CSE 333 – SECTION 2

gdb, valgrind, pointers & structs

Questions, Comments, Concerns

- Do you have any questions?
- Exercises going ok?
- Lectures make sense?
- Homework 1 – START EARLY!

Upcoming Dates:

- No Class on Monday July 5th!
- Due Wednesday 7/7: Exercise 3 @ 11 am PDT
- Due Thursday 7/8: HW1 @ 11:59 pm PDT

Motivation & Tools

- The projects are big, lots of potential for bugs
- **Debugging is a skill that you will need throughout your career**

- gdb (GNU Debugger) is a debugging tool
  - Handles more than just assembly.
  - Lots of helpful features to help with debugging
  - Very useful in tracking undefined behavior

- Valgrind is a memory debugging tool
  - Checks for various memory errors
  - If you are running into odd behavior, running valgrind may point out the cause.

Exercise 1: Debugging with gdb

Segmentation fault

- Causes of segmentation fault
  - Dereferencing uninitialized pointer
  - Null pointer
  - A previously freed pointer
  - Accessing end of an array
  - …

- gdb (GNU Debugger) is very helpful for identifying the source of a segmentation fault
  - backtrace

gdb Breakpoints

- Usage:
  - break <function_name>
  - break <filename:line#>
    - Example: break CSE333.c:20
      // ^ sets breakpoint for when Verify333 fails
- Can advance with:
  - continue
  - next
  - step
  - finish
- More info linked from the course website!

Other Essential gdb Commands

- run <command_line_args>
- backtrace
- frame, up, down
- print <expression>
- quit
- breakpoints
  - (see next slide)

Man pages

- If you are unsure of what a C library function does, use man to find more information.
  - Example: man strcpy

- Note: man also supports various unix commands, but doesn’t hold info for C++

Exercise 2: Leaky code and Valgrind Demo

Memory Errors

- Use of uninitialized memory
- Reading/writing memory after it has been freed – Dangling pointers
- Reading/writing to the end of malloc'd blocks
- Reading/writing to inappropriate areas on the stack
- Memory leaks where pointers to malloc'd blocks are lost

Valgrind is your friend!!

Exercise 3: Structs and Pointers

Pointers and Structs

Defining structs

To define a struct, we use the **`struct`** statement.

A struct typically has a name (a tag), and one or more members.

The **`struct`** statement defines a new type.

```
struct fruit_st {
  Orchard* origin;
  int volume;
};
```

Pointers and Structs

Typedef

The C Programming language provides the keyword **typedef**, which defines an alternate name for a type

```
typedef struct fruit_st {
  Orchard* origin;
  int volume;
} Fruit;
```

Pointers and Structs

Define Names for Both Struct and Pointer

We could also use **typedef** to define names for both the struct instance and pointers to the struct at the same time.

```
typedef struct orchard_st {
  char name[20] ;
} Orchard;
```

