2° 异常情况处理

以add(newEntry)为例，有错可能是因为 { 先前创建包无效 / newEntry为无效对象，若检测到想处理 ⇒ { do "no op" / 返回 false布尔值 / 抛出异常

Bag是简单的ADT

例. 用包实现：生成一些随机数，计算每种数生成多少个

以实施者思考：数据表示 <=> 操作实现

- 数组不错 private T [ ] bag ;（可以储多个值并操作）但：一旦创建便固定，包需要改变

  引入一个变量 private int numberOfEntries;

包的 Physical size：数组的位置数

包的 Logical size：包中存储的元素数； ↓ 该变量保持逻辑尺寸

∴ 把元素加入包不一定改变物理尺寸，但改变逻辑尺寸

? 物理尺寸用完了怎么办 — 采用固定大小 1° / 动态调整大小 2°

1° Fixed Size Array

- Physical size 由形参（数组长度）传递
- Logical Size 由 numberOfEntries 变量维护

优点：易于实现

缺点：① 程序员防止用完浪费空间 ② 使用者用完

- 创建后 Physical size 不变
- 一旦填满（LS==PS），不能加元素

```java
public boolean add(T newEntry)
{
    checkIntegrity();
    boolean result = true;
    if (isArrayFull())      //无空间不能加
    {
        result = false;
    }
    else
    {   //加
        bag[numberOfEntries] = newEntry;
        numberOfEntries++;
    }
    return result;
}
```

⇐ 简单例子

```java
public boolean remove(T anEntry)   //从包中删除指定条目，成功返回true
{
    checkIntegrity();
    int index = getIndexOf(anEntry);   // → 简单按序查找
    T result = removeEntry(index);
    return anEntry.equals(result);     // ↓ 如何处理空参数？
}
```

```java
private int getIndexOf(T anEntry)   //private方法，隐于接口
{
    int where = -1;
    boolean found = false;
    int index = 0;
    while (!found && (index < numberOfEntries))   //循环条件：没找到项目且未到数组尾
    {
        if (anEntry.equals(bag[index]))
        {
            found = true;
            where = index;
        } // end if
        index++;
    } // end while
    return where;
}
```

找到为true，位置，找不到index=-1

```java
private T removeEntry(int givenIndex)
{
      T result = null;

      if (!isEmpty() && (givenIndex >= 0))
      {
         result = bag[givenIndex];
         int lastIndex = numberOfEntries - 1;
         bag[givenIndex] = bag[lastIndex];

         bag[lastIndex] = null;

         numberOfEntries--;

      }
     return result;
}
```

移除条目 //测试index是否有效

//从最后一个条目转换给定条目

//从最后一个条目去除引用

```java
public boolean remove(T anEntry)
{
        checkIntegrity();
        int index = getIndexOf(anEntry);
        T result = removeEntry(index);
        return anEntry.equals(result);
}
```

