Programming in C and C++

Lecture 6: Aliasing, Graphs, and Deallocation

David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

The C API for Dynamic Memory Allocation

- `void *malloc(size_t size)`

  Allocate a pointer to an object of size `size`
- `void free(void *ptr)`

  Deallocate the storage `ptr` points to
- Each allocated pointer must be deallocated exactly once along each execution path through the program.
- Once deallocated, the pointer must not be used any more.

One Deallocation Per Path

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int *pi = malloc(sizeof(int));
  scanf("%d", pi);                    // Read an int
  if (*pi % 2) {
    printf("Odd!\n");
    free(pi);                         // WRONG!
  }
}
```

One Deallocation Per Path

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int *pi = malloc(sizeof(int));
  scanf("%d", pi);                    // Read an int
  if (*pi % 2) {
    printf("Odd!\n");
    free(pi);                         // WRONG!
  }
}
```

- This code fails to deallocate `pi` if `*pi` is even

One Deallocation Per Path

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int *pi = malloc(sizeof(int));
  scanf("%d", pi);                    // Read an int
  if (*pi % 2) {
    printf("Odd!\n");
  }
  free(pi);                           // OK!
}
```

- This code fails to deallocate `pi` if `*pi` is even
- Moving it ensures it always runs

A Tree Data Type

```
1       struct node {
2         int value;
3         struct node *left;
4         struct node *right;
5       };
6       typedef struct node Tree;
```

- This is the tree type from Lab 4.
- It has a value, a left subtree, and a right subtree
- An empty tree is a `NULL` pointer.

A Tree Data Type

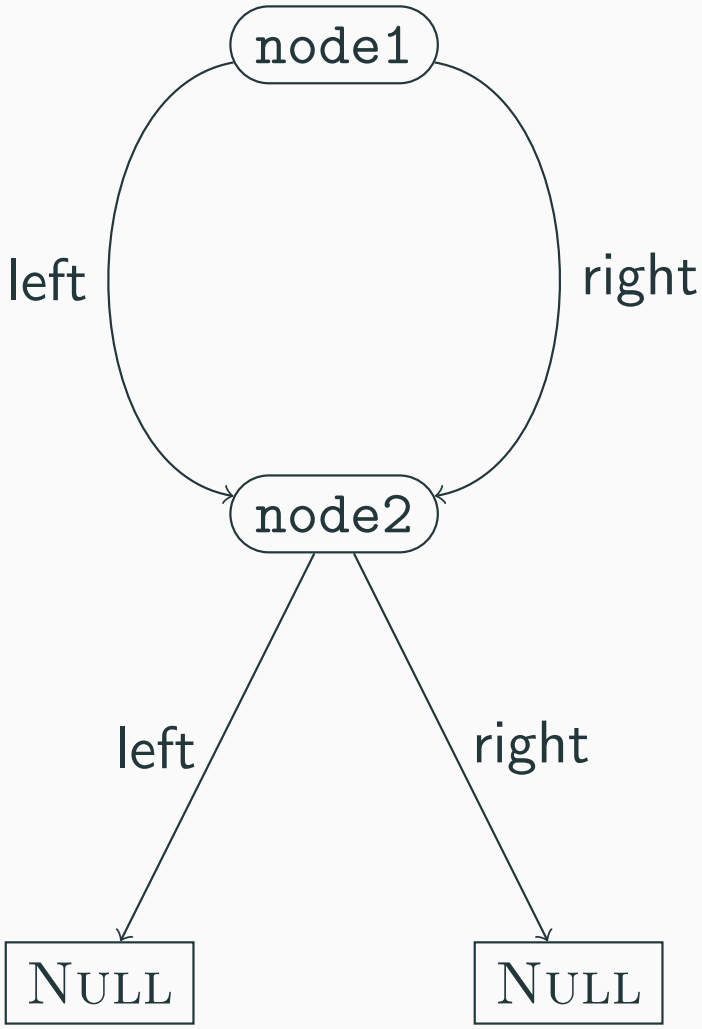
```
1   Tree *node(int value, Tree *left, Tree *right) {
2     Tree *t = malloc(sizeof(tree));
3     t->value = value;
4     t->right = right;
5     t->left = left;
6     return t;
7   }
8   void tree_free(Tree *tree) {
9     if (tree != NULL) {
10      tree_free(tree->left);
11      tree_free(tree->right);
12      free(tree);
13    }
14  }
```

A Directed Acyclic Graph (DAG)

