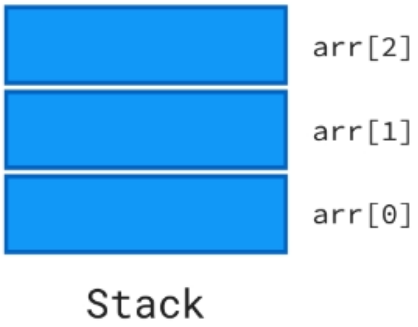
栈的实现

栈是一种先进后出的数据结构。当我们在浏览器的一个标签页访问不同的网页的时候，你如果使用回退按钮，可以回退以前访问的页面，这个实现的技术就是栈。

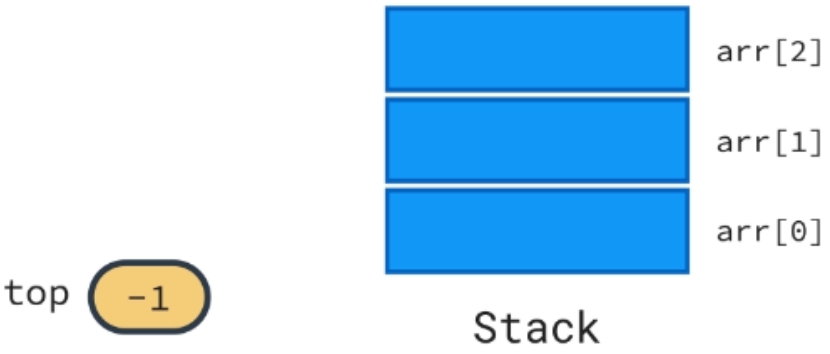
关于栈的实现我们可以使用数组和链表两种方式。我们首先使用数组的方式，我们假定实现一个能容纳三个元素的栈。为此我们定义size等于3，在定义一个大小为3的int类型的数组。对于每个栈来说，有一个名为栈顶top的特性，来指定放置在最上面元素的位置。

```
#include<stdio.h>
#define size 3
int stack[size];
int top = -1;
```

注意，栈最下面是栈的底步。为此栈底在arr[0]的位置。



因为一个元素都没有，为此这个栈的栈顶top目前为-1。



因为栈是空的，为此我们需要推送一些数据到栈里面要，这个过程我们简称为压栈。但是在压栈之前，我们可以先写两个方法，第一个方法isStackEmpty为判断栈是不是为空。当top等于-1的时候，为空，返回1，否则返回0。

```
int isStackEmpty()
{
    if(top == - 1)
        return 1;
    return 0;
}
```

一个判断栈是不是满了的方法isStackFull，当top为容量-1的时候，就表示满了。满了返回1，否则返回0。

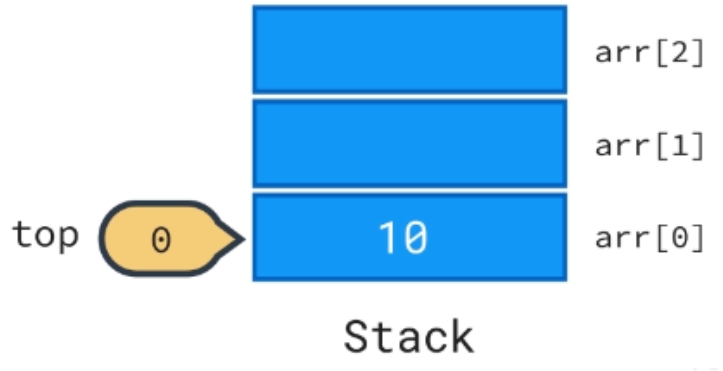
```
int isStackFull()
{
   if(top == size - 1)
            return 1;
   return 0;
}
```

接下来我们压入一个元素10到栈里面，其具体代码如下。实现我们要判断栈是不是满了，满了就无法在压入任何的元素。没有满，我们就需要将栈顶top从原本的-1向上走一格变为0，然后将传入的10放到数组的0位置。

