Aplicaciones Prácticas y Implementación de 2D Segment Trees: Un Estudio Detallado

1st Vasco Diaz Hurtado
Ciencia de la computación
Algoritmos y Estructuras de Datos
vasco.diaz@utec.edu.pe
202210119

2nd Denzel Bautista Rodriguez
Ciencia de la computación
Algoritmos y Estructuras de Datos
denzel.bautista@utec.edu.pe
202210461

***Abstract*—En este trabajo presentaremos un análisis detallado del 2D Segment Tree, una estructura de datos fundamental para el manejo eficiente de datos bidimensionales. Abordaremos su definición, métodos de construcción, complejidad algorítmica, operaciones fundamentales y una variedad de aplicaciones prácticas, incluyendo visualización de datos y análisis geoespacial.**

Mediante análisis comparativos, demostraremos la eficacia del 2D Segment Tree en la optimización de consultas y actualizaciones en matrices bidimensionales, resaltando su capacidad para gestionar eficientemente grandes volúmenes de información a través de una organización jerárquica. Este aspecto es esencial para comprender cómo el 2D Segment Tree facilita operaciones rápidas y eficientes, permitiendo la actualización dinámica y la consulta de datos en subrectángulos específicos. Además, discutiremos las ventajas y desventajas del 2D Segment Tree en comparación con otras estructuras de datos similares como el QuadTree y el K-d Tree. Este análisis no solo evidencia la importancia crítica del 2D Segment Tree en aplicaciones actuales, sino que también propone vías para futuras investigaciones, buscando expandir sus horizontes aplicativos y mejorar su eficiencia operativa.

***Index Terms*—2D Segment Tree, estructuras de datos, consultas espaciales, visualización de datos, optimización de actualizaciones**

I. Introducción

El 2D Segment Tree es una estructura de datos avanzada que facilita la manipulación y consulta eficiente de datos en matrices bidimensionales. Esta herramienta extiende las capacidades del árbol de segmentos tradicional a dos dimensiones, abriendo un abanico de posibilidades en diversas áreas de la computación.

Entre sus aplicaciones más destacadas se encuentran el análisis y procesamiento avanzado de imágenes digitales, la optimización de consultas en entornos de bases de datos complejas, la gestión inteligente de espacios en gráficos por computadora y el manejo eficaz de datos geoespaciales.

La estructura jerárquica del árbol permite descomponer una matriz en submatrices manejables, mejorando significativamente el rendimiento en operaciones de consulta y actualización. Este informe se adentrará en los principios fundamentales del 2D Segment Tree, explorando su construcción, funcionamiento y las ventajas que ofrece en el tratamiento de datos multidimensionales.

II. Descripción del 2D Segment Tree

A. Definición

Un 2D Segment Tree es una estructura de datos avanzada utilizada para realizar consultas y actualizaciones eficientes en matrices bidimensionales. Es una extensión del Segment Tree unidimensional, que se utiliza comúnmente para manejar intervalos en un arreglo. En el caso de una matriz bidimensional, el 2D Segment Tree permite realizar operaciones como sumas de submatrices, actualizaciones de elementos individuales y consultas de rangos en tiempo logarítmico en ambas dimensiones (LibreIM, 2015).

B. Estructura del 2D Segment Tree

La estructura del 2D Segment Tree es jerárquica y se construye en dos niveles:

1. Nivel de Filas:

- Primero, se construyen Segment Trees unidimensionales (Segment Trees) para cada fila de la matriz original. Cada Segment Tree de fila maneja los elementos de estas organizados como un array y permite consultas y actualizaciones dentro de las mismas.

2. Nivel de Columnas:

- Luego, se construye un Segment Tree bidimensional (2D Segment Tree) sobre los Segment Trees de filas. Este 2D Segment Tree de columnas permite consultas y actualizaciones a nivel de columnas, combinando los resultados de los Segment Trees de filas para manejar submatrices completas.

Cada nodo en el 2D Segment Tree representa un subrectángulo de la matriz original. La raíz del árbol representa toda la matriz, y cada nodo hijo representa un subrectángulo correspondiente a una subdivisión de su nodo padre. Esta división continúa hasta llegar a subrectángulos de tamaño unitario (elementos individuales de la matriz).

