

CSE 1320

Week of 04/24/2023

Instructor : Donna French

Using dynamic memory allocation to read a file with variable length fields.

```
1 Tacos|Flamin' Hot Doritos Locos Taco|Seasoned Beef, Cheese, Lettuce
2 Tacos|Flamin' Hot Doritos Locos Tacos Supreme|Seasoned Beef, Cheese, Lettuce, Tomatoes, Sour Cream
3 Burritos|Loaded Taco Grande Burrito|Seasoned Beef, Cheese, Lettuce, Red Strips, Sour Cream
```

```

typedef struct
{
    char *category;
    char *name;
    char *whatsincluded;
}
TACOBELL;

int main(int argc, char *argv[])
{
    TACOBELL Menu[20] = {};
    char *token = NULL;
    char filename[20] = {};
    FILE *FH = NULL;
    char FileLine[200];
    int MenuCount = 0;
    int i;

```

```

strcpy(filename, argv[1]);
FH = fopen(filename, "r+");

if (FH == NULL)
{
    printf("File did not open");
    exit(0);
}

```

```

1 Tacos|Flamin' Hot Doritos Locos Taco|Seasoned Beef, Cheese, Lettuce
2 Tacos|Flamin' Hot Doritos Locos Tacos Supreme|Seasoned Beef, Cheese, Lettuce, Tomatoes, Sour Cream
3 Burritos|Loaded Taco Grande Burrito|Seasoned Beef, Cheese, Lettuce, Red Strips, Sour Cream

```

```
while (fgets(FileLine, sizeof(FileLine)-1, FH))
{
    token = strtok(FileLine, "|");
    Menu[MenuCount].category = malloc(strlen(token)*sizeof(char)+1);
    strcpy(Menu[MenuCount].category, token);

    token = strtok(NULL, "|");
    Menu[MenuCount].name = malloc(strlen(token)*sizeof(char)+1);
    strcpy(Menu[MenuCount].name, token);

    token = strtok(NULL, "|");
    Menu[MenuCount].whatsincluded = malloc(strlen(token)*sizeof(char)+1);
    strcpy(Menu[MenuCount].whatsincluded, token);

    MenuCount++;
}
```

```
1 Tacos|Flamin' Hot Doritos Locos Taco|Seasoned Beef, Cheese, Lettuce
2 Tacos|Flamin' Hot Doritos Locos Tacos Supreme|Seasoned Beef, Cheese, Lettuce, Tomatoes, Sour Cream
3 Burritos|Loaded Taco Grande Burrito|Seasoned Beef, Cheese, Lettuce, Red Strips, Sour Cream
```

```
for (i = 0; i < MenuCount; i++)
{
    printf("Category : %s\nName      : %s\n\nWhat's Included : %s\n\n",
        Menu[i].category, Menu[i].name, Menu[i].whatsincluded);
}
```

```
for (i = 0; i < MenuCount; i++)
{
    free(Menu[i].category);
    free(Menu[i].name);
    free(Menu[i].whatsincluded);
}
```


```
1 Tacos|Flamin' Hot Doritos Locos Taco|Seasoned Beef, Cheese, Lettuce
2 Tacos|Flamin' Hot Doritos Locos Tacos Supreme|Seasoned Beef, Cheese, Lettuce, Tomatoes, Sour Cream
3 Burritos|Loaded Taco Grande Burrito|Seasoned Beef, Cheese, Lettuce, Red Strips, Sour Cream
```

```
int main(void)
{
    int test = 0;
    printf("Enter a value ");
    scanf("%d", &test);

    int i;
    for (i = test; i > 0; i--)
        printf("%d ", i);
    for (i = 1; i <= test; i++)
        printf("%d ", i);

    return 0;
}
```

 frenchdm@omega:~

[frenchdm@omega ~]\$ 



What is the condition that makes it stop?

```
int main()
{
    int test = 0;
    printf("Enter a value ");
    scanf("%d", &test);

    int i;
    for (i = test; i > 0; i--)
        printf("%d ", i);
    for (i = 1; i <= test; i++)
        printf("%d ", i);

    return 0;
}
```

5	4	3	2	1	1	2	3	4	5
---	---	---	---	---	---	---	---	---	---

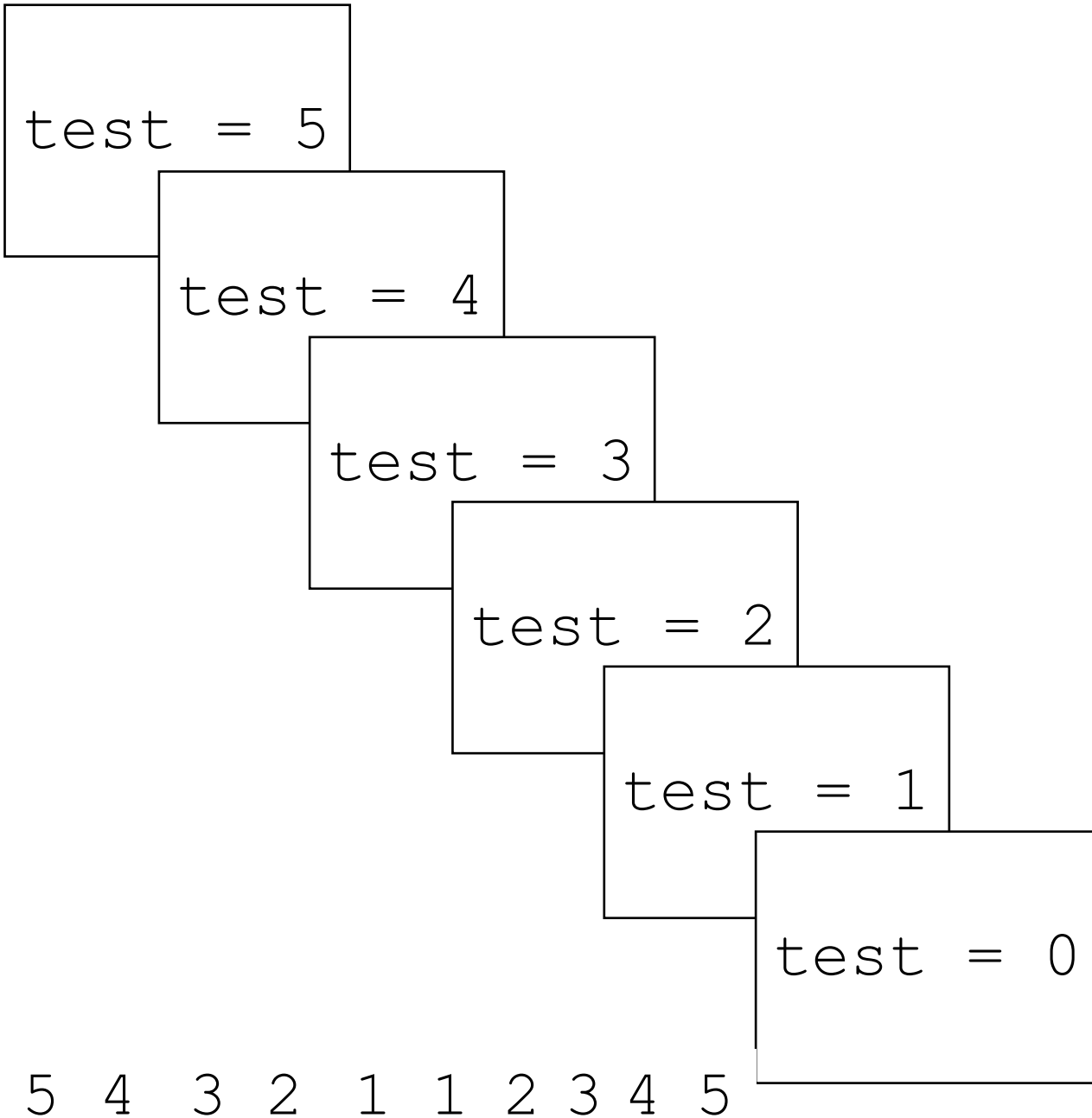
```
void printFun(int test)
{
    if (test < 1)
        return;

    else
    {
        printf("%d ", test);

        printFun(test-1);

        printf("%d ", test);

        return;
    }
}
```

```
void printFun(int test)
{
    if (test < 1)
        return;

    else
    {
        printf("%d ", test);
        printFun(test-1);
        printf("%d ", test);
        return;
    }
}
```

```
void printFun(int test)
{
    if (test < 1)
        return;

    else
    {
        printf("%d ", test);

        printFun(test-1);

        printf("%d ", test);

        return;
    }
}
```

```
void printFun(int test)
{
    if (test >= 1)
    {
        printf("%d ", test);

        printFun(test-1);

        printf("%d ", test);
    }
}
```

Example Using Recursion: Fibonacci Series

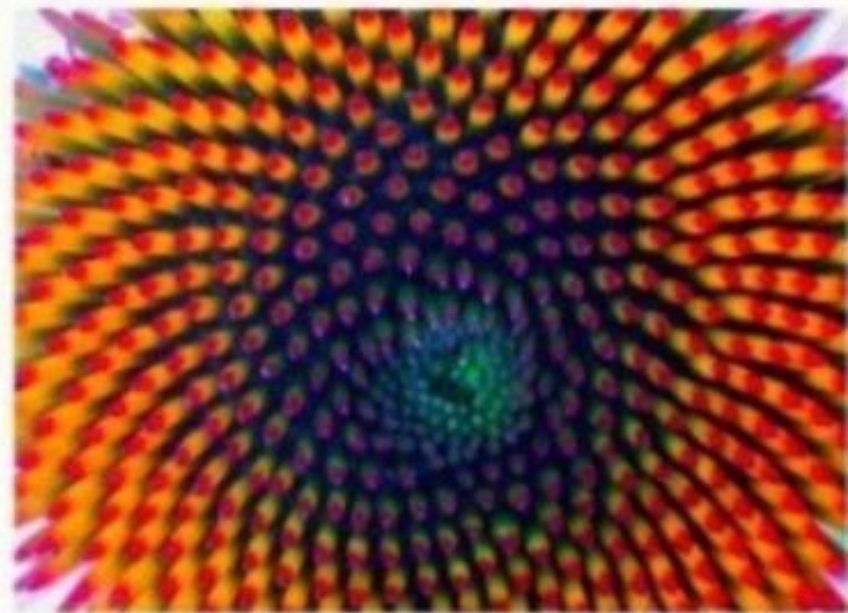
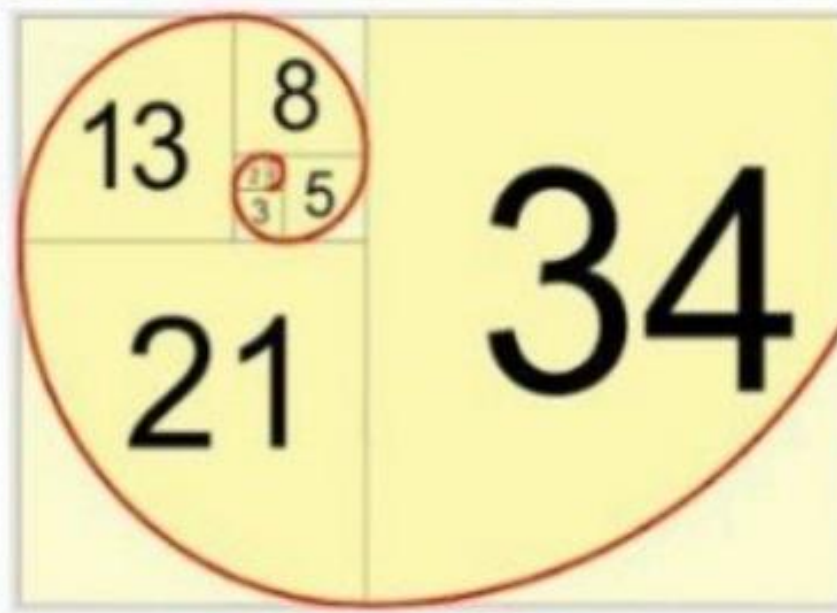
The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

The series occurs in nature and, in particular, describes a form of spiral.

The ratio of successive Fibonacci numbers converges to a constant value of 1.618....



