PIXEL PRINT ESTUDIO

(Manual Técnico)

ARCHIVO MAIN:

1. **Declaración de Módulos y Variables:**

    - El programa utiliza varios módulos como **DynamicQueueModule**, **json_module**, **linked_list_uso**, **cola_clientes**, **lista_ventanillas**, **ColaDeImpresion_**, **ListaCircular**, y **pilaImg**.
    - Se declaran variables como **nombres**, **apellidos**, **choice**, **json**, **clientesEnCola**, **lista_vent**, entre otras.

```
program main
  !use DynamicQueueModule
  use json_module
  use linked_list_uso
  use cola_clientes
  use lista_ventanillas
  use ColaDeImpresion_
  use ListaCircular
  use pilaImg

  implicit none
  character(50), dimension(5) :: nombres = ["Pedro", "Maria", "Carlo", "Laura", "Pedro"]
  | character(50), dimension(5) :: apellidos = ["Gomez", "Lopez", "Avila", "Perez", "Veliz"]
  integer :: choice
  type(json_file) :: json   ! Se declara una variable del tipo json_file
  | type(json_value), pointer :: listPointer, personPointer, attributePointer
  | type(json_core) :: jsonc  ! Se declara una variable del tipo json_core para acceder a las funciones básicas de JSON
  | character(:), allocatable :: nombreCliente
  | integer :: imgPCliente, imgGCliente, contadorParaEliminarEnCola
  | integer :: i, size, contadorDePasos
  | | | | ! Se declaran variables enteras
  | logical :: found
  type(cola) :: clientesEnCola
  type(ListaVentanillas) :: lista_vent

  call inicializar_cola(clientesEnCola)
  contadorDePasos = 1
  contadorParaEliminarEnCola = 0
```

2. **Inicialización y Menú Principal:**

    - Se inicializan las estructuras de datos, como la cola de clientes (**clientesEnCola**) y la lista de ventanillas (**lista_vent**).
    - Se implementa un bucle **do** que presenta un menú al usuario, donde puede elegir entre varias opciones mediante la lectura de la variable **choice**.

```
call inicializar_cola(clientesEnCola)
contadorDePasos = 1
contadorParaEliminarEnCola = 0
do
    call printMenu()
    read(*,*) choice

    select case (choice)
        case (1)
            call option1()
        case (2)
            call option2()
        case (3)
            call option3()
        case (4)
            call option4()
        case (5)
            call option5()
        case (6)
            exit
        case default
            print *, "Opción no válida. Introduce un número del 1 al 4."
    end select
end do
```

