Tree-Traversal Orientation Analysis

Stephen Curial, Kevin Andrusky and José Nelson Amaral
University of Alberta

Overview

A compiler-based analysis to determine the predominant orientation of traversal of trees.

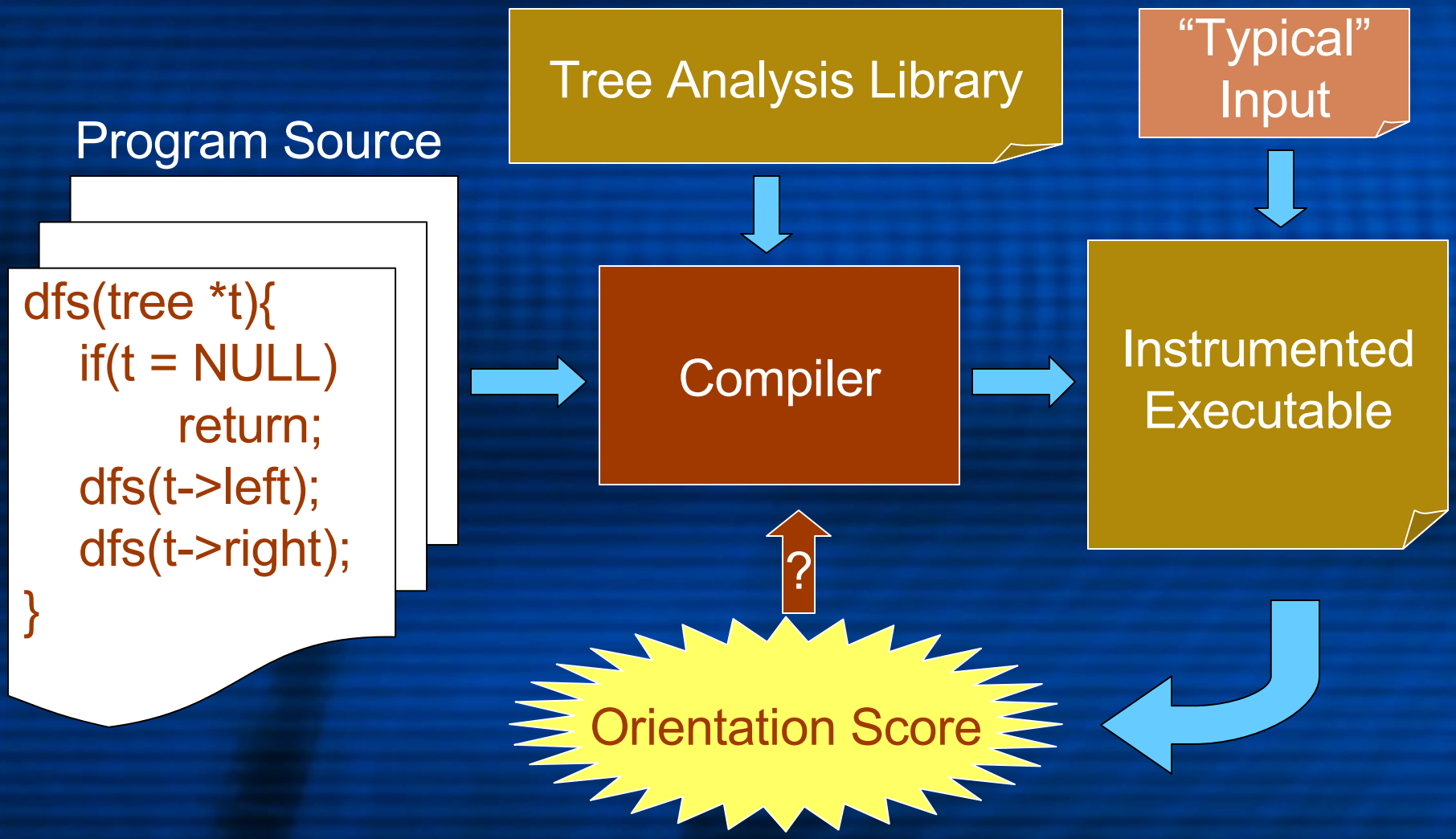
The compiler *automatically inserts instrumentation* into the the program to create an instrumented executable that *computes an orientation score* for each tree in the program.

Overview

A compiler-based analysis to determine the predominant orientation of traversal of trees.

The compiler *automatically inserts instrumentation* into the the program to create an instrumented executable that *computes an orientation score* for each tree in the program.

Overview


Program Source
dfs(tree *t){
if(t = NULL)
return;
dfs(t->left);
dfs(t->right);
}
Tree Analysis Library
"Typical" Input
Compiler
Instrumented Executable
?
Orientation Score

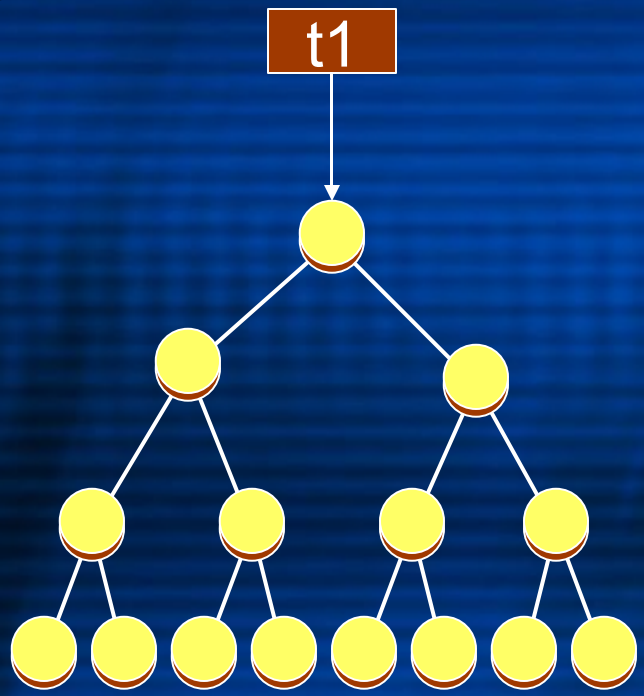
Tree-Traversal Orientation Analysis

Returns an *orientation score* $\in$ [-1,1]

- 1 is depth-first traversal
- -1 is breadth first traversal
- 0 is an unknown traversal

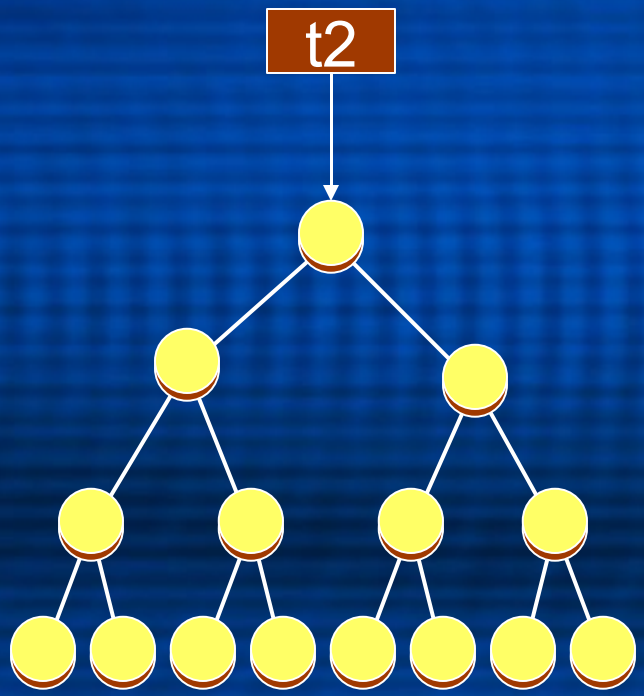
```
struct tree *t1, *t2;
init(t1,t2);
dfs(t1);
bfs(t2);
…
```

Overview



…

orientation score = 1.0



…

orientation score = -1.0

Calculating the Orientation Score

Every memory access to a tree is classified as:

- Depth-First
- Breadth-First
- Both
- Neither

The *orientation score* of each tree is calculated as:

$$\frac{num\_depth - num\_breadth}{num\_accesses} \quad [-1,1]$$

Calculating the Orientation Score

A linked-list access could be classified as both Breadth- and Depth-First.

- In this analysis a memory access to a 1-arity tree is classified as Depth-First.

- An alternative is to provide a three-dimensional score that also classifies linked-list traversal.

Determining if an Access is Depth-First

The TAL maintains a stack of *choice-points* for each tree.

Each choice-point consists of:

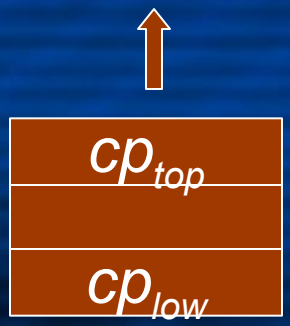
1. the address of a node, t, in the tree,
2. all n children of t, $c_1 \ldots c_n$, with a boolean representing if they have been visited, and
3. a boolean value representing if the node has been visited

Determining if a Access is Depth-First

cp_{top}: the address of the choice point on top of the stack;

t: the address of the tree node been referenced now;

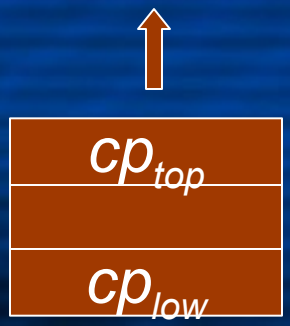
cp_{low}: the address of a choice point below cp_{top} in the stack;



Determining if a Access is Depth-First

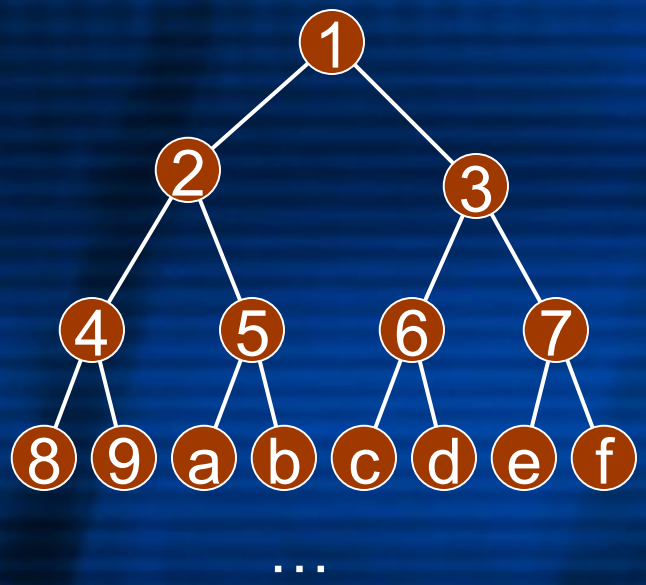
An access to tree node *t* is classified as depth-first if and only if:

- cp_{top} is equal to t; OR
- A child c_i of cp_{top} is equal to t; OR
- There exists a lower choice point in the stack cp_{low} such that cp_{low} *or a child of* cp_{low} is equal t AND all of the choice-points on the stack above cp_{low} have been visited.



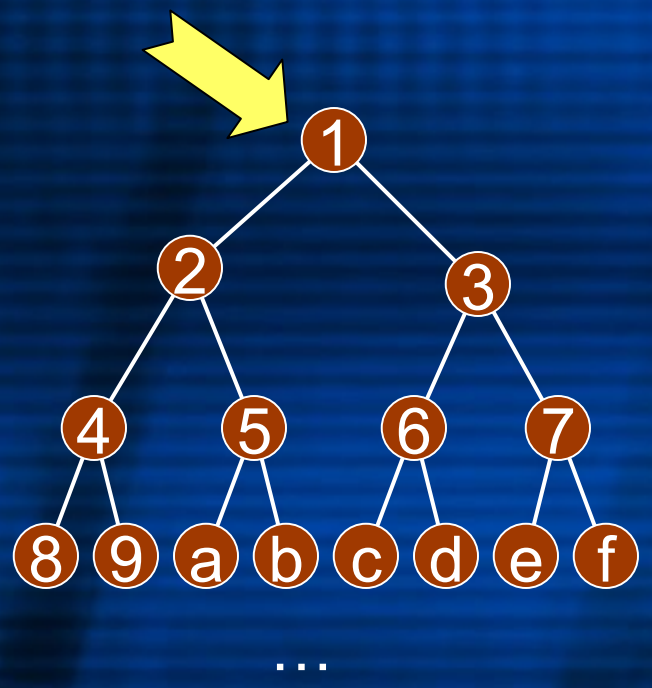
Determining if a Access is Depth-First

Every time a node is visited, a choice-point for that node is pushed onto the stack.



