LORAX

The Language that Speaks for the Trees

Lorax is an original language and compiler designed in a team as a final project for Columbia University’s compiler course Fall 2013.

The language is designed to make tree ADT creation and manipulation simple.

What will we look at?

- A brief introduction to the language
- A brief overview of the compiler
- A detailed look at memory management

Lorax in One Slide*

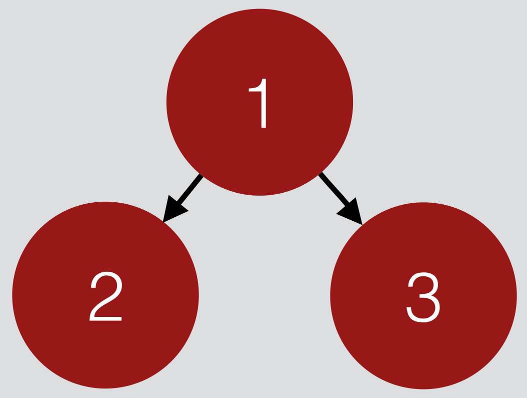
*not even close

Access the tree’s first child then dereference data member.

Trees are passed by reference.

```
int change_child(tree<int>t(2)) {
    t%0@ = 102;
}

int main() {
    tree <int>t(2);
    t = 1[2, 3];
    change_child(t);
    print("tree t = ", t, "\n");
}
```

Tree declaration and Definition. Tree of type integer with degree 2.



Tree literal degree and type checking

print(): variable argument accepting omni-types

String literal is syntactic sugar for a 1-degree character tree

Terminal Output

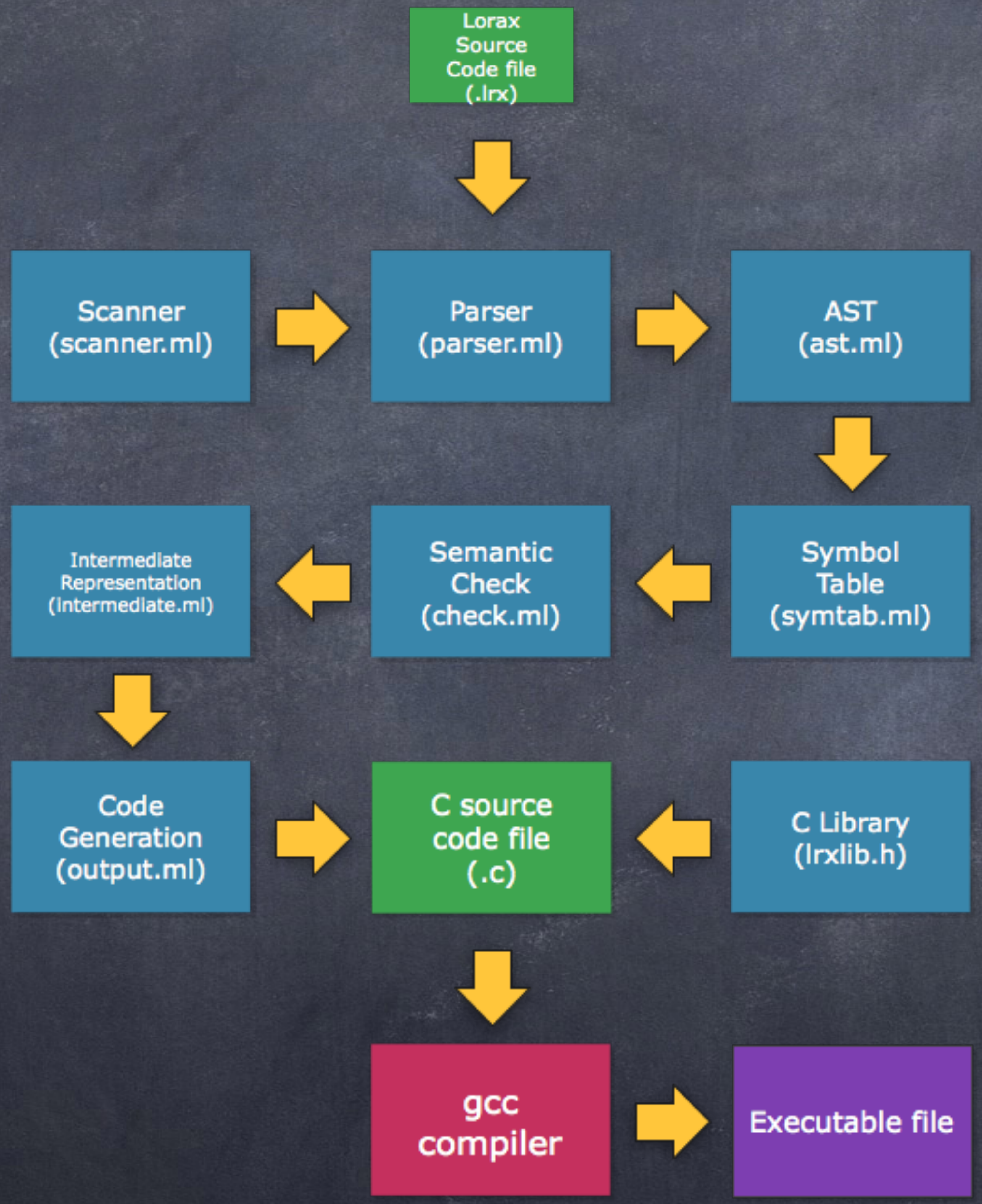
```
tree t =
1[102[null,null],
3[null,null]]
```

Full Example: Depth First Search

```
bool dfs(tree <int>t(2), int val) {
  int child;
  bool match;
  match = false;
  if (t == null) { return false; }
  if (t@ == val) { return true; }
  for (child = 0; child < degree(t); child = child + 1) {
    if (t%child != null) {
      if(t%child@ == val) { return true; }
      else { match = dfs(t%child, val); }
    }
  }
  return match;
}
int main() {
  tree <int>t(2);
  t = 1[2, 3[4, 5]];
  if (dfs(t, 3)) { print("found it\n"); }
  else { print("its not there\n"); }
}
```

How it Works

How it Works: Overview



- **scanner.ml**: separates source code into tokens
- **parser.ml**: parses tokens into AST
- **symtab.ml**: builds symbol table for all identifiers
- **check.ml**: validates AST
- **intermediate.ml**: flattens all but function def/calls into a list of three address like code
- **output.ml**: converts intermediate types to c compatible syntax

