数据结构常见问题分析

千磨万击还坚韧，任尔东西南北风。
飞花摘叶皆可伤人、草木竹石均可为剑。

1 入门分析

1.1什么是数据结构？

数据结构一般泛指数据的逻辑结构和存储结构，独立于具体编程语言。例如，我们在浏览一个网页时，看到的页面布局结构。拿到一本书时，看到的书的目录结构。打开手机上的地图软件，看到的图结构等等都和我们所说的数据结构有关系。

1.2什么是算法？

在编程领域中，可以将程序理解为数据结构+算法。算法就是操作特定结构数据的方法或技巧，具备正确性，可行性，有穷性，输入，输出等特征。例如对数据的查找，排序，运算等都会涉及到具体算法的应用。

1.3数据结构和算法是什么关系？

数据结构和算法是相辅相成的，数据结构服务于算法，算法作用于数据结构。

1.4为什么要学习数据结构和算法？

让“计算”更加的高效和低耗。一般越是注重技术的公司（尤其是大型的科技公司），越会注重考察数据结构与算法这类基础知识。因为相比短期能力，他们更看中你的长期潜力。

1.5学习数据结构和算法可以帮助到我们什么？

学习数据结构和算法可以更加有效的帮我们：

1） 建立时间复杂度、空间复杂度意识，写出高质量的代码。
2） 设计更好基础架构，提升编程技能，训练结构化思维的有效手段。

总之：学习数据结构和算法，可以有效帮助我们，在设计“高效，低耗”的软件系统时，起到很大的指导作用。如同具备九阳神功护体，见招拆招，对各种技术进行灵活应用。

1.6如何学习数据结构和算法？

在学习数据结构和算法的过程中，首先要对这个学科有一个整体的认识，比方说数据结构和算法这门课会涉及到哪些知识点，哪些是重点等，还有就是这些知识点的「来历」、「自身的特点」、「适合解决的问题」以及「实际的应用场景」。其次要基于大纲进行刻意的练习，我们要练习缺陷，练习弱点，练习不舒服，不爽，枯燥的地方。最后学会分享，在分享的过程中及时获取反馈和加强。

1.7什么是数据的逻辑结构？

数据的逻辑结构描述的是数据元素之间的逻辑关系，与数据的存储无关，是独立于计算机的一种结构。数据的逻辑结构可以看作是从具体问题抽象出来的数学模型。具体可分为线性结构（栈，队列），非线性结构（集合,树形结构，图结构）。

1.8什么是数据的存储结构？

数据的存储结构是数据元素以及其关系在计算机存储器内的表示,是逻辑结构用计算机语言的实现。例如顺序存储结构（位置相邻）、链状存储结构（指针关联）。

1.9什么是复杂度分析？

复杂度分析几乎占了数据结构和算法这门课的半壁江山，是数据结构和算法学习的精髓，它告诉我们在设计一个系统时，如何度量资源的消耗，运行的效率。它探讨的是一种心法，只有我们学会了分析问题的方式，才能见招拆招，做到无招胜有招。

1.10 如何度量程序的复杂度？

那么我们应该如何去度量程序中算法的优劣呢？主要还是从算法所占用的「时间」和「空间」两个维度去考量。

1） 时间维度：是指执行当前算法所消耗的时间，我们通常用「时间复杂度」来描述。
2） 空间维度：是指执行当前算法需要占用多少内存空间，我们通常用「空间复杂度」

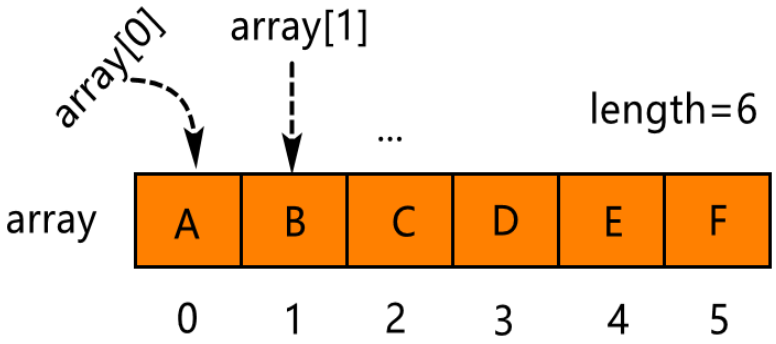
来描述。

因此，评价一个算法的效率主要是看它的时间复杂度和空间复杂度情况。然而，有的时候时间和空间却又是「鱼和熊掌」，不可兼得的，那么我们就需要从中去取一个平衡点。

2 数组(Array)应用分析

2.1什么是数组？

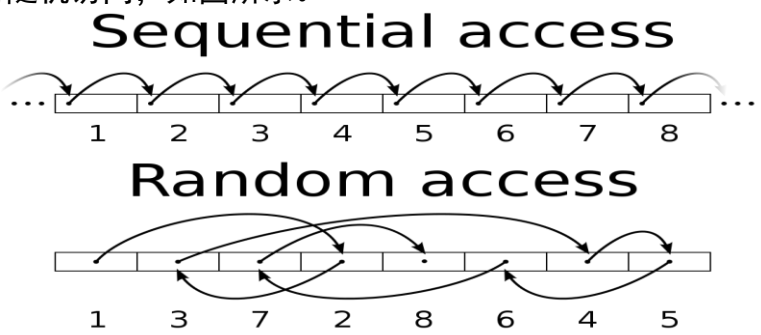
数组是一个有限的、类型相同的数据的集合，在内存中是一段连续的内存区域，这段内存区域一旦定义，其大小不改变。



数组的长度在定义时确定，数组中的元素的默认值由其数组类型决定。可通过数组下标访问数组中元素，下标的起始位置永远从 0 开始。

2.2如何查找数组中元素？

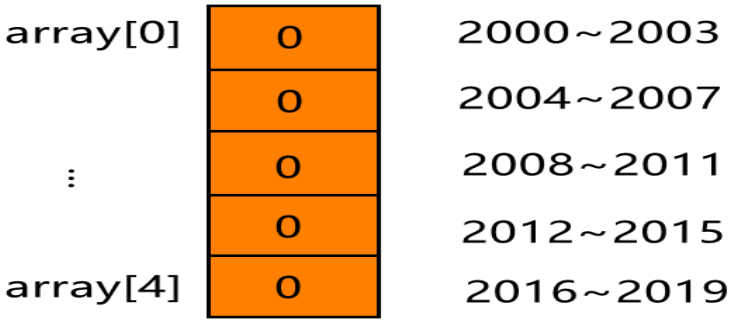
数组在访问操作方面有着独特的性能优势，因为数组不仅仅支持顺序访问，还支持随机访问，如图所示。



我们可以通过下标随机访问数组中任何一个元素，其原理是因为数组中元素

的存储是连续的，所以我们可以通过数组内存空间的首地址加上元素的偏移量计算出某一个元素的内存地址，例如，array[n]的地址 = array 数组内存空间的首地址 + 每个元素大小*n。

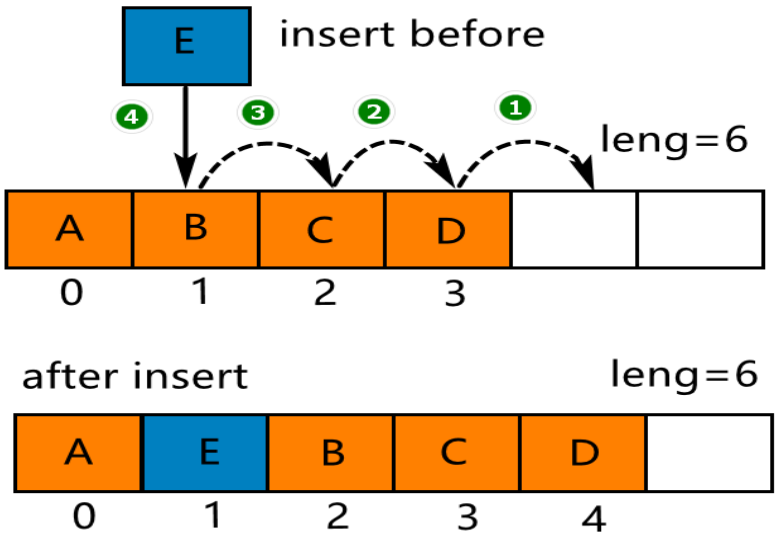
构建一个长度为 5 的整数数组 int[]array=new int[5],在内存中会开辟一块连续的内容空间，假设其首地址为 2000，其它元素的地址表示，如图所示。



总之，当我们通过下标去访问数组中的数据时，并不需要从头开始依次遍历数组，因此数组的访问时间复杂度是 O(1)，当然这里需要注意，如果不是通过下标去访问，而是通过内容去查找数组中的元素，则时间复杂度不是 O(1)，极端的情况下需要遍历整个数组的元素，时间复杂度可能是 O(n)，当然通过不同的查找算法所需的时间复杂度是不一样的。

2.3如何实现数组中数据的插入和删除？

数组元素的连续性，导致数组在插入和删除元素的时候效率比较低。如果要在数组中间插入一个新元素，就必须要将要相邻的后面的元素全部往后移动一个位置，留出空位给这个新元素，如图所示。



数组插入时，首先要检测数组中是否有足够的空间可以存储新的元素，假如不足还需要

对数组进行扩容。一般会重新申请一个相对原数组 1.5 倍大小的存储空间（因为数组在内存中是一块连续的内存空间，一旦定义其长度不可以再进行修改），并且把原来的数据拷贝到新申请的空间上。假如现在数组空间是足够的，新元素要插入在数组的最开头位置，那整个原始数组都需要向后移动一位，此时的时间复杂度为最坏情况即 O(n)，如果新元素要插入的位置是最末尾，则无需其它元素移动，则此时时间复杂度为最好情况即 O(1)，所以平均而言数组插入的时间复杂度是 O(n)。

Java 中数组插入操作，其代码示例如下：

```
    // function to search a key to
// be deleted
static int findElement(int arr[], int n, int key) {
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == key)
            return i;

    return -1;
}
```

数组的删除与插入类似，假如不是删除最后一个有效元素，其它元素在删除以后，后续元素都要向前移动，如图-5 所示：

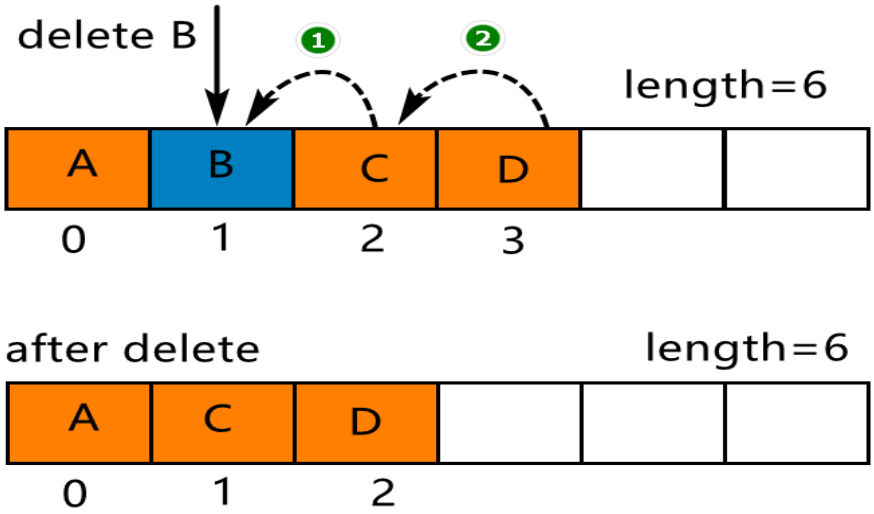


图-5

数组删除时，如果删除数组末尾的数据，则最好情况时间复杂度为 O(1)。如果删除开头的数据，则最坏情况时间复杂度为 O(n)。所以平均情况时间复杂度也为 O(n)。Java 中的数组删除操作，其实现代码如下：

```
// function to search a key to
// be deleted
static int findElement(int arr[], int n, int key) {
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == key)
            return i;
```

```
    return -1;
}
```

```
        // Function to delete an element
    static int deleteElement(int arr[], int n, int key)  {
        // Find position of element to be
        // deleted
        int pos = findElement(arr, n, key);

        if (pos == -1)
        {
            System.out.println("Element not found");
            return n;
        }

        // Deleting element
        int i;
        for (i = pos; i< n - 1; i++)
            arr[i] = arr[i + 1];

        return n -1;
    }
}
```

2.4JAVA 中手写简易 ArrayList 容器？

```
    public class SimpleArrayContainer {

    private Object[] array;
    private int size;
    public SimpleArrayContainer() {
        this(10);
    }
    public SimpleArrayContainer(int capacity) {
        this.array=new Object[capacity];
    }

    private Object[] copyOf(Object[] source,int length) {
        //1)创建新数组，并定义其长度为原数组的 2 倍大小
        Object[] newArray=new Object[2*length];
        //2)拷贝原有数组内容到新数组
        for(int i=0;i<source.length;i++) {
            newArray[i]=source[i];
        }
        //3)返回新创建的数组
        return newArray;
    }
```

```
    //定义向数组 size 位置添加新元素的方法
    public void add(Object element) {
        //1.数组是否已满了，满了则扩容
        if(size==array.length) {
            //array=copyOf(array, 2*array.length);
            array=Arrays.copyOf(array, 2*array.length);
        }
        //2.添加新的元素
        array[size++]=element;
    }
    //定义在指定位置添加新的元素的方法
    public void add(int index,Object element) {
        if(index<0||index>size)
            throw new IndexOutOfBoundsException();
        if(size==array.length) {
            array=Arrays.copyOf(array, 2*array.length);
        }
        System.arraycopy(array, index, array, index+1, size-index);
        array[index]=element;
        size++;
    }
    //定义按指定位置删除数组元素的方法
    public Object remove(int index) {
        //1.参数校验
        if(index<0||index>=size)
            throw new IndexOutOfBoundsException();
        //2.获取 index 位置元素
        Object element=array[index];
        //3.移动元素
//      int numMoved=size-index-1;
//      if(numMoved>0)
//      System.arraycopy(array, index+1, array, index, numMoved);
//      //4.修改最后一个元素的值，并且有效元素个数减 1
//      this.array[--size]=null;
        fastRemove(index);
        return element;
    }
    private void fastRemove(int index) {
        int numMoved=size-index-1;
        if(numMoved>0)
            System.arraycopy(array, index+1, array, index, numMoved);
        array[--size]=null;
    }
    //按元素值移除元素
    public boolean remove(Object element) {
        if(element==null) {
            for(int index=0;index<size;index++) {
                if(array[index]==null) {
                    //移除元素
                    fastRemove(index);
                    return true;
                }
```

```
            }
        }else {
            for(int index=0;index<size;index++) {
                if(array[index].equals(element)) {
                    //移除元素
                    fastRemove(index);
                    return true;
                }
            }

        }
        return false;
    }

    public Object get(int index) {
        if(index<0||index>=size)
            throw new IndexOutOfBoundsException();
        return this.array[index];
    }

    @Override
    public String toString() {
        return Arrays.toString(array);
    }
    public int size() {
        return size;
    }
    public static void main(String[] args) {
        SimpleArrayContainer container=new SimpleArrayContainer(2);
        container.add(100);//size
        container.add(200);
        container.add(300);
        System.out.println(container);//[100,200,300]
        container.add(1, 400);
        System.out.println(container);//[100,400,200,300]
        container.remove(1);
        System.out.println(container);//[100,200,300]
        container.remove((Object)200);
        System.out.println(container);//[100,300]
        Object element=container.get(0);
        System.out.println(element);
    }
}
```

