Calculators Grow on Trees

Problem: Basic Calculator II

Previously, we made the Calculator that converted Infix Notation to Postfix Notation.

https://leetcode.com/problems/basic-calculator-ii/

Problem: Basic Calculator II

Previously, we made the Calculator that converted Infix Notation to Postfix Notation.

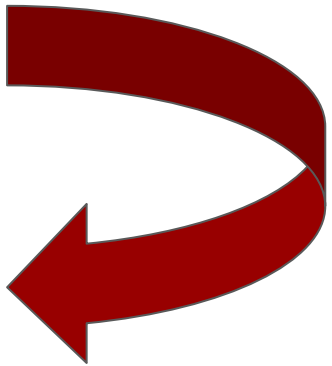
Infix: 5 + 8 - 2 * 10 / 2

https://leetcode.com/problems/basic-calculator-ii/

Problem: Basic Calculator II

Previously, we made the Calculator that converted Infix Notation to Postfix Notation.

Infix: 5 + 8 - 2 * 10 / 2



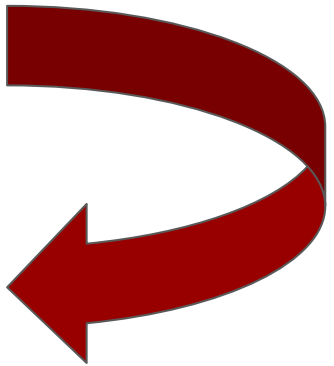
https://leetcode.com/problems/basic-calculator-ii/

Problem: Basic Calculator II

Previously, we made the Calculator that converted Infix Notation to Postfix Notation.

Infix: 5 + 8 - 2 * 10 / 2

Postfix: 5 8 + 2 10 * 2 / -



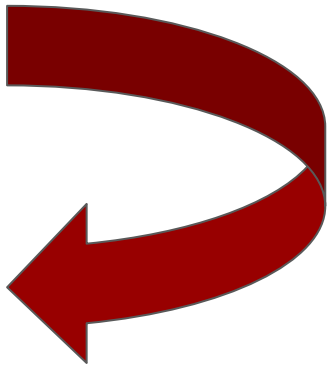
https://leetcode.com/problems/basic-calculator-ii/

Review: Basic Calculator II

Do you remember the name of the Algorithm that converted the expression from Infix to Postfix Notation?

Infix: 5 + 8 - 2 * 10 / 2

Postfix: 5 8 + 2 10 * 2 / -



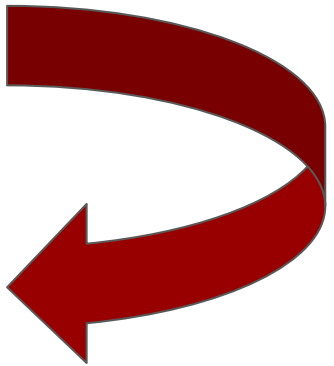
https://leetcode.com/problems/basic-calculator-ii/

Review: Shunting Yard Algorithm

Do you remember the name of the Algorithm that converted the expression from Infix to Postfix Notation?

Infix: 5 + 8 - 2 * 10 / 2

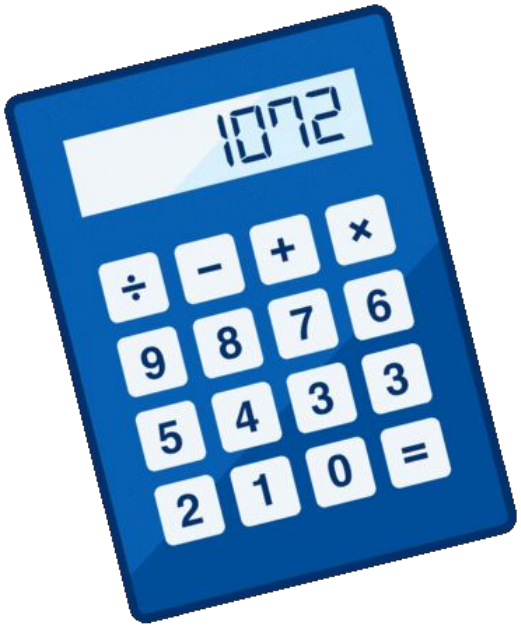
Postfix: 5 8 + 2 10 * 2 / -



https://leetcode.com/problems/basic-calculator-ii/

Problem: Advance Calculator

Now, we are going to add more functionalities in our Calculator.



Problem: Advance Calculator

We are going to create a Menu based system that will take the expression in Infix Notation as input.

```
-------Advance Calculator-------
Enter your Expression in Infix Notation
5+8-2*10/2
```

Problem: Advance Calculator

After that we will have the following options.

```
-------Advance Calculator-------
1. View the Expression in Infix Notation
2. View the Expression in Prefix Notation (Polish Notation)
3. View the Expression in Postfix Notation (Reverse Polish Notation)
4. Evaluate the Expression
5. Exit
Option ->
```

Problem: Advance Calculator

On pressing Option 1, it will show the expression in Infix notation with Parentheses.

```
-------Advance Calculator-------
1. View the Expression in Infix Notation
2. View the Expression in Prefix Notation (Polish Notation)
3. View the Expression in Postfix Notation (Reverse Polish Notation)
4. Evaluate the Expression
5. Exit
Option -> 1
( ( 5 + 8 )- ( ( 2 * 10 )/ 2 ))
Press Any Key to Continue
```

Problem: Advance Calculator

On pressing Option 2, it will show the expression in Prefix notation.

```
-------Advance Calculator-------
1. View the Expression in Infix Notation
2. View the Expression in Prefix Notation (Polish Notation)
3. View the Expression in Postfix Notation (Reverse Polish Notation)
4. Evaluate the Expression
5. Exit
Option -> 2
- + 5 8 / * 2 10 2
Press Any Key to Continue
```

Problem: Advance Calculator

On pressing Option 3, it will show the expression in Postfix notation.

```
-------Advance Calculator-------
1. View the Expression in Infix Notation
2. View the Expression in Prefix Notation (Polish Notation)
3. View the Expression in Postfix Notation (Reverse Polish Notation)
4. Evaluate the Expression
5. Exit
Option -> 3
5 8 + 2 10 * 2 / -
Press Any Key to Continue
```

Problem: Advance Calculator

On pressing Option 4, it will show the result of the expression after applying all the computations.

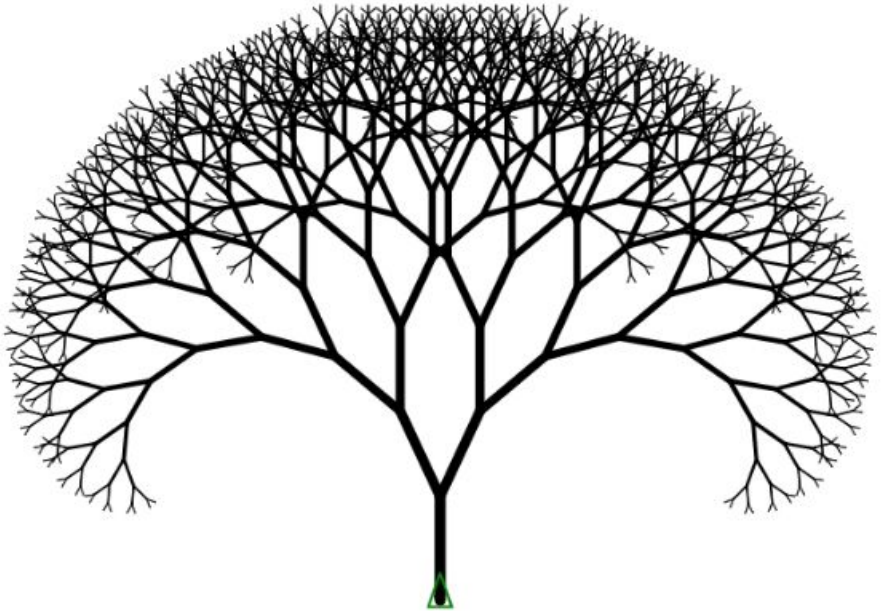
```
-------Advance Calculator-------
1. View the Expression in Infix Notation
2. View the Expression in Prefix Notation (Polish Notation)
3. View the Expression in Postfix Notation (Reverse Polish Notation)
4. Evaluate the Expression
5. Exit
Option -> 4
Result: 3
Press Any Key to Continue
```

Solution: Advance Calculator

Let's implement the solution in an efficient manner.

Solution: Advance Calculator

The most efficient method is through Binary Trees since most of the arithmetic operators are binary.

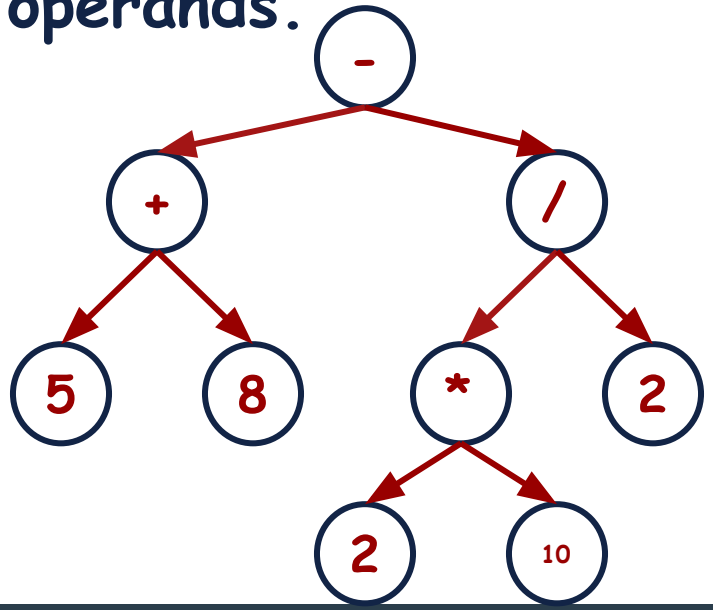


Solution: Advance Calculator

The specific order of the binary tree to be used is such that the node is an operator and its left and right children will be operands.

