Threaded Binary and Expression Trees

Threaded Binary Tree

- Each node in BT has additional pointers, known as threads

- Threads link to it’s in-order predecessor and successor.

- These threads facilitate efficient in-order traversal by allowing easy navigation from one node to the next without the need for recursive function calls or a stack.

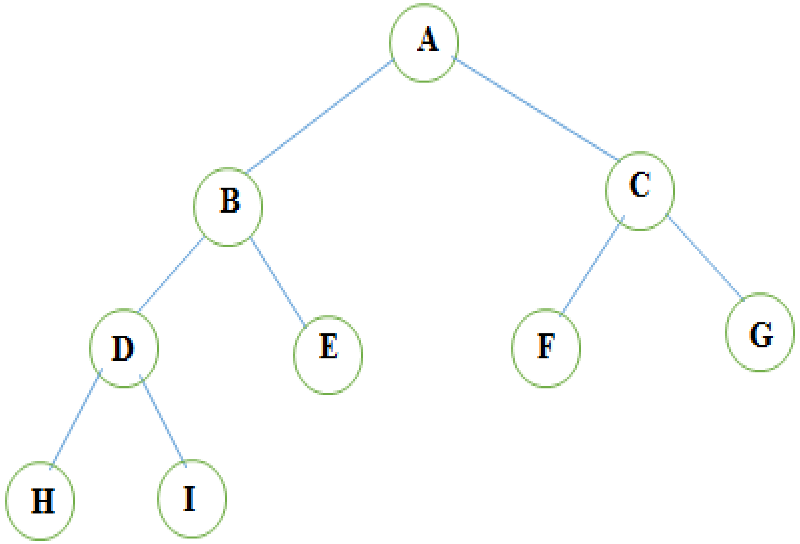
- Threaded binary trees optimize in-order traversal operations in scenarios where memory or stack usage needs to be minimized.

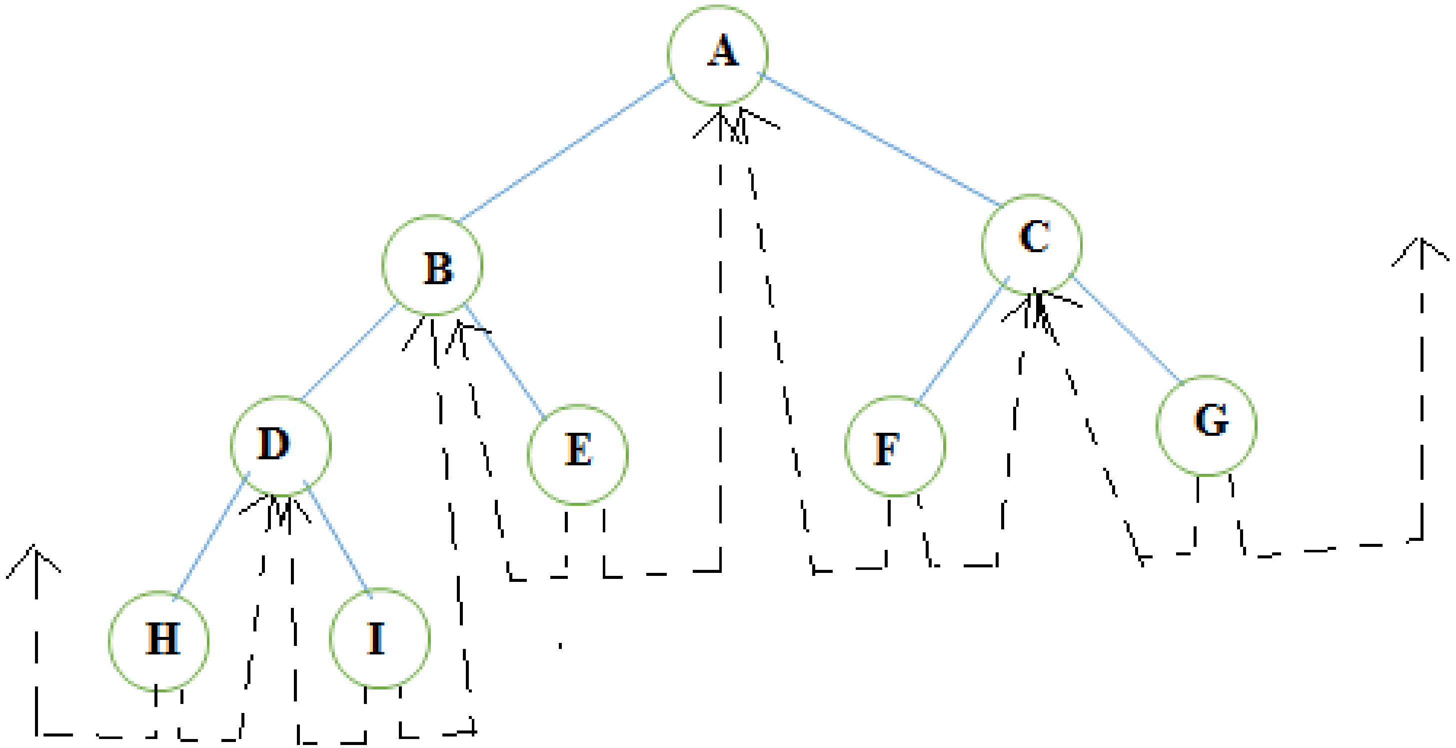
Rule 1: If root->lcl is null, replace it with a pointer to the inorder predecessor of Tree.

