CSE 1320

Week of 04/10/2023

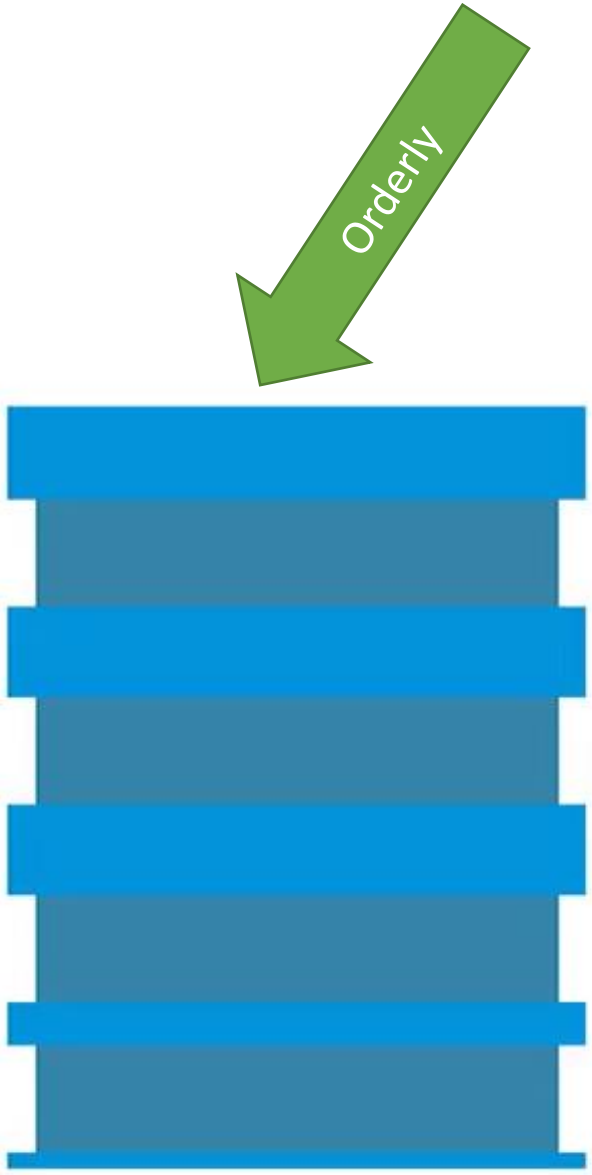
Instructor : Donna French

Layout of Memory

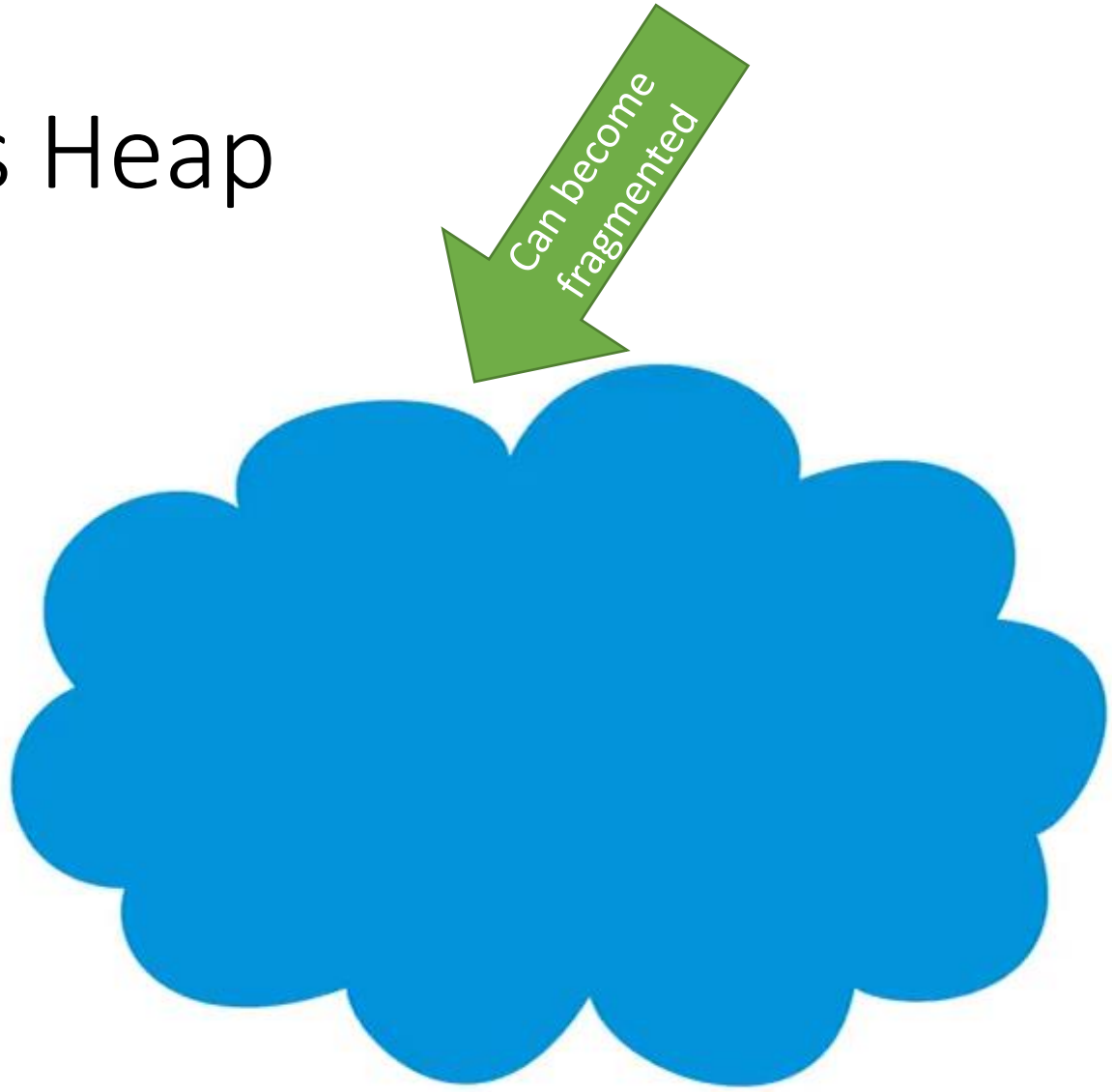
Higher Addresses

Lower Addresses

Stack vs Heap



Stack



Heap

Heap Memory vs Stack Memory

- **Stack** is used for static memory allocation
  - Memory is managed for you
  - Variables cannot be resized
  - Access is easier and faster and cache friendly
  - Not flexible – allotted memory cannot be changed
  - Faster access – allocation and deallocation
- **Heap** is used for dynamic memory allocation
  - Memory management needs to be done manually
  - Variables can be resized
  - Causes more cache misses because of being dispersed throughout memory
  - Flexible and allotted memory can be altered
  - Slower access – allocation and deallocation

Both are stored in the computer's RAM .

Dynamic Allocation and De-Allocation of Memory

Functions for dynamic allocation and de-allocation of memory

`malloc()`
`calloc()`
`realloc()`

`free()`

Must include `stdlib.h` to use them

Dynamic Allocation and De-Allocation of Memory

```
void *malloc(size_t size)
```

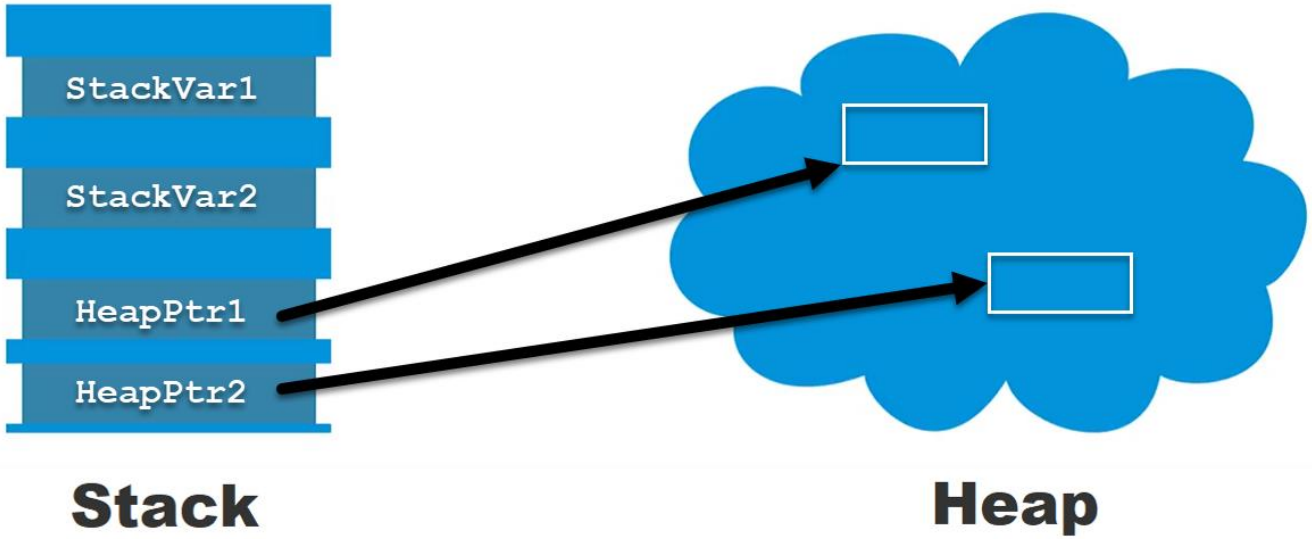
one parameter – the number of bytes to allocate