2.5基于 JAVA 如何实现数组元素的旋转？

初始数组为 Input,然后向左旋转数组实现 Output 数组结果

```
Input :  arr[] = [1, 2, 3, 4, 5, 6, 7]

rotate Left

Output : arr[] = [3, 4, 5, 6, 7, 1, 2]
```

初始数组为 Input,然后向右旋转数组实现 Output 数组结果

```
Input :  arr[] = [1, 2, 3, 4, 5, 6, 7]

rotate right

Output : arr[] = [6, 7, 1, 2, 3, 4, 5]
```

具体实现方案如下：

```
package com.cy.pj.ds.array;

import java.util.Arrays;

public class RotateLeftArray {

    //左转方案1：时间复杂度为O(n),辅助空间复杂度为O(count)
    static void doRotateLeft01(int[] source,int count) {
        //1.构建临时数组，用于存储需要移动数组右边的元素
        int[] temp=new int[count];
        for(int i=0;i<temp.length;i++) {//空间复杂度 S=O[count]
            temp[i]=source[i];
        }
        //2.移动source 数组中剩余元素
        for(int i=count;i<source.length;i++) {//时间复杂度 t=O[n]
            source[i-count]=source[i];
        }
        //3.将temp 数组中的元素存储到source 数组
        for(int i=0;i<temp.length;i++) {
            source[source.length-(count-i)]=temp[i];
        }
    }

    //左转方案2：时间复杂度O(n*count)，辅助空间O(1)
    //分多次移动数组元素
    //[1,2,3,4,5,6,7]
    //[2,3,4,5,6,7,1]
    //[3,4,5,6,7,1,2]
    static void doRotateLeft02(int[] source,int count) {
        for(int i=0;i<count;i++) {//外层循环控制移动次数，T=O(n*count)
            int temp=source[0],j;//空间复杂度,S=O(1)
```

```
            for(j=0;j<source.length-1;j++) {//移动元素的个数
                source[j]=source[j+1];
            }
            source[j]=temp;
        }
    }
    //右转方案1：时间复杂度为O(n*count)，辅助空间为O(1)
    static void doRotateRight01(int[] source,int count) {
        int len=source.length-1;
        for(int i=0;i<count;i++) {
            int temp=source[len],j;
            for(j=len;j>0;j--) {
                source[j]=source[j-1];
            }
            source[j]=temp;
        }
    }

    public static void main(String[] args) {
        //rotate left
        //intput:[1,2,3,4,5,6,7]
        //rotate left
        //output:[3,4,5,6,7,1,2]
        int[] source= {1,2,3,4,5,6,7};
        //doRotateLeft01(source,2);
        //doRotateLeft02(source,2);

        //rotate right
        //intput:[1,2,3,4,5,6,7]
        //rotate right
        //output:[6,7,1,2,3,4,5]

        doRotateRight01(source, 2);
        System.out.println(Arrays.toString(source));

    }

}
```

2.6基于 JAVA 实现有序数组的二分查找操作？

```
static int binarySearch(int arr[],
                        int low, int high, int key)
{
    if (high < low)
        return -1;

    /* low + (high - low)/2; */
```

```
    int mid = (low + high)/2;
    if (key == arr[mid])
        return mid;
    if (key > arr[mid])
        return binarySearch(arr, (mid + 1), high, key);
    return binarySearch(arr, low, (mid -1), key);
}
```

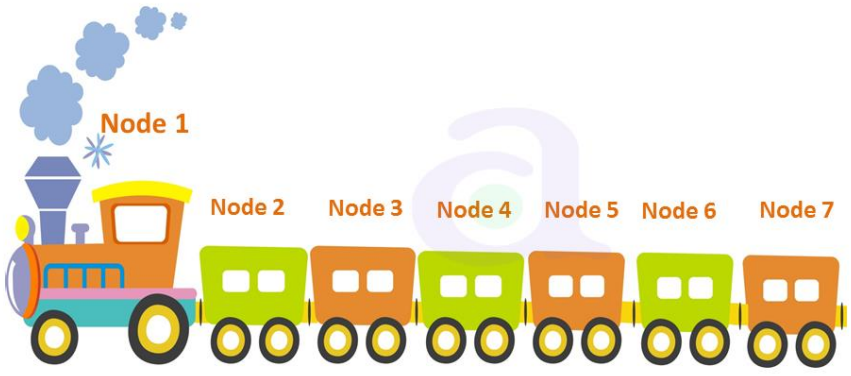
2.7数组的优势和劣势是什么？

数组连续的内存空间，可以借助 CPU 的缓存机制，预读数组中的数据，所以数据也就具备了很好的“随机访问”的性能。但有利就有弊，这个限制也让数组的很多操作变得非常低效。例如，数组中删除、插入一个数据时，为了保证连续性，就需要做大量的数据移动操作。

3 链表(Linked List)应用分析

3.1什么是链表？

链表是一种物理存储单元上并不要求连续的存储结构（当然也可以连续的，数组要求必须连续），链表中数据元素之间的逻辑顺序，是通过链表中的指针指向进行实现的（逻辑上相邻但物理上不一定相邻，数组中逻辑和物理上都相邻），如图所示：



其中：链表结构中，为了表示每个数据元素与其直接后继数据元素之间的逻辑关系,除了要存储其本身的数据之外,还需存储一个指针，用于指向其直接后继元素(其实是直接后继的存

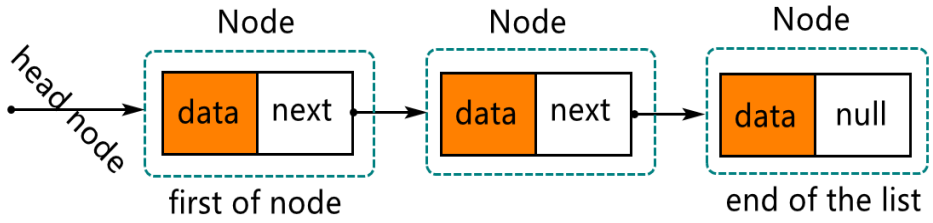
储位置)，这两部分信息就是我们所说的一个结点元素。多个节点元素通过指针域建立关系并形成链表。

3.2 你知道的链表结构类型有哪些？

链表也分为好几种，常见的有单向链表，单向循环链表，双向链表，双向循环链表。

3.3 单向链表有什么特点？

最简单的一种链表，每一个节点（Node）除了存储数据之外，只有一个指针（后继指针，也称引用）指向后面一个节点（Node），这个链表称为单向链表。其基本形态如图所示：



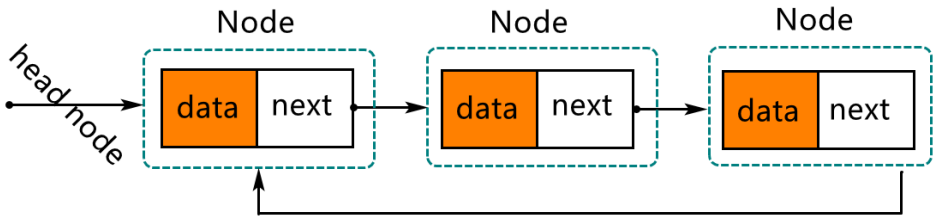
对于”单链表”而言,有两个节点比较特殊，分别是第一个节点和最后一个节点。我们一般习惯于将第一个节点称为“头节点”（head），最后一个节点称为“尾节点”（tail）。“头节点”一般用于记录链表的基地址，通过此地址获取链表中的节点对象。“尾节点”的指针域不指向任何节点，指针域的值为 null，表示最后一个节点。

3.4 基于 JAVA 如何定义单项链表中的节点类型？

```
class Node{
    Object data;
    Node next;

    Node(Object element, Node next) {
        this.data= element;
        this.next = next;
    }
}
```

3.5 什么是单向循环链表？

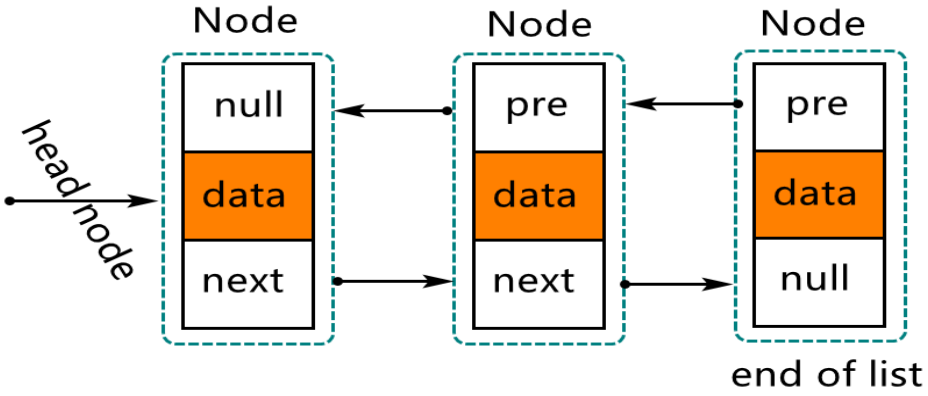
循环链表就是一种特殊的单向链表，只不过在单向链表的基础上，将尾节点的指针指向了 Head 节点，使之首尾相连。如图所示：



单向循环链表相对于单向链表的优点是从链尾到链头比较方便。当要处理的数据具有环型结构特点时，就特别适合采用单向循环链表，这样可以很大程度上减少代码量。

3.6什么是双向链表？

双向链表与单向链表的区别是前者是 2 个方向都有指针，后者只有 1 个方向的指针。双向链表的每一个节点都有 2 个指针，一个指向前驱节点，一个指向后继节点。如图所示：



双向链表相对于单向链表，需要额外的一个空间来存储前驱结点的地址。所以，如果存储同样多的数据，双向链表要比单链表占用更多的内存空间。但双向链表支持双向遍历，这样也带来了双向链表操作的灵活性。例如，单从结构上来看，双向链表可以支持 O(1)时间复杂度的情况下找到前驱结点，正是这样的特点，也使双向链表在某些情况下的插入、删除等操作都要比单向链表更加简单和高效。

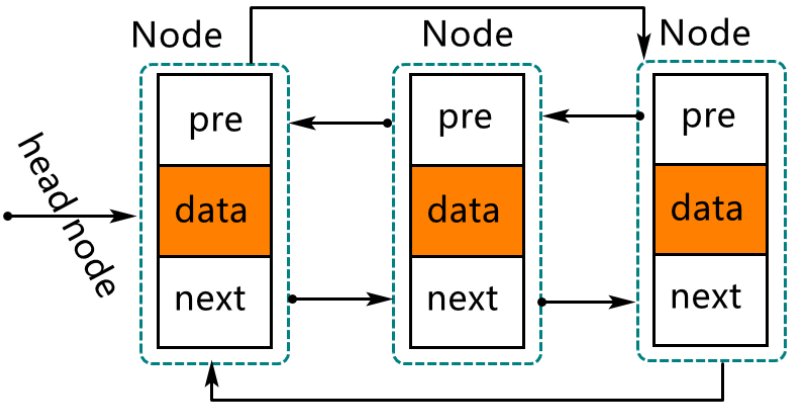
3.7基于 Java 如何定义双向链表节点类型？

```
class Node {
    Object element;
    Node next;
    Node pre;

    Node(Object element, Node pre, Node next) {
```

```
        this.element= element;
        this.pre= pre;
        this.next = next;
    }
}
```

3.8什么是双向循环链表？

双向循环链表只是在双向链表的基础上，添加了头节点与尾节点的双向引用，如图所示：



说明:双向循环链表可以从任意节点开始向前或向后查找节点。

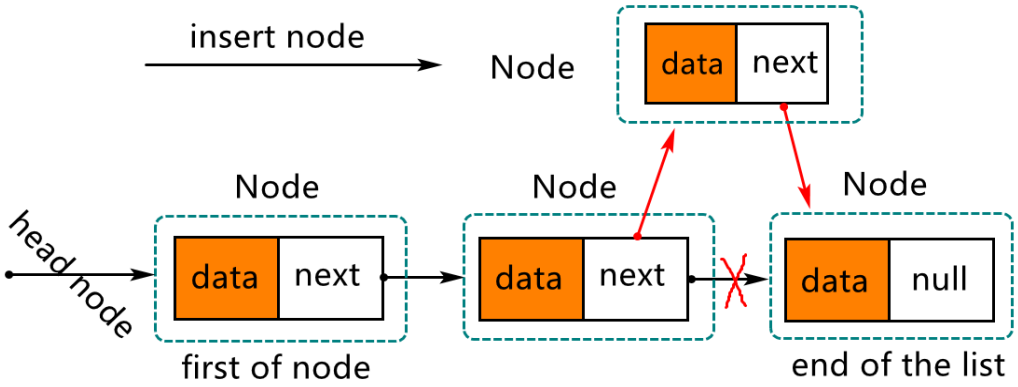
3.9链表中节点数据的查找过程是怎样的？

链表的优势并不在与访问，因为链表无法通过首地址和下标去计算出某一个节点的地址，所以链表中如果要查找某个节点，则需要一个节点一个节点的遍历，因此链表的访问时间复杂度为 O(n)，但对于双向链表而言，假如已经确定某个节点的位置，再去查找其上个节点的位置，可以直接通过前驱指针直接获取上一个点对象地址即可，这一些要比单向链表有优势。

