**EXPERIMENT 10**

**AIM**

Write a menu driven program in C to implement BFS and DFS

**1. BFS:-**

**THEORY**

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.

That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

**Algorithm:**

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]

Step 6: EXIT

**Complexity of Binary Search Algorithm:**

***Space complexity*** In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated. The space complexity is therefore proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level O(bd).

If the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as O ( | E | + | V | ), where | E | is the total number of edges in G and | V | is the number of nodes or vertices.

***Time complexity*** In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches O(bd). However, the time complexity can also be expressed as O( | E | + | V | ), since every vertex and every edge will be explored in the worst case.

***Completeness*** Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge.

***Optimality*** Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node. But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

**Applications of Breadth First Search Algorithm**:-

* Breadth-first search can be used to solve many problems such as: Finding all connected components in a graph G.
* Finding all nodes within an individual connected component.
* Finding the shortest path between two nodes, u and v, of an unweighted graph.
* Finding the shortest path between two nodes, u and v, of a weighted graph.

**1. DFS:-**

**THEORY**

The depth-first search algorithm (Fig. 13.22) progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the deadend, we backtrack to find another path P’. The algorithm terminates when backtracking leads back to the starting node A. In this algorithm, edges that lead to a new vertex are called *discovery edges* and edges that lead to an already visited vertex are called *back edges*.

**Algorithm:**

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting nodeAon the stack and set its STATUS=2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS=3 (processed state)

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS=1) and

set their STATUS=2 (waiting state) [END OF LOOP]

Step 6: EXIT

**Features of Depth-First Search Algorithm**

***Space complexity*** The space complexity of a depth-first search is lower than that of a breadth-first search.

***Time complexity*** The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V| + |E|)).

***Completeness*** Depth-first search is said to be a complete algorithm. If there is a solution, depth-first search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

**Applications of Depth-First Search Algorithm**

* Depth-first search is useful for: Finding a path between two specified nodes, u and v, of an unweighted graph.
* Finding a path between two specified nodes, u and v, of a weighted graph.
* Finding whether a graph is connected or not.
* Computing the spanning tree of a connected graph.

**EXPERIMENT 8**

**AIM:**

WAP in C to implement Binary Search Tree Menu driven program

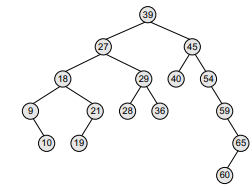
**THEORY**

A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order. In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree.

Binary search trees also speed up the insertion and deletion operations. The tree has a speed advantage when the data in the structure changes rapidly. Binary search trees are considered to be efficient data structures especially when compared with sorted linear arrays and linked lists.

In a sorted array, searching can be done in O(log2n) time, but insertions and deletions are quite expensive. In contrast, inserting and deleting elements in a linked list is easier, but searching for an element is done in O(n) time. However, in the worst case, a binary search tree will take O(n) time to search for an element. The worst case would occur when the tree is a linear chain of nodes

Due to its efficiency in searching elements, binary search trees are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.



In the following figure. The root node is 39. The left sub-tree of the root node consists of nodes 9, 10, 18, 19, 21, 27, 28, 29, and 36. All these nodes have smaller values than the root node. The right sub-tree of the root node consists of nodes 40, 45, 54, 59, 60, and 65. Recursively, each of the sub-trees also obeys the binary search tree constraint.

Also each of the node has two, one or no child. So the given tree is binary search tree.

Inorder traversal of this tree gives data in ascending order.

**Algorithm**

**1. INSERT**

Insert (TREE, VAL)

Step 1: IF TREE = NULL

Allocate memory for TREE

SET TREE->DATA = VAL

SET TREE->LEFT = TREE->RIGHT = NULL

ELSE

IF VAL<TREE->DATA

Insert(TREE->LEFT, VAL)

ELSE

Insert(TREE->RIGHT, VAL)

[END OF IF]

[END OF IF]

Step 2: END

**2. DELETE**

Delete (TREE, VAL)

Step 1: IF TREE = NULL

Write "VAL not found in the tree"

ELSE IF VAL < TREE->DATA

Delete(TREE->LEFT, VAL)

ELSE IF VAL > TREE->DATA

Delete(TREE->RIGHT, VAL)

ELSE IF TREE->LEFT AND TREE->RIGHT

SET TEMP = findLargestNode(TREE->LEFT)

SET TREE->DATA = TEMP->DATA

Delete(TREE->LEFT, TEMP->DATA)

ELSE

SET TEMP = TREE

IF TREE->LEFT = NULL AND TREE->RIGHT = NULL

SET TREE = NULL

ELSE IF TREE->LEFT != NULL

SET TREE = TREE->LEFT

ELSE

SET TREE = TREE->RIGHT

[END OF IF]

FREE TEMP

[END OF IF]

Step 2: END

**3. DISPLAY**

**3.1. PREORDER**

PREORDER(TREE)

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: Write TREE->DATA

Step 3: PREORDER(TREE->LEFT)

Step 4: PREORDER(TREE->RIGHT)

[ END OF LOOP ]

Step 5: END

**3.2. INORDER**

INORDER(TREE)

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: INORDER(TREE->LEFT)

Step 3: Write TREE->DATA

Step 4: INORDER(TREE->RIGHT)

[ END OF LOOP ]

Step 5: END

**3.3. POSTORDER**

POSTORDER(TREE)

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: POSTORDER(TREE->LEFT)

Step 3: POSTORDER(TREE->RIGHT)

Step 4: Write TREE->DATA

[ END OF LOOP ]

Step 5: END

***Complexity of Insert and delete***

The insert function requires time proportional to the height of the tree in the worst case. It takes O(log n) time to execute in the average case and O(n) time in the worst case. Because, If the current node’s value is less than that of the new node, then the right sub-tree is traversed, else the left sub-tree is traversed. The insert function continues moving down the levels of a binary tree until it reaches a leaf node. The new node is added by following the rules of the binary search trees. That is, if the new node’s value is greater than that of the parent node, the new node is inserted in the right sub-tree, else it is inserted in the left sub-tree.

The delete function requires time proportional to the height of the tree in the worst case. It takes O(log n) time to execute in the average case and Ω(n) time in the worst case. Because each time we first search the element and delete it so it is same for finding height of tree.

**Features**

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.
* Both subtrees of each node are also BSTs i.e. they have the above two properties

**Application**

* In multilevel indexing in the database
* For dynamic sorting
* For managing virtual memory areas in Unix kernel

**Advantages**

* We have an ordering of keys stored in the tree. Any time we need to traverse the increasing (or decreasing) order of keys, we just need to do the in-order (and reverse in-order) traversal on the tree.
* We can implement order statistics with binary search tree - Nth smallest, Nth largest element. This is because it is possible to look at the data structure as a sorted array.
* We can also do range queries - find keys between N and M (N <= M).
* BST can also be used in the design of memory allocators to speed up the search of free blocks (chunks of memory), and to implement best fit algorithms where we are interested in finding the smallest free chunk with size greater than or equal to size specified in allocation request.

**Disadvantages**

* The main disadvantage is that we should always implement a balanced binary search tree - AVL tree, Red-Black tree, Splay tree. Otherwise the cost of operations may not be logarithmic and degenerate into a linear search on an array.