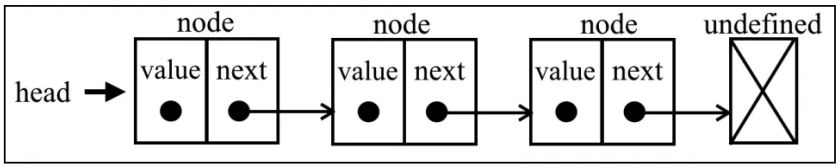
链表

链表数据结构

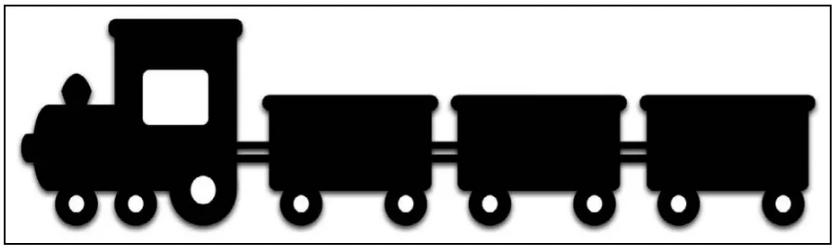
要存储多个元素，数组可能是最常用的数据结构。每种语言都实现了数组。这种数据结构非常方便，提供了一个便利的[]语法来访问其元素。然而，这种数据结构有一个缺点：（在大多数语言中）数组的大小是固定的，从数组的起点或中间插入或移除项的成本很高，因为需要移动元素。

链表存储有序的元素集合，但不同于数组，链表中的元素在内存中并不是连续放置的。每个元素由一个存储元素本身的节点和一个指向下一个元素的引用（也称指针或链接）组成。下图展示了一个链表的结构。



相对于传统的数组，链表的一个好处在于，添加或移除元素的时候不需要移动其他元素。然而，链表需要使用指针，因此实现链表时需要额外注意。在数组中，我们可以直接访问任何位置的任何元素，而要想访问链表中间的一个元素，则需要从起点（表头）开始迭代链表直到找到所需的元素。

现实中也有一些链表的例子。那就是火车。一列火车是由一系列车厢（也 称车皮）组成的。每节车厢或车皮都相互连接。你很容易分离一节车皮，改变它的位置、添加或 移除它。下图演示了一列火车。每节车皮都是链表的元素，车皮间的连接就是指针。



创建链表

理解了链表(LinkedList)是什么之后，现在就要开始实现我们的数据结构了。以下是实现LinkedList的

“骨架”。

```JavaScript
(function (window) {

  var prototype = {};

  function createLinkedList() {

    var res = {
      count: 0,     // {1}
      head: null,  // {2}
    };

    res.__proto__ = prototype;

    return res;
  }

  window.createLinkedList = createLinkedList;

})(window);
```

对于 LinkedList 数据结构，我们从声明 count 属性开始（行{1}），它用来存储链表中的 元素数量。

由于该数据结构是动态的，我们还需要将第一个元素的引用保存下来。我们可以用一个叫作 head 的元素保存引用（行{2}）。

要表示链表中的第一个以及其他元素，我们需要一个工厂函数，叫作 createNode。其返回值表示我们想要添加到链表中的元素。它包含一个 element 属性，该属性表示要加入链表元素的值；以及一个 next 属性，该属性是指向链表中下一个元素的指针。它的代码如下所示：

```JavaScript
function createNode(element) {
    return {
      element: element,
      next: null,
    };
  }
```

然后就是链表的方法。在实现这些方法之前，我们先来看看它们的职责。

- push(element)：向链表尾部添加一个新元素。
- insert(element, position)：向链表的特定位置插入一个新元素。
- getElementAt(index)：返回链表中特定位置的元素。不存在则返回 undefined。
- remove(element)：从链表中移除一个元素。
- indexOf(element)：返回元素在链表中的索引。如果链表中没有该元素则返回-1。
- removeAt(position)：从链表的特定位置移除一个元素。
- isEmpty()：如果链表中不包含任何元素，返回 true，如果链表长度大于 0则返回 false。
- size()：返回链表包含的元素个数，与数组的 length 属性类似。
- toString()：返回表示整个链表的字符串。