3. **Subrutinas de Opciones del Menú:**
    - **printMenu() y printParametrosIniciales():** Subrutinas para imprimir el menú y los parámetros iniciales respectivamente.
    - **option1():** Subrutina que permite al usuario cargar clientes desde un archivo JSON o establecer la cantidad de ventanillas.
    - **option2():** Subrutina que simula la ejecución de un paso, generando aleatoriamente clientes y realizando operaciones como encolar y desencolar.
    - **option3():** Subrutina que imprime el estado en memoria de las estructuras, en este caso, grafica la pila de cada ventanilla.
    - **option4() y option5():** Subrutinas que están incompletas (**option4()** tiene un comentario sugiriendo añadir el código correspondiente).

```
subroutine printMenu()
    print *, "Menú:"
    print *, "1. Parametros iniciales"
    print *, "2. Ejecutar paso"
    print *, "3. Estado en memoria de las estructuras"
    print *, "4. Reportes"
    print *, "5. Acerca de"
    print *, "6. Salir"
    print *, "Elige una opción:"
end subroutine

subroutine printParametrosIniciales()
    print *, "a. Carga masiva de clientes"
    print *, "b. Cantidad de ventanillas"
end subroutine
```

```
subroutine option1()
  integer :: contador
  integer :: cantidad
  character(len=5) :: caracterId
    character(1) :: choice2
    character(100)::nombreArchivoJs
    print *, "Has seleccionado la Opción 1."
    call printParametrosIniciales()
    read(*, '(A)') choice2

    select case (choice2)
    case ("a")

        ! call json%initialize()
   !call json%load(filename="C:\Users\Javier Avila\Desktop\fortranjs\fortranjs\datos.json")
      print *, "Escribe la ruta del archivo .js"
      read(*, '(A)') nombreArchivoJs
      call json%initialize()    ! Se inicializa el módulo JSON
      call json%load(filename=nombreArchivoJs)  ! Se carga el archivo JSON llamado 'data.json'
          ! Se imprime el contenido del archivo JSON (opcional

      call json%info('',n_children=size)

      call json%get_core(jsonc)              ! Se obtiene el núcleo JSON para acceder a sus funciones básicas
      call json%get('', listPointer, found)
      print *, "-------------------------------------------"
      print *, "             Clientes Cargados"
      print *, "-------------------------------------------"

      do i = 1, size
        print *, "-------------------------------------------"
        print *, "               Cliente", i
        print *, "-------------------------------------------"
```

```
    call json%get_core(jsonc)              ! Se obtiene el núcleo JSON para acceder a sus funciones básicas
    call json%get('', listPointer, found)
    print *, "-----------------------------------------------"
    print *, "              Clientes Cargados"
    print *, "-----------------------------------------------"

    do i = 1, size
      print *, "-----------------------------------------------"
      print *, "                 Cliente", i
      print *, "-----------------------------------------------"
      ! Se inicia un bucle sobre el número de elementos en el JSON
      call jsonc%get_child(listPointer, i, personPointer, found = found)  ! Se obtiene el i-ésimo hijo de lis
      call jsonc%get_child(personPointer, 'nombre', attributePointer, found = found)  ! Se obtiene el valor a
      if (found) then                      ! Si se encuentra el valor asociado con la clave 'nombre'
        call jsonc%get(attributePointer, nombreCliente)  ! Se obtiene el valor y se asigna a la variable 'n
        print *, trim(nombreCliente)           ! Se imprime el nombre sin espacios en blanco adicionales
      end if

      call jsonc%get_child(personPointer, 'img_g', attributePointer, found = found)  ! Se obtiene el valor as
      if (found) then                      ! Si se encuentra el valor asociado con la clave 'nombre'
        call jsonc%get(attributePointer, imgGCliente)  ! Se obtiene el valor y se asigna a la variable 'nom
        print *, imgGCliente           ! Se imprime el nombre sin espacios en blanco adicionales
      end if

      call jsonc%get_child(personPointer, 'img_p', attributePointer, found = found)  ! Se obtiene el valor as
      if (found) then                      ! Si se encuentra el valor asociado con la clave 'nombre'
        call jsonc%get(attributePointer, imgPCliente)  ! Se obtiene el valor y se asigna a la variable 'nom
        print *, imgPCliente           ! Se imprime el nombre sin espacios en blanco adicionales
      end if

      call agregar_cliente(clientesEnCola, i, nombreCliente, imgGCliente, imgPCliente)
```

```
    end do
    print *, "-----------------------------------------------"
    print *, "          Termino Cargar Clientes"
    print *, "-----------------------------------------------"


  ! Liberar recursos
  call json%destroy()

  case ("b")

    print *, "Ingrese la cantidad de ventanillas que desea:"
    read(*,*) cantidad
    call agregar_ventanilla(lista_vent, cantidad)
    print *, "-----------------------------------------------"
    print *, "              Ventanillas Creadas"
    print *, "-----------------------------------------------"
    call recorrer_lista_ventanillas(lista_vent)
    print *, "-----------------------------------------------"
    print *, "-----------------------------------------------"

  case default
    print *, "Opción no válida. Introduce 'a' o 'b'."
end select
end subroutine
```

```
subroutine option2()
  integer :: iNuevo, num_clientes, iterador, VentanillaDesencolada
  logical :: colaVaciaVar
  character(len=100) :: nombreClienteDesencolado
  real :: random_value


  call random_seed()
  call random_number(random_value)

  ! Generar aleatoriamente la cantidad de clientes
  num_clientes = floor(random_value*4)

  do iterador = 1, num_clientes
      call random_number(random_value)

    ! Genera aleatoriamente el nombre del cliente
    nombreCliente = trim(nombres(mod(floor(random_value*5), 5) + 1))
    nombreCliente =  trim(nombreCliente //" "// apellidos(mod(floor(random_value*5), 5) + 1))


    ! Genera aleatoriamente la cantidad de imágenes por cliente (entre 0 y 4)
    call random_number(random_value)

    imgGCliente = floor(random_value * 5)
    call random_number(random_value)

    imgPCliente = floor(random_value * 5)
    call agregar_cliente(clientesEnCola, iterador, nombreCliente, imgGCliente, imgPCliente)
  end do

  call ColaVacia(ColaDeImpresionImgPequenas, colaVaciaVar)
  if (colaVaciaVar) then
    print *, "Se desencolo P--------------------------"
    call DesencolarImpresion(ColaDeImpresionImgPequenas, nombreClienteDesencolado, VentanillaDesencolada)
    call AgregarImagenAlCliente(ListaCircularDeEspera, nombreClienteDesencolado, &
    "ImgP", VentanillaDesencolada, contadorDePasos)
```

```
    call ColaVacia(ColaDeImpresionImgPequenas, colaVaciaVar)
    if (colaVaciaVar) then
      print *, "Se desencolo P--------------------------"
      call DesencolarImpresion(ColaDeImpresionImgPequenas, nombreClienteDesencolado, VentanillaDesencolada)
      call AgregarImagenAlCliente(ListaCircularDeEspera, nombreClienteDesencolado, &
       "ImgP", VentanillaDesencolada, contadorDePasos)


    end if
    call ColaVacia(ColaDeImpresionImgGrandes, colaVaciaVar)
    if (colaVaciaVar .and. contadorParaEliminarEnCola >= 1) then
      print *, "Se desencolo G--------------------------"
      call DesencolarImpresion(ColaDeImpresionImgGrandes, nombreClienteDesencolado, VentanillaDesencolada)
      call AgregarImagenAlCliente(ListaCircularDeEspera, nombreClienteDesencolado, &
       "ImgG", VentanillaDesencolada, contadorDePasos)
      contadorParaEliminarEnCola = 0
    else if (colaVaciaVar .and. contadorParaEliminarEnCola < 2) then
      contadorParaEliminarEnCola = contadorParaEliminarEnCola + 1

    end if

      print *, "-----------------------------------------"
      print *, "             PASO ", contadorDePasos
      print *, "-----------------------------------------"
      call pop_cliente(clientesEnCola, i, nombreCliente, imgGCliente, imgPCliente)
      call pasarClienteVentanilla(lista_vent,nombreCliente, imgPCliente, imgGCliente)

      print *, "-----------------------------------------"
      print *, "-----------------------------------------"


    !subroutine Desencolar(c, nombreCliente, ventanilla)
      contadorDePasos = contadorDePasos + 1



  end subroutine
```

```
  subroutine option3()
      print *, "Has seleccionado la Opción 3."
      call graficarPilaDeCadaVentanilla(lista_vent)
  end subroutine

  subroutine option4()
    print *, "Has seleccionado la Opción 3."
    ! Aquí puedes agregar el código correspondiente a la opción 3
end subroutine

subroutine option5()
  print *, "Has seleccionado la Opción 5."
  print *, "Nombre: Javier Alejandro Avila Flores"
  print *, "Curso: Laboratorio Estructuras de Datos B"
  print *, "Carnet: 202200392"

end subroutine
```

