ELIMINATING RECURSION

TAIL RECURSION

```
 1 class Counter1
 2 {
 3    public static void main(String[] args)
 4    {
 5       countDown(2);
 6    }
 7    //---------------------------------
 8    public static void countDown(int n)
 9    {
10       if (n > 0)
11       {
12          System.out.println(n);
13          countDown(n - 1);      // tail recursive call
14       }
15    }
16 }
```

Figure 1

Line 13 in Fig. 1 is a call of `countdown` *within* the `countDown` method. For this reason, we refer to the call on line 13 as a **recursive call**. When the execution of line 13 completes, nothing else in the current invocation of `countDown` is executed. `countdown` simply returns to its caller. We refer to a recursive call of this sort as a **tail recursive call** because it occurs at the "tail end" of the execution of the method.

The recursive call on line 13 is in effect a jump to the beginning of the `countDown` method, but with a value of `n` one less the its current value. That is, the action of the recursion in `countDown` is essentially a loop in which the value of `n` is 2 for the first iteration, 1 for the second, and 0 for the third. Thus, to eliminate the recursion in `countDown`, we simply change the `if` to a `while`, and the recursive call to a statement that decrements `n` by 1 (see Fig. 2). The programs in Fig. 1 and Fig. 2 are equivalent. They both display

```
2
1
```

Note that when `n` is 0, `countDown` simply returns to its caller. Thus, it does not display 0.

```
 1 class Counter2
 2 {
 3    public static void main(String[] args)
 4    {
 5       countDown(2);
 6    }
 7    //---------------------------------
 8    public static void countDown(int n)
 9    {
10       while (n > 0)             // while in place of if
11       {
12          System.out.println(n);
13          n = n - 1;             // decrement parameter in place of recursive call
14       }
15    }
16 }
```

Figure 2

```
 1 class Counter3
 2 {
 3    public static void main(String[] args)
 4    {
 5       countDown(2);
 6    }
 7    //---------------------------------
 8    public static void countDown(int n)
 9    {
10       if (n > 0)
11       {
12          System.out.println(n);
13          countDown(n - 1);      // tail recursion
14       }
15       else
16          System.out.println("bottom");
17    }
18 }
```

Figure 3

You might think that the recursive call on line 13 in Fig. 3 is not tail recursive because executable code (the `println` on line 16) follows the call on line 13. But line 16 is executed only if the `if` part of the statement is *not* executed. Thus, after the recursive call on line 13, no further execution occurs. `countDown` immediately returns to its caller.

Let's call the first execution of `countdown`, which is triggered by line 5, as level 1. Level 2 is the second execution of `countDown`, which is triggered by the recursive call on line 13. Level 3 is the third (and last) execution of `countDown`, which is also triggered by the recursive call on line 13. As `countDown` recurses down to level 0, it displays `n` (except at level 0). Thus, it displays 2, then 1. `n` is 0 at level 3 so the `else` part of the `if` statement is executed, which displays "`bottom`". Then each level returns to the previous level: Level 3 returns to level 2; level 2 then *immediately* (because the call is tail recursive) returns to level 1; level 1 then *immediately* returns to `main`, and the program ends. During these returns, nothing is displayed because the recursive calls are tail recursive. Thus, the program displays

```
2
1
bottom
```

`countDown` in Fig. 3 in effect executes a loop. In each iteration, `n` is displayed and decremented. Thus, the loop version (see Fig. 4) is easily implemented with a `while` loop which displays and then decrements `n`. When `n` reaches 0, the loop exit occurs after which "`bottom`" is displayed. The programs in Fig. 3 and Fig. 4 are equivalent.

```
 1 class Counter4
 2 {
 3    public static void main(String[] args)
 4    {
 5       countDown(2);
 6    }
 7    //---------------------------------
 8    public static void countDown(int n)
 9    {
10       while (n > 0)
11       {
12          System.out.println(n);
13          n = n - 1;
14       }
15       System.out.println("bottom");
16    }
17 }
```

Figure 4

RECURSION THAT IS NOT TAIL RECURSIVE

```
 1 class Counter5
 2 {
 3    public static void main(String[] args)
 4    {
 5       countUp(2);
 6    }
 7    //---------------------------------
 8    public static void countUp(int n)
 9    {
10       if (n > 0)
11       {
12          countUp(n - 1);        // not tail recursive
13          System.out.println(n);
14       }
15    }
16 }
```