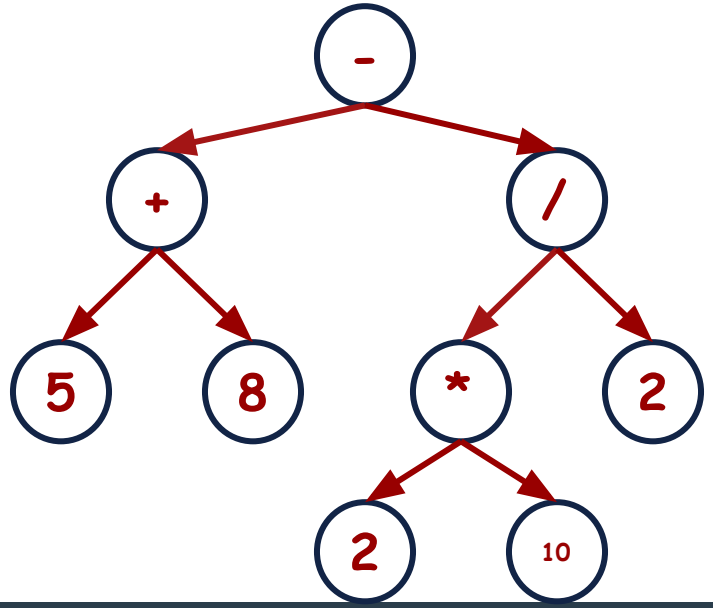
Solution: Advance Calculator

The specific order of the binary tree to be used is such that the node is an operator and its left and right children will be operands.



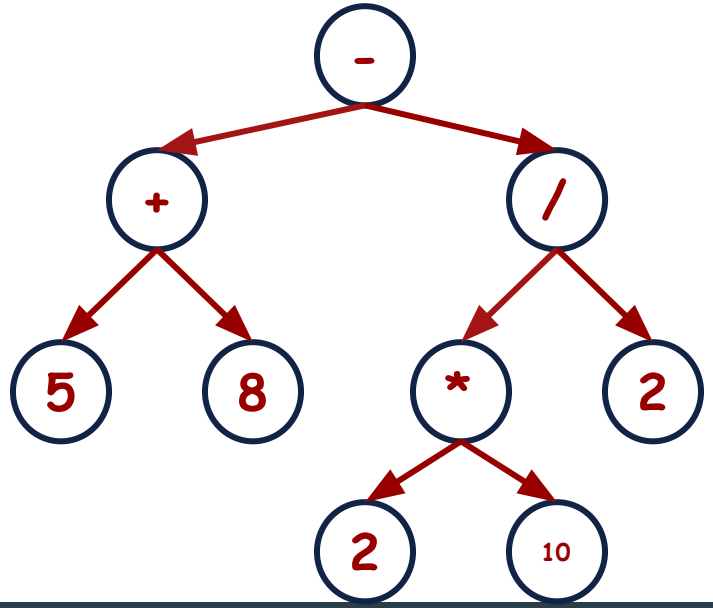
Expression Trees

This is a new Data Structure known as Expression Trees.



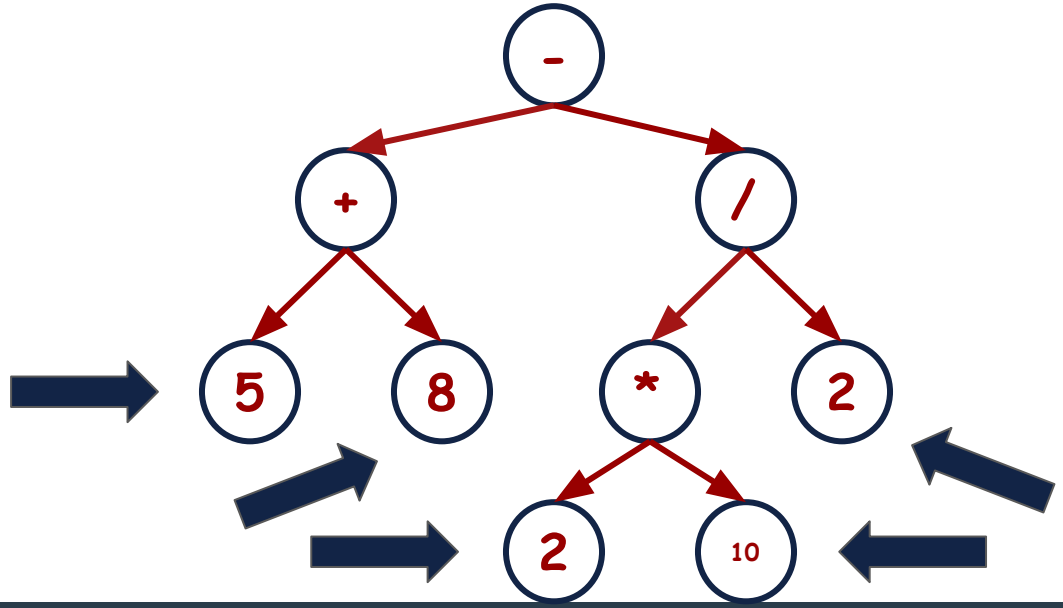
Expression Trees

In the Expression Tree, where are all the operands? Do you see a pattern?



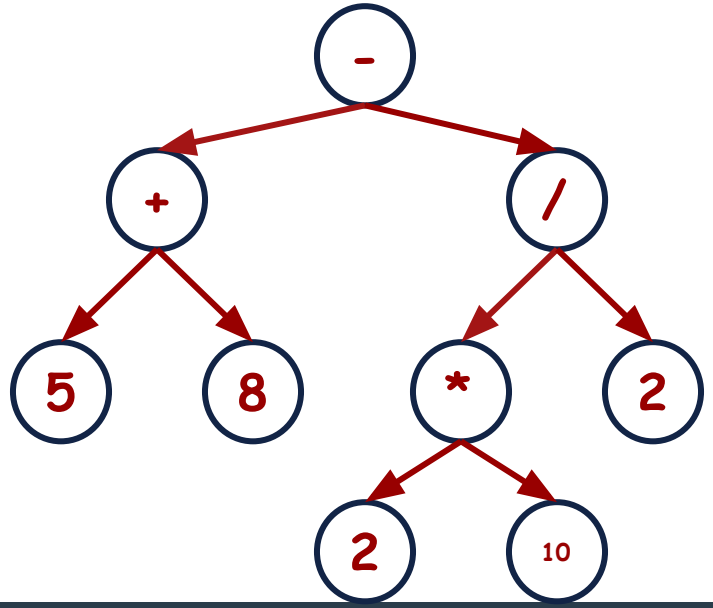
Expression Trees

All the operands are at the leaf nodes.



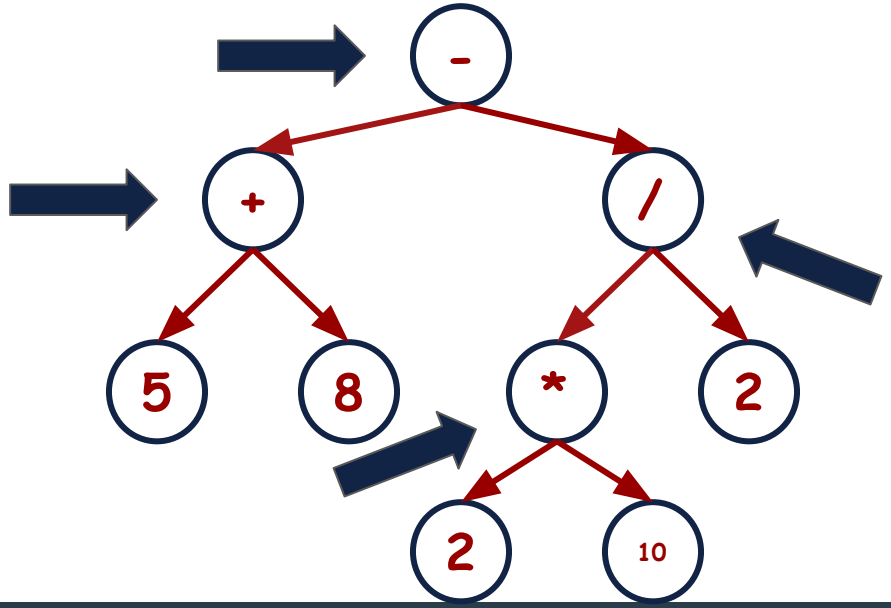
Expression Trees

In the Expression Tree, where are all the operators? Do you see a pattern?



Expression Trees

All the operators are at the inner nodes.



Expression Trees: Implementation

Mostly Postfix Notation is used to create the Expression trees.

5	8	+	2	10	*	2	/	-

Expression Trees:

5	8	+	2	10	*	2	/	-

Following Structure will be used to store the operands and operators in the TreeNode.

```
struct TreeNode
{
    string val;
    TreeNode *left;
    TreeNode *right;
};
```

Expression Trees:

5	8	+	2	10	*	2	/	-

Lets see the algorithm Step by Step.

