by eamon 06/03/2015

Chapter 8 Data Structure

Min Stack
Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.
push(x) -- Push element x onto stack.
pop() -- Removes the element on top of the stack.
top() -- Get the top element.
getMin() -- Retrieve the minimum element in the stack.
Answer:

```
/*
Very very trick. we should use EQUALS here instead of "=="
push
6
1
7 1
4 4
pop 6 1 1 7 4 4
notice here pop method has no return value, void.
pop section: 概念上, 用了两个stack去实现minStack, 实际宏观功能上, 还是视为一个stack
*/
public class MaxStackSol {
	public static class MaxStack {
		Stack<Integer> stack1 = new Stack<Integer>();
		Stack<Integer> stack2 = new Stack<Integer>();
		public void push(int x) {
			stack1.push(x);
			if (stack2.isEmpty() || x <= stack2.peek()) {
				stack2.push(x);
			}
		}

		public void pop() {
			if (stack1.peek().equals(stack2.peek())) {
				stack2.pop();
			}
			stack1.pop();
		}

		public int top() {
			return stack1.peek();
		}

		public int getMin() {
			return stack2.peek();
		}
	}
```

```
        public static void main(String[] args) {
                MaxStack s = new MaxStack();
                s.push(4);
                s.push(7);
                s.push(1);
                s.push(6);
                System.out.print(s.getMin());
        }
}
```

Implement Queue by Two Stacks
As the title described, you should only use two stacks to implement a queue's actions.
The queue should support push(element), pop() and top() where pop is pop the first(a.k.a front) element in the queue.
Both pop and top methods should return the value of first element.
Example
For push(1), pop(), push(2), push(3), top(), pop(), you should return 1, 2 and 2
Challenge
implement it by two stacks, do not use any other data structure and push, pop and top should be O(1) by AVERAGE.
Answer:

```
/*
每个数进出Stack1和Stack2各1次, 所以两个操作的均摊复杂度均为O(1)
*/
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
Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.
Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].
The largest rectangle is shown in the shaded area, which has area = 10 unit.
For example,
Given height = [2,1,5,6,2,3],
return 10.
Answer:

```
/*
O(n) runtime
*/
public class Solution {
    public int largestRectangleArea(int[] height) {
        if (height == null || height.length == 0) {
            return 0;
        }

        Stack<Integer> s = new Stack<Integer>();
        int len = height.length;
        int max = 0;
        int i = 0;
        while (i <= len) {
            if (s.isEmpty() || (i < len && height[i] >= height[s.peek()])) {
                s.push(i);
                i++;
            } else {
                int h = height[s.pop()];
                int width;
                if (s.isEmpty()) {
                    width = i;
                } else {
                    width = i - s.peek() - 1;
                }
                max = Math.max(max, h * width);
            }
        }
        return max;
    }
}
```

Sum Root to Leaf Numbers
Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.
An example is the root-to-leaf path 1->2->3 which represents the number 123.
Find the total sum of all root-to-leaf numbers.
For example,

```
   1
  / \
 2   3
```

The root-to-leaf path 1->2 represents the number 12.
The root-to-leaf path 1->3 represents the number 13.
Return the sum = 12 + 13 = 25.
Answer:

```
public class SumNumsSol {
	public static int sumNumbers(TreeNode root) {
		return dfs(root, 0);
	}

	public static int dfs(TreeNode root, int pre) {
		if (root == null) {
			return 0;
		}

		int cur = pre * 10 + root.val;
		if (root.left == null && root.right == null) {
			return cur;
		}
		return dfs(root.left, cur) + dfs(root.right, cur);
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
	}

	public static void main(String[] args) {
		TreeNode t1 = new TreeNode(1);
		TreeNode t2 = new TreeNode(2);
		TreeNode t3 = new TreeNode(3);
		t1.left = t2;
		t1.right = t3;
		System.out.print(sumNumbers(t1));
	}
}
```

Max Tree
Given an integer array with no duplicates. A max tree building on this array is defined as follow:
The root is the maximum number in the array
The left subtree and right subtree are the max trees of the subarray divided by the root number.
Construct the max tree by the given array.
Example
Given [2, 5, 6, 0, 3, 1], the max tree constructed by this array is:

```
   6
  / \
 5   3
/   / \
2   0   1
```

Challenge
O(n) time and memory.
Answer:

```
public class MaxTreeSol {
    public static TreeNode maxTree(int[] A) {
        int len = A.length;
        TreeNode[] rst = new TreeNode[len];
        for (int i = 0; i < len; i++) {
            rst[i] = new TreeNode(0);
        }
        int cnt = 0;
        for (int i = 0; i < len; i++) {
            TreeNode tmp = new TreeNode(A[i]);
            while (cnt > 0 && A[i] > rst[cnt - 1].val) {
                tmp.left = rst[cnt - 1];
                cnt--;
            }
            if (cnt > 0) {
                rst[cnt - 1].right = tmp;
            }
            rst[cnt] = tmp;
            cnt++;
        }
        return rst[0];
    }

