CS122 Algorithms and Data Structures

MW 11:00 am - 12:15 pm, MSEC 101

Instructor: Xiao Qin

Lecture 10: Binary Search Trees and Binary Expression Trees

Uses for Binary Trees…
-- Binary Search Trees

- Use for storing and retrieving information
- Insert, delete, and search faster than with a linked list
- Take advantage of log*n* height
- Idea: Store information in an ordered way (keys)

A Property of Binary Search Trees

- The key of the root is larger than any key in the left subtree
- The key of the root is smaller than any key in the right subtree
- Note: Duplicated keys are not allowed

Traversing Binary Search Trees

- There are three ways to traverse a binary tree
- Inorder Traversal:
  - Visit left subtree;
  - Visit root;
  - Visit right subtree;

Traversing Binary Search Trees (cont.)

- An Example:

```
void inorder_print(Node *root) {
    if (root != NULL) {
            inorder_print(root->left_child);
            count << root->info;
            inorder_print(root->right_child);
    }
}
```

Searching a Binary Search Trees

- Locate an element in a binary search tree

```
void search(Node *root, object key) {
    if (root == NULL) return -1;
    if (root->info == key) return root->info;
    else {
        if (key < root->info)
            search(root->left_child, key);
        else search(root->right_child, key);
    }
}
```

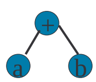
Inserting an Element in a Binary Search Trees

- Search for the Position in the tree where the element would be found
- Insert the element in the position
- Note: a newly inserted node is a leaf
- Running time is:
  - O(n) the worst case
  - O(lg*n*) if the tree is balanced

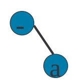
Uses for Binary Trees… -- Binary Expression Trees

- Binary trees are a good way to express arithmetic expressions.
  - The leaves are operands and the other nodes are operators.
  - The left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.

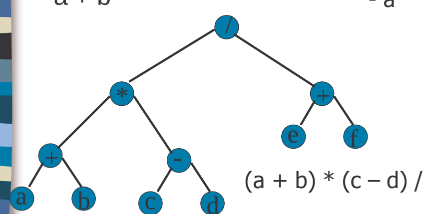
Binary Expression Trees: Examples



a + b



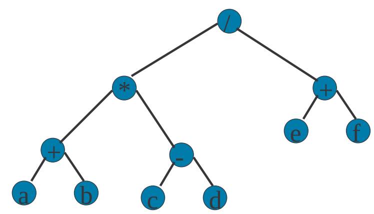
- a



(a + b) * (c – d) / (e + f)

Merits of Binary Tree Form

- Left and right operands are easy to visualize
- Code optimization algorithms work with the binary tree form of an expression
- Simple recursive evaluation of expression



Levels Indicate Precedence

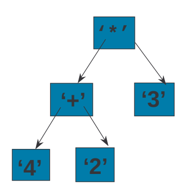
The levels of the nodes in the tree indicate their relative precedence of evaluation (**we do not need parentheses to indicate precedence).**

Operations at lower levels of the tree are evaluated later than those at higher levels.

The operation at the root is always the last operation performed.