```
1       // Initialize node2
2       Tree *node2 = node(2, NULL, NULL);
3
4       // Initialize node1
5       Tree *node1 = node(1, node2, node2); // node2 repeated
6
7       // note node1->left == node1->right == node2!
```

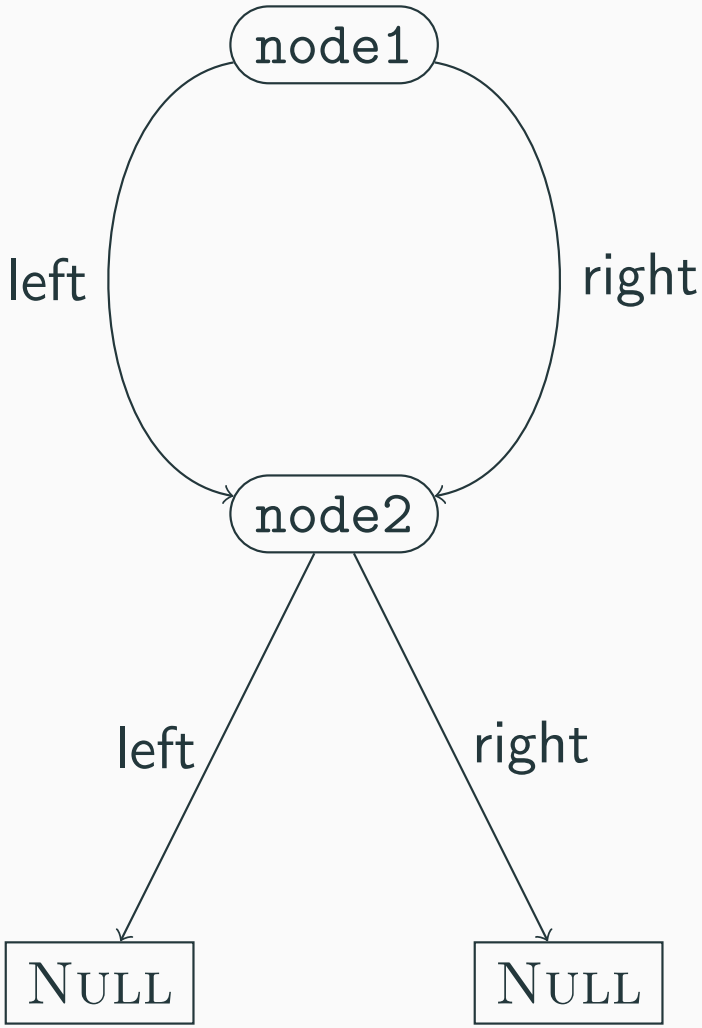
What kind of “tree” is this?

The shape of the graph



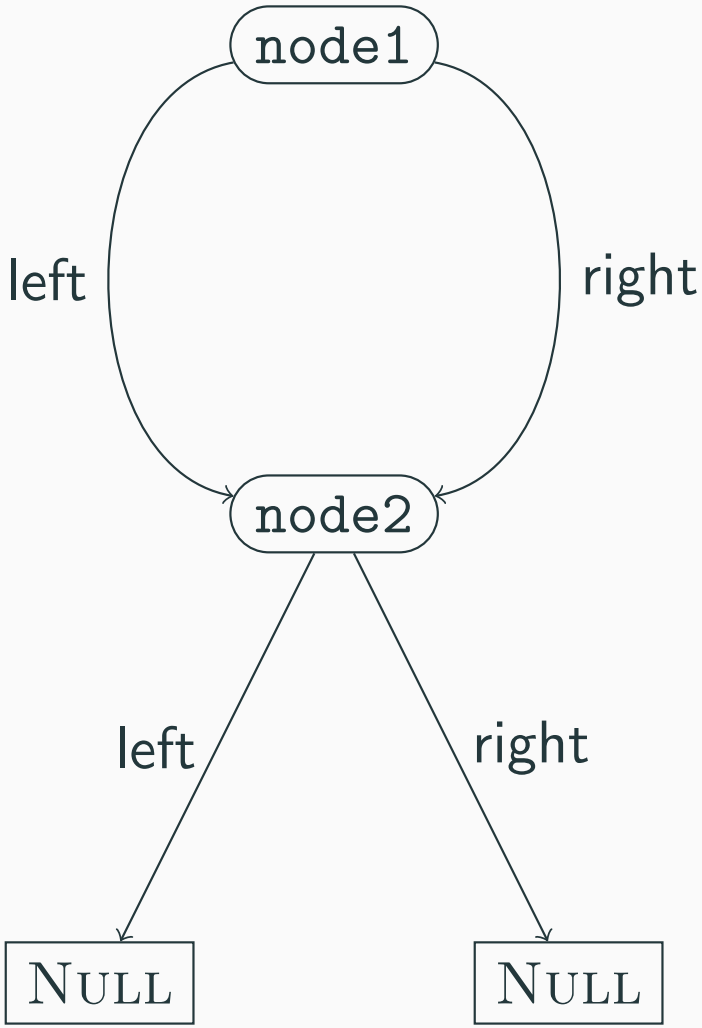
- `node1` has *two* pointers to `node2`
- This is a directed acyclic graph, not a tree.
- `tree_free(node1)` will call `tree_free(node2)` *twice*!

Evaluating `free(node1)`



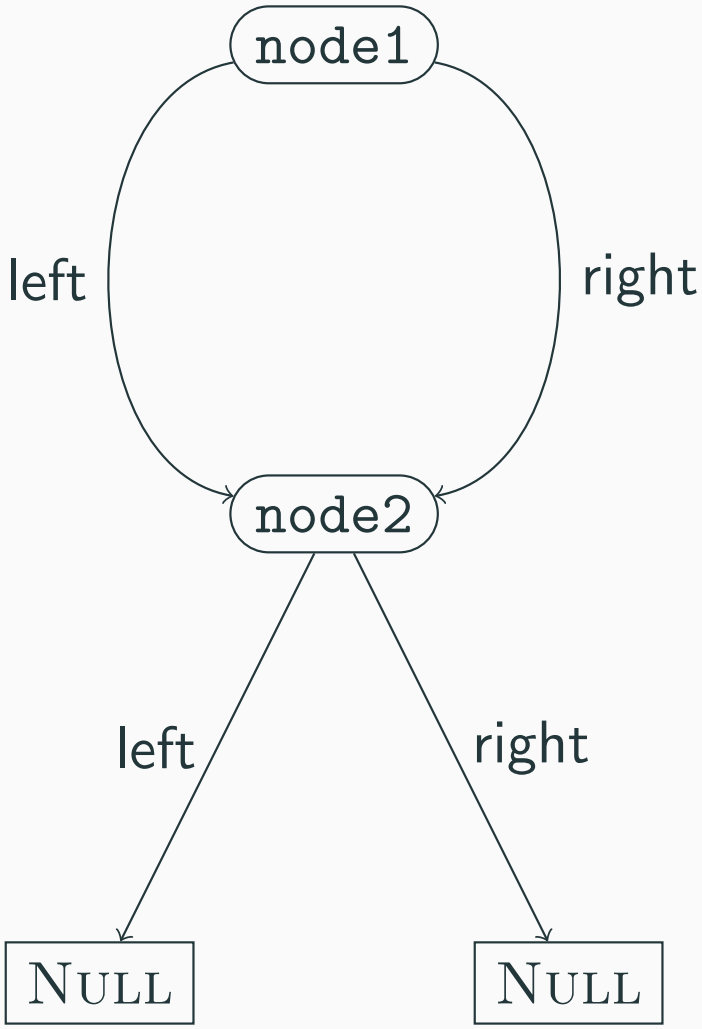

