1. 假设你的同事写了一个资源节点类KNode，它里面可能存着单个int（表示资源ID），也可能存着一个KNode列表，表示一组资源节点：

class KNode {

  value;

  constructor(v) {

    this.value = v;

  }

  getInt() {} // return number

  isInt() {} // return boolean

  getList() {} // return [KNode]: KNode对象的数组

}

现需要实现一个迭代器KNodeIterator，包含如下接口:

class KNodeIterator {

  kNodes;

  constructor(v) {

    this.kNodes = v; // 要求O(1)

  }

  hasNext() {} // return boolean 要求O(1), 最坏情况下O(N)

  next() {} // return number 要求O(1)

}

为简化这个问题，你仅需要正确支持如下（调用）用例：

// const kNodes = [

//   new KNode([new KNode(5), new KNode(4)]),

//   new KNode(3),

//   new KNode([new KNode(2), new KNode(1)])

// ];

const kNodes = [

  new KNode([new KNode(5), new KNode(4)]),

  new KNode(3),

  new KNode([new KNode(2), new KNode(1)]),

  new KNode([])

];

const iter = new KNodeIterator(kNodes);

while (iter.hasNext) {

  const resourceId = iter.next();

  console.log('resourceId - ', resourceId);

}

**输入:** [[5,4],3,[2,1],[]]

**输出:** [5,4,3,2,1]

**解释: 输入一个KNode数组，**通过重复调用 next 直到 hasNext 返回false，next 返回的元素的顺序应该是: [5,4,3,2,1]。

**输入:** [1,[3,[7]],8]

**输出:** [1,3,7,8]

**解释: 输入一个KNode数组，**通过重复调用 next 直到 hasNext 返回false，next 返回的元素的顺序应该是: [1,3,7,8]。

// const kNodes = [

//   new KNode([new KNode(5), new KNode(4)]),

//   new KNode(3),

//   new KNode([new KNode(2), new KNode(1)])

// ];

class  KNode  {

    value;

    constructor(v)  {     this.value  =  v;   }

    getInt()  {

            return this.isInt() ? this.value : false;

        }  // return number

    isInt()  {

            return typeof this.value === 'number' && !isNaN(this.value);

        }  // return boolean

    getList()  {

            if (Array.isArray(this.value)) {

                return this.value;

            }

            return false;

        }  // return [KNode]: KNode对象的数组

}

class  KNodeIterator  {

    //当前列表

    kNodes;

    //遍历至当前列表所在的位置

    step = -1;

    //列表栈

    list\_stack = [];

    //指针栈

    step\_stack = [];

    constructor(v)  {

        this.kNodes  =  v;  // 要求O(1)

        // this.list = v;

    }

    hasNext()  {

            let tmp\_step = this.step + 1;

            let tmp\_list = this.kNodes;

            let tmp\_list\_stack = this.list\_stack;

            let tmp\_step\_stack = this.step\_stack;

            //循环遍历至返回结果

            //大体思路与next()相同

            while (true) {

                // console.log(tmp\_list);

                // console.log(tmp\_step);

                //当遍历当前列表至结尾时

                if (tmp\_step >= tmp\_list.length) {

                    //判断栈是否为空，若空则遍历已经结束，则没有下一个元素

                    if (tmp\_step\_stack.length == 0) {

                        return false;

                    };

                    //非空则恢复上一次的现场，继续进行遍历

                    tmp\_list = tmp\_list\_stack.pop();

                    tmp\_step = tmp\_step\_stack.pop() + 1;

                    continue;

                    //当当前节点为数字节点时，返回

                } else if (!Array.isArray(this.kNodes[this.step]) && tmp\_list[tmp\_step].isInt()) {

                    return true;

                    //当前节点为列表且此列表为空列表、直接跳过

                } else if (tmp\_list[tmp\_step].length == 0) {

                    tmp\_step++;

                    continue;

                    //当前节点为列表且非空、当前场景入栈并遍历当前节点

                } else {

                    tmp\_list\_stack.push(tmp\_list);

                    tmp\_step\_stack.push(tmp\_step);

                    tmp\_list = tmp\_list[tmp\_step].getList();

                    tmp\_step = 0;

                    continue;

                }

            }

        }  // return boolean 要求O(1), 最坏情况下O(N)

    //通过栈实现深度优先遍历、每一步操作为O(1)

    next()  {

            this.step++;

            //当遍历完当前列表后，从栈中寻找上一次遍历的现场并恢复

            if (this.step >= this.kNodes.length) {

                this.kNodes = this.list\_stack.pop();

                this.step = this.step\_stack.pop();

                return this.next();

                //当当前元素为int时，返回值

            } else if (!Array.isArray(this.kNodes[this.step]) && this.kNodes[this.step].isInt()) {

                return this.kNodes[this.step].getInt();

                //或者当前元素为array时，将当前现场入栈，深入进行遍历

            } else {

                this.list\_stack.push(this.kNodes);

                this.step\_stack.push(this.step);

                this.kNodes = this.kNodes[this.step].getList();

                this.step = -1;

                return this.next();

            }

        }  // return number 要求O(1)

}

console.log("案例1--------------------------");

const  kNodes  =   [

    new  KNode([new  KNode(5),  new  KNode(4)]),

    new  KNode(3),

    new  KNode([new  KNode(2),  new  KNode(1)]),

    new  KNode([])

];

const  iter  =  new  KNodeIterator(kNodes);

while  (iter.hasNext())  {

    const  resourceId  =  iter.next();

    console.log('resourceId - ',  resourceId);

}

console.log("案例2---------------------------");

const kNodes\_1 = [

    new KNode([new KNode(1),

        new KNode([new KNode(3),

            new KNode([new KNode(7)])

        ]),

        new KNode(8)

    ])

]

const iter\_1 = new KNodeIterator(kNodes\_1);

while (iter\_1.hasNext()) {

    const  resourceId  =  iter\_1.next();

    console.log('resourceId - ',  resourceId);

}