Figure 5

The recursion on line 12 in Fig. 5 is *not* tail recursive because a statement (the `println` on line 13) follows the call. Recall that on each call of a method, its parameters are created and initialized with the corresponding arguments in the call. Thus, on the first execution of `countUp` (which we refer to as level 1), the parameter `n` is created and initialized with 2 (which is the argument in the call on line 5). On the second execution of `countUp` (level 2), *another* `n` is created and initialized with 1. On the last call of `countUp` (level 3) a *third* `n` is created and initialized with 0. The recursion mechanism in effect saves the value of the parameter `n` each time `countUp` is called. Thus, each level of recursion has its own `n`. When level 3 returns to level 2, it is easy for the recursive mechanism to reestablish the `n` at level 2 (the `n` with 1) because *it still exists*. Line 13 can then display that `n`. Similarly, when level 2 returns to level 1, it is easy for the recursive mechanism to reestablish the `n` at level 1 (the `n` with 2). Line 13 can then display that `n`. The program in Fig. 5 counts up. That is, it displays

```
1
2
```

To get the loop version of the program in Fig. 5, we have to include code that saves each non-zero value of `n`. The data structure that is natural for this purpose is the stack. A stack is a LIFO (last-in-first-out) data structure. That is, the last item added to a stack is the first one to be removed. Only one end of the stack (the **top**) is used for additions and removals. We call the operation of adding an item to the top of the stack a `push`, and the operation of removing an item from the top of the stack a `pop`.

A stack is easily implemented with an array. Here is the code that creates a stack of integers:

```
int[] stack = new int[100];
int top = -1;
```

The variable `top` holds the index of the last item pushed onto the stack or -1 if the stack is empty. To push the value in `n` onto the stack, we use

```
stack[++top] = n;
```

We preincrement `top` so it has the index of the next available slot in the `stack` array. We then assign `n` to that slot. To pop the top of the stack into `n`, we use

```
n = stack[top--];
```

`top` has the index of the top item on the stack. Thus, we first use the index it contains and then postdecrement it. To determine if the stack is empty, we simply check if the value in `top` is -1.

Let's now look at Fig. 6, the loop version of the program in Fig. 5.

```
 1 class Counter6
 2 {
 3    public static void main(String[] args)
 4    {
 5       countup(2);
 6    }
 7    //---------------------------------
 8    public static void countup(int n)
 9    {
10       int[] stack = new int[100];
11       int top = -1;
12       while (n > 0)
13       {
14          stack[++top] = n;        // save n before decrementing it
15          n = n - 1;
16       }
17       while (top != -1)
18       {
19          n = stack[top--];       // restore n from stack
20          System.out.println(n);  // display n
21       }
22    }
23 }
```

Figure 6

We replace the recursive call with a `while` loop (lines 12 to 16) that decrements `n` on each iteration. But before it decrements `n`, it saves it by pushing it on a stack. This `while` loop simulates what happens in the recursive version as it recurses down to the bottom level. The second `while` loop (lines 17 to 21 simulates what happens in the recursive version when each level returns to its caller. It restores `n` by popping it from the stack, and then displays it. It continues this until the stack becomes empty.

The technique we used to eliminate recursion in the program in Fig. 5 can be used to eliminate recursion in the program that traverses a linked list in reverse order (see Fig. 7).

```
 1 class LT1
 2 {
 3    static Node head = null; // static so addNode/traverse can access
 4    public static void main(String[] args)
 5    {
 6       addNode(1);      // add node with 1 to the beginning of list
 7       addNode(2);      // add node with 2 to the beginning of list
 8       addNode(3);      // add node with 3 to the beginning of list
 9       rtraverse(head); // last-to-first traversal
10    }
11    //---------------------------------
12    public static void addNode(int value)
13    {
14       Node p = new Node(); // create new node
15       p.x = value;         // initialize x field in this node
16       p.link = head;       // make new node point to current 1st node
17       head = p;            // make head point to new node
18    }
19    //---------------------------------
20    public static void rtraverse(Node p)  // traverse from last to 1st
21    {
22       if (p != null)
23       {
24          rtraverse(p.link);        // display tail in reverse order
25          System.out.println(p.x);  // display data in 1st node
26       }
27    }
28 }
29 //===================================================================
30 class Node         // Node definition
31 {
32    Node link;      // points to next node or is null if at last node
33    int x;
34 }
```

