CMC UNIVERSITY

CHAPTER 4. TREES

CMC UNIVERSITY

Outline

- ✓ Basic concepts
- ✓ Tree traversal
- ✓ Binary trees
- ✓ Binary search trees
- ✓ Balanced binary trees

Basic concepts

- A tree is a *non-linear data structure*, as there is a parent-child relationship between the items. The top of the tree's data structure is known as a root node. This is the ancestor of all other nodes in the tree.
- There should not be any cycle among the nodes in trees. The tree structure has nodes to form a hierarchy, and *a tree that has no nodes is called an empty tree*.

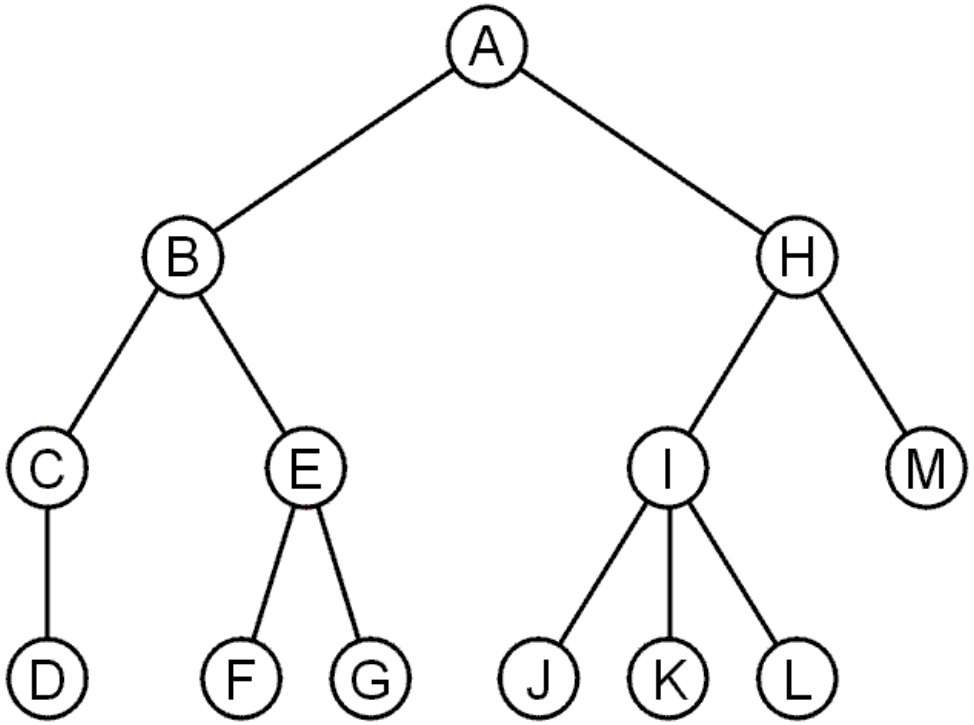


Figure 1: Example tree data structure

Basic concepts

- Node: Each circled letter in the preceding diagram represents a node. A node is any data structure that stores data.

- Root node: The root node is the first node from which all other nodes in the tree descend from. In other words, a root node is a node that does not have a parent node. In every tree, there is always one unique root node. The root node is node A in the above example tree.

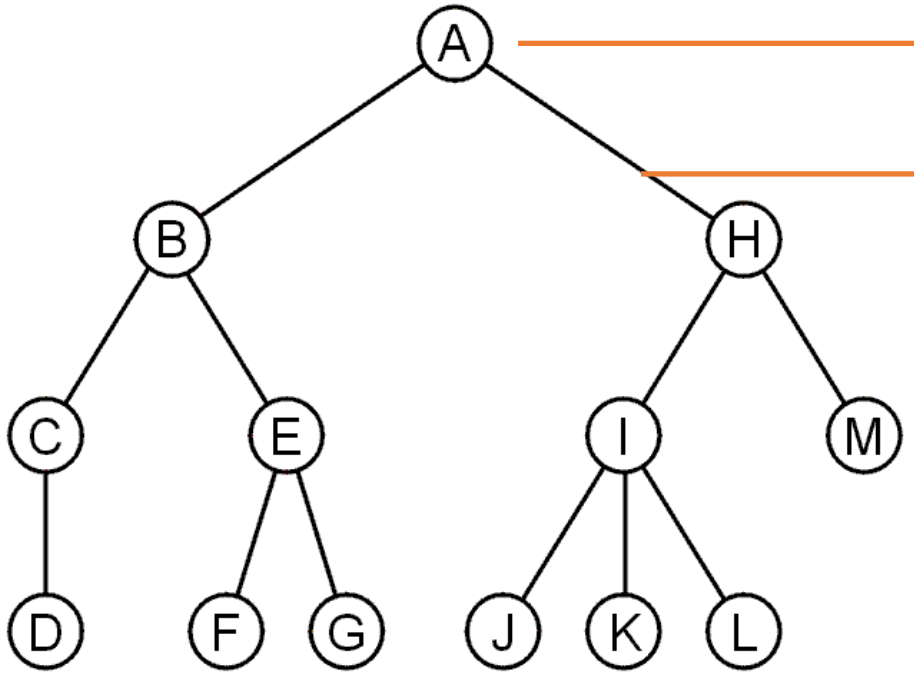


Figure 2: Example tree with node and edge

- Edge: The connection among any given two nodes in the tree is called an edge. The total number of edges in a given tree will be a maximum of one less than the total nodes in the tree.

Basic concepts

• Subtree: A subtree is a tree whose nodes descend from some other tree. For example, nodes H, I, and L form a subtree of the original tree.

• Parent: A node that has a subtree is the parent node of that subtree. For example, node H is the parent of nodes I, and node I is the parent of nodes J, K and L.

• Child: This is a node that is descendant from a parent node. For example, nodes B and H are children of parent node A, while nodes I and M are the children of parent node H.

- All nodes will have zero or more child nodes or children

- For all nodes other than the root node, there is one parent node. Example: H is the parent I

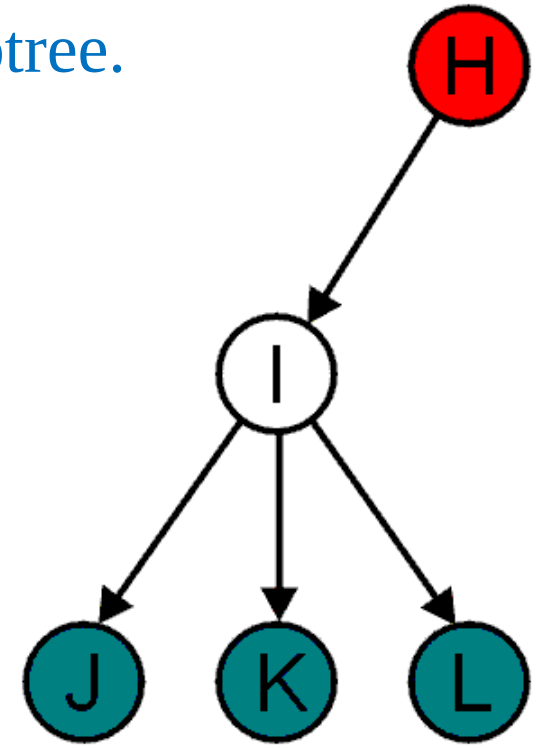


Figure 3. Example of parent, child nodes

CMC UNIVERSITY

Basic concepts

- Degree: The total number of children of a given node is called the degree of the node.

A tree consisting of only one node has a degree of 0.

deg(A) = 2, deg(I) = 3, deg(J) = 0.

- Leaf node: The leaf node does not have any children and is the terminal node of the given tree.

All other nodes are said to be *internal nodes*, that is, they are internal to the tree.

The degree of the leaf node is always 0

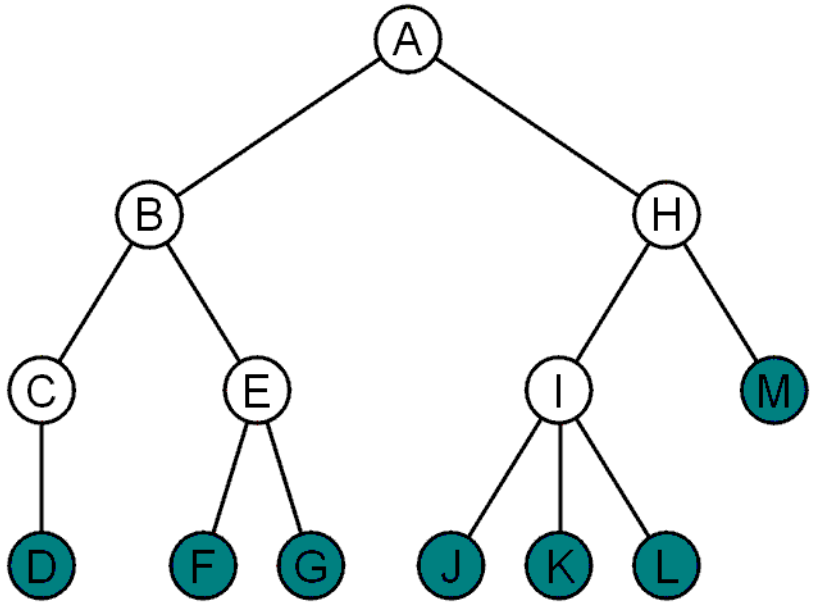


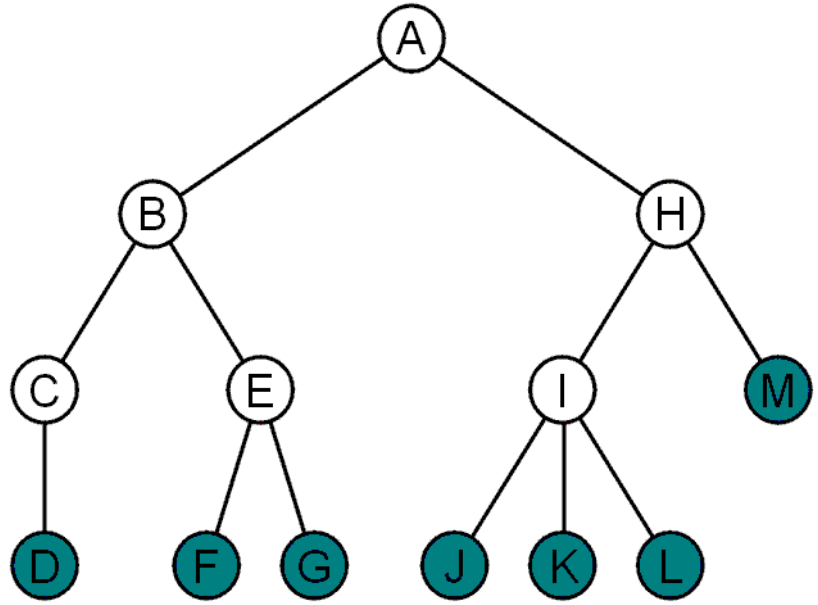
Figure 4: Example tree with leaf nodes

Basic concepts

- Sibling: All nodes with the same parent node are siblings. For example, nodes J, K, L are siblings.
- Level: The root node of the tree is considered to be at level 0. The children of the root node are considered to be at level 1, and the children of the nodes at level 1 are considered to be at level 2, and so on.

Example:

- root node A is at level 0,
- nodes B and H are at level 1,
- nodes C, E, I and M are at level 2.



CMC UNIVERSITY

Basic concepts

- Path: A path is a sequence of nodes $(a_0, a_1, ..., a_n)$

  where a_{k+1} is a child of a_k

The ***length*** of this path is n

Example: the path (B, E, G) has length 2 (Figure 5)

- Height of a tree: The total number of nodes in the longest path of the tree is the height of the tree.

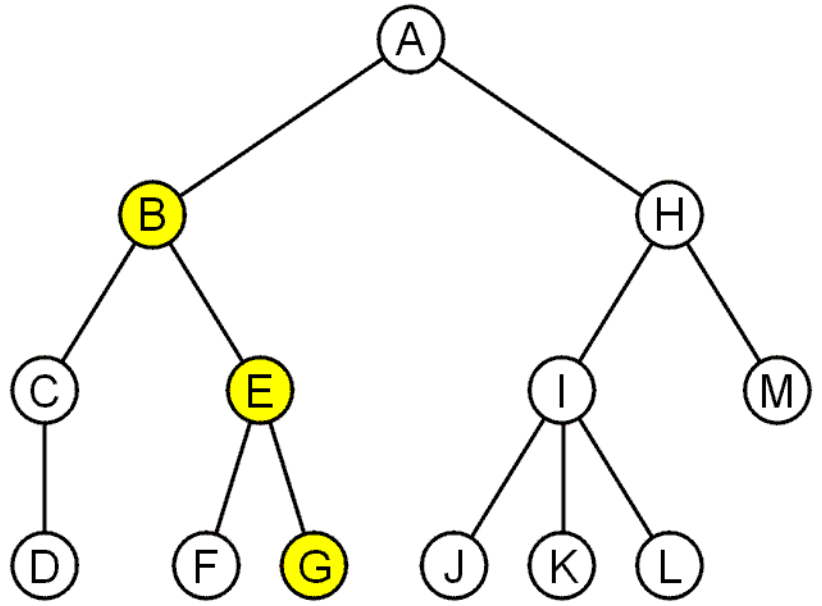


Figure 5: Example of Path

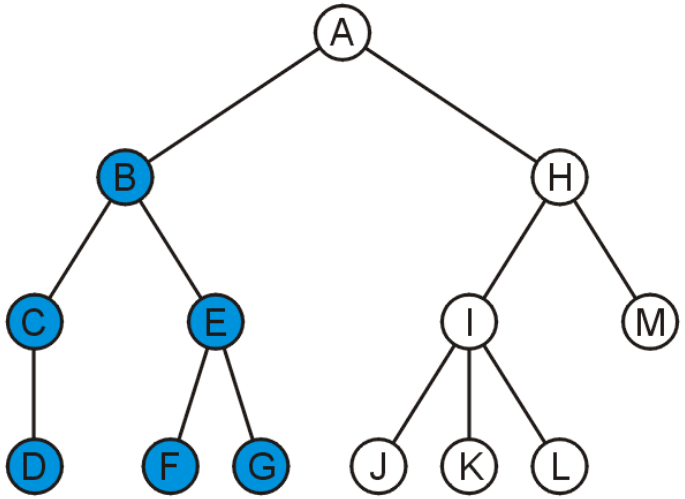
Example: the height of the tree (Figure 5) is 4, as the longest paths, A-B-C-D, A-B-E-F, A-B-E-G, A-H-I-J, A-H-I-K, and A-H-I-L, all have a total number of 4 nodes each.

- Depth: The depth of a node is the number of edges from the root of the tree to that node. In the preceding tree example, the depth of node is the length of path

Example: E has depth 2, L has depth 3 (Figure 5)

CMC UNIVERSITY

Basic concepts



If a path exists from node a to node b:

- a is an ancestor of b
- b is a descendent of a

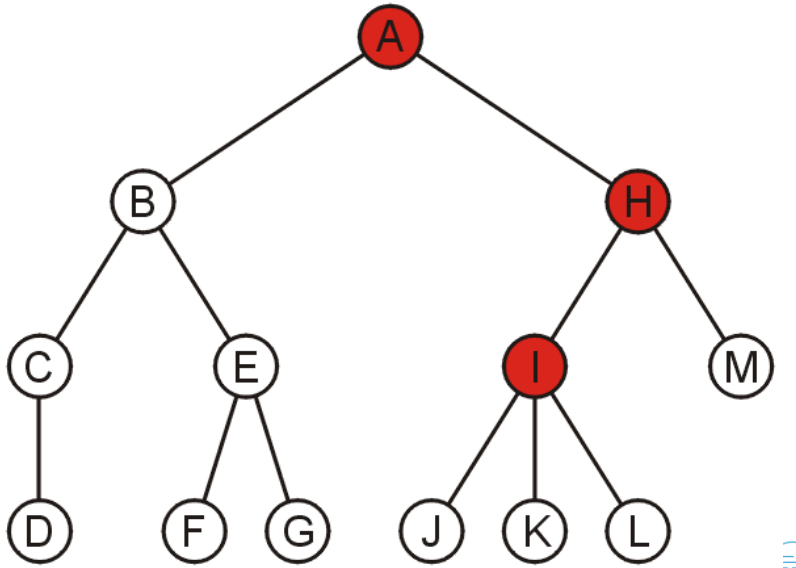
Thus, a node is both an ancestor and a descendant of itself

- We can add the adjective strict to exclude equality: a is a strict descendent of b if a is a descendant of b but $a \neq b$

The root node is an ancestor of all nodes

Example: The descendants of node B: B, C, D, E, F, G

The ancestors of node I: I, H, A



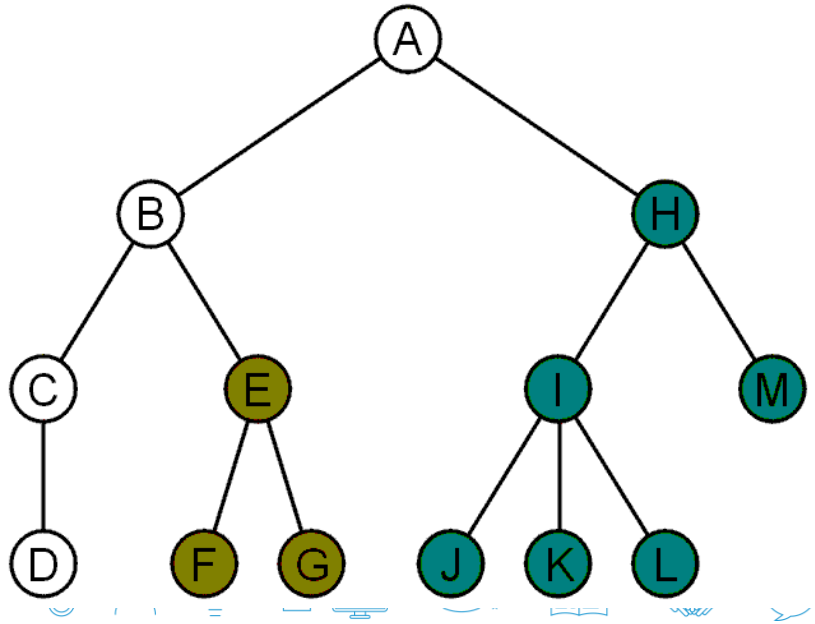
CMC UNIVERSITY

Basic concepts

The tree can be recursively defined as follows:

1. A node is a tree. That node is also the root of the tree.

2. If $T_1, T_2, ..., T_k$ are trees, where $r_1, r_2, ..., r_k$ are the roots respectively, r is a node and there is a parent-child relationship with $r_1, r_2, ..., r_k$, then a new tree T is created with r as the root. The node r is called the parent of $r_1, r_2, ..., r_k$; otherwise, $r_1, r_2, ..., r_k$ are called children of r. The trees $T_1, T_2, ..., T_k$ are called subtrees of r.



Basic concepts

There are many objects with a tree structure.