return value – address of the first byte in the newly allocated buffer

The memory allocated is **uninitialized**.

```
int main(void)
{
    int StackVar1;
    int StackVar2 = 0;
    int *HeapPtr1 = NULL;
    int *HeapPtr2 = NULL;

    HeapPtr1 = malloc(sizeof(int));
    HeapPtr2 = malloc(sizeof(int));

    return 0;
}
```

```
(gdb) p &StackVar1
$8 = (int *) 0x7fffffffe798
(gdb) p &StackVar2
$9 = (int *) 0x7fffffffe79c
(gdb) p &HeapPtr1
$10 = (int **) 0x7fffffffe7a0
(gdb) p &HeapPtr2
$11 = (int **) 0x7fffffffe7a8
(gdb) p HeapPtr1
$12 = (int *) 0x601010
(gdb) p HeapPtr2
$13 = (int *) 0x601030
```



malloc1Demo.c

```
int StackVar1 = 0;
int StackVar2 = 0;
int *HeapPtr1 = NULL;
int *HeapPtr2 = NULL;

HeapPtr1 = malloc(sizeof(int));
HeapPtr2 = malloc(sizeof(int));

printf("StackVar1 = %d\nStackVar2 = %d"
       "\nHeapPtr1 = %d\nHeapPtr2 = %d",
       StackVar1, StackVar2,
        *HeapPtr1, *HeapPtr2);

*HeapPtr1 = 100;
*HeapPtr2 = 200;

printf("\nHeapPtr1 = %d\nHeapPtr2 = %d\n",
       *HeapPtr1, *HeapPtr2);
```

```
StackVar1 = 0
StackVar2 = 0
HeapPtr1 = 0
HeapPtr2 = 0
HeapPtr1 = 100
HeapPtr2 = 200
```

malloc2Demo.c

```
int *ArrayPtr1 = NULL;
int ArraySize = 0;
int i = 0;

printf("How big of an array do you want to create? ");
scanf("%d", &ArraySize);

ArrayPtr1 = malloc(ArraySize*sizeof(int));

for (i = 0; i < ArraySize; i++)
{
   *(ArrayPtr1+i) = i+65;
}

for (i = 0; i < ArraySize; i++)
{
   printf("Element[%d] = %c\n", i, *(ArrayPtr1 + i));
}
```

```
How big of an array do you want to create? 11
Element[0] = A
Element[1] = B
Element[2] = C
Element[3] = D
Element[4] = E
Element[5] = F
Element[6] = G
Element[7] = H
Element[8] = I
Element[9] = J
Element[10] = K
```

malloc3Demo.c

Dynamic Allocation and De-Allocation of Memory

```
void *calloc(size_t n, size_t size)
```

first parameter – the number of items

second parameter – the size of each item

return value – address of the first byte in the newly allocated buffer

The memory allocated is **initialized to 0**.

Dynamic Allocation and De-Allocation of Memory

`malloc() vs calloc()`

`malloc()` does not initialize the memory it allocates.

`calloc()` does initialize the memory it allocates.

`mvscallocDemo.c`

```
Calling malloc()

Printing array
contents after
malloc()

ArrayPtr[0] = 0
ArrayPtr[1] = 0
ArrayPtr[2] = 0
ArrayPtr[3] = 0
ArrayPtr[4] = 0
ArrayPtr[5] = 0
ArrayPtr[6] = 0
ArrayPtr[7] = 0
ArrayPtr[8] = 0
ArrayPtr[9] = 0
```

```
Printing array contents
after filling with random
numbers

ArrayPtr[0] = 1804289383
ArrayPtr[1] = 846930886
ArrayPtr[2] = 1681692777
ArrayPtr[3] = 1714636915
ArrayPtr[4] = 1957747793
ArrayPtr[5] = 424238335
ArrayPtr[6] = 719885386
ArrayPtr[7] = 1649760492
ArrayPtr[8] = 596516649
ArrayPtr[9] = 1189641421


freeing memory
```

```
Calling malloc() again

Printing array contents
after malloc()


ArrayPtr[0] = 0
ArrayPtr[1] = 0
ArrayPtr[2] = 1681692777
ArrayPtr[3] = 1714636915
ArrayPtr[4] = 1957747793
ArrayPtr[5] = 424238335
ArrayPtr[6] = 719885386
ArrayPtr[7] = 1649760492
ArrayPtr[8] = 596516649
ArrayPtr[9] = 1189641421
```

mvscallocDemo.c

```
Calling calloc()

Printing array
contents after
calloc()

ArrayPtr[0] = 0
ArrayPtr[1] = 0
ArrayPtr[2] = 0
ArrayPtr[3] = 0
ArrayPtr[4] = 0
ArrayPtr[5] = 0
ArrayPtr[6] = 0
ArrayPtr[7] = 0
ArrayPtr[8] = 0
ArrayPtr[9] = 0
```

```
Printing array contents
after filling with random
numbers

ArrayPtr[0] = 1025202362
ArrayPtr[1] = 1350490027
ArrayPtr[2] = 783368690
ArrayPtr[3] = 1102520059
ArrayPtr[4] = 2044897763
ArrayPtr[5] = 1967513926
ArrayPtr[6] = 1365180540
ArrayPtr[7] = 1540383426
ArrayPtr[8] = 304089172
ArrayPtr[9] = 1303455736


freeing memory
```

```
Calling calloc() again

Printing array contents
after calloc()


ArrayPtr[0] = 0
ArrayPtr[1] = 0
ArrayPtr[2] = 0
ArrayPtr[3] = 0
ArrayPtr[4] = 0
ArrayPtr[5] = 0
ArrayPtr[6] = 0
ArrayPtr[7] = 0
ArrayPtr[8] = 0
ArrayPtr[9] = 0
```

mvscallocDemo.c

Dynamic Allocation and De-Allocation of Memory

So when to use `malloc()` vs `calloc()`?

Zeroing out the memory may take a little time, so you probably want to use `malloc()` if performance is an issue.

If initializing the memory is more important, use `calloc()`.

Dynamic Allocation and De-Allocation of Memory

```
void free(void *ptr)
```

one parameter – pointer to the allocated space

`free()` should be used when allocated memory is no longer needed in order to avoid memory leaks.

A memory leak is caused when a program fails to release discarded memory causing impaired performance or failure.

```
How big of an array shall we create? 5
Calling malloc() for ArrayPtr

ArrayPtr 0x601010 - Enter array element 0 1
ArrayPtr 0x601014 - Enter array element 1 2
ArrayPtr 0x601018 - Enter array element 2 3
ArrayPtr 0x60101c - Enter array element 3 4
ArrayPtr 0x601020 - Enter array element 4 5

Printing ArrayPtr

ArrayPtr[0] = 1
ArrayPtr[1] = 2
ArrayPtr[2] = 3
ArrayPtr[3] = 4
ArrayPtr[4] = 5
```

```
Printing ArrayPtr

ArrayPtr[0] = 0
ArrayPtr[1] = 0
ArrayPtr[2] = 3
ArrayPtr[3] = 4
ArrayPtr[4] = 5
```

```
(gdb) p ArrayPtr
$8 = (int *) 0x601010
(gdb) p *ArrayPtr@5
$9 = {1, 2, 3, 4, 5}

31                  free(ArrayPtr);

(gdb) p ArrayPtr
$10 = (int *) 0x601010
(gdb) p *ArrayPtr@5
$11 = {0, 0, 3, 4, 5}
```

malloc5Demo.c

```
Breakpoint 2, main () at malloc5Demo.c:31
31                  free(ArrayPtr);
(gdb) step
32                  ArrayPtr = NULL;
(gdb) p ArrayPtr
$1 = (int *) 0x601010
(gdb) step
34                  printf("\nPrinting ArrayPtr\n\n");
(gdb) p ArrayPtr
$2 = (int *) 0x0


Printing ArrayPtr


Program received signal SIGSEGV, Segmentation fault.
0x00000000004006db in main () at malloc5Demo.c:38
```

malloc5Demo.c

Dynamic Allocation and De-Allocation of Memory

```
void *realloc(void *ptr, size_t newsize)
```

first parameter – pointer to the first byte of memory that was previously allocated using `malloc()` or `calloc()`

second parameter – new size of the block in bytes

return value – pointer to new block of memory. Will change your pointer if needed to allocate the new contiguous block of memory.