C. Restricciones y Tipos de Datos

No todos los tipos de datos pueden utilizarse para construir un 2D Segment Tree. Los tipos de datos deben cumplir ciertas condiciones para que las operaciones de combinación (como la suma) sean válidas y eficientes.

Tipos de Datos Adecuados:

- **Números Enteros**: Son los más comunes y fáciles de manejar, permitiendo operaciones como sumas, máximos, mínimos, etc.
- **Números de Punto Flotante**: También se pueden usar, aunque se debe tener cuidado con la precisión en operaciones de suma.
- **Booleanos**: Se pueden utilizar para operaciones lógicas, como consultas de "todos verdaderos" en un subrectángulo.

Tipos de Datos No Adecuados:

- **Cadenas de Texto**: No son adecuadas para un 2D Segment Tree porque las operaciones de combinación (como la suma) no tienen un significado claro.
- **Estructuras Complejas**: Tipos de datos que no soportan operaciones de combinación simples y eficientes.

III. Operaciones Fundamentales

En esta sección, describiremos más a detalle las operaciones fundamentales del 2D Segment Tree: construcción, actualización y consulta. Cabe resaltar que un Segment Tree está diseñado para manejar dichas operaciones en intervalos de un arreglo, como encontrar la suma, el máximo, el mínimo, etc., en un rango específico. No está diseñado para cambios en la estructura del arreglo, como la eliminación o inserción de elementos, que alterarían la longitud y los índices del arreglo.

A. Construcción

La construcción del 2D Segment Tree implica la creación de un árbol de Segment Trees unidimensionales. El proceso comienza construyendo un Segment Tree para cada fila de la matriz original y luego construyendo un Segment Tree de estos Segment Trees.

B. Actualización

La operación de actualización en un 2D Segment Tree modifica un elemento específico en la matriz y actualiza los nodos correspondientes en el árbol de segmentos bidimensional. Este proceso implica actualizar los Segment Trees en ambas dimensiones (filas y columnas) para mantener la integridad de los datos y asegurar que futuras consultas devuelvan resultados correctos.

C. Consulta

La operación de consulta en un 2D Segment Tree se utiliza para recuperar información sobre un subrectángulo de la matriz. Esto se hace combinando los resultados de los Segment Trees que cubren el rango consultado, lo que permite obtener rápidamente la suma de los elementos en un subrectángulo específico.

IV. Otras Operaciones y Algoritmos

En esta sección, exploraremos procedimientos adicionales que expanden la funcionalidad del 2D Segment Tree más allá de las operaciones básicas de construcción, actualización y consulta de suma. Estas operaciones adicionales aprovechan la estructura única del 2D Segment Tree para realizar consultas y manipulaciones de datos bidimensionales de manera eficiente.

A. Consulta de Mínimo y Máximo

Además de las consultas de suma, el 2D Segment Tree se puede adaptar para realizar consultas de mínimo y máximo en subrectángulos. Estas operaciones son útiles en aplicaciones que requieren encontrar los valores extremos dentro de una región específica de la matriz.

- **Consulta de Mínimo**: Para realizar una consulta de mínimo, cada nodo del Segment Tree debe almacenar el valor mínimo de su subrectángulo correspondiente. Durante la consulta, se combinan los valores mínimos de los nodos relevantes para obtener el mínimo total del subrectángulo consultado.
- **Consulta de Máximo**: De manera similar, para una consulta de máximo, cada nodo almacena el valor máximo de su subrectángulo. La combinación de estos valores durante la consulta proporciona el máximo total del subrectángulo consultado.

B. Producto de Elementos

El 2D Segment Tree también se puede utilizar para calcular el producto de los elementos en un subrectángulo. Esta operación es útil en aplicaciones donde se requiere el producto de varios elementos dentro de una región específica de la matriz.

- **Cálculo del Producto**: Cada nodo del Segment Tree almacena el producto de los elementos en su subrectángulo. Durante una consulta, se combinan los productos de los nodos relevantes para obtener el producto total del subrectángulo.

C. GCD (Máximo Común Divisor) y LCM (Mínimo Común Múltiplo)

El 2D Segment Tree puede adaptarse para calcular el máximo común divisor (GCD) y el mínimo común múltiplo (LCM) de los elementos en un subrectángulo. Estas operaciones son útiles en aplicaciones que requieren manipulación de datos numéricos.

