DATA STRUCTURES AND ITS APPLICATIONS

UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Implementation of Binary Expression Tree

Shylaja S S
Department of Computer Science & Engineering

Expression Tree

- An expression can be represented using the **Expression Tree** data structure
- Such a tree is built normally for translating the code as data and then analysing and evaluating expressions
- **Immutable**: To change the expression another tree has to be constructed

Expression Tree Construction

- Normally a postfix expression is used in constructing the Expression tree
- When an operand is received, a new node is created which will be a leaf in the expression tree
- If an operator, it connects to two leaves
- Stack DS is used as intermediary storing place of node's address

Expression Tree Construction

Postfix Expression: abc*+

Symbol = a

Address=100

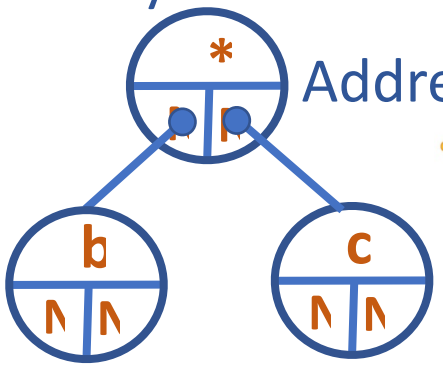
Symbol=b

Address=150

Symbol=c

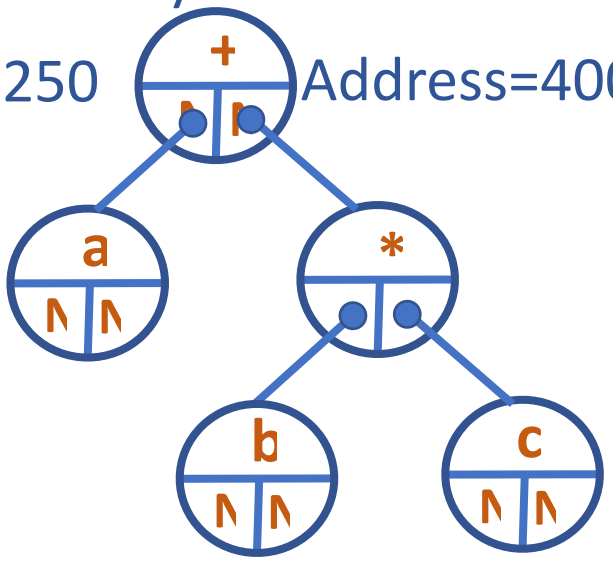
Address=300

Symbol = *

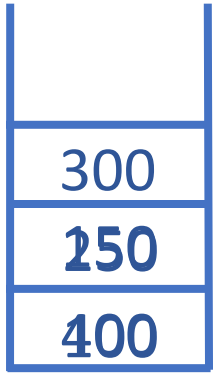


Address=250

Symbol = +



Address=400



Expression Tree Construction

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

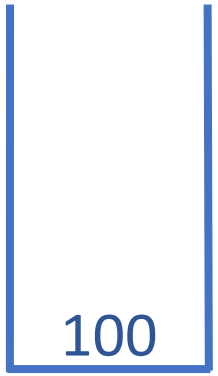
Expression Tree Construction

Postfix Expression: **a b c * +**

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

Symbol = a

Address=100



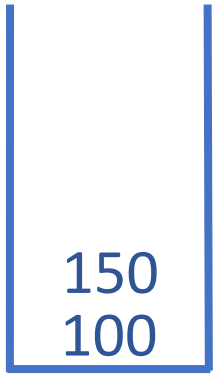
Expression Tree Construction

Postfix Expression: **a b c * +**

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

Symbol = b

Address=150



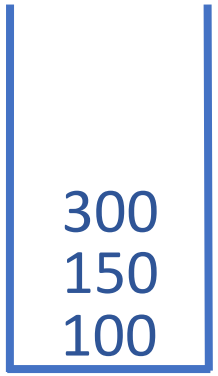
Expression Tree Construction

Postfix Expression: **a b c * +**

