Chapter 12: Threaded Binary Trees

Data Structures and Algorithms with C

n+1 NULL pointers | Threading rules | Head node | Finding the successor | Stackless traversal | Insert right / left | Morris traversal

Source: Horowitz–Sahni–Anderson-Freed 5.5

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/dsa_c/slides/Chapter_12_Threaded_Binary_Trees.html

Learning Outcomes

Count the null pointers

Prove that a binary tree with n nodes has n+1 NULL pointers.

Build threads

Apply the rule: left NULL → inorder predecessor, right NULL → inorder successor.

Tell them apart

Define the structure fully with the left_thread / right_thread flags and a head node.

Traverse without a stack

Traverse inorder (and preorder) with `insucc` using O(1) extra space.

Insert

Merge the two cases of `insert_right` into one routine; derive `insert_left`.

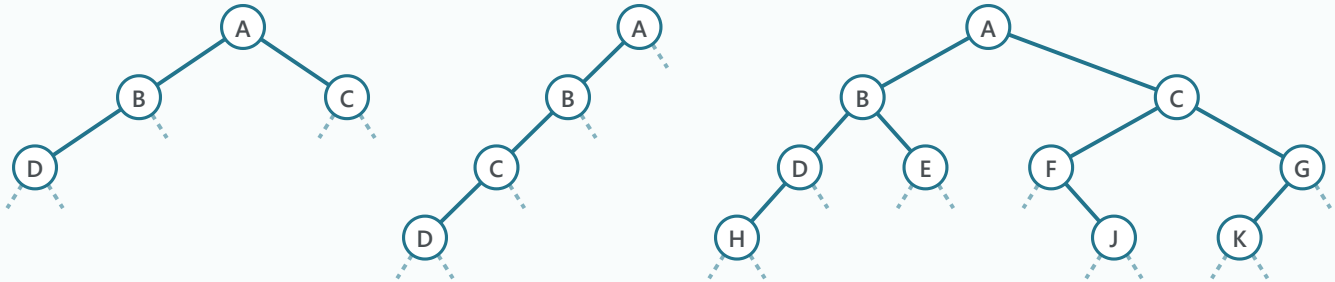
Write good code

A clearly defined routine, a meaningful comment, the right place for the NULL check.

n+1 NULL Pointers in n Nodes

- Each node has 2 pointer fields → **2n** pointers in the tree.
- Every node except the root is pointed to by its parent → **n − 1** pointers are used.
- The remaining **2n − (n−1) = n + 1** pointers are NULL: more than half the pointers are wasted!

> **"A year spent in artificial intelligence is enough to make one believe in God."**
> — Alan J. Perlis



`n = 4 NULL = 5` `n = 4 NULL = 5` `n = 10 NULL = 11`

Short dashed lines are NULL pointers; in all three examples there are n + 1 of them.

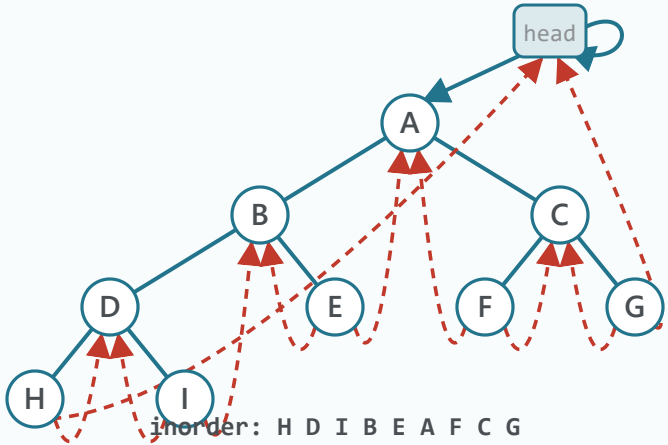
The Idea: Make the Null Pointers Point the Way

- A. J. Perlis and C. Thornton showed that NULL pointers can be used to point to **other nodes** of the tree; they called these pointers **threads**.
- The gain: an inorder traversal needs **no stack and no recursion**; from each node we go straight to its successor.
- The price: **one bit** per pointer saying whether it is a child or a thread.

Recursive traversal uses an O(h) call stack, iterative traversal an O(h) stack. A threaded tree needs **O(1)** extra space.

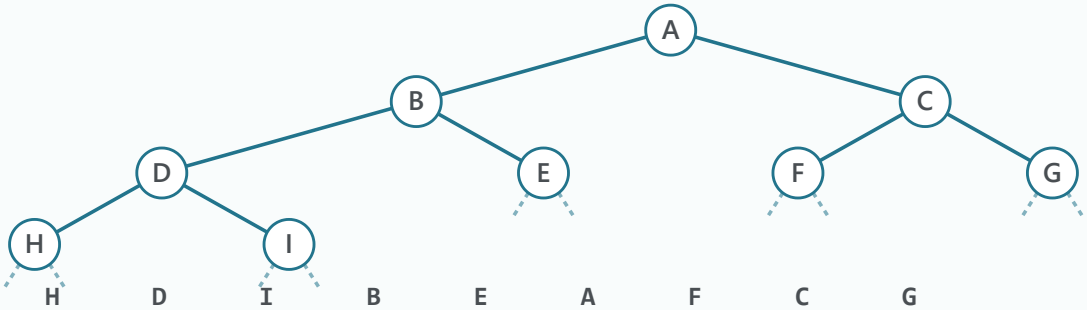
Rules for Building a Threaded Binary Tree

1. If `p->left_child == NULL`, `p->left_child` points to the node visited just **before** p in inorder: the NULL is replaced by p's **inorder predecessor**.
2. If `p->right_child == NULL`, `p->right_child` points to the node visited just **after** p: the NULL is replaced by p's **inorder successor**.



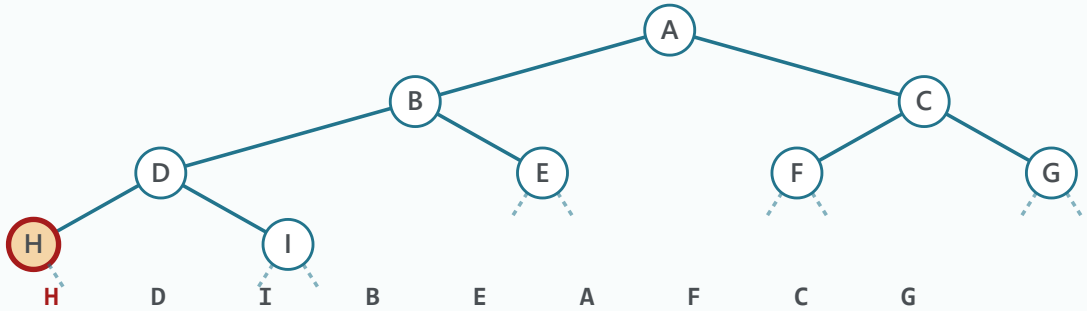
The dashed red arrows are threads. They “weave” the tree's inorder sequence into the tree itself.

Adding the Threads One by One



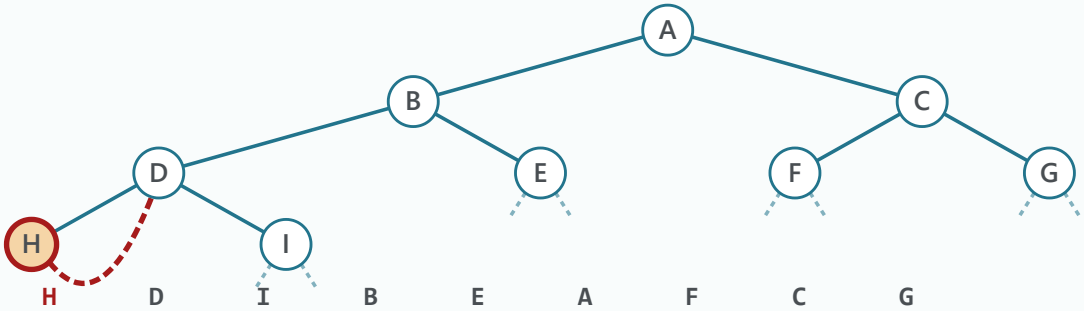
An ordinary binary tree: 9 nodes, 10 NULL pointers (dashed). Inorder: H D I B E A F C G.

Adding the Threads One by One



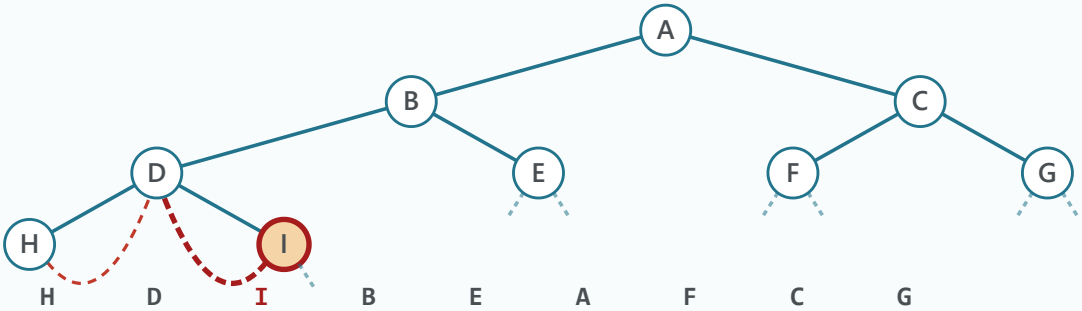
H's **left** pointer is NULL → it points to its inorder **predecessor**, but none exists: it dangles until the head node is added.

Adding the Threads One by One



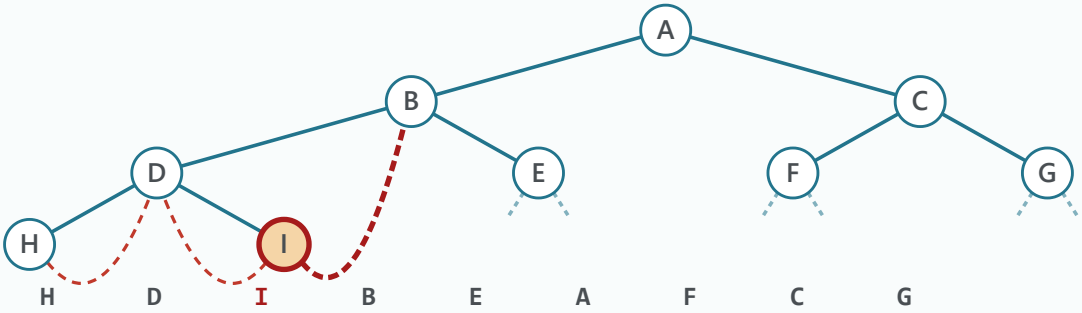
H's **right** pointer is NULL → it points to its inorder **successor**, D.

Adding the Threads One by One



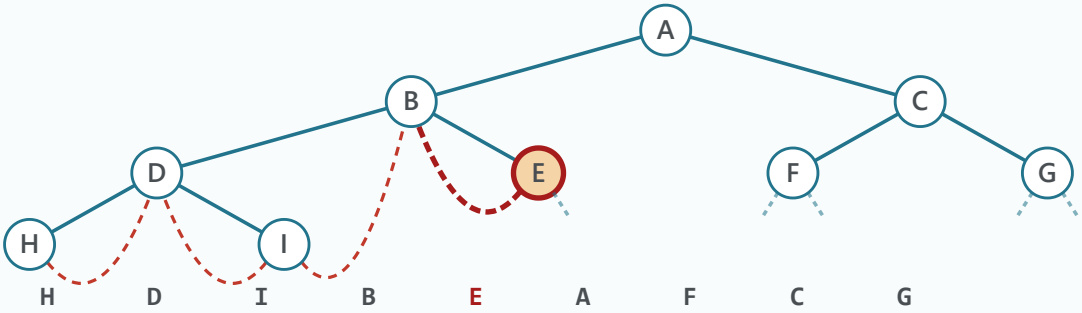
I's **left** pointer is NULL → it points to its inorder **predecessor**, D.

Adding the Threads One by One



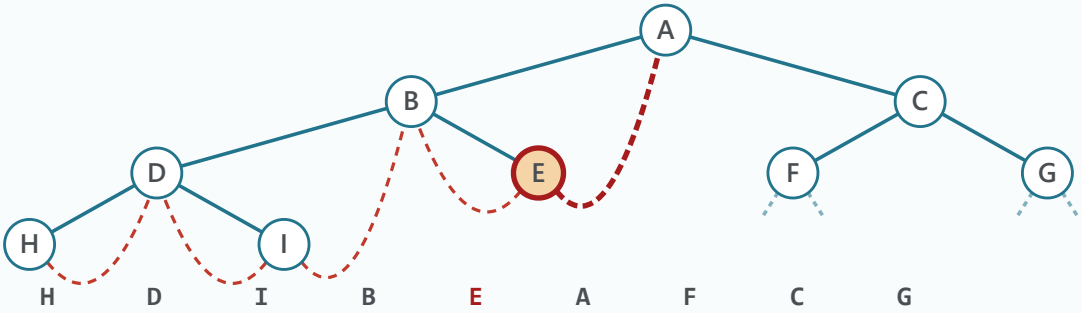
I's **right** pointer is NULL → it points to its inorder **successor**, B.

