“零基础”数据结构入门

Array, String, Linked List, Tree, Trie,
Stack, Queue, Deque, PriorityQueue(Heap),
HashMap, HashSet, TreeMap, TreeSet
Disjoint-Set(Union Find), Graph
Segment Tree(zkw Tree), Binary Index Tree(Fenwick Tree)

你们的好朋友Eddie

很多教材有自己的分类，参考常见的8大类

实际面试考察的则比教材上多一点，更有侧重

常考

Array, String(non-primitive data type), Linked List, Tree(BT, BST), Stack, Queue, PriorityQueue(Heap), HashMap, HashSet, Trie

少考

Disjoint-Set(Union Find), Deque, Graph

一般不考，但是用来一题多解更快。

TreeMap, TreeSet

Segment Tree(zkw Tree), Binary Index Tree(Fenwick Tree)

你们的好朋友古城

Array 数组

数组是可以再内存中连续存储多个元素的结构，在内存中的分配也是连续的，数组中的元素通过数组下标进行访问，数组下标从0开始。例如下面这段代码就是将数组的第一个元素赋值为 1。

初始化　int[] nums = new int[100];

赋值　　nums[0] = 1;

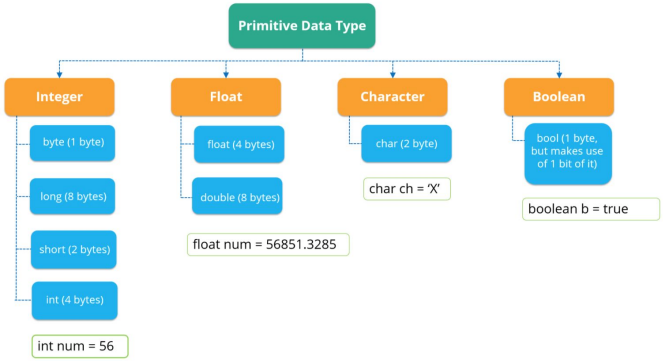
使用　　nums[0]　时间复杂度O(1) nums[index]

优点：
1、按照索引查询元素速度快
2、按照索引遍历数组方便

缺点：
1、数组的大小固定后就无法 扩容了
2、数组只能存储一种类型的数据
3、添加，删除的操作慢，因 为要移动其他的元素。

适用场景：
频繁查询，对存储空间要求不大，很少增加和 删除的情况。

String 字符串



严格说不是数据结构，是一种non-primitive data type，

primitive type指的是**boolean, int, char, double, long, byte, short, float**

non-primitive type指的是

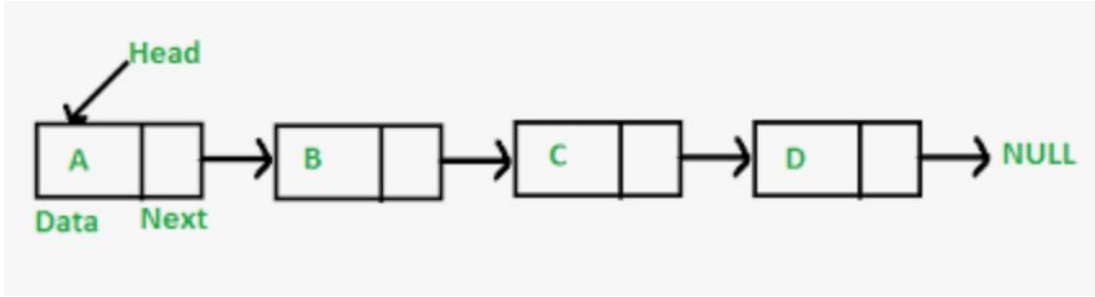
String is a sequence of characters. But in Java, a string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

```
//2 String
String str = new String( original: "古城666");
str = new String(new char[]{'古', '城', '6', '6', '6'});
```

常用method
str.substring();
str.charAt(index);
str1.compareTo(str2);

Linked List 链表

链表是物理存储单元上非连续的、非顺序的存储结构，数据元素的逻辑顺序是通过链表的指针地址实现，每个元素包含两个结点，一个是存储元素的数据域 (内存空间)，另一个是指向下一个结点地址的指针域。根据指针的指向，链表能形成不同的结构，例如单链表，双向链表，循环链表等。



常用method，赋值取值时间复杂度均为O(1)
初始化ListNode head = new ListNode(0);
赋值 head.next = new ListNode(1);
取值 head.val

链表的优点：
链表是很常用的一种数据 结构，不需要初始化容量，可以任意加减元素；
添加或者删除元素时只需要改变前后两个元素结点的指针域指向地址即可，所以添加，删除很快

