DATA STRUCTURES AND ITS APPLICATIONS

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

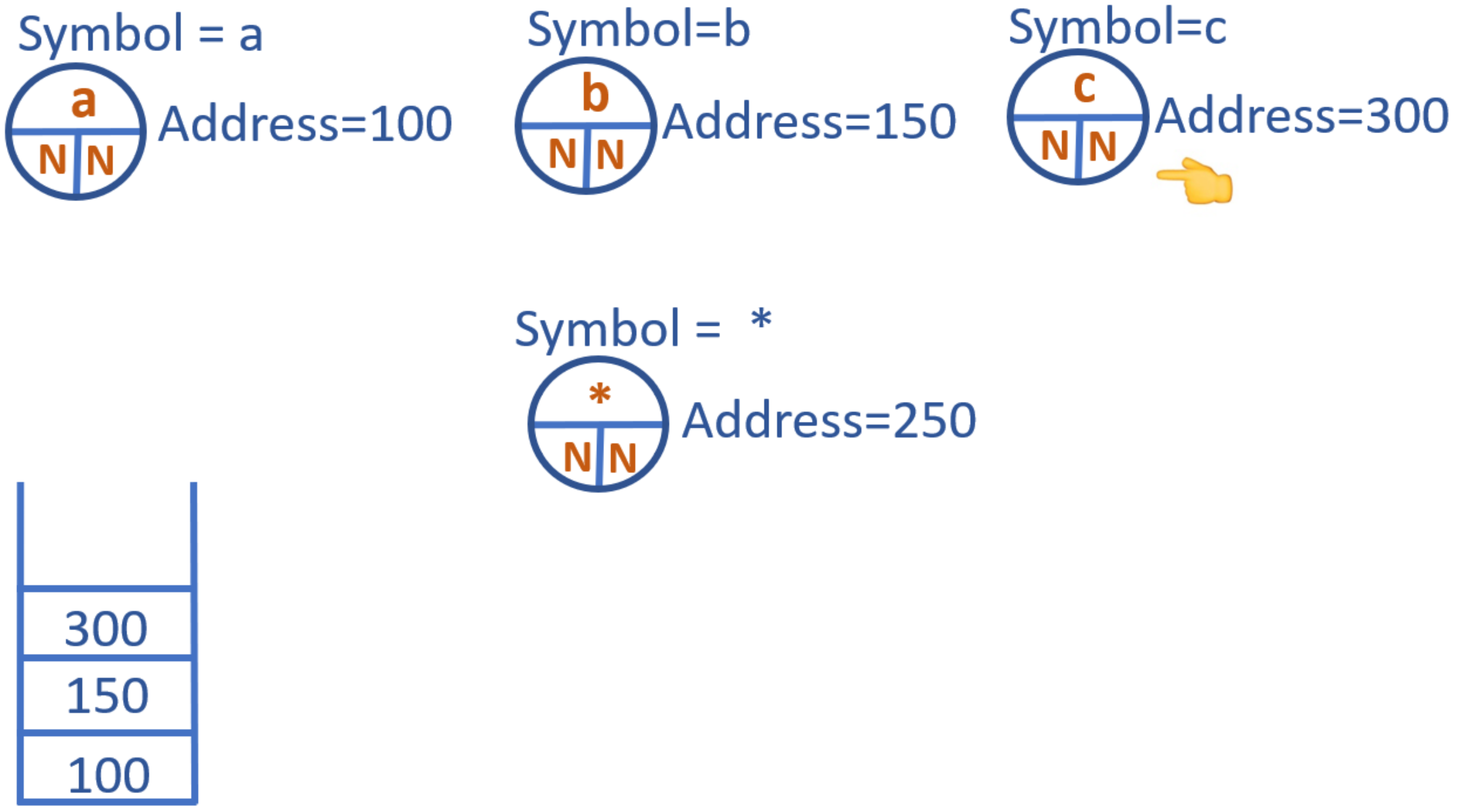
- An expression can be represented using the **Expression Tree** data structure
- Such a tree is built normally for translating the code as data and then analysing and evaluating expressions
- **Immutable**: To change the expression another tree has to be constructed