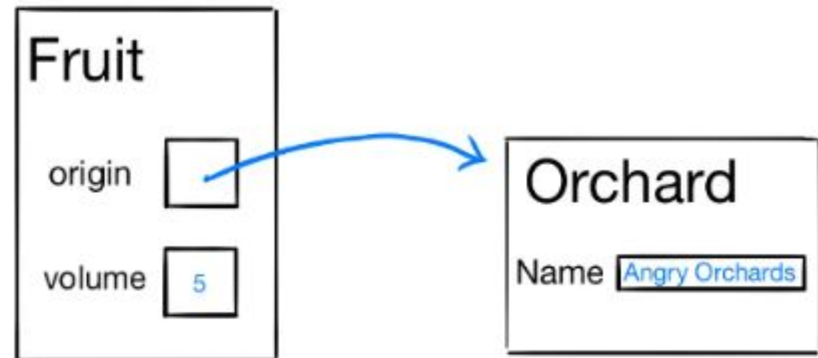
Pointers and Structs

Fruits & Orchards

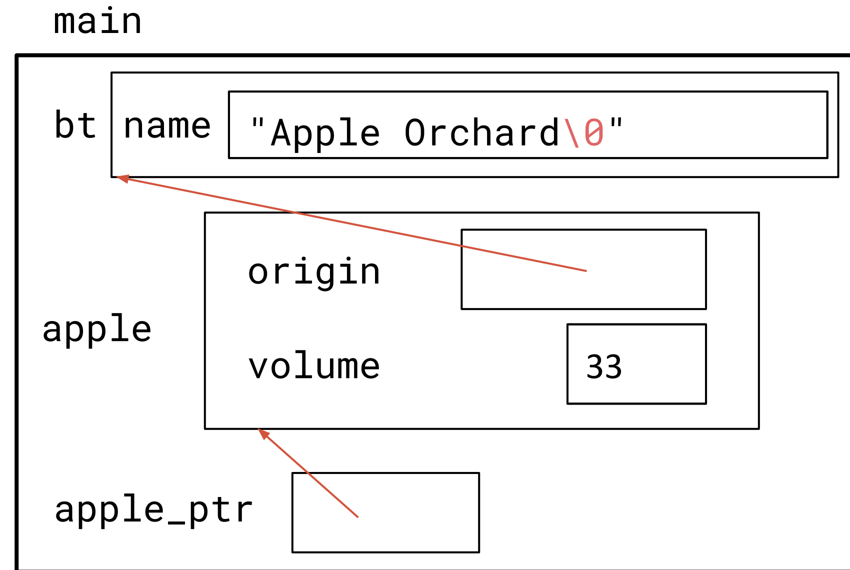
```
typedef struct fruit_st {
  Orchard* origin;
  int volume;
} Fruit;
```

```
typedef struct orchard_st {
  char name[20] ;
} Orchard;
```

```
Orchard o;
o.name = "Angry Orchards";
Fruit f;
f.origin = &o;
f.volume = 5;
```



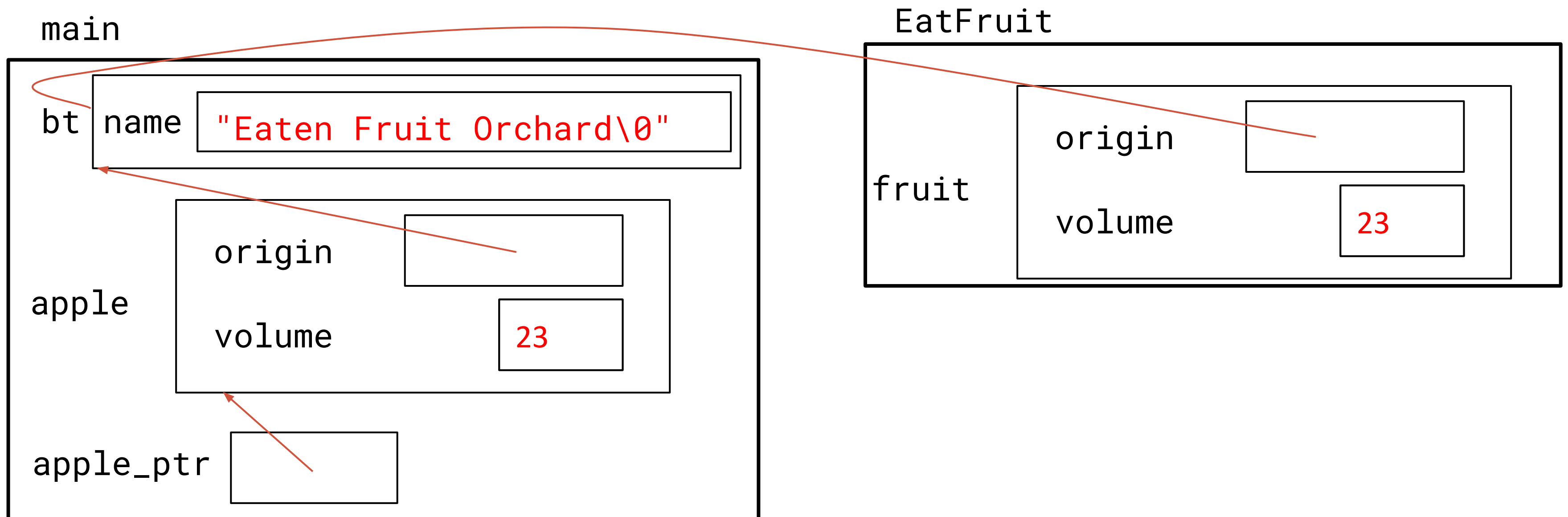
```c
int main(int argc, char* argv[]) {
  Orchard bt;
  strcpy(bt.name, "Apple Orchard");

  Fruit apple;
  Fruit* apple_ptr = &apple;
  apple.origin = &bt;
  apple.volume = 33;
  apple_ptr->volume = apple.volume;

  printf("1. %d, %s \n", apple_ptr->volume,
      apple_ptr->origin->name);
  …
```



console output

1, 33, Apple Orchard

```
…
apple.volume = EatFruit(apple);
printf("2. %d, %s \n", apple_ptr->volume,
    apple_ptr->origin->name);
```

