另一个方法：double the size　　实现步骤：

```
// Assume A is as shown
int [] temp = new int[2 * A.length];
for (int i = 0; i < A.length; i++)
    temp[i] = A[i];
A = temp;
```

temp

A

i	value
0	10
1	20
2	30
3	40

i	value
0	10
1	20
2	30
3	40
4	0
5	0
6	0
7	0

- When we are finished, "old" array is garbage – no longer accessible

建立新数组

复制

让A指向temp所指的更长的array [reassign the reference]

keep track of the number of locations that are actually being used in the array

[追踪数组中实际使用的位置数] —用数组数据之外的容器外变量

数组的大小与数组中存储的元素数量不一定相同

physical size　　logical size

另：ArrayList类

关于数组的通用操作：添加对象至末尾 / 找到对象 / 遍历数组，有一个通用类 ArrayList

~~(由于Data Abstraction，我们不需知道很详细)~~

ArrayList的实现方法

—数据分为两个部分：an array 实际存储信息；an int 追踪存储元素的数量

—大多数操作与 logical size 有关 —实际存储的元素数量.

—physical size 由视角抽象化而来

Note：将项目添加到ArrayList时，在添加引用，而不是复制数据.

二维数组：arrays of arrays

例：int [] [] A = new int [4] [8];

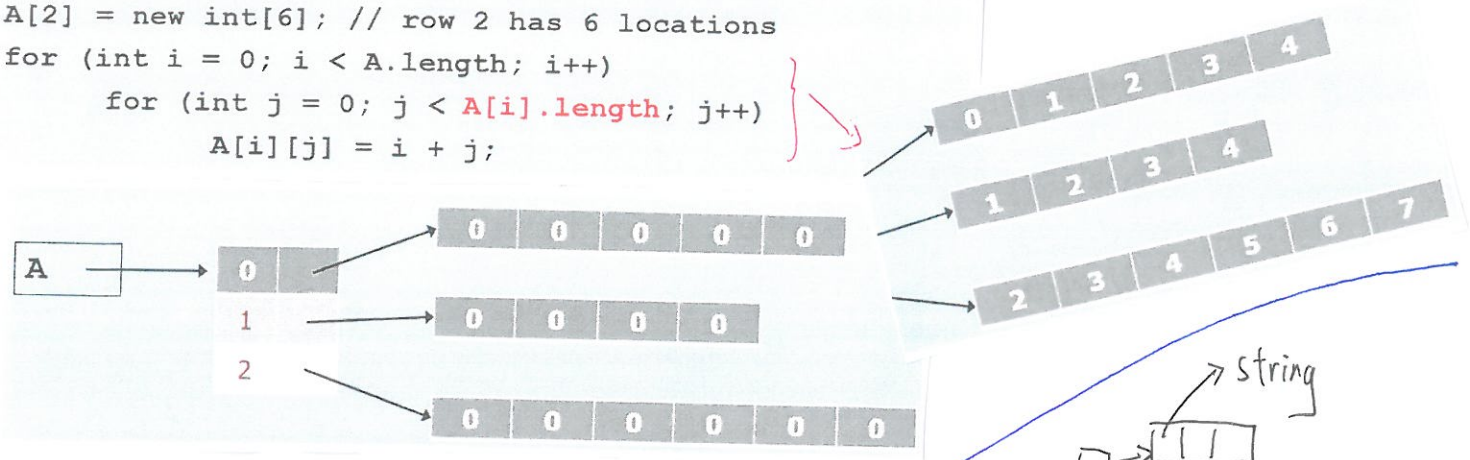
row行：an array of items　　column列：specific item within row

此时，为了遍历二维数组，使用nested loop

(column不一定长度相同

2D Array Example

```
int [][] A = new int[3][];  // no value in 2nd bracket
A[0] = new int[5]; // row 0 has 5 locations
A[1] = new int[4]; // row 1 has 4 locations
A[2] = new int[6]; // row 2 has 6 locations
for (int i = 0; i < A.length; i++)
      for (int j = 0; j < A[i].length; j++)
            A[i][j] = i + j;
```



可以有三级引用


```
String [][] A;
A = new String[4][];  // reference to 2-D array
A[0] = new String[3]; // reference to 1 row
A[0][0] = new String("Hello"); // reference to a String
```

Lecture 13

Sort 分类

给一个有N个元素的array

- sorted in ascending order (升序): A[i] < A[j], for all i < j
- sorted in descending order (降序): A[i] > A[j], for all i < j

