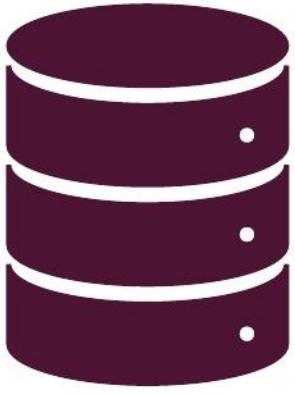


数据结构

WENYE 李
CUHK-SZ

大纲

栈与实现　队列与实现　示例

堆栈

栈作为一种数据结构与栈的内存区域无关！它们是完全不同的东西。

用栈把栈称为数据结构，用栈内存把栈内存区称为栈内存。

要点:

之所以叫栈，是因为它的行为就像现实世界的栈，成堆的书等。

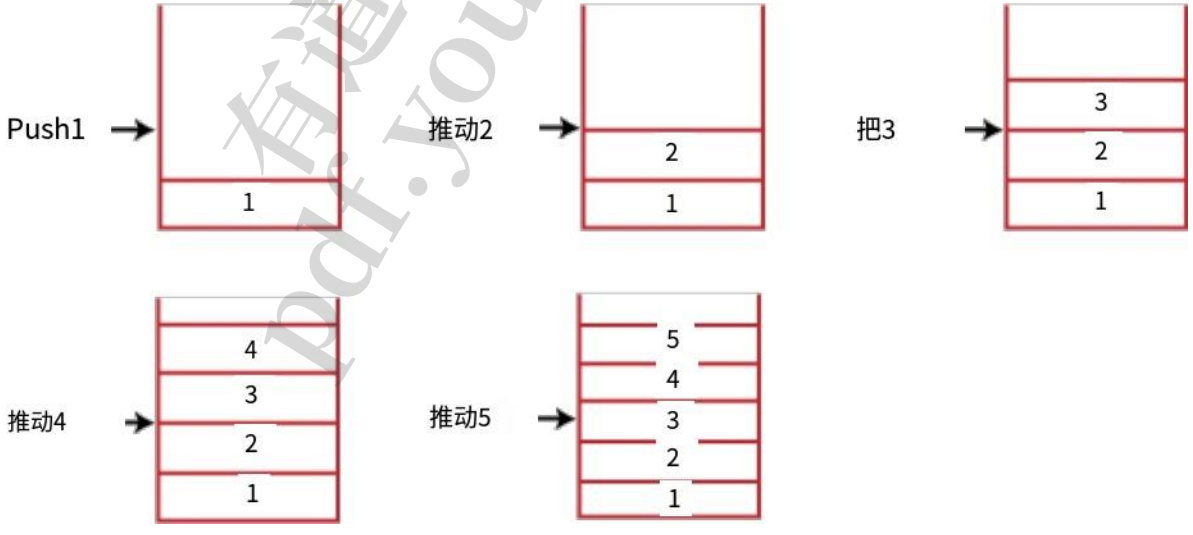
堆栈是一种具有预定义容量的抽象数据类型，这意味着它可以存储有限大小的元素。

它是一种按照一定顺序插入和删除元素的数据结构，这种顺序可以是后进后出(LIFO)或者是前进后出(FILO)。

堆栈

+ Stack 工作在 LIFO 模式上。

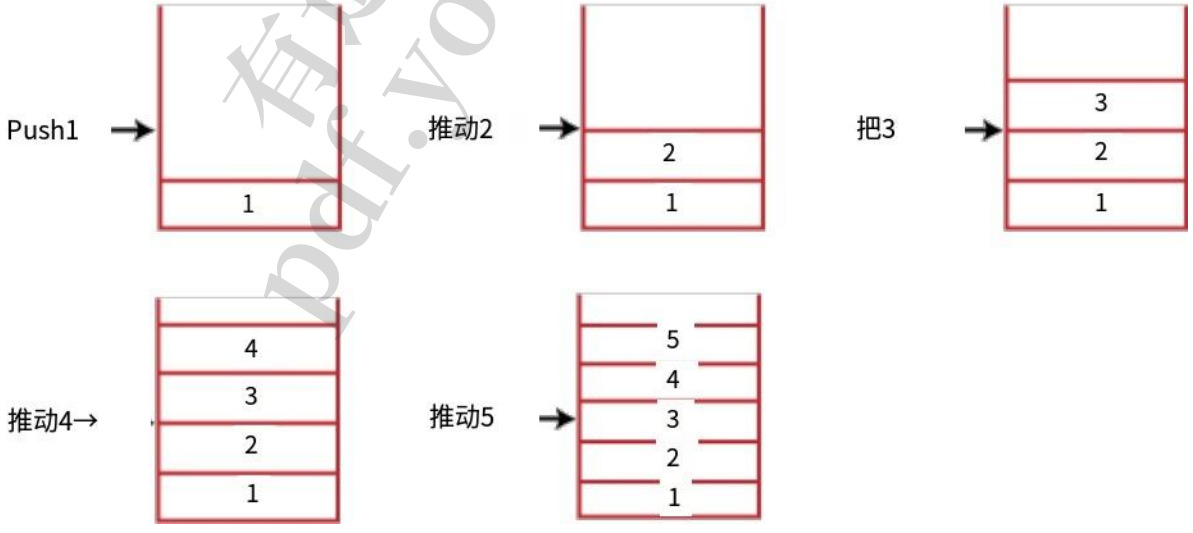
下图中栈中有 5 个内存块;栈的大小为 5。 假设我们想将元素存储在栈中，假设栈为空。我们一个接一个地推入元素，直到栈满。



