Estructuras de Datos

Ericka Araya-C20553

***Resumen*—En este estudio, hemos evaluado y comparado los tiempos de ejecución de diversas estructuras de datos: la lista enlazada simple, el árbol de búsqueda binaria, el árbol rojinegro y la tabla de dispersión. A través de experimentos detallados, analizamos los tiempos de ejecución para las operaciones de inserción y búsqueda, tanto en escenarios de inserción aleatoria como ordenada.**

***Palabras clave*—estructura de datos, lista enlazada simple, árboles de búsqueda binaria, tablas de dispersión, árboles rojinegros, inserción, búsqueda.**

I. Introducción

En este trabajo, se profundiza en el análisis de los experimentos llevados a cabo para observar los tiempos de ejecución de las funciones de inserción y búsqueda de las estructuras de datos de Lista Enlazada Simple, Árboles de Búsqueda Binaria, Árboles Rojinegros y Tablas de Dispersión.

La Lista Enlazada Simple utiliza nodos enlazados secuencialmente para almacenar elementos, insertando nuevos elementos al principio de la lista y buscando iterativamente elementos por clave.

El Árbol de Búsqueda Binaria organiza elementos de manera jerárquica, manteniendo la propiedad de orden para facilitar búsquedas eficientes y asegurando que el árbol esté balanceado de manera natural.

Los Árboles Rojinegros son una variante de los árboles binarios de búsqueda que garantizan un balanceo adicional, utilizando colores en los nodos para mantener la estructura balanceada y asegurar tiempos de búsqueda óptimos.

La Tabla de Dispersión utiliza una función hash para asignar claves a índices de una tabla, insertando elementos mediante el cálculo del índice y realizando búsquedas directas en la posición calculada, optimizando el acceso a los datos.

En las pruebas se realizaron dos experimentos para cada estructura, donde en una ocasión se insertó 1,000,000 de elementos con llaves seleccionadas aleatoriamente en el rango [0, 2n), y se realizó una búsqueda de 10,000 llaves. En la otra prueba se insertó de igual manera 1,000,000 de elementos, pero con llaves insertadas en orden de 0 a n-1.

El artículo se estructura en dos partes principales: la primera se centra en el análisis individual de cada estructura de datos, evaluando la consistencia de los tiempos de inserción y búsqueda y las variaciones en el promedio de duración para las dos pruebas (Aleatoriamente y Ordenadamente) realizadas de las funciones ya mencionadas.

La segunda parte se encarga de comparar los tiempos promedio de las búsquedas, tanto con inserción aleatoria como con inserción ordenada, de cada estructura de datos, con el objetivo de comprender cómo varían los tiempos promedio de búsqueda utilizando diferentes estructuras de datos. Más específicamente, esta comparación tiene como objetivo identificar fortalezas y debilidades de cada algoritmo en términos de tiempo de ejecución y eficiencia.

Además de comparar estructuras de datos, también se pueden identificar casos de uso específicos para cada una. Por ejemplo, la Lista Enlazada puede ser más eficiente en escenarios donde las inserciones frecuentes son primordiales, mientras que los Árboles Binarios de Búsqueda y los Árboles Rojinegros pueden ser más adecuados para aplicaciones que requieren búsquedas rápidas y eficientes en conjuntos de datos más grandes y ordenados. Las Tablas de Dispersión, por otro lado, destacan en escenarios donde se necesitan tiempos de acceso rápidos y predecibles a datos.

II. Metodología

Para llevar a cabo la evaluación de los tiempos de ejecución de las estructuras de datos, se siguió un enfoque metodológico estructurado que implicaba varias etapas bien definidas.

1. Implementación de las funciones: Se implementaron las funciones de inserción y búsqueda para las estructuras de datos de Lista Enlazada Simple, Árbol de Búsqueda Binaria, Árbol Rojinegro y Tabla de Dispersión como clases en un entorno de desarrollo en C++. Esto aseguró la reutilización y mantenimiento efectivos de las funciones. Cada función se diseñó específicamente para realizar las operaciones de inserción y búsqueda de nodos con llaves en las respectivas estructuras de datos.

2. Implementación del código de los algoritmos: El código de las funciones realizadas se basó en el pseudocódigo proporcionado en los libros “Data Structures and Algorithms” de Aho, Hopcroft y Ullman [1], así como en “Introduction to Algorithms” de Cormen, Leiserson, Rivest y Stein [2]. Se garantizó que la implementación de las funciones fuera fiel a los conceptos y técnicas descritas en estos libros.

