Natalia Mariel Calderón Echeverría
202200007
Estructuras de Datos
Primer semestre 2024

MANUAL TÉCNICO

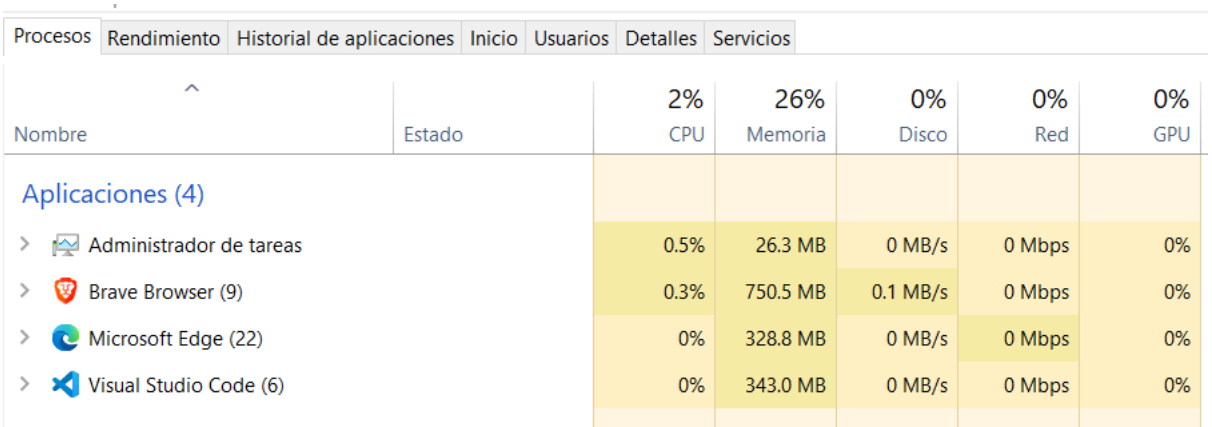
Requerimientos básicos:

- **Lenguaje programado:** FORTRAN
- **Compilador: GFORTRAN**
- **Librerías:**
    - **Json module**

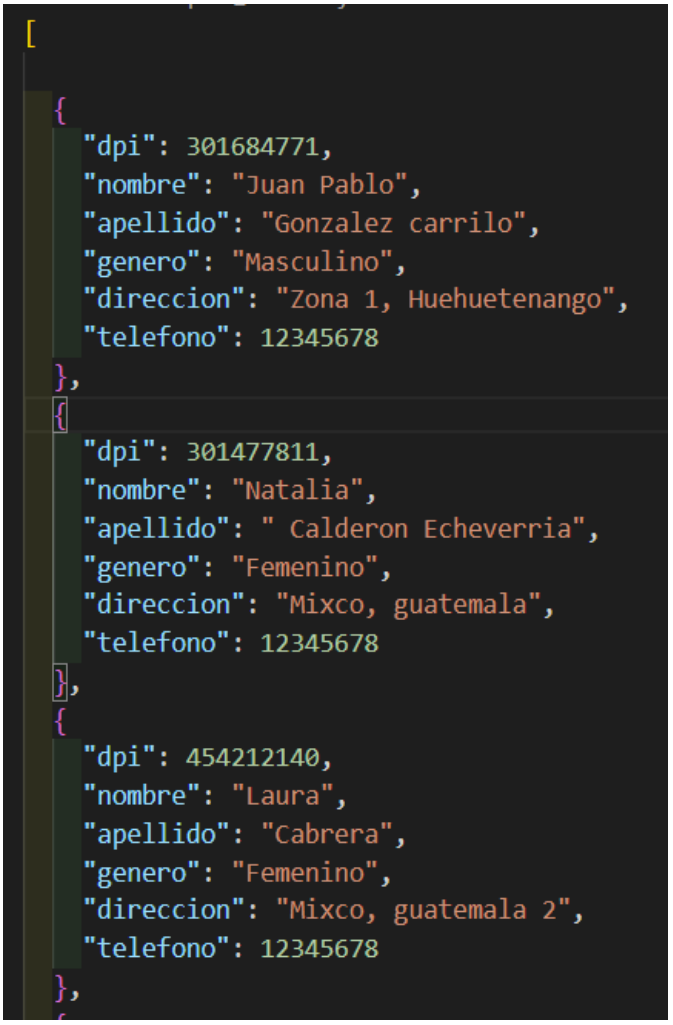
      Json module forma parte fundamental del siguiente proyecto ya que gracias a esta librería es posible leer los archivos json, que son los que contienen la información necesaria para llenar las estructuras de datos disponibles.
    - **Graphviz**

      Graphviz hace posible que crear las representación gráfica de dichas estructuras, gracias a esta librería es posible generar las rutas, las rutas optimas, la tabla hash y el arbol b de sucursales..