一种排序方法: SelectionSort

0	1	2	3	4	5	6	7	Step
35	50	20	40	75	**10**	15	60	0
10	50	20	40	75	35	**15**	60	1
10	**15**	**20**	40	75	35	50	60	2
10	**15**	**20**	40	75	**35**	50	60	3
10	**15**	**20**	**35**	75	**40**	50	60	4
10	**15**	**20**	**35**	**40**	75	**50**	60	5
10	**15**	**20**	**35**	**40**	**50**	75	**60**	6
10	**15**	**20**	**35**	**40**	**50**	**60**	**75**	Finished

① 找到最小换入index 0
② 找到次小换入index 1
⋮
n-1个数，需要n-2步，index n-1自动精确。

通过 nested loop 实现 Selection Sort

外部循环，counter i go through positions from 0 to A.length-2
内部循环，counter j find next smallest's location, & swap into location i

Java Generics (泛型，可以sort其他类型的数据)

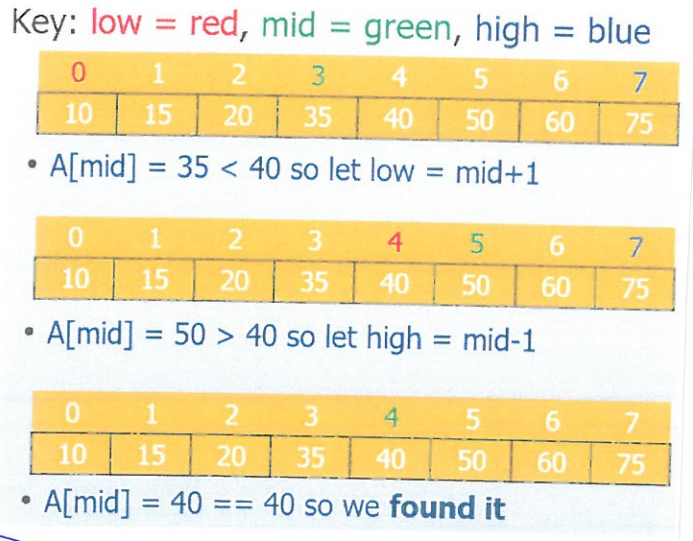
考虑 Sequential Search
- 最坏情况：查询每个元素 — linear run-time / 查询时间与项数成正比
- 当 data unsorted，只能用以上方法，

考虑 二叉搜索

找一个给定的k，猜一个数组中间值A[mid] => (low+high)/2

A[mid]==K，找到，任务完成
A[mid]<K，K在数组偏右方
A[mid]>K，K在数组偏左方

Ex：找到40
① 中值35<40，K在右边
② 更新，中值50>40，K在左边 } =)
③ 只剩40，找到



当一个元素不在数组内？(25)

① 中值35>25(K)，K在偏左方
此时 high=mid-1 最高值为原中值降一位

② 中值15<25(K)，K在偏右
high=mid+1，新最低值为原中值升一位

③ 中值20<25(K)，K在偏右
新最低值为原中值升一位.

④ 此时，low为35，high为20
无值

用while loop实现

Lecture 14

Java数组使用连续内存 (contiguous memory).

元素在内存中位置相邻

⇐ 给定第一个元素，根据偏移量能确定其它元素的位置

优点：① 直接访问单个元素 A[i]

② 直接访问允许高效算法，如 Binary Search

③ 数组更简单

缺点：① 内存的分配必须立即完成（太大或太小）。

② 插入或删除中间元素需要移动其它元素

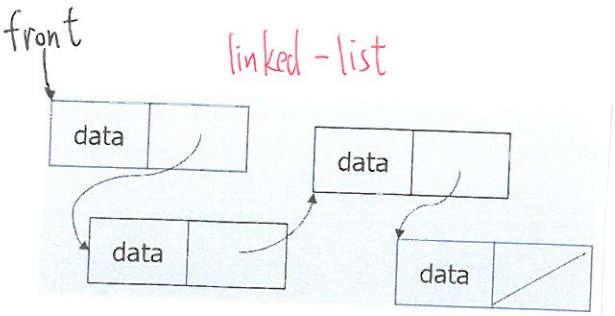
例

```
public boolean add(int loc, String val)
{
        if (loc >= 0 && loc <= size)
        {
                if (size == data.length)
                        resize(2 * size);
                for (int i = size; i > loc; i--)
                        data[i] = data[i-1];
                data[loc] = val;
                size++;
                return true;
        }
        return false;
}
```

