Trees

Jordi Cortadella and Jordi Petit
Department of Computer Science
Trees

Data are often organized hierarchically

source: https://en.wikipedia.org/wiki/Tree_structure
Filesystems

Trees
Company structure
Mind maps

- theoretical
- technical
- applied
- databases
- WWW
- algorithms
- data structures
- programming languages
- software engineering

Computer Science
Tree of Life

http://www.greennature.ca/
Probability trees
Parse trees
Image representation (quad-trees)
Decision trees

Tree: definition

- **Graph theory**: a tree is an undirected graph in which any two vertices are connected by exactly one path.

- **Recursive definition (CS)**. A non-empty tree $T$ consists of:
  - a root node $r$
  - a list of trees $T_1, T_2, ..., T_n$ that hierarchically depend on $r$. 

![Diagram of a tree with a root node $r$ and subtrees $T_1$, $T_2$, ..., $T_n$.]
• A is the root node.
• Nodes with no children are leaves (e.g., B and P).
• Nodes with the same parent are siblings (e.g., K, L and M).
• The depth of a node is the length of the path from the root to the node. Examples: depth(A)=0, depth(L)=2, depth(Q)=3.
Tree: representation with linked lists

```
struct TreeNode {
    Type element;
    list<TreeNode> children; // Linked list of children
};
```
Tree: representation with vectors

```cpp
struct TreeNode {
    Type element;
    vector<TreeNode> children; // Vector of children
};
```
Print a tree

```
struct Tree {
    string name;
    vector<Tree> children;
};

print(const Tree& T, int depth=0);
```
 /** Prints a tree indented according to depth.
 * Pre: The tree is not empty. */
 void print(const Tree& T, int depth) {

    // Print the root indented by 2*depth
    cout << string(2*depth, ' ') << T.name << endl;

    // Print the children with depth + 1
    for (const Tree& child: T.children)
        print(child, depth + 1);
}

This function executes a **preorder** traversal of the tree: each node is processed **before** the children.
Print a tree (postorder traversal)

**Postorder traversal:** each node is processed after the children.
/** Prints a tree (in postorder) indented according to depth. * Pre: The tree is not empty. */
void printPostOrder(const Tree& T, int depth) {

    // Print the children with depth + 1
    for (const Tree& child: T.children)
        printPostOrder(child, depth + 1);

    // Print the root indented by 2*depth
    cout << string(2*depth, ' ') << T.name << endl;
}

This function executes a postorder traversal of the tree: each node is processed after the children.
Binary trees

Nodes with at most two children.

```
struct BinTree {
    Type element;
    BinTree* left;
    BinTree* right;
};
```
Example: expression trees

Expression tree for: \( a + b \times c + (d \times e + f) \times g \)

Postfix representation: \( a \ b \ c \ + \ d \ e \ + \ f \ + \ g \ \times \times \)

How can the postfix representation be obtained?
Example: expression trees

```c++
struct ExprTree {
    char op; // operand or operator
    ExprTree* left;
    ExprTree* right;
};

using Expr = ExprTree*;

/** Builds an expression tree from a correct *
 * expression represented in postfix notation. */
Expr buildExpr(const string& expr);

/** Generates a string with the expression in *
 * infix notation. */
string infixExpr(const Expr T);

/** Evaluates an expression taking V as the value of the *
 * variables (e.g., V['a'] contains the value of a). */
int evalExpr(const Expr T, const map<char, int>& V);
```

Expressions are represented by strings in postfix notation in which the characters ‘a’…‘z’ represent operands and the characters ‘+’ and ‘*’ represent operators.
How to build an expression tree

\[ a \ast b \ast c + d \ast e + f + g \ast + \]

Trees

© Dept. CS, UPC
How to build an expression tree

```
ab c * + de * f + g * +
```

Stack

```
Stack
```

```
a
```
How to build an expression tree

\[ a \, b \, c \, * \, + \, d \, e \, * \, f \, + \, g \, * \, + \]

Stack

Trees
How to build an expression tree

a b c * + d e * f + g * +

Stack

a b c

© Dept. CS, UPC
How to build an expression tree

```
a b c * + d e * f + g * +
```

Trees

© Dept. CS, UPC
How to build an expression tree

\[ a \times c + d \times e + f + g \]

Stack

Trees
How to build an expression tree

```
a b c * + d e * f + g * +
```

Stack

```
+  
  / 
*/  
  /  
ab  
c  d
```
How to build an expression tree
How to build an expression tree

```
ab c * + de * f + g * +
```

Stack

```
+  
|  |
*  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

Trees © Dept. CS, UPC 31
How to build an expression tree

$ab (c + de) * f + g * *$

Stack

Trees
How to build an expression tree

```
  a b c * + d e * f + g *
```

Stack

```
  +
  / \
  *   +
 / \\ / \\ \
 a b c d e f
```
How to build an expression tree

```
ab c * + de * f + g **
```

Stack

Trees © Dept. CS, UPC
How to build an expression tree

Trees

© Dept. CS, UPC
How to build an expression tree

Trees © Dept. CS, UPC

Stack

a b c * + d e * f + g **
Example: expression trees

```cpp
Expr buildExpr(const string& expr) {
    stack<Expr> S;
    // Visit the chars of the string sequentially
    for (char c: expr) {
        if (c >= 'a' and c <= 'z') {
            // We have an operand in {'a'…'z'}. Create a leaf node.
            S.push(new ExprTree{c, nullptr, nullptr});
        } else {
            // c is an operator ('+' or '*')
            Expr right = S.top();
            S.pop();
            Expr left = S.top();
            S.pop();
            S.push(new ExprTree{c, left, right});
        }
    }
    // The stack has only one element and is freed after return
    return S.top();
}
```

Remember: `using Expr = ExprTree*;`
Example: expression trees

