



Universidad
Tecnológica
de Pereira

CAPÍTULO: LISTAS SIMPLEMENTE ENLAZADAS, DOBLEMENTE ENLAZADAS, CIRCULARES. PILAS Y COLAS.

CURSO DE ESTRUCTURA DE DATOS - Código IS304

Programa de Ingeniería de Sistemas y Computación

Profesor Hugo Humberto Morales Peña

Semestre Agosto/Diciembre de 2021

Función malloc()

La función `malloc()` se utiliza para asignar a un apuntador una localización de memoria. La asignación de memoria es dinámica.

Ejemplo 1:

```
char *p;  
int n = 10;  
p = (char *) malloc(n);
```

En la variable `p` está almacenada una dirección que apunta a un bloque de 10 bytes.

El programa completo en Lenguaje C/C++ sería:

```
#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
#include <math.h>  
  
int main()  
{  
    char *p;  
    int n = 10;  
    p = (char *) malloc(n);  
    strcpy(p, "Sencillo");  
    printf("%s\n", p);  
    return 0;  
}
```

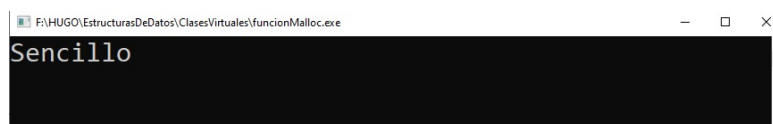


Figura 1: Salida del Ejemplo 1.

Ejemplo 2:

Almacenar el número entero 821 en una localización de memoria asignada por la función `malloc()`.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
    int *p;
    p = (int *) malloc(sizeof(int));
    printf("Direccion de memoria apuntada por el puntero p: %d\n", p);
    *p = 821;
    printf("Direccion de memoria apuntada por el puntero p: %d\n", p);
    printf("Contenido del bloque apuntado por p: %d\n", *p);
    return 0;
}
```

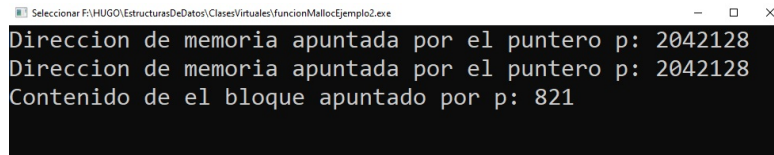


Figura 2: Salida del Ejemplo 2.

Ejemplo 3:

En el siguiente código se trabaja con una estructura propia.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

typedef struct R
{
    int a;
    float b;
    double c;
    char *p;
} estructura;

int main()
{
    int n = 10;
    estructura *q;
    q = (estructura *) malloc(sizeof(estructura));
    q->a = 8;
    q->b = 10.31;
    q->c = 10.32;
    q->p = (char *) malloc(n);
    strcpy(q->p, "Cadena");
    printf("%d %.2f %.21f %s\n", q->a, q->b, q->c, q->p);
    return 0;
}
```



Figura 3: Salida del Ejemplo 3.

Función free()

La función `free()` permite liberar bloques de memoria dinámica que han sido asignados por la función `malloc()`.

Ejemplo 4:

¿Se da un buen manejo de la memoria en el siguiente programa de Lenguaje C/C++?

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
    char *p, *q;
    int n = 12;
    p = (char *) malloc(n);
    strcpy(p, "Pascal");
    q = (char *) malloc(n);
    strcpy(q, "Lenguaje C");
    printf("%s, %s\n", p, q);
    q = p;
    printf("%s, %s\n", p, q);

    return 0;
}
```

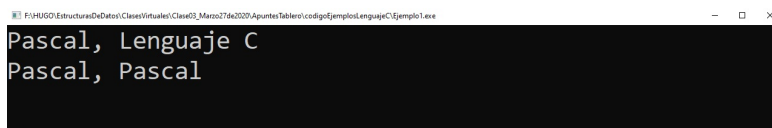


Figura 4: Salida del Ejemplo 4.

No se hace un buen uso de la memoria ... el bloque de memoria que originalmente estaba siendo apuntado por *q* ahora queda como un bloque de memoria “zombie”, el cual no se puede volver a utilizar hasta que se termine la ejecución del programa. Lo que se debe hacer es liberar previamente el bloque de memoria apuntada por *q* con la instrucción `free()`, así como se hace en el siguiente programa:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
    char *p, *q;
    int n = 12;
    p = (char *) malloc(n);
    strcpy(p, "Pascal");
    q = (char *) malloc(n);
    strcpy(q, "Lenguaje C");
    printf("%s, %s\n", p, q);
}
```

```

    free(q); /* la función free libera memoria dinámica
              asignada con la función malloc(); */
    q = p;
    printf("%s, %s\n", p, q);

    return 0;
}

```

Ejemplo 5:

¿Cuál es el resultado que se genera por la ejecución del siguiente programa en Lenguaje C/C++?

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

void sumarAyB(int a, int b, int c)
{
    c = a + b;
}

void sumarAyBprima(int a, int b, int *c)
{
    *c = a + b;
}

int main()
{
    int a = 10, b = 15, c = 0;

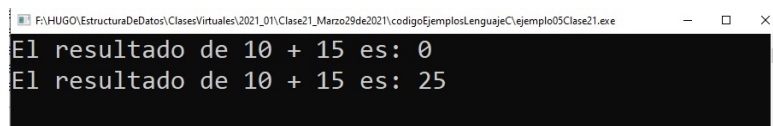
    sumarAyB(a, b, c);
    printf("El resultado de %d + %d es: %d\n", a, b, c);

    sumarAyBprima(a, b, &c);
    printf("El resultado de %d + %d es: %d\n", a, b, c);

    return 0;
}

```

El resultado que se genera por la ejecución del programa es el siguiente:



```

F:\HUGO\EstructuraDeDatos\Clases Virtuales\2021_01\Clase21_Marzo29de2021\codigoEjemplosLenguajeC\ejemplo05Clase21.exe
El resultado de 10 + 15 es: 0
El resultado de 10 + 15 es: 25

```

Figura 5: Resultado generado por el programa del Ejemplo 5.

La justificación del resultado generado se encuentra a continuación.

En la función `void sumarAyB(int a, int b, int c)` los tres parámetros se reciben por valor, por lo tanto el valor que se le asigna a la variable `c` solo está en el alcance de la función y no está reflejado por fuera de ella, por lo tanto en la función `main()` su valor sigue siendo de 0.

En la función `void sumarAyBprima(int a, int b, int *c)` las variables `a` y `b` se reciben por valor, mientras que la variable `c` se recibe por referencia, por lo tanto el valor que se le asigna a la variable `c` se ve reflejado en la función `main()` donde su valor ahora es 25.

Concepto de Lista

Una lista es un conjunto de variables encadenadas en sí a través de un apuntador. Cada variable se denomina nodo. Un nodo es una variable compuesta por dos partes; la información (key) y un apuntador a la próxima variable.

Ejemplo 6:

En el siguiente programa se crea una lista que contiene los números 1, 2 y 3.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
};

int main()
{
    struct node *head, *q;
    head = (struct node *) malloc(sizeof(struct node));
    head->key = 1;
    q = (struct node *) malloc(sizeof(struct node));
    q->key = 2;
    head->next = q;
    q->next = (struct node *) malloc(sizeof(struct node));
    q->next->key = 3;
    q->next->next = NULL;
    printf(" %d -> %d -> %d\n",
           head->key, head->next->key, head->next->next->key);

    return 0;
}
```

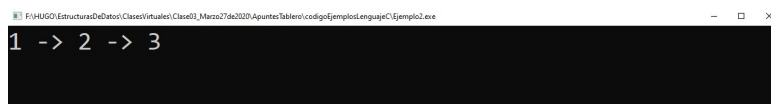


Figura 6: Salida del programa en en cual se trabaja el Ejemplo 6.

Ejemplo 7:

En el siguiente programa se crea una lista que contiene los números del 1 al 100, después se recorre la lista desde el primero hasta el último nodo imprimiendo la información, por último se recorre la lista borrando uno a uno cada nodo.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
};
```

```

int main()
{
    struct node *head, *newNode, *actualNode;
    int n = 100;
    head = NULL;

    while(n >= 1)
    {
        newNode = (struct node *) malloc(sizeof(struct node));
        newNode->key = n;
        newNode->next = head;
        head = newNode;
        n--;
    }

    actualNode = head;

    while(actualNode != NULL)
    {
        printf("%d -> ", actualNode->key);
        actualNode = actualNode->next;
    }
    printf("NULL\n");

    while(head != NULL)
    {
        actualNode = head;
        head = head->next;
        free(actualNode);
    }

