Chap 5. Trees (3)

Contents

5.4 Additional Binary Tree Operations

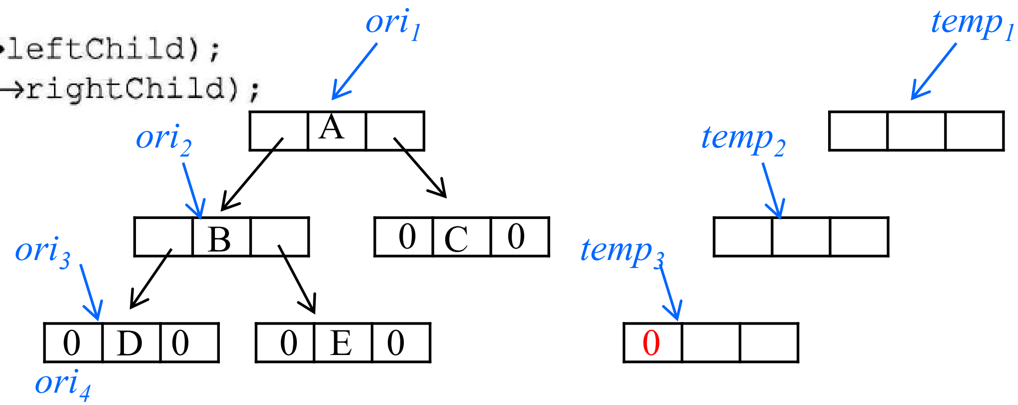
5.4.1 Copying Binary trees

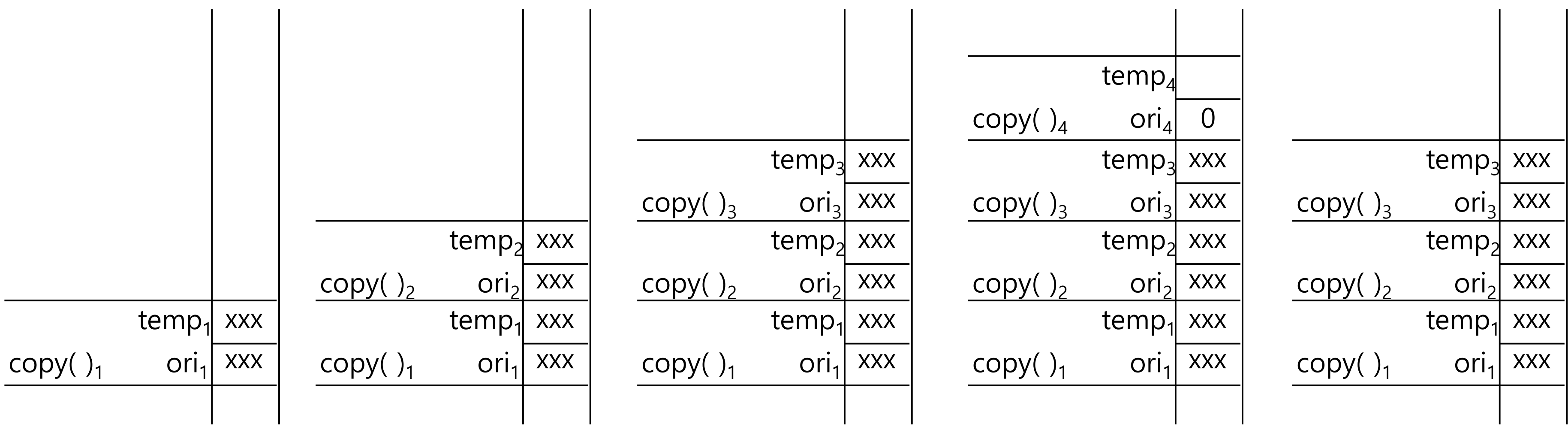
- A slightly modified version of *postorder* traversal

```
treePointer copy(treePointer original)
{/* this function returns a treePointer to an exact copy
    of the original tree */
  treePointer temp;
  if (original) {
    MALLOC(temp, sizeof(*temp));
    temp→leftChild = copy(original→leftChild);
    temp→rightChild = copy(original→rightChild);
    temp→data = original→data;
    return temp;
  }
  return NULL;
}
```

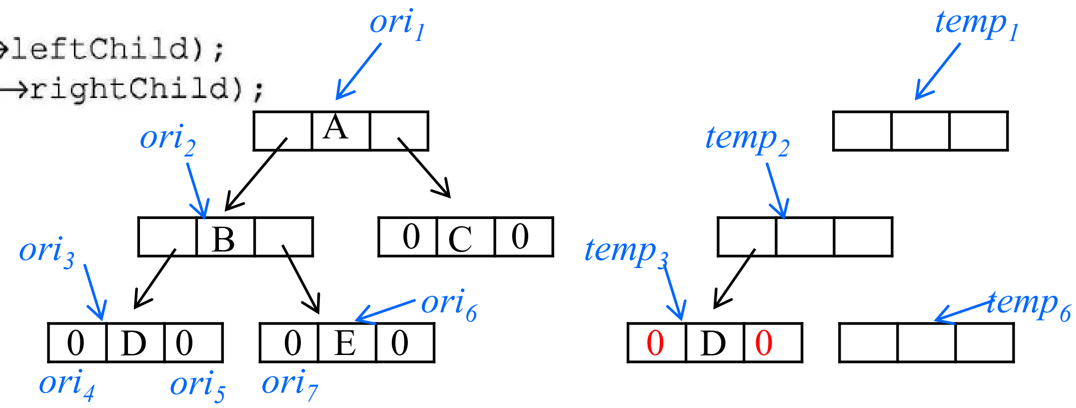
Program 5.6: Copying a binary tree

```
treePointer copy(treePointer original)
{
   treePointer temp;
   if (original) {
      MALLOC(temp, sizeof(*temp));
      temp→leftChild = copy(original→leftChild);
      temp→rightChild = copy(original→rightChild);
      temp→data = original→data;
      return temp;
   }
   return NULL;
}
```





```
treePointer copy(treePointer original)
{
   treePointer temp;
   if (original) {
      MALLOC(temp, sizeof(*temp));
      temp→leftChild = copy(original→leftChild);
      temp→rightChild = copy(original→rightChild);
      temp→data = original→data;
      return temp;
   }
   return NULL;
}
```



