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 10-12)
   - 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 10-12)
2. Practice debugging (lecture 12, precepts 5 and 9)
   - Especially using `gdb` and `meminfo`

Assignment 4 Goals

1. Gain more familiarity with data structures (lecture 10, precepts 10-12)
2. Practice debugging (lecture 12, precepts 5 and 9)
3. Take responsibility for your own testing (lectures 9 and 13)
   - Some of the testing cases/code will not be written for you (eep!)
   - You will write a "checker" that verifies an AO's internal state to make sure it's sound

Assignment 4 Goals

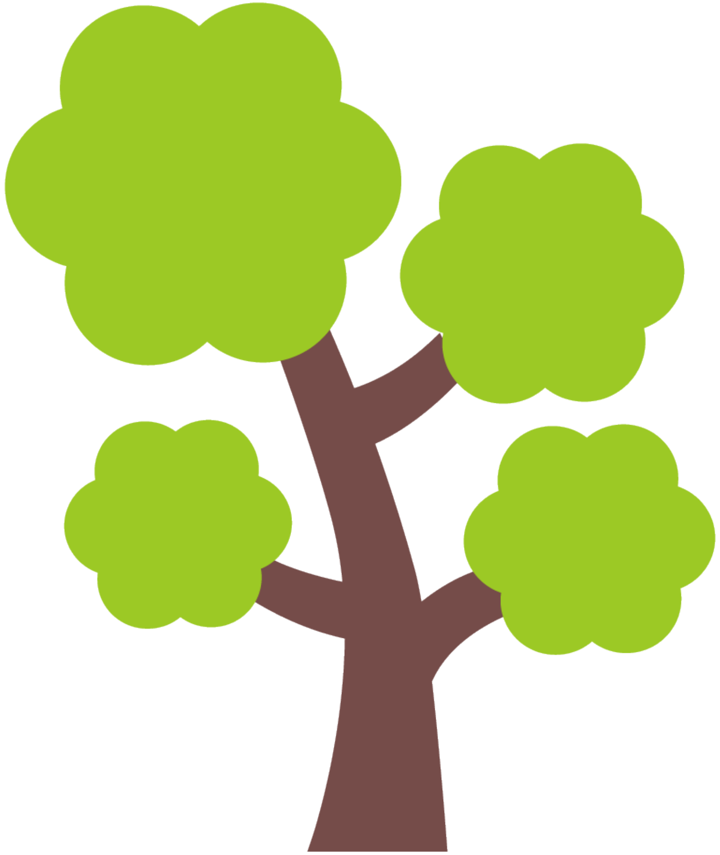
1. Gain more familiarity with data structures (lecture 10, precepts 10-12)
2. Practice debugging (lecture 12, precepts 5 and 9)
3. Take responsibility for your own testing (lectures 9 and 13)
4. Design your own modules and interfaces (lecture 13)
   - We will give you a high-level interface and client 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 10-12)
2. Practice debugging (lecture 12, precepts 5 and 9)
3. Take responsibility for your own testing (lectures 9 and 13)
4. Design your own modules and interfaces (lecture 13)
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 FAANG from scratch on day 1

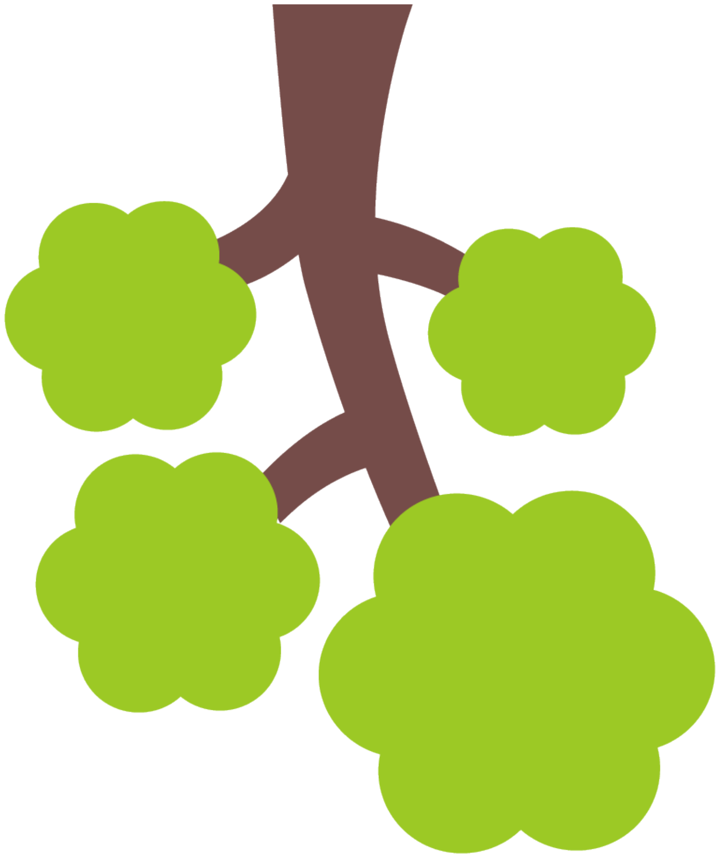
The Sting (1973)

Trees

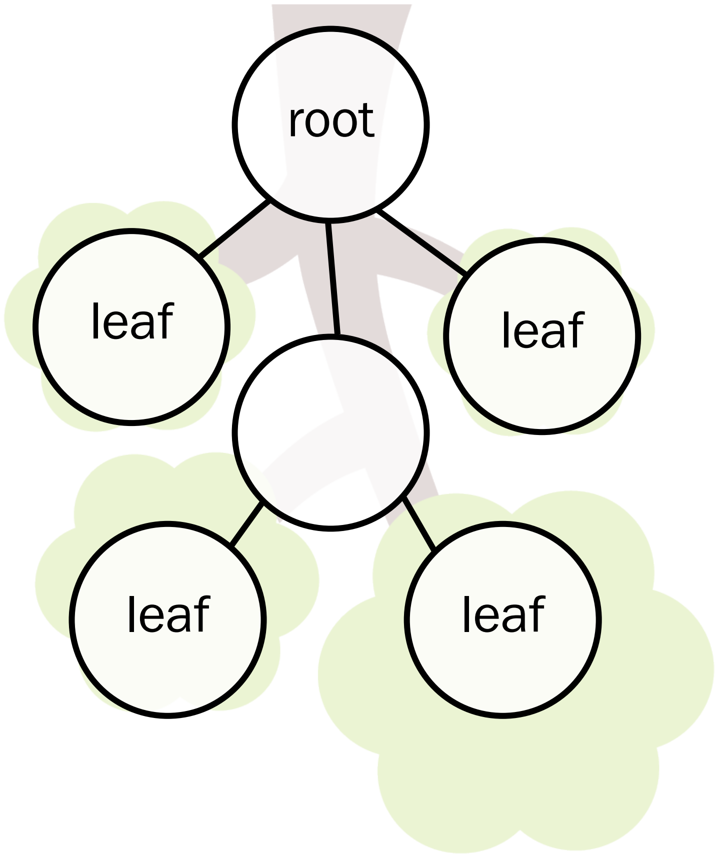


stockio.com

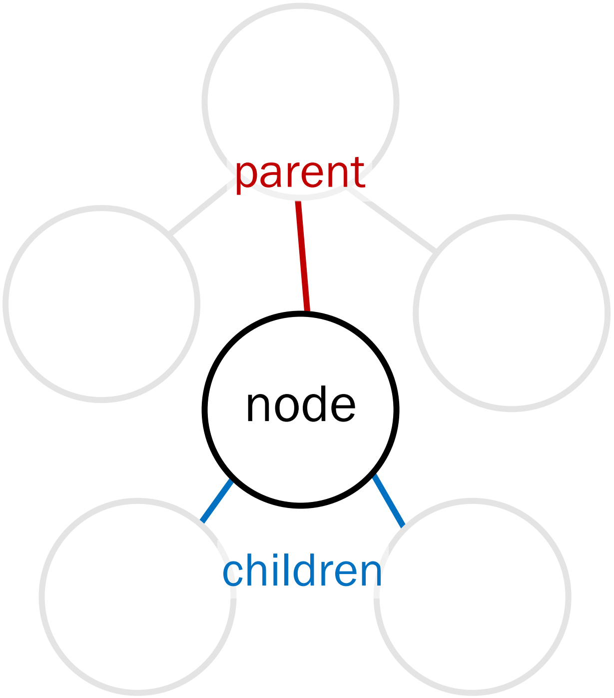
Trees (as seen by computer scientists)



