Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Escuela de Ciencias y Sistemas

Organización de Lenguajes y Compiladores 1

Primer Semestre 2024

Ing. Luis Espino

Kevin Martinez

Proyecto1 Fase 2

Pixel Print Studio

Manual Tecnico

César Fernando Sazo Quisquinay

202202906

Objetivo Principal:

Creación de un sistema de creación de imágenes a partir de estructuras de datos no lineales, como son los árboles o matrices dispersas, accediendo a distintas funciones de estos como poder visualizarlos gráficamente, poder iterar sobre ellos y acceder o modificar información de cada uno como parte del procedimiento de todas las funcionalidades del programa.

Funcionamiento General

La idea general del programa se baso en la anidación de estructuras, es decir que dentro de un nodo de algún árbol puede contener un árbol u otro tipo de estructura. Como es el caso de árbol de capas que dentro de cada nodo “Capa” contiene una matriz dispersa que genera la grafica de la imagen como tal, entonces la lógica aplicada seria: Un cliente puede contener uno o varios álbumes, cada álbum puede contener una o varias imágenes, cada imagen esta formada por un árbol de capas que a su vez una capa esta formada por una matriz dispersa, en esta lógica se basó el funcionamiento del programa, declarando distintos tipos de variables en cada nodo de distintos árboles como fue necesario.

Funcionamiento de los elementos

Clientes:

Los clientes son almacenados en un árbol B, ordenados por medio de su dpi, cada cliente es almacenado como un objeto “Cliente”, que como atributos contiene cada estructura que puede contener un cliente como el árbol de imágenes o capas, a su vez identificadores como el nombre, etc.

```
type:: Cliente
    integer :: id
    integer(8) :: dpi
    character(len=20) :: nombre, password
    type(lista_album) :: miListaAlbum
    type(ArbolImagenes) :: miArbolImg
    type(ArbolCapas) :: miArbolCapas
end type Cliente
```

La declaración del árbol se hizo por medio de arreglos para no complicar ni retrasar el proceso de creación del árbol, haciendo representación de que cada nodo del árbol es un arreglo de datos limitado por el orden que sea el árbol, en esta caso es un árbol B de grado 5.

```
type BTreeNode
    type(Cliente) :: cliente(0:MAXI+1)
    integer :: num = 0
    type(nodeptr) :: link(0:MAXI+1)
end type BTreeNode
```

