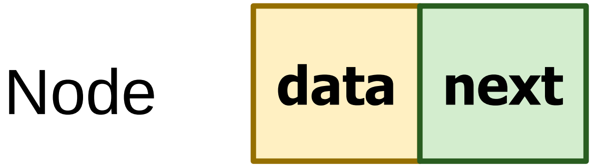
高级语言C++程序设计
Lecture 9 类和对象

简单数据结构

链表 (Linked List)

结点（Node）

```
class Node{
  public:
      int data;
      Node* next;
      Node(int i) {
          data = i;
          next = NULL;
      }
};
```



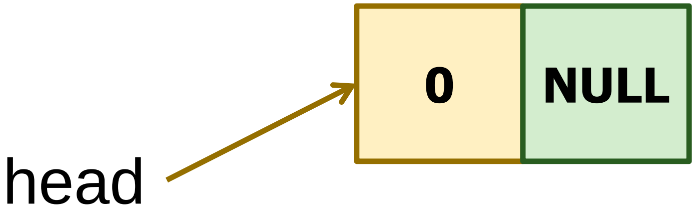
Node

data

next

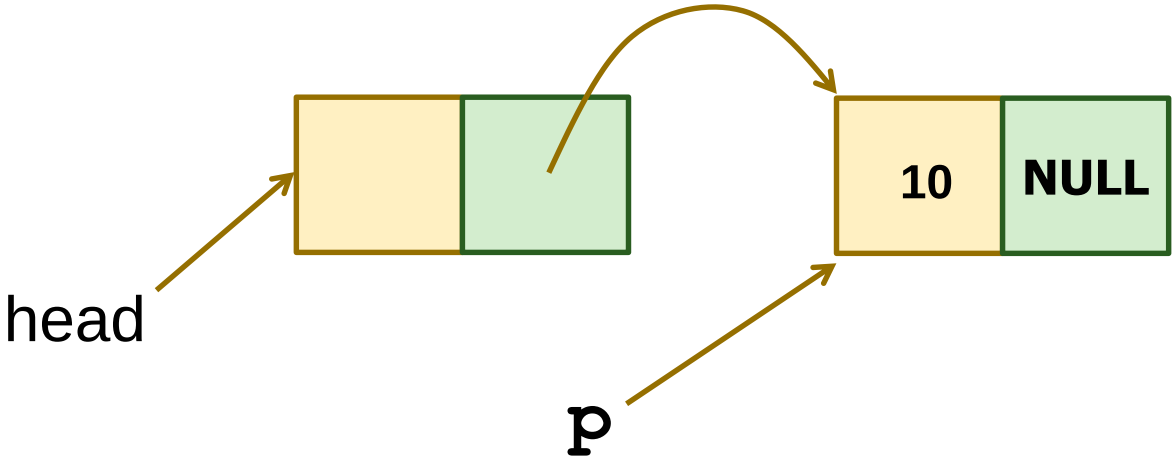
创建头结点

```
Node *head = new Node(0);
```



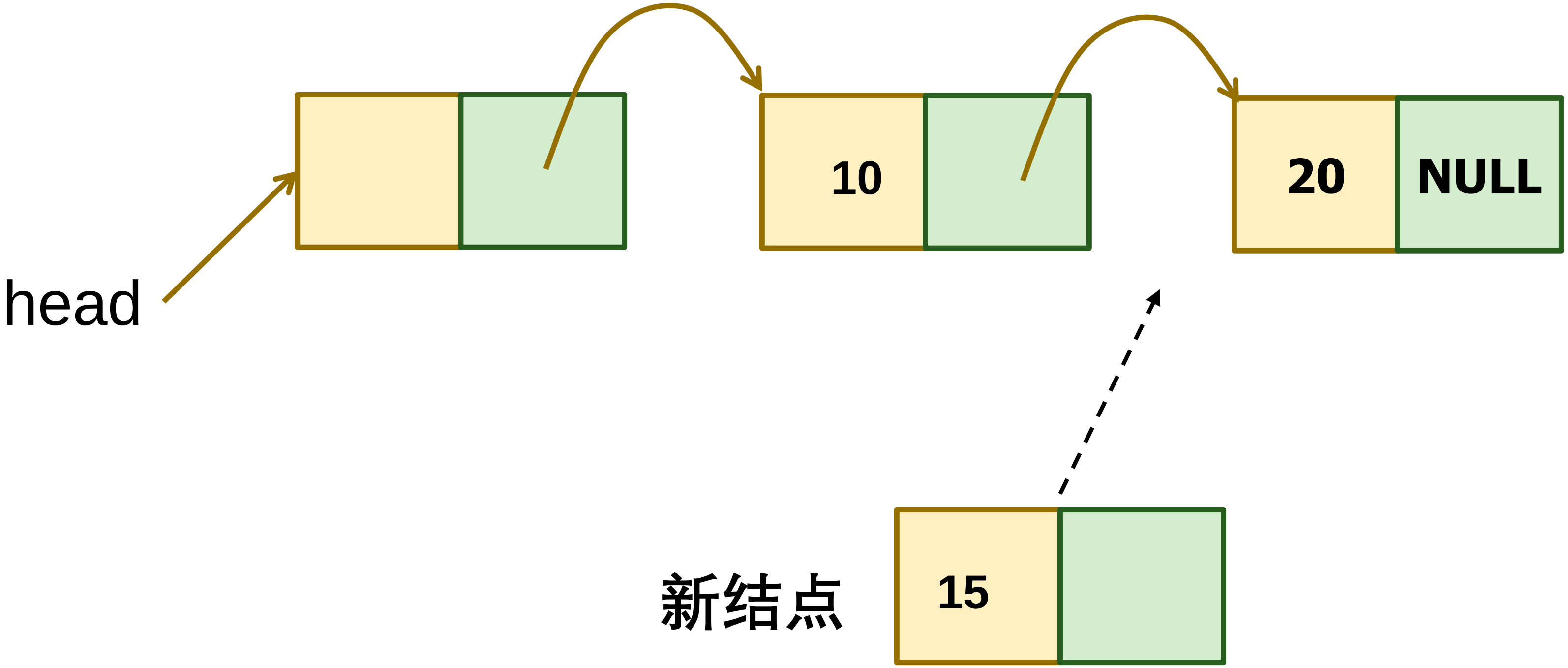
插入第一个结点

```
Node *p = new Node(10);
p->next = head->next;
head->next = p;
```



在第k位置插入结点

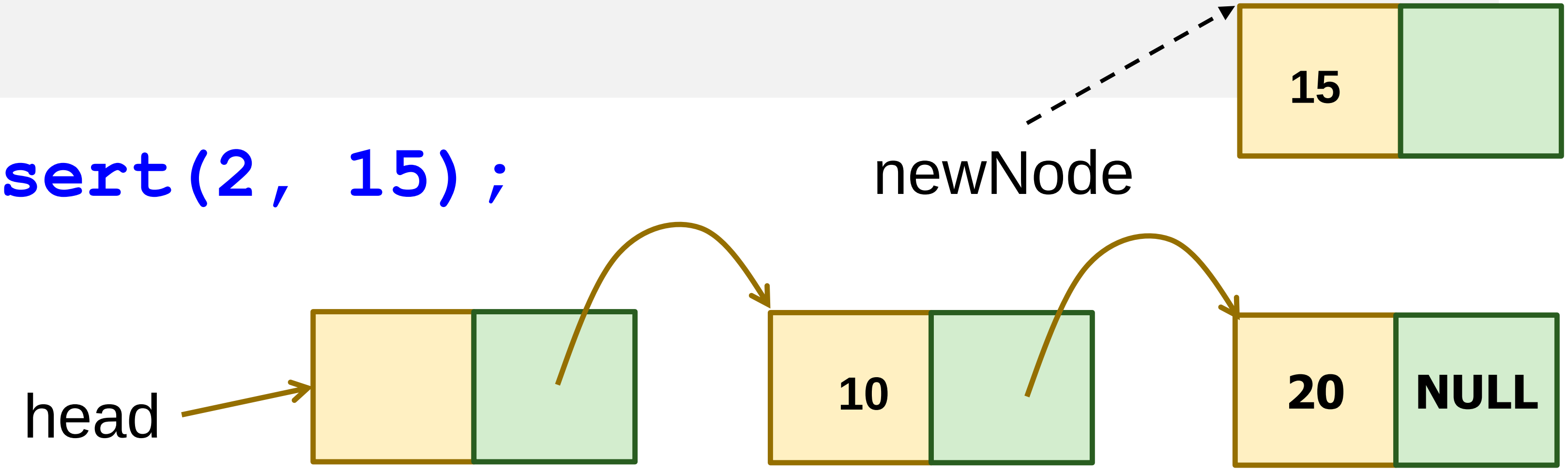
假设链表有2个node，在第2个node位置插入一个新的结点，新结点的元素为15



在第k位置插入结点

```
void Insert(int k, int data) {
    Node *newNode = new Node(data);
    Node *p = head;
    for(int i=1; i<k; i++)
        p = p->next;
    newNode->next = p->next;
    p->next = newNode;
}
```

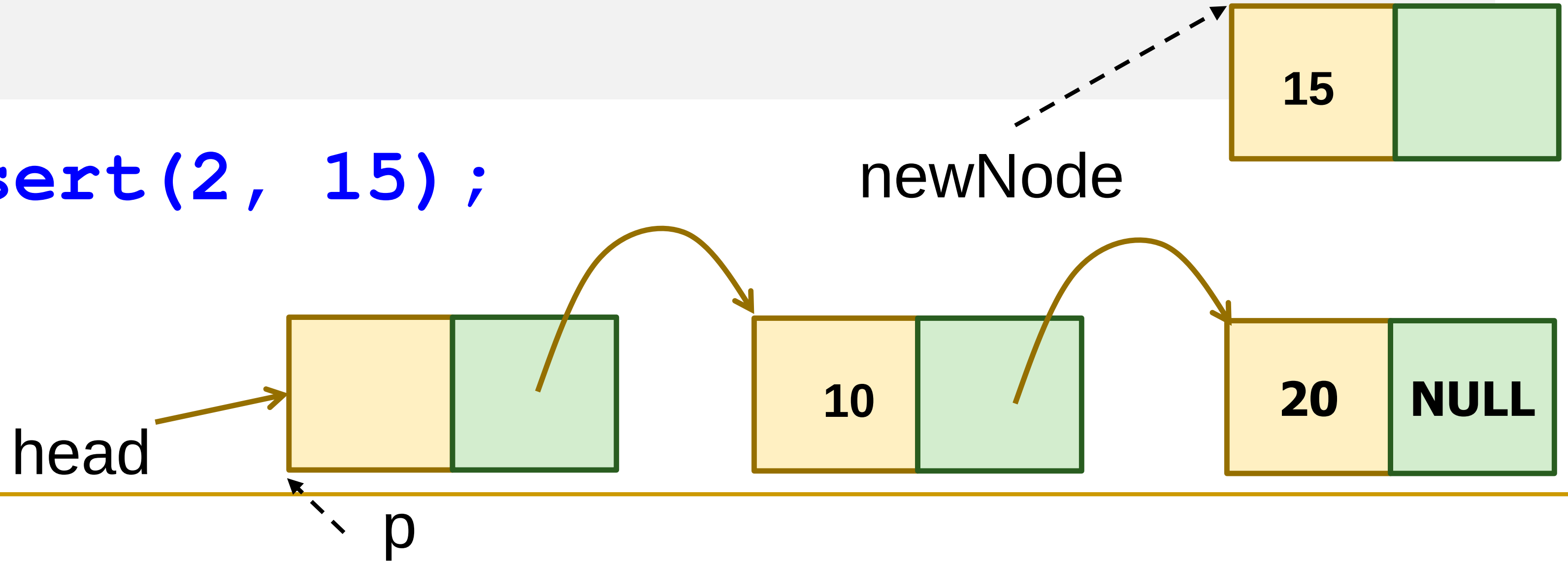
```
Insert(2, 15);
```



在第k位置插入结点

```
void Insert(int k, int data) {
    Node *newNode = new Node(data);
    Node *p = head;
    for(int i=1; i<k; i++)
        p = p->next;
    newNode->next = p->next;
    p->next = newNode;
}
```

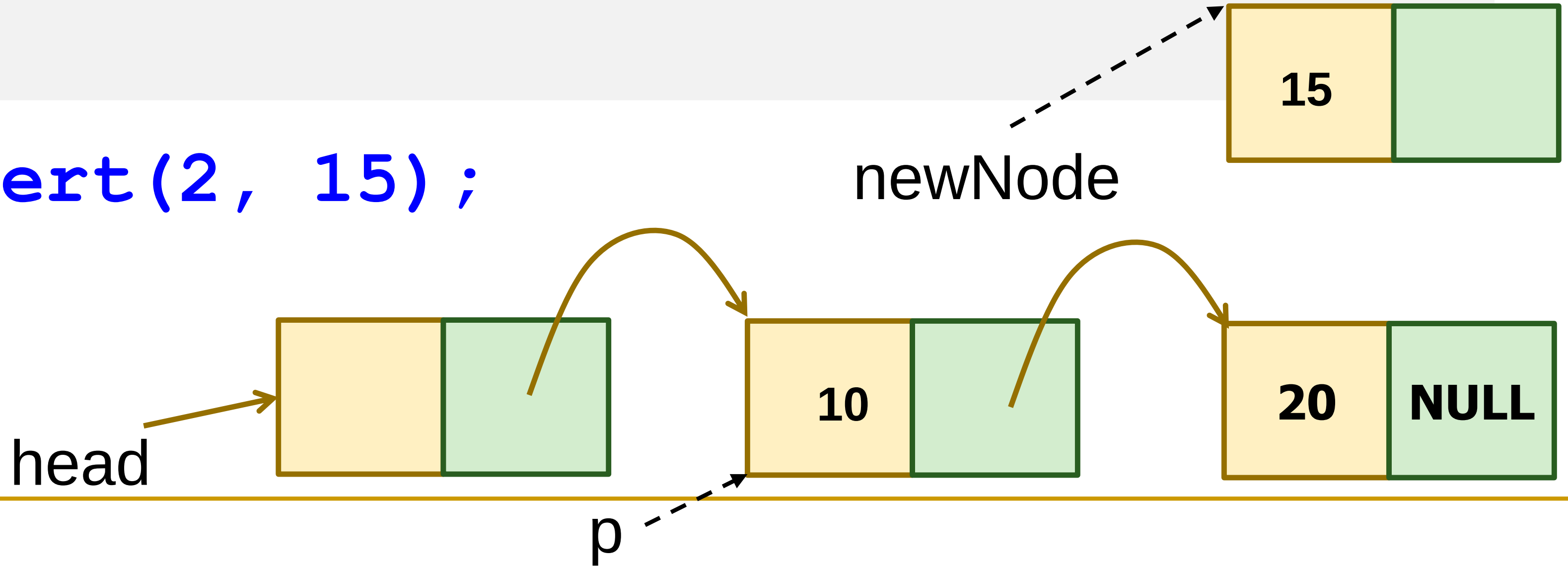
```
Insert(2, 15);
```



在第k位置插入结点

```
void Insert(int k, int data) {
    Node *newNode = new Node(data);
    Node *p = head;
    for(int i=1; i<k; i++)
        p = p->next;
    newNode->next = p->next;
    p->next = newNode;
}
```

