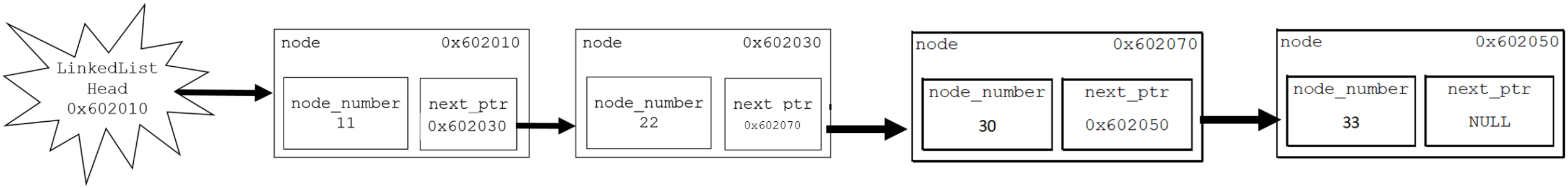
CSE 1320

Week of 04/10/2023

Instructor : Donna French

Linked List – Display the linked list

```
struct node *TempPtr;
TempPtr = LinkedListHead;

/* Traverse the linked list and display the node number */
while (TempPtr != NULL)
{
   printf("\nNode Number %d\t\tNode Address %p     Node Next Pointer %p\n",
          TempPtr->node_number, TempPtr, TempPtr->next_ptr);
   TempPtr = TempPtr->next_ptr;
}
```

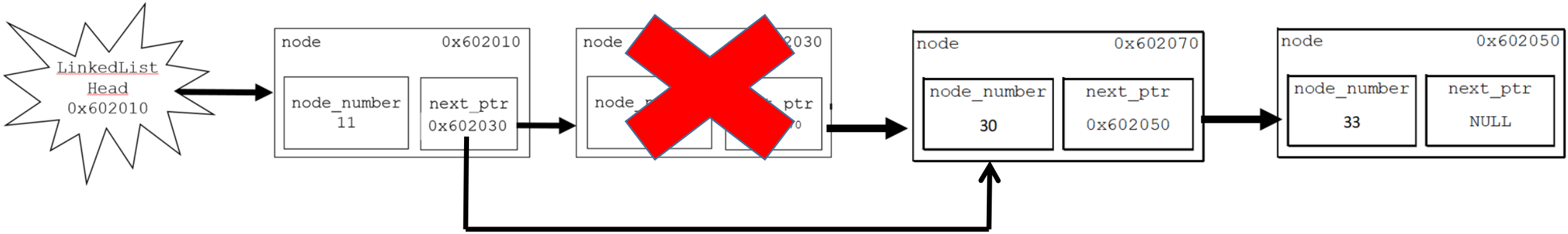


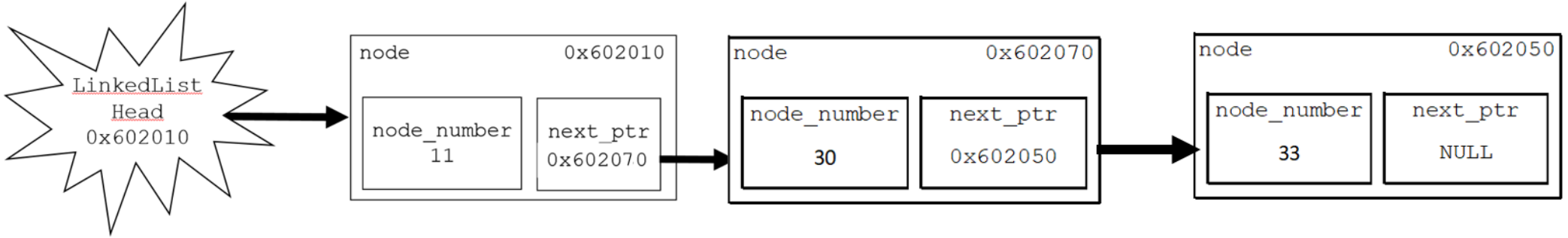
Linked List – Count Nodes

```
struct node *TempPtr;
int node_count = 0;

TempPtr = LinkedListHead;

/* Traverse the list until finding the node pointing at NULL */
while (TempPtr != NULL)
{
   TempPtr = TempPtr->next_ptr;
   node_count++;
}
```

Linked List – Delete node

Deleting a node





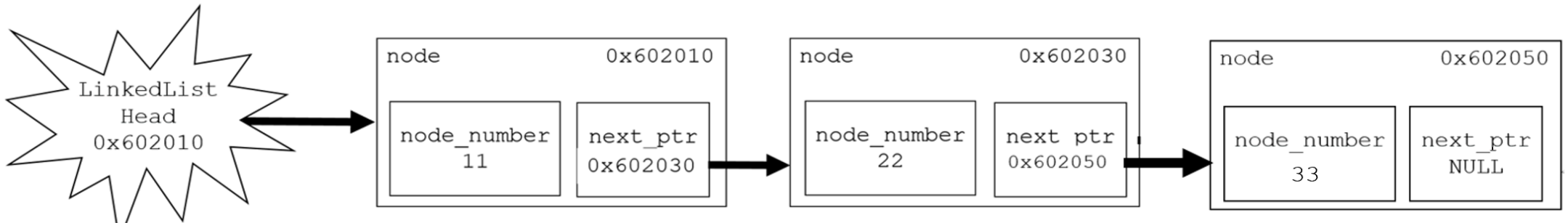
```
Node Number 11          Node Address 0x602010     Node Next Pointer 0x602030

Node Number 22          Node Address 0x602030     Node Next Pointer 0x602070

Node Number 30          Node Address 0x602070     Node Next Pointer 0x602050

Node Number 33          Node Address 0x602050     Node Next Pointer (nil)

Linked List Menu
====================
1. Insert a node
2. Display all nodes
3. Count the nodes
4. Delete a node
5. Add node to start
6. Add node to end
7. Exit

Enter your choice : 4
```

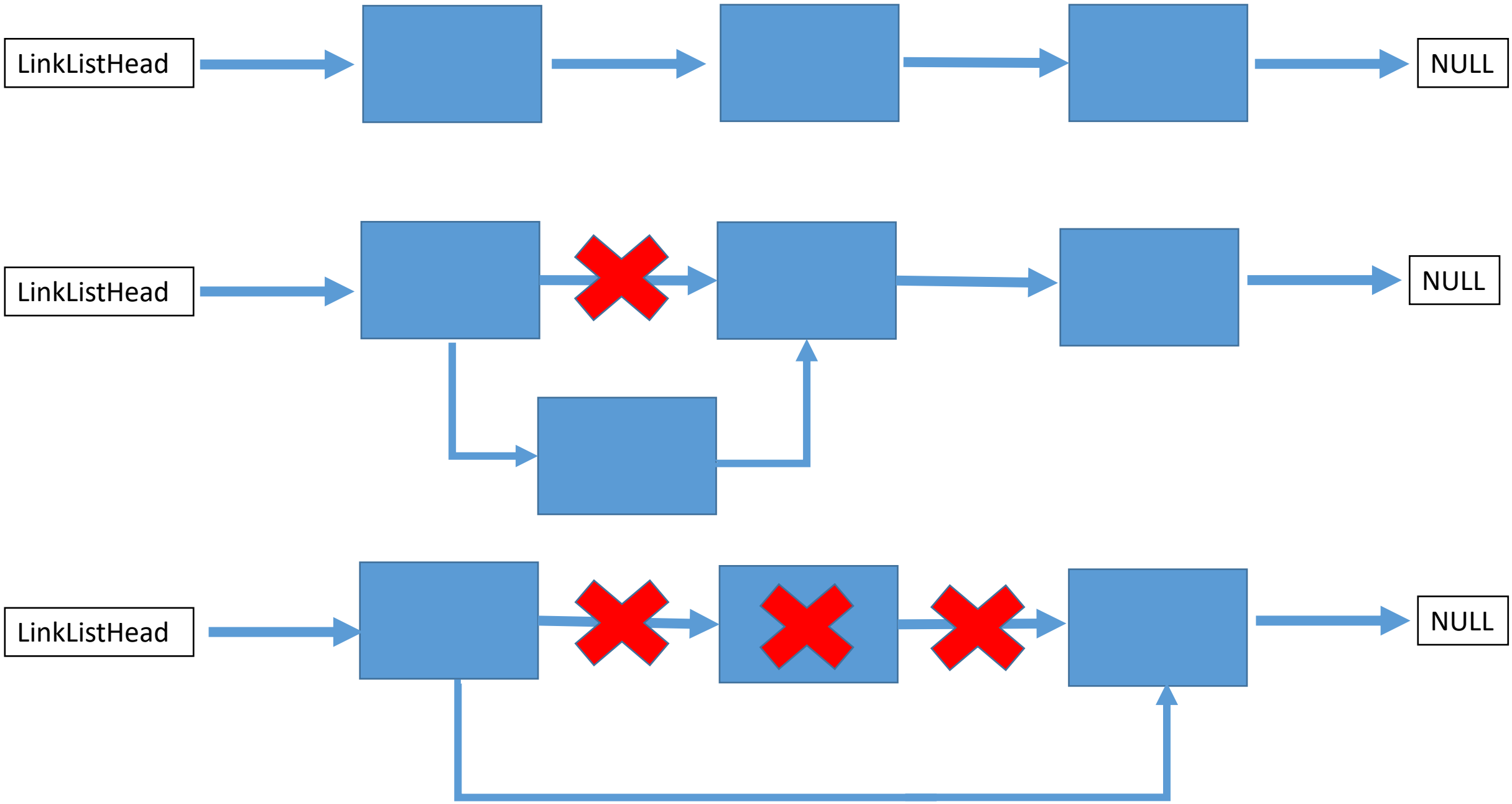
```
Enter your choice : 4

Enter the node number to delete : 22

Node 22 was successfully deleted


Node Number 11          Node Address 0x602010     Node Next Pointer 0x602070

Node Number 30          Node Address 0x602070     Node Next Pointer 0x602050

Node Number 33          Node Address 0x602050     Node Next Pointer (nil)
```

```
TempPtr = LinkedListHead;  PrevPtr = NULL;

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

if (TempPtr->node_number == NumberOfNodeToDelete)
{
    if (TempPtr == LinkedListHead)
    {
        LinkedListHead = TempPtr->next_ptr;
    }
    else
    {
        PrevPtr->next_ptr = TempPtr->next_ptr;
    }
    free(TempPtr);
}
else
{
    printf
}
```