//获取移除条目位置

检查完整性

//移除

Lecture 6

2° Dynamic Size Array

想法：创建时数组大小就固定，一旦填满，创建更大新数组，复制数据，重分配其为工作数组

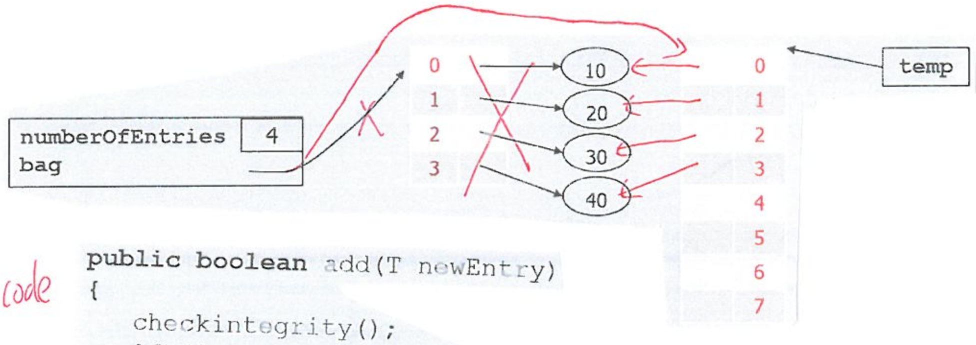
2.1制作多大的？新数组一般是旧数组的2倍

2.2如何复制？copy index by index — 复制的只是reference，对象没变

2.3旧数组被垃圾回收

流程
1. 创建2倍大小新数组
2. 复制数据（引用）
3. reassign bag reference
4. 垃圾回收旧数组
5. 方法结束时temp消失

⇒ code



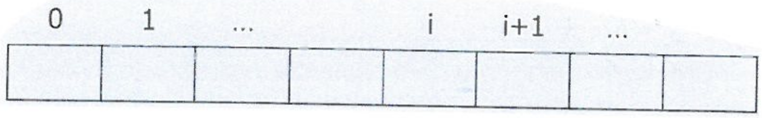

```java
public boolean add(T newEntry)
{
    checkintegrity();
    if (isArrayFull())
    {    doubleCapacity();    }
    bag[numberOfEntries] = newEntry;
    numberOfEntries++;
    return true;
}
```

未改变

```java
private void doubleCapacity()
{
   int newLength = 2 * bag.length;
   checkCapacity(newLength);
   bag = Arrays.copyOf(bag, 2 * bag.length);
}
```

//并非必需，确定尺寸

包的实现使用连续内存 → 内存中位置相邻



好处：— 用偏移量从第一个推出其它位置

— 直接访问单独元素（访问A[i]）

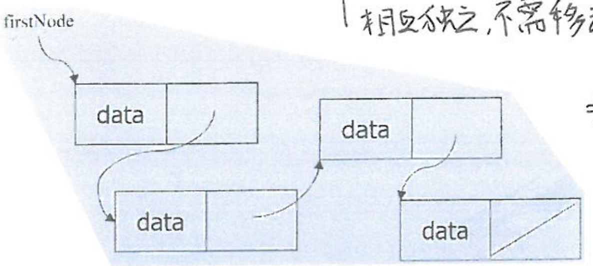
— 直接访问允许使用例如Binary Search的高效算法

缺点：— 内存分配必须立即完成 →（太大、太小、复制耗时）

— 在数组中间插入&删除元素需要移动其他元素

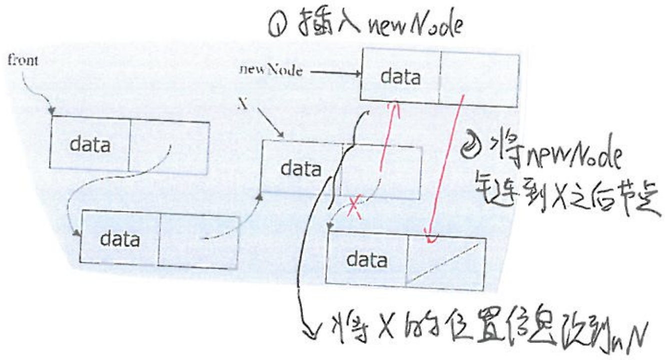
解决缺点？ —将内存分成小的、独立的片段，与集合中的元素对应

⇓ { 精准分配
相互独立，不需移动

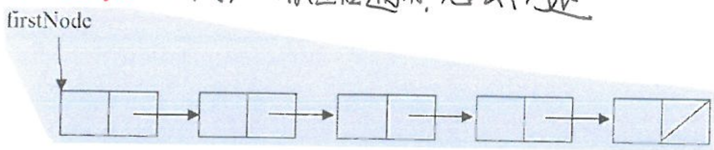


⇒用链表的思想追踪 { 一部分储存数据
另一部分储存下一片的位置信息

用该思想处理连续内存 ⟶



单链表：单向，从前往后遍历，无法倒退



另：—双链表
每个节点有两个链接(前、后)
双向

—循环链表
反向: back to front

链表的实现— self-referential data type 自引用

```
class Node
{
    private T data;
    private Node next;   // link的名字
    ...
```

//实现包接口方法

```
public class LinkedBag<T> implements BagInterface<T>
{
   private Node firstNode;
   private int numberOfEntries;
   ...
   private class Node                                  //Node
   {
      private T data;                                   // data
      private Node next;                                // next   //数据、下节点信息

      private Node(T dataPortion)                       // Node为private inner类
      { this(dataPortion, null);  }

      private Node(T dataPortion, Node nextNode)
      { data = dataPortion;  next = nextNode; }
   }
   ...
}  //LinkedBag类
```