```
1    free(node1);
```

Evaluating `free(node1)`



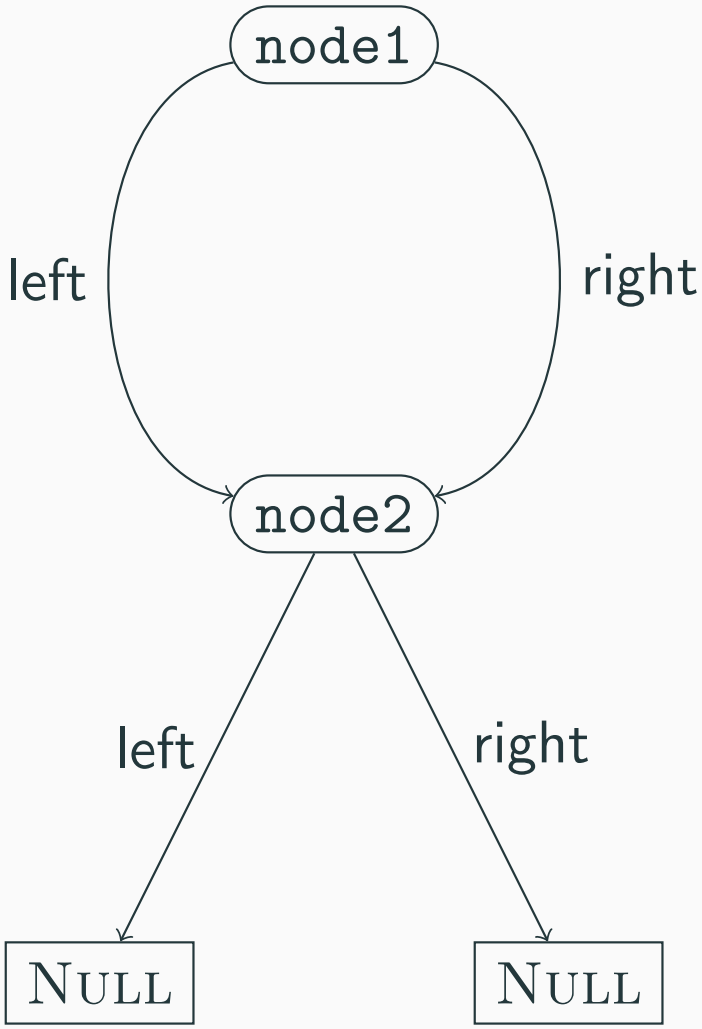

