CSE 1320

Week of 04/15/2019

Instructor : Donna French

Using Recursion

The programs we've discussed are generally structured as functions that call one another in a disciplined, hierarchical manner.

For some types of problems, it's useful to have functions call themselves.

A recursive function is a function that calls itself either directly or indirectly through another function.

Recursion is a complex topic discussed at length in upper-level computer science courses.

Using Recursion

Recursion occurs when a function or subprogram calls itself or calls a function which in turn calls the original function.

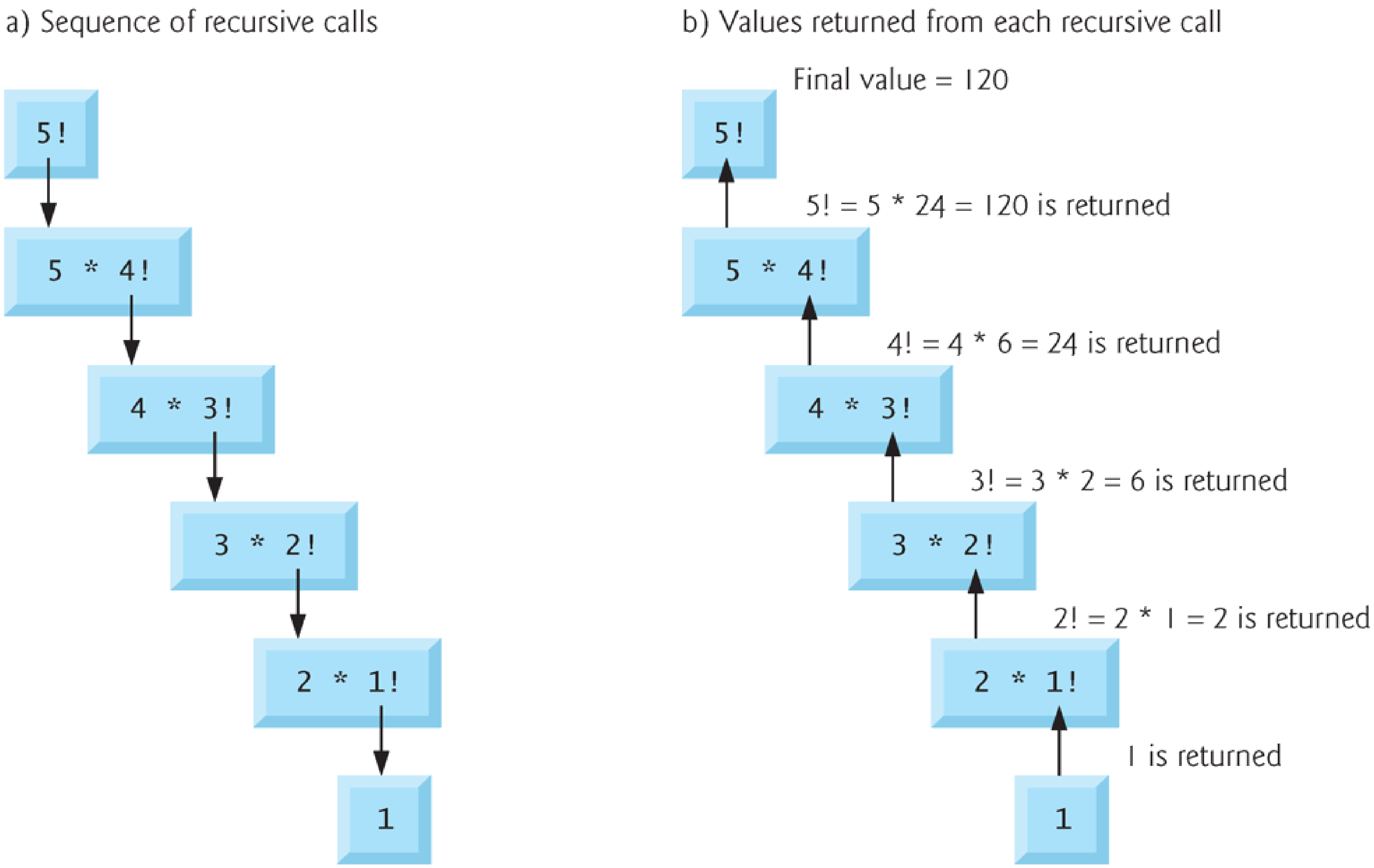
A simple example of a mathematical recursion is factorial

```
1! = 1
2! = 2*1 = 2
3! = 3*2*1 = 6
4! = 4*3*2*1 = 24
5! = 5*4*3*2*1 = 120
```

```
n! = n * (n - 1)!
```

Using Recursion

```
n! = n * (n - 1)!

int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return (n * factorial(n - 1));
}
```