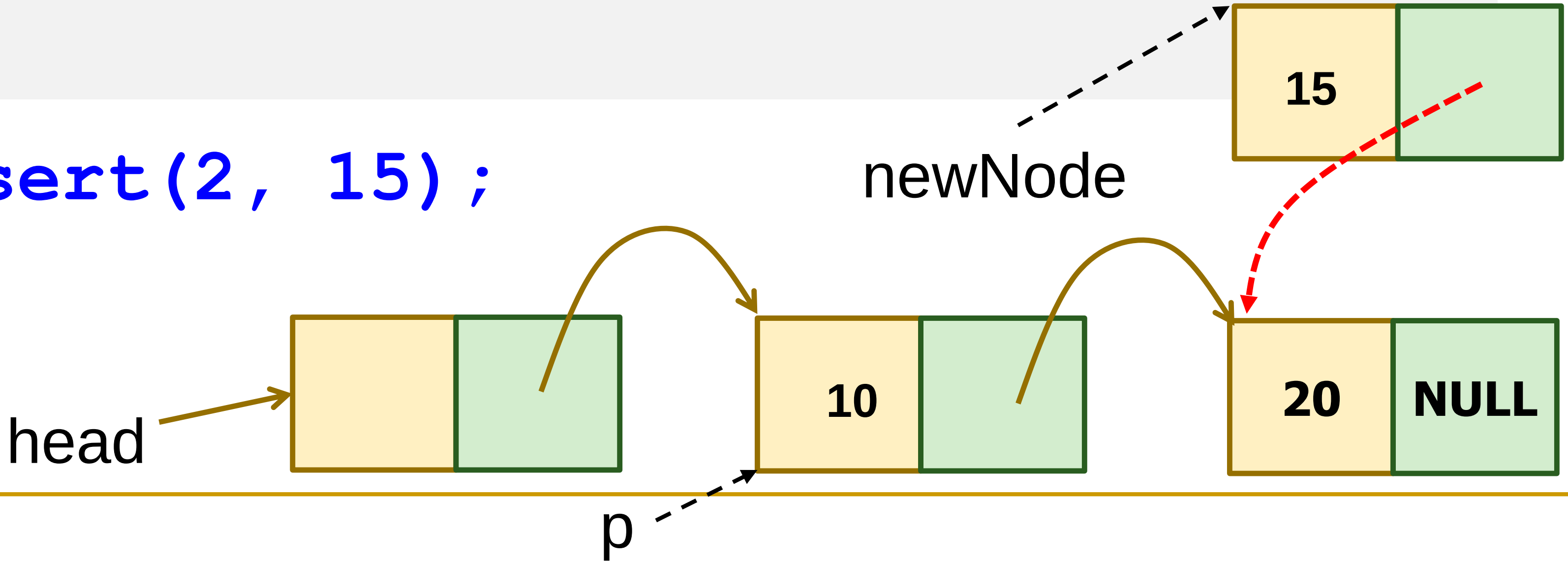
Insert(2, 15);



在第k位置插入结点

```
void Insert(int k, int data) {
    Node *newNode = new Node(data);
    Node *p = head;
    for(int i=1; i<k; i++)
        p = p->next;
    newNode->next = p->next;
    p->next = newNode;
}
```

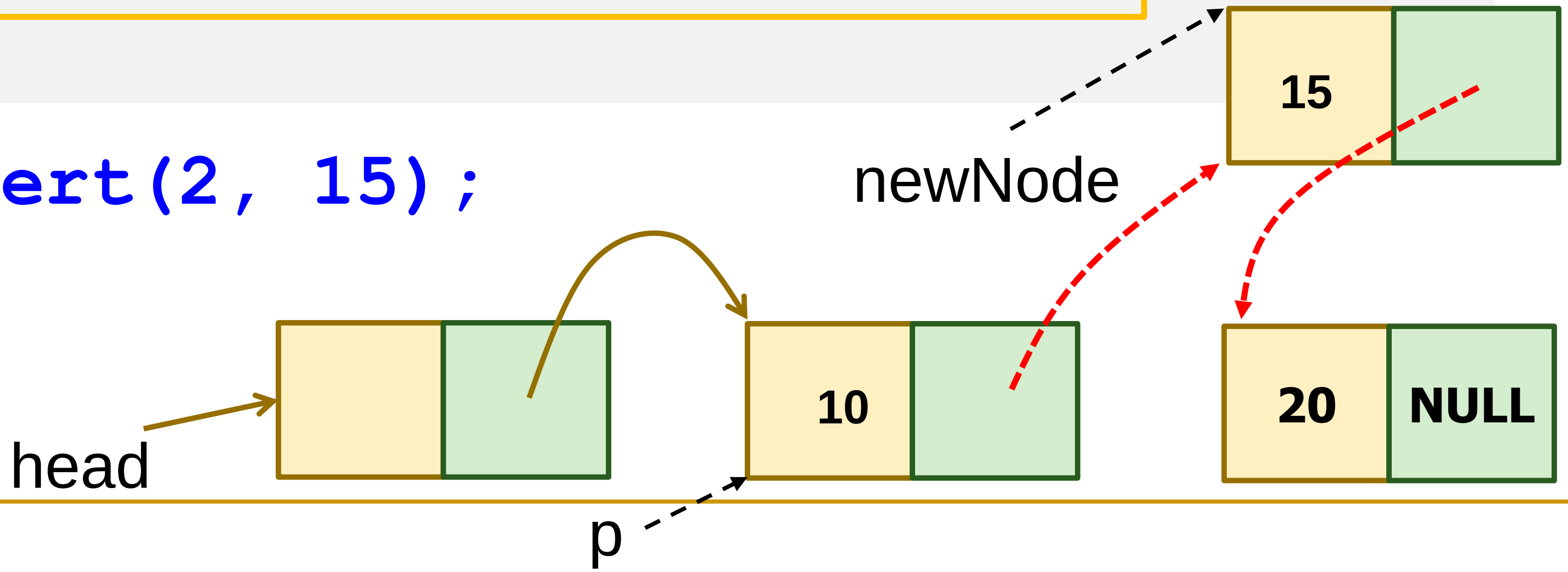
```
Insert(2, 15);
```



在第k位置插入结点

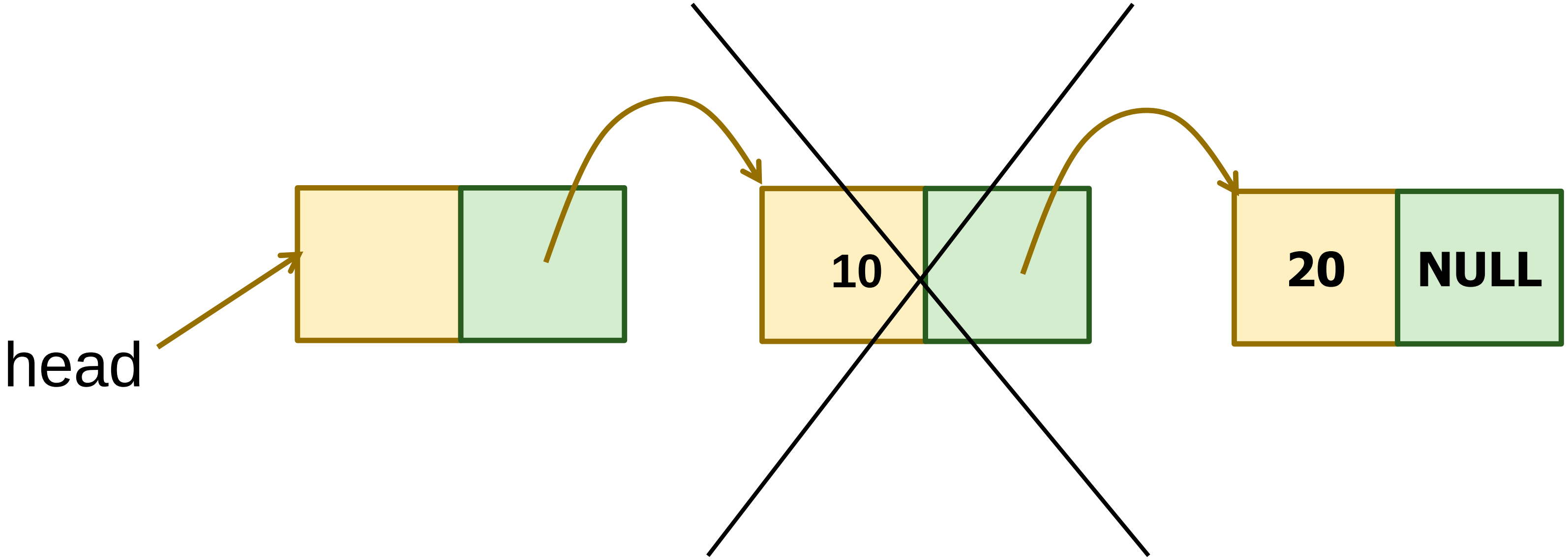
```
void Insert(int k, int data) {
    Node *newNode = new Node(data);
    Node *p = head;
    for(int i=1; i<k; i++)
        p = p->next;
    newNode->next = p->next;
    p->next = newNode;
}
```

```
Insert(2, 15);
```



删除第k个结点

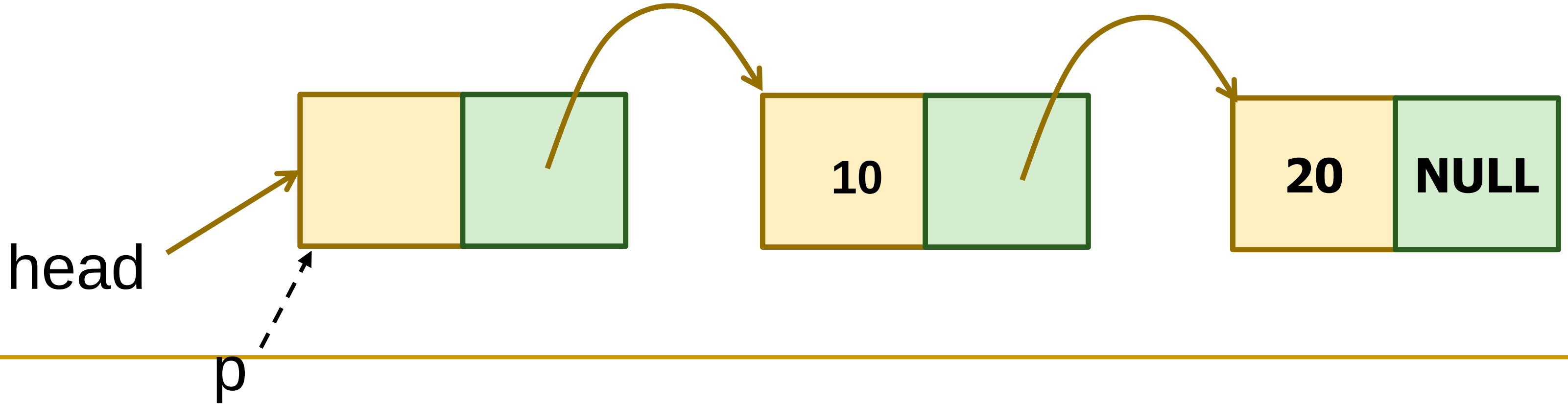
假设链表有2个node，删除第1个node



删除第k个结点

```
void Delete(int k) {
    Node *p = head;
    for (int i=1;i<k;i++)
        p = p->next;
    Node *q = p->next;
    p->next  = q->next;
    delete q;
}
```

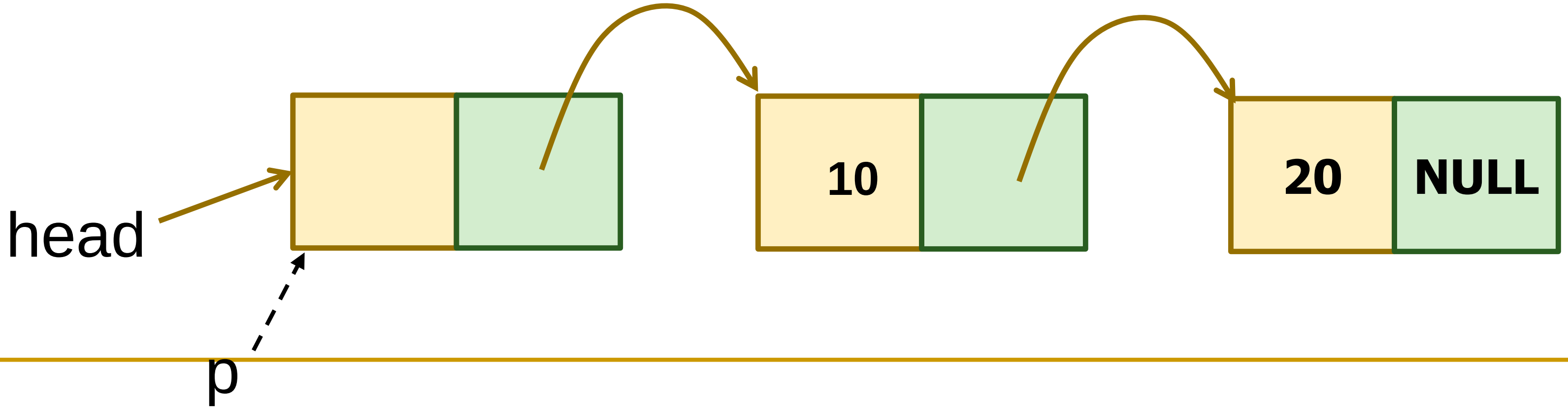
Delete(1);