    return 0;
}

```

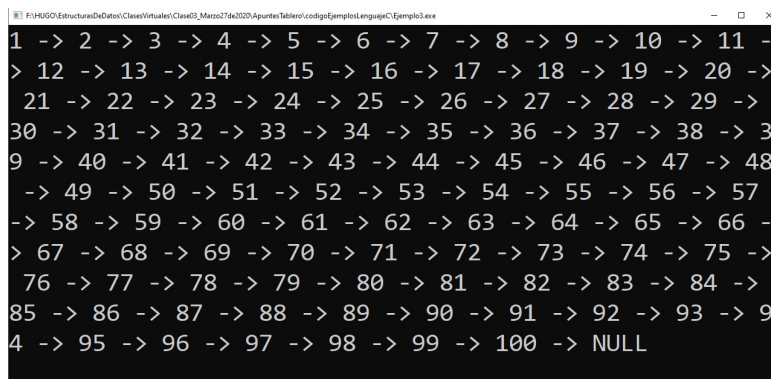


Figura 7: Salida del programa en en cual se trabaja el Ejemplo 7.

Ejemplo 8:

Hacer un programa en Lenguaje C/C++ que realice el mismo trabajo que el realizado por el Ejemplo 7, pero ahora trabajando con funciones, para lo cual el programa debe tener:

1. Una función para crear una lista ordenada de forma ascendente con los primeros n números enteros positivos. La función debe recibir como parámetro el número entero positivo n y debe devolver un puntero a la cabeza de la lista.
2. Una función para imprimir el contenido de la lista, para lo cual la función debe recibir un puntero a la cabeza de la lista.

3. Una función para borrar todos los elementos de la lista, para lo cual la función debe recibir un puntero a la cabeza de la lista y debe retornar el puntero de la cabeza de la lista apuntando a una lista vacía (apuntando a NULL).

El siguiente programa cumple con los requerimientos pedidos:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
};

struct node *makeLinkedList(int n)
{
    struct node *head, *newNode;
    head = NULL;

    while(n >= 1)
    {
        newNode = (struct node *) malloc(sizeof(struct node));
        newNode->key = n;
        newNode->next = head;
        head = newNode;
        n--;
    }

    return head;
}

void printLinkedList(struct node *head)
{
    struct node *actualNode;
    actualNode = head;

    while(actualNode != NULL)
    {
        printf("%d -> ", actualNode->key);
        actualNode = actualNode->next;
    }
    printf("NULL\n");
}

struct node *deleteLinkedList(struct node *head)
{
    struct node *actualNode;

    while(head != NULL)
    {
        actualNode = head;
        head = head->next;
        free(actualNode);
    }

    return head;
}

int main()
{
    struct node *head;
    int n;

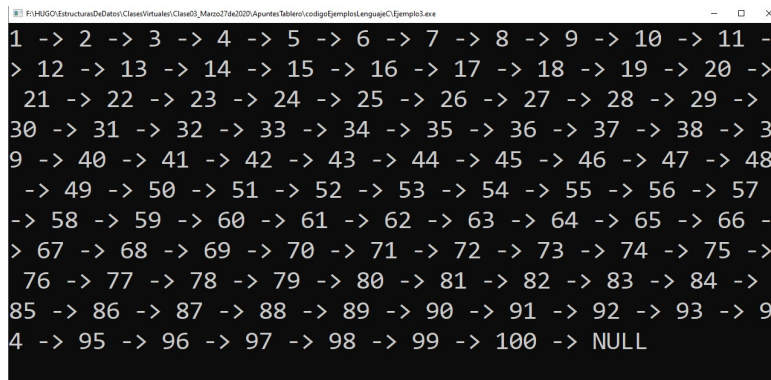
    while(scanf("%d", &n) != EOF)
```

```

{
    head = makeLinkedList(n);
    printLinkedList(head);
    head = deleteLinkedList(head);
    printLinkedList(head);
}

return 0;
}

```



```

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 ->
12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 18 -> 19 -> 20 ->
21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 ->
30 -> 31 -> 32 -> 33 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 ->
40 -> 41 -> 42 -> 43 -> 44 -> 45 -> 46 -> 47 -> 48 ->
49 -> 50 -> 51 -> 52 -> 53 -> 54 -> 55 -> 56 -> 57 ->
58 -> 59 -> 60 -> 61 -> 62 -> 63 -> 64 -> 65 -> 66 ->
67 -> 68 -> 69 -> 70 -> 71 -> 72 -> 73 -> 74 -> 75 ->
76 -> 77 -> 78 -> 79 -> 80 -> 81 -> 82 -> 83 -> 84 ->
85 -> 86 -> 87 -> 88 -> 89 -> 90 -> 91 -> 92 -> 93 -> 94 ->
95 -> 96 -> 97 -> 98 -> 99 -> 100 -> NULL

```

Figura 8: Salida por pantalla del programa de listas que hace uso de funciones.

Ejemplo 9:

Hacer un programa en Lenguaje C/C++ que permita trabajar con listas ordenadas de forma ascendente, para lo cual el programa debe tener:

1. Una función que permita **insertar** un elemento en una lista que está ordenada de forma ascendente, la lista obviamente debe seguir conteniendo sus elementos en orden ascendente después de la inserción del elemento. El prototipo de la función **tiene** que ser el siguiente, recibir como parámetros el puntero a la cabeza de la lista y el número entero a insertar, y devolver el puntero a la cabeza de la lista.
2. Una función para **imprimir** el contenido de la lista, para lo cual la función **tiene** que recibir un puntero a la cabeza de la lista.
3. Una función que permita **borrar** la primera ocurrencia de un elemento en una lista que está ordenada de forma ascendente, la lista obviamente seguirá conteniendo sus elementos en orden ascendente después del borrado del elemento. El prototipo de la función **tiene** que ser el siguiente, recibir como parámetros el puntero a la cabeza de la lista y el número entero a borrar, y devolver el puntero a la cabeza de la lista.

La solución en Lenguaje C/C++ es la siguiente:

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
};

struct node *InsertElementInAscendentLinkedList(struct node *head, int element)
{
    struct node *newNode, *previous, *actual;

```



```

newNode = (struct node *) malloc(sizeof(struct node));
newNode->key = element;

if(head == NULL)
{
    newNode->next = head;
    head = newNode;
}
else
{
    if(element <= head->key)
    {
        newNode->next = head;
        head = newNode;
    }
    else
    {
        previous = head;
        actual = head->next;

        while(actual != NULL)
        {
            if(element > actual->key)
            {
                previous = actual;
                actual = actual->next;
            }
            else
                break;
        }

        newNode->next = actual;
        previous->next = newNode;
    }
}

return head;
}

void PrintAscendentLinkedList(struct node *head)
{
    struct node *actual;
    actual = head;

    while(actual != NULL)
    {
        printf("%d -> ", actual->key);
        actual = actual->next;
    }

    printf("NULL\n");
}

struct node *DeleteElementOfAscendentLinkedList(struct node *head, int element)
{
    struct node *previous, *actual;

    if(head == NULL)
        printf("The ascendent linked list is empty.\n");
    else
    {
        if(element < head->key)
            printf("The %d is not in the ascendent linked list.\n", element);
        else
        {
            if(element == head->key)
            {
                actual = head;

```

```

        head = head->next;
        free(actual);
    }
    else
    {
        previous = head;
        actual = head->next;

        while(actual != NULL)
        {
            if(element > actual->key)
            {
                previous = actual;
                actual = actual->next;
            }
            else
                break;
        }

        if(actual == NULL)
            printf("The %d is not in the ascendent linked list.\n", element);
        else
        {
            if(actual->key != element)
                printf("The %d is not in the ascendent linked list.\n", element);
            else
            {
                previous->next = actual->next;
                free(actual);
            }
        }
    }
}

return head;
}

int main()
{
    int operation, element;
    struct node *head;
    head = NULL;

    while(scanf("%d %d", &operation, &element) != EOF)
    {
        if(operation == 1) /* Insert */
        {
            head = InsertElementInAscendentLinkedList(head, element);
            PrintAscendentLinkedList(head);
        }
        else
        {
            if(operation == 2) /* Delete */
            {
                head = DeleteElementOfAscendentLinkedList(head, element);
                PrintAscendentLinkedList(head);
            }
            else
                printf("Bad use. \n 1. Insert \n 2. Delete\n");
        }
    }
    return 0;
}

```

```
F:\HUGO\EstructuraDeDatos\Clases Virtuales\2021_01\Clase23_Abril05de2021\codigoEjemplos\enguaje\C\ejercicio02\Clase23_ListasAscendentes.exe
1 10
10 -> NULL
1 5
5 -> 10 -> NULL
1 20
5 -> 10 -> 20 -> NULL
1 12
5 -> 10 -> 12 -> 20 -> NULL
2 5
10 -> 12 -> 20 -> NULL
2 12
10 -> 20 -> NULL
2 20
10 -> NULL
1 25
10 -> 25 -> NULL
2 20
The 20 is not in the ascendent linked list.
10 -> 25 -> NULL
2 30
The 30 is not in the ascendent linked list.
10 -> 25 -> NULL
```

Figura 9: Salida por pantalla del programa que manipula una lista ordenada ascendentemente.

Reto de Programación: “Sumar Todos”¹

Si!, el nombre del problema refleja el trabajo a realizar; es justamente sumar un conjunto de números. Usted puede sentirse motivado a escribir un programa en su lenguaje de programación favorito para sumar todo el conjunto de números. Tal problema debe ser algo muy sencillo para su nivel de programación. Por este motivo le vamos a poner un poco de sabor y vamos a volver el problema mucho más interesante.

Ahora, la operación de suma tiene un costo, y el costo es el resultado de sumar dos números. Por lo tanto, el costo de sumar 1 y 10 es 11. Por ejemplo, si usted quiere sumar los números 1, 2 y 3, hay tres formas de hacerlo:

- Forma 1:
 $1 + 2 = 3$, costo = 3
 $3 + 3 = 6$, costo = 6
Costo total = 9
- Forma 2:
 $1 + 3 = 4$, costo = 4
 $2 + 4 = 6$, costo = 6
Costo total = 10
- Forma 3:
 $2 + 3 = 5$, costo = 5
 $1 + 5 = 6$, costo = 6
Costo total = 11

Yo creo que usted ya ha entendido su misión, sumar un conjunto de números enteros tal que el costo de la operación suma sea mínimo.

¹<https://www.hackerrank.com/contests/utp-open-2018/challenges/add-all-2-1>

Formato de Entrada:

La entrada contiene múltiples casos de prueba. Cada caso de prueba debe comenzar con un número entero positivo N ($2 \leq N \leq 5000$), seguido por N números enteros positivos (todos son mayores o iguales a 1 y menores o iguales a 100000). La entrada es finalizada por un caso donde el valor de N es igual a cero. Este caso no debe ser procesado.

Formato de Salida:

Para cada caso de prueba imprimir una sola línea con un número entero positivo que representa el costo total mínimo de la operación suma después de sumar los N números.

Ejemplo de Entrada:

```
3
1 2 3
4
1 2 3 4
5
5 4 3 2 1
0
```

Ejemplo de Salida:

```
9
19
33
```

Solución del reto “Sumar Todos”

Se plantea una solución utilizando listas ordenadas de forma ascendente. Solución que desde el punto de vista de complejidad computación en tiempo de ejecución es muy ineficiente. En el peor de los casos la inserción o borrado de un elemento tiene que recorrer uno a uno los n elementos de la lista ordenada, por lo tanto dichas operaciones tienen un costo computacional de $O(n)$. Lo más costo en tiempo de ejecución en la siguiente solución es insertar los n elementos en la lista ordenada, donde, insertar n elementos en la lista ordenada tiene un costo computacional de $n \cdot O(n) = O(n^2)$. En el reto de programación n como máximo toma un valor de 5000, por lo tanto para cada caso de prueba el costo computacional es $O(n^2) = O(5000^2) = O(25000000) = O(2,5 \cdot 10^7) = O(10^7)$.

La solución en Lenguaje C/C++ utilizando *Listas Ordenadas de Forma Ascendente* es:

```
/* **** */
/* Source: UVA Online Judge */
/* Name Problem: 10954 - Add All. */
/* Problem Setter: Md. Kamruzzaman (KZaman) */
/* **** */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define infinite 2147483647

struct node
{
    int key;
    struct node *next;
};

struct node *InsertElementInAscendentLinkedList(struct node *head, int element)
```