Trees (as implemented by computer scientists)



Trees (as implemented by computer scientists)

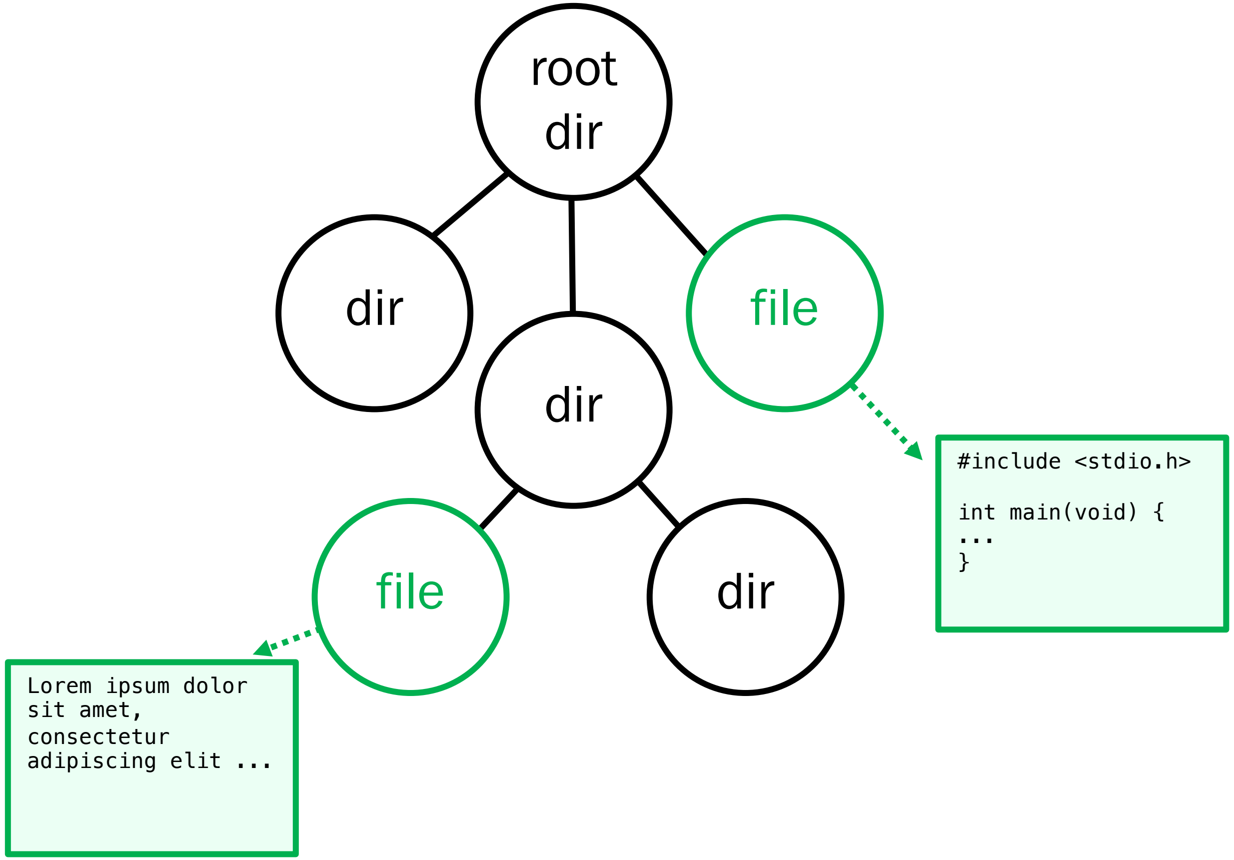


Trees and Filesystems

- Trees encode hierarchical relationships
- So do filesystems
  - A directory can hold files or other directories ("folders", if you must ...)
  - All directories and files are reachable from the root