Determining if a Access is Depth-First

Every time a node is visited, a choice-point for that node is pushed onto the stack.



t = 1

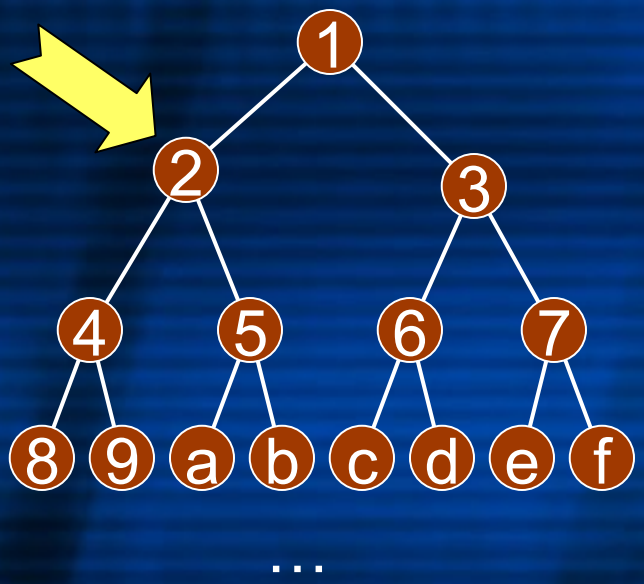
1 (2,3) c_{top}

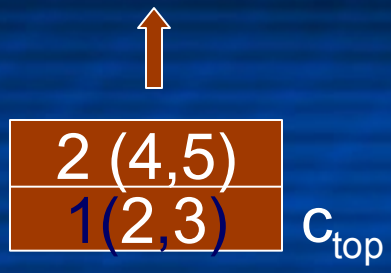
node (c1,c2)

Determining if a Access is Depth-First

Condition 2: $t = c_1 \rightarrow 2$ is DFS

t = 2



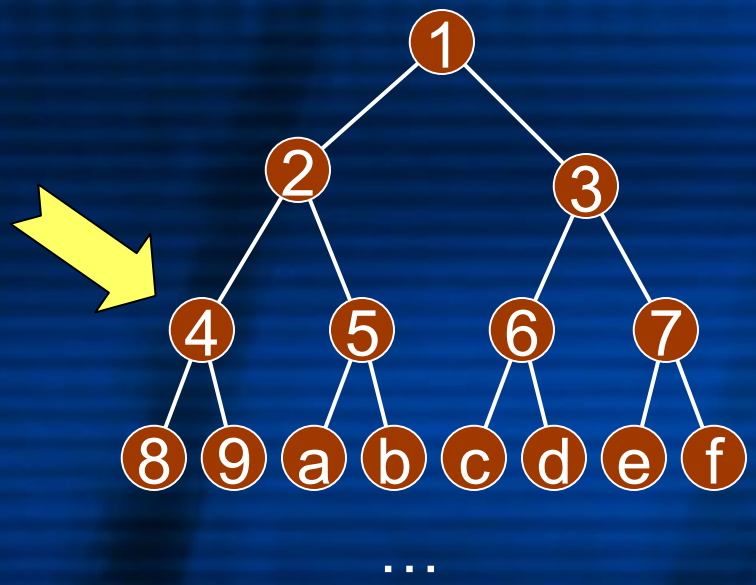


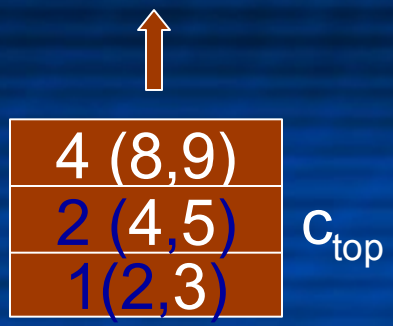
Dark color means visited

Determining if a Access is Depth-First

Condition 2: $t = c_1 \rightarrow 4$ is DFS

t = 4

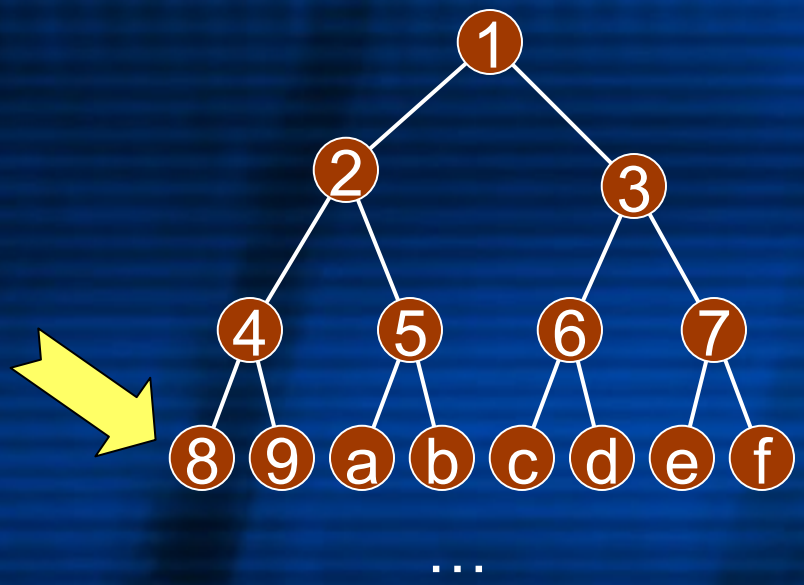


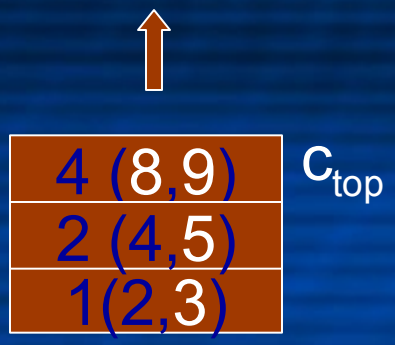


Determining if a Access is Depth-First

Condition 2: $t = c_1 \rightarrow 8$ is DFS

t = 8



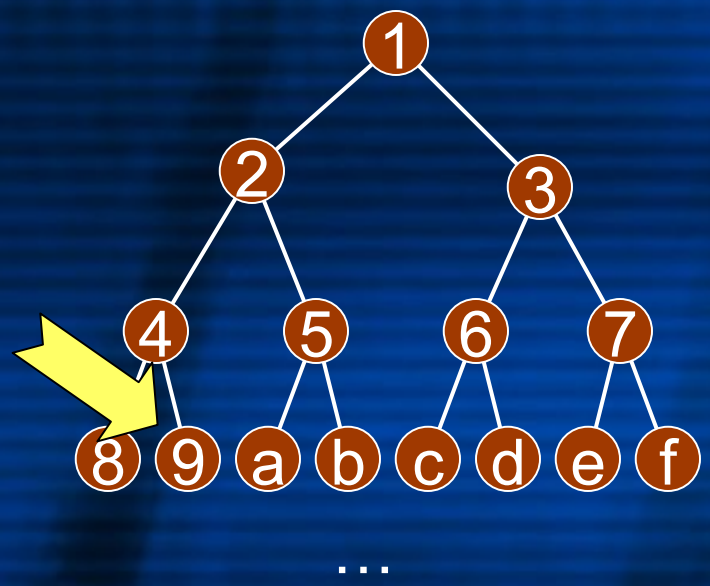


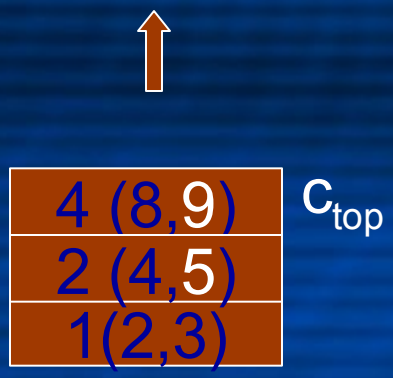
Note: Children of 8 are null thus 8 is not a choice point.

Determining if a Access is Depth-First

<u>Condition 2</u>: $t = c_2 \rightarrow 9$ is DFS

t = 9



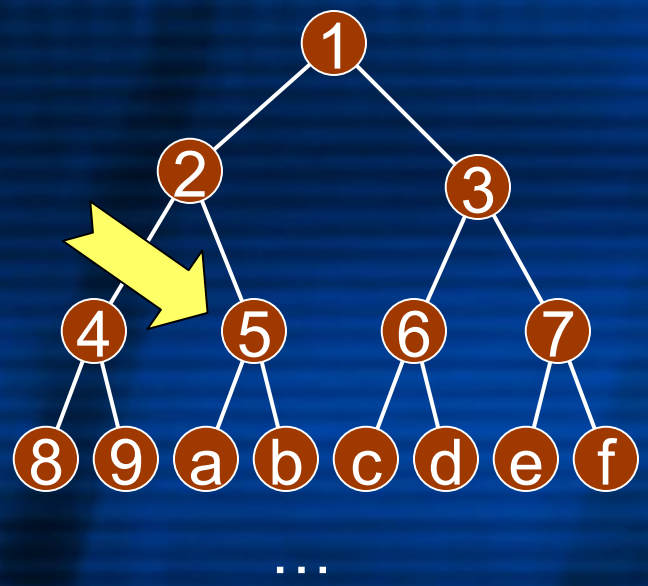


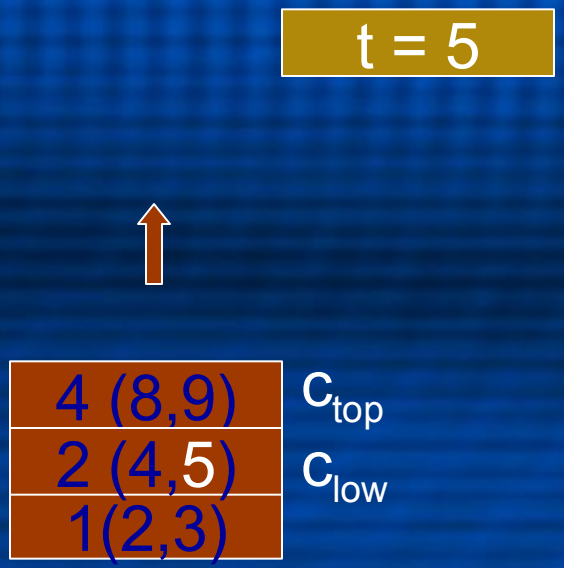
Note: Children of 9 are null thus 9 is not a choice point.

Determining if a Access is Depth-First

<u>Condition 3</u>: $t = c_{low}.c_2 \rightarrow 5$ is DFS

Pop everything above c_{low}

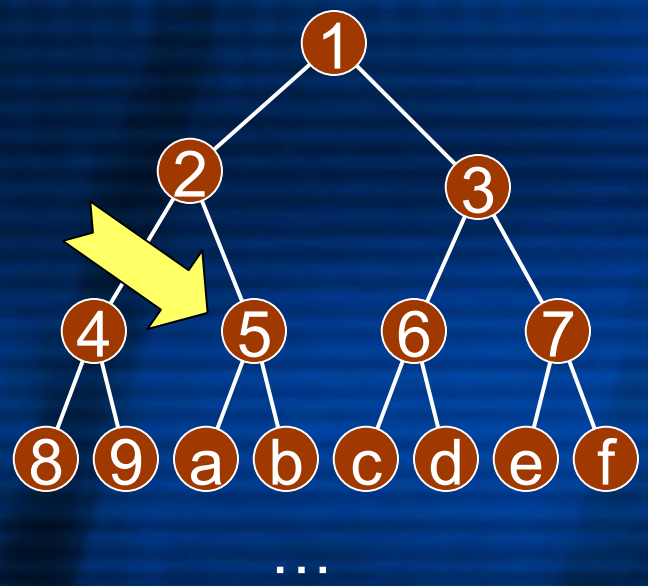


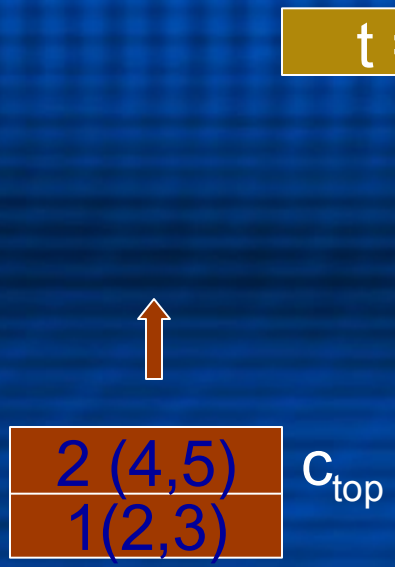


Determining if a Access is Depth-First

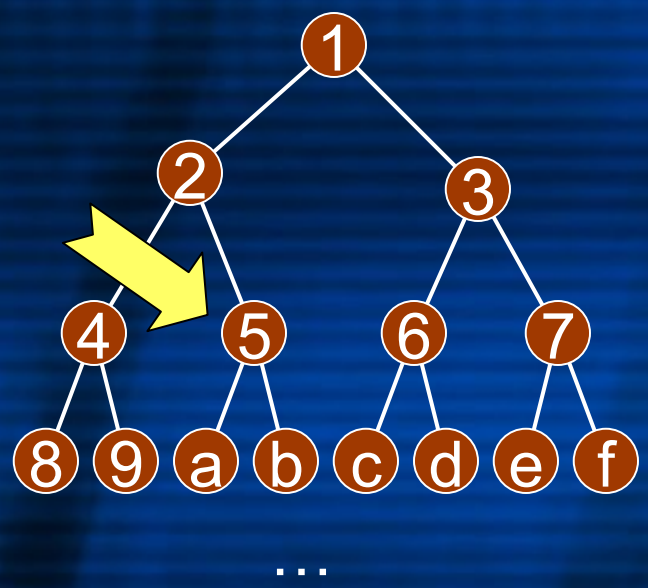
<u>Condition 3</u>: $t = c_{low}.c_2 \rightarrow 5$ is DFS

