# 包

数据结构(data structure)是使用一种程序设计语言实现的抽象数据类型(abstract data type🡪ADT)。

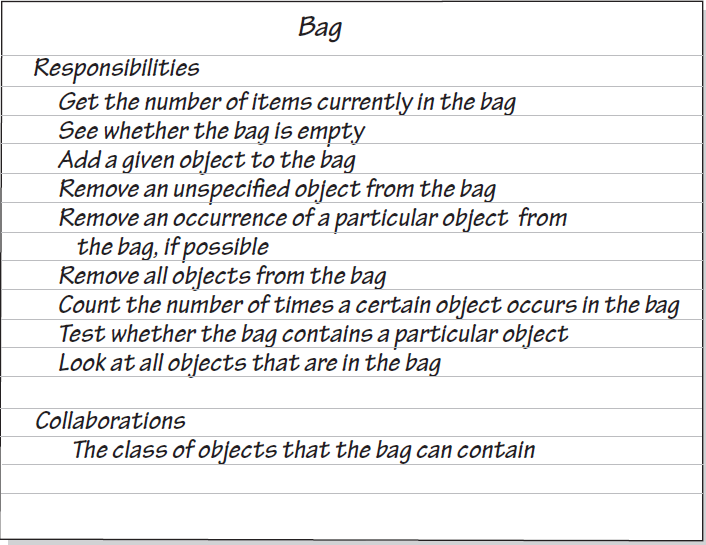
集合(collection)是一种对象，它将其他对象组成一组，并为它的客户提供不同的服务。ADT不一定是集合。

在Java中，容器(container)是一个对象，它的类派生于标准类Container。在Java中，不把包(bag)看作一种容器，而是一种集合。

包是没有特定次序的对象的有限集合。这些对象具有相同或相关的数据类型。包可以含有重复项。

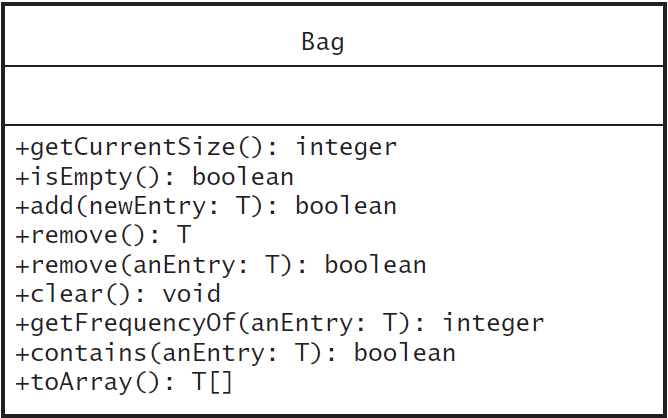
## 包的行为

得到当前包中的项数；查看包是否为空；将给定对象添加到包中；从包中删除一个未指定的对象；可能的话，从包中删除一个具体对象的一次出现；从包中删除所有对象。



包中的内容是无序的。另外，第一个删除操作只删除它能删除的任何对象。这个操作就像是伸手到袋子里把东西拿出来一样。而第二个删除操作是在包中查找某项。如果找到它，则拿出它。如果包中有多个相等的对象都满足你的查找条件，则删除其中的任意一个。如果在包中找不到该对象、则不能删除它，且要直接表示出来。最后一个删除操作只是清空包中的所有对象。

因为包是一个抽象数据类型，所以我们仅描述它的数据并说明它的操作。不指明如何保存数据或如何实现它的操作。



因为添加操作或者成功或者不成功，所以我们可以让方法add返回一个布尔值。标准Java接口Collection规定，如果添加没有成功则发生异常。显示一条错误信息并不是好的选择，因为你应该让客户决定所有的书面输出。

有3个动作涉及从包中删除项：删除所有的项；删除任意一项；删除某个项。从包中删除一项，则该方法可以简单地返回被删除的对象。如果包中不含有某项，可以让方法返回一个布尔值。定义一个方法来返回保存这些项的数组。

## 包类的Java接口

类接口不含有数据域、构造方法、私有方法或保护方法。



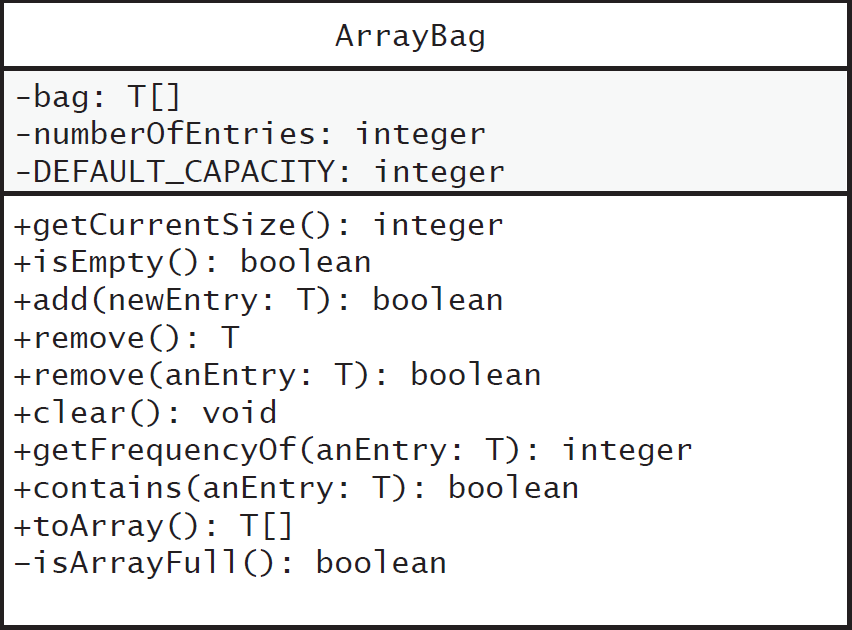
**package** DataStructuresAndAbstractions.Bag;  
  
**public interface** BagInterface<T>  
{  
 */\*\*  
 \* Gets the current number of entries in this bag.  
 \** ***@return*** *The integer number of entries currently in the bag.  
 \*/* **public int** getCurrentSize();  
  
 */\*\*  
 \* Sees whether this bag is empty.  
 \** ***@return*** *True if the bag is empty, or false if not.  
 \*/* **public boolean** isEmpty();  
  
 */\*\*  
 \* Adds a new entry to this bag.  
 \** ***@param newEntry*** *The object to be added as a new entry.  
 \** ***@return*** *True if the addition is successful, or false if not.  
 \*/* **public boolean** add(T newEntry);  
  
 */\*\*  
 \* Removes one unspecified entry from this bag, if possible.  
 \** ***@return*** *Either the removed entry, if the removal was successful, or null.  
 \*/* **public** T remove();  
  
 */\*\*  
 \* Removes one occurrence of a given entry from this bag.  
 \** ***@param anEntry*** *The entry to be removed.  
 \** ***@return*** *True if the removal was successful, or false if not.  
 \*/* **public boolean** remove(T anEntry);  
  
 */\*\*  
 \* Removes all entries from this bag.  
 \*/* **public void** clear();  
  
 */\*\*  
 \* Counts the number of times a given entry appears in this bag.  
 \** ***@param anEntry*** *The entry to be counted.  
 \** ***@return*** *The number of times anEntry appears in the bag.  
 \*/* **public int** getFrequencyOf(T anEntry);  
  
 */\*\*  
 \* Tests whether this bag contains a given entry.  
 \** ***@param anEntry*** *The entry to be tested.  
 \** ***@return*** *True if the bag contains anEntry, or false if not.  
 \*/* **public boolean** contains(T anEntry);  
  
 */\*\*  
 \* Retrieves all entries that are in this bag.  
 \** ***@return*** *A newly allocated array of all the entries in the bag.  
 \* Note: If the bag is empty, the returned array is empty.  
 \*/* **public** T[] toArray();  
}

## 使用ADT包

我们不需要知道程序员如何实现这个包，只需知道ADT包做什么就可以了。一旦我们选择了包中对象的数据类型，这个数据类型就包含在接口名后面的尖括号中。包中的所有项必须是这个数据类型或这个数据类型的子类型。如果是基本数据类型，则可以将对应的包装类的实例放入包中。例如，不是使用基本数据类型int的实例，而是使用包装类Integer的实例。

## 使用数组实现包

### 使用固定大小的数组实现ADT包



#### 核心方法



**package** DataStructuresAndAbstractions.Bag;  
  
*/\*\*  
 \* LISTING 2-1 An outline of the class ArrayBag  
 \** ***@param <T>*** *\*/***public final class** ArrayBag<T> **implements** BagInterface<T>  
{  
 **private final** T[] **bag**;  
 **private int numberOfEntries**;  
 **private static final int *DEFAULT\_CAPACITY*** = 25;  
  
 **public** ArrayBag()  
 {  
 **this**(***DEFAULT\_CAPACITY***);  
 }  
  
 **public** ArrayBag(**int** capacity)  
 {  
 *// bag = new T[capacity]; // SYNTAX ERROR  
 // 当分配数组时不能使用泛型* @SuppressWarnings(**"unchecked"**)  
 T[] tempBag = (T[]) **new** Object[capacity]; *// Unchecked cast* **bag** = tempBag;  
 **numberOfEntries** = 0;  
 }  
  
  
 */\*\*  
 \* Adds a new entry to this bag.  
 \** ***@param newEntry*** *The object to be added as a new entry.  
 \** ***@return*** *True if the addition is successful, or false if not.  
 \*/* @Override  
 **public boolean** add(T newEntry)  
 {  
 **boolean** result = **true**;  
  
 **if** (isArrayFull())  
 result = **false**;  
 **else** {  
 **bag**[**numberOfEntries**] = newEntry;  
 **numberOfEntries**++;  
 }  
  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Retrieves all entries that are in this bag.  
 \** ***@return*** *A newly allocated array of all the entries in this bag.  
 \*/* @Override  
 **public** T[] toArray()  
 {  
 *// The cast is safe because the new array contains null entries.* @SuppressWarnings(**"unchecked"**)  
 T[] result = (T[]) **new** Object[**numberOfEntries**]; *// Unchecked cast* **for** (**int** index = 0; index < **numberOfEntries**; index++)  
 result[index] = **bag**[index];  
  
 **return** result; *// 类不应该返回指向其私有数据域的数组的引用。  
 // return bag; // 直接返回bag* }  
  
  
 */\*\*  
 \* Test the bag whether is full.  
 \** ***@return*** *True if the bag is full.  
 \*/* **private boolean** isArrayFull()  
 {  
 **return numberOfEntries** >= **bag**.**length**;  
 }  
}

#### 让实现安全

当实现一个ADT时，需要注意两个问题：

·构造方法没有完全执行;

·客户试图创建一个其容量超出给定范围的包。

因此需在类中增加下列两个数据域：

**private boolean initialized** = **false**;  
**private static final int *MAX\_CAPACITY*** = 10000;



**package** DataStructuresAndAbstractions.Bag;  
  
**public final class** ArrayBag<T> **implements** BagInterface<T>  
{  
 **private final** T[] **bag**;  
 **private int numberOfEntries**;  
 **private static final int *DEFAULT\_CAPACITY*** = 25;  
  
 */\*  
 避免以下两个问题，修改构造方法  
 ·如果构造方法没有完全执行;  
 ·如果客户试图创建一个其容量超出给定范围的包  
 \*/* **private boolean initialized** = **false**;  
 **private static final int *MAX\_CAPACITY*** = 10000;  
  
 **public** ArrayBag()  
 {  
 **this**(***DEFAULT\_CAPACITY***);  
 }  
  
 **public** ArrayBag(**int** desiredCapacity)  
 {  
 **if** (desiredCapacity <= ***MAX\_CAPACITY***)  
 {  
 @SuppressWarnings(**"unchecked"**)  
 T[] tempBag = (T[]) **new** Object[desiredCapacity];  
 **bag** = tempBag;  
 **numberOfEntries** = 0;  
 **initialized** = **true**;  
 }  
 **else  
 throw new** IllegalStateException(**"Attempt to create a bag whose capacity exceeds allowed maximum"**);  
 }  
  
 */\*\*  
 \* Throws an exception if this object is not initialized.  
 \*/* **private void** checkInitialization()  
 {  
 **if** (! **initialized**)  
 **throw new** SecurityException(**"ArrayBag object is not initialized properly."**);  
 }  
  
  
 */\*\*  
 \* Adds a new entry to this bag.  
 \** ***@param newEntry*** *The object to be added as a new entry.  
 \** ***@return*** *True if the addition is successful, or false if not.  
 \*/* @Override  
 **public boolean** add(T newEntry)  
 {  
 checkInitialization();  
 **boolean** result = **true**;  
 **if** (isArrayFull())  
 result = **false**;  
 **else** {  
 **bag**[**numberOfEntries**] = newEntry;  
 **numberOfEntries**++;  
 }  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Retrieves all entries that are in this bag.  
 \** ***@return*** *A newly allocated array of all the entries in this bag.  
 \*/* @Override  
 **public** T[] toArray()  
 {  
 checkInitialization();  
 *// The cast is safe because the new array contains null entries.* @SuppressWarnings(**"unchecked"**)  
 T[] result = (T[]) **new** Object[**numberOfEntries**]; *// Unchecked cast* **for** (**int** index = 0; index < **numberOfEntries**; index++)  
 result[index] = **bag**[index];  
  
 **return** result;  
 }

*/\*\*  
 \* Test the bag whether is full.  
 \** ***@return*** *True if the bag is full.  
 \*/* **private boolean** isArrayFull()  
 {  
 **return numberOfEntries** >= **bag**.**length**;  
 }  
}

#### 测试核心方法

因为ArrayBag实现了BagInterface，所以Java语法检查程序将查看这个接口中声明的每个方法的定义。但不必等到完成它们的定义后才开始测试，对可暂时忽略的方法可以给出它们的不完整定义。

一个不完整定义的方法称为存根(stub)。存根仅需要让语法检查器通过即可。例如，对于每个返回一个值的方法，通过添加一个return语句让其返回一个哑值来避免语法错误。返回布尔值的方法可以返回假。返回对象的方法返回null。而void方法可以简单地只有一个空方法体。

#### 实现更多的方法

@Override  
**public int** getCurrentSize()  
{  
 **return numberOfEntries**;  
}  
  
@Override  
**public boolean** isEmpty()  
{  
 **return numberOfEntries** == 0;  
}

方法isEmpty和getCurrentSize没有调用checkInitialization。虽然它们能调用，但我们不想因不必要的安全检查而使客户的性能降低。两个方法都涉及数据域**numberOfEntries**。即使构造方法没有完成它的初始化，也没有将这个域设置为0，但Java使用默认值将它初始化为0。

@Override  
**public int** getFrequencyOf(T anEntry)  
{  
 checkInitialization();  
 **int** counter = 0;  
 **for** (**int** index = 0; index < **numberOfEntries**; index++)  
 {  
 **if** (anEntry.equals(**bag**[index]))  
 *// anEntry.equals(bag[index]) // Compares values  
 // anEntry == bag[index] // WRONG! Compares locations (addresses)* counter++;  
 }  
 **return** counter;  
}  
  
@Override  
**public boolean** contains(T anEntry)  
{  
 checkInitialization();  
 **boolean** found = **false**;  
 **int** index = 0;  
 **while** (!found && (index < **numberOfEntries**))  
 {  
 **if** (anEntry.equals(**bag**[index]))  
 found = **true**;  
 index++;  
 }  
 **return** found;  
}

##### 删除项的方法

@Override  
**public void** clear()  
{  
 **while** (!isEmpty())  
 remove();  
}

循环中每次要删除哪个项是不重要的。所以，我们调用删除一个不确定项的remove方法。因为remove方法将调用checkInitialization，所以clear不需要显式地调用它。

删除不确定项：从包中删除一个项，涉及从数组中删除它。虽然我们能访问数组bag中的任何项，但最后一项是最容易删除的。为此，

• 访问最后一项，它能被返回。

• 将项的数组元素设置为null 。

• **numberOfEntries**减1。

**numberOfEntries**减1，就会忽略最后一项，意味着它已被高效地删除了。

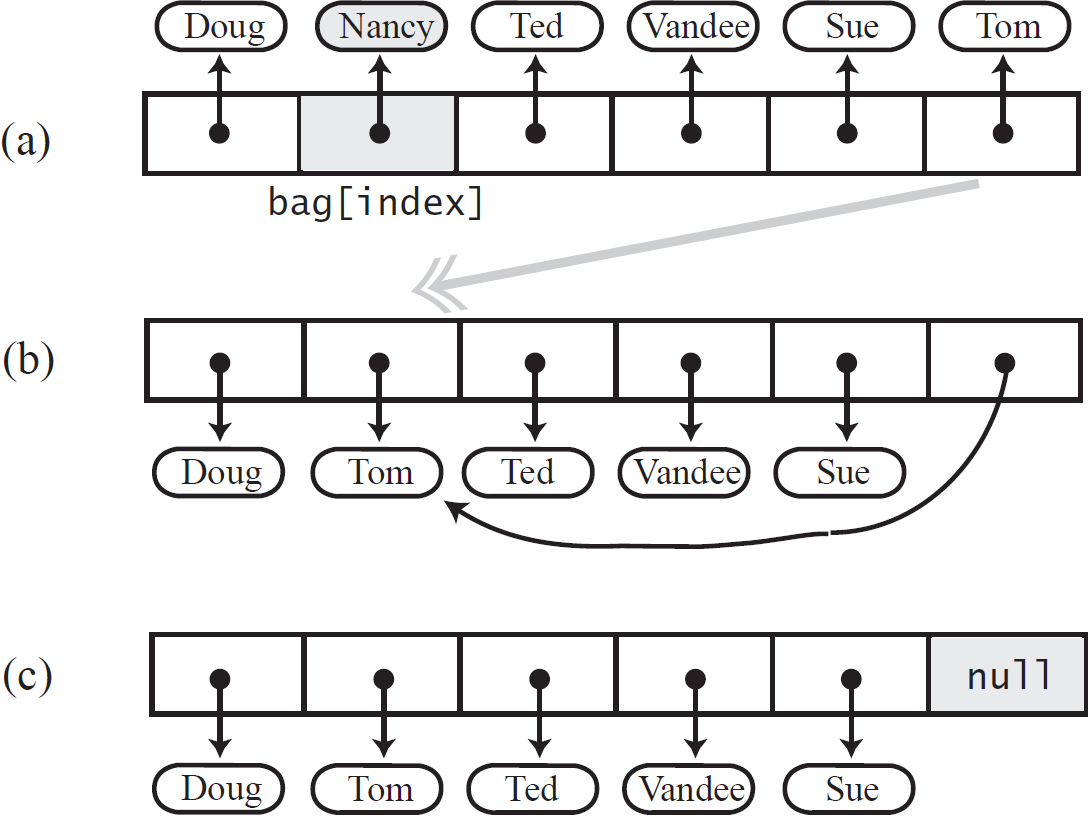
@Override  
**public** T remove()  
{  
 checkInitialization();  
 T result = **null**;  
 **if** (**numberOfEntries** > 0)  
 {  
 result = **bag**[**numberOfEntries** - 1];  
 **bag**[**numberOfEntries** - 1] = **null**; // 标记被删除对象可进行垃圾回收  
 **numberOfEntries**--;  
 }  
 **return** result;  
}

删除指定的项：如果项在包中出现多次，则仅删除它的一次出现。没有指定删除哪次出现。我们只删除查找时遇到的anEntry的首次出现。

方法一：现在需要删除bag[index]中的项。如果只写bag[index] = null; 则删除bag[index]中指向的项的引用，但数组中会留下空隙。即包的内容不再占据连续的数组位置。可以移动随后的项来去掉这个空隙，并将指向最后项的重复引用替换为null。



方法二：不需要维护包中项的具体次序。所以删除项后，不是移动数组项，而是用数组中最后面的项替换被删除的项。找到bag[index]中的anEntry后将bag[numberOfEntries-1]中的项复制到bag[index]中，然后将bag[numberOfEntries-1]中的项替换为null，最后numberOfEntries-1。



私有方法removeEntry的定义：

*/\*\*  
 \* Removes and returns the entry at a given index within the array bag.  
 \** ***@param givenIndex*** *\** ***@return*** *If no such entry exists, returns null.  
 \* Preconditions: 0 <= givenIndex < numberOfEntries; checkInitialization has been called.  
 \*/***private** T removeEntry(**int** givenIndex)  
{  
 T result = **null**;  
 **if** (!isEmpty() && (givenIndex >= 0))  
 {  
 result = **bag**[givenIndex];  
 **bag**[givenIndex] = **bag**[**numberOfEntries** - 1];  
 **bag**[**numberOfEntries** - 1] = **null**;  
 **numberOfEntries**--;  
 }  
 **return** result;  
}

修改remove方法：

@Override  
**public** T remove()  
{  
 checkInitialization();  
 T result = removeEntry(**numberOfEntries** - 1);  
 */\*  
 T result = null;  
 if (numberOfEntries > 0)  
 {  
 result = bag[numberOfEntries - 1];  
 bag[numberOfEntries - 1] = null; // 标记被删除对象可进行垃圾回收  
 numberOfEntries--;  
 }\*/* **return** result;  
}

删除指定项方法：

@Override  
**public boolean** remove(T anEntry)  
{  
 checkInitialization();  
 **int** index = getIndexOf(anEntry);  
 T result = removeEntry(index);  
 **return** anEntry.equals(result);  
}

找到要删除项的索引：

**private int** getIndexOf(T anEntry)  
{  
 **int** where = -1;  
 **boolean** found = **false**;  
 **int** index = 0;  
 **while** (!found && (index < **numberOfEntries**))  
 {  
 **if** (anEntry.equals(**bag**[index]))  
 {  
 found = **true**;  
 where = index;  
 }  
 index++;  
 }  
 **return** where;  
}

contains方法修改：

**public boolean** contains(T anEntry)  
{  
 checkInitialization();  
 **return** getIndexOf(anEntry) > -1;  
}

#### 代码



**public interface** BagInterface<T>  
{  
 */\*\*  
 \* Gets the current number of entries in this bag.  
 \** ***@return*** *The integer number of entries currently in the bag.  
 \*/* **public int** getCurrentSize();  
  
 */\*\*  
 \* Sees whether this bag is empty.  
 \** ***@return*** *True if the bag is empty, or false if not.  
 \*/* **public boolean** isEmpty();  
  
 */\*\*  
 \* Adds a new entry to this bag.  
 \** ***@param newEntry*** *The object to be added as a new entry.  
 \** ***@return*** *True if the addition is successful, or false if not.  
 \*/* **public boolean** add(T newEntry);  
  
 */\*\*  
 \* Removes one unspecified entry from this bag, if possible.  
 \** ***@return*** *Either the removed entry, if the removal was successful, or null.  
 \*/* **public** T remove();  
  
 */\*\*  
 \* Removes one occurrence of a given entry from this bag.  
 \** ***@param anEntry*** *The entry to be removed.  
 \** ***@return*** *True if the removal was successful, or false if not.  
 \*/* **public boolean** remove(T anEntry);  
  
 */\*\*  
 \* Removes all entries from this bag.  
 \*/* **public void** clear();  
  
 */\*\*  
 \* Counts the number of times a given entry appears in this bag.  
 \** ***@param anEntry*** *The entry to be counted.  
 \** ***@return*** *The number of times anEntry appears in the bag.  
 \*/* **public int** getFrequencyOf(T anEntry);  
  
 */\*\*  
 \* Tests whether this bag contains a given entry.  
 \** ***@param anEntry*** *The entry to be tested.  
 \** ***@return*** *True if the bag contains anEntry, or false if not.  
 \*/* **public boolean** contains(T anEntry);  
  
 */\*\*  
 \* Retrieves all entries that are in this bag.  
 \** ***@return*** *A newly allocated array of all the entries in the bag.  
 \* Note: If the bag is empty, the returned array is empty.  
 \*/* **public** T[] toArray();  
}



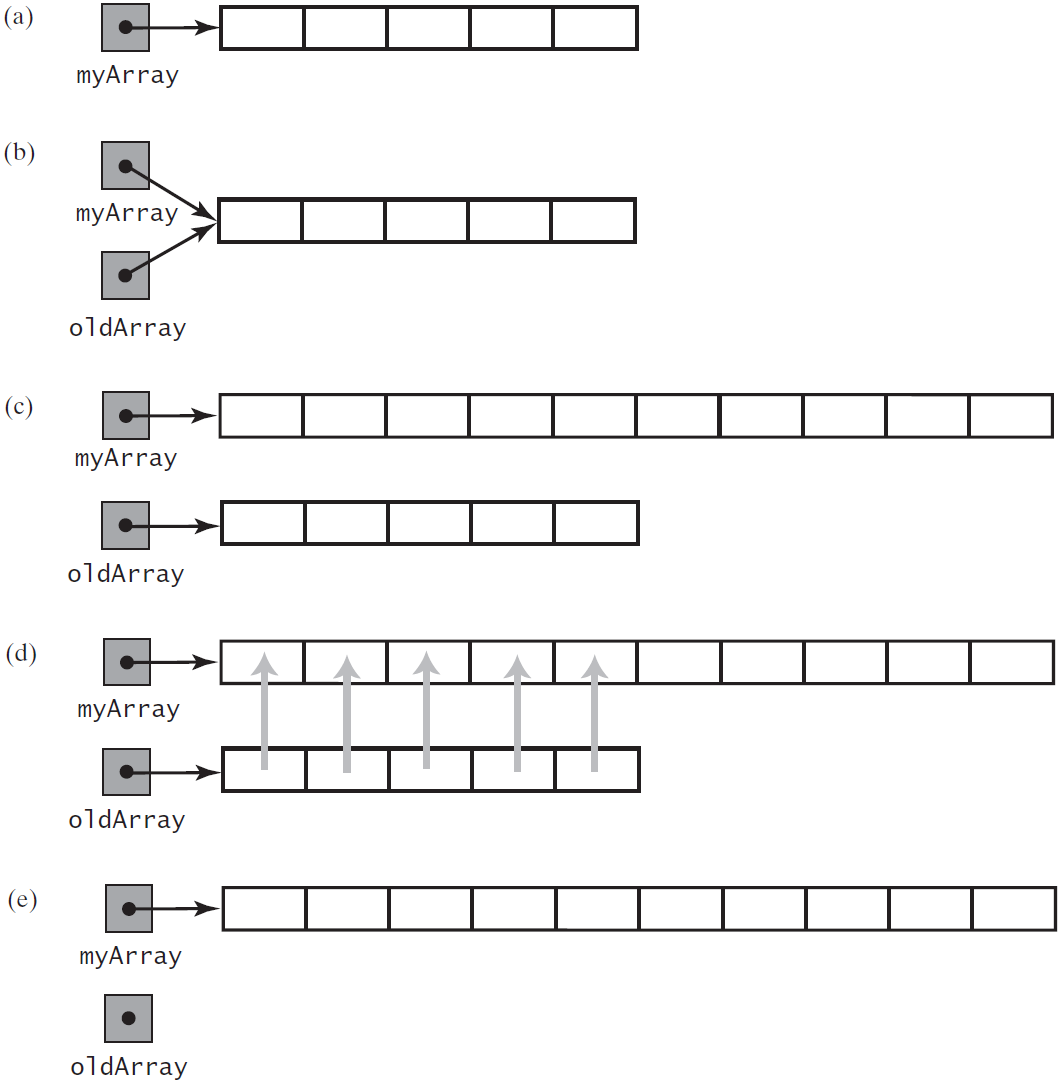
**public final class** ArrayBag<T> **implements** BagInterface<T> *// 声明为终态类，不会再派生其他类*{  
 **private final** T[] **bag**;  
 **private int numberOfEntries**;  
 **private static final int *DEFAULT\_CAPACITY*** = 25;  
  
 */\*  
 避免以下两个问题，修改构造方法  
 ·如果构造方法没有完全执行;  
 ·如果客户试图创建一个其容量超出给定范围的包  
 \*/* **private boolean initialized** = **false**;  
 **private static final int *MAX\_CAPACITY*** = 10000;  
  
 **public** ArrayBag()  
 {  
 **this**(***DEFAULT\_CAPACITY***);  
 }  
  
 **public** ArrayBag(**int** desiredCapacity)  
 {  
 **if** (desiredCapacity <= ***MAX\_CAPACITY***)  
 {  
 @SuppressWarnings(**"unchecked"**)  
 T[] tempBag = (T[]) **new** Object[desiredCapacity];  
 **bag** = tempBag;  
 **numberOfEntries** = 0;  
 **initialized** = **true**;  
 }  
 **else  
 throw new** IllegalStateException(**"Attempt to create a bag whose capacity exceeds allowed maximum"**);  
 }  
  
 */\*\*  
 \* Throws an exception if this object is not initialized.  
 \*/* **private void** checkInitialization()  
 {  
 **if** (! **initialized**)  
 **throw new** SecurityException(**"ArrayBag object is not initialized properly."**);  
 }  
  
  
 */\*\*  
 \* Adds a new entry to this bag.  
 \** ***@param newEntry*** *The object to be added as a new entry.  
 \** ***@return*** *True if the addition is successful, or false if not.  
 \*/* @Override  
 **public boolean** add(T newEntry)  
 {  
 checkInitialization();  
 **boolean** result = **true**;  
 **if** (isArrayFull())  
 result = **false**;  
 **else** {  
 **bag**[**numberOfEntries**] = newEntry;  
 **numberOfEntries**++;  
 }  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Retrieves all entries that are in this bag.  
 \** ***@return*** *A newly allocated array of all the entries in this bag.  
 \*/* @Override  
 **public** T[] toArray()  
 {  
 checkInitialization();  
 *// The cast is safe because the new array contains null entries.* @SuppressWarnings(**"unchecked"**)  
 T[] result = (T[]) **new** Object[**numberOfEntries**]; *// Unchecked cast* **for** (**int** index = 0; index < **numberOfEntries**; index++)  
 result[index] = **bag**[index];  
  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Test the bag whether is full.  
 \** ***@return*** *True if the bag is full.  
 \*/* **private boolean** isArrayFull()  
 {  
 **return numberOfEntries** >= **bag**.**length**;  
 }  
  
  
 */\*\*  
 \* Gets the current number of entries in this bag.  
 \** ***@return*** *The integer number of entries currently in the bag.  
 \*/* @Override  
 **public int** getCurrentSize()  
 {  
 **return numberOfEntries**;  
 }  
  
  
 */\*\*  
 \* Sees whether this bag is empty.  
 \** ***@return*** *True if the bag is empty, or false if not.  
 \*/* @Override  
 **public boolean** isEmpty()  
 {  
 **return numberOfEntries** == 0;  
 }  
  
  
 */\*\*  
 \* Counts the number of times a given entry appears in this bag.  
 \** ***@param anEntry*** *The entry to be counted.  
 \** ***@return*** *The number of times anEntry appears in the bag.  
 \*/* @Override  
 **public int** getFrequencyOf(T anEntry)  
 {  
 checkInitialization();  
 **int** counter = 0;  
 **for** (**int** index = 0; index < **numberOfEntries**; index++)  
 {  
 **if** (anEntry.equals(**bag**[index]))  
 *// anEntry.equals(bag[index]) // Compares values  
 // anEntry == bag[index] // WRONG! Compares locations (addresses)* counter++;  
 }  
 **return** counter;  
 }  
  
  
 */\*\*  
 \* Tests whether this bag contains a given entry.  
 \** ***@param anEntry*** *The entry to be tested.  
 \** ***@return*** *True if the bag contains anEntry, or false if not.  
 \*/  
 /\*  
 @Override  
 public boolean contains(T anEntry)  
 {  
 checkInitialization();  
 boolean found = false;  
 int index = 0;  
 while (!found && (index < numberOfEntries))  
 {  
 if (anEntry.equals(bag[index]))  
 found = true;  
 index++;  
 }  
 return found;  
 }\*/* **public boolean** contains(T anEntry)  
 {  
 checkInitialization();  
 **return** getIndexOf(anEntry) > -1;  
 }  
  
  
 */\*\*  
 \* Removes all entries from this bag.  
 \*/* @Override  
 **public void** clear()  
 {  
 **while** (!isEmpty())  
 remove();  
 }  
  
  
 */\*\*  
 \* Removes one unspecified entry from this bag, if possible.  
 \** ***@return*** *Either the removed entry, if the removal was successful, or null.  
 \*/* @Override  
 **public** T remove()  
 {  
 checkInitialization();  
 T result = removeEntry(**numberOfEntries** - 1);  
 */\*  
 T result = null;  
 if (numberOfEntries > 0)  
 {  
 result = bag[numberOfEntries - 1];  
 bag[numberOfEntries - 1] = null; // 标记被删除对象可进行垃圾回收  
 numberOfEntries--;  
 }\*/* **return** result;  
 }  
  
  
 */\*\*  
 \* Removes and returns the entry at a given index within the array bag.  
 \** ***@param givenIndex*** *\** ***@return*** *If no such entry exists, returns null.  
 \* Preconditions: 0 <= givenIndex < numberOfEntries; checkInitialization has been called.  
 \*/* **private** T removeEntry(**int** givenIndex)  
 {  
 T result = **null**;  
 **if** (!isEmpty() && (givenIndex >= 0))  
 {  
 result = **bag**[givenIndex];  
 **bag**[givenIndex] = **bag**[**numberOfEntries** - 1];  
 **bag**[**numberOfEntries** - 1] = **null**;  
 **numberOfEntries**--;  
 }  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Removes one occurrence of a given entry from this bag.  
 \** ***@param anEntry*** *The entry to be removed.  
 \** ***@return*** *True if the removal was successful, or false if not.  
 \*/* @Override  
 **public boolean** remove(T anEntry)  
 {  
 checkInitialization();  
 **int** index = getIndexOf(anEntry);  
 T result = removeEntry(index);  
 **return** anEntry.equals(result);  
 }  
  
  
 */\*\*  
 \* Locates a given entry within the array bag.  
 \** ***@param anEntry*** *\** ***@return*** *Returns the index of the entry, if located, or -1 otherwise.  
 \* Precondition: checkInitialization has been called.  
 \*/* **private int** getIndexOf(T anEntry)  
 {  
 **int** where = -1;  
 **boolean** found = **false**;  
 **int** index = 0;  
 **while** (!found && (index < **numberOfEntries**))  
 {  
 **if** (anEntry.equals(**bag**[index]))  
 {  
 found = **true**;  
 where = index;  
 }  
 index++;  
 }  
 **return** where;  
 }  
}



**public class** ArrayBagTest  
{  
 **public static void** main(String[] args)  
 {  
 String[] contentsOfBag = {**"A"**, **"A"**, **"B"**, **"A"**, **"C"**, **"A"**};  
  
 *// Tests on an empty bag* BagInterface<String> newBag = **new** ArrayBag<>(contentsOfBag.**length**);  
 System.***out***.println(**"Testing an initially empty bag:"**);  
 *testIsEmpty*(newBag, **true**);  
  
 String[] testStrings1 = {**""**, **"B"**};  
 *testFrequency*(newBag, testStrings1);  
 *testContains*(newBag, testStrings1);  
 *testRemove*(newBag, testStrings1);  
  
 *// Adding strings* System.***out***.println(**"\nAdding "** + contentsOfBag.**length** + **" string(s) to the bag."**);  
 *testAdd*(newBag, contentsOfBag);  
  
 *// Tests on a bag that is not empty* System.***out***.println();  
 System.***out***.println();  
 *testIsEmpty*(newBag, **false**);  
 String[] testStrings2 = {**"A"**, **"B"**, **"C"**, **"D"**, **"Z"**};  
 *testFrequency*(newBag, testStrings2);  
 *testContains*(newBag, testStrings2);  
  
 *// Removing strings* String[] testStrings3 = {**""**, **"B"**, **"A"**, **"C"**, **"Z"**};  
 *testRemove*(newBag, testStrings3);  
  
 System.***out***.println(**"\n\nClearing the bag:"**);  
 newBag.clear();  
 *displayBag*(newBag);  
  
 *// Filling an initially empty bag to capacity* System.***out***.println(**"\nTesting an initially empty bag that will be filled to capacity:"**);  
 BagInterface<String> newBag2 = **new** ArrayBag<>(7);  
 String[] contentsOfBag2 = {**"A"**, **"B"**, **"A"**, **"C"**, **"B"**, **"C"**, **"D"**};  
 *testAdd*(newBag2, contentsOfBag2);  
  
 System.***out***.println(**"\nTry to add another string to the bag:"**);  
 **if** (newBag2.add(**"another string"**))  
 {  
 System.***out***.println(**"Adding another string to the bag!"**);  
 *displayBag*(newBag2);  
 }  
 **else** System.***out***.println(**"Added a string beyond the capacity of the bag!"**);  
 }  
  
  
 */\*\*  
 \* Tests the method isEmpty.  
 \** ***@param newBag*** *\** ***@param correctResult*** *correctResult guesses what isEmpty() should return.  
 \*/* **private static void** testIsEmpty(BagInterface<String> newBag, **boolean** correctResult)  
 {  
 System.***out***.println(**"Testing isEmpty() with:"**);  
 **if** (newBag.isEmpty())  
 System.***out***.println(**"an empty bag:"**);  
 **else** System.***out***.println(**"a bag that is not empty:"**);  
  
 **if** (newBag.isEmpty() && correctResult)  
 System.***out***.println(**"该包是空的，判断正确！"**);  
 **else if** (newBag.isEmpty() && !correctResult)  
 System.***out***.println(**"该包是空包，判断错误！"**);  
 **else if** (!newBag.isEmpty() && correctResult)  
 System.***out***.println(**"该包不是空包，判断错误！"**);  
 **else** System.***out***.println(**"该包不是空包，判断正确！"**);  
 }  
  
  
 */\*\*  
 \* Tests the method getFrequencyOf()  
 \** ***@param newBag*** *\** ***@param testString*** *\*/* **private static void** testFrequency(BagInterface<String> newBag, String[] testString)  
 {  
 System.***out***.println(**"\nTesting the method of getFrequencyOf:"**);  
 **for** (**int** index = 0; index < testString.**length**; index++)  
 {  
 String oneString = testString[index];  
 **if** (!oneString.equals(**""**) && oneString != **null**)  
 System.***out***.println(**"In this bag, the count of "** + oneString +  
 **" is "** + newBag.getFrequencyOf(oneString));  
 }  
 }  
  
  
 */\*\*  
 \* Tests the method contains.  
 \** ***@param newBag*** *\** ***@param testString*** *\*/* **private static void** testContains(BagInterface<String> newBag, String[] testString)  
 {  
 System.***out***.println(**"\nTesting the method contains:"**);  
 **for** (**int** index = 0; index < testString.**length**; index++)  
 {  
 String oneString = testString[index];  
 **if** (!oneString.equals(**""**) && oneString != **null**)  
 System.***out***.println(**"Does the bag contains "** + oneString + **" ? "** + newBag.contains(oneString));  
 }  
 }  
  
  
 */\*\*  
 \* Tests the two remove methods.  
 \** ***@param newBag*** *\** ***@param testString*** *\*/* **private static void** testRemove(BagInterface<String> newBag, String[] testString)  
 {  
 System.***out***.println(**"\nTesting methods remove:"**);  
 System.***out***.println(**"Before using remove method: "**);  
 *displayBag*(newBag);  
  
