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

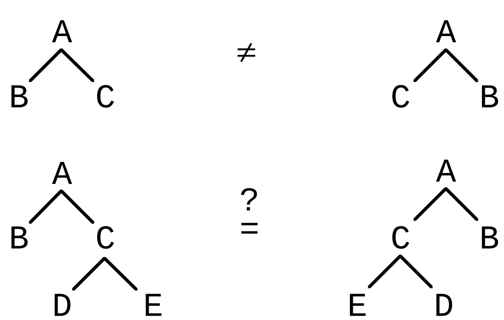
- TREE ≡ a branching structure between nodes
- A finite set T of one or more nodes such that:

  1. one element of the set is distinguished, ROOT(T)
  2. the remaining nodes of T are partitioned into $m \geq 0$ disjoint sets T_1, T_2, … T_m and each of these sets is in turn a tree.
     - trees T_1, T_2, … 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

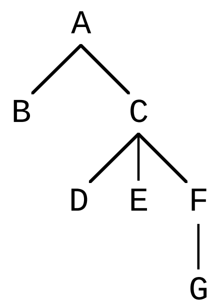
- ORDERED TREE ≡ if the relative order of the subtrees T_1, T_2, … T_m is important
- ORIENTED TREE ≡ order is not important



ordered

- Computer representation ⇒ ordered!

TERMINOLOGY



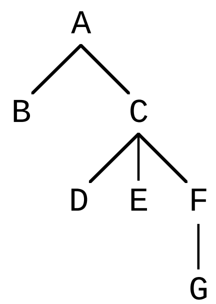
- Counterintuitive!

- 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

```
level(X) ≡ if father(X)=Ω then 0
           else 1+level(father(X));

Ex: level(G) = 1+level(F)
                 1+level(C)
                   1+level(A)
                     0
```

TERMINOLOGY



level 0

level 1

level 2

level 3

- Counterintuitive!

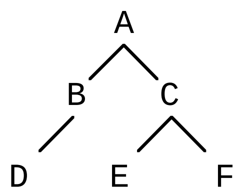
- DEGREE ≡ number of subtrees of a node
- Terminal node ≡ *leaf* ≡ degree 0
- BRANCH NODE ≡ non-terminal node

- 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

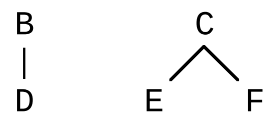
```
level(X) ≡ if father(X)=Ω then 0
          else 1+level(father(X));

Ex: level(G) = 1+level(F)
                1+level(C)
                  1+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



has the forest

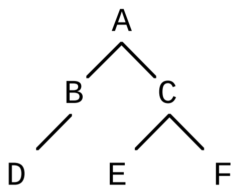


- BINARY TREE ≡ a finite set of nodes which either is empty *or* a root and two disjoint binary trees called the *left* and *right* subtrees of the root

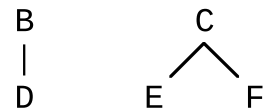
- Is a binary tree a special case of a tree?

FORESTS AND BINARY TREES

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

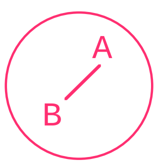


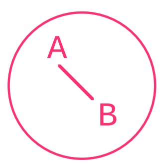
has the forest



- 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

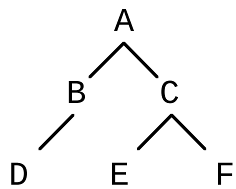




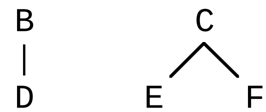
1 and 2 are different binary trees

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



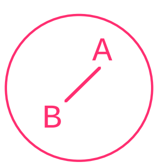
has the forest

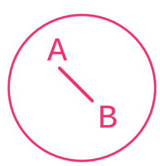


- BINARY TREE ≡ a finite set of nodes which either is empty *or* a root and two disjoint binary trees called the *left* and *right* subtrees of the root

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

NO! An entirely different concept



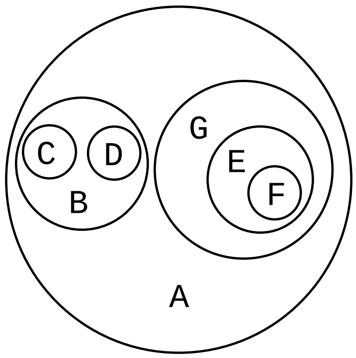


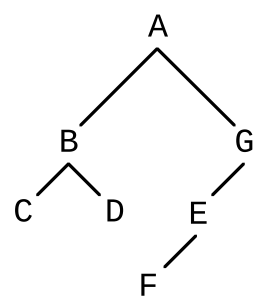
1 and 2 are different binary trees

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’)





- Nested parentheses

Tree (root subtree_1 subtree_2 ... subtree_n)

```
(A (B (C) (D)) (G (E (F))))
```

Binary tree (root left right)

```
(A (B (C () ()) (D () ()))
   (G (E (F () ()) ()) ()))
```

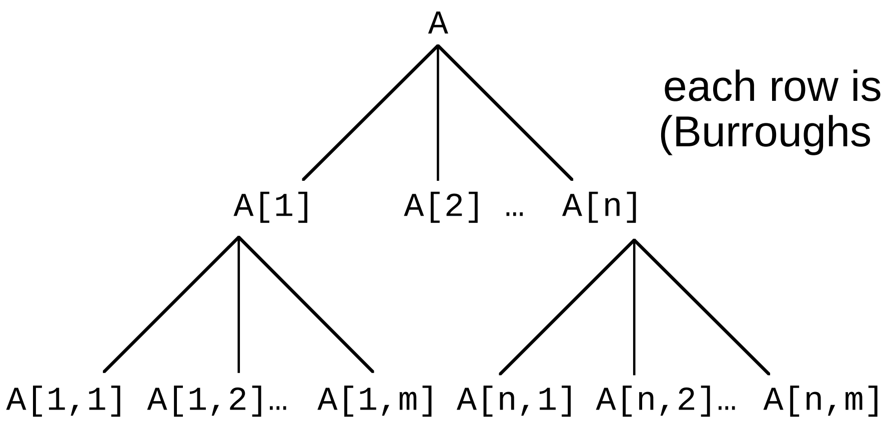
- Indentation