Figure 7

To eliminate the recursive call on line 24, we change the `if` statement on line 22 to a `while` loop which on each iteration modifies `p` so that it points to the next node on the list. But before it does that, it saves `p` by pushing it on a stack:

```
24       while (p != null)
25       {
26          stack[++top] = p;    // save p on the stack
27          p = p.link;          // move p to next node on list
28       }
```

A second `while` loop pops a node pointer off the stack, assigns it to `p`, and then displays `p.x` (the data in the node that `p` points to):

```
29       while (top != -1)
30       {
31          p = stack[top--];
32          System.out.println(p.x);
33       }
```

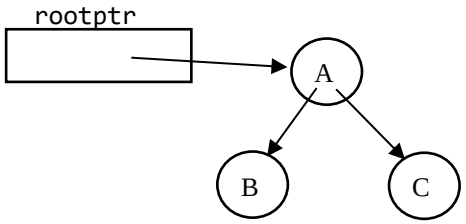
Thus, the pointers to the nodes in the linked list are pushed in their natural order (i.e., first to last). But they are popped and used to access the data on the linked list in reverse order. The result is that the data in displayed in reverse order (i.e., from last to first). Fig. 8 shows the complete program.

```
 1 class LT2
 2 {
 3    static Node head = null; // static so addNode/traverse can access
 4    public static void main(String[] args)
 5    {
 6       addNode(1);      // add node with 1 to the beginning of list
 7       addNode(2);      // add node with 2 to the beginning of list
 8       addNode(3);      // add node with 3 to the beginning of list
 9       rtraverse(head); // last-to-first traversal
10    }
11    //---------------------------------
12    public static void addNode(int value)
13    {
14       Node p = new Node(); // create new node
15       p.x = value;         // initialize x field in this node
16       p.link = head;       // make new node point to current 1st node
17       head = p;            // make head point to new node
18    }
19    //---------------------------------
20    public static void rtraverse(Node p)  // traverse from last to 1st
21    {
22       Node[] stack = new Node[100];
23       int top = -1;
24       while (p != null)
25       {
26          stack[++top] = p;
27          p = p.link;
28       }
29       while (top != -1)
30       {
31          p = stack[top--];
32          System.out.println(p.x);
33       }
34    }
35 }
36 //=================================================================
37 class Node         // Node definition
38 {
39    Node link;      // points to next node or is null if at last node
40    int x;
41 }
```

Figure 8

ELIMINATING RECURSION IN TREE TRAVERSAL PROGRAM

Before we discuss how to eliminate recursion in tree traversal programs, let's review the terminology used for trees. Consider the following tree:



Node A is the **root node**. It is the one node that has no node pointing to it. In this tree, the root node is also the **parent node** of nodes B and C. Node B is the **left child** of node A. Node C is the **right child** of node A. Nodes B and C are **leaf nodes** because they have neither a left child node nor a right child node. The **height** of the tree (i.e., the number of levels) is 2. A **binary tree** is a tree in which each node has at most two child nodes.

A binary tree of height 1 has one node. A binary tree of height 2 has as many as 3 nodes. A binary tree of height 3 has as many as 7 nodes. Generalizing, we can say that a tree of height *n* can have as many as $2^n - 1$ nodes. Thus, a tree of height 30 can have as many as $2^{30} - 1$, which is roughly one billion. Clearly, the traversal of a typical tree with a height of only 30 will *necessarily* require a great deal of computation time because it has so many nodes.