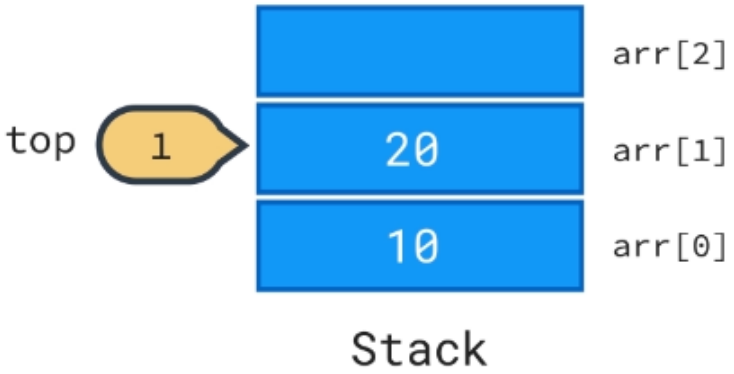
```
void push(int val)
{
    if(isStackFull())
        return;
    else
    {
        ++top;
        stack[top]=val;
        printf("压入元素%d到索引为%d的位置\n",val,top);
```

```
    }
}
```

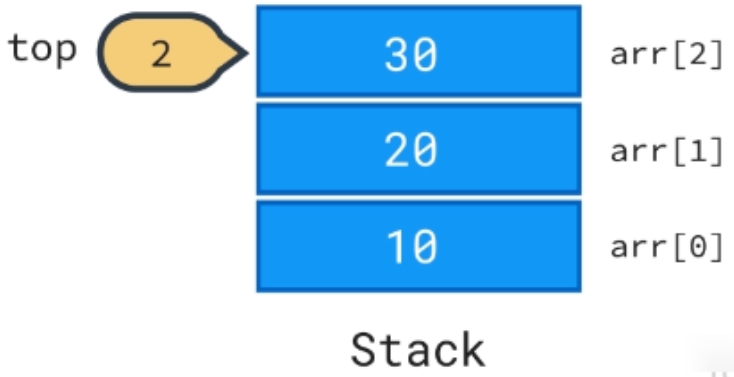
10插入栈的结构如下：



压入20的栈的结构如下：



压入30的栈的结构如下，这时候栈满了无法再压入其他的元素。

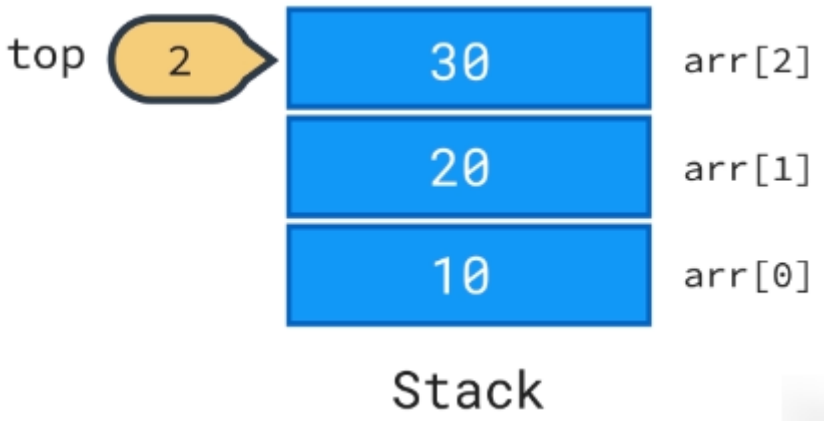


当要删除元素的时候，我们是从栈顶开始删除，这就是所谓的先进后出，后进先出的特性。要删除元素，必须确保栈里面得有，为此我们判断栈是不是空，是的话就直接返回-1。如果不为空，我们就可以删除从栈顶开始删除元素了。

