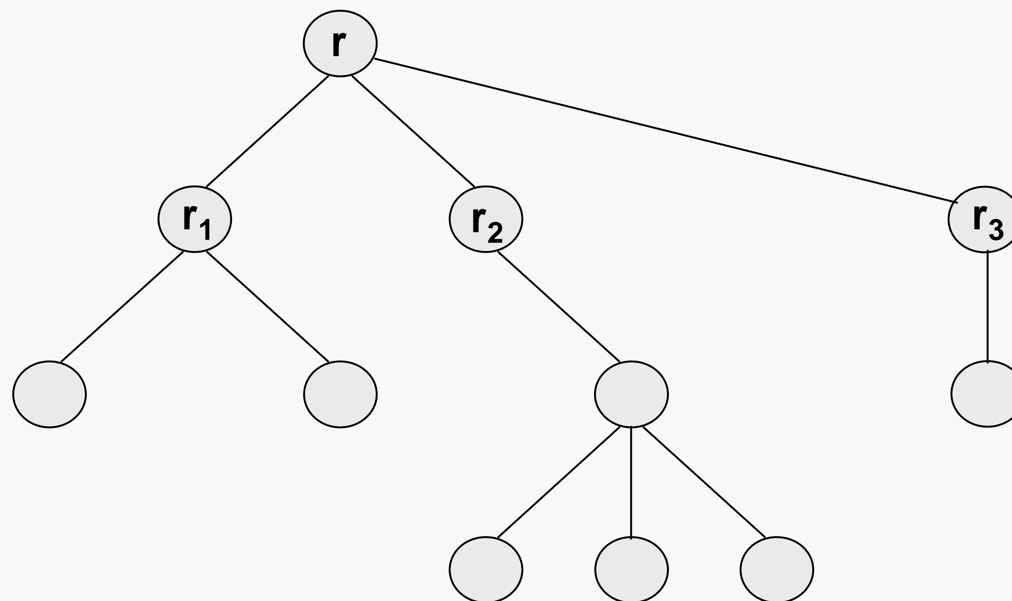
General Trees

A general tree T is a finite set of one or more nodes such that there is one designated node r, called the root of T, and the remaining nodes are partitioned into $n \geq 0$ disjoint subsets $T_1, T_2, \ldots, T_n$, each of which is a tree, and whose roots $r_1, r_2, \ldots, r_n$, respectively, are children of r.

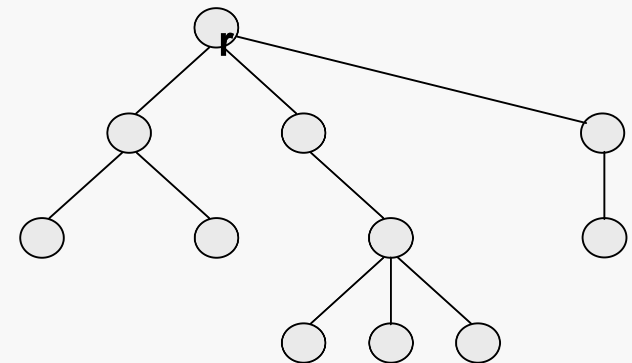


©2000 McQuain WD

Representing General Trees

The representation of a general tree poses some hard choices:

- **May the number of children per node vary over a large range (possibly with no logical upper limit)?**
- **Setting an upper limit renders some trees unrepresentable.**
- **Even with an upper limit, allocating a fixed number of pointers within each node may waste a huge amount of space.**
- **How can the pointers be organized for easy traversal? Does this require a secondary data structure within the node?**
- **Does the scheme provide for efficient search? node insertion? node deletion?**



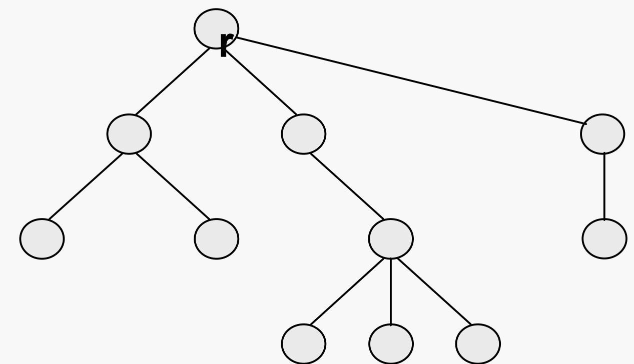
©2000 McQuain WD

Linked Nodes Implementation

The binary node type presented earlier can be extended for a general tree representation. The pointers can be managed by:

