LINKED LIST

Linked List Operations

01
Basics

02
Inserting

03
Searching

04
Deleting

Linked List

- The linked list is a linear data structure where each node has two parts.

- 1. Data
- 2. Reference to the next node

Data & Reference to the next node

- int age;
- char name[20];

- Head Node - Starting node of a linked list.

- Last Node - Node with reference pointer as NULL.

Selecting a data type for the linked list

- Data - it can be any data type. int,char,float,double etc.
- Reference Part - It will hold the address of the next node. So, it is a type pointer.
- Here, we need to group two different data types (**heterogeneous**).

Which data type is used to group the different data types?

- That is a structure. So, every node in a linked list is a structure data type.

```
struct node
{
    int data;
    struct node *next;
};
```

where,

data - used to store the integer information.

struct node *next - It is used to refer to the next node. It will hold the address of the next node.

Create and allocate memory for 3 nodes

```c
struct node
{
    int data;
    struct node *next;
};

struct node *head,*middle,*last;

head   = malloc(sizeof(struct node));
middle = malloc(sizeof(struct node));
last   = malloc(sizeof(struct node));
```

Assign values to each node

```c
head->data   = 10;
middle->data = 20;
last->data   = 30;
```

malloc(sizeof(struct node))

- This statement declares a variable named *head* as type of *pointer to struct node*.(The structure is possibly defined somewhere before this code snippet)

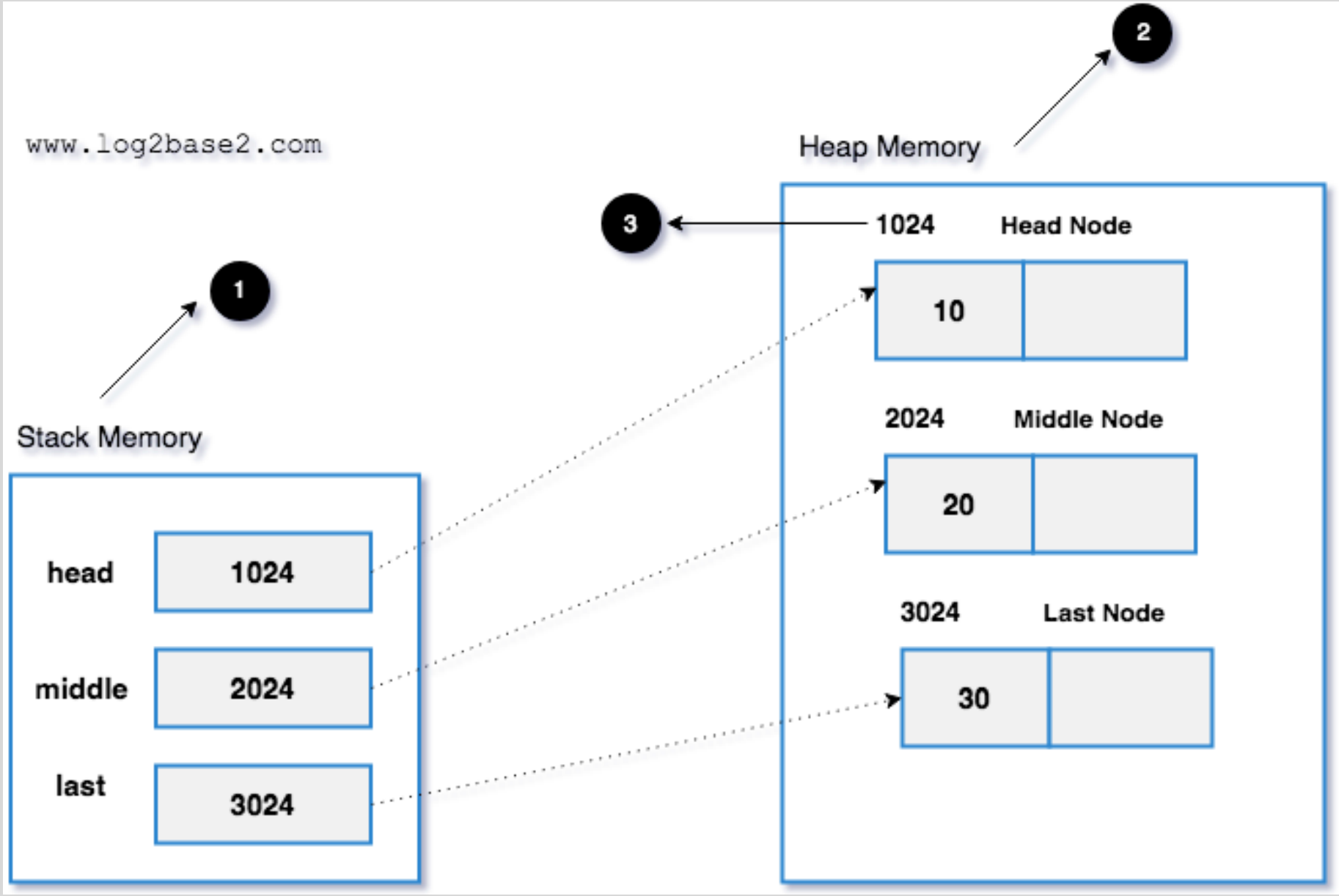
- *malloc()* is a standard library function in C that allocates memory of size equal to the size of *head structure* in the heap region during runtime and returns a *void pointer* to that location.

```
struct node
{
    int data;
    struct node *next;
};

struct node *head,*middle,*last;

head   = malloc(sizeof(struct node));
middle = malloc(sizeof(struct node));
last   = malloc(sizeof(struct node));
```