- **Cálculo de GCD**: Cada nodo del Segment Tree almacena el GCD de los elementos en su subrectángulo. Durante una consulta, se combinan los GCD de los nodos relevantes para obtener el GCD total del subrectángulo.
- **Cálculo de LCM**: De manera similar, cada nodo del Segment Tree almacena el LCM de los elementos en su subrectángulo. La combinación de estos LCM durante la consulta proporciona el LCM total del subrectángulo.

D. Operaciones Booleanas

El 2D Segment Tree también puede adaptarse para realizar operaciones booleanas como AND, OR y XOR en subrectángulos. Estas operaciones son útiles en aplicaciones que requieren combinaciones lógicas de datos en regiones específicas de la matriz.

- **Operación AND**: Cada nodo del Segment Tree almacena el resultado de la operación AND de su subrectángulo. Durante la consulta, se combinan los resultados de los nodos relevantes utilizando la operación AND.

- **Operación OR**: De manera similar, cada nodo almacena el resultado de la operación OR de su subrectángulo, y los resultados se combinan durante la consulta utilizando la operación OR.
- **Operación XOR**: Para la operación XOR, cada nodo almacena el resultado de la operación XOR de su subrectángulo. La combinación de estos resultados durante la consulta proporciona el resultado XOR total del subrectángulo consultado.

Estas operaciones adicionales amplían la versatilidad del 2D Segment Tree en aplicaciones prácticas, permitiendo un manejo más dinámico y flexible de los datos bidimensionales. Cada una de estas operaciones aprovecha la estructura jerárquica del 2D Segment Tree para realizar consultas y actualizaciones de manera eficiente.

V. Pruebas y Ejemplos de Uso

A. Construcción del 2D Segment Tree

Para ilustrar el proceso de construcción del 2D Segment Tree, consideremos una matriz de ejemplo de 4×4:

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix}$$

El 2D Segment Tree se construye en dos fases: primero construimos los Segment Trees para cada fila y luego construimos el Segment Tree para las columnas.

Construcción de Segment Trees de Filas:

Para cada fila de la matriz, construimos un Segment Tree unidimensional. Por ejemplo, para la primera fila $[1, 2, 3, 4]$, el Segment Tree se construye de la siguiente manera:

- Nivel de hojas: $[1, 2, 3, 4]$
- Nivel intermedio: $[1 + 2, 3 + 4] = [3, 7]$
- Raíz: $3 + 7 = 10$

El Segment Tree completo para la primera fila es:

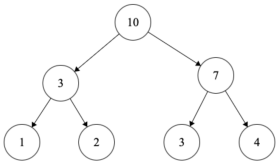

Fig. 1. Segment Tree para la primera fila de la matriz.

Realizamos este proceso para cada fila de la matriz original.

Construcción del Segment Tree de Columnas:

Después de construir los Segment Trees para cada fila, se combinan los resultados para construir el Segment Tree de columnas. Consideremos las primeras dos filas:

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{matrix}$$

Los Segment Trees de estas filas son:

Para combinar estas filas en el Segment Tree de columnas:

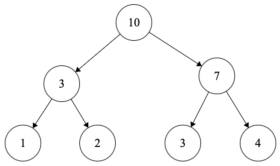

Fig. 2. Segment Tree para la primera fila de la matriz.

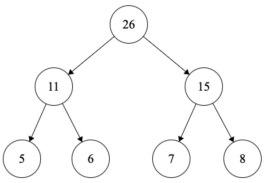

Fig. 3. Segment Tree para la primera fila de la matriz.

- Sumamos las raíces: $10 + 26 = 36$
- Sumamos los nodos intermedios: $[3 + 11, 7 + 15] = [14, 22]$
- Sumamos las hojas correspondientes: $[1 + 5, 2 + 6, 3 + 7, 4 + 8] = [6, 8, 10, 12]$

El Segment Tree de columnas para las primeras dos filas es:

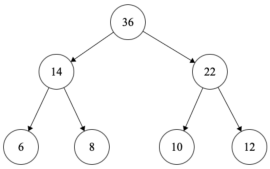

Fig. 4. Segment Tree para las primeras dos filas combinadas.

Este proceso se repite para todas las combinaciones de filas y columnas hasta que se construya el Segment Tree completo.

B. Operaciones de Consulta y Actualización

Una vez construido el 2D Segment Tree, podemos realizar consultas y actualizaciones de manera eficiente.

