Threaded representation of binary trees

Notice three important facts:

(a) the traversal algorithms studied so far spend most of their time manipulating a stack;

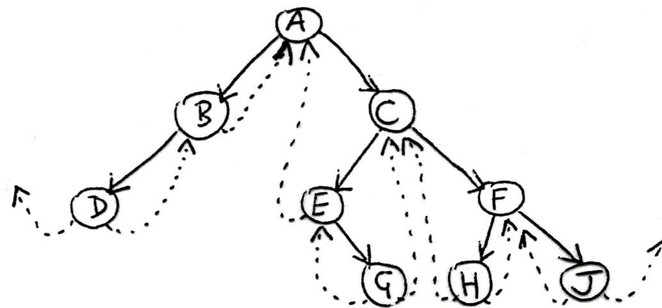
(b) the storage space required for the stack is potentially large;

(c) the majority of pointers (`LLINK` and `RLINK`) in any binary tree are `nil`.

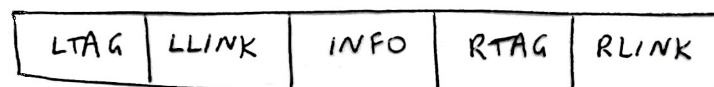
Threaded representation obviates the need for a stack, making use of pointer fields which would otherwise have the value `nil`. The most common convention for threaded representation is:

> if the left subtree is empty, `LLINK` points to the *in-order predecessor*
>
> if the right subtree is empty, `RLINK` points to the *in-order successor.*

These special pointers are known as *threads.* Here is an example of a threaded binary tree:



But any program examining the tree must be able to distinguish between a branch and a thread. So we introduce two additional fields into each node giving us, for threaded trees, nodes of the following form:

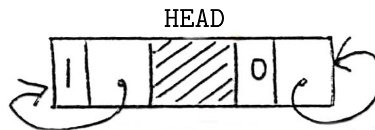


`LTAG` and `RTAG` obviously need be only one-bit fields. Because of word-length considerations, these fields will often not make any difference to the amount of storage used per node; the convention adopted, for instance, might be to use negated pointer values to indicate threads. The convention used in these notes is:

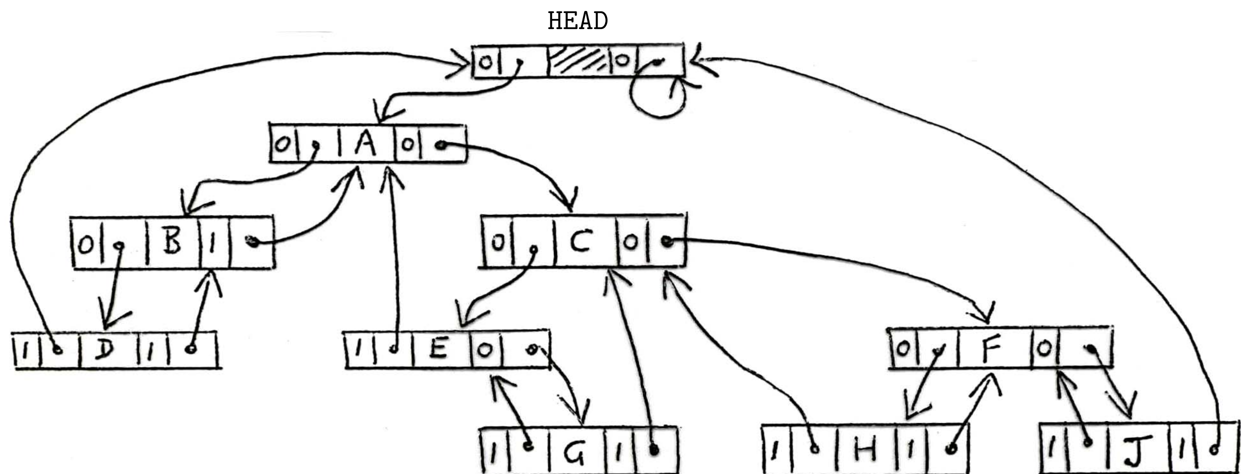
if `LTAG` $= 0$, `LLINK` points to the left child;
if `LTAG` $= 1$, `LLINK` points to the in-order predecessor;
if `RTAG` $= 0$, `RLINK` points to the right child;
if `RTAG` $= 1$, `RLINK` points to the in-order successor.