If the head address is the node to delete

Change address stored in the head




LinkListHead
NULL
LinkListHead
NULL
LinkListHead
NULL

```
void AddNode(int NewNodeNumber, node **LinkedListHead)
{
   node *TempPtr, *NewNode;

   NewNode = malloc(sizeof(node));
   NewNode->node_number = NewNodeNumber;
   NewNode->next_ptr = NULL;

   if (*LinkedListHead == NULL)
   {
      *LinkedListHead = NewNode;
   }
   else
   {
      TempPtr = *LinkedListHead;

      while (TempPtr->next_ptr != NULL)
         TempPtr = TempPtr->next_ptr;

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

When calling a function to add a node, we need to pass the address of the linked list head.

`LinkedListHead` is a pointer so when we need to pass it to a function that will UPDATE it, then we need to pass the address of `LinkedListHead`.

```
AddNode(11, &LinkedListHead);
```

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

node *LinkedListHead = NULL;
```

Normally, we would not have the struct's tag at the top.

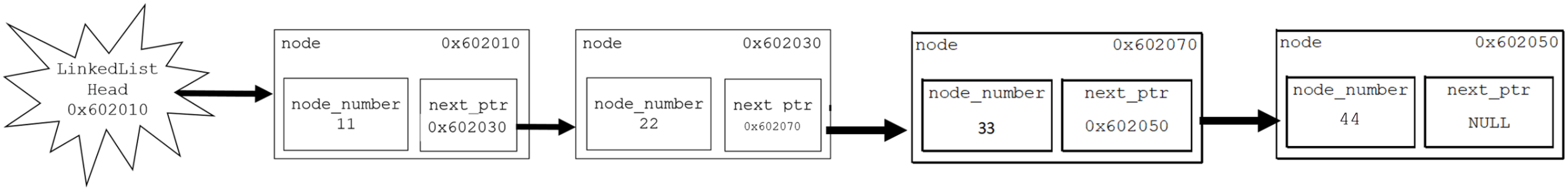
When using a pointer to the struct inside the struct, we have to include the tag at the top so that the compiler knows the struct's name before getting to the pointer declaration.

When calling a function like `DisplayLinkedList`, we need to pass the head of the linked list but we do not need to pass the address of it since this function will not change it.

```
DisplayLinkedList(LinkedListHead);
```

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

    while (TempPtr != NULL)
    {
       printf("\nNode Number %d\t\tNode Address %p     Node Next Pointer %p\n",
                 TempPtr->node_number, TempPtr, TempPtr->next_ptr);
       TempPtr = TempPtr->next_ptr;
    }
}
```

What if we used `TempPtr->next_ptr != NULL`?



```
Node Number 11          Node Address 0x1d1e010     Node Next Pointer 0x1d1e030

Node Number 22          Node Address 0x1d1e030     Node Next Pointer 0x1d1e050

Node Number 33          Node Address 0x1d1e050     Node Next Pointer 0x1d1e070

Node Number 44          Node Address 0x1d1e070     Node Next Pointer (nil)
```

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++;
}
```

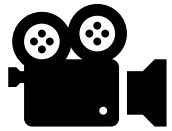
```
1 Tacos|Flamin' Hot Doritos Locos Taco|Seasoned Beef, Cheese, Lettuce
2 Tacos|Flamin' Hot Doritos Locos Tacos Supreme|Seasoned Beef, Cheese, Lettuce, Tomatoes, Sour Cream
3 Burritos|Loaded Taco Grande Burrito|Seasoned Beef, Cheese, Lettuce, Red Strips, Sour Cream
```

```
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);
}
```

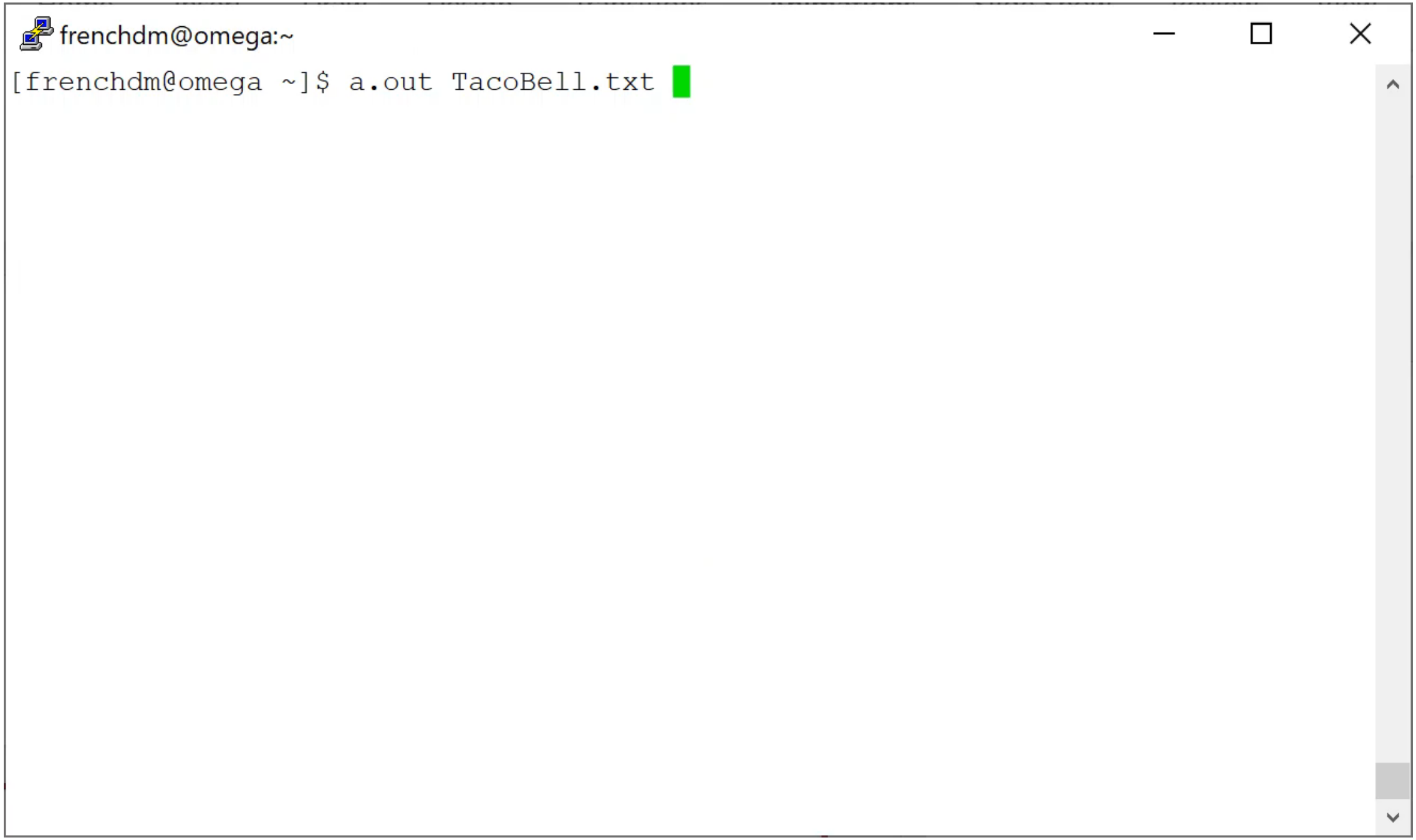
```
1 Tacos|Flamin' Hot Doritos Locos Taco|Seasoned Beef, Cheese, Lettuce
2 Tacos|Flamin' Hot Doritos Locos Tacos Supreme|Seasoned Beef, Cheese, Lettuce, Tomatoes, Sour Cream
3 Burritos|Loaded Taco Grande Burrito|Seasoned Beef, Cheese, Lettuce, Red Strips, Sour Cream
```

```
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
```



```c
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);
}

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);
}
```

```
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
```


