UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERIA
ESCUELA DE CIENCIAS Y SISTEMAS
LABORATORIO ESTRUCTURA DE DATOS

Manual Técnico Fase 2

Pixel Print Studio

Nombre: Sergio Joel Rodas Valdez
Carné: 202200271
Sección: C

Introducción

Este manual técnico proporciona una guía completa para navegar y utilizar aplicaciones Fortran diseñadas para la gestión avanzada de imágenes jerárquicas. La aplicación tiene la capacidad de adaptarse a diferentes necesidades de gestión de imágenes basadas en estructuras de datos complejas como árboles de búsqueda binaria (BST), árboles AVL, árboles B, matrices dispersas y listas enlazadas, que en conjunto forman la aplicación. Procesamiento organizado de imágenes. El manual está dividido en dos partes principales una para administradores del sistema y otra para usuarios finales. Para los administradores, las operaciones clave como insertar, eliminar y modificar clientes en la estructura de datos se explican en detalle y se acompañan de visualizaciones gráficas para garantizar la integridad del sistema y el rendimiento óptimo. Los usuarios recibirán instrucciones paso a paso sobre cómo cargar datos por lotes, generar y administrar imágenes a partir de capas y visualizar la estructura de datos subyacente. El manual también cubre aspectos técnicos en profundidad, como la generación de imágenes utilizando diversas técnicas, incluido el recorrido de árboles y la manipulación de capas individuales, brindando una descripción completa de las operaciones disponibles en la aplicación.

Manual Técnico

Este manual técnico describe la aplicación desarrollada en Fortran, diseñada para el registro y gestión de imágenes por capas de los clientes. La aplicación se fundamenta en estructuras de datos no lineales, incluyendo árboles de tipo BST (Binary Search Tree), AVL (Adelson-Velsky and Landis Tree), y B-tree, así como matrices dispersas y listas simples. Estas estructuras constituyen el núcleo de la aplicación, permitiendo una gestión eficiente y organizada de los datos. Se detallará cada estructura de datos para proporcionar un entendimiento profundo de su integración y funcionamiento dentro de la aplicación. La característica principal de la aplicación es un generador de imágenes por capas, que aprovecha elementos predefinidos en archivos JSON para crear imágenes completas. Este manual se divide en dos secciones principales, detallando las funcionalidades disponibles para administradores y usuarios, con el objetivo de facilitar el uso y administración del sistema.

Administrador

En este apartado del manual técnico, nos enfocamos en la administración de clientes, una función clave asignada al administrador del sistema. Esta gestión incluye la modificación, eliminación y generación de reportes de clientes. La estructura de datos central para esta tarea es un árbol B de orden 5. La implementación se basa en una clase Cliente, que incluye atributos como dpi, nombre, y password. Cada instancia de Cliente está vinculada a sus propias estructuras de datos: un árbol AVL, un árbol BST, una lista de listas para organizar los álbumes y una matriz dispersa que gestiona las capas de imágenes. El árbol B se estructura alrededor de nodos denominados node_treeB. Cada nodo contiene un arreglo de objetos de tipo Cliente, un identificador que cuenta el número de valores o claves (para determinar cómo dividir la página en el árbol), y un conjunto de enlaces a sus nodos hijos, representados por un arreglo de punteros del tipo node_treeB. La estructura del árbol B se completa con un puntero que actúa como la raíz del árbol. Los procedimientos asociados a esta estructura, que permiten la ejecución de las operaciones de gestión de clientes, serán detallados en este apartado del manual.

- **Insertar un cliente en el árbol B:** esta funcionalidad es implementada por la definición según Knuth que nos dice lo siguiente si el árbol b está vacío, crear el nodo raíz e insertar la llave que sería nuestro nodo, Si el árbol no está vacío buscar el nodo correcto para insertar el nodo es decir que se tendrá que ir ordenado de menor a mayor, Si el nodo correcto está lleno tendríamos que hacer lo siguiente:
    - Insertar la llave en orden ascendente
    - Partir el nodo en dos izquierda y nodo derecho y el nodo que quede en medio ser ahora nuestro nodo padre y esta va tener los nodos hijos izquierdos y derechos
    - Si el nodo padre llego a k llaves repetir el proceso anterior mente descrito para dicho nodo
    - Si el nodo correcto no está lleno insertar el nodo en orden ascendente

  Y es así como funciona el método insertar mediante la implantación de Knuth

```
type nodeptr
    type (node_treeB), pointer :: ptr => null()
end type nodeptr

type cliente
    integer*8 :: dpi !dpi
    character(len=20) :: nombre
    character(len=20) :: password
    type(bst) :: bst_tree
    type(avl) :: avl_tree
    type(Matriz) :: my_matriz
    type(list_album) :: my_album
    contains
        procedure :: report_info_admin
        procedure :: report_infor_user
end type

type node_treeB  !Estrcutura principal Nodo arbol B
    type(cliente) :: cliente(0:5)
    integer :: num = 0 ! cuantas claves o valores
    type(nodeptr) :: link(0:5) !  link : son los que c
end type node_treeB

type TreeB
    type(node_treeB), pointer :: root => null()
    contains
        !subrutinas para insertar un cliente y modific
        procedure :: insertB
        procedure :: setValue
        procedure :: insertNode
        procedure :: splitNode
        procedure :: createNode
        procedure :: traversal
        procedure :: create_dot_treeB
        procedure :: create_graph_treeB
        procedure :: search_node_Btree
        procedure :: modify_node_Btree
```

