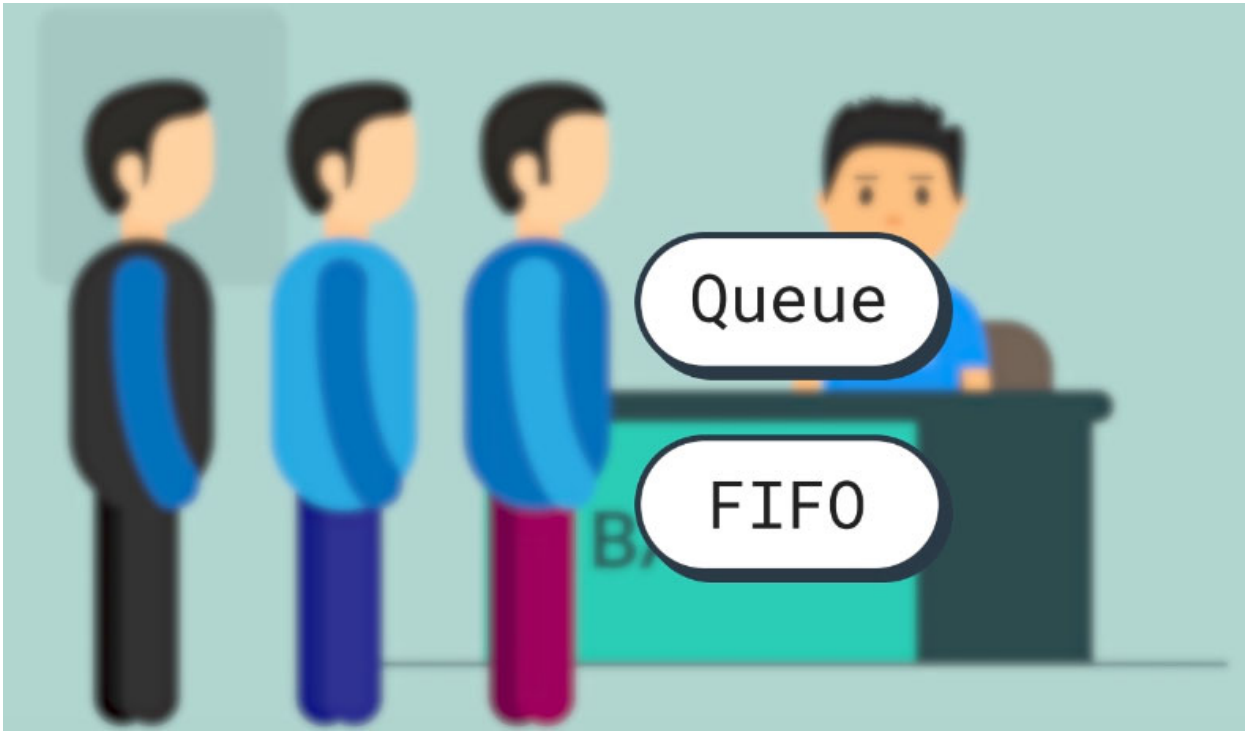
队列Queue

对于排队来说，我们奉行的是先进先出。

也就是所谓的First in，First out。



来对于队列来说，我们也可以使用数组和链表两种数据结构来实现。

这里先使用数组来实现队列，我们可以使用如下的代码来实现一个三个容量大小的数组。

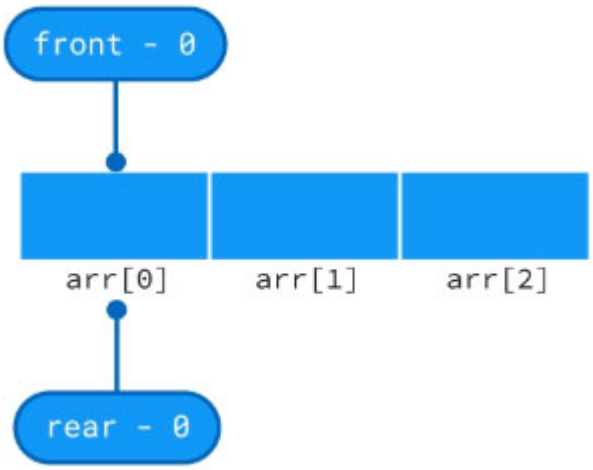
```
#define size 3
int arr[size];
```

其结构如下：

对于队列来说，它存在两个很重要的东西，一个是队头front，一个是队尾rear。

```
int front  = 0;
int rear   = 0;
```