Example Using Recursion: Fibonacci Series

The Fibonacci series may be defined recursively as follows:

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(2) = 1$$

$$\text{fibonacci}(3) = 2$$

$$\text{fibonacci}(4) = 3$$

$$\text{fibonacci}(5) = 5$$

$$\text{fibonacci}(6) = 8$$

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

Example Using Recursion: Fibonacci Series

The Fibonacci series may be defined recursively as follows:

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

We can create a program to calculate the n^{th} Fibonacci number recursively using a function we'll call `fibonacci`.

```
unsigned long long int result = fibonacci(number);
```

```
unsigned long long int fibonacci(unsigned int n)
{
    if (n == 0 || n == 1)
    {
        return n;
    }
    else
    {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Enter an integer: 0

```
23             unsigned long long int result = fibonacci(number);
```

fibonacci (n=0) at recur2Demo.c:6

```
4     unsigned long long int fibonacci(unsigned int n)
5     {
6         if (n == 0 || n == 1)
7         {
8             return n;
9         }
```

Fibonacci(0) = 0

Fibonacci(1) = 1

Fibonacci(2) = 1

Fibonacci(3) = 2

Fibonacci(4) = 3

Fibonacci(5) = 5

Using Recursion

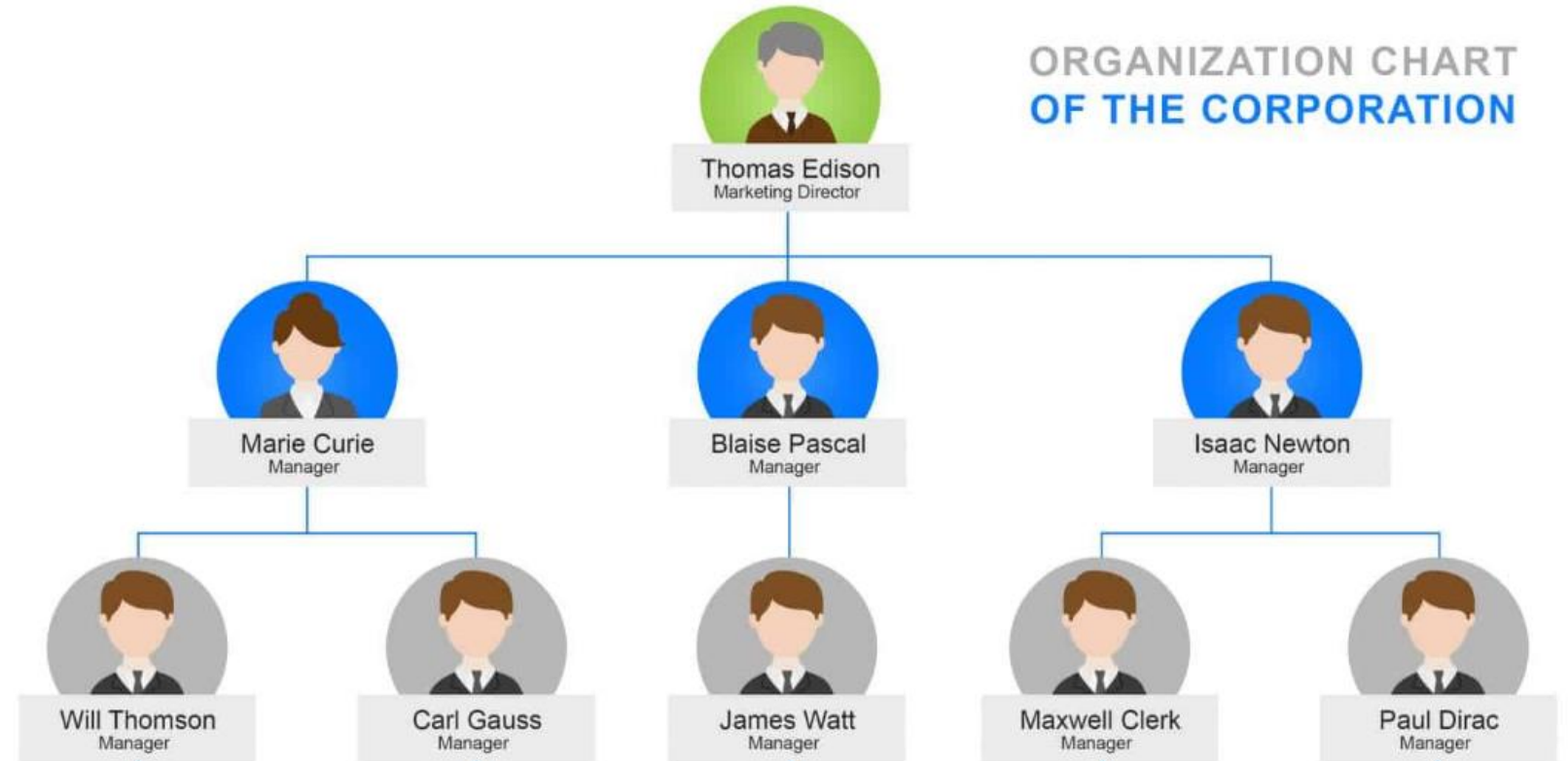
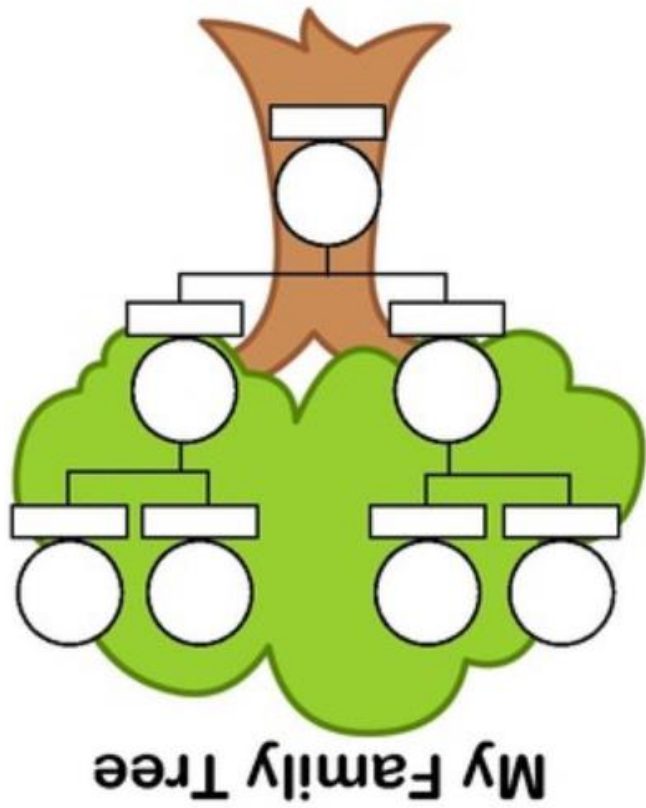
Any problem that can be solved recursively can also be solved iteratively.

A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug.

Another reason to choose a recursive solution is that an iterative solution may not be apparent.

Tree Data Structure

- Linked lists, stacks, and queues are all **linear** structures
 - one node follows another
 - each node contains a pointer to the next node
- Trees are **non-linear** structures
 - more than one node can follow another
 - each node contains pointers to an arbitrary number of nodes
 - the number of nodes that follow can vary from one node to another.
- Trees organize data hierarchically instead of linearly.



Binary Tree

What is a binary tree?

A binary tree is a non-linear tree-like data structure consisting of nodes where each node has up to two child nodes, creating the branches of the tree.

The two children are usually called the left and right nodes.

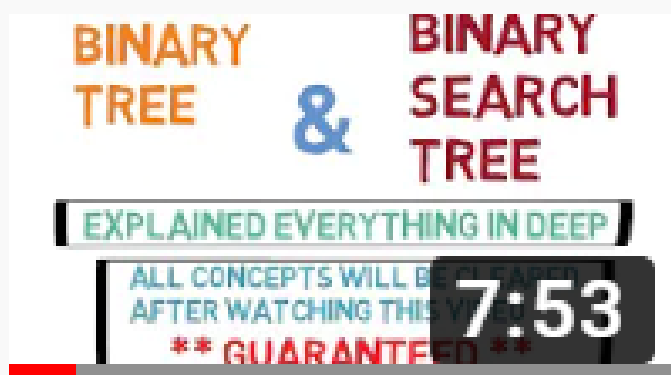
Parent nodes are nodes with children. Parent nodes can be child nodes themselves.

Binary trees are used to implement binary search trees and binary heaps. They are also often used for sorting data as in a heap sort.



Introduction to Tree Data Structure

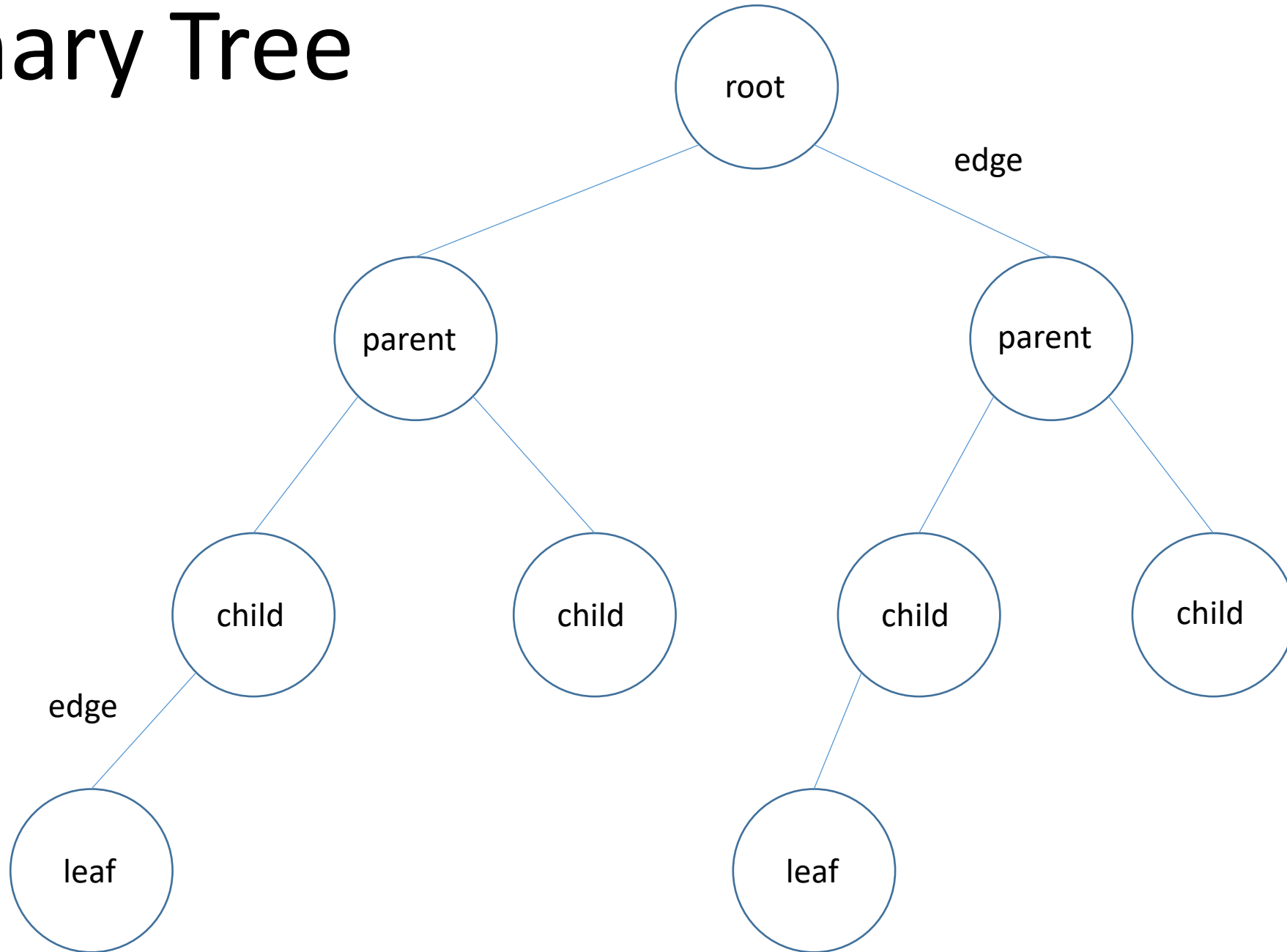
Codearchery



Binary Tree and Binary Search Tree in Data Structure

Codearchery

Binary Tree



Binary Tree

Tree Vocabulary

topmost node

node directly under another node

node directly above another node

node with no children

link between two nodes

length of the path from the root

length of the path from the node to

the deepest leaf reachable from it

root of the tree

child

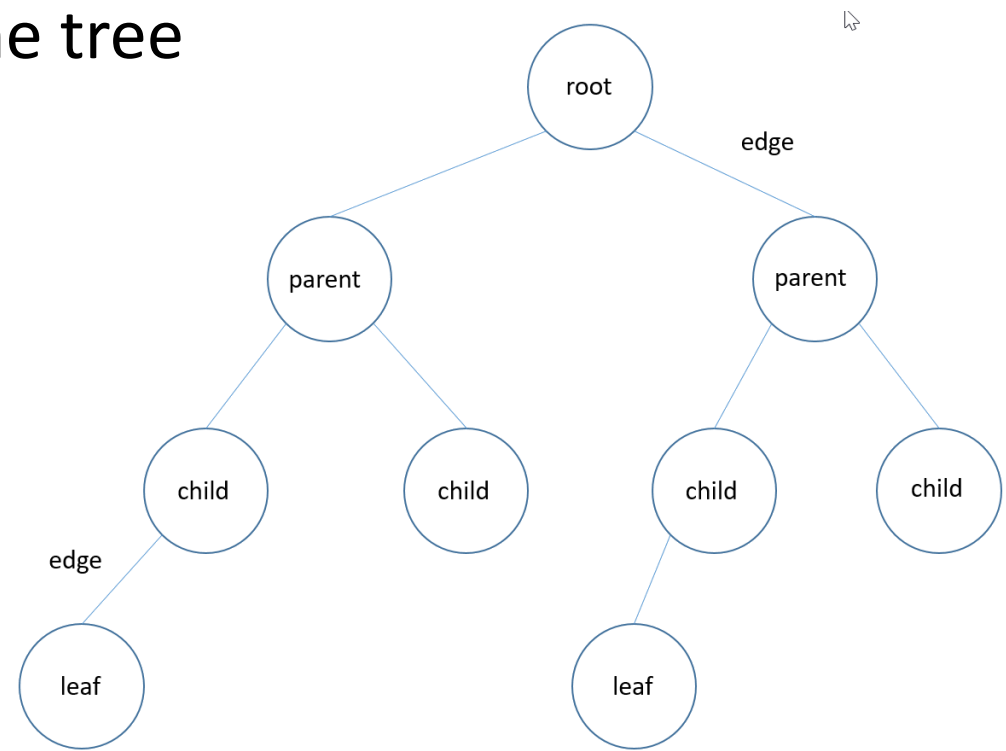
parent

leaf

edge

depth

height



Linked List vs Binary Tree

Linked List Node