适用场景：
数据量较小，需要频繁增加，删除操作的场景

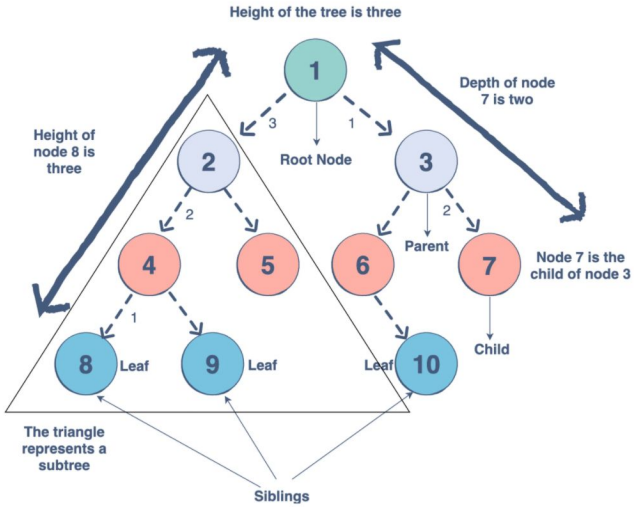
缺点：
因为含有大量的指针域，占用空间较大；
查找元素需要遍历链表来查找，非常耗时。

```
28  class ListNode {
29      int val;
30      ListNode next;
31      public ListNode(int val) {
32          this.val = val;
33      }
34  }
```

Tree 树

树是一种数据结构，它是由n(n>=1)个有限节点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

每个节点有零个或多个子节点；
没有父节点的节点称为根节点；
每一个非根节点有且只有一个父节点；
除了根节点外，每个子节点可以分为多个不相交的子树；



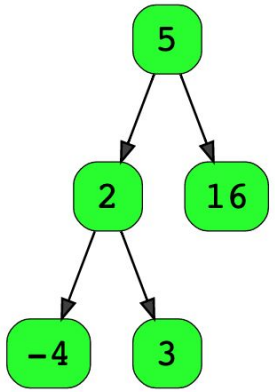
Binary Tree二叉树是树的特殊一种，具有如下特点：

1、每个结点最多有两颗子树
2、左子树和右子树是有顺序的

初始化

```
TreeNode root = new TreeNode(0);
root.left = new TreeNode(1);
root.right = new TreeNode(2);
```

Binary Search Tree



left < parent < right

```
35  class TreeNode {
36      int val;
37      TreeNode left;
38      TreeNode right;
39      public TreeNode(int val) {
40          this.val = val;
41      }
42  }
```

Trie 前缀树或字典树 **pronounce** it /ˈtraɪ/ (as "try"), in an attempt to distinguish it verbally from "tree".

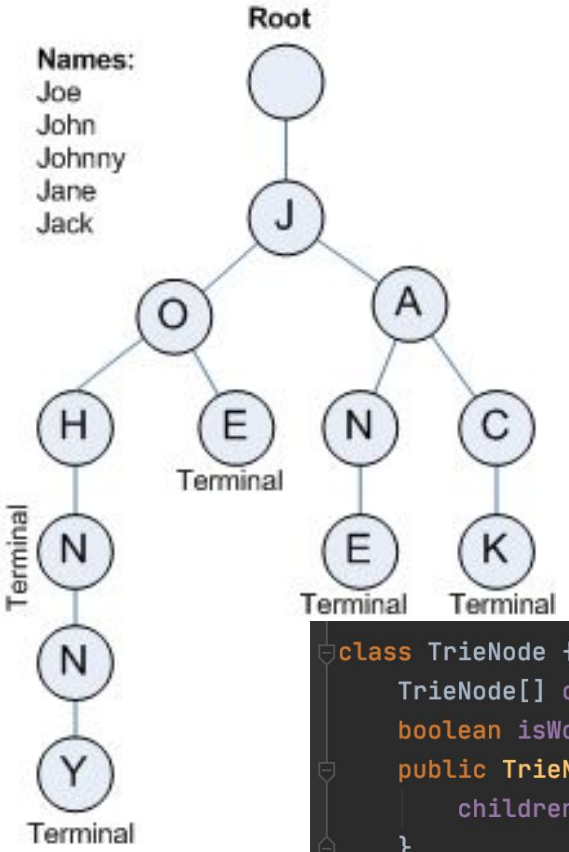
Trie树，又叫**字典树**、**前缀树（Prefix Tree）**、**单词查找树** 或 **键树**，是一种多叉树结构。如下图：

Trie is like a N-array Tree.

字典树的性质

1. 根节点（Root）不包含字符，除根节点外的每一个节点都仅包含一个字符；
2. 从根节点到某一节点路径上所经过的字符连接起来，即为该节点对应的字符串；
3. 任意节点的所有子节点所包含的字符都不相同；

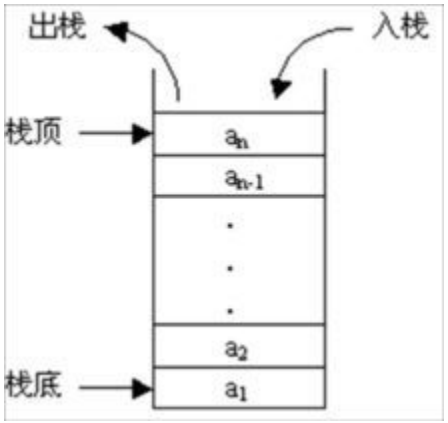
基础method: addWord(), searchWord()
searchPrefix(), 时间复杂度均为word.length

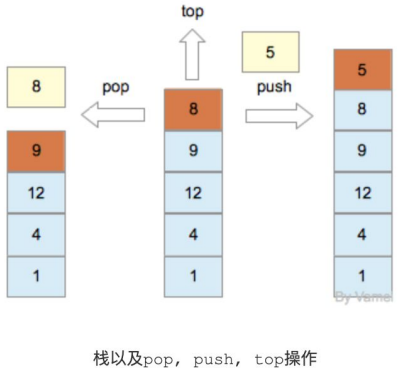


详解请见Trie专题PPT

```
class TrieNode {
    TrieNode[] children;
    boolean isWord;
    public TrieNode() {
        children = new TrieNode[26];
    }
```