```
1  if (node1 != NULL) {
2    tree_free(node1->left);
3    tree_free(node1->right);
4    free(node1);
5  }
```

Evaluating `free(node1)`



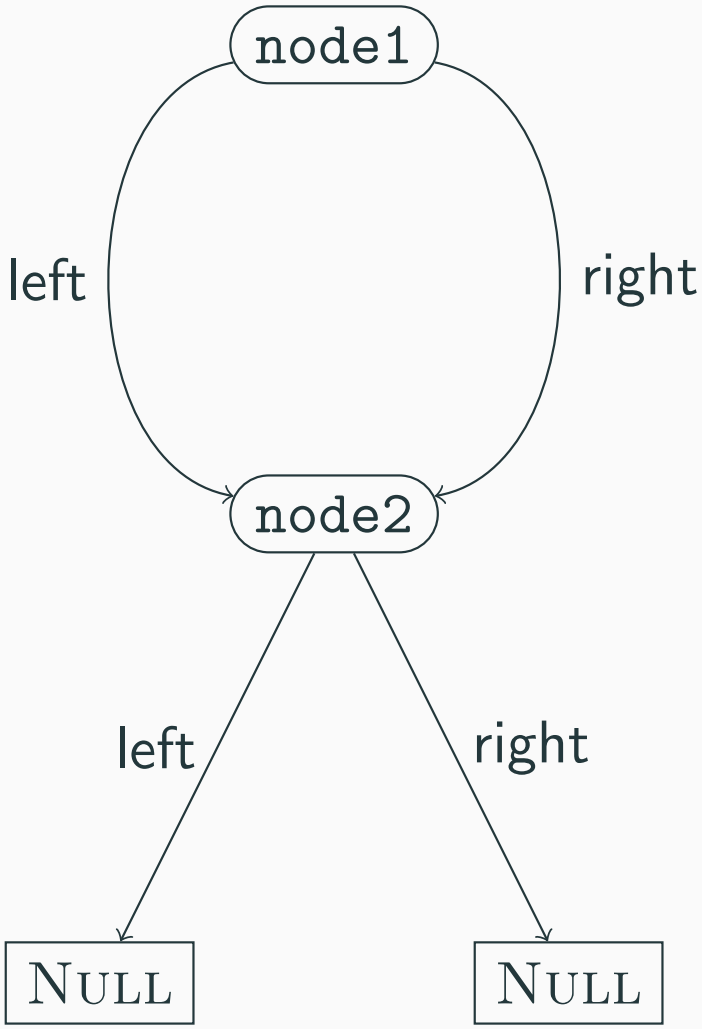

```
1    tree_free(node1->left);
2    tree_free(node1->right);
3    free(node1);
```

Evaluating `free(node1)`



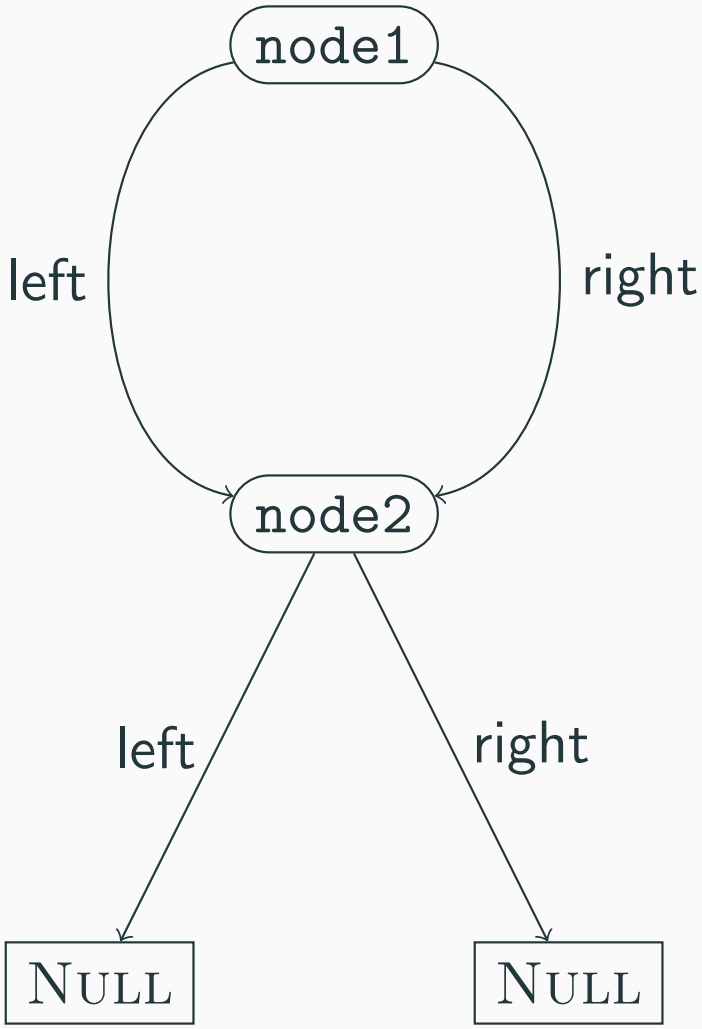