Figure 9 shows a program that creates a binary tree and then traverses it with the standard method that uses double recursion (see lines 54 to 62). The use of double recursion in a method is often an indication that the method is grossly inefficient. However, the double recursion in the `inorder` traversal method in Fig. 9 is simply an indication that traversals of trees with even a relatively small height necessarily require a great deal of computation. Thus, the `inorder` method in Fig. 9 is a perfectly reasonably way to traverse a binary tree for most applications.

```
 1 import java.util.Random;
 2 class TT1
 3 {
 4    public static void main(String[] args)
 5    {
 6       MyTree t = new MyTree();
 7       Random r = new Random();
 8       for (int i = 1; i <= 10; i++)     // insert 10 numbers
 9          t.insert(r.nextInt(300));      // numbers range from 0 to 299
10       t.inorder();                      // inorder traversal
11    }
12 }
13 //=====================================================
14 class MyTree
15 {
16    private class Node
17    {
18       private int x;
19       private Node left;
20       private Node right;
21    }
22    //--------------------------------------
23    private Node rootptr = null;
24
25    public void insert(int xx)
26    {
27       Node n = new Node();
28       n.x = xx;
```

```
29       if (rootptr == null)
30          rootptr = n;
31       else
32       {
33          Node trailing = null, leading = rootptr;
34          while (leading != null)
35          {
36             trailing = leading;
37             if (xx < leading.x)
38                leading = leading.left;
39             else
40                leading = leading.right;
41          }
42          if (xx < trailing.x)
43             trailing.left = n;
44          else
45             trailing.right = n;
46        }
47    }
48    //---------------------------------------
49    public void inorder()
50    {
51       inorder(rootptr);
52    }
53    //---------------------------------------
54    private void inorder(Node p)    // uses double recursion
55    {
56       if (p != null)               // standard inorder traversal
57       {
58          inorder(p.left);          // traverse left subtree with recursive call
59          System.out.println(p.x);  // visit root
60          inorder(p.right);         // traverse right subtree with recursive call
61       }
62    }
63 }
```

Figure 9

The second recursive call in Fig. 9 (line 60) is tail recursive. Thus, it can be easily eliminated by changing the `if` on line 56 to `while` and changing line 60 to an assignment statement that assigns `p.right` to `p`. We get the program in Fig. 10 (see lines 54 to 62).

```
 1 import java.util.Random;
 2 class TT2
 3 {
 4    public static void main(String[] args)
 5    {
 6       MyTree t = new MyTree();
 7       Random r = new Random();
 8       for (int i = 1; i <= 10; i++)     // insert 10 numbers
 9          t.insert(r.nextInt(300));      // numbers range from 0 to 299
10       t.inorder();                      // inorder traversal
11    }
12 }
13 //================================================
14 class MyTree
15 {
16    private class Node
```

```
17    {
18       private int x;
19       private Node left;
20       private Node right;
21    }
22    //---------------------------------------
23    private Node rootptr = null;
24
25    public void insert(int xx)
26    {
27       Node n = new Node();
28       n.x = xx;
29       if (rootptr == null)
30          rootptr = n;
31       else
32       {
33          Node trailing = null, leading = rootptr;
34          while (leading != null)
35          {
36             trailing = leading;
37             if (xx < leading.x)
38                leading = leading.left;
39             else
40                leading = leading.right;
41          }
42          if (xx < trailing.x)
43             trailing.left = n;
44          else
45             trailing.right = n;
46        }
47    }
48    //---------------------------------------
49    public void inorder()
50    {
51       inorder(rootptr);
52    }
53    //---------------------------------------
54    private void inorder(Node p)
55    {
56       while (p != null)    // inorder traversal with tail recursion eliminated
57       {
58          inorder(p.left);  // this call is not tail recursive
59          System.out.println(p.x);
60          p = p.right;
61       }
62    }
63 }
```

Figure 10

In Fig. 10, we still have one recursive call (line 58). This call is not tail recursive. To eliminate this recursive call, we will use the technique we used for the `countUp` (Fig. 5) and `rtraverse` (Fig. 7) examples. Recall that in these examples, the non-tail recursive call is within an `if` statement (see Fig. 5, line 12 and Fig. 7, line 24). In contrast, the recursive call in Fig. 10 is within a `while` statement—not an `if` statement (see lines 56 and 58). As a result, the elimination of the recursive call is more complicated.

