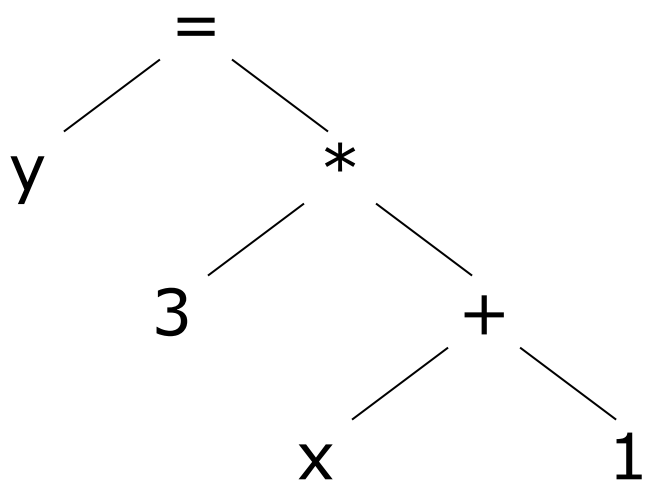
Expression Tree

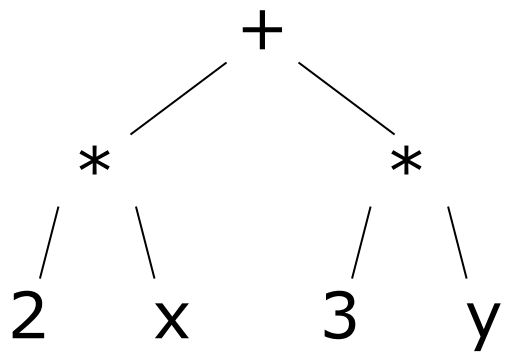
An expression tree is a tree that represents an expression.



y = 3 * (x + 1)

Expression Tree

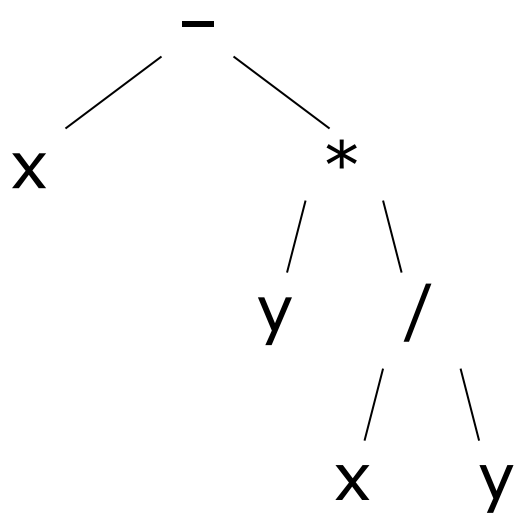
An expression tree is a tree that represents an expression.



2*x + 3*y

Expression Tree

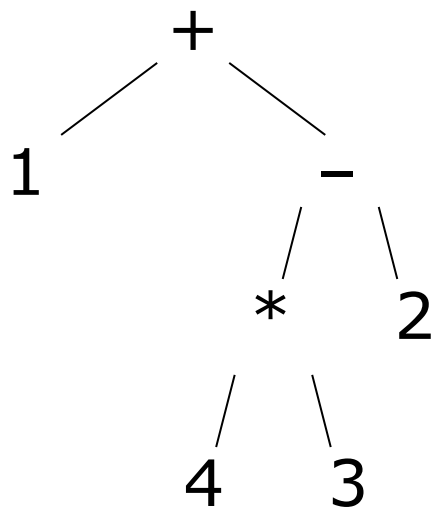
An expression tree is a tree that represents an expression.



x – (y * (x / y))

Expression Tree

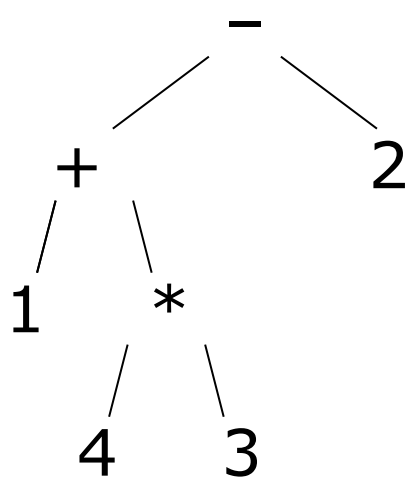
An expression tree is a tree that represents an expression.



1 + (4 * 3 – 2)

Expression Tree

An expression tree is a tree that represents an expression.



(1 + 4 * 3) – 2

Expression Tree

An expression tree is a tree that represents an expression.

5

5

Expression Tree

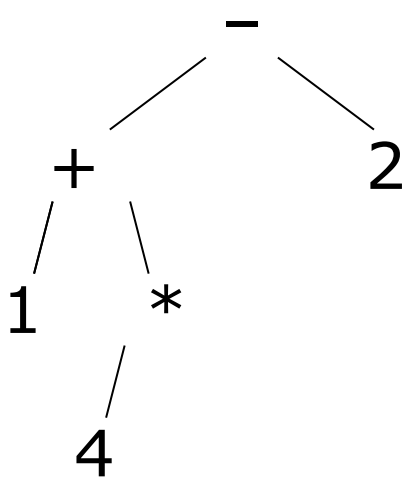
An expression tree is a tree that represents an expression.

x

x

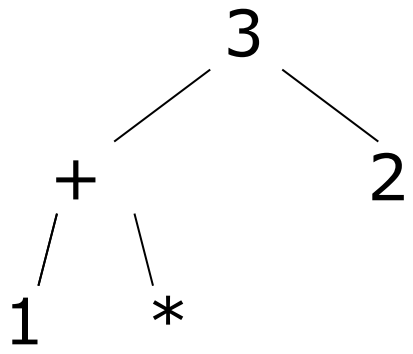
Expression Tree

This is NOT an expression tree.

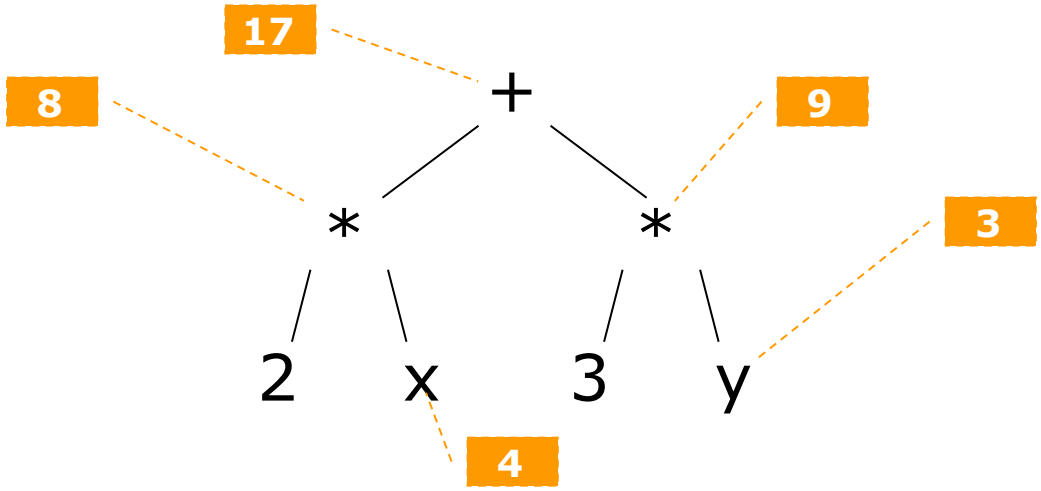


Expression Tree

This is NOT an expression tree.



