数据结构的陨落篇

数据结构是基础，合理使用数据结构可以降低时间复杂度和空间复杂度。

该篇将介绍两类数据结构：

线性数据结构：Queue, Stack and HashMap；

树形数据结构：Heap and Trie。

数据结构的定义：

Data structure is a way to organize data. It provides some methods to handle data stream, e.g. insert, delete, etc.

Part 1 – Queue

Queue是java.util包中的一个接口。当我们要使用queue这种数据结构的时候，我们往往使用的是实现了Queue接口的LinkedList，我们把它当作一个queue来使用。

Queue包含如下操作：

1）offer(e)。O(1)，在队尾添加一个元素；

2）poll()。O(1)，在队首移除一个元素；

3）peek()。O(1)，查看队首第一个元素；

由于Queue先入先出的性质，所以经常在广度优先遍历当中使用Queue。

Part 2 – Stack

Stack是java.util包中的一个类，我们可以直接使用，其父类为Vector -> AbstractList等。但我们要使用stack这种数据结构的时候，我们往往使用的是具有stack所有功能的LinkedList，我们把它当作一个stack来使用。

Stack包含如下操作：

1）push(e)。O(1)，在栈首添加一个元素；

2）pop()。O(1)，在栈首移除一个元素；

3）peek()。O(1)，查看栈首第一个元素；

由于Stack先入后出的性质，所以经常在深度优先遍历当中使用Stack。

Min Stack，提供 2 种方法。第一种，使用一个 min 栈来来保存最小值历史信息；第二种，使用一个 min 值，采用在原栈中保存差值的方法来将空间复杂度降至 O(1)。

```
// Solution 1 - Double Stacks
class MinStack {

    LinkedList<Integer> stack;
    LinkedList<Integer> min;

    public MinStack(){
        stack = new LinkedList<Integer>();
        min = new LinkedList<Integer>();
    }

    public void push(int x) {
        stack.push(x);
        if(min.isEmpty() || x <= min.peek()){
            min.push(x);
        }
    }

    public void pop() {
        if(stack.isEmpty()){
            return;
        } else {
            int value = stack.pop();
            if(value == min.peek()){
                min.pop();
            }
        }
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return min.peek();
    }
}
```

```
// Solution 2 - Cur-Min Gap
class MinStack {
    long min;
    LinkedList<Long> stack;
    public MinStack(){
        min = 0;
        stack = new LinkedList<Long>();
    }

    public void push(int x) {
        if(stack.isEmpty()){
            min = x;
            stack.push(0L);
        } else if(x >= min){
            stack.push(x - min);
        } else {
            stack.push(x - min);
            min = x;
        }
    }

    public void pop() {
        if(stack.isEmpty()){
            return;
        } else {
            long value = stack.pop();
            if(value < 0){
                min = min - value;
            }
        }
    }

    public int top() {
        return (int)(stack.peek() < 0 ? min : stack.peek() + min);
    }

    public int getMin() {
        return (int)min;
    }
}
```

Implement Queue by Stacks，将stack1当作buffer，暂存进入的元素，将stack2当作输出栈，当stack2栈为空又执行pop()或peek()时，将stack1中元素转移到stack2中即可。代码如下：

```
public class Solution {
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public Solution() {
        stack1 = new Stack<Integer>();
        stack2 = new Stack<Integer>();
    }

    public void push(int element) {
        stack1.push(element);
    }

    public int pop() {
        if(stack2.isEmpty()){
            while(!stack1.isEmpty()){
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }

    public int top() {
        if(stack2.isEmpty()){
            while(!stack1.isEmpty()){
                stack2.push(stack1.pop());
            }
        }
        return stack2.peek();
    }
}
```

Largest Rectangle in Histogram，使用一个栈来巧妙解决该问题，是Maximal Rectangle的一个Subroutine。具体操作如下：

Maintain an incremental stack:

(1) if a[i] >= stack top: push a[i] into stack

(2) if a[i] < stack top: keep poping element out from stack util the top of stack is smaller than current.