删除第k个结点

```
void Delete(int k) {
    Node *p = head;
    for (int i=1;i<k;i++)
        p = p->next;
    Node *q = p->next;
    p->next  = q->next;
    delete q;
}
```

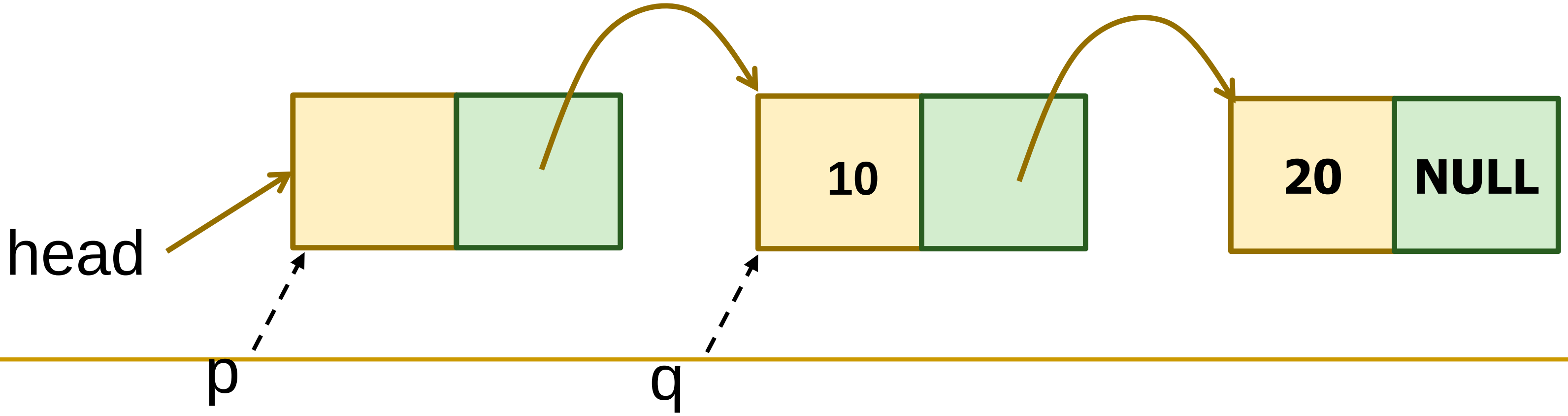
Delete(1);



删除第k个结点

```
void Delete(int k) {
    Node *p = head;
    for (int i=1;i<k;i++)
        p = p->next;
    Node *q = p->next;
    p->next  = q->next;
    delete q;
}
```

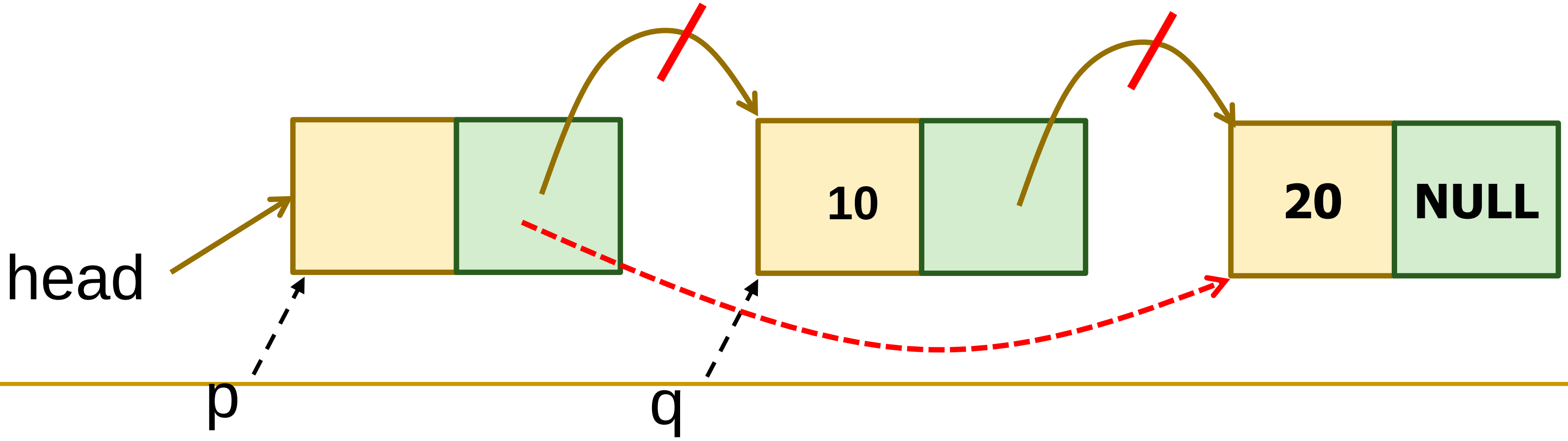
Delete(1);



删除第k个结点

```
void Delete(int k) {
    Node *p = head;
    for (int i=1;i<k;i++)
        p = p->next;
    Node *q = p->next;
    p->next  = q->next;
    delete q;
}
```

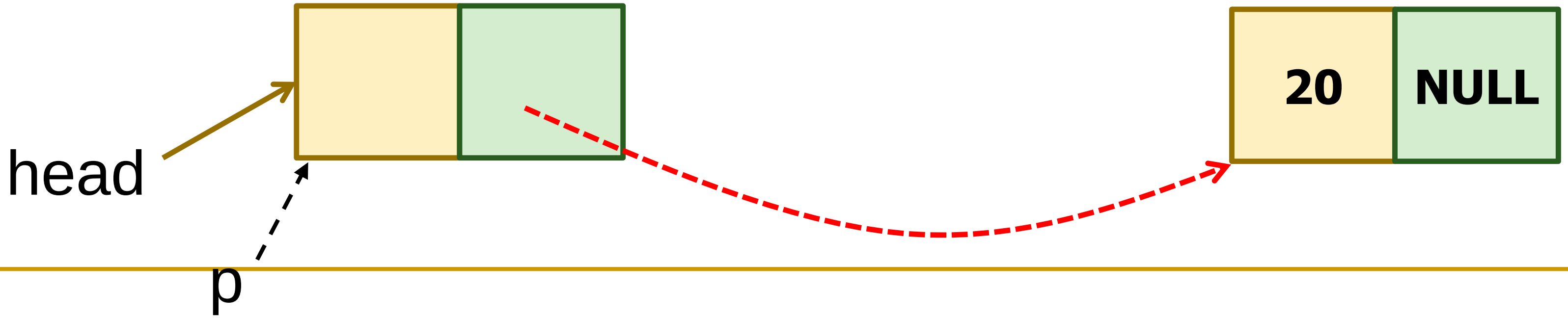
Delete(1);



删除第k个结点

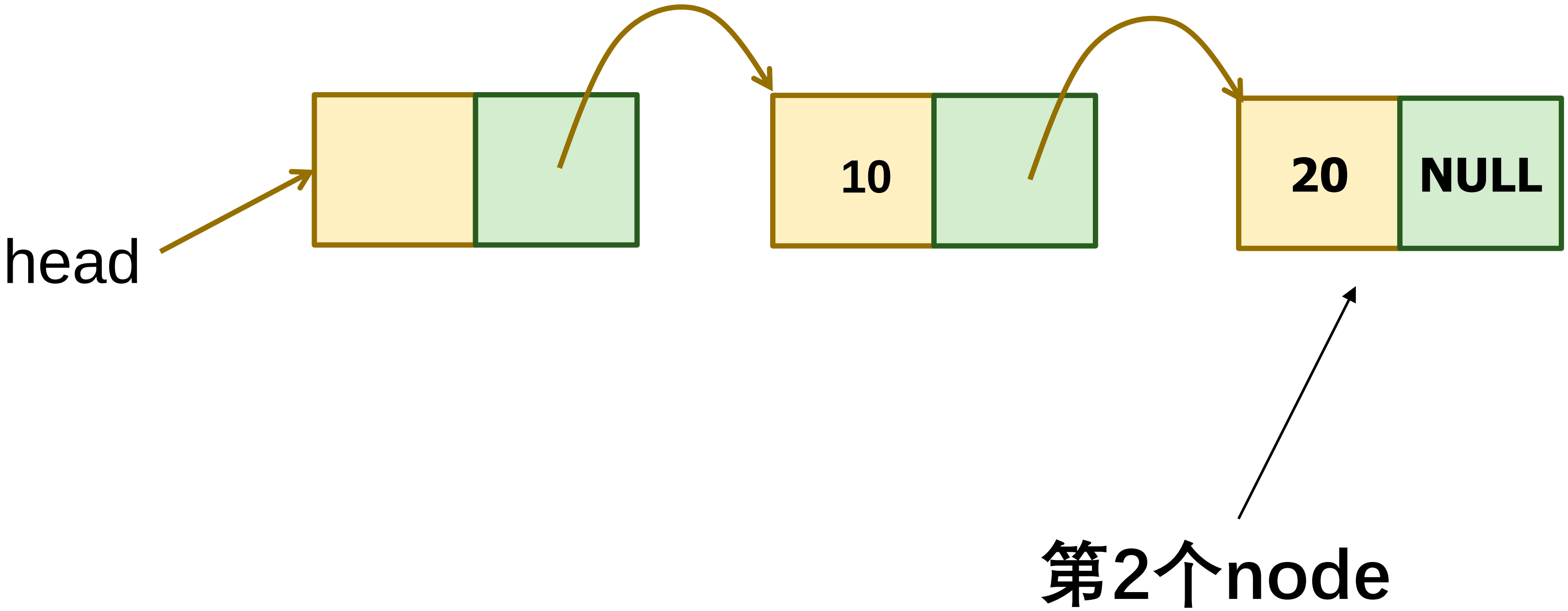
```
void Delete(int k) {
    Node *p = head;
    for (int i=1;i<k;i++)
        p = p->next;
    Node *q = p->next;
    p->next  = q->next;
    delete q;
}
```

Delete(1);



查找第k个结点

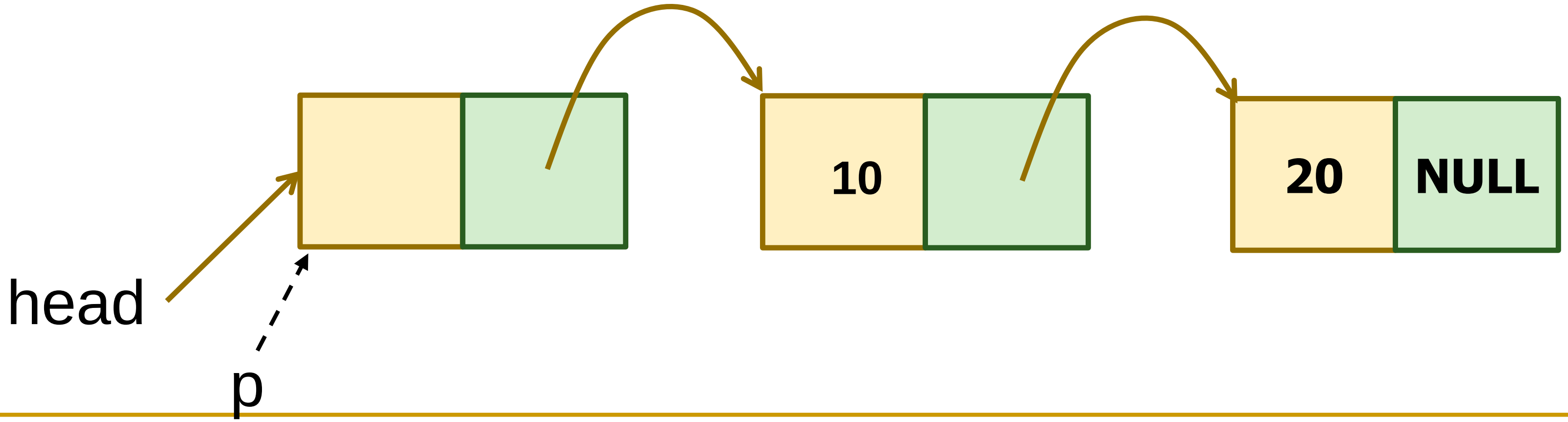
假设链表有2个node，查找第2个node，返回值



查找第k个结点

```
int Find(int k) {
    Node *p = head;
    for (int i=1;i<k;i++)
        p = p->next;
    Node *q = p->next;
    return q->data;
}
```

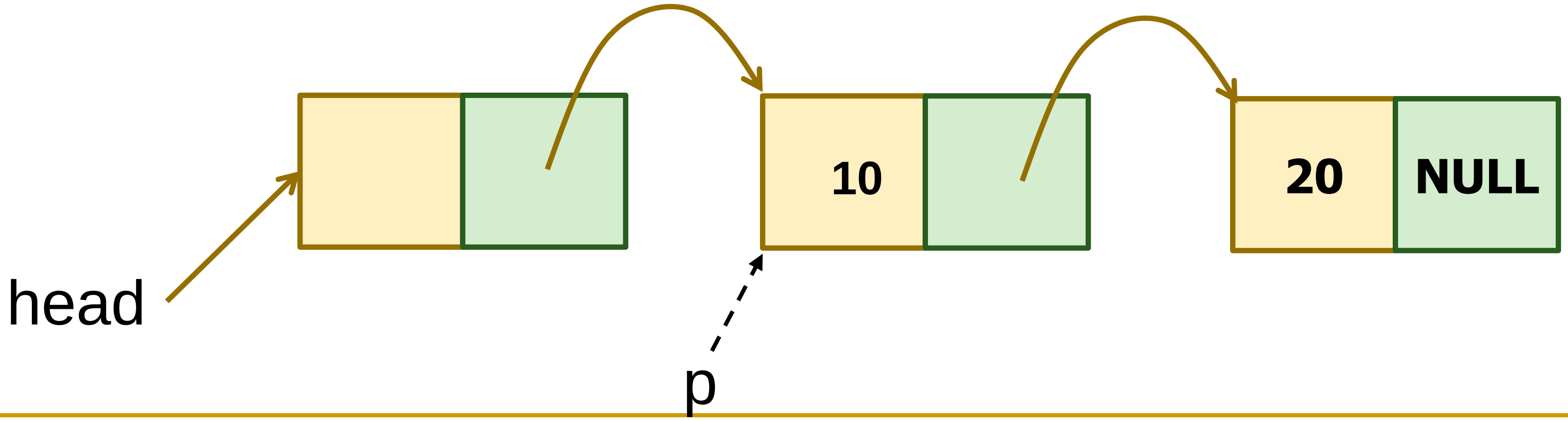
Find(2);



