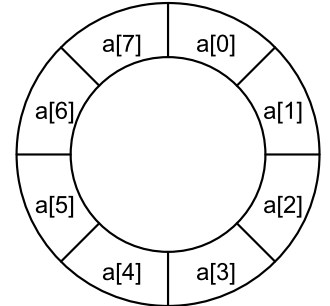


1. キュー（待ち行列）とは、先入れ先出し方式（FIFO, 先着順処理）のデータ構造である。これを配列を使って素朴な方法で実現すると、先頭のデータを取り出すアルゴリズムは以下ようになる。この方法の計算量を  $O$  記法で表せ。取り出し処理を高速にするためにはどうすればいいか考えてみよ。

```
data = a[0]; n--;
for (i = 0; i < n; i++)
    a[i] = a[i + 1];
```

2. リングバッファは右図のように配列の末尾と先頭がつながっているものとして扱うデータ構造である。下記はリングバッファによってキューを実現したプログラムである。キューの末尾にデータを追加する操作を**エンキュー**、キューの先頭からデータを取り出す操作を**デキュー**という。適切に空欄を埋めてプログラムを完成させ、動作を確認せよ。また、計算量の改善を考察せよ。



```
/* Queue.java */
public class Queue {

    private int[] array; // リングバッファ
    private int head; // キューの先頭 (添字)
    private int num; // 格納データの個数

    public Queue(int size) {
        array = new int[size];
        head = 0;
        num = 0;
    }

    public void enqueue(int data)
        throws Exception {
        // 空きがない場合をチェック
        if (num
            ) {

            throw new Exception("full!");
        }
        // キューの末尾の位置 (添字) を求める
        int tail = (head + num) %

        array[tail] = data;
    }

    public int dequeue() throws Exception {
        // データがない場合をチェック
        if (num
            ) {

            throw new Exception("empty!");
        }
        int data = array[head];

        // 先頭位置を1つ進める
        head = (head + 1) %

        return data;
    }

    public void printAll() {
        System.out.print("[ ");
        for (int i = 0; i < num; i++) {
            int k = (head + i) % array.length;
            System.out.print(array[k] + " ");
        }
        System.out.println("]");
    }
}

/* Program.java */
import java.util.Scanner;

public class Program {

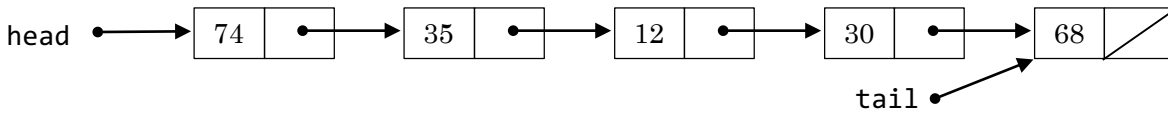
    public static void main(String[] args)
        throws Exception {

        // 容量8個のキューの新規生成
        Queue queue = new Queue(8);
        Scanner sc = new Scanner(System.in);

        for (int i = 0; i < 10; i++) {
            System.out.print("number? ");
            int data = sc.nextInt();

            if (data > 0) {
                // 入力が正数ならその数を格納
                queue.enqueue(data);
            } else {
                // 入力が負 or 0なら取り出し
                data = queue.dequeue();
                System.out.println(data);
            }
        }
        queue.printAll();
    }
}
}
```

3. 下記のプログラムは、2.と同様のキューを単方向リストに末尾を指すフィールド `tail` を加えて実現したものである。適切に空欄を埋めてプログラムを完成させ、動作を確認せよ。



```

/* Queue.java */
public class Queue {

    // 連結リストの先頭と末尾のノードを指す
    private Node head, tail;

    public Queue() {
        head = tail = null;
    }

    public void printAll() {
        // 第8回の3.と全く同じ
    }

    // リストの末尾にノードを加える
    public void enqueue(int data) {
        Node n = new Node(data);

        // もしリストが空だった場合
        if (head == null) {
            head = tail =
        }
        else { // そうでない場合
            = n;

            tail =
        }
    }
}

// リストの先頭からノードを取り出す
public int dequeue() throws Exception {
    // データがない場合をチェック
    if (head == ) {

        throw new Exception("empty!");
    }
    // data を取り出し、新しい head を設定
    int data = head.data;
    head =

    // リストが空になった場合
    if (head == null)
        tail =

    return data;
}

}

/* Node.java */
// 第8回の3.と全く同じ

/* Program.java */
/* メインプログラムは今回の2.のものを
   そのまま利用する。ただし new Queue(8)
   は new Queue() に書き換える。
*/

```

4. 下記は、任意のクラス `E` のデータを要素とする連結リストのプログラムの一部である。引数で指定されたノード `p` の次のノードを削除するメソッド `removeNext` と、同様にノード `p` の次に新しいノードを挿入するメソッド `insertNext` とを完成させ、他のメソッドを適切に補ってプログラムを動作させよ。

```

public class Node<E> {
    public E data;
    Node next;
    public Node(E d) { data = d; }
}

public class List<E> {
    private Node<E> head = null;

    // ノード p の“次”のノードを削除
    public void removeNext(Node<E> p) {
        if (p == null) {
            head = head.next;
        } else if (p.next != null) {
            p.next =
        }
    }
}

// ノード p の“次”に新ノード n を挿入
public void insertNext(
    Node<E> p, Node<E> n) {
    if (p == null) {
        n.next = head;
        head = n;
    } else {
        n.next =

        p.next =
    }
}

// 全要素を表示するメソッド printAll() や
// データの値からノードを検索するメソッド
// Node<E> search(E data) 等を補うとよい
}

```