El proceso de agregar un nodo al árbol B, es decir agregar un cliente, comienza con la llamada a la subrutina insert, que asigna un identificador único al cliente y verifica si necesita crear un nuevo nodo raíz a través de la función setValue. Esta función recursiva busca el lugar correcto en el árbol para insertar el nuevo valor. Si el nodo actual está vacío (es decir, no está asociado), el nuevo valor se convierte en el valor a insertar y el proceso indica que se debe crear un nuevo nodo. Si el nodo no está vacío, setValue encuentra la posición correcta comparando el DPI del cliente con los DPIs en el nodo actual. Si el nodo tiene espacio (menos de MAXI clientes), se inserta el cliente directamente en el nodo en la posición adecuada mediante insertNode. Si el nodo está lleno, se divide usando splitNode, creando un nuevo nodo y redistribuyendo los valores entre el nodo original y el nuevo nodo. splitNode también devuelve un valor al nodo padre (o crea uno nuevo si es necesario), asegurando que el árbol mantenga sus propiedades balanceadas. Este proceso recursivo asegura que el árbol se mantenga equilibrado, con todos los nodos excepto la raíz teniendo entre MINI y MAXI valores, y que los valores estén ordenados para permitir búsquedas eficientes.

```
subroutine insert(this, val)
    class(ArbolClientes), intent(inout) :: this
    type(Cliente), intent(inout) :: val
    type(Cliente):: i
    type(BTreeNode), pointer :: child
    allocate(child)
    val%id = this%contador
    this%contador =  this%contador + 1
    if (this%setValue(val, i, this%root, child)) then
            this%root => this%createNode(i, child)
    end if
end subroutine insert
```

```
function createNode(this,val, child) result(newNode)
    class(ArbolClientes), intent(inout) :: this
    type(Cliente), intent(in) :: val
    type(BTreeNode), pointer, intent(in) :: child
    type(BTreeNode), pointer :: newNode
    integer :: i
    allocate(newNode)
    newNode%cliente(1) = val
    newNode%num = 1
    newNode%link(0)%ptr => this%root
    newNode%link(1)%ptr => child
    do i = 2, MAXI
            newNode%link(i)%ptr => null()
    end do
end function createNode
```

```
subroutine splitNode(this,val, pval, pos, node, child, newnode)
    class(ArbolClientes), intent(inout) :: this
    type(Cliente), intent(in) :: val
    integer, intent(in) :: pos
    type(Cliente), intent(inout) :: pval
    type(BTreeNode), pointer, intent(inout) :: node,  newnode
    type(BTreeNode), pointer, intent(in) ::  child
    integer :: median, i, j
    if (pos > MINI) then
            median = MINI + 1
    else
            median = MINI
    end if
    if (.not. associated(newnode)) then
        allocate(newnode)
    do i = 0, MAXI
            newnode%link(i)%ptr => null()
        enddo
    end if
    j = median + 1
    do while (j <= MAXI)
            newnode%cliente(j - median) = node%cliente(j)
            newnode%link(j - median)%ptr => node%link(j)%ptr
            j = j + 1
    end do
    node%num = median
    newnode%num = MAXI - median
    if (pos <= MINI) then
            call this%insertNode(val, pos, node, child)
    else
            call this%insertNode(val, pos - median, newnode, child)
    end if
    pval = node%cliente(node%num)
    newnode%link(0)%ptr => node%link(node%num)%ptr
    node%num = node%num - 1
end subroutine splitNode
```

```
subroutine insertNode(this,val, pos, node, child)
    class(ArbolClientes), intent(inout) :: this
    type(Cliente), intent(in) :: val
    integer, intent(in) :: pos
    type(BTreeNode), pointer, intent(inout) :: node
    type(BTreeNode), pointer, intent(in) :: child
    integer :: j
    j = node%num
    do while (j > pos)
            node%cliente(j + 1) = node%cliente(j)
            node%link(j + 1)%ptr => node%link(j)%ptr
            j = j - 1
    end do
    node%cliente(j + 1) = val
    node%link(j + 1)%ptr => child
    node%num = node%num + 1
end subroutine insertNode
```

Álbumes:

Esta es una estructura de datos compuesta por una lista doblemente enlazada de álbumes, donde cada nodo de álbum contiene a su vez una lista enlazada de imágenes. La lista de álbumes permite operaciones básicas como agregar un álbum, agregar o eliminar una imagen en un álbum específico, mostrar los álbumes e imágenes, generar una visualización gráfica de la estructura y contar el número de álbumes e imágenes. Cada álbum tiene un identificador único y mantiene una referencia tanto al álbum anterior como al siguiente, permitiendo navegación bidireccional. Dentro de cada álbum, las imágenes se gestionan mediante una lista enlazada simple, con cada nodo de imagen apuntando a la siguiente imagen en el álbum. Esta organización facilita la adición y eliminación de imágenes manteniendo los álbumes separados y ordenados. La función graficar_albums genera una representación gráfica de la lista de álbumes y sus imágenes usando Graphviz, destacando la estructura anidada y las conexiones entre los álbumes y las imágenes contenidas dentro de ellos.

```
subroutine agregarAlbum(lista, id)
    class(lista_album), intent(inout) :: lista
    character(len=10), intent(in) :: id
    type(NodoAlbum), pointer :: nuevoAlbum, actual

    allocate(nuevoAlbum)
    nuevoAlbum%id = id
    nuevoAlbum%anterior => null()
    nuevoAlbum%siguiente => null()

    if (.not. associated(lista%head)) then
        lista%head => nuevoAlbum
    else
        actual => lista%head
        do while(associated(actual%siguiente))
            actual => actual%siguiente
        end do
            actual%siguiente => nuevoAlbum
            nuevoAlbum%anterior => actual
    end if
end subroutine agregarAlbum
```

```
subroutine addImagen(lista,id,img)
    class(lista_album), intent(inout) :: lista
    character(len=10), intent(in) :: id
    integer, intent(in) :: img
    type(NodoAlbum), pointer :: actual

    actual => lista%head
    do while(associated(actual))
        if (actual%id == id) then
            call actual%listImagenes%agregarImagen(img)
        end if
        actual => actual%siguiente
    end do
end subroutine addImagen

subroutine deleteImagen(lista,img)
    class(lista_album), intent(inout) :: lista
    integer, intent(in) :: img
    type(NodoAlbum), pointer :: actual
    actual => lista%head
    do while(associated(actual))
        call actual%listImagenes%eliminarImagen(img)
        actual => actual%siguiente
    end do
end subroutine
```

```
subroutine agregarImagen(lista, id)
    class(lista_imagen), intent(inout) :: lista
    integer, intent(in) :: id
    type(NodoLimagen), pointer :: nuevoNodo, actual

    allocate(nuevoNodo)
    nuevoNodo%id = id
    nuevoNodo%siguiente => null()

    if (.not. associated(lista%head)) then
        lista%head => nuevoNodo
    else
        actual => lista%head
        do while(associated(actual%siguiente))
            actual => actual%siguiente
        end do
            actual%siguiente => nuevoNodo
    end if

end subroutine agregarImagen
```