查找第k个结点

```
int Find(int k) {
    Node *p = head;
    for (int i=1;i<k;i++)
        p = p->next;
    Node *q = p->next;
    return q->data;
}
```

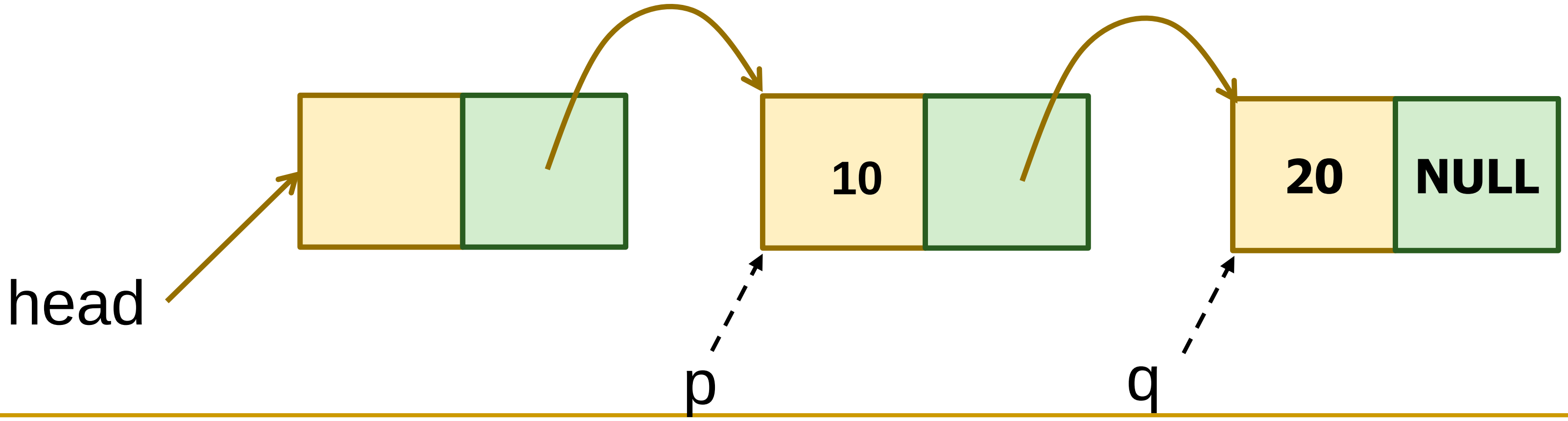
Find(2);



查找第k个结点

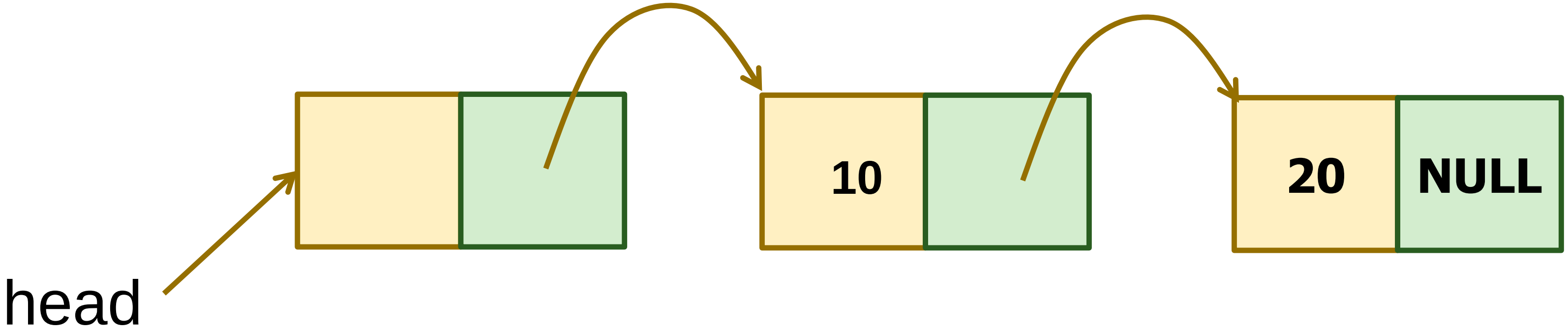
```
int Find(int k) {
    Node *p = head;
    for (int i=1;i<k;i++)
        p = p->next;
    Node *q = p->next;
    return q->data;
}
```

Find(2);

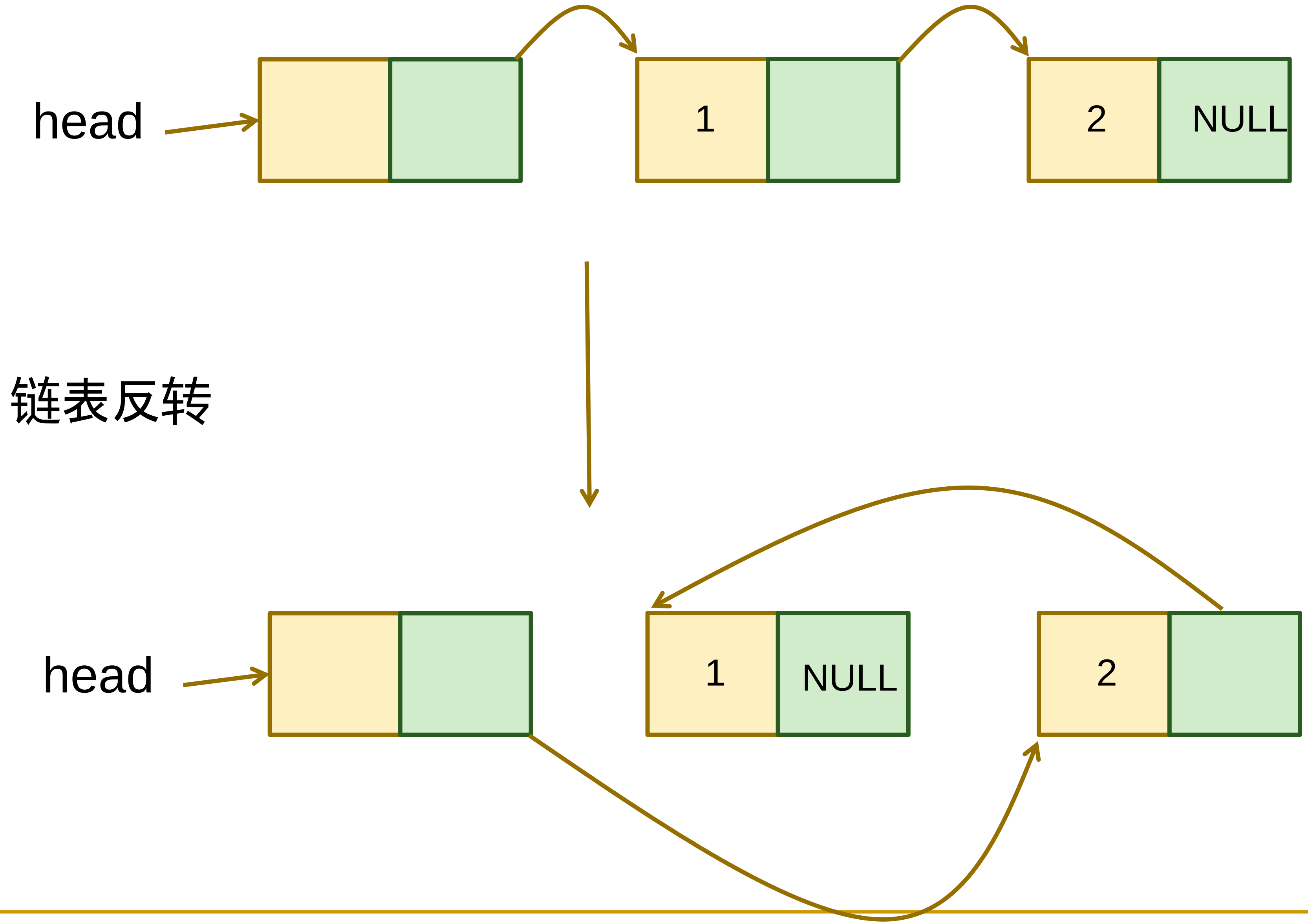


遍历所有结点

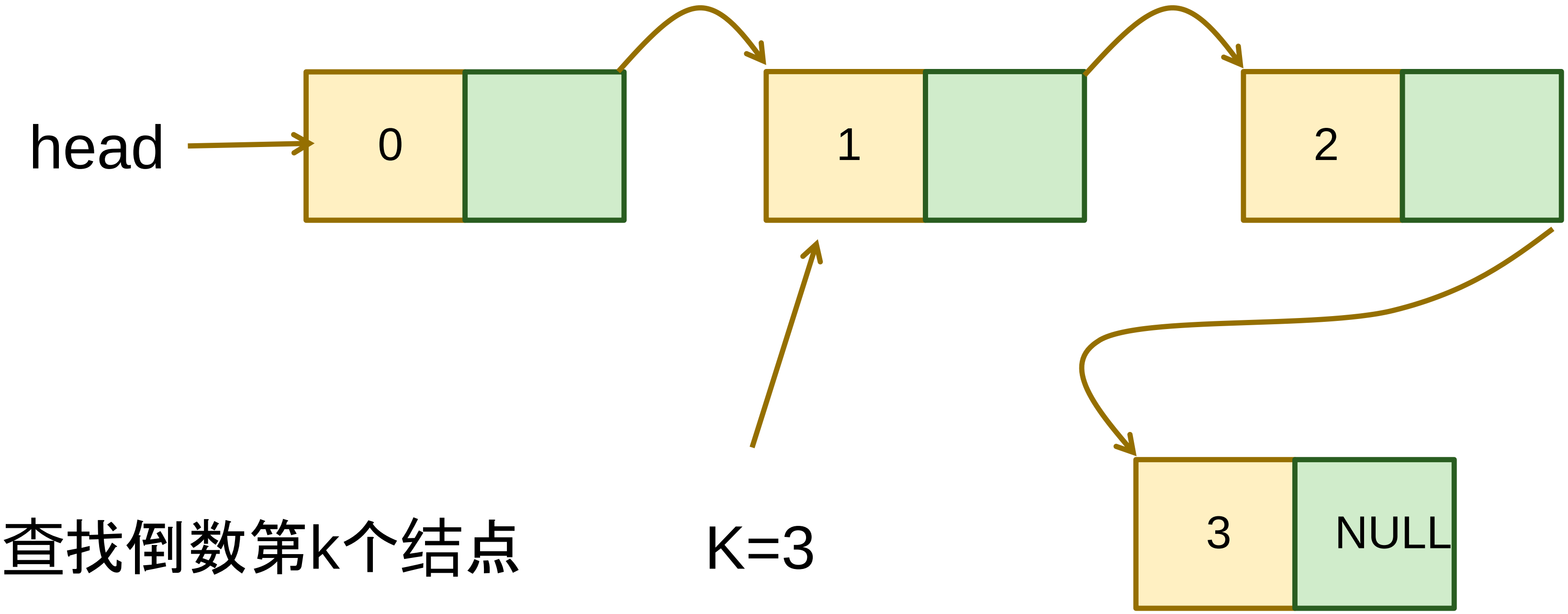
```
void Print() {
      Node *p = head;
      while(p){
          p = p->next;
          if(p != NULL)
              cout<<p->data<<endl;
      }
}
```