Requerimientos del equipo:



- **Json de entrada**
  - **técnicos**



```
[
  {
    "dpi": 301684771,
    "nombre": "Juan Pablo",
    "apellido": "Gonzalez carrilo",
    "genero": "Masculino",
    "direccion": "Zona 1, Huehuetenango",
    "telefono": 12345678
  },
  {
    "dpi": 301477811,
    "nombre": "Natalia",
    "apellido": " Calderon Echeverria",
    "genero": "Femenino",
    "direccion": "Mixco, guatemala",
    "telefono": 12345678
  },
  {
    "dpi": 454212140,
    "nombre": "Laura",
    "apellido": "Cabrera",
    "genero": "Femenino",
    "direccion": "Mixco, guatemala 2",
    "telefono": 12345678
  },
```

La estructura general del archivo consiste en una lista de objetos, donde cada objeto representa a una persona. Dentro de cada objeto, hay varios pares clave-valor que representan diferentes atributos de la persona, como su **número de identificación personal (DPI), nombre, apellido, género, dirección y número de teléfono.**

Los atributos están separados por comas y se encierran entre llaves {}. Cada objeto dentro de la lista está separado por comas, lo que indica que son elementos individuales en la lista de datos.Este se utiliza para llenar la tabla hash.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
tecnicos	-1	-1	-1	301684771	200778456	454212140	301477811	201301456	-1	-1	-1	200325101	-1	-1

- **Sucursales**

Este json describe una serie de registros relacionados con ubicaciones y credenciales que representan cada una de las sucursales del proyectos. Cada objeto en la lista representa una sucursal con atributos como "id", "departamento", "direccion" y "password".

Estos atributos proporcionan información sobre la identificación única del registro, el departamento al que pertenece la ubicación, la dirección específica y la contraseña asociada. Los registros están organizados en una lista, lo que indica que son elementos individuales y están separados por comas.

Esta información es almacenada en un árbol B, en el cual se encuentra toda la información relacionada, siendo el id el número por el cual se ordenan. La contraseña aparece encriptada en los árboles, y en los archivos de persistencia por razones de seguridad.

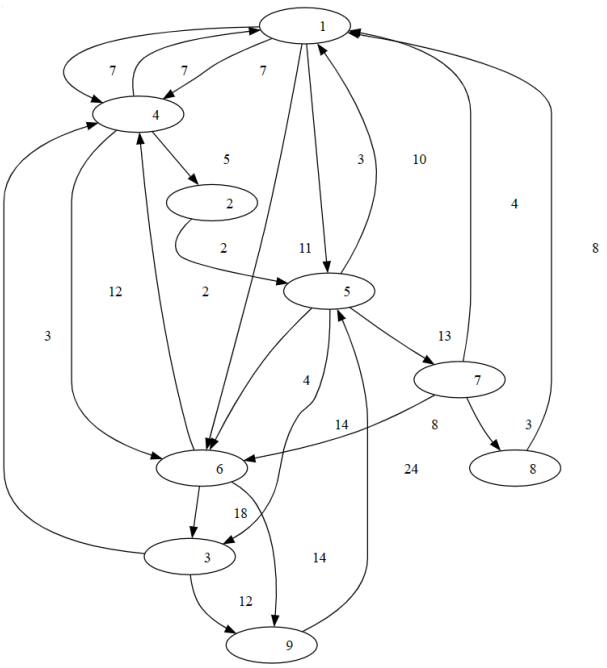

```
[
    {
        "id": 1,
        "departamento": "Guatemala",
        "direccion": "ova calle A ZONA 1",
        "password": "12345"

    },
    {
        "id": 3,
        "departamento": "Guatemala",
        "direccion": "ova calle A ZONA 2",
        "password": "6789"
    },
    {
        "id": 2,
        "departamento": "Huehuetenango",
        "direccion": "ova calle A ZONA 3",
        "password": "12345"
    },
    {
        "id": 4,
        "departamento": "Huehuetenango",
        "direccion": "ova calle A ZONA 4",
        "password": "6789"
    },
```