	$temp_5$	
copy( )$_5$	ori_5	0
	$temp_3$	xxx
copy( )$_3$	ori_3	xxx
	$temp_2$	xxx
copy( )$_2$	ori_2	xxx
	$temp_1$	xxx
copy( )$_1$	ori_1	xxx

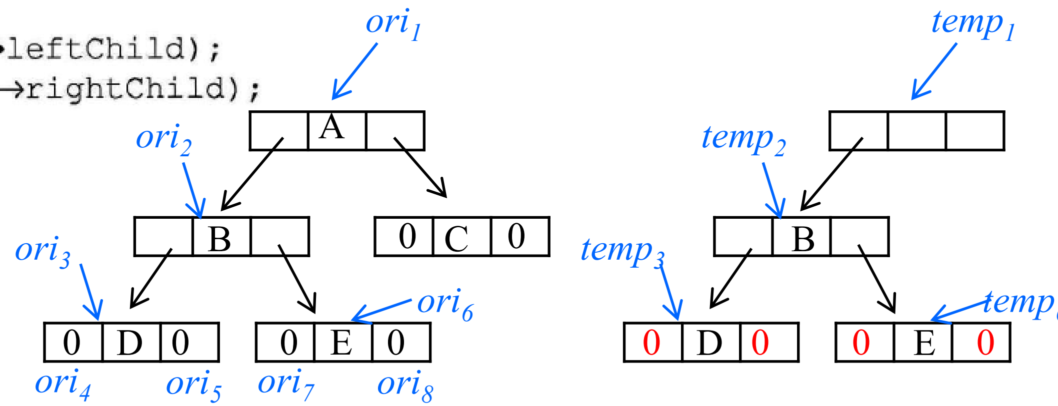
	$temp_3$	xxx
copy( )$_3$	ori_3	xxx
	$temp_2$	xxx
copy( )$_2$	ori_2	xxx
	$temp_1$	xxx
copy( )$_1$	ori_1	xxx

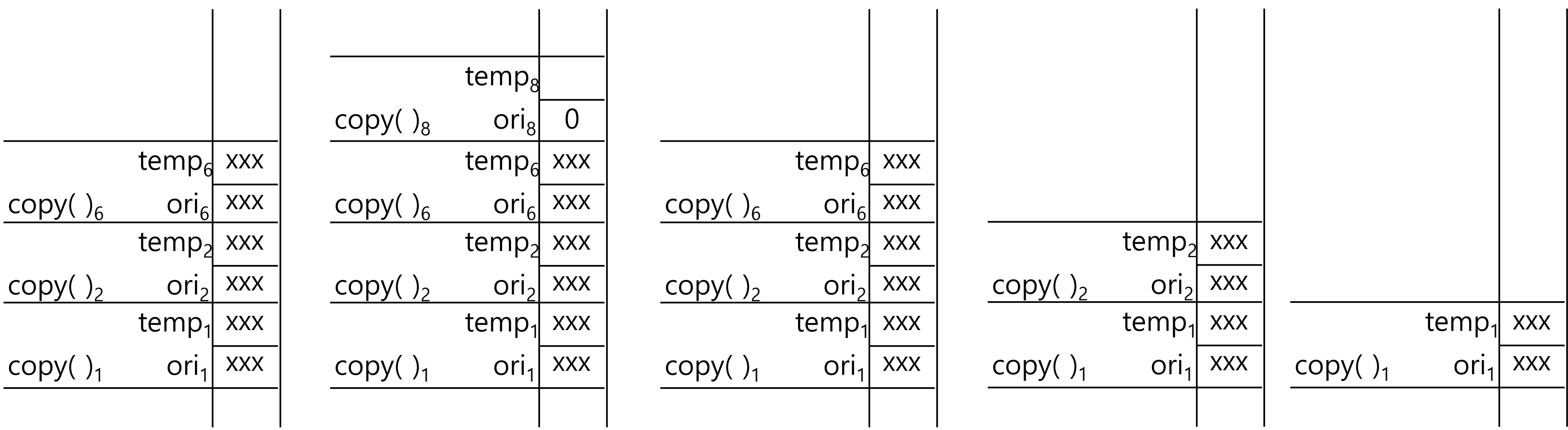
	$temp_2$	xxx
copy( )$_2$	ori_2	xxx
	$temp_1$	xxx
copy( )$_1$	ori_1	xxx

	$temp_6$	xxx
copy( )$_6$	ori_6	xxx
	$temp_2$	xxx
copy( )$_2$	ori_2	xxx
	$temp_1$	xxx
copy( )$_1$	ori_1	xxx