In Fig. 10, the recursive call on line 58 is repeatedly executed taking the traversal down the left pointers starting from the root node. When `p` goes null, the recursive mechanism returns to the previous level—a level in which `p` necessarily is pointing to a node with no left child (which is why `p` went null). But it may have a right child. Thus, line 60 in Fig. 10 assigns `p` the pointer to the right child and then repeats the `while` loop that starts on line 56. We can replicate this behavior without using recursion using a `goto` statement and a label. A `goto` statement transfers control to the specified label. Java does *not* support the `goto` statement. But because it is the easiest way to show the flow of control in the non-recursive inorder traversal of a binary tree, we use it in Fig. 11.

```
 1 private void inorder(Node p)
 2 {
 3       Node[] stack = new Node[100];
 4       int top = -1;
 5       while (p != null)
 6       {
 7          stack[++top] = p;          // saves current p on the stack
 8          p = p.left;
 9          continue;                  // causes a jump back to line 5
10
11 LLL:     System.out.println(p.x);   // labels not supported in Java
12          p = p.right;
13       }
14       if (top != -1)
15       {
16         p = stack[top--];
17         goto LLL;                   // goto not supported in Java
18       }
19 }
20
21
```

Figure 11

In Fig. 11, the recursive call is simulated on lines 7, 8, and 9. Line 7 saves the current `p` by pushing it onto the stack. Line 8 assigns `p` its new value (the pointer to left child node). Line 9 then jumps back to line 5, which is the beginning of the `while` loop. When `p` goes null, the exit from the `while` loop occurs. Line 16 then restores the previous `p` by popping the stack. The `goto` on line 27 then transfers control to the code at the label `LLL`, which is on line 11 at which point the `while` loop continues its execution with the new `p`.

Our last step in the conversion to an inorder traversal that does not use recursion is to eliminate the use of the `goto` statement. This is easy to do. We simply replace line 17 with lines 11 and 12 (see lines 69 and 70 in Fig. 12). We terminate the body of the `while` loop where the `continue` statement is (see line 64 in Fig. 12) by replacing `continue` with `}`. Finally, we have to continue the traversal process until the stack goes empty. To do this, we use an outer `while` loop whose exit test specifies the constant `true` (which, of course, is always true). This loop, however, is not an infinite loop because of the `break` statement on line 67. This `break` statement breaks out of the outer `while` loop when the stack goes empty.

```
 1 import java.util.Random;
 2 class TT4
 3 {
 4    public static void main(String[] args)
 5    {
 6       MyTree t = new MyTree();
 7       Random r = new Random();
 8       for (int i = 1; i <= 10; i++)     // insert 10 numbers
 9          t.insert(r.nextInt(300));      // numbers range from 0 to 299
10       t.inorder();                      // inorder traversal
11    }
12 }
13 //=================================================
14 class MyTree
15 {
16    private class Node
17    {
18       private int x;
19       private Node left;
20       private Node right;
21    }
22    //-----------------------------------------
23    private Node rootptr = null;
24
25    public void insert(int xx)
26    {
27       Node n = new Node();
28       n.x = xx;
29       if (rootptr == null)
30          rootptr = n;
31       else
32       {
33          Node trailing = null, leading = rootptr;
34          while (leading != null)
35          {
36             trailing = leading;
37             if (xx < leading.x)
38                leading = leading.left;
39             else
40                leading = leading.right;
41          }
42          if (xx < trailing.x)
43             trailing.left = n;
44          else
45             trailing.right = n;
46        }
47    }
48    //-----------------------------------------
49    public void inorder()
50    {
51       inorder(rootptr);
52    }
53    //-----------------------------------------
54    private void inorder(Node p)
55    {
56       Node[] stack = new Node[100];
57       int top = -1;
58       while (true)      // not an infinite loop because of break statement
```

```
59       {
60          while (p != null)
61          {
62             stack[++top] = p;
63             p = p.left;
64          }
65
66          if (top == -1)         // if no more paths to follow, break out of loop
67             break;
68          p = stack[top--];      // restore p for parent node
69          System.out.println(p.x);
70          p = p.right;           // traverse right subtree
71       }
72    }
73 }
```

Figure 12

To fully understand recursion, you really need to see what happens at the machine level when a recursive method is executed. An excellent description of this is in our book *C and C++ Under the Hood*, 2nd Edition (see Chapter 6) available from Amazon.

In 1968, Donald Knuth proposed the following problem: Traverse a binary tree without using recursion or an auxiliary data structure, such as a stack. Several solutions to this problem have been proposed which we briefly summarize here:

1) **Link inversion** with tag bits (Schorr, Waite). As the traversal proceeds down the tree, the pointers are reversed (i.e., changed from parent pointing to child to child pointing to parent) so that the algorithm can backtrack back up the tree allowing it to traverse the other paths in the tree. A tag bit in each node is required.
2) **Link inversion with no tag bits** (Robson). Does not require tag bits as the Schorr-Waite algorithm does.
3) **Dynamically threading tree** (Morris algorithm). As the traversal proceeds down the tree, some of the null link fields are assigned pointers so that the algorithm can backtrack by following these pointers.
4) **D-tree algorithm** (Mayo, Pardo, Dos Reis). The link fields do not contain absolutes addresses but address differences. Such a tree is called a **D-tree**. This allows backtracking without any modification of the tree during traversal.