Example: the table of contents of a book, or a book chapter (Figure 7), arithmetic expressions (Figure 6) are represented as trees

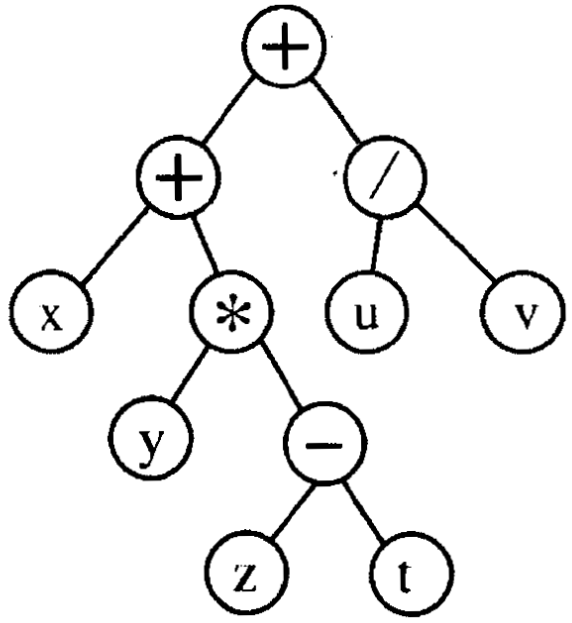


Figure 6. Tree representing arithmetic expression x + y *(z – t) + u/v

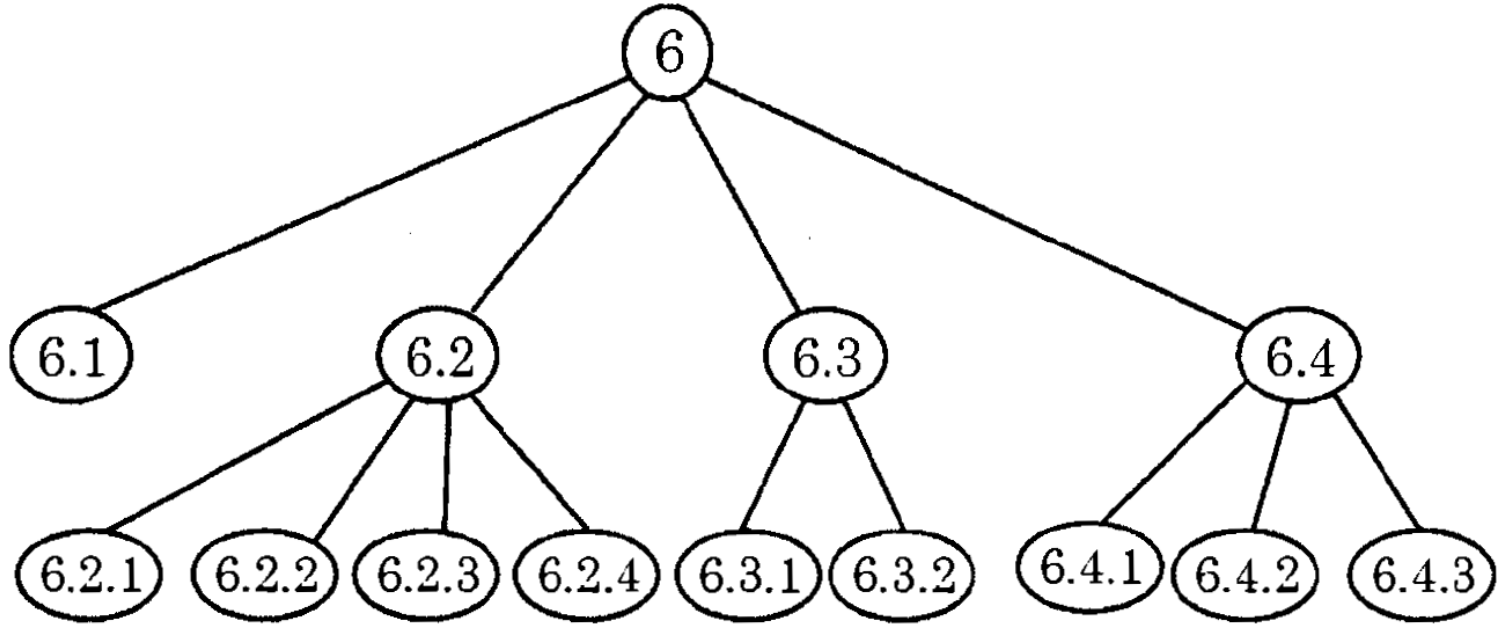


Figure 7. Tree showing the table of contents of a book chapter

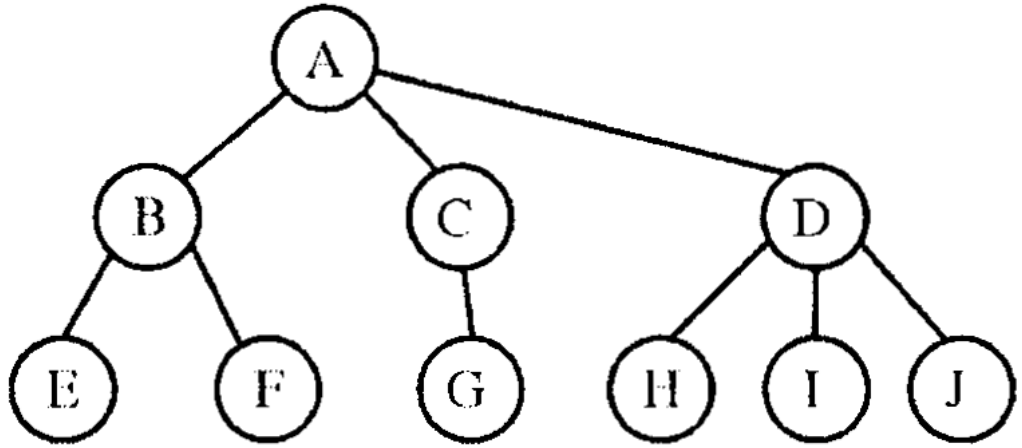
Tree representation

Representing trees with pointers

For any node in the general tree, it has:
- A leftmost child node (first child)
- A sibling next to the right

Assume that the subtrees are numbered from left to right.

Example: For node B, the leftmost child is E, the right sibling is C



Tree representation

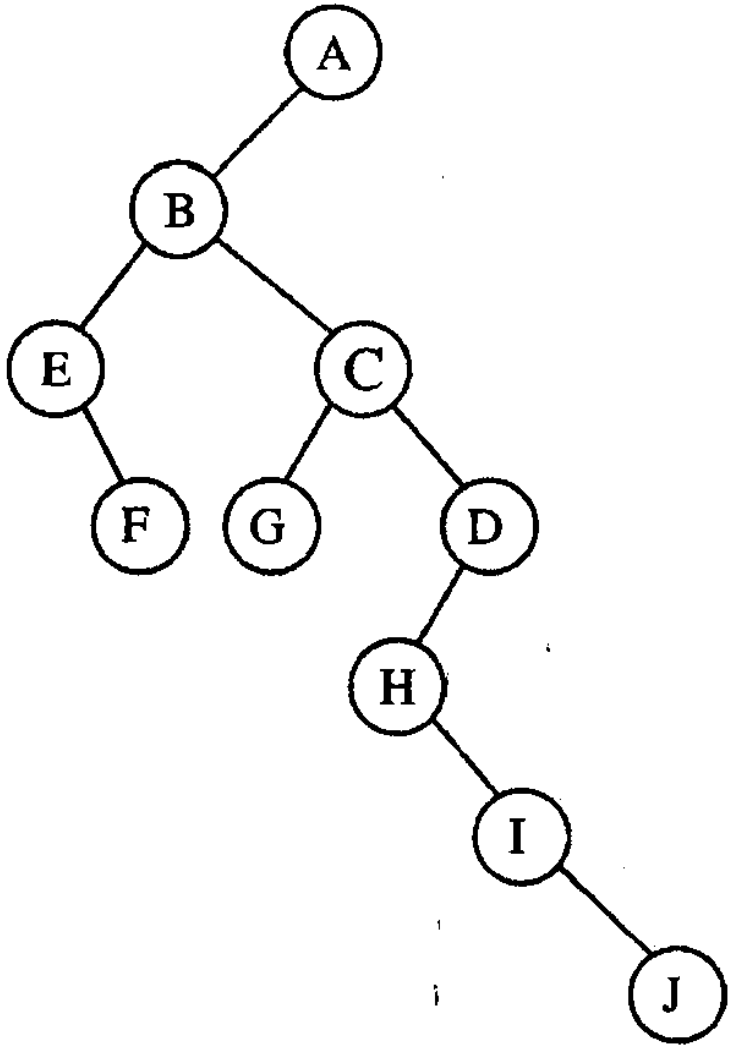
Representing trees with pointers

If each node has a specification:

CHILD	DATA	SIBLING

- CHILD: pointer to the leftmost node
- SIBLING: pointer to the right next sibling node

The above general tree can be represented by the equivalent binary tree as follows:



CMC UNIVERSITY

Tree traversal

Pre-order traversal

Pre-order tree traversal traverses the tree in the order of the root node, browse the subtrees of the root in the pre-order

If T is empty, do nothing

If T is non-empty:

- First, we visit the root node A.
- Traverse the first subtree T1 of the root T in the pre-order
- Browse the remaining subtrees T2, T3, …, Tk of the root T in the pre-order

CMC UNIVERSITY

Tree traversal

In-order traversal

If T is empty, do nothing

If T is non-empty:

- Traverse the first subtree T1 of the root of T in-order
- Visit the root node
- Traverse the remaining subtrees T2, T3, …, Tk of the root T in the in-order

CMC UNIVERSITY

Tree traversal

Post-order traversal

If T is empty, do nothing

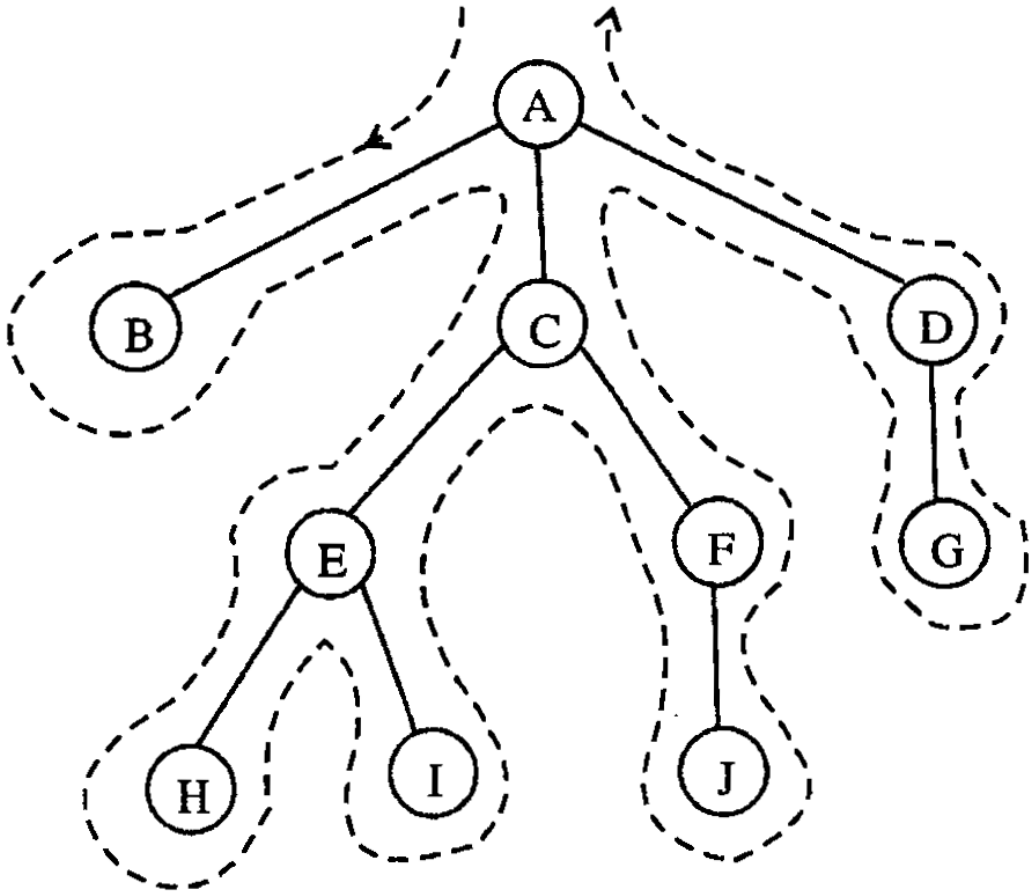
If T is non-empty:

- Traverse the first subtree T1 of the root of T post-order
- Traverse the remaining subtrees T2, T3, …, Tk of the root T in the post-order
- Visit the root node

Tree traversal

Example

- Pre-order: A B C E H I F J D G
- In-order: B A H E I C J F G D
- Post-order: B H I E J F C G D A



CMC UNIVERSITY

BINARY TREES

Definition and properties

- A binary tree is a collection of nodes, where the nodes in the tree can have zero, one, or two child nodes. A simple binary tree has a maximum of two children, that is, the ***left child*** and the ***right child*** (Figure 1).

The nodes in the binary tree are organized in the form of the left subtree and right subtree.

Example: a tree of 5 nodes (Figure 2) that has a root node, R, and left subtree, T1, and right subtree, T2.

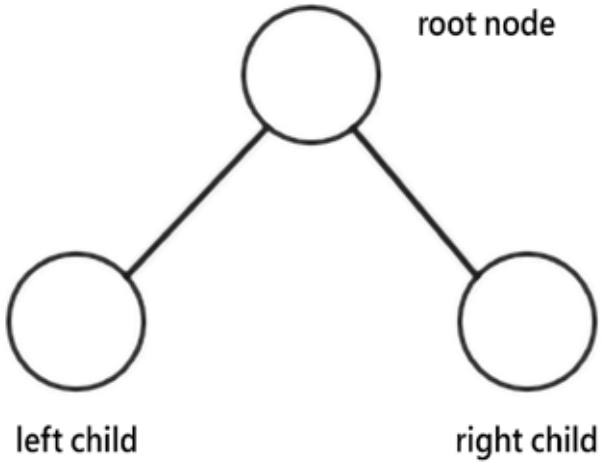


Figure 1. Example of a binary tree

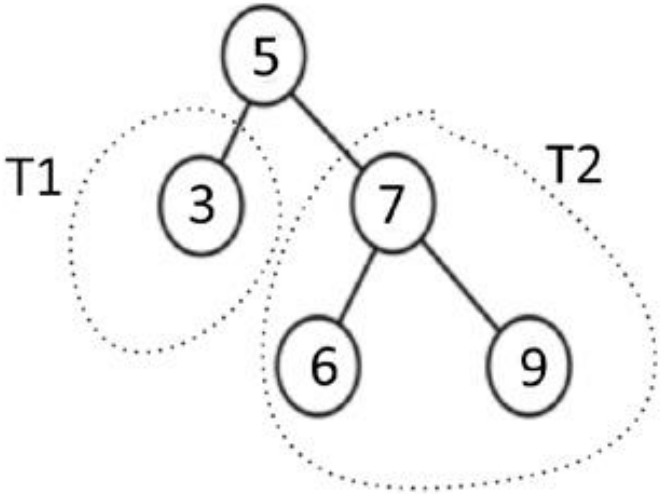


Figure 2. An Example binary tree of 5 nodes

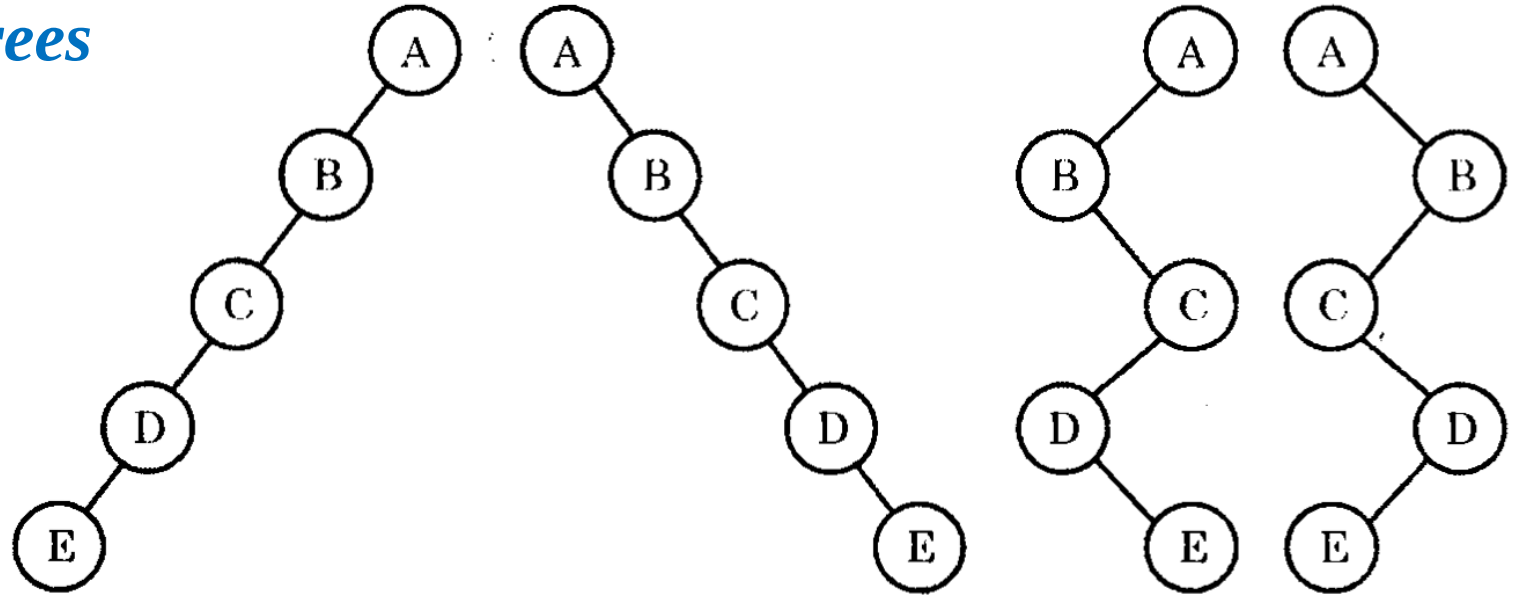
A regular binary tree has no other rules as to how elements are arranged in the tree. It should only satisfy the condition that each node should have a maximum of two children.

Binary search tree

Some special forms of binary trees should be noted:

- Trees a), b), c), d) are called degenerate binary trees, because they are essentially linear lists.

- Tree a) is called a ***left-skewed tree***
- Tree b) is called ***right-skewed tree***
- Trees c), d) are called ***zigzag trees***



a) *b)* *c)* *d)*

Definition and properties

A tree is called a full binary tree if all the nodes of a binary tree have either zero or two children, and if there is no node that has one child.