    public static class TreeNode {
        public int val;
        public TreeNode left, right;
        public TreeNode(int val) {
            this.val = val;
            this.left = this.right = null;
        }
    }

    public static void main(String[] args) {
        int[] A = {2, 5, 6, 0, 3, 1};
        TreeNode rst = maxTree(A);
        System.out.print(LOrder(rst));
    }

    public static List<List<Integer>> LOrder(TreeNode root) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>();
        if (root == null) {
            return rst;
        }
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.offer(root);
        while (!q.isEmpty()) {
            List<Integer> list = new ArrayList<Integer>();
            int size = q.size();
            for (int i = 0; i < size; i++) {
                TreeNode tmp = q.poll();
                list.add(tmp.val);
                if (tmp.left != null) {
                    q.offer(tmp.left);
                }
                if (tmp.right != null) {
                    q.offer(tmp.right);
                }
            }
            rst.add(list);
        }
        return rst;
    }
}
```

LRU Cache
Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.
get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.
Answer:

```
/*
SOL 1
使用HashMap + 双向链表实现:
1. 如果需要移除老的节点, 我们从头节点移除.
2. 如果某个节点被访问(SET/GET), 将其移除并挂在双向链表的结尾.
3. 链表满了后, 我们删除头节点.
4. 最近访问的节点在链尾, 最久被访问的节点在链头.
addTail is to add new node in front of the tail
SOL 2
1. OverRide removeEldestEntry 函数, 在Size达到最大值, 删除最长时间未访问的节点
```

2. 在Get/ Set的时候, 都更新节点, 即删除之, 再添加之, 这样它会作为最新的节点加到双向链表中
*/
```
public class LRUSol {
	public static class LRUCache1 {
		private class DLink {
			DLink pre;
			DLink next;
			int val;
			int key;
			DLink(int key, int val) {
				this.val = val;
				this.key = key;
				pre = null;
				next = null;
			}
		}

		HashMap<Integer, DLink> map;
		DLink head;
		DLink tail;
		int capacity;

		public void removeFirst() {
			removeNode(head.next);
		}
		public void removeNode(DLink node) {
			node.pre.next = node.next;
			node.next.pre = node.pre;
		}
		public void addToTail(DLink node) {
			tail.pre.next = node;
			node.pre = tail.pre;
			node.next = tail;
			tail.pre = node;
		}

		public LRUCache1(int capacity) {
			map = new HashMap<Integer, DLink>();
			head = new DLink(-1, -1);
			tail = new DLink(-1, -1);
			head.next = tail;
			tail.pre = head;
			this.capacity = capacity;
		}

		public int get(int key) {
			if (map.get(key) == null) {
				return -1;
			}
			DLink node = map.get(key);
			removeNode(node);
			addToTail(node);
			return node.val;
		}

		public void set(int key, int value) {
			DLink node = map.get(key);
			if (node == null) {
				node = new DLink(key, value);
				map.put(key, node);
			} else {
				node.val = value;
				removeNode(node);
			}
			addToTail(node);
			if (map.size() > capacity) {
				map.remove(head.next.key);
				removeFirst();
			}
		}
	}

	public static class LRUCache2 {
		LinkedHashMap<Integer, Integer> map;
		int capacity;

		public LRUCache2(final int capacity) {
			map = new LinkedHashMap<Integer, Integer>(capacity) {
				private static final long serialVersionUID = 1L;
				protected boolean removeEldestEntry(Map.Entry eldest) {
					return size() > capacity;
				}
			};
			this.capacity = capacity;
		}

		public int get(int key) {
			Integer ret = map.get(key);
			if (ret == null) {
				return -1;
			} else {
				map.remove(key);
				map.put(key, ret);
			}
			return ret;
		}

		public void set(int key, int value) {
				map.remove(key);
				map.put(key, value);
			}
	}

	public static void main(String[] args) {
		LRUCache1 lrc2 = new LRUCache1(2);
		lrc2.set(1, 3);
		lrc2.set(2, 2);
		lrc2.set(1, 4);
		lrc2.set(4, 2);
		System.out.print(lrc2.get(1));
	}
}
```

Longest Consecutive Sequence
Given an unsorted array of integers, find the length of the longest consecutive elements sequence.
For example,
Given [100, 4, 200, 1, 3, 2],
The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.
Your algorithm should run in O(n) complexity.
Answer:
/*
SOL 1
Sort & search: space O(1), time O(n logn)
HashMap: space O(n), time O(n)
用HashMap来空间换时间.
1. 在map中创建一些集合来表示连续的空间. 比如, 如果有[3, 4, 5]这样的一个集合, 我们表示为key: 3, value: 5和key: 5, value: 3两个集合, 并且把这2个放在hashmap中. 这样我们可以在O(1)的时间查询某个数字开头和结尾的集合.
2. 来了一个新的数字时, 比如: N=6, 我们可以搜索以N-1结尾, 以N+1开头的集合有没有存在. 从1中可以看到, key: 5是存在的, 这样我们可以删除3, 5和5, 3这两个key-value对, 同样我们要查以7起头的集合有没有存在, 同样可以删除以7起始的集合. 删除后我们可以更新left, right的值, 也就是合并和扩大集合.
3. 合并以上这些集合, 创建一个以新的left, right作为开头, 结尾的集合, 分别以left, right作为key存储在map中. 并且更新max (表示最长连续集合)
remove(Object key)
Removes the mapping for the specified key from this map if present.
SOL 2
把所有的数字放在hashset中, 来一个数字后, 取出HashSet中的某一元素x, 找x - 1, x - 2, ..., x + 1, x + 2, ... 是否也在set里
*/