Consulta de Suma en un Subrectángulo:

Consideremos una consulta para encontrar la suma de los elementos en el subrectángulo definido por las esquinas superiores izquierdas $(1, 1)$ y las esquinas inferiores derechas $(2, 2)$. El subrectángulo es:

$$\begin{matrix} 6 & 7 \\ 10 & 11 \end{matrix}$$

La suma de los elementos en este subrectángulo es $6 + 7 + 10 + 11 = 34$.

Para realizar esta consulta en el 2D Segment Tree, seguimos estos pasos:

- Navegamos a través de los nodos relevantes en el Segment Tree de columnas.

- Para cada nodo, realizamos una consulta en el Segment Tree de filas correspondiente.
- Combinamos los resultados de las consultas parciales para obtener la suma total.

Actualización de un Elemento:

Consideremos que queremos actualizar el elemento en la posición $(1, 1)$ de la matriz original a 20. La matriz actualizada es:

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 20 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix}$$

Para realizar esta actualización en el 2D Segment Tree, seguimos estos pasos:

- Actualizamos el nodo hoja correspondiente en el Segment Tree de filas.
- Propagamos el cambio hacia arriba en el Segment Tree de filas, actualizando los nodos internos.
- Actualizamos el Segment Tree de columnas, propagando el cambio a través de los nodos relevantes.

Estos pasos aseguran que todas las consultas futuras reflejen correctamente el valor actualizado.

C. Casos de Prueba

Para validar nuestra implementación del , consideramos los siguientes casos de prueba:

Caso de Prueba 1: Consulta de Suma en Subrectángulo

Matriz Original:

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix}$$

Consulta: Suma de elementos en el subrectángulo definido por $(0, 0)$ a $(1, 1)$.

Resultado Esperado: $1 + 2 + 5 + 6 = 14$

Caso de Prueba 2: Actualización de Elemento y Consulta Posterior

Actualización: Actualizar el elemento en $(2, 2)$ a 20.

Matriz Actualizada:

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 20 & 12 \\ 13 & 14 & 15 & 16 \end{matrix}$$

Consulta: Suma de elementos en el subrectángulo definido por $(1, 1)$ a $(2, 2)$.

Resultado Esperado: $6 + 7 + 10 + 20 = 43$

Estos casos de prueba demuestran la correctitud y eficiencia de nuestra implementación del 2D Segment Tree para consultas de suma y actualizaciones de elementos.

VI. Complejidades

La complejidad de los árboles de segmentos 2D es un aspecto crucial para entender su eficiencia en el manejo de consultas y actualizaciones en un contexto bidimensional. Según Zhang (2020), la creación del árbol de segmentos tiene una complejidad de espacial de O(NM), donde cada árbol de segmentos de nodo toma O(M) de espacio para construirse y habrá un total de O(N) nodos en nuestro árbol. O dicho de otra forma, N es el número de filas y M es el número de columnas de la matriz original. En las siguientes tablas se encuentran las complejidades temporales y espaciales de las operaciones mencionadas anteriormente. La mayoría de las operaciones en un 2D Segment Tree tienen una complejidad temporal de $O(\log N \cdot \log M)$ debido a su estructura jerárquica. Al realizar operaciones como construcción, actualización o consulta, el algoritmo navega a través de los niveles del árbol, optimizando el tiempo de ejecución.

TABLE I
Complejidades Temporales del 2D Segment Tree

Operación	Complejidad Temporal
Construcción	$O(N \cdot M \cdot \log N \cdot \log M)$
Actualización	$O(\log N \cdot \log M)$
Consulta	$O(\log N \cdot \log M)$
Mínimo/Máximo	$O(\log N \cdot \log M)$
Producto	$O(\log N \cdot \log M)$
GCD/LCM	$O(\log N \cdot \log M)$
Operaciones Booleanas	$O(\log N \cdot \log M)$

TABLE II
Complejidades Espaciales del 2D Segment Tree

Operación	Complejidad Espacial
Construcción	$O(N \cdot M)$
Actualización	$O(1)$
Consulta	$O(1)$
Mínimo/Máximo	$O(N \cdot M)$
Producto	$O(N \cdot M)$
GCD/LCM	$O(N \cdot M)$
Operaciones Booleanas	$O(N \cdot M)$

VII. Comparación con Otras Estructuras

En esta sección, comparamos el 2D Segment Tree con otras estructuras de datos similares utilizadas para manejar datos bidimensionales: el QuadTree y el K-d Tree.