Estructura del árbol B

```
recursive function setValue(this, val, pval, node_temp, child) result(res)
    class(TreeB), intent(inout) :: this
    type(cliente), intent(in) :: val
    type(cliente), intent(inout) :: pval

    type(node_treeB), pointer, intent(inout) :: node_temp
    type(node_treeB), pointer, intent(inout) :: child
    type(node_treeB), pointer :: newnode ! nuevo nodo
    integer :: pos
    logical :: res

    allocate(newnode)
    if (.not. associated(node_temp)) then
        pval = val
        child => null()
        res = .true. ! insertado un nodo
        return
    end if

    if (val%dpi < node_temp%cliente(1)%dpi) then ! nuevo valor < node_temp[1] ---
        pos = 0
    else
        pos = node_temp%num
        do while (val%dpi < node_temp%cliente(pos)%dpi .and. pos > 1) ! si valor
            pos = pos - 1
        end do
        if (val%dpi == node_temp%cliente(pos)%dpi) then ! o el valor es igual a l
            print *, "No se permiten duplicados"
            res = .false.
            return
        end if
    end if
```

Función para insertar un cliente (lógica para dividir una página si es necesario)

- **Eliminar un cliente en el árbol B:** esta funcionalidad también tiene reglas para mantener la estructura correcta del árbol B que se describe a continuación

    - Si el orden del árbol es m, cada hoja debe tener al menos (m/2)-1 claves Si la clave a eliminar se encuentra en una hora, se elimina directamente
    - Si al realizar la eliminación, el nodo mantiene el mínimo número de claves, finaliza en caso de no ser así de realiza la distribución.

  Eliminación de claves: Para la eliminación de claves Si el elemento no se encuentra en una hoja, se debe "subir" la clave que se encuentra mas a la izquierda del subárbol derecho. Si al subir esta clave, en la hoja respectiva no se cumple el mínimo numero de claves se debe realizar una distribución.

  Redistribución: Si una hoja vecina inmediata tiene suficientes claves disponibles realiza un préstamo la clave que se encuentra mas a la izquierda sube y la clave del nodo padre baja.
  Si no se cuenta con las suficientes claves, la hoja donde se ha eliminado la clave, la hoja adyacente y la clave del padre se juntan en un solo nodo realizando una combinación.

```
recursive subroutine deleteRegister(this, actual, value, found)
    class(TreeB), intent(inout) :: this
    type(node_treeB), pointer, intent(inout) :: actual
    integer*8, intent(in) :: value
    logical, intent(out) :: found
    integer :: k

    if(associated(actual))then
        found = this%search_node_on_page(actual, value, k)
        if (found) then
            if (.not. associated(actual%link(k-1)%ptr))then ! si eres un nojo hoja
                call this%quitar(actual, k)
            else
                call this%sucesor(actual, k)
                call this%deleteRegister(actual%link(k)%ptr, actual%cliente(k)%dpi, found)
            end if
        else
            call this%deleteRegister(actual%link(k)%ptr, value, found)
        end if

        if(associated(actual%link(k)%ptr))then
            if(actual%link(k)%ptr%num < MINI)then
                call this%restablecer(actual, k)
            end if
        end if
    end if
end subroutine deleteRegister
```

Función para eliminar un cliente

- **Modificar un cliente:** Para esta implementación solo se busca el nodo mediante el DPI y si es correcto entonces modifica el nombre y la contraseña del usuario esta función es mas sencilla que insertar y eliminar.

Reporte de Administrador: En la sección "Reporte de Administrador" del manual técnico, abordamos el proceso detallado de cómo se gestiona y visualiza la información de los usuarios en el sistema. La visualización de datos es esencial y se logra mediante una estructura de lista simple, que facilita el recorrido por niveles para acceder a los identificadores de imágenes (id_imagen) asociados a cada usuario.
La funcionalidad del sistema permite determinar y mostrar la cantidad de imágenes que posee cada usuario, utilizando una función específica para contar estas imágenes. Este procedimiento implica un recorrido por el árbol B, durante el cual se presentan detalles como el nombre, DPI y el total de imágenes de cada usuario, mostrados de manera organizada por nivel.
Además, se ha implementado una funcionalidad crítica de búsqueda. Esta operación consiste en localizar a un usuario específico dentro del árbol B. Al encontrar el nodo correspondiente, el sistema despliega la información detallada del cliente. Si el nodo buscado no existe en el árbol, no se ejecuta ninguna acción. Esta funcionalidad asegura que los administradores puedan acceder rápidamente a los datos completos de los usuarios para su gestión y análisis.

```
subroutine report_info_admin(this)
    class(cliente), intent(in) :: this
    integer :: count_node_avl
    integer :: count_node_bst

    print*, ""
    print*, "DPI: ", this%dpi
    print*, "Nombre: ", trim(this%nombre)
    print*, "Password: ", trim(this%password)

    print *, ""
    print *, "Informacion del cliente"
    print *, "-----------------------"
    call list_albums_and_images(this%my_album) ! reco

    count_node_avl = total_nodes_avl(this%avl_tree) !
    count_node_bst = total_nodes_bst(this%bst_tree) !

    print *, "Total de imagenes: ", count_node_avl
    print *, "Total de capas:",  count_node_bst
    print *, "-----------------------"
    print *, ""
end subroutine report_info_admin
```

