Manual Técnico

Introducción

Este documento proporciona una descripción detallada del Sistema de Gestión Integral desarrollado en Fortran, diseñado para facilitar la administración y operación de sucursales, técnicos, y rutas de servicio técnico. El sistema utiliza avanzadas estructuras de datos y algoritmos, incluyendo árboles AVL, tablas hash, y grafos, además de implementar un sistema de blockchain para garantizar la integridad de los datos críticos.

Módulos del Sistema

El sistema se compone de varios módulos interconectados que proporcionan diversas funcionalidades:

1. **Módulo Principal (menu_program):** Gestiona la interfaz de usuario y las interacciones principales.
2. **Módulo de Árbol AVL (Avl_Tree):** Administra las operaciones de estructura de datos para un almacenamiento eficiente y búsqueda de datos.
3. **Módulo de Tabla Hash (hash_table):** Se encarga del manejo de datos técnicos utilizando una estructura de tabla hash.
4. **Módulo de Blockchain (block_chain):** Asegura la integridad de las transacciones mediante la tecnología blockchain.

Descripción de los Módulos

Módulo Principal: menu_program

Funcionalidades:

- **Autenticación de Usuarios:** Verifica las credenciales de los usuarios para permitir acceso al sistema.
- **Navegación de Menús:** Permite al usuario navegar por las opciones de carga de archivos, gestión de sucursales y generación de reportes.
- **Gestión de Sesiones:** Controla el inicio y cierre de sesiones de usuario.

Procesos Principales:

- **login**: Autentica al usuario comparando las credenciales ingresadas con valores predeterminados.
- **menu_programa**: Despliega el menú principal y captura la opción seleccionada por el usuario.
- **menu_sucursal**: Gestiona las operaciones específicas disponibles para una sucursal seleccionada.

Módulo de Árbol AVL: Avl_Tree

Funcionalidades:

- **Inserción y Búsqueda:** Permite insertar nuevos nodos y buscar nodos existentes utilizando un balanceo AVL para mantener el árbol equilibrado.
- **Visualización:** Genera representaciones gráficas del árbol para análisis visual.

Estructuras de Datos y Métodos:

- **Node_t**: Define la estructura de un nodo dentro del árbol, incluyendo datos como identificador, departamento y dirección.
- **Tree_t**: Representa el árbol AVL y contiene métodos para manipular estos árboles, incluyendo inserciones y búsquedas.

```
contains

function new_branch(id,dept,dir,contra) result(nodePtr)
    type(Node_t), pointer :: nodePtr
    integer, intent(in) :: id
    character(:), allocatable,intent(in) ::dept,dir,contra
    allocate(nodePtr)
    nodePtr%id = id
    nodePtr%dept = dept
    nodePtr%direccion = dir
    nodePtr%password = contra
    nodePtr%Factor = 0
    nodePtr%Left => null()
    nodePtr%Right => null()
end function new_branch

subroutine new_avl(self)
    class(Tree_t), intent(inout) :: self
    self%root => null()
end subroutine new_avl

function rotationII(n, n1) result(result_node)
    type(Node_t), pointer :: n, n1, result_node

    n%Left => n1%Right
    n1%Right => n
    if (n1%Factor == -1) then
        n%Factor = 0
        n1%Factor = 0
    else
```

```
        else
            n%Factor = -1
            n1%Factor = 1
        end if
        result_node => n1
    end function rotationII

    function rotationDD(n, n1) result(result_node)
        type(Node_t), pointer :: n, n1, result_node

        n%Right => n1%Left
        n1%Left => n
        if (n1%Factor == 1) then
            n%Factor = 0
            n1%Factor = 0
        else
            n%Factor = 1
            n1%Factor = -1
        end if
        result_node => n1
    end function rotationDD

    function rotationDI(n, n1) result(result_node)
        type(Node_t), pointer :: n, n1, result_node, n2

        n2 => n1%Left
        n%Right => n2%Left
        n2%Left => n
        n1%Left => n2%Right
```

```
function rotationDI(n, n1) result(result_node)
    type(Node_t), pointer :: n, n1, result_node, n2

    n2 => n1%Left
    n%Right => n2%Left
    n2%Left => n
    n1%Left => n2%Right
    n2%Right => n1
    if (n2%Factor == 1) then
        n%Factor = -1
    else
        n%Factor = 0
    end if
    if (n2%Factor == -1) then
        n1%Factor = 1
    else
        n1%Factor = 0
    end if
    n2%Factor = 0
    result_node => n2
end function rotationDI

function rotationID(n, n1) result(result_node)
    type(Node_t), pointer :: n, n1, result_node, n2
    n2 => n1%Right
    n%Left => n2%Right
    n2%Right => n
    n1%Right => n2%Left
    n2%Left => n1
    if (n2%Factor == 1) then
        n1%Factor = -1
```

```
subroutine insert_node(tree, id,dir,dept,contra)
    class(Tree_t), intent(inout) :: tree
    integer, intent(in) :: id
    logical :: increase
    character(:), allocatable,intent(in) ::dept,dir,contra
    increase = .false.
    tree%root => recursive_insert(tree%root, id,dir,dept,contra, increase)
end subroutine insert_node

recursive function recursive_insert(root, id,dir,dept,contra, increase) result(result_node
    type(Node_t), pointer :: root, result_node, n1
    logical :: increase
    character(:), allocatable,intent(in) ::dept,dir,contra
    integer, intent(in) :: id

    if (.not. associated(root)) then
        allocate(result_node)
        root => new_branch(id,dir,dept,contra)
        increase = .true.
    else if (id < root%id) then
        root%Left => recursive_insert(root%Left, id,dir,dept,contra, increase)
        if (increase) then
        select case (root%Factor)
            case (RIGHT_HEAVY)
                root%Factor = 0
                increase = .false.
            case (BALANCED)
                root%Factor = -1
            case (LEFT_HEAVY)
```