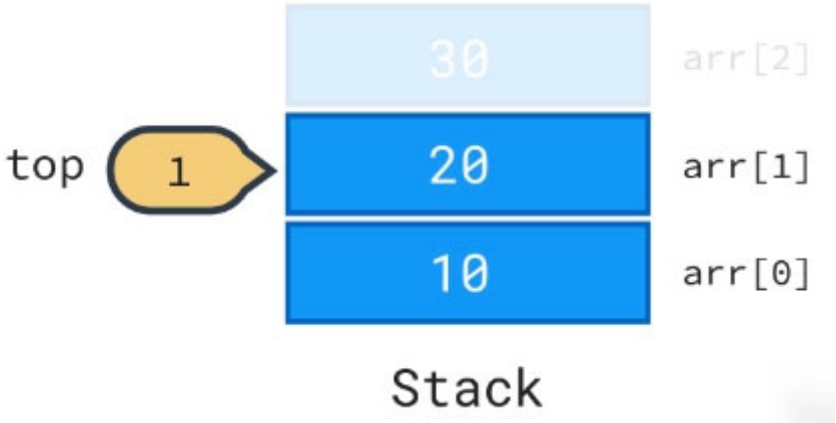
```
int pop()
{
    if(isStackEmpty())
        return -1;
    else
    {
```

```
        printf("从索引为%d的位置弹出元素%d\n",top,stack[top]);
        return stack[top--];
    }
}
```

当我们调用pop方法的时候，先读取到栈顶的元素，也就是30这个元素。打印要删除的是2这个位置以及30这个元素值。



返回要删除的值的目的是为了让别人知道删除了什么。pop有弹出元素的意思，但是这里实际上并没有把30这个值真正删除，而只是top减一，栈顶元素变成了20。30这个元素变成了栈的无效数据而已。