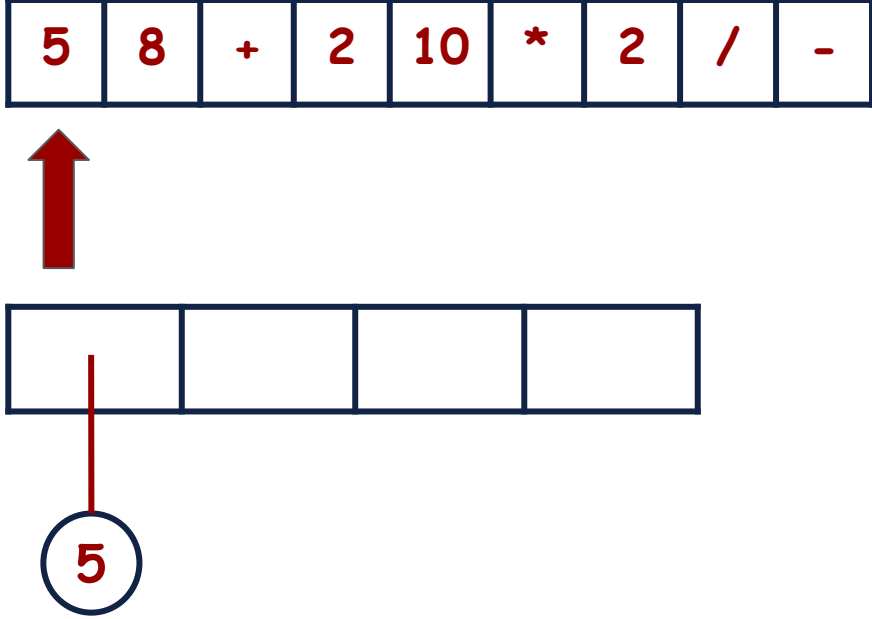
Expression Trees:

First of all, define a stack that can store the addresses of TreeNodes.

stack<TreeNode *>

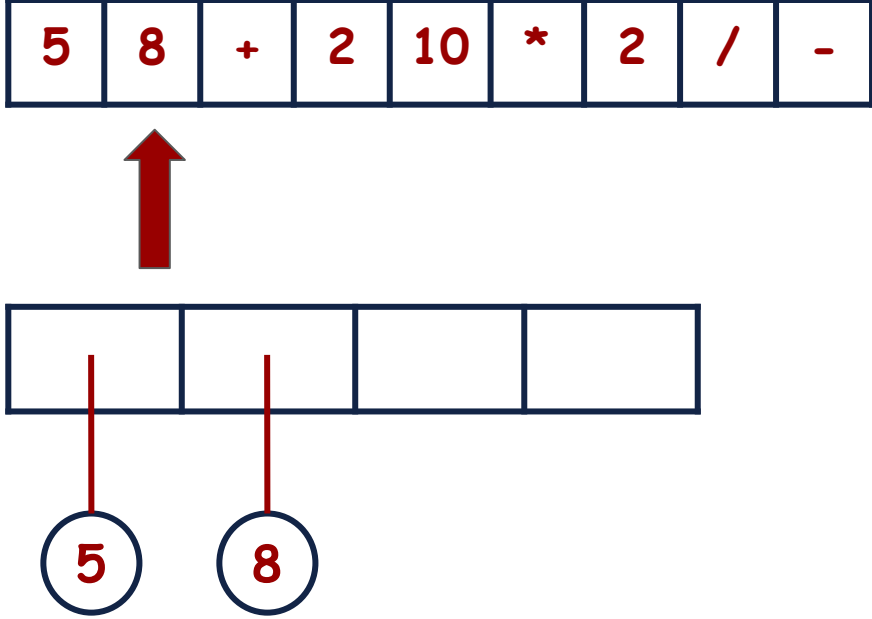
Expression Trees:

If the element in the vector is operand then create a TreeNode of that element with left and right pointers as NULL and push it onto the stack.



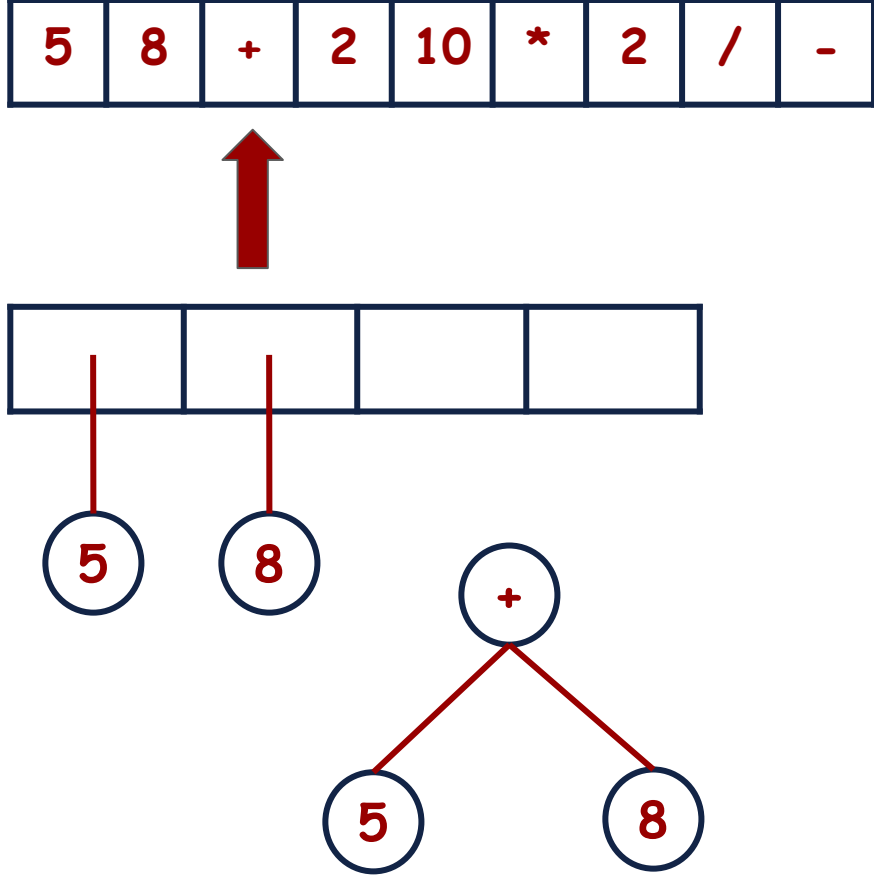
Expression Trees:

If the next element in the vector is operand then create a TreeNode of that element with left and right pointers as NULL and push it onto the stack.



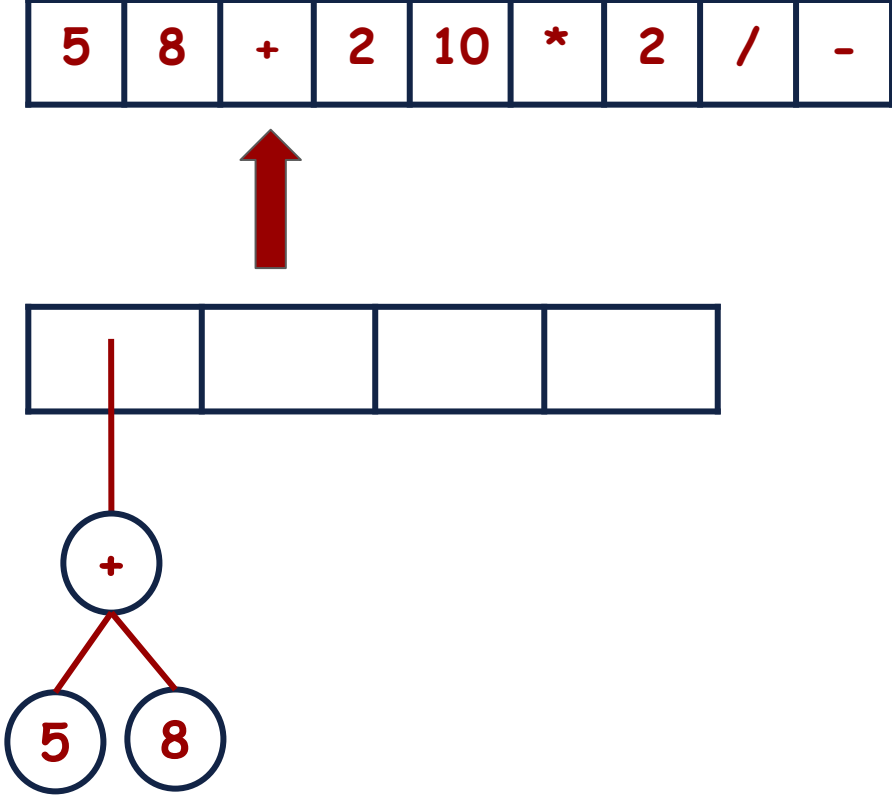
Expression Trees:

If the next element is operator then pop 2 TreeNodes from the stack, create a TreeNode of the operator with left and right pointers that popped from the stack.