Pop everything above c_{low}

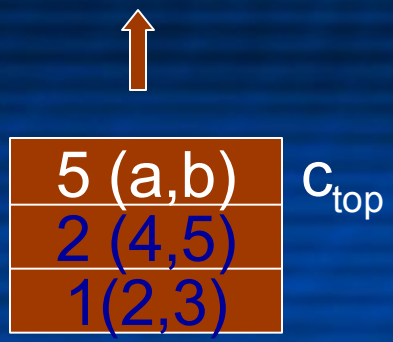




Determining if a Access is Depth-First

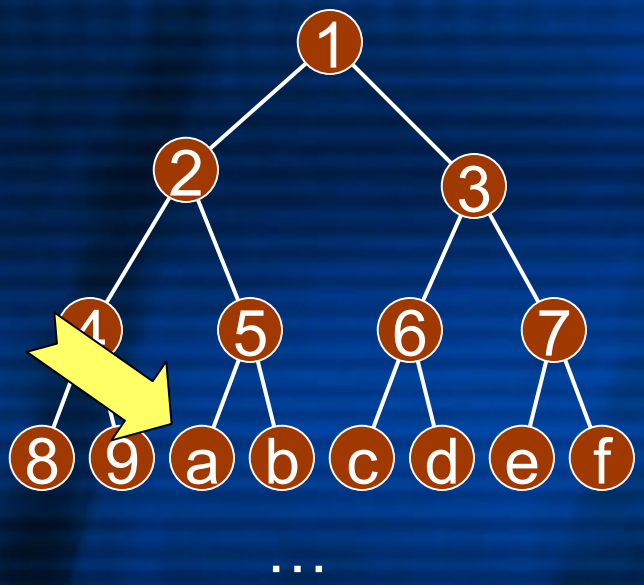


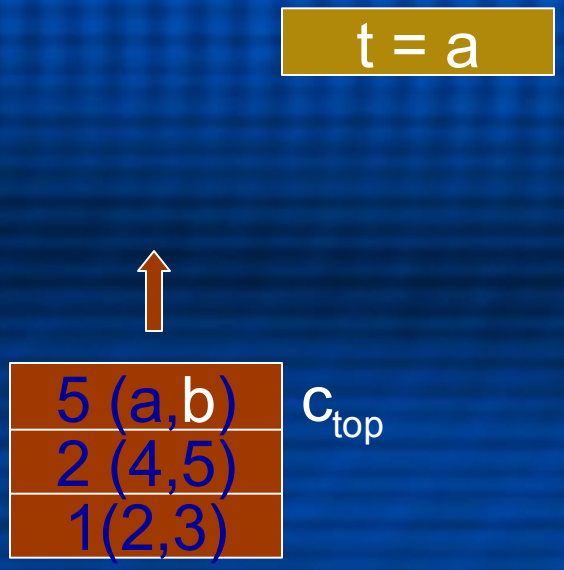
t = 5



Determining if a Access is Depth-First

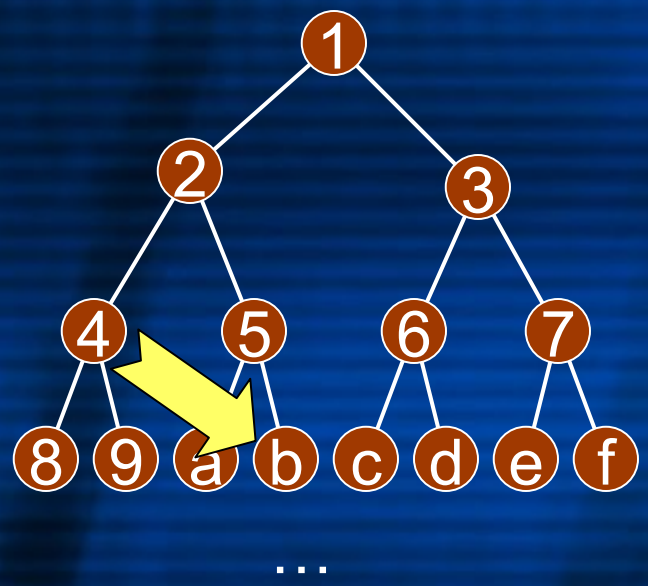
<u>Condition 2</u>: $t = c_1 \rightarrow a$ is DFS

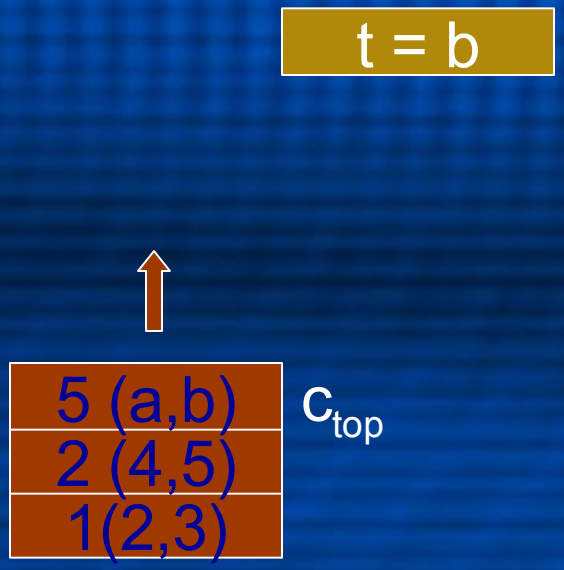




Determining if a Access is Depth-First

<u>Condition 2</u>: $t = c_2 \rightarrow b$ is DFS

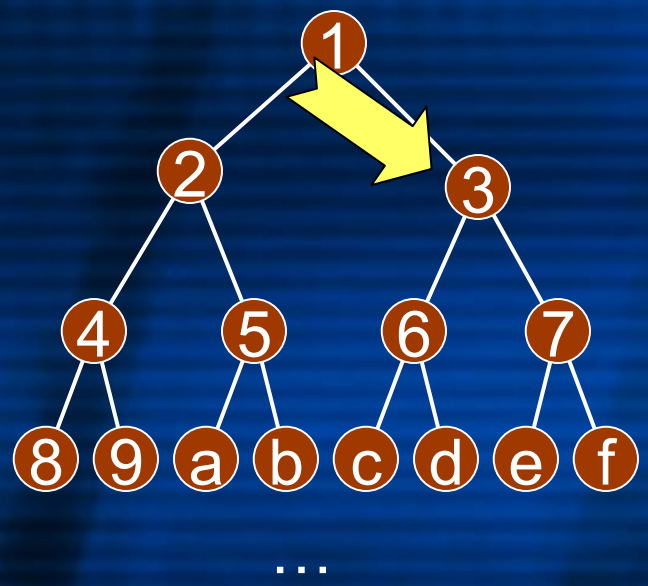


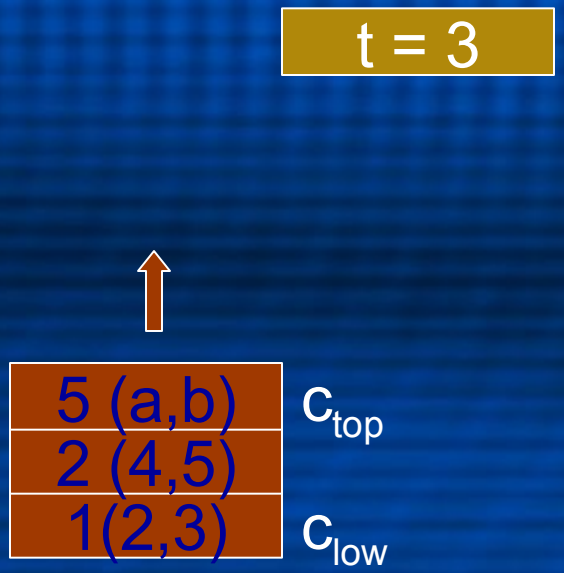


Determining if a Access is Depth-First

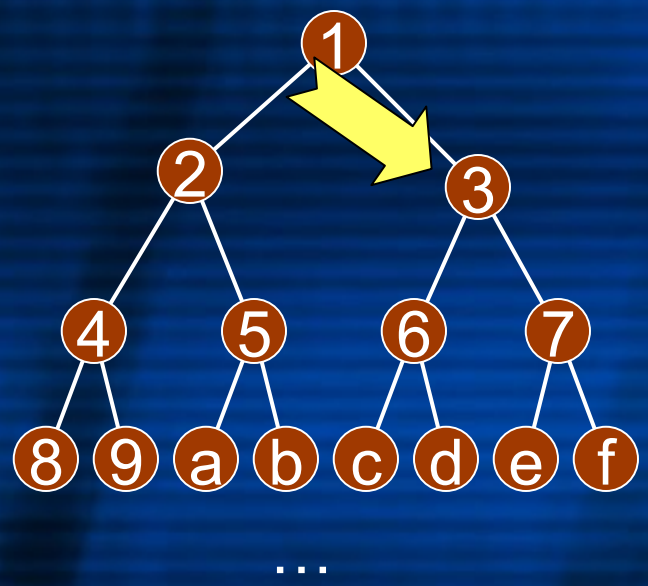
<u>Condition 3</u>: $t = c_{low}.c_2 \rightarrow b$ is DFS

Pop everything above c_{low}

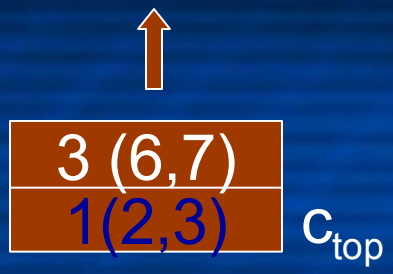




Determining if a Access is Depth-First



t = 3



Determining if a Access is Breadth-First

The TAL maintains an *open* and *next list* for each tree.

- They are stored as double-ended bit-vectors.

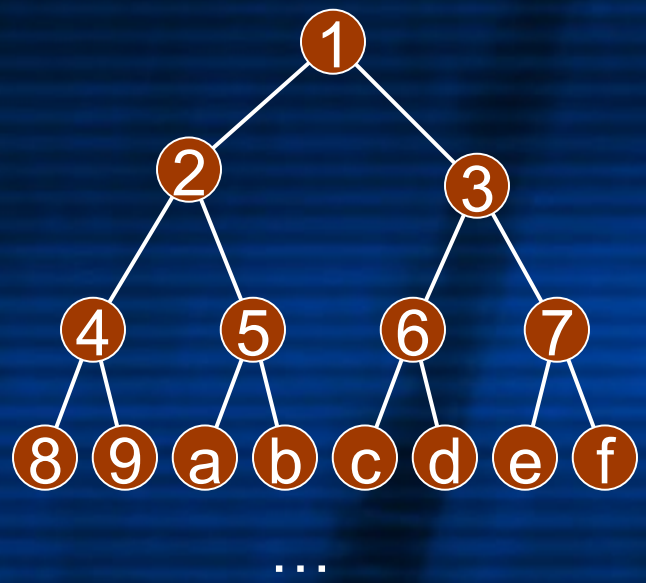
If an access is found in the open list it is classified as Breadth-First.

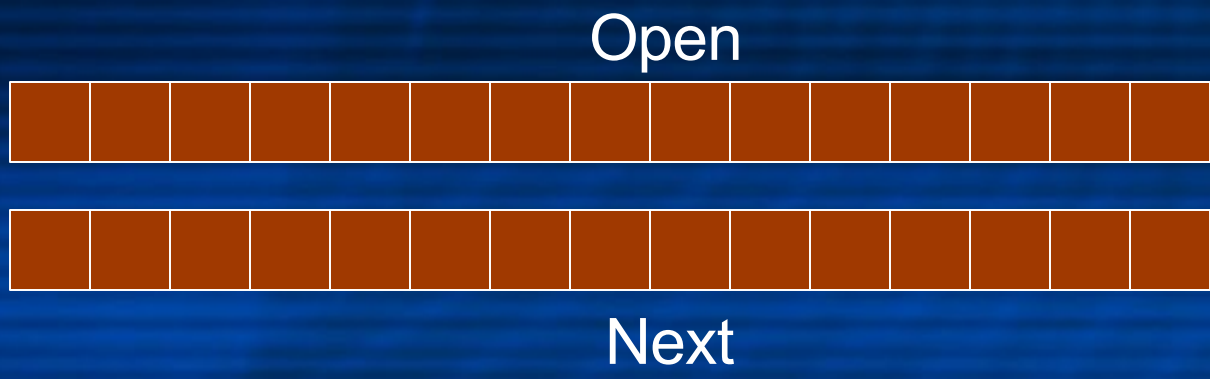
Open	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Next	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

Determining if a Access is Breadth-First

When tree node *t* is accessed:

- The children of *t* are added to the *next list.*
- If *t* is in the open list it is removed from the open list and the access is classified as breadth first.
- When the *open list* is empty it is swapped with the *next list.*