frenchdm@omega:~
[frenchdm@omega ~]$ a.out TacoBell.txt

Stack

Stack is a linear data structure which follows a particular order in which the operations are performed. The order will be LIFO (Last In First Out).

Stack

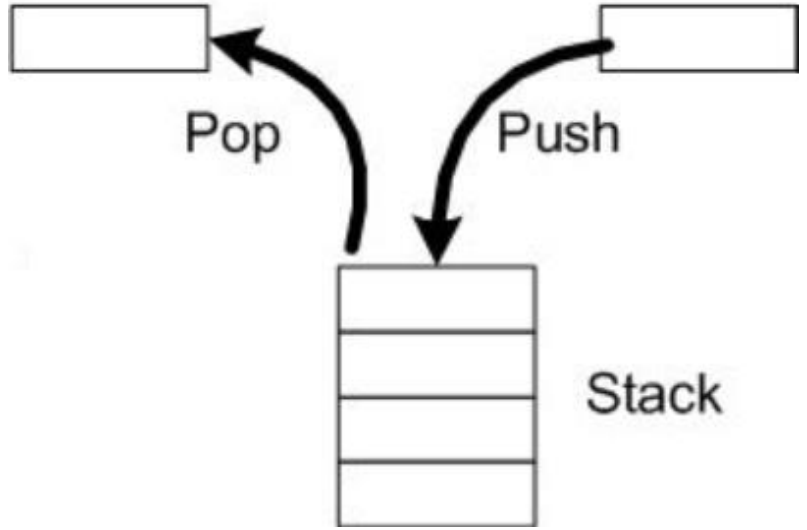
Operations on a Stack

Push Adds an item to the stack.

Pop Removes an item from the stack.

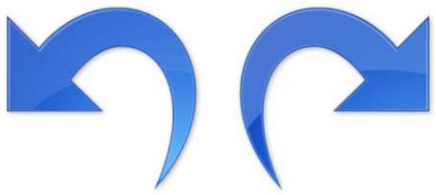
Peek Returns top element of stack.

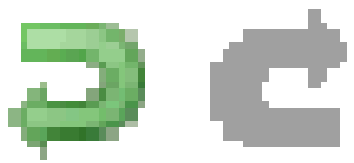
IsEmpty Returns TRUE if stack is empty, else FALSE.

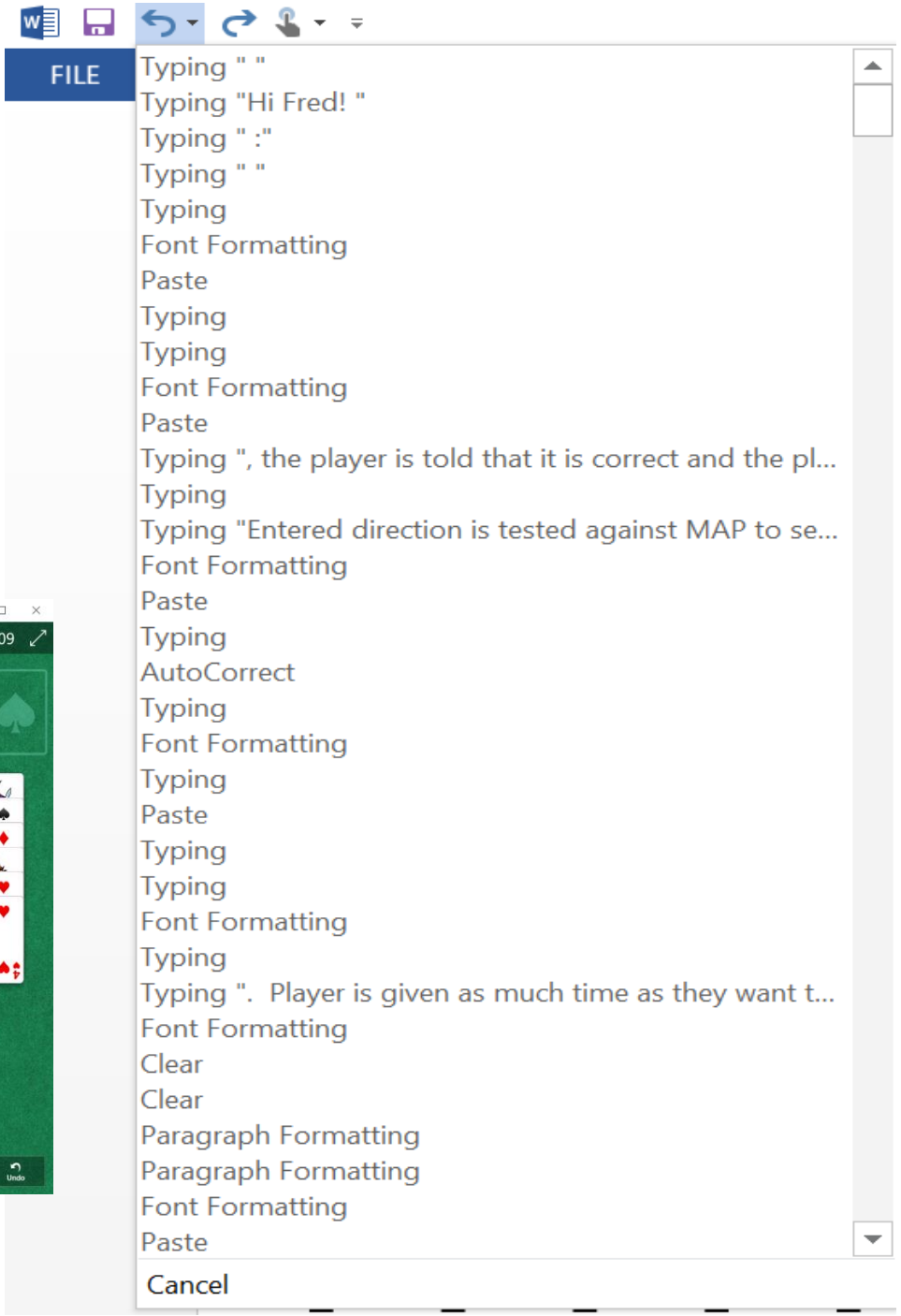


Stack

Word processors, text editors, some games use undo and redo capabilities.





