# 数据结构库

## 为什么要学习数据结构

### 1.1数据结构分类

1. 线性结构：数组，栈，队列，链表，哈希表；...
2. 树结构：二叉树，二分搜索树，AVL，红黑树，Treap，Splay，堆，Trie，线段树，K-D树，并查集，哈夫曼树；...
3. 图结构：邻接矩阵，邻接表；

### 1.2计算机世界的数据结构

1. 数据库：（SQL）

树结构：AVL，红黑树，Treap，伸展树，B树；

线性结构：哈希表

2）操作系统：（快速在多任务间切换）

线性结构：系统栈；

树结构：优先队列：堆；

3）文件压缩：

树结构：哈夫曼树；

1. 通讯录：

树结构：Trie - 前缀树；

1. 算法：以数据结构为基石，寻路算法（图论算法）

线性结构：DFS（深度优先遍历）：栈；

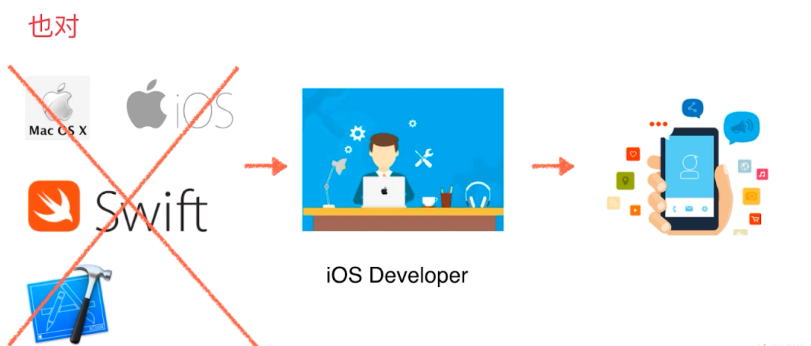
线性结构：BFS（广度优先遍历）：队列；

### 1.3数据结构目录

1. 数组，栈，队列，链表；
2. 二分搜索树，堆，线段树，Trie；
3. 并查集，AVL，红黑树，哈希表；

### 1.4数据结构有什么作用



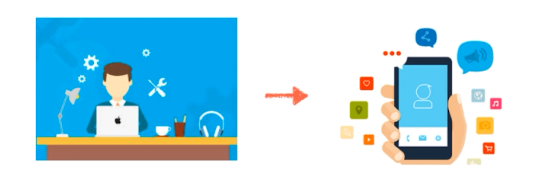






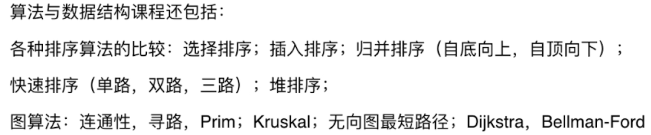


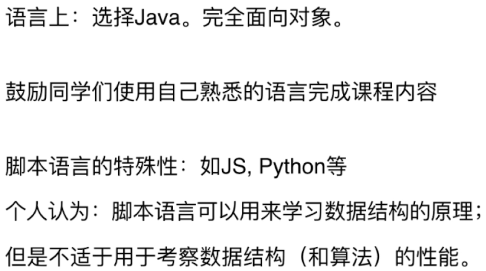


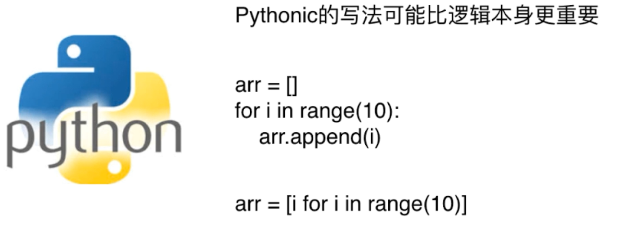


### 1.5注意事项

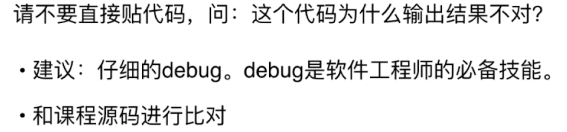


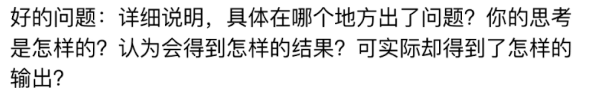


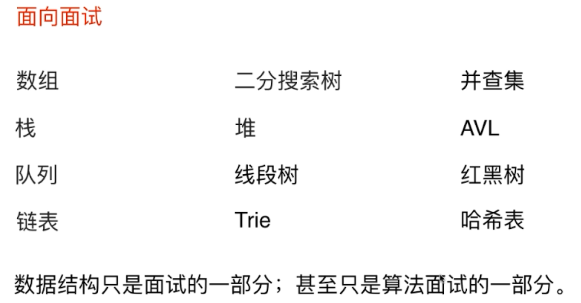


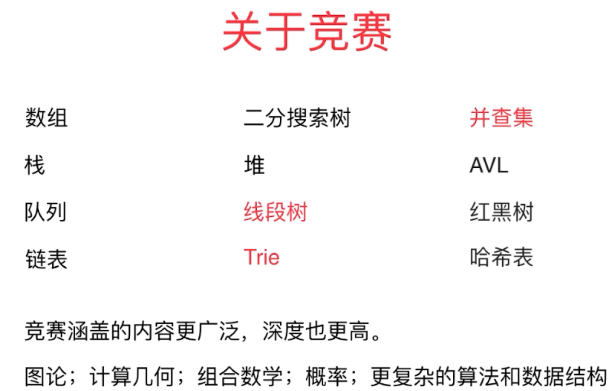




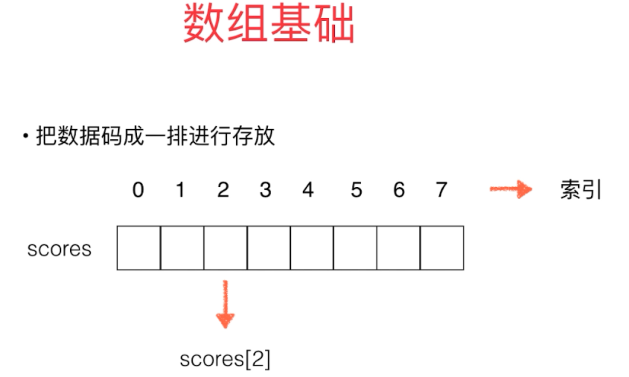










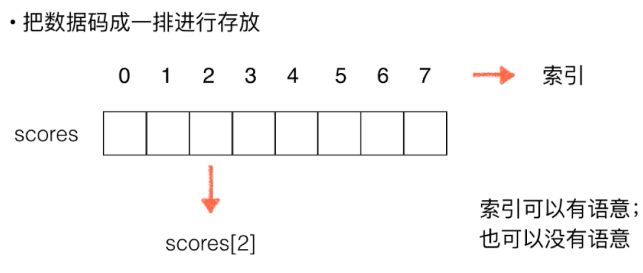


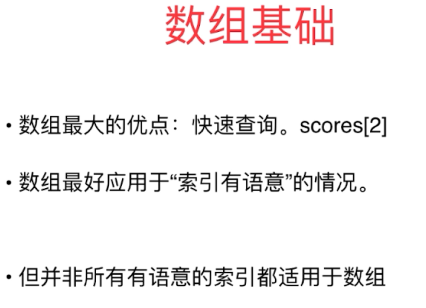
## 第2节不要小瞧数组

### 2.1数组

|  |
| --- |
| **public class** Main {   **public static void** main(String[] args) {  **int** arr[] = **new int**[10];  **for** (**int** i = 0; i < arr.**length**; i++) {  arr[i] = i;  }   **int**[] scores = **new int**[]{100,99,66};  **for** (**int** i =0; i< scores.**length** ;i++){  System.***out***.println(scores[i]);  }   scores[0] = 98;  **for** (**int** score:scores) {  System.***out***.println(score);  }  } } |

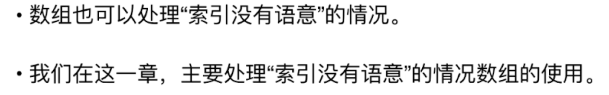
### 2.2二次封装属于我们自己的数组

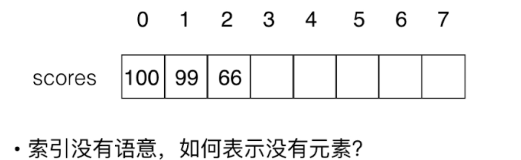


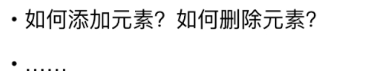


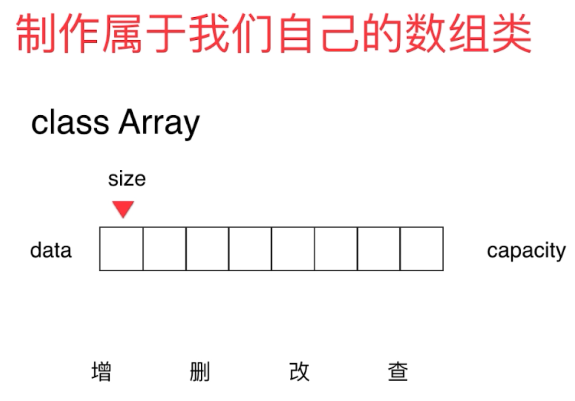


空间开辟，无须开辟这么大的空间









|  |
| --- |
| */\*\*  \* Created by BYF on 2019/1/26.  \*/* **public class** Array {  **private int**[] **data**;  **private int size**;  *// 构造函数，传入数组的容量capacity构造Array* **public** Array(**int** capacity){  **data** = **new int**[capacity];  **size** = 0;  }  *// 无参数的构造函数，默认数组容量capacity=10* **public** Array(){  **this**(10);  }  *// 获取数组中的元素个数* **public int** getSize(){  **return size**;  }  *// 获取数组的容量* **public int** getCapacity() {  **return data**.**length**;  }  *// 返回数组是否为空* **public boolean** isEmpty(){  **return size** == 0;  }  *// 向所有元素后添加一个新元素* **public void** addList(**int** e){  add(**size**, e);  }  **public void** addFirst(**int** e){  add(0,e);  }  *// 在第index个位置插入一个新元素e* **public void** add(**int** index, **int** e){  **if** (**size** == **data**.**length**){  **throw new** IllegalArgumentException(**"Add failed. Array is full."**);  }  **if** (index < 0 || index > **size**){  **throw new** IllegalArgumentException(**"Add failed. Require index >= 0 and index <= size"**);  }  **for** (**int** i = **size** - 1;i >= index ; i--){  **data**[i+1] = **data**[i];  }  **data**[index] = e;  **size** ++;  }  *// 获取index索引位置的元素* **int** get(**int** index){  **if** (index < 0 || index >= **size**){  **throw new** IllegalArgumentException(**"Get failed. Require index >= 0 and index < size"**);  }  **return data**[index];  }  *// 查找数组中是否有元素e* **public boolean** contains(**int** e){  **for** (**int** i = 0; i < **size**; i++) {  **if** (**data**[i] == e) {  **return true**;  }  }  **return false**;  }   *// 查找数组中元素e所在的索引，* **public int** find(**int** e){  **for** (**int** i = 0; i < **size**; i++) {  **if** (**data**[i] == e)  **return** i;  }  **return** -1;  }  *// 从数组中删除index位置的元素，返回删除的元素* **public int** remove(**int** index){  **if** (index < 0 || index > **size**){  **throw new** IllegalArgumentException(**"Remove failed. Require index >= 0 and index < size"**);  }  **int** ret = **data**[index];  **for** (**int** i = index + 1 ; i < **size**; i++) {  **data**[i - 1] = **data**[i];  }  **size** --;  **return** ret;  }  *// 从数组中删除第一个元素，返回删除的元素* **public int** removeFirst(){  **return** remove(0);  }  *// 从数组中删除最后一个元素，返回删除的元素* **public int** removeLast(){  **return** remove(**size**);  }  *// 从数组中删除e* **public void** removeElement(**int** e){  **int** index = find(e);  **if** (index != -1)  remove(index);  }  @Override  **public** String toString(){  StringBuilder res = **new** StringBuilder();  res.append(String.*format*(**"Array: size = %d, capacity = %d\n"**, **size**, **data**.**length**));  res.append(**'['**);  **for** (**int** i = 0; i < **size**; i++) {  res.append(**data**[i]);  **if** (i != **size** - 1){  res.append(**','**);  }  }  res.append(**']'**);  **return** res.toString();  } } |

|  |
| --- |
| **public class** Main {   **public static void** main(String[] args) {  Array arr = **new** Array(20);  **for** (**int** i = 0; i < 10; i++) {  arr.addList(i);  }  arr.add(1,100);  System.***out***.println(arr);  arr.addFirst(-1);  System.***out***.println(arr);  System.***out***.println(arr.get(2));  arr.remove(2);  System.***out***.println(arr);  arr.removeElement(4);  System.***out***.println(arr);  arr.removeFirst();  System.***out***.println(arr);  } } |

Array: size = 11, capacity = 20

[0,100,1,2,3,4,5,6,7,8,9]

Array: size = 12, capacity = 20

[-1,0,100,1,2,3,4,5,6,7,8,9]

100

Array: size = 11, capacity = 20

[-1,0,1,2,3,4,5,6,7,8,9]

Array: size = 10, capacity = 20

[-1,0,1,2,3,5,6,7,8,9]

Array: size = 9, capacity = 20

[0,1,2,3,5,6,7,8,9]

### 2.3向数组中添加元素

|  |
| --- |
| *// 向所有元素后添加一个新元素* **public void** addList(**int** e){  add(**size**, e);  }  **public void** addFirst(**int** e){  add(0,e);  }  *// 在第index个位置插入一个新元素e* **public void** add(**int** index, **int** e){  **if** (**size** == **data**.**length**){  **throw new** IllegalArgumentException(**"Add failed. Array is full."**);  }  **if** (index < 0 || index > **size**){  **throw new** IllegalArgumentException(**"Add failed. Require index >= 0 and index <= size"**);  }  **for** (**int** i = **size** - 1;i >= index ; i--){  **data**[i+1] = **data**[i];  }  **data**[index] = e;  **size** ++;  } |