```

{
    struct node *newNode, *previous, *actual;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(head == NULL)
    {
        newNode->next = head;
        head = newNode;
    }
    else
    {
        if(element <= head->key)
        {
            newNode->next = head;
            head = newNode;
        }
        else
        {
            previous = head;
            actual = head->next;

            while(actual != NULL)
            {
                if(element > actual->key)
                {
                    previous = actual;
                    actual = actual->next;
                }
                else
                    break;
            }

            newNode->next = actual;
            previous->next = newNode;
        }
    }

    return head;
}

struct node *DeleteElementOfAscendentLinkedList(struct node *head, int element)
{
    struct node *previous, *actual;

    if(head == NULL)
        printf("The ascendent linked list is empty.\n");
    else
    {
        if(element < head->key)
            printf("The %d is not in the ascendent linked list.\n", element);
        else
        {
            if(element == head->key)
            {
                actual = head;
                head = head->next;
                free(actual);
            }
            else
            {
                previous = head;
                actual = head->next;

                while(actual != NULL)
                {
                    if(element > actual->key)

```

```

        {
            previous = actual;
            actual = actual->next;
        }
        else
            break;
    }

    if(actual == NULL)
        printf("The %d is not in the ascendent linked list.\n", element);
    else
    {
        if(actual->key != element)
            printf("The %d is not in the ascendent linked list.\n", element);
        else
        {
            previous->next = actual->next;
            free(actual);
        }
    }
}

}

return head;
}

long long int solver(struct node *head)
{
    int value;
    long long result=0;

    while(head->next != NULL)
    {
        value = head->key;
        head = DeleteElementOfAscendentLinkedList(head, head->key);
        value += head->key;
        head = DeleteElementOfAscendentLinkedList(head, head->key);
        head = InsertElementInAscendentLinkedList(head, value);
        result += value;
    }
    free(head);
    return result;
}

int main()
{
    int n, i, key;
    struct node *head;

    while(scanf("%d", &n) && (n > 0))
    {
        head = NULL;

        for(i=1; i<=n; i++)
        {
            scanf("%d", &key);
            head = InsertElementInAscendentLinkedList(head, key);
        }

        printf("%lld\n", solver(head));
    }

    return 0;
}

```

```
E:\HUGO\EstructuraDeDatos\ x + v
3
1 2 3
9
4
1 2 3 4
19
5
5 4 3 2 1
33
0
```

Figura 10: Salida del programa para el reto “Sumar Todos”.

Listas Circulares Simplemente Enlazadas

Una lista circular simplemente enlazada es aquella donde el campo `next` del último elemento en la lista (`tail`) apunta al primer elemento en la lista.

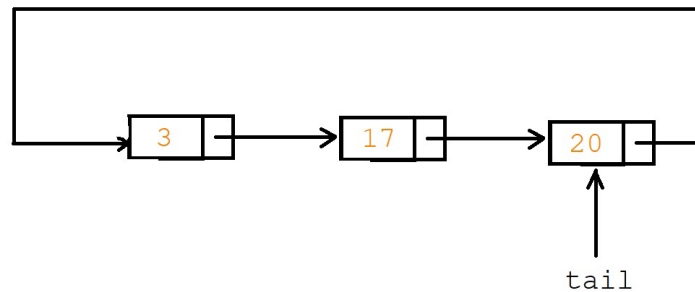


Figura 11: Ejemplo de lista circular simplemente enlazada.

Inserción de un Elemento en una lista Circular Simplemente Enlazada

Por el momento se considerará que después de insertar un elemento en la lista circular éste quedará en la última posición de la lista, adicionalmente, el puntero a la lista será `tail` y el cual apuntará al último nodo en la lista, esto resulta ser muy estratégico ... el siguiente del último nodo es el primer nodo en la lista!

Inicialmente tenemos un lista circular vacía, como se ilustra en la siguiente figura 12:



Figura 12: Inicialmente la lista circular está vacía **tail** apunta a **NULL**.

Si se quiere insertar el número 3 en la lista circular entonces esta debería de quedar como se ilustra en la siguiente figura 13:

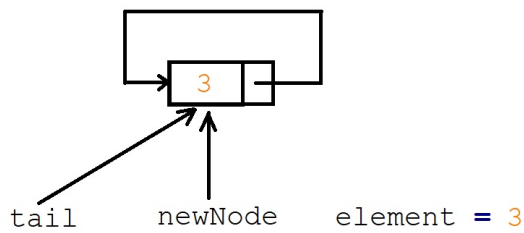


Figura 13: Lista circular que contiene un solo nodo con el valor 3.

El código en Lenguaje C/C++ que considera éste caso es:

```
struct node *insertElementInCircularLinkedList(struct node *tail, int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        /* falta el código para el caso en el cual la lista circular contiene
        al menos un nodo. */
    }

    return tail;
}
```

Ahora considerar el caso en el cual se quiere insertar el número 17 en la lista circular que contiene un solo nodo con el valor de 3, el paso a paso de lo que se debería de hacer se ilustra en las figuras 14, 15 y 16:


```
newNode->next = tail->next;
```

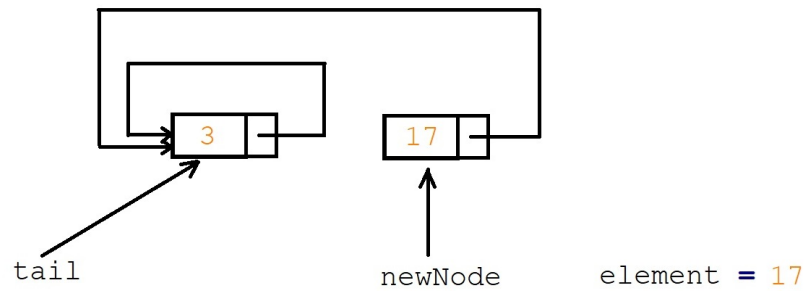


Figura 14: Instrucción que enlaza el campo **next** del **newNode** al nodo apuntado por **tail->next**.

```
tail->next = newNode;
```

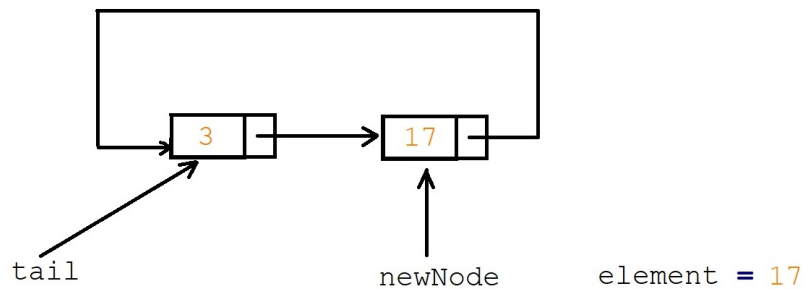


Figura 15: Instrucción que enlaza el campo **next** del **tail** al nodo apuntado por el **newNode**.

```
tail = newNode;
```

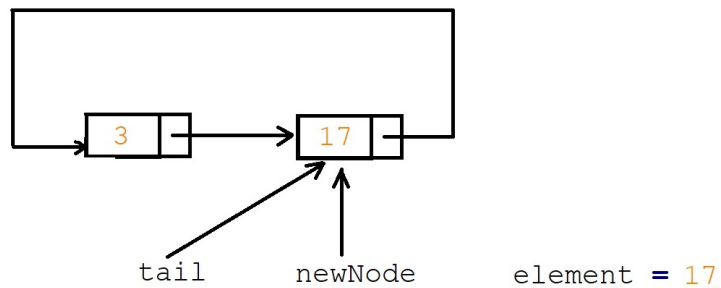


Figura 16: Instrucción que actualiza el puntero **tail** al **newNode**, al nuevo último.

El código en Lenguaje C/C++ de la función completa es:

```
struct node *insertElementInCircularLinkedList(struct node *tail, int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;
```

```

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

```

Ejercicio 1:

Realizar el paso a paso para insertar el número 20 en la lista circular que se ilustra en las figura 17:

```
insertElementInCircularLinkedList(tail, 20);
```

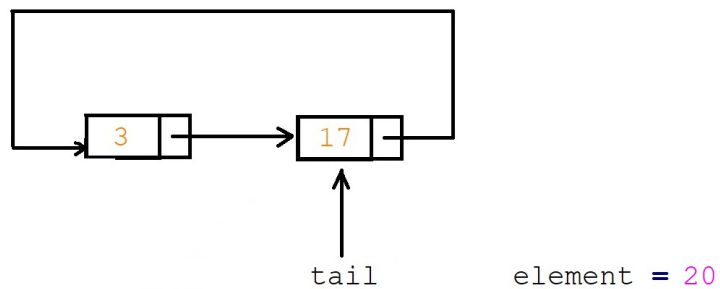


Figura 17: Insertar el número 20 en la lista circular.

Borrado del primer nodo en una lista Circular Simplemente Enlazada

El código en Lenguaje C/C++ para borrar el primer nodo de la lista es el siguiente:

```

struct node *deleteFirstNodeInCircularLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular linked list is empty.\n");
    else
    {
        if(tail == tail->next)
        {
            free(tail);
            tail = NULL;
        }
        else
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            free(firstNode);
        }
    }
    return tail;
}

```

Impresión desde el primero hasta el último de los elementos de una Lista Circular Simplemente Enlazada

El código en Lenguaje C/C++ para imprimir los elementos de una lista circular simplemente enlazada es el siguiente:

