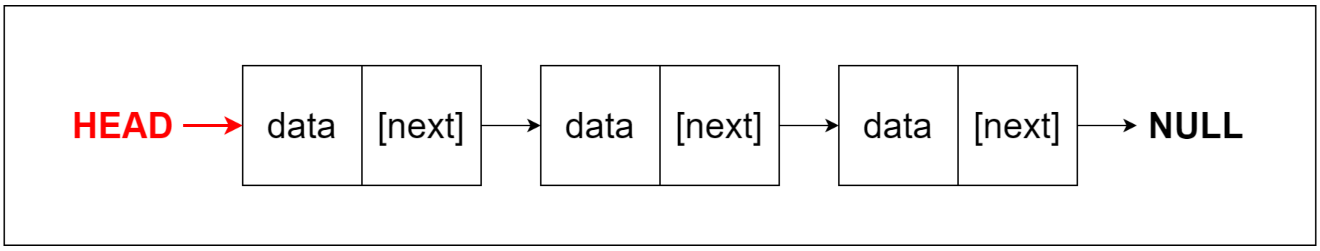
Types of Linked List

Let's explore the world of linked lists, like different ways to arrange your favorite toys. There are three main types of linked lists, each with its own special features:

1. **Singly Linked List**: This is like a line of toys where each toy holds hands with the next one. You can move only forward in this line. Each toy knows about the toy in front of it, but not the one behind.
2. **Doubly Linked List**: Imagine toys holding hands with both the one in front and the one behind. This allows you to move both forward and backward in the line. Each toy knows about its neighbors in both directions.
3. **Circular Linked List**: Picture a line of toys, but the last one is holding hands with the first one, forming a circle. You can keep going around the circle. This type is like a special kind of game where there's no end or beginning.

So, just like arranging your toys in different patterns, we can use these types of linked lists to store and manage data in various ways.

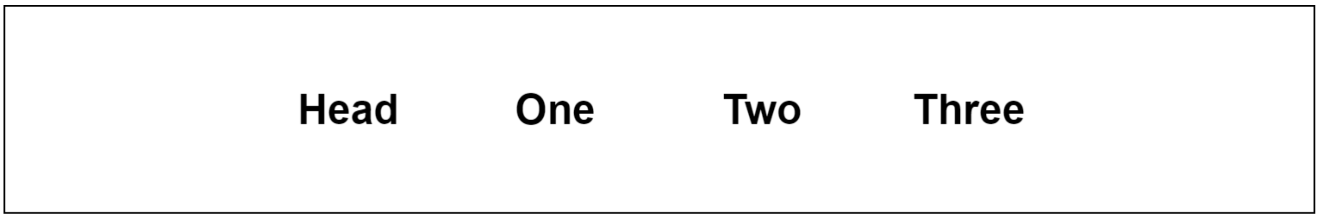
Singly Linked List



Imagine you have train cars, and each car holds a passenger and a map to the next station. In programming terms, we call these "nodes", and each node has two parts: one for the passenger's information (we'll call it "data"), and another for the map to the next station (we'll call it "next").