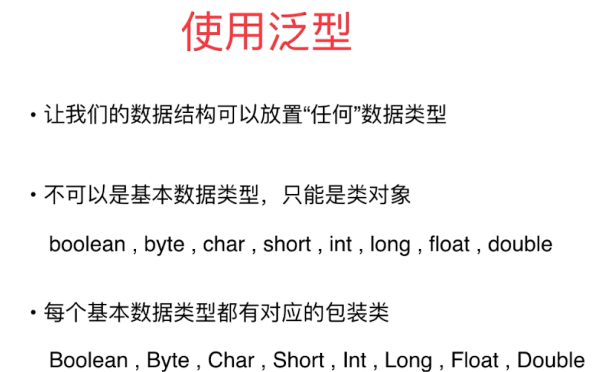
### 2.4查询和修改元素

|  |
| --- |
| *// 获取index索引位置的元素* **int** get(**int** index){  **if** (index < 0 || index >= **size**){  **throw new** IllegalArgumentException(**"Get failed. Require index >= 0 and index < size"**);  }  **return data**[index]; } |

### 2.5包含，搜索和删除元素

|  |
| --- |
| *// 查找数组中是否有元素e* **public boolean** contains(**int** e){  **for** (**int** i = 0; i < **size**; i++) {  **if** (**data**[i] == e) {  **return true**;  }  }  **return false**; }  *// 查找数组中元素e所在的索引，* **public int** find(**int** e){  **for** (**int** i = 0; i < **size**; i++) {  **if** (**data**[i] == e)  **return** i;  }  **return** -1; } *// 从数组中删除index位置的元素，返回删除的元素* **public int** remove(**int** index){  **if** (index < 0 || index > **size**){  **throw new** IllegalArgumentException(**"Remove failed. Require index >= 0 and index < size"**);  }  **int** ret = **data**[index];  **for** (**int** i = index + 1 ; i < **size**; i++) {  **data**[i - 1] = **data**[i];  }  **size** --;  **return** ret; } *// 从数组中删除第一个元素，返回删除的元素* **public int** removeFirst(){  **return** remove(0); } *// 从数组中删除最后一个元素，返回删除的元素* **public int** removeLast(){  **return** remove(**size**); } *// 从数组中删除e* **public void** removeElement(**int** e){  **int** index = find(e);  **if** (index != -1)  remove(index); } |

### 2.6使用泛型



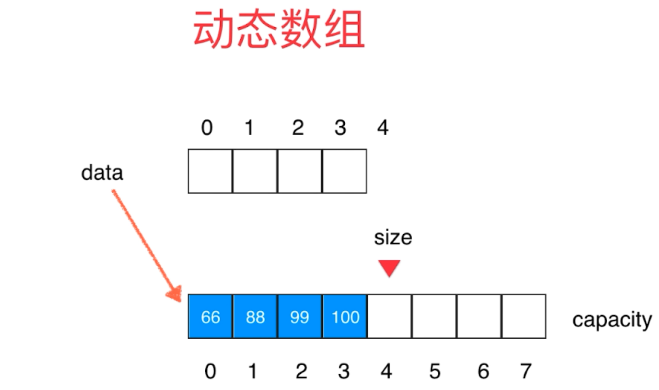
|  |
| --- |
| **public class** Array<E> {  **private** E[] **data**;  **private int size**;  *// 构造函数，传入数组的容量capacity构造Array* **public** Array(**int** capacity){  **data** = (E[]) **new** Object[capacity];  **size** = 0;  }  *// 无参数的构造函数，默认数组容量capacity=10* **public** Array(){  **this**(10);  }  *// 获取数组中的元素个数* **public int** getSize(){  **return size**;  }  *// 获取数组的容量* **public int** getCapacity() {  **return data**.**length**;  }  *// 返回数组是否为空* **public boolean** isEmpty(){  **return size** == 0;  }  *// 向所有元素后添加一个新元素* **public void** addList(E e){  add(**size**, e);  }  **public void** addFirst(E e){  add(0,e);  }  *// 在第index个位置插入一个新元素e* **public void** add(**int** index, E e){  **if** (**size** == **data**.**length**){  **throw new** IllegalArgumentException(**"Add failed. Array is full."**);  }  **if** (index < 0 || index > **size**){  **throw new** IllegalArgumentException(**"Add failed. Require index >= 0 and index <= size"**);  }  **for** (**int** i = **size** - 1;i >= index ; i--){  **data**[i+1] = **data**[i];  }  **data**[index] = e;  **size** ++;  }  *// 获取index索引位置的元素* E get(**int** index){  **if** (index < 0 || index >= **size**){  **throw new** IllegalArgumentException(**"Get failed. Require index >= 0 and index < size"**);  }  **return data**[index];  }  *// 查找数组中是否有元素e* **public boolean** contains(E e){  **for** (**int** i = 0; i < **size**; i++) {  **if** (**data**[i].equals(e)) {  **return true**;  }  }  **return false**;  }   *// 查找数组中元素e所在的索引，* **public int** find(E e){  **for** (**int** i = 0; i < **size**; i++) {  **if** (**data**[i].equals(e))  **return** i;  }  **return** -1;  }  *// 从数组中删除index位置的元素，返回删除的元素* **public** E remove(**int** index){  **if** (index < 0 || index > **size**){  **throw new** IllegalArgumentException(**"Remove failed. Require index >= 0 and index < size"**);  }  E ret = **data**[index];  **for** (**int** i = index + 1 ; i < **size**; i++) {  **data**[i - 1] = **data**[i];  }  **size** --;  **data**[**size**] = **null**; *// loitering objects != memory leak* **return** ret;  }  *// 从数组中删除第一个元素，返回删除的元素* **public** E removeFirst(){  **return** remove(0);  }  *// 从数组中删除最后一个元素，返回删除的元素* **public** E removeLast(){  **return** remove(**size**);  }  *// 从数组中删除e* **public void** removeElement(E e){  **int** index = find(e);  **if** (index != -1)  remove(index);  }  @Override  **public** String toString(){  StringBuilder res = **new** StringBuilder();  res.append(String.*format*(**"Array: size = %d, capacity = %d\n"**, **size**, **data**.**length**));  res.append(**'['**);  **for** (**int** i = 0; i < **size**; i++) {  res.append(**data**[i]);  **if** (i != **size** - 1){  res.append(**','**);  }  }  res.append(**']'**);  **return** res.toString();  } } |

|  |
| --- |
| **public class** Student {  **private** String **name**;  **private int score**;   **public** Student(String studentName, **int** studentScore){  **name** = studentName;  **score** = studentScore;  }  @Override  **public** String toString(){  **return** String.*format*(**"Student(name: %s, score: %d)"**, **name**, **score**);  }   **public static void** main(String[] args){  Array<Student> arr = **new** Array<>();  arr.addList(**new** Student(**"Alice"**, 100));  arr.addList(**new** Student(**"Bob"**, 66));  arr.addList(**new** Student(**"Charlie"**, 88));   System.***out***.println(arr);  } } |

Array: size = 3, capacity = 10

[Student(name: Alice, score: 100),Student(name: Bob, score: 66),Student(name: Charlie, score: 88)]

### 2.7动态数组

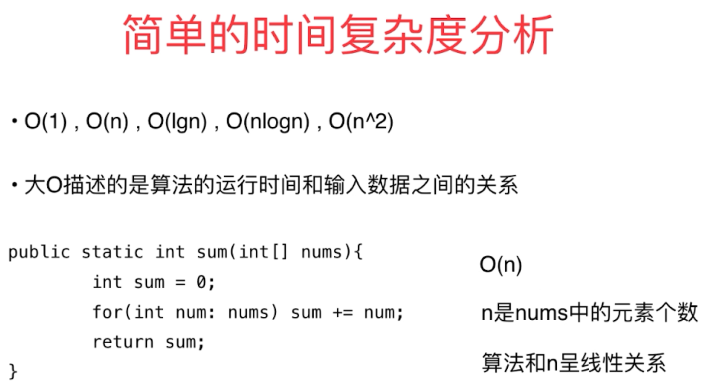


|  |
| --- |
| **public void** add(**int** index, E e){  **if** (index < 0 || index > **size**){  **throw new** IllegalArgumentException(**"Add failed. Require index >= 0 and index <= size"**);  }  **if** (**size** == **data**.**length**){  resize(2 \* **data**.**length**);  }  **for** (**int** i = **size** - 1;i >= index ; i--){  **data**[i+1] = **data**[i];  }  **data**[index] = e;  **size** ++; }  **private void** resize(**int** newCapacity){  E[] newData = (E[]) **new** Object[newCapacity];  **for** (**int** i = 0; i < **size**; i++) {  newData[i] = **data**[i];  }  **data** = newData; } |

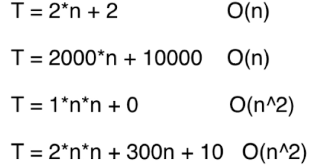
|  |
| --- |
| *// 从数组中删除index位置的元素，返回删除的元素* **public** E remove(**int** index){  **if** (index < 0 || index > **size**){  **throw new** IllegalArgumentException(**"Remove failed. Require index >= 0 and index < size"**);  }  E ret = **data**[index];  **for** (**int** i = index + 1 ; i < **size**; i++) {  **data**[i - 1] = **data**[i];  }  **size** --;  **data**[**size**] = **null**; *// loitering objects != memory leak* **if** (**size** == **data**.**length** / 2) {  resize(**data**.**length**/2);  }  **return** ret; } |

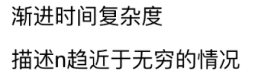
|  |
| --- |
| **public class** Main {   **public static void** main(String[] args) {  Array<Integer> arr = **new** Array<>();  **for** (**int** i = 0; i < 10; i++) {  arr.addList(i);  }  System.***out***.println(arr);  arr.add(1,100);  System.***out***.println(arr);  arr.addFirst(-1);  System.***out***.println(arr);  System.***out***.println(arr.get(2));  arr.remove(2);  System.***out***.println(arr);  arr.removeElement(4);  System.***out***.println(arr);  arr.removeFirst();  System.***out***.println(arr);  } } |
| Array: size = 10, capacity = 10  [0,1,2,3,4,5,6,7,8,9]  Array: size = 11, capacity = 20  [0,100,1,2,3,4,5,6,7,8,9]  Array: size = 12, capacity = 20  [-1,0,100,1,2,3,4,5,6,7,8,9]  100  Array: size = 11, capacity = 20  [-1,0,1,2,3,4,5,6,7,8,9]  Array: size = 10, capacity = 10  [-1,0,1,2,3,5,6,7,8,9]  Array: size = 9, capacity = 10  [0,1,2,3,5,6,7,8,9] |