How it Works: Compiler Output

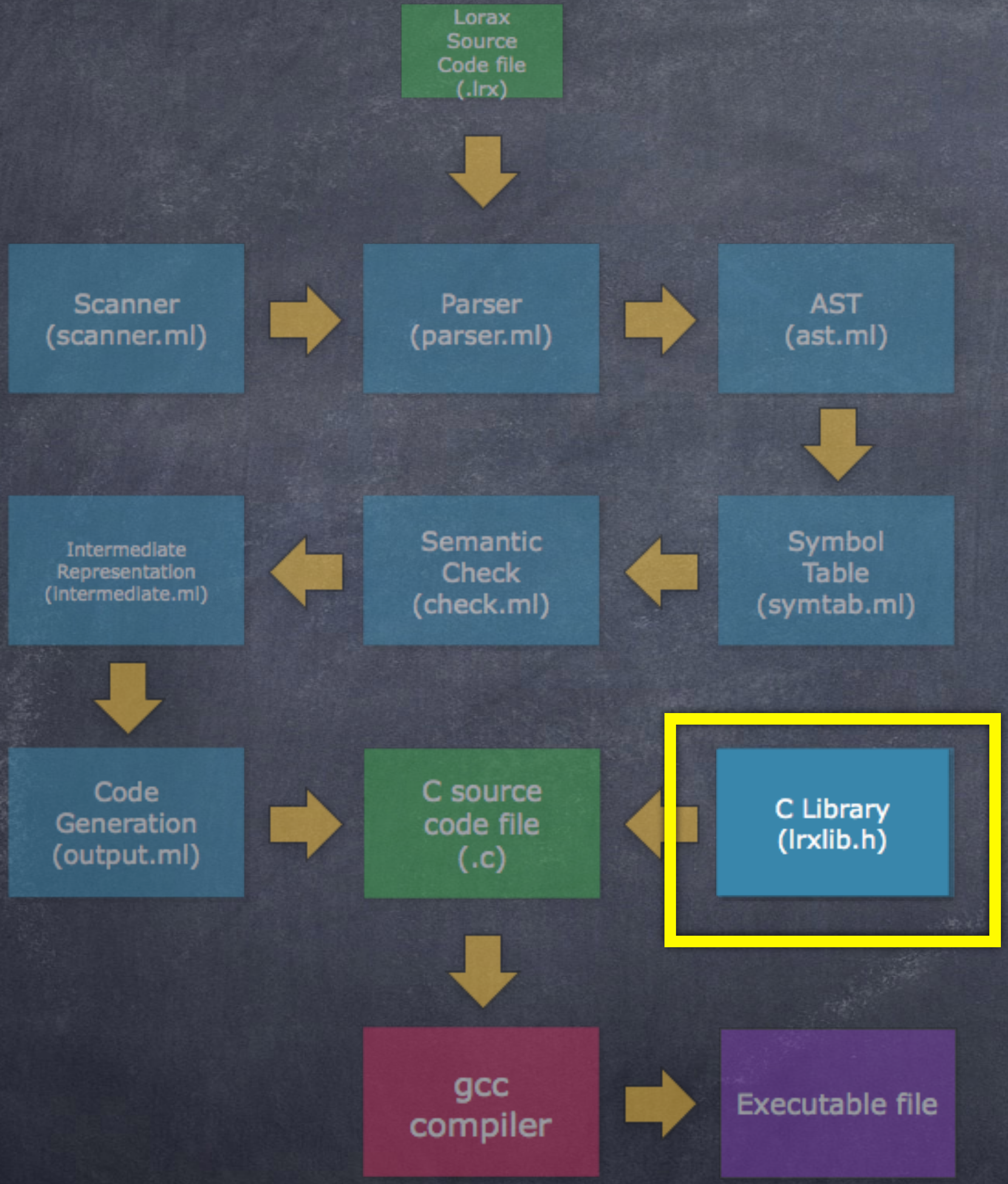
simple.lrx

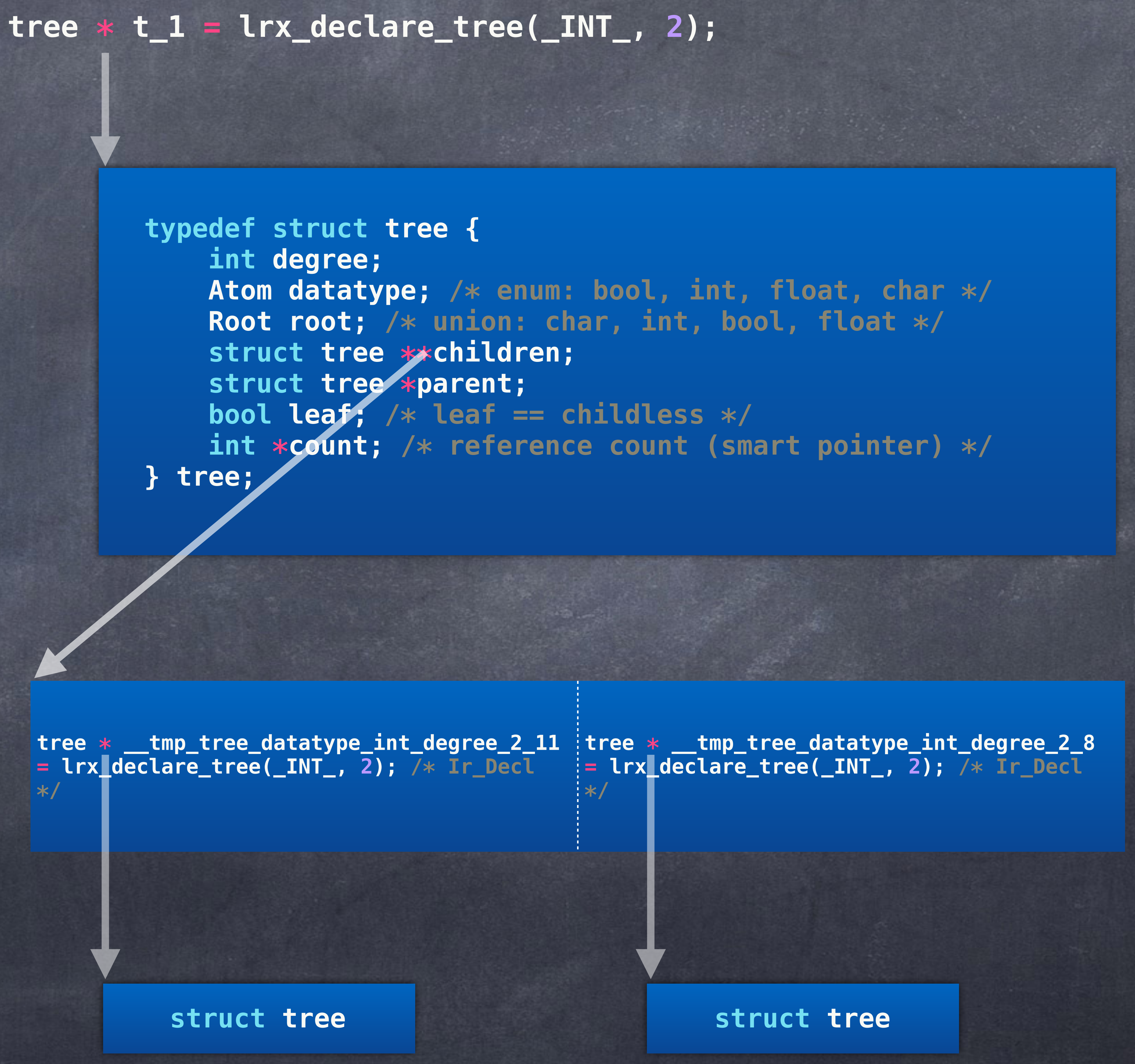
```
int main() {
    int x;
    x = 3 + 4;
}
```

simple.lrx_lrxtmp.c

```
#include "lrxlib.h"
int main();

int main()
{
int x_1 = 0; /* Ir_Decl */
int __tmp_int_3 = 0; /* Ir_Decl */
int __tmp_int_1 = 0; /* Ir_Decl */
int __tmp_int_2 = 0; /* Ir_Decl */
int __tmp_int_0 = 0; /* Ir_Decl */

__tmp_int_1 = 3;

__tmp_int_2 = 4;

__tmp_int_3 = __tmp_int_1 + __tmp_int_2;

x_1 = __tmp_int_3;

goto __LABEL_0;
__LABEL_1:
return __tmp_int_0;
__LABEL_0:


goto __LABEL_1;
}
```

How it Works: #include “lrxlib.h”





```
tree * t_1 = lrx_declare_tree(_INT_, 2);
```

```
typedef struct tree {
    int degree;
    Atom datatype; /* enum: bool, int, float, char */
    Root root; /* union: char, int, bool, float */
    struct tree **children;
    struct tree *parent;
    bool leaf; /* leaf == childless */
    int *count; /* reference count (smart pointer) */
} tree;
```

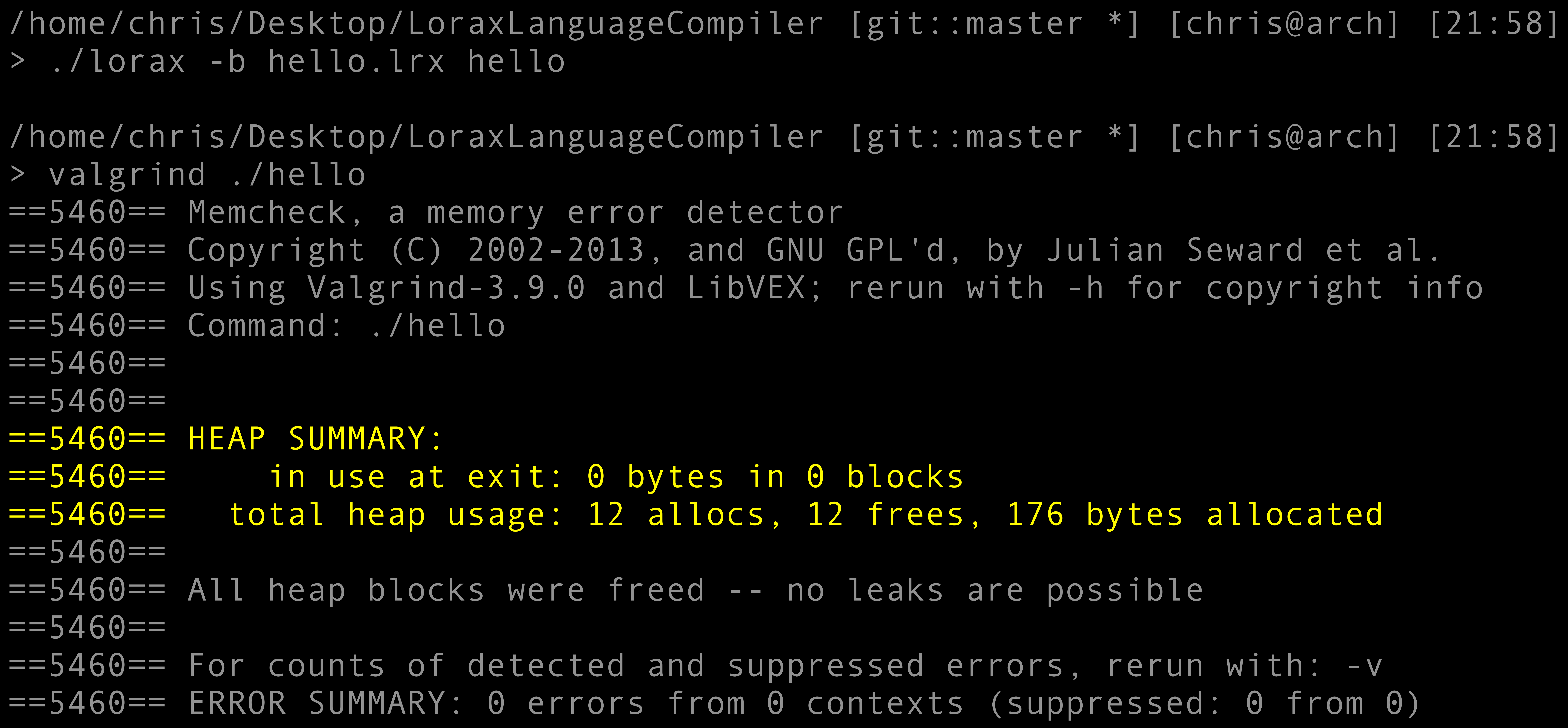
```
tree * __tmp_tree_datatype_int_degree_2_11
= lrx_declare_tree(_INT_, 2); /* Ir_Decl
*/
```

```
tree * __tmp_tree_datatype_int_degree_2_8
= lrx_declare_tree(_INT_, 2); /* Ir_Decl
*/
```

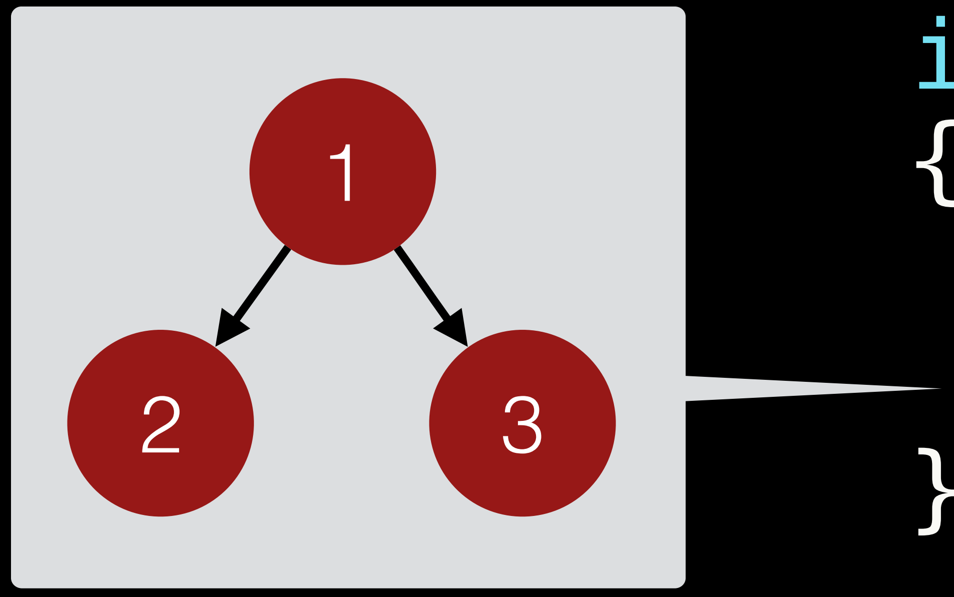
```
struct tree
```

```
struct tree
```