All these algorithms are illustrated with the C code below. The best way to learn how these algorithms work is to go through code instruction by instruction to see exactly what each one does. You can also refer the references that are listed after the C code for additional information on each algorithm.

```
/* Tree traversal C code for 64-bit system using
   1) Traversal using recursion
   2) Link inversion with tag bits (Schorr-Waite)
   3) Link inversion with no tag bits (Robson)
   4) Dynamically threading tree (Morris algorithm)
   5) D-tree algorithm (Mayo, Pardo, Dos Reis)
*/
#include <stdio.h>
#include <stdlib.h>
typedef struct NODE* LINK;
struct NODE
{
   int data;
   int tag;        // not used for D-tree algorithm
```

```
   int rightchild; // set to 1 in right child nodes, 0 otherwise
   LINK left;      // pointer to left child node
   LINK right;     // pointer to right child node
};
LINK rootptr = NULL;
//=================================================
// adds node to a binary tree
void addnode(LINK *r, LINK newnode, int rightchild)
{
    if (*r == NULL)
    {
       *r = newnode;
       newnode->rightchild = rightchild; // set to 1 for right child
       return;
    }
    if (newnode->data < (*r)->data)
    {
       addnode(&(*r)->left, newnode, 0);  // 0 indicates not right child
    }
    else
    {
       addnode(&(*r)->right, newnode, 1); // 1 indicates right child node
    }
}
//=================================================
// traverse using recursion
void recurse(LINK p)
{
    if (p)
    {
       printf("preorder %d\n", p->data);
       recurse(p->left);
       printf("             inorder %d\n", p->data);
       recurse(p->right);
       printf("                         postorder %d\n", p->data);
    }
}
//=================================================
// musical chairs on three links
void rotate(LINK *x, LINK *y, LINK *z)
{
    LINK temp;
    temp = *x;
    *x = *y;
    *y = *z;
    *z = temp;
}
//=================================================
// traverse using link inversion with tags technique (Schorr-Waite)
```

```
void linkInversionWithTags(LINK rootptr)
{
    LINK current=rootptr, prev=rootptr;
    if (rootptr != NULL)
       while (1)
       {
           // advance
           while (current != NULL)
           {
              printf("preorder %d\n", current->data);
              rotate(&prev, &current, &current->left);
           }

           // backtrack
           while (prev->tag == 1)
           {
               prev->tag = 0;    // reset tag
               printf("                          postorder %d\n", prev->data);
               rotate(&prev, &prev->right, &current);
               if (current == rootptr) return;
           }

           // switch
           printf("             inorder %d\n", prev->data);
           prev->tag = 1;
           rotate(&prev->right, &prev->left, &current);
       }
}
//=================================================
// traverse using link inversion without tags (Robson)
void linkInversionWithoutTags(LINK rootptr)
{
   LINK current=rootptr, prev=rootptr, top=NULL, leaf, savetop;
   if (rootptr != NULL)
      while (1)
      {
          while (current != NULL)
          {
             printf("preorder %d\n", current->data);
             rotate(&prev, &current, &current->left);  // advance
          }
          if (prev->right == NULL)             // prev points to a leaf node
             leaf = prev;                      // remember this leaf node
          while (prev->right == NULL ||
                 prev->left == NULL ||
                 (top && (top->right == prev))
                )
          {
             if (prev->right == NULL)          // no right subtree
```

```
            {
               printf("             inorder %d\n", prev->data);
               printf("                          postorder %d\n", prev->data);
               rotate(&prev, &prev->left, &current);  // backtrack left
            }
            else
            if (prev->left == NULL)           // no left subtree/
            {
               printf("                          postorder %d\n", prev->data);
               rotate(&prev, &prev->right, &current);  // backtract right
            }
            else
            {
               savetop = top;
               top = top->left;  // pop stack
               savetop->left = savetop->right = NULL;  // reset leaf
               printf("                          postorder %d\n", prev->data);
               rotate(&prev, &prev->right, &current);  // backtrack
            }
            if (current == rootptr) return;
         }
         printf("             inorder %d\n", prev->data);  // switch
         if (current != NULL)
         {
            leaf->right = prev;
            leaf->left = top;    // push prev node onto stack
            top = leaf;
         }
         rotate(&prev->right, &prev->left, &current);
      }
}
//================================================
// Dynamically thread and unthread tree (Morris algorithm)
void morris(LINK rootptr)
{
    LINK current, predecessor;

    if (rootptr == NULL)
        return;

    current = rootptr;
    while (current != NULL)
    {

        if (current->left == NULL)
        {
            printf("             inorder %d\n", current->data);
            current = current->right;
        }
```

```
        else
        {
            // Find the inorder predecessor of current node
            predecessor = current->left;
            while (predecessor->right != NULL && predecessor->right != current)
                predecessor = predecessor->right;