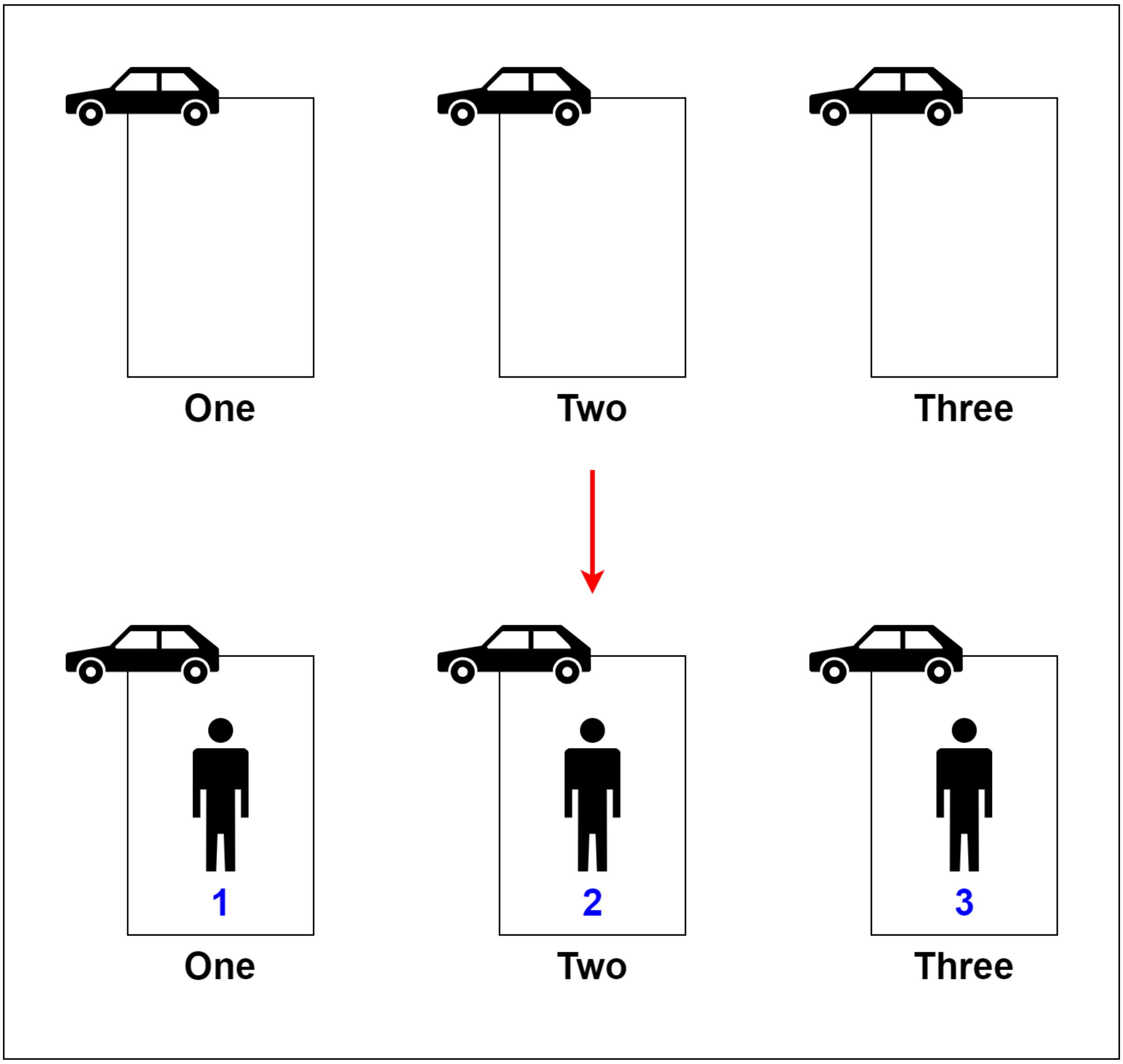
For example, let's create a small train with three cars. First, we prepare the train cars:

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

/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;
```



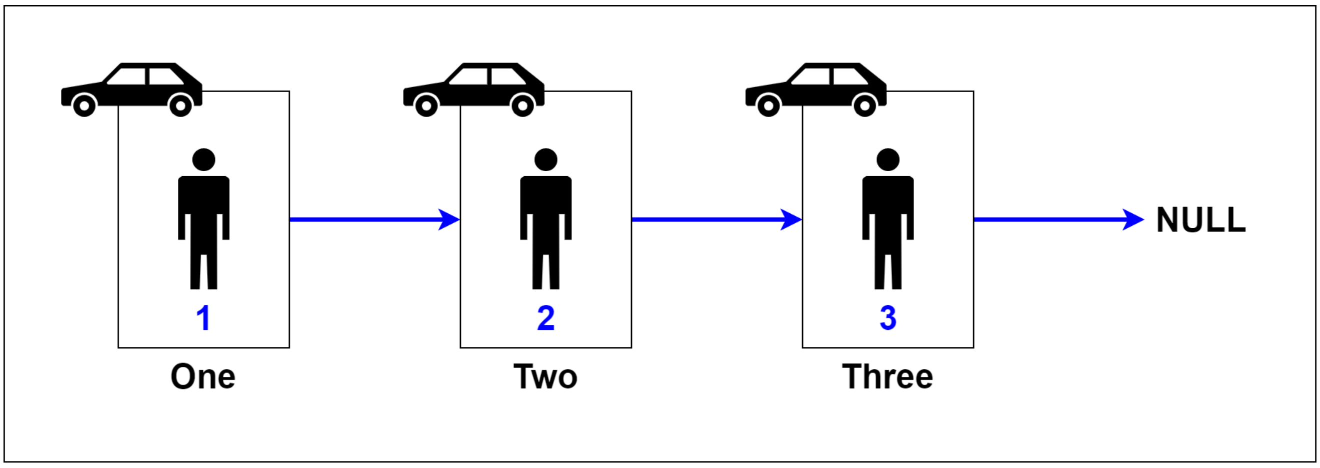
Then we allocate memory for each car and give them passengers:

```
/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;
```



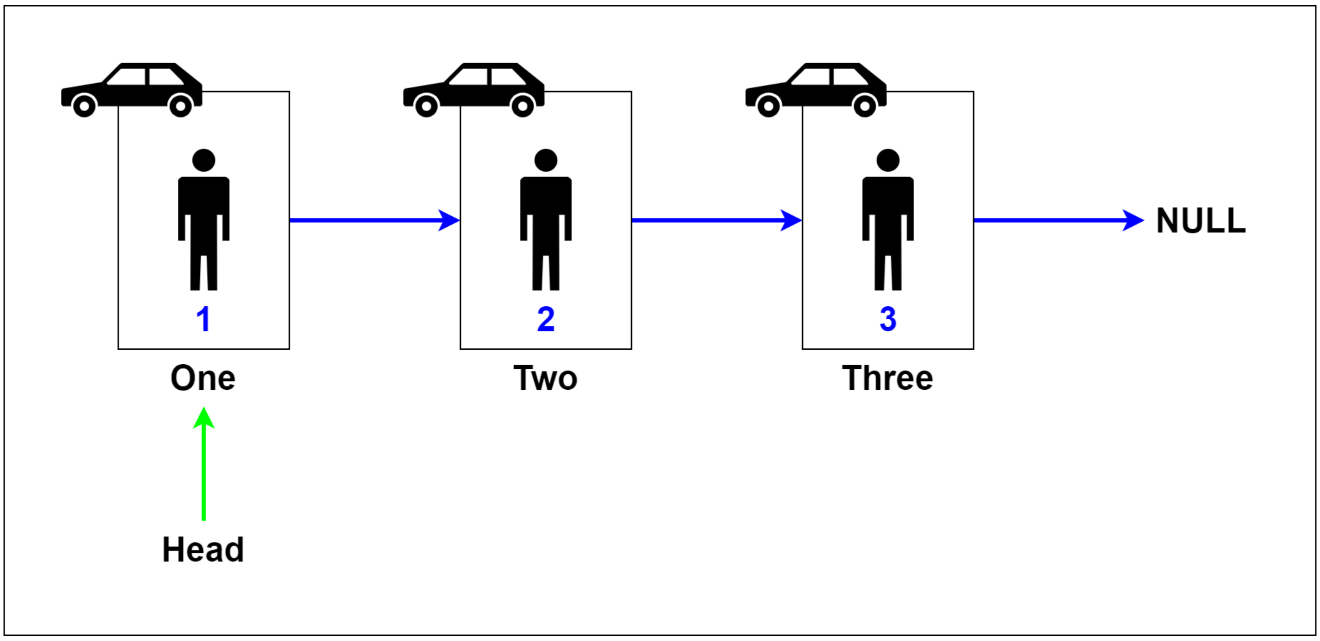
Now, we connect the train cars one after the other:

```
/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;
```



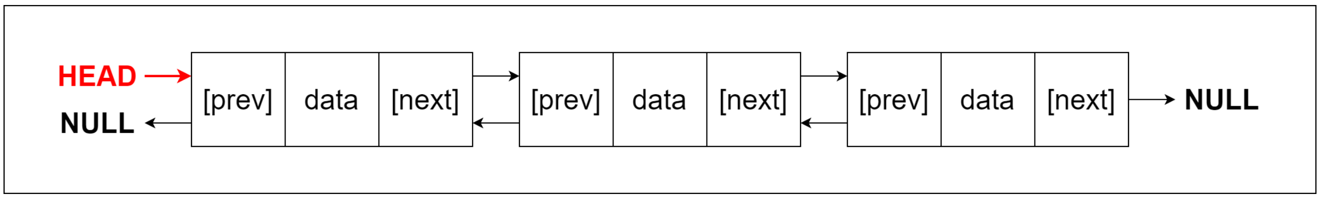
Finally, we mark the first car as the starting point of our train:

```
/* Save address of first node in head */
head = one;
```



And there you have it! A linked list where each car knows its passenger's information and the address of the next car on the track. This is how a singly linked list works, helping us organize and manage data in a linear fashion.

Doubly Linked List



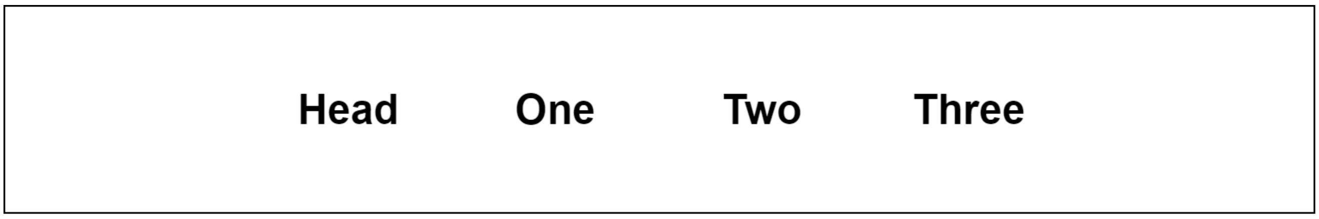
let's continue our journey into linked lists with a special kind called the "doubly linked list". It's like having a train track where each train car not only knows about the next station but also the previous one, allowing us to travel in both directions.

Imagine each train car still has a passenger and a map to the next station, but now they also have a map to the previous station. This way, you can move forward and backward in the train line.

In programming, we use the term "node" to represent each train car. And a node in a doubly linked list has three parts: one for the passenger's information ("data"), one for the map to the next station ("next"), and another for the map to the previous station ("prev").

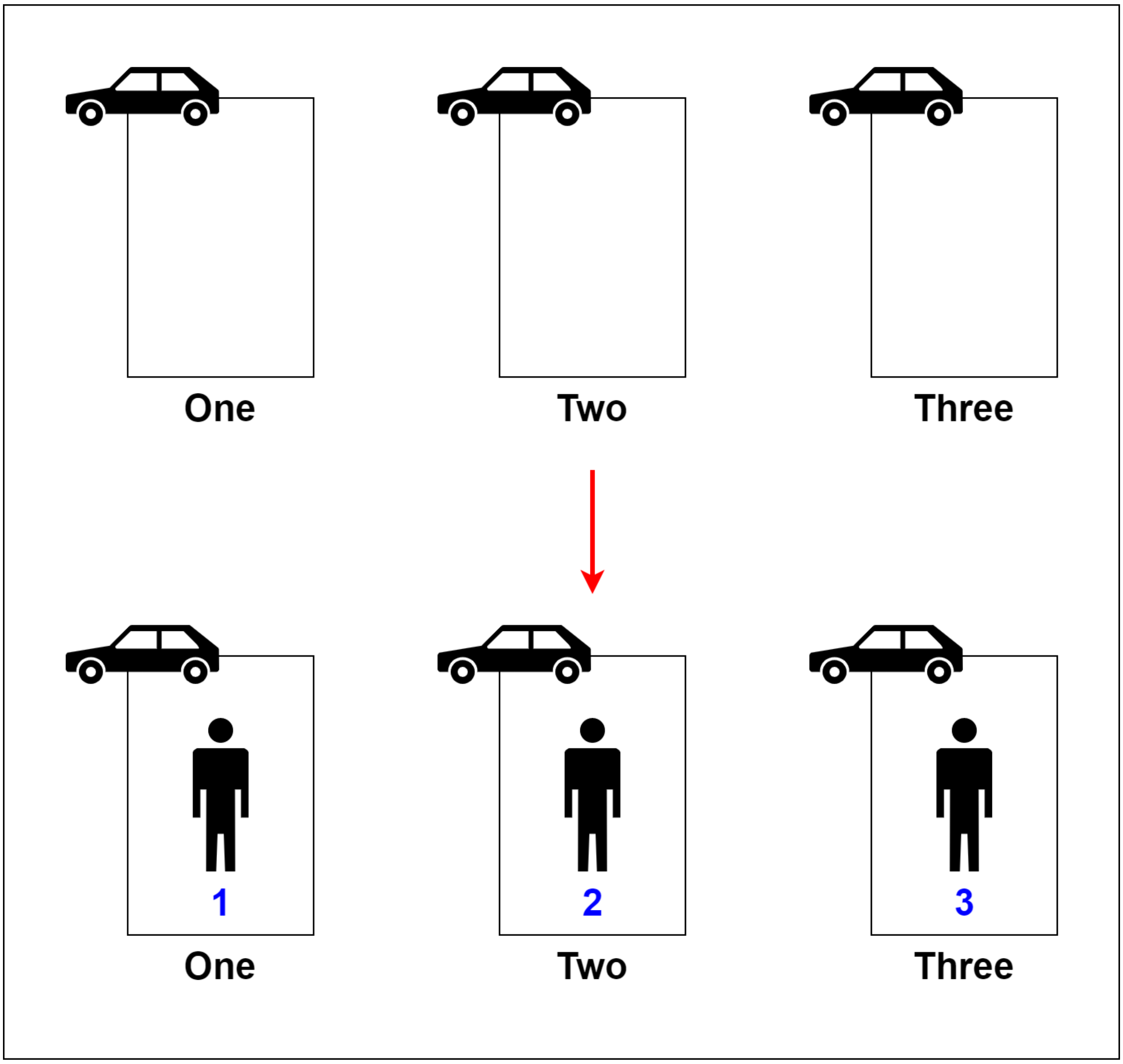
Let's create a little train with three cars to understand better:

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

/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;
```