ARC in Lorax



```
/home/chris/Desktop/LoraxLanguageCompiler [git::master *] [chris@arch] [21:58]
> ./lorax -b hello.lrx hello

/home/chris/Desktop/LoraxLanguageCompiler [git::master *] [chris@arch] [21:58]
> valgrind ./hello
==5460== Memcheck, a memory error detector
==5460== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==5460== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==5460== Command: ./hello
==5460==
==5460==
==5460== HEAP SUMMARY:
==5460==     in use at exit: 0 bytes in 0 blocks
==5460==   total heap usage: 12 allocs, 12 frees, 176 bytes allocated
==5460==
==5460== All heap blocks were freed -- no leaks are possible
==5460==
==5460== For counts of detected and suppressed errors, rerun with: -v
==5460== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```



hello.lrx

```
int main()
{
  tree <int>a(2);
  a = 1[2, 3];
}
```

hello.lrx_lrxtmp.c

```
#include "lrxlib.h"
int main();

int main()
{
tree * a_1 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
tree * __tmp_tree_datatype_int_degree_2_5 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
int __tmp_int_4 = 0; /* Ir_Decl */
int __tmp_int_3 = 0; /* Ir_Decl */
int __tmp_int_2 = 0; /* Ir_Decl */
int *__tmp_int_9 = NULL; /* Ir_At_Ptr */
tree * __tmp_tree_datatype_int_degree_2_11 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
int *__tmp_int_6 = NULL; /* Ir_At_Ptr */
tree * __tmp_tree_datatype_int_degree_2_8 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
int *__tmp_int_13 = NULL; /* Ir_At_Ptr */
int __tmp_int_1 = 0; /* Ir_Decl */
int __tmp_int_0 = 0; /* Ir_Decl */

__tmp_int_4 = 2;

__tmp_int_3 = 3;

__tmp_int_2 = 1;

__tmp_int_9 = &__tmp_int_4; /* Ir_Ptr */
tree * __tmp_tree_datatype_int_degree_2_10[2]; /* Ir_Leaf */
__tmp_tree_datatype_int_degree_2_10[1] = NULL; /* c_of_leaf */
__tmp_tree_datatype_int_degree_2_10[0] = NULL; /* c_of_leaf */

lrx_define_tree(__tmp_tree_datatype_int_degree_2_11, __tmp_int_9, __tmp_tree_datatype_int_degree_2_10);

__tmp_int_6 = &__tmp_int_3; /* Ir_Ptr */
tree * __tmp_tree_datatype_int_degree_2_7[2]; /* Ir_Leaf */
__tmp_tree_datatype_int_degree_2_7[1] = NULL; /* c_of_leaf */
__tmp_tree_datatype_int_degree_2_7[0] = NULL; /* c_of_leaf */

lrx_define_tree(__tmp_tree_datatype_int_degree_2_8, __tmp_int_6, __tmp_tree_datatype_int_degree_2_7);

tree * __tmp_tree_datatype_int_degree_2_12[2]; /* Ir_Child_Array */
/* Filling with NULL preemptively */
__tmp_tree_datatype_int_degree_2_12[1] = NULL; /* c_of_leaf */
__tmp_tree_datatype_int_degree_2_12[0] = NULL; /* c_of_leaf */

__tmp_tree_datatype_int_degree_2_12[0] = __tmp_tree_datatype_int_degree_2_11; /* Ir_Internal */
__tmp_tree_datatype_int_degree_2_12[1] = __tmp_tree_datatype_int_degree_2_8; /* Ir_Internal */
__tmp_int_13 = &__tmp_int_2; /* Ir_Ptr */
lrx_define_tree(__tmp_tree_datatype_int_degree_2_5, __tmp_int_13, __tmp_tree_datatype_int_degree_2_12);

lrx_assign_tree_direct(&a_1, &__tmp_tree_datatype_int_degree_2_5);

__tmp_int_1 = 0;

goto __LABEL_2;
__LABEL_3:
return __tmp_int_1;
__LABEL_2:
lrx_destroy_tree(a_1);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_5);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_11);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_8);

goto __LABEL_3;

goto __LABEL_0;
__LABEL_1:
return __tmp_int_0;
__LABEL_0:
lrx_destroy_tree(a_1);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_5);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_11);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_8);

goto __LABEL_1;
}
```

Declaring Temporary Trees

hello.lrx_lrxtmp.c

```
tree * a_1 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
tree * __tmp_tree_datatype_int_degree_2_5 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
tree * __tmp_tree_datatype_int_degree_2_11 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
tree * __tmp_tree_datatype_int_degree_2_8 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
```

lrxlib.h

```
struct tree *lrx_declare_tree(Atom type, int deg) {
    assert(deg >= 0);
    struct tree *t = (struct tree *)malloc(sizeof(struct tree));
    assert(t);

    t->degree = deg;
    t->datatype = type;
    t->count = (int *)malloc(sizeof(int));
    assert(t->count);
    *(t->count) = 1;

    switch (type) {
        case _BOOL_: t->root.bool_root = false; break;
        case _INT_: t->root.int_root = 0; break;
        case _FLOAT_: t->root.float_root = 0.0; break;
        case _CHAR_: case _STRING_:
         if (t->degree == 1) {
             LrxLog("Declare string\n");
            t->datatype = _STRING_;
         }

       t->root.char_root = '\0';
         break;
    }

    t->is_null = true;
    t->leaf = true;
    if (t->degree > 0) {
        t->children = (struct tree **)malloc(sizeof(struct tree *) * t->degree);
        assert(t->children);
        memset((t->children), 0, sizeof(struct tree*) * t->degree);
    }

    t->parent = NULL;
    return t;
}
```

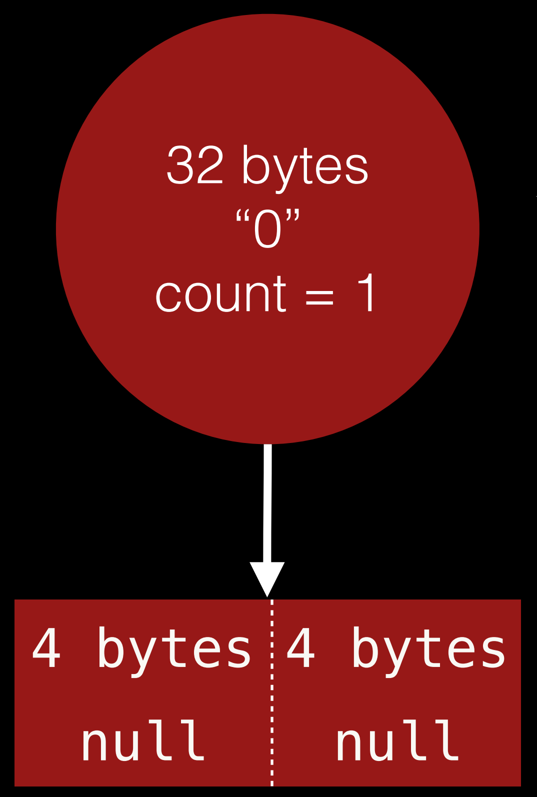
Initialize Reference Count = 1

Declaring Temporary Trees

hello.lrx_lrxtmp.c

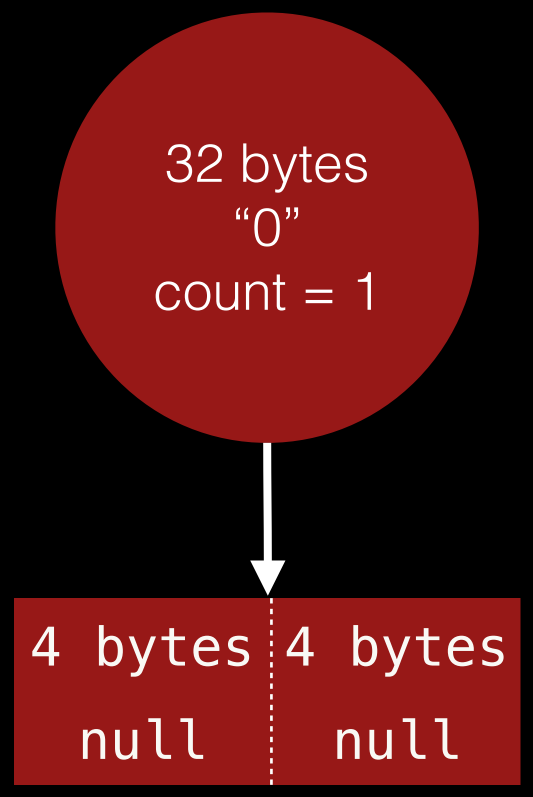
```
tree * a_1 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
tree * __tmp_tree_datatype_int_degree_2_5 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
tree * __tmp_tree_datatype_int_degree_2_11 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
tree * __tmp_tree_datatype_int_degree_2_11 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
```

a_1 = 0x89e7008

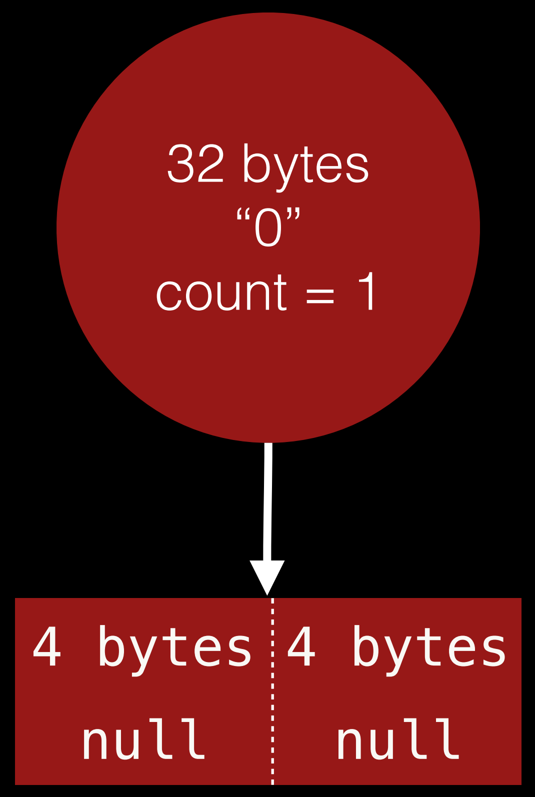


Convenient container to match type/ degree during assignment (used later)

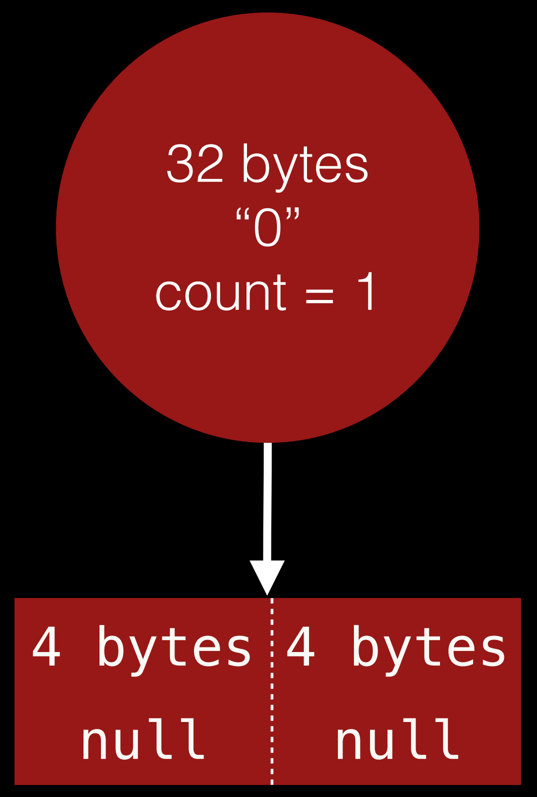
__tmp_tree_datatype_int_degree_2_5 = 0x89e7050



__tmp_tree_datatype_int_degree_2_11 = 0x89e7098



__tmp_tree_datatype_int_degree_2_8 = 0x89e70e0



