Basic algorithms with linked lists and binary trees

Linked List Traversal

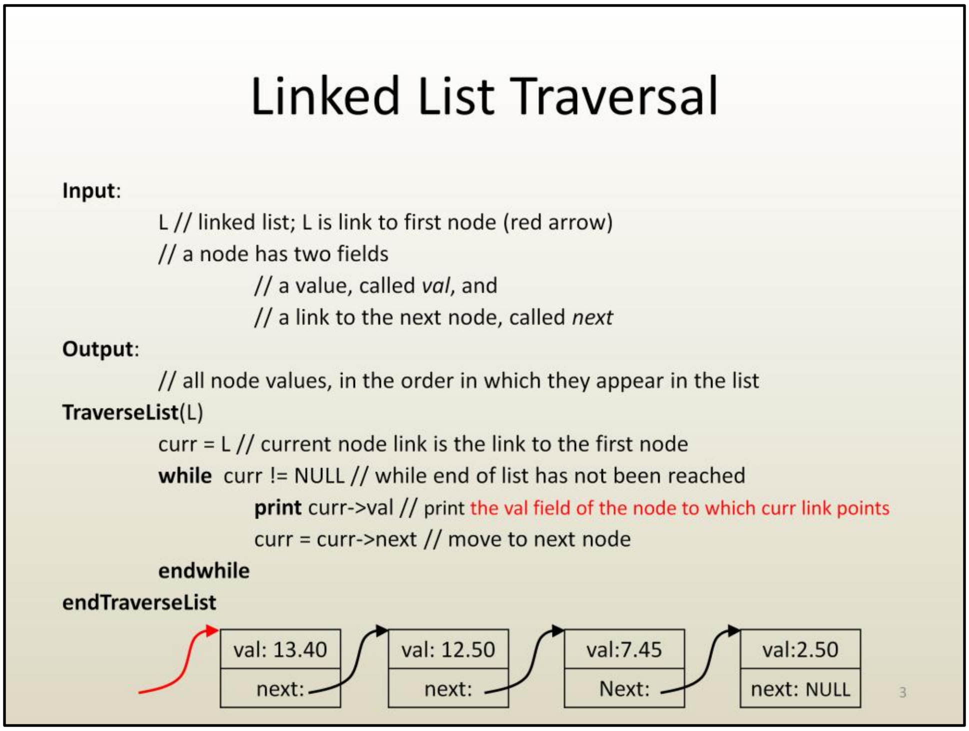
- A generic algorithm for visiting all list nodes
- Can be specialized for many goals
  - Find minimum / maximum
  - Search for given value
  - Count number of occurrences of given value
- Building block for more complex algorithms
  - Sorting

2

The ability to traverse the data structure is an essential component of almost any algorithm. You have to be able to get to the data in order to process it.

Whereas in the case of arrays traversing the data structure is straight forward and it is achieved with one or several nested for loops, in the case of irregular data structures traversals have to follow the more complex topology of the data structure.

Let’s first have a look at traversing a linked list. Once we’ll know how to traverse a linked list we’ll have an excellent start for devising algorithms that achieve more complex data processing.



Here is the basic linked list traversal algorithm. In this example the linked list nodes store a transaction value. The traversal algorithm prints out all transaction values.

The input to the algorithm is the linked list, specified with the link to the first node. In addition to the payload, i.e. the transaction value, a node also stores a link to the next node.

The output of the algorithm is the printout of all transaction values.

The algorithm is called TraverseList.

The variable curr is a link to the current node. Initially, curr is a link to the first node, i.e. L.

Then the list is traversed with a while loop which keep going “while” the link curr is not null. The notation “!=“ means not equal.

The body of the while loop prints out the value of the current node and moves to the next node. Moving to the next node is done by assigning to curr the link to the next node.

The notation “LINK”->”FIELD” gives access to the field called “FIELD” of the node pointed to by the link “LINK”.

For the last element, the next link is NULL. This sets curr to NULL, the condition curr != NULL is false, and the while loop stops.

Trace

curr	curr->val	curr->next	Print Out
L	13.40	A_1	13.40
A1	12.50	A2	12.50
A2	7.45	A3	7.45
A3	2.50	NULL	2.50
NULL			

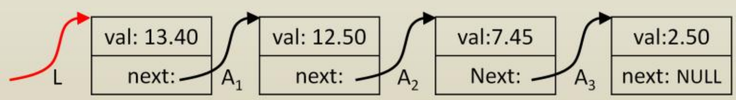
Input:

L // linked list; L is link to first node (red arrow)
// a node has two fields
// a value, called *val*, and
// a link to the next node, called *next*

Output:

// all node values, in the order in which they appear in the list

TraverseList(L)
curr = L // current node link is the link to the first node
while curr != NULL // while end of list has not been reached
print curr->val // print the val field of the node
curr = curr->next // move to next node
endwhile
endTraverseList



4

Here is a trace of the algorithm on the linked list shown.

Initially, curr is set to L.

Curr is not NULL, thus the while loop is executed the first time. The value of curr, which is 13.40, is printed out, and curr is set to the link to the next element, i.e. the link to the second element. We reach the end of the body of the while loop, so we have to go check the condition to see if we need to execute the while loop one more time.

Curr is not NULL, thus the while loop is executed for the second time. We print out 12.50, set curr to the link to the next element, and check the while loop condition again.

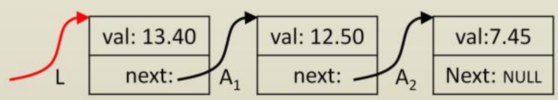
Curr is not NULL , thus the while loop is executed for the third time. We print out 7.45, we set curr to the link to the next element, and check the while loop condition again.

Curr is still not NULL, thus the while loop is executed for the fourth time. We print out 2.50, we set curr to the link to the next element which is NULL, and check the while loop condition again.

Curr is NULL, thus curr != NULL is false, so the while loop is not executed anymore and we reach the end of the algorithm marked by endTraverseList.

Insert node in sorted list

```
Input:
        L // linked list with (val, next) nodes
        V // value to be inserted
Output:
        // linked list with new node storing V, sorted
InsertSortedList(L, V)
        N = link to new node // create new node
        N->val = V // val of new node is input value V; don't know next yet
        curr = L // current node link is the link to the first node
        prev = NULL // link to previous node, initially null
        while  curr != NULL // while end of list has not been reached
                if curr->val < V then // if the current value is smaller than V
                        break // stop loop; insert btw prev and curr
                endif
                prev = curr // move to next node, update prev & curr
                curr = curr->next
        endwhile
        N->next = curr
        if prev != NULL then
                prev->next = N
        else
                L = N
        endif
endInsertSortedList
```



Here is how the generic linked list traversal algorithm can be specialized to allow inserting a node in a sorted list. The traversal is needed in order to find the insertion point.

The input to the algorithm is the linked list and the value to be inserted V.

The output is the linked list with a new node that stores V.

The first two instructions of the algorithm create the new node and set its value. Notice that at this point the new node is not linked into the list, i.e. it has not yet been inserted. In other words, its next field has not yet been specified, nor is there a list node that has its next field be a link to the new node.

In order to find the insertion point, we need to find the two nodes in between which the new node has to be inserted. For this we need to keep track of the current **pair** of nodes, and not just of the current node. We do this using a second variable, called prev, which stores a link to the previous node. As before, curr is a variable that stores the link to the current node.

Initially curr is a link to the first node, prev is NULL, since there is no node preceding the first node.

The while loop iterates “while” the end of the list hasn’t been reached.

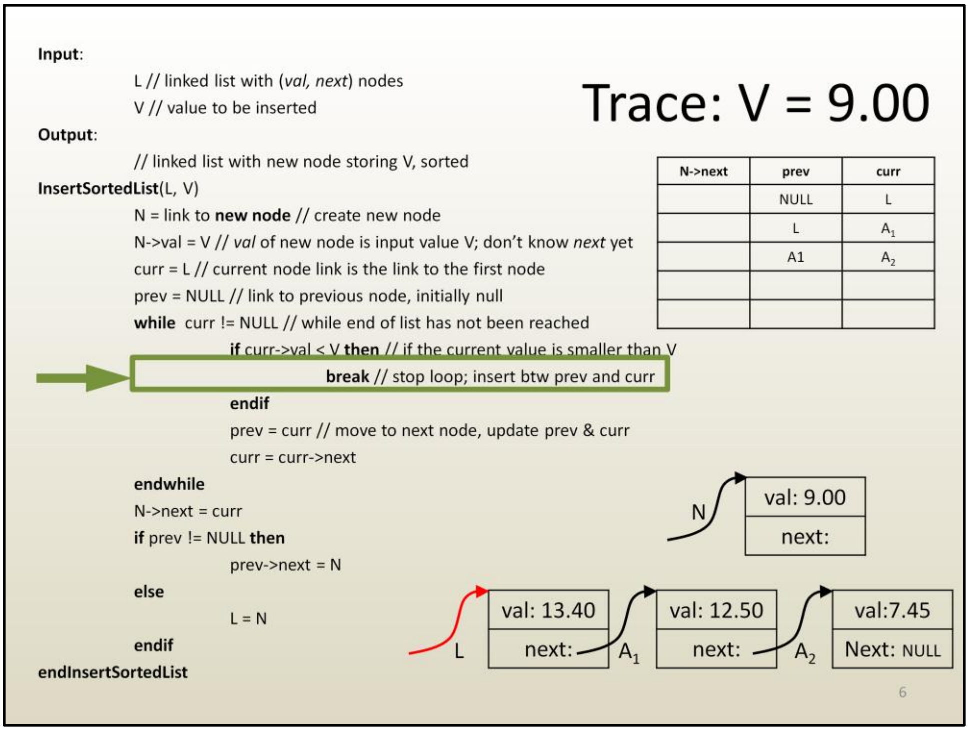
If the value stored at the current node is less than the value we need to insert, we have found the insertion point (i.e. after prev and before curr), and we stop the while loop iteration immediately with the “break” instruction. Once that break is executed, execution resumes after the while loop.

If the value stored at the current node is greater or equal than V, we continue iterating over the linked list: previous node is advanced to current node, current node is advanced to next node. Now prev and curr point to the next pair of nodes.

After the while loop we need to link in the new node. The next field of the new node has to be set (A), and the next field of an existing node has to link to the new node (B).

(A) After the while loop, curr will store a link to the node following the new node. If the while loop was terminated by the break instruction, curr will link to a node with a transaction smaller than the new transaction. If the while loop was terminated because the end of the list was reached, curr will be NULL, which is also correct: the new transaction is the smallest in the list, therefore it has to be inserted at the end of the list, and there is no node after the new node, hence the next field of the new node has to be NULL.

(B) If the new transaction is larger than any of the transactions in the list, the new node has to be first in list, and the prev variable is NULL. Consequently there are two cases: prev is not NULL, case in which the next field of prev has to be set to the new node, and prev is NULL, when there is no node with a next link to the new node, but the link L to the first node identifying the list has to be updated to a link to the new node N.