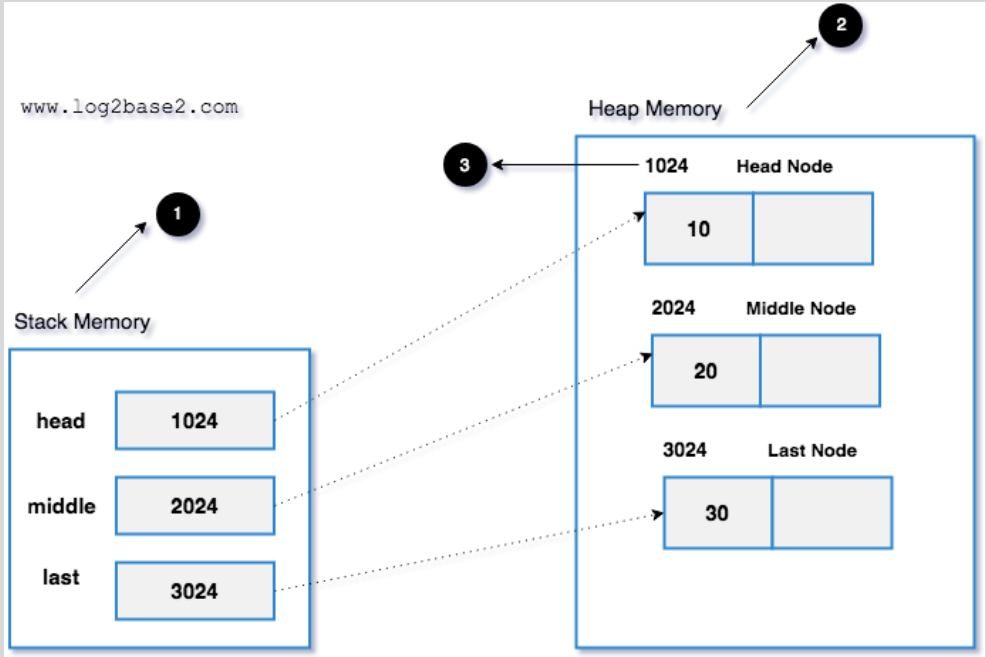
Dynamically allocates memory of size enough to hold *node structure* and type cast it to *pointer to node structure*, then assigns to *head* variable.


www.log2base2.com
Heap Memory
2
3
1024
Head Node
1
10
2024
Middle Node
Stack Memory
20
3024
Last Node
head
1024
30
middle
2024
last
3024

1. Stack memory stores all the local variables and function calls (static memory).
Example: int a = 10;

2. Heap memory stores all the dynamically allocated variables.
Example: int *ptr = malloc(sizeof(int)); Here, memory will be allocated in the heap section. And the ptr resides in the stack section and receives the heap section memory address on successful memory allocation.

3. Address of the dynamic memory which will be assigned to the corresponding variable.



Stack and Heap

- Stack is used for static memory allocation and Heap for dynamic memory allocation, both stored in the computer's RAM .

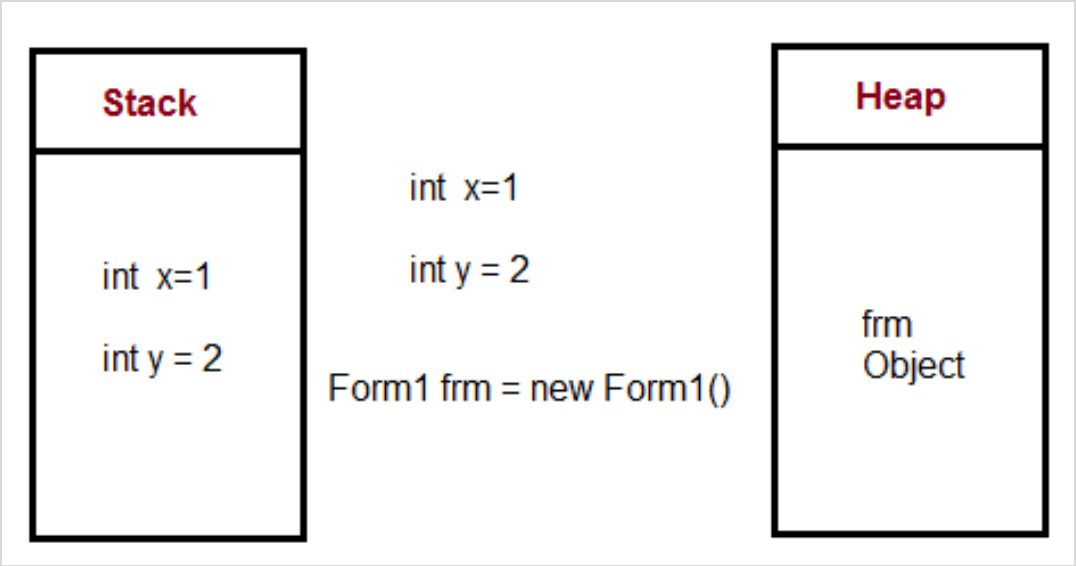
Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and it's allocation is dealt with when the program is compiled.