Rule 2: If root->rcl is null, replace it with a pointer to the inorder successor of Tree.

Rule 3: There must not be any loose threads. Therefore a threaded binary tree must have a head node of which the left child points to the first node.

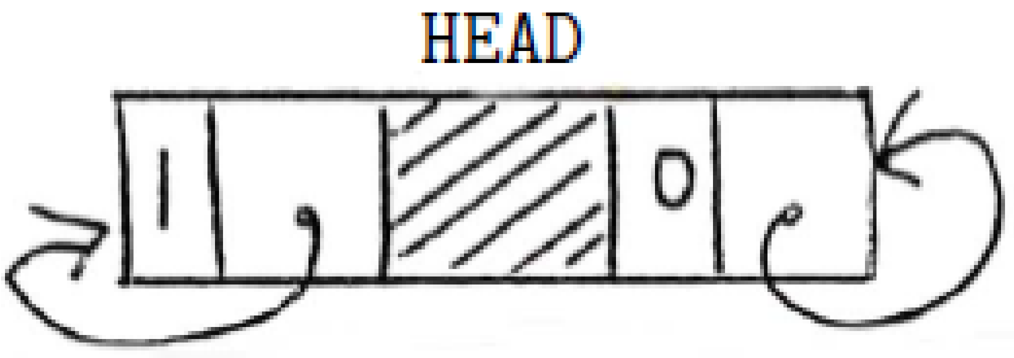
```
struct  ThreadedTreeNode {
       int                       LeftThread;   /* if it is TRUE, then Left */
       ThreadedTree              Left;      /* is a thread, not a child ptr.   */
       ElementType  Element;
       int                       RightThread; /* if it is TRUE, then Right */
       ThreadedTree              Right;     /* is a thread, not a child ptr.   */
};
```