```
A
  B
    C
    D
  G
    E
      F
```

- Dewey decimal notation: `2.1 2.2.2 2.3.4.5`

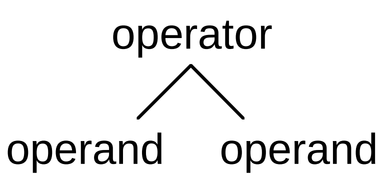
APPLICATIONS

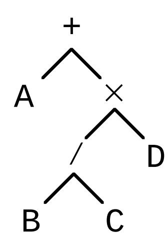
- Segmentation of large rectangular arrays – `A[n,m]`



each row is a segment
(Burroughs computers)

- Algebraic formulas





`A+((B/C)×D)`

1. no need for parentheses
   - but `A–B+C = (A–B)+C`
     `≠ A–(B+C)`

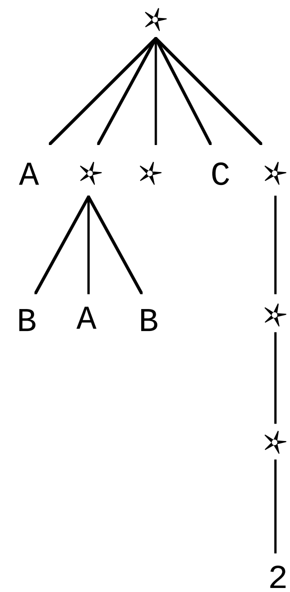
2. code generation

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

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

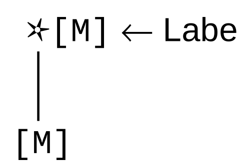


- Index notation:

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

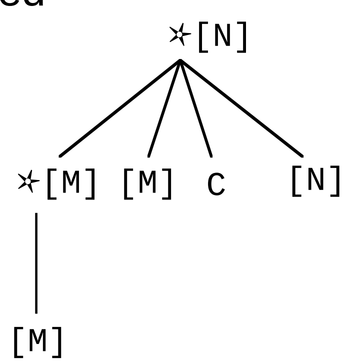
- Differences between LISTS and trees:

  1. no data appears in the nodes representing LISTS - i.e., ✢
  2. LISTS may be recursive `M=(M)`



  3. LISTS may overlap (i.e., need not be disjoint)
     - equivalently, subtrees may be shared

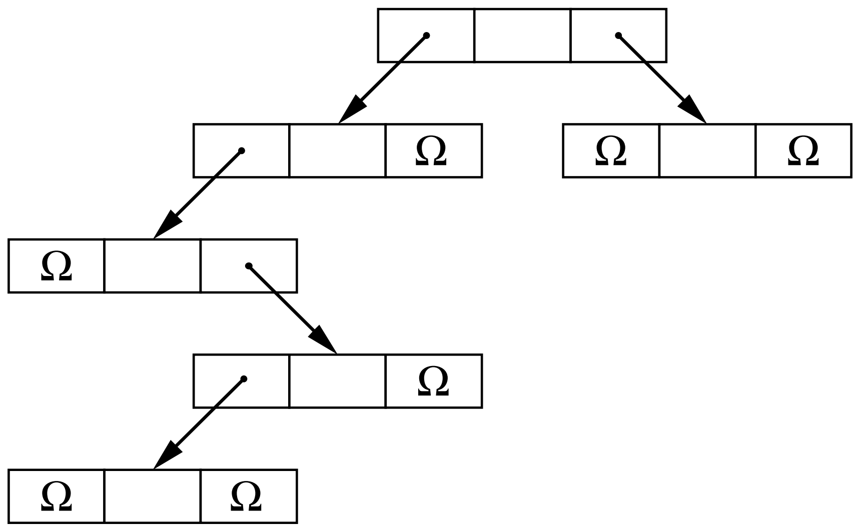
  `N=(M,M,C,N)`



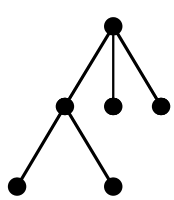
TRAVERSING BINARY TREES

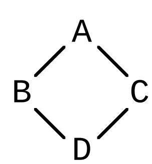
- Representation

LLINK	INFO	RLINK



- 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





ABD and ACD

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   K   G H  J
```

preorder =

inorder =

postorder =

- 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

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

inorder yields `10 15 20 30 45`

- Ex:

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

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

inorder =

postorder =

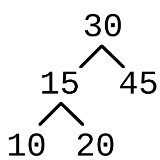
- Inorder traversal requires a stack to go back up the tree:

```
D

B

A
```

TRAVERSAL ORDERS

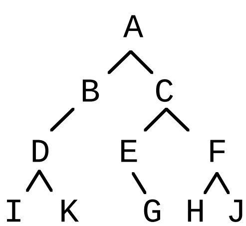
1. Preorder ≡ root, left subtree, right subtree
   - depth-first search
2. Inorder ≡ left subtree, root, right subtree
   - binary search tree
3. Postorder ≡ left subtree, right subtree, root
   - code generation

- Binary search tree: left < root < right



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 =

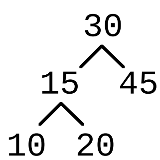
- Inorder traversal requires a stack to go back up the tree:

```
D

B

A
```

TRAVERSAL ORDERS

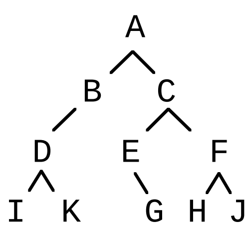
1. Preorder ≡ root, left subtree, right subtree
   - depth-first search
2. Inorder ≡ left subtree, root, right subtree
   - binary search tree
3. Postorder ≡ left subtree, right subtree, root
   - code generation

- Binary search tree: left < root < right



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
```

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;
```

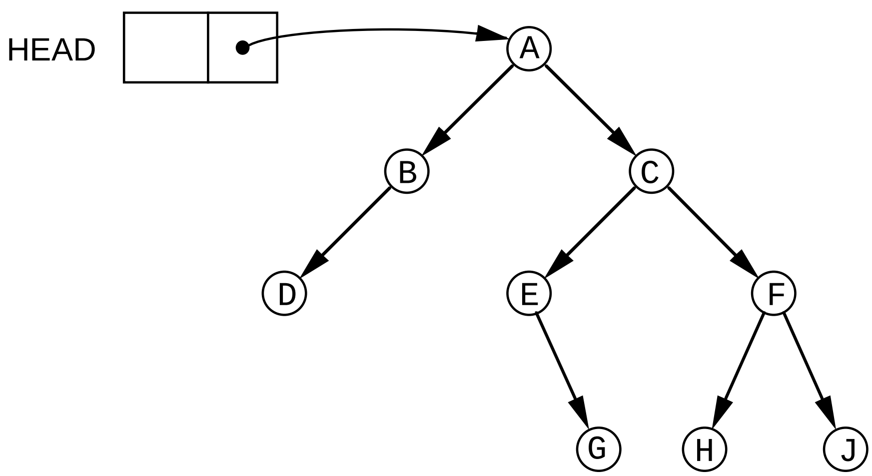
THREADED BINARY TREES

- 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

LLINK	LTAG	INFO	RTAG	RLINK

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!