The stack is always reserved in a LIFO order, the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack, freeing a block from the stack is nothing more than adjusting one pointer.

Variables allocated on the heap have their memory allocated at run time and accessing this memory is a bit slower, but the heap size is only limited by the size of virtual memory .

Element of the heap have no dependencies with each other and can always be accessed randomly at any time. You can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time.

Stack and Heap



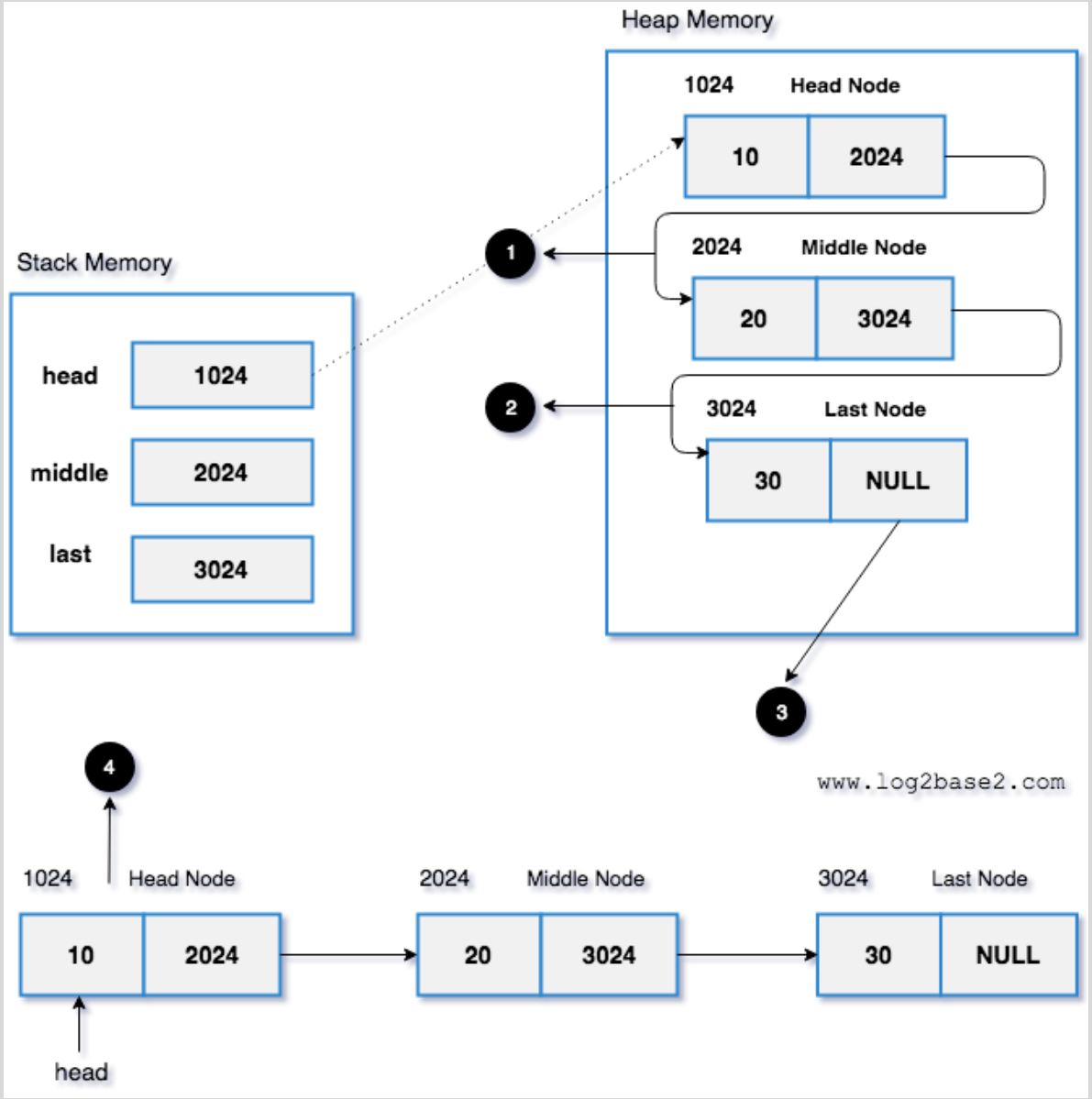
You can use the stack if you know exactly how much data you need to allocate before compile time and it is not too big. You can use heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.

In a multi-threaded situation each thread will have its own completely independent stack but they will share the heap. Stack is thread specific and Heap is application specific. The stack is important to consider in exception handling and thread executions.

Linking each nodes

- headnode->middlenode->lastnode->NULL

- head->next = middle;
- middle->next = last;
- last->next = NULL; //NULL indicates the end of the linked list



1. head => next = middle. Hence head => next holds the memory address of the middle node (2024).

2. middle => next = last. Hence middle => next holds the memory address of the last node (3024).