代码如下：

```
public int largestRectangleArea(int[] height) {
    if(height == null || height.length == 0){
        return 0;
    }
    int maxArea = 0;
    LinkedList<Integer> stack = new LinkedList<Integer>();
    for(int i = 0; i <= height.length; i++){
        int value = (i == height.length ? -1 : height[i]);
        while(!stack.isEmpty() && height[stack.peek()] > value){
            int index = stack.pop();
            int h = height[index];
            int w = stack.isEmpty() ? i : i - stack.peek() - 1;
            maxArea = Math.max(maxArea, w * h);
        }
        stack.push(i);
    }
    return maxArea;
}
```

Max Tree，这是 LintCode 上的一道题，其实这种树叫笛卡树(Cartesian Tree)。直接递归建树的话复杂度最差会退化到 O(n ^ 2)。经典建树方法用到了单调堆栈，我们堆栈里存放的树只有左子树，没有右子树，而且根结点最大。规则如下：

（1）如果新来一个数，比栈顶的树根的数小，则把这个数作为一个单独的结点压入堆栈。

（2）否则，不断从堆栈里弹出树，新弹出的树以旧弹出的树为右子树连接起来，直到目前堆栈顶的树根的数大于新来的数。然后，弹出的那些数已经形成了一个新的树，这个树作为新结点的左子树，并把这个新树压入堆栈。

最后还要按照（2）的方法把所有树弹出来，每个旧弹出的树作为新弹出树的右子树。

这样的堆栈是单调堆栈，越靠近堆栈顶的数越小。

代码如下：

```
public TreeNode maxTree(int[] A) {
    if(A == null || A.length == 0){
        return null;
    }
    LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
    for(int i = 0; i < A.length; i++){
        if(stack.isEmpty() || stack.peek().val >= A[i]){
            stack.push(new TreeNode(A[i]));
        } else {
            TreeNode cur = stack.pop();
            while(!stack.isEmpty() && stack.peek().val < A[i]){
                TreeNode pre = stack.pop();
                pre.right = cur;
                cur = pre;
            }
            TreeNode newNode = new TreeNode(A[i]);
            newNode.left = cur;
            stack.push(newNode);
        }
    }
    TreeNode root = stack.pop();
    while(!stack.isEmpty()){
        stack.peek().right = root;
        root = stack.pop();
    }
    return root;
}
```

Part 3 – HashMap

HashMap是java.util包中的一个类，我们可以直接使用。HashMap提供高速的存，取和查找功能，但其中数据是无序的。要输出数据遵照输入的顺序，需要其子类LinkedHashMap，要其中数据按照自然顺序进行排序的话，需要TreeMap，是用红黑树实现的一种Map。

HashMap包含如下操作：

1）put(K key, V value)。O(1)；

2）remove(Object key)。O(1)；

3）get(Object key)。O(1)；

HashMap的遍历方式有entrySet()，keySet()和values()三种，以entrySet()为主。

Hash Function – typically from string to integer. i.e. md5, Magic Number 33.

Collision – Open Hashing (Linked List) Vs. Closed Hashing(Array).

Closed Hashing一般不支持删除操作。

LRU Cache，这道题的关键是选用正确的数据结构，在此用的结构相当于LinkedHashMap。

LinkedHashMap = DoublyLinkedList + HashMap.

DoublyListNode{key, value, prev, next;}

HashMap<key, DoublyListNode>

Newest node append to tail.

Eldest node remove from head.

head和tail有点类似于Dummy Node的用法，在改变头尾时，可以避免分情况讨论。

```
public class LRUCache {
    private class Node{
        int key;
        int value;
        Node prev;
        Node next;

        Node(int key, int value){
            this.key = key;
            this.value = value;
            this.prev = null;
            this.next = null;
        }
    }

    private int capacity;
    private HashMap<Integer, Node> map;
    private Node head;
    private Node tail;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        map = new HashMap<Integer, Node>();
```

```
        head = new Node(-1, -1);
        tail = new Node(-1, -1);
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        if(!map.containsKey(key)){
            return -1;
        }
        Node current = map.get(key);
        current.prev.next = current.next;
        current.next.prev = current.prev;
        moveToTail(current);
        return map.get(key).value;
    }

    public void set(int key, int value) {
        if(get(key) != -1){
            map.get(key).value = value;
            return;
        }
        if(map.size() == capacity){
            map.remove(head.next.key);
            head.next = head.next.next;
            head.next.prev = head;
        }
        Node newNode = new Node(key, value);
        map.put(key, newNode);
        moveToTail(newNode);
    }

    public void moveToTail(Node current){
        current.prev = tail.prev;
        current.next = tail;
        current.prev.next = current;
        tail.prev = current;
    }
}
```

Longest Consecutive Sequence，重点同样是对数据结构的选择，使用HashSet可以将时间复杂度降低到O(n)。

```
public int longestConsecutive(int[] num) {
    if(num == null || num.length == 0){
        return 0;
    }
    HashSet<Integer> set = new HashSet<Integer>();
    for(int i = 0; i < num.length; i++){
        set.add(num[i]);
    }
    int max = 1;
    while(!set.isEmpty()){
        Iterator<Integer> iter = set.iterator();
        int value = iter.next();
        iter.remove();
        int len = 1;
        int left = value - 1;
        while(set.contains(left)){
            set.remove(left--);
            len++;
        }
        int right = value + 1;
        while(set.contains(right)){
            set.remove(right++);
            len++;
        }
        max = Math.max(max, len);
    }
    return max;
}
```

Part 4 – Heap

Heap是一种概念上的数据结构，我们往往使用PriorityQueue来实现Heap的所有功能。

Heap包含如下操作：

1）offer(E e)。O(logN)；

2）poll()。O(logN)；

3）peek()。O(1)；

Low level data structure – Dynamic Array

Heap{ elems[], size; }

elems[1] – root, also the minimum elem in elems.

i's left child: i * 2, i's right child: i * 2 + 1.