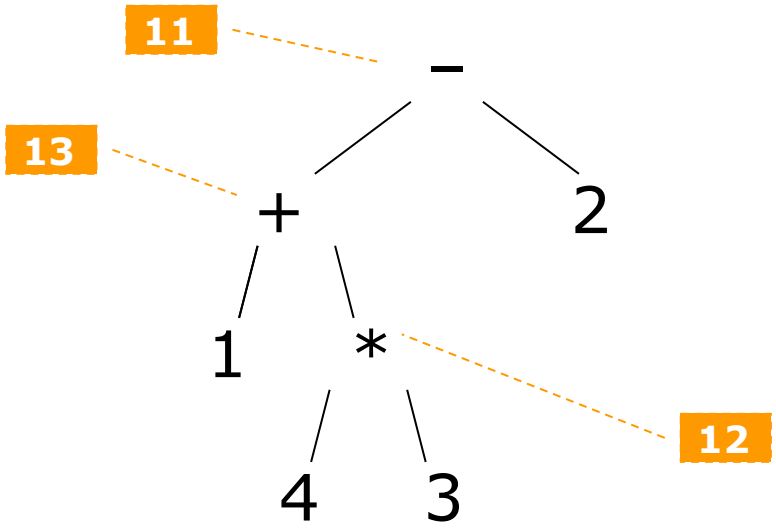
Evaluating an Expression



{x = 4, y = 3}

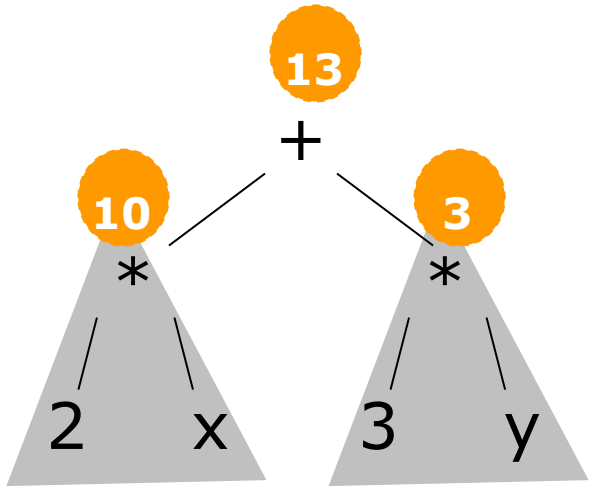
2*x + 3*y

Evaluating an Expression



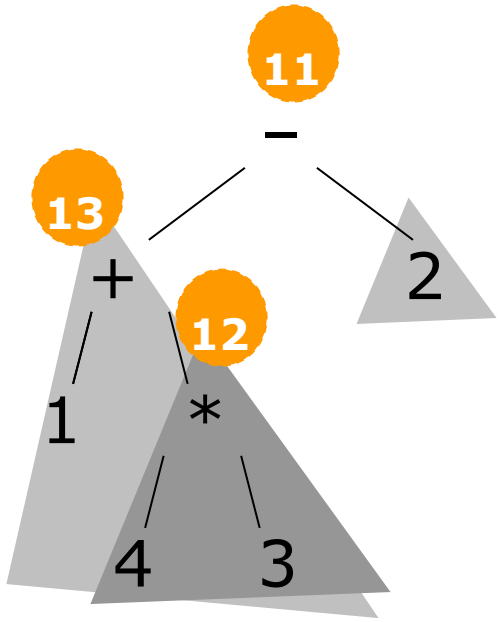
1 + 4 * 3 – 2

Evaluating an Expression: Another Viewpoint



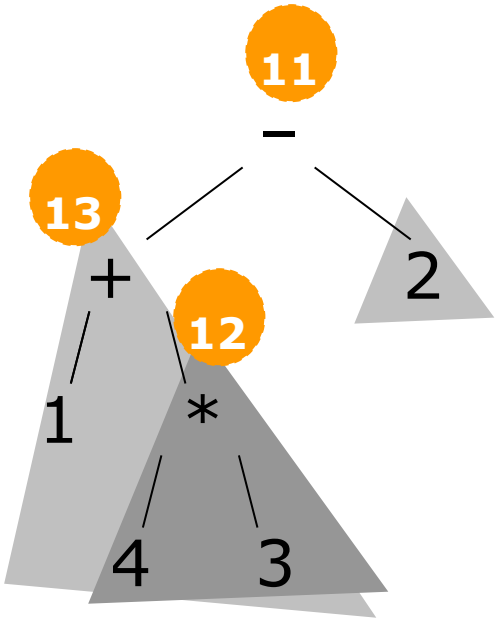
{x = 5, y = 1}

Evaluating an Expression: Another Viewpoint



1 + 4 * 3 – 2

Evaluating an Expression: Another Viewpoint



1 + 4 * 3 – 2

```
ExprTreeADT t1 = …;
…
int Result = Eval(t1);
```

```
/* In the implementation file of ExprTreeADT
*/
int Eval(ExprTreeADT t) {
    …
    …
    LHS = Eval(LeftExprSubtree(t));
    RHS = Eval(RightExprSubtree(t));

    op = ExprTreeRoot(t);
    return (LHS op RHS);
}
```

Evaluating an Expression: Another Viewpoint

5

```
/* In the implementation file of ExprTreeADT
*/
int Eval(ExprTreeADT t) {
    ...
    if (Height(t) == 1) return ExprTreeRoot(t);
    ...
    LHS = Eval(LeftExprSubtree(t));
    RHS = Eval(RightExprSubtree(t));

    op = ExprTreeRoot(t);
    return (LHS op RHS);
}
```

5

Evaluating an Expression: Another Viewpoint

What should **ExprTreeRoot(t)** return?

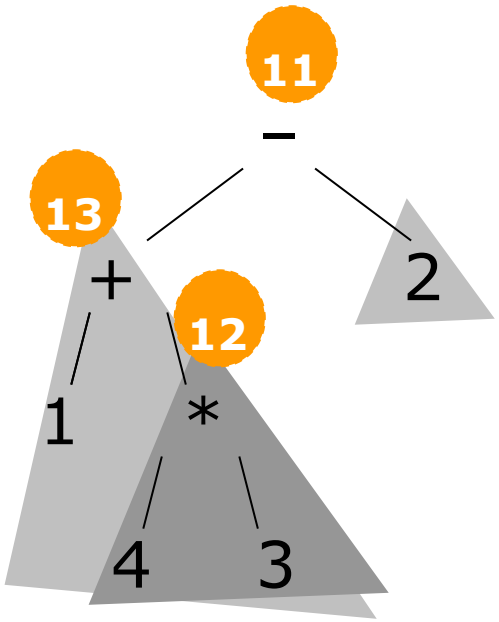
- an integer?
- an operator? (*But do you know how to return an operator, after all!*)

```
/* In the implementation file of ExprTreeADT
*/
int Eval(ExprTreeADT t) {
   ...
   if (Height(t) == 1) return ExprTreeRoot(t);
   ...
   LHS = Eval(LeftExprSubtree(t));
   RHS = Eval(RightExprSubtree(t));

   op = ExprTreeRoot(t);
   return (LHS op RHS);
}
```

Evaluating an Expression: Another Viewpoint

What should **ExprTreeRoot(t)** return?