11

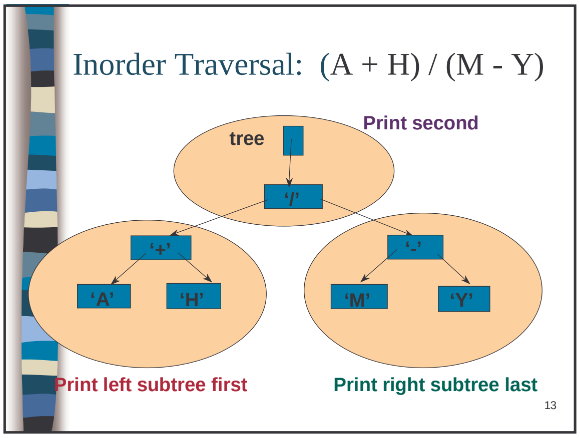
A Binary Expression Tree

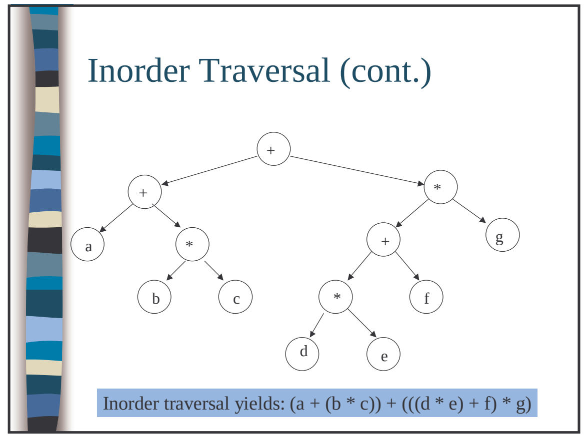


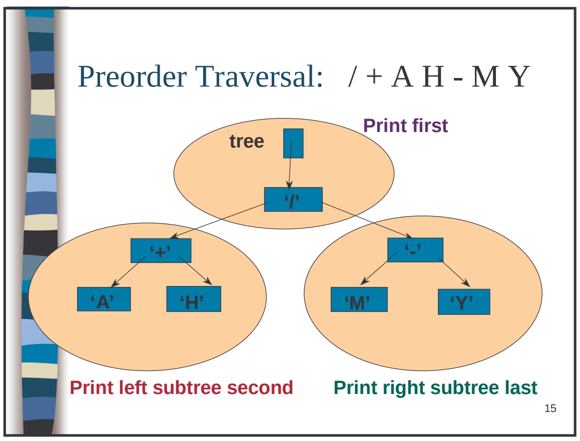
What value does it have?

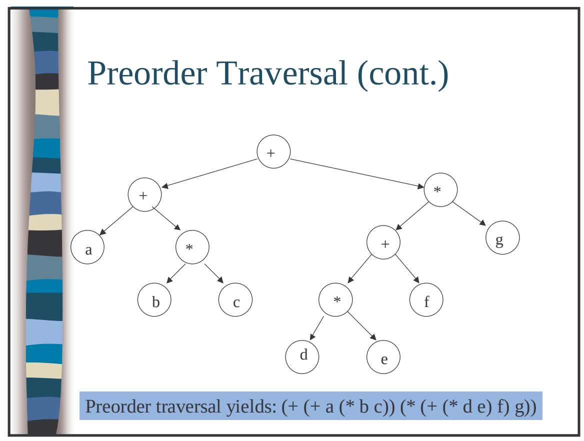
(4 + 2) * 3 = 18

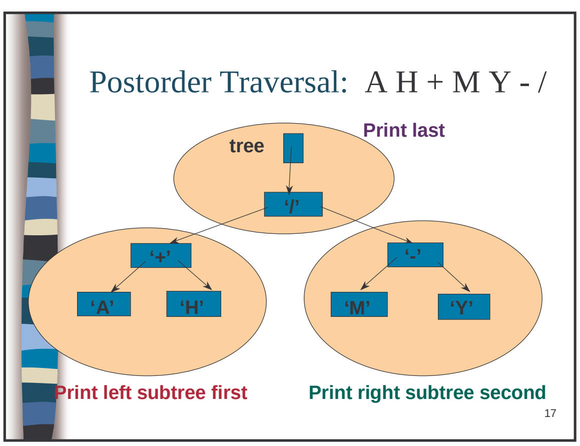
12

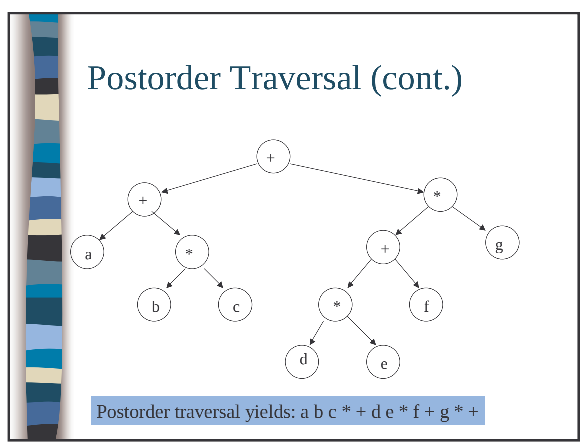

Inorder Traversal: (A + H) / (M - Y)
Print second
tree
'/'
'+'
'A'
'H'
'-'
'M'
'Y'
Print left subtree first
Print right subtree last
13


Inorder Traversal (cont.)
Inorder traversal yields: (a + (b * c)) + (((d * e) + f) * g)


Preorder Traversal: / + A H - M Y
Print first
tree
'/'
'+'
'A'
'H'
'-'
'M'
'Y'
Print left subtree second
Print right subtree last
15


Preorder Traversal (cont.)
Preorder traversal yields: (+ (+ a (* b c)) (* (+ (* d e) f) g))

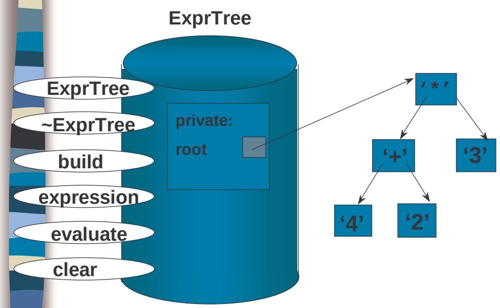

Postorder Traversal: A H + M Y - /
Print last
tree
'/'
'+'
'A'
'H'
'-'
'M'
'Y'
Print left subtree first
Print right subtree second
17


