COS 217: Introduction to Programming Systems

Assignment 4:
Directory and File Trees

PRINCETON UNIVERSITY

Assignment 4 Goals

1. Gain more familiarity with data structures (lecture 10, precepts 10, 11, and 15)
   - Beyond the simplest linked lists, trees
   - Introduce the Abstract Object (AO) model
   - Similar to Abstract Data Type (ADT), but there's only one of them
   - Don't pass an "object" to functions – they implicitly use the appropriate `static` variables

Abstract Data Type

```
struct myADT {
    int var1, var2;
};
typedef struct myADT *myADT_T;

void myADT_func1(myADT_T obj, int param)
{ ... }
```

Abstract Object

```
static int myAO_var1, myAO_var2;



void myAO_func1(int param)
{ ... }
```

Assignment 4 Goals

1. Gain more familiarity with data structures (lecture 10, precepts 10, 11, and 15)
2. Practice debugging (lecture 11, precepts 5 and 9)
   - Especially using `gdb`

Assignment 4 Goals

1. Gain more familiarity with data structures (lecture 10, precepts 10, 11, and 15)
2. Practice debugging (lecture 11, precepts 5 and 9)
3. Take responsibility for your own testing (lecture 9)
   - Much of the testing code will not be written for you
   - You will write a "checker" that verifies an AO's internal state to make sure it's sound

Assignment 4 Goals

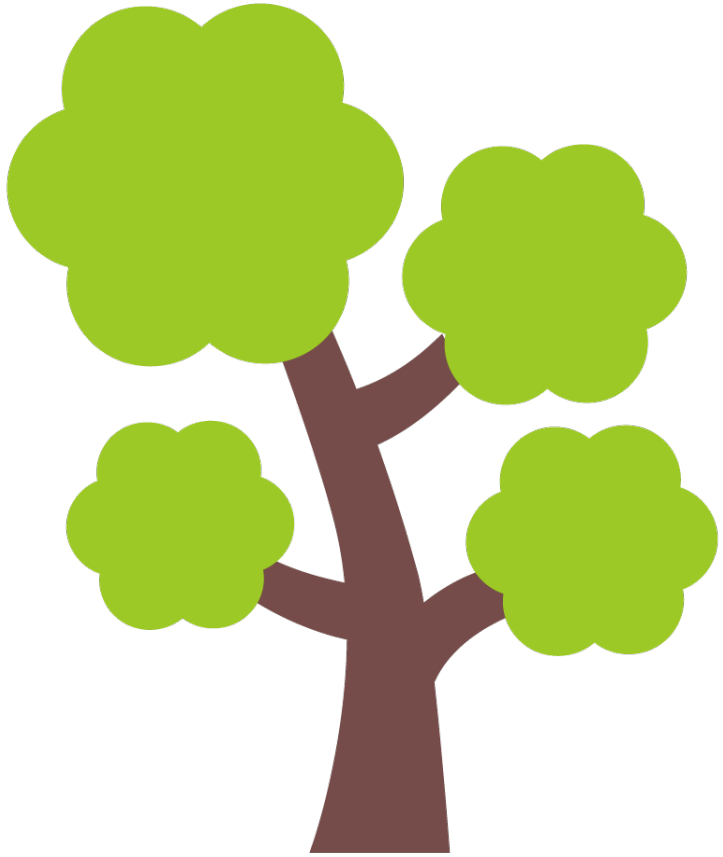
1. Gain more familiarity with data structures (lecture 10, precepts 10, 11, and 15)
2. Practice debugging (lecture 11, precepts 5 and 9)
3. Take responsibility for your own testing (lecture 9)
4. Design your own modules and interfaces (lecture 12)
   - We will give you a high-level interface and client code
   - You will decide what other modules to write, and what interfaces they have

Assignment 4 Goals

1. Gain more familiarity with data structures (lecture 10, precepts 10, 11, and 15)
2. Practice debugging (lecture 11, precepts 5 and 9)
3. Take responsibility for your own testing (lecture 9)
4. Design your own modules and interfaces (lecture 12)
5. Read code that you didn't write
   - Unusual assignment: large parts of it don't involve writing code
   - Mimics the real world: you won't re-write GooBook from scratch on day 1

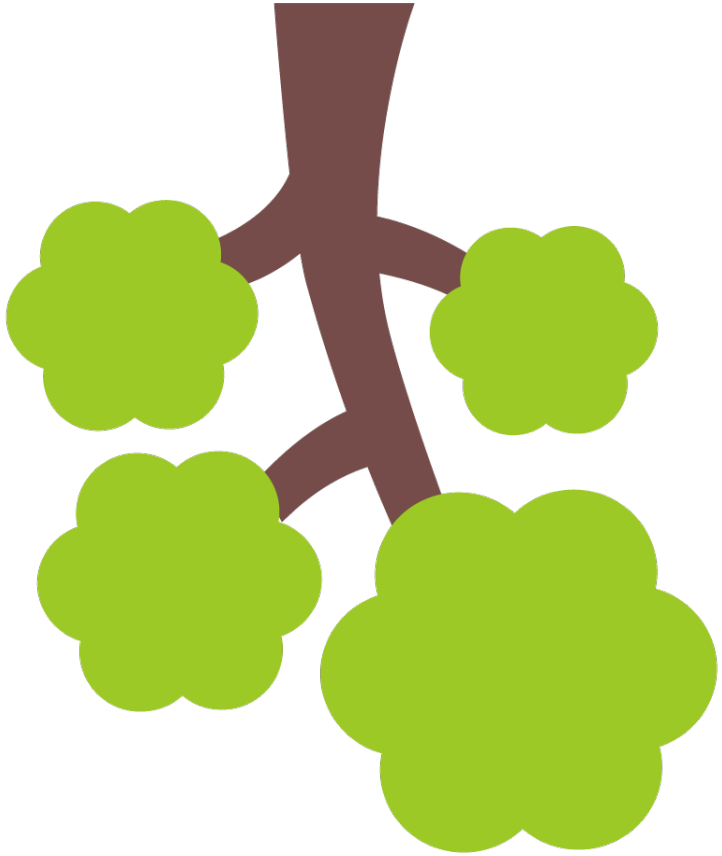
The Sting (1973)

