

## 数组基础

数据的索引

对于索引的理解

## 封装数组

动态数组设计过程

V1.0版本：基础数组结构

V1.1版本：向数组中添加元素

V1.2版本：向指定的位置插入指定的元素，提供一个addFirst

V1.3版本：在数组中查询元素和修改元素

V1.4版本：包含搜索和删除

V1.5版本：使用泛型

V1.5版本：动态数组

V1.51经典版程序

V1.6版本：使用泛型

数组时间复杂度算法简单分析

1，简单复杂度分析

2，均摊复杂度和防止复杂度的震荡

3，复杂度震荡

## 栈 (stack)

展示元素入栈的过程

展示元素出栈的过程

小结：

案例1

案例2

栈的基本实现

Array

Stack

ArrayStack

Main

栈的时间复杂度分析

Stack案例实操（编译器对括号的匹配报错机制）

## 队列

定义：

Array

ArrayQueue

Queue

循环队列与数组队列的时间复杂度分析

## 链表

数据存储在“节点”（node）中

数组结构和链表结构的对比

给链表中添加元素

在表头添加数据

在链表指定索引处添加节点

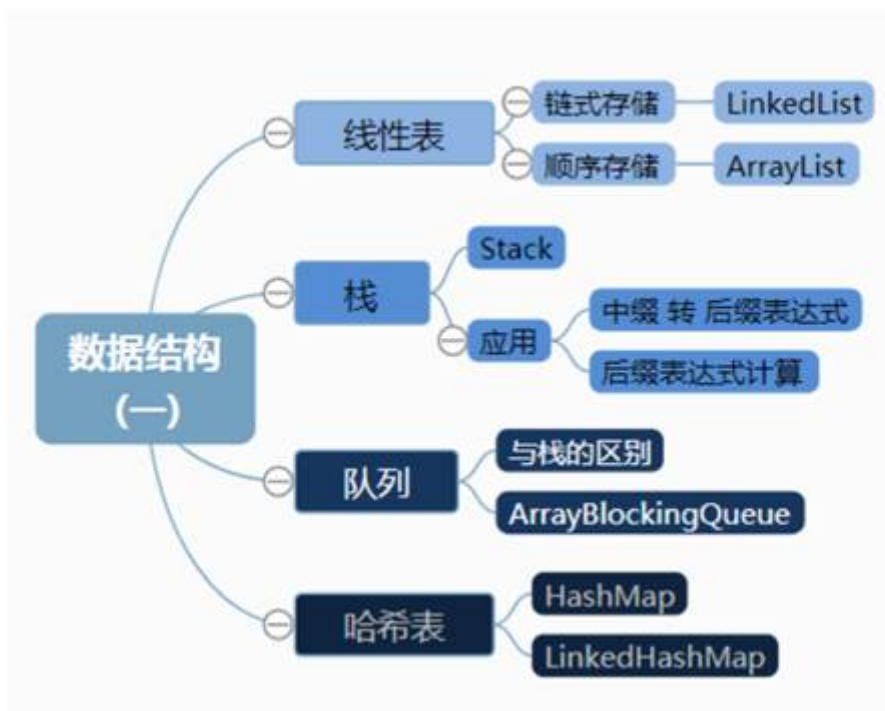
给链表使用虚拟头结点

链表的遍历，查询和修改

从链表中删除元素

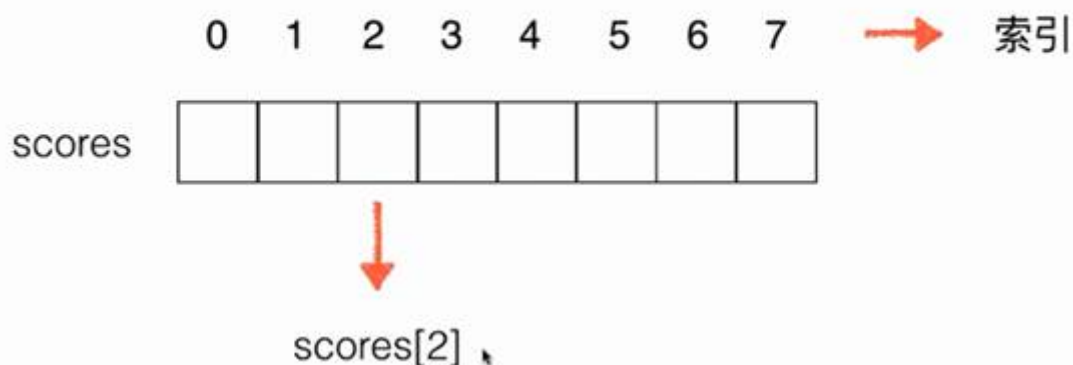
使用链表实现栈

使用链表实现队列结构



## 数组基础

- 把数据码成一排进行存放



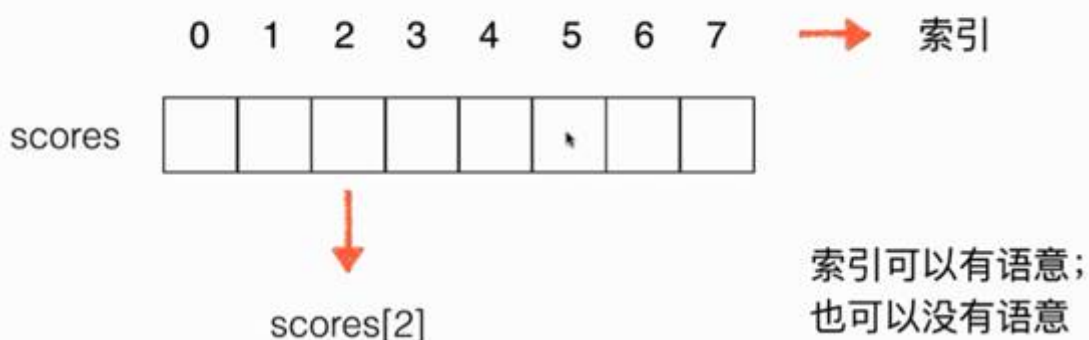
给数组取一个名字Array ——arr

真实环境中更需要给数组取一个有实际意义的名字

## 数据的索引

索引概念很重要，可以有语义也可以没有语义

- 把数据码成一排进行存放



## 对于索引的理解

- 1, 数组的最大优点: 快速查询——scores【2】
- 2, 数组最好应用于“索引有语意”的情况
- 3, 但并非所有有语意的索引都适用于数组
- 身份证号: 50010520189898
- 4, 数组也可以处理索引没有语意的情况
- 5, 本章处理主要就是处理“索引没有语意”的情况数组的使用

需求: 遍历数组, 打印一个班的成绩

```
public class Main {  
    public static void main(String[] args) {  
        int[] arr = new int[20];  
        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 i = 0 ; i < scores.length ; i ++)  
            System.out.println(scores[i]);  
    }  
}
```

## 封装数组

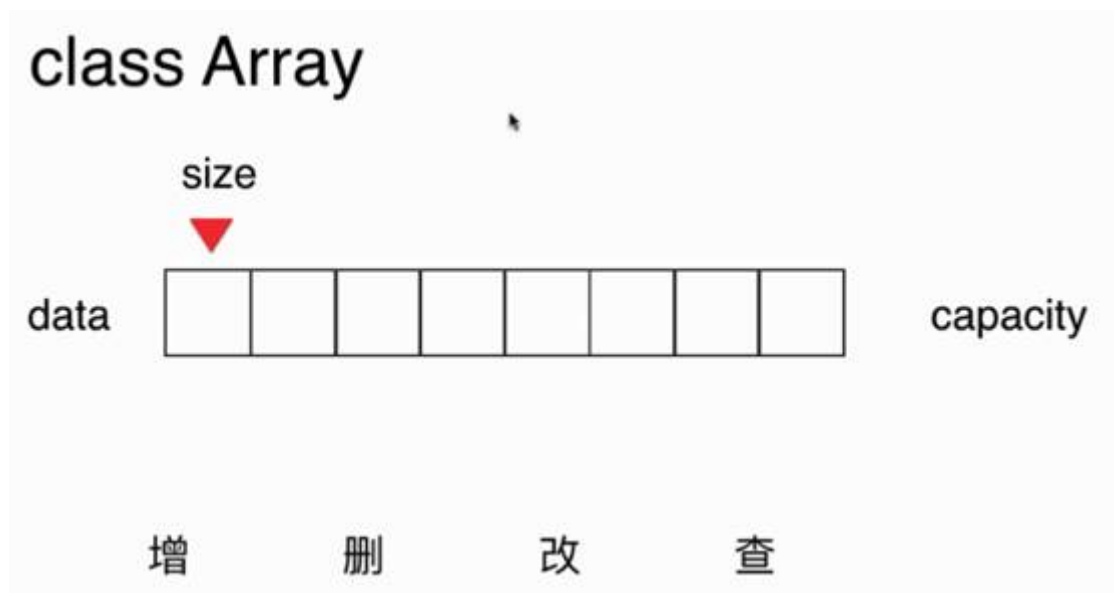
	0	1	2	3	4	5	6	7
scores	100	99	66					

索引没有语意, 如何表示没有元素?

如何添加元素? 如何删除元素?

提出需求: 需要基于java数组, 二次封装属于我们自己的数组类, 区别于java本身的静态数组, 性能媲美静态数组

## 动态数组设计过程



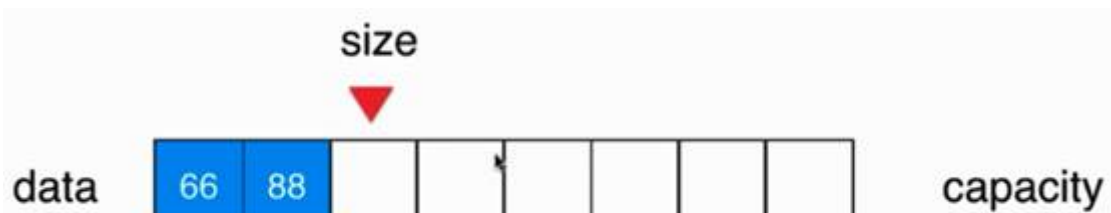
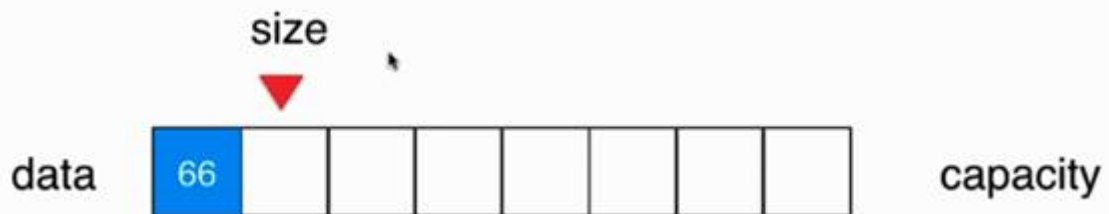
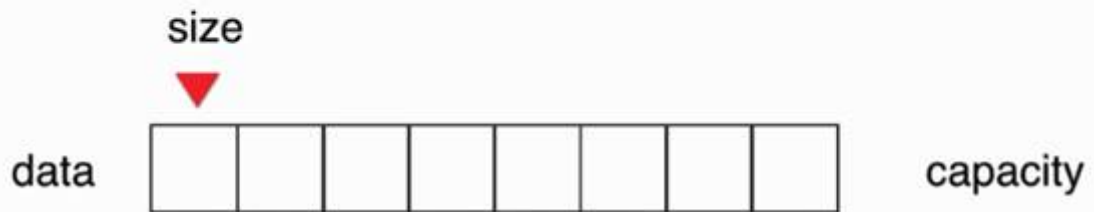
Capacity: 容量

## V1.0版本：基础数组结构

```
1  class MyArray{
2      //私有化数据
3      private int[] data;
4      //定义长度
5      private int size;
6      //构造函数，传入数组的容量capacity构造array
7      public MyArray(int capacity){
8          data=new int[capacity];
9          size=0;
10     }
11     //无参构造函数，默认数组的容量capacity=10
12     public MyArray(){
13         this(10);
14     }
15     //获取数组中的元素个数
16     public int getSize(){
17         return size;
18     }
19     //获取数组容量
20     public int getCapacity(){
21         return data.length;
22     }
23     //返回数组是否为空 不是 为空 非空 为空 true
24     public boolean isEmpty(){
25         return size==0;
26     }
27 }
28
```

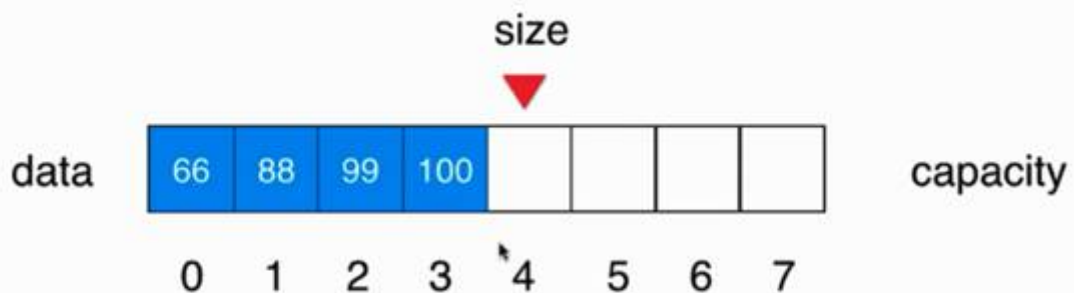
## V1.1版本：向数组中添加元素

## 向数组末添加元素



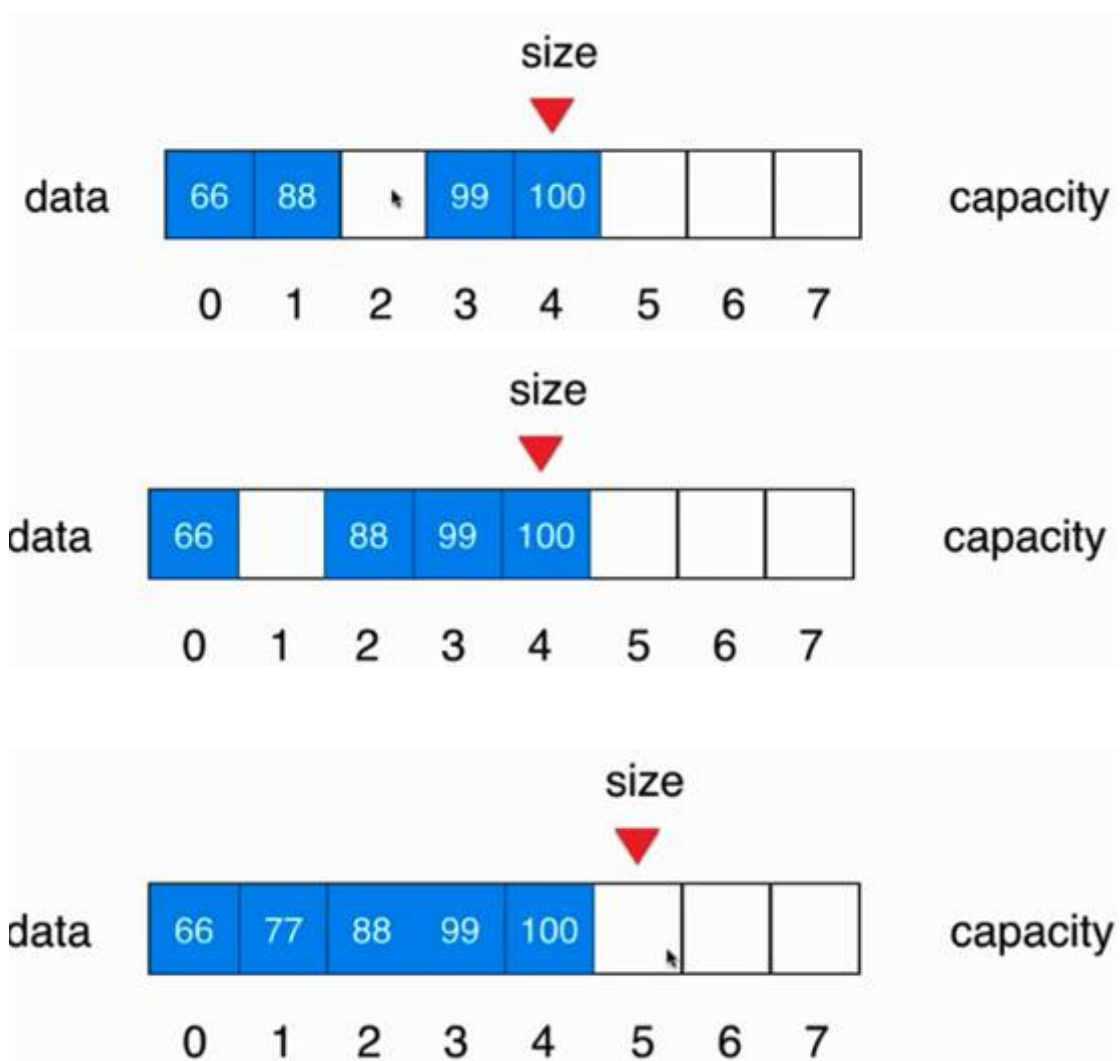
```
1 //向所有元素后添加一个新元素
2 public void addLast(int e){
3     if(size==data.length)
4         throw new IllegalArgumentException("addlast failed,array is
full");
5 //    data[size++]=e;不便于阅读
6     data[size]=e;
7     size++;
8 }
9
```