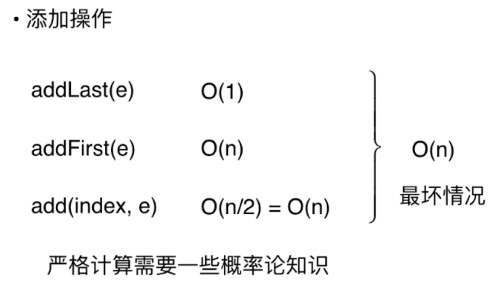
### 2.8简单的复杂度分析

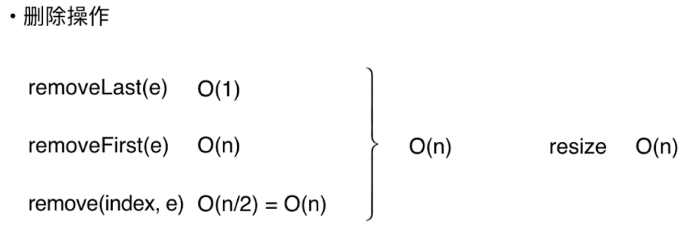


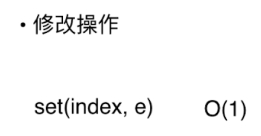




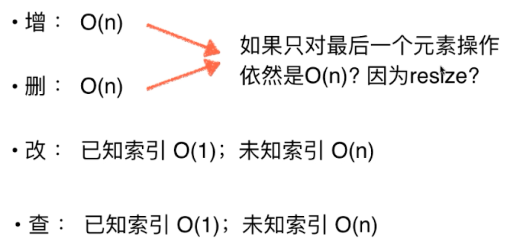




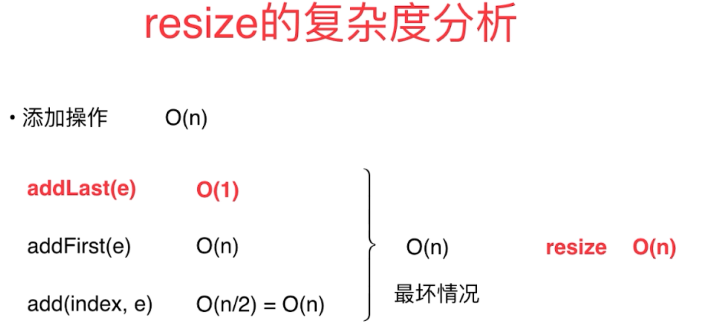


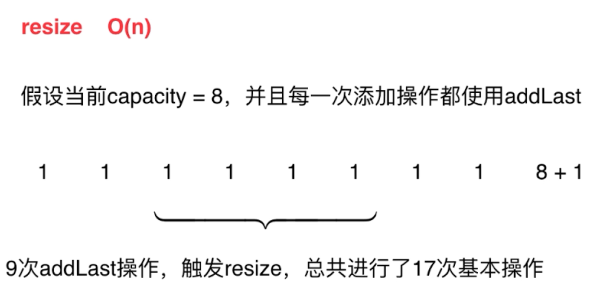


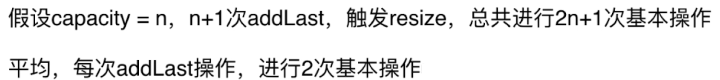


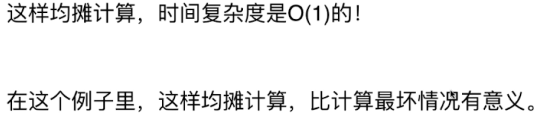


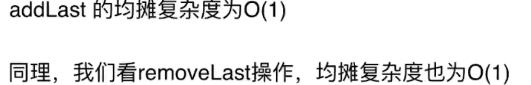
### 2.9均摊复杂度和防止复杂度的震荡

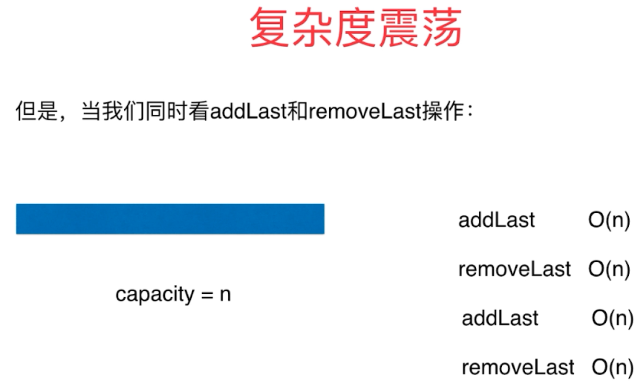


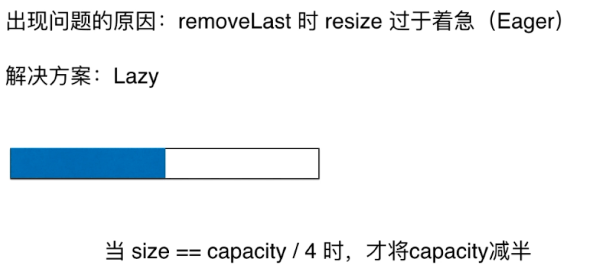








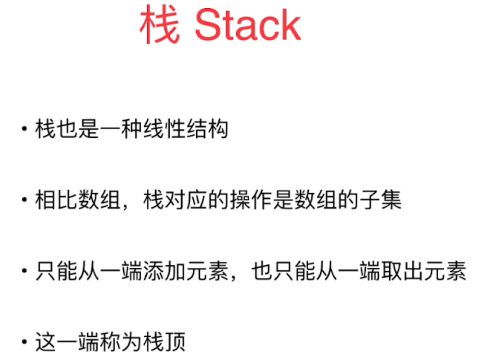


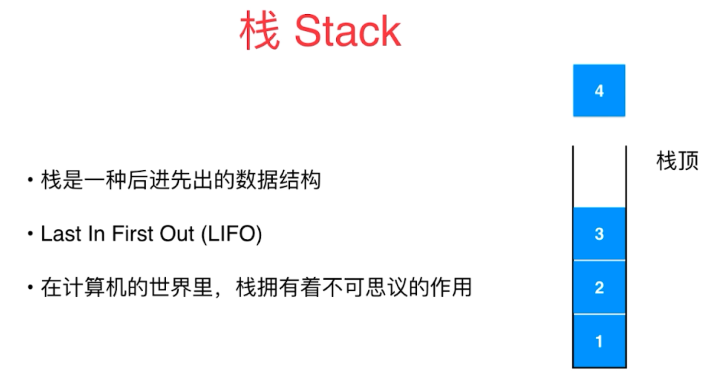


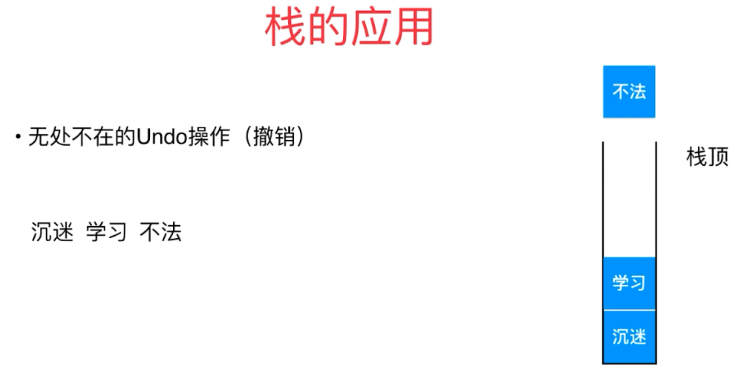
|  |
| --- |
| *// 从数组中删除index位置的元素，返回删除的元素* **public** E remove(**int** index){  **if** (index < 0 || index > **size**){  **throw new** IllegalArgumentException(**"Remove failed. Require index >= 0 and index < size"**);  }  E ret = **data**[index];  **for** (**int** i = index + 1 ; i < **size**; i++) {  **data**[i - 1] = **data**[i];  }  **size** --;  **data**[**size**] = **null**; *// loitering objects != memory leak* **if** (**size** == **data**.**length** / 4 && **data**.**length** / 2 != 0) {  resize(**data**.**length**/2);  }  **return** ret; } |

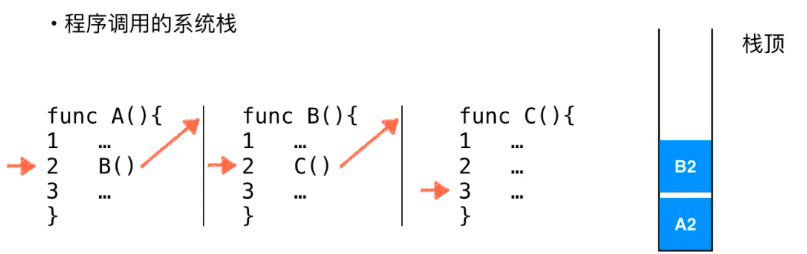
## 第3节栈和队列

### 3.1栈和栈的应用：撤销操作和系统栈

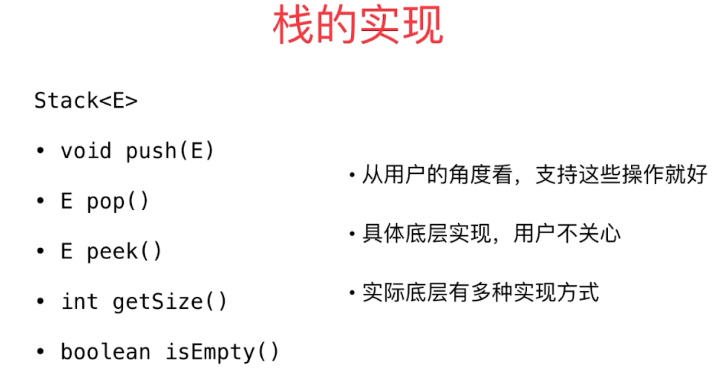








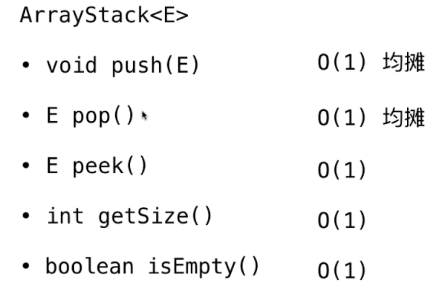
### 3.2栈的基本实现



|  |
| --- |
| **public interface** Stack<E> {  **public void** push(E e);  **public** E pop();  **public** E peek();  **public int** getSize();  **public boolean** isEmpty(); } |

|  |
| --- |
| **public class** ArrayStack<E> **implements** Stack<E> {  Array<E> **array**;   **public** ArrayStack(**int** capacity){  **array** = **new** Array<>(capacity);  }  **public** ArrayStack(){  **array** = **new** Array<E>();  }  @Override  **public void** push(E e) {  **array**.addList(e);  }   @Override  **public** E pop() {  **return array**.removeLast();  }   @Override  **public** E peek() {  **return array**.getLast();  }   @Override  **public int** getSize() {  **return array**.getSize();  }   **public int** getCapacity(){  **return array**.getCapacity();  }   @Override  **public boolean** isEmpty() {  **return array**.isEmpty();  }   @Override  **public** String toString(){  StringBuilder res = **new** StringBuilder();  res.append(**"Stack: "**);  res.append(**"["**);  **for** (**int** i = 0; i < **array**.getSize(); i++) {  res.append(**array**.get(i));  **if** (i != **array**.getSize() - 1)  res.append(**", "**);  }  res.append(**"] top"**);  **return** res.toString();   } } |

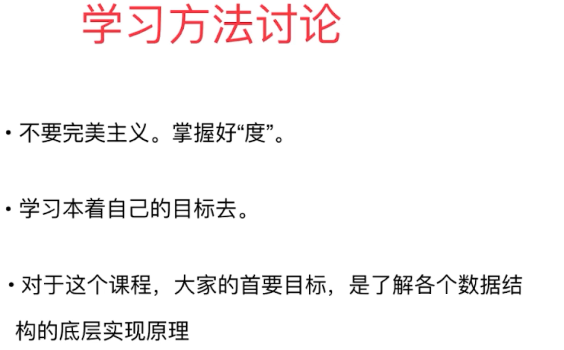
|  |
| --- |
| **public class** Main {  **public static void** main(String[] args) {  ArrayStack<Integer> stack = **new** ArrayStack<>();  **for** (**int** i = 0; i < 5; i++) {  stack.push(i);  System.***out***.println(stack);  }  stack.pop();  System.***out***.println(stack);  } } |
| Stack: [0] top  Stack: [0, 1] top  Stack: [0, 1, 2] top  Stack: [0, 1, 2, 3] top  Stack: [0, 1, 2, 3, 4] top  Stack: [0, 1, 2, 3] top |



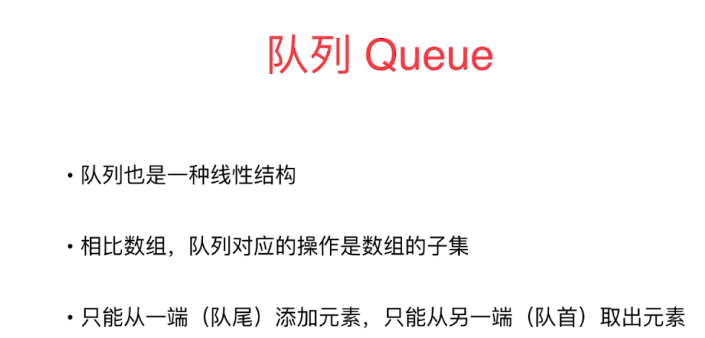
### 3.3栈的另一个应用：括号匹配

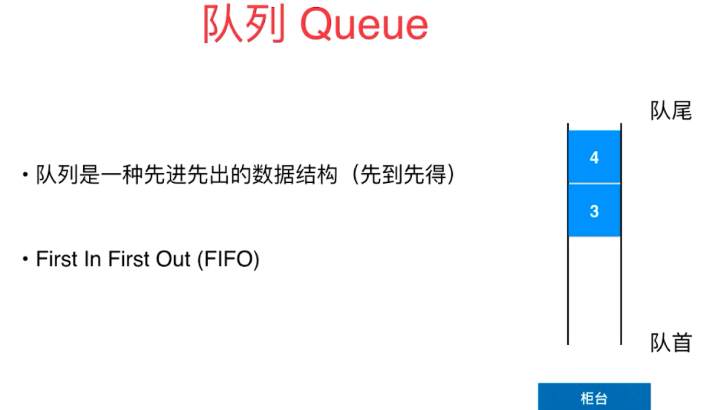
|  |
| --- |
| **import** java.util.Stack;  **class** Solution {  **public boolean** isValid(String s) {  Stack<Character> stack = **new** Stack<>();  **for** (**int** i = 0; i < s.length(); i++) {  Character c = s.charAt(i);  **if** (c == **'{'** || c == **'['** || c == **'('**)  stack.push(c);  **else** {  **if** (stack.isEmpty())  **return false**;  Character topChar = stack.pop();  **if** (**'{'** == topChar && **'}'** != c)  **return false**;  **if** (**'['** == topChar && **']'** != c)  **return false**;  **if** (**'('** == topChar && **')'** != c)  **return false**;  }  }  **return** stack.isEmpty();  }   **public static void** main(String[] args) {  System.***out***.println((**new** Solution()).isValid(**"{}"**));  System.***out***.println((**new** Solution()).isValid(**"{()[]}"**));  } } |