4. **Comentarios Adicionales:**

- Se utiliza la función **random_number** para generar valores aleatorios y simular ciertos comportamientos.
- Se manejan colas de impresión (**ColaDeImpresionImgPequenas** y **ColaDeImpresionImgGrandes**) y una lista circular (**ListaCircularDeEspera**).
- La simulación avanza en pasos (**contadorDePasos**) y realiza operaciones según la opción seleccionada por el usuario.

5. **Liberación de Recursos:**
    - Se llama a **json%destroy()** para liberar los recursos asociados con la manipulación de archivos JSON en la opción 1.
6. **Ciclo Principal:**
    - El programa permanece en un bucle hasta que el usuario elige la opción de salir (**6**).

ARCHVIO DE MODULOS:

Módulo pilaImg

- Implementa una pila (stack) de imágenes.
- Permite agregar imágenes a la pila.
- Permite eliminar imágenes de la pila.
- Proporciona una función para verificar si la pila está vacía.

```
module pilaImg
  implicit none

  ! Puedes ajustar el tamaño máximo de la palabra según tus necesidades
  integer, parameter :: max_longitud_palabra = 100

  ! Aquí puedes agregar todos los datos que quieras que lleve tu nodo, como un nombre u otra información
  type, public :: Nodo
    character(max_longitud_palabra) :: palabra
    character(max_longitud_palabra) :: nomClientePila

    type(Nodo), pointer :: siguiente
  end type Nodo

  type, public :: Pilas
    type(Nodo), pointer :: tope => null()
  end type Pilas

contains

  subroutine agregar(p, palabra, nomClientePila)
    type(Pilas), intent(inout) :: p
    character(len=*), intent(in) :: palabra
    character(len=*), intent(in) :: nomClientePila
    type(Nodo), pointer :: nuevo_nodo

    allocate(nuevo_nodo)
    nuevo_nodo%palabra = trim(palabra)
    nuevo_nodo%nomClientePila = trim(nomClientePila)
    nuevo_nodo%siguiente => p%tope
    p%tope => nuevo_nodo
  end subroutine agregar

  subroutine graficar(this, filename)
    class(Pilas), intent(in) :: this
    character(len=*) :: filename

    integer :: unit
    type(Nodo), pointer :: current
    integer :: count


    ! Abrir el archivo DOT
    open(unit, file=filename, status='replace')
    write(unit, *) 'digraph Pila {'
    write(unit, *) '    node [shape=box, style=filled, color=blue, fillcolor=pink];' ! Aplicar atributos a todos los nodos
    ! Escribir nodos y conexiones
    current => this%tope
    count = 0
    do while (associated(current))
        count = count + 1
        write(unit, *) '    "Node', count, '" [label="', current%palabra, '"];'
        if (associated(current%siguiente)) then
            write(unit, *) '    "Node', count, '" -> "Node', count+1, '";'
        end if
        current => current%siguiente
    end do
```

```
    ! Generar el archivo PNG utilizando Graphviz
    call system('dot -Tpng ' // trim(filename) // ' -o ' // trim(adjustl(filename)) // '.png')

    print *, 'Graphviz file generated: ', trim(adjustl(filename)) // '.png'
end subroutine graficar

  subroutine eliminar(p, nomClientePila)
    type(Pilas), intent(inout) :: p
    character(len=*), intent(out) :: nomClientePila
    type(Nodo), pointer :: nodo_aux

    if (associated(p%tope)) then
      nodo_aux => p%tope
      nomClientePila = trim(nodo_aux%nomClientePila)
      p%tope => nodo_aux%siguiente
      deallocate(nodo_aux)
    else
      ! Aquí podrías manejar el caso cuando la pila está vacía
      print *, 'La pila está vacía.'
    end if
  end subroutine eliminar


  function estaVacia(p) result(vacia)
    type(Pilas), intent(in) :: p
    logical :: vacia
    vacia = .not. associated(p%tope)
  end function estaVacia

end module pilaImg
```