Recursive evaluation of 5!

```c
int main(void)
{
	int input, output;

	printf("Enter an input for the factorial ");
	scanf("%d", &input);

	output = factorial(input);

	printf("The result of %d! is %d\n\n", input, output);

	return 0;
}

int factorial(int n)
{
	if (n == 0)
		return 1;
	else
		return (n * factorial(n - 1));
}
```

```
Enter an input for the factorial 4
The result of 4! is 24
```

```
int factorial(int n)
{
   if (n == 0)
      return 1;
   else
      return (n * factorial(n - 1));
}
```

```
Enter 4

Calls factorial with 4

factorial(4)
    if 0, then return 1 else return (4 * factorial(4-1)
                                     return (4 * 6)
        factorial(3)
            if 0, then return 1 else return (3 * factorial(3-1)
                                             return (3 * 2)
                factorial(2)
                    if 0, then return 1 else return (2 * factorial(2-1)
                                                     return (2 * 1)
                        factorial(1)
                            if 0, then return 1 else return (1 * factorial(1-1)
                                                             return (1 * 1)
                                factorial(0)
                                    if 0, then return 1 else return (0 * factorial(0-1)
                                                  return 1
```

4! = 4 * 3* 2 * 1 = 24

Using Recursion

A function's execution environment includes local variables and parameters and other information like a pointer to the memory containing the global variables.

This execution environment is created every time a function is called.

Recursive functions can use a lot of memory quickly since a new execution environment is created each time the recursive function is called.

```
(gdb) bt
#0  factorial (n=0) at frDemo.c:6
#1  0x00000000004004fd in factorial (n=1) at frDemo.c:9
#2  0x00000000004004fd in factorial (n=2) at frDemo.c:9
#3  0x00000000004004fd in factorial (n=3) at frDemo.c:9
#4  0x00000000004004fd in factorial (n=4) at frDemo.c:9
#5  0x000000000040053d in main () at frDemo.c:19
```

After processing `n=0`

```
#0  0x00000000004004fd in factorial (n=1) at frDemo.c:9
#1  0x00000000004004fd in factorial (n=2) at frDemo.c:9
#2  0x00000000004004fd in factorial (n=3) at frDemo.c:9
#3  0x00000000004004fd in factorial (n=4) at frDemo.c:9
#4  0x000000000040053d in main () at frDemo.c:19
```

After processing `n=1`

```
#0  0x00000000004004fd in factorial (n=2) at frDemo.c:9
#1  0x00000000004004fd in factorial (n=3) at frDemo.c:9
#2  0x00000000004004fd in factorial (n=4) at frDemo.c:9
#3  0x000000000040053d in main () at frDemo.c:19
```

After processing `n=2`

```
#0  0x00000000004004fd in factorial (n=3) at frDemo.c:9
#1  0x00000000004004fd in factorial (n=4) at frDemo.c:9
#2  0x000000000040053d in main () at frDemo.c:19
```

After processing `n=3`

```
#0  0x00000000004004fd in factorial (n=4) at frDemo.c:9
#1  0x000000000040053d in main () at frDemo.c:19
```

After processing `n=4`

```
#0  0x000000000040053d in main () at frDemo.c:19
```

Recursive Program to Sum Range of Natural Numbers

```
int main(void)
{
   int num;

   printf("Enter a positive integer: ");
   scanf("%d", &num);

   printf("Sum of all natural numbers from %d to 1 = %d\n",
           num, addNumbers(num));

   return 0;
}
```

Recursive Program to Sum Range of Natural Numbers

```
int addNumbers(int n)
{
   if (n != 0)
   {
      return n + addNumbers(n-1);
   }
   else
   {
      return n;
   }
}
```