- **allocating a uniform-sized array of node pointers and limiting the number of children a node is allowed to have;**
- **allocating a dynamically-sized array of node pointers, custom-fit to the number of children the node currently has (and re-sizing manually as needed);**
- **using a linked list of node pointers;**
- **using a vector of node pointers, growing as needed.**

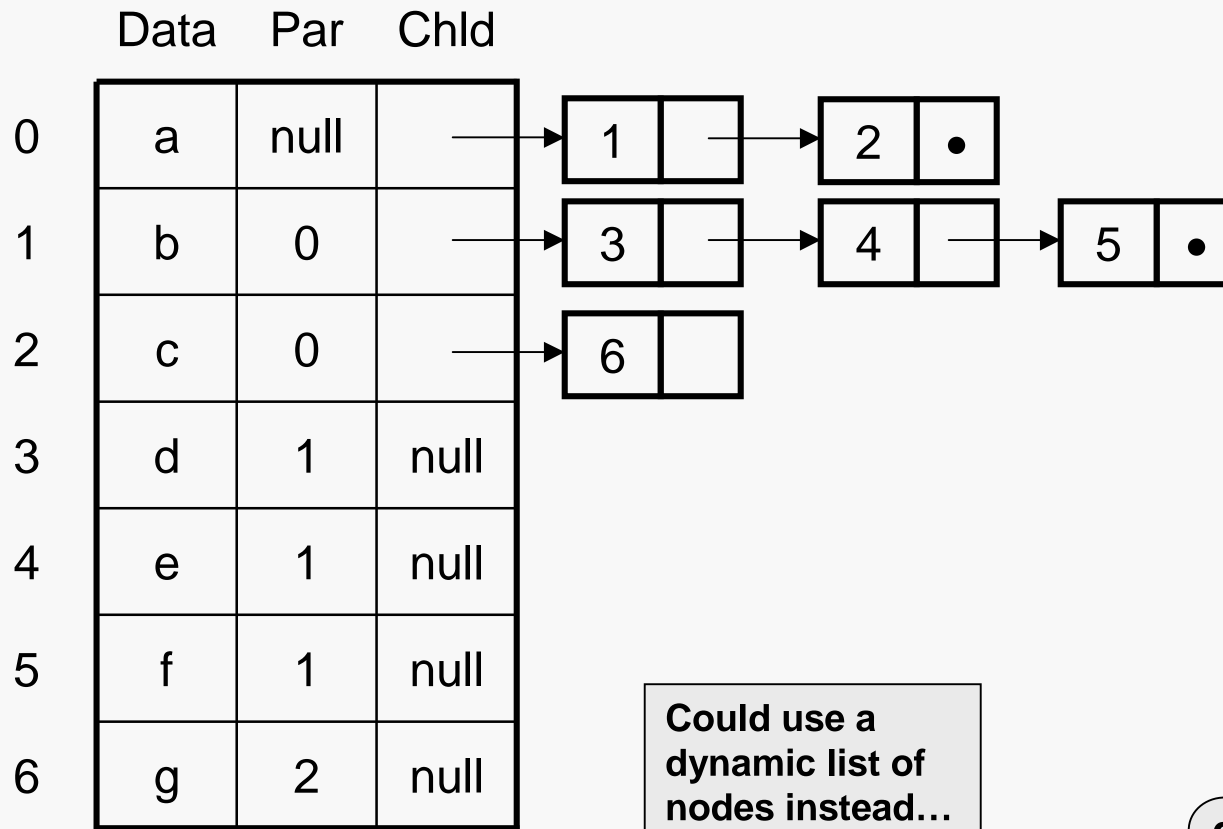
Each approach has pros and cons.