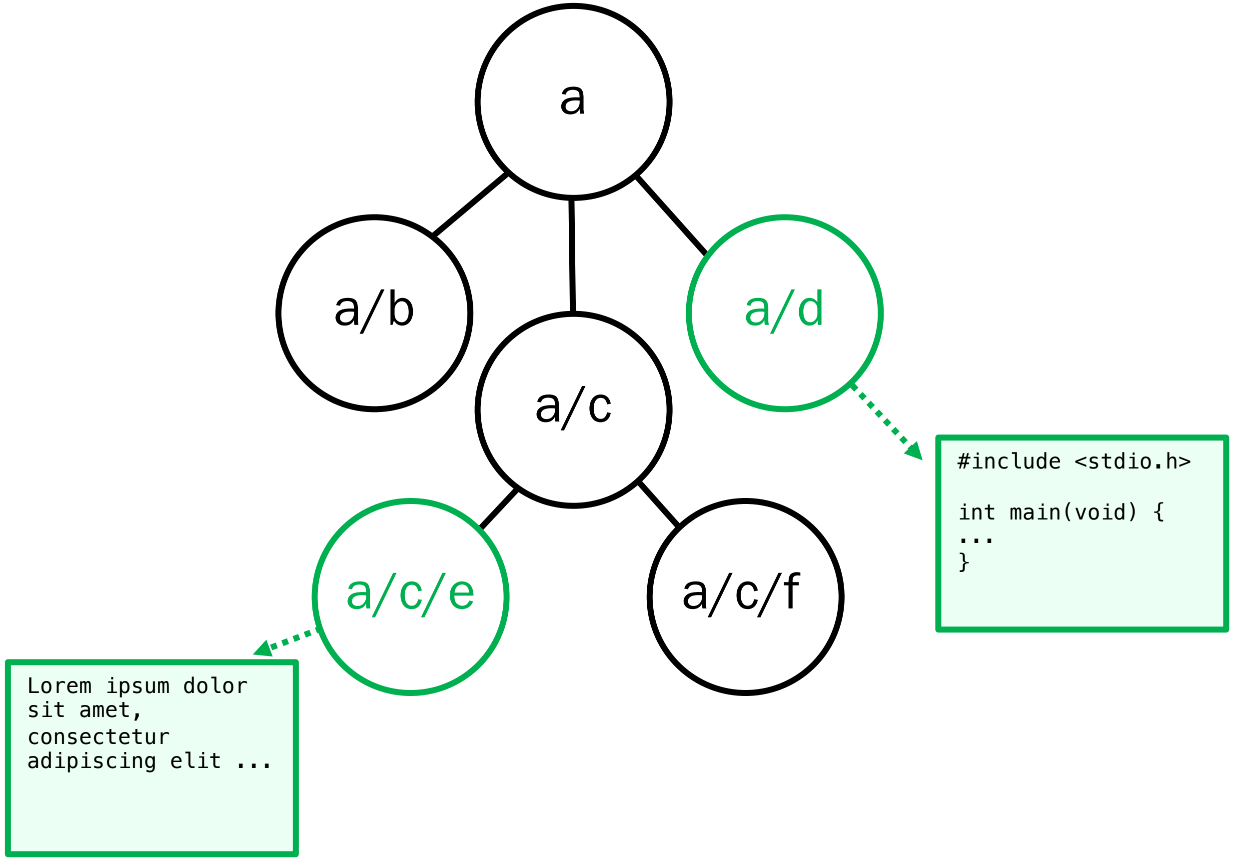
Filesystems as Trees

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



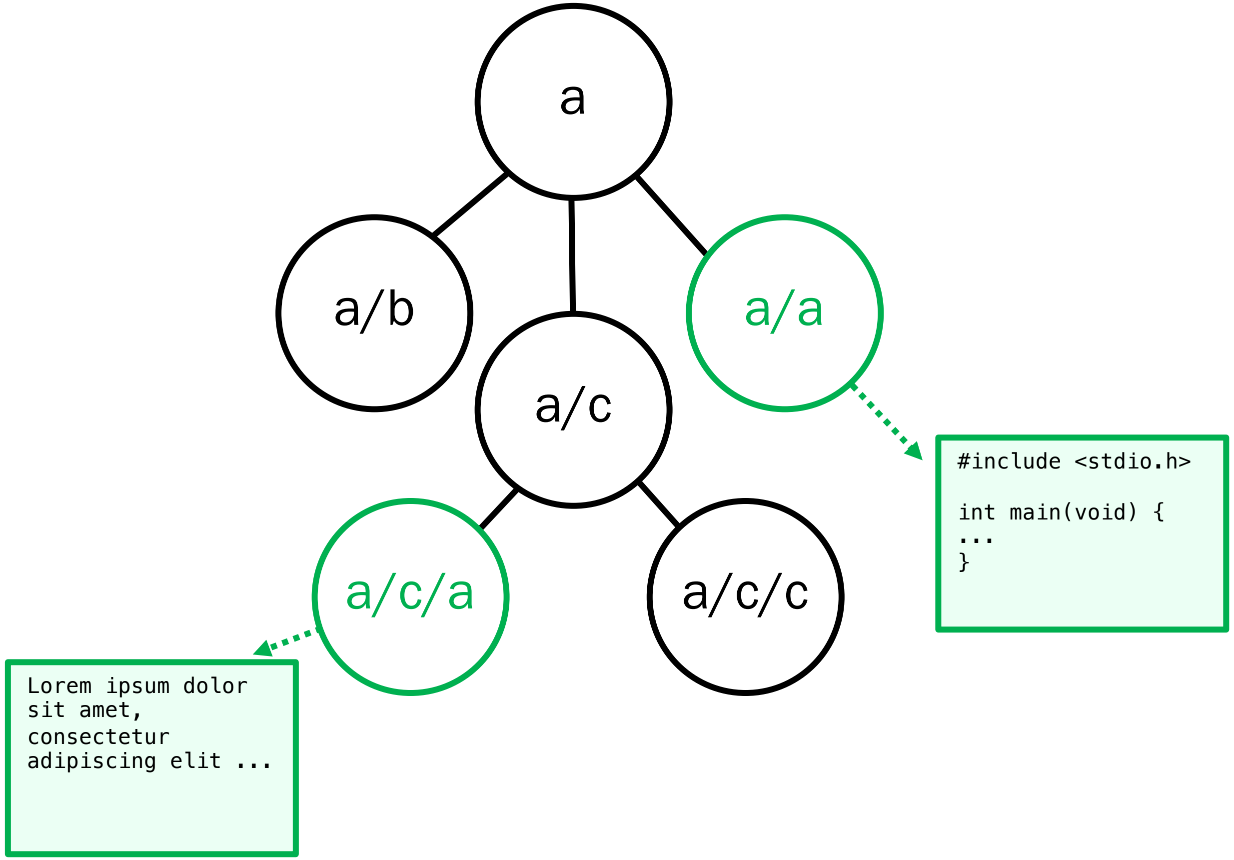
Filesystems as Trees

- 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 not-so-great implementations
- You have access to the API, and client code
- You do not have access to all 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_insert(const char* pcPath);`	Inserts a new path into the tree, if possible. (Like mkdir -p)
`boolean BDT_contains(const char* pcPath);`	Returns TRUE if the tree contains a Node with absolute path pcPath?
`int BDT_rm(const char* pcPath);`	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 look at some (renamed) excerpts from `bdt_client.c`

```
assert(BDT_init() == SUCCESS);
assert(BDT_insert("a") == SUCCESS);
assert(BDT_insert("a/b/c") == SUCCESS);
assert(BDT_contains ("a") == TRUE);
assert(BDT_contains ("a/b") == TRUE);
assert(BDT_contains ("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: a4def.h Definitions

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


/* In lieu of a proper boolean datatype */
enum bool { FALSE, TRUE };
/* Make enumeration "feel" more like a builtin type */
typedef enum bool boolean;
```

Part 1 – Behind the Scenes: bdt.c Definitions

BDT Abstract Object static variable declarations:

```
/* 1. a flag for being in an initialized state (TRUE) or not (FALSE) */
static boolean bIsInitialized;

/* 2. a pointer to the root node in the hierarchy */
static struct node *psRroot;

/* 3. a counter of the number of nodes in the hierarchy */
static size_t ulCount;
```

Part 1 – Behind the Scenes: Trace (at program start)

bIsInitialized	psRoot	ulCount
FALSE	•	0

How do we know that these are the initial values, given that we did not initialize them explicitly?

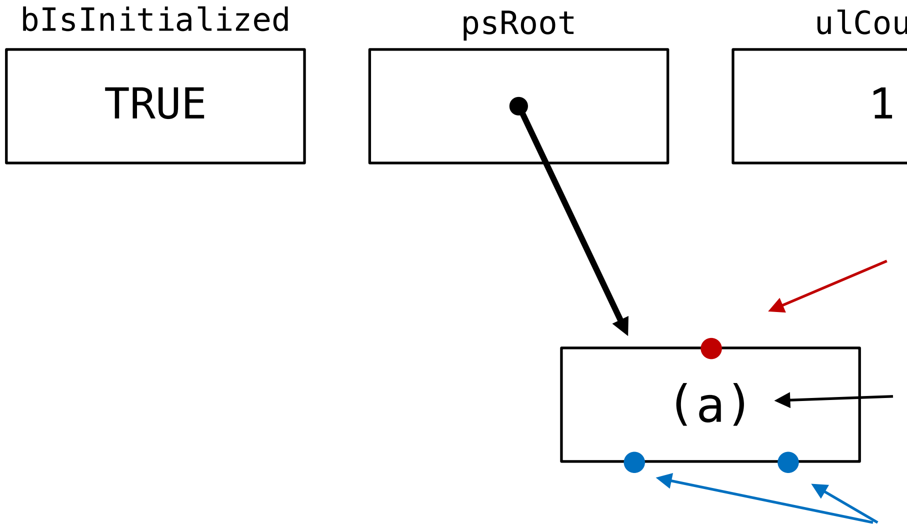
(Hint: what section of memory are they in?)