链表反转



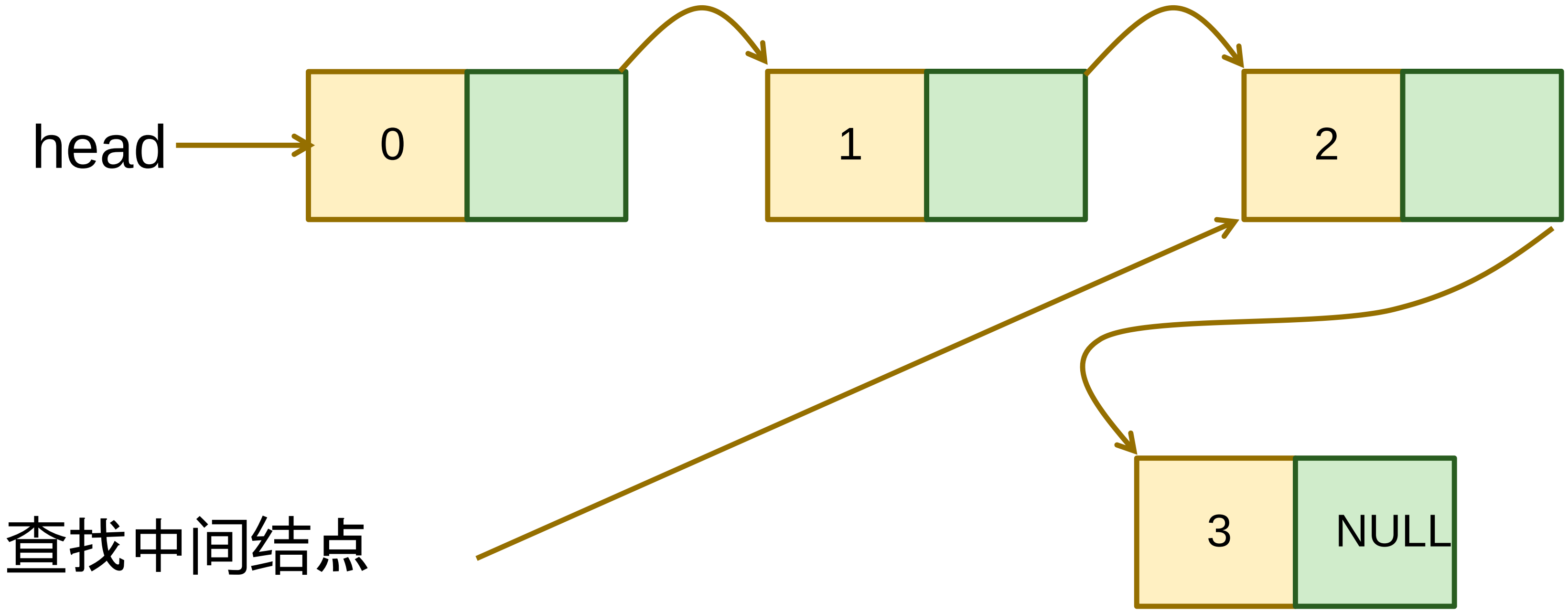
查找倒数第k个结点



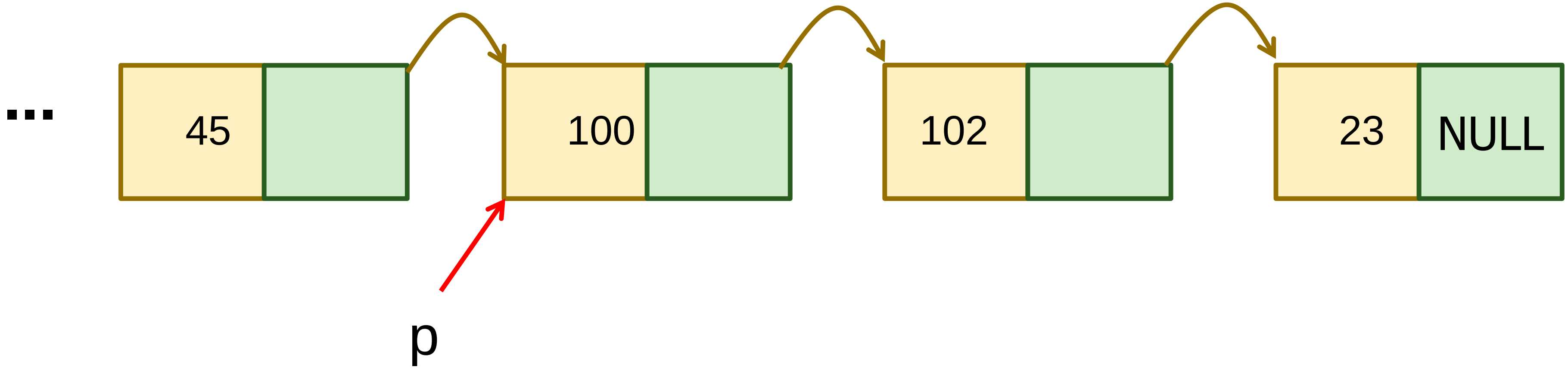
查找倒数第k个结点

K=3

查找中间结点

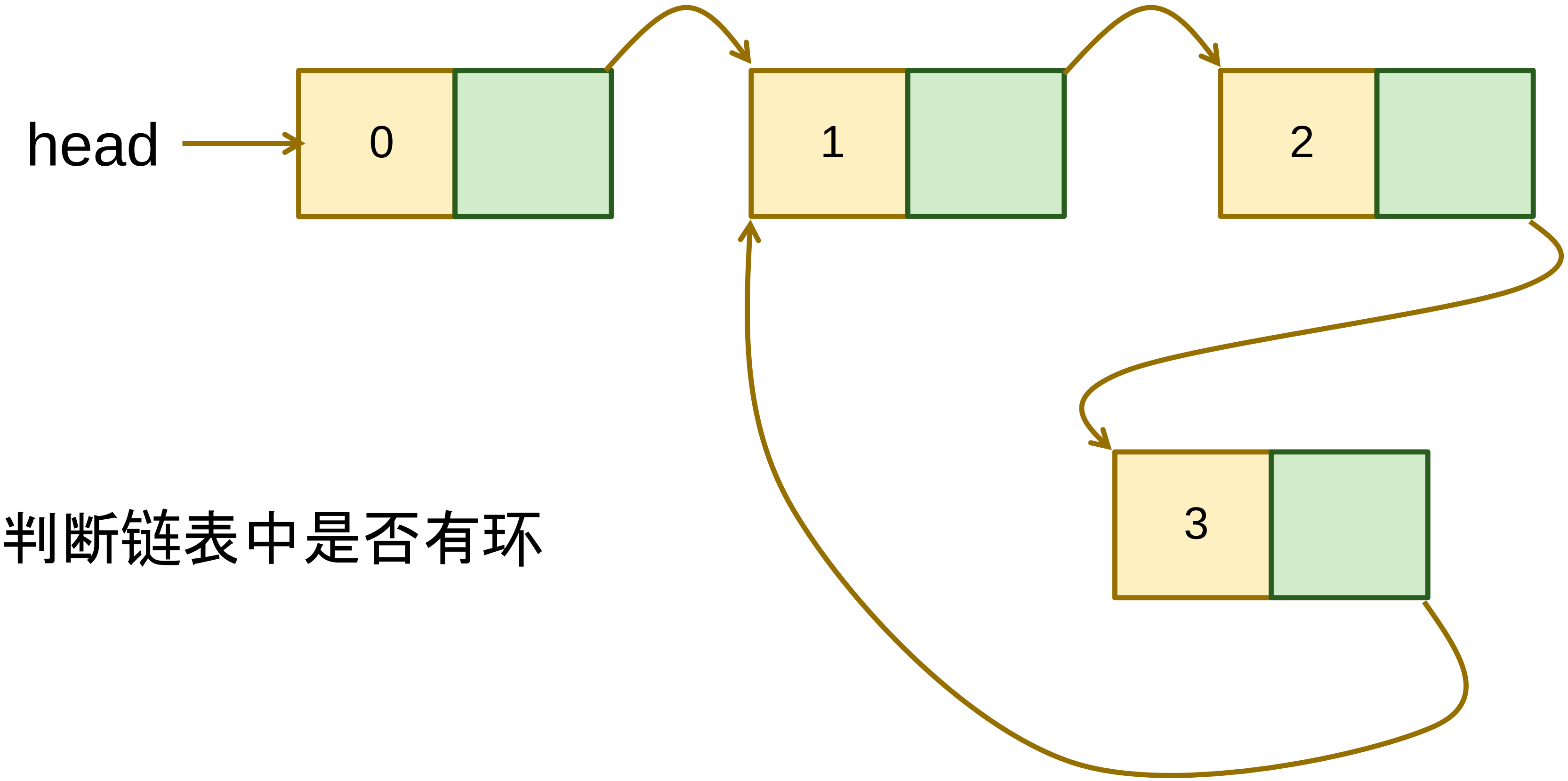


删除无头链表某个结点



不知道链表头指针，只知道中间某个节点的指针为p，如何删掉p所指向的结点？

判断是否有环



判断链表中是否有环

链表类 (Linked List)

```
class List {
  private:
      Node* head;
      int nodeNum; //结点个数
  public:
      List();
      ~List();
      void Insert(int k, int data);
      //在第k个位置插入新结点，数据为data
      void Delete(int k); //删除第k个结点
      int Find(int k); //查找第k个结点
      void Print(); //打印链表的所有元素
};
```

链表类 (Linked List)

```
List::List() {
    nodeNum = 0; //初始化结点个数
    head = new Node(0); //创建表头
}

List::~List() {
    for(int i = nodeNum; i >=1; i --) {
        Delete(i);
    }
    delete head;
}
```

链表类 (Linked List)

```
int List::Find(int k) {
    if(k < 1 || k > nodeNum)
        return -1; //元素不存在
    Node *p = head;
    for (int i=1;i<k;i++)
         p = p->next;
    Node *q = p->next;
    return q->data;
}
```

链表类 (Linked List)

```
void List::Insert(int k, int data) {
    if(k < 1 || k > nodeNum + 1)
        return; //插入位置不合法
    Node *newNode = new Node(data);
    Node *p = head;
    for(int i=1; i<k; i++)
        p = p->next;
    newNode->next = p->next;
    p->next = newNode;
    nodeNum ++;
}
```

链表类 (Linked List)

```
void List::Delete(int k) {
    if(k < 1 || k > nodeNum)
        return; //位置不合法
    Node *p = head;
    for (int i=1;i<k;i++)
         p = p->next;
    Node *q = p->next;
    p->next  = q->next;
    delete q;
    nodeNum --;
}
```

链表类 (Linked List)

```
void List::Print() {
    Node *p = head;
    while(p) {
        p = p->next;
        if(p != NULL)
            cout<<p->data<<endl;
    }
}
```

链表类 (Linked List)

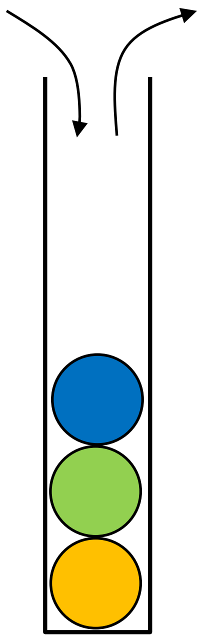
```
int main() {
    List *lt = new List();
    lt->Insert(1, 10);
    lt->Insert(2, 20);
    lt->Insert(3, 30);
    lt->Insert(4, 40);
    lt->Print();

    cout<<lt->Find(3)<<endl;

    lt->Delete(3);
    lt->Delete(1);
    lt->Print();
    return 0;
}
```

输出结果:

```
10
20
30
40
30
20
40
```

栈 (stack)

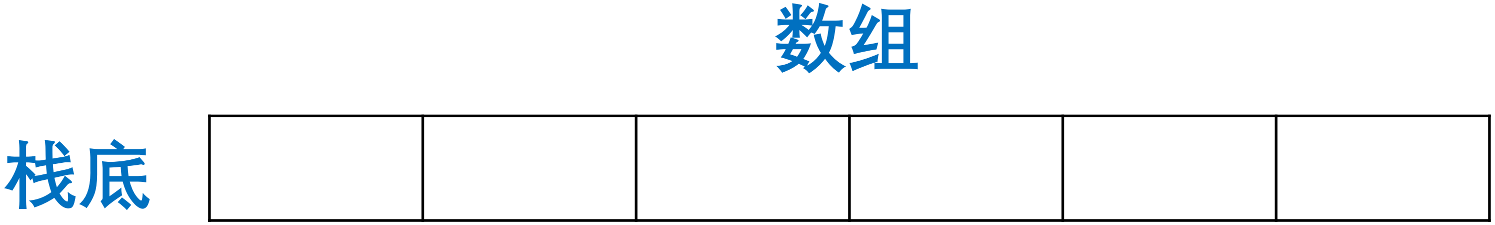
栈（Stack）



- 栈是一种线性数据结构，可以存入（插入）和取出（删除）元素
- 元素插入和删除操作都只能在栈的同一端进行，即“先入后出”
- 允许进行插入和删除操作的一端叫做栈顶(top)，另一端叫做栈底(bottom)
- 插入元素的操作称为入栈(push)，从栈中删除元素的操作称为出栈(pop)

栈

空栈

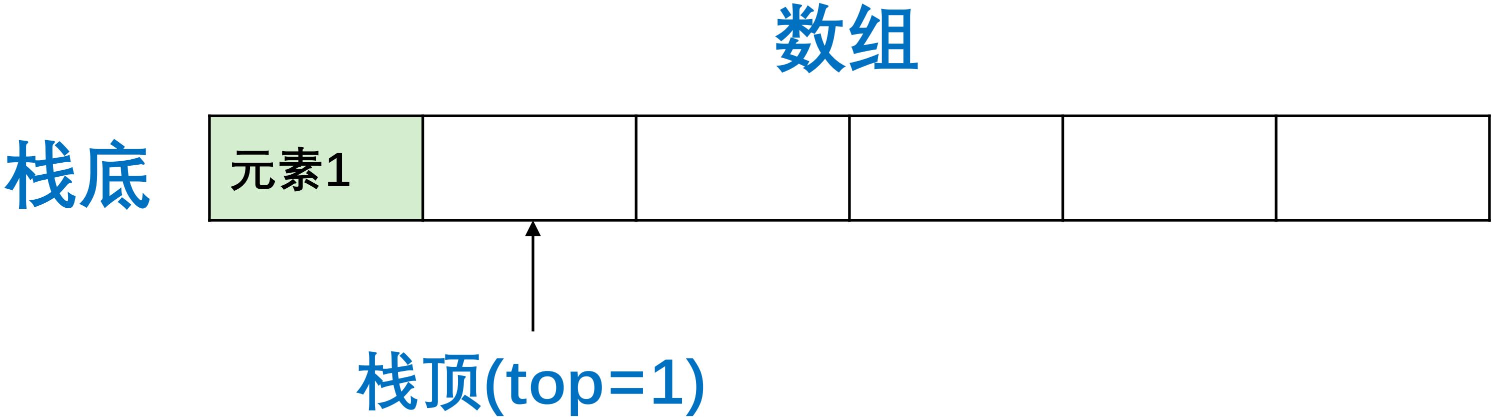


数组

栈底

栈

插入一个元素(push)



数组

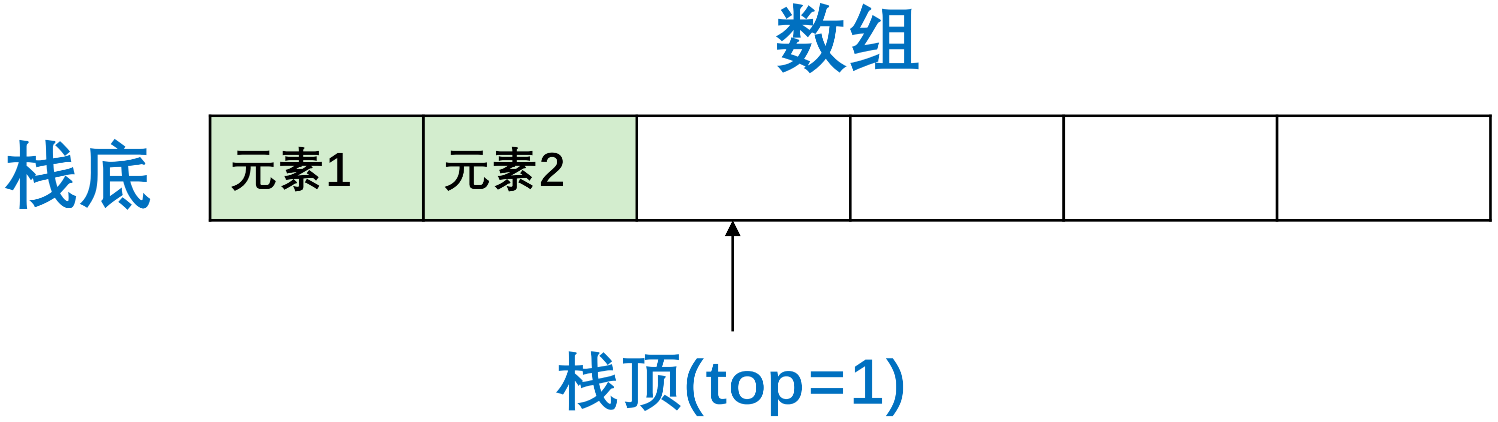
栈底

元素1

栈顶(top=1)