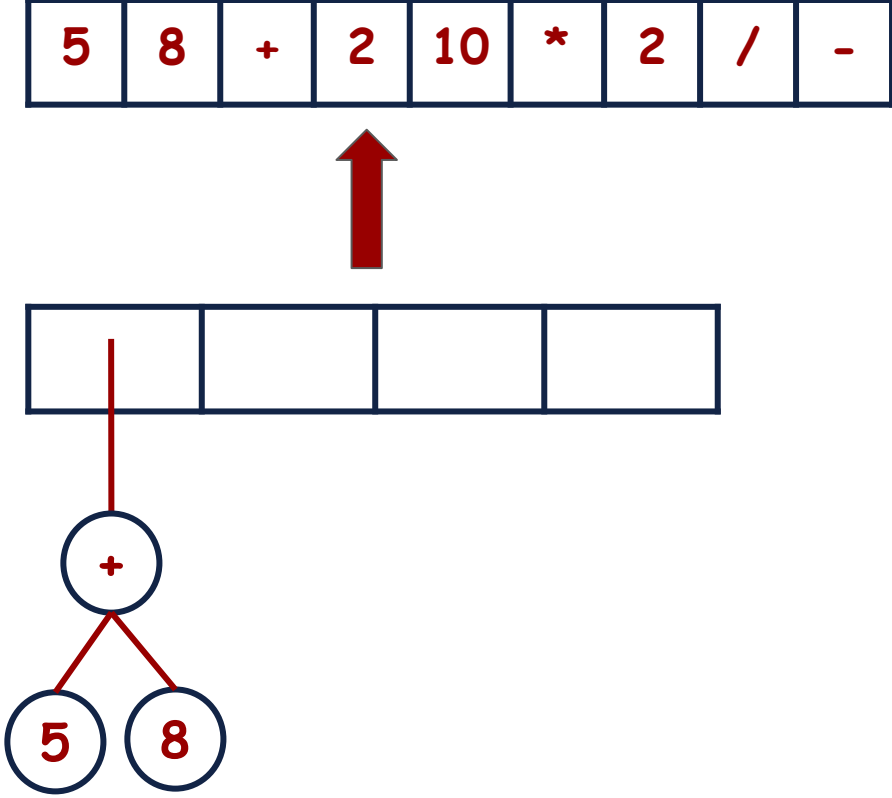
Expression Trees:

Then push the new node onto the stack.



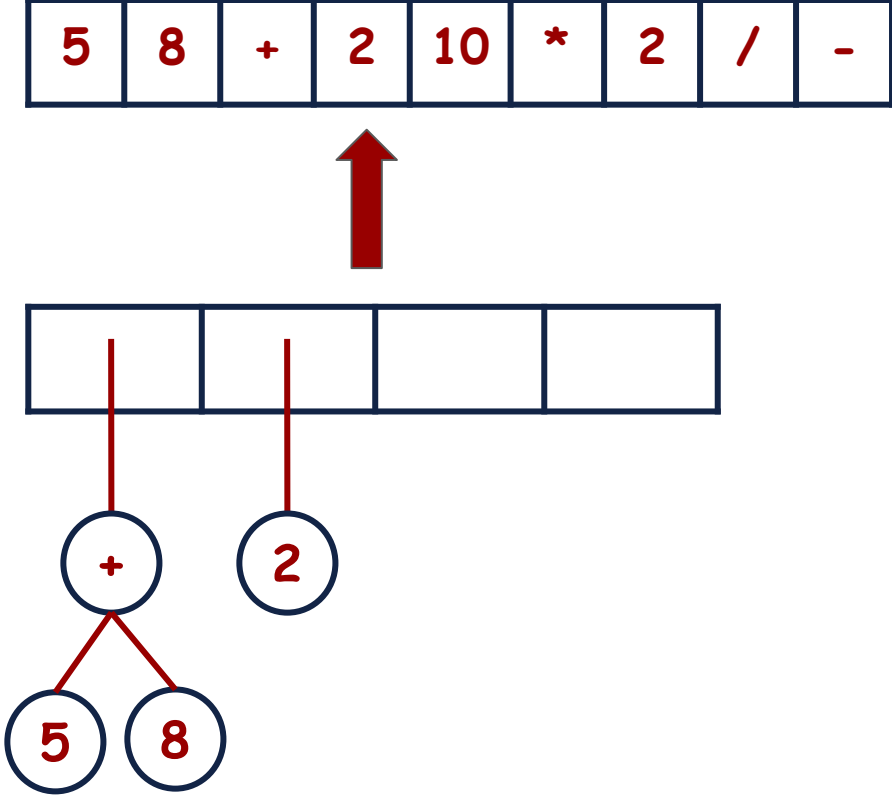
Expression Trees:

If the next element in the vector is operand then create a TreeNode of that element with left and right pointers as NULL and push it onto the stack.



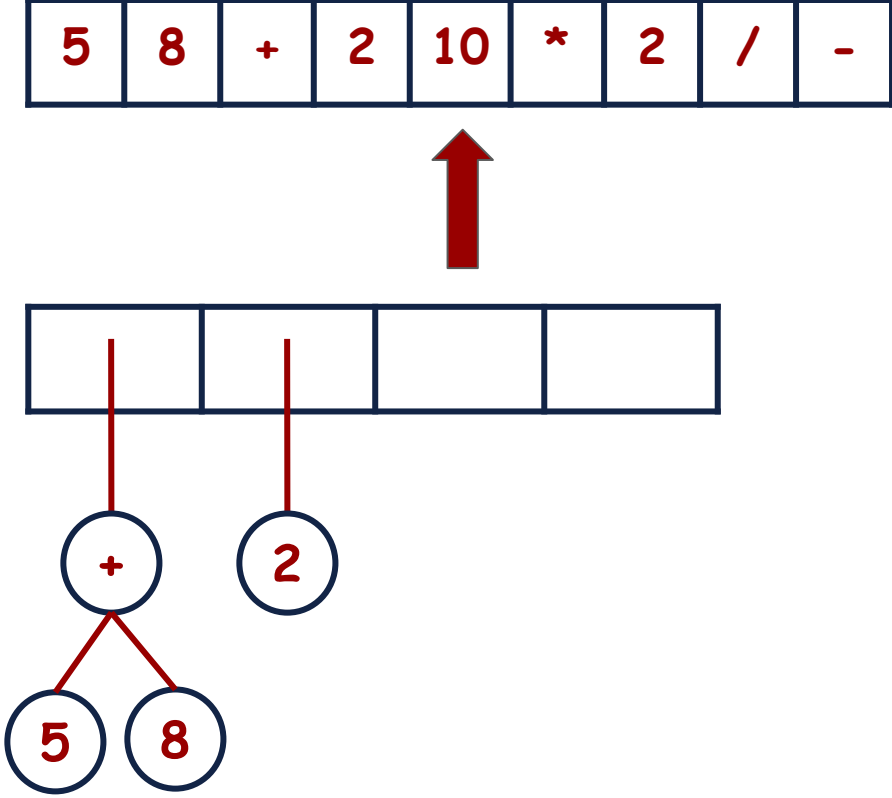
Expression Trees:

If the next element in the vector is operand then create a TreeNode of that element with left and right pointers as NULL and push it onto the stack.



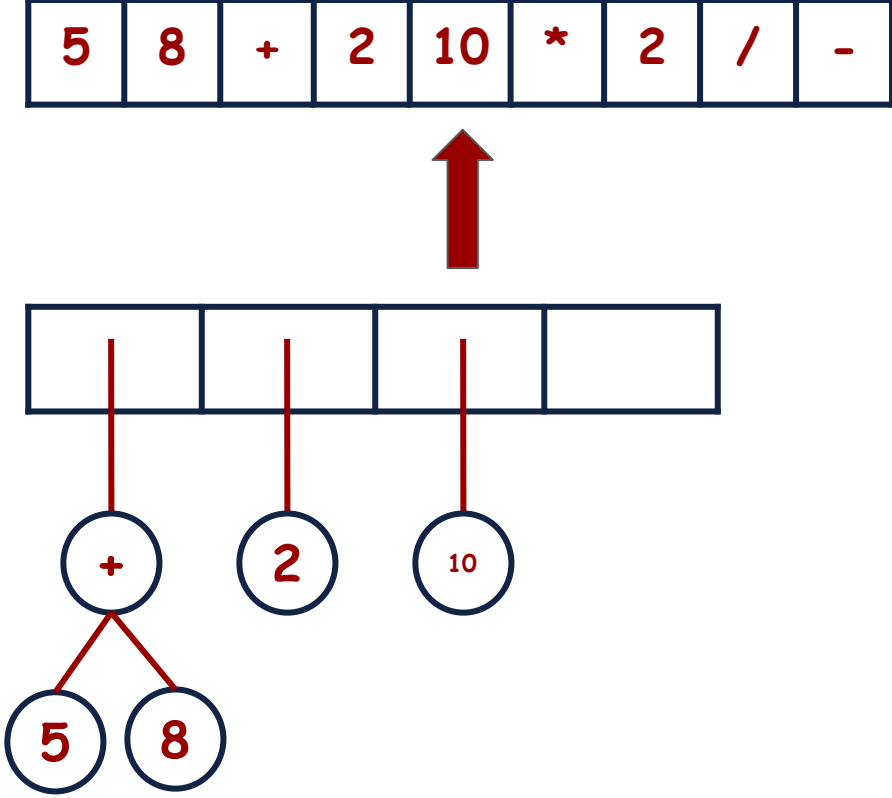
Expression Trees:

If the next element in the vector is operand then create a TreeNode of that element with left and right pointers as NULL and push it onto the stack.



