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

  ![Diagram](image)

  • Computer representation ⇒ ordered!

Copyright © 1998 by Hanan Samet
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: \( \text{level}(G) = 1 + \text{level}(F) = 1 + 1 + \text{level}(A) = 1 + 1 + 0 = 2 \)
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

\[
\text{level}(X) \equiv \text{if } \text{father}(X) = \Omega \text{ then } 0 \\
\quad \text{else } 1 + \text{level}(\text{father}(X));
\]

Ex: \( \text{level}(G) = 1 + \text{level}(F) \)
\[
1 + \text{level}(C) \\
1 + \text{level}(A) \\
0
\]
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

```
    A
   / \          has the forest
  B   C
 / \       B
D   E     C
    / \    / \
   F   D  E  F
```

• 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?
FORESTS AND BINARY TREES

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

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

has the forest

\[
\begin{array}{c}
B \\
D \\
C \\
E & F
\end{array}
\]

• BINARY TREE $\equiv$ 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

\[
\begin{array}{c}
A \\
B \\
A & B
\end{array}
\]

1 and 2 are different binary trees
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

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

• Indentation

\[ \text{Indentation} \]\n\[ A \]
\[ \quad B \]
\[ \quad \quad C \]
\[ \quad \quad D \]
\[ \quad G \]
\[ \quad \quad E \]
\[ \quad \quad \quad F \]

• Dewey decimal notation: 2.1 2.2.2 2.3.4.5

Copyright © 1998 by Hanan Samet
APPLICATIONS

• Segmentation of large rectangular arrays – $A[n,m]$

  each row is a segment (Burroughs computers)

  $A[1,1] \ A[1,2] \ldots \ A[1,m] \ A[n,1] \ A[n,2] \ldots \ A[n,m]$

• Algebraic formulas

  A operator operand operand

  $A + \left( \frac{B}{C} \right) \times D$

  1. no need for parentheses
    • but $A-B+C = (A-B)+C$
      $\neq A-(B+C)$

  2. code generation

    \begin{align*}
    LW & \quad 1, A \\
    LW & \quad 2, B \\
    DW & \quad 2, C \\
    MW & \quad 2, D \\
    AW & \quad 2, 1 \\
    \end{align*}
LISTs (with a capital L!)

- LIST ≡ a finite sequence of 0 or more atoms or LISTS

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

\( () \equiv \text{empty list} \)

- 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

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

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

- equivalently, subtrees may be shared

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

• Representation

• 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

Copyright © 1998 by Hanan Samet
TRAVERSAL ORDERS

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

• Binary search tree: \quad left < root < right

\[
\begin{array}{c}
30 \\
15 \quad 45 \\
10 \quad 20
\end{array}
\]

\text{inorder yields} \quad 10 \ 15 \ 20 \ 30 \ 45

• Ex:

\[
\begin{array}{c}
A \\
B \quad C \\
D \quad E \quad F \\
I \quad K \quad G \quad H \quad J
\end{array}
\]

\text{preorder} = \quad \text{inorder} = \quad \text{postorder} =

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

D
B
A
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

```
          30
         / \   
        15   45
       /   /   
      10  20  
```

inorder yields  10 15 20 30 45

• Ex:

```
              A
             / \  
            B   C
           /   /  
          D   E   F
         / \   /   
        I   G H   J
```

preorder =  A B D I K C E G F H J

inorder =

postorder =

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

D

B

A
TRAVERSAL ORDERS

1. Preorder \equiv \text{root, left subtree, right subtree} \\
   \quad \text{- depth-first search}

2. Inorder \equiv \text{left subtree, root, right subtree} \\
   \quad \text{- binary search tree}

3. Postorder \equiv \text{left subtree, right subtree, root} \\
   \quad \text{- code generation}

- Binary search tree: \quad \text{left < root < right}

```
                      30
                     / \  \
                   15   45
                  /   / \
                10  20
```

- Ex:

```
             A
            / \  \
           B   C
          /   / \ \\
         D   E   F
        /  \    / \\
       I   K   G  H  J
```

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

```
  D
  B
  A
```
TRAVERSAL ORDERS

1. Preorder \(\equiv\) root, left subtree, right subtree
   • depth-first search
2. Inorder \(\equiv\) left subtree, root, right subtree
   • binary search tree
3. Postorder \(\equiv\) left subtree, right subtree, root
   • code generation
   • Binary search tree: left < root < right

\begin{center}
\begin{tabular}{c}
\text{inorder yields} \\
10 & 15 & 20 & 30 & 45
\end{tabular}
\end{center}

• Ex:

\begin{center}
\begin{tabular}{c}
\text{preorder} = A B D I K C E G F H J \\
\text{inorder} = I D K B A E G C H F J \\
\text{postorder} = I K D B G E H J F C A
\end{tabular}
\end{center}