Método para mostrar la información de un cliente

Grafica del árbol B: En la sección "Gráfica del Árbol B" de nuestro manual técnico, detallamos el proceso para visualizar esta estructura de datos clave dentro de la aplicación. La herramienta elegida para esta tarea es Graphviz, un software de visualización gráfica que facilita la comprensión de las estructuras complejas. Para generar la visualización, se crea un archivo en formato DOT, el cual describe la estructura del Árbol B. Este archivo se genera mediante un recorrido programático a través del Árbol B, capturando y documentando cada nodo y su relación dentro de la estructura. Luego, Graphviz procesa este archivo DOT y produce una representación gráfica del árbol. Esta visualización gráfica es esencial para verificar la correcta construcción y funcionamiento del Árbol B.

Usuario

Carga masiva Usuario

En la sección "Proceso de Carga Masiva para la Generación de Estructuras e Imágenes" de nuestro manual técnico, se describe cómo los usuarios inician la configuración de sus recursos en la aplicación mediante la carga de datos. Este proceso es fundamental para establecer las estructuras necesarias que soportarán la generación y gestión de imágenes.

Carga Masiva de Capas: Inicia con la importación de capas a través de un archivo JSON, especificado por el usuario en consola. Este archivo se procesa para poblar el Árbol BB, estableciendo así la estructura fundamental de capas que se utilizarán en la creación de imágenes.

Carga Masiva de Imágenes: La segunda fase implica la carga de imágenes, donde el usuario importa un conjunto inicial de imágenes a través de otro archivo JSON. Estas imágenes se integran en la estructura del Árbol AVL, preparando el sistema para la manipulación y gestión de las imágenes.

Carga Masiva de Álbumes: La tercera etapa consiste en la carga de álbumes, donde se definen y estructuran los álbumes del usuario. Este proceso contribuye a la organización personalizada del portafolio de imágenes del usuario.

Para facilitar estas cargas masivas, se emplea la biblioteca json-fortran, la cual permite la lectura eficiente de los archivos JSON. Mediante punteros se extrae y procesa la información para su inserción en las estructuras correspondientes: Árbol BB, Árbol AVL, y la lista de álbumes.

Una vez completada la carga y establecidas las estructuras, las mismas quedan listas para su uso, permitiendo al usuario realizar diversas acciones dentro de la aplicación. Es crucial destacar que la información y estructuras de cada cliente se mantienen persistentes; es decir, se conservan entre sesiones, asegurando que, al cerrar y posteriormente reabrir la sesión, el usuario retome su trabajo con todas sus estructuras intactas y accesibles.

En esta parte el usuario puede realizar varias acciones desde registrarse en el sistemas hasta ver sus reportes, en este apartado se describirá la estructuras que fueron utilizadas , las cuales fueron la matriz dispersa, el árbol avl, el árbol bst y la lista de listas de mostrara en apartador cada parte y que acción realiza en nuestra aplicación

Matriz dispersa: En la sección "Matriz Dispersa" de nuestro manual técnico, se describe cómo esta estructura es fundamental para la representación de imágenes en la aplicación. Una matriz dispersa es utilizada para almacenar eficientemente los píxeles de una imagen, donde cada píxel se define por su posición (fila y columna) y su color. Esta organización permite la superposición de capas para formar la imagen completa.
Para manejar esta estructura, se define un nodo que contiene las propiedades de posición y color del píxel, así como apuntadores up, down, left, y right. Estos apuntadores sirven para navegar entre los píxeles adyacentes en la matriz, facilitando la identificación de relaciones entre ellos.
Los procedimientos principales de la matriz dispersa incluyen la inserción de píxeles, la construcción de la representación gráfica de la matriz y la creación de imágenes a través

de métodos como add_layer y per_layer. Estos métodos son cruciales para llamar las capas necesarias y construir la matriz dispersa completa que representa la imagen final.

```
type :: Node
    integer :: row,column
    character(len = 10):: color = "white"
    type(Node), pointer :: up=> null(), down=> null(), right => null(), left=>null()
end type

type :: Matriz
    type(Node), pointer :: root => null()
    integer :: width = 0
    integer :: height = 0
    contains
    procedure :: initialize_matriz
    procedure :: insert
    procedure :: insertRowHeader
    procedure :: insertColumHeader
    procedure :: insertInRow
    procedure :: insertInColumn
    procedure :: searchRow
    procedure :: searchColumn
    procedure :: existNode
    procedure :: showMatrixSparse
    procedure :: printColumnHeaders
    procedure :: getValue
    procedure :: addLayer
```