- **rutas**


```
{
    "grafo":[
        {
            "s1":4,
            "s2":2,
            "distancia":5,
            "imp_mantenimiento":3
        },
        {
            "s1":4,
            "s2":6,
            "distancia":12,
            "imp_mantenimiento":8
        },
        {
            "s1":4,
            "s2":1,
            "distancia":7,
            "imp_mantenimiento":5
        },
        {
            "s1":5,
            "s2":3,
            "distancia":14,
            "imp_mantenimiento":8
        },
        {
```


El json rutas describe la estructura de un grafo dirigido, donde los nodos representan sucursales y los bordes representan las rutas entre ellas. Cada objeto dentro de la lista "grafo" especifica una conexión entre dos sucursales, indicando la sucursal de salida ("s1"), la sucursal de llegada ("s2"), la distancia entre ellas y la importancia del mantenimiento de esa ruta.

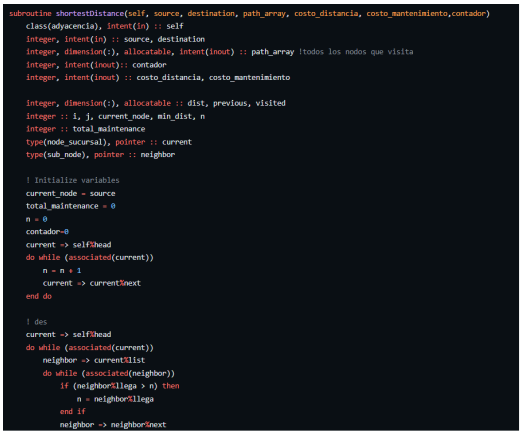
Esta estructura facilita la representación de relaciones entre sucursales en un sistema de distribución o transporte, donde la distancia y el mantenimiento son factores críticos a considerar. Esto hace posible que sea utilizada una lista de adyacencia para poder identificar las conexiones y realizar el grafo correspondiente que representa todas y cada una de las rutas de las sucursales y cómo se conectan entre si.



Los archivos json previamente descritos y las estructuras utilizadas permiten almacenar la información para su posterior análisis y ejecución. El programa consiste en una serie de estructuras que se relacionan entre sí simulando una red de sucursales que dan mantenimiento a las impresoras.

Se pretende calcular el camino que signifique menor gasto, es decir el camino más corto, para esto se hace uso de la información de las rutas recopilada en la matriz de adyacencia y se procede a analizar.

La subrutina shortestDistance calcula la ruta más corta entre un origen y un destino en un grafo representado por una lista de adyacencia. Utiliza el algoritmo de **Dijkstra**, inicializando un array de distancias con valores infinitos y marcando los nodos visitados a medida que avanza. Se actualizan las distancias si se encuentra un camino más corto y se calcula el costo total de mantenimiento. La ruta más corta se guarda en un array y se imprime junto con el costo total.



```
subroutine shortestDistance(self, source, destination, path_array, costo_distancia, costo_mantenimiento,contador)
    class(adyacencia), intent(in) :: self
    integer, intent(in) :: source, destination
    integer, dimension(:), allocatable, intent(inout) :: path_array !todos los nodos que visita
    integer, intent(inout):: contador
    integer, intent(inout) :: costo_distancia, costo_mantenimiento

    integer, dimension(:), allocatable :: dist, previous, visited
    integer :: i, j, current_node, min_dist, n
    integer :: total_maintenance
    type(node_sucursal), pointer :: current
    type(sub_node), pointer :: neighbor

    ! Initialize variables
    current_node = source
    total_maintenance = 0
    n = 0
    contador=0
    current => self%head
    do while (associated(current))
        n = n + 1
        current => current%next
    end do

    ! des
    current => self%head
    do while (associated(current))
        neighbor => current%list
        do while (associated(neighbor))
            if (neighbor%llega > n) then
                n = neighbor%llega
            end if
            neighbor => neighbor%next
```

```
            ! Marcar los visitados
            visited(current_node) = 1

            ! act
            current => self%head
            do while (associated(current))
                if (current%salida == current_node) then
                    neighbor => current%list
                    do while (associated(neighbor))
                        if (dist(current_node) + neighbor%distancia < dist(neighbor%llega)) then
                            dist(neighbor%llega) = dist(current_node) + neighbor%distancia
                            previous(neighbor%llega) = current_node
                            ! aa
                            total_maintenance = total_maintenance + neighbor%mantenimiento
                        end if
                        neighbor => neighbor%next
                    end do
                end if
                current => current%next
            end do
        end do

        ! guardar las sucursales
        allocate(path_array(n))
        current_node = destination
        i = 1
        do while (current_node /= source)
            if (current_node < 1 .or. current_node > n) then
                print *, "Error:", current_node
                exit
            end if
            print *,  current_node
            path_array(i) = current_node
            contador=contador+1
```

```
                do while (associated(neighbor))
                    if (neighbor%llega > n) then
                        n = neighbor%llega
                    end if
                    neighbor => neighbor%next
                end do
                current => current%next
            end do

            print *, "sucursales:", n

            allocate(dist(n))
            allocate(previous(n))
            allocate(visited(n))


            dist(:) = HUGE(dist)
            previous(:) = 0
            visited(:) = 0

            dist(source) = 0

            do i = 1, n
                ! no visitados
                min_dist = HUGE(dist)
                do j = 1, n
                    if (visited(j) == 0 .and. dist(j) < min_dist) then
                        min_dist = dist(j)
                        current_node = j
                    end if
                end do
```

```
            current_node = previous(current_node)
        end do
        path_array(i) = source
        print *, source
        contador=contador+1

        ! Print the path
        print *, "CAMINO CON MENOR DISTANCIA(COSTO) DE: ", source, " -> ", destination, "// DISTANCIA ", dist(destination)
        print *, "COSTO TOTAL DE MANTENIMIENTO:", total_maintenance-6
        costo_distancia=dist(destination)
        costo_mantenimiento=total_maintenance-6
        ! Deallocate
        deallocate(dist, previous, visited)
    end subroutine shortestDistance
```