Example Using Recursion: Fibonacci Series

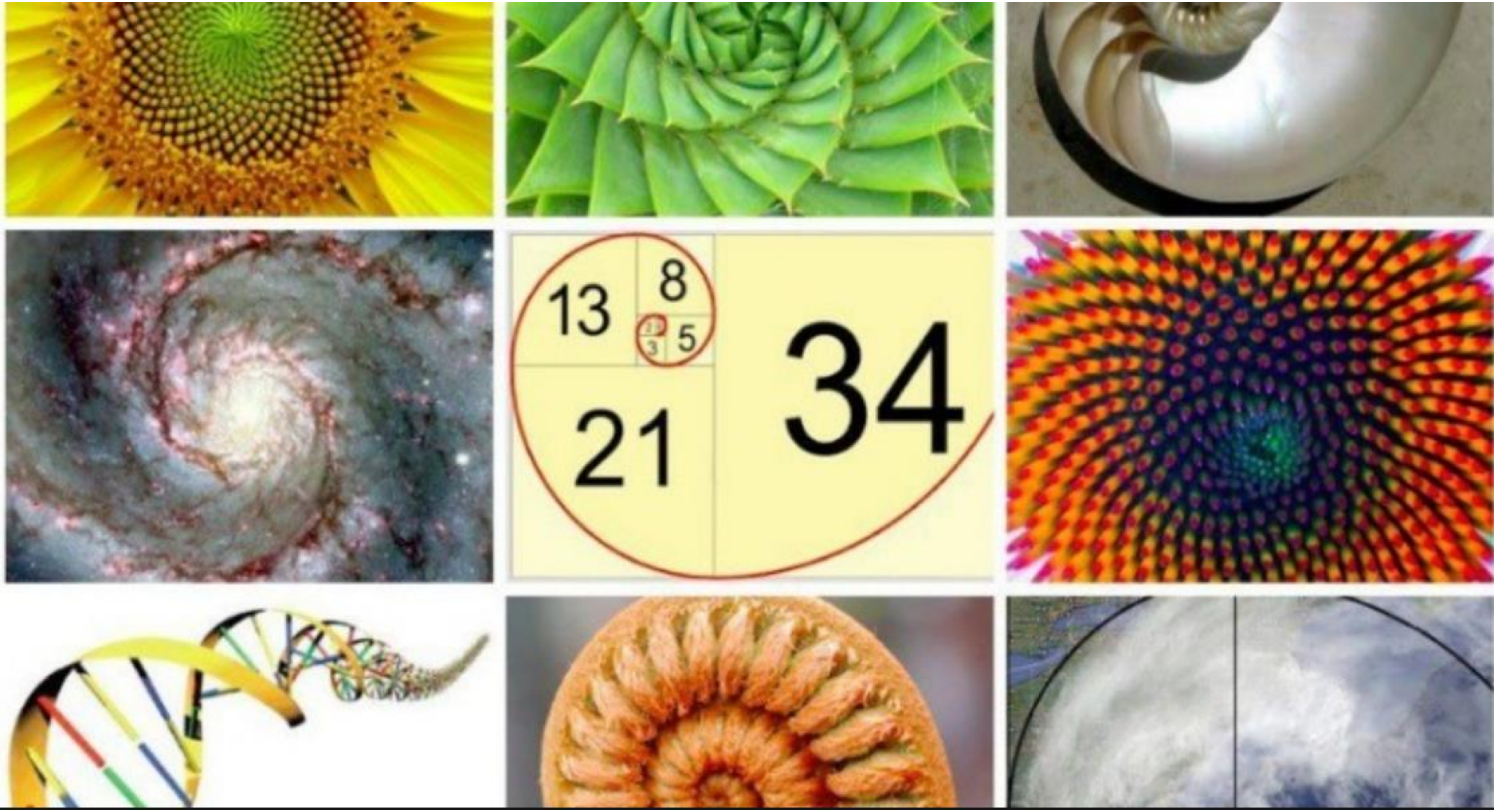
The Fibonacci series

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….


13
8
3
5
34
21

Example Using Recursion: Fibonacci Series

The Fibonacci series may be defined recursively as follows:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(6) = 8

fibonacci(*n*) = *fibonacci(n – 1) + fibonacci(n – 2)*

Example Using Recursion: Fibonacci Series

The Fibonacci series may be defined recursively as follows:

fibonacci(n) = *fibonacci(n – 1) + fibonacci(n – 2)*

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

```c
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.

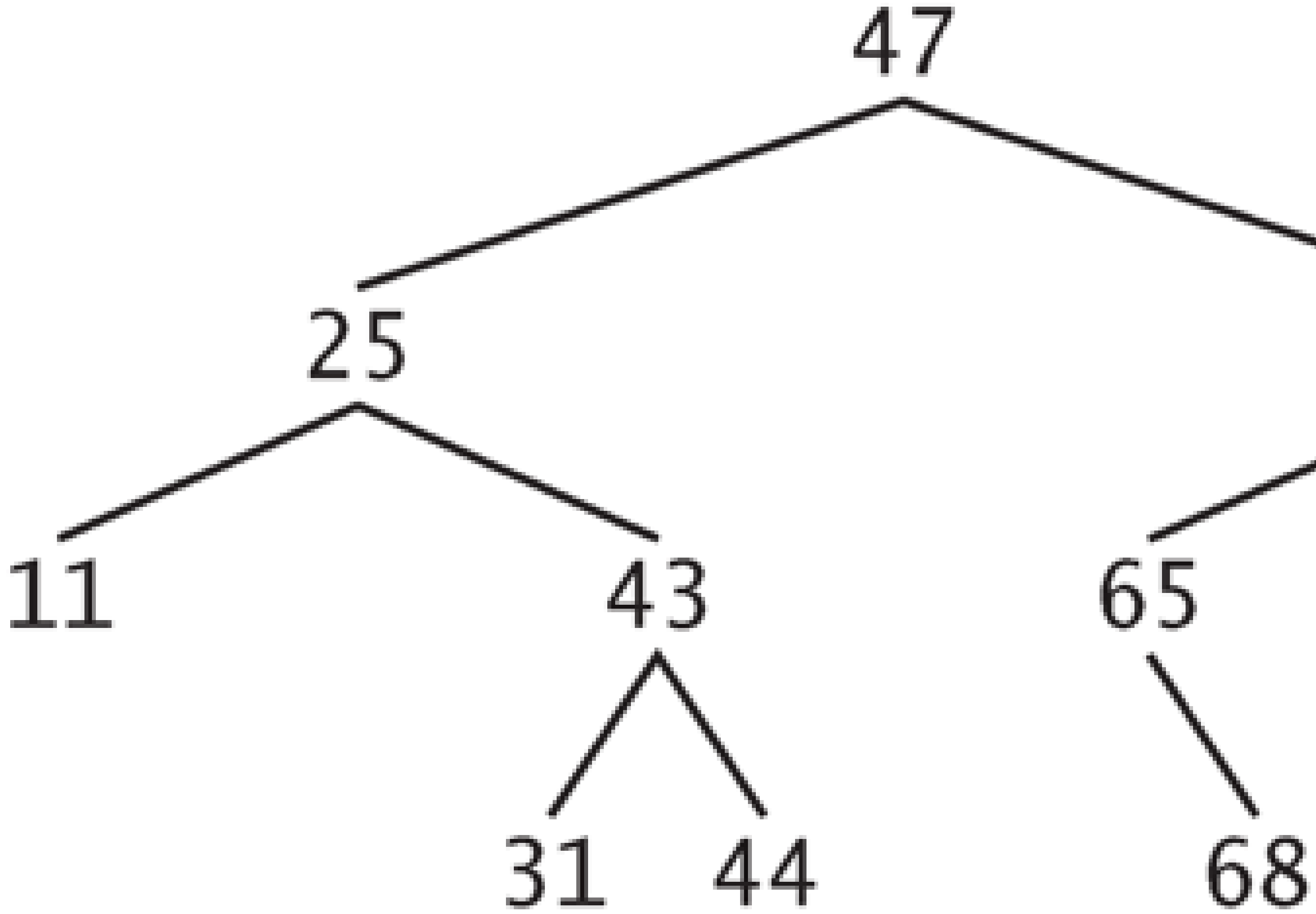
A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug.

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

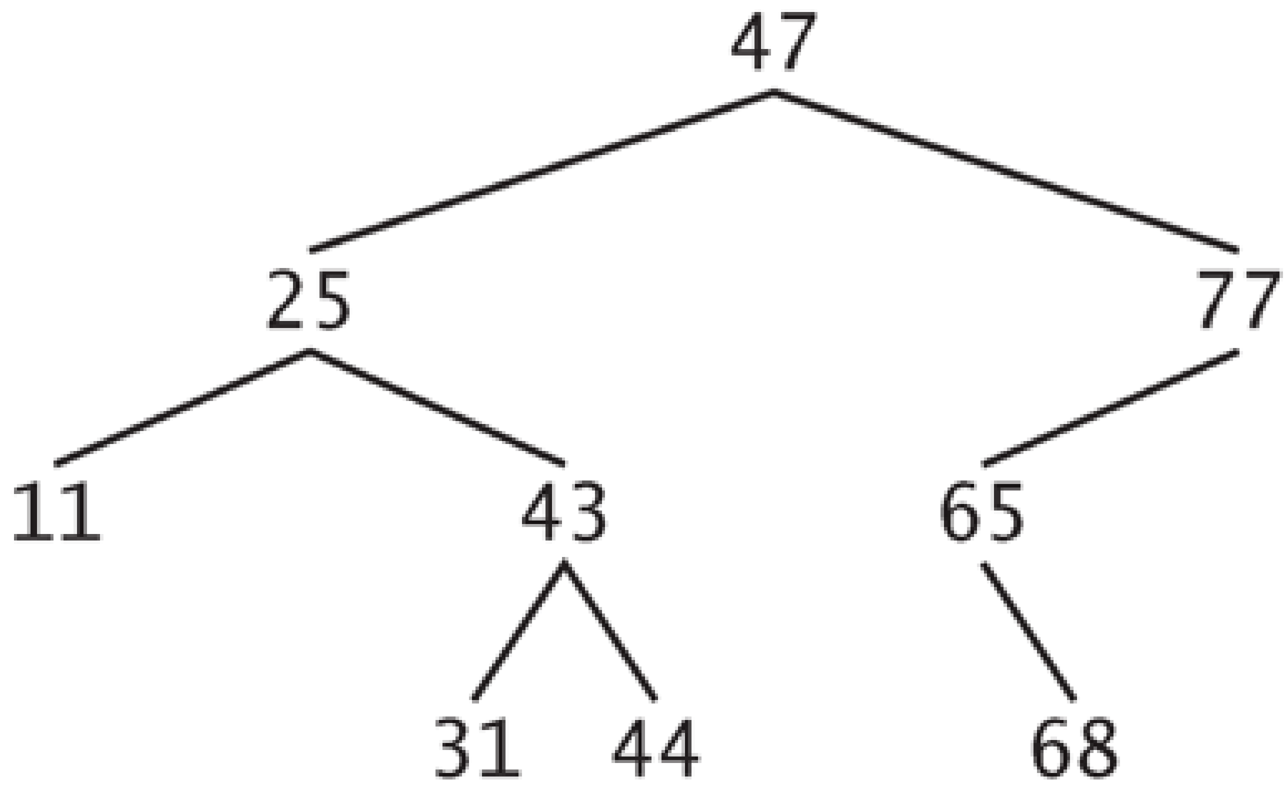
Binary Search Tree

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.


47
25
77
11
43
65
31
44
68