 **if** (newBag.isEmpty())  
 System.***out***.println(**"The bag is empty. So it is unnecessary to use \"remove\" method."**);  
 **else** {  
 **for** (**int** index = 0; index < testString.**length**; index++)  
 {  
 String oneString = testString[index];  
 **if** (oneString.equals(**""**) || oneString == **null**)  
 {  
 *// Testing remove()* System.***out***.println(**"\nRemoving one string from the bag:"**);  
 System.***out***.println(**"remove() returns "** + newBag.remove());  
 }  
 **else** {  
 *// Testing remove(oneString)* System.***out***.println(**"Removing \""** + oneString + **"\" from the bag:"**);  
 System.***out***.println(**"Whether remove \""** + oneString + **"\" ? "** + newBag.remove(oneString));  
 }  
 }  
 }  
  
 System.***out***.println(**"After using remove method: "**);  
 *displayBag*(newBag);  
 }  
  
  
 */\*\*  
 \* Tests the method toArray() and getCurrentSize() while displaying the bag.  
 \** ***@param aBag*** *\*/* **private static void** displayBag(BagInterface<String> aBag)  
 {  
 **if** (aBag.isEmpty())  
 System.***out***.println(**"The bag is empty."**);  
 **else** {  
 System.***out***.println(**"The bag contains "** + aBag.getCurrentSize() + **" string(s), as follow:"**);  
 *//for (int index = 0; index < aBag.toArray().length; index++)  
 //System.out.print(aBag.toArray()[index] + " ");* Object[] bagArray = aBag.toArray(); *// toArray()返回T[]* **for** (**int** index = 0; index < bagArray.**length**; index++)  
 System.***out***.print(bagArray[index] + **" "**);  
 }  
 }  
  
 **private static void** testAdd(BagInterface<String> aBag, String[] contents)  
 {  
 System.***out***.println(**"\nAdding contents to bag:"**);  
 System.***out***.println(**"Before using add method: "**);  
 *displayBag*(aBag);  
  
 **for** (**int** index = 0; index < contents.**length**; index++)  
 aBag.add(contents[index]);  
  
 System.***out***.println(**"After using add method: "**);  
 *displayBag*(aBag);  
 }  
}

### 使用可变大小的数组实现ADT包

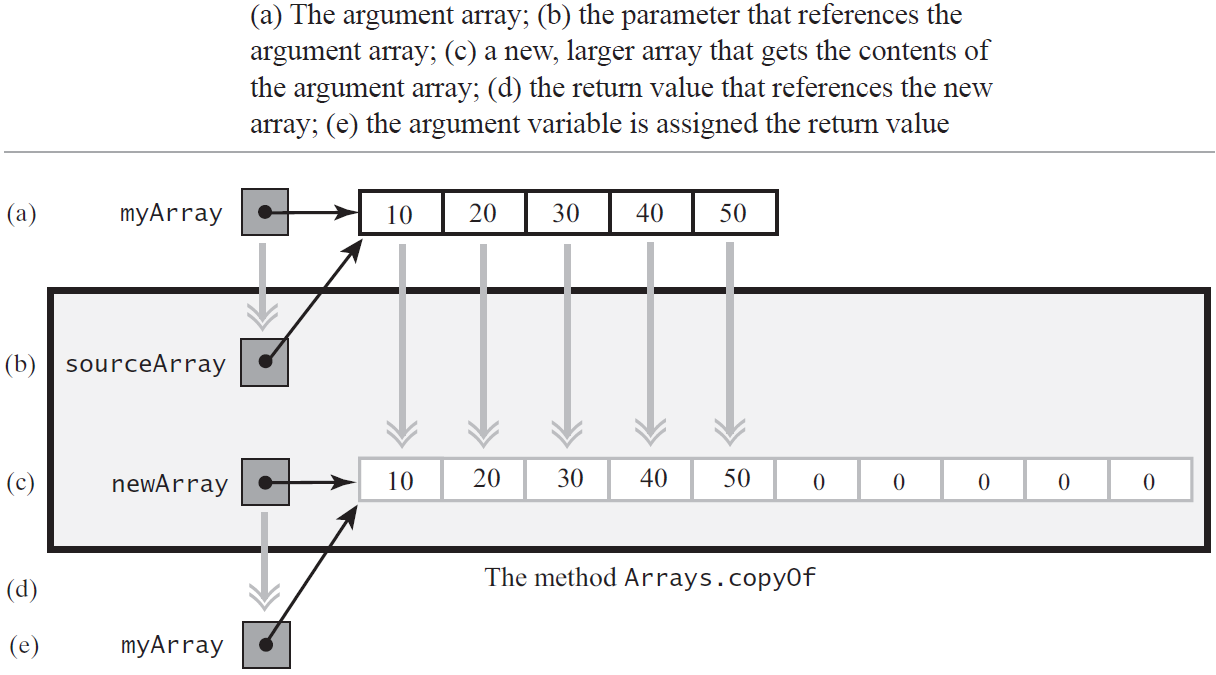
假定已有myArray指向的数组，先定义一个别名oldArray，它也指向这个数组。下一步是创建一个比原始数组更大的新数组，让myArray指向这个新数组。一般地新数组要两倍于原始数组的大小。最后一步是将原始数组的内容复制到新数组中，然后丢弃原始数组。



当数组不再被引用时，它的内存在垃圾回收时被收回。

***Arrays.copyOf(sourceArray, newLength)***

myArray = Arrays.copyOf(myArray, 2 \* myArray.length);



当在含有50 个项的满数组中添加一项时，在进行添加前先将50个元素的数组复制到100个元素的数组中。那么接下来的49次添加都可以快速完成而不需要复制数组，所以数组复制只需一次。

#### 包的新实现

·将类名改为ResizableArrayBag。

·从数组bag的声明中删除标识符final，以便可以调整它的大小。

·修改构造方法的名字以匹配新的类名。

·修改方法add的定义，让它总能容纳新项。该方法永远不会返回假，因为包永远不会满。



##### 构造器

**public** ResizableArrayBag(**int** initialCapacity)  
{  
 checkCapacity(initialCapacity);  
 @SuppressWarnings(**"unchecked"**)  
 T[] tempBag = (T[]) **new** Object[initialCapacity];  
 **bag** = tempBag;  
 **numberOfEntries** = 0;  
 **initialized** = **true**;  
}

*/\*\*  
 \* Creates a bag containing given entries.  
 \** ***@param contents*** *\*/***public** ResizableArrayBag(T[] contents)  
{  
 checkCapacity(contents.**length**);  
 **bag** = Arrays.*copyOf*(contents, contents.**length**);  
 **numberOfEntries** = contents.**length**;  
 **initialized** = **true**;  
}

##### 方法add

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

##### 方法checkCapacity()

**private void** checkCapacity(**int** capacity)  
{  
 **if** (capacity > ***MAX\_CAPACITY***)  
 **throw new** IllegalStateException(**"Attempt to create a bag whose capacity"** +  
 **" exceeds allowed maximum of "** + ***MAX\_CAPACITY***);  
}

##### 方法doubleCapacity()

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

## 使用链式数据实现包

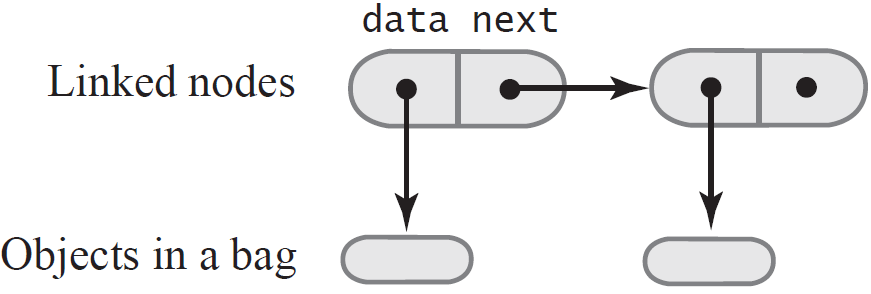
使用数组实现ADT包既有优点也有缺点，数组有固定的大小，所以它可能会满，或有一些未用的元素。当它变满时可以将项移到更大的数组中，改变数组的大小。虽然调整数组大小可以为包提供所需要的空间，但在每次扩展数组时必须移动数据。

### ADT包的链式实现

#### 私有类Node

结点是对象，一般结点与其他结点链在一起形成一个数据结构。每个结点都有两个数据域：一个域指向一段数据，而另一个域指向另一个结点。

域data含有指向包中对象的一个引用，data的数据类型表示为泛型T。T与包的类将声明为同一个泛型。域next中含有指向另一个结点的引用，它的数据类型是Node。结点不是指向另一个结点中的数据，而是指向整个结点。



**private class** Node  
{  
 **private** T **data**;  
 **private** Node **next**;  
  
 **private** Node(T dataPortion)  
 {  
 **this**(dataPortion, **null**);  
 }  
  
 **private** Node(T dataPortion, Node nextNode)  
 {  
 **data** = dataPortion;  
 **next** = nextNode;  
 }  
}

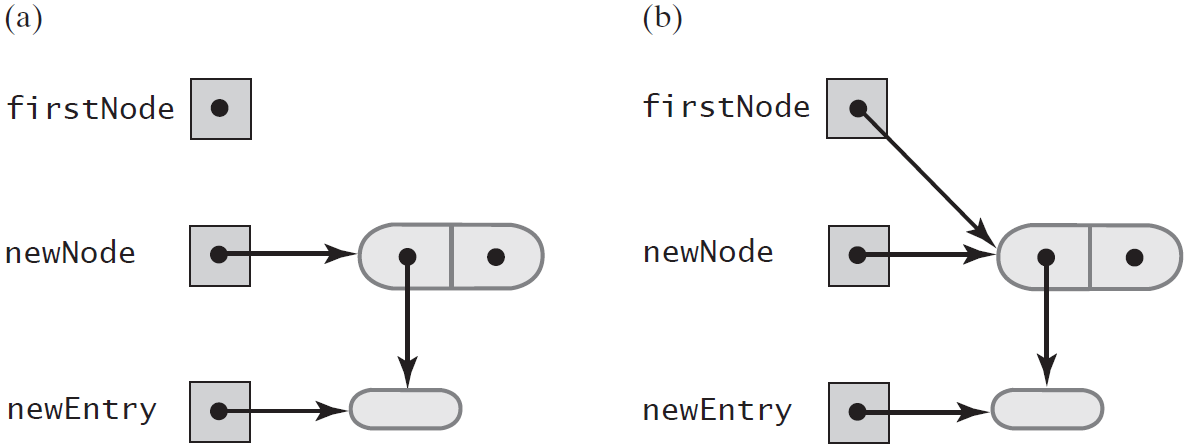
因为它的位置包含在另一个类内，所以Node是一个内层类(inner class)。因为Node是一个内层类，所以泛型T将与包含Node 的外层类声明的泛型是一样的。所以，没有在Node后写<T>。但如果Node不是内层类，而是有包访问或公有访问权限，则应该写Node<T>。这种情况下，Node还需要设置方法(set) 和获取方法(get)来访问它的数据域。

#### 方法add

初建结点链：

***Node newNode = new Node(newEntry);***

***firstNode = newNode;***



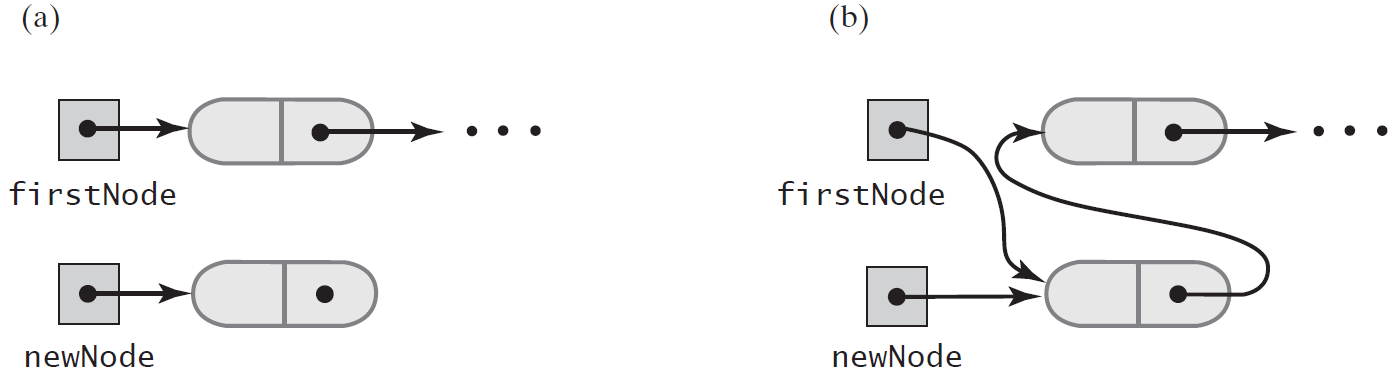
将新结点插入完成后，应该只有firstNode指向它。我们可以将newNode 设置为null。

添加到结点链中：

***Node newNode = new Node(newEntry);***

***newNode.next = firstNode;***

***firstNode = newNode;***



**public boolean** add(T newEntry)  
{  
 Node newNode = **new** Node(newEntry);  
 newNode.**next** = **firstNode**;  
 **firstNode** = newNode;  
 **numberOfEntries**++;  
 **return true**;  
}

#### 方法toArray

**public** T[] toArray()  
{  
 @SuppressWarnings(**"unchecked"**)  
 T[] result = (T[]) **new** Object[**numberOfEntries**]; *// Unchecked cast* **int** index = 0;  
 Node currentNode = **firstNode**;  
 **while** ((index < **numberOfEntries**) && (currentNode != **null**))  
 {  
 result[index] = currentNode.**data**;  
 index++;  
 currentNode = currentNode.**next**;  
 }  
 **return** result;  
}

#### 方法getFrequencyOf

**public int** getFrequencyOf(T anEntry)  
{  
 **int** frequency = 0;  
 **int** loopCounter = 0;  
 Node currentNode = **firstNode**;  
  
 **while** ((loopCounter < **numberOfEntries**) && (currentNode != **null**))  
 {  
 **if** (anEntry.equals(currentNode.**data**))  
 frequency++;  
 loopCounter++;  
 currentNode = currentNode.**next**;  
 }  
  
 **return** frequency;  
}

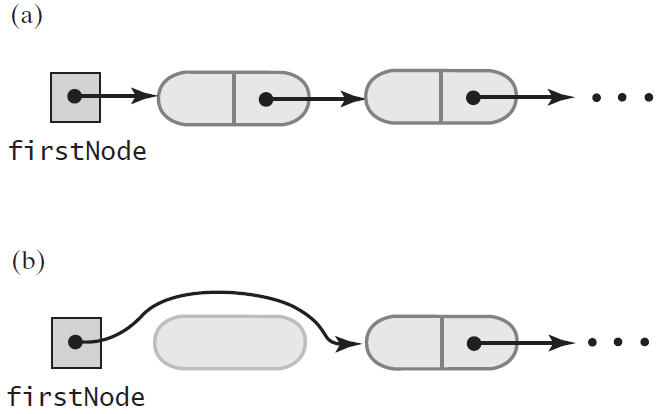
#### 方法contains

**public boolean** contains(T anEntry)  
{  
 **boolean** found = **false**;  
 Node currentNode = **firstNode**;  
  
 **while** ((!found) && currentNode != **null**)  
 {  
 **if** (anEntry.equals(currentNode.**data**))  
 found = **true**;  
 currentNode = currentNode.**next**;  
 }  
   
 **return** found;  
}

#### 方法remove

删除未指定项(删除第一个结点)：

**public** T remove()  
{  
 T result = **null**;  
 **if** (**firstNode** != **null**)  
 {  
 result = **firstNode**.**data**;  
 **firstNode** = **firstNode**.**next**; *// Remove the first node from chain* **numberOfEntries**--;  
 }  
 **return** result;  
}



删除给定的项：

要在结点链中删除指定的项，首先必须找到这个项。即，必须遍历链，并检查结点中的项。假定我们在结点N中找到所要找的项，如果结点N不在链的第一个位置，则可用下面的步骤删除它的项：

1. 用第一个结点中的项替换结点N中的项。
2. 从链中删除第一个结点。

查找含有给定项的结点：

**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;  
}

删除具体项：

**public boolean** remove(T anEntry)  
{  
 **boolean** result = **false**;  
 Node nodeN = getReferenceTo(anEntry);  
 **if** (nodeN != **null**)  
 {  
 nodeN.**data** = **firstNode**.**data**;  
 **firstNode** = **firstNode**.**next**;  
 **numberOfEntries**--; *// 注意别漏写* result = **true**;  
 }  
 **return** result;  
}

#### 方法clear

**public void** clear()  
{  
 **while** (!isEmpty())  
 remove();  
}

#### 方法isEmpty

**public boolean** isEmpty()  
{  
 **return numberOfEntries** == 0;  
}

#### 方法getCurrentSize

**public int** getCurrentSize()  
{  
 **return numberOfEntries**;  
}

#### 完整程序

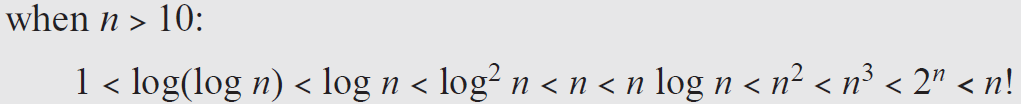


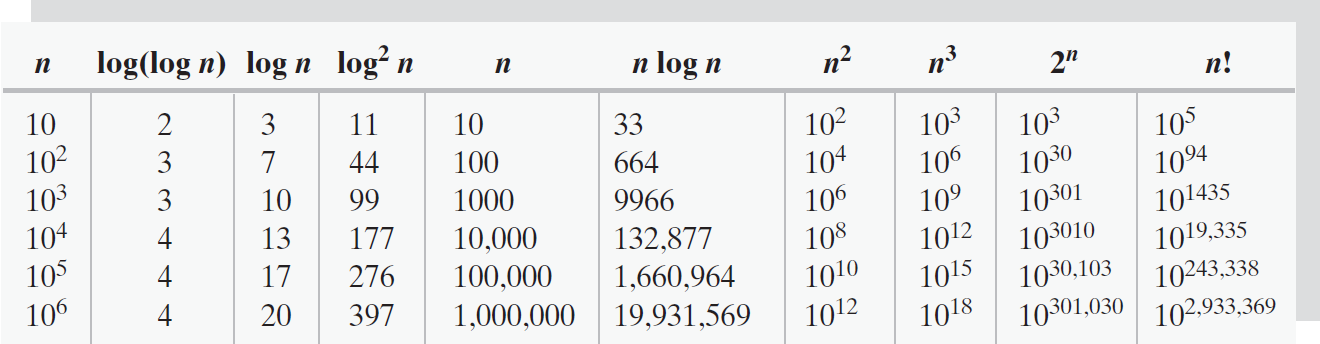
*/\*\*  
 \* A class of bags whose entries are stored in a chain of linked nodes.  
 \** ***@param <T>*** *\*/***public final class** LinkedBag<T> **implements** BagInterface<T>  
{  
 **private** Node **firstNode**; *// reference to the first node* **private int numberOfEntries**; *// the number of entries in the bag and nodes* **public** LinkedBag()  
 {  
 *// 所赋的这个值与省略构造方法时使用默认值赋给的值是一样的。  
 // 这些赋值不会失败。* **firstNode** = **null**;  
 **numberOfEntries** = 0;  
 }  
  
  
 */\*\*  
 \* Gets the current number of entries in this bag.  
 \** ***@return*** *The integer number of entries currently in the bag.  
 \*/* **public int** getCurrentSize()  
 {  
 **return numberOfEntries**;  
 }  
  
 */\*\*  
 \* Sees whether this bag is empty.  
 \** ***@return*** *True if the bag is empty, or false if not.  
 \*/* **public boolean** isEmpty()  
 {  
 **return numberOfEntries** == 0;  
 }  
  
  
 */\*\*  
 \* Removes all entries from this bag.  
 \*/* **public void** clear()  
 {  
 **while** (!isEmpty())  
 remove();  
 }  
  
  
 */\*\*  
 \* Removes one occurrence of a given entry from this bag.  
 \** ***@param anEntry*** *The entry to be removed.  
 \** ***@return*** *True if the removal was successful, or false if not.  
 \*/* **public boolean** remove(T anEntry)  
 {  
 **boolean** result = **false**;  
 Node nodeN = getReferenceTo(anEntry);  
 **if** (nodeN != **null**)  
 {  
 nodeN.**data** = **firstNode**.**data**;  
 **firstNode** = **firstNode**.**next**;  
 **numberOfEntries**--; *// 注意别漏写* result = **true**;  
 }  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Locates a given entry within this bag.  
 \** ***@param anEntry*** *\** ***@return*** *Returns a reference to the node containing the entry, if located, or null otherwise.  
 \*/* **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;  
 }  
  
  
 */\*\*  
 \* Removes one unspecified entry(the first one) from this bag, if possible.  
 \** ***@return*** *Return the removed entry, if the removal was successful, or null.  
 \*/* **public** T remove()  
 {  
 T result = **null**;  
 **if** (**firstNode** != **null**)  
 {  
 result = **firstNode**.**data**;  
 **firstNode** = **firstNode**.**next**; *// Remove the first node from chain* **numberOfEntries**--;  
 }  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Tests whether this bag contains a given entry.  
 \** ***@param anEntry*** *The entry to be tested.  
 \** ***@return*** *\*/* **public boolean** contains(T anEntry)  
 {  
 **boolean** found = **false**;  
 Node currentNode = **firstNode**;  
  
 **while** ((!found) && currentNode != **null**)  
 {  
 **if** (anEntry.equals(currentNode.**data**))  
 found = **true**;  
 currentNode = currentNode.**next**;  
 }  
  
 **return** found;  
 }  
  
  
 */\*\*  
 \* Counts the number of times a given entry appears in this bag.  
 \** ***@param anEntry*** *The entry to be counted.  
 \** ***@return*** *\*/* **public int** getFrequencyOf(T anEntry)  
 {  
 **int** frequency = 0;  
 **int** loopCounter = 0;  
 Node currentNode = **firstNode**;  
  
 **while** ((loopCounter < **numberOfEntries**) && (currentNode != **null**))  
 {  
 **if** (anEntry.equals(currentNode.**data**))  
 frequency++;  
 loopCounter++;  
 currentNode = currentNode.**next**;  
 }  
  
 **return** frequency;  
 }  
  
  
 */\*\*  
 \* Retrieves all entries that are in this bag.  
 \** ***@return*** *a newly allocated array of all the entries in the bag.  
 \*/* **public** T[] toArray()  
 {  
 @SuppressWarnings(**"unchecked"**)  
 T[] result = (T[]) **new** Object[**numberOfEntries**]; *// Unchecked cast* **int** index = 0;  
 Node currentNode = **firstNode**;  
 **while** ((index < **numberOfEntries**) && (currentNode != **null**))  
 {  
 result[index] = currentNode.**data**;  
 index++;  
 currentNode = currentNode.**next**;  
 }  
 **return** result;  
 }  
  
 */\*\*  
 \* Adds a new entry to this bag.  
 \* Add to the beginning of chain:  
 \** ***@param newEntry*** *The object to be added as a new entry.  
 \** ***@return*** *\*/* **public boolean** add(T newEntry)  
 {  
 Node newNode = **new** Node(newEntry);  
 newNode.**next** = **firstNode**;  
 **firstNode** = newNode;  
 **numberOfEntries**++;  
 **return true**;  
 }  
  
  
 */\*\*  
 \* inner class  
 \*/* **private class** Node  
 {  
 **private** T **data**;  
 **private** Node **next**;  
  
 **private** Node(T dataPortion)  
 {  
 **this**(dataPortion, **null**);  
 }  
  
 **private** Node(T dataPortion, Node nextNode)  
 {  
 **data** = dataPortion;  
 **next** = nextNode;  
 }  
 }  
}

# 算法的效率

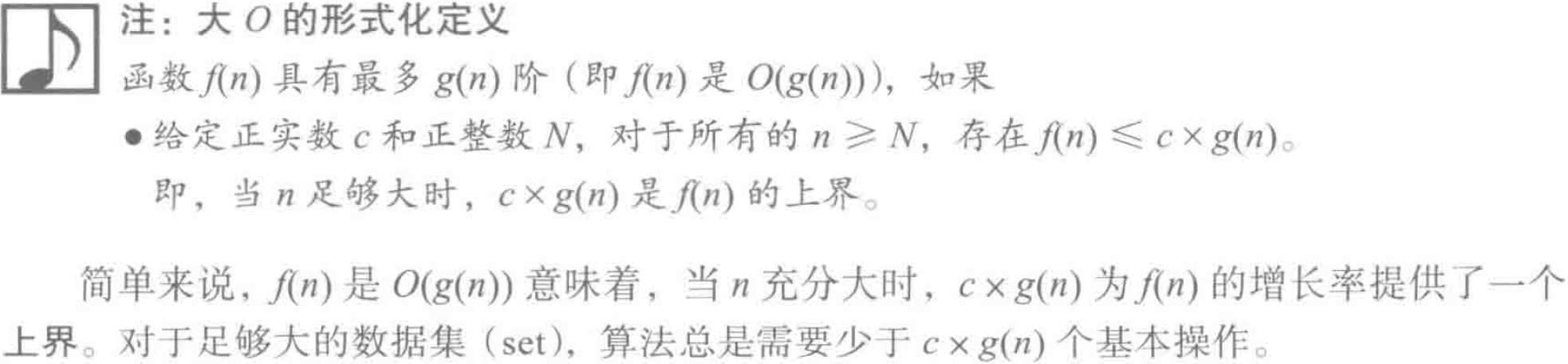
时间复杂度(time complexity)：运行它花了多少时间；

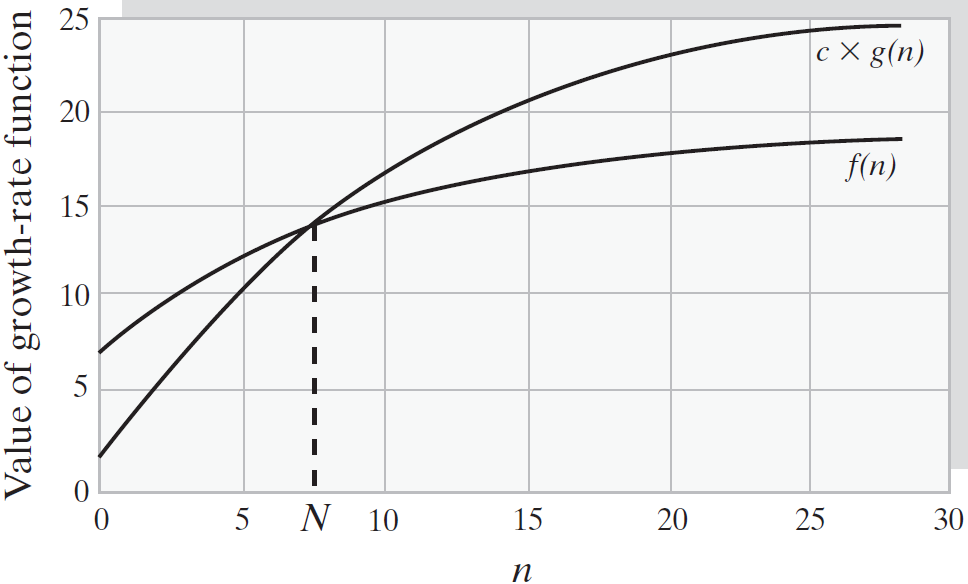
空间复杂度(space complexity)：运行它需要多少内存。



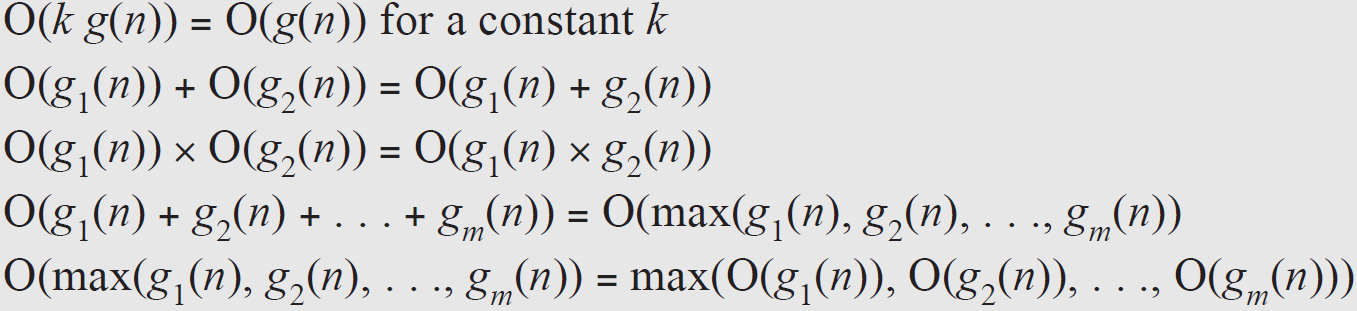


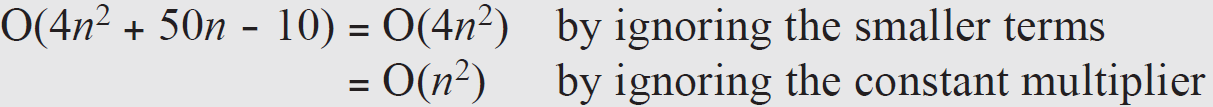
## 大O表示法

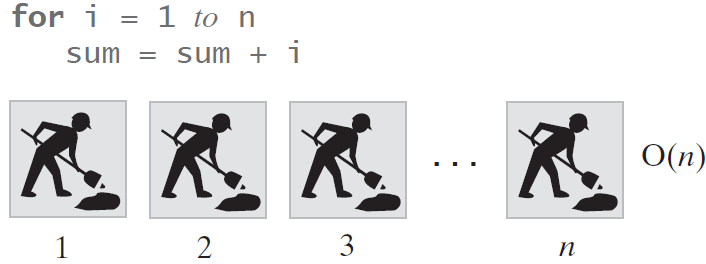


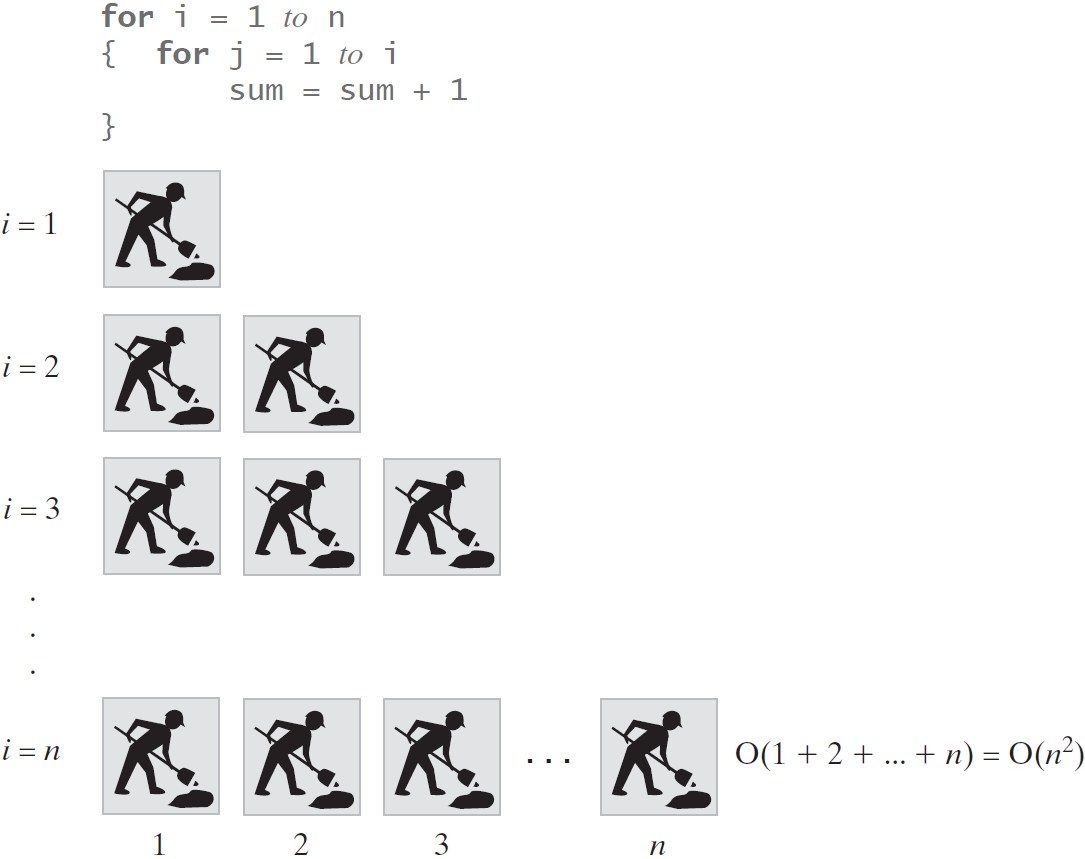


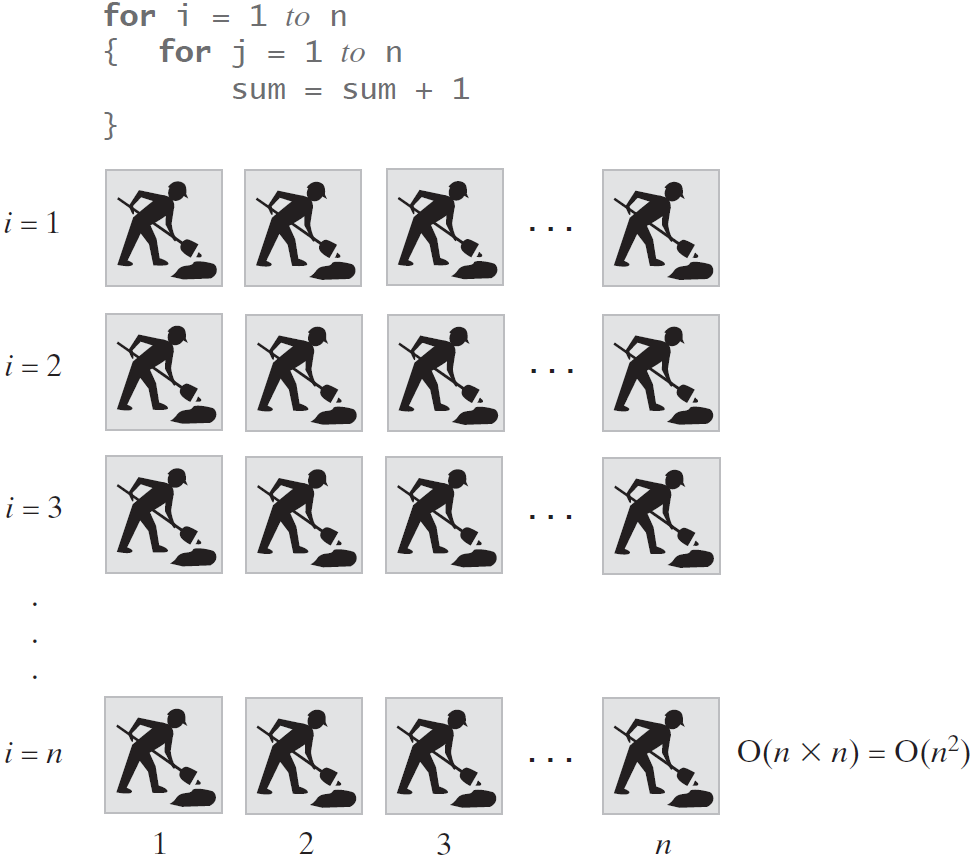
算法时间需求的上界应该尽可能地小。



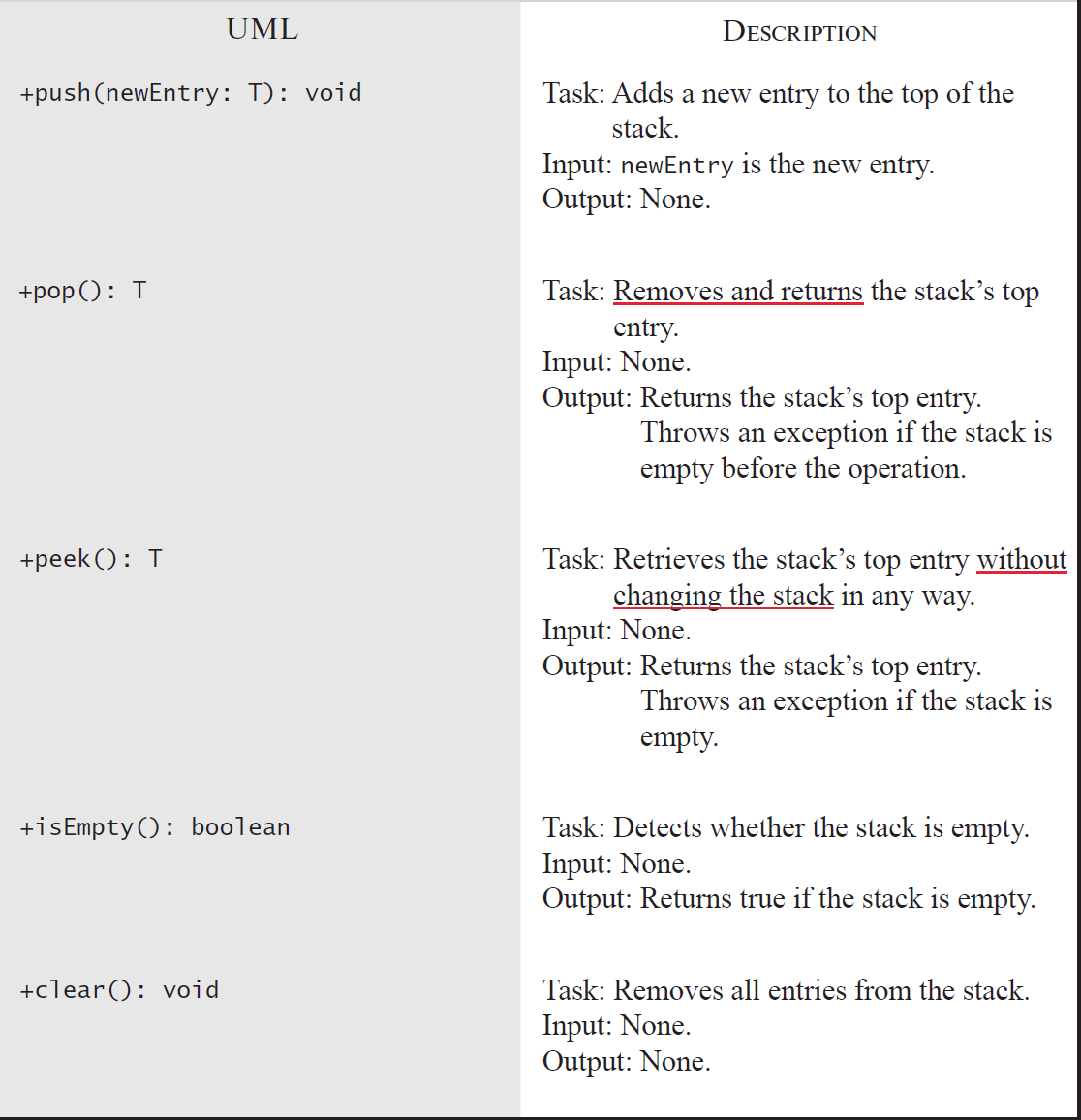








# 栈



## 栈接口

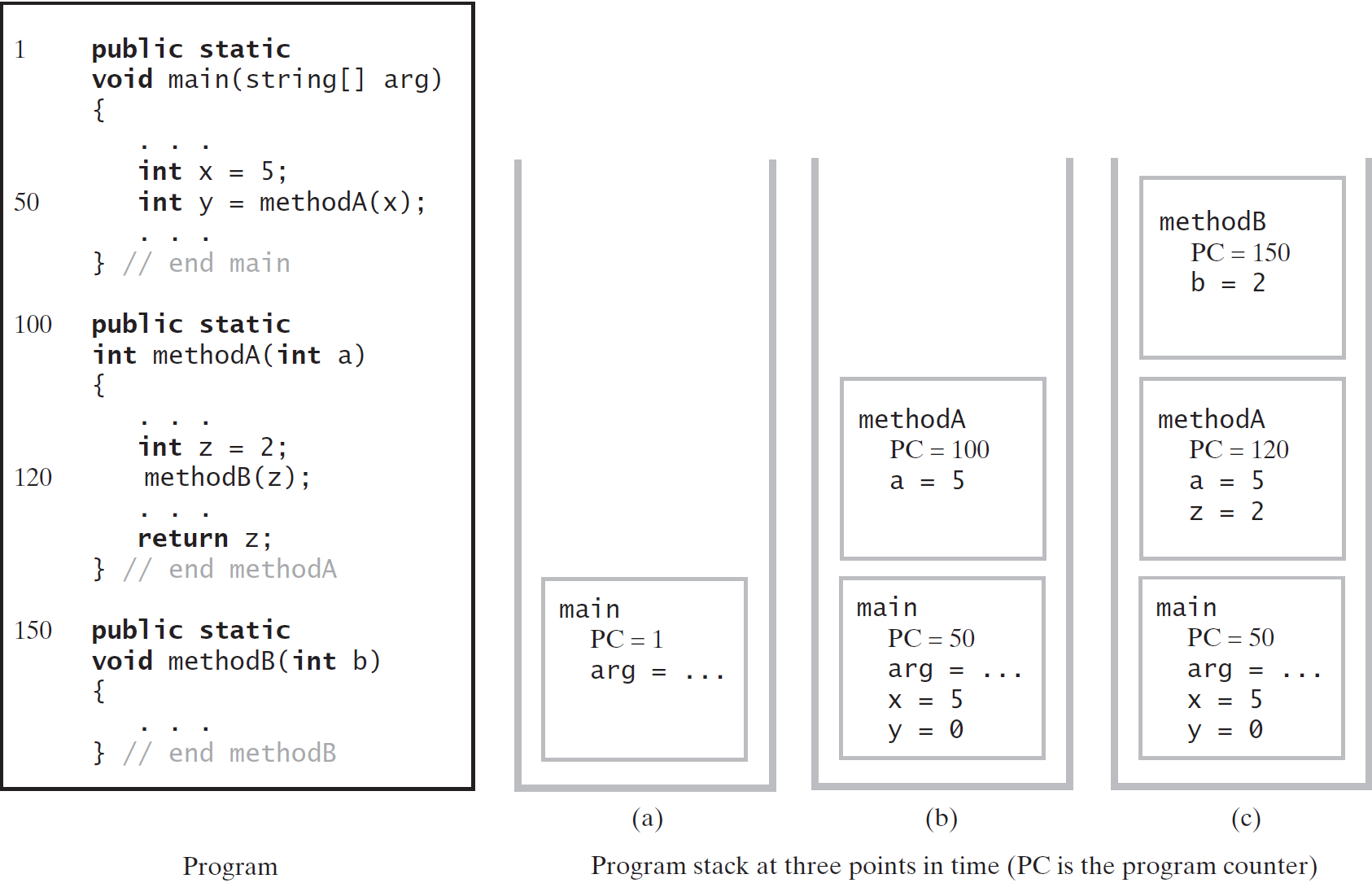


**package** Stack;  
  
**public interface** StackInterface<T>  
{  
 */\*\*  
 \* Adds a new entry to the top of this stack.  
 \** ***@param newEntry*** *An object to be added to the stack.  
 \*/* **public void** push(T newEntry);  
  