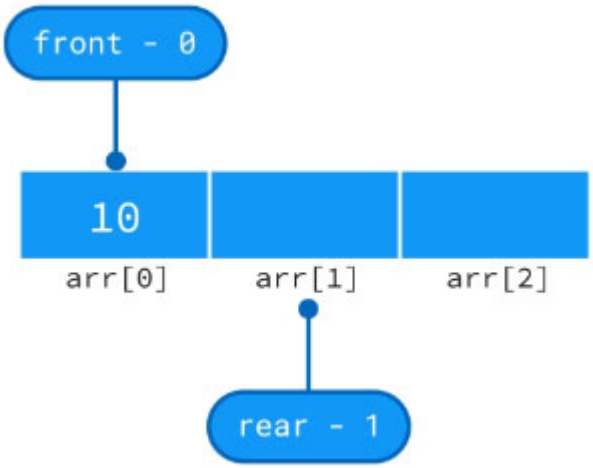
分别来指定队列里面最前面的元素和最后面的元素。对应的结果图如下：



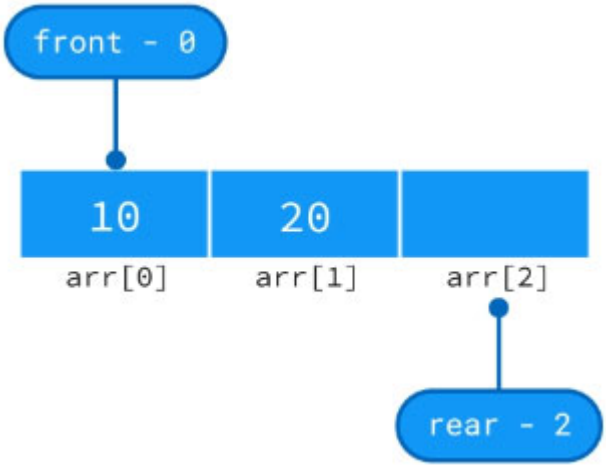
一开始队列是空的，为此我们要插入元素。每次入队的时候，队尾就要指向最后面的那个元素。而每次出队的时候，队尾不变，队头发生了变化，为此要修正队头。下面的代码就是队列的入队函数：

```
void enqueue(int val)
{
    if(isQueueFull())
        return;
    else
    {
        arr[rear] = val;
        rear++;
    }
}
```

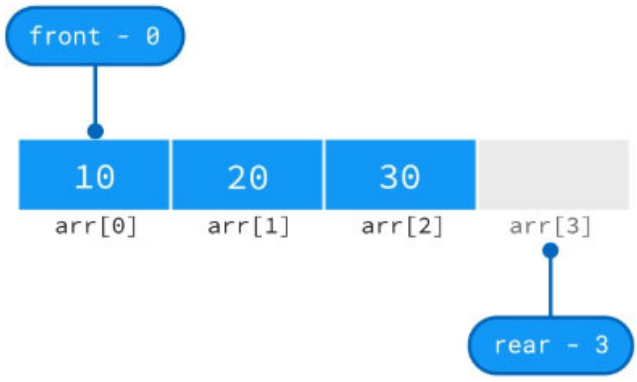
比如说我们调用enqueue(10)的时候，把10入队。要先判断队列是不是满了，满了就不能在入队了，否则就可以入队。入队的时候都是从队尾（索引为0）进入队列的，为此把队尾指向的 数组元素设置为传入的10，因为队伍增加了一个人，队尾增加1，指向下一个数组元素，rear变成了1。



调用enqueue(20)，队尾在索引为2的位置。



调用enqueue(30)后。队尾在索引为3的位置。



这时候队列满了，无法在压入其它元素了。队满的条件就是队尾rear==size。满了就返回1，不满就返回0。

```
int isQueueFull()
{
    if(rear == size)
        return 1;
```

```
    return 0;
}
```