```
void printCircularLinkedList(struct node *tail)
{
    struct node *actualNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        actualNode = tail->next;

        while(actualNode != tail)
        {
            printf("%d -> ", actualNode->key);
            actualNode = actualNode->next;
        }
        printf("%d ... \n", tail->key);
    }
}
```

Programa completo para el mantenimiento de una Lista Circular Simplemente Enlazada

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
};

struct node *insertElementInCircularLinkedList(struct node *tail, int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

struct node *deleteFirstNodeInCircularLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular linked list is empty.\n");
```

```

    else
    {
        if(tail == tail->next)
        {
            free(tail);
            tail = NULL;
        }
        else
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            free(firstNode);
        }
    }
    return tail;
}

void printCircularLinkedList(struct node *tail)
{
    struct node *actualNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        actualNode = tail->next;

        while(actualNode != tail)
        {
            printf("%d -> ", actualNode->key);
            actualNode = actualNode->next;
        }
        printf("%d ... \n", tail->key);
    }
}

int main()
{
    struct node *tail;
    int operation, element;

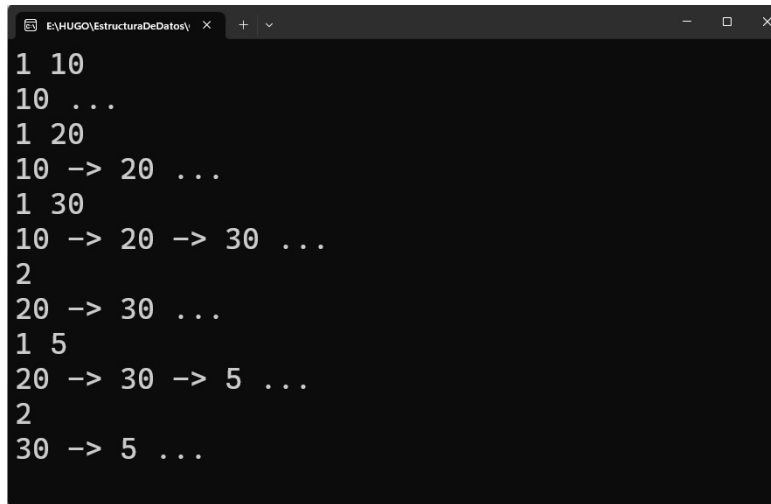
    tail = NULL;

    while(scanf("%d", &operation) != EOF)
    {
        if(operation == 1) /* Insert */
        {
            scanf("%d", &element);
            tail = insertElementInCircularLinkedList(tail, element);
            printCircularLinkedList(tail);
        }

        else
        {
            if(operation == 2) /* Delete */
            {
                tail = deleteFirstNodeInCircularLinkedList(tail);
                printCircularLinkedList(tail);
            }
            else
                printf("Bad use. \n 1. Insert\n 2. Delete\n");
        }
    }

    return 0;
}

```



```
E:\HUGO\EstructuraDeDatos\ x + -
1 10
10 ...
1 20
10 -> 20 ...
1 30
10 -> 20 -> 30 ...
2
20 -> 30 ...
1 5
20 -> 30 -> 5 ...
2
30 -> 5 ...
```

Figura 18: Salida del programa para el mantenimiento de listas circulares simplemente enlazadas.

Reto de Programación: “El Problema de Josefo”²

El **Problema de Josefo** (Josephus) hace referencia a *Flavio Josefo*, un historiador judío que vivió en el siglo I. Según lo que cuenta Josefo, un grupo de 40 soldados (entre los cuales se encontraba él) estaban atrapados en una cueva, rodeados de romanos. Decidieron hacer un pacto de muerte antes que ser capturados por sus enemigos, donde echarían a suertes quién mataría a quién. El último que quedara debería suicidarse.

Formalmente el problema de Josefo es: hay n personas paradas en un círculo (mirando al interior de éste) esperando a ser ejecutadas. Una de las personas se toma como punto de partida (posición uno) y a partir de ella se enumeran a todos los demás de forma consecutiva siguiendo el sentido de las manecillas del reloj, desde 1 hasta n . Posteriormente, se comienza a contar desde la persona de la posición 1 evitando a $k - 1$ personas y la persona número k es ejecutada. Ahora se comienza a contar nuevamente desde la persona que originalmente estaba en la posición $k + 1$, evitando a $k - 1$ personas y la persona número k es ejecutada. La eliminación continúa siguiendo siempre el sentido de las manecillas del reloj alrededor del círculo, el cual se hace cada vez más pequeño, hasta que sólo queda la última persona, la cual decidirá si vive o muere, si se entrega a los romanos o si cumple el pacto de muerte y se suicida, obviamente conocemos la decisión que tomó Josefo, gracias a la cual se conoce esta historia.

Josefo necesita de tu ayuda para poder determinar la posición en el círculo en la cual debe pararse para ser el último soldado que quede con vida y poder entregarse a los romanos.

Formato de Entrada:

La entrada contiene varios casos de prueba. Cada caso está compuesto de una sola línea, que contienen dos números enteros positivos n ($1 \leq n \leq 10^4$) y k ($1 \leq k \leq n$) que representan respectivamente el número de soldados atrapados en la cueva y valor del movimiento para asesinar soldados en el círculo. La entrada finaliza con un caso de prueba que contiene dos ceros, el cual no debe ser procesado.

Formato de Salida:

Para cada caso de prueba de la entrada, su programa debe imprimir en una sola línea el número entero que representa posición en el círculo en la cual se debe parar Josefo para ser el último soldado que quede con vida.

²<https://www.hackerrank.com/contests/utp-open-2018/challenges/josephuss-problem>

Ejemplo de Entrada:

```
1 1
10 1
10 5
10 10
5 5
5 4
0 0
```

Ejemplo de Salida:

```
1
10
3
8
2
1
```

Solución del reto “Problema de Josefo”

Se plantea una solución utilizando listas circulares simplemente enlazadas. Para cada caso de prueba se insertan los elementos del 1 al n en la lista circular. Ahora una observación clave, la función que elimina el primer nodo de la lista circular en esencia elimina el nodo siguiente del nodo apuntado por `tail`, entonces es simplemente mover el puntero `tail` al nodo que precede al nodo que se debe borrar y hacer uso de dicha función para borrar $n - 1$ elementos de la lista circular. El costo computacional en tiempo de ejecución de la solución es eliminar $n - 1$ nodos de la lista circular realizando $k - 1$ movimientos entre los nodos para garantizar que el nodo a eliminar es el que se encuentra en el movimiento k , es decir, $(n - 1) \cdot (k - 1) = n \cdot k - n - k + 1 = O(n \cdot k) = O(k \cdot n)$, como k está acotado superiormente por n entonces la solución en el peor de los casos es un $O(n^2)$ por caso de prueba.

La solución en Lenguaje C/C++ utilizando *Listas Circulares Simplemente Enlazadas* es:

```
/* **** */
/* Source: Data Structure - UTP */
/* Date: January 1st, 2024. */
/* Name Problem: Josephus's Problem. */
/* Problem Setter: Hugo Humberto Morales P. */
/* **** */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct node
{
    int key;
    struct node *next;
};

struct node *insertElementInCircularLinkedList(struct node *tail, int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
```

```

    {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

struct node *deleteFirstNodeInCircularLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular linked list is empty.\n");
    else
    {
        if(tail == tail->next)
        {
            free(tail);
            tail = NULL;
        }
        else
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            free(firstNode);
        }
    }
    return tail;
}

int main()
{
    struct node *tail;
    int n, k, i, j;
    tail = NULL;

    while(scanf("%d %d", &n, &k) && (n > 0) && (k > 0))
    {
        for(i = 1; i <= n; i++)
            tail = insertElementInCircularLinkedList(tail, i);

        for(i = 1; i < n; i++)
        {
            for(j = 1; j < k; j++)
                tail = tail->next;

            tail = deleteFirstNodeInCircularLinkedList(tail);
        }

        printf("%d\n", tail->key);
        tail = deleteFirstNodeInCircularLinkedList(tail);
    }

    return 0;
}

```

```
D:\HUGO\EstructuraDeDatos\ x + -
1 1
1
10 1
10
10 5
3
10 10
8
5 5
2
5 4
1
0 0
```

Figura 19: Salida del programa para el reto “Problema de Josefo”.

Listas Circulares Doblemente Enlazadas

A la estructura de datos `node` que se ha utilizado en las listas simplemente enlazadas hay que agregarle un puntero del tipo `struct node` al nodo previo, para que de esta forma un nodo tenga un puntero al nodo siguiente y un puntero al nodo previo (nodo anterior). El código en Lenguaje C/C++ de la estructura sería:

```
struct node
{
    int key;
    struct node *prev;
    struct node *next;
};
```

Una lista circular doblemente enlazada es aquella donde el campo `next` del último nodo en la lista (`tail`) apunta al primer nodo y el campo `prev` del primer nodo apunta al último nodo.

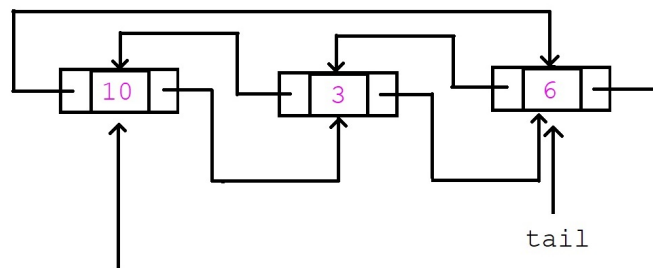


Figura 20: Ejemplo de lista circular doblemente enlazada.

Inserción de un Elemento en una lista Circular Doblemente Enlazada

Por el momento se considerará que después de insertar un elemento en la lista circular doblemente enlazada éste quedará en la última posición de la lista, adicionalmente, el puntero a la lista será `tail` y el cual apuntará al último elemento en la lista, esto resulta ser muy estratégico ... el siguiente del último nodo es el primer nodo en la lista y el nodo previo del primer nodo es el último nodo!

Inicialmente tenemos un lista circular doblemente enlazada vacía, como se ilustra en la figura 21:



Figura 21: Inicialmente la lista circular doblemente enlazada está vacía **tail** apunta a **NULL**.

Si se quiere insertar el número 3 en la lista circular doblemente enlazada entonces ésta debería de quedar como se ilustra en la figura 22:

```
insertElementInCircularDoublyLinkedList(tail, 3);
```

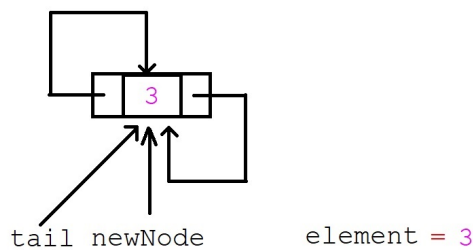


Figura 22: Lista circular doblemente enlazada que contiene un solo nodo con el valor 3.

