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

Linked List Traversal

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

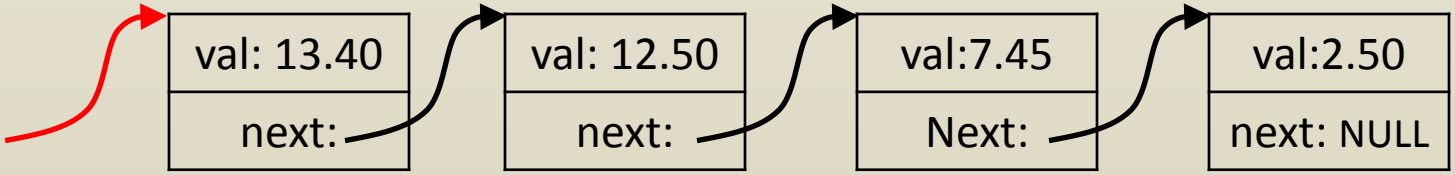
while curr != NULL // while end of list has not been reached

print curr->val // print the val field of the node to which curr link points

curr = curr->next // move to next node

endwhile

endTraverseList



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*

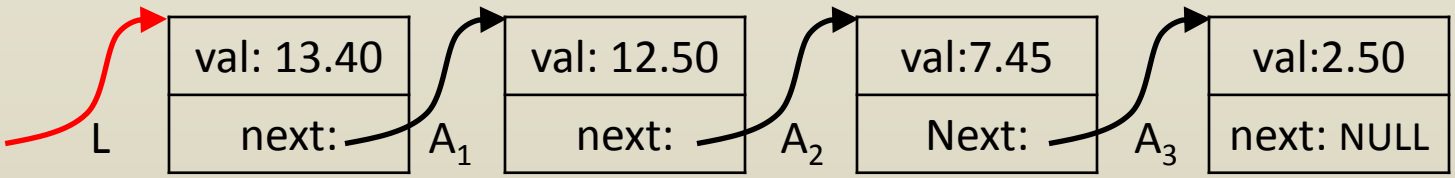
Output:

// 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



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**

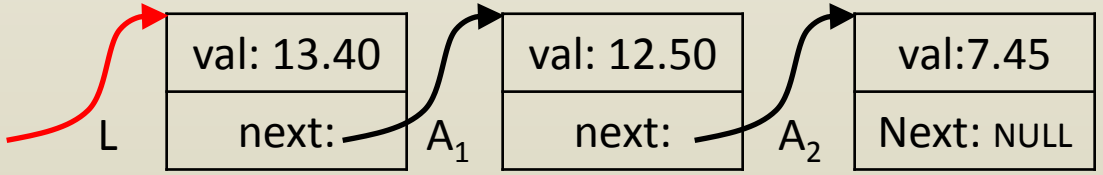
prev->next = N

else

L = N

endif

endInsertSortedList



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

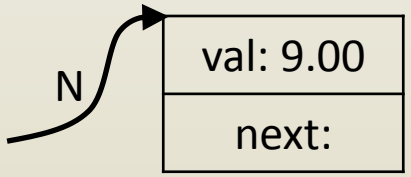
else

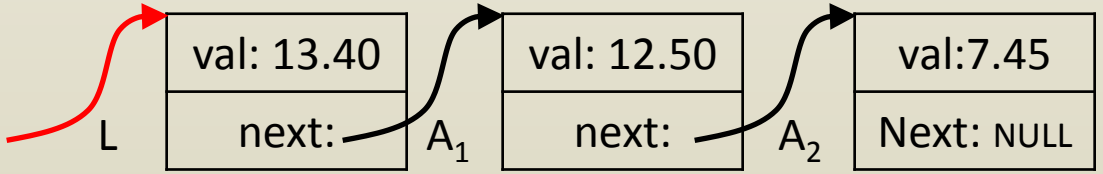
L = N

endif

endInsertSortedList

N->next	prev	curr
	NULL	L
	L	A_1
	A1	A_2





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

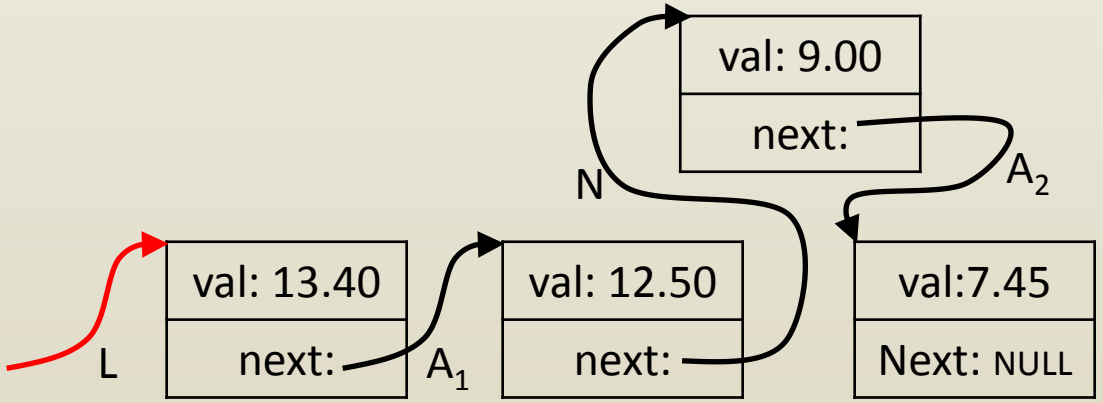
else

L = N

endif

endInsertSortedList

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



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

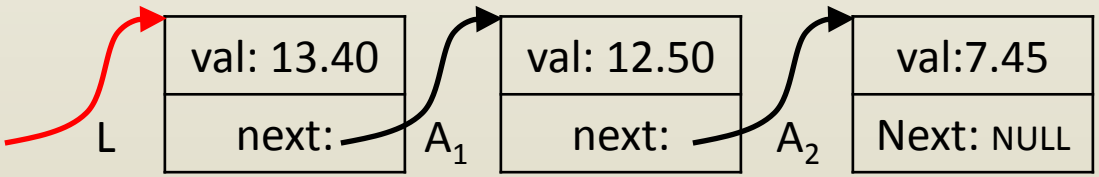
else

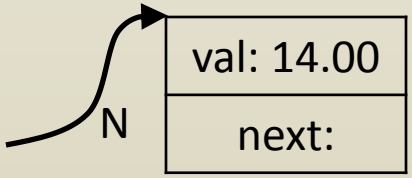
L = N

endif

endInsertSortedList

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





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

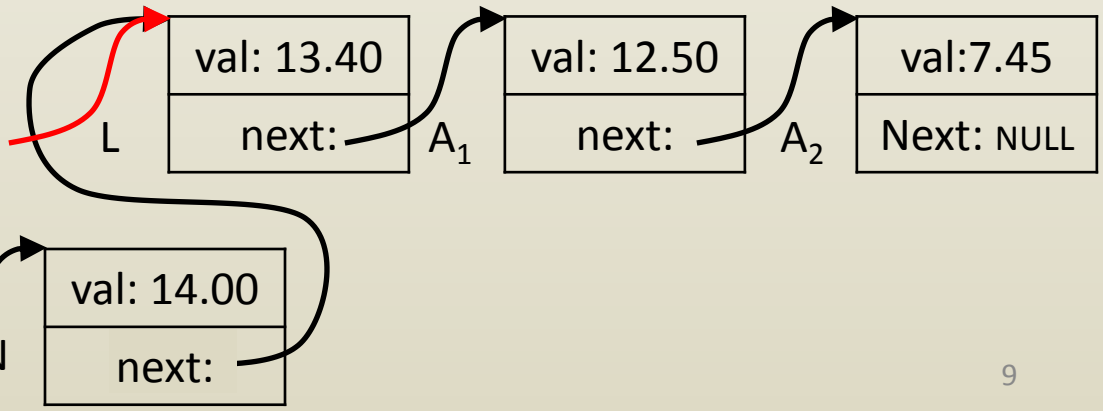
else

L = N

endif

endInsertSortedList

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



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

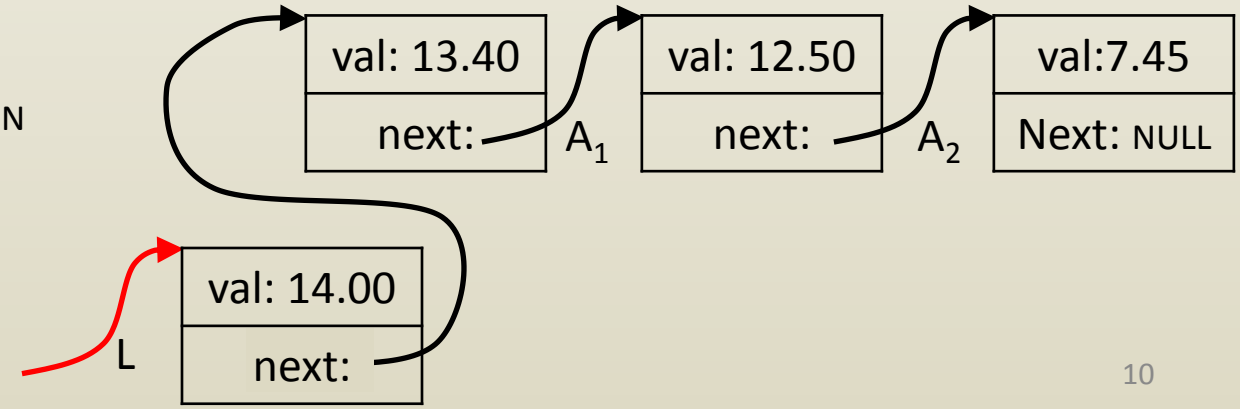
else

L = N

endif

endInsertSortedList

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



```
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

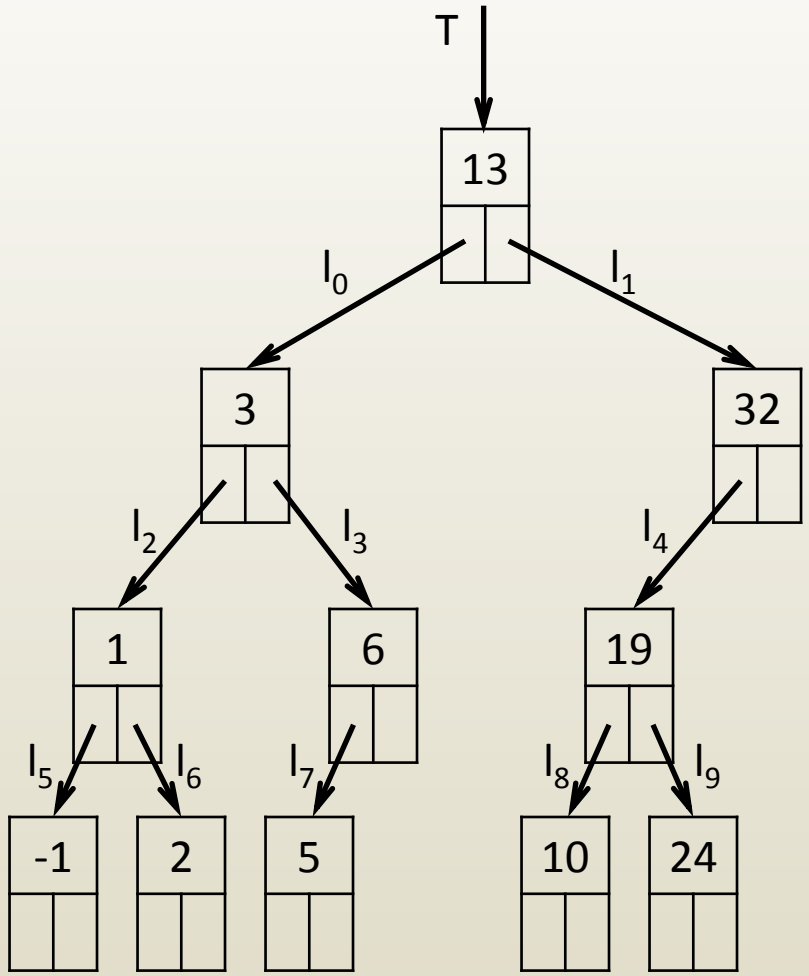
E. A and C

Binary tree traversal

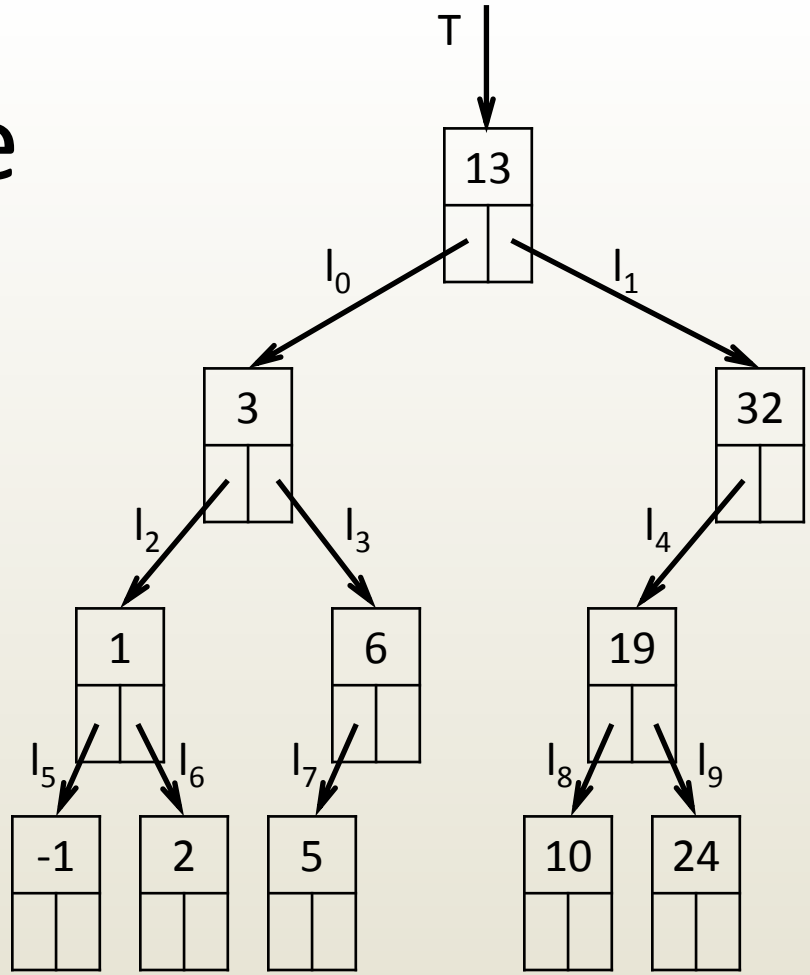
- A generic algorithm for visiting all tree nodes
- Can be specialized for many goals
- Start at root
- Visit child nodes
- Visit children of children
- Some bookkeeping needed
  - Which nodes have been visited?
  - Which are yet to be visited?

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



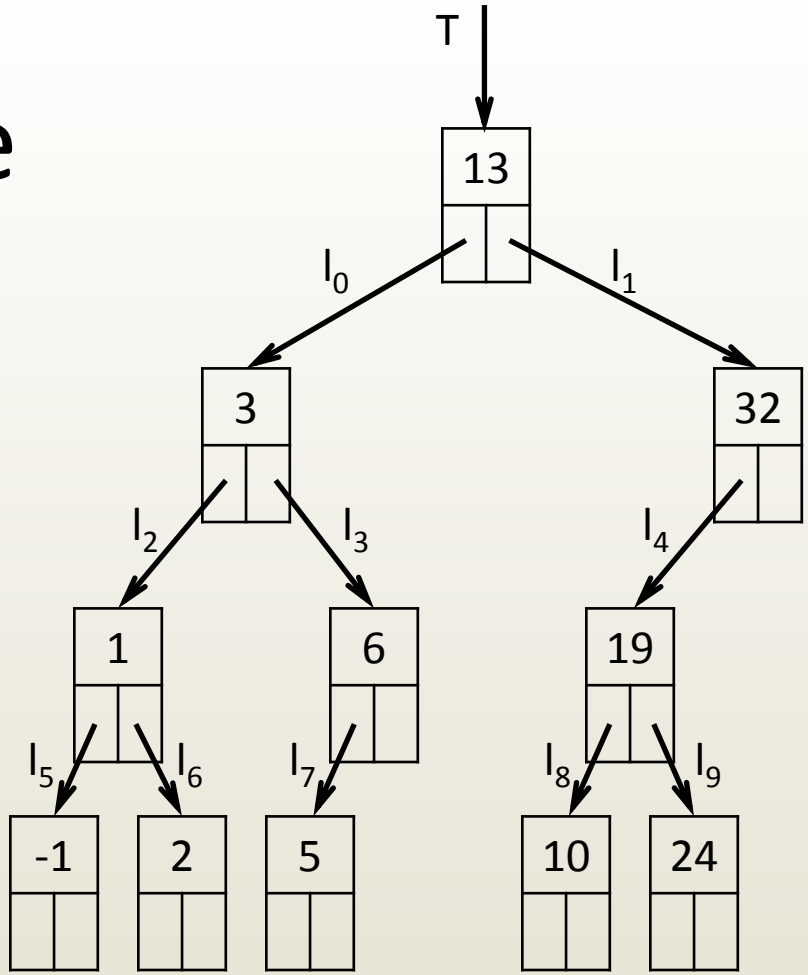
Trace



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	

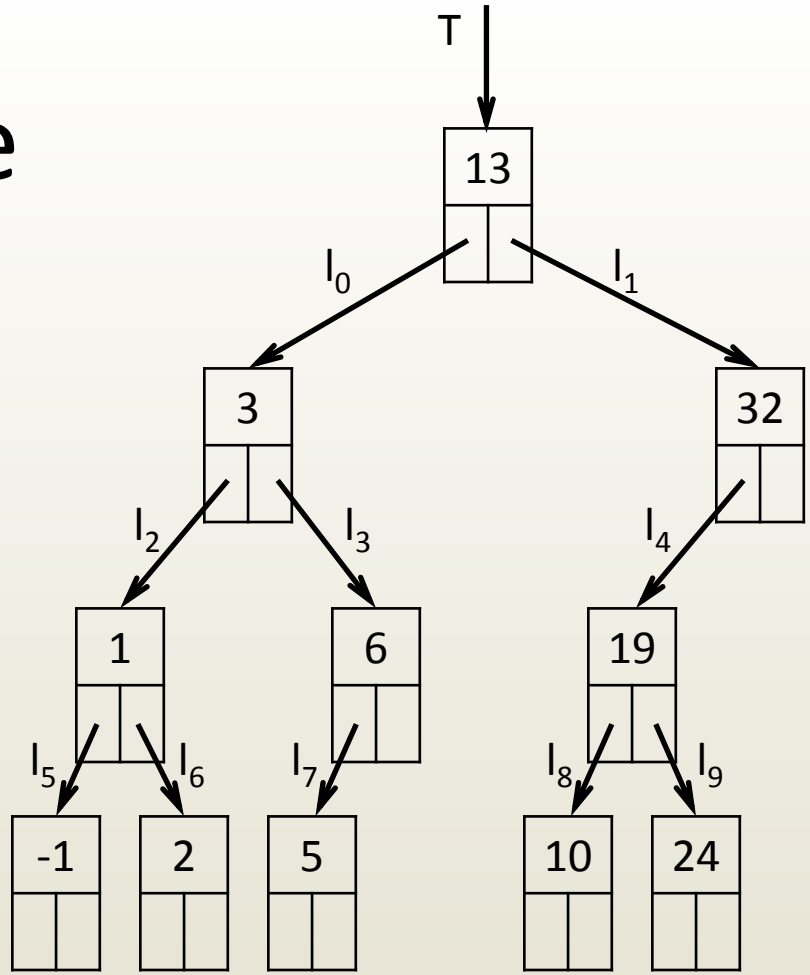
Trace



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

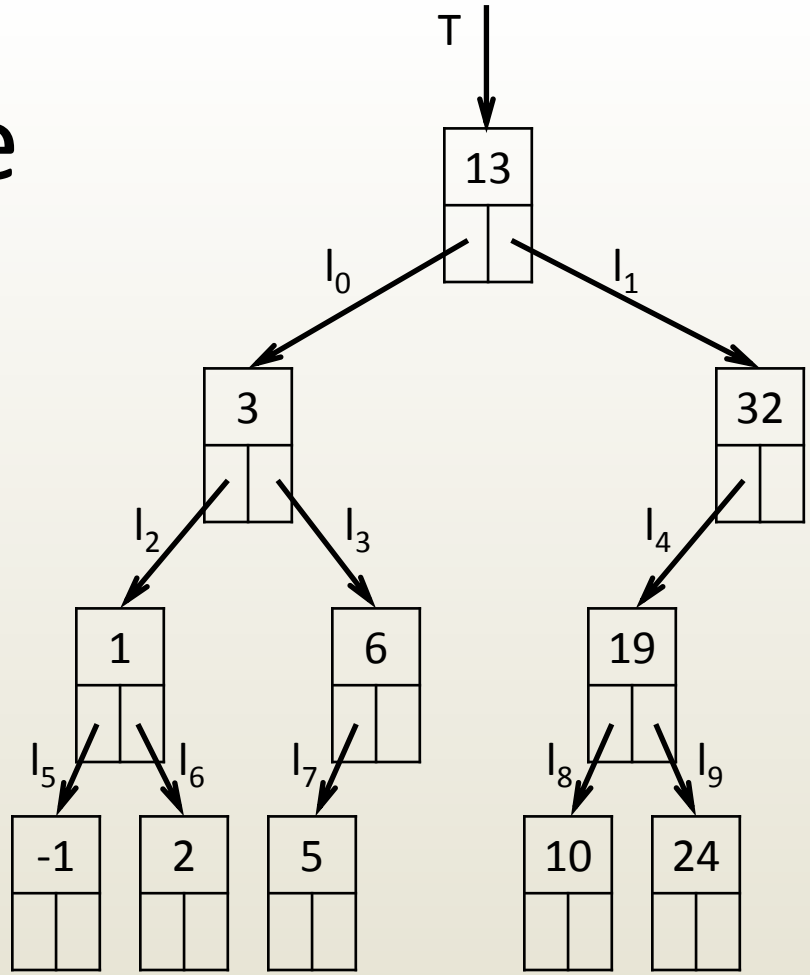
Trace



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	

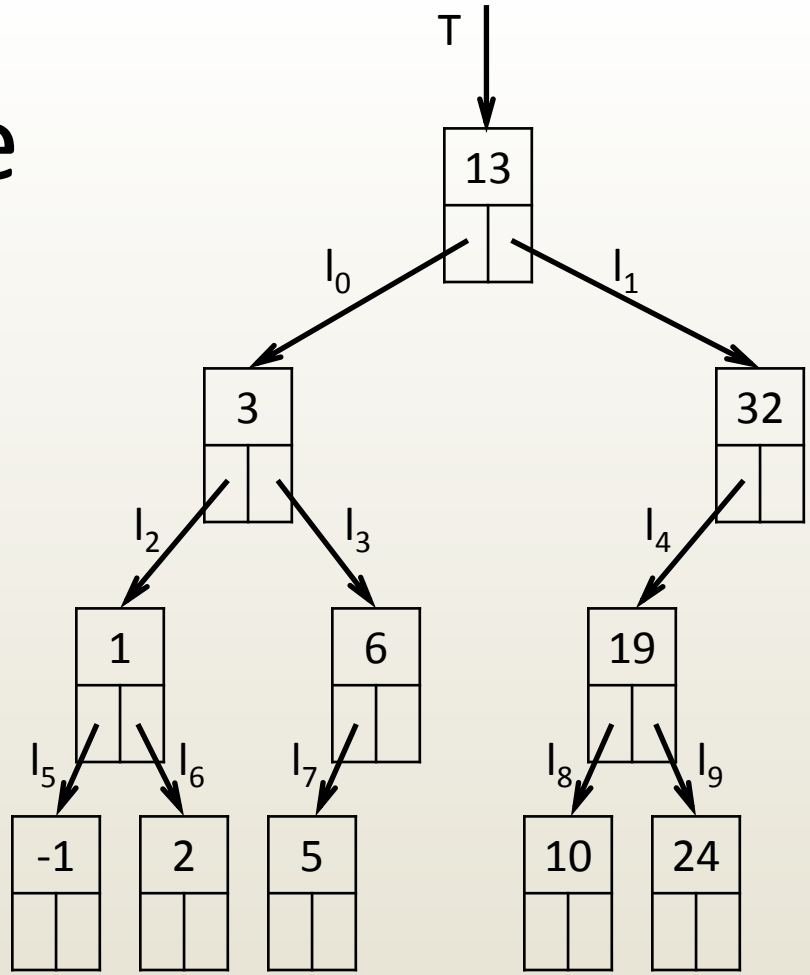
Trace



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

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		

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
```