必须移动旧值，以便为新项腾出空间。有时必须移动所有项

应对方法：将内存分配成小的独立的片段

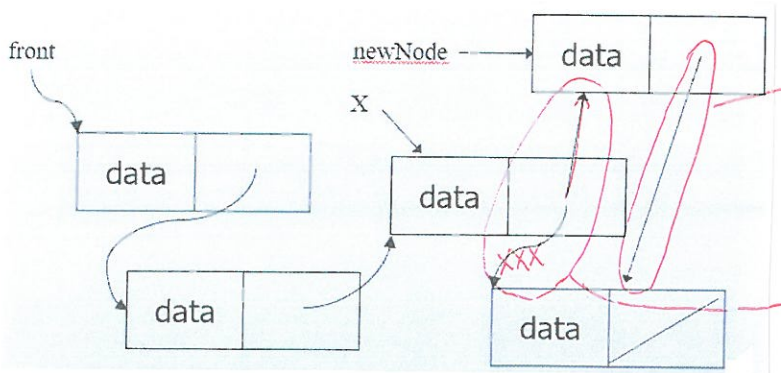
- 一部分用于存储数据
- 一部分用于存储下一个块的信息



实现方法在于知道列表头部 & 每个链接知道下个位置

一次一个链接，

任意位置链接，无需移动



Link the newNode to the node after X

Link next value for X to newNode

链表实现

data

store location of next link > self-referential data type [自引用类型]

```
public class SimpleLLList 仅在SLL内访问.
{
   private class Node
   {
       private T data;
       private Node next;
        ...
   }
   private Node front;  //
```

data type

Node是功能特定的Class

[节点]链表中链接的名称

不需详细了解.

在链表class内，Node为private inner class

- 在链表内被声明：private class Node
- 从而直接访问 next & data fields directly
- 链表外Node无用

SimpleAList & SimpleLList：有相同的功能

↓ 由基础数组实现

↓ 由基础链表实现

/把新值加进链表

①确定在何处放新值 ②为其创建新节点 ③将节点链到表里

Where to put new value?

1° public boolean add(String val)

- 在末尾添加新值
- 先遍历列表，再将新节点链至末尾

2° public boolean add(int loc, String val)

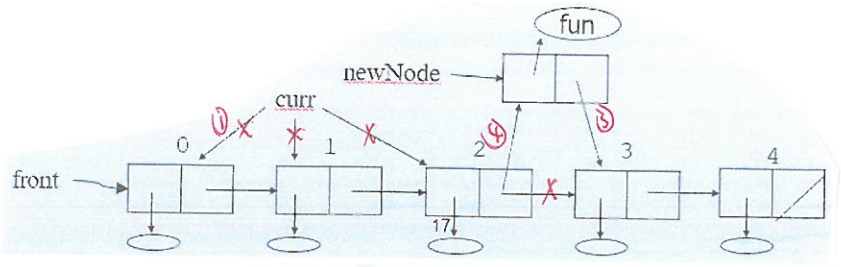
- 在任何位置(location loc)添加新值.

另法：添加在开头loc0，需change front variable

```
public boolean add(int loc, String val)
{
    if (loc >= 0 && loc <= size)  // make sure loc is legal
    {
        Node newNode = new Node(val);  // make new Node
        if (loc == 0)  // special case for front
        {
            newNode.next = front;  // link to OLD front
            front = newNode;   // make newNode new front
        }
        else
        {   // start at front and stop right before pos of
            Node curr = front;   // new node
            for (int i = 1; i < loc; i++)
                curr = curr.next;
            newNode.next = curr.next;  // link new Node
            curr.next = newNode;
        }
        size++;
        return true;
    }
    return false;
}
```

①②③④

位置为3，val为fun



两个要点：

1° 链表本质上是 sequential data structure [顺序数据结构].

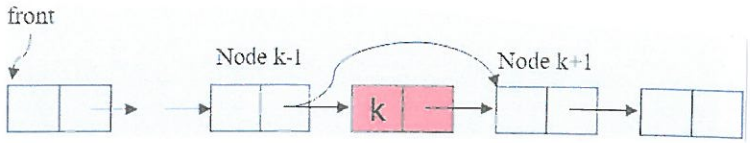
这意味着无法直接访问，想访k，必访 0, 1, ···, k-1

2° 必须保证节点正确链接

/ 在 loc_k 添加一个节点：到 loc_k-1；将要加的节点连到 loc_k 的前一个节点；

将节点 k-1 连到新节点.

/ ··· 删节点：



找到 k-1；将 k-1 与 k+1 连

(k仍旧存在，但已成为无用垃圾)

※ first node 与众不同.

change instance variable

另：next field of last node is null ⇒ null-terminated linked list

通过测试最后节点后面的引用来确定自己是否在尾.