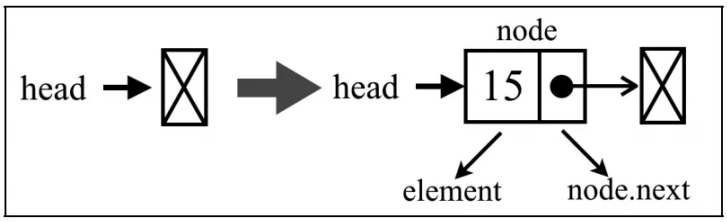
向链表尾部添加元素

LinkedList 对象尾部添加一个元素时，可能有两种场景：链表为空，添加的是第一个元素；链表不为空，向其追加元素。下面是我们实现的 push 方法：

```javascript
push(element) {
    var node = createNode(element); // {1}
    var current; // {2}
    if (this.head == null) { // {3}
      this.head = node;
    } else {
      current = this.head; // {4}
      while (current.next != null) { // {5} 获得最后一项
      current = current.next;
    }
      // 将其 next 赋为新元素，建立链接
    current.next = node; // {6}
    }
    this.count++; // {7}
  }
```

首先需要做的是把 element 作为值传入，创建 Node 项（行{1}）。

先来实现第一个场景：向空列表添加一个元素。当我们创建一个 LinkedList 对象时，head 会指向 undefined（或者是 null）。



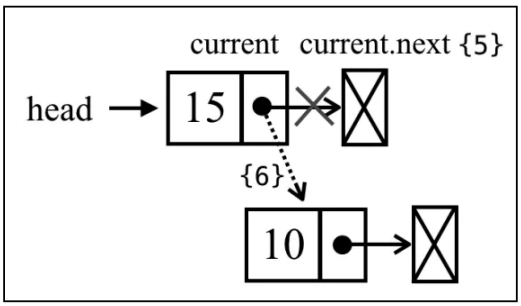
如果 head 元素为 undefined 或 null（列表为空——行{3}），就意味着在向链表添加第一个元素。因此要做的就是让 head 元素指向 node 元素。下一个 node 元素会自动成为 undefined。

> 链表最后一个节点的下一个元素始终是 undefined 或 null。

第二种场景，也就是向一个不为空的链表尾部添加元素。

要向链表的尾部添加一个元素，首先需要找到最后一个元素。记住，我们只有第一个元素的引用（行{4}），因此需要循环访问列表，直到找到最后一项。为此，我们需要一个指向链表中current项的变量（行{2}）。 在循环访问链表的过程中，当 current.next 元素为 undefined 或 null 时，我们就知道 已经到达链表尾部了（行{5}）。然后要做的就是让当前（也就是最后一个）元素的 next 指针指 向想要添加到链表的节点（行{6}）。

下图展示了向非空链表的尾部添加一个元素的过程:



当一个 Node 实例被创建时，它的 next 指针总是 undefined。这没问题，因为我们知道它 会是链表的最后一项。 当然，别忘了递增链表的长度，这样就能控制它并且轻松得到链表的长度（行{7}）。 我们可以通过以下代码来使用和测试目前创建的数据结构。

```JavaScript
var list = createLinkedList();
list.push("第一个元素");
list.push("第二个元素");

```

从链表中移除元素

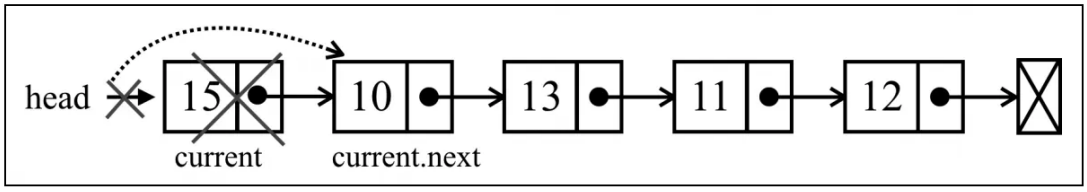
我们要实现两种 remove 方法： 第一种是从特定位置移除一个元素（removeAt），第二种是根据元素的值移除元素（稍后我们会 展示第二种 remove 方法）。和 push 方法一样，对于从链表中移除元素也存在两种场景：第一种是移除第一个元素，第二种是移除第一个元素之外的其他元素。