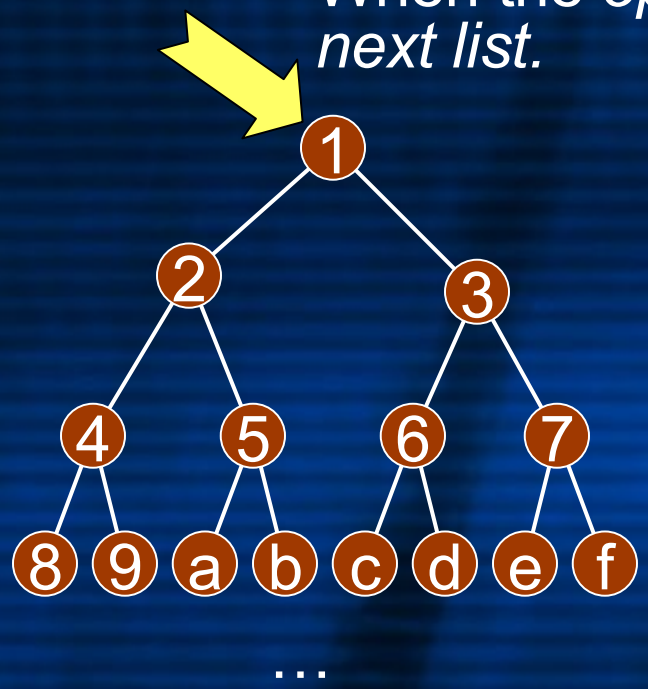


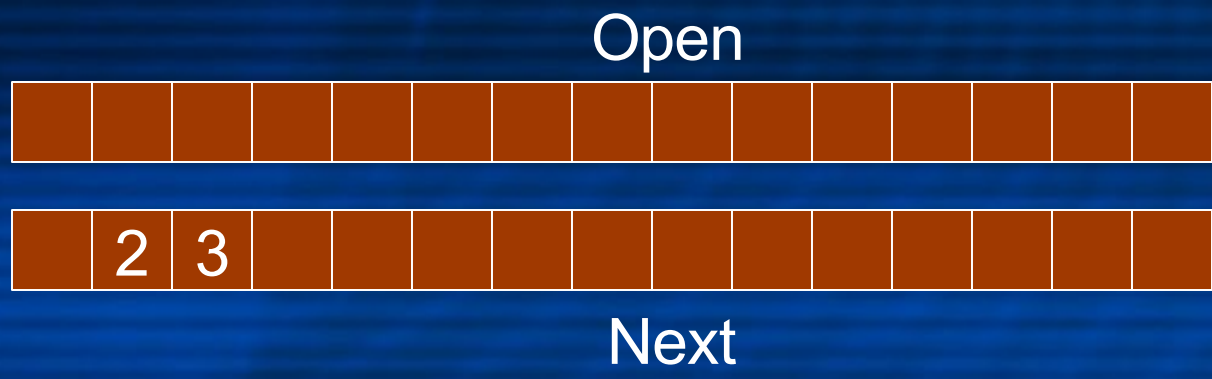


Determining if a Access is Breadth-First

When tree node *t* is accessed:

- The children of *t* are added to the *next list.*
- If *t* is in the open list it is removed from the open list and the access is classified as breadth first.
- When the *open list* is empty it is swapped with the *next list.*

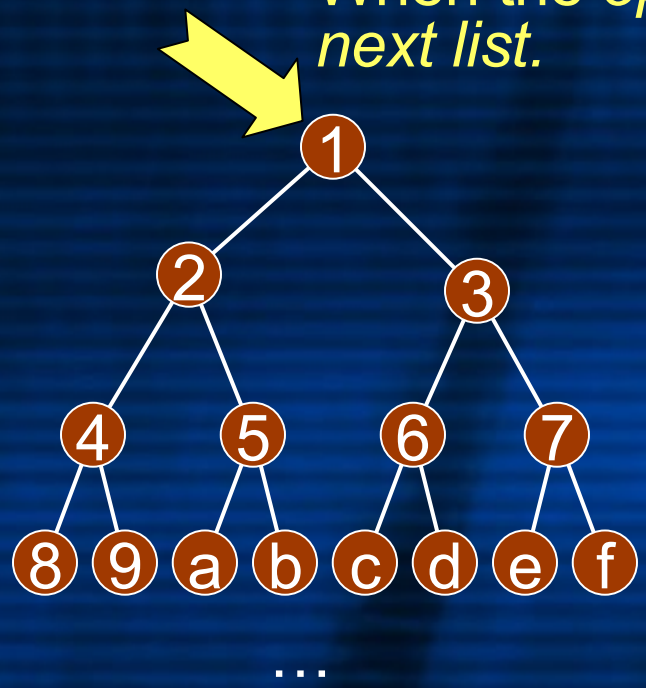


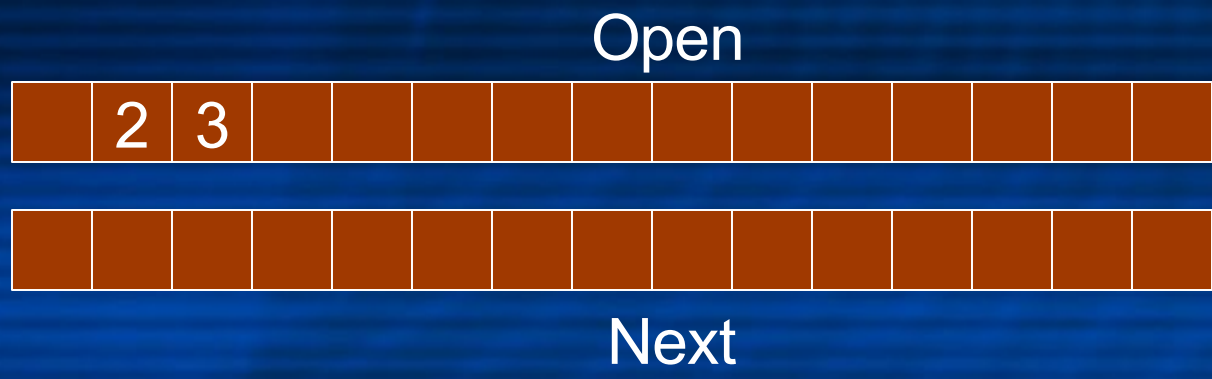


Determining if a Access is Breadth-First

When tree node *t* is accessed:

- The children of *t* are added to the *next list.*
- If *t* is in the open list it is removed from the open list and the access is classified as breadth first.
- When the *open list* is empty it is swapped with the *next list.*

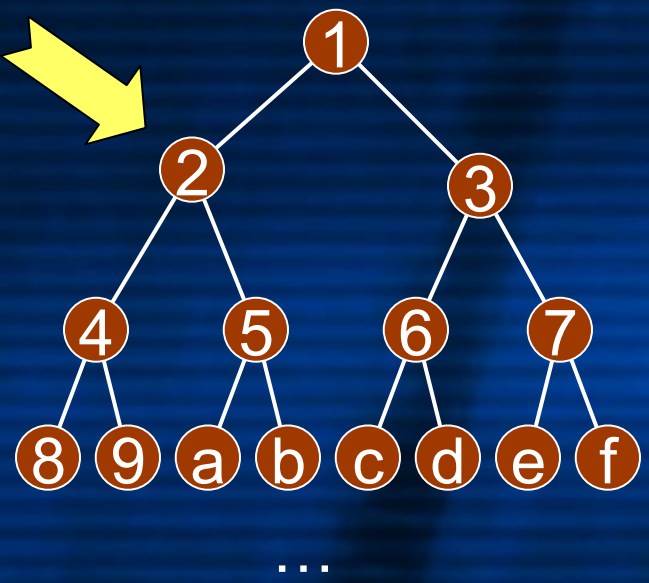


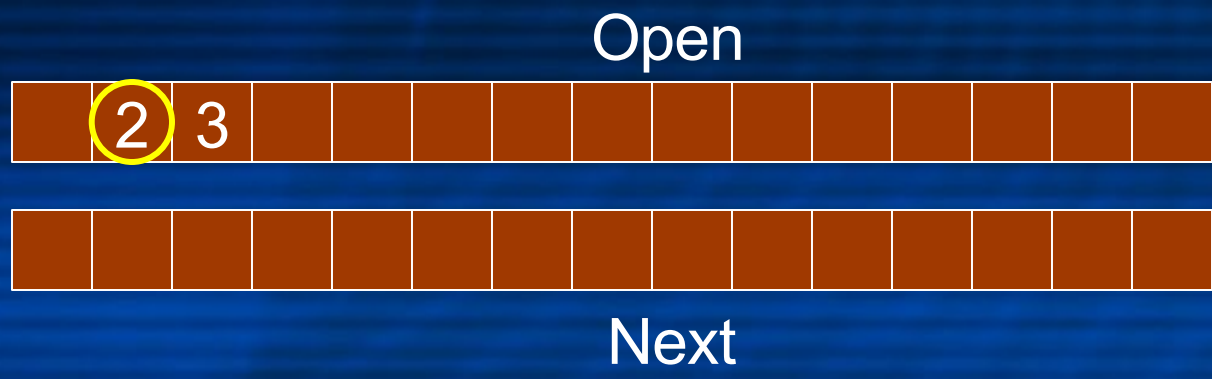


Determining if a Access is Breadth-First

When tree node *t* is accessed:

- The children of *t* are added to the *next list.*
- If *t* is in the open list it is removed from the open list and the access is classified as breadth first.
- When the *open list* is empty it is swapped with the *next list.*



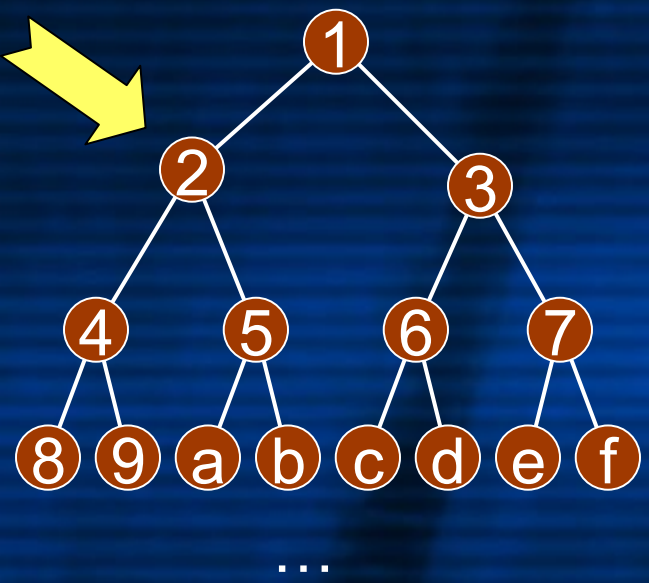


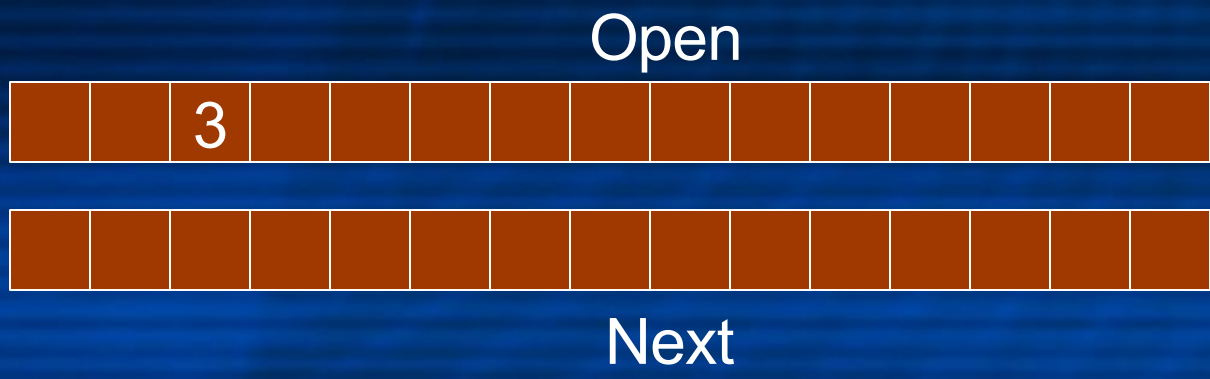
Access is classified as Breadth-First

Determining if a Access is Breadth-First

When tree node *t* is accessed:

- The children of *t* are added to the *next list.*
- If *t* is in the open list it is removed from the open list and the access is classified as breadth first.
- When the *open list* is empty it is swapped with the *next list.*

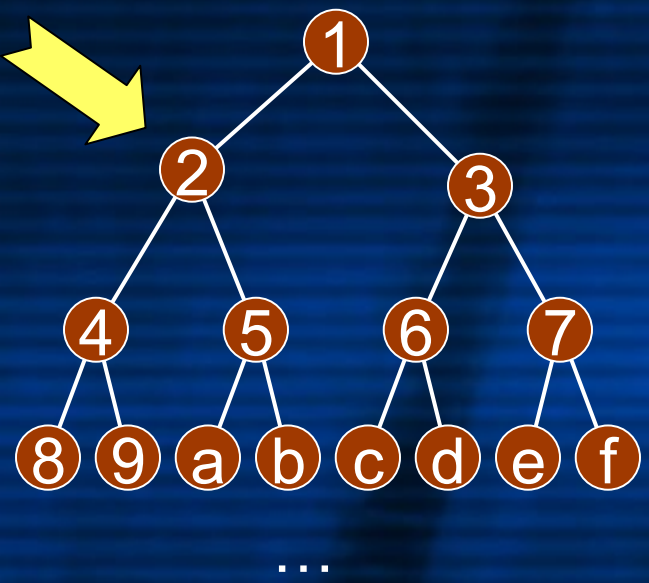


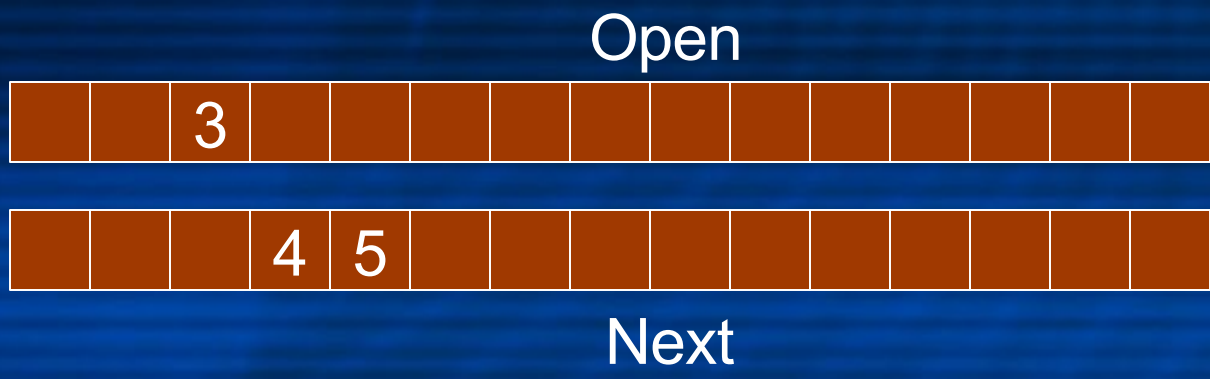


Determining if a Access is Breadth-First

When tree node *t* is accessed:

- The children of *t* are added to the *next list.*
- If *t* is in the open list it is removed from the open list and the access is classified as breadth first.
- When the *open list* is empty it is swapped with the *next list.*

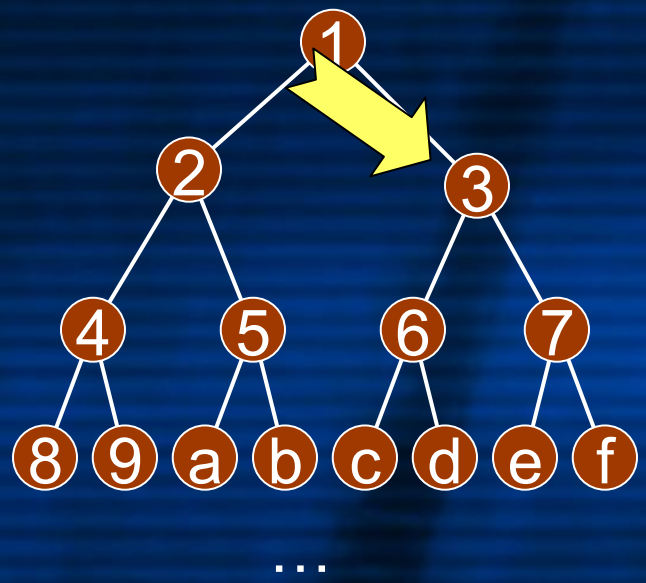




Determining if a Access is Breadth-First

When tree node *t* is accessed:

- The children of *t* are added to the *next list.*
- If *t* is in the open list it is removed from the open list and the access is classified as breadth first.
- When the *open list* is empty it is swapped with the *next list.*

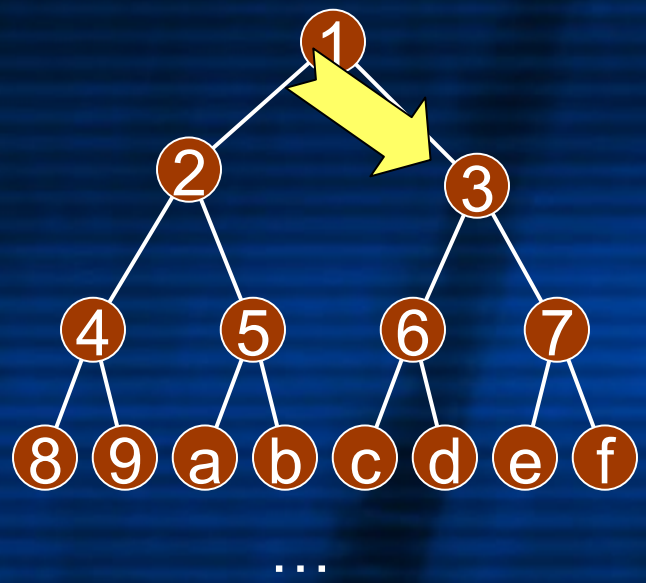


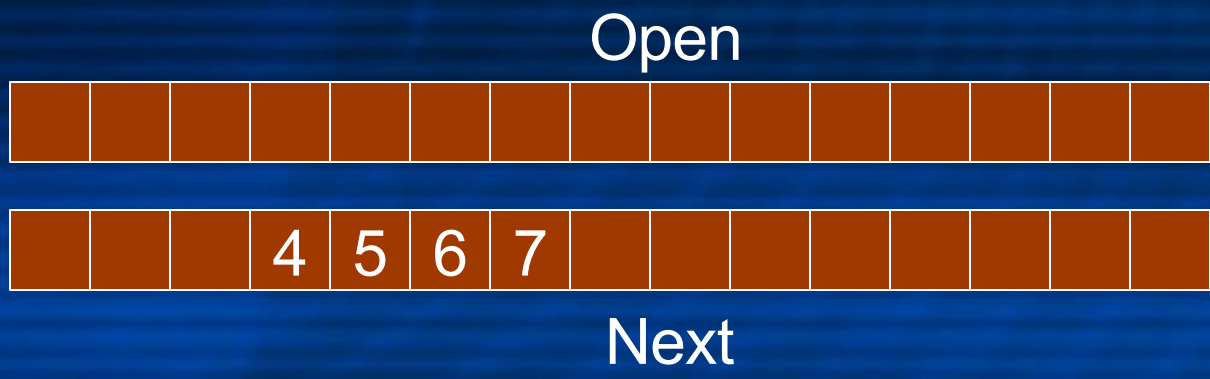
Access is classified as Breadth-First

Determining if a Access is Breadth-First

When tree node *t* is accessed:

- The children of *t* are added to the *next list.*
- If *t* is in the open list it is removed from the open list and the access is classified as breadth first.
- When the *open list* is empty it is swapped with the *next list.*

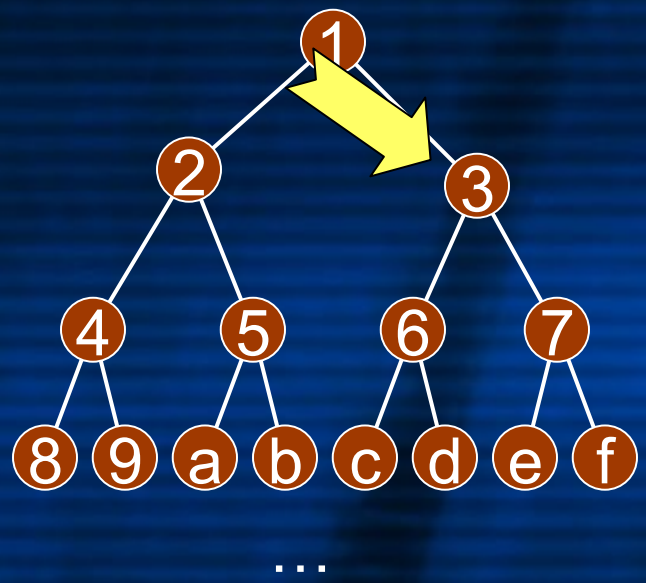




Determining if a Access is Breadth-First

When tree node *t* is accessed:

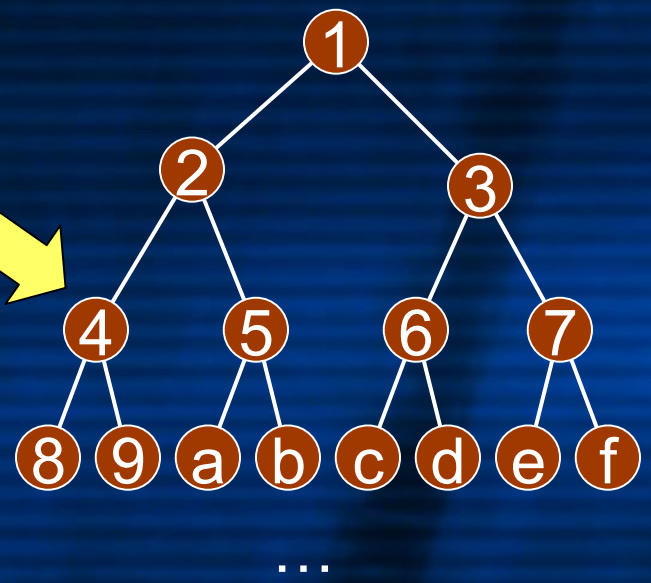
- The children of *t* are added to the *next list.*
- If *t* is in the open list it is removed from the open list and the access is classified as breadth first.
- When the *open list* is empty it is swapped with the *next list.*

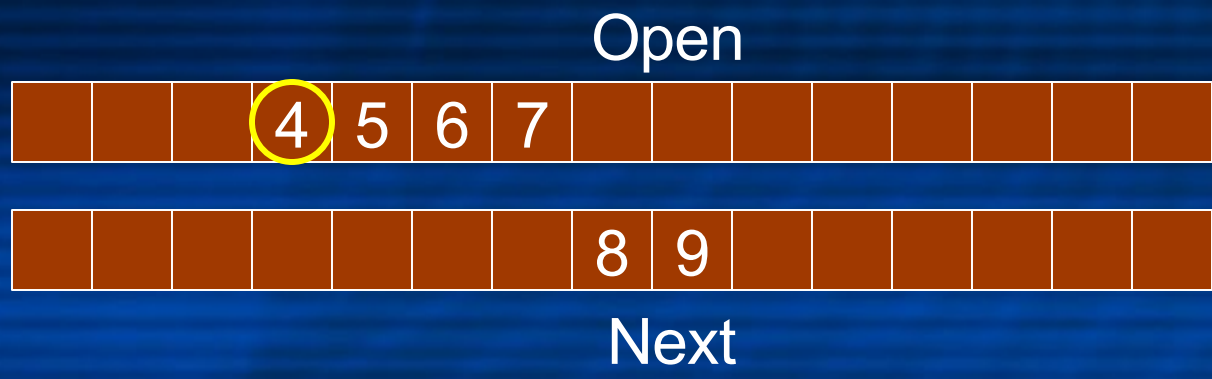


Determining if a Access is Breadth-First

When tree node *t* is accessed:

- The children of *t* are added to the *next list.*
- If *t* is in the open list it is removed from the open list and the access is classified as breadth first.
- When the *open list* is empty it is swapped with the *next list.*



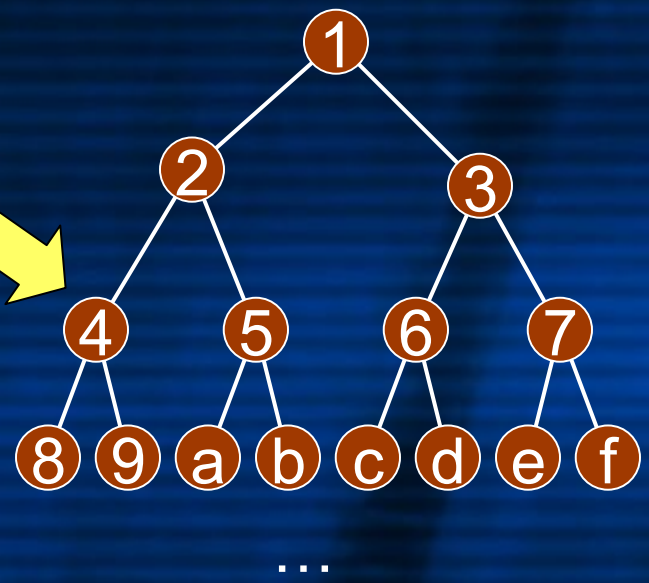


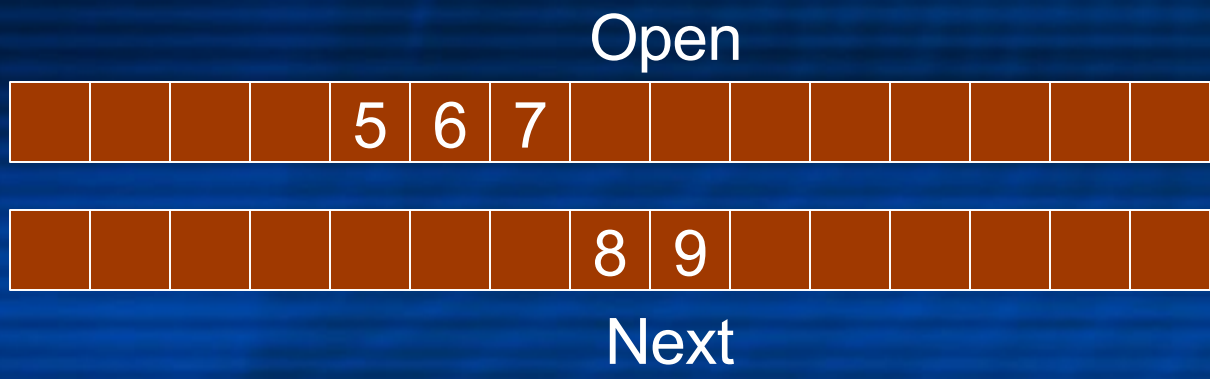
Access is classified as Breadth-First

Determining if a Access is Breadth-First

When tree node *t* is accessed:

- The children of *t* are added to the *next list.*
- If *t* is in the open list it is removed from the open list and the access is classified as breadth first.
- When the *open list* is empty it is swapped with the *next list.*





Implementation

- Built in the Open Research Compiler (ORC)
  - Operates on the WHIRL (Winning Hierarchical Intermediate Representation Language) IR.
  - Analysis requires Interprocedural Analysis (IPA).
    - Automatically identifies link-based tree data structures using Ghiya and Hendren's analysis [2] or programmer annotations.
  - Instrumentation can be inserted into each procedure without IPA.