```
struct node
{
    int node_number;
    struct node *next_ptr;
}

struct node *LinkedListHead;
```

Binary Tree Node

```
struct node
{
    int node_number;
    struct node *left_ptr;
    struct node *right_ptr;
};

struct node *root;
```


Binary Tree vs Linked List

Linked List

```

NewNode = malloc(sizeof(struct node));
NewNode->node_number = NodeNumber;
NewNode->next_ptr = NULL;
```

Binary Tree

```

NewNode = malloc(sizeof(struct node));
NewNode->node_number = NodeNumber;
NewNode->left_ptr = NULL;
NewNode->right_ptr = NULL;
```

Binary Tree vs Linked List

Add a node to the end of a linked list

```
NewNode = malloc(sizeof(struct node));
```

```
NewNode->node_number = NodeNumber;
```

Set the pointer of the last node to the new node

```
TempPtr->next_ptr = NewNode;
```

Add a node to a binary tree

```
NewNode = malloc(sizeof(struct node));
```

```
NewNode->node_number = NodeNumber;
```

```
NewNode->left_ptr = NULL;
```

```
NewNode->right_ptr = NULL;
```

Set the parent node's left or right ptr to the address of the new child

```
/* Allocates memory for a new node with the given data and sets the left and
   right pointers to NULL */
node *CreateNewNode(int NodeNumber)
{
    // Allocate memory and assign pointers
    node *node = malloc(sizeof(node));
    node->left_ptr = NULL;
    node->right_ptr = NULL;

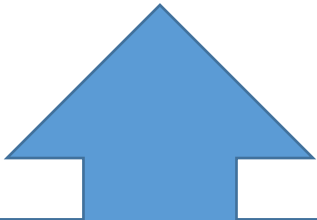
    // Assign data to this node
    node->node_number = NodeNumber;

    printf("Node Number %d %p\n", NodeNumber, node);

    return(node);
}
```

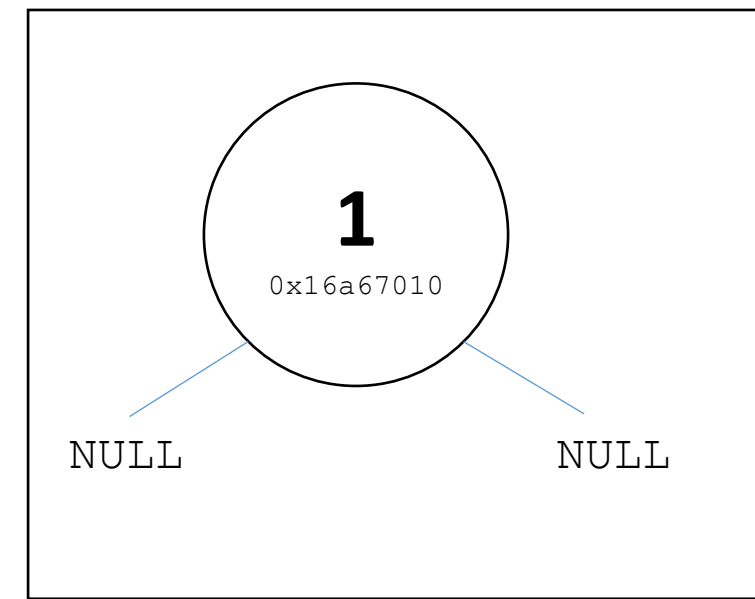
```
typedef struct node
{
    int node_number;
    struct node *left_ptr;
    struct node *right_ptr;
}
node;
```

```
Node Number 1 0x16a67010
```



Reminder!!
When you typedef a structure that contains a
pointer to the structure, you must use
typedef struct node
and not just
typedef struct

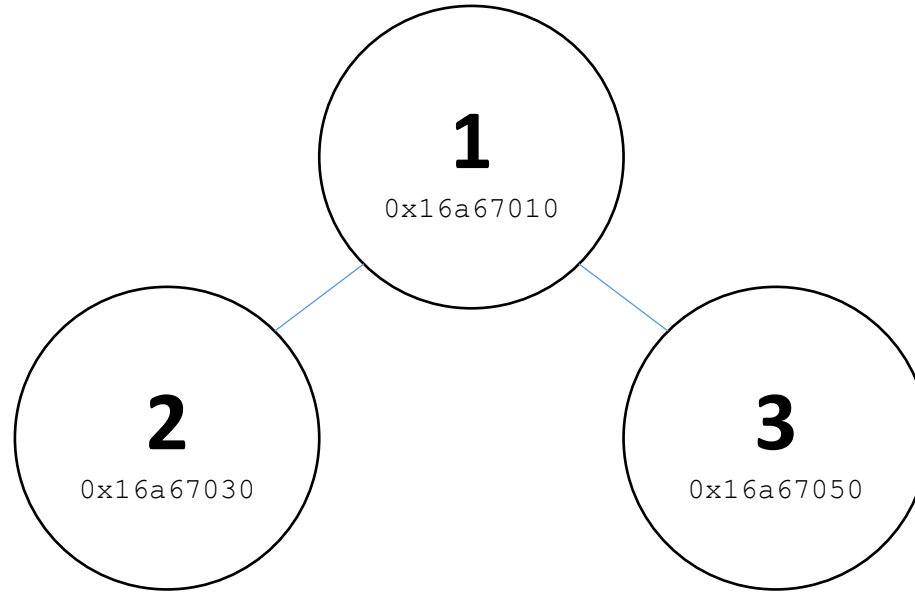
```
/* declare root of tree */  
struct node *root;  
  
/* create root and label with "1" */  
root = CreateNewNode(1);
```



```
// Print pointer values  
printf("\nleft_ptr(1) %p\tright_ptr(1) %p\n",  
        root->left_ptr, root->right_ptr);
```

<code>left_ptr(1) (nil)</code>	<code>right_ptr(1) (nil)</code>
--------------------------------	---------------------------------

```
root->left_ptr  = CreateNewNode(2);  
root->right_ptr = CreateNewNode(3);
```



```
printf("\nleft_ptr(2)  %p\tright_ptr(3)  %p\n",root->left_ptr, root->right_ptr);
```