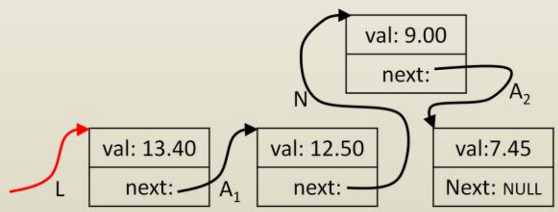
Let's trace the insertion algorithm for V equals 9.00.

The condition of the if statement curr->val < V is true when curr points to the last node, i.e. when curr is A2. Again, the break statement terminates the loop immediately and execution resumes after "endwhile".

Trace: V = 9.00

```
Input:
        L // linked list with (val, next) nodes
        V // value to be inserted
Output:
        // linked list with new node storing V, sorted
InsertSortedList(L, V)
        N = link to new node // create new node
        N->val = V // val of new node is input value V; don't know next yet
        curr = L // current node link is the link to the first node
        prev = NULL // link to previous node, initially null
        while  curr != NULL // while end of list has not been reached
                if curr->val < V then // if the current value is smaller than V
                        break // stop loop; insert btw prev and curr
                endif
                prev = curr // move to next node, update prev & curr
                curr = curr->next
        endwhile
        N->next = curr
        if prev != NULL then
                prev->next = N
        else
                L = N
        endif
endInsertSortedList
```

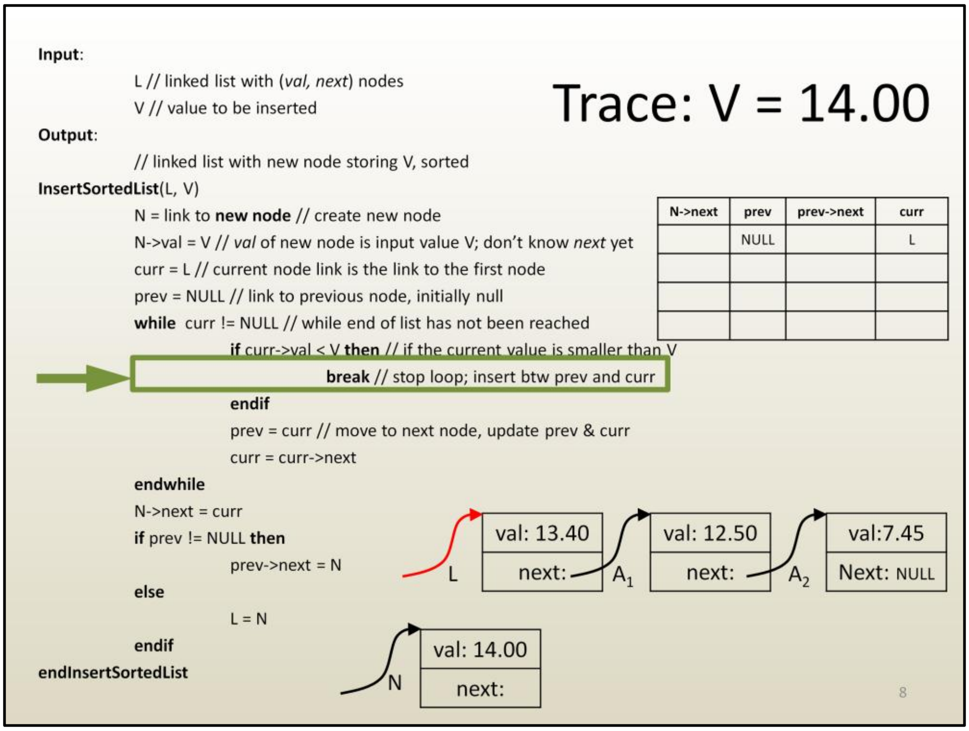
N->next	prev	prev->next	curr
	NULL		L
	L		A_1
	A_1		A_2
A_2		N	



7

The next field of the new node is set to A2, which is correct.

Prev is not NULL, thus the then branch is taken. The next field of the prev node is set to the new node, which is also correct, effectively linking in the new node.



Here is the trace for a case when the new transaction is largest than all transactions, meaning that it is larger than the first transaction in the list.

The breaking condition is met right away, with curr being the first element and prev being NULL.

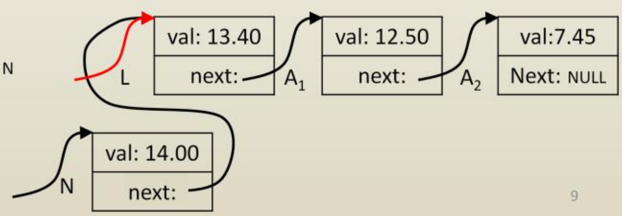
Input:
L // linked list with (*val, next*) nodes
V // value to be inserted

Trace: V = 14.00

Output:
// linked list with new node storing V, sorted

```
InsertSortedList(L, V)
    N = link to new node // create new node
    N->val = V // val of new node is input value V; don't know next yet
    curr = L // current node link is the link to the first node
    prev = NULL // link to previous node, initially null
    while curr != NULL // while end of list has not been reached
        if curr->val < V then // if the current value is smaller than V
            break // stop loop; insert btw prev and curr
        endif
        prev = curr // move to next node, update prev & curr
        curr = curr->next
    endwhile
    N->next = curr
    if prev != NULL then
        prev->next = N
    else
        L = N
    endif
endInsertSortedList
```

N->next	prev	prev->next	curr
	NULL		L
L			



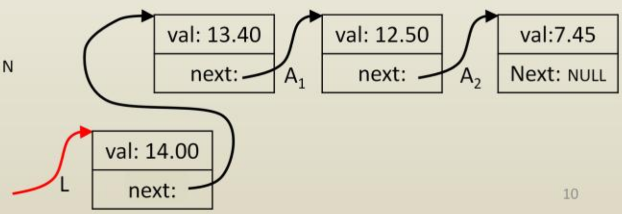
9

The next field of the new node is the first node, which is correct.

Trace: V = 14.00

```
Input:
        L // linked list with (val, next) nodes
        V // value to be inserted
Output:
        // linked list with new node storing V, sorted
InsertSortedList(L, V)
        N = link to new node // create new node
        N->val = V // val of new node is input value V; don't know next yet
        curr = L // current node link is the link to the first node
        prev = NULL // link to previous node, initially null
        while  curr != NULL // while end of list has not been reached
                if curr->val < V then // if the current value is smaller than V
                        break // stop loop; insert btw prev and curr
                endif
                prev = curr // move to next node, update prev & curr
                curr = curr->next
        endwhile
        N->next = curr
        if prev != NULL then
                prev->next = N
        else
                L = N
        endif
endInsertSortedList
```

N->next	prev	prev->next	curr
	NULL		L
L			



10

Prev is NULL thus there is no node whose next node should be the new node. In other words, the new node is the first node. However, the link to the new node, which identifies the list, has to be updated to be a link to the first node.

```
Input:
        L // linked list with (val, next) nodes
        V // value to be inserted
Output:
        // linked list with new node storing V, sorted
InsertSortedList(L, V)
        N = link to new node // create new node
        N->val = V // val of new node is input value V; don't know next yet
        curr = L // current node link is the link to the first node
        prev = NULL // link to previous node, initially null
        while  curr != NULL // while end of list has not been reached
                if curr->val < V then // if the current value is smaller than V
                        break // stop loop; insert btw prev and curr
                endif
                prev = curr // move to next node, update prev & curr
                curr = curr->next
        endwhile
        N->next = curr
        if prev != NULL then
                prev->next = N
        else
                L = N
        endif
endInsertSortedList
```

iClicker Q

What do the highlighted instructions achieve?

A. If the new node is last they set its next link to NULL.

B. If the new node is first they set the first node link to the new node.

C. If the new node is not first they set the next link of the preceding node to the new node.

D. A, B, and C

E. A and C

Binary tree traversal

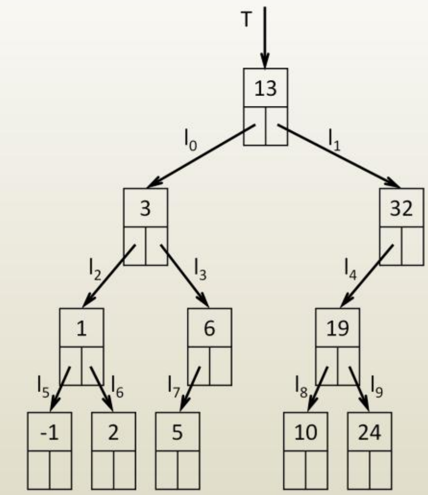
- A generic algorithm for visiting all tree nodes
- Can be specialized for many goals
- Start at root
- Visit child nodes
- Visit children of children
- Some bookkeeping needed
  - Which nodes have been visited?
  - Which are yet to be visited?

12

Let’s now devise an algorithm for traversing a binary tree. Unlike for the list, when traversing a binary tree, there are many forks in the road. At each internal node one could go left or right. Some bookkeeping is needed in order to traverse the entire binary tree.

Binary tree traversal

```
Input:
    T // link to root of binary tree
        // a node stores (val, left, right)
Output:
    // print all values stored in the tree
TraverseBT(T)
    L = empty list of links to nodes
    L = InsertFirstList(L, T)
    while L is not empty
        curr = ExtractFirstNode(L)
        if curr != NULL then
            print curr->val
            L = InsertFirstList(L, curr->left)
            L = InsertFirstList(L, curr->right)
        endif
    endwhile
endTraverseBT
```



Here is the binary tree traversal algorithm. The binary tree node has three fields: a value called val, which is the payload, and two links to children nodes, called left and right, which model the tree structure.

The algorithm makes use of an auxiliary data structure: a linked list L that stores links to nodes that are yet to be visited. Whereas in the previous example the linked list stored transaction values, now the payload of the linked list node is a link, a link to tree nodes (not to be confused with the next link to list nodes).

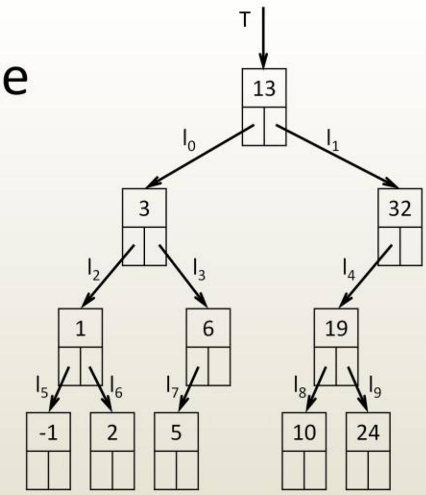
The traversal starts at the root. The link to the root is inserted into the linked list L.

Then the traversal is achieved using a while loop that keeps iterating while L is not empty.

Each loop iteration extracts the first link in L, and, if it is a valid link, the value stored is printed and the links to the left and right children are inserted in L. The insertion places the new element first in the list. If, on the other hand, the link extracted from L is NULL, nothing is done (i.e. there is no else branch).