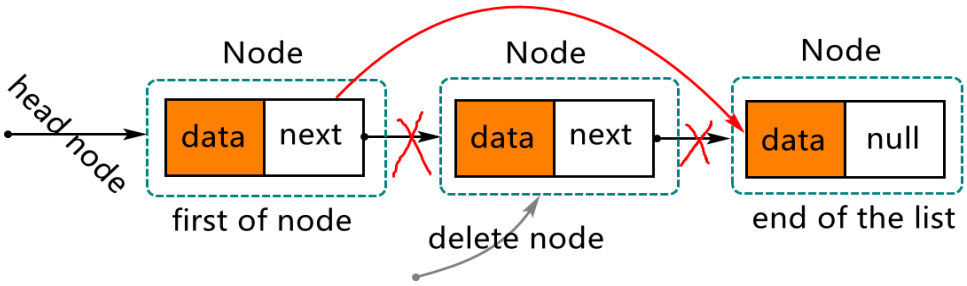
3.10 链表中数据的插入和删除过程是怎样的？

我们知道在进行数组的插入、删除操作时，为了保持内存数据的连续性，需要做大量的数据移动，所以时间复杂度是 O(n)。而在链表中插入或者删除一个数据，我们并不需要为了保持内存的连续性而移动节点，因为链表中节点的存储空间本身就不需要连续。所以，在链表中插入和删除一个节点时是非常快速的。我们只需要修改指针的指向即可。

链表中节点的插入，如下图所示：



链表中节点的删除，如下图所示：



对于链表而言，无论是执行插入还是删除操作，他们的时间复杂度可以理解为 O(1)。但是在插入和删除之前，需要遍历查找的时间复杂度为 O(n)。根据时间复杂度分析中的加法法则，插入或删除某个值的结点时，对应的总时间复杂度仍旧为 O(n)。

3.11 基于 JAVA 技术如何实现一个单向链表？

```
package com.cy.pj.ds.linked;
import java.util.NoSuchElementException;

public class SimpleSingleLinkedList {
    //头节点
    private Node head;
    //记录有效元素的个数
    private int size;
    //每一个节点的类型
    class Node{
        Object data;
        Node next;
        public Node(Object data) {
            this.data=data;
```

```
        }
    }

    public Node getHead() {
        return head;
    }
    //定义向链表头部位置添加节点的方法
    public void addFirst(Object data) {
        //1.create new node
        Node newNode=new Node(data);
        //2.make next of new node as head
        newNode.next=head;
        //3.move the head to new node;
        head=newNode;
        //4.update size
        size++;
    }
    //定义在链表尾部添加新节点的方法
    public void addLast(Object data) {
        //1.创建新节点
        Node newNode=new Node(data);
        //2.判定头节点是否为空，假如头节点为空则当前节点应该为头节点
        if(head==null) {head=newNode;return;}
        //3.获取链表中的最后一个节点
        Node last=head;
        while(last.next!=null)last=last.next;
        //4.设置最后一个节点的next 节点为当前的新节点
        last.next=newNode;
        //5.更新size 的值
        size++;
    }

    //定义在指定位置添加新的节点
    public void add(int index,Object data) {
        //检查 index 的范围
        if(index<0||index>size)
            throw new IndexOutOfBoundsException();
        //添加头部节点(index==0)
        if(index==0) {addFirst(data);return;}
        //添加尾部节点(index==size)
        if(index==size) {addLast(data);return;}
        //查找 index-1 位置的节点，并在 index 位置添加新节点
        Node preNode=node(index-1);
        Node newNode=new Node(data);
        newNode.next=preNode.next;
        preNode.next=newNode;
        //更新size 的值。
        size++;
    }

    //按位置查找节点
    Node node(int index) {
```

```
        Node node=head;
        for(int i=0;i<index;i++) {
            node=node.next;
        }
        return node;
    }
    //==================remove node============
    //移除头部节点
    public void removeFirst() {
        if(size==0||head==null)
            throw new NoSuchElementException();
        head=head.next;
        size--;
    }
    //移除尾部节点
    public void removeLast() {
        if(size==0||head==null)
            throw new NoSuchElementException();
        if(size==1) {
            head=null;
        }else {
            Node preNode=node(size-2);
            preNode.next=null;
        }
        size--;
    }
    //按指定位置移除节点

    public void removeNode(int index) {
        //1.越界检查
        if(index<0||index>=size)
            throw new IndexOutOfBoundsException();
        //2.判定是否为头节点并移除
        if(index==0) {
            removeFirst();
            return;
        }
        //3.判定是否为尾节点并移除。
        if(index==size-1) {
            removeLast();
            return;
        }
        //4.查找节点进行移除。
        Node preNode=node(index-1);
        preNode.next=preNode.next.next;
        //5.修改 size 的值
        size--;
    }
    //基于节点 data 属性值移除节点对象
    public void removeNode(Object data) {
        //1.获得 head 节点并进行判定和移除
        Node temp=head,pre=null;
        if(temp!=null&&temp.data.equals(data)) {
```

```
            head=temp.next;
            size--;
            return;
        }
        //2.查找对应节点
        while(temp!=null&&!temp.data.equals(data)){
            pre=temp;
            temp=temp.next;
        }
        if(temp==null)return;
        //3.移除节点
        pre.next=temp.next;
        //4.更新size的值
        size--;
    }
    @Override
    public String toString() {
        StringBuilder sb=new StringBuilder("[");
        Node node=head;
        while(node!=null) {
            sb.append(node.data).append(",");
            node=node.next;
        }
        if(sb.length()>1)sb.deleteCharAt(sb.length()-1);
        sb.append("]");
        return sb.toString();
    }
    public int size() {
        return size;
    }

    public static void main(String[] args) {
        //doTestAddRemove();
        doTestContainsLoop();
    }

    private static void doTestContainsLoop() {
        SimpleSingleLinkedList list=new SimpleSingleLinkedList();
        list.addFirst(100);
        list.addFirst(200);
        list.addFirst(300);
        System.out.println(list);//300,200,100
        list.detactLoop();
        list.head.next.next.next=list.head;
        list.detactLoop();
    }



    private static void doTestAddRemove() {
        SimpleSingleLinkedList list=new SimpleSingleLinkedList();
        list.addFirst(100);
        list.addFirst(200);
```

```
        list.addFirst(300);
        System.out.println(list);//[300,200,100]
        list.addLast(400);
        System.out.println(list);//[300,200,100,400]
        list.add(1, 500);
        System.out.println(list);//[300,500,200,100,400]
        list.removeFirst();
        System.out.println(list);//[500,200,100,400]
        list.removeLast();//
        System.out.println(list);//[500,200,100]
        list.removeNode(1);//index
        System.out.println(list);//[500,100]
        list.removeNode((Object)100);//data
        System.out.println(list);//[500]
    }
}
```

3.12 基于 JAVA 技术如何实现一个双向链表？

```
package com.cy.pj.ds.linked;

import java.util.NoSuchElementException;

public class SimpleDoubleLinkedList {

    //first of node
    private Node first;
    //last of node
    private Node last;
    //number of elements
    private int size;
    //the type of node
    class Node{
        //store the object
        Object element;
        //prev of the node;
        Node prev;
        //next of the node
        Node next;
        public Node( Node prev, Object element, Node next) {
            this.element=element;
            this.prev=prev;
            this.next=next;
        }
```

```
    }


    //添加头节点
    public void addFirst(Object element) {
        //1.存储第一个节点(后续要修改节点)
        Node f=first;
        //2.创建新节点
        Node newNode=new Node(null, element, f);
        //3.设置新的头节点
        first=newNode;
        //4.对首届点进行判定
        if(f==null) {
            last=newNode;
        }else {
            f.prev=newNode;
        }
        //5.更新size的值
        size++;
    }

    //在链表尾部添加新节点
    public void addLast(Object element) {
        Node l=last;
        Node newNode=new Node(l, element,null);
        last=newNode;
        if(l==null) {
            first=newNode;
        }else {
            l.next=newNode;
        }
        size++;
    }

    //在指定位置添加节点
    public void add(int index,Object element) {
        if(index<0||index>size)
            throw new IndexOutOfBoundsException();
        if(index==size) {
            addLast(element);
        }else {//index>=0 and index<size
            //seach node
            Node searchNode=node(index);
            Node pred=searchNode.prev;
            Node newNode=new Node(pred, element, searchNode);
            searchNode.prev=newNode;
            if(pred==null) {
                first=newNode;
            }else {
                pred.next=newNode;
            }
            size++;
        }
```

```
    }

    //search node
    Node node(int index) {
        if(index<(size>>1)) {
            Node x=first;
            for(int i=0;i<index;i++) {//从前向后
                x=x.next;
            }
            return x;
        }else {
            Node x=last;
            for(int i=size-1;i>index;i--) {//从后向前
                x=x.prev;
            }
            return x;
        }
    }

    //定义删除头节点的方法
    public Object removeFirst() {
        //1.获取头节点
        Node f=first;
        if(f==null)
            throw new NoSuchElementException();
        //2.获取头节点的数据
        Object element=f.element;
        //3.获取头节点的下一个节点，并修改此节点为头节点
        Node next=f.next;
        f.element=null;//可以更好的进行gc;
        f.next=null;
        first=next;
        if(next==null) {
            last=null;
        }else {
            next.prev=null;//first
        }
        //4.更新size的值
        size--;
        return element;
    }

    //定义移除尾节点的方法
    public Object removeLast() {
        //1.获取尾节点
        Node l=last;
        if(l==null)
            throw new NoSuchElementException();
        //2.获取最后节点的element值
        Object element=l.element;
        //3.删除，修改最后节点
        Node prev=l.prev;
```

```
        l.element=null;
        l.prev=null;
        last=prev;
        if(prev==null)first=null;
        else prev.next=null;
        //4.修改size的值
        size--;
        return element;
    }

    //按指定位置移除元素节点
    public Object removeObject(int index) {
        if(index<0||index>=size)
            throw new IndexOutOfBoundsException();
        return unlink(node(index));
    }

    //从链表中卸载某个节点
    public Object unlink(Node x) {
        //1.存储节点元素值;
        Object element=x.element;
        //2.获取当前节点的下一个节点和上节点
        Node next=x.next;
        Node prev=x.prev;
        //3.修改节点指向
        if(prev==null) {
            first=next;
        }else {
            prev.next=next;
            x.prev=null;
        }
        if(next==null) {
            last=prev;
        }else {
            next.prev=prev;
            x.next=null;
        }
        //修改size的值;
        size--;
        x.element=null;
        return element;
    }
    //按照对象元素移除节点
    public boolean removeObject(Object element) {
        if(element==null) {
            for(Node x=first;x!=null;x=x.next) {
                if(x.element==null) {
                    unlink(x);
                    return true;
                }
            }
        }else {
            for(Node x=first;x!=null;x=x.next) {
```

```
                if(x.element.equals(element)) {
                    unlink(x);
                    return true;
                }
            }
        }
        return false;
    }
    public static void main(String[] args) {
        SimpleDoubleLinkedList list=new SimpleDoubleLinkedList();
        list.addFirst(100);//first=100,last=100
        list.addFirst(200);//first=200,last=100;
        list.addFirst(300);//first=300,last=100
        System.out.println(list);//[300,200,100];

        list.addLast(400);
        System.out.println(list);//[300,200,100,400];
        list.add(1, 500);
        System.out.println(list);//[300,500,200,100,400];
        list.removeFirst();
        System.out.println(list);//[500,200,100,400];
        list.removeLast();
        System.out.println(list);//[500,200,100];
        list.removeObject(1);
        System.out.println(list);//[500,100];
        list.removeObject((Object)100);
        System.out.println(list);//[500];

    }



    @Override
    public String toString() {
        StringBuilder sb=new StringBuilder("[");
        Node node=first;
        while(node!=null) {
            sb.append(node.element).append(",");
            node=node.next;
        }
        if(sb.length()>1)sb.deleteCharAt(sb.length()-1);
        sb.append("]");
        return sb.toString();
    }


}
```

3.13 基于 JAVA 技术如何实现链表的反转？

需求:初始链表为 Input,然后执行 reverse 后为 Output 数组结果

```
Input :  list = 1<=>2<=>3<=>4<=>5
reverse
Output : list = 5<=>4<=>3<=>2<=>1
```

在单项链表中添加如下反转方法:

```
//实现元素的倒置
public Node reverse(Node curr,Node prev) {
    if(curr.next==null) {
        head=curr;
        curr.next=prev;
        return head;
    }
    Node newPre=curr.next;
    curr.next=prev;
    reverse(newPre, curr);
    return head;
}
```

在双向链表中添加反转方法:

```
public void reverse() {
    Node temp=null;
    Node current=first;
    while(current!=null) {
        temp=current.prev;
        current.prev=current.next;
        current.next=temp;
        current=current.prev;
    }
    if(temp!=null) {
        first=temp.prev;
    }
}
```

创建测试类，进行链表倒置测试。

```
package com.cy.pj.ds.linked;
public class SimpleLinkedListReverse {
    public static void doTestSingleListReverse() {
        SimpleSingleLinkedList list=new SimpleSingleLinkedList();
        list.addLast(1);
        list.addLast(2);
        list.addLast(3);
        list.addLast(4);
```

```
        list.addLast(5);
        System.out.println(list);//1->2->3->4->5
        list.reverse(list.getHead(),null);
        System.out.println(list);//5->4->3->2->1
    }
    public static void doTestDoubleListReverse() {
        SimpleDoubleLinkedList list=new SimpleDoubleLinkedList();
        list.addLast(1);
        list.addLast(2);
        list.addLast(3);
        list.addLast(4);
        list.addLast(5);
        System.out.println(list);//1->2->3->4->5
        list.reverse();
        System.out.println(list);//5->4->3->2->1
    }
    public static void main(String[] args) {
        //doTestSingleListReverse();
        doTestDoubleListReverse();
    }

}
```