            // Make inorder predessor point to current node
            if (predecessor->right == NULL)
            {
                predecessor->right = current;
                current = current->left;
            }

            // Reset changed link field to NULL
            else
            {
                predecessor->right = NULL;
                printf("             inorder %d\n", current->data);
                current = current->right;
            }
        }
    }
}
//================================================
// Convert a standard binary tree to a D-tree
void convertToDtree(LINK p, LINK above)
{
    LINK left, right;
    if (p)
    {
       left = p->left;      // save left pointer
       right = p->right;    // save right pointer

       // if left ptr non-null then change to (address below) - (address above)
       if (left)
          p->left = (LINK)((long long)(left) - (long long)above);

       // if right ptr non-null then change to (address below) - (address above)
       if (right)           // if non-null then change
          p->right = (LINK)((long long)(right) - (long long)above);

       convertToDtree(left, p);
       convertToDtree(right, p);
    }
}
//================================================
// Traverse D-tree
void traverseDtree(LINK rootptr)
```

```
{
    LINK child = rootptr, parent = rootptr, childSave, parentSave;
    int right = 0;

    if (rootptr == NULL) return;
    while (1)
    {
        // advance if left not null
        while (child->left != NULL)
        {
           printf("preorder %d\n", child->data);

           childSave = child;
           child = (LINK)((long long)parent + (long long)(child->left));
           parent = childSave;
        }
        printf("preorder %d\n", child->data);

        /* Backtrack if right subtree already traversed or if null.
           When a backtrack from the right occurs, the next operation
           should be another backtrack operation (because the backtrack
           from the right indicates the right subtree has just been
           traversed). The variable right is set to true (i.e., 1) on
           a backtrack from the right, which has the effect of continuing
           the backtracking sequence. Backtracking also continues if the
           right link in the child node is null (because in that case
           there is no right subtree to traverse).
        */
        while (right || child->right == NULL)
        {
            if (child->right == NULL)
               printf("             inorder %d\n", child->data);
            printf("                          postorder %d\n", child->data);

            if (child == rootptr) return;

            parentSave = parent;
            if (child->rightchild == 1)      // 1 if right child
            {
               // backtrack from right
               parent = (LINK)((long long)child - (long long)(parent->right));
               right = 1;
            }
            else
            {
               // backtrack from left
               parent = (LINK)((long long)child - (long long)(parent->left));
               right = 0;
            }
```

```
            child = parentSave;
        }
        printf("             inorder %d\n", child->data);

        // switch child so child points to right subtree
        childSave = child;
        child = (LINK)((long long)parent + (long long)(child->right));
        parent = childSave;
    }
 }
//================================================
int main(void)
{
    LINK newnode, p;
    int x;
    printf("Enter integers terminated with Ctrl-z (Windows) or Ctrl-d\n");
    // create binary search tree
    while (scanf("%d", &x) == 1)
    {
        newnode =(LINK ) malloc(sizeof(struct NODE));
        newnode->data = x;
        newnode->left = NULL;
        newnode->right = NULL;
        addnode(&rootptr, newnode, 0);   // 0 indicates not a right child
    }
    printf("\n");

    printf("\nTraversal using recursion\n");
    recurse(rootptr);

    printf("\nSchorr-Waite (link inversion with tag bits)\n");
    linkInversionWithTags(rootptr);

    printf("\nRobson (link inversion without tag bits)\n");
    linkInversionWithoutTags(rootptr);

    printf("\nMorris (dynamically threading tree)\n");
    morris(rootptr);

    // Convert standard binary tree to D-tree
    convertToDtree(rootptr, rootptr);
    printf("\nD-tree\n");
    traverseDtree(rootptr);
}

/* Sample run
Enter integers terminated with Ctrl-z (Windows) or Ctrl-d
2 1 3
```

```
Traversal using recursion
preorder 2
preorder 1
             inorder 1
                         postorder 1
             inorder 2
preorder 3
             inorder 3
                         postorder 3
                         postorder 2

Schorr-Waite (link inversion with tag bits)
preorder 2
preorder 1
             inorder 1
                         postorder 1
             inorder 2
preorder 3
             inorder 3
                         postorder 3
                         postorder 2

Robson (link inversion without tag bits)
preorder 2
preorder 1
             inorder 1
                         postorder 1
             inorder 2
preorder 3
             inorder 3
                         postorder 3
                         postorder 2

Morris (dynamically threading tree)
             inorder 1
             inorder 2
             inorder 3

D-tree
preorder 2
preorder 1
             inorder 1
                         postorder 1
             inorder 2
preorder 3
             inorder 3
                         postorder 3
                         postorder 2
*/
```