Node在LinkedBag中声明，LinkedBag的方法可以访问Node类

⇓ —LinkedBag需能访问节点的data & next
—Node中编写accessor与mutator便可以

只有在LinkedBag类内才能这么做

↓LinkedBag外无法使用Node为缺点

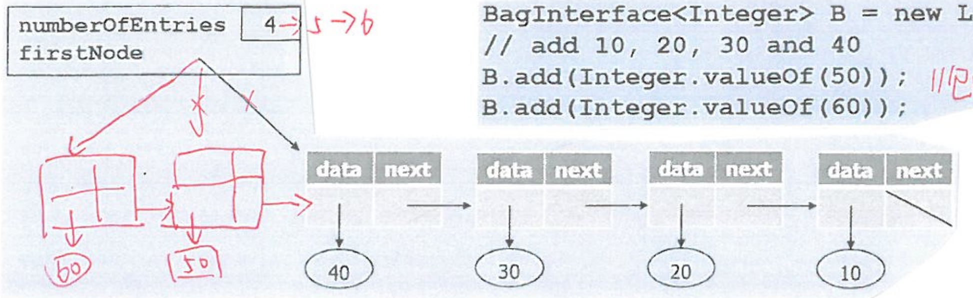
Lecture 7

一些BagInterface方法
例：插入元素

```
public boolean add (T newEntry)
{
   Node newNode = new Node(newEntry);   //创建新Node
   newNode.next = firstNode;            //将该新节点链至前端
   firstNode = newNode;                 //set front to new Node
   numberofEntries++;                   //增加条目

   return true;
}
```

↗不排序，加在front容易

```
BagInterface<Integer> B = new LinkedBag<Integer>();
// add 10, 20, 30 and 40
B.add(Integer.valueOf(50));   //包内加入
B.add(Integer.valueOf(60));
```



创建新节点，放入newEntry
新节点变成front of list

∴只能插入在front of bag

—特殊情况—空包(firstNode is null)

寻找特殊情况 — contains()方法

— sequential search //顺序搜索
(从头开始找，直至找到或末尾)

判断是否含有元素

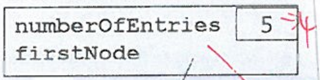
```
public boolean contains(T anEntry)   //判断T中是否含有anEntry
{
      boolean found = false;
      Node currentNode = firstNode;   //从列表头开始找

      while (!found && (currentNode != null))
      {
            if (anEntry.equals(currentNode.data))   //找着了
                  found = true;
            else
                  currentNode = currentNode.next;   //没找到，下一个
      }   //结束while loop
      return found;
}   //结束contains
```

另一个操作：取出

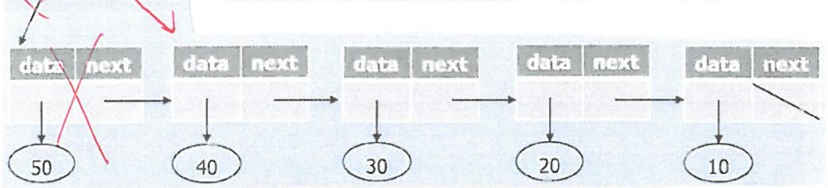
public boolean remove(T anEntry)

—找到条目 < 返回布尔值
reference to actual node

—移除—取消链接该节点




```
firstNode = firstNode.next;
numberOfEntries--;   //删前节点
```



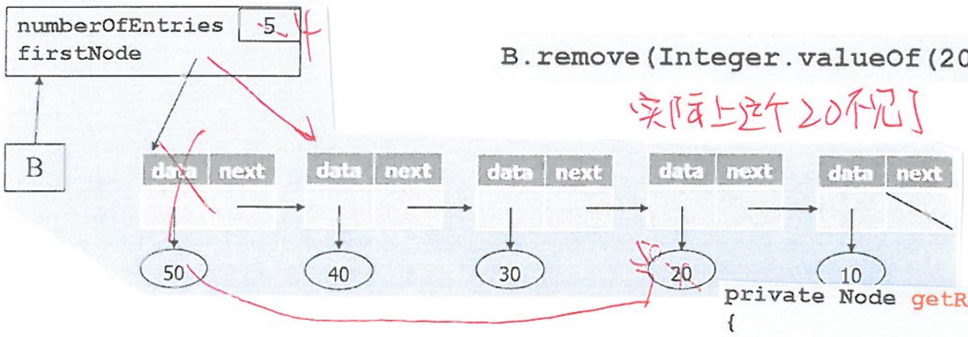
因为front Node容易删，所以将front Node数据给想删的Node，删掉front就行