Trace

Input:
T // link to root of binary tree
// a node stores (*val, left, right*)
Output:
// print all values stored in the tree
TraverseBT(T)
L = empty list of links to nodes
L = **InsertFirstList**(L, T)
while L is not empty
curr = **ExtractFirstNode**(L)
if curr != NULL **then**
print curr->val
L = **InsertFirstList**(L, curr->left)
L = **InsertFirstList**(L, curr->right)
endif
endwhile
endTraverseBT



	L	Curr	Print Out
1	T		
2	empty	T	
3	empty	T	13
4	l_1, l_0	T	
5	l_0	l_1	

14

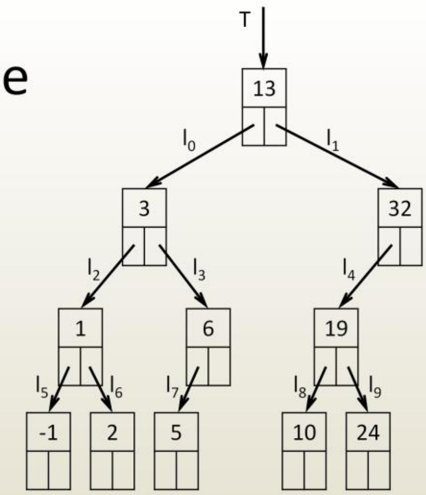
Let’s trace the traversal algorithm on the binary tree given to the right.

At first, T is the only link in L. Consequently ExtractFirstNode returns T, which is assigned to curr.

L is not empty, so the while loop body is executed for the first time.
- Curr is not null, it is an actual link to a node, so the value stored by the node, 13, is printed out
- the left and right links are inserted into L.

Now L stores two links, I1 and I0, in this order. L is not empty thus the while loop body is executed for a second time. The first link stored in L is extracted, which is I1.

Trace



Input:
T // link to root of binary tree
// a node stores (*val, left, right*)
Output:
// print all values stored in the tree
TraverseBT(T)
L = empty list of links to nodes
L = **InsertFirstList**(L, T)
while L is not empty
curr = **ExtractFirstNode**(L)
if curr != NULL **then**
print curr->val
L = **InsertFirstList**(L, curr->left)
L = **InsertFirstList**(L, curr->right)
endif
endwhile
endTraverseBT

	L	Curr	Print Out
5	l_0	l_1	32
6	NULL, l_4, l_0	l_1	
7	l_4, l_0	NULL	
8	l_9, l_8, l_0	l_4	19
9	l_8, l_0	l_9	24

15

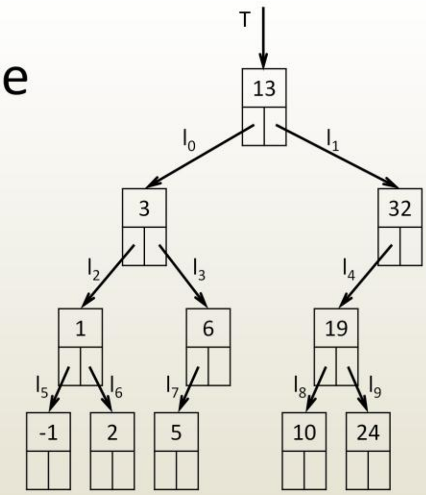
it is a valid link, it's value, 32, is printed out, and the children links I4 and NULL are inserted in L.

L is not empty, another while loop iteration, the NULL link is extracted, nothing else is done.

L is not empty, another while loop iteration, I4 is extracted, valid link, 19 is printed out, I8 and I9 are inserted into L, and so on.

Trace

Input:
  T // link to root of binary tree
    // a node stores (*val, left, right*)
Output:
  // print all values stored in the tree
TraverseBT(T)
  L = empty list of links to nodes
  L = **InsertFirstList**(L, T)
  while L is not empty
    curr = **ExtractFirstNode**(L)
    if curr != NULL **then**
      print curr->val
      L = **InsertFirstList**(L, curr->left)
      L = **InsertFirstList**(L, curr->right)
    endif
  endwhile
endTraverseBT

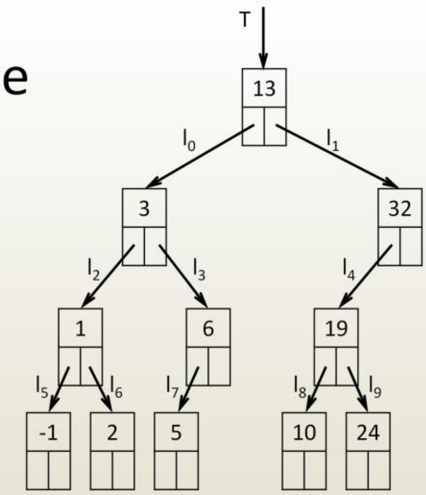


	L	Curr	Print Out
9	NULL, NULL, l_8, l_0	l_9	24
10	NULL, l_8, l_0	NULL	
11	l_8, l_0	NULL	
12	NULL, NULL, l_0	l_8	10
13	NULL, l_0	NULL	

16

Trace

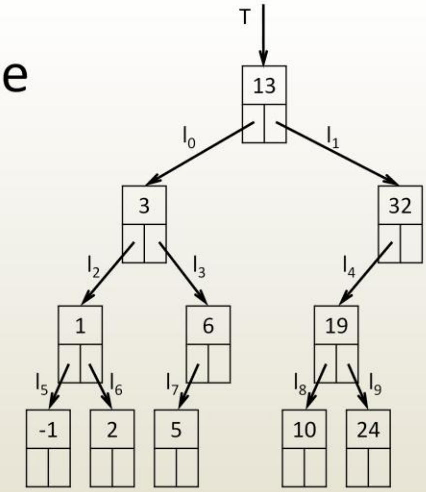
Input:
T // link to root of binary tree
// a node stores (*val, left, right*)
Output:
// print all values stored in the tree
TraverseBT(T)
L = empty list of links to nodes
L = **InsertFirstList**(L, T)
while L is not empty
curr = **ExtractFirstNode**(L)
if curr != NULL **then**
print curr->val
L = **InsertFirstList**(L, curr->left)
L = **InsertFirstList**(L, curr->right)
endif
endwhile
endTraverseBT



	L	Curr	Print Out
13	NULL, l_0	NULL	
14	l_0	NULL	
15	l_3, l_2	l_0	3
16	NULL, l_7, l_2	l_3	6
17	NULL, NULL, l_2	l_7	5

17

Trace




```
Input:
    T // link to root of binary tree
        // a node stores (val, left, right)
Output:
    // print all values stored in the tree
TraverseBT(T)
    L = empty list of links to nodes
    L = InsertFirstList(L, T)
    while L is not empty
        curr = ExtractFirstNode(L)
        if curr != NULL then
            print curr->val
            L = InsertFirstList(L, curr->left)
            L = InsertFirstList(L, curr->right)
        endif
    endwhile
endTraverseBT
```

	L	Curr	Print Out
17	NULL, NULL, l_2	l_7	5
18	l_6, l_5	l_2	1
19	NULL, NULL, l_5	l_6	2
19	NULL, NULL	l_5	-1
21	empty		

18

Once L is empty, the algorithm terminates. All nodes were traversed, all their info was printed out.

iClicker Q

```
Input:
    T // link to root of binary tree
        // a node stores (val, left, right)
Output:
    // print all values stored in the tree
TraverseBT(T)
    L = empty list of links to nodes
    L = InsertFirstList(L, T)
    while L is not empty
        curr = ExtractFirstNode(L)
        if curr != NULL then
            print curr->val
            L = InsertFirstList(L, curr->left)
            L = InsertFirstList(L, curr->right)
        endif
    endwhile
endTraverseBT
```

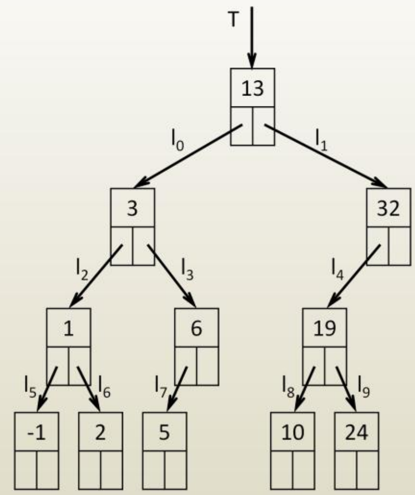
```
Input:
    T // link to root of binary tree
        // a node stores (val, left, right)
Output:
    // print all values stored in the tree
TraverseBTM(T)
    L = empty list of links to nodes
    L = InsertFirstList(L, T)
    while L is not empty
        curr = ExtractFirstNode(L)
        print curr->val
        if curr->left != NULL
            L = InsertFirstList(L, curr->left)
        if curr->right != NULL
            L = InsertFirstList(L, curr->right)
    endwhile
endTraverseBT
```

Is the modified algorithm **TraverseBTM** correct?

A. Yes B. No

Binary tree node count

```
Input:
    T // link to root of binary tree
          // a node stores (val, left, right)
Output:
    n // count of nodes
CountNodesBT(T)
    L = empty list of links to nodes
    L = InsertList(L, T)
    n= 0
    while L is not empty
        curr = ExtractFirstNode(L)
        if curr != NULL then
            n= n+ 1 // old "print curr->val"
            L = InsertList(L, curr->left)
            L = InsertList(L, curr->right)
        endif
    endwhile
    return n
endCountNodesBT
```



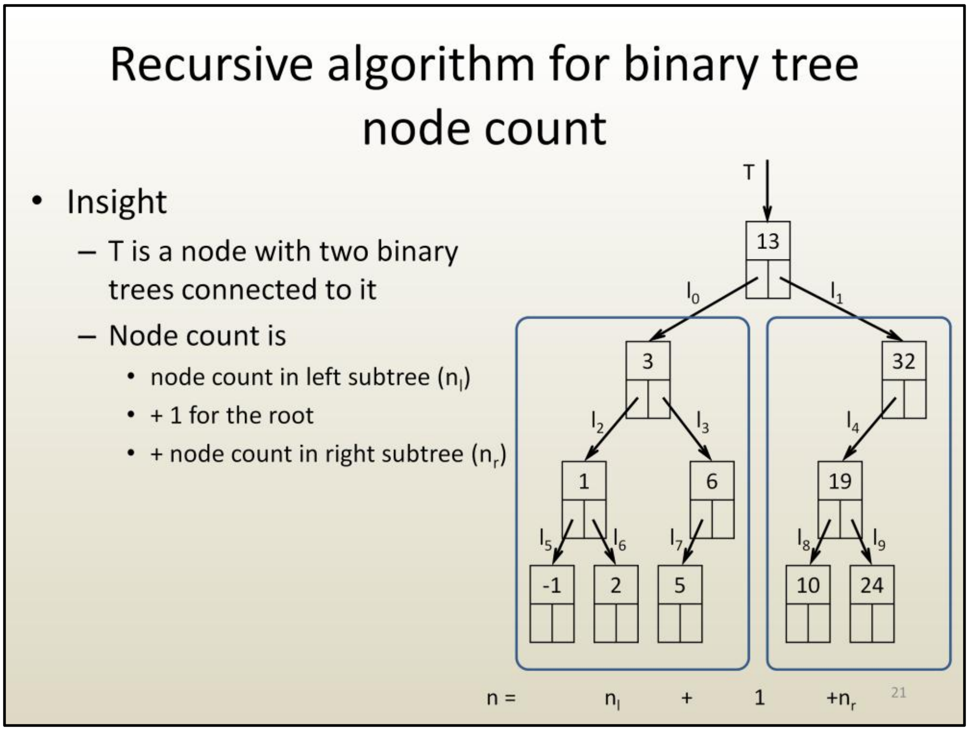
20

Let's modify the basic binary tree traversal algorithm seen earlier in order to count the nodes in the tree.