```
1    tree_free(node2);
2    tree_free(node2);
3    free(node1);
```

Evaluating `free(node1)`



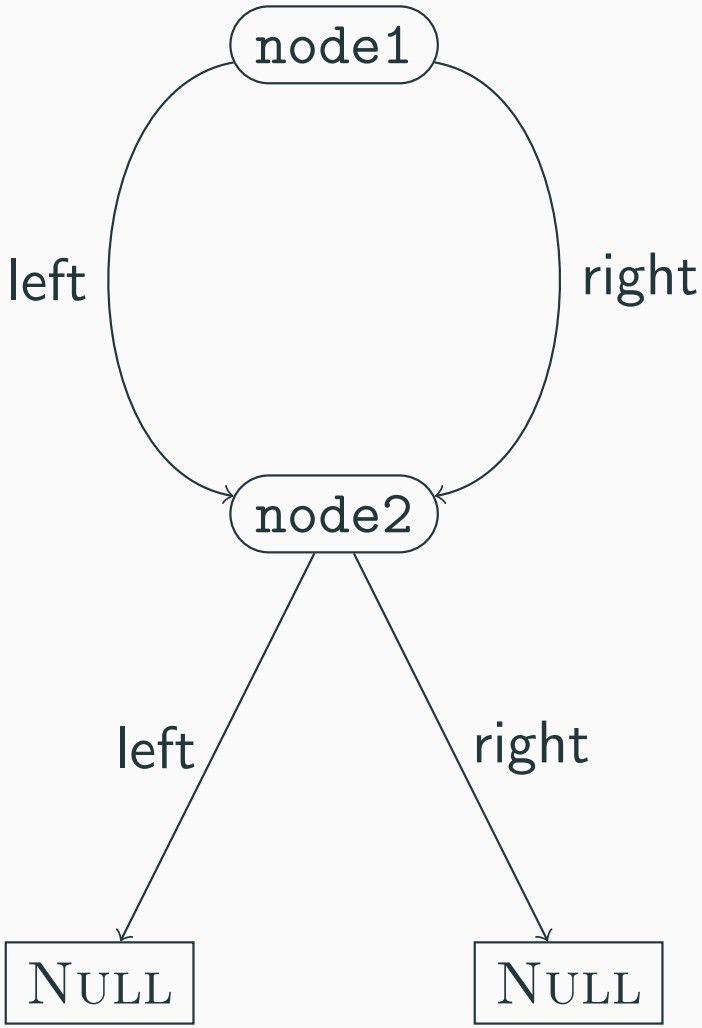

```
1    if (node2 != NULL) {
2      tree_free(node2->left);
3      tree_free(node2->right);
4      free(node2);
5    }
6    tree_free(node2);
7    free(node1);
```

Evaluating `free(node1)`



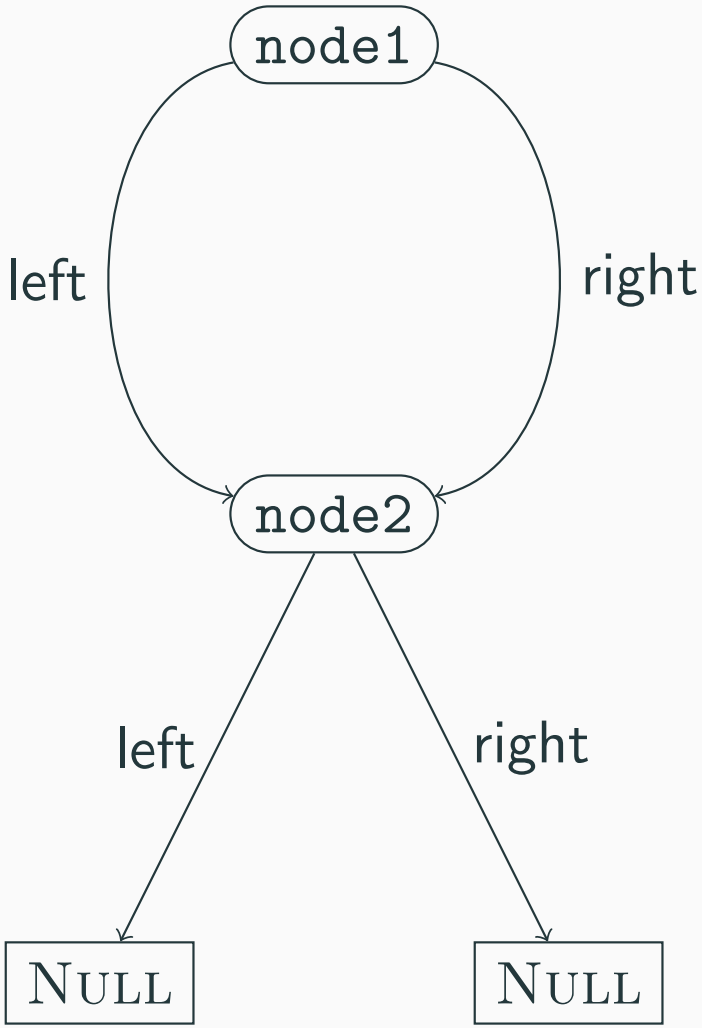