But this is only one of a number of ways of threading a tree. For example, a common method is *right-threading*, where each node has four fields (`LLINK`, `INFO`, `RTAG`, `RLINK`). `LLINK` is `nil` if the left subtree is empty, and `RTAG` and `RLINK` are as above.
(Many algorithms only require right-threading.)

In the representation of a threaded binary tree, it is convenient to use a special node HEAD — the “list” head — which is always present, even for an empty tree, when its value is:



Conventionally, `HEAD.RLINK = HEAD` and `HEAD.RTAG = 0` for any threaded binary tree. The tree shown earlier would therefore be represented as:



Without having to use a stack, it is a simple matter to traverse a threaded binary tree in in-order. Further, given a pointer P to any node in the tree, we can find its in-order successor directly as follows:

```
Pointer insucc(Pointer P)
{   if (P↑RTAG = 1)
        return P↑RLINK;
    else
    {   P ← P↑RLINK;
        while (P↑LTAG = 0)
            P ← P↑LLINK;
        return P;
    }
}
```

Similarly, a function to produce the pre-order successor:

```
Pointer presucc(Pointer P)
{   if (P↑LTAG = 0)
        return P↑LLINK;
    else
    {   while (P↑RTAG = 1)
            P ← P↑RLINK;
        return P↑RLINK;
    }
}
```

Using repeated calls to either of these functions, we can start traversal from any node. Starting from HEAD, a complete in-order traversal is:

```
R ← insucc(&HEAD);
while (R ≠ &HEAD)
{   VISIT(R);
    R ← insucc(R);
}
```

and similarly for pre-order. Both pre-order and in-order are thus achieved more efficiently with a threaded binary tree than with a normal binary tree and a stack. In contrast, *post-order* traversal is complex and inefficient if a stack is not used. See Knuth (page 561, exercise 19).

Insertion of nodes

The requirement for the insertion of a node into a binary tree may be stated thus:

either (right-insertion)

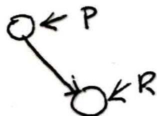
> "if *P has an empty right subtree, attach node *Q as the right subtree of *P; otherwise insert *Q between *P and *(P↑RLINK),"

or (left-insertion)

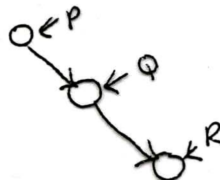
> "if *P has an empty left subtree, attach node *Q as the left subtree of *P; otherwise insert *Q between *P and *(P↑LLINK)."

Inserting *Q between *P and *R will be interpreted as the first of the following alternatives rather than the second:

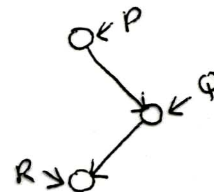
Initial state



this



not this



The insertion operations are almost trivial for an *unthreaded* binary tree. For a threaded tree they turn out to be reasonably straightforward. The following is for right-insertion:

```
Q↑RLINK ← P↑RLINK;
Q↑RTAG ← P↑RTAG;
Q↑LLINK ← P;
Q↑LTAG ← 1;
P↑RLINK ← Q;
P↑RTAG ← 0;
if (Q↑RTAG = 0)
    insucc(Q)↑LLINK ← Q;
```

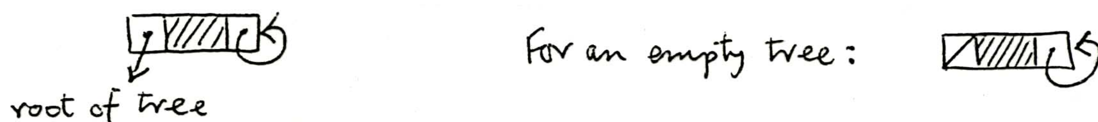
Note that in many machine representations RLINK/RTAG can be set by a single machine instruction, making this an efficient process provided the in-order successor function insucc(Q) is not called; and this function will never be called when Q is inserted as a *leaf* (which it usually is during initial set-up). A similar algorithm can be defined for left-insertion.