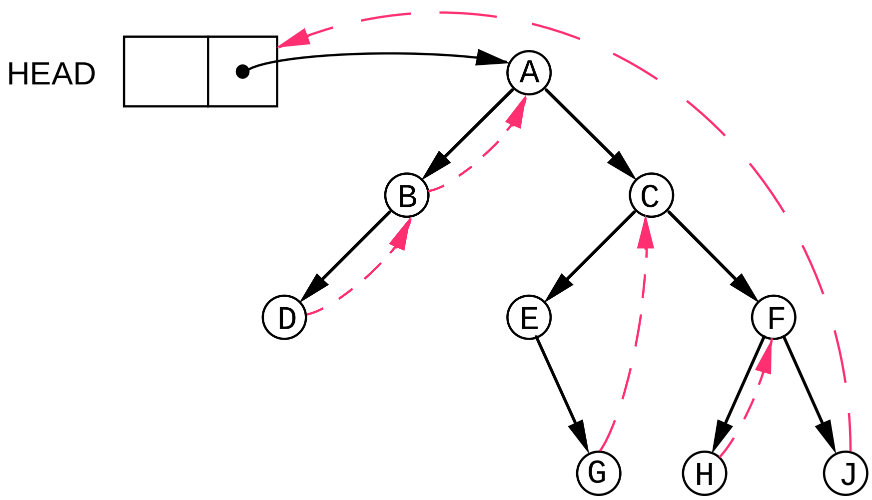
THREADED BINARY TREES

- 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

LLINK	LTAG	INFO	RTAG	RLINK

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!

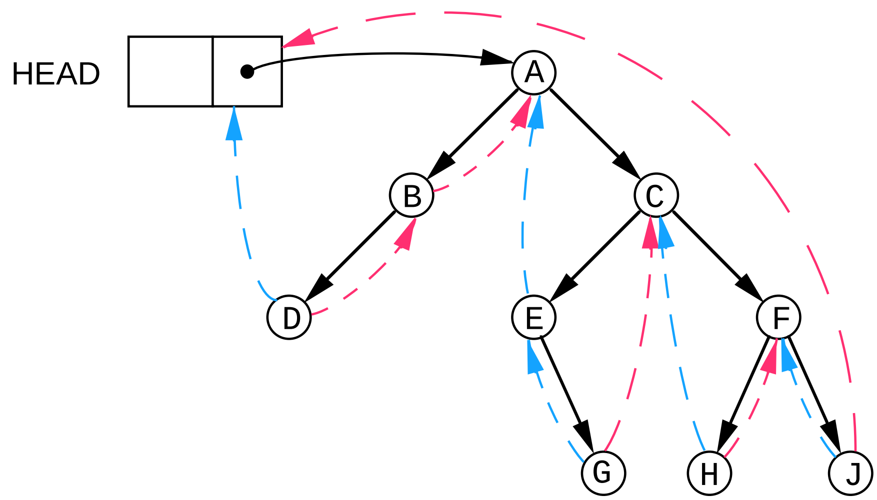
THREADED BINARY TREES

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

LLINK	LTAG	INFO	RTAG	RLINK

Ex:

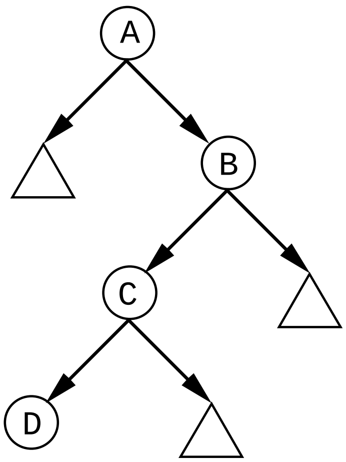


- If address of ROOT(T) < address of left and right sons, then don't need the TAG fields
- Threads will point to lower addresses!

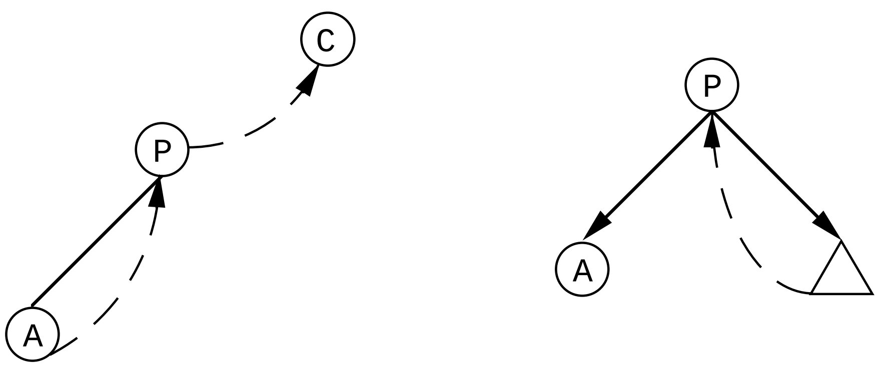
OPERATIONS ON THREADED BINARY TREES

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

```
1. Q←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

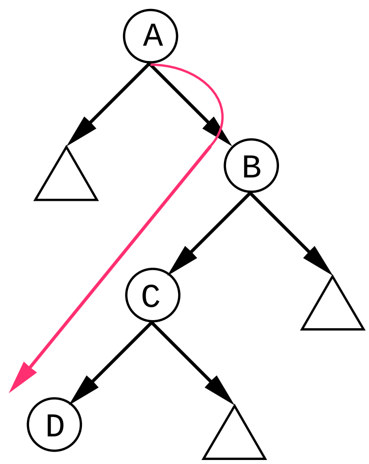



```
1.   RLINK(Q)←RLINK(P);    RTAG(Q)←RTAG(P);

     RLINK(P)←Q;           RTAG(P)←1;

     LLINK(Q)←P;           LTAG(Q)←0;

2.   if RTAG(Q)=1 then LLINK(Q$)←Q;
```

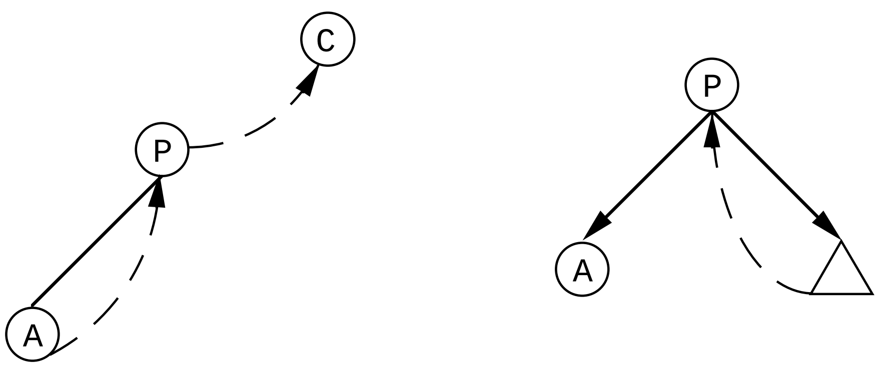
OPERATIONS ON THREADED BINARY TREES

- 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

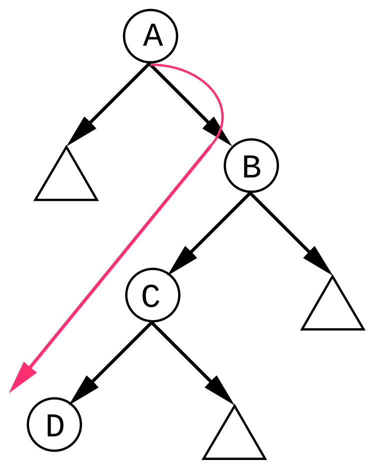



```
1.   RLINK(Q)←RLINK(P);     RTAG(Q)←RTAG(P);

     RLINK(P)←Q;            RTAG(P)←1;

     LLINK(Q)←P;            LTAG(Q)←0;