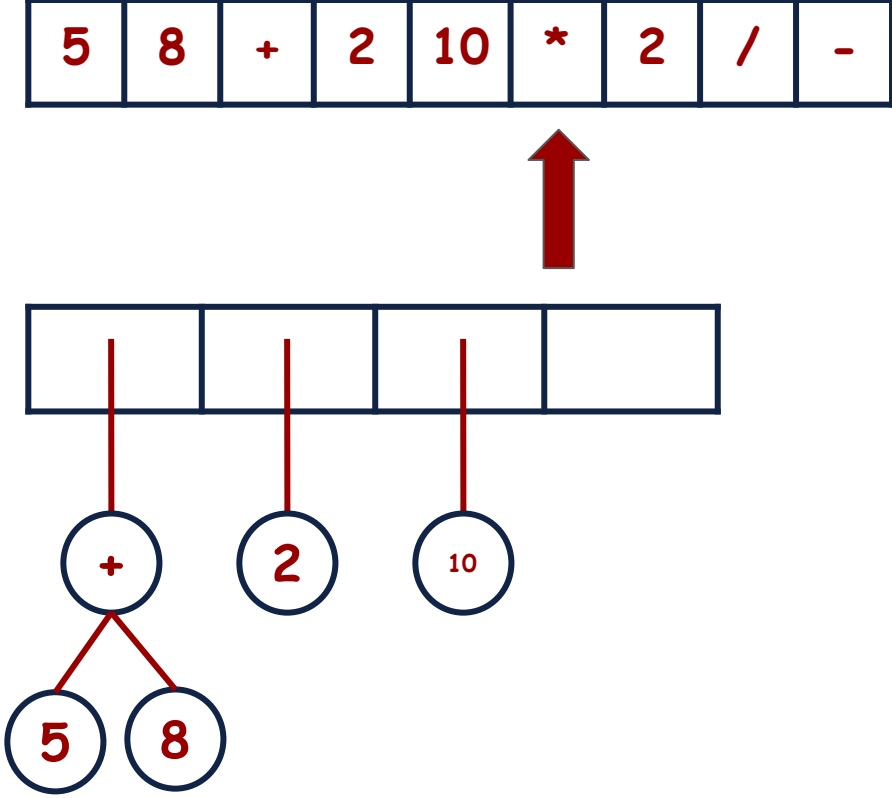
Expression Trees:

If the next element in the vector is operand then create a TreeNode of that element with left and right pointers as NULL and push it onto the stack.



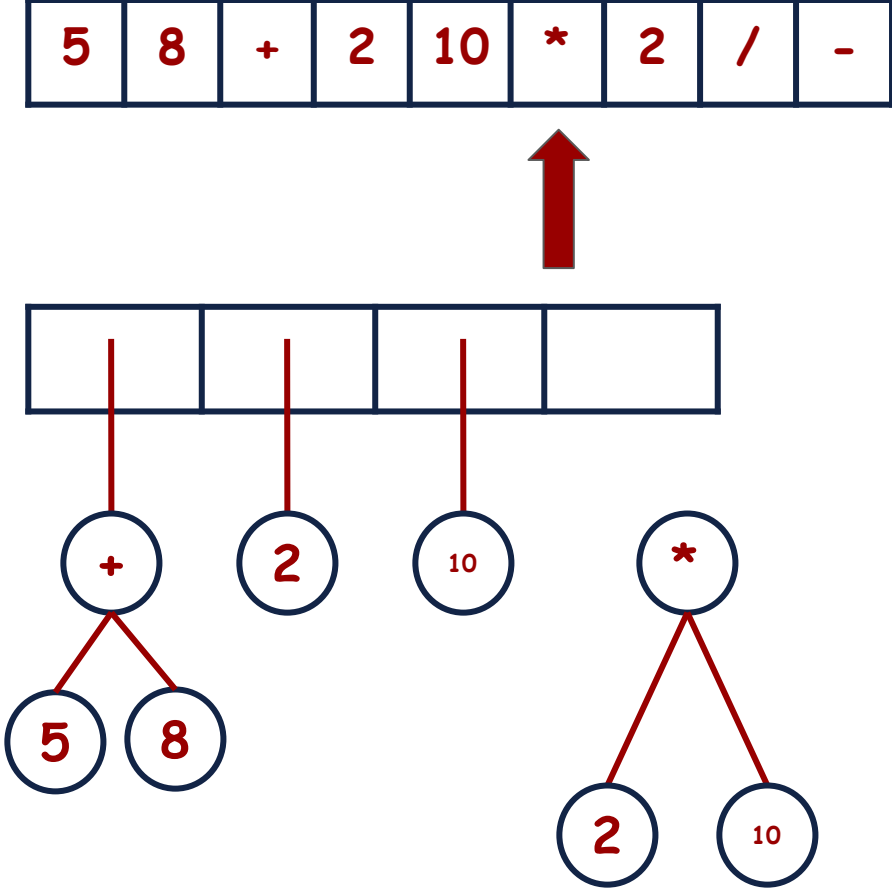
Expression Trees:

If the next element is operator then pop 2 TreeNodes from the stack, create a TreeNode of the operator with left and right pointers that popped from the stack.



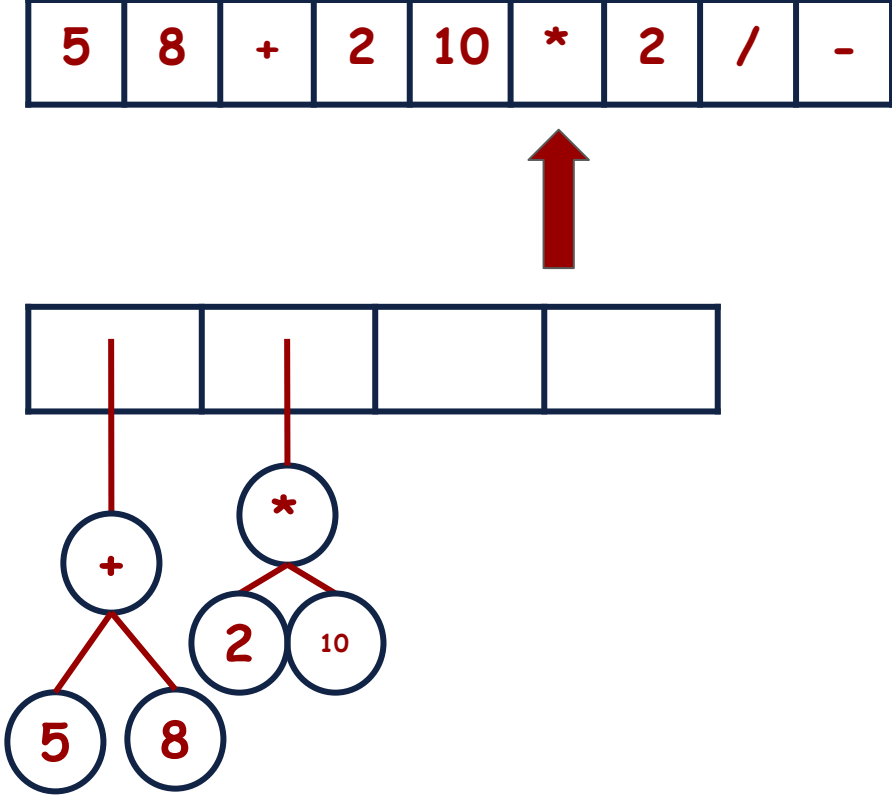
Expression Trees:

If the next element is operator then pop 2 TreeNodes from the stack, create a TreeNode of the operator with left and right pointers that popped from the stack.

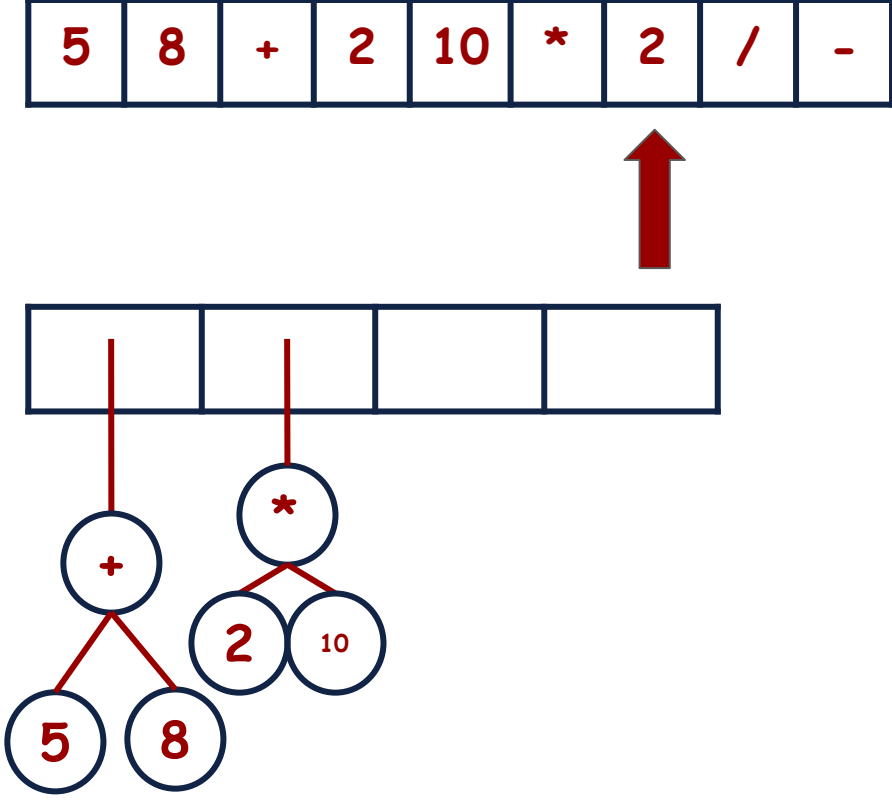


Expression Trees:

Then push the new node onto the stack.