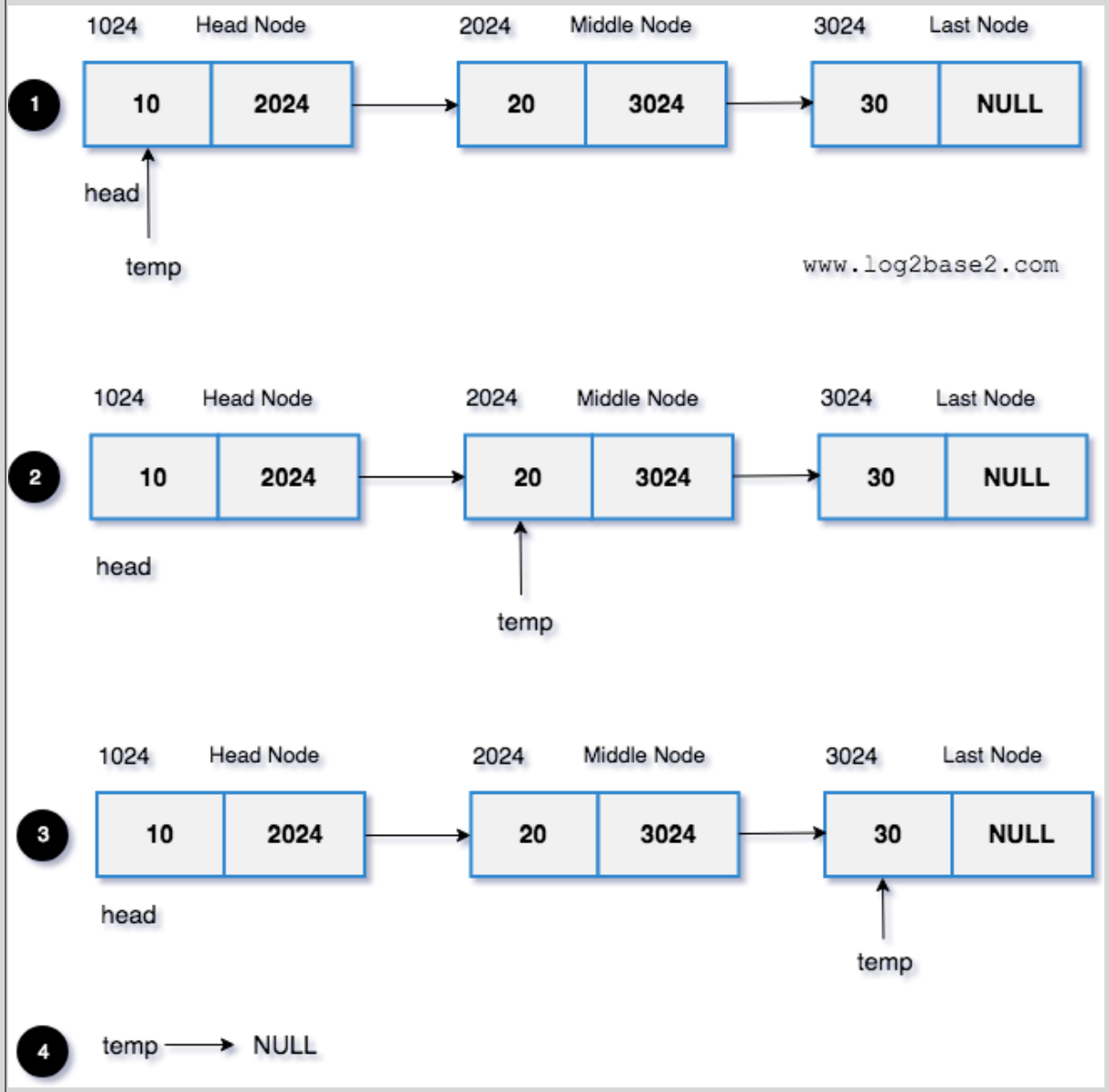
3. last => next = NULL which indicates it is the last node in the linked list.

4. The simplified version of the heap memory section.

Print each node data in a linked list

- To print each node's data, traverse the linked list till the end.

- 1. Create a temporary node(temp) and assign the head node's address.
- 2. Print the data which present in the temp node.
- 3. After printing the data, move the temp pointer to the next node.
- 4. Do the above process until we reach the end.



1. temp points to the head node. temp => data = 10 will be printed. temp will point to the next node (Middle Node).

2. temp != NULL. temp => data = 20 will be printed. Again temp will point to the next node (Last Node).

3. temp != NULL. temp => data = 30 will be printed. Again temp will point to the next node which is NULL.

4. temp == NULL. Stop the process we have printed the whole linked list.

Why do we need to use the temp node instead head?

If we use the head pointer instead of the temp while printing the linked list, we will miss the track of the starting node. (After printing the data head node will point the NULL).

To avoid that, we should not change the head node's address while processing the linked list. We should always use a temporary node to manipulate the linked list.

Code

```
struct node *temp = head;

while(temp != NULL)
{
    printf("%d ",temp->data);
    temp = temp->next;
}
```

Sample Linked List Implementation

```
int main() {
struct node  //node structure
 {
    int data;
    struct node *next;  };
 //declaring nodes
 struct node *head,*middle,*last;
 //allocating memory for each node
 head   = malloc(sizeof(struct node));
 middle = malloc(sizeof(struct node));
 last   = malloc(sizeof(struct node));
 //assigning values to each node
 head->data   = 10;
 middle->data = 20;
 last->data   = 30;
 //connecting each nodes. head->middle->last
 head->next   = middle;
 middle->next = last;
 last->next   = NULL;
 //temp is a reference for head pointer.
 struct node *temp = head;
 //till the node becomes null, printing each nodes data
 while(temp != NULL)
 {
    printf("%d->",temp->data);
    temp = temp->next;   }
 printf("NULL");
 return 0;
}
```

INSERTING A NODE AT THE BEGINNING OF A LINKED LIST

Inserting

- The new node will be added at the beginning of a linked list.