```
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
```



BST Traversal in Inorder

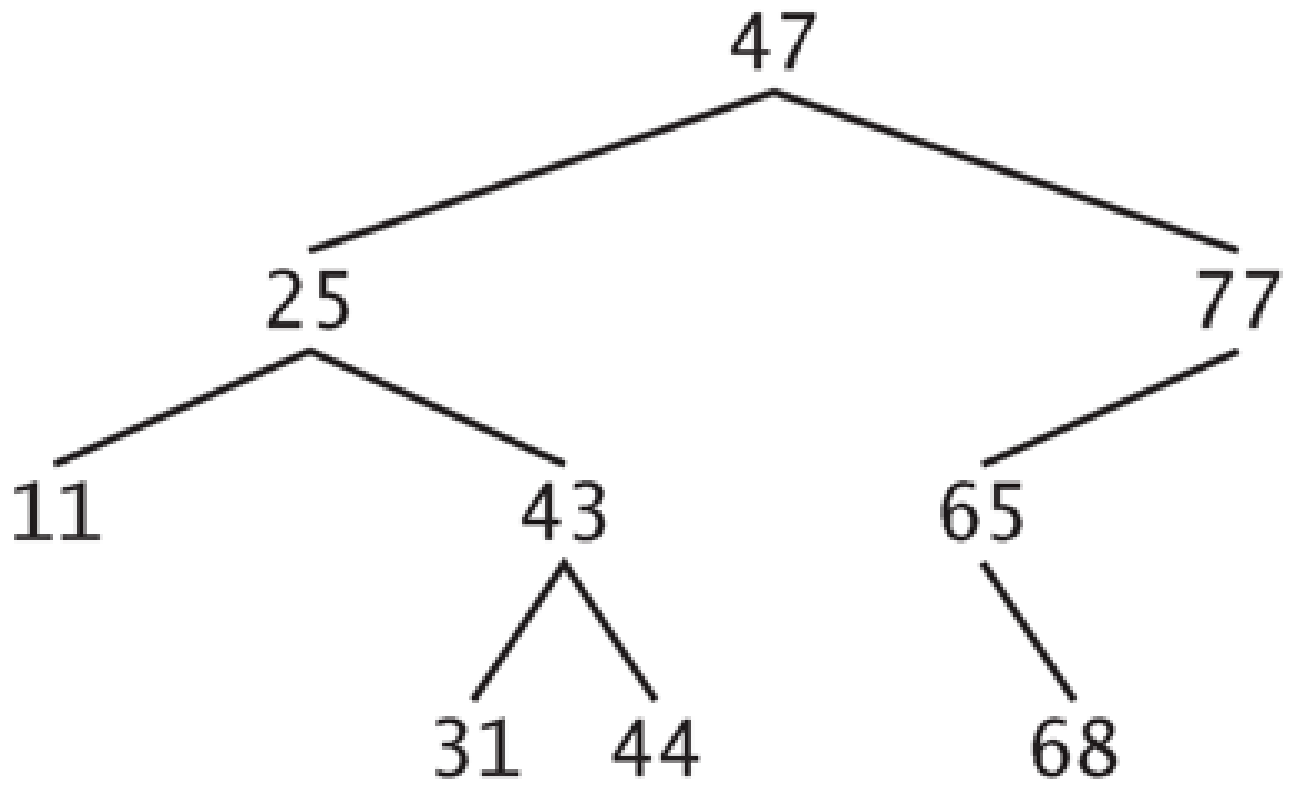
```
Node4-11  Node2-25  Node7-31
Node5-43  Node8-44  Node1-47
Node6-65  Node9-68  Node3-77
```

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
```



```c
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 = (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 !!!");
    }
}
```

```
Inorder(root);

Preorder(root);

Postorder(root);
```

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

Traversing a Linked List

```
void DisplayLinkedList(node *LinkedListHead)
{
    node *TempPtr = LinkedListHead;

    while (TempPtr != NULL)
    {
        printf("\nNode Number %d\n", TempPtr->node_number);
        TempPtr = TempPtr->next_ptr;
    }
}
```