Trees

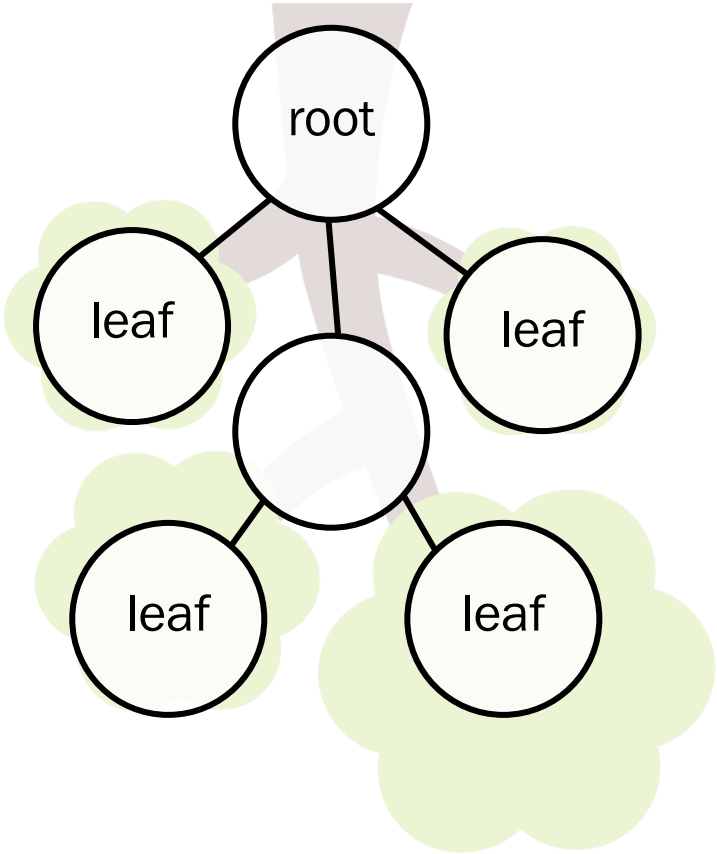


stockio.com

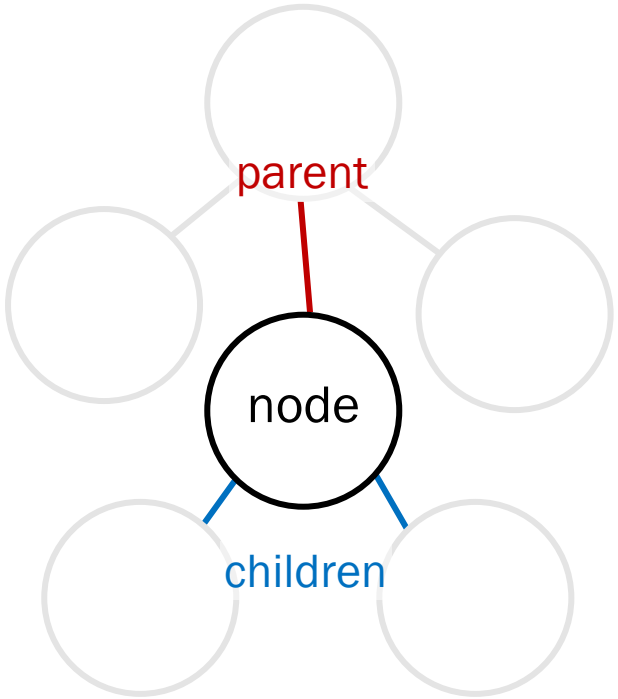
Trees (as seen by computer scientists)



Trees (as implemented by computer scientists)



Trees (as implemented by computer scientists)

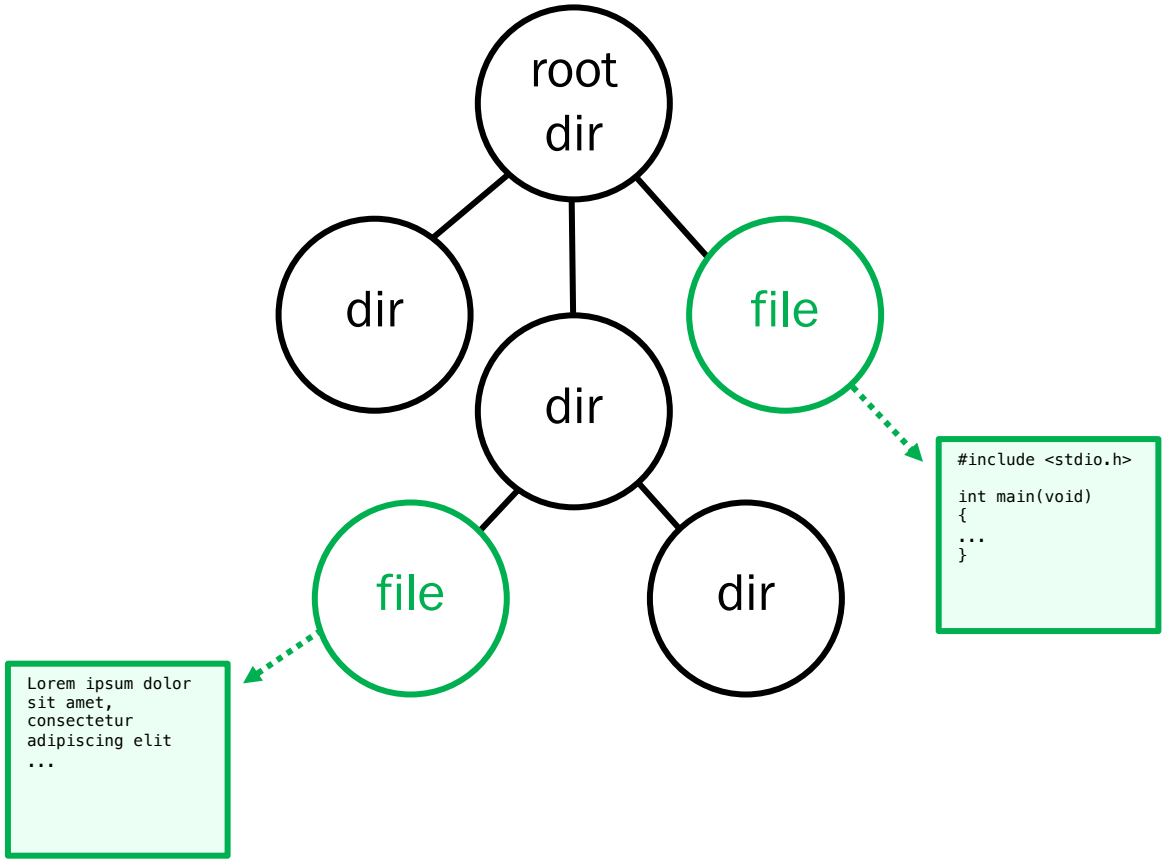


Trees and Filesystems

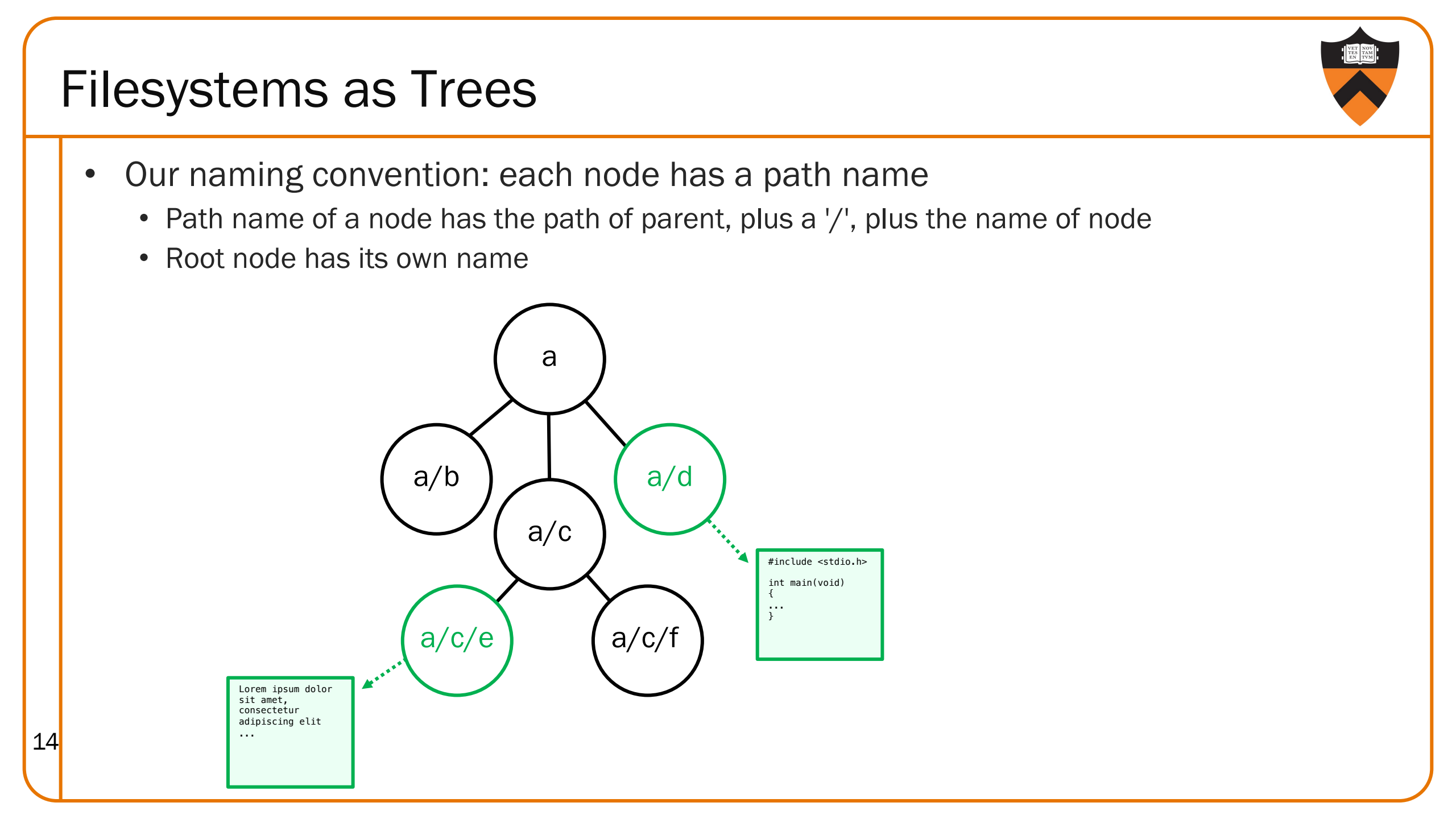
- Trees encode hierarchical relationships
- So do filesystems
  - A directory can hold files or other directories ("folders", for those who have succumbed to the brainwashing)
  - All directories and files reachable from the root