- Assume that the linked list has elements: 20 => 30 => 40 => NULL
- If we insert 100, it will be added at the beginning of a linked list.
- After insertion, the new linked list will be
- 100 => 20 => 30 => 40 => NULL

Algorithm

- 1. Declare a head pointer and make it as NULL.
- 2. Create a new node with the given data.
- 3. Make the new node points to the head node.
- 4. Finally, make the new node as the head node.

Declare a head pointer and make it as NULL

```
struct node
{
    int data;
    struct node *next;
};

struct node *head = NULL;
```

Create a new node with the given data.

```
void addFirst(struct node **head, int val)
{
    //create a new node
        struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;
    }
```

Make the newnode points to the head node

```
void addFirst(struct node **head, int val)
{
    //create a new node
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;
        newNode->next = *head;
    }
```

Make the new node as the head node.

```
void addFirst(struct node **head, int val)
{
      //create a new node
      struct node *newNode = malloc(sizeof(struct node));
      newNode->data = val;
      newNode->next = *head;
            *head = newNode;

}
```

Visual Representation

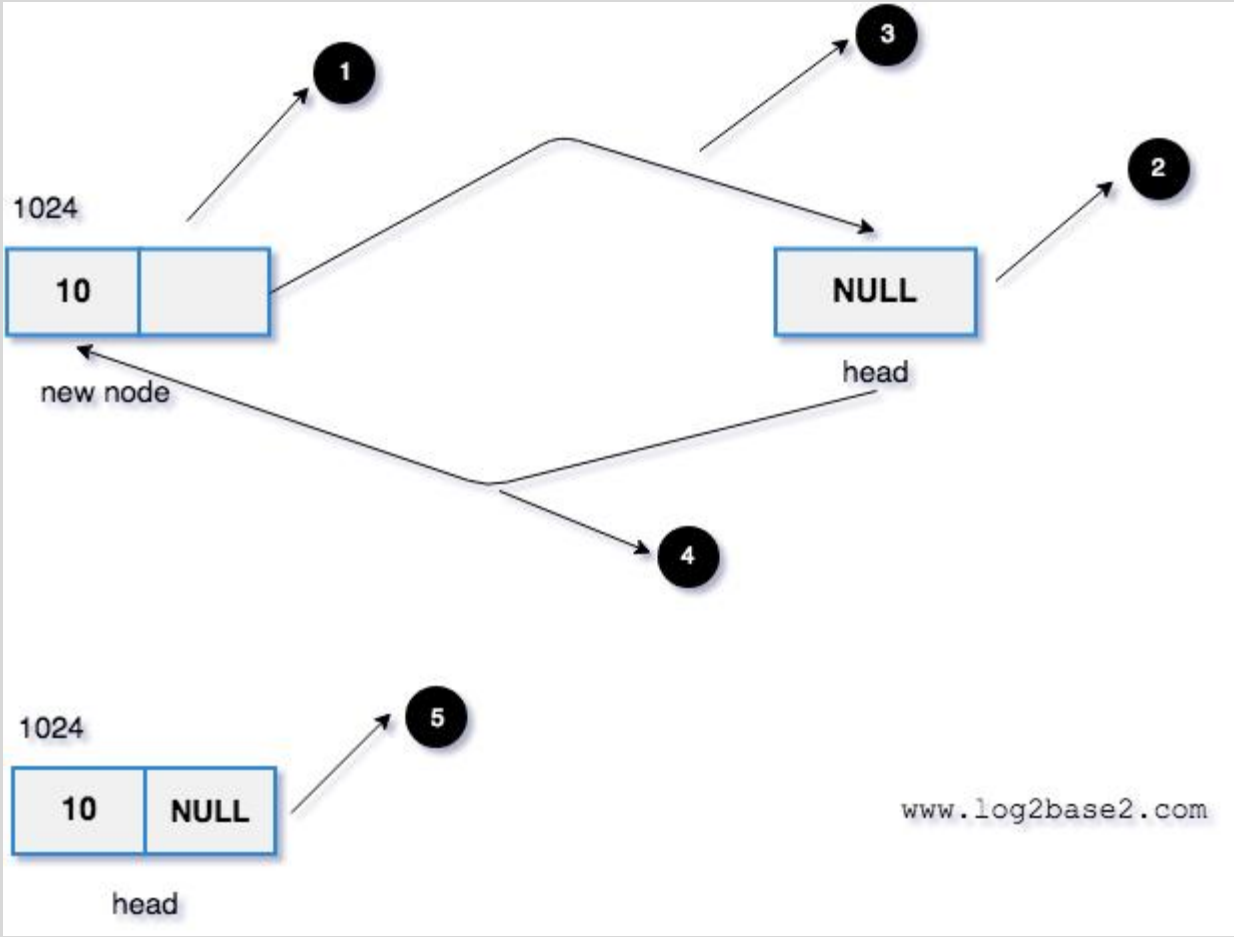
1. Allocated node with data as 10.