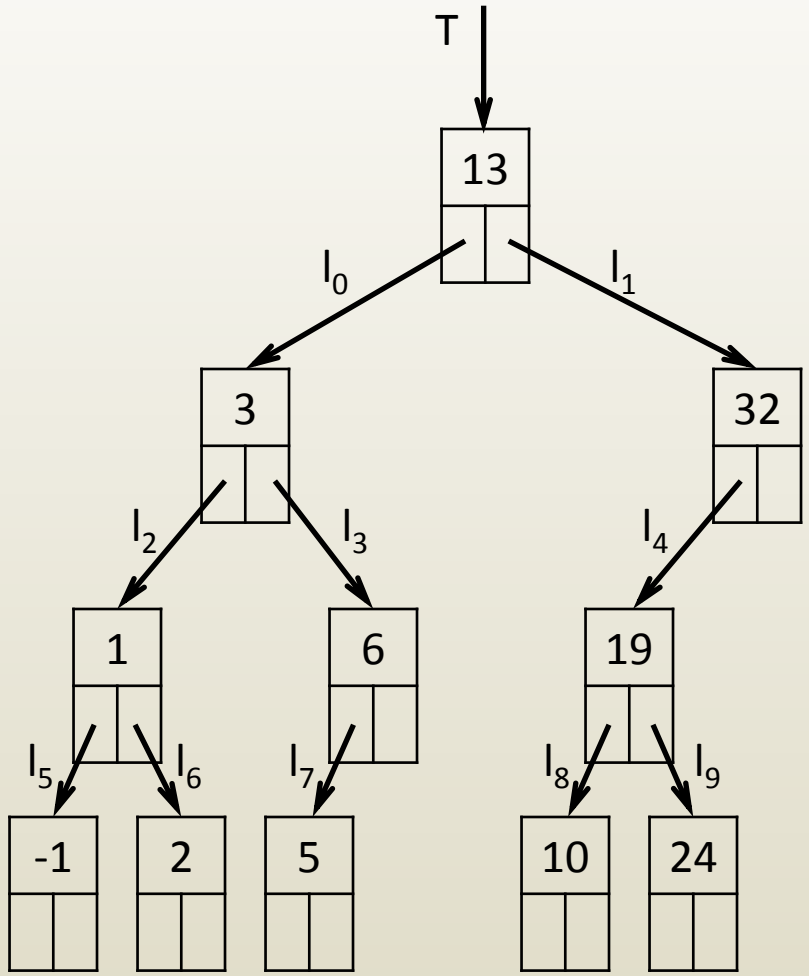
```
Input:
    T // link to root of binary tree
          // a node stores (val, left, right)
Output:
    // print all values stored in the tree
TraverseBTM(T)
    L = empty list of links to nodes
    L = InsertFirstList(L, T)
    while L is not empty
        curr = ExtractFirstNode(L)
        print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

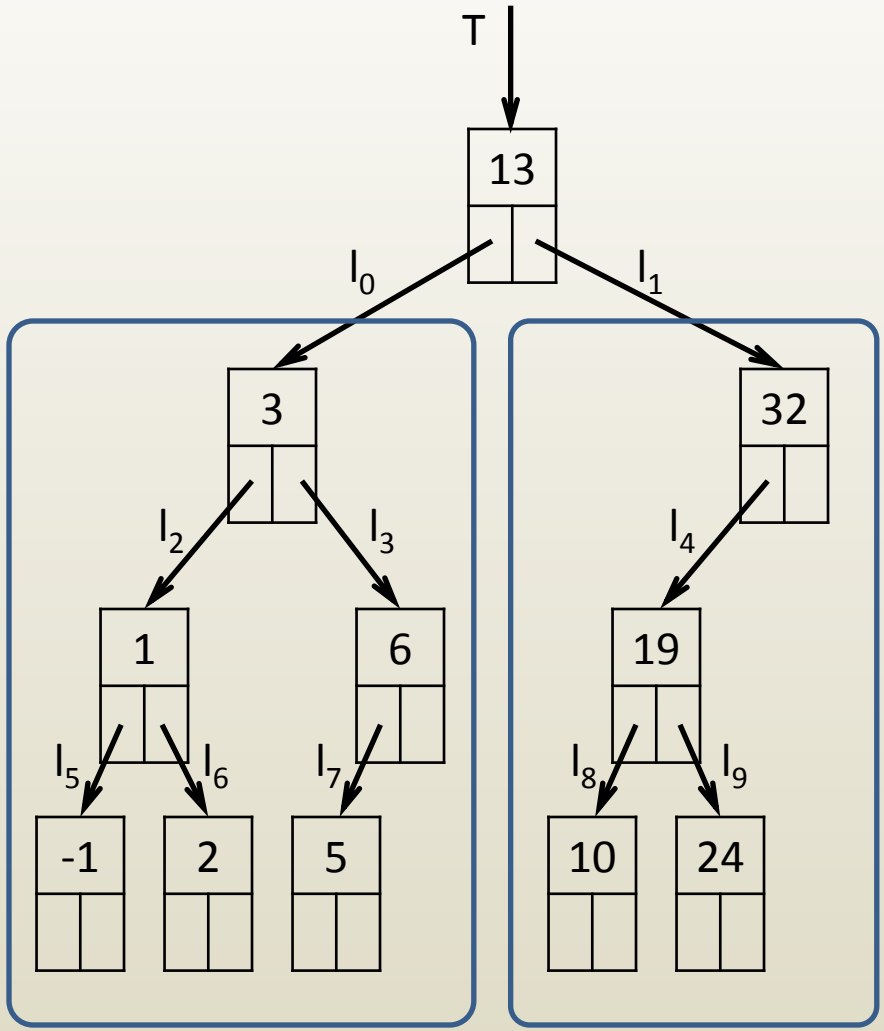
Binary tree node count

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



Recursive algorithm for binary tree node count

- Insight
  - T is a node with two binary trees connected to it
  - Node count is
    - node count in left subtree (n_l)
    - + 1 for the root
    - + node count in right subtree (n_r)



$n = n_l + 1 + n_r$

Recursive algorithm for binary tree node count

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

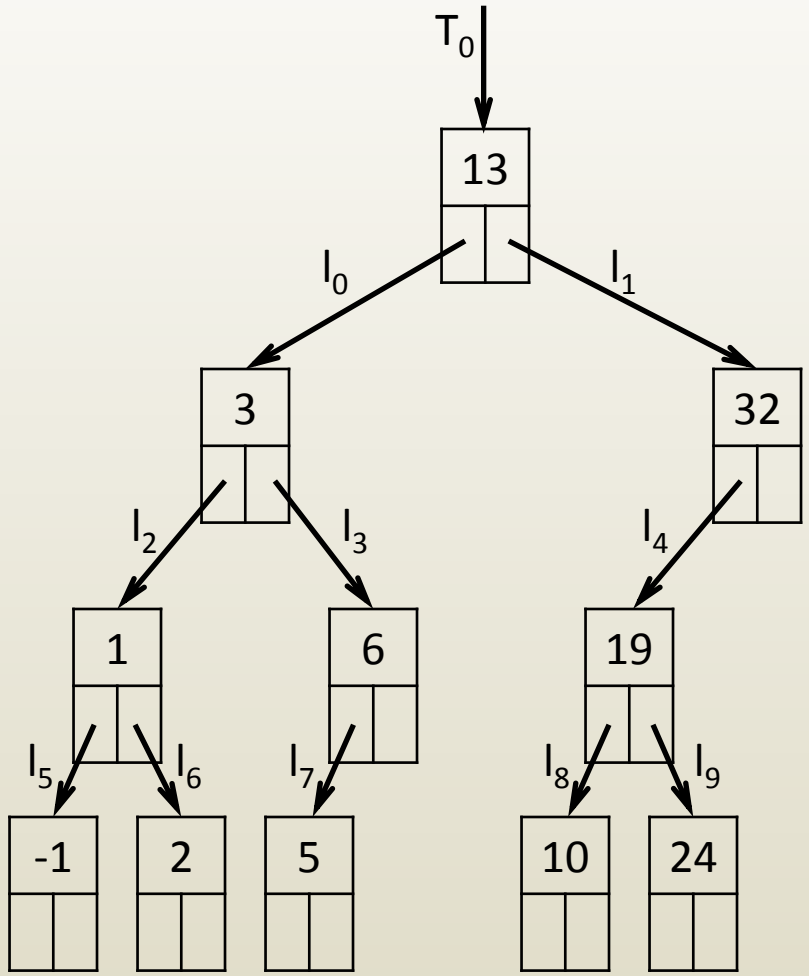
if T == NULL

return 0

endif

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

endCountBTR



Recursive algorithm for binary tree node count

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

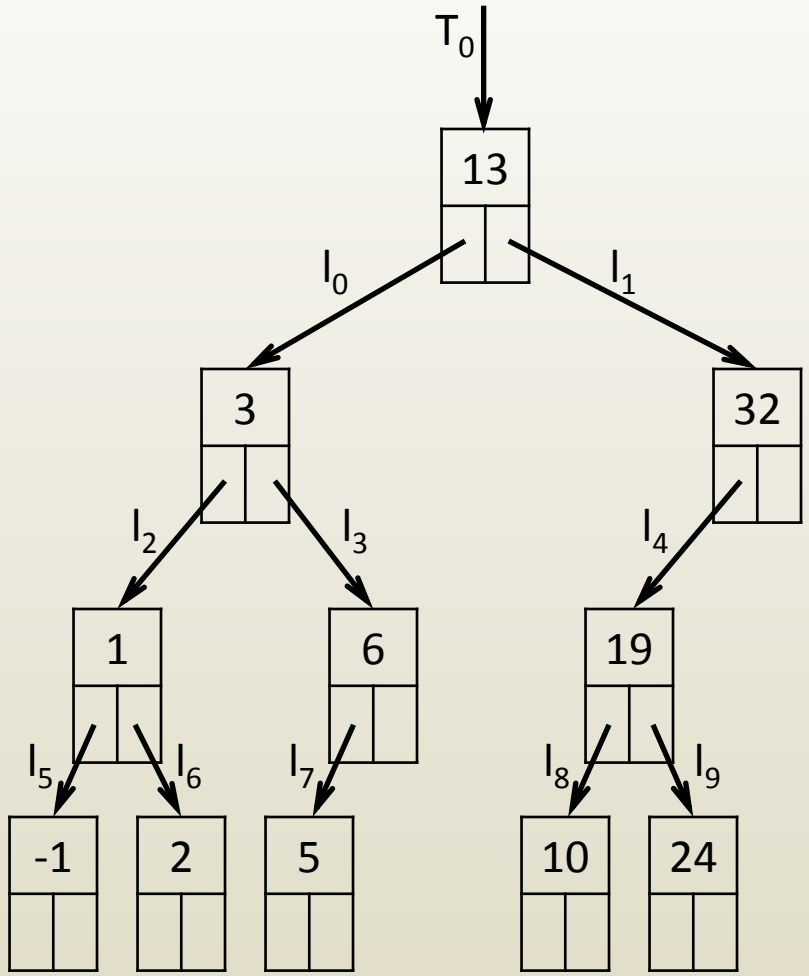
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

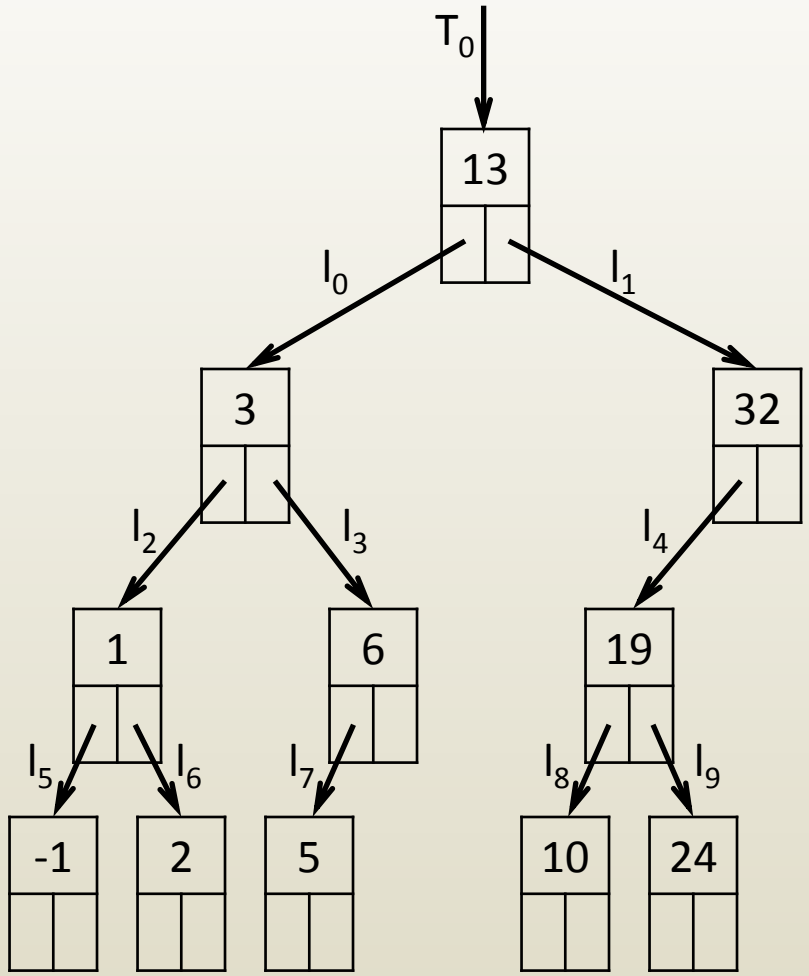
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