El algoritmo de Dijkstra es un método de búsqueda de caminos más cortos en un grafo ponderado, donde cada arista tiene un peso que representa la distancia entre dos nodos(en este caso es la distancia entre sucursales). Comienza desde un nodo de origen y calcula la distancia más corta a todos los demás nodos del grafo.

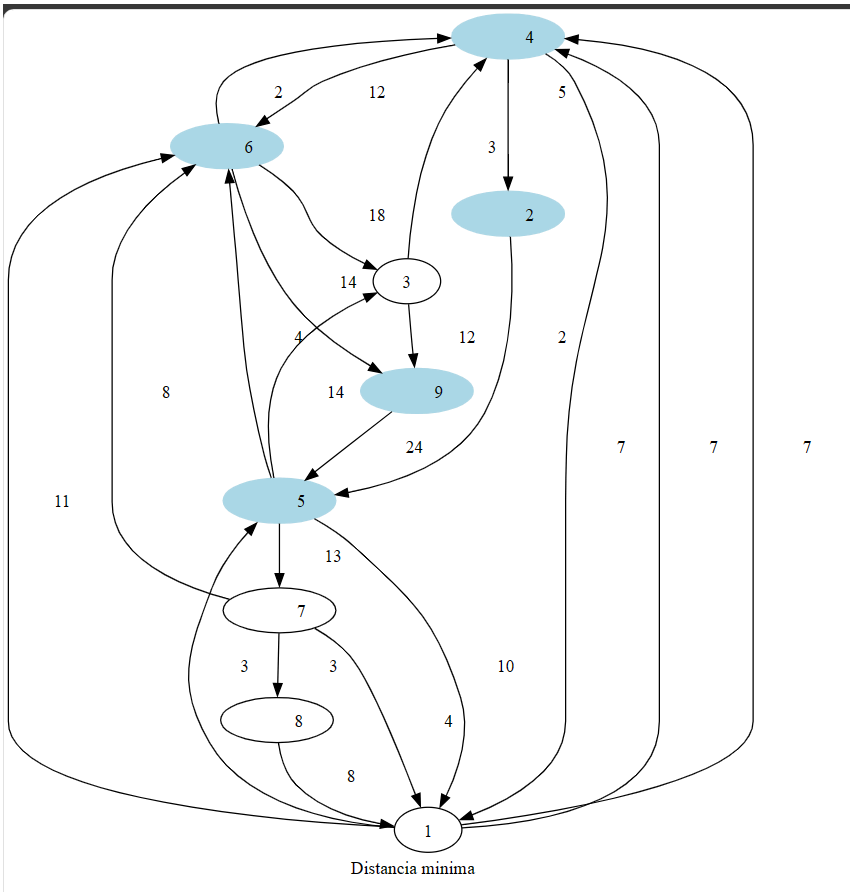
El algoritmo mantiene un conjunto de nodos no visitados y una lista de distancias mínimas conocidas desde el nodo de origen. En cada iteración, selecciona el nodo no visitado con la distancia mínima conocida, lo marca como visitado y actualiza las distancias mínimas de sus nodos vecinos si se encuentra una distancia más corta. Este proceso continúa hasta que se hayan visitado todos los nodos o se haya encontrado la distancia más corta al nodo de destino.

