|  |  |
| --- | --- |
| **Practical No: 07** | |
|  | |
| **Explanation/ Stepwise-Procedure/ Algorithm** | Write a menu-driven program to  i. Create a binary search tree  ii. Traverse the tree in inorder, preorder and postorder  iii. Search  iv. delete a node from the tree. |
| **Theory:** | What is a tree? A tree is a kind of data structure that is used to represent the data in hierarchical form. It can be defined as a collection of objects or entities called as nodes that are linked together to simulate a hierarchy. Tree is a non-linear data structure as the data in a tree is not stored linearly or sequentially. What is a Binary Search tree? A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.Let's understand the concept of Binary search tree with an example.  https://lh3.googleusercontent.com/lccWu8_UEVzUJEbkihMHUN1IuHdO0GA2xwCygjBkVnd_VUI87ZSeaQcxrRp1VSm39Tr4UgP_ldWFUz6X0-bF2cCKyNe-i5L_Qg8zijRCAYPZiG9FQT41hZOLg7fNrejo9ohgd7ymB4PbKb0fs31Usw  In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.  Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.  Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.  https://lh5.googleusercontent.com/MEbIs6I_EicloC_UuA_-O8Fi3UQdjQ65wjBrG96eUxd_sBjLvQfadbvmoBOinn4g4nYy9A5oikpXs5uhMYmYSksRCmPoIeyM3_D5frqpGZwnnO0j9LOz43QSPDK7YhnJiRYdrKHzTduyuj2Y2xXSzg  In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree. Advantages of Binary search tree  * Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element. * As compared to array and linked lists, insertion and deletion operations are faster in BST.  Example of creating a binary search tree Now, let's see the creation of binary search tree using an example.  Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**   * First, we have to insert **45** into the tree as the root of the tree. * Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element. * Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.   Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -  **Step 1 - Insert 45.**  Binary Search tree  **Step 2 - Insert 15.**  As 15 is smaller than 45, so insert it as the root node of the left subtree.  https://lh5.googleusercontent.com/kmE7Ya2bMMVQlkNukLaQ8FU5SLSXBfKt6PioWeLXIx-wFyPgnC69EFCXCw79BMUuQBJ2oZET1Kd7aZDQDgd6hTJRLzUya9FtAB5x6lJDHjD0c6renWIz80pPtUSRudqJFoax8GDRI4RVIbFTYpwcuQ  **Step 3 - Insert 79.**  As 79 is greater than 45, so insert it as the root node of the right subtree.  https://lh4.googleusercontent.com/XJBGbIzhZnZR1mAVVgtJIXdKi1DiEB6I_voQJBf4WdljouhVqHabcFR9S4RNGoGycs7figscxQXFwcEdr55nCT0yHAIyqZo1WodoD-X-Pk_zvbRbplPeIPfIPvgre0Js9Snl6zn6Bl_VXCVhyoYwzA  **Step 4 - Insert 90.**  90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.  https://lh5.googleusercontent.com/PXWhmVboio2FuWiWnxCEtibG4c9se_Dga6Gr0QKLRkO2rNfQhaGCqAKYKpYSuF7bvSlpmb8PODXzJNDVGGRUu9Wa4srfMQc5YDHPwN0a5qxi39cTR5kneZ0Mo-JaQtSdvAGvely9C5UYpEPD2sdH9g  **Step 5 - Insert 10.**  10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.  https://lh5.googleusercontent.com/TC7mLLRe-sABcen4LC2ajyrPUiz_a6tI8gjSzA_n-eEaYAGhXGAgtR8AszsYxRvzVkHq2kc1gxyKeQG25Ie-cxKm0UgJVB_rCtQJq7uB22BHo_WavhKrQToQ5RbPQ2QkoIdLRQEGFSarA6kEZT7_ew  **Step 6 - Insert 55.**  55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.  https://lh5.googleusercontent.com/KQiUgmTJzxZu-Ge7ntt6dO5t8Px8-6Zylamu2irB8zieS_5jgYqd-FiTkTD0dWuDPrpfcMZq5XBCkrJX2XmRNBHQ6xJpemnd0v7TkeeVDTuqCvBLeju-NnWLG9NjVc-bOP553m6VxykHNz3e2tpgaA  **Step 7 - Insert 12.**  12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.  https://lh3.googleusercontent.com/SHromwq-YfK42bZ0crOfk90jVxCwaIt85Zga80DWg1ykuOaCRX6KHJ7mfeMsv7qkSSLblw3X_Mzyc8oEJ4VsumPiyAdDWABQejZVfCiFwRcKCONkE-_L6L6xF8rk9r3sKYtYS2VWKldDPrP8rgjXtQ  **Step 8 - Insert 20.**  20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.  https://lh4.googleusercontent.com/Kw4jCOt2at21pNdV190tGE5W1q--5zyDZzguWtuJrZT61irxi6Vaz4dcDt8lEbrLvQTZJrR2Um1R_cOK6jj9e3JW2g6wdUVhw7GNn6cMcjnSmb9w8xfPBuEQdNq3hu2uM-SKFYTnmkqwXFta7S1mrg  **Step 9 - Insert 50.**  50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.  https://lh3.googleusercontent.com/xvBAoePWRsSP2Q1wyrKX-83eaFICO25Oz8LDdKnevLMaXjGgbUOCNnpZ7-A4TUrg6HMKrLC97En7ddCcZFOQzeLLf_hTwRYpUZD9MQQWAMPe5MC3ZFYpu2zUb4ezN7j3FErHkpOpNnnexW5znn3kkw  Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.  We can perform insert, delete and search operations on the binary search tree.  Let's understand how a search is performed on a binary search tree. Searching in Binary search tree Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -   1. First, compare the element to be searched with the root element of the tree. 2. If root is matched with the target element, then return the node's location. 3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree. 4. If it is larger than the root element, then move to the right subtree. 