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

Symbol = c

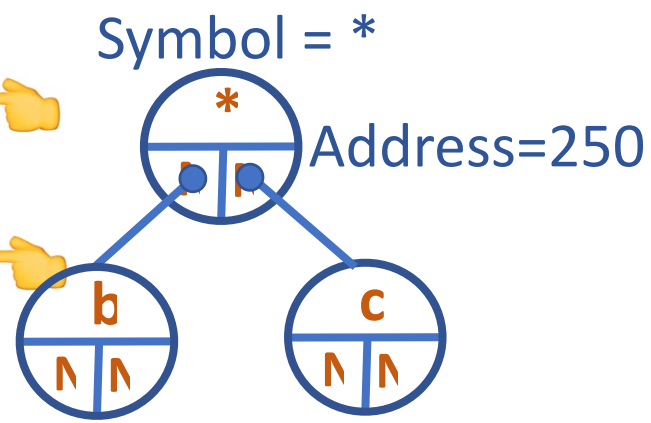
Address=300

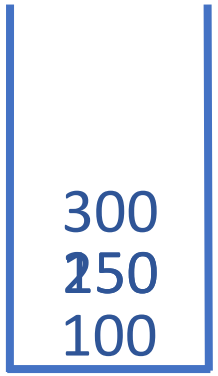


Expression Tree Construction

Postfix Expression: **a b c * +**

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

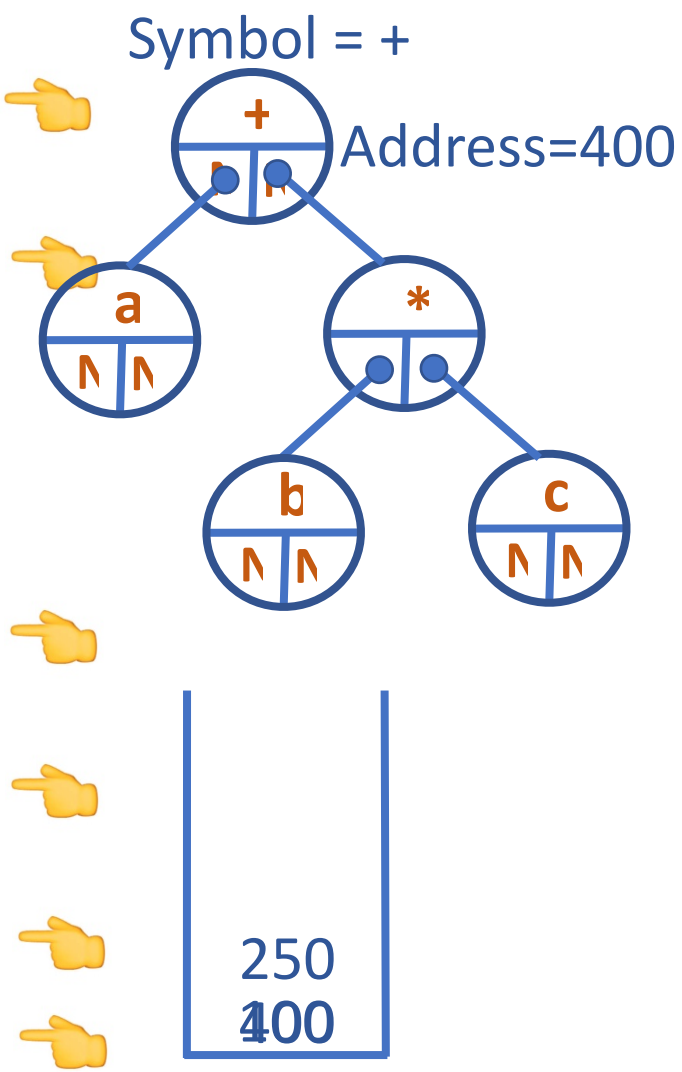




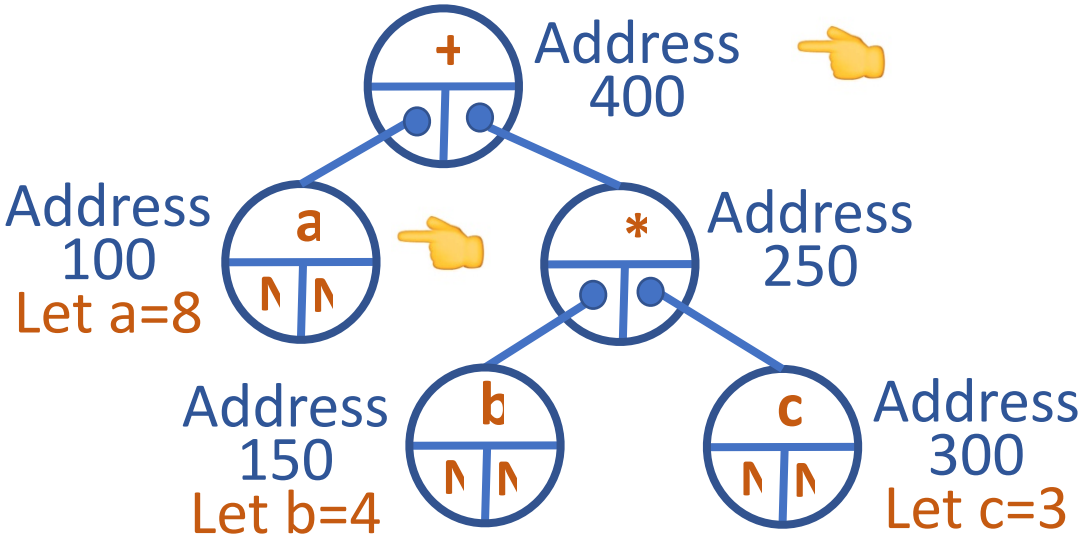
Expression Tree Construction

Postfix Expression: **a b c * +**

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



Expression Tree Evaluation



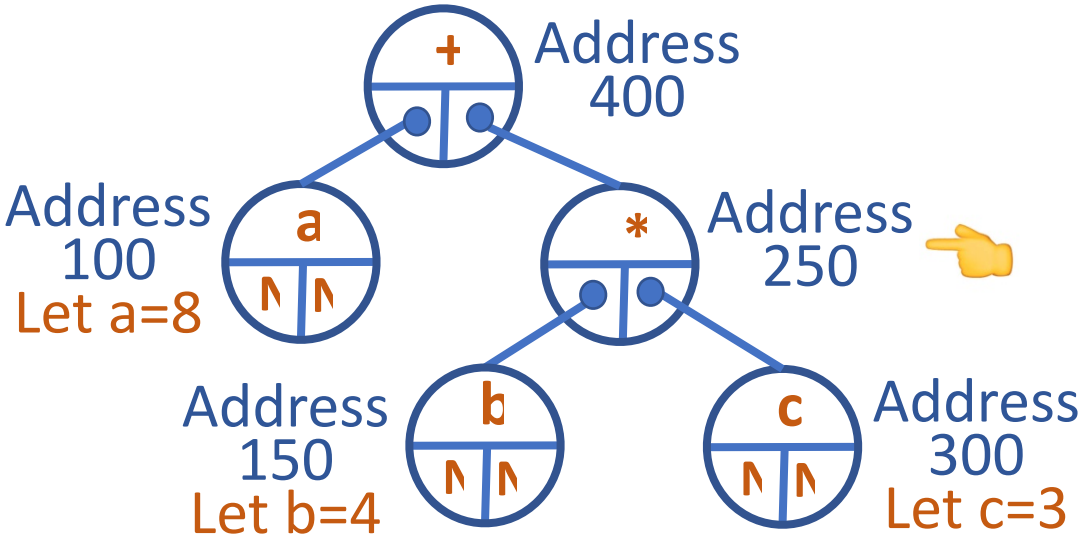
eval(400)
return eval(100) + eval(250)

eval(100)
return 8

- Think in terms of recursion

```
eval(t)// 't' has the address of the root node of expression tree
  if t->data is an operator
    return eval (t->left) t->data eval(t->right)
  return t->data
```