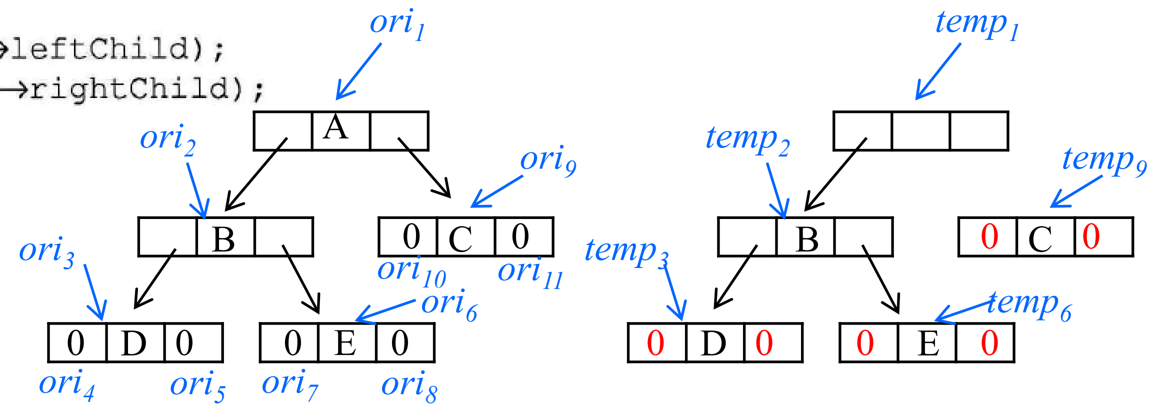
	$temp_7$	
copy( )$_7$	ori_7	0
	$temp_6$	xxx
copy( )$_6$	ori_6	xxx
	$temp_2$	xxx
copy( )$_2$	ori_2	xxx
	$temp_1$	xxx
copy( )$_1$	ori_1	xxx

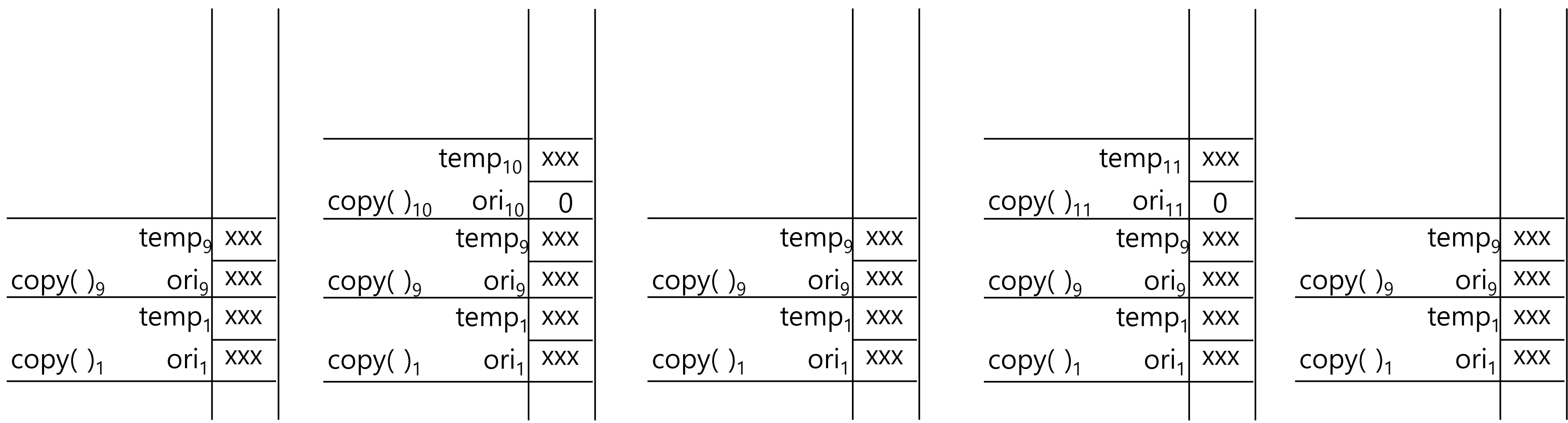
```
treePointer copy(treePointer original)
{
  treePointer temp;
  if (original) {
    MALLOC(temp, sizeof(*temp));
    temp→leftChild = copy(original→leftChild);
    temp→rightChild = copy(original→rightChild);
    temp→data = original→data;
    return temp;
  }
  return NULL;
}
```