Adding the Threads One by One



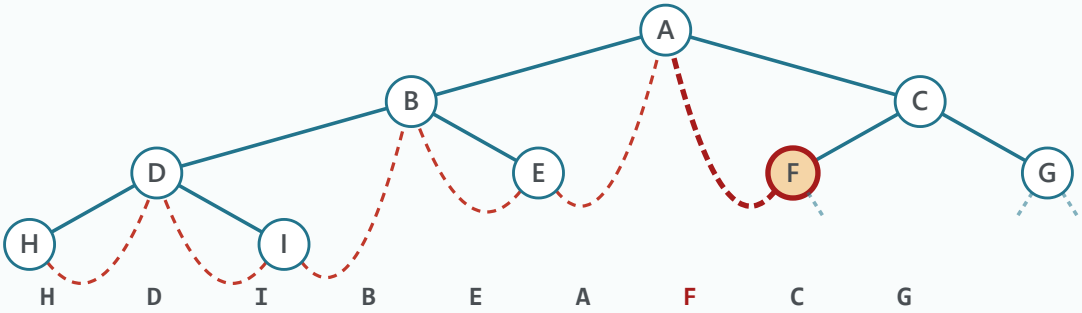
E's **left** pointer is NULL → it points to its inorder **predecessor**, B.

Adding the Threads One by One



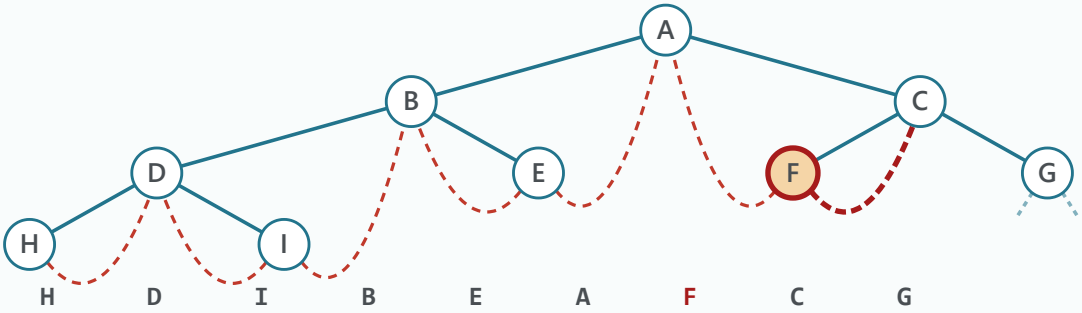
E's **right** pointer is NULL → it points to its inorder **successor**, A.

Adding the Threads One by One



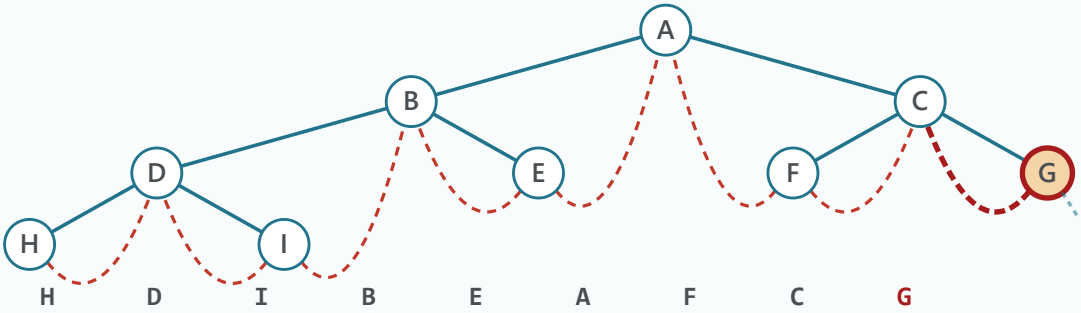
F's **left** pointer is NULL → it points to its inorder **predecessor**, A.

Adding the Threads One by One



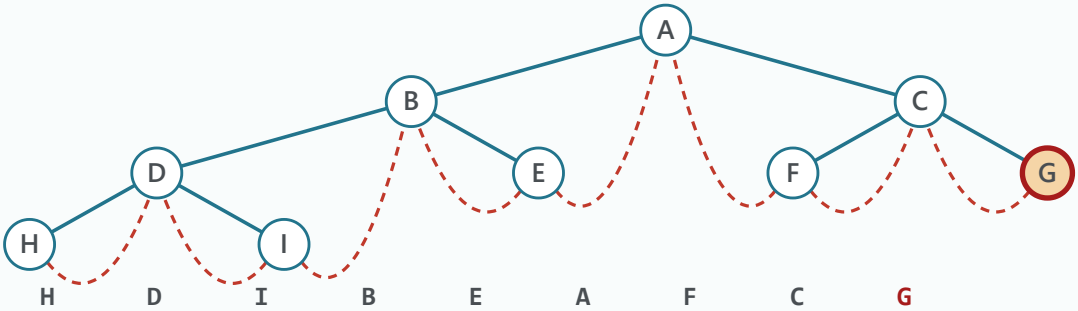
F's **right** pointer is NULL → it points to its inorder **successor**, C.

Adding the Threads One by One



G's **left** pointer is NULL → it points to its inorder **predecessor**, C.

Adding the Threads One by One



G's **right** pointer is NULL → it points to its inorder **successor**, but none exists: it dangles until the head node is added.

How Do We Tell the Compiler an Arrow Is Red?

```
typedef struct threaded_tree *thread_ptr;
struct threaded_tree {
    short int    left_thread; /* TRUE: thread */
    thread_ptr left_child;
    char         data;
    thread_ptr right_child;
    short int    right_thread; /* TRUE: thread */
};
```

A struct definition ends with the closing `;` (line 8); forgetting it is a common mistake. `TRUE/FALSE` : `enum { FALSE, TRUE }`.

INSTRUCTOR "Does the compiler have red pointers? How would we know they are red? We have to store two extra pieces of information."

INSTRUCTOR "Doesn't space complexity go up? Let it; time complexity matters more than space complexity. Besides, I could do this with just 2 bits."

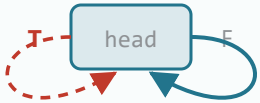
Two Dangling Threads and the Head Node

- H's left thread and G's right thread have nothing to point to: they **dangle**.
- We add a **head** node: the root of the actual tree is reached through `L→left_child`.
- The two dangling threads point to the head; the head's right pointer always points to itself, with right thread FALSE.
- In an empty tree the left pointer also points to the head, with left thread TRUE; it becomes FALSE once a node is inserted.

left_thread	left_child	data	right_child	right_thread
TRUE	→ head	–	→ head	FALSE

An empty threaded tree

The Head Node: Empty Tree and Nonempty Tree

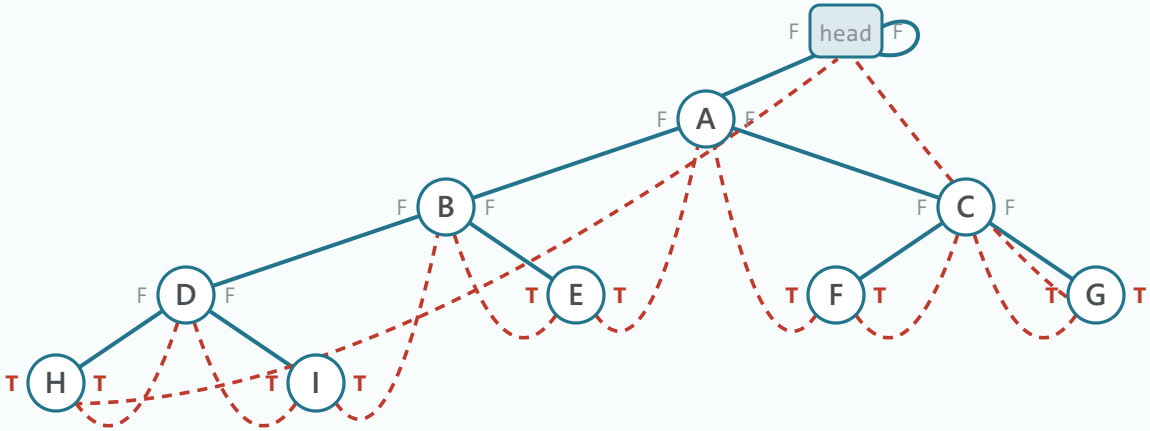


left_child → itself (thread, TRUE)

right_child → itself (FALSE)

Empty tree: only the head; both of its pointers point to itself.

The Head Node: Empty Tree and Nonempty Tree



Nonempty tree: the head's left points to the root (A); H's left thread and G's right thread are tied to the head. T/F: the thread flags.

The Tree Is Threaded for Inorder Traversal

INSTRUCTOR “We build the threaded tree for inorder traversal. You could not build it for preorder anyway.”

traversal	order
inorder	H D I B E A F C G
preorder	A B D H I E C F G
postorder	H I D E B F G C A

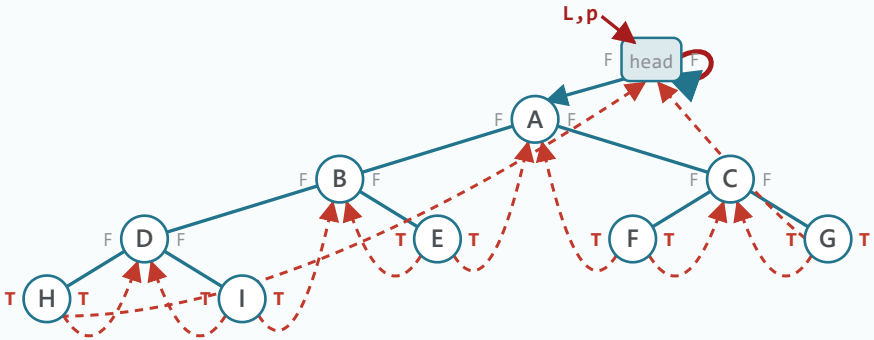
Threads follow inorder predecessors and successors; preorder and postorder traversals can also be done with them (more circuitously).

Finding the Inorder Successor

1. If `p→right_thread == TRUE`, `p→right_child` (being a thread) already is p's successor.
2. If `p→right_thread == FALSE`, start at the right child and follow `left_child` links until `left_thread` is TRUE: the **leftmost** node of the right subtree is the successor.

```
thread_ptr insucc(thread_ptr L) {
  // successor of the (non-NULL) node L
  thread_ptr p = L->right_child;
  if (!L->right_thread)         // right subtree?
    while (!p->left_thread)     // go to its leftmost
      p = p->left_child;
  return p;
}
```

insucc(): for the Head, A and I



`insucc(head)` : p = head->right_child = head.

Step 1 / 16

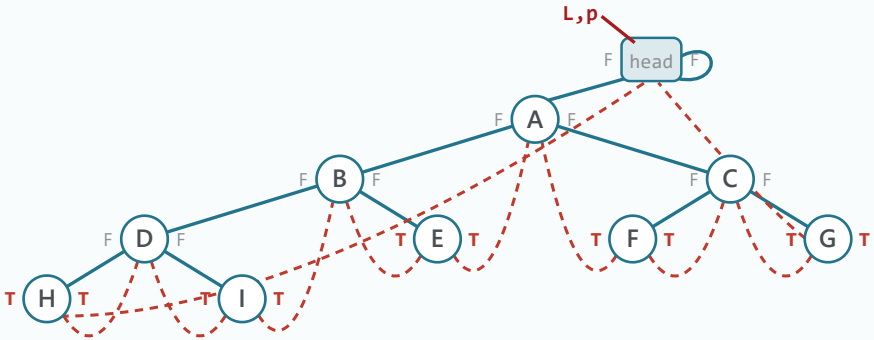
```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
▶ 3   thread_ptr p = L->right_child;
4   if (!L->right_thread)          // right subtree?
5     while (!p->left_thread)    // go to its leftmost
6       p = p->left_child;
7   return p;
8 }
```

VARIABLES

`L = &head` `p = &head`

`L→right_thread = FALSE`

insucc(): for the Head, A and I



right_thread is FALSE: the head has a real right subtree; the successor is its leftmost node.

Step 2 / 16

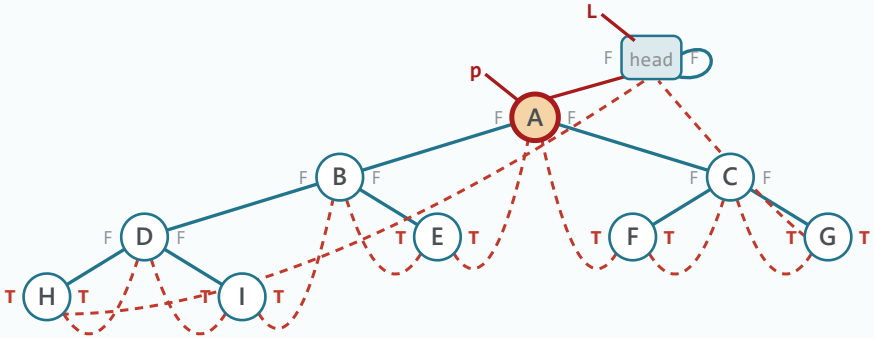
```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
3   thread_ptr p = L->right_child;
▶   if (!L->right_thread)          // right subtree?
5     while (!p->left_thread)     // go to its leftmost
6       p = p->left_child;
7   return p;
8 }
```

VARIABLES

`L = &head` `p = &head`

`L→right_thread = FALSE`

insucc(): for the Head, A and I



head->left_thread FALSE → go left: p = A.

Step 3 / 16

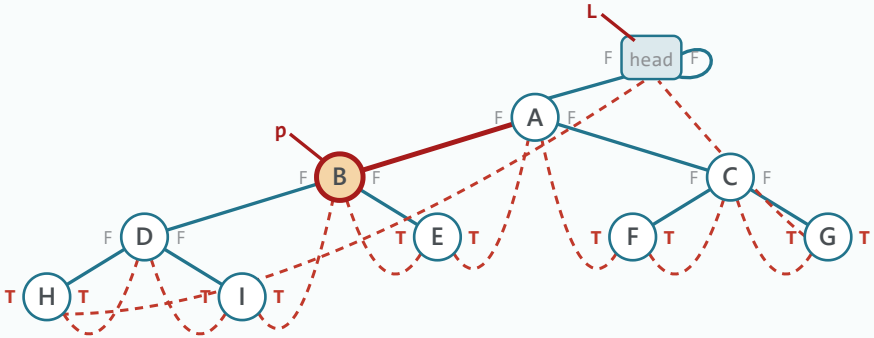
```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
3   thread_ptr p = L->right_child;
4   if (!L->right_thread)          // right subtree?
▶     while (!p->left_thread)    // go to its leftmost
▶       p = p->left_child;
7   return p;
8 }
```

VARIABLES

`L = &head` `p = &node(A)`

`L→right_thread = FALSE`

insucc(): for the Head, A and I



A->left_thread FALSE → go left: p = B.