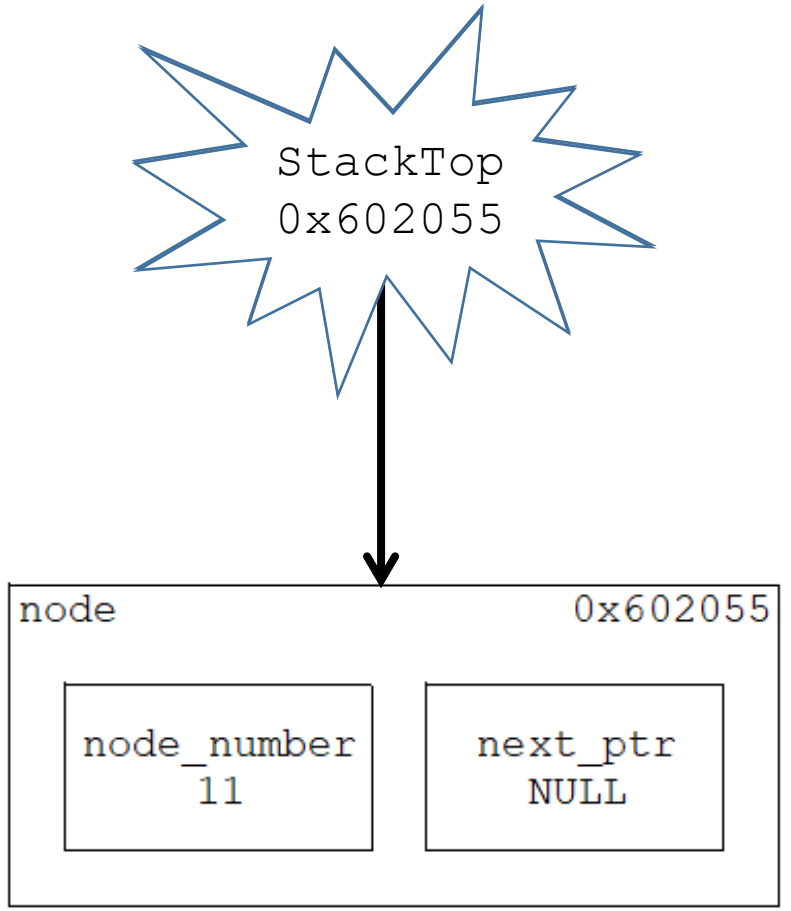


Stack

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

node *StackTop = NULL;
```

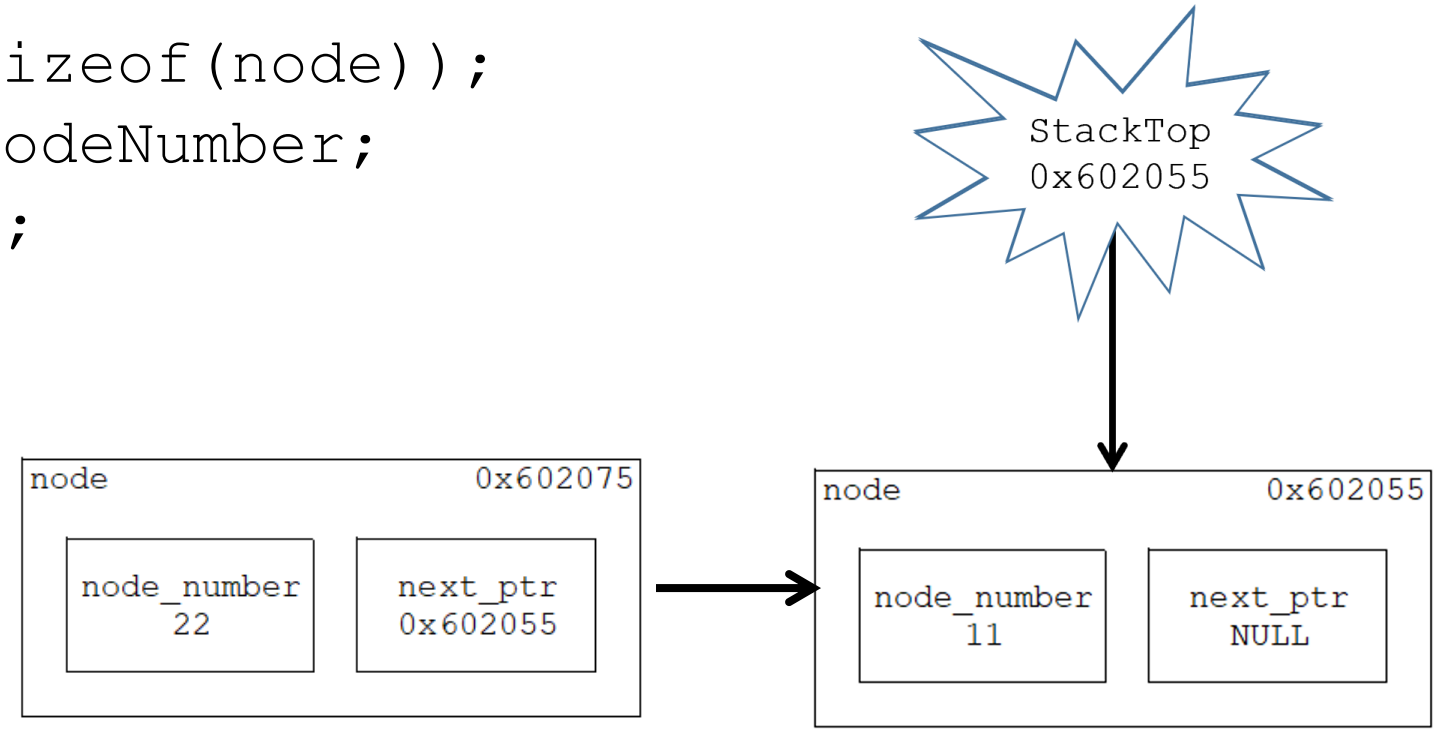
Stack Push

```
void push(node **StackTop, int NodeNumber)
{
    node *NewNode = malloc(sizeof(node));
    NewNode->node_number = NodeNumber;
    NewNode->next_ptr = NULL;