```
176 Bytes Allocated = [malloc(32 bytes struct tree) + malloc(4 bytes for int count) + malloc(8 bytes for tree’s children pointers)] * 4 trees
176 Bytes in use.
```

Defining Tree Values and Children

hello.lrx_lrxtmp.c

```
lrx_define_tree(__tmp_tree_datatype_int_degree_2_11, __tmp_int_9, __tmp_tree_datatype_int_degree_2_10);
lrx_define_tree(__tmp_tree_datatype_int_degree_2_8, __tmp_int_6, __tmp_tree_datatype_int_degree_2_7);

tree * __tmp_tree_datatype_int_degree_2_12[2]; /* Ir_Child_Array */
__tmp_tree_datatype_int_degree_2_12[0] = __tmp_tree_datatype_int_degree_2_11;
__tmp_tree_datatype_int_degree_2_12[1] = __tmp_tree_datatype_int_degree_2_8;
lrx_define_tree(__tmp_tree_datatype_int_degree_2_5, __tmp_int_13, __tmp_tree_datatype_int_degree_2_12);
```

lrxlib.h

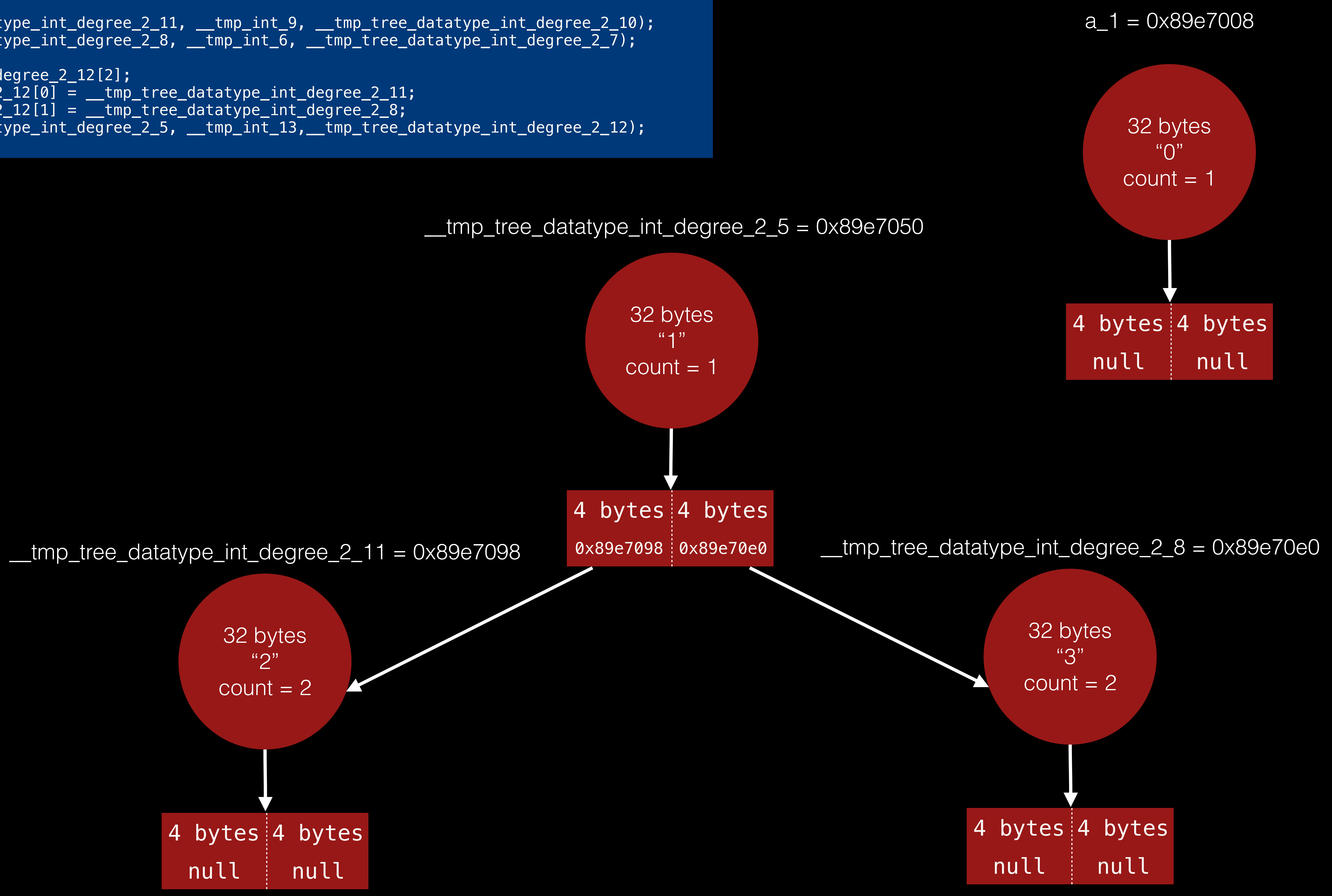
```
struct tree *lrx_define_tree(struct tree *t, void *root_data, struct tree **children){
    /* set root data */
    switch (t->datatype) {
        case _BOOL_: t->root.bool_root = *((bool *)root_data); break;
        case _INT_: t->root.int_root = *((int *)root_data); break;
        case _FLOAT_: t->root.float_root = *((float *)root_data); break;
        case _CHAR_: case _STRING_: t->root.char_root = *((char *)root_data); break;
    }

    t->is_null = false;

    if (children == NULL){
        return t;
    }

    /* set pointers to children */
    int num_children = t->degree;
    int i;
    int null = 0;
    for (i = 0; i < num_children; ++i) {
        if (children[i] != NULL){
            children[i]->parent = t;
            *(children[i]->count) += 1;
            t->children[i] = children[i];
        } else {
            null +=1;
        }
    }

    if(null != num_children) {
        t->leaf = false;
    }

    return t;
}
```

Increment Child Reference Count

Defining Tree Values and Children

hello.lrx_lrxtmp.c

```
lrx_define_tree(__tmp_tree_datatype_int_degree_2_11, __tmp_int_9, __tmp_tree_datatype_int_degree_2_10);
lrx_define_tree(__tmp_tree_datatype_int_degree_2_8, __tmp_int_6, __tmp_tree_datatype_int_degree_2_7);

tree * __tmp_tree_datatype_int_degree_2_12[2];
__tmp_tree_datatype_int_degree_2_12[0] = __tmp_tree_datatype_int_degree_2_11;
__tmp_tree_datatype_int_degree_2_12[1] = __tmp_tree_datatype_int_degree_2_8;
lrx_define_tree(__tmp_tree_datatype_int_degree_2_5, __tmp_int_13,__tmp_tree_datatype_int_degree_2_12);
```



Assigning Temporary Tree to Symbol

hello.lrx_lrxtmp.c

```
lrx_assign_tree_direct(&a_1, &__tmp_tree_datatype_int_degree_2_5);
```

lrxlib.h

```
/* t1 = t2. Lhs is the tree pointer we need without dereference */
struct tree **lrx_assign_tree_direct(struct tree **lhs, struct tree **rhs) {
    if (lhs == rhs) {
        return lhs;
   }

   if (lhs && rhs && *rhs && *lhs) {
        if ((*rhs)->degree == 0) {
            int lhs_degree = (*lhs)->degree;
            (*rhs)->degree = lhs_degree;
            (*rhs)->children =
           (struct tree **)malloc(sizeof(struct tree *) * lhs_degree);
            assert((*rhs)->children);
            memset(((*rhs)->children), 0, sizeof(struct tree*) * lhs_degree);
        }
        assert((*lhs)->degree == (*rhs)->degree);
    }

    if (*lhs) {
        if ((*lhs)->parent) {
            ((*lhs)->parent)->leaf = false;
        }
    }

    lrx_destroy_tree(*lhs);
    *lhs = *rhs;
    if (*rhs) {
        if ((*rhs)->count) {
            *((*rhs)->count) += 1;
        }
    }