3. Diseño del procedimiento experimental: Se diseñó un procedimiento experimental detallado para evaluar el tiempo de ejecución de las funciones de inserción y búsqueda, en ambas estructuras de datos. Este procedimiento incluyó la selección de conjuntos de datos que consistían en llaves aleatorias y llaves ordenadas. En cada experimento, se insertaron 1.000.000 de elementos en la estructura de datos y posteriormente se realizaron 10.000 búsquedas. Se aseguraron condiciones controladas para minimizar variaciones en los tiempos de ejecución debido a factores externos.

4. Registro y análisis de tiempos de ejecución: Se registraron los tiempos de cada ejecución de las funciones de inserción y búsqueda para los conjuntos de datos aleatorios y ordenados. Se calculó el tiempo promedio de ejecución para cada conjunto de datos y se organizó la información en tablas para facilitar su análisis. Además, se utilizaron técnicas de visualización de

datos para crear gráficos que representen los tiempos promedio, lo que permitió analizar con mayor claridad las diferencias entre los tiempos promedio de inserción y búsqueda para las dos estructuras de datos.

5. Análisis de resultados: Se llevó a cabo un análisis detallado de los resultados obtenidos, centrándose en identificar patrones y diferencias significativas en los tiempos de ejecución de las funciones de inserción y búsqueda. Se examinaron las barras y curvas producidas por las gráficas para comprender cómo variaban los tiempos de ejecución en función de la naturaleza de los datos insertados (aleatorios y ordenados).

III. Resultados

Los tiempos de ejecución de las corridas de todas las estructuras de datos se muestran en los cuadros I, II, III y IV.

Al analizar la estructura de datos de lista enlazada simple, observamos que su desempeño es bastante consistente tanto en las operaciones de inserción como en las de búsqueda. En el caso de la inserción, los tiempos son relativamente bajos, con una ligera diferencia entre inserción aleatoria y ordenada. Sin embargo, la verdadera disparidad se presenta en las operaciones de búsqueda. Aquí es donde la lista enlazada simple muestra una notable debilidad, especialmente en el caso de la búsqueda aleatoria, donde los tiempos son considerablemente más altos en comparación con la búsqueda ordenada. Este comportamiento sugiere que la estructura de lista enlazada simple puede no ser la más adecuada para aplicaciones que requieren un alto desempeño en operaciones de búsqueda, especialmente cuando se trata de búsquedas no secuenciales. Figuras 1, 2.

Por otro lado, al analizar el árbol de búsqueda binaria, observamos un desempeño más variado dependiendo del tipo de operación. En el caso de la inserción, los tiempos son generalmente más altos que los de la lista enlazada simple, especialmente para la inserción aleatoria. Sin embargo, para la inserción ordenada, el árbol de búsqueda binaria muestra un desempeño comparable, e incluso ligeramente mejor en algunas corridas. Donde realmente brilla el árbol de búsqueda binaria es en las operaciones de búsqueda. Tanto para búsquedas aleatorias como ordenadas, el árbol de búsqueda binaria presenta tiempos significativamente más bajos en comparación con la lista enlazada simple. Esto sugiere que el árbol de búsqueda binaria es una opción más eficiente en términos de búsqueda, especialmente para conjuntos de datos grandes o cuando la distribución de las claves no es conocida de antemano, donde hay un incremento gradual en el tiempo de ejecución, en línea con las expectativas del algoritmo. Figuras 3 y 4.

En cuanto al árbol rojinegro, los tiempos de inserción son más altos en comparación con la lista enlazada simple y el árbol de búsqueda binaria, debido a la necesidad de mantener el balance del árbol. La inserción aleatoria presenta tiempos más altos que la inserción ordenada, pero ambos son significativamente más altos que en las otras estructuras. Sin embargo, las búsquedas en el árbol rojinegro son eficientes, con tiempos

Cuadro I
TIEMPO DE EJECUCIÓN DE OPERACIONES PARA LA ESTRUCTURA DE DATOS DE LISTA ENLAZADA SIMPLE.