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



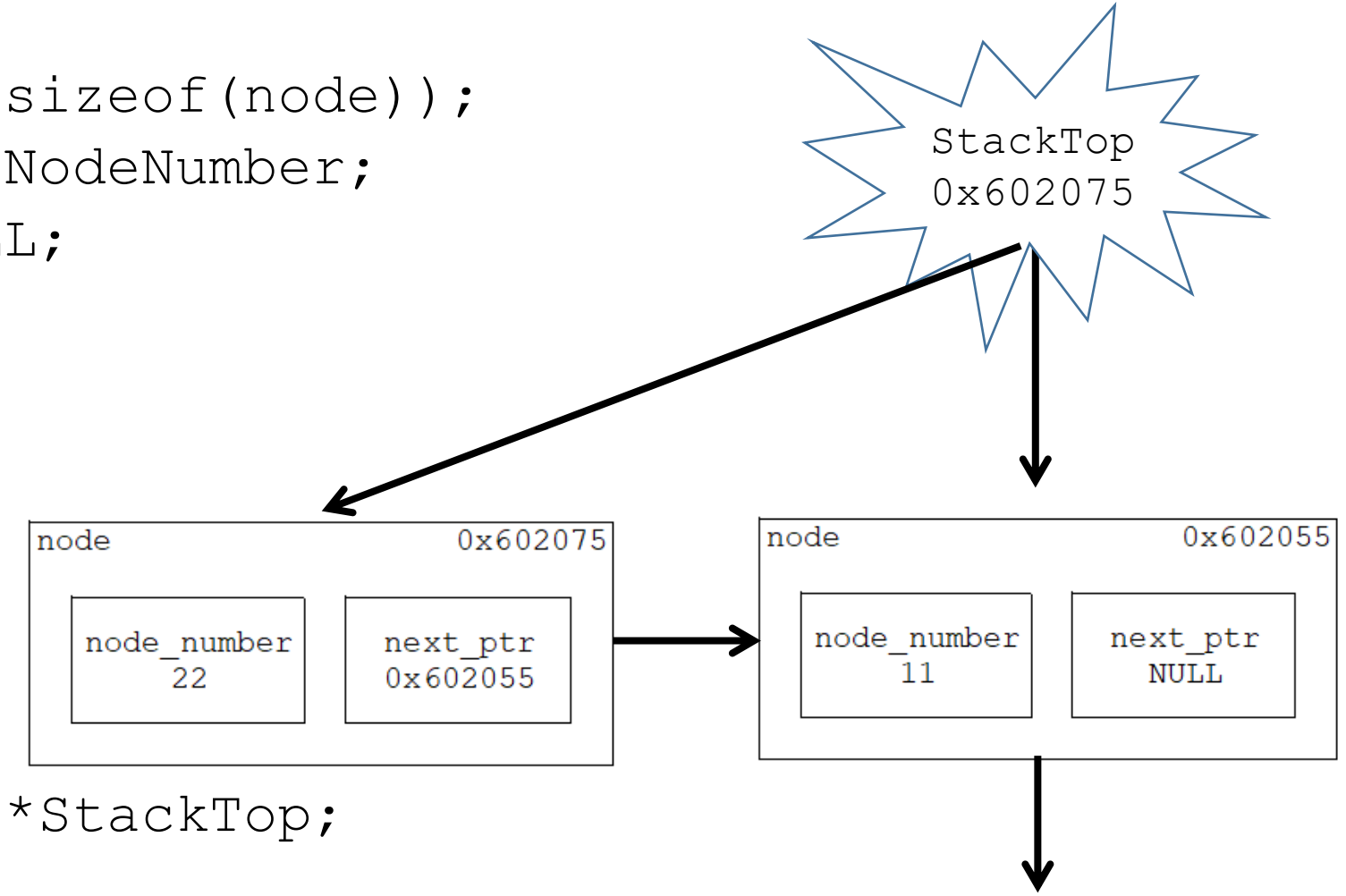
Stack Push

```
void push(node **StackTop, int NodeNumber)
{
    node *NewNode = malloc(sizeof(node));
    NewNode->node_number = NodeNumber;
    NewNode->next_ptr = NULL;

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



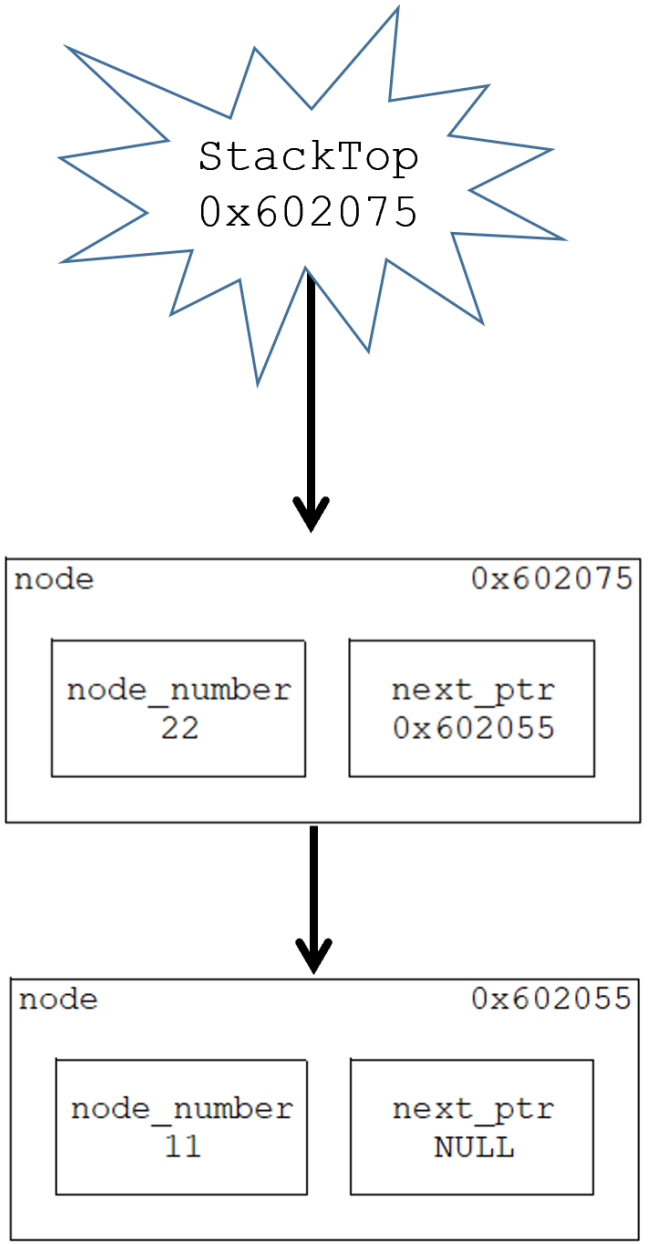
Stack Push

```
void push(node **StackTop, int NodeNumber)
{
    node *NewNode = malloc(sizeof(node));
    NewNode->node_number = NodeNumber;
    NewNode->next_ptr = NULL;

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



Stack Pop

```
void pop(node **StackTop)
{
    node *TempPtr = (*StackTop)->next_ptr;

    if (*StackTop == NULL)
    {
        printf("Stack is empty\n\n");
    }
    else
    {
        free(*StackTop);
        *StackTop = TempPtr;
    }
}
```



Queue

Queue is a linear data structure which follows a particular order in which the operations are performed. The order will be FIFO (First In First Out).

When you stand in line waiting for something, the person at the head of the line goes first and anyone new is added to the back of the line.

Queue

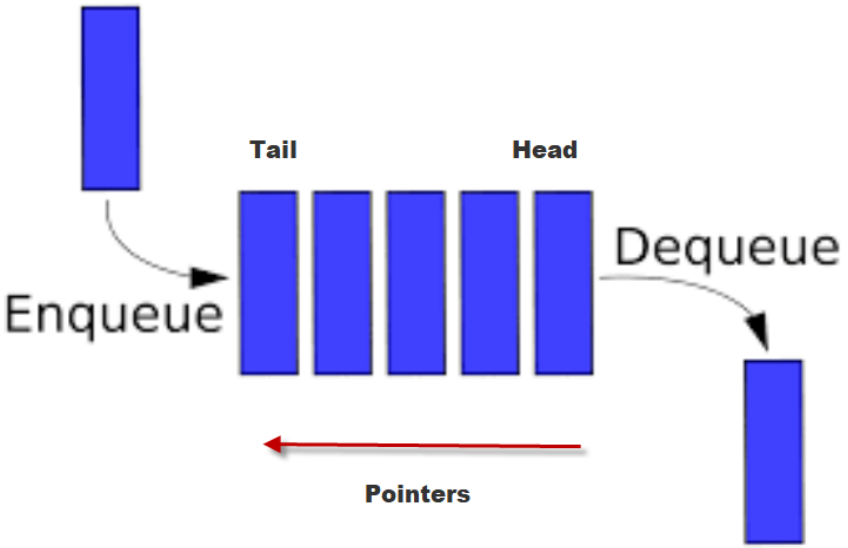
Operations on a Queue

Enqueue Adds an item to the queue.

Dequeue Removes an item from the queue.