## V1.2版本：向指定的位置插入指定的元素，提供一个addFirst



把77插入到索引为1的位置

77



代码写完后需要修改刚才的增加函数

```

1      //向所有元素后添加一个新元素
2      public void addLast(int e){
3          //      if(size==data.length)
4          //          throw new IllegalArgumentException("addlast failed,array is
full");
5          ////      data[size++]=e;不便于阅读
6          //      data[size]=e;
7          //      size++;
8          add(size, e);
9
10     }
11
12
13     // 在所有元素前添加一个新元素
14     public void addFirst(int e){
15         add(0, e);
16     }
17
18     // 在index索引的位置插入一个新元素e
19     public void add(int index, int e){
20
21         if(size == data.length)
22             throw new IllegalArgumentException("Add failed. Array is
full.");
23

```

```

24         if(index < 0 || index > size)
25             throw new IllegalArgumentException("Add failed. Require index >=
0 and index <= size.");
26
27         for(int i = size - 1; i >= index ; i --)
28             data[i + 1] = data[i];
29         data[index] = e;
30         size ++;
31     }
32

```

## V1.3版本：在数组中查询元素和修改元素

```

1         // 获取index索引位置的元素
2         public int get(int index){
3             if(index < 0 || index >= size)
4                 throw new IllegalArgumentException("Get failed. Index is
illegal.");
5             return data[index];
6         }
7
8         // 修改index索引位置的元素为e
9         public void set(int index, int e){
10             if(index < 0 || index >= size)
11                 throw new IllegalArgumentException("Set failed. Index is
illegal.");
12             data[index] = e;
13         }
14
15         @Override
16         public String toString(){
17
18             StringBuilder res = new StringBuilder();
19             res.append(String.format("Array: size = %d , capacity = %d\n", size,
data.length));
20             res.append('[');
21             for(int i = 0 ; i < size ; i ++){
22                 res.append(data[i]);
23                 if(i != size - 1)
24                     res.append(", ");
25             }
26             res.append(']');
27             return res.toString();
28         }
29

```

## V1.4版本：包含搜索和删除

```

1         // 查找数组中是否有元素e
2         public boolean contains(int e){
3             for(int i = 0 ; i < size ; i ++){
4                 if(data[i] == e)
5                     return true;
6             }
7             return false;
8         }

```

```

9
10 // 查找数组中元素e所在的索引，如果不存在元素e，则返回-1
11 public int find(int e){
12     for(int i = 0 ; i < size ; i ++){
13         if(data[i] == e)
14             return i;
15     }
16     return -1;
17 }
18
19 // 从数组中删除index位置的元素，返回删除的元素
20 public int remove(int index){
21     if(index < 0 || index >= size)
22         throw new IllegalArgumentException("Remove failed. Index is
illegal.");
23     int ret = data[index];
24     for(int i = index + 1 ; i <= size ; i ++){
25         data[i - 1] = data[i];
26     }
27     size --;
28     return ret;
29 }
30
31 // 从数组中删除第一个元素，返回删除的元素
32 public int removeFirst(){
33     return remove(0);
34 }
35
36 // 从数组中删除最后一个元素，返回删除的元素
37 public int removeLast(){
38     return remove(size - 1);
39 }
40
41 // 从数组中删除元素e
42 public void removeElement(int e){
43     int index = find(e);
44     if(index != -1)
45         remove(index);
46 }

```

## V1.5版本：使用泛型

理由：



- 让我们的数据结构可以放置“任何”数据类型

- 不可以是基本数据类型，只能是类对象

boolean , byte , char , short , int , long , float , double

- 每个基本数据类型都有对应的包装类

Boolean , Byte , Char , Short , Int , Long , Float , Double

代码改造: Array

```
1 package com.haoyu;
2
3 public class Array<E> {
4
5     private E[] data;
6     private int size;
7
8     // 构造函数，传入数组的容量capacity构造Array
9     public Array(int capacity){
10         data = (E[])new Object[capacity];
11         size = 0;
12     }
13
14     // 无参数的构造函数，默认数组的容量capacity=10
15     public Array(){
16         this(10);
17     }
18
19     // 获取数组的容量
20     public int getCapacity(){
21         return data.length;
22     }
23
24     // 获取数组中的元素个数
25     public int getSize(){
26         return size;
27     }
28
29     // 返回数组是否为空
30     public boolean isEmpty(){
31         return size == 0;
32     }
33
34     // 在index索引的位置插入一个新元素e
35     public void add(int index, E e){
36
37         if(size == data.length)
```

```
38         throw new IllegalArgumentException("Add failed. Array is
full.");
39
40         if(index < 0 || index > size)
41             throw new IllegalArgumentException("Add failed. Require index
>= 0 and index <= size.");
42
43         for(int i = size - 1; i >= index ; i --)
44             data[i + 1] = data[i];
45
46         data[index] = e;
47
48         size ++;
49     }
50
51     // 向所有元素后添加一个新元素
52     public void addLast(E e){
53         add(size, e);
54     }
55
56     // 在所有元素前添加一个新元素
57     public void addFirst(E e){
58         add(0, e);
59     }
60
61     // 获取index索引位置的元素
62     public E get(int index){
63         if(index < 0 || index >= size)
64             throw new IllegalArgumentException("Get failed. Index is
illegal.");
65         return data[index];
66     }
67
68     // 修改index索引位置的元素为e
69     public void set(int index, E e){
70         if(index < 0 || index >= size)
71             throw new IllegalArgumentException("Set failed. Index is
illegal.");
72         data[index] = e;
73     }
74
75     // 查找数组中是否有元素e
76     public boolean contains(E e){
77         for(int i = 0 ; i < size ; i ++){
78             if(data[i].equals(e))
79                 return true;
80         }
81         return false;
82     }
83
84     // 查找数组中元素e所在的索引，如果不存在元素e，则返回-1
85     public int find(E e){
86         for(int i = 0 ; i < size ; i ++){
87             if(data[i].equals(e))
88                 return i;
89         }
90         return -1;
91     }
```

```

92
93 // 从数组中删除index位置的元素，返回删除的元素
94 public E remove(int index){
95     if(index < 0 || index >= size)
96         throw new IllegalArgumentException("Remove failed. Index is
illegal.");
97
98     E ret = data[index];
99     for(int i = index + 1 ; i < size ; i ++){
100         data[i - 1] = data[i];
101     }
102     size --;
103     data[size] = null; // loitering objects != memory leak
104     return ret;
105 }
106
107 // 从数组中删除第一个元素，返回删除的元素
108 public E removeFirst(){
109     return remove(0);
110 }
111
112 // 从数组中删除最后一个元素，返回删除的元素
113 public E removeLast(){
114     return remove(size - 1);
115 }
116
117 // 从数组中删除元素e
118 public void removeElement(E e){
119     int index = find(e);
120     if(index != -1)
121         remove(index);
122 }
123
124 @Override
125 public String toString(){
126     StringBuilder res = new StringBuilder();
127     res.append(String.format("Array: size = %d , capacity = %d\n",
size, data.length));
128     res.append('[');
129     for(int i = 0 ; i < size ; i ++){
130         res.append(data[i]);
131         if(i != size - 1)
132             res.append(", ");
133     }
134     res.append(']');
135     return res.toString();
136 }
137 }
138

```

测试类: student

```

1 package com.haoyu;
2
3 public class Student {
4
5     private String name;

```

```

6     private int score;
7
8     public Student(String studentName, int studentScore){
9         name = studentName;
10        score = studentScore;
11    }
12
13    @Override
14    public String toString(){
15        return String.format("Student(name: %s, score: %d)", name, score);
16    }
17
18    public static void main(String[] args) {
19
20        Array<Student> arr = new Array<Student>();
21        arr.addLast(new Student("Alice", 100));
22        arr.addLast(new Student("Bob", 66));
23        arr.addLast(new Student("Charlie", 88));
24        System.out.println(arr);
25    }
26 }
27
28

```

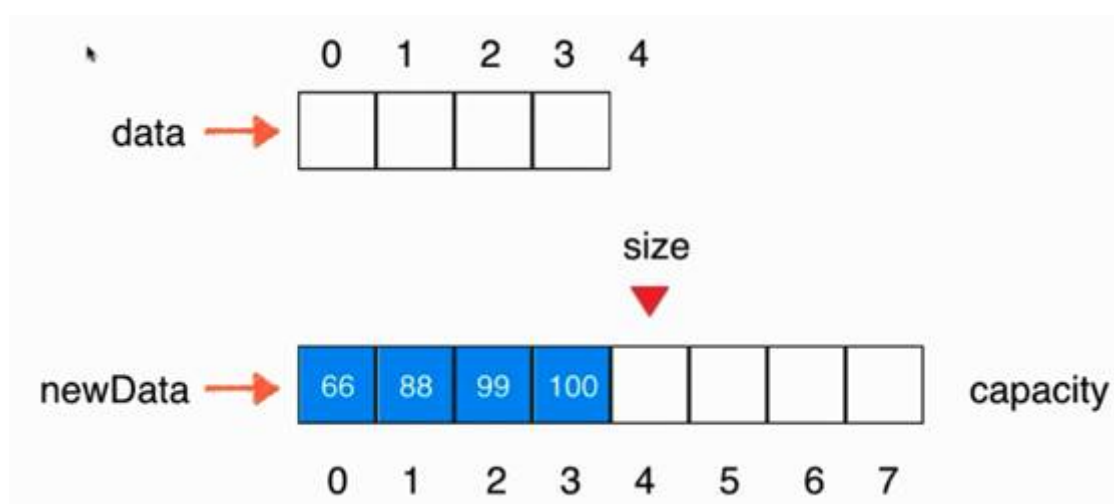
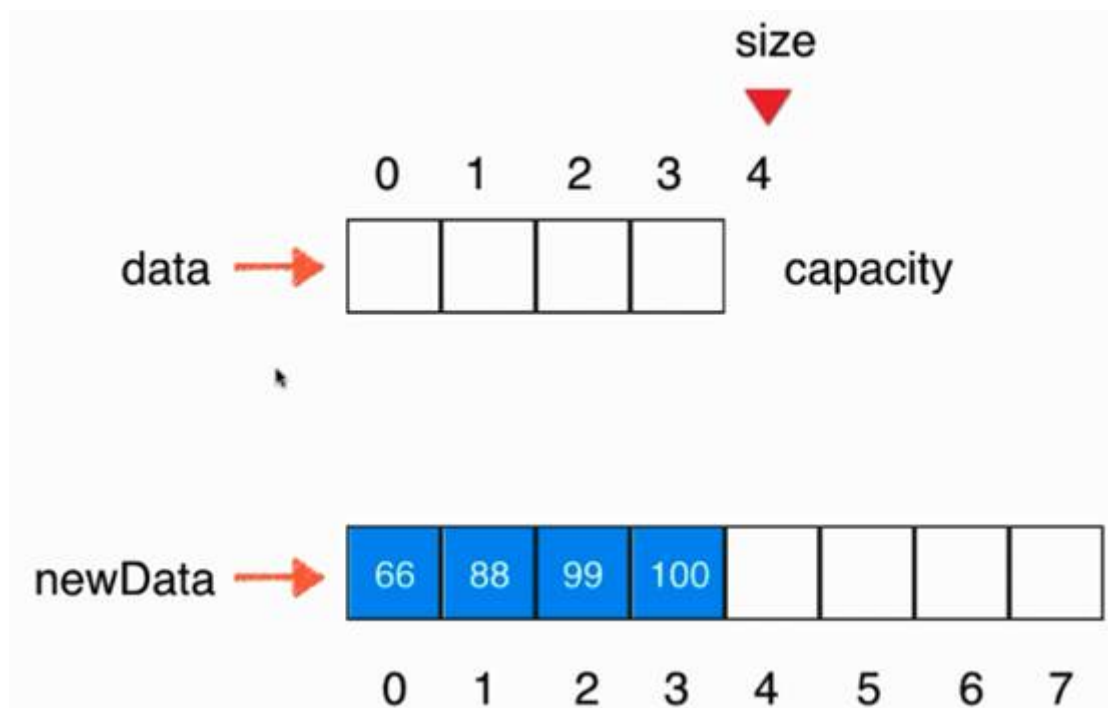
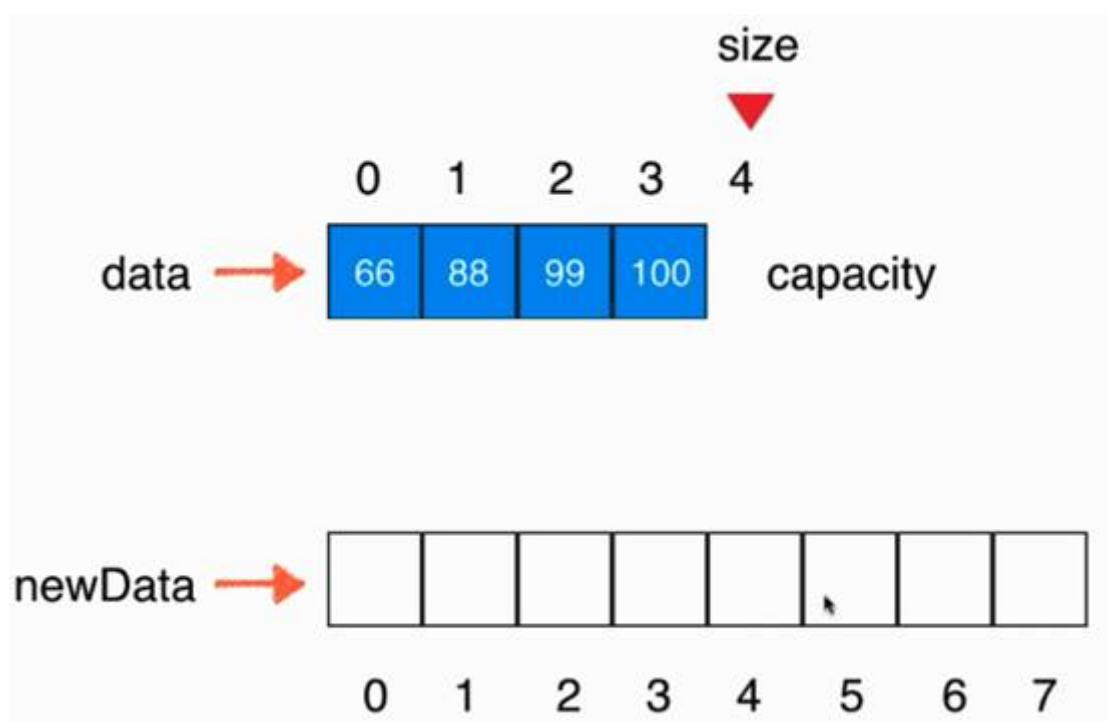
打印类:

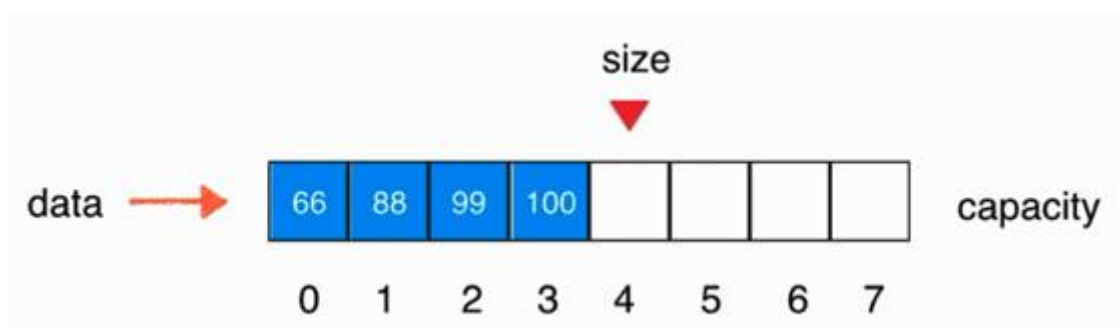
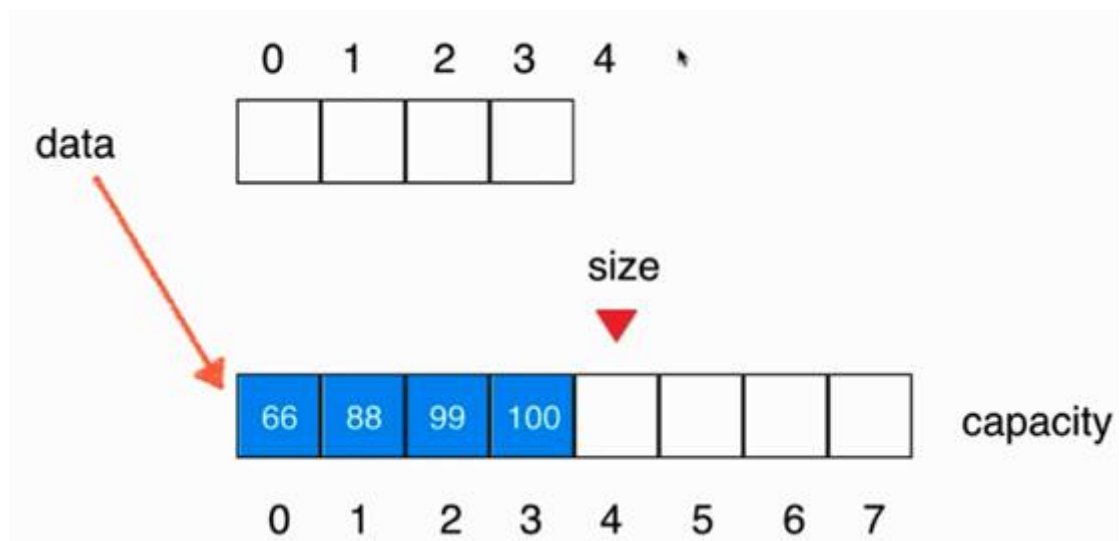
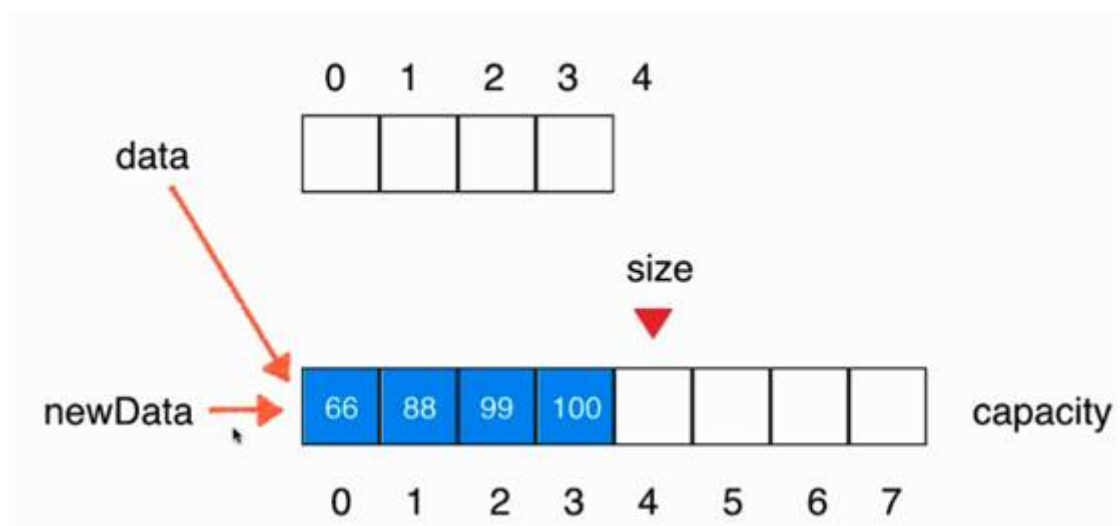
```

1     public class Main {
2
3         public static void main(String[] args) {
4
5             Array<Integer> arr = new Array<Integer>(20);
6             for(int i = 0 ; i < 10 ; i ++){
7                 arr.addLast(i);
8                 System.out.println(arr);
9
10            arr.add(1, 100);
11            System.out.println(arr);
12
13            arr.addFirst(-1);
14            System.out.println(arr);
15            // [-1, 0, 100, 1, 2, 3, 4, 5, 6, 7, 8, 9]
16
17            arr.remove(2);
18            System.out.println(arr);
19
20            arr.removeElement(4);
21            System.out.println(arr);
22
23            arr.removeFirst();
24            System.out.println(arr);
25        }
26    }
27

```

## V1.5版本：动态数组





```

1  // 在index索引的位置插入一个新元素e
2  public void add(int index, E e){
3
4      if(index < 0 || index > size)
5          throw new IllegalArgumentException("Add failed. Require index >=
0 and index <= size.");
6
7      if(size == data.length)
8          resize(2 * data.length);
9
10     for(int i = size - 1; i >= index ; i --)
11         data[i + 1] = data[i];
12
13     data[index] = e;
14
15     size ++;
16 }

```

```

17
18 // 将数组空间的容量变成newCapacity大小
19 private void resize(int newCapacity){
20
21     E[] newData = (E[])new Object[newCapacity];
22     for(int i = 0 ; i < size ; i ++){
23         newData[i] = data[i];
24     }
25     data = newData;
26

```

测试:

```

1 public class Main {
2     public static void main(String[] args) {
3         Array<Integer> arr = new Array<Integer>();
4         for(int i = 0 ; i < 10 ; i ++){
5             arr.addLast(i);
6             System.out.println(arr);
7
8             arr.add(1, 100);
9             System.out.println(arr);
10
11            arr.addFirst(-1);
12            System.out.println(arr);
13
14            arr.remove(2);
15            System.out.println(arr);
16
17            arr.removeElement(4);
18            System.out.println(arr);
19
20            arr.removeFirst();
21            System.out.println(arr);
22        }
23    }
24

```

## V1.51经典版程序

```

1 import java.util.Arrays;
2 /**
3  * <p>Title: Demo19.java</p>
4  * <p>Description:
5  * 125经典版 </p>
6  * <p>Copyright: Copyright (c) 2017</p>
7  * <p>Company: com.haoyu</p>
8  * @author 大师
9  * @date 2019年8月14日
10 * @version 1.0
11 */
12 //定义集合--多功能的简便操作的数组
13 //my 我的 array 数组 list 列表
14 //我的数组增强功能后的列表类--线性数组集合类
15 class MyArrayList{
16     //声明要准备好空间，等待后面存入元素

```

```

17     private int[] data;
18     //定义元素个数
19     private int size;
20     //定义一个初始化数组容量大小
21     private int capacity=16;
22     //new的时候保证初始化空间与个数
23     public MyArrayList(int capacity) {
24         //更改初始化的长度
25         //健壮性判断
26         if(capacity<=0) {
27             data=new int[this.capacity];
28         }else {
29             data=new int[capacity];
30         }
31         //由于现在没有存储元素，因此元素个数=0
32         this.size=0;
33     }
34     //无参构造函数，默认数组的容量capacity=10
35     public MyArrayList(){
36         //调用其他的带参数的对应构造函数
37         this(10);
38     }
39     public int getSize() {
40         return size;
41     }
42     //获取数组容量--数组定义后开辟的空间个数
43     public int getCapacity(){
44         //数组在初始化后就开辟的空间个数，只不过里面目前有没有元素。不清楚
45         return data.length;
46     }
47     //返回数组"是为空" false isEmpty-不为空 true isEmpty-为空
48     public boolean isEmpty(){
49         return size==0;
50     }
51     //在数组末尾增加元素
52     //last 最后 e-element:元素
53     //MyArrayList [data=[1, 2, 3, 0, 0, 0, 0, 0, 0, 0]]
54     // public void addLast(int e) {
55     //     //判断一下当前的空间是否已经容量满员。如果已经满员了，就不应该执行增加操作了
56     //     if(size==data.length) {
57     //         //一旦出现问题，throw 抛出去一个问题 xxxException exception: 异常，
58     //         //后面的java代码不会再继续执行
59     //         throw new RuntimeException("容量已满");
60     //     }
61     //     //size当前位置增加一个元素（赋值一个元素）
62     //     data[size]=e;
63     //     //size往后走一位
64     //     size++;
65     // }
66
67     public void addLast(int e) {
68         insert(size,e);
69     }
70     public void addFirst(int e) {
71         insert(0,e);
72     }
73     //插入元素 index=插入的位置--数组元素角标

```



```

74 //index-0  ++ e-element: 元素
75 public void insert(int index,int e) {
76     //判断一下当前的空间是否已经容量满员。如果已经满员了，就不应该执行增加操作了
77     if(size==data.length) {
78         //一旦出现问题，throw 抛出去一个问题 xxxException exception: 异常，
问题
79         //后面的java代码不会继续执行
80         // throw new RuntimeException("容量已满");
81         resize(2*data.length);
82     }
83     //插歪了，让代码停止执行
84     if(index < 0 || index > size) {
85         throw new IllegalArgumentException("Add failed. Require index
86         >= 0 and index <= size.");
87     }
88     //MyArrayList [data=[1, 2, 3, 0, 0, 0, 0, 0, 0, 0]]
89     //size=3
90     //i=size-1=2 data[2]=3
91     //i-- i>=index index=0
92     //2 1 0:存在元素要依次往后走一位 e=data[index=0]
93     for(int i=size-1;i>=index;i--) {
94         data[i+1]=data[i];
95     }
96     //剩下的[index]=e
97     data[index]=e;
98     //由于是增加了一个元素
99     size++;
100
101     if(size==data.length) {
102         resize(2*data.length);
103     }
104
105     //获取元素 index-0~size-1
106     public int get(int index) {
107         //查歪了
108         if(index < 0 || index > size-1) {
109             throw new IllegalArgumentException("Add failed. Require index
110             >= 0 and index <= size.");
111         }
112         return data[index];
113     }
114
115     //修改制定位置的元素
116     public void set(int index,int e) {
117         //查歪了
118         if(index < 0 || index > size-1) {
119             throw new IllegalArgumentException("Add failed. Require index
120             >= 0 and index <= size.");
121         }
122         data[index]=e;
123     }
124
125     //查看数组是否包含元素
126     public boolean contains(int e) {
127         //从0角标开始往后遍历，直到size-1结束
128         for(int i=0;i<size;i++) {
129             if(data[i]==e) {

```

```

128         return true;
129     }
130 }
131 return false;
132 }
133 //获取索引角标
134 //如果不在范围内, 返回-1
135 public int getIndex(int e) {
136     for(int i=0;i<size;i++) {
137         if(data[i]==e) {
138             return i;
139         }
140     }
141     return -1;
142 }
143 // 从数组中删除index位置的元素, 返回删除的元素
144 public int remove(int index){
145     if(index < 0 || index >= size)
146         throw new IllegalArgumentException("Remove failed. Index is
illegal.");
147
148     int ret = data[index];
149     for(int i = index + 1 ; i <= size ; i ++ )
150         data[i - 1] = data[i];
151     size --;
152 //     1 2 3 4 5 0 0 0 size:5 length:8
153 //     1 2 3 4 0 0 0 0 size:4 length:8
154 //     1 2 3 0         size:3 length:4
155     if(size==data.length/2-1) {
156         delResize();
157     }
158
159     return ret;
160 }
161
162 //扩容操作 re再一次 size大小 resize再一次确定大小
163 //2*data.length
164 public void resize(int newCapacity) {
165     int[] newData=new int[newCapacity];
166     //把原来少元素的数组赋值给长的数组
167     for(int i=0;i<data.length;i++) {
168         newData[i]=data[i];
169     }
170     data=newData;//数组名字指向的是整个数组的内存起始地址
171 }
172 //缩容操作
173 public void delResize() {
174     int[] newData=new int[data.length/2];
175     for(int i=0;i<newData.length;i++) {
176         newData[i]=data[i];
177     }
178     data=newData;
179 }
180
181 @Override
182 public String toString() {
183     return "MyArrayList [data=" + Arrays.toString(data) + "]";
184 }

```

```
185  
186 }  
187
```

## V1.6版本：使用泛型

```
1  public class Array<E> {  
2  
3      private E[] data;  
4      private int size;  
5  
6      // 构造函数，传入数组的容量capacity构造Array  
7      public Array(int capacity){  
8          data = (E[])new Object[capacity];  
9          size = 0;  
10     }  
11  
12     // 无参数的构造函数，默认数组的容量capacity=10  
13     public Array(){  
14         this(10);  
15     }  
16  
17     // 获取数组的容量  
18     public int getCapacity(){  
19         return data.length;  
20     }  
21  
22     // 获取数组中的元素个数  
23     public int getSize(){  
24         return size;  
25     }  
26  
27     // 返回数组是否为空  
28     public boolean isEmpty(){  
29         return size == 0;  
30     }  
31  
32     // 在index索引的位置插入一个新元素e  
33     public void add(int index, E e){  
34  
35         if(index < 0 || index > size)  
36             throw new IllegalArgumentException("Add failed. Require index  
37             >= 0 and index <= size.");  
38  
39         if(size == data.length)  
40             resize(2 * data.length);  
41  
42         for(int i = size - 1; i >= index ; i --)  
43             data[i + 1] = data[i];  
44  
45         data[index] = e;  
46  
47         size ++;  
48     }  
49  
50     // 向所有元素后添加一个新元素  
51     public void addLast(E e){
```

```
51         add(size, e);
52     }
53
54     // 在所有元素前添加一个新元素
55     public void addFirst(E e){
56         add(0, e);
57     }
58
59     // 获取index索引位置的元素
60     public E get(int index){
61         if(index < 0 || index >= size)
62             throw new IllegalArgumentException("Get failed. Index is
illegal.");
63         return data[index];
64     }
65
66     // 修改index索引位置的元素为e
67     public void set(int index, E e){
68         if(index < 0 || index >= size)
69             throw new IllegalArgumentException("Set failed. Index is
illegal.");
70         data[index] = e;
71     }
72
73     // 查找数组中是否有元素e
74     public boolean contains(E e){
75         for(int i = 0 ; i < size ; i ++){
76             if(data[i].equals(e))
77                 return true;
78         }
79         return false;
80     }
81
82     // 查找数组中元素e所在的索引，如果不存在元素e，则返回-1
83     public int find(E e){
84         for(int i = 0 ; i < size ; i ++){
85             if(data[i].equals(e))
86                 return i;
87         }
88         return -1;
89     }
90
91     // 从数组中删除index位置的元素，返回删除的元素
92     public E remove(int index){
93         if(index < 0 || index >= size)
94             throw new IllegalArgumentException("Remove failed. Index is
illegal.");
95
96         E ret = data[index];
97         for(int i = index + 1 ; i < size ; i ++){
98             data[i - 1] = data[i];
99         }
100         size --;
101         data[size] = null; // loitering objects != memory leak
102
103         if(size == data.length / 2)
104             resize(data.length / 2);
105         return ret;
106     }
```

```

106
107 // 从数组中删除第一个元素，返回删除的元素
108 public E removeFirst(){
109     return remove(0);
110 }
111
112 // 从数组中删除最后一个元素，返回删除的元素
113 public E removeLast(){
114     return remove(size - 1);
115 }
116
117 // 从数组中删除元素e
118 public void removeElement(E e){
119     int index = find(e);
120     if(index != -1)
121         remove(index);
122 }
123
124 @Override
125 public String toString(){
126
127     StringBuilder res = new StringBuilder();
128     res.append(String.format("Array: size = %d , capacity = %d\n",
size, data.length));
129     res.append('[');
130     for(int i = 0 ; i < size ; i ++){
131         res.append(data[i]);
132         if(i != size - 1)
133             res.append(", ");
134     }
135     res.append(']');
136     return res.toString();
137 }
138
139 // 将数组空间的容量变成newCapacity大小
140 private void resize(int newCapacity){
141
142     E[] newData = (E[])new Object[newCapacity];
143     for(int i = 0 ; i < size ; i ++){
144         newData[i] = data[i];
145     }
146     data = newData;
147 }
148

```

## 数组时间复杂度算法简单分析

### 1， 简单复杂度分析

通过时间复杂度分析出算法的性能如何

时间复杂度通常是如下表示的

$O(1)$  ,  $O(n)$  ,  $O(\lg n)$  ,  $O(n \lg n)$  ,  $O(n^2)$

读作大O1 大On大O nlogn 大On平方

这里的这个大O就是描述的算法的运行时间和输入数据之间的关系

什么是运行时间和输入数据之间的关系呢？通过下面例子来演示

```
public static int sum(int[] nums){  
    int sum = 0;  
    for(int num: nums) sum += num;  
    return sum;  
}
```

$O(n)$   
 $n$ 是nums中的元素个数  
算法和 $n$ 呈线性关系

也就是说这里产生了 $n$ 个数，那么 $n$ 的数量是多少，对应的时间也就线性增加，但其实，每个 $n$ 并非和时间系数为1的。

比如：操作每一个数（temp），需要从这个这个数组中通过for循环取出来，然后需要取出sum并与temp加在一起重新再赋值给sum，对于每个数其实都是需要这么多操作的，那么这样的操作所花费的时间系数，我们称之为 $C1$ ，那么在开始计算之前可能还需要赋值sum=0，完成计算后还要返回这个sum，这些每次都有的操作所花费的等同的时间叫做 $c2$

· 为什么要用大O，叫做 $O(n)$ ？ 忽略常数。实际时间  $T = c1*n + c2$

为什么要忽略这个 $c1$ ， $c2$ 呢，因为拿这里的 $c1$ 来说，就算是直接使用，基于不同的语言，执行时间段也是不同的，就算是执行时间相同，底层的操作系统的汇编层面或者机器语言所花费的解析时间也不同，而且不同的cpu也是不同的，因此 $c2$ 也是同理。接下来看一组结论和案例对比：

$T = 2*n + 2$	$O(n)$
$T = 2000*n + 10000$	$O(n)$
$T = 1*n*n + 0$	$O(n^2)$

$T = 2*n + 2$	$O(n)$	
$T = 2000*n + 10000$	$O(n)$	渐进时间复杂度
$T = 1*n*n + 0$	$O(n^2)$	描述 $n$ 趋近于无穷的情况

这里的时间复杂度描述的不是临界值，而是 $n$ 趋近于无穷时候，这个算法谁块谁慢，同理，在这种情况下，低阶项实际上也很小，可以看做也是一个常数，忽略不计

$T = 2*n*n + 300n + 10 \quad O(n^2)$

分析自定义数组的各项操作

• 添加操作  $O(n)$

addLast(e)	$O(1)$	}	$O(n)$ 最坏情况	resize	$O(n)$
addFirst(e)	$O(n)$				
add(index, e)	$O(n/2) = O(n)$				

严格计算需要一些概率论知识

针对删除操作，删除1个跟删除n个平均来看，就是 $n/2$ ， $1/2$ 也是一个常数系数，舍去系数 $O(n)$

删除操作  $O(n)$

removeLast(e)	$O(1)$	}	$O(n)$	resize	$O(n)$
removeFirst(e)	$O(n)$				
remove(index, e)	$O(n/2) = O(n)$				

修改操作

set(index, e)  $O(1)$

• 查找操作

get(index)  $O(1)$

contains(e)  $O(n)$

find(e)  $O(n)$

结论：

- 增：  $O(n)$
  - 删：  $O(n)$
  - 改： 已知索引  $O(1)$ ；未知索引  $O(n)$
  - 查： 已知索引  $O(1)$ ；未知索引  $O(n)$
- 如果只对最后一个元素操作  
依然是 $O(n)$ ? 因为resize?

