COMP 250

Lecture 19

binary trees, expression trees

Oct. 24, 2016
Binary tree:
each node has at most two children.
Maximum number of nodes in a binary tree?

Height $h$ (e.g. 3)
Maximum number of nodes in a binary tree?

$n = 1 + 2 + 4 + 8 + 2^h = 2^{h+1} - 1$
Minimum number of nodes in a binary tree?

$$n = h + 1$$

Height $h$
(e.g. 3)
class BTree<T> {
    BTNode<T> root;
}

class BTNode<T> {
    T e;
    BTNode<T> leftchild;
    BTNode<T> rightchild;
}
Binary Tree Traversal (depth first)

Rooted tree (last lecture)

preorder(root){
    if (root is not empty){
        visit root
        for each child of root
            preorder(child)
    }
}

Binary tree
Binary Tree Traversal (depth first)

Rooted tree (last lecture)

```plaintext
preorder(root){
    if (root is not empty){
        visit root
        for each child of root
            preorder(child)
    }
}
```

Binary tree

```plaintext
preorderBT(root){
    if (root is not empty){
        visit root
        preorderBT(root.left)
        preorderBT(root.right)
    }
}
```
preorderBT (root){
    if (root is not empty){
        visit root
        preorderBT( root.left )
        preorderBT( root.right )
    }
}

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

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

Pre order: abdecfg
In order: debafcg
Post order: edbfgca
Example of binary tree:
“Syntax (sub)Tree” of Assignment 2

```plaintext
statement = if boolean then statement else statement end
```

```plaintext
statement = if boolean then statement else statement end
```
The *statement* nodes form a binary (sub)tree.
Expression Tree

e.g. 3 + 4 * 2

(3 + 4) * 2

3 + (4 * 2)
My Windows calculator says $3 + 4 \times 2 = 14$.

Why? $(3 + 4) \times 2 = 14$.

Whereas.... if I google “3+4*2”, I get 11.

$3 + (4\times2) = 11$. 
Example of expression tree

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

\(^\text{^ is exponentiation}\)

We consider binary operators only
\[ \text{e.g. we don't consider } 3 + -4 = 3 + (-4) \]

Precedence ordering makes brackets unnecessary.
\[ \text{i.e. } (a - (b / c)) + (d * (e ^ (f ^ g))) \]
Expression Tree

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

Internal nodes are operators. Leaves are numbers.
Infix, prefix, postfix expressions

Infix: $a* b$

Prefix: $* a b$

Postfix: $a b *$
Infix, prefix, postfix expressions

\[
\begin{align*}
\text{baseExp} & \quad = \quad \text{variable} \mid \text{integer} \\
\text{op} & \quad = \quad + \mid - \mid \ast \mid / \mid ^
\text{inExp} & \quad = \quad \text{baseExp} \mid \text{inExp} \ \text{op} \ \text{inExp} \\
\text{preExp} & \quad = \quad \text{baseExp} \mid \ \text{op} \ \text{preExp} \ \text{prefExp} \\
\text{postExp} & \quad = \quad \text{baseExp} \mid \ \text{postExp} \ \text{postExp} \ \text{op}
\end{align*}
\]
If we traverse an expression tree, in which order do we ‘visit’ nodes?

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

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

postorder traversal gives postfix expression: \( a b c / - d e f g ^ ^ * + \)
If we were given an expression tree, then how would we evaluate the expression?
If we were given an expression tree, then we could evaluate it using a postorder traversal:

```java
evalExpTree(root){
  if (root is a leaf)    // root is a number
    return value
  else{                 // the root is an operator
    firstOperand = evalExpTree(root.leftchild)
    secondOperand = evalExpTree(root.rightchild)
    return evaluate(firstOperand, root, secondOperand)
  }
}
```

However, in practice we are not given an expression tree.
How to evaluate expressions?

Infix expressions are awkward to evaluate because of precedence ordering.

ASIDE: One can convert an infix expression to a postfix expression: [http://wcipeg.com/wiki/Shunting_yard_algorithm](http://wcipeg.com/wiki/Shunting_yard_algorithm)
Details omitted here. For your interest only.

We next show how to evaluate a postfix expression using a stack.
Use a stack to evaluate postfix expression:

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

```
stack over time

a
ab
abc
a(bc/)
( a(bc/) - )
( a(bc/) - ) d
( a(bc/) - ) d e
( a(bc/) - ) d e f
( a(bc/) - ) d e f g
( a(bc/) - ) d e (f g ^)
( a(bc/) - ) d (e (f g ^) ^)
( a(bc/) - ) (d (e (f g ^) ^) * )
(( a(bc/) - ) (d (e (f g ^) ^) * ) +)
```
Algorithm: Use a stack to evaluate a postfix expression

Let expression be a list of elements.

s = empty stack
cur = head of expression list
while (cur != null){
    if ( cur.element is a base expression )
        s.push( cur.element )
    else{
        // cur.element is an operator
        operand1 = s.pop()
        operand2 = s.pop()
        operator = cur.element // for clarity only
        s.push( evaluate( operand1, operator, operand2 ) )
    }
    cur = cur.next
}
Prefix expressions called “Polish Notation”
(after Polish logician Jan Lucasewicz 1920’s)

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

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

\[ 5 \times 4 + 3 \]

\[ 5 \text{ <enter> } \]
\[ 4 \text{ <enter> } \]
\[ * \text{ <enter> } \]
\[ 3 \text{ <enter> } \]
\[ + \text{ <enter> } \]

No “=” symbol needed on keyboard.