Head Get the head of the queue

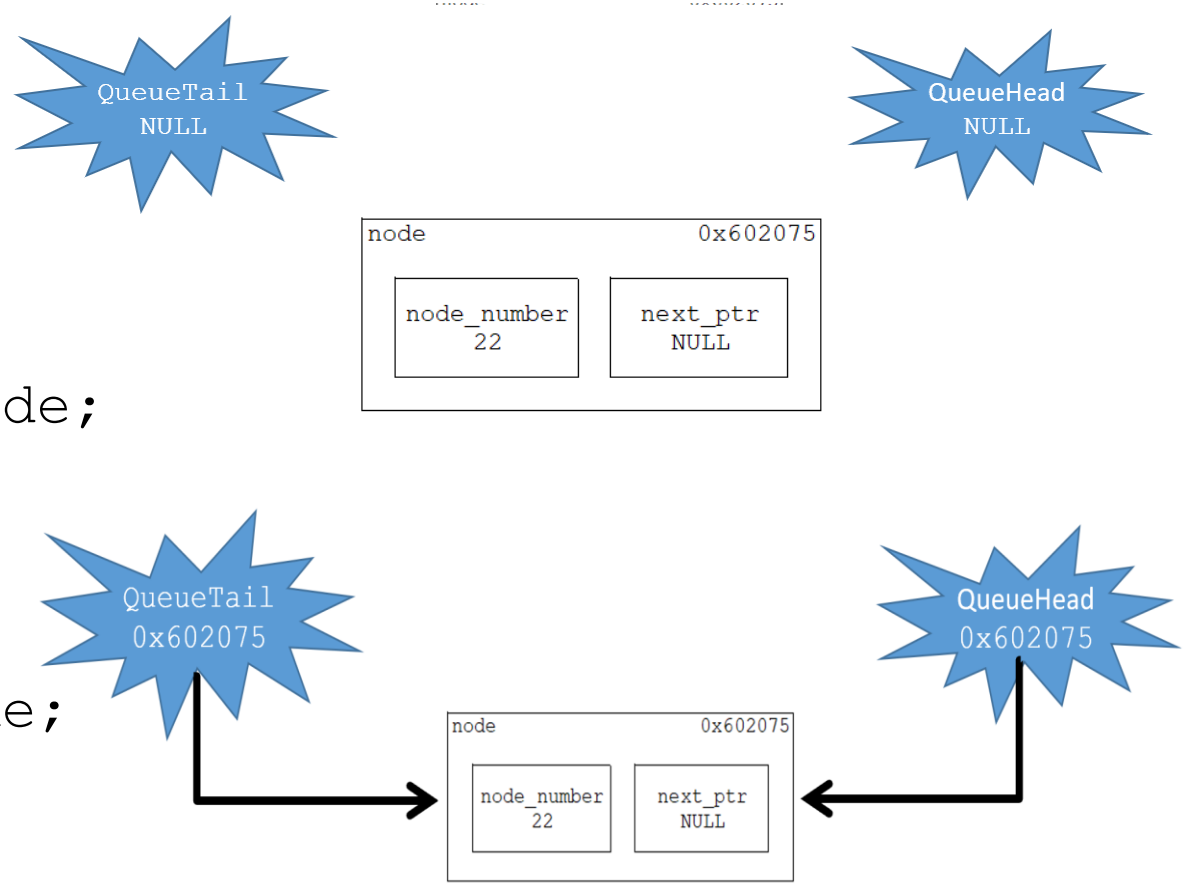
Tail Get the tail of the queue



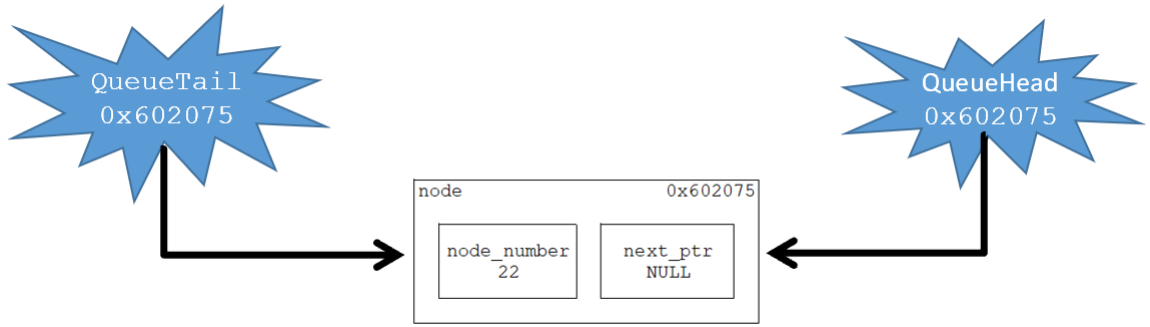
Queue

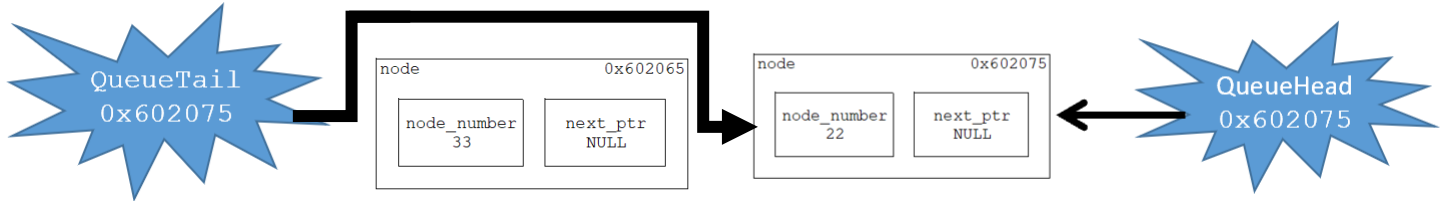
```
typedef struct node
{
    int node_number;
    struct node *next_ptr;
}
node;

node *QueueHead = NULL, *QueueTail =  NULL;
```

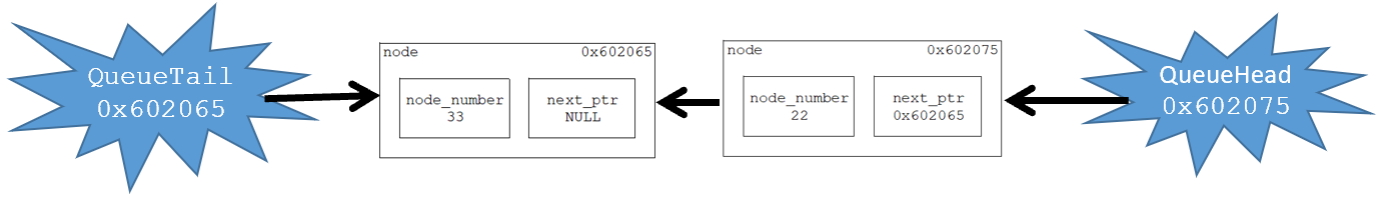
```
void enQueue(int NewNodeNumber, node **QueueHead, node **QueueTail)
{
    node *NewNode = malloc(sizeof(node));
    NewNode->node_number = NewNodeNumber;
    NewNode->next_ptr = NULL;

    /* Queue is empty */
    if (*QueueHead == NULL)
    {
        *QueueHead = *QueueTail = NewNode;
    }
    else
    {
        (*QueueTail)->next_ptr = NewNode;
        *QueueTail = NewNode;
    }
}
```



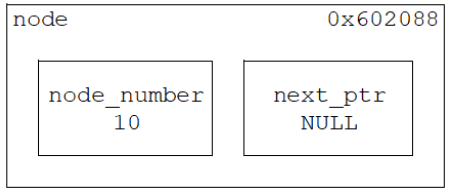
```
void enQueue(int NewNodeNumber, node **QueueHead, node **QueueTail)
{
    node *NewNode = malloc(sizeof(node));
    NewNode->node_number = NewNodeNumber;
    NewNode->next_ptr = NULL;

    /* Queue is empty */
    if (*QueueHead == NULL)
    {
        *QueueHead = *QueueTail = NewNode;
    }
    else
    {
        (*QueueTail)->next_ptr = NewNode;
        *QueueTail = NewNode;
    }
}
```