```cpp
/** Returns a string with an infix representation of T. */
string infixExpr(const Expr T) {

    // Let us first check the base case (an operand)
    if (T->left == nullptr) return string(1, T->op);

    // We have an operator. Return ( T->left T->op T->right )
    return "(" +
        infixExpr(T->left) +
        T->op +
        infixExpr(T->right) +
        ")";
}
```

**Inorder** traversal: node is visited *between* the left and right children.
/** Evaluates an expression taking V as the value of the variables (e.g., V[‘a’] contains the value of a). */
int evalExpr(const Expr T, const map<char, int>& V) {
    if (T->left == nullptr) return V.at(T->op);
    int l = evalExpr(T->left, V);
    int r = evalExpr(T->right, V);
    return T->op == ‘+’ ? l+r : l*r;
}

/** Example of usage of ExprTree. */
int main() {
    Expr T = buildExpr("abc++de*f+g++");
    cout << infixExpr(T) << endl;
    cout << "Eval = 
    " << evalExpr(T, {{‘a’,3}, {‘b’,1}, {‘c’,0}, {‘d’,5}, {‘e’,2}, {‘f’,1}, {‘g’,6}})
    << endl;
    freeExpr(T); // Not implemented yet
}
Exercises

• Design the function `freeExpr`.

• Modify `infixExpr` for a nicer printing:
  – Minimize number of parenthesis.
  – Add spaces around + (but not around *).

• Extend the functions to support other operands, including the unary – (e.g., –a/b).
**Traversals**: algorithm to visit the nodes of a tree in some specific order.

The actions performed when visiting each node can be a parameter of the traversal algorithm.

```cpp
struct TreeNode {
    Tinfo info;
    TreeNode* left;
    TreeNode* right;
};

using Tree = TreeNode*;

using visitor = void (int &);

// This function matches the type visitor
void print(int& i) {
    cout << i << endl;
}

void traversal(Tree T, visitor v);
```
Tree traversals

Preorder: A B D G H E I C F J K

```c
void preorder(Tree T, visitor v) {
    if (T != nullptr) {
        v(T->elem);
        preorder(T->left, v);
        preorder(T->right, v);
    }
}
```
Tree traversals

Preorder:    A B D G H E I C F J K
Postorder:   G H D I E B J K F C A

void preorder(Tree T, visitor v) {
    if (T != nullptr) {
        v(T->elem);
        preorder(T->left, v);
        preorder(T->right, v);
    }
}

void postorder(Tree T, visitor v) {
    if (T != nullptr) {
        postorder(T->left, v);
        postorder(T->right, v);
        v(T->elem);
    }
}
Tree traversals

Preorder: A B D G H E I C F J K
Postorder: G H D I E B J K F C A
Inorder: G D H B E I A J F K C

```cpp
void inorder(Tree T, visitor v) {
    if (T != nullptr) {
        inorder(T->left, v);
        v(T->elem);
        inorder(T->right, v);
    }
}
```
Tree traversals

Preorder: A B D G H E I C F J K
Postorder: G H D I E B J K F C A
Inorder: G D H B E I A J F K C
By levels: A B C D E F G H I J K

```cpp
void byLevels(Tree T, visitor v) {
    queue<Tree> Q; Q.push(T);
    while (not Q.empty()) {
        T = Q.front(); Q.pop();
        if (T != nullptr) {
            v(T->elem);
            Q.push(T->left); Q.push(T->right);
        }
    }
}
```
EXERCISES
Traversals: Full Binary Trees

• A Full Binary Tree is a binary tree where each node has 0 or 2 children.

• Draw the full binary trees corresponding to the following tree traversals:
  – Preorder: 2 7 3 6 1 4 5; Postorder: 3 6 7 4 5 1 2
  – Preorder: 3 1 7 4 9 5 2 6 8; Postorder: 1 9 5 4 6 8 2 7 3

• Given the pre- and post-order traversals of a binary tree (not necessarily full), can we uniquely determine the tree?
  – If yes, prove it.
  – If not, show a counterexample.
Traversals: Binary Trees

• Draw the binary trees corresponding the following traversals:
  – Preorder: 3 6 1 8 5 2 4 7 9; Inorder: 1 6 3 5 2 8 7 4 9
  – Level-order: 4 8 3 1 2 7 5 6 9; Inorder: 1 8 5 2 4 6 7 9 3
  – Postorder: 4 3 2 5 9 6 8 7 1; Inorder: 4 3 9 2 5 1 7 8 6

• Describe an algorithm that builds a binary tree from the preorder and inorder traversals.
We want to draw the skeleton of a binary tree as it is shown in the figure. For that, we need to assign \((x, y)\) coordinates to each tree node. The layout must fit in a pre-defined bounding box of size \(W \times H\), with the origin located in the top-left corner.

Design the function

\[
\text{void draw(Tree T, double W, double H)}
\]

to assign values to the attributes \(x\) and \(y\) of all nodes of the tree in such a way that the lines that connect the nodes do not cross.

Suggestion: calculate the coordinates in two steps. First assign \((x, y)\) coordinates using some arbitrary unit. Next, shiftSCALE the coordinates to exactly fit in the bounding box.