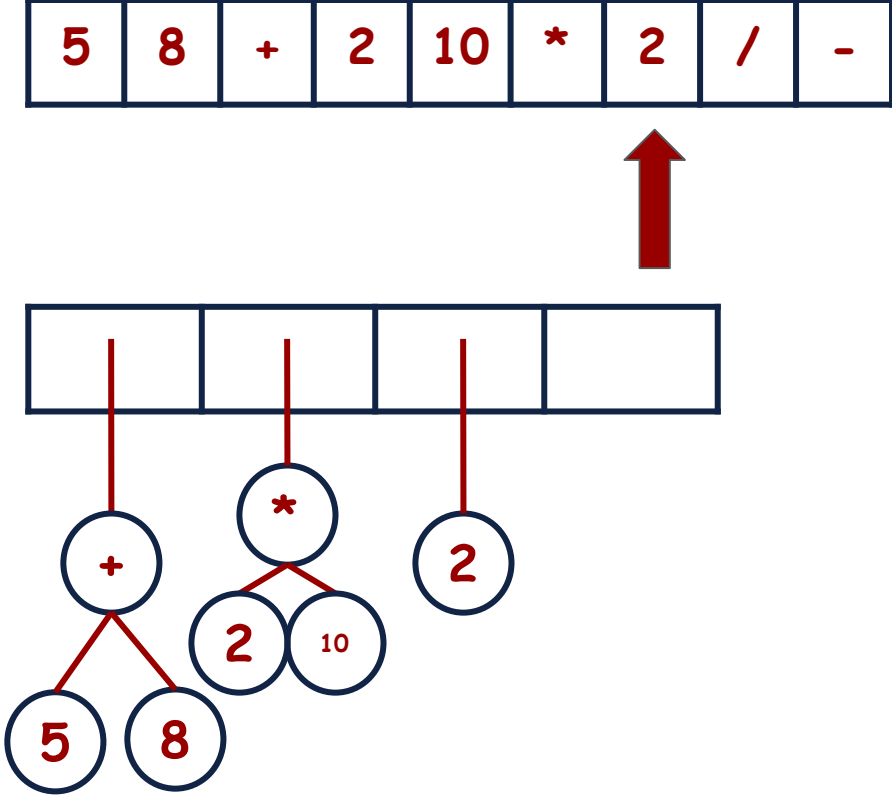
Expression Trees:



If the next element in the vector is operand then create a TreeNode of that element with left and right pointers as NULL and push it onto the stack.

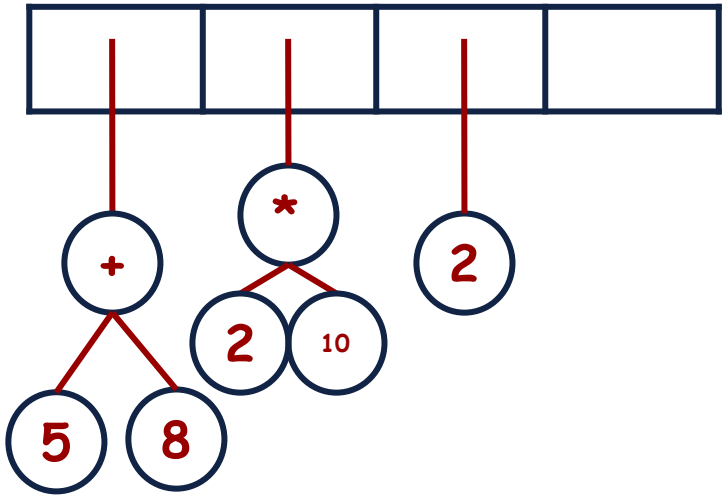
Expression Trees:

If the next element in the vector is operand then create a TreeNode of that element with left and right pointers as NULL and push it onto the stack.



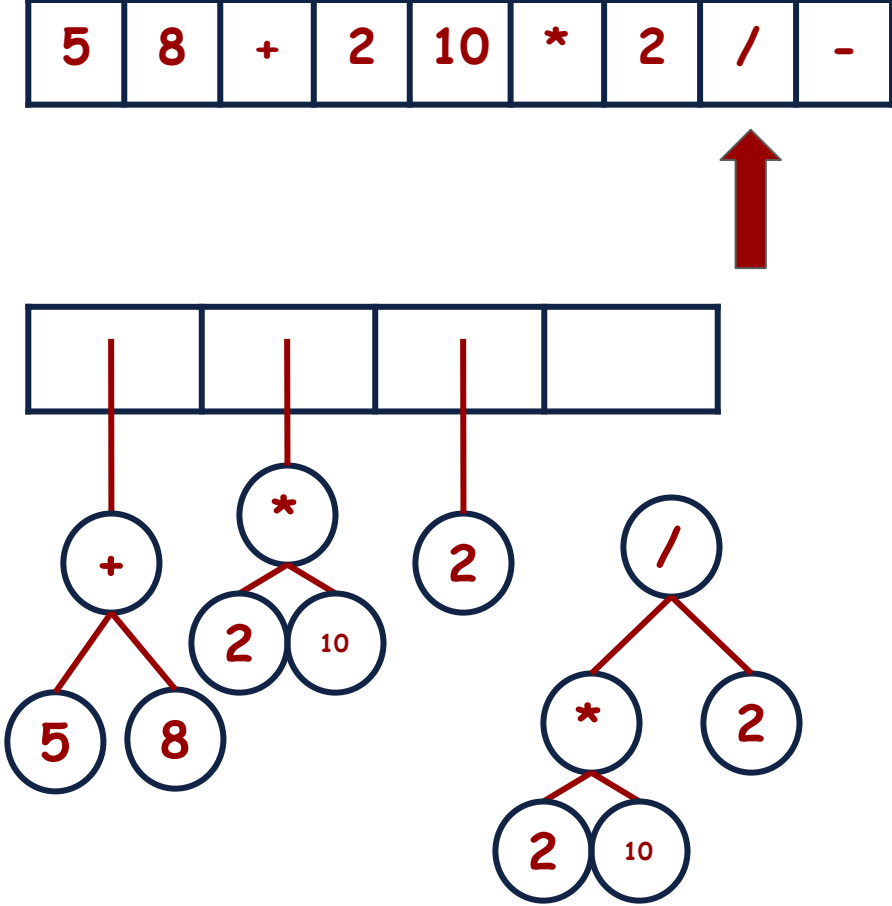
Expression Trees:

If the next element is operator then pop 2 TreeNodes from the stack, create a TreeNode of the operator with left and right pointers that popped from the stack.



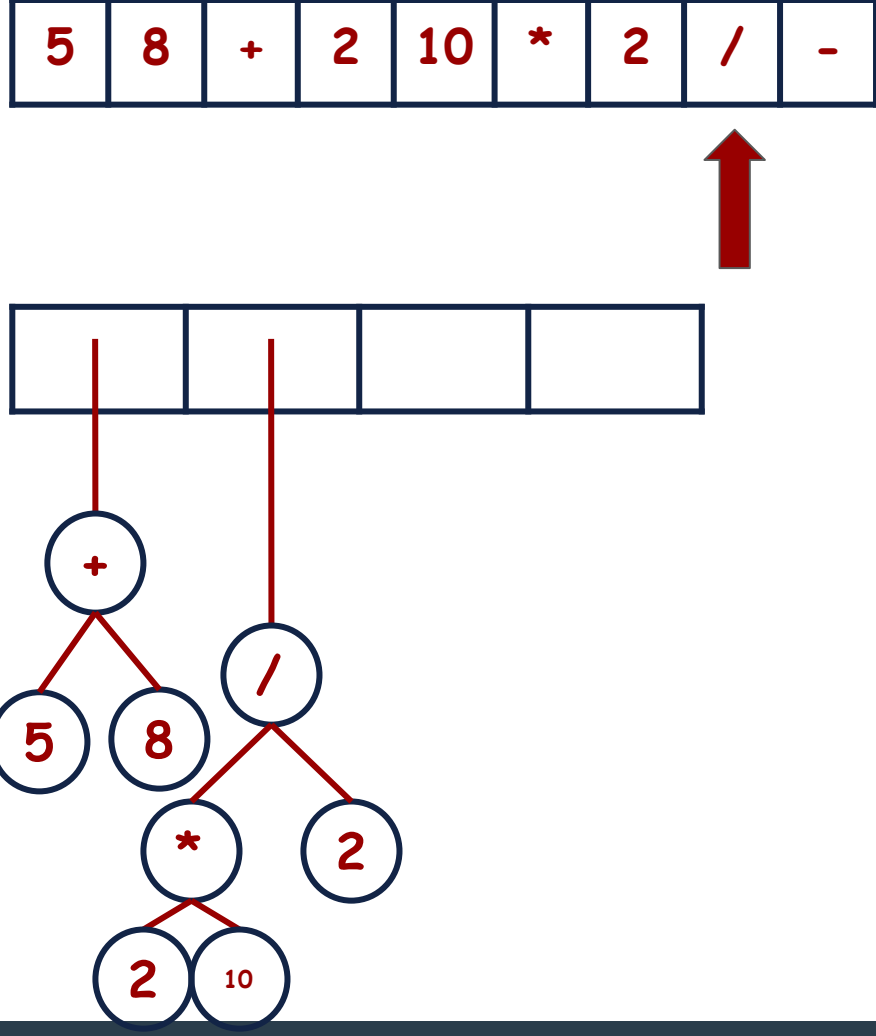
Expression Trees:

If the next element is operator then pop 2 TreeNodes from the stack, create a TreeNode of the operator with left and right pointers that popped from the stack.