A. QuadTree

El QuadTree es una estructura de datos jerárquica que divide el espacio en cuadrantes, donde cada nodo tiene hasta cuatro hijos. Es útil para operaciones espaciales como la búsqueda por rango y la detección de colisiones. "Dividimos el espacio bidimensional actual en cuatro cajas. Si una caja contiene uno o más puntos, creamos un objeto hijo que almacena el espacio bidimensional de la caja" (Geeks for Geeks, 2024).

El QuadTree se utiliza en aplicaciones como la compresión de imágenes y la búsqueda del punto más cercano en un área bidimensional. Sin embargo, "la estructura puede necesitar ser reequilibrada con frecuencia" (Geeks for Geeks, 2024), lo que

lo hace menos eficiente para actualizaciones dinámicas en los datos.

Por otro lado, el 2D Segment Tree permite realizar consultas y actualizaciones eficientes en una matriz bidimensional. La construcción de un 2D Segment Tree tiene una complejidad de $O(N \cdot M \cdot \log N \cdot \log M)$ y las operaciones de actualización y consulta tienen una complejidad de $O(\log N \cdot \log M)$. Esto lo hace ideal para problemas que requieren múltiples consultas y actualizaciones sin necesidad de reequilibrar la estructura.

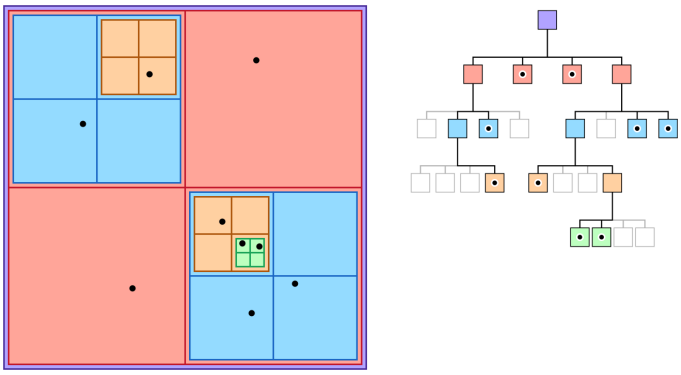

Fig. 5. Ejemplo de un QuadTree

B. K-d Tree

El K-d Tree es una estructura de datos utilizada para organizar puntos en un espacio k-dimensional. Es especialmente eficiente para operaciones como la búsqueda de vecinos más cercanos y la búsqueda por rango. "Un K-d Tree es un árbol de búsqueda binario en el que cada nodo es un punto en el espacio k-dimensional" (Baeldung, 2024).

Las operaciones de inserción y eliminación en un K-d Tree pueden ser complejas. "La inserción en un K-d Tree requiere dividir el espacio en hiperplanos alternativos y puede llevar un tiempo de $O(\log N)$ en promedio" (Baeldung, 2024). Sin embargo, la eliminación es más complicada debido a la necesidad de reestructurar el árbol, lo que puede afectar su eficiencia.

El 2D Segment Tree, como ya se ha discutido previamente, ofrece una estructura más sencilla para actualizaciones y consultas en datos bidimensionales, con una complejidad de $O(\log N \cdot \log M)$ para ambas operaciones.

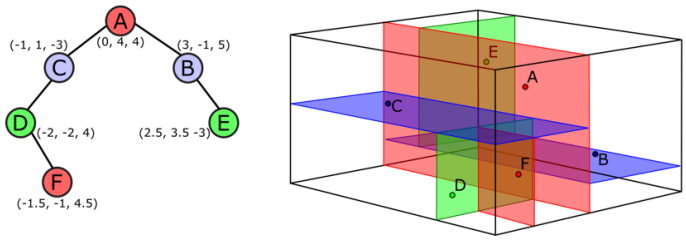

Fig. 6. Ejemplo de un K-d Tree

El 2D Segment Tree se diferencia de estas estructuras en su capacidad para manejar eficientemente actualizaciones y consultas en matrices bidimensionales, manteniendo una complejidad logarítmica en ambas operaciones. Esto lo hace especialmente adecuado para aplicaciones que requieren modificaciones frecuentes de los datos y consultas rápidas.