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

D
B
A
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 Ω links
- Use 1-bit tag fields to indicate presence of a link
- If Ω link, then use field to store links to other parts of the structure to aid the traversal of the tree

Unthreaded:              Threaded:
LLINK(p) = Ω            LTAG(p) = 0,
                   LLINK(p) = $p$ = inorder predecessor of p
LLINK(p) = q ≠ Ω       LTAG(p) = 1,
                   LLINK(p) = q
RLINK(p) = Ω           RTAG(p) = 0,
                   RLINK(p) = p$ = inorder successor of p
RLINK(p) = q ≠ Ω       RTAG(p) = 1,
                   RLINK(p) = q

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

Ex: HEAD

- If address of ROOT(T) < address of left and right sons, then don’t need the TAG fields
- Threads will point to lower addresses!
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:  Threaded:
\[
\begin{align*}
\text{LLINK}(p) &= \Omega & \text{LLINK}(p) &= \Omega \\
\text{RTAG}(p) &= 0, & \text{LLINK}(p) &= \text{inorder predecessor of } p \\
\text{RTAG}(p) &= 1, & \text{LLINK}(p) &= q \\
\text{RLINK}(p) &= \Omega & \text{RTAG}(p) &= 0, \\
\text{RLINK}(p) &= q & \text{RLINK}(p) &= \text{inorder successor of } p \\
\text{RLINK}(p) &= q & \text{RLINK}(p) &= q
\end{align*}
\]

Ex:  HEAD

- If address of \( \text{ROOT}(T) < \) address of left and right sons, then don't need the TAG fields
- Threads will point to lower addresses!

Copyright © 1998 by Hanan Samet
THREADED BINARY TREES

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

Unthreaded: Threaded:

\[
\begin{align*}
\text{LLINK}(p) &= \Omega & \text{LTAG}(p) &= 0, \\
\text{LLINK}(p) &= \Omega & \text{LLINK}(p) &= p = \text{inorder predecessor of } p \\
\text{LLINK}(p) &= q \neq \Omega & \text{LTAG}(p) &= 1, & \text{LLINK}(p) &= q \\
\text{RLINK}(p) &= \Omega & \text{RTAG}(p) &= 0, & \text{RLINK}(p) &= p = \text{inorder successor of } p \\
\text{RLINK}(p) &= q \neq \Omega & \text{RTAG}(p) &= 1, & \text{RLINK}(p) &= q
\end{align*}
\]

Ex: HEAD

- If address of \text{ROOT}(T) < address of left and right sons, then don't need the \text{TAG} fields
- Threads will point to lower addresses!

Copyright © 1998 by Hanan Samet
OPERATIONS ON THREADED BINARY TREES

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

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

• Insert node Q as the right subtree of node P

1. RLINK(Q)←RLINK(P); RTAG(Q)←RTAG(P);
   RLINK(P)←Q; RTAG(P)←1;
   LLINK(Q)←P; LTAG(Q)←0;
2. if RTAG(Q)=1 then LLINK(Q$)←Q;
OPERATIONS ON THREADED BINARY TREES

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

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

• Insert node Q as the right subtree of node P

1. RLINK(Q) ← RLINK(P); RTAG(Q) ← RTAG(P);
   RLINK(P) ← Q; RTAG(P) ← 1;
   LLINK(Q) ← P; LTAG(Q) ← 0;
2. if RTAG(Q) = 1 then LLINK(Q$) ← Q;
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 \( \text{LLINK}(Q) \leftarrow Q \);
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); \quad \text{RTAG}(Q)\leftarrow\text{RTAG}(P);
   \text{RLINK}(P)\leftarrow Q; \quad \text{RTAG}(P)\leftarrow 1;
   \text{LLINK}(Q)\leftarrow P; \quad \text{LTAG}(Q)\leftarrow 0;
2. if \text{RTAG}(Q)=1 then \text{LLINK}(Q)\leftarrow Q;
OPERATIONS ON THREADABLE BINARY TREES

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

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

• Insert node Q as the right subtree of node P

1. \( RLINK(Q) \leftarrow RLINK(P) \);
2. if RTAG(Q)=1 then LLINK(Q$) \leftarrow Q;
OPERATIONS ON THREADED BINARY TREES

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

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

• Insert node Q as the right subtree of node P

1. RLINK(Q) ← RLINK(P); RTAG(Q) ← RTAG(P);
   RLINK(P) ← Q; RTAG(P) ← 1;
   LLINK(Q) ← P; LTAG(Q) ← 0;
2. if RTAG(Q) = 1 then LLINK(Q$) ← 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
   • ‘LTAG(P)=1’ test can be replaced by ‘LLINK(P)=Ω’ test
PRINCIPLES OF RECURSION

• Two binary trees $T_1$ and $T_2$ 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.