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

a | N | N   Address=100

Symbol=b

b | N | N   Address=150

Symbol=c

c | N | N   Address=300

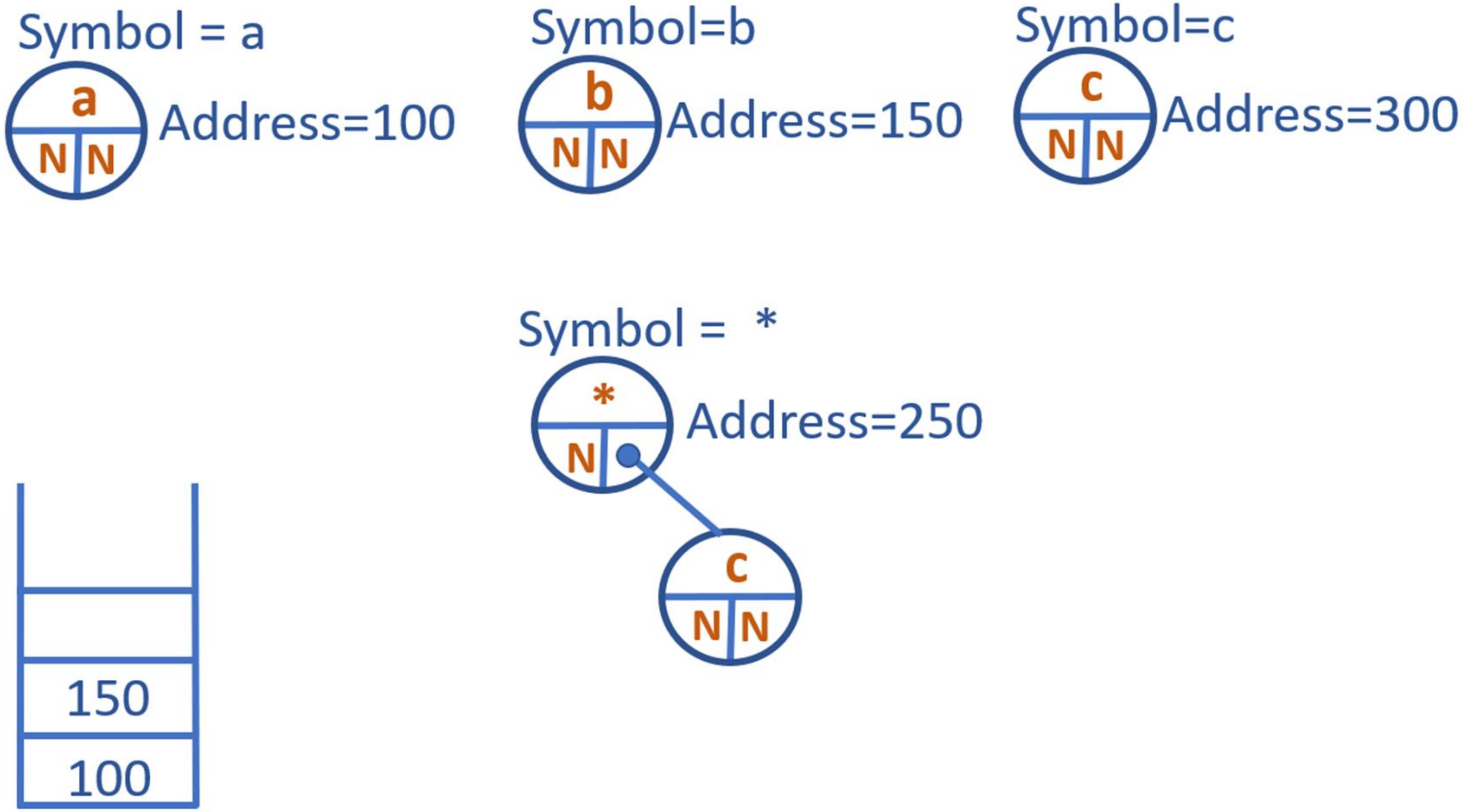
Symbol = *

* | N | N   Address=250

300
150
100

Expression Tree Construction

Postfix Expression: abc*+



Symbol = a

Address=100

Symbol=b

Address=150

Symbol=c

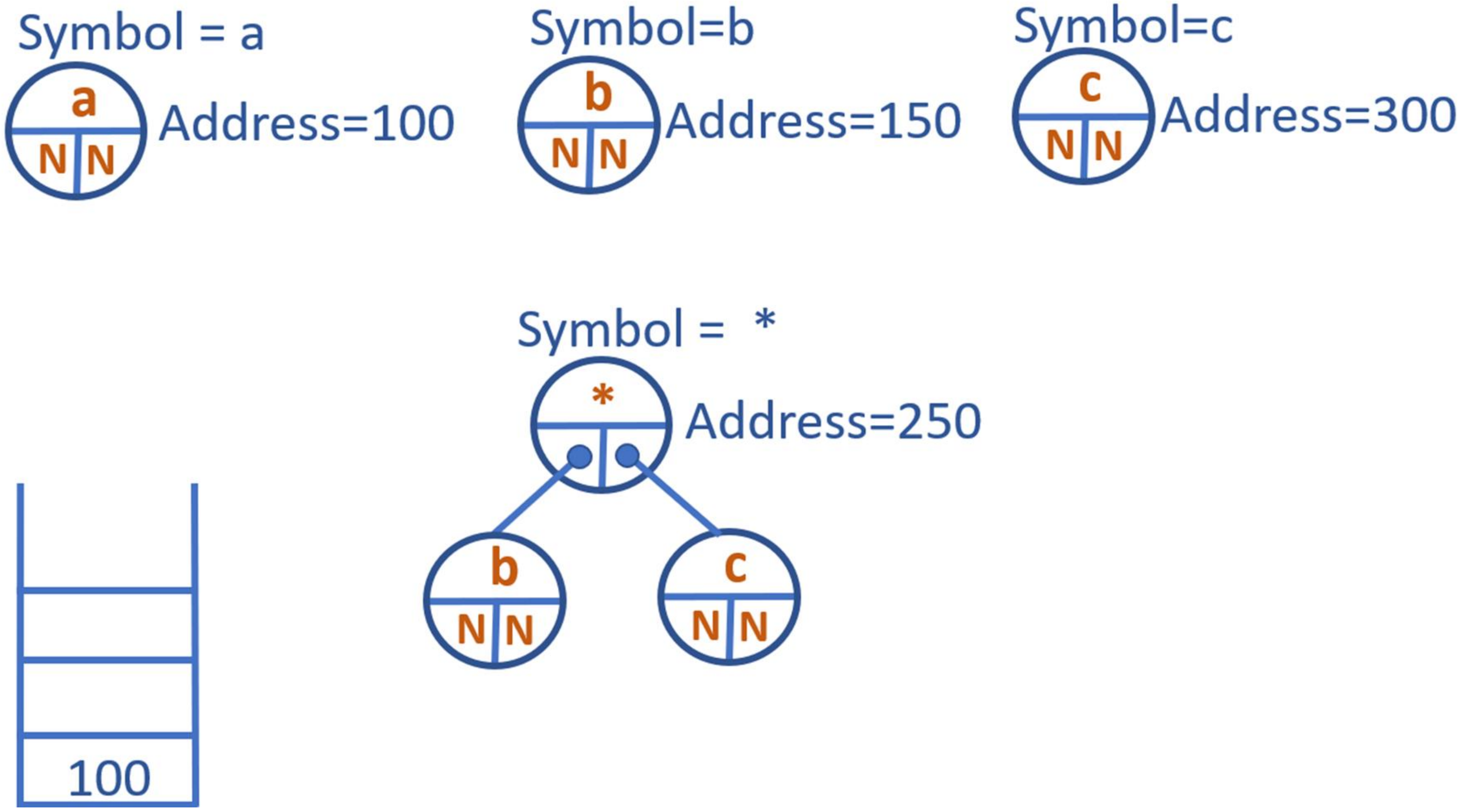
Address=300

Symbol = *

Address=250

Expression Tree Construction

Postfix Expression: abc*+



Symbol = a

Address=100

Symbol=b

Address=150

Symbol=c