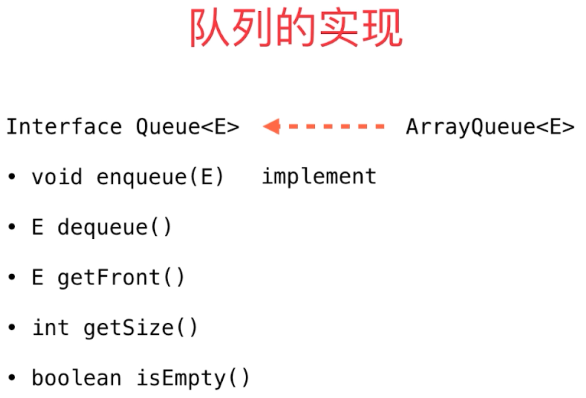
### 3.4关于Leetcode的更多说明



### 3.5数组队列



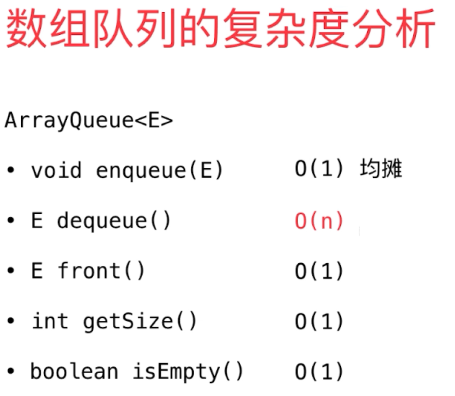




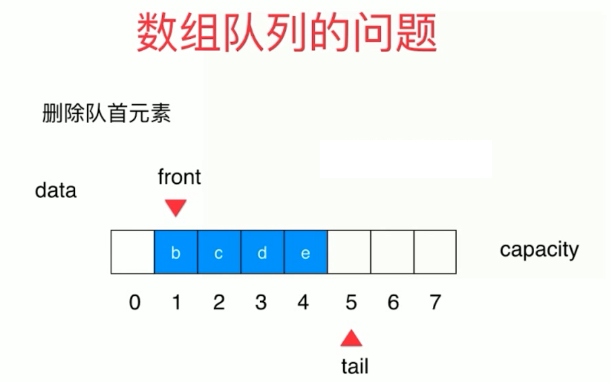
|  |
| --- |
| */\*\*  \* Created by BYF on 2019/1/26.  \*/* **public class** Array<E> {  **private** E[] **data**;  **private int size**;  *// 构造函数，传入数组的容量capacity构造Array* **public** Array(**int** capacity){  **data** = (E[]) **new** Object[capacity];  **size** = 0;  }  *// 无参数的构造函数，默认数组容量capacity=10* **public** Array(){  **this**(10);  }  *// 获取数组中的元素个数* **public int** getSize(){  **return size**;  }  *// 获取数组的容量* **public int** getCapacity() {  **return data**.**length**;  }  *// 返回数组是否为空* **public boolean** isEmpty(){  **return size** == 0;  }  *// 向所有元素后添加一个新元素* **public void** addLast(E e){  add(**size**, e);  }  **public void** addFirst(E e){  add(0,e);  }  *// 在第index个位置插入一个新元素e* **public void** add(**int** index, E e){  **if** (index < 0 || index > **size**){  **throw new** IllegalArgumentException(**"Add failed. Require index >= 0 and index <= size"**);  }  **if** (**size** == **data**.**length**){  resize(2 \* **data**.**length**);  }  **for** (**int** i = **size** - 1;i >= index ; i--){  **data**[i+1] = **data**[i];  }  **data**[index] = e;  **size** ++;  }  *// 获取index索引位置的元素* **public** E get(**int** index){  **if** (index < 0 || index >= **size**){  **throw new** IllegalArgumentException(**"Get failed. Require index >= 0 and index < size"**);  }  **return data**[index];  }  **public** E getFirst(){  **return** get(0);  }  **public** E getLast(){  **return** get(**size** - 1);  }  *// 查找数组中是否有元素e* **public boolean** contains(E e){  **for** (**int** i = 0; i < **size**; i++) {  **if** (**data**[i].equals(e)) {  **return true**;  }  }  **return false**;  }   *// 查找数组中元素e所在的索引，* **public int** find(E e){  **for** (**int** i = 0; i < **size**; i++) {  **if** (**data**[i].equals(e))  **return** i;  }  **return** -1;  }  *// 从数组中删除index位置的元素，返回删除的元素* **public** E remove(**int** index){  **if** (index < 0 || index > **size**){  **throw new** IllegalArgumentException(**"Remove failed. Require index >= 0 and index < size"**);  }  E ret = **data**[index];  **for** (**int** i = index + 1 ; i < **size**; i++) {  **data**[i - 1] = **data**[i];  }  **size** --;  **data**[**size**] = **null**; *// loitering objects != memory leak* **if** (**size** == **data**.**length** / 4 && **data**.**length** / 2 != 0) {  resize(**data**.**length**/2);  }  **return** ret;  }  *// 从数组中删除第一个元素，返回删除的元素* **public** E removeFirst(){  **return** remove(0);  }  *// 从数组中删除最后一个元素，返回删除的元素* **public** E removeLast(){  **return** remove(**size**);  }  *// 从数组中删除e* **public void** removeElement(E e){  **int** index = find(e);  **if** (index != -1)  remove(index);  }  @Override  **public** String toString(){  StringBuilder res = **new** StringBuilder();  res.append(String.*format*(**"Array: size = %d, capacity = %d\n"**, **size**, **data**.**length**));  res.append(**'['**);  **for** (**int** i = 0; i < **size**; i++) {  res.append(**data**[i]);  **if** (i != **size** - 1){  res.append(**','**);  }  }  res.append(**']'**);  **return** res.toString();  }   **private void** resize(**int** newCapacity){  E[] newData = (E[]) **new** Object[newCapacity];  **for** (**int** i = 0; i < **size**; i++) {  newData[i] = **data**[i];  }  **data** = newData;  } } |

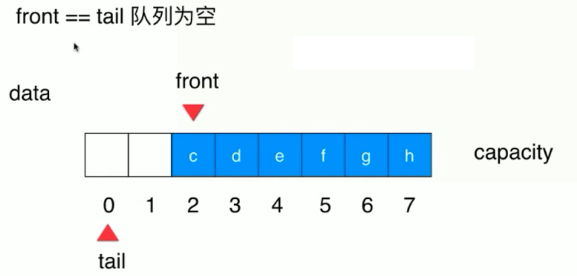
|  |
| --- |
| **public interface** Queue<E> {  **int** getSize();  **boolean** isEmpty();  **void** enqueue(E e);  E dequeue();  E getFront(); } |

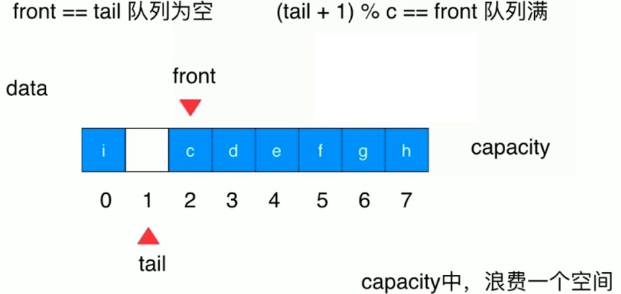
|  |
| --- |
| */\*\*  \* Created by BYF on 2019/1/27.  \*/* **public class** ArrayQueue<E> **implements** Queue<E> {  **private** Array<E> **array**;  **public** ArrayQueue(**int** capacity){  **array** = **new** Array<E>(capacity);  }  **public** ArrayQueue(){  **array** = **new** Array<E>();  }   @Override  **public int** getSize() {  **return array**.getSize();  }   @Override  **public boolean** isEmpty() {  **return array**.isEmpty();  }   **public int** getCapacity(){  **return array**.getCapacity();  }   @Override  **public void** enqueue(E e) {  **array**.addLast(e);  }   @Override  **public** E dequeue() {  **return array**.removeFirst();  }   @Override  **public** E getFront() {  **return array**.getFirst();  }  @Override  **public** String toString(){  StringBuilder res = **new** StringBuilder();  res.append(String.*format*(**"Queue: size = %d, capacity = %d "**, **array**.getSize(), **array**.getCapacity()));  res.append(**"front ["**);  **for** (**int** i = 0; i < **array**.getSize(); i++) {  res.append(**array**.get(i));  **if** (i != **array**.getSize() - 1){  res.append(**','**);  }  }  res.append(**"] tail"**);  **return** res.toString();  }   **public static void** main(String[] args) {  ArrayQueue<Integer> queue = **new** ArrayQueue<>();  **for** (**int** i = 0; i < 10; i++) {  queue.enqueue(i);  System.***out***.println(queue);  **if** (i % 3 == 2){  queue.dequeue();  System.***out***.println(queue);  }  }  } } |
| Queue: size = 1, capacity = 10 front [0] tail  Queue: size = 2, capacity = 10 front [0,1] tail  Queue: size = 3, capacity = 10 front [0,1,2] tail  Queue: size = 2, capacity = 5 front [1,2] tail  Queue: size = 3, capacity = 5 front [1,2,3] tail  Queue: size = 4, capacity = 5 front [1,2,3,4] tail  Queue: size = 5, capacity = 5 front [1,2,3,4,5] tail  Queue: size = 4, capacity = 5 front [2,3,4,5] tail  Queue: size = 5, capacity = 5 front [2,3,4,5,6] tail  Queue: size = 6, capacity = 10 front [2,3,4,5,6,7] tail  Queue: size = 7, capacity = 10 front [2,3,4,5,6,7,8] tail  Queue: size = 6, capacity = 10 front [3,4,5,6,7,8] tail  Queue: size = 7, capacity = 10 front [3,4,5,6,7,8,9] tail |



### 3.6循环队列

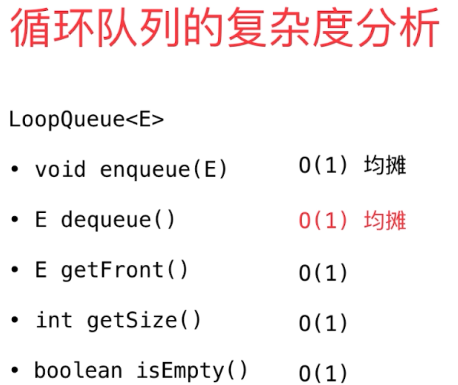






|  |
| --- |
| **public class** LoopQueue<E> **implements** Queue<E> {  **private** E[] data;  **private int front**,**tail**;  **private int size**;   **public** LoopQueue(**int** capacity){  data = (E[]) **new** Object[capacity + 1];  **front** = 0;  **tail** = 0;  **size** = 0;  }  **public** LoopQueue(){  **this**(10);  }  **public int** getCapacity(){  **return** data.**length** - 1;  }  @Override  **public boolean** isEmpty(){  **return front** == **tail**;  }  @Override  **public int** getSize(){  **return size**;  }   @Override  **public void** enqueue(E e){  **if** ((**tail** + 1) % data.**length** == **front**)  resize(getCapacity() \* 2);  data[**tail**] = e;  **tail** = (**tail** + 1) % data.**length**;  **size** ++;  }   @Override  **public** E dequeue(){  **if** (isEmpty())  **throw new** IllegalArgumentException(**"Cannot dequeue from an empty queue."**);  E ret = data[**front**];  data[**front**] = **null**;  **front** = (**front** + 1) % data.**length**;  **size** --;  **if** (**size** <= getCapacity() / 4 && getCapacity() / 2 != 0)  resize(getCapacity() / 2);  **return** ret;  }  @Override  **public** E getFront(){  **if** (isEmpty())  **throw new** IllegalArgumentException(**"Queue is empty"**);  **return** data[**front**];  }  @Override  **public** String toString(){  StringBuilder res = **new** StringBuilder();  res.append(String.*format*(**"Queue: size = %d, capacity = %d\n"**, **size**, getCapacity()));  res.append(**"front ["**);  **for** (**int** i = **front**; i != **tail**; i = (i + 1) % data.**length**) {  res.append(data[i]);  **if** ((i + 1) % data.**length** != **tail**){  res.append(**','**);  }  }  res.append(**"] tail"**);  **return** res.toString();  }   **private void** resize(**int** newCapacity){  E[] newData = (E[]) **new** Object[newCapacity + 1];  **for** (**int** i = 0; i < **size**; i++) {  newData[i] = data[(i + **front**) % data.**length**];  }  data = newData;  **front** = 0;  **tail** = **size**;  }  **public static void** main(String[] args) {  LoopQueue<Integer> queue = **new** LoopQueue<>();  **for** (**int** i = 0; i < 10; i++) {  queue.enqueue(i);  System.***out***.println(queue);  **if** (i % 3 == 2){  queue.dequeue();  System.***out***.println(queue);  }  }  } } |
| Queue: size = 1, capacity = 10  front [0] tail  Queue: size = 2, capacity = 10  front [0,1] tail  Queue: size = 3, capacity = 10  front [0,1,2] tail  Queue: size = 2, capacity = 5  front [1,2] tail  Queue: size = 3, capacity = 5  front [1,2,3] tail  Queue: size = 4, capacity = 5  front [1,2,3,4] tail  Queue: size = 5, capacity = 5  front [1,2,3,4,5] tail  Queue: size = 4, capacity = 5  front [2,3,4,5] tail  Queue: size = 5, capacity = 5  front [2,3,4,5,6] tail  Queue: size = 6, capacity = 10  front [2,3,4,5,6,7] tail  Queue: size = 7, capacity = 10  front [2,3,4,5,6,7,8] tail  Queue: size = 6, capacity = 10  front [3,4,5,6,7,8] tail  Queue: size = 7, capacity = 10  front [3,4,5,6,7,8,9] tail |

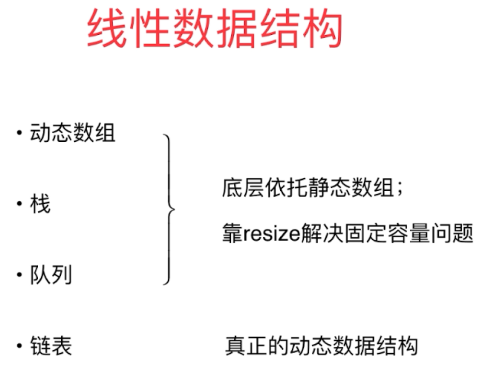
### 3.7数组队列和循环队列的比较

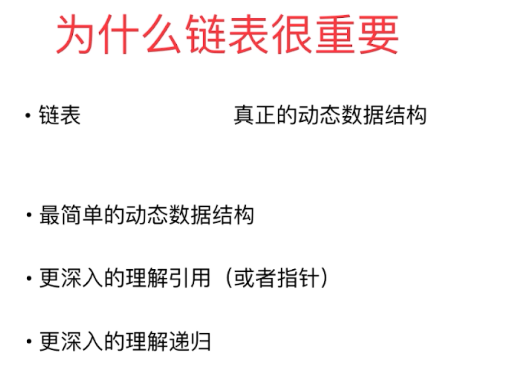


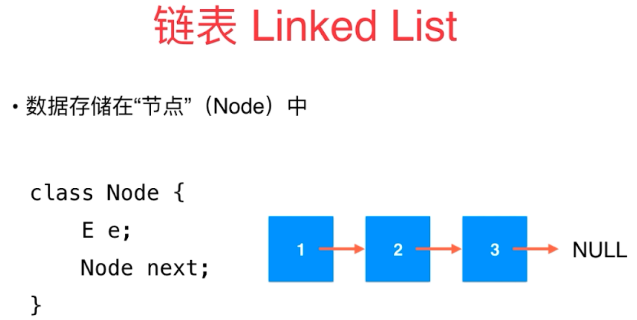
|  |
| --- |
| **import** java.util.Random;  **public class** Main {   **private static double** testQueue(Queue<Integer> q, **int** opCount){  **long** startTime = System.*nanoTime*();  Random random = **new** Random();  **for** (**int** i = 0; i < opCount; i++) {  q.enqueue(random.nextInt(Integer.***MAX\_VALUE***));  }  **for** (**int** i = 0; i < opCount; i++) {  q.dequeue();  }   **long** endTime = System.*nanoTime*();  **return** (endTime - startTime)/1000000000.0;  }   **public static void** main(String[] args) {  **int** opCount = 100000;   ArrayQueue<Integer> arrayQueue = **new** ArrayQueue<>();  **double** time1 = *testQueue*(arrayQueue, opCount);  System.***out***.println(**"ArrayQueue, time: "** + time1 + **"s"**);   LoopQueue<Integer> loopQueue = **new** LoopQueue<>();  **double** time2 = *testQueue*(loopQueue, opCount);  System.***out***.println(**"LoopQueue, time: "** + time2 + **"s"**);  } } |
| ArrayQueue, time: 4.427975688s  LoopQueue, time: 0.013530017s |