  
 */\*\*  
 \* Removes and returns this stack's top entry.  
 \** ***@return*** *The object at the top of the stack.  
 \** ***@throws*** *java.util.EmptyStackException if the stack is empty before the operation.  
 \*/* **public** T pop();  
  
  
 */\*\*  
 \* Retrieves this stack's top entry without changing the stack.  
 \** ***@return*** *The object at the top of the stack.  
 \** ***@throws*** *java.util.EmptyStackException if the stack is empty before the operation.  
 \*/* **public** T peek();  
  
  
 */\*\*  
 \* Detects whether this stack is empty.  
 \** ***@return*** *True if the stack is empty.  
 \*/* **public boolean** isEmpty();  
  
  
 */\*\*  
 \* Removes all entries from this stack.  
 \*/* **public void** clear();  
}

## 程序栈

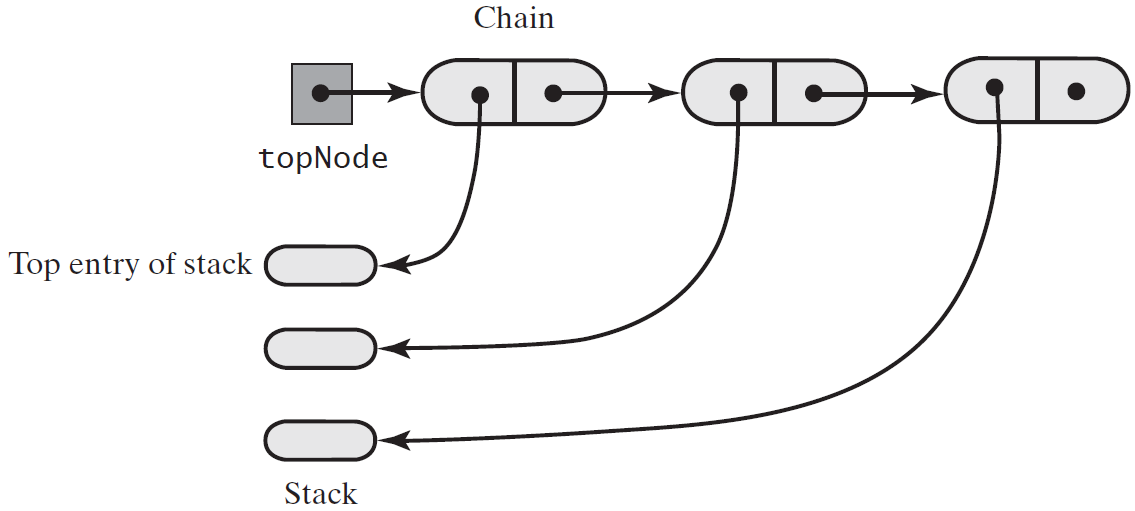
当执行程序时，称为程序计数器(program counter: PC)的一个特殊内存位置指向当前指令。

当调用方法时，程序运行时环境为方法创建一个称为活动记录(activatio record)或框架(frame)的对象。活动记录显示运行期间方法的状态。具体来说，活动记录含有方法的实参、局部变量和指向当前指令的引用——程序计数器的副本。调用方法时，活动记录入栈，这个栈称为程序栈(program stack), 或在Java中称为Java栈(Java stack)。因为一个方法可以调用另一个方法，所以程序栈中常常含有多个活动记录。栈顶的记录属于当前正在运行的方法。刚刚压在栈顶下面的记录属于调用当前方法的方法。



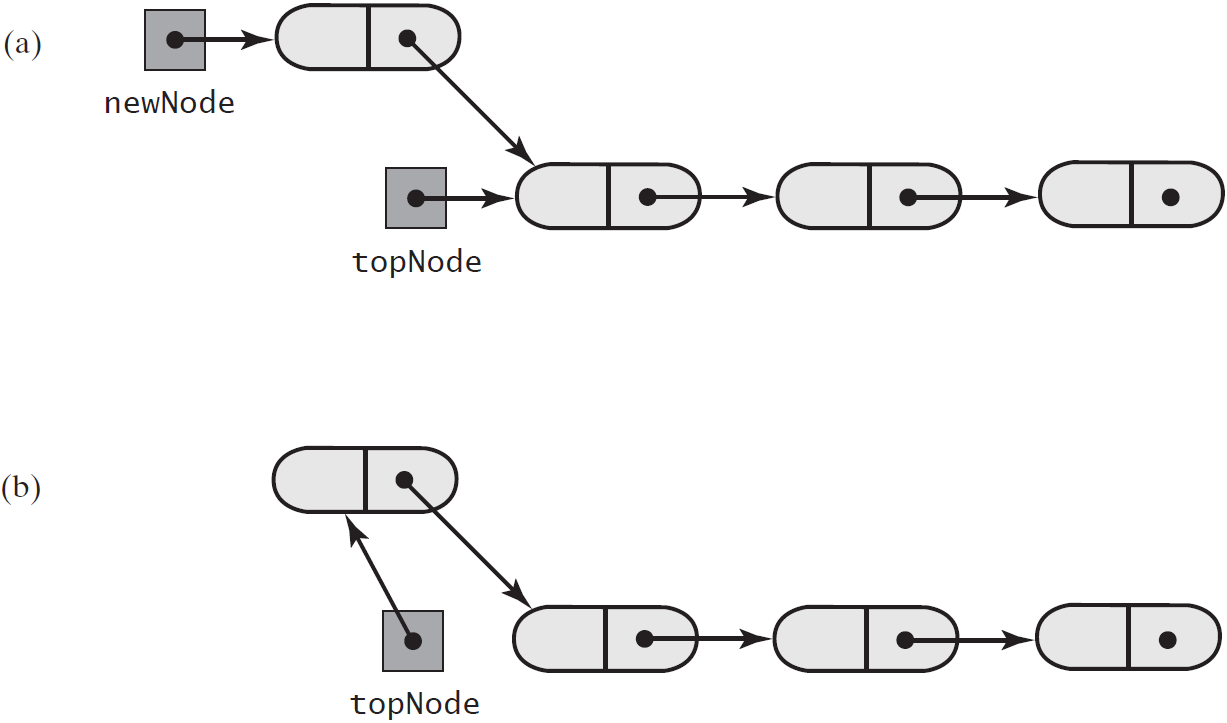
## 链式实现

如果使用结点链实现栈，则首结点应该指向栈顶元素。



### 方法push

**public void** push(T newEntry)  
{  
 Node newNode = **new** Node(newEntry, **topNode**);  
 **topNode** = newNode;  
  
 *// topNode = new Node(newEntry, topNode);*}



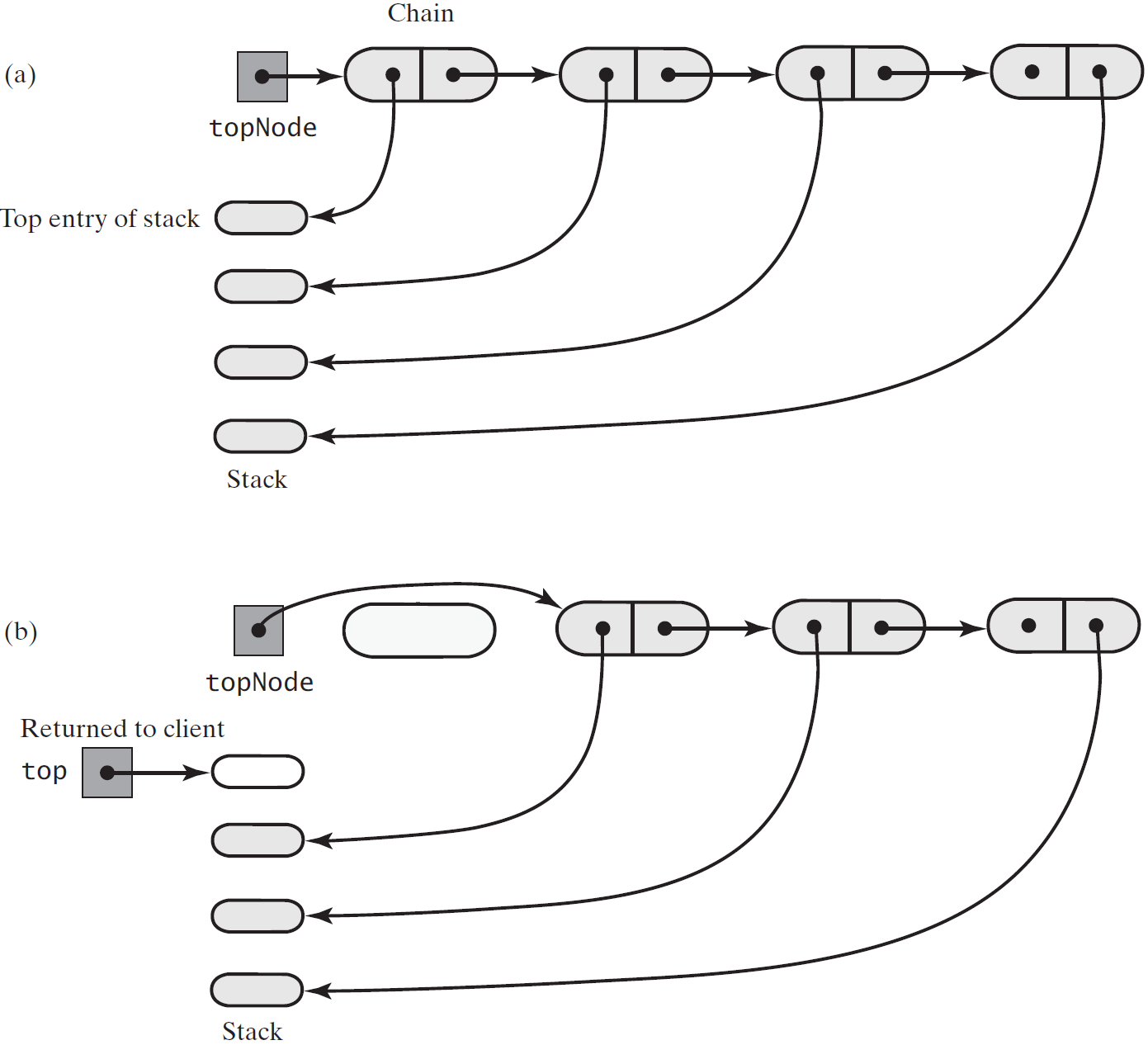
### 方法peek

**public** T peek()  
{  
 **if** (!isEmpty())  
 **return topNode**.getData();  
 **else  
 throw new** EmptyStackException();  
}

### 方法pop

**public** T pop()  
{  
 T top = peek();  
 **assert topNode** != **null**;  
 **topNode** = **topNode**.getNextNode();  
 **return** top;  
}

断言是一个包含布尔表达式的语句，在执行这个语句时假定该表达式为true；如果表达式的值为false，那么系统会报告一个AssertionError。



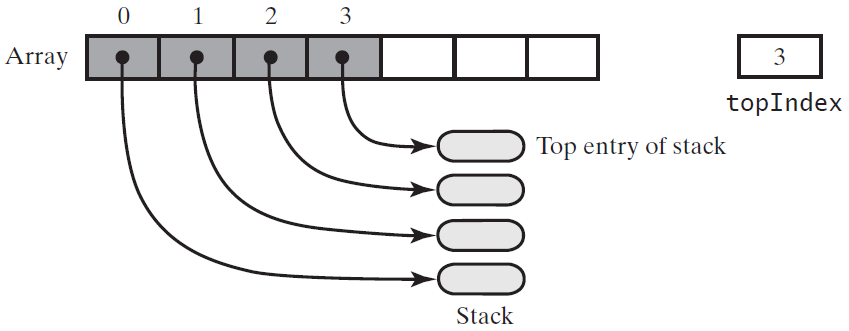
### 完整程序



**package** Stack;  
  
**import** java.util.EmptyStackException;  
  
**public final class** LinkedStack<T> **implements** StackInterface<T>  
{  
 **private** Node **topNode**;  
  
 **public** LinkedStack()  
 {  
 **topNode** = **null**;  
 }  
  
  
 */\*\*  
 \* Removes and returns this stack's top entry.  
 \** ***@return*** *The object at the top of the stack.  
 \** ***@throws*** *java.util.EmptyStackException if the stack is empty before the operation.  
 \*/* **public** T pop()  
 {  
 T top = peek();  
 **assert topNode** != **null**;  
 **topNode** = **topNode**.getNextNode();  
 **return** top;  
 }  
  
  
 */\*\*  
 \* Retrieves this stack's top entry without changing the stack.  
 \** ***@return*** *The object at the top of the stack.  
 \** ***@throws*** *java.util.EmptyStackException if the stack is empty before the operation.  
 \*/* **public** T peek()  
 {  
 **if** (!isEmpty())  
 **return topNode**.getData();  
 **else  
 throw new** EmptyStackException();  
 }  
  
  
 */\*\*  
 \* Adds a new entry to the top of this stack.  
 \** ***@param newEntry*** *An object to be added to the stack.  
 \*/* **public void** push(T newEntry)  
 {  
 Node newNode = **new** Node(newEntry, **topNode**);  
 **topNode** = newNode;  
  
 *// topNode = new Node(newEntry, topNode);* }  
  
  
 */\*\*  
 \* Detects whether this stack is empty.  
 \** ***@return*** *True if the stack is empty.  
 \*/* **public boolean** isEmpty()  
 {  
 **return topNode** == **null**;  
 }  
  
  
 */\*\*  
 \* Removes all entries from this stack.  
 \*/* **public void** clear()  
 {  
 **topNode** = **null**;  
 }  
  
 **private class** Node  
 {  
 **private** T **data**;  
 **private** Node **next**;  
  
 **private** Node(T dataPortion)  
 {  
 **this**(dataPortion, **null**);  
 }  
  
 **private** Node(T dataPortion, Node linkPortion)  
 {  
 **data** = dataPortion;  
 **next** = linkPortion;  
 }  
  
 **private** T getData(){**return data**;}  
 **private void** setData(T newData){**data** = newData;}  
  
 **private** Node getNextNode(){**return next**;}  
 **private void** setNextNode(Node nextNode){**next** = nextNode;}  
 }  
}

## 基于数组的实现

如果使用数组实现栈，则数组的第一个位置是栈底，数组最后占用的位置才指向栈顶元素：



### 在栈顶添加push, ensureCapacity

**public void** push(T newEntry)  
{  
 checkInitialization();  
 ensureCapacity();  
 **stack**[**topIndex** + 1] = newEntry;  
 **topIndex**++;  
}  
  
**private void** ensureCapacity()  
{  
 **if** (**topIndex** == **stack**.**length** - 1) *// If array is full, double its size* {  
 **int** newLength = 2 \* **stack**.**length**;  
 checkCapacity(newLength);  
 **stack** = Arrays.*copyOf*(**stack**, newLength);  
 }  
}

### 获取栈顶peek, isEmpty

**public** T peek()  
{  
 checkInitialization();  
 **if** (isEmpty())  
 **throw new** EmptyStackException();  
 **else  
 return stack**[**topIndex**];  
}

**public boolean** isEmpty()  
{  
 **return topIndex** < 0;  
}

### 删除栈顶pop, clear

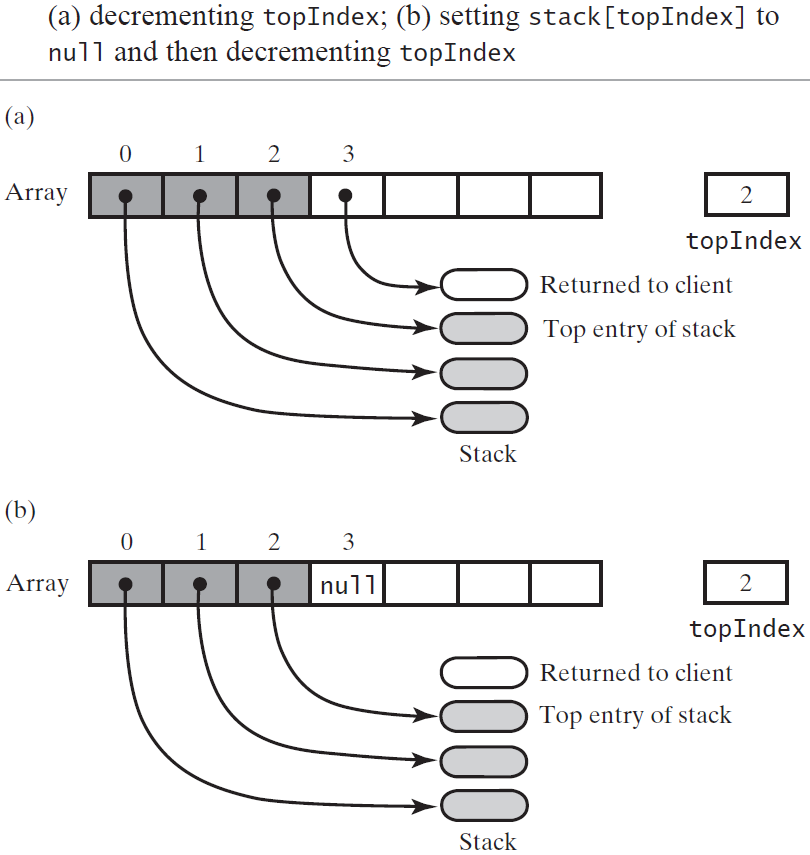
**public** T pop()  
{  
 **if** (isEmpty())  
 **throw new** EmptyStackException();  
 **else** {  
 T top = **stack**[**topIndex**];  
 **stack**[**topIndex**] = **null**;  
 **topIndex**--;  
 **return** top;  
 }  
}

**public void** clear()  
{  
 **while** (!isEmpty())  
 pop();  
}

### 完整程序



**package** Stack;  
  
**import** java.util.Arrays;  
**import** java.util.EmptyStackException;  
  
**public final class** ArrayStack<T> **implements** StackInterface<T>  
{  
 **private** T[] **stack**;  
 **private int topIndex**;  
 **private boolean initialized** = **false**;  
 **private static final int *DEFAULT\_CAPACITY*** = 50;  
 **private static final int *MAX\_CAPACITY*** = 10000;  
  
 **public** ArrayStack()  
 {  
 **this**(***DEFAULT\_CAPACITY***);  
 }  
  
 **public** ArrayStack(**int** initialCapacity)  
 {  
 checkCapacity(initialCapacity);  
  
 @SuppressWarnings(**"unchecked"**)  
 T[] tempStack = (T[]) **new** Object[initialCapacity];  
  
 **stack** = tempStack;  
 **topIndex** = -1;  
 **initialized** = **true**;  
 }  
  
 **private void** checkCapacity(**int** capacity)  
 {  
 **if** (capacity > ***MAX\_CAPACITY***)  
 **throw new** IllegalStateException(**"Exceed the maximum "** + ***MAX\_CAPACITY***);  
 }  
  
 **private void** checkInitialization()  
 {  
 **if** (!**initialized**)  
 **throw new** SecurityException(**"Stack is not initialized properly!"**);  
 }  
  
  
 */\*\*  
 \* Adds a new entry to the top of this stack.  
 \** ***@param newEntry*** *An object to be added to the stack.  
 \*/* **public void** push(T newEntry)  
 {  
 checkInitialization();  
 ensureCapacity();  
 **stack**[**topIndex** + 1] = newEntry;  
 **topIndex**++;  
 }  
  
 **private void** ensureCapacity()  
 {  
 **if** (**topIndex** == **stack**.**length** - 1) *// If array is full, double its size* {  
 **int** newLength = 2 \* **stack**.**length**;  
 checkCapacity(newLength);  
 **stack** = Arrays.*copyOf*(**stack**, newLength);  
 }  
 }  
  
  
 */\*\*  
 \* Retrieves this stack's top entry without changing the stack.  
 \** ***@return*** *The object at the top of the stack.  
 \** ***@throws*** *java.util.EmptyStackException if the stack is empty before the operation.  
 \*/* **public** T peek()  
 {  
 checkInitialization();  
 **if** (isEmpty())  
 **throw new** EmptyStackException();  
 **else  
 return stack**[**topIndex**];  
 }  
  
  
 */\*\*  
 \* Detects whether this stack is empty.  
 \** ***@return*** *True if the stack is empty.  
 \*/* **public boolean** isEmpty()  
 {  
 **return topIndex** < 0;  
 }  
  
  
 */\*\*  
 \* Removes and returns this stack's top entry.  
 \** ***@return*** *The object at the top of the stack.  
 \** ***@throws*** *java.util.EmptyStackException if the stack is empty before the operation.  
 \*/* **public** T pop()  
 {  
 **if** (isEmpty())  
 **throw new** EmptyStackException();  
 **else** {  
 T top = **stack**[**topIndex**];  
 **stack**[**topIndex**] = **null**;  
 **topIndex**--;  
 **return** top;  
 }  
 }  
  
 */\*\*  
 \* Removes all entries from this stack.  
 \*/* **public void** clear()  
 {  
 **while** (!isEmpty())  
 pop();  
 }  
}



## 基于向量的实现

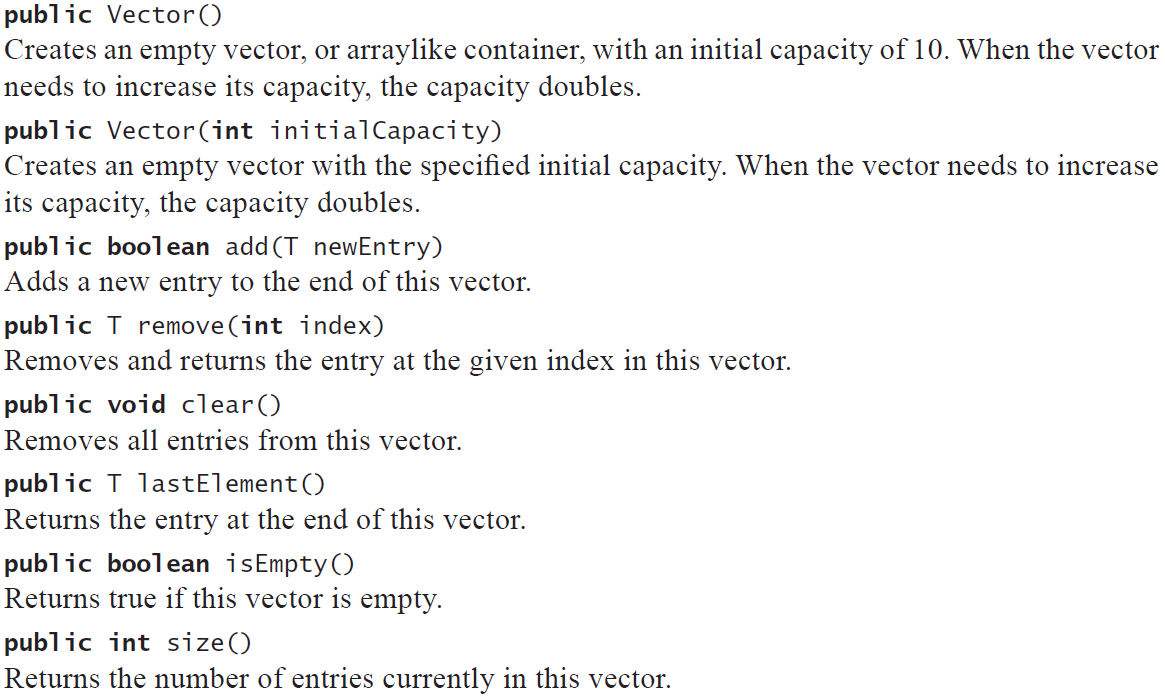
让栈按需增大的一种办法是将元素保存在可变大小的数组中，另一种办法是使用向量(vector)替代数组。

向量是一个对象，其行为类似于高级数组。与数组中的项一样向量中的项也有下标，且从0开始。与数组不同的是，向量有设置或访问项的方法。你可以创建一个指定大小的向量，而且它的大小按需增长。

### Java类库：类Vector

Java类库中包含类Vector, 其实例(称为向量)的行为类似于一个可变大小的数组。

实现ADT栈时会用到的Vector的构造方法和方法：



### 使用向量实现ADT栈

让向量的第一个元素指向栈底元素，不需要维护栈顶元素的下标，但能从向量的大小推出这个下标，而向量的大小很容易得到。因为Vector的实现基于动态可变大小的数组，所以这样实现的栈的性能与基于数组的实现方式是一样的。



**package** Stack;  
  
**import** java.util.EmptyStackException;  
**import** java.util.Vector;  
  
**public final class** VectorStack<T> **implements** StackInterface<T>  
{  
 **private** Vector<T> **stack**;  
 **private boolean initialized** = **false**;  
 **private static final int *DEFAULT\_CAPACITY*** = 50;  
 **private static final int *MAX\_CAPACITY*** = 10000;  
  
 **public** VectorStack()  
 {  
 **this**(***DEFAULT\_CAPACITY***);  
 }  
  
 **public** VectorStack(**int** initialCapacity)  
 {  
 checkCapacity(initialCapacity);  
 **stack** = **new** Vector<>(initialCapacity);  
 **initialized** = **true**;  
 }  
  
 **private void** checkCapacity(**int** capacity)  
 {  
 **if** (capacity > ***MAX\_CAPACITY***)  
 **throw new** IllegalStateException(**"Exceeds the maximum: "** + ***MAX\_CAPACITY***);  
 }  
  
 **public void** push(T newEntry)  
 {  
 checkInitialization();  
 **stack**.add(newEntry); *// We use Vector’s method add to add an entry to the end of the vector, that is, to the top of the stack.* }  
  
 **private void** checkInitialization()  
 {  
 **if** (!**initialized**)  
 **throw new** SecurityException(**"Stack is not initialized."**);  
 }  
  
 **public** T peek()  
 {  
 checkInitialization();  
 **if** (isEmpty())  
 **throw new** EmptyStackException();  
 **else  
 return stack**.lastElement();  
 }  
  
 **public boolean** isEmpty()  
 {  
 **return stack**.isEmpty();  
 }  
  
 **public** T pop()  
 {  
 checkInitialization();  
 **if** (isEmpty())  
 **throw new** EmptyStackException();  
 **else  
 return stack**.remove(**stack**.size() - 1);  
 }  
  
 **public void** clear()  
 {  
 **stack**.clear();  
 }  
}

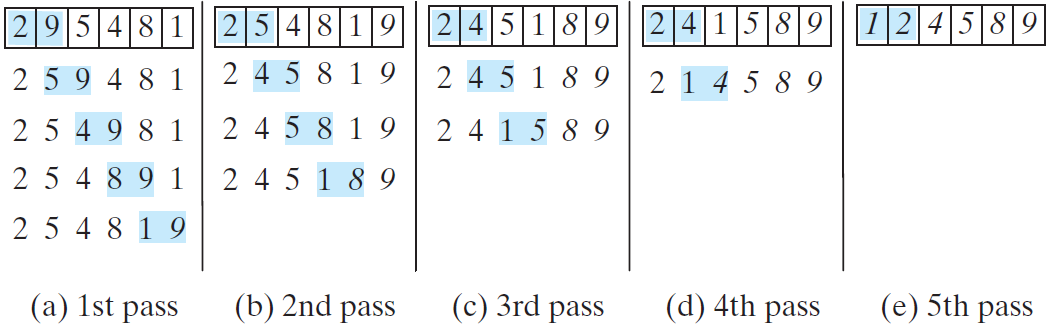
因为Java实现类Vector时使用数组，所以VectorStack是基于数组实现的ADT栈。它使用一个可变大小的数组来保存栈的元素，所以栈可以按需增长。

写VectorStack肯定要比写基于数组实现的代码容易。因为VectorStack的方法调用了Vector的方法，而运行时间要多于ArrayStack方法的运行时间。但是，多出的这些时间常常微不足道。

# 排序

## 冒泡排序

After the first pass, the last element becomes the largest in the array. After the second pass, the second-to-last element becomes the second largest in the array. This process is continued until all elements are sorted.



*// Listing 23.4 BubbleSort.java***public class** BubbleSort  
{  
 **public static void** main(String[] args)  
 {  
 **int**[] myList = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};  
 *bubbleSort*(myList);  
 **for** (**int** i = 0; i < myList.**length**; i ++)  
 System.***out***.print(myList[i] + **" "**);  
 }  
  
 **public static void** bubbleSort(**int**[] list)  
 {  
 **boolean** needNextPass = **true**; // 全部都已经排好了  
 **for** (**int** k = 1; k < list.**length** && needNextPass; k ++) *// 第k次排序* {  
 needNextPass = **false**;  
 **for** (**int** i = 0; i < list.**length** - k; i ++) *//列表末尾有已排好的数，无须再排* {  
 **if** (list[i] > list[i + 1])  
 {  
 **int** temp = list[i];  
 list[i] = list[i + 1];  
 list[i + 1] = temp;  
  
 needNextPass = **true**;   
 }  
 }  
 }  
 }  
}

### 冒泡排序的时间效率

在最坏情况下，冒泡排序的时间效率为。

## 选择排序

对于要排序的数组号数组中的对象必须是Comparable的。所以T表示的类必须实现接口Comparable。

就数组a来说，选择排序找到数组中最小的项，将它与a[0]交换。然后，忽略a [0]，排序找到下一个最小的项并交换到a[1]，以此类推。

### 迭代法

**package** Sort;  
  
*/\*\*  
 \* A class of static, iterative methods for sorting an array of  
 \* Comparable objects from smallest to largest.  
 \*/***public class** SortArray  
{  
 */\*\*  
 \* Sorts the first n objects in an array into ascending order.  
 \** ***@param a*** *An array of Comparable objects.  
 \** ***@param n*** *An integer > 0.  
 \** ***@param <T>*** *\*/* **public static** <T **extends** Comparable<? **super** T>> **void** selectionSort(T[] a, **int** n)  
 {  
 **for** (**int** index = 0; index < n - 1; index++)  
 {  
 **int** indexOfNextSmallest = *getIndexOfSmallest*(a, index, n - 1);  
 *swap*(a, index, indexOfNextSmallest);  
 }  
 }  
  
  
 */\*\*  
 \* Finds the index of the smallest value in a portion of an array a.  
 \* Precondition: a.length > last >= first >= 0.  
 \** ***@param a*** *\** ***@param first*** *\** ***@param last*** *\** ***@param <T>*** *\** ***@return*** *Returns the index of the smallest value among  
 \*/* **private static** <T **extends** Comparable<? **super** T>> **int** getIndexOfSmallest(T[] a, **int** first, **int** last)  
 {  
 T min = a[first];  
 **int** indexOfMin = first;  
 **for** (**int** index = first + 1; index <= last; index++)  
 {  
 **if** (a[index].compareTo(min) < 0)  
 {  
 min = a[index];  
 indexOfMin = index;  
 }  
 }  
 **return** indexOfMin;  
 }  
  
  
 */\*\*  
 \* Swaps the array entries a[i] and a[j].  
 \** ***@param a*** *\** ***@param i*** *\** ***@param j*** *\*/* **private static void** swap(Object[] a, **int** i, **int** j)  
 {  
 Object temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
}

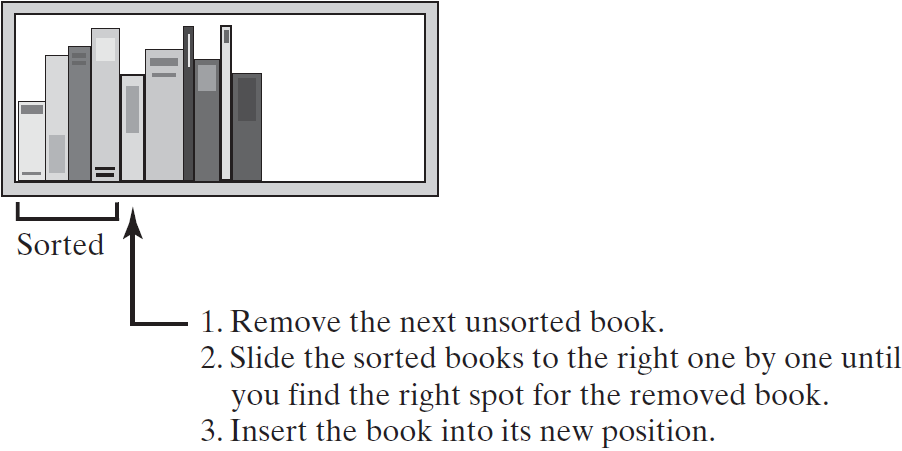
### 递归法

**public static** <T **extends** Comparable<? **super** T>> **void** RecursiveSelectionSort(T[] a, **int** n)  
{  
 *recursiveSelectionSort*(a, 0, n - 1);  
}  
  
**private static** <T **extends** Comparable<? **super** T>> **void** recursiveSelectionSort(T[] a, **int** low, **int** high)  
{  
 **if** (low < high)  
 {  
 **int** indexOfNextSmallest = *getIndexOfSmallest*(a, low, high);  
 *swap*(a, low, indexOfNextSmallest);  
 *recursiveSelectionSort*(a, low + 1, high);  
 }  
}

### 选择排序的效率

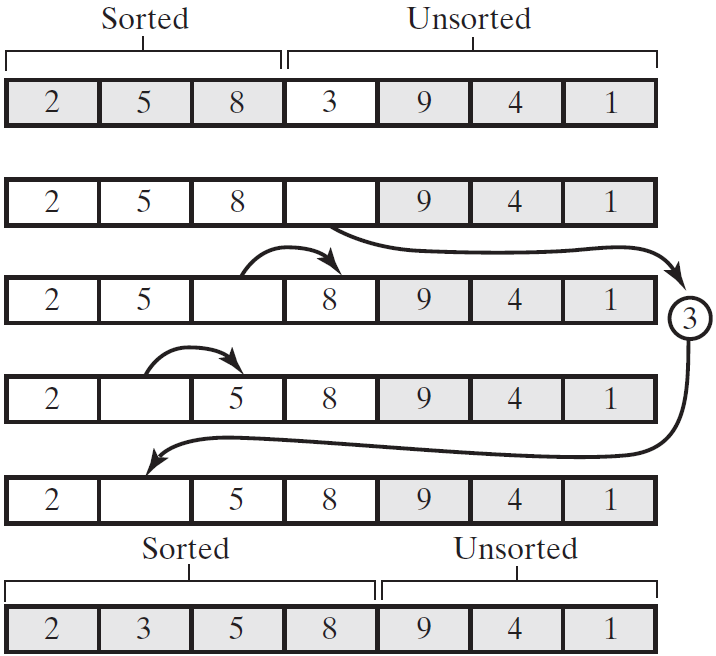
在迭代方法SelectionSort中，for循环执行n-1次，所以它分别调用*getIndexOfSmallest*和*swap*方法各n-1次。在n-1次调用*getIndexOfSmallest*中，last是n-1, 而first从0变到n-2。每次调用*getIndexOfSmallest*时，它的循环执行last-first次。因为last-first从(n-1)-0(即n-1)变到(n-1)-(n-2)(即1)，所以这个循环将总共执行(n-1)+(n-2)+……+1次。因为循环中的每个操作都是，所以选择排序时间复杂度是，但仅执行次交换。