    return lhs;
}
```

Increment Right-Hand Side

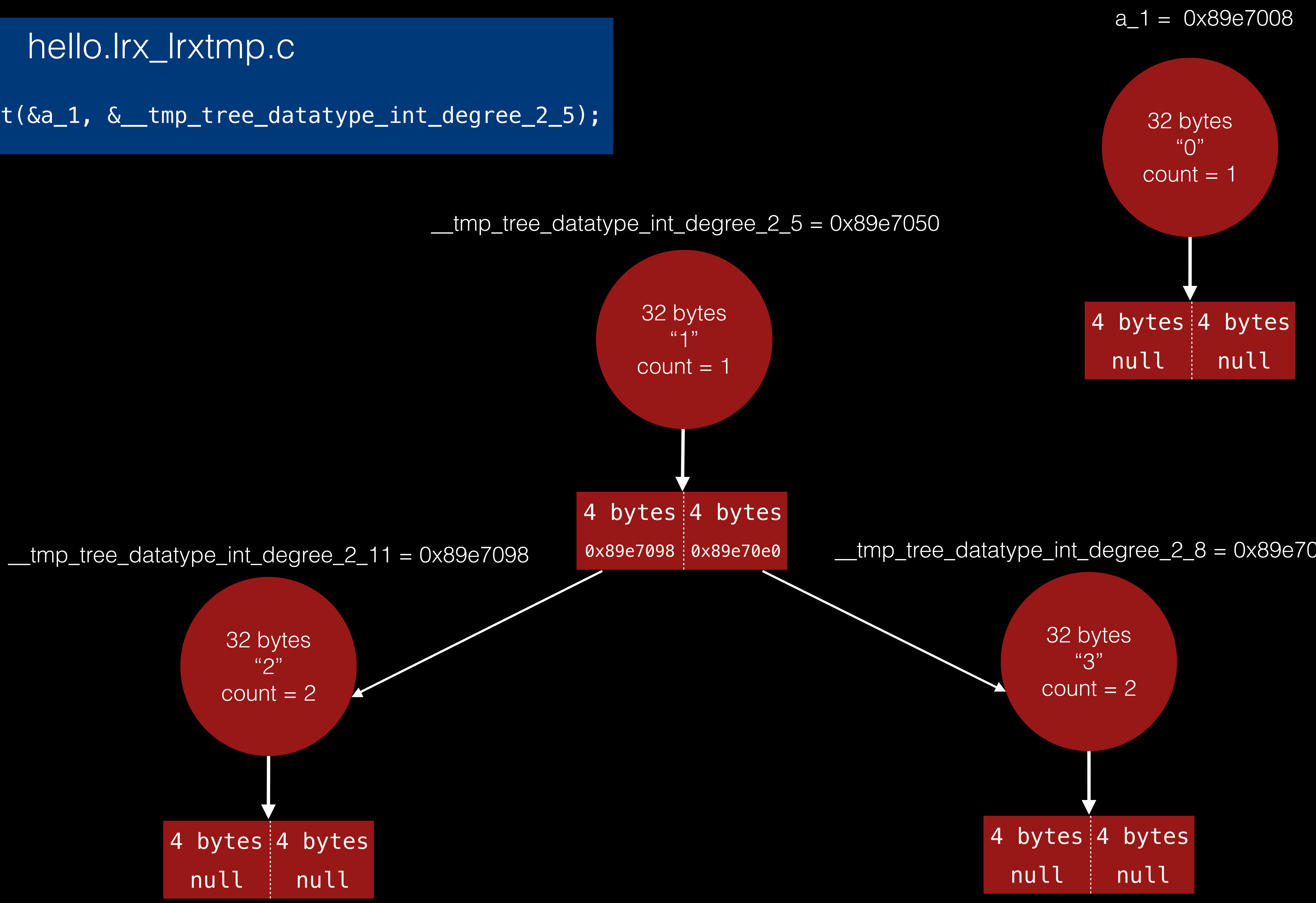
Decrement and Destroy Left-Hand Side

```
void lrx_destroy_tree(struct tree *t) {

    if (t == NULL) {
        return;
    }

    *(t->count) -= 1;
    if (*(t->count) == 0) {

        if (t->children) {
            int i;
            for (i = 0; i < t->degree; ++i){
                lrx_destroy_tree(t->children[i]);
            }
            free(t->children);
        }

        free(t->count);
        free(t);
    }
}
```

Assigning Temporary Tree to Symbol

hello.lrx_lrxtmp.c

```
lrx_assign_tree_direct(&a_1, &__tmp_tree_datatype_int_degree_2_5);
```



First, destroy left-hand side of assignment.

```
44 Bytes Freed = [free(32 bytes for a_1 struct tree) + free(4 bytes for a_1’s counter) + free(8 bytes for a_1’s children pointers)]
132 Bytes in use
```

Leaving Scope: Destroy Trees

hello.lrx_lrxtmp.c

```
lrx_destroy_tree(a_1);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_5);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_11);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_8);
```

lrxlib.h

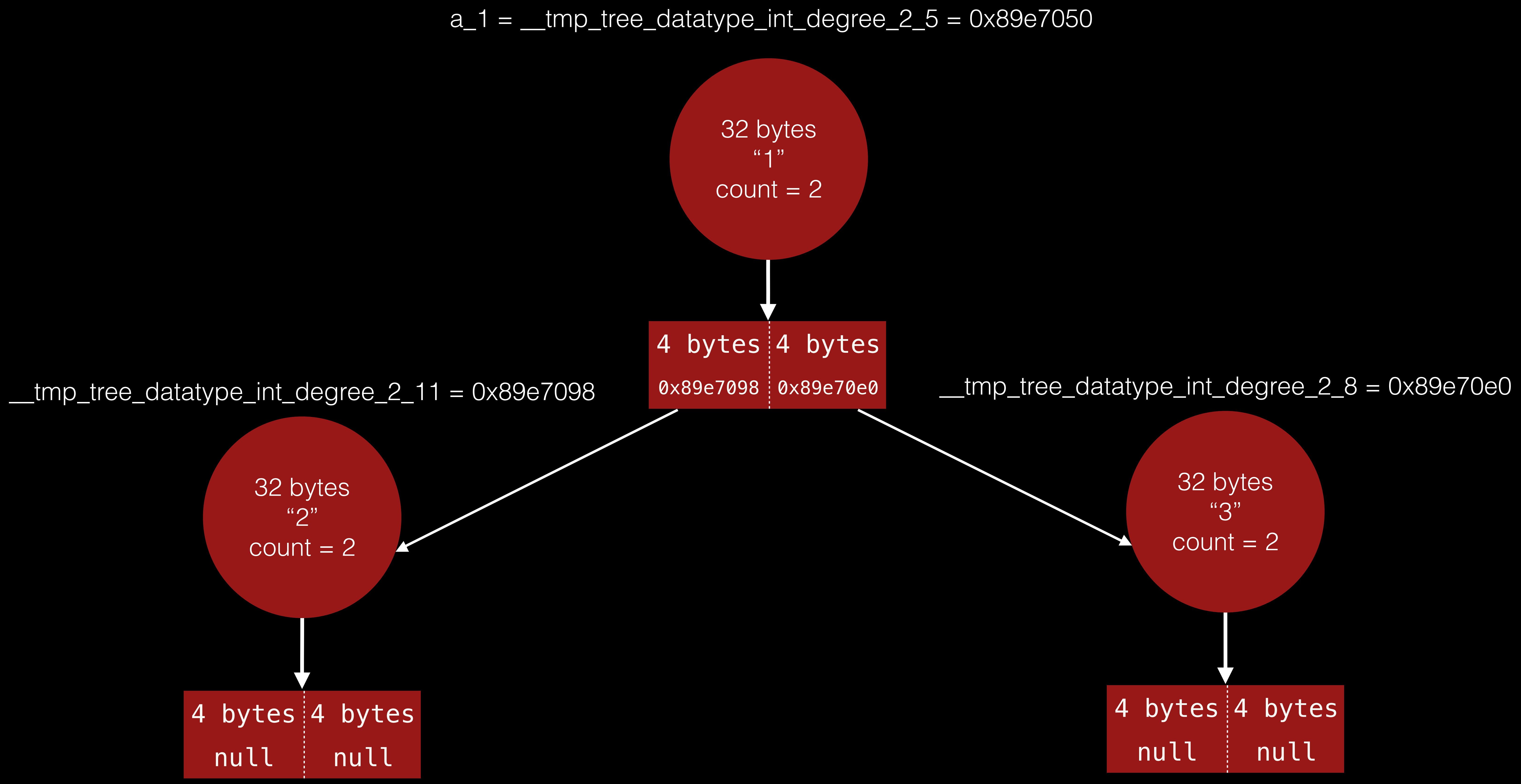
```
void lrx_destroy_tree(struct tree *t) {

    if (t == NULL) {
        return;
    }

    *(t->count) -= 1;
    if (*(t->count) == 0) {

        if (t->children) {
            int i;
            for (i = 0; i < t->degree; ++i){
                lrx_destroy_tree(t->children[i]);
            }
            free(t->children);
        }

        free(t->count);
        free(t);
    }
}
```

Decrement and Destroy When Count == 0

Leaving Scope: Destroy Trees

hello.lrx_lrxtmp.c

```
lrx_destroy_tree(a_1);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_5);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_11);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_8);
```



a_1 = __tmp_tree_datatype_int_degree_2_5 = 0x89e7050

32 bytes
“1”
count = 2

4 bytes | 4 bytes
0x89e7098 | 0x89e70e0

__tmp_tree_datatype_int_degree_2_11 = 0x89e7098

32 bytes
“2”
count = 2

4 bytes | 4 bytes
null | null

__tmp_tree_datatype_int_degree_2_8 = 0x89e70e0

32 bytes
“3”
count = 2

4 bytes | 4 bytes
null | null

```
132 Bytes Freed = [free(32 bytes struct tree) + free(4 bytes for int count) + free(8 bytes for tree’s children pointers)] * 3 trees
0 Bytes in use.
```

Leaving Scope: Destroy Trees

hello.lrx_lrxtmp.c

