TREES

Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu
TREE DEFINITION

• TREE ≡ a branching structure between nodes

• A finite set \( T \) of one or more nodes such that:
  
  1. one element of the set is distinguished, \( \text{ROOT}(T) \)
  
  2. the remaining nodes of \( T \) are partitioned into \( m \geq 0 \) disjoint sets \( T_1, T_2, \ldots, T_m \) and each of these sets is in turn a tree.
     
     • trees \( T_1, T_2, \ldots, T_m \) are the *subtrees* of the root

• Recursive definition – easy to prove theorems about properties of trees.

Ex: prove true for 1 node
    assume true for \( n \) nodes
    prove true for \( n+1 \) nodes

• ORDERED TREE ≡ if the relative order of the subtrees \( T_1, T_2, \ldots, T_m \) is important

• ORIENTED TREE ≡ order is not important

\[
\begin{align*}
\text{ordered} & \quad \neq \\
geq & \\
\end{align*}
\]

• Computer representation \( \Rightarrow \) ordered!
TERMINOLOGY

• **Counterintuitive!**

- **DEGREE** ≡ number of subtrees of a node
- Terminal node ≡ *leaf* ≡ degree 0
- **BRANCH NODE** ≡ non-terminal node

- Root is the *father* of the roots of its subtrees
- Roots of subtrees of a node are *brothers*
- Roots of subtrees of a node are *sons* of the node
- The root of the tree has no father!
- A is an *ancestor* of C, E, G, ...
- G is a *descendant* of A

\[
\text{level}(X) \equiv \begin{cases} 
0 & \text{if } \text{father}(X) = \Omega \\
1 + \text{level}(\text{father}(X)) & \text{otherwise}
\end{cases}
\]

Ex: \( \text{level}(G) = 1 + \text{level}(F) \)

\[
\begin{align*}
1 + \text{level}(C) \\
1 + \text{level}(A) \\
0
\end{align*}
\]
FORESTS AND BINARY TREES

• **FOREST** ≡ a set (usually ordered) of 0 or more disjoint trees, 
or equivalently: 
  the nodes of a tree excluding the root

• BINARY TREE ≡ a finite set of nodes which either is empty *or* 
a root and two disjoint binary trees called the 
*left* and *right* subtrees of the root

• Is a binary tree a special case of a tree?

  NO! An entirely different concept

  1  and  2  are different binary trees

  1 has an empty right subtree
  2 has an empty left subtree
  But as ‘trees’ 1 and 2 are identical!
OTHER REPRESENTATIONS OF TREES

• Nested sets (also known as ‘bubble diagrams’)

  \[ \text{Tree} \quad (\text{root subtree}_1 \, \text{subtree}_2 \, \ldots \, \text{subtree}_n) \]
  \[ (A \ (B \ (C) \ (D)) \ (G \ (E \ (F))) \ ) \]  

• Nested parentheses

  \[ \text{Tree} \quad (\text{root left right}) \]
  \[ (A \ (B \ (C) \ (D) \ ) \ (D \ (C) \ (B) ) \ ) \]
  \[ (G \ (E \ (F) \ ) \ (F) \ ) \ (E) \ ) \]  

• Indentation

  \[ \begin{array}{cccc}
  & A & \\
  & B & \\
  C & \\
  D & \\
  G & \\
  E & \\
  F & \\
  \end{array} \]

• Dewey decimal notation: \[ 2.1 \, 2.2.2 \, 2.3.4.5 \]
APPLICATIONS

- Segmentation of large rectangular arrays – \( A_{n,m} \)


  *each row is a segment (Burroughs computers)*

- Algebraic formulas

  ![Diagram of a tree structure with + at the root, A, \( \times \), D, B, and C as children.]

  \[ A + ((B \div C) \times D) \]

  1. no need for parentheses
  - but \( A - B + C = (A - B) + C \)
  \[ A - (B + C) \]

  2. code generation

    LW 1, A
    LW 2, B
    DW 2, C
    MW 2, D
    AW 2, 1

Copyright © 1998 by Hanan Samet
LISTs (with a capital L!)