## 插入排序



### 迭代法

对数组的插入排序将数组分隔(partition) (即划分)为两部分。第一部分是有序的，初始时仅含有数组中的第一项。第二部分含有其余的项。算法从未排序部分移走第一项，并将它插入有序部分中合适的有序位置。，从有序部分的末尾开始，朝着开头方向，通过将待排序项与各有序项进行比较来选择合适的位置。当比较时，将有序部分的数组项右移，为插入腾出空间。

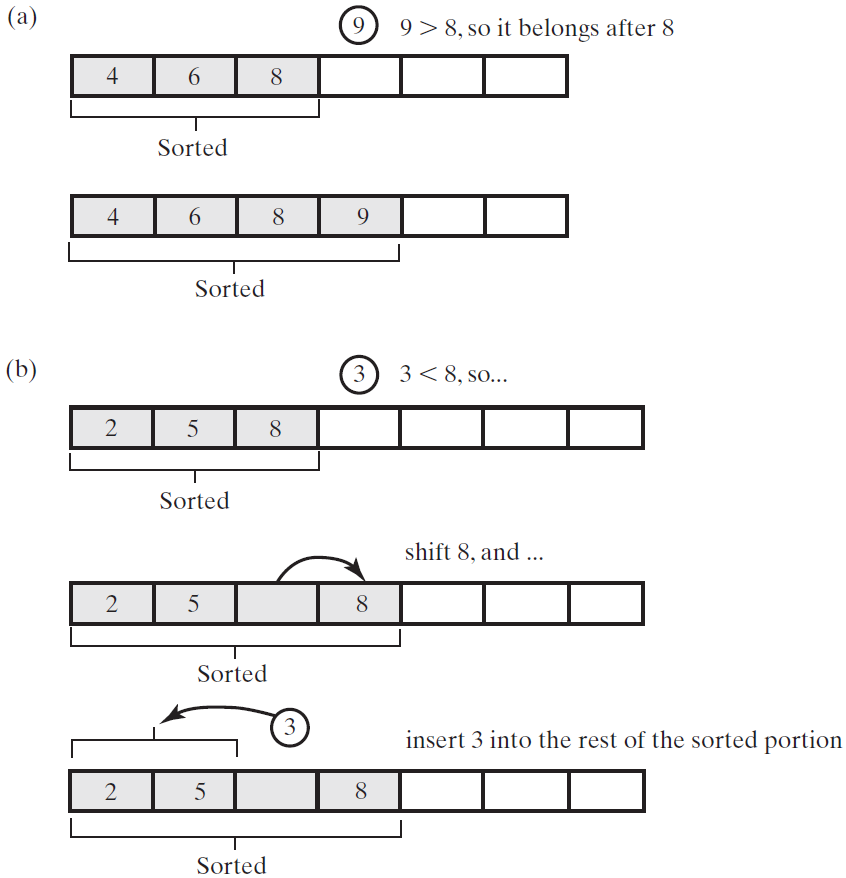


**public class** InsertionSort  
{  
 **public static void** main(String[] args)  
 {  
 **int**[] myList = {1, 9, 4, 6, 5, -4};  
 *insertSort*(myList);  
 **for** (**int** i : myList)  
 System.***out***.print(i + **" "**);  
 }  
  
 **private static void** insertSort(**int**[] list)  
 {  
 **for** (**int** i = 1; i < list.**length**; i++)  
 {  
 **int** currentValue = list[i];  
 **int** j;  
 **for** (j = i - 1; j >=0 && currentValue < list[j]; j--)  
 {  
 list[j + 1] = list[j]; *//把数往后移* }  
 *// 跳出for循环时，指针指向的是不满足条件的那个数* list[j + 1] = currentValue;  
 }  
 }  
}

### 递归法

**public static** <T **extends** Comparable<? **super** T>> **void** recursiveInsertionSort(T[] a, **int** first, **int** last)  
{  
 **if** (first < last)  
 {  
 *// Sort all but the last entry  
 recursiveSelectionSort*(a, first, last - 1);  
 *// Insert the last entry in sorted order* insertInOrder(a[last], a, first, last - 1);  
 }  
}

如果要插入的项大于或等于数组有序部分的最后项，则这个插入项就放在最后项的后面；否则，我们将有序部分的最后项移到数组中下一个更高的位置，并将插入项插入剩余部分中。



**public static** <T **extends** Comparable<? **super** T>> **void** recursiveInsertionSort(T[] a, **int** first, **int** last)  
{  
 **if** (first < last)  
 {  
 *// Sort all but the last entry  
 recursiveSelectionSort*(a, first, last - 1);  
 *// Insert the last entry in sorted order  
 insertInOrder*(a[last], a, first, last - 1);  
 }  
}  
  
**private static** <T **extends** Comparable<? **super** T>> **void** insertInOrder(T anEntry, T[] a, **int** begin, **int** end)  
{  
 **if** (anEntry.compareTo(a[end]) >= 0)  
 a[end + 1] = anEntry;  
 **else if** (begin < end)  
 {  
 a[end + 1] = a[end];  
 *insertInOrder*(anEntry, a, begin, end - 1);  
 }  
 **else** *// begin == end && anEntry < a[end]* {  
 a[end + 1] = a[end];  
 a[end] = anEntry;  
 }  
}

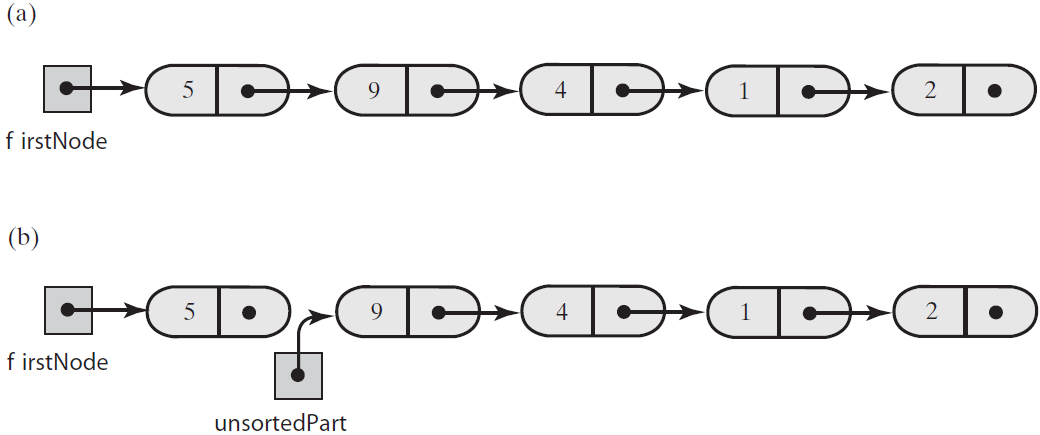
### 插入排序的效率

插入排序时间复杂度是。数组越接近有序，插入排序要做的工作越少。

### 链式结点的插入排序

对链式结点链进行排序可能是困难的，但插入排序提供了一个完成这个任务的合理方法。

将链划分为两部分。第一部分是有序的，初始时它仅含有第一个结点。第二部分是无序的，初始时它含有链中其余的结点。





**package** Sort;  
  
**public class** LinkedGroup<T **extends** Comparable<? **super** T>>  
{  
 **private** Node **firstNode**;  
 **int length**; *// Number of objects in the group* **private void** insertInOrder(Node nodeToInsert)  
 {  
 T item = nodeToInsert.getData();  
 Node currentNode = **firstNode**;  
 Node previousNode = **null**;  
  
 *// Locate insertion point* **while** ((currentNode != **null**) && (item.compareTo(currentNode.getData()) > 0))  
 {  
 previousNode = currentNode;  
 currentNode = currentNode.getNextNode();  
 }  
  
 *// Make the insertion* **if** (previousNode != **null**)  
 {  
 *// Insert between previousNode and currentNode* previousNode.setNextNode(nodeToInsert);  
 nodeToInsert.setNextNode(currentNode);  
 }  
 **else** *// Insert at beginning* {  
 nodeToInsert.setNextNode(**firstNode**);  
 **firstNode** = nodeToInsert;  
 }  
 }  
  
 **private void** insertionSort()  
 {  
 *// If fewer than two items are in the list, there is nothing to do* **if** (**length** > 1)  
 {  
 **assert firstNode** != **null**;  
  
 *// Break chain into 2 pieces: sorted and unsorted* Node unsortedPart = **firstNode**.getNextNode();  
 **assert** unsortedPart != **null**;  
 **firstNode**.setNextNode(**null**);  
  
 **while** (unsortedPart != **null**)  
 {  
 Node nodeToInsert = unsortedPart;  
 unsortedPart = unsortedPart.getNextNode();  
 insertInOrder(nodeToInsert);  
 }  
 }  
 }  
  
 **private class** Node  
 {  
 **private** T **data**;  
 **private** Node **next**;  
  
 **private** Node(T dataPortion)  
 {  
 **this**(dataPortion, **null**);  
 }  
  
 **private** Node(T dataPortion, Node nextNode)  
 {  
 **data** = dataPortion;  
 **next** = nextNode;  
 }  
  
 **private** T getData(){**return data**;}  
 **private void** setData(T newData){**data** = newData;}  
  
 **private** Node getNextNode(){**return next**;}  
 **private void** setNextNode(Node nextNode){**next** = nextNode;}  
 }  
}

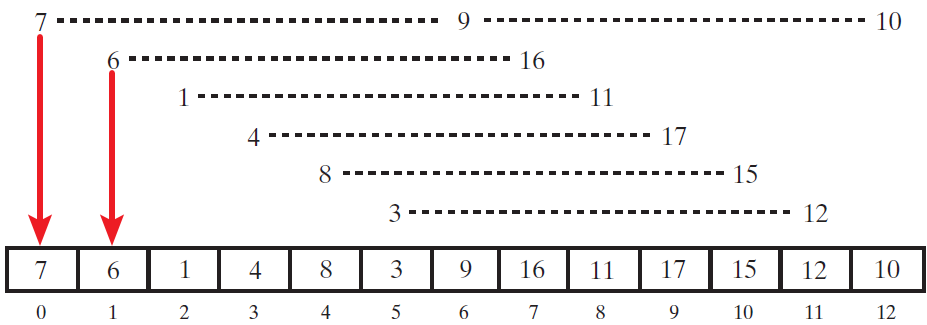
## 希尔Shell排序

希尔排序是插入排序的变体。在插入排序过程中，数组项只移动到相邻位置。当项与正确的有序位置相距甚远时，它必须进行很多次这样的移动。所以当数组完全无序时，插入排序要花很多时间。但当数组基本有序时，插入排序有很好的效率。

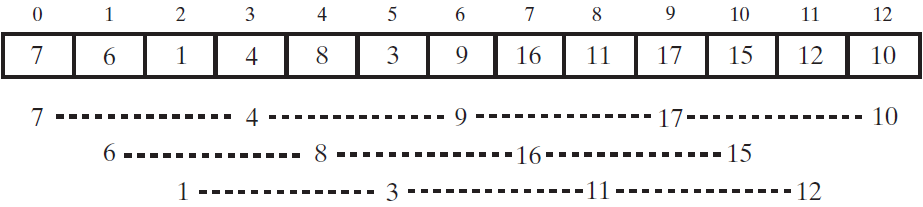
下图显示了一个数组及每隔5项组成的子数组。第一个子数组含有整数10、9和7；第二个子数组含有16和6；等等。



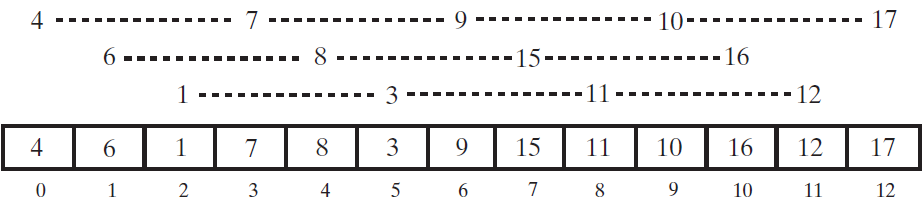
现在使用插入排序分别对这6个子数组进行排序，排序后数组比原始状态“更有序”了。



现在形成新的子数组，这次减小下标之间的间隔。Shell建议下标间的初始间隔是n/2，且每趟排序中这个值减半直到为1。示例数组有13项，所以从间隔为6开始。现在将间隔减小到3。



对得到的三个子数组进行希尔排序：



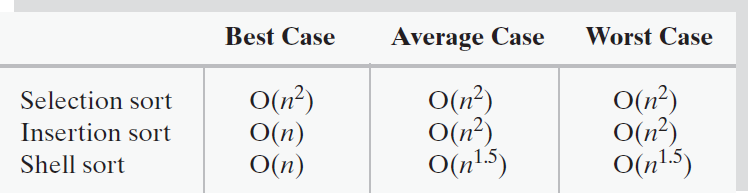
将当前间隔3除以2得到l。所以最后一步只是对整个数组进行普通的插入排序。

如果你使用的是任何下标间隔，只要最后一个是1，希尔排序就都能奏效。但不是任何序列都能使希尔排序有高效率。

**public static** <T **extends** Comparable<? **super** T>> **void** shellSort(T[] a, **int** first, **int** last)  
{  
 **int** n = last - first + 1; *// number of array entries* **int** space = n / 2;  
 **while** (space > 0)  
 {  
 **for** (**int** begin = first; begin < first + space - 1; begin++) *// 分了几个子数组  
 incrementalInsertionSort*(a, begin, last, space);  
 space /= 2;  
 }  
}  
  
**private static** <T **extends** Comparable<? **super** T>> **void** incrementalInsertionSort(T[] a, **int** first, **int** last, **int** space)  
{  
 *// 对每个子数组进行插入排序* **for** (**int** unsorted = first + space; unsorted <= last; unsorted += space)  
 {  
 T nextToInsert = a[unsorted];  
 **int** index = unsorted - space;  
 **while** ((index >= first) && (nextToInsert.compareTo(a[index]) < 0))  
 {  
 a[index + space] = a[index];  
 index -= space;  
 }  
 a[index + space] = nextToInsert;  
 }  
}

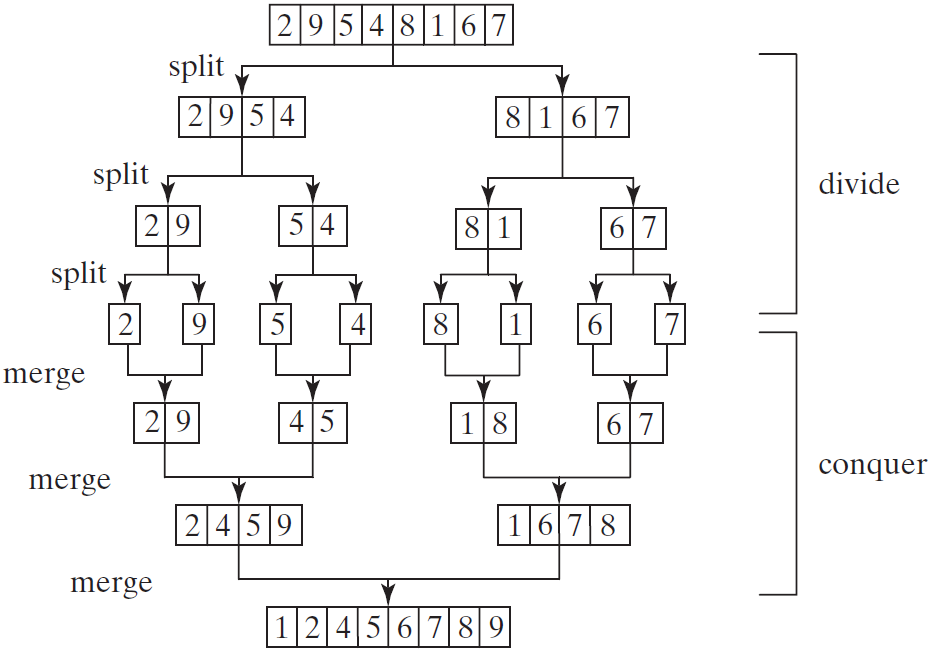
### 希尔排序的效率

当space为偶数时，将其加1，则最坏情形可以改进为。



## 归并排序

归并排序算法将数组分为两半，对每部分递归地应用归并排序。在两部分都排好序后，对它们进行归并。

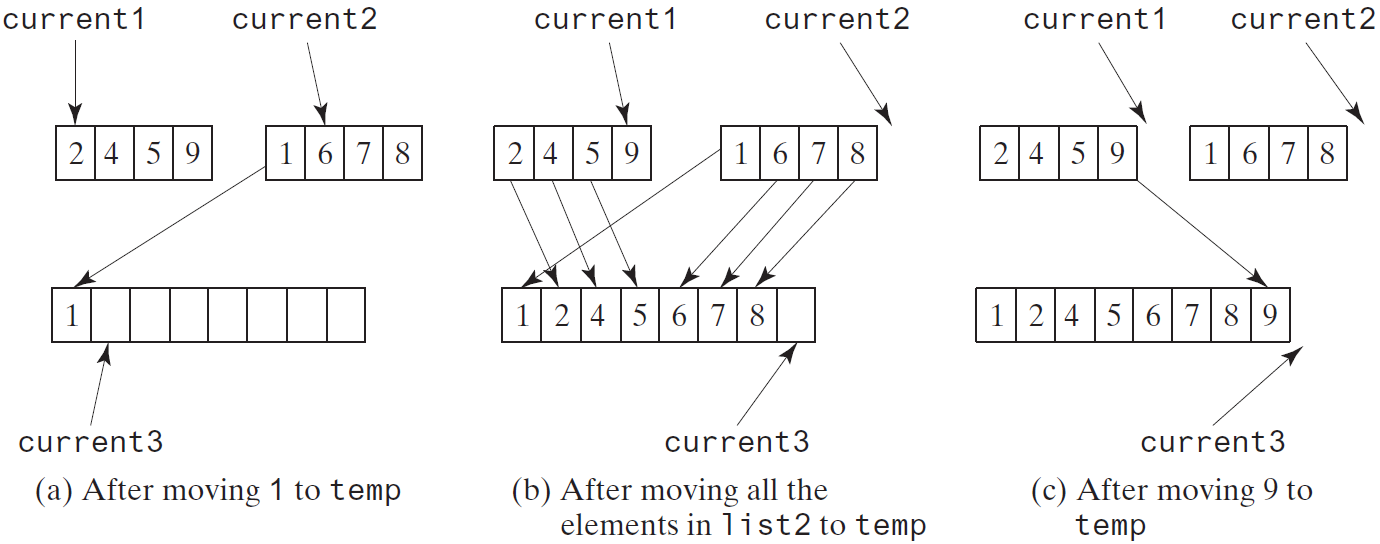


归并排序使用分而治之的方法对数组排序！

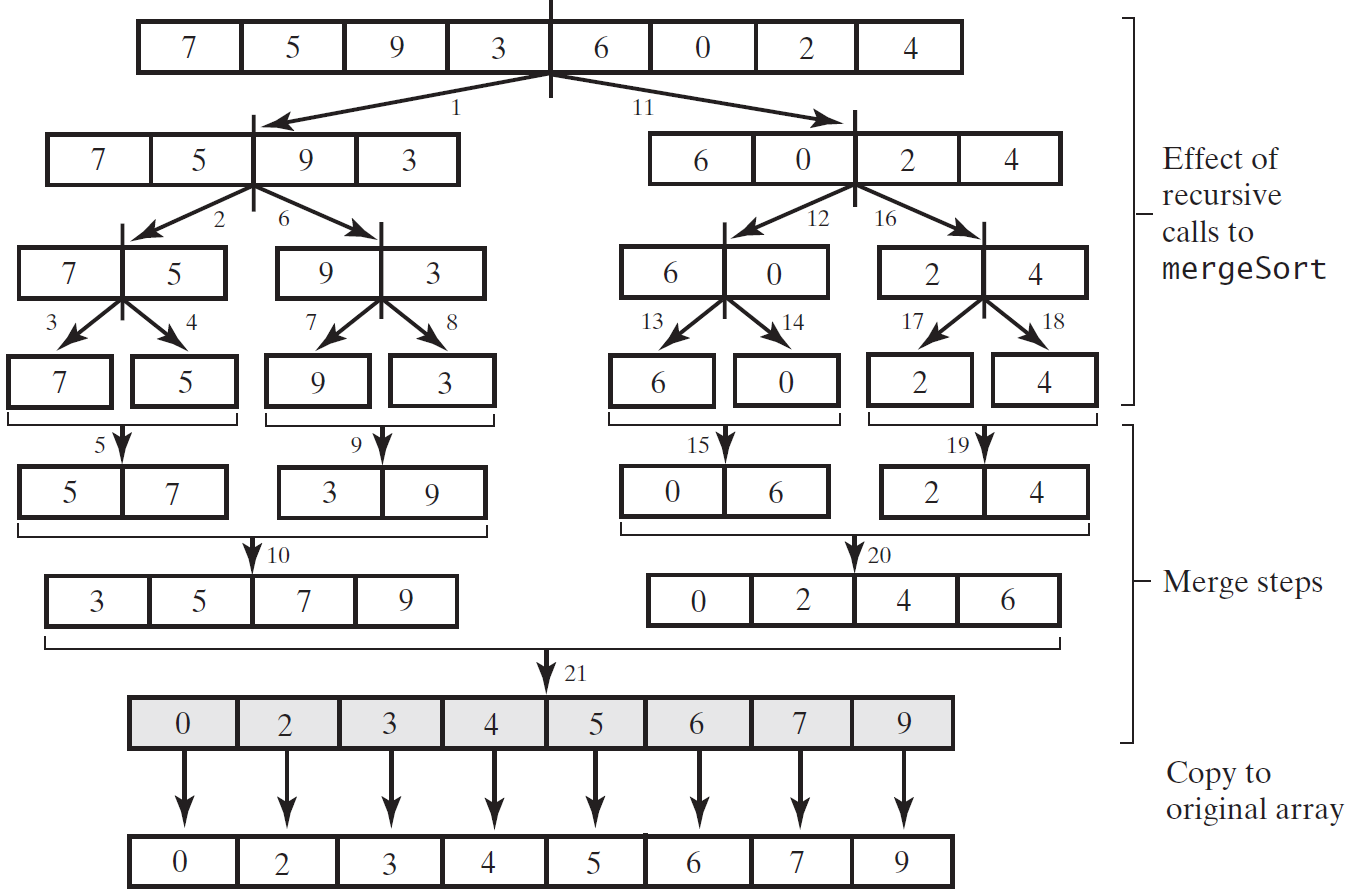
递归调用持续将数组划分为子数组，直到每个子数组只包含一个元素。然后，该算法将这些小的子数组归并为稍大的有序子数组，直到最后形成一个有序的数组。

### 将两个有序数组归并为一个有序数组

**private static void** merge(**int**[] list1, **int**[] list2, **int**[] temp)  
{  
 **int** current1 = 0; *// Current index in list1* **int** current2 = 0; *// Current index in list2* **int** current3 = 0; *// Current index in temp* **while** (current1 < list1.**length** && current2 < list2.**length**)  
 {  
 **if** (list1[current1] < list2[current2])  
 temp[current3++] = list1[current1++];  
 **else** temp[current3++] = list2[current2++];  
 }  
  
 **while** (current1 < list1.**length**)  
 temp[current3++] = list1[current1++];  
  
 **while** (current2 < list2.**length**)  
 temp[current3++] = list2[current2++];  
}



箭头上的数字表示递归调用及合并的次序：





**package** Sort;  
  
**public class** MergeSort  
{  
 **public static void** main(String[] args)  
 {  
 **int**[] list = {7, 5, 9, 3, 6, 0, 2, 4};  
 *mergeSort*(list);  
 **for** (**int** i = 0; i < list.**length**; i++)  
 System.***out***.print(list[i] + **" "**);  
 }  
  
 **private static void** mergeSort(**int**[] list)  
 {  
 **if** (list.**length** > 1)  
 {  
 *// divide the first half* **int**[] firstHalf = **new int**[list.**length** / 2];  
 System.*arraycopy*(list, 0, firstHalf, 0, list.**length** / 2);  
 *mergeSort*(firstHalf);  
  
 *// divide the second half* **int** secondHalfLength = list.**length** - list.**length** / 2;  
 **int**[] secondHalf = **new int**[secondHalfLength];  
 System.*arraycopy*(list, list.**length** / 2, secondHalf, 0, secondHalfLength);  
 *mergeSort*(secondHalf);  
  
 *// Merge firstHalf with secondHalf into a list  
 merge*(firstHalf, secondHalf, list);  
 }  
 }  
  
  
 **private static void** merge(**int**[] list1, **int**[] list2, **int**[] temp)  
 {  
 **int** current1 = 0; *// Current index in list1* **int** current2 = 0; *// Current index in list2* **int** current3 = 0; *// Current index in temp* **while** (current1 < list1.**length** && current2 < list2.**length**)  
 {  
 **if** (list1[current1] < list2[current2])  
 temp[current3++] = list1[current1++];  
 **else** temp[current3++] = list2[current2++];  
 }  
  
 **while** (current1 < list1.**length**)  
 temp[current3++] = list1[current1++];  
  
 **while** (current2 < list2.**length**)  
 temp[current3++] = list2[current2++];  
 }  
}

### 归并排序的效率

归并排序的时间复杂度为，优于冒泡排序、选择排序和插入排序。java.util.Arrays类中的sort方法是使用归并排序算法的变体来实现的。

## 快速排序

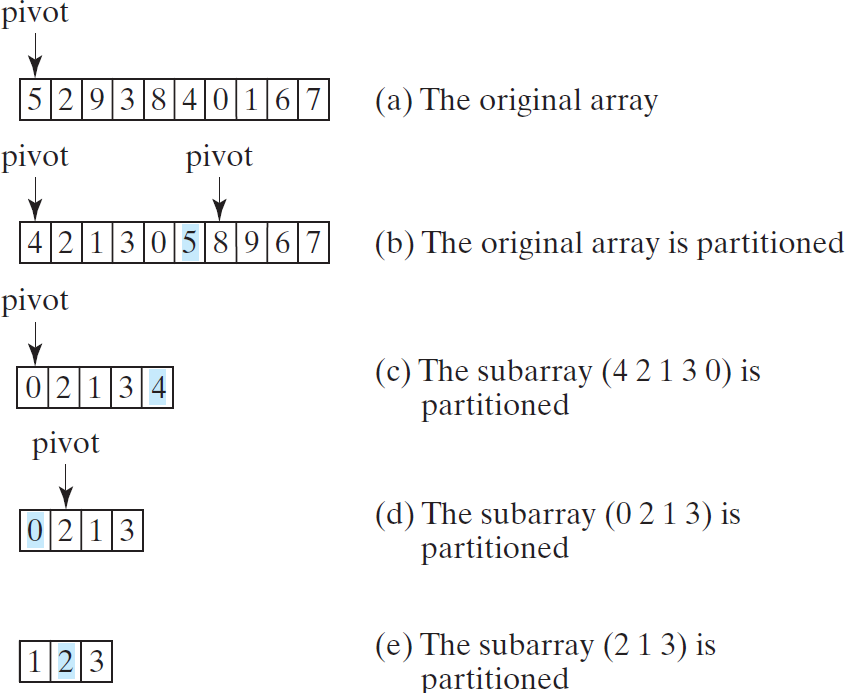
快速排序工作机制如下，该算法在数组中选择一个称为主元(pivot)的元素，将数组分为两部分，使得第一部分中的所有元素都小于或等于主元，而笫二部分中的所有元素都大于主元。对第一部分递归地应用快速排序算法，然后对笫二部分递归地应用快速排序算法。

主元的选择会影响算法的性能。在理想情况下，应该选择能平均划分两部分的主元。为了简单起见，假定将数组的第一个元素选为主元。



根据上图可得：

**private static int** partition(**int**[] list, **int** first, **int** last)  
{  
 **int** pivot = list[first];  
 **int** low = first + 1;  
 **int** high = last;  
  
 **while** (high > low)  
 {  
 **while** (low <= high && list[low] <= pivot)  
 low++;  
 **while** (low <= high && list[high] > pivot)  
 high++;  
  
 **if** (high > low)  
 {  
 **int** temp = list[high];  
 list[high] = list[low];  
 list[low] = temp;  
 }  
 }  
  
 *// 此时high = low* **while** (high > first && list[high] >= pivot)  
 high--;  
  
 **if** (pivot > list[high])  
 {  
 list[first] = list[high];  
 list[high] = pivot;  
 **return** high;  
 }  
 **else  
 return** first;  
}





**package** Sort;  
  
**public class** QuickSort  
{  
 **public static void** main(String[] args)  
 {  
 **int**[] list = {7, 5, 9, 3, 6, 0, 2, 4};  
 *quickSort*(list);  
 **for** (**int** i = 0; i < list.**length**; i++)  
 System.***out***.print(list[i] + **" "**);  
 }  
  
 **private static void** quickSort(**int**[] list)  
 {  
 *quickSort*(list, 0, list.**length** - 1);  
 }  
  
 **private static void** quickSort(**int**[] list, **int** first, **int** last)  
 {  
 **if** (last > first)  
 {  
 **int** pivotIndex = *partition*(list, first, last);  
 *quickSort*(list, first, pivotIndex - 1);  
 *quickSort*(list, pivotIndex + 1, last);  
 }  
 }  
  
 **private static int** partition(**int**[] list, **int** first, **int** last)  
 {  
 **int** pivot = list[first];  
 **int** low = first + 1;  
 **int** high = last;  
  
 **while** (high > low)  
 {  
 **while** (low <= high && list[low] <= pivot)  
 low++;  
 **while** (low <= high && list[high] > pivot)  
 high--;  
  
 **if** (high > low)  
 {  
 **int** temp = list[high];  
 list[high] = list[low];  
 list[low] = temp;  
 }  
 }  
  
 *// 此时high = low* **while** (high > first && list[high] >= pivot)  
 high--;  
  
 **if** (pivot > list[high])  
 {  
 list[first] = list[high];  
 list[high] = pivot;  
 **return** high;  
 }  
 **else  
 return** first;  
 }  
}

### 快速排序的效率

归并排序和快速排序都使用了分而治之法。对于归并排序，大量的工作是将两个子线性表进行归并，归并是在子线性表都排好序后进行的。对于快速排序，大量的工作是将线性表划分为两个子线性表，划分是在子线性表排好序前进行的。

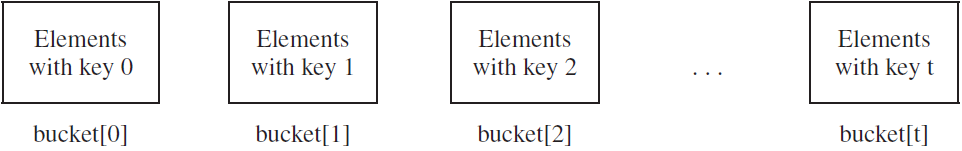
在最差情况下，归并排序的效率高于快速排序，但是，在平均情况下，两者的效率相同。

归并排序在归并两个子数组时需要一个临时数组，而快速排序不需要额外的数组空间。因此，快速排序的空间效率高于归并排序。

## 桶排序和基数排序Bucket and Radix Sorts

目前所讨论的所有排序算法都是可以用在任何键值类型(例如，整数、字符串以及任何可比较的对象)上的通用排序算法。这些箕法都是通过比较它们的键值来对元素排序的。已经证明，基于比较的排序算法的复杂度不会好于。但是，如果键值是整数，那么可以使用桶排序，而无须比较这些键值。

假设键值的范围是从0到t。我们需要t+1个标记为0, 1,…, t的桶。如果元素的键值是i，那么就将该元素放入桶i中。每个桶中都放着具有相同键值的元素。

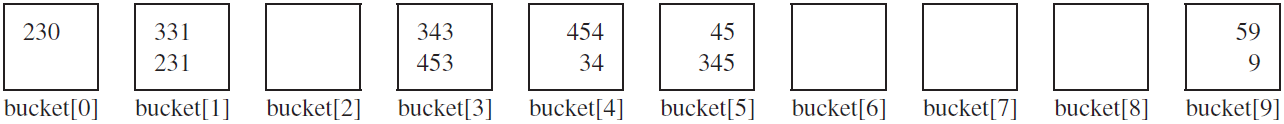


可以使用ArrayList来实现一个桶：

假定键值是正整数。基数排序(radix sort)的思路就是将这些键值基于它们的基数位置分为子组。然后反复地从最小的基数位置开始，对其上的键值应用桶排序。

331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9

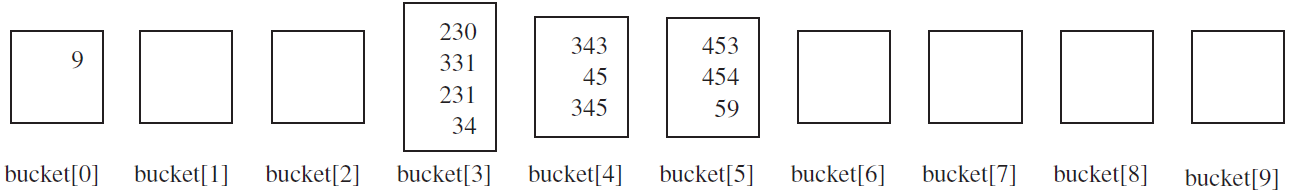
在最后一位基数位置上应用桶排序。这些元素被按如下方式放在桶中：



将元素从桶中删除之后，它们以下面的顺序排列：



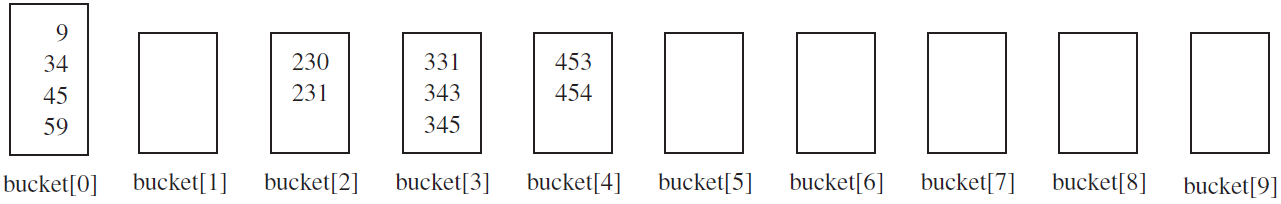
在倒数第二位基数位置上应用桶排序。这些元素被按如下方式放在桶中：



将元素从桶中删除之后，它们以下面的顺序排列：



在倒数第三位基数位置上应用桶排序。这些元素被按如下方式放在桶中：



将元素从桶中删除之后，它们以下面的顺序排列：





**package** Sort;  
  
**import** java.util.Arrays;  
  
**public class** RadixSort  
{  
 **public int**[] sort(**int**[] sourceArray)  
 {  
 *// 对 arr 进行拷贝，不改变参数内容* **int**[] arr = Arrays.*copyOf*(sourceArray, sourceArray.**length**);  
 **int** maxDigit = getMaxDigit(arr);  
 **return** radixSort(arr, maxDigit);  
 }  
  
  
 */\*\*  
 \* 获取最大的数有几位  
 \** ***@param arr*** *\** ***@return*** *\*/* **private int** getMaxDigit(**int**[] arr)  
 {  
 **int** maxValue = getMaxValue(arr);  
  
 **if** (maxValue == 0)  
 **return** 1;  
 **else** {  
 **int** length = 0;  
 **for** (**int** temp = maxValue; temp != 0; temp /= 10)  
 length++;  
 **return** length;  
 }  
 }  
  
  
 */\*\*  
 \* 获取最大的值  
 \** ***@param arr*** *\** ***@return*** *\*/* **private int** getMaxValue(**int**[] arr)  
 {  
 **int** maxValue = arr[0];  
 **for** (**int** value : arr)  
 **if** (maxValue < value)  
 maxValue = value;  
 **return** maxValue;  
 }  
  
  
 */\*\*  
 \* 基数排序  
 \** ***@param arr*** *\** ***@param maxDigit*** *\** ***@return*** *\*/* **private int**[] radixSort(**int**[] arr, **int** maxDigit)  
 {  
 **int** mod = 10;  
 **int** dev = 1;  
 **for** (**int** i = 0; i < maxDigit; i++, dev \*= 10, mod \*= 10)  
 {  
 *// 考虑负数的情况，这里扩展一倍队列数，其中 [0-9]对应负数，[10-19]对应正数 (bucket + 10)* **int**[][] bucket = **new int**[mod\*2][0]; *// 创建了长度为20的桶，来存放0-19  
  
 // 将数放入桶中* **for** (**int** j = 0; j < arr.**length**; j++)  
 {  
 **int** index = ((arr[j] % mod) /dev) +mod;  
 bucket[index] = arrayAppend(bucket[index], arr[j]); *// 桶中的一个位置可能会放多个数* }  
  
 **int** pos = 0;  
 **for** (**int**[] index : bucket)  
 **for** (**int** value : index)  
 arr[pos++] = value;  
 }  
  
 **return** arr;  
 }  
  
  
 */\*\*  
 \* 将数放入桶中的对应位置，自动扩容，保存数字  
 \** ***@param arr*** *\** ***@param value*** *\** ***@return*** *\*/* **private int**[] arrayAppend(**int**[] arr, **int** value)  
 {  
 arr = Arrays.*copyOf*(arr, arr.**length** + 1);  
 arr[arr.**length** - 1] = value;  
 **return** arr;  
 }  
}



**package** Sort;  
  
**public class** Test  
{  
 **public static void** main(String[] args)  
 {  
 **int**[] arr = {331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9};  
 RadixSort radixSort = **new** RadixSort();  
 arr = radixSort.sort(arr);  
 **for** (**int** i : arr)  
 System.***out***.print(i + **" "**);  
 }  
}