堆栈

当我们在堆栈中输入新元素时，它从上到下。栈从下到上被填满。

对于 delete 操作，它遵循 LIFO 模式。首先输入值 1，因此只有在删除所有其他元素后才会删除。



堆栈

栈是一种遵循 LIFO 原则的线性数据结构。

Stack 有一端。它只包含一个指针 top 指针指向栈的最顶层元素。

每当一个元素被添加到栈中，它就被添加到栈的顶部，并且该元素只能从栈中删除。

因此，栈可以被定义为一个容器，其中插入和删除可以从称为栈顶的一端进行。

栈上的操作

push():当我们在栈中插入一个元素时，这个操作被称为push。如果栈已满，则发生溢出条件。

pop():当我们从栈中删除一个元素时，这个操作被称为pop。如果栈为空意味着栈中不存在元素，这种状态被称为下溢状态。

isEmpty():确定栈是否为空。

isFull():确定栈是否满。

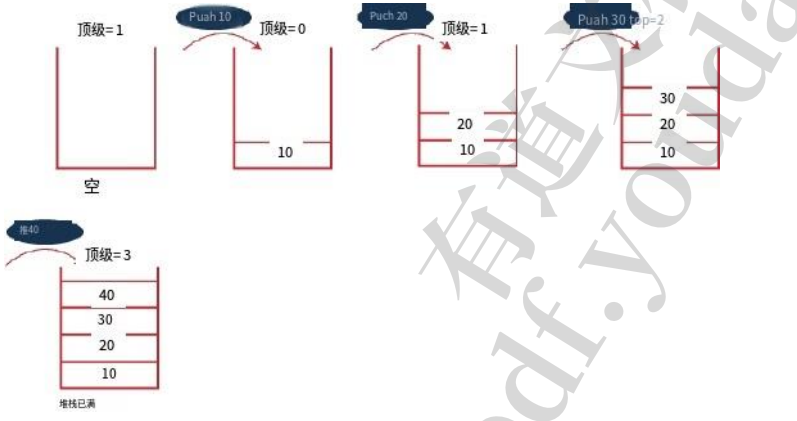
peek():返回给定位置的元素。

count():返回栈中可用元素的总数。

change():改变给定位置的元素。

display():打印栈中所有可用的元素。

推动操作



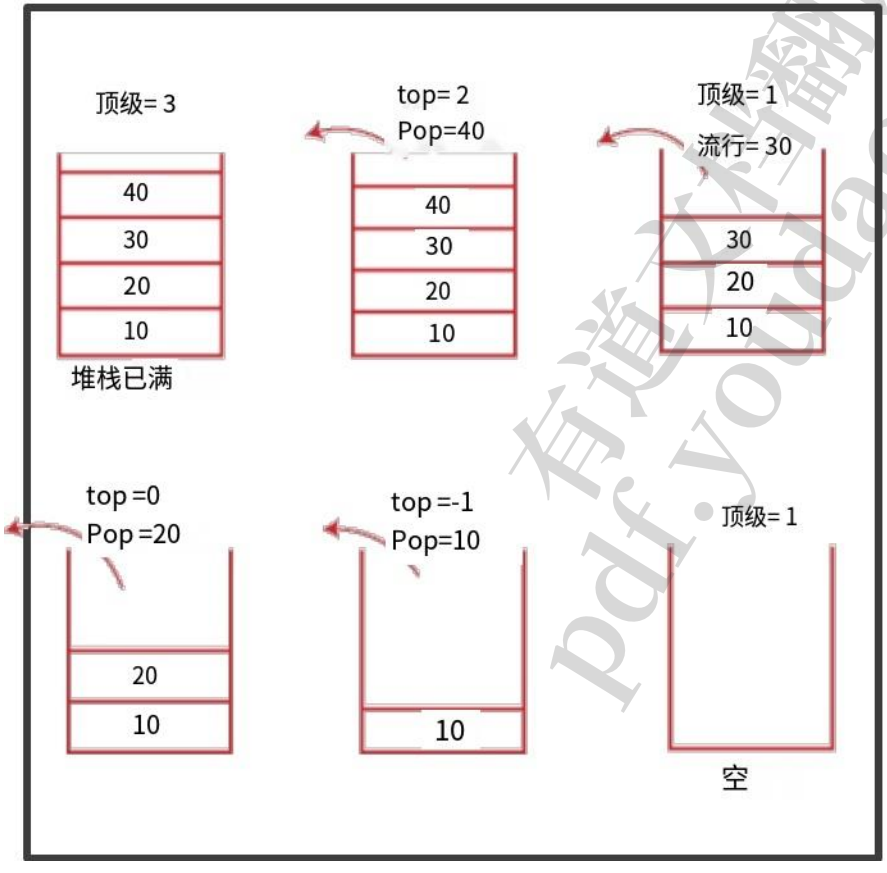
在将元素插入栈之前，我们会检查栈是否满。

如果我们尝试将元素插入栈中，而栈已满，则会发生溢出条件。当我们初始化一个栈时，我们将top的值设置为-1，以检查栈是否为空。

当新元素被压入栈中时，首先，top 的值被递增，即。， top=top+1，元素将被放置在 top 的新位置。

元素将被插入，直到我们达到堆栈的最大大小。

流行的操作



删除元素之前栈，我们检查栈是否为空的。

如果我们尝试从中删除元素空栈，则下溢情况发生。

如果栈不为空，我们首先访问由顶部指向的元素。

一旦执行 pop 操作，则 Top 减 1，即。，=(。

栈上的操作

方法	修饰符和 类型	方法描述
empty0	布尔	该方法检查堆栈是否为空。
push(E item)	E	该方法将一个元素推入(插入)到堆栈顶部。
pop0	E	该方法从堆栈顶部移除一个元素，并返回与的值相同的元素 该函数。
peek ()	相互 作用 E	该方法查看堆栈的顶部元素，但不移除它。
搜索(对象 o)		该方法搜索指定的对象并返回对象的位置。

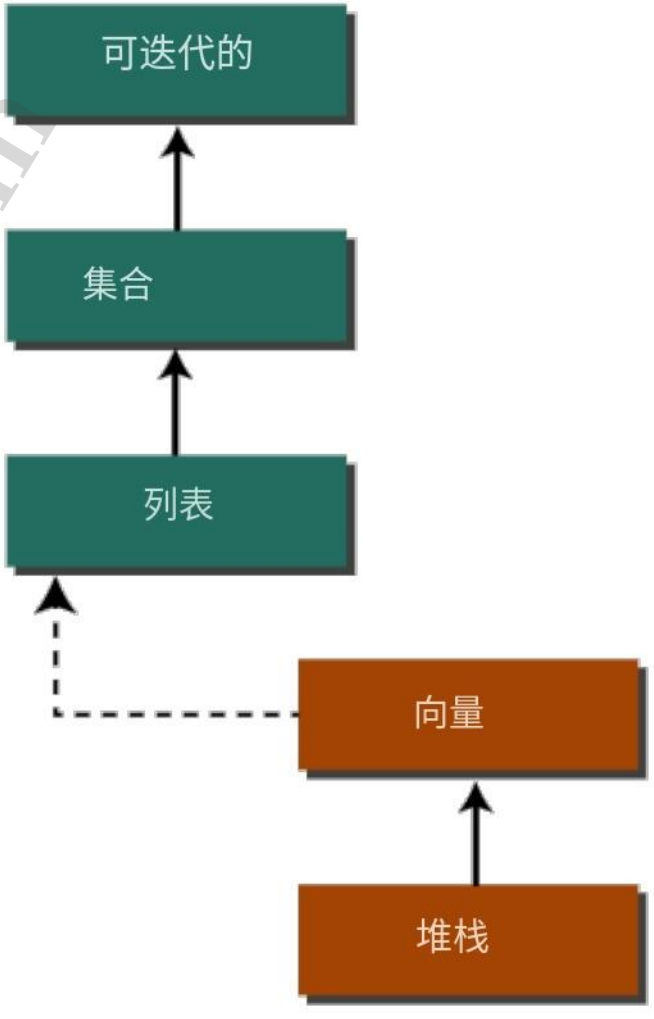
Java 中的栈

在 Java 中，Stack 是一个跌倒的类在集合框架下即继承了 Vector 类。

它还实现了接口列表，Collection, Iterable, Cloneable，可序列化的。

的 LIFO 栈对象。

在使用 Stack 类之前，我们必须导入 java。util 包。



```
进口java.util.Stack;

  公共类StackEmptyMethodExample
。
        public static void main(String[] args)
      。
            //创建一个Stack类的实例
            Stack<Integer> stk = new Stack<>();
            //检查堆栈是否为空
            Boolean result = stk.empty();
            system . out。println(“堆栈是空的吗?+ result);
            //将元素推入堆栈
            stk.push(78);
            stk.push(113);
            stk.push(90);
            stk.push(120);
            //打印堆栈中的元素
            system . out。println(“Elements in Stack:” + stk);
            Result = stk.empty();
            system . out。println(“堆栈是空的吗?+ result);
      ~
}
```