$\text{similar}(T_1, T_2) =$
  if $\text{empty}(T_1)$ and $\text{empty}(T_2)$ then $T$
  else $\text{similar}(\text{left}(T_1), \text{left}(T_2))$ and $\text{similar}(\text{right}(T_1), \text{right}(T_2))$;

• Will similar work?
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

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

• Will similar work?

• No! base case does not handle case when one of the trees is empty and the other one is not
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

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

• Will similar work?

• No! base case does not handle case when one of the trees is empty and the other one is not

• Simplifying:

\[
A \text{ and } B = \text{ if } A \text{ then } B \\
A \text{ or } B = \text{ else } F
\]
PRINCIPLES OF RECURSION

• Two binary trees $T_1$ and $T_2$ are said to be \textit{similar} if they have the same shape or structure
• Formally:
  1. they are both empty \textit{or}
  2. they are both non-empty and their left and right subtrees respectively are similar

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

• Will similar work?
• No! base case does not handle case when one of the trees is empty and the other one is not
• Simplifying:
  \[
  A \text{ and } B = \begin{cases} 
  A \text{ then } B & \text{if } A \\
  F & \text{else}
\end{cases} \quad A \text{ or } B = \begin{cases} 
  A \text{ then } T & \text{if } A \\
  B & \text{else}
\end{cases}
\]

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

• Two binary trees \( T_1 \) and \( T_2 \) 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

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

• Will \( \text{similar} \) work?

• No! base case does not handle case when one of the trees is empty and the other one is not

• Simplifying:

\[
\begin{align*}
A \land B &= \ \text{if } A \text{ then } B \ \text{ else } F \\
A \lor B &= \ \text{if } A \text{ then } T \ \text{ else } B
\end{align*}
\]

\[
similar(T_1, T_2) =
\begin{cases}
    \text{true} & \text{if } \text{empty}(T_1) \ \text{ then } \text{empty}(T_2) \\
    \text{false} & \text{else if } \text{empty}(T_2) \\
    \text{true} & \text{else if } \text{similar}(\text{left}(T_1), \text{left}(T_2)) \ \text{and} \\
    \text{similar}(\text{right}(T_1), \text{right}(T_2)) \\
    \text{false} & \text{else}
\end{cases}
\]
EQUIVALENCE OF BINARY TREES

• Two binary trees T1 and T2 are said to be equivalent if they are similar and corresponding nodes contain the same information.

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

Copyright © 1998 by Hanan Samet
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.

```
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
  \[ n! = n \cdot (n - 1)! \]

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

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

  \[
  \text{fact}(3) = 3 \text{\* fact}(2) \\
  \quad 2 \text{\* fact}(1) \\
  \quad \quad 1 \text{\* fact}(0) \\
  \quad \quad \quad 1 
  \]

  \[= 6\]

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

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

  Solution is iterative

• Recursion implemented on computer using stack instructions.
• Dec-system 10: \text{PUSH, POP, PUSHJ, POPJ}
• Stack pointer format: (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 \( x > \frac{2^{n+1} - 1}{3} \)
• A *forest* is an ordered set of 0 or more trees
• There exists a *natural correspondence* between forests and binary trees

![Diagram of forest 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 \( T_i \)
- If \( n = 0 \), \( B(F) \) is empty
- 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 \( B(T_2, T_3, \ldots, T_n) \)

**Traversals 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

Copyright © 1998 by Hanan Samet
• 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)
  \[ F = (T_1, T_2, \ldots, T_n) \]
  \[ T_{i,1}, T_{i,2}, \ldots, T_{i,m} \text{ are subtrees of } T_i \]
  1. If \( n = 0 \), \( B(F) \) is empty
  2. If \( n > 0 \), root of \( B(F) \) is \( \text{root}(T_1) \)
     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

\[ \text{preorder} = \ A \ B \ C \ K \ D \ E \ H \ F \ J \ G \]
**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)**

\[ \text{F} = (T_1, T_2, \ldots, T_n) \]

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

---

**Traversals 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** = \[ B \ K \ C \ A \ H \ E \ J \ F \ G \ D \]
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

$\text{preorder} = A\ B\ C\ K\ D\ E\ H\ F\ J\ G$
$\text{postorder} = B\ K\ C\ A\ H\ E\ J\ F\ G\ D$
$\equiv \text{inorder of binary tree}$
EQUIVALENCE RELATION

- Given: relations as to what is equivalent to what (a ≡ b)
- Goal: is x ≡ y?

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

- Ex: S = {1 .. 9}
  1 ≡ 5  6 ≡ 8  7 ≡ 2  9 ≡ 8  3 ≡ 7  4 ≡ 2  9 ≡ 3
  is 2 ≡ 6?
EQUIVALENCE RELATION

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

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

• Ex: S = {1 .. 9}
  1≡5  6≡8  7≡2  9≡8  3≡7  4≡2  9≡3
  is 2 ≡ 6 ?
  Yes, since 2≡7≡3≡9≡8≡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?
EQUIVALENCE RELATION

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

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

• Ex: S = {1 .. 9}
  1≡5  6≡8  7≡2  9≡8  3≡7  4≡2  9≡3
  is 2 ≡ 6 ?
  Yes, since 2≡7≡3≡9≡8≡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;
```