### 基数排序效率

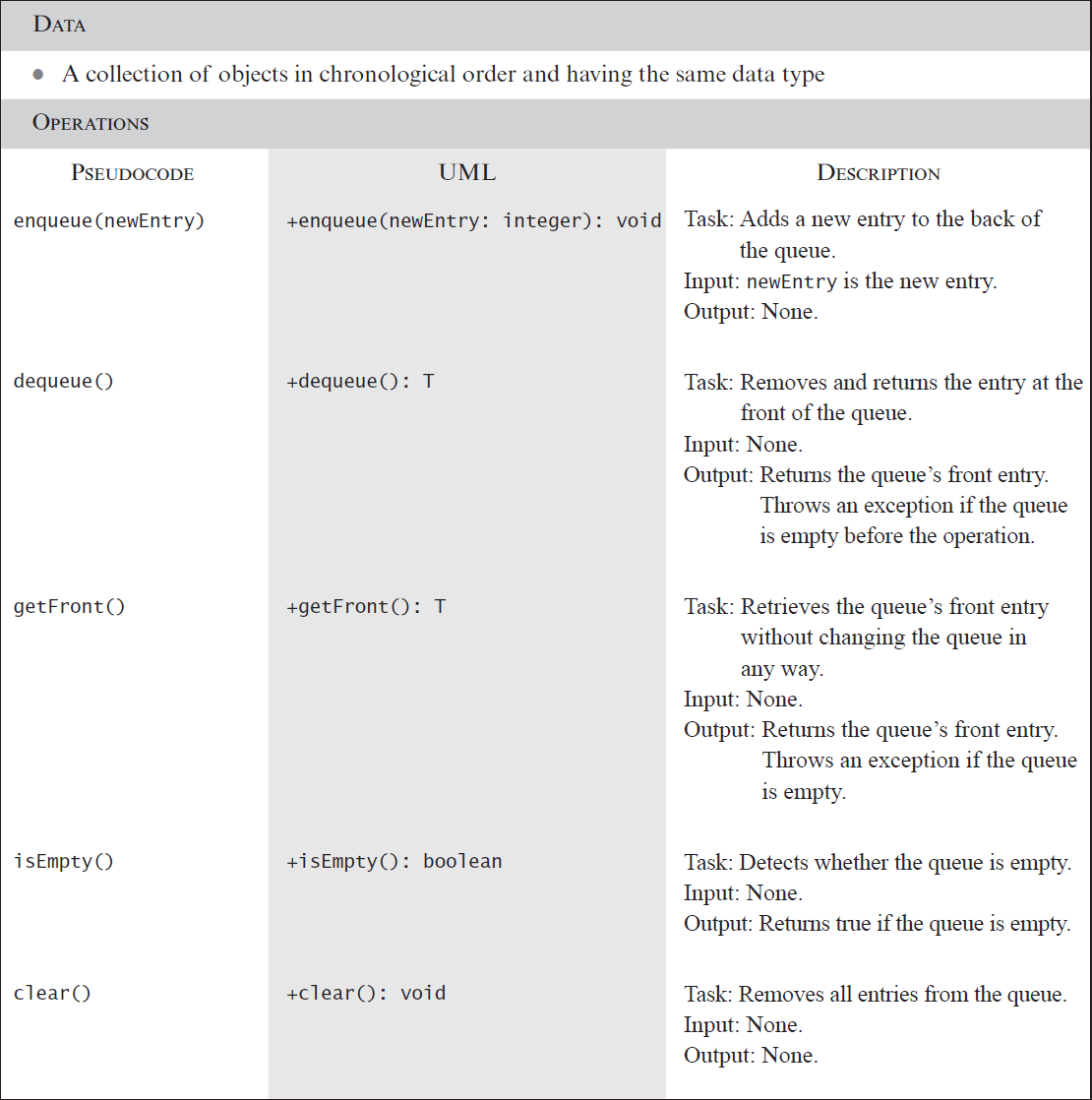
基数排序需要耗费时间对带整数键值的个元素排序，其中是所有键值中基数位数目的最大值。

基数排序的困难之一是，桶的个数依赖于要排序的字符串的类型。整数排序需要10个桶；字排序至少需要26个桶。

# 队列

## ADT队列

队列是先进先出，和栈一样，限制对其中项的访问。





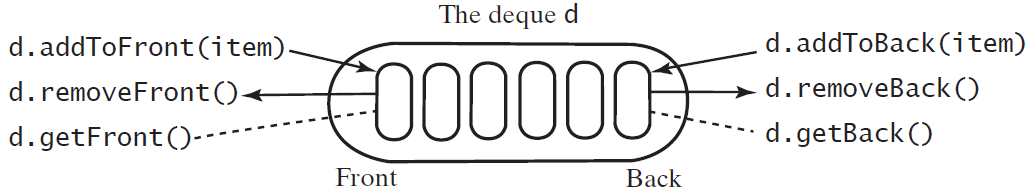
**package** Queue;  
  
**public interface** QueueInterface<T>  
{  
 */\*\*  
 \* Adds a new entry to the back of this queue.  
 \** ***@param newEntry*** *An object to be added.  
 \*/* **public void** enqueue(T newEntry);  
  
  
 */\*\*  
 \* Removes and returns the entry at the front of this queue.  
 \** ***@return*** *The object at the front of the queue.  
 \** ***@throws*** *EmptyQueueException if the queue is empty before the operation.  
 \*/* **public** T dequeue();  
  
  
 */\*\*  
 \* Retrieves the entry at the front of this queue.  
 \** ***@return*** *The object at the front of the queue.  
 \** ***@throws*** *EmptyQueueException if the queue is empty.  
 \*/* **public** T getFront();  
  
  
 */\*\*  
 \* Detects whether this queue is empty.  
 \** ***@return*** *True if the queue is empty, or false otherwise.  
 \*/* **public boolean** isEmpty();  
  
  
 */\*\*  
 \* Removes all entries from this queue.  
 \*/* **public void** clear();  
}

### Java类库：接口Queue

Java.util含有接口Queue。

## ADT双端队列

双端队列的行为更像是双端栈。





**package** Queue;  
  
**public interface** DequeInterface<T>  
{  
 */\*\*  
 \* Adds a new entry to the front/back of this deque.  
 \** ***@param newEntry*** *An object to be added.  
 \*/* **public void** addToFront(T newEntry);  
 **public void** addToBack(T newEntry);  
  
  
 */\*\*  
 \* Removes and returns the front/back entry of this deque.  
 \** ***@return*** *The object at the front/back of the deque.  
 \** ***@throws*** *EmptyQueueException if the deque is empty before the operation  
 \*/* **public** T removeFront();  
 **public** T removeBack();  
  
  
 */\*\*  
 \* Retrieves the front/back entry of this deque.  
 \** ***@return*** *The object at the front/back of the deque.  
 \** ***@throws*** *EmptyQueueException if the deque is empty.  
 \*/* **public** T getFront();  
 **public** T getBack();  
  
  
 */\*\*  
 \* Detects whether this deque is empty.  
 \** ***@return*** *True if the deque is empty, or false otherwise.  
 \*/* **public boolean** isEmpty();  
  
  
 */\*\*  
 \* Removes all entries from this deque.  
 \*/* **public void** clear();  
}

### Java类库：接口Deque和类ArrayDeque

Java.util含有接口Deque。

如果想使用标准类而不是自己创建的一个栈，应该使用标准类ArrayDeque 的实例而不是标准类Stack。ArrayDeque是新的类，它提供的栈的实现比Stack 更快。Stack仍保留在Java类库中，以支持之前已写的Java程序。

## ADT优先队列

优先队列可以使用compareTo来比较对象(对象是Comparable的)的优先级。



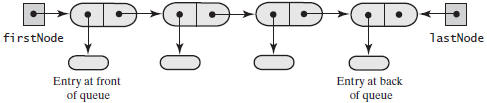
**package** Queue;  
  
**public interface** PriorityQueueInterface<T **extends** Comparable<? **super** T>>  
{  
 */\*\*  
 \* Adds a new entry to this priority queue.  
 \** ***@param newEntry*** *An object to be added.  
 \*/* **public void** add(T newEntry);  
  
  
 */\*\*  
 \* Removes and returns the entry having the highest priority.  
 \** ***@return*** *Either the object having the highest priority or, if the priority queue is empty before the operation, null.  
 \*/* **public** T remove();  
  
  
 */\*\*  
 \* Retrieves the entry having the highest priority.  
 \** ***@return*** *Either the object having the highest priority or, if the priority queue is empty, null.  
 \*/* **public** T peek();  
  
  
 */\*\*  
 \* Detects whether this priority queue is empty.  
 \** ***@return*** *True if the priority queue is empty, or false otherwise.  
 \*/* **public boolean** isEmpty();  
  
  
 */\*\*  
 \* Removes all entries from this priority queue.  
 \*/* **public void** clear();  
}

### Java类库：类PriorityQueue

Java.util含有类PriorityQueue。

## 队列的线性链式实现

如果使用链式结点链来实现队列，则队列的两端必须在链的两端。如果仅有一个指向链的头引用，则访问链的最后结点时需要遍历整个链，这样访问的效率不高。可以增加一个尾引用(tail reference) (指向链中的最后结点的外部引用)。并且将队列的前端放到链的开头，让队列的后端放到了链尾，仅在队列的后端添加项。



### EmptyQueueException

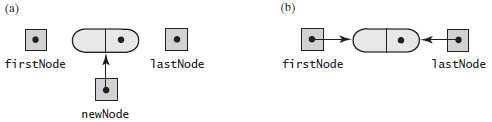


**package Queue**;  
  
**public class** EmptyQueueException **extends** RuntimeException  
{  
 **public** EmptyQueueException()  
 {  
 **this**(**null**);  
 }  
  
 **public** EmptyQueueException(String message)  
 {  
 **super**(message);  
 }  
}

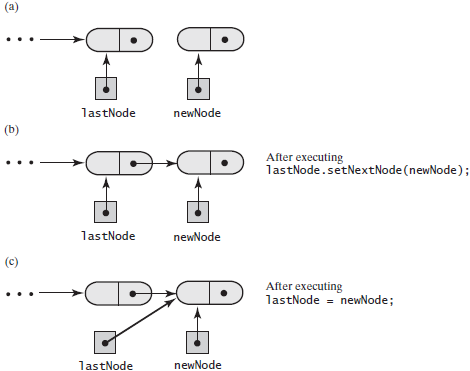
### 方法enqueue

**public void** enqueue(T newEntry)  
{  
 Node newNode = **new** Node(newEntry, **null**);  
 **if** (isEmpty())  
 **firstNode** = newNode;  
 **else  
 lastNode**.setNextNode(newNode);  
 **lastNode** = newNode;  
}

**if** (isEmpty())：



**else：**



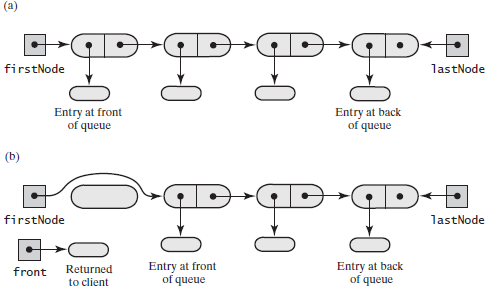
### 方法getFront

@Override  
**public** T getFront()  
{  
 **if** (isEmpty())  
 **throw new** EmptyQueueException();  
 **else  
 return firstNode**.getData();  
}

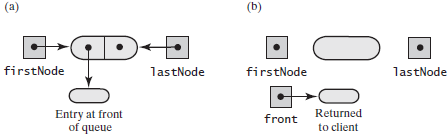
### 方法dequeue

@Override  
**public** T dequeue()  
{  
 T front = getFront();  
 **assert firstNode** != **null**;  
 **firstNode**.setData(**null**);  
 **firstNode** = **firstNode**.getNextNode();  
 **if** (**firstNode** == **null**)  
 **lastNode** = **null**;  
 **return** front;  
}

含有多项的队列：



含有一项的队列：



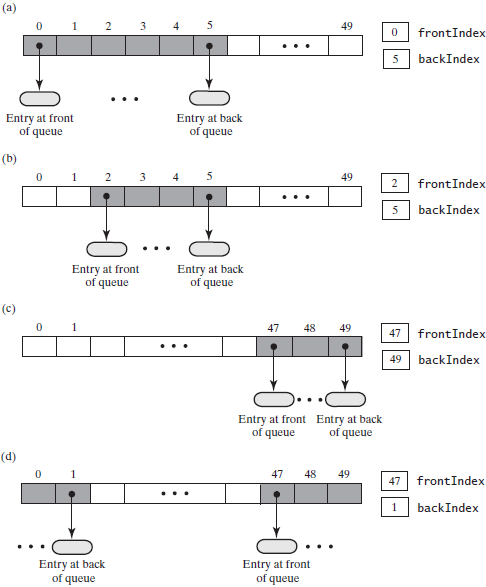
### 完整代码



**package** Queue;  
  
*/\*\*  
 \* A class that implements a queue of objects by using a chain of linked nodes.  
 \** ***@param <T>*** *\*/***public final class** LinkedQueue<T> **implements** QueueInterface<T>  
{  
 **private** Node **firstNode**; *// References node at front of queue* **private** Node **lastNode**; *// References node at back of queue* **public** LinkedQueue()  
 {  
 **firstNode** = **null**;  
 **lastNode** = **null**;  
 }  
  
  
 */\*\*  
 \* Adds a new entry to the back of this queue.  
 \** ***@param newEntry*** *An object to be added.  
 \*/* @Override  
 **public void** enqueue(T newEntry)  
 {  
 Node newNode = **new** Node(newEntry, **null**);  
 **if** (isEmpty())  
 **firstNode** = newNode;  
 **else  
 lastNode**.setNextNode(newNode);  
 **lastNode** = newNode;  
 }  
  
  
 */\*\*  
 \* Retrieves the entry at the front of this queue.  
 \** ***@return*** *The object at the front of the queue.  
 \** ***@throws*** *EmptyQueueException if the queue is empty.  
 \*/* @Override  
 **public** T getFront()  
 {  
 **if** (isEmpty())  
 **throw new** EmptyQueueException();  
 **else  
 return firstNode**.getData();  
 }  
  
  
 */\*\*  
 \* Removes and returns the entry at the front of this queue.  
 \** ***@return*** *The object at the front of the queue.  
 \** ***@throws*** *EmptyQueueException if the queue is empty before the operation.  
 \*/* @Override  
 **public** T dequeue()  
 {  
 T front = getFront();  
 **assert firstNode** != **null**;  
 **firstNode**.setData(**null**);  
 **firstNode** = **firstNode**.getNextNode();  
 **if** (**firstNode** == **null**)  
 **lastNode** = **null**;  
 **return** front;  
 }  
  
 */\*\*  
 \* Detects whether this queue is empty.  
 \** ***@return*** *True if the queue is empty, or false otherwise.  
 \*/* @Override  
 **public boolean** isEmpty()  
 {  
 **return** (**firstNode** == **null**) && (**lastNode** == **null**);  
 }  
  
  
 */\*\*  
 \* Removes all entries from this queue.  
 \*/* @Override  
 **public void** clear()  
 {  
 **firstNode** = **null**;  
 **lastNode** = **null**;  
 }  
  
 **private class** Node  
 {  
 **private** T **data**;  
 **private** Node **next**;  
  
 **private** Node(T dataPortion)  
 {  
 **this**(dataPortion, **null**);  
 }  
  
 **private** Node(T dataPortion, Node linkPortion)  
 {  
 **data** = dataPortion;  
 **next** = linkPortion;  
 }  
  
 **private** T getData(){**return data**;}  
 **private void** setData(T newData){**data** = newData;}  
  
 **private** Node getNextNode(){**return next**;}  
 **private void** setNextNode(Node nextNode){**next** = nextNode;}  
 }  
}

## 基于数组实现队列

一旦队列到达数组的末端，可以将随后的项添加到数组的开头，让数组好似是一个循环，这样它的第一个位置接在它的最后位置之后。为此，对下标使用取模运算。具体来说，将项添加到队列中时，将backindex加1再对数组大小取模。



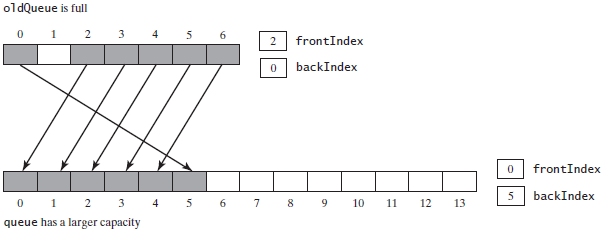
### 方法enqueue

方法enqueue调用私有方法ensureCapacity。如果数组满了，它倍增数组，然后将新项紧接着放在数组巳占用的最后位置的后面。要确定这个位置的下标，需要将backindex加1。但因为数组是循环的，所以使用运算符%，当backIndex达到最大值时让它变为0。

@Override  
**public void** enqueue(T newEntry)  
{  
 checkInitialization();  
 ensureCapacity();  
 **backIndex** = (**backIndex** + 1) % **queue**.**length**;  
 **queue**[**backIndex**] = newEntry;  
}

### 方法ensureCapacity

当增大数组的大小时，必须将它的项复制到新分配的空间中。不过我们需要谨慎，因为这里的数组是循环数组，我们必须将项按照它们在队列中出现的次序进行复制。

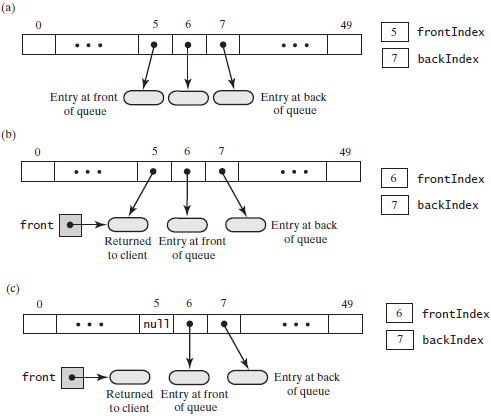


**private void** ensureCapacity()  
{  
 **if** (**frontIndex** == ((**backIndex** + 2) % **queue**.**length**)) *// If array is full, double size of array* {  
 T[] oldQueue = **queue**;  
 **int** oldSize = oldQueue.**length**;  
 **int** newSize = 2 \* oldSize;  
 checkCapacity(newSize - 1);  
  
 @SuppressWarnings(**"unchecked"**)  
 T[] tempQueue = (T[]) **new** Object[newSize];  
 **queue** = tempQueue;  
 **for** (**int** index = 0; index < oldSize - 1; index++)  
 {  
 **queue**[index] = oldQueue[**frontIndex**];  
 **frontIndex** = (**frontIndex** + 1) % oldSize;  
 }  
 **frontIndex** = 0;  
 **backIndex** = oldSize - 2;  
 }  
}

### 方法getFront

@Override  
**public** T getFront()  
{  
 **if** (isEmpty())  
 **throw new** EmptyQueueException();  
 **else  
 return queue**[**frontIndex**];  
}

### 方法dequeue



@Override  
**public** T dequeue()  
{  
 **if** (isEmpty())  
 **throw new** EmptyQueueException();  
 **else** {  
 T front = **queue**[**frontIndex**];  
 **queue**[**frontIndex**] = **null**;  
 **frontIndex** = (**frontIndex** + 1) % **queue**.**length**;  
 **return** front;  
 }  
}

### 方法isEmpty和方法clear

**public boolean** isEmpty()  
{  
 **return frontIndex** == ((**backIndex** + 1) % **queue**.**length**);  
}  
  
  
**public void** clear()  
{  
 **while** (!isEmpty())  
 dequeue();  
}

### 完整代码



**package** Queue;  
  
*/\*\*  
 \* A class that implements a queue of objects by using an array.  
 \** ***@param <T>*** *\*/***public final class** ArrayQueue<T> **implements** QueueInterface<T>  
{  
 **private** T[] **queue**; *// Circular array of queue entries and one unused location* **private int frontIndex**;  
 **private int backIndex**;  
 **private boolean initialized** = **false**;  
 **private static final int *DEFAULT\_CAPACITY*** = 50;  
 **private static final int *MAX\_CAPACITY*** = 10000;  
  
 **public** ArrayQueue()  
 {  
 **this**(***DEFAULT\_CAPACITY***);  
 }  
  
 **public** ArrayQueue(**int** initialCapacity)  
 {  
 checkCapacity(initialCapacity);  
  
 @SuppressWarnings(**"unchecked"**)  
 T[] tempQueue = (T[]) **new** Object[initialCapacity + 1];  
  
 **queue** = tempQueue;  
 **frontIndex** = 0;  
 **backIndex** = initialCapacity; *// 数组的大小为0~initialCapacity = initialCapacity + 1* **initialized** = **true**;  
 }  
  
  
 */\*\*  
 \* Removes and returns the entry at the front of this queue.  
 \** ***@return*** *The object at the front of the queue.  
 \** ***@throws*** *EmptyQueueException if the queue is empty before the operation.  
 \*/* @Override  
 **public** T dequeue()  
 {  
 **if** (isEmpty())  
 **throw new** EmptyQueueException();  
 **else** {  
 T front = **queue**[**frontIndex**];  
 **queue**[**frontIndex**] = **null**;  
 **frontIndex** = (**frontIndex** + 1) % **queue**.**length**;  
 **return** front;  
 }  
 }  
  
  
 */\*\*  
 \* Retrieves the entry at the front of this queue.  
 \** ***@return*** *The object at the front of the queue.  
 \** ***@throws*** *EmptyQueueException if the queue is empty.  
 \*/* @Override  
 **public** T getFront()  
 {  
 **if** (isEmpty())  
 **throw new** EmptyQueueException();  
 **else  
 return queue**[**frontIndex**];  
 }  
  
  
 */\*\*  
 \* Adds a new entry to the back of this queue.  
 \** ***@param newEntry*** *An object to be added.  
 \*/* @Override  
 **public void** enqueue(T newEntry)  
 {  
 checkInitialization();  
 ensureCapacity();  
 **backIndex** = (**backIndex** + 1) % **queue**.**length**;  
 **queue**[**backIndex**] = newEntry;  
 }  
  
  
 */\*\*  
 \* Doubles the size of the array queue if it is full.  
 \* Precondition: checkInitialization has been called.  
 \*/* **private void** ensureCapacity()  
 {  
 **if** (**frontIndex** == ((**backIndex** + 2) % **queue**.**length**)) *// If array is full, double size of array* {  
 T[] oldQueue = **queue**;  
 **int** oldSize = oldQueue.**length**;  
 **int** newSize = 2 \* oldSize;  
 checkCapacity(newSize - 1);  
  
 @SuppressWarnings(**"unchecked"**)  
 T[] tempQueue = (T[]) **new** Object[newSize];  
 **queue** = tempQueue;  
 **for** (**int** index = 0; index < oldSize - 1; index++)  
 {  
 **queue**[index] = oldQueue[**frontIndex**];  
 **frontIndex** = (**frontIndex** + 1) % oldSize;  
 }  
 **frontIndex** = 0;  
 **backIndex** = oldSize - 2;  
 }  
 }  
  
 **private void** checkCapacity(**int** capacity)  
 {  
 **if** (capacity > ***MAX\_CAPACITY***)  
 **throw new** IllegalStateException(**"Exceed the maximum "** + ***MAX\_CAPACITY***);  
 }  
  
  
 **private void** checkInitialization()  
 {  
 **if** (!**initialized**)  
 **throw new** SecurityException(**"Queue is not initialized properly!"**);  
 }  
  
  
 **public boolean** isEmpty()  
 {  
 **return frontIndex** == ((**backIndex** + 1) % **queue**.**length**);  
 }  
  
  
 **public void** clear()  
 {  
 **while** (!isEmpty())  
 dequeue();  
 }  
}

## 队列的循环链式实现

在循环数组表示中，队列使用了固定大小数组中可用位置中的一部分。当向队列中添加一项时，使用数组中下一个未占用的位置。当从队列中删除一项时，数组的这个位置可用于队列以后的使用。因为添加和删除都在队列的端点处进行，所以队列占用了循环数组中的连续位置。可用位置也是连续的，因为数组是连续的。所以循环数组有两部分：一部分含有队列，另一部分可用于队列。

A two-part circular linked chain that represents a queue:

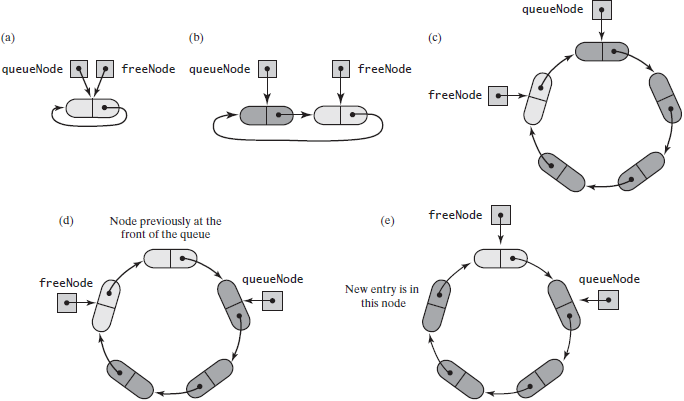
(a) when it is empty;

(b) after adding one entry;

(c) after adding three more entries;

(d) after removing the front entry;

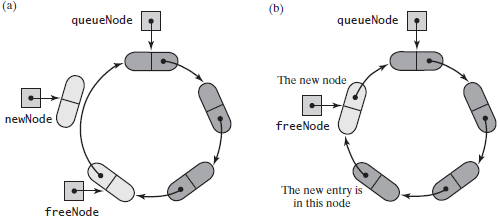
(e) after adding one more entry



queueNode指向巳分配给队头的结点；freeNode指向跟在队列后端后的第一个可用结点。如果queueNode等于freeNode，则队列为空。当从队列删除项时，将这个结点保留在结点环中而不是释放它。

### 方法enqueue

向队列添加项之前，必须看看链中是否有可用结点。如果没有，必须分配一个新结点并将它链接到链中。将新结点插入链中freeNode所指结点的后面。



**public void** enqueue(T newEntry)  
{  
 **freeNode**.setData(newEntry);  
 **if** (isChainFull())  
 {  
 *// Allocate a new node and insert it after the node that freeNode references* Node newNode = **new** Node(**null**, **freeNode**.getNextNode());  
 **freeNode**.setNextNode(newNode);  
 }  
 **freeNode** = **freeNode**.getNextNode();  
}

### 方法isChainFull

**private boolean** isChainFull()  
{  
 **return queueNode** == **freeNode**.getNextNode();  
}

### 方法getFront

@Override  
**public** T getFront()  
{  
 **if** (isEmpty())  
 **throw new** EmptyQueueException();  
 **else  
 return queueNode**.getData();  
}

### 方法dequeue

**public** T dequeue()  
{  
 T front = getFront();  
 **assert** !isEmpty();  
 **queueNode**.setData(**null**);  
 **queueNode** = **queueNode**.getNextNode();  
 **return** front;  
}

### 方法isEmpty和clear

**public boolean** isEmpty()  
{  
 **return queueNode** == **freeNode**;  
}

**public void** clear()  
{  
 **if** (!isEmpty())  
 dequeue();  
}

## 双端队列的双向链式实现

内部结点既指向下一个结点也指向前一个结点，第一个结点和最后一个结点都含有一个null引用。



### 方法addToBack

**public void** addToBack(T newEntry)  
{  
 DLNode newNode = **new** DLNode(**lastNode**, newEntry, **null**);  
 **if** (isEmpty())  
 **firstNode** = newNode;  
 **else  
 lastNode**.setNextNode(newNode);  
 **lastNode** = newNode;  
}  
  
  
**public boolean** isEmpty()  
{  
 **return firstNode** == **null** && **lastNode** == **null**;  
}



### 方法addToFirst

**public void** addToFront(T newEntry)  
{  
 DLNode newNode = **new** DLNode(**null**, newEntry, **firstNode**);  
 **if** (isEmpty())  
 **lastNode** = newNode;  
 **else  
 firstNode**.setPreviousNode(newNode);  
 **firstNode** = newNode;  
}

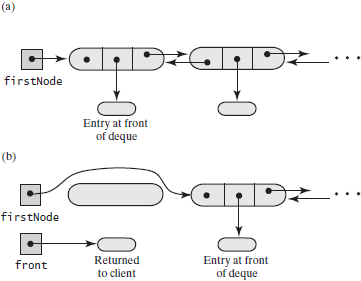
### 方法getBack和getFront

**public** T getBack()  
{  
 **if** (isEmpty())  
 **throw new** EmptyQueueException();  
 **else  
 return lastNode**.getData();  
}

**public** T getFront()  
{  
 **if** (isEmpty())  
 **throw new** EmptyQueueException();  
 **else  
 return firstNode**.getData();  
}

### 方法removeFront

**public** T removeFront()  
{  
 T front = getFront();  
 **assert firstNode** != **null**;  
 **firstNode** = **firstNode**.getNextNode();  
 **if** (**firstNode** == **null**)  
 **lastNode** = **null**;  
 **else  
 firstNode**.setPreviousNode(**null**);  
 **return** front;  
}



### 方法removeBack

**public** T removeBack()  
{  
 T back = getBack();  
 **assert lastNode** != **null**;  
 **lastNode** = **lastNode**.getPreviousNode();  
 **if** (**lastNode** == **null**)  
 **firstNode** = **null**;  
 **else  
 lastNode**.setNextNode(**null**);  
 **return** back;  
}

### 完整代码



**package** Queue;  
  
**public final class** LinkedDeque<T> **implements** DequeInterface<T>  
{  
 **private** DLNode **firstNode**; *// References node at front of deque* **private** DLNode **lastNode**; *// References node at back of deque* **public** LinkedDeque()  
 {  
 **firstNode** = **null**;  
 **lastNode** = **null**;  
 }  
  
  
 */\*\*  
 \* Removes and returns the front entry of this deque.  
 \** ***@return*** *The object at the front of the deque.  
 \** ***@throws*** *EmptyQueueException if the deque is empty before the operation  
 \*/* @Override  
 **public** T removeFront()  
 {  
 T front = getFront();  
 **assert firstNode** != **null**;  
 **firstNode** = **firstNode**.getNextNode();  
 **if** (**firstNode** == **null**)  
 **lastNode** = **null**;  
 **else  
 firstNode**.setPreviousNode(**null**);  
 **return** front;  
 }  
  
  
 */\*\*  
 \* Removes and returns the back entry of this deque.  
 \** ***@return*** *The object at the back of the deque.  
 \** ***@throws*** *EmptyQueueException if the deque is empty before the operation  
 \*/* @Override  
 **public** T removeBack()  
 {  
 T back = getBack();  
 **assert lastNode** != **null**;  
 **lastNode** = **lastNode**.getPreviousNode();  
 **if** (**lastNode** == **null**)  
 **firstNode** = **null**;  
 **else  
 lastNode**.setNextNode(**null**);  
 **return** back;  
 }  
  
  
 */\*\*  
 \* Retrieves the back entry of this deque.  
 \** ***@return*** *The object at the back of the deque.  
 \** ***@throws*** *EmptyQueueException if the deque is empty.  
 \*/* @Override  
 **public** T getBack()  
 {  
 **if** (isEmpty())  
 **throw new** EmptyQueueException();  
 **else  
 return lastNode**.getData();  
 }  
  
  
 */\*\*  
 \* Retrieves the front entry of this deque.  
 \** ***@return*** *The object at the front of the deque.  
 \** ***@throws*** *EmptyQueueException if the deque is empty.  
 \*/* @Override  
 **public** T getFront()  
 {  
 **if** (isEmpty())  
 **throw new** EmptyQueueException();  
 **else  
 return firstNode**.getData();  
 }  
  
  
 */\*\*  
 \* Adds a new entry to the back of this deque.  
 \** ***@param newEntry*** *\*/* @Override  
 **public void** addToBack(T newEntry)  
 {  
 DLNode newNode = **new** DLNode(**lastNode**, newEntry, **null**);  
 **if** (isEmpty())  
 **firstNode** = newNode;  
 **else  
 lastNode**.setNextNode(newNode);  
 **lastNode** = newNode;  
 }  
  
  
 */\*\*  
 \* Adds a new entry to the front of this deque.  
 \** ***@param newEntry*** *An object to be added.  
 \*/* @Override  
 **public void** addToFront(T newEntry)  
 {  
 DLNode newNode = **new** DLNode(**null**, newEntry, **firstNode**);  
 **if** (isEmpty())  
 **lastNode** = newNode;  
 **else  
 firstNode**.setPreviousNode(newNode);  
 **firstNode** = newNode;  
 }  
  
  
 **public boolean** isEmpty()  
 {  
 **return firstNode** == **null** && **lastNode** == **null**;  
 }  
  
  
 **public void** clear()  
 {  
 **firstNode** = **null**;  
 **lastNode** = **null**;  
 }  
  
  
 **private class** DLNode  
 {  
 **private** T **data**;  
 **private** DLNode **next**;  
 **private** DLNode **previous**;  
  
 **public** DLNode(DLNode previousNode, T nodeData, DLNode nextNode)  
 {  
 **data** = nodeData;  
 **previous** = previousNode;  
 **next** = nextNode;  
 }  
  
 **private** T getData(){**return data**;}  
 **private void** setData(T newData){**data** = newData;}  
  
 **private** DLNode getNextNode(){**return next**;}  
 **private void** setNextNode(DLNode nextNode){**next** = nextNode;}  
  
 **private** DLNode getPreviousNode(){**return previous**;}  
 **private void** setPreviousNode(DLNode previousNode){**previous** = previousNode;}  
 }  
}

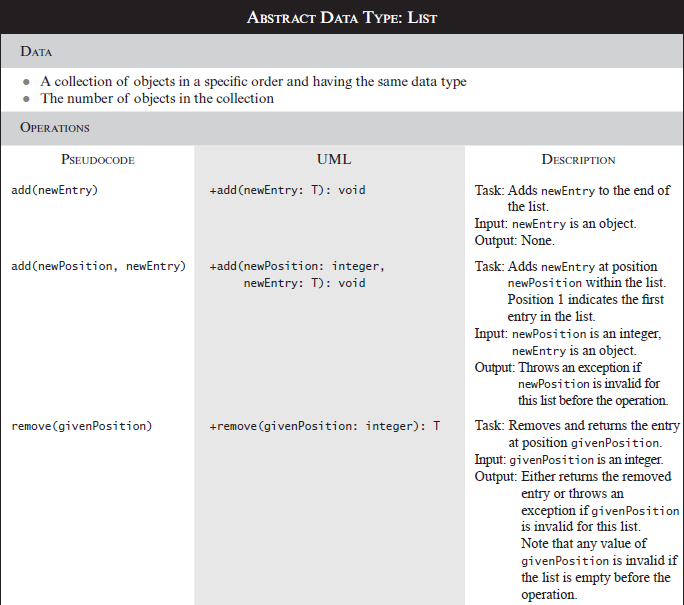
## 优先队列

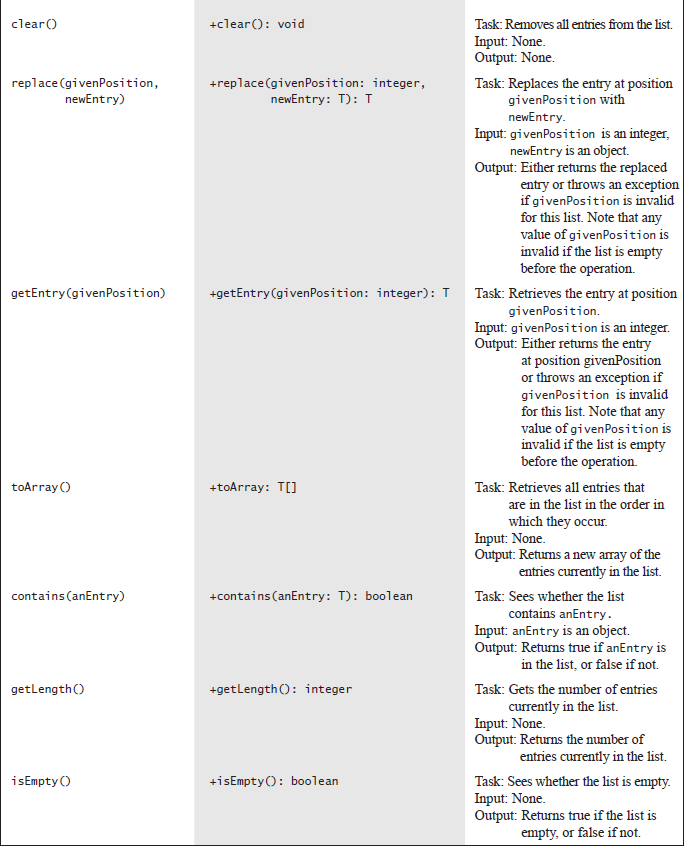
有序表可以按优先级次序维护优先队列中的项；可以使用数组或链来实现优先队列，但更高效的实现是使用堆！

# 线性表

线性表(List)是一个集合。

## 线性表接口

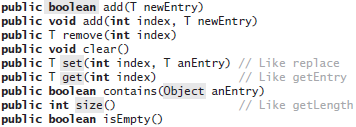






**package** Lists;  
  
*/\*\*  
 \* An interface for the ADT list.  
 \* Entries in a list have positions that begin with 1.  
 \** ***@param <T>*** *\*/***public interface** ListInterface<T>  
{  
 */\*\*  
 \* Adds a new entry to the end of this list.  
 \** ***@param newEntry*** *The object to be added as a new entry.  
 \*/* **public void** add(T newEntry);  
  
  
 */\*\*  
 \* Adds a new entry at a specified position within this list.  
 \** ***@param newPosition*** *\** ***@param newEntry*** *\** ***@throws*** *IndexOutOfBoundsException if either newPosition < 1 or newPosition > getLength() + 1.  
 \*/* **public void** add(**int** newPosition, T newEntry);  
  
  
 */\*\*  
 \* Removes the entry at a given position from this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the removed entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* **public** T remove(**int** givenPosition);  
  
  
 */\*\*  
 \* Removes all entries from this list.  
 \*/* **public void** clear();  
  
  
 */\*\*  
 \* Replaces the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@param newEntry*** *\** ***@return*** *The original entry that was replaced.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* **public** T replace(**int** givenPosition, T newEntry);  
  
  
 */\*\*  
 \* Retrieves the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the indicated entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* **public** T getEntry(**int** givenPosition);  
  
  
 */\*\*  
 \* Retrieves all entries that are in this list in the order in which they occur in the list.  
 \** ***@return*** *A newly allocated array of all the entries in the list. If the list is empty, the returned array is empty.  
 \*/* **public** T[] toArray();  
  
  
 */\*\*  
 \* Whether this list contains a given entry.  
 \** ***@param anEntry*** *\** ***@return*** *True if the list contains anEntry, or false if not.  
 \*/* **public boolean** contains(T anEntry);  
  
  
 */\*\*  
 \* Gets the length of this list.  
 \** ***@return*** *The integer number of entries currently in the list.  
 \*/* **public int** getLength();  
  
  
 */\*\*  
 \* Whether this list is empty.  
 \** ***@return*** *True if the list is empty, or false if not.  
 \*/* **public boolean** isEmpty();  
}

Java类库：接口List和类ArrayList



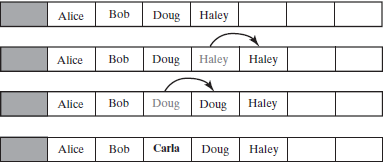
Java类库含有一个ADT线性表，它使用可变大小的数组来实现。称为ArrayList的这个类实现了接口java.util.List。

List<String> myList = new ArrayList<>();

## 使用数组实现线性表

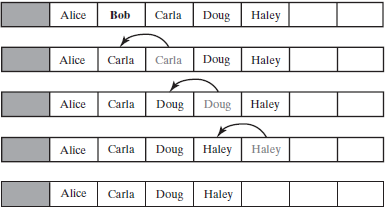
### 在线性表的给定位置添加

*/\*\*  
 \* Adds a new entry at a specified position within this list.  
 \** ***@param newPosition*** *\** ***@param newEntry*** *\** ***@throws*** *IndexOutOfBoundsException if either newPosition < 1 or newPosition > getLength() + 1.  
 \*/*@Override  
**public void** add(**int** newPosition, T newEntry)  
{  
 checkInitialization();  
 **if** ((newPosition >= 1) && (newPosition <= **numberOfEntries** + 1))  
 {  
 **if** (newPosition <= **numberOfEntries**)  
 makeRoom(newPosition);  
 **list**[newPosition] = newEntry;  
 **numberOfEntries**++;  
 ensureCapacity(); *// Ensure enough room for next add* }  
 **else  
 throw new** IndexOutOfBoundsException(**"Given position of add's new entry is out of bounds."**);  
}  
  