A
B
C
D
E
F
G
H
I

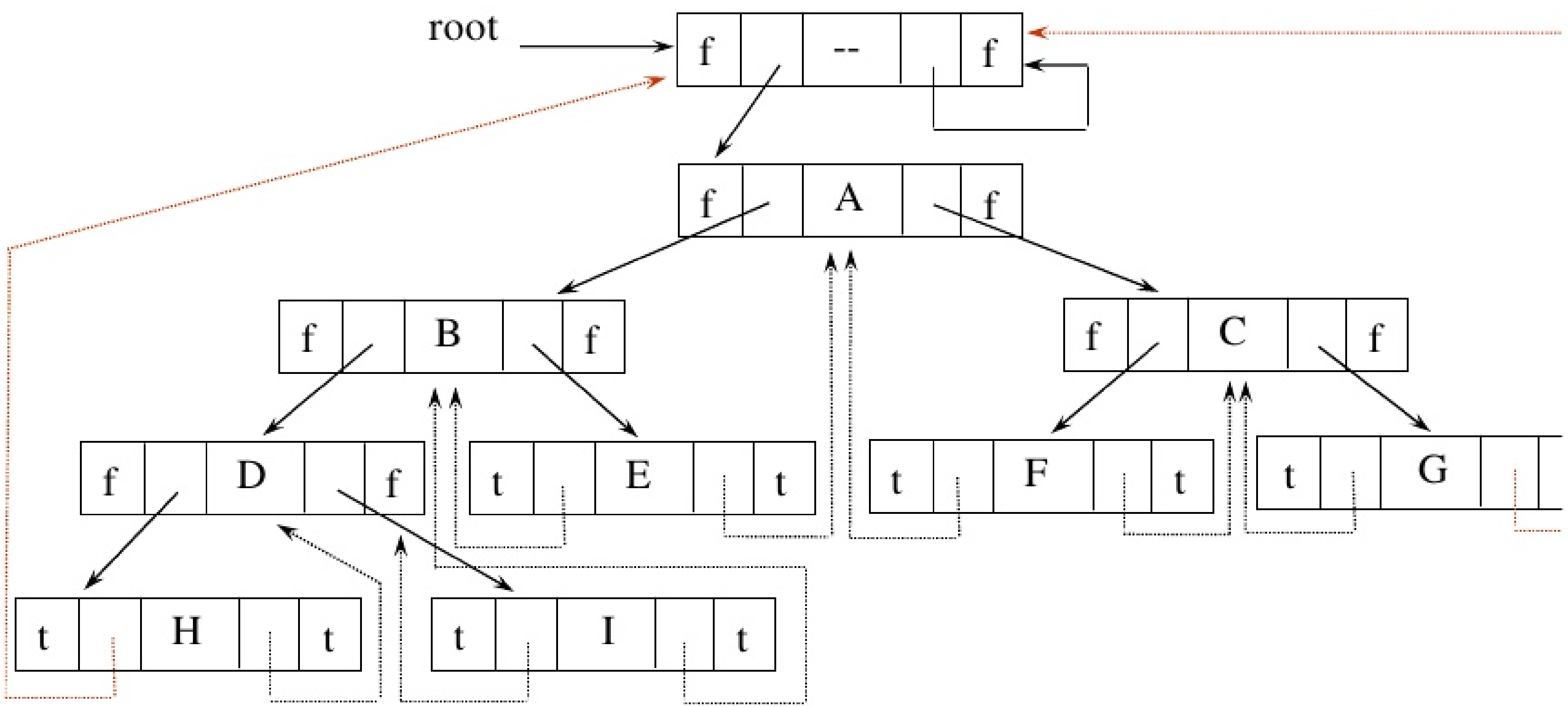

A
B
C
D
E
F
G
H
I

- **Assume that ptr is an arbitrary node in a threaded binary tree, then the following constraints hold:**
- ❑ **If** ptr->leftThread = TRUE or 1**, then** ptr->lcl **contains thread.**
- ❑ **If** ptr->rightThread = TRUE or 1**, then** ptr->rcl **contains thread.**
- **Traditionally,** root->rlink = root **and** root->rightThread = 0 **for any threaded binary tree.**
- **The root points to the header node of the tree, while** root->llink **points to the start of the first node of the actual tree.**
- **The loose thread from the right most node and the left most node is handled by having them pointed to the header node.**

