链表

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。 相比于线性表顺序结构，操作复杂。由于不必须按顺序存储，链表在插入的时候可以达到O(1)的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要O(n)的时间，而线性表和顺序表相应的时间复杂度分别是O(logn)和O(1)。

使用链表结构可以克服数组链表需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。链表最明显的好处就是，常规数组排列关联项目的方式可能不同于这些数据项目在记忆体或磁盘上顺序，数据的存取往往要在不同的排列顺序中转换。链表允许插入和移除表上任意位置上的节点，但是不允许随机存取。链表有很多种不同的类型：单向链表，双向链表以及循环链表。链表可以在多种编程语言中实现。像Lisp和Scheme这样的语言的内建数据类型中就包含了链表的存取和操作。程序语言或面向对象语言，如C,C++和Java依靠易变工具来生成链表。

实现一个简单的链表

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}
class LinkList {
  constructor() {
    this.length = 0;
     this.head = null;
     this.tail = null;
   }
   // 添加节点
   append(data) {
     const new_node = new Node(data);
     if (this.isEmpty()) {
       this.head = new_node;
     } else {
       this.tail.next = new_node;
     }
     this.tail = new_node;
     this.length += 1;
     return true;
   }
   //插入节点
   insert(data, pos) {
     const length = this.length;
     if (pos === length) {
       return  this.append(data);;
     }
     const new_node = new Node(data);
```

```
    let cur_node = this.head;
    let index = 0;
    while (index < pos - 1) {
      cur_node = cur_node.next;
      index += 1;
    }
    const next_node = cur_node.next;
    cur_node.next = new_node;
    new_node.next = next_node;
    if (pos === 0) {
      this.head = new_node;
    }
    this.length += 1;
    return true;
  }
  // 判空
  isEmpty() {
    return this.length === 0;
  }
  // 大小
  size() {
    return this.length;
  }
  // 打印
  print() {
    let cur_node = this.head;
    while (cur_node) {
      console.log(cur_node);
      cur_node = cur_node.next;
    }
  }
  // 判断是否存在
  indexOf(data) {
    let cur_node = this.head;
    while (cur_node) {
      if (cur_node.data === data) {
        return cur_node;
      }
      cur_node = cur_node.next;
    }
    return null;
  }
  // 删除
  remove(pos) {
    let cur_node = this.head;
    if (pos === 0) {
      this.head = cur_node.next;
```

```
      if(!this.head) {
          this.tail = null;
      }
      this.length -= 1;
      return cur_node;
    }
    let index = 0;
    while (index < pos - 1) {
      cur_node = cur_node.next;
      index += 1;
    }
    const remove_node = cur_node.next;
    cur_node.next = remove_node.next;
    // console.log(cur_node);
    if (index === this.length) {
      this.tail = cur_node;
    }
    this.length -= 1;
    return remove_node;
  }
}
```

算法题

1.合并两个有序链表

```

var Node = function (data) {
  this.data = data;
  this.next = null;
}

var node1 = new Node(1);
var node2 = new Node(4);
var node3 = new Node(9);
var node4 = new Node(2);
var node5 = new Node(5);
var node6 = new Node(6);
var node7 = new Node(10);


node1.next = node2;
node2.next = node3;

node4.next = node5;
node5.next = node6;
node6.next = node7;

function merge_link(head1, head2) {
```

```
  //在这里实现你的代码
  var merge_head = null;
  var merge_tail = null;
  var cur_node1 = head1;
  var cur_node2 = head2;
  while (cur_node1 && cur_node2) {
    var min_node;
    if (cur_node1.data > cur_node2.data) {
      min_node = cur_node2;
      cur_node2 = cur_node2.next;
    } else {
      min_node = cur_node1;
      cur_node1 = cur_node1.next;
    }
    if (merge_head) {
      const node = new Node(min_node.data);
      merge_tail.next = node;
      merge_tail = node;
    } else {
      merge_head = new Node(min_node.data);
      merge_tail = merge_head;
    }
  }
  let res_node = cur_node1 || cur_node2;
  while (res_node) {
    merge_tail.next = res_node;
    res_node = res_node.next;
  }

  return merge_head;
};

print(merge_link(node1, node4));

function print(node) {
  var curr_node = node;
  while (curr_node) {
    console.log(curr_node.data);
    curr_node = curr_node.next;
  }
};
```

2. 查找单链表中的倒数第K个节点（k > 0）

```
// 实现函数reverse_find,  返回链表倒数第k个节点的数值
var Node = function (data) {
  this.data = data;
  this.next = null;
```

```
}

var node1 = new Node(1);
var node2 = new Node(2);
var node3 = new Node(3);
var node4 = new Node(4);
var node5 = new Node(5);


node1.next = node2;
node2.next = node3;
node3.next = node4;
node4.next = node5;


function reverse_find(head, k) {
  let cur_node = head;
  let res_node = head;
  let s_index = 0;
  while (cur_node) {
    s_index += 1;
    cur_node = cur_node.next;
    if (s_index > k) {
      res_node = res_node.next;
    }
  }
  return res_node;
};

```

3 查找单链表的中间结点（普通模式）

```

// 实现函数find_middle,  查找并返回链表的中间节点
var Node = function (data) {
  this.data = data;
  this.next = null;
};
var node1 = new Node(1);
var node2 = new Node(2);
var node3 = new Node(3);
var node4 = new Node(4);
var node5 = new Node(5);
node1.next = node2;
node2.next = node3;
node3.next = node4;
node4.next = node5;
function find_middle(head) {
```

```
  // 在这里实现你的代码,返回倒数第k个节点的值
  var fast = head;
  var slow = head;
  while (fast && fast.next) {
    fast = fast.next.next;
    slow = slow.next;
  }
  return slow.data;
};
console.log(find_middle(node1));
```