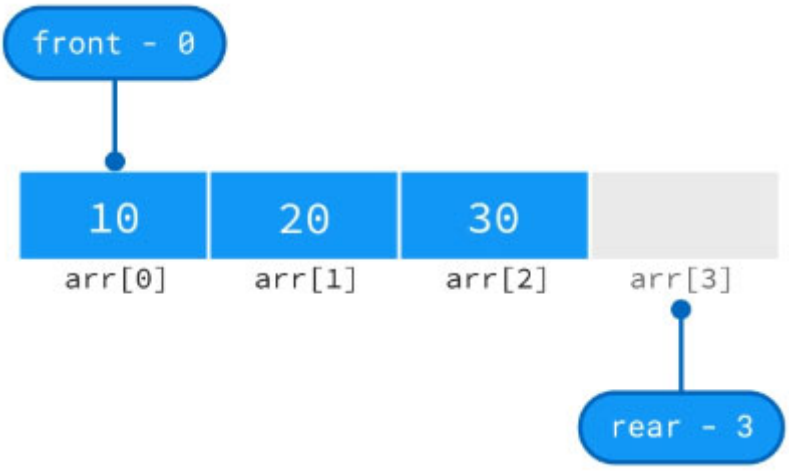
当队列满了，我们要考虑元素的出队了，dequeue()就是元素出队的函数，每次都是将队头的元素出队，也就是所谓的先进先出。dequeue()函数的返回值为int，表示我们要知道出去的那个元素的值是多少，以便做记录或者保存。

```
int dequeue()
```

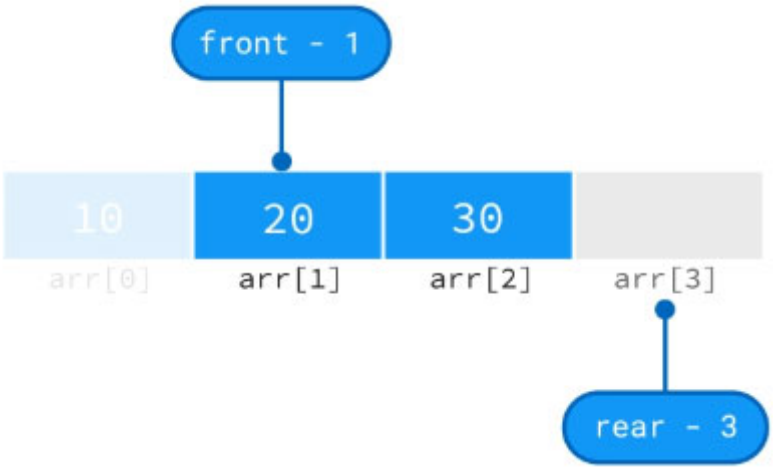
出队的完整代码如下，当队列不为空的时候，我们可以讲队头first指向的元素出队。

```
int dequeue()
{
    if(isQueueEmpty())
        return -1;
    else
        return arr[front++];
}
```

假定我们的队列是这样的：



第一次调用dequeue()方法的时候，我们要把队列的10删除，为此front++后指向了1，表示10已经无效了，相当于出队了。



当front等于rear的时候，表示队列空格了。

队空的代码如下：

```
int isQueueEmpty()
{
    if(rear==front)
        return 1;
    else
        return 0;
}
```