Implementation

Calls our Tree Analysis Library (TAL)

`register_tree_access_with_children()`

Takes Parameters:

```
void *addr_deref
int tree_id
int num_children
void *child_addr, …
```

Implementation

- **Analysis is safe and will not cause a correct program to fail.**
  - **Replace calls to malloc/calloc/realloc with TALs wrapper functions.**
    - **If memory allocation fails, the analysis is abandoned and the memory used by the analysis is freed.**
  - **Memory address are calculated and accessed for the instrumentation in locations that are safe.**
    - **i.e. Will not dereference null pointers**

```
if( tree != NULL && tree->data == 1)
```

How do we identify tree references?

Example Code:

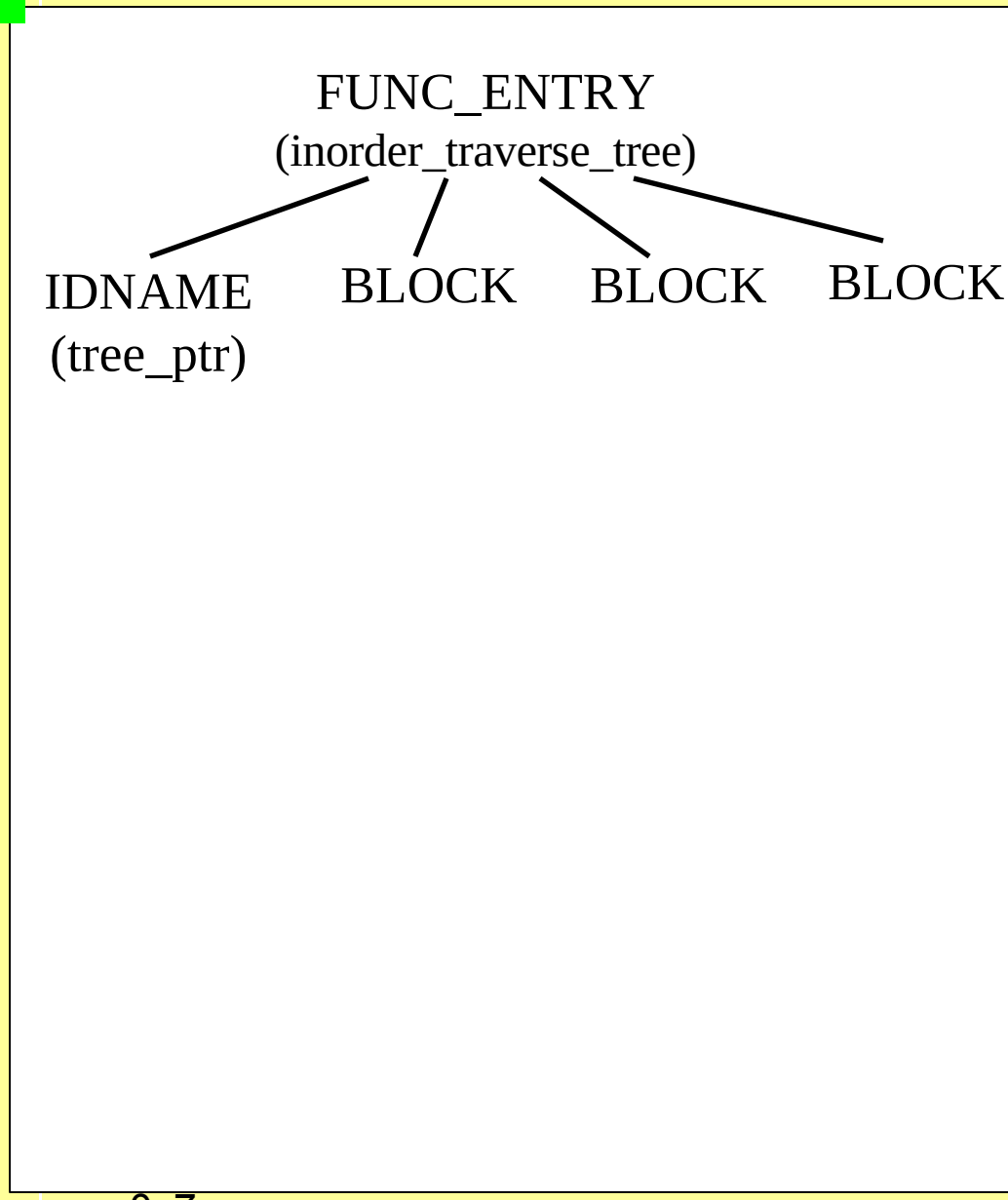
```
/* inorder tree traversal */
void inorder_traverse_tree(struct tn *tree_ptr)
{

  if(tree_ptr == NULL)
    return;

  inorder_traverse_tree(tree_ptr->left);
  tree_ptr->data++;
  inorder_traverse_tree(tree_ptr->right);

}
```

```
LOC 1 94 void inorder_traverse_tree(struct tn *tree_ptr)
 LOC 1 95 {
FUNC_ENTRY <1,25,inorder_traverse_tree>
 IDNAME 0 <2,1,tree_ptr>
BODY
 LOC 1 96   if(tree_ptr == NULL)
 IF
   U8U8LDID 0 <2,1,tree_ptr> T<32,anon_ptr.,8>
   U8INTCONST 0 (0x0)
  I4U8EQ
 THEN
  BLOCK
 LOC 1 97     return;
  RETURN
  END_BLOCK
 ELSE
 LOC 1 96
  BLOCK
  END_BLOCK
 END_IF
LOC 1 100
 LOC 1 101   inorder_traverse_tree(tree_ptr->left);
    U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>
   U8U8ILOAD 8 T<30,tn,8> T<31,anon_ptr.,8>
       <field_id:2>
  U8PARM 2 T<31,anon_ptr.,8> #  by_value
 VCALL 126 <1,25,inorder_traverse_tree> # flags 0x7e
```

```
void inorder_traverse_tree(struct tn
   *tree_ptr)
{ if(tree_ptr == NULL)
    return;
  inorder_traverse_tree(tree_ptr-
   >left);
  tree_ptr->data++;
  inorder_traverse_tree(tree_ptr-
   >right);
}
```

```
LOC 1 102   tree_ptr->data++;
    U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>
   I4I4ILOAD 0 T<30,tn,8> T<31,anon_ptr.,8> <field_id:1>
   I4INTCONST 1 (0x1)
  I4ADD
  U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>
 I4ISTORE 0 T<31,anon_ptr.,8> <field_id:1>
 LOC 1 103   inorder_traverse_tree(tree_ptr->right);
    U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>
   U8U8ILOAD 16 T<30,tn,8> T<31,anon_ptr.,8> <field_id:3>
  U8PARM 2 T<31,anon_ptr.,8> #  by_value
 VCALL 126 <1,25,inorder_traverse_tree> # flags 0x7e
 RETURN
 END_BLOCK
```

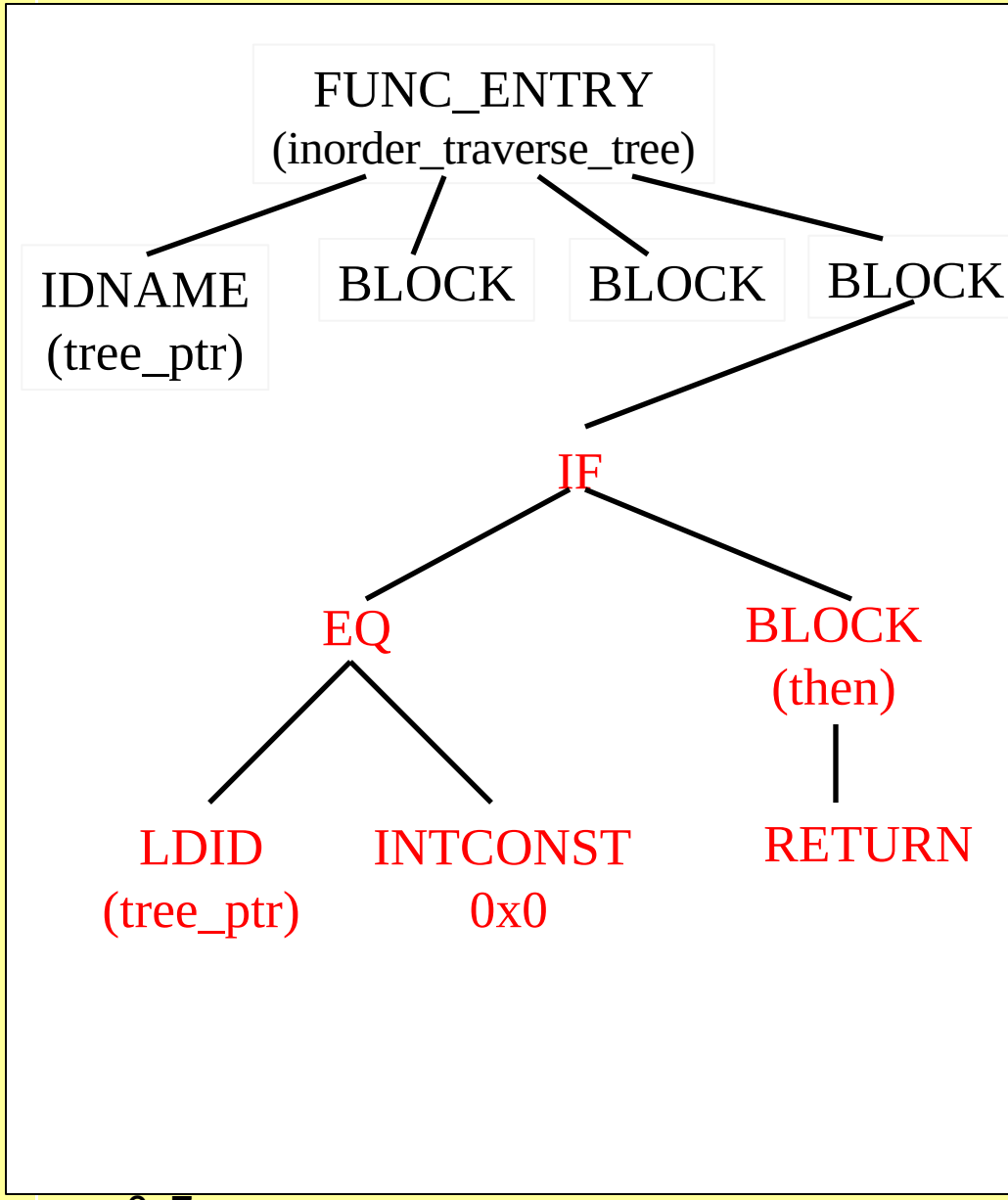
```
LOC 1 94  void inorder_traverse_tree(struct tn *tree_ptr)
 LOC 1 95 {
FUNC_ENTRY <1,25,inorder_traverse_tree>
 IDNAME 0 <2,1,tree_ptr>
BODY
 BLOCK
 END_BLOCK
 BLOCK
 END_BLOCK
 BLOCK
  U8U8LDID 0 <2,1,tree_ptr> T<32,anon_ptr.,8>
  U8INTCONST 0 (0x0)
 I4U8EQ
THEN
 BLOCK
LOC 1 97     return;
 RETURN
 END_BLOCK
ELSE
LOC 1 96
 BLOCK
 END_BLOCK
END_IF
```




```
    0x7e
RETURN
END_BLOCK
```

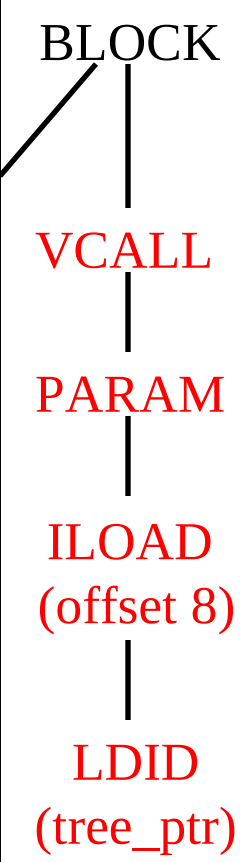
```
LOC 1 94 void inorder_traverse_tree(struct tn *tree_ptr)
 LOC 1 95 {
FUNC_ENTRY <1,25,inorder_traverse_tree>
 IDNAME 0 <2,1,tree_ptr>
BODY
 BLOCK
 END_BLOCK
 BLOCK
 END_BLOCK
 BLOCK
 PRAGMA 0 120 <null-st> 0 (0x0) # PREAMBLE_END
 LOC 1 96   if(tree_ptr == NULL)
 IF
  U8U8LDID 0 <2,1,tree_ptr> T<32,anon_ptr.,8>
```

```
LOC 1 96   if(tree_ptr == NULL)
 IF
  U8U8LDID 0 <2,1,tree_ptr> T<32,anon_ptr.,8>
  U8INTCONST 0 (0x0)
 I4U8EQ
 THEN
  BLOCK
 LOC 1 97     return;
  RETURN
  END_BLOCK
```