reallocDemo.c

```
How many train cars do you have? 3

Enter who's in train car 1 Clowns
Enter who's in train car 2 Tiger
Enter who's in train car 3 Lion

Train has been created

ENGINE + Car1 Clowns + Car2 Tiger + Car3 Lion

Do you want to add more cars?  How many more?
```

```
Do you want to add more cars?  How many more? 2

Enter who's in train car 4 Zebra

Enter who's in train car 5 Gorilla


Train has been created


ENGINE + Car1 Clowns + Car2 Tiger + Car3 Lion + Car4 Zebra + Car5 Gorilla
```

reallocDemo.c

```
struct TrainCar
{
   char Type[20];
   int  Number;
};

struct TrainCar *TrainCarPtr = NULL;

printf("How many train cars do you have? ");
scanf("%d", &TrainCarCount);

TrainCarCount++;  // Add 1 for the engine

TrainCarPtr = malloc(TrainCarCount * sizeof(struct TrainCar));




printf("Do you want to add more cars?  How many more? ");
scanf("%d", &AdditionalTrainCars);

TrainCarPtr = realloc(TrainCarPtr,
                      (TrainCarCount+AdditionalTrainCars) * sizeof(struct TrainCar));
```

Dynamic Allocation and De-Allocation of Memory

```
void *realloc(void *ptr, size_t newsize)
```

Old data is not lost and newly allocated memory is not initialized.

If `realloc()` fails, then `NULL` will be returned and the old memory remains unaffected.

Dynamic Allocation and De-Allocation of Memory

When we `malloc()` memory, it is contiguous like a train.

When we use `realloc()`, we can add more cars to the train but they are always added to the end.

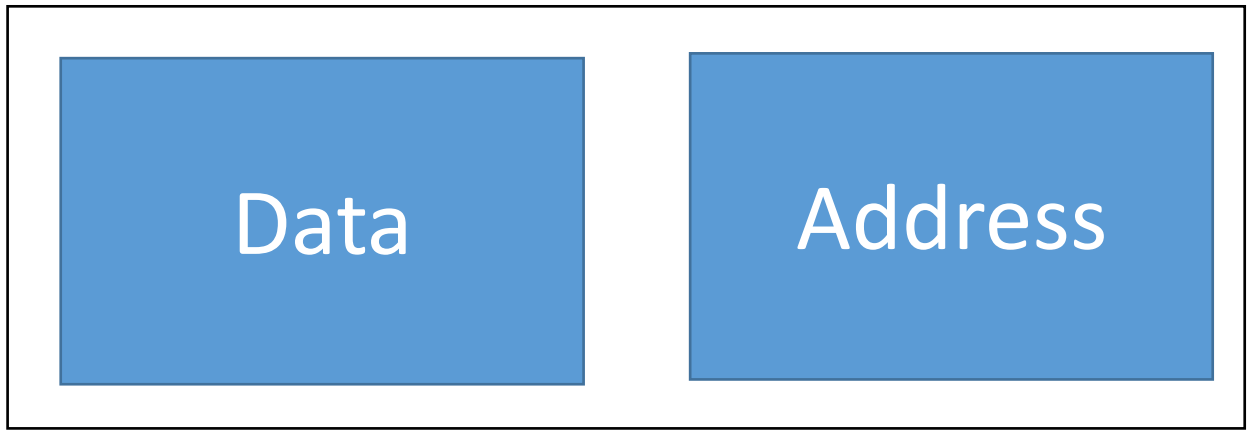
Question – How do we move cars around? How do we delete or insert cars?

Linked List

Linked list is a linear data structure which consists of groups of nodes in a sequence.

Each node holds its own data and address of the next node; hence, forming a chain like structure.

node

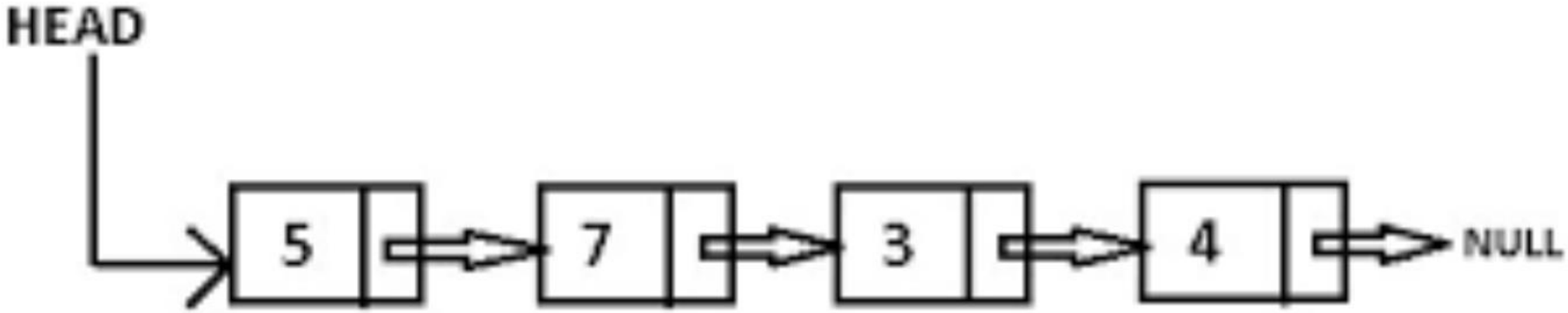


Linked List

Advantages	Disadvantages
Dynamic; therefore, only allocate memory when required	Memory is wasted due to extra storage needed for pointers
Insertion and deletion operations can be easily implemented	No element can be accessed randomly – sequential access
Stacks and queues can be easily executed	