Lecture 15

Static variables 静态变量：与类本身相关联的变量.

用class访问的方法：Class Name. variable Name

用对象访问的方法 (within object)：variable Name

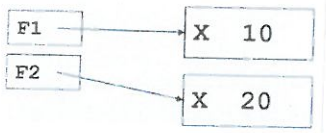
(outside object)：object Name. variable Name

人人类或者本对象外访问，数据应当public

```
public class Foo
{
    public int X;
    public static int Y;
}
```

```
// main
Foo F1 = new Foo();
F1.X = 10;
Foo F2 = new Foo();
F2.X = 20;
F1.Y = 30;
F2.Y = 40;
Foo.Y = 50;
```

=>



静态变量：通过 class 或 object 访问

对于静态变量，何时使用 variable，何时使用 method

1° Variable 用于储存对象的基本属性

可以通过 matutor 改变，但不应被废弃

2° Method 应当通过使用变量计算/确定值，已于能会变化

Copying Objects 复制对象

1° 使用 copy constructor 对类

采用相同类型的实参，复制对象.

例：String newString = newString(oldString);

2° 使用 clone 方法.

注意复制的具体内容

Shallow Copy：旧对象中的每个实例变量分别被给新对象中对应的实例变量
- 如果实例变量是对对象的引用，对象同样被共享。

Deep Copy：复制基本类型

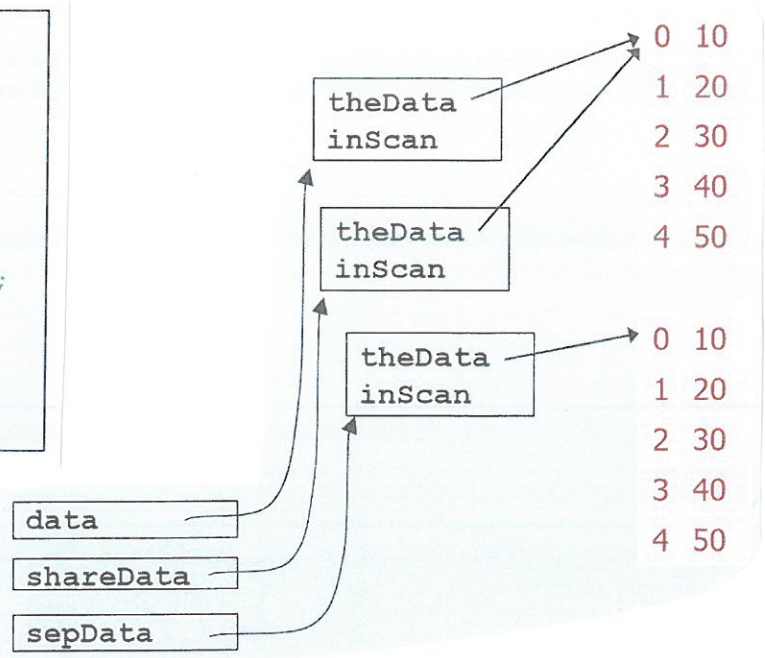
对于引用类型，引用会被分别被给，"follow the reference" & copy object.

```
public Scores(Scores oldOne, int flag)
{
    if (flag == 0)
    {
        theData = oldOne.theData;
        inScan = oldOne.inScan;
    }
    else
    {
        theData = new double[oldOne.theData.length];
        for (int i = 0; i < theData.length; i++)
            theData[i] = oldOne.theData[i];
        inScan = oldOne.inScan;
    }
}
```

```
// in main
Scores data = new Scores(num, scan);
// initialization and Scanner not shown - see figure

// Make shallow copy
Scores shareData = new Scores(data, 0);

// make deeper copy
Scores sepData = new Scores(data, 1);
```



Deep copy 更难实现 —

以 SimpleLList 为例 < Shallow copy 只复制 front and size variables [即只有一份节点列表]
>- The entire list of Nodes is shared by both copies.
Deeper copy 创造一组全新的节点，将数据复制到节点中

根据需求选择 shallow or deep

Returning reference from methods

一个方法只能返回一个值，但值可以是对包含任意数据的对象的引用

composition/aggregation — 一个对象可以包含对其他对象的引用

? 当实例变量返回一个对象内一个对象对其的引用

回到Original还是Copy?

Returning references to the original 更危险.