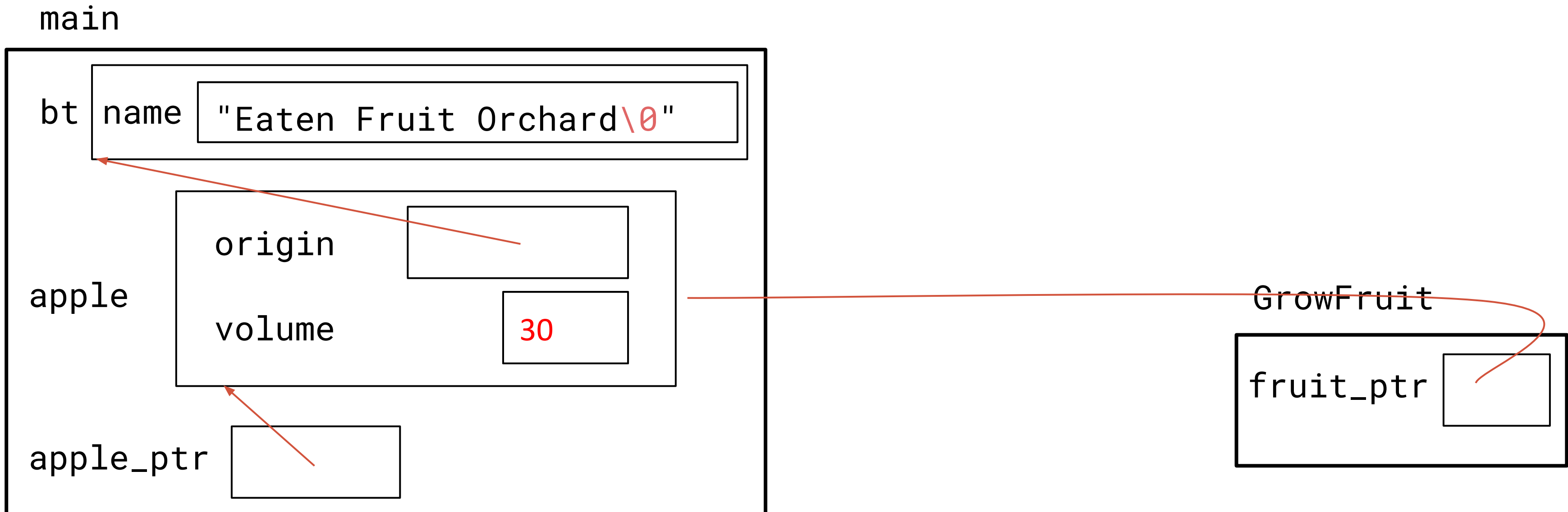



```
int EatFruit(Fruit fruit) {
  fruit.volume -= 10;
  strcpy(fruit.origin->name,
      "Eaten Fruit Orchard");
  return fruit.volume;
}
```

console output

```
1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
```

```
…
GrowFruit(apple_ptr);
printf("3. %d, %s \n", apple_ptr->volume,
    apple_ptr->origin->name);
```