VIII. Componente Interesante: Implementación y Análisis de Estructuras de Datos en 2D

En esta sección, presentamos la implementación detallada de un 2D Segment Tree en C++, así como las implementaciones de QuadTree y KD-Tree. Además, realizamos pruebas de rendimiento para evaluar la eficiencia de cada estructura de datos en términos de tiempo de construcción y consulta. Finalmente, comparamos los resultados obtenidos con los benchmarks teóricos y analizamos las diferencias observadas teóricamente.

A. Construcción

Algorithm 1 Construcción del 2D Segment Tree

function BUILD($data$)
  $n \leftarrow data.rows$
  $m \leftarrow data.columns$
  $tree \leftarrow 2D\_array(4 * n, 4 * m)$
  BUILDX(1, 0, n - 1)
end function
function BUILDY(vx, lx, rx, vy, ly, ry)
  if $ly = ry$ **then**
    if $lx = rx$ **then**
      $tree[vx][vy] \leftarrow data[lx][ly]$
    else
      $tree[vx][vy] \leftarrow tree[vx*2][vy] + tree[vx*2+1][vy]$
    end if
  else
    $my \leftarrow (ly + ry)/2$
    BUILDY($vx, lx, rx, vy * 2, ly, my$)
    BUILDY($vx, lx, rx, vy * 2 + 1, my + 1, ry$)
    $tree[vx][vy] \leftarrow tree[vx][vy*2] + tree[vx][vy*2+1]$
  end if
end function
function BUILDX(vx, lx, rx)
  if $lx \neq rx$ **then**
    $mx \leftarrow (lx + rx)/2$
    BUILDX($vx * 2, lx, mx$)
    BUILDX($vx * 2 + 1, mx + 1, rx$)
  end if
  BUILDY($vx, lx, rx, 1, 0, m - 1$)
end function

B. Actualizacion

Algorithm 2 Actualización del 2D Segment Tree

```
1: function UPDATE(x, y, new_val)
2:     UPDATEX(1, 0, n - 1, x, y, new_val)
3: end function
4: function UPDATEY(vx, lx, rx, vy, ly, ry, x, y, new_val)
5:     if ly = ry then
6:         if lx = rx then
7:             tree[vx][vy] ← new_val
8:         else
9:             tree[vx][vy] ← tree[vx∗2][vy]+tree[vx∗2+
   1][vy]
10:        end if
11:    else
12:        my ← (ly + ry)/2
13:        if y ≤ my then
14:            UPDATEY(vx, lx, rx, vy ∗
   2, ly, my, x, y, new_val)
15:        else
16:            UPDATEY(vx, lx, rx, vy ∗ 2 + 1, my +
   1, ry, x, y, new_val)
17:        end if
18:        tree[vx][vy] ← tree[vx][vy∗2]+tree[vx][vy∗2+
   1]
19:    end if
20: end function
21: function UPDATEX(vx, lx, rx, x, y, new_val)
22:    if lx ≠ rx then
23:        mx ← (lx + rx)/2
24:        if x ≤ mx then
25:            UPDATEX(vx ∗ 2, lx, mx, x, y, new_val)
26:        else
27:            UPDATEX(vx ∗ 2 + 1, mx +
   1, rx, x, y, new_val)
28:        end if
29:    end if
30:    UPDATEY(vx, lx, rx, 1, 0, m − 1, x, y, new_val)
31: end function
```

Pasos para la Actualización:

1. Identificar el Nodo Hoja: Primero, localizamos el nodo hoja que corresponde al elemento que deseamos actualizar. Esto se hace ajustando los índices de la fila y la columna para acceder al nodo correcto en la estructura del árbol.

2. Actualizar el Nodo Hoja: Modificamos el valor del nodo hoja con el nuevo valor proporcionado.

3. Actualizar los Nodos de la Fila: Luego, recorremos hacia arriba desde el nodo hoja hasta la raíz, actualizando los nodos en la fila. Cada nodo se actualiza combinando los valores de sus dos nodos hijos.

4. Actualizar los Nodos de la Columna: Finalmente, recorremos hacia arriba desde la fila modificada, actualizando los nodos en las columnas correspondientes. Cada nodo se actualiza combinando los valores de los nodos hijos de las filas inferiores.