- 通过返回的引用修改对象,将影响调用方
- 导致数据无效,破坏封装

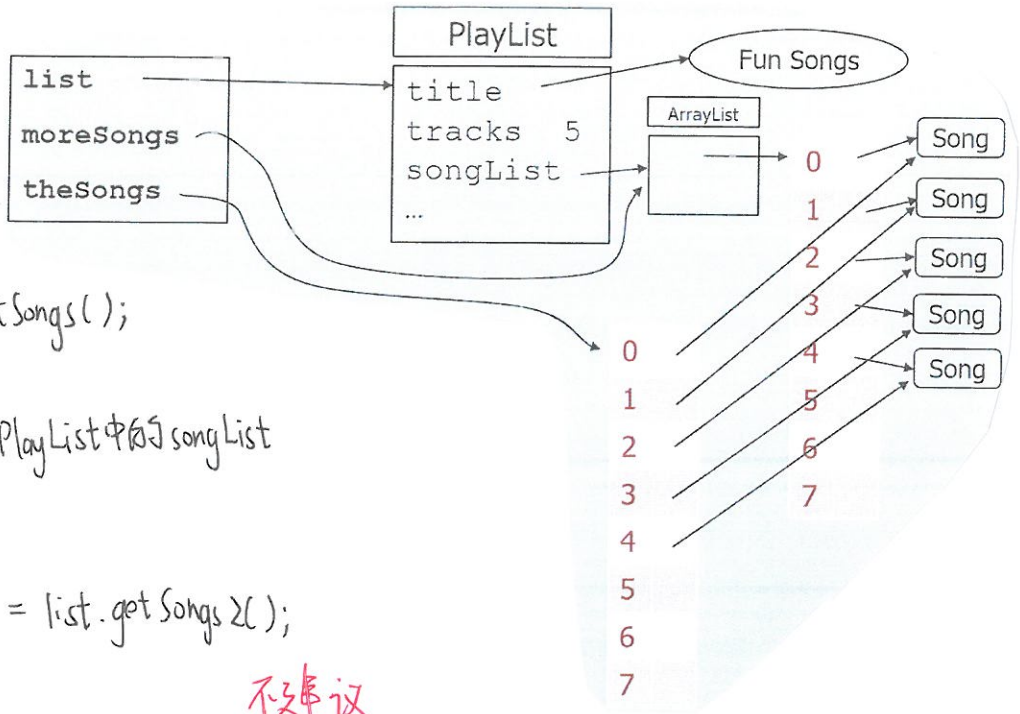
例. PlayList.class

Method getSongs() returns a copy(new array) containing Songs

Method getSongs2() returns a reference to the underlying ArrayList within PlayList

返回对 PlayList 中基础ArrayList的引用

(can +/- songs)




```
Song[] theSongs = list.getSongs();
//返回包含songs的新数组
//更改 theSongs数组不影响 PlayList中的 songList
//单个的Song对象仍旧被共享

ArrayList<Song> moreSongs = list.getSongs2();
//moreSongs使用相同数组
//更改PlayList 破坏封装
```

不建议.

this reference: 通常在实例方法中,同时访问实例变量和方法变量

- 方法变量与实例变量名称相同,更改方法变量

this: 伪实例变量,对对象的自引用

消除实例变量与方法变量之间的歧义

All Java classes have this

例

```
public class Foo
{
    private int X, Y;          // 实例
    public Foo(int X, int Y) // match instance var names   // 方法  与实例变量名称相匹配
    {
        this.X = X;    // disambiguates the vars
        this.Y = Y;
    }
```

消歧.

Garbage Collection
↓无其他引用的原始对象(无法访问)
清理内存

Java重要功能依赖于类/对象，但Java的基础类型不是类
例 ArrayList<T> 中T为reference type

Wrapper classes 处理这个问题
(包装器)：包装primitive values周围的对象，使其与其它类兼容

我们不能存储int，但能存储integer
↙ ↘
整型，数据类型　　整数，具体数字

任何Java primitive type都有对应的wrapper
例：Integer, Float, Double, Boolean (包装类)
例：i = new Integer(20); //生成一个对象
接受一个integer　↘ primitive type integer

Wrapper还提供 — Integer.parseInt()　将String转换为Int
　　static method
　　Character.isLetter()　测试字母是否为character

对wrapper class的算术运算
逻辑：k=i+j;
实际操作：k = new Integer(i.intValue() + j.intValue());
获取Integer对象的原始值，创建一个新的Integer 对象

Wrapper的一个功能：将字符串解析为primitive values
例：Double.parseDouble()

例：字符串"1234" 在ASCII码 (49 50 51 52 53)
转为int数据类型

除此，还能：大小写转换，字母&数字检查