Inorder-threaded (inthreaded) binary tree

In the basic binary tree, if a node does not have a left-child (right-child), then its left-pointer (right-pointer) is set to nullptr.

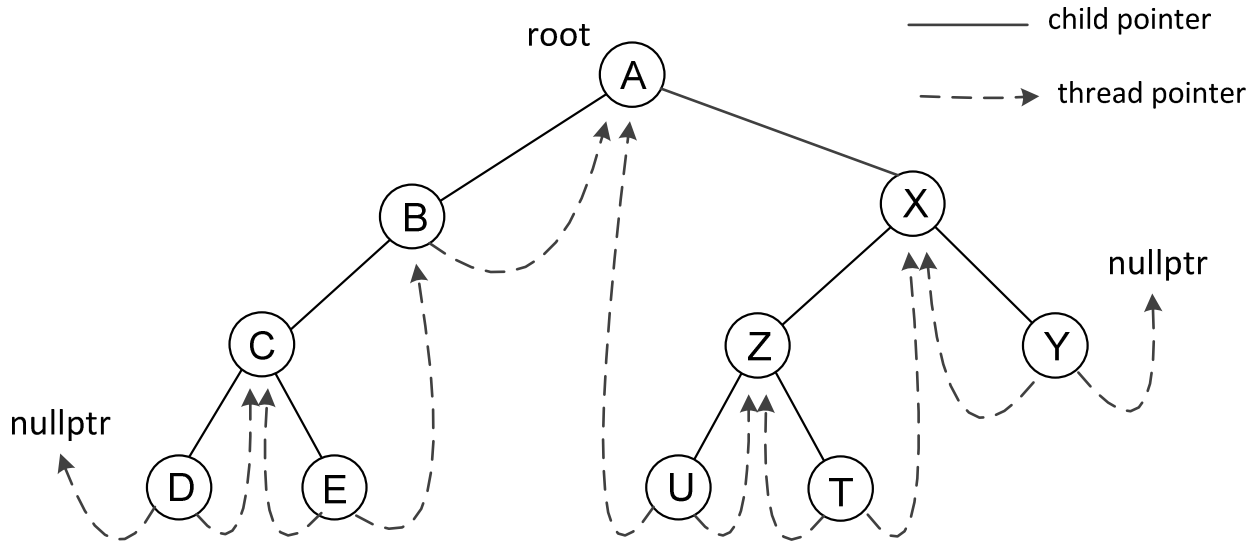
In an inthreaded binary tree, if a node does not have a left-child, its left-pointer is set to point to its inorder predecessor.

If a node does not have a right-child, its right-pointer is set to point to its inorder successor.

```
struct threadedNode
{
   int info;
   bool lthread, rthread; // left-thread, right-thread
   threadedNode *left, *right;
}
// If lthread == false, left points to left-child.
// If lthread == true, left points to inorder predecessor.

// If rthread == false, right points to right-child.
// If rthread == true, right points to inorder successor.
```

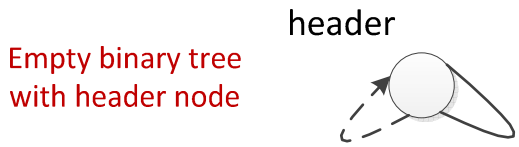
Example inthreaded binary tree



To avoid having 2 dangling pointers with null value, a header node is added to serve as a sentinel.