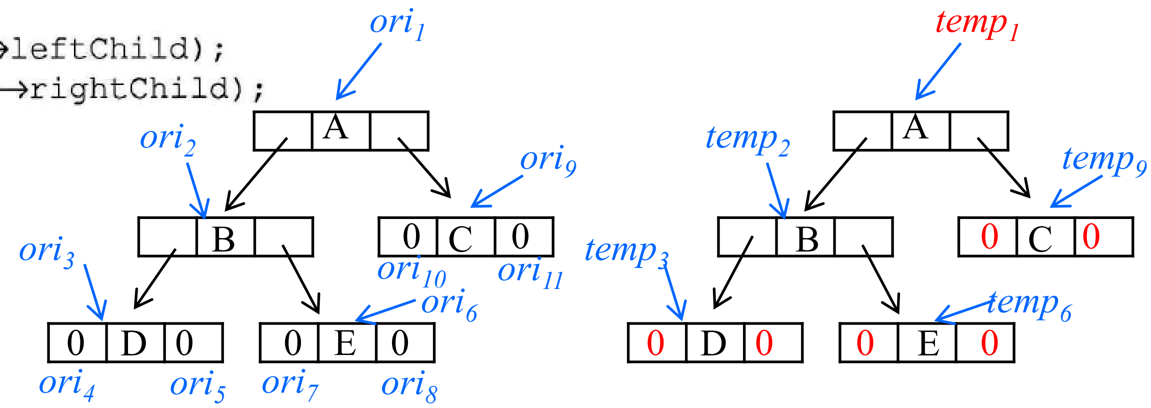


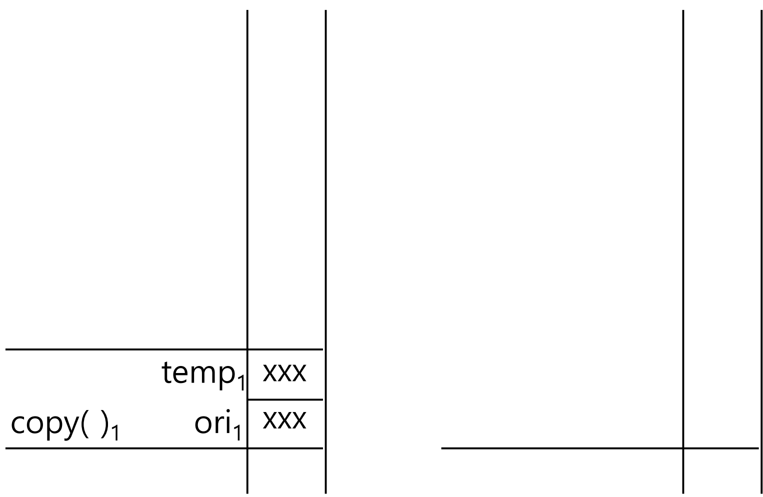
```
treePointer copy(treePointer original)
{
   treePointer temp;
   if (original) {
      MALLOC(temp, sizeof(*temp));
      temp→leftChild = copy(original→leftChild);
      temp→rightChild = copy(original→rightChild);
      temp→data = original→data;
      return temp;
   }
   return NULL;
}
```





```
treePointer copy(treePointer original)
{
   treePointer temp;
   if (original) {
      MALLOC(temp, sizeof(*temp));
      temp→leftChild = copy(original→leftChild);
      temp→rightChild = copy(original→rightChild);
      temp→data = original→data;
      return temp;
   }
   return NULL;
}
```





Contents

5.4 Additional Binary Tree Operations

5.4.2 Testing Equality

- Determining the equivalence of two binary trees
- Equivalent binary trees have the *same structure* and the *same information* in the corresponding nodes.
- A modification of *preorder* traversal

```
int equal(treePointer first, treePointer second)
{/* function returns FALSE if the binary trees first and
    second are not equal, Otherwise it returns TRUE */
   return ((!first && !second) || (first && second &&
              (first→data == second→data) &&
              equal(first→leftChild,second→leftChild) &&
              equal(first→rightChild, second→rightChild))
}
```

Program 5.7: Testing for equality of binary trees

5.4.3 The Satisfiability Problem

- Consider the set of formulas from $\{x_1, .., x_n\}$ and $\{\wedge$ (and), $\vee$ (or), $\neg$(not) $\}$

- The *variables* are *Boolean variables*
  - Have only two possible values, *true* or *false*

- Set of expressions are defined by the following rules
  - A variable is an expression
  - If x and y are expression, then $\neg x$, $x \wedge y$, $x \vee y$ are expressions
  - Parentheses can be used to alter the normal order of evaluation

- formula of propositional calculus : $x_1 \vee (x_2 \wedge \neg x_3)$
  - If x_1 and x_3 are *false* and x_2 is *true*, it is *true*

- ***The satisfiability problem***
  - Is there an assignment of values to the variables that causes the value of the expression to be true?

- The most obvious algorithm
  - let (x_1, .., x_n) take on all possible combinations of *true* and *false* values and to check the formula for each combination
    - O($g2^n$), or exponential time, where g is the time to substitute values for x_1, x_2, …, x_n and evaluate the expression.
  - ***Postorder*** evaluation

- $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$

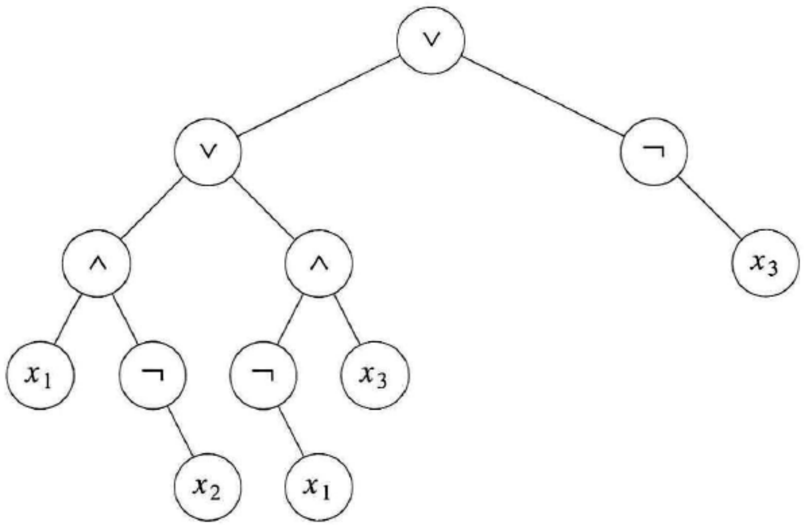


Figure 5.18: Propositional formula in a binary tree

For $n = 3$,
All possible combinations
of *true* = *t*, *false* = *f*
$(t, t, t), (t, t, f), (t, f, t), (t, f, f)$
$(f, t, t), (f, t, f), (f, f, t), (f, f, f)$

```
typedef enum {not,and,or,true,false} logical;
typedef struct node *treePointer;
typedef struct node {
        treePointer leftChild;
        logical     data;    // the value of a variable or an operator
        short int   value;   // TRUE/FALSE
        treePointer rightChild;
        } node;
```