```
lrx_destroy_tree(a_1);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_5);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_11);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_8);
```

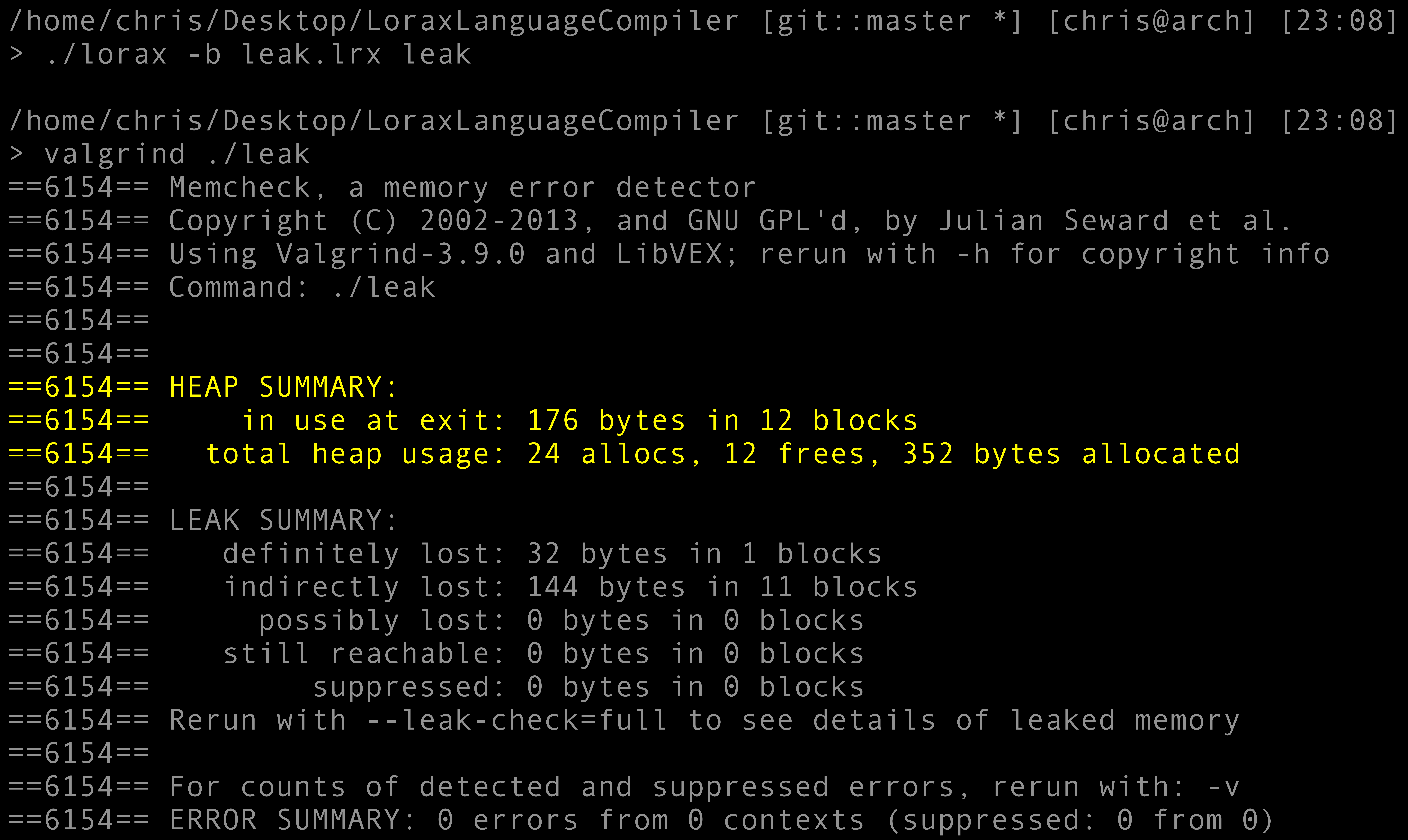
```
0 Bytes in use.
```

```
132 Bytes Freed = [free(32 bytes struct tree) + free(4 bytes for int count) + free(8 bytes for tree’s children pointers)] * 3 trees
0 Bytes in use.
```

Amazing Right?

Except for reference cycles.*

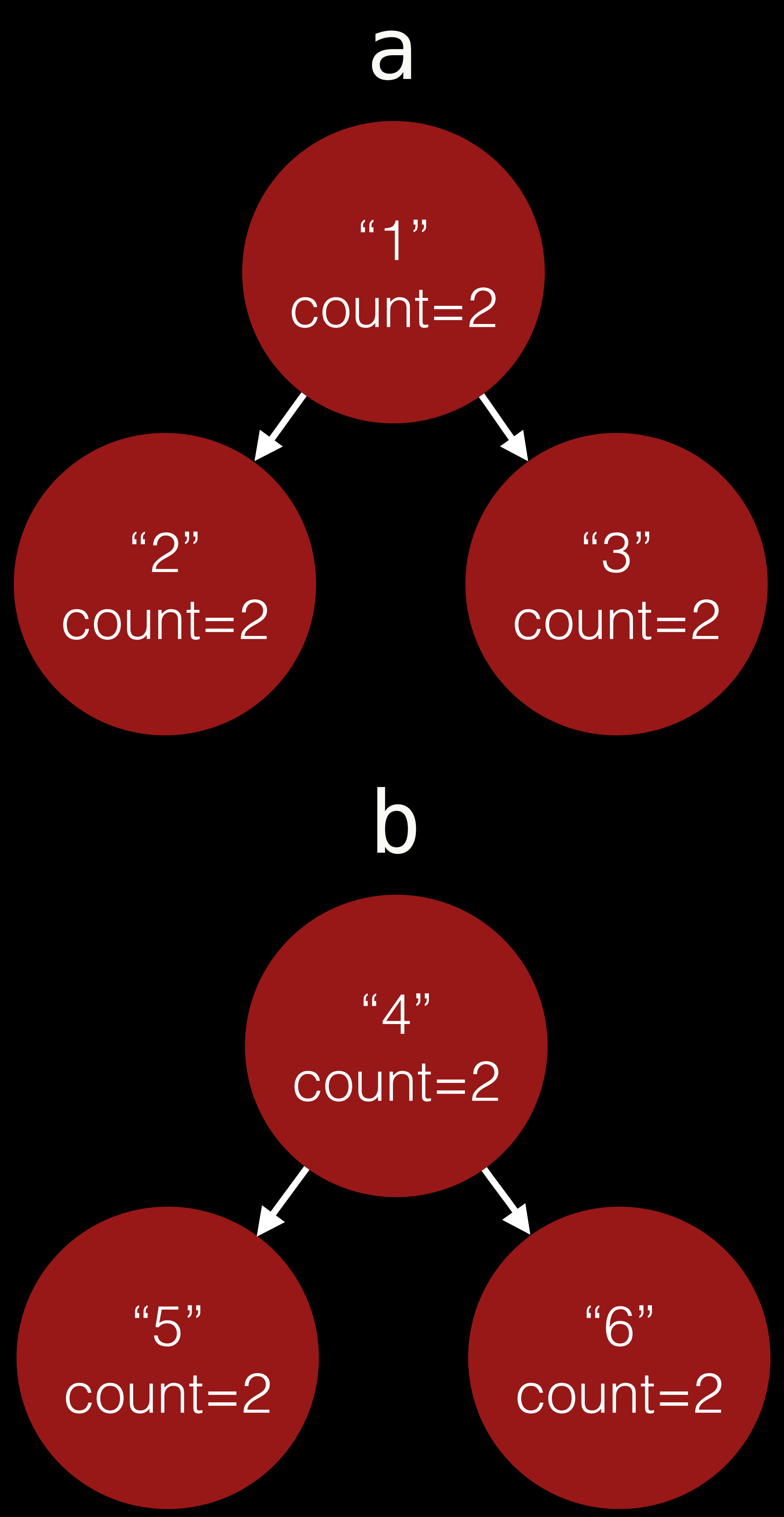
*If Lorax is used to design tree data structures there should never be a reference cycle.
The following example is contrived and of course not recommended.



```
/home/chris/Desktop/LoraxLanguageCompiler [git::master *] [chris@arch] [23:08]
> ./lorax -b leak.lrx leak

/home/chris/Desktop/LoraxLanguageCompiler [git::master *] [chris@arch] [23:08]
> valgrind ./leak
==6154== Memcheck, a memory error detector
==6154== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==6154== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==6154== Command: ./leak
==6154==
==6154==
==6154== HEAP SUMMARY:
==6154==     in use at exit: 176 bytes in 12 blocks
==6154==   total heap usage: 24 allocs, 12 frees, 352 bytes allocated
==6154==
==6154== LEAK SUMMARY:
==6154==    definitely lost: 32 bytes in 1 blocks
==6154==    indirectly lost: 144 bytes in 11 blocks
==6154==      possibly lost: 0 bytes in 0 blocks
==6154==    still reachable: 0 bytes in 0 blocks
==6154==         suppressed: 0 bytes in 0 blocks
==6154== Rerun with --leak-check=full to see details of leaked memory
==6154==
==6154== For counts of detected and suppressed errors, rerun with: -v
==6154== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

leak.lrx

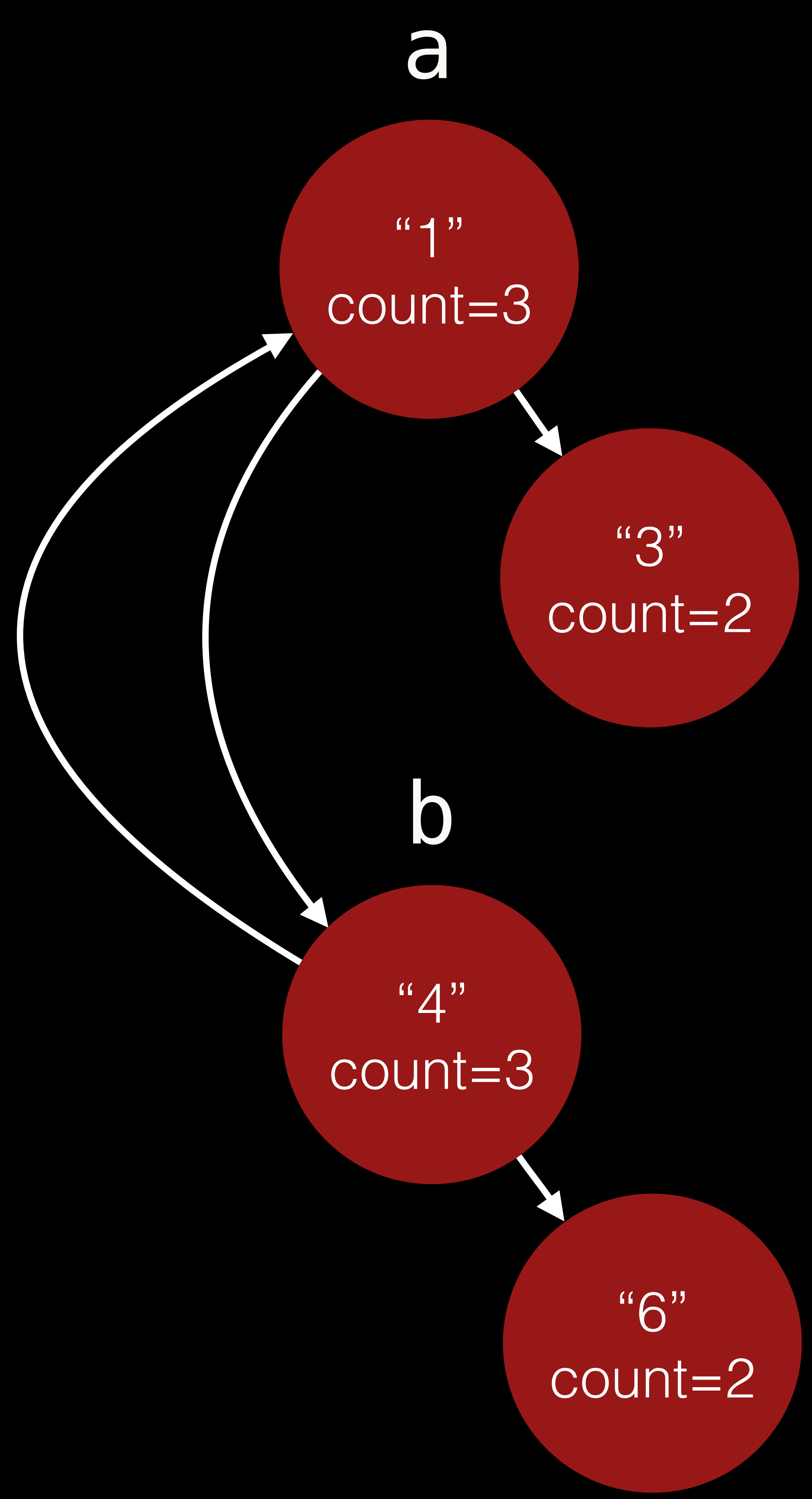
```
int main()
{
 tree <int>a(2);
 tree <int>b(2);

 a = 1[2, 3];
 b = 4[5, 6];

 a%0 = b;
 b%0 = a;

 return 0;
}
```



```
264 Bytes Allocated = {[malloc(32 bytes struct tree) + malloc(4 bytes for int count) + malloc(8 bytes for tree’s children pointers)] ∗ 3 trees} ∗ 2 Lorax Trees
264 Bytes in use.
```