输出:

```
堆栈是空的吗?真正的
堆叠元素:[78,113,90,120]
堆栈是空的吗?假
```

```
进口java.util。*;
公共类StackPushPopExample{
    public static void main(String args[]) {
        //创建Stack类的对象
        Stack< Integer> stk = new Stack<>();
        System.out.println("stack: " + stk);

        pushelmnt(stk, 20);

        pushelmnt(stk, 13);
        pushelmnt(stk, 89);
        pushelmnt(stk, 90);
        pushelmnt(stk, 11);
        pushelmnt (stk、45);
        pushelmnt (stk、18);
        //从堆栈中弹出元素
        popelmnt(stk);
        popelmnt(stk);

        //如果堆栈为空则抛出异常

            popelmnt(stk);
        } catch (EmptyStackException e){系统。println(“空栈
        ”);

    }
    //执行push操作
    static void pushelmnt(Stack stk, int x) {
        //调用push()方法
        stk.push(new Integer(×));
        System.out.println("push -> " + ×);
        //打印修改后的堆栈
        System.out.println("stack: " + stk);

    //执行弹出操作
    静态无效popelmnt(堆栈stk) {
        system . out。Print ("pop -> ");
        //调用pop()方法
        整数x =(整数)stk.pop();
        System.out.println(×);
        //打印修改后的堆栈
        System.out.println("stack: " + stk);
    }
}
```

输出:

```
栈:[]
推-> 20
栈:[20]
Push ->
堆叠:[20,13]
推-> 89
堆叠:[20,13,89]
推-> 90
叠:[20,13,89,90]
Push -> 11 .
Stack: [20, 13, 89, 90, 11]
推-> 45
堆叠:[20,13,89,90,11,45]
Push -> 18 .
Stack: [20, 13, 89, 90, 11, 45, 18]
Pop -> 18
Stack: [20,13,89,90,11,45]
Pop -> 45
Stack: [20,13,89,90,11]
Pop -> 11 .
Stack: [20, 13, 89, 90]
```

```
进口java.util.Stack;

公共类StackPeekMethodExample
{
    public static void main(String[] args)
    {
        Stack<String> stk = new Stack<>();
        //将元素推入Stack
        stk.push(“苹果”);
        stk.push(“葡萄”);
        stk.push(“芒果”);
        stk.push(“橙色”);
        System.out.println("Stack: " + stk);
        //从堆栈顶部访问元素
        String fruits = stk.peek();
        / /打印堆栈
        system . out。println("Element at top: " + fruits);
    ~
}
```

输出:

```
堆叠:[苹果，葡萄，芒果，橙子]
堆栈顶部的元素:橙色
```

```
进口java.util.Stack;
公共类StackSearchMethodExample
{
    public static void main(String[] args)
    {
        Stack<String> stk = new Stack<>();
        //将元素推入Stack
        stk.push(“Mac书”);
        stk.push("HP");
        stk.push(“戴尔”);
        stk.push(华硕);
        System.out.println("Stack: " + stk);
        //搜索一个元素
        int location = stk.search("HP");
        system . out。println(“Dell的位置:”+位置);
    }
}
```

```
进口java.util.Stack;

公共类StackSizeExample
。
        public static void main (String[] args)
      。
            Stack stk = new Stack();
            stk.push(22);
            stk.push(33);
            stk.push(44);
            stk.push(55);
            stk.push(66);
            //检查堆栈是否为空
            boolean rslt = stk.empty();
            system . out。println(“堆栈是否为空?”+ rslt);
            //找到Stack的大小
            Int x = stk.size();
            system . out。println(“堆栈大小是:”+ x);
      }
}
```


输出:

```
堆栈是否为空?false堆栈大小为:5
```

```
进口java.util.Iterator;
进口java.util.Stack;
公共类StackIterationExample1
{
    public static void main (String[] args)
    。
        //创建Stack类的对象
        Stack stk = new Stack();
        //将元素放入堆栈
        stk.push("BMW");
        stk.push(“奥迪”);
        stk.push(“法拉利”);
        stk.push(“布加迪”
        );stk.push(“捷豹”);
        //在堆栈上迭代
        迭代器迭代器= stk.iterator();
        而(iterator.hasNext ())
        ｜
            对象值= iterator.next();
            System.out.println(值);
        }
    ~
}
```

输出:

```
BMW
奥迪
法拉利
布加迪
捷豹
```

```
进口java.util。*;
  公共类StackIterationExample2
{
    public static void main (String[] args)
    {
        //创建一个Stack类的实例
        Stack< Integer> stk = new Stack<>();
        //将元素放入堆栈
        stk.push(119);
        stk.push(203);
        stk.push(988);
        system . out。println("迭代堆栈使用forEach()方法:");
        //调用forEach()方法在堆栈上迭代
        stk.forEach (n - >
        。
            System.out.println (n);
        });
    }
}
```

输出:

```
使用forEach()方法在堆栈上迭代:
119
203
988
```

```
进口java.util.Iterator;
进口java.util.ListIterator;
进口java.util.Stack;

公共类StackIterationExample3

        public static void main (String[] args)
    。
            Stack< Integer> stk = new Stack<>();
            stk.push(119);
            stk.push(203);
            stk.push(988);
            ListIterator<Integer> ListIterator = stk.listIterator(stk.size());
            system . out。println(“从上到下迭代堆栈:”);
            而(ListIterator.hasPrevious ())
            {
                Integer avg = ListIterator.previous();
                System.out.println (avg);
            }
    }
}
```

输出:

```
从上到下迭代堆栈:
988
203
119
```

实现

我们创建一个大小为 100 的静态数组，哪个是堆栈的最大大小可以有。最初，我们的栈是空的。

伪代码用 C-表示风格。

```
struct CharStack1000

    Char buffer[1000];
    Int top = -1;
    //声明堆栈时，top的默认值为-1。
    // -1表示我们的堆栈为空
};

void push(CharStack1000 &stack, char newElement)
{
    + + stack.top;
    堆栈。缓冲(堆栈。top] = newElement;
}

字符前置(CharStack1000&stack)

    返回栈。缓冲(堆栈。高层);
}

无效弹出(CharStack1000&stack)
{
    ——stack.top;
}

int size(CharStack1000 &stack)
{
    返回(堆栈。Top + 1);//简单
}

bool isEmptyStack( CharStack1000 &stack )
{
    返回(stack.)Top == -1);
}
```

```
import static java.Lang.System.exit;//使用链表创
建堆栈
class StackUsingLinkedlist {

    //链表节点私有类节点{int数据;//整数数据节点链接;//引用变
    量节点类型

    }
    //创建全局顶级引用变量global
    节点上;
    / /构造函数
    StackUsingLinkedlist() {
        这一点。Top = null;
    }
    //在堆栈中添加一个元素x的实用函数public void push(int ×) f //在开始插入//创
    建新的节点temp并分配内存//检查堆栈(堆)是否已满。然后插入一个//元素会导
    致堆栈溢出，如果(temp == null) {
        节点temp = new Node();



            System.out.print("\nHeap Overflow");
            返回;
        }
        //将数据初始化为临时数据字段
        Temp.data = x;
        //将top引用放入temp链接
        Temp.link = top;
        //更新顶部引用
        Top = temp;


     //检查堆栈是否为空的实用函数公共布尔isEmpty() {return top == null;

    }
    //返回堆栈顶部元素的实用函数公共int peek() {

        //检查堆栈是否为空
        if (!isEmpty()) {
            返回top.data;
        } else {

            system . out。println(“堆栈为空”);返回1;

        }
    }
    //从堆栈中弹出顶部元素的实用函数public void pop(){//从开始移除

        //检查堆栈溢出if (top == null) {

            System.out.print("\nStack Underflow");
            返回;
        }

        //更新top指针指向下一个节点top = (top).link;

    }
    公共无效显示(){
        //检查堆栈下溢
        If (top == null) {
                system . out。printf (" \ nStack下溢”);退出
                (1);
        其他}{
            节点temp = top;
            While (temp != null) {
                //打印节点数据
                system . out。printf (" % d - >”,temp.data);
                //将temp链接赋值给temp . data
                Temp = Temp .link;
            }
        }
    }
}
公共类GFG {
    public static void main(String[] args){StackUsingLinkedlist obj =新
    StackUsingLinkedlist();

        obj.push(11);obj.push(22);obj.push(44);

        obj.push(33);

        //打印堆栈元素
        obj.display ();
        //打印Stack的Top元素
        System.out.printf("\nTop element is %d\n", obj.peek());
        obj.pop();  //删除栈的顶部元素
                    obj.pop();
        // print堆栈元素
        obj.display ();
        //打印Stack的Top元素
        System.out.printf("\nTop element is %d\n", obj.peek());
    }
}
```