El código en Lenguaje C/C++ que considera éste caso es:

```
struct node *insertElementInCircularDoublyLinkedList(struct node *tail, int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        newNode->prev = newNode;
        tail = newNode;
    }
    else
    {
        /* falta el código para el caso en el cual la lista circular
           doblemente enlazada contiene al menos un nodo. */
    }

    return tail;
}
```

Ahora considerar el caso en el cual se quiere insertar el número 6 en la lista circular doblemente enlazada que contiene un solo nodo con el valor de 3, el paso a paso de lo que se debería de hacer se ilustra en las figuras 23,

24, 25, 26, 27 y 28:

Primero, se debe pedir de forma dinámica memoria para un nodo nuevo el cual es apuntado por **newNode**, luego se debe asignar en el campo **key** el valor de 6

```
insertElementInCircularDoublyLinkedList(tail, 6);
```

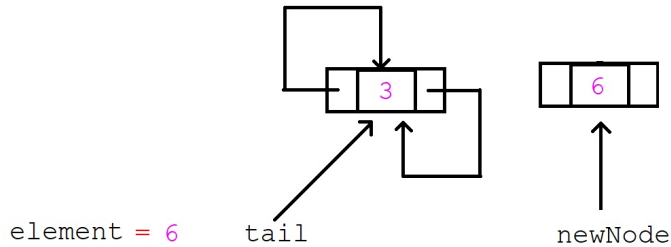


Figura 23: Se genera un nuevo nodo el cual es apuntado por **newNode**, en el campo **key** es almacenado el número 6.

Segundo, el campo **next** del nodo apuntado por **newNode** se pone a apuntar al nodo apuntado por **tail→next**

```
newNode->next = tail->next;
```

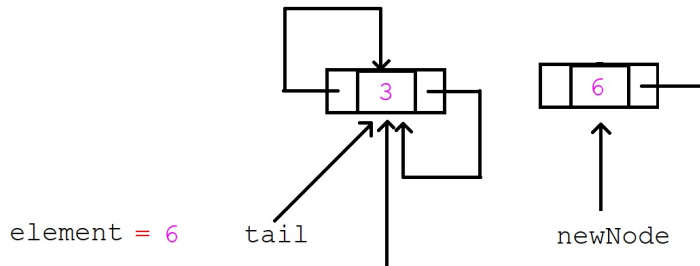


Figura 24: Instrucción que enlaza el campo **next** del **newNode** al nodo apuntado por **tail→next**.

Tercero, el campo **prev** del nodo apuntado por **newNode→next** se pone a apuntar al nodo apuntado por **newNode**

```
newNode->next->prev = newNode;
```

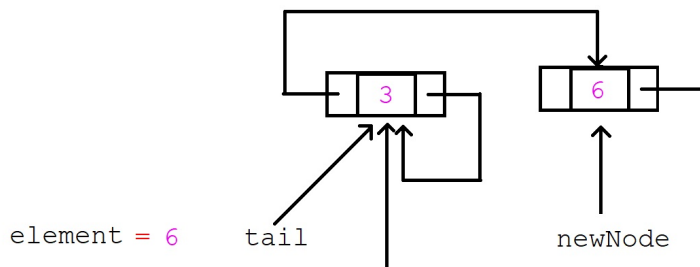


Figura 25: Instrucción que enlaza el campo **prev** del primer nodo de la lista al nodo apuntado por el **newNode**.

Cuarto, el campo **prev** del nodo apuntado por **newNode** se pone a apuntar al nodo apuntado por **tail**

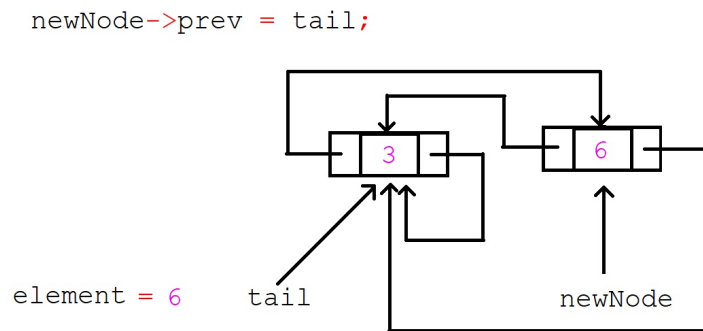


Figura 26: Instrucción que enlaza el campo **prev** del **newNode** al nodo apuntado por **tail**.

Quinto, el campo **next** del nodo apuntado por **tail** se pone a apuntar al nodo apuntado por **newNode**

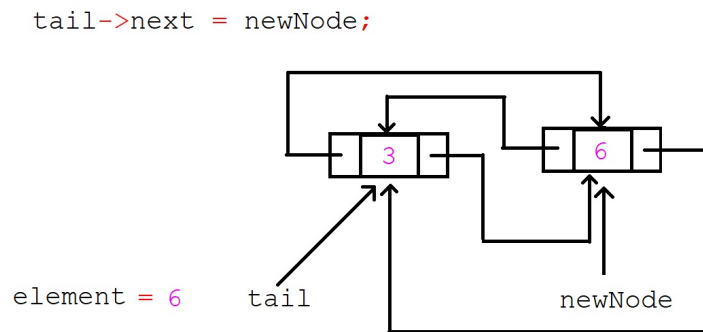


Figura 27: Instrucción que enlaza el campo **next** de **tail** al nodo apuntado por **newNode**.

Sexto, por último, el puntero **tail** se pone a apuntar al nodo apuntado por **newNode**, dicho nodo ahora es el último en la lista circular doblemente enlazada

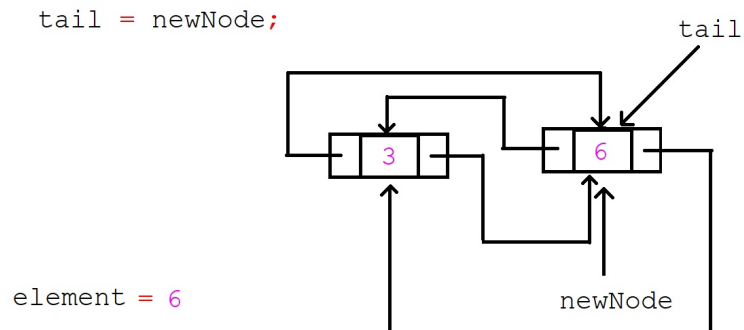


Figura 28: Instrucción que actualiza el puntero **tail** al **newNode**, al nuevo último.

El código en Lenguaje C/C++ de la función completa es:

```
struct node *insertElementInCircularDoublyLinkedList(struct node *tail, int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        newNode->prev = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        newNode->next->prev = newNode;
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}
```

Ejercicio 2:

Realizar el paso a paso para insertar el número 15 en la lista circular doblemente enlazada que se ilustra en la figura 29:

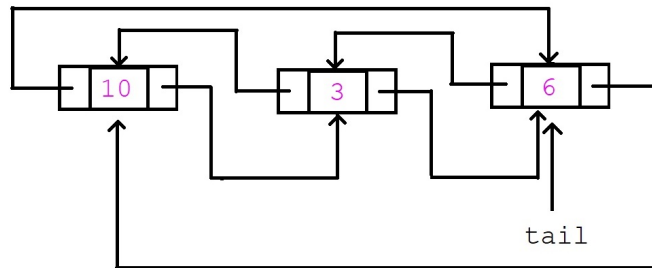


Figura 29: Insertar el número 15 en la lista circular doblemente enlazada.

Borrado del Primer nodo en una lista Circular Doblemente Enlazada

El paso a paso para borrar el primer nodo de una lista circular doblemente enlazada se ilustra en las figuras 30, 31, 32, 33 y 34:

```
deleteFirstNodeInCircularDoublyLinkedList (tail);
```

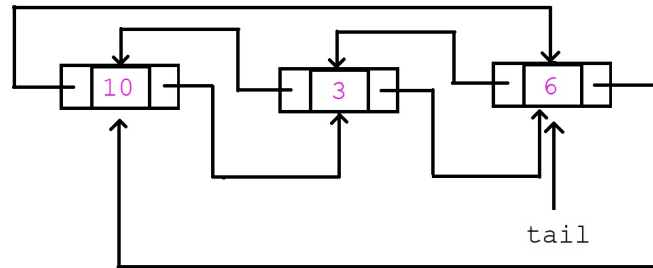


Figura 30: Lista circular doblemente enlazada de la cual se borrará el primer nodo.

```
firstNode = tail->next;
```

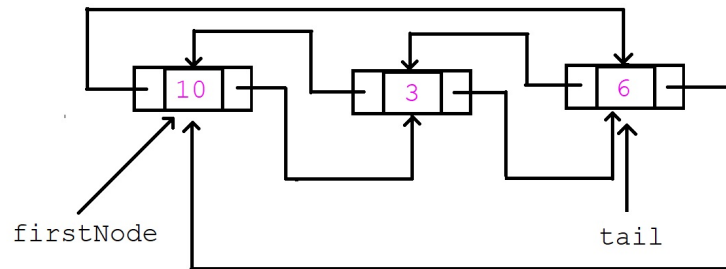


Figura 31: Instrucción que enlaza el puntero **firstNode** al nodo apuntado por **tail→next**.

```
tail->next = firstNode->next;
```

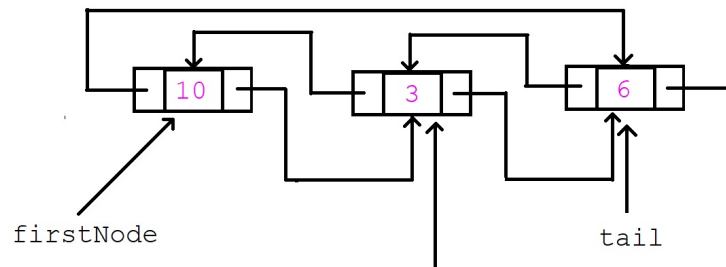


Figura 32: Instrucción que enlaza el campo **next** del último nodo al nodo apuntado por **firstNode→next**.

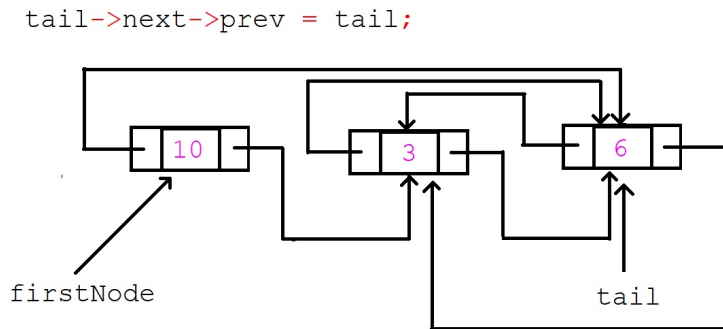


Figura 33: Instrucción que enlaza el campo **prev** del nodo apuntado por **tail**→**next** al nodo apuntado por **tail**.