La subrutina ***graphWithColor*** visualiza el grafo con colores, resaltando los nodos que forman parte de la ruta más corta. Genera un archivo DOT que describe el grafo y luego utiliza la herramienta Graphviz para convertirlo en un archivo SVG. Los nodos que forman parte de la ruta más corta se llenan con un color especial para resaltarlos, lo que facilita su identificación visual.

```
299         subroutine graphWithColor(self, color_array)
300             class(adyacencia), intent(in) :: self
301             integer, dimension(:), intent(in) :: color_array ! azul
302             integer :: io, i
303             character(len=100) :: command
304             type(node_sucursal), pointer :: current_node
305             type(sub_node), pointer :: current_subnode, subsub
306 
307             !command = "dot -Tpng ./distancia_min.dot -o ./distancia_min.png"
308             io = 1
309             !print *, "color_array: ", color_array
310             ! Open DOT file
311             open(newunit=io, file="./distancia_min.dot")
312             write(io, *) "digraph G {"
313             write(io, *) "label=""Distancia minima"""
314             ! Traverse
315             current_node => self%head
316             do while(associated(current_node))
317                 ! si esta en el array azul
318                 !print *, current_node%salida
319                 if (any(current_node%salida == color_array )) then
320                     write(io, *) current_node%salida, "[label = """, current_node%salida, """, style=filled, color = lightblue]"
321                 else
322                     subsub => current_node%list
323                     do while(associated(subsub))
324                         if (any(subsub%llega == color_array)) then
325                             write(io, *) subsub%llega, "[label = """, subsub%llega, """, style=filled, color = lightblue]"
326                         else
327                             write(io, *) current_node%salida, "[label = """, current_node%salida, """]"
328                         end if
329                         subsub => subsub%next
330                     end do
331                 end if
```

Finalmente, se muestra la imagen generada con la ruta más corta marcada.

En la visualización con colores, los nodos que forman parte de la ruta más corta se resaltan en azul claro para distinguirlos del resto de los nodos. Esto ayuda a identificar rápidamente la ruta óptima y proporciona una representación visual clara de la solución encontrada por el algoritmo de Dijkstra. La imagen generada se guarda en formato SVG para su fácil visualización y posterior análisis.



También para mayor seguridad de los datos se implementó una lista de blockchain, en la cual se almacenan las rutas generadas, para poder implementar esto es necesario tener un árbol de merkle que permita conocer la raíz de dicha ruta, es por eso que se implementó un arbol de merkle con la encriptación de los datos con la función sha256. La función sha256 opera mediante un proceso de compresión de bloques, donde los datos de entrada se dividen en bloques de 512 bits y se

procesan en una serie de rondas para generar el hash final. Durante cada ronda, se aplican una serie de operaciones no lineales y de mezcla que aseguran la aleatoriedad y la distribución uniforme de los bits en el hash resultante.

El árbol de merkle se construye en base a la ruta calculada, y se aplica la encriptación sha256 a la cadena resultante de la sucursal de llegada, la dirección de la sucursal de llegada, la sucursal de salida, la dirección de la sucursal de salida y el costo entre esas sucursales.