```
Node Number 2  0x16a67030
```

```
Node Number 3  0x16a67050
```

```
left_ptr(2)  0x16a67030   right_ptr(3)  0x16a67050
```

Binary Tree vs Binary Search Tree

Binary Tree

Each node can have a maximum of two child nodes and there is no order to how the nodes are organized in the tree.

Binary Search Tree

Each node can have a maximum of two child nodes and there is a relative order to how the nodes are organized in the tree.

Binary Search Tree

A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node, and the values in any right subtree are greater than the value in its parent node.

The shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

Binary Search Tree

What makes a binary tree a binary search tree?

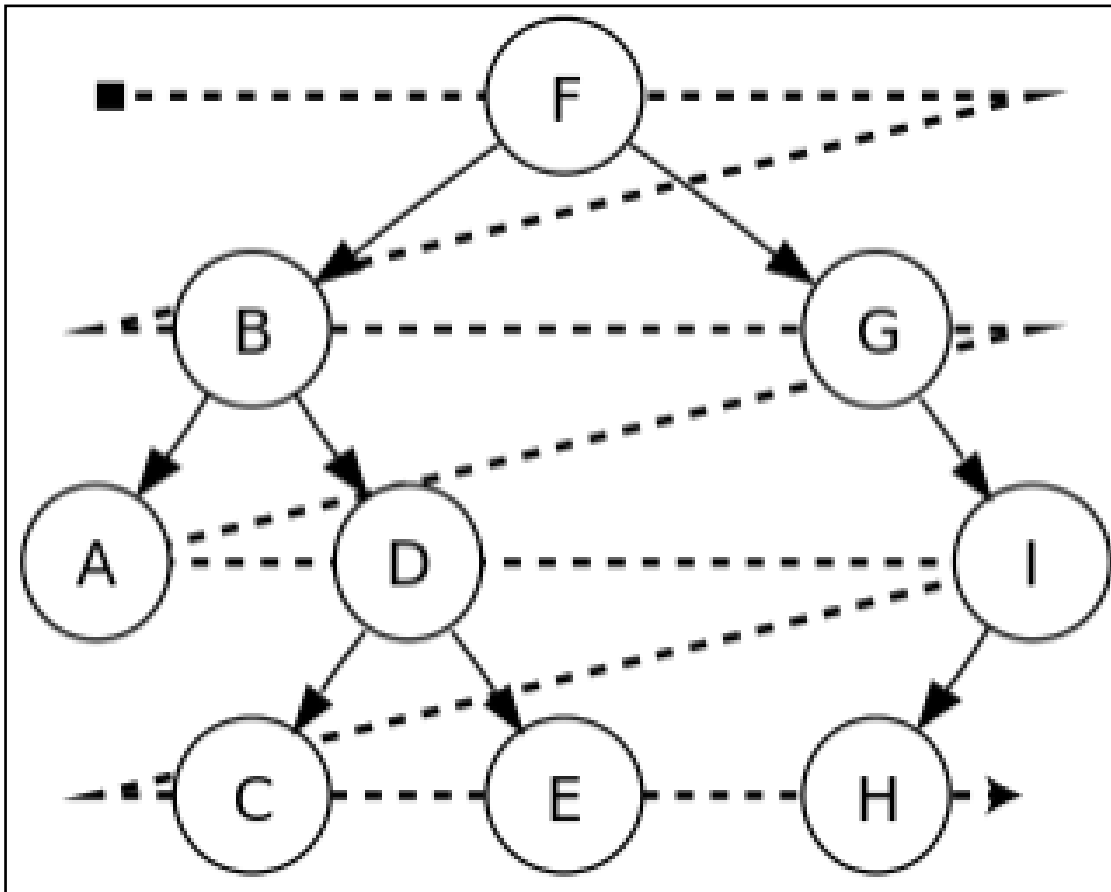
- All nodes in the left subtree are less than the root
- All nodes in the right subtree are greater than the root
- Each subtree is itself a binary search tree
- No duplicates allowed*

Binary Search Tree (BST)

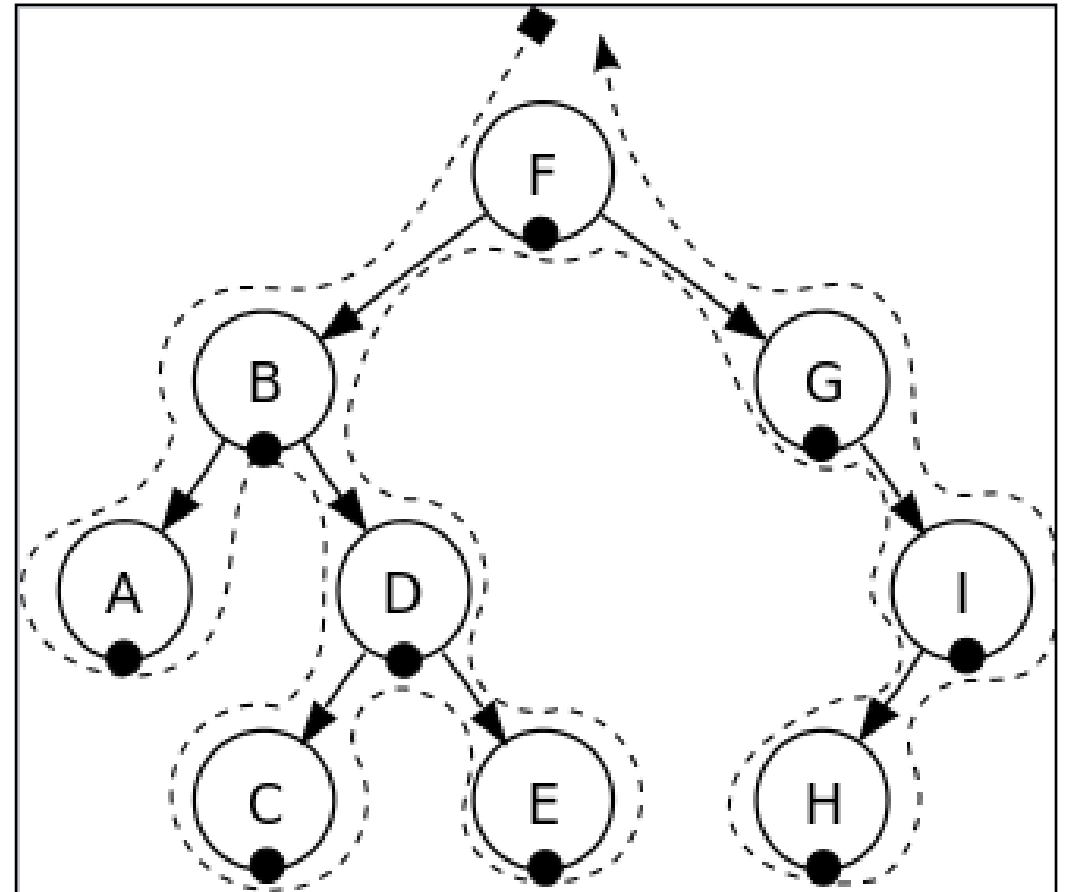
- Linear data structures (linked list, queues and stacks) are traversed in a linear order. Tree structures are traversed in multiple ways - from any given node, there is more than one possible next node in the traversal path
- Tree structures may be traversed in
 - Breadth-first Order
 - Depth-first Order

BST Breadth-first vs Depth-first Traversal

Breadth-first



Depth-first



BST Depth-first Traversals

- Inorder Traversal
 - Gives us the nodes in increasing order
- Preorder Traversal
 - Parent nodes are visited before any of its child nodes
 - Used to create a copy of the tree