The output of the algorithm is an integer n that will store the number of nodes in the tree.

Initially, n is 0. As nodes are visited, instead of printing out the value stored by the node, we now increment n by 1, i.e. we add one more to the node count.

At the end, after the while loop terminates, n is returned (e.g. returned to the main algorithm that used this node counting algorithm as a sub-algorithm, or printed out, etc.). The algorithm will count the nodes correctly because all nodes are traversed.



If you look at the binary tree data structure closer you notice that it is a recursive data structures. This means that a binary tree is a node with a binary tree as its left child and a binary tree as its right child.

This fact enables a simple, intuitive approach for traversing a binary tree, and for binary tree algorithms in general.

For example, in the case of counting nodes, one can find the number of nodes in the tree by adding the number of nodes in the left sub-tree to the number of nodes in the right sub-tree and by adding one for the root. This approach to counting nodes is called recursive.

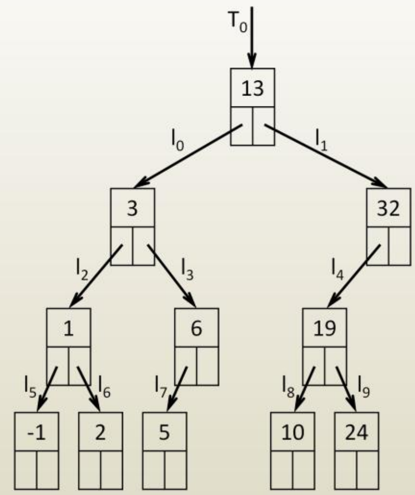
Notice how we define counting nodes, using counting nodes.

“**Counting nodes** in tree is done by **counting nodes** in left sub-tree and by **counting nodes** in right sub-tree”

Recursive algorithm for binary tree node count

Input:
  T // link to root of binary tree
Output:
  // count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR(T->left) + 1 + CountBTR (T->right)
endCountBTR
```



22

Here is the pseudocode of the recursive node counting algorithm.

If the tree is empty, there are no nodes, thus return 0.

If the tree is not empty, count how many nodes (a) there are in the tree rooted at the left child, count how many nodes there are in the tree rooted at the right child (b), and return a + 1 + b, where 1 stands for the root node.

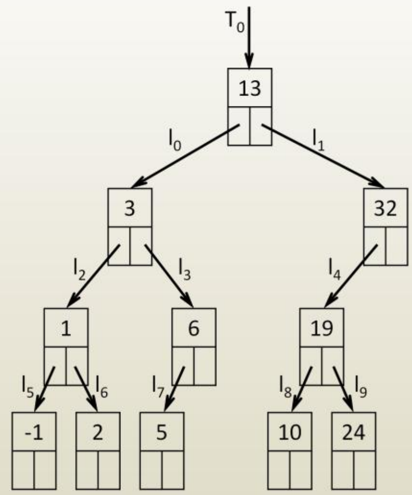
Notice how short and elegant the recursive variant of node count is.

Also notice that there is no explicit loop. The loop is implicit. We keep calling CountBTR until we reach leafs.

Recursive algorithm for binary tree node count

Input:
  T // link to root of binary tree
Output:
  // count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR(T->left) + 1 + CountBTR (T->right)
endCountBTR
```



Recursive because CountBTR calls CountBTR

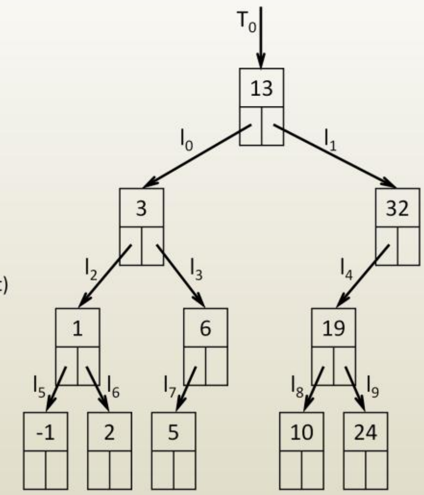
Recursion is a very powerful paradigm for designing algorithms

23

The CountBTR is a recursive algorithm because it uses itself as a sub-algorithm.

Trace

Input:
T // link to root of binary tree
Output:
// count of nodes
CountBTR(T)
if T == NULL
return 0
endif
return CountBTR (T->left) + 1 + **CountBTR** (T->right)
endCountBTR



Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	CountBTR(l_2)+1+CountBTR(l_3)
CountBTR(l_2)	CountBTR(l_5)+1+CountBTR(l_6)
CountBTR(l_5)	CountBTR(NULL)+1+CountBTR(NULL)
CountBTR(NULL)	0

24

Let’s trace the recursive node counting algorithm on the binary tree to the right.

There are many “nested” sub-algorithm calls and the trace has to keep track of them. By nested sub-algorithm calls we mean making another sub-algorithm call before the previous sub-algorithm call completes its work and returns.

The first call to CountBTR is done on the entire tree. T is not NULL, it is T_0 thus we do not return 0 (“then” keyword omitted here).

As the line “return CountBTR(T->left) + 1 + CountBTR(T->right)” is executed, CountBTR(T->left) is executed first and everything else is put on hold until CountBTR(T->left) returns a value (i.e. the number of nodes in the left sub-tree).

Since T is T0, T->left is I0, so CountBTR(T->left) means CountBTR(I0).

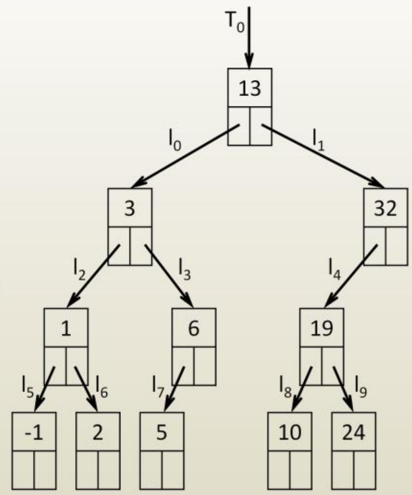
I0 is not NULL, the if condition is false, and another CountBTR(T->left) call is made, which now corresponds to CountBTR(I2).

Similarly the CountBTR(I5) call is made, and then CountBTR(T->left) call is made, which now corresponds to CountBTR(NULL). Since now T is NULL, 0 is returned and no other calls are made.

Trace

Input:
 T // link to root of binary tree
Output:
 // count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + CountBTR (T->right)
endCountBTR
```



Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	CountBTR(l_2)+1+CountBTR(l_3)
CountBTR(l_2)	CountBTR(l_5)+1+CountBTR(l_6)
CountBTR(l_5)	0+1+0

25

Similarly, CountBTR(T->right) for T equal to I5 returns a 0. At this point we are ready to complete the return statement

return CountBTR(I5->left) + 1 + CountBTR(I5->right), since both CountBTR calls have returned.

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

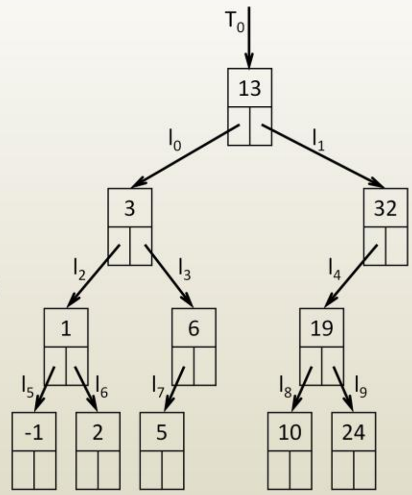
if T == NULL

return 0

endif

return CountBTR (T->left) + 1 + **CountBTR** (T->right)

endCountBTR



Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	CountBTR(l_2)+1+CountBTR(l_3)
CountBTR(l_2)	CountBTR(l_5)+1+CountBTR(l_6)
CountBTR(l_5)	1

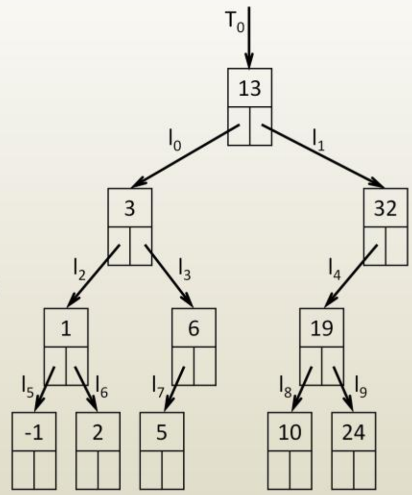
26

We now essentially know that the number of nodes in the tree with root I5 is 1. The call CountBTR(I5) returns 1.

Before CountBTR(I2) can return, we need to execute CountBTR(I6).

Trace

Input:
    T // link to root of binary tree
Output:
    // count of nodes
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + **CountBTR** (T->right)
endCountBTR



Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	CountBTR(l_2)+1+CountBTR(l_3)
CountBTR(l_2)	1+1+CountBTR(l_6)
CountBTR(l_6)	CountBTR(NULL)+1+CountBTR(NULL)

27

Like before, I6 is not null, thus we count the number of nodes in its left sub-tree + 1 + the number of nodes in its right subtree.

Trace

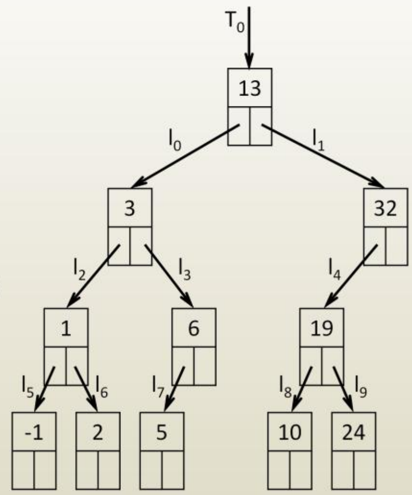
Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + **CountBTR** (T->right)
endCountBTR



Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	CountBTR(l_2)+1+CountBTR(l_3)
CountBTR(l_2)	1+1+CountBTR(l_6)
CountBTR(l_6)	0+1+0

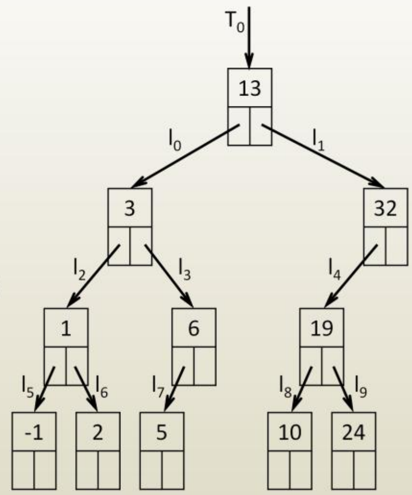
28

This amounts to 0+1+0, which is 1.

Trace

Input:
T // link to root of binary tree
Output:
// count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + CountBTR (T->right)
endCountBTR
```



Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	CountBTR(l_2)+1+CountBTR(l_3)
CountBTR(l_2)	1+1+CountBTR(l_6)
CountBTR(l_6)	1

29

Now we know that CountBTR(I6) is 1, and we return that value to the CountBTR(I2) call.