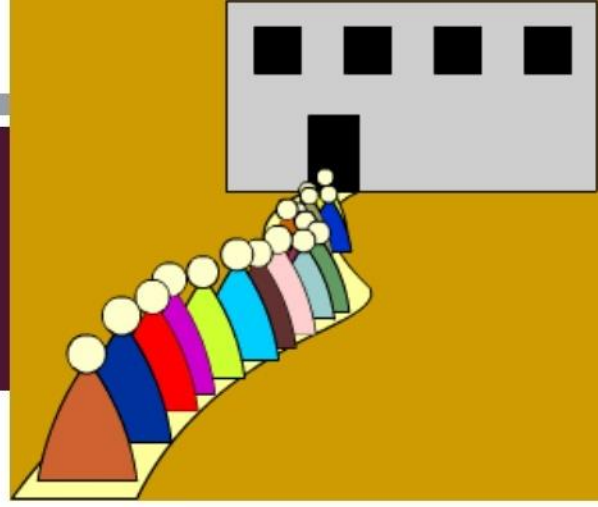
输出:

```
44->33->22->11->
顶部元素是44
22->11->
顶部元素是22
```

大纲

栈与实现　队列与实现　示例

线性队列



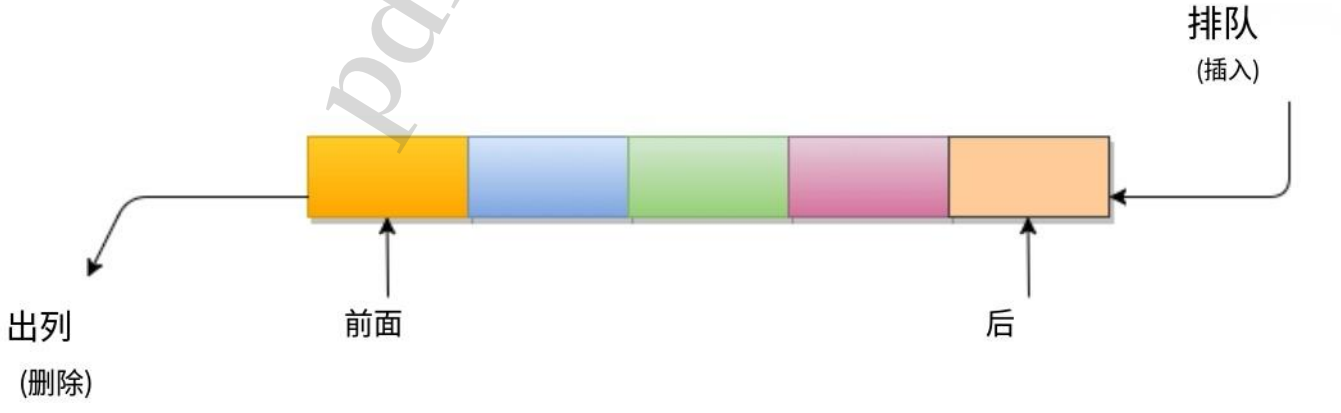
队列:一个有序的列表，使

插入操作要在称为后方的一端执行

删除在另一端执行的操作，称为前端

队列被称为先进先出列表(FIFO)。

例如，排队买火车票的人排成一排。



队列的应用程序

广泛用于单个共享资源的等待列表，如打印机，磁盘，CPU。 用于数据的异步传输(其中数据不是在两个进程之间以相同的速率传输)，例如。管道，文件IO，套接字。

在大多数应用程序中用作缓冲区，如MP3媒体播放器，CD播放器等 用于维护媒体播放器中的播放列表，以便从播放列表中添加和删除歌曲。

在操作系统中用于处理中断。

队列的应用程序

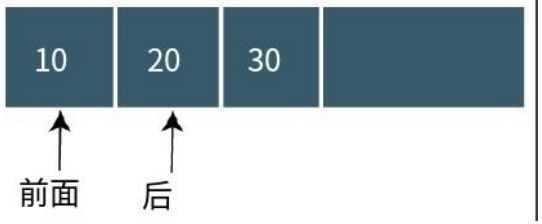
假设我们有一个在网络中不同机器之间共享的打印机。打印机一次可以处理一个请求。

当任何打印请求来自网络时，如果打印机很忙，打印机的程序就会将打印请求放入队列中。

如果队列中有可用的请求，打印机就会从队列前端接收一个请求，并为其提供服务。

计算机中的处理器被用作共享资源。处理器必须执行多个请求，但处理器一次可以执行单个进程。 进程被保存在一个队列中等待执行。

线性队列

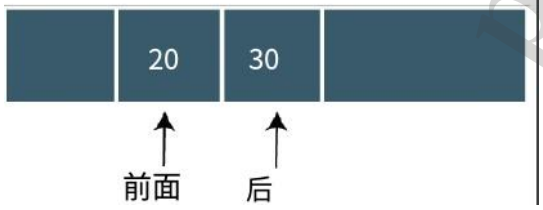


插入发生在后面，删除发生在前面。

元素是从后面插入的。为了在队列中插入更多的元素，后面的值在每次插入时都会增加。

front 指针指向下一个元素，而 front 指针之前指向的元素被删除。

缺点:只能从后端插入。



如果前三个元素从队列中删除，我们就不能插入更多的元素，即使可用的空间在 a 线性队列。线性队列显示溢出条件为尾部指向队列的最后一个元素。

队列的操作

Enqueue:将元素插入队列的尾部。返回 void。

Dequeue:从队列前端删除。它返回已从前端删除的元素。

Peek:返回队列中前端指针指向的元素，但不删除它。

队列溢出(isfull):当队列完全满时，则显示溢出条件。

队列下溢(isempty):当队列为空时，抛出下溢条件。

插入 a 的算法
元素

检查队列是否已被填满
将后排与 Max - 1 进行比较。如果是，那么

如果要将项目作为第一个插入元素，在这种情况下设置的值为 0，然后插入后端的元素。

否则继续增加的值
将每一个元素一一拉后插入以背面为索引的。

- 步骤1:IF REAR = MAX -1
  Write OVERFLOW

  转步骤

  [if结尾]
- 步骤2:IF FRONT = -1 and REAR = -1 SET FRONT = REAR = 0

  其他的

  设置rear = rear + 1

  [if结尾]
- 第三步:设置QUEUE[REAR] = NUM
- 第四步:EXIT

删除元素的算法

如果，front 的值是-1 或 front 的值大于 rear，写一个下溢消息并退出。否则，继续增加 front 的值，每次返回存储在队列前端的物品。

- 步骤1:IF FRONT =-1或FRONT>后方
  编写下溢
  其他的
  设置val = queue [front]
  设置front = front + 1
  [if结尾]
- 第二步:EXIT

