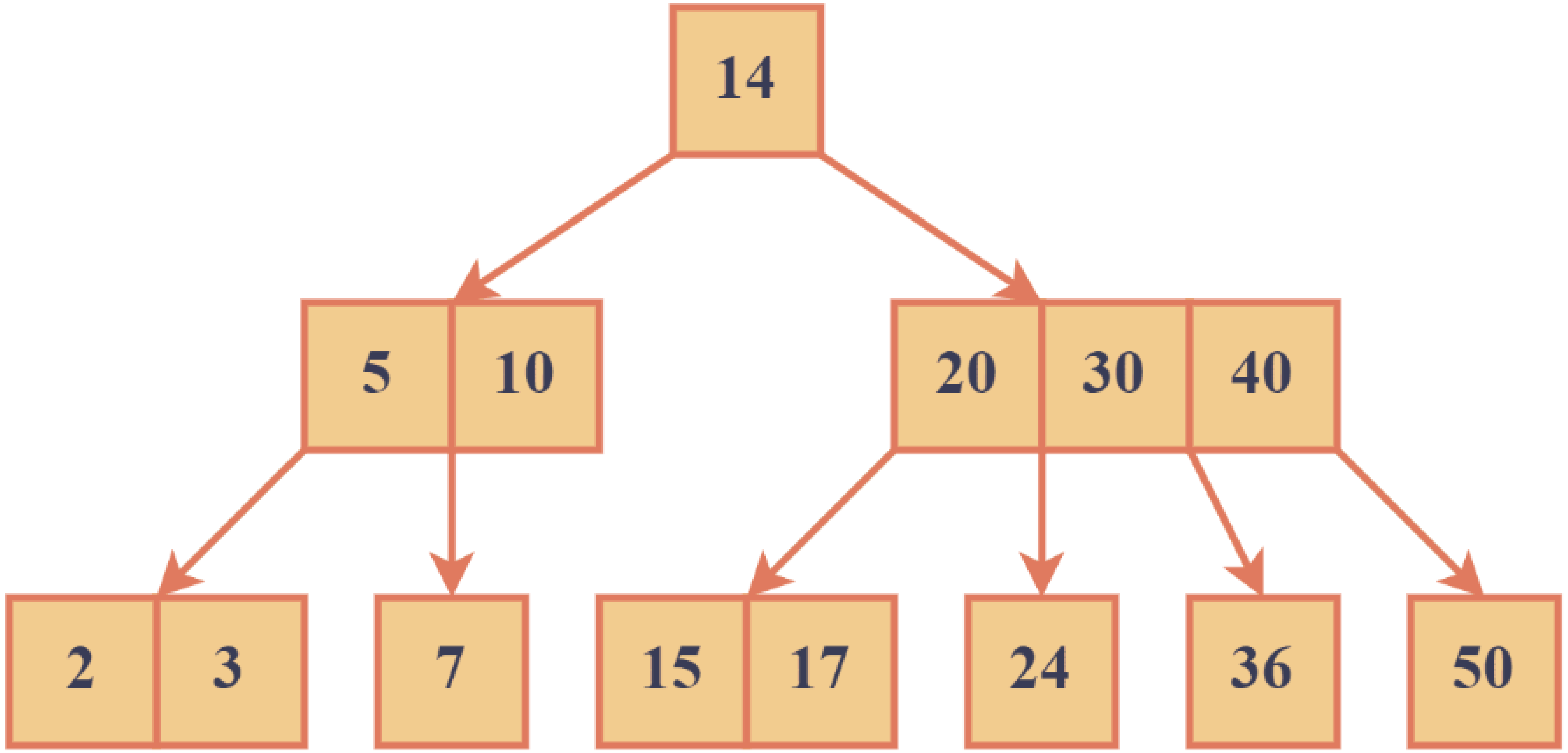
B-Trees

B-trees have a variable number of children per node, typically ranging from 2 to hundreds.

Like 2-3 trees, B-trees are sorted so that keys are ordered from left to right.

2-3 trees are designed to optimize disk reads by ensuring that each node takes up a fixed amount of space on the disk.

This allows for efficient storage and retrieval of large amounts of data and is commonly used in databases and file systems. Let’s look at an example of a B-tree:



In the example above, we can see that node with keys [5,10] has two children, and the node with [20, 30, 40] has four children.

Operations on Multiway Trees:

Multiple operations are supported by multiway trees, allowing for effective data modification and retrieval.

Insertion: Adding a new node to the tree while making sure it preserves the structure and characteristics of the tree.

Deletion: A node is deleted from the tree while still preserving its integrity.