Trace

Input:

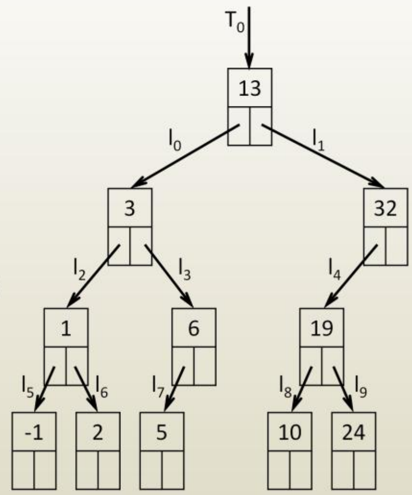
T // link to root of binary tree

Output:

// count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + CountBTR (T->right)
endCountBTR
```

Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	CountBTR(l_2)+1+CountBTR(l_3)
CountBTR(l_2)	1+1+1



30

CountBTR(I2) evaluates to 3, which is correct: the tree with root link I2 has 3 nodes.

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

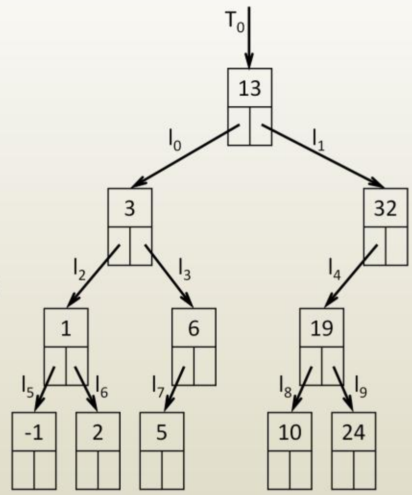
if T == NULL

return 0

endif

return CountBTR (T->left) + 1 + **CountBTR** (T->right)

endCountBTR



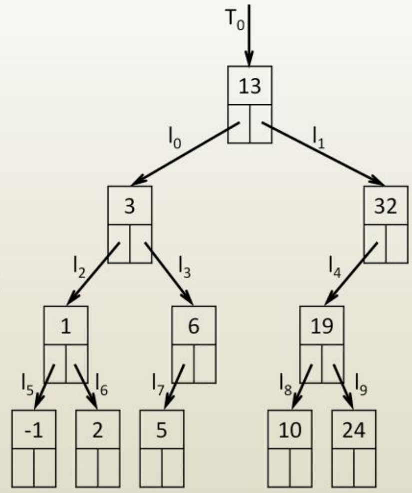
Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	CountBTR(l_2)+1+CountBTR(l_3)
CountBTR(l_2)	3

31

Trace

Input:
T // link to root of binary tree
Output:
// count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + CountBTR (T->right)
endCountBTR
```



Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	3+1+CountBTR(l_3)

32

Before we can finalize CountBTR(I0) we need to evaluate CountBTR(I3).

Trace

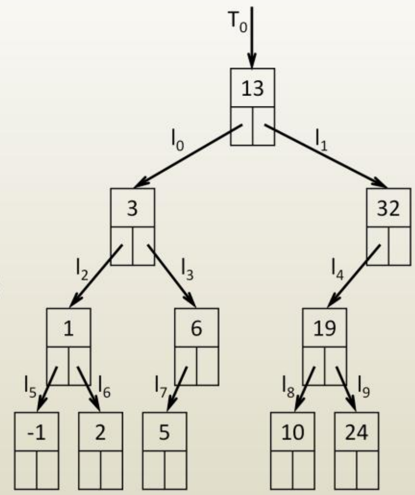
Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + **CountBTR** (T->right)
endCountBTR



Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	3+1+CountBTR(l_3)
CountBTR(l_3)	CountBTR(l_7)+1+CountBTR(NULL)

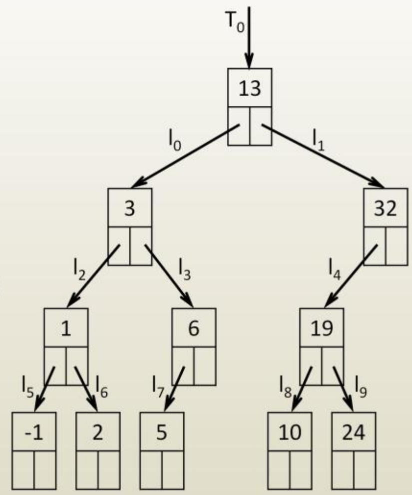
33

This implies evaluating CountBTR(l7).

Trace

Input:
T // link to root of binary tree
Output:
// count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + CountBTR (T->right)
endCountBTR
```



Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	3+1+CountBTR(l_3)
CountBTR(l_3)	CountBTR(l_7)+1+CountBTR(NULL)
CountBTR(l_7)	0+1+0

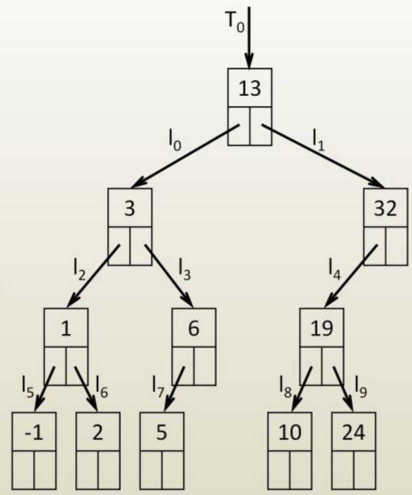
34

Which is a leaf node and evaluates to 1.

Trace

Input:
T // link to root of binary tree
Output:
// count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + CountBTR (T->right)
endCountBTR
```



Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	3+1+CountBTR(l_3)
CountBTR(l_3)	1+1+CountBTR(NULL)

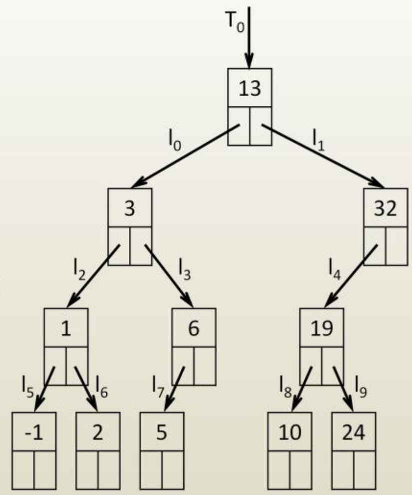
35

The node rooted at l3 doesn’t have a right child.

Trace

Input:
T // link to root of binary tree
Output:
// count of nodes
CountBTR(T)
if T == NULL
return 0
endif
return CountBTR (T->left) + 1 + **CountBTR** (T->right)
endCountBTR

Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	3+1+CountBTR(l_3)
CountBTR(l_3)	1+1+0

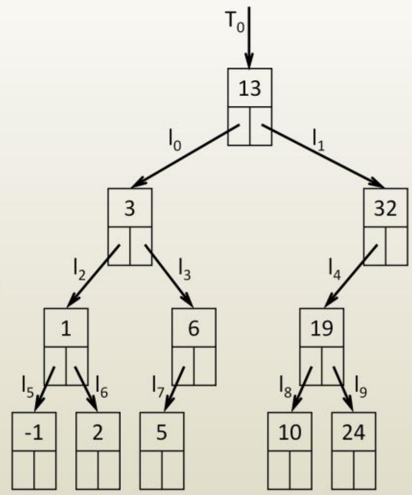


36

So there are 0 nodes in the right sub-tree, for an overall node count of 2.

Trace

Input:
T // link to root of binary tree
Output:
// count of nodes
CountBTR(T)
if T == NULL
return 0
endif
return CountBTR (T->left) + 1 + **CountBTR** (T->right)
endCountBTR



Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	3+1+2

37

Now we know how many nodes there are in the sub-tree with root link l0: 6.

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

if T == NULL

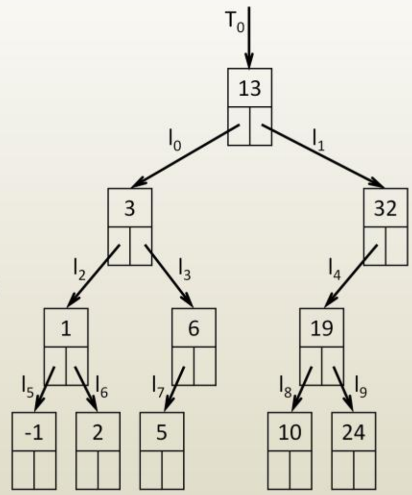
return 0

endif

return CountBTR (T->left) + 1 + **CountBTR** (T->right)

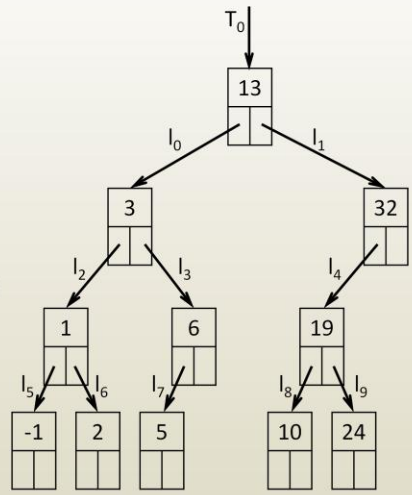
endCountBTR

Call	Return
CountBTR(T_0)	CountBTR(l_0)+1+CountBTR(l_1)
CountBTR(l_0)	6



Trace

Input:
  T // link to root of binary tree
Output:
  // count of nodes
CountBTR(T)
  if T == NULL
    return 0
  endif
  return CountBTR (T->left) + 1 + **CountBTR** (T->right)
endCountBTR



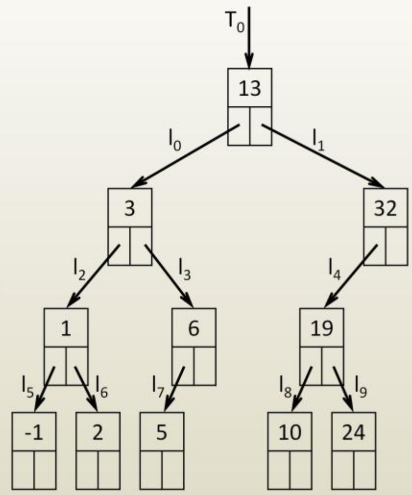
Call	Return
CountBTR(T_0)	6+1+CountBTR(I_1)

39

Before we know how many nodes there are in the tree, we need to also know how many nodes are in the right sub-tree of I1.

