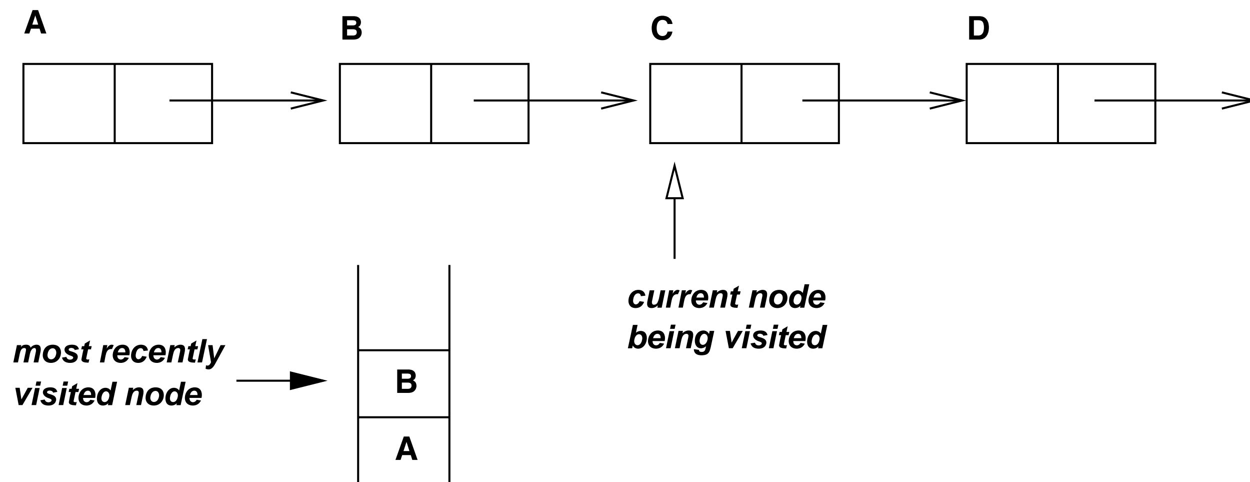
Linked List Traversal

Suppose you have a singly linked list and you traverse it up to some node. If you need, say, to go backwards a few nodes, there is ostensibly no way to do it without starting all over from the front again.

Or is there?

One way to do so without starting over again is to maintain a *stack* of the references of previously visited nodes. Then to go backwards, we simply pop the node references.

Keeping traversal path in a stack



Potential problem

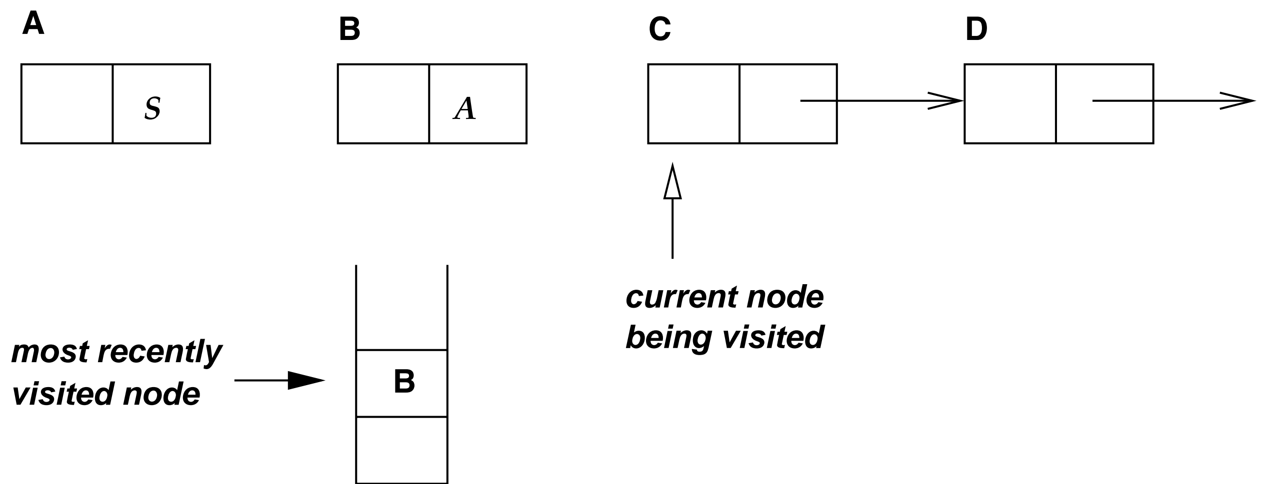
The stack solution is fine, but it requires in the worst case a stack depth as large as the linked list size. In which case, perhaps we should simply implement a *doubly linked list* instead.