leftChild	*data*	*value*	*rightChild*

```
for (all 2^n possible combinations) {
   generate the next combination;
   replace the variables by their values;
   evaluate root by traversing it in postorder;
   if (root→value) {
      printf(<combination>);
      return;
   }
}
printf("No satisfiable combination\n");
```

Program 5.8: First version of satisfiability algorithm

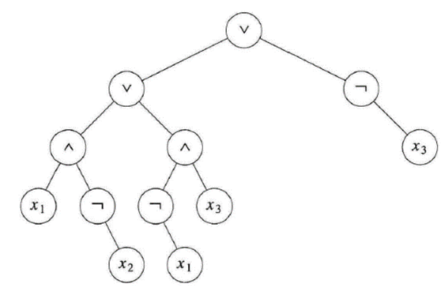
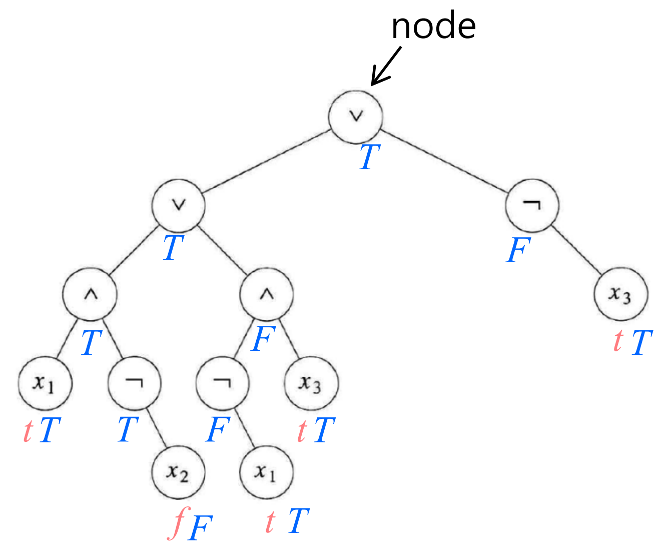


Figure 5.18: Propositional formula in a binary tree

```
void postOrderEval(treePointer node)
{/* modified post order traversal to evaluate a
    propositional calculus tree */
   if (node) {
      postOrderEval(node→leftChild);
      postOrderEval(node→rightChild);
      switch(node→data) {
         case not:   node→value =
                     !node→rightChild→value;
                     break;
         case and:   node→value =
                     node→rightChild→value &&
                     node→leftChild→value;
                     break;
         case or:    node→value =
                     node→rightChild→value ||
                     node→leftChild→value;
                     break;
         case true:  node→value = TRUE;
                     break;
         case false: node→value = FALSE;
      }
   }
}
```

Program 5.9: Postorder evaluation function



ex) for a combination $(x_1, x_2, x_3) = (t, f, t)$

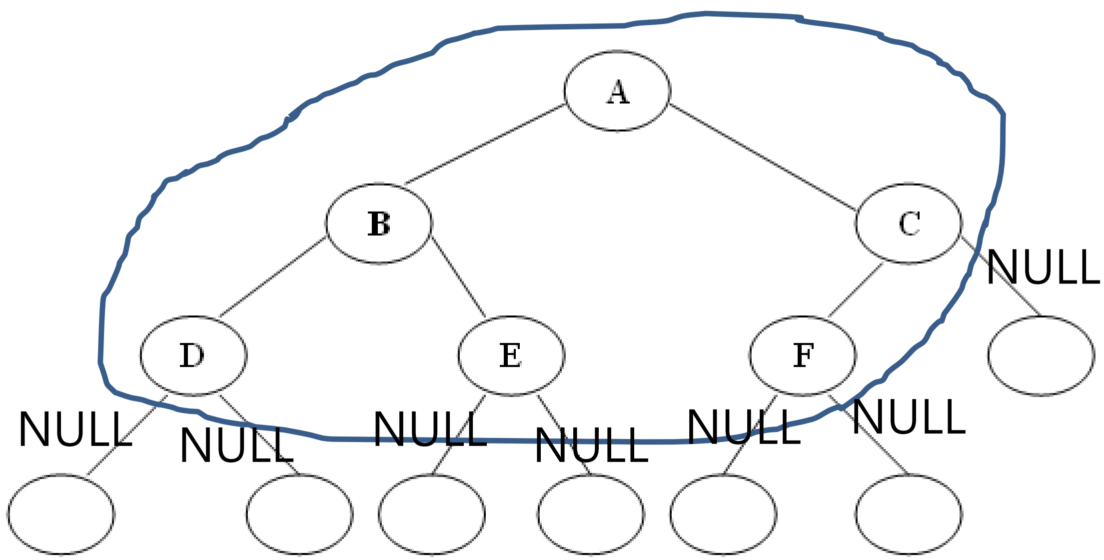
not/and/or in the data field of non-leaf nodes

true/false in the data field of leaf nodes, x_1, x_2, and x_3

Threaded Binary Tree

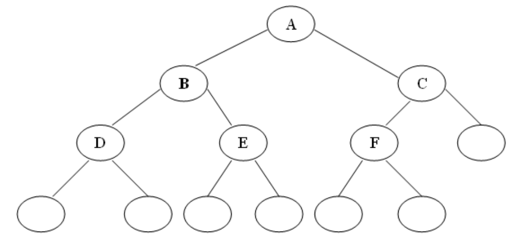
Threaded Binary Tree

- In a linked representation of a binary tree, the number of null links (null pointers) are actually more than non-null pointers.
- Consider the following binary tree:



A Binary tree with the null pointers

Threaded Binary Tree



A Binary tree with the null pointers

- In above binary tree, there are 7 null pointers & actual 5 pointers.
- In all there are 12 pointers.
- We can generalize it that for any binary tree with n nodes there will be (n+1) null pointers and 2n total pointers.
- The objective here to make effective use of these null pointers.
- to replace all the null pointers by the appropriate pointer values called threads.

5.51. Threads

Construct the threads

(1) If *leftChild* is null, replace *leftChild* with a pointer to the node that would be visited before *ptr* in an inorder traversal.

That is we replace the null link with a pointer to the *inorder predecessor* of *ptr.*

(2) If *rightChild* is null, replace *rightChild* with a pointer to the node that would be visited after *ptr* in an inorder traversal.

That is we replace the null link with a pointer to the *inorder successor* of *ptr.*

5.51. Threads

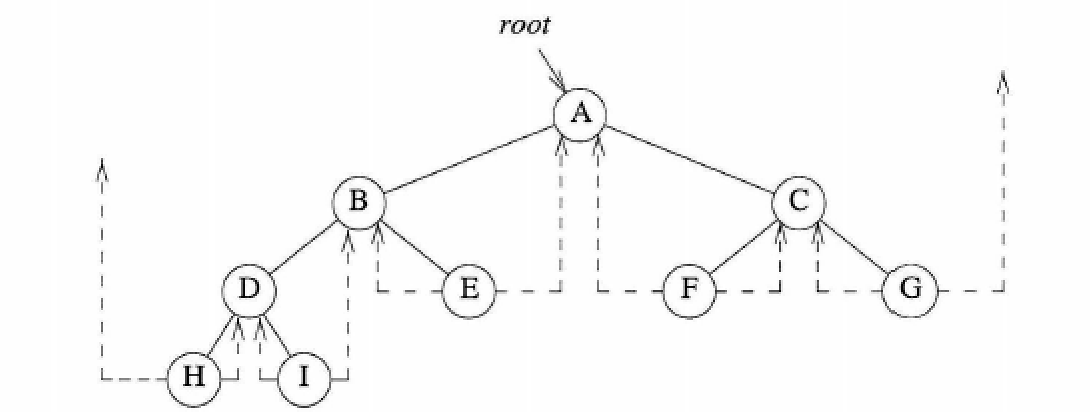


Figure 5.21: Threaded tree corresponding to Figure 5.10(b)

```
typedef struct threadedTree  *threadedPointer;
typedef struct threadedTree {
        short int leftThread;
        threadedPointer leftChild;
        char data;
        threadedPointer rightChild;
        short int rightThread;
} ;
```

5.5.1. Threads

An empty binary tree is represented by its header node as in Figure 5.22.

