入选理由

1. 跳表就一道题，非他莫属了^_^。

> 作为了解即可，很少有手写跳表的。

标签

- 跳表

难度

- 困难

题目地址（1206. 设计跳表）

https://leetcode-cn.com/problems/design-skiplist/

题目描述

不使用任何库函数，设计一个跳表。

跳表是在 O(log(n)) 时间内完成增加、删除、搜索操作的数据结构。跳表相比于树堆与红黑树，其功能与性能相当，并且跳表的代码长度相较下更短，其设计思想与链表相似。

例如，一个跳表包含 [30, 40, 50, 60, 70, 90]，然后增加 80、45 到跳表中，以下图的方式操作：

Artyom Kalinin [CC BY-SA 3.0], via Wikimedia Commons

跳表中有很多层，每一层是一个短的链表。在第一层的作用下，增加、删除和搜索操作的时间复杂度不超过 O(n)。跳表的每一个操作的平均时间复杂度是 O(log(n))，空间复杂度是 O(n)。

在本题中，你的设计应该要包含这些函数：

bool search(int target) : 返回 target 是否存在于跳表中。 void add(int num): 插入一个元素到跳表。 bool erase(int num): 在跳表中删除一个值，如果 num 不存在，直接返回 false. 如果存在多个 num ，删除其中任意一个即可。 了解更多 : https://en.wikipedia.org/wiki/Skip_list

注意，跳表中可能存在多个相同的值，你的代码需要处理这种情况。

样例:

```
Skiplist skiplist = new Skiplist();

skiplist.add(1);
skiplist.add(2);
skiplist.add(3);
skiplist.search(0);   // 返回 false
skiplist.add(4);
skiplist.search(1);   // 返回 true
skiplist.erase(0);    // 返回 false，0 不在跳表中
skiplist.erase(1);    // 返回 true
skiplist.search(1);   // 返回 false，1 已被擦除
约束条件：

0 <= num, target <= 20000
最多调用 50000 次 search, add, 以及 erase操作。
```

思路

因为是设计题，具体参考讲义， 这里说两个注意点

1. 可以想象调表是一个网状结构，每个节点有两个指针，往右和往下
2. 寻找节点的时候，可以想象从最左上角开始往右搜索，网络每层是有序的
3. 插入时记录每层可能需要插入的位置，从下往上逐个插入，是否插入策略由抛硬币决定
4. 删除时，从上往下删，把每层符合要求的节点从当前层链表删除

代码

代码支持： JS,Python，CPP,Java

JS Code：

```
//  维护一个next指针和down指针
function Node(val, next = null, down = null) {
  this.val = val;
  this.next = next;
  this.down = down;
}

var Skiplist = function () {
  this.head = new Node(null);
};
```

```
/**
 * @param {number} target
 * @return {boolean}
 */
Skiplist.prototype.search = function (target) {
  let head = this.head;
  while (head) {
    // 链表有序，从前往后走
    while (head.next && head.next.val < target) {
      head = head.next;
    }
    if (!head.next || head.next.val > target) {
      // 向下走
      head = head.down;
    } else {
      return true;
    }
  }
  return false;
};

/**
 * @param {number} num
 * @return {void}
 */
Skiplist.prototype.add = function (num) {
  const stack = [];
  let cur = this.head;
  // 用一个栈记录每一层可能会插入的位置
  while (cur) {
    while (cur.next && cur.next.val < num) {
      cur = cur.next;
    }
    stack.push(cur);
    cur = cur.down;
  }

  // 用一个标志位记录是否要插入，最底下一层一定需要插入(对应栈顶元素)
  let isNeedInsert = true;
  let downNode = null;
  while (isNeedInsert && stack.length) {
    let pre = stack.pop();
    // 插入元素，维护 next/down 指针
    pre.next = new Node(num, pre.next, downNode);
    downNode = pre.next;
    // 抛硬币确定下一个元素是否需要被添加
    isNeedInsert = Math.random() < 0.5;
  }

  // 如果人品好，当前所有层都插入了改元素，还需要继续往上插入，则新建一层，表现为新建一层元素
  if (isNeedInsert) {
```

```
    this.head = new Node(null, new Node(num, null, downNode), this.head);
  }
};

/**
 * @param {number} num
 * @return {boolean}
 */
Skiplist.prototype.erase = function (num) {
  let head = this.head;
  let seen = false;
  while (head) {
    // 在当前层往前走
    while (head.next && head.next.val < num) {
      head = head.next;
    }
    // 往下走
    if (!head.next || head.next.val > num) {
      head = head.down;
    } else {
      // 找到了该元素
      seen = true;
      // 从当前链表删除
      head.next = head.next.next;
      // 往下
      head = head.down;
    }
  }
  return seen;
};
```