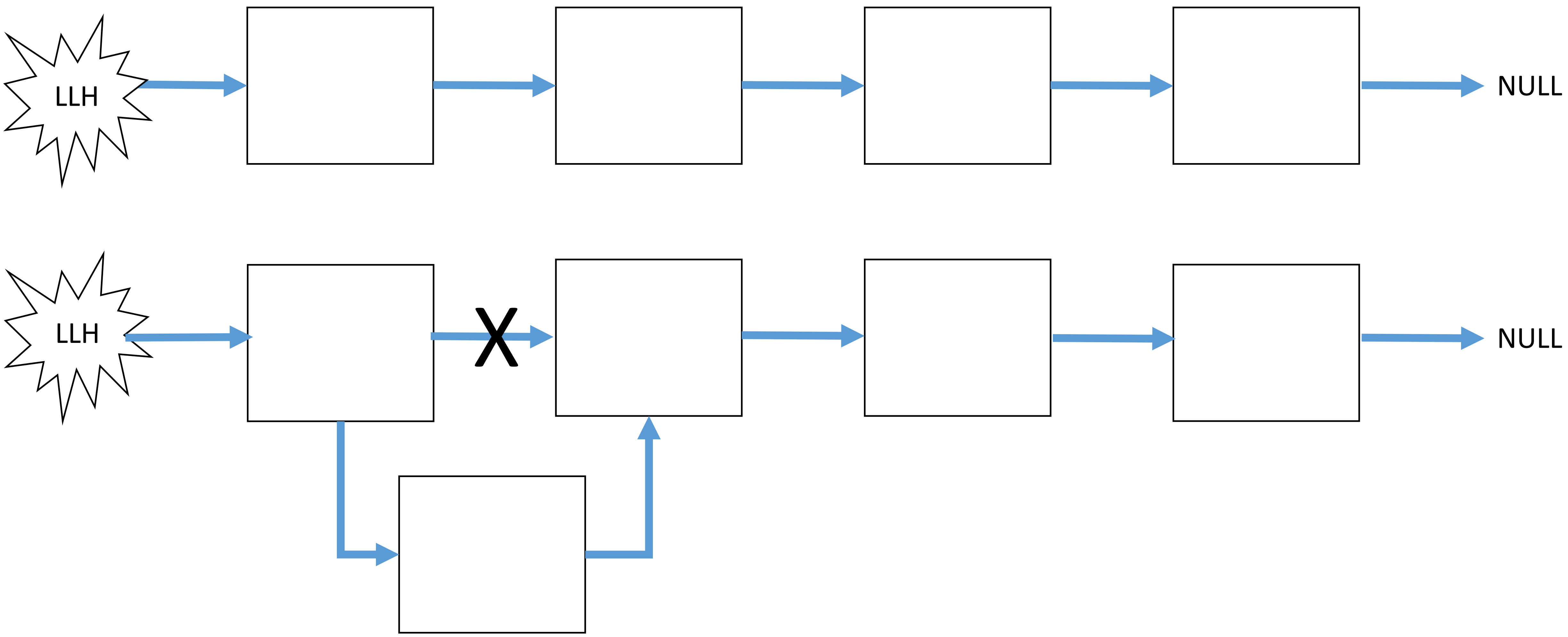
Traversing a Queue

```
void DisplayQueue(node *QueueHead)
{
    node *TempPtr = QueueHead;

    while (TempPtr != NULL)
    {
        printf("Queue node %d\n", TempPtr->node_number);
                TempPtr =  TempPtr->next_ptr;
    }
}
```

Traversing a Stack

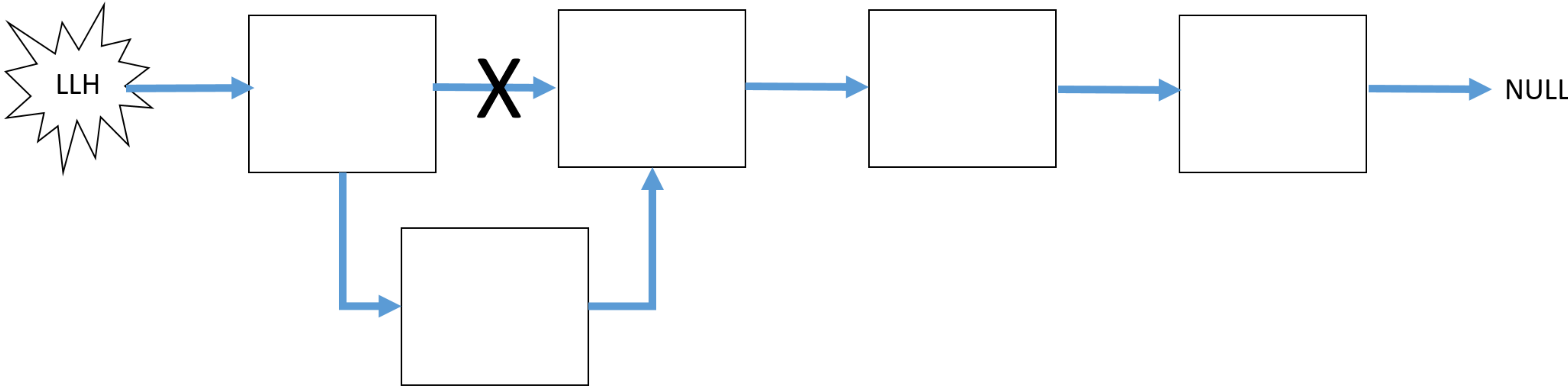
```
void DisplayStack(node *StackTop)
{
    node *TempPtr = StackTop;

    while (TempPtr != NULL)
    {
        printf("Stack node %d\n", TempPtr->node_number);
                TempPtr =  TempPtr->next_ptr;
    }
}
```