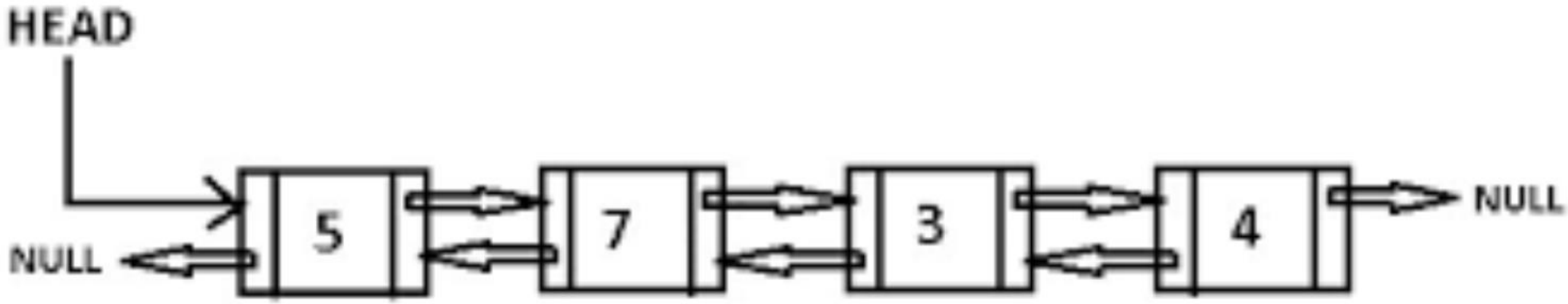
Linked List

Single Linked List



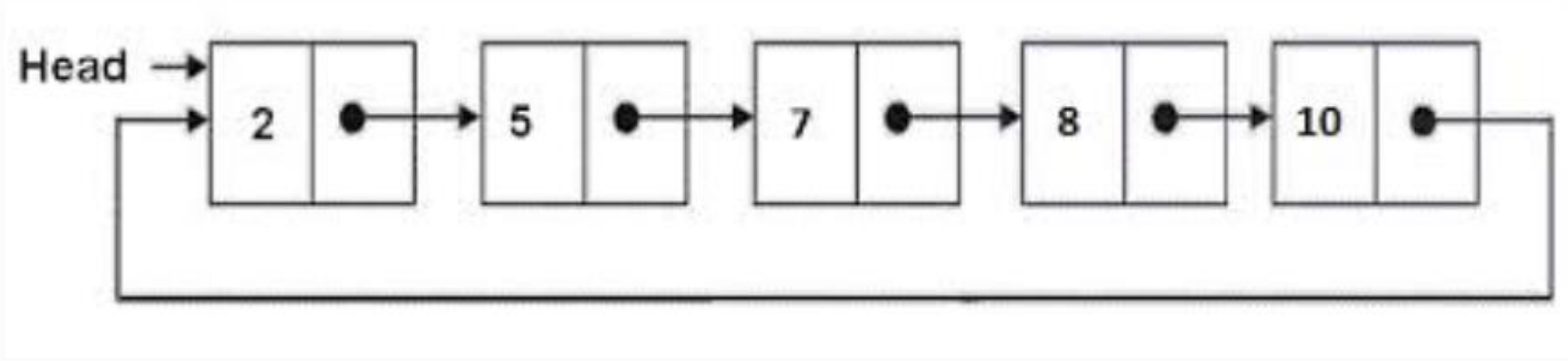
Linked List

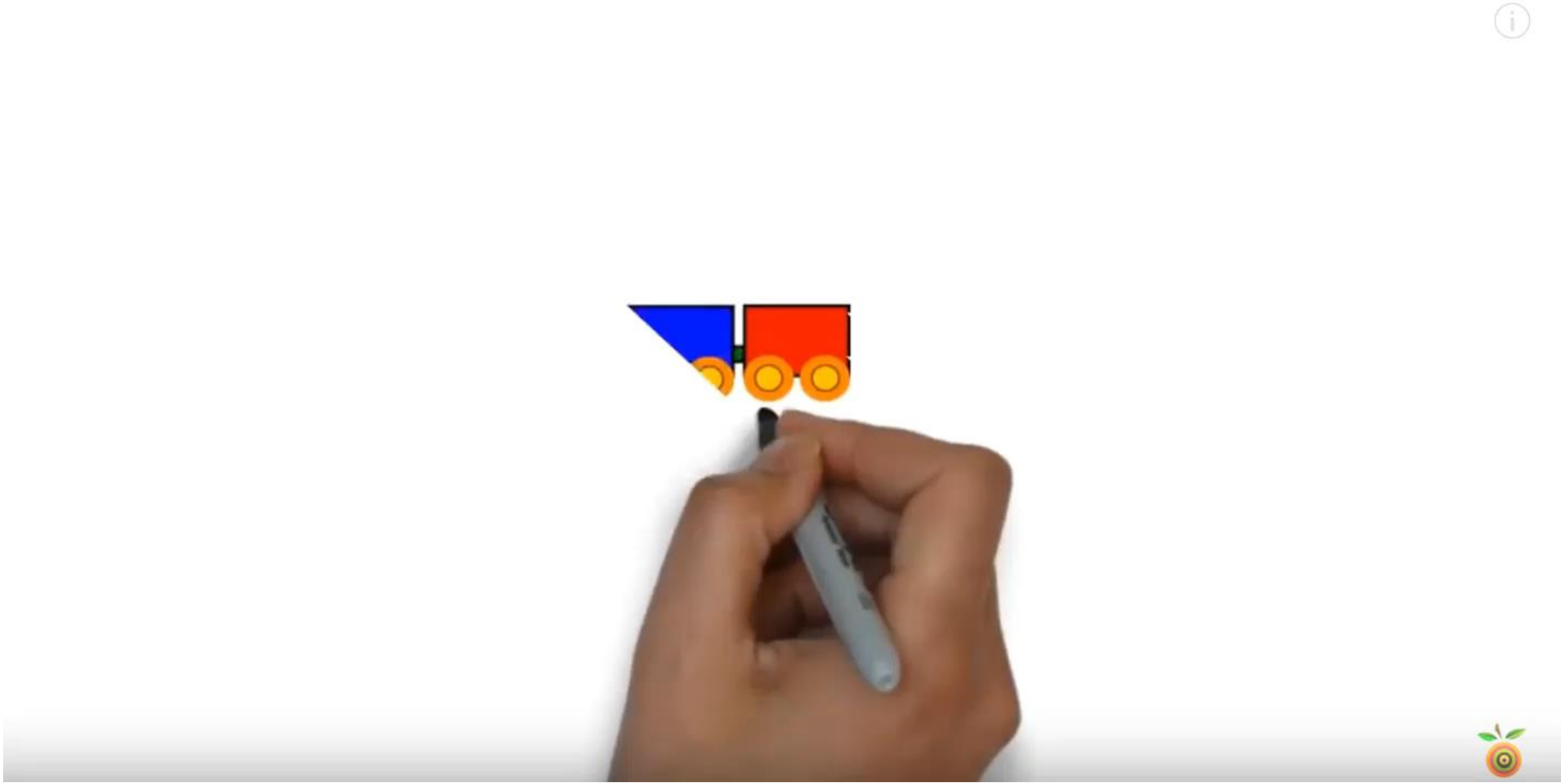
Double Linked List



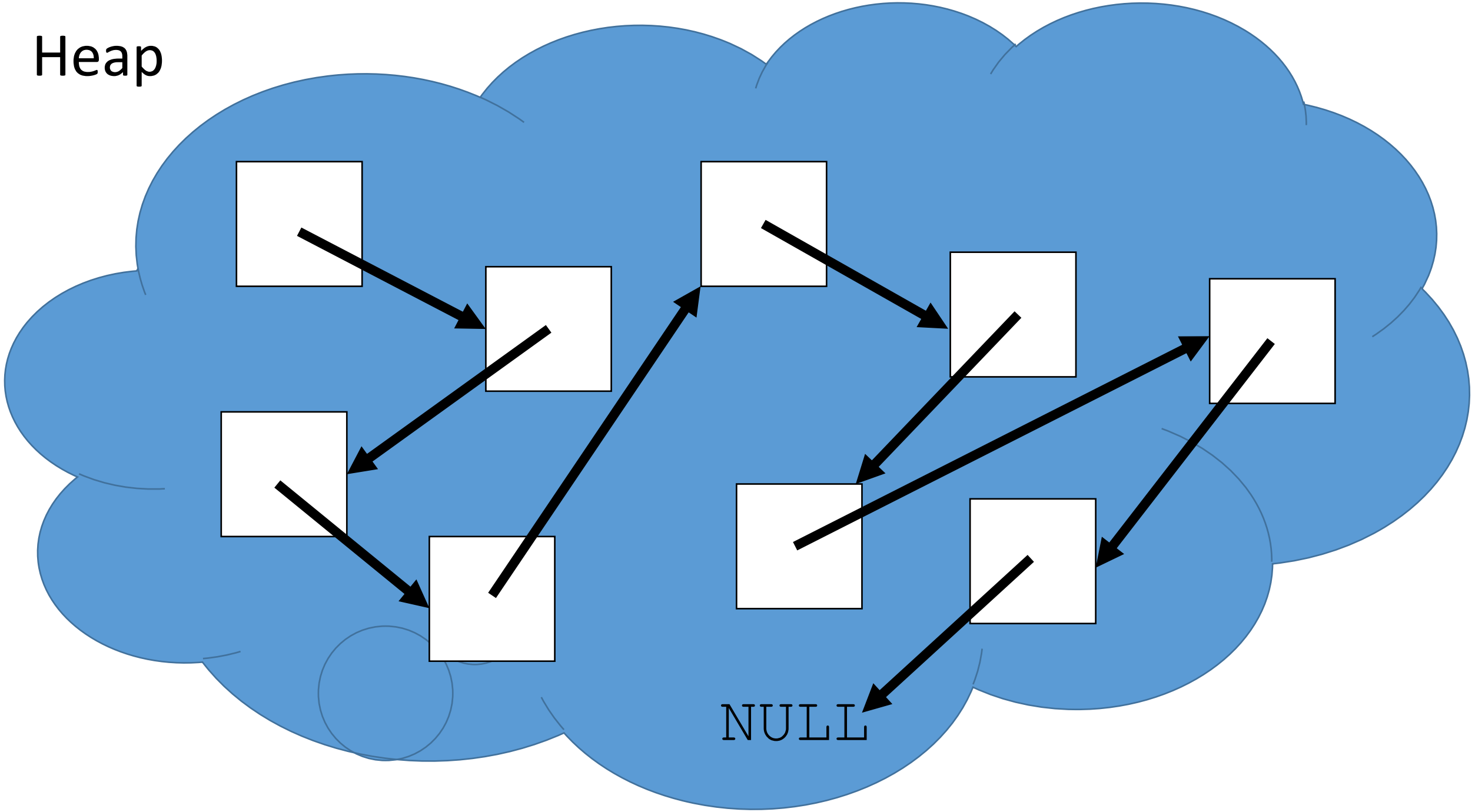
Linked List

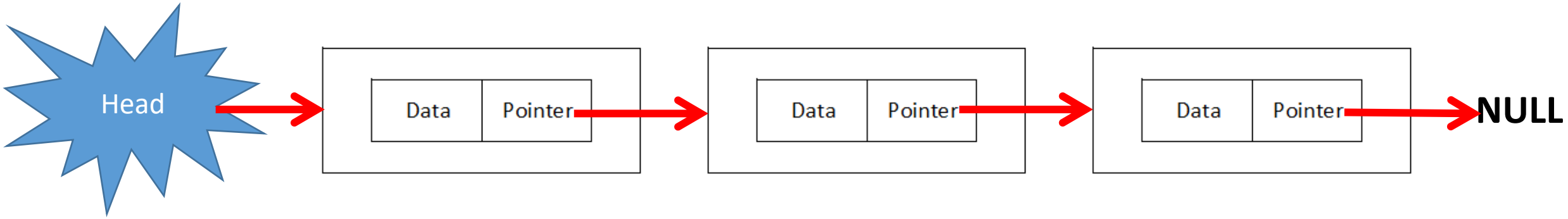
Circular Linked List





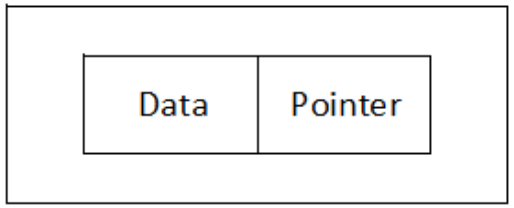
Heap





Add a node

Insert a node



Delete a node

Linked List

Creating the head node of the linked list

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

struct node *LinkedListHead;