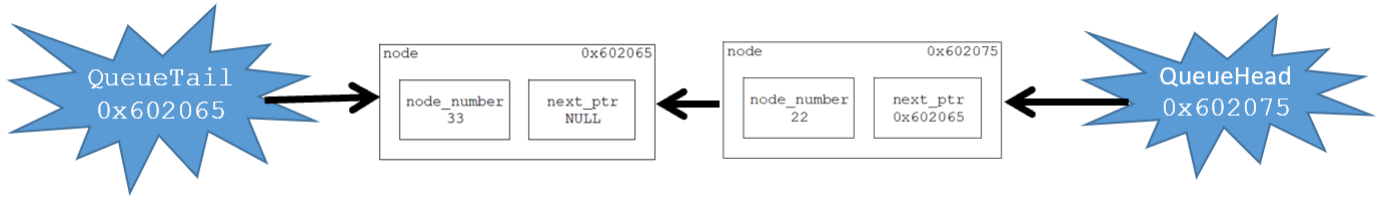


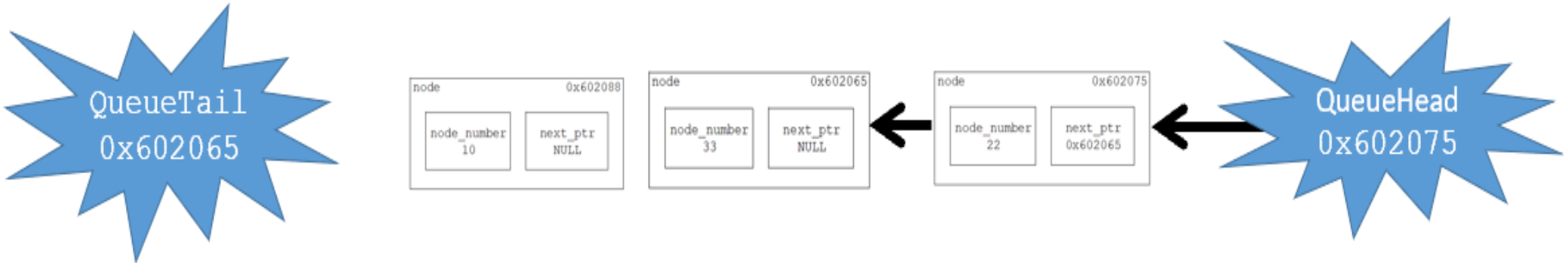


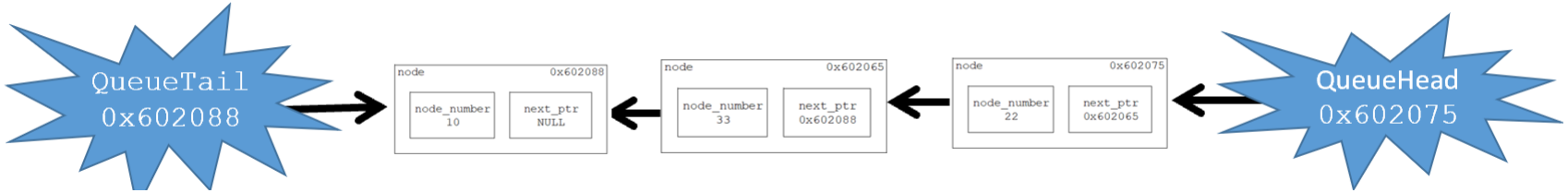


```
void enQueue(int NewNodeNumber, node **QueueHead, node **QueueTail)
{
    node *NewNode = malloc(sizeof(node));
    NewNode->node_number = NewNodeNumber;
    NewNode->next_ptr = NULL;

    /* Queue is empty */
    if (*QueueHead == NULL)
    {
        *Quei
    }
    else
    {
        (*QueueTail)->next_ptr = NewNode;
        *QueueTail = NewNode;
    }
}
```



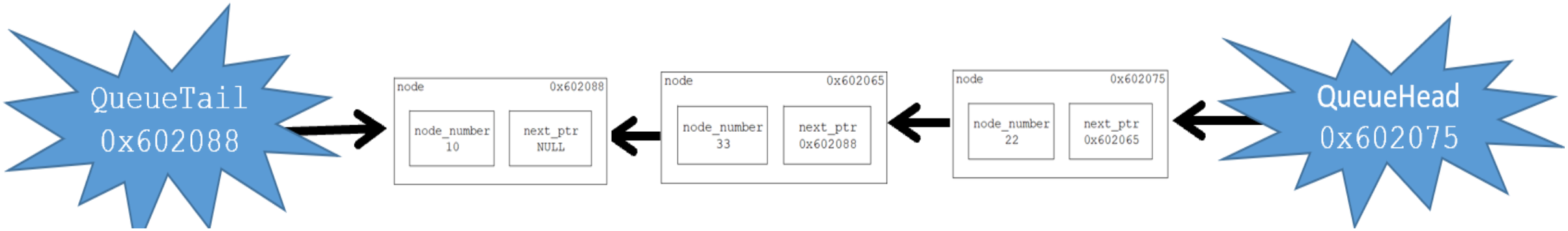






Queue Dequeue

```
void deQueue(node **QueueHead)
{
    node *TempPtr = (*QueueHead)->next_ptr;
```




```
    else
    {
        free(*QueueHead);
        *QueueHead = TempPtr;
    }
}
```

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;
    }
}
```

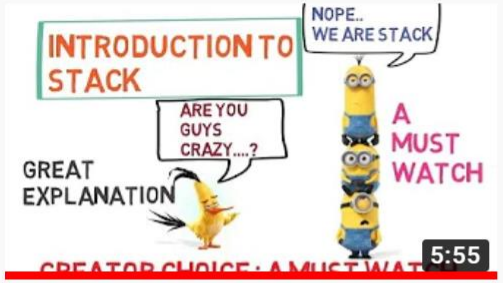
Traversing a Queue

```
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;
    }
}
```

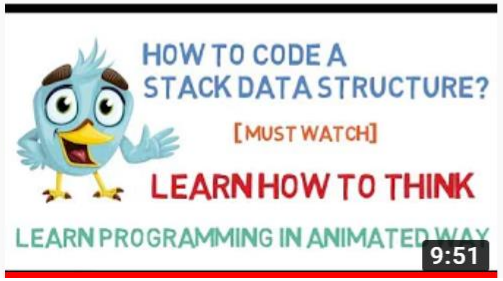
Codearchery Videos on YouTube



Introduction to Stack Data Structure

Codearchery • 18K views • 2 years ago

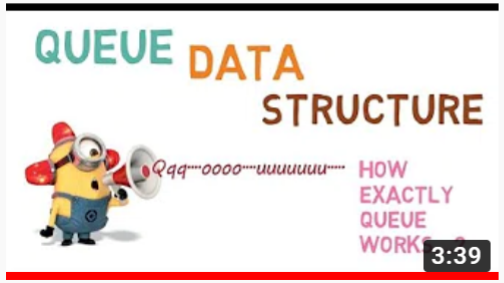
In this video will learn about **stack**. with an example of reverse a array. I hope you'll like this video. What is **Stack** Data Structure?



How to Code a Stack Data Structure

Codearchery • 8.8K views • 2 years ago

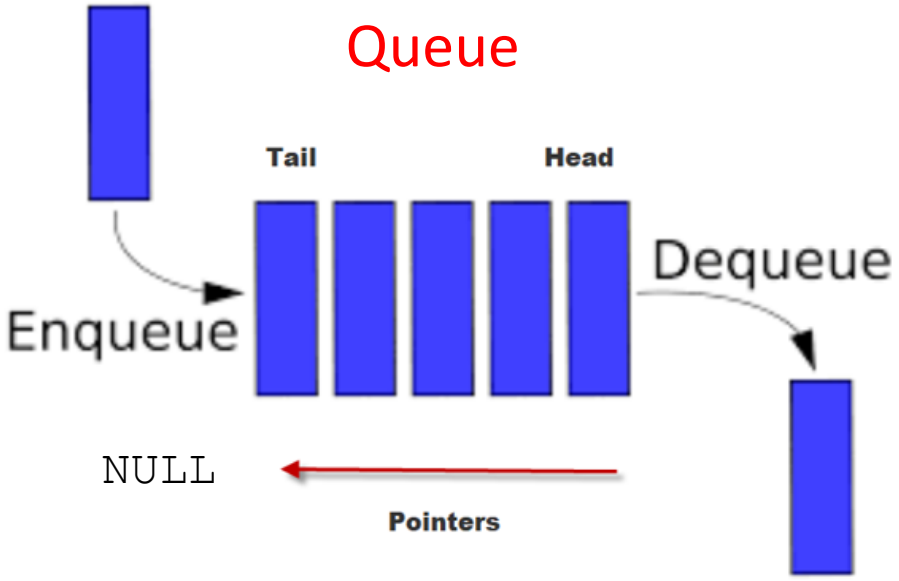
In this Video we will learn to Code **Stack** Data Structure. It's too Easy Once You Understand it I hope you will enjoy watching this ...

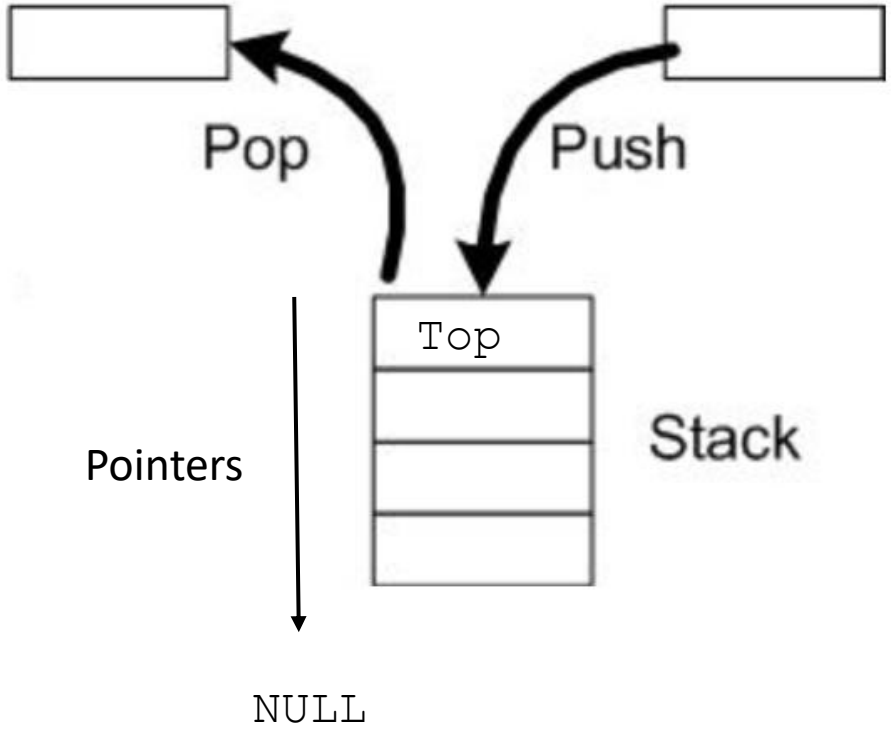


Introduction to Queue DS (Explained With Animation)

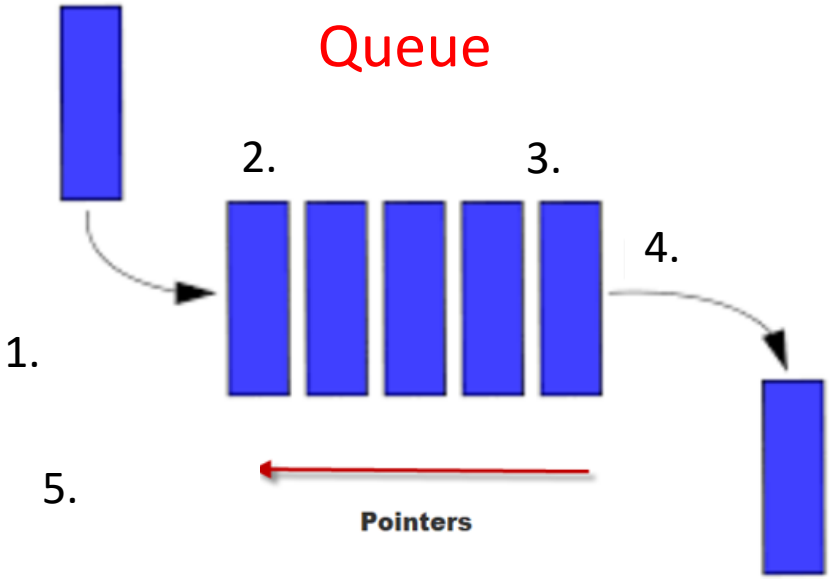
Codearchery • 10K views • 2 years ago

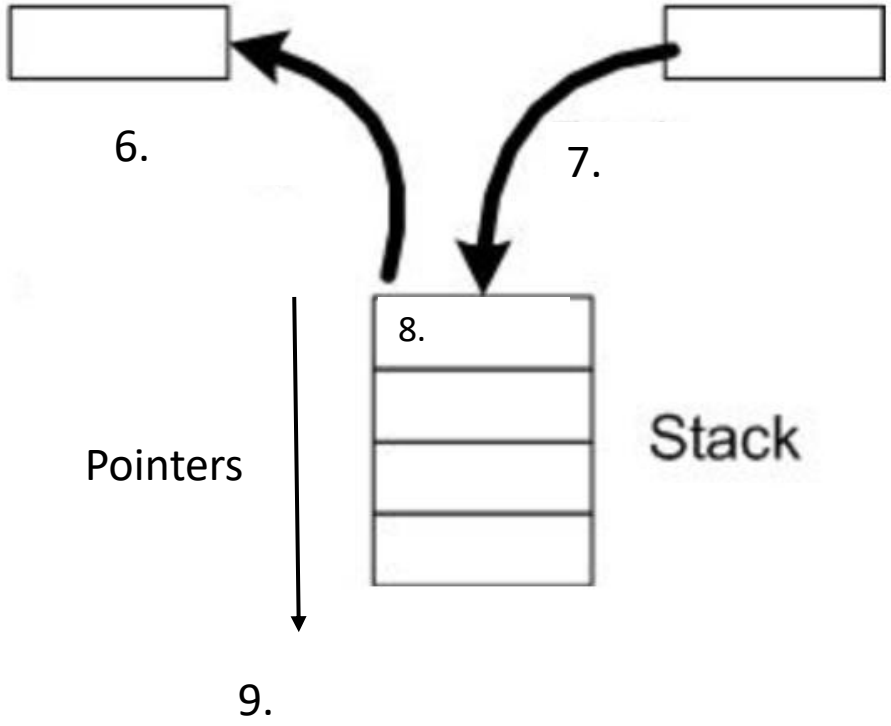
In this video I have explained everything about **Queue**. What is **queue**? Operation of **queue**? Applications of **queue**? I hope you ...






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





10.