```
1    tree_free(node2->left);
2    tree_free(node2->right);
3    free(node2);
4    tree_free(node2);
5    free(node1);
```

Evaluating `free(node1)`



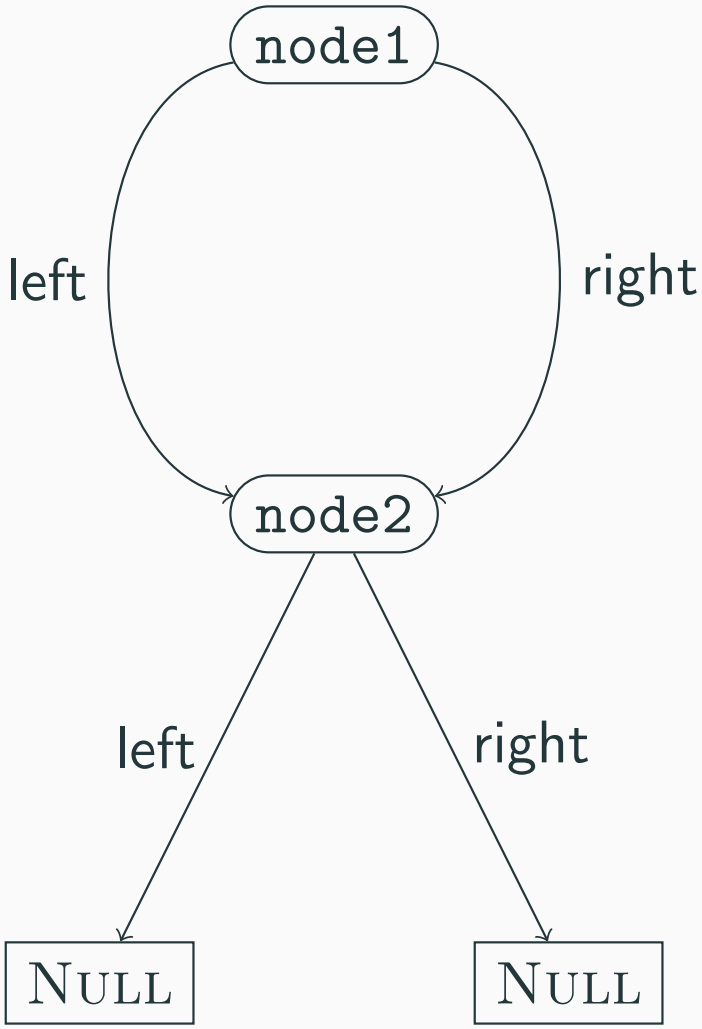

```
1    tree_free(NULL);
2    tree_free(NULL);
3    free(node2);
4    tree_free(node2);
5    free(node1);
```

Evaluating `free(node1)`



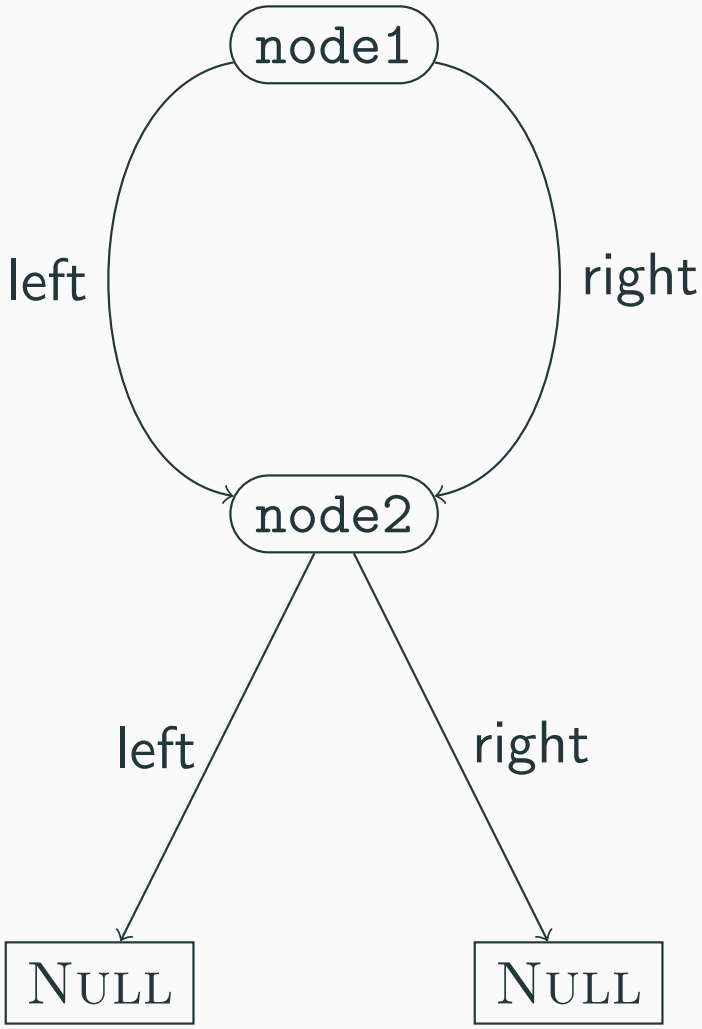