```
public class LCSeqSol {
	public static int longestConsecutive1(int[] nums) {
		if (nums == null) {
			return 0;
		}
```

```
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        int max = 0;
        int len = nums.length;
        for (int i = 0; i < len; i++) {
            if (map.get(nums[i]) != null) {
                continue;
            }

            int left = nums[i];
            int right = nums[i];
            Integer bound = map.get(nums[i] - 1);
            if (bound != null && bound < left) {
                left = bound;
                map.remove(left);
                map.remove(nums[i] - 1);
            }

            bound = map.get(nums[i] + 1);
            if (bound != null && bound > right) {
                right = bound;
                map.remove(right);
                map.remove(nums[i] + 1);
            }

            map.put(left, right);
            map.put(right, left);
            max = Math.max(max, right - left + 1);
        }
        return max;
    }

    public static int longestConsecutive2(int[] nums) {
        if (nums == null) {
            return 0;
        }

        HashSet<Integer> set = new HashSet<Integer>();
        for (int i : nums) {
            set.add(i);
        }
        int max = 0;
        for (int i : nums) {
            set.remove(i);
            int sum = 1;
            int tmp = i - 1;
            while (set.contains(tmp)) {
                set.remove(tmp);
                sum++;
                tmp--;
                }
                tmp = i + 1;
                while (set.contains(tmp)) {
                    set.remove(tmp);
                    sum++;
                    tmp++;
                }
                max = Math.max(max, sum);
            }
            return max;
        }

        public static void main(String[] args) {
            int[] nums = {100, 4, 200, 1, 3, 2};
            System.out.print(longestConsecutive2(nums));
        }
}
```

Valid Parentheses
Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.
Answer:

```
/*
使用stack来解决的简单题目. 所有的字符依次入栈
1. 遇到成对的括号弹栈, 弹栈不成对返回false.
2. 栈为空只能压入左括号
3. 扫描完成时, 栈应该为空, 否则返回FALSE.
*/
public class isValidSol {
        public static boolean isValid1(String s) {
            if (s == null || s.length() == 0) {
                return true;
            }

            Stack<Character> stack1 = new Stack<Character>();
            int len = s.length();
            for (int i = 0; i < len; i++) {
                char c = s.charAt(i);
                if (stack1.isEmpty()) {
                    if (c == ')' || c == ']' || c == '}') {
                        return false;
                    }
                    stack1.push(c);
                    continue;
                }

                if (c == ')' && stack1.peek() == '('
                        || c == ']' && stack1.peek() == '['
                        || c == '}' && stack1.peek() == '{') {
                    stack1.pop();
                } else if (c == '(' || c == '[' || c == '{') {
                    stack1.push(c);
                } else {
                    return false;
                }
            }
            return stack1.isEmpty();
        }

        public static boolean isValid2(String s) {
            if (s == null) {
                return false;
            }

            int len = s.length();
            Stack<Character> stack2 = new Stack<Character>();
            for (int i = 0; i < len; i++) {
                char c = s.charAt(i);
                switch(c) {
                case '(':
                case '[':
                case '{':
                    stack2.push(c);
                    break;
                case ')':
                    if (!stack2.isEmpty() && stack2.peek() == '(') {
                        stack2.pop();
                    } else {
                        return false;
                    }
                    break;
                case ']':
                    if (!stack2.isEmpty() && stack2.peek() == '[') {
                        stack2.pop();
                    } else {
                        return false;
                    }
                    break;
                case '}':
                    if (!stack2.isEmpty() && stack2.peek() == '{') {
                        stack2.pop();
                    } else {
                        return false;
                    }
                    break;
                }
```

```
		}
		return stack2.isEmpty();
	}
	/*
	if there are other pairs, like (a, 1) (b, 2) (c, 3)
	*/
	/*
	keySet() includes '(' '[' '{' this section must be put into stack first, following the order of 
normal sentence
	value() includes ')' ']' '}' then this section later
	*/
	public static boolean isValid3(String s) {
		if (s == null || s.length() % 2 == 1) {
			return false;
		}

		HashMap<Character, Character> map = new HashMap<Character, Character>();
		map.put('(', ')');
		map.put('[', ']');
		map.put('{', '}');
		/*
		 * map.put('a', '1');
		 * map.put('b', '2');
		 */
		Stack<Character> stack3 = new Stack<Character>();
		for (int i = 0; i < s.length(); i++) {
			char c = s.charAt(i);
			if (map.containsKey(c)) {
				stack3.push(c);
			} else if (map.containsValue(c)) {
				if (!stack3.isEmpty() && map.get(stack3.peek()) == c) {
					stack3.pop();
				} else {
					return false;
				}
			}
		}
		return stack3.isEmpty();
	}

	public static void main(String[] args) {
		String test = "([)]";
		System.out.print(isValid1(test) + "\n");
		System.out.print(isValid2(test) + "\n");
		System.out.print(isValid3(test) + "\n");
	}
}
```

Generate Parentheses
Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.
For example, given n = 3, a solution set is:
"((()))", "(()())", "(())()", "()(())", "()()()"
Answer:

```
/*
九章算法的递归模板.
1. Left代表余下的'('的数目
2. right代表余下的')'的数目
3. 注意right余下的数目要大于left, 否则就是非法的, 比如, 先放一个')'就是非法的.
4. 任何一个小于0, 也是错的.
5. 递归的时候, 我们只有2种选择, 就是选择'('还是选择')'
6. 递归的时候, 一旦在结果的路径上尝试过'('还是选择')', 都需要回溯, 即是
sb.deleteCharAt(sb.length() - 1);
*/
/*
left: the left Parentheses
right: the right Parentheses
left < right means that we have more ( then we can add ).
*/
public class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> rst = new ArrayList<String>();
        if (n == 0) {
            return rst;
        }

        dfs(n, n, new StringBuilder(), rst);
        return rst;
    }

    public void dfs(int left, int right, StringBuilder sb, List<String> rst) {
        if (left == 0 && right == 0) {
            rst.add(sb.toString());
            return;
        }

        if (left < 0 || right < 0 || left > right) {
            return;
        }

        dfs(left - 1, right, sb.append('('), rst);
        sb.deleteCharAt(sb.length() - 1);
        dfs(left, right - 1, sb.append(')'), rst);
        sb.deleteCharAt(sb.length() - 1);
    }
}
```

Longest Valid Parentheses
Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.
For "(()", the longest valid parentheses substring is "()", which has length = 2.
Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.
Answer:

```
/*
1. tmp 表示当前计算的一套完整的括号集的长度. 完整的指的是消耗掉栈中所有的'('.
2. sum 表示数个完整的括号集的总长.
例子:
有一套完整的括号集，可以加到前面的一整套括号集上
() (()())
 1   2  第二套括号集可以加过来
3. 不完整的括号集:
这种情况也是需要计算的. 也可能是一个未完成的括号集, 比如:
() (()() 在这里 ()() 是一个未完成的括号集, 可以独立出来计算, 作为阶段性的结果
4. 栈为空时, 栈中没有'(',出现')', 则必须重置计算
*/
public class Solution {
    public int longestValidParentheses(String s) {
        if (s == null) {
            return 0;
        }
        int len = s.length();
        Stack<Integer> stack = new Stack<Integer>();
        int sum = 0;
        int max = 0;
        for (int i = 0; i < len; i++) {
            char c = s.charAt(i);
            if (c == '(') {
                stack.push(i);
            } else {
                if (stack.isEmpty()) {
                    sum = 0;
                } else {
                    int tmp = i - stack.pop() + 1;
                    if (stack.isEmpty()) {
                        sum += tmp;
                        max = Math.max(max, sum);
                    } else {
                        max = Math.max(max, i - stack.peek());
                    }
                }
            }
        }
        return max;
    }
}
```

Implement strStr()
Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.
Answer:
```
public class Solution {
   public int strStr(String haystack, String needle) {
      if(haystack == null || needle == null) {
         return -1;
      }

      int i, j;
      for(i = 0; i < haystack.length() - needle.length() + 1; i++) {
         for(j = 0; j < needle.length(); j++) {
            if(haystack.charAt(i + j) != needle.charAt(j)) {
               break;
            }
         }
         if(j == needle.length()) {
            return i;
         }
      }
      return -1;
   }
}
```

Implement Trie (Prefix Tree)
Implement a trie with insert, search, and startsWith methods.
Note:
You may assume that all inputs are consist of lowercase letters a-z.
Answer:
```
public class TrieSol {
	public static class TrieNode {
		// Initialize your data structure here.
		boolean have;
		TrieNode[] children;

		public TrieNode() {
			have = false;
			children = new TrieNode[26];
		}
	}

	public static class Trie {
		private TrieNode root;

		public Trie() {
			root = new TrieNode();
		}
		// Inserts a word into the trie.
```