2.   if RTAG(Q)=1 then LLINK(Q$)←Q;
```

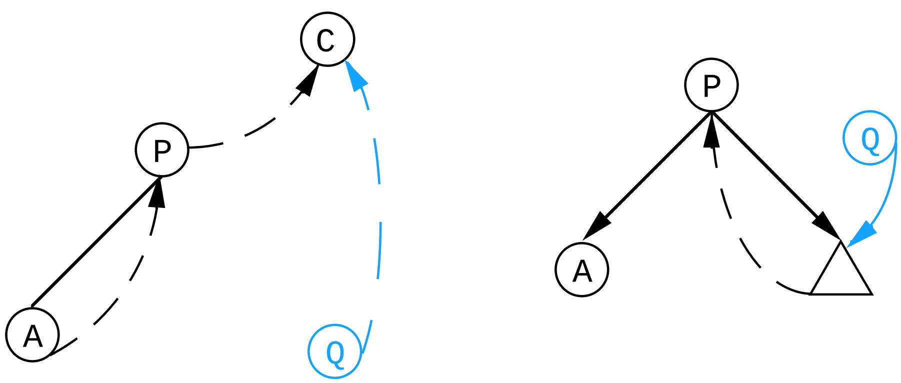
OPERATIONS ON THREADED BINARY TREES

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

```
1. Q←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

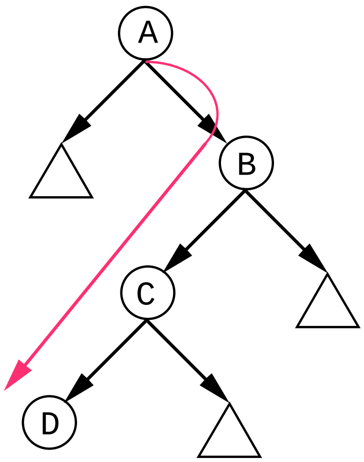



```
1.   RLINK(Q)←RLINK(P);    RTAG(Q)←RTAG(P);

     RLINK(P)←Q;           RTAG(P)←1;

     LLINK(Q)←P;           LTAG(Q)←0;

2.   if RTAG(Q)=1 then LLINK(Q$)←Q;
```

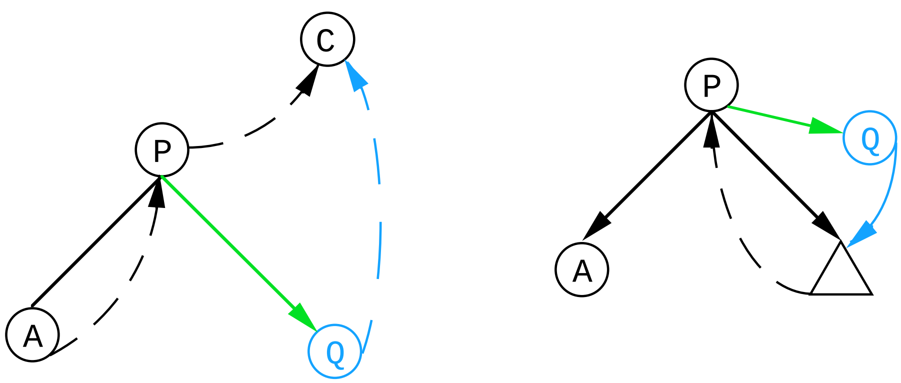
OPERATIONS ON THREADED BINARY TREES

- 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

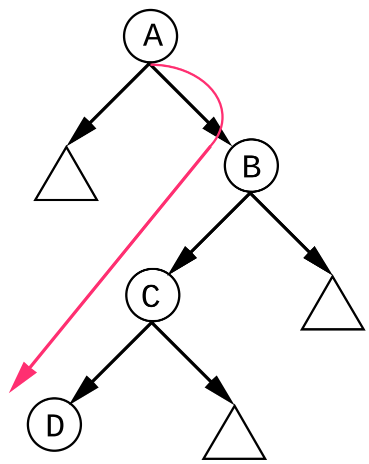



```
1.   RLINK(Q)←RLINK(P);     RTAG(Q)←RTAG(P);

     RLINK(P)←Q;            RTAG(P)←1;

     LLINK(Q)←P;            LTAG(Q)←0;

2.   if RTAG(Q)=1 then LLINK(Q$)←Q;
```

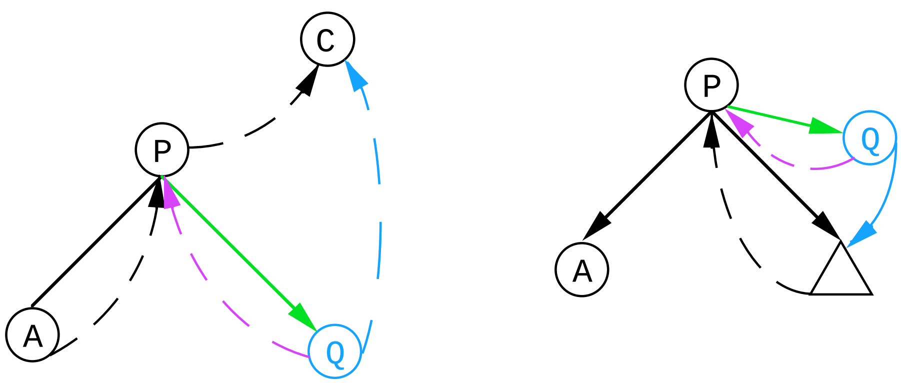
OPERATIONS ON THREADED BINARY TREES

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

```
1. Q←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

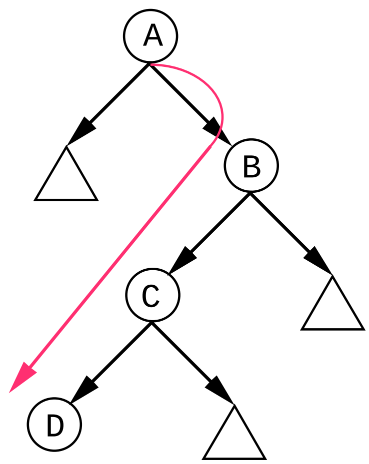



```
1.   RLINK(Q)←RLINK(P);     RTAG(Q)←RTAG(P);

     RLINK(P)←Q;            RTAG(P)←1;

     LLINK(Q)←P;            LTAG(Q)←0;