Módulo de Tabla Hash: hash_table

Funcionalidades:

- **Almacenamiento de Técnicos:** Almacena y recupera información de técnicos utilizando hashing para una búsqueda eficiente.
- **Rehashing Automático:** Aumenta el tamaño de la tabla automáticamente cuando el factor de carga supera un umbral predeterminado.

```
module hash_table
    implicit none
    private
    integer :: table_size = 7
    real, parameter :: R = 0.618034
    integer, parameter :: MAX_USED_PERCENTAGE = 70
    type tecnico
    integer(8) ::key,telefono
    character(:), allocatable::nombre,apellido,direccion
    end type tecnico
    type, public :: HashTable
        integer :: elements = 0
        type(tecnico), allocatable :: array(:)

        contains
        procedure :: insert
        procedure :: print
        procedure :: search
        procedure, private :: solve_collision
        procedure :: grafico
        procedure :: get_data
    end type HashTable
contains
    subroutine insert(self, key,nombre,apellido,direccion,telefono)
        class(HashTable), intent(inout) :: self
        type(HashTable) :: newTable
        integer(8), intent(in) :: key,telefono
        character(:), allocatable::nombre,apellido,direccion
        type(tecnico), allocatable :: oldArray(:)
        real :: used_percentage
        integer(8) :: pos
```

```
    ! If the table is empty, allocate it
    if(.not. allocated(self%array)) then
        allocate(self%array(0:table_size-1))
        self%array(:)%key = -1  ! Initialize all the elements to -1
    end if

    pos = get_position(key)

    ! If the position is already occupied, solve the collision
    if(self%array(pos)%key /= -1 .and. self%array(pos)%key /= key) then
        call self%solve_collision(key, pos)
    end if

    ! Store the key in the table
    self%array(pos)%key=key
    self%array(pos)%nombre=nombre
    self%array(pos)%apellido= apellido
    self%array(pos)%direccion=direccion
    self%array(pos)%telefono=telefono
    self%elements = self%elements + 1

    ! Check if the table is more than 75% full
    used_percentage = (self%elements * 1.0/table_size) * 100
    if(used_percentage > MAX_USED_PERCENTAGE) then
        ! Deallocate the table
        oldArray = self%array
        deallocate(self%array)
        ! Rehash the table
        newTable = rehashing(oldArray)
```

```
function rehashing(oldArray) result(newTable)
    type(tecnico), intent(in) :: oldArray(:)
    integer :: i
    type(HashTable) :: newTable

    ! Initialize the new table
    table_size = table_size*2
    allocate(newTable%array(0:table_size-1))
    newTable%array(:)%key = -1
    ! Insert the elements in the new table
    do i = 1, size(oldArray)
        if(oldArray(i)%key /= -1) then
        call newTable%insert(oldArray(i)%key,oldArray(i)%nombre,oldArray(i)%apellido,oldArr
        oldArray(i)%telefono)
        end if
    end do
end function rehashing
```

```
subroutine solve_collision(self, key, pos)
    class(HashTable), intent(inout) :: self
    integer(8), intent(in) :: key
    integer(8), intent(inout) :: pos
    integer(8) :: i, step, new_pos

    i = 1  ! Contador de colisiones
    ! Continúa buscando una nueva posición mientras la posición actual esté ocupada y no
    do while (self%array(pos)%key /= -1 .and. self%array(pos)%key /= key)
        ! Calcula el paso usando la fórmula de doble dispersión
        step = mod(key, 7) + 1
        new_pos = mod(pos + step * i, table_size)
        ! Si encuentra una posición libre, sale del bucle
        if (self%array(new_pos)%key == -1) then
            pos = new_pos
            exit
        endif
        i = i + 1  ! Incrementa el contador de colisiones
    end do
end subroutine solve_collision
```