```
        public void insert(String word) {
            TrieNode cur = root;
            int len = word.length();
            for (int  i = 0; i < len; i++) {
                int c = word.charAt(i) - 'a';
                if (cur.children[c] == null) {
                    cur.children[c] = new TrieNode();
                }
                cur = cur.children[c];
            }
            cur.have = true;
        }
        // Returns if the word is in the trie.
        public boolean search(String word) {
            TrieNode cur = root;
            int len = word.length();
            for (int i = 0; i < len; i++) {
                int c = word.charAt(i) - 'a';
                if (cur.children[c] == null) {
                    return false;
                }
                cur = cur.children[c];
            }
            return cur.have;
        }
        // Returns if there is any word in the trie
     // that starts with the given prefix.
        public boolean startsWith(String prefix) {
            TrieNode cur = root;
            int len = prefix.length();
            for (int i = 0; i < len; i++) {
                int c = prefix.charAt(i) - 'a';
                if (cur.children[c] == null) {
                    return false;
                }
                cur = cur.children[c];
            }
            return true;
        }
    }
    // Your Trie object will be instantiated and called as such:
    // Trie trie = new Trie();
    // trie.insert("somestring");
    // trie.search("key");
    public static void main(String[] args) {
        Trie trie = new Trie();
        trie.insert("wuhan");
        trie.insert("us");
        System.out.print(trie.search("ma"));
        System.out.print("\n");
        System.out.print(trie.startsWith("wu"));
    }
}
```

Add and Search Word - Data structure design
Design a data structure that supports the following two operations:
void addWord(word)
bool search(word)
search(word) can search a literal word or a regular expression string containing only letters a-z or .. A . means it can represent any one letter.
For example:
addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
Note:
You may assume that all words are consist of lowercase letters a-z.
You should be familiar with how a Trie works. If not, please work on this problem: Implement Trie (Prefix Tree) first.
Answer:

```
/*
字典树 + dfs
search runtime O(1)
*/
public class WordDictSol {
    public static class WordDictionary {
        private TrieNode root;

        public WordDictionary() {
            root = new TrieNode();
        }
        // Adds a word into a data structure.
        public void addWord(String word) {
            TrieNode cur = root;
            int len = word.length();
            for (int i = 0; i < len; i++) {
                int c = word.charAt(i) - 'a';
                if (cur.children[c] == null) {
                    cur.children[c] = new TrieNode();
                }
                cur = cur.children[c];
            }
            cur.have = true;
        }
        // Returns if the word is in the data structure. A word could
        // contain the dot character '.' to represent any one letter.
        public boolean search(String word) {
            TrieNode cur = root;
            int len = word.length();
            return dfs(cur, word, 0, len);
        }

        private boolean dfs(TrieNode node, String word, int pos, int len) {
            if (node == null || (pos == len && !node.have)) {
                return false;
            }
            if (pos == len && node.have) {
                return true;
            }
            if (word.charAt(pos) == '.') {
                for (int i = 0; i < 26; i++) {
                    boolean tmp = dfs(node.children[i], word, pos + 1, len);
                    if (tmp) {
                        return tmp;
                    }
                }
                return false;
            } else {
                int c = word.charAt(pos) - 'a';
                return dfs(node.children[c], word, pos + 1, len);
            }
        }
        /*
        public boolean search(String word) {
            TrieNode cur = root;
            int len = word.length();
            for (int i = 0; i < len; i++) {
                int c = word.charAt(i) - 'a';
                if (cur.children[c] == null) {
                    return false;
                }
                cur = cur.children[c];
            }
            return cur.have;
        }
        */
    }

    public static class TrieNode {
        boolean have;
        TrieNode[] children;
        public TrieNode() {
            have = false;
            children = new TrieNode[26];
            }
        }
        // Your WordDictionary object will be instantiated and called as such:
        // WordDictionary wordDictionary = new WordDictionary();
        // wordDictionary.addWord("word");
        // wordDictionary.search("pattern");
        public static void main(String[] args) {
            WordDictionary wordDictionary = new WordDictionary();
            wordDictionary.addWord("bad");
            wordDictionary.addWord("dad");
            wordDictionary.addWord("mad");
            System.out.print(wordDictionary.search("pad") + "\n");
            System.out.print(wordDictionary.search("bad") + "\n");
            System.out.print(wordDictionary.search(".ad") + "\n");
            System.out.print(wordDictionary.search("b..") + "\n");
        }
}
```

Word Search II
Given a 2D board and a list of words from the dictionary, find all words in the board.
Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.
For example,
Given words = ["oath","pea","eat","rain"] and board =

```
[
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]
```

Return ["eat","oath"].
Note:
You may assume that all inputs are consist of lowercase letters a-z.
You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?
If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: Implement Trie (Prefix Tree) first.
Answer:

```
public class WordSearch2Sol {
   public static List<String> findWords(char[][] board, String[] words) {
      List<String> rst = new ArrayList<String>();
      Trie trie = new Trie();
      for (String word : words) {
         trie.insert(word);
      }
      boolean[][] visit = new boolean[board.length][board[0].length];
```