Filesystems as Trees

- Small extension of plain trees
  - All interior nodes are directories
  - *Some* leaves are files, with associated contents

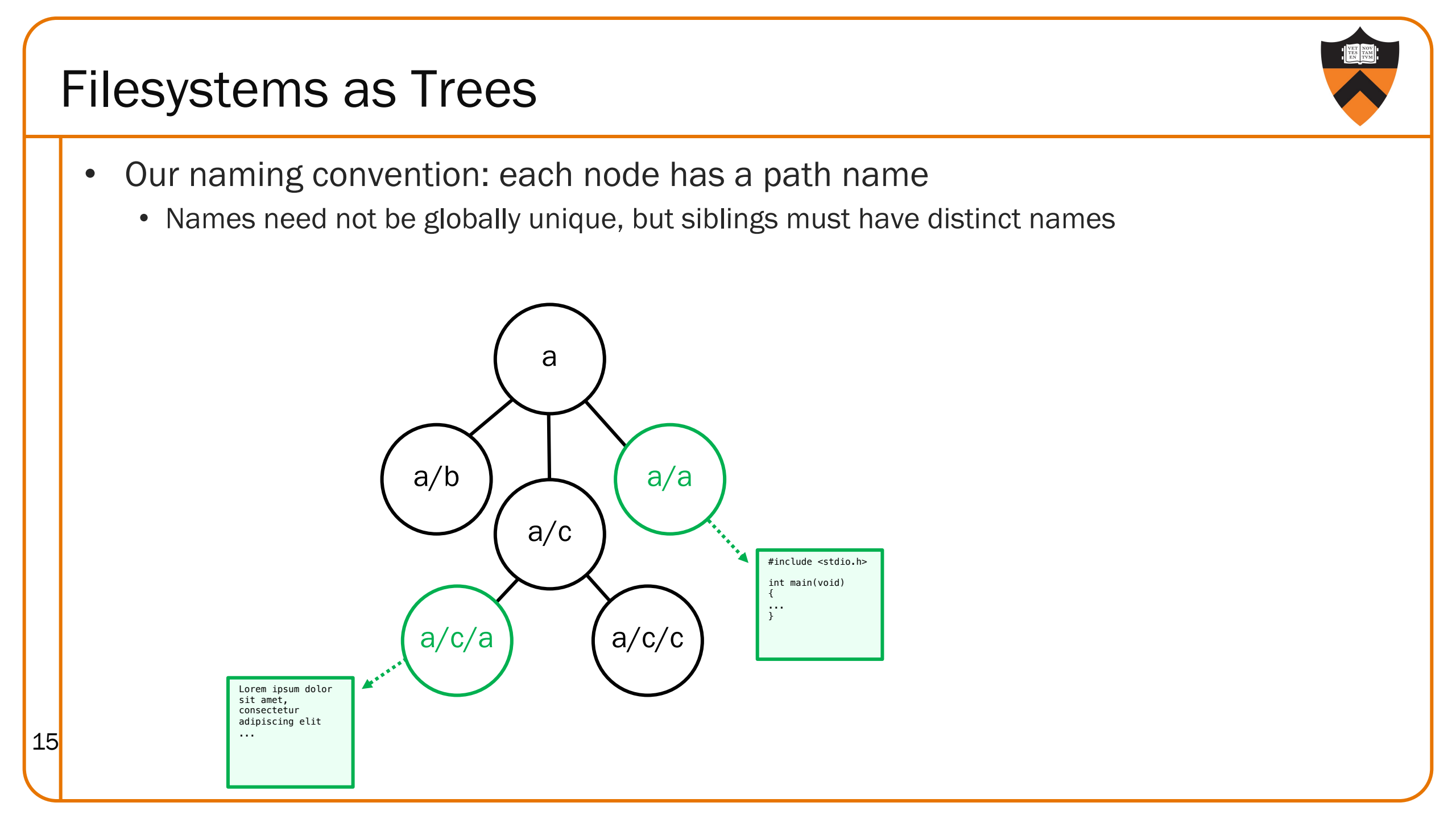


Filesystems as Trees



- Our naming convention: each node has a path name
  - Path name of a node has the path of parent, plus a '/', plus the name of node
  - Root node has its own name

Filesystems as Trees



- Our naming convention: each node has a path name
  - Names need not be globally unique, but siblings must have distinct names

A4 Premise

- Someone has created a filesystem-as-tree API
  … plus some questionable implementations
- You have access to the API, and client code
- You do not have access to the implementations
- Parts 1 and 2: figure out why the implementations are buggy
- Part 3: refactor, rework, and extend a partial implementation to match new API

PART 1

Part 1 Simplifications

Simplification #1: no files – everything's a directory. (Also for part 2.)

Simplification #2: binary trees – no more than 2 children per node.

Put these together, and we have Binary Directory Trees (BDTs).

Part 1 API

Summary of API in `bdt.h` (but read it yourself for details, including error handling!)

`int BDT_init(void);`	Sets the data structure to initialized status.
`int BDT_destroy(void);`	Removes all contents and returns data structure to uninitialized status.
`int BDT_insertPath(char* path);`	Inserts a new path into the tree, if possible. (Like mkdir -p)
`boolean BDT_containsPath(char* path);`	Does the tree contain a Node representing the full path parameter?
`int BDT_rmPath(char* path);`	Removes the directory hierarchy rooted at path. (Like rm -r)
`char* BDT_toString(void);`	Returns a string representation of the data structure.

Part 1 Functionality

So, how does this work? Let's take a look at some excerpts from `bdt_client.c`

```
assert(BDT_init() == SUCCESS);
assert(BDT_insertPath("a/b/c") == SUCCESS);
assert(BDT_containsPath("a") == TRUE);
assert(BDT_containsPath("a/b") == TRUE);
assert(BDT_containsPath("a/b/c") == TRUE);
assert((temp = BDT_toString()) != NULL);
fprintf(stderr, "%s\n", temp);
```

```
        a
        a/b
        a/b/c
```

Part 1 – Behind the Scenes

Typedefs:

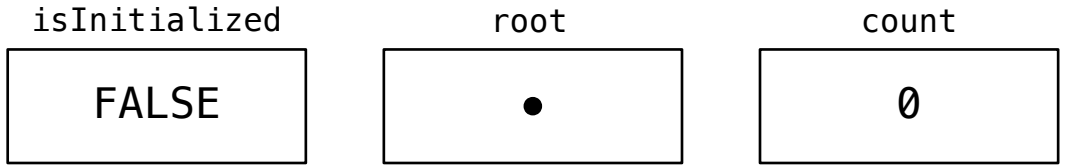
```
/* Return statuses */
enum { SUCCESS,
       INITIALIZATION_ERROR, PARENT_CHILD_ERROR , ALREADY_IN_TREE,
       NO_SUCH_PATH, CONFLICTING_PATH, MEMORY_ERROR
};

/* In lieu of a proper boolean datatype */
typedef enum bool { FALSE, TRUE } boolean;
```

Part 1 – Behind the Scenes

BDT Abstract Object static variables:

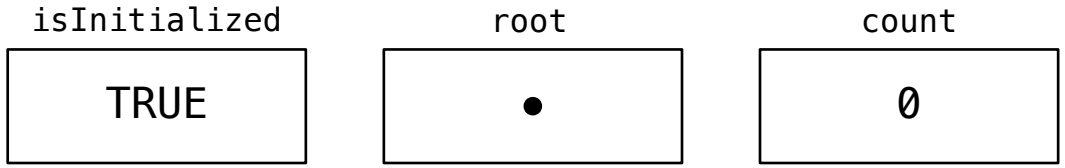
```
/* a flag for if it is in an initialized state (TRUE) or not (FALSE) */
static boolean isInitialized;

/* a pointer to the root Node in the hierarchy */
static Node_T root;

/* a counter of the number of Nodes in the hierarchy */
static size_t count;
```

Part 1 – Behind the Scenes



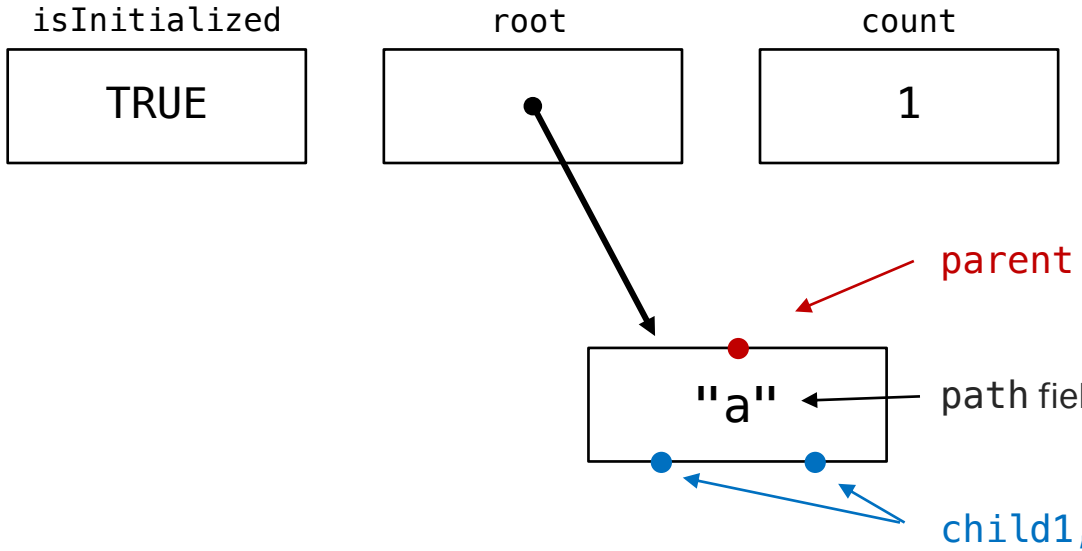
How do we know that these are the initial values, given that we did not initialize them explicitly? (Hint: what section of memory are they in?)

Part 1 – Behind the Scenes



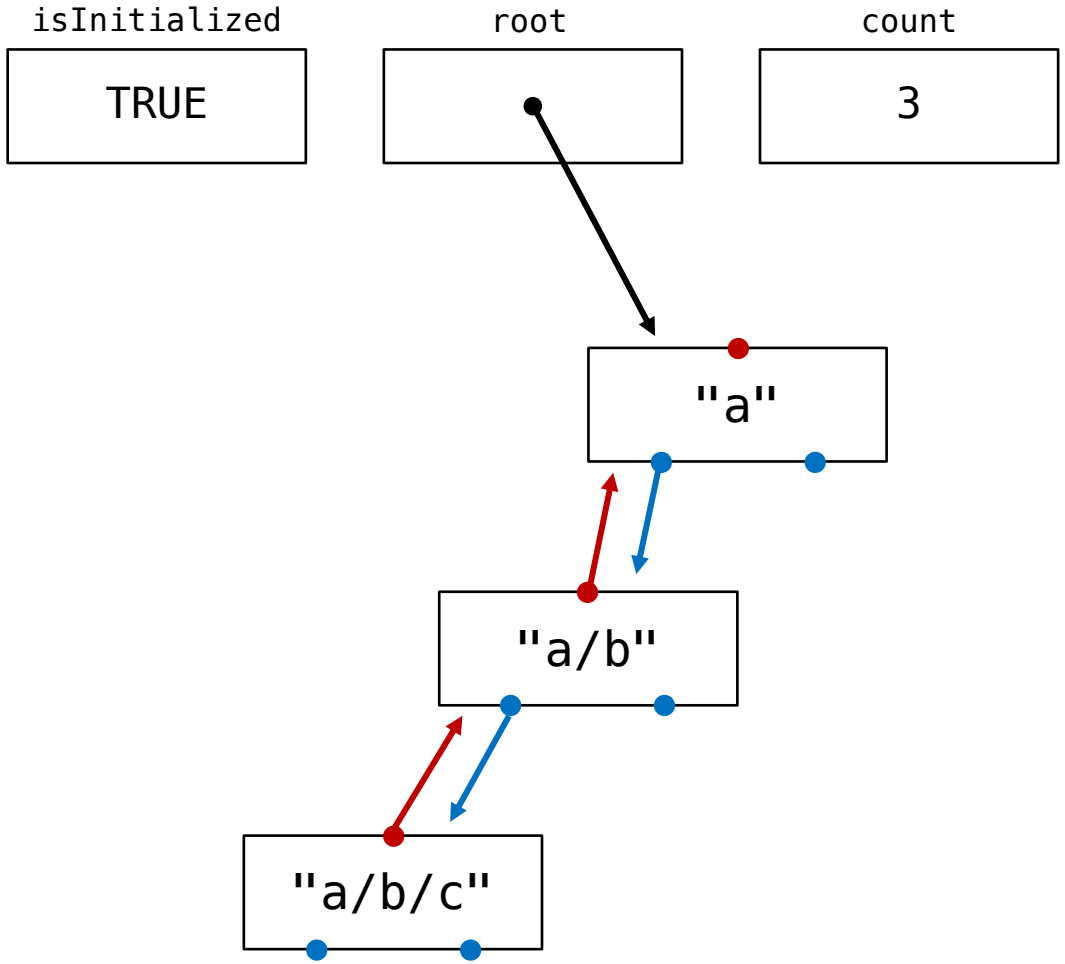

```
BDT_init();
```

Part 1 – Behind the Scenes



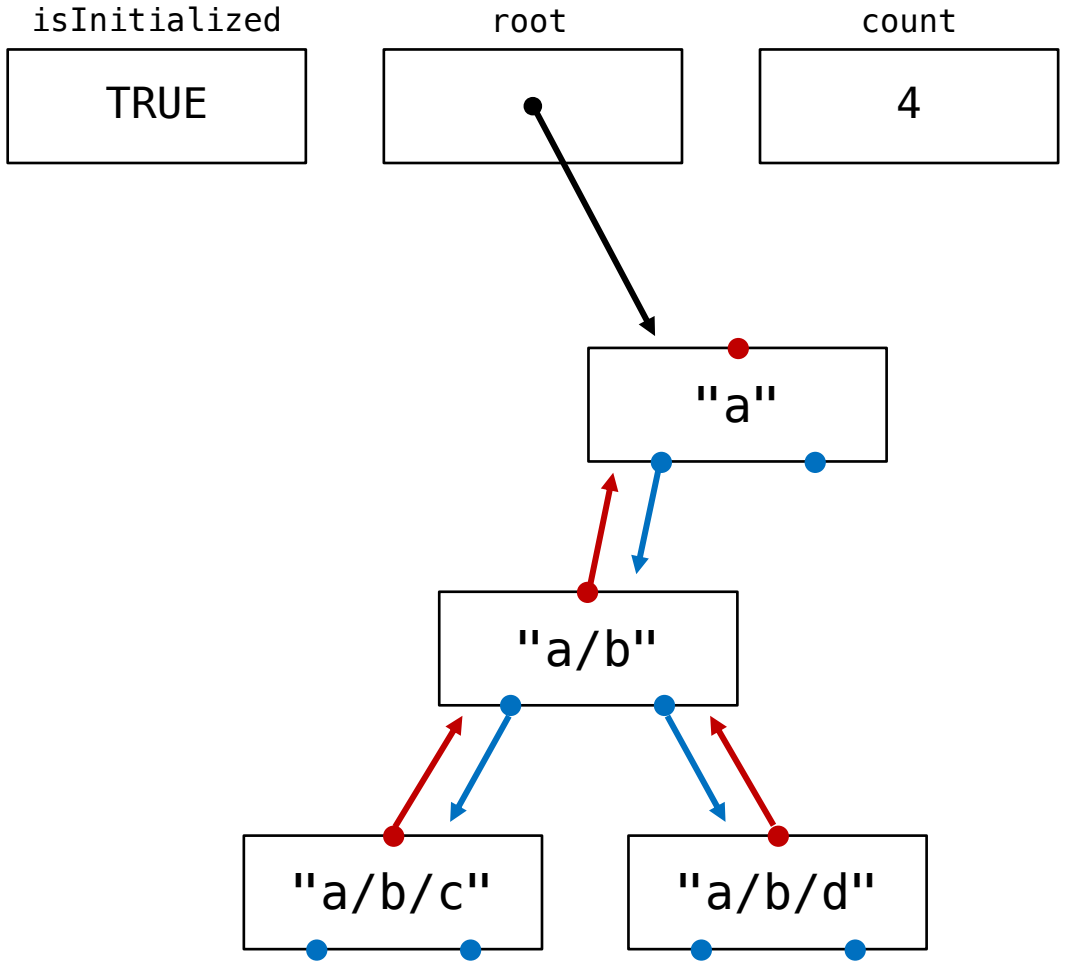

```
BDT_init();
BDT_insertPath("a");
```