3.14 链表的优势和劣势是什么？

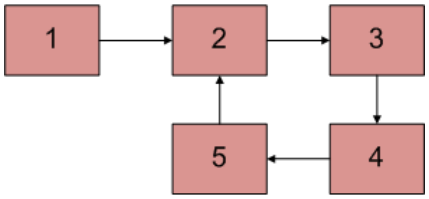
链表本身没有大小的限制，内存无需连续，天然地支持动态扩容，所以插入或删除性能相对较好，尤其是双向链表。但链表的随机访问性能相对较差，因为链表中的数据并非连续存储的，所以无法像数组那样，根据首地址和下标，通过寻址公式就能直接计算出对应的内存地址，而是需要根据指针一个结点一个结点地依次遍历，直到找到相应的结点。

3.15 双向循环链表相对于单向链表的优势是什么？

双向链表相对于单向链表最大的优势就是节点的查找速度。对于单链表而言删除某个结点，需要知道其前驱结点 b，而单链表并不支持直接获取前驱结点，所以，为了找到前驱结点，我们还是要从头结点开始遍历链表，直到 p->next=b，说明 p 是 b 的前驱结点。但是对于双向链表来说，这种情况就比较有优势了。因为双向链表中的结点已经保存了前驱结点的指针，不需要像单链表那样遍历。所以，单链表删除操作需要 O(n)的时间复杂度，而双向链表只需要在 O(1)的时间复杂度内就搞定了！

3.16 如何判断链表中是否有环？

需求：定义初始链表，判断如下链表是否有环，并计算环的长度



在单向链表中添加如下如方法：

```
    //计算环中节点个数
    int countNodes(Node node) {
        int count=1;
        Node temp=node;
        while(temp.next!=node) {
            count++;
            temp=temp.next;
        }
        return count;
    }
```

```
    //测试是否有环
    boolean detactLoop() {
        Node slow=head,fast=head;
        while(slow!=null&&fast!=null&&fast.next!=null) {
            slow=slow.next;
            fast=fast.next.next;
            if(slow==fast) {
                System.out.println("Found
loop ,nodes.size="+countNodes(slow));
                return true;
            }
        }
        return false;
    }
```

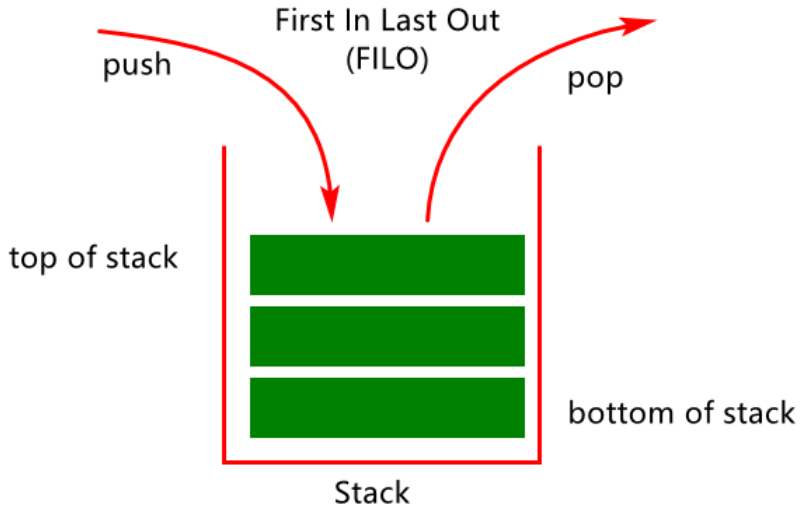
添加测试方法，例如：

```
static void doTestContainsLoop() {
    SimpleSingleLinkedList list=
            new SimpleSingleLinkedList();
    list.addFirst(100);
    list.addFirst(200);
    list.addFirst(300);
    list.head.next.next.next=list.head;
    System.out.println(list.detectLoop());
}
```

4 栈(Stack)结构应用分析

4.1什么是栈(Stack)?

栈（Stack）是一种先进后出(FILO-First In Last Out)，操作上受限的线性表。其限制指的是，仅允许在表的一端进行插入和删除运算。这一端称为栈顶（top），相对地，把另一端称为栈底(bottom)。如图所示：



对于栈而言，我们生活中也有很多这样的应用，例如一摞叠在一起的盘子。我们平时放盘子的时候，都是从下往上一个一个放；取的时候，我们也是从上往下一个一个地依次取，不能从中间任意抽出。后进者先出，先进者后出，这就是典型的“栈”结构。

4.2栈(Stack)有哪些应用场景？

实际的软件项目中很多地方都会用到这种栈结构，例如：

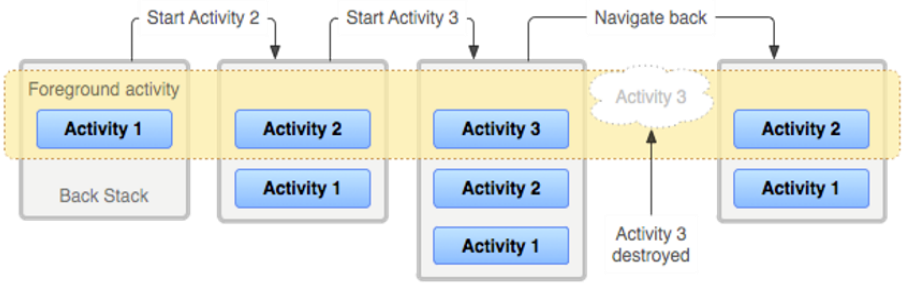
1）Java 中虚拟机内部方法调用栈。

2）运算表达式的语法分析，词法分析。

3）浏览器内置的回退栈(back stack)。

4）手机中 APP 的回退栈(back stack)。

我们现在以手机中的 APP 为例进行分析，如图所示：



在 android 手机上我们每打开一个 app 都会创建一个回退栈，栈中存储每次打开的界面对象，新打开的 UI 界面会处于栈顶。

4.3基于 Java 定义栈结构规范？

```
package com.cy.pj.ds.stack;
/**栈接口规范的定义*/
public interface Stack {
    /**
     * 压栈
     * @param item
     */
    void push(Object item);
    /**
     * 出栈
     * @return
     */
    Object pop();
    /**
     * 获取栈顶元素，但不出栈。
     * @return
     */
    Object peek();
    /**
     * 获取栈中有效元素个数
     * @return
     */
    int size();
    /**
     * 判定栈是否为空
     * @return
     */
    boolean isEmpty();
}
```

4.4基于 Java 数组实现栈(Stack)?

```
/**
 * 基于数组实现栈结构中的数据操作
 * @author qilei
 */
public class BoundedArrayStack implements Stack {

    private Object[] array;
    private int size;
    public BoundedArrayStack(int capacity) {
        this.array=new Object[capacity];
    }
    @Override
    public void push(Object item) {
        if(size==array.length)
            throw new IllegalStateException("Stack is full");
        this.array[size++]=item;
    }

    @Override
    public Object pop() {
        if(size==0)
            throw new NoSuchElementException("Stack is empty");
        Object result=array[size-1];
        array[--size]=null;
        return result;
    }

    @Override
    public Object peek() {
        if(size==0)
            throw new NoSuchElementException("Stack is empty");
        return array[size-1];//栈顶元素
    }

    @Override
    public int size() {
        return size;
    }
    @Override
    public boolean isEmpty() {
        return size==0;
    }
    public static void main(String[] args) {
        BoundedArrayStack stack=new BoundedArrayStack(3);
        stack.push(100);
        stack.push(200);
        stack.push(300);
        System.out.println(stack.peek());
        //stack.push(400);
```

```
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.isEmpty());
    }

}
```

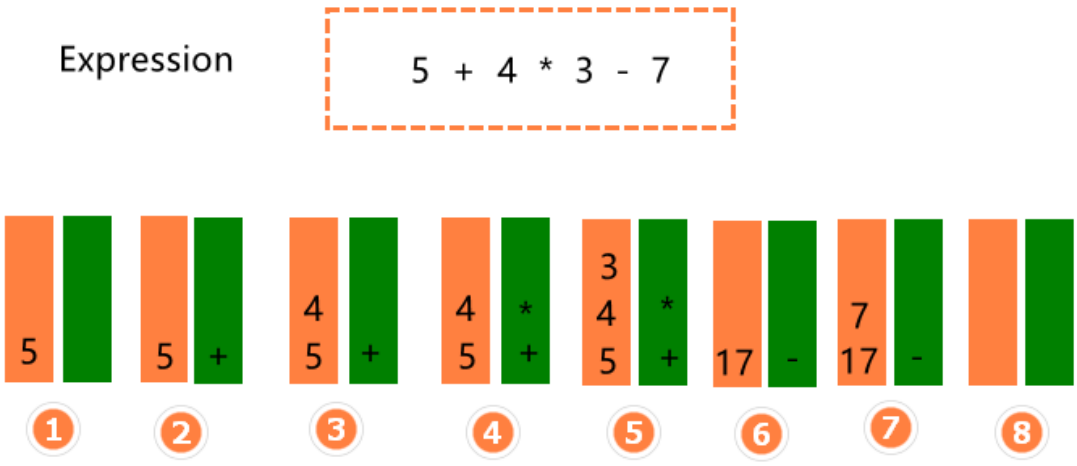
4.5基于 Java 链表实现栈(Stack)?

```
public class LinkedStack implements Stack {

    private Node top=null;
    class Node{
        private Object data;
        private Node next;
        public Node(Object data,Node next) {
            this.data=data;
            this.next=next;
        }
    }
    @Override
    public void push(Object item) {
        //新节点为栈顶元素
        top=new Node(item, top);
    }

    @Override
    public Object pop() {
        Object item=peek();
        top=top.next;
        return item;
    }

    @Override
    public Object peek() {
        if(top==null)throw new NoSuchElementException("Stack is empty");
        return top.data;
    }

    @Override
    public int size() {
        int count=0;
        Node node=top;
        while(node!=null) {
            node=node.next;
            count++;
        }
```

```
        return count;
    }
    @Override
    public boolean isEmpty() {
        return top==null;
    }
    public static void main(String[] args) {
        LinkedStack stack=new LinkedStack();
        stack.push(100);//栈底
        stack.push(200);
        stack.push(300);//栈顶
        System.out.println(stack.size());
        System.out.println(stack.peek());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        //System.out.println(stack.pop());
        System.out.println(stack.isEmpty());
    }
}
```

4.6如何基于栈实现表达式求值？

栈是一种重要的数据结构，而表达式求值是程序设计语言编译中的一个基本问题，编译系统通过栈对表达式进行语法分析、词法分析，最终获得正确的结果。例如，在使用栈进行表达式计算时，一般要设计两个栈，其中一个用来保存操作数，另一个用来保存运算符。我们从左向右遍历表达式，当遇到数字，我们就直接压入操作数栈；当遇到运算符，就与运算符栈的栈顶元素进行比较，若比运算符栈顶元素优先级高，就将当前运算符压入栈，若比运算符栈顶元素的优先级低或者相同，从运算符栈中取出栈顶运算符，从操作数栈顶取出 2 个操作数，然后进行计算，把计算完的结果压入操作数栈，继续比较。如图所示：



4.7如何基于栈实现函数调用实践？

操作系统给每个线程分配了一块独立的内存空间，这块内存被组织成“栈”这种结构，用来存储函数调用时的临时变量。每进入一个函数，就会将其中的临时变量作为栈帧入栈，当被调用函数执行完成，返回之后，将这个函数对应的栈帧出栈。

```
StackTraceElement[] stackTrace =
        Thread.currentThread().getStackTrace();

exception.printStackTrace();
```

4.8如何基于栈实现括号匹配分析？

在进行括号匹配的语法校验时,可以用栈保存匹配的左括号,从左到右一次扫描字符串，当扫描到左括号时，则将其压入栈中。当扫描到右括号时，从栈顶取出一个左括号，如果两个括号能匹配上，则继续扫描剩下的字符串。如果扫描过程中，遇到不能配对的右括号，或者栈中没有数据，则说明为非法格式。当所有的括号都扫描完成之后，如果栈为空，则说明字符串为合法格式；否则，说明未匹配的左括号为非法格式。例如：

```
    static boolean isMatching(String expression){
    Stack stack = new BoundedArrayStack(10) ;
    for(int index=0 ; index<expression.length();index++){
        char c=expression.charAt(index);
        switch(expression.charAt(index)){
            case '(':stack.push(c) ; break ;
            case '{':stack.push(c) ; break ;
            case ')':
                if(!stack.isEmpty()
                        &&stack.peek()==(Character)'(') {
                    stack.pop() ;
                }
                break ;
            case '}':
                if(!stack.isEmpty()
                        &&stack.peek()==(Character)'{'){
                    stack.pop();
                }
        }
    }
    if(stack.isEmpty())return true ;
    return false ;
}
```

4.9手机 APP 中回退栈是如何应用的？

在 android 手机上我们每打开一个 app 都会创建一个回退栈，栈中存储每次打开的界面对象，新打开的 UI 界面会处于栈顶，当我们点击手机上的回退操作时，会移除栈顶元素，将后续元素作为栈顶，然后进行激活。

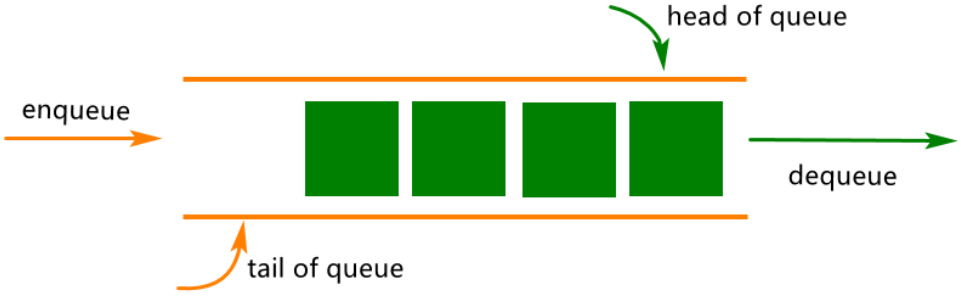
5 队列(Queue)结构应用分析

5.1什么是队列？

队列（Queue）这种结构非常好理解。你可以把它想象成超市中排队结账，排在前面的先结账出队，排在后面的后结账出队。后来的人只能站在末尾，不允许插队。如图所示：

类似超市排队结账,还有地铁,机场,火车站排队进站等这种满足先进者先出结构，就是我们要探讨的典型的“队列”。

队列（Queue）跟栈（Stack）非常相似，也是操作受限的一种逻辑结构，最基本的操作也是两个：入队(enqueue)，放一个数据到队列尾部；出队(dequeue)，从队列头部取一个元素。如图所示：