Empty Threaded BT

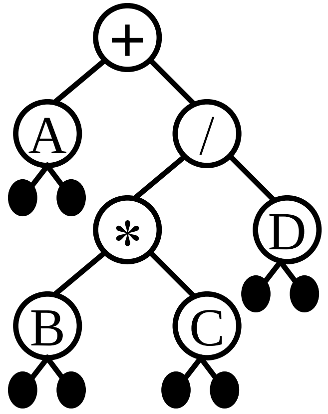


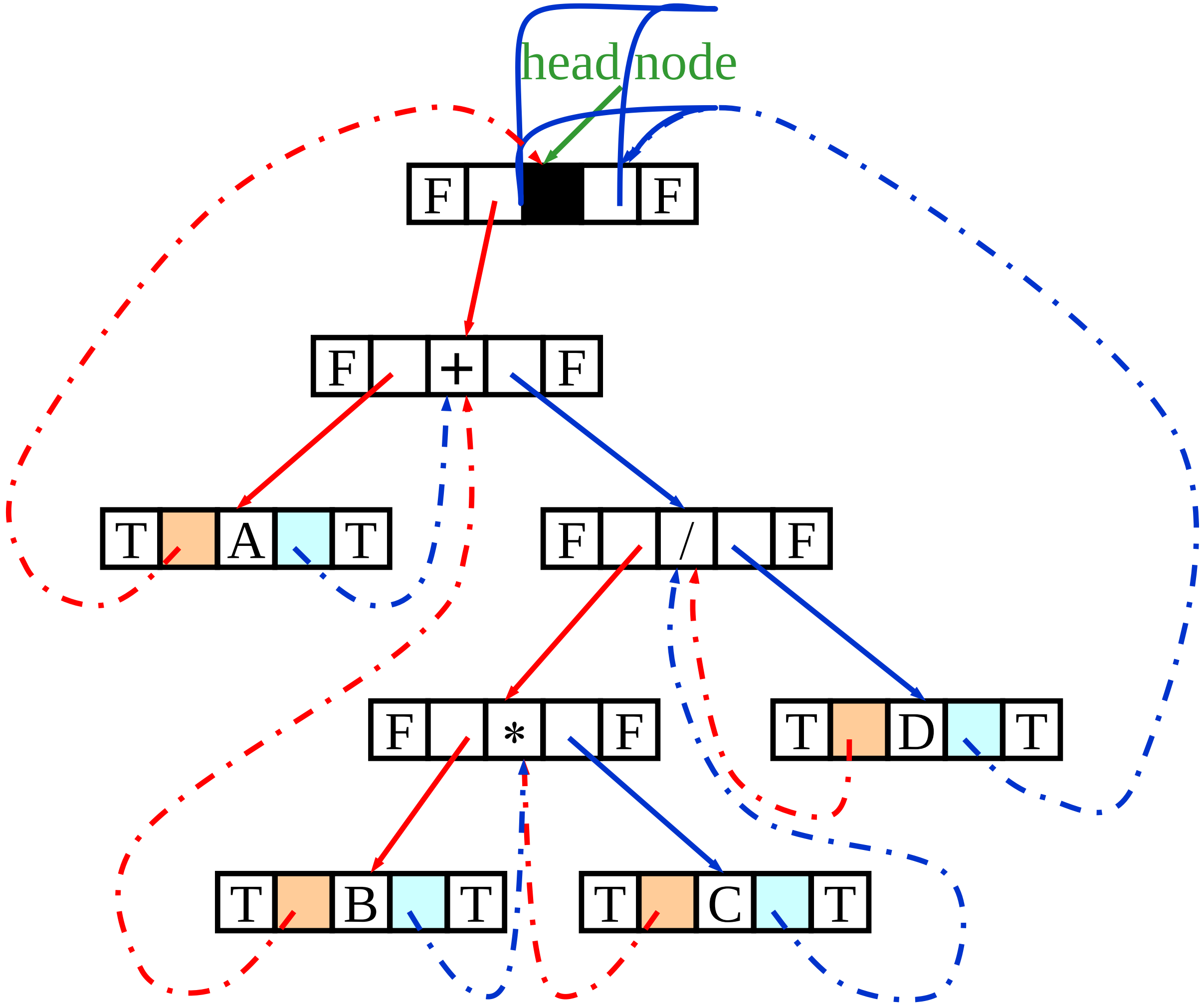
Memory Representation of A Threaded BT



〖Example〗 Given the syntax tree of an expression (infix)

$$A + B * C / D$$





```
void tinorder(node *root)
{
 node *temp=root;
 for(;;)
 {
  temp=in_suc(temp);
  if(temp==root)
      break;
printf(“ %d ”temp->info);
 }
}
```

```
node *in_suc(node *root)
{
 node *temp;
 temp=root->rlink;
 if(!root->rthread)
 {
  while(!temp->lthread)
      temp=temp->llink;
 }
return temp;
}
```

