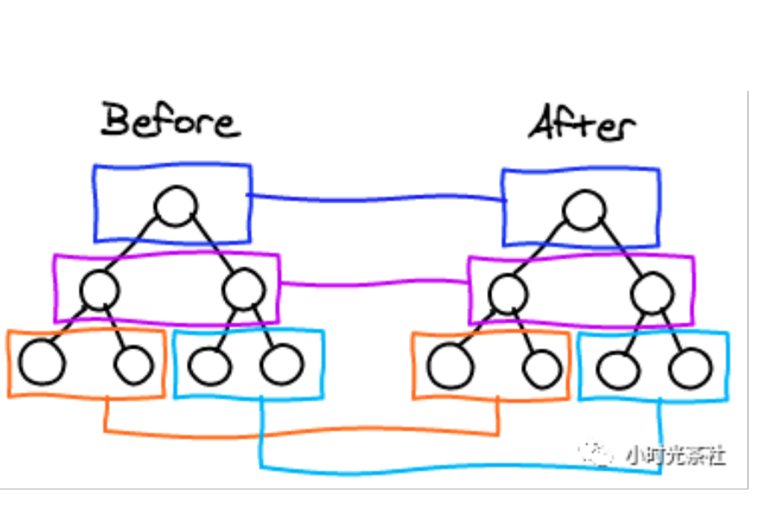


为了实现高效的DOM操作，一套高效的虚拟DOM diff算法显得很有必要。

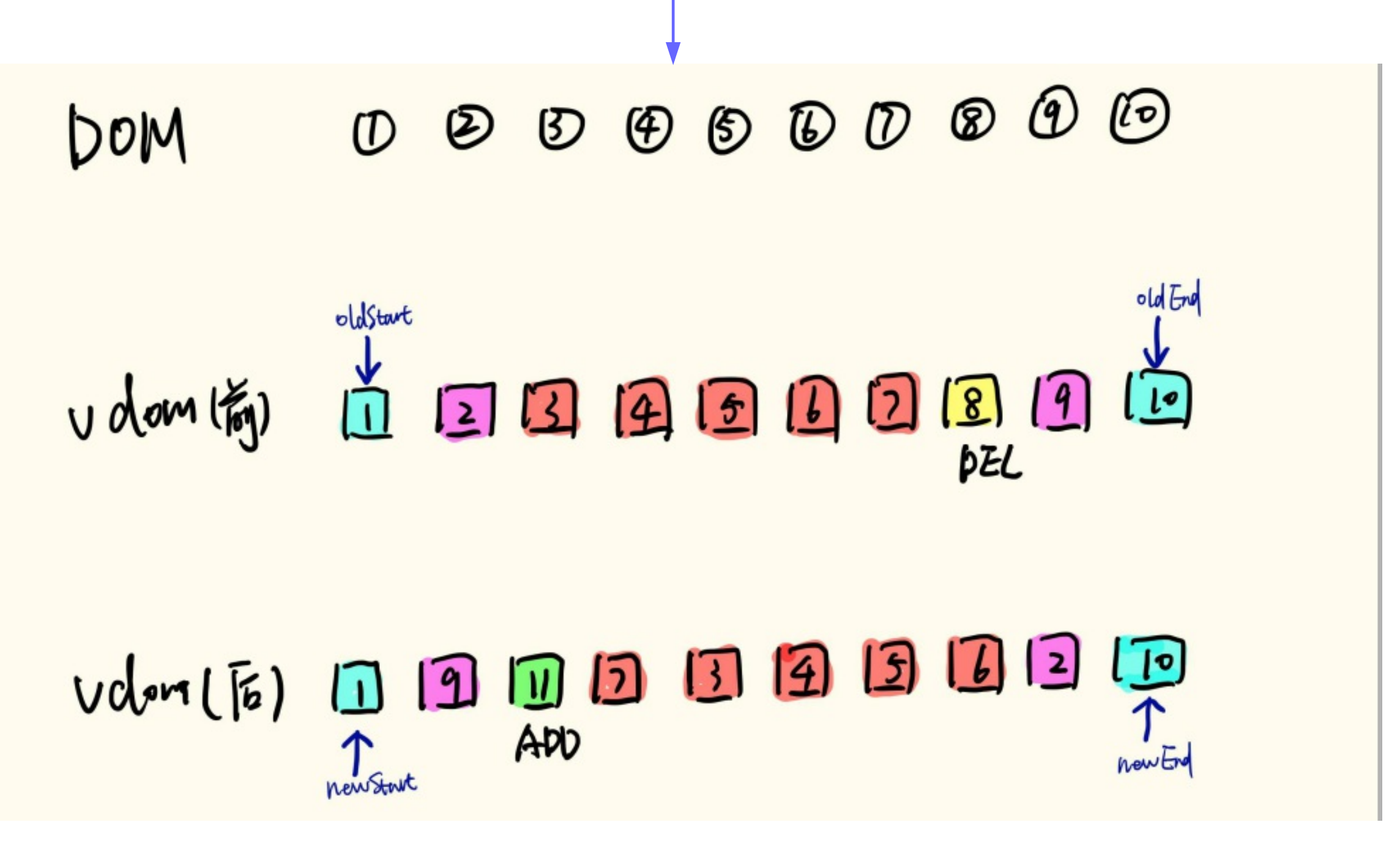
Vue的diff算法是基于snabbdom改造过来的

这是一张很经典的图，出自《React' s diff algorithm》，Vue的diff算法也同样，

即仅在同级的vnode间做diff，递归地进行同级vnode的diff，最终实现整个DOM树的更新。

那同级vnode diff的细节又是怎样的呢？正是本文所要讲的。

下面先考虑一个简单的方法来完成diff 和 dom更新



第一次比较: 头部相同、尾部相同的节点：如1、10，

新 DOM : 1 2 3 4 5 6 7 8 9 10

old vdom 标记 1 和 10 为已处理 new vdom 的1 和10 也标记为已经处理

第二次比较: 头尾相同的节点：如2、9（处理完头部相同、尾部相同节点之后）

new 中 9 的位置是 => 2

在old中 找到9这个值的位置 => 9

在真实 DOM中 把第9个元素直接移动到第二个元素的前面

新 DOM : 1 9 3 4 5 6 7 8 2 10

old vdom 标记 2 和 9 为已处理

第三次比较: 新增的节点：11, 并放在dom列表第三个元素前面

新 DOM : 1 9 11 3 4 5 6 7 8 2 10 ;

new vdom 标记 11 为已经处理

第四次比较: 找到7, 把它放入dom中第4个位置

新 DOM : 1 9 11 7 3 4 5 6 8 2 10

newvdom: 1 9 11 7 3 4 5 6 2 10

标记 oldvdom的 7 为已处理

第五次比较:

3 4 5 6 位置一样, 节点也一样, 标记为已处理, 此时 new vdom 的元素都已经被标记完, 说明已处理完

新 DOM : 1 9 11 7 3 4 5 6 8 2 10

newvdom: 1 9 11 7 3 4 5 6 2 10

在 oldvdom中 最后剩下 8 没有被标记为已处理, 则删除 对应真实 dom中的 元素8

vue也采用了类似的标记处理, 处理过的节点Vue会在oldVdom和newVdom中同时将它标记为已处理

但vue做了优化

1.同类型的的节点不做移动, 原地复用, 只更新其内容

比如2个不同的div，在DOM上它们是不一样的，但是它们属于同类节点

如果10个都是div, 那么整个diff过程中就没有移动DOM的操作了。

2. 定义两对指针, 逐渐缩小范围

Vue不断对vnode进行处理同时移动指针

直到其中任意一对起点和终点相遇。

（1）、第一部分是一个循环，循环内部是一个分支逻辑，

每次循环只会进入其中的一个分支，每次循环会处理一个节点，

处理之后将节点标记为已处理

（oldVdom和newVdom都要进行标记，

如果节点只出现在其中某一个vdom中，则另一个vdom中不需要进行标记），

标记的方法有2种，当节点正好在vdom的指针处，移动指针将它排除到未处理列表之外即可，

否则就要采用其他方法，Vue的做法是将节点设置为undefined。

（2）、循环结束之后，可能newVdom或者oldVdom中还有未处理的节点，

如果是newVdom中有未处理节点，则这些节点是新增节点，做新增处理。

如果是oldVdom中有这类节点，则这些是需要删除的节点，相应在DOM树中删除之

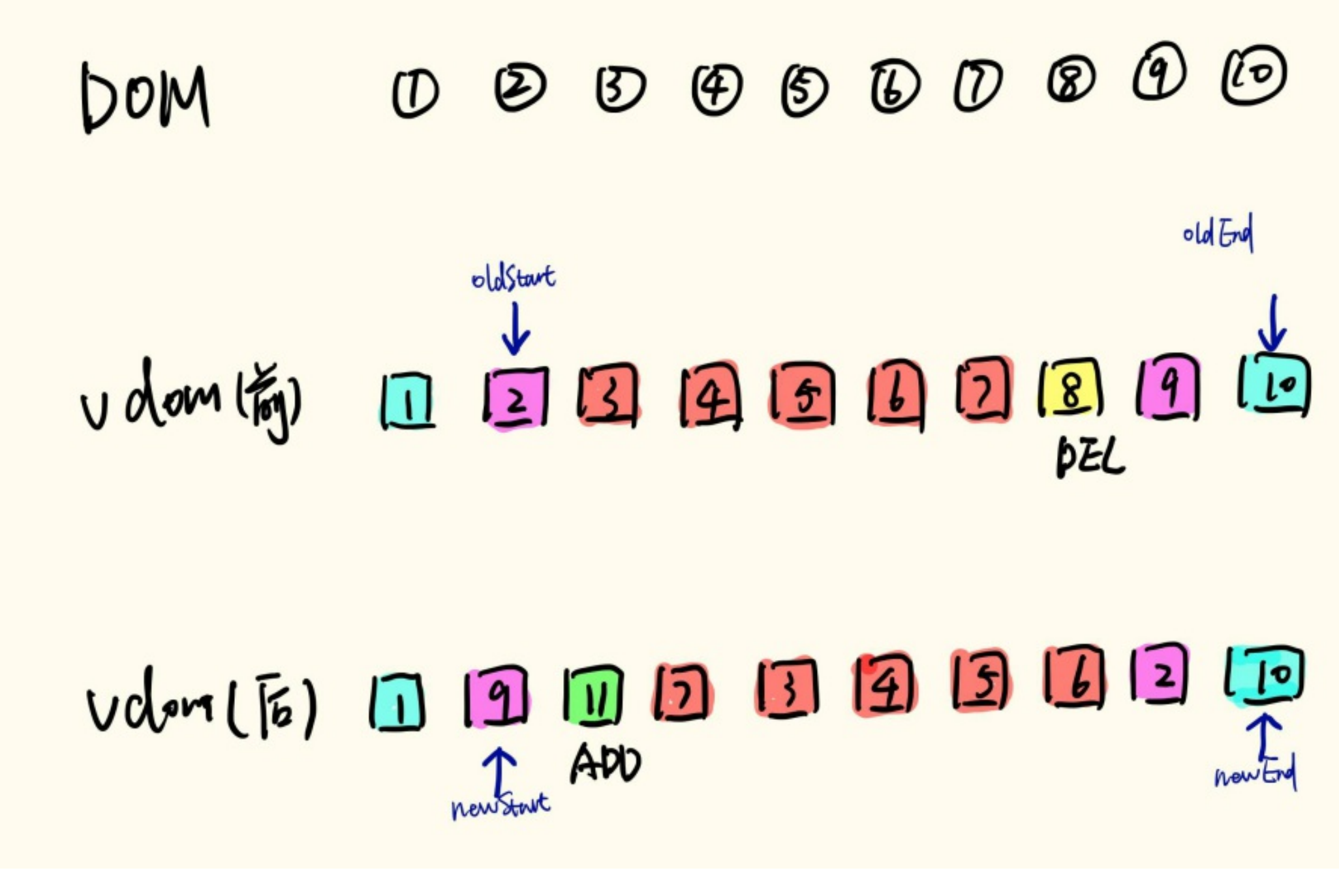
整个过程是逐步找到更新前后vdom的差异，然后将差异反应到DOM树上（也就是patch），