2.   if RTAG(Q)=1 then LLINK(Q$)←Q;
```

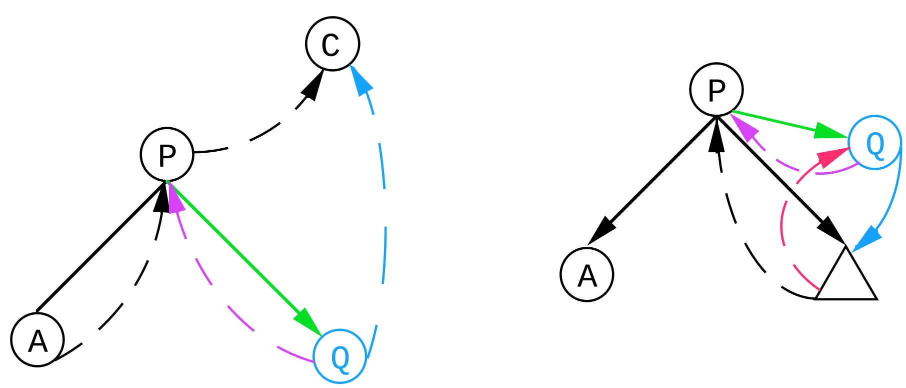
OPERATIONS ON THREADED BINARY TREES

- 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



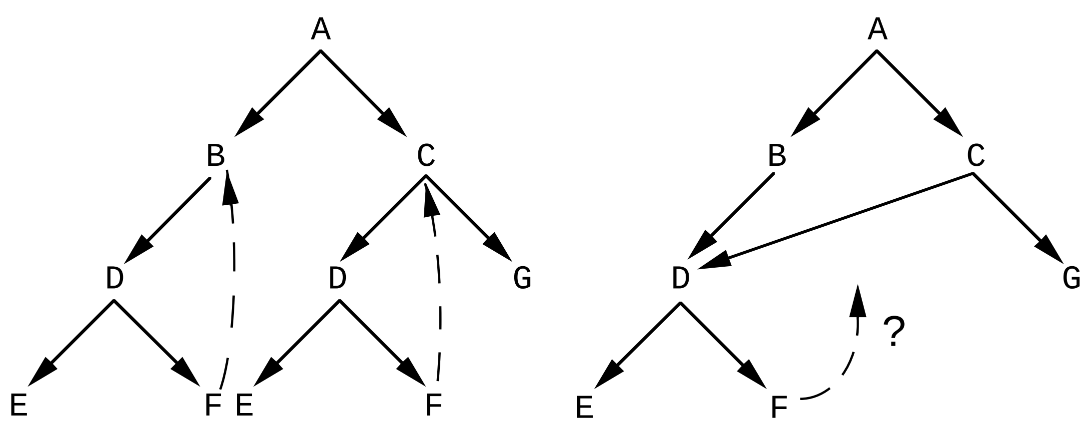

```
1.   RLINK(Q)←RLINK(P);     RTAG(Q)←RTAG(P);

     RLINK(P)←Q;            RTAG(P)←1;

     LLINK(Q)←P;            LTAG(Q)←0;

2.   if RTAG(Q)=1 then LLINK(Q$)←Q;
```

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

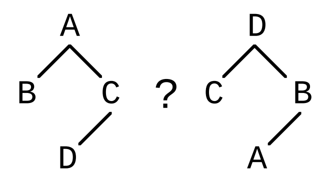
   Ex: two choices for the inorder successor of F



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 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

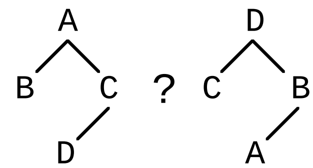



```
similar(T1,T2) =
   if empty(T1) and empty(T2) then T

   else similar(left(T1),left(T2)) and
        similar(right(T1),right(T2));
```

- 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



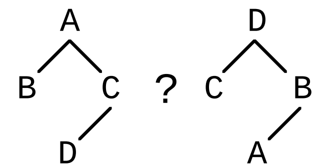

```
similar(T1,T2) =
   if empty(T1) and empty(T2) then T
   else if empty(T1) or empty(T2) then F
   else similar(left(T1),left(T2)) and
        similar(right(T1),right(T2));
```

- 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




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

- Will similar work?

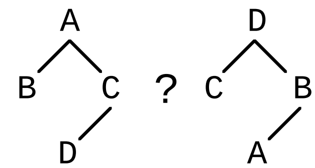
- No! base case does not handle case when one of the trees is empty and the other one is not

- Simplifying:

```
A and B =  if A then B        A or B =
           else F
```

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

- Will similar work?

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

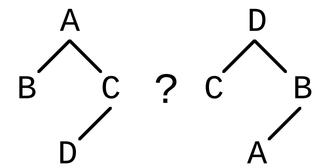
- Simplifying:

```
A and B =  if A then B          A or B =  if A then T
           else F                         else B
```

```
  similar(T1,T2) =
    if empty(T1) then
      if empty(T2) then T
      else F
    else if empty(T2) then F
    else if similar(left(T1),left(T2)) then
            similar(right(T1),right(T2))
    else F ;
```

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

- Will similar work?

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

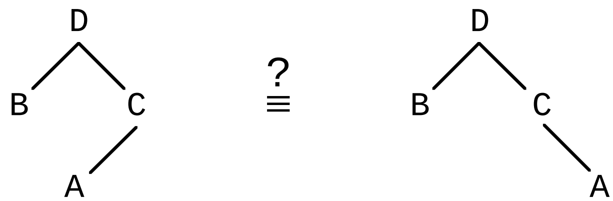
- Simplifying:

```
A and B =  if A then B          A or B =  if A then T
           else F                         else B
```

```
similar(T1,T2) =
  if empty(T1) then empty(T2)
   if empty(T2) then T
   else F
  else if empty(T2) then F                         and
  else if similar(left(T1),left(T2)) then
          similar(right(T1),right(T2))
  else F ;
```

EQUIVALENCE OF BINARY TREES

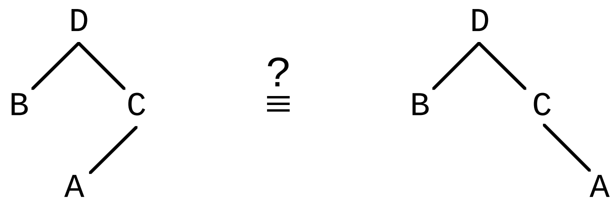
- Two binary trees T1 and T2 are said to be *equivalent* if they are similar *and* corresponding nodes contain the same information




```
equivalent(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));
```

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



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

```
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));
```

RECURSION SUMMARY

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

- Ex: Factorial

  $n! = n \cdot (n-1)\,!$

```
fact(n) = if n=0 then 1
          else n*fact(n-1);
```

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