Part 1 – Behind the Scenes: Trace (after initialization)

bIsInitialized	psRoot	ulCount
TRUE	•	0

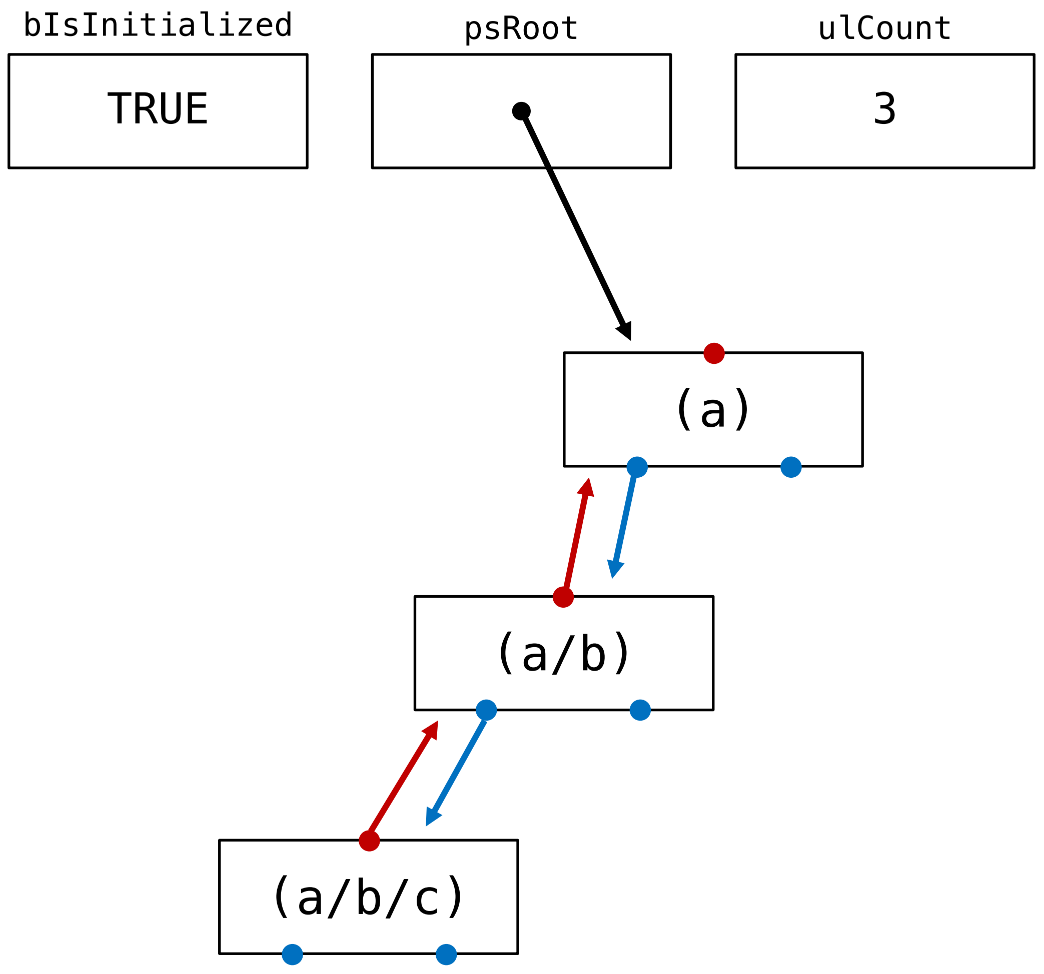
```
BDT_init();
```

Part 1 – Behind the Scenes: Trace (one-node insert)



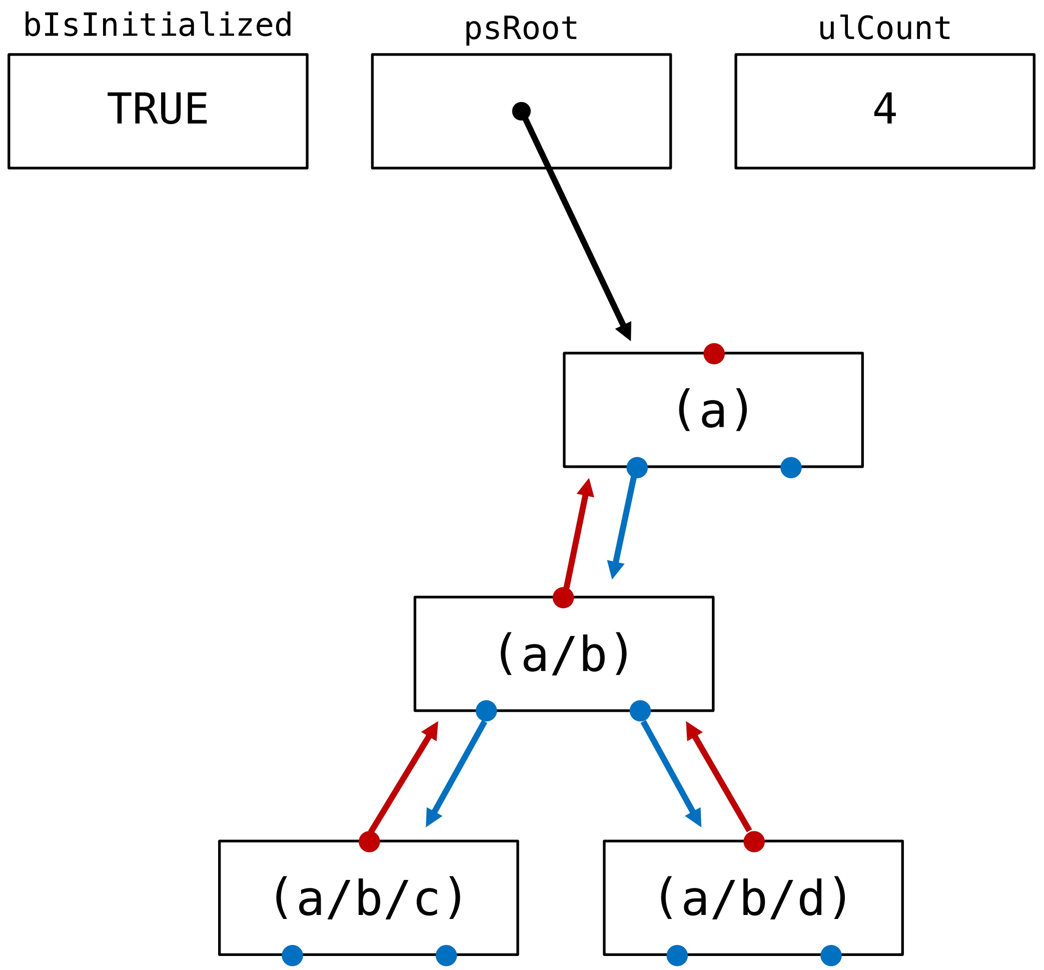

```
BDT_init();
BDT_insert("a");
```

Part 1 – Behind the Scenes: Trace (multi-node insert)