栈

再插入一个元素(push)



数组

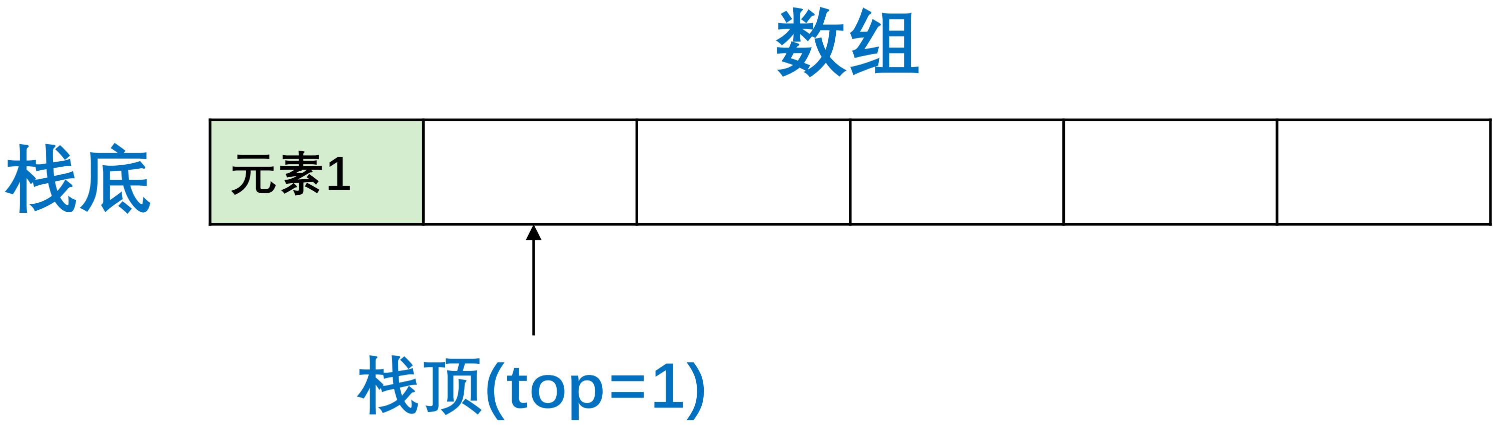
栈底

元素1	元素2				

栈顶(top=1)

栈

删除一个元素(pop)



数组

栈底

元素1

栈顶(top=1)

设计栈类

```
const int maxsize = 6; //栈大小，最多可存多少元素
class Stack {
      float data[maxsize];//用来存元素的数组
      int top {0};//栈顶
public:
      bool empty(void); //判断栈是否为空
      void push(float a); //插入一个元素
      float pop(void);//删除一个元素
};
bool Stack::empty(void) {
      return top == 0 ? true : false;
}
```

设计栈类

```
void Stack::push(float a) {
     if (top == maxsize) {
          cout<<“Stack is full!"<<endl;
          return;
     }
     data[top] = a;
     top++;
}
float Stack::pop(void) {
     if (top == 0) {
          cout<<"Stack is underflow!"<<endl;
          return 0;
     }
     top--;
     return data[top];
}
```

设计栈类

```
void main() {
     Stack s1, s2;
     for(int i = 1;i <= maxsize;i++)
         s1.push(2 * i);
     for(int i = 1;i <= maxsize;i++)
         cout<<s1.pop()<<"  ";
     for(int i = 1;i <= maxsize;i++)
         s1.push(2.5*i);
     for(int i = 1;i <= maxsize;i++)
         s2.push(s1.pop());
     cout<<endl;
     do
          cout<<s2.pop()<<"  ";
     while (!(s2.empty()));
}
```

设计栈类

程序运行结果：

```
12  10  8  6  4  2
2.5  5  7.5  10  12.5  15
```

栈的应用-中缀表达式转后缀表达式

中缀表达式: 常用的算术表达式,运算符在运算数中间,运算需要考虑优先级

2 + 9 / 3 - 5

后缀表达式: 计算机用的算术表达式,运算符在运算数后面,从左到右运算，无需考虑优先级

2 9 3 / + 5 -

栈的应用-中缀表达式转后缀表达式

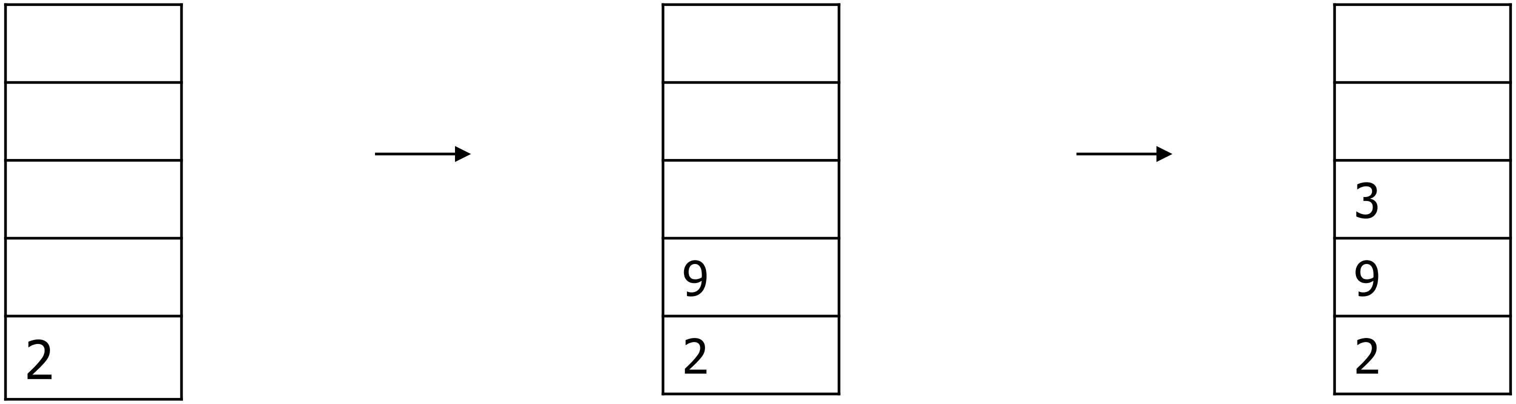
计算机如何计算后缀表达式?

2 9 3 / + 5 -

栈的应用-中缀表达式转后缀表达式

从左到右读取表达式，遇到操作数，入栈，遇到运算符就弹出相应的运算数，先弹出的作为右操作数，后弹出的作为左操作数，运算后再把结果入栈，最终结果就是栈顶的值

2 9 3 / + 5 -

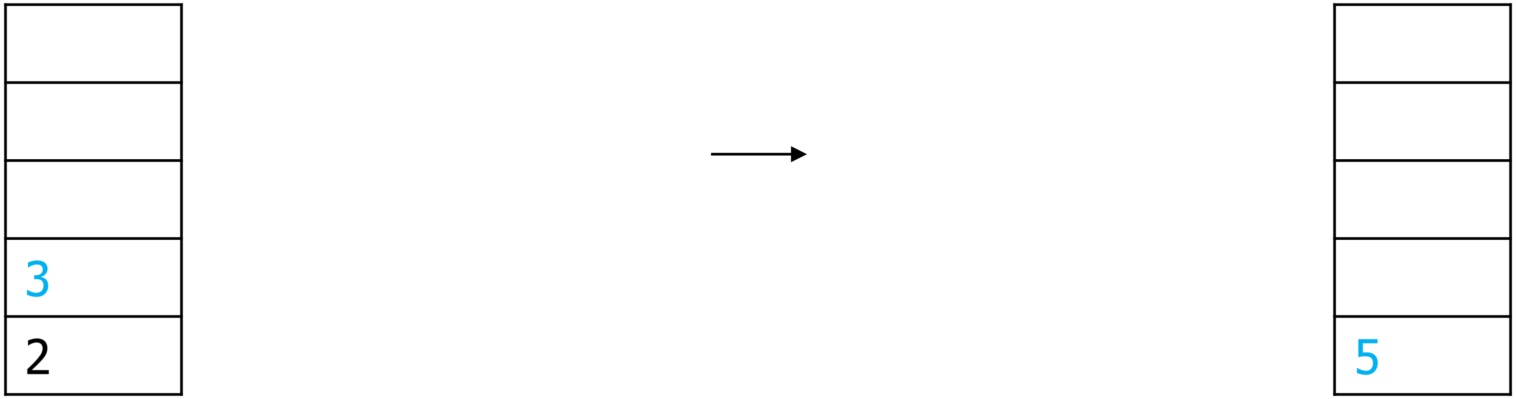


读取2，入栈　　读取9，入栈　　读取3，入栈

栈的应用-中缀表达式转后缀表达式

从左到右读取表达式，遇到操作数，入栈，遇到运算符就弹出相应的运算数，先弹出的作为右操作数，后弹出的作为左操作数，运算后再把结果入栈，最终结果就是栈顶的值

2 9 3 / + 5 -



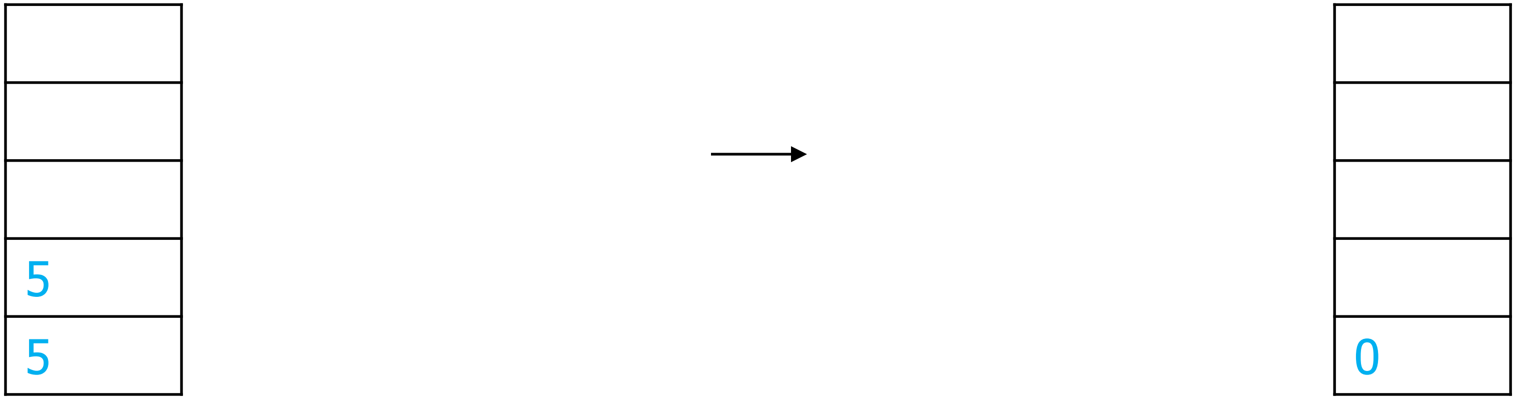
读取/，3和9出栈，
计算9/3，结果3入栈

读取+，3和2出栈，
计算2+3，结果5入栈

栈的应用-中缀表达式转后缀表达式

从左到右读取表达式，遇到操作数，入栈，遇到运算符就弹出相应的运算数，先弹出的作为右操作数，后弹出的作为左操作数，运算后再把结果入栈，最终结果就是栈顶的值

2 9 3 / + 5 -



读取5，入栈

读取-，5和5出栈，计算5-5，结果0入栈，为最终结果

栈的应用-中缀表达式转后缀表达式

如何将中缀表达式转换为后缀表达式?