removeAt 方法的代码如下所示:

```JavaScript
function removeAt(index) {
  // 检查越界值
  if (index >= 0 && index < this.count) { // {1}
    let current = this.head; // {2}
    // 移除第一项
    if (index === 0) { // {3}
      this.head = current.next;
    } else {
      let previous; // {4}
      for (let i = 0; i < index; i++) { // {5}
        previous = current; // {6}
        current = current.next; // {7}
      }
      // 将 previous 与 current 的下一项链接起来：跳过 current，从而移除它
      previous.next = current.next; // {8}
    }
    this.count--; // {9}
    return current.element;
  }
  return undefined; // {10}
}
```

由于该方法要得到需要移除的元素的 index（位置），我们需要验 证该 index 是有效的（行{1}）。从 0（包括 0）到链表的长度（count 一 1，因为 index 是从零开始的）都是有效的位置。如果不是有效的位置，就返回 undefined（行{10}，即没有从列 表中移除元素）。

第一种场景：我们要从链表中移除第一个元素（position === 0——行 {3}）。下图展示了这个过程。

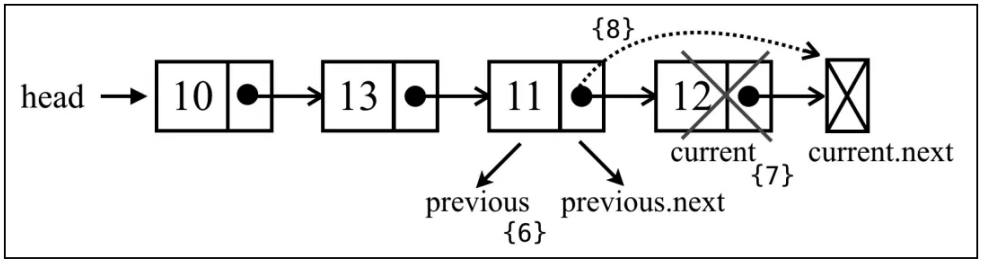


如果想移除第一个元素，要做的就是让 head 指向列表的第二个元素。我们将用current 变量创建一个对链表中第一个元素的引用（行{2}——我们还会用它来迭代链表）。这样 current 变量就是对链表中第一个元素的引用。如果把 head 赋为current.next，就会移除第一个元素。我们也可以直接把 head 赋为 head.next（不使用current 变量作为替代）。

假设我们要移除链表的最后一个或者中间某个元素。为此，需要迭代链表的节点，直到到达目标位置（行{5}）。一个重要细节是：current 变量总是为对所循环列表的当前元素的引用（行{7}）。我们还需要一个对当前元素的前一个元素的引用（行{6}），它被命名为 previous（行{4}）。

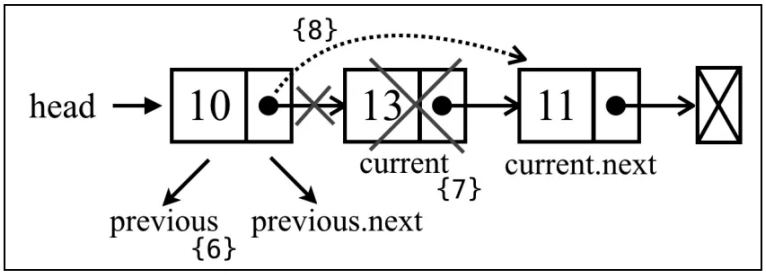
在迭代到目标位置之后，current 变量会持有我们想从链表中移除的节点。因此，要从链表中移除当前元素，要做的就是将 previous.next 和 current.next 链接起来（行{8}）。这样，当前节点就会被丢弃在计算机内存中，等着被垃圾回收器清除。

移除最后一个元素的情况：



对于最后一个元素，当我们在行{8}跳出循环时，current 变量将是对链表中最后一个节点的引用（要移除的节点）。current.next 的值将是 undefined（因为它是最后一个节点）。由于还保留了对 previous 节点的引用（当前节点的前一个节点），previous.next 就指向了current。那么要移除 current，要做的就是把 previous.next 的值改变为 current.next。