Módulo lista_ventanillas

- Define un tipo de nodo (**NodoVentanilla**) para representar cada ventanilla.
- Crea una lista de ventanillas (**ListaVentanillas**) utilizando nodos de tipo **NodoVentanilla**.
- Incluye funciones para agregar ventanillas a la lista, pasar clientes a las ventanillas, atender clientes y recorrer la lista de ventanillas.
- Implementa la función **graficarPilaDeCadaVentanilla** para generar gráficos de las pilas de imágenes en cada ventanilla.

```
module lista_ventanillas
  use ColaDeImpresion_
  use pilaImg

  use ListaCircular

  implicit none

  type, public :: NodoVentanilla
    integer :: id
    logical :: enUsoPorCliente
    type(Pilas) :: pilaImagenes
    character(len = 100) :: nomClienteVentanilla
    integer :: contG, contP, imgPPasarCola, imgGPasarCola

    type(NodoVentanilla), pointer :: siguiente => null()
  end type NodoVentanilla

  type, public :: ListaVentanillas
  type(NodoVentanilla), pointer :: inicio => null()

  end type ListaVentanillas
  type(ListaCircularType) :: ListaCircularDeEspera
  type(ColaImpresion) :: ColaDeImpresionImgGrandes
  type(ColaImpresion) :: ColaDeImpresionImgPequenas


  contains
```

```
contains


subroutine pasarClienteVentanilla(lista, nombreClienteAtender, imgPAtender, imgGAtender)
  class(ListaVentanillas), intent(inout) :: lista
  character(*), intent(in) :: nombreClienteAtender
  integer, intent(in) :: imgPAtender, imgGAtender
  type(NodoVentanilla), pointer :: actual
  integer :: sumarImgEnVent

  if (associated(lista%inicio)) then
    actual => lista%inicio
    do while (associated(actual))
      if (.not. actual%enUsoPorCliente) then
        ! La ventanilla está disponible
        actual%contP = imgPAtender
        actual%contG = imgGAtender
        actual%imgPPasarCola = imgPAtender
        actual%imgGPasarCola = imgGAtender

        actual%nomClienteVentanilla = nombreClienteAtender
        actual%enUsoPorCliente = .true.
        print *, "Cliente atendido en la ventanilla ", actual%id
        print *, "Nombre del cliente: ", nombreClienteAtender
        print *, "Imagen P a atender: ", imgPAtender
        print *, "Imagen G a atender: ", imgGAtender

        exit
      else if (actual%enUsoPorCliente) then

        if (actual%contG >0) then
          print *,"agrego a la pila la imagen de: " // actual%nomClienteVentanilla
          call agregar(actual%pilaImagenes, "imgG", actual%nomClienteVentanilla)
          actual%contG = actual%contG - 1

        else if (actual%contP >0) then
          print *,"agrego a la pila la imagen de: " // actual%nomClienteVentanilla
          call agregar(actual%pilaImagenes, "imgP", actual%nomClienteVentanilla)
          actual%contP = actual%contP - 1
        else
          sumarImgEnVent = actual%imgGPasarCola + actual%imgPPasarCola

          actual%enUsoPorCliente = .false.
          call atenderClientes(actual, sumarImgEnVent)

        end if
      end if
      actual => actual%siguiente
    end do


  else
    print *, "La lista está vacía."
  end if
end subroutine pasarClienteVentanilla
```

```
subroutine agregar_ventanilla(lista, cantidad)
    class(ListaVentanillas), intent(inout) :: lista
    type(NodoVentanilla), pointer :: nuevoNodo, actual
    integer, intent(in) :: cantidad
    integer :: i

    do i = 1, cantidad
        allocate(nuevoNodo)
        nuevoNodo%id = i
        nuevoNodo%enUsoPorCliente = .false.
        nuevoNodo%siguiente => null()

        if (associated(lista%inicio)) then
            actual => lista%inicio
            do while(associated(actual%siguiente))
                actual => actual%siguiente
            end do
            actual%siguiente => nuevoNodo
        else
            lista%inicio => nuevoNodo
        end if
    end do
end subroutine agregar_ventanilla

subroutine atenderClientes(actual, imgSumar)
  integer, intent(in) :: imgSumar
  type(NodoVentanilla), pointer :: actual
  character(max_longitud_palabra) :: palabraEliminada
  integer :: con

  do con = 1, imgSumar
    call eliminar(actual%pilaImagenes,palabraEliminada)
  end do
  write(*,*)"Se vacio la pila con las img de:  "// trim(palabraEliminada)

  call AgregarNodoCircular(ListaCircularDeEspera ,trim(palabraEliminada), actual%imgGPasarCola, actual%imgPPasarCola , actual%id)


  do con = 1, actual%imgGPasarCola
    call Encolar(ColaDeImpresionImgGrandes, actual%nomClienteVentanilla, actual%id)
  end do

  do con = 1, actual%imgPPasarCola
    call Encolar(ColaDeImpresionImgPequenas, actual%nomClienteVentanilla, actual%id)
  end do
  print*, "COLA IMPRESORA GRANDE_____________________________________"
  call ImprimirCola(ColaDeImpresionImgGrandes)
  print*, "COLA IMPRESORA PEQUENA_____________________________________"
  call ImprimirCola(ColaDeImpresionImgPequenas)

end subroutine atenderClientes
```

```
subroutine recorrer_lista_ventanillas(lista)
    class(ListaVentanillas), intent(in) :: lista
    type(NodoVentanilla), pointer :: actual

    if (associated(lista%inicio)) then
        actual => lista%inicio
        do while (associated(actual))
            print *, "Vantanilla ", actual%id
            actual => actual%siguiente
        end do
    else
        print *, "La lista está vacía."
    end if
end subroutine recorrer_lista_ventanillas
```

```
subroutine graficarPilaDeCadaVentanilla(lista)
type(ListaVentanillas), intent(in) :: lista
    type(NodoVentanilla), pointer :: actual

    character (50):: nombreArchivo
    if (associated(lista%inicio)) then
        actual => lista%inicio

        do while (associated(actual))
            write (nombreArchivo, '(A,I0)') "Pila_", actual%id
            call graficar(actual%pilaImagenes, nombreArchivo)

            actual => actual%siguiente
        end do
    else
        print *, "La lista está vacía."
    end if
end subroutine graficarPilaDeCadaVentanilla
end module lista_ventanillas
```