2 + 9 / 3 – 5 -> 2 9 3 / + 5 -

1. 从左到右进行遍历
2. 运算数，直接输出
3. 左括号，直接压入堆栈
4. 右括号，不断弹出栈顶运算符并输出直到遇到左括号(弹出但不输出)
5. 运算符，如果是栈顶是左括号，直接将运算符入栈，否则将该运算符与栈顶运算符进行比较，如果优先级高于栈顶运算符则压入堆栈，如果优先级低于等于栈顶运算符则将栈顶运算符弹出并输出，然后比较新的栈顶运算符，直到优先级大于栈顶运算符或者栈空，再将该运算符入栈
6. 如果对象处理完毕，则按顺序弹出并输出栈中所有运算符

栈的应用-中缀表达式转后缀表达式

2 + 9 / 3 – 5 -> 2 9 3 / + 5 -

步骤	待处理表达式	栈顶状态（低->高）	输出状态
1	2 + 9 / 3 - 5		
2	+ 9 / 3 - 5		2
3	9 / 3 - 5	+	2
4	/ 3 - 5	+	2 9
5	3 - 5	+ /	2 9
6	- 5	+ /	2 9 3
7	5	-	2 9 3 / +
8		-	2 9 3 / + 5
9			2 9 3 / + 5 -

栈的应用-中缀表达式转后缀表达式

2*(9+6/3–5) -> 2 9 6 3 / + 5 - * 4 +

步骤	待处理表达式	栈顶状态（低→高）	输出状态
1	2*(9+6/3-5)+4		
2	*(9+6/3-5)+4		2
3	(9+6/3-5)+4	*	2
4	9+6/3-5)+4	* (	2
5	+6/3-5)+4	* (	2 9
6	6/3-5)+4	* (+	2 9
7	/3-5)+4	* (+	2 9 6
8	3-5)+4	* (+ /	2 9 6
9	-5)+4	* (+ /	2 9 6 3

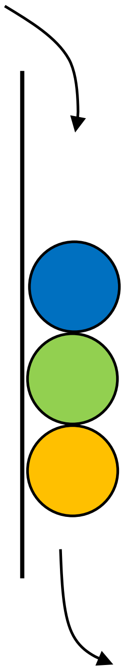
栈的应用-中缀表达式转后缀表达式

2*(9+6/3–5) -> 2 9 6 3 / + 5 - * 4 +

步骤	待处理表达式	栈顶状态（低→高）	输出状态
10	5)+4	* (-	2 9 6 3 / +
11	)+4	* (-	2 9 6 3 / + 5
12	+4	*	2 9 6 3 / + 5 -
13	4	+	2 9 6 3 / + 5 - *
14		+	2 9 6 3 / + 5 - * 4
15			2 9 6 3 / + 5 - * 4 +

队列 (queue)

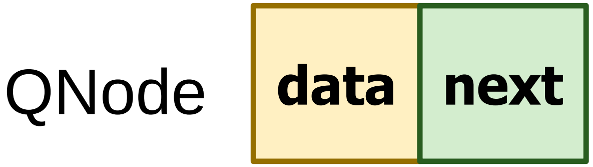
队列（Queue）



- 队列是一种线性数据结构，可以存入（插入）和取出（删除）元素
- 元素插入和删除操作在不同端进行，即“先入先出”
- 允许插入的一端叫队列尾(rear)，允许删除的一端叫队列头(front)
- 插入元素的操作称为入队(queue)，从队列中删除元素的操作称为出队(dequeue)

结点（QNode）

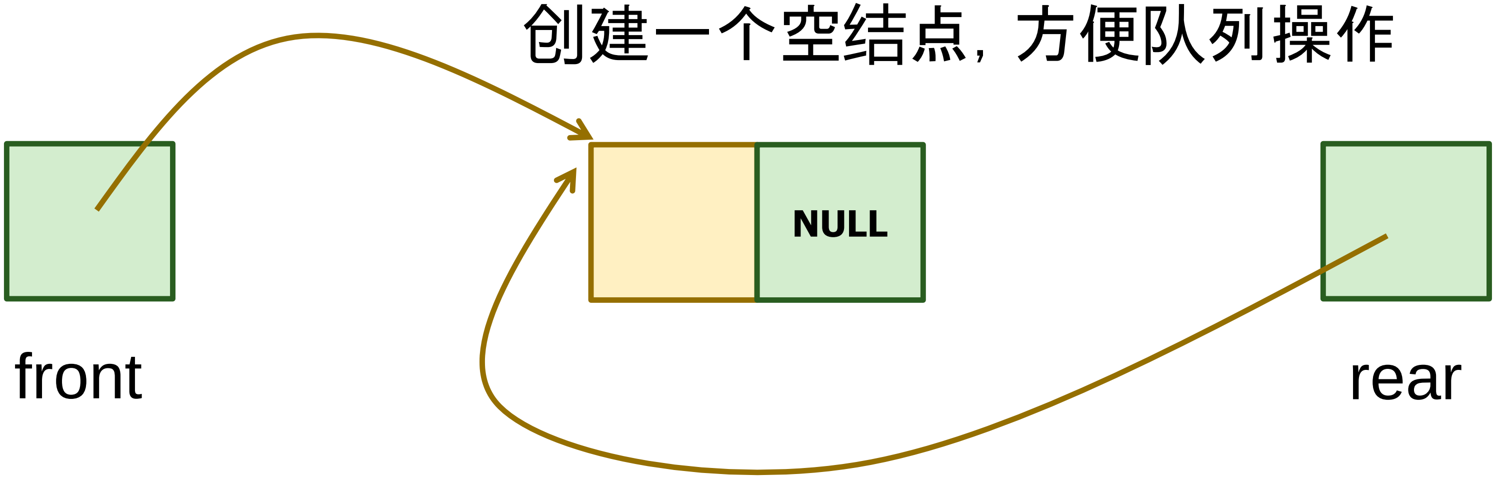
```
class QNode{
  public:
      int data;
      QNode* next;
};
```



队列初始化

```
QNode *front; //队列头
QNode *rear; //队列尾
front = rear = new Qnode();
front->next = rear;
rear->next = nullptr;
```

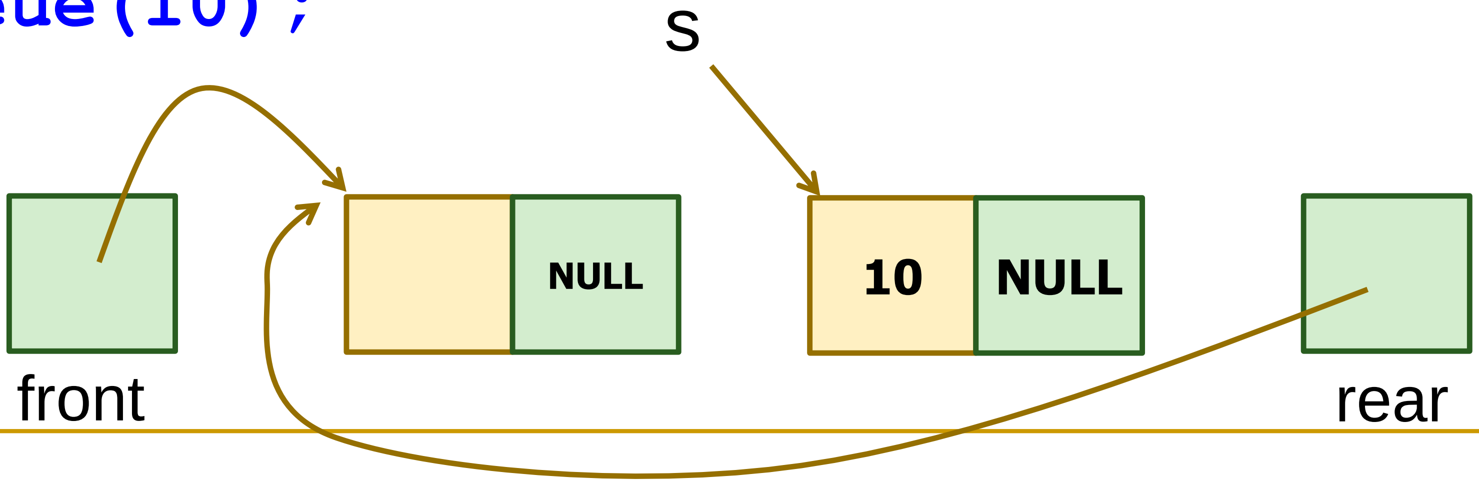
创建一个空结点，方便队列操作



入队列

```
void queue(int x) {
    QNode *s = new QNode();
    s->data = x;
    s->next = nullptr;
    rear->next = s;
    rear = s;
}
```

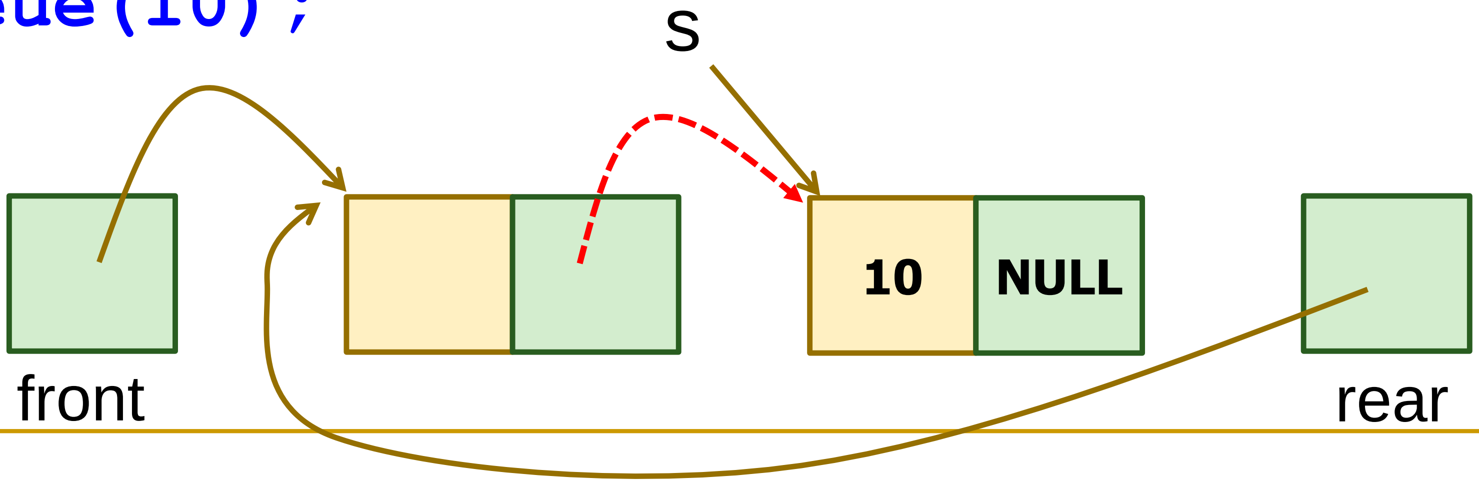
```
queue(10);
```



入队列

```
void queue(int x) {
    QNode *s = new QNode();
    s->data = x;
    s->next = nullptr;
    rear->next = s;
    rear = s;
}
```

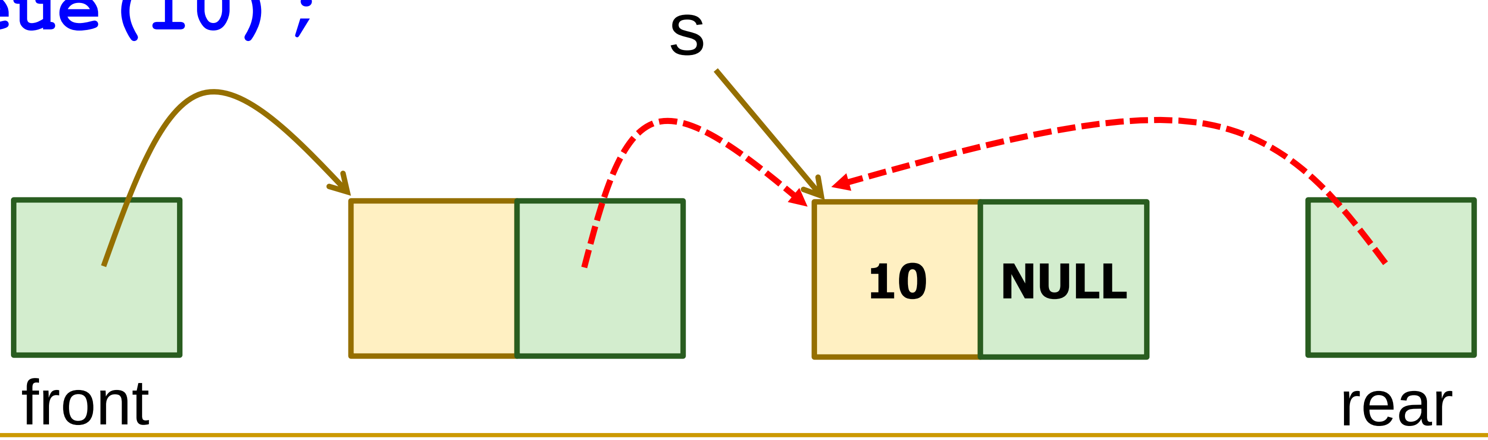
```
queue(10);
```



入队列

```
void queue(int x) {
    QNode *s = new QNode();
    s->data = x;
    s->next = nullptr;
    rear->next = s;
    rear = s;
}
```

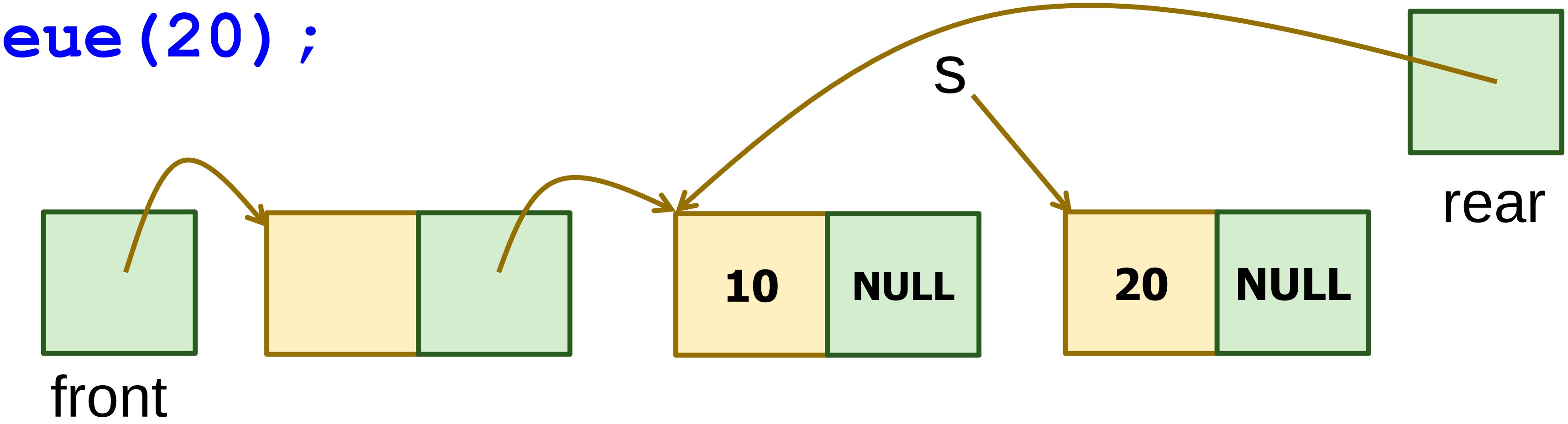
```
queue(10);
```



入队列

```
void queue(int x) {
    QNode *s = new QNode();
    s->data = x;
    s->next = nullptr;
    rear->next = s;
    rear = s;
}
```