Estructuras de Datos y Métodos:

- **tecnico**: Representa la información de un técnico, incluyendo DPI, nombre, y otros detalles personales.
- **HashTable**: Gestiona un array de técnicos y proporciona métodos para insertar datos y resolver colisiones.

Módulo de Blockchain: block_chain

Funcionalidades:

- **Generación de Bloques:** Crea bloques que encapsulan datos transaccionales y los enlaza en una cadena.
- **Integridad de Datos:** Utiliza hashes y un árbol de Merkle para asegurar la integridad de la cadena de bloques.

Estructuras de Datos y Métodos:

- **block**: Define la estructura de un bloque en la cadena.
- **chainer**: Gestiona la cadena de bloques completa, permitiendo añadir bloques y generar representaciones de la cadena.

```
contains
    subroutine generate_block(this, new_data, new_branches, isPrinters)
    class(block), intent(inout) :: this
    type(result_list) :: new_data
    type(Tree_t):: new_branches
    type(result), pointer :: current
    type(Node_t), pointer :: b_origin, b_destination
    type(merkle) :: new_merkle
    logical, intent(in) :: isPrinters
    integer, dimension(8) :: values
    character(20) :: timestamp
    this%index = block_id
    block_id = block_id + 1
    call date_and_time(values=values)
    write(timestamp, '(I0, A, I0, A, I0, A, I0, A, I0, A, I0)') values(3), '-', values(2) &
        , '-', values(1), '::', values(5), ':', values(6), ':', values(7)
    this%nonce = 4560
    this%timestamp = trim(timestamp)
    this%data = new_data
    this%branches = new_branches
    current => new_data%head
    do while ( associated(current) )
        b_origin => new_branches%searchBranch(current%id)
        if ( associated(b_origin) ) then
            if ( associated(current%next) ) then
                b_destination => new_branches%searchBranch(current%next%id)
                if ( associated(b_destination) ) then
                    if(isPrinters) then
                    call new_merkle%add_data(b_origin%id, b_origin%dept, b_destination%id,&
                    b_destination%dept, current%next%weight*80)
```

Interacción entre Módulos

Los módulos interactúan entre sí para proporcionar una experiencia integrada. Por ejemplo, el módulo principal (**menu_program**) utiliza **Avl_Tree** para gestionar datos de sucursales y **hash_table** para manejar información de técnicos. Además, el módulo **block_chain** se utiliza para registrar operaciones críticas y asegurar su integridad.