Expression Tree Evaluation



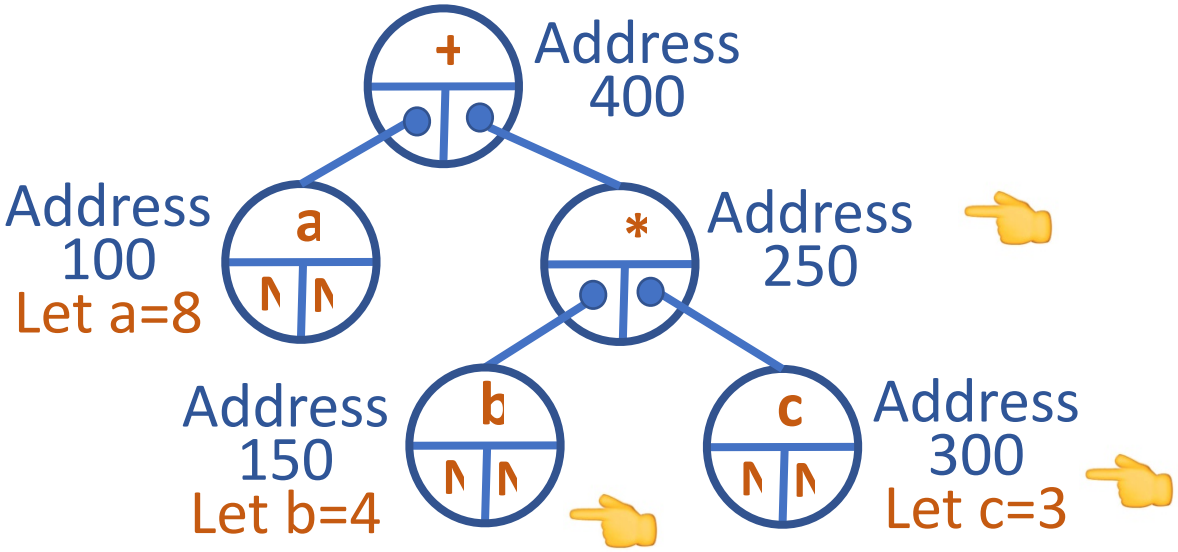
eval(400)
return 8 + eval(250)

eval(250)
return eval(150) * eval(300)

- Think in terms of recursion

```
eval(t)// 't' has the address of the root node of expression tree
  if t->data is an operator
    return eval (t->left) t->data eval(t->right)
  return t->data
```

Expression Tree Evaluation



eval(400)
return 8 + eval(250)

eval(250)
return eval(150) * eval(300)