Módulo cola_clientes

- Define tipos para representar clientes y nodos de una cola de clientes.
- Implementa funciones para inicializar la cola, agregar clientes y extraer clientes de la cola.

```
module cola_clientes
  implicit none

  private
  type, public :: cliente
    integer :: id_cliente
    character(len=100) :: nombre_cliente
    integer :: imagen_g
    integer :: imagen_p
  end type cliente

  type, public :: nodo
    type(cliente) :: datos
    type(nodo), pointer :: siguiente
  end type nodo

  type, public :: cola
    type(nodo), pointer :: frente
    type(nodo), pointer :: final
  end type cola

  public :: inicializar_cola, agregar_cliente, pop_cliente
```

```
contains

  subroutine inicializar_cola(c)
    type(cola), intent(out) :: c
    c%frente => null()
    c%final => null()
  end subroutine inicializar_cola

  subroutine agregar_cliente(c, id_cliente, nombre_cliente, imagen_g, imagen_p)
    type(cola), intent(inout) :: c
    integer, intent(in) :: id_cliente, imagen_g, imagen_p
    character(len=*), intent(in) :: nombre_cliente
    type(nodo), pointer :: nuevo_nodo

    allocate(nuevo_nodo)
    nuevo_nodo%datos%id_cliente = id_cliente
    nuevo_nodo%datos%nombre_cliente = nombre_cliente
    nuevo_nodo%datos%imagen_g = imagen_g
    nuevo_nodo%datos%imagen_p = imagen_p
    nuevo_nodo%siguiente => null()

    if (associated(c%final)) then
      c%final%siguiente => nuevo_nodo
    else
      c%frente => nuevo_nodo
    end if

    c%final => nuevo_nodo
  end subroutine agregar_cliente


  subroutine pop_cliente(c, id, nombre, imagen_g, imagen_p)
    type(cola), intent(inout) :: c
    integer, intent(out) :: id, imagen_g, imagen_p
    character(len=*), intent(out) :: nombre
    type(nodo), pointer :: nodo_aux

    if (.not. associated(c%frente)) then

      return
    end if

    id = c%frente%datos%id_cliente
    nombre = c%frente%datos%nombre_cliente
    imagen_g = c%frente%datos%imagen_g
    imagen_p = c%frente%datos%imagen_p

    nodo_aux => c%frente
    c%frente => c%frente%siguiente
    deallocate(nodo_aux)
  end subroutine pop_cliente

end module cola_clientes
```

