Create a class BST

|  |
| --- |
| class BST{  //properties  //operations  }; |

Properties of BST class

|  |  |
| --- | --- |
| class BST{  //properties     |  | | --- | | //create node structure  struct Node{  int data;  Node\* left;  Node\* right;  Node(int val=0){ data=val; left=right=nullptr;}  }  //root pointer of type node  Node\* root; |   //operations  }; |

Operations of BST

|  |
| --- |
| //constructor  BST(){  root=nullptr;  } |

1. Add a new node in BST (Iteration)

|  |
| --- |
| void insertNode(int newData) {  Node\* newNode = new Node(newData);    if (root == NULL) {  root = newNode;  return;  }  Node\* temp = root;    while (temp != NULL) {  if (newNode->data == temp->data) {  return;  }  else if (newNode->data < temp->data && temp->left == NULL) {  temp->left = newNode;  break;  }  else if (newNode->data < temp->data) {  temp = temp->left;  }  else if (newNode->data > temp->data && temp->right == NULL) {  temp->right = newNode;  break;  }  else {  temp = temp->right;  }  }  } |

Another logic

|  |
| --- |
| void insertNewNode(Node\*& root, int val){  Node\* newNode=new Node(val);  if(root==nullptr){  root=newNode;  return;  }  if(root->data==val){  return;  }  Node\* temp=root;  while(true){  //left sub tree  if(val<temp->data){  if(temp->left!=nullptr)  temp=temp->left;  else{  temp->left=newNode; //newnode  break;  }  }  //right sub tree  else{  if(temp->right!=nullptr)  temp=temp->right;  else{  temp->right=newNode;  break;  }  }  }  } |

1. Add a new node in BST (Recursion)

//root node is pass by reference

|  |
| --- |
| //private method  void insertNode(Node\*& root, int val){  if(root ==nullptr){  root=new Node(val);  }  if(val==root->data){  return;  }  else if(val>root->data){  insertNode(root->right, val);  }  else{  insertNode(root->left, val);  }  }  //public method  void insertNode(int val){  insertNode(root, val);  } |

//root node pass by value

|  |
| --- |
| Node\* insertNode (Node\* root, int value) {  if (root == nullptr) {  return new Node(value);  }  if (value < root->data) {  root->left = insertNode (root->left, value);  } else {  root->right = insertNode (root->right, value);  }  return root;  } |

1. Search function (iteration)

|  |
| --- |
| //private method  Node\* searchNode(Node\* root, int key){  if(root==nullptr){  return root;  }  Node\* temp=root;  while(!temp){  if(key==temp->data){  return temp;  }  else if(key>temp->data){  temp=temp->right;  }  else{  temp=temp->left;  }  }  return temp;  }  //public method  void searchNode(int key){  Node\* foundNode=searchNode(root, key);  if(foundNode!=nullptr)  cout<<"Element found!!"<<endl;  else  cout<<"Element not found!"<<endl;  } |

1. Search function (Recursion)

|  |
| --- |
| //private method  bool searchRecursion(Node\* node, int key) {  if (node == nullptr) {  return false;  }  if (node->data == key) {  return true;  }  if (key < node->data) {  return searchRecursion(node->left, key);  }  return searchRecursion(node->right, key);  }  //public method  void searchRecursion(int key) {  bool flag= searchRecursion(root, key);  if(flag)  cout<<"Element found!!"<<endl;  else  cout<<"Element not found!!"<<endl;  } |