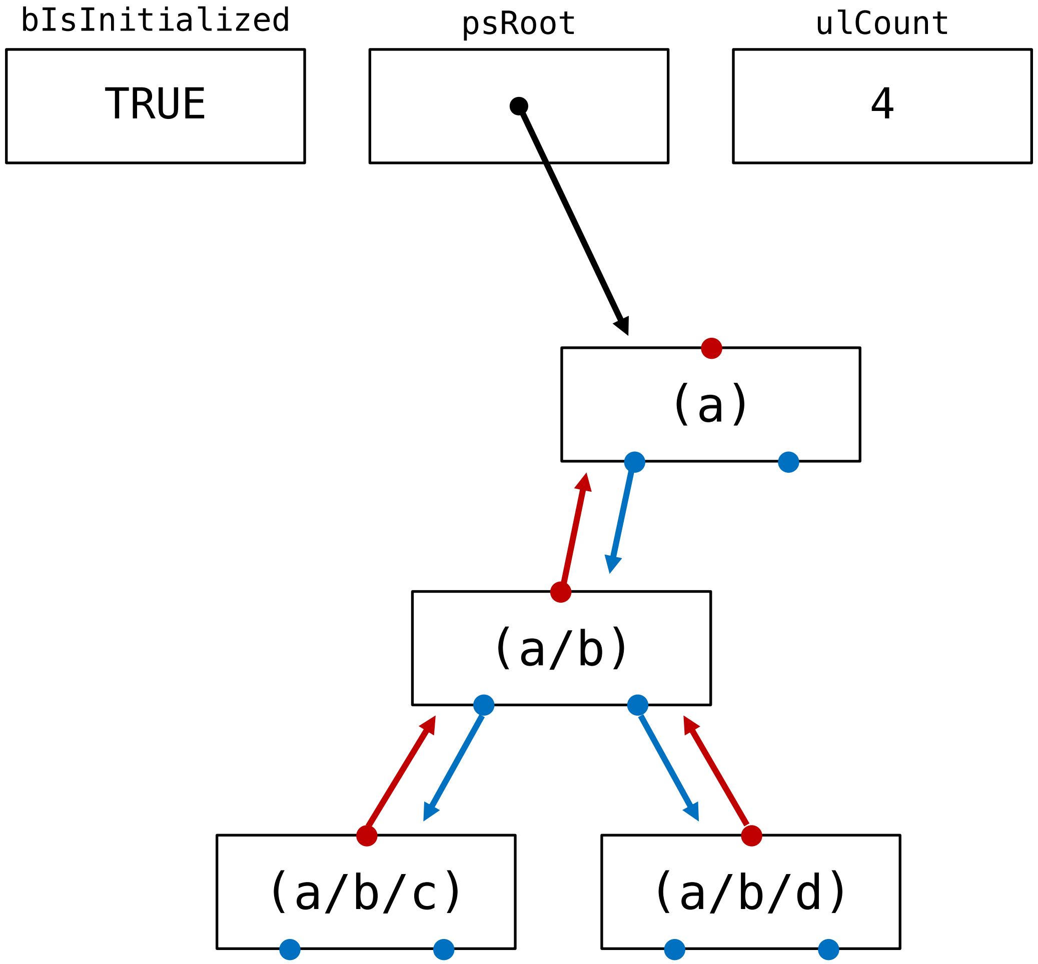

```
BDT_init();
BDT_insert("a");
BDT_insert("a/b/c");
```

Part 1 – Behind the Scenes: Trace (second child)




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

Part 1 – Error Statuses (duplicate, bad root, 3rd child)

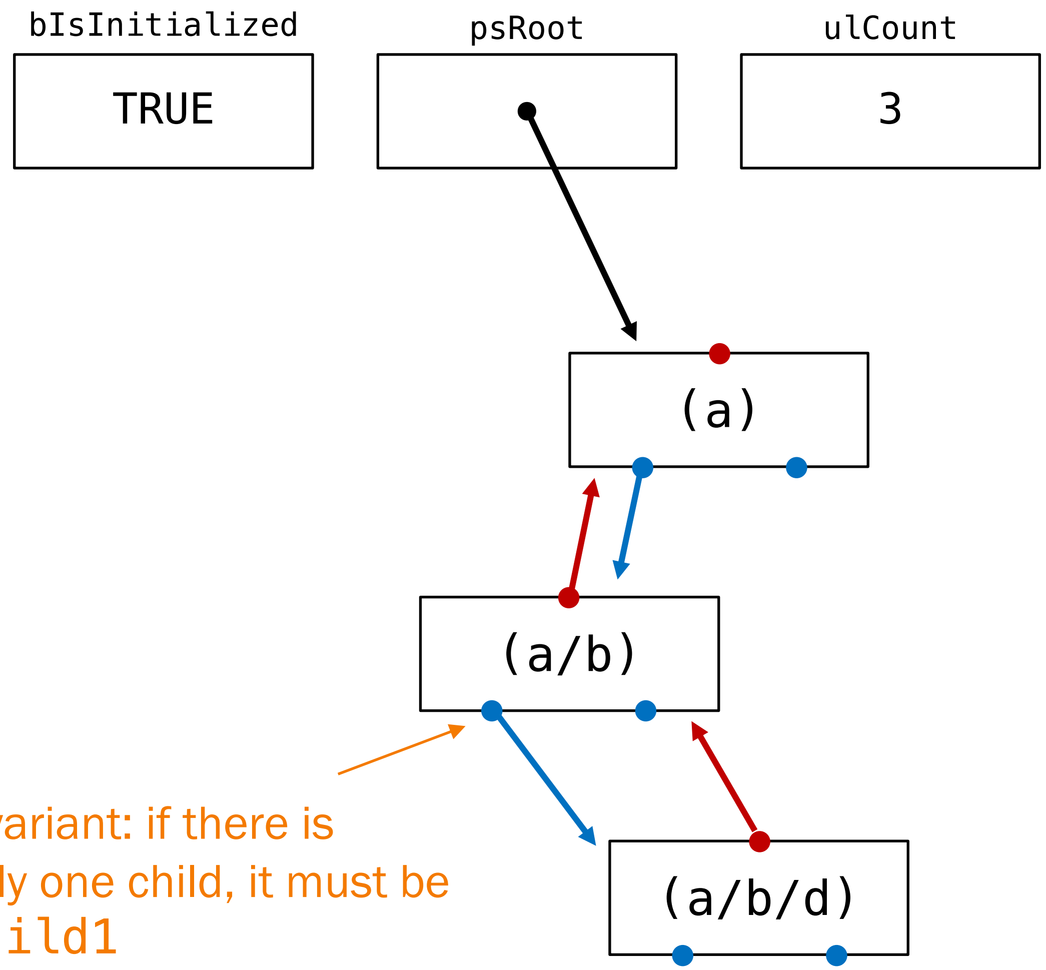



```
assert(BDT_insertPath("a/b/c")
        == ALREADY_IN_TREE);

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

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

Part 1 – Special Case: Promotion



Invariant: if there is only one child, it must be `child1`

```
BDT_init();
BDT_insert("a");
BDT_insert("a/b/c");
BDT_insert("a/b/d");
BDT_rm("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 -c dynarray.c
gcc217 -g -c path.c
gcc217 -g -c bdt_client.c
gcc217 -g dynarray.o path.o bdtGood.o bdt_client.o -o bdtGood
gcc217 -g dynarray.o path.o bdtBad1.o bdt_client.o -o bdtBad1
gcc217 -g dynarray.o path.o bdtBad2.o bdt_client.o -o bdtBad2
gcc217 -g dynarray.o path.o bdtBad3.o bdt_client.o -o bdtBad3
gcc217m -g -c dynarray.c -o dynarrayM.o
gcc217m -g -c path.c -o pathM.o
gcc217m -g -c bdt_client.c -o bdt_clientM.o
gcc217m -g dynarrayM.o pathM.o bdtBad4.o bdt_clientM.o -o bdtBad4
gcc217m -g dynarrayM.o pathM.o bdtBad5.o bdt_clientM.o -o bdtBad5
```

```
$ ./bdtGood
Checkpoint 1:
1root
1root/2child
1root/2child/3grandchild
1root/2second

...
```

Part 1 – What to Do

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

Star Wars: Episode II (2002)