Node不是数据，是访问数据的指针

⇓

```
B.remove(Integer.valueOf(20));
```

实际上这个20不见了



找到20的reference
把50复制过去
删除最前面的50

```
private Node getReferenceTo(T anEntry)
{
      boolean found = false;
      Node currentNode = firstNode;

      while (!found && (currentNode != null))
      {
            if (anEntry.equals(currentNode.data))
                  found = true;
            else
                  currentNode = currentNode.next;
      }
      return currentNode;   ——返回值不同
}
```

```java
public boolean remove(T anEntry)
{
        boolean result = false;
        Node nodeN = getReferenceTo(anEntry); //获得 anEntry的Reference.
        if (nodeN != null)
        {
           nodeN.data = firstNode.data;
           firstNode = firstNode.next;   //从第一节点复制数据,删除第一节点.
           numberOfEntries--;
           result = true;
        } //
       return result; //操作成功返回true
}
```

先找到要删除的节点的ref.

⇒ LinkedBag 删除 first item
（order 不重要）

Node类作为单独的(非内部)类
可以使Node reuse

```java
public class Node<T>
{
  private T         data; //数据部分
  private Node<T> next; //链接到下一个节点.
  ...
```

此时 LinkedList 中有
private Node<T> firstNode;

Node<T> class 内应定义一些方法来访问 next & data

{ accessor 例: getData(), getNextNode()
  mutator 例: setData(), setNextNode()

⇒ 新的 remove() 方法

```java
public boolean remove(T anEntry) // remove()用了Node<T>的版本
{
        boolean result = false;
        Node<T> nodeN = getReferenceTo(anEntry);
        if (nodeN != null) //获得anEntry的Reference
        {
           nodeN.setData(firstNode.getData());
           firstNode = firstNode.getNextNode();  //获取一节的数据

           numberOfEntries--;
           result = true;
        }
        return result;
}
```

另一个ADT: List

数据: — 按特定顺序排列且数据类型相同的对象的集合
（数据没有排序）
— 集合中对象的数量

操作: { - add(newEntry)
- add(newPosition, newEntry)
- remove(givenPosition)
- clear()
- replace(givenPosition, newEntry)
- getEntry(givenPosition)
- toArray()
- contains(anEntry)
- getLength()
- isEmpty()

另: 从1起头而不是0

List作为有效结构,有以下例功能 { Last in First out 行为
First in Last out 行为
通过索引访问数据,在指定位置插入,改变,删除数据
在list中搜索一个元素

如何把List当包用?-需要加入包的方法
但并不一定十分合适

Lecture 8

List的实现 { data — 表示对象及逻辑大小的集合
operation — 与数据相关

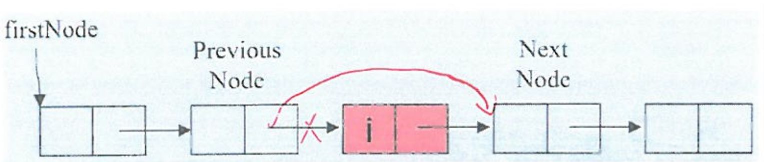
用链接的数据结构实现ListInterface

与LinkedBag的相同点 {
- 单链表结构
- 包含data & next的Node inner class
- 在front添加元素
- 查找元素

重要不同点: ListInterface要求数据维持 positional order
- 不能随意移动数据

public T remove(int givenposition) 为例

步骤 ① 找到位于givenposition(i)的对象
- 使用getNodeat()方法
② remove (重链接)



getNodeAt() 方法 (任务目标为返回给定位置的节点的引用
条件: 列表不空,给定位置合理)

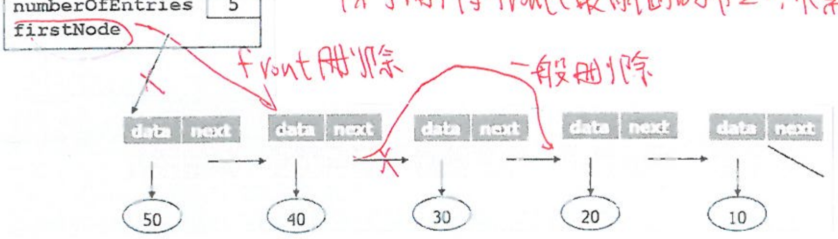
```
private Node getNodeAt(int givenPosition)
{
    Node currentNode = firstNode; //定到初始

    for (int counter = 1; counter < givenPosition; counter++) //遍历列表以定位所需节点
        currentNode = currentNode.getNextNode();

    return currentNode;
}
```

currentNode这变量是一直走的。

这是个private方法 (满足precon时才能调用)