Stack 栈 Last In First Out (LIFO) or First In Last Out (FILO)





栈以及pop, push, top操作

甲骨文官方doc推荐使用Deque来代替Stack, 因为内部实现更合理, Vector vs Queue interface, 这里只是知道即可不需要深入了解

A more complete and consistent set of LIFO stack operations is provided by the `Deque` interface and its implementations, which should be used in preference to this class. For example:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

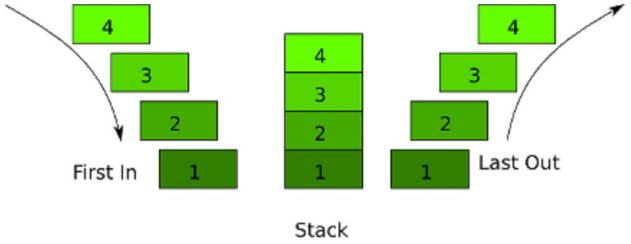
栈是一种特殊的线性表，仅能在线性表的一端操作，栈顶允许操作，栈底不允许操作。栈的特点是:先进后出，或者说是后进先出，从栈顶放入元素的操作叫入栈，取出元素叫出栈。

栈的结构就像一个集装箱，越先放进去的东西越晚才能拿出来，所以，栈常应用于实现递归功能方面的场景，DFS均可以使用栈来实现

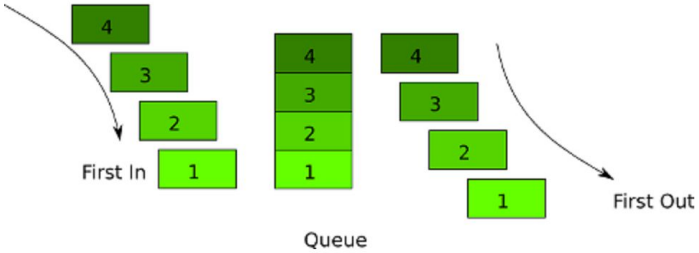
初始化Stack<Integer> stack = new Stack<>();
常用method,时间复杂度均为O(1)
stack.push(num)
stack.peek()　　//will not remove, 只看栈顶
stack.pop()　　//栈顶元素弹出来
stack.isEmpty()　//查看栈是否为空

更复杂的单调栈请见专题PPT
基础算法(六) -- 单调栈

Queue 队列 First In First Out (FIFO)




```
Queue<Integer> queue = new LinkedList<>();
 queue.offer(1); //推荐
 queue.offer(2);
 queue.poll();    //return 1
 queue.poll();    //return 2
 时间复杂度均为O(1)
```



注意区分这里的LinkedList class, 和之前的数据结构Linked List（中间有空格）

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

队列与栈一样，也是一种线性表，不同的是，队列可以在一端添加元素，在另一端取出元素，也就是:先进先出。从一端放入元素的操作称为入队，取出元素为出队

常用来实现BFS 宽度优先搜索的遍历

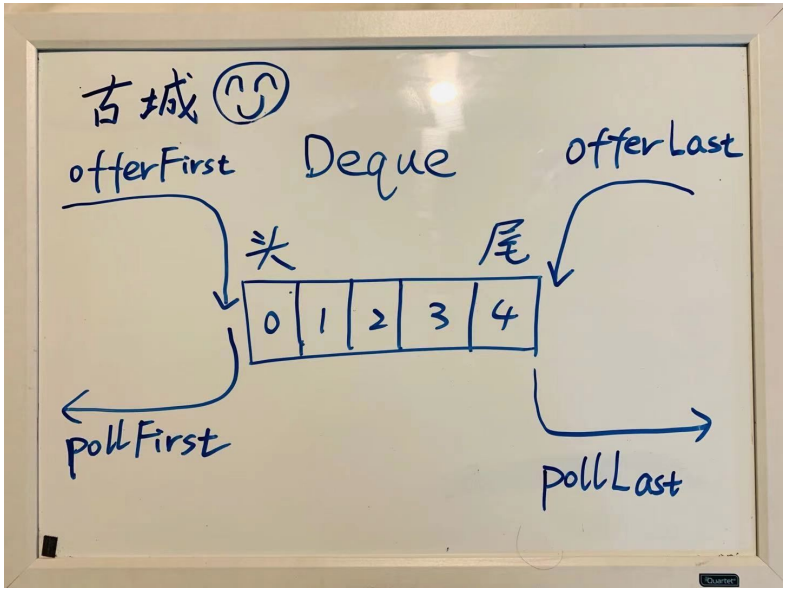
Deque 双端队列（全名Double-ended queue）

我们知道，Queue是队列，只能一头进，另一头出。

如果把条件放松一下，允 许两头都进，两头都出，这种队列叫双端队列（Double Ended Queue），学名Deque。

Java集合提供了接口Deque来实现一个双端队列，它的功能是

既可以添加到队尾，也可以添加到队首；
既可以从队首获取，又可以从队尾获取。



初始化 Deque<Integer> deque = new ArrayDeque<>();
常用method 时间复杂度均为O(1)

```
deque.offerLast(1); // 1
deque.addLast(2);  // 1 -> 2
deque.offerFirst(0); // 0 -> 1 -> 2
deque.peekFirst();  // return 0
deque.pollFirst();    // return 0
deque.pollLast();    // return 2
```