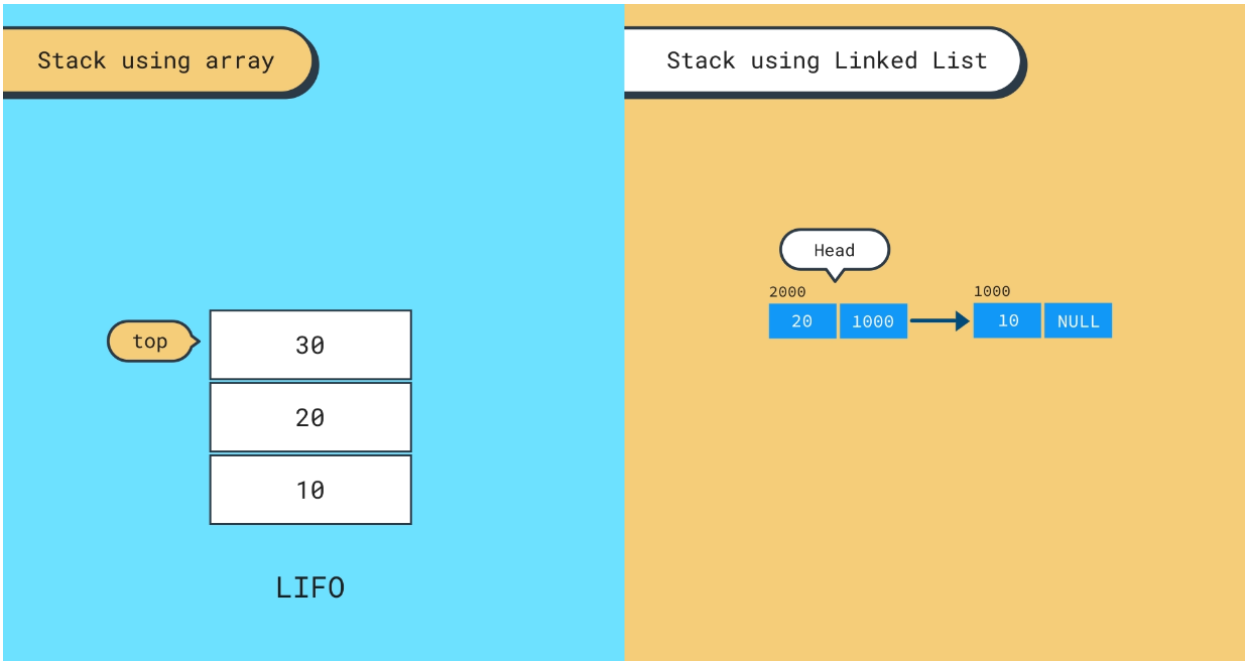


下面是使用数组实现栈的具体代码：

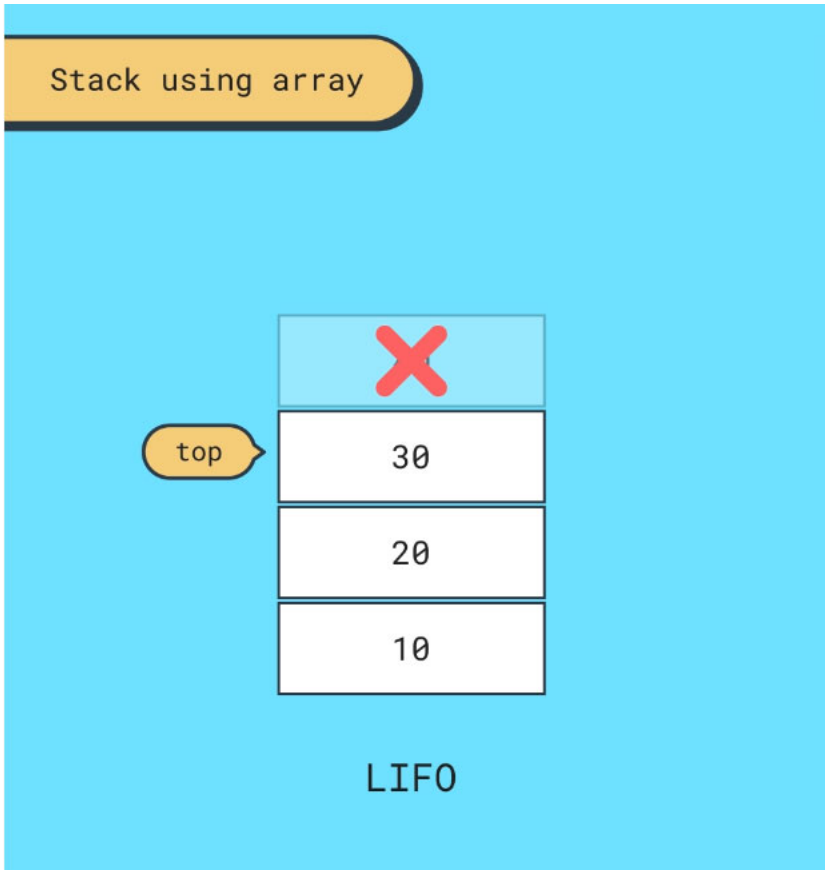
```
#include<stdio.h>

#define size 3
int stack[size];
int top = -1;

int isStackFull()
{
    if(top == size - 1)
        return 1;
    return 0;
}
```

```
13
14 void push(int val)
15 {
16     if(isStackFull())
17         return;
18     else
19     {
20         ++top;
21         stack[top]=val;
22         printf("压入元素%d到索引为%d的位置\n",val,top);
23     }
24 }
25
26 int isStackEmpty()
27 {
28     if(top == - 1)
29         return 1;
30     return 0;
31 }
32
33 int pop()
34 {
35     if(isStackEmpty())
36         return -1;
37     else
38     {
39         printf("从索引为%d的位置弹出元素%d\n",top,stack[top]);
40         return stack[top--];
41     }
42 }
43
44 int main()
45 {
46     push(10);
47     push(20);
48     push(30);
49     push(40);//压不进去了
50     push(40);//压不进去了
51     pop();
52     pop();
53     return 0;
54 }
```