		Tiempo (ms)			
		Corrida			
Operación	Elementos	1	2	3	Prom
Inserción aleatoria	1000000	75	89	72	79
Inserción ordenada	1000000	46	45	45	45
Búsqueda aleatoria	10000	40594	40516	42809	41306
Búsqueda ordenada	10000	38569	38722	38755	38682

Cuadro II
TIEMPO DE EJECUCIÓN DE OPERACIONES PARA LA ESTRUCTURA DE DATOS DE ÁRBOL DE BÚSQUEDA BINARIA.

		Tiempo (ms)			
		Corrida			
Operación	Elementos	1	2	3	Prom
Inserción aleatoria	1000000	1088	1476	1572	1379
Inserción ordenada	1000000	90	89	85	88
Búsqueda aleatoria	10000	12	15	16	14
Búsqueda ordenada	10000	3	3	3	3

Cuadro III
TIEMPO DE EJECUCIÓN DE OPERACIONES PARA LA ESTRUCTURA DE DATOS DE ÁRBOL ROJINEGRO.

		Tiempo (ms)			
		Corrida			
Operación	Elementos	1	2	3	Prom
Inserción aleatoria	1000000	3186	4012	4255	3818
Inserción ordenada	1000000	1955	1527	1544	1675
Búsqueda aleatoria	10000	23	29	33	28
Búsqueda ordenada	10000	22	20	15	19

Cuadro IV
TIEMPO DE EJECUCIÓN DE OPERACIONES PARA LA ESTRUCTURA DE DATOS DE TABLA DE DISPERSIÓN.

		Tiempo (ms)			
		Corrida			
Operación	Elementos	1	2	3	Prom
Inserción aleatoria	1000000	928	554	568	683
Inserción ordenada	1000000	231	189	211	210
Búsqueda aleatoria	10000	10	7	7	8
Búsqueda ordenada	10000	7	7	6	7

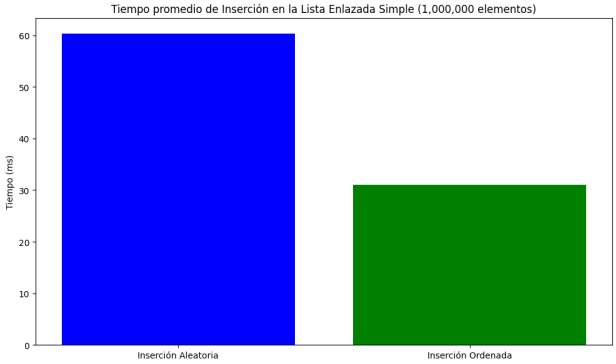

Figura 1. Tiempos promedio de inserción en la Lista Enlazada Simple.

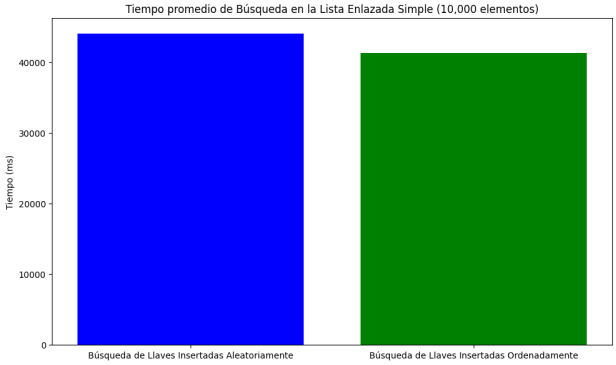

Figura 2. Tiempos promedio de búsqueda en la Lista Enlazada Simple.

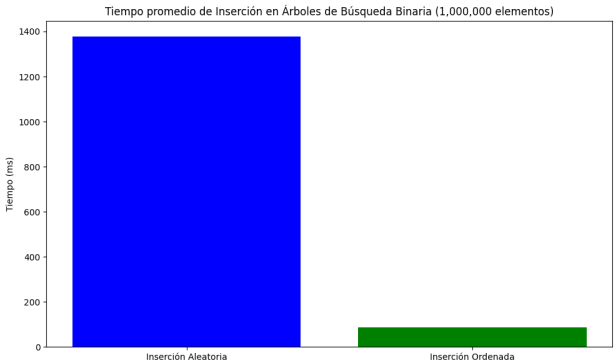

Figura 3. Tiempos promedio de inserción en el Árbol de Búsqueda Binaria.