leak.lrx

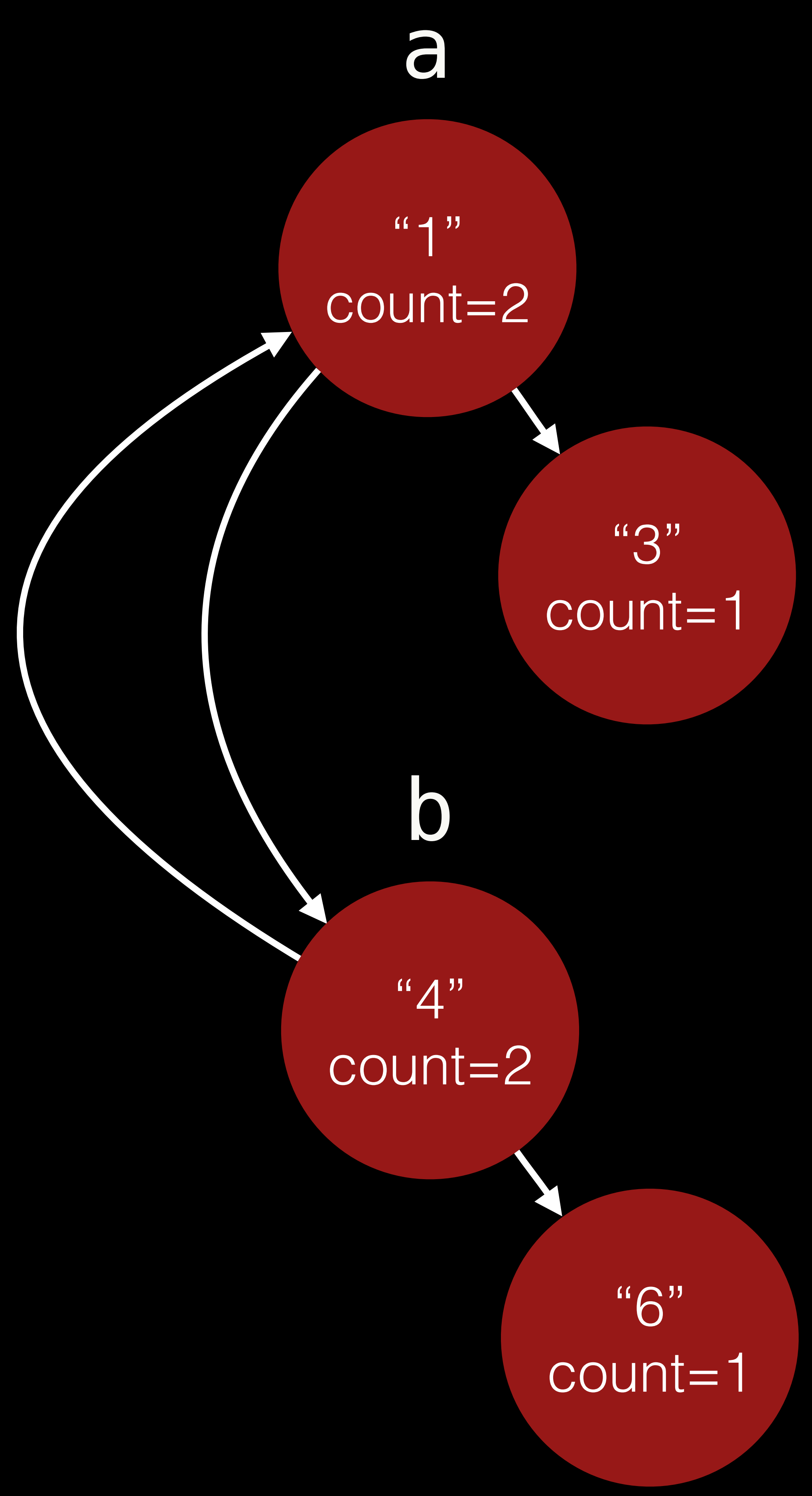
```
int main()
{
 tree <int>a(2);
 tree <int>b(2);

 a = 1[2, 3];
 b = 4[5, 6];

 a%0 = b;
 b%0 = a;

 return 0;
}
```



```
88 Bytes Freed = [free(32 bytes for struct tree) + free(4 bytes for counter) + free(8 bytes for children pointers)] ∗ 2 trees
176 Bytes in use.
```

leak.lrx

```
int main()
{
 tree <int>a(2);
 tree <int>b(2);

 a = 1[2, 3];
 b = 4[5, 6];

 a%0 = b;
 b%0 = a;

 return 0;
}
```



176 Bytes Lost!

What Happened?

A reference cycle and parents won’t release their children.

Assignment to Child

hello.lrx_lrxtmp.c

```c
__tmp_int_4 = 0;
__tmp_tree_datatype_int_degree_2_5 = lrx_access_child(&a_1, __tmp_int_4);
lrx_assign_tree_direct(__tmp_tree_datatype_int_degree_2_5, &b_1);

__tmp_int_2 = 0;
__tmp_tree_datatype_int_degree_2_3 = lrx_access_child(&b_1, __tmp_int_2);
lrx_assign_tree_direct(__tmp_tree_datatype_int_degree_2_3, &a_1);
```