Or is there a sneaky way to get around this?

Observe that in the stack information for the path traversed, the stack addresses *duplicate* the reference field information in the traversed nodes. Is there a way to avoid this?

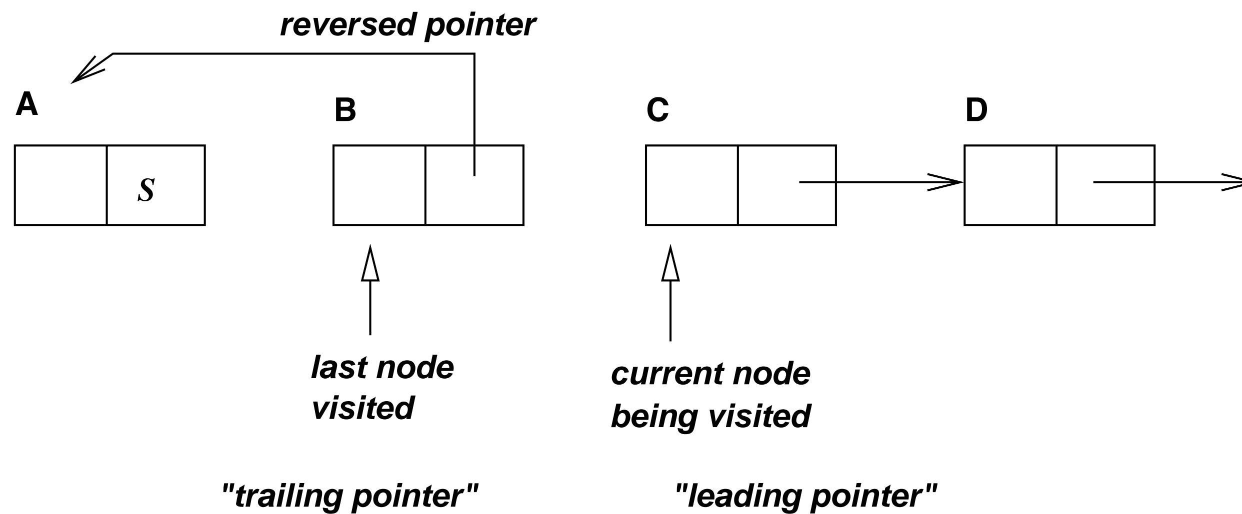
Yes! *Embed the stack in the reference fields of the traversed nodes.*