问题：删除和增加的分析完全使用最坏时间复杂度来分析是不合理的，因为并不是所有的操作都会触发这个容积的扩容

## 2，均摊复杂度和防止复杂度的震荡

分析增加操作中触发resize操作的条件

添加操作	$O(n)$	
<b>addLast(e)</b>	<b><math>O(1)</math></b>	$\left. \begin{array}{l} O(n) \\ \text{最坏情况} \end{array} \right\} \text{resize } O(n)$
addFirst(e)	$O(n)$	
add(index, e)	$O(n/2) = O(n)$	

分析：假如一个数组的capacity是10个元素，那么添加10个元素才可能会触发一次resize，此时触发resize之后数组的容量就会变成20，此时再添加10个才会再次触发这个resize，这个时候会变成capacity为40，也就是再添加20个数，才会触发resize，也就是说不会是每次添加一个元素都会触发resize，而我们却一直用最坏时间复杂度分析，这样是不合理的

再次深入分析，案例如下：

在resize之前，所有的操作都是 $O(1)$ 级别，而在addlast为第9个时候需要扩容，那么第九次等于for循环所有数组里的值进入新数组的时间和，再加1次add操作

假设当前capacity = 8，并且每一次添加操作都使用addLast

1    1    1    1    1    1    1    1    8 + 1

9次addLast操作，触发resize，总共进行了17次基本操作

所以，对于addLast来说，9次操作，平均来讲，每次的操作接近2次基本操作



结论

**resize  $O(n)$**

9次addLast操作，触发resize，总共进行了17次基本操作

平均，每次addLast操作，进行2次基本操作

假设capacity = n，n+1次addLast，触发resize，总共进行2n+1次基本操作

平均，每次addLast操作，进行2次基本操作

这样均摊计算，时间复杂度是 $O(1)$ 级别，在这样的例子里，这样均摊计算比计算最坏情况有意义

**resize  $O(n)$**

**addLast 的均摊复杂度为 $O(1)$**

按照这样理解removeLast的均摊时间复杂度也是 $O(1)$ 级别

但是这样会引发下一个问题

### 3，复杂度震荡

初始条件

但是，当我们同时看addLast和removeLast操作：



capacity = n

这个时候增加一个元素，触发的是扩容操作，加一个元素



addLast  $O(n)$

马上又进行removeLast的操作，此时又会触发缩容的操作，再次调用resize，时间复杂度依然是 $O(n)$



addLast  $O(n)$

removeLast  $O(n)$

capacity = n

一直这样循环呢？

	addLast	$O(n)$
capacity = n	removeLast	$O(n)$
	addLast	$O(n)$
	removeLast	$O(n)$

解决办法

出现问题的原因：removeLast 时 resize 过于着急（Eager）

解决方案：Lazy



扩容没有办法减少时间复杂度的增加，但是缩小的时候，并不着急把扩到二倍的数组容量减少为原来的1倍，同理这个时候要扩容也不用再次 $O(n)$ 的addLast操作而是 $O(1)$



当全部元素只剩下原来的4分1，也就是说

当  $size == capacity / 4$  时，才将capacity减半

```

1  public class Array<E> {
2      private E[] data;
3      private int size;
4      // 构造函数，传入数组的容量capacity构造Array
5      public Array(int capacity){
6          data = (E[])new Object[capacity];
7          size = 0;
8      }
9
10     // 无参数的构造函数，默认数组的容量capacity=10
11     public Array(){
12         this(10);
13     }
14
15     // 获取数组的容量
16     public int getCapacity(){
17         return data.length;
18     }
19
20     // 获取数组中的元素个数
21     public int getSize(){
22         return size;
23     }

```

```
24
25 // 返回数组是否为空
26 public boolean isEmpty(){
27     return size == 0;
28 }
29
30 // 在index索引的位置插入一个新元素e
31 public void add(int index, E e){
32
33     if(index < 0 || index > size)
34         throw new IllegalArgumentException("Add failed. Require index
35         >= 0 and index <= size.");
36
37     if(size == data.length)
38         resize(2 * data.length);
39
40     for(int i = size - 1; i >= index ; i --)
41         data[i + 1] = data[i];
42
43     data[index] = e;
44
45     size ++;
46 }
47
48 // 向所有元素后添加一个新元素
49 public void addLast(E e){
50     add(size, e);
51 }
52
53 // 在所有元素前添加一个新元素
54 public void addFirst(E e){
55     add(0, e);
56 }
57
58 // 获取index索引位置的元素
59 public E get(int index){
60     if(index < 0 || index >= size)
61         throw new IllegalArgumentException("Get failed. Index is
62         illegal.");
63     return data[index];
64 }
65
66 // 修改index索引位置的元素为e
67 public void set(int index, E e){
68     if(index < 0 || index >= size)
69         throw new IllegalArgumentException("Set failed. Index is
70         illegal.");
71     data[index] = e;
72 }
73
74 // 查找数组中是否有元素e
75 public boolean contains(E e){
76     for(int i = 0 ; i < size ; i ++){
77         if(data[i].equals(e))
78             return true;
79     }
80     return false;
81 }
```

```

79
80 // 查找数组中元素e所在的索引，如果不存在元素e，则返回-1
81 public int find(E e){
82     for(int i = 0 ; i < size ; i++){
83         if(data[i].equals(e))
84             return i;
85     }
86     return -1;
87 }
88
89 // 从数组中删除index位置的元素，返回删除的元素
90 public E remove(int index){
91     if(index < 0 || index >= size)
92         throw new IllegalArgumentException("Remove failed. Index is
illegal.");
93
94     E ret = data[index];
95     for(int i = index + 1 ; i < size ; i++){
96         data[i - 1] = data[i];
97     }
98     size--;
99     data[size] = null; // loitering objects != memory leak
100 //对于动态数组来说，不能够在扩容的时候让他的值等于0
101     if(size == data.length / 4 && data.length / 2 != 0)
102         resize(data.length / 2);
103     return ret;
104 }
105
106 // 从数组中删除第一个元素，返回删除的元素
107 public E removeFirst(){
108     return remove(0);
109 }
110
111 // 从数组中删除最后一个元素，返回删除的元素
112 public E removeLast(){
113     return remove(size - 1);
114 }
115
116 // 从数组中删除元素e
117 public void removeElement(E e){
118     int index = find(e);
119     if(index != -1)
120         remove(index);
121 }
122
123 @Override
124 public String toString(){
125
126     StringBuilder res = new StringBuilder();
127     res.append(String.format("Array: size = %d , capacity = %d\n",
size, data.length));
128     res.append('[');
129     for(int i = 0 ; i < size ; i++){
130         res.append(data[i]);
131         if(i != size - 1)
132             res.append(", ");
133     }
134     res.append(']');
135     return res.toString();

```

```

135     }
136
137     // 将数组空间的容量变成newCapacity大小
138     private void resize(int newCapacity){
139
140         E[] newData = (E[])new Object[newCapacity];
141         for(int i = 0 ; i < size ; i ++){
142             newData[i] = data[i];
143         }
144         data = newData;
145     }
146

```

## 栈 (stack)

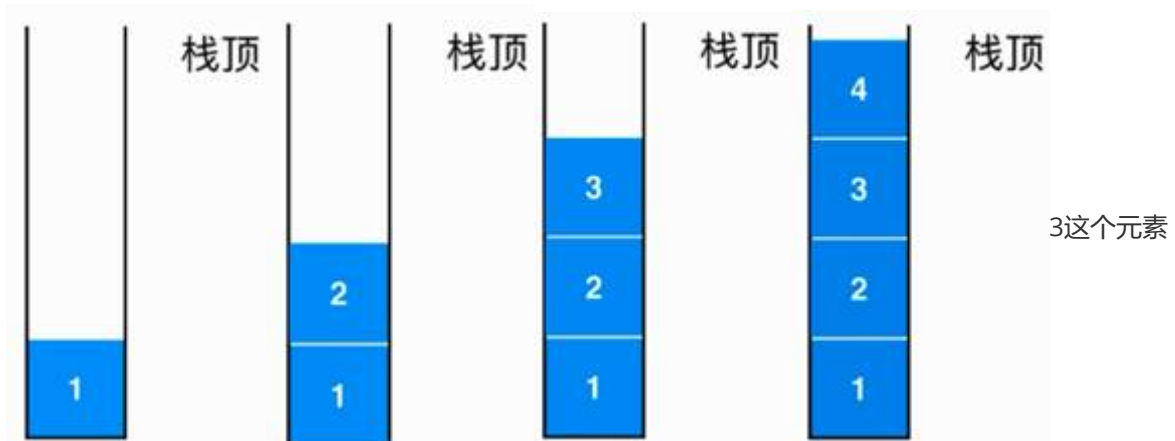
Stack是一种线性结构

相比数组，栈对应的操作是数组的子集，而且他的操作更少

只能从唯一的一端添加元素，也只能从这个唯一的一端取出元素

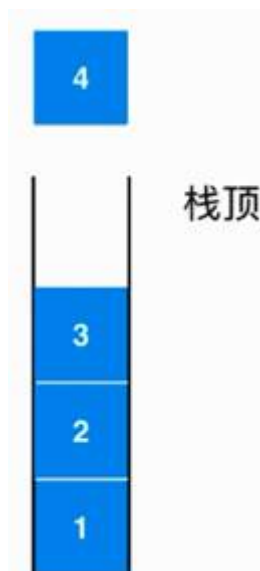
这个唯一的一端称为栈顶

### 展示元素入栈的过程



只能在这个位置，不可以插入2和1元素之间

### 展示元素出栈的过程



## 小结:

- 1, 栈是一种后进先出的数据结构
- 2, Last In First Out(LIFO)
- 3, Stack在计算机的运用里拥有不可思议的作用

### 案例1

沉迷学习无法自拔

无处不在的Undo操作（撤销）

沉迷 学习 不法



最直接的一个案例，比如word中的文字撤销操作，比如以下的撤销不法，当这个不法两个字被撤销后，不需要再保留他们，直接出栈

无处不在的Undo操作（撤销）

沉迷 学习



输入正确的内容和顺序

无处不在的Undo操作（撤销）

沉迷 学习 无法 自拔



## 案例2

程序调用的系统栈

• 程序调用的系统栈

func A(){	func B(){	func C(){
1 ...	1 ...	1 ...
2 B()	2 C()	2 ...
3 ...	3 ...	3 ...
}	}	}

栈顶

首先执行A这个函数

→ func A(){  
1 ...  
2 B()  
3 ...  
}

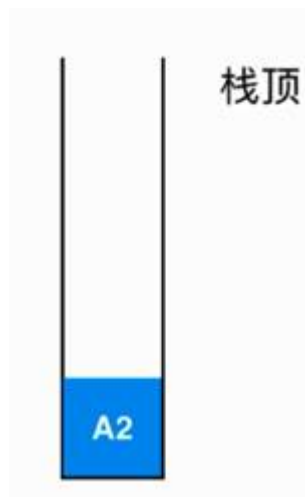
顺序执行1,2,3, 行, 当在执行到第二行的时候

func A(){  
1 ...  
→ 2 B()  
3 ...  
}

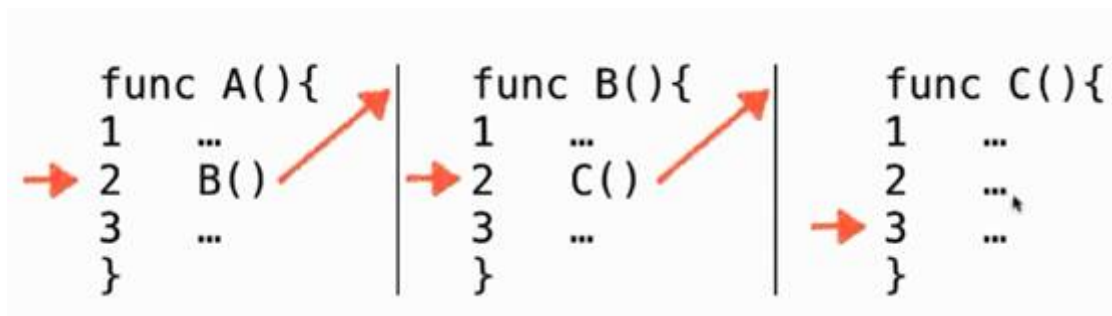
会跳去执行B这个函数, a函数会暂时中断

func A(){	func B(){
1 ...	1 ...
→ 2 B()	2 C()
3 ...	3 ...
}	}

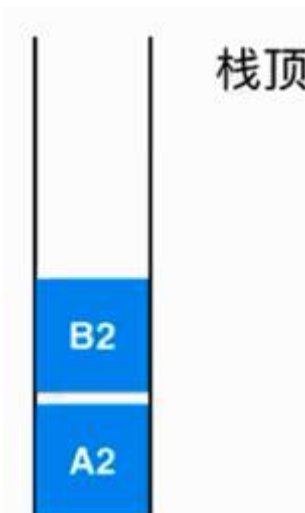
此时可以在系统栈中认为, A函数执行到了第二行, 记为A2



然后继续执行



同理



最后C函数在执行完成之后，就会回到B2继续执行，然后执行完成，B2出栈，剩下一个A2，继续执行A函数的内容最后A2出栈函数全部执行完毕

## 栈的基本实现



Stack<E>

- void push(E)
- E pop()
- E peek()
- int getSize()
- boolean isEmpty()

栈的实现有很多种数据结构的方式，数组实现只是其中一种

Stack<E>

- void push(E)
  - E pop()
  - E peek()
  - int getSize()
  - boolean isEmpty()
- 从用户的角度看，支持这些操作就好
  - 具体底层实现，用户不关心
  - 实际底层有多种实现方式

栈的实现结构

Interface Stack<E>

- void push(E)
- E pop()
- E peek()
- int getSize()
- boolean isEmpty()