Estructura de la matriz dispersa

Árbol BB: En la sección "Árbol BB" del manual técnico, exploramos la estructura responsable de almacenar las capas utilizadas para generar diversas imágenes. Cada nodo dentro de este árbol tiene potencialmente dos hijos: izquierdo (left) y derecho (right), lo que permite estructurar de manera jerárquica las capas de imagen. Un nodo se considera hoja cuando no posee hijos. Dentro de cada nodo se almacena un identificador único, id_capa, que sirve para organizar el árbol, y un elemento de datos que representa la capa en sí, usualmente una matriz que contiene el arreglo de píxeles de esa capa específica. De esta forma, cada capa en el árbol BB está compuesta por un conjunto de píxeles que, cuando se combinan de acuerdo con las operaciones definidas en el árbol, pueden formar partes o la totalidad de una imagen.

Este método organizativo no solo facilita la recuperación y manipulación de capas individuales, sino que también optimiza el proceso de generación de imágenes, permitiendo construir imágenes completas o parciales basadas en las selecciones de capas. La funcionalidad detallada de generación de imágenes y cómo interactúa con el árbol BB se explicará más adelante en el manual.

```
type :: node_bst
    integer :: id_capa ! Guia para ordenar mi bs
    type(Matriz), pointer :: capa => null()
    type(node_bst), pointer :: left => null()
    type(node_bst), pointer :: right => null()
end type

type :: bst
    type(node_bst), pointer :: root => null()
    contains
    procedure :: add_bst
    procedure :: preorder
    procedure :: inorder
    procedure :: postorder
    procedure :: amplitud

    procedure :: per_layer
    procedure :: process_layer ! crear matriz co
    procedure :: text_preorden ! text label prec
    procedure :: text_inorden  ! text label inor
    procedure :: text_postorder ! text label pos
    procedure :: search_node_bst ! buscar un noc
    procedure :: inicializate_bst ! inicializar
    procedure :: tree_bst_heigth ! obtener la al
    procedure :: grap_bst ! graficar mi arbol b
    procedure :: process_level ! procesar por ni
    procedure :: inorderTraversal

    ! listar por order (reportes usuario)
    procedure :: list_preoder
    procedure :: list_inorder
    procedure :: list_postorder
    procedure :: print_leaf_nodes

end type
```

Estructura del árbol BB

Árbol Avl: En la sección "Árbol AVL" de nuestro manual técnico, describimos una estructura de datos crítica para la eficiencia de la aplicación el árbol avl es el encargado de almacenar las imágenes mediante el id_imagen, y su árbol correspondiente que nos servirá para generar la imagen a partir de las capas que lo conforman. El Árbol AVL, una forma avanzada de árbol binario, es esencial por su capacidad para mantenerse balanceado en todo momento, lo cual asegura que las operaciones de inserción, búsqueda y eliminación se realicen en tiempo óptimo.

A diferencia de un árbol binario de búsqueda (BB) común, que simplemente inserta nodos, el Árbol AVL ajusta su estructura mediante un control de balance estricto, con factores de balance que varían entre -1 y 1. Este equilibrio se logra a través de rotaciones específicas, que se aplican según la distribución de pesos (alturas de subárboles) en el árbol.

La estructura de un Árbol AVL incluye la propiedad de altura en cada nodo, utilizada para evaluar la necesidad de balancear el árbol a través de rotaciones. Estas rotaciones pueden ser simples (izquierda o derecha) o dobles (izquierda-derecha o derecha-izquierda), dependiendo del desbalance identificado.

Para la inserción de nodos, el árbol AVL realiza comprobaciones continuas de balance, aplicando las rotaciones necesarias para mantener el árbol equilibrado. Esto asegura una estructura óptima para accesos rápidos, fundamental para el rendimiento general de la aplicación.

Al eliminar un nodo, se debe considerar la reestructuración del árbol para mantener el balance. Si se elimina un nodo que tiene descendencia, se procede a reemplazarlo con el sucesor apropiado, generalmente el hijo mayor de los menores del nodo eliminado, y luego se verifica si son necesarias rotaciones para restaurar el balance del árbol.
La implementación de subrutinas específicas para cada tipo de rotación, junto con la evaluación continua de la altura y balance, garantiza que el Árbol AVL se mantenga balanceado en todo momento, optimizando así las operaciones de gestión de datos.

```
type node_avl
    integer :: id_imagen
    integer :: height = 1
    type(bst), pointer :: tree_bst => null()
    type(node_avl), pointer :: left => null()
    type(node_avl), pointer :: right => null()
end type

type avl
    type(node_avl), pointer :: root => null()
    contains
    procedure :: add_avl
    procedure :: preorder_avl
    procedure :: grap_avl
    procedure :: grap_avl_bst
    procedure :: delete_avl
    procedure :: search_node ! buscar nodo avl
end type
```