Past trail embedded in past reference fields



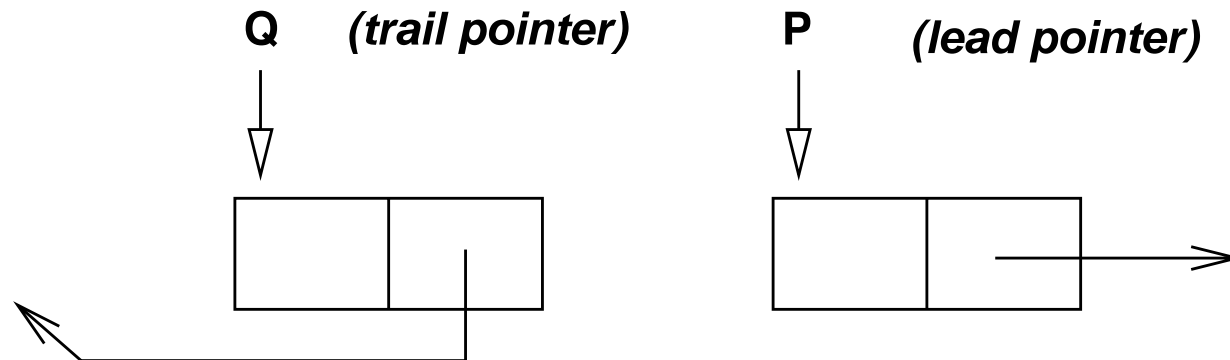
S = "start indicator"
A is now embedded in reference of B.
But to get rid of the stack we need a reference to B.

Pointer reversal and Lead/Trail Pointer Pair



Forward traversal uses the lead/trail pointer pairs, updating them in the obvious way. Backward traversal reverses their roles.

Forward/Backward are Duals




```
FORWARD:     forward = P.link;
            P.link = Q;          /* link reversed
            Q = P;
            P = forward;

BACKWARD:    back = Q.link;
            Q.link = P;          I* link restored
            P = Q;
            Q = back:
```

You cannot tell direction!

Important observation: Imagine you are in the middle of a long linked list with reversed pointers. *Just by looking at the local neighborhood there is no way to tell if you are moving "forward" or "backward" as the situation and code is completely symmetric!*

Therefore, if you must know the "direction", some extra information (in this case a Boolean flag will do) must be kept. If you do not do that, you run the risk of an infinite loop since the "start link" and "end link" could simply have been set to *null*.

This is critical for pointer-reversal techniques in general.

Local symmetry can be exploited

Here is a way to use the local symmetry of the traversal code to have *ONE* method to do first a forward then a backward traversal on the same list.

If the movement skeleton code with parameters $P1, P2$ is:

```
P1, P2 : Node;
Move = P1.link;
if Move = null then return {XXX};  \* throw something here?
P1.link = P2;
P2 = P1;
P1 = Move;
```

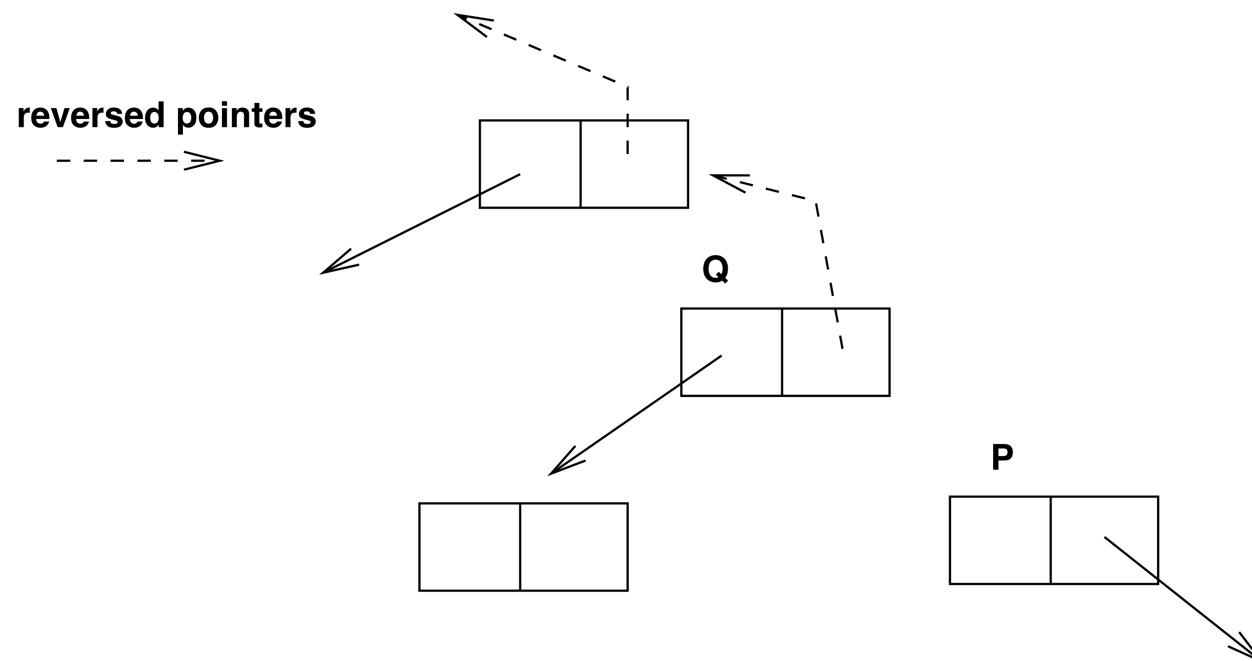
The external call to this method is responsible for setting the Flags for indicating/changing direction of traversal, and detecting execptions.