移除链表中间的元素的情况：



current 变量是对要移除节点的引用。previous 变量是对要移除节点的前一个节点的引用。那么要移除 current 节点，需要做的就是将 previous.next 与 current.next 链接起来。 因此，我们的逻辑对这两种情况都适用。

循环迭代链表直到目标位置

在 remove 方法中，我们需要迭代整个链表直到到达我们的目标索引 index（位置）。循环到目标 index 的代码片段在 LinkdedList 类的方法中很常见。因此，我们可以重构代码，将这部分逻辑独立为单独的方法，这样就可以在不同的地方复用它。那么，我们就来创建getElementAt 方法。

```JavaScript
function getElementAt(index) {
  if (index >= 0 && index <= this.count) { // {1}
    let node = this.head; // {2}
    for (let i = 0; i < index && node != null; i++) { // {3}
      node = node.next;
    }
    return node; // {4}
  }
  return undefined; // {5}
}
```

为了确保我们能迭代链表直到找到一个合法的位置，需要对传入的 index 参数进行合法性验证（行{1}）。如果传入的位置是不合法的参数，我们返回 undefined，因为这个位置在链表中并不存在（行{5}）。然后，我们要初始化 node 变量，该变量会从链表的第一个元素 head（行{2}）

开始，迭代整个链表。如果你想和 LinkdedList 类中的其他方法保持相同的模式，也可以将 node 变量重命名为 current。

然后，我们会迭代整个链表直到目标 index（行{3}）。结束循环时，node 元素（行{4}）将是 index 位置元素的引用。你也可以在 for 循环中使用 i = 1; i <= index 来获得相同的结果。

重构 remove 方法

我们可以使用刚创建的 getElementAt 方法来重构 remove 方法。将行{4}～行{8}替换为以下代码。

```JavaScript
if (index === 0) {
 // 第一个位置的逻辑
} else {
 const previous = this.getElementAt(index - 1);
 current = previous.next;
 previous.next = current.next;
}
this.count--; // {9}
```

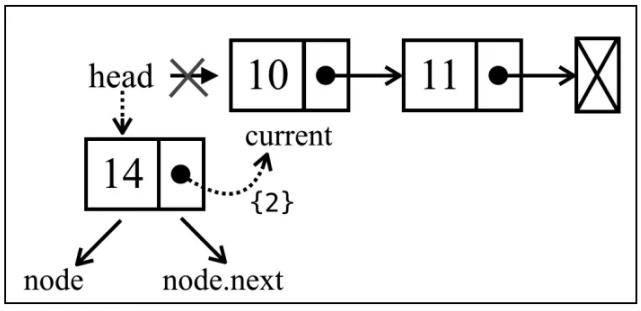
在任意位置插入元素

实现 insert 方法，使用该方法可以在任意位置插入一个元素。

```JavaScript
function insert(element, index) {
  if (index >= 0 && index <= this.count) { // {1}
    const node = new Node(element);
    if (index === 0) {
      // 在第一个位置添加
      const current = this.head;
      node.next = current; // {2}
      this.head = node;
    } else {
      const previous = this.getElementAt(index - 1); // {3}
      const current = previous.next; // {4}
      node.next = current; // {5}
      previous.next = node; // {6}
    }
    this.count++; // 更新链表的长度
    return true;
  }
  return false; // {7}
}
```

由于我们处理的是位置（索引），就需要检查越界值（行{1}，跟 remove 方法类似）。如果越界了，就返回 false 值，表示没有添加元素到链表中（行{7}）。