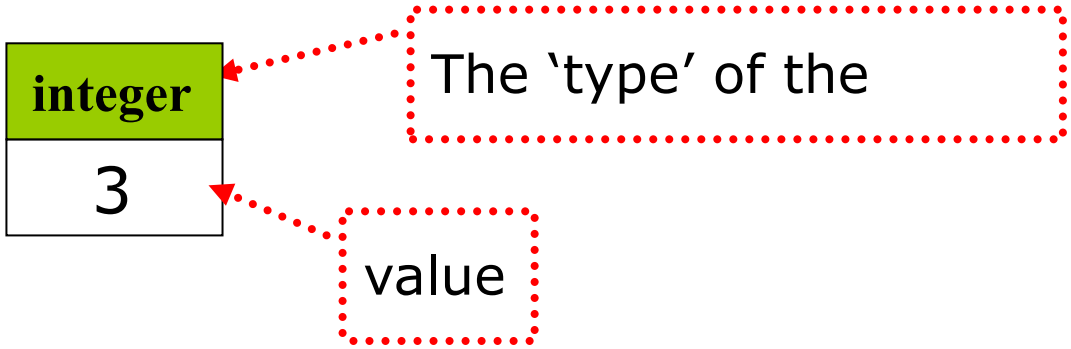
- an integer?
- an operator? (*But do you know how to return an operator, after all!*)



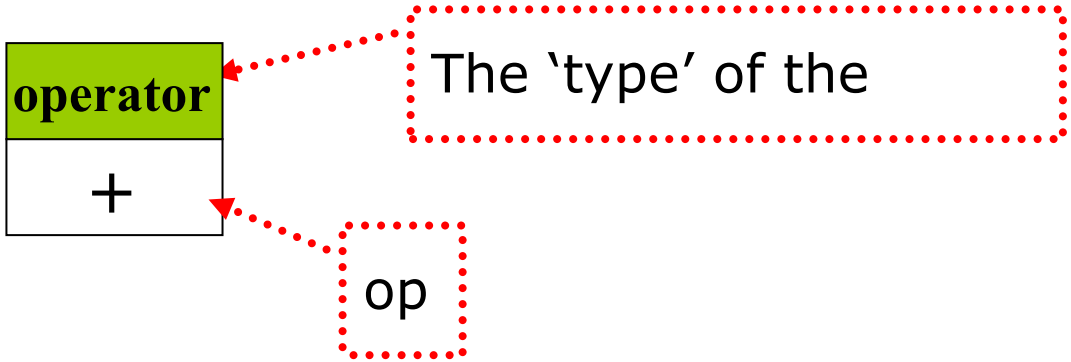
We see that a tree node can be an integer or an operator (For the time being, let's forget that a node can also be a variable.)

A Node of an Expression Tree
(Type is **ExprTreeNodeADT**)

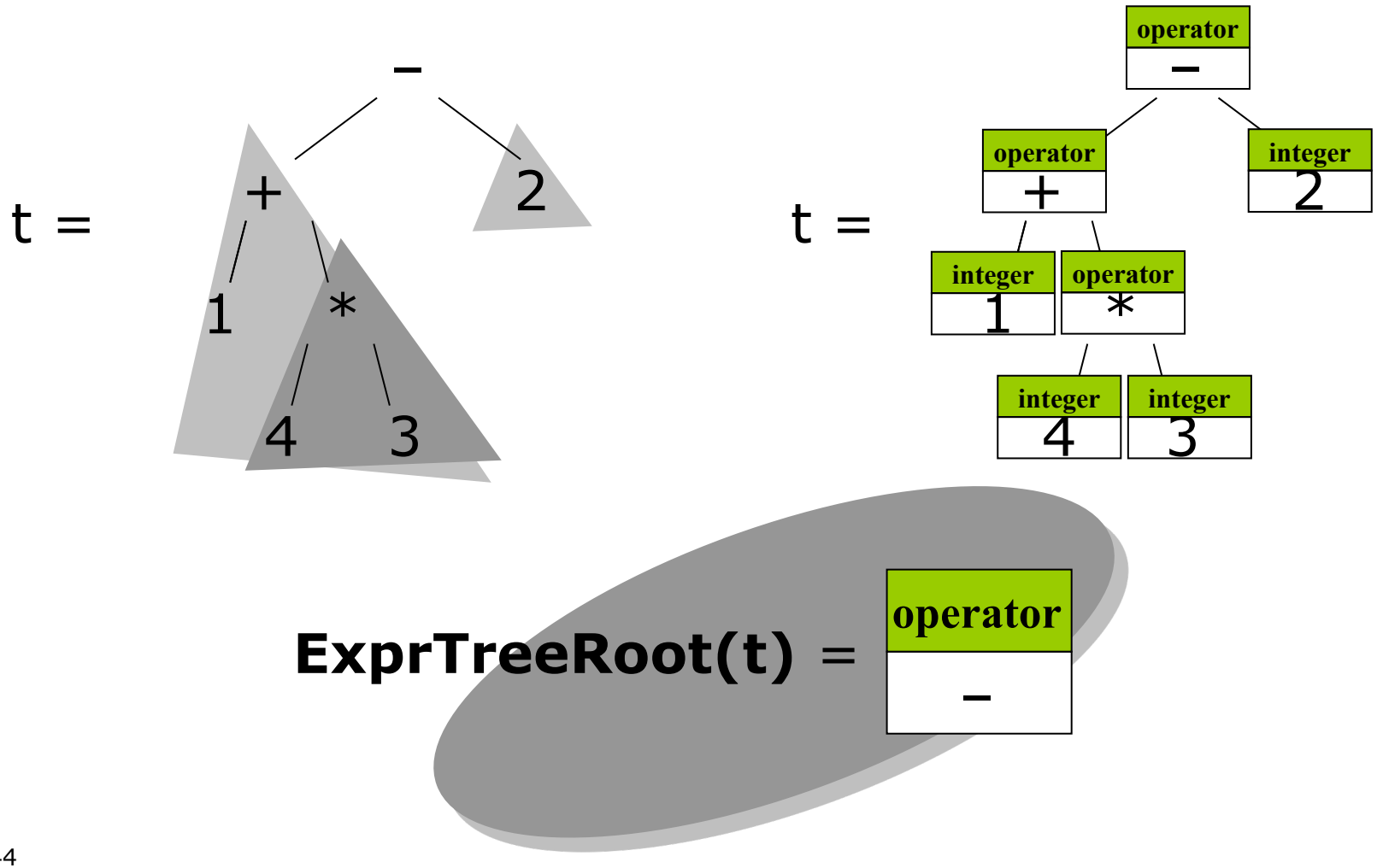
- If it is an integer:



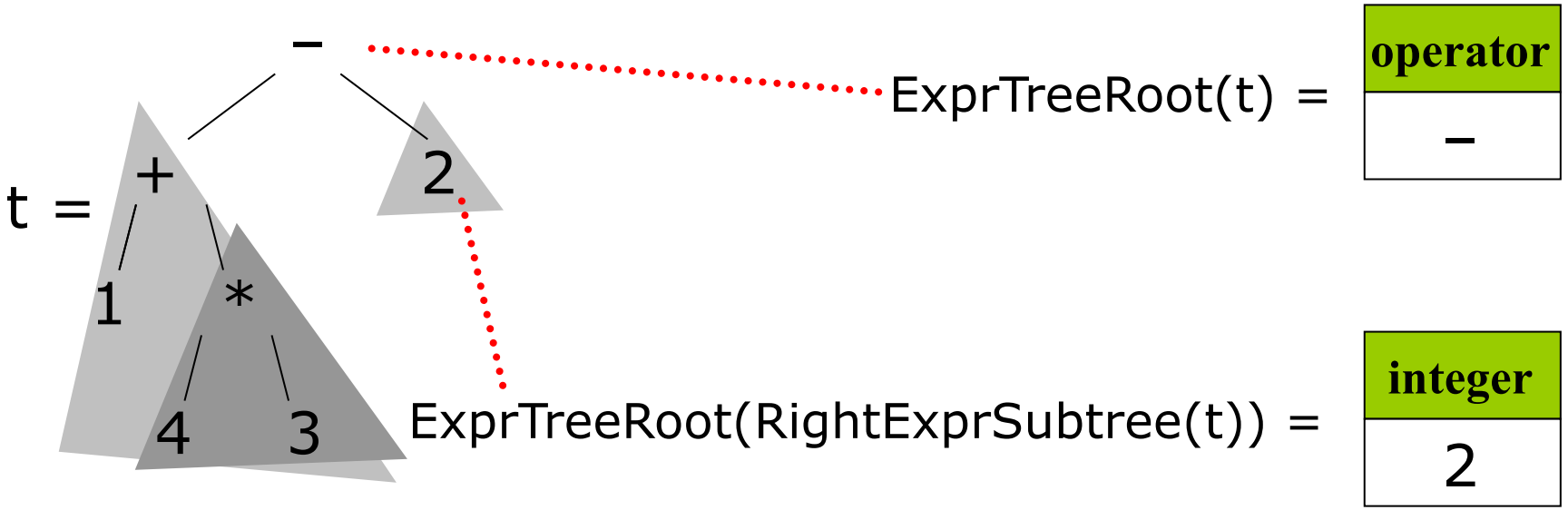
- If it is an operator:



Evaluating an Expression: Another Viewpoint

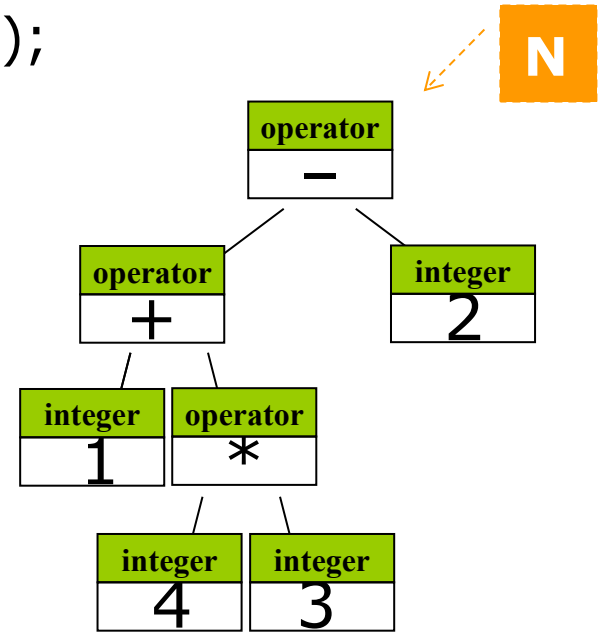


We introduce three more operations **NodeType**, **Node-Value** and **NodeOp**:



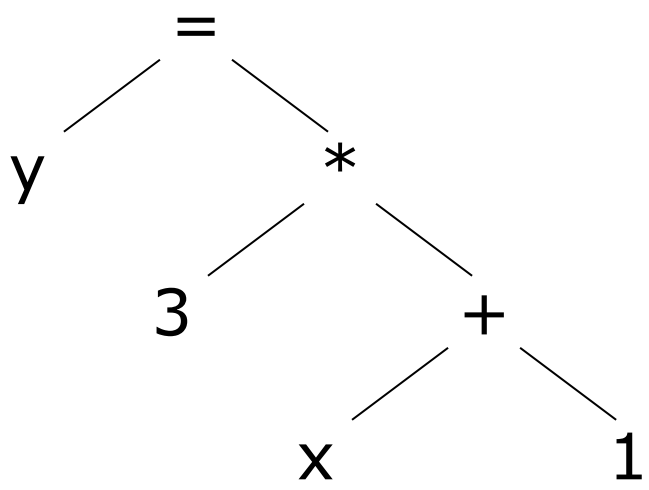
- **NodeType**(ExprTreeRoot(t)) returns **operator**
- **NodeOp**(ExprTreeRoot(t)) returns **'-'**
- **NodeType**(ExprTreeRoot(RightExprSubtree(t))) returns **integer**
- **NodeValue**(ExprTreeRoot(RightExprSubtree(t))) returns **2**

```
int Eval(ExprTreeADT t) {
   ExprTreeNodeADT N = ExprTreeRoot(t);
   switch (NodeType(N)) {
      case integer : return NodeValue(N);
      case operator : {
         int lhs = Eval(LeftExprSubtree(t));
         int rhs = Eval(RightExprSubtree(t));
         switch (NodeOp(N)) {
            case '+' : return lhs + rhs;
            case '–' : return lhs – rhs;
            case '*' : return lhs * rhs;
            case '/' : return lhs / rhs;
         }
   }
}
```



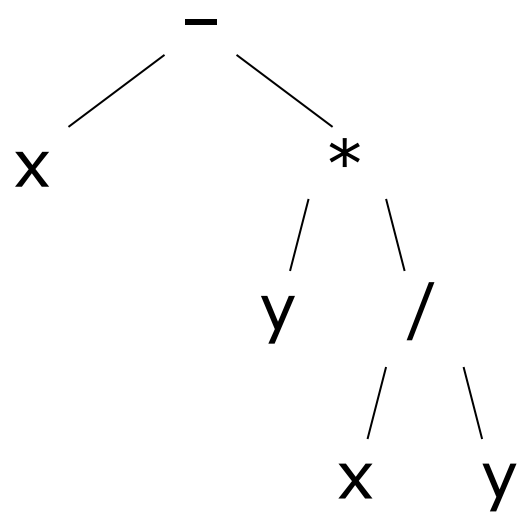
Traversing an Expression Tree

Now let’s consider how to reconstruct an expression from an expression tree.



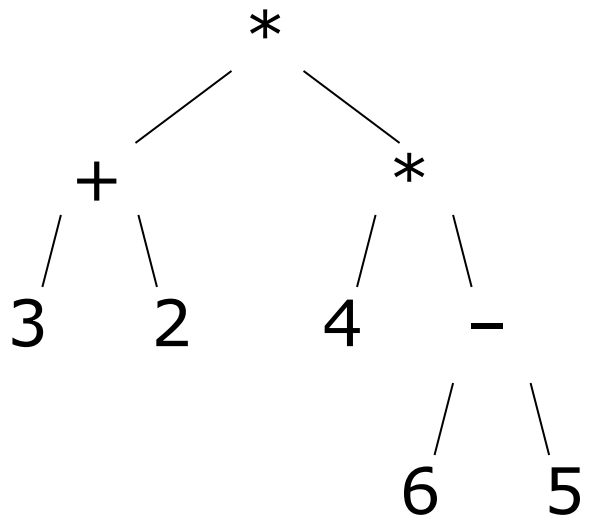
Traversing an Expression Tree

Now let’s consider this expression tree.



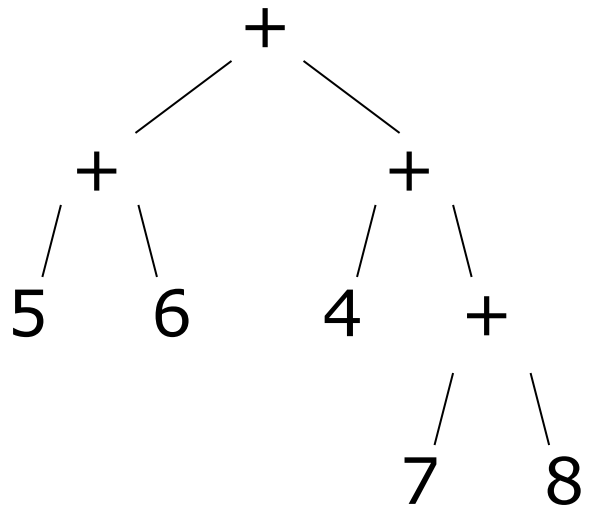
Traversing an Expression Tree

Now let’s consider this expression tree.



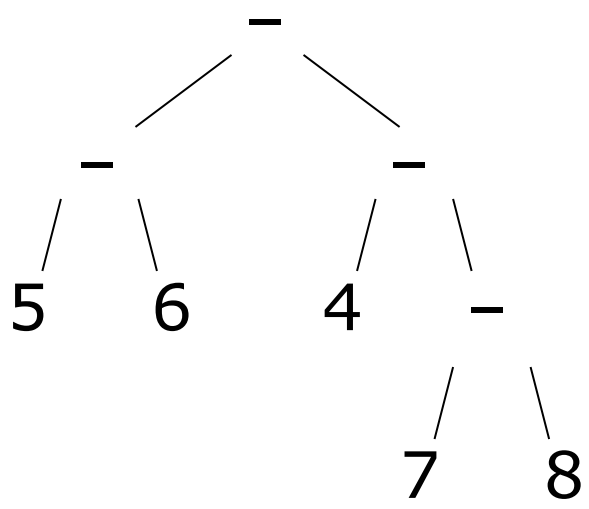
Traversing an Expression Tree

Now let’s consider this expression tree.