如果直接写`deque.offer()`，我们就需要思考，`offer()`实际上是`offerLast()`，我们明确地写上`offerLast()`，不需要思考就能一眼看出这是添加到队尾。

因此，使用`Deque`，推荐总是明确调用`offerLast()`/`offerFirst()`或者`pollFirst()`/`pollLast()`方法。

`Deque`是一个接口，它的实现类有`ArrayDeque`和`LinkedList`。

更难的单调队列请见
基础算法(七) -- 单调队列

PriorityQueue(Heap) 堆(最大堆，最小堆)

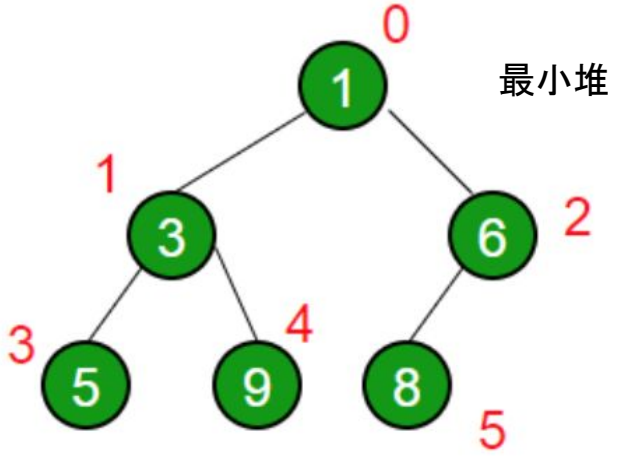
堆分为两种：*最大堆*和*最小堆*，两者的差别在于节点的排序方式。

在最大堆中，父节点的值比每一个子节点的值都要大。在最小堆中，父节点的值比每一个子节点的值都要小。这就是所谓的“堆属性”，并且这个属性对堆中的每一个节点都成立。Java PriorityQueue默认最小堆

初始化 PriorityQueue<Integer> pq = new PriorityQueue<>(); 默认最小堆 初始化一个一个加是O(nlogn), 一次很多数O(n)
常用method

```
pq.offer(2);  //时间复杂度O(logn)
pq.add(0);   //时间复杂度O(logn)
pq.add(1);
pq.peek();  //return 0  时间复杂度O(1)
pq.poll();   //return 0  时间复杂度O(logn)
pq.poll();   //return 1  时间复杂度O(logn)
```

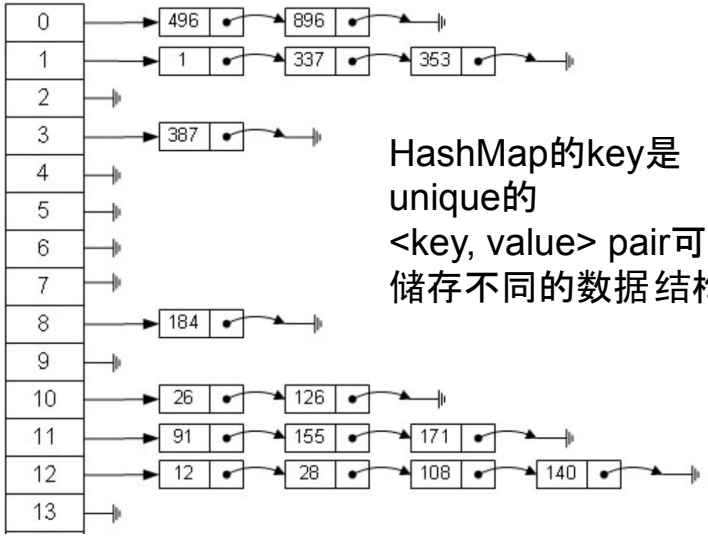
考察点一般为pq应用和array手动实现heap



heapify指的136这三个数字的小三角，如果parent 1不小于3.6就要交换位置。然后一路往下call到叶子节点

详解及更难的heap sort请看
基础数据结构(三) -- Heap

Map 散列表，也叫哈希表 (Java HashMap)



HashMap的key是unique的
<key, value> pair可以储存不同的数据结构

初始化 Map<String, Integer> map = new HashMap<>();
常用method **O(1)的时间复杂度添加和查找**

```
map.put("A", 0);
map.put("B", 1);
map.put("C", 2);
map.get("A"));              //return 0
map.get("C"));              //return 2
map.containsKey("B")); //return true
map.toString());           //{A=0, B=1, C=2}
```

散列表，也叫哈希表，是根据关键码和值 (key和value) 直接进行访问的数据结构，通过key和value来映射到集合中的一个位置,这样就可以很快找到集合中的对应元素。

记录的存储位置=f(key)

这里的对应关系 **f** 成为散列函数，又称为哈希 (hash函数)，而散列表就是把Key通过一个固定的算法函数既所谓的哈希函数转换成一个整型数字，然后就将该数字对数组长度进行取余，取余结果就当作数组的下标，将value存储在以该数字为下标的数组空间里，这种存储空间可以充分利用数组的查找优势来查找元素，所以查找的速度很快。