```
1    if (NULL != NULL) {
2      tree_free(NULL->left);
3      tree_free(NULL->right);
4      free(node1);
5    }
6    tree_free(NULL);
7    free(node2);
8    tree_free(node2);
9    free(node1);
```

Evaluating `free(node1)`



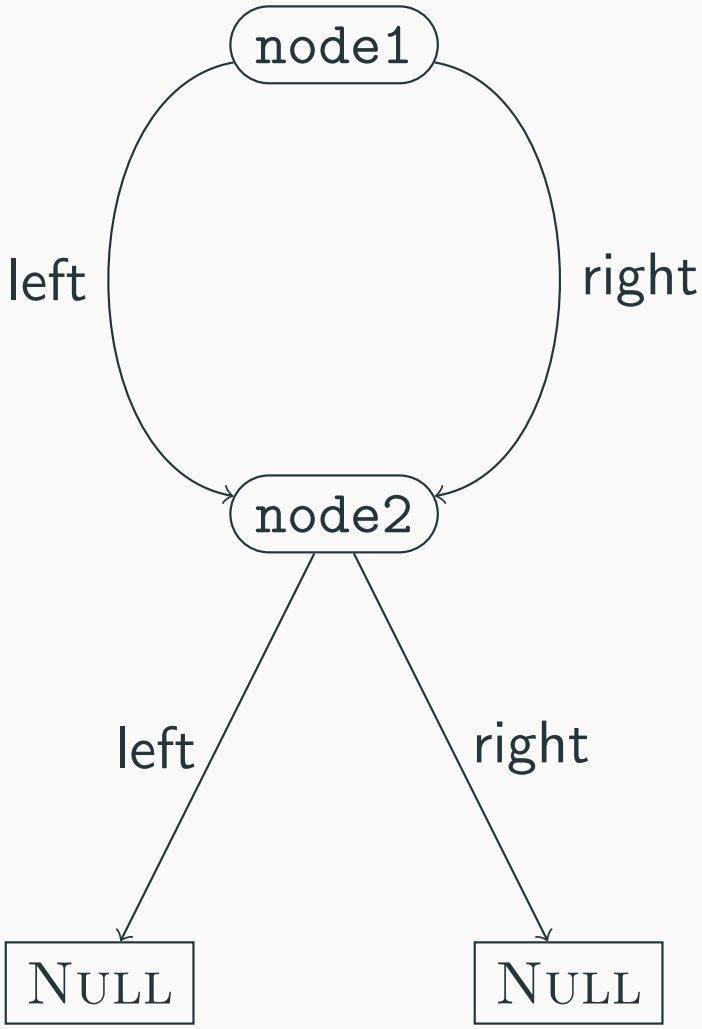

```
1    tree_free(NULL);
2    free(node2);
3    tree_free(node2);
4    free(node1);
```

Evaluating `free(node1)`



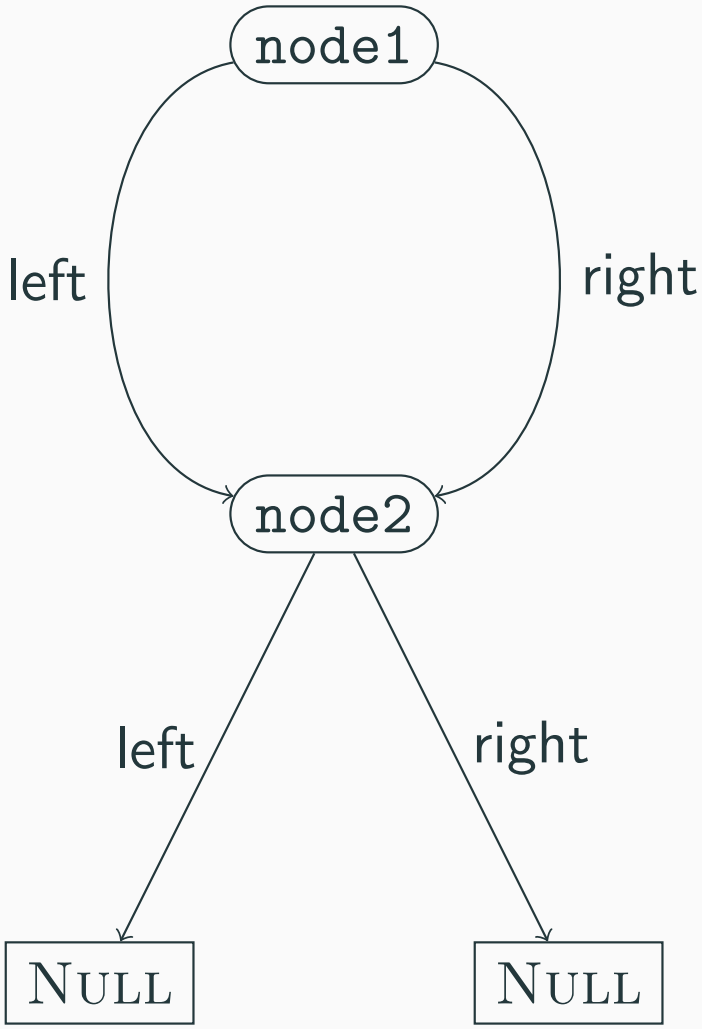