← - - - - - ArrayStack<E>

implement

对应的array的数组，增加部分加粗标红

# Array

```
1
2 public class Array<E> {
3
4     private E[] data;
5     private int size;
6
7     // 构造函数，传入数组的容量capacity构造Array
8     public Array(int capacity){
9         data = (E[])new Object[capacity];
10        size = 0;
11    }
12
13    // 无参数的构造函数，默认数组的容量capacity=10
14    public Array(){
15        this(10);
16    }
17
18    // 获取数组的容量
19    public int getCapacity(){
20        return data.length;
21    }
22
23    // 获取数组中的元素个数
24    public int getSize(){
25        return size;
26    }
27
28    // 返回数组是否为空
29    public boolean isEmpty(){
30        return size == 0;
31    }
32
33    // 在index索引的位置插入一个新元素e
34    public void add(int index, E e){
35
36        if(index < 0 || index > size)
37            throw new IllegalArgumentException("Add failed. Require index
38            >= 0 and index <= size.");
39
40        if(size == data.length)
41            resize(2 * data.length);
42
43        for(int i = size - 1; i >= index ; i --)
44            data[i + 1] = data[i];
45
46        data[index] = e;
47
48        size ++;
49    }
50
51    // 向所有元素后添加一个新元素
52    public void addLast(E e){
53        add(size, e);
54    }
```

```
55 // 在所有元素前添加一个新元素
56 public void addFirst(E e){
57     add(0, e);
58 }
59
60 // 获取index索引位置的元素
61 public E get(int index){
62     if(index < 0 || index >= size)
63         throw new IllegalArgumentException("Get failed. Index is
illegal.");
64     return data[index];
65 }
66
67 public E getLast(){
68     return get(size - 1);
69 }
70
71 public E getFirst(){
72     return get(0);
73 }
74
75 // 修改index索引位置的元素为e
76 public void set(int index, E e){
77     if(index < 0 || index >= size)
78         throw new IllegalArgumentException("Set failed. Index is
illegal.");
79     data[index] = e;
80 }
81
82 // 查找数组中是否有元素e
83 public boolean contains(E e){
84     for(int i = 0 ; i < size ; i ++){
85         if(data[i].equals(e))
86             return true;
87     }
88     return false;
89 }
90
91 // 查找数组中元素e所在的索引, 如果不存在元素e, 则返回-1
92 public int find(E e){
93     for(int i = 0 ; i < size ; i ++){
94         if(data[i].equals(e))
95             return i;
96     }
97     return -1;
98 }
99
100 // 从数组中删除index位置的元素, 返回删除的元素
101 public E remove(int index){
102     if(index < 0 || index >= size)
103         throw new IllegalArgumentException("Remove failed. Index is
illegal.");
104
105     E ret = data[index];
106     for(int i = index + 1 ; i < size ; i ++){
107         data[i - 1] = data[i];
108     }
109     size --;
110     data[size] = null; // loitering objects != memory leak
```

```

110
111         if(size == data.length / 4 && data.length / 2 != 0)
112             resize(data.length / 2);
113         return ret;
114     }
115
116     // 从数组中删除第一个元素，返回删除的元素
117     public E removeFirst(){
118         return remove(0);
119     }
120
121     // 从数组中删除最后一个元素，返回删除的元素
122     public E removeLast(){
123         return remove(size - 1);
124     }
125
126     // 从数组中删除元素e
127     public void removeElement(E e){
128         int index = find(e);
129         if(index != -1)
130             remove(index);
131     }
132
133     @Override
134     public String toString(){
135
136         StringBuilder res = new StringBuilder();
137         res.append(String.format("Array: size = %d , capacity = %d\n",
size, data.length));
138         res.append('[');
139         for(int i = 0 ; i < size ; i ++){
140             res.append(data[i]);
141             if(i != size - 1)
142                 res.append(", ");
143         }
144         res.append(']');
145         return res.toString();
146     }
147
148     // 将数组空间的容量变成newCapacity大小
149     private void resize(int newCapacity){
150
151         E[] newData = (E[])new Object[newCapacity];
152         for(int i = 0 ; i < size ; i ++){
153             newData[i] = data[i];
154         }
155         data = newData;
156     }
157

```

## Stack

```
1 public interface Stack<E> {
2     int getSize();
3     boolean isEmpty();
4     void push(E e);
5     E pop();
6     E peek();
7 }
8
```

## ArrayStack

```
1 public class ArrayStack<E> implements Stack<E> {
2
3     private Array<E> array;
4
5     public ArrayStack(int capacity){
6         array = new Array<>(capacity);
7     }
8
9     public ArrayStack(){
10         array = new Array<>();
11     }
12
13     @Override
14     public int getSize(){
15         return array.getSize();
16     }
17
18     @Override
19     public boolean isEmpty(){
20         return array.isEmpty();
21     }
22
23     public int getCapacity(){
24         return array.getCapacity();
25     }
26
27     @Override
28     public void push(E e){
29         array.addLast(e);
30     }
31
32     @Override
33     public E pop(){
34         return array.removeLast();
35     }
36
37     @Override
38     public E peek(){
39         return array.getLast();
40     }
41
42     @Override
43     public String toString(){
44         StringBuilder res = new StringBuilder();
45         res.append("Stack: ");
46     }
47 }
```

```

46         res.append('[');
47         for(int i = 0 ; i < array.getSize() ; i ++){
48             res.append(array.get(i));
49             if(i != array.getSize() - 1)
50                 res.append(", ");
51         }
52         res.append("] top");
53         return res.toString();
54     }
55 }
56

```

## Main

```

1  public class Main {
2
3      public static void main(String[] args) {
4
5          ArrayStack<Integer> stack = new ArrayStack<>();
6
7          for(int i = 0 ; i < 5 ; i ++){
8              stack.push(i);
9              System.out.println(stack);
10         }
11
12         stack.pop();
13         System.out.println(stack);
14     }
15 }
16

```

## 栈的时间复杂度分析

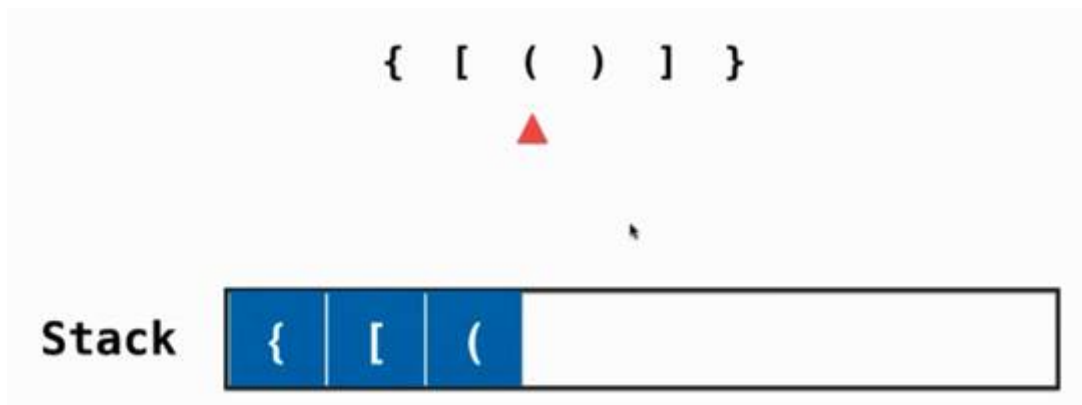
### Stack案例实操（编译器对括号的匹配报错机制）

给定一个只包括 '(' , ')' , '{' , '}' , '[' , ']' 的字符串，判断字符串是否有效。  
 括号必须以正确的顺序关闭，"()" 和 "()[]{}" 是有效的但是 "(]" 和 "([)]" 不是。

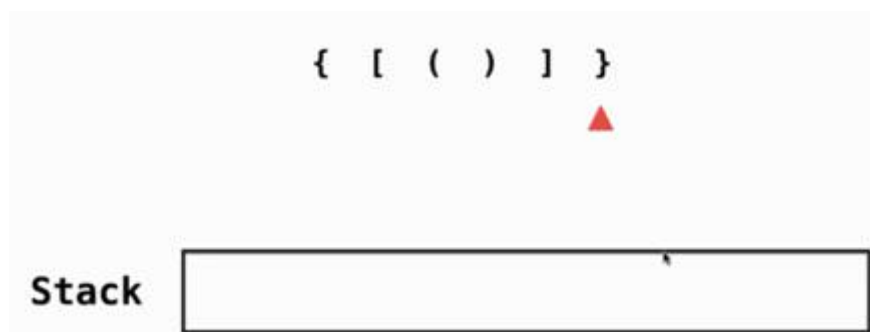
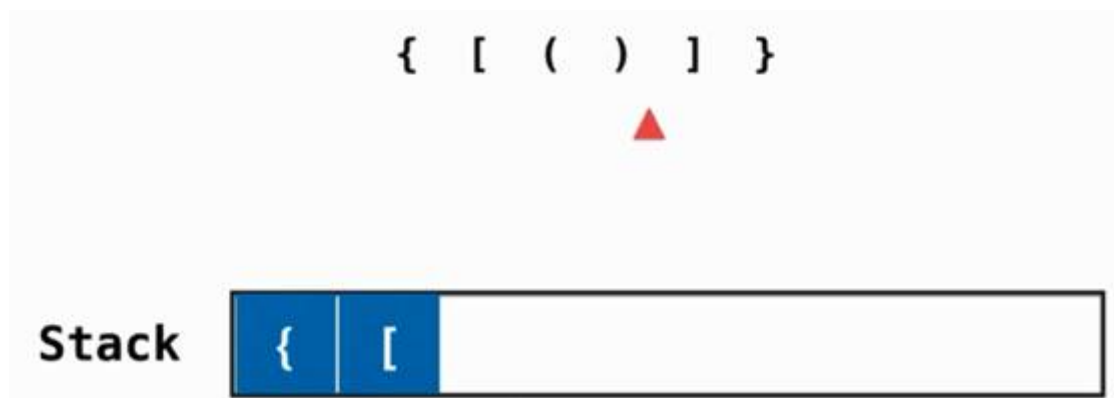
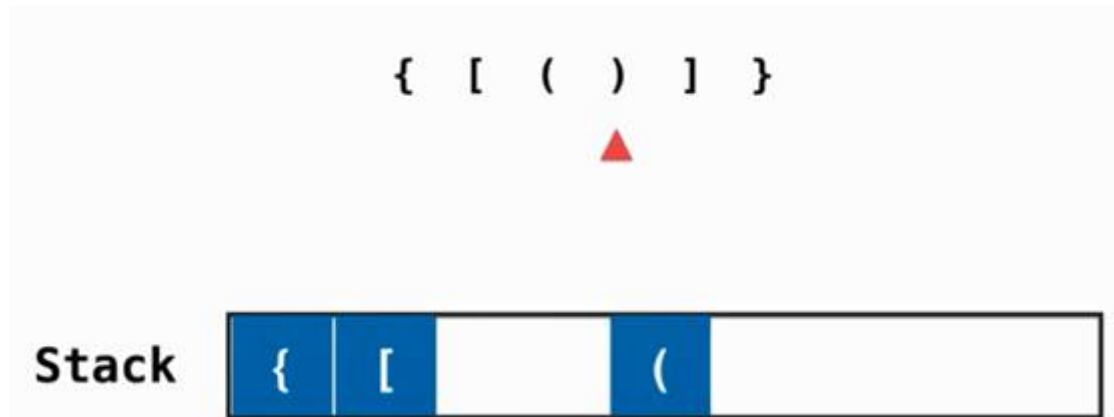
分析思路



逐一加入栈结构



当开始匹配右括号的时候，需要看当前栈顶是否是和他匹配的，如果匹配那么当前栈顶的括号就可以出栈了



当全部出栈，那么当前的匹配就是一个合法的字符串

代码实现

```
1 import java.util.Stack;
2
3 class Solution {
4
5     public boolean isValid(String s) {
```

```

6
7     Stack<Character> stack = new Stack<>();
8     for(int i = 0 ; i < s.length() ; i ++){
9         char c = s.charAt(i);
10        if(c == '(' || c == '[' || c == '{')
11            stack.push(c);
12        else{
13            if(stack.isEmpty())
14                return false;

15
16            char topChar = stack.pop();
17            if(c == ')' && topChar != '(')
18                return false;
19            if(c == ']' && topChar != '[')
20                return false;
21            if(c == '}' && topChar != '{')
22                return false;
23        }
24    }
25    return stack.isEmpty();
26 }
27
28 public static void main(String[] args) {
29
30     System.out.println((new Solution()).isValid("()[]{}"));
31     System.out.println((new Solution()).isValid("([])"));
32 }
33 }
34

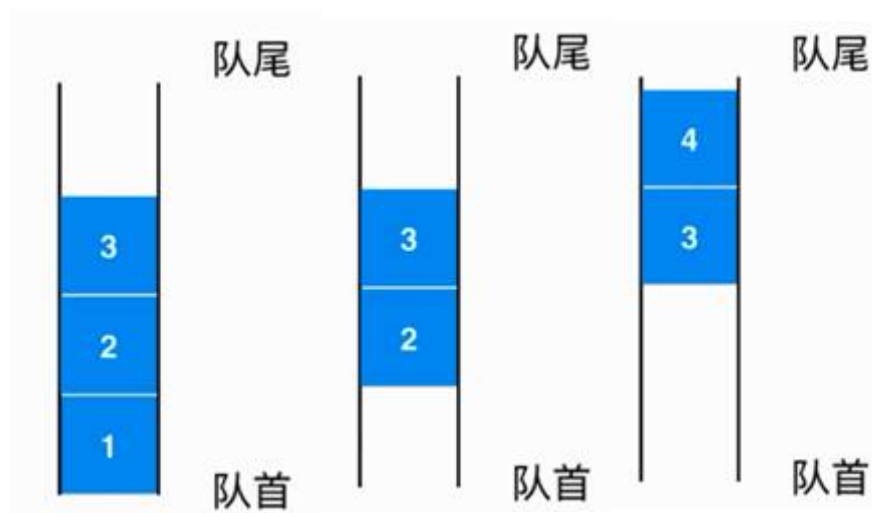
```

## 队列

### 定义:

队列是一种线性结构

只能从一端（队尾）添加元素，只能从另一端（队首）取出元素



结论：队列是一种先进先出的数据结构（First in First out）



Queue<E>

- void enqueue(E)
- E dequeue()
- E getFront()
- int getSize()
- boolean isEmpty()

队列的实现

```
Interface Queue<E> ◀----- ArrayQueue<E>
• void enqueue(E)    implement
• E dequeue()
• E getFront()
• int getSize()
• boolean isEmpty()
```

队列时间复杂度分析

ArrayQueue<E>

- void enqueue(E)       $O(1)$  均摊
- E dequeue()       $O(n)$
- E front()       $O(1)$
- int getSize()       $O(1)$
- boolean isEmpty()       $O(1)$

## Array

```
1
2 public class Array<E> {
```

```
3
4     private E[] data;
5     private int size;
6
7     // 构造函数，传入数组的容量capacity构造Array
8     public Array(int capacity){
9         data = (E[])new Object[capacity];
10        size = 0;
11    }
12
13    // 无参数的构造函数，默认数组的容量capacity=10
14    public Array(){
15        this(10);
16    }
17
18    // 获取数组的容量
19    public int getCapacity(){
20        return data.length;
21    }
22
23    // 获取数组中的元素个数
24    public int getSize(){
25        return size;
26    }
27
28    // 返回数组是否为空
29    public boolean isEmpty(){
30        return size == 0;
31    }
32
33    // 在index索引的位置插入一个新元素e
34    public void add(int index, E e){
35
36        if(index < 0 || index > size)
37            throw new IllegalArgumentException("Add failed. Require index
38            >= 0 and index <= size.");
39
40        if(size == data.length)
41            resize(2 * data.length);
42
43        for(int i = size - 1; i >= index ; i --)
44            data[i + 1] = data[i];
45
46        data[index] = e;
47
48        size ++;
49    }
50
51    // 向所有元素后添加一个新元素
52    public void addLast(E e){
53        add(size, e);
54    }
55
56    // 在所有元素前添加一个新元素
57    public void addFirst(E e){
58        add(0, e);
59    }
```

```

60 // 获取index索引位置的元素
61 public E get(int index){
62     if(index < 0 || index >= size)
63         throw new IllegalArgumentException("Get failed. Index is
illegal.");
64     return data[index];
65 }
66
67 public E getLast(){
68     return get(size - 1);
69 }
70
71 public E getFirst(){
72     return get(0);
73 }
74
75 // 修改index索引位置的元素为e
76 public void set(int index, E e){
77     if(index < 0 || index >= size)
78         throw new IllegalArgumentException("Set failed. Index is
illegal.");
79     data[index] = e;
80 }
81
82 // 查找数组中是否有元素e
83 public boolean contains(E e){
84     for(int i = 0 ; i < size ; i++){
85         if(data[i].equals(e))
86             return true;
87     }
88     return false;
89 }
90
91 // 查找数组中元素e所在的索引，如果不存在元素e，则返回-1
92 public int find(E e){
93     for(int i = 0 ; i < size ; i++){
94         if(data[i].equals(e))
95             return i;
96     }
97     return -1;
98 }
99
100 // 从数组中删除index位置的元素，返回删除的元素
101 public E remove(int index){
102     if(index < 0 || index >= size)
103         throw new IllegalArgumentException("Remove failed. Index is
illegal.");
104
105     E ret = data[index];
106     for(int i = index + 1 ; i < size ; i++){
107         data[i - 1] = data[i];
108     }
109     size--;
110     data[size] = null; // loitering objects != memory leak
111
112     if(size == data.length / 4 && data.length / 2 != 0)
113         resize(data.length / 2);
114     return ret;

```

```

115
116 // 从数组中删除第一个元素, 返回删除的元素
117 public E removeFirst(){
118     return remove(0);
119 }
120
121 // 从数组中删除最后一个元素, 返回删除的元素
122 public E removeLast(){
123     return remove(size - 1);
124 }
125
126 // 从数组中删除元素e
127 public void removeElement(E e){
128     int index = find(e);
129     if(index != -1)
130         remove(index);
131 }
132
133 @Override
134 public String toString(){
135
136     StringBuilder res = new StringBuilder();
137     res.append(String.format("Array: size = %d , capacity = %d\n",
size, data.length));
138     res.append('[');
139     for(int i = 0 ; i < size ; i ++){
140         res.append(data[i]);
141         if(i != size - 1)
142             res.append(", ");
143     }
144     res.append(']');
145     return res.toString();
146 }
147
148 // 将数组空间的容量变成newCapacity大小
149 private void resize(int newCapacity){
150
151     E[] newData = (E[])new Object[newCapacity];
152     for(int i = 0 ; i < size ; i ++){
153         newData[i] = data[i];
154     }
155     data = newData;
156 }
157