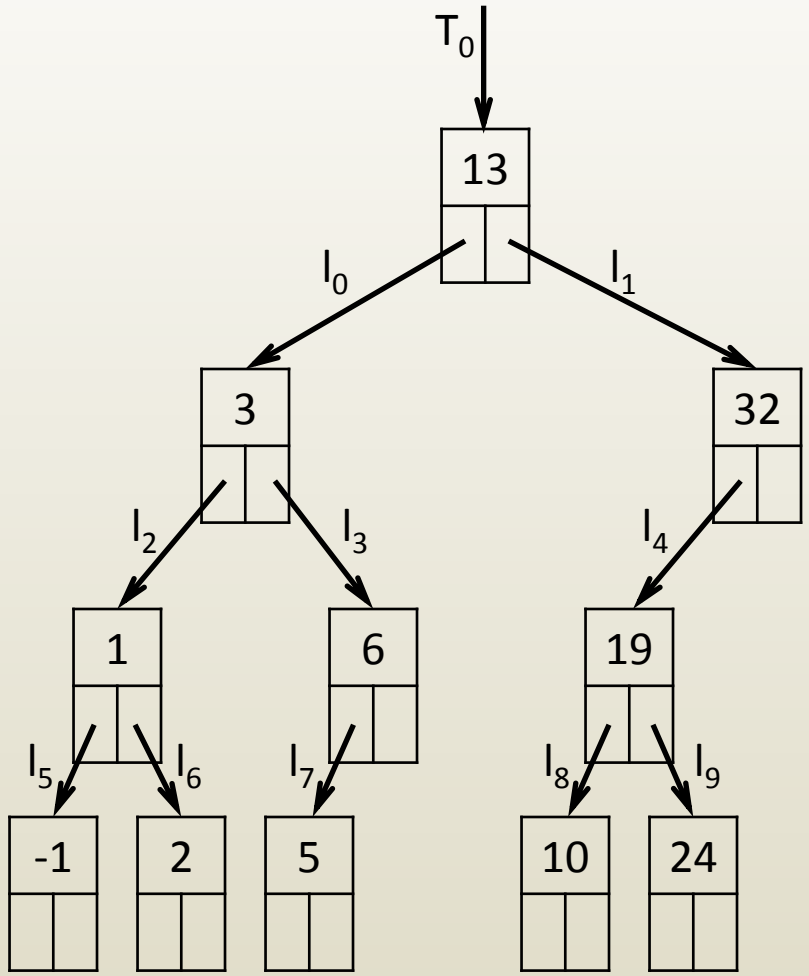
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

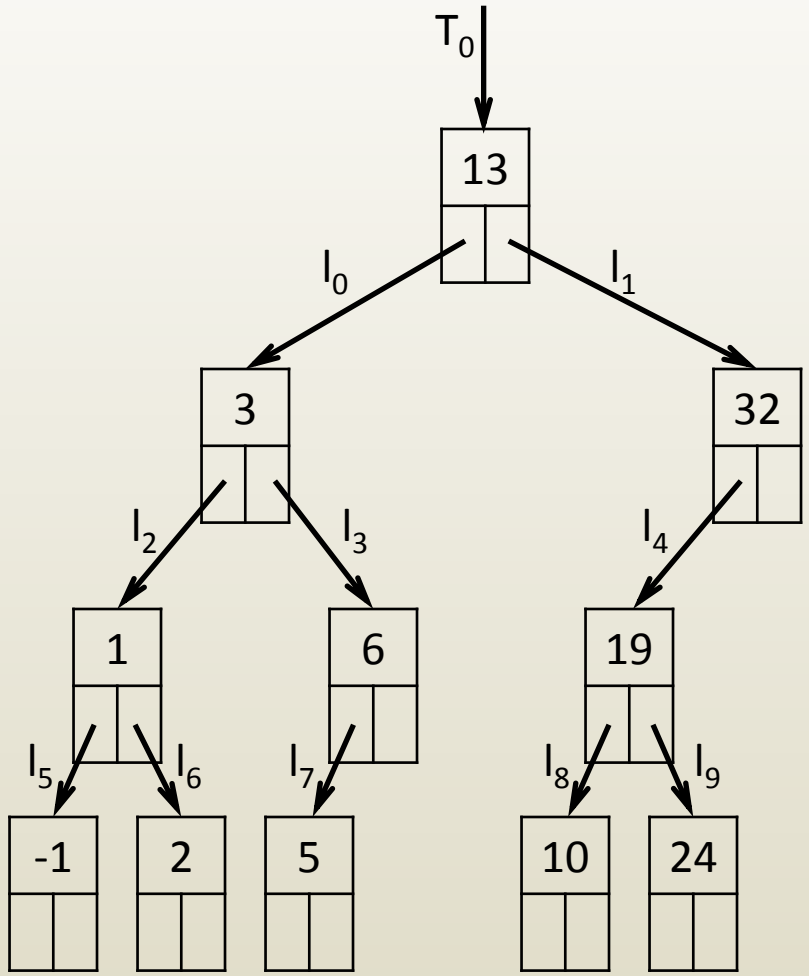
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

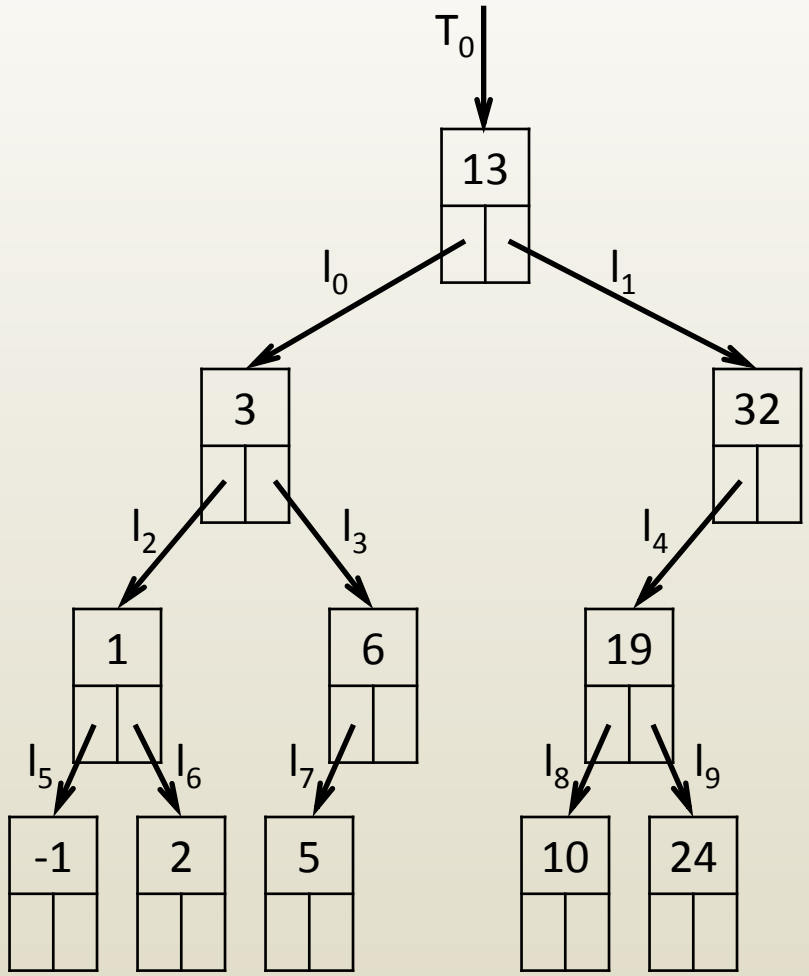
if T == NULL

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)

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

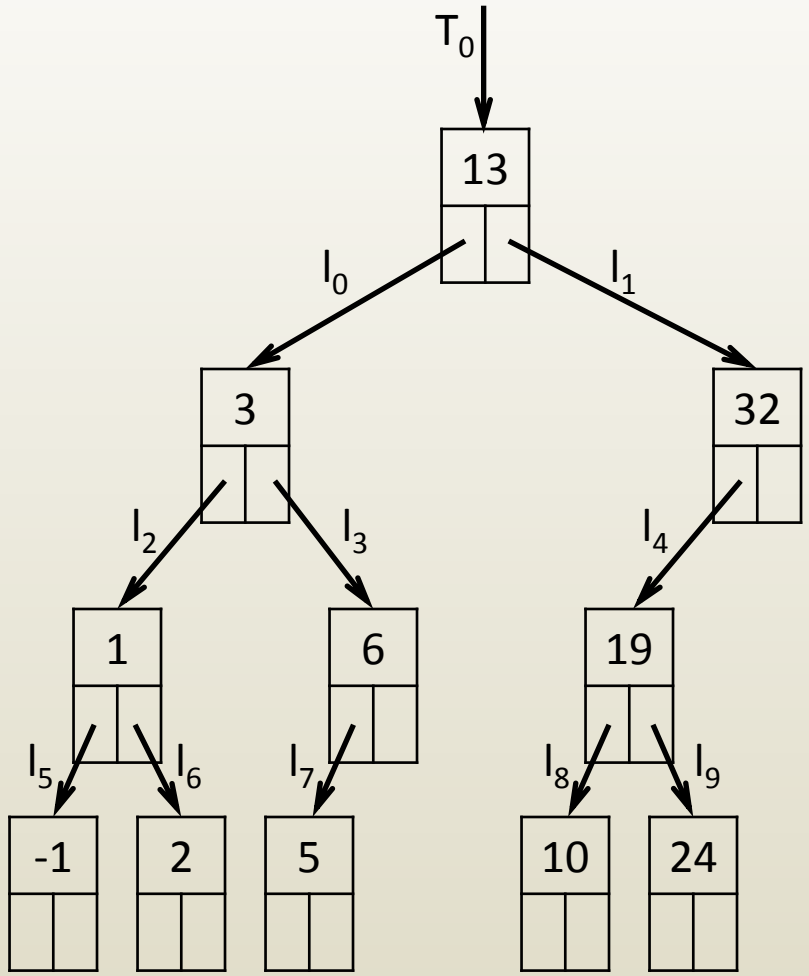
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

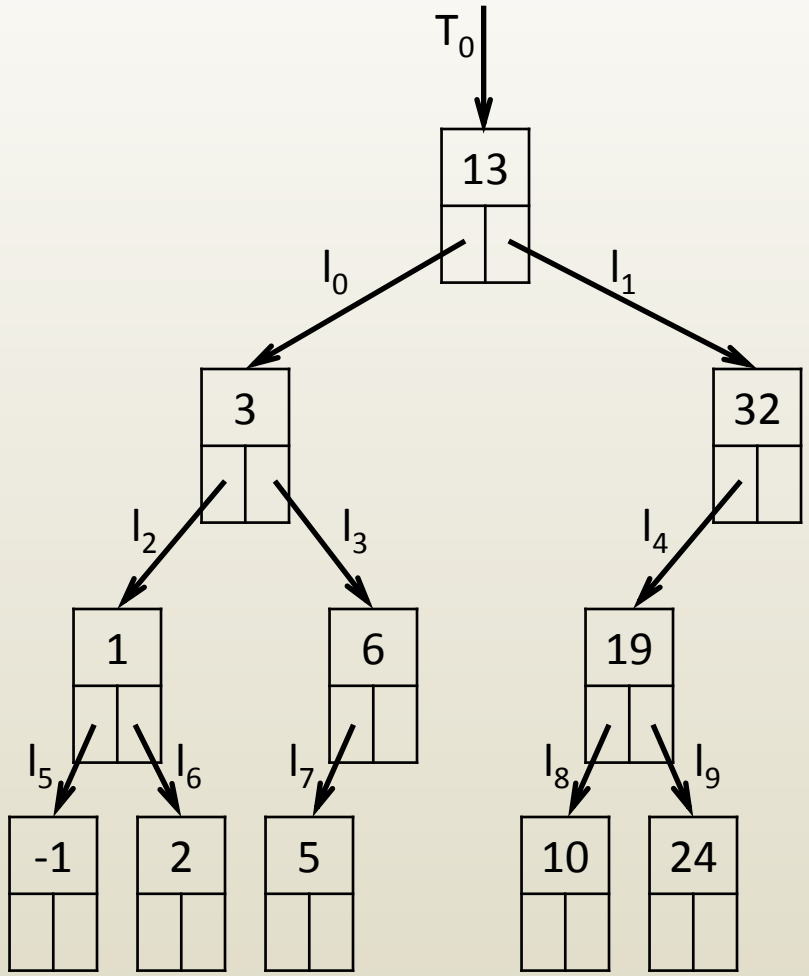
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

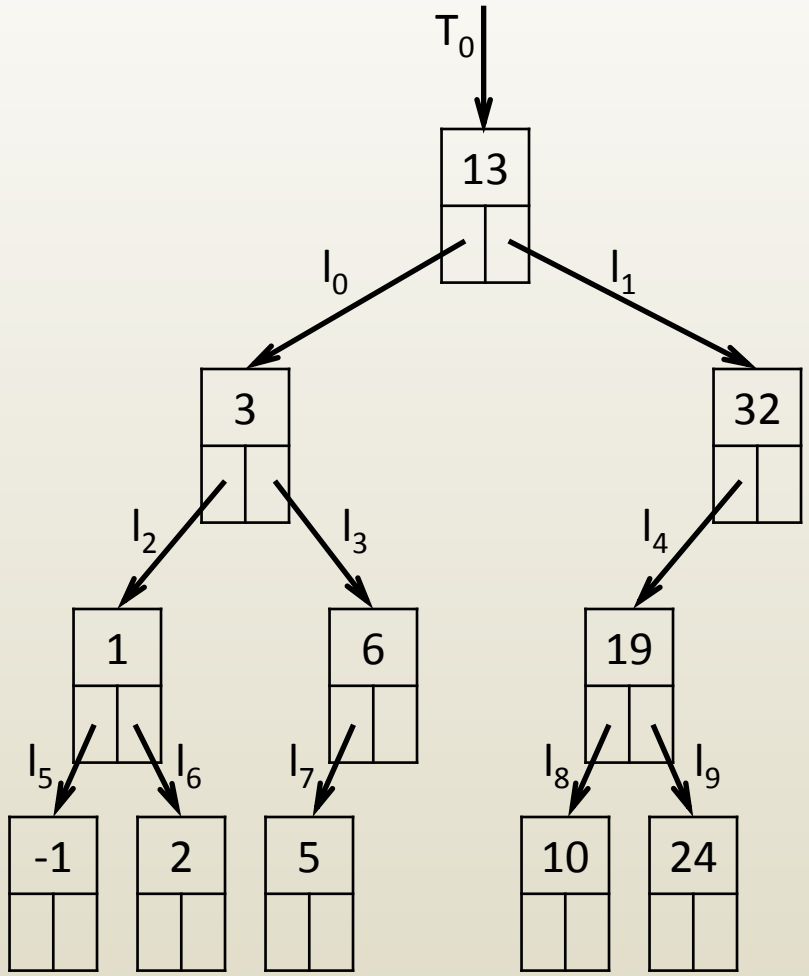
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

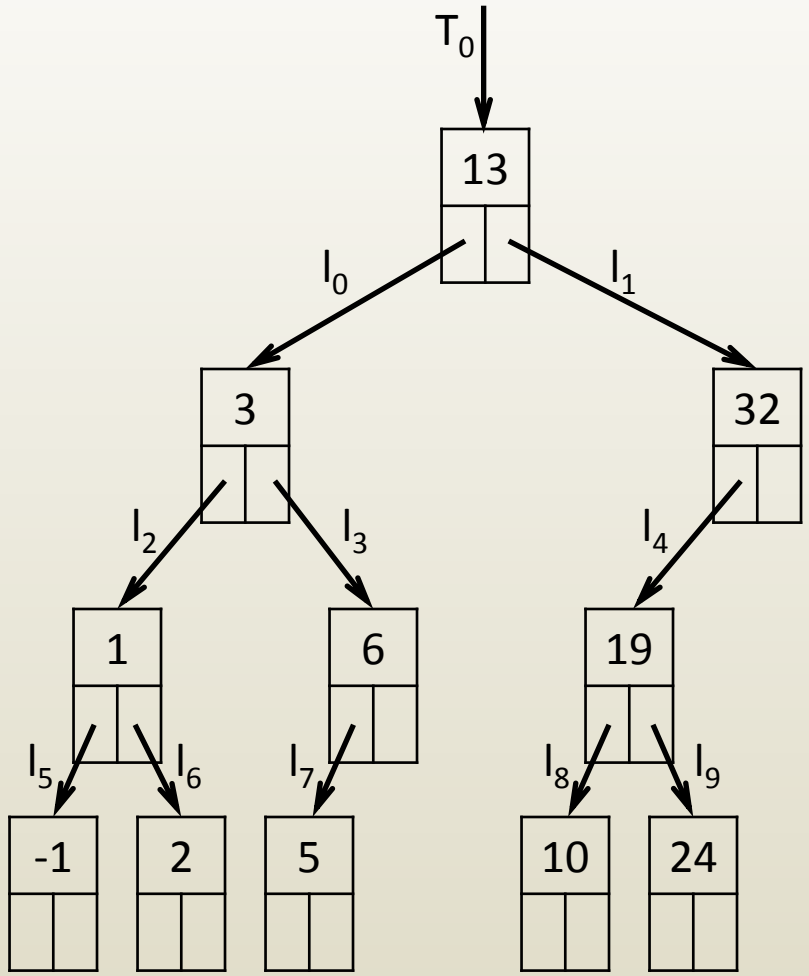
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

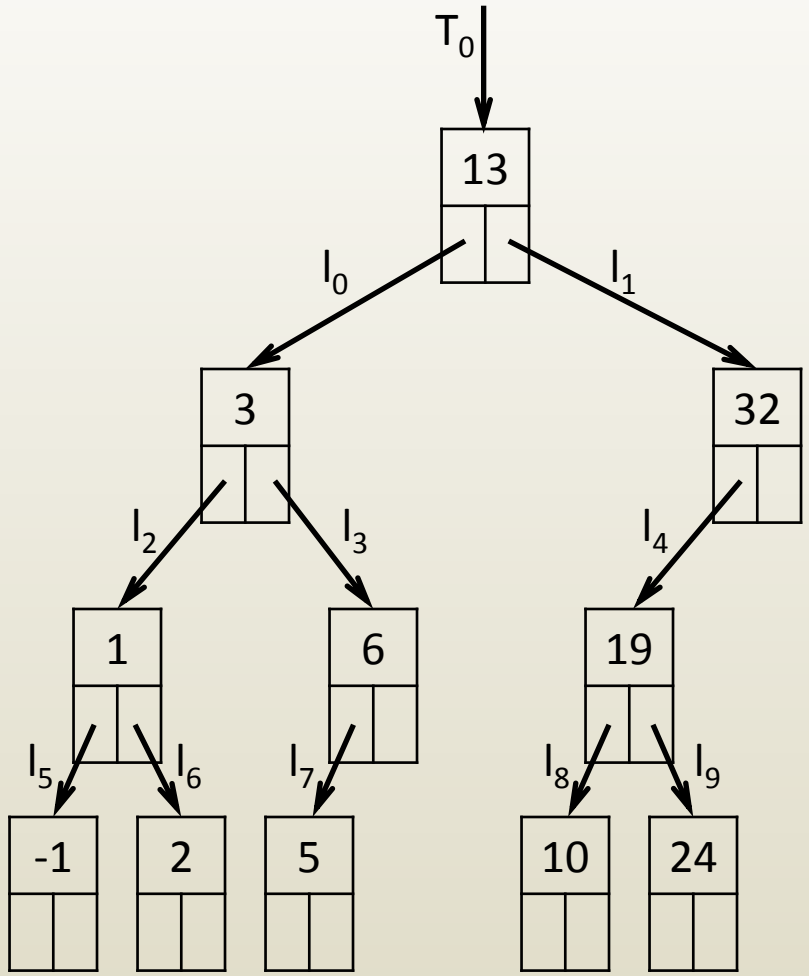
if T == NULL

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)

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

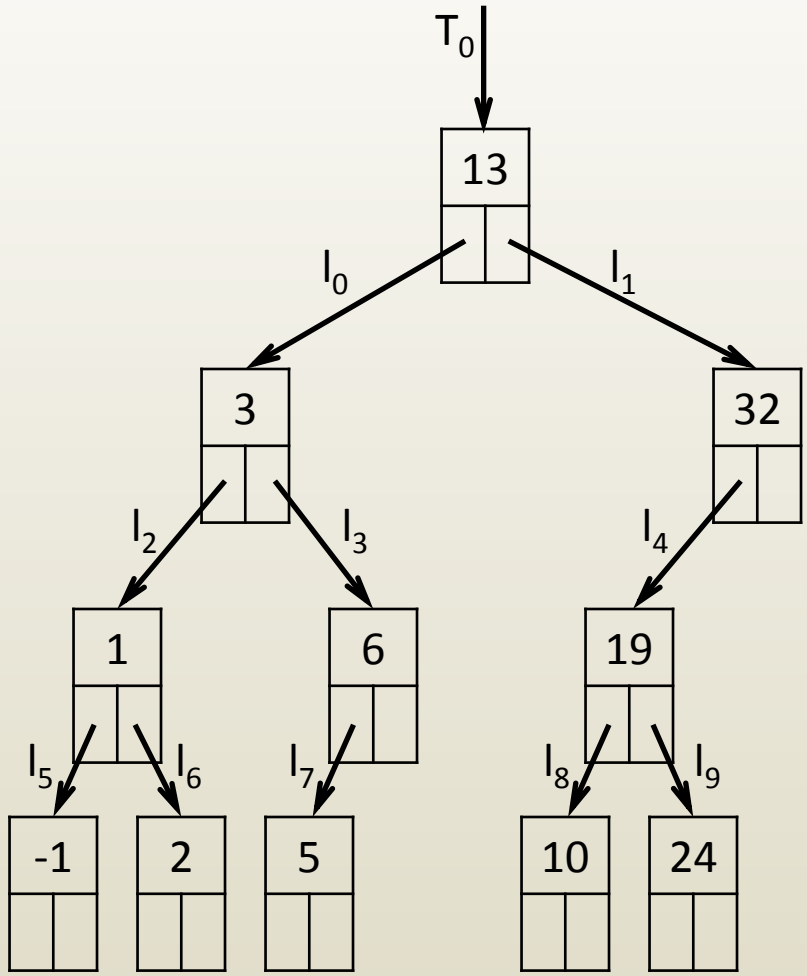
if T == NULL

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)

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

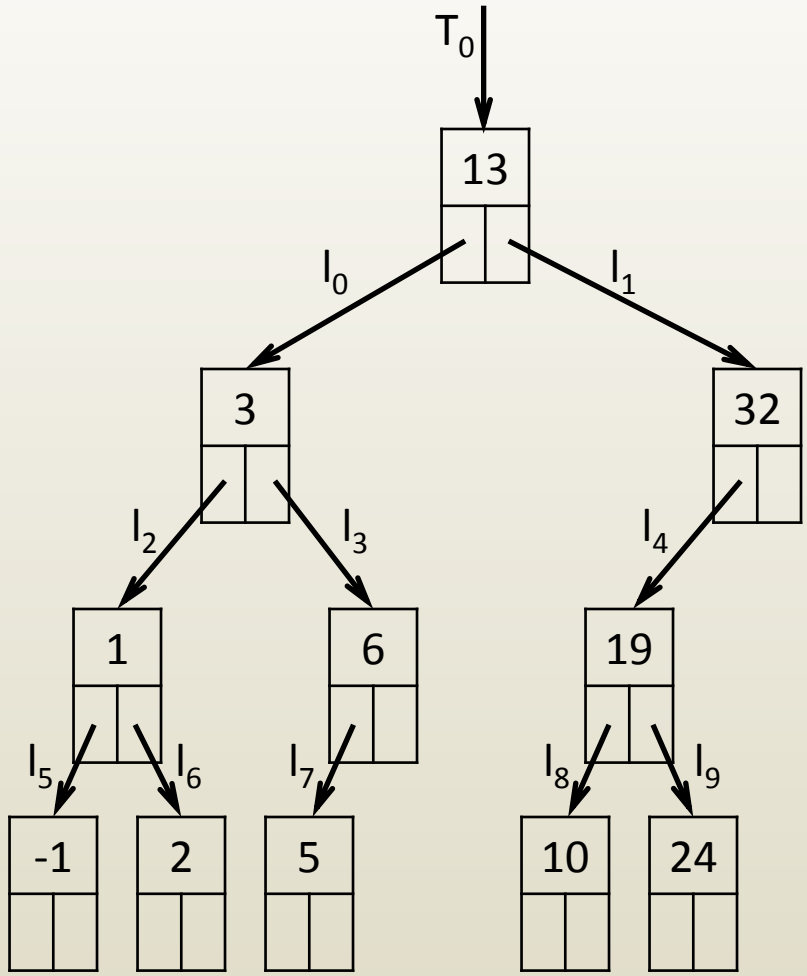
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

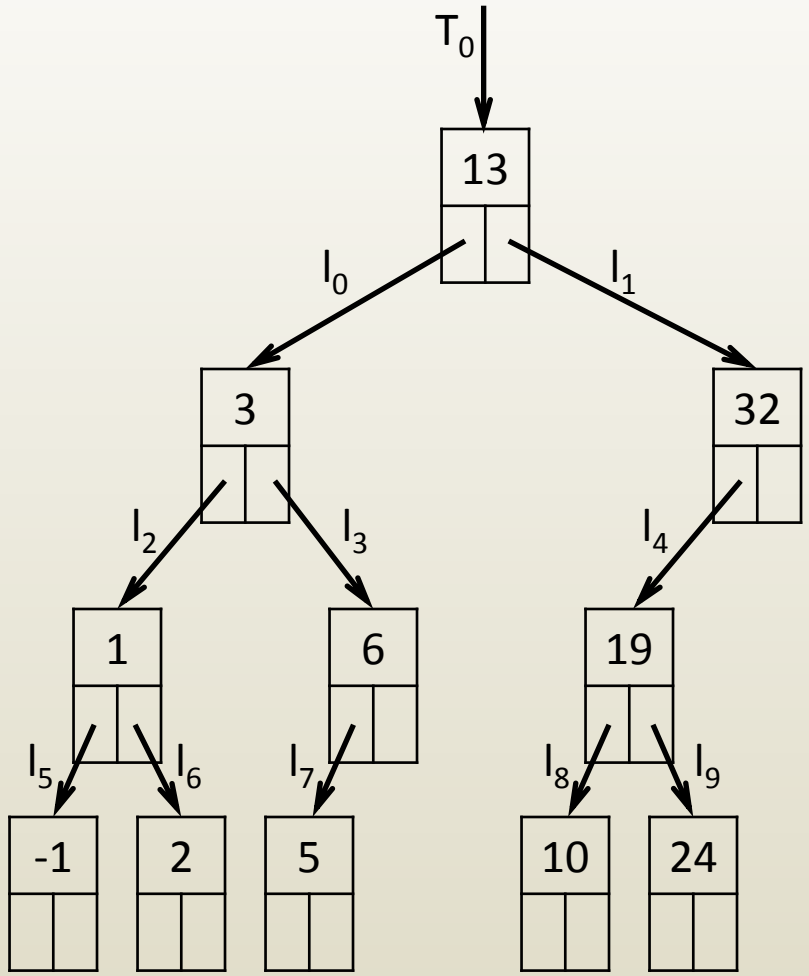
if T == NULL

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)

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