```
fact(3) = 3*fact(2)
           2*fact(1)
             1*fact(0)
               1
        = 6
```

  Using an accumulator variable and a call `fact2(n,1)`:

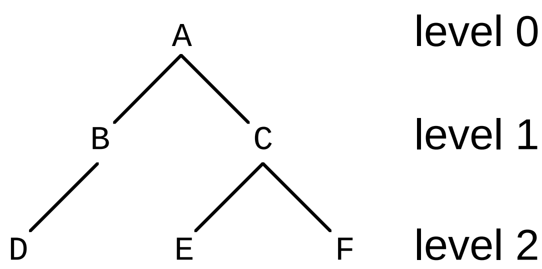
```
fact2(n,total) = if n=0 then total
                 else fact2(n-1,n*total);
```

  Solution is iterative

- Recursion implemented on computer using stack instructions.
- Dec-system 10: `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 Ω links are at the highest level), use a sequential storage allocation scheme so that links become unnecessary



1	A
2	B
3	C
4	D
5	Ω
6	E
7	F

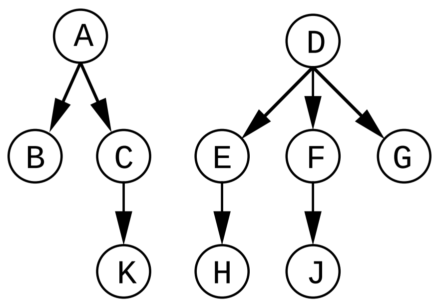
- If n is the highest level at which a node is found, then at most $2^{n+1} - 1$ words are needed

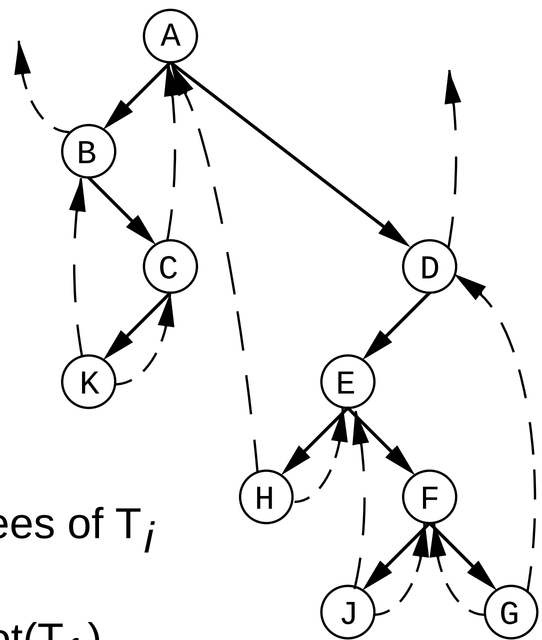
- Storage allocation method:
  1. root has address 1
  2. left son of x has address $2 * \text{address}(x)$
  3. right son of x has address $2 * \text{address}(x) + 1$

- 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 $x > (2^{n+1} - 1) / 3$

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 $B(T_2, T_3, \ldots, T_n)$

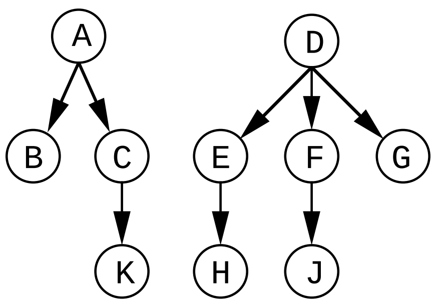
- Traversal of forests

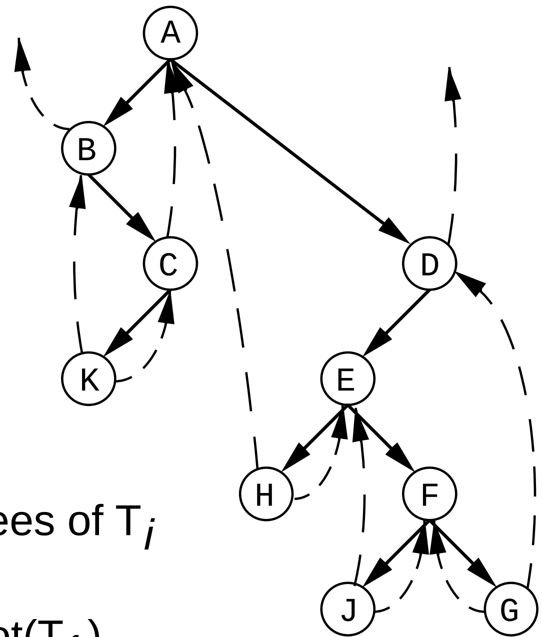
  preorder:
  1. visit root of first tree
  2. traverse subtrees of first tree in preorder
  3. traverse remaining subtrees in preorder

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

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, …, T_n)

  $T_{i,1}$, $T_{i,2}$, …, $T_{i,m}$ 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($T_{1,1}$, $T_{1,2}$, …, $T_{1,m}$)
     right subtree of B(F) is B(T_2, T_3, …, 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

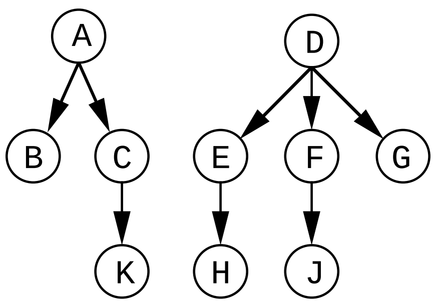
postorder:

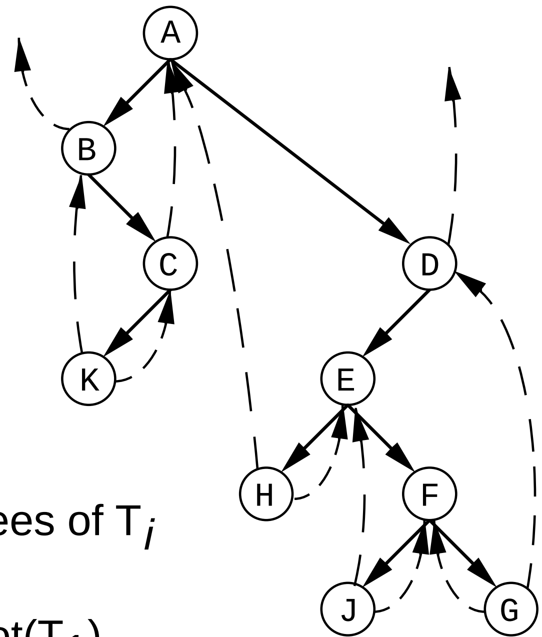
1. traverse subtrees of first tree in postorder
2. visit root of first tree
3. traverse remaining subtrees in postorder

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

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, …, T_n)

  $T_{i,1}$, $T_{i,2}$, …, $T_{i,m}$ 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($T_{1,1}$, $T_{1,2}$, …, $T_{1,m}$)

     right subtree of B(F) is B(T_2, T_3, …, T_n)

- Traversal of forests

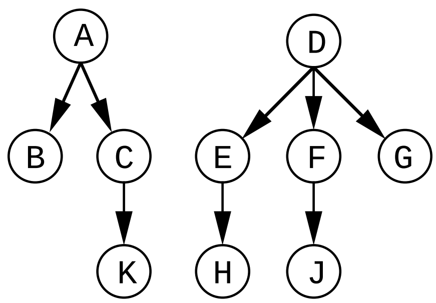
preorder:	postorder:
1. visit root of first tree	1. traverse subtrees of first tree in postorder
2. traverse subtrees of first tree in preorder	2. visit root of first tree
3. traverse remaining subtrees in preorder	3. traverse remaining subtrees in postorder

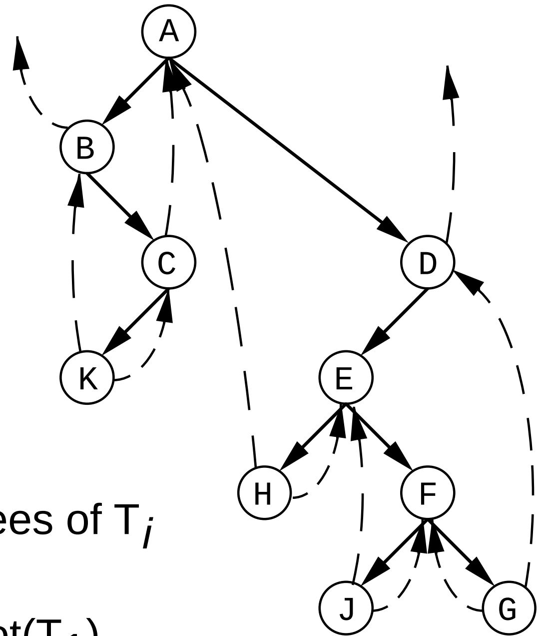
preorder = A B C K D E H F J G

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

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 $2 \equiv 6$?

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?

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 $2 \equiv 7 \equiv 3 \equiv 9 \equiv 8 \equiv 6$

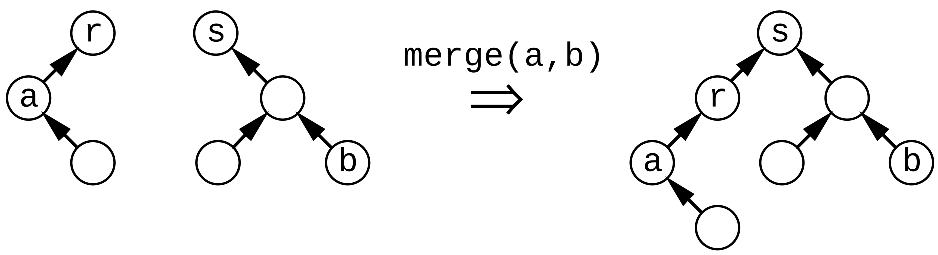
- Partitions S into disjoint subsets or *equivalence classes*
- Two elements equivalent iff they belong to same class
- What are the equivalence classes in this example?

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

ALGORITHM

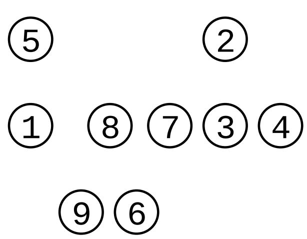
- Represent each element as a node in forest of trees
- Trees consist only of father links (nil at roots)
- Each (nonredundant) relation merges two trees into one
- Basic strategy:

```
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*):

```
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;
```

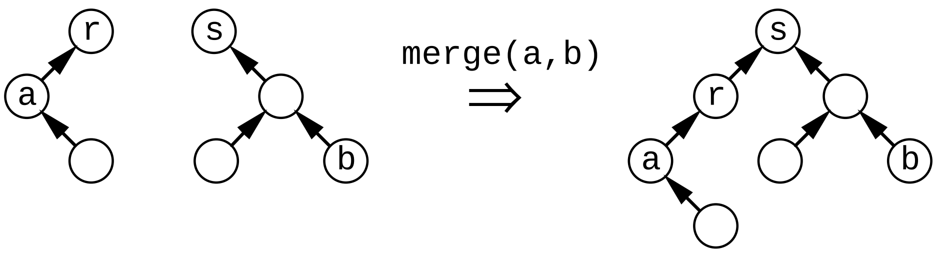



```
father(k):
        k: 1 2 3 4 5 6 7 8 9
```

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:

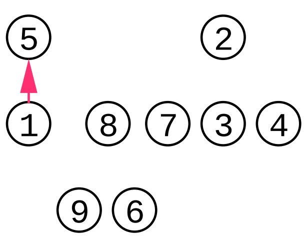
```
for each relation a≡b do
  begin
    find root node r of tree containing a; /* Find step */
    find root node s of tree containing b;
    if they differ, merge the two trees; /* Union step */
  end;
```



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

```
for every element i do 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;
```

1≡5

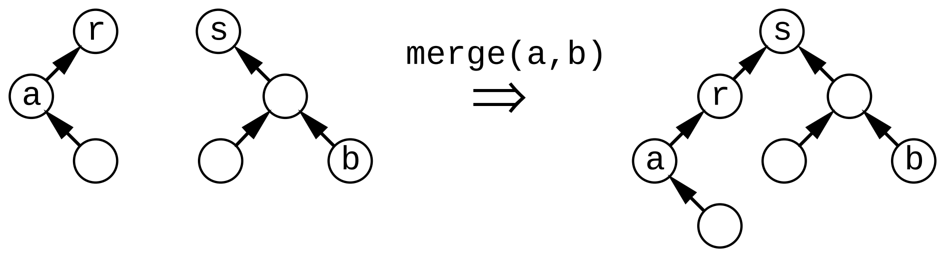


father(k): 5
k: 1 2 3 4 5 6 7 8 9

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:

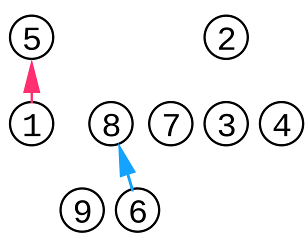
```
for each relation a≡b do
  begin
    find root node r of tree containing a; /* Find step */
    find root node s of tree containing b;
    if they differ, merge the two trees; /* Union step */
  end;
```



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

```
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;
```

1≡5
6≡8

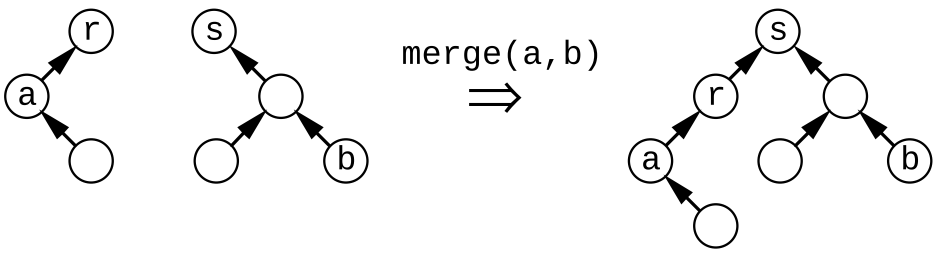


father(k):	5					8			
k:	1	2	3	4	5	6	7	8	9

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:

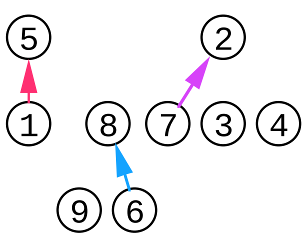
```
for each relation a≡b do
  begin
    find root node r of tree containing a; /* Find step */
    find root node s of tree containing b;
    if they differ, merge the two trees; /* Union step */
  end;
```



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

```
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;
```