2. Head points to NULL.

3. New node -> next points to the head which is NULL. So newnode->next = NULL.

4. Make the head points to the new node. Now, the head will hold the address of the new node which is 1024.

5. Finally, the new linked list.



Visual Representation

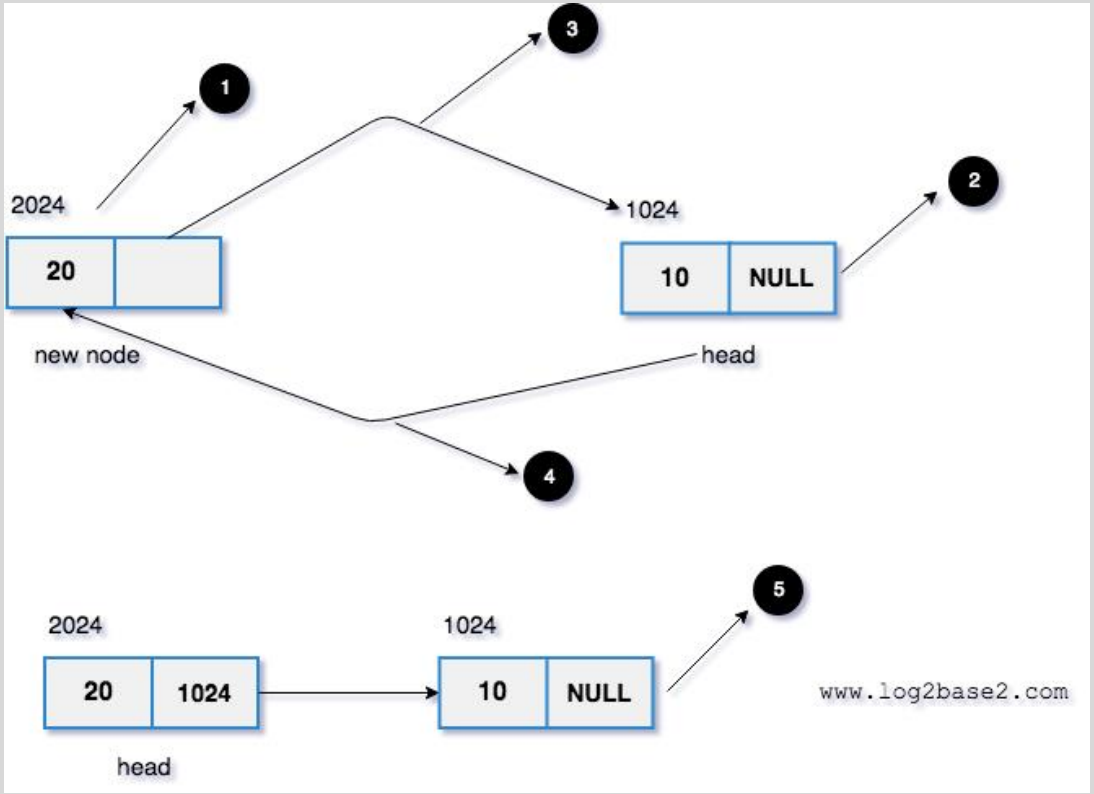
1. Allocated node with data as 20.

2. Head points to the memory address 1024 (It has only one node. 10->NULL).

3. New node -> next points to the head which is 1024. So newnode->next = 1024 (10->NULL) will be added back to the new node.

4. Make the head points to the new node. Now, the head will hold the address of the new node which is 2024

5. Finally, the new linked list.



Visual Representation

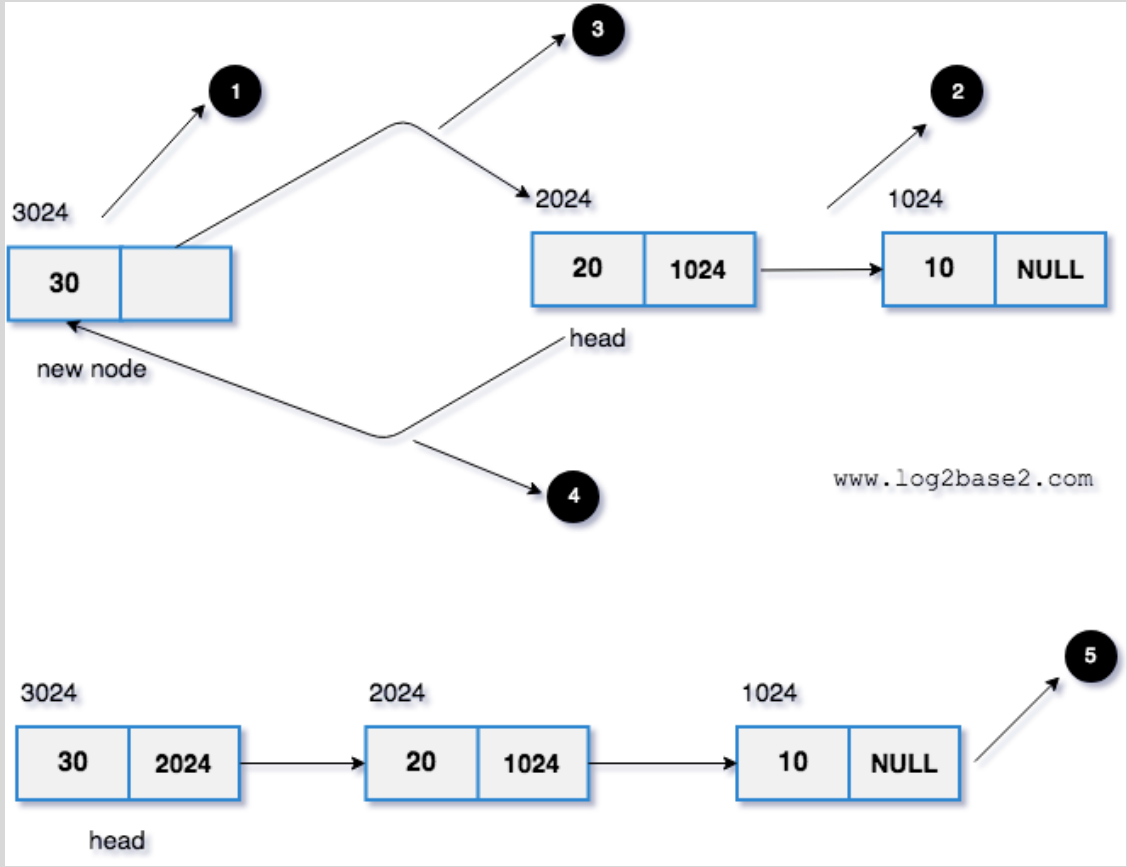
1. Allocated node with data as 30.