如果2个不同的string input作为key在hashmap中index出现了冲突如何处理？
一般使用两种方法

1. 挂链表 **Separate Chaining**
2. 开放地址法 **open addressing**

好处：查找比纯链表快，插入删除比纯数组快；

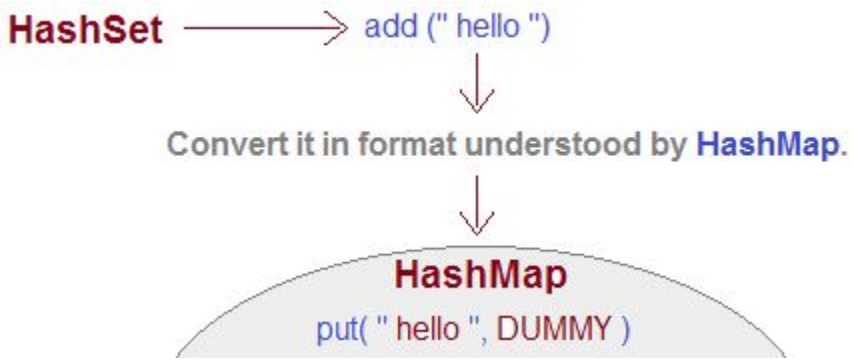
Set（Java HashSet)

特点：

它不允许出现重复元素；

不保证和集合中元素的顺序

允许包含值为null的元素，但最多只能有一个null元素



它是基于HashMap实现的，HashSet底层使用HashMap来保存所有元素，因此HashSet 的实现比较简单，相关HashSet的操作，基本上都是直接调用底层HashMap的相关方法来完成

初始化Set<Integer> set = new HashSet<>();
常用method: **均为O(1)时间复杂度**

```
set.add(1);
set.add(2);
set.contains(1);  //return true
set.contains(3);  //return false
```

TreeMap (Java only)

TreeMap is implemented based on red-black tree structure, and it is ordered by the key.

和hashmap几乎一样的用法，但是提供了key本身有顺序了

时间复杂度O(logn)
put(key, value)
lowerKey() <
floorKey() <=
higherKey() >
ceilingKey() >=

```
124         //12. treeMap
125         TreeMap<Integer, Integer> map2 = new TreeMap<>();
126         map2.put(1, 1);
127         map2.put(0, 0);
128         map2.put(3, 3);
129         map2.get(1);
130         System.out.println("12. treeMap");
131         System.out.println(map2.firstKey());  //return 0
132         System.out.println(map2.floorKey(1)); //return 1 是小于等于1
133         System.out.println(map2.lowerKey(1)); //return 0 必须是小于1
134         System.out.println(map2.lowerKey(0)); //return null
135         System.out.println(map2.ceilingKey(2)); //return 3
```

面试不做要求，因为这只是java有的class，不适用treemap也可以解题，只是用treemap代码可以更短一点方便

TreeSet (Java Only)

几乎和hashSet一样，唯一区别是element现在有顺序了，Order

TreeSet are stored in a sorted and ascending order. **TreeSet** does not preserve the insertion order of elements but elements are sorted by keys.

for insertion order of elements, use LinkedHashSet,
经典题目LRU cache就是手动实现LinkedHashMap

零基础的同学可以跳过，不是必须的，只java选手选择性使用

常用method　first(); //返回最小的元素
lower(num);
floor(num);
higher(num);
ceiling(num);

```
137         //13. TreeSet
138         TreeSet<Integer> treeSet = new TreeSet<>();
139         treeSet.add(0);
140         treeSet.add(1);
141         treeSet.add(3);
142         System.out.println("13. TreeSet");
143         System.out.println(treeSet.lower( e: 1)); //return 0  <
144         System.out.println(treeSet.floor( e: 1)); //return 1  <=
145         System.out.println(treeSet.higher( e: 1));//return 3  >
146
```

Disjoint-Set(Union Find)

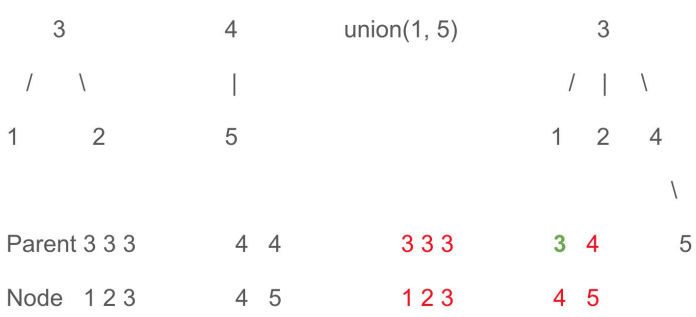
并查集 是一种树形的数据结构，用于处理不交集的合并 (union)及查询(find)问题。找几个数字是否在同一个 group里面

Find：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。

Union：将两个子集合并成同一个集合。

Union Find is a data structure that keeps track of elements which are split into one or more disjoint sets. Its has two primary operations: ***find*** and ***union***.