5.2队列有哪些应用场景？

队列的应用非常广泛，特别是一些具有某些额外特性的队列，比如循环队列、阻塞队列、并发队列。它们在很多偏底层的系统、框架、中间件的开发中，起着关键性的作用。比如高性能队列 Disruptor，Linux 环形缓存，都用到了循环并发队列；Java concurrent 并发包中 BlockingQueue 基于阻塞特性实现公平，非公平等特性。

5.3JAVA 中常见队列有哪些？

单端队列：只支持一端入队(enqueue)，一端出队（dequeue）。
双端队列：支持队列的两端进行入队和出队操作。
循环队列：可提供更好的性能，降低时间复杂度。
阻塞队列：生产者和消费者应用模型中的一种容器，在队列空或满的时候进行阻塞。
优先级队列：支持按优先级操作的的队列结构（内部对元素进行排序）。
….。

5.4Java 中队列规范的定义？

```
package com.cy.pj.ds.queue;

public interface Queue {

    void enqueue(Object element);
    Object dequeue();
    int size();
    boolean isEmpty();
}
```

5.5基于 Java 数组如何实现队列？

```
package com.cy.pj.ds.queue;
/**
 * 基于数组结构进行队列实现
 */
public class SimpleArrayQueue implements Queue {
```

```
    private Object[] array;
    private int size;

    public SimpleArrayQueue() {
        this(10);
    }
    public SimpleArrayQueue(int capacity) {
        this.array=new Object[capacity];
    }

    @Override
    public void enqueue(Object element) {
        //1.队列是否已满，满了则抛出异常
        if(size==array.length)
            throw new IllegalStateException("Queue is full");
        //2.存储数据
        array[size]=element;
        //3.有效元素个数加1
        size++;
    }

    @Override
    public Object dequeue() {
        //1.判断队列是否为空
        if(size==0)
            throw new IllegalStateException("Queue is empty");
        //2.获取对头元素
        Object temp=array[0];
        //3.移动元素
//    for(int i=0;i<array.length-1;i++) {
//       array[i]=array[i+1];
//    }
        System.arraycopy(array, 1, array, 0, size-1);
        //4.设置size-1位置元素为空
        array[size-1]=null;
        //5.有效元素个数减一
        size--;
        return temp;
    }
    @Override
    public int size() {
        return size;
    }
    @Override
    public boolean isEmpty() {
        return size==0;
    }
    public static void main(String[] args) {
        SimpleArrayQueue queue=new SimpleArrayQueue(3);
        queue.enqueue(100);
        queue.enqueue(200);
        queue.enqueue(300);
        System.out.println(queue.size());
```

```
        System.out.println(queue.dequeue());//FIFO 100
        System.out.println(queue.dequeue());//FIFO 200
        System.out.println(queue.dequeue());//FIFO 300
        System.out.println(queue.isEmpty());
    }
}
```

每次出队的时候，数组的元素整体往左移动，这样队列出队的时间复杂度就为 O(N)，那么有什么办法可以降低队列出队操作的时间复杂度吗？

5.6基于 Java 链表如何实现队列？

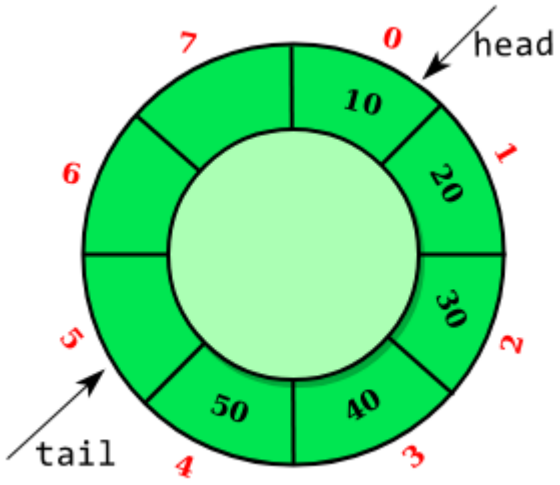
```
package com.cy.pj.ds.queue;
/**
 * 基于单链表结构进行基础队列的实现
 * @author qilei
 */
public class SimpleLinkedQueue implements Queue {

    private Node head;
    private Node tail;
    class Node{
        private Object element;
        private Node next;
        public Node(Object element,Node next) {
            this.element=element;
            this.next=next;
        }
    }
    @Override
    public void enqueue(Object element) {
        if(head==null) {
            head=tail=new Node(element,null);
        }else {
            Node newNode=new Node(element, null);
            tail.next=newNode;
            tail=newNode;
        }
    }
    @Override
    public Object dequeue() {
        if(head==null)return null;
        Node oldHead=head;
        head=oldHead.next;
        Object element=oldHead.element;
        oldHead.next=null;
        oldHead.element=null;
        return element;
```

```
    }
    @Override
    public int size() {
        int count=0;
        Node node=head;
        while(node!=null) {
            node=node.next;
            count++;
        }
        return count;
    }
    @Override
    public boolean isEmpty() {
        //return size==0;
        return head==null;
    }
    public static void main(String[] args) {
        SimpleLinkedQueue queue=new SimpleLinkedQueue();
        queue.enqueue(100);
        queue.enqueue(200);
        queue.enqueue(300);
        System.out.println(queue.size());
        System.out.println(queue.dequeue());//100
        System.out.println(queue.dequeue());//200
        System.out.println(queue.dequeue());//300
        System.out.println(queue.isEmpty());
    }
}
```

基于链表的实现方式，可以实现一个支持无限排队的无界队列，但是可能会导致过多的请求排队等待，请求处理的响应时间过长。所以，针对响应时间比较敏感的系统，基于链表实现是不合适的(例如线程池)。

5.7 什么是循环队列？

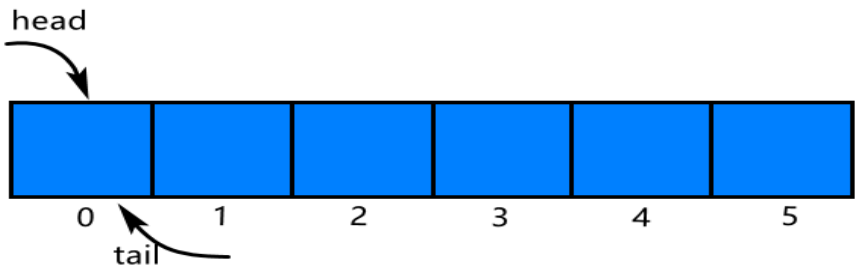
循环队列是让队列形成一种环形结构，它以循环的方式去存储元素，但还是会按照队列的先进先出的原则去操作，如图所示：



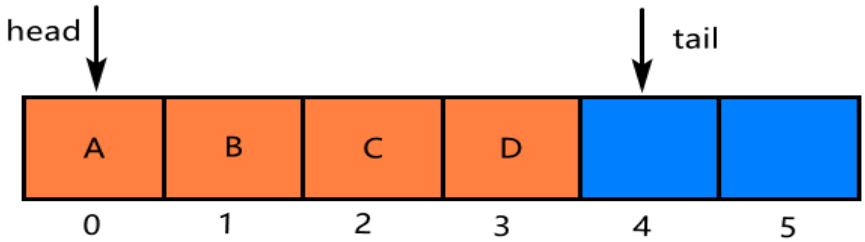
5.8为什么要使用循环队列？

在基础队列应用中，基于数组方式实现的简易队列，我们发现一个问题，每次出队都会涉及到数组中元素的移动，时间的复杂度比较高，那如何进行优化呢？

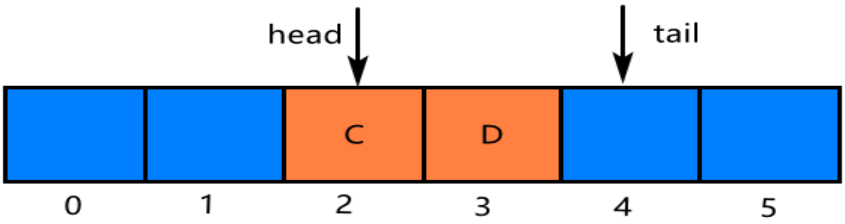
我们现在为队列添加两个变量,它们分别为 head 和 tail，其初始值都为下标 0，都指向数组中的第一个元素，如图所示：



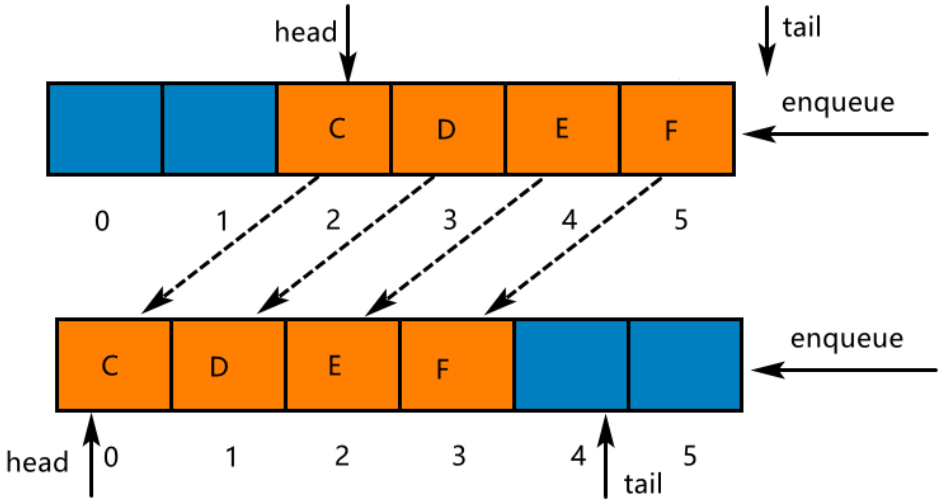
现在我们向队列添加 A,B,C,D 四个元素，每添加一个元素 tail 的下标就向后移动一个位置，当四个元素添加完毕，此时 tail 移动到下标为 4 的位置。如图所示：



当从队列中出队一个元素，head 的下标也会向后移动一个位置，假设现在出队 A，B 两个元素，此时 head 的位置，如图所示：



随着不停地进行入队、出队操作，head 和 tail 都会持续往后移动。当 tail 移动到最右边，即使数组中还有空闲空间，也无法继续往队列中添加数据了。那我们怎么办呢？那就是移动元素，如图所示：



通过这样的设计可以适当减少元素移动次数，出队操作的时间复杂度仍然是 O(1)，但入队操作的时间复杂度不是O(1).那我们还能继续优化吗？此时可以借助”循环队列”，降低时间复杂度，减少队列中元素的移动，充分利用队列空间。

5.9基于 Java 如何实现循环队列？

循环队列通过 head 和 tail 两个变量操作入队和出队，假如“head==tail”则表示队列为空，“head=(tail+1)%capacity”则表示队列已满。当进行入队时 tail=(tail+1)%capacity，出队时 head=(head+1)%capacity。其详细代码如下：

```
package com.cy.pj.ds.queue;
/**
 * 循环队列实现
 * @author qilei
 *
 */
public class CircleArrayQueue implements Queue{

    private Object[] array;
    private int capacity;
    private int head;
    private int tail;
```

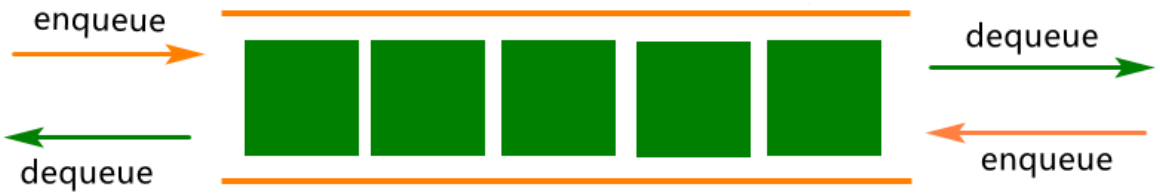
```
public CircleArrayQueue(int capacity) {
    this.capacity=capacity+1;
    this.array=new Object[this.capacity];
}
@Override
public void enqueue(Object element) {
    //1.入队操作
    array[tail]=element;
    //2.修改tail变量的值，并进行判定
    int newTail=(tail+1)%this.capacity;
    if(newTail==head)
        throw new IllegalStateException("Queue is full");
    tail=newTail;
}

@Override
public Object dequeue() {
    if(head==tail)
        throw new IllegalStateException("Queue is empty");
    Object temp=array[head];
    array[head]=null;
    head=(head+1)%capacity;
    return temp;
}

@Override
public int size() {
    int count=tail-head;
    if(count<0)
        count+=this.capacity;
    return count;
}

@Override
public boolean isEmpty() {
    return head==tail;
}

public static void main(String[] args) {
    CircleArrayQueue queue=new CircleArrayQueue(3);
    System.out.println("head="+queue.head);//0
    System.out.println("tail="+queue.tail);//0
    queue.enqueue(100);
    queue.enqueue(200);
    queue.enqueue(300);
    System.out.println(queue.size());//3
    System.out.println("head="+queue.head);//0
    System.out.println("tail="+queue.tail);//3
    System.out.println(queue.dequeue());//100
    System.out.println("head="+queue.head);//1
    System.out.println("tail="+queue.tail);//3
    queue.enqueue(400);
    System.out.println("head="+queue.head);//1
```

```
        System.out.println("tail="+queue.tail);//0

    }
}
```

循环队列中随着不断入队操作的执行，tail 指向了队尾的后一个位置，也就是新元素将要被插入的位置，如果该位置和 head 相等了，那么必然说明当前状态已经不能容纳一个元素入队（间接的说明队满）。因为这种情况是和队空(head==tail)的判断条件是一样的，所以我们选择舍弃一个节点位置，tail 指向下一个元素的位置，我们使用 tail+1 判断下一个元素插入之后，是否还能再加入一个元素，如果不能了说明队列满，不能容纳当前元素入队（其实还剩下一个空位置），当然这是牺牲了一个节点位置来实现和判断队空的条件进行区分。

5.10 什么是双端队列？

双端队列（Double-ended queue）是一种特殊的队列，简称为 Deque。支持队列两端的入队和出队操作。同时具备了栈(Stack)和队列(Queue)特性,如图所示：



5.11 为什么会有双端队列？

双端队列在很多场景都有应用，Java 中 ForkJoin 模式下的工作窃取(允许其它线程从自己的线程队列尾部获取任务、执行任务)