Part 1 – Behind the Scenes



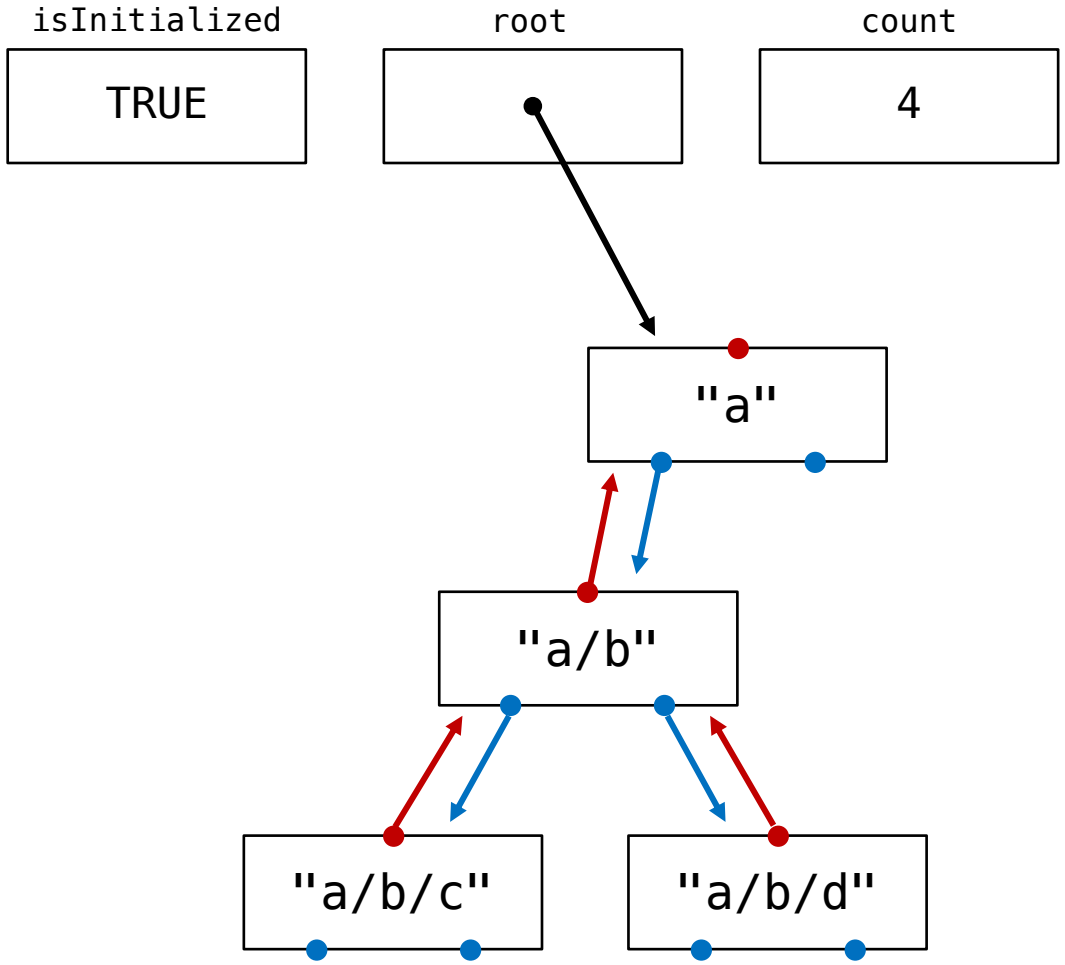

```
BDT_init();
BDT_insertPath("a");
BDT_insertPath("a/b/c");
```

Part 1 – Behind the Scenes



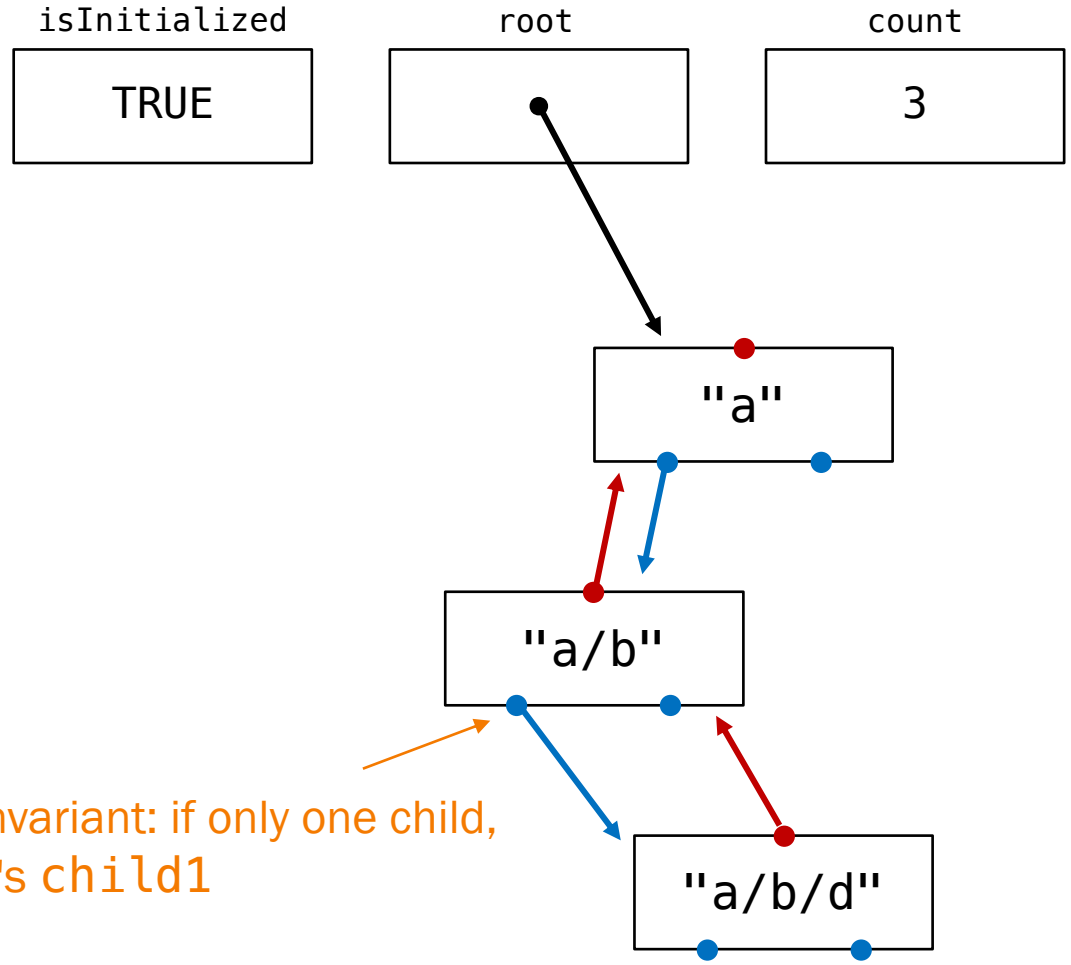

```
BDT_init();
BDT_insertPath("a");
BDT_insertPath("a/b/c");
BDT_insertPath("a/b/d");
```

Part 1 – Returning Errors




```
assert(BDT_insertPath("a/b/c")
       == ALREADY_IN_TREE);

assert(BDT_insertPath("d/e/f")
       == CONFLICTING_PATH);

assert(BDT_insertPath("a/b/e")
       == PARENT_CHILD_ERROR);
```

Part 1 – Special Case




```
BDT_init();
BDT_insertPath("a");
BDT_insertPath("a/b/c");
BDT_insertPath("a/b/d");
BDT_rmPath("a/b/c");
```

Part 1 – What to Do

Great! So… we need to implement the bdt.h API. No problem.