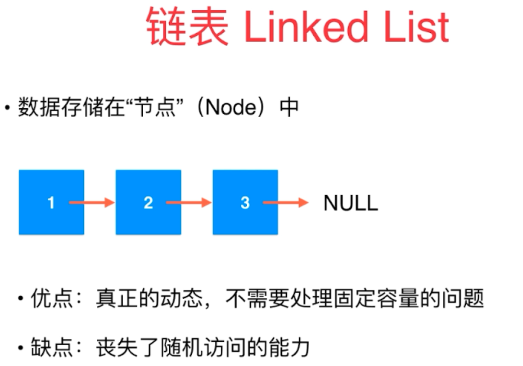
## 第4节最基础的动态数据结构：链表

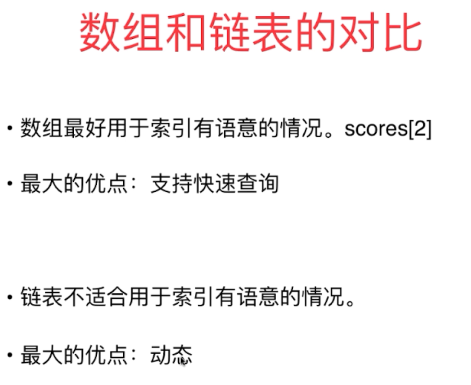
### 4.1什么是链表





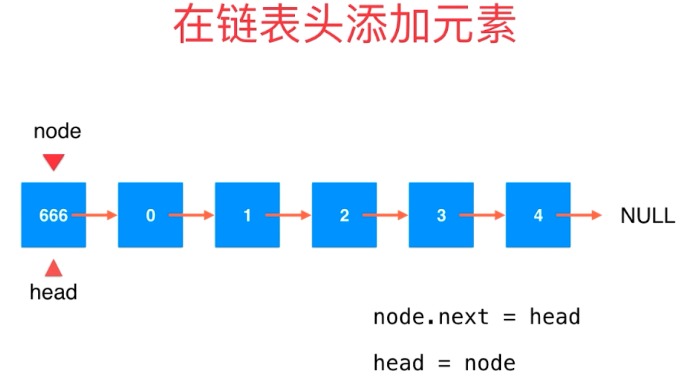


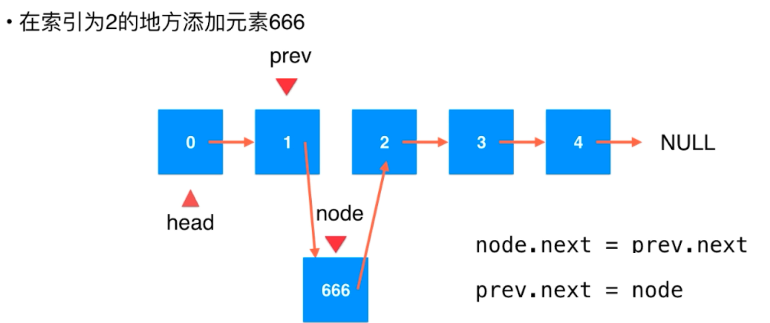


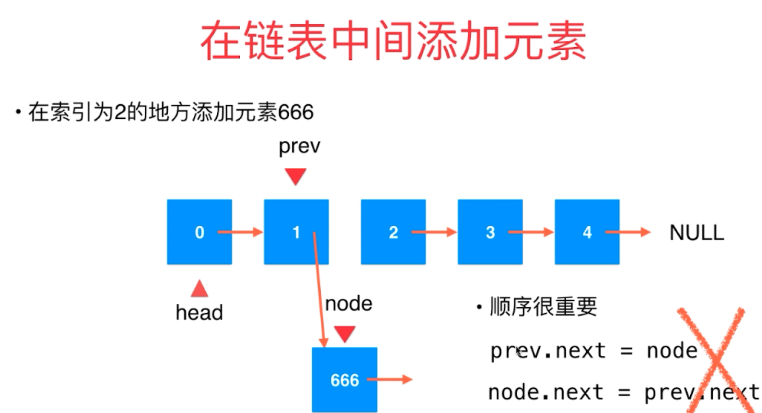


|  |
| --- |
| **public class** LinkedList<E> {  **private class** Node{  **public** E **e**;  **public** Node **next**;   **public** Node(E e, Node next){  **this**.**e** = e;  **this**.**next** = next;  }  **public** Node(E e){  **this**(e, **null**);  }  **public** Node(){  **this**(**null**,**null**);  }  @Override  **public** String toString(){  **return e**.toString();  }  } } |

### 4.2在链表中添加元素







|  |
| --- |
| **public class** LinkedList<E> {  **private class** Node{  **public** E **e**;  **public** Node **next**;   **public** Node(E e, Node next){  **this**.**e** = e;  **this**.**next** = next;  }  **public** Node(E e){  **this**(e, **null**);  }  **public** Node(){  **this**(**null**,**null**);  }  @Override  **public** String toString(){  **return e**.toString();  }  }  **private** Node **head**;  **int size**;   **public** LinkedList(){  **head** = **null**;  **size** = 0;  }  *// 获取链表中的元素个数* **public int** getSize(){  **return size**;  }  *// 返回链表是否为空* **public boolean** isEmpty(){  **return size** == 0;  }  *// 在链表头添加新的元素e* **public void** addFirst(E e){  **head** = **new** Node(e, **head**);  **size** ++;  }  *// 在链表中间添加新的元素e* **public void** add(**int** index, E e){  **if** (index < 0 || index > **size**) {  **throw new** IllegalArgumentException(**"Add failed. Illegal index"**);  }  **if** (index == 0){  addFirst(e);  }  **else** {  Node prev = **head**;  **for** (**int** i = 0; i < index - 1; i++) {  prev = prev.**next**;  }  */\*Node node = new Node(e);  node.next = prev.next;  prev.next = node;\*/* prev.**next** = **new** Node(e, prev.**next**);  **size** ++;  }  }  *// 在链表尾部添加新的元素e* **public void** addLast(E e) {  add(**size**, e);  } } |

### 4.3使用链表的虚拟头结点

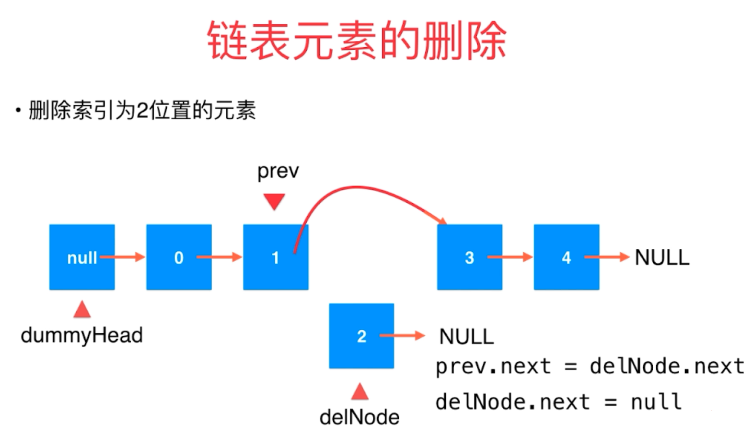
|  |
| --- |
| **public class** LinkedList<E> {  **private class** Node{  **public** E **e**;  **public** Node **next**;   **public** Node(E e, Node next){  **this**.**e** = e;  **this**.**next** = next;  }  **public** Node(E e){  **this**(e, **null**);  }  **public** Node(){  **this**(**null**,**null**);  }  @Override  **public** String toString(){  **return e**.toString();  }  }  **private** Node **dummyHead**;  **int size**;   **public** LinkedList(){  **dummyHead** = **new** Node(**null**,**null**);  **size** = 0;  }  *// 获取链表中的元素个数* **public int** getSize(){  **return size**;  }  *// 返回链表是否为空* **public boolean** isEmpty(){  **return size** == 0;  }  *// 在链表中间添加新的元素e* **public void** add(**int** index, E e){  **if** (index < 0 || index > **size**) {  **throw new** IllegalArgumentException(**"Add failed. Illegal index"**);  }  Node prev = **dummyHead**;  **for** (**int** i = 0; i < index; i++) {  prev = prev.**next**;  }  */\*Node node = new Node(e);  node.next = prev.next;  prev.next = node;\*/* prev.**next** = **new** Node(e, prev.**next**);  **size** ++;  }  *// 在链表头添加新的元素e* **public void** addFirst(E e){  add(0,e);  }  *// 在链表尾部添加新的元素e* **public void** addLast(E e) {  add(**size**, e);  } } |

### 4.4链表的遍历，查询和修改

|  |
| --- |
| **public class** LinkedList<E> {  **private class** Node{  **public** E **e**;  **public** Node **next**;   **public** Node(E e, Node next){  **this**.**e** = e;  **this**.**next** = next;  }  **public** Node(E e){  **this**(e, **null**);  }  **public** Node(){  **this**(**null**,**null**);  }  @Override  **public** String toString(){  **return e**.toString();  }  }  **private** Node **dummyHead**;  **int size**;   **public** LinkedList(){  **dummyHead** = **new** Node(**null**,**null**);  **size** = 0;  }  *// 获取链表中的元素个数* **public int** getSize(){  **return size**;  }  *// 返回链表是否为空* **public boolean** isEmpty(){  **return size** == 0;  }  *// 在链表中间添加新的元素e* **public void** add(**int** index, E e){  **if** (index < 0 || index > **size**) {  **throw new** IllegalArgumentException(**"Add failed. Illegal index"**);  }  Node prev = **dummyHead**;  **for** (**int** i = 0; i < index; i++) {  prev = prev.**next**;  }  */\*Node node = new Node(e);  node.next = prev.next;  prev.next = node;\*/* prev.**next** = **new** Node(e, prev.**next**);  **size** ++;  }  *// 在链表头添加新的元素e* **public void** addFirst(E e){  add(0,e);  }  *// 在链表尾部添加新的元素e* **public void** addLast(E e) {  add(**size**, e);  }  *// 获得链表的第index(0-based)个位置的元素  // 在链表中不是一个常用操作，练习用* **public** E get(**int** index) {  **if** (index < 0 || index > **size**) {  **throw new** IllegalArgumentException(**"Get failed. Illegal index"**);  }  Node cur = **dummyHead**.**next**;  **for** (**int** i = 0; i < index; i++) {  cur = cur.**next**;  }  **return** cur.**e**;  }  *// 获得链表第一个元素* **public** E getFirst() {  **return** get(0);  }  *// 获得链表最后一个元素* **public** E getLast() {  **return** get(**size** - 1);  }  *// 修改链表的第index(0-based)个位置的元素  // 在链表中不是一个常用操作，练习用* **public void** set(**int** index, E e) {  **if** (index < 0 || index > **size**) {  **throw new** IllegalArgumentException(**"Set failed. Illegal index"**);  }  Node cur = **dummyHead**.**next**;  **for** (**int** i = 0; i < index; i++) {  cur = cur.**next**;  }  cur.**e** = e;  }  *// 在链表中查找是否包含元素e* **public boolean** contains(E e){  Node cur = **dummyHead**.**next**;  **while** (cur != **null**){  **if** (cur.**e**.equals(e)) {  **return true**;  }  cur = cur.**next**;  }  **return false**;  }  @Override  **public** String toString() {  StringBuilder res = **new** StringBuilder();  Node cur = **dummyHead**.**next**;  **while** (cur != **null**){  res.append(cur.**e** + **"->"**);  cur = cur.**next**;  }  res.append(**"NULL"**);  **return** res.toString();  }   } |

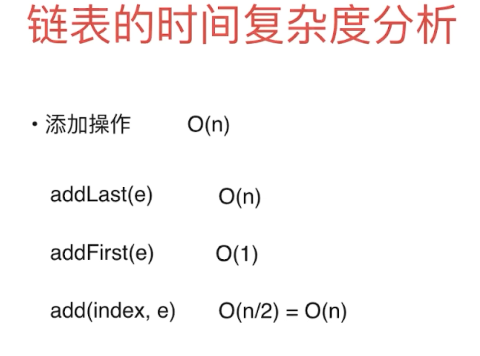
|  |
| --- |
| **public class** Main {   **public static void** main(String[] args) {  LinkedList<Integer> linkedList = **new** LinkedList<>();  **for** (**int** i = 0; i < 5; i++) {  linkedList.addFirst(i);  System.***out***.println(linkedList);  }  linkedList.add(2,666);  System.***out***.println(linkedList);  } } |
| 0->NULL  1->0->NULL  2->1->0->NULL  3->2->1->0->NULL  4->3->2->1->0->NULL  4->3->666->2->1->0->NULL |

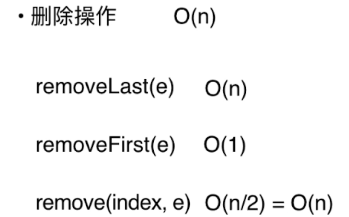
### 4.5从链表中删除元素

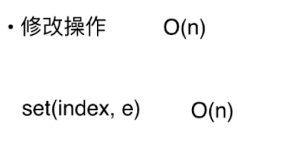


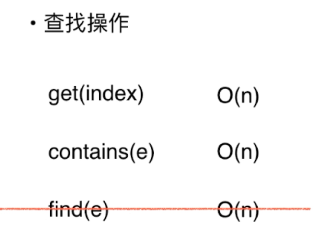
|  |
| --- |
| *// 删除链表的第index(0-based)个位置的元素 // 在链表中不是一个常用操作，练习用* **public** E remove(**int** index){  **if** (index < 0 || index > **size**) {  **throw new** IllegalArgumentException(**"Remove failed. Illegal index"**);  }  Node prev = **dummyHead**;  **for** (**int** i = 0; i < index; i++) {  prev = prev.**next**;  }  Node delNode = prev.**next**;  prev.**next** = delNode.**next**;  **size** --;  delNode.**next** = **null**;  **return** delNode.**e**; } *// 删除链表第一个元素* **public** E removeFirst(){  **return** remove(0); } *// 删除链表最后一个元素* **public** E removeLast() {  **return** remove(**size** - 1); } |