ListInterface如果index无效,会删除front value,发生 special case → 导致firstNode的改变

除了删除front(最前面的节点),不需要改变firstNode



一般是将before链到next.
而front无 before node,因此改变 firstNode

```
public T remove(int givenPosition) //移除给定位置的值
{
   T result = null; //初始化返回值
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries)) //满足precon - 给定位置合理
   {                                                               - 列表不空.
      if (givenPosition == 1) // 删除的是第一个
      {
         result = firstNode.getData();
         firstNode = firstNode.getNextNode(); //改变firstNode的link,至下一个节点.
      }
      else // 常规操作
      {
         Node nodeBefore = getNodeAt(givenPosition-1); //找到前节点.
         Node nodeToRemove = nodeBefore.getNextNode(); // 前节点后的是删除节点.
         result = nodeToRemove.getData(); //删
         Node nodeAfter = nodeToRemove.getNextNode(); //删节点后一个为后节点
         nodeBefore.setNextNode(nodeAfter); //前节点与后节点相连link
      } // end if 小的循环
      numberOfEntries--;
      return result;
   } // end if 大的循环
   else throw new IndexOutOfBoundsException("Illegal Index"); //抛出无效index
}
```

Special Case: First Node

Normal Case

Special Case: Invalid Index

First & Last Reference

我们可以添加一个lastNode变量，从而在end插入Node而不需遍历整个list

? 当出现特殊情况 { remove last node
remove only node

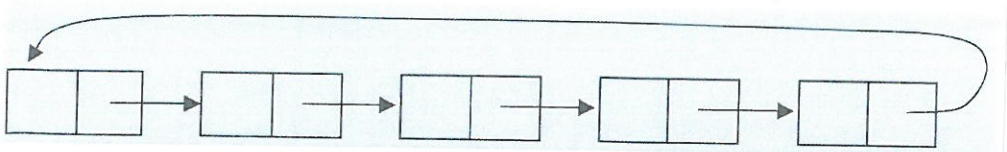
```
public T remove(int givenPosition) //有lastNode的版本
{
   T result = null; //初始化返回值
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries)) //满足precon
   {
      if (givenPosition == 1)
      {
         result = firstNode.getData();
         firstNode = firstNode.getNextNode();
         if (numberOfEntries == 1)
            lastNode = null;
      }
      else
      {
         Node nodeBefore = getNodeAt(givenPosition-1);
         Node nodeToRemove = nodeBefore.getNextNode();
         Node nodeAfter = nodeToRemove.getNextNode();
         nodeBefore.setNextNode(nodeAfter);
         result = nodeToRemove.getData();
         if (givenPosition == numberOfEntries)
            lastNode = nodeBefore;
      } // end if
      numberOfEntries--;
   } // end if
   else throw new IndexOutOfBoundsException("Illegal Index");

   return result;
}
```

⇐ { 删除only node
它本质删除了front node. (因为只有一个node!!).

⇐ { 删除last node.
切换lastNode

Circular Linked List 循环链表

last node连回front node



适用于Queue队列

Doubly Linked List 双链表

每个node都除指向 previous 和 next

可以从两个方向遍历列表 (例如用于Deque-双向队列)

↓引用front或end

Lecture 9

Array Implementation for List

好处: 一存储多个值进行不同操作 private T[] list;

一追踪逻辑尺寸(几个对象): private int numberOfEntries;

一动态调整

以add方法为例: public void add(int newPosition, T newEntry)

如何在任意索引处添加元素

— direct access that index

— shift => 为新元素腾出空间

Note: 数组的第0index不会使用

一它列出的是 [numberOfEntries]

```
public void add(int givenPosition, T newEntry)
{
        checkIntegrity(); //完整性检验, ADT都有这步
        if ((givenPosition >= 1) &&
            (givenPosition <= numberOfEntries + 1)) //givenPosition满足大条件
        {
            if (givenPosition <= numberOfEntries) //满足小条件, 挪位
                makeRoom(givenPosition);
            list[givenPosition] = newEntry; //把newEntry放进去
            numberOfEntries++; //条目数增加
            ensureCapacity();
        }
        else
            throw new IndexOutOfBoundsException("Error"); //在最后一位, 没法make room
}
```