Trace

Input:
  T // link to root of binary tree
Output:
  // count of nodes
CountBTR(T)
  if T == NULL
    return 0
  endif
  return CountBTR (T->left) + 1 + **CountBTR** (T->right)
endCountBTR



Call	Return
CountBTR(T_0)	6+1+CountBTR(l_1)
CountBTR(l_1)	CountBTR(l_4)+1+CountBTR(NULL)

40

This implies first knowing how many nodes there are in the left sub-tree of I1, which has root link I4.

Trace

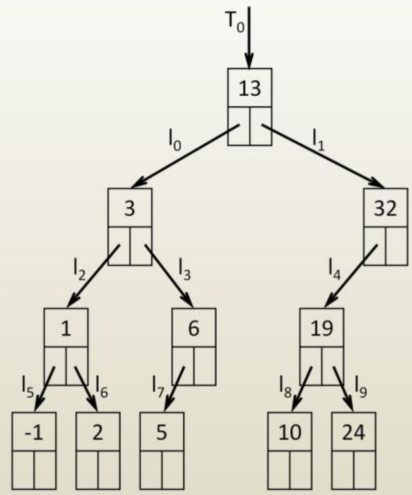
Input:

T // link to root of binary tree

Output:

// count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + CountBTR (T->right)
endCountBTR
```



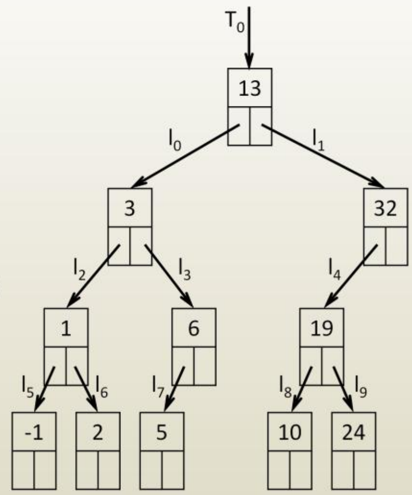
Call	Return
CountBTR(T_0)	6+1+CountBTR(l_1)
CountBTR(l_1)	CountBTR(l_4)+1+CountBTR(NULL)
CountBTR(l_4)	CountBTR(l_8)+1+CountBTR(l_9)

41

This implies first knowing the number of nodes in the sub-tree with root link I8.

Trace

Input:
T // link to root of binary tree
Output:
// count of nodes
CountBTR(T)
if T == NULL
return 0
endif
return CountBTR (T->left) + 1 + **CountBTR** (T->right)
endCountBTR



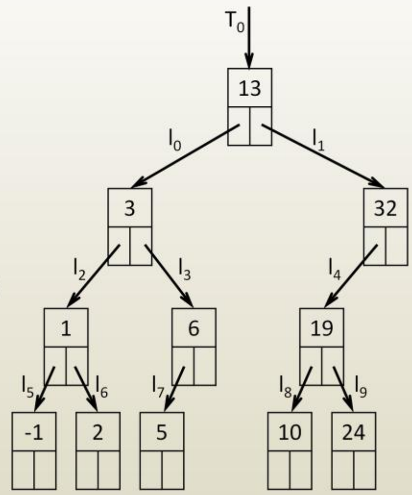
Call	Return
CountBTR(T_0)	6+1+CountBTR(l_1)
CountBTR(l_1)	CountBTR(l_4)+1+CountBTR(NULL)
CountBTR(l_4)	CountBTR(l_8)+1+CountBTR(l_9)
CountBTR(l_8)	0+1+0

42

Which is 1 since it’s a leaf node (CountBTR on left and right children links are each 0 because links are NULL).

Trace

Input:
T // link to root of binary tree
Output:
// count of nodes
CountBTR(T)
if T == NULL
return 0
endif
return CountBTR (T->left) + 1 + **CountBTR** (T->right)
endCountBTR



Call	Return
CountBTR(T_0)	6+1+CountBTR(l_1)
CountBTR(l_1)	CountBTR(l_4)+1+CountBTR(NULL)
CountBTR(l_4)	1+1+CountBTR(l_9)
CountBTR(l_9)	0+1+0

43

To finalize the node count at I4 we need to count the number of nodes in the right sub-tree, with root link I9.

Trace

Input:

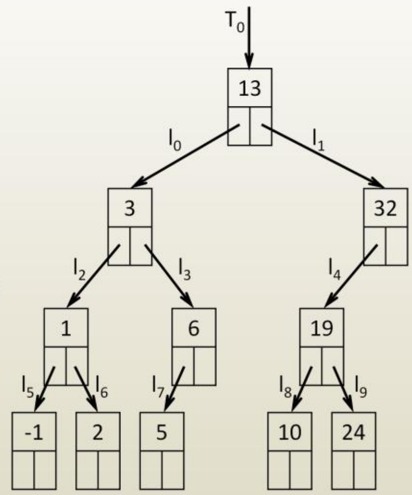
T // link to root of binary tree

Output:

// count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + CountBTR (T->right)
endCountBTR
```

Call	Return
CountBTR(T_0)	6+1+CountBTR(l_1)
CountBTR(l_1)	CountBTR(l_4)+1+CountBTR(NULL)
CountBTR(l_4)	1+1+1



44

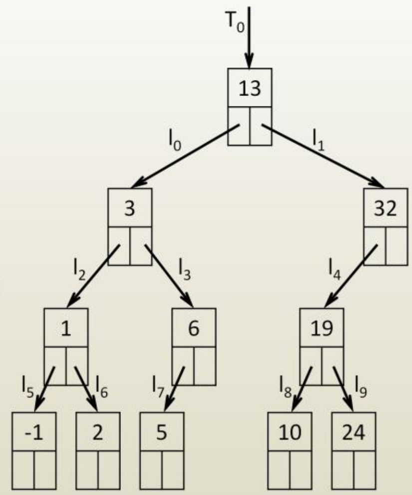
That’s also 1, for an overall number of nodes in the sub-tree with root link I4 of 3.

Trace

Input:
  T // link to root of binary tree
Output:
  // count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + CountBTR (T->right)
endCountBTR
```

Call	Return
CountBTR(T_0)	6+1+CountBTR(l_1)
CountBTR(l_1)	3+1+0

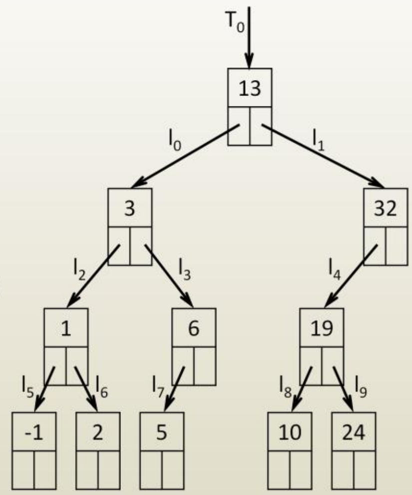


45

Now we know the number of nodes in sub-tree with root link of I1, 4 (this sub-tree doesn’t have a right child hence the 0 in 3+1+0).

Trace

Input:
T // link to root of binary tree
Output:
// count of nodes
CountBTR(T)
if T == NULL
return 0
endif
return CountBTR (T->left) + 1 + **CountBTR** (T->right)
endCountBTR



Call	Return
CountBTR(T_0)	6+1+4

46

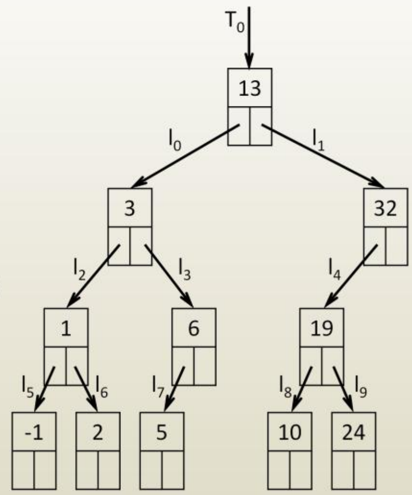
Now we know the number of nodes in the left and right sub-trees of the original tree, thus we know the overall number of nodes: 11.

Trace

Input:
T // link to root of binary tree
Output:
// count of nodes

```
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + CountBTR (T->right)
endCountBTR
```

Call	Return
CountBTR(T_0)	11



47

The essential component of this node counting algorithm is the recursive traversal of the binary tree.

iClicker Q

```
Input:
    T // link to root of binary tree
Output:
    // count of nodes
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR (T->left) + 1 + CountBTR (T->right)
endCountBTR
```