特别要提一下Vue的patch是即时的，并不是打包所有修改最后一起操作DOM

（React则是将更新放入队列后集中处理），朋友们会问这样做性能很差吧？

实际上现代浏览器对这样的DOM操作做了优化，并无差别。

vue diff 分步解析
不同颜色代表不同的类型,
比如div 和 span是不同的类型



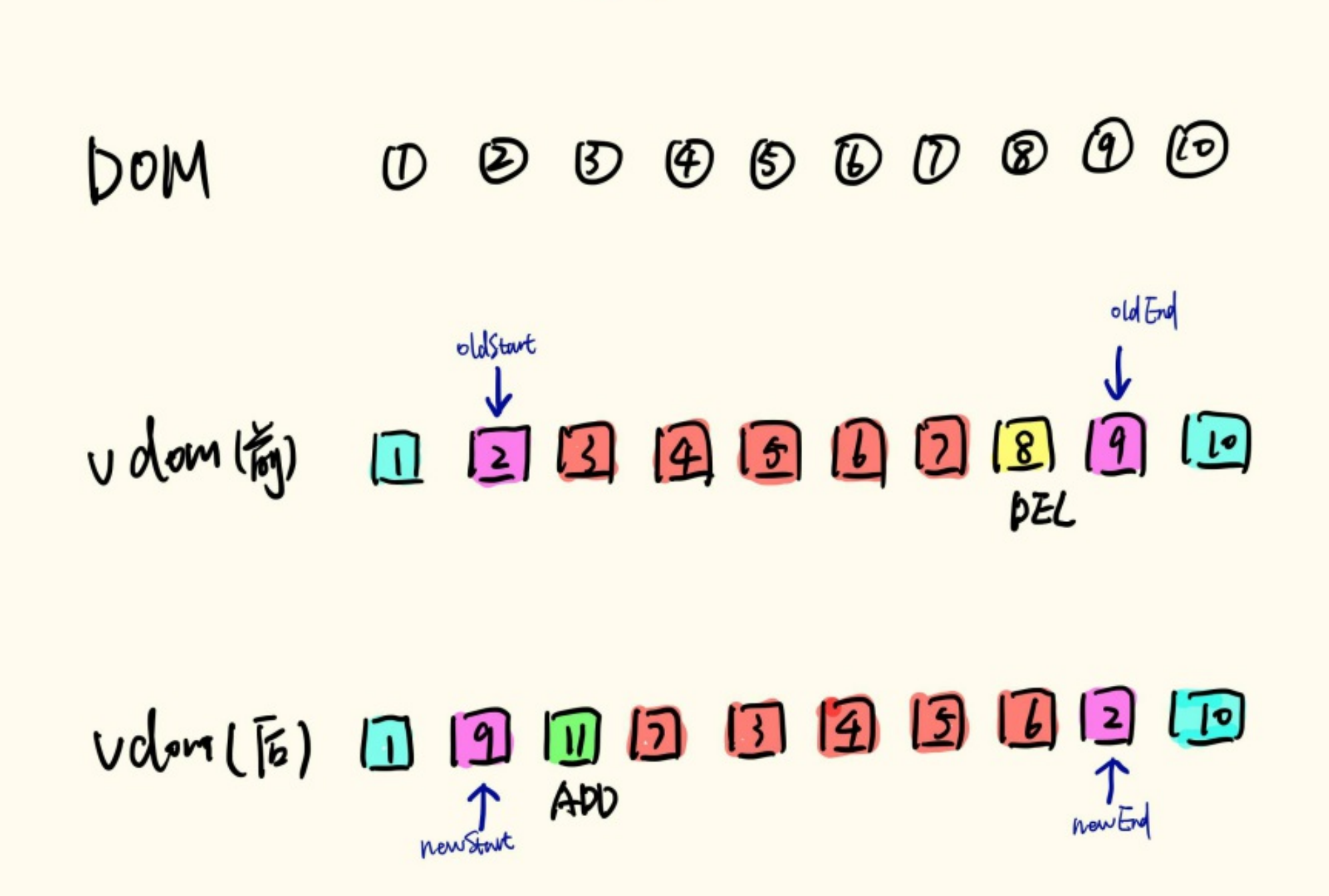
（1）、处理头部的同类型节点，即oldStart和newStart指向同类节点的情况，如上图中的节点1

