1. Traverse the binary tree given above in pre, post and inorder.
   1. Preorder Traversal:
   2. Post Traversal:
   3. In-order Traversal:

|  |
| --- |
|  |

1. Draw the expression tree of the given algebraic expression and traverse the tree in pre, post and inorder.



|  |
| --- |
|  |

1. Complete the given class to implement a binary search tree

1

class

Node

2

{

3

public

:

Node

\*

left;

Node

\*

right;

int

data;

4

}

;

5

class

bst

6

{

Node

\*

root;

7

public

:

bst();

8

bool

isempty

()

;

9

void

insert

(

int

item);

10

bool

search

(

int

item);

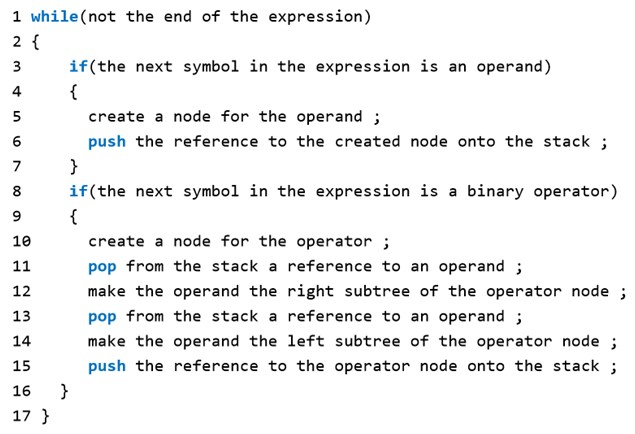
}

;

|  |
| --- |
| #include<iostream>  #include<conio.h>  using namespace std;  class Node  {  public:  int data;  Node\* left;  Node\* right;  };  class BST  {  private:  Node\* root;  public:  BST()  {  root = NULL;  }  Node\* getRoot()  {  return root;  }  void insertNode(int data)  {  Node\* next = new Node;  next->data = data;  next->left = NULL;  next->right = NULL;  if (root == NULL)  {  root = next;  }  else  {  Node\* parent = root;  Node\* current = root;    while (current != NULL)  {  if (data < current->data)  {  parent = current;  current = current->left;  }  else if (data > current->data)  {  parent = current;  current = current->right;  }  else if (data == current->data)  {  cout << "Node already exists..." << endl;  return;  }  }  if (data < parent->data)  {  parent->left = next;  }  if (data > parent->data)  {  parent->right = next;  }  }  }  bool isEmpty() {  if (root==NULL)  {  return true;  }  return false;  }  bool search(int data)  {  Node\* p = root;  while (p != NULL)  {  if (data < p->data)  {  p = p->left;  }  else if (data > p->data)  {  p = p->right;  }  else if (p->data == data)  {  return true;  }  }  return false;  }  }; |

1. Extend the above BST class to include functions for pre, post and in-order traversal of the tree. Use recursion to implement the traversal functions.

|  |
| --- |
| #include<iostream>  #include<conio.h>  using namespace std;  class Node  {  public:  int data;  Node\* left;  Node\* right;  };  class BST  {  private:  Node\* root;  public:  BST()  {  root = NULL;  }  Node\* getRoot()  {  return root;  }  void insertNode(int data)  {  Node\* next = new Node;  next->data = data;  next->left = NULL;  next->right = NULL;  if (root == NULL)  {  root = next;  }  else  {  Node\* parent = root;  Node\* current = root;    while (current != NULL)  {  if (data < current->data)  {  parent = current;  current = current->left;  }  else if (data > current->data)  {  parent = current;  current = current->right;  }  else if (data == current->data)  {  cout << "Node already exists..." << endl;  return;  }  }  if (data < parent->data)  {  parent->left = next;  }  if (data > parent->data)  {  parent->right = next;  }  }  }  void preOrder(Node\* root)  {  if (root == NULL)  {  return;  }  else  {  cout << " " << root->data;  preOrder(root->left);  preOrder(root->right);  }  }  void inOrder(Node\* root)  {  if (root == NULL)  {  return;  }  else  {  inOrder(root->left);  cout << " " << root->data;  inOrder(root->right);  }  }  void postOrder(Node\* root)  {  if (root == NULL)  {  return;  }  else  {  postOrder(root->left);  postOrder(root->right);  cout << " " << root->data;  }  }  bool isEmpty() {  if (root==NULL)  {  return true;  }  return false;  }  bool search(int data)  {  Node\* p = root;  while (p != NULL)  {  if (data < p->data)  {  p = p->left;  }  else if (data > p->data)  {  p = p->right;  }  else if (p->data == data)  {  return true;  }  }  return false;  }  };  int main()  {  BST obj;  obj.insertNode(14);  obj.insertNode(2);  obj.insertNode(11);  obj.insertNode(1);  obj.insertNode(3);  obj.insertNode(10);  obj.insertNode(30);  obj.insertNode(7);  obj.insertNode(40);  Node\* root = obj.getRoot();  cout << endl << "In-Order traversal is: ";  obj.inOrder(root);  cout << endl << "Pre-Order traversal is: ";  obj.preOrder(root);  cout << endl << "Post-Order traversal is: ";  obj.postOrder(root);  cout << endl << obj.search(402) << endl;  \_getch();  return 0;  } |
|  |

9. Write a program to convert PostFix into Expression Tree. Algorithm is given below:

|  |
| --- |
| class expNode  {  public:  char data;  expNode\* left;  expNode\* right;  };  void postfixToExpressionTree(string postfix)  {  int i = 0;  stack<expNode\*>s;  while (i<postfix.length())  {  if (postfix[i] == '/' || postfix[i] == '\*' || postfix[i] == '+' || postfix[i] == '-')  {  expNode\* p = new expNode;  p->data = postfix[i];  expNode\* q = s.top();  p->right = q;  s.pop();  expNode\* r = s.top();  p->left = r;  s.pop();  s.push(p);  }  else if (postfix[i] >= 47 || postfix[i] <= 55)  {  expNode\* p = new expNode;  p->data = postfix[i];  p->left = NULL;  p->right = NULL;  s.push(p);  }  }  } |