  
*/\*\*  
 \* Makes room for a new entry at newPosition.  
 \** ***@param newPosition*** *\* Precondition: 1 <= newPosition <= numberOfEntries + 1;  
 \* numberOfEntries is list’s length before addition;  
 \* checkInitialization has been called.  
 \*/***private void** makeRoom(**int** newPosition)  
{  
 **assert** (newPosition >= 1) && (newPosition <= **numberOfEntries** + 1);  
 **int** newIndex = newPosition;  
 **int** lastIndex = **numberOfEntries**;  
  
 *// Move each entry to next higher index, starting at end of list and continuing until the entry at newIndex is moved* **for** (**int** index = lastIndex; index >= newIndex; index--)  
 **list**[index + 1] = **list**[index];  
}



### 方法remove

*/\*\*  
 \* Removes the entry at a given position from this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the removed entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/*@Override  
**public** T remove(**int** givenPosition)  
{  
 checkInitialization();  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 T result = **list**[givenPosition]; *// Get entry to be removed  
  
 // Move subsequent entries toward entry to be removed, unless it is last in list* **if** (givenPosition < **numberOfEntries**)  
 removeGap(givenPosition);  
 **numberOfEntries**--;  
 **return** result;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to remove operation."**);  
}  
  
  
*/\*\*  
 \* Shifts entries that are beyond the entry to be removed to the next lower position.  
 \* Precondition: 1 <= givenPosition < numberOfEntries;  
 \* numberOfEntries is list’s length before removal;  
 \* checkInitialization has been called.  
 \** ***@param givenPosition*** *\*/***private void** removeGap(**int** givenPosition)  
{  
 **assert** (givenPosition >= 1) && (givenPosition < **numberOfEntries**);  
 **int** removedIndex = givenPosition;  
 **int** lastIndex = **numberOfEntries**;  
 **for** (**int** index = removedIndex; index < lastIndex; index++)  
 **list**[index] = **list**[index + 1];  
}



### 方法replace和getEntry

*/\*\*  
 \* Replaces the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@param newEntry*** *\** ***@return*** *The original entry that was replaced.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/*@Override  
**public** T replace(**int** givenPosition, T newEntry)  
{  
 checkInitialization();  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 T originalEntry = **list**[givenPosition];  
 **list**[givenPosition] = newEntry;  
 **return** originalEntry;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to replace operation."**);  
}  
  
  
*/\*\*  
 \* Retrieves the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the indicated entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/*@Override  
**public** T getEntry(**int** givenPosition)  
{  
 checkInitialization();  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 **return list**[givenPosition];  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to getEntry operation."**);  
}

### 方法contains

**public boolean** contains(T anEntry)  
{  
 checkInitialization();  
 **boolean** found = **false**;  
 **int** index = 1;  
 **while** (!found && (index <= **numberOfEntries**))  
 {  
 **if** (anEntry.equals(**list**[index]))  
 found = **true**;  
 index++;  
 }  
 **return** found;  
}

### 完整代码



**package** Lists;  
  
**import** java.lang.reflect.Array;  
**import** java.util.Arrays;  
  
*/\*\*  
 \* A class that implements a list of objects by using an array.  
 \* Entries in a list have positions that begin with 1.  
 \* Duplicate entries are allowed.  
 \** ***@param <T>*** *\*/***public class** AList<T> **implements** ListInterface<T>  
{  
 **private** T[] **list**; *// Array of list entries; ignore list[0]* **private int numberOfEntries**;  
 **private boolean initialized** = **false**;  
 **private static final int *DEFAULT\_CAPACITY*** = 25;  
 **private static final int *MAX\_CAPACITY*** = 10000;  
  
 **public** AList()  
 {  
 **this**(***DEFAULT\_CAPACITY***);  
 }  
  
 **public** AList(**int** initialCapacity)  
 {  
 **if** (initialCapacity < ***DEFAULT\_CAPACITY***)  
 initialCapacity = ***DEFAULT\_CAPACITY***;  
 **else** checkCapacity(initialCapacity);  
  
 @SuppressWarnings(**"unchecked"**)  
 T[] tempList = (T[]) **new** Object[initialCapacity + 1]; *// Position begins with 1* **list** = tempList;  
 **numberOfEntries** = 0;  
 **initialized** = **true**;  
 }  
  
  
 */\*\*  
 \* Whether this list contains a given entry.  
 \** ***@param anEntry*** *\** ***@return*** *True if the list contains anEntry, or false if not.  
 \*/* @Override  
 **public boolean** contains(T anEntry)  
 {  
 checkInitialization();  
 **boolean** found = **false**;  
 **int** index = 1;  
 **while** (!found && (index <= **numberOfEntries**))  
 {  
 **if** (anEntry.equals(**list**[index]))  
 found = **true**;  
 index++;  
 }  
 **return** found;  
 }  
  
  
 */\*\*  
 \* Replaces the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@param newEntry*** *\** ***@return*** *The original entry that was replaced.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T replace(**int** givenPosition, T newEntry)  
 {  
 checkInitialization();  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 T originalEntry = **list**[givenPosition];  
 **list**[givenPosition] = newEntry;  
 **return** originalEntry;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to replace operation."**);  
 }  
  
  
 */\*\*  
 \* Retrieves the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the indicated entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T getEntry(**int** givenPosition)  
 {  
 checkInitialization();  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 **return list**[givenPosition];  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to getEntry operation."**);  
 }  
  
  
 */\*\*  
 \* Removes the entry at a given position from this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the removed entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T remove(**int** givenPosition)  
 {  
 checkInitialization();  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 T result = **list**[givenPosition]; *// Get entry to be removed  
  
 // Move subsequent entries toward entry to be removed, unless it is last in list* **if** (givenPosition < **numberOfEntries**)  
 removeGap(givenPosition);  
 **numberOfEntries**--;  
 **return** result;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to remove operation."**);  
 }  
  
  
 */\*\*  
 \* Shifts entries that are beyond the entry to be removed to the next lower position.  
 \* Precondition: 1 <= givenPosition < numberOfEntries;  
 \* numberOfEntries is list’s length before removal;  
 \* checkInitialization has been called.  
 \** ***@param givenPosition*** *\*/* **private void** removeGap(**int** givenPosition)  
 {  
 **assert** (givenPosition >= 1) && (givenPosition < **numberOfEntries**);  
 **int** removedIndex = givenPosition;  
 **int** lastIndex = **numberOfEntries**;  
 **for** (**int** index = removedIndex; index < lastIndex; index++)  
 **list**[index] = **list**[index + 1];  
 }  
  
  
 */\*\*  
 \* Retrieves all entries that are in this list in the order in which they occur in the list.  
 \** ***@return*** *A newly allocated array of all the entries in the list. If the list is empty, the returned array is empty.  
 \*/* @Override  
 **public** T[] toArray()  
 {  
 checkInitialization();  
 @SuppressWarnings(**"unchecked"**)  
 T[] result = (T[]) **new** Object[**numberOfEntries**];  
 **for** (**int** index = 0; index < **numberOfEntries**; index++)  
 result[index] = **list**[index + 1];  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Display all entries in a list  
 \** ***@param list*** *\*/* **public static void** displayList(ListInterface<String> list)  
 {  
 System.***out***.println(**"The list contains "** + list.getLength() + **" entries, as follows: "**);  
 Object[] listArray = list.toArray();  
 **for** (**int** index = 0; index < listArray.**length**; index++)  
 System.***out***.print(listArray[index] + **" "**);  
 }  
  
  
 */\*\*  
 \* Adds a new entry to the end of this list.  
 \** ***@param newEntry*** *The object to be added as a new entry.  
 \*/* @Override  
 **public void** add(T newEntry)  
 {  
 checkInitialization();  
 *//list[numberOfEntries++] = newEntry;* **list**[**numberOfEntries** + 1] = newEntry;  
 **numberOfEntries**++;  
 ensureCapacity();  
 }  
  
  
 */\*\*  
 \* Adds a new entry at a specified position within this list.  
 \** ***@param newPosition*** *\** ***@param newEntry*** *\** ***@throws*** *IndexOutOfBoundsException if either newPosition < 1 or newPosition > getLength() + 1.  
 \*/* @Override  
 **public void** add(**int** newPosition, T newEntry)  
 {  
 checkInitialization();  
 **if** ((newPosition >= 1) && (newPosition <= **numberOfEntries** + 1))  
 {  
 **if** (newPosition <= **numberOfEntries**)  
 makeRoom(newPosition);  
 **list**[newPosition] = newEntry;  
 **numberOfEntries**++;  
 ensureCapacity(); *// Ensure enough room for next add* }  
 **else  
 throw new** IndexOutOfBoundsException(**"Given position of add's new entry is out of bounds."**);  
 }  
  
  
 */\*\*  
 \* Makes room for a new entry at newPosition.  
 \** ***@param newPosition*** *\* Precondition: 1 <= newPosition <= numberOfEntries + 1;  
 \* numberOfEntries is list’s length before addition;  
 \* checkInitialization has been called.  
 \*/* **private void** makeRoom(**int** newPosition)  
 {  
 **assert** (newPosition >= 1) && (newPosition <= **numberOfEntries** + 1);  
 **int** newIndex = newPosition;  
 **int** lastIndex = **numberOfEntries**;  
  
 *// Move each entry to next higher index, starting at end of list and continuing until the entry at newIndex is moved* **for** (**int** index = lastIndex; index >= newIndex; index--)  
 **list**[index + 1] = **list**[index];  
 }  
  
  
 */\*\*  
 \* Doubles the capacity of the array list if it is full.  
 \* Precondition: checkInitialization has been called.  
 \* 假定数组中至少还有添加一次的空间，一旦添加使得数组满了，就扩大数组  
 \*/* **private void** ensureCapacity()  
 {  
 **int** capacity = **list**.**length** - 1; *// 假定数组中至少还有添加一次的空间* **if** (**numberOfEntries** >= capacity)  
 {  
 **int** newCapacity = 2 \* capacity;  
 checkCapacity(newCapacity);  
 **list** = Arrays.*copyOf*(**list**, newCapacity + 1);  
 }  
 }  
  
  
 **private void** checkCapacity(**int** capacity)  
 {  
 **if** (capacity > ***MAX\_CAPACITY***)  
 **throw new** IllegalStateException(**"Exceed allowed maximum "** + ***MAX\_CAPACITY***);  
 }  
  
  
 **private void** checkInitialization()  
 {  
 **if** (!**initialized**)  
 **throw new** SecurityException(**"List is not initialized properly!"**);  
 }  
  
  
 **public boolean** isEmpty()  
 {  
 **return numberOfEntries** == 0;  
 }  
  
 **public int** getLength()  
 {  
 **return numberOfEntries**;  
 }  
  
 **public void** clear()  
 {  
 **while** (!isEmpty())  
 {  
 **int** index = **numberOfEntries** - 1;  
 remove(index);  
 }  
 }  
}

## 使用链式数据实现线性表

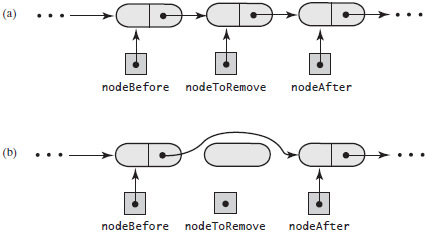
使用数组实现ADT线性表，优点缺点共存，或者数组有固定的大小，或者当它满时需移到一个更大的数组中。因为固定大小的数组可能导致线性表满，所以类AList使用可变大小的数组根据线性表的需求提供空间。但是当扩展数组时这个策略需要移动数据。另外，为新项腾出空间或去掉删除后留下的空隙，数组都需要移动数据。

### 从不同位置删除结点

删除第一个结点：



删除除第一个结点以外的其他结点：



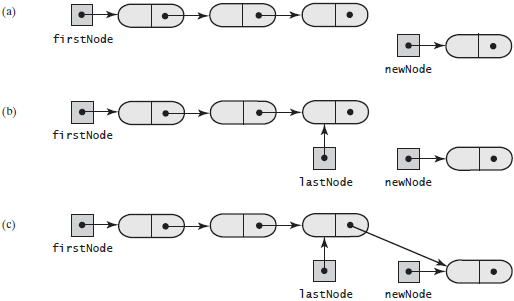
**public** T remove(**int** givenPosition)  
{  
 T result = **null**;  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 **if** (givenPosition == 1)  
 {  
 result = **firstNode**.getData();  
 **firstNode** = **firstNode**.getNextNode();  
 }  
 **else** {  
 Node nodeBefore = getNodeAt(givenPosition - 1);  
 Node nodeToRemove = nodeBefore.getNextNode();  
 result = nodeToRemove.getData();  
 Node nodeAfter = nodeToRemove.getNextNode();  
 nodeBefore.setNextNode(nodeAfter);  
 }  
 **numberOfEntries**--;  
 **return** result;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to remove operation."**);  
}

### 方法getNodeAt

**private** Node getNodeAt(**int** givenPosition)  
{  
 **assert** (**firstNode** != **null**) && (givenPosition >= 1) && (givenPosition <= **numberOfEntries**);  
 Node currentNode = **firstNode**;  
  
 *// Traverse the chain to locate the desired node, skipped if givenPosition is 1* **for** (**int** counter = 1; counter < givenPosition; counter++)  
 currentNode = currentNode.getNextNode();  
 **assert** currentNode != **null**;  
   
 **return** currentNode;  
}

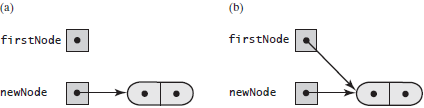
### 添加到线性表的表尾

**public void** add(T newEntry)  
{  
 Node newNode = **new** Node(newEntry);  
 **if** (isEmpty())  
 **firstNode** = newNode;  
 **else** {  
 Node lastNode = getNodeAt(**numberOfEntries**);  
 lastNode.setNextNode(newNode);  
 }  
 **numberOfEntries**++;  
}

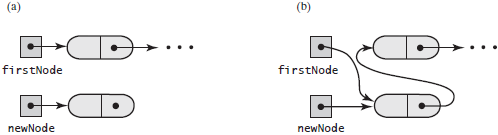


### 在线性表的给定位置添加

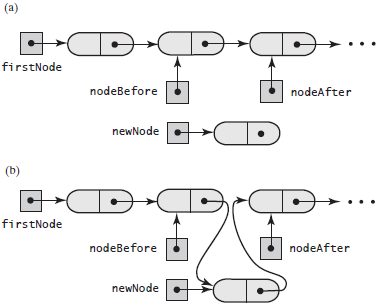
将结点添加在空链中：



新结点添加在链头：



将结点添加在链中：



**public void** add(**int** newPosition, T newEntry)  
{  
 **if** ((newPosition >= 1) && (newPosition <= **numberOfEntries** + 1))  
 {  
 Node newNode = **new** Node(newEntry);  
 **if** (newPosition == 1)  
 {  
 newNode.setNextNode(**firstNode**);  
 **firstNode** = newNode;  
 }  
 **else** {  
 Node nodeBefore = getNodeAt(newPosition - 1);  
 Node nodeAfter = nodeBefore.getNextNode();  
 newNode.setNextNode(nodeAfter);  
 nodeBefore.setNextNode(newNode);  
 }  
 **numberOfEntries**++;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to add operation."**);  
}

### 方法isEmpty和toArray

*/\*\*  
 \* Whether this list is empty.  
 \** ***@return*** *True if the list is empty, or false if not.  
 \*/*@Override  
**public boolean** isEmpty()  
{  
 **boolean** result;  
 **if** (**numberOfEntries** == 0)  
 {  
 **assert firstNode** == **null**;  
 result = **true**;  
 }  
 **else** {  
 **assert firstNode** != **null**;  
 result = **false**;  
 }  
 **return** result;  
}  
  
  
*/\*\*  
 \* Retrieves all entries that are in this list in the order in which they occur in the list.  
 \** ***@return*** *A newly allocated array of all the entries in the list. If the list is empty, the returned array is empty.  
 \*/*@Override  
**public** T[] toArray()  
{  
 @SuppressWarnings(**"unchecked"**)  
 T[] result = (T[]) **new** Object[**numberOfEntries**];  
 **int** index = 0;  
 Node currentNode = **firstNode**;  
 **while** ((index < **numberOfEntries**) && (currentNode != **null**))  
 {  
 result[index] = currentNode.getData();  
 currentNode = currentNode.getNextNode();  
 index++;  
 }  
 **return** result;  
}

### 方法replace

**public** T replace(**int** givenPosition, T newEntry)  
{  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 Node desiredNode = getNodeAt(givenPosition);  
 T originalEntry = desiredNode.getData();  
 desiredNode.setData(newEntry);  
 **return** originalEntry;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to replace operation."**);  
}

### 方法getEntry

**public** T getEntry(**int** givenPosition)  
{  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 **return** getNodeAt(givenPosition).getData();  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to getEntry operation."**);  
}

### 方法contains

**public boolean** contains(T anEntry)  
{  
 **boolean** found = **false**;  
 Node currentNode = **firstNode**;  
 **while** (!found && (currentNode != **null**))  
 {  
 **if** (anEntry.equals(currentNode.getData()))  
 found = **true**;  
 **else** currentNode = currentNode.getNextNode();  
 }  
 **return** found;  
}

### 完整代码



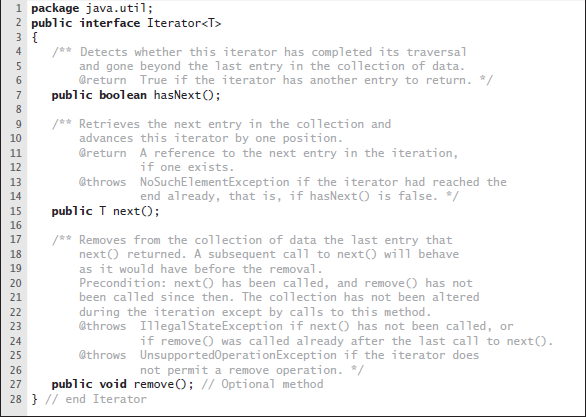
**package** Lists;  
  
*/\*\*  
 \* A linked implementation of the ADT list.  
 \** ***@param <T>*** *\*/***public class** LList<T> **implements** ListInterface<T>  
{  
 **private** Node **firstNode**;  
 **private int numberOfEntries**;  
  
 **public** LList()  
 {  
 initializeDataFields();  
 }  
  
 **public void** clear()  
 {  
 initializeDataFields();  
 }  
  
 **private void** initializeDataFields()  
 {  
 **firstNode** = **null**;  
 **numberOfEntries** = 0;  
 }  
  
  
 */\*\*  
 \* Returns a reference to the node at a given position.  
 \** ***@param givenPosition*** *\** ***@return*** *\* Precondition: The chain is not empty; 1 <= givenPosition <= numberOfEntries.  
 \*/* **private** Node getNodeAt(**int** givenPosition)  
 {  
 **assert** (**firstNode** != **null**) && (givenPosition >= 1) && (givenPosition <= **numberOfEntries**);  
 Node currentNode = **firstNode**;  
  
 *// Traverse the chain to locate the desired node, skipped if givenPosition is 1* **for** (**int** counter = 1; counter < givenPosition; counter++)  
 currentNode = currentNode.getNextNode();  
 **assert** currentNode != **null**;  
  
 **return** currentNode;  
 }  
  
  
 */\*\*  
 \* Adds a new entry to the end of this list.  
 \** ***@param newEntry*** *The object to be added as a new entry.  
 \*/* @Override  
 **public void** add(T newEntry)  
 {  
 Node newNode = **new** Node(newEntry);  
 **if** (isEmpty())  
 **firstNode** = newNode;  
 **else** {  
 Node lastNode = getNodeAt(**numberOfEntries**);  
 lastNode.setNextNode(newNode);  
 }  
 **numberOfEntries**++;  
 }  
  
  
 */\*\*  
 \* Adds a new entry at a specified position within this list.  
 \** ***@param newPosition*** *\** ***@param newEntry*** *\** ***@throws*** *IndexOutOfBoundsException if either newPosition < 1 or newPosition > getLength() + 1.  
 \*/* @Override  
 **public void** add(**int** newPosition, T newEntry)  
 {  
 **if** ((newPosition >= 1) && (newPosition <= **numberOfEntries** + 1))  
 {  
 Node newNode = **new** Node(newEntry);  
 **if** (newPosition == 1)  
 {  
 newNode.setNextNode(**firstNode**);  
 **firstNode** = newNode;  
 }  
 **else** {  
 Node nodeBefore = getNodeAt(newPosition - 1);  
 Node nodeAfter = nodeBefore.getNextNode();  
 newNode.setNextNode(nodeAfter);  
 nodeBefore.setNextNode(newNode);  
 }  
 **numberOfEntries**++;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to add operation."**);  
 }  
  
  
 */\*\*  
 \* Whether this list is empty.  
 \** ***@return*** *True if the list is empty, or false if not.  
 \*/* @Override  
 **public boolean** isEmpty()  
 {  
 **boolean** result;  
 **if** (**numberOfEntries** == 0)  
 {  
 **assert firstNode** == **null**;  
 result = **true**;  
 }  
 **else** {  
 **assert firstNode** != **null**;  
 result = **false**;  
 }  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Retrieves all entries that are in this list in the order in which they occur in the list.  
 \** ***@return*** *A newly allocated array of all the entries in the list. If the list is empty, the returned array is empty.  
 \*/* @Override  
 **public** T[] toArray()  
 {  
 @SuppressWarnings(**"unchecked"**)  
 T[] result = (T[]) **new** Object[**numberOfEntries**];  
 **int** index = 0;  
 Node currentNode = **firstNode**;  
 **while** ((index < **numberOfEntries**) && (currentNode != **null**))  
 {  
 result[index] = currentNode.getData();  
 currentNode = currentNode.getNextNode();  
 index++;  
 }  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Removes the entry at a given position from this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the removed entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T remove(**int** givenPosition)  
 {  
 T result = **null**;  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 **if** (givenPosition == 1)  
 {  
 result = **firstNode**.getData();  
 **firstNode** = **firstNode**.getNextNode();  
 }  
 **else** {  
 Node nodeBefore = getNodeAt(givenPosition - 1);  
 Node nodeToRemove = nodeBefore.getNextNode();  
 result = nodeToRemove.getData();  
 Node nodeAfter = nodeToRemove.getNextNode();  
 nodeBefore.setNextNode(nodeAfter);  
 }  
 **numberOfEntries**--;  
 **return** result;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to remove operation."**);  
 }  
  
  
 */\*\*  
 \* Replaces the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@param newEntry*** *\** ***@return*** *The original entry that was replaced.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T replace(**int** givenPosition, T newEntry)  
 {  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 Node desiredNode = getNodeAt(givenPosition);  
 T originalEntry = desiredNode.getData();  
 desiredNode.setData(newEntry);  
 **return** originalEntry;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to replace operation."**);  
 }  
  
  
 */\*\*  
 \* Retrieves the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the indicated entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T getEntry(**int** givenPosition)  
 {  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 **return** getNodeAt(givenPosition).getData();  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to getEntry operation."**);  
 }  
  
  
 */\*\*  
 \* Whether this list contains a given entry.  
 \** ***@param anEntry*** *\** ***@return*** *True if the list contains anEntry, or false if not.  
 \*/* @Override  
 **public boolean** contains(T anEntry)  
 {  
 **boolean** found = **false**;  
 Node currentNode = **firstNode**;  
 **while** (!found && (currentNode != **null**))  
 {  
 **if** (anEntry.equals(currentNode.getData()))  
 found = **true**;  
 **else** currentNode = currentNode.getNextNode();  
 }  
 **return** found;  
 }  
  
  
 **public int** getLength()  
 {  
 **return numberOfEntries**;  
 }  
  
 **private class** Node  
 {  
 **private** T **data**;  
 **private** Node **next**;  
  
 **private** Node(T dataPortion)  
 {  
 **this**(dataPortion, **null**);  
 }  
  
 **private** Node(T dataPortion, Node nextNode)  
 {  
 **data** = dataPortion;  
 **next** = nextNode;  
 }  
  
 **private** T getData(){**return data**;}  
 **private void** setData(T newData){**data** = newData;}  
  
 **private** Node getNextNode(){**return next**;}  
 **private void** setNextNode(Node nextNode){**next** = nextNode;}  
 }  
}

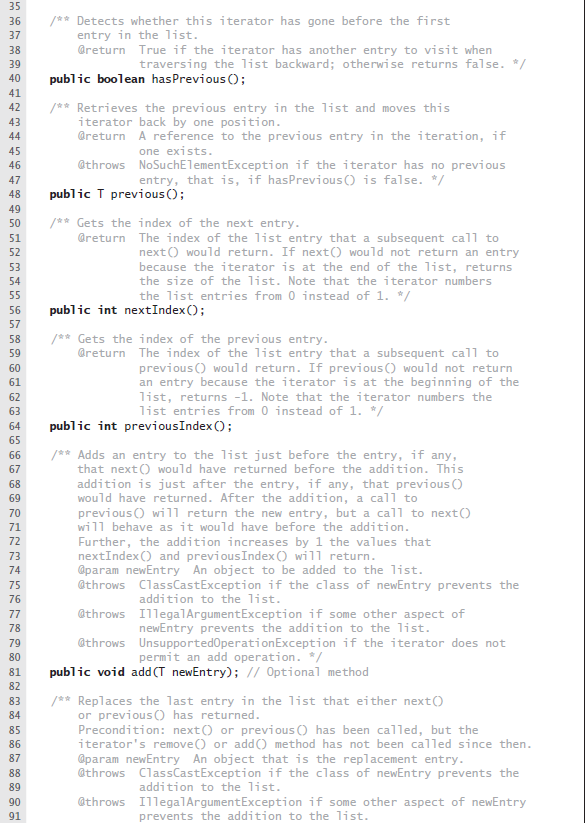
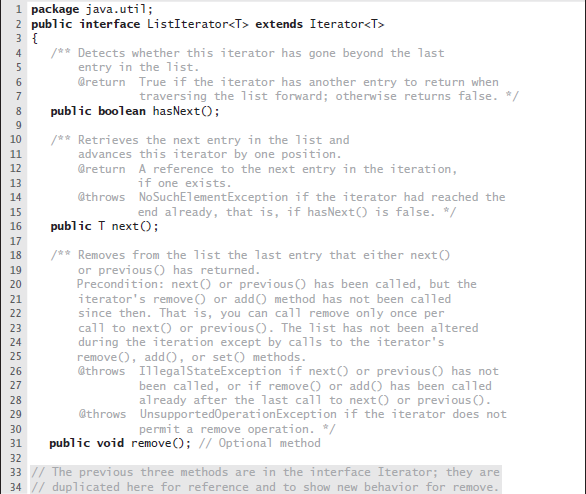
### 细化实现

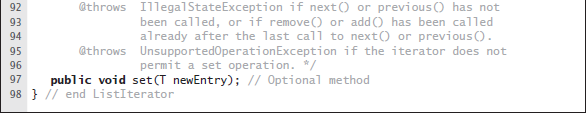
中328；英P415

## 迭代器

迭代器是一个能遍历数据集合(collection)的对象。在遍历过程中，可以查看数据项、修改数据项、添加数据项及删除数据项。Java类库含有两个接口*Iterator*和*ListIterator*，它们说明了用于迭代器的方法。







## ADT线性表的迭代器

这个类是公有的，且独立于所说的实现ADT的类。当然两个类必须以某种方式交互。我们将这样的迭代器类的实例称为独立类迭代器(separate class iterator)。另一种方法是，迭代器类可以是实现ADT的类的私有内层类。我们将这个内层类的实例称为内层类迭代器(inner class iterator)。内层类迭代器更好一些

### 独立类迭代器



**package** Lists;  
  
**import** java.util.Iterator;  
**import** java.util.NoSuchElementException;  
  
**public class** SeparateIterator<T> **implements** Iterator<T>  
{  
 **private** ListInterface<T> **list**;  
 **private int nextPosition**; *// Position of entry last returned by next()* **private boolean wasNextCalled**; *// Needed by remove* **public** SeparateIterator(ListInterface<T> myList)  
 {  
 **list** = myList;  
 **nextPosition** = 0;  
 **wasNextCalled** = **false**;  
 }  
  
 @Override  
 **public boolean** hasNext()  
 {  
 **return nextPosition** < **list**.getLength();  
 }  
  
 @Override  
 **public** T next()  
 {  
 **if** (hasNext())  
 {  
 **wasNextCalled** = **true**;  
 **nextPosition**++;  
 **return list**.getEntry(**nextPosition**);  
 }  
 **else  
 throw new** NoSuchElementException(**"Illegal call to next(), iterator is after end of list."**);  
 }  
  
 @Override  
 **public void** remove()  
 {  
 **if** (**wasNextCalled**)  
 {  
 *// nextPosition was incremented by the call to next(),  
 // so it is the position number of the entry to be removed* **list**.remove(**nextPosition**);  
 **nextPosition**--;  
 **wasNextCalled** = **false**;  
 }  
 **else  
 throw new** IllegalStateException(**"Illegal call to remove()."**);  
 }  
}

独立类迭代器必须使用ADT的公有方法访问ADT的数据。但是，某些ADT，例如栈，没有提供足够的能够访问数据的公有访问方法，所以不能有这种方式的迭代器。

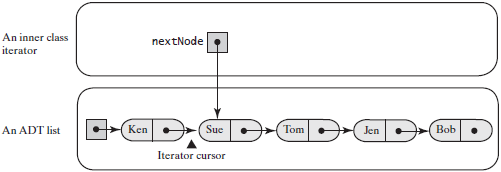
### 内层类迭代器

通过使用独立类迭代器，可以同时对线性表进行多个不同的迭代。但是独立类迭代器属于一个公有类，所以它们仅能通过ADT操作来间接访问线性表的数据域。因此，比起其他类型的迭代，它花更多的时间。对于一个不是线性表的ADT，独立类迭代器对进行迭代的数据域的访问能力不足。

另一种选择是将迭代器类定义为ADT的内层类。因为得到的迭代器对象不是ADT的对象，所以在同一时间可以存在多个迭代过程。另外，因为迭代器属于内层类，所以它能直接访问ADT的数据域。由于这些原因，内层类迭代器通常比独立类迭代器更可取。

#### 链式实现

内层迭代器可以直接访问ADT的底层数据结构——链表：





**package** Lists;  
  
**import** java.util.Iterator;  
  
*//public interface ListWithIteratorInterface<T> extends ListInterface<T>, Iterable<T>***public interface** ListWithIteratorInterface<T> **extends** ListInterface<T>  
{  
 **public** Iterator<T> getIterator();  
}



**package** Lists;  
  
**import** java.util.Iterator;  
**import** java.util.NoSuchElementException;  
  
**public class** LinkedListWithIterator<T> **implements** ListWithIteratorInterface<T>  
{  
 **private** Node **firstNode**;  
 **private int numberOfEntries**;  
  
  
 **public** LinkedListWithIterator()  
 {  
 initializeDataFields();  
 }  
  
 **private void** initializeDataFields()  
 {  
 **firstNode** = **null**;  
 **numberOfEntries** = 0;  
 }  
  
  
 **public** Iterator<T> getIterator()  
 {  
 **return new** IteratorForLinkedList();  
 }  
  
  
 **private class** IteratorForLinkedList **implements** Iterator<T>  
 {  
 **private** Node **nextNode**; *// 记录迭代数据* **private** IteratorForLinkedList()  
 {  
 **nextNode** = **firstNode**;  
 }  
  
 **public** T next()  
 {  
 **if** (hasNext())  
 {  
 Node returnNode = **nextNode**; *// Get next node* **nextNode** = **nextNode**.getNextNode();  
 **return** returnNode.getData();  
 }  
 **else  
 throw new** NoSuchElementException(**"Illegal call to next(); iterator is after end of list."**);  
 }  
  
 **public boolean** hasNext()  
 {  
 **return nextNode** != **null**;  
 }  
  
 *// An iterator that does not allow the removal of items during a traversal is not unusual.* **public void** remove()  
 {  
 **throw new** UnsupportedOperationException(**"remove() is not supported by this iterator."**);  
 }  
 }  
  
  
 **public void** clear()  
 {  
 initializeDataFields();  
 }  
  
 */\*\*  
 \* Returns a reference to the node at a given position.  
 \** ***@param givenPosition*** *\** ***@return*** *\* Precondition: The chain is not empty; 1 <= givenPosition <= numberOfEntries.  
 \*/* **private** Node getNodeAt(**int** givenPosition)  
 {  
 **assert** (**firstNode** != **null**) && (givenPosition >= 1) && (givenPosition <= **numberOfEntries**);  
 Node currentNode = **firstNode**;  
  
 *// Traverse the chain to locate the desired node, skipped if givenPosition is 1* **for** (**int** counter = 1; counter < givenPosition; counter++)  
 currentNode = currentNode.getNextNode();  
 **assert** currentNode != **null**;  
  
 **return** currentNode;  
 }  
  
  
 */\*\*  
 \* Adds a new entry to the end of this list.  
 \** ***@param newEntry*** *The object to be added as a new entry.  
 \*/* @Override  
 **public void** add(T newEntry)  
 {  
 Node newNode = **new** Node(newEntry);  
 **if** (isEmpty())  
 **firstNode** = newNode;  
 **else** {  
 Node lastNode = getNodeAt(**numberOfEntries**);  
 lastNode.setNextNode(newNode);  
 }  
 **numberOfEntries**++;  
 }  
  
  
 */\*\*  
 \* Adds a new entry at a specified position within this list.  
 \** ***@param newPosition*** *\** ***@param newEntry*** *\** ***@throws*** *IndexOutOfBoundsException if either newPosition < 1 or newPosition > getLength() + 1.  
 \*/* @Override  
 **public void** add(**int** newPosition, T newEntry)  
 {  
 **if** ((newPosition >= 1) && (newPosition <= **numberOfEntries** + 1))  
 {  
 Node newNode = **new** Node(newEntry);  
 **if** (newPosition == 1)  
 {  
 newNode.setNextNode(**firstNode**);  
 **firstNode** = newNode;  
 }  
 **else** {  
 Node nodeBefore = getNodeAt(newPosition - 1);  
 Node nodeAfter = nodeBefore.getNextNode();  
 newNode.setNextNode(nodeAfter);  
 nodeBefore.setNextNode(newNode);  
 }  
 **numberOfEntries**++;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to add operation."**);  
 }  
  
  
 */\*\*  
 \* Whether this list is empty.  
 \** ***@return*** *True if the list is empty, or false if not.  
 \*/* @Override  
 **public boolean** isEmpty()  
 {  
 **boolean** result;  
 **if** (**numberOfEntries** == 0)  
 {  
 **assert firstNode** == **null**;  
 result = **true**;  
 }  
 **else** {  
 **assert firstNode** != **null**;  
 result = **false**;  
 }  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Retrieves all entries that are in this list in the order in which they occur in the list.  
 \** ***@return*** *A newly allocated array of all the entries in the list. If the list is empty, the returned array is empty.  
 \*/* @Override  
 **public** T[] toArray()  
 {  
 @SuppressWarnings(**"unchecked"**)  
 T[] result = (T[]) **new** Object[**numberOfEntries**];  
 **int** index = 0;  
 Node currentNode = **firstNode**;  
 **while** ((index < **numberOfEntries**) && (currentNode != **null**))  
 {  
 result[index] = currentNode.getData();  
 currentNode = currentNode.getNextNode();  
 index++;  
 }  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Removes the entry at a given position from this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the removed entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T remove(**int** givenPosition)  
 {  
 T result = **null**;  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 **if** (givenPosition == 1)  
 {  
 result = **firstNode**.getData();  
 **firstNode** = **firstNode**.getNextNode();  
 }  
 **else** {  
 Node nodeBefore = getNodeAt(givenPosition - 1);  
 Node nodeToRemove = nodeBefore.getNextNode();  
 result = nodeToRemove.getData();  
 Node nodeAfter = nodeToRemove.getNextNode();  
 nodeBefore.setNextNode(nodeAfter);  
 }  
 **numberOfEntries**--;  
 **return** result;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to remove operation."**);  
 }  
  
  
 */\*\*  
 \* Replaces the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@param newEntry*** *\** ***@return*** *The original entry that was replaced.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T replace(**int** givenPosition, T newEntry)  
 {  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 Node desiredNode = getNodeAt(givenPosition);  
 T originalEntry = desiredNode.getData();  
 desiredNode.setData(newEntry);  
 **return** originalEntry;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to replace operation."**);  
 }  
  
  
 */\*\*  
 \* Retrieves the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the indicated entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T getEntry(**int** givenPosition)  
 {  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 **return** getNodeAt(givenPosition).getData();  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to getEntry operation."**);  
 }  
  
  
 */\*\*  
 \* Whether this list contains a given entry.  
 \** ***@param anEntry*** *\** ***@return*** *True if the list contains anEntry, or false if not.  
 \*/* @Override  
 **public boolean** contains(T anEntry)  
 {  
 **boolean** found = **false**;  
 Node currentNode = **firstNode**;  
 **while** (!found && (currentNode != **null**))  
 {  
 **if** (anEntry.equals(currentNode.getData()))  
 found = **true**;  
 **else** currentNode = currentNode.getNextNode();  
 }  
 **return** found;  
 }  
  
  
 **public int** getLength()  
 {  
 **return numberOfEntries**;  
 }  
  
  
 **private class** Node  
 {  
 **private** T **data**;  
 **private** Node **next**;  
  
 **private** Node(T dataPortion)  
 {  
 **this**(dataPortion, **null**);  
 }  
  
 **private** Node(T dataPortion, Node nextNode)  
 {  
 **data** = dataPortion;  
 **next** = nextNode;  
 }  
  
 **private** T getData(){**return data**;}  
 **private void** setData(T newData){**data** = newData;}  
  
 **private** Node getNextNode(){**return next**;}  
 **private void** setNextNode(Node nextNode){**next** = nextNode;}  
 }  
}