```

## ArrayQueue

```

1 public class ArrayQueue<E> implements Queue<E> {
2
3     private Array<E> array;
4
5     public ArrayQueue(int capacity){
6         array = new Array<>(capacity);
7     }
8
9     public ArrayQueue(){
10        array = new Array<>();

```

```

11     }
12
13     @Override
14     public int getSize(){
15         return array.getSize();
16     }
17
18     @Override
19     public boolean isEmpty(){
20         return array.isEmpty();
21     }
22
23     public int getCapacity(){
24         return array.getCapacity();
25     }
26
27     @Override
28     public void enqueue(E e){
29         array.addLast(e);
30     }
31
32     @Override
33     public E dequeue(){
34         return array.removeFirst();
35     }
36
37     @Override
38     public E getFront(){
39         return array.getFirst();
40     }
41
42     @Override
43     public String toString(){
44         StringBuilder res = new StringBuilder();
45         res.append("Queue: ");
46         res.append("front [");
47         for(int i = 0 ; i < array.getSize() ; i ++){
48             res.append(array.get(i));
49             if(i != array.getSize() - 1)
50                 res.append(", ");
51         }
52         res.append("] tail");
53         return res.toString();
54     }
55
56     public static void main(String[] args) {
57
58         ArrayQueue<Integer> queue = new ArrayQueue<>();
59         for(int i = 0 ; i < 10 ; i ++){
60             queue.enqueue(i);
61             System.out.println(queue);
62             if(i % 3 == 2){
63                 queue.dequeue();
64                 System.out.println(queue);
65             }
66         }
67     }
68 }

```

## Queue

```

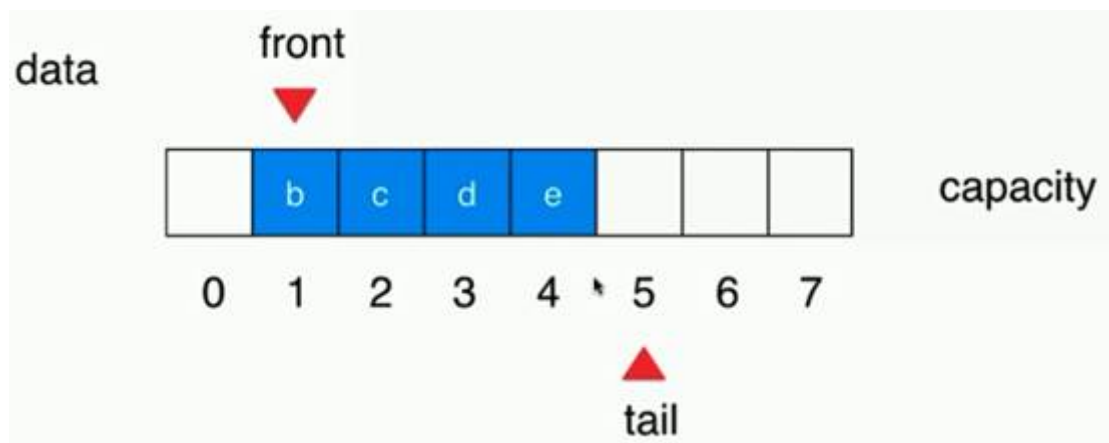
1 public interface Queue<E> {
2
3     int getSize();
4     boolean isEmpty();
5     void enqueue(E e);
6     E dequeue();
7     E getFront();
8 }
9

```

引出一个问题

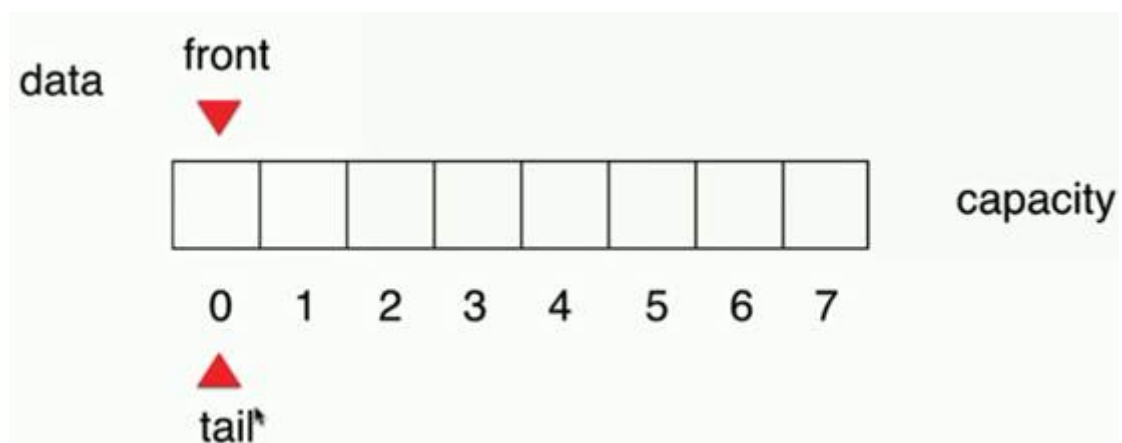
删除队首元素会引发 $O(n)$ 的操作

因此提出指向一个队首和队尾的指针

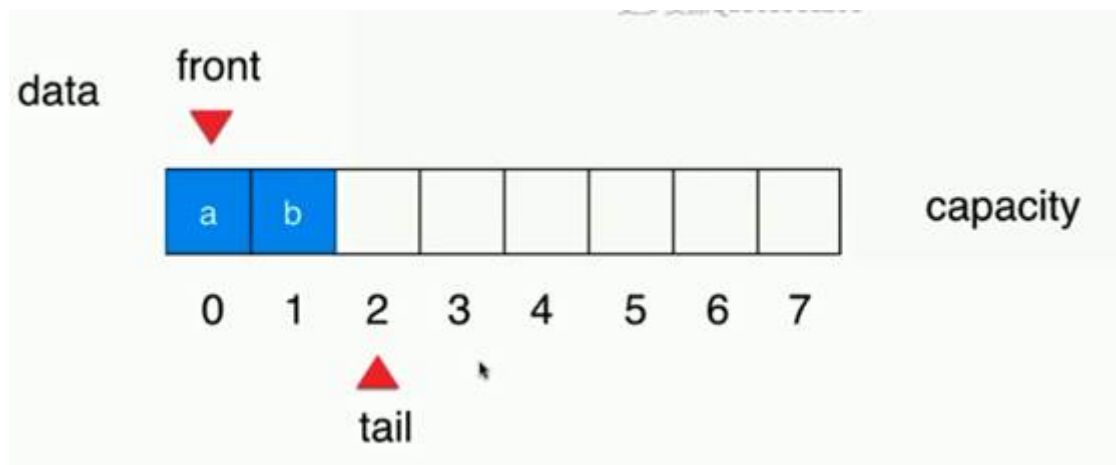


循环队列

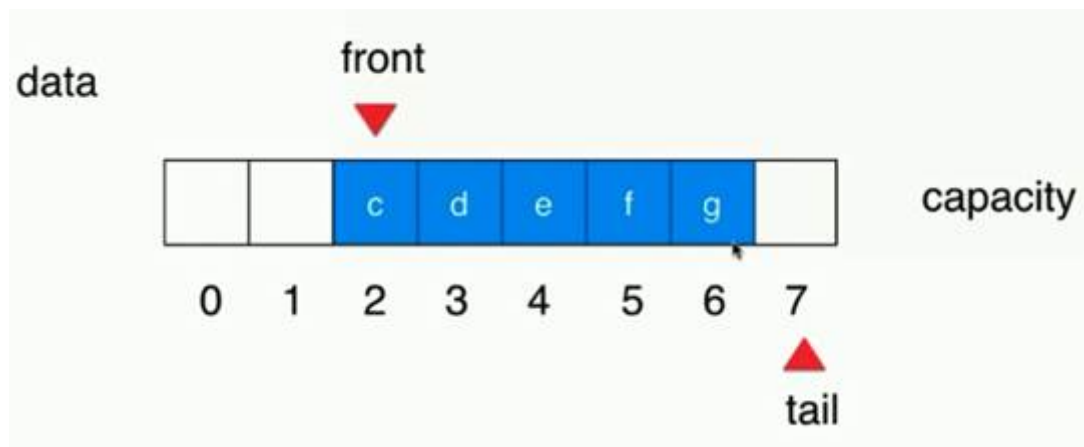
Front和tail相等的时候队列为空



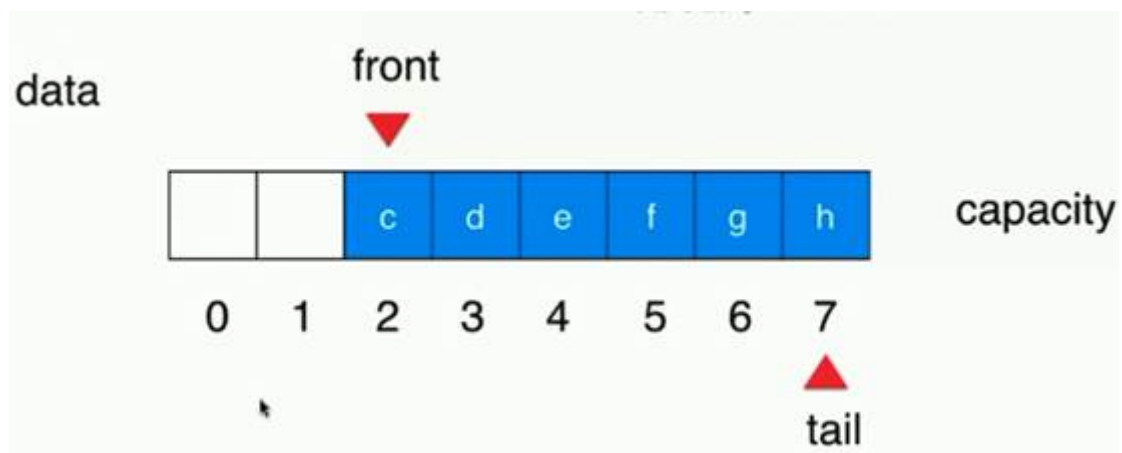
当往队首添加元素的时候



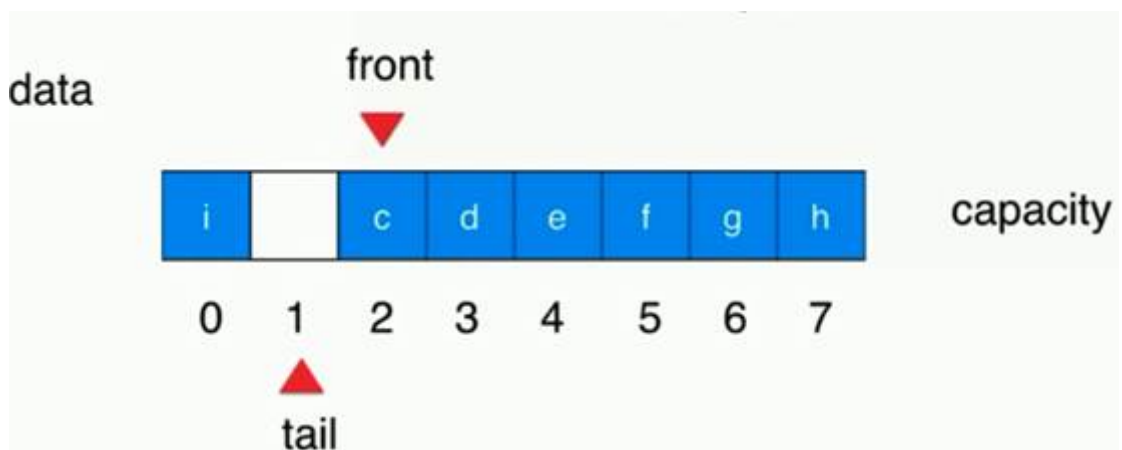
如果出现出队的同时又出现入队



当队尾出入队到容量的极限的时候，会先去看一下front之前有没有位置，



如果有空位



Tail会到回到容量的起始，再依次往后， $(tail+1) \% capacity(data.length)=front$ ——》队列满，整个循环队列的结构是有意识地浪费了一个空间

```
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];  
}
```

解释一下循环队列的代码：

Newdata0-front

Newdata1-front+1

真正的队列偏移是front+i的，由于又是循环队列，因此防止数组越界， $(front+i)\%data.length$

代码实现

```
1 public class LoopQueue<E> implements Queue<E> {  
2  
3     private E[] data;  
4     private int front, tail;  
5     private int size; // 有兴趣的同学，在完成这一章后，可以思考一下：  
6                         // LoopQueue中不声明size，如何完成所有的逻辑？  
7                         // 这个问题可能会比大家想象的要难一点点： )  
8  
9     public LoopQueue(int capacity){  
10         data = (E[])new Object[capacity + 1];  
11         front = 0;  
12         tail = 0;  
13         size = 0;  
14     }  
15  
16     public LoopQueue(){  
17         this(10);  
18     }  
19  
20     public int getCapacity(){  
21         return data.length - 1;  
22     }  
23  
24     @Override  
25     public boolean isEmpty(){  
26         return front == tail;  
27     }  
28  
29     @Override  
30     public int getSize(){  
31         return size;  
32     }  
33  
34     @Override  
35     public void enqueue(E e){  
36  
37         if((tail + 1) % data.length == front)  
38             resize(getCapacity() * 2);
```



```

39
40     data[tail] = e;
41     tail = (tail + 1) % data.length;
42     size ++;
43 }
44
45 @Override
46 public E dequeue(){
47
48     if(isEmpty())
49         throw new IllegalArgumentException("Cannot dequeue from an
empty queue.");
50
51     E ret = data[front];
52     data[front] = null;
53     front = (front + 1) % data.length;
54     size --;
55     if(size == getCapacity() / 4 && getCapacity() / 2 != 0)
56         resize(getCapacity() / 2);
57     return ret;
58 }
59
60 @Override
61 public E getFront(){
62     if(isEmpty())
63         throw new IllegalArgumentException("Queue is empty.");
64     return data[front];
65 }
66
67 private void resize(int newCapacity){
68
69     E[] newData = (E[])new Object[newCapacity + 1];
70     for(int i = 0 ; i < size ; i ++){
71         newData[i] = data[(i + front) % data.length];
72
73     }
74     data = newData;
75     front = 0;
76     tail = size;
77
78 }
79
80 @Override
81 public String toString(){
82
83     StringBuilder res = new StringBuilder();
84     res.append(String.format("Queue: size = %d , capacity = %d\n",
size, getCapacity()));
85     res.append("front ");
86     for(int i = front ; i != tail ; i = (i + 1) % data.length){
87         res.append(data[i]);
88         if((i + 1) % data.length != tail)
89             res.append(", ");
90     }
91     res.append("] tail");
92     return res.toString();
93 }
94
95 public static void main(String[] args){

```

```

95     LoopQueue<Integer> queue = new LoopQueue<>();
96     for(int i = 0 ; i < 10 ; i ++){
97         queue.enqueue(i);
98         System.out.println(queue);
99
100         if(i % 3 == 2){
101             queue.dequeue();
102             System.out.println(queue);
103         }
104     }
105 }
106 }
107

```

## 循环队列与数组队列的时间复杂度分析

LoopQueue<E>

- void enqueue(E)            0(1) 均摊
- E dequeue()                0(1) 均摊
- E getFront()               0(1)
- int getSize()              0(1)
- boolean isEmpty()         0(1)

```

1  import java.util.Random;
2
3  public class Main {
4
5      // 测试使用q运行opCount个enqueue和dequeue操作所需要的时间，单位：秒
6      private static double testQueue(LoopQueue<Integer> q, int opCount){
7
8          long startTime = System.nanoTime();
9
10         Random random = new Random();
11         for(int i = 0 ; i < opCount ; i ++){
12             q.enqueue(random.nextInt(Integer.MAX_VALUE));
13         }
14         for(int i = 0 ; i < opCount ; i ++){
15             q.dequeue();
16         }
17
18         long endTime = System.nanoTime();
19
20         return (endTime - startTime) / 1000000000.0;
21     }
22
23     public static void main(String[] args) {
24
25         int opCount = 100000;
26
27     }
28 }