A perfect binary tree has all the nodes in the binary tree filled, and it doesn't have space vacant for any new nodes; if we add new nodes, they can only be added by increasing the tree's height

A complete binary tree is filled with all possible nodes except with a possible exception at the lowest level of the tree. All nodes are also filled on the left side

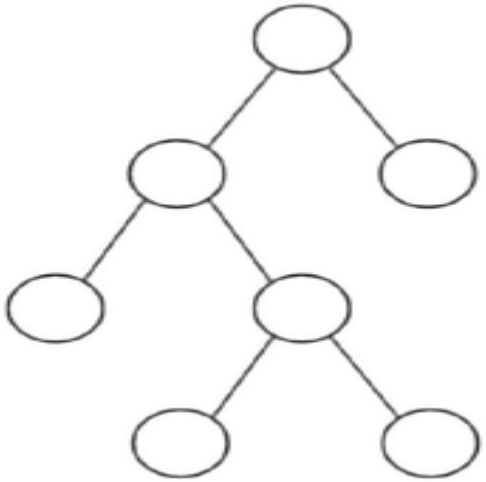


Fig.3. A full binary tree

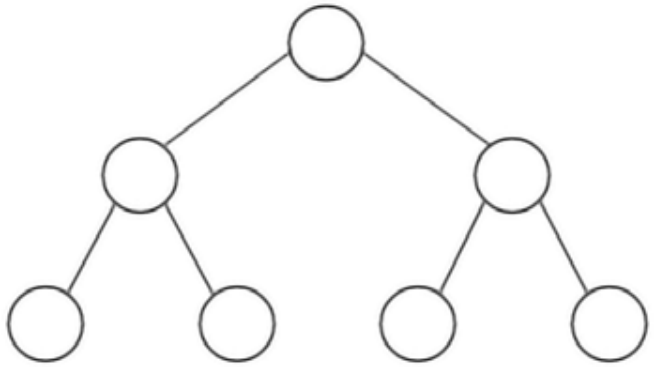


Fig.4. A perfect binary tree

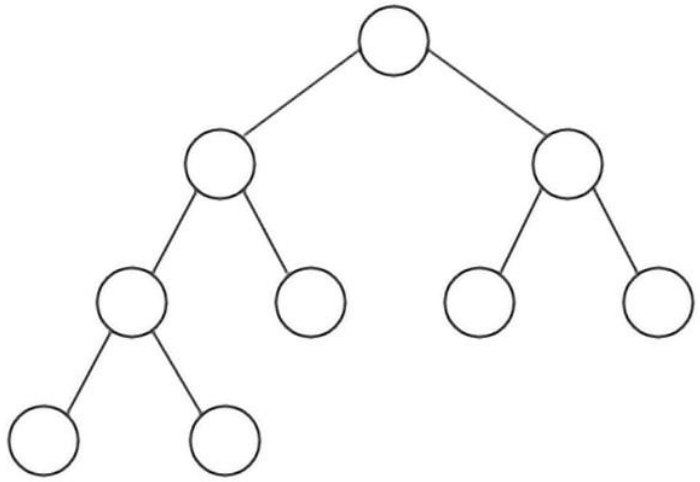


Fig 5. A complete binary tree

Definition and properties

A binary tree can be balanced or unbalanced. In a balanced binary tree, the difference in height of the left and right subtrees for every node in the tree is no more than 1.

An unbalanced binary tree is a binary tree that has a difference of more than 1 between the right subtree and left subtree.

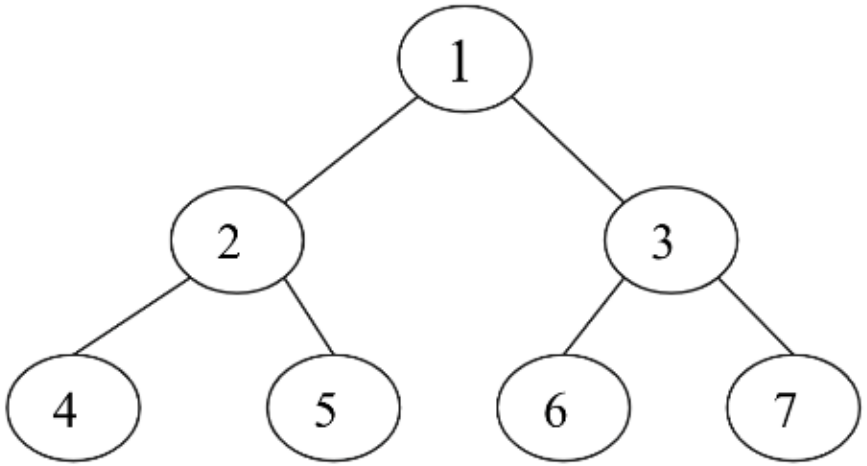


Fig.6. A balanced binary tree

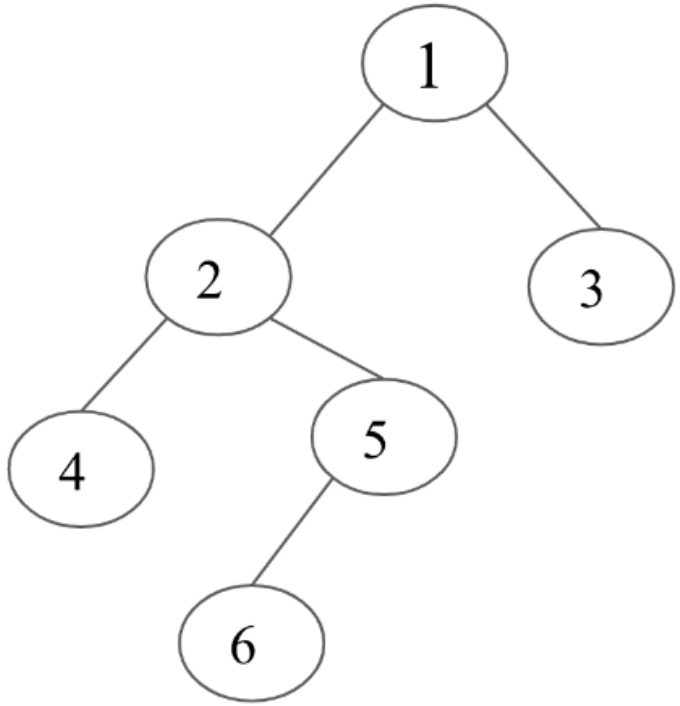


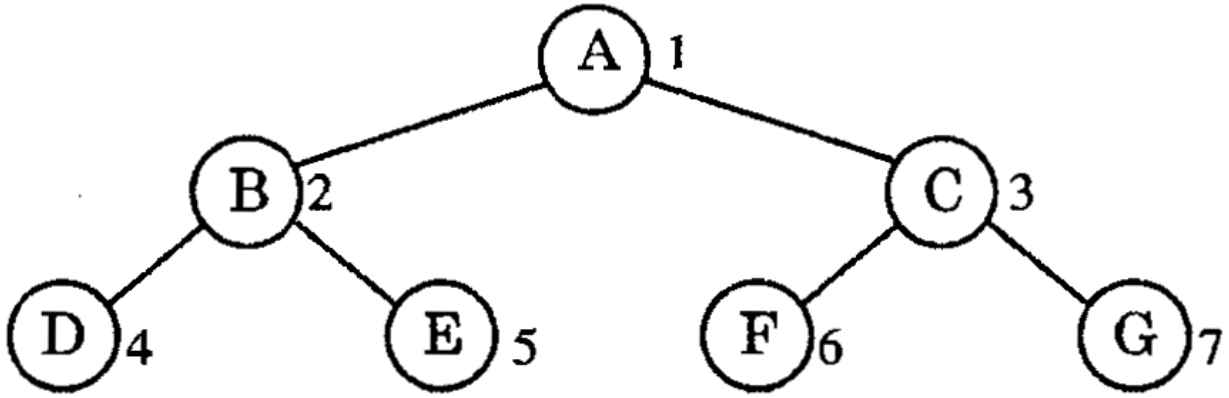
Fig.7. An unbalanced binary tree

Binary tree representation

Representing Binary Trees with array

- If there is a perfect binary tree, we can easily number the nodes in that tree in order from level 1 up, level after level, and from left to right for the nodes at each level

The rule: the child of the i-th node is the 2i and 2i + 1 nodes, or the father of j-th node is $\lfloor j/2 \rfloor$
→ store a perfect binary tree with a vector V according to the principle: the i-th node of the tree is stored at V[i]. For this storage method, if the parent node's address is known, the child node's address can be calculated and vice versa.



A	B	C	D	E	F	G
V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]

Binary tree representation

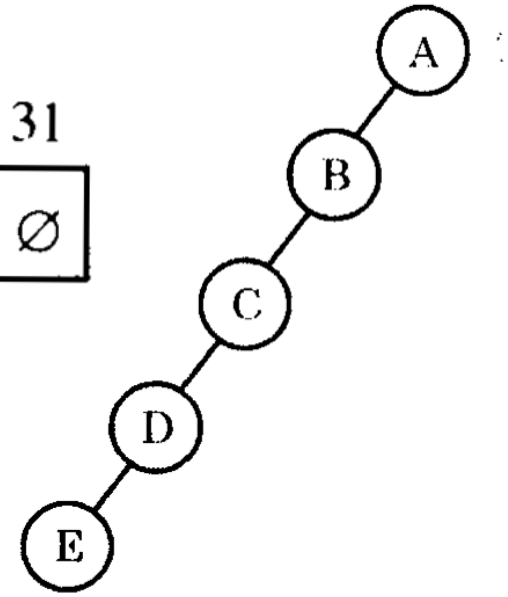
Representing Binary Trees with array

With a complete binary tree, this storage method is still appropriate.

For other types of binary trees, this storage method can be wasteful because many memory elements are left blank (corresponding to an empty subtree).

For example, with the following left-skewed tree, we must store it as a vector of 31 elements where only 5 elements are non-empty.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...	31
A	B	∅	C	∅	∅	∅	D	∅	∅	∅	∅	∅	∅	∅	E	∅	∅	...	∅



In addition, if the tree is always volatile, i.e. the insertions and removals of nodes are frequently performed, then this storage method exposes the disadvantage of arrays.

CMC UNIVERSITY

Binary tree representation

Representing Binary Trees with pointers

In this representation, each node corresponds to an element with the following structure:

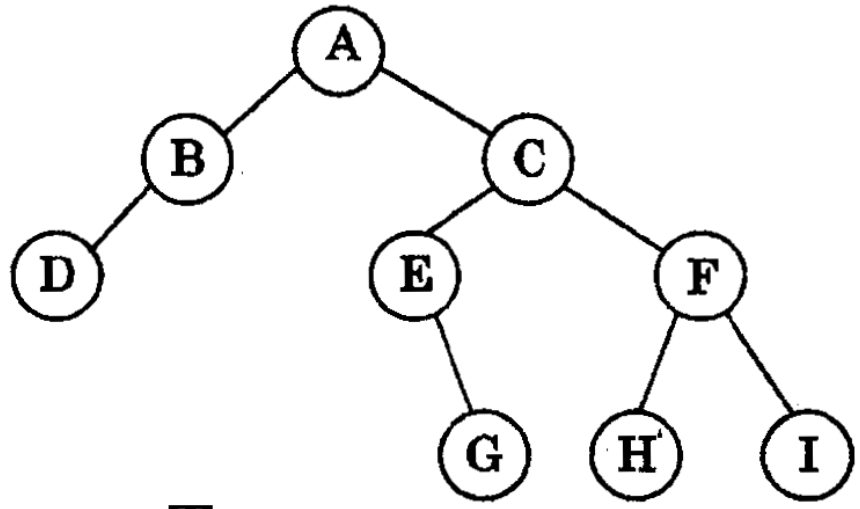
LEFT	DATA	RIGHT

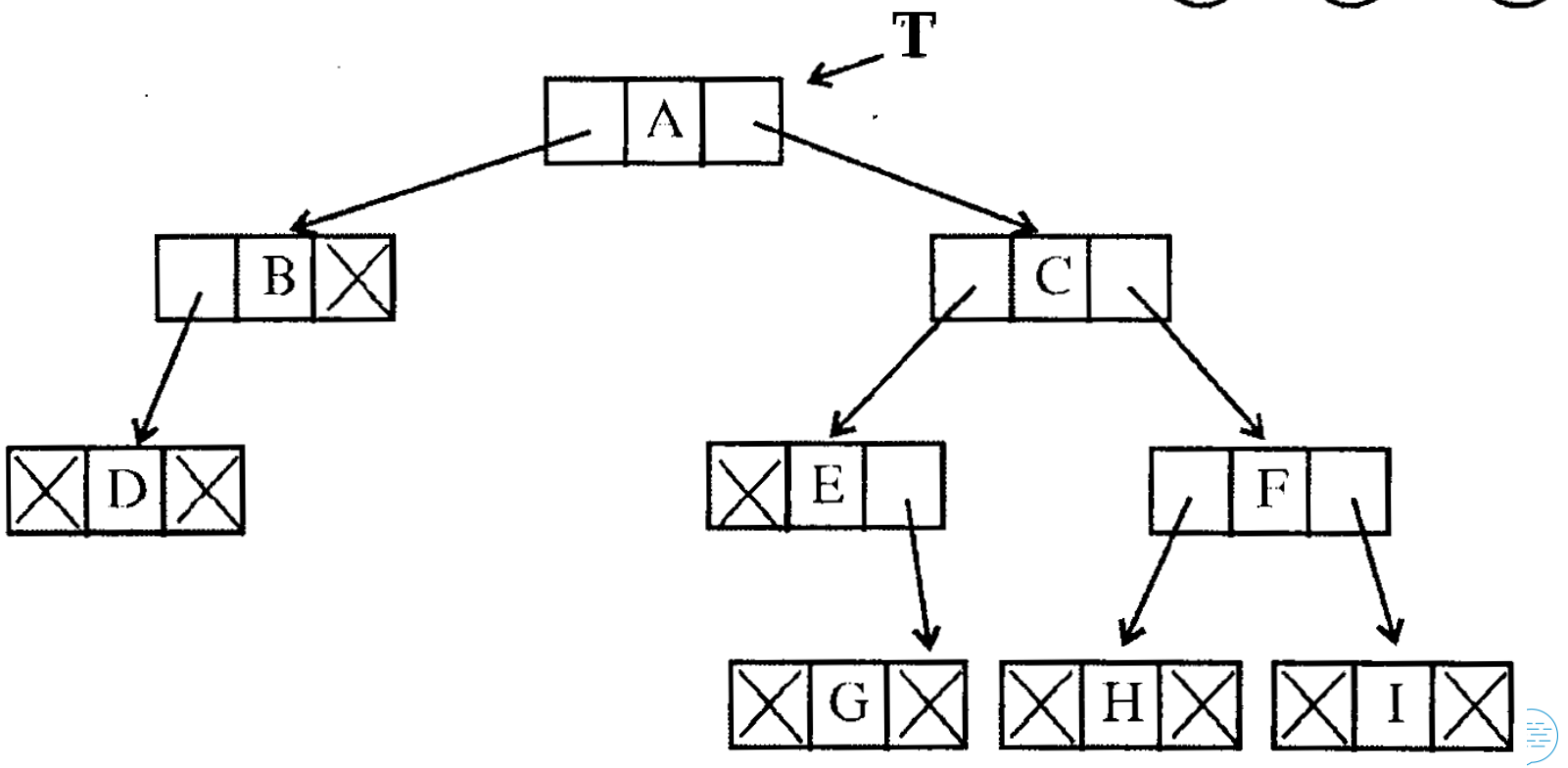
- DATA corresponds to the data of the node
- LEFT corresponds to the pointer, pointing to the left subtree of that node
- RIGHT corresponds to the pointer, pointing to the right subtree of that node

Binary tree representation

Representing Binary Trees with pointers

- In order to access the tree nodes, a pointer T is required, which points to the root node of the tree.
- By convention, if the tree is empty, then T = None
- With this representation, the parent node can directly access the child node, but not vice versa





Implementation of tree nodes

In a binary tree node, each node will contain data items and two references that will point to their left and right children, respectively

```
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None
```

```
n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")
```

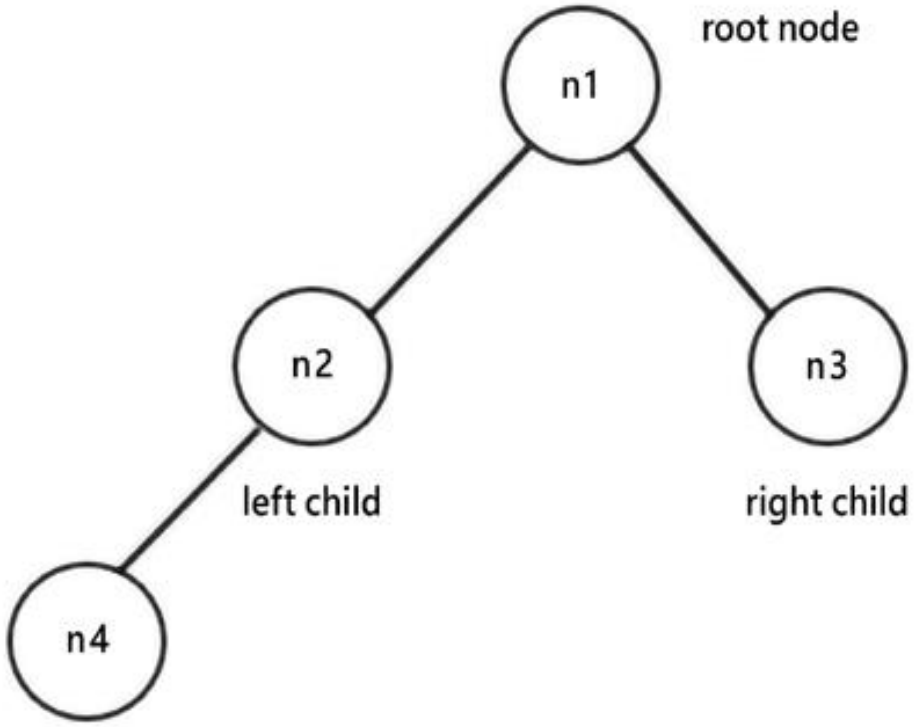


Fig.8. An example binary tree of 4 nodes

Implementation of tree nodes

Next, we connect the nodes to each other according to the previously discussed properties of a binary tree. n_1 will be the root node, with n_2 and n_3 as its children. Furthermore, n_4 will be the left child of n_2.

```
n1.left_child = n2
n1.right_child = n3
n2.left_child = n4
```

CMC UNIVERSITY

Tree traversal

For traversing algorithm, we start from the root node, print out the node, and move down the tree to the next left node. We keep doing this until we have reached the end of the left subtree, like so

```
current = n1
while current:
    print(current.data)
    current = current.left_child
```

Tree traversal

In-order traversal

In-order tree traversal works as follows: we start traversing the left subtree recursively, and once the left subtree is visited, the root node is visited, and then finally the right subtree is visited recursively. It has the following three steps:

- We start traversing the left subtree and call an ordering function recursively
- Next, we visit the root node
- Finally, we traverse the right subtree and call an ordering function recursively