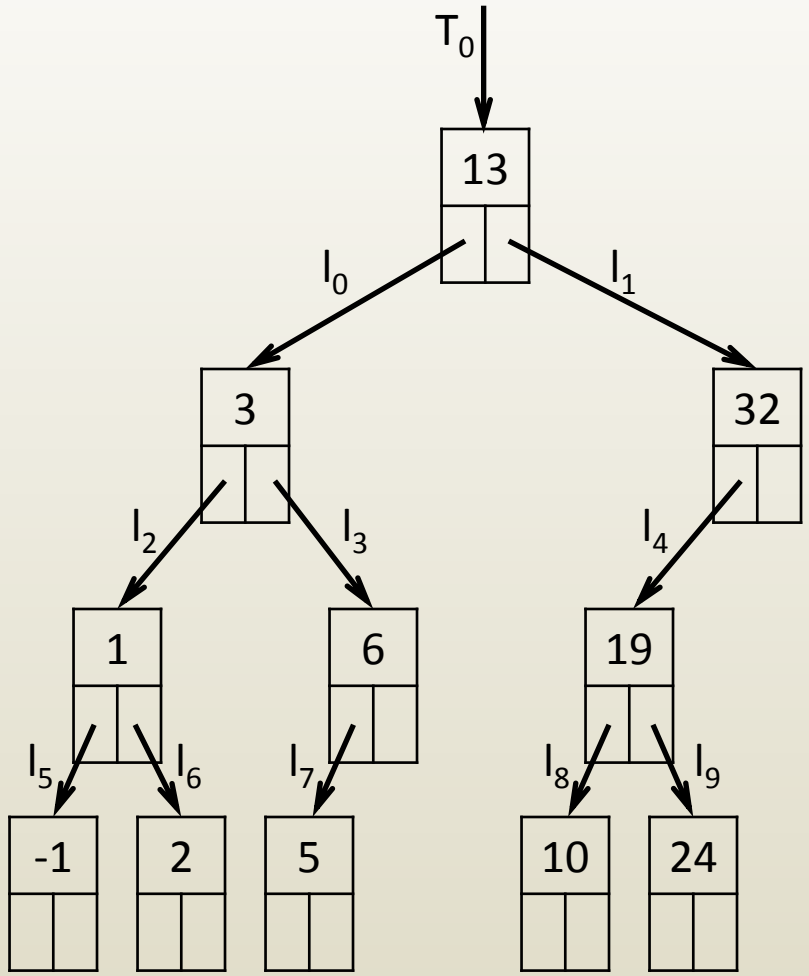
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

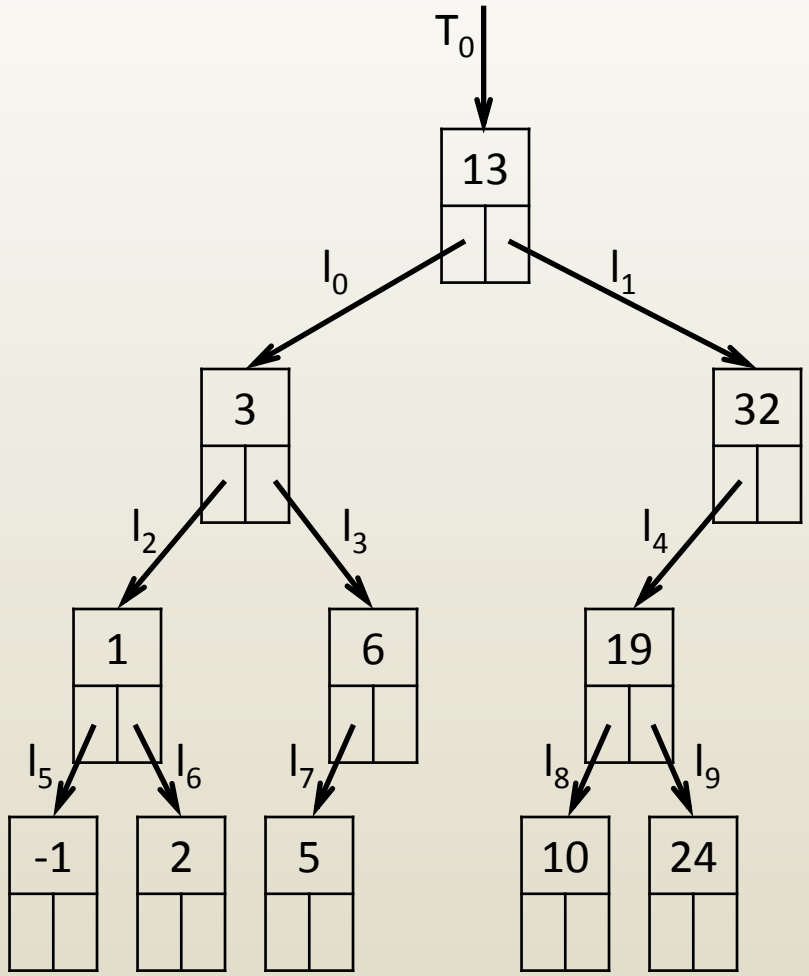
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

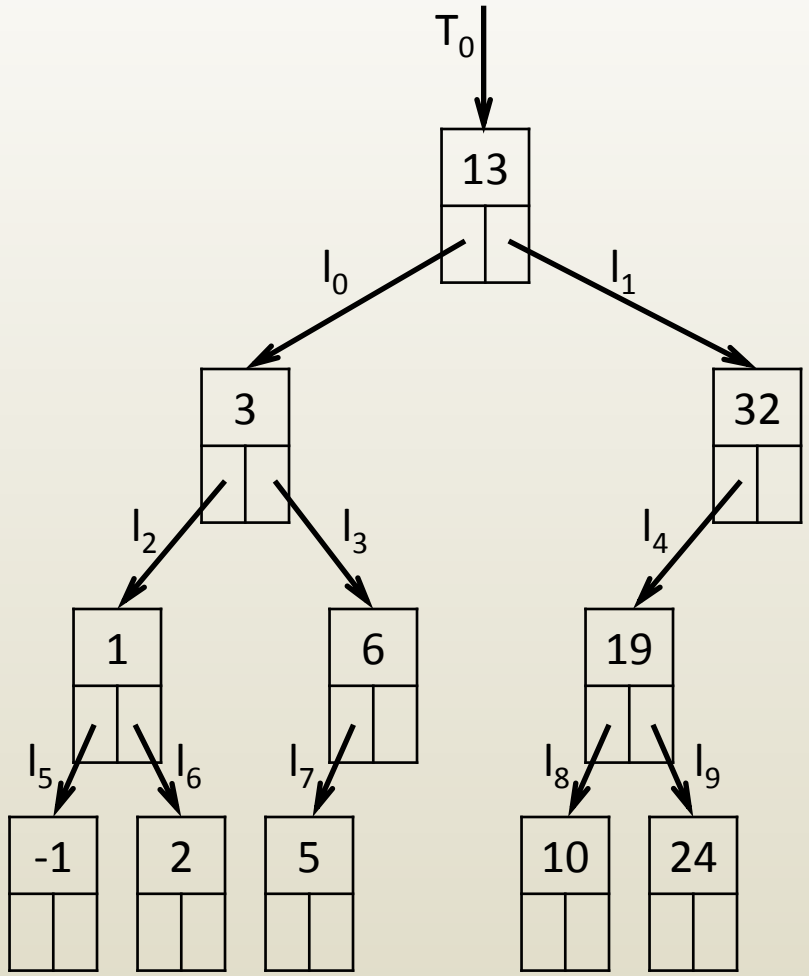
if T == NULL

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)

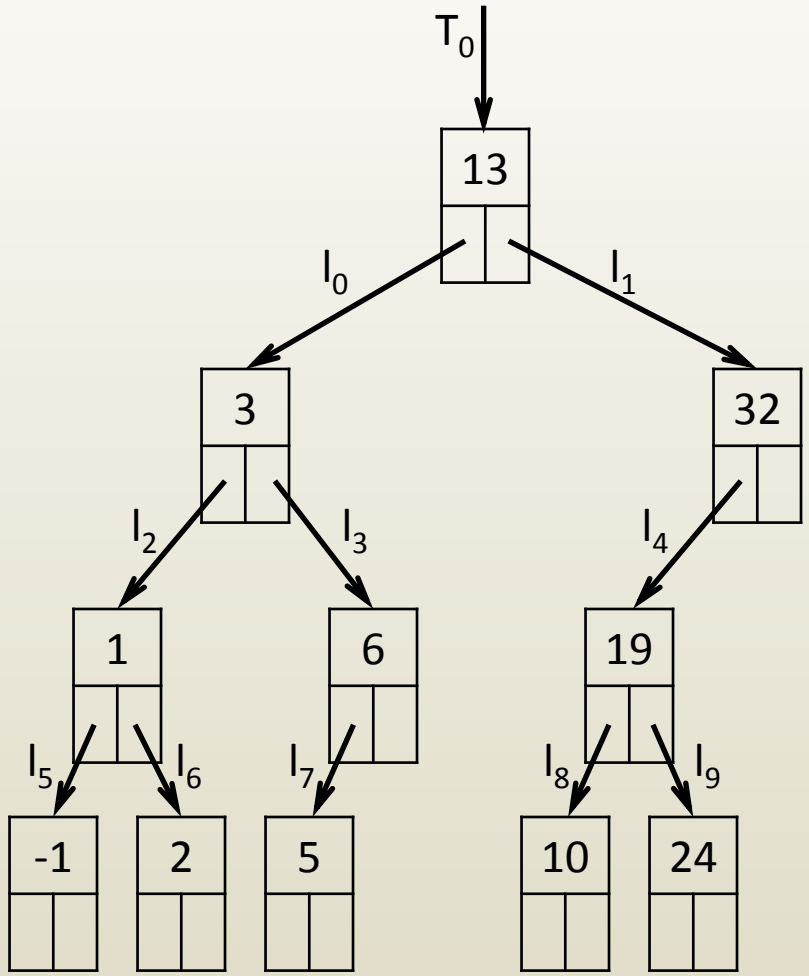
if T == NULL

return 0

endif

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

endCountBTR



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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

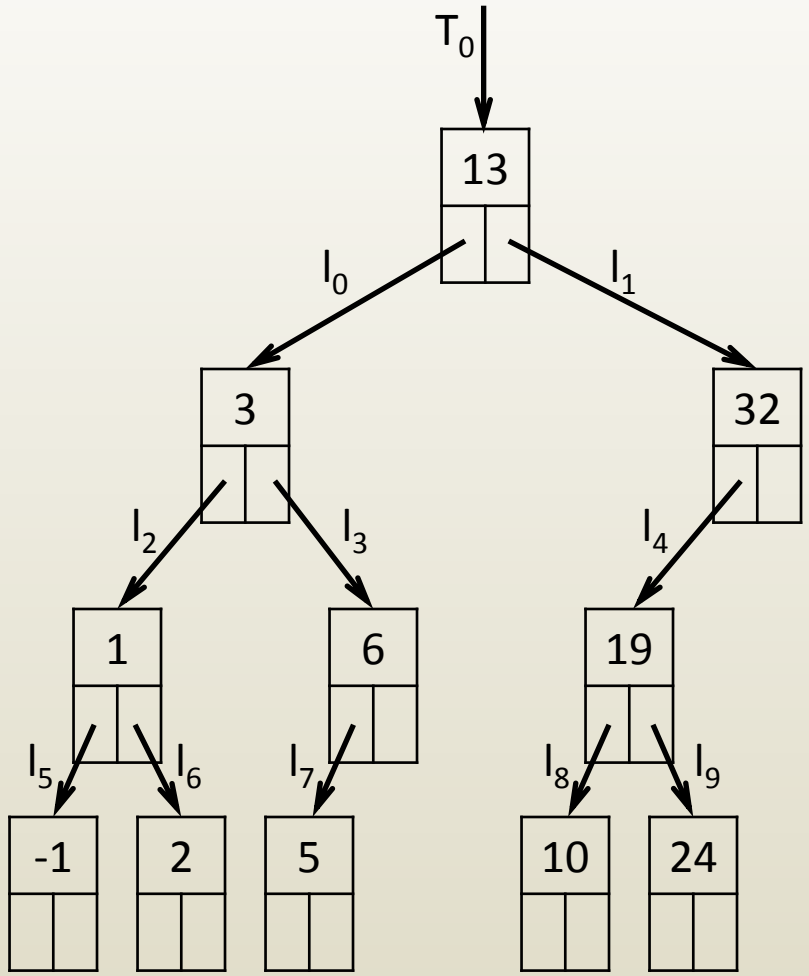
if T == NULL

return 0

endif

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

endCountBTR



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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

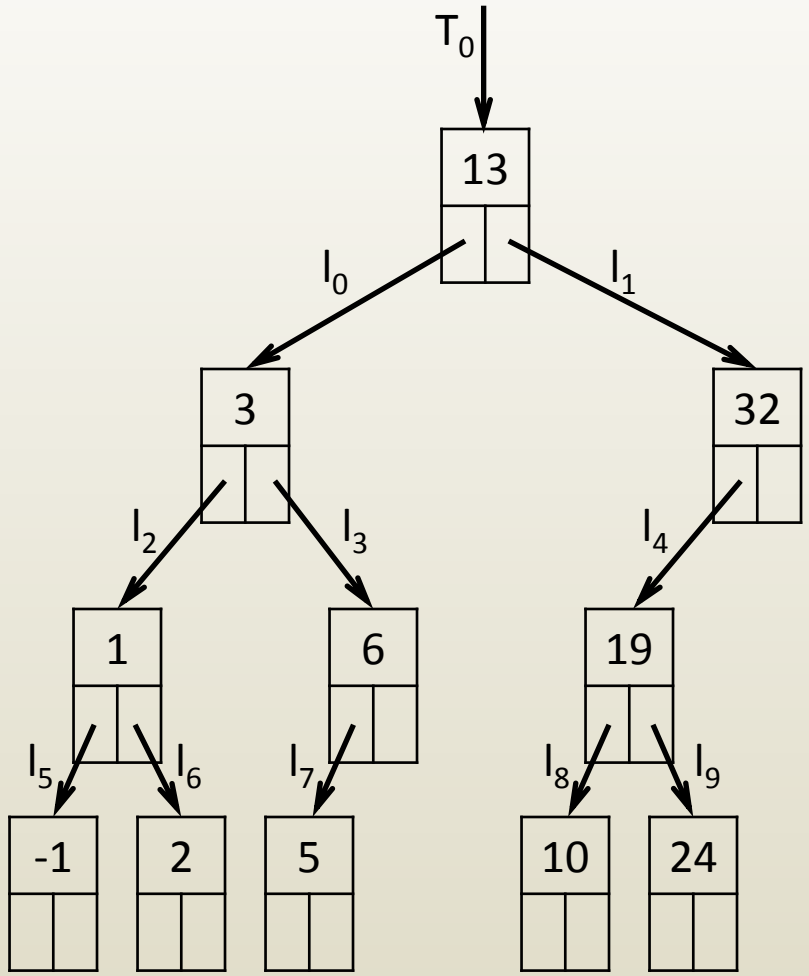
if T == NULL

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)	CountBTR(l_8)+1+CountBTR(l_9)

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

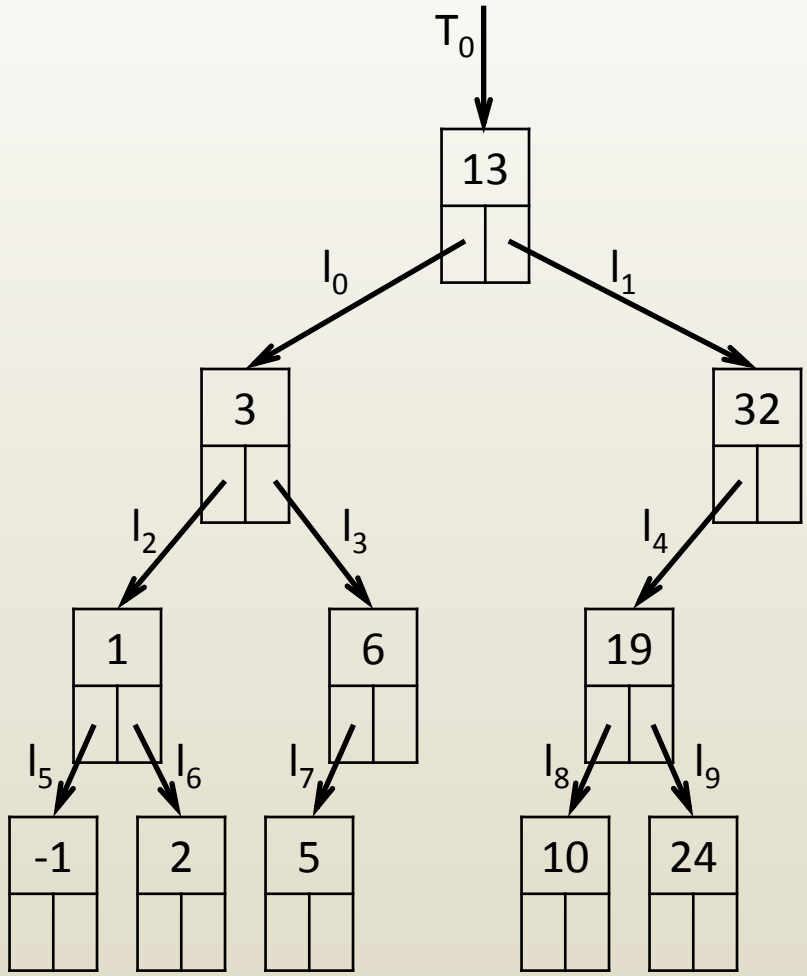
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

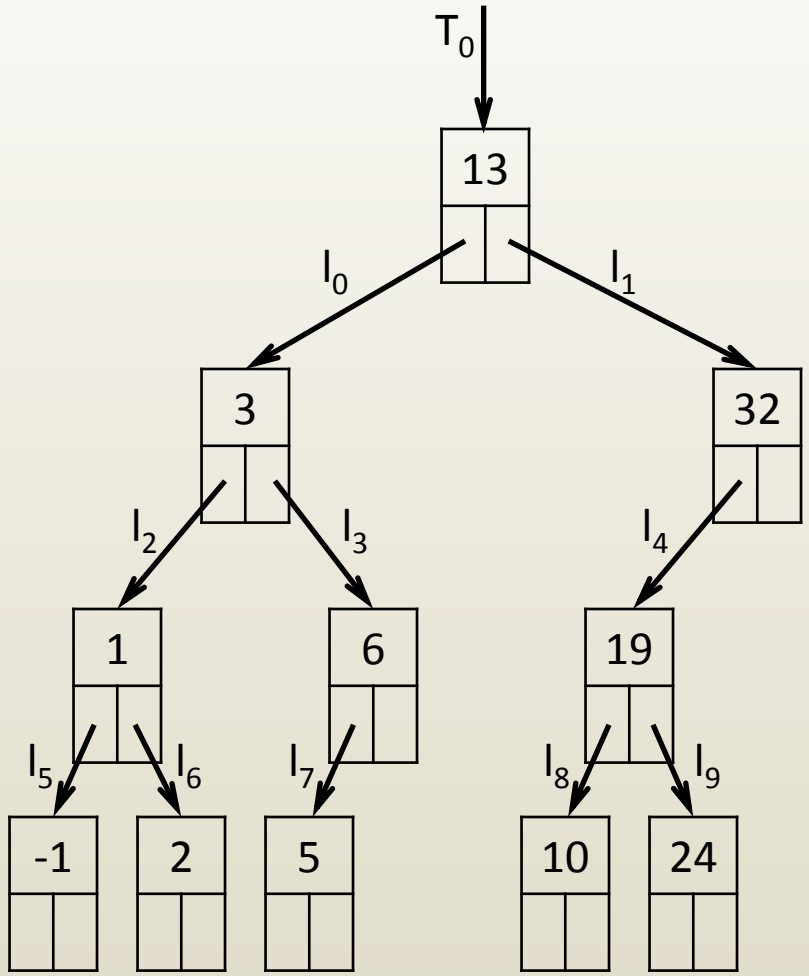
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

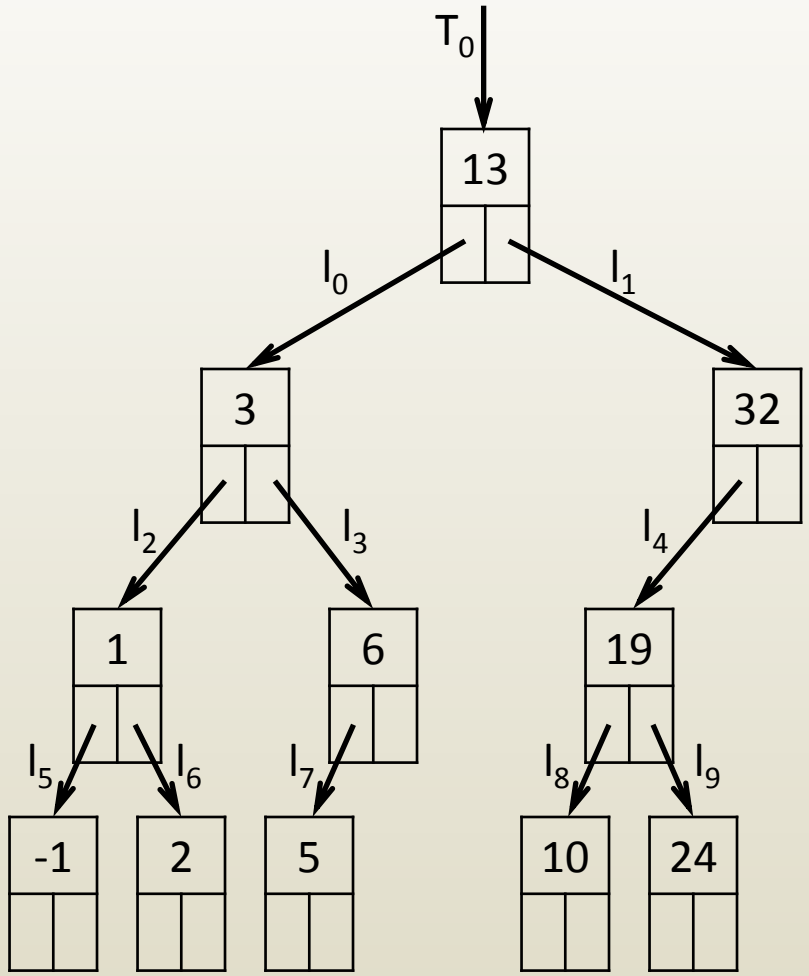
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

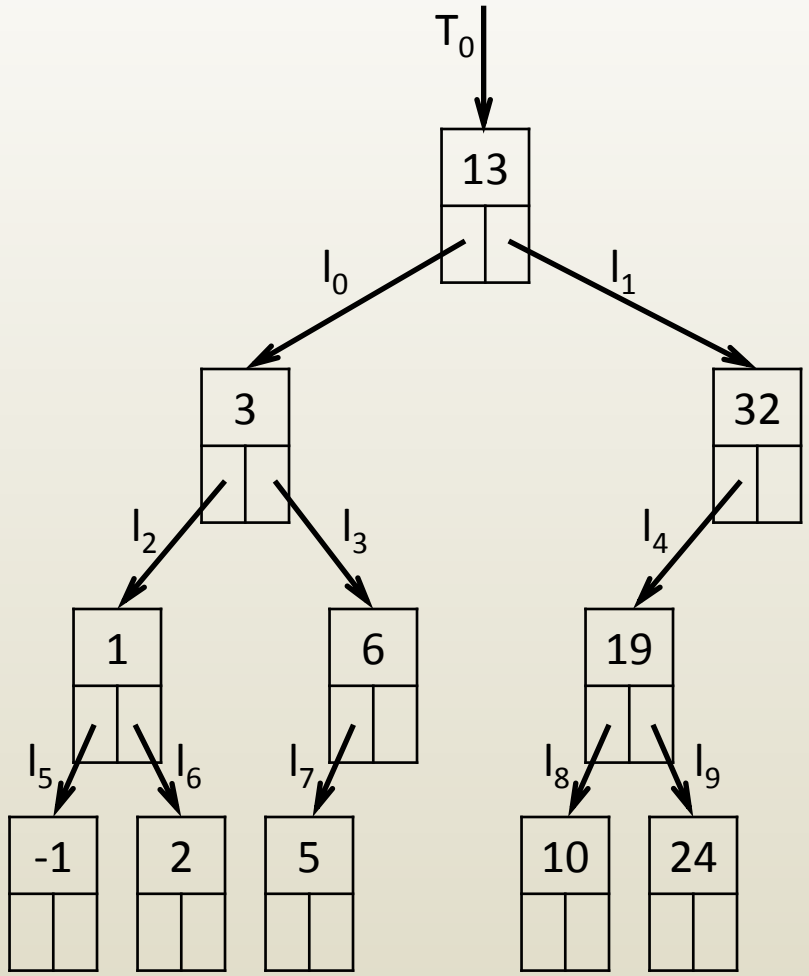
if T == NULL