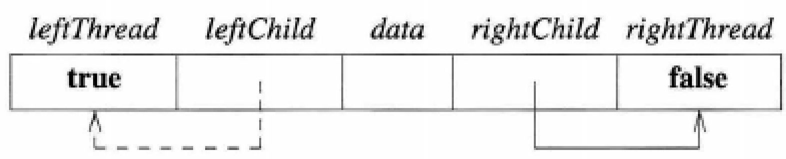


Figure 5.22: An empty threaded binary tree

5.5.1. Threads

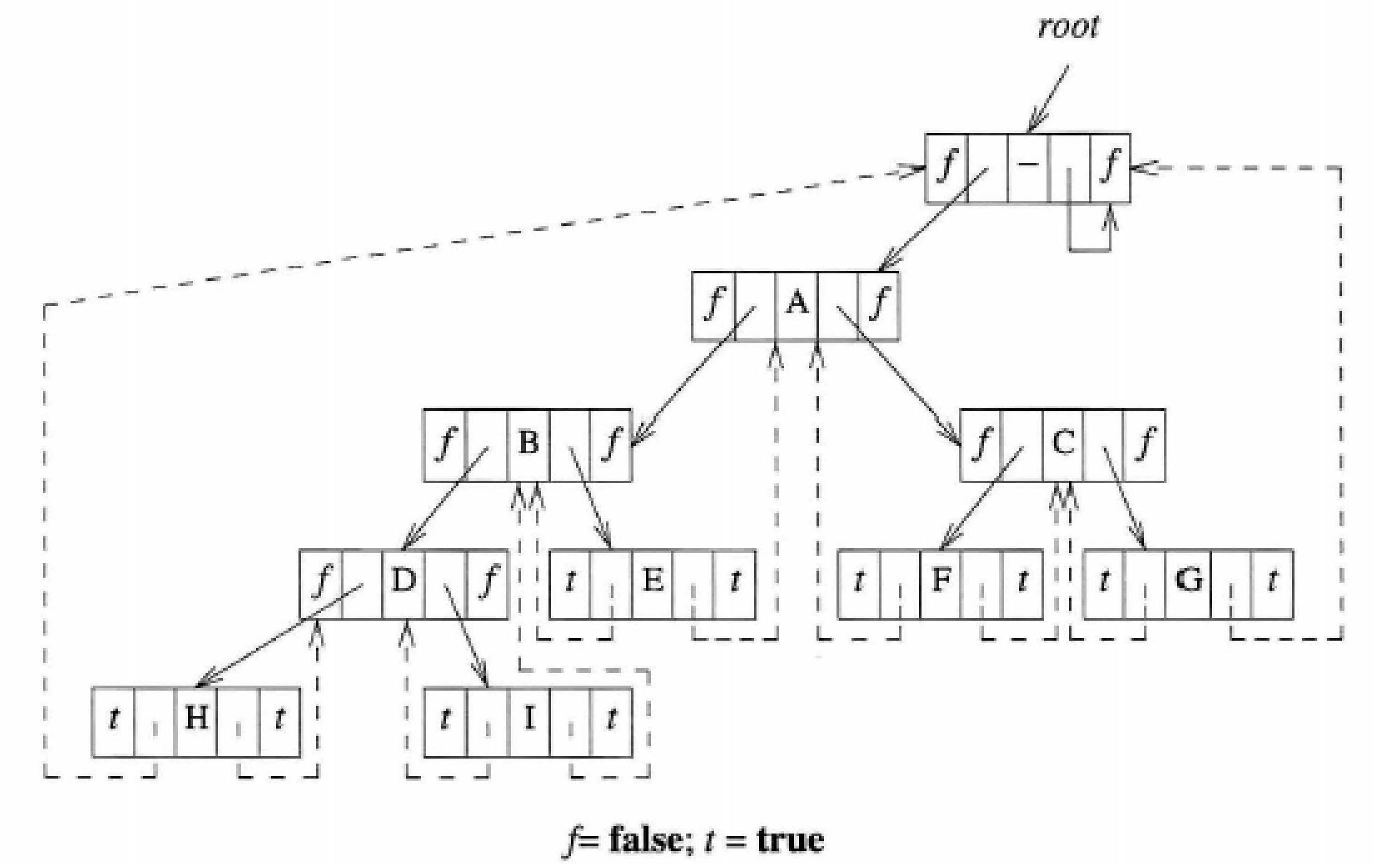
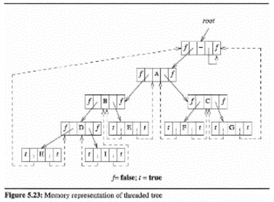


Figure 5.23: Memory representation of threaded tree

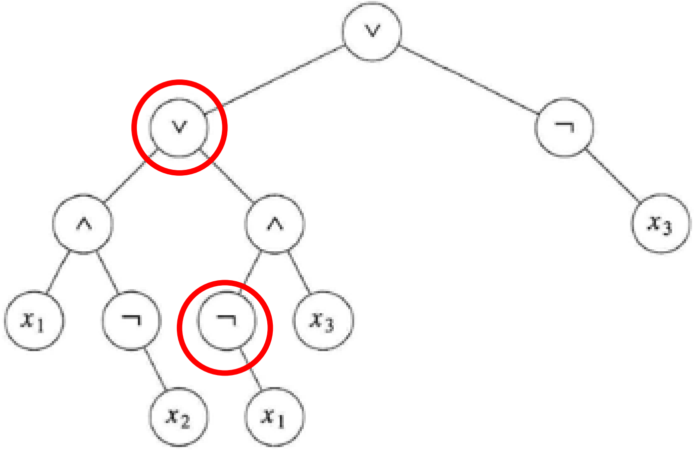
5.5.2. Inorder Traversal of a Threaded Binary Tree

- By using the threads, we can perform an inorder traversal without making use of a stack.
- The function *insucc* finds the inorder successor of any node in a threaded tree without using a stack.



```
threadedPointer insucc(threadedPointer tree)
{/* find the inorder sucessor of tree in a threaded binary
    tree */
   threadedPointer temp;
   temp = tree→rightChild;
   if (!tree→rightThread)
      while (!temp→leftThread)
         temp = temp→leftChild;
   return temp;
}
```

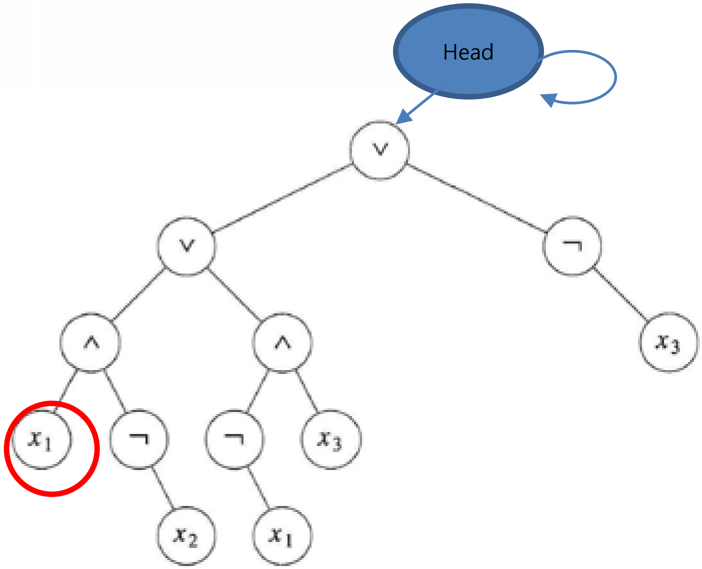
Program 5.10: Finding the inorder successor of a node