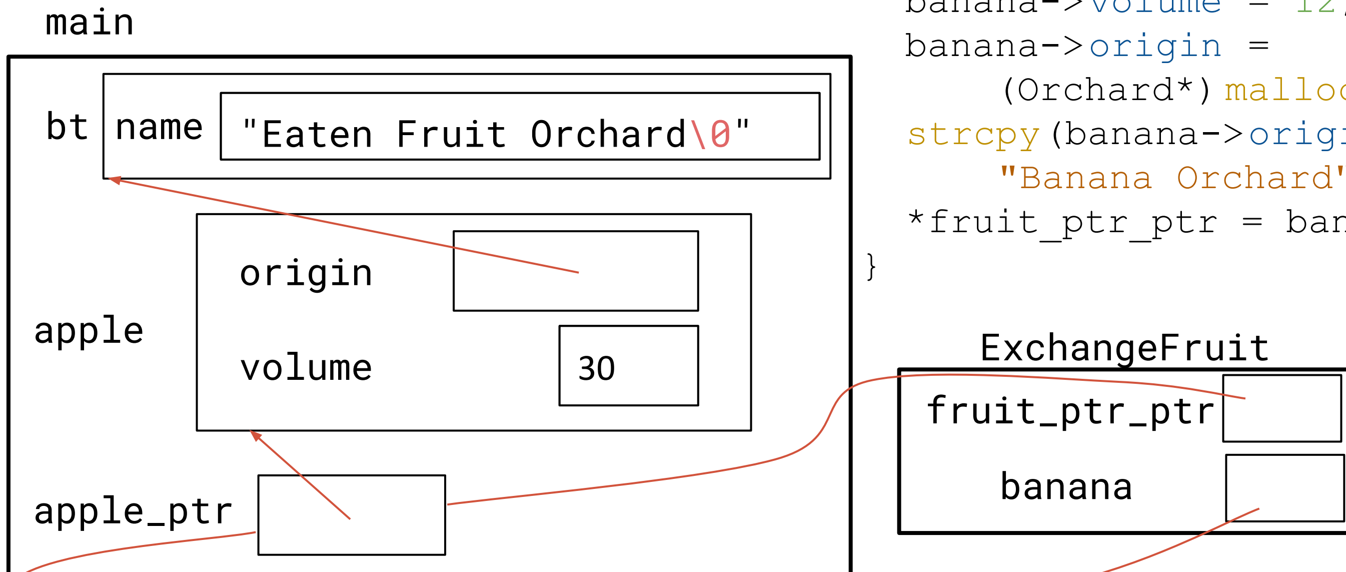


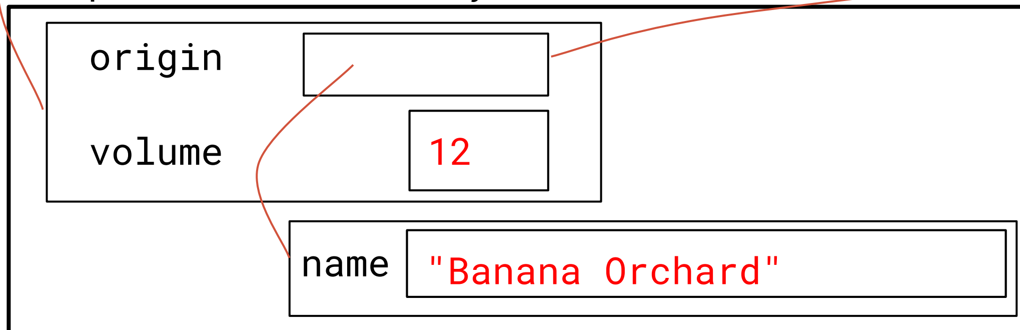
console output

1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
3, 30, Eaten Fruit Orchard

```
void GrowFruit(Fruit* fruit_ptr) {
  fruit_ptr->volume += 7;
}
```

```
void ExchangeFruit(Fruit** fruit_ptr_ptr) {
    Fruit *banana =
        (Fruit*) malloc(sizeof(Fruit));
    banana->volume = 12;
    banana->origin =
        (Orchard*) malloc(sizeof(Orchard));
    strcpy(banana->origin->name,
        "Banana Orchard");
    *fruit_ptr_ptr = banana;
}
```