2. Head points to the memory address 2024 (It has two nodes. 20->10->NULL).

3. New node -> next points to the head which is 2024. So newnode->next = 2024 (20->10->NULL) will be added back to the new node.

4. Make the head points to the new node. Now, the head will hold the address of the new node which is 3024.

5. Finally, the new linked list.



Summary

- LL Introduction
- Insertion

Definition

```
struct node
{
    int data;
    struct node *next;
};
```

To point the front and rear node

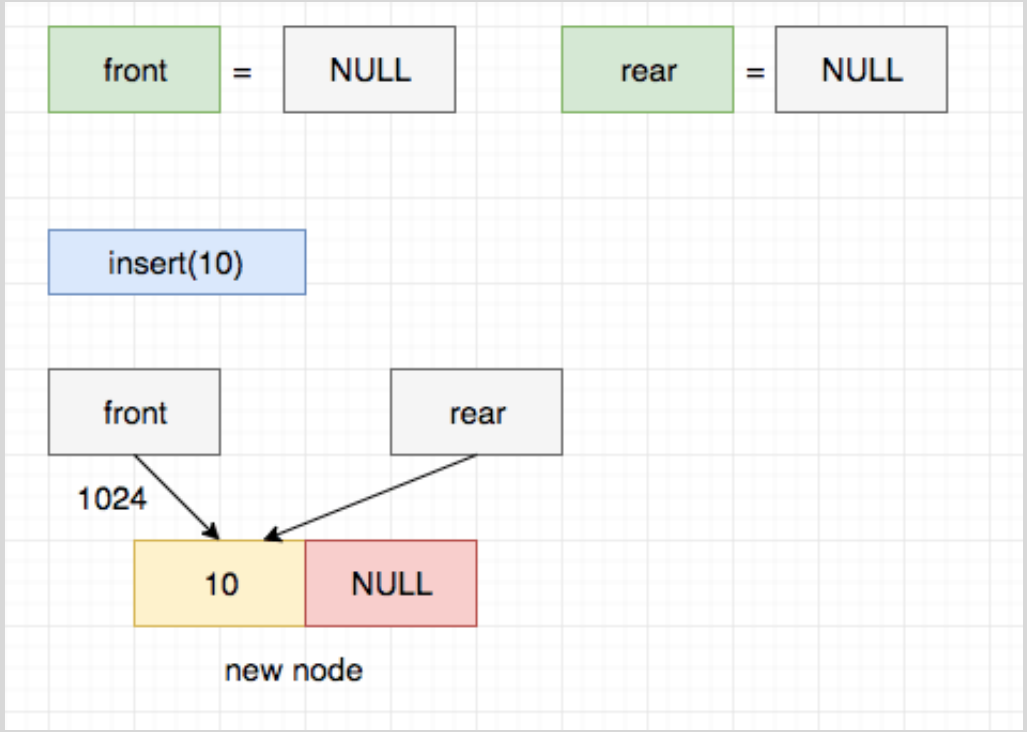
```
struct node *front = NULL, *rear = NULL;
```