Step 4 / 16

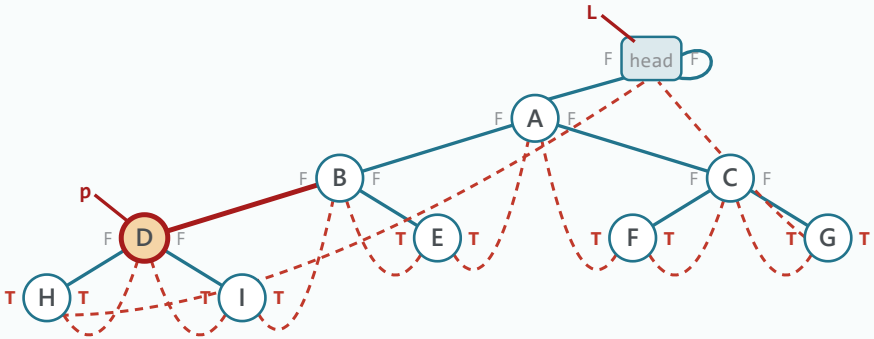
```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
3   thread_ptr p = L->right_child;
4   if (!L->right_thread)         // right subtree?
▶     while (!p->left_thread)   // go to its leftmost
▶       p = p->left_child;
7   return p;
8 }
```

VARIABLES

`L = &head` `p = &node(B)`

`L→right_thread = FALSE`

insucc(): for the Head, A and I



B->left_thread FALSE → go left: p = D.

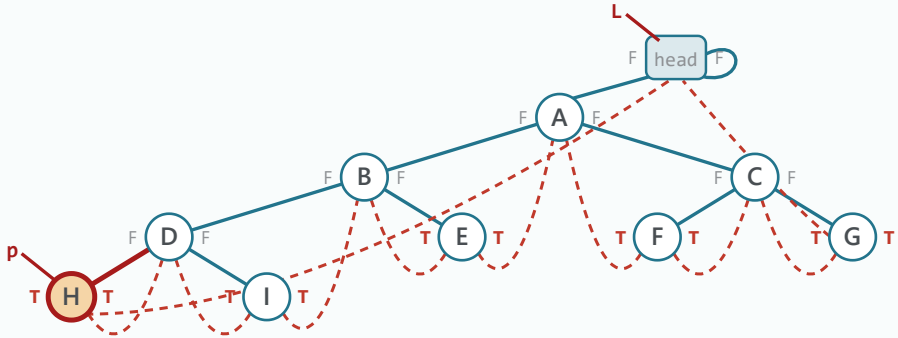
Step 5 / 16

```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
3   thread_ptr p = L->right_child;
4   if (!L->right_thread)          // right subtree?
▶     while (!p->left_thread)    // go to its leftmost
▶       p = p->left_child;
7   return p;
8 }
```

VARIABLES

`L = &head` `p = &node(D)` `L→right_thread = FALSE`

insucc(): for the Head, A and I



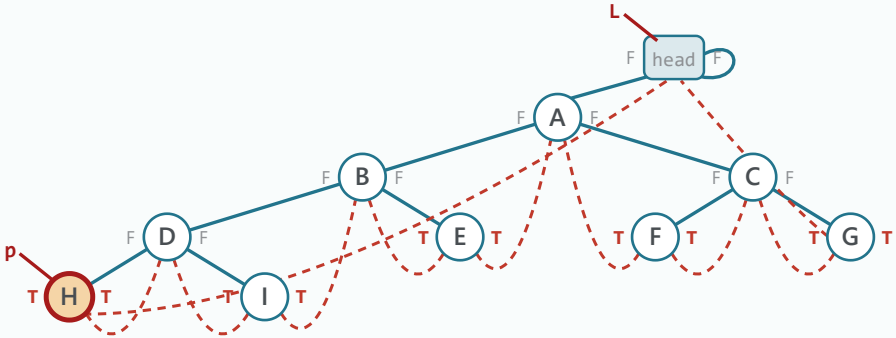
D->left_thread FALSE → go left: p = H.


```
1  thread_ptr insucc(thread_ptr L) {
2    // successor of the (non-NULL) node L
3    thread_ptr p = L->right_child;
4    if (!L->right_thread)          // right subtree?
▶      while (!p->left_thread)    // go to its leftmost
▶        p = p->left_child;
7    return p;
8  }
```

VARIABLES

`L = &head` `p = &node(H)` `L→right_thread = FALSE`

insucc(): for the Head, A and I



H->left_thread TRUE: cannot go further left → the leftmost node.

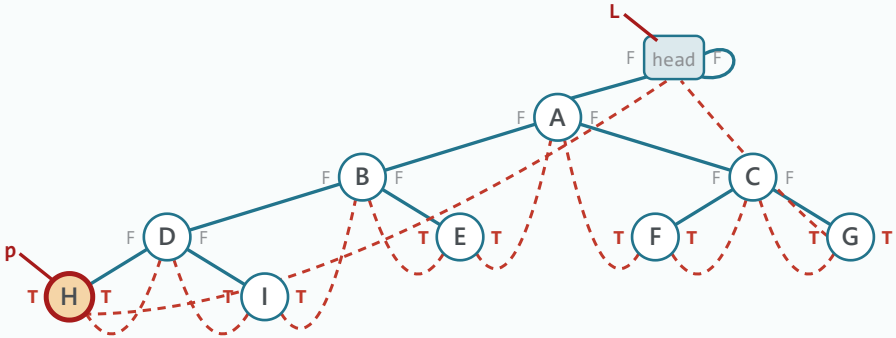
Step 7 / 16

```
1  thread_ptr insucc(thread_ptr L) {
2    // successor of the (non-NULL) node L
3    thread_ptr p = L->right_child;
4    if (!L->right_thread)          // right subtree?
▶      while (!p->left_thread)    // go to its leftmost
6        p = p->left_child;
7    return p;
8  }
```

VARIABLES

`L = &head` `p = &node(H)` `L→right_thread = FALSE`

insucc(): for the Head, A and I



Returns **H**: the head's inorder successor. (The head's successor = the first node visited.)

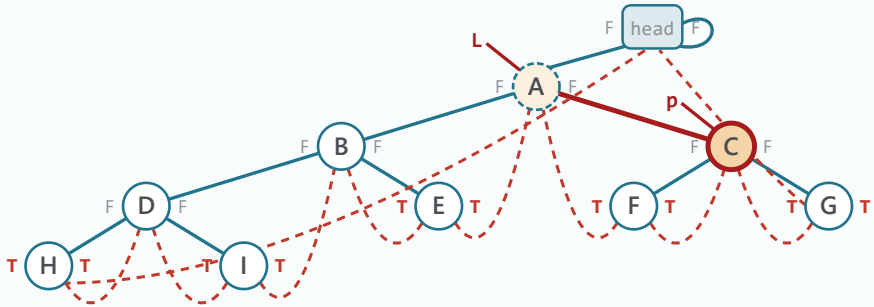
Step 8 / 16

```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
3   thread_ptr p = L->right_child;
4   if (!L->right_thread)          // right subtree?
5     while (!p->left_thread)    // go to its leftmost
6       p = p->left_child;
▶   return p;
8 }
```

VARIABLES

`L = &head` `p = &node(H)` `L→right_thread = FALSE`

insucc(): for the Head, A and I



`insucc(A)` : p = A->right_child = C.

Step 9 / 16

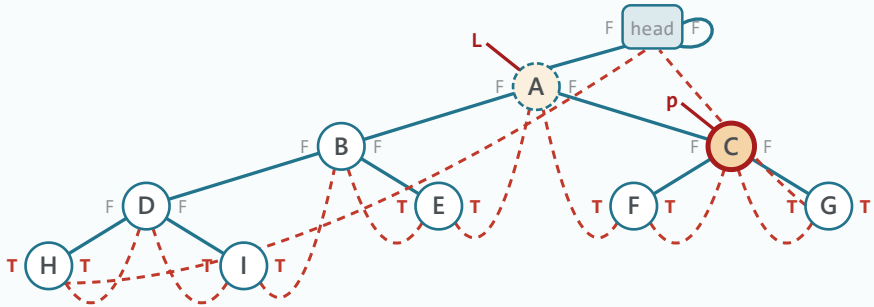
```
1  thread_ptr insucc(thread_ptr L) {
2    // successor of the (non-NULL) node L
▶    thread_ptr p = L->right_child;
4    if (!L->right_thread)         // right subtree?
5      while (!p->left_thread)    // go to its leftmost
6        p = p->left_child;
7    return p;
8  }
```

VARIABLES

`L = &node(A)` `p = &node(C)`

`L→right_thread = FALSE`

insucc(): for the Head, A and I



right_thread is FALSE: A has a real right subtree; the successor is its leftmost node.

Step 10 / 16

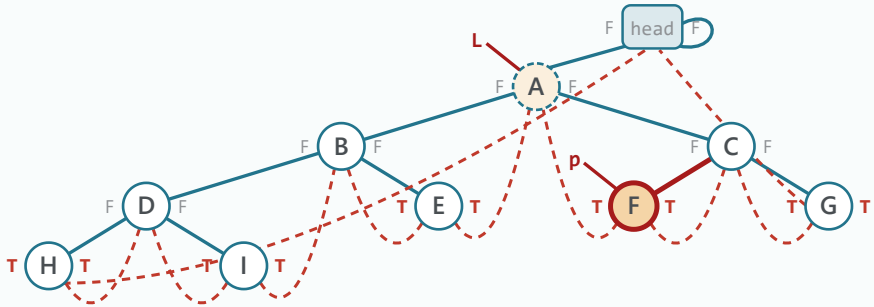
```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
3   thread_ptr p = L->right_child;
▶   if (!L->right_thread)         // right subtree?
5     while (!p->left_thread)    // go to its leftmost
6       p = p->left_child;
7   return p;
8 }
```

VARIABLES

`L = &node(A)` `p = &node(C)`

`L→right_thread = FALSE`

insucc(): for the Head, A and I



C->left_thread FALSE → go left: p = F.

Step 11 / 16

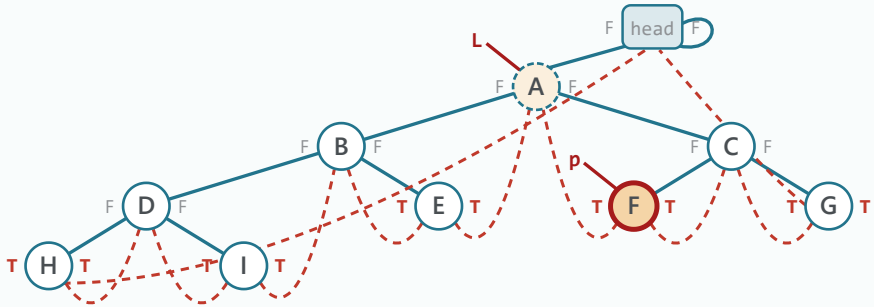
```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
3   thread_ptr p = L->right_child;
4   if (!L->right_thread)        // right subtree?
▶     while (!p->left_thread)   // go to its leftmost
▶       p = p->left_child;
7   return p;
8 }
```

VARIABLES

`L = &node(A)` `p = &node(F)`

`L→right_thread = FALSE`

insucc(): for the Head, A and I



F->left_thread TRUE: cannot go further left → the leftmost node.

Step 12 / 16

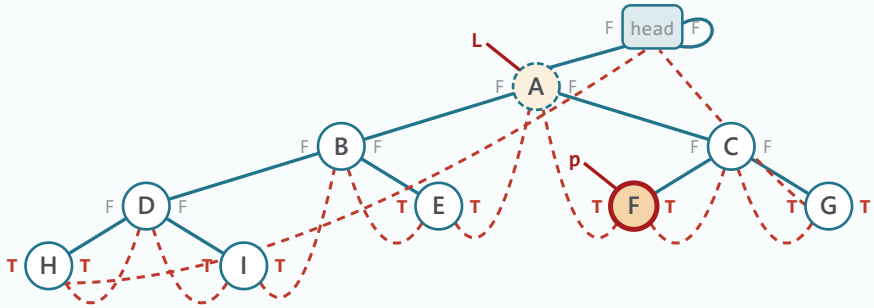
```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
3   thread_ptr p = L->right_child;
4   if (!L->right_thread)        // right subtree?
▶     while (!p->left_thread)   // go to its leftmost
6       p = p->left_child;
7   return p;
8 }
```

VARIABLES

L = &node(A) p = &node(F)

L→right_thread = FALSE

insucc(): for the Head, A and I



Returns **F**: A's inorder successor.

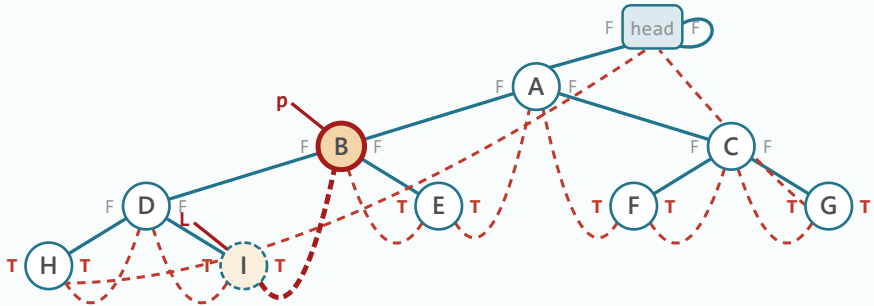
Step 13 / 16

```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
3   thread_ptr p = L->right_child;
4   if (!L->right_thread)         // right subtree?
5     while (!p->left_thread)    // go to its leftmost
6       p = p->left_child;
▶   return p;
8 }
```

VARIABLES

`L = &node(A)` `p = &node(F)` `L→right_thread = FALSE`

insucc(): for the Head, A and I



`insucc(I)` : p = I->right_child = B.