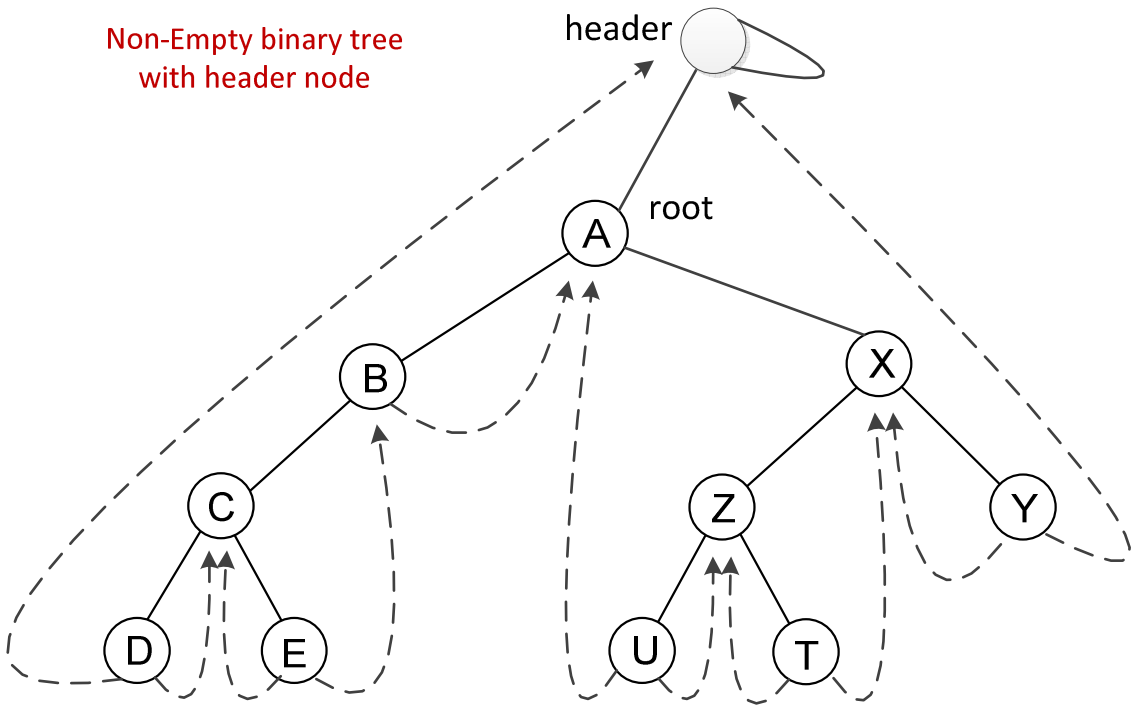
If the tree is empty

- right pointer of the header is treated as right-child, and it points back to the header
- left pointer of the header is treated as a thread, and it points back to the header



If the tree is not empty

- left pointer (treated as left-child) of the header points to the root of the binary tree
- left pointer of the first node in the inorder sequence is a thread that points to the header
- right pointer of the last node in the inorder sequence is a thread that points to the header



With an inthreaded binary tree, we can traverse the tree in inorder without using recursion, nor stack.

```
threadedNode* insucc(threadedNode *p)
{
   // return the inorder successor of p

   threadedNode *succ = p->right;
   // If p->rthread is true, succ is pointing to the
   // inorder successor, no further processing is required.

   if (p->rthread == false)
   // inorder successor is the left most descendant
   // in right subtree
   {
      while (succ->lthread == false)
         succ = succ->left;
   }
   return succ;
}
// Remark: for an empty binary tree
//         insucc(header) = header


void inorder(threadedNode* header)
{
   threadedNode *p = insucc(header);

   while (p != header)
   {
      cout << p->info << endl;  // visit node p
      p = insucc(p);
   }
}
```

We can perform a reversed inorder traversal with an inthreaded binary tree

```
threadedNode* inpred(threadedNode *p)
{
   // return the inorder predecessor of p

   threadedNode *pred = p->left;
   // If p->lthread is true, pred is pointing to the
   // inorder predecessor.

   if (p->lthread == false)
   // inorder predecessor is the right most descendant
   // in left subtree
   {
      while (pred->rthread == false)
         pred = pred->right;
   }
   return pred;
}

// if the tree is empty, inpred(header) = header

// if the tree is not empty, inpred(header) = last node
// of the inorder sequence
```

We can also traverse an inthreaded binary tree in preorder without using recursion, nor stack.

Preorder successor of node p:

- If p has a left child, then the left child is the preorder successor of p.
- If p does not have a left child, but it has a right child, then the right child is the preorder successor of p.
- If p is a leaf node, and it is the last node in the preorder traversal of the left subtree of some node q. Follows the right-thread until reaching a node q with a right child. The right child of q is the preorder successor of p.

```
threadedNode* presucc(threadedNode *p)
{
   // return the preorder successor of p

   threadedNode *succ;

   if (p->lthread == false)
      succ = p->left;
   else if (p->rthread == false)
      succ = p->right;
   else
   {
      succ = p;
      while (succ->rthread)
         succ = succ->right;
      succ = succ->right;
   }
   return succ;
}

void preorder(threadedNode* header)
{
   threadedNode *p = presucc(header);

   while (p != header)
   {
      cout << p->info << endl;  // visit node p
      p = presucc(p);
   }
}
```

Algorithm to find the parent of a node in an inthreaded binary tree

- For any node p, there exists a nearest ancestor q, such that p is in its right subtree.
- If all else fails, the header is such ancestor node.
- We can find this ancestor node by following left pointers until we have followed a thread.
- If p is the right child of q, then q is its parent - otherwise, we follow left pointers in the right subtree of q until we hit p.

```
threadedNode* parent(threadedNode *p)
{
   // return the parent of p

   threadedNode *q = p;
   while (q->lthread == false)
         q = q->left;
   q = q->left;  // follow the left-thread

   if (p != q->right)
   {
      q = q->right;
      while (p != q->left)
         q = q->left;
   }

   return q;
}
```

The worst case time complexity for finding the parent is $O(h)$, where h is the height of the tree.

With the function to find the parent of a node, we can perform a reversed preorder traversal.

The preorder predecessor of node p:

- If p is the left child of its parent q, then q is the preorder predecessor of p.
- If p is the right child of its parent q and q does not have a left child, then q is the preorder predecessor of p.
- If p is the right child of its parent q and q has an non-empty left subtree, then the preorder predecessor of p is the last node in the preorder traversal of the left subtree of q.

```
threadedNode* prepred(threadedNode *p)
{
   // return the preorder predecessor of p

   threadedNode *q = parent(p);
   if (p == q->left || q->lthread)
      return q;

   q = q->left;
   while (q->rthread == false || q->lthread == false)
      if (q->rthread == false)
         q = q->right;
      else
         q = q->left;

   return q;
}
```

It is also possible to traverse an inthreaded binary tree in postorder without using recursion, nor stack.