Imágenes:

Esta estructura está definida como un árbol AVL, que es una estructura de datos “autoequilibrada”. Al insertar un nuevo nodo, se verifica primero si la raíz está establecida; si no lo está, el nuevo nodo se convierte en la raíz. Si ya existe una raíz, se inserta el nodo de manera recursiva, manteniendo el árbol equilibrado. Durante la inserción, se compara el ID del nuevo nodo con los IDs de los nodos existentes para encontrar la ubicación correcta, siguiendo las propiedades de búsqueda binaria (nodos menores a la izquierda y mayores a la derecha). Después de insertar el nodo, se verifica y corrige el balance del árbol mediante rotaciones simples o dobles (derecha o izquierda) para asegurar que la altura de los subárboles difiera en no más de uno, lo cual es característico de los árboles AVL.

Además de la inserción, el módulo proporciona rutinas para obtener la altura de un nodo, realizar rotaciones (simples y dobles) para mantener el árbol balanceado, contar el número total de imágenes, y extraer el top 5 de imágenes con más capas, evidenciando así la multifuncionalidad del árbol para diferentes operaciones sobre los datos que contiene. Las rotaciones son esenciales para mantener el equilibrio del árbol después de inserciones o eliminaciones, asegurando que las operaciones de búsqueda, inserción y eliminación se puedan realizar en tiempo logarítmico. La capacidad de graficar el árbol y sus capas, junto con la inserción de capas en imágenes específicas, subraya la utilidad del árbol AVL en la gestión eficiente y organizada de datos complejos, como lo serían las imágenes y sus capas en un contexto de edición o almacenamiento de imágenes.