```
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (trie.root == null) {
                    return rst;
                }
                search(board, i, j, rst, visit, trie.root, trie);
            }
        }
        return rst;
    }

    private static void search(char[][] board, int i, int j, List<String> rst, boolean[][] visit, TrieNode
node, Trie trie) {
        if (node == null) {
            return;
        }
        int c = board[i][j] - 'a';
        TrieNode nextNode = node.children[c];
        if (nextNode == null) {
            return;
        }
        if (nextNode.have) {
            rst.add(nextNode.word);
            trie.remove(nextNode.word);
        }
        int row = board.length;
        int col = board[0].length;
        visit[i][j] = true;
        if (i + 1 < row && !visit[i + 1][j]) {
            search(board, i + 1, j, rst, visit, nextNode, trie);
        }
        if (i - 1 >= 0 && !visit[i - 1][j]) {
            search(board, i - 1, j, rst, visit, nextNode, trie);
        }
        if (j + 1 < col && !visit[i][j + 1]) {
            search(board, i, j + 1, rst, visit, nextNode, trie);
        }
        if (j - 1 >= 0 && !visit[i][j - 1]) {
            search(board, i, j - 1, rst, visit, nextNode, trie);
        }
        visit[i][j] = false;
    }

    public static class TrieNode {
        boolean have;
        TrieNode[] children;
        String word;
        public TrieNode() {
            have = false;
            children = new TrieNode[26];
        }
    }

    public static class Trie {
        private TrieNode root;

        public Trie() {
            root = new TrieNode();
        }

        public void insert(String word) {
            TrieNode cur = root;
            int len = word.length();
            for (int i = 0; i < len; i++) {
                int c = word.charAt(i) - 'a';
                if (cur.children[c] == null) {
                    cur.children[c] = new TrieNode();
                }
                cur = cur.children[c];
            }
            cur.have = true;
            cur.word = word;
        }

        public void remove(String word) {
            TrieNode cur = root;
            int len = word.length();
            for (int i = 0; i < len; i++) {
                int c = word.charAt(i) - 'a';
                TrieNode parent = cur;
                cur = cur.children[c];
            }
            cur.have = false;
            cur.word = null;
        }
    }

    public static void main(String[] args) {
            String[] words = {"oath", "pea", "eat", "rain"};
            char[][] board = {
                                {'o','a','a','n'},
                                {'e','t','a','e'},
                                {'i','h','k','r'},
                                {'i','f','l','v'}
                        };
            System.out.print(findWords(board, words));
    }
}
```

Merge k Sorted Lists
Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.
Answer:

```
/*
这道题目在分布式系统中非常常见, 来自不同client的sorted list要在central server上面merge起来
SOL 1 Divide Conquer
先把k个list分成两半, 然后继续划分, 直到剩下两个list就合并起来, 合并时会用到Merge Two
Sorted Lists这道题,
假设总共有k个list, 每个list的最大长度是n,
T(k) = 2T(k / 2) + O(nk)
    = 2(2T(k / 2^2) + O(nk / 2)) + O(nk) = 4T(k / 4) + 2O(nk)
    = 4(2T(k / 2^3) + O(nk / 2^2)) + 2O(nk) = 8T(k / 2^3) + 3O(nk)
    ...
    = O(nklogk)
空间复杂度的话是递归栈的大小O(logk).
SOL 2, SOL 3,
维护一个大小为k的堆, 每次取堆顶的最小元素放到结果中, 然后读取该元素的下一个元素放入堆
中, 重新维护好。因为每个链表是有序的, 每次又是去当前k个元素中最小的, 所以当所有链表都读
完时结束, 这个时候所有元素按从小到大放在结果链表中。
这个算法每个元素要读取一次, 即是k*n次, 然后每次读取元素要把新元素插入堆中要logk的复杂
度, 所以总时间复杂度是O(nklogk)。
空间复杂度是堆的大小, 即为O(k).
note: “空间复杂度”指占内存大小, “堆排序”每次只对一个元素操作, 是就地排序, 所用辅助空间
O(1), 注意和本题区别
SOL 3
note: 结合LinkedList的性质来考虑这里, left == null代表left这个LinkedList的头结点是空指针, 即
这个LinkedList是空, 故放right前面, 故返回1
*/
public class mergeKListsSol {
	// SOL 1
	public static ListNode mergeKLists1(List<ListNode> lists) {
		if (lists == null || lists.size() == 0) {
			return null;
		}
		return helper1(lists, 0, lists.size() - 1);
	}

	public static ListNode helper1(List<ListNode> lists, int l, int r) {
		if (l < r) {
			int mid = l + (r - 1) / 2;
			return merge(helper1(lists, l, mid), helper1(lists, mid + 1, r));
		}
		return lists.get(l);
	}

	public static ListNode merge(ListNode n1, ListNode n2) {
		ListNode dummy = new ListNode(0);
		ListNode cur = dummy;
		while (n1 != null && n2 != null) {
			if (n1.val < n2.val) {
				cur.next = n1;
				n1 = n1.next;
			} else {
				cur.next = n2;
				n2 = n2.next;
			}
			cur = cur.next;
		}
		if (n1 != null) {
			cur.next = n1;
		} else {
			cur.next = n2;
		}
		return dummy.next;
	}
	// SOL 2 min heap
	public static ListNode mergeKLists2(List<ListNode> lists) {
		if (lists == null || lists.size() == 0) {
			return null;
		}

		Queue<ListNode> heap2 = new PriorityQueue<ListNode>(lists.size(),
Comparator2);
		for (int i = 0; i < lists.size(); i++) {
			if (lists.get(i) != null) {
				heap2.add(lists.get(i));
			}
		}
		ListNode dummy = new ListNode(0);
		ListNode tail = dummy;
		while (!heap2.isEmpty()) {
			ListNode head = heap2.poll();
			tail.next = head;
			tail = head;
			if (head.next != null) {
				heap2.add(head.next);
			}
		}
		return dummy.next;
	}

	private static Comparator<ListNode> Comparator2 = new Comparator<ListNode>() {
		public int compare(ListNode left, ListNode right) {
			if (left == null) {
				return 1;
			} else if (right == null) {
				return -1;
```