 - File systems use it to track your movement through directories
- Postorder Traversal
 - Used to delete the tree
 - File systems use it to delete folders and the files under them

BST Depth-first Traversals

Depth First Tree Traversals

Preorder

Root, Left, Right

4 2 1 3 5

Postorder

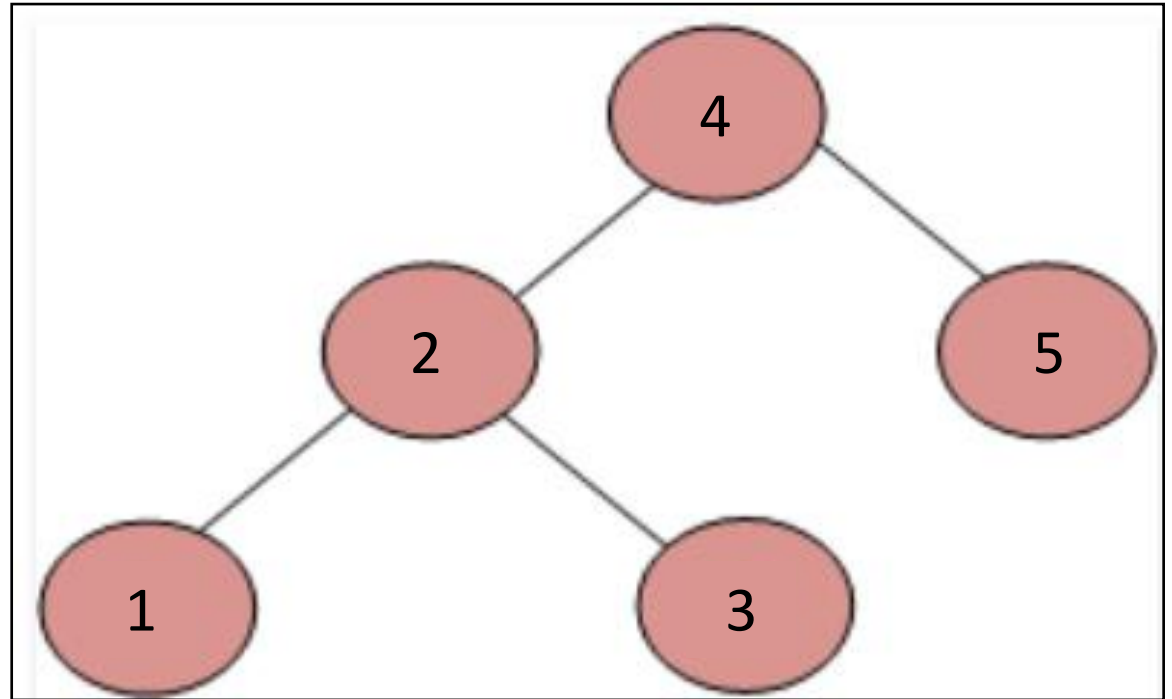
Left, Right, Root

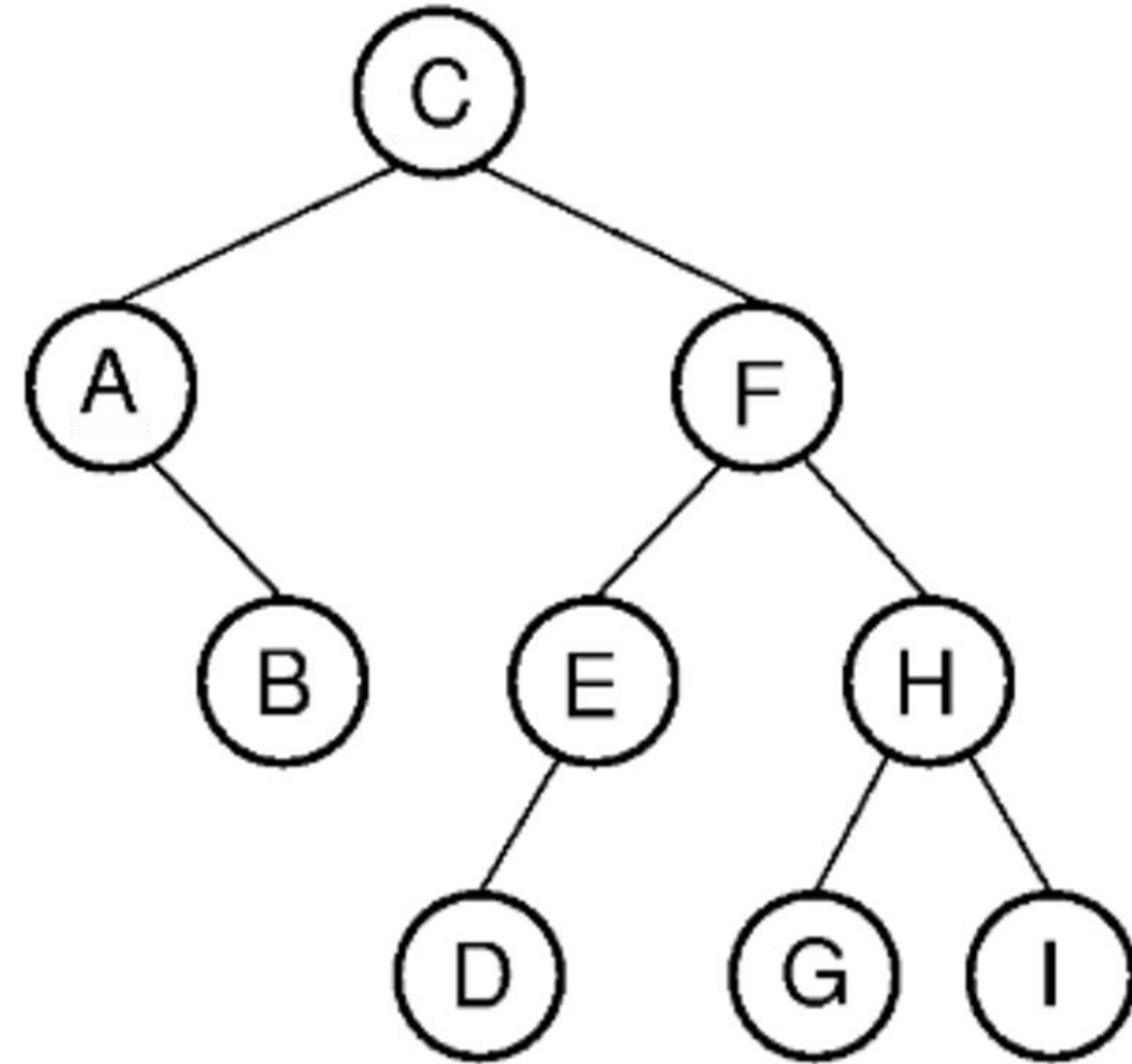
1 3 2 5 4

Inorder

Left, Root, Right

1 2 3 4 5





Depth First Tree Traversals

Preorder

Root, Left, Right

C A B F E D H G I

Postorder

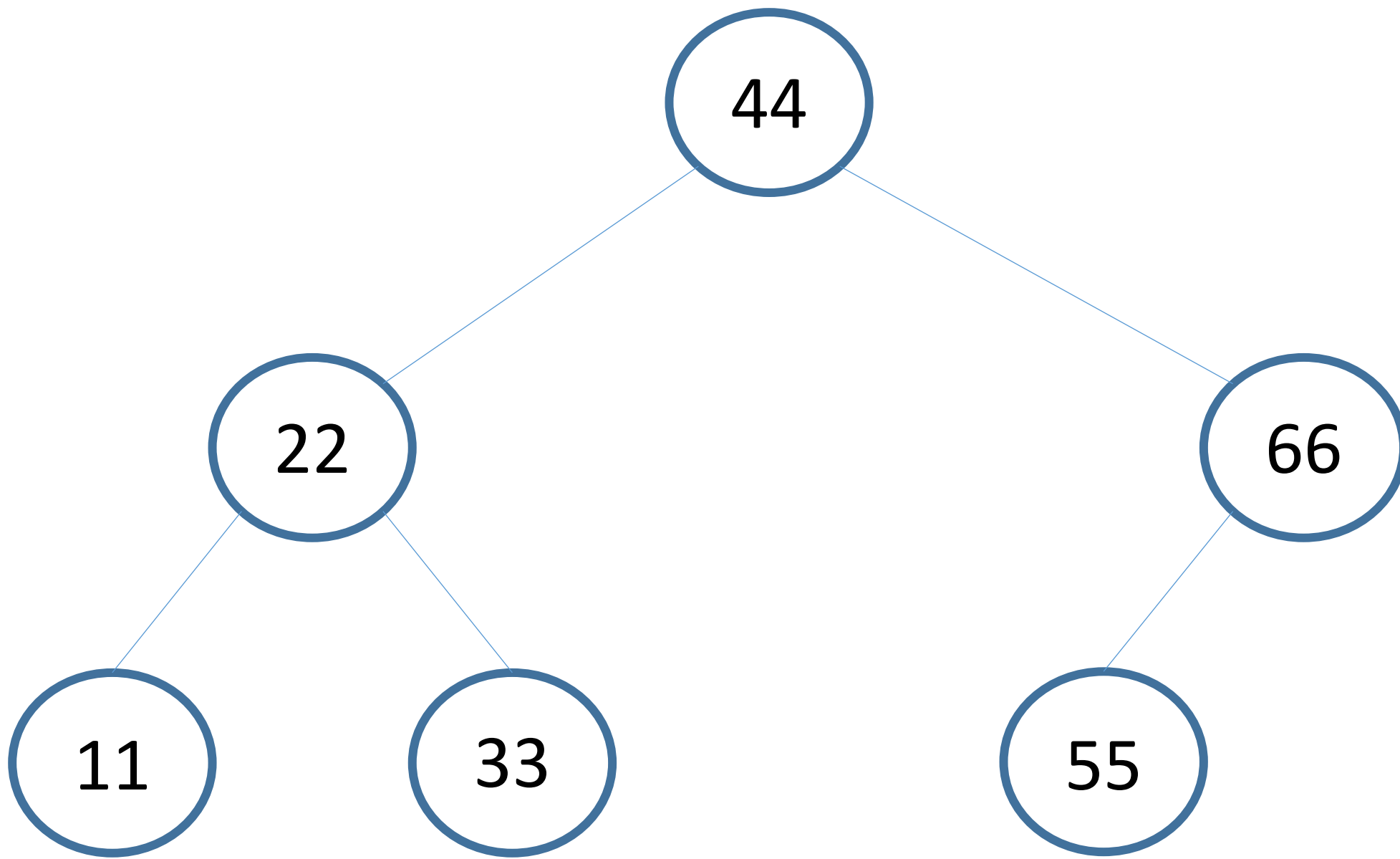
Left, Right, Root

B A D E G I H F C

Inorder

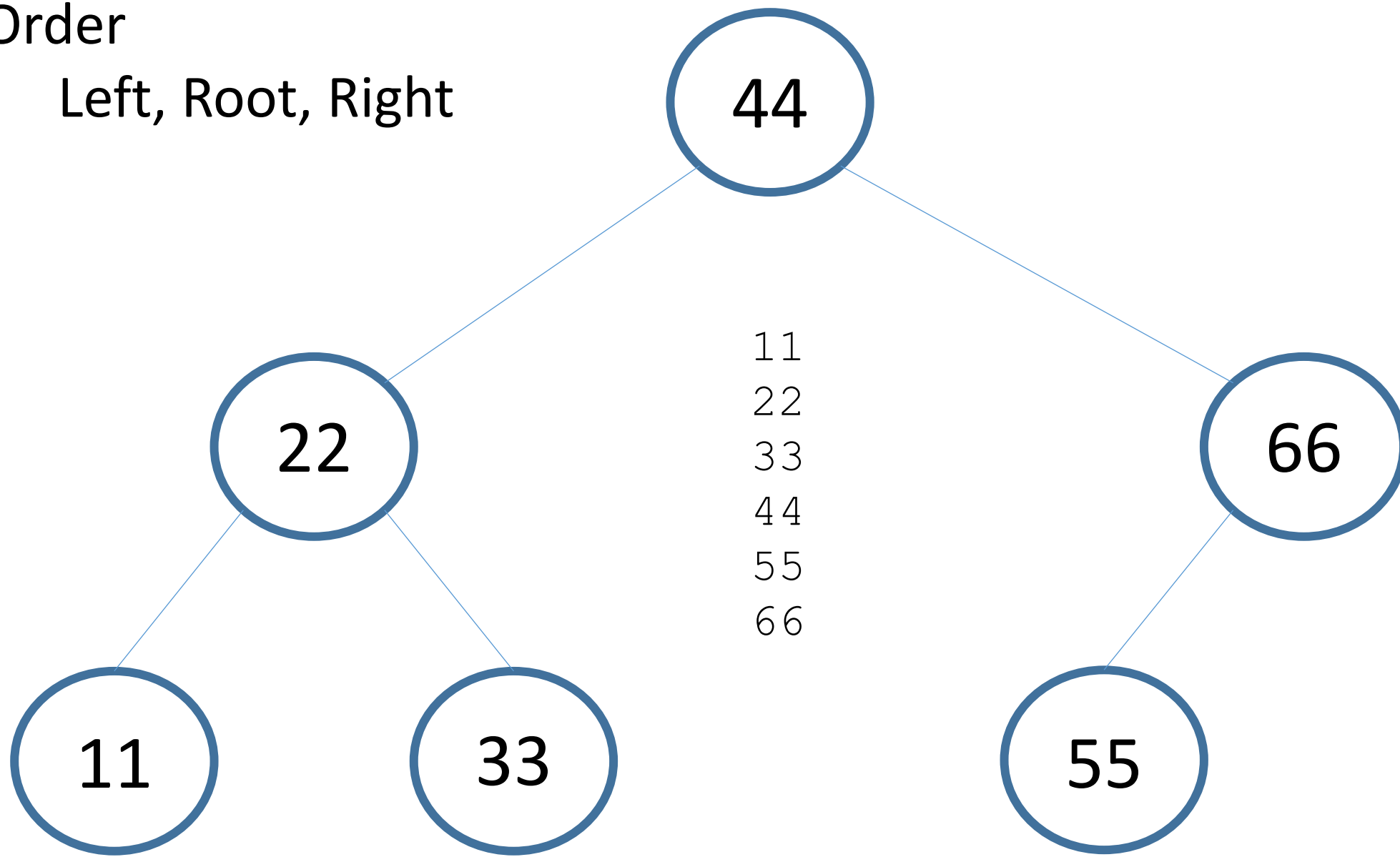
Left, Root, Right

A B C D E F G H I

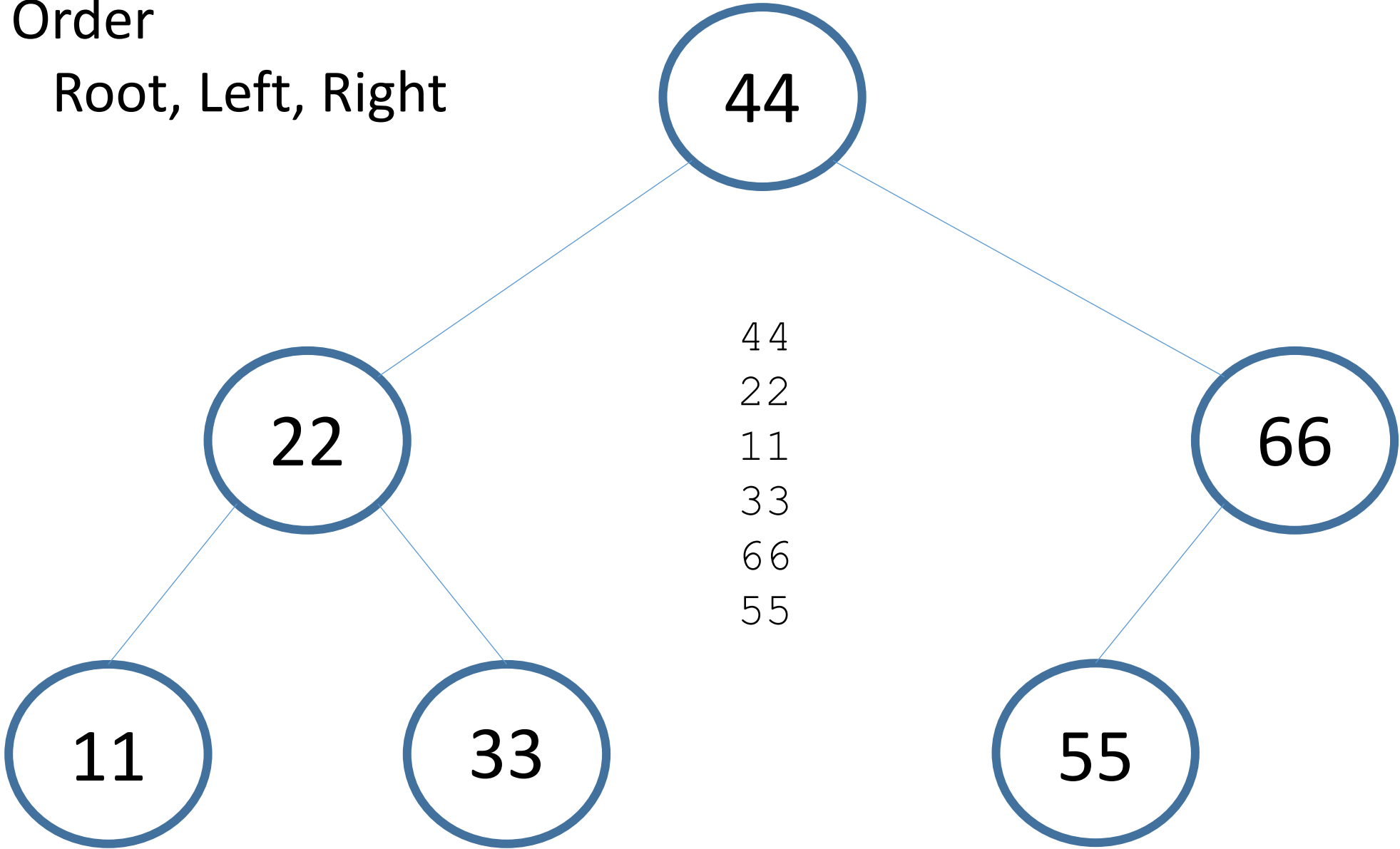


In Order

Left, Root, Right

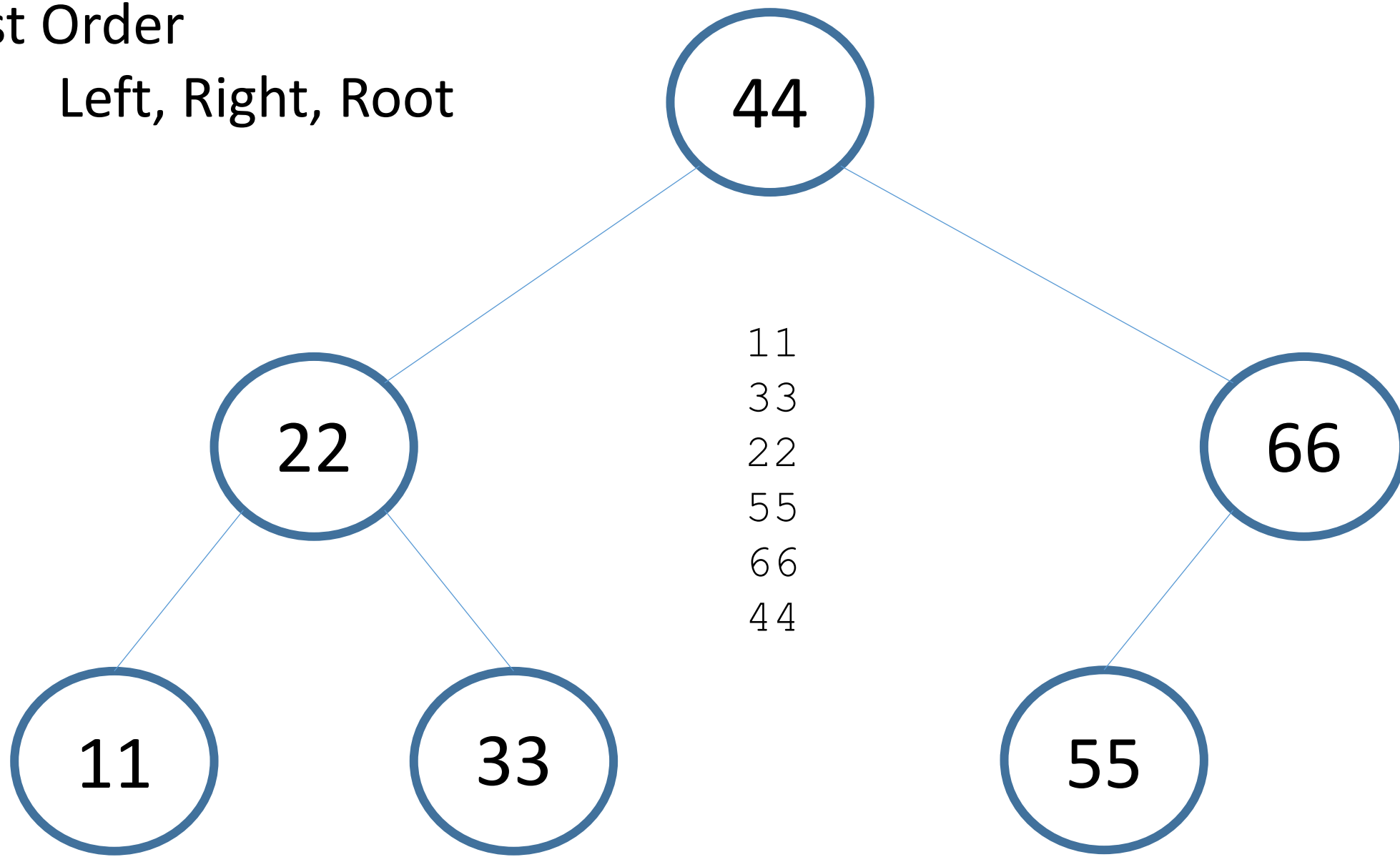


Pre Order
Root, Left, Right



Post Order

Left, Right, Root



How many nodes in the tree? 9

Enter data for node 1 : 47

Enter data for node 2 : 25

Enter data for node 3 : 77

Enter data for node 4 : 11

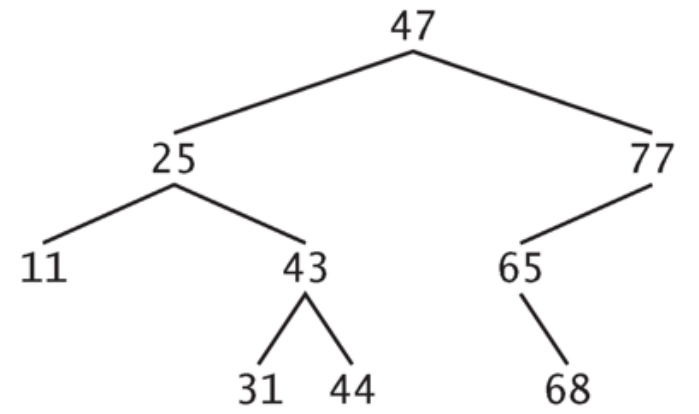
Enter data for node 5 : 43

Enter data for node 6 : 65

Enter data for node 7 : 31

Enter data for node 8 : 44

Enter data for node 9 : 68



47

25

77

11

42

65

31

44

68

BST Traversal in Inorder

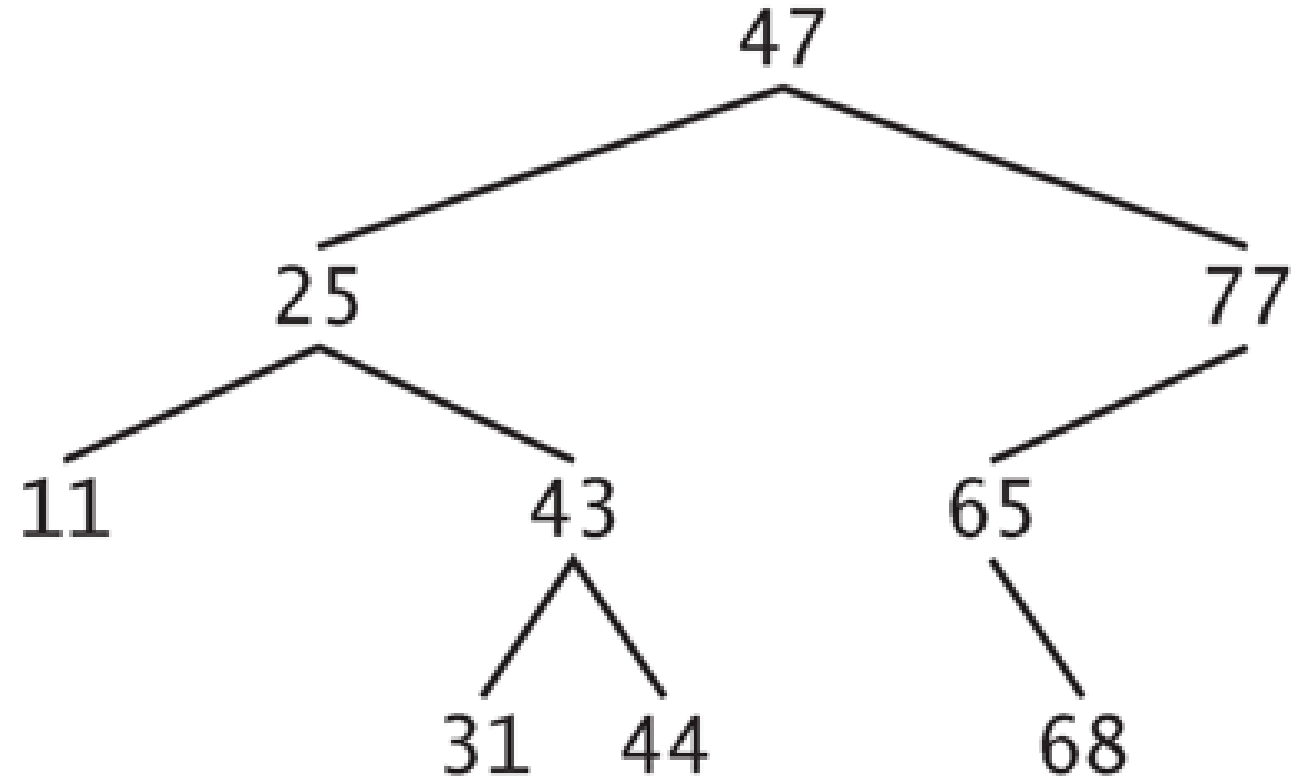
Node4-11	Node2-25	Node7-31
Node5-43	Node8-44	Node1-47
Node6-65	Node9-68	Node3-77

BST Traversal in Preorder

Node1-47	Node2-25	Node4-11
Node5-43	Node7-31	Node8-44
Node3-77	Node6-65	Node9-68

BST Traversal in Postorder

Node4-11	Node7-31	Node8-44
Node5-43	Node2-25	Node9-68
Node6-65	Node3-77	Node1-47