5.12 基于 Java 链表如何实现双端队列？

第一步:定义队列接口

```
package com.cy.pj.ds.queue;
/**
 * 双端队列接口
 * @author qilei
```

```
 */
public interface Deque {
    void addFirst(Object element);
    void addLast(Object element);
    Object removeFirst();
    Object removeLast();
    int size();
    boolean isEmpty();
    //....
}
```

第二步：定义队列接口实现

```
package com.cy.pj.ds.queue;
/**
 * 简易双端队列实现
 * @author qilei
 */
public class SimpleLinkedDeque implements Deque {

    transient Node first;
    transient Node last;
    transient int  size;
    static class Node{
        private Object element;
        private Node next;
        private Node prev;
        public Node(Node prev,Object element,Node next) {
            this.prev=prev;
            this.element=element;
            this.next=next;
        }
    }

    @Override
    public void addFirst(Object element) {
        //1.获取第一个节点
        Node oldFirst=first;
        //2.创建新节点
        Node newNode=new Node(null, element, oldFirst);
        //3.设置第一个节点
        first=newNode;
        if(oldFirst==null) {
            last=newNode;
        }else {
            oldFirst.prev=newNode;
        }
        //4.修改 size 的值
        size++;
    }
```

```
@Override
public void addLast(Object element) {
    //1.获取最后一个节点
    Node oldLast=last;
    //2.创建新节点
    Node newNode=new Node(oldLast, element, null);
    //3.设置最后一个节点
    last=newNode;
    if(oldLast==null) {
        first=newNode;
    }else {
        oldLast.next=newNode;
    }
    //4.更新size的值
    size++;
}

@Override
public Object removeFirst() {
    //1.获取第一个节点并进行判定
    Node oldFirst=first;
    if(oldFirst==null)
        throw new IllegalStateException("Queue is empty");
    //2.修改第一个节点并设置新的first节点
    Object temp=oldFirst.element;
    Node newFirst=oldFirst.next;
    oldFirst.element=null;
    oldFirst.next=null;
    //oldFirst.prev=null;
    first=newFirst;
    if(newFirst==null) {
        last=null;
    }else {
        newFirst.prev=null;
    }
    //3.更新size的值
    size--;
    return temp;
}

@Override
public Object removeLast() {
    //1.获取last节点并进行校验
    Node oldLast=last;
    if(oldLast==null)
        throw new IllegalStateException("Queue is empty");
    //2.修改last节点并设置新的last
    Object temp=oldLast.element;
    Node newLast=oldLast.prev;
    oldLast.element=null;//help gc
    oldLast.prev=null;
    last=newLast;
    if(newLast==null) {
```

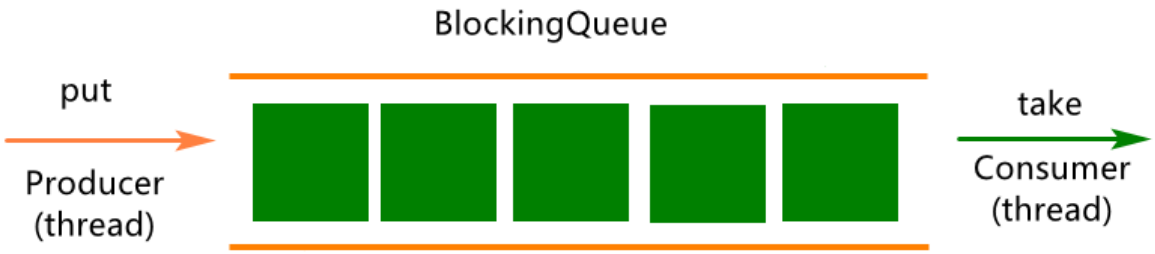
```
            first=null;
        }else {
            newLast.next=null;
        }
        //3.更新size的值
        size--;
        return temp;
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean isEmpty() {
        return size==0;
    }
    public static void main(String[] args) {
        SimpleLinkedDeque dq=new SimpleLinkedDeque();
        dq.addLast(100);//first
        dq.addLast(200);
        dq.addLast(300);//last
        System.out.println(dq.removeFirst());//100
        System.out.println(dq.removeFirst());//200
        System.out.println(dq.removeFirst());//300
        //=========================
        dq.addFirst(400);
        dq.addFirst(500);
        System.out.println(dq.removeLast());
        System.out.println(dq.removeLast());


    }

}
```

5.13 什么是阻塞式队列?

阻塞(Blocking)式队列，顾名思义，首先它是一个队列（Queue），然后在这个队列中加入了阻塞(Blocking)式功能（例如去饭店吃饭，满员了可在等候区排队等待）。

5.14 为什么需要阻塞式队列?

阻塞式队列（BlockingQueue）经常应用于生产者和消费者模式，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。当队列中没有数据的情况下，消费端的所有线程都会被自动阻塞（挂起），直到有数据放入队列。当队列中填满数据的情况下，生产端的所有线程都会被自动阻塞（挂起），直到队列中有空的位置，线程被自动唤醒。如图所示：



5.15 基于 Java 如何实现阻塞队列？

```
package com.cy.pj.ds.queue;
/**
 * 阻塞式队列实现
 */

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class SimpleArrayBlockingQueue {
    //store data
    Object[] array;
    //index for dequeue
    int takeIndex;
    //index for enqueue
    int putIndex;
    //number of elements
    int size;
    //lock guarding all access
    ReentrantLock lock;
    //Condition for dequeue
    Condition takeCondition;
    ////Condition for enqueue
    Condition putCondition;

    public SimpleArrayBlockingQueue(int capacity) {
        if(capacity<0)
            throw new IllegalArgumentException();
        this.array=new Object[capacity];
        lock=new ReentrantLock();
        takeCondition=lock.newCondition();
        putCondition=lock.newCondition();
    }
```

```
    //enqueue
    public void put(Object element) throws InterruptedException {
        //1.获取锁对象
        final ReentrantLock lock=this.lock;
        lock.lockInterruptibly();
        try {
            //2.校验容器是否已满，满了则等待
            while(size==array.length)putCondition.await();
            //3.存储数据
            array[putIndex]=element;
            if(++putIndex==array.length)putIndex=0;
            size++;
            //4.通知消费者取数据
            takeCondition.signalAll();;
        }finally {
            //5.释放锁
            lock.unlock();
        }
    }
    //dequeue
    public Object take() throws InterruptedException {
        //1.获取锁对象
        final ReentrantLock lock=this.lock;
        lock.lockInterruptibly();
        try {
            //2.检测容器是否为空，空则等待
            while(size==0)takeCondition.await();
            //3.移除元素
            Object element=array[takeIndex];
            array[takeIndex]=null;
            if(++takeIndex==array.length)takeIndex=0;
            size--;
            //4.通知生产者放数据
            putCondition.signalAll();
            return element;
        }finally {
            //5.方法锁
            lock.unlock();
        }
    }
    public int size() {
        final ReentrantLock lock=this.lock;
        lock.lock();
        try {
            return size;
        }finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        SimpleArrayBlockingQueue q=new SimpleArrayBlockingQueue(3);
        new Thread() {//Producer
```

```
            int i=100;
            public void run() {
                try {
                    while(true) {
                        q.put(i++);
                        try{Thread.sleep(500);}catch(Exception e) {}
                    }
                }catch (Exception e) {
                    e.printStackTrace();
                }
            };
        }.start();
        new Thread() {//Consumer
            public void run() {
                try {
                    while(true) {
                        System.out.println(q.take());
                    }
                }catch (Exception e) {
                    e.printStackTrace();
                }
            };
        }.start();
    }
}
```

在 Java 的 JUC 包中，提供了很多基于阻塞方式实现的队列，BlockingQueue 接口是一种阻塞式队列接口，基于此接口的实现类对象解决了高效、安全“传输”数据的问题。通过这些高效并且线程安全的队列类，为我们快速搭建高质量的多线程程序带来极大的便利。

5.16 Java 中的 ArrayBlockQueue 如何实现的？

ArrayBlockingQueue 是一个有边界的阻塞队列，它的内部实现是一个数组。它的容量是有限的，我们必须在其初始化的时候指定它的容量大小，容量大小一旦指定，其大小不可改变。其内部的阻塞方式是通过重入锁 ReenterLock 和 Condition 条件队列实现的，但是队列中的锁是没有分离的，即添加操作和移除操作采用的同一个 ReenterLock 锁，这样就会导致入队和出队操作不能同时进行。

5.17 Java 中 LinkedBlockingQueue 如何实现的?

LinkedBlockingQueue 采用的是一种基于单链表实现的阻塞式无界队列。此队列在添加一个元素时会创建一个新的 Node 对象。删除一个元素时要移除一个节点对象。频发的创建和销毁可能对 GC 操作有较大影响。但是，此队列中的锁(Lock)是分离的，其添加操作采用的是 putLock，移除操作采用的则是 takeLock，这样能大大提高队列的吞吐量，也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。

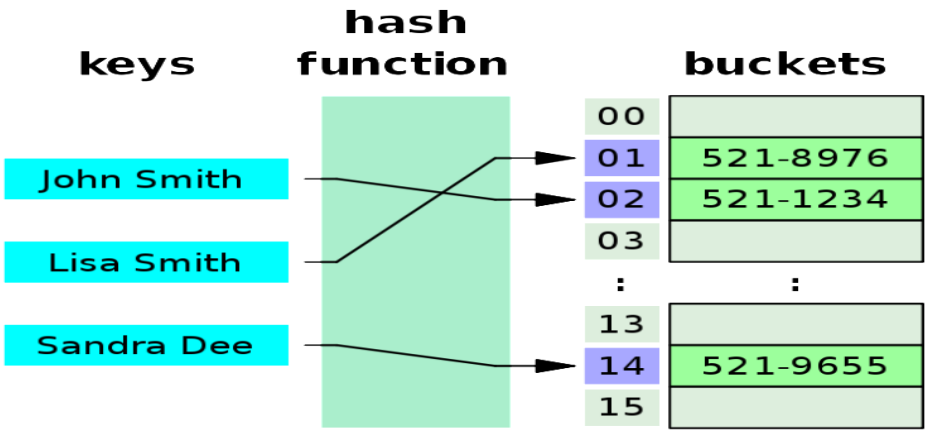
5.18 Java 中 ConcurrentLinkedQueue 如何实现的？

ConcurrentLinkedQueue 是一个基于单链表实现的、线程安全的、非阻塞式无界队列。此队列的设计也非常考验设计功底，其内部全程使用了 cas 操作，并且在边界控制方面也引入了哨兵机制。总之，设计复杂程度远远高于直接使用锁(Lock)对象方式的线程安全队列的实现。

6 散列(Hash)基础分析

6.1什么是散列表？

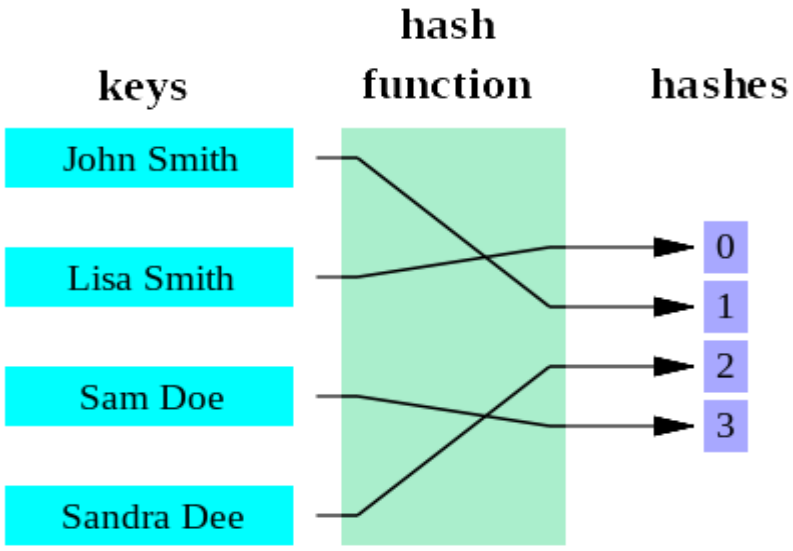
散列表又称哈希表(Hash Table)，是一种将键(key)映射到值的数据结构，是对数组应用的推广，它基于“散列设计算法”将关键码（Key）映射为数组下标，然后将关键码对应的数据存储在数组中。这个过程类似于字典设计(基于字典关键码找到对应的词条)。如图所示：



其中，图中的 buckets 为桶数组(又称“散列表”-hash table)，桶数组中基于桶(bucket)直接存储数据。

6.2如何理解散列设计？

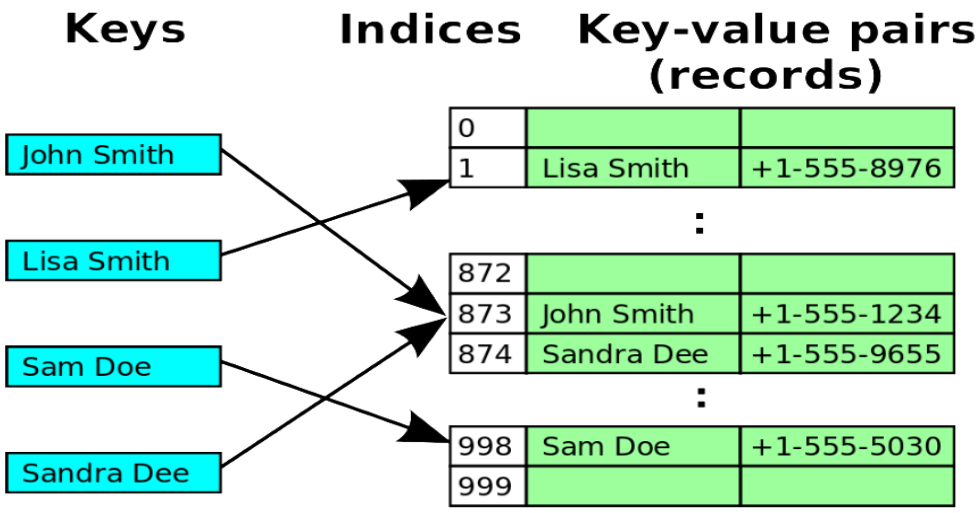
散列设计是一种设计思想，它通过一定的算法将 key 转换为散列表的下标。这种对算法的封装我们称之为“散列函数”，通过散列函数计算得出的值称之为“散列值”(或哈希值)。如下图所示：