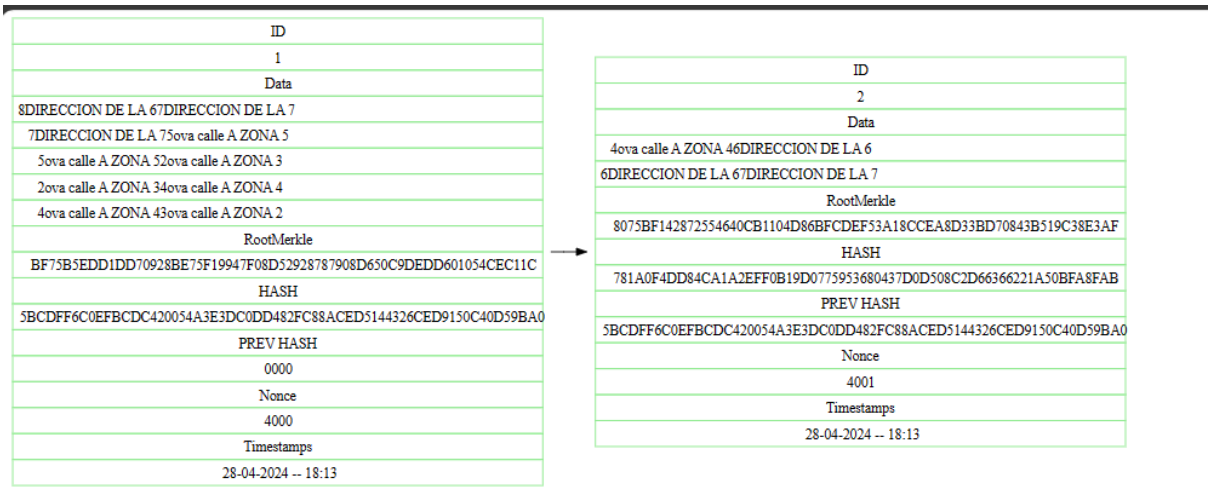
Posteriormente se permite visualizar una representación del árbol de merkle construido.

El arbol de merkle permite que podamos crear una blockchain, confiable haciendo uso de la raiz dek mismo y de los datos recopilados. Es posible crear una blockchain que permita tener un registro de todas las transacciones realizadas y posteriormente crear un archivo en donde se guarde cada bloque.


```
{
    "INDEX":2,
    "TIMESTAMP":"28-04-2024 -- 18:13",
    "NONCE": 4001,
    "DATA": [
        {
            "surcursal_0":8,
            "direccion_o":"8va avenida zona 15, Guatemala",
            "surcursal_d":8,
            "direccion_d":"zona 5 de villa nueva, Guatemala",
            "costo":5000
        },

        {
            "surcursal_0":2,
            "direccion_o":"10va avenida zona 15, Guatemala",
            "surcursal_d":9,
            "direccion_d":"zona 5 de mixco, Guatemala",
            "costo":5000
        }
    ],
    "PREVIOUSHASH":"A0D10BAD43D62815E947FFAD2FF575B29DB6C7BA8EB23B1A1EF5CDCE82F9C68D",
    "ROOKMRKLE": "7E3852D1F07830CC9D3B29EB04C7F9C619064DED4A5A0FE98F59267E7CEE1073",
    "HASH": "780E6FF4ADCDC751584E66640C903335D17522FD0C3A97B85A9D9DEACE323336"}

```


Este archivo JSON presenta una estructura organizada y detallada que describe transacciones entre las sucursales,, contiene información clave como el índice de la transacción, la marca de

tiempo y elementos de verificación como el nonce, el hash previo y el hash actual. La sección "DATA" es especialmente significativa, ya que contiene una matriz de objetos que representan las rutas calculadas y seleccionadas, así también como el costo de las mismas. C



ID
1
Data
8DIRECCION DE LA 67DIRECCION DE LA 7
7DIRECCION DE LA 75ova calle A ZONA 5
5ova calle A ZONA 52ova calle A ZONA 3
2ova calle A ZONA 34ova calle A ZONA 4
4ova calle A ZONA 43ova calle A ZONA 2
RootMerkle
BF75B5EDD1DD70928BE75F19947F08D5292878790 8D650C9DEDD601054CEC11C
HASH
5BCDFF6C0EFBCDC420054A3E3DC0DD482FC88ACED5144326CED9150C40D59BA0
PREV HASH
0000
Nonce
4000
Timestamps
28-04-2024 -- 18:13

ID
2
Data
4ova calle A ZONA 46DIRECCION DE LA 6
6DIRECCION DE LA 67DIRECCION DE LA 7
RootMerkle
8075BF142872554640CB1104D86BFCDEF53A18CCEA8D33BD70843B519C38E3AF
HASH
781A0F4DD84CA1A2EFF0B19D0775953680437D0D508C2D66366221A50BFA8FAB
PREV HASH
5BCDFF6C0EFBCDC420054A3E3DC0DD482FC88ACED5144326CED9150C40D59BA0
Nonce
4001
Timestamps
28-04-2024 -- 18:13