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
Genealogical 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 $T_1, T_2, \ldots, T_n$ that hierarchically depend on $r$. 

![Diagram of a tree with a root node $r$ and subtrees $T_1, T_2, \ldots, T_n$.](image)
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

```cpp
struct TreeNode {
    Type element;
    list<TreeNode*> children; // Linked list of children
};
```
struct TreeNode {
  Type element;
  vector<TreeNode*> children;
};
struct Tree {
    string name;
    vector<Tree*> children;
};

print(Tree* T, int depth=0);
/** Prints a tree indented according to depth. */
print(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 (Tree* child: T->children) print(child, depth + 1);
}
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. */
printPostOrder(Tree* T, int depth) {
    // Print the children with depth + 1
    for (Tree* child: T->children) print(child, depth + 1);
    // Print the root indented by 2*depth
    cout << string(2*depth, ' ') << T->name << endl;
}

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

Nodes with at most two children.

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

Expression tree for: \( a + b\cdot c + (d\cdot e + f) \cdot g \)
Postfix representation: \( a\ b\ c\ +\ +\ d\ e\ f\ +\ g\ *\ *\ + \)

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

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

Expressions are represented by strings in postfix notation in which the characters ‘a’…‘z’ represent operands and the characters ‘+’ and ‘*’ represent operators.

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

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

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

```cpp
ExprTree* buildExpr(const string& expr) {
    stack<ExprTree*> 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 '*')
            ExprTree* right = S.top();
            S.pop();
            ExprTree* 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();
}
```
/** Returns a string with an infix representation of T. */
string infixExpr(const ExprTree* 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 ExprTree* T, const map<char,int>& V) {
    if (T->left == nullptr) return V[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() {
    ExprTree* 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).
Tree traversals

**Traversal**: 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
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

```c
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

```c
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

```c
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);
        }
    }
}
```