Inserting a Node into Linked List

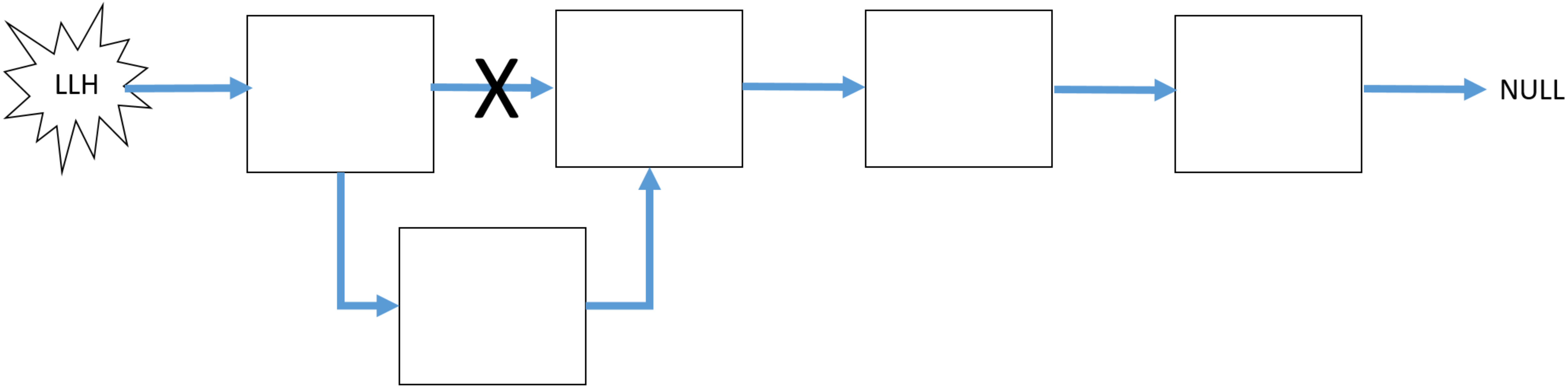


ICQ15

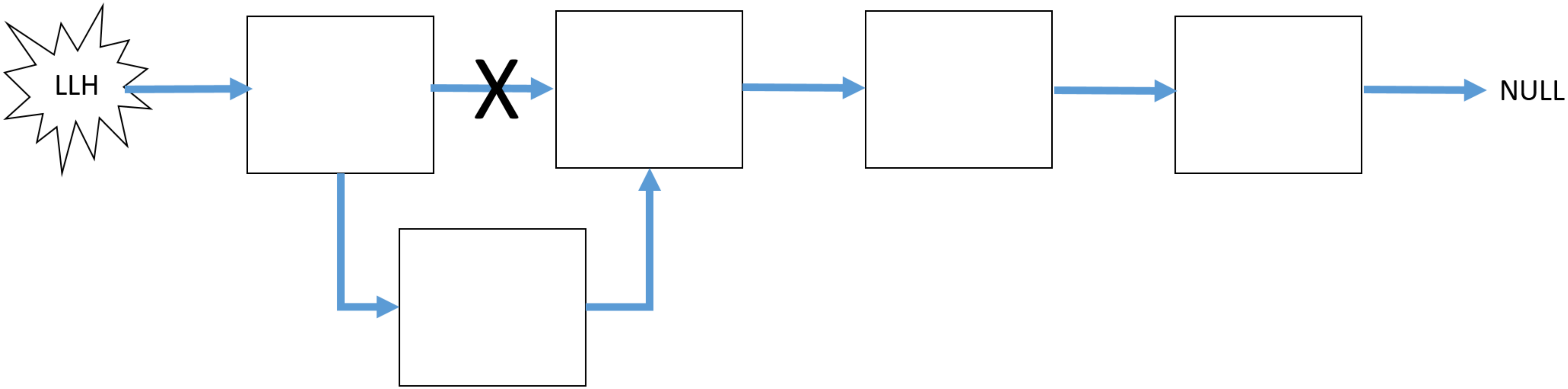
```
void InsertNode(int NodeNumberToInsert, node **LinkedListHead)
{
    node *TempPtr, *PrevPtr, *NewNode;

    PrevPtr = NULL;
    TempPtr = *LinkedListHead;

    while (TempPtr != NULL && TempPtr->node_number < NodeNumberToInsert)
    {
        PrevPtr = TempPtr;
        TempPtr = TempPtr->next_ptr;
    }

    NewNode = malloc(sizeof(node));
    NewNode->node_number = NodeNumberToInsert;
    NewNode->next_ptr = TempPtr;

    if (PrevPtr == NULL)
    {
        *LinkedListHead = NewNode;
    }
    else
    {
        PrevPtr->next_ptr = NewNode;
    }
}
```



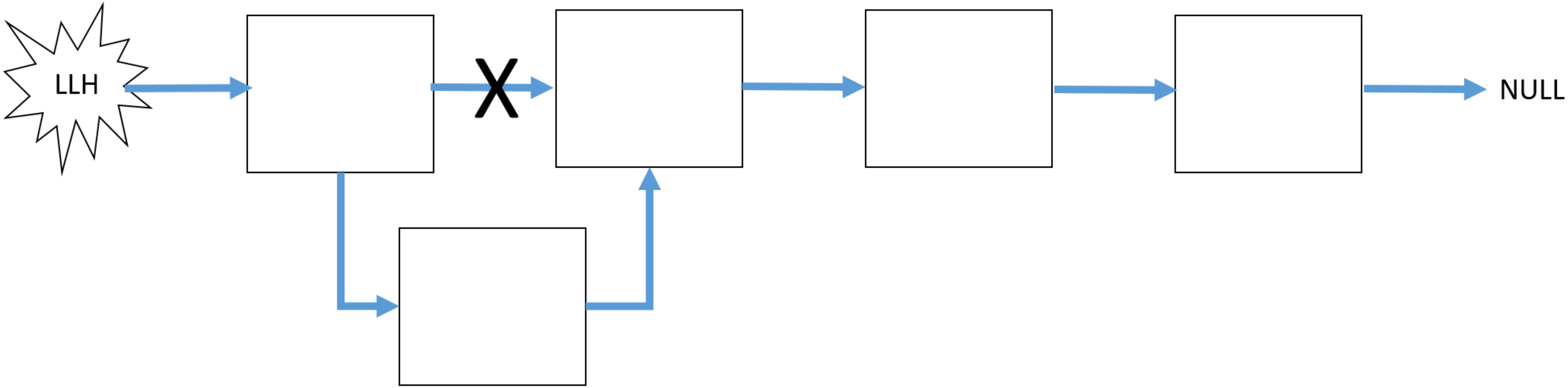

```
void InsertNode(int NodeNumberToInsert, node **LinkedListHead)
{
    node *TempPtr, *PrevPtr, *NewNode;

    PrevPtr = NULL;
    TempPtr = *LinkedListHead;
```




```
while (TempPtr != NULL && TempPtr->node_number < NodeNumberToInsert)
{
    PrevPtr = TempPtr;
    TempPtr = TempPtr->next_ptr;
}
```