其中，图中的 0,1,2,3 为通过散列函数计算得出的哈希值，这些值对应哈希表中的数组下标。我们在设计散列算法时，通常要遵循几个基本原则,例如：

1）散列(Hash)计算得到的散列值应该是一个非负整数；（因为数组的下标从 0 开始）
2）如果 key1 = key2，那 hash(key1) == hash(key2)；（相同 key，得到的散列值也相同）
3）如果 key1 ≠ key2，那 hash(key1) ≠ hash(key2)；（不好确定）

在进行散列设计时，对于 key 不同的计算，应尽量保证 hash 值也不相同，但这样的设计，可能要付出的更多的计算成本，时间成本。所以 key 不同，hash 值相同的这种现象还是会存在的，我们把它称之为散列冲突。如下图所示：

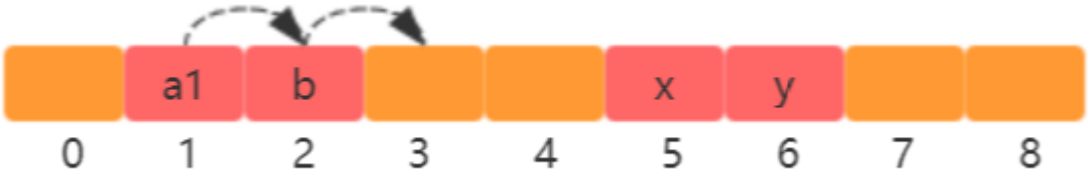


6.3如何解决散列冲突？

散列设计既然无法避免散列冲突，那出现了散列冲突以后，如何应对呢？我们常用的解决方案有两类，开放寻址法（open addressing）和链表法（chaining）。

开放寻址法解决散列冲突：

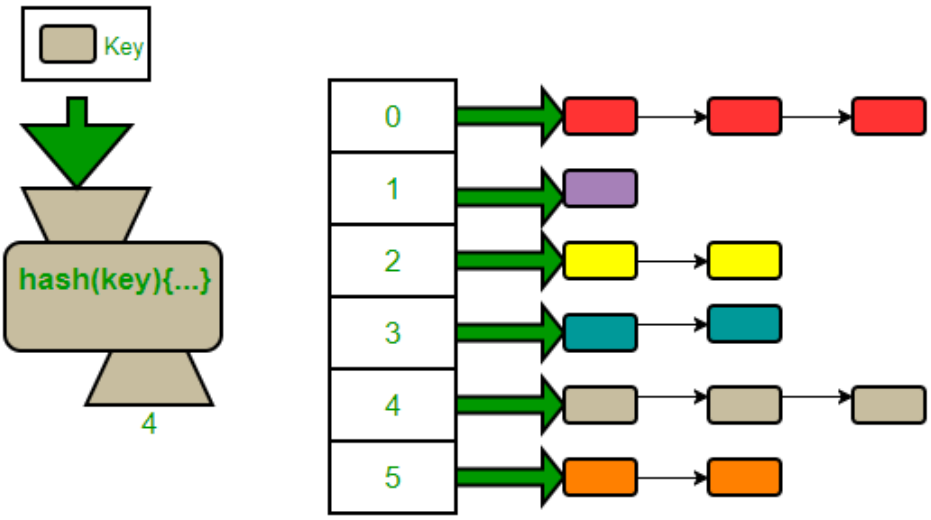
当出现了散列冲突以后，开放寻址是要重新探测一个新的空闲位置，然后将其插入。那如何重新探测新的位置呢？常用的方法有线性探测(Linear Probing)，二次探测（Quadratic probing）和双重散列（Double hashing）等，我们首先来看一下线性探测(Linear Probing)，如下图所示：



在线性探测中，每次探测的步长是 1，那它探测的下标序列依次是 hash(key)+0，hash(key)+1，hash(key)+2……。你可能会发现，此方法其实存在很大问题。当散列表中插入的数据越来越多时，散列冲突发生的可能性就会越来越大，空闲位置会越来越少，线性探测的时间就会越来越久。极端情况下， 我们可能需要探测整个散列表，所以最坏情况下的时间复杂度为 O(n)。对于二次探测，跟线性探测很像，它每次探测的步长就变成了原来的“2 次方”，其探测的下标序列就是 hash(key)+0，hash(key)+1^2 ，hash(key)+2^2 ……。所谓双重散列，意思就是不仅要使用一个散列函数。可能要使用一组散列函数 hash1(key)，hash2(key)，hash3(key)…。总之，开放寻址中，不管采用哪种探测方法，当散列表中空闲位置不多的时候，散列冲突的概率就会大大提高。

链表法解决散列冲突：

当出现了散列冲突以后，链表法相比开放寻址法，它要简单很多。也是一种更加常用的散列冲突解决办法。 在散列表中，每个“桶（bucket）”或者“槽（slot）”会对应一张链表，所有散列值相同的元素，我们都放到相同槽位对应的链表中。如图所示：



我们在散列表中进行数据插入的时，通过散列函数计算出对应的散列槽位，将其插入到对应链表中即可，所以插入的时间复杂度是 O(1)。当删除一个元素时，我们同样通过散列函数计算出对应的槽位， 然后遍历链表找到对应元素进行删除即可。其时间复杂度可能会大一些，例如 O(K)。

6.4Java 中散列应用分析与实践？

基于散列设计思想,实现简易的 HashMap,用于存储多个键值对，代码如下：

```
package com.cy.pj.ds.hash;

import java.util.ArrayList;

/**简易散列表操作实现*/
public class SimpleHashMap {
    //定义链表中节点类型
    class HashNode{
        private Object key;
        private Object value;
        private HashNode next;
        public HashNode(Object key,Object value) {
            this.key=key;
            this.value=value;
        }
    }
    //定义散列表(桶数组)
    private ArrayList<HashNode> bucketArray;
    //定义桶的个数
    private int numBuckets;
```

```
//定义 size 记录有效元素个数
private int size;
//通过构造方法对散列表进行初始化
public SimpleHashMap(int numBuckets) {
    this.numBuckets=numBuckets;
    bucketArray=new ArrayList<>();
    for(int i=0;i<numBuckets;i++) {
        bucketArray.add(null);
    }
}
//定义一个散列函数
public int hash(Object key) {
    int hashCode=key.hashCode();
    return hashCode%numBuckets;
}
//定义数据添加函数
public void put(Object key,Object value) {
    //1.基于 key 获取其散列值(下标值)
    int index=hash(key);
    //2.获取散列表中的桶对象(链表节点)
    HashNode head=bucketArray.get(index);
    //3.检测链表中是否有 key 相同的元素，key 相同值覆盖
    while(head!=null) {
        if(head.key.equals(key)) {
            head.value=value;
            return;
        }
        head=head.next;
    }
    //4.添加新的 key/value 到桶中
    //4.1 获取指定下标对应的桶对象的 head 节点
    head=bucketArray.get(index);
    //4.2 创建新的 node 节点
    HashNode newNode=new HashNode(key, value);
    //4.3 将新节点设置为当前桶中的头节点
    newNode.next=head;
    //4.4 将指定 index 位置的元素设置为新的链表头节点
    bucketArray.set(index, newNode);
    //4.5 执行 size++操作
    size++;
    //5.对散列表进行扩容设计
    if((1.0*size)/numBuckets>=0.8) {
        ArrayList<HashNode> temp=bucketArray;
        numBuckets*=2;//将桶个数设置为原有桶个数的 2 倍
        bucketArray=new ArrayList<>();//新的散列表
        for(int i=0;i<numBuckets;i++) {
            bucketArray.add(null);
        }
        size=0;
        //将原有散列表中的数据拷贝到新的散列表中
        for(HashNode headNode:temp) {
            while(headNode!=null) {
```

```
                put(headNode.key, headNode.value);
                headNode=headNode.next;
            }
        }
    }
}
public Object get(Object key) {
    //1.对key进行散列求值
    int index=hash(key);
    //2.获取index对应的桶
    HashNode head=bucketArray.get(index);
    //3.获取key对应的value值
    while(head!=null) {
        if(head.key.equals(key)) {
            return head.value;
        }
        head=head.next;
    }
    return null;
}
//基于key删除指定元素
public Object remove(Object key) {
    //1.对key进行散列求值
    int index=hash(key);
    //2.获取index对应的桶
    HashNode head=bucketArray.get(index);
    //3.在桶查找key对应的节点，然后进行删除操作
    HashNode prev=null;
    while(head!=null) {
        if(head.key.equals(key))break;
        prev=head;
        head=head.next;
    }
    if(head==null)return null;
    if(prev!=null) {
        prev.next=head.next;
    }else {
        bucketArray.set(index, head.next);
    }
    size--;
    return head.value;
}

public int size() {
    return size;
}
public static void main(String[] args) {
    SimpleHashMap map=new SimpleHashMap(2);
    map.put("this", 1);
    map.put("coder", 2);
    map.put("this", 3);
    map.put("hello", 4);
    map.put("welcome", 5);
```

```
        System.out.println(map.size);
        System.out.println(map.get("coder"));
        map.remove("this");
        System.out.println(map.size);
        System.out.println(map.get("this"));
    }
}
```

6.5如何对散列(Hash)函数进行设计？

对于散列函数的设计，一般要遵循如下几个原则：

- 对于给定的 key，经过散列计算，得到的散列值应该是一个非负整数。
- 对于相同的 key，经过同样的散列计算，应该得到的散列值也相同。
- 对于不同的 key，经过相同的散列计算，得到的散列值应尽量不相同。

除此之外,还要尽量少散列冲突，即使有冲突，也要保证将数据能够均匀的分配到散列表的每个桶内。

6.6数据插入时线性探测过程是怎样的？

当我们向散列表中插入数据时，如果某个数据经过散列计算之后，要进行存储的位置已经被占用了，也就是说出现了散列冲突。此时就需要从当前位置开始，依次向后查找，检查是否有空闲位置，直到找到插入位置为止。

6.7开放寻址有什么优势和劣势？

开放寻址是在散列冲突以后，基于某种策略重新探测新的空闲位置的方法。
- 优势：查询速度快(数据都在数组中),序列化也方便。
- 劣势：数据量越大冲突的几率就越大，探测时间就会越长。

总之，当数据量比较小、装载因子小的时候，适合采用开放寻址法。

6.8散列冲突中链表的解决方案的时间复杂度是多少？

当插入数据的时候，我们需要通过散列函数计算出对应的散列槽位，将其插入到对应的链表中即可，所以插入的时间复杂度为 O(1)。当查找、删除一个元素时，首先需要通过散列函数计算对应的槽位，然后依次遍历链表中的元素。对于散列比较均匀的散列函数，每个桶内的链表的节点个数 k=n/ m，其中 n 表示散列表中数据的个数，m 表示散列表中槽的个数，所以是时间复杂度为 O(k)。

6.9链表方式解决散列冲突有什么优点？

链表方法是在散列冲突以后，将元素作为链表头节点或尾节点插入到散列值对应的散列表位置。

- 优势，内存利用率高，解决冲突的时间更快。
- 缺陷，桶中节点元素内存地址不连续，导致查询性能可能会降低。

总之，基于链表的散列冲突处理方法比较适合存储大对象（此时可忽略指针占用空间）、大数据量的散列表。而且，比起开放寻址法，它更加灵活，支持更多的优化策略，比如用红黑树代替链表。

6.10 Java 中 HashMap 源码分析？

1）初始大小设计

HashMap 默认的初始大小是 16，当然这个默认值是可以设置的，如果事先知道大概的数据量有多大，可以通过修改默认初始大小，减少动态扩容(2 的 n 次方)的次数，这样会大大提高 HashMap 的性能。

2）装载因子和动态扩容设计

最大装载因子默认是 0.75，当 HashMap 中元素个数超过 0.75*capacity（capacity 表示散列表的容 量）的时候，就会启动扩容，每次扩容都会扩容为原来的两倍大小。

3）为什么扩容因子为 0.75?
为什么不是 0.5,也不是 1 呢?是因为这个 0.75 是在时间和空间上取的相对平衡值,假如在 1 的时候扩容,数组中数据越多,产生散列冲突的几率越大,一旦产生散列冲突数据就会转换为链表进行存储,而链表方式会影响查询效率． 假如在 0.5 时进行扩容,但又没有那么多元素要进行存储,可能会产生大量的空间浪费.

4)散列冲突及解决方案设计

HashMap 底层采用链表法来解决冲突。即使负载因子和散列函数设计得再合理，也免不了会出现 链表过长的情况，一旦出现链表过长，则会严重影响 HashMap 的性能。 于是，在 JDK1.8 版本中，为了对 HashMap 做进一步优化，官方引入了红黑树。而当链表长度太 长（默认超过 8）时，链表就转换为红黑树。我们可以利用红黑树快速增删改查的特点，提高 HashMap 的性能。当红黑树结点个数小于或等于 6 的时候，又会将红黑树转化为链表。因为在数据量 较小的情况下，红黑树要维护平衡，比起链表来，性能上的优势并不明显。

5）为什么是链表长度达到 8 的时,进行红黑树转换?