Rotaciones:

```
! Rotacion simple por la izquierda
function rsi(this, t1) result(t2)
    class(ArbolImagenes), intent(in) :: this
    type(NodoImagen), intent(in), pointer :: t1
    type(NodoImagen), pointer :: t2
    t2 => t1%left
    t1%left => t2%right
    t2%right => t1
    t1%altura = this%getMax(this%getAltura(t1%left), this%getAltura(t1%right))+1
    t2%altura = this%getMax(this%getAltura(t2%left), t1%altura)+1
end function rsi

! Rotacion simple por la derecha
function rsd(this, t1) result(t2)
    class(ArbolImagenes), intent(in) :: this
    type(NodoImagen), intent(in), pointer :: t1
    type(NodoImagen), pointer :: t2
    t2 => t1%right
    t1%right => t2%left
    t2%left => t1
    t1%altura = this%getmax(this%getAltura(t1%left), this%getAltura(t1%right))+1
    t2%altura = this%getmax(this%getAltura(t2%right), t1%altura)+1
end function rsd

! Rotacion doble por la izquierda
function rdi(this, tmp) result(res)
    class(ArbolImagenes), intent(in) :: this
    type(NodoImagen), intent(in), pointer :: tmp
    type(NodoImagen), pointer :: res
    tmp%left => this%rsd(tmp%left)
    res => this%rsi(tmp)
end function rdi
```

```
recursive subroutine add_recursivo(this, id, temp)
    class(ArbolImagenes), intent(inout) :: this
    integer, intent(in) :: id
    type(NodoImagen), pointer, intent(inout) :: temp
    integer :: alturaRight, alturaLeft, m
    if (.not. associated(temp)) then
        allocate(temp)
        temp%id =  id
        temp%altura = 0
    else if (id < temp%id) then
        call this%add_recursivo(id,temp%left)
        if ((this%getAltura(temp%left) - this%getAltura(temp%right)) == 2) then
            if (id < temp%left%id) then
                temp => this%rsi(temp)
            else
                temp => this%rdi(temp)
            end if
        end if
    else
        call this%add_recursivo(id,temp%right)
        if ((this%getAltura(temp%right) - this%getAltura(temp%left)) == 2) then
            if (id > temp%right%id) then
                temp => this%rsd(temp)
            else
                temp => this%rdd(temp)
            end if
        end if
    end if
    alturaRight = this%getAltura(temp%right)
    alturaLeft = this%getAltura(temp%left)
    m = this%getMax(alturaRight,alturaLeft)
    temp%altura = m + 1
end subroutine add_recursivo
```

Capas:

La estructura utilizada para el almacenamiento de las capas es un árbol binario de búsqueda (ABB) para gestionar capas dentro de una estructura organizada, en la que cada nodo contiene una clave y una matriz dispersa asociada a esa capa. Al insertar un nuevo nodo en el árbol, se compara la clave del nodo a insertar con las claves de los nodos existentes para decidir si se debe colocar a la izquierda o a la derecha, manteniendo así el orden del árbol. La inserción es recursiva y garantiza que el árbol mantenga sus propiedades de búsqueda binaria, donde cada nodo tiene una clave menor que todas las claves del subárbol derecho y mayor que todas las del subárbol izquierdo.

