Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Expression Trees

Dr. Shylaja S S
Ms. Kusuma K V

Expression Tree

An expression tree is built up from the simple operands and operators of an (arithmetic or logical) expression by placing the simple operands as the leaves of a binary tree and the operators as internal nodes.

For each binary operator, the left subtree contains all the simple operands and operators in the left operand of the given operator, and the right subtree contains everything in the right operand.

For a unary operator, one of the two subtrees will be empty. We traditionally write some unary operators to the left of their operands, such as '-'(unary negation) or the standard functions like log() and cos(). Other unary operators are written on the right, such as the factorial function ()! or the function that takes the square of a number, () ^{2}. Sometimes either side is permissible, such as the derivative operator, which can be written as d/dx on the left, or as ()' on the right, or the incrementing operator ++ in the C language (where the actions on the left and right are different). If the operator is written on the left, then in the expression tree we take its left subtree as empty, so that the operand appear on the right side of the operator in the tree, just as they do in the expression. If the operator appears on the right, then its right subtree will be empty, and the operands will be in the left subtree of the operator.

The expression trees of few simple expressions are shown in Figure 1.

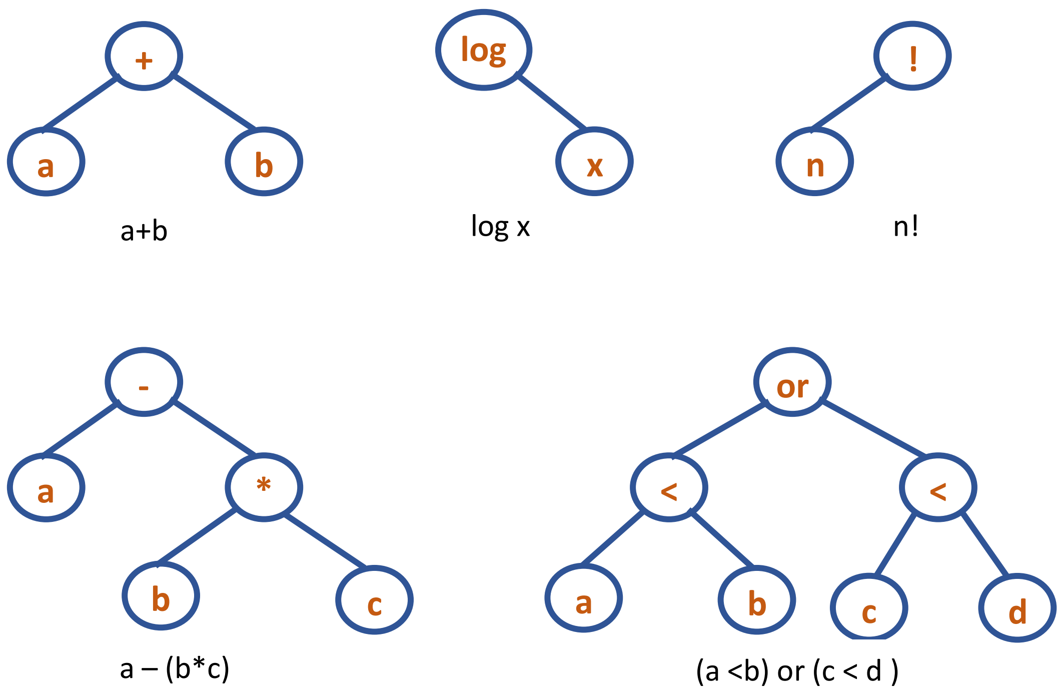


Figure 1: Expression Trees

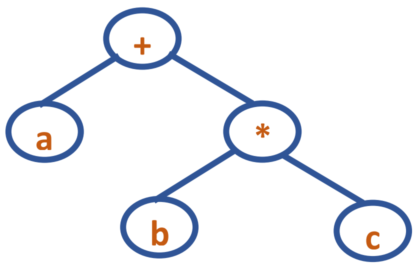
Construction and Evaluation of an Expression Tree

```
1) Scan the postfix expression till the end, one symbol at a time
        a)  Create a new node, with symbol as info and left and right link as NULL
        b) If symbol is an operand, push address of node to stack
        c) If symbol is an operator
                i) Pop address from stack and make it right child of new node
                ii) Pop address from stack and make it left child of new node
                iii) Now push address of new node to stack
 2) Finally, stack has only element which is the address of the root of expression tree
```