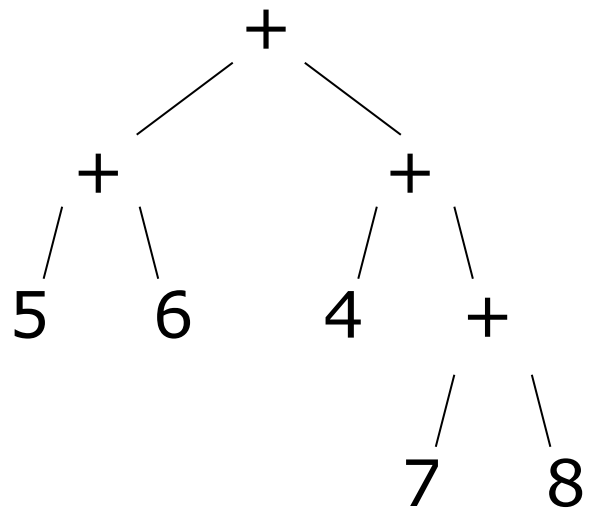
Traversing an Expression Tree

Now let’s consider this expression tree.

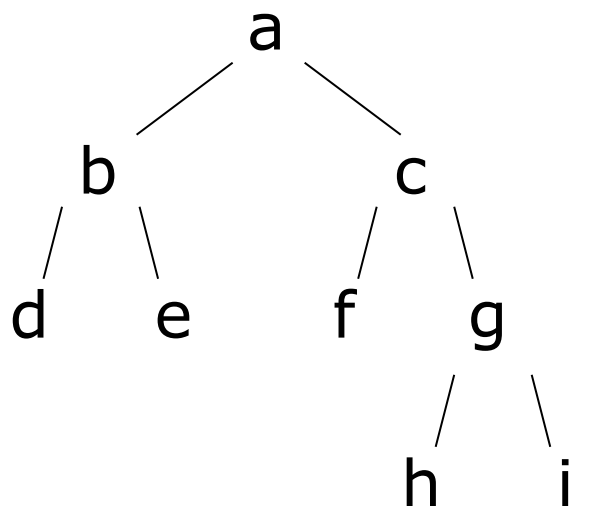


Traversing an Expression Tree

What we are doing is called an **'in-order' traversal of a tree**.



5+6+4+7+8

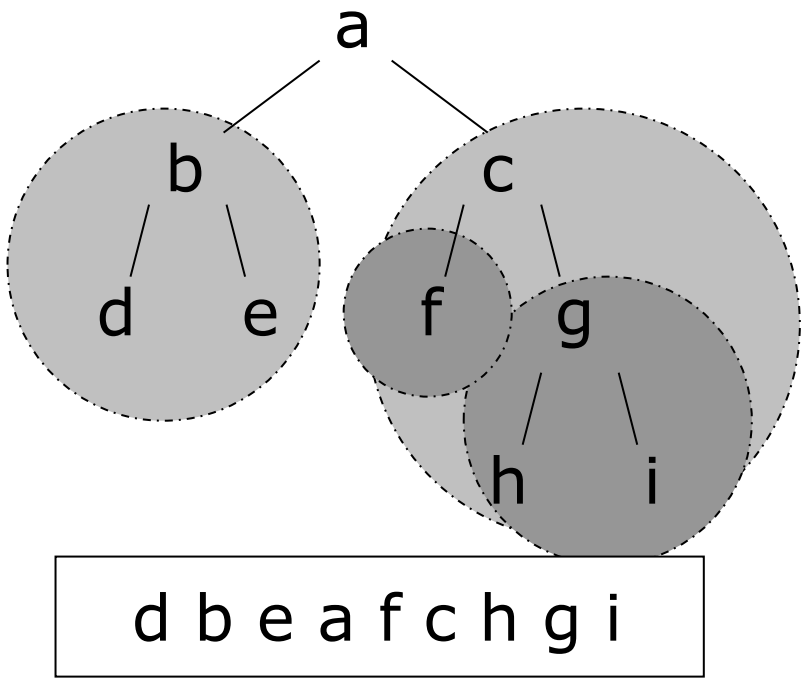


d b e a f c h g i

In-Order Traversal of a Tree

In an in-order traversal of a tree, we 'visit' the tree nodes in this order:

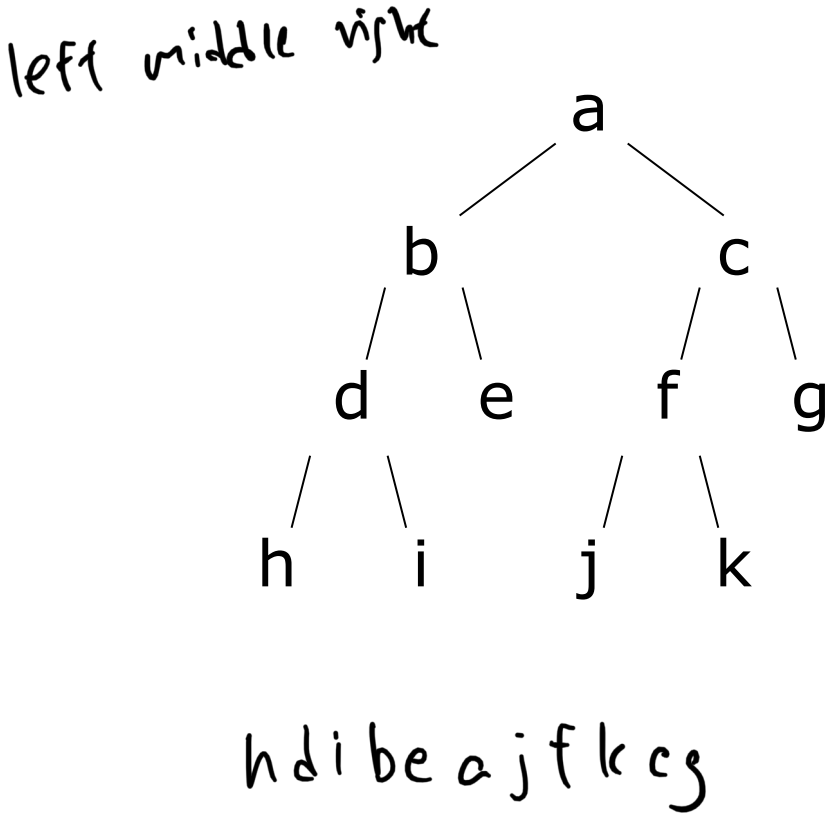
- We first visit the nodes in the left subtree, using in-order traversal.
- We then visit the root.
- We finally visit the nodes in the right subtree, using in-order traversal.



In-Order Traversal of a Tree

Consider an in-order traversal of this tree:

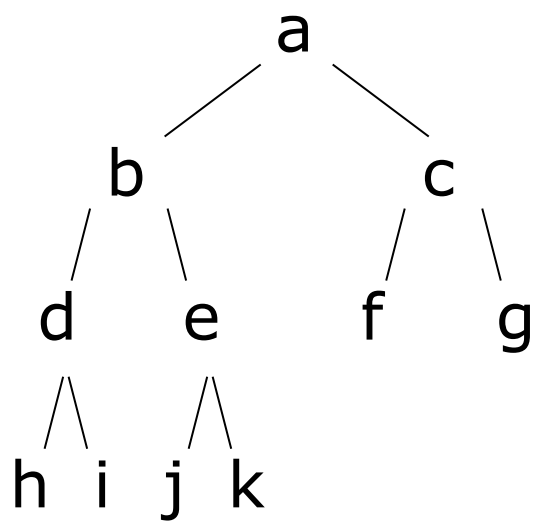
- We first visit the nodes in the left subtree, using in-order traversal.
- We then visit the root.
- We finally visit the nodes in the right subtree, using in-order traversal.