Módulo ColaDeImpresion_

- Implementa una cola de impresión utilizando nodos.
- Proporciona funciones para encolar, desencolar e imprimir la cola de impresión.

```
module ColaDeImpresion_
  implicit none

  type, public :: Nodo
    character(50) :: nombreClienteImprimiendo
    integer :: idVentanillaAtendioImprimiendo
    type(Nodo), pointer :: siguiente
  end type Nodo

  type, public :: ColaImpresion
    type(Nodo), pointer :: frente => null()
    type(Nodo), pointer :: final => null()
  end type ColaImpresion

  contains


  subroutine Encolar(c, nombreCliente, idVentanillaAtendioImprimiendo)
    type(ColaImpresion), intent(inout) :: c
    character(len = *), intent(in) :: nombreCliente
    integer, intent(in) :: idVentanillaAtendioImprimiendo


    type(Nodo), pointer :: nuevoNodo
    allocate(nuevoNodo)
    nuevoNodo%nombreClienteImprimiendo = nombreCliente
    nuevoNodo%idVentanillaAtendioImprimiendo = idVentanillaAtendioImprimiendo

    nuevoNodo%siguiente => null()

    if (.not. associated(c%frente)) then
      c%frente => nuevoNodo
      c%final => nuevoNodo
    else
      c%final%siguiente => nuevoNodo
      c%final => nuevoNodo
    end if
  end subroutine Encolar
```

```
subroutine DesencolarImpresion(c, nombreCliente, idVentanillaAtendioImprimiendo)
    type(ColaImpresion), intent(inout) :: c
    character(len = *), intent(out) :: nombreCliente
    integer, intent(out) :: idVentanillaAtendioImprimiendo


    type(Nodo), pointer :: nodoDesencolado

    if (.not. associated(c%frente)) then
        print *, 'La cola está vacía.'
        return
    end if

    nodoDesencolado => c%frente
    c%frente => nodoDesencolado%siguiente

    nombreCliente = nodoDesencolado%nombreClienteImprimiendo
    idVentanillaAtendioImprimiendo = nodoDesencolado%idVentanillaAtendioImprimiendo


    deallocate(nodoDesencolado)
end subroutine DesencolarImpresion

subroutine ImprimirCola(c)
  type(ColaImpresion), intent(in) :: c

  type(Nodo), pointer :: nodoActual

  if (.not. associated(c%frente)) then

      return
  end if

  print *, 'Elementos en la cola de impresión:'
  nodoActual => c%frente
  do while (associated(nodoActual))
      print *, 'Cliente: ', nodoActual%nombreClienteImprimiendo
      nodoActual => nodoActual%siguiente
  end do
nd subroutine ImprimirCola

ubroutine ColaVacia(c, vacia)
 type(ColaImpresion), intent(in) :: c
 logical, intent(inout) :: vacia

 vacia =  associated(c%final)
nd subroutine ColaVacia

nd module ColaDeImpresion_
```

Módulo ListaClientesMod

- Define un tipo de nodo para representar clientes atendidos.
- Implementa funciones para agregar clientes a la lista y para imprimir la lista de clientes atendidos.

```fortran
module ListaClientesMod
implicit none
  type, public :: Nodo
    character(50) :: nombreClienteAtendido
    integer :: CantImgImpresas
    integer :: VentanillaAtendida
    integer :: CantidadTotalPasos
    type(Nodo), pointer :: siguiente => null()
  end type Nodo

  type, public :: ListaClientesAtendidos
    type(Nodo), pointer :: cabeza => null()
  end type ListaClientesAtendidos

contains

  subroutine agregarCliente(lista, nombre, cantImpresas, ventanilla, cantPasos)
    ! Agrega un nuevo cliente a la lista
    type(ListaClientesAtendidos), intent(inout) :: lista
    character(len = *), intent(in) :: nombre
    integer, intent(in) :: cantImpresas, ventanilla, cantPasos


    type(Nodo), pointer :: nuevoNodo
    allocate(nuevoNodo)
    nuevoNodo%nombreClienteAtendido = nombre
    nuevoNodo%CantImgImpresas = cantImpresas
    nuevoNodo%VentanillaAtendida = ventanilla
    nuevoNodo%CantidadTotalPasos = cantPasos

    nuevoNodo%siguiente => lista%cabeza
    lista%cabeza => nuevoNodo

  end subroutine agregarCliente

  subroutine imprimirListaClientesAtendidos(lista)
    ! Imprime la lista de clientes
    type(ListaClientesAtendidos), intent(in) :: lista
    type(Nodo), pointer :: actual

    actual => lista%cabeza
    do while (associated(actual))
      print *, "Cliente: ", actual%nombreClienteAtendido
      print *, "Cantidad de Imágenes Impresas: ", actual%CantImgImpresas
      print *, "Ventanilla Atendida: ", actual%VentanillaAtendida
      print *, "Cantidad Total de Pasos: ", actual%CantidadTotalPasos
      print *, "--------------------------------------"
      actual => actual%siguiente
    end do
  end subroutine imprimirListaClientesAtendidos

end module ListaClientesMod
```