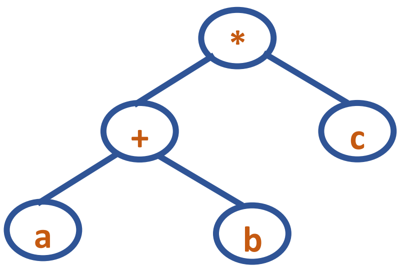
```
//Binary Expression Tree
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#define MAX 50

typedef struct node
{
        char info;
        struct node *left,*right;
}NODE;

typedef struct tree
{
        NODE* root;
}TREE;

typedef struct stack
{
        NODE* s[50];
        int top;
}STACK;

void init_trr(TREE *pt)
{
        pt->root=NULL;
}

void init_stack(STACK *ps)
{
        ps->top=-1;
}
```

```
int push(STACK *ps,NODE* e)
{
	if(ps->top==MAX-1)
		return 0;
	ps->top++;
	ps->s[ps->top]=e;

	return 1;
}

NODE* pop(STACK *ps)
{
	//Assumption: empty condition checked before entering the pop
	NODE *t=ps->s[ps->top];
	ps->top--;

	return t;
}

//Expression Tree Evaluation
float eval(NODE* r)
{
	float res;
	switch(r->info)
	{
		case '+': return (eval(r->left)+eval(r->right));
					break;
		case '-': return (eval(r->left)-eval(r->right));
					break;
		case '*': return (eval(r->left)*eval(r->right));
					break;
		case '/': return (eval(r->left)/eval(r->right));
					break;
		default: return r->info - '0';
	}
	return res;
}

float eval_tree(TREE *pt)
{
	return eval(pt->root);
}
```

```
int main()
{
        char postfix[MAX];
        STACK sobj;
        TREE tobj;
        NODE *temp;
        init_stack(&sobj);
        printf("Enter a valid postfix expression\n");
        scanf("%s",postfix);

        int i=0;
        while(postfix[i] != '\0')
        {
                temp = (NODE*) malloc(sizeof(NODE));

                temp->info = postfix[i];
                temp->left=NULL;
                temp->right=NULL;

                if(isdigit(postfix[i]))
                        push(&sobj,temp);
                else
                {
                        temp->right=pop(&sobj);
                        temp->left=pop(&sobj);
                        push(&sobj,temp);
                }
                i++;
        }
        tobj.root=pop(&sobj);
        printf("%f",eval_tree (&tobj));
        return 0;
}
```



(i) a + (b*c)



(ii) (a + b)*c

Figure 2: Expressions and their binary tree representations

Traversal of the binary expression trees in Figure 2 is as follows:

Figure 2(i): Preorder: +a*bc	Inorder: a+b*c	Postorder: abc*+
Figure 2(ii): Preorder: *+abc	Inorder: a+b*c	Postorder: ab+c*

It can be observed that traversing the binary expression tree in preorder and postoder yields the corresponding prefix and postfix form of the infix expression. But we can see that the inorder traversal of the binary expression trees doesn't always yield the infix form of the expression. This is because the binary expression tree does not contain parentheses, since the ordering of the operations is implied by the structure of the tree.

General Expression Tree

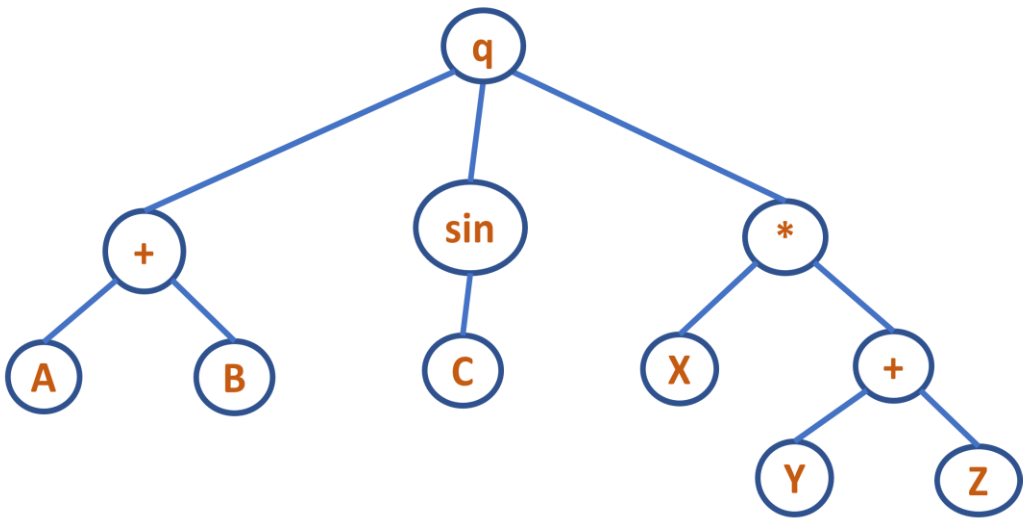


Figure: Tree representation of an arithmetic expression

Here node can be either an operand or an operator

```
struct treenode
{
  short int utype;
   union{
      char operator[MAX];
       float val;
     }info;
   struct treenode *child;
   struct treenode *sibling;
};
typedef struct treenode TREENODE;
```

```
void replace(TREENODE *p)
{
  float val;
  TREENODE *q,*r;
  if(p->utype == operator)
 {
   q = p->child;
   while(q != NULL)
    {
      replace(q);
      q = q->next;
    }
 }
  value = apply(p);
  p->utype = OPERAND;
  p->val = value;
  q = p->child;
  p->child = NULL;
  while(q != NULL)
  {
      r = q;
      q = q->next;
      free(r);
  }
 }
}

float eval(TREENODE *p)
{
 replace(p);
 return(p->val);
 free(p);
}
```

Constructing a Tree

```
void setchildren(TREENODE *p,TREENODE *list)
{
   if(p == NULL) {
     printf("invalid insertion");
     exit(1);
   }
  if(p->child != NULL) {
    printf("invalid insertion");
    exit(1);
  }
  p->child = list;
}

void addchild(TREENODE *p,int x)
{
  TREENODE *q;
  if(p==NULL)
  {
    printf("void insertion");
    exit(1);
  }
  r = NULL;
  q = p->child;

  while(q != NULL)
  {
    r = q;
    q = q->next;
  }
  q = getnode();
  q->info = x;
  q->next = NULL;

  if(r==NULL)
    p->child=q;
  else
    r->next=q;
}
```