Step 14 / 16

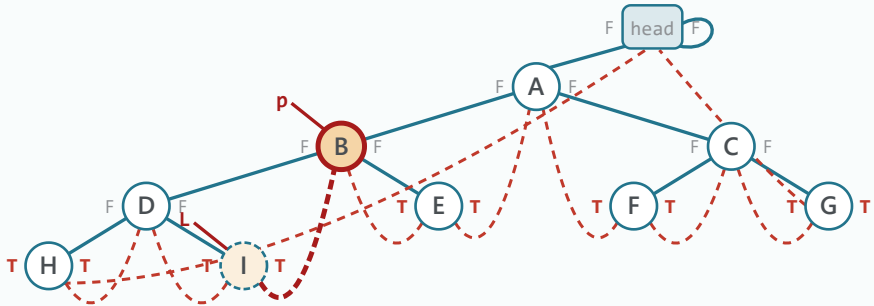
```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
▶   thread_ptr p = L->right_child;
4   if (!L->right_thread)         // right subtree?
5     while (!p->left_thread)    // go to its leftmost
6       p = p->left_child;
7   return p;
8 }
```

VARIABLES

`L = &node(I)` `p = &node(B)`

`L->right_thread = TRUE`

insucc(): for the Head, A and I



I's right pointer is a **thread** (TRUE): the successor is B directly. The loop is skipped.

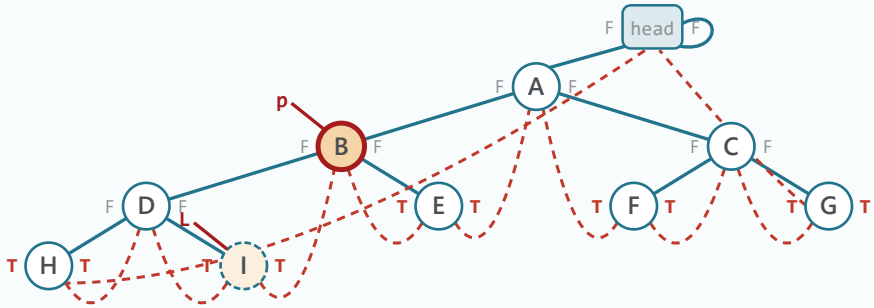
Step 15 / 16

```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
3   thread_ptr p = L->right_child;
▶   if (!L->right_thread)         // right subtree?
5     while (!p->left_thread)    // go to its leftmost
6       p = p->left_child;
7   return p;
8 }
```

VARIABLES

`L = &node(I)` `p = &node(B)` `L→right_thread = TRUE`

insucc(): for the Head, A and I



Returns **B**: I's inorder successor.

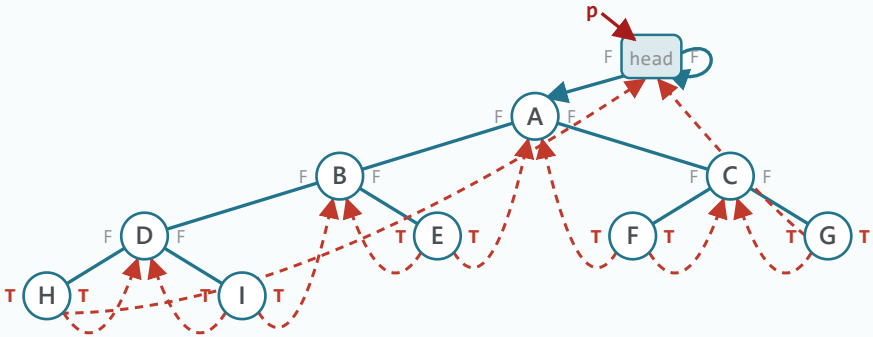
Step 16 / 16

```
1 thread_ptr insucc(thread_ptr L) {
2   // successor of the (non-NULL) node L
3   thread_ptr p = L->right_child;
4   if (!L->right_thread)         // right subtree?
5     while (!p->left_thread)    // go to its leftmost
6       p = p->left_child;
  ▶ return p;
8 }
```

VARIABLES

`L = &node(I)` `p = &node(B)` `L→right_thread = TRUE`

tinorder(): – H D I B E A F C G –



p = L: start at the head. No stack, no recursion; one pointer is enough.

Step 1 / 11

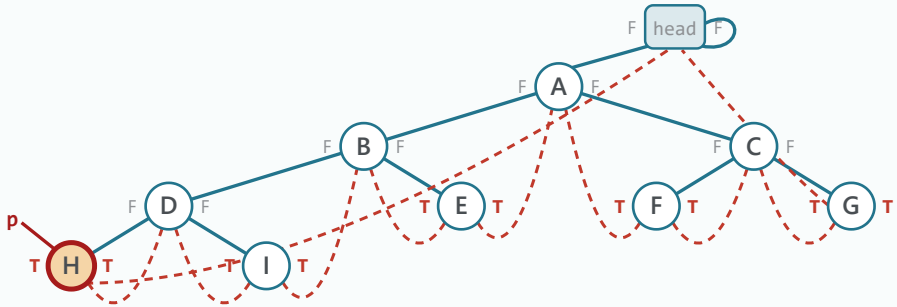
```
1 void tinorder(thread_ptr L) {
▶   thread_ptr p = L;        // L: the head node
3   for (;;) {
4     p = insucc(p);
5     if (p == L) break;       // back at the head?
6     printf("%c ", p->data);
7   }
8 }
```

VARIABLES

L = &head p = &head

OUTPUT

tinorder(): – H D I B E A F C G –



p = succ(head) = **H** (leftmost of the right subtree); printed.

Step 2 / 11

```
1 void tinorder(thread_ptr L) {
2   thread_ptr p = L;        // L: the head node
3   for (;;) {
▶     p = insucc(p);
5     if (p == L) break;       // back at the head?
▶     printf("%c ", p->data);
7   }
8 }
```

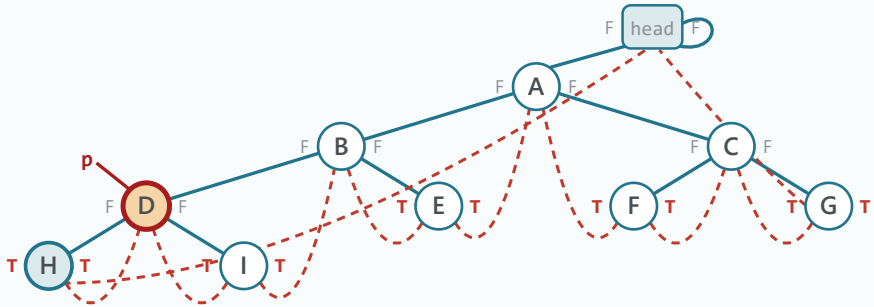
VARIABLES

`L = &head` `p = &node(H)`

OUTPUT

```
H
```

tinorder(): – H D I B E A F C G –



p = succ(H) = **D** (directly via the thread); printed.

Step 3 / 11

```
1 void tinorder(thread_ptr L) {
2   thread_ptr p = L;         // L: the head node
3   for (;;) {
▶     p = insucc(p);
5     if (p == L) break;        // back at the head?
▶     printf("%c ", p->data);
7   }
8 }
```

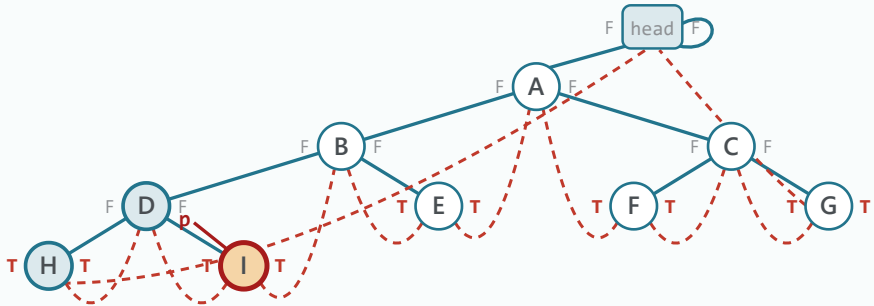
VARIABLES

`L = &head` `p = &node(D)`

OUTPUT

```
H D
```

tinorder(): – H D I B E A F C G –



p = succ(D) = **I** (leftmost of the right subtree); printed.

Step 4 / 11

```
1 void tinorder(thread_ptr L) {
2   thread_ptr p = L;         // L: the head node
3   for (;;) {
▶     p = insucc(p);
5     if (p == L) break;        // back at the head?
▶     printf("%c ", p->data);
7   }
8 }
```

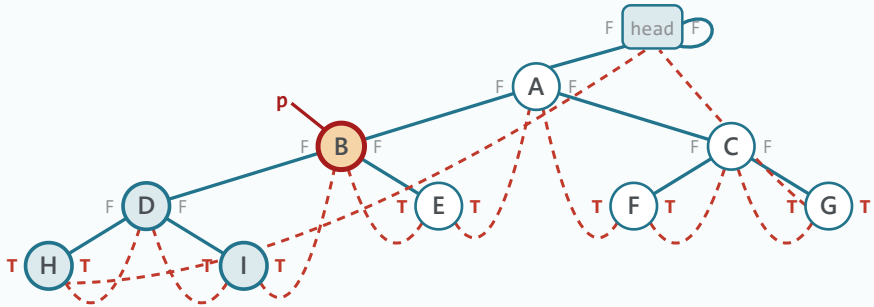
VARIABLES

`L = &head` `p = &node(I)`

OUTPUT

```
H D I
```

tinorder(): – H D I B E A F C G –



p = succ(I) = **B** (directly via the thread); printed.

Step 5 / 11

```
1 void tinorder(thread_ptr L) {
2   thread_ptr p = L;         // L: the head node
3   for (;;) {
▶     p = insucc(p);
5     if (p == L) break;        // back at the head?
▶     printf("%c ", p->data);
7   }
8 }
```

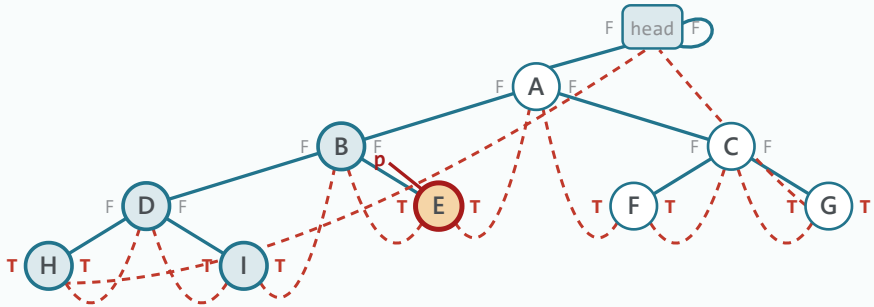
VARIABLES

`L = &head` `p = &node(B)`

OUTPUT

```
H D I B
```

tinorder(): – H D I B E A F C G –



p = succ(B) = **E** (leftmost of the right subtree); printed.

Step 6 / 11

```
1 void tinorder(thread_ptr L) {
2   thread_ptr p = L;         // L: the head node
3   for (;;) {
▶     p = insucc(p);
5     if (p == L) break;        // back at the head?
▶     printf("%c ", p->data);
7   }
8 }
```

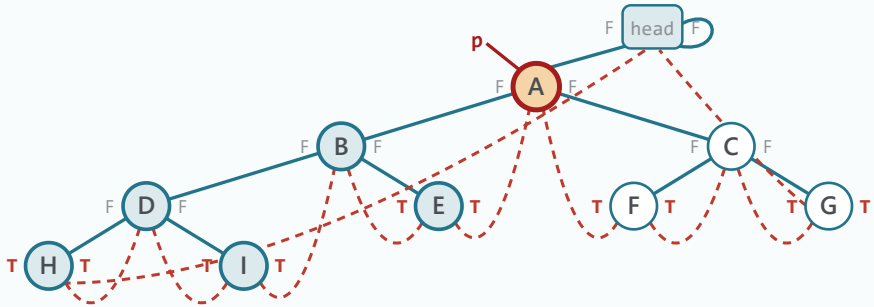
VARIABLES

`L = &head` `p = &node(E)`

OUTPUT

```
H D I B E
```

tinorder(): – H D I B E A F C G –



p = succ(E) = **A** (directly via the thread); printed.

Step 7 / 11

```
1 void tinorder(thread_ptr L) {
2   thread_ptr p = L;         // L: the head node
3   for (;;) {
▶     p = insucc(p);
5     if (p == L) break;        // back at the head?
▶     printf("%c ", p->data);
7   }
8 }
```

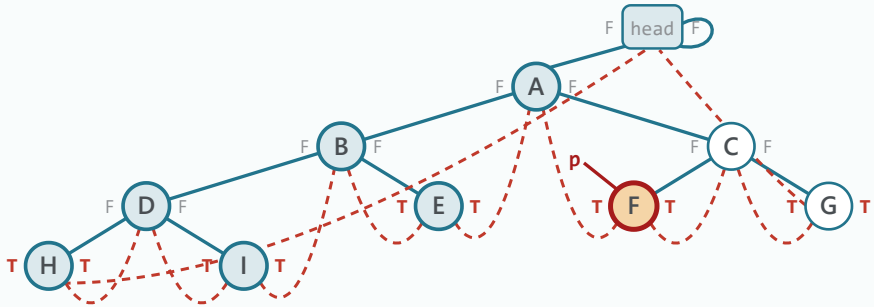
VARIABLES

`L = &head` `p = &node(A)`

OUTPUT

```
H D I B E A
```

tinorder(): – H D I B E A F C G –



p = succ(A) = **F** (leftmost of the right subtree); printed.