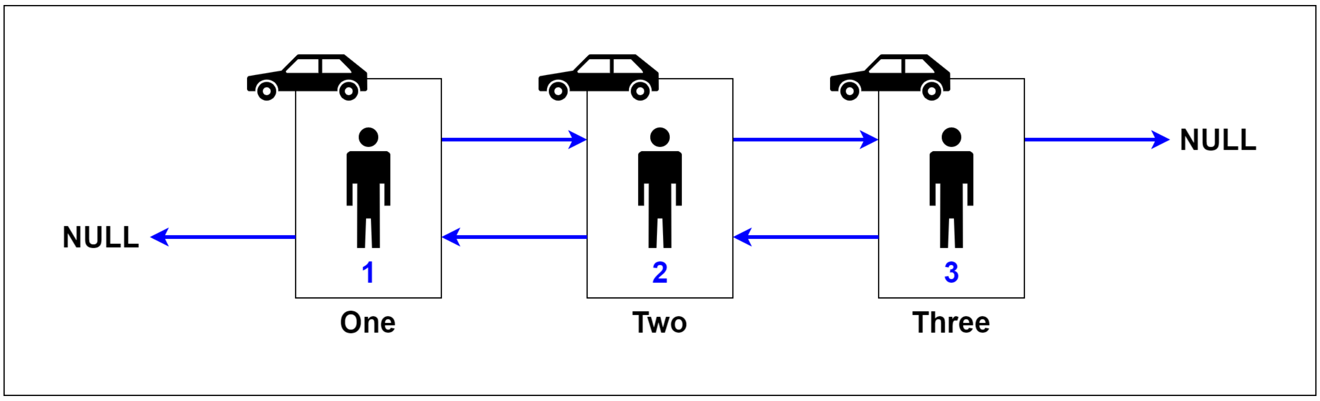
Then, we allocate memory for each car and give them passengers:

```
/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;
```



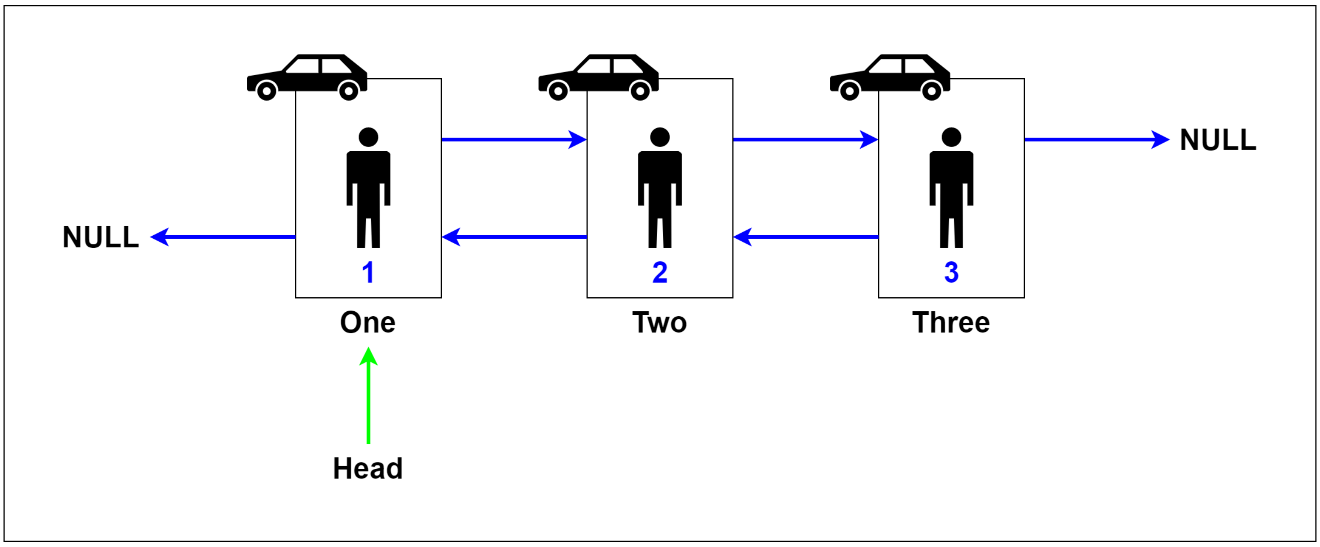
Now comes the interesting part: we connect the train cars not only forward but also backward:

```
/* Connect nodes */
one->next = two;
one->prev = NULL;

two->next = three;
two->prev = one;

three->next = NULL;
three->prev = two;
```