Expression Tree

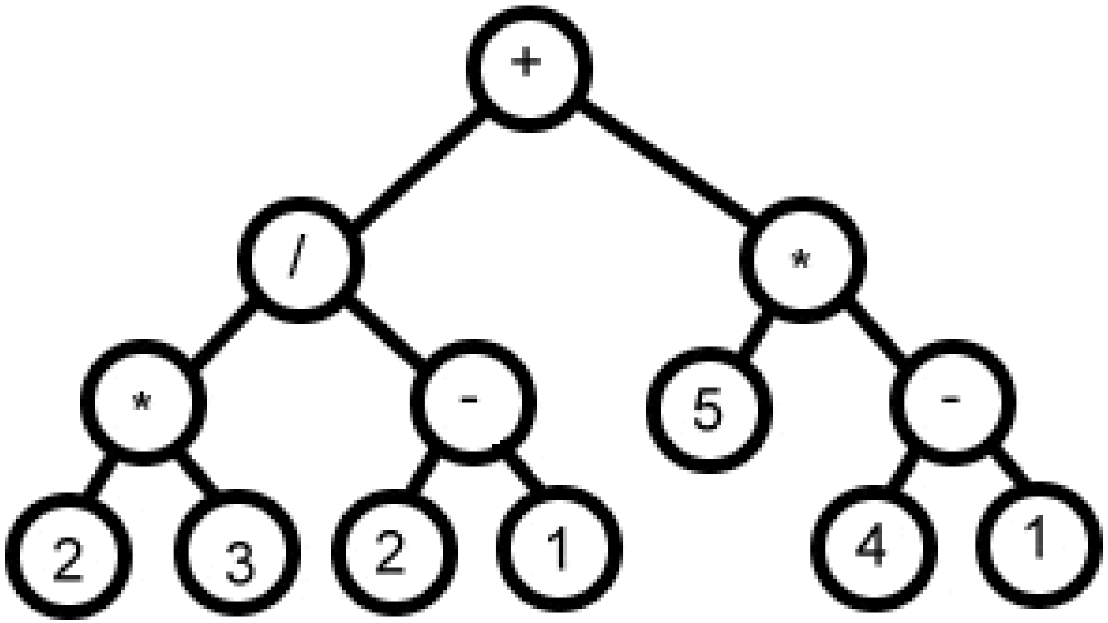
- An expression tree is a specialized binary tree used to represent mathematical expressions or arithmetic operations.

- Each node in the tree represents an operand or operator in the expression.

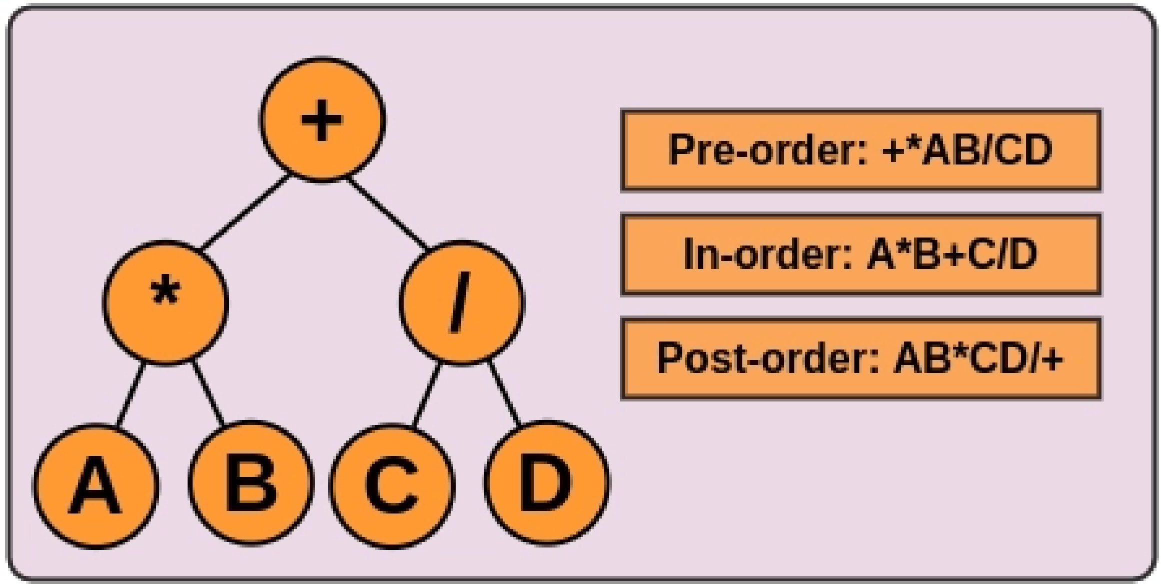
- The leaf nodes of the tree hold operands (e.g., numbers or variables), while internal nodes represent operators (e.g., addition, subtraction, multiplication, division).

- The structure of the expression tree reflects the hierarchical arrangement of the expression, making it a convenient data structure for evaluating and manipulating mathematical expressions.