example



Basic implementation

```
function MakeSet(x)
    x.parent := x
```

```
function Find(x)
    if x.parent == x
       return x
    else
       return Find(x.parent)
```

```
function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    xRoot.parent := yRoot
```

```
39  class DSU {
40      int[] parent;
41      public DSU(int N) {
42          parent = new int[N];
43          for (int i = 0; i < N; i++) parent[i] = i;
44      }
45      public int find(int x) {
46          if (parent[x] != x) parent[x] = find(parent[x]);
47          return parent[x];
48      }
49      public void union(int x, int y) {
50          parent[find(x)] = find(y);
51      }
52  }
```

with path compression

union(x, y) find(x)时间复杂度极快,接近常数级别

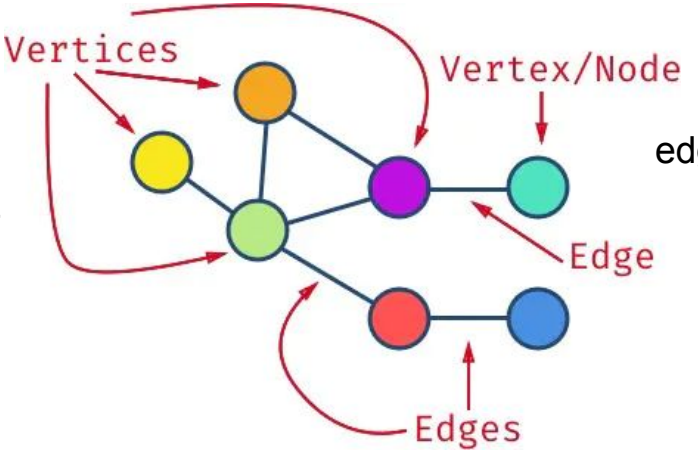
log* is the iterated logarithm.

优化：
path compression
union by rank

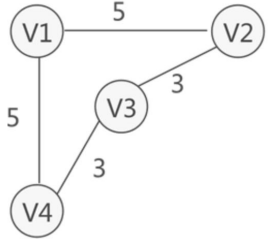
题目及其优化请见
基础数据结构(二) -- 并查集

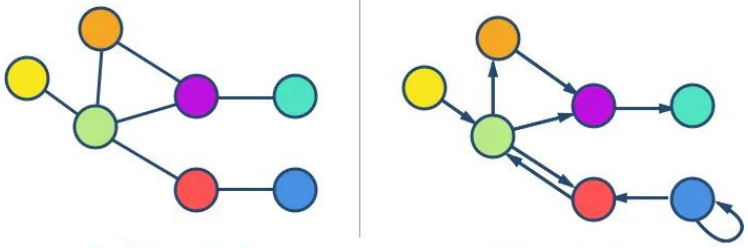
Graph 图

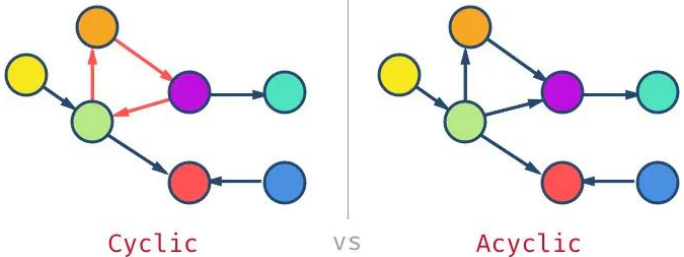
我们知道，数据之间的关系有 3 种，分别是 "一对一"、"一对多" 和 "多对多"，前两种关系的数据可分别用线性表和树结构存储，本节学习存储具有"多对多"逻辑关系数据的结构——图存储结构。



edge本身可以有权重

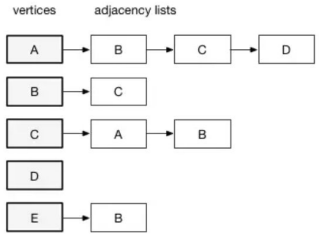






图的表示有两种主要方式：

1. 邻接表
2. 邻接矩阵

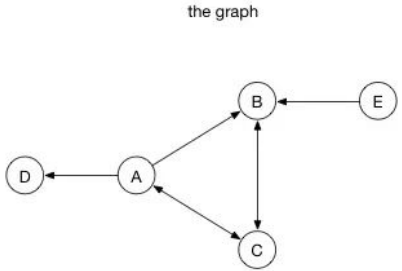


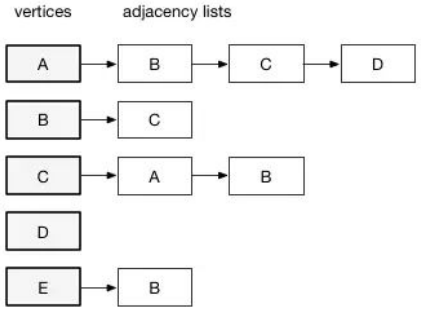
	a	b	c	d	e
a	1	1	–	–	–
b	–	–	1	–	–
c	–	–	–	1	–
d	–	1	1	–	–