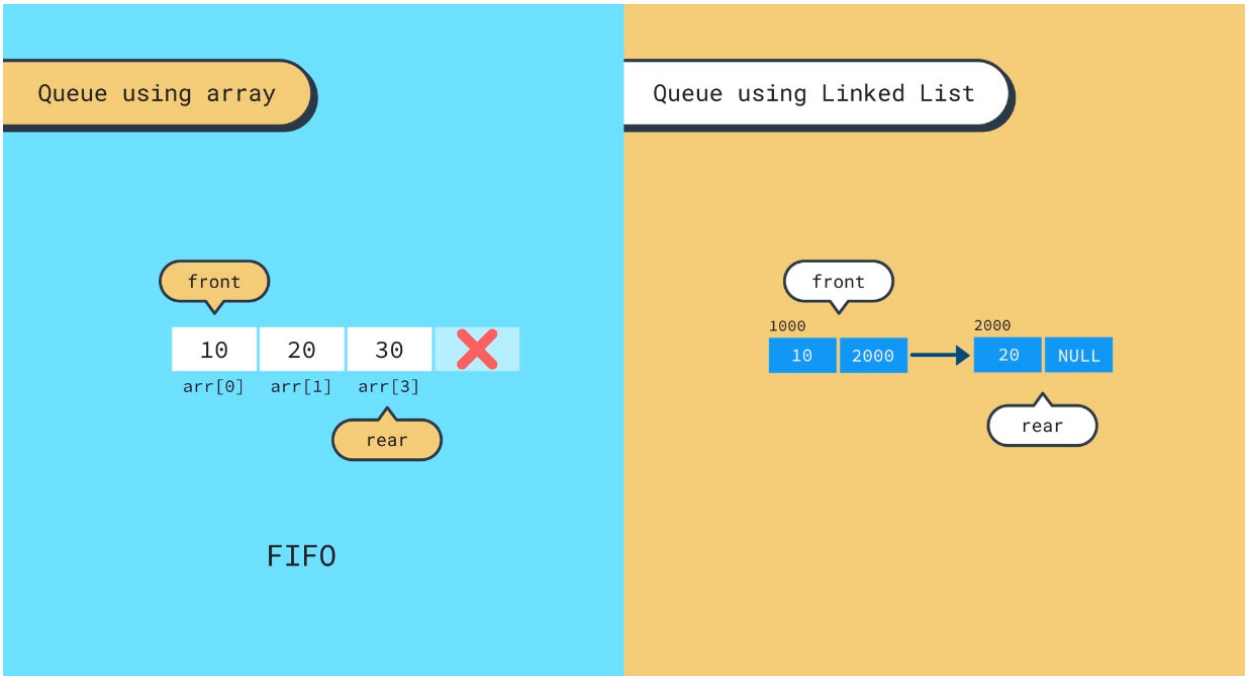
队列的最终的代码入下：

```
#include<stdio.h>

#define size 3
int arr[size];

int front  = 0;
int rear   = 0;

int isQueueFull()
```

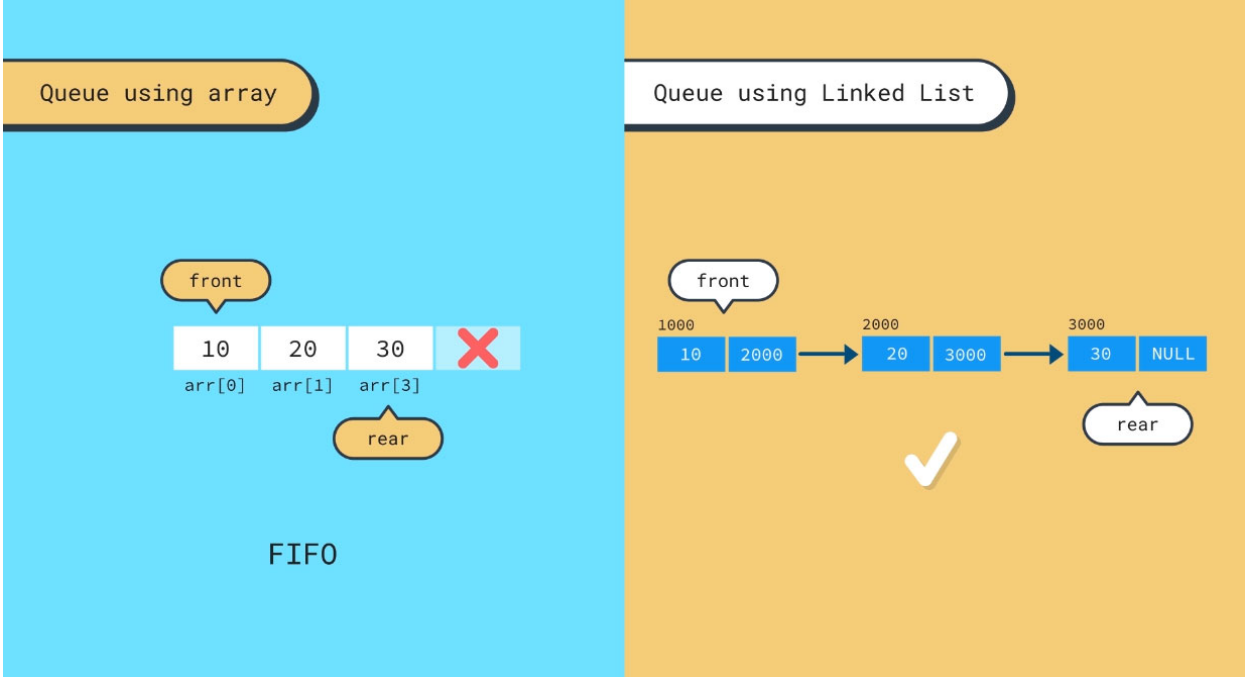
```
{
    if(rear == size)
        return 1;
    return 0;
}

void enqueue(int val)
{
    if(isQueueFull())
        return;
    else
    {
        arr[rear] = val;
        rear++;
    }
}

int isQueueEmpty()
{
    if(rear==front)
        return 1;
    else
        return 0;
}

int dequeue()
{
    if(isQueueEmpty())
        return -1;
    else
        return arr[front++];
}
int main()
{
    enqueue(10);
    enqueue(20);
    enqueue(30);
    enqueue(40);
    printf("%d ",dequeue());
    printf("%d ",dequeue());
    printf("%d ",dequeue());
    printf("%d ",dequeue());
```

```
52
53      return 0;
54  }
```