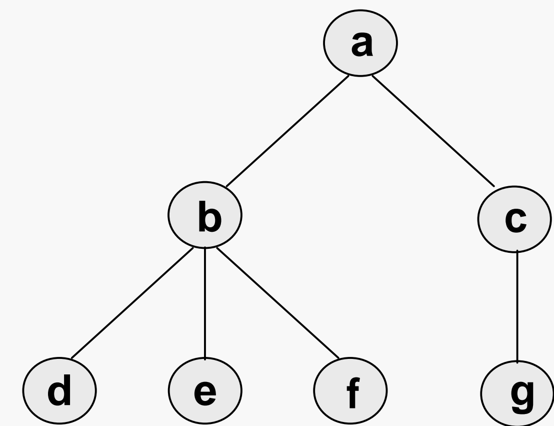


©2000 McQuain WD

List of Children Representation

For each tree node, store its data element, a (logical) pointer to its parent node, and a list of (logical) pointers to its children, using a array as the underlying physical structure:



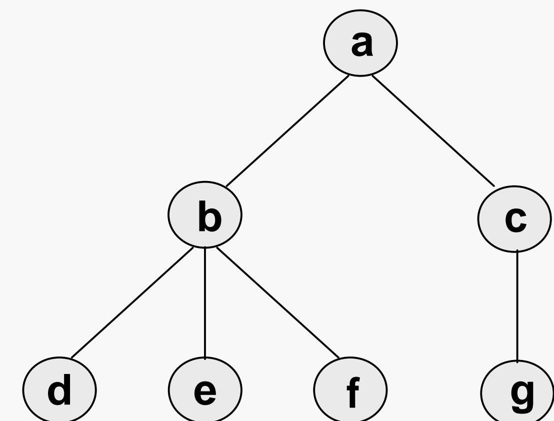


©2000 McQuain WD

Left-Children/Right-Sibling Representation

For each tree node, store its data element, a (logical) pointer to its parent node, and (logical) pointers to its left child and its right sibling, using a array as the underlying physical structure:

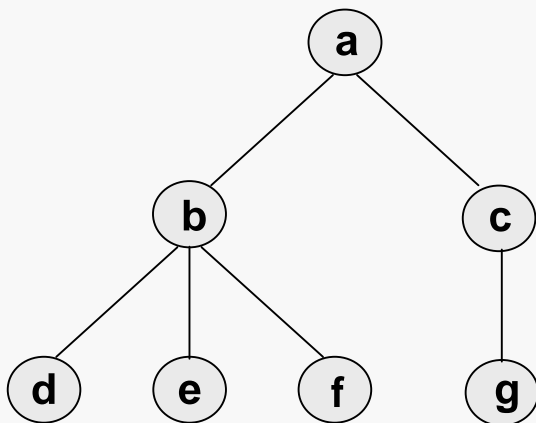
	Data	Par	Left Child	Right Sibling
0	a	null	1	null
1	b	0	3	2
2	c	0	6	null
3	d	1	null	4
4	e	1	null	5
5	f	1	null	null
6	g	2	null	null



©2000 McQuain WD

Parent Pointer Representation

For each tree node, store its data element and a (logical) pointer to its parent node, using a array as the underlying physical structure:



0	a	null
1	b	0
2	c	0
3	d	1
4	e	1
5	f	1
6	g	2

©2000 McQuain WD

Equivalence Relations

Let S be a set. An *equivalence relation* E on S is a collection of ordered pairs of elements of S such that:

- for every x in S, (x, x) is in E *reflexivity*
- for every x and y in S, if (x, y) is in E then (y, x) is also in E *symmetry*
- for every x, y and z in S, if (x, y) and (y, z) are in E then (x, z) is also in E *transitivity*

If (x, y) is in E then we say *x is equivalent to y*.

If x is in S, the set of all elements z of S such that (x, y) is in E is called the *equivalence class* of x, denoted [x].

©2000 McQuain WD

Example

Let S be the set of integers 0 through 20. Define an equivalence relation E on S:

x is equivalent to y if and only if x % 3 and y % 3 are equal.

Then:

[0] = {0, 3, 6, 9, 12, 15, 18}

[1] = {1, 4, 7, 10, 13, 16, 19}

[2] = {2, 5, 8, 11, 14, 17, 20}

Note that every element of S is in exactly one of these equivalence classes, and that no two different equivalence classes have any elements in common.