Además de la inserción, el módulo incluye procedimientos para imprimir el árbol en orden (lo que resulta en las claves ordenadas), graficar la estructura del árbol, buscar nodos específicos por clave para recuperar o modificar sus matrices, y recorridos del árbol (preorden, inorden, postorden) que son fundamentales para operaciones como la visualización o el procesamiento en profundidad del árbol. Las rutinas de graficado permiten visualizar la estructura del árbol de manera gráfica, utilizando la herramienta Graphviz. Los métodos de recorrido de amplitud y límite permiten explorar el árbol nivel por nivel o hasta un cierto límite, facilitando el análisis o la manipulación de subconjuntos de nodos. La funcionalidad de contar capas y obtener la raíz son utilidades que ofrecen información rápida sobre la estructura y tamaño del árbol. En conjunto, estas herramientas proporcionan una gestión eficiente y flexible de las capas, permitiendo operaciones complejas de manejo y visualización de datos en estructuras anidadas como pueden ser imágenes o mapas compuestos por capas.

```
! Subrutina recursiva para insertar el nodo en la posición correcta
recursive subroutine insertarRecursivo(nodo, nuevoNodo)
    type(NodoCapa), pointer, intent(inout) :: nodo
    type(NodoCapa), pointer, intent(in) :: nuevoNodo

    if (nuevoNodo%capa%key < nodo%capa%key) then
        if (.not. associated(nodo%left)) then
            nodo%left => nuevoNodo
        else
            call insertarRecursivo(nodo%left, nuevoNodo)
        end if
    else if (nuevoNodo%capa%key > nodo%capa%key) then
        if (.not. associated(nodo%right)) then
            nodo%right => nuevoNodo
        else
            call insertarRecursivo(nodo%right, nuevoNodo)
        end if
    else
        print *, "La clave ya existe en el árbol."
    end if
end subroutine insertarRecursivo
```

```
! Subrutina recursiva para buscar un nodo
recursive function buscarRecursivo(nodo, key) result(nodoEncontrado)
    type(NodoCapa), pointer, intent(in) :: nodo
    integer, intent(in) :: key
    type(NodoCapa), pointer :: nodoEncontrado

    ! Si el nodo actual es nulo, no se encontró el nodo buscado
    if (.not. associated(nodo)) then
        nodoEncontrado => null()
        return
    end if

    ! Si la clave buscada es menor que la clave del nodo actual, buscar en el subárbol izquierdo
    if (key < nodo%capa%key) then
        nodoEncontrado => buscarRecursivo(nodo%left, key)
    ! Si la clave buscada es mayor que la clave del nodo actual, buscar en el subárbol derecho
    else if (key > nodo%capa%key) then
        nodoEncontrado => buscarRecursivo(nodo%right, key)
    ! Si la clave buscada es igual a la clave del nodo actual, hemos encontrado el nodo
    else
        nodoEncontrado => nodo
    end if
end function buscarRecursivo
```

```
subroutine ingresarMatriz(arbol, key, fila, columna, color)
    class(ArbolCapas), intent(in) :: arbol
    integer, intent(in) :: key, fila, columna
    character(len=7), intent(in) :: color

    call matrizRecusrivo(arbol%raiz, key, fila, columna, color)

end subroutine ingresarMatriz

! Subrutina recursiva para buscar un nodo
recursive subroutine matrizRecusrivo(nodo, key, fila, columna, color)
    type(NodoCapa), pointer, intent(in) :: nodo
    integer, intent(in) :: key, fila, columna
    character(len=7), intent(in) :: color

    ! Si la clave buscada es menor que la clave del nodo actual, buscar en el subárbol izquierdo
    if (key < nodo%capa%key) then
        call matrizRecusrivo(nodo%left, key,fila,columna,color)
    ! Si la clave buscada es mayor que la clave del nodo actual, buscar en el subárbol derecho
    else if (key > nodo%capa%key) then
        call matrizRecusrivo(nodo%right, key, fila, columna, color)
    ! Si la clave buscada es igual a la clave del nodo actual, hemos encontrado el nodo
    else
        call nodo%capa%matriz%agregarMatriz(fila,columna,color)
    end if
end subroutine  matrizRecusrivo
```

```
subroutine preorder(this, tmp)
    class(ArbolCapas), intent(in) :: this
    type(NodoCapa), intent(in), pointer :: tmp
    if( .not. associated(tmp)) then
        return
    end if
    write (*, '(A,I0)', advance='no') ' Capa: ',tmp%capa%key
    call this%preorder(tmp%left)
    call this%preorder(tmp%right)
end subroutine preorder

subroutine inorder(this, tmp)
    class(ArbolCapas), intent(in) :: this
    type(NodoCapa), intent(in), pointer :: tmp
    if( .not. associated(tmp)) then
        return
    end if
    call this%inorder(tmp%left)
    write (*, '(A,I0)', advance='no') ' Capa: ',tmp%capa%key
    call this%inorder(tmp%right)
end subroutine inorder
```

Generación de Imágenes:

La realización de las imágenes se hacen por medio de una estructura lineal de datos, en esta caso una matriz dispersa, esta estructura es eficiente cuando la mayoría de estos son valores por defecto (como ceros) y solo unos pocos difieren de ese valor por defecto. En el contexto de esta matriz, se utilizan nodos para almacenar las posiciones (fila y columna) y el color (o cualquier otro valor relevante) de los elementos no predeterminados. La inserción de un nodo comienza verificando si existe un nodo cabeza (`head`); si no, se crea uno. Luego, busca si ya existen encabezados de fila y columna para las posiciones dadas; de no ser así, los crea. Finalmente, inserta el nuevo nodo en la posición correcta, manteniendo las referencias adecuadas con los nodos adyacentes para preservar la estructura de la matriz.