LinkedListHead = NULL;
```

Using `node_number` as a simple example – could be many types of data

`next_ptr` is a pointer to the `struct` a pointer to the `struct` is OK – just a `struct` would not compile

pointer to the linked list is referred to as the "head" of the list

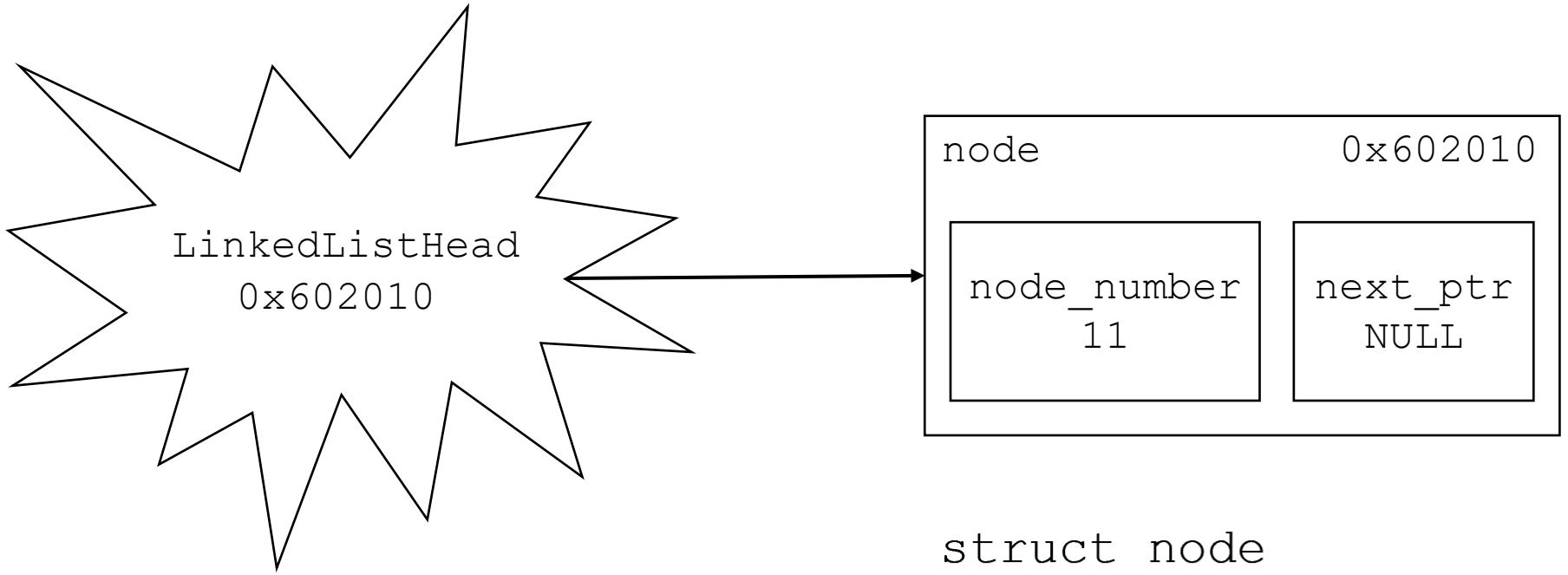
Set to `NULL` since it is not pointing to anything yet

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

int main(void)
{
    struct node *LinkedListHead;

    LinkedListHead = NULL;

(gdb) p LinkedListHead
$1 = (struct node *) 0x0

(gdb) ptype LinkedListHead
type = struct node {
    int node_number;
    struct node *next_ptr;
} *
```

pointer

```
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 : 5


Enter the node number to add to the start of the list : 11

Node Number 11          Node Address 0x602010     Node Next Pointer (nil)
```

Linked List – Add the first link

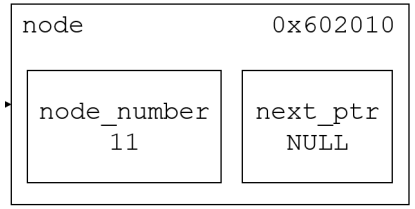



```
struct node *LinkedListHead;

LinkedListHead = NULL;
```

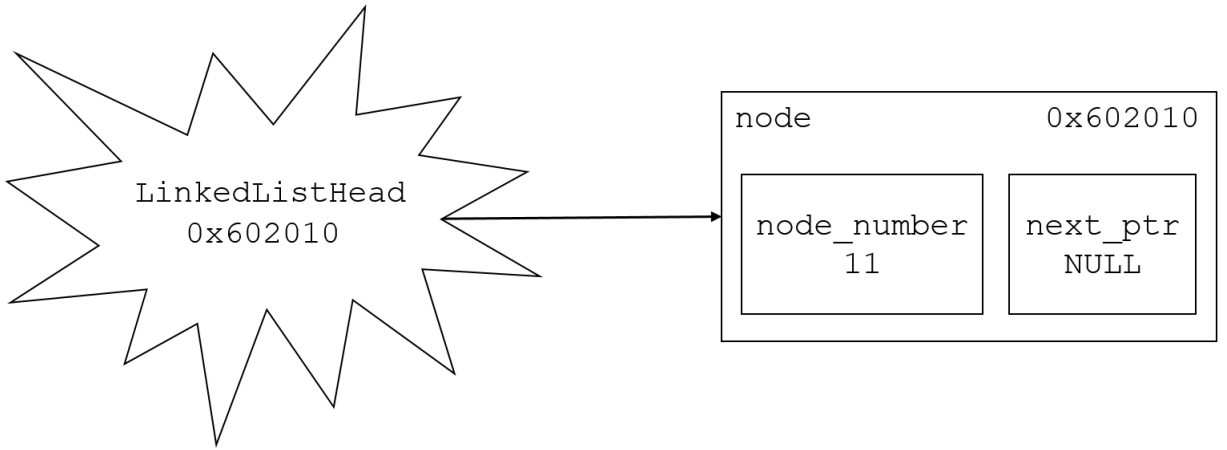
```
struct node
{
    int node_number;
    struct node *next_ptr;
};
```

Linked List – Add Node to Start

```
struct node *NewNode;

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

/* Head pointer is NULL so put new node address in it*/
if (LinkedListHead == NULL)
{
   LinkedListHead = NewNode;
}
```