G-D-H-B-E-A-C-F

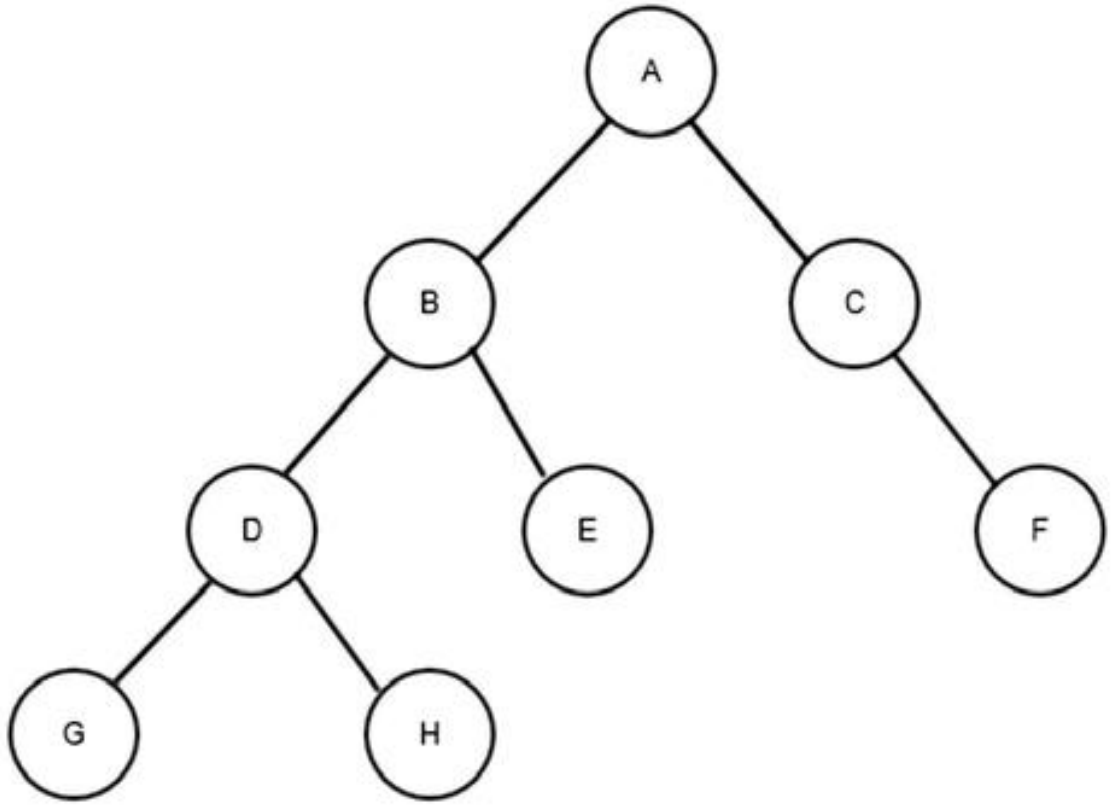


Fig. 9. An example binary tree for in-order tree traversal

CMC UNIVERSITY

Tree traversal

In-order traversal

```
def inorder(root_node):
    current = root_node
    if current is None:
        return
    inorder(current.left_child)
    print(current.data)
    inorder(current.right_child)

inorder(n1)
```

- Firstly, check if the current node is null or empty. If it is not empty, visit the node by printing the visited node. In this case, we first recursively call the inorder function with current.left_child, then we visit the root node, and finally, we recursively call the inorder function with current.right_child.

- Finally, apply the above in-order traversal algorithm on the above sample tree of 4 nodes with n_1 as the root node

Output:
left grandchild node
left child node
root node
right child node

Tree traversal

Pre-order traversal

Pre-order tree traversal traverses the tree in the order of the root node, the left subtree, and then the right subtree. It works as follows:

1. We start traversing with the root node

2. Next, we traverse the left subtree and call an ordering function with the left subtree recursively

3. Next, we visit the right subtree and call an ordering function with the right subtree recursively

A-B-D-G-H-E-C-F

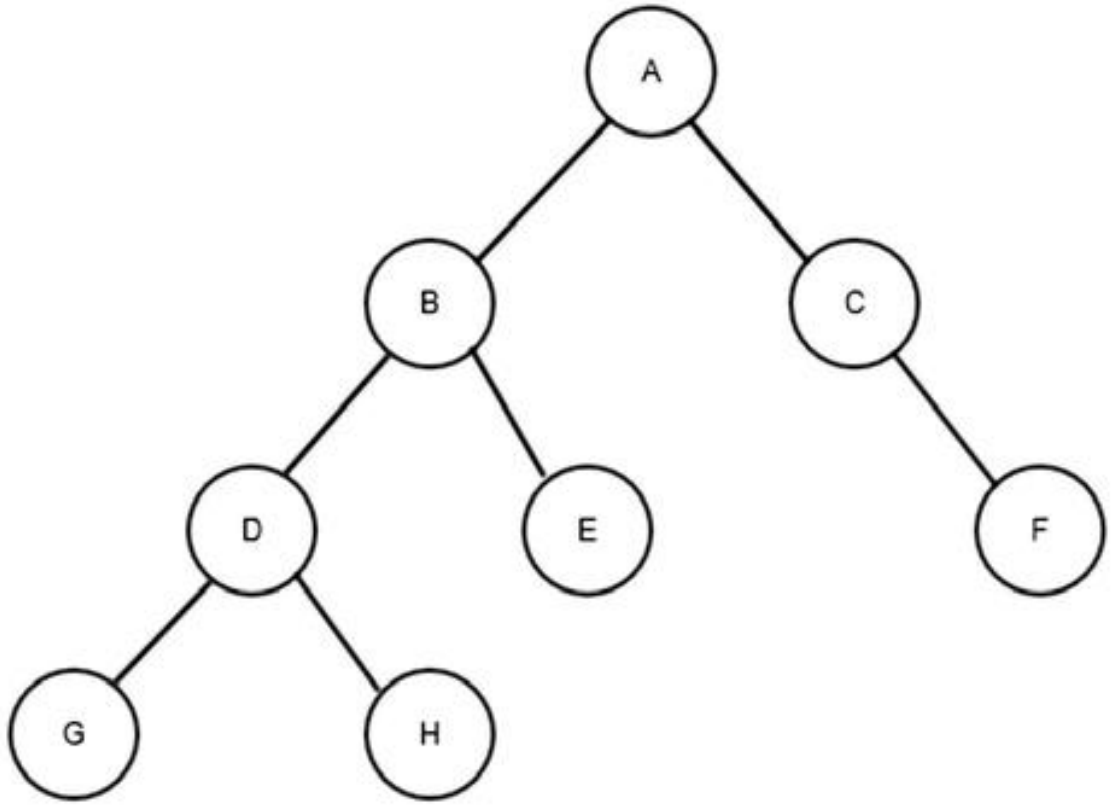


Fig.10. An example binary tree for pre-order tree traversa

CMC UNIVERSITY

Tree traversal

Pre-order traversal

```
def preorder(root_node):
    current = root_node
    if current is None:
        return
    print(current.data)
    preorder(current.left_child)
    preorder(current.right_child)

preorder(n1)
```

First, check if the current node is null or empty. If it is empty, it means the tree is an empty tree, and if the current node is not empty, then we traverse the tree using the pre-order algorithm.

The pre-order traversal algorithm traverses the tree in the order of root, left subtree, and right subtree recursively, as shown in the above code

Output: root node
left child node
left grandchild node
right child node

Tree traversal

Post-order traversal

Post-order tree traversal works as follows:

1. We start traversing the left subtree and call an ordering function recursively

2. Next, we traverse the right subtree and call an ordering function recursively

3. Finally, we visit the root node

So, in a nutshell, for post-order tree traversal, we visit the nodes in the tree in the order of the left subtree, the right subtree, and finally the root node.

G-H-D-E-B-F-C-A

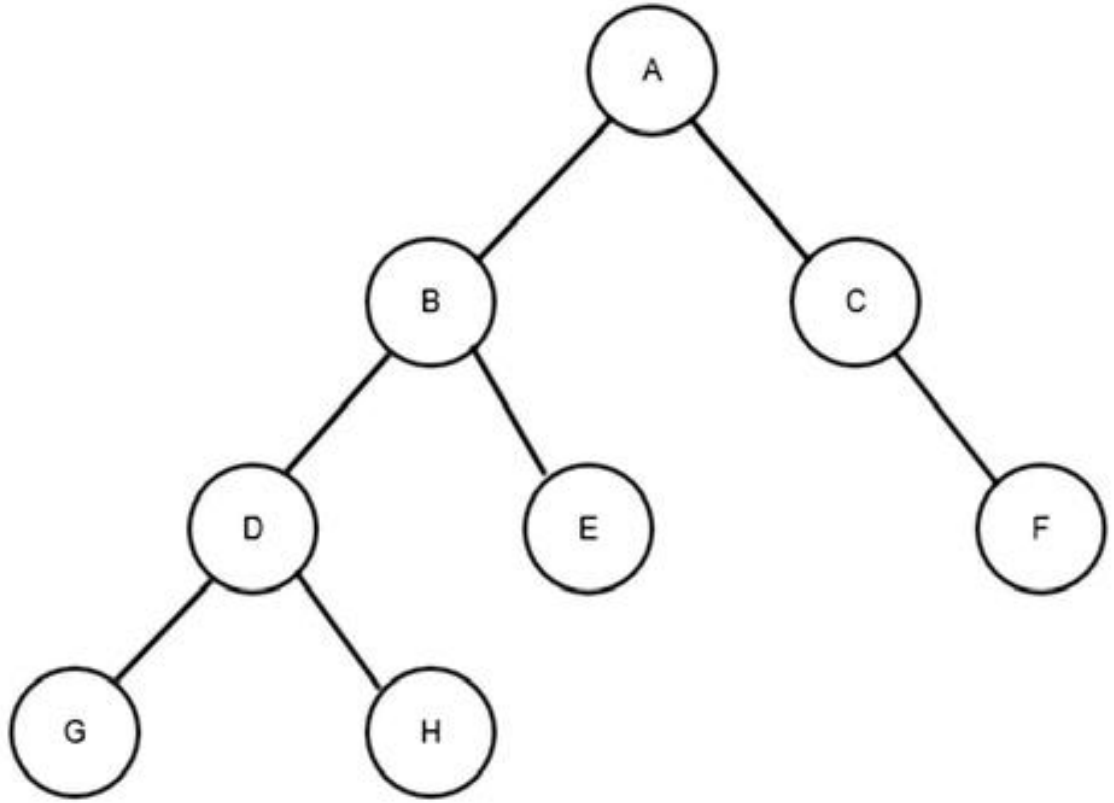


Fig.11. An example binary tree for post-order tree traversal

CMC UNIVERSITY

Tree traversal

Post-order traversal

```
def postorder( root_node):
    current = root_node
    if current is None:
        return
    postorder(current.left_child)
    postorder(current.right_child)
    print(current.data)

postorder(n1)
```

First, check if the current node is null or empty. If it is not empty, we traverse the tree using the post-order algorithm as discussed, and finally, when we apply the above post-order traversal algorithm on the above sample tree of four nodes with n1 as the root node.

Output: left grandchild node
left child node
right child node
root node

CMC UNIVERSITY

Tree traversal

Level-order traversal

We start by visiting the root of the tree before visiting every node on the next level of the tree. Then, we move on to the next level in the tree, and so on.

This kind of tree traversal is how breadth-first traversal in a graph works, as it broadens the tree by traversing all the nodes in a level before going deeper into the tree.

Tree traversal

Level-order traversal

Example: In Figure 12:

- Start by visiting the root node at level 0, with a value of 4 (by printing out its value).

- Next, move to level 1 and visit all the nodes at this level (with the values 2 and 8).

- Finally, move to the next level in the tree, level 3, and we visit all the nodes at this level (1, 3, 5, 10).

Thus, the level-order tree traversal for this tree is : 4, 2, 8, 1, 3, 5, 10

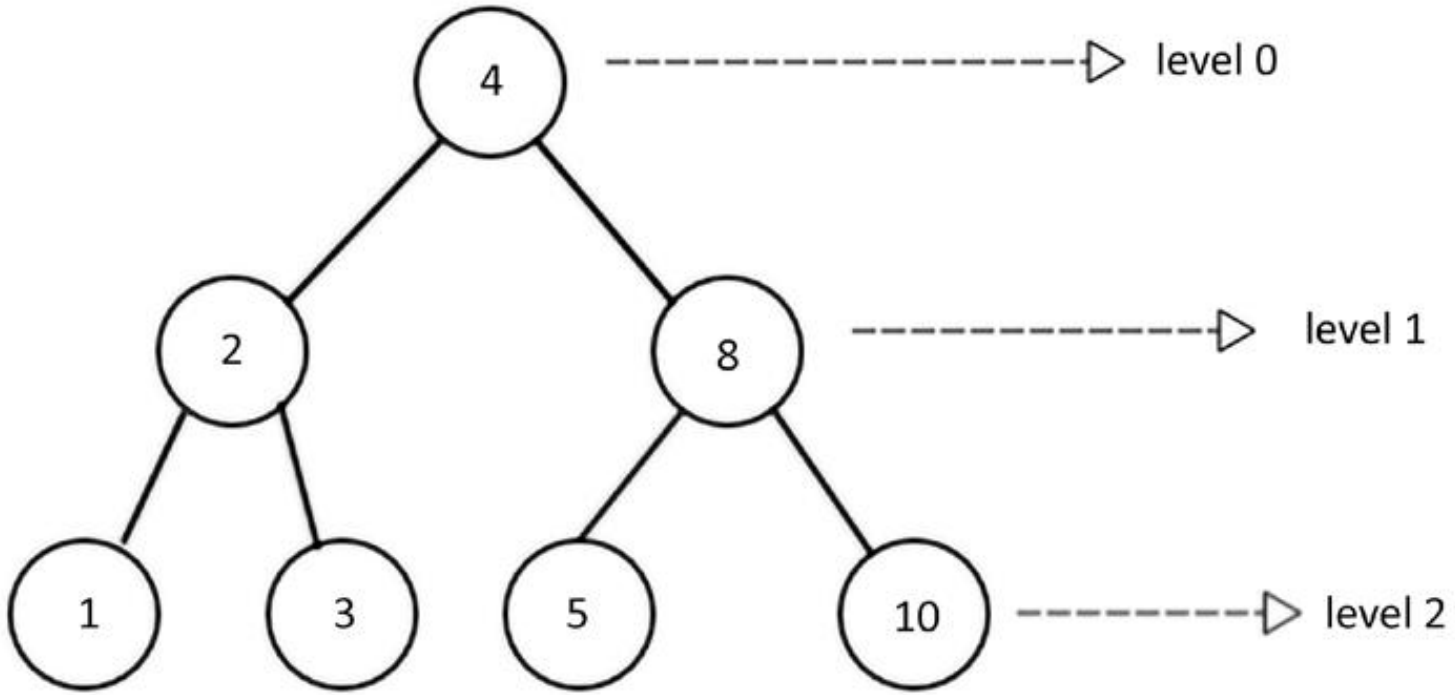


Fig.12. An example binary tree for level-order tree traversal

CMC UNIVERSITY

Tree traversal

Level-order traversal

This level-order tree traversal is implemented using a queue data structure.

- Start by visiting the root node, put it into a queue. The node at the front of the queue is accessed (dequeued), which can then be either printed or stored for later use.

- Next, the left child node is added to the queue, followed by the right node. Thus, when traversing at any given level of the tree, all the data items of that level are firstly inserted in the queue from left to right.

- After that, all the nodes are visited from the queue one by one. This process is repeated for all the levels of the tree

Example: In figure 12

CMC UNIVERSITY

Tree traversal

Level-order traversal

Example: In figure 12

- The traversal of the preceding tree using this algorithm will enqueue root node 4, dequeue it, and visit the node.

- Next, nodes 2 and 8 are enqueued, as they are the left and right nodes at the next level. Node 2 is dequeued so that it can be visited.

- Next, its left and right nodes, nodes 1 and 3, are enqueued. At this point, the node at the front of the queue is node 8. We dequeue and visit node 8, after which we enqueue its left and right nodes.

This process continues until the queue is empty.

CMC UNIVERSITY

Tree traversal

Level-order traversal

```
from collections import deque
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None
```

```
n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")
```

```
n1.left_child = n2
n1.right_child = n3
n2.left_child = n4
```

CMC UNIVERSITY

Tree traversal

Level-order traversal

```
def level_order_traversal(root_node):
    list_of_nodes = []
    traversal_queue = deque([root_node])
    while len(traversal_queue) > 0:
        node = traversal_queue.popleft()
        list_of_nodes.append(node.data)
        if node.left_child:
            traversal_queue.append(node.left_child)
            if node.right_child:
                traversal_queue.append(node.right_child)
    return list_of_nodes


print(level_order_traversal(n1))
```

CMC UNIVERSITY

Tree traversal

Level-order traversal

If the number of elements in traversal_queue is greater than zero, the body of the loop is executed.

The node at the front of the queue is popped off and added to the list_of_nodes list. The first if statement will enqueue the left child node if the node provided with a left node exists. The second if statement does the same for the right child node. Further, the list_of_nodes list is returned in the last statement

The output:

['root node', 'left child node', 'right child node', 'left grandchild node']

CMC UNIVERSITY

Tree traversal

Some important applications of binary trees:

1. Binary trees as expression trees are used in compilers

2. It is also used in Huffman coding in data compression

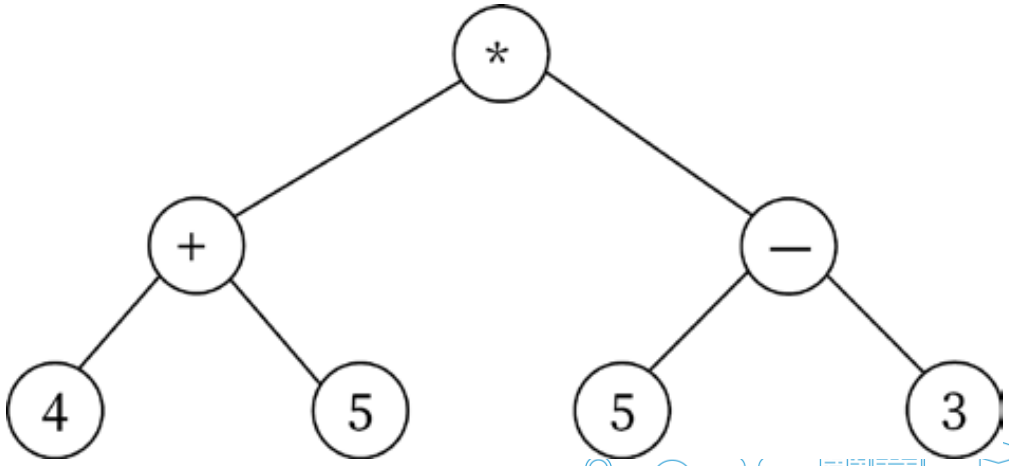
3. Binary search trees are used for efficient searching, insertion, and deletion of a list of items

4. Priority Queue (PQ), which is used for finding and deleting minimum or maximum items in a collection of elements in logarithm time in the worst case

CMC UNIVERSITY

Expression trees

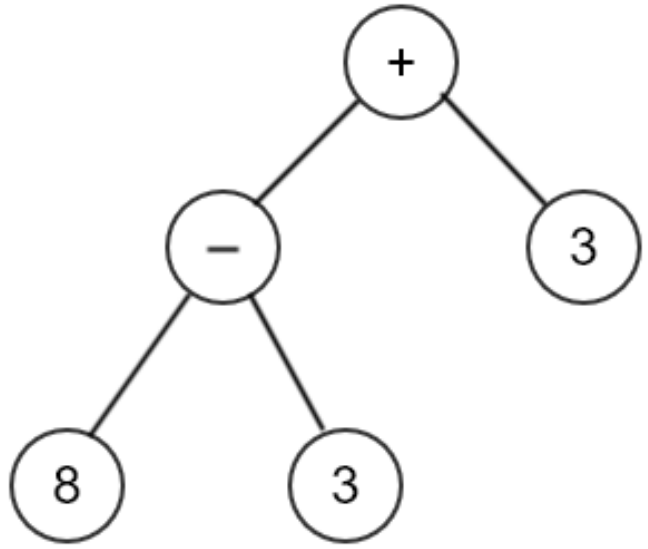
- An arithmetic expression can also be represented using a binary tree, which is also known as an expression tree. The infix notation is a commonly used notation to express arithmetic expressions
- In an expression tree, all the leaf nodes contain operands and non-leaf nodes contain the operators. It is also worth noting that an expression tree will have one of its subtrees (right or left) empty in the case of a unary operator.
- The arithmetic expression is shown using three notations: infix, postfix, or prefix. The in-order traversal of an expression tree produces the infix notation.