Estructura del árbol Avl

```
recursive subroutine add_avl_rec(root, id, new_tree_bst)
    type(node_avl), pointer, intent(inout) :: root
    type(bst), pointer,  intent(in) :: new_tree_bst
    integer, intent(in) :: id

    if (.not. associated(root)) then !si no estas asociado
        allocate(root)
        root%id_imagen = id
        root%tree_bst => new_tree_bst
        root%height = 1
    end if

    if (id < root%id_imagen) then
        call add_avl_rec(root%left, id, new_tree_bst)
    else if(id > root%id_imagen)then
        call add_avl_rec(root%right, id, new_tree_bst)
    end if

    root%height = maxheigth(getheight(root%left), getheight(root%right)) + 1

    ! Si es mayor a +1 entonces puede ser una rotacion (simple izquiera)
    if(get_balance(root) > 1) then
        if(get_balance(root%right) < 0)then ! doble derecha izquierda "-"
            root%right => right_rotation(root%right)
            root => left_rotation(root)
        else
            root => left_rotation(root) ! Aqui es una rotacion simple a la izq
        end if

    end if

    if(get_balance(root) < -1)then
        if(get_balance(root%left) > 0)then
            root%left => left_rotation(root%left)
            root => right_rotation(root)
        else
        root => right_rotation(root)
        end if
    end if
end subroutine add_avl_rec
```

Subrutina add_avl_rec con las rotaciones mencionadas

Lista de lista simple (Album): Para esta parte fue necesario de crear una lisa de listas para ya que definimos un album, un album en donde se encuentras todas sus respectivas imágenes que puede tener un album, para ello fue necesario crear una lista y dentro de esa lista existirá una lista simple que en donde se van a guardar todas nuestras imágenes que le pertenecen a ese album, es importante mencionar que cuando se elimina una imagen desde nuestro avl, también se eliminara del album que la posea.

```
type sub_list
    integer :: id_img
    type(sub_list), pointer :: next => null()
end type

type node_list
    character(len=36) :: album_name
    type(node_list), pointer :: next => null()
    type(sub_list), pointer :: list_i => null()
end type

type list_album
 type(node_list), pointer :: head => null()
 contains
    procedure :: add_album
    procedure :: add_list_img
    procedure :: remove_img_by_id
    procedure :: grap_album
end type
```

Estructura de nuestro álbum

Generación de imágenes

Esta es la parte principal de la aplicación cuando un cliente tiene sus imágenes y a cargado sus capas el cliente puede generar imágenes con varias opciones las cuales se mencionarán:

Por Recorrido Limitado: La funcionalidad de generación de imágenes por Recorrido Limitado se fundamenta en el procesamiento del árbol de capas perteneciente al usuario. Este proceso permite al usuario crear imágenes personalizadas utilizando diferentes métodos de recorrido del árbol, como Preorden, Inorden y Postorden. La mecánica de esta funcionalidad se desarrolla en varias etapas clave:

Inicio del Recorrido: Se desarrollan y emplean funciones especializadas para navegar a través del árbol de capas siguiendo el método de recorrido elegido. Este paso es crucial para determinar la secuencia en la que se accede a las capas, lo que influirá en la composición final de la imagen.

Selección de Capas: Durante el recorrido, el sistema identifica las capas individuales mediante id_capa de cada nodo. Esta identificación es esencial para definir las capas que serán integradas en la generación de la imagen.

Construcción de la Matriz Combinada: Utilizando la subrutina process_layer, el sistema recupera y combina las capas seleccionadas en una matriz unificada. Esta matriz actúa como la base para la imagen que será generada, integrando las capas de acuerdo con el orden establecido en el recorrido del árbol. La imagen se produce en función del tipo de recorrido y el número de capas especificado. Este proceso transforma la matriz combinada en la imagen final, que refleja la estructura y el orden del recorrido del árbol de capas.

Por árbol de imágenes: En la sección sobre "Generación de Imágenes por Árbol de Imágenes", describimos el proceso detallado para crear imágenes a partir de estructuras de datos complejas. Este proceso comienza con la búsqueda de un nodo específico en el Árbol AVL, donde se almacenan todas las imágenes. La búsqueda se inicia con el ingreso del id_Imagen por parte del usuario. Si la imagen correspondiente al id_Imagen existe dentro del Árbol AVL, se inicia el proceso de reconstrucción de la imagen utilizando las capas almacenadas en un Árbol BST asociado a dicho nodo AVL. El proceso de generación de la imagen implica la combinación de todas las capas contenidas en el Árbol BST. Este procedimiento sigue una metodología similar a la del recorrido limitado, con la diferencia de que ahora se busca y se utilizan todas las capas asociadas específicamente al nodo AVL en cuestión. Se realiza un recorrido en amplitud (o recorrido por niveles) dentro del Árbol BST, lo que permite visitar todas las capas de manera secuencial, apilando cada una de ellas para formar la imagen final. El Árbol AVL juega un papel crucial en este proceso, ya que organiza las imágenes de manera balanceada, facilitando un acceso rápido y eficiente a cada una de ellas. A su vez, el Árbol BST dentro de cada nodo AVL asegura que las capas de cada imagen se manejen de forma estructurada, permitiendo una reconstrucción lógica y ordenada de las imágenes a partir de sus capas.

```
! :::: binary tree amplitud
! ::::procesar todos los nodos por medio de cada nivel
subroutine amplitud(this, matriz_combined)
    class(bst), intent(in) :: this
    type(matriz), pointer, intent(inout) :: matriz_combined
    integer :: h, level
    h = this%tree_bst_heigth(this%root)
    do level = 1, h
        call this%process_level(this%root, level, matriz_combined)
    end do
end subroutine amplitud

!::::: recorrer por nivel
recursive subroutine process_level(this ,temp, level, matriz_com_ampl)
    class(bst), intent(in) :: this
    type(node_bst), pointer, intent(in) :: temp
    type(Matriz), pointer, intent(inout) :: matriz_com_ampl
    integer, intent(in) :: level

    if (.not. associated(temp)) return
    if (level == 1) then
        call this%process_layer(temp%id_capa, matriz_com_ampl)
    else
        call this%process_level(temp%left, level-1, matriz_com_ampl)
        call this%process_level(temp%right, level-1, matriz_com_ampl)
    end if
end subroutine process_level
```