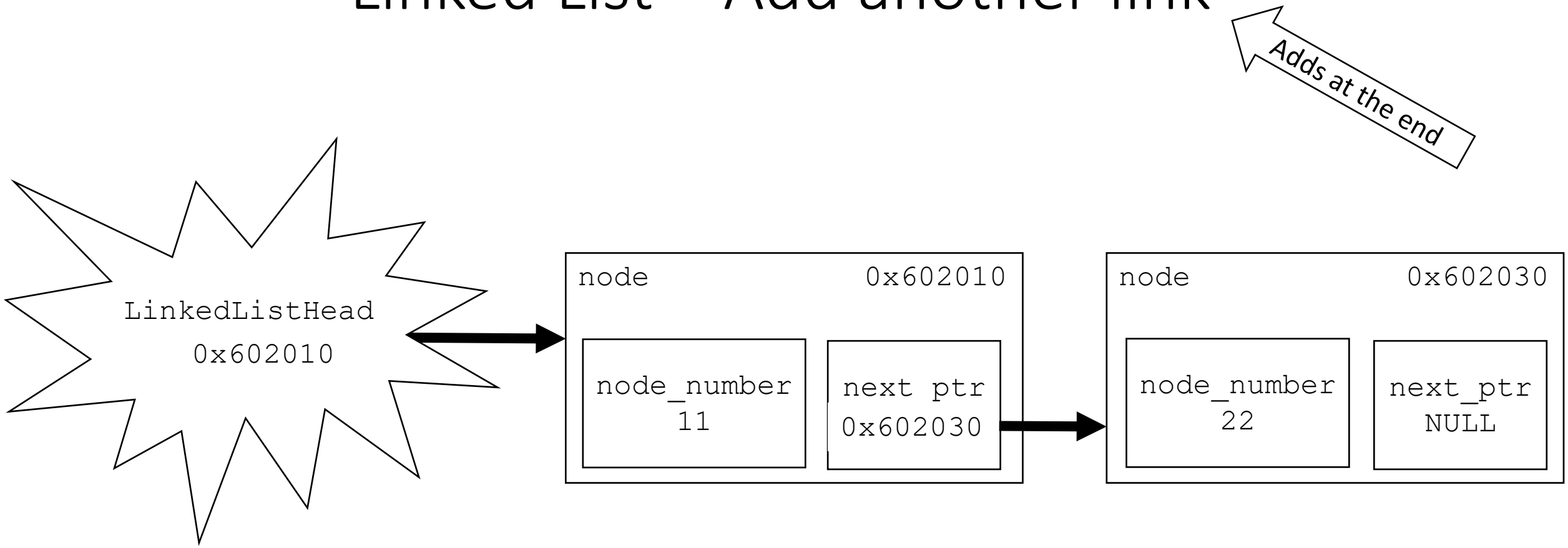


```
47          NewNode = malloc(sizeof(struct node));
(gdb) p NewNode
$1 = (struct node *) 0x602010

48          NewNode->node_number = NodeNumberToAdd;
49          NewNode->next_ptr = NULL;

(gdb) p *NewNode
$3 = {node_number = 11, next_ptr = 0x0}


(gdb) p LinkedListHead
$4 = (struct node *) 0x0

52          if (LinkedListHead == NULL)
54             LinkedListHead = NewNode;

(gdb) p LinkedListHead
$5 = (struct node *) 0x602010
```



```
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 : 6


Enter the node number to add to the end of the list : 22

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

Node Number 22          Node Address 0x602030     Node Next Pointer (nil)
```

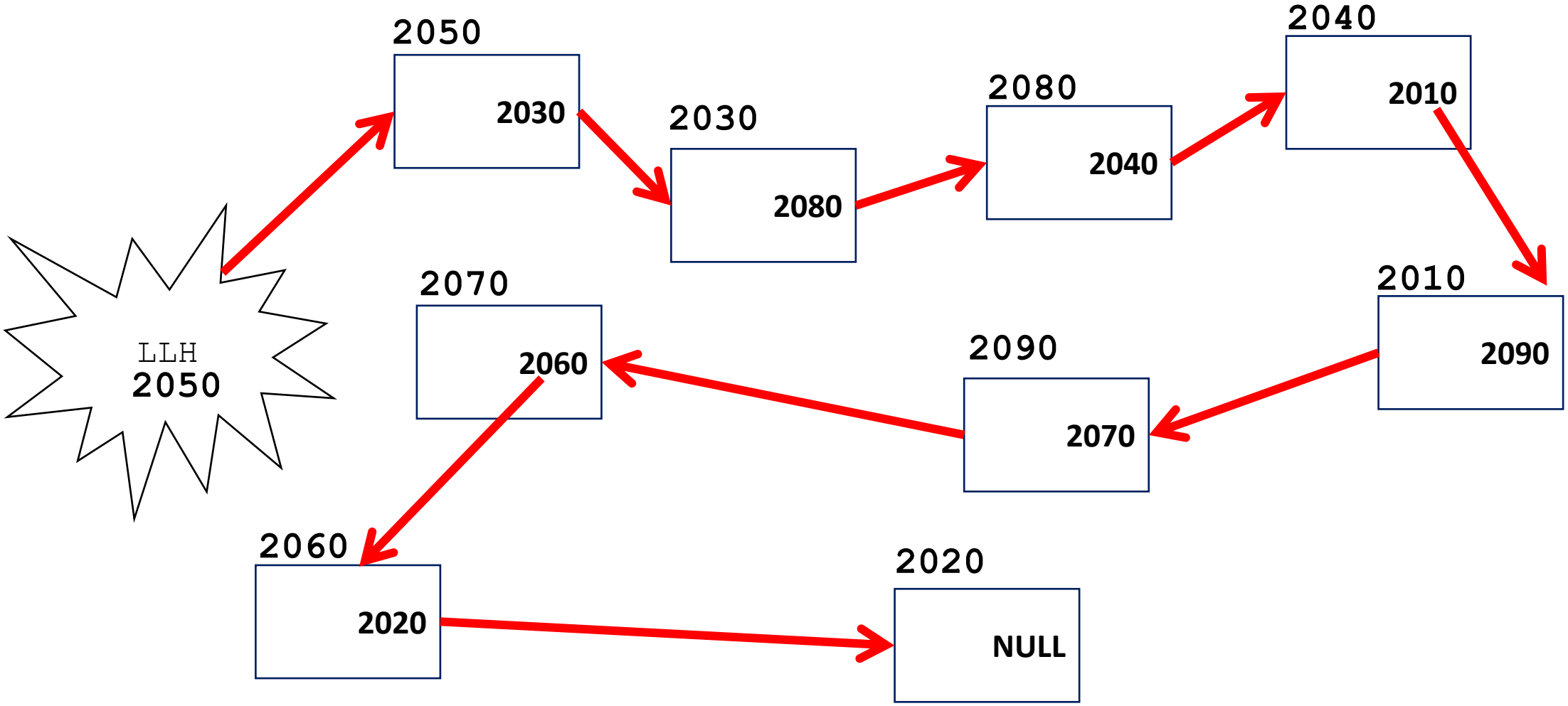
Linked List – Add another link



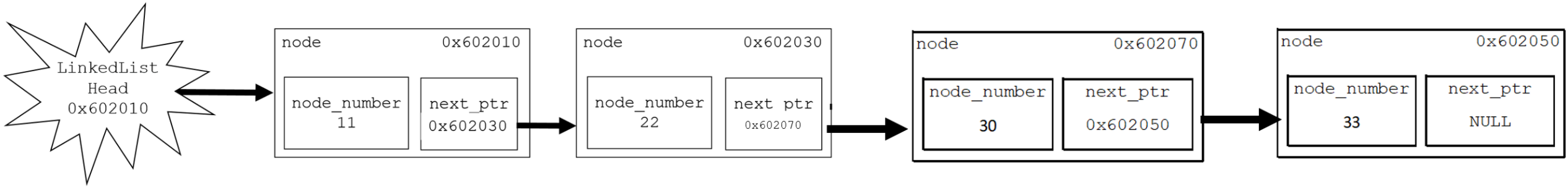
Linked List – Add Node to End

```
struct node *TempPtr, *NewNode;

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

TempPtr = LinkedListHead; // Start at the head

/* Traverse the linked list to find the end node */
while (TempPtr->next_ptr != NULL)
   TempPtr = TempPtr->next_ptr;

/* Change end node to point to new node */
TempPtr->next_ptr = NewNode;
```

```
TempPtr = LinkedListHead;

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



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



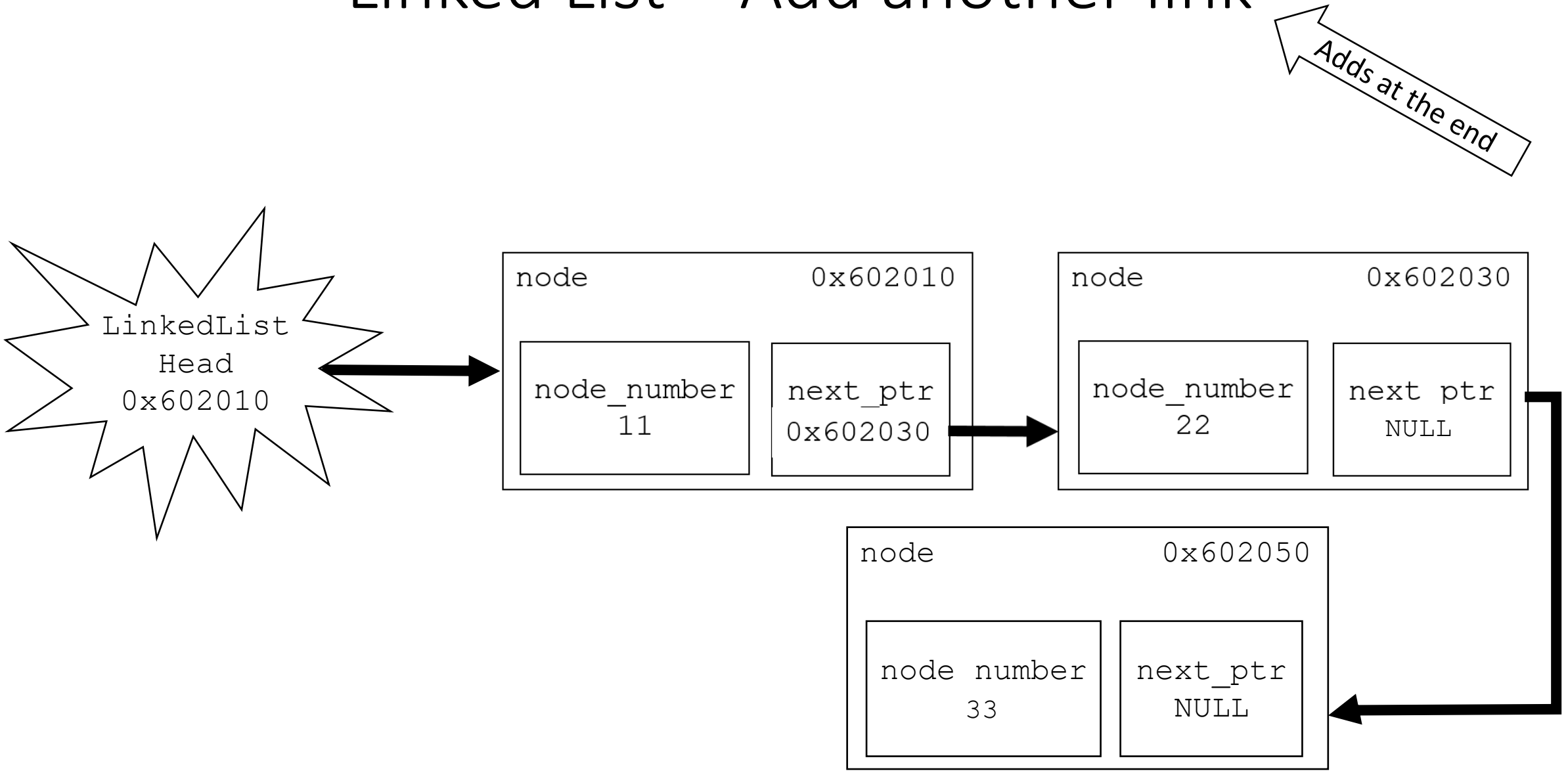