```
typedef struct node
{
    int node_data;
    struct node *right;
    struct node *left;
}
node;

node *root = NULL;

AddBSTNode (&root, node_data);
```

```
void AddBSTNode(node **current_node, int add_data)
{
    if (*current_node == NULL)
    {
        *current_node = malloc(sizeof(node));
        (*current_node)->left = (*current_node)->right = NULL;
        (*current_node)->node_data = add_data;
    }
    else
    {
        if (add_data < (*current_node)->node_data )
            AddBSTNode(&(*current_node)->left, add_data);

        else if(add_data > (*current_node)->node_data )
            AddBSTNode(&(*current_node)->right, add_data);

        else
            printf(" Duplicate Element !! Not Allowed !!!");
    }
}
```

root is set to NULL

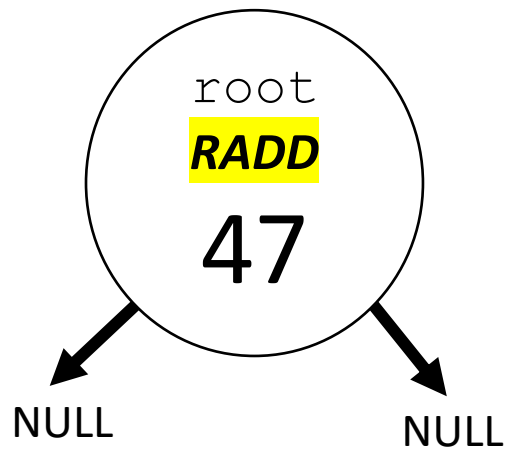
AddBSTNode(address of root, 47);

current_node is NULL

malloc a new node (address **RADD**)

set left and right pointers to NULL

set node_data to 47



Variable
node_data;
right;
left;

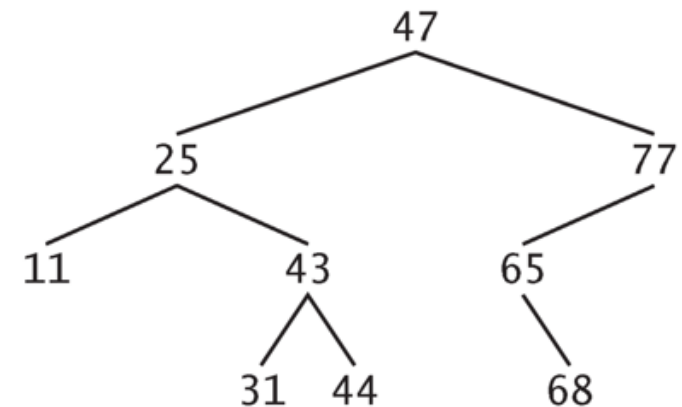
Memory Address

ND47 (47)
ROOTRP (NULL)
ROOTLP (NULL)

```
node *root = NULL;
```

```
AddBSTNode(&root, node_data);
```

```
void AddBSTNode(node **current_node, int add_data)
{
    if (*current_node == NULL)
    {
        *current_node = malloc(sizeof(node));
        (*current_node)->left = (*current_node)->right = NULL;
        (*current_node)->node_data = add_data;
    }
}
```



root is set to **RADD**

AddBSTNode (**RADD**, 25) ;

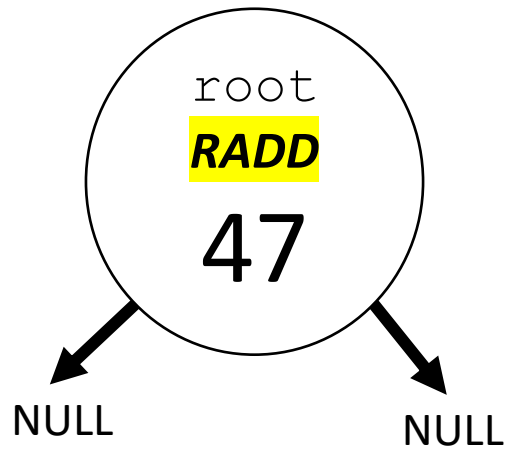
current_node is **not** NULL so else

add_data is 25 and node_data is 47

$25 < 47$

so call AddBSTNode () with the address of current node's (**RADD**) left pointer variable (**ROOTLP**) which is currently NULL

AddBSTNode (**ROOTLP**, 25) ;

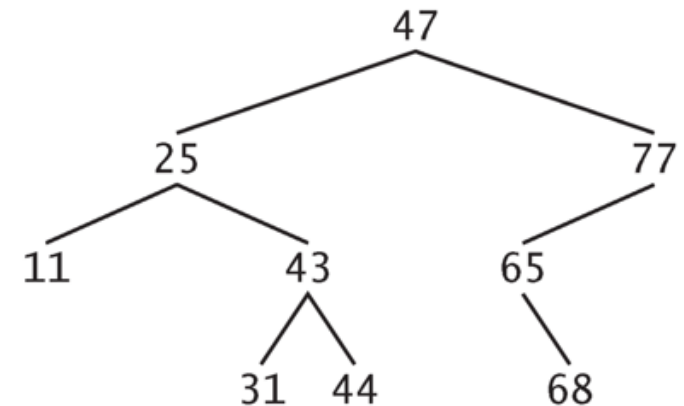


Variable
node_data;
right;
left;

```
void AddBSTNode(node **current_node, int add_data)
{
    else
    {
        if (add_data < (*current_node)->node_data )
            AddBSTNode(&(*current_node)->left, add_data);

        else if(add_data > (*current_node)->node_data )
            AddBSTNode(&(*current_node)->right, add_data);
    }
}
```

Memory Address
ND47 (47)
ROOTRP (NULL)
ROOTLP (NULL)



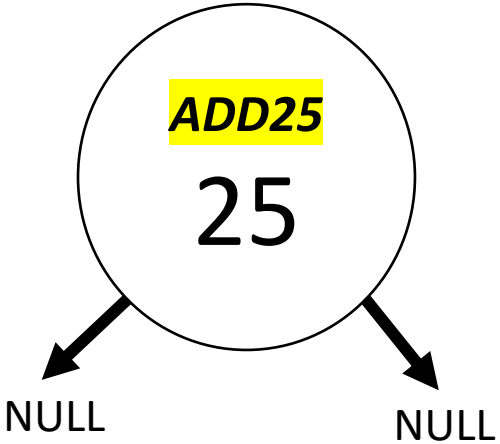

```
AddBSTNode ( ROOTLP, 25) ;
```

current_node is NULL

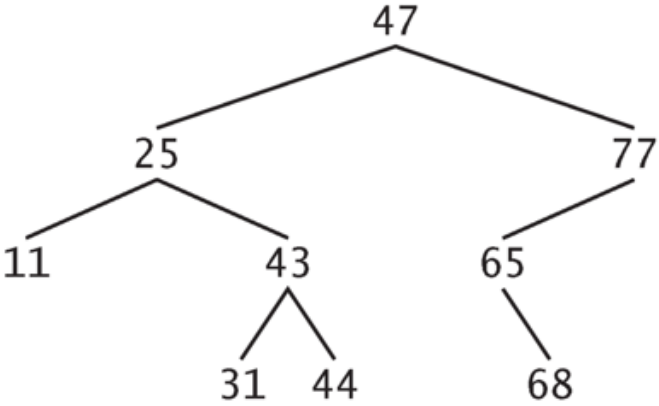
malloc a new node (address **ADD25**)
set left and right pointers to NULL
set node_data to 25

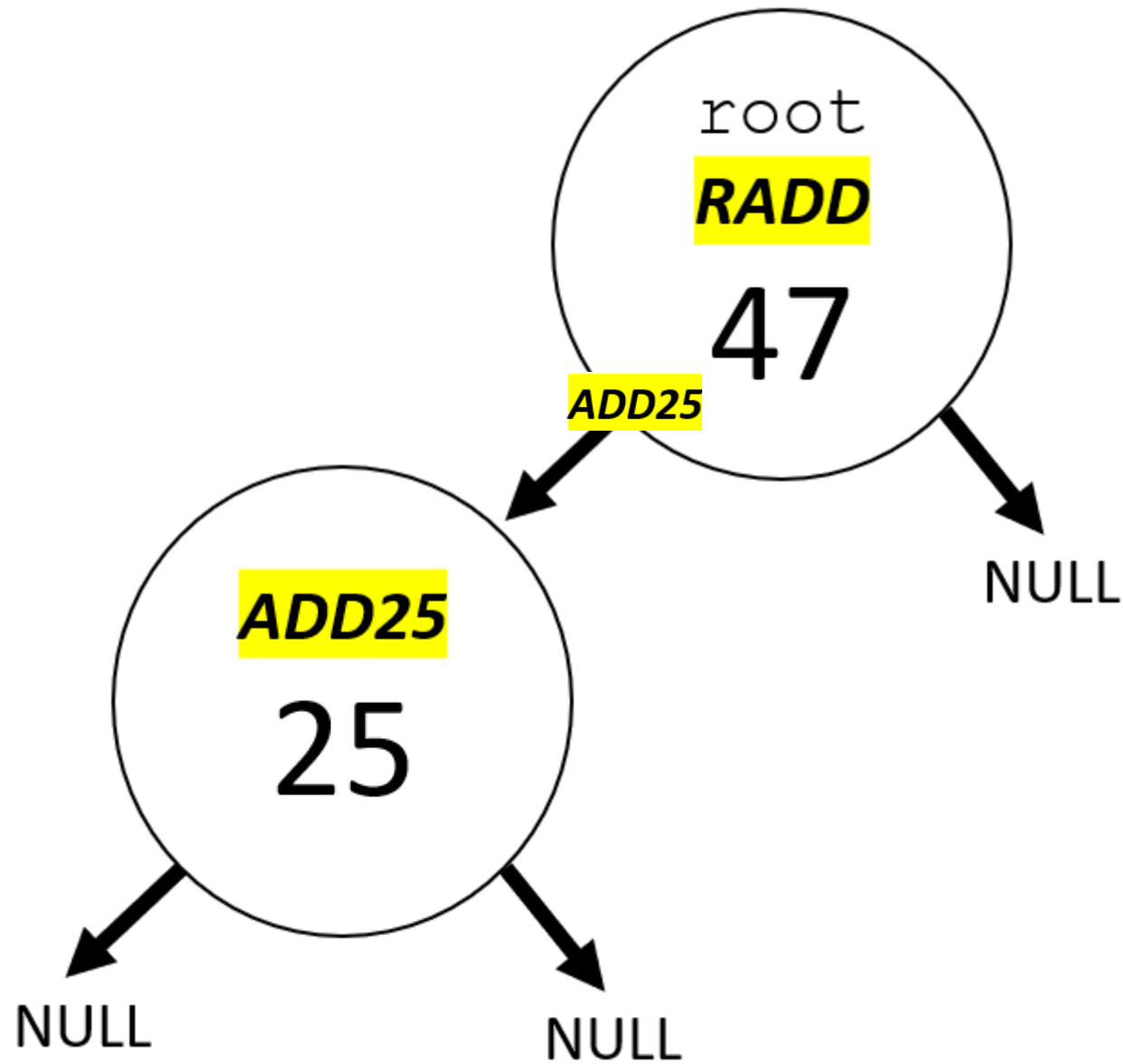
ADD25 is stored in current_node/**ROOTLP**

```
void AddBSTNode (node **current_node, int add_data)
{
    if (*current_node == NULL)
    {
        *current_node = malloc (sizeof (node));
        (*current_node) -> left = (*current_node) -> right = NULL;
        (*current_node) -> node_data = add_data;
    }
}
```



Variable	Memory Address
node_data;	ND25 (25)
right;	RP25 (NULL)
left;	LP25 (NULL)





Variable
node_data;
right;
left;

Memory Address
ND47 (47)
ROOTRP (NULL)
ROOTLP (ADD25)

root is set to **RADD**

AddBSTNode (**RADD**, 77) ;

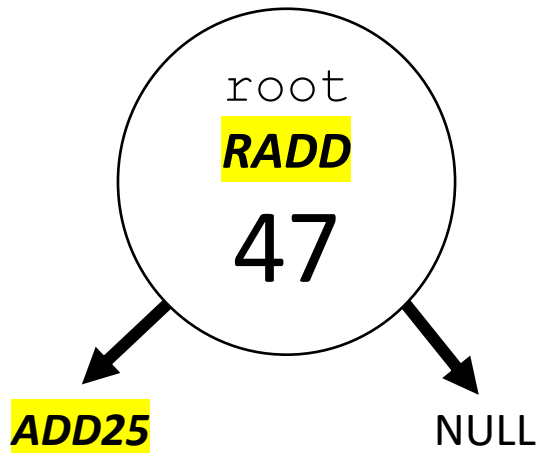
current_node is **not** NULL so else

add_data is 77 and node_data is 47

$77 > 47$

so call AddBSTNode () with the address of current node's (**RADD**) right pointer variable (**ROOTRP**) which is currently NULL

AddBSTNode (**ROOTRP**, 77) ;

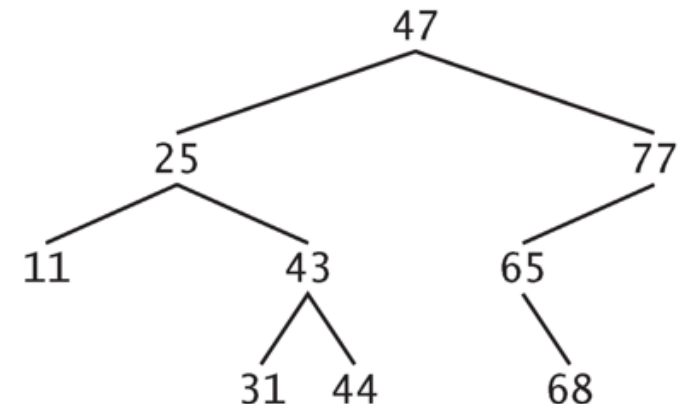


Variable
node_data;
right;
left;