那么我们也可以使用链表来实现队列。这是因为使用数组的时候，当队列超出数组的容量的时候，我们无法在插入数据。



而链表因为没有容量的限制（当然不能突破内存的大小），可以使用任意的内存位置来实现数据的插入。如下图右侧所示，插入的节点就的队尾rear。



使用链表的时候，我们要声明两个指针变量front和rear，指向链表中的队头和队尾节点。

```
#include<stdio.h>
#include<stdlib.h>

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

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

-

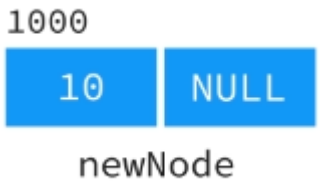
因为一开始链表为空，所以接下来我们来实现一个入队的方法。

```
void enqueue(int val)
{

}
```

所谓的入队，就是要新建一个节点，然后将这个节点插入到链表的队尾，为此将节点的数据设置为val，因为是最后一个节点，为此next为NULL。

```
1 void enqueue(int val)
2 {
3     struct node *newNode = malloc(sizeof(struct node));
4     newNode->data = val;
5     newNode->next = NULL;
6 }
```

当第一次调用enqueue方法并传入10时，其结构如图：

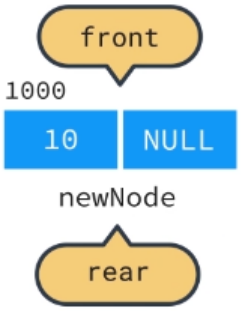


因为这个节点是链表的第一个节点，为此这个节点既是队头又是队尾。所以我们要把front和rear都指向这第一个节点。

为此我们要在第一个节点生成后，将front和rear都指向这个节点。代码为6-7行。

```
1 void enqueue(int val)
2 {
3     struct node *newNode = malloc(sizeof(struct node));
4     newNode->data = val;
5     newNode->next = NULL;
6     if(front == NULL && rear == NULL)
7         front = rear = newNode;
8 }
```

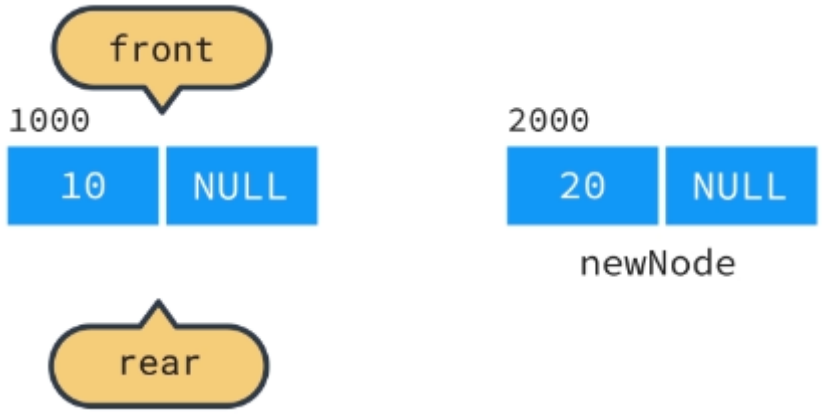
当执行6-7代码后，链表的结构如下图所示：



接下来我们可以插入20节点，当我们使用enqueue方法并传入20时。我们会新建一个新节点20。