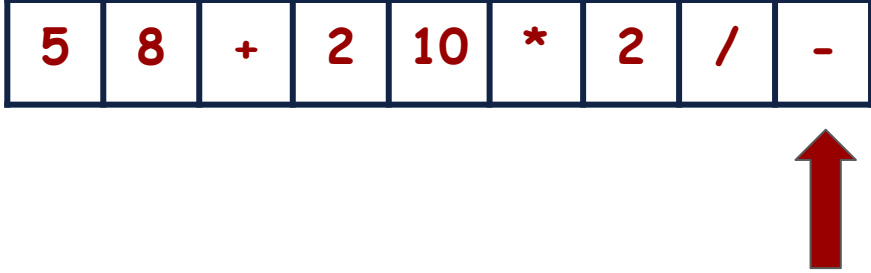


Expression Trees:

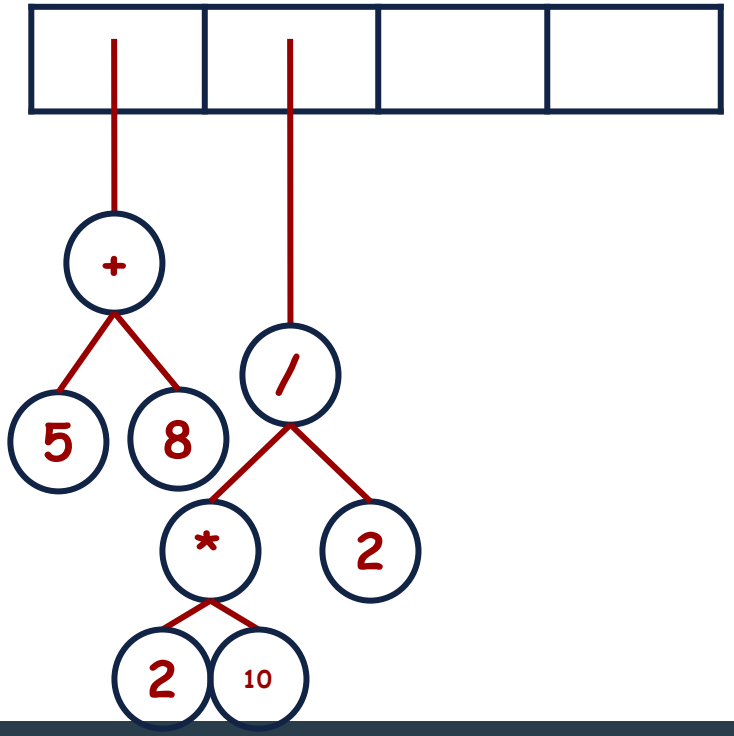
Then push the new node onto the stack.



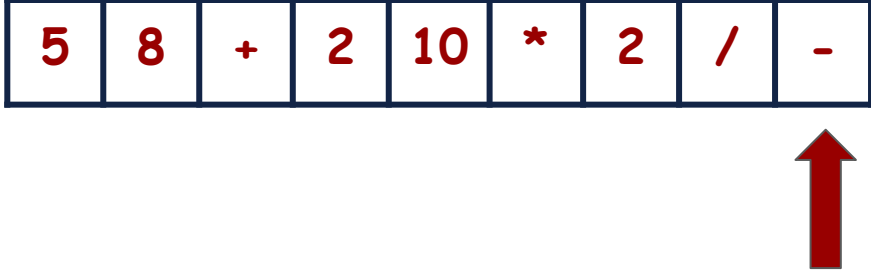
Expression Trees:



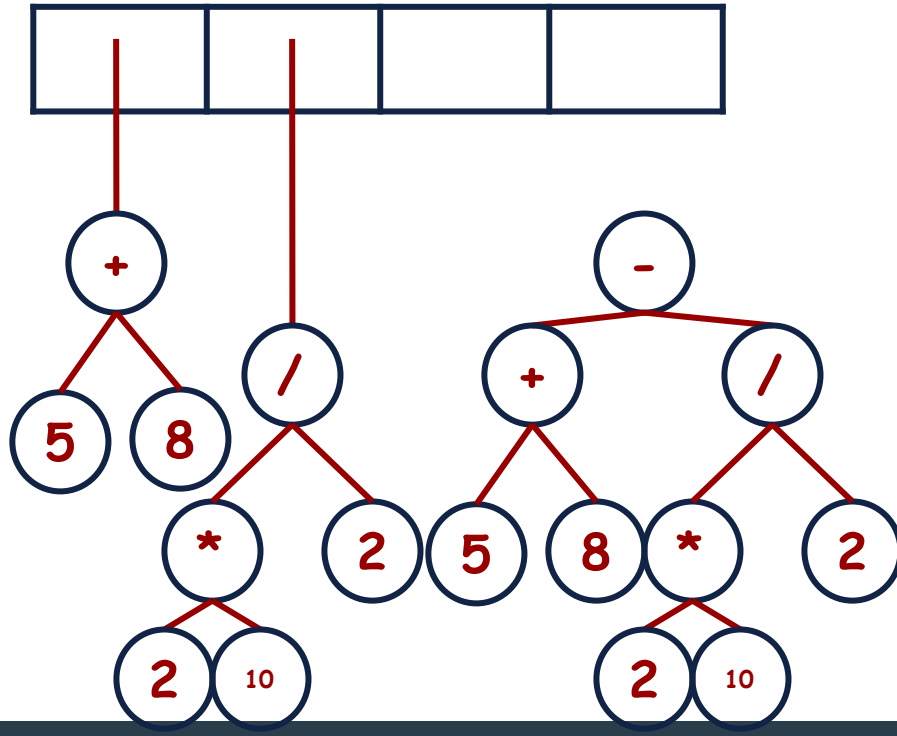
If the next element is operator then pop 2 TreeNodes from the stack, create a TreeNode of the operator with left and right pointers that popped from the stack.



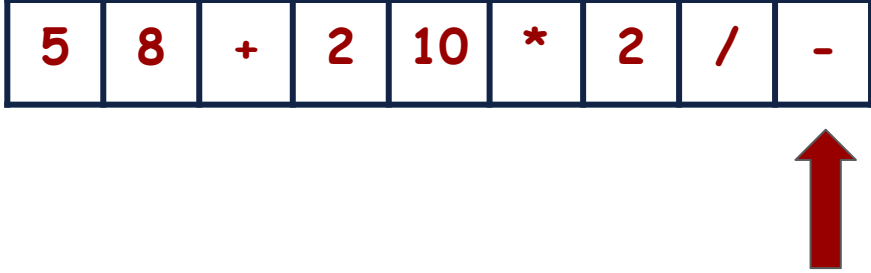
Expression Trees:



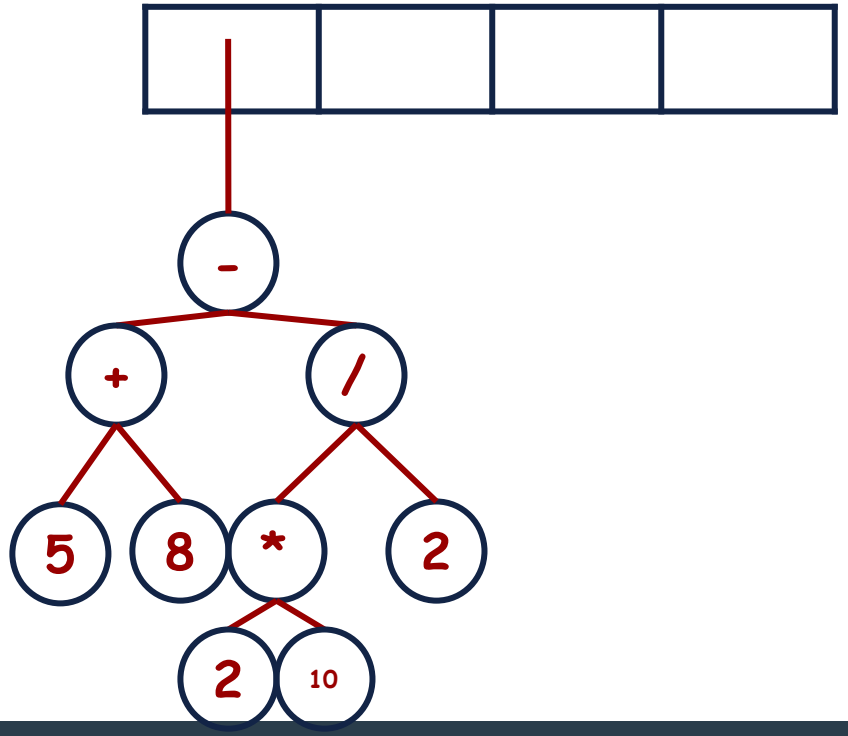
If the next element is operator then pop 2 TreeNodes from the stack, create a TreeNode of the operator with left and right pointers that popped from the stack.