Recorrido por amplitud (generar imagen por árbol de imágenes)

Por capa: En la sección "Proceso de Generación de Imágenes a partir de Capas Cargadas", describimos cómo los usuarios pueden crear imágenes utilizando las capas previamente cargadas en el sistema. Inicialmente, la aplicación solicita al usuario que especifique el número de capas que desea graficar, permitiendo la selección de una o varias capas. Una vez determinada la cantidad y las capas específicas, la aplicación procede a buscarlas en el Árbol BB general, que almacena todas las capas disponibles.

Para cada capa especificada por el usuario, la aplicación verifica su existencia en el Árbol BB. Al localizar un nodo correspondiente en el árbol, la capa se selecciona y se apila con las demás elegidas por el usuario, siguiendo el orden en que fueron ingresadas. Este proceso se repite hasta que se han procesado todas las capas especificadas al principio. Una vez completada la selección y apilamiento de capas, es importante mencionar para aplicar las capas se generó una subrutina “per_layer” la cuales en la encargada de apilar las capas conformen iban siendo ingresadas por el usuario, la aplicación finaliza el proceso combinando las capas apiladas para formar la imagen final. Esta imagen resultante se procesa y se presenta al usuario, completando así el ciclo de generación de imágenes a partir de las capas cargadas en el sistema. Este método permite a los usuarios personalizar y generar imágenes complejas a partir de elementos individuales, aprovechando la estructura organizada y eficiente del Árbol BB.

```
case (3) ! Generacion de imagenes por capa
    print *, "Ingrese el numero de id's de capas para generar la imagen"
    read(*, *) count_id

    if(allocated(ids_capas))then ! si tiene datos inicializar para volver a utilizar
        deallocate(ids_capas)
    end if
    allocate(ids_capas(count_id))

    if(associated(matrizC))then
        deallocate(matrizC)
    end if

    ! si no estas asociado
    allocate(matrizC)
    call matrizC%initialize_matriz()

    ! Leer cada elemento
    print *, "Ingrese id's de las capas a graficar"
    do i = 1, count_id
        read(*, *) ids_capas(i)
        call clienteNode%cliente(index)%bst_tree%per_layer(ids_capas(i), matrizC)
    end do

    if(associated(matrizC))then
        call matrizC%create_table("imagen_por_capas.dot")
        deallocate(ids_capas)
    end if

end select
```

Generación de imágenes por capas

Visualización de estructuras

En la sección "Visualización de Estructuras del Sistema" de nuestro manual técnico, destacamos la importancia de poder visualizar y validar la correcta construcción de las estructuras de datos utilizadas, como el Árbol BB, Árbol AVL, la matriz dispersa y la lista de listas de álbumes. Aunque previamente se detallaron las composiciones de estas estructuras, es crucial contar con herramientas que permitan visualizarlas para confirmar su integridad y funcionamiento correcto. La aplicación proporciona diversas funcionalidades para la visualización de estas estructuras, facilitando una comprensión más profunda de su comportamiento y estructura interna. Para el Árbol BB y el Árbol AVL, se generan

representaciones gráficas que muestran la disposición y el balance de los nodos, permitiendo a los usuarios visualizar la organización y las relaciones entre ellos. En cuanto a la matriz dispersa, la visualización se enfoca en representar lógicamente las conexiones entre los nodos, ilustrando cómo cada uno se enlaza con sus adyacentes en todas las direcciones: arriba, abajo, derecha e izquierda. Esta representación detallada ayuda a comprender la estructura de la capa y las relaciones de los píxeles. Por último, la lista de listas que representa los álbumes se visualiza a través de una imagen que expone claramente cómo se estructuran los álbumes y las imágenes contenidas dentro de ellos. Esta funcionalidad de visualización es fundamental para asegurar que las estructuras de datos no solo se construyen correctamente.

```
case(3) ! vizualisacion de estructuras
    op_visualize_structure = 0
    do while(op_visualize_structure /= 6)
        print *, "::::: Visualizacion del estado de las estructuras :::::"
        print *, "[1] Arbol Binario -> Capas"
        print *, "[2] Arbol Avl -> Imagenes "
        print *, "[3] Listado de Albuns"
        print *, "[4] Arbol Avl y ver sus capas"
        print *, "[5] Mostrar Capa - Matriz dispersa"
        print *, "[6] volver al menu usuario"
        print *, "Ingrese la estrcutura que desee vizualizar"
        read(*, *) op_visualize_structure
        select case(op_visualize_structure)
         case (1)
            call clienteNode%cliente(index)%bst_tree%grap_bst("arbol_bst")
         case (2)
            call clienteNode%cliente(index)%avl_tree%grap_avl("arbol_avl")
         case (3)
            call clienteNode%cliente(index)%my_album%grap_album("list_album.dot")
         case (4)
            print *, "Ingrese el id de la imagen"
            read(*,*) selec_img
            call clienteNode%cliente(index)%avl_tree%grap_avl_bst("arbol_avl_bst", selec_img)
        case (5)
            print *, "Ingrese el id de la capa"
            read(*,*) selec_img

            if(associated(matrizC)) then ! Desalogar mi matriz para reutilizarla
                deallocate(matrizC)
            end if

            allocate(matrizC) ! Dar espacio de memoria a mi matriz
            call matrizC%initialize_matriz()
            call clienteNode%cliente(index)%bst_tree%per_layer(selec_img, matrizC)
            if(associated(matrizC))then
                call matrizC%create_dot("grap_matriz")
            end if
        end select
    end do
```