Las rutinas de búsqueda (`searchColumn` y `searchRow`) y verificación (`nodeExits`) son fundamentales para mantener la integridad de la matriz al insertar o modificar nodos, asegurando que no haya duplicados y que la matriz se expanda adecuadamente al añadir nuevos elementos. `graficarMatrizDispersa` y `generarImagen` son métodos para visualizar la estructura y el contenido de la matriz, respectivamente, lo que puede ser especialmente útil para representar gráficamente datos complejos como imágenes pixeladas, donde cada nodo podría representar un pixel de color diferente del fondo o valor predeterminado. Estas rutinas utilizan Graphviz para generar gráficos que ilustran tanto la estructura lógica de la matriz (con `graficarMatrizDispersa`) como una representación visual del contenido representado por la matriz (con `generarImagen`), facilitando la interpretación y el análisis de los datos almacenados. La capacidad de "vaciar" la matriz (`vaciarMatriz`) permite reiniciar o reutilizar la estructura para nuevos datos sin la necesidad de crear una nueva instancia, optimizando el uso de recursos en aplicaciones que requieran múltiples operaciones sobre matrices dispersas.

```
subroutine graficarMatrizDispersa(self)
    class(matrizDispersa), intent(inout) :: self
    character(len=18) :: filename = "capaLogico"
    integer :: i, j, fileUnit, iostat
    type(nodo_matriz), pointer :: aux
    type(nodovalor) :: val
    character(len=256) :: dotPath, pngPath
    aux => self%head%down

    dotPath = 'dot/' // trim(filename) // '.dot'
    pngPath = 'img/' // trim(adjustl(filename))

    print *,"Generando matriz dispersa..."
    open(newunit=fileUnit, file=dotPath, status='replace', iostat=iostat)
    if (iostat /= 0) then
        print *, "Error al abrir el archivo."
        return
    end if

    write(fileUnit, *) "graph matriz {"
    write(fileUnit, *) "    node [shape=box];"

    write(fileUnit,'(A)') '"Origen" [label="-1", group = 1]'

    !Generacion de encabezados de cada columna
    do j=0, self%ancho
        write(fileUnit,'(A,I0,A,I0,A,I0,A)') '"Col', j, '"[label="', j, '", group =', j+2, ']'
        if (.not. j == 0) then
        if (j /= self%ancho) then
            write(fileUnit,'(A,I0,A,I0,A)') '"Col', j, '" --  "Col', j+1, '"'
        end if
        else
            write(fileUnit,'(A,I0,A,I0,A)') '"Origen" --  "Col', j, '"'
            write(fileUnit,'(A,I0,A,I0,A)') '"Col', j, '" --  "Col', j+1, '"'
        end if
    end do
```

```
write(fileUnit, *) ' { rank=same; "Origen";'

do j=0, self%ancho
    write(fileUnit, '(A,I0,A)', advance='no') ' "Col', j, '";'
end do

write(fileUnit, *) " }"

! Generacion de encabezados de cada Fila
do i=0, self%largo
    write(fileUnit,'(A,I0,A,I0,A)') '"Fil', i, '"[label="', i, '", group = 1]'
    if (.not. i == 0) then
    if (i /= self%largo) then
        write(fileUnit,'(A,I0,A,I0,A)') '"Fil', i, '" --  "Fil', i+1, '"'
    end if
    else
        write(fileUnit,'(A,I0,A,I0,A)') '"Origen" --  "Fil', i, '"'
        write(fileUnit,'(A,I0,A,I0,A)') '"Fil', i, '" --  "Fil', i+1, '"'
    end if
end do

do i = 0, self%largo
    do j = 0, self%ancho
        val = self%getValor(i,j)
        if(.not. val%existe) then
            write(fileUnit,'(A,I0,A,I0,A,I0,A)') '"F',i, 'C', j, '" [label=" ", group=',  j+2,']'
        else
            write(fileUnit, '(A,I0,A,I0,A,A,A)') '"F',i, 'C', j, '" [label=" ", style=filled, fillcolor="', &
            val%valor, '"]'
        end if
        if (i == 0 .and. j == 0) then
            write(fileUnit,'(A,I0,A,A,I0,A,I0,A)') '"Fil', i, '" -- ', '"F',i, 'C', j, '"'
            write(fileUnit,'(A,I0,A,A,I0,A,I0,A)') '"Col', j, '" -- ', '"F',i, 'C', j, '"'
        else if (i == 0) then
            write(fileUnit,'(A,I0,A,A,I0,A,I0,A)') '"Col', j, '" -- ', '"F',i, 'C', j, '"'
        else if (j == 0) then
```