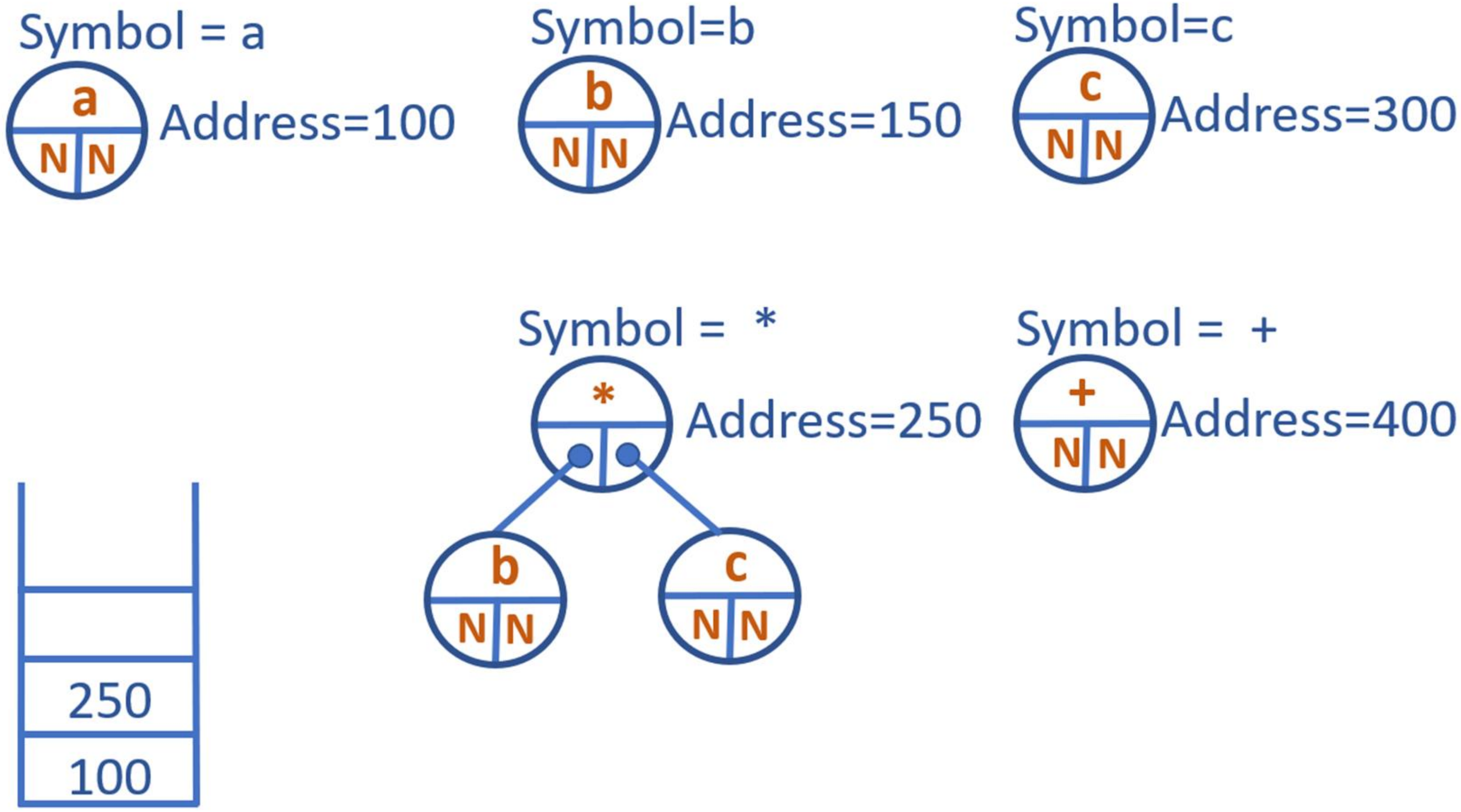
Address=300

Symbol = *

Address=250

Expression Tree Construction

Postfix Expression: abc*+



Symbol = a

Address=100

Symbol=b

Address=150

Symbol=c

Address=300

Symbol = *

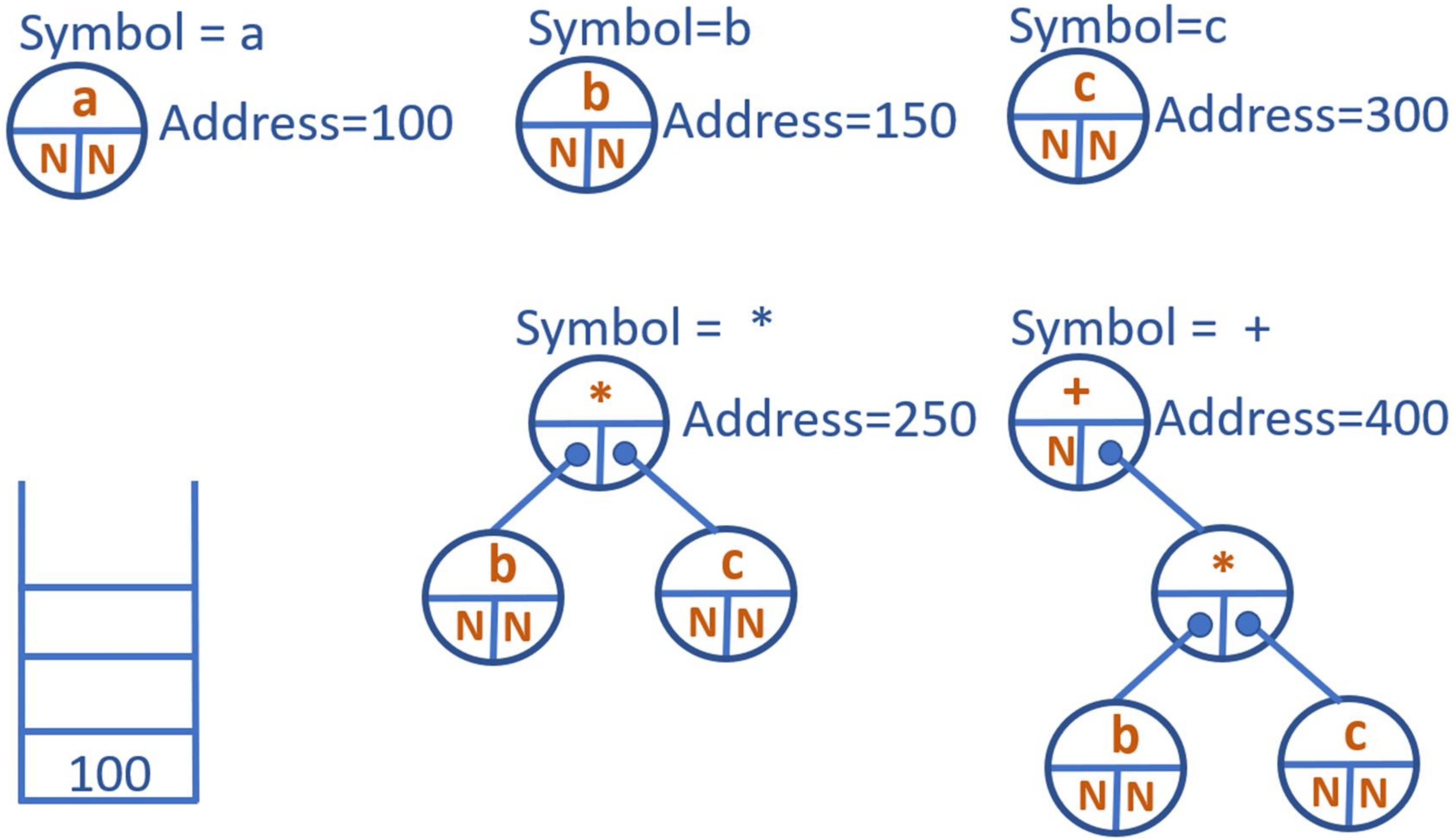
Address=250

Symbol = +

Address=400

Expression Tree Construction

Postfix Expression: abc*+



Symbol = a

Address=100

Symbol=b

Address=150

Symbol=c

Address=300

Symbol = *

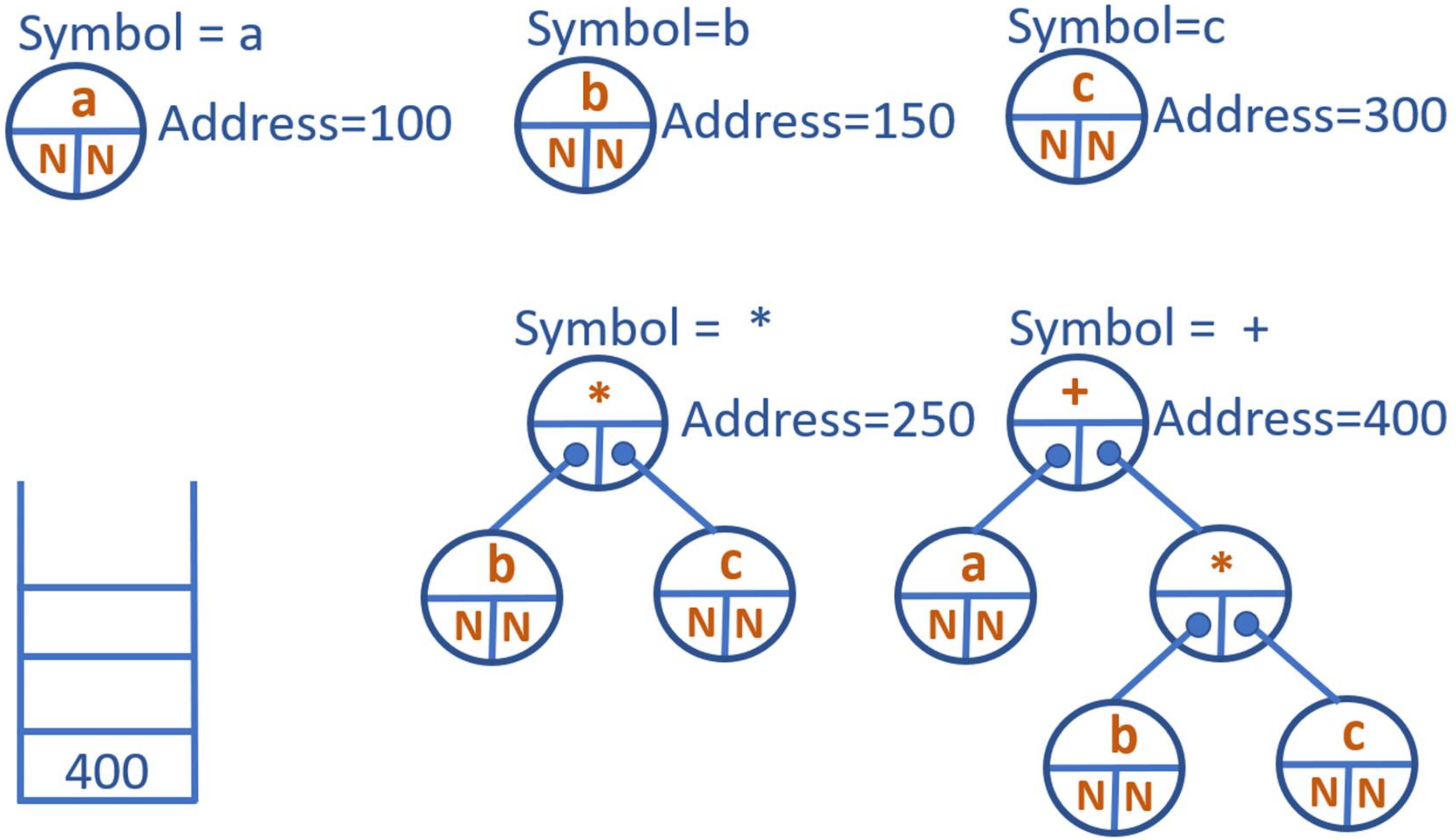
Address=250

Symbol = +

Address=400

Expression Tree Construction

Postfix Expression: abc*+