In-Order Traversal of a Tree

Consider an in-order traversal of this tree:

- We first visit the nodes in the left subtree, using in-order traversal.
- We then visit the root.
- We finally visit the nodes in the right subtree, using in-order traversal.

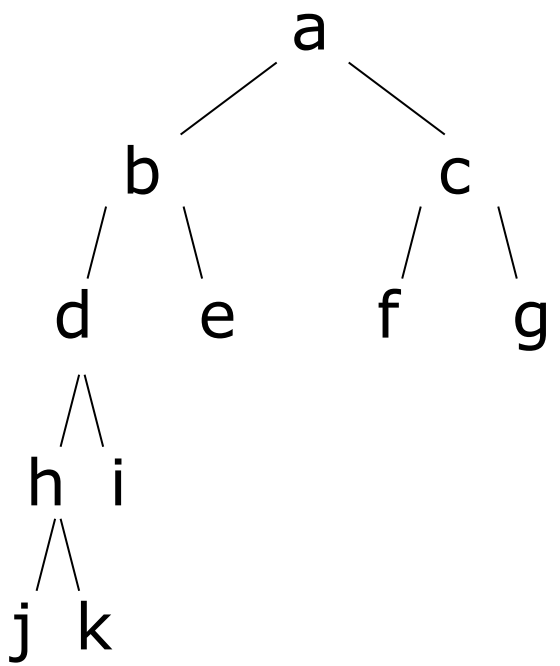


hdibjekafcg

In-Order Traversal of a Tree

Consider an in-order traversal of this tree:

- We first visit the nodes in the left subtree, using in-order traversal.
- We then visit the root.
- We finally visit the nodes in the right subtree, using in-order traversal.

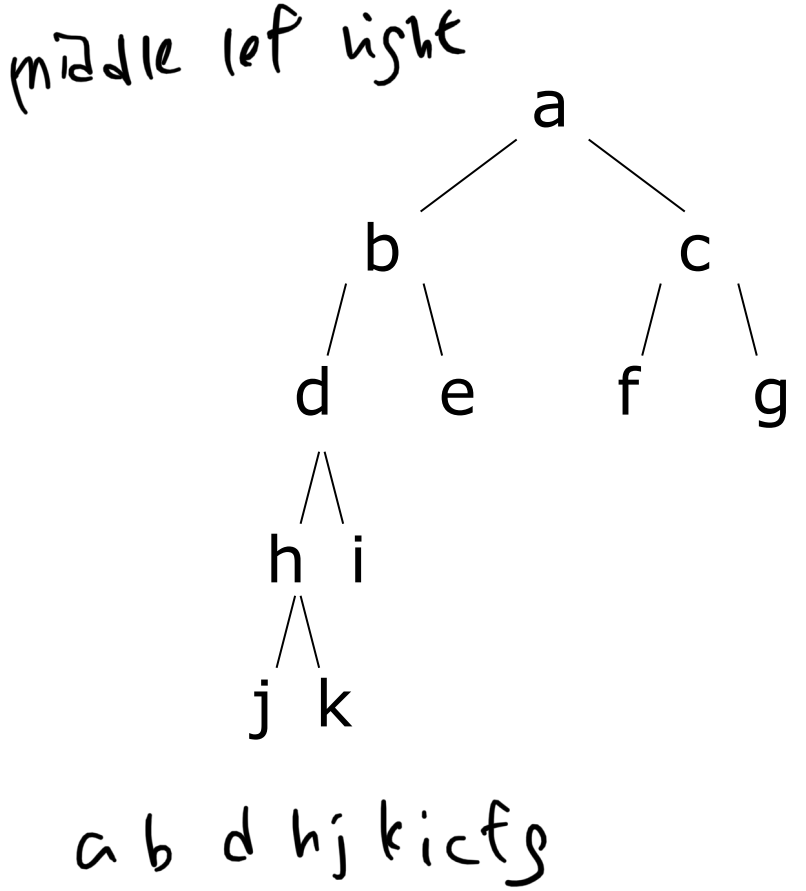


jhkdibeafcg

Pre-Order Traversal of a Tree

Similarly, we have the **'pre-order' traversal of a tree**:

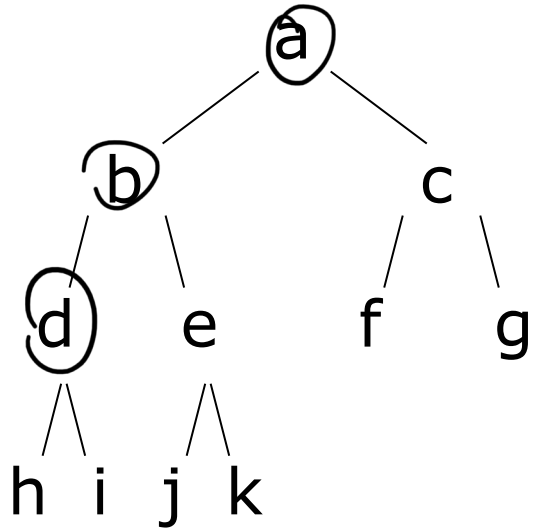
- We first visit the root.
- We then visit the nodes in the left subtree, using pre-order traversal.
- We finally visit the nodes in the right subtree, using pre-order traversal.



Pre-Order Traversal of a Tree

Consider a pre-order traversal of this tree:

- We first visit the root.
- We then visit the nodes in the left subtree, using pre-order traversal.
- We finally visit the nodes in the right subtree, using pre-order traversal.

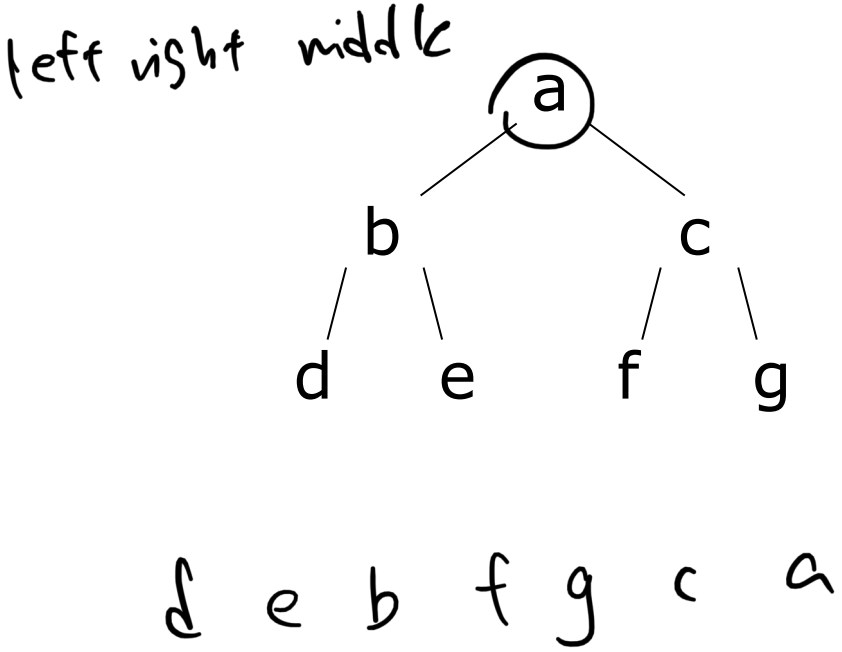


a b d h i e j k c f g

Post-Order Traversal of a Tree

Finally, we have the **'post-order' traversal of a tree**:

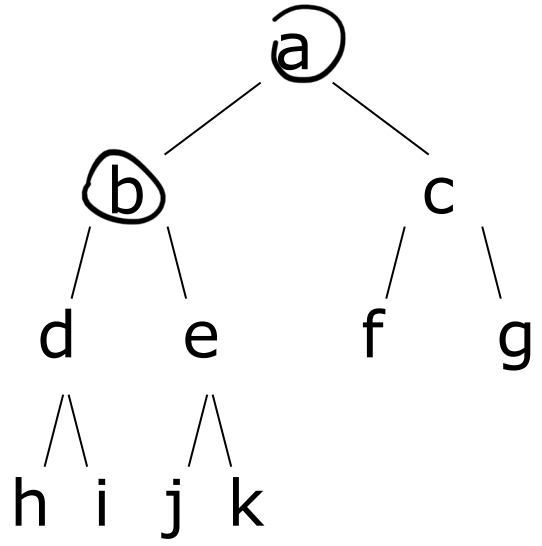
- We first visit the nodes in the left subtree, using post-order traversal.
- We then visit the nodes in the right subtree, using post-order traversal.



Post-Order Traversal of a Tree

Consider a post-order traversal of this tree:

- We first visit the nodes in the left subtree, using post-order traversal.
- We then visit the nodes in the right subtree, using post-order traversal.

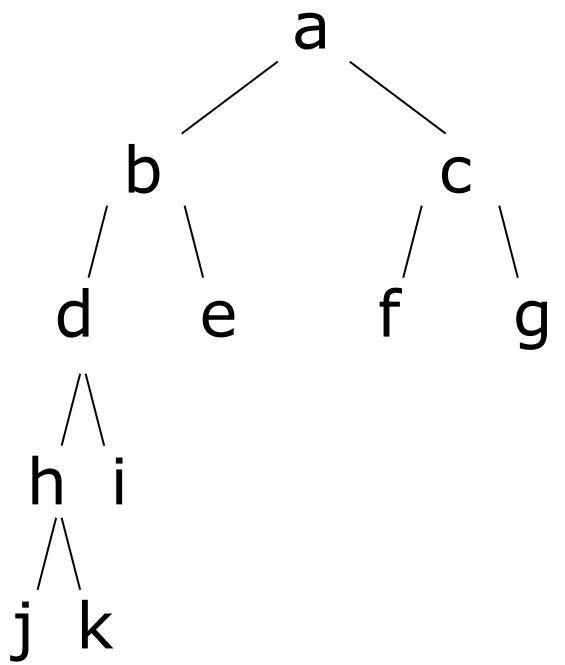


h i d j k e b f g c a

Post-Order Traversal of a Tree

Consider a post-order traversal of this tree:

- We first visit the nodes in the left subtree, using post-order traversal.
- We then visit the nodes in the right subtree, using post-order traversal.



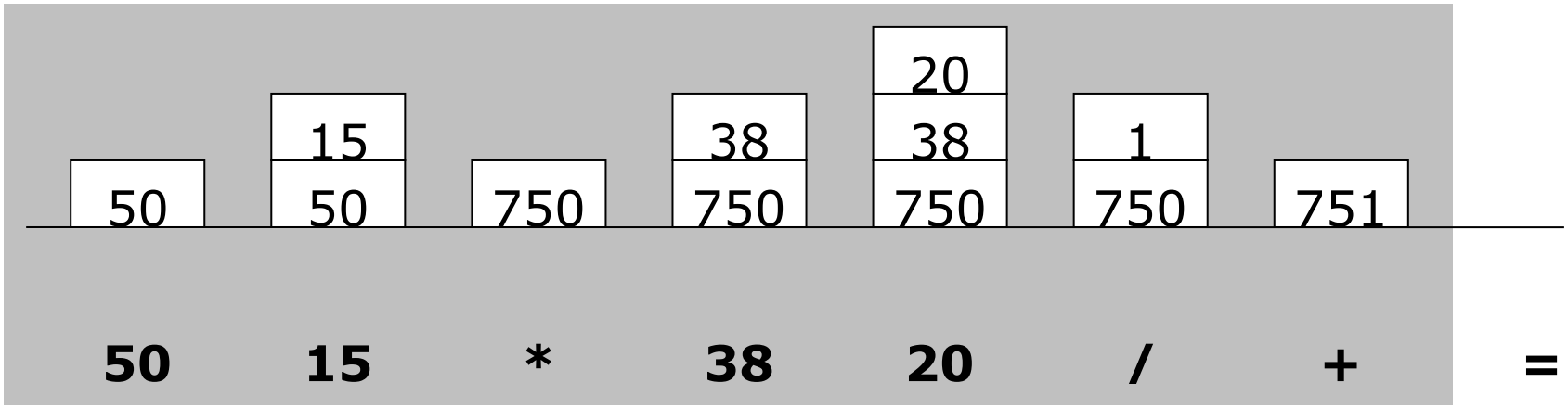
j k h i d e b f g c a

The Reverse Polish Notation RPN (Long Time Ago …)

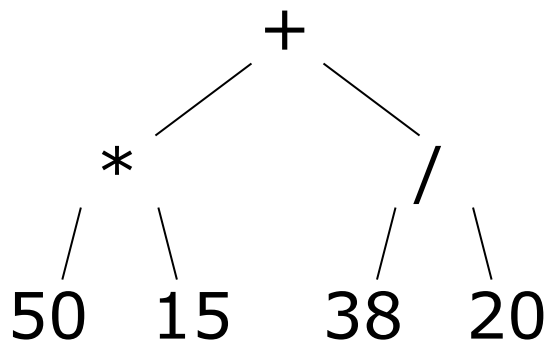
50 * 15 + 38 / 20

50 15 * 38 20 / + =

50↵
15↵
*↵
38↵
20↵
/↵
+↵
=↵



50 * 15 + 38 / 20



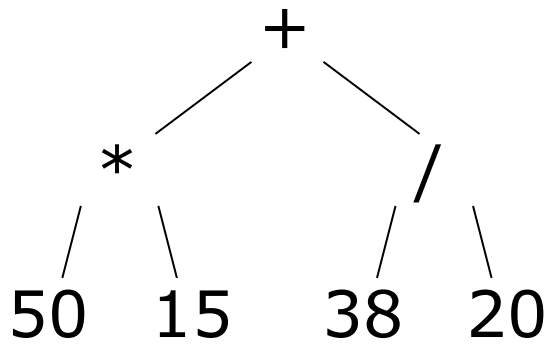
Post order (left right middle)