这种情况下，将节点1的变更更新到DOM，然后对其进行标记，

标记方法是oldStart和newStart后移1位即可，过程中不需要移动DOM

（更新DOM或许是要的，比如属性变更了，文本内容变更了等等）

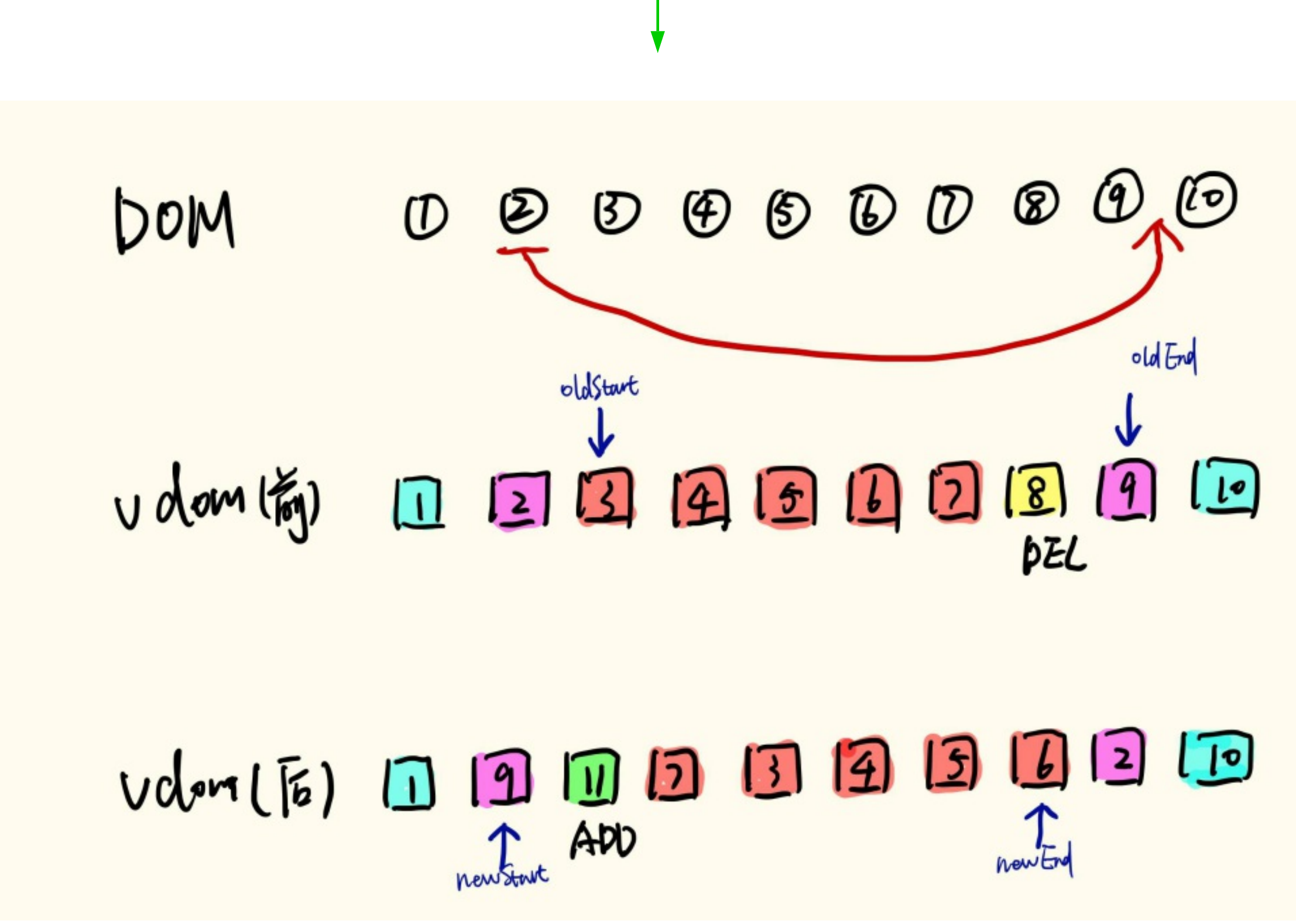
对比顺序: 先头尾看看是否相同 --> old vnode 从左向右 找 - ->new vnode从左向右找