CMC UNIVERSITY

Expression trees

- In prefix notation (Polish notation), the operator comes before its operands. For example, (8 - 3) + 3, be represented as + - 8 3 3 in prefix notation.
- The pre-order traversal of an expression tree results in the prefix notation of the arithmetic expression
- Postfix, or reverse Polish notation (RPN), places the operator after its operands, such as. The postfix notation for the preceding expression tree is 8 3 - 3 +



CMC UNIVERSITY

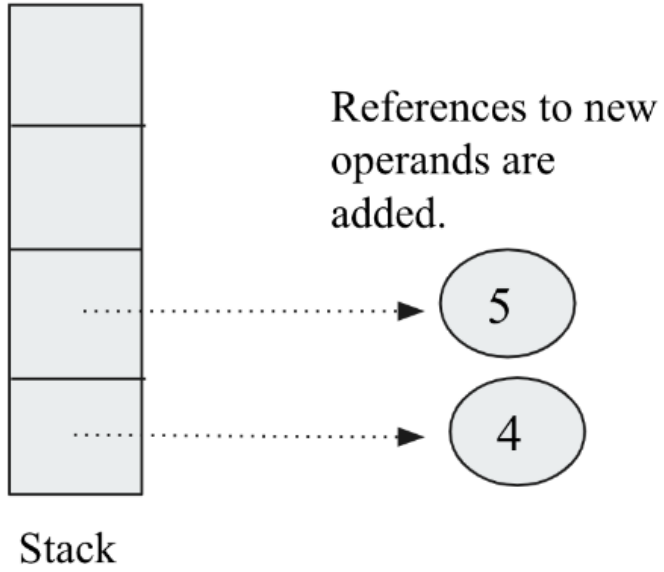
Parsing a reverse Polish expression

- To create an expression tree from the postfix notation, a stack is used:
- If the symbol is an operand, then its references are pushed in to the stack,
- If the symbol is an operator, then we pop two pointers from the stack and form a new subtree, whose root is the operator. The first reference popped from the stack is the right child of the subtree, and the second reference becomes the left child of the subtree.
- Further, a reference to this new subtree is pushed into the stack. In this manner, all the symbols of the postfix notation are processed to create the expression tree

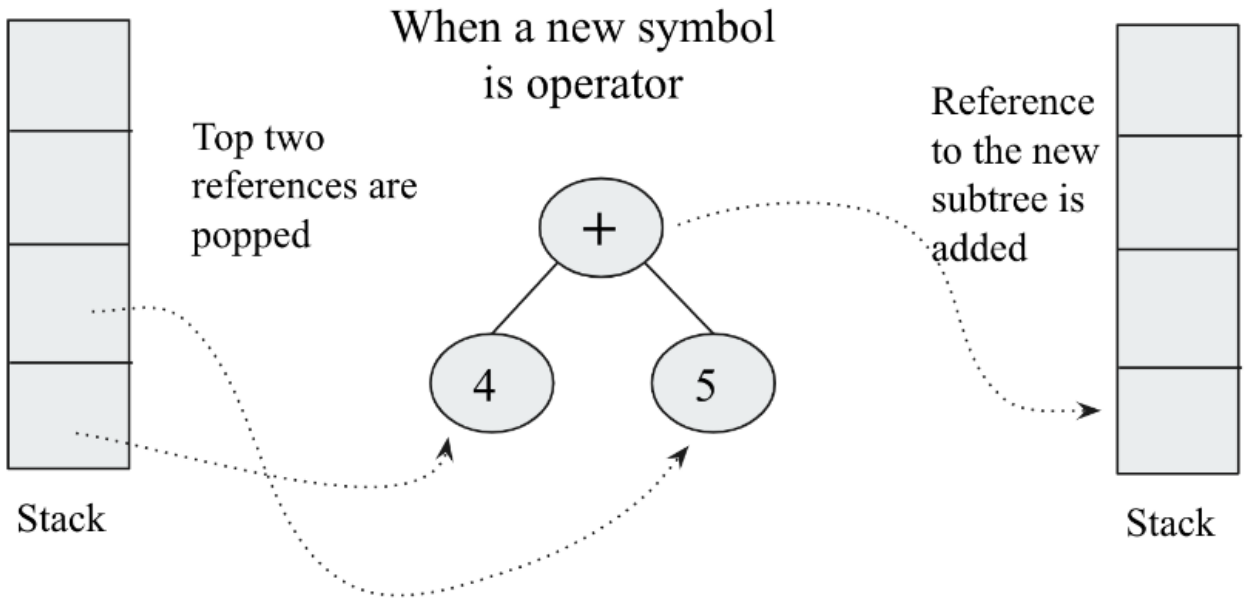
Parsing a reverse Polish expression

Example: 4 5 + 5 3 - *

Firstly, we push symbols 4 and 5 onto the stack, and then we process the next symbol +



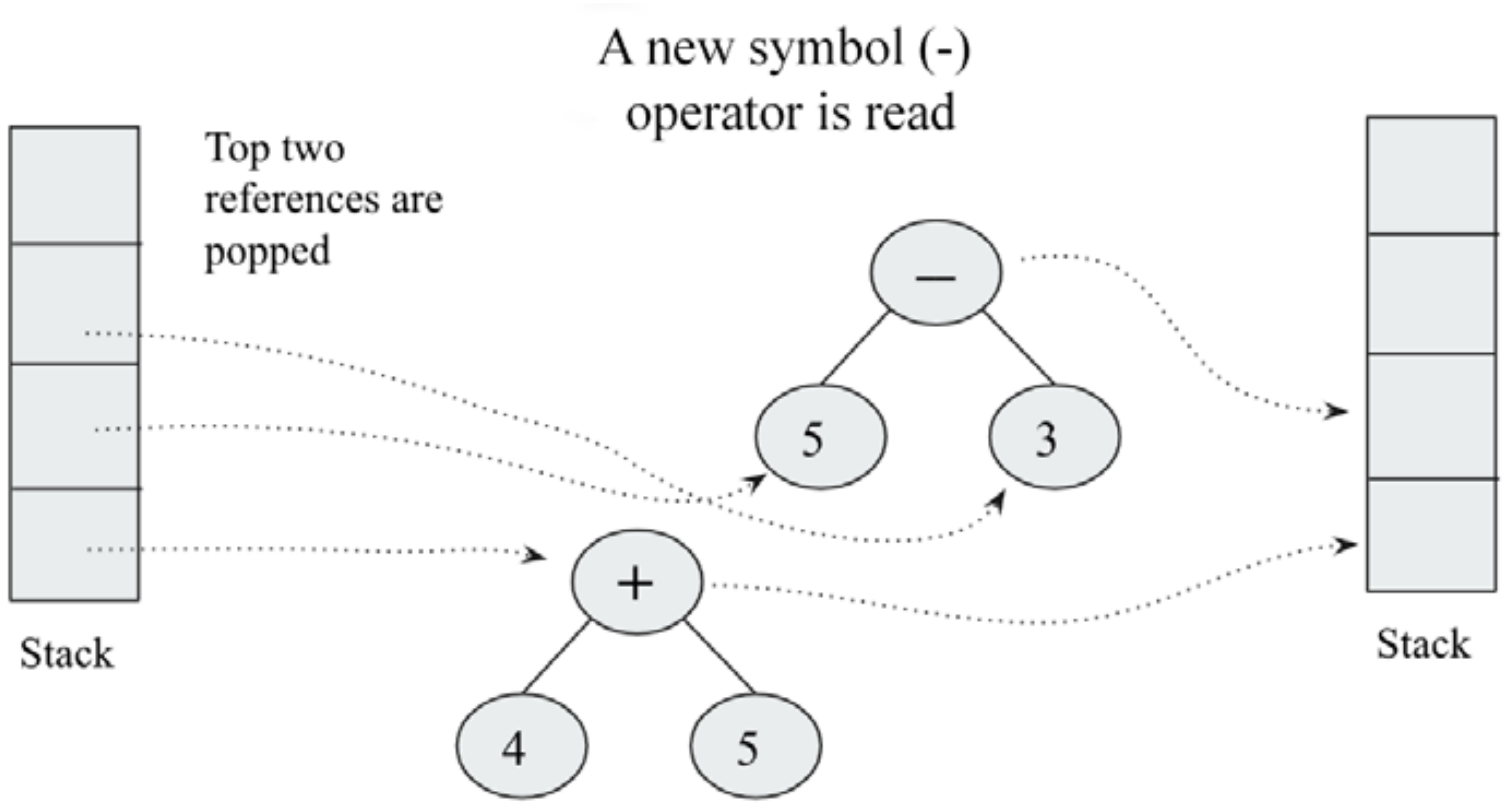
When the new symbol + is read, it is made into a root node of a new subtree, and then two references are popped from the stack, and the topmost reference is added as the right of the root node, and the next popped reference is added as the left child of the subtree



Parsing a reverse Polish expression

Example: 4 5 + 5 3 - *

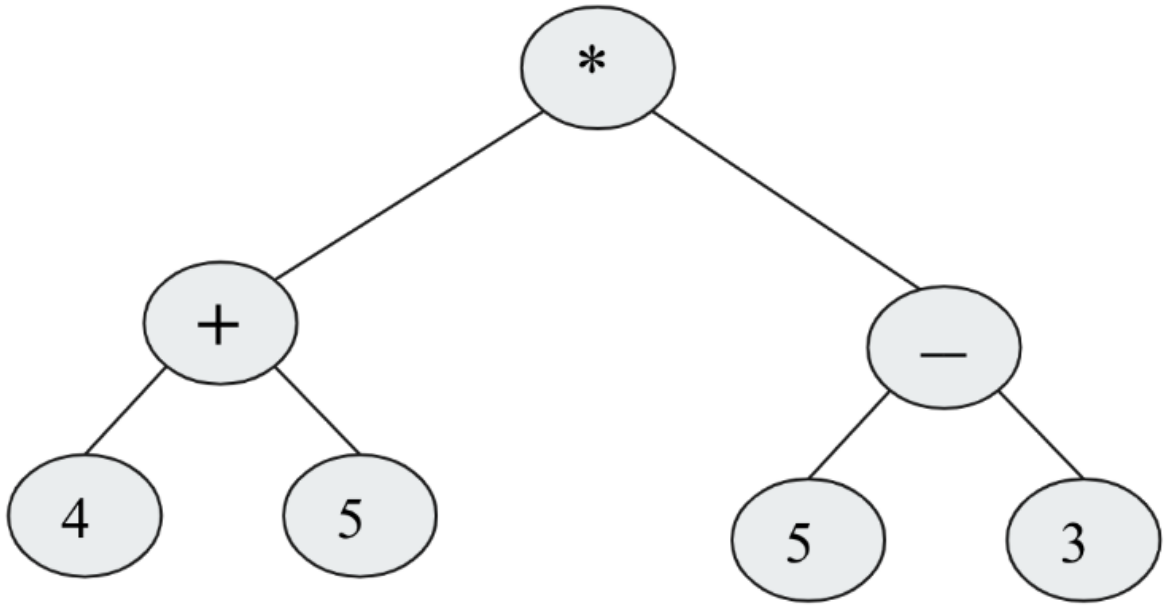
The next symbols are 5 and 3, and they are pushed into the stack. Next, when a new symbol is an operator (-), it is created as the root of the new subtree, and two top references are popped and added to the right and left child of this root respectively



Parsing a reverse Polish expression

Example: 4 5 + 5 3 - *

The next symbol is the operator *; as we have done so far, this will be created as the root, and then two references will be popped from the stack



CMC UNIVERSITY Parsing a reverse Polish expression

```
class TreeNode:
    def __init__(self, data=None):
        self.data = data
        self.right = None
        self.left = None
class Stack:
    def __init__(self):
        self.elements = []

    def push(self, item):
        self.elements.append(item)

    def pop(self):
        return self.elements.pop()
```

We will look at building a tree for an expression written in postfix notation. For this, we define a tree node.

Define the stack class

CMC UNIVERSITY Parsing a reverse Polish expression

```
expr = "4 5 + 5 3 - *".split()
stack = Stack()
```

an example of an arithmetic expression and set up the stack

In the first statement, the split() method splits on whitespace by default. The expr is a list with the values 4, 5, +, 5, 3, -, *

Each element of the expr list is going to be either an operator or an operand:

- If it is an operand, we embed it in a tree node and push it onto the stack.
- If it is an operator, we embed the operator into a tree node and pop its two operands into the node's right and left children.

Here, we have to take care to ensure that the first pop reference goes into the right child.

CMC UNIVERSITY Parsing a reverse Polish expression

```
for term in expr:
    if term in "+-*/":
        node = TreeNode(term)
        node.right = stack.pop()
        node.left = stack.pop()
    else:
        node = TreeNode(int(term))
    stack.push(node)
```

Notice that we perform a conversion from string to int in the case of an operand. You could use float() instead, if you wish to support floating-point operands.

At the end of this operation, we should have one single element in the stack, and that holds the full tree.

CMC UNIVERSITY Parsing a reverse Polish expression

To evaluate the expression, we can use the following function

```
def calc(node):
    if node.data == "+":
        return calc(node.left) + calc(node.right)
    elif node.data == "-":
        return calc(node.left) - calc(node.right)
    elif node.data == "*":
        return calc(node.left) * calc(node.right)
    elif node.data == "/":
        return calc(node.left) / calc(node.right)
    else:
        return node.data
```

```
root = stack.pop()
result = calc(root)
print(result)
```

CMC UNIVERSITY

BINARY SEARCH TREES

CMC UNIVERSITY

Searching

A search operation is carried out to find the location of the desired data item from a collection of data items. The search algorithm returns the location of the searched value where it is present in the list of items and if the data item is not present, it returns None.

There are two different ways in which data can be organized, which can affect how a search algorithm works:

- First, the search algorithm is applied to a list of items that is already sorted; that is, it is applied to an ordered set of items. For example, {1, 3, 5, 7, 9, 11, 13, 15, 17}.
- The search algorithm is applied to an unordered set of items, which is not sorted. For example, {11, 3, 45, 76, 99, 11, 13, 35, 37}.

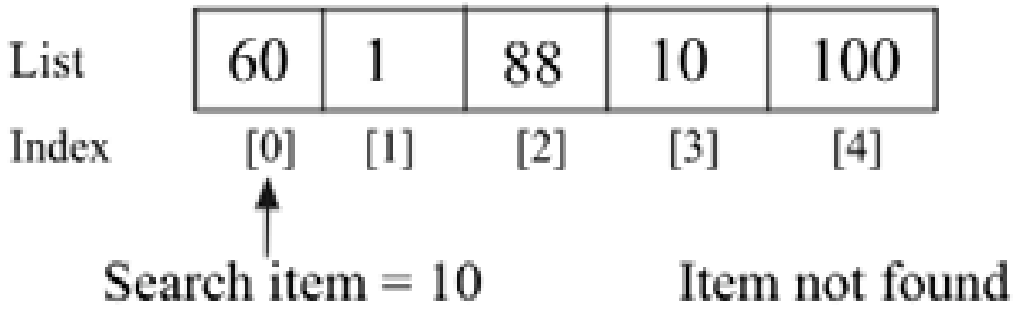
CMC UNIVERSITY

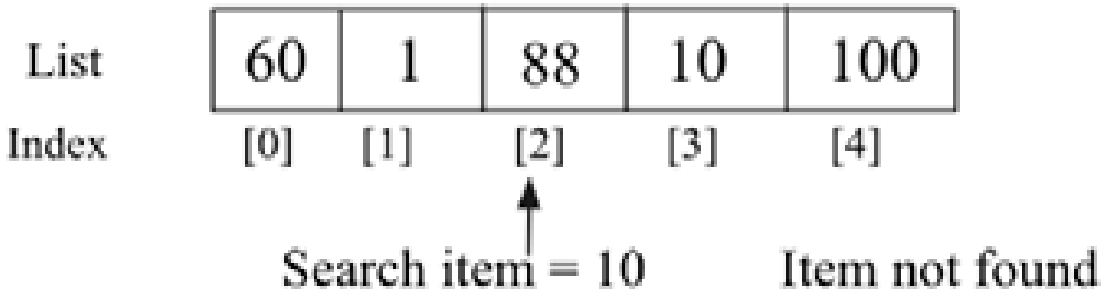
Linear search

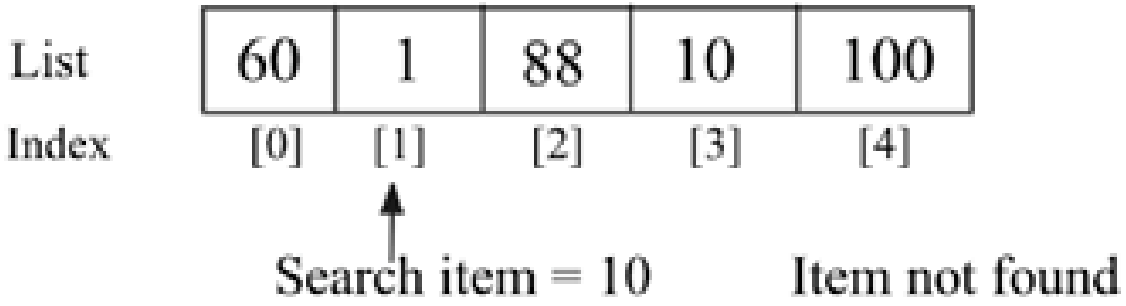
Unordered linear search

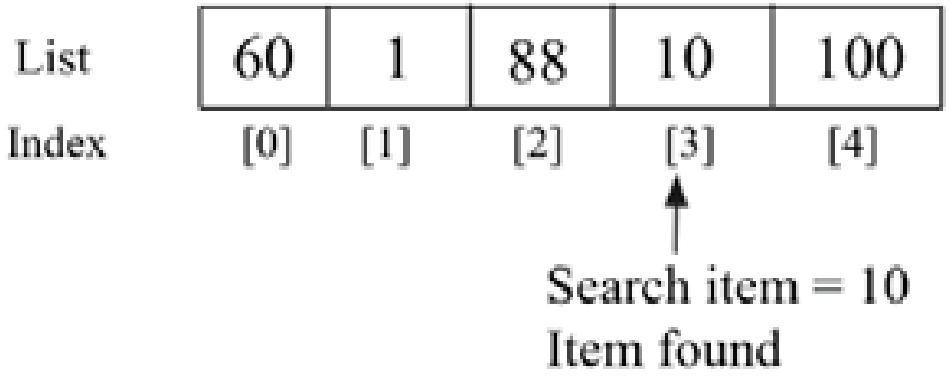
The unordered linear search is a linear search algorithm in which the given list of date items is not sorted. We linearly match the desired data item with the data items of the list one by one till the end of the list or until the desired data item is found.