```

```

24
25     ArrayQueue<Integer> arrayQueue = new ArrayQueue<>();
26     double time1 = testQueue(arrayQueue, opCount);
27     System.out.println("ArrayQueue, time: " + time1 + " s");
28
29     LoopQueue<Integer> loopQueue = new LoopQueue<>();
30     double time2 = testQueue(loopQueue, opCount);
31     System.out.println("LoopQueue, time: " + time2 + " s");
32 }
33 }
34

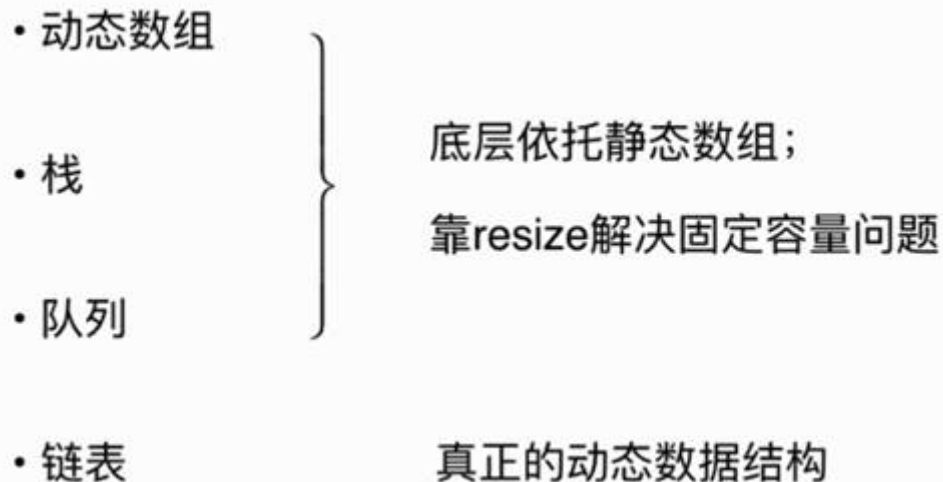
```

## 链表

前瞻课程：

内部类

链表是一种真正的动态数据结构



最简单的动态数据结构

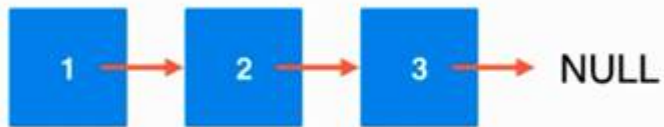
更深入的理解引用（或者指针）

更深入的理解递归

辅助组成其他数据结构

数据存储在“节点”（node）中

```
class Node {
    E e;
    Node next;
}
```



优点：真正的动态，不需要处理固定容量的问题,不需要跟动态数组一样，一下子new出来这么多的空间

缺点：丧失了随机访问的能力（无法如数组那样，根据索引查询元素，只能根据指向线索进行索引）

## 数组结构和链表结构的对比

数组最好用于索引有语意的情况。scores[2]

最大的优点：支持快速查询

链表不适合用于索引有语意的情况。

最大的优点：动态

动态链表基础结构

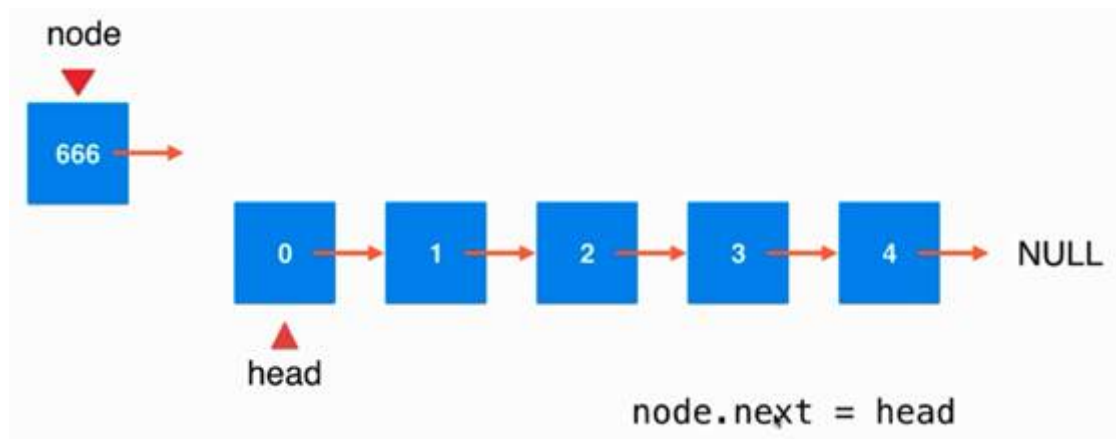
```

1 public class LinkedList<E> {
2     private class Node{
3         public E e;
4         public Node next;
5         public Node(E e, Node next){
6             this.e = e;
7             this.next = next;
8         }
9         public Node(E e){
10             this(e, null);
11         }
12         public Node(){
13             this(null, null);
14         }
15         @Override
16         public String toString(){
17             return e.toString();
18         }
19     }
20 }
21 
```

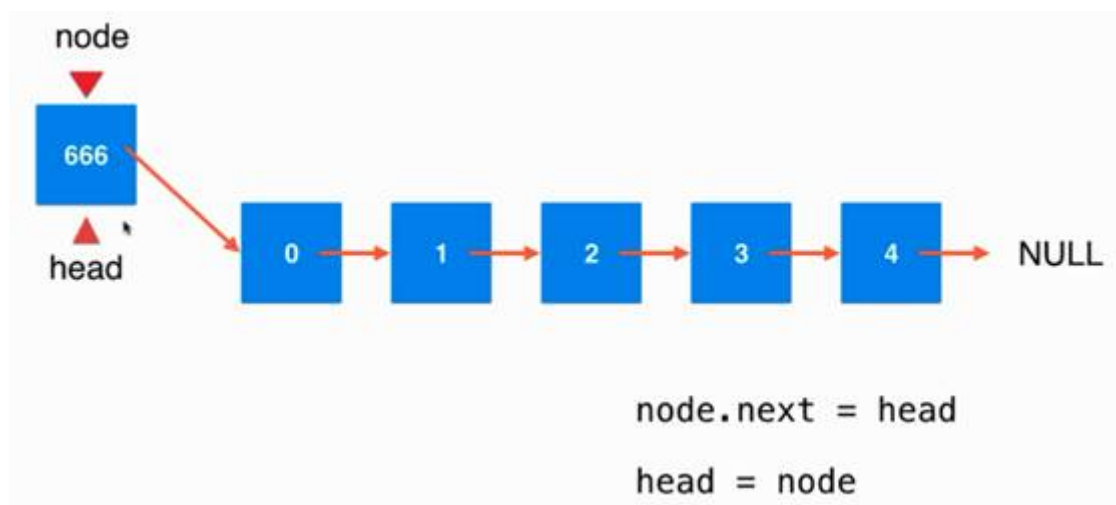
## 给链表中添加元素



## 在表头添加数据



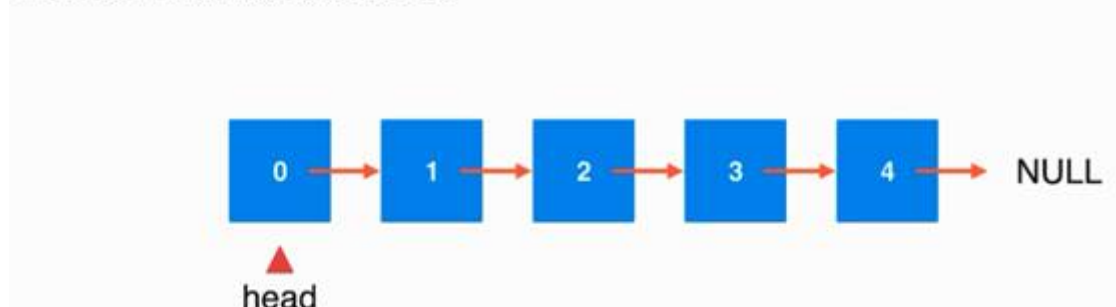
将head指向node



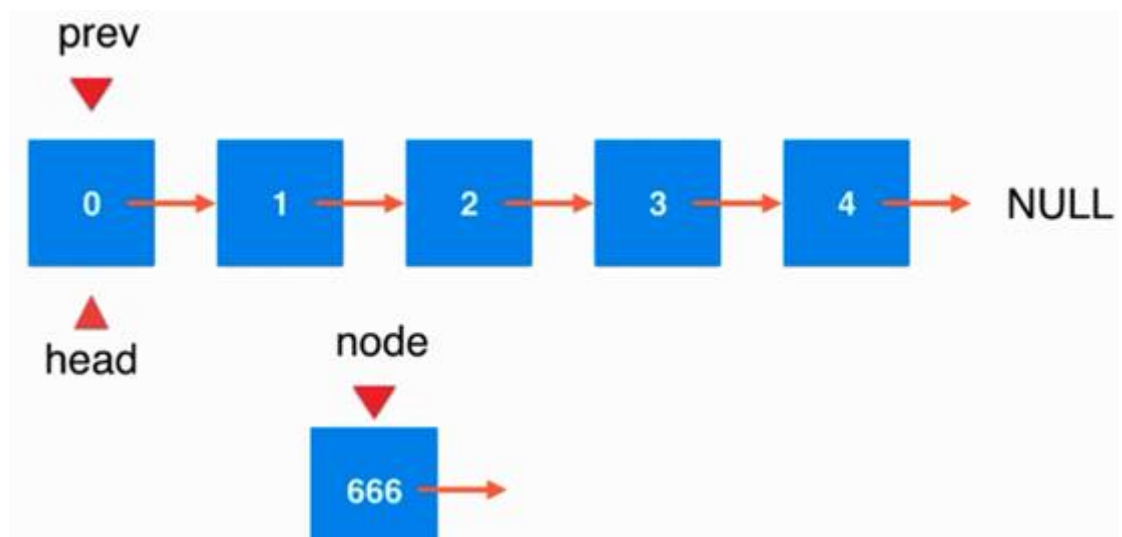
于是node就成为了该链表的head，进入链表中后，成为了链表的一部分

## 在链表指定索引处添加节点

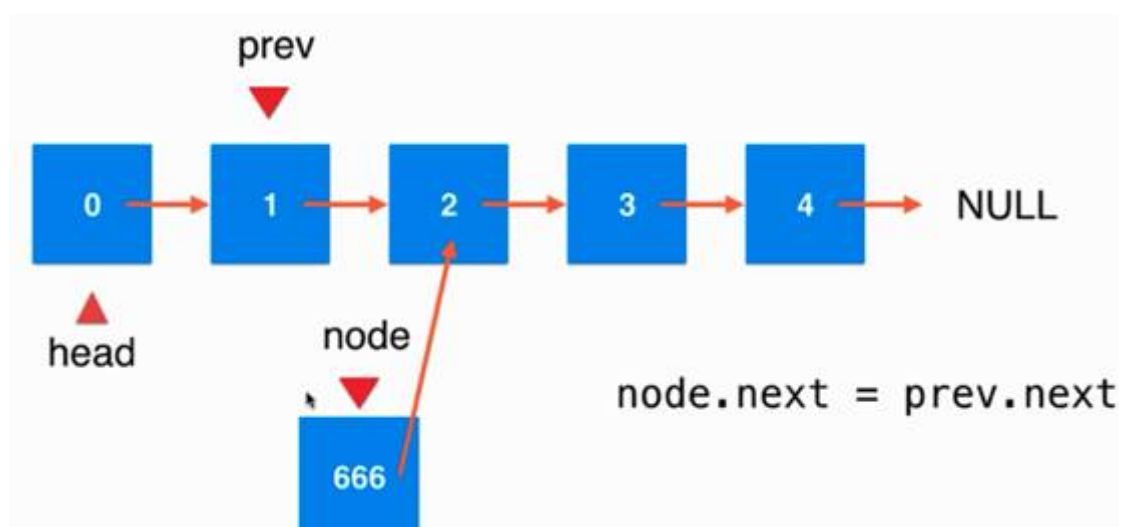
在索引为2的地方添加元素666



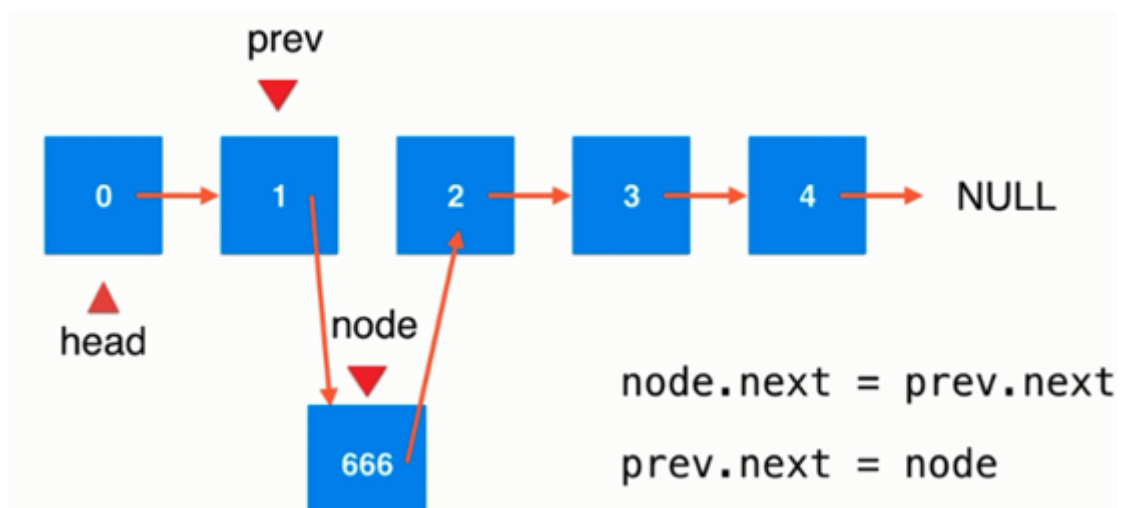
Head节点处有一个prev节点指标



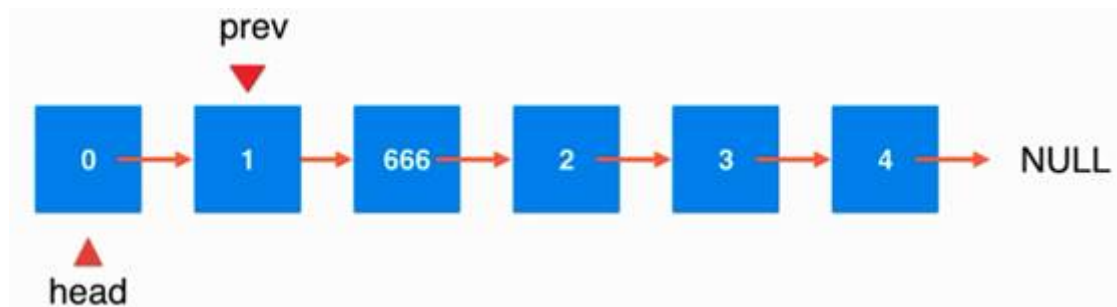
把这个prev插入的需要插入的节点:前一个节点



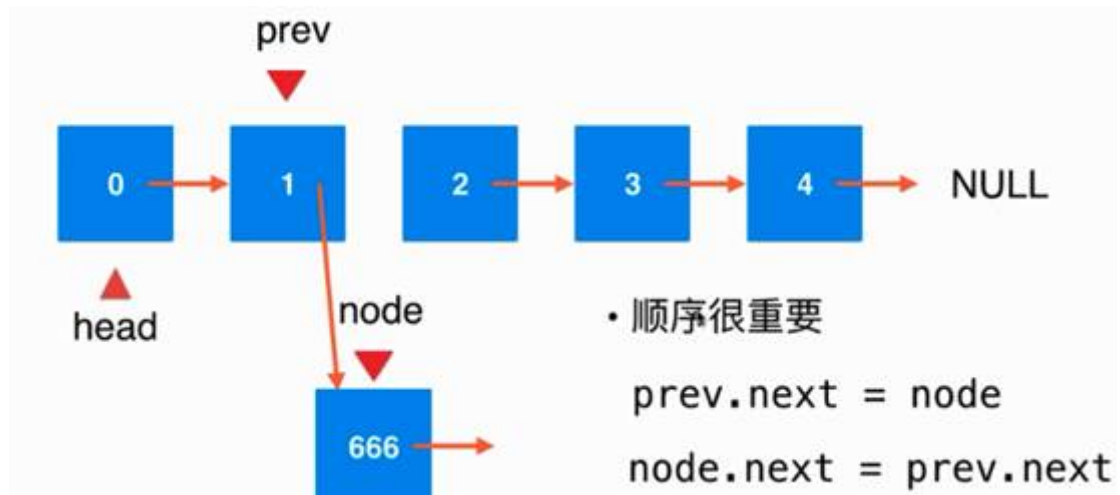
插入过程中关系的转换体现



插入成功后的链表样式



思考一下，执行插入的时候顺序能否发生变化



代码实现：

```

1 public class LinkedList<E> {
2
3     private class Node{
4         public E e;
5         public Node next;
6
7         public Node(E e, Node next){
8             this.e = e;
9             this.next = next;
10        }
11
12        public Node(E e){
13            this(e, null);
14        }
15
16        public Node(){
17            this(null, null);
18        }
19
20        @Override
21        public String toString(){
22            return e.toString();
23        }
24    }
25
26    private Node head;
27    private int size;
28
29    public LinkedList(){
30        head = null;