Search: Finding a particular node or value within the tree using search.

Traversal: Traversal is the process of going through every node in the tree in a particular order, such as pre-order, in-order, or post-order.

Balancing (for B-trees): To maintain quick search and retrieval times, balancing (for B-trees) involves making sure the tree maintains its balance after insertions and deletions.

Insertion

To insert a new key into a multi-way search tree, we start at the root node and traverse the tree until we find the appropriate leaf node.

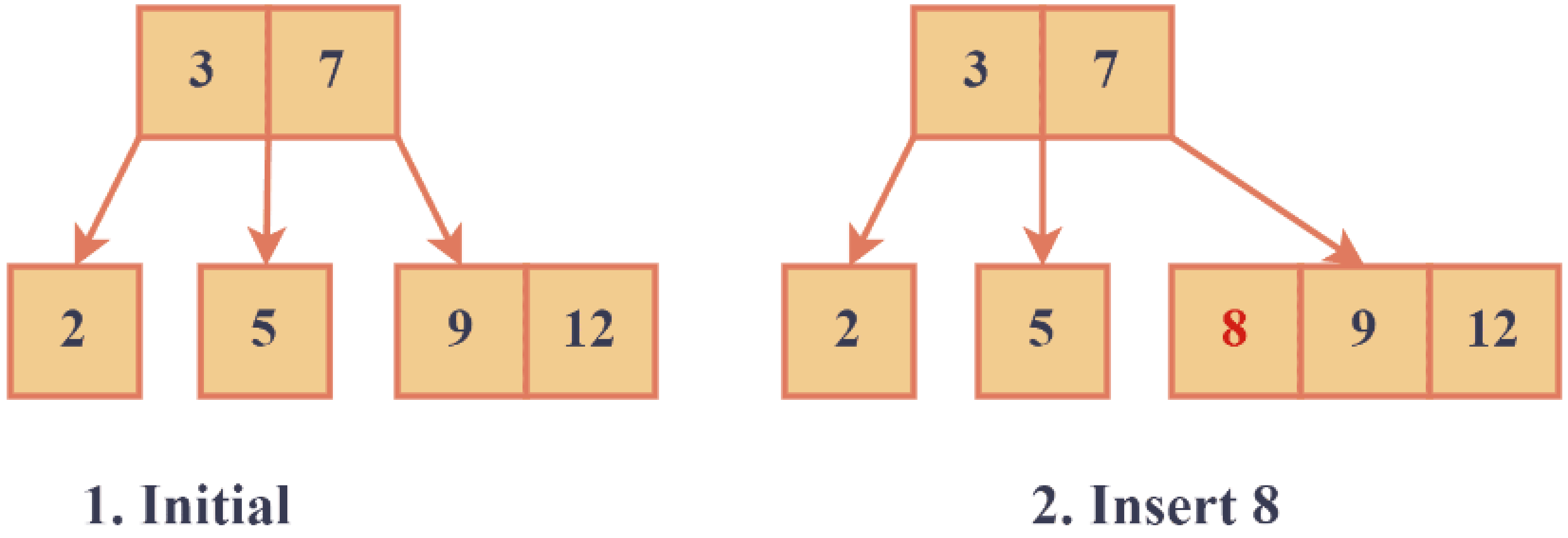
Once we reach the leaf node, we insert the new key into its correct position.

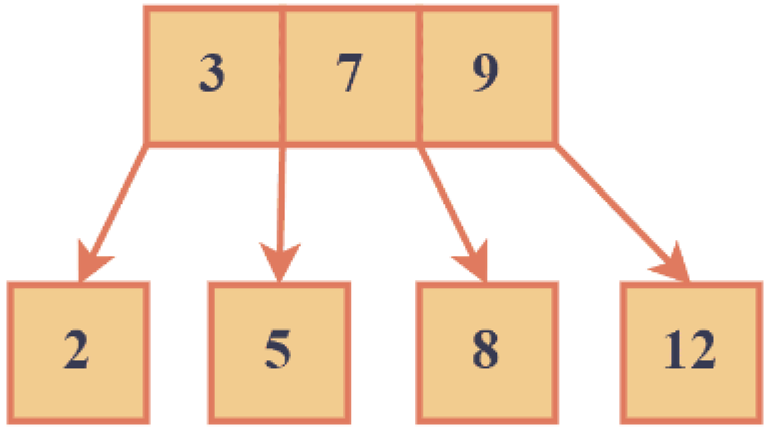
If the node is already full or becomes full after the insertion, we split it into two nodes and insert the median key into the parent node.

We repeat his process recursively until the entire tree is balanced.