经过大量计算、测试，链表的长度达到 8 的几率已经很小,所以可以直接基于 8 作为链表转红黑树的边界值。
为什么不是大于呢，因为链表长度较长查询效率就会越低。为什么不是 7 呢？链表结点数量比较小时，应用
红黑树还要进行树的平衡设计，需要的成本相对比较高。

6）为什么红黑树节点个数小于 6 的时要转换链表呢？

假如是 7 则数据在链表和红黑树之间来回转换可能会比较频繁，这样就需要更长的时间消耗。

7)线程(thread)安全设计

HashMap 本身并不是线程安全的对象，所以仅可以应用在线程安全的环境。在线程不安全的环境推荐使用 ConcurrentHashMap，此 map 在 JDK8 中采用了 CAS 算法保证对 map 的操作是线程安全的；

6.11 JDK7 和 JDK8 中的 hashmap 有什么不同？

1.7 中采用数组+链表，1.8 采用的是数组+链表/红黑树，即在 1.8 中链表长度超过一定长度后就改成红黑树存储。

1.7 的底层节点为 Entry，1.8 为 node ，但是本质一样，都是 Map.Entry 的实现
1.7 扩容时需要重新计算哈希值和索引位置，1.8 并不重新计算哈希值，巧妙地采用和扩容后容量进行&操作来计算新的索引位置。

1.7 是采用表头插入法插入链表，1.8 采用的是尾部插入法。

在 1.7 中采用表头插入法，在扩容时会改变链表中元素原本的顺序，以至于在并发场景下导致链表成环的问题；在 1.8 中采用尾部插入法，在扩容时会保持链表元素原本的顺序，就不会出现链表成环的问题了。

6.1 Hashmap 中的负载因子为什么是 0.75？

负载因子为 0.75f 是空间与时间的均衡

如果负载因子小，意味着阈值变小。比如容量为 10 的 HashMap，负载因子为 0.5f，那么存储 5 个就会扩容到 20，出现哈希冲突的可能性变小，但是空间利用率不高。适用于有足够内存并要求查询效率的场景。

相反如果阈值为 1 ，那么容量为 10，就必须存储 10 个元素才进行扩容，出现冲突的概率变大，极端情况下可能会从 O(1)退化到 O(n)。适用于内存敏感但不要求要求查询效率的场景

6.2 为何 HashMap 的数组长度一定是 2 的次幂？

数组长度保持 2 的次幂，length-1 的低位都为 1，会使得获得的数组索引 index 更加均匀，减少 hash 冲突。保证得到的新的数组索引和老数组索引一致(大大减少了之前已经散列良好的老数组的数据位置重新调换)。。、、

6.3 说说 ConcurrentHashMap 对象？

ConcurrentHashmap(1.8)这个并发集合是线程安全的 HashMap,在 jdk1.7 中是采用 Segment + HashEntry + ReentrantLock 的方式进行实现的，而 1.8 中放弃了 Segment 臃肿的设计，取而代之的是采用 Node + CAS + Synchronized 来保证并发安全进行实现。

JDK1.8 的实现降低锁的粒度，JDK1.7 版本锁的粒度是基于 Segment 的，包含多个 HashEntry，而 JDK1.8 锁的粒度就是 HashEntry（首节点）
JDK1.8 版本的数据结构变得更加简单，使得操作也更加清晰流畅，因为已经使用 synchronized 来进行同步，所以不需要分段锁的概念，也就不需要 Segment 这种数据结构了，由于粒度的降低，实现的复杂度也增加了
JDK1.8 使用红黑树来优化链表，基于长度很长的链表的遍历是一个很漫长的过程，而红黑树的遍历效率是很快的，代替一定阈值的链表，这样形成一个最佳拍档。

7 树(Tree)结构分析

7.1什么是树？

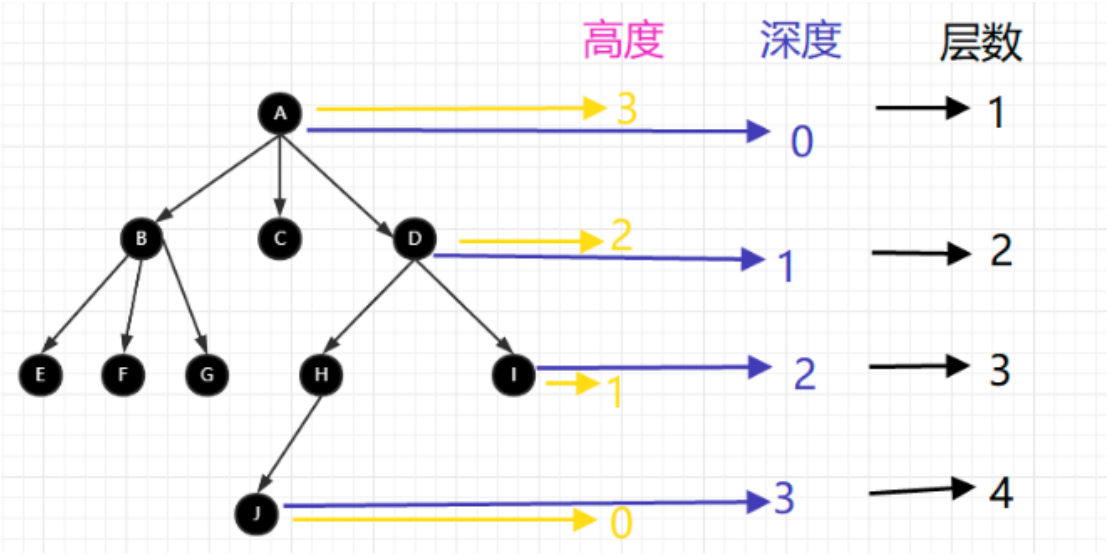
树是一种非线性的数据结构，它是由 n（n>=0）个有限节点组成一个具有层次关系的集合。把它叫做树是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

1.每个节点有零个或多个子节点；
2.没有父节点的节点称为根节点；
3.每一个非根节点有且只有一个父节点；
4.除了根节点外，每个子节点可以分为多个不相交的子树

7.2树中的相关名词如何理解？

1．高度：当前节点到叶子节点的最长路径

2．深度：根节点到当前节点经过的边数
3．层数：节点的深度+1 树的高度：即根节点的高度（就是根节点到叶子节点的最长路径）
4．父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点
5．子节点：一个节点包含的子树节点
6．兄弟节点：具有相同父节点的节点称为兄弟节点
7．叶节点：没有子节点的节点（也叫页子节点） 以上是关于树的一些常见的概念。



7.3什么是二叉树？

二叉树的每个节点最多有两个子节点。

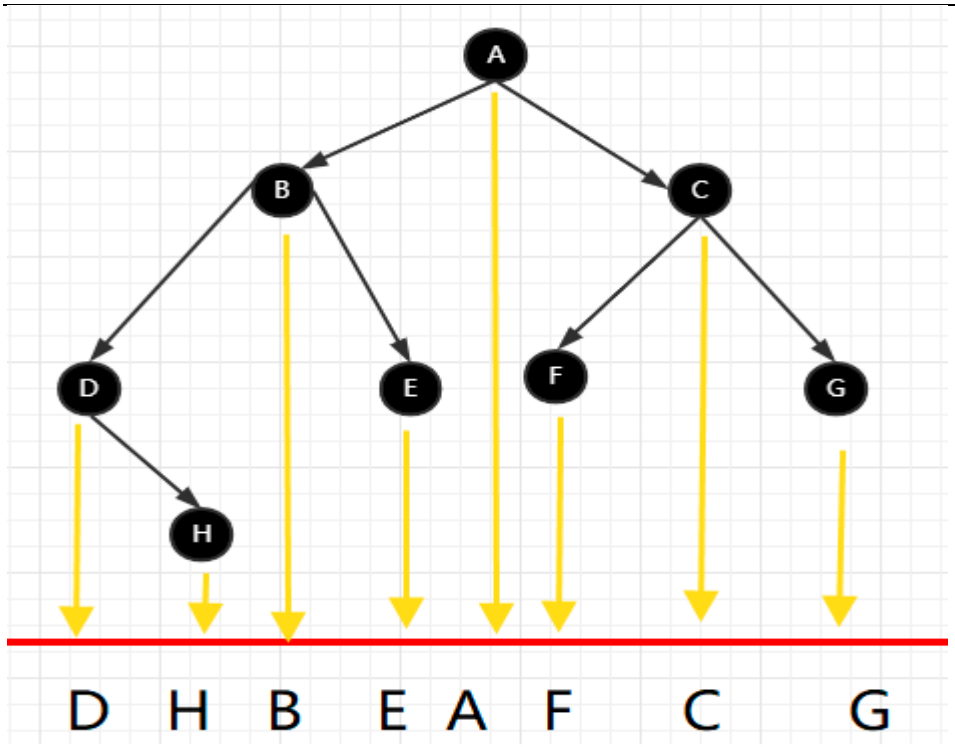
7.4什么是二叉搜索树？

二叉搜索树其实就是二叉树，只不过又有一些额外的条件限制。其额外条件如下：

① 若它的左子树不为空，那么左子树上面的所有节点的值均小于它的根节点的值
② 若它的右子树不为空，那么右子树上面的所有的节点的值均大于它的根节点的值
③ 它的左右的树叶分别为二叉排序树

其中重点强调下二叉搜索树的中序遍历（因为这是最常见的）。中序遍历的规则是：先遍历左子树，再遍历根节点，然后遍历右子树

例如下面这个二叉搜索树的遍历的结果：D-H-B-E-A-F-C-G



二分查找树的最大的缺点是依赖有序数组，而数组的缺点就是不能扩容，还有就是在添加和删除元素的时候需要移动数组，性能不理想。还有就是二叉树的特点就是每个节点的最多只有两个子节点，结合二叉搜索树的特点就是 左子节点 < 根节点 < 右子节点，那么在极端情况下，树可能会变为链表。那时间复杂度就变成了 O(n)。

7.5什么是 AVL 树？

AVL 树也叫平衡二叉树，他的时间复杂度是 O(logn)，AVL 的左右树的高度差也叫平衡因子（平衡因子就是从某个节点开始，他的左右子树的节点数差），即平衡因子不大于 1。

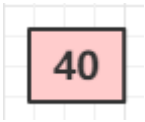
AVL 树在插入数据的时候会不断地调整，因为高度相差不大于 1 真的太严格了。那这样在频繁插入的时候必然需要一直调整树的结构，让其保持平衡。

7.6什么是 2-3 树？

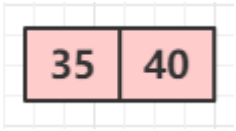
1．2-3 树 是平衡树
2．2 叉节点，有两个分树，节点中有一个元素，左树元素更小，右树元素节点更大
3．3 叉节点，有三个子树，节点中有两个元素，左树元素更小，右树元素更大，中间树介于两个父元素之间？

案例分析：

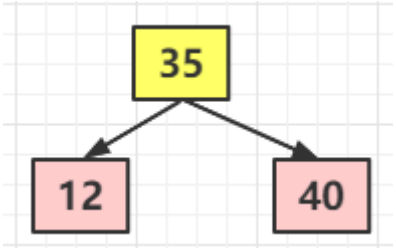
1) 假设现在有一个节点 40，那啥也别说了，第一个节点啥都不做，老实呆着就行；



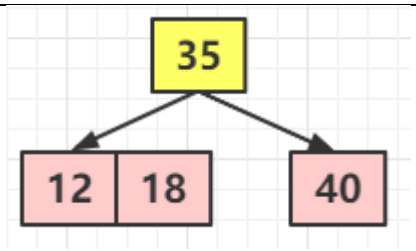
2) 下一个节点 35 ，先从根节点开始，发现 40 > 35 ，此时理论上 35 应该添加到 40 的左子树上，但是对于 2-3 树，并不是你想的那样子，记住核心的一句话对于 2-3 树的添加，永远不会添加到一个空的节点去，只会跟最后找到的叶子节点做融合（不明白也没事，先把这个过程看完），这样变成了一个 3 节点。此时这颗树依旧是平衡的。这个 3 节点的含义是因为接下来的数据可能是小于 35 ，可能是在 35 到 40 之间，也可能是大于 40 的，所以这个节点能放三个节点。



3) 下一个节点是 12 ，按照我们上面解释的 3 节点的含义，12 应该在 3 节点的左侧。那这个时候按照 3 节点的定义，那这个岂不是 4 节点了？其实这个时候答案已经很明显了，就是此时该树会分裂成一个正常的二叉树，也就是这样子的，这棵树依旧是平衡的。

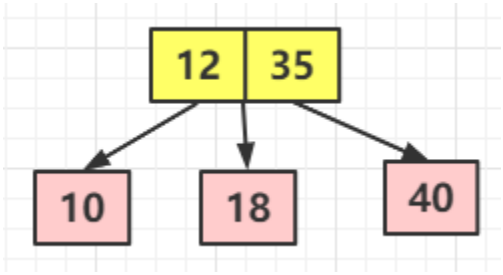


4) 继续添加节点 18 ，自己能脑补下该怎么添加吗？这时候就很简单了，18 < 35 ，就添加到左子节点，此时左子节点不为空，那么就可以继续添加，而 18 > 12，理论上应该是添加到 12 的右子节点，但是由于对于 2-3 树的添加，永远不会添加到一个空的节点去，只会跟最后找到的叶子节点做融合。这个的理论指导，又因为此时 12 是一个 2 节点，所以即可进行融合，将 18 放在 12 的右侧水平对齐。



5) 继续添加 10 ，10 < 35， 到左子树查找，10 < 12 但是 12 的左子树为空，所以 10 先临时和 12 做一个融合，

但是这个时候 12 节点已经变成了 4 节点，所以需要拆解。但是这样的话 2-3 树就不是一颗绝对平衡的树了，显然不能这样拆解，或者是需要做其他操作来保持其绝对平衡。此时我们看上面的图，12 节点实际上是 10 和 18 的根节点了，接着往上查找，12 的父节点是 35 而其是一个 2 节点，所以 12 就顺理成章的和 35 融合起来，也就是下面这样子的。



依次类推可以继续添加，然后融合拆分。

7.7什么是红黑树？

1．根节点是【黑色】
2．每个节点要么是【黑色】要么是【红色】
3．每个【红色】节点的两个子节点一定都是【黑色】
4．每个叶子节点（NIL）都是【黑色】
5． 任意一个节点的路径到叶子节点所包含的【黑色】节点的数量是相同的---这个也称之为【黑色完美平衡】
6．新插入的节点必须是【红色】->为什么？如果新插入的节点是【黑色】，那不管是在插入到那里，一定会破坏黑色完美平衡的，因为任意一个节点的路径到叶子节点的黑色节点的数量肯定不一样了（第 6 点我自己加的，实际特性的定义是前 5 个）

那红黑树在添加和删除节点的时候是靠什么来维持平衡的呢？那就是左旋、右旋加变色，其含义如下：
左旋:以某个节点作为固定支撑点（围绕该节点旋转）,其右子节点变为旋转节点的父节点，右子节点的左子节点变为旋转节点的右子节点，左子节点保持不变

右旋：以某个节点作为固定支撑点(围绕该节点旋转),其左子节点变为旋转节点的父节点，左子节点的右子节点变为旋转节点的左子节点，右子节点保持不变

变色:节点的颜色由红色变成黑色，或者是由黑色变成红色。

8 总结（Summary）

总之，数据结构和算法是一门相对比较难的学科，有些人刚刚接触可能就放弃了，但是数据结构一旦领会，将终身受益。对于其学习过程我们要学会刻意练习，并在整个学习过程中基于角色的不同，迅速调整我们的思维习惯和方式而非仅仅充实一下知识库。