Step 8 / 11

```
1 void tinorder(thread_ptr L) {
2   thread_ptr p = L;         // L: the head node
3   for (;;) {
▶     p = insucc(p);
5     if (p == L) break;        // back at the head?
▶     printf("%c ", p->data);
7   }
8 }
```

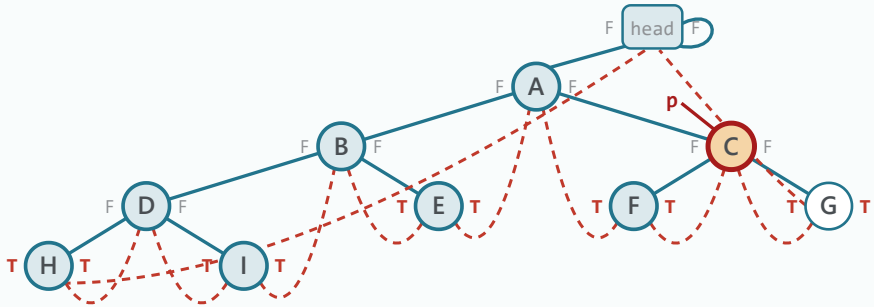
VARIABLES

`L = &head` `p = &node(F)`

OUTPUT

```
H D I B E A F
```

tinorder(): – H D I B E A F C G –



p = succ(F) = **C** (directly via the thread); printed.

Step 9 / 11

```
1 void tinorder(thread_ptr L) {
2   thread_ptr p = L;         // L: the head node
3   for (;;) {
▶     p = insucc(p);
5     if (p == L) break;        // back at the head?
▶     printf("%c ", p->data);
7   }
8 }
```

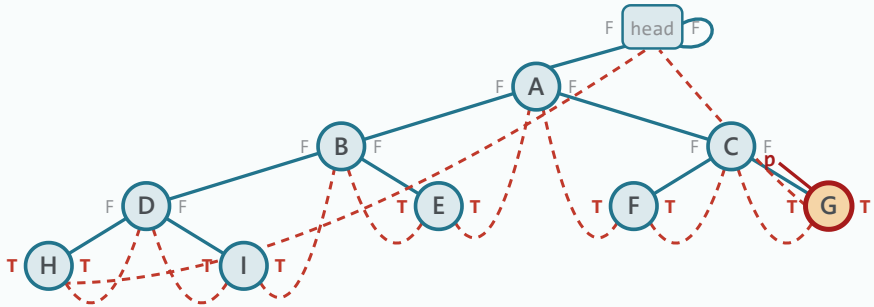
VARIABLES

L = &head p = &node(C)

OUTPUT

```
H D I B E A F C
```

tinorder(): – H D I B E A F C G –



p = succ(C) = **G** (leftmost of the right subtree); printed.

Step 10 / 11

```
1 void tinorder(thread_ptr L) {
2   thread_ptr p = L;         // L: the head node
3   for (;;) {
▶     p = insucc(p);
5     if (p == L) break;        // back at the head?
▶     printf("%c ", p->data);
7   }
8 }
```

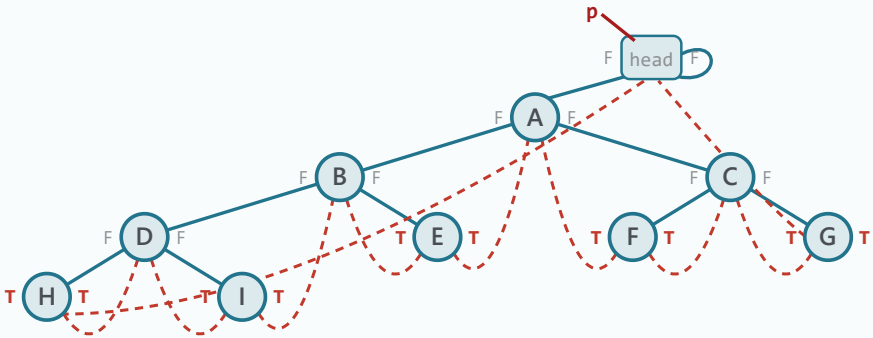
VARIABLES

`L = &head` `p = &node(G)`

OUTPUT

```
H D I B E A F C G
```

tinorder(): – H D I B E A F C G –



G's successor (directly via the thread) is the head: the traversal is done → `break`.

Step 11 / 11

```
1 void tinorder(thread_ptr L) {
2   thread_ptr p = L;        // L: the head node
3   for (;;) {
▶     p = insucc(p);
▶     if (p == L) break;       // back at the head?
6     printf("%c ", p->data);
7   }
8 }
```

VARIABLES

`L = &head` `p = &head`

OUTPUT

```
H D I B E A F C G
```

The Cost of Threaded Traversal

- Each thread and each real left/right link is followed at most once → **O(n)** time.
- Extra space **O(1)**: just the pointer p (recursive traversal needs an O(h) call stack, iterative an O(h) stack).
- From any node we can move to its successor without restarting the traversal.

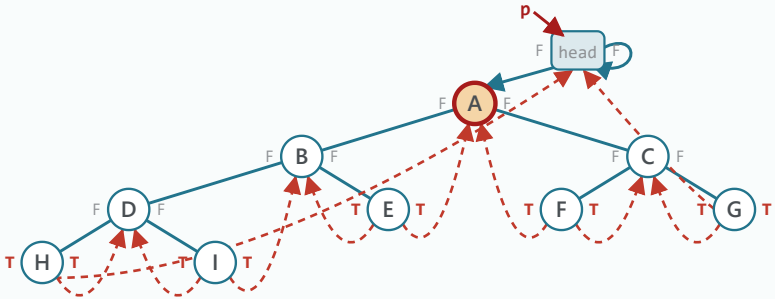
INSTRUCTOR "These two gentlemen say: 'In this tree, threaded for inorder, I can traverse iteratively in both postorder and preorder.' Preorder is easy to write; postorder is harder."

INSTRUCTOR "You read the papers on a problem; it has been solved in $O(n^2)$. Sitting down and writing an algorithm that brings it down to $O(n^{1.89})$ is a big deal! That algorithm goes into the literature."

Line numbers

To strip the line numbers from code listings, hold ALT in the editor, select the column and press DEL.

Preorder Traversal with the Same Threads



succ(head) = **A** (left child).

Step 1 / 10

```
1 thread_ptr presucc(thread_ptr p) {
▶   if (!p->left_thread)        // left child if any
▶     return p->left_child;
4   while (p->right_thread)      // climb the threads
5     p = p->right_child;
6   return p->right_child;        // first real child
7 }
```

VARIABLES

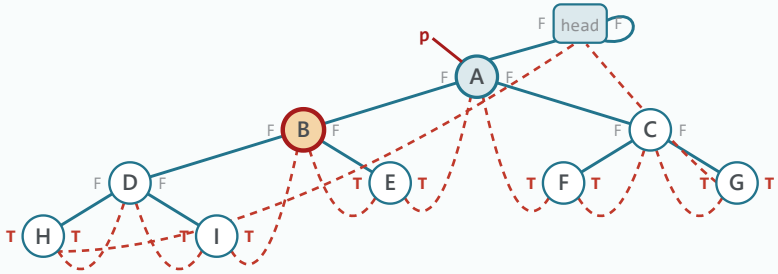
`p = &head` `result = &node(A)`

OUTPUT

```
A
```

If there is a left child, it is the successor; otherwise climb the right threads and step to the first real right child. Postorder needs parent information (harder).

Preorder Traversal with the Same Threads



succ(A) = **B** (left child).

Step 2 / 10

```
1 thread_ptr presucc(thread_ptr p) {
▶   if (!p->left_thread)        // left child if any
▶     return p->left_child;
4   while (p->right_thread)      // climb the threads
5     p = p->right_child;
6   return p->right_child;        // first real child
7 }
```

VARIABLES

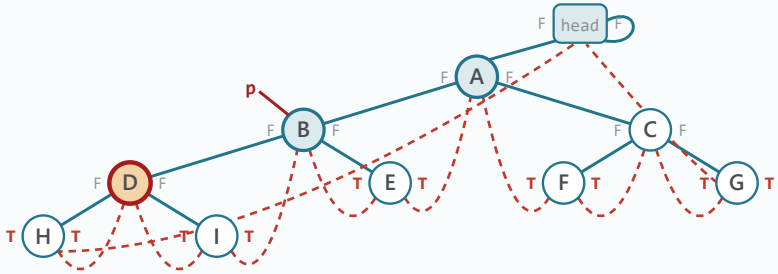
`p = &node(A)` `result = &node(B)`

OUTPUT

```
A B
```

If there is a left child, it is the successor; otherwise climb the right threads and step to the first real right child. Postorder needs parent information (harder).

Preorder Traversal with the Same Threads



succ(B) = **D** (left child).

Step 3 / 10

```
1 thread_ptr presucc(thread_ptr p) {
▶   if (!p->left_thread)        // left child if any
▶     return p->left_child;
4   while (p->right_thread)      // climb the threads
5     p = p->right_child;
6   return p->right_child;        // first real child
7 }
```

VARIABLES

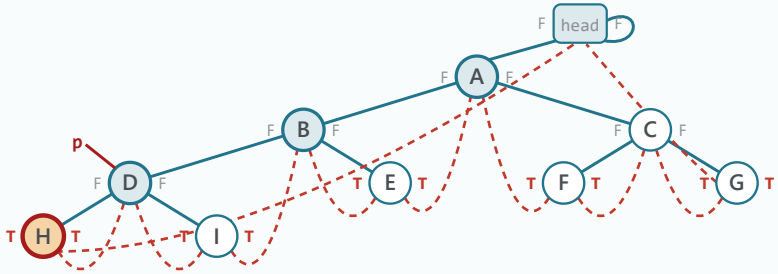
`p = &node(B)` `result = &node(D)`

OUTPUT

```
A B D
```

If there is a left child, it is the successor; otherwise climb the right threads and step to the first real right child. Postorder needs parent information (harder).

Preorder Traversal with the Same Threads



succ(D) = **H** (left child).

Step 4 / 10

```
1 thread_ptr presucc(thread_ptr p) {
▶   if (!p->left_thread)         // left child if any
▶     return p->left_child;
4   while (p->right_thread)       // climb the threads
5     p = p->right_child;
6   return p->right_child;         // first real child
7 }
```

VARIABLES

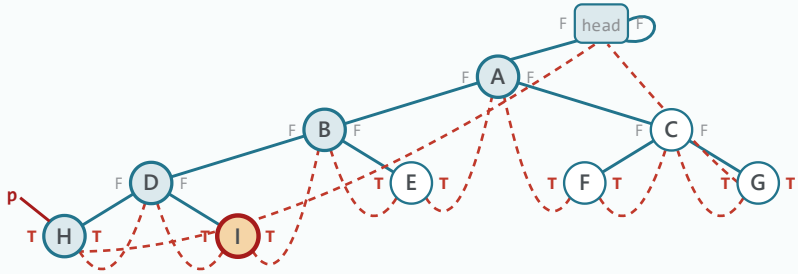
`p = &node(D)` `result = &node(H)`

OUTPUT

```
A B D H
```

If there is a left child, it is the successor; otherwise climb the right threads and step to the first real right child. Postorder needs parent information (harder).

Preorder Traversal with the Same Threads



succ(H) = **I** (right child reached via the right threads through D).

Step 5 / 10

```
1 thread_ptr presucc(thread_ptr p) {
2   if (!p->left_thread)        // left child if any
3     return p->left_child;
▶   while (p->right_thread)      // climb the threads
▶     p = p->right_child;
▶   return p->right_child;       // first real child
7 }
```

VARIABLES

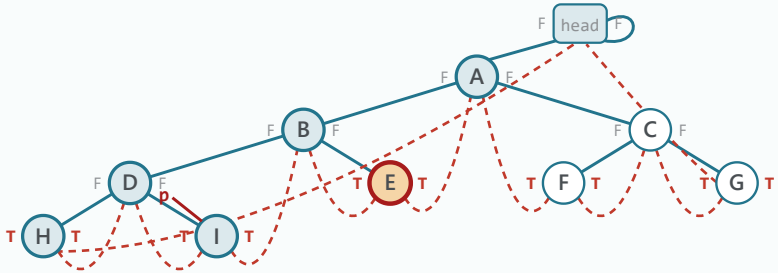
`p = &node(H)` `result = &node(I)`

OUTPUT

```
A B D H I
```

If there is a left child, it is the successor; otherwise climb the right threads and step to the first real right child. Postorder needs parent information (harder).

Preorder Traversal with the Same Threads



succ(I) = **E** (right child reached via the right threads through B).

Step 6 / 10

```
1 thread_ptr presucc(thread_ptr p) {
2   if (!p->left_thread)        // left child if any
3     return p->left_child;
▶   while (p->right_thread)      // climb the threads
▶     p = p->right_child;
▶   return p->right_child;        // first real child
7 }
```

VARIABLES

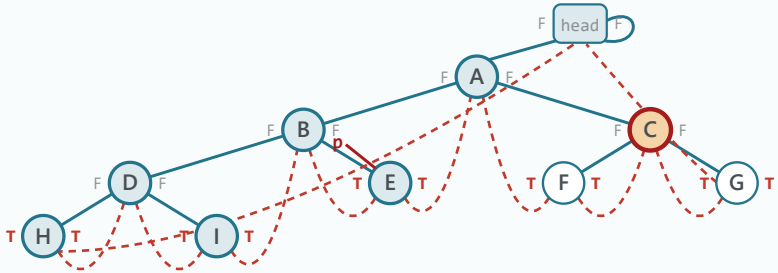
`p = &node(I)` `result = &node(E)`

OUTPUT

```
A B D H I E
```

If there is a left child, it is the successor; otherwise climb the right threads and step to the first real right child. Postorder needs parent information (harder).