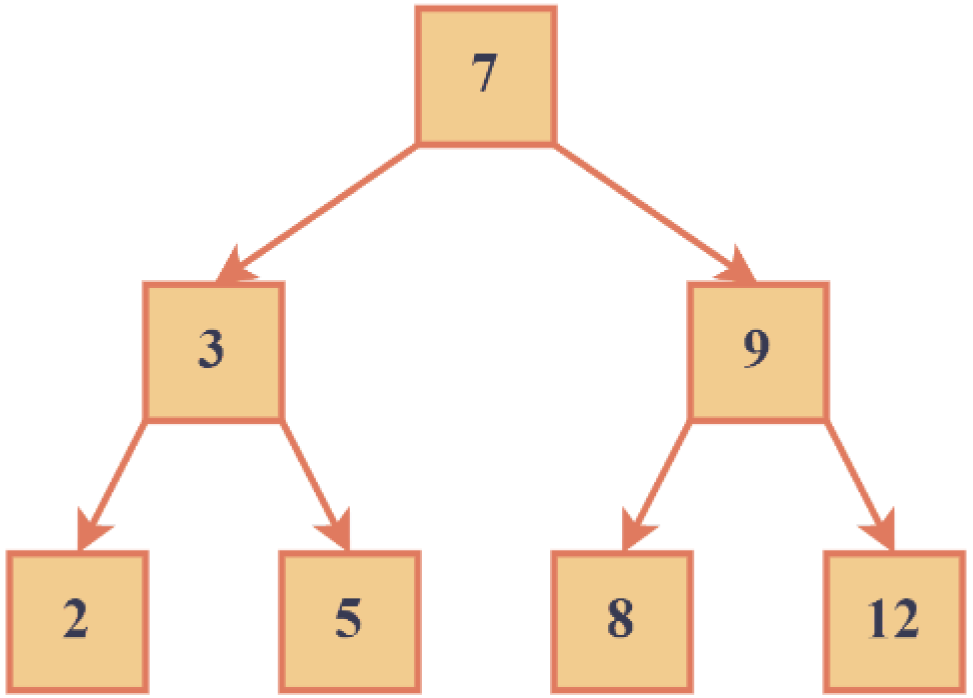
Example

Let’s try to insert number 8 in this tree:





3. Split and Promote



4. Split and Promote

Starting at the root node, we compare 8 to the keys in the node and determine that it should be inserted in the rightmost child.
We insert 8 into it.
We promote the middle key (9) to the parent node ([3 7]) to create a new parent node.
We promote the middle key (7) to create a new parent node.
Now, the tree is balanced and satisfies the properties of a 2-3 tree.

The time complexity of insertion is O(\log N) time, where N is the number of nodes in the tree.

This is due to the balanced structure of the tree, where each node has either 2 or 3 children. This ensures that the height of the tree is at most log base 2 of n.

```
algorithm Insertion23MultiwaySearchTree(T, k):
    // INPUT
    //    T = pointer to the root of the 2-3 tree
    //    k = the key to be inserted
    // OUTPUT
    //    Updated 2-3 tree T with k inserted

    Find the appropriate leaf node N for key k by traversing the
tree from the root

    if N has less than 2 keys:
        Insert k into N
    else:
        Split N into two nodes Left and Right with median key m
```

```
if k < m:
        Insert k into Left
    else:
        Insert k into Right

    if N is the root:
        Create new root node R with m as its only key and children Left and Right
    else:
        Promote m to the parent node of N
        Set Left and Right as children of the appropriate keys in the parent node

        if the parent node is now full:
            Recursively split the parent node
```