```
1  void enqueue(int val)
2  {
3      struct node *newNode = malloc(sizeof(struct node));
4      newNode->data = val;
5      newNode->next = NULL;
6      if(front == NULL && rear == NULL)
7          front = rear = newNode;
8  }
```

结构体如下：

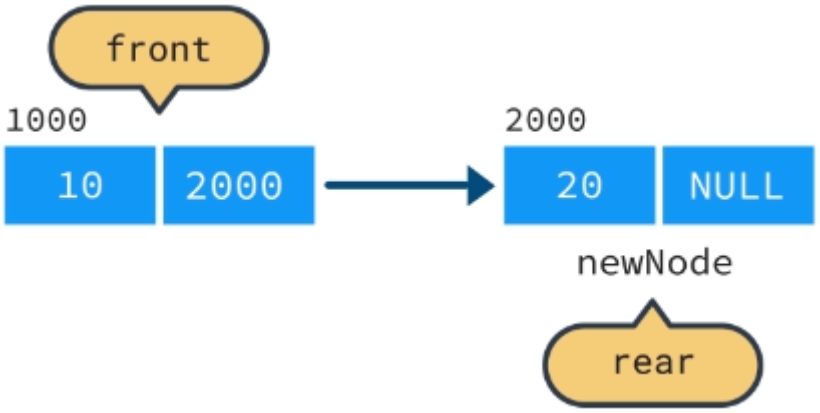


这时候因为front和rear都不为空，所以不会执行第7行的代码，但是我们要做的事情是将newNode接入到队头节点的后面。为此我们添加9-13的五行代码。