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

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

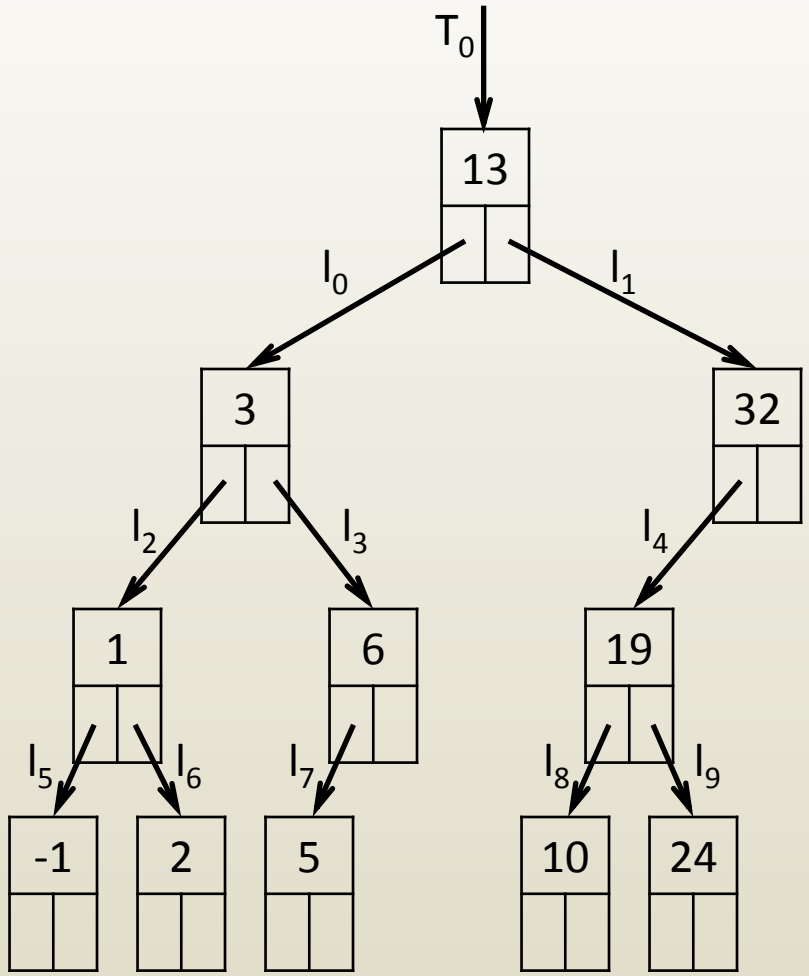
if T == NULL

return 0

endif

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

endCountBTR



Call	Return
CountBTR(T_0)	6+1+4

Trace

Input:

T // link to root of binary tree

Output:

// count of nodes

CountBTR(T)

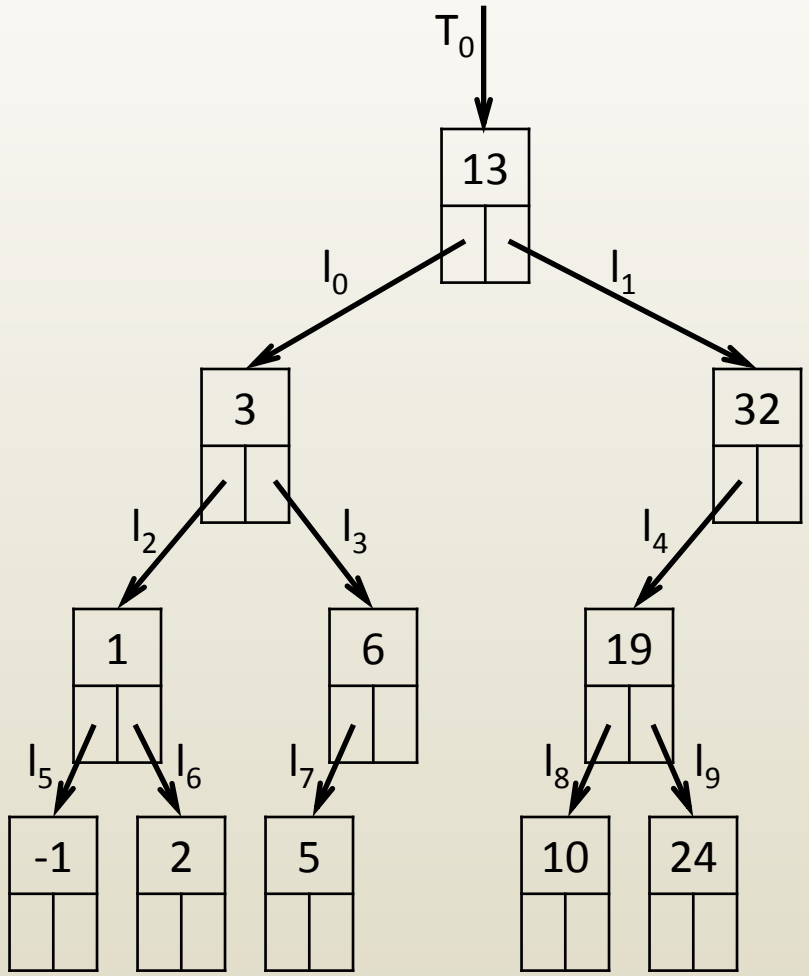
if T == NULL

return 0

endif

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

endCountBTR



Call	Return
CountBTR(T_0)	11

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 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

Recursion problem solving paradigm

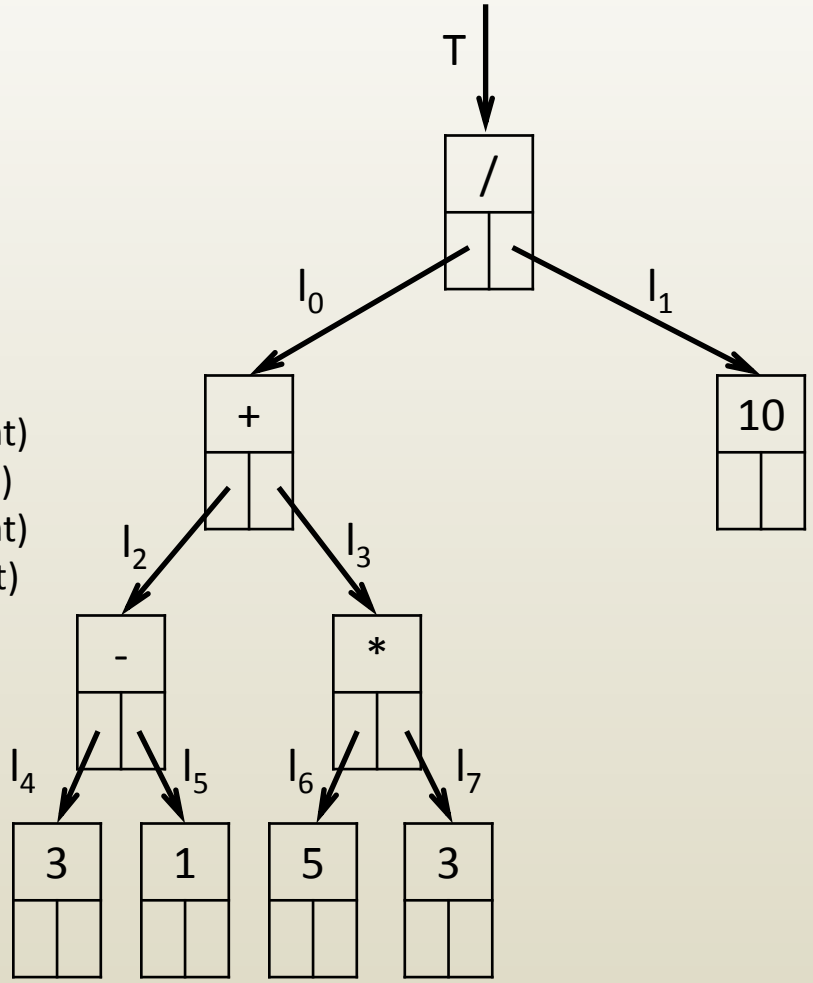
- You don’t solve the problem directly
- Split the problem until it becomes trivial
- Compute solution to problem by combining solutions of sub-problems
- Examples
  - Counting nodes in binary trees
  - Evaluating arithmetic expression
    - Value of leaf is number stored at leaf
    - Value for tree 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

Evaluating arithmetic expression

Input:
T // link to root of arithmetic expression binary tree
Output:
// expression printed out with parentheses
EvalAEBTR(T)
if T-> left == NULL
return T->val
endif
switch T->symbol
case '+': **return EvalAEBTR**(T->left) + **EvalAEBTR**(T->right)