```
$ ./bdtBad1
bdtBad1: bdt_client.c:24: main: Assertion
  `BDT_insert("1root/2child/3grandchild") == INITIALIZATION_ERROR'
  failed.
Aborted (core dumped)
```

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 bdt*
bdt.h      bdtBad1.o  bdtBad2.o      bdtBad3.o
bdtBad4.o  bdtBad5.o  bdtGood.o      bdt_client.o
bdtBad1    bdtBad2    bdtBad3        bdtBad4
bdtBad5    bdtGood    bdt_client.c   bdt_clientM.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 (what function is fine).
- 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: Node_T

New / repurposed code: `nodeDT.h`, `dynarray.h` and `dynarray.c`

Node definition:

```
typedef struct node *Node_T;
struct node {
   /* this directory's absolute path*/
   Path_T oPPath;

   /* this node's parent, NULL for the root of the directory tree */
   Node_T oNParent;

   /* this directory's children (subdirectories) stored in sorted order by pathname */
   DynArray_T oDChildren;
};
```

Part 2 – Behind the Scenes: DynArray_T

DynArrays implement dynamically resizable arrays

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

DynArray definition:

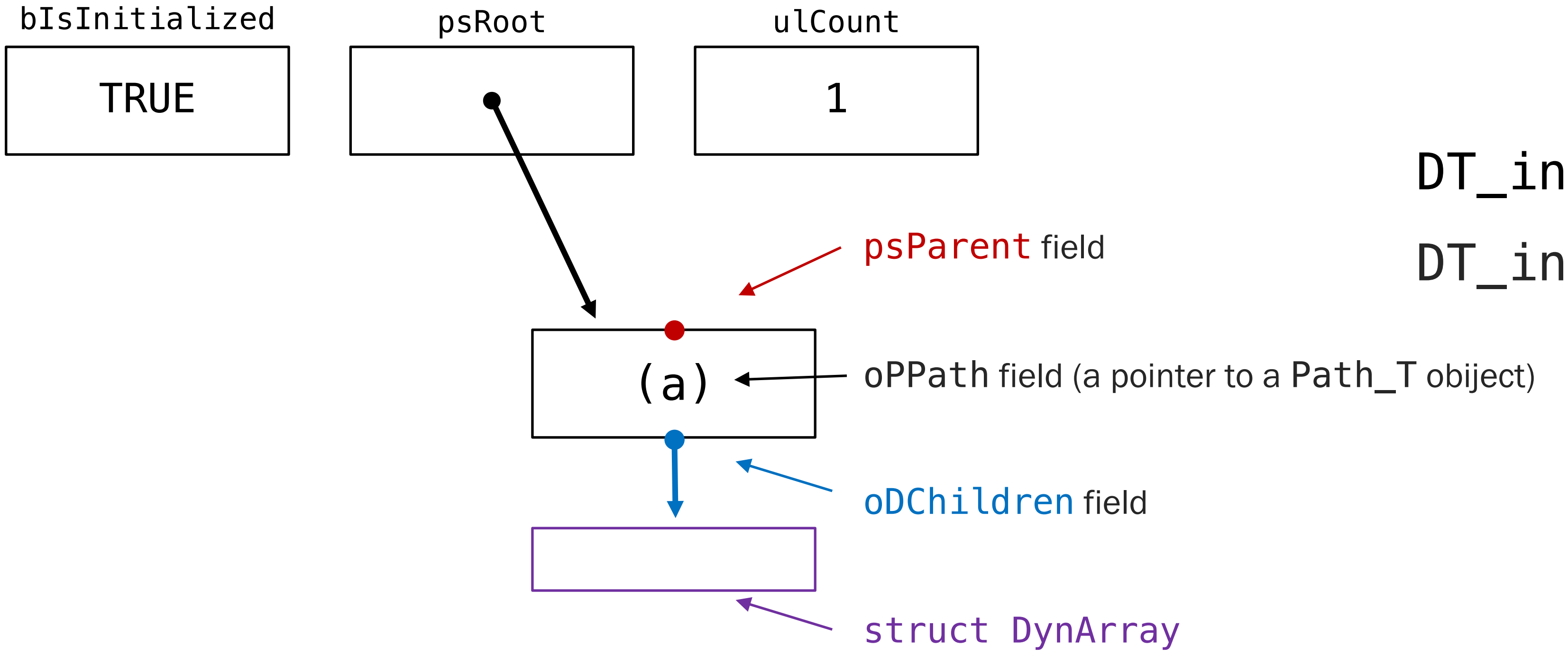
```
typedef struct DynArray *DynArray_T;
struct DynArray {
   /* The number of elements in the DynArray from the client's point of view. */
   size_t uLength;

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

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

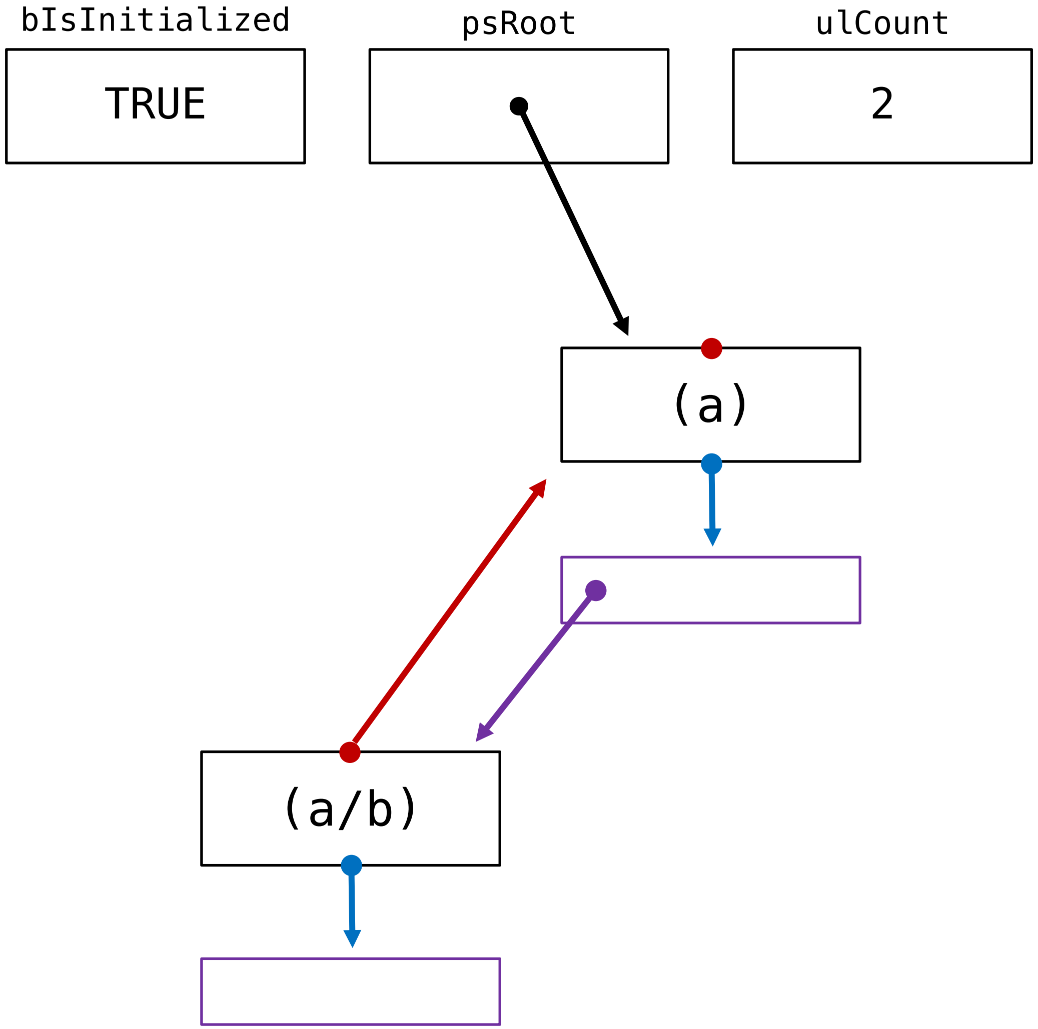
Pointer to array of void*
(allows resizing)