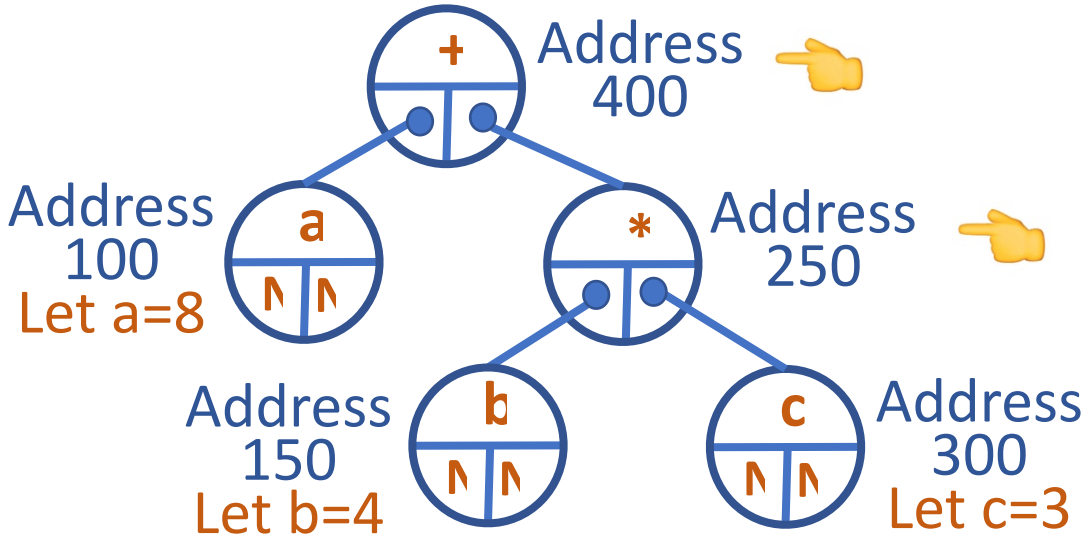
eval(150)
return 4

eval(300)
return 3

- Think in terms of recursion

```
eval(t)   // 't' has the address of the root node of expression tree
  if t->data is an operator
    return eval (t->left) t->data eval(t->right)
  return t->data
```

Expression Tree Evaluation



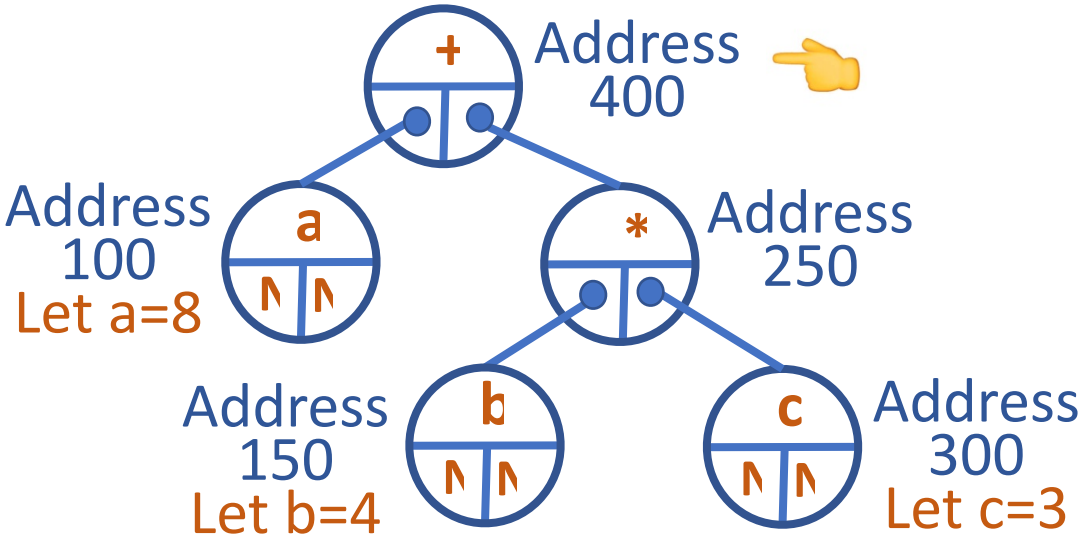
eval(400)
return 8 + eval(250)

eval(250)
return 12

- Think in terms of recursion

```
eval(t)     // 't' has the address of the root node of expression tree
  if t->data is an operator
    return eval (t->left) t->data eval(t->right)
  return t->data
```

Expression Tree Evaluation



eval(400)