Módulo listaImagenesImpresas

- Implementa una lista de imágenes impresas utilizando nodos.
- Proporciona funciones para agregar imágenes a la lista e imprimir la lista de imágenes impresas.

```
module listaImagenesImpresas
  implicit none

  ! Definición del tipo de nodo
  type, public :: nodoImp
    character(10) :: imgTipo
    integer :: imgId
    type(nodoImp), pointer :: siguiente
  end type nodoImp



  ! Definición del tipo de lista
  type, public :: listaImgImp
    type(nodoImp), pointer :: cabeza => null()
  end type listaImgImp



contains

  ! Subrutina para agregar un nodo a la lista
  subroutine agregarNodoImpresion(L, tipo, id)
    type(listaImgImp), intent(inout) :: L
    character(len = *), intent(in) :: tipo
    integer, intent(in) :: id

    type(nodoImp), pointer :: nuevoNodo

    ! Crear un nuevo nodo y asignar los valores
    allocate(nuevoNodo)
    nuevoNodo%imgTipo = tipo
    nuevoNodo%imgId = id

    ! Enlazar el nuevo nodo al inicio de la lista
    nuevoNodo%siguiente => L%cabeza
    L%cabeza => nuevoNodo
  end subroutine agregarNodoImpresion

  ! Subrutina para imprimir la lista
  subroutine imprimirLista(L)
    type(listaImgImp), intent(in) :: L
    type(nodoImp), pointer :: actual

    if (.not. associated(L%cabeza)) then
      write(*, '(A)') 'La lista de imágenes impresas está vacía.'
      return
    end if

    actual => L%cabeza
    do while (associated(actual))
      print *, 'Tipo:', actual%imgTipo, ', ID:', actual%imgId
      actual => actual%siguiente
    end do
  end subroutine imprimirLista

end module listaImagenesImpresas
```