• **LIST** ≡ a finite sequence of 0 or more atoms or **LISTS**

\[
L = (A, (B, A, B), (()), C, (( (2) )))
\]

( ) ≡ **empty list**

• **Index notation:**

\[
L[2] = (B, A, B) \\
L[2, 1] = B \\
L[5, 2] \\
L[5, 1, 1]
\]

• **Differences between **LISTS** and trees:**

1. no data appears in the nodes representing **LISTS** - i.e., *

2. **LISTS** may be recursive

\[
M = (M) \\
[M] \leftarrow \text{Label}
\]

3. **LISTS** may overlap (i.e., need not be disjoint)

• equivalently, subtrees may be shared

\[
N = (M, M, C, N)
\]
TRAVERSING BINARY TREES

• Representation

\[
\begin{array}{c}
\text{LLINK} \\
\text{INFO} \\
\text{RLINK}
\end{array}
\]

\[
\begin{array}{cc}
\Omega & \Omega \\
\Omega & \Omega
\end{array}
\]

• Applications:
  1. code generation in compilers
  2. game trees in artificial intelligence
  3. detect if a structure is really a tree
     • TREE ≡ one path from each node to another node
       (unlike graph)
     • no cycles
TRAVERSAL ORDERS

1. Preorder ≡ root, left subtree, right subtree
   · depth-first search
2. Inorder ≡ left subtree, root, right subtree
   · binary search tree
3. Postorder ≡ left subtree, right subtree, root
   · code generation

   • Binary search tree: left < root < right

   ![Binary search tree diagram]

   inorder yields 10 15 20 30 45

   • Ex:
     preorder = A B D I K C E G F H J
     inorder = I D K B A E G C H F J
     postorder = I K D B G E H J F C A

   • Inorder traversal requires a stack to go back up the tree:

     D
     B
     A

Copyright © 1998 by Hanan Samet
INORDER TRAVERSAL ALGORITHM

procedure inorder(tree pointer T);
begin
    stack A;
    tree pointer P;
    A←Ω;
    P←T;
    while not(P=Ω and A=Ω) do
        begin
            if P=Ω then
                begin
                    P←A;       /* Pop the stack */
                    visit(ROOT(P));
                    P←RLINK(P);
                end
            else
                begin
                    A←P;       /* Push on the stack */
                    P←LLINK(P);
                end;
        end;
end;

Using recursion:

procedure inorder(tree pointer T);
begin
    if T=Ω then return
    else
        begin
            inorder(LLINK(T));
            visit(ROOT(T));
            inorder(RLINK(T));
        end;
end;
THREADED BINARY TREES

- Binary tree representation has too many $\Omega$ links
- Use 1-bit tag fields to indicate presence of a link
- If $\Omega$ link, then use field to store links to other parts of the structure to aid the traversal of the tree

Unthreaded:  

definitions:

- $LLINK(p) = \Omega$
- $LLINK(p) = q \neq \Omega$
- $RLINK(p) = \Omega$
- $RLINK(p) = q \neq \Omega$

Threaded:

- $LLINK(p) = \Omega$, $LTAG(p) = 0$, $LLINK(p) = p$ (inorder predecessor of $p$)
- $LLINK(p) = q$, $LTAG(p) = 1$
- $RLINK(p) = \Omega$, $RTAG(p) = 0$, $RLINK(p) = p$ (inorder successor of $p$)
- $RLINK(p) = q$, $RTAG(p) = 1$

<table>
<thead>
<tr>
<th>LLINK</th>
<th>LTAG</th>
<th>INFO</th>
<th>RTAG</th>
<th>RLINK</th>
</tr>
</thead>
</table>

Ex:  

- If address of $ROOT(T)$ < address of left and right sons, then don’t need the TAG fields
- Threads will point to lower addresses!
OPERATIONS ON THREADED BINARY TREES

• Find the inorder successor of node P (P$)

1. $Q \leftarrow \text{RLINK}(P)$; /* right thread points to P$ */
2. if $\text{RTAG}(P)=1$ then
   begin /* not a thread */
     while $\text{LTAG}(Q)=1$ do $Q \leftarrow \text{LLINK}(Q)$;
   end;