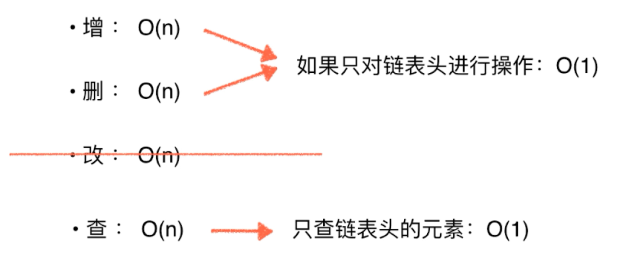
|  |
| --- |
| **public class** Main {   **public static void** main(String[] args) {  LinkedList<Integer> linkedList = **new** LinkedList<>();  **for** (**int** i = 0; i < 5; i++) {  linkedList.addFirst(i);  System.***out***.println(linkedList);  }  linkedList.add(2,666);  System.***out***.println(linkedList);  linkedList.remove(2);  System.***out***.println(linkedList);  linkedList.removeFirst();  System.***out***.println(linkedList);  linkedList.removeLast();  System.***out***.println(linkedList);  } } |
| 0->NULL  1->0->NULL  2->1->0->NULL  3->2->1->0->NULL  4->3->2->1->0->NULL  4->3->666->2->1->0->NULL  4->3->2->1->0->NULL  3->2->1->0->NULL  3->2->1->NULL |











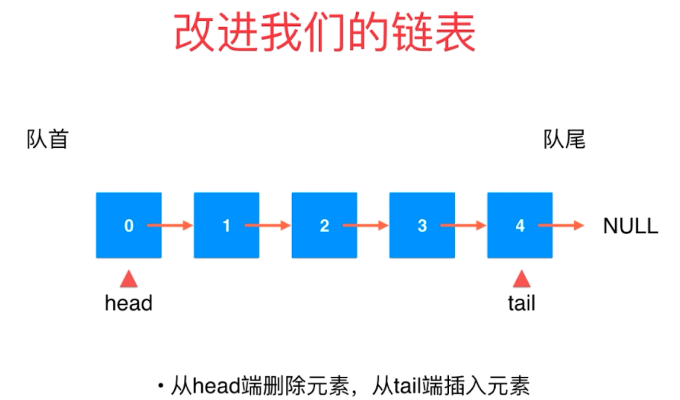
### 4.6使用链表实现栈

|  |
| --- |
| */\*\*  \* Created by BYF on 2019/1/30.  \*/* **public class** LinkedListStack<E> **implements** Stack<E> {  **private** LinkedList<E> **list**;  **private int size**;  **public** LinkedListStack(){  **list** = **new** LinkedList<E>();  **size** = 0;  }   @Override  **public void** push(E e) {  **list**.addFirst(e);  }   @Override  **public** E pop() {  **return list**.removeFirst();  }   @Override  **public** E peek() {  **return list**.getFirst();  }   @Override  **public int** getSize() {  **return size**;  }   @Override  **public boolean** isEmpty() {  **return size** == 0;  }  @Override  **public** String toString() {  StringBuilder res = **new** StringBuilder();  res.append(**"Stack: top "**);  res.append(**list**);  **return** res.toString();  }   **public static void** main(String[] args) {  LinkedListStack<Integer> stack = **new** LinkedListStack<>();  **for** (**int** i = 0; i < 5; i++) {  stack.push(i);  System.***out***.println(stack);  }  stack.pop();  System.***out***.println(stack);  } } |
| Stack: top 0->NULL  Stack: top 1->0->NULL  Stack: top 2->1->0->NULL  Stack: top 3->2->1->0->NULL  Stack: top 4->3->2->1->0->NULL  Stack: top 3->2->1->0->NULL |

**ArrayStack与LinkedListStack对比**

|  |
| --- |
| **import** java.util.Random;  **public class** Main {  **public static double** testStack(Stack<Integer> stack, **int** opCount){  **long** startTime = System.*nanoTime*();  Random random = **new** Random();  **for** (**int** i = 0; i < opCount; i++) {  stack.push(random.nextInt(Integer.***MAX\_VALUE***));  }  **for** (**int** i = 0; i < opCount; i++) {  stack.pop();  }  **long** endTime = System.*nanoTime*();  **return** (endTime - startTime) / 1000000000.0;  }  **public static void** main(String[] args) {  */\*ArrayStack<Integer> stack = new ArrayStack<>();  for (int i = 0; i < 5; i++) {  stack.push(i);  System.out.println(stack);  }  stack.pop();  System.out.println(stack);\*/* **int** opCount = 100000;  ArrayStack<Integer> arrayStack = **new** ArrayStack<>();  **double** time1 = *testStack*(arrayStack, opCount);  System.***out***.println(**"ArrayStack cost: "** + time1 + **"s"**);   LinkedListStack<Integer> linkedListStack = **new** LinkedListStack<>();  **double** time2 = *testStack*(linkedListStack, opCount);  System.***out***.println(**"LinkedListStack cost: "** + time2 + **"s"**);  } } |
| **ArrayStack cost: 0.020467886s**  **LinkedListStack cost: 0.018915238s** |
| opCount : 10,000,000  **ArrayStack cost: 3.107667405s**  **LinkedListStack cost: 6.350335576s** |

### 4.7使用链表实现队列：带有尾指针的链表





链表是单向的，tail无法删除tail之前的元素，所以删除是需要从header往后查找size-1个元素才能进行删除，但是从tail入队是不需要进行遍历的操作；

|  |
| --- |
| */\*\*  \* Created by BYF on 2019/1/30.  \*/* **public class** LinkedListQueue<E> **implements** Queue<E> {  **public class** Node{  **public** E **e**;  **public** Node **next**;  **public** Node(E e, Node next){  **this**.**e** = e;  **this**.**next** = next;  }  **public** Node(E e){  **this**(e,**null**);  }  **public** Node(){  **this**(**null**,**null**);  }  @Override  **public** String toString() {  **return e**.toString();  }  }  **private** Node **head**,**tail**;  **private int size**;  **public** LinkedListQueue(){  **head** = **null**;  **tail** = **null**;  **size** = 0;  }  @Override  **public int** getSize() {  **return size**;  }   @Override  **public boolean** isEmpty() {  **return size** == 0;  }   @Override  **public void** enqueue(E e) {  **if** (**tail** == **null**){  **tail** = **new** Node(e);  **head** = **tail**;  } **else** {  **tail**.**next** = **new** Node(e);  **tail** = **tail**.**next**;  }  **size** ++;  }   @Override  **public** E dequeue() {  **if** (isEmpty()){  **throw new** IllegalArgumentException(**"Cannot dequeue from an empty queue."**);  }  Node retNode = **head**;  **head** = **head**.**next**;  retNode.**next** = **null**;  **if** (**head** == **null**){  **tail** = **null**;  }  **size** --;  **return** retNode.**e**;  }   @Override  **public** E getFront() {  **if** (isEmpty())  **throw new** IllegalArgumentException(**"Queue is empty."**);  **return head**.**e**;  }  @Override  **public** String toString(){  StringBuilder res = **new** StringBuilder();  res.append(**"LinkedListQueue: head "**);  Node cur = **head**;  **while** (cur != **null**){  res.append(cur.**e** + **"->"**);  cur = cur.**next**;  }  res.append(**"NULL tail"**);  **return** res.toString();   }   **public static void** main(String[] args) {  LinkedListQueue<Integer> queue = **new** LinkedListQueue<>();  **for** (**int** i = 0; i < 10; i++) {  queue.enqueue(i);  System.***out***.println(queue);  **if** (i % 3 == 2){  queue.dequeue();  System.***out***.println(queue);  }  }  }  } |
| LinkedListQueue: head 0->NULL tail  LinkedListQueue: head 0->1->NULL tail  LinkedListQueue: head 0->1->2->NULL tail  LinkedListQueue: head 1->2->NULL tail  LinkedListQueue: head 1->2->3->NULL tail  LinkedListQueue: head 1->2->3->4->NULL tail  LinkedListQueue: head 1->2->3->4->5->NULL tail  LinkedListQueue: head 2->3->4->5->NULL tail  LinkedListQueue: head 2->3->4->5->6->NULL tail  LinkedListQueue: head 2->3->4->5->6->7->NULL tail  LinkedListQueue: head 2->3->4->5->6->7->8->NULL tail  LinkedListQueue: head 3->4->5->6->7->8->NULL tail  LinkedListQueue: head 3->4->5->6->7->8->9->NULL tail |

**对比LoopQueue与ArrayQueue**

|  |
| --- |
| **import** java.util.Random;  **public class** Main {   **private static double** testQueue(Queue<Integer> q, **int** opCount){  **long** startTime = System.*nanoTime*();  Random random = **new** Random();  **for** (**int** i = 0; i < opCount; i++) {  q.enqueue(random.nextInt(Integer.***MAX\_VALUE***));  }  **for** (**int** i = 0; i < opCount; i++) {  q.dequeue();  }   **long** endTime = System.*nanoTime*();  **return** (endTime - startTime)/1000000000.0;  }   **public static void** main(String[] args) {  **int** opCount = 100000;   ArrayQueue<Integer> arrayQueue = **new** ArrayQueue<>();  **double** time1 = *testQueue*(arrayQueue, opCount);  System.***out***.println(**"ArrayQueue, time: "** + time1 + **"s"**);   LoopQueue<Integer> loopQueue = **new** LoopQueue<>();  **double** time2 = *testQueue*(loopQueue, opCount);  System.***out***.println(**"LoopQueue, time: "** + time2 + **"s"**);   LinkedListQueue<Integer> linkedListQueue = **new** LinkedListQueue<>();  **double** time3 = *testQueue*(loopQueue, opCount);  System.***out***.println(**"LinkedListQueue, time: "** + time3 + **"s"**);  } } |
| ArrayQueue, time: 4.546770789s  LoopQueue, time: 0.017511647s  LinkedListQueue, time: 0.009680473s |

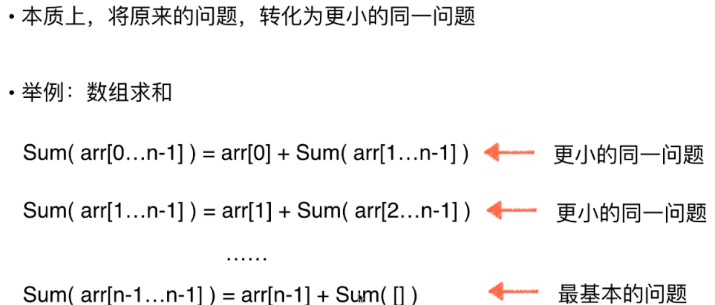
## 第5节链表和递归

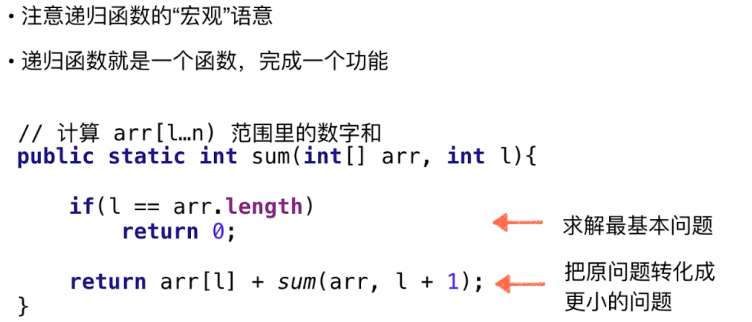
### 5.1 Leetcode中和链表相关的问题

从一个链表删除所有指定元素。

|  |
| --- |
| */\*\*  \* Created by BYF on 2019/1/31.  \*/* **public class** ListNode {  **int val**;  ListNode **next**;  **public** ListNode(**int** val){  **this**.**next** = **null**;  **this**.**val** = val;  }  *// 链表节点的构造函数  // 使用arr为入参，创建一个链表，当前的listNode为链表的头结点* **public** ListNode(**int** arr[]){  **if** (arr == **null** || arr.**length** == 0) {  **throw new** IllegalArgumentException(**"arr can not be empty."**);  }  **this**.**val** = arr[0];  ListNode cur = **this**;  **for** (**int** i = 1; i < arr.**length**; i++) {  cur.**next** = **new** ListNode(arr[i]);  cur = cur.**next**;  }  }  *// 以当前节点的头结点的链表信息字符串* **public** String toString() {  StringBuilder res = **new** StringBuilder();  ListNode cur = **this**;  **while** (cur != **null**) {  res.append(cur.**val** + **"->"**);  cur = cur.**next**;  }  res.append(**"NULL"**);  **return** res.toString();  }   **public static void** main(String[] args) {  **int**[] arr = {1, 2, 3, 5, 6, 1, 6};  ListNode head = **new** ListNode(arr);  System.***out***.println(head);  (**new** Solution()).removeElements(head,6);  System.***out***.println(head);   } } |
| 1->2->3->5->6->1->6->NULL  1->2->3->5->1->NULL |

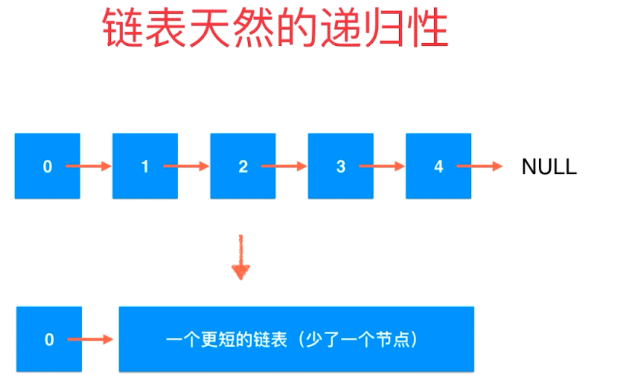
### 5.2链表与递归

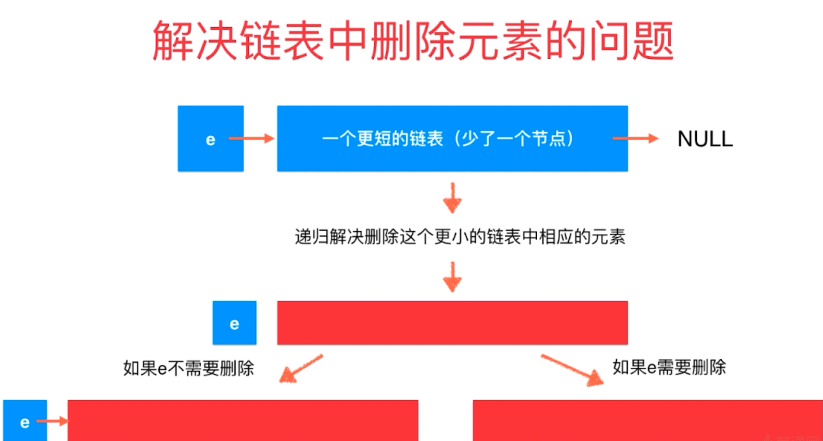




|  |
| --- |
| **public class** Sum {  **public static int** sum(**int**[] arr) {  **return** *sum*(arr, 0);  }  *// 计算区间[l,n)内所有的数字和* **private static int** sum(**int**[] arr, **int** l) {  **if** (l == arr.**length**)  **return** 0;  **return** arr[l] + *sum*(arr, l +1);  }   **public static void** main(String[] args) {  **int**[] nums = {1,2,3,4,5,6,7,8};  System.***out***.println(*sum*(nums));  } } |