```
                }
                return left.val - right.val;
            }
        };
        // SOL 3 min heap
        public static ListNode mergeKLists3(List<ListNode> lists) {
            if (lists == null || lists.size() == 0) {
                return null;
            }
            int size = lists.size();
            PriorityQueue<ListNode> heap3 = new PriorityQueue<ListNode>(size,
                        new Comparator<ListNode>() {
                public int compare(ListNode o1, ListNode o2) {
                    return o1.val - o2.val;
                }
            });

            for (ListNode node : lists) {
                if (node != null) {
                    heap3.offer(node);
                }
            }
            ListNode dummy = new ListNode(0);
            ListNode tail = dummy;
            while (!heap3.isEmpty()) {
                ListNode cur = heap3.poll();
                tail.next = cur;
                tail = tail.next;
                if (cur.next != null) {
                    heap3.offer(cur.next);
                }
            }
            return dummy.next;
        }

        public static class ListNode {
            int val;
            ListNode next;
            ListNode(int x) {
                val = x;
            }
        }

        public static void main(String[] args) {
            ListNode n1 = new ListNode(1);
            ListNode n2 = new ListNode(2);
            ListNode n3 = new ListNode(3);
            ListNode n4 = new ListNode(2);
            ListNode n5 = new ListNode(4);
```

```
		ListNode n6 = new ListNode(6);
		ListNode n7 = new ListNode(3);
		ListNode n8 = new ListNode(5);
		ListNode n9 = new ListNode(7);
		n1.next = n2;
		n2.next = n3;
		n3.next = null;
		n4.next = n5;
		n5.next = n6;
		n6.next = null;
		n7.next = n8;
		n8.next = n9;
		n9.next = null;
		List<ListNode> lists = new ArrayList<ListNode>();
		lists.add(n1);
		lists.add(n4);
		lists.add(n7);
		for (int i = 0; i < lists.size(); i++) {
			ListNode tmp = lists.get(i);
			while (tmp != null) {
				System.out.print(tmp.val);
				tmp = tmp.next;
			}
			System.out.print("\n");
		}
		ListNode rst1 = mergeKLists3(lists);
		while (rst1 != null) {
			System.out.print(rst1.val);
			rst1 = rst1.next;
		}
	}
}
```

Kth Largest Element in an Array
Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.
For example,
Given [3,2,1,5,6,4] and k = 2, return 5.
Note:
You may assume k is always valid, 1 ≤ k ≤ array's length.
Answer:

```
/*
SOL 1
使用改进的Quicksort partition, O(n) time, O(1) space
note: quickselect worst case O(n^2) best/average case O(n)
Choose the last one as the pivot
left: the first one which is bigger than pivot. Change pivot.
*/
```

```
public class KthLargestSol {
	// SOL 1 Quick Sort
	public static int findKthLargest1(int[] nums, int k) {
		if (k < 1 || nums == null) {
			return 0;
		}

		return helper1(nums.length - k + 1, nums, 0, nums.length - 1);
	}

	public static int helper1(int x, int[] nums, int start, int end) {
		int pivot = nums[end];
		int left = start;
		int right = end;
		while (true) {
			while (nums[left] < pivot && left < right) {
				left++;
			}
			while (nums[right] >= pivot && right > left) {
				right--;
			}
			if (left == right) {
				break;
			}
			swap(nums, left, right);
		}
		swap(nums, left, end);
		if (x == left + 1) {
			return pivot;
		} else if (x < left + 1) {
			return helper1(x, nums, start, left - 1);
		} else {
			return helper1(x, nums, left + 1, end);
		}
	}

	public static void swap(int[] nums, int n1, int n2) {
		int tmp = nums[n1];
		nums[n1] = nums[n2];
		nums[n2] = tmp;
	}
	// SOL 2 min heap
	public static int findKthLargest2(int[] nums, int k) {
		if (nums == null || nums.length == 0) {
			return 0;
		}

		int len = nums.length;
		Queue<Integer> heap = new PriorityQueue<Integer>();
		for (int i = 0; i < k; i++) {
			heap.offer(nums[i]);
		}
		for (int j = k; j < len; j++) {
			if (nums[j] > heap.peek()) {
				heap.poll();
				heap.offer(nums[j]);
			} else {
				continue;
			}
		}
		return heap.peek();
	}

	public static void main(String[] args) {
		int[] nums = {3,2,1,5,6,4};
		int k = 2;
		System.out.print(findKthLargest2(nums, k));
	}
}
```

Largest Number
Given a list of non negative integers, arrange them such that they form the largest number.
For example, given [3, 30, 34, 5, 9], the largest formed number is 9534330.
Note: The result may be very large, so you need to return a string instead of an integer.
Answer:

```
/*
类max heap
贪心思路: 对于两个备选数字a和b, 如果str(a) + str(b) > str(b) + str(a), 则a在b之前, 否则b在a之前
按照此原则对原数组从大到小排序即可
时间复杂度O (nlogn) heap sort
易错样例:
Input: [0,0]
Output: "00"
Expected: "0"
*/
public class LargestNumSol {
	public static String largestNumber(int[] nums) {
		if (nums == null) {
			return null;
		}

		ArrayList<Integer> list = new ArrayList<Integer>();
		for (int i : nums) {
			list.add(i);
		}

		Collections.sort(list, new Comparator<Integer>() {
			public int compare(Integer o1, Integer o2) {
```

```
                String s1 = "" + o1 + o2;
                String s2 = "" + o2 + o1;
                return s2.compareTo(s1);
            }
        });

        StringBuilder sb = new StringBuilder();
        for (int i : list) {
            sb.append(i);
        }

        if (sb.charAt(0) == '0') {
            return "0";
        }

        return sb.toString();
    }

    public static void main(String[] args) {
        int[] nums = {3, 30, 34, 5, 9};
        System.out.print(largestNumber(nums));
    }
}

Find median with min heap and max heap in Java
股市上一个股票的价格从开市开始是不停的变化的, 需要开发一个系统, 给定一个股票, 它能实时
显示从开市到当前时间的这个股票的价格的中位数(中值)
*/
/*
SOL 1
1. 维持两个heap, 一个是最小堆, 一个是最大堆.
2. 一直使maxHeap的size大于minHeap.
3. 当两边size相同时, 比较新插入的value, 如果它大于minHeap的最大值, 把它插入到minHeap. 并
且把minHeap的最小值移动到maxHeap.
SOL 2
maxHeap保存较小的半边数据, minHeap保存较大的半边数据.
1. 无论如何, 直接把新值插入到maxHeap.
2. 当minHeap为空, 直接退出.
3. 当maxHeap比minHeap多2个值, 直接移动一个值到minHeap即可.
4. 当maxHeap比minHeap多1个值, 比较顶端的2个值, 如果maxHeap的最大值大于minHeap的最
小值, 交换2个值即可.
5. 当maxHeap较大时, 中值是maxHeap的顶值, 否则取2者的顶值的中间值.
*/
public class FindMedianSol {
    private static PriorityQueue<Integer> maxHeap, minHeap;

    public static void main(String[] args) {
        // Style 1
```

```
		/*
		Comparator<Integer> revCmp = new Comparator<Integer>() {
			@Override
			public int compare(Integer left, Integer right) {
				return right.compareTo(left);
			}
		};

		maxHeap = new PriorityQueue<Integer>(20, revCmp);
		minHeap = new PriorityQueue<Integer>(20);
		*/
		// Style 2
		maxHeap = new PriorityQueue<Integer>(20, new Comparator<Integer>() {
			public int compare(Integer o1, Integer o2) {
				return o2 - o1;
			}
		});
		minHeap = new PriorityQueue<Integer>(20);

		addNumber2(6);
		addNumber2(4);
		addNumber2(3);
		addNumber2(10);
		addNumber2(12);
		System.out.println(minHeap);
		System.out.println(maxHeap);
		System.out.println(getMedian());
		addNumber2(5);
		System.out.println(minHeap);
		System.out.println(maxHeap);
		System.out.println(getMedian());
		addNumber2(7);
		addNumber2(8);
		System.out.println(minHeap);
		System.out.println(maxHeap);
		System.out.println(getMedian());
	}
	/*
	public static void addNumber1(int value) {
		if (maxHeap.size() == minHeap.size()) {
			if (minHeap.peek() != null && value > minHeap.peek()) {
				maxHeap.offer(minHeap.poll());
				minHeap.offer(value);
			} else {
				maxHeap.offer(value);
			}
		} else {
			if (value < maxHeap.peek()) {
				minHeap.offer(maxHeap.poll());
```

```
					maxHeap.offer(value);
				} else {
					minHeap.offer(value);
				}
			}
		}
		*/
		public static void addNumber2(int value) {
			maxHeap.offer(value);
			if (maxHeap.size() - minHeap.size() == 2) {
				minHeap.offer(maxHeap.poll());
			} else {
				if (minHeap.isEmpty()) {
					return;
				}
				if (minHeap.peek() < maxHeap.peek()) {
					minHeap.offer(maxHeap.poll());
					maxHeap.offer(minHeap.poll());
				}
			}
		}

		public static double getMedian() {
			if (maxHeap.isEmpty()) {
				return -1;
			}

			if (maxHeap.size() > minHeap.size()) {
				return maxHeap.peek();
			} else {
				return (double)(minHeap.peek() + maxHeap.peek()) / 2;
			}
		}
}
/*
 * outputs:
 * [10, 12]
 * [6, 3, 4]
 * 6.0
 * [6, 12, 10]
 * [5, 3, 4]
 * 5.5
 * [7, 8, 10, 12]
 * [6, 5, 4, 3]
 * 6.5
 */
```