```
void AddBSTNode(node **current_node, int add_data)
{
    else
    {
        if (add_data < (*current_node)->node_data )
            AddBSTNode(&(*current_node)->left, add_data);

        else if(add_data > (*current_node)->node_data )
            AddBSTNode(&(*current_node)->right, add_data);
    }
}
```

Memory Address
ND47 (47)
ROOTRP (NULL)
ROOTLP (ADD25)



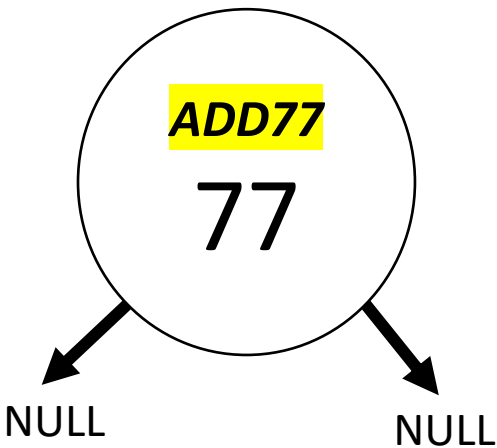
```
AddBSTNode ( ROOTRP, 77 ) ;
```

current_node is NULL

malloc a new node (address **ADD77**)
set left and right pointers to NULL
set node_data to 77

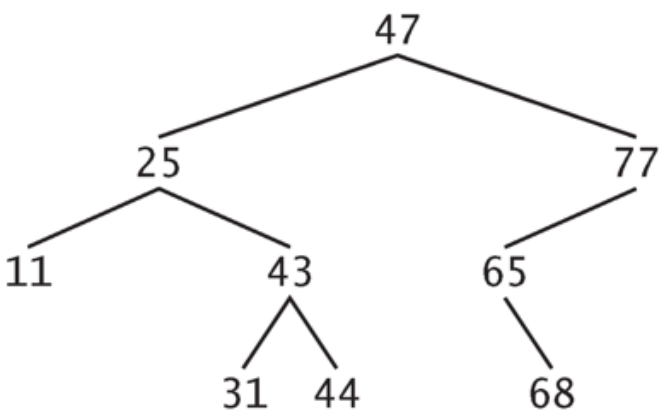
ADD77 is stored in current_node/**ROOTRP**

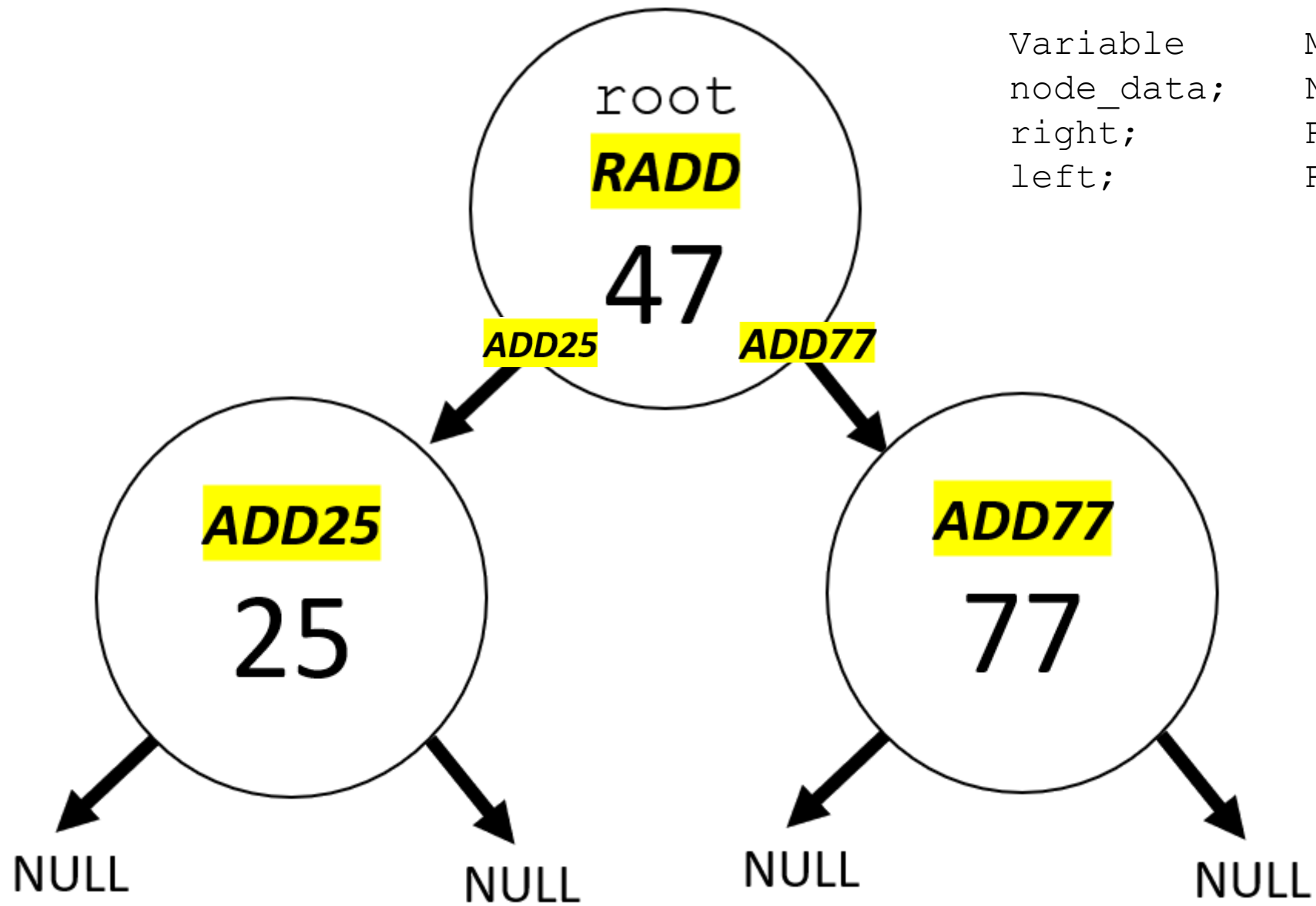
```
void AddBSTNode(node **current_node, int add_data)
{
    if (*current_node == NULL)
    {
        *current_node = malloc(sizeof(node));
        (*current_node)->left = (*current_node)->right = NULL;
        (*current_node)->node_data = add_data;
    }
}
```



Variable
node_data;
right;
left;

Memory Address
ND77 (77)
RP77 (NULL)
LP77 (NULL)





Variable	Memory Address
node_data;	NDADD (47)
right;	ROOTRP (ADD77)
left;	ROOTLP (ADD25)

root is set to **RADD**

AddBSTNode (**RADD**, 11) ;

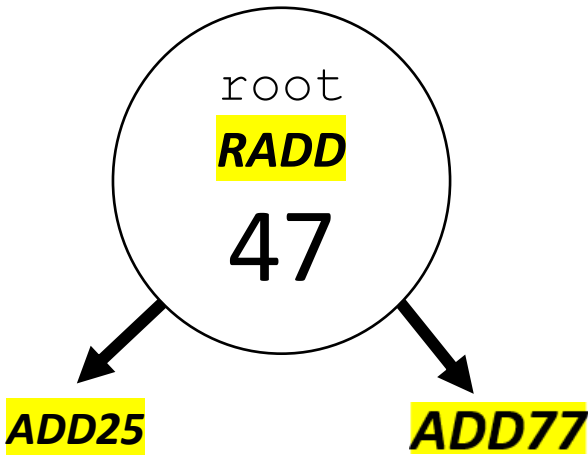
current_node is **not** NULL so else

add_data is 11 and node_data is 47

11 < 47

so call AddBSTNode () with the address of current node's (**RADD**) left pointer variable (**ROOTLP**) which is currently set to **ADD25**

AddBSTNode (**ROOTLP**, 11) ;

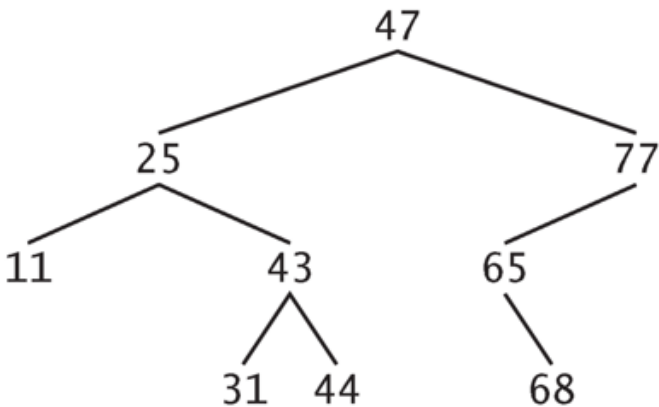


Variable
node_data;
right;
left;

Memory Address
ND47 (47)
ROOTRP (ADD77)
ROOTLP (ADD25)

```
void AddBSTNode(node **current_node, int add_data)
{
    else
    {
        if (add_data < (*current_node)->node_data )
            AddBSTNode(&(*current_node)->left, add_data);

        else if(add_data > (*current_node)->node_data )
            AddBSTNode(&(*current_node)->right, add_data);
    }
}
```



```
AddBSTNode (ADD25, 11);
```

current_node is **not** NULL so else

add_data is 11 and node_data is 25

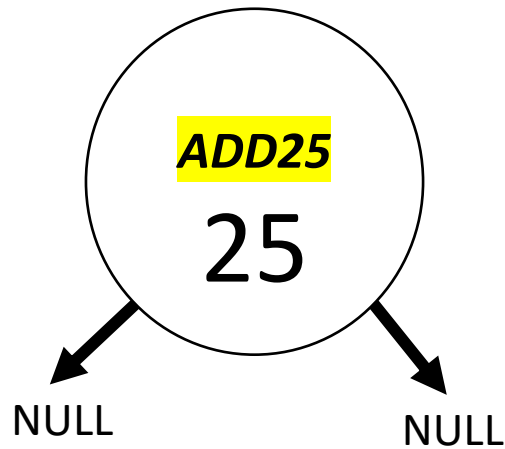
$11 < 25$

so call AddBSTNode () with the address of current node's (ADD25) left pointer variable (LP25) which is currently NULL

```
AddBSTNode (LP25, 11);
```

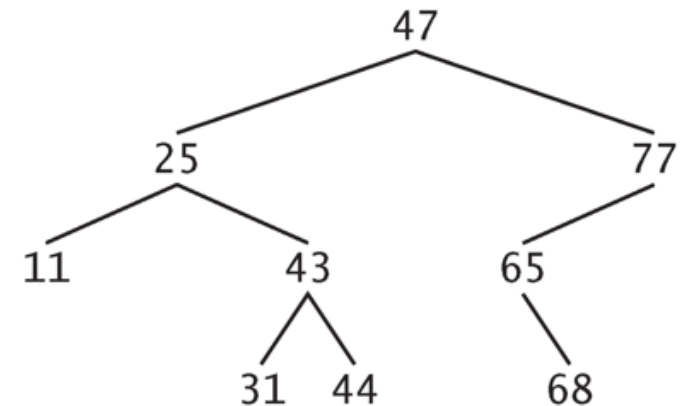
```
void AddBSTNode(node **current_node, int add_data)
{
    else
    {
        if (add_data < (*current_node)->node_data )
            AddBSTNode(&(*current_node)->left, add_data);