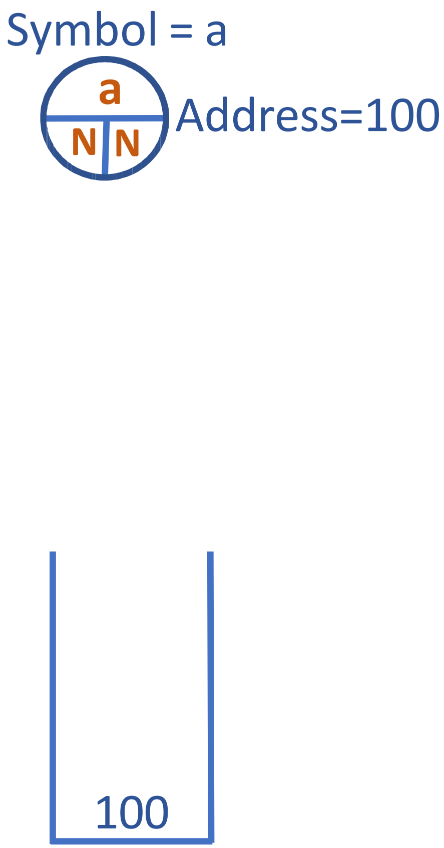
Expression Tree Construction

- 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 bc * +**

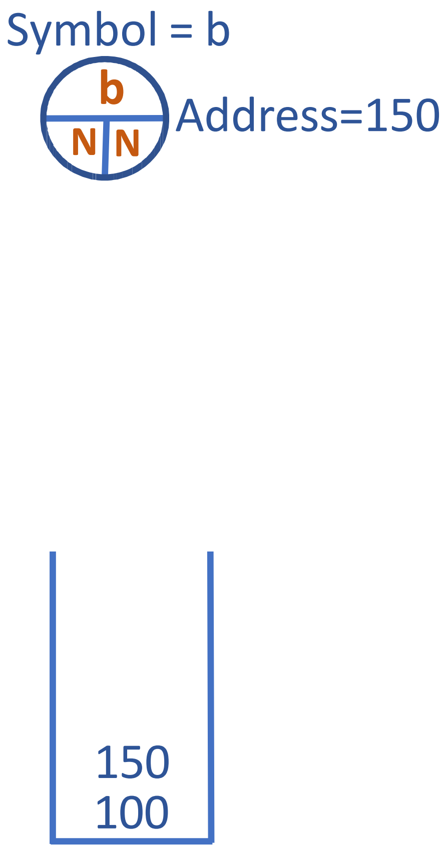
- Scan the postfix expression till the end, one symbol at a time
    - Create a new node, with symbol as info and left and right link as NULL
    - If symbol is an operand, push address of node to stack
    - If symbol is an operator
        - Pop the address from stack and make it right child of new 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 bc * +**

- 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