Preorder Traversal with the Same Threads



succ(E) = **C** (right child reached via the right threads through A).

Step 7 / 10

```
1 thread_ptr presucc(thread_ptr p) {
2   if (!p->left_thread)        // left child if any
3     return p->left_child;
▶   while (p->right_thread)      // climb the threads
▶     p = p->right_child;
▶   return p->right_child;        // first real child
7 }
```

VARIABLES

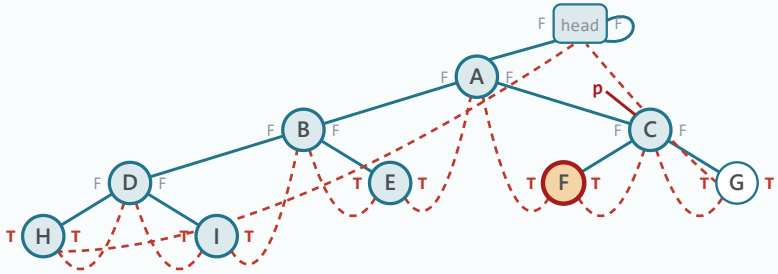
`p = &node(E)` `result = &node(C)`

OUTPUT

```
A B D H I E C
```

If there is a left child, it is the successor; otherwise climb the right threads and step to the first real right child. Postorder needs parent information (harder).

Preorder Traversal with the Same Threads



succ(C) = **F** (left child).

Step 8 / 10

```
1 thread_ptr presucc(thread_ptr p) {
▶   if (!p->left_thread)        // left child if any
▶     return p->left_child;
4   while (p->right_thread)      // climb the threads
5     p = p->right_child;
6   return p->right_child;        // first real child
7 }
```

VARIABLES

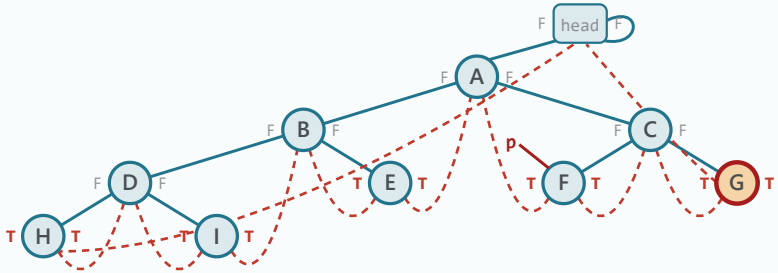
`p = &node(C)` `result = &node(F)`

OUTPUT

```
A B D H I E C F
```

If there is a left child, it is the successor; otherwise climb the right threads and step to the first real right child. Postorder needs parent information (harder).

Preorder Traversal with the Same Threads



succ(F) = **G** (right child reached via the right threads through C).

Step 9 / 10

```
1 thread_ptr presucc(thread_ptr p) {
2   if (!p->left_thread)        // left child if any
3     return p->left_child;
▶   while (p->right_thread)      // climb the threads
▶     p = p->right_child;
▶   return p->right_child;        // first real child
7 }
```

VARIABLES

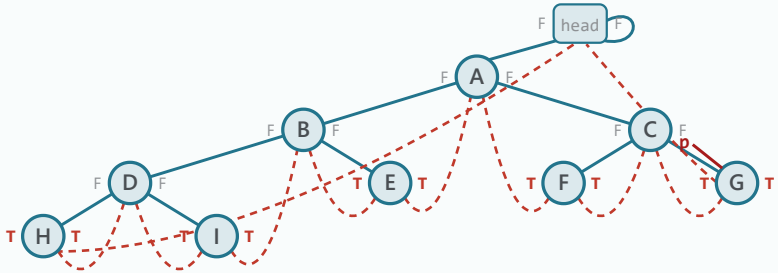
`p = &node(F)` `result = &node(G)`

OUTPUT

```
A B D H I E C F G
```

If there is a left child, it is the successor; otherwise climb the right threads and step to the first real right child. Postorder needs parent information (harder).

Preorder Traversal with the Same Threads



G: following the right threads leads to the head → head: the traversal ends.

Step 10 / 10

```
1 thread_ptr presucc(thread_ptr p) {
2   if (!p->left_thread)        // left child if any
3     return p->left_child;
▶   while (p->right_thread)      // climb the threads
▶     p = p->right_child;
▶   return p->right_child;        // first real child
7 }
```

VARIABLES

`p = &node(G)` `result = &head`

OUTPUT

```
A B D H I E C F G
```

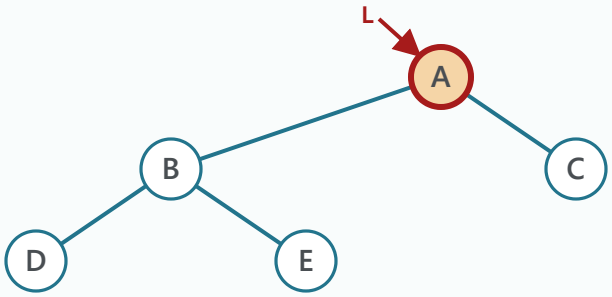
If there is a left child, it is the successor; otherwise climb the right threads and step to the first real right child. Postorder needs parent information (harder).

Morris Traversal: Temporary Threads

```
void morris_inorder(tree_ptr L) {
  while (L) {
    if (!L->left) {              /* no left: print, right */
      printf("%c", L->data);  L = L->right;
    } else {
      tree_ptr p = L->left;     /* inorder predecessor */
      while (p->right && p->right != L)
        p = p->right;
      if (!p->right) {            /* add a temp thread */
        p->right = L;  L = L->left;
      } else {                       /* remove the thread */
        p->right = NULL;
        printf("%c", L->data);  L = L->right;
      }
    }
  }
}
```

J. M. Morris (1979). No flag fields needed: threads are added and removed during the traversal; since the tree changes meanwhile, it is unsafe for concurrent readers.

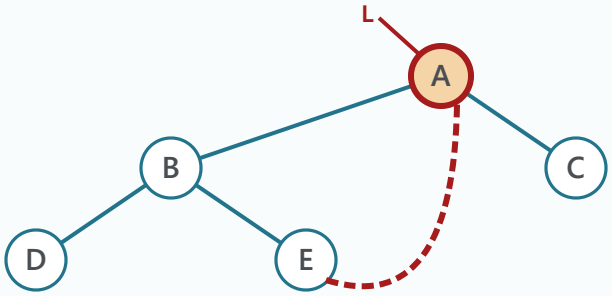
Morris Traversal Step by Step: A(B(D,E),C)



output:

Start: no stack. At each node with a left subtree, a **temporary thread** goes into its predecessor's (empty) right pointer.

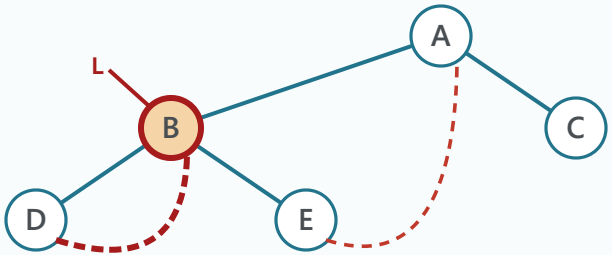
Morris Traversal Step by Step: A(B(D,E),C)



output:

A's predecessor is E; its right is empty → add thread E → A, go left.

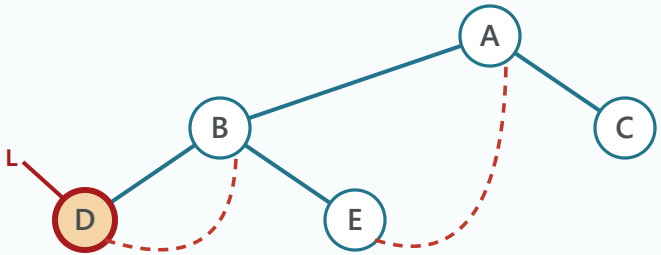
Morris Traversal Step by Step: A(B(D,E),C)



output:

B's predecessor is D; its right is empty → add thread D → B, go left.

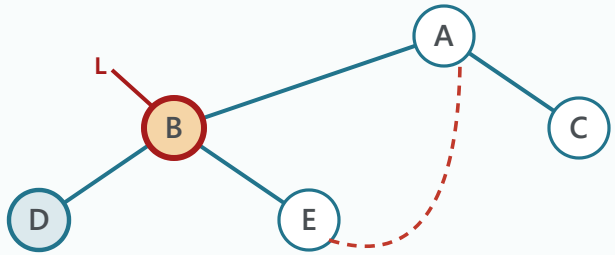
Morris Traversal Step by Step: A(B(D,E),C)



output: D

D has no left: print it, go right (or along the temporary thread) → B.

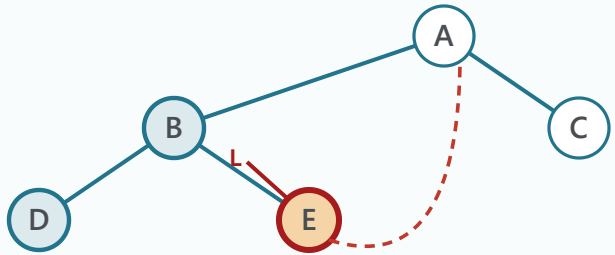
Morris Traversal Step by Step: A(B(D,E),C)



output: DB

Back at B through the thread from its predecessor D: remove the thread, print B, go right.

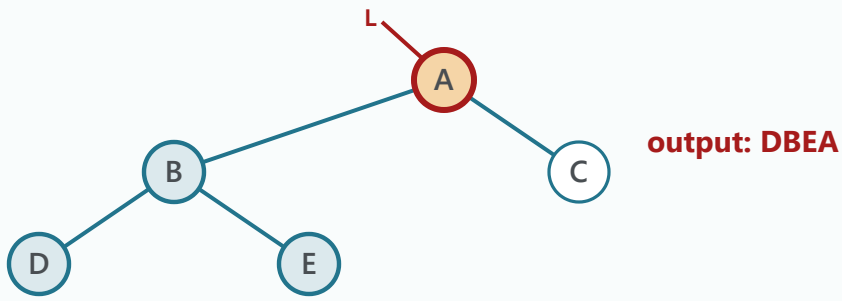
Morris Traversal Step by Step: A(B(D,E),C)



output: DBE

E has no left: print it, go right (or along the temporary thread) → A.

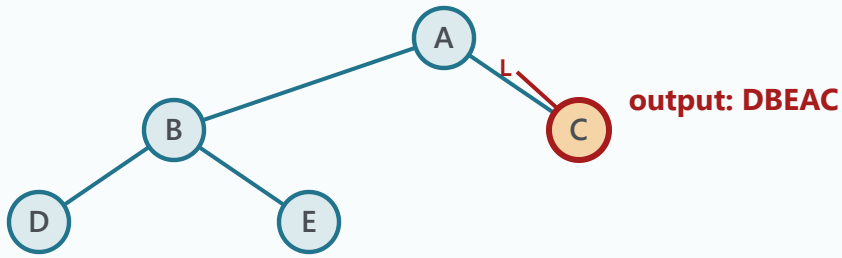
Morris Traversal Step by Step: A(B(D,E),C)



output: DBEA

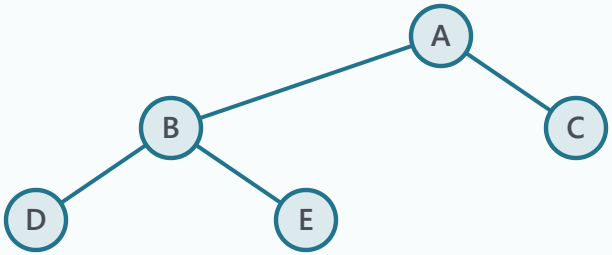
Back at A through the thread from its predecessor E: remove the thread, print A, go right.

Morris Traversal Step by Step: A(B(D,E),C)



C has no left: print it, go right (or along the temporary thread) → NULL.

Morris Traversal Step by Step: A(B(D,E),C)



output: DBEAC

Done: **DBEAC**. The tree is restored; extra space O(1), time O(n).

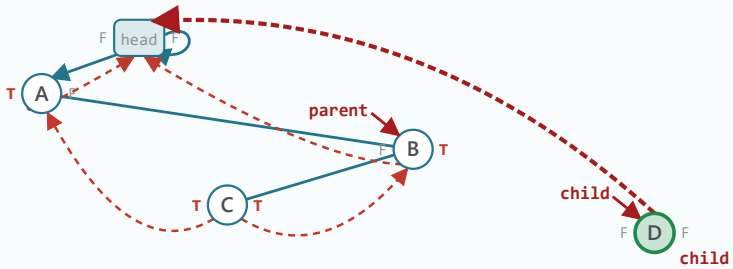
Inserting a Node into a Threaded Tree

- Insertion is not a simple routine on its own; it is **a bit tricky**.
- Inserting to the right and to the left differ → two routines: `insert_right`, `insert_left`.
- Inserting to the right has two cases: (1) the parent has **no** right subtree, (2) it **has** one.
- In a BST, insertion was a single assignment at an empty child slot; here the threads (red arrows) must also point to the right places.

```
child->right_child  = parent->right_child;
child->right_thread = TRUE;  // parent: no right
child->left_child   = parent;
child->left_thread  = TRUE;
parent->right_child   = child;
parent->right_thread = FALSE;  // was TRUE
```

Case 1: insertion in 6 assignments.

Case 1: Inserting Under a Parent with No Right Subtree



child->right_child = parent->right_child: the parent's right thread (its successor) moves to the child. (Whether it is the head does not matter.)