Tree traversal with pointer reversal

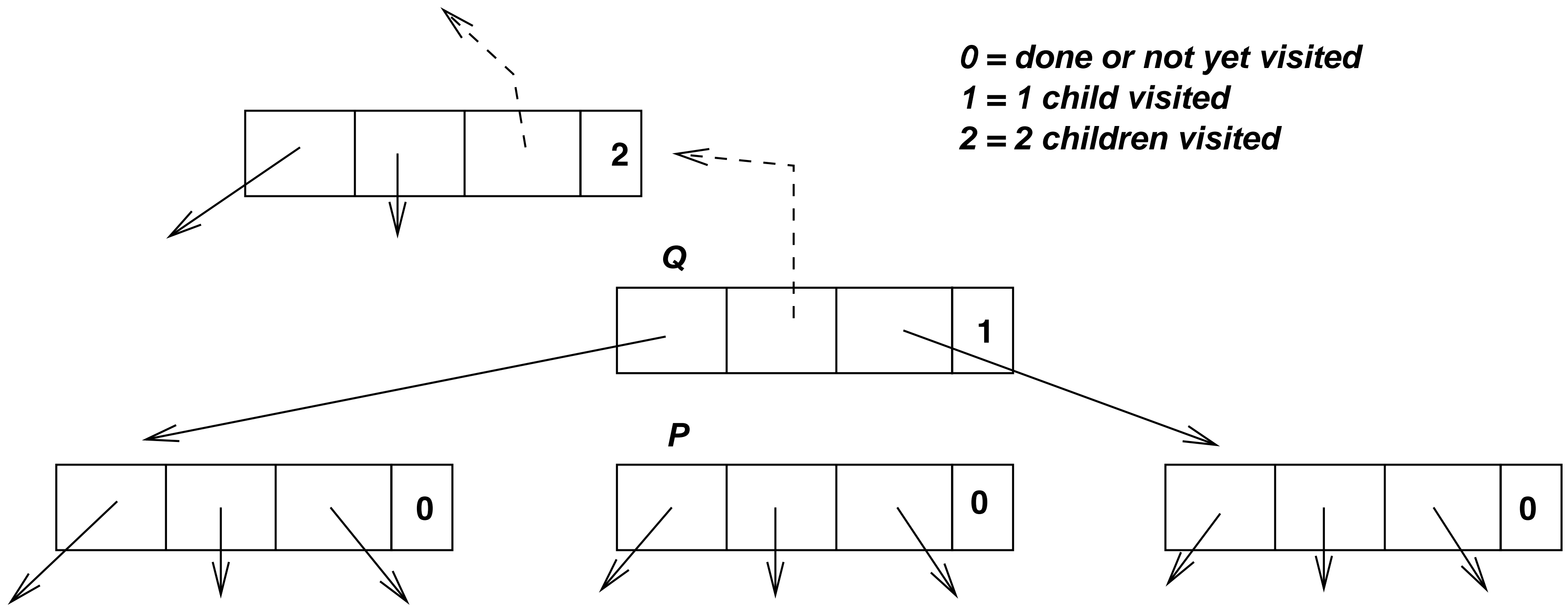
The same idea extends to tree traversal. The normally recursive traversal with *implicit stacks* can be made iterative by embedding the stack in the nodes using pointer reversal.