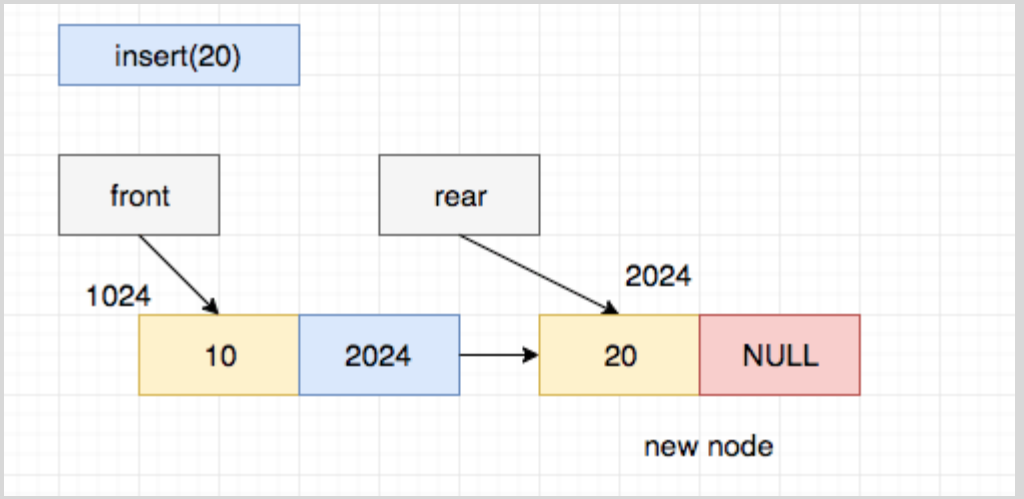
Enqueue function

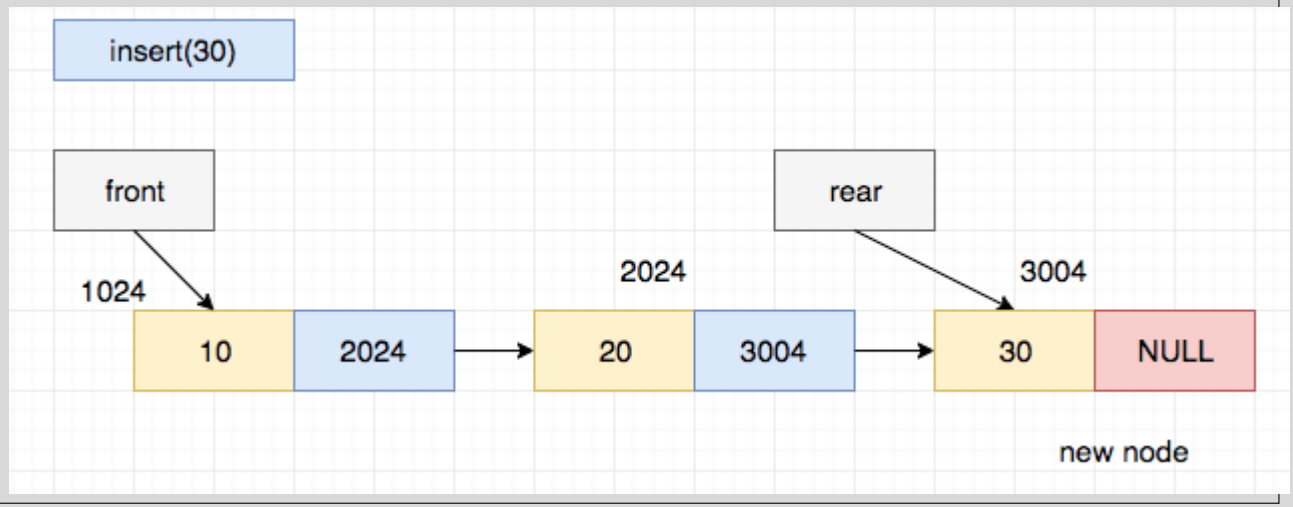
- Enqueue function will add the element at the end of the linked list.
- Using the rear pointer, we can track the last inserted element.
- 1. Declare a new node and allocate memory for it.
- 2. If front == NULL,
- make both front and rear points to the new node.
- 3. Otherwise,
- add the new node in rear->next.
- make the new node as the rear node. i.e. rear = new node

Enqueue

```
enqueue(int val)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;
    newNode->next = NULL;
    //if it is the first node
    if(front == NULL && rear == NULL)
        //make both front and rear points to the new node
        front = rear = newNode;
    else
    {
        //add newnode in rear->next
        rear->next = newNode;
        //make the new node as the rear node
        rear = newNode;
    } }
```


front
=
NULL
rear
=
NULL
insert(10)
front
rear
1024
10
NULL
new node


insert(20)
front
rear
1024
2024
10
2024
20
NULL
new node


insert(30)
front
rear
1024
2024
3004
10
2024
20
3004
30
NULL
new node

Dequeue function

- Dequeue function will remove the first element from the queue.
- 1.Check whether the queue is empty or not
- 2.If it is the empty queue (front == NULL)
- We can't dequeue the element.
- 3.Otherwise,
- Make the front node points to the next node. i.e front = front->next;
- if front pointer becomes NULL, set the rear pointer also NULL.
- Free the front node's memory.

Dequeue

```
dequeue()
{
    //used to freeing the first node after dequeue
    struct node *temp;

    if(front == NULL)
        printf("Queue is Empty. Unable to perform dequeue\n");
    else
    {
        //take backup
        temp = front;
        //make the front node points to the next node
        //logically removing the front element
        front = front->next;

        //if front == NULL, set rear = NULL
        if(front == NULL)
            rear = NULL;

        //free the first node
        free(temp);
    }

}
```

Why do we need to use the temp node instead head?

- If we use the head pointer instead of the temp while printing the linked list, we will miss the track of the starting node. (After printing the data head node will point the NULL).

- To avoid that, we should not change the head node's address while processing the linked list. We should always use a temporary node to manipulate the linked list.

P#2 Linked List

Write a MENU DRIVEN C program to perform queue operations using single linked list.

Submission procedure remains same

Time is at 11 AM