```
    private int[] items; // 用来存数据的数组
    private int size; // 记录当前有多少个元素


    public AList() {
        items = new int[100]; // 初始大小 100
        size = 0;
    }


    public void addLast(int x) {
        items[size] = x; // 把 x 放到数组的第 size 个位置
        size += 1; // 让 size 增加 1
    }


    public int getLast() {
        return items[size - 1]; // 返回最后一个元素
    }


    public int get(int i) {
        return items[i]; // 获取第 i 个元素
    }


    public int size() {
        return size; // 返回当前的元素个数
    }
}
```

📌 总结
items 是一个数组，默认大小是 100。
size 记录当前有多少个元素。
addLast(x) 把 x 添加到最后。
getLast() 获取最后一个元素。
get(i) 获取第 i 个元素。
AList 的删除操作 removeLast()
如果要删除最后一个元素，最简单的方法是 让 size - 1：
public int removeLast() {
int returnItem = items[size - 1]; // 取出最后一个元素
size -= 1; // 直接让 size 减 1，相当于忽略最后的元素
return returnItem;
}
📌 问题 这样虽然 逻辑上删除了元素，但 数据其实还在内存里，只是 程序不再访问它。
如何真正释放内存?
在 Java 里，只有当一个对象没有引用时，垃圾回收器（GC）才会回收它。 如果 items 存的不是整数，而是对象，比如 String，那么 删除元素后，原来的对象还是在内存里，浪费空间!
改进代码
public Glorp deleteBack() {
Glorp returnItem = getBack();
items[size - 1] = null; // 释放对象的引用，避免占用内存
size -= 1;
return returnItem;
}
📌 为什么要 null 掉?
Java 不会自动清理数组里的对象，所以需要手动 null 掉，不然会 浪费内存。
这种 没用但还占内存的引用 叫做 "loitering"（游离）。

🔷 AList 的不变性（Invariants）
不变性（Invariants）指的是，在程序运行的任何时间点，这些条件必须保持为真，否则代码逻辑可能存在问题。
新元素插入的位置始终是 size
items[size] = x; 表示新元素永远存储在 size 指定的位置。
size 永远是当前列表中的元素个数
size 记录的是实际存储的元素个数，而不是数组的长度。
列表的最后一个元素始终存储在 size - 1 位置
getLast() 通过 return items[size - 1]; 获取最后一个元素。
🔷 代码功能解析
方法 作用 返回值
AList() 初始化一个大小为 100 的整数数组 -
addLast(int x) 在 size 位置添加元素 x，并更新 size -
getLast() 获取最后一个元素，即 items[size - 1] int（最后一个元素）
get(int i) 获取索引 i 处的元素 int（items[i]）
size() 获取当前元素个数 int（size）
🔷 代码的局限性
容量固定
数组大小被初始化为 100，如果 size 超过 100，就会数组越界。
解决方案：当 size 达到 items.length 时，创建一个更大的数组并复制数据（即 动态扩容）。
没有删除元素的方法
目前 AList 只有添加、获取和查询大小的方法，没有删除元素的方法。
解决方案：可以实现 removeLast()，删除最后一个元素，并减少 size。
AList Invariants:
● The position of the next item to be inserted is always size.
● size is always the number of items in the AList.
● The last item in the list is always in position size - 1.
🔷 removeLast() 的作用
当调用 removeLast() 时：
减少 size 的值
size 记录的是列表中元素的个数，删除最后一个元素后，size 应该减少 1。
最后一个元素仍然留在数组中
数组的大小 不会改变，只是 size 变小了。
这意味着删除的元素可能仍然在内存中，但不会再被访问（逻辑上已经被删除）。
✔ 只有 size 发生变化，数组内容仍然存在，但逻辑上最后一个元素被忽略。

## 🔥 AList 是什么?

AList 就是 动态数组，它和 Java 自带的 ArrayList 很像。
简单来说，它是 自己实现的数组类，可以：
存储数据
动态扩容
支持增删查操作

```
private void resize(int capacity) {
int[] a = new int[capacity];
System.arraycopy(items, 0, a, 0, size);
items = a;
}
public void addLast(int x) {
if (size == items.length) {
resize(size + 1);
}
items[size] = x;
size += 1;
}
```