Nope – we've done that for you!

```
$ make
gcc217 -g dynarray.o bdtGood.o bdt_client.o -o bdtGood
gcc217 -g dynarray.o bdtBad1.o bdt_client.o -o bdtBad1
gcc217 -g dynarray.o bdtBad2.o bdt_client.o -o bdtBad2
gcc217 -g dynarray.o bdtBad3.o bdt_client.o -o bdtBad3
gcc217 -g dynarray.o bdtBad4.o bdt_client.o -o bdtBad4
gcc217 -g dynarray.o bdtBad5.o bdt_client.o -o bdtBad5
```

Part 1 – What to Do

Great! So… we need to implement the bdt.h API. No problem.

Nope – we've done that for you!

```
$ ./bdtGood
a
a/b
a/b/c

a
a/b
a/b/c
a/b/d
...
```

Part 1 – What to Do

Star Wars: Episode II (2002)

OK, so what's there for us to do?

```
$ ./bdtBad1
bdtBad1: bdt_client.c:22: main:
  Assertion `BDT_insertPath("a/b/c") == INITIALIZATION_ERROR' failed.
Aborted
```

Part 1 – What to Do

Ah. OK, no problem. Let's just take a look at bdtBad1.c and ...

```
$ cat bdtBad1.c
cat: bdtBad1.c: No such file or directory
```

```
$ ls
Makefile   bdtBad2    bdtBad4    bdtGood       dynarray.c
bdt.h      bdtBad2.o  bdtBad4.o  bdtGood.o     dynarray.h
bdtBad1    bdtBad3    bdtBad5    bdt_client.c  dynarray.o
bdtBad1.o  bdtBad3.o  bdtBad5.o  bdt_client.o
```

Part 1 – What to Do

Wait, you mean we don't get to see the source? That's cruel…

I didn't say that.

So then what do you expect us to do?

```
$ gdb bdtBad1
```

Or, more likely,
run gdb from within emacs

… and the fun begins!

Part 1 – What to Do

What you must do: debug.

- You do not have to identify the bug itself, only its location.
- Simply the function is fine; no need to be more granular.
- But, this must be the location of the underlying error, which is not necessarily where the error manifests itself or is "noticed" by the client.

PART 2

Part 2 Simplifications

Simplification: no files – everything's a directory.

But now, trees of *arbitrary* branching factor are allowed.

So now we have ~~Binary~~ Directory Trees (DTs).

Part 2 – Behind the Scenes

New / restructured code: `a4def.h`, `node.h`, `dynarray.h` and `dynarray.c`

Node definition:

```
typedef struct node* Node_T;
struct node {
   /* the full path of this directory */
   char* path;

   /* the parent directory of this directory, NULL for the root of the directory tree */
   Node_T parent;

   /* the subdirectories of this directory stored in sorted order by pathname */
   DynArray_T children;
};
```

Part 2 – Behind the Scenes

DynArrays implement dynamically resizable arrays

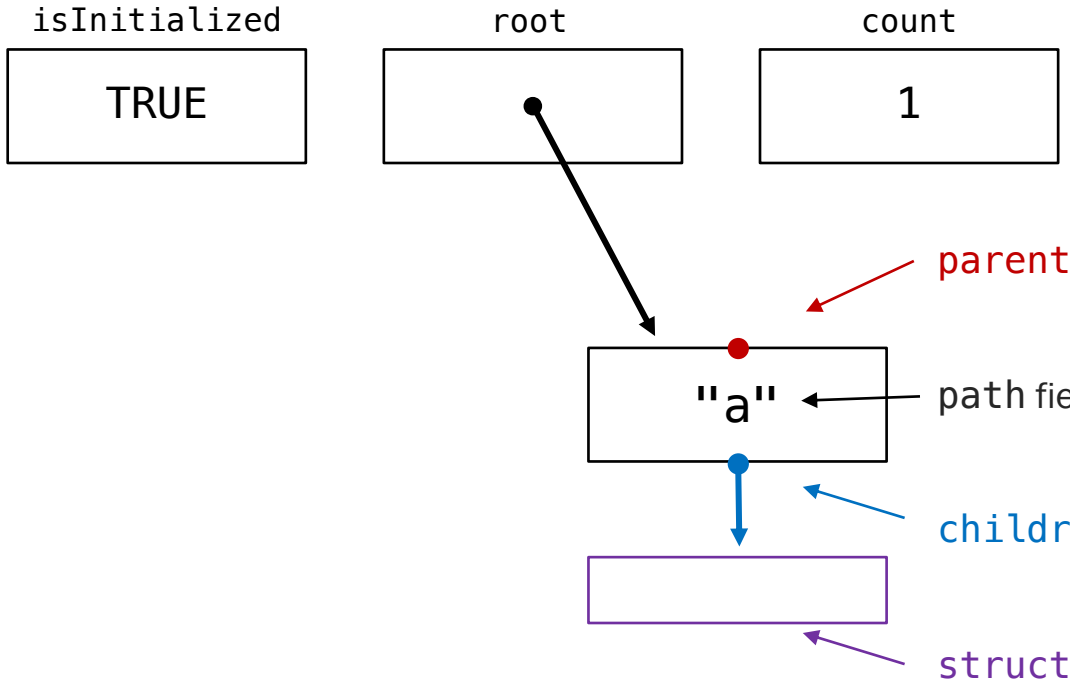
- *We've implemented them for you. Correctly. We think. Aren't we nice?*

DynArray definition:

```
typedef struct DynArray *DynArray_T;
struct DynArray {
   /* The number of elements in the DynArray from the client's point of view. */
   size_t uLength;

   /* The number of elements in the array that underlies the DynArray. */
   size_t uPhysLength;

   /* The array that underlies the DynArray. */
   const void **ppvArray;
};
```