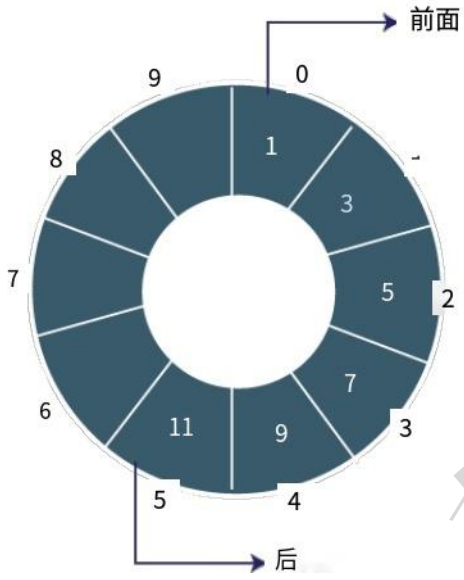
实现

顺序分配:队列中的顺序分配可以使用数组实现。

链表分配:队列中的链表分配可以使用链表实现。

```
//表示队列的类
2类Queue {
    私有int[] arr;              /, 数组，用于存储队列元素
    私有int前端;                // front指向队列51中的front元素
    Private int rear;           // rear指向队列中的最后一个元素
                                //队列的最大容量
    私人int容量;私有int计数;队列的当前大小

    //初始化队列的构造函数
    Queue(int size) {
        Arr = new int[size];
        容量=大小;
        正面= 0;
        Rear = -1;
        计数= 0;
    }

    //将front元素从队列中取出
    Public void dequeue() {
        //检查队列下溢
        如果(isEmpty ())

            system . out。println(“下溢\ nProgram终止”);
            System.exit(1);
        }

        system . out。println(“删除”+      arr[front]);

        Front = (Front + 1) %容量;
        数,
    }

    //向队列公共void中添加一个项的实用函数enqueue(int
    item) {
        //检查队列溢出
        if (isFull()) {
            system . out。println(“溢出\ nProgram终止”);
            System.exit(1);
        }

        system . out。println(“插入”          项);

        Rear = (Rear + 1) %容量;
        Arr[后方]= item;
        数+ +;
    }

    //返回队列前面元素的实用函数
    公共int peek() {
        if (isEmpty()) {
            system . out。println(“下溢\ nProgram终止”);
            System.exit(1);
        }
        返回arr(前);
    }

    //返回队列大小的实用函数
    Public int size() {
        返回计数;
    }

    //检查队列是否为空的实用函数公共布尔值isEmpty()
        Return (size() == 0);
    }

    //检查队列是否满的实用函数公共布尔值isFull()

        返回(size() == capacity);
    }
}

主f
    public static void main (String[] args) {
        //创建一个容量为5的队列
        Queue q = new Queue(5);
        问:排队(1);问:排
        队(2);
        问:排队(3);问:出
        列();
        system . out。println(“The front element is”+ q.peek());system .
        out。println(“The front element is”+ q.peek());system . out。
        println(“队列大小是”+ q.size());

        q.dequeue ();
        q.dequeue ();
        if (q.isEmpty()) {
            system . out。println(“队列为空”);
        } else {
            system . out。println(“队列不是空的”);
        }
    }
}
```

循环队列



所有的节点都表示为圆形。

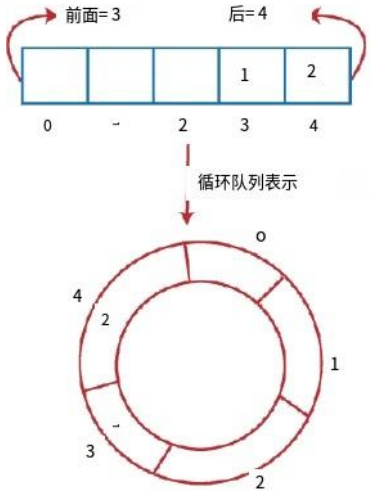
类似于线性队列，只是的最后一个元素队列连接到第一个元素。

也称为环形缓冲区，因为所有的端点都连接到另一个结束。

克服了线性队列中出现的缺点。

如果在循环队列中有空闲空间，则可以通过简单地增加 rear 的值在空闲空间中添加新元素。

循环队列



在上面的数组中，只有两个元素，其他三个位置都是空的。后方是队列的最后一个位置。如果我们尝试插入元素，那么它将显示队列中没有空格。

解决方案:后面是队伍的最后一个位置，前面是指向某处而不是 0th 的位置。

循环队列的应用

•内存管理:循环队列通过将元素放置在未使用的位置来更有效地管理内存(比线性队列)。

•CPU 调度:操作系统也使用循环队列插入进程，然后执行它们。

交通系统:在计算机控制的交通系统中，红绿灯是循环队列的最好例子之一。每隔一段时间，交通灯就会一个接一个亮起来。比如红灯亮一分钟，黄灯亮一分钟，然后绿灯。绿灯亮后，红灯亮。

排队算法

步骤1:IF (REAR+1)%MAX = FRONT
写“OVERFLOW”
转到第4步
[End OF IF]

SET FRONT = REAR = 0步骤2:IF FRONT = -1 and REAR = -1

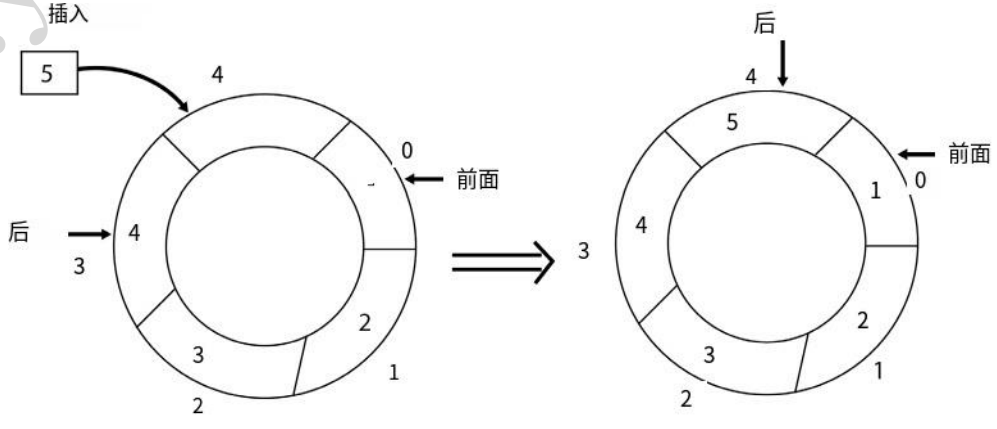
ELSE IF REAR = MAX - 1 and FRONT != 0
集合后方= 0
其他的
Set rear = (rear + 1) % Max
[if结尾]