📌 Array Resizing（数组扩容）
🔷 为什么要扩容?
在 AList（基于数组的列表）中，数组的长度是固定的。
当 size == items.length（数组满了）时，再 addLast(x) 会导致 数组越界错误（IndexOutOfBoundsException）。
解决方案：创建更大的数组，然后 把原来的数据复制过去。
🔷 扩容的实现方法
当 size == items.length 时：
创建一个新数组，长度为 size + 1
java
Copy code
int[] a = new int[size + 1];
这里的 size + 1 只是个示例，实际上可以是 2 × size（更常见的策略）。
把旧数组 items 的内容复制到 a
System.arraycopy(items, 0, a, 0, size);
System.arraycopy(src, srcPos, dest, destPos, length);
这个方法会高效地复制 items 的所有元素到 a，避免 for 循环。
把 items 指向新数组
items = a;
旧的 items 被丢弃，新的 items 现在有更大的容量。
🔷 Java 代码示例
💡 正确的扩容方式
private void resize(int newCapacity) {
int[] a = new int[newCapacity]; // 创建一个更大的数组
System.arraycopy(items, 0, a, 0, size); // 复制旧数据
items = a; // 更新 items 指向新数组
}
public void addLast(int x) {
if (size == items.length) {
resize(size *2); // 扩容为原来的 2 倍*
}
*items[size] = x;*
*size += 1;*
*}*
*扩容倍数通常是 size* 2，避免频繁扩容。
System.arraycopy 比 for 循环更高效。
🔷 关键点总结
✅ 数组是定长的，所以 AList 需要扩容机制。
✅ 扩容通常是 size * 2，这样避免频繁分配内存。
✅ 使用 System.arraycopy 复制数组，比 for 循环快。
✅ 扩容后，items 指向新数组，旧数组被丢弃。

数组（Array）在扩容（Resizing）时的性能问题，并通过图表对比了两种数据结构（SLLList 和 AList）在插入数据时的性能差异。
1️⃣ 插入大量数据时的性能问题
插入 100,000 个元素时，导致大约 50 亿次对象创建。
计算机的运行速度虽然在 GHz 级别（每秒可执行数十亿次操作），但插入 100,000 个元素 依然会导致明显的性能开销，所以 可能需要几秒钟。
2️⃣ 右侧的两张图表：对比不同数据结构的插入性能
（左图）SLLList（单链表）—— 插入时间线性增长
横轴（X 轴）：插入的元素数量。
纵轴（Y 轴）：插入所需的时间。
线性增长（直线）：SLLList 使用 addFirst() 操作，每次插入都只需修改一个指针，所以时间复杂度是 $O(1)$。
（右图）AList（动态数组）—— 插入时间呈指数增长
横轴（X 轴）：插入的元素数量。
纵轴（Y 轴）：插入所需的时间。
指数增长（曲线陡增）：AList 是一个数组列表，当数组满了，需要 创建更大的数组，并拷贝所有元素，导致时间复杂度高于 $O(1)$，通常是 摊还 $O(n)$。

📌 几何扩容 vs 线性扩容
🚀 1️⃣ 线性扩容（Unusably Bad：非常糟糕的扩容方式）
public void addLast(int x) {
if (size == items.length) {
resize(size + RFACTOR);
}
items[size] = x;
size += 1;
}
🔴 为什么这个方法很糟糕?
扩容时只增加一个固定量（RFACTOR），导致：
每次插入新元素时，数组可能需要频繁扩容。
大量的数组拷贝操作（System.arraycopy），极大降低效率。
时间复杂度退化：最坏情况下，插入 n 个元素的时间复杂度为 $O(n^2)$，因为每次扩容都要复制数组。
🔥 2️⃣ 几何扩容（Great Performance：更快的扩容方式）
public void addLast(int x) {
if (size == items.length) {
resize(size *RFACTOR);*
*}*
*items[size] = x;*
*size += 1;*
*}*
✅ *为什么这个方法更好?*
*扩容时，数组大小按倍数（RFACTOR）增长，比如 size* 2 或 size * 1.5。
减少扩容的次数：
比如，size = 10，如果 RFACTOR = 2：
变 10 → 20 → 40 → 80，扩容次数大大减少!
相比之下，线性扩容 size + 10，需要扩容 10 → 20 → 30 → 40 → 50，次数明显增多!
摊还时间复杂度（Amortized Complexity）降低：
几何扩容摊还时间复杂度为 $O(1)$，意味着在大多数情况下，插入操作是 常数时间。
线性扩容则是 $O(n)$，因为每次扩容都要移动大量元素。

📌 Memory Efficiency（内存效率）
AList（动态数组）在 时间效率 之外，还需要 空间效率，并介绍了一种优化策略 "Usage Ratio"（使用率） 来决定何时缩小数组。
$R = \frac{\text{size}}{\text{items.length}}$

- `size` 是当前数组中的元素个数。
- `items.length` 是数组的总容量。
  如果在 **R = 0.5 时就缩小数组**，可能会导致 **频繁的扩容和缩小**，从而影响性能。
- 例如：
- **插入 50% 容量时扩容**，但又 **删除 50% 时缩容**，会不断重复这个过程。
- **使用 R = 0.25，可以有效减少这种频繁的 resize 操作**。