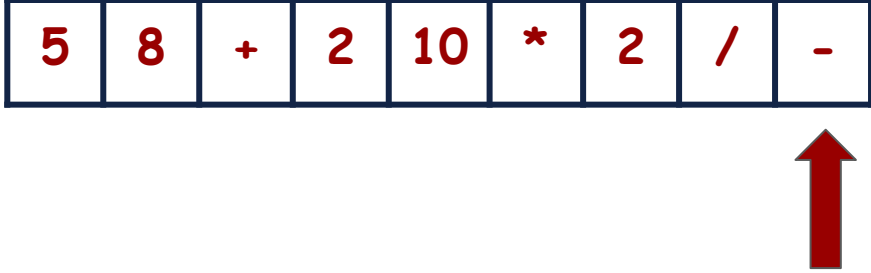
Expression Trees:



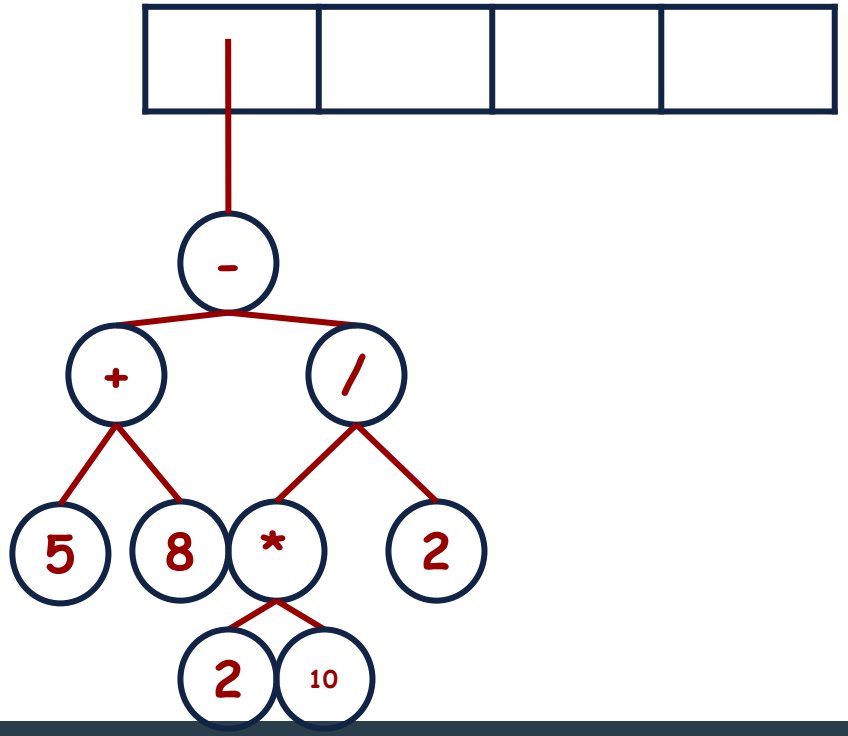
Then push the new node onto the stack.



Expression Trees:



Now the top element of the stack contains the root node of the Expression Tree.



Expression Trees: Implementation

```
class ExpressionTree
{
public:
  TreeNode *root;
  ExpressionTree()
  {
      root = NULL;
  }
  TreeNode *createNode(string val)
  {
     TreeNode *record = new TreeNode();
     record->val = val;
     record->left = NULL;
     record->right = NULL;
     return record;
  }
```

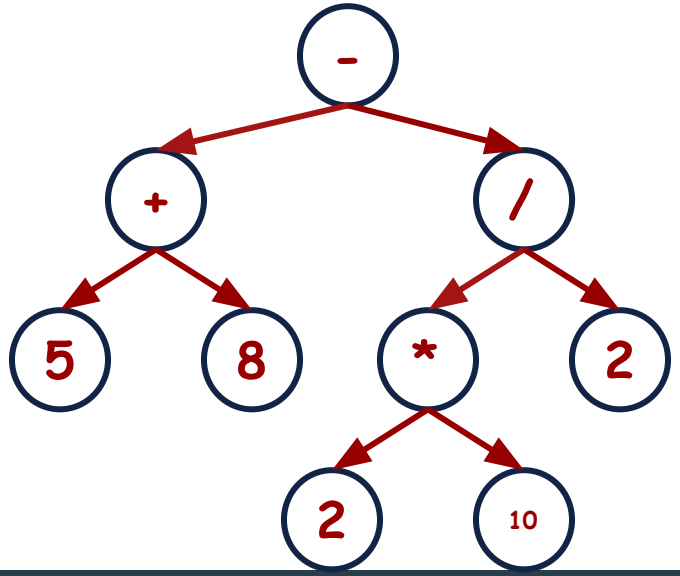
```
TreeNode *createNodeWithChildren(string
val, TreeNode *left, TreeNode *right)
{
     TreeNode *record = new TreeNode();
     record->val = val;
     record->left = left;
     record->right = right;
     return record;
}

bool isOperator(string value)
{
     if (value == "+" || value == "-" ||
value == "*" || value == "/")
          return true;
     return false;
}
```

Expression Trees: Implementation

```
void createExpressionTree(vector<string> expression){
    stack<TreeNode *> s;
    for (int x = 0; x < expression.size(); x++){
        if (isOperator(expression[x])){
            TreeNode *rNode = s.top();
            s.pop();
            TreeNode *lNode = s.top();
            s.pop();
            TreeNode *node = createNodeWithChildren(expression[x], lNode, rNode);
            s.push(node);
        }
        else{
            TreeNode *node = createNode(expression[x]);
            s.push(node);
        }
    }
    root = s.top();
}
```

Expression Trees

Once the Expression Tree is created, we can do Inorder traversal for Infix Notation, Preorder traversal for Prefix Notation and Postorder traversal for Postfix Notation



Expression Trees: Implementation

```
void inOrderTraversal(TreeNode *node)
    {
        if (node == NULL)
        {
            return;
        }
        if (isOperator(node->val))
         {
            cout << "( ";
         }
        inOrderTraversal(node->left);
        cout << node->val << " ";
        inOrderTraversal(node->right);
        if (isOperator(node->val))
         {
            cout << ")";
         }
    }
```

Expression Trees: Implementation

```
void preOrderTraversal(TreeNode *node)
    {
        if (node == NULL)
        {
            return;
        }
        cout << node->val << " ";
        preOrderTraversal(node->left);
        preOrderTraversal(node->right);
    }
```

Expression Trees: Implementation

```
void postOrderTraversal(TreeNode *node)
    {
        if (node == NULL)
        {
            return;
        }
        postOrderTraversal(node->left);
        postOrderTraversal(node->right);
        cout << node->val << " ";
    }
```

Learning Objective

Students should be able to create **Expression Trees** for solving the expressions effectively.

Self Assessment 01

Create an Expression Tree from the following Expression.

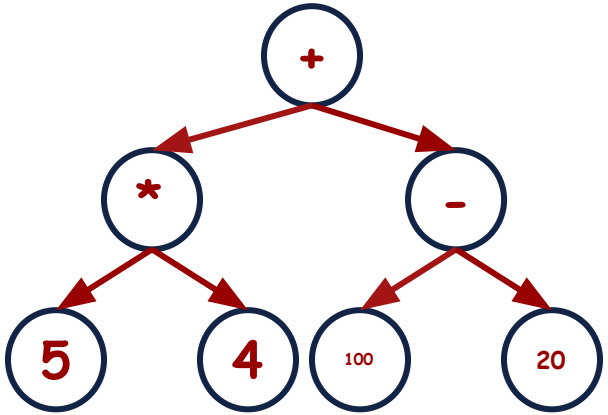
2 * 3 / (2-1) + 5 * (4-1)

Self Assessment 02

Given a full binary expression tree consisting of basic binary operators (+ , - ,*, /) and some integers, Your task is to evaluate the expression tree.

Self Assessment 02

Input:

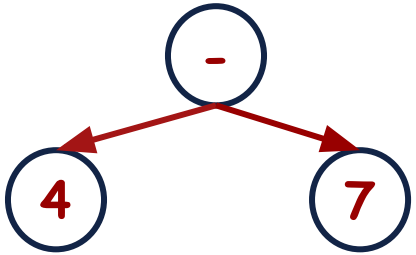


Output: 100

Explanation:
((5 * 4) + (100 - 20)) = 100

Self Assessment 02

Input:



Output: -3

Explanation:
4 - 7 = -3

Self Assessment 03

A **binary expression tree** is a kind of binary tree used to represent arithmetic expressions. Each node of a binary expression tree has either zero or two children.
Leaf nodes (nodes with 0 children) correspond to operands (variables), and internal nodes (nodes with two children) correspond to the operators.
In this problem, we only consider the '+' operator (i.e. addition).
You are given the roots of two binary expression trees, **root1** and **root2**.
Return true if the two binary expression trees are equivalent. Otherwise, return false.

Two binary expression trees are equivalent if they evaluate to the same value regardless of what the variables are set to.

Self Assessment 03

Example 1:
Input: root1 = [x], root2 = [x]
Output: true

Example 2:
Input: root1 = [+,a,+,null,null,b,c], root2 = [+,+,b,c,a]
Output: true
Explaination: a + (b + c) == (b + c) + a

Example 3:
Input: root1 = [+,a,+,null,null,b,c], root2 = [+,+,b,d,a]
Output: false
Explaination: a + (b + c) != (b + d) + a

Self Assessment 03

Constraints:

- The number of nodes in both trees are equal, odd and, in the range [1, 4999].
- Node.val is '+' or a lower-case English letter.
- It's guaranteed that the tree given is a valid binary expression tree.