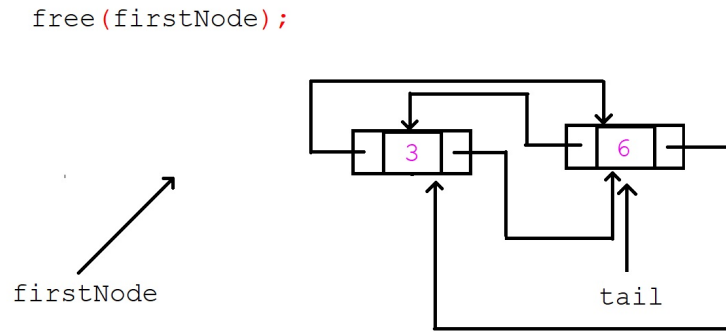


Figura 34: Instrucción para liberar la memoria del nodo apuntado por **firstNode**.

El código en Lenguaje C/C++ para borrar el primer nodo de la lista circular doblemente enlazada es el siguiente:

```
struct node *deleteFirstNodeInCircularDoublyLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular doubly linked list is empty.\n");
    else
    {
        if(tail == tail->next)
        {
            free(tail);
            tail = NULL;
        }
        else
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            tail->next->prev = tail;
            free(firstNode);
        }
    }
    return tail;
}
```

Impresión desde el primero hasta el último de los elementos de una Lista Circular Doblemente Enlazada

El código en Lenguaje C/C++ para imprimir los elementos del primero al último de una lista circular doblemente enlazada es el siguiente:

```
void printFromFirstToLastCircularDoublyLinkedList(struct node *tail)
{
    struct node *actualNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        actualNode = tail->next;

        while(actualNode != tail)
        {
            printf("%d -> ", actualNode->key);
            actualNode = actualNode->next;
        }
        printf("%d ... \n", tail->key);
    }
}
```

Impresión desde el último hasta el primero de los elementos de una Lista Circular Doblemente Enlazada

El código en Lenguaje C/C++ para imprimir los elementos del último al primero de una lista circular doblemente enlazada es el siguiente:

```
void printFromLastToFirstCircularDoublyLinkedList(struct node *tail)
{
    struct node *actualNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        actualNode = tail;

        while(actualNode != tail->next)
        {
            printf("%d -> ", actualNode->key);
            actualNode = actualNode->prev;
        }
        printf("%d ... \n", tail->next->key);
    }
}
```

Programa completo para el mantenimiento de una Lista Circular Doblemente Enlazada

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
    struct node *prev;
};

struct node *insertElementInCircularDoublyLinkedList(struct node *tail, int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        newNode->prev = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        newNode->next->prev = newNode;
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

struct node *deleteFirstNodeInCircularDoublyLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular doubly linked list is empty.\n");
    else
    {
        if(tail == tail->next)
        {
            free(tail);
            tail = NULL;
        }
        else
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            tail->next->prev = tail;
            free(firstNode);
        }
    }

    return tail;
}

void printFromFirstToLastCircularDoublyLinkedList(struct node *tail)
{
    struct node *actualNode;
```



```

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        actualNode = tail->next;

        while(actualNode != tail)
        {
            printf("%d -> ", actualNode->key);
            actualNode = actualNode->next;
        }
        printf("%d ...\n", tail->key);
    }
}

void printFromLastToFirstCircularDoublyLinkedList(struct node *tail)
{
    struct node *actualNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        actualNode = tail;

        while(actualNode != tail->next)
        {
            printf("%d -> ", actualNode->key);
            actualNode = actualNode->prev;
        }
        printf("%d ...\n", tail->next->key);
    }
}

int main()
{
    struct node *tail;
    int operation, element;

    tail = NULL;

    while(scanf("%d", &operation) != EOF)
    {
        if(operation == 1) /* Insert */
        {
            scanf("%d", &element);
            tail = insertElementInCircularDoublyLinkedList(tail, element);
            printFromFirstToLastCircularDoublyLinkedList(tail);
            printFromLastToFirstCircularDoublyLinkedList(tail);
        }
        else
        {
            if(operation == 2) /* Delete */
            {
                tail = deleteFirstNodeInCircularDoublyLinkedList(tail);
                printFromFirstToLastCircularDoublyLinkedList(tail);
                printFromLastToFirstCircularDoublyLinkedList(tail);
            }
            else
                printf("Bad use. \n 1. Insert\n 2. Delete\n");
        }
    }

    return 0;
}

```

```
F:\HUGO\EstructuraDeDatos\Clases Virtuales\2021_02\Clase25_Octubre22de2021\codigoEjemplos\LenguajeC\CircularDoublyLinkedList.exe
1 10
10 ...
10 ...
1 3
10 -> 3 ...
3 -> 10 ...
1 6
10 -> 3 -> 6 ...
6 -> 3 -> 10 ...
1 15
10 -> 3 -> 6 -> 15 ...
15 -> 6 -> 3 -> 10 ...
2
3 -> 6 -> 15 ...
15 -> 6 -> 3 ...
```

Figura 35: Salida del programa para el mantenimiento de listas circulares doblemente enlazadas.

Borrar un elemento en una lista Circular Doblemente Enlazada

El código en Lenguaje C/C++ para borrar la primera ocurrencia de un elemento en una lista circular doblemente enlazada es el siguiente:

```
struct node *deleteElementOfCircularDoublyLinkedList(struct node *tail, int element)
{
    struct node *actualNode;
    int flag = FALSE;

    if(tail == NULL)
        printf("The circular doubly linked list is empty.\n");
    else
    {
        actualNode = tail->next;

        while((actualNode != tail) && (flag == FALSE))
        {
            if(actualNode->key == element)
            {
                flag = TRUE;
                break;
            }
            else
                actualNode = actualNode->next;
        }

        if(flag == FALSE)
        {
            if(tail->key == element)
            {
                flag = TRUE;
                actualNode = tail;
            }
        }

        if(tail == tail->next)
        {
            if(flag == TRUE)
```

```

        {
            free(tail);
            tail = NULL;
        }
        else
            printf("The %d is not in the circular doubly linked list.\n", element);
    }
    else
    {
        if(flag == FALSE)
            printf("The %d is not in the circular doubly linked list.\n", element);
        else
        {
            actualNode->next->prev = actualNode->prev;
            actualNode->prev->next = actualNode->next;
            if(actualNode == tail)
                tail = tail->prev;
            free(actualNode);
        }
    }
}
return tail;
}

```

Programa completo para el mantenimiento de una Lista Circular Doblemente Enlazada en la cual se inserta o se borra un elemento específico

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define TRUE 1
#define FALSE 0

struct node
{
    int key;
    struct node *next;
    struct node *prev;
};

struct node *insertElementInCircularDoublyLinkedList(struct node *tail, int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        newNode->prev = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        newNode->next->prev = newNode;
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

```

```

struct node *deleteElementOfCircularDoublyLinkedList(struct node *tail, int element)
{
    struct node *actualNode;
    int flag = FALSE;

    if(tail == NULL)
        printf("The circular doubly linked list is empty.\n");
    else
    {
        actualNode = tail->next;

        while((actualNode != tail) && (flag == FALSE))
        {
            if(actualNode->key == element)
            {
                flag = TRUE;
                break;
            }
            else
                actualNode = actualNode->next;
        }

        if(flag == FALSE)
        {
            if(tail->key == element)
            {
                flag = TRUE;
                actualNode = tail;
            }
        }

        if(tail == tail->next)
        {
            if(flag == TRUE)
            {
                free(tail);
                tail = NULL;
            }
            else
                printf("The %d is not in the circular doubly linked list.\n", element);
        }
        else
        {
            if(flag == FALSE)
                printf("The %d is not in the circular doubly linked list.\n", element);
            else
            {
                actualNode->next->prev = actualNode->prev;
                actualNode->prev->next = actualNode->next;
                if(actualNode == tail)
                    tail = tail->prev;
                free(actualNode);
            }
        }
    }

    return tail;
}

void printFromFirstToLastCircularDoublyLinkedList(struct node *tail)
{
    struct node *actualNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        actualNode = tail->next;
    }
}

```

```

        while(actualNode != tail)
        {
            printf("%d -> ", actualNode->key);
            actualNode = actualNode->next;
        }
        printf("%d ...\n", tail->key);
    }
}

void printFromLastToFirstCircularDoublyLinkedList(struct node *tail)
{
    struct node *actualNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        actualNode = tail;

        while(actualNode != tail->next)
        {
            printf("%d -> ", actualNode->key);
            actualNode = actualNode->prev;
        }
        printf("%d ...\n", tail->next->key);
    }
}

int main()
{
    struct node *tail;
    int operation, element;

    tail = NULL;

    while(scanf("%d %d", &operation, &element) != EOF)
    {
        if(operation == 1) /* Insert */
        {
            tail = insertElementInCircularDoublyLinkedList(tail, element);
            printFromFirstToLastCircularDoublyLinkedList(tail);
            printFromLastToFirstCircularDoublyLinkedList(tail);
        }
        else
        {
            if(operation == 2) /* Delete */
            {
                tail = deleteElementOfCircularDoublyLinkedList(tail, element);
                printFromFirstToLastCircularDoublyLinkedList(tail);
                printFromLastToFirstCircularDoublyLinkedList(tail);
            }
            else
                printf("Bad use. \n 1. Insert\n 2. Delete\n");
        }
    }

    return 0;
}

```

```
F:\HUGO\EstructuraDeDatos\Clases Virtuales\2021_02\Clase25_Octubre22de2021\codigoEjemplos\LanguageC\CircularDoublyLinkedListV2.exe
1 10
10 ...
10 ...
1 3
10 -> 3 ...
3 -> 10 ...
1 6
10 -> 3 -> 6 ...
6 -> 3 -> 10 ...
1 15
10 -> 3 -> 6 -> 15 ...
15 -> 6 -> 3 -> 10 ...
2 15
10 -> 3 -> 6 ...
6 -> 3 -> 10 ...
2 15
The 15 is not in the circular doubly linked list.
10 -> 3 -> 6 ...
6 -> 3 -> 10 ...
```

Figura 36: Salida del programa para el mantenimiento de listas circulares doblemente enlazadas donde se inserta y se borra un elemento específico.

Reto de Programación: “Rifa de Josefo”³

Queremos hacer una rifa entre los estudiantes de este grupo de Estructura de Datos y Pepito Pérez (estudiante de este curso) sugiere que utilicemos el problema de Josefo para determinar quien es el ganador, pero, ... ustedes y yo sabemos que se puede conocer de antemano la posición ganadora si se conocen los valores de n (el total de estudiantes en la rifa) y k (cantidad de movimientos para ir sacando estudiantes del círculo, de la ronda).