Part 2 – Behind the Scenes: Trace (initialize, insert)



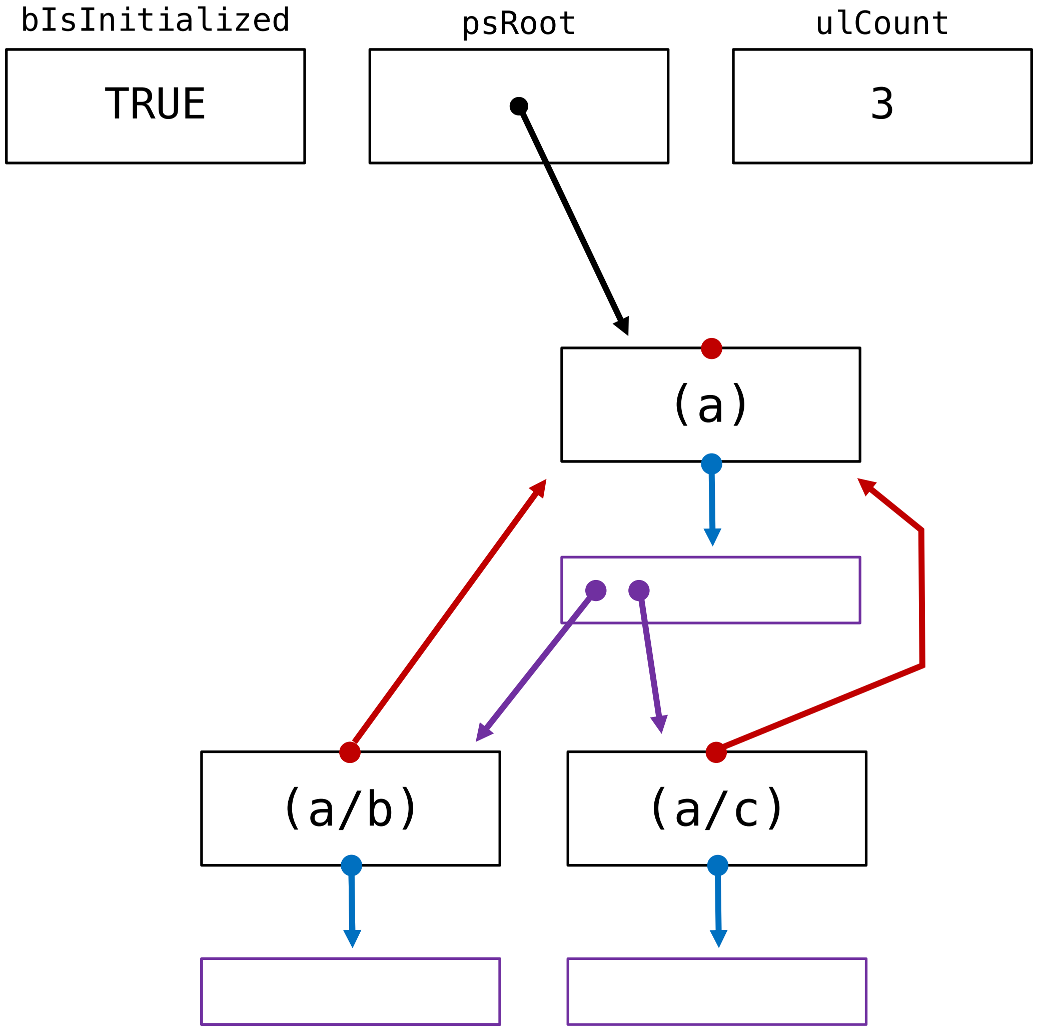

```
DT_init();
DT_insert("a");
```

Part 2 – Behind the Scenes: Trace (insert a child)



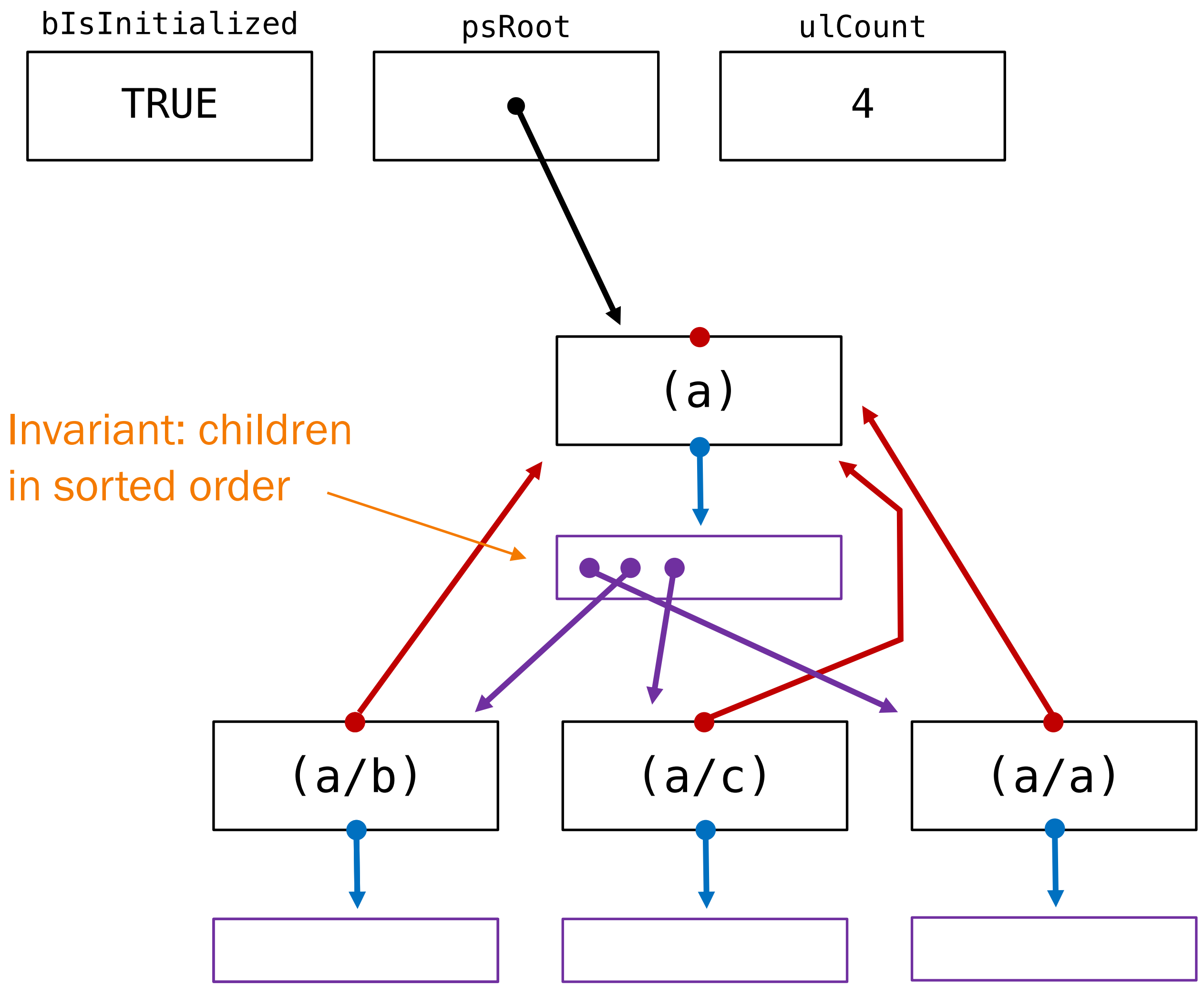

```
DT_init();
DT_insert("a");
DT_insert("a/b");
```