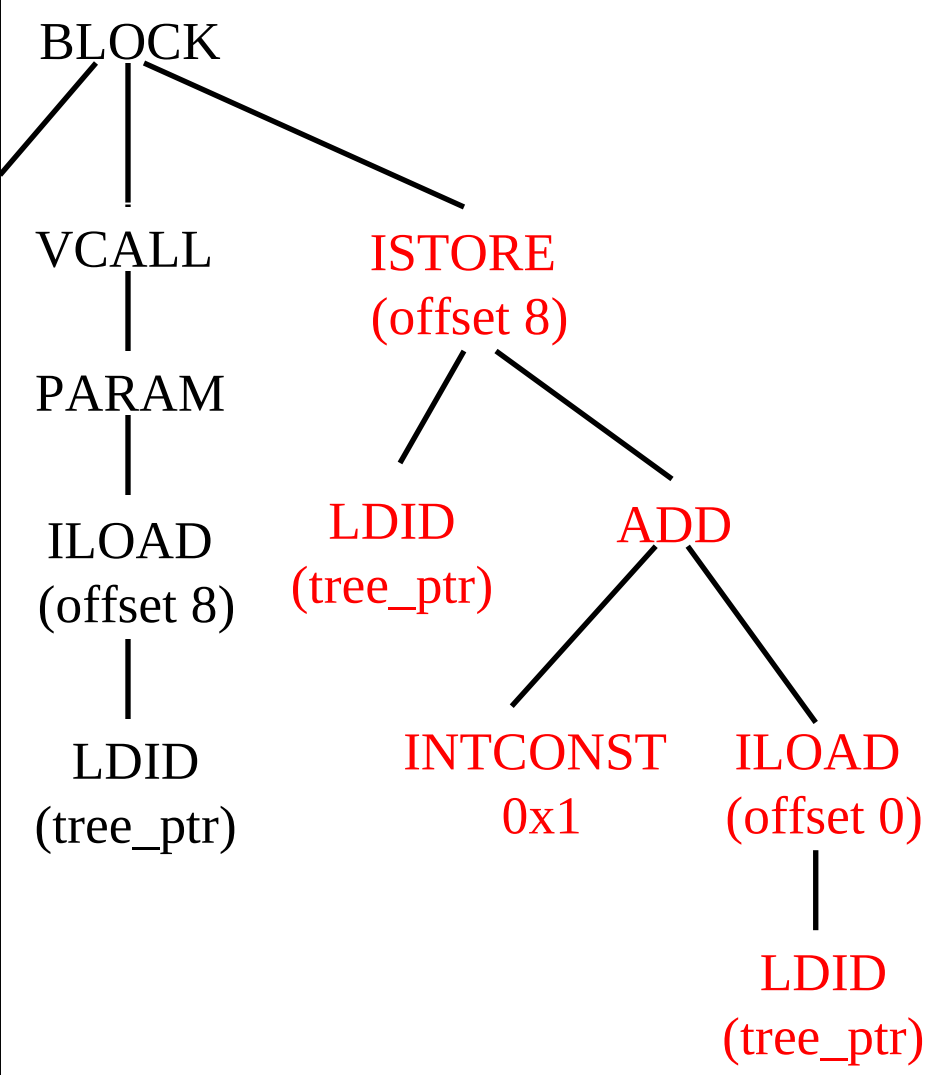

```
    0x7e
RETURN
END_BLOCK
```

LOC 1 94 void inorder_traverse_tree(struct tn *tree_ptr)




```
LOC 1 100
LOC 1 101  inorder_traverse_tree(tree_ptr->left);
   U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>
   U8U8ILOAD 8 T<30,tn,8> T<31,anon_ptr.,8>
      <field_id:2>
   U8PARM 2 T<31,anon_ptr.,8> #  by_value
   VCALL 126 <1,25,inorder_traverse_tree> # flags 0x7e
      0x7e
LOC 1 102   tree_ptr->data++;
   U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>
   I4I4ILOAD 0 T<30,tn,8> T<31,anon_ptr.,8>
      <field_id:1>
   I4INTCONST 1 (0x1)
   I4ADD
   U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>
I4ISTORE 0 T<31,anon_ptr.,8> <field_id:1>
LOC 1 103   inorder_traverse_tree(tree_ptr->right);
   U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>
   U8U8ILOAD 16 T<30,tn,8> T<31,anon_ptr.,8>
      <field_id:3>
   U8PARM 2 T<31,anon_ptr.,8> #  by_value
VCALL 126 <1,25,inorder_traverse_tree> # flags
      0x7e
RETURN
END_BLOCK
```

LOC 1 94 void inorder_traverse_tree(struct tn *tree_ptr)



LOC 1 100

LOC 1 101 inorder_traverse_tree(tree_ptr->left);

U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>

U8U8ILOAD 8 T<30,tn,8> T<31,anon_ptr.,8> <field_id:2>

U8PARM 2 T<31,anon_ptr.,8> # by_value

VCALL 126 <1,25,inorder_traverse_tree> # flags

LOC 1 102 tree_ptr->data++;

U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>

I4I4ILOAD 0 T<30,tn,8> T<31,anon_ptr.,8> <field_id:1>

I4INTCONST 1 (0x1)

I4ADD

U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>

I4ISTORE 0 T<31,anon_ptr.,8> <field_id:1>

I4ISTORE 0 T<31,anon_ptr.,8> <field_id:1>

LOC 1 103 inorder_traverse_tree(tree_ptr->right);

U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>

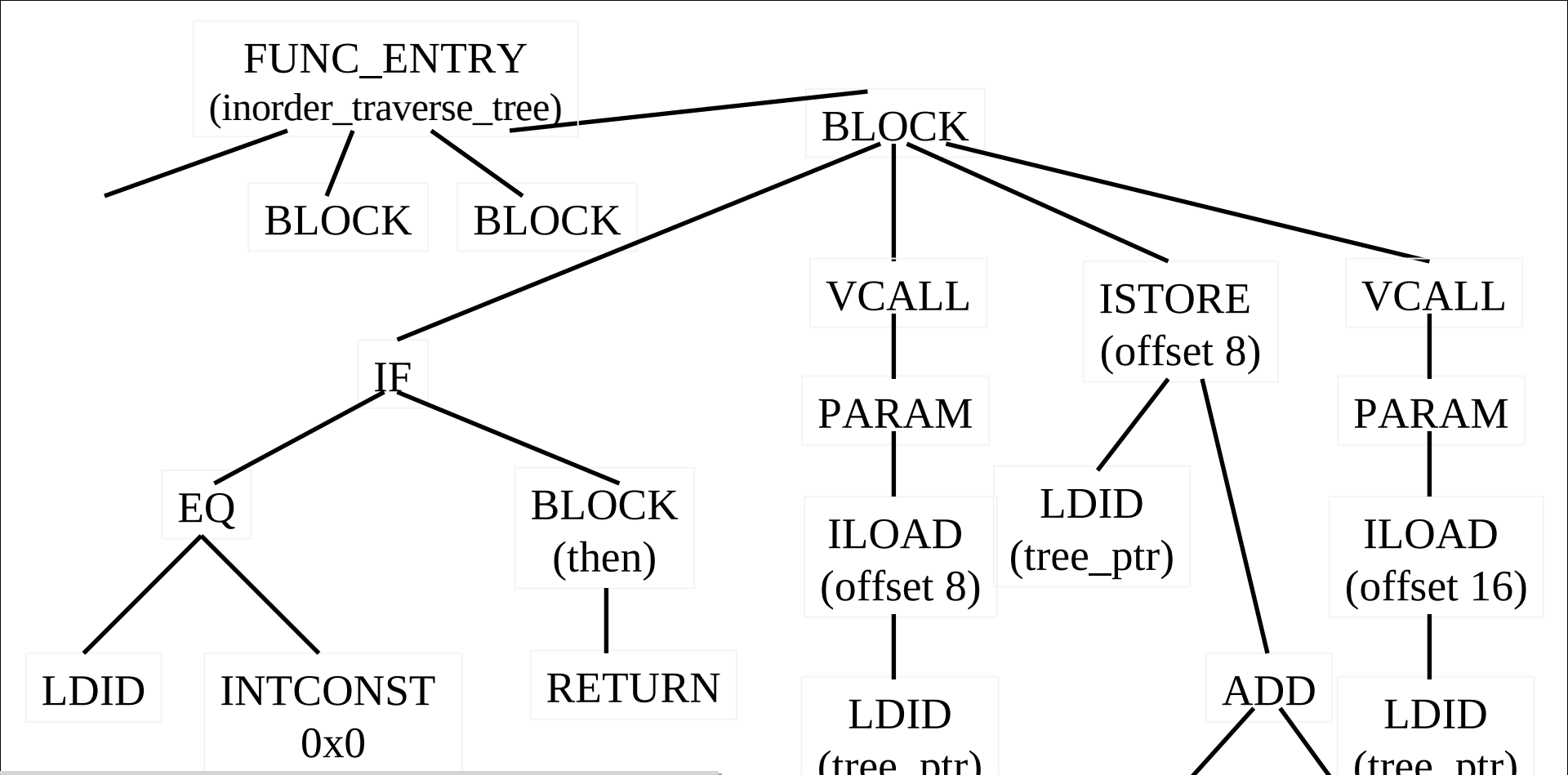
U8U8ILOAD 16 T<30,tn,8> T<31,anon_ptr.,8> <field_id:3>

U8PARM 2 T<31,anon_ptr.,8> # by_value

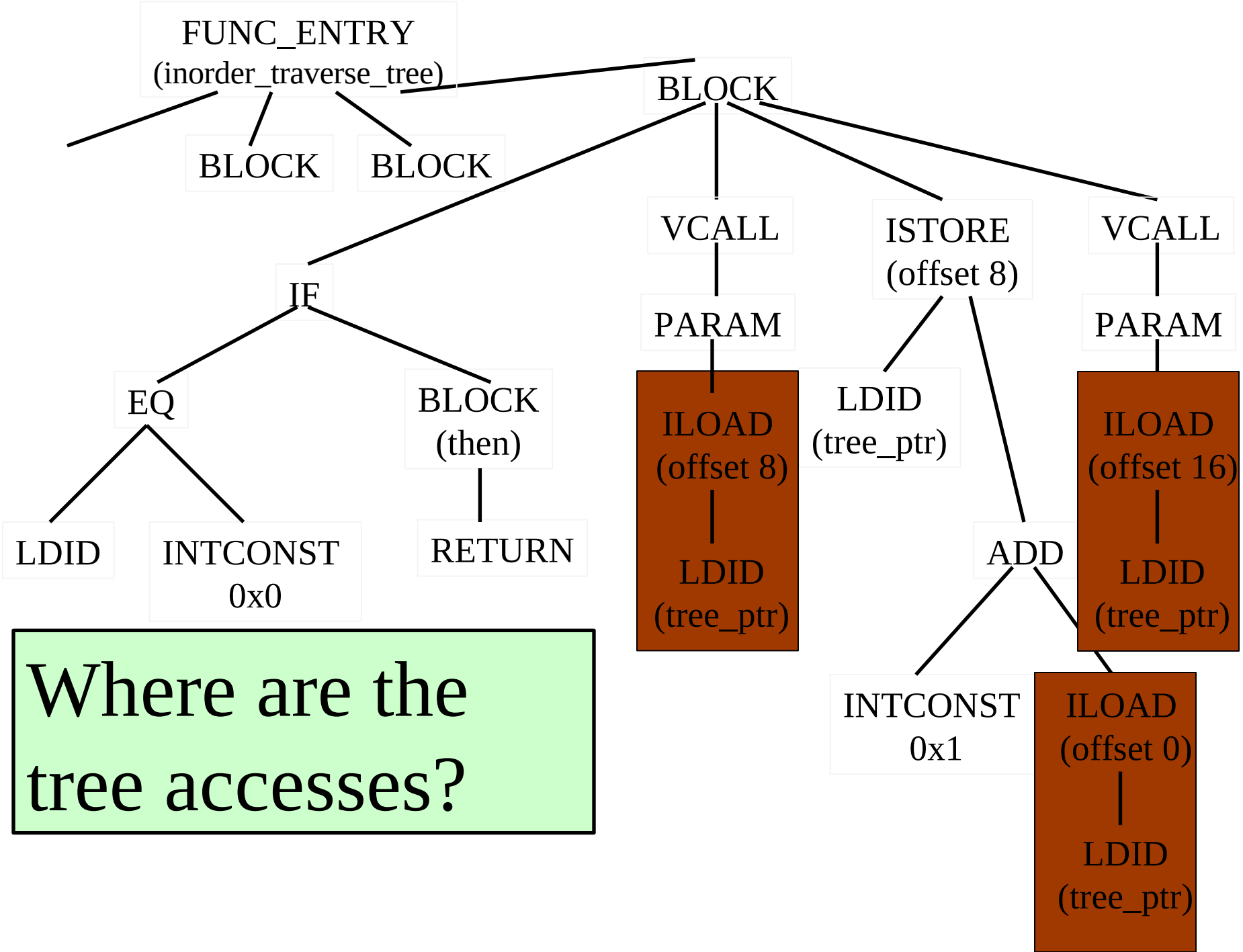
VCALL 126 <1,25,inorder_traverse_tree> # flags 0x7e