#### 数组实现

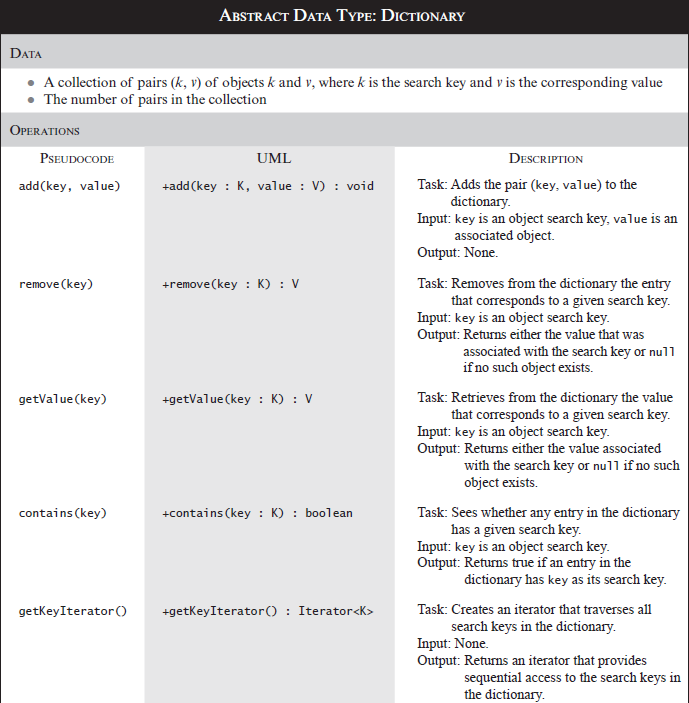


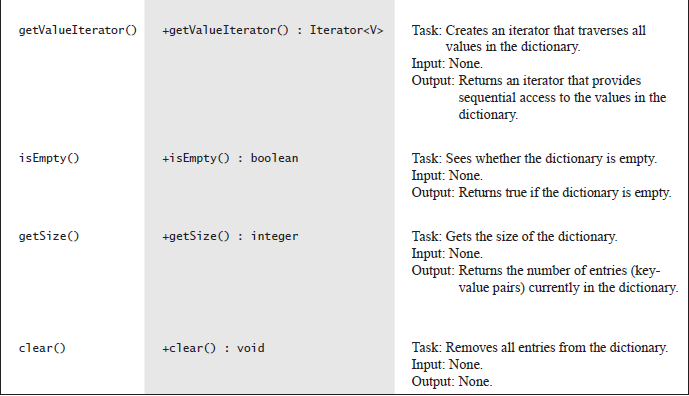
**package** Lists;  
  
**import** java.util.Arrays;  
**import** java.util.Iterator;  
**import** java.util.NoSuchElementException;  
  
**public class** ArrayListWithIterator<T> **implements** ListWithIteratorInterface<T>  
{  
 **private** T[] **list**; *// Array of list entries; ignore list[0]* **private int numberOfEntries**;  
 **private boolean initialized** = **false**;  
 **private static final int *DEFAULT\_CAPACITY*** = 25;  
 **private static final int *MAX\_CAPACITY*** = 10000;  
  
 **public** ArrayListWithIterator()  
 {  
 **this**(***DEFAULT\_CAPACITY***);  
 }  
  
 **public** ArrayListWithIterator(**int** initialCapacity)  
 {  
 **if** (initialCapacity < ***DEFAULT\_CAPACITY***)  
 initialCapacity = ***DEFAULT\_CAPACITY***;  
 **else** checkCapacity(initialCapacity);  
  
 @SuppressWarnings(**"unchecked"**)  
 T[] tempList = (T[]) **new** Object[initialCapacity + 1];  
 **list** = tempList;  
 **numberOfEntries** = 0;  
 **initialized** = **true**;  
 }  
  
  
 **public** Iterator<T> getIterator()  
 {  
 **return new** IteratorForArrayList();  
 }  
  
 **private class** IteratorForArrayList **implements** Iterator<T>  
 {  
 **private int nextIndex**;  
 **private boolean wasNextCalled**;  
  
 **private** IteratorForArrayList()  
 {  
 **nextIndex** = 1;  
 **wasNextCalled** = **false**;  
 }  
  
 **public boolean** hasNext()  
 {  
 **return nextIndex** <= **numberOfEntries**;  
 }  
  
 **public** T next()  
 {  
 **if** (hasNext())  
 {  
 **wasNextCalled** = **true**;  
 T nextEntry = **list**[**nextIndex**];  
 **nextIndex**++;  
 **return** nextEntry;  
 }  
 **else  
 throw new** NoSuchElementException(**"Illegal call to next()."**);  
 }  
  
 **public void** remove()  
 {  
 **if** (**wasNextCalled**)  
 {  
 *// nextIndex was incremented by the call to next, so it is  
 // 1 larger than the position number of the entry to be removed* ArrayListWithIterator.**this**.remove(**nextIndex** - 1);  
 **nextIndex**--;  
 **wasNextCalled** = **false**;  
 }  
 **else  
 throw new** IllegalStateException(**"Illegal call to remove()"**);  
 }  
 }  
  
 */\*\*  
 \* Whether this list contains a given entry.  
 \** ***@param anEntry*** *\** ***@return*** *True if the list contains anEntry, or false if not.  
 \*/* @Override  
 **public boolean** contains(T anEntry)  
 {  
 checkInitialization();  
 **boolean** found = **false**;  
 **int** index = 1;  
 **while** (!found && (index <= **numberOfEntries**))  
 {  
 **if** (anEntry.equals(**list**[index]))  
 found = **true**;  
 index++;  
 }  
 **return** found;  
 }  
  
  
 */\*\*  
 \* Replaces the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@param newEntry*** *\** ***@return*** *The original entry that was replaced.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T replace(**int** givenPosition, T newEntry)  
 {  
 checkInitialization();  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 T originalEntry = **list**[givenPosition];  
 **list**[givenPosition] = newEntry;  
 **return** originalEntry;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to replace operation."**);  
 }  
  
  
 */\*\*  
 \* Retrieves the entry at a given position in this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the indicated entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T getEntry(**int** givenPosition)  
 {  
 checkInitialization();  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 **return list**[givenPosition];  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to getEntry operation."**);  
 }  
  
  
 */\*\*  
 \* Removes the entry at a given position from this list.  
 \** ***@param givenPosition*** *\** ***@return*** *A reference to the removed entry.  
 \** ***@throws*** *IndexOutOfBoundsException if either givenPosition < 1 or givenPosition > getLength().  
 \*/* @Override  
 **public** T remove(**int** givenPosition)  
 {  
 checkInitialization();  
 **if** ((givenPosition >= 1) && (givenPosition <= **numberOfEntries**))  
 {  
 **assert** !isEmpty();  
 T result = **list**[givenPosition]; *// Get entry to be removed  
  
 // Move subsequent entries toward entry to be removed, unless it is last in list* **if** (givenPosition < **numberOfEntries**)  
 removeGap(givenPosition);  
 **numberOfEntries**--;  
 **return** result;  
 }  
 **else  
 throw new** IndexOutOfBoundsException(**"Illegal position given to remove operation."**);  
 }  
  
  
 */\*\*  
 \* Shifts entries that are beyond the entry to be removed to the next lower position.  
 \* Precondition: 1 <= givenPosition < numberOfEntries;  
 \* numberOfEntries is list’s length before removal;  
 \* checkInitialization has been called.  
 \** ***@param givenPosition*** *\*/* **private void** removeGap(**int** givenPosition)  
 {  
 **assert** (givenPosition >= 1) && (givenPosition < **numberOfEntries**);  
 **int** removedIndex = givenPosition;  
 **int** lastIndex = **numberOfEntries**;  
 **for** (**int** index = removedIndex; index < lastIndex; index++)  
 **list**[index] = **list**[index + 1];  
 }  
  
  
 */\*\*  
 \* Retrieves all entries that are in this list in the order in which they occur in the list.  
 \** ***@return*** *A newly allocated array of all the entries in the list. If the list is empty, the returned array is empty.  
 \*/* @Override  
 **public** T[] toArray()  
 {  
 checkInitialization();  
 @SuppressWarnings(**"unchecked"**)  
 T[] result = (T[]) **new** Object[**numberOfEntries**];  
 **for** (**int** index = 0; index < **numberOfEntries**; index++)  
 result[index] = **list**[index + 1];  
 **return** result;  
 }  
  
  
 */\*\*  
 \* Display all entries in a list  
 \** ***@param list*** *\*/* **public static void** displayList(ListInterface<String> list)  
 {  
 System.***out***.println(**"The list contains "** + list.getLength() + **" entries, as follows: "**);  
 Object[] listArray = list.toArray();  
 **for** (**int** index = 0; index < listArray.**length**; index++)  
 System.***out***.print(listArray[index] + **" "**);  
 }  
  
  
 */\*\*  
 \* Adds a new entry to the end of this list.  
 \** ***@param newEntry*** *The object to be added as a new entry.  
 \*/* @Override  
 **public void** add(T newEntry)  
 {  
 checkInitialization();  
 *//list[numberOfEntries++] = newEntry;* **list**[**numberOfEntries** + 1] = newEntry;  
 **numberOfEntries**++;  
 ensureCapacity();  
 }  
  
  
 */\*\*  
 \* Adds a new entry at a specified position within this list.  
 \** ***@param newPosition*** *\** ***@param newEntry*** *\** ***@throws*** *IndexOutOfBoundsException if either newPosition < 1 or newPosition > getLength() + 1.  
 \*/* @Override  
 **public void** add(**int** newPosition, T newEntry)  
 {  
 checkInitialization();  
 **if** ((newPosition >= 1) && (newPosition <= **numberOfEntries** + 1))  
 {  
 **if** (newPosition <= **numberOfEntries**)  
 makeRoom(newPosition);  
 **list**[newPosition] = newEntry;  
 **numberOfEntries**++;  
 ensureCapacity(); *// Ensure enough room for next add* }  
 **else  
 throw new** IndexOutOfBoundsException(**"Given position of add's new entry is out of bounds."**);  
 }  
  
  
 */\*\*  
 \* Makes room for a new entry at newPosition.  
 \** ***@param newPosition*** *\* Precondition: 1 <= newPosition <= numberOfEntries + 1;  
 \* numberOfEntries is list’s length before addition;  
 \* checkInitialization has been called.  
 \*/* **private void** makeRoom(**int** newPosition)  
 {  
 **assert** (newPosition >= 1) && (newPosition <= **numberOfEntries** + 1);  
 **int** newIndex = newPosition;  
 **int** lastIndex = **numberOfEntries**;  
  
 *// Move each entry to next higher index, starting at end of list and continuing until the entry at newIndex is moved* **for** (**int** index = lastIndex; index >= newIndex; index--)  
 **list**[index + 1] = **list**[index];  
 }  
  
  
 */\*\*  
 \* Doubles the capacity of the array list if it is full.  
 \* Precondition: checkInitialization has been called.  
 \* 假定数组中至少还有添加一次的空间，一旦添加使得数组满了，就扩大数组  
 \*/* **private void** ensureCapacity()  
 {  
 **int** capacity = **list**.**length** - 1; *// 假定数组中至少还有添加一次的空间* **if** (**numberOfEntries** >= capacity)  
 {  
 **int** newCapacity = 2 \* capacity;  
 checkCapacity(newCapacity);  
 **list** = Arrays.*copyOf*(**list**, newCapacity + 1);  
 }  
 }  
  
  
 **private void** checkCapacity(**int** capacity)  
 {  
 **if** (capacity > ***MAX\_CAPACITY***)  
 **throw new** IllegalStateException(**"Exceed allowed maximum "** + ***MAX\_CAPACITY***);  
 }  
  
  
 **private void** checkInitialization()  
 {  
 **if** (!**initialized**)  
 **throw new** SecurityException(**"List is not initialized properly!"**);  
 }  
  
  
 **public boolean** isEmpty()  
 {  
 **return numberOfEntries** == 0;  
 }  
  
 **public int** getLength()  
 {  
 **return numberOfEntries**;  
 }  
  
 **public void** clear()  
 {  
 **while** (!isEmpty())  
 {  
 **int** index = **numberOfEntries** - 1;  
 remove(index);  
 }  
 }  
}

# 字典

ADT字典也称映射(map)、表(table)或关联数组，它包含由两部分构成的项：关键字(查找键)、与键对应的值。

## ADT字典的主要操作







**package** Dictionary;  
  
**import** java.util.Iterator;  
  
*/\*\*  
 \* An interface for a dictionary with distinct search keys.  
 \** ***@param <K>*** *\** ***@param <V>*** *\*/***public interface** DictionaryInterface<K, V>  
{  
 */\*\*  
 \* Adds a new entry to this dictionary. If the given search key already  
 \* exists in the dictionary, replaces the corresponding value.  
 \** ***@param key*** *An object search key of the new entry.  
 \** ***@param value*** *An object associated with the search key.  
 \** ***@return*** *Either null if the new entry was added to the dictionary  
 \* or the value that was associated with key if that value was replaced.  
 \*/* **public** V add(K key, V value);  
  
  
 */\*\*  
 \* Removes a specific entry from this dictionary.  
 \** ***@param key*** *An object search key of the entry to be removed.  
 \** ***@return*** *Either the value that was associated with the search key  
 \* or null if no such object exists.  
 \*/* **public** V remove(K key);  
  
  
 */\*\*  
 \* Retrieves from this dictionary the value associated with a given search key.  
 \** ***@param key*** *An object search key of the entry to be retrieved.  
 \** ***@return*** *Either the value that is associated with the search key  
 \* or null if no such object exists.  
 \*/* **public** V getValue(K key);  
  
  
 */\*\*  
 \* Sees whether a specific entry is in this dictionary.  
 \** ***@param key*** *An object search key of the desired entry.  
 \** ***@return*** *True if key is associated with an entry in the dictionary.  
 \*/* **public boolean** contains(K key);  
  
  
 */\*\*  
 \* Creates an iterator that traverses all search keys in this dictionary.  
 \** ***@return*** *An iterator that provides sequential access to the search keys in the dictionary.  
 \*/* **public** Iterator<K> getKeyIterator();  
  
  
 */\*\*  
 \* Creates an iterator that traverses all values in this dictionary.  
 \** ***@return*** *An iterator that provides sequential access to the values in this dictionary.  
 \*/* **public** Iterator<V> getValueIterator();  
  
  
 */\*\*  
 \* Sees whether this dictionary is empty.  
 \** ***@return*** *True if the dictionary is empty.  
 \*/* **public boolean** isEmpty();  
  
  
 */\*\*  
 \* Gets the size of this dictionary.  
 \** ***@return*** *The number of entries (key-value pairs) currently in the dictionary.  
 \*/* **public int** getSize();  
  
  
 */\*\*  
 \* Removes all entries from this dictionary.  
 \*/* **public void** clear();  
}

### 实现实例

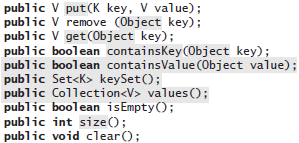
DictionaryInterface<String, Student> database = new Dictionary<>();

Iterator<String> keyIterator = database.getKeyIterator();

Iterator<Student> valueIterator = database.getValueIterator();

## Java类库：接口Map

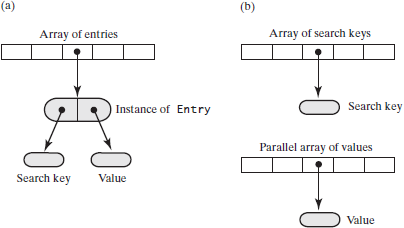
java.util含有接口Map<K, V>。



## 字典的实现

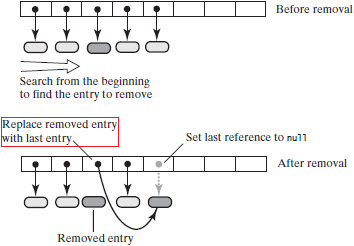
### 基于数组的实现

使用数组表示字典中的项有两种方法：每个项包含两部分——一个查找键和一个值。可以将这两部分封装到一个对象中。或使用并行数组——两个数组分别表示查找键和对应的值。



#### 基于数组的无序字典

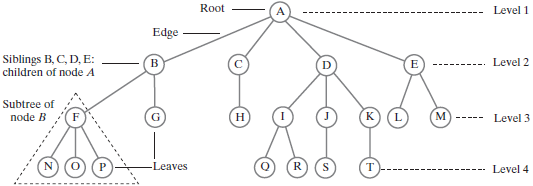
##### remove方法



**public** V remove(K key)  
{  
 checkInitialization();  
 V result = **null**;  
 **int** keyIndex = locateIndex(key);  
 **if** (keyIndex < **numberOfEntries**)  
 {  
 result = **dictionary**[keyIndex].getValue();  
 **dictionary**[keyIndex] = **dictionary**[**numberOfEntries** - 1];  
 **dictionary**[**numberOfEntries** - 1] = **null**;  
 **numberOfEntries**--;  
 }  
 **return** result;  
}

# 树

树(tree)是一组由边(edge)相连的结点(node)，边表示结点之间的关系。结点按层组织，层表示结点的层次。最上层的单结点称为根。



树的每个后继层中的结点是前一层中的结点的孩子(children)。有孩子的结点称为其孩子的父结点(parent)。结点A是结点B、C、D 和E的父结点。因为这些孩子有相同的父结点，所以它们称为兄弟(sibling)。它们也称为结点A的后代(descendant)，而结点A是它们的祖先(ancestor)。结点P没有孩子，这样的结点称为叶子(leaf)。非叶子结点(即有孩子的结点)称为内部结点(interior)或非叶子结点(nonleaf)。这样的结点也是父结点。

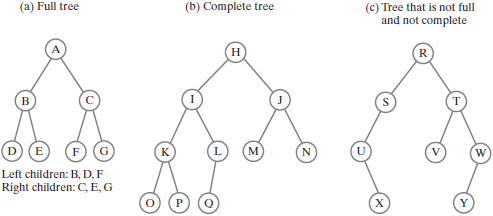
一般地，树中的每个结点可以有任意多个孩子。有时称这样的树为一般树(general tree)。如果每个结点的孩子不多于n个，则该树称为n叉树(n-ary tree)。不是每棵一般树都是n叉树。如果每个结点最多有两个孩子，则该树称为二叉树(binary tree)。

树的高度(height)是树中的层。

## 几种树

### 二叉树

二叉树中的每个结点最多有两个孩子，它们称为左孩子(left child) 和右孩子(right child)。二叉树的每棵子树还是二叉树。



### 满二叉树和完全二叉树

高度为h的二叉树中，若其所有的叶子结点都在h层上，且每个非叶子结点(父)都恰有两个孩子，则该树称为满树(full)。

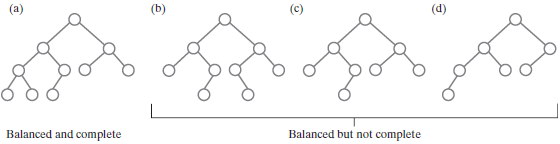
如果二叉树中除最后一层外的所有层都含有最多的结点，最后一层的结点从左至右填充，则该树是完全树。

满二叉树中的所有叶子结点都在同一层中，每个非叶子结点都恰有两个孩子。在完全二又树中，到倒数笫二层都是满的，且最后一层的叶子结点从左至右填充。

### 平衡二叉树

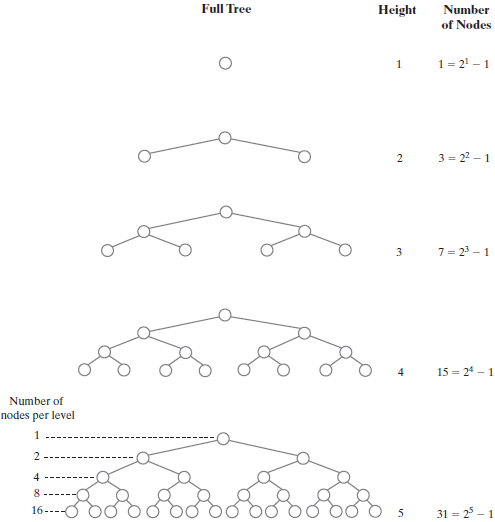
若二叉树中的每个结点有两棵高度完全相等的子树，则该树称为完全平衡树(completely balanced)。唯一的完全平衡二叉树是满树。

如果树中的每个结点的子树的高度差不大于1，则树称为高度平衡的(height balanced)，或简称为平衡的(balanced)。



在二叉树中，其子树的高度差不大于1的结点称为平衡结点(balanced node)。所以平衡二叉树中的所有结点都是平衡的。

### 满树或完全树的高度



含n个结点的满树的高度是；含n个结点的完全树的高度是对向上取整。

## 树的遍历

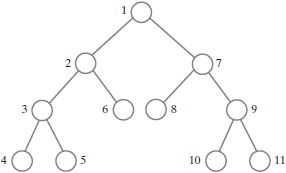
在树的遍历或迭代中，对每个数据项的访问或处埋次数必须恰好一次，不过对项的访问次序不是唯一的。遍历可以经过一个结点但在那个时刻并不访问。

### 二叉树的遍历

二叉树的树根的子树还是二叉树，因此利用二叉树具有的递归特性来定义遍历是很自然的。

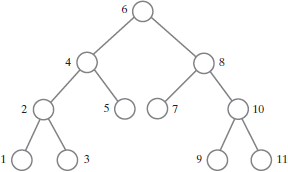
#### 前序遍历preorder traversal

在访问根的子树之前访问根。然后访问根的左子树中的所有结点，再访问根的右子树中的所有结点。



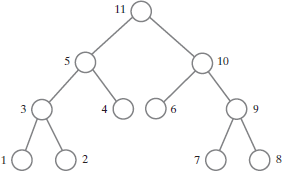
#### 中序遍历inordcr traversal

访问二叉树的根的子树中间访问二叉树的根。访问根的左子树中的所有结点；访问根；访问根的右子树中的所有结点。



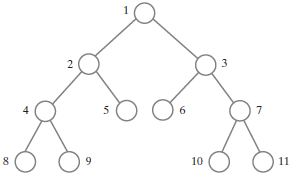
#### 后序遍历postorder traversal

在访问了二叉树的根的子树中的结点之后访问树的根。访问根的左子树中的所有结点；访问根的右子树中的所有结点；访问根。



#### 层序遍历level-order traversal

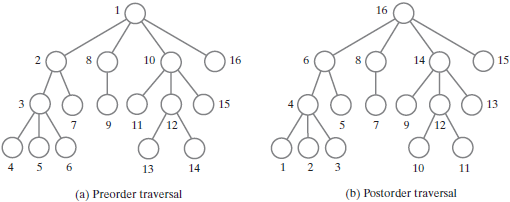
从根开始，每次访问一层中的结点。在同一层中，从左至右访问结点。



层序遍历是广度优先遍历(breadth-first traversal)的示例；前序遍历是深度优先遍历(depth-first traversal)。

#### 一般树的遍历

一般树的遍历有层序、前序和后序。对一般树而言，中序遍历不好定义。



## 树的Java接口

### 所有树的接口



**package** Tree;  
  
**public interface** TreeInterface<T>  
{  
 **public** T getRootData();  
 **public int** getHeight();  
 **public int** getNumberOfNode();  
 **public boolean** isEmpty();  
 **public void** clear();  
}

遍历树的一种方法是使用有方法hasNext和next的迭代器、这由接口java.util.Iterator提供。可以定义一个方法，返回一个迭代器。因为有多种不同的遍历，所以树类可以有多个方法，每个方认返回一种不同的迭代器。



**package** Tree;  
  
**import** java.util.Iterator;  
  
**public interface** TreeIteratorInterface<T>  
{  
 **public** Iterator<T> getPreOrderIterator();  
 **public** Iterator<T> getInOrderIterator();  
 **public** Iterator<T> getPostOrderIterator();  
 **public** Iterator<T> getLevelOrderIterator();  
}

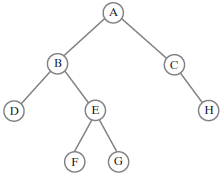
### 二叉树的接口

可以为基本的二叉树定义一个接口，并将接口TreeInterface和TreeIteratorInterface 中已有的方法添加进来。因为Java接口可以从多个接口派生。



**package** Tree;  
  
**public interface** BinaryTreeInterface<T> **extends** TreeInterface<T>, TreeIteratorInterface<T>  
{  
 */\*\*  
 \* Sets this binary tree to a new one-node binary tree.  
 \** ***@param rootData*** *The object that is the data for the new tree's root.  
 \*/* **public void** setTree(T rootData);  
  
  
 */\*\*  
 \* Sets this binary tree to a new binary tree.  
 \** ***@param rootData*** *The object that is the data for the new tree's root.  
 \** ***@param leftTree*** *The left subtree of the new tree.  
 \** ***@param rightTree*** *The right subtree of the new tree.  
 \*/* **public void** setTree(T rootData, BinaryTreeInterface<T> leftTree, BinaryTreeInterface<T> rightTree);  
}

两个setTree方法将巳有的二叉树对象转换为一棵由己给参数组成的新树。第一个方法从给出的数据对象来建立一棵单结点树。第二个方法建立一棵树，该树的根结点包括所给出的数据对象，并且两棵给定的二叉树是其子树。



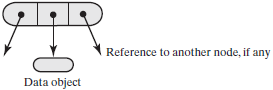
先创建叶子结点：



## 树的实现

树的最常见实现是使用链式结构。

### 二叉树中的结点



二叉树中结点含有一个指向数据对象的引用和指向左孩子及右孩子的引用、左孩子及右孩子是树中的其他结点。指向孩子的两个引用都可以是null。如果它们全是null，则该结点是叶子结点。

### 二叉结点类

#### 方法getNumberOfNodes

*/\*\*  
 \* Counts the nodes in the subtree rooted at this node.  
 \** ***@return*** *The number of nodes in the subtree rooted at this node.  
 \*/***public int** getNumberOfNodes()  
{  
 **int** leftNumber = 0;  
 **int** rightNumber = 0;  
 **if** (**leftChild** != **null**)  
 leftNumber = **leftChild**.getNumberOfNodes();  
 **if** (**rightChild** != **null**)  
 rightNumber = **rightChild**.getNumberOfNodes();  
 **return** leftNumber + rightNumber + 1;  
}

#### 方法getHeight

*/\*\*  
 \* Computes the height of the subtree rooted at this node.  
 \** ***@return*** *The height of the subtree rooted at this node.  
 \*/***public int** getHeight(){**return** getHeight(**this**);}  
  
**private int** getHeight(BinaryNode<T> node)  
{  
 **int** height = 0;  
 **if** (node != **null**)  
 height = 1 + Math.*max*(getHeight(node.getLeftChild()), getHeight(node.getRightChild()));  
 **return** height;  
}

#### 方法copy

*/\*\*  
 \* Copies the subtree rooted at this node.  
 \** ***@return*** *The root of a copy of the subtree rooted at this node.  
 \*/***public** BinaryNode<T> copy()  
{  
 BinaryNode<T> newRoot = **new** BinaryNode<>(**data**);  
 **if** (**leftChild** != **null**)  
 newRoot.setLeftChild(**leftChild**.copy());  
 **if** (**rightChild** != **null**)  
 newRoot.setRightChild(**rightChild**.copy());  
 **return** newRoot;  
}

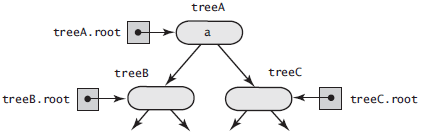
#### 完整代码



**package** Tree;  
  
**public class** BinaryNode<T>  
{  
 **private** T **data**;  
 **private** BinaryNode<T> **leftChild**;  
 **private** BinaryNode<T> **rightChild**;  
  
 **public** BinaryNode()  
 {  
 **this**(**null**);  
 }  
  
 **public** BinaryNode(T dataPortion)  
 {  
 **this**(dataPortion, **null**, **null**);  
 }  
  
 **public** BinaryNode(T dataPortion, BinaryNode<T> newLeftChild, BinaryNode<T> newRightChild)  
 {  
 **data** = dataPortion;  
 **leftChild** = newLeftChild;  
 **rightChild** = newRightChild;  
 }  
  
  
 */\*\*  
 \* Retrieves the data portion of this node.  
 \** ***@return*** *The object in the data portion of the node.  
 \*/* **public** T getData(){**return data**;}  
  
  
 */\*\*  
 \* Sets the data portion of this node.  
 \** ***@param newData*** *The data object.  
 \*/* **public void** setData(T newData){**data** = newData;}  
  
  
 */\*\*  
 \* Retrieves the left child(node) of this node.  
 \** ***@return*** *The node that is this node's left child.  
 \*/* **public** BinaryNode<T> getLeftChild(){**return leftChild**;}  
  
  
 */\*\*  
 \* Sets this node's left child to a given node.  
 \** ***@param newLeftChild*** *The node that is this node's left child.  
 \*/* **public void** setLeftChild(BinaryNode<T> newLeftChild){**leftChild** = newLeftChild;}  
  
  
 */\*\*  
 \* Detects whether this node has a left child.  
 \** ***@return*** *True if the node has a left child.  
 \*/* **public boolean** hasLeftChild(){**return leftChild** != **null**;}  
  
  
 */\*\*  
 \* Detects whether this node is a leaf.  
 \** ***@return*** *True if the node is a leaf.  
 \*/* **public boolean** isLeaf(){**return** (**leftChild** == **null**) && (**rightChild** == **null**);}  
  
  
 */\*\*  
 \* Retrieves the right child(node) of this node.  
 \** ***@return*** *The node that is this node's right child.  
 \*/* **public** BinaryNode<T> getRightChild(){**return rightChild**;}  
  
  
 */\*\*  
 \* Sets this node's right child to a given node.  
 \** ***@param newRightChild*** *The node that is this node's right child.  
 \*/* **public void** setRightChild(BinaryNode<T> newRightChild){**rightChild** = newRightChild;}  
  
  
 */\*\*  
 \* Detects whether this node has a right child.  
 \** ***@return*** *True if the node has a right child.  
 \*/* **public boolean** hasRightChild(){**return rightChild** != **null**;}  
  
  
 */\*\*  
 \* Counts the nodes in the subtree rooted at this node.  
 \** ***@return*** *The number of nodes in the subtree rooted at this node.  
 \*/* **public int** getNumberOfNodes()  
 {  
 **int** leftNumber = 0;  
 **int** rightNumber = 0;  
 **if** (**leftChild** != **null**)  
 leftNumber = **leftChild**.getNumberOfNodes();  
 **if** (**rightChild** != **null**)  
 rightNumber = **rightChild**.getNumberOfNodes();  
 **return** leftNumber + rightNumber + 1;  
 }  
  
  
 */\*\*  
 \* Computes the height of the subtree rooted at this node.  
 \** ***@return*** *The height of the subtree rooted at this node.  
 \*/* **public int** getHeight(){**return** getHeight(**this**);}  
  
 **private int** getHeight(BinaryNode<T> node)  
 {  
 **int** height = 0;  
 **if** (node != **null**)  
 height = 1 + Math.*max*(getHeight(node.getLeftChild()), getHeight(node.getRightChild()));  
 **return** height;  
 }  
  
  
 */\*\*  
 \* Copies the subtree rooted at this node.  
 \** ***@return*** *The root of a copy of the subtree rooted at this node.  
 \*/* **public** BinaryNode<T> copy()  
 {  
 BinaryNode<T> newRoot = **new** BinaryNode<>(**data**);  
 **if** (**leftChild** != **null**)  
 newRoot.setLeftChild(**leftChild**.copy());  
 **if** (**rightChild** != **null**)  
 newRoot.setRightChild(**rightChild**.copy());  
 **return** newRoot;  
 }  
}

### ADT二叉树的实现

由treeB和treeC建立treeA， treeA与treeB和treeC共享结点，所以因为改变treeB而改变了treeA。这是不欢迎的。



一种解决办法是复制treeB和treeC中的结点，那么treeA独立于treeB和treeC。以后不论是对treeB还是对treeC的改变都不会影响treeA。但复制结点的开销大。

#### 方法privateSetTree

1. 用给定的数据创建根结点r；
2. 如果左子树存在且不为空，则将它的根结点链接为r的左孩子；
3. 如果右子树存在且不为空，且与左子树不是同一棵树，则将它的根结点链接为r的右孩子。但如果右子树和左子树是相同的、则将右子树的复制，链接在r上。
4. 如果左子树存在，且与调用privateSetTree的树对象不是同一棵树，则将该子树的数据域root设置为null。
5. 如果右子树存在，且与调用privateSetTree的树对象不是同一棵树，则将该子树的数据域root设置为null。

**private void** privateSetTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree)  
{  
 **root** = **new** BinaryNode<>(rootData);  
 **if** (leftTree != **null**)  
 **root**.setLeftChild(leftTree.**root**);  
 **if** (rightTree != **null**)  
 **root**.setRightChild(rightTree.**root**);  
}

改进🡪

**private void** privateSetTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree)  
{  
 **root** = **new** BinaryNode<>(rootData);  
 **if** ((leftTree != **null**) && !leftTree.isEmpty())  
 **root**.setLeftChild(leftTree.**root**);  
 **if** ((rightTree != **null**) && !rightTree.isEmpty())  
 {  
 **if** (rightTree != leftTree)  
 **root**.setRightChild(rightTree.**root**);  
 **else  
 root**.setRightChild(rightTree.**root**.copy());  
 }  
 **if** ((leftTree != **null**) && (leftTree != **this**))  
 leftTree.clear();  
 **if** ((rightTree != **null**) && (rightTree != **this**))  
 rightTree.clear();  
}

#### 实现访问和赋值方法

@Override  
**public boolean** isEmpty()  
{  
 **return root** == **null**;  
}  
  
@Override  
**public void** clear()  
{  
 **root** = **null**;  
}  
  
@Override  
**public** T getRootData()  
{  
 **if** (isEmpty())  
 **throw new** EmptyTreeException();  
 **else  
 return root**.getData();  
}  
  
**public void** setRootData(T rootData){ **root**.setData(rootData);}  
  
**protected void** setRootNode(BinaryNode<T> rootNode){**root** = rootNode;}  
**protected** BinaryNode<T> getRootNode(){**return root**;}

#### 计算高度和结点个数

调用BinaryNode中的getHeight和getNumberOfNodes方法：

@Override  
**public int** getHeight(){**return root**.getHeight();}  
  
@Override  
**public int** getNumberOfNodes(){**return root**.getNumberOfNodes();}

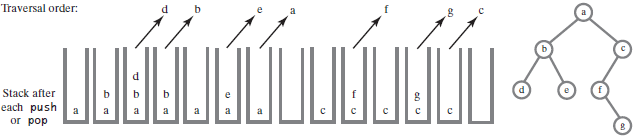
#### 遍历

##### 递归地遍历二叉树

*/\*\*  
 \* 中序遍历  
 \*/***public void** inOrderTraverse(){inOrderTraverse(**root**);}  
**private void** inOrderTraverse(BinaryNode<T> node)  
{  
 **if** (node != **null**)  
 {  
 inOrderTraverse(node.getLeftChild());  
 System.***out***.println(node.getData());  
 inOrderTraverse(node.getRightChild());  
 }  
}  
  
  
*/\*\*  
 \* 前序遍历  
 \*/***public void** preOrderTraverse(){preOrderTraverse(**root**);}  
**private void** preOrderTraverse(BinaryNode<T> node)  
{  
 **if** (node != **null**)  
 {  
 System.***out***.println(node.getData());  
 preOrderTraverse(node.getLeftChild());  
 preOrderTraverse(node.getLeftChild());  
 }  
}  
  
  
*/\*\*  
 \* 后序遍历  
 \*/***public void** postOrderTraverse(){postOrderTraverse(**root**);}  
**private void** postOrderTraverse(BinaryNode<T> node)  
{  
 **if** (node != **null**)  
 {  
 postOrderTraverse(node.getLeftChild());  
 postOrderTraverse(node.getRightChild());  
 System.***out***.println(node.getData());  
 }  
}

像inOrderTraverse这样的方法不难实现，但该方法仅在遍历时显示数据。

##### 中序遍历迭代版本



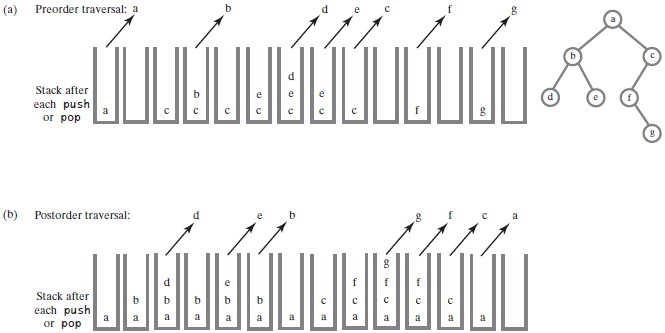
**public void** iterativeInOrderTraverse()  
{  
 StackInterface<BinaryNode<T>> nodeStack = **new** LinkedStack<>();  
 BinaryNode<T> currentNode = **root**;  
 **while** (!nodeStack.isEmpty() || (currentNode != **null**))  
 {  
 *// Find leftmost node with no left child* **while** (currentNode != **null**)  
 {  
 nodeStack.push(currentNode);  
 currentNode = currentNode.getLeftChild();  
 }  
   
 *// Visit leftmost node, then traverse its right subtree* **if** (!nodeStack.isEmpty())  
 {  
 BinaryNode<T> nextNode = nodeStack.pop();  
 **assert** nextNode != **null**; *// Since nodeStack was not empty before the pop* System.***out***.println(nextNode.getData());  
 currentNode = nextNode.getRightChild();  
 }  
 }  
}

##### 使用迭代器进行遍历

Java的接口Iterator声明了方法hasNext和next。这些方法可让客户在遍历过程中的任何时刻获取当前结点中的数据。就是说，客户可以获取一个结点的数据，对它进行操作，还可能做其他事情，然后再获取迭代中下一个结点中的数据。

@Override  
**public** Iterator<T> getInOrderIterator(){**return new** InOrderIterator();}  
  
  
**private class** InOrderIterator **implements** Iterator<T>  
{  
 **private** StackInterface<BinaryNode<T>> **nodeStack**;  
 **private** BinaryNode<T> **currentNode**;  
  
 **public** InOrderIterator()  
 {  
 **nodeStack** = **new** LinkedStack<>();  
 **currentNode** = **root**;  
 }  
  
 **public boolean** hasNext()  
 {  
 **return** !**nodeStack**.isEmpty() || (**currentNode** != **null**);  
 }  
  
 **public** T next()  
 {  
 BinaryNode<T> nextNode = **null**;  
  
 *// Find leftmost node with no left child* **while** (**currentNode** != **null**)  
 {  
 **nodeStack**.push(**currentNode**);  
 **currentNode** = **currentNode**.getLeftChild();  
 }  
  
 *// Get leftmost node, then move to its right subtree* **if** (!**nodeStack**.isEmpty())  
 {  
 nextNode = **nodeStack**.pop();  
 **assert** nextNode != **null**; *// Since nodeStack was not empty before the pop* **currentNode** = nextNode.getRightChild();  
 }  
 **else  
 throw new** NoSuchElementException();  
  
 **return** nextNode.getData();  
 }  
  
 **public void** remove()  
 {  
 **throw new** UnsupportedOperationException();  
 }  
}

##### 迭代的前序、后序和层序遍历



#### 完整代码



**package** Tree;  
  
**import** Stack.LinkedStack;  
**import** Stack.StackInterface;  
  
**import** java.util.Iterator;  
**import** java.util.NoSuchElementException;  
*// import StackAndQueuePackage.\*; // Needed by tree iterators  
  
/\*\*  
 \* A class that implements the ADT binary tree.  
 \*/***public class** BinaryTree<T> **implements** BinaryTreeInterface<T>  
{  
 **private** BinaryNode<T> **root**;  
  