ADT的local变量

makeRoom() 方法的工作 — shifting算法 ← correct side很重要

↓wrong side导致复制

这是private方法

```
private void makeRoom(int givenPosition)
{
    //givenPosition满足了之前的条件
    //把每一个条目都移到下一个更高的index, 从end开始, 直到位于newPosition的条目移了
    int newIndex = givenPosition;  int lastIndex = numberOfEntries;
    for (int index = lastIndex; index >= newIndex; index--)
        list[index+1] = list[index];
} // end makeRoom
```

关于删除数据?: public T remove(int givenPosition)

一我们的做法与add相反
- 移除元素
- 旁边条目移过去以填补空白
- numberOfEntries递减

```
public T remove(int givenPosition)
{
      checkIntegrity();
      if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))  // 满足位置条件
      {

            T result = list[givenPosition];  // 获取要删除的项目


            if (givenPosition < numberOfEntries)  // 后续项目物向删除位(非last)
                  removeGap(givenPosition);
            list[numberOfEntries] = null;  // 删除操作
            numberOfEntries--;
            return result;
      }
      else
            throw new IndexOutOfBoundsException("Error");  // 否则抛出Error.
} // end remove
```

removeGap()算法

```
private void removeGap(int givenPosition)
{
      int removedIndex = givenPosition;  // 移除索引为给定位置
      int lastIndex = numberOfEntries;  // 最后索引为项目数
      for (int index = removedIndex; index < lastIndex; index++)  // 把该条目之后的条目们都向前移到更低位
            list[index] = list[index+1];  // 移动位
} // end removeGap
```

List

标准List的实现：ArrayList是一个标准类(动态扩展,实现List Interface)

- Vector类 > public E remove(int index)
- LinkedList 双向链表,也能实现Queue双队列

算法效率

例：顺序搜索N次,而二叉搜索只需$\log_2 N$次. 样本大时差距很大.

因此执行前分析算法

可以通过已实现的算法经验预估,更重要 Asymptotic Analysis.

1) 用关键指令控制算法整体运行时间

例：排序算法本质是彼此的比较,运行时间与比较的次数成正比

2) 用公式表示关键指令的数量如何增加 => variable N表主问题.

- Worst Time：处理问题最多用多少关键指令?
- Average Time：...、平均用多少...?

也例：排序算法的最差情况：$F(N) = (N^2/2) - (N/2)$, 运行时间为$O(N^2)$ 以数量级计算

忽略常数与低数量级

Lecture 10

asymptotic analysis的简例

1° 恒定时间O(1)，与N相独立

```
Y = X;
Z = X + Y;
```

2° 线性时间O(N)，与N增长速率相同

```
for (int i = 0; i < N; i++)
    do_some_constant_time_operation;
```

做N次操作

3° 平方时间$O(N^2)$ 做N×N次操作

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
       do_some_constant_time_op;
```

以sorting算法为例(key操作加比较)

//顺序搜索 O(N) 最多迭代N次

二叉搜索 $O(\log_2 N)$

每次切减两半，$N=2^k$迭代

```
//在[]a查找对象key
public static int binarySearch(Object[] a, Object key) {
        int low = 0;
        int high = a.length-1;

        while (low <= high) {
            int mid =(low + high)/2;
            Object midVal = a[mid];
            int cmp = ((Comparable)midVal).compareTo(key);
            if (cmp < 0)
                low = mid + 1;
            else if (cmp > 0)
                high = mid - 1;
            else
                return mid;  //找着
        }
        return -(low + 1);  //没找着 → worst case
    }
```

另：add(newEntry)

Runtime for resizable array：①分配一个新数组，把数据复制过去，O(N)．②到最后插入，O(1)

加起来为O(N)．

但有时可直接add，只需O(1)．

↓

Amortized Time 均摊耗时间：一系列操作的平均时间

对于动态数组，调整大小有2种方案 {①每次尺寸+1 ②每次尺寸double（通常）

①每次把尺寸+1

insert i需要assign i items =>O(N)

需N次add()操作．

总操作：$1+2+\cdots+(N-1+N = N(N+1)/2 \Rightarrow O(N^2)$ N次．

平均时间则为$O(N^2)/N = O(N)$ 一次操作的时间．

for ResizableArrayBag