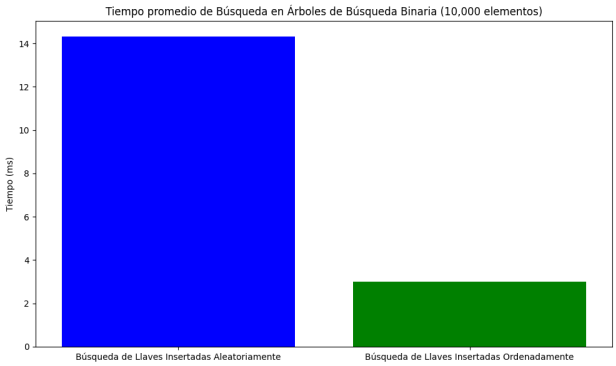

Figura 4. Tiempos promedio de búsqueda en el Árbol de Búsqueda Binaria.

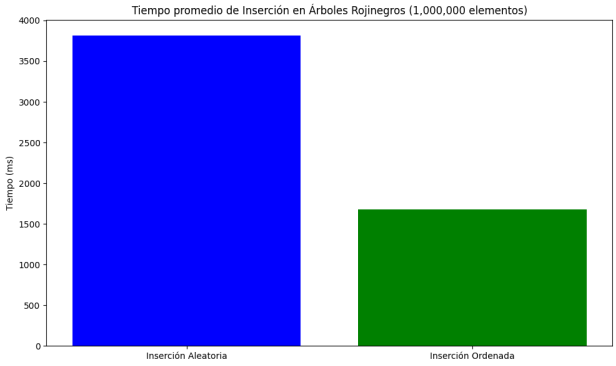

Figura 5. Tiempos promedio de inserción en el Árbol Rojinegro.

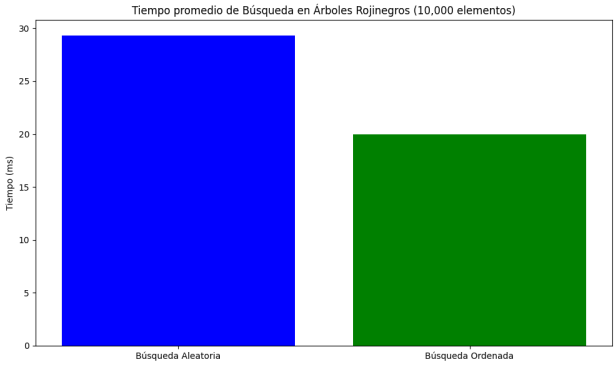

Figura 6. Tiempos promedio de búsqueda en el Árbol Rojinegro.

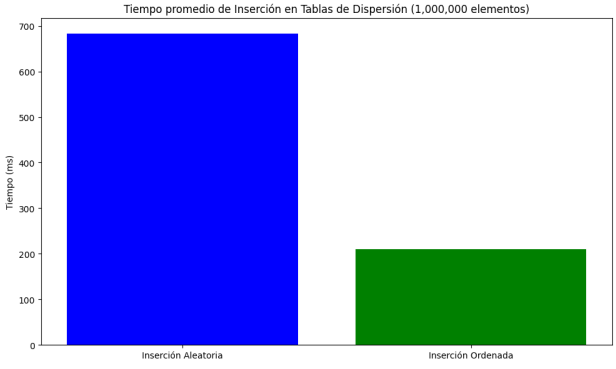

Figura 7. Tiempos promedio de inserción en la Tabla de Dispersión.

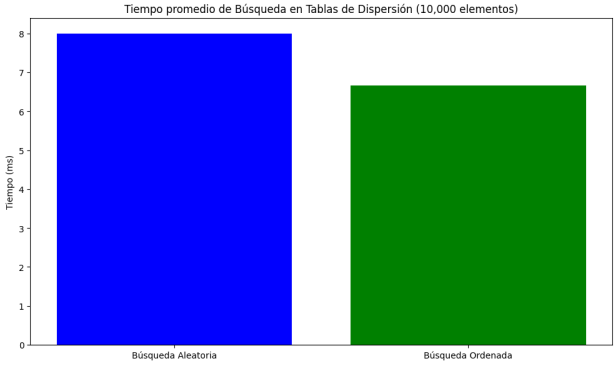

Figura 8. Tiempos promedio de búsqueda en la Tabla de Dispersión.