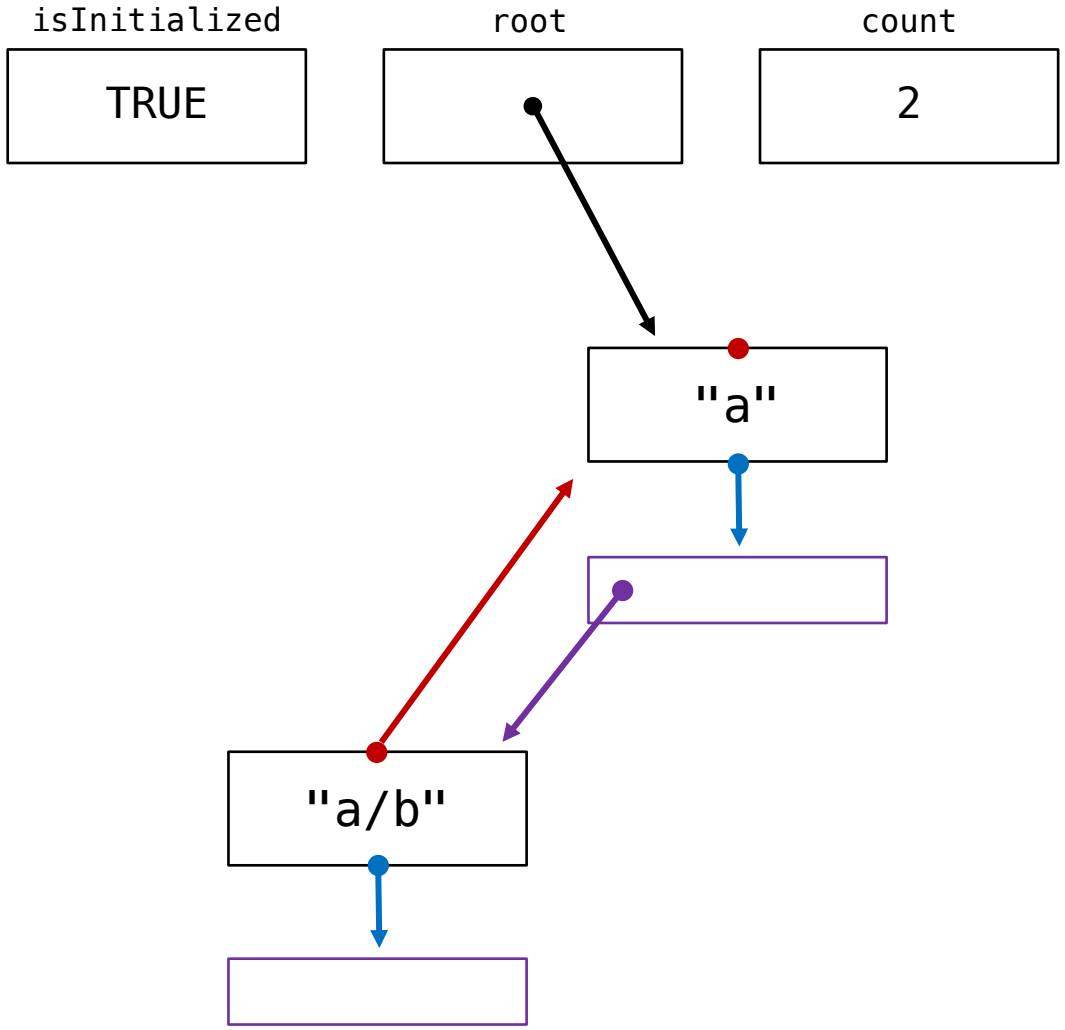
Pointer to array of `void*` – allows resizing

Part 2 – Behind the Scenes



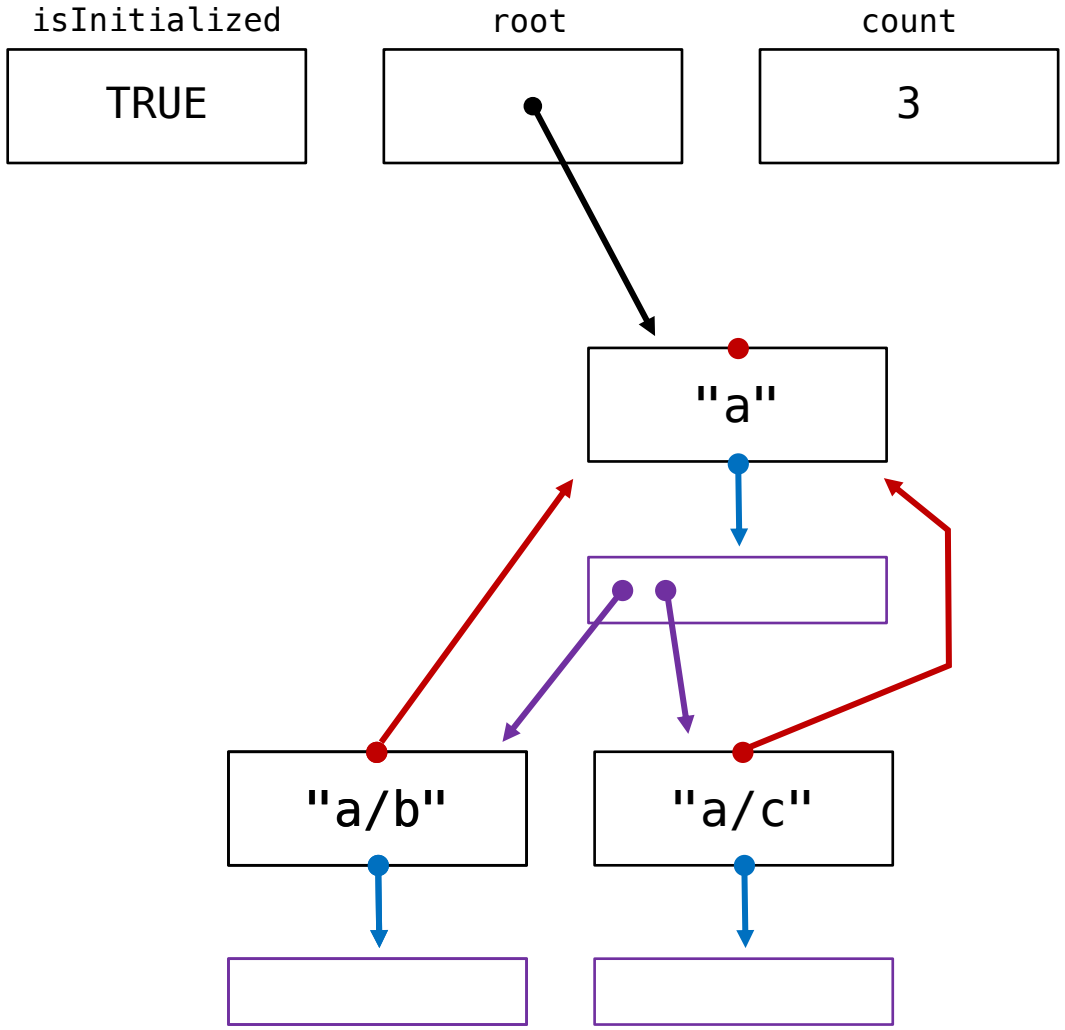

```
DT_init();
DT_insertPath("a");
```

Part 2 – Behind the Scenes



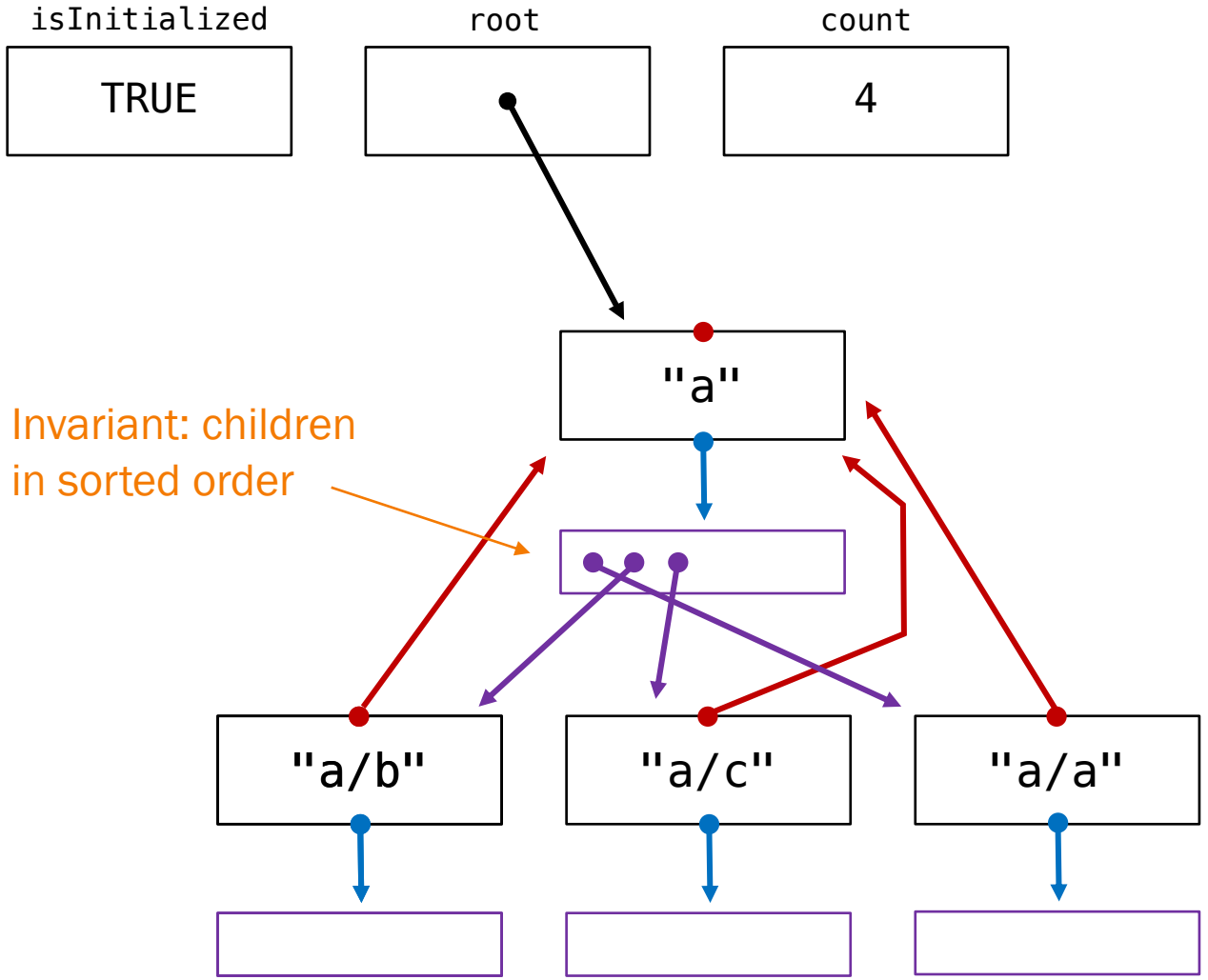

```
DT_init();
DT_insertPath("a");
DT_insertPath("a/b");
```