©2000 McQuain WD

Theorems on Equivalence Classes

Thm: Let E be an equivalence relation on a set S. Then if x and y are elements of S, either $[x] = [y]$ or $[x] \cap [y] = \emptyset$.

Thm: Let E be an equivalence relation on a set S. Then S equals the union of the distinct equivalence classes under E.

The latter theorem is usually described as saying that an equivalence relation partitions a set S into disjoint subsets, rather like cutting a piece of paper into a jigsaw puzzle.

So what does this have to do with trees? We can represent each equivalence class as a general tree, and a partitioning as a collection (forest) of such trees.

©2000 McQuain WD

Example

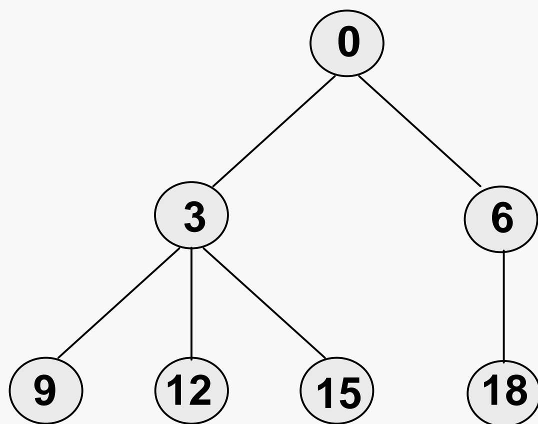
Recalling the earlier example:

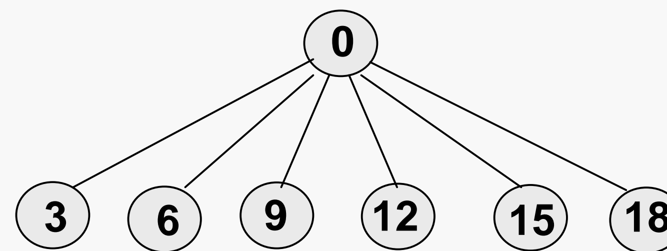
[0] = {0, 3, 6, 9, 12, 15, 18}

[1] = {1, 4, 7, 10, 13, 16, 19}

[2] = {2, 5, 8, 11, 14, 17, 20}

The equivalence class [0] may be represented by (any) general tree containing the listed values. For example:





Given this representation, how do we determine if two values are equivalent?

©2000 McQuain WD

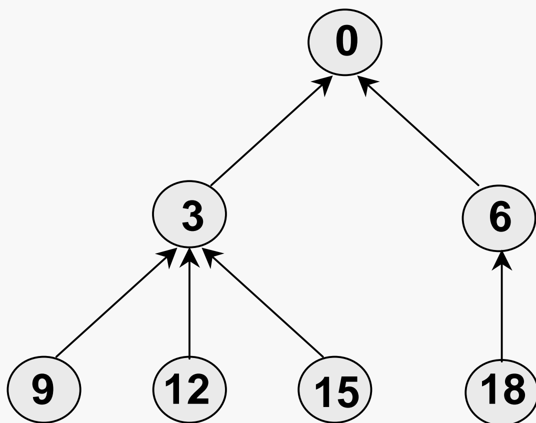
Determining Equivalence

Two values are equivalent if they are in the same tree.

Two values are equivalent if the root nodes of their trees are the same.

We can find the root node by following parent pointers upward as far as possible.

For this problem, no other type of traversal is necessary, so the parent "pointer" representation described earlier is sufficient.



