COMP 250

Lecture 25

binary trees, expression trees

March 10, 2022
Binary tree: each node has *at most* two children.
Maximum number of nodes in a binary tree?

\[ n = 1 + 2 + 4 + 8 + \cdots 2^h \]

\[ = 1 + x + x^2 + x^3 + \cdots x^h = \frac{x^{h+1} - 1}{x - 1}, \text{ where } x = 2 \]
Maximum number of nodes in a binary tree?

\[ n = 1 + 2 + 4 + 8 + \cdots 2^h = 2^{h+1} - 1 \]

\[ = 1 + x + x^2 + x^3 + \cdots x^h = \frac{x^{h+1} - 1}{x-1}, \text{ where } x = 2 \]
Minimum number of nodes in a binary tree?

\[ n = h + 1 \]
Implementation in Java

class BinaryTree<T>{
    BTNode<T> root;
    
    class BTNNode<T>{
        T e;
        BTNode<T> leftchild;
        BTNode<T> rightchild;
    }
}
Recall: depth first tree traversal

```
// pre-order

depthFirst(root){
    visit root
    for each child of root
        depthFirst(child)
}
```

```
// post-order

depthFirst(root){
    for each child of root
        depthFirst(child)
    visit root
}
```

We write these slightly differently for a binary tree (next slide).
preorderBT (root) {
    if (root is not null) {
        visit root
        preorderBT (root.left)
        preorderBT (root.right)
    }
}

postorderBT (root) {
    if (root is not null) {
        postorderBT (root.left)
        postorderBT (root.right)
        visit root
    }
}

inorderBT (root) {
    if (root is not null) {
        inorderBT (root.left)
        visit root
        inorderBT (root.right)
    }
}
Example

Pre order:  \( a \ b \ d \ e \ c \ f \ g \)

In order:  \( d \ e \ b \ a \ f \ c \ g \)

Post order:  \( e \ d \ b \ f \ g \ c \ a \)
We often write expressions such as $3 + 4 \times 2$. We can write and evaluate such expressions using trees. There are two ways to do so for this example.

- $(3 + 4) \times 2$
- $3 + (4 \times 2)$

Expression Tree
My Windows calculator says $3 + 4 \cdot 2 = 14$.

$(3 + 4) \cdot 2 = 14$.

Whereas....
if I google "3+4*2", I get $11$.  

$3 + (4\cdot2) = 11$. 
We can make expressions using binary operators +, -, *, /, ^

e.g. \[ a - \frac{b}{c} + d \times \left( e^{f^g} \right) \]

^ is exponentiation: \[ e^{f^g} \] means \[ e^{(f^g)} \]

Operator precedence ordering makes brackets unnecessary.

\[ (a - (b / c)) + (d \times (e^{(f^g)})) \]

We don’t consider unary operators e.g. \[ 3 + -4 = 3 + (-4) \]
If we traverse an expression tree, and *print out* the node label, what expression is printed out?

preorder traversal gives: \( + - a / b c * d ^ e ^ f g \)

inorder traversal gives: \( a - b / c + d * e ^ f ^ g \)

postorder traversal gives: \( a b c / - d e f g ^ ^ * + \)
If we traverse an expression tree, and \textit{print out} the node label, what expression is printed out?

```
"pre-fix" expression:   + - a / b c * d ^ e ^ f g
"in-fix" expression :   a - b / c + d * e ^ f ^ g
"post-fix" expression :  a b c / - d e f g ^ ^ * +
```
ASIDE: “Formal language” for prefix expressions

```
baseExp   =   a | b | c | d    ....    |   z
op        =   + | - | * | / | ^
preExp    =   baseExp   | op   preExp   preExp
```

where | means “or”.

This gives you a hint of how programming languages are formally defined. e.g. COMP 330 Theory of Computation.
ASIDE: “Formal language” for expressions

\[
\begin{align*}
\text{baseExp} & = a \mid b \mid c \mid d \mid \ldots \text{ etc} \\
\text{op} & = + \mid - \mid * \mid / \mid ^\
\text{preExp} & = \text{baseExp} \mid \text{op} \text{ preExp} \text{ preExp} \\
inExp & = \text{baseExp} \mid \text{inExp} \text{ op} \text{ inExp} \\
\text{postExp} & = \text{baseExp} \mid \text{postExp} \text{ postExp} \text{ op}
\end{align*}
\]

Use only one.
Prefix expressions are called “Polish Notation”. (after Polish logician Jan Łukasiewicz 1920’s)

Postfix expressions are called “Reverse Polish notation” (RPN)
Prefix expressions are called “Polish Notation”
(after Polish logician Jan Lukasiewicz 1920’s)

Postfix expressions are called “Reverse Polish notation” (RPN)

Some calculators (esp. Hewlett Packard) require users to input expressions using RPN.

Calculate $3 + 4 \times 2$
which is $3 \ 4 \ 2 \ \ast \ +$ in RPN

3 <enter>
4 <enter>
2
$\ast \ \rightarrow \ \text{yields 8}$
$+ \ \rightarrow \ \text{yields 11}$

No “=” symbol on keyboard.
There are lots of youtube videos showing how to use RPN calculators, e.g. this video.

On the previous slide, where we had $3+4\times2$, ... with a real RPN calculator, you would first do the $4 \times 2$ and then add 3.
Suppose we are given an expression tree. How can we evaluate the corresponding expression?

Hint: traverse the tree. But how?