图类型查考不多，一般基本 BFS, DFS，拓扑排序即可深入学习可考虑最短路径，最小生成树等，见图的基础算法系列

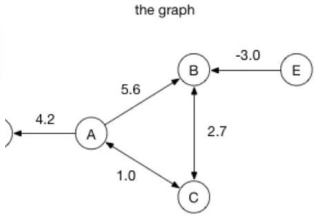
Graph的表示

邻接表





邻接矩阵



adjacency matrix

	A	B	C	D	E
A	0	5.6	1.0	4.2	0
B	0	0	2.7	0	0
C	1.0	2.7	0	0	0
D	0	0	0	0	0
E	0	-3.0	0	0	0

```
163         List[] graph = new ArrayList[4];
164         for (int i = 0; i < graph.length; i++) graph[i] = new ArrayList<>();
165         graph[0].add(1); graph[0].add(3);
166         graph[1].add(2); graph[1].add(0);                    // 邻接表     0 -- 1
167         graph[2].add(1); graph[2].add(3);                    //            |    |
168         graph[3].add(0); graph[3].add(2);                    //            3 -- 2
169         System.out.println(graph[0].toString()); // 0 -> {1, 3}
170         System.out.println(graph[1].toString()); // 1 -> {2, 0}
171
172         Map<Integer, List<Integer>> graph2 = new HashMap<>();
173         for (int i = 0; i < 4; i++) graph2.put(i, new ArrayList<>());
174         graph2.get(0).add(1); graph2.get(0).add(3);
175         graph2.get(1).add(2); graph2.get(1).add(0);
176         graph2.get(2).add(1); graph2.get(2).add(3);
177         graph2.get(3).add(2); graph2.get(3).add(0);
178         System.out.println(graph2.toString()); //{0=[1, 3], 1=[2, 0], 2=[1, 3], 3=[2, 0]}
```

```
181         //邻接矩阵     0 -- 1
182         //           |    |
183         //           3 -- 2
184         boolean[][] graph3 = new boolean[4][4];
185         for (int i = 0; i < 4; i++) graph3[i][i] = true;
186         graph3[0][1] = true; graph3[0][3] = true;
187         graph3[1][2] = true; graph3[1][0] = true;
188         graph3[2][1] = true; graph3[2][3] = true;
189         graph3[3][2] = true; graph3[3][0] = true;
190         System.out.println("      0      1      2      3");
191         for (int i = 0; i < 4; i++) {
192             System.out.println(i + " " + Arrays.toString(graph3[i]));
193         }
194 //        0      1      2      3
195 //        0 [true, true, false, true]
196 //        1 [true, true, true, false]
197 //        2 [false, true, true, true]
198 //        3 [true, false, true, true]
```

Segment Tree 线段树

用来解决什么 问题？

给定一个 长度为 n 的序列，需要 频繁地求其中某个区 间的最值，以及更新某个区间的所有值。

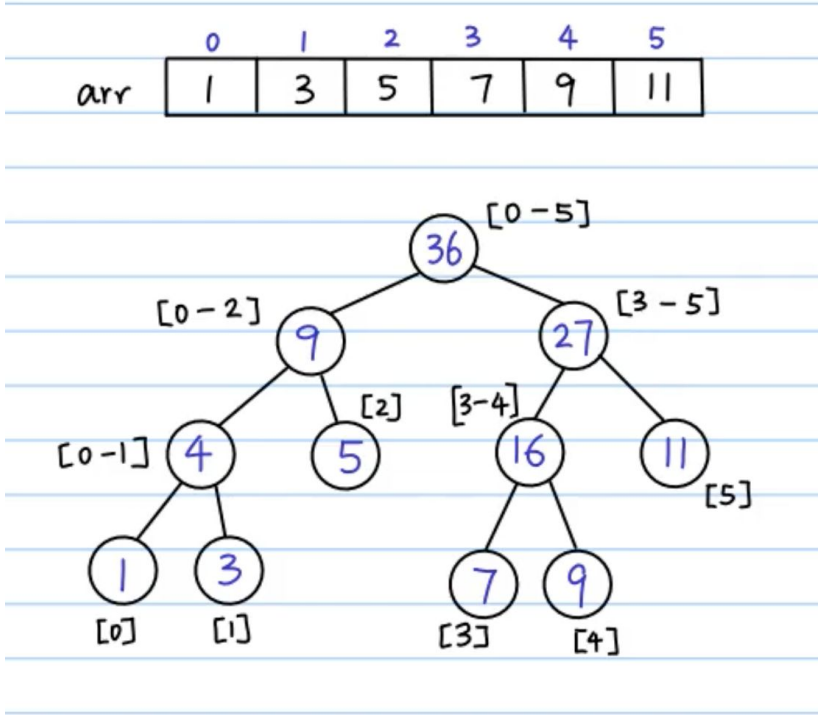
可以logn求区间最大，最小，rangeSum等

常用method：

1.单点修改（logn）update(index, value)

2.区间修改（这里需要用到lazy propogation来优化到logn，完全不在面 试范围）

3.区间查询（logn 区间求和，求区 间最大值，求区间最小值，区间最小公倍数，区间最大公因数）rangeSum(i, j) 一般为[i, j]

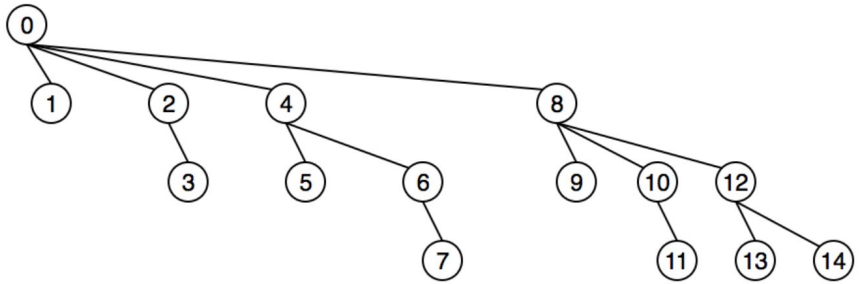


实现方式有
tree指针实现的线段树
zkw线段树

面试极少考，入门不做要求。一般求rangeSum可以考虑用Binary Index Tree或者preFix sum解决。详细请看《数据结构扩展(二) -- 线段树》