Internal Mathod: sift up, sift down.

Median II, We maintain a max heap and a min heap. At any index i, the max heap stores the elements small or equal than the current median and the min heap stores the elements that are larger than the current median. Because in the problem, we select the median as at the position (n-1)/2, so we should always keep the size of max heap equal or one larger than the min heap. At odd index, we try to rebalance the size of max heap and min heap, while at even index, we try to make the size of max heap one larger than that of the min heap. By doing this, at each position, after inserting A[i] into the heaps, the root value of the max heap is the median.

NOTE: if we select (n-1)/2+1 as median, we can then keep the min heap equal or one larger than the max heap, the root value of the min heap is the median.

```
public int[] medianII(int[] nums) {
    int[] res = new int[nums.length];
    PriorityQueue<Integer> minHeap = new PriorityQueue<>(nums.length, new
Comparator<Integer>(){
        public int compare(Integer i, Integer j){
            return i - j;
        }
    });
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>(nums.length, new
Comparator<Integer>(){
        public int compare(Integer i, Integer j){
            return j - i;
        }
    });
    maxHeap.offer(nums[0]);
    res[0] = nums[0];
```

```
        for(int i = 1; i < nums.length; i++){
            if((i & 1) == 1){       //i is odd index.
                int median = maxHeap.peek();
                if(nums[i] < median){
                    minHeap.offer(maxHeap.poll());
                    maxHeap.offer(nums[i]);
                } else {
                    minHeap.offer(nums[i]);
                }
            } else {                //i is even index.
                int median = maxHeap.peek();
                if(nums[i] < median){
                    maxHeap.offer(nums[i]);
                } else {
                    minHeap.offer(nums[i]);
                    maxHeap.offer(minHeap.poll());
                }
            }
            res[i] = maxHeap.peek();
        }
        return res;
    }
```

Part 5 – Trie

在计算机科学中，trie，又称前缀树或字典樹，是一种有序树，用于保存关联数组，其中的键通常是字符串。与二叉查找树不同，键不是直接保存在节点中，而是由节点在树中的位置决定。一个节点的所有子孙都有相同的前缀，也就是这个节点对应的字符串，而根节点对应空字符串。一般情况下，不是所有的节点都有对应的值，只有叶子节点和部分内部节点所对应的键才有相关的值。

Word Search II, DFS. For every word in the list, check every position of board, if the char at some position matches the first char in the word, then start a DFS at this position.

```
public ArrayList<String> wordSearchII(char[][] board, ArrayList<String> words) {
    ArrayList<String> res = new ArrayList<String>();
    if(board == null || board.length == 0 || board[0].length == 0){
        return res;
    }
    int rowNum = board.length;
    int colNum = board[0].length;
    for (int i = 0; i < words.size(); i++){
        String word = words.get(i);
        if(word.length() == 0){
            continue;
        }
        for (int j = 0; j < rowNum; j++){
            boolean valid = false;
            for (int k = 0; k < colNum; k++)
                if(board[j][k] == word.charAt(0)){
                    boolean[][] visited = new boolean[rowNum][colNum];
                    valid = isValidWord(board, visited, word, 0, j, k);
                    if(valid){
                        res.add(word);
                        break;
                    }
                }
            if(valid){
                break;
            }
        }
    }
    return res;
}


public boolean isValidWord(char[][] board, boolean[][] visited, String word, int pos,
int x, int y){
    if (x < 0 || x >= board.length || y < 0 || y >= board[0].length){
        return false;
    }
    if (word.charAt(pos) != board[x][y] || visited[x][y]){
        return false;
    }
```

```
        if (pos == word.length() - 1){
            return true;
        }
        visited[x][y] = true;
        if (isValidWord(board, visited, word, pos + 1, x + 1, y) ||
           isValidWord(board, visited, word, pos + 1, x - 1, y) ||
           isValidWord(board, visited, word, pos + 1, x, y + 1) ||
           isValidWord(board, visited, word, pos + 1, x, y - 1)){
                return true;
           } else {
                visited[x][y]=false;
                return false;
        }
}
```