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 could be proportional to the size of the tree;
(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:

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

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, another 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 \texttt{HEAD} — the “list” (or tree) head — which is always present, even in an empty tree. By convention, \( \texttt{HEAD}^{\uparrow} \texttt{RLINK} = \texttt{HEAD} \) and \( \texttt{HEAD}^{\uparrow} \texttt{RTAG} = 0 \) for any threaded binary tree. For a non-empty tree, \( \texttt{HEAD}^{\uparrow} \texttt{LLINK} \) points to the root and \( \texttt{HEAD}^{\uparrow} \texttt{LTAG} = 0 \); for an empty tree, \( \texttt{HEAD}^{\uparrow} \texttt{LLINK} = \texttt{HEAD} \) and \( \texttt{HEAD}^{\uparrow} \texttt{LTAG} = 1 \).

The tree shown earlier would therefore be represented as:

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

\begin{verbatim}
Treenode insucc(Treenode P)
{
    if (P^{\uparrow}RTAG = 1)
        return P^{\uparrow}RLINK;
    else
        { P ← P^{\uparrow}RLINK;
          while (P^{\uparrow}LTAG = 0)
            P ← P^{\uparrow}LLINK;
          return P;
        }
}
\end{verbatim}

Similarly, the following is a function that returns a pointer to the pre-order successor:

\begin{verbatim}
Treenode presucc(Treenode P)
{
    if (P^{\uparrow}LTAG = 0)
        return P^{\uparrow}LLINK;
    else
        { while (P^{\uparrow}RTAG = 1)
          P ← P^{\uparrow}RLINK;
        return P^{\uparrow}RLINK;
        }
}
\end{verbatim}

Note that both of these functions only make use of the right threads.
Using repeated calls to either of these functions, we can start traversal from any node. Starting from \textbf{HEAD}, a complete in-order traversal is produced by:

\begin{verbatim}
P ← insucc(HEAD);
while (P ≠ HEAD)
{
    VISIT(P);
    P ← insucc(P);
}
\end{verbatim}

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 node \( P \) 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 node \( P \) has an empty left subtree, attach node \( Q \) as the left subtree of \( P \); otherwise insert \( Q \) between \( P \) and \( P↑LLINK \).”

For right-insertion, inserting \( Q \) between \( P \) and \( R \) will be interpreted as the first of the following alternatives rather than the second:

\begin{center}
\begin{tabular}{c|c|c}
<table>
<thead>
<tr>
<th>Initial state</th>
<th>this</th>
<th>not this</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
\end{tabular}
\end{center}

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

\begin{verbatim}
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;
\end{verbatim}

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

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, successor functions must rely on previous traversal history having been saved in a stack, unless they are to be wildly inefficient. Moreover, a successor function could not return the successor of an arbitrary node referred to by a parameter P, but would just return the successor of the node that was returned last time. This leads to the conclusion that, for unthreaded trees, successor functions are really calls to co-routines (or co-functions) — with resumption of the calling program substituted for VISIT — and each co-routine/co-function would need its own stack.

Erasing a binary tree

To erase a binary tree, we need to return (release) the nodes of the tree to the storage pool one by one. Clearly this can be done in post-order; however, for an unthreaded tree, it can also be done in the other two orders if we take care to save the necessary information before releasing the node. The following algorithms show how to do this in pre-order and in-order, both (a) recursively and (b) non-recursively using a stack.

```c
void pre-erase(Treenode P)
    if (P ≠ nil)
        { Q ← P↑LLINK;
          T ← P↑RLINK;
          RELEASE(P);
          pre-erase(Q);
          pre-erase(T);
        }
void pre-erase(Treenode P)
    { Treenode-stack A;
      CLEAR(A);
      while (true)
        if (P ≠ nil)
            { Q ← P↑LLINK;
              STACK(A, P↑RLINK);
              RELEASE(P);
              P ← Q;
            }
        else
            if (EMPTY(A))
                return;
            else
                P ← UNSTACK(A);
    }
```
void in-erase(Treenode P)
if (P \neq \text{nil})
{   in-erase(P\uparrow\text{LLINK});
    Q \leftarrow P\uparrow\text{RLINK};
    RELEASE(P);
    in-erase(Q);
}

void in-erase(Treenode P)
{   Treenode-stack A;
    CLEAR(A);
    while (true)
        if (P \neq \text{nil})
            {   STACK(A, P);
                P \leftarrow P\uparrow\text{LLINK};
            }
        else
            if (\text{EMPTY}(A))
                return;
            else
                {   P \leftarrow \text{UNSTACK}(A);
                    Q \leftarrow P\uparrow\text{RLINK};
                    RELEASE(P);
                    P \leftarrow Q;
                }
    }

For a threaded binary tree, the traversal must be in in-order:

P \leftarrow \text{insucc}(	ext{HEAD});
while (P \neq \text{HEAD})
{   Q \leftarrow \text{insucc}(P);
    RELEASE(P);
    P \leftarrow Q;
}

\text{HEAD}\uparrow\text{LLINK} \leftarrow \text{HEAD};
\text{HEAD}\uparrow\text{LTAG} \leftarrow 1;