```
Input:
    T // link to root of binary tree
Output:
    // count of nodes
CountBTRM(T)
    if T->left != NULL then
        leftCount = CountBTR (T->left)
    else
        leftCount = 0
    if T->right != NULL then
        rightCount = CountBTR (T->right)
    else
        rightCount = 0
    return leftCount + 1 + rightCount
endCountBTR
```

Is the modified algorithm **CountBTRM** correct?

A. Yes B. No

Elements of recursive algorithm

- Recursive call
  - Algorithm calls itself on subsets of the input data
  - One ore more recursive calls
    - For binary tree we had two recursive calls, one for each child
- Termination condition
  - At some point recursion has to stop
  - For example, don’t go beyond leafs
    - Leafs don’t have children, referring to children leafs causes algorithm to crash
- Work to be done before, between, and after recursive calls
  - For example, print “(“, print string at current node, print “)”

Recursion is a very powerful paradigm for designing algorithms.

Recursive algorithms have the following elements:

There is always a recursive call, when the algorithm uses itself as a sub-algorithm.

There is always a termination condition. Don’t make recursive calls forever, at some point stop making recursive calls.

The actual work is interspersed with the recursive calls. What needs to get done before the recursive calls, between the recursive calls, after the recursive calls?

Recursion problem solving paradigm

- You don't solve the problem directly
- Split the problem until it becomes trivial
- Compute solution to problem by combining solutions of sub-problems
- Examples
  - Counting nodes in binary trees
    - No node means 0
    - Number of nodes in tree is number of nodes in left subtree plus 1 plus number of nodes in right subtree

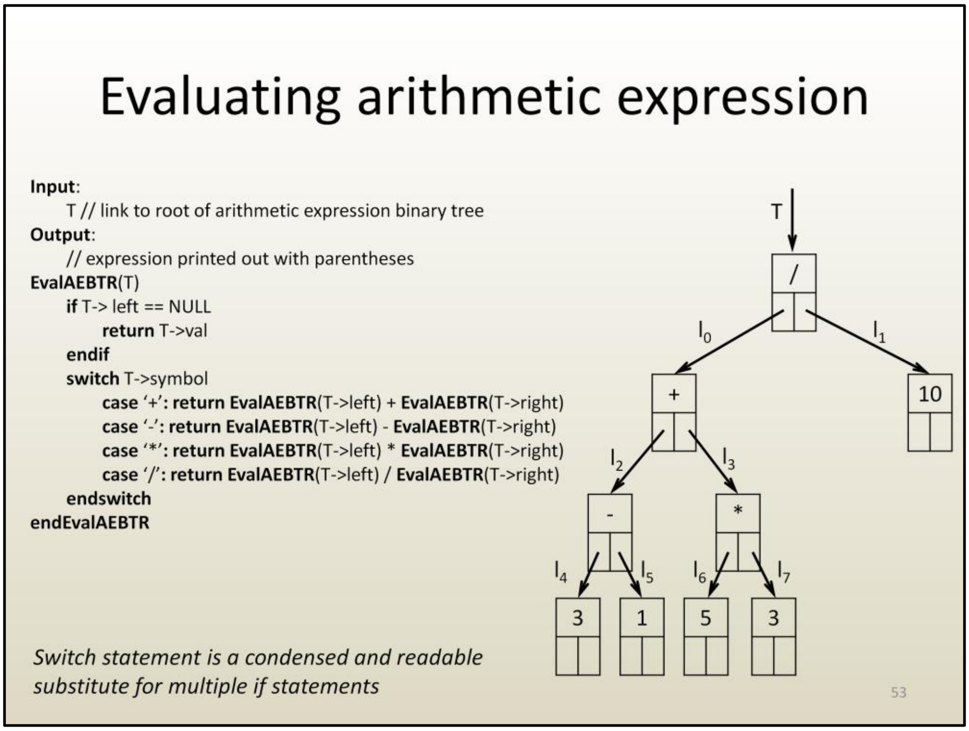
The beauty of recursion is that you don't solve the problem directly. Instead, the problem is split into smaller/simpler sub-problems, until the sub-problems become trivial. Then the solution of the overall problem is obtained by combining the solutions of the sub-problems.

Recursion problem solving paradigm

- You don’t solve the problem directly
- Split the problem until it becomes trivial
- Compute solution to problem by combining solutions of sub-problems
- Examples
  - Counting nodes in binary trees
  - Evaluating arithmetic expression
    - Value of leaf is number stored at leaf
    - Value for tree rooted at internal node is obtained by applying operation stored at internal node to the values of the left and right subtrees

Recursion problem solving paradigm

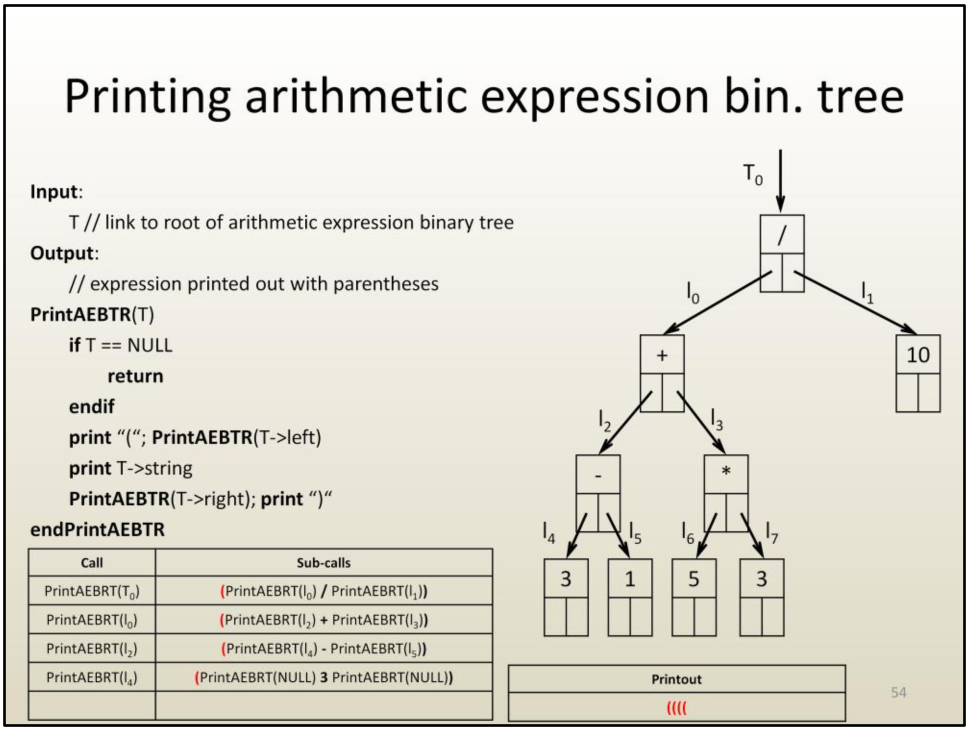
- You don't solve the problem directly
- Split the problem until it becomes trivial
- Compute solution to problem by combining solutions of sub-problems
- Examples
  - Counting nodes in binary trees
  - Evaluating arithmetic expression
  - Printing arithmetic expression
    - Printout of leaf is string at leaf
    - Printout for internal node is
      - Open parenthesis,
      - Followed by printout for left subtree,
      - Followed by string at current node,
      - Followed by printout for right subtree
      - Followed by closed parenthesis



Here is the pseudocode algorithm for evaluating an arithmetic expression stored in a binary tree.

A switch statement is used, which is a condensed way of writing multiple if statements.

The algorithm works as follows:
- if the current tree is a leaf, return the value stored by the leaf; this makes sense because the value of an operand, e.g. 3, is the actual operand;
- if the current tree is not a leaf, find out what operation the node corresponds to, and return the value of the left sub-tree operated with the value of the right sub-tree; the actual return can of course only be done once the left and right sub-tree recursive calls return; the various alternatives for the symbol (i.e. the operator) are tested using a switch statement; only one branch of the switch statement is taken.



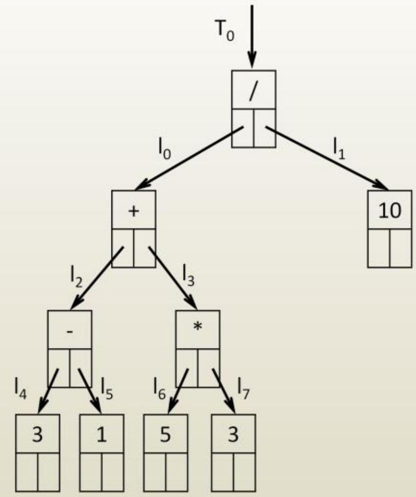
The recursive traversal of binary trees can be specialized to achieve other tasks.

In this example the recursive traversal is used to print out the arithmetic expression stored in a binary tree.

At high level:
- nothing is printed out on a NULL link,
- for a valid link node, one prints out
    - an open parenthesis,
    - the left operand (which could be an expression in and by itself),
    - the operator,
    - the right operand (which also could be an expression in and by itself),
    - and finally a closed parenthesis.

First, several open parentheses are printed out, as the algorithm digs deeper and deeper in the sub-expressions of the original expression.

Printing arithmetic expression bin. tree



Input:
T // link to root of arithmetic expression binary tree
Output:
// expression printed out with parentheses

```
PrintAEBTR(T)
    if T == NULL
        return
    endif
    print "("; PrintAEBTR(T->left)
    print T->string
    PrintAEBTR(T->right); print ")"
endPrintAEBTR
```

Call	Sub-calls
PrintAEBRT(T_0)	**(**PrintAEBRT(l_0) **/** PrintAEBRT(l_1)**)**
PrintAEBRT(l_0)	**(**PrintAEBRT(l_2) **+** PrintAEBRT(l_3)**)**
PrintAEBRT(l_2)	**(**PrintAEBRT(l_4) **-** PrintAEBRT(l_5)**)**
PrintAEBRT(l_4)	**(3)**

Printout
((((3)

55

Then a leaf is reached, with link I4.

Nothing is printed for its left link, which is NULL.

The string at the leaf is then printed out, i.e. 3.

Nothing is printed for its right link, which is also NULL.

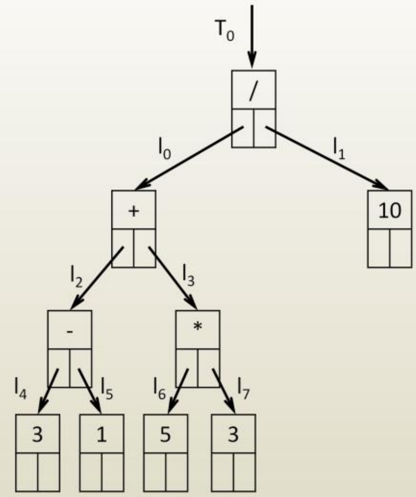
A closed parenthesis is also printed.

Now we go back and continue the printing of the node with root I2.

Printing arithmetic expression bin. tree

Input:
    T // link to root of arithmetic expression binary tree
Output:
    // expression printed out with parentheses

```
PrintAEBTR(T)
    if T == NULL
        return
    endif
    print "("; PrintAEBTR(T->left)
    print T->string
    PrintAEBTR(T->right); print ")"
endPrintAEBTR
```



Call	Sub-calls
PrintAEBRT(T_0)	**(PrintAEBRT(l_0) / PrintAEBRT(l_1))**
PrintAEBRT(l_0)	**(PrintAEBRT(l_2) + PrintAEBRT(l_3))**
PrintAEBRT(l_2)	**((3) - PrintAEBRT(l_5))**

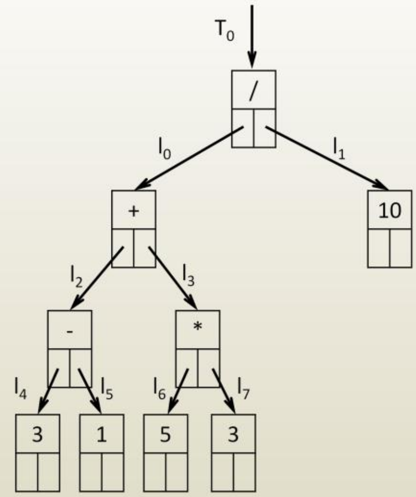
Printout
((((3)-

56

We are done with the left link I4 of I2, so we print the string at the current node which is the minus sign, and now we have to print the expression for the right link, I5.

Printing arithmetic expression bin. tree

```
Input:
    T // link to root of arithmetic expression binary tree
Output:
    // expression printed out with parentheses
PrintAEBTR(T)
    if T == NULL
        return
    endif
    print "("; PrintAEBTR(T->left)
     print T->string
    PrintAEBTR(T->right); print ")"
endPrintAEBTR
```



Call	Sub-calls
PrintAEBRT(T_0)	**(**PrintAEBRT(l_0) **/** PrintAEBRT(l_1)**)**
PrintAEBRT(l_0)	**(**PrintAEBRT(l_2) **+** PrintAEBRT(l_3)**)**
PrintAEBRT(l_2)	**((3) -** PrintAEBRT(l_5)**)**
PrintAEBRT(l_5)	**(**PrintAEBRT(NULL) **1** PrintAEBRT(NULL)**)**

Printout
((((3)-

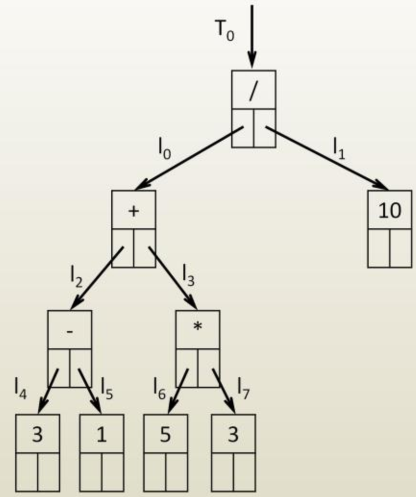
57

Like before, this implies printing out the expressions for the sub-trees.

Printing arithmetic expression bin. tree

Input:
T // link to root of arithmetic expression binary tree
Output:
// expression printed out with parentheses