Example: Find item 10 in the following sequence: {60, 1, 88, 10, 100}









CMC UNIVERSITY

Linear search

Unordered linear search

```
def search(unordered_list, term):
    for i, item in enumerate(unordered_list):
        if term == unordered_list[i]:
            return i
    return None
```

CMC UNIVERSITY

Linear search

Unordered linear search

```
list1 = [60, 1, 88, 10, 11, 600]


search_term = 10
index_position = search(list1, search_term)
print(index_position)
```

```
list2 = ['packt', 'publish', 'data']
search_term2 = 'data'
Index_position2 = search(list2, search_term2)
print(Index_position2)
```

The worst-case time complexity of an unordered linear search is O(n).

CMC UNIVERSITY

Linear search

Ordered linear search

If the data elements are already arranged in a sorted order, then the linear search algorithm can be improved. The linear search algorithm in a sorted list of elements has the following steps:

1. Move through the list sequentially

2. If the value of a search item is greater than the object or item currently under inspection in the loop, then quit and return None

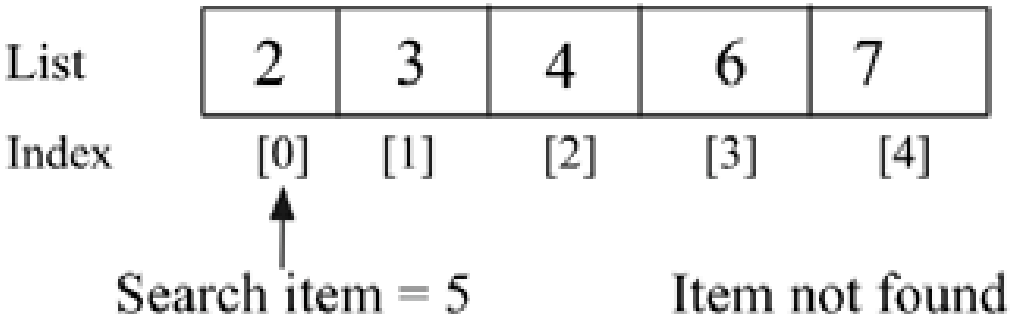
In the process of iterating through the list, if the value of the search term is less than the current item in the list, then there is no need to continue with the search.

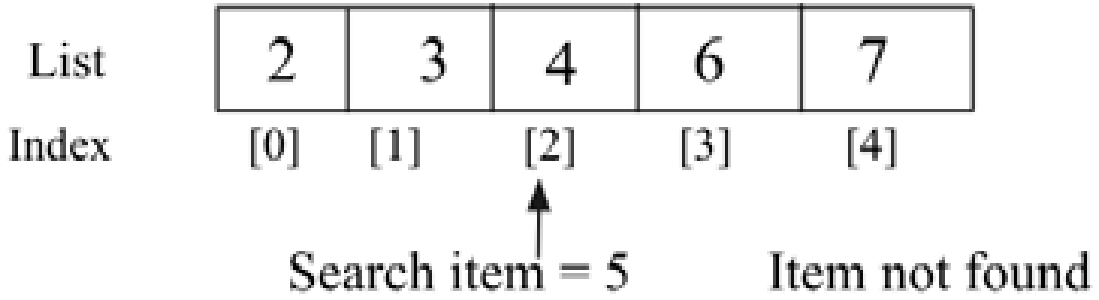
Example: search for item 5 in the list of items {2, 3, 4, 6, 7}

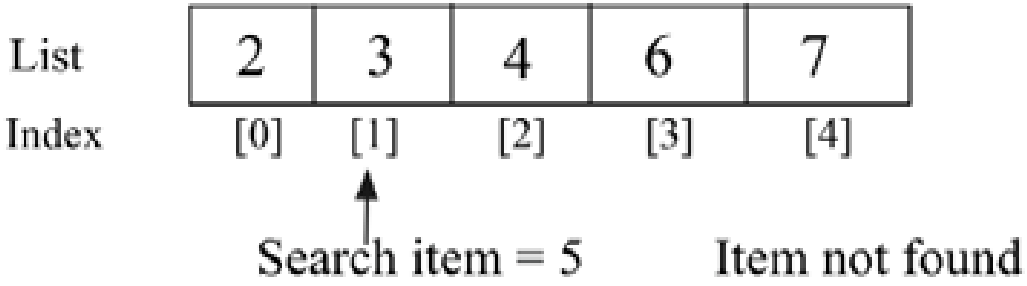
CMC UNIVERSITY

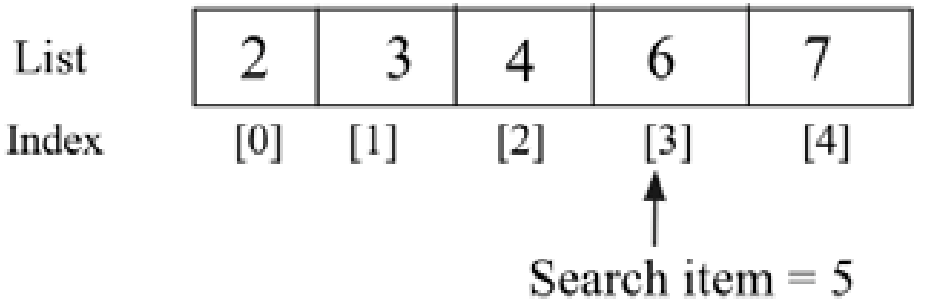
Linear search

Ordered linear search









CMC UNIVERSITY

Linear search

Ordered linear search

```
def search_ordered(ordered_list, term):
    ordered_list_size = len(ordered_list)
    for i in range(ordered_list_size):
        if term == ordered_list[i]:
            return i
        elif ordered_list[i] > term:
            return None
    return None
```

CMC UNIVERSITY

Linear search

Ordered linear search

```
list1 = [2, 3, 4, 6, 7]

search_term = 5
index_position1 = search_ordered(list1, search_term)

if index_position1 is None:
    print("{} not found".format(search_term))
else:
    print("{} found at position {}".format(search_term, index_position1))
```

CMC UNIVERSITY

Linear search

Ordered linear search

```
list2 = ['book','data','packt', 'structure']

search_term2 = 'structure'
index_position2 = search_ordered(list2, search_term2)

if index_position2 is None:
    print("{} not found".format(search_term2))
else:
    print("{} found at position {}".format(search_term2, index_position2))
```

The worst-case time complexity of an ordered linear search is O(n).

CMC UNIVERSITY

Jump search

- The jump search algorithm is an improvement over linear search for searching for a given element from an ordered (or sorted) list of elements.

- This method uses the divide-and-conquer strategy in order to search for the required element, compare the search value at different intervals in the list, which reduces the number of comparisons compared to the linear search

CMC UNIVERSITY

Jump search

Algorithm:

- Firstly, divide the sorted list of data into subsets of data elements called blocks. Within each block, the highest value will lie within the last element (the array is sorted)

- Next, start comparing the search value with the last element of each block.

There can be three conditions:

1. If the search value is greater than the last element of the block, compare it with the next block.

2. If the search value is less than the last element of the block, it means the desired search value must be present in the current block, apply linear search in this block and return the index position.

3. If the search value is the same as the compared element of the block, we return the index position of the element and we return the candidate.

CMC UNIVERSITY

Jump search

Generally, the size of the block is taken as $\sqrt{n}$, since it gives the best performance for a given array of length n.

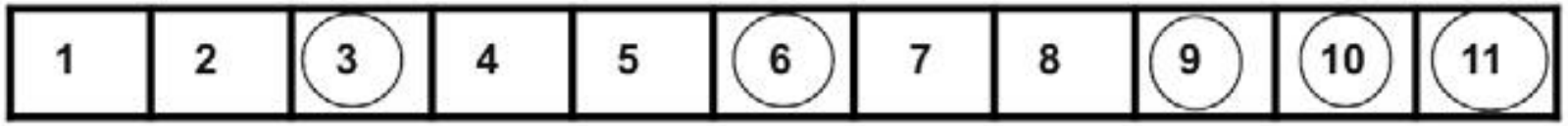
In the worst-case situation, we will have to make n/m number of jumps (n is the total number of elements, and m is the block size) if the last element of the last block is greater than the item to be searched, and we will need $m - 1$ comparisons for linear search in the last block.

Therefore, the total number of comparisons will be $((n/m) + m - 1)$, which will minimize when $m = \sqrt{n}$. So the size of the block is taken as $\sqrt{n}$ since it gives the best performance.

Jump search

Example: search for item 10 in the list {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

Block size (m) = 3
Let's say the item to be searched is 10.



Index (i) → [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]

Block 1 | Block 2 | Block 3 | Block 4

1
is A[0] <= item
i.e. 1 <10
Increment i.

2
Increment the index i:
i = i +m
= 0 + 3 = 3
Is A[i + m -1] < item
A[2] < item
3 < 10,
Move to the next block

3
Increment the index i:
i = i +m
= 3 + 3 = 6
Is A[i + m -1] < item
A[5] < item
6 < 10,
Move to the next block

4
Increment the index i:
i = i +m
= 6 + 3 = 9
Is A[i + m -1] < item
A[8] < item
9 < 10,
Move to the next block

5
Increment the index i:
i = i +m
= 9 + 2 = 11
Is A[i + m -1] < item
A[10] < item
11 < 10, Condition false, so, we stop here, and do linear search in the current block.

Jump search

The implementation of the jump searching algorithms:

- Firstly, we implement the linear search algorithm

```
1  def linear_search(subA, term):
2      print("Entering Linear Search")
3      size = len(subA)
4      for i in range(size):
5          if term == subA[i]:
6              return i
7          elif subA[i] > term:
8              return -1
9      return -1
```

Jump search

- Next, we implement the jump_search() method as follows:

```
10  def jump_search(A, item):
11      import math
12      print("Entering Jump Search")
13      sizeA = len(A)
14      block_size = int(math.sqrt(sizeA))
15      i = 0
16      while i != len(A)-1 and A[i] <= item:
17          print("Block under consideration - {}".format(A[i:i
                +block_size]))
18          if i+ block_size > len(A):
19              block_size = len(A) - i
20              block_list = A[i: i+block_size]
21              j = linear_search(block_list, item)
22              if j == -1:
23                  print("Element not found")
24                  return
25              return i + j
26          if A[i + block_size -1] == item:
27              return i+block_size-1
```

CMC UNIVERSITY

Jump search

```
26          if A[i + block_size -1] == item:
27              return i+block_size-1
28          elif A[i + block_size - 1] > item:
29              block_array = A[i: i + block_size - 1]
30              j = linear_search(block_array, item)
31              if j == -1:
32                  print("Element not found")
33                  return
34              return i + j
35          i += block_size
36
37  print(jump_search([1,2,3,4,5,6,7,8,9, 10, 11], 10))
```

Output

```
Entering Jump Search
Block under consideration - [1, 2, 3]
Block under consideration - [4, 5, 6]
Block under consideration - [7, 8, 9]
Block under consideration - [10, 11]
Entering Linear Search
9
```

CMC UNIVERSITY

Jump search

Jump search performs linear search on a block, so first it finds the block in which the element is present and then applies linear search within that block.

The size of the block depends on the size of the array. If the size of the array is n, then the block size may be $\sqrt{n}$. If it does not find the element in that block, it moves to the next block.

The jump search first finds out in which block the desired element may be present. For a list of n elements, and a block size of m, the total number of jumps possible will be n/m jumps.

When the size of the block is $\sqrt{n}$, the worst-case time complexity will be $O(\sqrt{n})$

CMC UNIVERSITY

Binary search

The binary search algorithm finds a given item from the given sorted list of items.

The binary search algorithm works as follows:

- It starts searching for the item by dividing the given list in half.

    + If the search item is smaller than the middle value then it will look for the searched item only in the first half of the list,

    + If the search item is greater than the middle value it will only look at the second half of the list.

- Repeat the same process every time until we find the search item, or we have checked the whole list.

In the case of a non-numeric list of data items, for example, if we have string data items, then we should sort the data items in alphabetical order (similar to how a contact list is stored on a phone).

Binary search

Example: search for item 43 from a list of 12 items

If we want to search 43 in the given list

since 43>37
We look only at the second half

since 43>37
We now look only at the first half of the list

Search item 43 is found.
The function will return the index position

CMC UNIVERSITY

Binary search

Implementation of the binary search algorithm on an ordered list of items

```
def binary_search_iterative(ordered_list, term):
    size_of_list = len(ordered_list) - 1
    index_of_first_element = 0
    index_of_last_element = size_of_list
    while index_of_first_element <= index_of_last_element:
        mid_point = (index_of_first_element + index_of_last_element)/2
        if ordered_list[mid_point] == term:
            return mid_point
        if term > ordered_list[mid_point]:
            index_of_first_element = mid_point + 1
        else:
            index_of_last_element = mid_point - 1
    if index_of_first_element > index_of_last_element:
        return None
```

CMC UNIVERSITY

Binary search

Implementation of the binary search algorithm on an ordered list of {10, 30, 100, 120, 500}

```
list1 = [10, 30, 100, 120, 500]


search_term = 10
index_position1 = binary_search_iterative(list1, search_term)
if index_position1 is None:
    print("The data item {} is not found".format(search_term))
else:
    print("The data item {} is found at position {}".format(search_term,
index_position1))
```

CMC UNIVERSITY

Binary search

Implementation of the binary search algorithm on an ordered list of {'book', 'data', 'packt', 'structure'}

```
list2 = ['book','data','packt', 'structure']
```

```
search_term2 = 'structure'
index_position2 = binary_search_iterative(list2, search_term2)
if index_position2 is None:
    print("The data item {} is not found".format(search_term2))
else:
    print("The data item {} is found at position {}".format(search_term2,
index_position2))
```

Binary search

In binary search, we repeatedly divide the search space (i.e. the list in which the desired item may lie) in half. Therefore, at each iteration, the size of the array reduces by half;

For example, at iteration 1, the size of the list is n, in iteration 2, the size of the list becomes n/2, in iteration 3 the size of the list becomes $n/2^2$, and after k iterations the size of the list becomes $n/2^k$. At that time the size of the list will be equal to 1.

That means:

$$\Rightarrow \quad n/2^k = 1$$

$$\Rightarrow \log_2(n) = \log_2(2k)$$

$$\Rightarrow \log_2(n) = k \log_2(2)$$

$$\Rightarrow k = \log_2(n)$$

the binary search algorithm has the worst-case time complexity of O(log n).

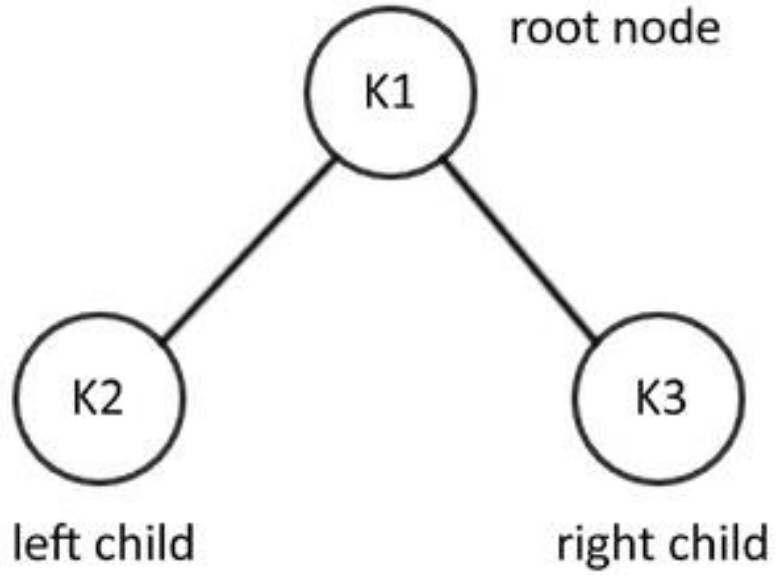
CMC UNIVERSITY

Binary search tree

- A binary search tree (BST) is a special kind of binary tree. It is one of the most important and commonly used data structures in computer science applications.
- A binary search tree provides very fast search, insertion, and deletion operations.
- A binary tree is called a binary search tree if the value at any node in the tree is greater than the values in all the nodes of its left subtree, and less than (or equal to) the values of all the nodes of the right subtree.