5.5.2. Inorder Traversal of a Threaded Binary Tree

```
void tinorder(threadedPointer tree)
{/* traverse the threaded binary tree inorder */
   threadedPointer temp = tree;
   for (;;) {
      temp = insucc(temp);
      if (temp == tree) break;
      printf("%3c", temp→data);
   }
}
```

Program 5.11: Inorder traversal of a threaded binary tree



5.5.3 Inserting a Node into a Threaded Binary Tree

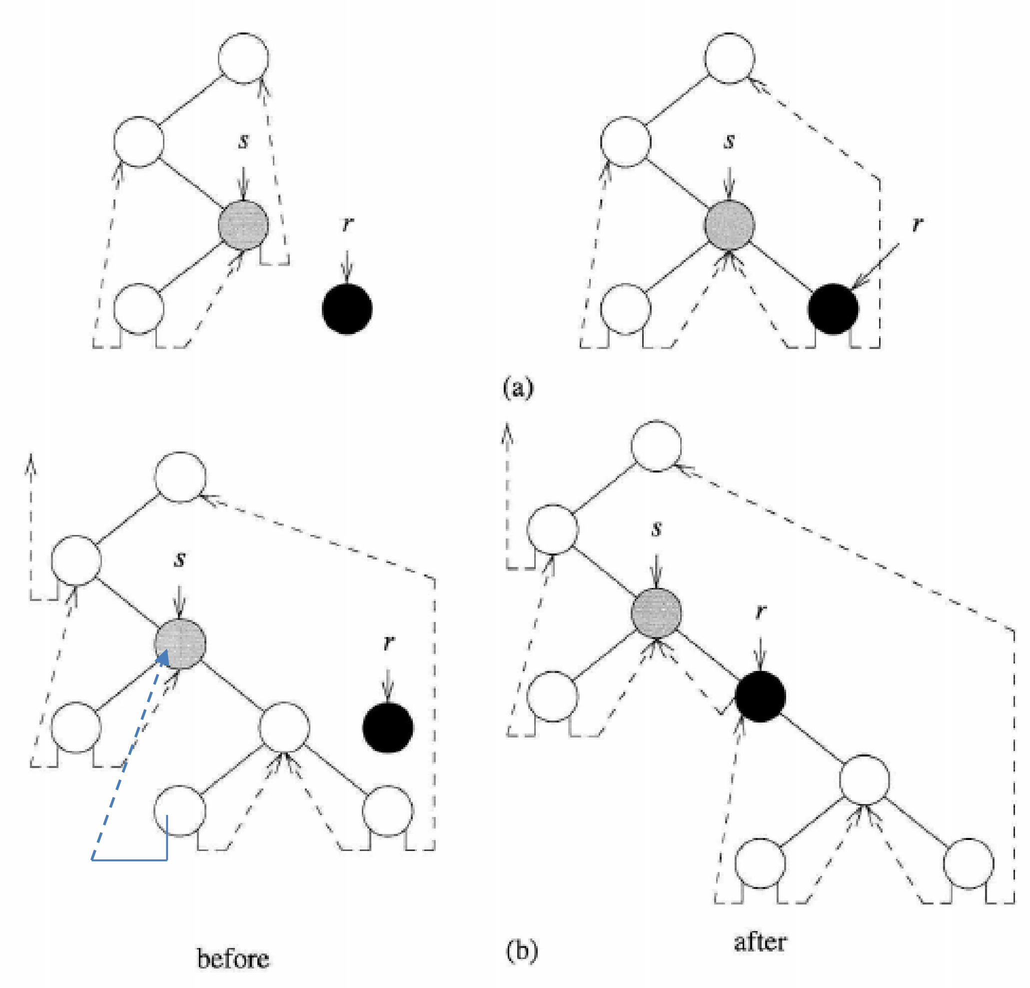
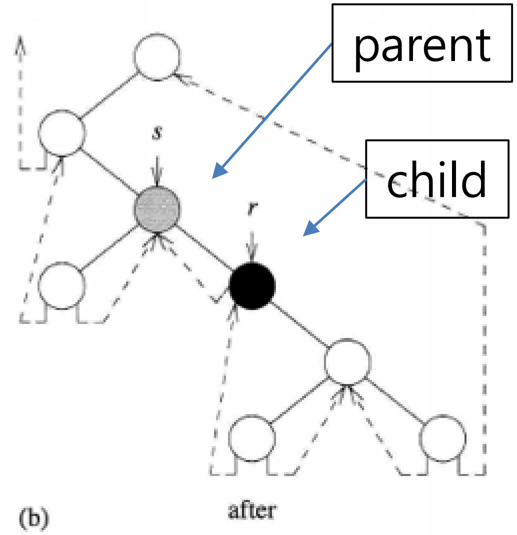


Figure 5.24: Insertion of *r* as a right child of *s* in a threaded binary tree

5.5.3 Inserting a Node into a Threaded Binary Tree

```
void insertRight(threadedPointer s, threadedPointer r)
{/* insert r as the right child of s */
   threadedPointer temp;
   r→rightChild =      s →rightChild;
   r→rightThread =      s →rightThread;
   r→leftChild =      s ;
   r→leftThread = TRUE;
   s→rightChild =      r ;
   s→rightThread = FALSE;
   if (!r→rightThread) {
      temp = insucc(r);
      temp→leftChild = r;
   }
}
```



Program 5.12: Right insertion in a threaded binary tree