第三步:SET QUEUE[REAR] = VAL

第四步:EXIT

- 首先，检查队列是否已满。
- 首先将 front 和 rear 都设置为-1。要插入第一个元素，前面和后面都设置为 0。

要插入一个新元素，尾部将递增。



出列算法

步骤1:IF FRONT =-1
写上“UNDERFLOW”
Goto第四步
[END of IF]

第二步:SET VAL = QUEUE[FRONT]

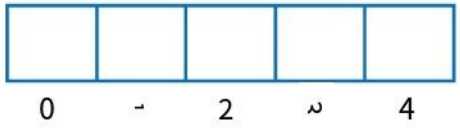
第三步:IF FRONT = REAR
设置front = rear = -1
其他的
If front = Max -1
设置front =0
其他的
设置front = front + 1
[IF结尾]
[if结尾]

第四步:EXIT

首先，检查队列是否为空。如果队列为空，我们就不能执行出队操作。

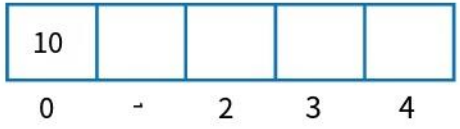
当元素被删除时，front 的值减少 1。

如果只剩下一个要删除的元素，则 front 和 rear 重置为-1。



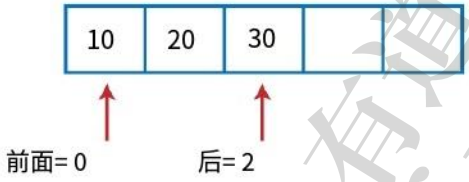
前面= 1

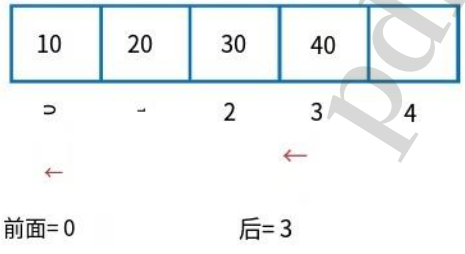
后= 1

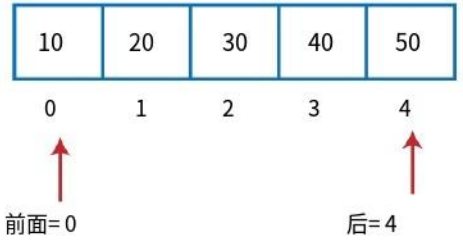


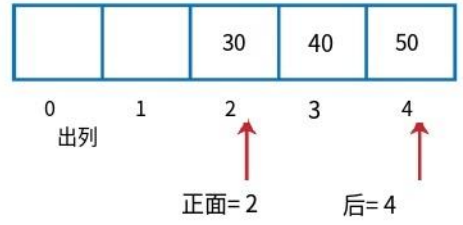
前面= 0

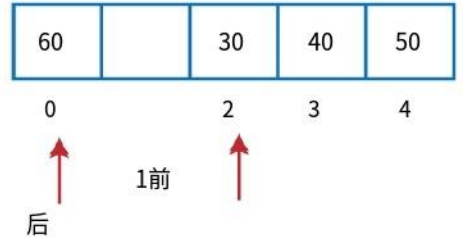
后= 0

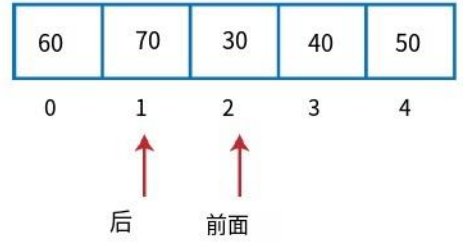












使用数组实现循环队列

```
进口. io . *;
class CircularQ {
    int Q[] = new int[100];
    Int n，前，后;
    静态BufferedReader br = new BufferedReader
            InputStreamReader(系统));
    public CircularQ(int nn) {

        n =神经网络;前=后= 0;

    }
    Public void enqueue(int v)
    {if((rear+1) % n != front) {
            rear = (rear+1)%n;
            Q[rear] = v;

        其他的
            system . out。println(“队列已满!”);
    }
    公共int dequeue() {
        int v;
        if(front!=rear) {
            Front = (Front +1)%n;
            v = Q[正面];
            返回v;

        其他的
            返回-9999;
    }
    公共虚空disp() {
        int我;
        if(front != rear) {
            I = (front +1) %n;
            while(i!=rear) {
                System.out.println(Q[i]);
                I = (I +1) % n;
            }
        }
        其他的
            system . out。println(“队列为空!”);
    }
```

```
public static void main(String args[])抛出IOException f
System.outprint("输入队列的大小:");int size =
Integer.parseInt(br.readLine());循环q调用=新的循环q(大小);
    int选择;
    Boolean exit = false;
    而退出(!){
        System.out.print("\n1 : Add\n2 : Delete\n" +
            "3: Display\n4: Exit\n\n your Choice: ");
        choice = Integer.parseInt(br.readLine());
        switch(choice) {
            案例1    :
                system . out。print("\nEnter要添加的数字:");
                int num = Integer.parseInt(br.readLine());
                call.enqueue(num);
                打破;
            案例二:
                Int pop = call.dequeue();If (pop
                != -9999)
                其他的system . out。println("\nDeleted: " +弹出);

                打破;system . out。println(“\nQueue是空的!”);

            案例3:
                call.disp();
                打破;
            案例4:
                Exit = true;
                打破;
            默认值:
                system . out。println(“\nWrong Choice !”);
                打破;
        }
    }
}
}
```

```
输入队列的大小                    : 5

1:添加
2:删除
3:显示
4:退出

Your Choice: 1

输入要添加的数字:31

1:添加
2:删除
3:显示
4:退出

你的选择:1

输入要添加的数字:28

1:添加
2:删除
3:显示
4:退出

你的选择:1

输入要添加的数字:9

1:添加
2:删除
3:显示
4:退出

你的选择:1

输入要添加的数字:56

1:添加
2:删除
3:显示
4:退出

你的选择:3
31
28
9

1:添加
2:删除
3:显示
4 :退出
```

优先队列

的行为类似于普通队列，除了每个元素都有一些优先级　具有最高优先级的元素将在优先级队列中排在第一位。

优先级队列中元素的优先级将决定元素从优先级队列中删除的顺序。

仅支持可比元素

元素要么按升序排列，要么按降序排列。

例如，假设我们在一个优先级队列中插入值 1、3、4、8、14、22，这些值的顺序是从最小到最大的。因此，1 的优先级最高，而 22 的优先级最低。

优先级队列的特征

优先队列中的每个元素都有相应的优先级。

具有较高优先级的元素将在删除较低优先级的元素之前被删除。

•如果优先级队列中的两个元素具有相同的优先级，它们将使用 FIFO 原则进行安排。

优先级队列的特点

考虑一个具有以下值的优先队列:1、3、4、8、14、22

所有值按升序排列。现在，我们将观察执行以下操作后优先级队列的样子:

- poll():从优先队列中删除最高优先级的元素。`1`元素具有最高优先级，因此它将从优先队列中删除。
- add(2):在优先队列中插入`2`元素。因为 2 是所有数字中最小的元素，所以它将获得最高的优先级。
- poll():将'2'元素从优先队列中删除，因为它具有最高优先级队列。
- add(5):在 4 之后插入 5 个元素，因为 5 大于 4 且小于 8，所以它将在优先队列中获得第三高的优先级。

实现

优先级队列可以通过四种方式实现：数组

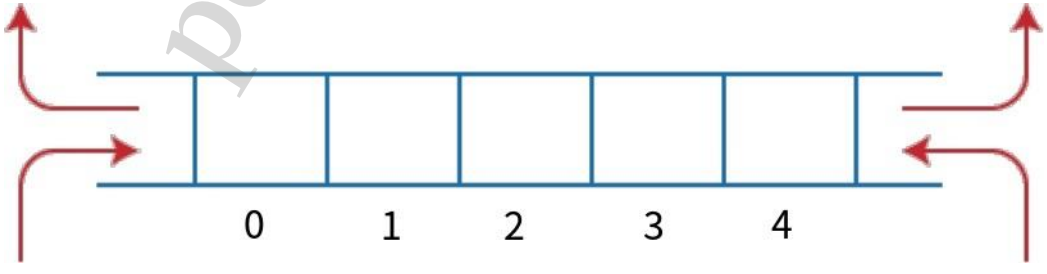
- 链表
- 堆数据结构
- 二叉查找树

堆数据结构是最高效的方式(稍后会回来)。

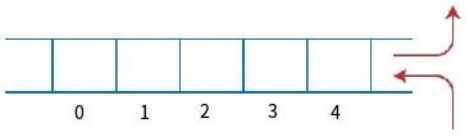
双端队列

双端队列是一种线性的数据结构，一种广义的队列。不遵循 FIFO 原则。

插入和删除可以从两端发生。

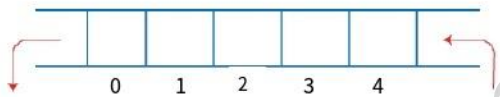


双端队列的例子



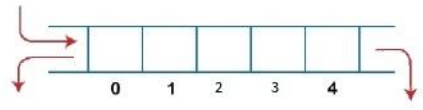
既可以用作栈，也可以用作队列。

插入和删除操作可以从一边。堆栈遵循 LIFO 规则，其中两个插入和删除只能从一端执行。

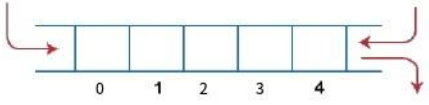


插入可以在一端进行，而删除可以在另一端完成。队列跟随在 FIFO 规则中，元素插入在一端和从另一端删除。

双端队列的例子



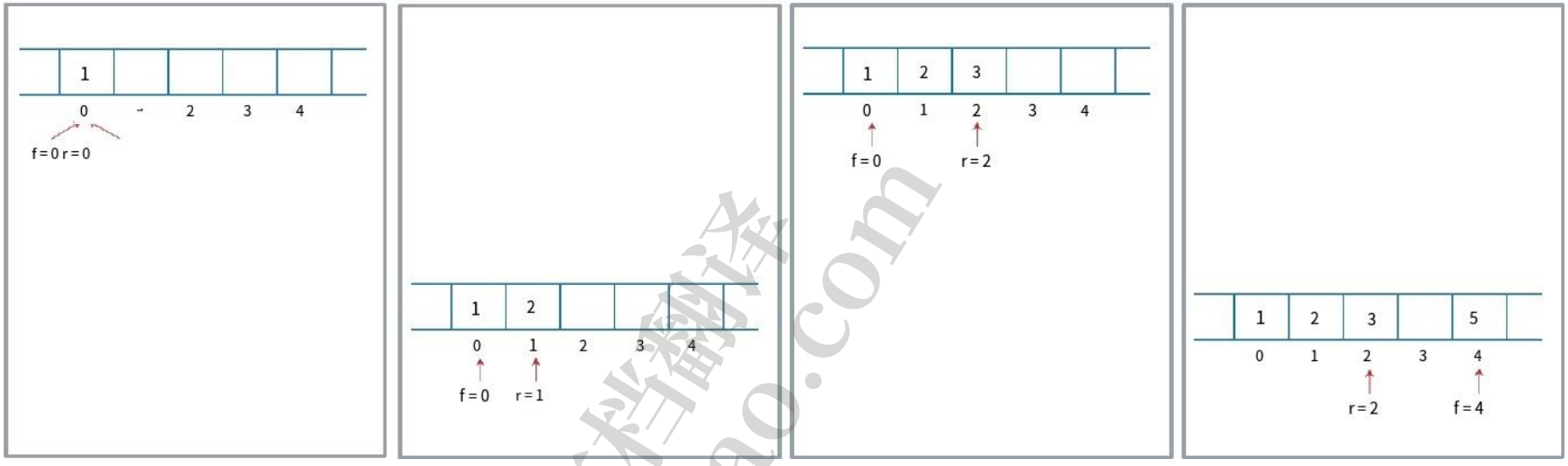
输入受限队列：应用了一些限制插入。插入时应用于一端删除从两端执行。



输出受限队列：对其施加了一些限制删除操作。只能应用删除操作从一端开始，而从两端都可以插入结束。

双端队列的操作

- enqueue_front():用于从前端插入元素。
- enqueue_rear():用于从后端插入元素。
- dequeue_front():用于从前端删除元素。
- dequeue_rear():用于从后端删除元素。
- getfront():用于返回 deque 的前端元素。
- getrear():用于返回 deque 的后元素。



排队

最初，deque 是空的。front 和 rear 都是-1，即。， f = -1, r = -1。

由于 deque 是空的，所以无论从前端还是后端插入元素都是一样的。假设我们插入了元素 1，那么 front 等于 0,rear 也等于 0。要从后端插入元素，需要增加后端，即。后=后+ 1。现在，后方指向第二个元素，前方指向第一个元素。

再次从后方插入元素，先增加后方，现在后方指向第三个元素。

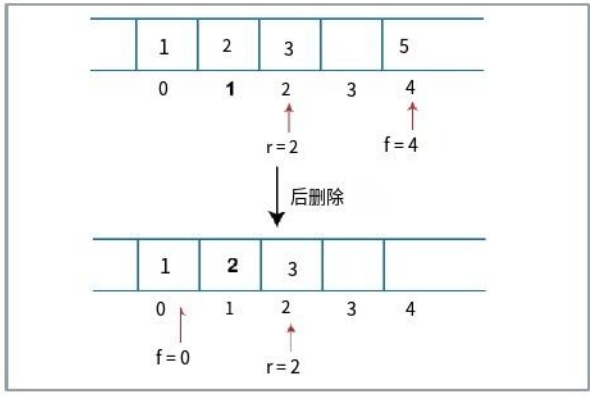
要从前端插入元素，从前端插入元素，则将 front 的值减 1。然后 front 指向-1 位置，这不是数组中的任何有效位置。所以，将 front 设为(n-1)，当 n 为 5 时，就等于 4。