5. Repeat the above procedure recursively until the match is found. 6. If the element is not found or not present in the tree, then return NULL.   Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.  **Step1:**  Binary Search tree  **Step2:**  Binary Search tree  **Step3:**  Binary Search tree  Now, let's see the algorithm to search an element in the Binary search tree.  https://lh5.googleusercontent.com/BJoNA6dGW2tfEeLVI91c3cGffF3EIlGDyxYmsyANU0LxMmGp4DuyZSoHz03V2ER_JWpWhStJ1jtLL6zTwlzS7kPPBrXASOd3Mf5g8WGXIagLmMI0RrKF8oF__Zl4z_uYdmX1_RnqDLQ7BIJza-8WhQ Deletion in Binary Search tree In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -   * The node to be deleted is the leaf node, or, * The node to be deleted has only one child, and, * The node to be deleted has two children   We will understand the situations listed above in detail. When the node to be deleted is the leaf node It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.  We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.  Binary Search tree When the node to be deleted has only one child In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.  We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.  So, the replaced node 79 will now be a leaf node that can be easily deleted.  Binary Search tree When the node to be deleted has two children This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -   * First, find the inorder successor of the node to be deleted. * After that, replace that node with the inorder successor until the target node is placed at the leaf of tree. * And at last, replace the node with NULL and free up the allocated space.   The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.  We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.  Binary Search tree  Now let's understand how insertion is performed on a binary search tree. Insertion in Binary Search tree A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.  Now, let's see the process of inserting a node into BST using an example.  Binary Search tree  Binary Search tree The complexity of the Binary Search tree Let's see the time and space complexity of the Binary search tree. We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.   1. Time Complexity   https://lh4.googleusercontent.com/Bu92qgdX_o0RWMhBf-qCG1QqXlr19Jp36-gt8-ozK48sjJsulgrtfh_7Wif9VX2ERocmN8cIHKYhEE_B5MGWFKUWT0v75K65gNDLPglED4nRIDKE3VbLs7d7jub4klJkfMXs2h_wEMtMLkBQ38UFCA   1. Space complexity:   https://lh6.googleusercontent.com/ku7waMWDe8cVaSCIN2SAHYE2DTMLhne8baSLZ8OCMqUg1px5F2qdoI-L973kGCbuOrNnb0DCmCZlU8tHU3FMJJXj1JiDSYoSBg2e9NSViyXFMe_cLEAnb0XZZPbCZ6GZ09I1CKuN01mi8Y9eUKjY4g  **In order traversal:**  **Steps**  1.Traverse the left sub-tree in in-order  2.Visit the root  3.Traverse the right sub-tree in in-order  **Algorithm**  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  **Pre-order traversal:**  **Steps**  1.Visit the root node  2.traverse the left sub-tree in pre-order  3.traverse the right sub-tree in pre-order  **Algorithm**  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  **Post-order traversal:**  **Steps:**  1. Traverse the left subtree by calling the postorder function recursively.  2. Traverse the right subtree by calling the postorder function recursively.  3. Access the data part of the current node.  **Algorithm**:  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 |
| **Source Code/Algorithm/Flow Chart:** | #include <stdio.h>  #include <stdlib.h>  typedef struct Node {      int value;      struct Node\* left;      struct Node\* right;      struct Node\* parent;  } Node;  // Function to create a new node  Node\* createNode(int value) {      Node\* newNode = (Node\*)malloc(sizeof(Node));      newNode->value = value;      newNode->left = NULL;      newNode->right = NULL;      newNode->parent = NULL;      return newNode;  }  // Function to insert a node into a BST  Node\* insertNode(Node\* root, int value) {      if (root == NULL) {          return createNode(value);      }      if (value < root->value) {          root->left = insertNode(root->left, value);          root->left->parent = root;      } else if (value > root->value) {          root->right = insertNode(root->right, value);          root->right->parent = root;      }      return root;  }  // Function to find the minimum node in a BST  Node\* findMin(Node\* node) {      while (node->left != NULL) {          node = node->left;      }      return node;  }  // Function to delete a node from a BST  Node\* deleteNode(Node\* root, int value) {      if (root == NULL) {          return NULL;      }      if (value < root->value) {          root->left = deleteNode(root->left, value);      } else if (value > root->value) {          root->right = deleteNode(root->right, value);      } else {          if (root->left == NULL) {              Node\* temp = root->right;              free(root);              return temp;          } else if (root->right == NULL) {              Node\* temp = root->left;              free(root);              return temp;          }          Node\* temp = findMin(root->right);          root->value = temp->value;          root->right = deleteNode(root->right, temp->value);      }      return root;  }  // Function to search for a value in a BST  Node\* searchNode(Node\* root, int value) {      if (root == NULL || root->value == value) {          return root;      }      if (root->value < value) {          return searchNode(root->right, value);      } else {          return searchNode(root->left, value);      }  }  // Function to find the minimum value in a BST  int findMinValue(Node\* root) {      while (root->left != NULL) {          root = root->left;      }      return root->value;  }  // Function to find the maximum value in a BST  int findMaxValue(Node\* root) {      while (root->right != NULL) {          root = root->right;      }      return root->value;  }  // Function to perform an inorder traversal of a BST  void inorderTraversal(Node\* root) {      if (root == NULL) {          return;      }      inorderTraversal(root->left);      printf("%d ", root->value);      inorderTraversal(root->right);  }  // Function to perform a preorder traversal of a BST  void preorderTraversal(Node\* root) {      if (root == NULL) {          return;      }      printf("%d ", root->value);      preorderTraversal(root->left);      preorderTraversal(root->right);  }  // Function to perform a postorder traversal of a BST  void postorderTraversal(Node\* root) {      if (root == NULL) {          return;      }      postorderTraversal(root->left);      postorderTraversal(root->right);      printf("%d ", root->value);  }  // Function to perform a level order traversal of a BST using a queue  void levelOrderTraversal(Node\* root) {      if (root == NULL) {          return;      }      Node\* queue[100];      int front = 0, rear = 0;      queue[rear++] = root;      while (front < rear) {          Node\* node = queue[front++];          printf("%d ", node->value);          if (node->left != NULL) {              queue[rear++] = node->left;          }          if (node->right != NULL) {              queue[rear++] = node->right;          }      }  }  // Function to count the total number of nodes in a BST  int countNodes(Node\* root) {      if (root == NULL) {          return 0;      }      return 1 + countNodes(root->left) + countNodes(root->right);  }  int main() {      Node\* root = NULL;      root = insertNode(root, 5);      root = insertNode(root, 3);      root = insertNode(root, 7);      root = insertNode(root, 2);      root = insertNode(root, 4);      root = insertNode(root, 6);      root = insertNode(root, 8);      printf("Inorder Traversal: ");      inorderTraversal(root);      printf("\n");      printf("Preorder Traversal: ");      preorderTraversal(root);      printf("\n");      printf("Postorder Traversal: ");      postorderTraversal(root);      printf("\n");      printf("Level Order Traversal: ");      levelOrderTraversal(root);      printf("\n");      printf("Total number of nodes: %d\n", countNodes(root));      Node\* temp = searchNode(root, 4);      if (temp != NULL) {          printf("Found node with value %d\n", temp->value);      } else {          printf("Node with value %d not found\n", 4);      }      root = deleteNode(root, 7);      printf("Inorder Traversal after deletion: ");      inorderTraversal(root);      printf("\n");      return 0;  } |
| **Output Screenshots (if applicable)** | MENU ---  1 - Insert an element into tree  2 - Delete an element from the tree  3 - Inorder Traversal  4 - Preorder Traversal  5 - Postorder Traversal  6- Search node  7- Delete node  8 - Exit    Enter your choice : 1  Enter data of node to be inserted : 40    Enter your choice : 1  Enter data of node to be inserted : 20    Enter your choice : 1  Enter data of node to be inserted : 10    Enter your choice : 1  Enter data of node to be inserted : 30    Enter your choice : 1  Enter data of node to be inserted : 60    Enter your choice : 1  Enter data of node to be inserted : 80    Enter your choice : 1  Enter data of node to be inserted : 90    Enter your choice : 3  10 -> 20 -> 30 -> 40 -> 60 -> 80 -> 90 -> |
| **Conclusion** | Thus, we have studied and implemented Binary Search Tree and performed given operations on it. |
| **Post Lab Questions:** | * What is a binary search tree? * What is a self-balanced tree? * What is the Red-Black tree data structure? |