```

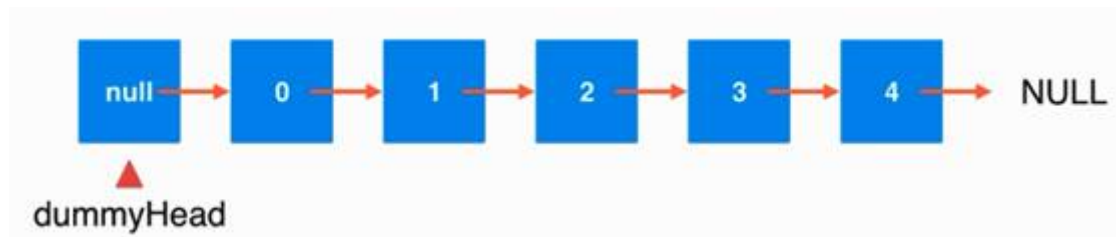
```

31         size = 0;
32     }
33
34     // 获取链表中的元素个数
35     public int getSize(){
36         return size;
37     }
38
39     // 返回链表是否为空
40     public boolean isEmpty(){
41         return size == 0;
42     }
43
44     // 在链表头添加新的元素e
45     public void addFirst(E e){
46         //         Node node = new Node(e);
47         //         node.next = head;
48         //         head = node;
49
50         head = new Node(e, head);
51         size ++;
52     }
53
54     // 在链表的index(0-based)位置添加新的元素e
55     // 在链表中不是一个常用的操作，练习用：)
56     public void add(int index, E e){
57
58         if(index < 0 || index > size)
59             throw new IllegalArgumentException("Add failed. Illegal
index.");
60
61         if(index == 0)
62             addFirst(e);
63         else{
64             Node prev = head;
65             for(int i = 0 ; i < index - 1 ; i ++){
66                 prev = prev.next;
67             }
68             //         Node node = new Node(e);
69             //         node.next = prev.next;
70             //         prev.next = node;
71
72             prev.next = new Node(e, prev.next);
73             size ++;
74         }
75     }
76     // 在链表末尾添加新的元素e
77     public void addLast(E e){
78         add(size, e);
79     }
80 }
81

```



## 给链表使用虚拟头结点



为链表添加一个虚拟的空节点，注意，这个头结点是根本不存在的，是虚拟的，只是为了逻辑方便实现，添加的一个虚拟空的元素，否则就会对处理头结点有不同的逻辑，可以类别循环队列

```
1 public class LinkedList<E> {
2
3     private class Node{
4         public E e;
5         public Node next;
6
7         public Node(E e, Node next){
8             this.e = e;
9             this.next = next;
10        }
11
12        public Node(E e){
13            this(e, null);
14        }
15
16        public Node(){
17            this(null, null);
18        }
19
20        @Override
21        public String toString(){
22            return e.toString();
23        }
24    }
25
26    private Node dummyHead;
27    private int size;
28
29    public LinkedList(){
30        dummyHead = new Node();
31        size = 0;
32    }
33
34    // 获取链表中的元素个数
35    public int getSize(){
36        return size;
37    }
38
39    // 返回链表是否为空
40    public boolean isEmpty(){
41        return size == 0;
42    }
43
44    // 在链表的index(0-based)位置添加新的元素e
45    // 在链表中不是一个常用的操作，练习用：)
```

```

46     public void add(int index, E e){
47
48         if(index < 0 || index > size){
49             throw new IllegalArgumentException("Add failed. Illegal
index.");
50         }
51         Node prev = dummyHead;
52         for(int i = 0 ; i < index ; i ++){
53             prev = prev.next;
54         }
55         prev.next = new Node(e, prev.next);
56         size ++;
57     }
58
59     // 在链表头添加新的元素e
60     public void addFirst(E e){
61         add(0, e);
62     }
63
64     // 在链表末尾添加新的元素e
65     public void addLast(E e){
66         add(size, e);
67     }
68 }
69

```

## 链表的遍历，查询和修改

对于链表来说，查询并非是一个常用操作，目前用于练习使用

LinkedList

```

1  public class LinkedList<E> {
2
3      private class Node{
4          public E e;
5          public Node next;
6
7          public Node(E e, Node next){
8              this.e = e;
9              this.next = next;
10         }
11
12         public Node(E e){
13             this(e, null);
14         }
15
16         public Node(){
17             this(null, null);
18         }
19
20         @Override
21         public String toString(){
22             return e.toString();
23         }
24     }
25

```

```

26     private Node dummyHead;
27     private int size;
28
29     public LinkedList(){
30         dummyHead = new Node();
31         size = 0;
32     }
33
34     // 获取链表中的元素个数
35     public int getSize(){
36         return size;
37     }
38
39     // 返回链表是否为空
40     public boolean isEmpty(){
41         return size == 0;
42     }
43
44     // 在链表的index(0-based)位置添加新的元素e
45     // 在链表中不是一个常用的操作，练习用：)
46     public void add(int index, E e){
47
48         if(index < 0 || index > size){
49             throw new IllegalArgumentException("Add failed. Illegal
index.");
50         }
51
52         Node prev = dummyHead;
53         for(int i = 0 ; i < index ; i ++){
54             prev = prev.next;
55         }
56
57         prev.next = new Node(e, prev.next);
58         size ++;
59
60     // 在链表头添加新的元素e
61     public void addFirst(E e){
62         add(0, e);
63     }
64
65     // 在链表末尾添加新的元素e
66     public void addLast(E e){
67         add(size, e);
68     }
69
70     // 获得链表的第index(0-based)个位置的元素
71     // 在链表中不是一个常用的操作，练习用：)
72     public E get(int index){
73
74         if(index < 0 || index >= size){
75             throw new IllegalArgumentException("Get failed. Illegal
index.");
76         }
77
78         Node cur = dummyHead.next;
79         for(int i = 0 ; i < index ; i ++){
80             cur = cur.next;
81         }
82         return cur.e;
83     }

```

```

82 // 获得链表的第一个元素
83 public E getFirst(){
84     return get(0);
85 }
86
87 // 获得链表的最后一个元素
88 public E getLast(){
89     return get(size - 1);
90 }
91
92 // 修改链表的第index(0-based)个位置的元素为e
93 // 在链表中不是一个常用的操作，练习用：)
94 public void set(int index, E e){
95     if(index < 0 || index >= size)
96         throw new IllegalArgumentException("Set failed. Illegal
index.");
97
98     Node cur = dummyHead.next;
99     for(int i = 0 ; i < index ; i ++){
100         cur = cur.next;
101     }
102     cur.e = e;
103 }
104
105 // 查找链表中是否有元素e
106 public boolean contains(E e){
107     Node cur = dummyHead.next;
108     while(cur != null){
109         if(cur.e.equals(e))
110             return true;
111         cur = cur.next;
112     }
113     return false;
114 }
115
116 @Override
117 public String toString(){
118     StringBuilder res = new StringBuilder();
119
120     // Node cur = dummyHead.next;
121     // while(cur != null){
122     //     res.append(cur + "->");
123     //     cur = cur.next;
124     // }
125     for(Node cur = dummyHead.next ; cur != null ; cur = cur.next)
126         res.append(cur + "->");
127     res.append("NULL");
128
129     return res.toString();
130 }
131 }

```

Main

```

1 public class Main {
2
3     public static void main(String[] args) {

```

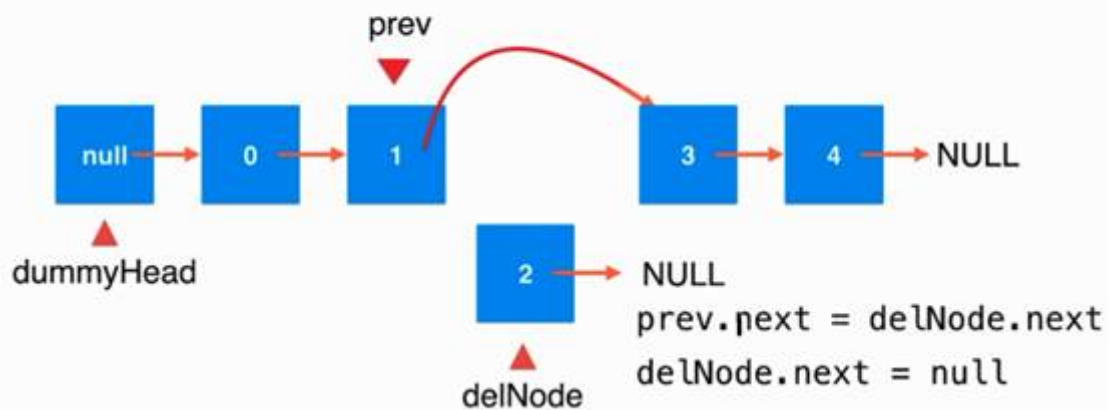
```

4
5     LinkedList<Integer> linkedList = new LinkedList<>();
6     for(int i = 0 ; i < 5 ; i ++){
7         linkedList.addFirst(i);
8         System.out.println(linkedList);
9     }
10
11     linkedList.add(2, 666);
12     System.out.println(linkedList);
13 }
14 }
15

```

## 从链表中删除元素

• 删除索引为2位置的元素



```

1 // 从链表中删除index(0-based)位置的元素，返回删除的元素
2 // 在链表中不是一个常用的操作，练习用：)
3 public E remove(int index){
4     if(index < 0 || index >= size)
5         throw new IllegalArgumentException("Remove failed. Index is
6         illegal.");
7
8     Node prev = dummyHead;
9     for(int i = 0 ; i < index ; i ++){
10         prev = prev.next;
11
12     Node retNode = prev.next;
13     prev.next = retNode.next;
14     retNode.next = null;
15     size --;
16
17     return retNode.e;
18 }
19
20 // 从链表中删除第一个元素，返回删除的元素
21 public E removeFirst(){
22     return remove(0);
23 }
24
25 // 从链表中删除最后一个元素，返回删除的元素

```

```

25     public E removeLast(){
26         return remove(size - 1);
27     }
28
29     // 从链表中删除元素e
30     public void removeElement(E e){
31
32         Node prev = dummyHead;
33         while(prev.next != null){
34             if(prev.next.e.equals(e))
35                 break;
36             prev = prev.next;
37         }
38
39         if(prev.next != null){
40             Node delNode = prev.next;
41             prev.next = delNode.next;
42             delNode.next = null;
43             size--;
44         }
45     }
46

```

时间复杂度计算

添加操作	$O(n)$
addLast(e)	$O(n)$
addFirst(e)	$O(1)$
add(index, e)	$O(n/2) = O(n)$
删除操作	$O(n)$
removeLast(e)	$O(n)$
removeFirst(e)	$O(1)$
remove(index, e)	$O(n/2) = O(n)$

查找操作	$O(n)$
get(index)	$O(n)$
contains(e)	$O(n)$
<del>find(e)</del>	<del><math>O(n)</math></del>

总结

- 增：  $O(n)$  → 如果只对链表头进行操作：  $O(1)$
- 删：  $O(n)$  → 如果只对链表头进行操作：  $O(1)$
- 改：  $O(n)$  →
- 查：  $O(n)$  → 只查链表头的元素：  $O(1)$

## 使用链表实现栈

```
Interface Stack<E>    ←----- LinkedListStack<E>
• void push(E)        implement
• E pop()
• E peek()
• int getSize()
• boolean isEmpty()
```

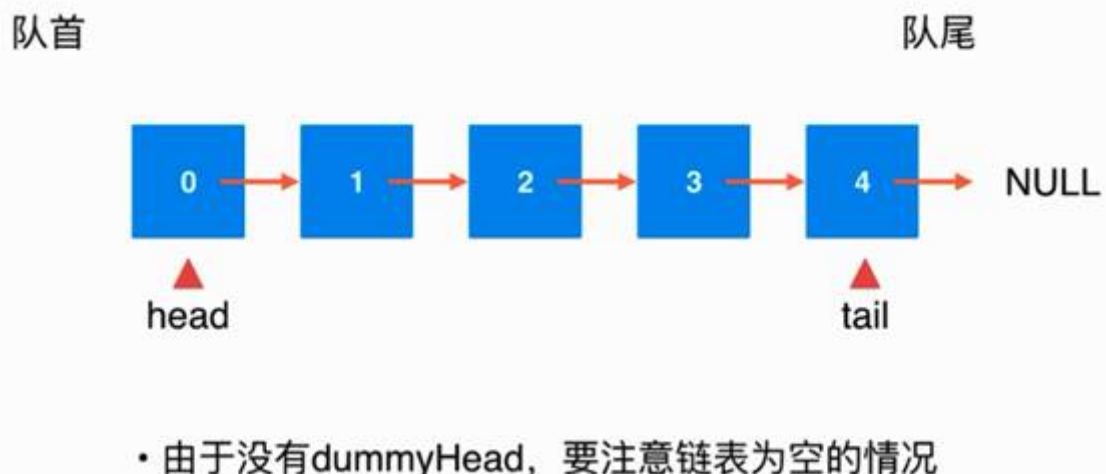
Stack

1 |

Main

1 |

## 使用链表实现队列结构



```
1 public class LinkedListQueue<E> implements Queue<E> {
2
```

```
3     private class Node{
4         public E e;
5         public Node next;
6
7         public Node(E e, Node next){
8             this.e = e;
9             this.next = next;
10        }
11
12        public Node(E e){
13            this(e, null);
14        }
15
16        public Node(){
17            this(null, null);
18        }
19
20        @Override
21        public String toString(){
22            return e.toString();
23        }
24    }
25
26    private Node head, tail;
27    private int size;
28
29    public LinkedListQueue(){
30        head = null;
31        tail = null;
32        size = 0;
33    }
34
35    @Override
36    public int getSize(){
37        return size;
38    }
39
40    @Override
41    public boolean isEmpty(){
42        return size == 0;
43    }
44
45    @Override
46    public void enqueue(E e){
47        if(tail == null){
48            tail = new Node(e);
49            head = tail;
50        }
51        else{
52            tail.next = new Node(e);
53            tail = tail.next;
54        }
55        size ++;
56    }
57
58    @Override
59    public E dequeue(){
60        if(isEmpty())
```



```

61         throw new IllegalArgumentException("Cannot dequeue from an
empty queue.");
62
63         Node retNode = head;
64         head = head.next;
65         retNode.next = null;
66         //判断一下整个队列为空的情况
67         if(head == null)
68             tail = null;
69         size--;
70         return retNode.e;
71     }
72
73     @Override
74     public E getFront(){
75         if(isEmpty())
76             throw new IllegalArgumentException("Queue is empty.");
77         return head.e;
78     }
79
80     @Override
81     public String toString(){
82         StringBuilder res = new StringBuilder();
83         res.append("Queue: front ");
84
85         Node cur = head;
86         while(cur != null) {
87             res.append(cur + "->");
88             cur = cur.next;
89         }
90         res.append("NULL tail");
91         return res.toString();
92     }
93
94     public static void main(String[] args){
95
96         LinkedListQueue<Integer> queue = new LinkedListQueue<>();
97         for(int i = 0 ; i < 10 ; i ++){
98             queue.enqueue(i);
99             System.out.println(queue);
100
101             if(i % 3 == 2){
102                 queue.dequeue();
103                 System.out.println(queue);
104             }
105         }
106     }
107 }
108

```

```

1 package com.myLinkedList;
2
3 public class LinkedList2<E> {
4
5     private class Node{

```

```

6         public E e;
7         public Node next;
8
9         public Node(E e, Node next){
10             this.e = e;
11             this.next = next;
12         }
13
14         public Node(E e){
15             this(e, null);
16         }
17
18         public Node(){
19             this(null, null);
20         }
21
22
23         @Override
24         public String toString() {
25             // return e.toString();
26             return "Node [e=" + e + ", next=" + next + "]";
27         }
28
29
30     }
31
32     private Node dummyHead;
33     private int size;
34
35     public LinkedList2(){
36         dummyHead = new Node();
37         size = 0;
38     }
39
40     // 获取链表中的元素个数
41     public int getSize(){
42         return size;
43     }
44
45     // 返回链表是否为空
46     public boolean isEmpty(){
47         return size == 0;
48     }
49
50     // 在链表的index(0-based)位置添加新的元素e
51     // 在链表中不是一个常用的操作，练习用：）
52     public void insert(int index, E e){
53
54         if(index < 0 || index > size) {
55             throw new IllegalArgumentException("Add failed. Illegal
index.");
56         }
57         Node prev = dummyHead;
58         for(int i = 0 ; i < index ; i ++){
59             prev = prev.next;
60         }
61         prev.next = new Node(e, prev.next);
62         size ++;

```

```
63     }
64
65     // 在链表头添加新的元素e
66     public void addHead(E e){
67         insert(0, e);
68     }
69
70     // 在链表末尾添加新的元素e
71     public void addLast(E e){
72         insert(size, e);
73     }
74
75     @Override
76     public String toString() {
77         return "LinkedList2 [dummyHead=" + dummyHead + ", size=" + size +
78         "];"
79     }
80 }
```