        else if(add_data > (*current_node)->node_data )
            AddBSTNode(&(*current_node)->right, add_data);
    }
}
```



Variable
node_data;
right;
left;

Memory Address
ND25 (25)
RP25 (NULL)
LP25 (NULL)



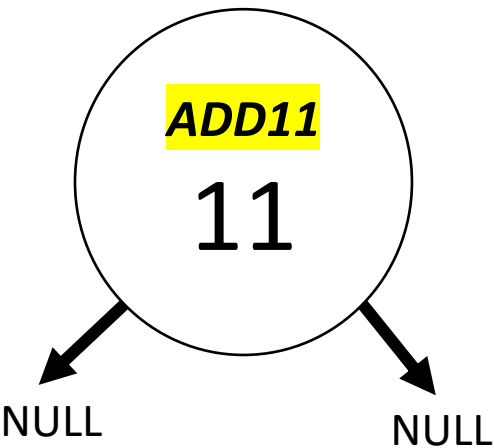
```
AddBSTNode(LP25, 11);
```

current_node is NULL

malloc a new node (address ADD11)
set left and right pointers to NULL
set node_data to 11

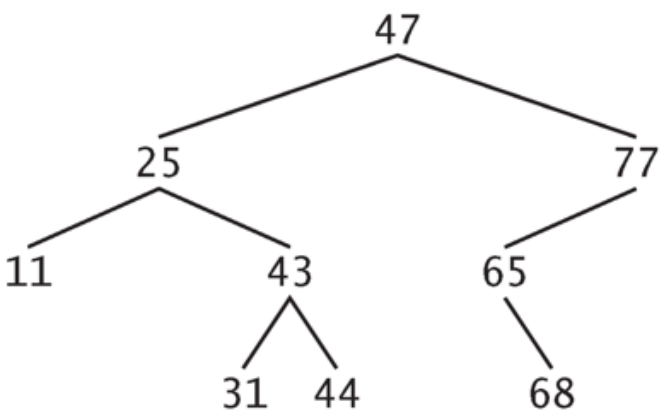
ADD11 is stored in current_node/LP25

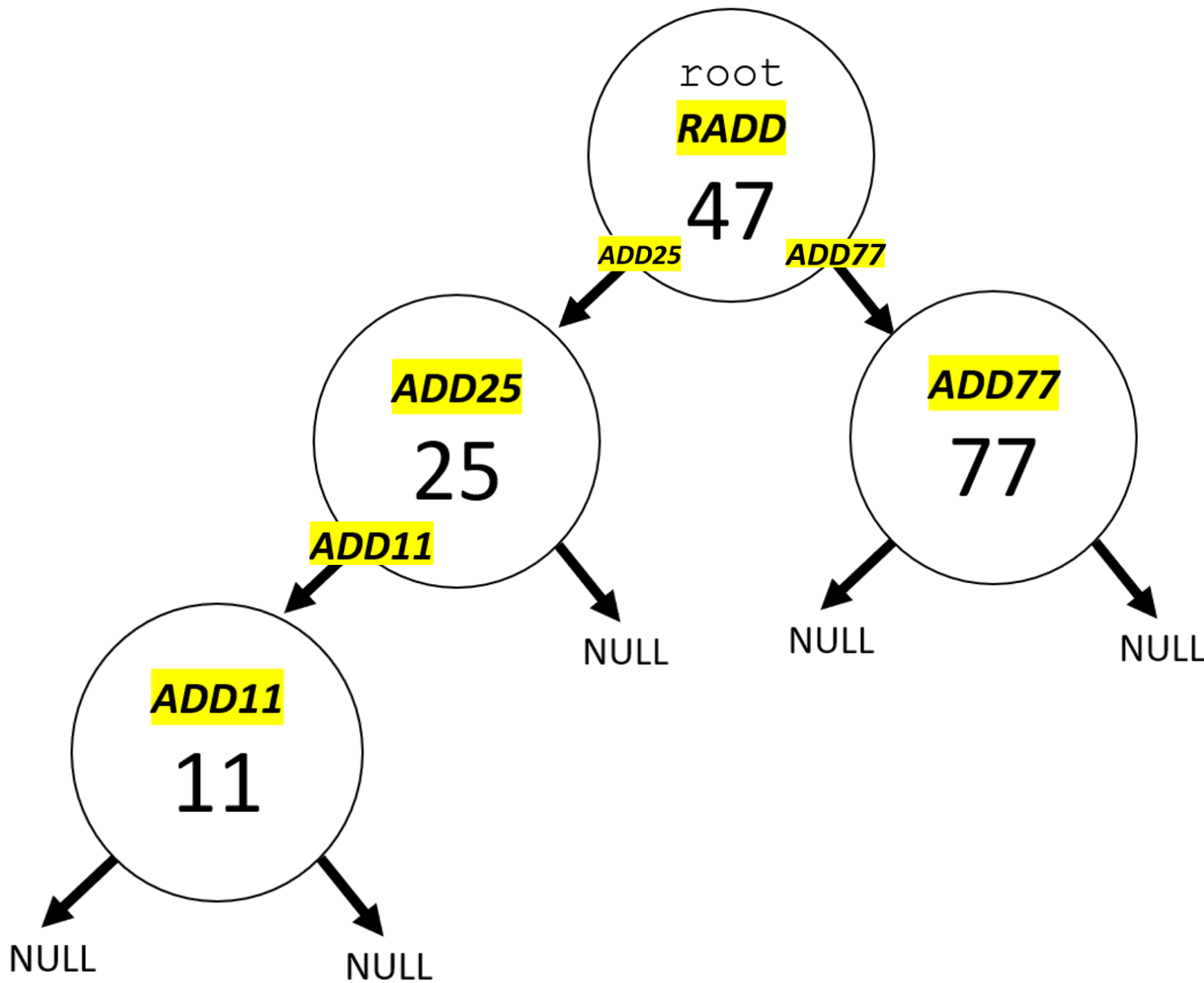
```
void AddBSTNode(node **current_node, int add_data)
{
    if (*current_node == NULL)
    {
        *current_node = malloc(sizeof(node));
        (*current_node)->left = (*current_node)->right = NULL;
        (*current_node)->node_data = add_data;
    }
}
```



Variable
node_data;
right;
left;

Memory Address
ND11 (11)
RP11 (NULL)
LP11 (NULL)





Variable	Memory Address
node_data;	ND47 (47)
right;	ROOTRP (ADD77)
left;	ROOTLP (ADD25)

Variable	Memory Address
node_data;	ND77 (77)
right;	RP77 (NULL)
left;	LP77 (NULL)

Variable	Memory Address
node_data;	ND25 (25)
right;	RP25 (NULL)
left;	LP25 (ADD11)

Variable	Memory Address
node_data;	ND11 (11)
right;	RP11 (NULL)
left;	LP11 (NULL)

```
Inorder(root);  
  
Preorder(root);  
  
Postorder(root);
```

```
void Preorder(node *tree_node)  
{  
    if(tree_node != NULL)  
    {  
        printf("Node%d",  
               tree_node->node_data);  
        Preorder(tree_node->left);  
        Preorder(tree_node->right);  
    }  
}
```

```
void Inorder(node *tree_node)  
{  
    if(tree_node != NULL)  
    {  
        Inorder(tree_node->left);  
        printf("Node%d",  
               tree_node->node_data);  
        Inorder(tree_node->right);  
    }  
}
```

```
void Postorder(node *tree_node)  
{  
    if(tree_node != NULL)  
    {  
        Postorder(tree_node->left);  
        Postorder(tree_node->right);  
        printf("Node%d",  
               tree_node->node_data);  
    }  
}
```

Coding Assignment 7

For this assignment, I am providing multiple files in Canvas. Download all of them and understand what each one is for before starting on the assignment.

`Movie Theater Seat Map Files.zip` – contains the eight movie theater seat map files.

`Command Line Files.zip` – contains the file of theaters and their zip codes for the BST and the file of customers for the queue. These are the data input files required by the program on the command line.

`Libraries.zip` – contains 10 files – 5 .c files and 5.h files for the libraries this assignment uses. They are Movie Theater, linked list, queue, stack and Binary Search Tree (BST) libraries.

`Code7_outline.c` – the code outline that you are start with for this assignment. Rename this file to `Code7_XXXXXXXXXX.c` where `XXXXXXXXXX` is your student id.

Make sure that you understand what each file is for once you have read through this entire assignment. All of the files should be in the directory where you are compiling/running your code.

You should use `Code7_outline.c` to start this assignment. Expand your `makefile` as needed to compile/link the movie theater, linked list, stack, queue and BST libraries. You should be able to compile `Code7_outline.c` as is once you create the proper `makefile`. It will run but won't do anything, but it will compile. Make sure it does before making any changes.

You will be required to use the functions as defined in the provided library files. You must not alter these library files. Your versions will not be used for testing; therefore, if you alter them to make your code work, your code will fail when graded. You can alter `Code7_outline.c` as you want but your version must call the library functions with the same name, parameters and return types in order to remain compatible with the libraries. The only files you will submit are the `makefile` and the `Code7_XXXXXXXXXX.c` file.

Your code will be tested by compiling it with the original libraries and will be tested with other data files (more/fewer customers, more/fewer movie theaters, and different theater seat maps).

This assignment has several purposes

1. Use linked list, queue, stack and BST functions.
2. Build new code using given constraints
3. Practice reading and understanding existing code and how to use it without altering it

All of these skills will be needed to be successful on the upcoming FEQs.

Command Line Parameters

A `get_command_line_parameter()` function is in the provided outline which means this code will allow you to run the program like this...

```
./Code7_xxxxxxxxxx.e QUEUE=queue.txt ZIPFILE=zip.txt RECEIPTNUMBER=1
```

This allows us to run the program with the parameters in any order. Here's a description of each parameter. Do not alter this function – GTA testing will involve running the program with the parameters in different orders to verify that this function is in place and that you are not hardcoding the queue file as `argv[1]` for example.

RECEIPTNUMBER=

The starting value of this run's receipt numbers. Treat this value as a number.

QUEUE=

Name of file containing a list of customer names. The file containing the list of customers is the `queue.txt` file in the "Command Line Files.zip". The example file is

Bugs Bunny

Daffy Duck

Tweety

Elmer Fudd

Marvin the Martian

Sylvester

ZIPFILE=

Name of the file containing the name of the theater (max 40 characters), its zip code (max 5 characters), the name of the file containing the theater's seat map (max 100 characters) and the dimensions of the seat map. Make no assumptions about the format of the file name – use the name as-is – don't add or subtract anything – use it as shown in the file. This file will consist of one line of X's and O's (letter O).

```
AMC The Parks At Arlington 18|76015|file1.txt|10x9
AMC Dine-In Clearfork 8|76109|file2.txt|11x5
AMC EastChase 9|76120|file13.txt|10x10
AMC Palace 9|76102|myfile.it|5x4
Studio Movie Grill|76011|cat.dog|4x10
Movie Tavern at Green Oaks|76017|abc.def|6x7
Studio Movie Grill Arlington|76018|xxxx.zzz|9x9
```

Sample myfile.it

```
OOOOOOOOOOOOOOOOOOOOOOOOOO
```

Sample cat.dog

```
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
```

The file containing the list of movie theaters is the zip.txt file in the "Command Line Files.zip". The files listed inside the zip.txt file (file1.txt, file2.txt, cat.dog, etc...) are in the "Movie Theater Seat Map Files.zip" on Canvas.

Queue

Read in a file of names (`QUEUE=`). The file consists of a list of customer names. For example

```
Bugs Bunny
Daffy Duck
Tweety
Elmer Fudd
Marvin the Martian
Sylvester
```

Put the customer names in a queue. Each node of the queue will contain a `char` array of size 30 to hold the customer's name. Please note that the customer name may contain spaces (use `fgets`). If the file open fails, print a message and exit.

Binary Search Tree

Use the data from `ZIPFILE=` to create a binary search tree. Use the zip code as the value used to create the tree. Each node of the binary search tree will contain the name of the theater, the zip code, the name of the file containing the theater's seat map and the dimensions of the seat map. If the file open fails, print a message and exit.

Stack

Each time a customer is served (buys tickets), a receipt is created and added to a stack. A receipt (stack node) will contain the receipt number and a **linked list** head that points to the linked list containing each seat sold per customer. Seats will be inserted in order in the linked list so that when the receipts are printed, the seats print in order. If a customer buys C3, A1 and E7, then the linked list of seats in the stack node will be in A1, C3, E7 order. The receipt number should start with the value passed on the command line and be incremented each time a receipt is created. One receipt per customer.

Linked List

Each stack node (customer receipt) will contain a linked list head that will point to the linked list of tickets that customer purchased. A library function has been provided (`ReturnAndFreeLinkedListNode()`) that will print the tickets one at a time and will free each node of the linked list as it returns the ticket from the linked list. After all tickets have printed/been freed, the stack node can be popped/freed.

Process

Process the command line arguments.

Create a generic 2D array of size 26x20 which will be used to hold any of the seat maps. We will only use rows A-Z and seats 1-20. This will ensure we can display the seat map on the screen.

Read the `QUEUE=` file and create the queue of customers.

Read the `ZIPFILE=` and create the binary search tree.

Print the menu

1. Sell tickets to next customer
2. See who's in line
3. See the seat map for a given theater
4. Print today's receipts

Be sure to validate the choice from the menu.

Your program should stop running/gracefully complete when the line is empty. There is no option to quit before your job of selling tickets is complete.

Sell tickets to next customer

Dequeue the head of the queue. Print the customer's name and display the list of theater names and zip codes from the binary search tree (BST). Prompt the customer to enter the zip code of the theater from which they wish to purchase tickets. Using the zip code, search the BST for that zip code. Once found, extract the name of the file containing the seat map and the dimensions of the theater. Open the file containing the seat map and read it into the 2D array using the dimensions found in the BST node. If the data in the file exceeds the dimensions allowed by that particular theater, then display a message about an invalid seat map and prompt the customer to pick a different theater.

Display the seat map and prompt for how many tickets they would like to purchase. For each ticket, prompt the customer to pick a seat. Translate the input seat choice to a row-column pair. Validate that the row,column is inside the bounds of the theater. If the seat is invalid, then tell the customer, redisplay the seat map and prompt for a new choice for seat. Once the customer has entered a valid seat, check if the seat has been sold. If it has been sold, then inform the customer and prompt for a new seat choice. Revalidate the seat choice before checking if it is available. Customer should be able to select multiple invalid seats and sold seats but should still be able to buy the desired number of tickets.

After completing a sale to a customer, write the seat map back out to the file from which you read it. This will allow you to keep track of sold seats between customers. Create a receipt for the customer on the stack. The linked list of tickets should be formed as tickets are sold (you **cannot** store the sold tickets in an array and then write them to the stack node's linked list).

See who's in line

This option should traverse the queue and print each customer's name in order from head to tail to show who's in line.

See the seat map for a given theater

When this option is chosen, display the list of theater names and zip codes and allow the input of a zipcode. Find that zip code in the BST, read the file containing the seat map into your generic array and display it. This functionality should mostly be a reuse of the code from selling tickets.

Print today's receipts

When this option is chosen, the contents of the stack should be displayed by popping each node from the stack and displaying its contents. Once the receipts have been displayed, the stack should be have completely emptied and a rerun should state that today's receipts have already been displayed. The receipt number should be displayed along with the tickets that were sold on that receipt. The tickets should have been stored in the linked list in the stack node in order; therefore, they should print in order when the receipts are displayed.

Testing

Your code will be tested two ways.

1. A file of my choice will be used for the customer list and the theater zip list. Your program will need to use both of those files correctly – sell tickets and display the line of customers and print the day's receipts.
2. The GTA's will compile your Code7.c file with MY versions of MovieTheater.c/.h, LinkLib.c/h, BSTLib.c/h, QueueLib.c/h and StackLib.c/h. You should not force your Code7.c to work by changing something in these files. You can change what you want in the Code7_outline.c file as long as your changes still work with the provided files.

Miscellaneous

1. The linked list head, queue head, queue tail, binary search tree root and stack head are declared in `main()` and passed as needed to other functions. No globals allowed.