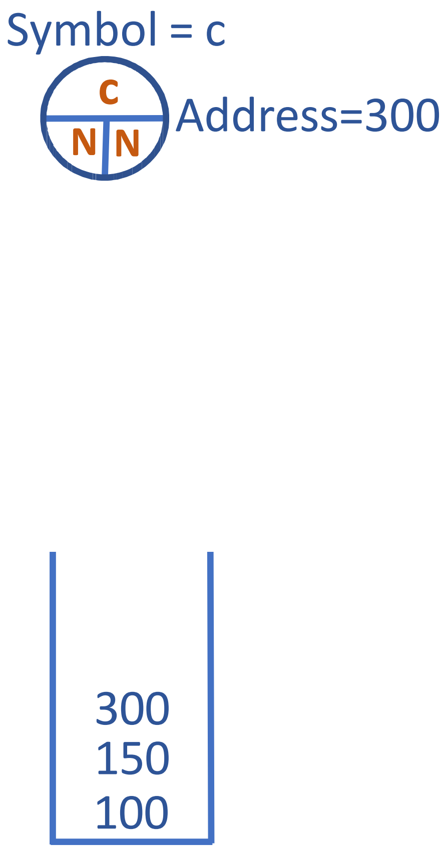
Symbol = b

Address=150

Expression Tree Construction

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

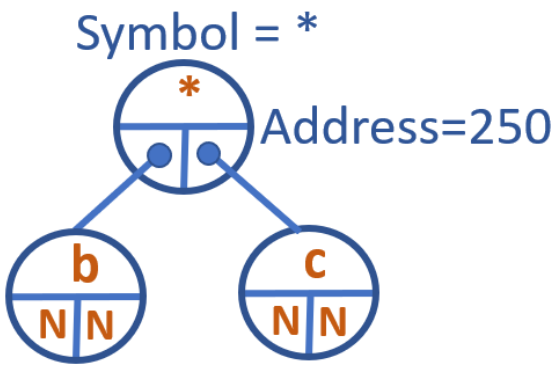
- Scan the postfix expression till the end, one symbol at a time
  - Create a new node, with symbol as info and left and right link as NULL
  - If symbol is an operand, push address of node to stack
  - If symbol is an operator
    - Pop the address from stack and make it right child of new node
   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