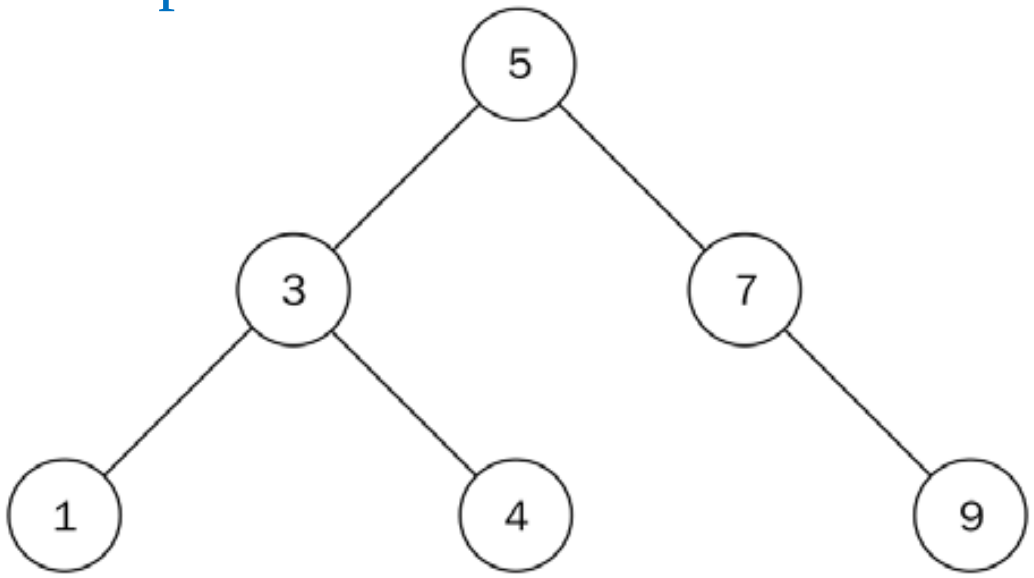
For example, if K1, K2, and K3 are key values in a tree of three nodes, then it should satisfy the following conditions:

- The key values K2 <= K1
- The key values K3 > K1

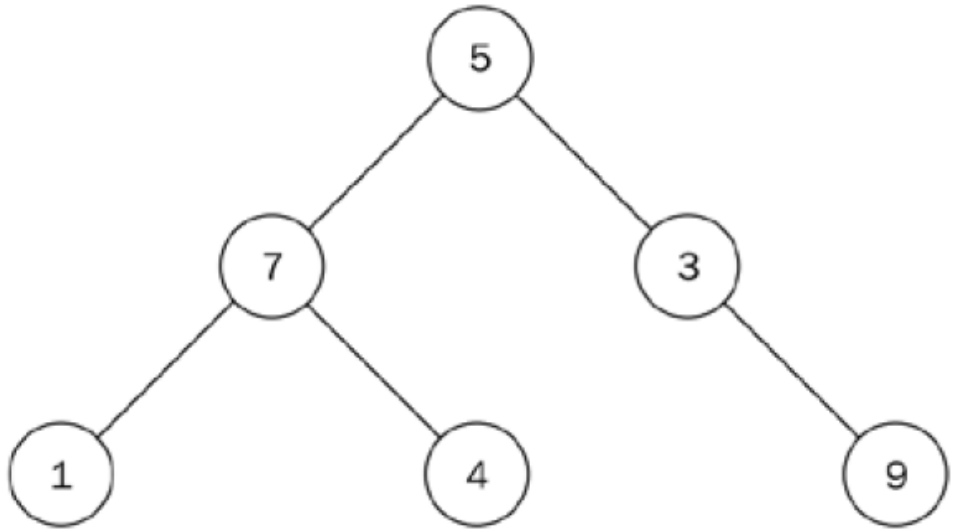


Binary search tree

Example:



Binary search tree of six nodes



A binary tree that is not a binary search tree

Since we need to keep track of the root node of the tree, we start by creating a Tree class that holds a reference to the root node:

```
class Tree:
    def __init__(self):
        self.root_node = None
```

CMC UNIVERSITY

Binary search tree operations

The operations that can be performed on a binary search tree are insert, delete, finding min, finding max, and searching

Inserting nodes: In order to insert a new element into a binary search tree, we have to ensure that the properties of the binary search tree are not violated after adding the new element.

- In order to insert a new element, we start by comparing the value of the new node with the root node: if the value is less than the root value, then the new element will be inserted into the left subtree; otherwise, it will be inserted into the right subtree. In this manner, we go to the end of the tree to insert the new element.

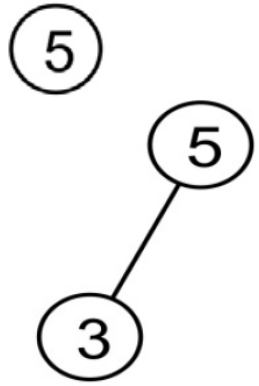
CMC UNIVERSITY

Binary search tree operations

Inserting nodes:

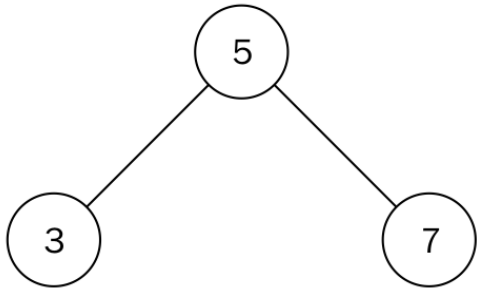
Example: create a binary search tree by inserting data items 5, 3, 7, 1

- Insert 5: create a node with its data attribute set to 5, since it is the first node

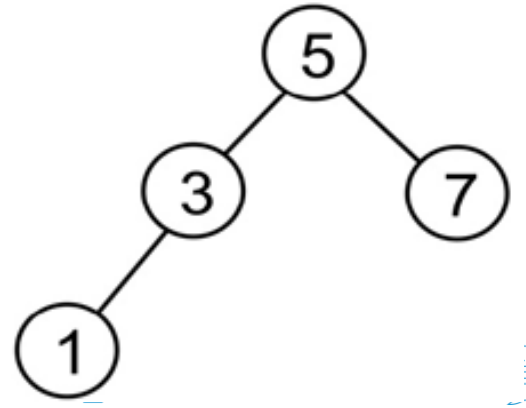


- Insert 3: the data value of 3 is compared with the existing node value, 5, of the root node. Since 3 < 5, it will be placed in the left subtree of node 5

- Insert 7: start from the root node with value 5, since 7 > 5, the node with a value of 7 is placed to the right of this root:



- Insert 1: Starting from the root of the tree, 1 < 5, go to the left subtree of 5, 1 < 3, move a level below node 3 and to its left. There is no node there, create a node with a value of 1 and associate it with the left pointer of node 3 to obtain the final tree.



CMC UNIVERSITY

Binary search tree operations

Inserting nodes

The Python implementation of the insert method to add the nodes in the binary search tree is given as follows

```
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None
```

```
class Tree:
    def __init__(self):
        self.root_node = None
```

CMC UNIVERSITY

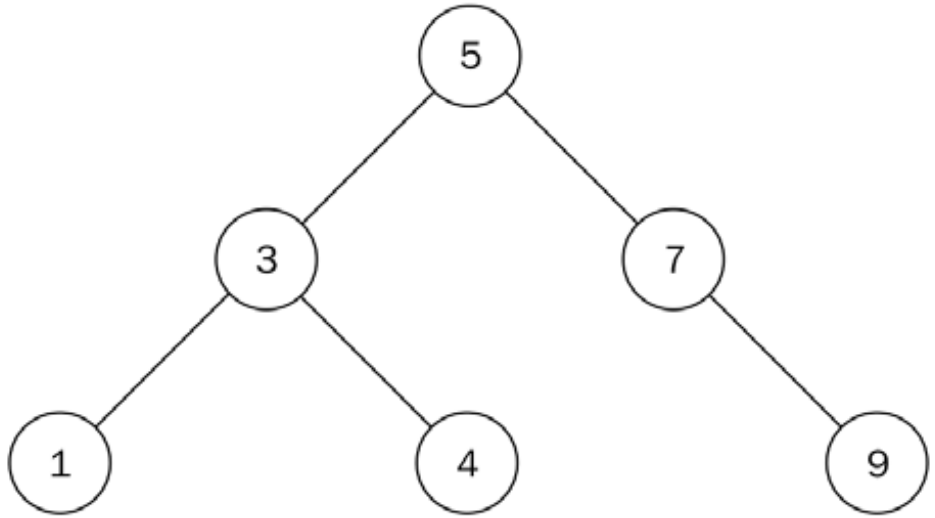
Binary search tree operations

```
def insert(self, data):
    node = Node(data)
    if self.root_node is None:
        self.root_node = node
        return self.root_node
    else:
        current = self.root_node
        parent = None
        while True:
            parent = current
            if node.data < parent.data:
                current = current.left_child
                if current is None:
                    parent.left_child = node
                    return self.root_node
            else:
                current = current.right_child
                if current is None:
                    parent.right_child = node
                    return self.root_node
```

CMC UNIVERSITY

Binary search tree operations

```
def inorder(self, root_node):
    current = root_node
    if current is None:
        return
    self.inorder(current.left_child)
    print(current.data)
    self.inorder(current.right_child)
```




```
tree = Tree()
r = tree.insert(5)
r = tree.insert(3)
r = tree.insert(7)
r = tree.insert(9)
r = tree.insert(1)
r = tree.insert(4)

tree.inorder(r)
```

Output

```
1
3
4
5
7
9
```

CMC UNIVERSITY

Binary search tree operations

Searching the tree

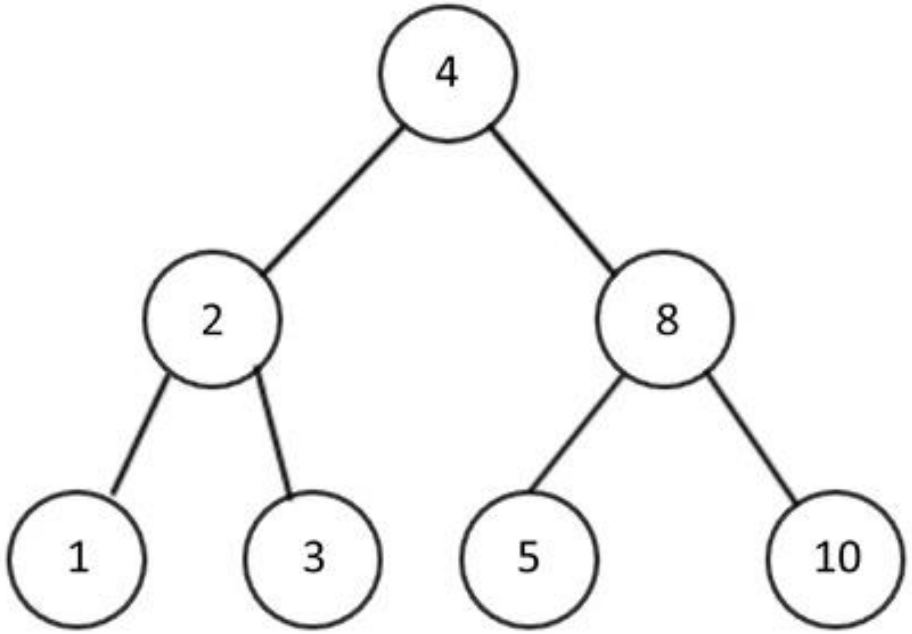
Example: binary search tree that has nodes 1, 2, 3, 4, 8, 5,10, as shown in the Figure.

Search for a node with a value of 5:

- start from the root node 4, 4 < 5

- move to the right subtree 8, 8 > 5

- move to the left subtree 5, 5 = 5, return "item found".



CMC UNIVERSITY

Binary search tree operations

```
def search(self, data):
    current = self.root_node
    while True:
        if current is None:
            print("Item not found")
            return None
        elif current.data is data:
            print("Item found", data)
            return data
        elif current.data > data:
            current = current.left_child
        else:
            current = current.right_child
```

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)


tree.search(9)
```

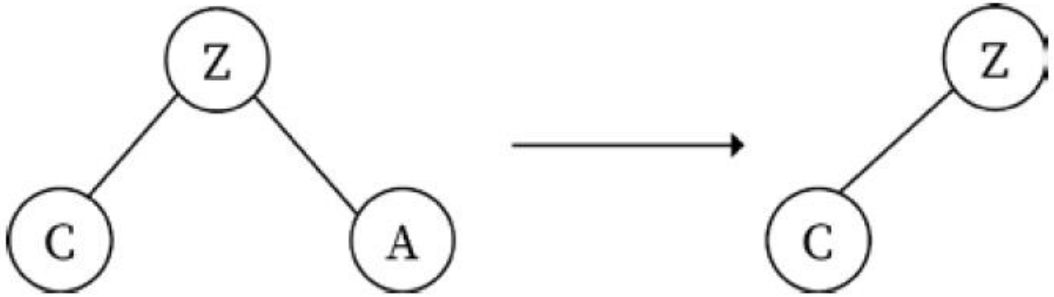
CMC UNIVERSITY

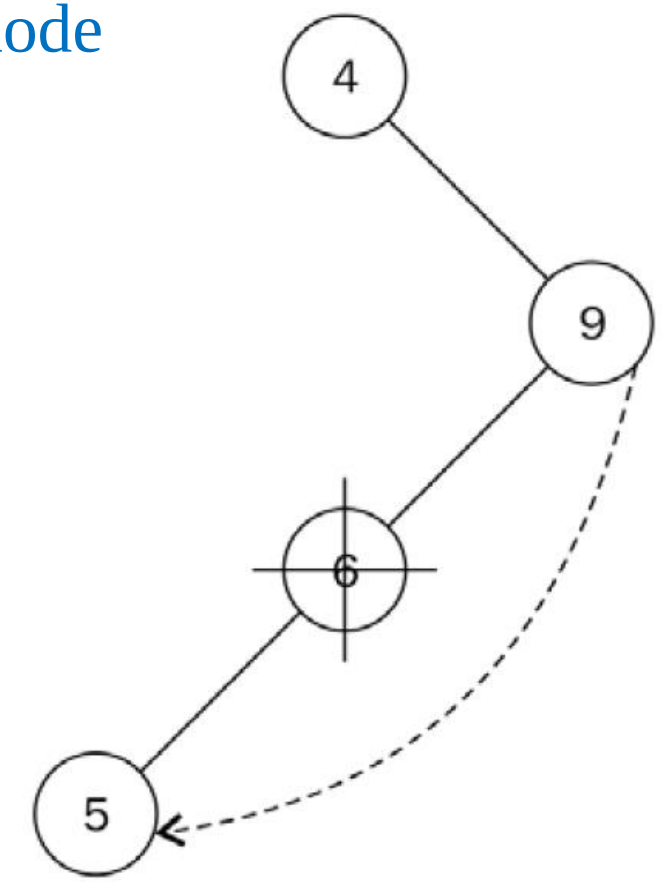
Binary search tree operations

Deleting nodes

There are three possible scenarios that we need to take care of during this process. The node that we want to remove might have the following:

(1) No children: If there is no leaf node, directly remove the node





(2) One child: swap the value of that node with its child, and then delete the node (the parent of that node is made to point to the child of that particular node)

Binary search tree operations

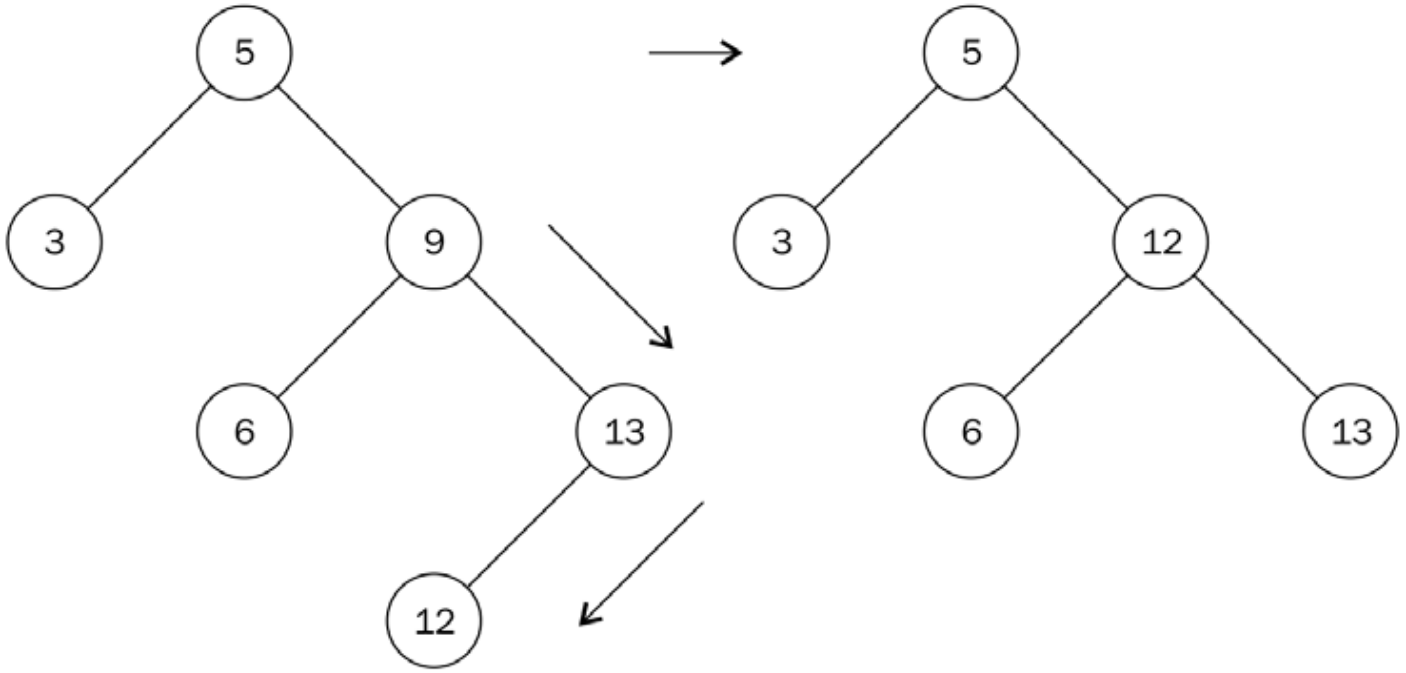
Deleting nodes

(3) Two children: first find the in-order successor or predecessor, swap their values, and then delete that node

Example: Delete node 9 in the following figure

- Find a successor node of node 9:
the node that has the minimum value in the right subtree, node 12 (the first element when apply the in-order traversal on the right subtree of the node 9)

- Move the content of the successor node 12 into the node 9 to be deleted.
- Delete the successor node 12



CMC UNIVERSITY

Binary search tree operations

Deleting nodes

To implement the above algorithm using Python, we need to write a helper method to get the node that we want to delete along with the reference to its parent node

```
def get_node_with_parent(self, data):
    parent = None
    current = self.root_node
    if current is None:
        return (parent, None)
    while True:
        if current.data == data:
            return (parent, current)
        elif current.data > data:
            parent = current
            current = current.left_child
        else:
            parent = current
            current = current.right_child

    return (parent, current)
```

CMC UNIVERSITY

Binary search tree operations

Deleting nodes

The only difference is that before we update the current variable inside the loop, we store its parent with parent = current. The method to do the actual removal of a node begins with this search

```
def remove(self, data):
    parent, node = self.get_node_with_parent(data)

    if parent is None and node is None:
        return False

    # Get children count
    children_count = 0

    if node.left_child and node.right_child:
        children_count = 2
    elif (node.left_child is None) and (node.right_child is None):
        children_count = 0
    else:
        children_count = 1
```

CMC UNIVERSITY

Binary search tree operations

Deleting nodes

The first part of the if statement handles the case where the node has no children:

```
if children_count == 0:
    if parent:
        if parent.right_child is node:
            parent.right_child = None
        else:
            parent.left_child = None
    else:
        self.root_node = None
```

CMC UNIVERSITY

Binary search tree operations

Deleting nodes

In cases where the node to be deleted has only one child, the elif part of the if statement does the following:

```
elif children_count == 1:
    next_node = None
    if node.left_child:
        next_node = node.left_child
    else:
        next_node = node.right_child

    if parent:
        if parent.left_child is node:
            parent.left_child = next_node
        else:
            parent.right_child = next_node
    else:
        self.root_node = next_node
```

CMC UNIVERSITY

Binary search tree operations

Deleting nodes

Lastly, we handle the condition where the node we want to delete has two children:

We update the node that's about to be removed with the value of the in-order successor with

node.data = leftmost_node.data:

```
else:
    parent_of_leftmost_node = node
    leftmost_node = node.right_child
    while leftmost_node.left_child:
        parent_of_leftmost_node = leftmost_node
        leftmost_node = leftmost_node.left_child

    node.data = leftmost_node.data
```

```
    if parent_of_leftmost_node.left_child == leftmost_node:
        parent_of_leftmost_node.left_child = leftmost_node.right_
child
    else:
        parent_of_leftmost_node.right_child = leftmost_node.right_
child
```

CMC UNIVERSITY

Binary search tree operations

Deleting nodes

The following code demonstrates how to use the remove method in the Tree class:

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)


tree.search(9)
tree.remove(9)
tree.search(9)
```