1≡5
6≡8
7≡2

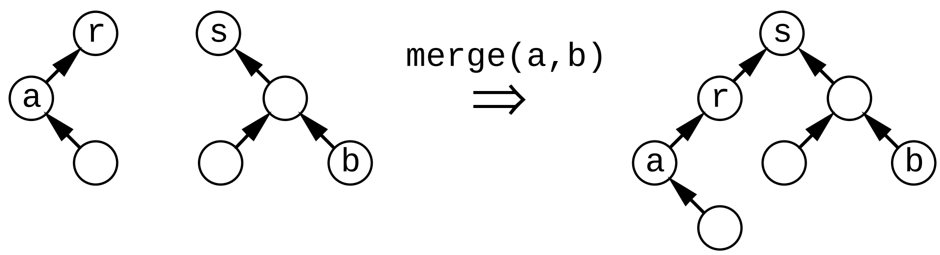


father(k):	5					8	2		
k:	1	2	3	4	5	6	7	8	9

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:

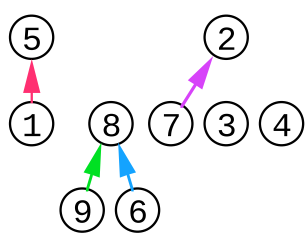
```
for each relation a≡b do
  begin
    find root node r of tree containing a; /* Find step */
    find root node s of tree containing b;
    if they differ, merge the two trees; /* Union step */
  end;
```



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

```
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;
```

1≡5
6≡8
7≡2
9≡8

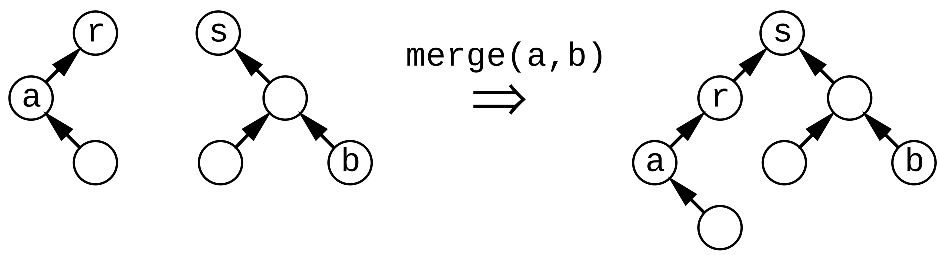


father(k):	5					8	2		8
k:	1	2	3	4	5	6	7	8	9

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:

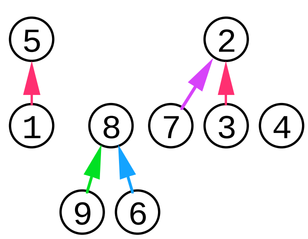
```
for each relation a≡b do
  begin
    find root node r of tree containing a; /* Find step */
    find root node s of tree containing b;
    if they differ, merge the two trees; /* Union step */
  end;
```



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

```
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;
```

1≡5
6≡8
7≡2
9≡8
3≡7

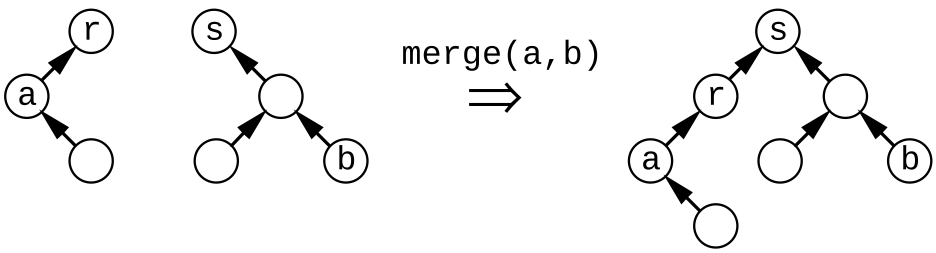


father(k):	5		2			8	2		8
k:	1	2	3	4	5	6	7	8	9

ALGORITHM

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

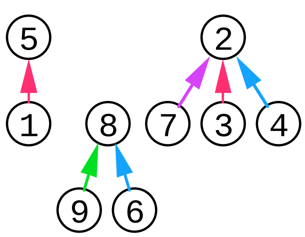
```
for each relation a≡b do
  begin
    find root node r of tree containing a; /* Find step */
    find root node s of tree containing b;
    if they differ, merge the two trees; /* Union step */
  end;
```



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

```
for every element i do 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;
```

1≡5
6≡8
7≡2
9≡8
3≡7
4≡2

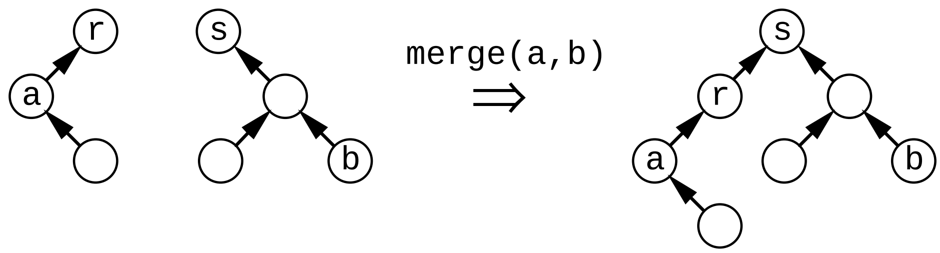


father(k):	5		2	2		8	2		8
k:	1	2	3	4	5	6	7	8	9

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:

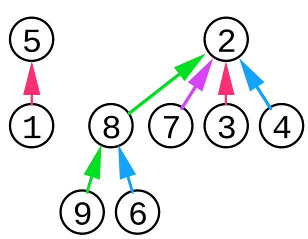
```
for each relation a≡b do
  begin
    find root node r of tree containing a; /* Find step */
    find root node s of tree containing b;
    if they differ, merge the two trees; /* Union step */
  end;
```



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

```
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;
```

1≡5
6≡8
7≡2
9≡8
3≡7
4≡2
9≡3

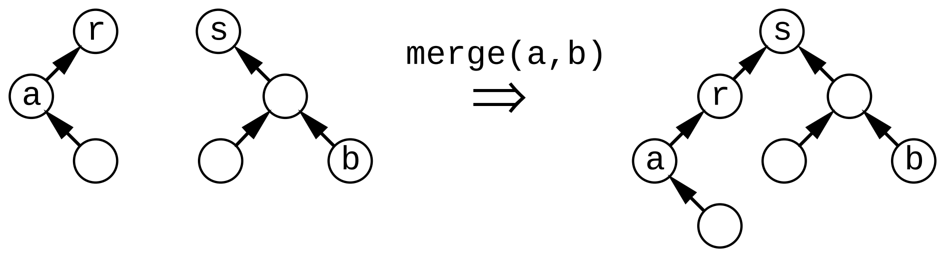


father(k):	5		2	2		8	2	2	8
k:	1	2	3	4	5	6	7	8	9

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:

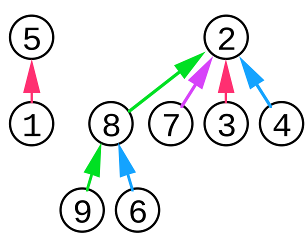
```
for each relation a≡b do
  begin
    find root node r of tree containing a; /* Find step */
    find root node s of tree containing b;
    if they differ, merge the two trees; /* Union step */
  end;
```



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

```
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;
```

1≡5
6≡8
7≡2
9≡8
3≡7
4≡2
9≡3



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)