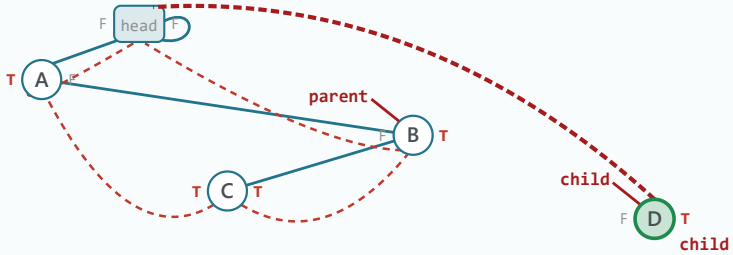
Step 1 / 6

```
▶ child->right_child  = parent->right_child;
2 child->right_thread = TRUE;  // parent: no right
3 child->left_child   = parent;
4 child->left_thread  = TRUE;
5 parent->right_child   = child;
6 parent->right_thread = FALSE;  // was TRUE
```

VARIABLES

- `parent = &node(B)`
- `child = &node(D)`
- `parent→right_thread = TRUE`
- `child→right_thread = FALSE`

Case 1: Inserting Under a Parent with No Right Subtree



child->right_thread = TRUE: the parent had no subtree, so the child cannot either → red arrow.

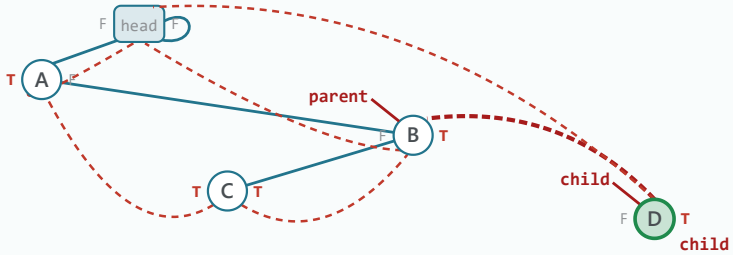
Step 2 / 6

```
1 child->right_child  = parent->right_child;
▶ child->right_thread = TRUE;  // parent: no right
3 child->left_child   = parent;
4 child->left_thread  = TRUE;
5 parent->right_child   = child;
6 parent->right_thread = FALSE;  // was TRUE
```

VARIABLES

- `parent = &node(B)`
- `child = &node(D)`
- `parent→right_thread = TRUE`
- `child→right_thread = TRUE`

Case 1: Inserting Under a Parent with No Right Subtree



child->left_child = parent: the child's inorder predecessor is the parent.

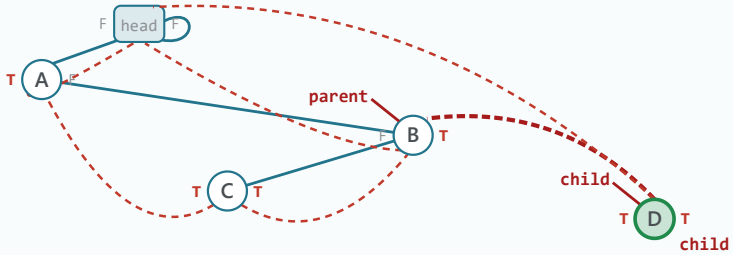
Step 3 / 6

```
1 child->right_child  = parent->right_child;
2 child->right_thread = TRUE;  // parent: no right
▶ child->left_child   = parent;
4 child->left_thread = TRUE;
5 parent->right_child   = child;
6 parent->right_thread = FALSE;  // was TRUE
```

VARIABLES

- `parent = &node(B)`
- `child = &node(D)`
- `parent→right_thread = TRUE`
- `child→right_thread = TRUE`

Case 1: Inserting Under a Parent with No Right Subtree



child->left_thread = TRUE: this is a thread too (red).

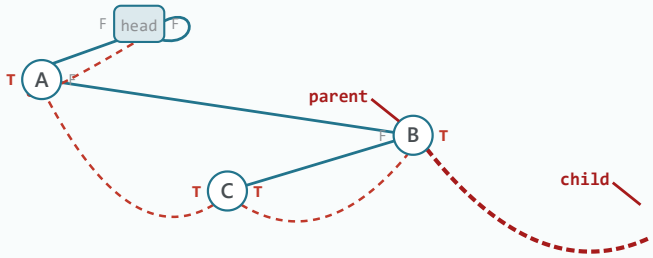
Step 4 / 6

```
1 child->right_child  = parent->right_child;
2 child->right_thread = TRUE;  // parent: no right
3 child->left_child   = parent;
▶ child->left_thread = TRUE;
5 parent->right_child   = child;
6 parent->right_thread = FALSE;  // was TRUE
```

VARIABLES

- `parent = &node(B)`
- `child = &node(D)`
- `parent→right_thread = TRUE`
- `child→right_thread = TRUE`

Case 1: Inserting Under a Parent with No Right Subtree



parent->right_child = child: a real (black) link from parent to child.

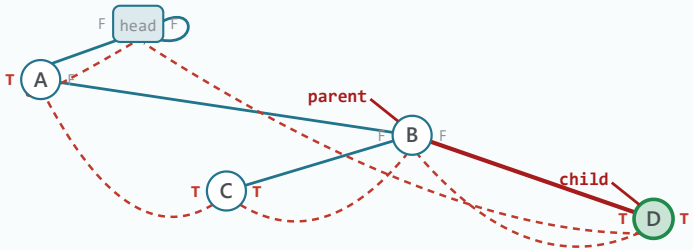
Step 5 / 6

```
1 child->right_child  = parent->right_child;
2 child->right_thread = TRUE;  // parent: no right
3 child->left_child   = parent;
4 child->left_thread  = TRUE;
▶ parent->right_child   = child;
6 parent->right_thread = FALSE;  // was TRUE
```

VARIABLES

- `parent = &node(B)`
- `child = &node(D)`
- `parent→right_thread = TRUE`
- `child→right_thread = TRUE`

Case 1: Inserting Under a Parent with No Right Subtree



parent->right_thread = FALSE: the parent's right is now the child, not a thread. Done: 6 assignments.

Step 6 / 6

```
1 child->right_child  = parent->right_child;
2 child->right_thread = TRUE;  // parent: no right
3 child->left_child   = parent;
4 child->left_thread  = TRUE;
5 parent->right_child   = child;
▶ parent->right_thread = FALSE;  // was TRUE
```

VARIABLES

- parent = &node(B)
- child = &node(D)
- parent→right_thread = FALSE
- child→right_thread = TRUE

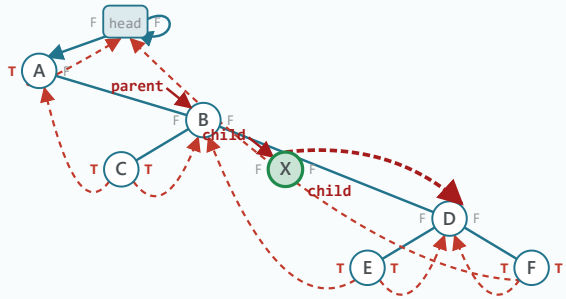
Case 2: the Parent Has a Right Subtree

- Goal: D is B's child, and we make X B's child instead; but the D subtree must **not be lost**, and the red pointers must still point to the right places.
- X's predecessor becomes B. E, the leftmost node of the D subtree, used to have B as its predecessor; now E is X's **successor**.
- So: **the child must become the predecessor of the parent's old successor.**
- In the case 2 code, line 2 is `child->right_thread = FALSE` (the parent's subtree is now the child's); E's left thread must also be fixed.

RESULT: by updating the predecessor of the parent's old successor in case 2, and writing assignment 2 as `child→right_thread = parent→right_thread`, we get **a single routine** that handles both cases.

insert_right(B, X): One Routine, Case 2

```
1  void insert_right(thread_ptr parent,
2                    thread_ptr child) {
3    // make child the right child of parent
4    thread_ptr p;
▶    child->right_child   = parent->right_child;
6    child->right_thread = parent->right_thread;
7    child->left_child   = parent;
8    child->left_thread = TRUE;
9    parent->right_child    = child;
10   parent->right_thread  = FALSE;
11   if (!child->right_thread) { // parent had a subtree?
12     p = insucc(child);
13     p->left_child = child;     // succ's pred is child
14   }
15 }
```



child->right_child = parent->right_child: B's old right subtree (D) goes to X's right.

Step 1 / 9

VARIABLES

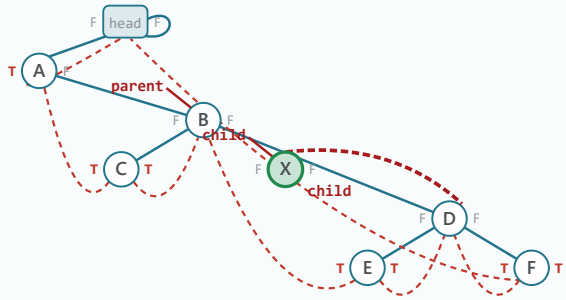
`parent = &node(B)` `child = &node(X)`

`parent→right_thread = FALSE`

`child→right_thread = FALSE`

insert_right(B, X): One Routine, Case 2

```
1  void insert_right(thread_ptr parent,
2                    thread_ptr child) {
3    // make child the right child of parent
4    thread_ptr p;
5    child->right_child   = parent->right_child;
▶    child->right_thread = parent->right_thread;
7    child->left_child   = parent;
8    child->left_thread = TRUE;
9    parent->right_child    = child;
10   parent->right_thread  = FALSE;
11   if (!child->right_thread) { // parent had a subtree?
12     p = insucc(child);
13     p->left_child = child;     // succ's pred is child
14   }
15 }
```



child->right_thread = parent->right_thread (FALSE): the parent's subtree is now the child's; the link stays black.

Step 2 / 9

VARIABLES

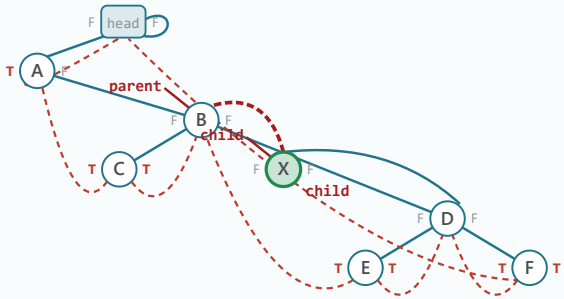
`parent = &node(B)` `child = &node(X)`

`parent→right_thread = FALSE`

`child→right_thread = FALSE`

insert_right(B, X): One Routine, Case 2

```
1  void insert_right(thread_ptr parent,
2                    thread_ptr child) {
3    // make child the right child of parent
4    thread_ptr p;
5    child->right_child   = parent->right_child;
6    child->right_thread = parent->right_thread;
▶    child->left_child   = parent;
8    child->left_thread = TRUE;
9    parent->right_child    = child;
10   parent->right_thread  = FALSE;
11   if (!child->right_thread) { // parent had a subtree?
12     p = insucc(child);
13     p->left_child = child;     // succ's pred is child
14   }
15 }
```



child->left_child = parent: X's predecessor is B.

Step 3 / 9

VARIABLES

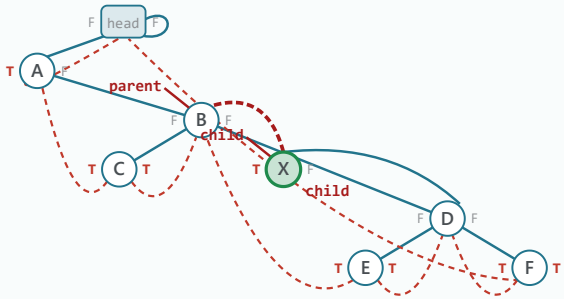
`parent = &node(B)` `child = &node(X)`

`parent→right_thread = FALSE`

`child→right_thread = FALSE`

insert_right(B, X): One Routine, Case 2

```
1  void insert_right(thread_ptr parent,
2                    thread_ptr child) {
3    // make child the right child of parent
4    thread_ptr p;
5    child->right_child   = parent->right_child;
6    child->right_thread = parent->right_thread;
7    child->left_child   = parent;
▶    child->left_thread = TRUE;
9    parent->right_child    = child;
10   parent->right_thread  = FALSE;
11   if (!child->right_thread) { // parent had a subtree?
12     p = insucc(child);
13     p->left_child = child;     // succ's pred is child
14   }
15 }
```



child->left_thread = TRUE: red arrow.

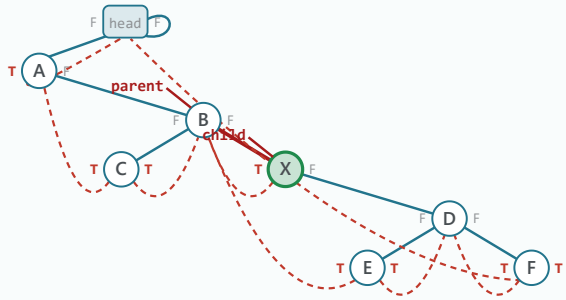
Step 4 / 9

VARIABLES

- `parent = &node(B)`
- `child = &node(X)`
- `parent→right_thread = FALSE`
- `child→right_thread = FALSE`

insert_right(B, X): One Routine, Case 2

```
1  void insert_right(thread_ptr parent,
2                    thread_ptr child) {
3    // make child the right child of parent
4    thread_ptr p;
5    child->right_child   = parent->right_child;
6    child->right_thread = parent->right_thread;
7    child->left_child   = parent;
8    child->left_thread = TRUE;
▶    parent->right_child    = child;
10   parent->right_thread  = FALSE;
11   if (!child->right_thread) { // parent had a subtree?
12     p = insucc(child);
13     p->left_child = child;     // succ's pred is child
14   }
15 }
```



parent->right_child = child: real link B → X.