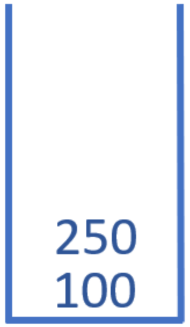


Expression Tree Construction

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

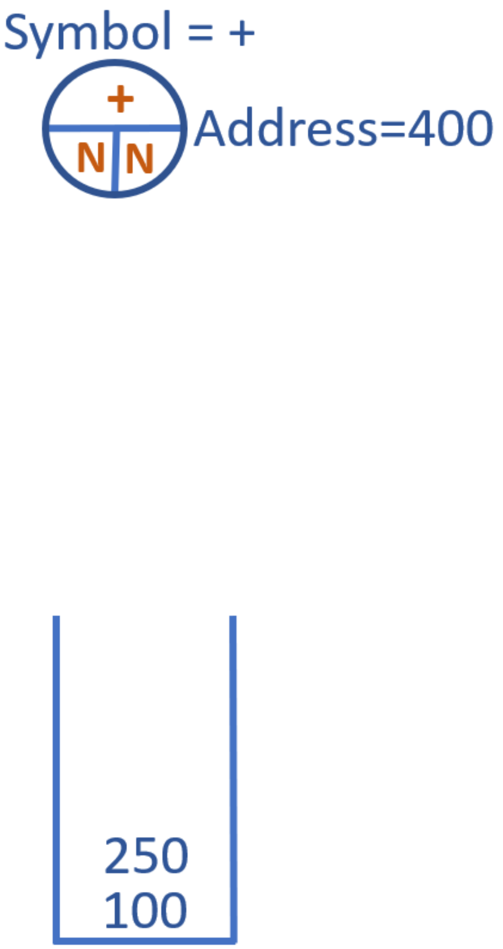




Expression Tree Construction

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

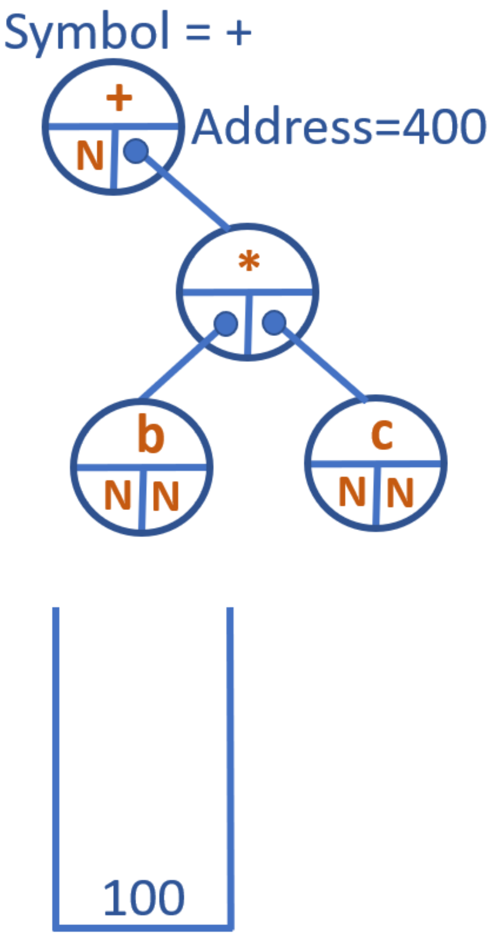


DATA STRUCTURES AND ITS APPLICATIONS

Expression Tree Construction

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

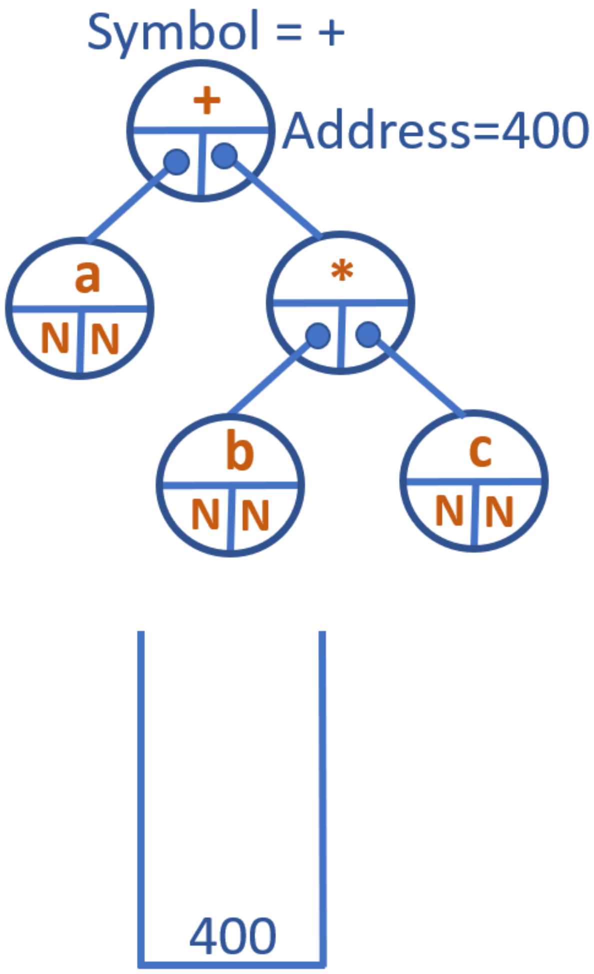
- Scan the postfix expression till the end, one symbol at a time
  - Create a new node, with symbol as info and left and right link as NULL
  - If symbol is an operand, push address of node to stack
  - If symbol is an operator
    - Pop the address from stack and make it right child of new node
   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