```
while (TempPtr->next_ptr != NULL)

TempPtr
--------
0x602010
0x602030
0x602070
0x602050
```

Final Value of TempPtr

```
while (TempPtr != NULL)

TempPtr
--------
0x602010
0x602030
0x602070
0x602050
NULL
```

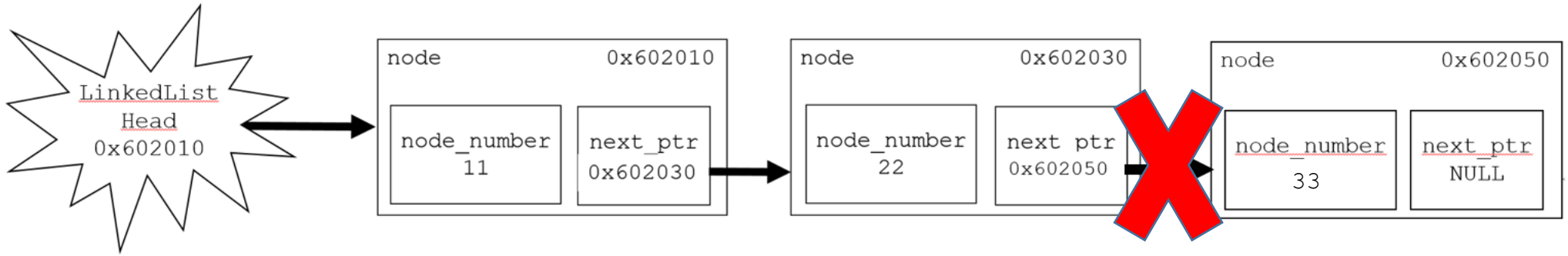
Final Value of TempPtr

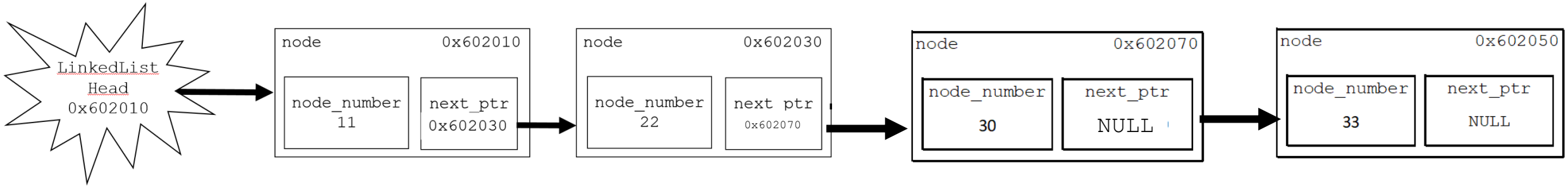
Linked List – Add another link



Linked List

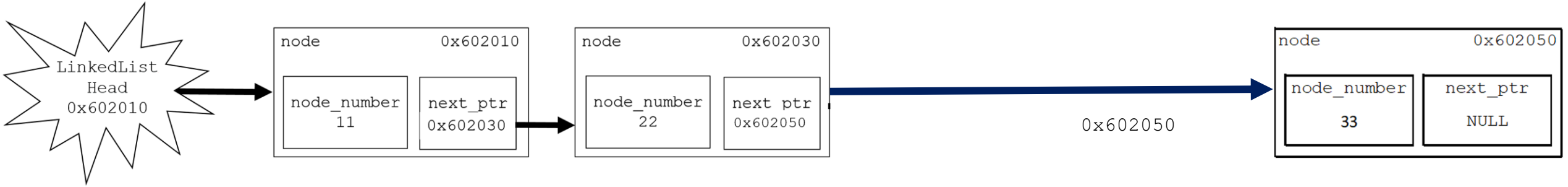
Inserting a node





Linked List – Insert Node

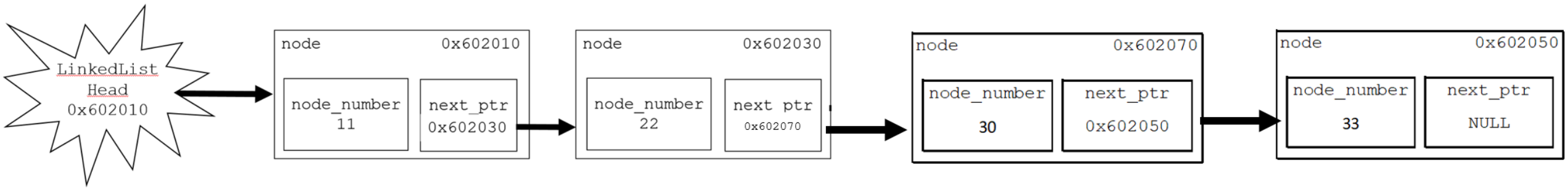
```
struct node *TempPtr, *NewNode, *PrevPtr;

PrevPtr = NULL;
TempPtr = LinkedListHead;

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




```
/* If PrevPtr is still NULL, then we are at the start of the list */
if (PrevPtr == NULL)
{
      LinkedListHead = NewNode;
}
else
{
      PrevPtr->next_ptr = NewNode;
}
```

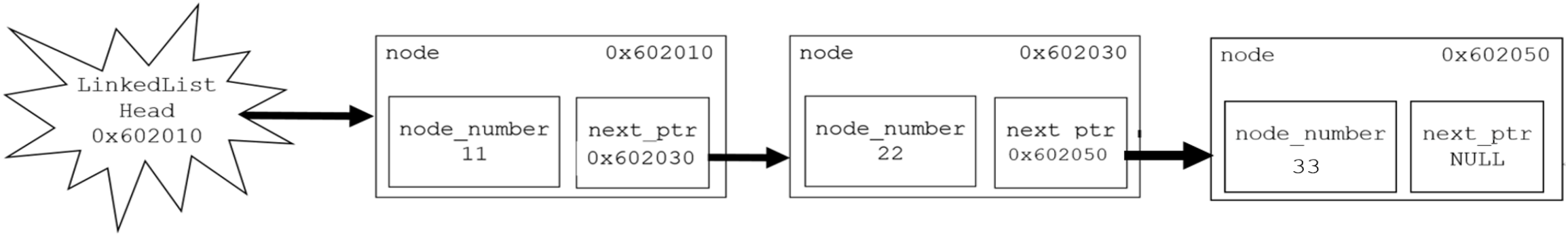
```
NewNode = malloc(sizeof(struct node));
NewNode->node_number = NodeNumberToInsert;
NewNode->next_ptr = TempPtr;
```

```
Enter the node number to insert : 30


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 – Insert Node At start

```
struct node *TempPtr, *NewNode, *PrevPtr;

PrevPtr = NULL;
TempPtr = LinkedListHead;