1. Tree traversal (Depth first search)
   1. Preorder Traversal

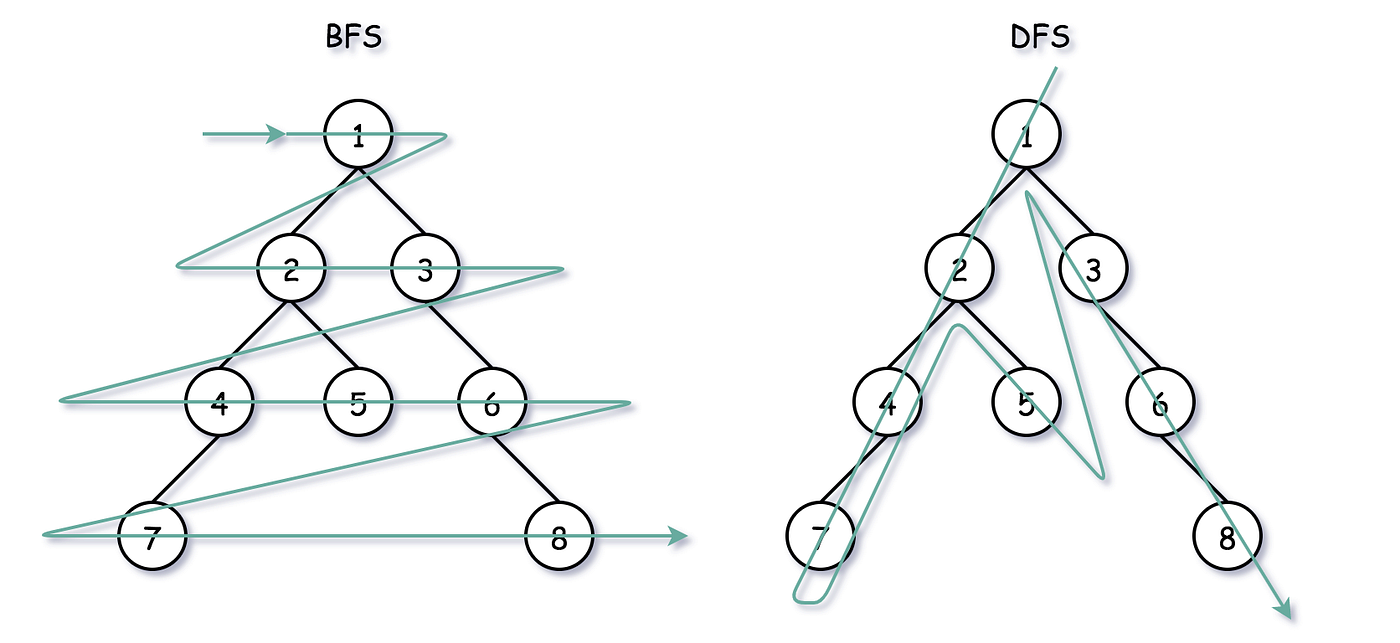
|  |
| --- |
| //private method  void preorder(Node\* root) {  if (root == nullptr) return;  cout << root->data << " "; // Visit the root  preorder(root->left); // Traverse left  preorder(root->right); // Traverse right  }  //public method  void preorder(){  preorder(root);  } |

* 1. Inorder Traversal

|  |
| --- |
| //private method  void inorder(Node\* root){  if(root==nullptr){  return;  }  inorder(root->left);  cout<<root->data<<" ",  inorder(root->right);  }  //public method  void inorder(){  inorder(root);  } |

* 1. Postorder Traversal

|  |
| --- |
| //private method  void postorder(Node\* root) {  if (root == nullptr) return;  postorder(root->left); // Traverse left  postorder(root->right); // Traverse right  cout << root->data << " "; // Visit the root  }  //public method  void postorder(){  postorder(root);  } |



1. Breadth First Search (BFS) /traversal function

|  |
| --- |
| //private methods  int height(Node\* root) {  if (root == nullptr) {  return -1;  } else {  int lheight = height(root->left);  int rheight = height(root->right);  return (lheight > rheight) ? lheight + 1 : rheight + 1;  }  }  void printGivenLevel(Node\* root, int level) {  if (root == nullptr) {  return;  } else if (level == 0) {  cout << root->data << " ";  } else {  printGivenLevel(root->left, level - 1);  printGivenLevel(root->right, level - 1);  }  }  void printLevelOrderBFS(Node\* root) {  int h = height(root);  for (int i = 0; i <= h; i++) {  printGivenLevel(root, i);  }  }  //public methods  void printLevelOrderBFS(){  cout<<"BFS"<<endl;  printLevelOrderBFS(root);  } |

1. Delete a node

|  |
| --- |
| //private method  Node\* minValueNode(Node\* node) {  Node\* current = node;  while (current && current->left != nullptr) {  current = current->left;  }  return current;  }  Node\* deleteNode(Node\* root, int value) {  if (root == nullptr) {  return root;  }  if (value < root->data) {  root->left = deleteNode(root->left, value);  }  else if (value > root->data) {  root->right = deleteNode(root->right, value);  }  else {  if (root->left == nullptr) {  Node\* temp = root->right;  delete root;  return temp;  }  else if (root->right == nullptr) {  Node\* temp = root->left;  delete root;  return temp;  }  else {  Node\* temp = minValueNode(root->right);  root->data = temp->data;  root->right = deleteNode(root->right, temp->data);  }  }  return root;  }  //public method  void deleteNode(int key){  Node\* n = deleteNode(root, key);  if(n)  cout<<"Node deleted !"<<endl;  else  cout<<"Node doesn't exist in tree hence not deleted !!"<<endl;  } |