（2）、处理尾部的同类型节点，即oldEnd和newEnd指向同类节点的情况，如上图中的节点10

与情况（1）类似，这种情况下，将节点10的变更更新到DOM，

然后oldEnd和newEnd前移1位进行标记，同样也不需要移动DOM



（3）、处理头尾/尾头的同类型节点，即oldStart和newEnd，以及oldEnd和newStart指向同类节点的情况，

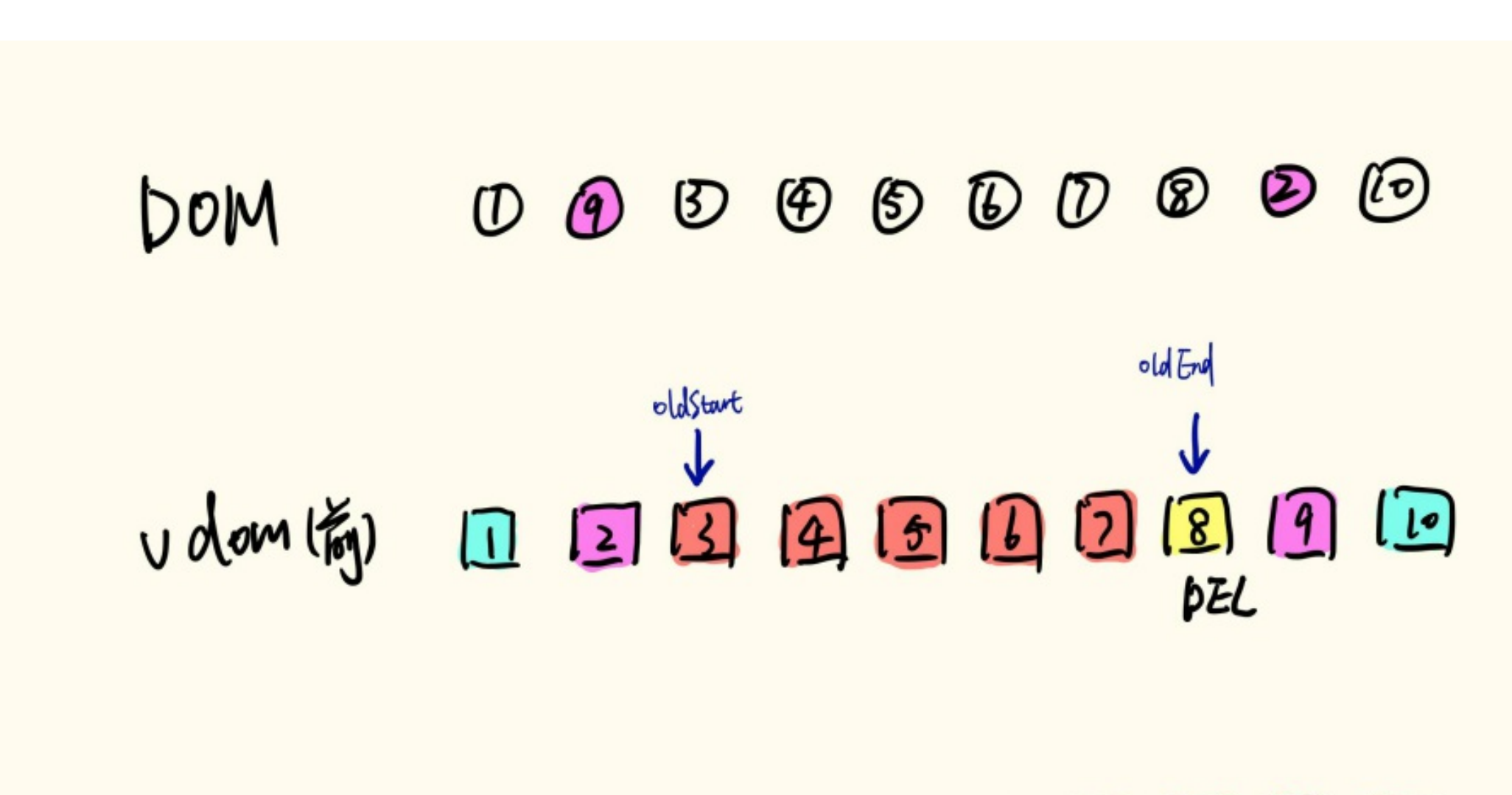
如图中的节点2和节点9

先看节点2，其实是往后移了，移到哪里？

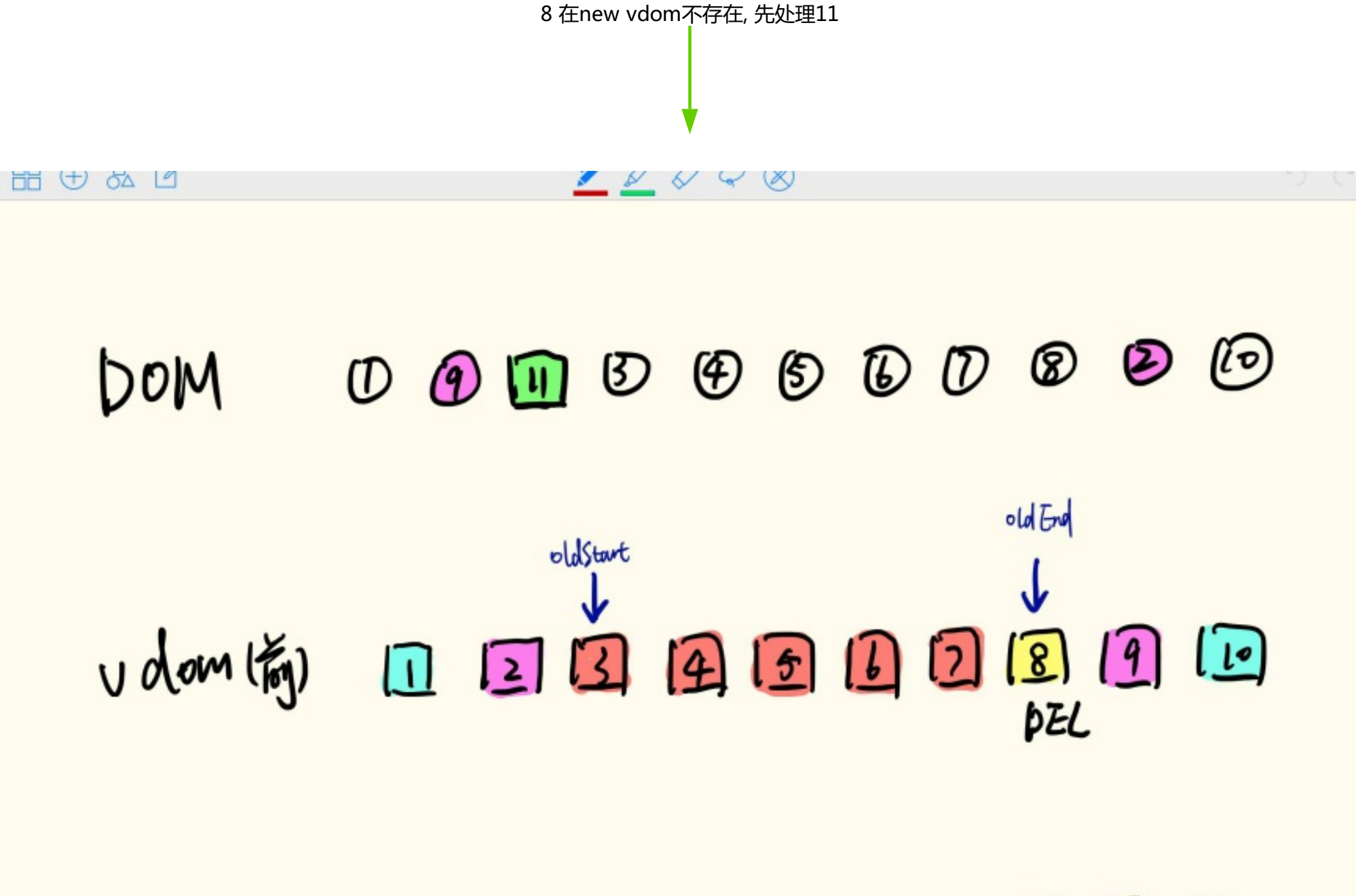
移到oldEnd指向的节点（即节点9）后面，移动之后标记该节点

，将oldStart后移1位，newEnd前移一位