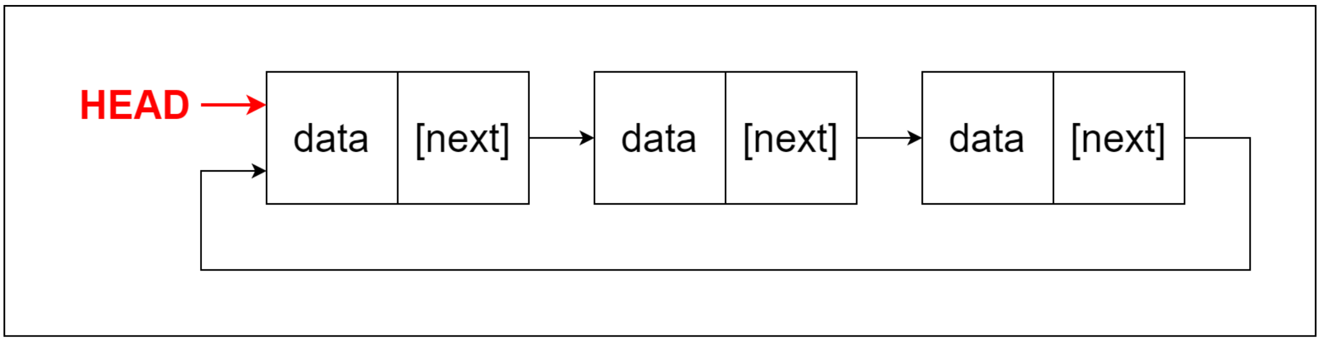
Finally, we mark the first car as the starting point of our doubly linked list:

```
/* Save address of first node in head */
head = one;
```



And that's how a doubly linked list works! Each train car remembers the station ahead and the one behind, giving us more flexibility to navigate through our data.

Circular Linked List



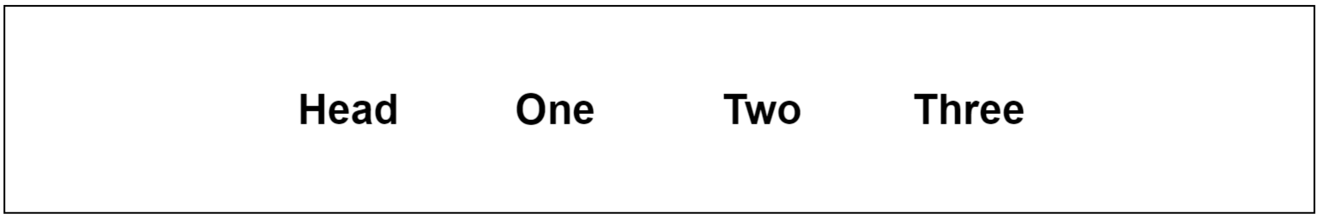
Circular linked list is like having a chain of friends holding hands in a circle – the last friend is holding hands with the first friend, creating a loop of connections.

Now, a circular linked list can be either of two types: singly linked or doubly linked. In a singly linked circular list, each friend only knows about the one in front of them. Imagine the last friend in the circle holding hands with the first friend to complete the loop.

On the other hand, in a doubly linked circular list, every friend knows both the friend in front and the friend behind. So, it's like they're all holding hands with each other in a big circle.