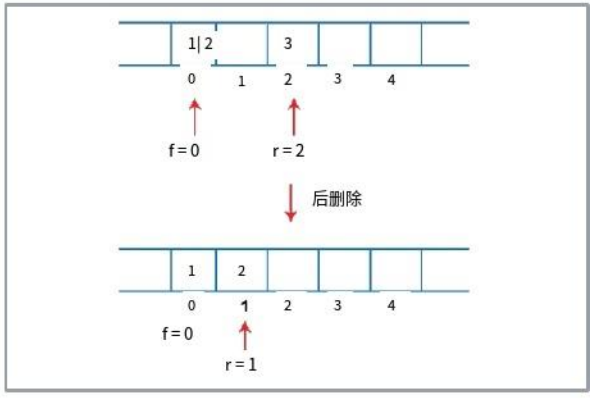
出列

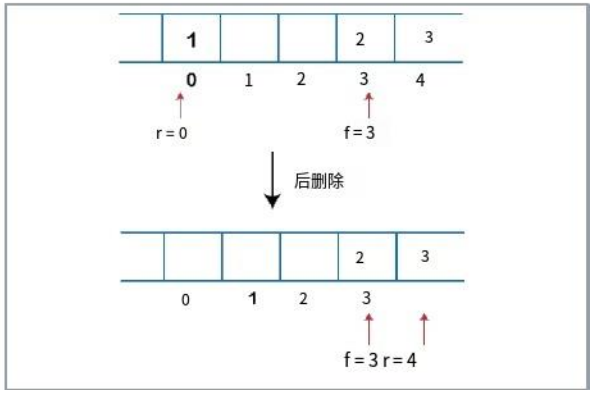
假设 front 指向最后一个元素。删除一个元素，设置 front=front+1。目前，正面是 4，变成了 5，无效。因此，如果 Front 指向最后一个元素，则 Front 被设置为 0 删除操作。

从后端删除元素，然后递减后端值减 1，即。,后面= rear-1。

假设 rear 指向第一个元素。删除从后端开始的元素，设置 rear=n-1，其中 n 是数组的大小。







大纲

栈与实现　队列与实现　示例

```
//A链表节点类
    int数据;                    //整数数据
    节点下;
    /指向下一个节点的指针public node (int data) {
        I/在分配的节点中设置数据并返回
        这一点。数据=数据;
        这一点。Next = null;
    }
}
cLass队列{
    private static节点rear = null, front = null;//前端元素出队列的实用函数
    Public static int dequeue(){//开始删除
        If (front == null) {
            System.out.print("\nQueue Underflow");
            System.exit(1);
        }
        Node temp = front;
        System.out.printf("Removing %d\n", temp.data);
        //提前到下一个节点
        Front = Front .next;
        //如果列表变为空
        If (front == null) {
            Rear = null;
        }

        //释放被移除节点的内存，//可选地返回被移除的项目int item = temp.data;
        返回项目;

    }
    //向队列公共静态void enqueue(int item)中添加一个项目的
    实用函数{                                                    I /末尾插入
        //在堆中分配一个新节点node node = new node (item);system .
        out。printf(“插入%d\n”， item);If (front == null)f .
        //特殊情况:队列为空

            Front = node;       //初始化前后都是rear =
                                node;
        }其他{
            // update rear

            后方。下=节点;后=节点;

        }
    }
```

```
    //返回队列顶部元素的实用函数

        //检查是否有空队列
        If (front != null) {
            返回front.data;
        {} else {
            System.exit(1);
        }

        返回1;
    }
    //检查队列是否为空的实用函数公共静态布尔isEmpty() {
        返回rear == null && front == null;
    }
}

cLass Main {
    public static void main(String[] args) {Queue q = new
    Queue();
        q.enqueue(1);
        q.enqueue (2);
        q.enqueue(3);
        q.enqueue(4);
        system . out。printf("The front element is %d\n"， q.peek());
        q.dequeue ();
        q.dequeue ();
        q.dequeue ();
        q.dequeue ();
        if (q.isEmpty()) {
            system . out。print(“队列为空”);
        {} else {
            system . out。print(“队列不是空的”);
        }
    }
}
```

使用链表实现队列

```
进口java.util.ArrayDeque;
进口java.util.Queue;
//使用两个队列实现堆栈
class Stack<T> {
    队列<T> q1, q2;
    / /构造函数
    public Stack() {
        q1 = new ArrayDeque<>();
        q2 = new ArrayDeque<>();
    }
    //在堆栈中插入一个数据add(T data) {

        I/将所有元素从第一个队列移到第二个队列，同时(!q1.isEmpty()) {
            q2.add (q1.peek
            ());q1.poll ();
        }
        //将给定的项推入第一个队列
        q1.add(数据);

        //将所有元素从第二个队列移回第一个队列while (!q2.isEmpty()){
            q1.add (q2.peek
            ());q2.pol1 ();
        }
    }
    //从堆栈中移除最上面的项
    public T poll() {

        //如果第一个队列是空的if (q1.isEmpty()) {
            System.out.println(“下溢!!”);
            System.exit(0);
        }

        T front = a1.peek();q1.poll ();

        返回前线;
    }
}
class Main {

    public static void main(String[] args) {int[] keys = 1,2,3,4,5};//将上述键插入堆
    栈中

        Stack<Integer> s = new stack<Integer>();

        For (int key: keys) {s.add(key);
        }

        For (int I = 0;I <= keys.length;i++) {System.out.println(s.poll());

        }
    }
}
```

实现栈使用队列

```
进口java.util.Stack;//使用两个堆栈实现一个队列

class Queue<T> {
        private Stack<T> s1, s2;
        / /构造函数
        队列(){
                s1 = new Stack<>();
                s2 = new Stack<>();
         }

        公共无效队列(T数据){
                                                    //向队列中添加一项I/将第一个堆栈中的所
                                                    有元素移到第二个堆栈while
                                                    (!s1.isEmpty()) {
                        s2.push (s1.pop ());
                 }

                //将项目推入第一个stacks1.push(data);


                //将所有元素从第二个堆栈移回第一个堆栈while (!s2.isEmpty()) {
                        s1.push (s2.pop ());
                 }
         }
        //从队列公共T dequeue()中移除一个项目{
                //如果第一个堆栈为空
                if (s1.isEmpty()) System.out.println("底
                {
                流!!”);
                        System.exit(0);
                 }

                //返回第一个堆栈的顶部项

         }
}
class Main {
        public static void main(String[] args) {
                Int [] key = {1,2,3,4,5};
                Queue<Integer> q = new Queue<Integer>();
                  //插入以上键
                 for (int kev: kevs)f
                        q.enqueue(关键);
                 }
                System.out.println (q.dequeue                    //打印1
                ());System.out.println (q.dequeue ());            //打印2
         }
}
```

实现一个
队列使用
两个堆栈

例子

约瑟夫问题

有 n 个人站成一圈等着被处死。第一个人被处决后，有 k- 1 个人被跳过，第 k 个人被处决。然后，又有 k-1 个人被跳过，第 k 个人被处决。消灭过程围绕着圆圈进行(随着被处决的人被移走，圆圈变得越来越小)，直到只剩下最后一个人，他被赋予了自由。

任务是在给定 n 和 k 的情况下，在初始的圆圈中选择让你生存的位置。

谢谢