Expression Tree

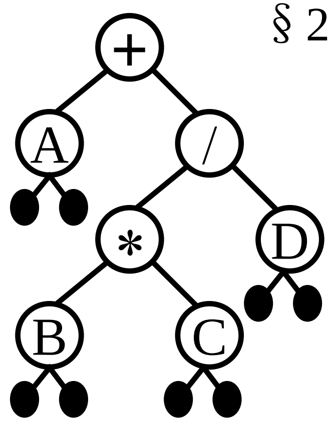


Expression tree for 2*3/(2-1)+5*(4-1)



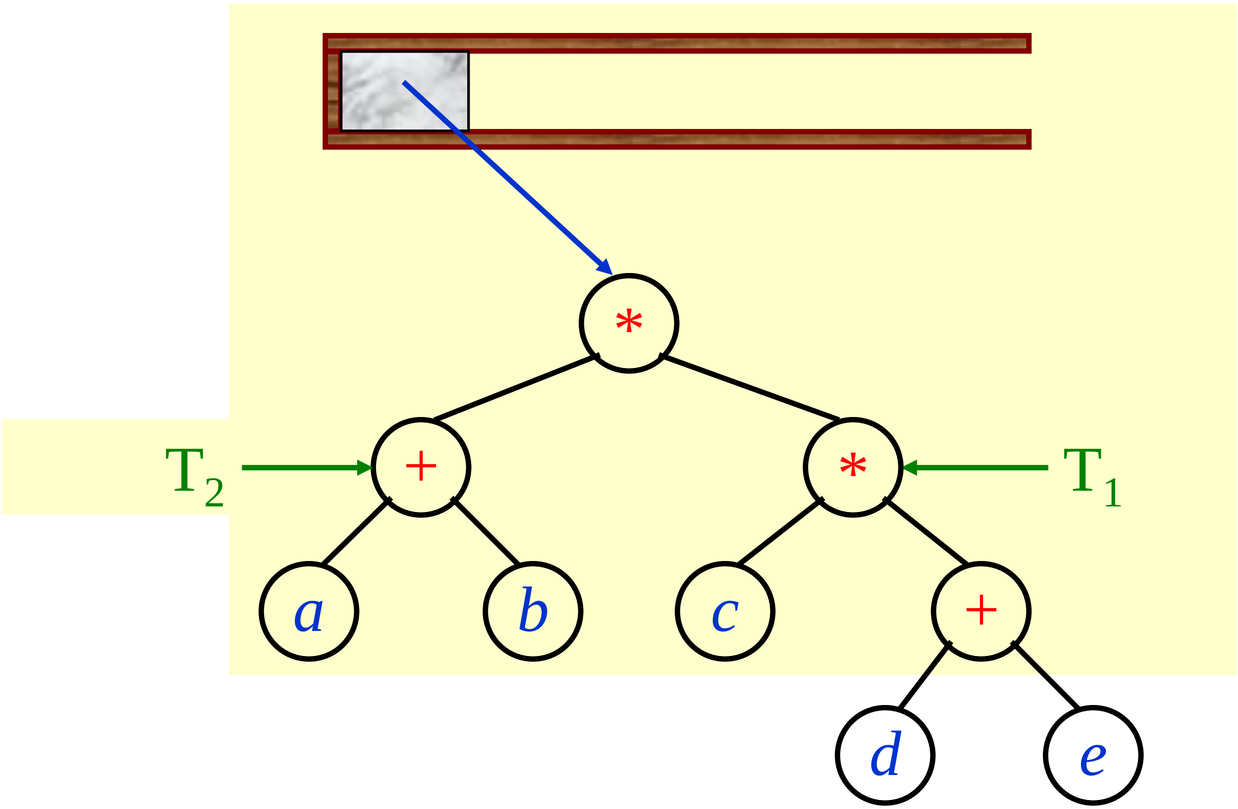
❖ Expression Trees (syntax trees)