Python Code:

```
class Skiplist:

    def __init__(self):
        left = [Node(-1) for _ in range(16)]
        right = [Node(20001) for _ in range(16)]
        for i in range(15):
            left[i].right = right[i]
            left[i].down = left[i + 1]
            right[i].down = right[i + 1]
        left[-1].right = right[-1]
        self.root = left[0]

    def search(self, target: int) -> bool:
        cur = self.root
        while cur:
            if target < cur.right.val: # in range
```

```
                cur = cur.down
            elif target > cur.right.val: # next range
                cur = cur.right
            else:
                return True
        return False

    def add(self, num: int) -> None:
        cur = self.root
        stack = []
        while cur:
            if cur.right.val >= num:
                stack.append(cur)
                cur = cur.down
            else:
                cur = cur.right
        pre = None

        while stack:
            cur = stack.pop()
            node = Node(num)
            node.right = cur.right
            cur.right = node
            if pre:
                node.down = pre
            pre = node
            if random.randint(0, 1):
                break

    def erase(self, num: int) -> bool:
        cur = self.root
        removed = False
        while cur:
            if num < cur.right.val:
                cur = cur.down
            elif num > cur.right.val:
                cur = cur.right
            else:
                cur.right = cur.right.right
                removed = True
                cur = cur.down # remove all down
        return removed


class Node:
    def __init__(self, val=0, right=None, down=None):
        self.val = val
        self.right = right
        self.down = down
```

CPP Code:

```
struct node
{
  node* right;
  node* down;
    int val;
  node(int iVal)
  {
      val = iVal;
      right = NULL;
      down = NULL;
  }

};
class Skiplist {
public:
    node* head;
    Skiplist() {
        head = new node(0);
    }

    bool search(int target) {

        bool ret = false;
        node* current = head;

        while(current)
        {
          //  cout << "current->right" << (current->right? current->right->val: -1) << "\n";
            while(current->right && current->right->val < target)
                current = current->right;

            if(current->right == NULL || current->right->val >target)
            {
                current = current->down;
            }
            else
                return true;
        }

        return false;
    }

    void add(int num) {
        node* current = head;
        vector<node*> insertNodeList;
        while(current)
        {
            while(current->right && current->right->val < num)
                current = current->right;
            insertNodeList.push_back(current);
            current = current->down;
        }

        node* downNode = NULL;
```

```
        bool insertUp = true;
        while(insertNodeList.size() && insertUp)
        {
            node* prevNode = insertNodeList.back();
            insertNodeList.pop_back();

            // insert new node
            node* insertNode = new node(num);
            insertNode->right = prevNode->right;
            insertNode->down = downNode;
            prevNode->right = insertNode;

            downNode = insertNode;

            insertUp = rand()%2; // 0 or 1. 50% possible to creat up node. you may use another setting.
        }

        if(insertUp) // when insertNodeList is empty.
        {

            node* insertNode = new node(num);
            insertNode->right = NULL;
            insertNode->down = downNode;

            node* newHeadNode = new node(0);
            newHeadNode->right = insertNode;
            newHeadNode->down = head;

            head = newHeadNode;
        }

        //search(num);
    }

    bool erase(int num) {

        node* current = head;
        bool ret = false;
        while(current)
        {
            while(current->right && current->right->val< num  )
                current= current->right;
            if(current->right ==NULL || current->right->val>num)
            {
                current = current->down;
            }
            else
            {
                ret = true;
             //   delete current->right;
                current->right = current->right->right;
                current = current->down;
            }
        }

        return ret;
```

```
    }
};

/**
 * Your Skiplist object will be instantiated and called as such:
 * Skiplist* obj = new Skiplist();
 * bool param_1 = obj->search(target);
 * obj->add(num);
 * bool param_3 = obj->erase(num);
 */
```

Java Code：

```
class Node{
    int val;
    Node right;
    Node down;

    public Node(int val){
        this.val = val;
    }
}

class Skiplist {
    private Node head;

    public Skiplist() {
        Node[] left = new Node[16];
        Node[] right = new Node[16];
        for(int i = 0; i < 16; i++){
            left[i] = new Node(-1);
            right[i] = new Node(20001);
        }
        for(int i = 0; i < 15; i++){
            left[i].right = right[i];
            left[i].down = left[i + 1];
            right[i].down = right[i + 1];
        }
        left[15].right = right[15];
        head = left[0];
    }

    public boolean search(int target) {
        Node cur = head;
        while(cur != null){
            if(cur.right.val > target){
                cur = cur.down;
            }else if(cur.right.val < target){
```

```
                cur = cur.right;
            }else{
                return true;
            }
        }
        return false;
    }

    public void add(int num) {
        Node cur = head;
        Deque<Node> stack = new LinkedList<>();
        while(cur != null){
            if(cur.right.val >= num){
                stack.push(cur);
                cur = cur.down;
            }else{
                cur = cur.right;
            }
        }
        Node pre = null;
        while(!stack.isEmpty()){
            cur = stack.pop();
            Node node = new Node(num);
            node.right = cur.right;
            cur.right = node;
            if(pre != null) node.down = pre;
            pre = node;

            if(Math.random() < 0.5) break;
        }
    }

    public boolean erase(int num) {
        Node cur = head;
        boolean isRemoved = false;
        while(cur != null){
            if(cur.right.val >= num){
                if(cur.right.val == num){
                    isRemoved = true;
                    cur.right = cur.right.right;
                }
                cur = cur.down;
            }else{
                cur = cur.right;
            }
        }
        return isRemoved;
    }
}
```

复杂度分析

参考讲义

上一页 下一页