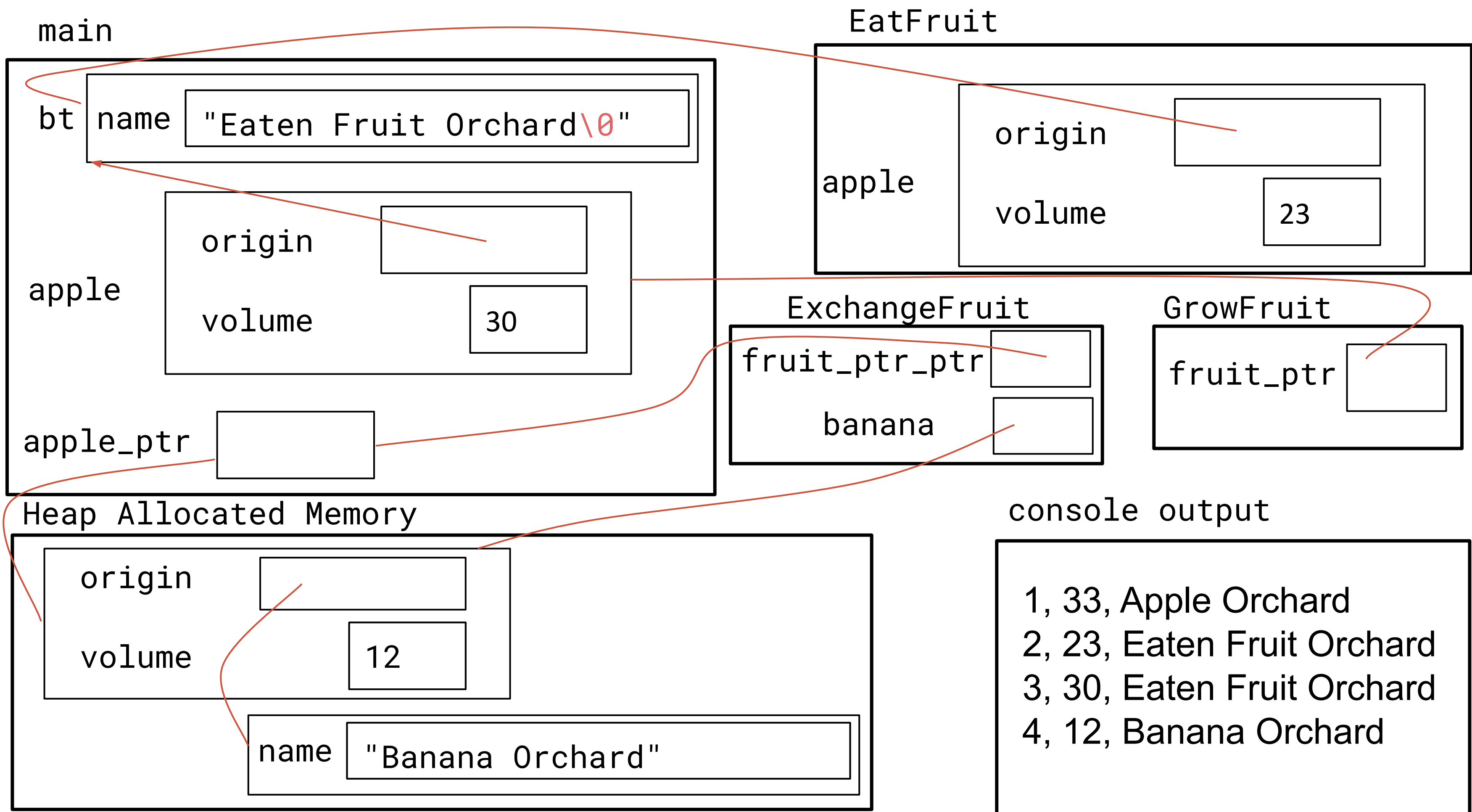




console output

1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
3, 30, Eaten Fruit Orchard
4, 12, Banana Orchard

```
ExchangeFruit(&apple_ptr);
printf("4. %d, %s \n", apple_ptr->volume,
    apple_ptr->origin->name);
```


main
bt
name
"Eaten Fruit Orchard\0"
apple
origin
volume
30
apple_ptr
EatFruit
apple
origin
volume
23
ExchangeFruit
fruit_ptr_ptr
banana
GrowFruit
fruit_ptr
Heap Allocated Memory
origin
volume
12
name
"Banana Orchard"
console output
1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
3, 30, Eaten Fruit Orchard
4, 12, Banana Orchard

Exercise 4:
Reverse a Linked List
[extra practice]

Linked List Reversal

Given a linked list with nodes defined as follows:

```
struct Node {
  int value;
  struct Node* next;
};
```

Complete the function to reverse the linked list and return the head of the resulting list.

```
struct Node* Reverse(struct Node* head) {

}
```

Linked List Reversal

```
struct Node* Reverse(struct Node* head) {
  struct Node* prev = NULL;
  struct Node* next = NULL;
  struct Node* current = head;

  while (current != NULL) {
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
   }

  return prev;
}
```

```
struct Node {
  int value;
  struct Node* next;
};
```

Exercise 5:
Sorted Array to Binary Search Tree [extra practice]

Sorted Array to Binary Search Tree

A node in a tree is defined as follows:

```
struct TreeNode {
  int value;
  struct TreeNode* left;
  struct TreeNode* left;
};
```

Complete the implementation of the sortedArrayToBST function to convert a sorted integer array into a **balanced** binary search tree. The client to this method will invoke it as follows:

```
struct TreeNode* SortedArrayToBST(int[] arr, int low, int high) {

}
```

Sorted Array to Binary Search Tree

```
struct TreeNode {
  int value;
  struct TreeNode* left;
  struct TreeNode* left;
};
```

```
struct TreeNode *SortedArrayToBST(int[] arr, int low, int high) {
  if (low > high) {
    return NULL;
  }

  // Make the middle element the root of this subtree.
  int mid = (low + high) / 2;
  struct TreeNode *root = (struct TreeNode*)malloc(sizeof(TreeNode));
  root->value = arr[mid];

  // Construct the left subtree and assign it to be the left child.
  root->left = SortedArrayToBST(arr, low, mid - 1);

  // Construct the right subtree and assign it to be the right child.
  root->right = SortedArrayToBST(arr, mid + 1, high);

  return root;
}
```