### 5.3链表的天然递归结构性质



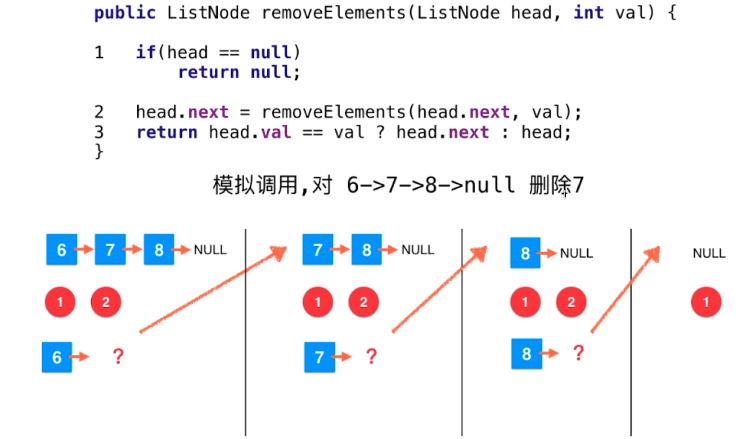


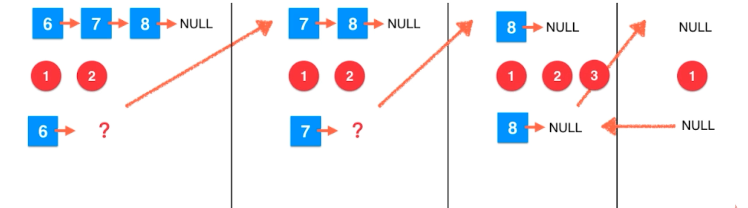
|  |
| --- |
| **public class** Solution3 {  **public** ListNode removeElement(ListNode head, **int** val){  **if** (head == **null** ) {  **return** head;  }  ListNode res = removeElement(head.**next**, val);  **if** (head.**val** == val) {  **return** res;  } **else** {  head.**next** = res;  **return** head;  }  }   **public static void** main(String[] args) {  **int**[] nums = {1,2,3,4,5,6,1,6};  ListNode head = **new** ListNode(nums);  System.***out***.println(head);  System.***out***.println((**new** Solution3()).removeElement(head, 6));  } } |

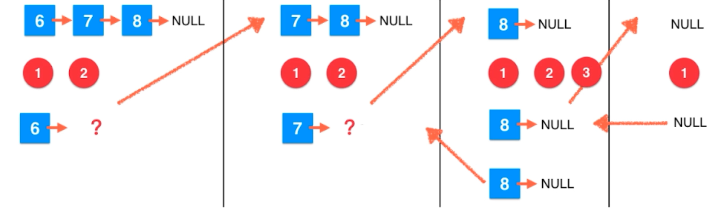
|  |
| --- |
| **public class** Solution3 {  **public** ListNode removeElement(ListNode head, **int** val){  **if** (head == **null** )  **return** head;  head.**next** = removeElement(head.**next**, val);  **return** head.**val** == val ? head.**next** : head;  }   **public static void** main(String[] args) {  **int**[] nums = {1,2,3,4,5,6,1,6};  ListNode head = **new** ListNode(nums);  System.***out***.println(head);  System.***out***.println((**new** Solution3()).removeElement(head, 6));  } } |

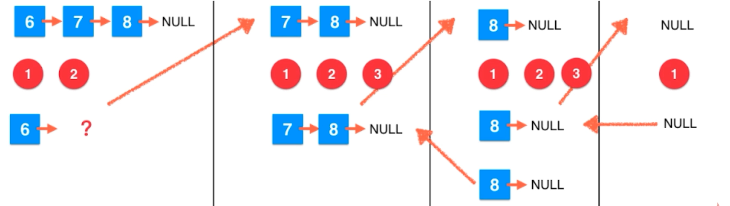
1->2->3->4->5->6->1->6->NULL

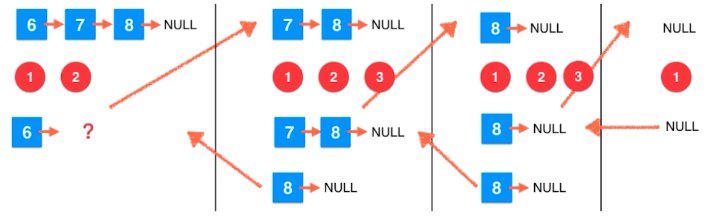
1->2->3->4->5->1->NULL

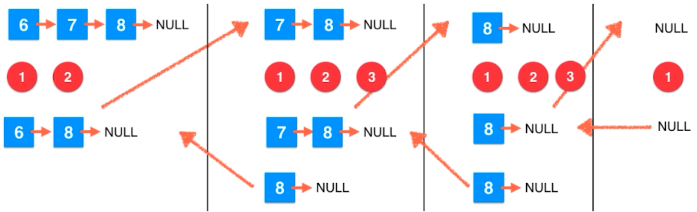


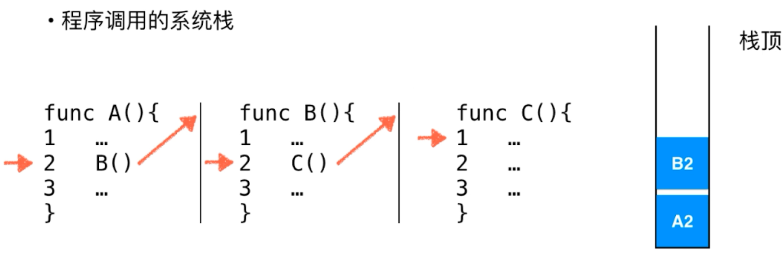










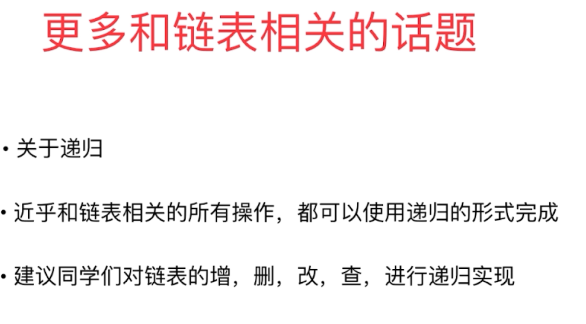


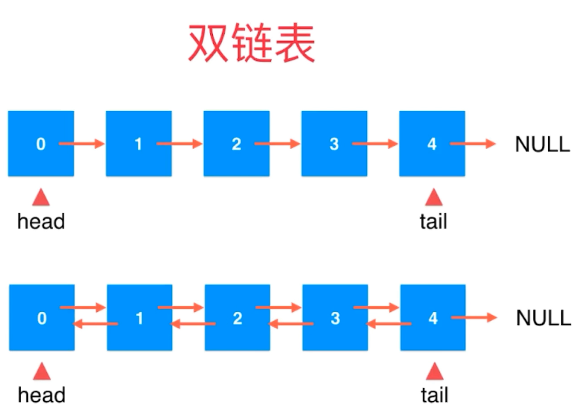


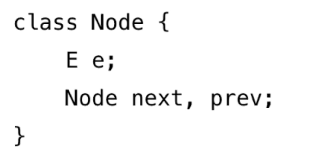
注意：栈的空间有限，超限则会抛出StackOverFlow异常

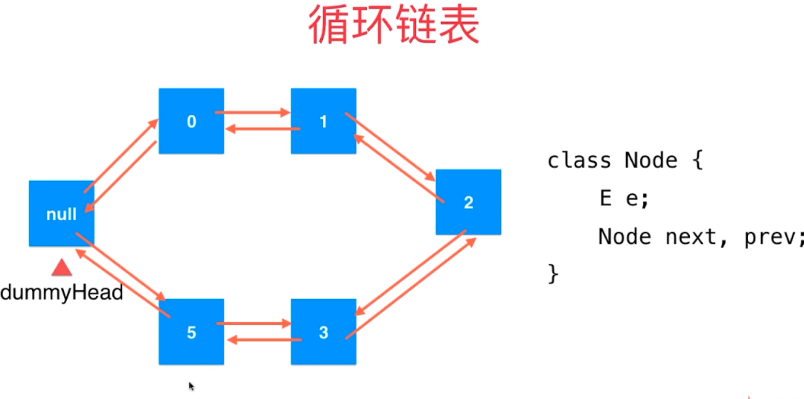
|  |
| --- |
| **public class** Solution3 {  **public** ListNode removeElement(ListNode head, **int** val,**int** depth){  String depthString = generateDepth(depth);  System.***out***.print(depthString);  System.***out***.println(**"Call: remove "** + val + **" in "** + head);  **if** (head == **null** ) {  System.***out***.print(depthString);  System.***out***.println(**"Return: "** + head);  **return** head;  }  ListNode res = removeElement(head.**next**, val, depth + 1 );  System.***out***.print(depthString);  System.***out***.println(**"After remove "** + val + **": "** + res);  ListNode ret;  **if** (head.**val** == val) {  ret = res;  } **else** {  head.**next** = res;  ret = head;  }  System.***out***.print(depthString);  System.***out***.println(**"Return "** + ret);  **return** ret;  }   **private** String generateDepth(**int** depth) {  StringBuilder res = **new** StringBuilder();  **for** (**int** i = 0; i < depth; i++) {  res.append(**"--"**);  }  **return** res.toString();  }   **public static void** main(String[] args) {  **int**[] nums = {1,2,3,4,5,6,1,6};  ListNode head = **new** ListNode(nums);  System.***out***.println(head);  System.***out***.println((**new** Solution3()).removeElement(head, 6,0));  } } |
| 1->2->3->4->5->6->1->6->NULL  Call: remove 6 in 1->2->3->4->5->6->1->6->NULL  --Call: remove 6 in 2->3->4->5->6->1->6->NULL  ----Call: remove 6 in 3->4->5->6->1->6->NULL  ------Call: remove 6 in 4->5->6->1->6->NULL  --------Call: remove 6 in 5->6->1->6->NULL  ----------Call: remove 6 in 6->1->6->NULL  ------------Call: remove 6 in 1->6->NULL  --------------Call: remove 6 in 6->NULL  ----------------Call: remove 6 in null  ----------------Return: null  --------------After remove 6: null  --------------Return null  ------------After remove 6: null  ------------Return 1->NULL  ----------After remove 6: 1->NULL  ----------Return 1->NULL  --------After remove 6: 1->NULL  --------Return 5->1->NULL  ------After remove 6: 5->1->NULL  ------Return 4->5->1->NULL  ----After remove 6: 4->5->1->NULL  ----Return 3->4->5->1->NULL  --After remove 6: 3->4->5->1->NULL  --Return 2->3->4->5->1->NULL  After remove 6: 2->3->4->5->1->NULL  Return 1->2->3->4->5->1->NULL  1->2->3->4->5->1->NULL |