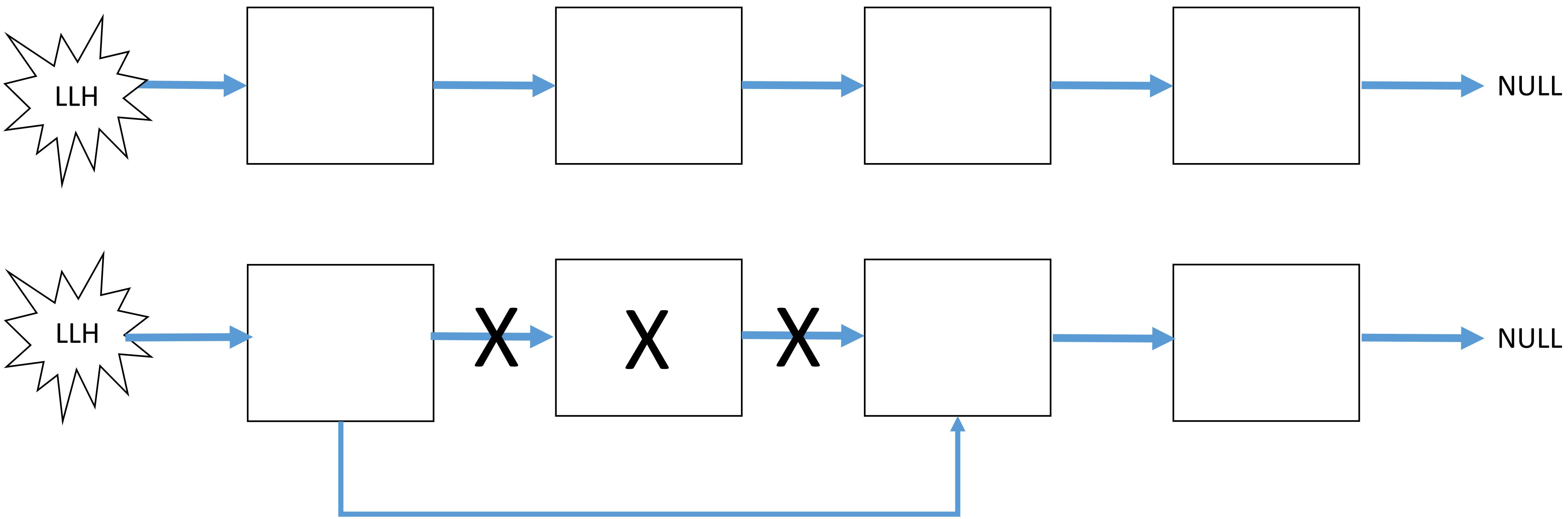



```
NewNode = malloc(sizeof(node));
NewNode->node_number = NodeNumberToInsert;
NewNode->next_ptr = TempPtr;
```




```
if (PrevPtr == NULL)
{
    *LinkedListHead = NewNode;
}
else
{
    PrevPtr->next_ptr = NewNode;
}
```

Deleting a Node from a Linked List



```
void DeleteNode(int NumberOfNodeToDelete, node **LinkedListHead)
{
    node *TempPtr, *PreviousNode;

    TempPtr = *LinkedListHead;

    while (TempPtr != NULL)
    {
        if (TempPtr->node_number == NumberOfNodeToDelete)
        {
            if (TempPtr == *LinkedListHead
            {
                *LinkedListHead = TempPtr->next_ptr;
            }
            else
            {
                PreviousNode->next_ptr = TempPtr->next_ptr;
            }
            free(TempPtr);
        }
        else
        {
            PreviousNode = TempPtr;
            TempPtr = TempPtr->next_ptr;
        }
    }
}
```

ICQ14

```
void AddBSTNode(node **current_node, int add_data)
{
    if (*current_node == NULL)
    {
        *current_node = (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 !!!");
    }
}
```

Action Items

Coding Assignment 6
DUE: APR 18, 2019
Coding Assignments

Crash Course : Quiz 10
DUE: APR 15, 2019
Crash Course

Homework 9
DUE: APR 15, 2019
Homework