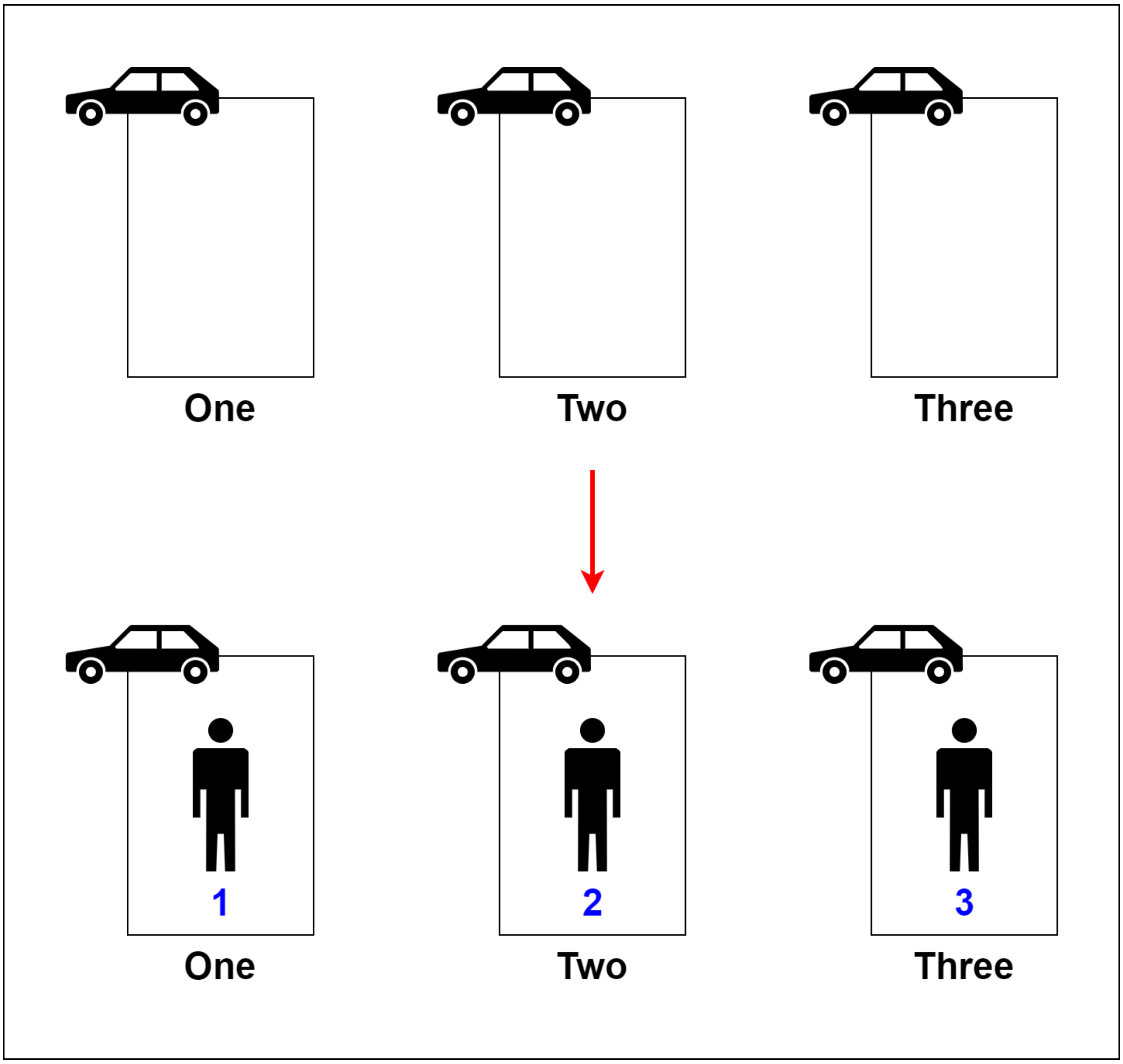
Now, let's create a simple example to understand this better. Imagine we have three friends, and we want to link them up in a circular singly linked list:

```c
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;
```



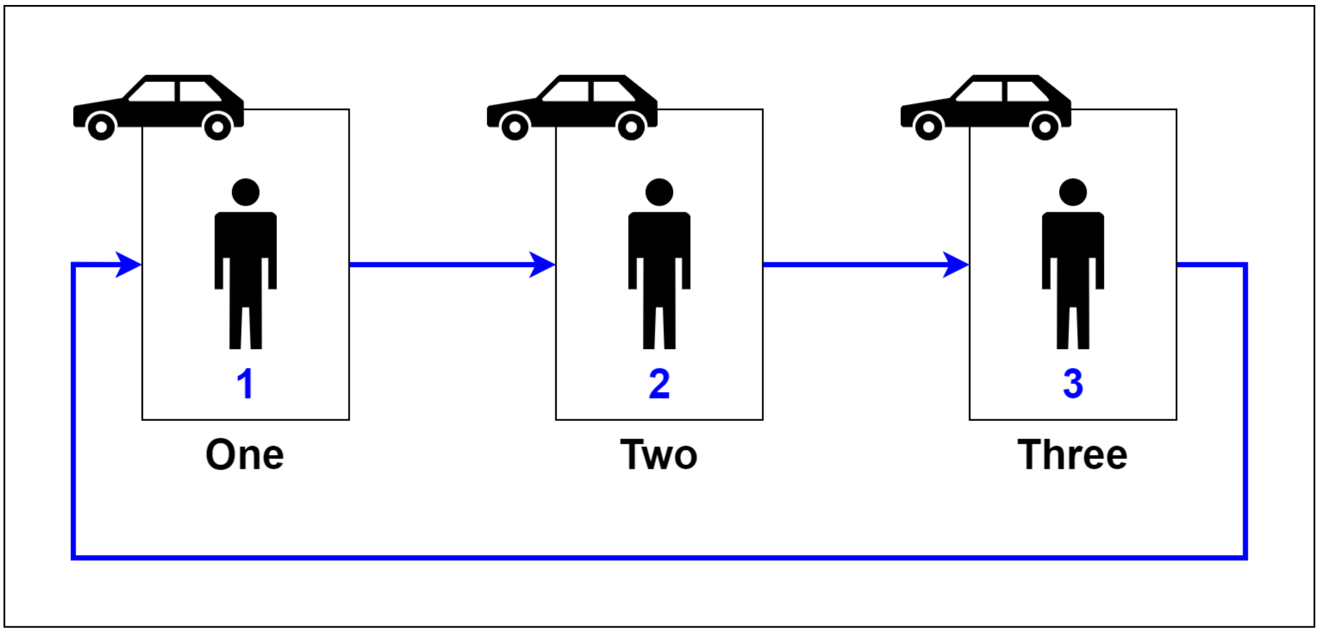
We'll give each friend a name (data):

```c
/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;
```