• Algorithm (also known as union-find):

```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;
```

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

```
5  2
1  8  7  3  4
9  6
```
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:

for each relation \( a \equiv b \) 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;

merge(a,b)

Algorithm (also known as union-find):

for every element \( i \) do father(\( i \)) \( \leftarrow \Omega \)
while input_not_exhausted do
  begin
    get_pair(a,b);
    while father(a) \( \neq \Omega \) do a \( \leftarrow \) father(a);
    while father(b) \( \neq \Omega \) do b \( \leftarrow \) father(b);
    if (a \( \neq \) b) then father(a) \( \leftarrow \) b;
  end;

father(k):

father(k): 5
k: 1 2 3 4 5 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;
```

- Algorithm (also known as union-find):

```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;
```

father(k):

5 8

k: 1 2 3 4 5 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;
```

```
merge(a,b)
```

• Algorithm (also known as union-find):

```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;
```

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

Copyright © 1998 by Hanan Samet
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:

  for each relation $a \equiv b$ 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;

- Algorithm (also known as union-find):

  for every element $i$ do $\text{father}(i) \leftarrow \Omega$
  while input_not_exhausted do
  begin
  get_pair($a$, $b$);
  while $\text{father}(a) \neq \Omega$ do $a \leftarrow \text{father}(a)$;
  while $\text{father}(b) \neq \Omega$ do $b \leftarrow \text{father}(b)$;
  if ($a \neq b$) then $\text{father}(a) \leftarrow b$;
  end;

  $1 \equiv 5$
  $6 \equiv 8$
  $7 \equiv 2$
  $9 \equiv 8$

  $\text{father}(k): 5 \quad 8 \quad 2 \quad 8$
  $k: 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9$

Copyright © 1998 by Hanan Samet
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:

\[
\text{for each relation } a\equiv b \text{ do begin}
\begin{align*}
\text{find root node } r \text{ of tree containing } a; & \quad /* \text{Find step} */ \\
\text{find root node } s \text{ of tree containing } b; & \\
\text{if they differ, merge the two trees; } & \quad /* \text{Union step} */ \\
\end{align*}
\text{end;}
\]

- Algorithm (also known as \textit{union-find}):

\[
\text{for every element } i \text{ do } \text{f}ather(i) \leftarrow \Omega \\
\text{while input not exhausted do begin}
\begin{align*}
\text{get_pair}(a,b); & \\
\text{while } \text{father}(a) \neq \Omega \text{ do } a \leftarrow \text{father}(a); & \\
\text{while } \text{father}(b) \neq \Omega \text{ do } b \leftarrow \text{father}(b); & \\
\text{if } (a \neq b) \text{ then } \text{father}(a) \leftarrow b; & \\
\end{align*}
\text{end;}
\]

\[
\begin{array}{cccccc}
1 \equiv 5 & 5 & 2 & 8 & 2 & 8 \\
6 \equiv 8 & \Rightarrow & & & & k: 1 2 3 4 5 6 7 8 9 \\
7 \equiv 2 & 1 & 8 & 7 & 3 & 4 \\
9 \equiv 8 & & & & & \\
3 \equiv 7 & & & & & \\
\end{array}
\]

\text{father}(k): 5 2 8 2 8
k: 1 2 3 4 5 6 7 8 9
ALGORITHM

- Represent each element as a node in a 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;
```

![Diagram of forest of trees with merging nodes](attachment://forest_diagram.png)

- Algorithm (also known as *union-find*):

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

- Example:

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

Copyright © 1998 by Hanan Samet
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 \equiv 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;
```

- Algorithm (also known as union-find):

```plaintext
for every element i do father(i) \leftarrow \Omega
while input_not_exhausted do
  begin
    get_pair(a, b);
    while father(a) \neq \Omega do a \leftarrow father(a);
    while father(b) \neq \Omega do b \leftarrow father(b);
    if (a \neq b) then father(a) \leftarrow b;
  end;
```

father(k): 5 2 2 8 2 2 8
k: 1 2 3 4 5 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;

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;
```

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