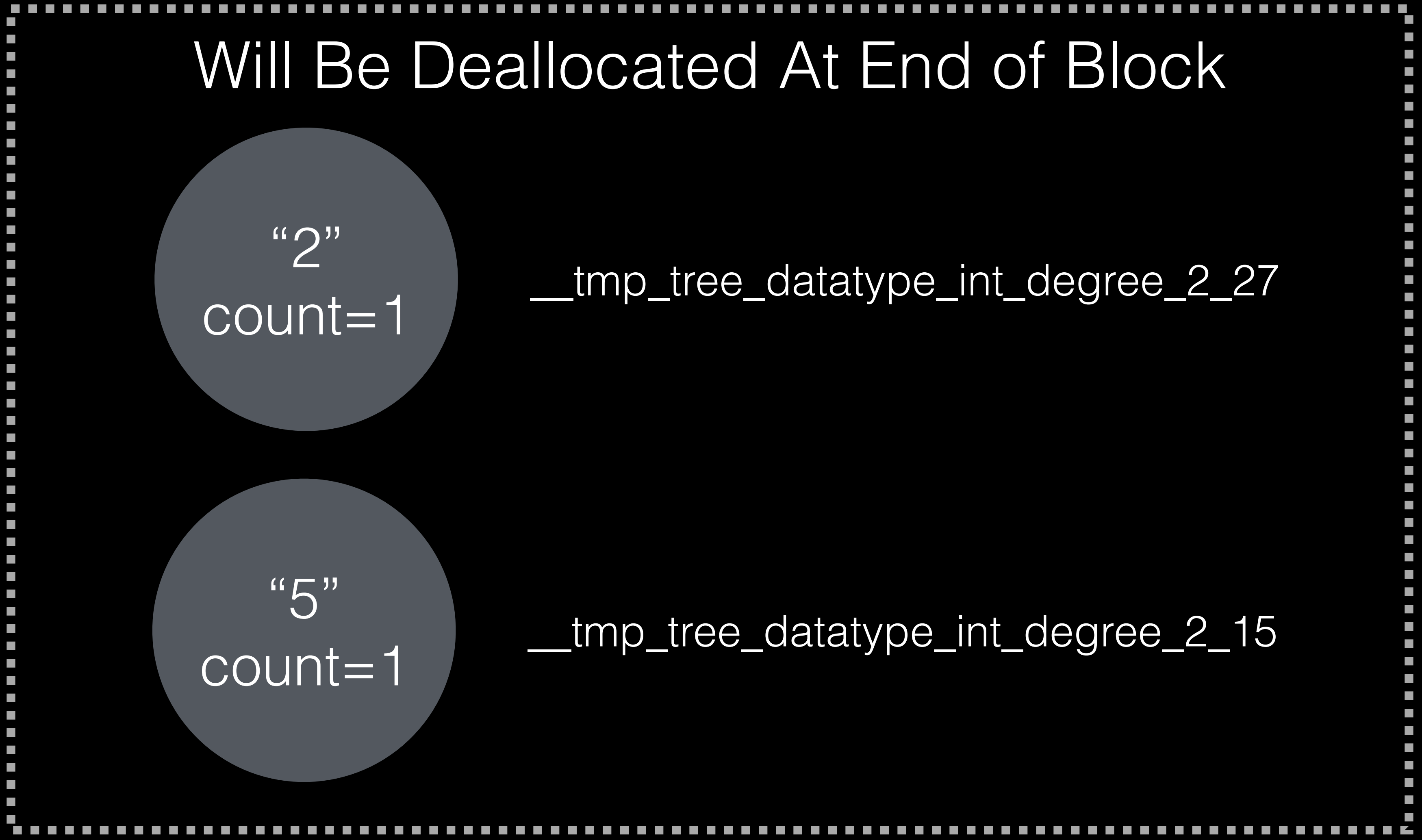
lrxlib.h

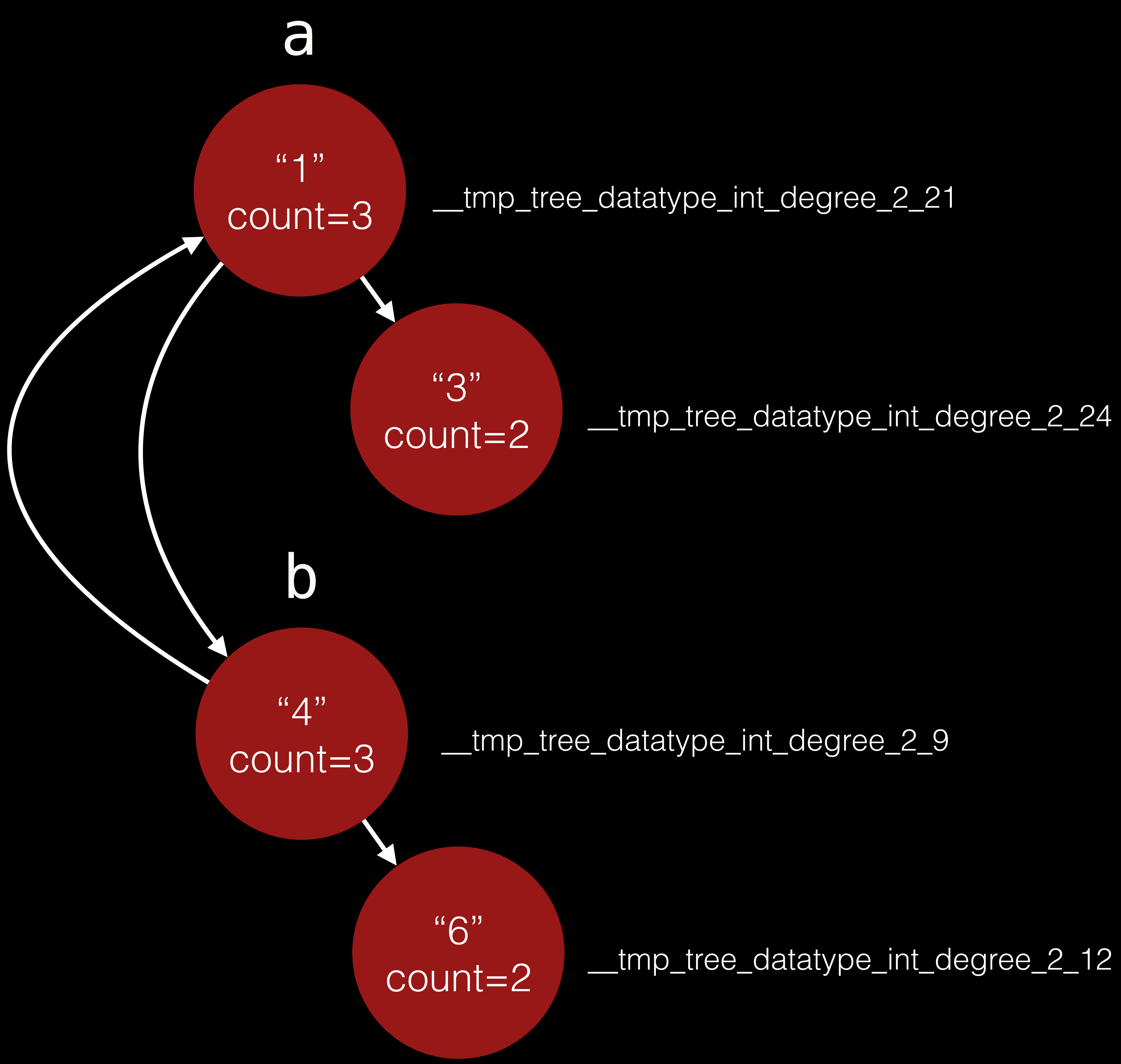
```c
/* t1 = t2%0 */
struct tree **lrx_access_child (struct tree **t, const int child) {
    assert(*t);
    assert(child < (*t)->degree);

    /* ptr to the parent's ptr to its child */
    return &((*t)->children[child]);
}
```

Assignment to Child

hello.lrx_lrxtmp.c

```
__tmp_int_4 = 0;
__tmp_tree_datatype_int_degree_2_5 = lrx_access_child(&a_1, __tmp_int_4);
lrx_assign_tree_direct(__tmp_tree_datatype_int_degree_2_5, &b_1);

__tmp_int_2 = 0;
__tmp_tree_datatype_int_degree_2_3 = lrx_access_child(&b_1, __tmp_int_2);
lrx_assign_tree_direct(__tmp_tree_datatype_int_degree_2_3, &a_1);
```


Will Be Deallocated At End of Block
"2"
count=1
__tmp_tree_datatype_int_degree_2_27
"5"
count=1
__tmp_tree_datatype_int_degree_2_15


a
"1"
count=3
__tmp_tree_datatype_int_degree_2_21
"3"
count=2
__tmp_tree_datatype_int_degree_2_24
b
"4"
count=3
__tmp_tree_datatype_int_degree_2_9
"6"
count=2
__tmp_tree_datatype_int_degree_2_12

```
88 Bytes Freed = [free(32 bytes for a_1 struct tree) + free(4 bytes for a_1’s counter) + free(8 bytes for a_1’s children pointers)] * 2 trees
176 Bytes in use.
```

Leaving Scope: Destroy Trees

hello.lrx_lrxtmp.c

```
lrx_destroy_tree(a_1);
lrx_destroy_tree(b_1);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_21);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_27);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_24);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_9);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_15);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_12);
```

lrxlib.h

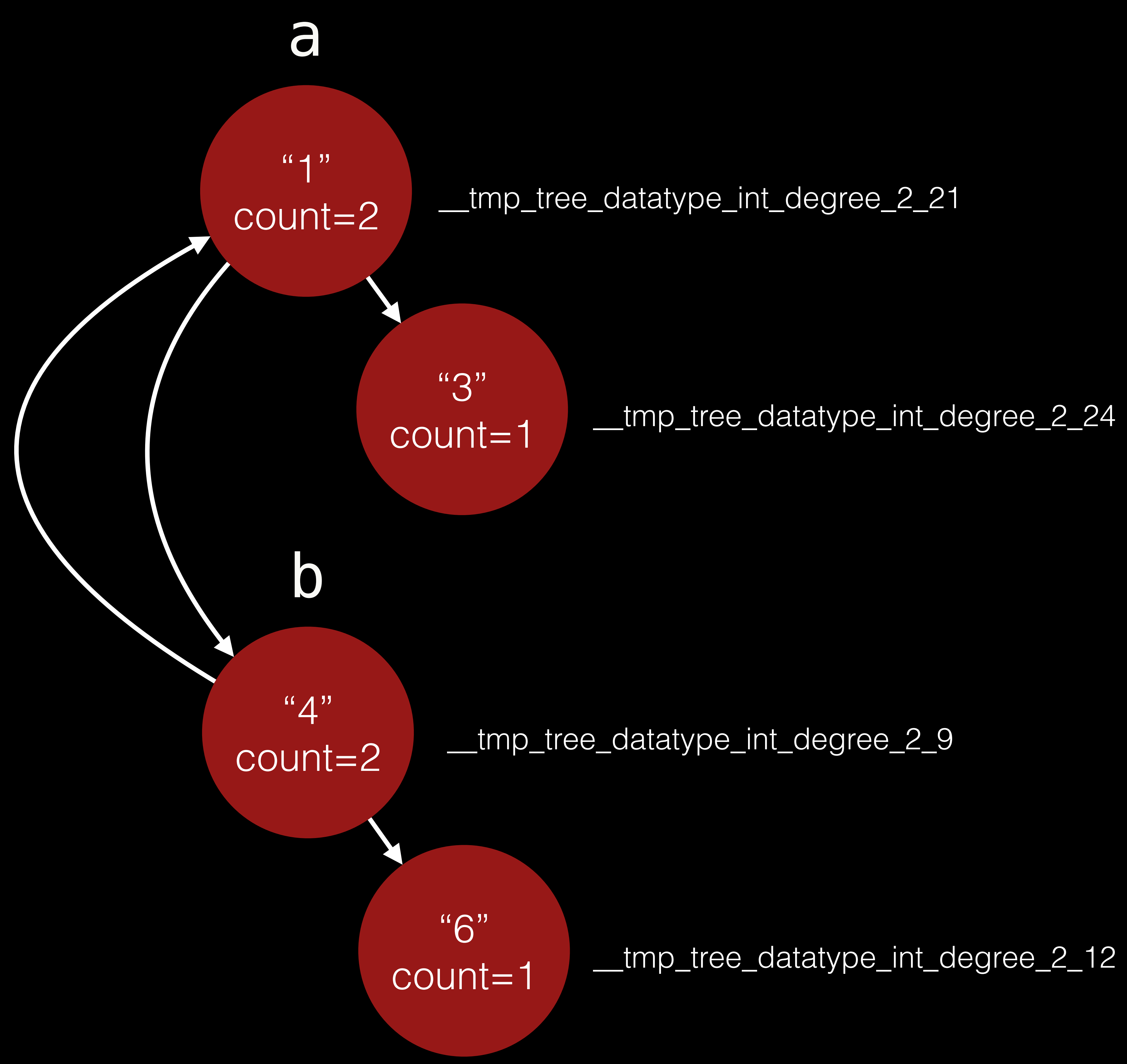
```
void lrx_destroy_tree(struct tree *t) {

    if (t == NULL) {
        return;
    }

    *(t->count) -= 1;
    if (*(t->count) == 0) {

        if (t->children) {
            int i;
            for (i = 0; i < t->degree; ++i){
                lrx_destroy_tree(t->children[i]);
            }
            free(t->children);
        }

        free(t->count);
        free(t);
    }
}
```

Decrement and Destroy When Count == 0

Leaving Scope: Destroy Trees

hello.lrx_lrxtmp.c

```
lrx_destroy_tree(a_1);
lrx_destroy_tree(b_1);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_21);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_27);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_24);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_9);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_15);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_12);
```


a
"1"
count=2
__tmp_tree_datatype_int_degree_2_21
"3"
count=1
__tmp_tree_datatype_int_degree_2_24
b
"4"
count=2
__tmp_tree_datatype_int_degree_2_9
"6"
count=1
__tmp_tree_datatype_int_degree_2_12

176 Bytes Lost!