Menú para visualizar las estructuras que posee el usuario

Gestión de imágenes

El usuario también puede gestionar imágenes ya sea añadiendo imágenes a el arbol avl como también eliminando imágenes y esta parte también es importante asi el usuario también puede crear sus propias imágenes y no solo generar las imágenes que tiene ya cargadas.

Registrar una imagen: En la sección "Registro de Imágenes", describimos el procedimiento que debe seguir el usuario para añadir nuevas imágenes al sistema. Este proceso inicia

con la solicitud al usuario de un identificador único (id) para la imagen. Si el id proporcionado ya está asociado a una imagen existente en el sistema, se muestra un mensaje de error en consola indicando que la imagen ya existe. Si el id es único, el sistema entonces solicita al usuario la cantidad de capas que compondrán la imagen. El usuario debe ingresar los identificadores (id) de las capas previamente cargadas en el sistema. La aplicación verifica la existencia de cada capa; si alguna de ellas no existe, se muestra un mensaje de error y se interrumpe el proceso de registro de la imagen, marcando la operación como fallida mediante una bandera. Cuando todas las capas son validadas y existen en el sistema, la aplicación procede a generar y agregar la imagen en el Árbol AVL. Es crucial en este punto asegurar que la inserción de la nueva imagen mantenga el balance del Árbol AVL, cumpliendo con las reglas de balance para optimizar la eficiencia en la búsqueda y acceso a las imágenes dentro del sistema.

Eliminar una imagen: En la sección "Eliminación de Imágenes" de nuestro manual técnico, detallamos el proceso que los usuarios deben seguir para remover imágenes previamente cargadas o ingresadas por ellos mismos. Inicialmente, la aplicación solicita al usuario que proporcione el identificador (id) de la imagen que desea eliminar. Se verifica la existencia del id para asegurar que la imagen a eliminar está presente en el sistema. Una vez confirmada la existencia de la imagen, se procede con la eliminación. Este proceso requiere considerar varios factores para mantener la integridad y el balance del árbol de imágenes, usualmente un Árbol AVL. Los escenarios a considerar durante la eliminación de un nodo (en este caso, una imagen) son:

Sin Nodo Derecho: Si el nodo a eliminar no tiene hijo derecho, el hijo izquierdo asume la posición del nodo eliminado.

Sin Nodo Izquierdo: Si el nodo a eliminar no tiene hijo izquierdo, el hijo derecho toma su lugar.

Con Hijos Derecho e Izquierdo: Si el nodo a eliminar tiene ambos hijos, se reemplaza con el menor de los mayores, es decir, el sucesor inmediato en el subárbol derecho.

Posteriormente, es crucial verificar si el árbol mantiene su balance después de la eliminación. Si el árbol resulta desbalanceado, se deben realizar las rotaciones necesarias para restablecer el equilibrio. Este enfoque asegura que la estructura del árbol se preserve y que el árbol permanezca balanceado, manteniendo así la eficiencia en operaciones de búsqueda y acceso posteriores. También es importante mencionar que al momento de eliminar una imagen y si existe esa imagen dentro de un álbum entonces, también se procede a eliminar ese elemento del álbum que contiene esa imagen.

```
recursive function deleteRec (currentNode, val) result(res)
    type(node_avl), pointer :: currentNode
    integer, intent(in) :: val
    type(node_avl), pointer :: res
    type(node_avl), pointer :: temp

    if(.not. associated(currentNode)) then
        res => currentNode
        return
    end if

    if(val < currentNode%id_imagen)then
        currentNode%left => deleteRec(currentNode%left, val)

    else if(val > currentNode%id_imagen)then
        currentNode%right => deleteRec(currentNode%right, val)

    else  ! Si no es mayor ni menor encontro el nodo a eliminar
        if(.not. associated(currentNode%left)) then
            temp => currentNode%right
            currentNode%tree_bst => null()
            deallocate(currentNode)
            res => temp
        else if(.not. associated(currentNode%right))then
            temp => currentNode%left
            currentNode%tree_bst => null()
            deallocate(currentNode)
            res => temp
        else  ! si hay un nodo izquierdo o derecho
            call get_mayor_of_minors(currentNode%left, temp)
            currentNode%id_imagen = temp%id_imagen
            currentNode%tree_bst => temp%tree_bst ! ----
            currentNode%left => deleteRec(currentNode%left, temp%id_imagen)
        end if
    end if

    if(.not. associated(currentNode)) return

    currentNode%height = maxheigth(getheight(currentNode%left), getheight(currentNode%right)) + 1

    if(get_balance(currentNode) > 1)then
        if(get_balance(currentNode%right) < 0)then
            currentNode%right => right_rotation(currentNode%right)
            currentNode => left_rotation(currentNode)
        else
            currentNode => left_rotation(currentNode)
        end if
    end if
```