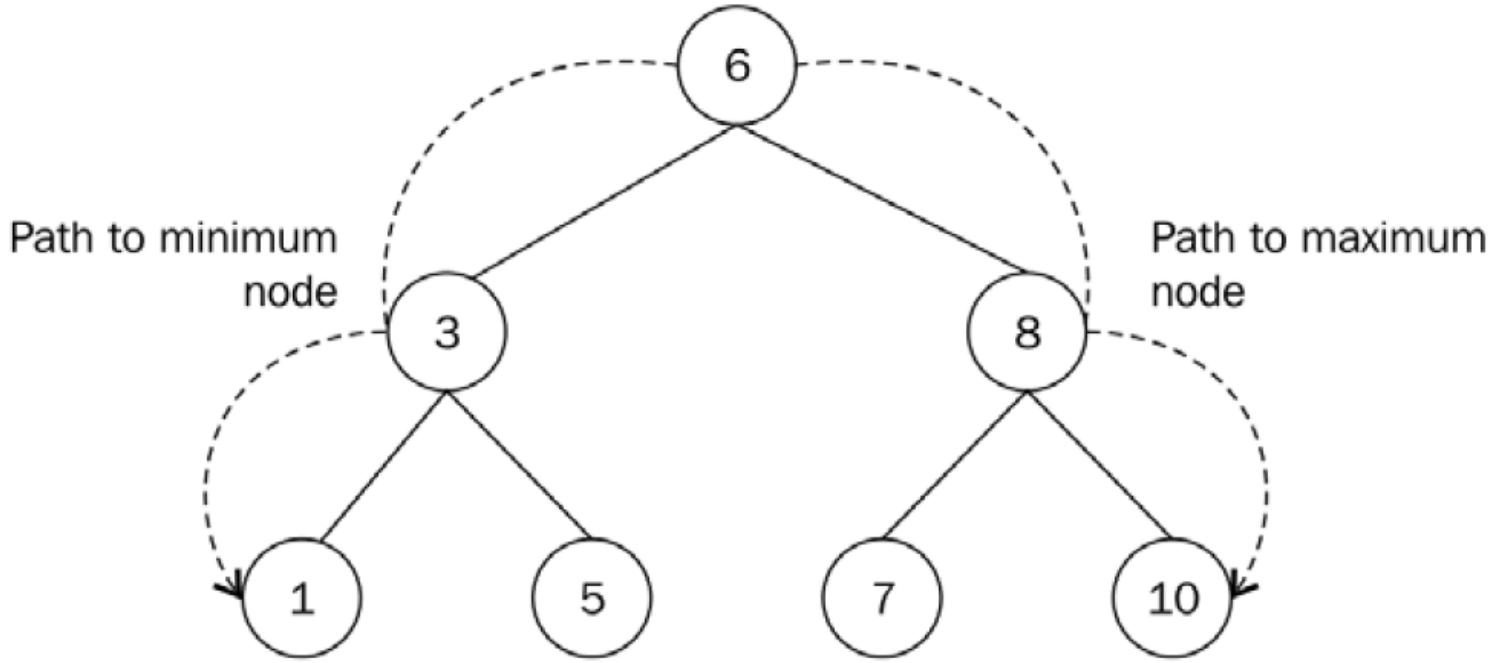
In the worst-case scenario, the remove operation takes O(h), where h is the height of the tree.

CMC UNIVERSITY

Binary search tree operations

Finding the minimum and maximum nodes

To find a node that has the smallest value in the tree, we start traversal from the root of the tree and visit the left node each time until we reach the end of the tree. Similarly, we traverse the right subtree recursively until we reach the end to find the node with the biggest value in the tree.



CMC UNIVERSITY

Binary search tree operations

Finding the minimum and maximum nodes

```
def find_min(self):
    current = self.root_node
    while current.left_child:
        current = current.left_child

    return current.data
```

```
def find_max(self):
    current = self.root_node
    while current.right_child:
        current = current.right_child

    return current.data
```

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)
print(tree.find_min())
print(tree.find_max())
```

CMC UNIVERSITY

Benefits of a binary search tree

Properties	Array	Linked list	BST
Data structure	Linear.	Linear.	Non-linear.
Ease of use	Easy to create and use. Average-case complexity for search, insert, and delete is `O(n)`.	Insertion and deletion are fast, especially with the doubly linked list.	Access of elements, insertion, and deletion is fast with the average-case complexity of `O(log n)`.
Access complexity	Easy to access elements. Complexity is `O(1)`.	Only sequential access is possible, so slow. Average- and worst-case complexity are `O(n)`.	Access is fast, but slow when the tree is unbalanced, with a worst-case complexity of `O(n)`.

CMC UNIVERSITY

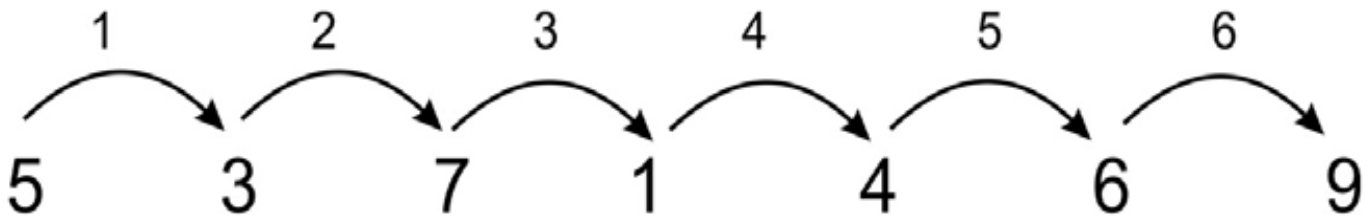
Benefits of a binary search tree

Properties	Array	Linked list	BST
Search complexity	Average- and worst-case complexity are `O(n)`.	It is slow due to sequential searching. Average- and worst-case complexity are `O(n)`.	Worst-case complexity for searching is `O(n)`.
Insertion complexity	Insertion is slow. Average- and worst-case complexity are `O(n)`.	Average- and worst-case complexity are `O(1)`.	The worst-case complexity for insertion is `O(n)`.
Deletion complexity	Deletion is slow. Average- and worst-case complexity are `O(n)`.	Average- and worst-case complexity are `O(1)`.	The worst-case complexity for deletion is `O(n)`.

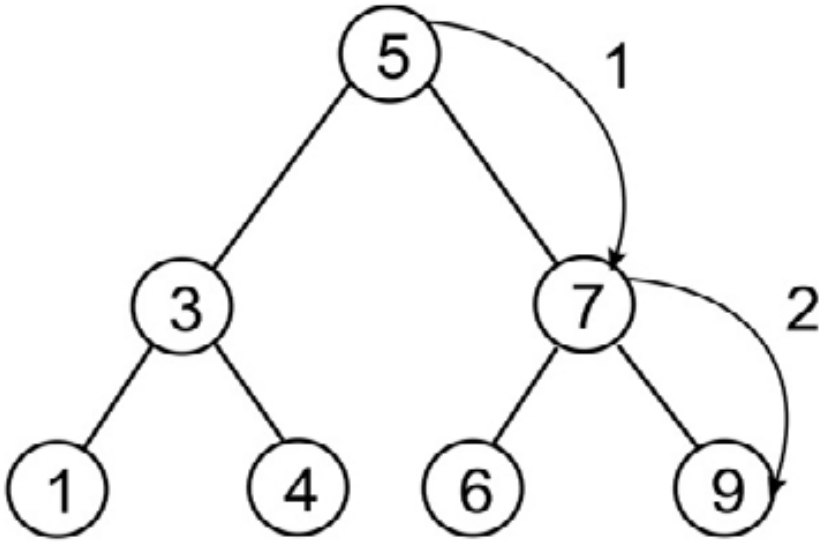
Benefits of a binary search tree

Example: We have the following data nodes 5, 3, 7, 1, 4, 6, 9

- If we use a list to store this data, the worst-case scenario will require to search through the entire list of 7 elements to find the item (require 6 comparisons to search for item 9)



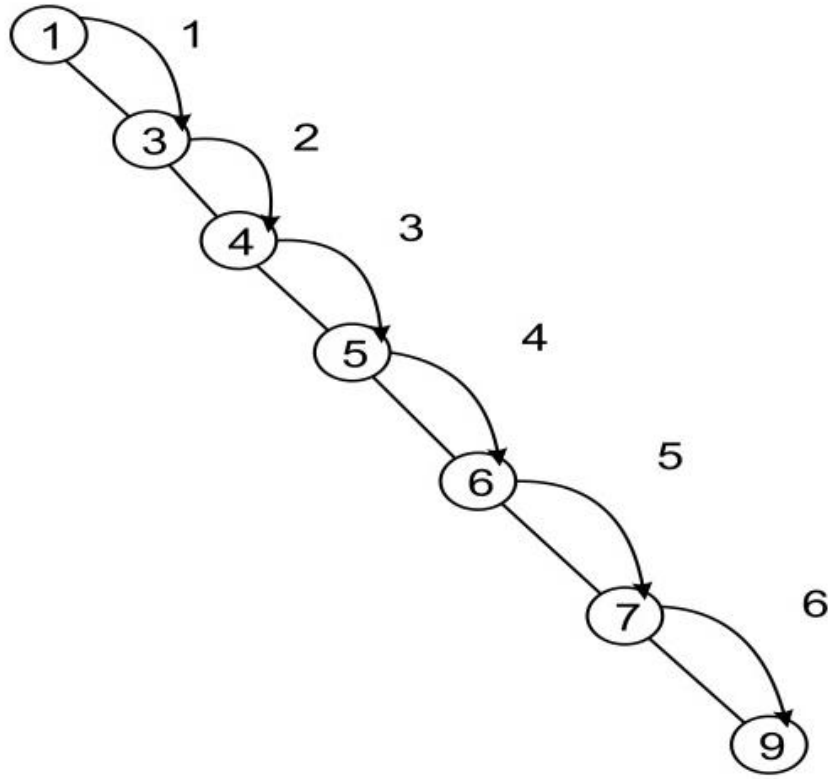
- If use a binary search tree to store these values, as shown in the following diagram, in the worst-case scenario, we will require two comparisons to search for item 9



Benefits of a binary search tree

Example: We have the following data nodes 5, 3, 7, 1, 4, 6, 9

If the elements are inserted into the tree in the order 1, 3, 4, 5, 6, 7, 9, as shown in the following Figure, then the tree would not be more efficient than the list.



Thus, it is important to use a method that can make the tree a self-balancing tree, which in turn will improve the search operation. Therefore, we should note that a binary search tree is a good choice if the binary tree is balanced.

CMC UNIVERSITY

BALANCED BINARY TREES

CMC UNIVERSITY

Balanced binary tree

Balanced may be defined by:

- Height balancing: comparing the heights of the two sub trees
- Null-path-length balancing: comparing the null-path-length of each of the two sub-trees (the length to the closest null sub-tree/empty node)
- Weight balancing: comparing the number of null sub-trees in each of the two sub trees

Balanced Binary trees are computationally efficient to perform operations.

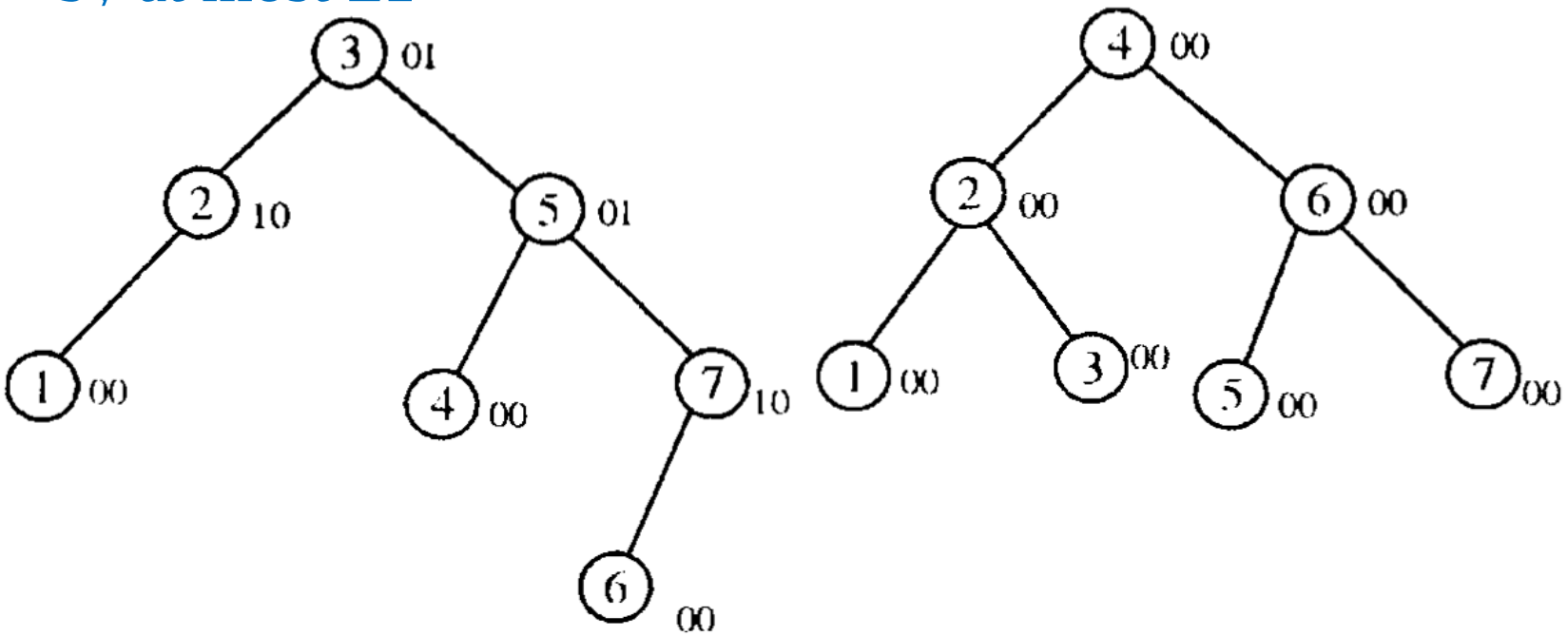
A balanced binary tree will follow the following conditions:

- The absolute difference of heights of left and right subtrees at any node is less than 1.
- For each node, its left subtree is a balanced binary tree.
- For each node, its right subtree is a balanced binary tree.

Balanced binary tree

AVL trees are self-balancing binary search trees. These trees are named after their two inventors G.M. Adel'son-Vel'skii and E.M. Landis.

An AVL tree is one that requires heights of left and right children of every node to differ by at most ±1



00: left and right subtrees have equal height (balanced)
01: the right subtree has a height greater than 1 (skewed to the right)
10: the left subtree has a height greater than 1 (skewed to the left)

We can see that a complete binary tree or a full binary tree is also an AVL balanced tree, but an AVL tree is not necessarily a complete or a full binary tree.

CMC UNIVERSITY

Balanced binary tree

Red-Black tree is a binary search tree in which every node is colored with either red or black. It is a type of self balancing binary search tree. It has a good efficient worst case running time complexity

Properties of Red Black Tree:

The Red-Black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties:

1. Root property: The root is black.

2. External property: Every leaf (Leaf is a NULL child of a node) is black in Red-Black tree.

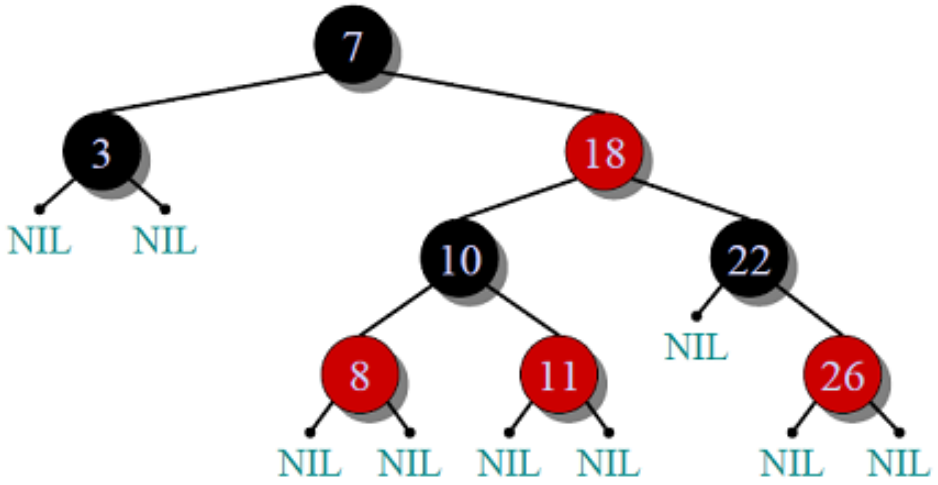
Balanced binary tree

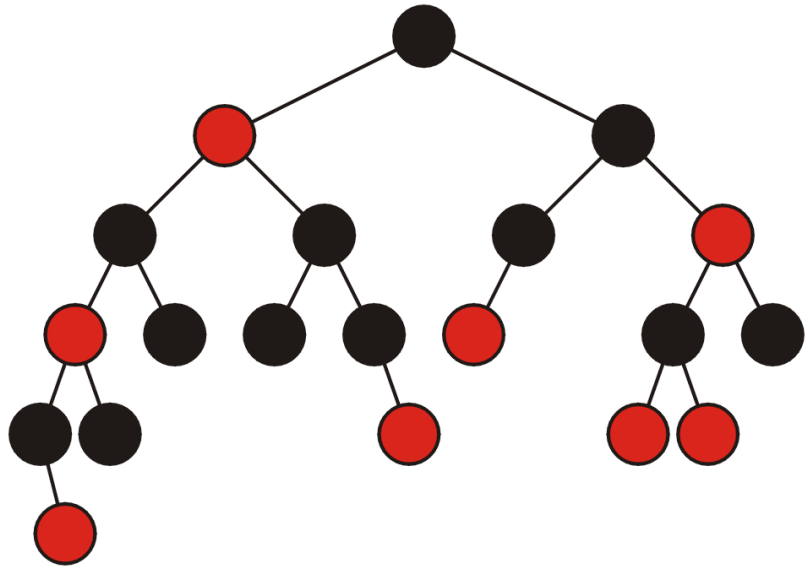
Red-Black tree

3. Internal property: The children of a red node are black. Hence possible parent of red node is a black node.

4. Depth property: All the leaves have the same black depth.

5. Path property: Every simple path from root to descendant leaf node contains same number of black nodes.





CMC UNIVERSITY

Balanced binary tree

Red-Black tree

Rules That Every Red-Black Tree Follows:

- Every node has a color either red or black.
- The root of the tree is always black.
- There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
- Every leaf (e.i. NULL node) must be colored BLACK.

CMC UNIVERSITY

Balanced binary tree

Exercises: (work in groups of 3 students)

1. Represent the AVL tree, perform the operations: search, max, min, insert, delete
2. Represent the Red - Black tree, perform the operations: search, max, min, insert, delete


CMC UNIVERSITY
THANK YOU