〖Example〗 Given an infix expression:
$A + B * C / D$

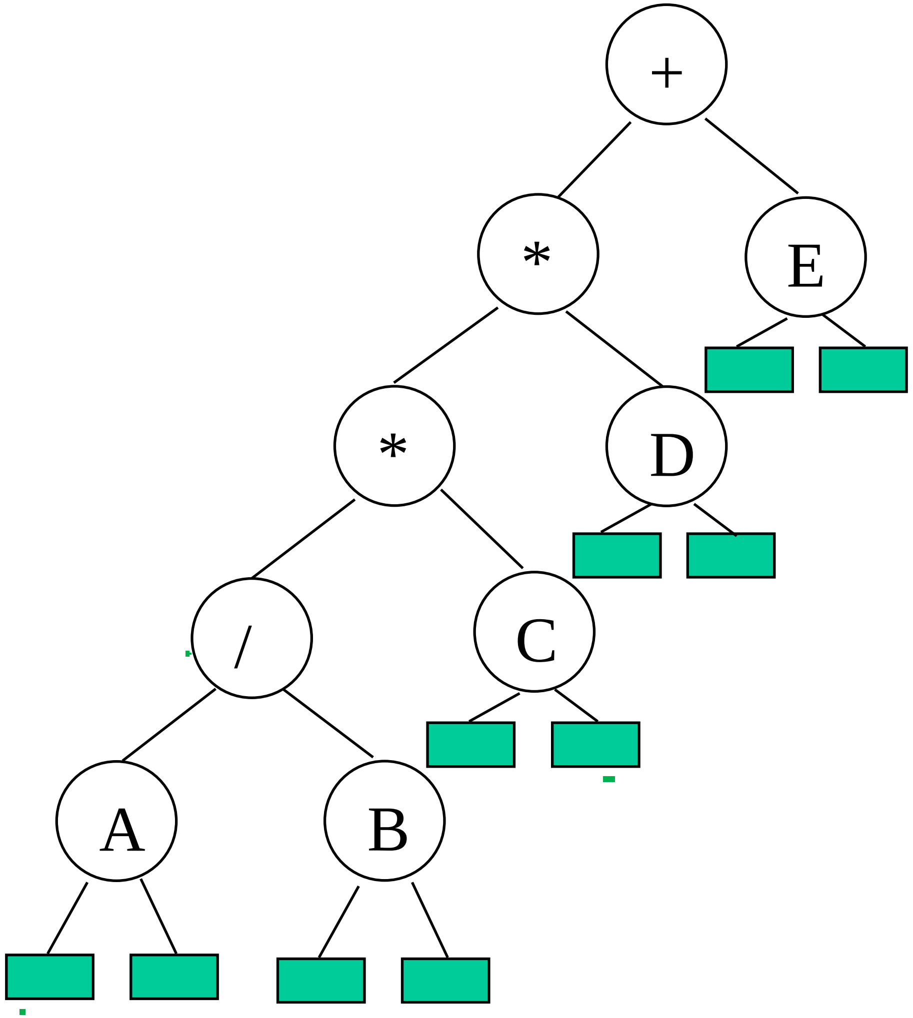


☞ Constructing an Expression Tree
(from postfix expression)

〖Example〗 $(a + b) * (c * (d + e)) = a\ b + c\ d\ e + * *$



Arithmetic Expression Using BT



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

A B / C * D * E +

postfix expression

level order traversal

+ * E * D / C A B

```
struct exp_tree
{

    struct exp_tree *left;
    char data;
    struct exp_tree *right;
};
struct exp_tree* create(char);
struct exp_tree *root=NULL;

struct exp_tree* create(char ele)
{
    struct exp_tree *temp=(struct exp_tree*)(malloc(sizeof(struct exp_tree)));
    temp->data=ele;
   temp->left=temp->right=NULL;
    return temp;
}
```

```
int eval(struct exp_tree* root)
{
    // empty tree
    if (root==NULL)         return 0;

    if (root->left==NULL && root->right==NULL)      return (root->data-48);
    int l_val = eval(root->left); // Evaluate left subtree
    int r_val = eval(root->right); // Evaluate right subtree
    // Check which operator to apply
    if (root->data=='+')  return l_val+r_val;
    if (root->data=='-')  return l_val-r_val;
    if (root->data=='*')  return l_val*r_val;
    if (root->data=='/')  return l_val/r_val;
}
```

```
void inorder(struct exp_tree* root)
{
        if(ptr!=NULL)
        {
                inorder(ptr->left);
                printf(“ %d ”, ptr->data);
                inorder(ptr->right);
        }
}
```

```
int main()
{   char postfix[10];
    int i=0,top=-1;
    struct exp_tree *stack[10];
    printf("Enter the postfix expression: ");    scanf("%s",postfix);
    while(i<strlen(postfix))
    {
            if(isalpha(postfix[i]))  stack[++top]=create(postfix[i]);
            else   //if operator
            {
             root=create(postfix[i]);           root->right=stack[top--];
            root->left=stack[top--];             stack[++top]=root;
        }
        i++;
    }
            inorder(root)
printf(" %d ",eval(root));
}
```