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

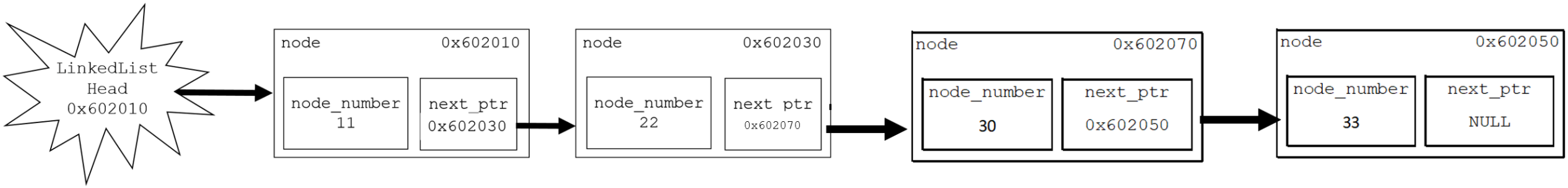



```
if (PrevPtr == NULL)
{
      LinkedListHead = NewNode;
}
else
{
      PrevPtr->next_ptr = NewNode;
}
```

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

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;
}
```

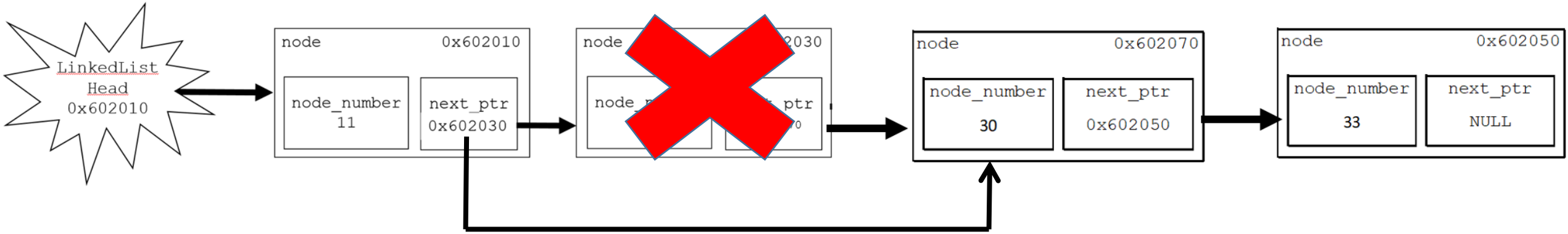


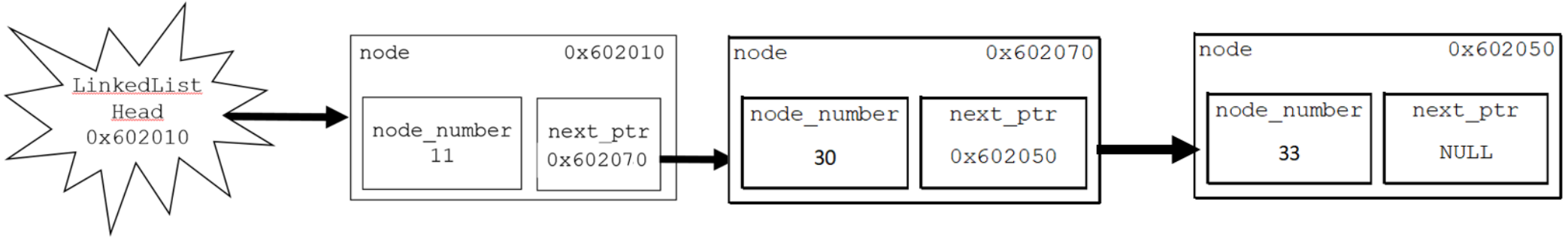
Linked List – Count Nodes

```
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





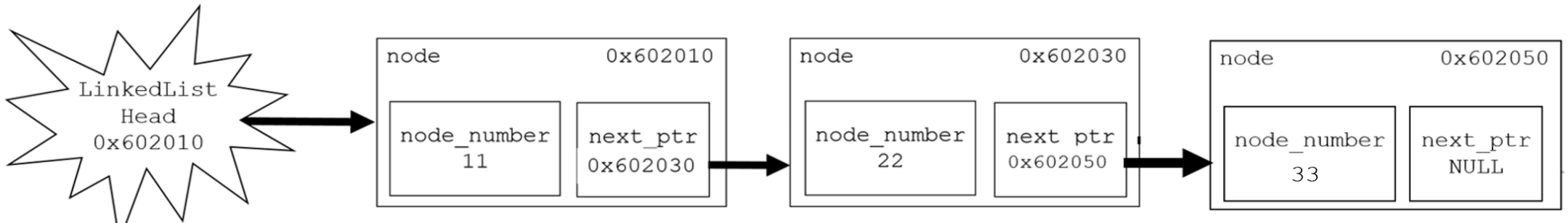
```
Node Number 11          Node Address 0x602010     Node Next Pointer 0x602030

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
```

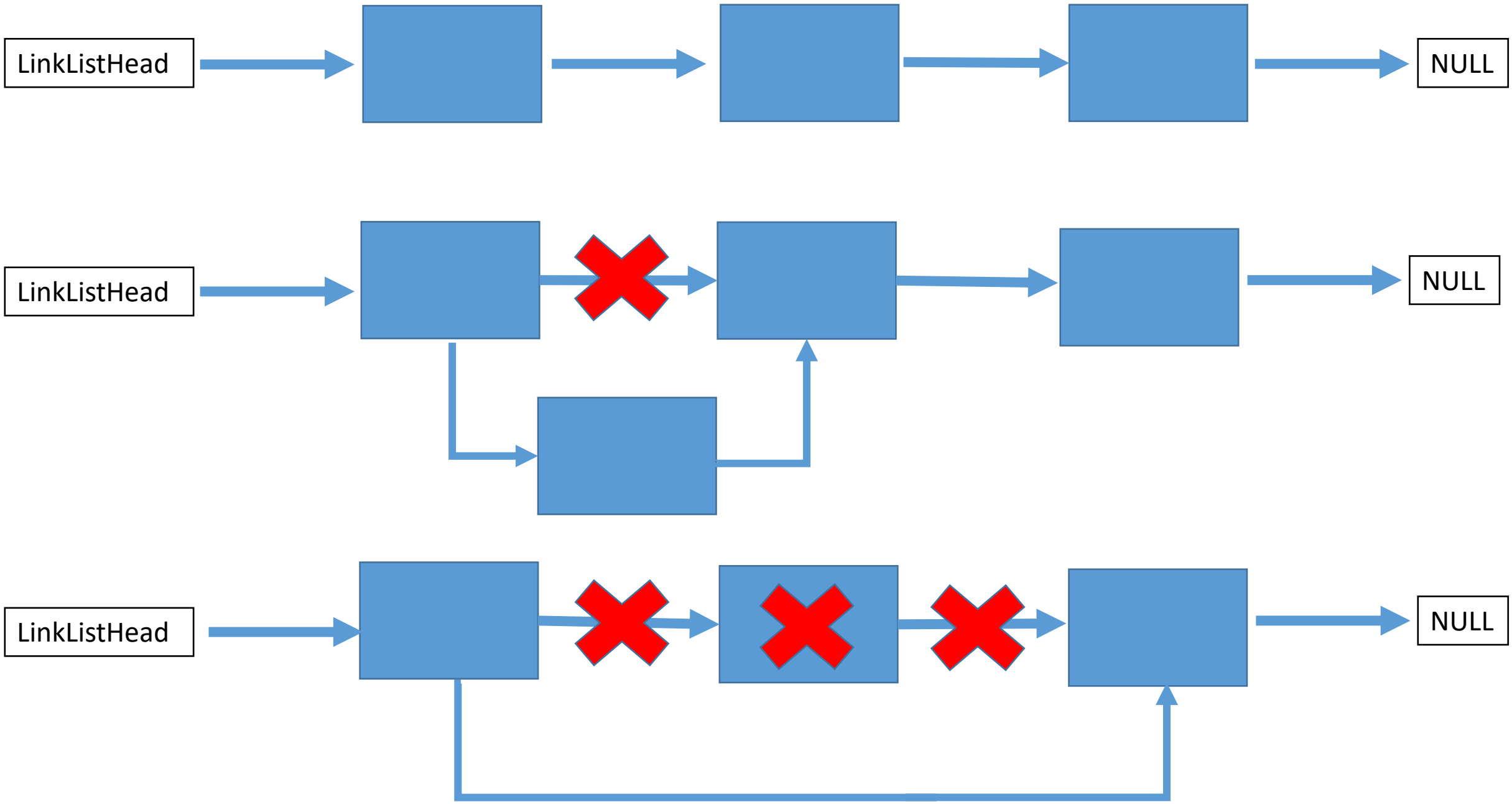
```
Enter your choice : 4

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