前面讲过使用数组实现的栈，今天我们换链表来实现栈。链表实现栈的好处就是不需要连续的空间，可以任意添加元素而不用担心扩容的问题，不像数组一旦确定了容量就不能改变。

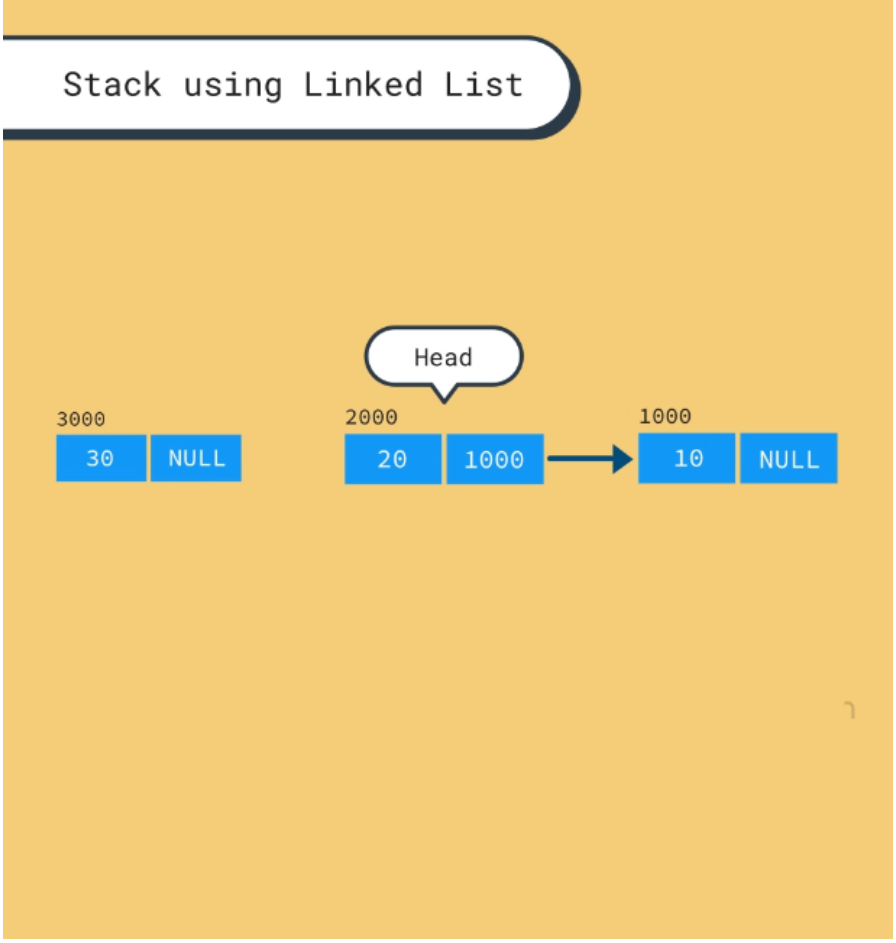


像前面的数组大小为3的时候，当你再想加入元素的时候不可以了，因为数组一定确定了大小就不能改变。

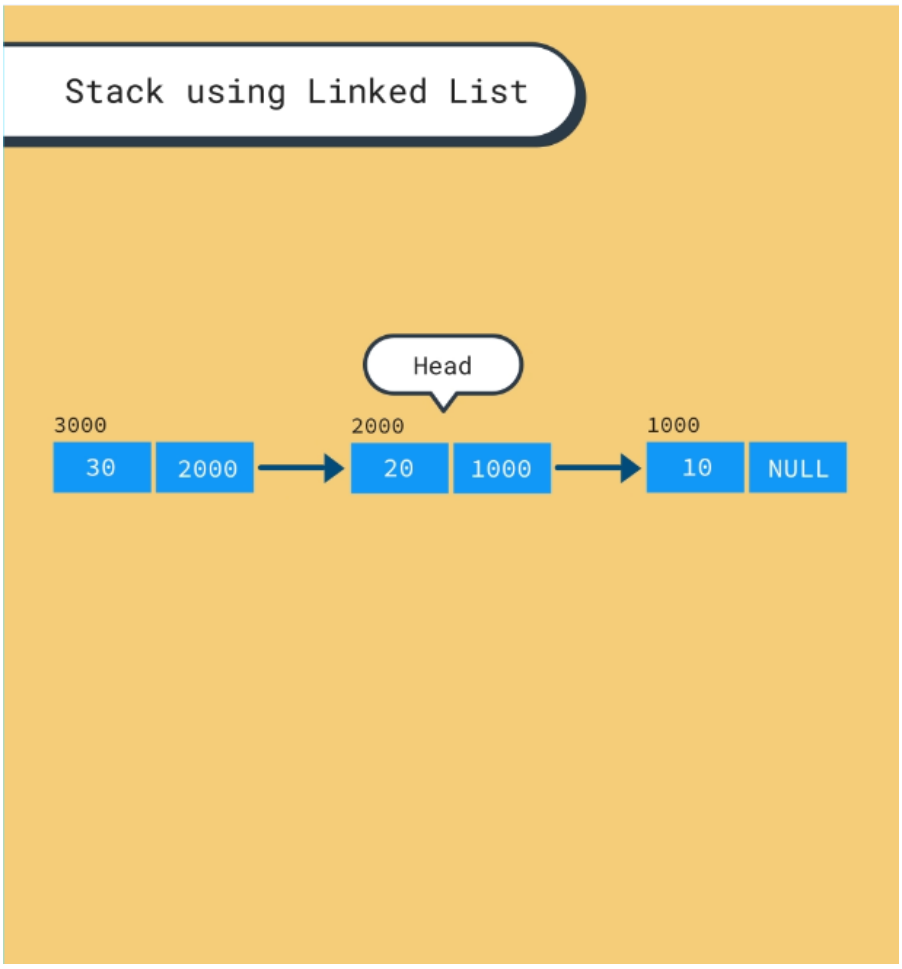


假定我们要这种情况下插入一个元素30。那么30将成为栈顶。链表因为是内存不连续的，为此不用考虑这个新增的元素放在哪里。

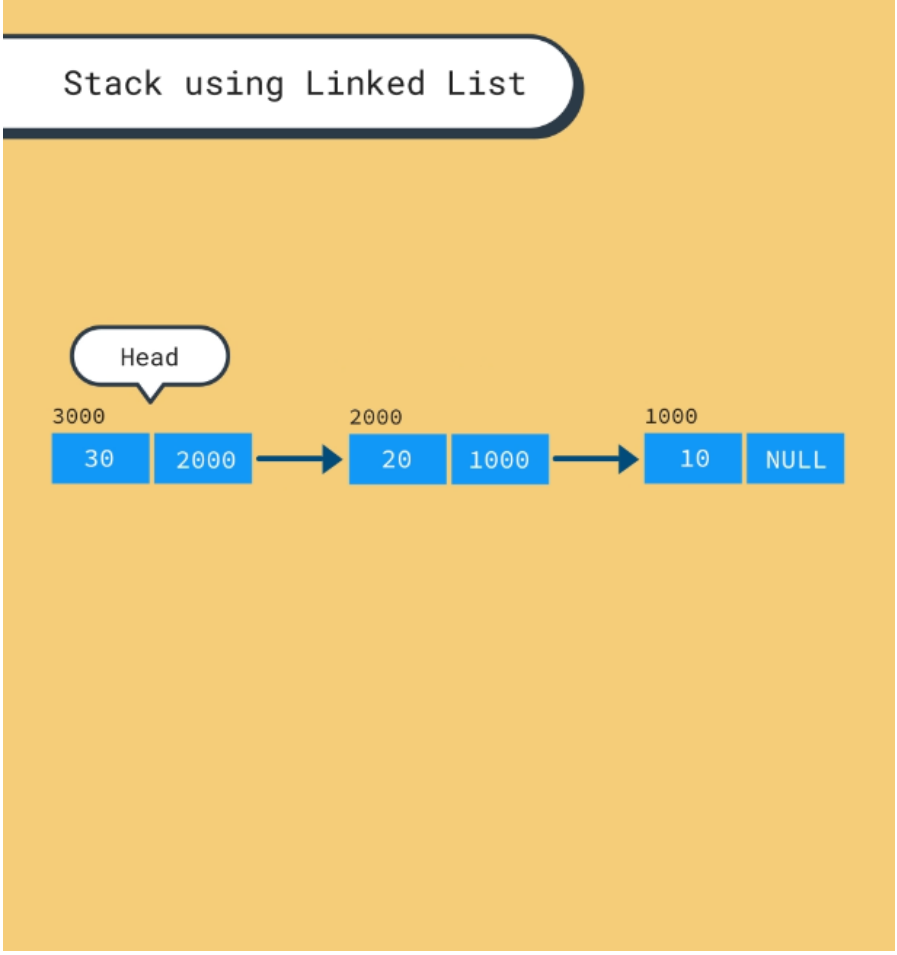
你只要在任意地址建立这个节点。



将新节点的next指向链表的Head。



再将新建的节点变成新的Head就进行了扩容和新增元素。