### 5.4更多和链表相关的话题







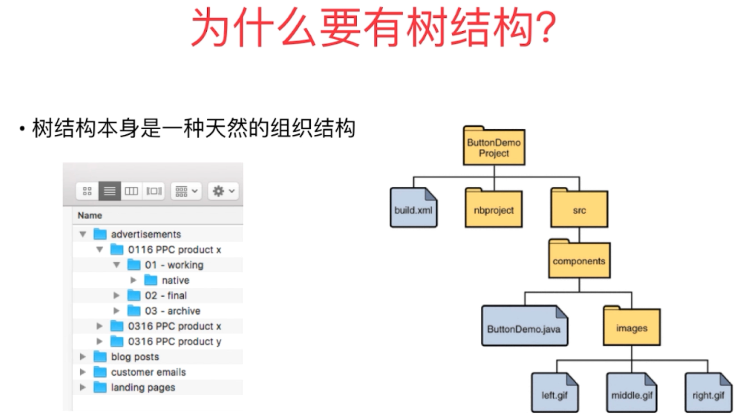


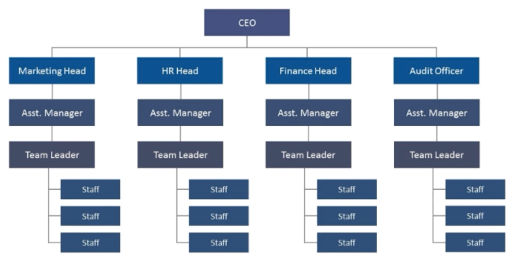
Java链表LinkedList实现原理

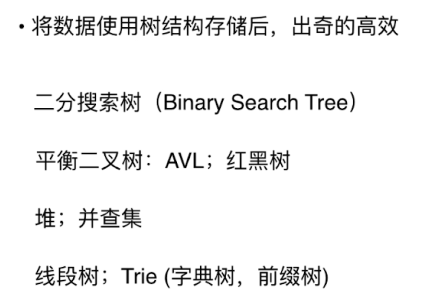


## 第6节二分搜索树

### 6.1为什么要研究二分搜索树



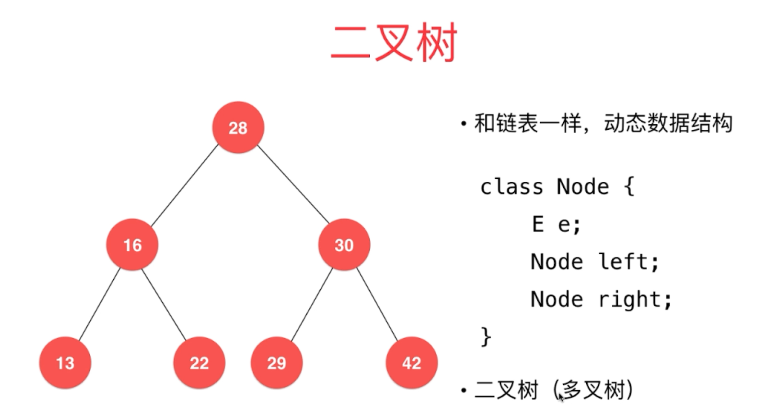


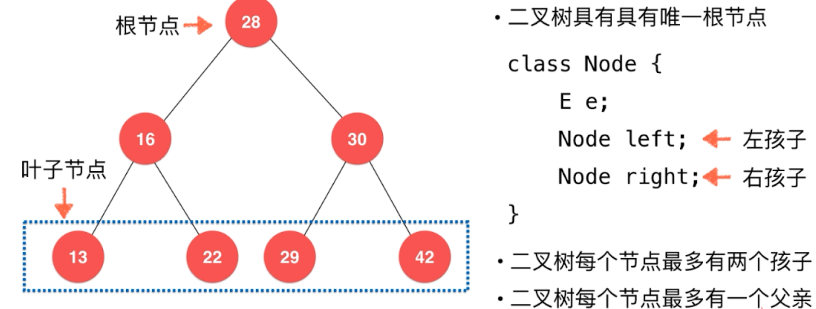


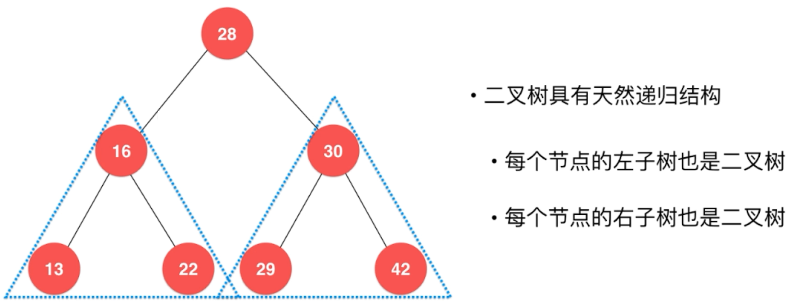
·高效

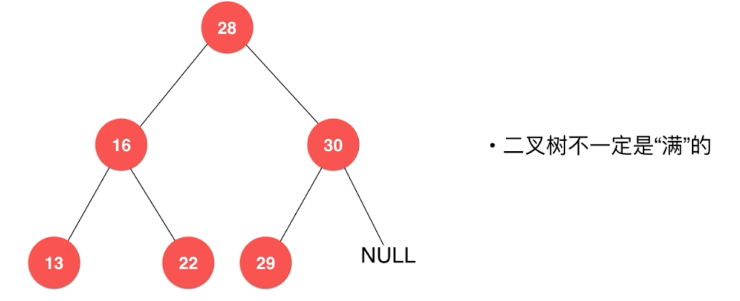
·处理特定问题不可或缺

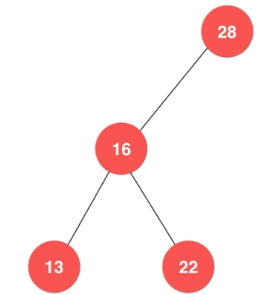
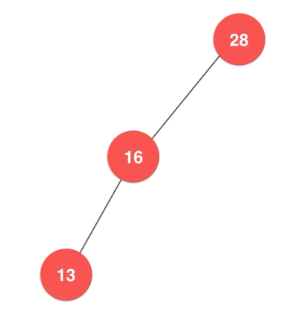
### 6.2二分搜索树基础

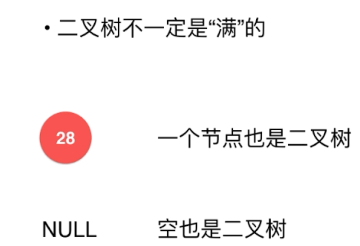


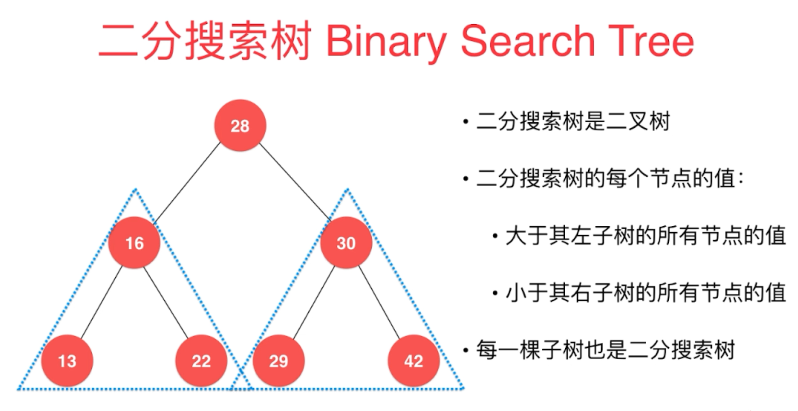


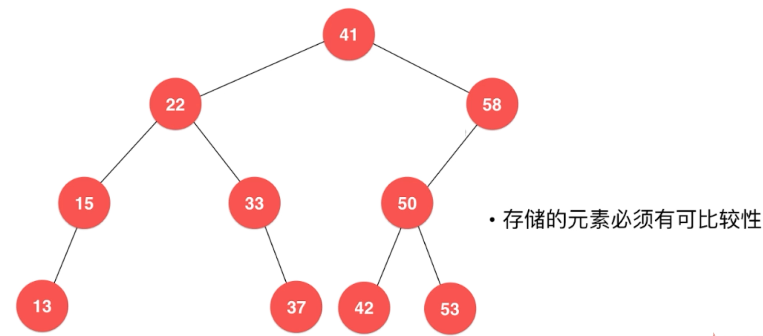












### 6.3向二分搜索树中添加元素

|  |
| --- |
| **public class** BST<E **extends** Comparable<E>> {  **public class** Node {  **private** E **e**;  **private** Node **left**, **right**;  **public** Node(E e){  **this**.**e** = e;  **left** = **null**;  **right** = **null**;  }  }  **private** Node **root**;  **private int size**;  **public** BST(){  **root** = **null**;  **size** = 0;  }  **public int** getSize(){  **return size**;  }  **public boolean** isEmpty(){  **return size** == 0;  }  *// 向二分搜索树中添加元素* **public** Node add(E e){  **if** (**root** == **null**) {  **root**.**e** = e;  **return root**;  }  **return** add(**root**, e);  }  *// 向以node为根的二分搜索树中插入元素e，递归算法  // 返回插入新节点后二分搜索树的根* **private** Node add(Node node, E e){  **if** (node.**e**.equals(e)){  **return** node;  } **else if** (node.**e**.compareTo(e) < 0 && node.**left** == **null**) {  node.**left** = **new** Node(e);  **size** ++;  **return** node;  } **else if** (node.**e**.compareTo(e) > 0 && node.**right** == **null**) {  node.**right** = **new** Node(e);  **size** ++;  **return** node;  }  **if** (node.**e**.compareTo(e) < 0){  **return** add(node.**left**, e);  } **else** {  **return** add(node.**right**, e);  }  }  } |

改进添加元素逻辑

|  |
| --- |
| **public class** BST<E **extends** Comparable<E>> {  **public class** Node {  **private** E **e**;  **private** Node **left**, **right**;  **public** Node(E e){  **this**.**e** = e;  **left** = **null**;  **right** = **null**;  }  }  **private** Node **root**;  **private int size**;  **public** BST(){  **root** = **null**;  **size** = 0;  }  **public int** getSize(){  **return size**;  }  **public boolean** isEmpty(){  **return size** == 0;  }  *// 向二分搜索树中添加元素* **public void** add(E e){  **root** = add(**root**, e);  }  *// 向以node为根的二分搜索树中插入元素e，递归算法  // 返回插入新节点后二分搜索树的根* **private** Node add(Node node, E e){  **if** (node == **null**) {  **size** ++;  node = **new** Node(e);  }  **if** (e.compareTo(node.**e**) < 0){  node.**left** = add(node.**left**, e);  } **else if** (e.compareTo(node.**e**) > 0){  node.**right** = add(node.**right**, e);  }  **return** node;  }  } |

### 6.4查找二分搜索树中元素e

|  |
| --- |
| *// 查找二分搜索树中是否包含元素e* **public boolean** contains(E e) {  **return** contains(**root**, e); } *// 查找以节点为node为根的二分搜索树，是否包含元素e* **private boolean** contains(Node node, E e){  **if** (node == **null**){  **return false**;  }  **if** (e.equals(node.**e**)){  **return true**;  } **else if** (e.compareTo(node.**e**) < 0) {  **return** contains(node.**left**, e);  } **else** {  **return** contains(node.**right**, e);  } } |

### 6.5二分搜索树遍历

|  |
| --- |
| **public class** BST<E **extends** Comparable<E>> {  **public class** Node {  **private** E **e**;  **private** Node **left**, **right**;  **public** Node(E e){  **this**.**e** = e;  **left** = **null**;  **right** = **null**;  }  }  **private** Node **root**;  **private int size**;  **public** BST(){  **root** = **null**;  **size** = 0;  }  **public int** getSize(){  **return size**;  }  **public boolean** isEmpty(){  **return size** == 0;  }  *// 向二分搜索树中添加元素* **public void** add(E e){  **root** = add(**root**, e);  }  *// 向以node为根的二分搜索树中插入元素e，递归算法  // 返回插入新节点后二分搜索树的根* **private** Node add(Node node, E e){  **if** (node == **null**) {  **size** ++;  node = **new** Node(e);  }  **if** (e.compareTo(node.**e**) < 0){  node.**left** = add(node.**left**, e);  } **else if** (e.compareTo(node.**e**) > 0){  node.**right** = add(node.**right**, e);  }  **return** node;  }  *// 查找二分搜索树中是否包含元素e* **public boolean** contains(E e) {  **return** contains(**root**, e);  }  *// 查找以节点为node为根的二分搜索树，是否包含元素e* **private boolean** contains(Node node, E e){  **if** (node == **null**){  **return false**;  }  **if** (e.equals(node.**e**)){  **return true**;  } **else if** (e.compareTo(node.**e**) < 0) {  **return** contains(node.**left**, e);  } **else** {  **return** contains(node.**right**, e);  }  }  *// 二分搜索树的前序遍历* **public void** preOrder() {  preOrder(**root**);  }  **private void** preOrder(Node node){  **if** (node == **null**) {  **return**;  }  System.***out***.println(node.**e**);  preOrder(node.**left**);  preOrder(node.**right**);  }  *// 二分搜索树中序遍历* **public void** inOrder() {  inOrder(**root**);  }  **private void** inOrder(Node node){  **if** (node == **null**) {  **return**;  }  inOrder(node.**left**);  System.***out***.println(node.**e**);  inOrder(node.**right**);  }  *// 二分搜索树的后续遍历* **public void** postOrder() {  postOrder(**root**);  }  **private void** postOrder(Node node){  **if** (node == **null**) {  **return**;  }  inOrder(node.**left**);  inOrder(node.**right**);  System.***out***.println(node.**e**);  }  @Override  **public** String toString(){  StringBuilder res = **new** StringBuilder();  generateBSTString(**root**, 0, res);  **return** res.toString();  }  *// 生成以node为根节点，深度为depth的描述二叉树的字符串* **private void** generateBSTString(Node node, **int** depth, StringBuilder res){  **if**(node == **null**){  res.append(generateDepthString(depth) + **"null\n"**);  **return**;  }  res.append(generateDepthString(depth) + node.**e** + **"\n"**);  generateBSTString(node.**left**, depth + 1, res);  generateBSTString(node.**right**, depth + 1, res);  }  **private** String generateDepthString(**int** depth){  StringBuilder res = **new** StringBuilder();  **for**(**int** i = 0 ; i < depth ; i ++)  res.append(**"--"**);  **return** res.toString();  } } |

### 6.6二分搜索树遍历的非递归实现

|  |
| --- |
| *// 二分搜索树的前序遍历的非递归实现* **public void** preOrderNR() {  preOrderNR(**root**); } **private void** preOrderNR(Node node){  Stack<Node> stack = **new** Stack<>();  stack.push(node);  **while** (!stack.isEmpty()) {  Node cur = stack.pop();  System.***out***.println(cur.**e**);  **if** (cur.**right** != **null**){  stack.push(cur.**right**);  }  **if** (cur.**left** != **null**) {  stack.push(cur.**left**);  }  } } |

### 6.7二分搜索树层序遍历

|  |
| --- |
| *// 二分搜索树的层序遍历* **public void** levelOrder(){  levelOrder(**root**); } **private void** levelOrder(Node node) {  Queue<Node> queue = **new** LinkedList<>();  queue.add(node);  **while** (! queue.isEmpty()) {  Node cur = queue.remove();  System.***out***.println(cur.**e**);  **if** (cur.**left** != **null**) {  queue.add(cur.**left**);  }  **if** (cur.**right** != **null**) {  queue.add(cur.**right**);  }  } } |

### 6.8查找、删除二分搜索树最大、最小元素

|  |
| --- |
| *// 寻找二分搜索树的最小节点* **public** E minimum(){  **if** (**size** == 0)  **throw new** IllegalArgumentException(**"BST is empty."**);  Node minNode = minimum(**root**);  **return** minNode.**e**; } **private** Node minimum(Node node){  **if** (node.**left** == **null**)  **return** node;  **return** minimum(node.**left**); } *// 寻找二分搜索树的最大元素* **public** E maximum(){  **if** (**size** == 0)  **throw new** IllegalArgumentException(**"BST is empty."**);  **return** maximum(**root**).**e**; } **private** Node maximum(Node node) {  **if** (node.**right** == **null**)  **return** node;  **return** maximum(node.**right**); } *// 从二分搜索树中删除最小值所在节点, 返回最小值* **public** E removeMin(){  E ret = minimum();  **root** = removeMin(**root**);  **return** ret; } |

|  |
| --- |
| *// 删除掉以node为根的二分搜索树中的最小节点 // 返回删除节点后新的二分搜索树的根* **private** Node removeMin(Node node){   **if**(node.**left** == **null**){  Node rightNode = node.**right**;  node.**right** = **null**;  **size** --;  **return** rightNode;  }   node.**left** = removeMin(node.**left**);  **return** node; } *// 从二分搜索树中删除最大值所在节点* **public** E removeMax(){  E ret = maximum();  **root** = removeMax(**root**);  **return** ret; }  *// 删除掉以node为根的二分搜索树中的最大节点 // 返回删除节点后新的二分搜索树的根* **private** Node removeMax(Node node){   **if**(node.**right** == **null**){  Node leftNode = node.**left**;  node.**left** = **null**;  **size** --;  **return** leftNode;  }   node.**right** = removeMax(node.**right**);  **return** node; } |

### 6.9求交集

集合、映射

TreeSet/TreeMap：平衡二叉树

HashSet/HashMap：哈希表