This is a small generalization of the linked list case. The topology of the reversed trail reflects the order of tree traversal.

Binary tree inorder traversal with pointer reversal



Arbitrary m-ary trees OK too



Snapshot of 3–ary tree traversal

Need a "children visited" count

The main difference with binary trees is that these m-ary trees need a count of how many children have been visited, so that when the traversal comes back to it from "below", the algorithm will know to stop looking for more children and go back "up". The count is is incremented each time a new child is picked up, and reset to 0 on the way up.

This count can be made a little more sophisticated.

An alternative is to forget counts and have a special link field (reference in it is distinct from null) that indicates "no more children" and is a signal to go back up the tree. This is good also for nodes of arbitrary arity.

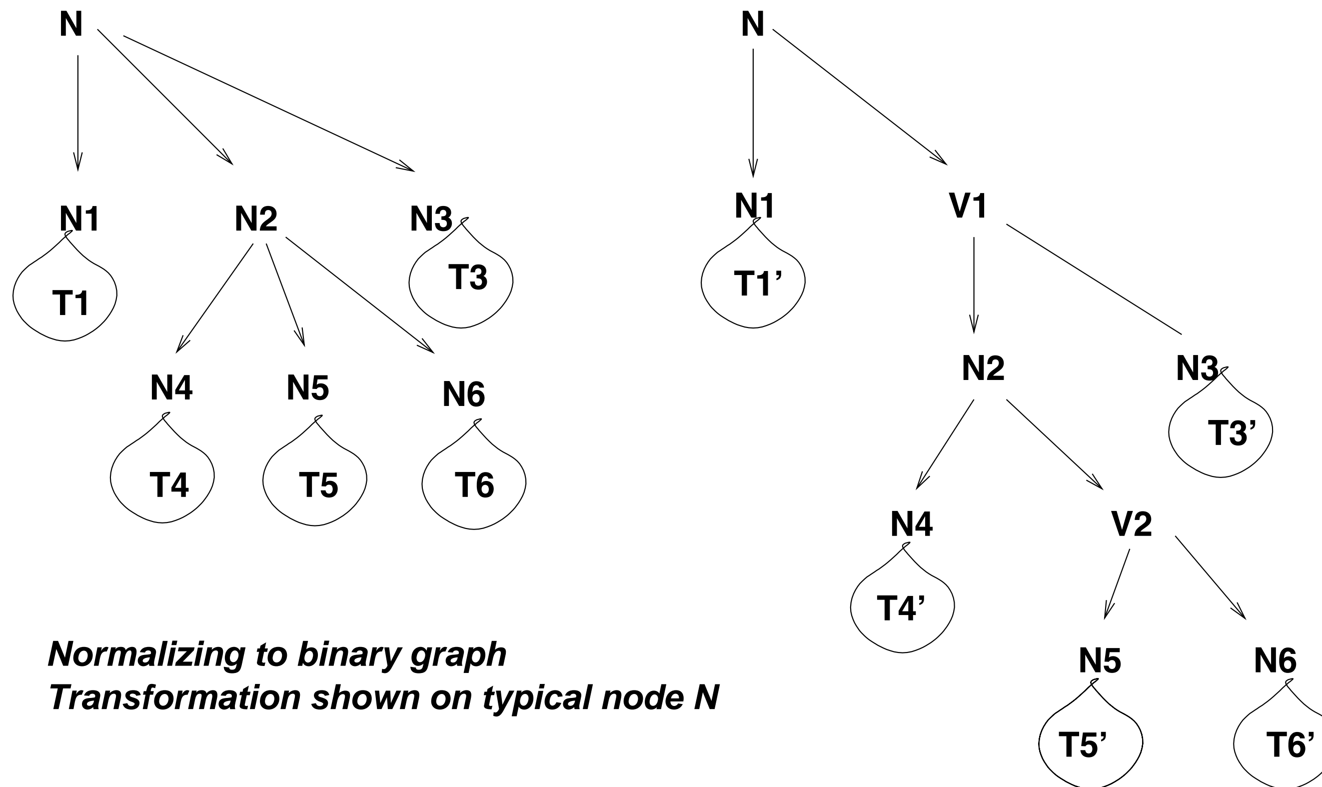
Normalizing arbitrary arity

In computer science, if one does not wish to deal with arbitrary, or even fixed m-arity for $m > 2$, there is standard *normalization* technique that will convert the data into *binary* (or *dyadic*) arity.

The idea, illustrated for a ternary relation R is this. We define $R(X, Y, Z) \leftrightarrow S(X, Y) \wedge T(Y, Z)$, for two new dyadic relations S and T. Then any triple $< x, y, z >$ is in R iff the pair $< x, y >$ is in S and the pair $< y, z >$ is in T.

This is the basis of making *uniform* nodes for arbitrarily connected graphs of data.

Normalization Idea



Graphs

A tree is a connected acyclic graph; a computer science tree is usually a rooted tree.

Therefore, in a tree traversal there are no separated paths that lead to the same node.

Can pointer-reversal techniques be used for *graph* traversal where two separated paths can converge to the same node (at different times) in the same traversal, i.e, paths "cross"?

The basic idea to handle this is a version of the *Ariadne Thread Algorithm*.

Theseus and Ariadne