```
{
    int node_number;
11.
}
```

12.

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.

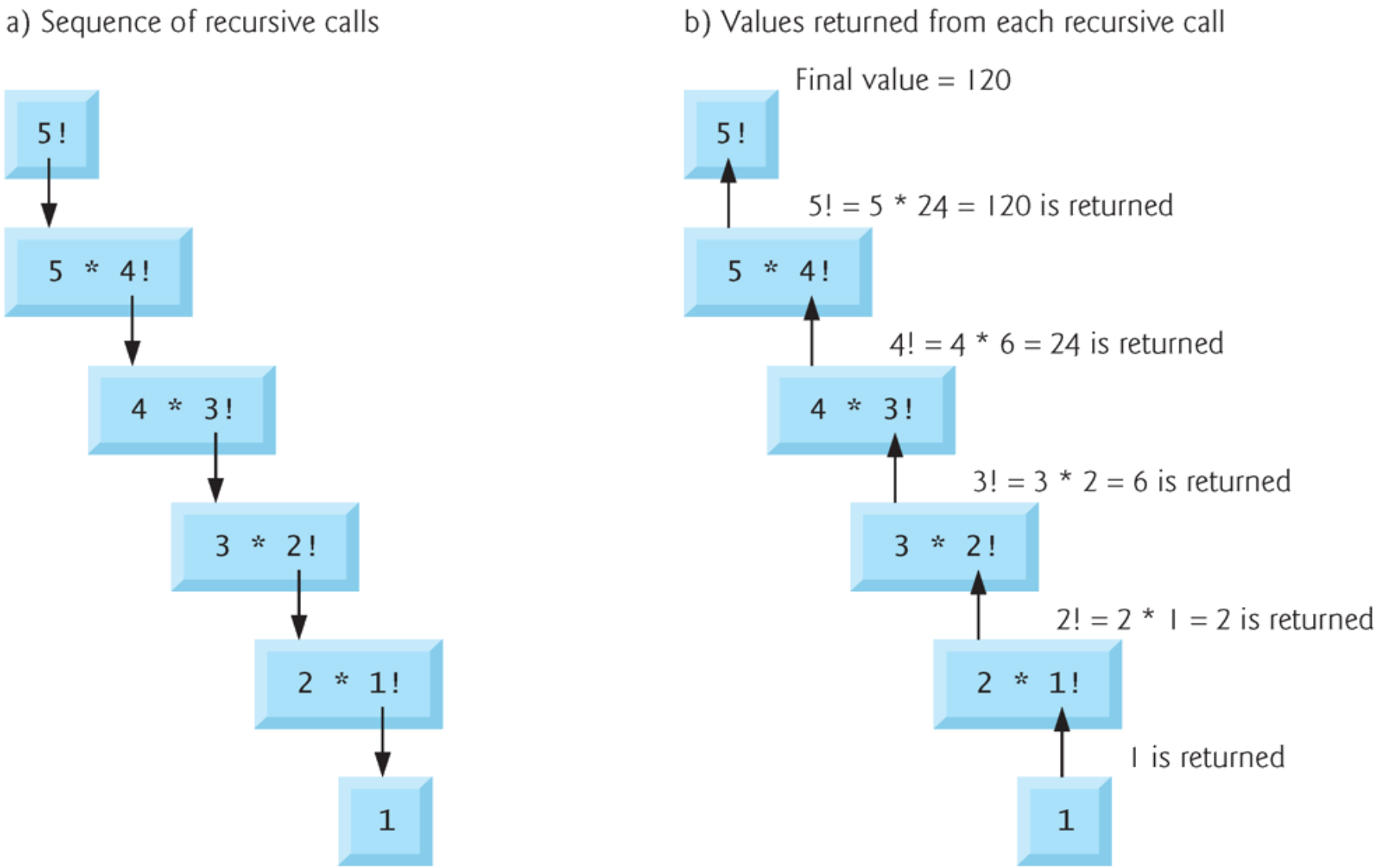
A simple example of a mathematical recursion is factorial

```
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!

```
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  0x00000000040053d in main () at frDemo.c:19
```

After processing `n=4`

```
#0  0x00000000040053d 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;
    }
}
```