Part 2 – Behind the Scenes




```
DT_init();
DT_insertPath("a");
DT_insertPath("a/b");
DT_insertPath("a/c");
```

Part 2 – Special Case




```
DT_init();
DT_insertPath("a");
DT_insertPath("a/b");
DT_insertPath("a/c");
DT_insertPath("a/a");
```

Part 2 – What to Do

*Great! So ***now*** do we get to implement the dt.h API?*

Nope – we've done that for you!

```
$ make
gcc217 -g -c dynarray.c
gcc217 -g -c nodeGood.c
gcc217 -g -c checkerDT.c
gcc217 -g -c dtGood.c
gcc217 -g -c dt_client.c
gcc217 -g dynarray.o nodeGood.o checkerDT.o dtGood.o dt_client.o -o dtGood
gcc217 -g dynarray.o nodeBad1a.o checkerDT.o dtBad1a.o dt_client.o -o dtBad1a
gcc217 -g dynarray.o nodeBad1b.o checkerDT.o dtBad1b.o dt_client.o -o dtBad1b
gcc217 -g dynarray.o nodeBad2.o checkerDT.o dtBad2.o dt_client.o -o dtBad2
gcc217 -g dynarray.o nodeBad3.o checkerDT.o dtBad3.o dt_client.o -o dtBad3
gcc217 -g dynarray.o nodeBad4.o checkerDT.o dtBad4.o dt_client.o -o dtBad4
gcc217 -g dynarray.o nodeBad5.o checkerDT.o dtBad5.o dt_client.o -o dtBad5
```

Part 2 – What to Do

And there are still broken implementations!

```
$ ./dtBad2
Segmentation fault
```

Part 2 – What to Do

Ah. Sigh. We'll just fire up gdb and …

```
$ gdb dtBad2
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
(gdb) where
#0  0x0000ffffbe69675c in strlen () from /lib64/libc.so.6
#1  0x0000000000402bdc in CheckerDT_Node_isValid (n=0x4300b0) at checkerDT.c:33
#2  0x0000000000401f10 in Node_create ()
#3  0x00000000004030b4 in DT_insertRestOfPath ()
#4  0x00000000004032b0 in DT_insertPath ()
#5  0x0000000000403d40 in main () at dt_client.c:41
```

Part 2 – What to Do

```
#2  0x0000000000401f10 in Node_create ()
#3  0x00000000004030b4 in DT_insertRestOfPath ()
#4  0x00000000004032b0 in DT_insertPath ()
```

Ummm... why don't we see information about these functions?

We didn't compile with "-g" to include debugging info.

Wait, you mean we don't get to see the source? That's cruel...

Sorry.

So then what do you expect us to do?

Part 2 – What to Do

What you must do: write a checker for the datastructure(s).

- Each function calls `CheckerDT_isValid` at entry and before returning.
- `checkerDT.c` has the beginnings of an implementation for you to fill in, including a full tree traversal.
- When your checker is good enough, it will identify even non-client-visible bugs:

Redirect stderr, since that's where `dt_client` prints

```
$ ./dtGood 2> good_result
$ ./dtBad4 2> bad4_result
$ diff good_result bad4_result
$
```

No output from `diff` means that files are identical (but, as suggested by the name, `dtBad4` is indeed buggy)

Part 2 – Step 2.5

Now examine our allegedly-good implementation in `dtGood.c` and `nodeGood.c` and figure out how a real COS 217 student would write it. Write a critique.

- Pay special attention to the principles from the modularity lecture.
- Are the interfaces what you need?
- Do you see ways to make the implementation better?
  Less complex? More efficient? Clearer?
  More extensible? (Hint, hint.)

PART 3

Part 3 Simplifications

Simplification: none.

Trees can now contain both directories and files.

- Files can't have children, but do have *contents* – a sequence of bytes of any size.

So now we have ~~Directory~~ File Trees (FTs).

Part 3 API

Summary of API in `ft.h` (but read it yourself for details, including error handling!)

- These functions are similar to what we had before:

`int FT_init(void);`	Sets the data structure to initialized status.
`int FT_destroy(void);`	Removes all contents and returns data structure to uninitialized status.
`int FT_insertDir(char* path);`	Inserts a new directory into the tree at path, if possible.
`boolean FT_containsDir(char* path);`	Does the tree contain the full path parameter as a directory?
`int FT_rmDir(char* path);`	Removes the FT hierarchy rooted at the directory path.
`char* FT_toString(void);`	Returns a string representation of the data structure.

Part 3 API (cont.)

Summary of API in `ft.h` (but read it yourself for details, including error handling!)

- And these functions are new-ish:

`int FT_insertFile(char *path, void *contents, size_t length);`	Inserts a new file at the given path, with the given contents.
`boolean FT_containsFile(char *path);`	Does the tree contain the full path parameter as a file?
`int FT_rmFile(char *path);`	Removes the file at path.
`int FT_stat(char *path, boolean* type, size_t* length);`	Does path exist in the hierarchy? If so, fill in type and length (if file).
`void *FT_getFileContents(char *path);`	Returns the contents of the file at the full path parameter.
`void *FT_replaceFileContents(char *path, void *newContents, size_t newLength);`	Replaces current contents of the file with newContents.

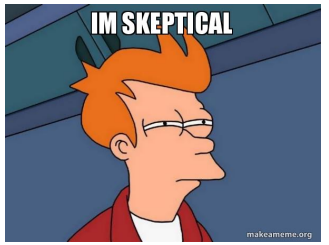
Part 3 – What to Do

OK, so what broken implementations have you got for us this time?

Well, you see, there's this nice blank editor…

Great! We'll just hack up dtGood.c from part 2…

Great! We'll just apply most of the style deductions in codePost (in the event you get it to work…)



So then what do you expect us to do?

Part 3 – What to Do

What you must do: design and write high-quality code for the interface in ft.h

- Think before you code
- Learn from the lessons in part 2.5
- Design appropriate interfaces you'll need
- Compose a Makefile
- Write supporting modules
- Implement the FT interface
  - Possibly borrowing ideas from dtGood.c
- Test your FT implementation
  - Definitely using ft_client.c
  - Possibly adding more tests that you think up (which you can verify against our sampleft.o)
  - Probably using ideas from the checker you wrote in part 2
- Critique your FT implementation

Partnered Assignment

For this assignment, you should partner with one (1) partner.

- Solo is grudgingly acceptable, but discouraged.
- Work together, mostly at the same time. We won't be quite as strict as COS 126, but it's not OK for you to each do half the work, and then cat it together.
- You may work with anyone in the class – not just from your precept.
- To find a partner, hang out after precept / lecture, or post on Ed, etc.