同样地，节点9也是类似的处理，处理完之后成了下面这样



8 在new vdom不存在, 先处理11



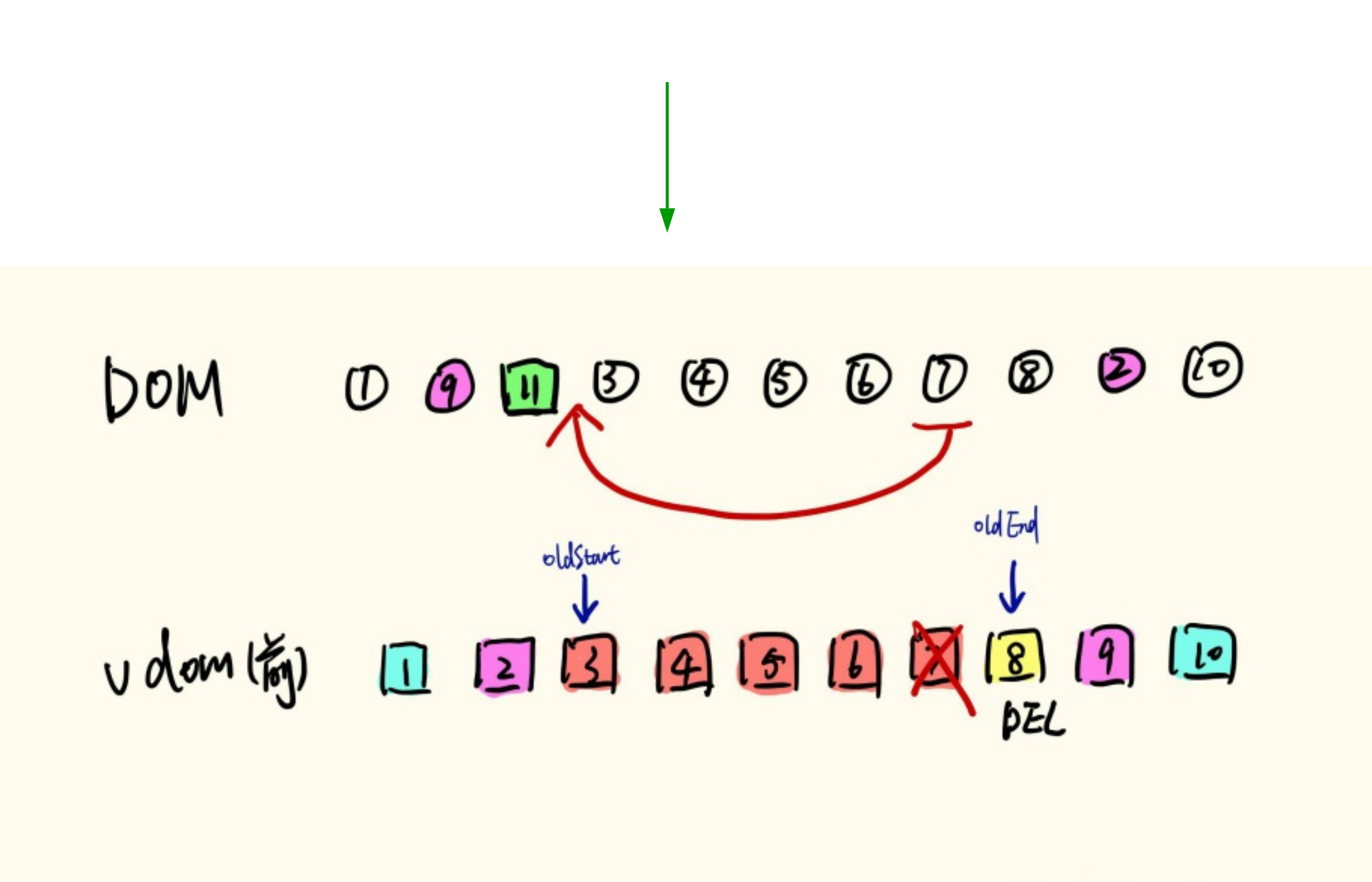
处理新增的节点

newStart来到了节点11的位置，在oldVdom中找不到节点11，说明它是新增的

那么就创建一个新的节点，插入DOM树，插到什么位置？

插到oldStart指向的节点（即节点3）前面，然后将newStart后移1位标记为已处理

（注意oldVdom中没有节点11，所以标记过程中它的指针不需要移动）



5）、处理更新的节点

经过第（4）步之后，newStart来到了节点7的位置，

在oldVdom中能找到它而且不在指针位置 (oleEnd 还指向8)