```
1    free(node2);
2    tree_free(node2);
3    free(node1);
```

Evaluating `free(node1)`




```
1    free(node2);
2    free(node2);
3    free(node1);
```

Evaluating `free(node1)`




```
1    free(node2);
2    free(node2);
3    free(node1);
```

`node2` is freed twice!

A Tree Data Type which Tracks Visits

```
1       struct node {
2         bool visited;
3         int value;
4         struct node *left;
5         struct node *right;
6       };
7       typedef struct node Tree;
```

- This tree has a value, a left subtree, and a right subtree
- An empty tree is a `NULL` pointer.
- it also has a *visited* field.

Creating Nodes of Tree Type

```
1  Tree *node(int value, Tree *left, Tree *right) {
2    Tree *t = malloc(sizeof(tree));
3    t->visited = false;
4    t->value = value;
5    t->right = right;
6    t->left = left;
7    return t;
8  }
```

1. Constructing a node sets the visited field to false
2. Otherwise returns the same fresh node as before

Freeing Nodes of Tree Type, Part 1

```
1   typedef struct TreeListCell TreeList;
2   struct TreeListCell {
3     Tree *head;
4     TreeList *tail;
5   }
6   TreeList *cons(Tree *head, TreeList *tail) {
7     TreeList *result = malloc(TreeListCell);
8     result->head = head;
9     result->tail = tail;
10    return result;
11  }
```

- This defines `TreeList` as a type of lists of tree nodes.
- `cons` dynamically allocates a new element of a list.

Freeing Nodes of Tree Type, Part 2

```
1   TreeList *getNodes(Tree *tree, TreeList *nodes) {
2     if (tree == NULL || tree->visited) {
3       return nodes;
4     } else {
5       tree->visited = true;
6       nodes = cons(tree, nodes);
7       nodes = getNodes(tree->right, nodes);
8       nodes = getNodes(tree->left, nodes);
9       return nodes;
10    }
11  }
```

- Add the unvisited nodes of `tree` to `nodes`.
- Finish if the node is a leaf or already visited
- Otherwise, add the current node and recurse

Freeing Nodes of Tree Type, Part 3

```
1   void tree_free(Tree *tree) {
2     NodeList *nodes = getNodes(tree, NULL);
3     while (nodes != NULL) {
4       Tree *head = nodes->head;
5       NodeList *tail = nodes->tail;
6       free(head);
7       free(nodes);
8       nodes = tail;
9     }
10  }
```

- To free a tree, get all the unique nodes in a list
- Iterate over the list, freeing the nodes
- Don't forget to free the list!
- We're doing dynamic allocation to free some data...

Summary

- Freeing trees is relatively easy
- Freeing DAGs or general graphs is much harder
- Freeing objects at most once is harder if there are multiple paths to them.

Arenas

```
1       struct node {
2         int value;
3         struct node *left;
4         struct node *right;
5       };
6       typedef struct node Tree;
```

- This is the original tree data type
- Let's keep this type, but change the (de)allocation API

Arenas

```
1   typedef struct arena *arena_t;
2   struct arena {
3     int size;
4     int current;
5     Tree *elts;
6   };
7
8   arena_t make_arena(int size) {
9     arena_t arena = malloc(sizeof(struct arena));
10    arena->size = size;
11    arena->current = 0;
12    arena->elts = malloc(size * sizeof(Tree));
13    return arena;
14  }
```

Arena allocation

```
1  Tree *node(int value, Tree *left, Tree *right,
2             arena_t arena) {
3    if (arena->current < arena->size) {
4      Tree *t = arena->elts + arena->current;
5      arena->current += 1;
6      t->value = value, t->left = left, t->right = right;
7      return t;
8    } else
9      return NULL;
10 }
```

To allocate a node from an arena:

1. Initialize `current` element
2. Increment `current`
3. Return the initialized node

Freeing an Arena

```
1    void free_arena(arena_t arena) {
2      free(arena->elts);
3      free(arena);
4    }
```

- We no longer free trees individually
- Instead, free a whole arena at a time
- All tree nodes allocated from the arena are freed at once

Example

```
1  arena_t a = make_arena(BIG_NUMBER);
2
3  Tree *node1 = node(0, NULL, NULL, a);
4  Tree *node2 = node(1, node1, node1, a); // it's a DAG now
5  // do something with the nodes...
6  free_arena(a);
```

- We allocate the arena
- We can build an arbitrary graph
- And free all the elements at once

Conclusion

- Correct memory deallocation in C requires thinking about control flow
- This can get tricky!
- Arenas are an idiom for (de)allocating big blocks at once
- Reduces need for thinking about control paths
- But can increase working set sizes