• Insert node Q as the right subtree of node P

1. $\text{RLINK}(Q) \leftarrow \text{RLINK}(P)$; $\text{RTAG}(Q) \leftarrow \text{RTAG}(P)$;
   $\text{RLINK}(P) \leftarrow Q$; $\text{RTAG}(P) \leftarrow 1$;
   $\text{LLINK}(Q) \leftarrow P$; $\text{LTAG}(Q) \leftarrow 0$;
2. if $\text{RTAG}(Q)=1$ then $\text{LLINK}(Q$)\$) \leftarrow Q$;
SUMMARY OF THREADING

1. Advantages
   • no need for a stack for traversal
   • will not run out of memory during inorder traversal
   • can find inorder successor of any node without having to traverse the entire tree

2. Disadvantages
   • insertion and deletion of nodes is slower
   • can’t share common subtrees in the threaded representation

Ex: two choices for the inorder successor of F

3. Right-threaded trees
   • inorder algorithms make little use of left threads
   • ‘\text{LTAG}(P)=1’ test can be replaced by ‘\text{LLINK}(P)=\Omega’ test
PRINCIPLES OF RECURSION

- Two binary trees T1 and T2 are said to be similar if they have the same shape or structure.
- Formally:
  1. they are both empty or
  2. they are both non-empty and their left and right subtrees respectively are similar

\[
\text{similar}(T_1,T_2) =
\begin{cases}
  \text{if empty}(T_1) \text{ and empty}(T_2) & \text{then } T \\
  \text{else if empty}(T_1) \text{ or empty}(T_2) & \text{then } F \\
  \text{else } & \text{similar(left}(T_1),\text{left}(T_2)) \text{ and} \\
  & \text{similar(right}(T_1),\text{right}(T_2)) \\
\end{cases}
\]

- Will similar work?
  - No! base case does not handle case when one of the trees is empty and the other one is not

- Simplifying:
  \[
  \begin{align*}
  A \text{ and } B &= \text{ if } A \text{ then } B \quad A \text{ or } B = \text{ if } A \text{ then } T \\
  &\quad \text{ else } F \\
  \end{align*}
  \]

\[
\begin{align*}
\text{similar}(T_1,T_2) &=
\begin{cases}
  \text{if empty}(T_1) & \text{then } \text{empty}(T_2) \\
  & \quad \begin{cases}
    \text{if empty}(T_2) & \text{then } T \\
    \text{else } & F \\
  \end{cases} \\
  \text{else if empty}(T_2) & \text{then } F \\
  \text{else if } & \text{similar(left}(T_1),\text{left}(T_2)) \text{ then} \\
  & \text{similar(right}(T_1),\text{right}(T_2)) \\
  \text{else } & F \\
\end{cases}
\end{align*}
\]
EQUIVALENCE OF BINARY TREES

- Two binary trees $T_1$ and $T_2$ are said to be *equivalent* if they are similar *and* corresponding nodes contain the same information.

```plaintext
equivalent(T1, T2) =
    if empty(T1) and empty(T2) then T
    else if empty(T1) or empty(T2) then F
    else root(T1) = root(T2) and
        equivalent(left(T1), left(T2)) and
        equivalent(right(T1), right(T2));
```

NO! we are dealing with binary trees and the left subtree of $C$ is not the same in the two cases.
RECURSION SUMMARY

• Avoids having to use an explicit stack in the algorithm
• Problem formulation is analogous to induction
• Base case, inductive case

• Ex: Factorial
  \[ n! = n \cdot (n - 1)! \]

  \[
  \text{fact}(n) = \begin{cases} 
  1 & \text{if } n=0 \\
  n \cdot \text{fact}(n-1) & \text{else}
  \end{cases};
  \]

  The result is obtained by peeling one’s way back along the stack

  \[
  \text{fact}(3) = 3 \cdot \text{fact}(2) \\
  = 3 \cdot (2 \cdot \text{fact}(1)) \\
  = 3 \cdot (2 \cdot (1 \cdot \text{fact}(0))) \\
  = 3 \cdot (2 \cdot (1 \cdot 1)) \\
  = 3 \cdot (2 \cdot 1) \\
  = 3 \cdot 2 \\
  = 6
  \]

  Using an accumulator variable and a call \text{fact2}(n, 1):

  \[
  \text{fact2}(n, \text{total}) = \begin{cases} 
  \text{total} & \text{if } n=0 \\
  \text{fact2}(n-1, n \cdot \text{total}) & \text{else}
  \end{cases};
  \]

  Solution is iterative

• Recursion implemented on computer using stack instructions.
• Dec-system 10: \text{PUSH, POP, PUSHJ, POPJ}
• Stack pointer format: \text{(count, address)}
• Can simulate stack if no stack instructions
COMPLETE BINARY TREES

When a binary tree is reasonably complete (most $\Omega$ links are at the highest level), use a sequential storage allocation scheme so that links become unnecessary.