return 208 + 12

Postfix abc*+ : 20

- Think in terms of recursion

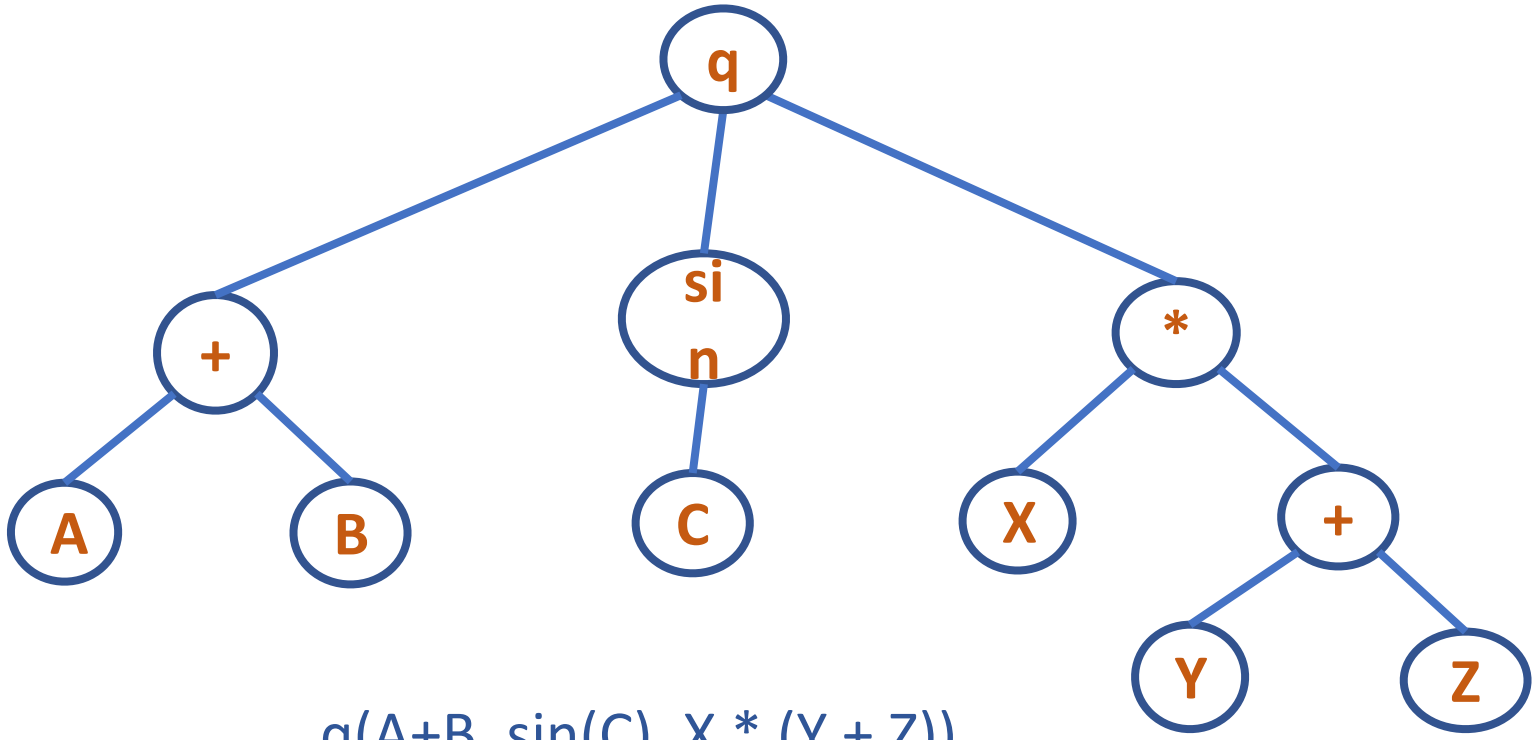
```
eval(t)     // 't' has the address of the root node of expression tree
  if t->data is an operator
    return eval (t->left) t->data eval(t->right)
  return t->data
```

General Expression Tree Evaluation

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

General Expression Tree Evaluation

Here node can be either an operand or an operator



q(A+B, sin(C), X * (Y + Z))

Tree representation of an arithmetic expression

General Expression Tree Evaluation

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

General Expression Tree Evaluation

```
  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);
  }
 }
}
```

General Expression Tree Evaluation

```
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;
}
```

Constructing a Tree

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

Constructing a Tree

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

Constructing a Tree

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

THANK YOU

Shylaja S S

Department of Computer Science & Engineering

shylaja.sharath@pes.edu

+91 9449867804