Binary Index Tree(Fenwick tree) 树状数组

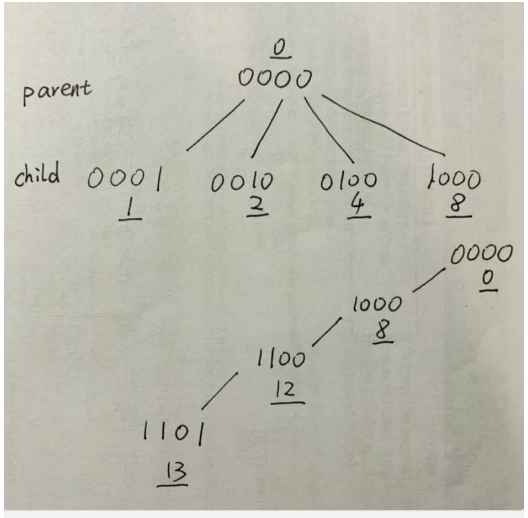
A Fenwick tree or binary indexed tree is a data structure that can efficiently update elements and calculate prefix sums in a table of numbers.



BI Tree for an array arr[] has following operations :
1. **update()** : Updates BI Tree for operation arr[index] += val
2. **getSum()** : Returns sum of arr[0..index]

We first initialize all values in BITree[] as 0.

Then we call **update()** operation for all indexes to insert values according to given array.



Summary

常考(70%)
Array, String(non-primitive data type), Linked List, Tree(BT, BST), Stack, Queue, PriorityQueue(Heap), HashMap, HashSet, Trie

少考(30%)
Disjoint-Set(Union Find), Deque, Graph

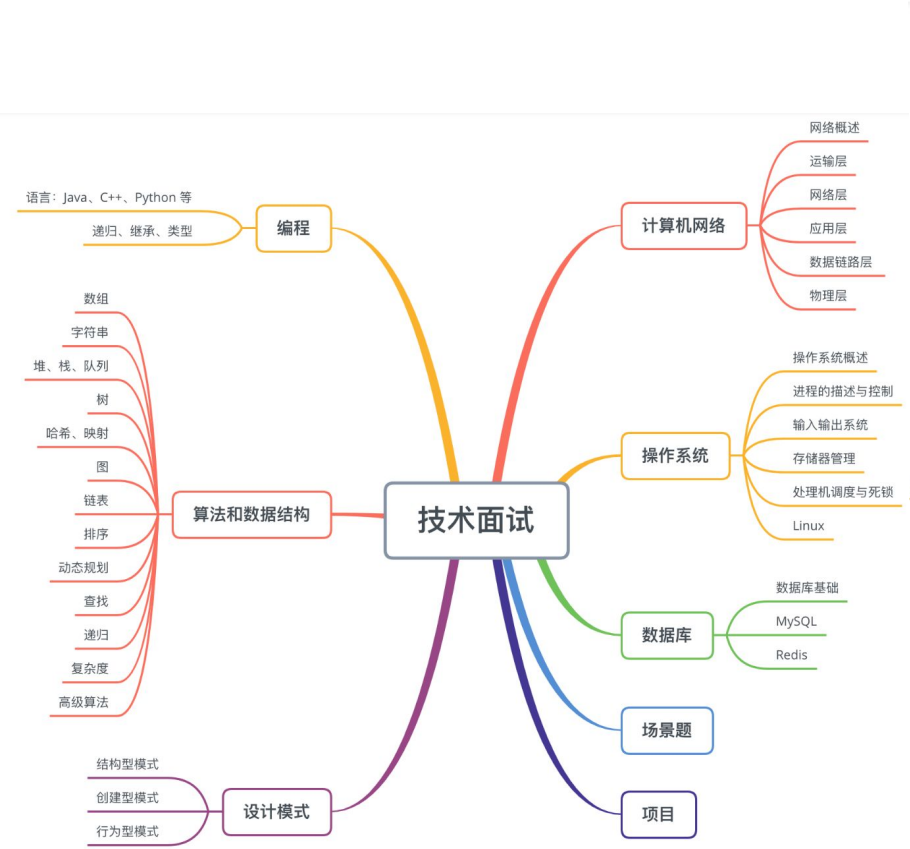
一般不考，但是用来一题多解更快。
TreeMap, TreeSet
Segment Tree(zkw Tree), Binary Index Tree(Fenwick Tree)

基本数据机构了解一般需要配合算法使用，
比如graph图基本的考点在于其中的搜索算法 (当然图考的相对少很多了)
又或者很多动态规划就是基于最基 础的string，array这种数据结构

万里长征始于足下，数据 结构 -> 算法 -> 同步刷题，祝大家面试顺利！

古城

视频都会上传录像，搜ppt title即可



万里长征，始于足下

面对压力别怕，
古城与你背上行囊前行