Successor functions applied to an unthreaded binary tree

For a threaded binary tree, it is possible to find the pre-order or in-order successor of any given node without performing a complete tree traversal. For an *unthreaded* tree, the functions `presuccU(P)` and `insuccU(P)` *must* rely on previous traversal history in a stack (unless they are to be wildly inefficient), and clearly the parameter `P` is redundant since what we must ask for is the successor of *the node we got last time*. This leads to the conclusion that, for unthreaded trees, the successor functions are really calls to *co-routines* (or *co-functions*) — with resumption of the calling program substituted for `VISIT`. However, it can be convenient to think of these successor functions as subroutines (or functions) and this can be realised by:

(a) providing every binary tree with a list-head, rather than a tree pointer

(b) adopting the convention that, for an unthreaded tree, the head node is of the form:



(c) calling the function first to find the successor of the head (which, in both cases, will give the first node in the appropriate traversal order)

(d) ensuring that the calling program recognises the end of the traversal by being given the value `HEAD` by the function. (If it does not recognise this, another traversal will begin when the function is next called.)

The algorithms that follow make use of an auxiliary stack `A`. If we wish to suspend traversal of one tree while undertaking a *complete* traversal of another, we may use stack `A` in the nested traversal by making some small changes to the function; for this reason the initial state of `A` is undefined. On the first call, the parameter `INIT` is a pointer to the head of the tree; thereafter its value is `nil`, since we are simply asking for the successor of the node we got last time we called the function. Since it is necessary that the contents of the stack `A` and the pointer `P` to the last node "visited" are preserved from one call of the function to the next, they are defined as *static* variables.

```
Pointer presuccU(Pointer INIT)
// pre-order successor for an unthreaded binary tree
{   static Pointer P;
    static Pointer_stack A;
    if (INIT ≠ nil) P ← INIT;
    if (P↑LLINK ≠ nil)
    {   STACK(A,P);
        P ← P↑LLINK;
    }
    else
    {   while (P↑RLINK = nil)
            P ← UNSTACK(A);
        P ← P↑RLINK;
    }
    return P;
}
```

```
Pointer insuccU(Pointer INIT)
// in-order successor for an unthreaded binary tree
{   static Pointer P;
    static Pointer_stack A;
    if (INIT ≠ nil) P ← INIT;
    if (P↑RLINK = nil)
        P ← UNSTACK(A);
    else
    {   P ← P↑RLINK;
        while (P↑LLINK ≠ nil)
        {   STACK(A,P);
            P ← P↑LLINK;
        }
    }
    return P;
}
```

To erase a binary tree

In these algorithms, the nodes of the tree are returned one by one to the storage pool. The reasons for selecting the *in-order* traversal sequence for this purpose should be clear:

```
// For an unthreaded tree
Q ← insuccU(&HEAD);
while (Q ≠ &HEAD)
{   R ← insuccU(nil);
    RELEASE(Q);
    Q ← R;
}
HEAD.LLINK ← nil;
```

```
// For a threaded tree
Q ← insucc(&HEAD);
while (Q ≠ &HEAD)
{   R ← insucc(Q);
    RELEASE(Q);
    Q ← R;
}
HEAD.LLINK ← &HEAD;
HEAD.LTAG ← 1;
```

To thread an unthreaded binary tree

Assume that the layout of each node is that appropriate to a threaded tree, i.e., the `LTAG` and `RTAG` fields are present in all nodes including `HEAD`, but that in the initial (unthreaded) version, these fields are not used, and a node with an empty left or right subtree has `LLINK = nil` or `RLINK = nil` (except for `HEAD.RLINK`).

```
void ThreadTree(Node HEAD)
{   Pointer Q, R;

    Q ← &HEAD:
    R ← insuccU(&HEAD);
    InsertThread(Q, R);
    while (R ≠ &HEAD)
    {   Q ← R;
        R ← insuccU(nil);
        InsertThread(Q, R);
    }
}

void InsertThread(Pointer q, Pointer r)
{   if (q↑RLINK = nil)
    {   q↑RLINK ← r;
        q↑RTAG ← 1;
        r↑LTAG ← 0;
    }
    else // (r↑LLINK = nil)
    {   r↑LLINK ← q;
        r↑LTAG ← 1;
        q↑RTAG ← 0;
    }
}
```