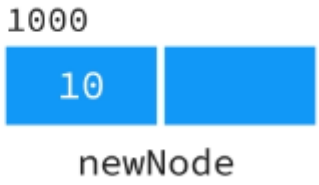
我们可以设定链表的头一开始为空。

```
struct node *head = NULL;
```

当我们调用push方法压入第一个元素的时候，我们会新建一个节点，将这个节点的值为10。

```
void push(int val)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;

    newNode->next = head;
    head = newNode;
}
```

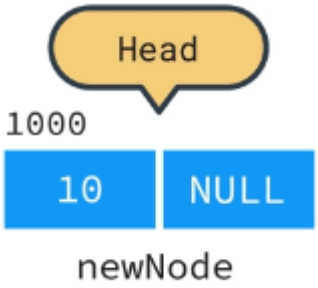
当上述代码执行到第4行时的结构如下：



我们将新节点的next设置为head，newNode->data也就null。

```
void push(int val)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;

    newNode->next = head;

    head = newNode;
}
```

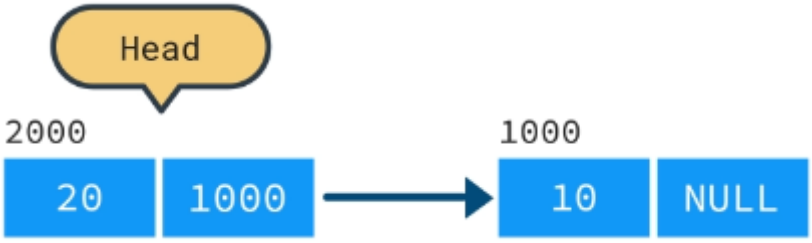
这时候让这个节点变成头，head=newNode。



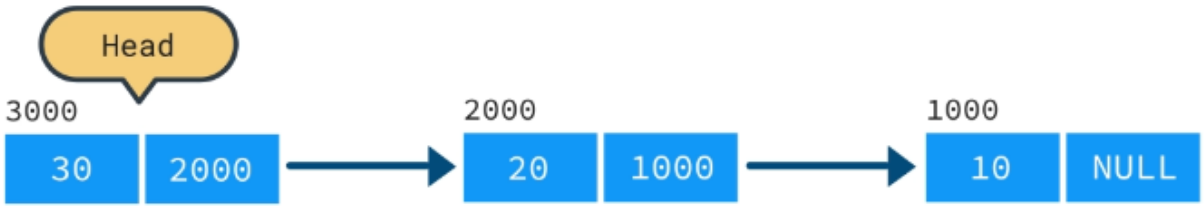
接下来继续调用push方法压入第二个元素20。

```
1 void push(int val)
2 {
3     struct node *newNode = malloc(sizeof(struct node));
4     newNode->data = val;
5     newNode->next = head;
6     head = newNode;
7 }
```