（查找oldVdom中oldStart到oldEnd区间内的节点），说明它的位置变了, 往前了一位

那么需要在DOM树中移动它，移到哪里？

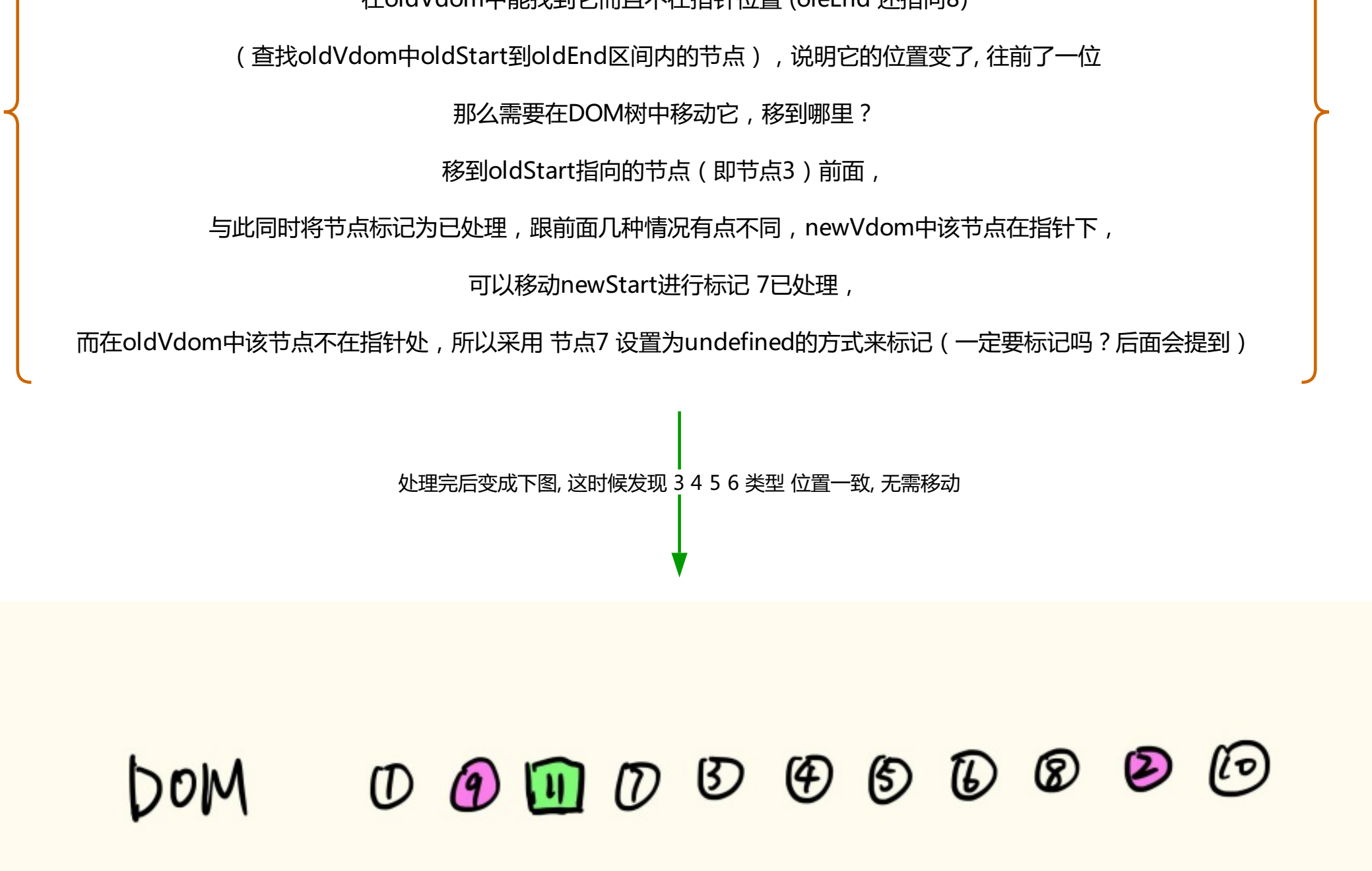
移到oldStart指向的节点（即节点3）前面，

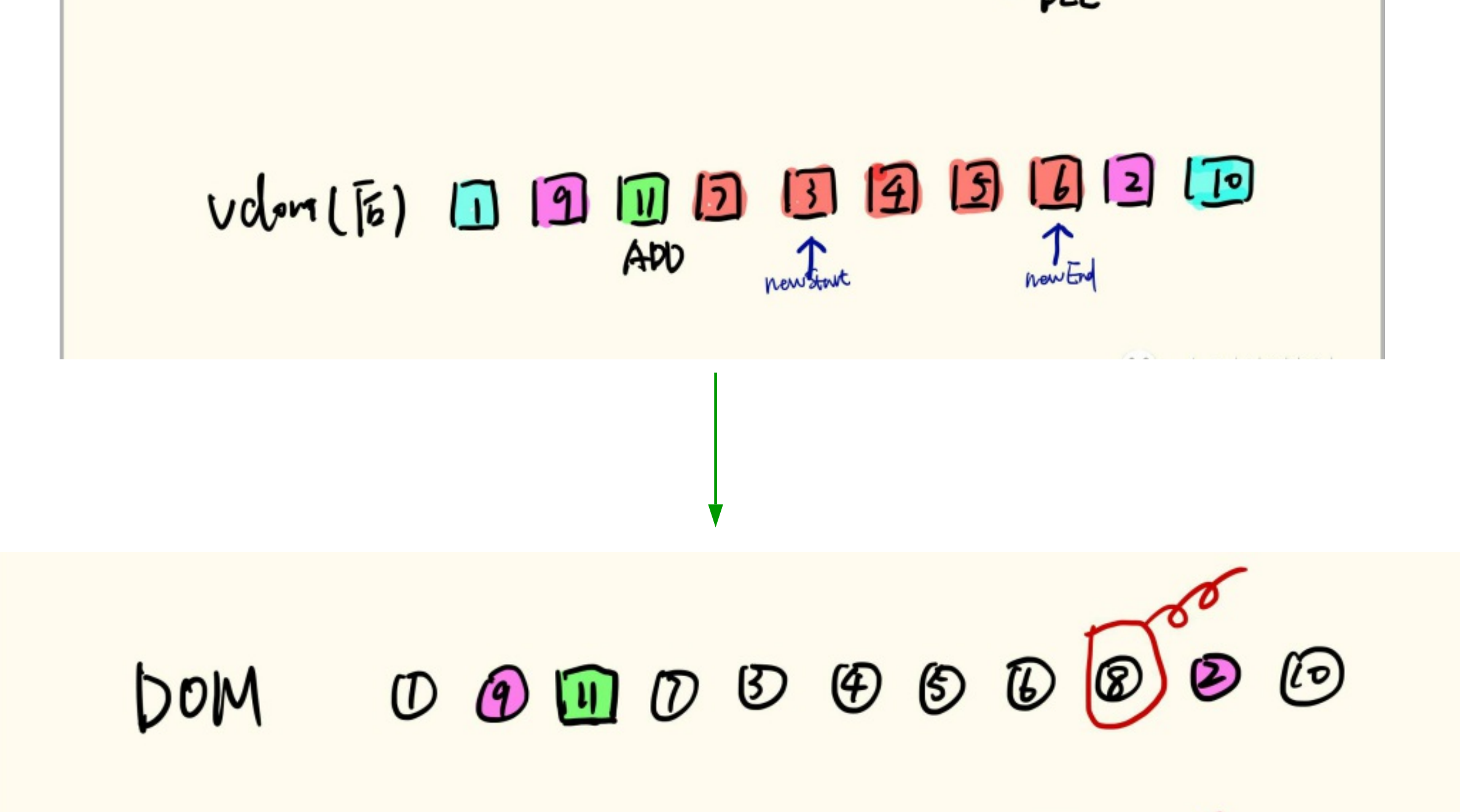
与此同时将节点标记为已处理，跟前面几种情况有点不同，newVdom中该节点在指针下，

可以移动newStart进行标记 7已处理，

而在oldVdom中该节点不在指针处，所以采用 节点7 设置为undefined的方式来标记（一定要标记吗？后面会提到）

处理完后变成下图, 这时候发现 3 4 5 6 类型 位置一致, 无需移动





7）、处理需删除的节点

经过前6步处理之后（实际上前6步是循环进行的），朋友们看newStart跨过了newEnd

(图中没有表示出来, 其实6比较后, newEnd应该移动到2的位置, 所以 和newStart重合了)，它们相遇啦！

而这个时候，oldStart和oldEnd还没有相遇，

说明这2个指针之间的节点（包括它们指向的节点，即上图中的节点7、节点8）是此次更新中被删掉的节点。

(应该只有 8 才是 被删除的, 下面会解释)

OK，那我们在DOM树中将它们删除，

再回到前面我们对节点7做了标记，

为什么标记是必需的？标记的目的是告诉Vue它已经处理过了，是需要出现在新DOM中的节点，

不要删除它，所以在这里只需删除节点8。

在应用中也可能会遇到oldVdom的起止点相遇了，但是newVdom的起止点没有相遇的情况，

这个时候需要对newVdom中的未处理节点进行处理，

这类节点属于更新中被加入的节点，需要将他们插入到DOM树中。

至此，整个diff过程结束了