case '-': **return EvalAEBTR**(T->left) - **EvalAEBTR**(T->right)
case '*': **return EvalAEBTR**(T->left) * **EvalAEBTR**(T->right)
case '/': **return EvalAEBTR**(T->left) / **EvalAEBTR**(T->right)
endswitch
endEvalAEBTR



Switch statement is a condensed and readable substitute for multiple if statements

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

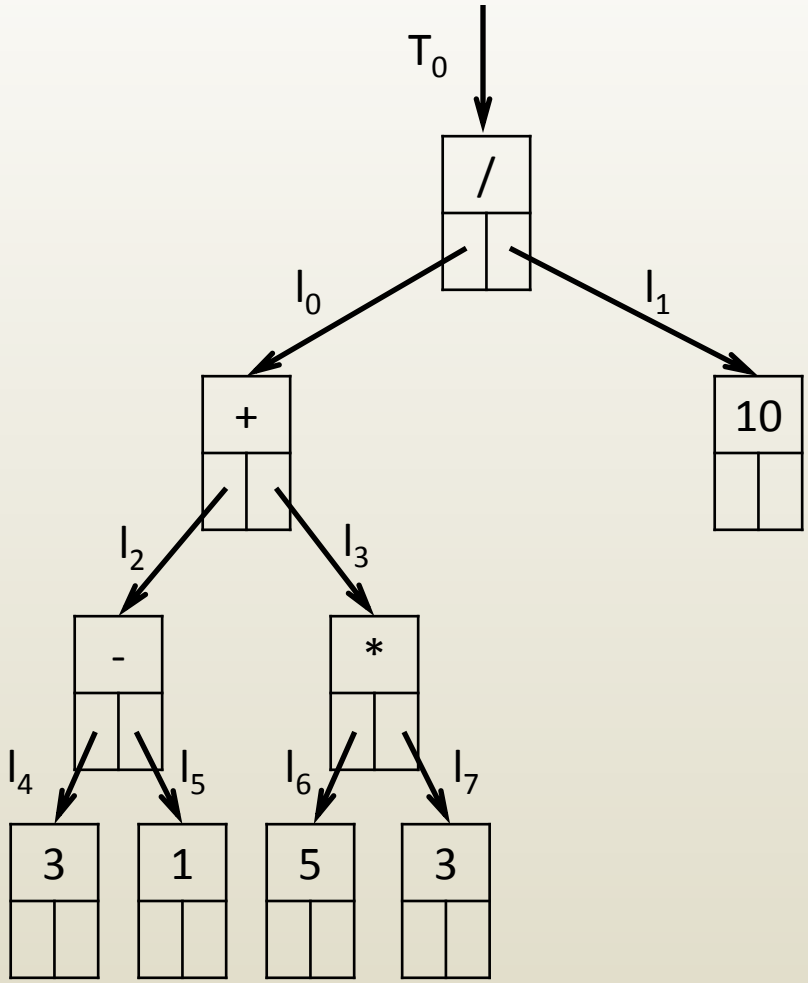
print "("; **PrintAEBTR**(T->left)

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)	(PrintAEBRT(NULL) **3** PrintAEBRT(NULL)**)**



Printout
((((

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

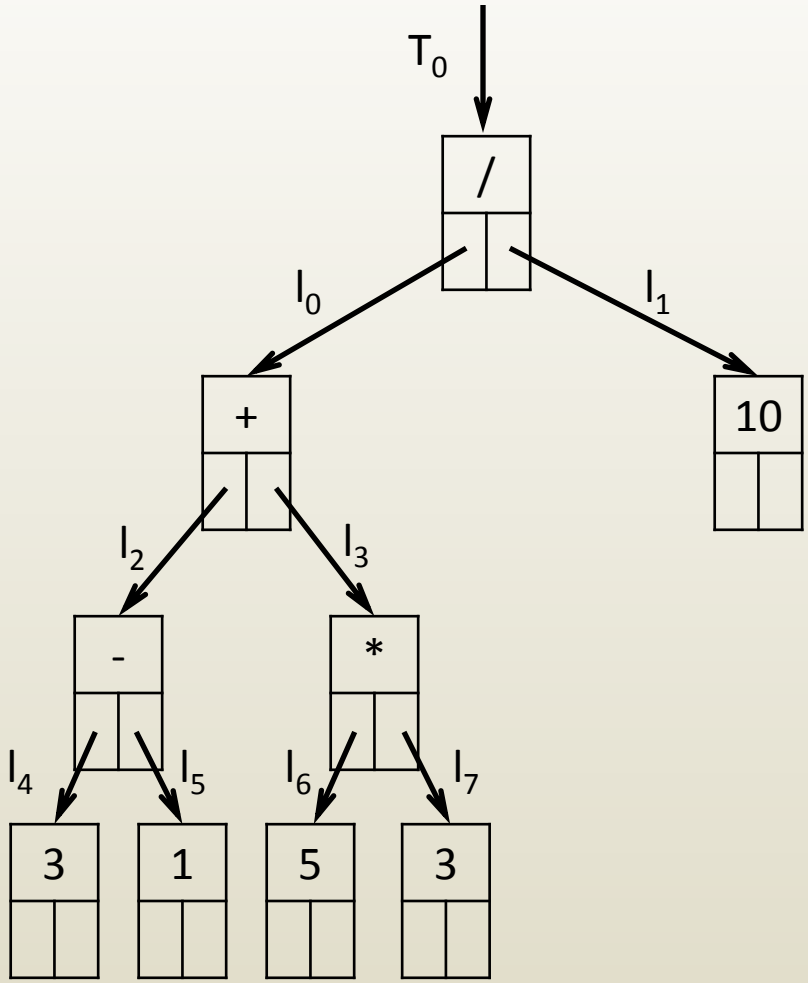
endif

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)

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

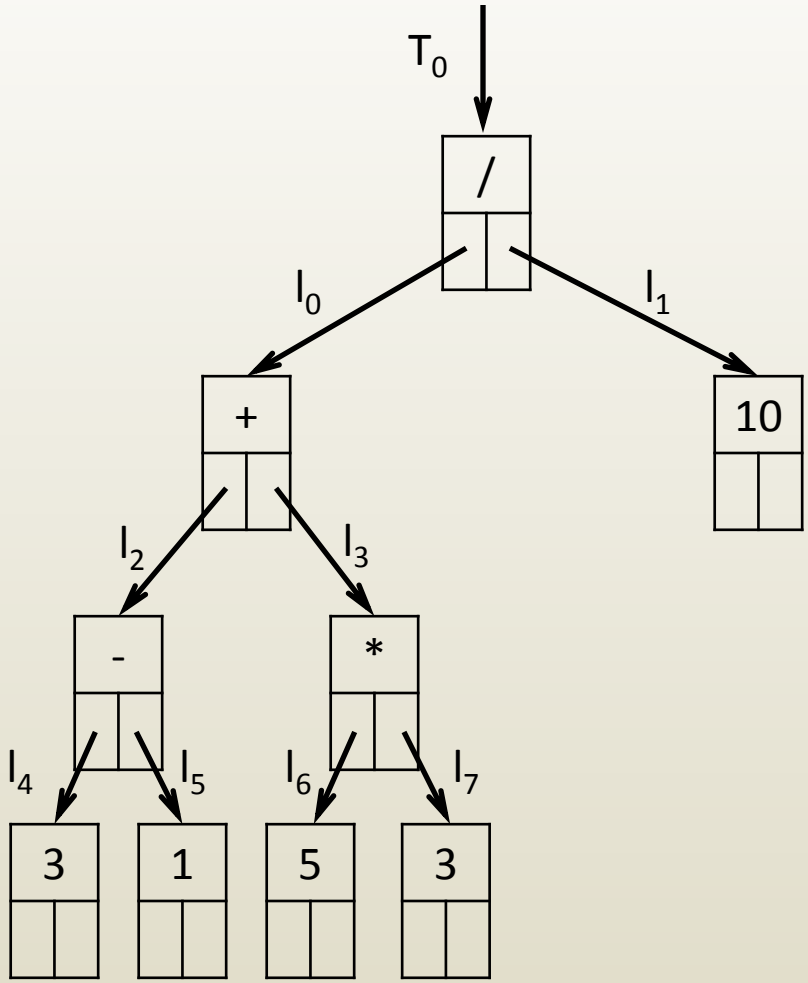
endif

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)-

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

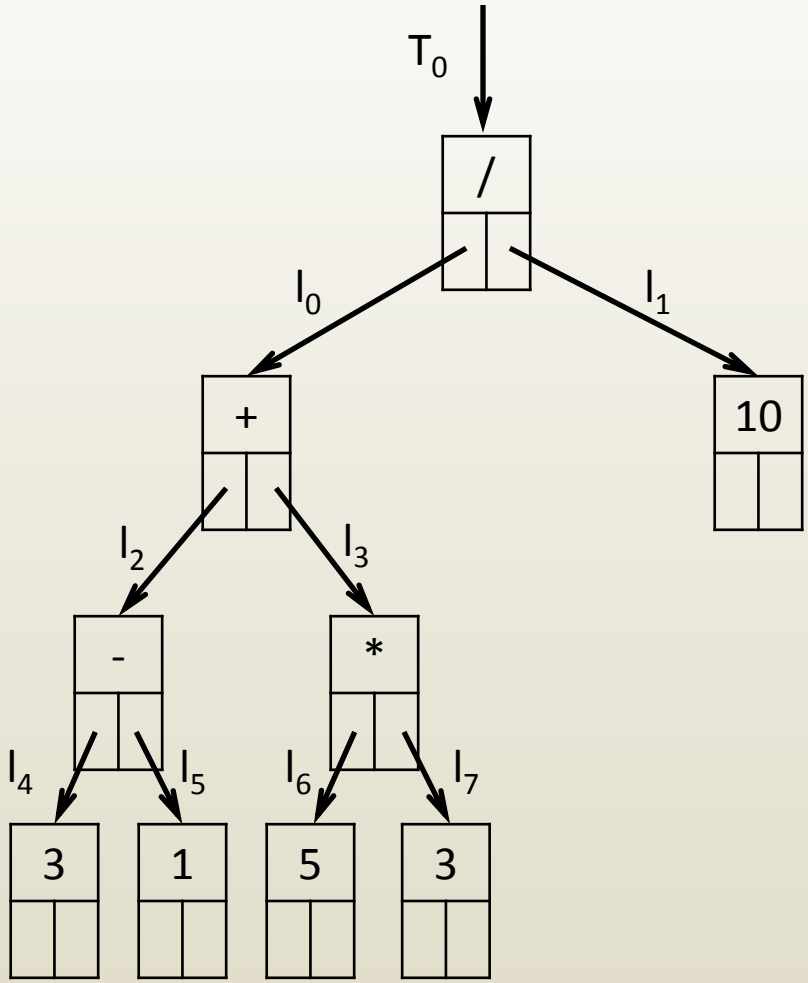
endif

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)-

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

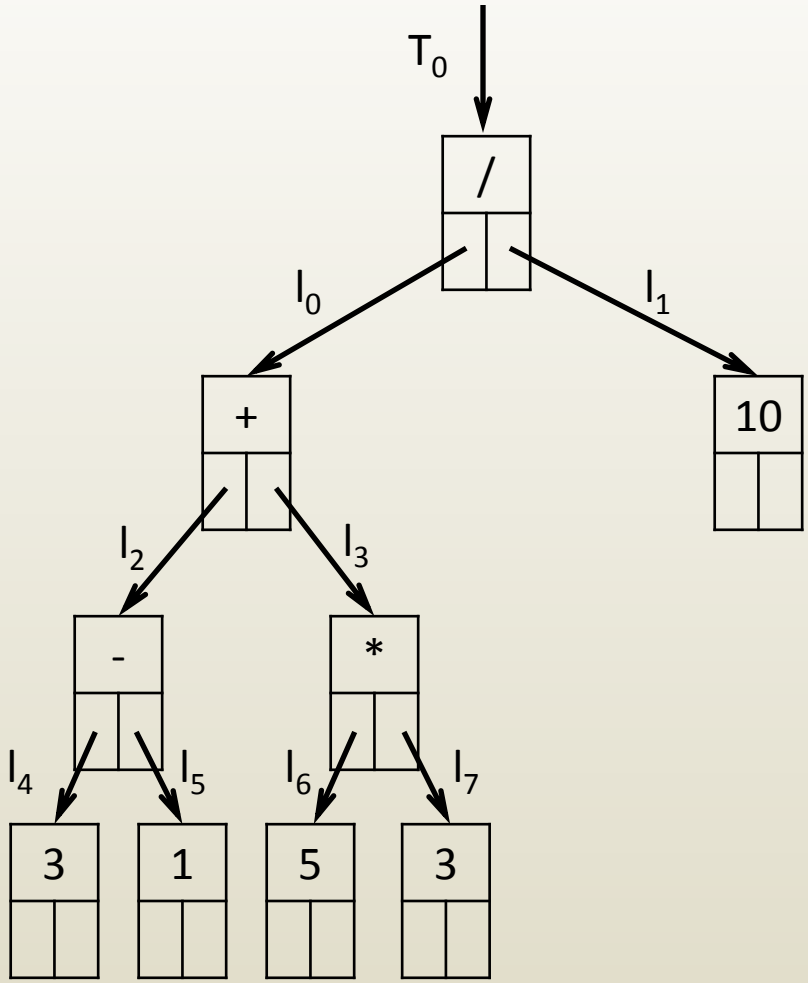
endif

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)

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

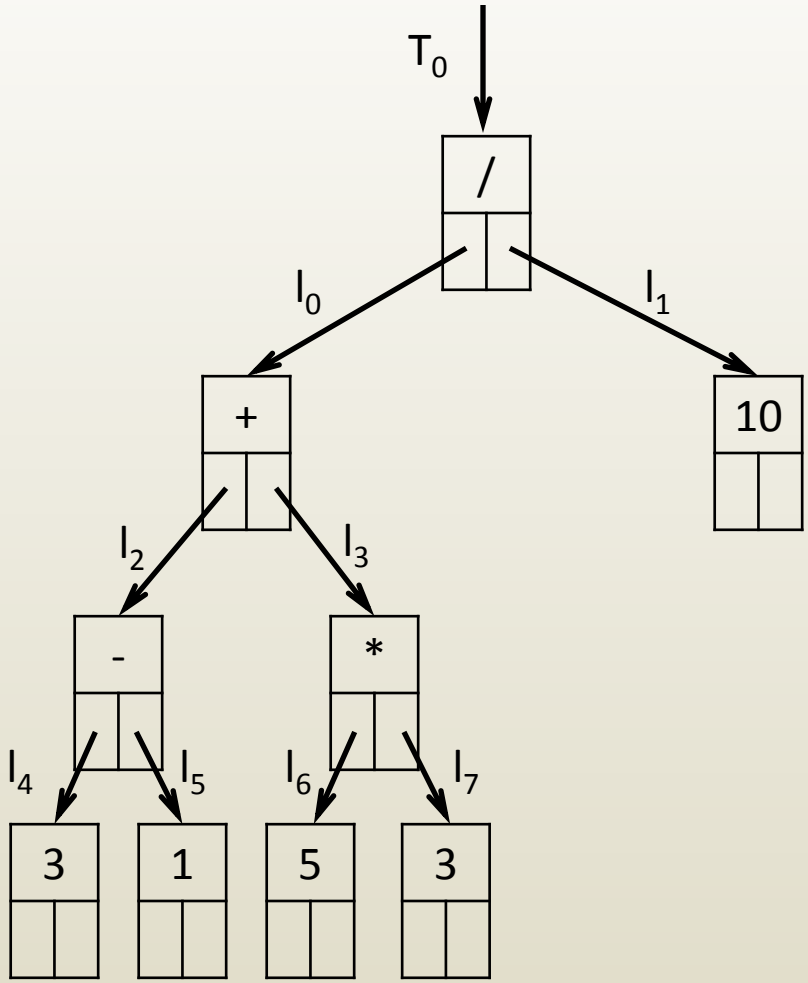
endif