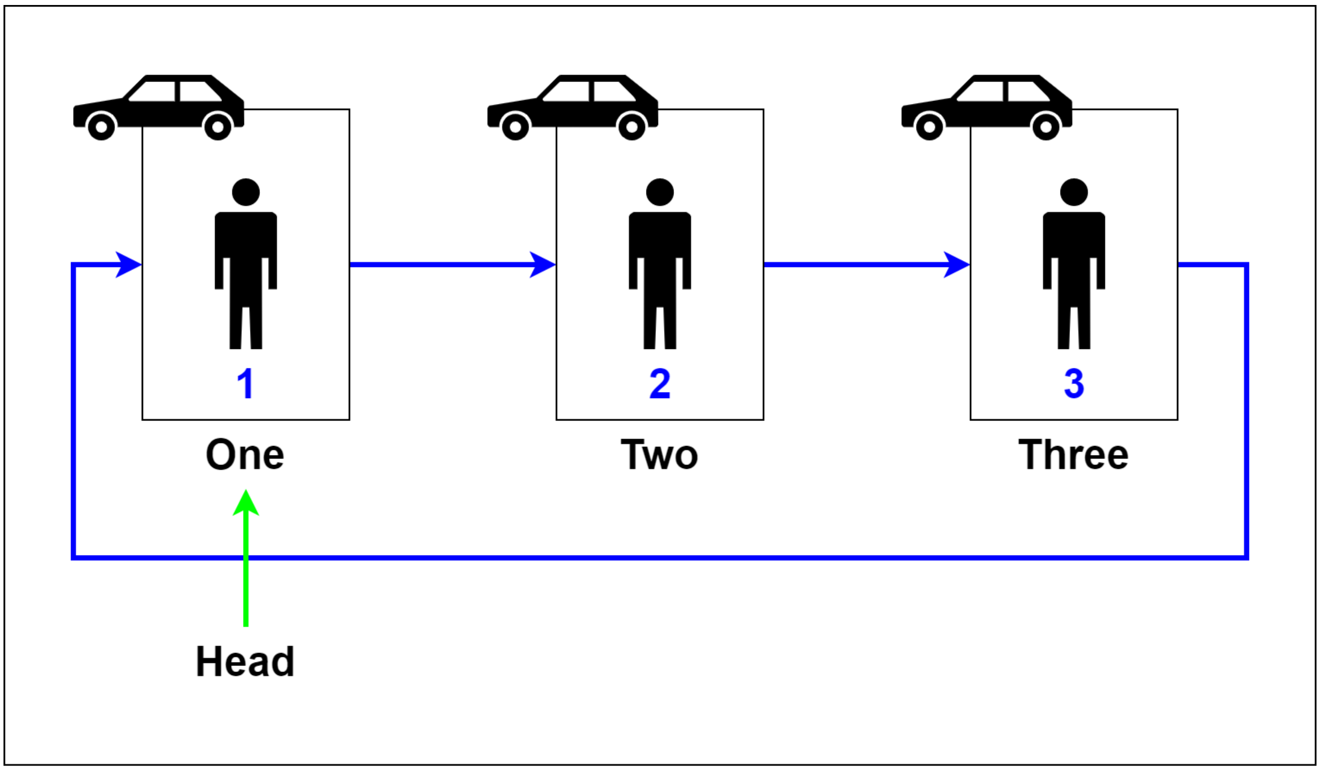
Now, let's connect them in a circular chain:

```
/* Connect nodes */
one->next = two;
two->next = three;
three->next = one;
```



And finally, we'll mark the first friend as the leader of our circular group:

```
/* Save address of first node in head */
head = one;
```



So, in a circular linked list, the last friend connects back to the first friend, making it a looping structure that can be pretty handy in various situations!