Searching

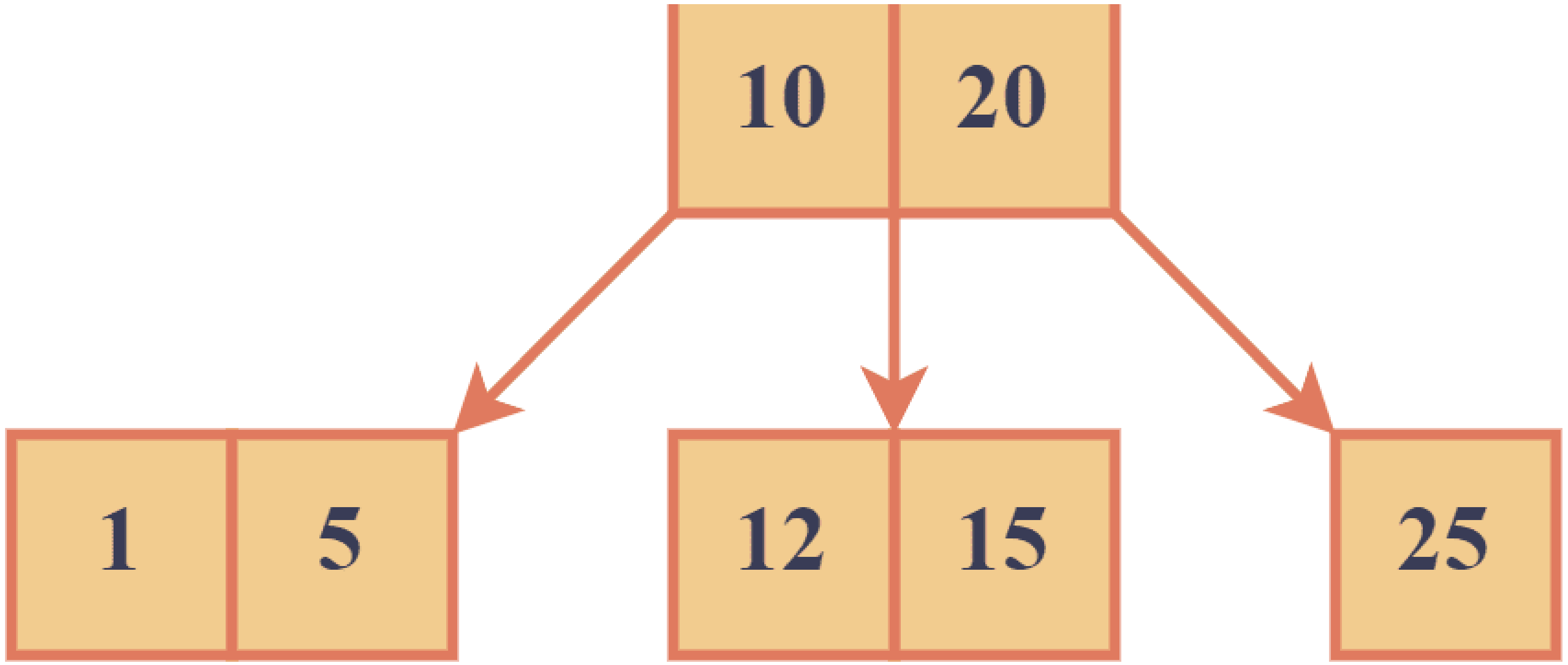
To search for a key in a multi-way search tree, we start at the root node and compare the key to the keys in the current node:

1. If the key matches one of the keys in the current node, we return the corresponding value.
2. If the key is smaller than the smallest key in the current node, we move to the leftmost child.
3. If the key is larger than the largest key in the current node, we move to the rightmost child.
4. If the key is between the smallest and largest, we move to the middle child.

This process is repeated until the key is found or the appropriate leaf node is reached.

Example

For example, let's try to find 11 in this tree:



We begin at the root node to search for the key 11 in the above example.
Since 11 is between 10 and 20, we move to the middle child.
The middle child node contains keys [12, 15], and since 11 is less than 12, we move to the left child of the middle node.

The left child node contains no keys, so we have reached a leaf node, and the key 11 is not found in the tree.

The time complexity of search in a multi-way search tree is also O(log N) in the average and worst cases.

This is because the tree is balanced, and each node can have at most 3 children, reducing the search space by a factor of 3 at each level.

Therefore, the search operation takes logarithmic time in the worst case.

```
algorithm SearchIn23Tree(T, k):
    // INPUT
    //    T = a 2-3 multi-way search tree
    //    k = a key to search for
    // OUTPUT
    //    The value associated with k if it exists in T, otherwise null

    currentNode <- the root node of T

    while currentNode is not a leaf node:
        if k is equal to the first key in currentNode:
            return the value associated with the first key

        else if k is equal to the second key in currentNode:
```

```
return the value associated with the second key

    else if k is less than the first key in currentNode:
        currentNode <- left child of currentNode

    else if k is greater than the second key in currentNode:
        currentNode <- rightmost child of currentNode

    else:
        currentNode <- middle child of currentNode

if k is equal to the first key in currentNode:
    return the value associated with the first key
else:
    return null
```

Advantages of Multi-Way Search Trees

Multi-way search trees have several advantages over binary search trees, including.

Compared to binary search trees, **they require fewer internal nodes to store items**.

Due to their fixed height, balanced multi-way search trees can be efficiently stored on disk and accessed quickly.

Finally, multi-way search trees are generally easier to implement than other advanced data structures like AVL and red-black trees, which have stricter balancing requirements.

Any Query????

Thank you……