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))

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

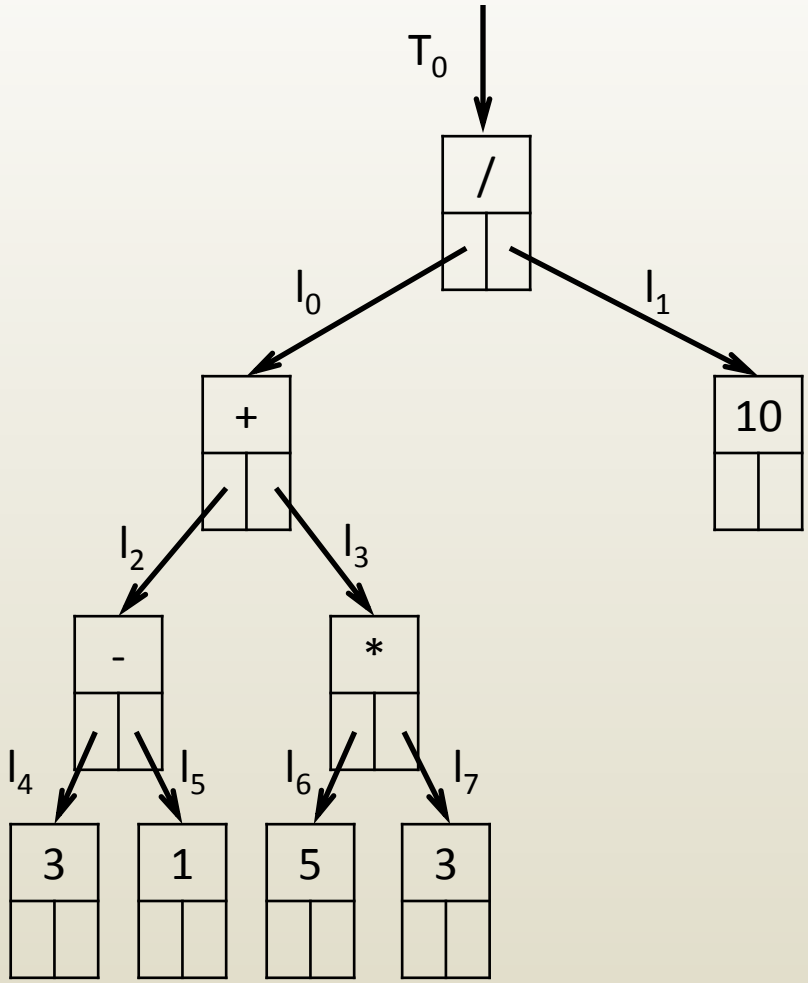
endif

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))** + PrintAEBRT(l_3)**)**

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

Printing arithmetic expression bin. tree

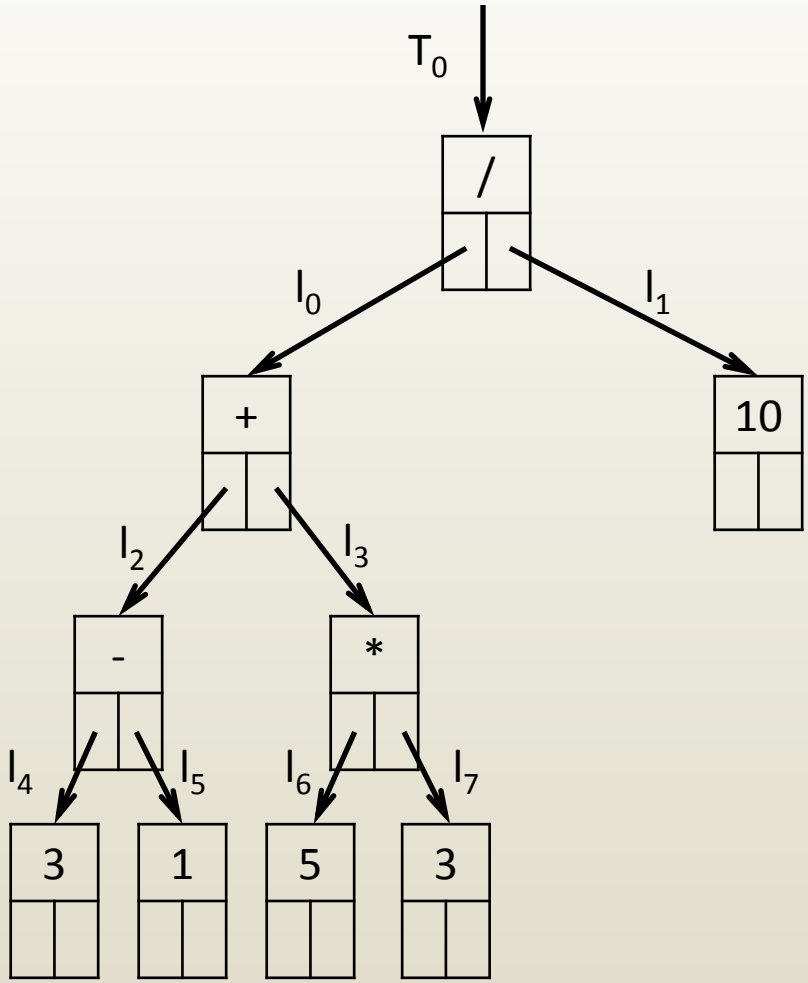
Input:

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(l_0) **/** PrintAEBRT(l_1)**)**
PrintAEBRT(l_0)	**(((3) – (1)) + ((5) * (3)))**

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

Printing arithmetic expression bin. tree

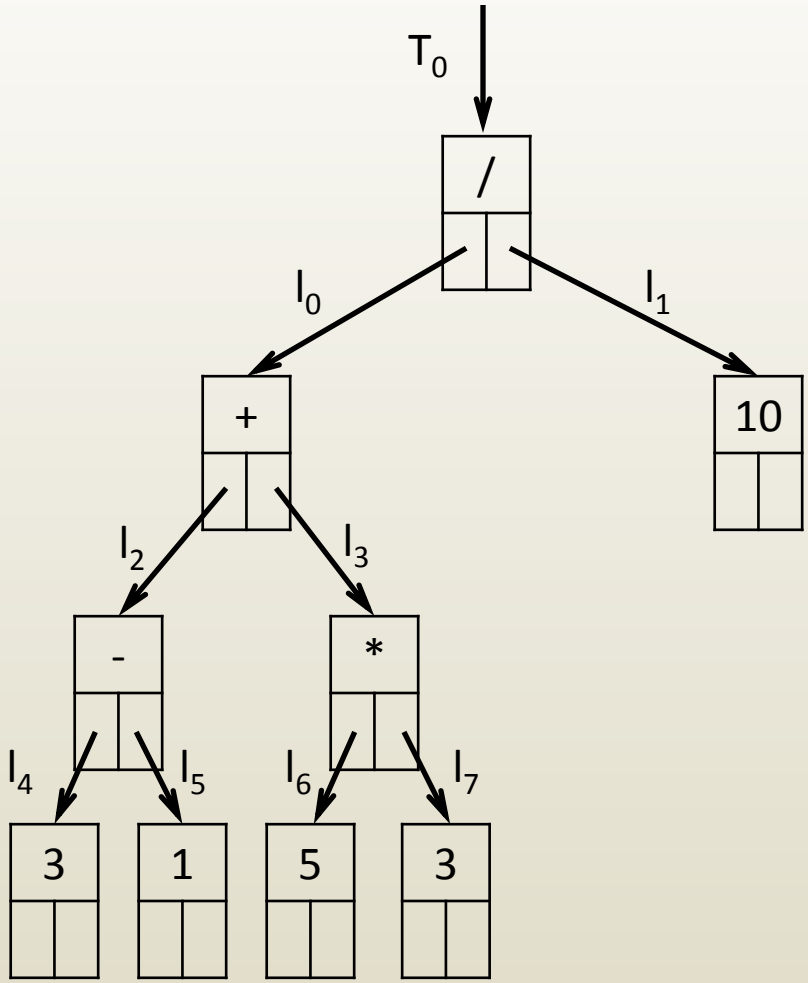
Input:

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)))/

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

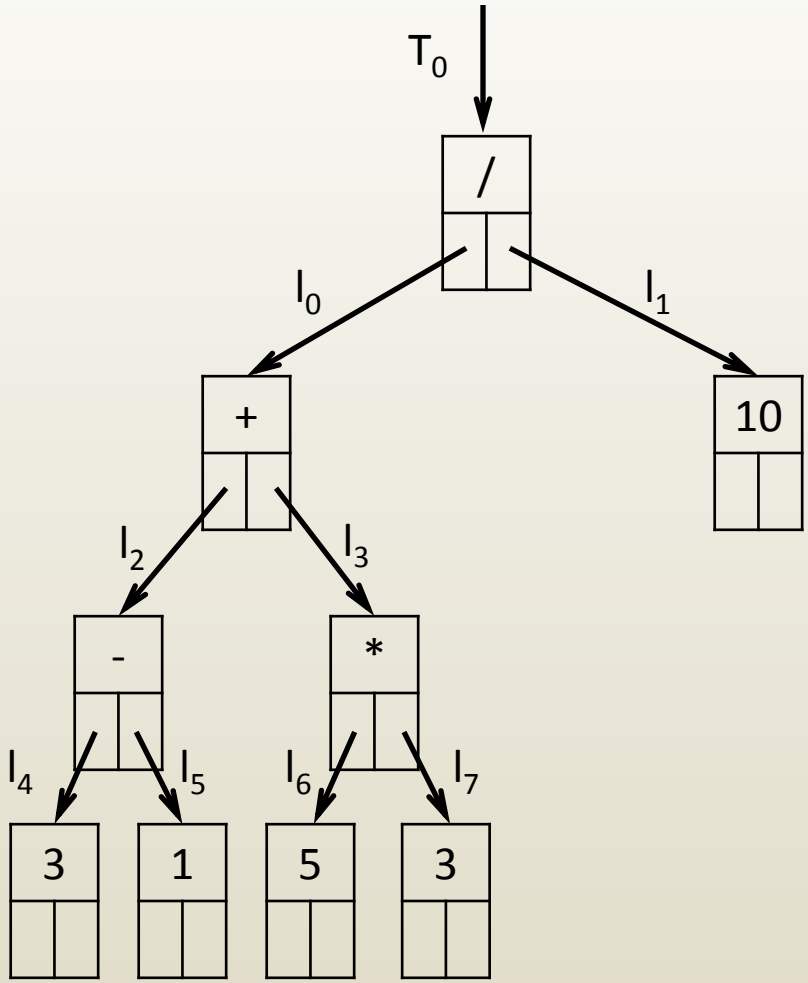
endif

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)

Printing arithmetic expression bin. tree

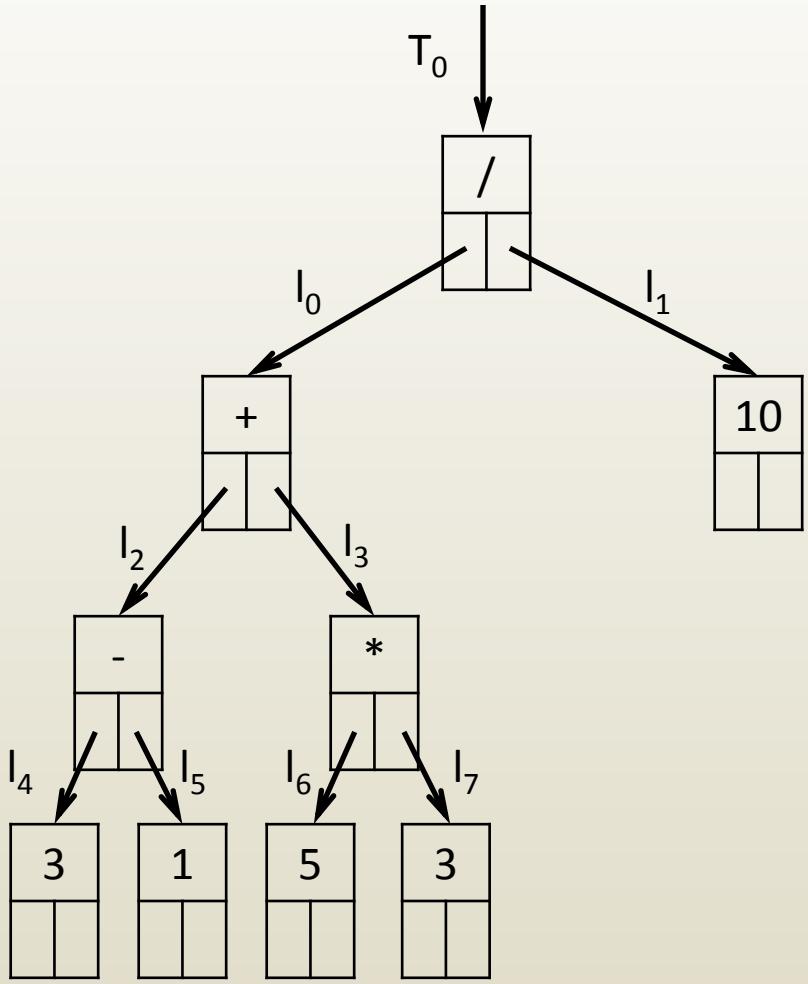
Input:

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))

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

Correct, but parentheses around operands are annoying

Printing arithmetic expression bin. tree

Input:

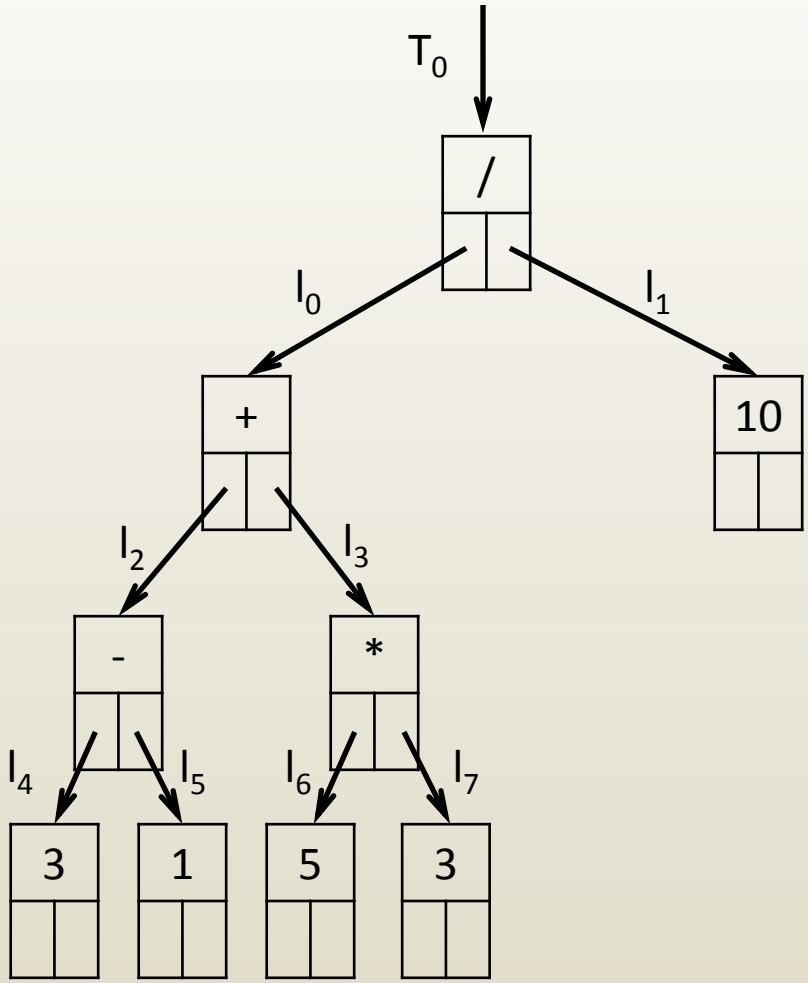
T // link to root of arithmetic expression binary tree

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