一。如何实现随机访问

数组是一种线性的数据结构，它有一组连续的内存空间，来存储一组相同类型的数据。

线性表，线性表的数据最多只有前和后两个方向。链表，队列，栈也是线性结构

非线性结构，如二叉树，堆，图

连续的内存和相同类型的数据，使得它可以支持**随机访问，**但是删除和插入，非常低效，需要大量数据搬移工作

内存首地址为 base_address=1000

寻址公式为：a[i]_address =base_address + i *data_type_size
data_type_size 为 4 个字节

二。低效的插入和删除

假设数组长度为 n,现在插入数据到数组第 k 个位置，最坏的时间复杂度是 O(n),平均情况时间复杂度是(1+2+…n)/n=O(n)。
假如数据中的数据没有规律，讲某个数组第 k 个位置的数据搬到数据最后，把新元素直接放入到第
k 个位置。

删除操作

数组 a[10] 中存储了 8 个元素：a，b，c，d，e，f，g，h。现在，我们要依次删除 a，b，c 三个元素。

为了避免 d，e，f，g，h 这几个数据会被搬移三次，我们可以先记录下已经删除的数据。每次的删除操作并不是真正地搬移当数组没有更多空间存储数据时，我们再触发执行一次真正的删除操作，这样就大大减少了删除操作导致的数据搬移。

三。警惕数组的访问越界问题

```
int main(int argc, char* argv[]){
    int i = 0;
    int arr[3] = {0};
    for(; i<=3; i++){
        arr[i] = 0;
        printf("hello world\n");
    }
    return 0;
```

}
会无限打印“hello world”
数组大小为 3，a[0]，a[1]，a[2]，而我们的代码因为书写错误，导致 for 循环的结束条件错写为了 i<=3 而非 i<3，所以当 i=3 时，数组 a[3] 访问越界。
在 C 语言中，只要不是访问受限的内存，所有的内存空间都是可以自由访问的，根据我们前面讲的数组寻址公式，a[3] 也会被定位到某块不属于数组的内存地址上，而这个地址正好是存储变量 i 的内存地址，那么 a[3]=0 就相当于 i=0，所以就会导致代码无限循环。

四。容器能否完全替代数组？

1.Java ArrayList 无法存储基本类型，比如 int、long，需要封装为 Integer、Long 类，而 Autoboxing、Unboxing 则有一定的性能消耗，所以如果特别关注性能，或者希望使用基本类型，就可以选用数组。
2. 如果数据大小事先已知，并且对数据的操作非常简单，用不到 ArrayList 提供的大部分方法，也可以直接使用数组。
3. 还有一个是我个人的喜好，当要表示多维数组时，用数组往往会更加直观。比如 Object[][] array；而用容器的话则需要这样定义：ArrayList<ArrayList> array。

为什么数组初始下标是 0

从数组存储的内存模型上来看，“下标”最确切的定义应该是“偏移（offset）”。前面也讲到，如果用 a 来表示数组的首地址，a[0] 就是偏移为 0 的位置，也就是首地址，a[k] 就表示偏移 k 个 type_size 的位置，所以计算 a[k] 的内存地址只需要用这个公式：
a[k]_address = base_address + (k)*type_size
但是，如果数组从 1 开始计数，那我们计算数组元素 a[k] 的内存地址就会变为：
a[k]_address = base_address + (k-1)*type_size
对比两个公式，我们不难发现，从 1 开始编号，每次随机访问数组元素都多了一次减法运算，对于 CPU 来说，就是多了一次减法指令。

课后题：

JVM 标记清除算法：
大多数主流虚拟机采用可达性分析算法来判断对象是否存活，在标记阶段，会遍历所有 GC ROOTS，将所有 GC ROOTS 可达的对象标记为存活。只有当标记工作完成后，清理工作才会开始。
不足：1.效率问题。标记和清理效率都不高，但是当知道只有少量垃圾产生时会很高效。2.空间问题。会产生不连续的内存空间碎片。

二维数组内存寻址：
对于 m * n 的数组，a [i][j] (i < m,j < n)的地址为：
address = base_address + (i * n + j) * type_size