postfix form: **50 15 * 38 20 / +**

(operator put in the end) => operand operand operator

(left middle right) operand operator operand

In-order : 50 * 15 + 38 / 20 ← in-fix form



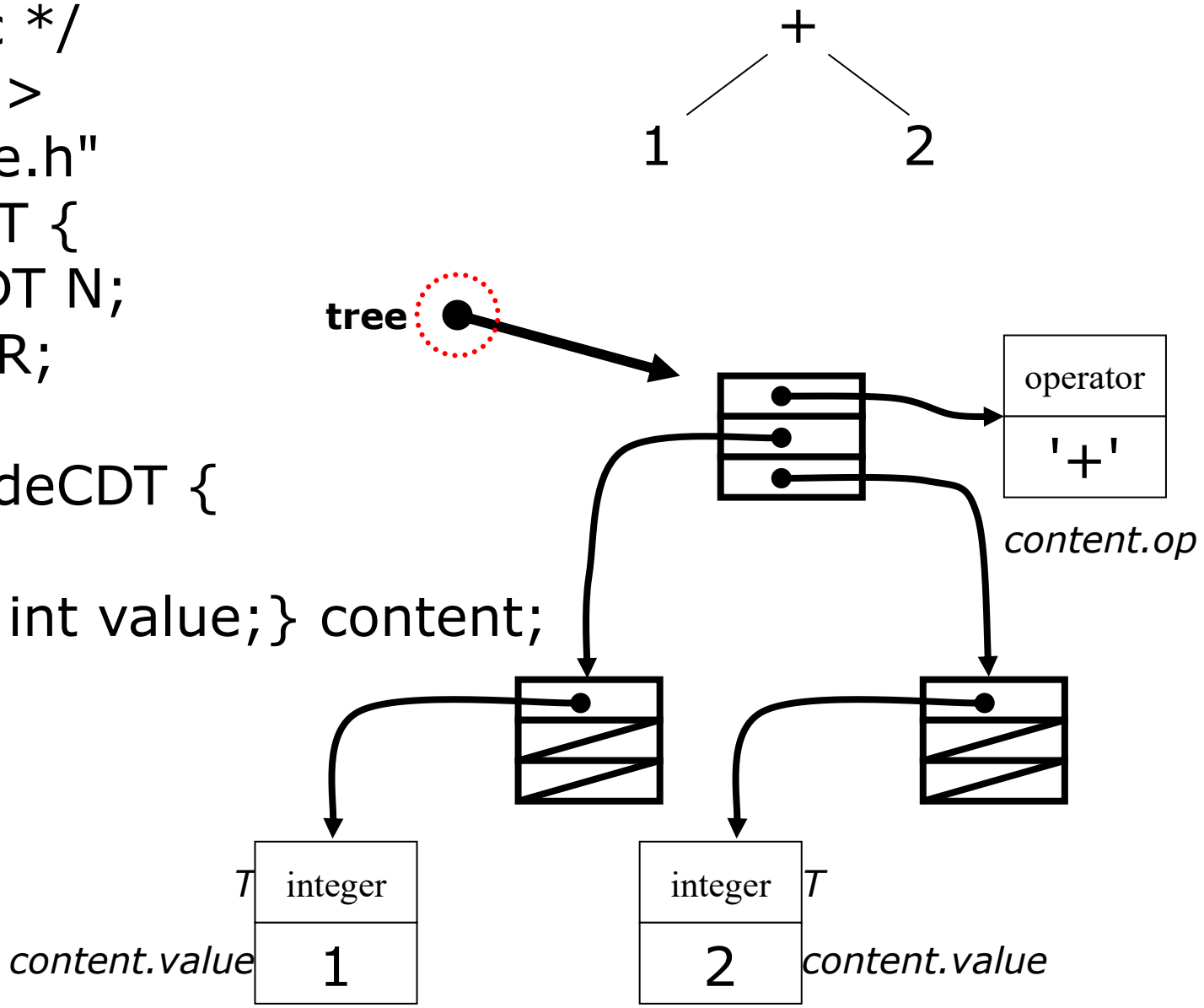
50 15 * 38 20 / +

Conclusion: the RPN of an expression can be obtained using a post-order traversal of the expression tree.

The Implementation

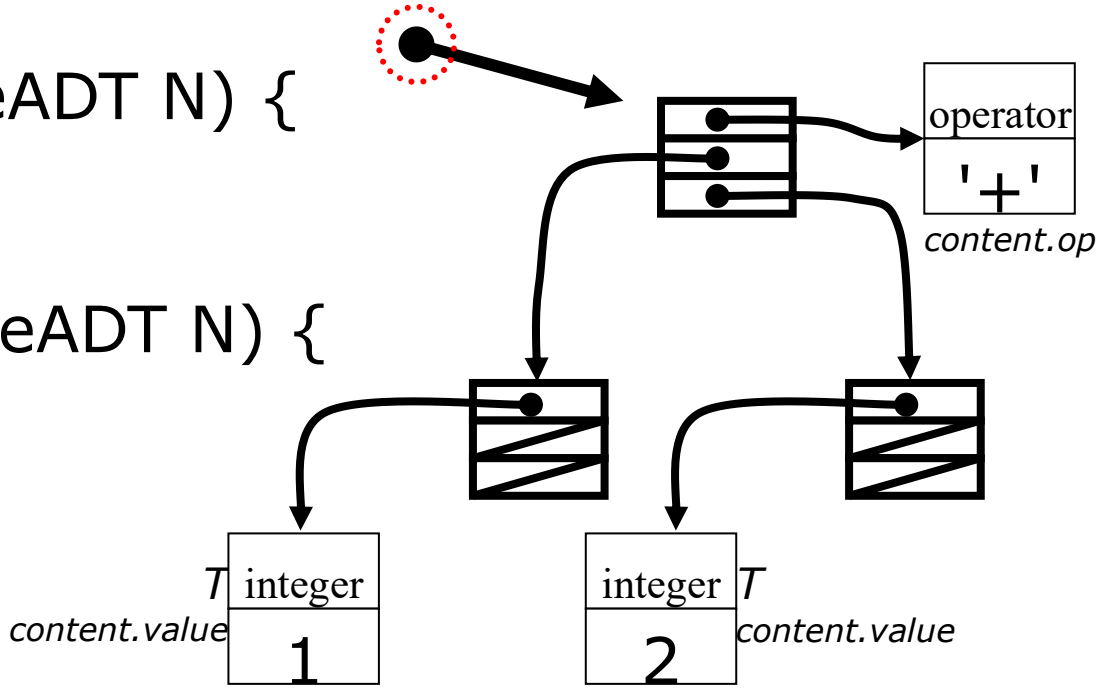
```
#include <stdbool.h>
typedef struct ExprTreeCDT * ExprTreeADT;
typedef struct ExprTreeNodeCDT *ExprTreeNodeADT;
enum nodetype {operator, integer};
typedef enum nodetype NodeTypeT;

NodeTypeT NodeType(ExprTreeNodeADT);
char NodeOp(ExprTreeNodeADT);
int NodeValue(ExprTreeNodeADT);
ExprTreeNodeADT ExprTreeRoot(ExprTreeADT);
ExprTreeADT EmptyExprTree(void);
ExprTreeADT NonemptyExprTree(ExprTreeNodeADT, ExprTreeADT, ExprTreeADT);
bool ExprTreeIsEmpty(ExprTreeADT);
ExprTreeADT LeftExprSubtree(ExprTreeADT);
ExprTreeADT RightExprSubtree(ExprTreeADT);
```

```
/* File: ExprTree.c */
#include <stdlib.h>
#include "ExprTree.h"
struct ExprTreeCDT {
   ExprTreeNodeADT N;
   ExprTreeADT L, R;
};
struct ExprTreeNodeCDT {
   NodeTypeT T;
   union {char op; int value;} content;
};
```



```
struct ExprTreeNodeCDT {
   NodeTypeT T;
   union {char op; int value;} content;
};
```

```
NodeTypeT NodeType(ExprTreeNodeADT N) {
   return N->T;
}
char NodeOp(ExprTreeNodeADT N) {
   return N->content.op;
}
int NodeValue(ExprTreeNodeADT N) {
   return N->content.value;
}
```



```
ExprTreeADT EmptyExprTree() {
  return (ExprTreeADT) NULL;
}

ExprTreeADT NonemptyExprSubtree(ExprTreeNode N,
        ExprTreeADT L, ExprTreeADT R) {
  ExprTreeADT t = (ExprTreeADT) malloc(sizeof(*t));
  t->N = N; t->L = L; t->R = R;
  return t;
}

bool ExprTreeIsEmpty(ExprTreeADT t) {
  return t == (ExprTreeADT) NULL;
}
```

```
ExprTreeNodeADT ExprTreeRoot(ExprTreeADT t) {
  return t->N;
}

ExprTreeADT LeftExprSubtree(ExprTreeADT t) {
  return t->L;
}

ExprTreeADT RightExprSubtree(ExprTreeADT t) {
  return t->R;
}
```