Función de eliminar recursivamente, para un nodo en el árbol AVL

Reporte de Usuario

En este apartado se muestra información acerca del usuario, pero se basa más en las capas que tiene el usuario.

- **Imágenes que más capas contienen:** Se inicia destacando las imágenes con el mayor número de capas, identificando así los elementos más complejos en la estructura del usuario. Esta lista se genera y muestra en consola, ofreciendo una perspectiva clara de las imágenes que más capas contienen.

- **Identificación de Capas Hojas:** Posteriormente, se analizan y listan las capas que no tienen descendencia, conocidas como 'hojas', proporcionando un vistazo a los elementos más simples o finales en la jerarquía de capas.

- **Profundidad del Árbol de Capas:** Se examina y reporta la profundidad del Árbol BB, que refleja el número de niveles que componen la estructura de capas, ofreciendo una medida de la complejidad y la escala del conjunto de capas del usuario.

- **Recorridos del Árbol de Capas:** Se presentan los recorridos (preorden, inorden y postorden) del árbol. Estos recorridos son esenciales para entender la organización interna y el método de acceso a las capas, fundamentales para la generación de imágenes.

```
subroutine report_infor_user(this)
    class(cliente), intent(in) :: this
    integer :: altura_bst


    if(associated(this%avl_tree%root))then
        call report_top_images(this%avl_tree)
    end if

    if (associated(this%bst_tree%root))then
        altura_bst = tree_bst_heigth(this%bst_tree, this%bst_tree%root)
        write(*, '(A, I0)') "Profundidad del arbol de capas: ", altura_bst
        print *, "Todas las capas que son hojas"
        call print_leaf_nodes(this%bst_tree, this%bst_tree%root)
        print *, ""
        print *, "--- Listar capas -- "
        print *, "Preorder"
        call list_preoder(this%bst_tree, this%bst_tree%root)
        print *, ""
        print *, "Inorder"
        call list_inorder(this%bst_tree, this%bst_tree%root)
        print *, ""
        print *, "PostOder"
        call list_postorder(this%bst_tree, this%bst_tree%root)
        print *, ""
    end if
end subroutine report_infor_user
```

Subrutina para mostrar la información de las capas en consola

Requisitos del Sistema

Software y Entorno de Desarrollo:

- Sistema operativo Windows 64
- Compilador gnu fortran 12.0.3
- Visual Studio Code -> Extensiones: Fortran Moderno

- Para la lectura del json se utilizó json-fortran: Es una biblioteca para Fortran que permite la serialización y deserialización de datos JSON. Facilita la lectura y escritura de datos JSON, permitiendo a las aplicaciones Fortran manipular fácilmente este formato de intercambio de datos ampliamente utilizado.

- cmake versión 3.29.0-rc1
- uuid_module. f90 -> para darle un nombre único a un nodo

Conclusiones

Estructuras de datos complejas: esta guía cubre el uso de estructuras de datos avanzadas como BST, AVL y árboles B, así como matrices dispersas y listas simples, que son esenciales para el procesamiento eficiente de imágenes en capas. Estas estructuras proporcionan un procesamiento de datos organizado y optimizado, que es fundamental para el rendimiento y la funcionalidad de las aplicaciones.

Funcionalidad de administrador y usuario: el manual destaca la división de la funcionalidad entre administradores y usuarios, con secciones dedicadas a cada función. Los administradores se centran en gestionar clientes y mantener la estructura de datos, mientras que los usuarios se centran en cargar y gestionar imágenes, lo que demuestra la versatilidad y amplitud de la aplicación.

Procesos de inserción, eliminación y modificación: explica en detalle cómo se realizan las operaciones de inserción, eliminación y modificación en un árbol B, enfatizando la importancia de mantener la estructura del árbol equilibrada y funcional. Esta parte de la guía es esencial para comprender cómo la aplicación mantiene la integridad de los datos.

Visualización y producción de imágenes: una guía técnica se centra no solo en la estructura y gestión de datos, sino también en cómo estas estructuras facilitan la producción y el procesamiento de imágenes. La visualización de estructuras como árboles B y matrices dispersas utilizando herramientas como Graphviz, así como procedimientos detallados para extraer imágenes de capas cargadas, ilustran el flujo de todo el sistema y la interacción de sus componentes.