- If $n$ is the highest level at which a node is found, then at most $2^{n+1} - 1$ words are needed.

- Storage allocation method:
  1. root has address 1
  2. left son of $x$ has address $2 \times \text{address}(x)$
  3. right son of $x$ has address $2 \times \text{address}(x) + 1$

- When should a complete binary tree be used?
  
  $n =$ highest level of the tree at which a node is found
  $x =$ # of nodes in tree
  3 words per node (left link, right link, info)
  use a complete binary tree when $x > \frac{2^{n+1} - 1}{3}$
FORESTS

• A forest is an ordered set of 0 or more trees
• There exists a natural correspondence between forests and binary trees

- Rigorous definition of $B(F)$

  $F = (T_1, T_2, \ldots, T_n)$
  $T_{i,1}, T_{i,2}, \ldots, T_{i,m}$ are subtrees of $T_i$
  1. If $n = 0$, $B(F)$ is empty
  2. If $n > 0$, root of $B(F)$ is root($T_1$)
     left subtree of $B(F)$ is $B(T_{1,1}, T_{1,2}, \ldots, T_{1,m})$
     right subtree of $B(F)$ is $B(T_2, T_3, \ldots, T_n)$

• Traversal of forests

  preorder:  
  1. visit root of first tree  
  2. traverse subtrees of first tree in preorder  
  3. traverse remaining subtrees in preorder

  postorder:  
  1. traverse subtrees of first tree in postorder  
  2. visit root of first tree  
  3. traverse remaining subtrees in postorder

  preorder = A B C K D E H F J G  
  postorder = B K C A H E J F G D  
  = inorder of binary tree
EQUIVALENCE RELATION

• Given: relations as to what is equivalent to what \((a \equiv b)\)
• Goal: is \(x \equiv y\)?

• Formal definition of an *equivalence relation*
  1. if \(x \equiv y\) and \(y \equiv z\) then \(x \equiv z\) (transitivity)
  2. if \(x \equiv y\) then \(y \equiv x\) (symmetry)
  3. \(x \equiv x\) (reflexivity)

• Ex: \(S = \{1 .. 9\}\)
  1 \(\equiv 5\)  6 \(\equiv 8\)  7 \(\equiv 2\)  9 \(\equiv 8\)  3 \(\equiv 7\)  4 \(\equiv 2\)  9 \(\equiv 3\)
  is \(2 \equiv 6\)?
  Yes, since \(2 \equiv 7 \equiv 3 \equiv 9 \equiv 8 \equiv 6\)

• Partitions \(S\) into disjoint subsets or *equivalence classes*
• Two elements equivalent iff they belong to same class
• What are the equivalence classes in this example?

\{1,5\} and \{2,3,4,6,7,8,9\}
ALGORITHM

- Represent each element as a node in forest of trees
- Trees consist only of father links (nil at roots)
- Each (nonredundant) relation merges two trees into one
- Basic strategy:

  ```plaintext
  for each relation a=b do
  begin
    find root node r of tree containing a; /* Find step */
    find root node s of tree containing b;
    if they differ, merge the two trees; /* Union step */
  end;
  ```

  ```plaintext
  for every element i do father(i) ← Ω
  while input_not_exhausted do
  begin
    get_pair(a,b);
    while father(a) ≠ Ω do a ← father(a);
    while father(b) ≠ Ω do b ← father(b);
    if (a ≠ b) then father(a) ← b;
  end;
  ```

- Algorithm (also known as union-find):

  ```plaintext
  father(k): 5 2 2 8 2 2 8
  k: 1 2 3 4 5 6 7 8 9
  ```

- More efficient with path compression and weight balancing
- Execution time “almost linear” (inverse of Ackermann function)