```
1  void enqueue(int val)
2  {
3      struct node *newNode = malloc(sizeof(struct node));
4      newNode->data = val;
5      newNode->next = NULL;
6
7      if(front == NULL && rear == NULL)
8          front = rear = newNode;
```

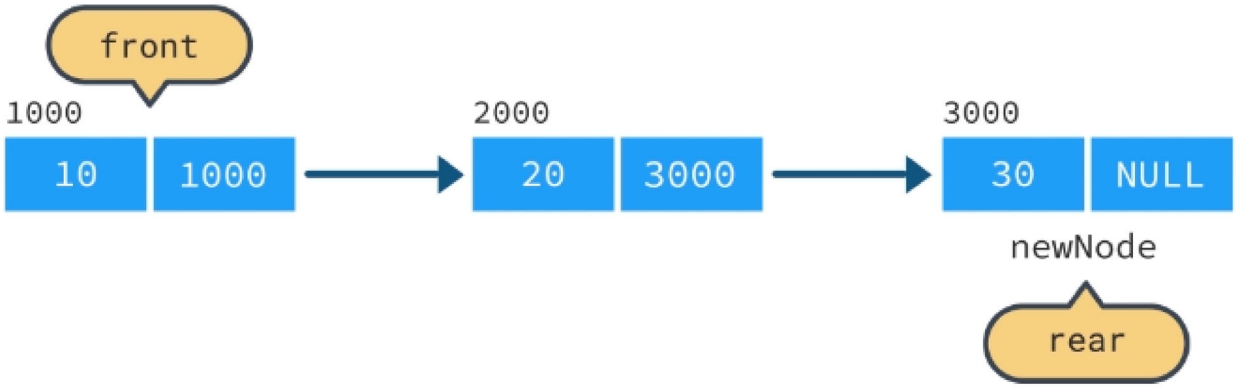
```
9       else
10      {
11          rear->next = newNode;
12          rear= newNode;
13      }
14  }
```

这样就实现了头指向了新节点，并且rear指针指向了新节点。

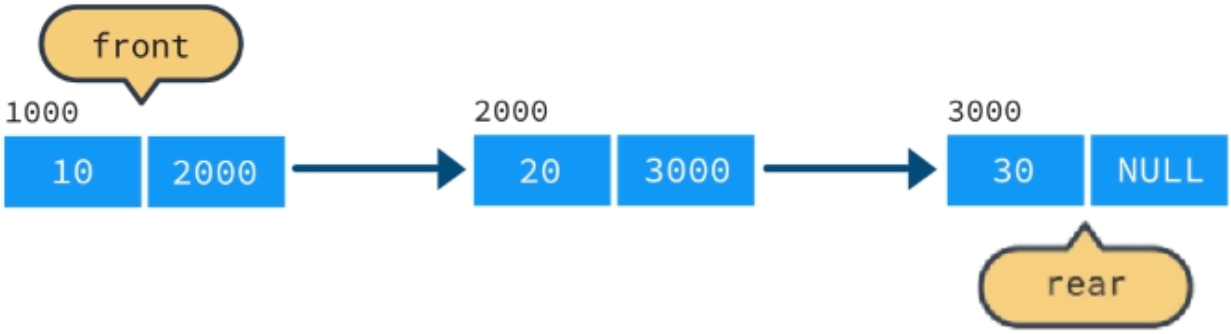


使用enqueue方法并传入30时，新节点建立。

当执行11-12行的代码后，链表结构如下：



接下来我们讲下出队。因为队列先进先出的特性，删除是从链表的头节点开始的。



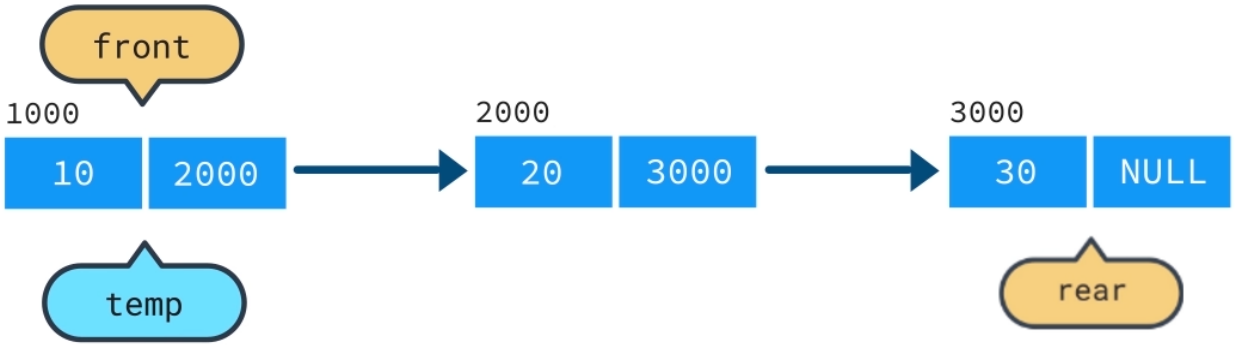
为此我们声明dequeue函数，返回值表示删除元素的值是多少，以便在后期进行使用或者记录。

```
int dequeue()
{
    struct node *temp;
}
```

因为要删除的是front，为此我们使用一个temp指针来指向将要删除的那个节点。当链表为空的时候，是无法让元素出队列的，为此我们要考虑这种特殊的情况。所以front==NULL的时候，表示链表为空，返回-1表示队列空，不能出队，值-1表示错误的意思。如果front不为NULL，表示可以删除front。为此我们使用temp记录要删除的front节点。