Step 5 / 9

VARIABLES

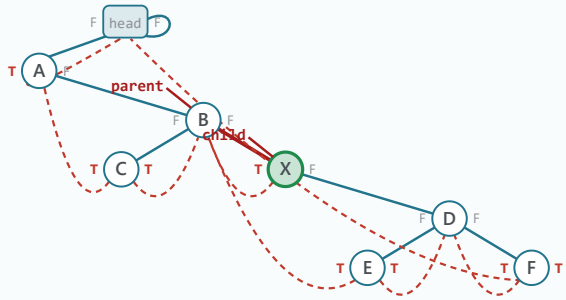
`parent = &node(B)` `child = &node(X)`

`parent→right_thread = FALSE`

`child→right_thread = FALSE`

insert_right(B, X): One Routine, Case 2

```
1  void insert_right(thread_ptr parent,
2                    thread_ptr child) {
3    // make child the right child of parent
4    thread_ptr p;
5    child->right_child   = parent->right_child;
6    child->right_thread = parent->right_thread;
7    child->left_child   = parent;
8    child->left_thread = TRUE;
9    parent->right_child    = child;
▶    parent->right_thread  = FALSE;
11   if (!child->right_thread) { // parent had a subtree?
12     p = insucc(child);
13     p->left_child = child;     // succ's pred is child
14   }
15 }
```



parent->right_thread = FALSE: it was FALSE already; needed for case 1, harmless here. “Don't conclude too soon that the code works!”

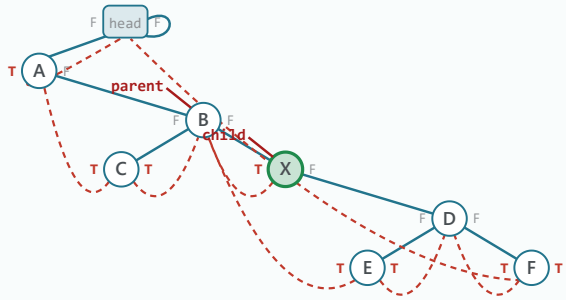
Step 6 / 9

VARIABLES

- parent = &node(B)
- child = &node(X)
- parent→right_thread = FALSE
- child→right_thread = FALSE

insert_right(B, X): One Routine, Case 2

```
1  void insert_right(thread_ptr parent,
2                    thread_ptr child) {
3    // make child the right child of parent
4    thread_ptr p;
5    child->right_child   = parent->right_child;
6    child->right_thread = parent->right_thread;
7    child->left_child   = parent;
8    child->left_thread = TRUE;
9    parent->right_child    = child;
10   parent->right_thread  = FALSE;
▶    if (!child->right_thread) { // parent had a subtree?
12     p = insucc(child);
13     p->left_child = child;     // succ's pred is child
14   }
15 }
```



child->right_thread FALSE → X has a right subtree: the left thread of X's successor still points to **B** and must be fixed.

Step 7 / 9

VARIABLES

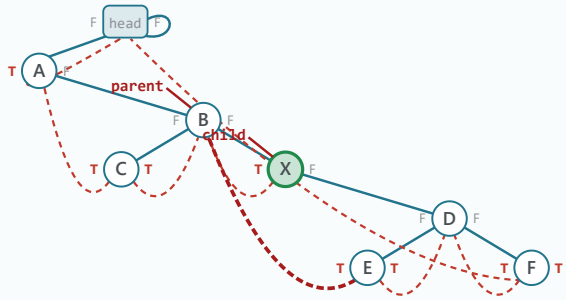
`parent = &node(B)` `child = &node(X)`

`parent→right_thread = FALSE`

`child→right_thread = FALSE`

insert_right(B, X): One Routine, Case 2

```
1  void insert_right(thread_ptr parent,
2                    thread_ptr child) {
3    // make child the right child of parent
4    thread_ptr p;
5    child->right_child   = parent->right_child;
6    child->right_thread = parent->right_thread;
7    child->left_child   = parent;
8    child->left_thread = TRUE;
9    parent->right_child    = child;
10   parent->right_thread  = FALSE;
11   if (!child->right_thread) { // parent had a subtree?
▶      p = insucc(child);
13     p->left_child = child;      // succ's pred is child
14   }
15 }
```



p = insucc(X) = **E** (leftmost of D). E's old predecessor was B.

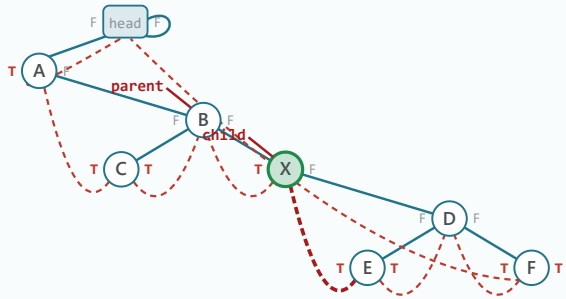
Step 8 / 9

VARIABLES

- `parent = &node(B)`
- `child = &node(X)`
- `parent→right_thread = FALSE`
- `child→right_thread = FALSE`
- `p = &node(E)`

insert_right(B, X): One Routine, Case 2

```
1  void insert_right(thread_ptr parent,
2                    thread_ptr child) {
3    // make child the right child of parent
4    thread_ptr p;
5    child->right_child   = parent->right_child;
6    child->right_thread = parent->right_thread;
7    child->left_child   = parent;
8    child->left_thread = TRUE;
9    parent->right_child    = child;
10   parent->right_thread  = FALSE;
11   if (!child->right_thread) { // parent had a subtree?
12     p = insucc(child);
▶      p->left_child = child;     // succ's pred is child
14   }
15 }
```



p->left_child = child: E's left thread now points to X. **The child became the predecessor of the parent's old successor.**

Step 9 / 9

VARIABLES

`parent = &node(B)` `child = &node(X)`

`parent→right_thread = FALSE`

`child→right_thread = FALSE`

`p = &node(E)`

INSERTING A NODE

insert_right Line by Line

[1–2] A clear definition

Clear name, clear parameters, clear return value.

[5–6]

The child's right pointer and right thread come from the parent: red (TRUE) is kept in case 1, black (FALSE) in case 2.

[7–8]

The child's inorder predecessor is the parent: left pointer to the parent, left thread TRUE (red).

[10]

FALSE into the parent's right thread: needless but harmless in case 2; required in case 1.

[11–13]

If the child's right thread is FALSE (case 2), the successor's left thread is redirected to the child; in case 1 it is one test and return.

Upper/lower case

C is case-sensitive: `parent->Right_child`, `parent->Right_thread` (capital R) do not compile. The condition checks whether the parent has a **right** child.

Where Should the NULL Check Go?

INSTRUCTOR "insert_right is not used on its own but inside another routine, so there is no need for a NULL check here. That check belongs in the routine that calls insert_right. But if someone is going to use insert_right on its own, then we must check this too."

- Example: when inserting into a threaded search tree, once we know whether to go right or left, we call `insert_right` or `insert_left` instead of a single assignment; the parent is never NULL.
- If the routine does not check its inputs, the **comment** must say so: "parent and child must not be NULL".
- A general library routine adds defensive checks (`assert` or an error return).

What Should a Comment Say?

```
/* BAD: repeats the code */
if (!child->right_thread)  // right thread FALSE?
/* GOOD: states intent */
if (!child->right_thread)  // parent had a subtree?
```

- A comment should say **what the code means to do**; it should not repeat the code.
- Then a reader can say “the intent is right but the code is wrong”: debugging gets easier.

INSTRUCTOR “Don't you dare write ‘is the parent's right child FALSE’ here! The next reader shouldn't say ‘I already know that.’ Comments should say things the code doesn't.”

Reducing to One Routine, and Software Engineering

INSTRUCTOR "If we write a separate routine for each case, the fun is gone: noChild_insertRight, hasChild_insertRight, noChild_insertLeft, hasChild_insertLeft. If we can merge them into one without adding complexity, good! … 'We'll nest if… if…, and write every case in one routine.' Avoid that. In that case, keep separate routines."

- First analyze the 3–5 different cases; then decide the routine's name, return value and parameters.
- Define the common work in a common way (here: copying the thread flag from the parent).
- Result: code that is not fragmented; simple, readable, easy to maintain.

INSTRUCTOR "I wish software engineering concepts were taught in first year, but it would be too early. … To me this degree should take 6 years."

QUESTION: Think About the Insert-Left Routine

Write `insert_left`, which inserts the node pointed to by child into the threaded tree as the **left** child of parent.

- Left ↔ right and predecessor ↔ successor swap: the parent's old **predecessor** becomes the child's predecessor, and its right thread is redirected to the child.
- Finding the inorder predecessor: the **rightmost** node of the left subtree if there is one, else the left thread.

Solution: insert_left and inpred

```
thread_ptr inpred(thread_ptr L) {
  thread_ptr p = L->left_child;
  if (!L->left_thread)
    while (!p->right_thread) p = p->right_child;
  return p;
}
void insert_left(thread_ptr parent, thread_ptr child) {
  child->left_child   = parent->left_child;
  child->left_thread = parent->left_thread;
  child->right_child   = parent;
  child->right_thread = TRUE;      // successor: parent
  parent->left_child    = child;
  parent->left_thread  = FALSE;
  if (!child->left_thread) {      // parent had a left subtree
    thread_ptr p = inpred(child);
    p->right_child = child;         // pred's succ is child
  }
}
```

Stay Alert When Coding!

INSTRUCTOR “Stay alert when coding! You can step on dynamite at any moment. ‘I'm doing this, but am I doing it right?’”

INSTRUCTOR “Don't conclude right after this line that the code works. There is still an if block to run.”

Threaded tree

+ Stackless traversal in O(1) space
+ Successor/predecessor of any node in O(h), amortized O(1) during traversal
+ NULL fields are not wasted

The price

– 2 flags per node
– Insert/delete are trickier
– Every routine must maintain the threads

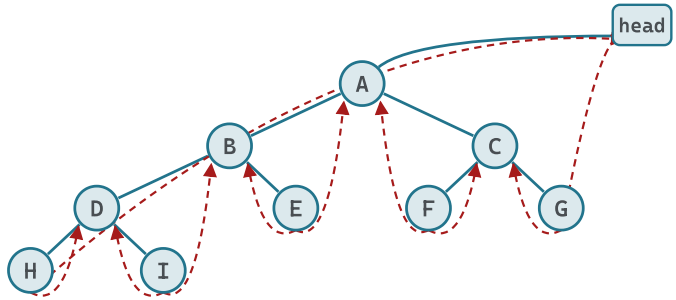
Lab: Threaded Inorder Traversal

H D I B E A F C G

Input (nodes in level order, - for empty) A, B, C, D, E, F, G, H, I Apply

CODE: **tinorder**

```
1 void tinorder(
2                       thread_ptr L) {
3   thread_ptr p = L;     // L: the head node
4   for (;;) {
5     p = insucc(p);
6     if (p == L) break; // back at the head?
7     printf("%c ", p->data);
8   }
9 }
```



p – L –

OUTPUT (none yet)

Threaded tree: an empty left link points to the inorder predecessor, an empty right link to the inorder successor (dashed arrows); the two outermost threads point to the head node. The traversal uses neither a stack nor recursion.

SUMMARY

Complexity Summary

operation	time	extra space	note
insucc	O(h) worst case	O(1)	O(1) if a thread
tinorder	O(n)	O(1)	no stack / recursion
threaded preorder traversal	O(n)	O(1)	same threads
insert_right / insert_left	O(h)	O(1)	successor/predecessor search
Morris traversal (no flags)	O(n)	O(1)	temporary threads
recursive inorder (for comparison)	O(n)	O(h)	call stack

SUMMARY

Common Mistakes

1. Taking `left_child` / `right_child` for a real child without checking the thread flag (infinite loop).
2. Forgetting the head node: H's left thread and G's right thread dangle.
3. Not updating the successor's left thread in case 2 of inserting right.
4. Hard-coding `child->right_thread` as TRUE and breaking case 2.
5. Searching for the root in an empty tree, forgetting the head's `left_thread = TRUE`.
6. Case: `Right_child` ≠ `right_child` (C is case-sensitive).

SUMMARY

Exercises

1. Draw the threads of the tree A(B(D, E), C(–, F)); how many threads are there?
2. Write `inpred`, which finds the inorder **predecessor**.
3. Write a routine that traverses a threaded tree in **reverse** inorder (G C F A …).
4. Is it correct to insert the first node into an empty tree (just the head) with `insert_left(head, x)`?
5. Why is postorder traversal hard with threads? What extra information makes it easy?
6. Write insertion into a threaded binary **search** tree using `insert_right` / `insert_left`.

▶ **Hints / answers**

SUMMARY

Chapter Summary

n + 1 NULL

n−1 of the 2n pointers are used; the rest become threads.

Rules

Left NULL → predecessor, right NULL → successor; flags tell them apart; the head collects the dangling threads.

Successor

Directly if the right thread is TRUE; otherwise the leftmost node of the right subtree.

Traversal

Stackless, O(n) time, O(1) space; preorder works too.

Insertion

insert_right: the thread flag is copied from the parent; in case 2 the child becomes the successor's predecessor.

Code quality

Clear definition, meaningful comments, the right place for the NULL check, one routine.

SUMMARY

References and Next Steps

- Horowitz–Sahni–Anderson-Freed, *Fundamentals of Data Structures in C*, 5.5 Threaded Binary Trees.
- A. J. Perlis, C. Thornton, “Symbol manipulation by threaded lists”, *CACM* 3(4), 1960.
- J. M. Morris, “Traversing binary trees simply and cheaply”, *Information Processing Letters* 9(5), 1979; Knuth, TAOCP 1, 2.3.1.

← Chapter 11: Search Trees

Course page

Chapter 13: Priority Queues and Heaps

The End

Chapter 12: Threaded Binary Trees

Next chapter: Chapter 13: Priority Queues and Heaps · Course page