得到的结构如下，可以看到所谓的压栈就是每次在链表的头插入节点。



在压入30元素后的链表结构如下：



接下来我们来做出栈的操作。所谓的的出栈操作，确实就是删除链表的头节点。那么删除元素有两种可能，一种是栈为空，一种是不为空。

为此我们可以写一个pop弹出元素的方法，返回值为-1表示栈空没有删除任何元素，否则就要删除头节点。把这个代码框架写出来先。

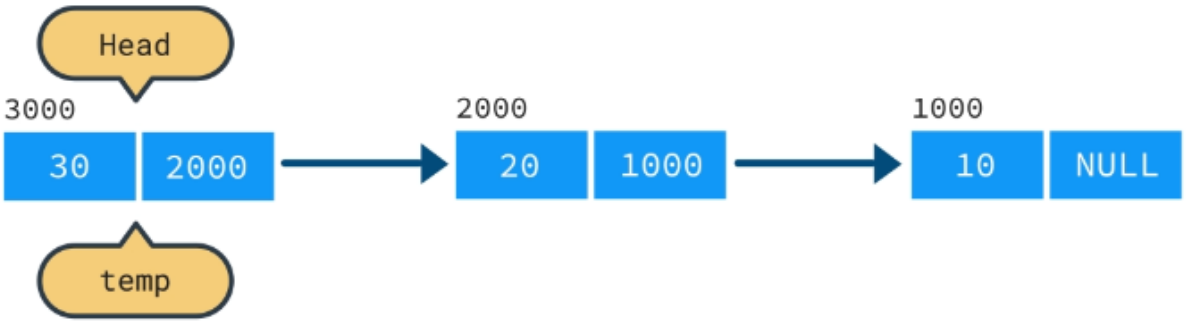
```
int pop()
{
    if(head==NULL)
    {
        printf("栈为空");
        return -1;
    }
    else
    {
       //栈不为空删除头节点的代码
    }
}
```

如果是链表不为NULL，那么就要删除头节点。

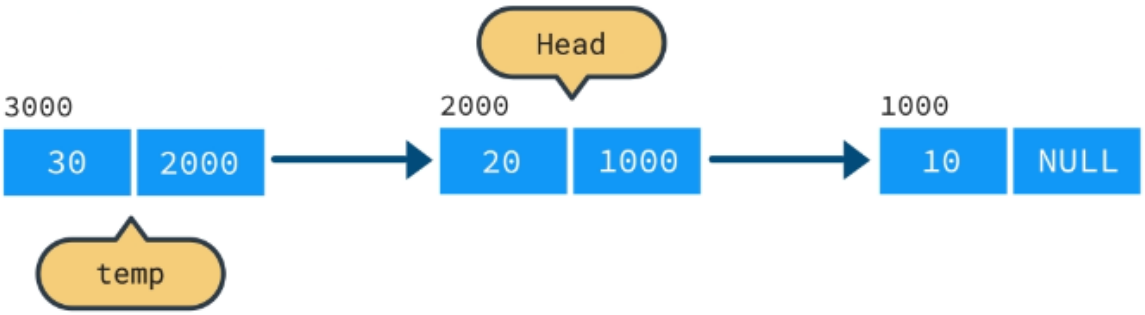
```
int pop()
{
    struct node *tmp;
    if(head==NULL)
    {
        printf("栈为空");
```

```
7           return -1;
8       }
9       else
10      {
11          printf("删除的元素为%d\n",head->data);
12          tmp=head;
13          head=head->next;
14          int result=tmp->data;
15          free(tmp);
16          return result;
17      }
18  }
```

为此我们说明一个指针tmp，来记住要删除的节点。



接下来第11行代码的地方现实要删除的元素是什么， 随后用12行代码来实现tmp记住当前的头，因为删除后第二个节点就要变成新的头了（13行代码处）。



接下来我们用变量result记住删除的节点值，能后释放tmp指针指向的节点占用的内存空间，最后将result返回以便告诉要告诉调用方删除的是那个元素。

链表实现的栈的完整代码如下：

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

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

struct node *head = NULL;

void push(int val)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;

    newNode->next = head;
    head = newNode;
}

int pop()
{
    struct node *tmp;
    if(head==NULL)
    {
        printf("栈为空");
        return -1;
    }
    else
    {
        printf("删除的元素为%d\n",head->data);
        tmp=head;
        head=head->next;
        int result=tmp->data;
        free(tmp);
        return result;
    }

}
```

```
40
41 int main()
42 {
43     push(10);
44     push(20);
45     push(30);
46     pop();
47     pop();
48     push(40);
49     pop();
50     pop();
51     pop();
52     return 0;
53 }
```