```
PrintAEBTR(T)
    if T == NULL
        return
    endif
    print "("; PrintAEBTR(T->left)
    print T->string
    PrintAEBTR(T->right); print ")"
endPrintAEBTR
```



Call	Sub-calls
PrintAEBRT(T_0)	**(PrintAEBRT(l_0) / PrintAEBRT(l_1))**
PrintAEBRT(l_0)	**(PrintAEBRT(l_2) + PrintAEBRT(l_3))**
PrintAEBRT(l_2)	**((3) - PrintAEBRT(l_5))**
PrintAEBRT(l_5)	**(1)**

Printout
(((3)-(1)

58

Since these links are NULL, nothing is printed out for the left and right links of I5. The printout for I5 is (1).

Printing arithmetic expression bin. tree

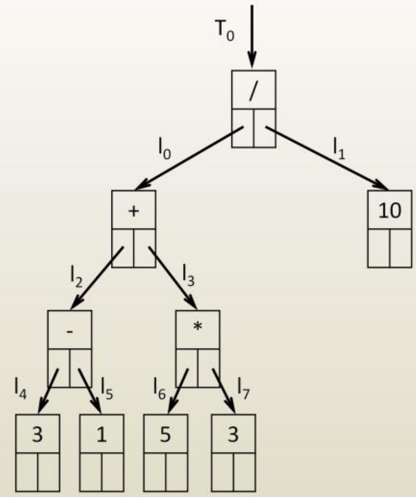
Input:

T // link to root of arithmetic expression binary tree

Output:

// expression printed out with parentheses

```
PrintAEBTR(T)
    if T == NULL
        return
    endif
    print "("; PrintAEBTR(T->left)
     print T->string
    PrintAEBTR(T->right); print ")"
endPrintAEBTR
```



Call	Sub-calls
PrintAEBRT(T_0)	**(**PrintAEBRT(l_0) **/** PrintAEBRT(l_1)**)**
PrintAEBRT(l_0)	**(**PrintAEBRT(l_2) **+** PrintAEBRT(l_3)**)**
PrintAEBRT(l_2)	**((3) – (1))**

Printout
((((3)-(1))

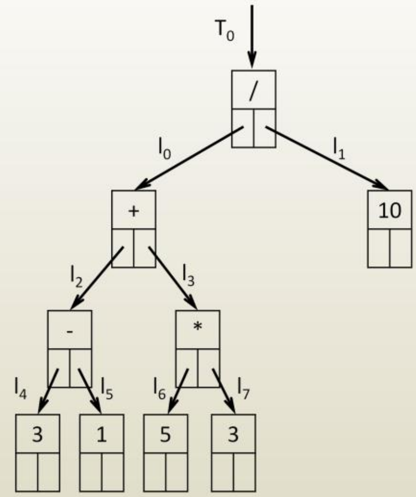
59

Now we are done with the print out for I2, it is ((3)-(1)).

Printing arithmetic expression bin. tree

Input:
T // link to root of arithmetic expression binary tree
Output:
// expression printed out with parentheses

```
PrintAEBTR(T)
    if T == NULL
        return
    endif
    print "("; PrintAEBTR(T->left)
     print T->string
    PrintAEBTR(T->right); print ")"
endPrintAEBTR
```



Call	Sub-calls
PrintAEBRT(T_0)	**(**PrintAEBRT(I_0) **/** PrintAEBRT(I_1)**)**
PrintAEBRT(I_0)	**(((3) – (1)) +** PrintAEBRT(I_3)**)**

Printout
((((3)-(1))+

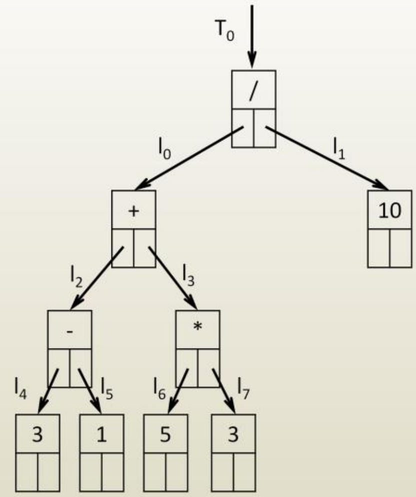
60

To finalize the printout for I0 we need to finalize the printout for I3.

Printing arithmetic expression bin. tree

Input:
T // link to root of arithmetic expression binary tree
Output:
// expression printed out with parentheses

```
PrintAEBTR(T)
    if T == NULL
        return
    endif
    print "("; PrintAEBTR(T->left)
     print T->string
    PrintAEBTR(T->right); print ")"
endPrintAEBTR
```



Call	Sub-calls
PrintAEBRT(T_0)	**(PrintAEBRT(l_0) / PrintAEBRT(l_1))**
PrintAEBRT(l_0)	**(((3) – (1)) + ((5) * (3)))**

Printout
((((3)-(1))+((5)*(3)))

61

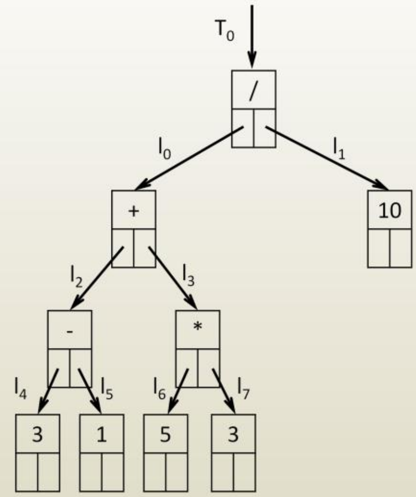
Like for I2, this amounts to a recursive printout for the tree with root link I3.

Printing arithmetic expression bin. tree

Input:
T // link to root of arithmetic expression binary tree
Output:
// expression printed out with parentheses

```
PrintAEBTR(T)
    if T == NULL
        return
    endif
    print "("; PrintAEBTR(T->left)
    print T->string
    PrintAEBTR(T->right); print ")"
endPrintAEBTR
```

Call	Sub-calls
PrintAEBRT(T_0)	(((3) – (1)) +((5) * (3))) / PrintAEBRT(l_1))



Printout
(((3)-(1))+((5)*(3)))/

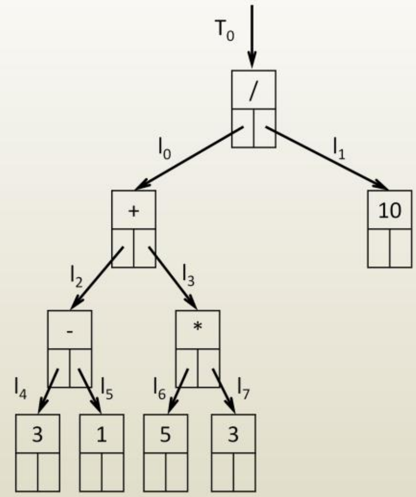
62

Before the entire printout is finalized, one has to print out the expression for the right tree of T0, which has root link I1.

Printing arithmetic expression bin. tree

Input:
T // link to root of arithmetic expression binary tree
Output:
// expression printed out with parentheses

PrintAEBTR(T)
  if T == NULL
    return
  endif
  print "("; **PrintAEBTR**(T->left)
  print T->string
  PrintAEBTR(T->right); **print** ")"
endPrintAEBTR



Call	Sub-calls
PrintAEBRT(T_0)	**(((3) – (1)) +((5) * (3))) /** PrintAEBRT(l_1)**)**
PrintAEBRT(l_1)	**(10)**

Printout
((((3)-(1))+((5)*(3)))/(10)

63

This is a simple leaf, which once printed out,

Printing arithmetic expression bin. tree

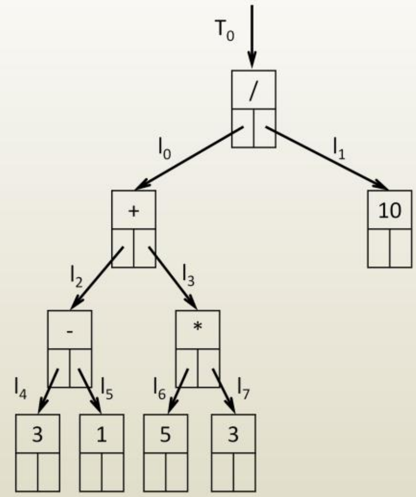
Input:

T // link to root of arithmetic expression binary tree

Output:

// expression printed out with parentheses

PrintAEBTR(T)
    if T == NULL
        return
    endif
    print "("; **PrintAEBTR**(T->left)
    print T->string
    PrintAEBTR(T->right); **print** ")"
endPrintAEBTR



Call	Sub-calls
PrintAEBRT(T_0)	(((3) – (1)) +((5) * (3))) / (10))

Correct, but parentheses around operands are annoying

Printout
((((3)-(1))+((5)*(3)))/(10))

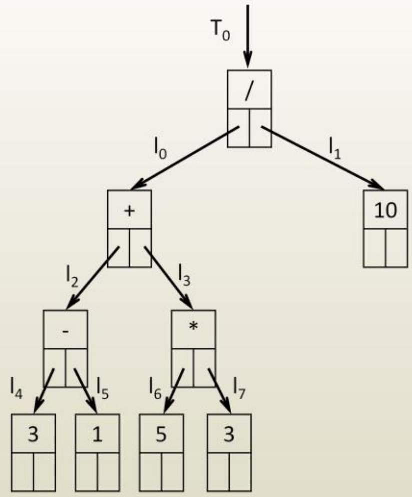
64

allows us to print out the final right parenthesis.

The printout is fine, but having parentheses around operands is unnecessary.

Printing arithmetic expression bin. tree

```
Input:
    T // link to root of arithmetic expression binary tree
Output:
    // expression printed out with parentheses
PrintAEBTR(T)
    if T == NULL
        return
    endif
    if T->left != NULL print "("
    PrintAEBTR(T->left)
     print T->string
    PrintAEBTR(T->right)
     if T->left != NULL print ")"
endPrintAEBTR
```



Call	Printout
PrintAEBRT(T_0)	(((3 – 1) + (5 * 3)) / 10)

Don't print parentheses for leaf nodes (i.e. operands)
It is assumed that there are no nodes with one child

65

These redundant parentheses can be avoided by printing out an open or a closed parenthesis only if the current node is not a leaf.

Here this is done by adding the condition that T->left should not be NULL, as a leaf node would.

With these additional instructions in place, the printout does not have the extraneous parentheses.