Here we assume the leaves have known values.
Use a **postorder traversal**: 

```java
evalExpressionTree(root) {
    if (root is a leaf)  // root is a number
        return value
    else{
        // root is an operator
        Evaluate left and right subtrees and then combine results.
    }
}
```
Use a **postorder traversal**:

```java
int evalExpressionTree(TreeNode root)
{
    if (root is a leaf) // root is a number
        return value
    else{ // root is an operator
        op = root.element
        firstOperand = evalExpressionTree ( root.leftchild )
        secondOperand = evalExpressionTree( root.rightchild)
        return evaluate(op, firstOperand, secondOperand)
    }
}
```

It is postorder because we need to evaluate the children before we can evaluate the node.
Suppose we are just given a postfix expression. How can we evaluate it? Data structure? Algorithm?

E.g. \(a \ b \ c \ / \ - \ d \ e \ f \ g \ ^\ ^\ +\)

Read symbols from left to right. Use a stack. (Next slides)
Example:

\[ a \ b \ c \ / \ - \ d \ e \ f \ g \ ^\ ^\ * \ + \]

This expression tree is not given. It is shown here so that you can visualize the expression more easily.
Example:

\[ a b c / - d e f g ^ ^ * + \]

This expression tree is not given. It is shown here so that you can visualize the expression more easily.
Example:

\[
\text{a b c} / \ - \ d \ e \ f \ g \ ^\wedge \ ^\wedge \ * \ +
\]

```
stack
  over
time
```

```
| a    | a    |
| a b   | a b c |
|       |      |
```

```
+    *
\_  \_
\- \ /  \
  a  b  c
d  e  A
f  g
```
Use a stack to evaluate postfix expression:

\[ \text{a b c/ - d e f g ^ ^ * +} \]

We don’t push the operator onto the stack. Instead we pop value twice, evaluate, and push the result.
Stack over time:

```
a b c / - d e f g ^ ^ * +
```

Now there is one value on the stack.
\[ a \ b \ c / - d \ e \ f \ g ^ ^ * + \]

Now there are five values on the stack.
Use a stack to evaluate postfix expression:

\[ a \ b \ c / - \ d \ e \ f \ g ^ ^ * + \]

<table>
<thead>
<tr>
<th>Stack</th>
<th>Over time</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a b</td>
<td></td>
</tr>
<tr>
<td>a b c</td>
<td></td>
</tr>
<tr>
<td>a ( b c / )</td>
<td></td>
</tr>
<tr>
<td>( a ( b c / ) - )</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>( a ( b c / ) - ) d e f g</td>
<td></td>
</tr>
<tr>
<td>( a ( b c / ) - ) d e ( f g ^ )</td>
<td></td>
</tr>
</tbody>
</table>

Now there are four values on the stack.
Use a stack to evaluate postfix expression:
\[ a \ b \ c / - \ d \ e \ f \ g ^ ^ * + \]

Now there are three values on the stack.
Use a stack to evaluate postfix expression:

\[ a \ b \ c / - \ d \ e \ f \ g ^ ^ * + \]

```
stack
over
time
```

```
( a ( b c / ) - )
( a ( b c / ) - ) d e f g
( a ( b c / ) - ) d e ( f g ^ )
( a ( b c / ) - ) d ( e ( f g ^ ) ^ )
( a ( b c / ) - ) ( d ( e ( f g ^ ) ^ ) * )
```

Now there are two values on the stack.
Use a stack to evaluate postfix expression:

```
a b c / - d e f g ^ ^ * +
```

```
stack
over time
```

```plaintext
a
a b
a b c
a ( b c / )
(a ( b c / ) - )

stack:

```plaintext
(a ( b c / ) - ) d e f g
(a ( b c / ) - ) d e ( f g ^ )
(a ( b c / ) - ) d ( e ( f g ^ ) ^ )
(a ( b c / ) - ) ( d ( e ( f g ^ ) ^ ) * )
(( a ( b c / ) - ) ( d ( e ( f g ^ ) ^ ) * ) + )
```

One value on the stack (the result).  
*Note this corresponded to a postorder traversal of an expression tree.*
Algorithm: Use a stack to evaluate a postfix expression

Let expression be a list of “tokens”.

s = empty stack
cur = first token of expression list
while (cur != null) {
    if (cur is a base expression) // value i.e. variable or number
        s.push( cur )
    else { // cur is an operator
        operand2 = s.pop()
        operand1 = s.pop()
        operator = cur.element // for clarity only
        s.push( evaluate(operand1, operator, operand2) )
    }
    cur = cur.next
}
Algorithm: Use a stack to evaluate a postfix expression

Let expression be a list of “tokens”.

s = empty stack
cur = first token of expression list
while (cur != null){
  if (cur is a base expression)
    s.push(cur)
  else{
    // cur is an operator
    operand2 = s.pop()
    operand1 = s.pop()
    op = cur  // not necessary (but easier to read)
    s.push(evaluate(op, operand1, operand2))
  }
  cur = next token in expression list
}  // terminates with result on the stack
As we just saw, **postfix expressions** *without brackets* are easy to evaluate. A similar algorithm works for **pre-fix expressions**. Read the expression from right to left and swap order of two operands when evaluating.

**Infix expressions** (with or without brackets) are trickier to evaluate, since you need to incorporate precedence ordering rules for the different operands. You can convert infix to postfix using the following:


As you know, the Java language expects expressions to be infix. The Java compiler converts infix expressions to postfix. At runtime, the JVM then uses a stack to evaluate the postfix expression (much simpler and faster).
## Coming up...

<table>
<thead>
<tr>
<th>Lectures</th>
<th>Tutorial + Assessments</th>
</tr>
</thead>
</table>
| Mon. March 14  
  Binary Search Trees | Assignment 3  
  due Wed. March 16 |
| Wed & Fri. March 16 & 18  
  Heaps | Quiz 4 (lectures 20-25)  
  Fri. March 18 |