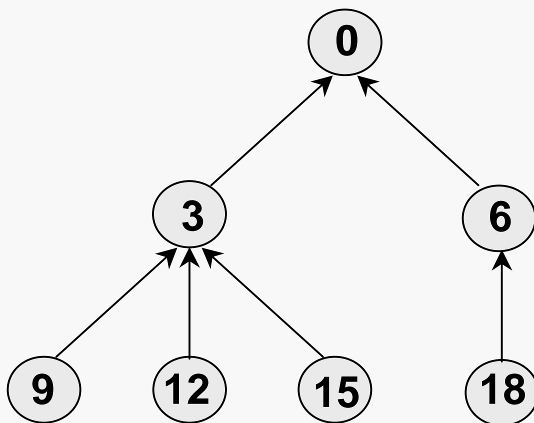
©2000 McQuain WD

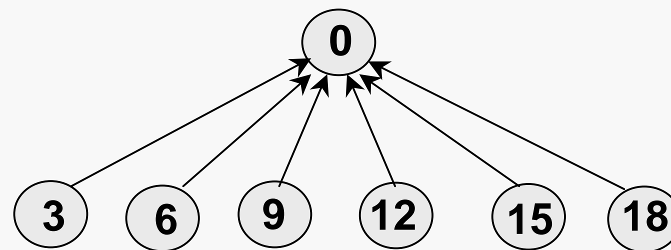
Path Compression

However, all equivalence trees are not equally useful for determining equivalence.

The tree on the right (below) is preferred because the path from each node to the root is of minimum length.

Given an equivalence tree, such as the one on the left, we may improve its performance by "compressing" it vertically. Whenever we start at a value, say x, and search for its root, we may then attach the node containing x directly to the root. This is known as *path compression*.





©2000 McQuain WD

Building an Equivalence Forest

Given a set S and data specifying which elements are equivalent (under some equivalence relation E) we may build a collection of trees that represent the equivalence classes of E.

Initially, each element is it's own class (all nodes are isolated).

Two elements, x and y, are equivalent if and only if the roots of their respective trees are the same.

Discovering that two elements, x and y, are equivalent implies that their equivalence classes must be merged (their respective trees must be joined in some manner).

Two general trees may be joined by making the root of one a child of the root of the other.

©2000 McQuain WD

Example

Let S = {A, B, C, D, E, F, G, H, I, J}. Initially:

0	1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I	J
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Given B~A:

0	1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I	J
-1	0	-1	-1	-1	-1	-1	-1	-1	-1

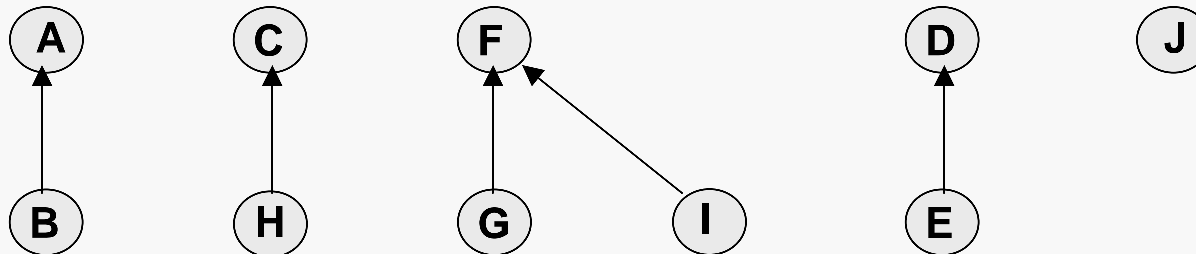
Given H~C,
G~F,
I~F,
E~D:

0	1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I	J
-1	0	-1	-1	3	-1	5	2	5	-1

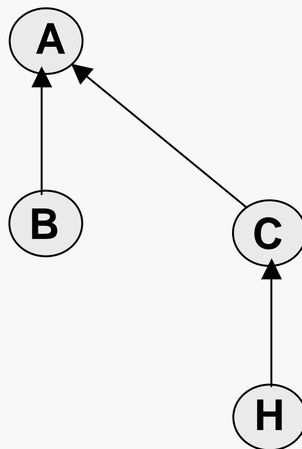
©2000 McQuain WD

Example

The last state corresponds to the forest:



Now, given that C~B we need to merge the first two trees:



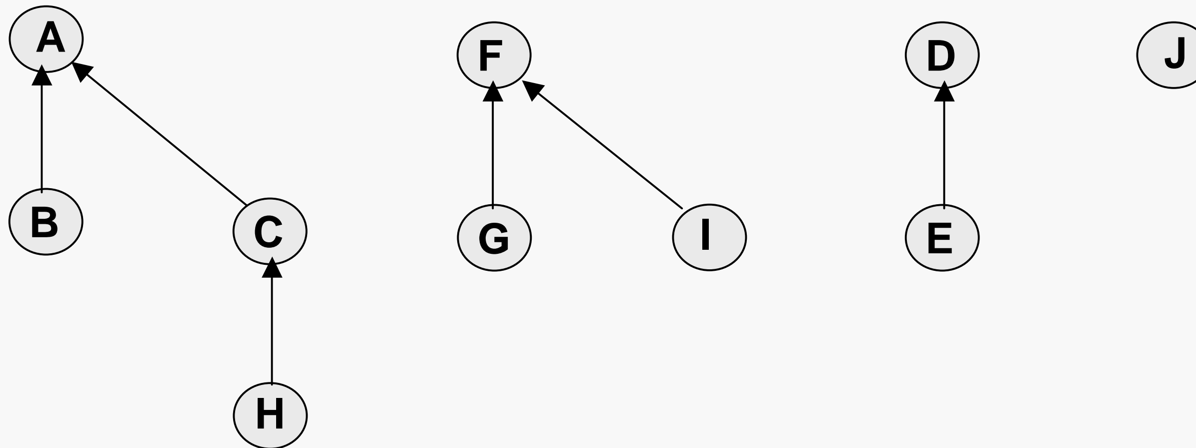
First find the root of each tree, and then make one root the parent of the other.

At this point, the choice of which is to be the parent is arbitrary.

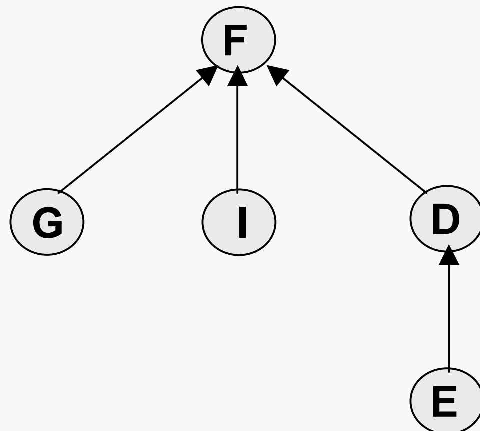
©2000 McQuain WD

Example

Now we have the following forest. Suppose we are given that E~G…



Again, we need to merge two trees:



Here, we apply the *Weighted Union Rule*:

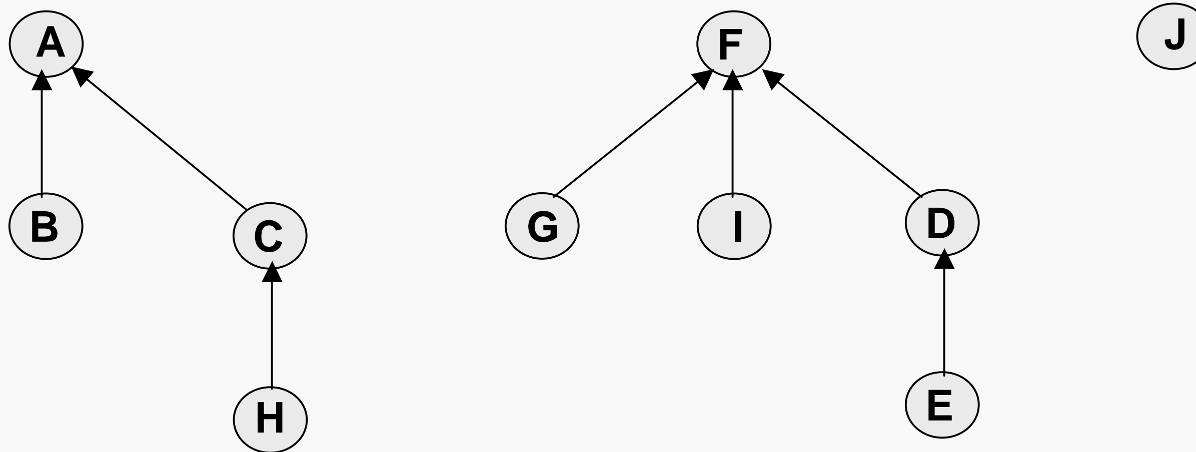
attach the smaller tree to the root of the larger tree.

This reduces the average node depth.

©2000 McQuain WD

Example

The end result is the following forest:



…which corresponds to the following parent pointer representation:

0	1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I	J
-1	0	0	5	3	-1	5	2	5	-1

©2000 McQuain WD