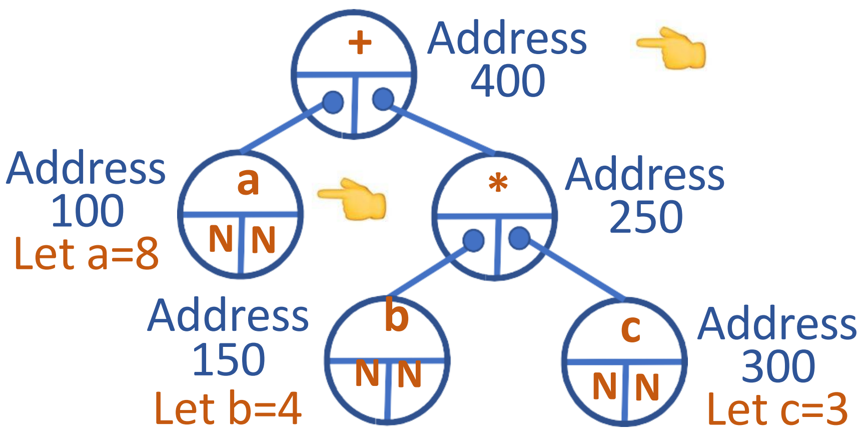
Expression Tree Construction

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



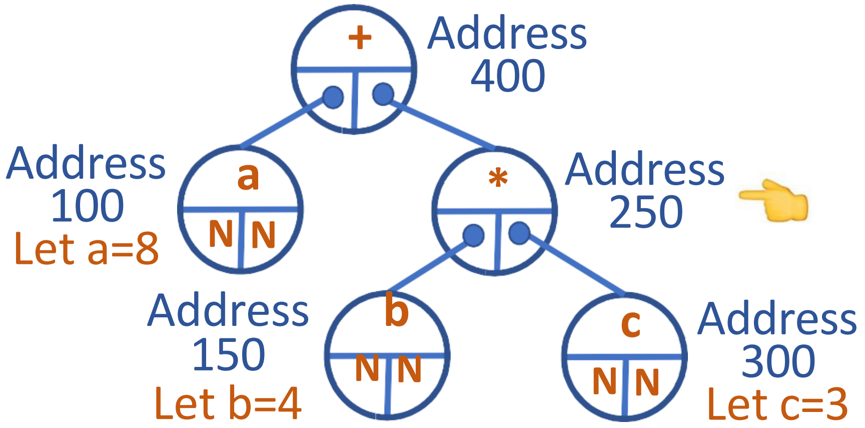
eval(400)
  return eval(100)+eval(250)

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



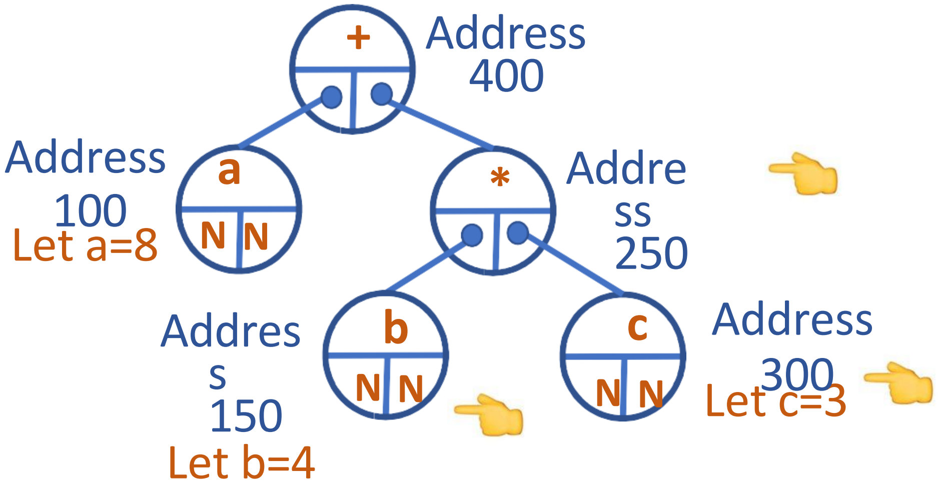
```
eval(400)
    return        8      +  eval(250)
eval(250)
    return eval(150)  *  eval(300)
```