Postorder Traversal (cont.)
Postorder traversal yields: a b c * + d e * f + g * +

Traversals and Expressions

- Note that the postorder traversal produces the postfix representation of the expression.
- Inorder traversal produces the infix representation of the expression.
- Preorder traversal produces a representation that is the same as the way that the programming language Lisp processes arithmetic expressions!

class ExprTree



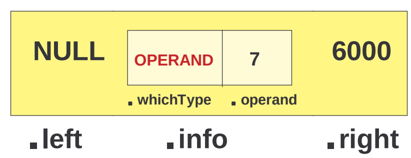
20

```
class   ExprTree  {
public:
    ExprTree ( );           // Constructor
    ~ExprTree ( );          // Destructor
    void   build ( );       // build tree from prefix expression
    void   expression ( ) const;
    // output expression in fully parenthesized infix form
    float  evaluate ( ) const;           // evaluate expression
    void   clear ( );                    // clear tree
    void   showStructure ( )  const;    // display tree

private:
    void   showSub( );
      . . .                    // recursive partners
    struct TreeNode *root;
};
```

Each node contains two pointers

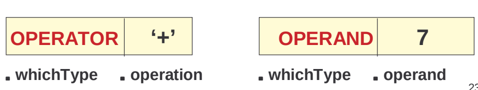
```
struct   TreeNode
{
    InfoNode    info ;        // Data member
    TreeNode*   left ;        // Pointer to left child
    TreeNode*   right ;       // Pointer to right child
};
```



22

InfoNode has 2 forms

```
enum  OpType  { OPERATOR,  OPERAND } ;
struct   InfoNode
{
    OpType      whichType;
    union                                    // ANONYMOUS union
    {
         char    operation ;
         int     operand ;
    }
};
```



23

```
int   Eval ( TreeNode*  ptr )

{     switch  ( ptr->info.whichType )
      {
          case OPERAND :  return  ptr->info.operand ;
          case OPERATOR :
             switch ( tree->info.operation )
             {
                case  '+'  :  return  ( Eval ( ptr->left ) + Eval ( ptr->right ) ) ;

                case  '-'  :  return  ( Eval ( ptr->left )  -  Eval ( ptr->right ) ) ;

                case  '*'  :  return  ( Eval ( ptr->left )  *  Eval ( ptr->right ) ) ;

                case  '/'  :  return  ( Eval ( ptr->left )  /  Eval ( ptr->right ) ) ;
             }
      }
}
```

24

Constructing an Expression Tree

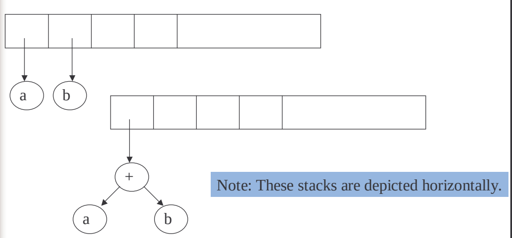
- There is a simple O(*N*) stack-based algorithm to convert a postfix expression into an expression tree.
- Recall we also have an algorithm to convert an infix expression into postfix, so we can also convert an infix expression into an expression tree without difficulty (in O(*N*) time).

Expression Tree Algorithm

- Read the postfix expression one symbol at at time:
  - If the symbol is an operand, create a one-node tree and push a pointer to it onto the stack.
  - If the symbol is an operator, pop two tree pointers T1 and T2 from the stack, and form a new tree whose root is the operator, and whose children are T1 and T2.
  - Push the new tree pointer on the stack.

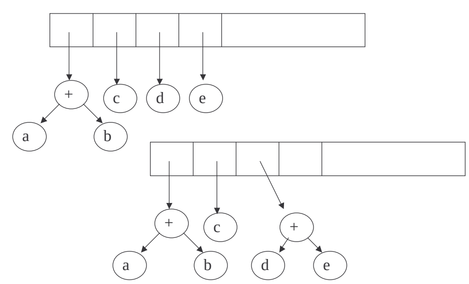
Example

a b + :



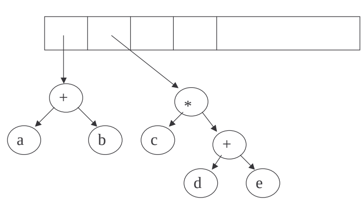
Example

a b + c d e + :



Example

a b + c d e + * :



Example

a b + c d e + * * :