```
int dequeue()
{
    struct node *temp;
    if(front==NULL)
    {
        printf("队列为空，无法执行出队的操作\n");
        return -1;
    }
    else
    {
        printf("出队的元素=%d\n",front->data);
        temp=front;
    }
}
```

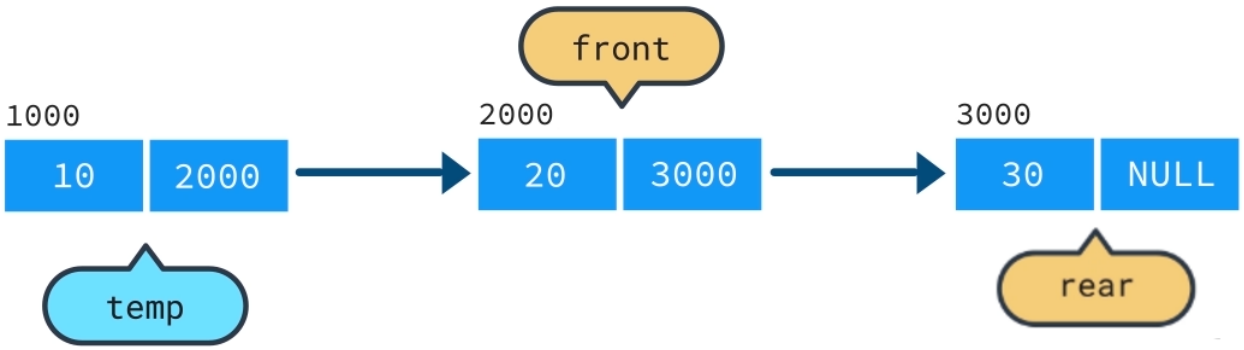
执行temp=front后的链表结构体如下：



当要删除的队首记录后，就需要将front移动到下一个节点的位置。为此我们使用添加第13行的front=front->next的代码，生成新的队首。

```
1  int dequeue()
2  {
3      struct node *temp;
4      if(front==NULL)
5      {
6          printf("队列为空，无法执行出队的操作\n");
7          return -1;
8      }
9      else
10     {
11         printf("出队的元素=%d\n",front->data);
12         temp=front;
13         front=front->next;
14     }
15 }
```

front=front->next执行后的链表结构体如下：

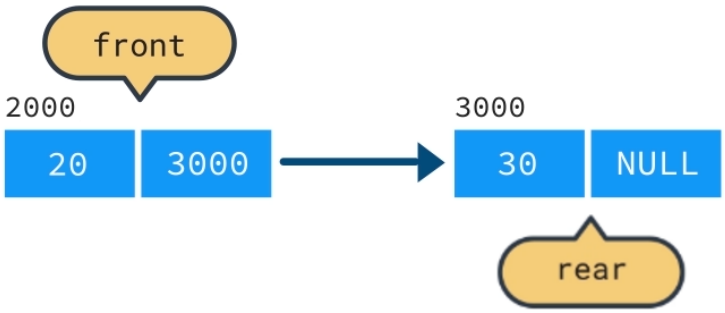


因为新的front也存在两种情况，一种是front指向的节点为NULL，一种是不为NULL。新的front为NULL的话，表示后面没有节点了，为此rear尾巴也是NULL，代码见15-16行。新的front不为NULL，那么就可以上次temp标记的那个要删除的前队首节点，使用free对其占用的节点内存进行清空。并返回这个删除节点的值。代码见18-20。

```
1  int dequeue()
2  {
```

```
    struct node *temp;
    if(front==NULL)
    {
        printf("队列为空，无法执行出队的操作\n");
        return -1;
    }
    else
    {
        printf("出队的元素=%d\n",front->data);
        temp=front;
        front=front->next;

        if(front==NULL)
            rear=NULL;

        int result=temp->data;
        free(temp);
        return  result;
    }
}
```

删除10节点后的链表结构体如下所示：



后面删除的过程就自己去推导了。最终的链表实现的队列完整代码如下：

```
#include<stdio.h>
#include<stdlib.h>

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

```c

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

void enqueue(int val)
{
    //Task 1: Correct the below logic
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;
    printf("%d入队\n",val);
    newNode->next = NULL;

    if(front == NULL && rear == NULL)
        front = rear = newNode;
    else
    {
        rear->next = newNode;
        rear = newNode;
    }
}

int dequeue()
{
    struct node *temp;
    if(front==NULL)
    {
        printf("队列为空，无法执行出队的操作\n");
        return -1;
    }
    else
    {
        printf("出队的元素=%d\n",front->data);
        temp=front;
        front=front->next;

        if(front==NULL)
            rear=NULL;

        int result=temp->data;
        free(temp);
        return  result;
    }
}
```

```
int main()
{
    dequeue();
    enqueue(10);
    enqueue(20);
    enqueue(30);
    dequeue();
    dequeue();
    dequeue();
    dequeue();

    return 0;
}
```