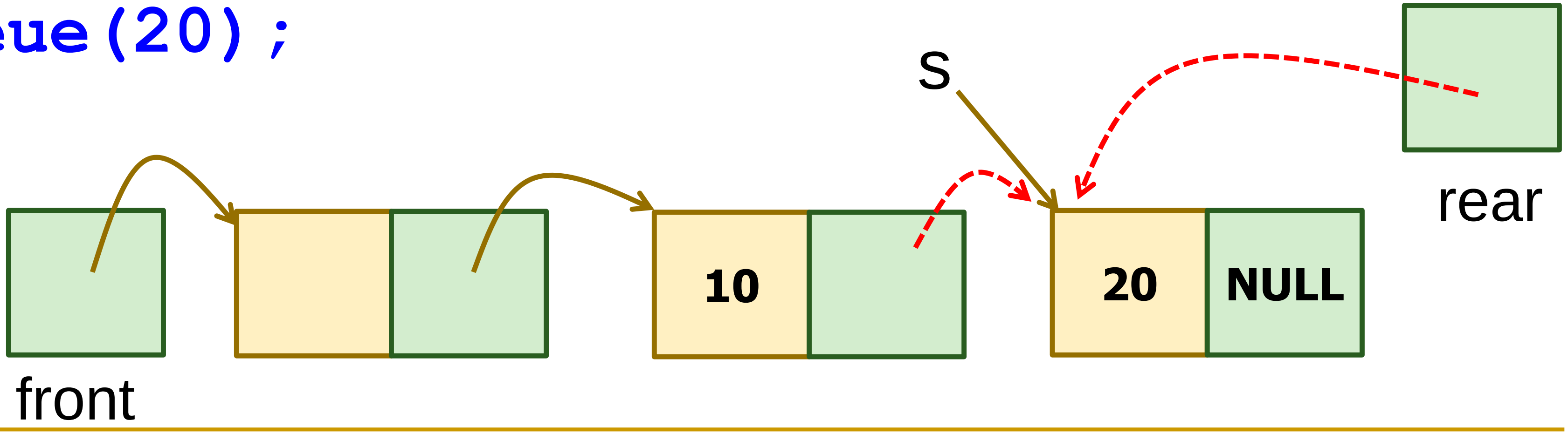
```
queue(20);
```



入队列

```
void queue(int x) {
    QNode *s = new QNode();
    s->data = x;
    s->next = nullptr;
    rear->next = s;
    rear = s;
}
```

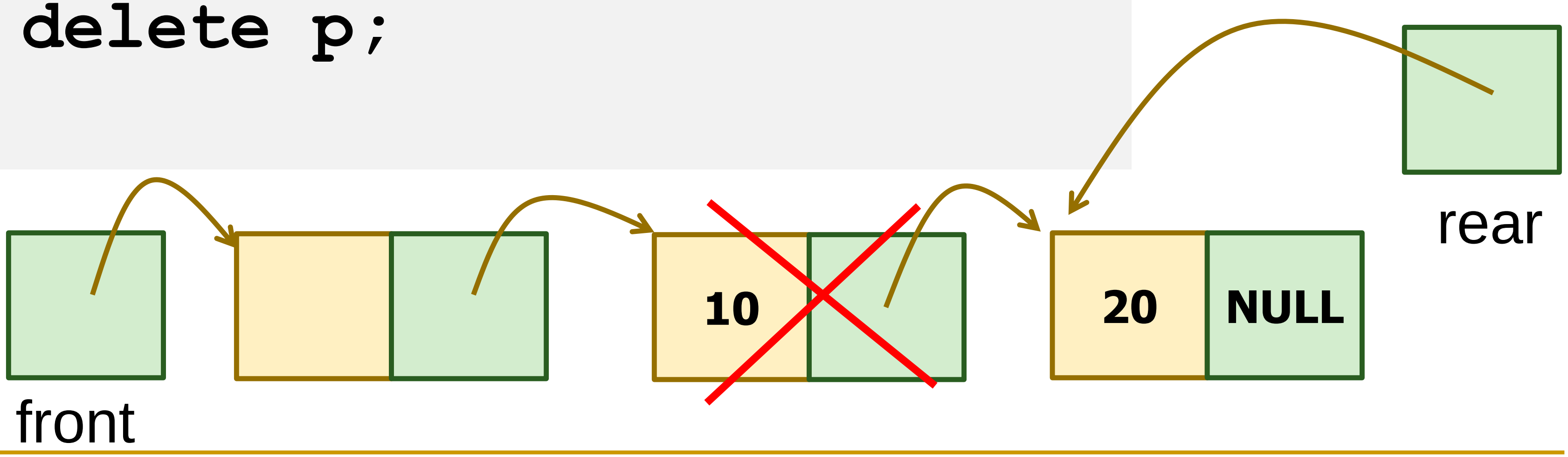
```
queue(20);
```



出队列

```
void dequeue() {
    if (empty()) {
        cout<<"Queue is empty";
        return;
    }
    QNode *p = front;
    front = front->next;
    delete p;
}
```

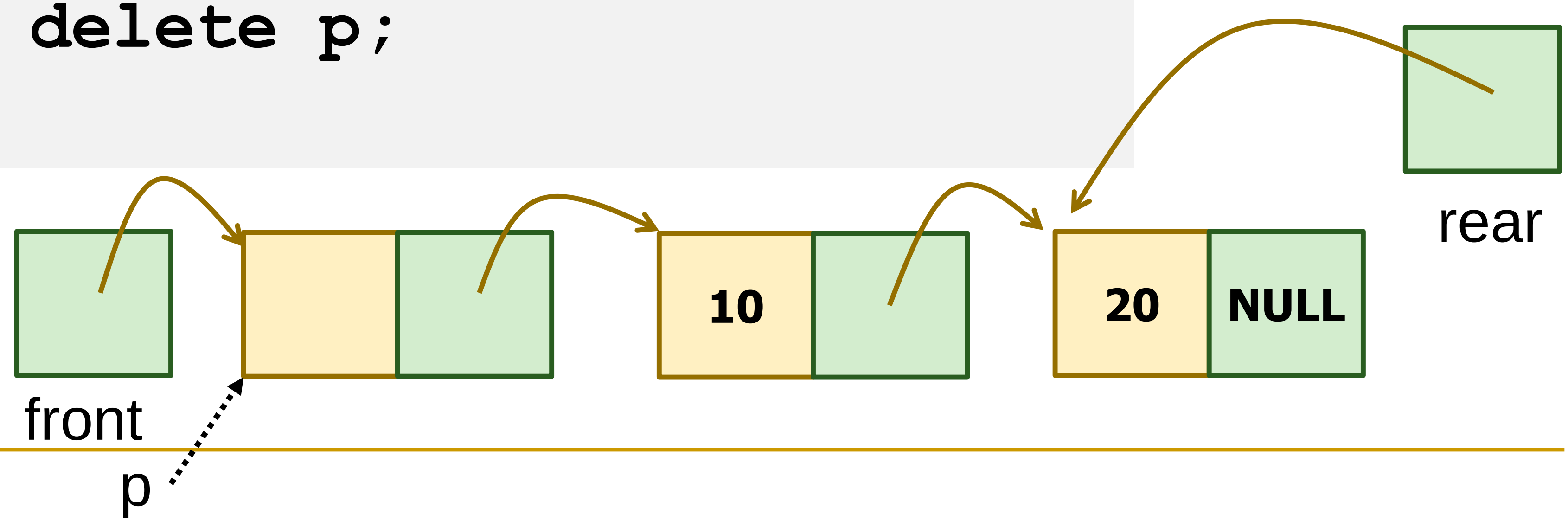
dequeue();



出队列

```
void dequeue() {
    if (empty()) {
        cout<<"Queue is empty";
        return;
    }
    QNode *p = front;
    front = front->next;
    delete p;
}
```

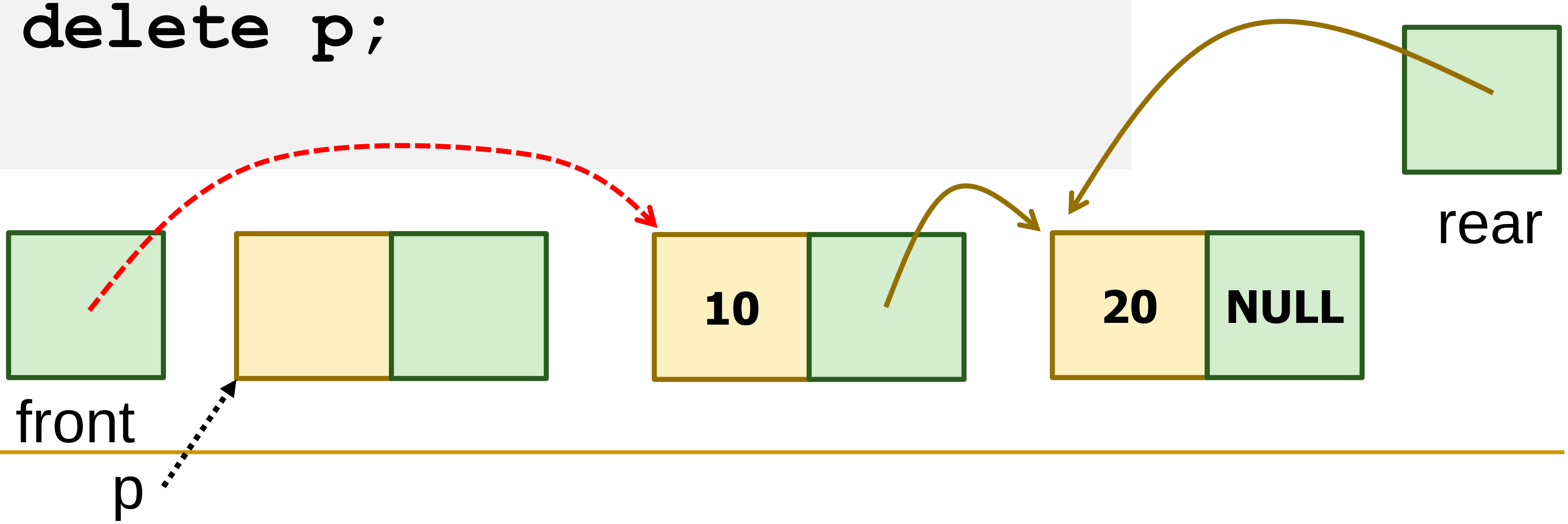
dequeue();



出队列

```
void dequeue() {
   if (empty()) {
     cout<<"Queue is empty";
     return;
   }
   QNode *p = front;
   front = front->next;
   delete p;
}
```

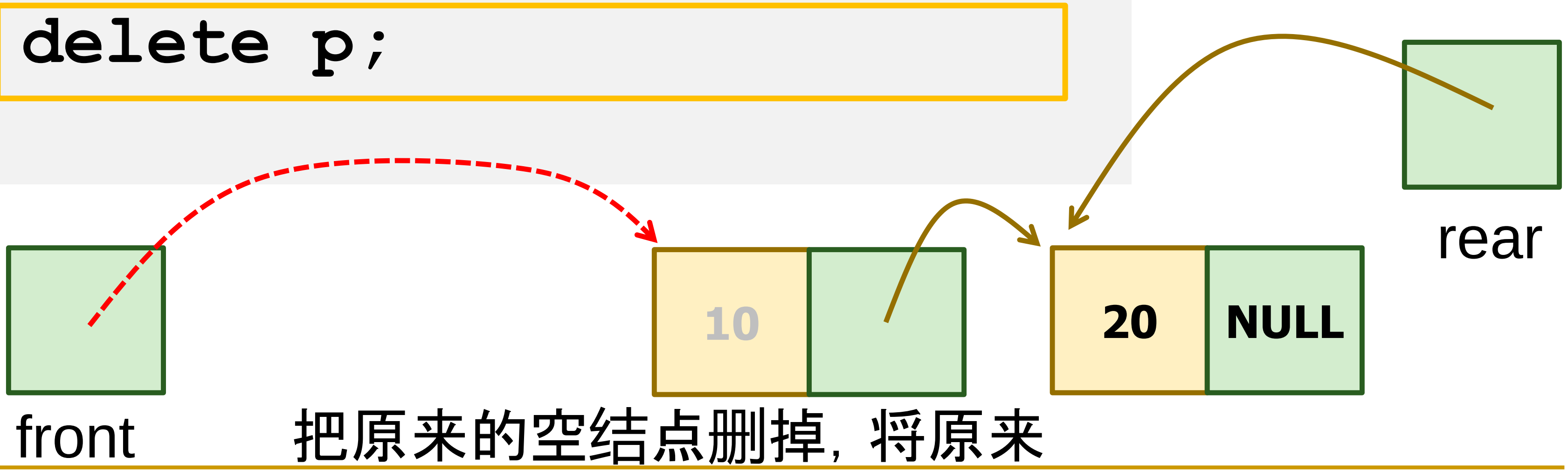
dequeue();



出队列

```
void dequeue() {
    if (empty()) {
       cout<<"Queue is empty";
       return;
    }
    QNode *p = front;
    front = front->next;
    delete p;
}
```

dequeue();



把原来的空结点删掉，将原来的第一个结点变为空结点！

队列类实现

```
class QNode {
public:
   int data;
   QNode *next;
};
```

```
class LQueue {
public:
   QNode *front;
   QNode *rear;
   void initQueue();
   int empty();
   void queue(int);
   void dequeue();
   int getFront();
   int getRear();
   void printQueue();
};
```

队列类实现

```
void LQueue::initQueue() {
   front = rear = new QNode();
   front->next = rear;
   rear->next = nullptr;
}
int LQueue::empty() {//判断队列是否为空
   return front->next == rear;
}
void LQueue::queue(int x) {//入队列
    QNode *s = new QNode();
    s->data = x;
    s->next = nullptr;
    rear->next = s;
    rear = s;
}
```

队列类实现

```
void LQueue::dequeue() {
   if (empty()) {
     cout << "LQueue is empty,can't pop";
     return;
   }
   QNode *p = front;
   front = front->next;
   delete p;
}
int LQueue::getFront() {
   return front->next->data;
}
int LQueue::getRear() {
   return rear->data;
}
```

队列类实现

```
void LQueue::printQueue() {
    QNode *q = front->next;
    while (q) {
        cout << q->data << " ";
        q = q->next;
    }
    cout << "\n";
}
```

队列类实现

```
int main() {
  LQueue L;
  L.initQueue();
  srand((unsigned)time(NULL));
  for (int i = 0;i < 10;i++) {
   L.queue(rand() % 100); //0-100之间的随机数
  }
  L.printQueue();
  L.dequeue();
  L.dequeue();
  L.printQueue();
  return 0;
}
```

输出结果:

```
53 94 43 60 74 97 25 4 99 38
43 60 74 97 25 4 99 38
```

END