C. Consulta

Algorithm 3 Consulta del 2D Segment Tree

```
1: function SUM(lx, rx, ly, ry)
2:     return SUMX(1, 0, n - 1, lx, rx, ly, ry)
3: end function
4: function SUMY(vx, vy, tly, try, ly, ry)
5:     if ly > ry then
6:         return 0
7:     end if
8:     if ly = tly and ry = try then
9:         return tree[vx][vy]
10:    end if
11:    tmy ← (tly + try)/2
12:    return SUMY(vx, vy ∗ 2, tly, tmy, ly, min(ry, tmy))
   + SUMY(vx, vy∗2+1, tmy+1, try, max(ly, tmy+1), ry)
13: end function
14: function SUMX(vx, tlx, trx, lx, rx, ly, ry)
15:    if lx > rx then
16:        return 0
17:    end if
18:    if lx = tlx and rx = trx then
19:        return SUMY(vx, 1, 0, m − 1, ly, ry)
20:    end if
21:    tmx ← (tlx + trx)/2
22:    return SUMX(vx ∗
   2, tlx, tmx, lx, min(rx, tmx), ly, ry) + SUMX(vx ∗
   2 + 1, tmx + 1, trx, max(lx, tmx + 1), rx, ly, ry)
23: end function
```

Pasos para la Consulta:

1. Inicialización de Variables: Inicialmente, establecemos una variable para almacenar el resultado de la consulta. Esta variable se irá actualizando a medida que se recorran los nodos relevantes del árbol.

2. Ajuste de Coordenadas: Ajustamos las coordenadas del subrectángulo que queremos consultar para que se correspondan con los nodos hoja en el árbol. Esto se hace sumando el tamaño de la matriz original a las coordenadas iniciales de la consulta.

3. Recorrido en el Nivel de Filas: Recorrremos el Segment Tree en el nivel de filas, considerando las filas que caen dentro del rango consultado. Durante este recorrido, si encontramos una fila que es un nodo hoja y cae dentro del rango, realizamos una consulta sobre las columnas en esa fila.

4. Recorrido en el Nivel de Columnas: Para cada fila que cae dentro del rango, realizamos un recorrido similar en el nivel de columnas. Esto implica considerar las columnas que caen dentro del rango consultado y combinar sus valores.

5. Combinación de Resultados: Los resultados de las consultas a nivel de filas y columnas se combinan para obtener el resultado final de la consulta. Este resultado representa la suma de los elementos en el subrectángulo especificado.

D. Complejidad de Construcción

2D Segment Tree (para suma)

- **Construcción**: $O(N \cdot M \cdot \log N \cdot \log M)$
- Donde N y M representan las dimensiones de la matriz. La construcción implica crear un árbol de segmentos para cada nivel de la matriz.

KD-Tree

- **Construcción**: $O(n \log n)$ en promedio
- Donde n es el número de puntos en el espacio 2D. La construcción implica ordenar los puntos y dividirlos recursivamente.

QuadTree

- **Construcción**: $O(n \log n)$ en promedio
- Similar al KD-Tree, el QuadTree divide recursivamente el espacio en cuadrantes, siendo más eficiente en datos distribuidos uniformemente.

E. Complejidad de Consultas

2D Segment Tree (para suma)

- **Consulta de rango**: $O(\log N \cdot \log M)$
- La consulta de rango en una matriz de $N \times M$ se realiza en $O(\log N \cdot \log M)$.

KD-Tree

- **Consulta de rango**: $O(n^{1-1/d} + k)$ en promedio
- Para 2D ($d = 2$), esto se simplifica a $O(\sqrt{n} + k)$, donde k es el número de puntos en el rango de consulta. En el peor caso, la complejidad puede ser $O(n)$ si el árbol está desbalanceado.

QuadTree

- **Consulta de rango**: $O(\sqrt{n} + k)$ en promedio
- Similar al KD-Tree, la eficiencia de las consultas en un QuadTree depende de la distribución de los puntos. En el peor caso, puede ser $O(n)$ si los datos están altamente concentrados en una región.