Módulo ListaCircular

- Implementa una lista circular de clientes en espera.
- Define tipos para nodos de la lista circular y la lista de clientes ya atendidos.
- Incluye funciones para agregar nodos a la lista circular, agregar imágenes a clientes en espera, eliminar nodos y clientes atendidos.

```
module ListaCircular
  use listaImagenesImpresas
  use ListaClientesMod
  implicit none

  type, public :: NodoCircular
    character(50) :: nombreClienteEspera
    integer :: cantImgEspera, contadorDeImsgenesRecibidas
    integer ::  idVentanillaAtendio
    type(listaImgImp) :: lista_imagenes_impresas
    type(NodoCircular), pointer :: siguiente
    type(NodoCircular), pointer :: anterior
  end type NodoCircular

  type, public :: ListaCircularType
    type(NodoCircular), pointer :: cabeza => null()
    type(NodoCircular), pointer :: ultimo => null()
  end type ListaCircularType

  type(ListaClientesAtendidos) :: ListaDeClientesQueYaFueronAtendidos

  contains

  subroutine AgregarNodoCircular(lista, nombreCliente, cantImgG, cantImgP, idVentanillaAtendio)
    type(ListaCircularType), intent(inout) :: lista
    type(NodoCircular), pointer :: nuevoNodo
    character(len = *), intent(in) :: nombreCliente
    integer, intent(in) :: cantImgG, cantImgP, idVentanillaAtendio


    allocate(nuevoNodo)
    nuevoNodo%nombreClienteEspera = nombreCliente
    nuevoNodo%cantImgEspera = cantImgP + cantImgG
    nuevoNodo%idVentanillaAtendio = idVentanillaAtendio
    nuevoNodo%contadorDeImsgenesRecibidas = 0

    if (associated(lista%cabeza)) then
      nuevoNodo%siguiente => lista%cabeza
      nuevoNodo%anterior => lista%ultimo
      lista%ultimo%siguiente => nuevoNodo
      lista%cabeza%anterior => nuevoNodo
    else
      nuevoNodo%siguiente => nuevoNodo
      nuevoNodo%anterior => nuevoNodo
    end if

    lista%cabeza => nuevoNodo
    lista%ultimo => nuevoNodo
  end subroutine AgregarNodoCircular
```

```
subroutine AgregarImagenAlCliente(lista, nombreCliente, nuevaImagen, idVentanillaAtendioPasar, contadorDePaso
  type(ListaCircularType), intent(inout) :: lista
  character(len = *), intent(in) :: nombreCliente
  character(len = *), intent(in) :: nuevaImagen
  integer, intent(in) :: idVentanillaAtendioPasar
  type(NodoCircular), pointer :: nodoActual
  logical :: clienteEncontrado
  integer, intent(in) :: contadorDePasos

  clienteEncontrado = .false.

  if (associated(lista%cabeza)) then
    nodoActual => lista%cabeza
    do

      if (trim(nodoActual%nombreClienteEspera) == trim(nombreCliente) &
      .and. nodoActual%idVentanillaAtendio == idVentanillaAtendioPasar) then
        ! Agregar la nueva imagen a la lista del cliente encontrado
        if (nodoActual%contadorDeImsgenesRecibidas < nodoActual%cantImgEspera)  then

          nodoActual%contadorDeImsgenesRecibidas = nodoActual%contadorDeImsgenesRecibidas +1
          print *, nuevaImagen // "FUE ENTREGADA"
          call agregarNodoImpresion(nodoActual%lista_imagenes_impresas, nuevaImagen &
          , nodoActual%contadorDeImsgenesRecibidas)

          print *, "*************Imagen entregada*************"

          if(nodoActual%contadorDeImsgenesRecibidas >= nodoActual%cantImgEspera) then

            call EliminarNodoPorNombreYVentanilla(lista, nombreCliente, idVentanillaAtendioPasar, &
            contadorDePasos , nodoActual%cantImgEspera)
            write(*, '(A, I5, I5)') trim(nombreCliente) // " Salio"
          end if
        else
          exit

        end if

        clienteEncontrado = .true.
        exit
      end if

      if (trim(nodoActual%nombreClienteEspera) == trim(lista%ultimo%nombreClienteEspera) &
      .and. nodoActual%idVentanillaAtendio == lista%ultimo%idVentanillaAtendio ) exit
      nodoActual => nodoActual%siguiente
    end do
  end if

  if (.not. clienteEncontrado) then
    write(*, '(A)') 'Cliente no encontrado en la lista.'
  end if
end subroutine AgregarImagenAlCliente
```

```
subroutine EliminarNodoPorNombreYVentanilla(lista, nombreCliente, numeroDeVentanilla, contadorDePasos, cantImgEspera)
  type(ListaCircularType), intent(inout) :: lista
  character(50), intent(in) :: nombreCliente
  integer, intent(in) :: numeroDeVentanilla
  integer, intent(in) :: contadorDePasos, cantImgEspera
  type(NodoCircular), pointer :: nodoActual, nodoEliminar

  logical :: nodoEncontrado
  nodoEncontrado = .false.



  if (associated(lista%cabeza)) then
    nodoActual => lista%cabeza
    do
      if (trim(nodoActual%nombreClienteEspera) == trim(nombreCliente) .and. &
       nodoActual%idVentanillaAtendio == numeroDeVentanilla) then
        ! Remove the node from the list
        if (trim(nodoActual%nombreClienteEspera) == trim(nodoActual%siguiente%nombreClienteEspera) .and. &
        nodoActual%idVentanillaAtendio == nodoActual%siguiente%idVentanillaAtendio) then
          ! Only one node in the list
          nullify(lista%cabeza)
          nullify(lista%ultimo)
        else
          if (trim(nodoActual%nombreClienteEspera) == trim (lista%cabeza%nombreClienteEspera) .and. &
           nodoActual%idVentanillaAtendio == lista%cabeza%idVentanillaAtendio) then
            ! If the node to be removed is the head, update the head
            lista%cabeza => nodoActual%siguiente
          end if
          if (trim(nodoActual%nombreClienteEspera) == trim(lista%ultimo%nombreClienteEspera) .and. &
          nodoActual%idVentanillaAtendio == lista%ultimo%idVentanillaAtendio) then
            ! If the node to be removed is the last, update the last
            lista%ultimo => nodoActual%anterior
          end if
          ! Update the pointers of adjacent nodes
          nodoActual%anterior%siguiente => nodoActual%siguiente
          nodoActual%siguiente%anterior => nodoActual%anterior
        end if

        ! Save the node to be deleted for printing later
        nodoEliminar => nodoActual

        ! Deallocate the node
        deallocate(nodoActual)

        nodoEncontrado = .true.
        exit
      end if

      if (trim(nodoActual%nombreClienteEspera) == trim(lista%ultimo%nombreClienteEspera) .and. &
       nodoActual%idVentanillaAtendio == lista%ultimo%idVentanillaAtendio) exit
      nodoActual => nodoActual%siguiente
    end do
  end if

  if (.not. nodoEncontrado) then
```

```
    end if

    if (.not. nodoEncontrado) then
      write(*, '(A)') 'Nodo no encontrado en la lista.'
    else
      write(*, '(A)') 'Nodo eliminado:'
      write(*, '(A, IS, IS)') trim(nombreCliente)
      write(*, '(A)') 'Imágenes Impresas:'
      call imprimirLista(nodoEliminar%lista_imagenes_impresas)
      call agregarCliente(ListaDeClientesQueYaFueronAtendidos, nombreCliente, &
      cantImgEspera, numeroDeVentanilla, contadorDePasos)
      call imprimirListaClientesAtendidos(ListaDeClientesQueYaFueronAtendidos)

    end if
  end subroutine EliminarNodoPorNombreYVentanilla


end module ListaCircular
```