El premio de la rifa es interesante, ... el ganador quedará exonerado del examen final!, por este motivo se propone la siguiente variante al problema de Josefo: “Se toma la lista de clase de los estudiantes de la materia, en la cual se encuentran numerados los estudiantes del 1 al n , se organizan estos números en círculo y se comienza a contar desde el número 1 hasta llegar al valor k (considerar que esto es en el mismo sentido de las manecillas del reloj), el estudiante con el numero k en la lista se retira del círculo y se comienza a contar desde el siguiente estudiante (en este caso sería el que está ubicado en la posición $k + 1$), pero ahora se realizan los k movimientos en el círculo en el sentido contrario al de las manecillas del reloj, se retira el estudiante donde terminó el conteo y se comienza de nuevo el conteo en el estudiante que seguiría en el sentido del conteo, es decir en el que estaría ubicado en la posición $k + 1$. El proceso continua así sucesivamente alternando el sentido horario y anti-horario hasta encontrar el ganador de la rifa”.

Formato de Entrada:

La entrada contiene varios casos de prueba. Cada caso está compuesto de una sola línea, que contienen dos números enteros positivos n ($1 \leq n \leq 10^4$) y k ($1 \leq k \leq n$) que representan respectivamente el número de estudiantes en la rifa y valor del movimiento para descartar estudiantes del círculo. La entrada finaliza con un caso de prueba que contiene dos ceros, el cual no debe ser procesado.

³<https://www.hackerrank.com/contests/utp-open-2018/challenges/josephus-lottery>

Formato de Salida:

Para cada caso de prueba de la entrada, su programa debe imprimir en una sola línea el número que representa en la lista de estudiantes del curso al estudiante ganador.

Ejemplo de Entrada:

```
1 1
10 1
10 5
10 10
5 5
5 4
0 0
```

Ejemplo de Salida:

```
1
6
2
5
4
2
```

Solución del reto “Rifa de Josefo”

Se plantea una solución utilizando listas circulares doblemente enlazadas en las cuales se simula los movimientos en el sentido de las manecillas del reloj al moverse de nodo a nodo con el puntero **next** (siguiente). De forma similar, se simula los movimientos en contra de la manecillas del reloj (movimiento anti-horario) al moverse de nodo a nodo con el puntero **prev** (previo).

Para cada caso de prueba se insertan los elementos del 1 al n en la lista circular. De nuevo tenemos la observación clave que ya habíamos trabajado en el reto de programación del Problema de Josefo, la función que elimina el primer nodo de la lista circular en esencia elimina el nodo siguiente del nodo apuntado por **tail** (en el caso específico de este reto, el siguiente nodo apuntado por **actualNode**), entonces es simplemente alternar los movimientos en sentido horario y anti-horario borrando el nodo que sigue del nodo apuntado por el **actualNode**.

El costo computacional en tiempo de ejecución de la solución es eliminar $n - 1$ nodos de la lista circular doblemente enlazada realizando $k - 1$ movimientos entre los nodos para garantizar que el nodo a eliminar es el que se encuentra en el movimiento k , alternando entre el sentido horario y anti-horario, es decir, $(n - 1) \cdot (k - 1) = n \cdot k - n - k + 1 = O(n \cdot k) = O(k \cdot n)$, como k está acotado superiormente por n entonces la solución en el peor de los casos es un $O(n^2)$ por caso de prueba.

La solución en Lenguaje C/C++ utilizando *Listas Circulares Doblemente Enlazadas* es:

```
/* **** */
/* Source: UTP Open 2015. */
/* Date: April 11th, 2015. */
/* Name Problem: Josephus Lottery. */
/* Problem Setter: Hugo Humberto Morales P. */
/* **** */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define CLOCKWISE 1
#define COUNTERCLOCKWISE 0
```

```

struct node
{
    int key;
    struct node *next;
    struct node *prev;
};

struct node *insertElementInCircularDoublyLinkedList(struct node *tail, int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        newNode->prev = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        newNode->next->prev = newNode;
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

struct node *deleteFirstNodeInCircularDoublyLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular doubly linked list is empty.\n");
    else
    {
        if(tail == tail->next)
        {
            free(tail);
            tail = NULL;
        }
        else
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            tail->next->prev = tail;
            free(firstNode);
        }
    }
    return tail;
}

int main()
{
    struct node *actualNode;
    int n, k, index, direction, move, element;
    actualNode = NULL;

    while(scanf("%d %d", &n, &k) && (n > 0) && (k > 0))
    {
        direction = CLOCKWISE;
        for(index = 1; index <= n; index++)
            actualNode = insertElementInCircularDoublyLinkedList(actualNode, index);
    }
}

```



```

while(actualNode != actualNode->next)
{
    if(direction == CLOCKWISE)
    {
        for(move = 1; move < k; move++)
            actualNode = actualNode->next;

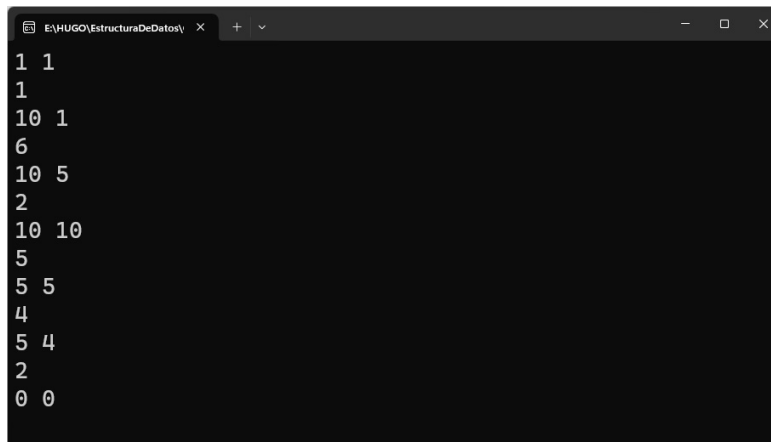
        actualNode = deleteFirstNodeInCircularDoublyLinkedList(actualNode);
        direction = COUNTERCLOCKWISE;
    }
    else
    {
        for(move = 1; move < k; move++)
            actualNode = actualNode->prev;

        actualNode = deleteFirstNodeInCircularDoublyLinkedList(actualNode);
        actualNode = actualNode->prev;
        direction = CLOCKWISE;
    }
}

printf("%d\n", actualNode->key);
actualNode = deleteFirstNodeInCircularDoublyLinkedList(actualNode);
}

return 0;
}

```



```

1 1
1
10 1
6
10 5
2
10 10
5
5 5
4
5 4
2
0 0

```

Figura 37: Salida del programa para el reto “Rifa de Josefo”.

Pilas y Colas

Las pilas y las colas son conjuntos dinámicos en los cuales los elementos son removidos del conjunto por la operación **delete** que es pre-establecida para cada uno de ellos.

En la **pila**, el elemento borrado es el último elemento insertado. En las pilas se implementa una política **LIFO** (**Last-In First-Out**).

En la **cola**, el elemento borrado es siempre el que lleva más tiempo en el conjunto. En las colas se implementa una política **FIFO** (**First-In First-Out**).

Pilas (Stacks)

La operación `insert` en pilas se llama `push`, y la operación `delete` se llama `pop`.

Para la implementación de pilas se utilizará listas simplemente enlazadas donde la inserción (`push`) y borrado (`pop`) de elementos se realizará por el nodo apuntado por el puntero `top`.

Función Push

```
struct node *push(struct node *top, int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;
    newNode->next = top;
    top = newNode;

    return top;
}
```

Función Pop

```
int pop(struct node **top)
{
    struct node *actualNode;
    int element;

    actualNode = *top;
    element = actualNode->key;
    *top = actualNode->next;
    free(actualNode);

    return element;
}
```

Función StackEmpty

```
int stackEmpty(struct node *top)
{
    if(top == NULL)
        return TRUE;
    else
        return FALSE;
}
```

Programa Completo para el Manejo de Pilas

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define TRUE 1
#define FALSE 0

struct node
{
    int key;
    struct node *next;
};
```

```

struct node *push(struct node *top, int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;
    newNode->next = top;
    top = newNode;

    return top;
}

int pop(struct node **top)
{
    struct node *actualNode;
    int element;

    actualNode = *top;
    element = actualNode->key;
    *top = actualNode->next;
    free(actualNode);

    return element;
}

void printStack(struct node *top)
{
    struct node *actualNode;

    if(top == NULL)
        printf("NULL\n");
    else
    {
        actualNode = top;
        while(actualNode != NULL)
        {
            printf("%d ", actualNode->key);
            actualNode = actualNode->next;
        }
        printf("\n");
    }
}

int stackEmpty(struct node *top)
{
    if(top == NULL)
        return TRUE;
    else
        return FALSE;
}

int main()
{
    struct node *top;
    int operation, element;

    top = NULL;

    while(scanf("%d", &operation) != EOF)
    {
        if(operation == 1) /* Push */
        {
            scanf("%d", &element);
            top = push(top, element);
            printStack(top);
        }
    }
}

```


Formato de Entrada:

La primera línea contiene un número entero positivo T ($1 \leq T \leq 100$) denotando el número de casos. Cada caso comienza con una línea que contiene un número entero positivo Q ($1 \leq Q \leq 10^4$) denotando el número de consultas a procesar, seguidas por exactamente Q líneas basadas en el siguiente formato:

- 1 V : Operación **Push** V ($0 \leq V \leq 10^4$), inserta V en el tope de la pila.
- 2: Operación **Pop**, borra el elemento del tope de la pila. Si la pila está vacía entonces no se hace nada.
- 3: Imprimir en una línea el valor absoluto de la diferencia entre el máximo y el mínimo valor que se encuentran en la pila. Si la pila está vacía, imprimir en una línea el mensaje: **Empty!**

Formato de Salida:

Para cada consulta del tipo 3 calcular e imprimir en una sola línea el valor absoluto de la diferencia entre el máximo y mínimo valor que se encuentran en la pila.

Ejemplo de Entrada:

```
2
12
2
1 10
1 15
3
2
2
3
1 18
3
1 10
1 1
3
3
3
1 999
3
```

Ejemplo de Salida:

```
5
Empty!
0
17
Empty!
0
```

Solución del reto “Consultas sobre la Pila”

Se plantea una solución utilizando la estructura de datos Pila. Las funciones **Push** y **Pop** (insertar y borrar) en la pila tienen un costo computacional en tiempo de ejecución de $O(1)$. También se puede consultar el valor absoluto de la diferencia entre los valores máximo y mínimo de la pila en un tiempo de ejecución de $O(1)$, donde simplemente es hacer la diferencia entre el valor máximo (**maxValue**) y el valor mínimo valor (**minValue**) del nodo que se encuentra en el tope de la pila. Obviamente, dichos campos fueron adicionados previamente a la estructura nodo (**node**). En el momento de insertar un nodo en el tope de la pila (operación **Push**) los valores de máximo y mínimo únicamente se determinan al comparar los valores de máximo y mínimo del nodo que se

encuentra en el tope de la pila con respecto al valor del elemento a insertar.

El costo computacional en tiempo de ejecución por cada caso de prueba se determina de la siguiente forma, se realizan Q operaciones sobre la pila, cada una de ellas con un costo de $O(1)$, por lo tanto, $Q \cdot O(1) = O(Q)$. Correr todos los T casos de prueba tiene un costo de $T \cdot O(Q) = O(T \cdot Q)$, como T esta acotado superiormente por $100 = 10^2$ y Q está acotado superiormente por 10^4 entonces la solución planteada en tiempo de ejecución es $O(T \cdot Q) = O(10^2 \cdot 10^4) = O(10^6)$.

La siguiente es la solución en Lenguaje C/C++ utilizando la estructura de datos *Pila*:

```
/* **** */
/* Source: ICPC Centroamerica 2020. */
/* Date: January 30th, 2021. */
/* Name Problem: Queries on the Stack. */
/* Problem Setter: Yonny Mondelo Hernandez. */
/* **** */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define TRUE 1
#define FALSE 0

struct node
{
    int key;
    int maxValue;
    int minValue;
    struct node *next;
};

struct node *push(struct node *top, int element, int maxValue, int minValue)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;
    newNode->maxValue = maxValue;
    newNode->minValue = minValue;
    newNode->next = top;
    top = newNode;

    return top;
}

int pop(struct node **top)
{
    struct node *actualNode;
    int element;

    actualNode = *top;
    element = actualNode->key;
    *top = actualNode->next;
    free(actualNode);

    return element;
}

int stackEmpty(struct node *top)
{
    if(top == NULL)
        return TRUE;
    else
        return FALSE;
}
```

```

}

int myMaximum(int value1, int value2)
{
    int maximum = value1;
    if(maximum < value2)
        maximum = value2;
    return maximum;
}

int myMinimum(int value1, int value2)
{
    int minimum = value1;
    if(minimum > value2)
        minimum = value2;
    return minimum;
}

struct node *deleteStack(struct node *top)
{
    struct node *actualNode;
    actualNode = top;

    while(actualNode != NULL)
    {
        top = actualNode->next;
        free(actualNode);
        actualNode = top;
    }

    return top;
}

int main()
{
    struct node *top;
    int operation, element, totalCases, idCase;
    int totalQueries, idQuery, maxValue, minValue;
    top = NULL;

    scanf("%d", &totalCases);
    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        scanf("%d", &totalQueries);
        for(idQuery = 1; idQuery <= totalQueries; idQuery++)
        {
            scanf("%d", &operation);
            if(operation == 1) /* Push */
            {
                scanf("%d", &element);
                if(stackEmpty(top) == TRUE)
                {
                    maxValue = element;
                    minValue = element;
                }
                else
                {
                    maxValue = myMaximum(element, top->maxValue);
                    minValue = myMinimum(element, top->minValue);
                }

                top = push(top, element, maxValue, minValue);
            }
            else
            {
                if(operation == 2) /* Pop */
                {
                    if(stackEmpty(top) != TRUE)

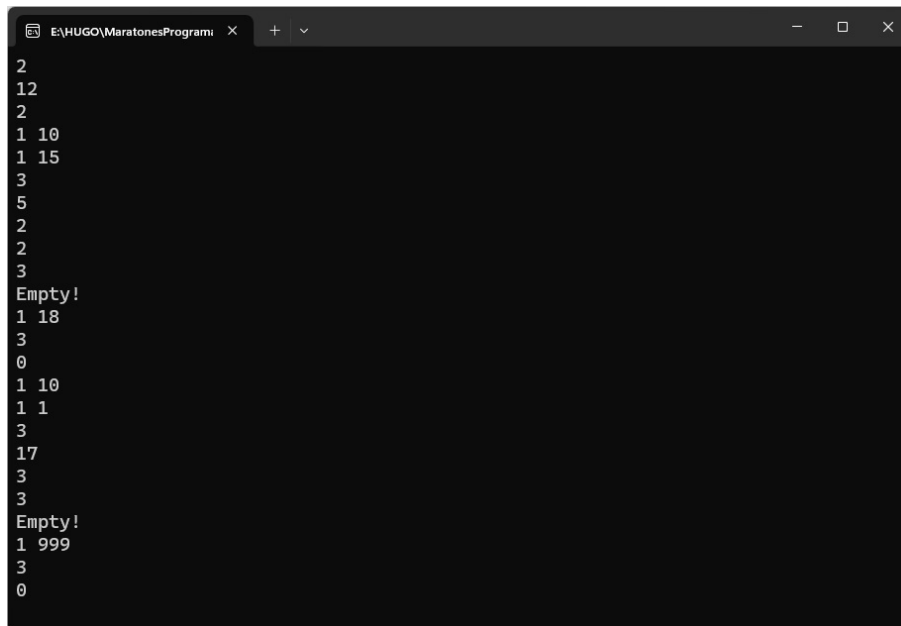
```

```

        element = pop(&top);
    }
    else
    {
        if(operation == 3)
        {
            if(stackEmpty(top) == TRUE)
                printf("Empty!\n");
            else
                printf("%d\n", top->maxValue - top->minValue);
        }
    }
}
}
top = deleteStack(top);
}

return 0;
}

```



```

2
12
2
1 10
1 15
3
5
2
2
3
Empty!
1 18
3
0
1 10
1 1
3
17
3
3
Empty!
1 999
3
0

```

Figura 39: Salida del programa para el reto “Consultas sobre la Pila”.

Colas (Queues)

La operación **insert** en colas se llama **enqueue**, y la operación **delete** se llama **dequeue**.

Para la implementación de colas se utilizará listas simplemente enlazadas circulares donde la inserción (**enqueue**) y borrado (**dequeue**) de elementos se realizará por el nodo apuntado por el puntero **tail**.

Función Enqueue

```

struct node *enqueue(struct node *tail, int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

```



```

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

```

Función Dequeue

```

int dequeue(struct node **tail)
{
    struct node *firstNode;
    int element;

    if(*tail == (*tail)->next)
    {
        element = (*tail)->key;
        free(*tail);
        *tail = NULL;
    }
    else
    {
        firstNode = (*tail)->next;
        element = firstNode->key;
        (*tail)->next = firstNode->next;
        free(firstNode);
    }

    return element;
}

```

Función QueueEmpty

```

int queueEmpty(struct node *tail)
{
    if(tail == NULL)
        return TRUE;
    else
        return FALSE;
}

```

Programa Completo para el Manejo de Colas

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define TRUE 1
#define FALSE 0

struct node
{
    int key;
    struct node *next;
}

```

```

};

struct node *enqueue(struct node *tail, int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

int dequeue(struct node **tail)
{
    struct node *firstNode;
    int element;

    if(*tail == (*tail)->next)
    {
        element = (*tail)->key;
        free(*tail);
        *tail = NULL;
    }
    else
    {
        firstNode = (*tail)->next;
        element = firstNode->key;
        (*tail)->next = firstNode->next;
        free(firstNode);
    }

    return element;
}

int queueEmpty(struct node *tail)
{
    if(tail == NULL)
        return TRUE;
    else
        return FALSE;
}

void printQueue(struct node *tail)
{
    struct node *actualNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        actualNode = tail->next;

        while(actualNode != tail)
        {
            printf("%d ", actualNode->key);
            actualNode = actualNode->next;
        }
    }
}

```

```

        printf("%d\n", tail->key);
    }
}

int main()
{
    struct node *tail;
    int operation, element;

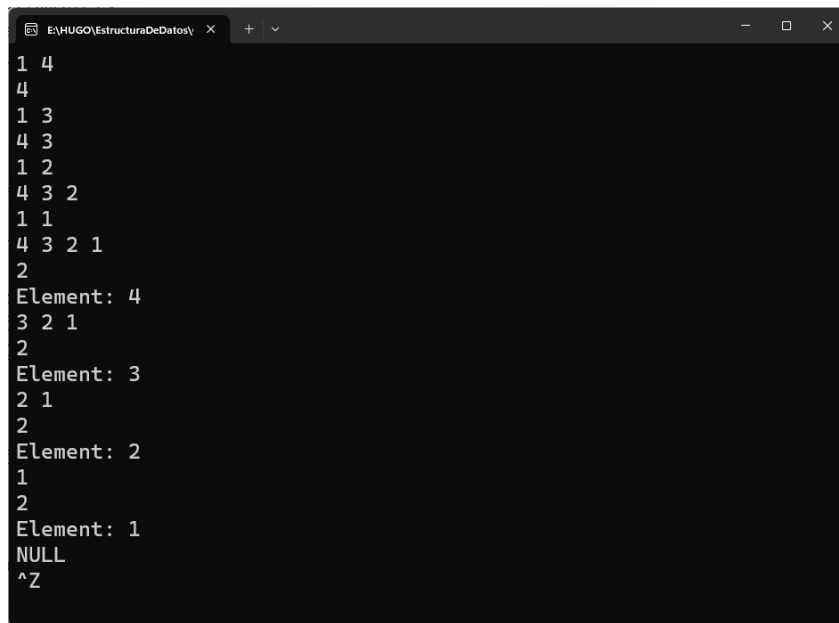
    tail = NULL;

    while(scanf("%d", &operation) != EOF)
    {
        if(operation == 1) /* Enqueue */
        {
            scanf("%d", &element);
            tail = enqueue(tail, element);
            printQueue(tail);
        }

        else
        {
            if(operation == 2) /* Dequeue */
            {
                if(queueEmpty(tail) != TRUE)
                {
                    element = dequeue(&tail);
                    printf("Element: %d\n", element);
                    printQueue(tail);
                }
                else
                {
                    printf("The queue is empty.\n");
                }
            }
            else
                printf("Bad use. \n 1. Enqueue\n 2. Dequeue\n");
        }
    }

    return 0;
}

```



```
E:\HUGO\EstructuraDeDatos' x + v - □ x
1 4
4
1 3
4 3
1 2
4 3 2
1 1
4 3 2 1
2
Element: 4
3 2 1
2
Element: 3
2 1
2
Element: 2
1
2
Element: 1
NULL
^Z
```

Figura 40: Salida del programa para el manejo de Colas.