如果位置合法，我们就要处理不同的场景。第一种场景是需要在链表的起点添加一个元素，也就是第一个位置，如下图所示。



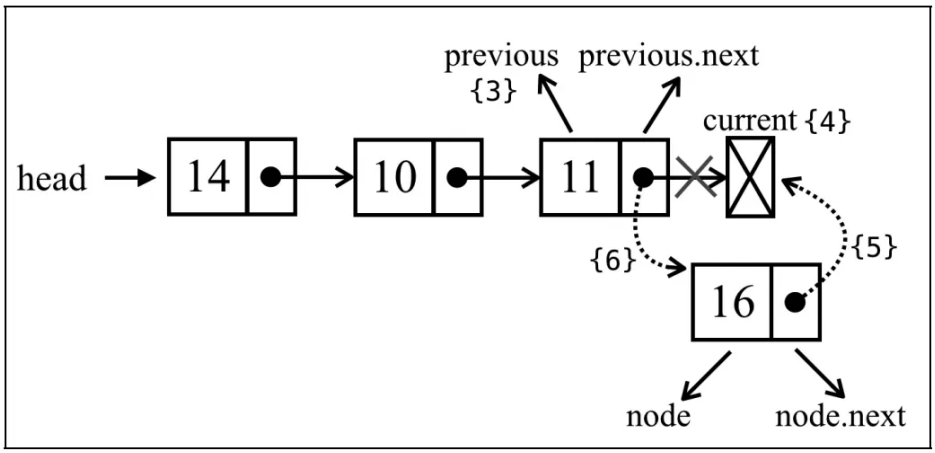
在上图中，current 变量是对链表中第一个元素的引用。我们需要做的是把 node.next 的值设为 current（链表中第一个元素，或简单地设为 head）。现在 head 和 node.next 都指向了

current。接下来要做的就是把 head 的引用改为 node（行{2}），这样链表中就有了一个新元素。

现在来处理第二种场景：在链表中间或尾部添加一个元素。首先，我们需要迭代链表，找到目标位置（行{3}）。这个时候，我们会循环至 index – 1 的位置，表示需要添加新节点位置的前一个位置。

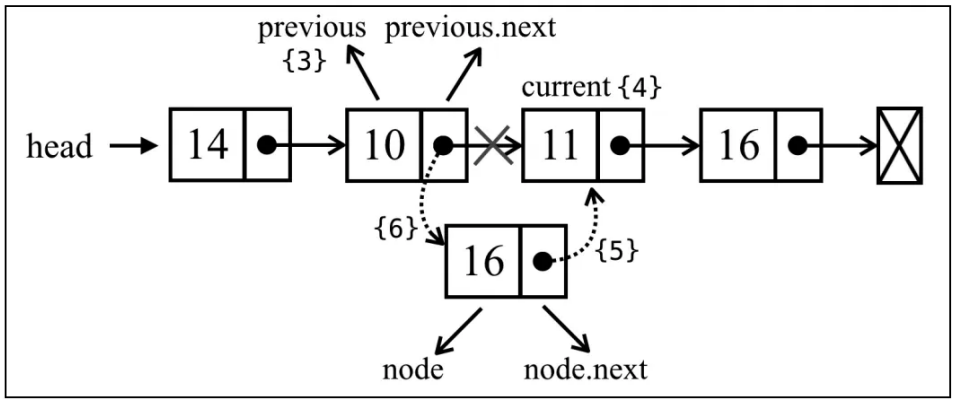
当跳出循环时，previous 将是对想要插入新元素的位置之前一个元素的引用，current变量（行{4}）将是我们想要插入新元素的位置之后一个元素的引用。在这种情况下，我们要在previous 和 current 之间添加新元素。因此，首先需要把新元素（node）和当前元素链接起来（行{5}），然后需要改变 previous 和 current 之间的链接。我们还需要让 previous.next指向 node（行{6}），取代 current。

我们通过一张图表来看看代码所做的事：



如果试图向最后一个位置添加一个新元素，previous 将是对链表最后一个元素的引用，而current 将是 undefined。在这种情况下，node.next 将指向 current，而 previous.next将指向 node，这样链表中就有了一个新元素。

现在来看看如何向链表中间添加一个新元素。



在这种情况下，我们试图将新元素（node）插入 previous 和 current 元素之间。首先，我们需要把 node.next 的值指向 current，然后把 previous.next 的值设为 node。这样列表中就有了一个新元素。

使用变量引用我们需要控制的节点非常重要，这样就不会丢失节点之间的链接。 我们可以只使用一个变量（previous），但那样会很难控制节点之间的链接。 因此，最好声明一个额外的变量来帮助我们处理这些引用。