- 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
```

DATA STRUCTURES AND ITS APPLICATIONS

Expression Tree Evaluation



eval(400)

return **8** **+** eval(250)

eval(250)

eval(150) + eval(300)

eval(150)

return **4**

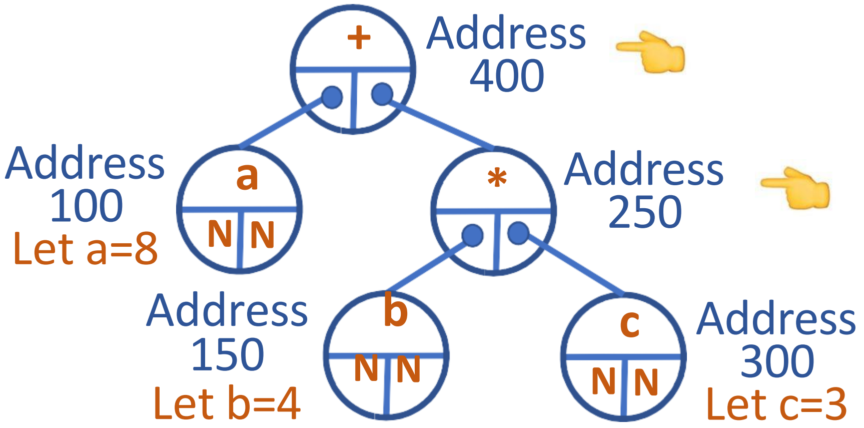
eval(300)

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



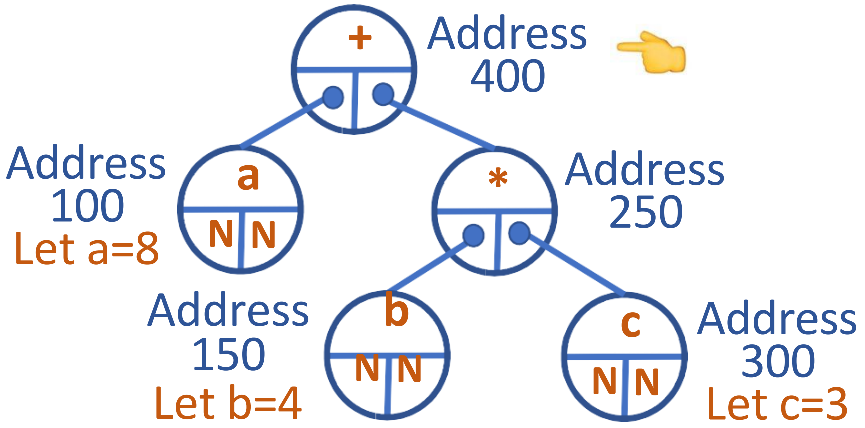
eval(400)
return 8 + eval(250)

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

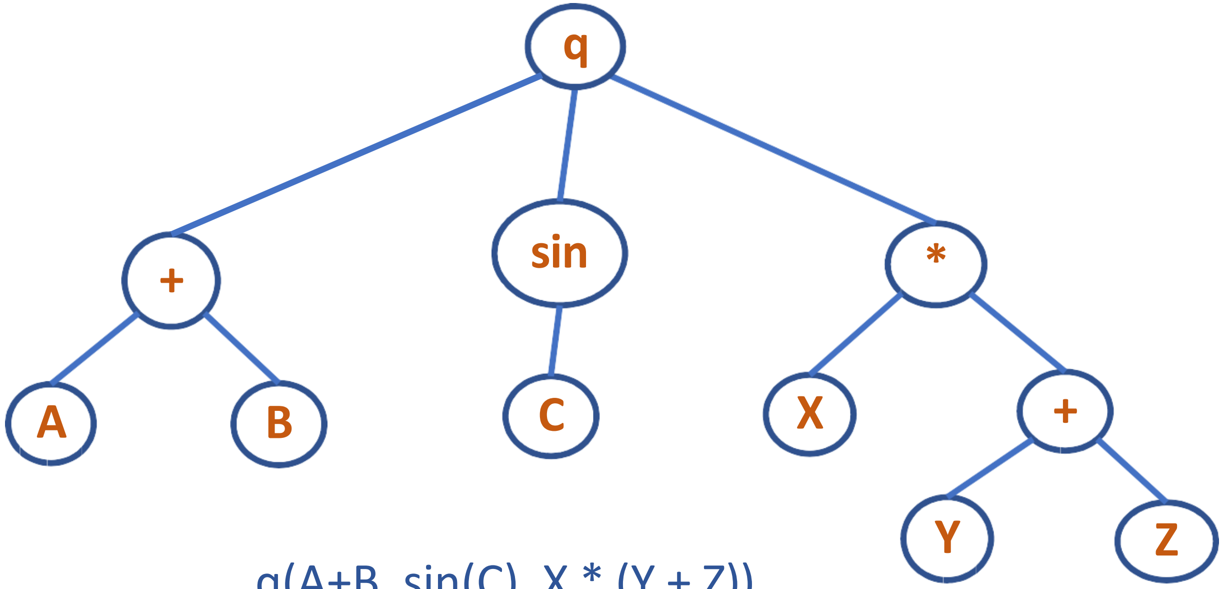
```
eval(t)     // 't' has the address of the root node of expression tree
   if t->data is an operator
      return eval (t->left) t->data eval(t->right)
   return t->data
```

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;
    }
     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;
}
```

Constructing a Tree

```
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;
}
```

1. What is the value of the expression tree :

```
     *
    / \
   +   5
  / \
 4   3
```

a) 20

b) 35

c) 15

d) 25

1. What is the value of the expression tree:

```
      *
     / \
    +   5
   / \
  4   3
```

a) 20

b) 35

c) 15

d) 25

2. Which of the following statements about Expression Trees is true?

A) An expression tree can only represent arithmetic operators, not operands.

B) Expression trees are mutable; you can directly modify them without reconstruction.

C) Expression trees are mainly used for analyzing and evaluating expressions.

D) Expression trees cannot be built from postfix expressions.

2. Which of the following statements about Expression Trees is true?

A) An expression tree can only represent arithmetic operators, not operands.

B) Expression trees are mutable; you can directly modify them without reconstruction.

C) Expression trees are mainly used for analyzing and evaluating expressions.

D) Expression trees cannot be built from postfix expressions.

3. While constructing an expression tree from a postfix expression using a stack, what happens when an operator is encountered?

A) It is pushed directly into the stack as a leaf node.

B) Two operands are popped from the stack and become children of the operator node.

C) The operator is discarded, and only operands are stored.

D) The operator replaces the top operand in the stack.

3. While constructing an expression tree from a postfix expression using a stack, what happens when an operator is encountered?

A) It is pushed directly into the stack as a leaf node.

B) Two operands are popped from the stack and become children of the operator node.

C) The operator is discarded, and only operands are stored.

D) The operator replaces the top operand in the stack.

4. For the postfix expression abc*+ with a=8, b=4, c=3, what will be the result of evaluating the corresponding expression tree?

A) 8 + (4 * 3) = 20

B) (8 * 4) + 3 = 35

C) (8 + 4) * 3 = 36

D) (8 * 3) + 4 = 28

4. For the postfix expression abc*+ with a=8, b=4, c=3, what will be the result of evaluating the corresponding expression tree?

A) 8 + (4 * 3) = 20

B) (8 * 4) + 3 = 35

C) (8 + 4) * 3 = 36

D) (8 * 3) + 4 = 28

5. An expression tree is used to represent:

a) Arithmetic expressions

b) Logical expressions

c) Both arithmetic and logical expressions

d) Only postfix expressions

5. An expression tree is used to represent:

a) Arithmetic expressions

b) Logical expressions

c) Both arithmetic and logical expressions

d) Only postfix expressions

THANK YOU

Shylaja S S

Department of Computer Science & Engineering

shylaja.sharath@pes.edu