References

1. Dos Reis A. J., Li Yun: "Traversing: a simple matter," Unix Review, June, 30-36 (1991)
2. Dwyer B.: "Simple Algorithms for Traversing a Tree Without an Auxiliary Stack," Information Processing Letters, 2, 143-145 (1974)
3. Knuth D. E.: The Art of Computer Programming Volume 1 First Edition Fundamental Algorithms, page 325, problem 21, Addison-Wesley, Reading, Massachusetts (1968)
4. Knuth D. E.: The Art of Computer Programming Volume 1 Third Edition Fundamental Algorithms, page 332, problem 21, Addison-Wesley, Reading, Massachusetts (1973)
5. Lindstrom G.: "Scanning list structures without stacks or tag bits," Information Processing Letters, 2, 47-51 (1973)
6. Mateti P., Manghirmalani R.: "Morris' tree traversal algorithm reconsidered," Science of Computer Programming, 11, 1, 29-43 (1988)
7. Mayo K., Pardo C., Dos Reis A.: "Another Solution to Knuth's Tree Traversal Problem", *International Journal of Algorithm Design and Analysis*, Volume 8, Issue 2, 2022.
8. Morris J. M.: "Traversing binary trees simply and cheaply," Information Processing Letters, 9, 197-200 (1979)
9. Robson J. M.: "An improved algorithm for traversing binary trees without auxiliary stack," Information Processing Letters, 2, 12-14 (1973)
10. Schorr H., Waite W. M., "An efficient machine independent procedure for garbage collection in various list structures," Comm. ACM 10, 8, 501-506 (1967)
11. Standish T. A.: Data Structure Techniques, Addison-Wesley, Reading, Massachusetts (1980)
12. http://en.wikipedia.org/wiki/XOR_linked_list

Homework Problems

1) By experimenting with the `inorder` methods in `TT1.java` and `TT4.java`, determine how inefficient the `inorder` method in `TT1.java` is compared with the `inorder` method in `TT4.java`. Is it worthwhile to eliminate recursion in tree traversals? In your experiments, comment out the `println` statement that displays the data in the root. The reduction in execution time will be the same for both versions. You will then be able to get a better estimate of the difference in execution times for the two versions of the `inorder` method. You will have to insert a large number of integers into the tree to get an execution time long enough to measure. Be sure to exclude the time it takes to create the tree. Time only the execution time of the `inorder` method. To time, you can use `System.currentTimeMillis()` which returns the current time in milliseconds as a type `long` value.

2) Modify `TT5.java` so that it does a preorder traversal (root, left subtree, right subtree) without recursion.

3) Modify `TT6.java` so that it does a postorder traversal (left subtree, right subtree, root) without recursion.

4) Modify `TT7.java` so that the stack is implemented with an `ArrayList` rather than an array. What is the advantage of using an `ArrayList`?

5) Convert the Java code in `TT4.java` to C code. Use the `goto` approach illustrated in Fig. 11. Note that C supports the `goto` statement, although it is rarely used.