 **public** BinaryTree()  
 {  
 **root** = **null**;  
 }  
  
 **public** BinaryTree(T rootData)  
 {  
 **root** = **new** BinaryNode<>(rootData);  
 }  
  
 **public** BinaryTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree)  
 {  
 privateSetTree(rootData, leftTree, rightTree);  
 }  
  
  
 */\*\*  
 \* Sets this binary tree to a new one-node binary tree.  
 \** ***@param rootData*** *The object that is the data for the new tree's root.  
 \*/* @Override  
 **public void** setTree(T rootData)  
 {  
 **root** = **new** BinaryNode<>(rootData);  
 }  
  
  
 */\*\*  
 \* Sets this binary tree to a new binary tree.  
 \** ***@param rootData*** *The object that is the data for the new tree's root.  
 \** ***@param leftTree*** *The left subtree of the new tree.  
 \** ***@param rightTree*** *The right subtree of the new tree.  
 \*/* @Override  
 **public void** setTree(T rootData, BinaryTreeInterface<T> leftTree, BinaryTreeInterface<T> rightTree)  
 {  
 privateSetTree(rootData, (BinaryTree<T>) leftTree, (BinaryTree<T>) rightTree);  
 }  
  
 **private void** privateSetTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree)  
 {  
 **root** = **new** BinaryNode<>(rootData);  
 **if** ((leftTree != **null**) && !leftTree.isEmpty())  
 **root**.setLeftChild(leftTree.**root**);  
 **if** ((rightTree != **null**) && !rightTree.isEmpty())  
 {  
 **if** (rightTree != leftTree)  
 **root**.setRightChild(rightTree.**root**);  
 **else  
 root**.setRightChild(rightTree.**root**.copy());  
 }  
 **if** ((leftTree != **null**) && (leftTree != **this**))  
 leftTree.clear();  
 **if** ((rightTree != **null**) && (rightTree != **this**))  
 rightTree.clear();  
 }  
  
  
 @Override  
 **public boolean** isEmpty()  
 {  
 **return root** == **null**;  
 }  
  
 @Override  
 **public void** clear()  
 {  
 **root** = **null**;  
 }  
  
 @Override  
 **public** T getRootData()  
 {  
 **if** (isEmpty())  
 **throw new** EmptyTreeException();  
 **else  
 return root**.getData();  
 }  
  
 **public void** setRootData(T rootData){ **root**.setData(rootData);}  
  
 **protected void** setRootNode(BinaryNode<T> rootNode){**root** = rootNode;}  
 **protected** BinaryNode<T> getRootNode(){**return root**;}  
  
  
 @Override  
 **public int** getHeight(){**return root**.getHeight();}  
  
 @Override  
 **public int** getNumberOfNodes(){**return root**.getNumberOfNodes();}  
  
  
 */\*\*  
 \* 中序遍历  
 \*/* **public void** inOrderTraverse(){inOrderTraverse(**root**);}  
 **private void** inOrderTraverse(BinaryNode<T> node)  
 {  
 **if** (node != **null**)  
 {  
 inOrderTraverse(node.getLeftChild());  
 System.***out***.println(node.getData());  
 inOrderTraverse(node.getRightChild());  
 }  
 }  
  
  
 */\*\*  
 \* 前序遍历  
 \*/* **public void** preOrderTraverse(){preOrderTraverse(**root**);}  
 **private void** preOrderTraverse(BinaryNode<T> node)  
 {  
 **if** (node != **null**)  
 {  
 System.***out***.println(node.getData());  
 preOrderTraverse(node.getLeftChild());  
 preOrderTraverse(node.getLeftChild());  
 }  
 }  
  
  
 */\*\*  
 \* 后序遍历  
 \*/* **public void** postOrderTraverse(){postOrderTraverse(**root**);}  
 **private void** postOrderTraverse(BinaryNode<T> node)  
 {  
 **if** (node != **null**)  
 {  
 postOrderTraverse(node.getLeftChild());  
 postOrderTraverse(node.getRightChild());  
 System.***out***.println(node.getData());  
 }  
 }  
  
  
 */\*\*  
 \* 中序遍历迭代版本  
 \*/* **public void** iterativeInOrderTraverse()  
 {  
 StackInterface<BinaryNode<T>> nodeStack = **new** LinkedStack<>();  
 BinaryNode<T> currentNode = **root**;  
 **while** (!nodeStack.isEmpty() || (currentNode != **null**))  
 {  
 *// Find leftmost node with no left child* **while** (currentNode != **null**)  
 {  
 nodeStack.push(currentNode);  
 currentNode = currentNode.getLeftChild();  
 }  
  
 *// Visit leftmost node, then traverse its right subtree* **if** (!nodeStack.isEmpty())  
 {  
 BinaryNode<T> nextNode = nodeStack.pop();  
 **assert** nextNode != **null**; *// Since nodeStack was not empty before the pop* System.***out***.println(nextNode.getData());  
 currentNode = nextNode.getRightChild();  
 }  
 }  
 }  
  
  
 @Override  
 **public** Iterator<T> getInOrderIterator(){**return new** InOrderIterator();}  
  
  
 **private class** InOrderIterator **implements** Iterator<T>  
 {  
 **private** StackInterface<BinaryNode<T>> **nodeStack**;  
 **private** BinaryNode<T> **currentNode**;  
  
 **public** InOrderIterator()  
 {  
 **nodeStack** = **new** LinkedStack<>();  
 **currentNode** = **root**;  
 }  
  
 **public boolean** hasNext()  
 {  
 **return** !**nodeStack**.isEmpty() || (**currentNode** != **null**);  
 }  
  
 **public** T next()  
 {  
 BinaryNode<T> nextNode = **null**;  
  
 *// Find leftmost node with no left child* **while** (**currentNode** != **null**)  
 {  
 **nodeStack**.push(**currentNode**);  
 **currentNode** = **currentNode**.getLeftChild();  
 }  
  
 *// Get leftmost node, then move to its right subtree* **if** (!**nodeStack**.isEmpty())  
 {  
 nextNode = **nodeStack**.pop();  
 **assert** nextNode != **null**; *// Since nodeStack was not empty before the pop* **currentNode** = nextNode.getRightChild();  
 }  
 **else  
 throw new** NoSuchElementException();  
  
 **return** nextNode.getData();  
 }  
  
 **public void** remove()  
 {  
 **throw new** UnsupportedOperationException();  
 }  
 }  
}

## 二叉查找树的实现

二叉查找树( binary search tree)是一棵二叉树，其结点含有Comparable对象，并按下列规则组织，对树中的每个结点：

• 结点中的数据大于结点的左子树中的所有数据

• 结点中的数据小于(等于)结点的右子树中的所有数据。

Comparable对象属于实现了接口Comparable的类。使用该类的方法compareTo来比较这样的对象。基于compareTo检测的数据域，各个类进行比较时，使用的基准也不同。

### 二叉查找树的接口



**package** Tree;  
  
**import** java.util.Iterator;  
  
**public interface** SearchTreeInterface<T **extends** Comparable<? **super** T>> **extends** TreeInterface<T>  
{  
 */\*\*  
 \* Searches for a specific entry in this tree.  
 \** ***@param entry*** *An object to be found.  
 \** ***@return*** *True if the object was found in the tree.  
 \*/* **public boolean** contains(T entry);  
  
  
 */\*\*  
 \* Retrieves a specific entry in this tree.  
 \** ***@param entry*** *An object to be found.  
 \** ***@return*** *Either the object that was found in the tree or null if no such object exists.  
 \*/* **public** T getEntry(T entry);  
  
  
 */\*\*  
 \* Adds a new entry to this tree, if it does not match an existing object in the tree.  
 \* Otherwise, replaces the existing object with the new entry.  
 \** ***@param newEntry*** *An object to be added to the tree.  
 \** ***@return*** *Either null if newEntry was not in the tree already,  
 \* or the existing entry that matched the parameter newEntry  
 \* and has been replaced in the tree.  
 \*/* **public** T add(T newEntry);  
  
  
 */\*\*  
 \* Removes a specific entry from this tree.  
 \** ***@param entry*** *An object to be removed.  
 \** ***@return*** *Either the object that was removed from the tree or null if no such object exists.  
 \*/* **public** T remove(T entry);  
  
  
 */\*\*  
 \* Creates an iterator that traverses all entries in this tree.  
 \** ***@return*** *An iterator that provides sequential and ordered access to the entries in the tree.  
 \*/* **public** Iterator<T> getInOrderIterator();  
}

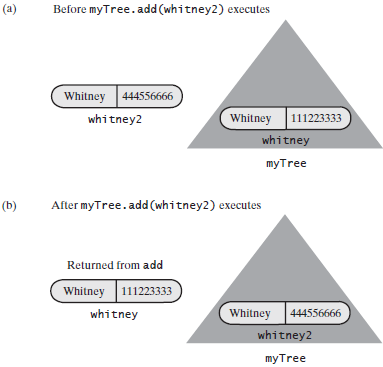
SearchTreeInterface<Person> myTree = new BinarySearchTree<>();

Person whitney = new Person("Whitney", "111223333");

Person returnValue = myTree.add(whitney); // null

Person whitney2 = new Person("Whitney", "444556666");

returnValue = myTree.add(whitney2);



### 方法getEntry和contains

*/\*\*  
 \* Retrieves a specific entry in this tree.  
 \** ***@param entry*** *An object to be found.  
 \** ***@return*** *Either the object that was found in the tree or null if no such object exists.  
 \*/*@Override  
**public** T getEntry(T entry)  
{  
 **return** findEntry(getRootNode(), entry);  
}  
  
**private** T findEntry(BinaryNode<T> rootNode, T entry)  
{  
 T result = **null**;  
 **if** (rootNode != **null**)  
 {  
 T rootEntry = rootNode.getData();  
 **if** (entry.equals(rootEntry))  
 result = rootEntry;  
 **else if** (entry.compareTo(rootEntry) < 0)  
 result = findEntry(rootNode.getLeftChild(), entry);  
 **else** result = findEntry(rootNode.getRightChild(), entry);  
 }  
   
 **return** result;  
}  
  
  
*/\*\*  
 \* Searches for a specific entry in this tree.  
 \** ***@param entry*** *An object to be found.  
 \** ***@return*** *True if the object was found in the tree.  
 \*/*@Override  
**public boolean** contains(T entry)  
{  
 **return** getEntry(entry) != **null**;  
}

### 添加一项

二叉查找树的每次添加都是添加了树的一个叶子结点。

#### 递归recursive实现

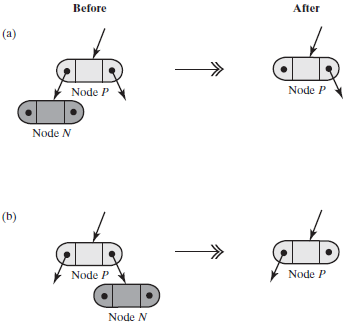
*/\*\*  
 \* Adds a new entry to this tree, if it does not match an existing object in the tree.  
 \* Otherwise, replaces the existing object with the new entry.  
 \* 说明，可以用二叉查找树来实现字典  
 \** ***@param newEntry*** *An object to be added to the tree.  
 \** ***@return*** *\*/*@Override  
**public** T add(T newEntry)  
{  
 T result = **null**;  
 **if** (isEmpty())  
 setRootNode(**new** BinaryNode<>(newEntry));  
 **else** result = addEntry(getRootNode(), newEntry);  
   
 **return** result;  
}  
  
  
*/\*\*  
 \* Adds newEntry to the nonempty subtree rooted at rootNode.  
 \** ***@param rootNode*** *\** ***@param newEntry*** *\** ***@return*** *\*/***private** T addEntry(BinaryNode<T> rootNode, T newEntry)  
{  
 **assert** rootNode != **null**;  
 T result = **null**;  
 **int** comparison = newEntry.compareTo(rootNode.getData());  
 **if** (comparison == 0)  
 {  
 result = rootNode.getData();  
 rootNode.setData(newEntry);  
 }  
 **else if** (comparison < 0)  
 {  
 **if** (rootNode.hasLeftChild())  
 result = addEntry(rootNode.getLeftChild(), newEntry);  
 **else** rootNode.setLeftChild(**new** BinaryNode<>(newEntry));  
 }  
 **else** {  
 **assert** comparison > 0;  
 **if** (rootNode.hasRightChild())  
 result = addEntry(rootNode.getRightChild(), newEntry);  
 **else** rootNode.setRightChild(**new** BinaryNode<>(newEntry));  
 }  
   
 **return** result;  
}

#### 迭代iterative实现

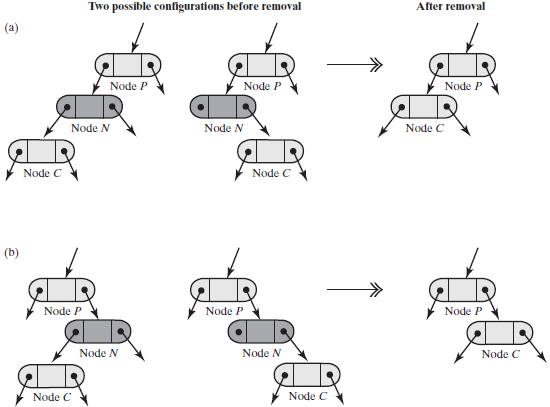
**private** T addEntry(T newEntry)  
{  
 BinaryNode<T> currentNode = getRootNode();  
 **assert** currentNode != **null**;  
 T result = **null**;  
 **boolean** found = **false**;  
 **while** (!found)  
 {  
 T currentEntry = currentNode.getData();  
 **int** comparison = newEntry.compareTo(currentEntry);  
  
 **if** (comparison == 0)  
 {  
 *// newEntry matches currentEntry; return and replace currentEntry* found = **true**;  
 result = currentEntry;  
 currentNode.setData(newEntry);  
 }  
 **else if** (comparison < 0)  
 {  
 **if** (currentNode.hasLeftChild())  
 currentNode = currentNode.getLeftChild();  
 **else** {  
 found = **true**;  
 currentNode.setLeftChild(**new** BinaryNode<>(newEntry));  
 }  
 }  
 **else** {  
 **assert** comparison > 0;  
 **if** (currentNode.hasRightChild())  
 currentNode = currentNode.getRightChild();  
 **else** {  
 found = **true**;  
 currentNode.setRightChild(**new** BinaryNode<>(newEntry));  
 }  
 }  
 }  
   
 **return** result;  
}

### 删除一项

#### 删除叶子结点中的项

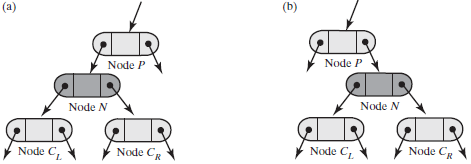


#### 删除仅有一个孩子的结点中的项



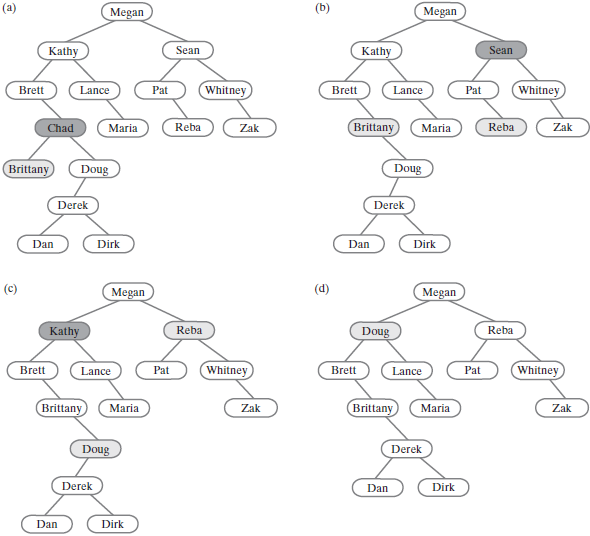
#### 删除有两个孩子的结点中的项

假定要删除的顶在结点N中，而N有两个孩子：



如果删除了结点N，则会让它的两个孩子都没有父结点。虽然结点P可以指向其中的一个，但它不能提供两个空间。所以删除结点N不是好的做法。

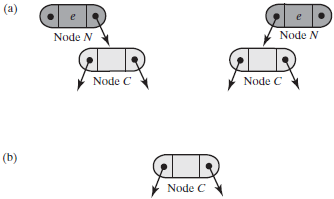
我们的目标是将项从树中删除，实际上不必为了删除项而删除结点N。要删除有两个孩子结点中的项，首先用另一个最多有1个孩子的结点中的项来替换这个项。然后从二叉查找树中删除第二个结点。



(a) A binary search tree; (b) after removing Chad; (c) after removing Sean;(d) after removing Kathy

#### 删除根中的项

删除树根中的项是一个特例，如果我们实际上删除了根结点，当根最多只有一个孩子时才会发生这种情况。如果根有两个孩子，可以替换根中的项并删除另一个结点。



#### 递归实现

*/\*\*  
 \* Removes a specific entry from this tree.  
 \** ***@param entry*** *An object to be removed.  
 \** ***@return*** *\*/*@Override  
**public** T remove(T entry)  
{  
 ReturnObject oldEntry = **new** ReturnObject(**null**);  
 BinaryNode<T> newRoot = removeEntry(getRootNode(), entry, oldEntry);  
 setRootNode(newRoot);  
  
 **return** oldEntry.getOldEntry();  
}  
  
**private class** ReturnObject  
{  
 T **oldEntry** = **null**;  
  
 **public** ReturnObject(T newEntry)  
 {  
 **oldEntry** = newEntry;  
 }  
  
 **void** setOldEntry(T newEntry){**oldEntry** = newEntry;}  
 T getOldEntry(){**return oldEntry**;}  
}  
  
*/\*\*  
 \* Removes an entry from the tree rooted at a given node.  
 \** ***@param rootNode*** *A reference to the root of a tree.  
 \** ***@param entry*** *the object to be removed.  
 \** ***@param oldEntry*** *An object whose data field is null.  
 \** ***@return*** *the root node of the resulting tree; if entry matches an entry in the tree,  
 \* oldEntry’s data field is the entry that was removed from the tree; otherwise it is null.  
 \*/***private** BinaryNode<T> removeEntry(BinaryNode<T> rootNode, T entry, ReturnObject oldEntry)  
{  
 **if** (rootNode != **null**)  
 {  
 T rootData = rootNode.getData();  
 **int** comparison = entry.compareTo(rootData);  
 **if** (comparison == 0)  
 {  
 oldEntry.setOldEntry(rootData);  
 rootNode = removeFromRoot(rootNode);  
 }  
 **else if** (comparison < 0)  
 {  
 BinaryNode<T> leftChild = rootNode.getLeftChild();  
 BinaryNode<T> subtreeRoot = removeEntry(leftChild, entry, oldEntry);  
 rootNode.setLeftChild(subtreeRoot);  
 }  
 **else** {  
 BinaryNode<T> rightChild = rootNode.getRightChild();  
 rootNode.setRightChild(removeEntry(rightChild, entry, oldEntry));  
 }  
 }  
  
 **return** rootNode;  
}  
  
*/\*\*  
 \* Removes the entry in a given root node of a subtree.  
 \** ***@param rootNode*** *the root node of the subtree.  
 \** ***@return*** *the root node of the revised subtree.  
 \*/***private** BinaryNode<T> removeFromRoot(BinaryNode<T> rootNode)  
{  
 *// Case 1: rootNode has two children* **if** (rootNode.hasLeftChild() && rootNode.hasRightChild())  
 {  
 *// Find node with largest entry in left subtree* BinaryNode<T> leftSubtreeRoot = rootNode.getLeftChild();  
 BinaryNode<T> largestNode = findLargest(leftSubtreeRoot);  
  
 *// Replace entry in root* rootNode.setData(largestNode.getData());  
  
 *// Remove node with largest entry in left subtree* rootNode.setLeftChild(removeLargest(leftSubtreeRoot));  
 }  
 *// Case 2: rootNode has at most one child* **else if** (rootNode.hasRightChild())  
 rootNode = rootNode.getRightChild();  
 **else** rootNode = rootNode.getLeftChild();  
  
 **return** rootNode;  
}  
  
*/\*\*  
 \* Finds the node containing the largest entry in a given tree.  
 \** ***@param rootNode*** *the root node of the tree.  
 \** ***@return*** *the node containing the largest entry in the tree.  
 \*/***private** BinaryNode<T> findLargest(BinaryNode<T> rootNode)  
{  
 **if** (rootNode.hasRightChild())  
 rootNode = findLargest(rootNode.getRightChild());  
 **return** rootNode;  
}  
  
*/\*\*  
 \* Removes the node containing the largest entry in a given tree.  
 \** ***@param rootNode*** *the root node of the tree.  
 \** ***@return*** *the root node of the revised tree.  
 \*/***private** BinaryNode<T> removeLargest(BinaryNode<T> rootNode)  
{  
 **if** (rootNode.hasRightChild())  
 {  
 BinaryNode<T> rightChild = rootNode.getRightChild();  
 rightChild = removeLargest(rightChild);  
 rootNode.setRightChild(rightChild);  
 }  
 **else** rootNode = rootNode.getLeftChild();  
   
 **return** rootNode;  
}

# 图

## 术语

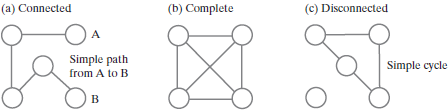
图是不同顶点(vertice)或结点(node)和不同边(edge)的集合。带有有向边的图称为有向图(directed grapgh, digraph)。

图中的两个顶点之间的路径(path)是边的序列。有向图中的路径必须考虑边的方向，故称为有向路径(directed path)。路径长度是路径所包含的边数。如果经过的顶点不重复，则称为简单路径。

回路(cycle)是开始顶点及结束顶点均为同一个顶点的路径。经过其他顶点仅一次回路的称为简单回路。没有回路的图称为无环图(acyclic)。

带权图(weighted graph)，其边上有一个值，这些值称为权(weight)或代价(cost)。

每对不同顶点之间都有路径可达的图称为连通图(connected)。完全图(complete graph)在每对不同顶点之间都有一条边。不连通的称为非连通图(disconnected)。



如果无向图中的两个顶点由边相连，则它们称为邻接的(adjacent)。邻接顶点称为邻居(neighbor)。如果存在一条起始于j终止于i的有向边，则称顶点i邻接于顶点j。



如果有向图有n个顶点，它能含有多少条边？如果图是完全图，则每个顶点都是其余所有顶点的邻居。所以每个顶点都是n-1条有向边的终止点。因此，有条边。无向完全图中的边数是这个数字的一半。

·图是有向的：条边；

·图是无向的：条边。

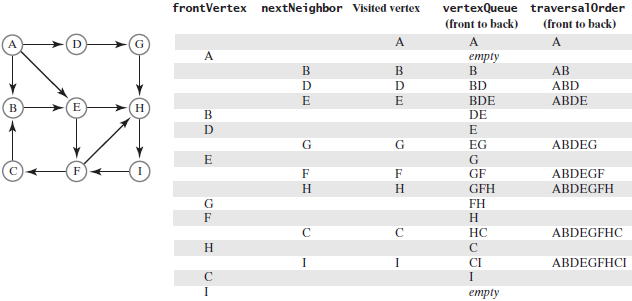
如果图含有相对较少的边，则图是稀疏图(sparse)。如果它含有很多边，则图是稠密图(dense)。

## 遍历

### 广度优先遍历

给定起始顶点，广度优先遍历访问起始顶点(origin vertex)，然后访问该起始顶点的邻居。然后考虑这些邻居的每一个，访问它的邻居。该遍历使用队列保存访问过的顶点。当从队列中删除一个顶点时，访问它并将该顶点未被访问的邻居入队(D被从队列中删除时，将D的邻居G入队列)。然后遍历次序就是顶点入队的次序。可以将这个遍历次序保存在第二个队列中。

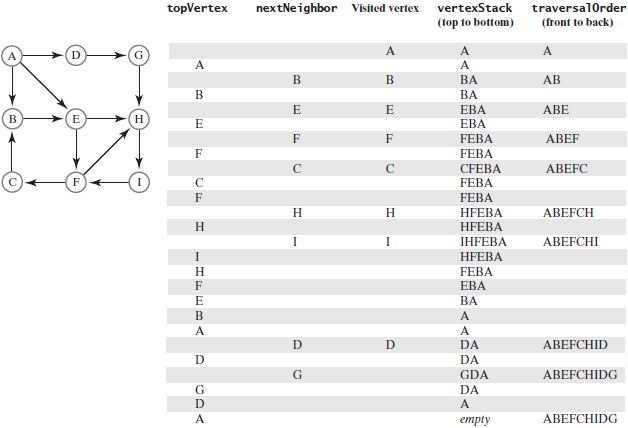
广度优先遍历访问一个顶点，然后在访问下一个顶点之前访问该顶点的每个邻居。访问邻居的次序可以任意，可以依赖于图的实现。



### 深度优先遍历

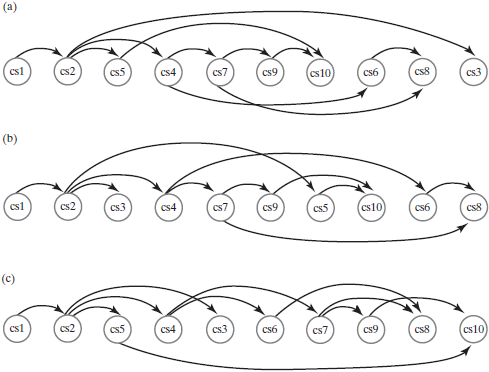
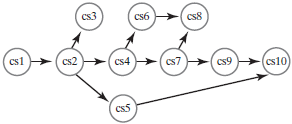
给定起始顶点，深度优先遍历访问起始顶点，然后访问该起始顶点的一个邻居，再然后访问该邻居的邻居。持续这个过程直到发现没有未访问的邻居时为止。回退一个顶点，再检查它的另一个邻居。这个遍历的递归描述时使用了栈。

初始时将起始顶点入栈。当栈顶中的顶点有未被访问的邻居时，访问它并将它的邻居入栈。如果不存在这样的邻居，则出栈。遍历次序是顶点入栈的次序。可以将这个遍历次序保存在一个队列中。



## 拓扑序

在有向无环图中，可以排列，使得顶点当从a到b之间存在有向边时顶点a排在顶点b的前面。这样排列的顶点次序称为拓扑序(topological order)。

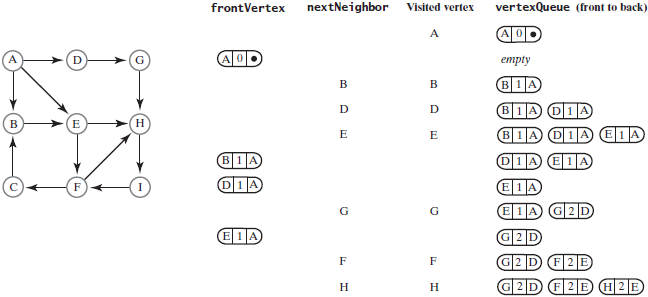


有环图不可能得到拓扑序。发现图中的顶点的拓扑序的过程称为拓扑排序(topological sort)。

## 路径

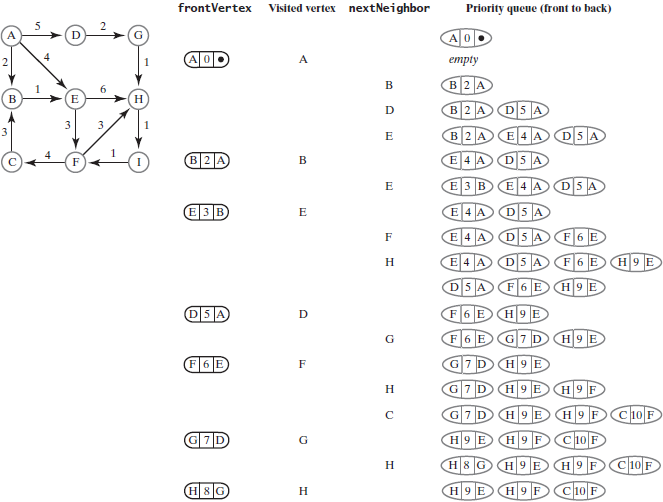
### 无权图中的最短路径

在无权图中，可以找到具有最短长度的路径，即：含有最少边数的路径。在无权图中，两个给定顶点之间的最短路径具有最短长度，即，它有最少的边。寻找这条路径的算法基于广度优先遍历。如果多条路径有同样的最短长度，则该算法将找到其中的一条。



### 带权图中的最短路径

在带权图中，两个给定顶点之间的最短路径具有最小的边权值和。寻找这条路径的算法基于广度优先遍历。带权图中的几条路径可能有相同的最小边权值和。算法将寻找这些路径中的一条。



## 图接口

### BasicGraphInterface



**package** Graphs;  
  
*/\*\*  
 \* An interface of methods providing basic operations for directed  
 \* and undirected graphs that are either weighted or unweighted.  
 \*/***public interface** BasicGraphInterface<T>  
{  
 */\*\*  
 \* Adds a given vertex to this graph.  
 \** ***@param vertexLabel*** *An object that labels the new vertex and is  
 \* distinct from the labels of current vertices.  
 \** ***@return*** *True if the vertex is added, or false if not.  
 \*/* **public boolean** addVertex(T vertexLabel);  
  
 */\*\*  
 \* Adds a weighted edge between two given distinct vertices that  
 \* are currently in this graph. The desired edge must not already  
 \* be in the graph. In a directed graph, the edge points toward  
 \* the second vertex given.  
 \** ***@param begin*** *An object that labels the origin vertex of the edge.  
 \** ***@param end*** *An object, distinct from begin, that labels the end vertex of the edge.  
 \** ***@param edgeWeight*** *The real value of the edge's weight.  
 \** ***@return*** *True if the edge is added, or false if not.  
 \*/* **public boolean** addEdge(T begin, T end, **double** edgeWeight);  
  
  
 */\*\*  
 \* Adds an unweighted edge between two given distinct vertices  
 \* that are currently in this graph. The desired edge must not already be in the graph.  
 \* In a directed graph, the edge points toward the second vertex given.  
 \** ***@param begin*** *An object that labels the origin vertex of the edge.  
 \** ***@param end*** *An object, distinct from begin, that labels the end vertex of the edge.  
 \** ***@return*** *True if the edge is added, or false if not.  
 \*/* **public boolean** addEdge(T begin, T end);  
  
  
 */\*\*  
 \* Sees whether an edge exists between two given vertices.  
 \** ***@param begin*** *An object that labels the origin vertex of the edge.  
 \** ***@param end*** *An object that labels the end vertex of the edge.  
 \** ***@return*** *True if an edge exists.  
 \*/* **public boolean** hasEdge(T begin, T end);  
  
  
 */\*\*  
 \* Sees whether this graph is empty.  
 \** ***@return*** *True if the graph is empty.  
 \*/* **public boolean** isEmpty();  
  
  
 */\*\*  
 \* Gets the number of vertices in this graph.  
 \** ***@return*** *The number of vertices in the graph.  
 \*/* **public int** getNumberOfVertices();  
  
  
 */\*\*  
 \* Gets the number of edges in this graph.  
 \** ***@return*** *The number of edges in the graph.  
 \*/* **public int** getNumberOfEdges();  
  
  
 */\*\*  
 \* Removes all vertices and edges from this graph.  
 \*/* **public void** clear();  
}

### GraphAlgorithmsInterface



**package** Graphs;  
  
**import** Queue.QueueInterface;  
**import** Stack.StackInterface;  
  
*/\*\*  
 \* An interface of methods that process an existing graph.  
 \** ***@param <T>*** *\*/***public interface** GraphAlgorithmsInterface<T>  
{  
 */\*\*  
 \* Performs a breadth-first traversal of this graph.  
 \** ***@param origin*** *An object that labels the origin vertex of the traversal.  
 \** ***@return*** *A queue of labels of the vertices in the traversal,  
 \* with the label of the origin vertex at the queue's front.  
 \*/* **public** QueueInterface<T> getBreadthFirstTraversal(T origin);  
  
  
 */\*\*  
 \* Performs a depth-first traversal of this graph.  
 \** ***@param origin*** *An object that labels the origin vertex of the traversal.  
 \** ***@return*** *A queue of labels of the vertices in the traversal,  
 \* with the label of the origin vertex at the queue's front.  
 \*/* **public** QueueInterface<T> getDepthFirstTraversal(T origin);  
  
  
 */\*\*  
 \* Performs a topological sort of the vertices in a graph without cycles.  
 \** ***@return*** *A stack of vertex labels in topological order, beginning with the stack's top.  
 \*/* **public** StackInterface<T> getTopologicalOrder();  
  
  
 */\*\*  
 \* Finds the path between two given vertices that has the shortest length.  
 \** ***@param begin*** *An object that labels the path's origin vertex.  
 \** ***@param end*** *An object that labels the path's destination vertex.  
 \** ***@param path*** *A stack of labels that is empty initially; at the completion of the method, this stack contains  
 \* the labels of the vertices along the shortest path;  
 \* the label of the origin vertex is at the top,  
 \* and the label of the destination vertex is at the bottom.  
 \** ***@return*** *The length of the shortest path.  
 \*/* **public int** getShortestPath(T begin, T end, StackInterface<T> path);  
  
  
 */\*\*  
 \* Finds the least-cost path between two given vertices.  
 \** ***@param begin*** *An object that labels the path's origin vertex.  
 \** ***@param end*** *An object that labels the path's destination vertex.  
 \** ***@param path*** *A stack of labels that is empty initially; at the completion of the method, this stack contains  
 \* the labels of the vertices along the cheapest path;  
 \* the label of the origin vertex is at the top, and  
 \* the label of the destination vertex is at the bottom.  
 \** ***@return*** *The cost of the cheapest path.  
 \*/* **public double** getCheapestPath(T begin, T end, StackInterface<T> path);  
}

### GraphInterface



**package** Graphs;  
  
**public interface** GraphInterface<T> **extends** BasicGraphInterface<T>,  
 GraphAlgorithmsInterface<T>  
{  
  
}

### 顶点的接口



**package** Graphs;  
  
**import** java.util.Iterator;  
  
**public interface** VertexInterface<T>  
{  
 */\*\*  
 \* Gets this vertex's label.  
 \** ***@return*** *The object that labels the vertex.  
 \*/* **public** T getLabel();  
  
  
 */\*\*  
 \* Marks this vertex as visited.  
 \*/* **public void** visit();  
  
 */\*\*  
 \* Removes this vertex's visited mark.  
 \*/* **public void** unvisited();  
  
  
 */\*\*  
 \* Sees whether this vertex is marked as visited.  
 \** ***@return*** *True if the vertex is visited.  
 \*/* **public boolean** isVisited();  
  
  
 */\*\*  
 \* Connects this vertex and a given vertex with a weighted edge.  
 \* The two vertices cannot be the same, and must not already have this edge between them.  
 \* In a directed graph, the edge points toward the given vertex.  
 \** ***@param endVertex*** *A vertex in the graph that ends the edge.  
 \** ***@param edgeWeight*** *A real-valued edge weight, if any.  
 \** ***@return*** *True if the edge is added, or false if not.  
 \*/* **public boolean** connect(VertexInterface<T> endVertex, **double** edgeWeight);  
  
  
 */\*\*  
 \* Connects this vertex and a given vertex with an unweighted edge.  
 \* The two vertices cannot be the same, and must not already have this edge between them.  
 \* In a directed graph, the edge points toward the given vertex.  
 \** ***@param endVertex*** *A vertex in the graph that ends the edge.  
 \** ***@return*** *True if the edge is added, or false if not.  
 \*/* **public boolean** connect(VertexInterface<T> endVertex);  
  
  
 */\*\*  
 \* Creates an iterator of this vertex's neighbors by following all edges that begin at this vertex.  
 \** ***@return*** *An iterator of the neighboring vertices of this vertex.  
 \*/* **public** Iterator<VertexInterface<T>> getNeighborIterator();  
  
  
 */\*\*  
 \* Creates an iterator of the weights of the edges to this vertex's neighbors.  
 \** ***@return*** *An iterator of edge weights for edges to neighbors of this vertex.  
 \*/* **public** Iterator<Double> getWeightIterator();  
  
  
 */\*\*  
 \* Sees whether this vertex has at least one neighbor.  
 \** ***@return*** *True if the vertex has a neighbor.  
 \*/* **public boolean** hasNeighbor();  
  
  
 */\*\*  
 \* Gets an unvisited neighbor, if any, of this vertex.  
 \** ***@return*** *Either a vertex that is an unvisited neighbor or null if no such neighbor exists.  
 \*/* **public** VertexInterface<T> getUnvisitedNeighbor();  
  
  
 */\*\*  
 \* Records the previous vertex on a path to this vertex.  
 \** ***@param predecessor*** *The vertex previous to this one along a path.  
 \*/* **public void** setPredecessor(VertexInterface<T> predecessor);  
  
  
 */\*\*  
 \* Gets the recorded predecessor of this vertex.  
 \** ***@return*** *Either this vertex's predecessor or null if no predecessor was recorded.  
 \*/* **public** VertexInterface<T> getPredecessor();  
  
  
 */\*\*  
 \* Sees whether a predecessor was recorded for this vertex.  
 \** ***@return*** *True if a predecessor was recorded.  
 \*/* **public boolean** hasPredecessor();  
  
  
 */\*\*  
 \* Records the cost of a path to this vertex.  
 \** ***@param newCost*** *The cost of the path.  
 \*/* **public void** setCost(**double** newCost);  
  
  
 */\*\*  
 \* Gets the recorded cost of the path to this vertex.  
 \** ***@return*** *The cost of the path.  
 \*/* **public double** getCost();  
}

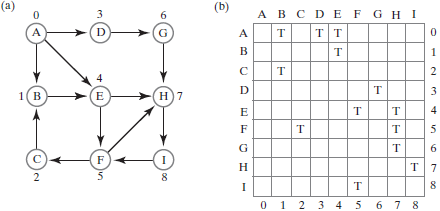
### 类Vertex的实现

## 图的实现

一般地，使用线性表或字典保存顶点，数组或线性表表示边。边的每种表示各有千秋，但线性表表示是最经典的。

### 邻接矩阵adjacency matrix

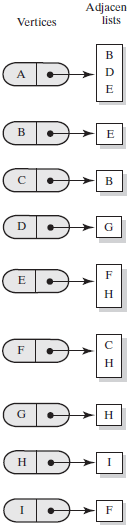
含n个顶点的图的邻接矩阵有n行n列。每行及每列对应图中的一个顶点。顶点编号为0到n-1，以匹配行下标和列下标。如果是位于矩阵i行j列的元素，则表示顶点i和顶点j之间是否存在边。对于无权图，矩阵中可以使用布尔值。对于带权图，当边存在时可以使用边的权值，否则使用无穷大。



邻接矩阵使用固定大小的空间，空间大小取决于图中的顶点数但不依赖于边数。稀疏图的邻接矩阵浪费了空间，因为图的边数相对较少。使用邻接矩阵查看图中两个给定顶点之间是否存在边是很快的。但如果你想知道某个顶点的所有邻居，则必须扫描矩阵的一整行。

### 邻接表adjacency list

给定顶点的邻接表仅表示以该顶点为起始点的边。



对于稀疏图，邻接表比邻接矩阵使用的内存更少；对于稠密图，邻接矩阵可能是更好的选择。