RETURN

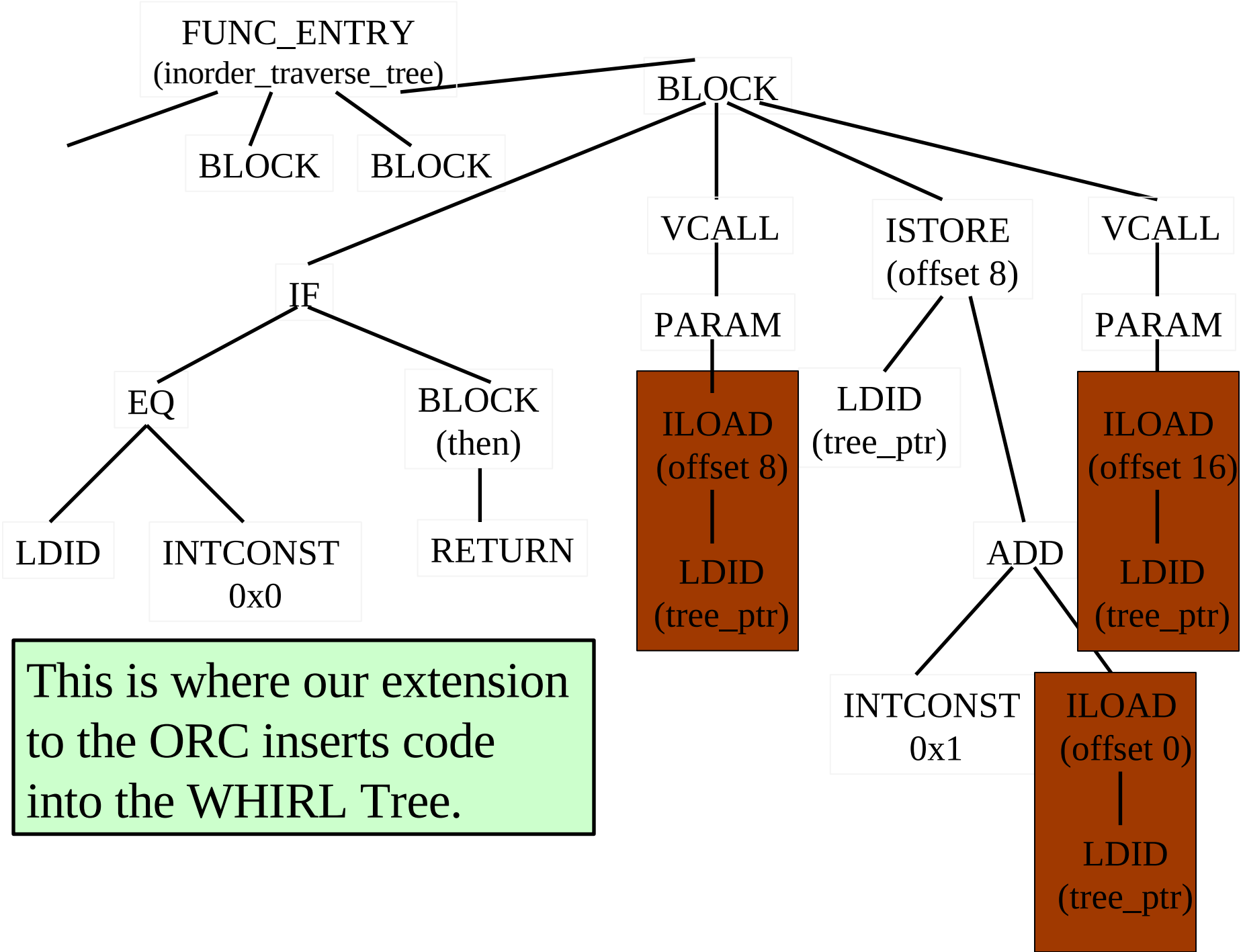
END_BLOCK




```
void inorder_traverse_tree(struct tn
   *tree_ptr)
{ if(tree_ptr == NULL)
    return;
  inorder_traverse_tree(tree_ptr-
   >left);
  tree_ptr->data++;
  inorder_traverse_tree(tree_ptr-
   >right);
}
```



Where are the tree accesses?

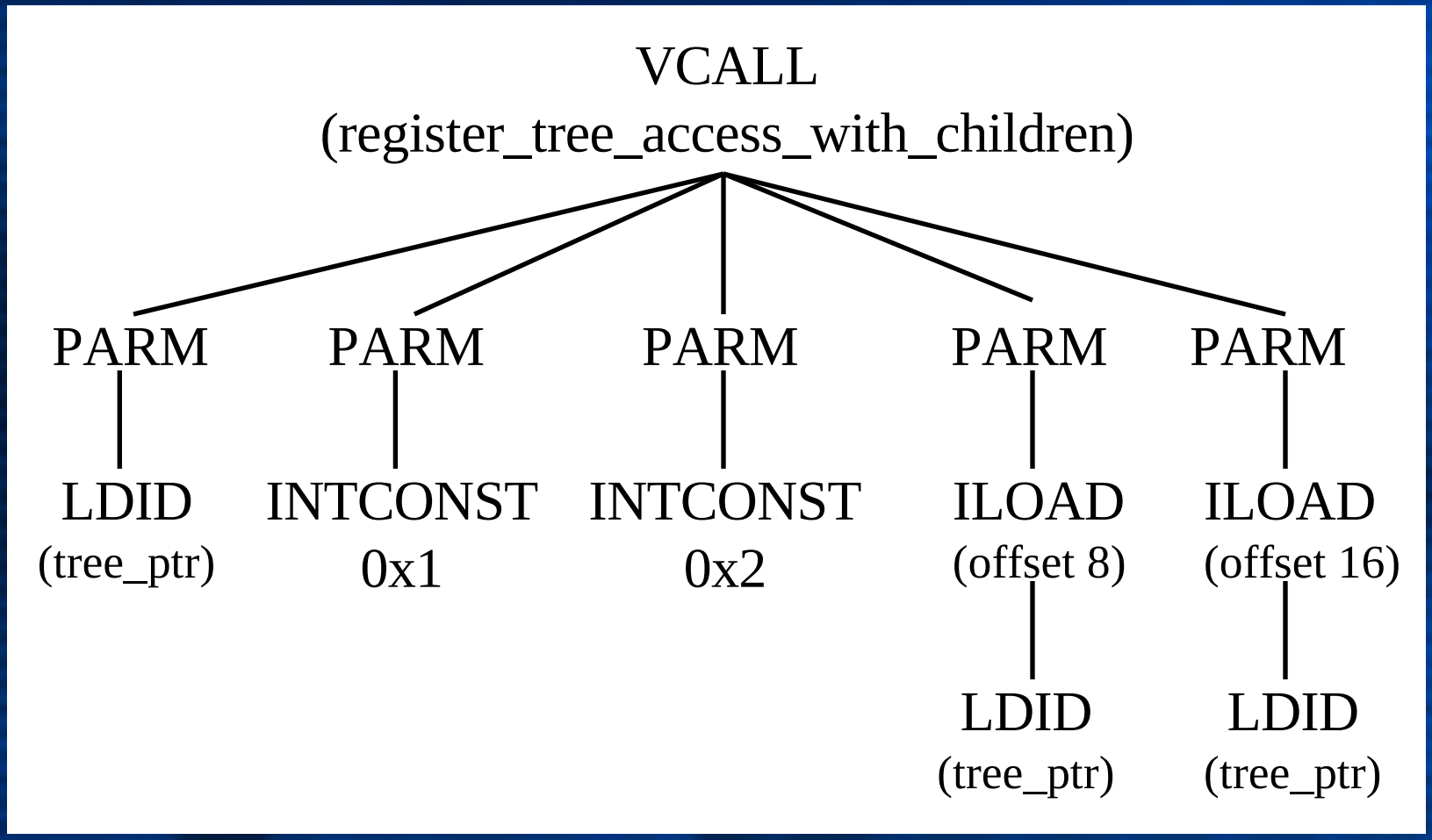


This is where our extension to the ORC inserts code into the WHIRL Tree.

The Nodes we Insert Into WHIRL

```
U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>
 U8PARM 2 T<31,anon_ptr.,8> #  by_value
U4INTCONST 1 (0x1)
 U4PARM 2 T<8,.predef_U4,4> #  by_value
  U4INTCONST 2 (0x2)
 U4PARM 2 T<8,.predef_U4,4> #  by_value
   U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>
  U8U8ILOAD 8 T<31,anon_ptr.,8> T<34,anon_ptr.,8>
 U8PARM 2 T<31,anon_ptr.,8> #  by_value
   U8U8LDID 0 <2,1,tree_ptr> T<31,anon_ptr.,8>
  U8U8ILOAD 16 T<31,anon_ptr.,8> T<34,anon_ptr.,8>
 U8PARM 2 T<31,anon_ptr.,8> #  by_value
VCALL 126 <1,21,register_access_with_children> # flags 0x7e
```

The Nodes we Insert Into WHIRL



Experimental Setup

Open Research Compiler v2.1

- Optimization level -O3

1.3 GHz Itanium2

1 GB of RAM

Synthetic Benchmarks

RandomDepth: DFS traversal of binary tree, children visited in random order;

BreadthFirst: BFS on binary tree;

DepthBreadth: DFS followed by BFS on binary tree;

NonStandard: A traversal that is neither BFS nor DFS;

MultiDepth: Several DFSs: in-order, pre-order, and post-order;

BreadthSearch: Several BFS searchers on a tree with random branching factor (2 to 6);

BinarySearch: Several searches in a binary tree.

Analysis Results
(Synthetic Benchmark)

Benchmark	Orientation Score	
	Expected	Experimental
Random Depth	1.0	1.000000
Breadth-First	-1.0	-0.999992
Depth-Breadth	0.0	0.000000
Non-Standard	0.0	0.136381
Multi Depth	1.0	0.999974
Breadth Search	-1.0	-0.995669
Binary Search	1.0	0.941225

Analysis Results
(Olden Benchmark)

Benchmark	Orientation Score	
	Expected	Experimental
BH	0.0	0.010266
Bisort	0.0	-0.001014
Health	1.0	0.807330
MST	Low positive	0.335852
Perimeter	Low positive	0.195177
Power	1.0	0.991617
Treeadd	1.0	1.000000
TSP	Low positive	0.173267

Runtime Overhead

(Synthetic Benchmark)

Benchmark	**Runtime (s)**	
	Original	**Instrumented**
Random Depth	0.90	10.72
Breadth-First	1.09	1.81
Depth-Breadth	1.33	7.84
Non-Standard	0.36	2.68
Multi Depth	0.47	3.94
Breadth Search	0.36	1.83
Binary Search	0.20	2.75

Runtime Overhead
(Olden Benchmark)

Benchmark	Runtime (s)	
	Original	Instrumented
BH	0.45	6.88
Bisort	0.03	1.01
Health	0.05	12.06
MST	5.71	11.42
Perimeter	0.02	1.55
Power	1.35	1.60
Treeadd	0.13	6.35
TSP	0.01	1.40

Memory Overhead
(Synthetic Benchmark)

Benchmark	Memory Usage (kbytes)	
	Original	Instrumented
Random Depth	854 976	855 040
Breadth-First	424 928	441 328
Depth-Breadth	220 112	252 896
Non-Standard	420 800	420 912
Multi Depth	529 344	652 288
Breadth Search	30 192	47 408
Binary Search	224 192	224 320

Memory Overhead
(Olden Benchmark)

Benchmark	Memory Usage (kbytes)	
	Original	Instrumented
BH	3 520	3 520
Bisort	3 008	3 040
Health	8 448	8 624
MST	4 752	6 272
Perimeter	6 064	6 528
Power	3 648	3 712
Treeadd	3 008	3 008
TSP	3 024	3 040

Possible Clients of the Analysis

Prefetching

- Luk and Mowry - History-Pointer Prefetching [7]
- Cahoon and McKinley - Jump-Pointer Prefetching [3]

```
struct myStruct{
    int data;
    struct myStruct *next;
};
```

```
struct myStruct{
    int data;
    struct myStruct *next;
    struct myStruct *prefetch;
};
```

Possible Clients of the Analysis

- Prefetching
  - In the first iteration through the pointer chasing loop there is no prefetching (initializing prefetch pointers).
  - If the traversal is known, the history pointers can be initialized as the data structure is allocated.
    - Eliminate compulsory cache misses

Possible Clients of the Analysis

- Modify Data Layout
  - Memory Allocation
    - Calder *et al.* [4] and Chilimbi, Hill and Larus [6]
    - Create a custom allocator or a reallocation method that (re)organizes the data structure.
    - Manual & semi-automatic techniques.
  - Automatically insert the allocation calls with a *hint* from our analysis.

Possible Clients of the Analysis

- Modify Data Layout
  - Structure Reorganization
    - Truong, Bodin and Seznec [8]
    - Chilimbi, Davidson and Larus [5]

Use analysis information with Chilimbi's data outlining or Truong's *ialloc* memory allocator to improve spatial locality.

Conclusion

- Developed an efficient analysis to *automatically* determine the orientation of tree traversals.
  - Instrumented applications only require 7% more memory.
  - There are many clients for this analysis that can potentially improve program performance.

Questions?

References

[1] J. Patterson. Modern Microprocessors A 90 Minute Guide! *http://www.pattosoft.com.au/Articles/ModernMicroprocessors/*

[2] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap directed pointers in C. POPL 1996

[3] B.Cahoon and K.S. McKinley. Data flow analysis for software prefetching linked data structures in Java. PACT '01

[4] B.Calder, C. Krintz, S. John, T. Austin. Cache-conscious data placement. ASPLOS-VIII, 1998

[5] T.M. Chilimbi, B. Davidson and J.R. Larus. Cache-conscious structure definition. PLDI '99

[6] T.M. Chilimbi, M.D. Hill and J.R. Larus. Cache-conscious structure layout. PLDI '99

[7] C.K. Luk and T.C. Mowry. Compiler-based prefetching for recursive data structures. ACM SIGOPS Operating Systems Review, 1996.

[8] D.N. Truong, F. Bodin and A. Seznec. Improving cache behavior of dynamically allocated data structures. PACT '98