Part 2 – Behind the Scenes: Trace (insert 2nd child)




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

Part 2 – Behind the Scenes: Trace (insert 3rd child)




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

Part 2 – What to Do

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

Nope – we've done that for you! (Again.)

```
$ make
gcc217 -g -c dynarray.c
gcc217 -g -c path.c
gcc217 -g -c checkerDT.c
gcc217 -g -c nodeDTGood.c
gcc217 -g -c dtGood.c
gcc217 -g -c dt_client.c
gcc217 -g dynarray.o path.o checkerDT.o nodeDTGood.o dtGood.o dt_client.o -o dtGood
gcc217 -g dynarray.o path.o checkerDT.o nodeDTBad1a.o dtBad1a.o dt_client.o -o dtBad1a
gcc217 -g dynarray.o path.o checkerDT.o nodeDTBad1b.o dtBad1b.o dt_client.o -o dtBad1b
gcc217 -g dynarray.o path.o checkerDT.o nodeDTBad2.o dtBad2.o dt_client.o -o dtBad2
gcc217 -g dynarray.o path.o checkerDT.o nodeDTBad3.o dtBad3.o dt_client.o -o dtBad3
gcc217 -g dynarray.o path.o checkerDT.o nodeDTBad4.o dtBad4.o dt_client.o -o dtBad4
```

Part 2 – What to Do

And there are still broken implementations!

```
$ ./dtBad2
dtBad2: dt_client.c:67: main: Assertion
  `DT_insert("1root/2child/3grandchild") == ALREADY_IN_TREE' failed.
Aborted (core dumped)
```

Part 2 – What to Do: I did type step, not next, right?

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


```
$ gdb dtBad2
(gdb) b 67
      Breakpoint 1 at 0x4044c4: file dt_client.c, line 67.
(gdb) run
      Breakpoint 1, main () at dt_client.c:67
67    assert(DT_insert("1root/2child/3grandchild") == ALREADY_IN_TREE);
(gdb) step
      dtBad2: dt_client.c:67: main: Assertion
      `DT_insert("1root/2child/3grandchild") == ALREADY_IN_TREE'
      failed.
      Program received signal SIGABRT, Aborted.
```


@dudewheresmycode

Part 2 – What to Do

Ummm... why don't we see info about / why can't we step into 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 data structure(s).

- Each mutator function calls `CheckerDT_isValid` before returning.
- `checkerDT.c` has the beginnings of an implementation for you to fill in, including a full tree traversal and a couple of demonstration check implementations:

```
$ ./dtBad1a

  Not initialized, but count is not 0
  dtBad1a: dtBad1a.c:320: DT_destroy: Assertion
  `CheckerDT_isValid(bIsInitialized, oNRoot, ulCount)' failed.
  Aborted (core dumped)

$ ./dtBad1b

  P-C nodes don't have P-C paths: (1root) (1root/2child/3grandchild)
  dtBad1b: nodeDTBad1b.c:165: Node_new: Assertion
  `CheckerDT_Node_isValid(*poNResult)' failed.
  Aborted (core dumped)
```

Part 2 – Step 2.5

Now examine our allegedly-good implementation in `dtGood.c` and `nodeDTGood.c` and contrast with how an A+ 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(const char* pcPath);`	Inserts a new directory into the tree, if possible. (Like mkdir -p)
`boolean FT_containsDir(const char* pcPath);`	Returns TRUE if the tree contains a directory with absolute path pcPath.
`int FT_rmDir(const char* pcPath);`	Removes the hierarchy rooted at directory pcPath. (Like rm –r)
`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(const char *pcPath, void *pvContents, size_t ulLength);`	Inserts a new file with absolute path pcPath, with the given contents and given length.
`boolean FT_containsFile(const char *pcPath);`	Returns TRUE if the tree contains a file with absolute path pcPath.
`int FT_rmFile(const char *pcPath);`	Removes the file at absolute path pcPath.
`int FT_stat(const char *pcPath, boolean *pbIsFile, size_t *pulSize);`	Does pcPath exist in the hierarchy? If so, pass back whether it's a file & its length, if so.
`void *FT_getFileContents(const char *pcPath);`	Returns the contents of the file at abs. path pcPath.
`void *FT_replaceFileContents(const char *pcPath, void *pvNewContents, size_t ulNewLength);`	Replaces current contents of the file at abs. path pcPath with pvNewContents and returns the old contents.

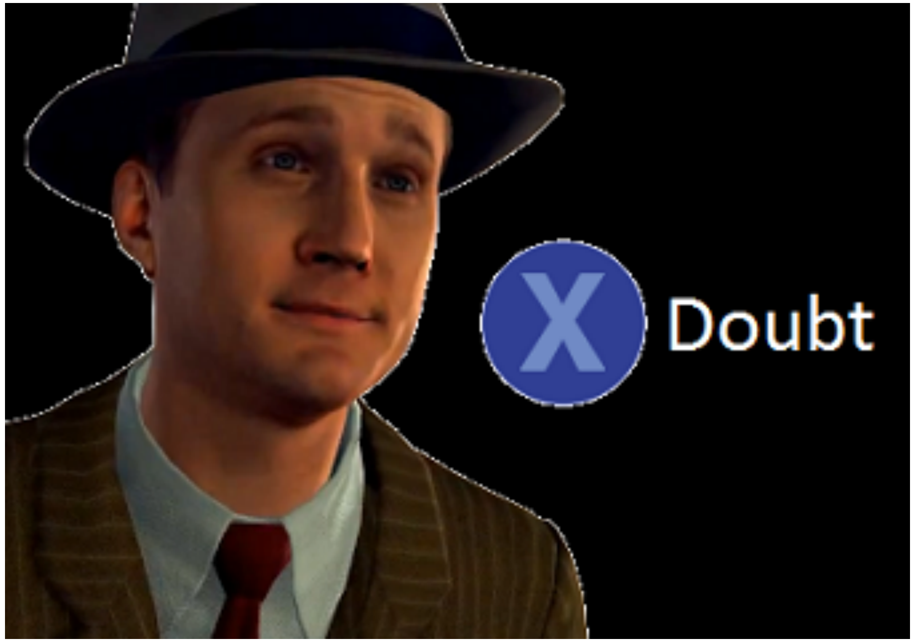
Part 3 – What to Do

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

Good news: no broken code. Bad news: no code at all.

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

Great! We'll sit back and watch while you create an impenetrable web of conflicting dependencies and broken contracts. Good luck spending the next 16 days continuously on gdb.



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 (note, though, you don't have to fix everything!)
- Design appropriate interfaces you'll need
- Compose a Makefile
- Write supporting modules
- Implement the FT interface
    - Likely borrowing ideas / code 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 your checkerDT (note, though, you don't have to write one for FT)
- Critique your FT implementation

Partnered Assignment

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

- Solo efforts are grudgingly acceptable, but ***strongly*** discouraged.
- Work together, mostly at the same time. We aren't 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 own precept.
- To find a partner, hang out after precept / lecture, post on Ed, etc.