The Minotaur in Crete, a half-bull half-man, demanded human victims annually from Crete's vassal Greece. The Athenian Prince Theseus volunteered to join the annual victims so that he could attempt to slay the Minotaur. Princess Ariadne of Crete fell in love with Theseus, and gave him a big ball of thread, advising him to unwind it as he went about the Labyrinth where Minotaur dwelled, awaiting his vitims to get lost before he set upon them. Ariadne said,"If you spy the thread crossing a junction, do not go into those passages through which the thread passes, but into a new one."

Theseus successfully avoided getting lost, and slew the Minotaur.

(The Aegean Sea was named after Theseus' father King Aegeaus, who drowned in it under unspeakably tragic circumstances. Theseus, the ungrateful wretch, dumped Ariadne later. He became, however, a wise king. The Minotaur was actually Ariadne's half sibling, conceived embarrassingly.)

Deutsch-Waite-Shorr Marking Algorithm

The modern version of Ariadne's Thread is the DWS marking algorithm using pointer reversal to embed the trail stack.

It *marks* a node the first time it is visited, so that it will know that it has been there before. If it sees such a mark, it *backtracks*.

It assumes binary normalized nodes.

In each node the link fields are *Link1* and *Link2*. In addition it has two Boolean fields, *Mark* and *Atom*. *Mark*'s function is obvious, set to 1 when first visited (hence initialized to 0). *Atom* indicates if the two Link fields are "real" or "null"; if null, backtracking occurs. Pointer reversal is carried out as usual. An external Boolean flag is used to signal "forward" or "backtrack".

Garbage Collection

Do we reset *Mark* for a node when we leave it on backtracking? If we do, it can be dangerous, as we have no means of knowing it was once visited should a path cross it again! So this is not reset.

But this is OK, as the most common use of the DWS algorithm is for *Mark-and-Sweep* garbage collection. The DWS is used to mark all nodes accessible from some initial node, and a sweep is then used to re-claim all un-marked nodes.

Details: We have left out the *Value* fields of nodes. This field usually has two pieces of information: a beginning address for a contiguous piece of (remote) data, and its length, or alternately its end address. This allows data to be inhomogeneous.