F. Comparación para Diferentes Tamaños de Datos

Tamaño de Datos	Estructura	Construcción	Consulta
10	2D Segment Tree	$O(10^2 \log^2 10) \approx O(1000)$	$O(\log^2 10) \approx O(4)$
	KD-Tree	$O(10 \log 10) \approx O(30)$	$O(\sqrt{10} + k) \approx O(3 + k)$
	QuadTree	$O(10 \log 10) \approx O(30)$	$O(\sqrt{10} + k) \approx O(3 + k)$
100	2D Segment Tree	$O(100^2 \log^2 100) \approx O(400000)$	$O(\log^2 100) \approx O(16)$
	KD-Tree	$O(100 \log 100) \approx O(200)$	$O(\sqrt{100} + k) \approx O(10 + k)$
	QuadTree	$O(100 \log 100) \approx O(200)$	$O(\sqrt{100} + k) \approx O(10 + k)$
1000	2D Segment Tree	$O(1000^2 \log^2 1000) \approx O(90000000)$	$O(\log^2 1000) \approx O(36)$
	KD-Tree	$O(1000 \log 1000) \approx O(3000)$	$O(\sqrt{1000} + k) \approx O(32 + k)$
	QuadTree	$O(1000 \log 1000) \approx O(3000)$	$O(\sqrt{1000} + k) \approx O(32 + k)$
10000	2D Segment Tree	$O(10000^2 \log^2 10000) \approx O(1600000000)$	$O(\log^2 10000) \approx O(64)$
	KD-Tree	$O(10000 \log 10000) \approx O(40000)$	$O(\sqrt{10000} + k) \approx O(100 + k)$
	QuadTree	$O(10000 \log 10000) \approx O(40000)$	$O(\sqrt{10000} + k) \approx O(100 + k)$
100000	2D Segment Tree	$O(100000^2 \log^2 100000) \approx O(25000000000)$	$O(\log^2 100000) \approx O(100)$
	KD-Tree	$O(100000 \log 100000) \approx O(500000)$	$O(\sqrt{100000} + k) \approx O(317 + k)$
	QuadTree	$O(100000 \log 100000) \approx O(500000)$	$O(\sqrt{100000} + k) \approx O(317 + k)$

Fig. 7. Comparación de Complejidades para Diferentes Tamaños de Datos

G. Resumen

- **2D Segment Tree**:
  - **Construcción**: Más costosa con $O(N \cdot M \cdot \log N \cdot \log M)$, especialmente para grandes cantidades de datos.
  - **Consulta**: Muy eficiente para consultas de rango con $O(\log N \cdot \log M)$.
- **KD-Tree**:
  - **Construcción**: Eficiente con $O(n \log n)$ en promedio, pero puede ser costosa en el peor caso.
  - **Consulta**: Eficiente con $O(\sqrt{n} + k)$ en promedio, pero puede ser $O(n)$ en el peor caso.
- **QuadTree**:
  - **Construcción**: Similar al KD-Tree con $O(n \log n)$ en promedio.
  - **Consulta**: Similar al KD-Tree con $O(\sqrt{n} + k)$ en promedio, y $O(n)$ en el peor caso.

REFERENCES

[1] LibreIM. (2015, July 17). Segment Tree. Recuperado de https://libreim.github.io/blog/2015/07/17/segment-tree/

[2] CP-Algorithms. (n.d.). Simple Segment Tree. Recuperado de https://cp-algorithms.com/data_structures/segment_tree.html#simple-2d-segment-tree

[3] Zhang, J. (2020). 2D Segment Trees. Recuperado de https://activities.tjhsst.edu/sct/lectures/2021/2021_05_07_2D_Segment_Trees.pdf

[4] Geeks for Geeks. (2024, Febrero 6). Quad Tree. Recuperado de https://www.geeksforgeeks.org/quad-tree/

[5] Baeldung. (2024). K-d Trees. Retrieved from https://www.baeldung.com/cs/k-d-trees

[6] Gaurav Sen. (2017, 19 de octubre). 2D Segment Tree - Data Structure and Algorithms https://www.youtube.com/watch?v=kKlZ9B3cS14. YouTube.

[7] Pepcoding. (2021, 12 de agosto). 2D Segment Tree — Range Query — Point Update https://www.youtube.com/watch?v=5dPkTfGkY5w. YouTube.

[8] StudySmarter. "Segment Tree: Structure, Python & Java Guide." *Computer Science Data Structures*. Recuperado de https://www.studysmarter.co.uk/explanations/computer-science/data-structures/segment-tree/