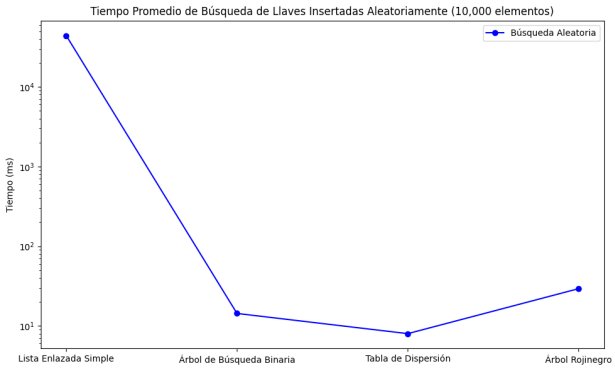

Figura 9. Gráfico comparativo de los tiempos promedio de búsqueda de llaves insertadas aleatoriamente.

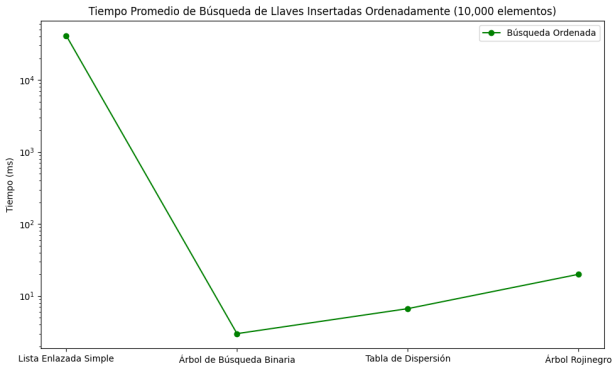

Figura 10. Gráfico comparativo de los tiempos promedio de búsqueda de llaves insertadas ordenadamente.

bajos tanto para datos aleatorios como ordenados, lo que demuestra la ventaja de mantener una estructura equilibrada para operaciones de búsqueda rápidas. Figuras 5 y 6.

Por último, la tabla de dispersión mostró un desempeño notablemente eficiente en las operaciones de búsqueda, tanto para datos aleatorios como ordenados, con tiempos muy bajos. Las inserciones también son rápidas, aunque hay una diferencia notable entre la inserción aleatoria y la ordenada, siendo la ordenada más eficiente. Esto destaca la tabla de dispersión como una estructura de datos altamente eficiente para aplicaciones que requieren operaciones rápidas de inserción y búsqueda. Figuras 7 y 8.

Al comparar los tiempos promedio de búsqueda aleatoria, las diferencias en eficiencia son notables. La lista enlazada simple muestra el mayor tiempo promedio con 41306 ms. Esto se debe a su naturaleza lineal, que requiere recorrer la lista completa en el peor de los casos. Por otro lado, el árbol de búsqueda binaria mejora significativamente, con un tiempo promedio de 14 ms gracias a su estructura que permite una búsqueda logarítmica. El árbol rojinegro, con un tiempo promedio de 28 ms, también ofrece una búsqueda eficiente, aunque ligeramente más alta debido al balance del árbol. La tabla de dispersión es la más eficiente para búsquedas aleatorias, con un tiempo promedio de 8 ms, facilitado por su acceso directo mediante funciones hash. Figura 9

Para la búsqueda ordenada, las diferencias en tiempos promedio también son evidentes. La lista enlazada simple muestra tiempos elevados de 38682 ms, aunque algo menores que en la búsqueda aleatoria debido a la secuencialidad. El árbol de búsqueda binaria mantiene una alta eficiencia con un tiempo promedio de 3 ms, beneficiándose de su estructura equilibrada. El árbol rojinegro presenta tiempos bajos con 19 ms, reflejando su capacidad de mantener el árbol balanceado. La tabla de dispersión sigue siendo muy eficiente, con un tiempo promedio de 7 ms, mostrando poca diferencia entre las búsquedas aleatorias y ordenadas. Figura 10

IV. Conclusiones

En conclusión, la lista enlazada simple no es adecuada para búsquedas debido a sus altos tiempos en comparación con otras estructuras, aunque es mejor en inserciones. El árbol de búsqueda binaria destaca por sus tiempos de búsqueda bajos y su estructura logarítmica. El árbol rojinegro proporciona un buen equilibrio entre tiempos de búsqueda e inserción, manejando bien los datos desbalanceados. La tabla de dispersión es la más eficiente en búsquedas, con tiempos consistentemente bajos, gracias a su acceso directo mediante funciones hash. La elección de la estructura de datos dependerá del contexto y necesidades específicas de la aplicación, pero para búsquedas rápidas, el árbol de búsqueda binaria, el árbol rojinegro y la tabla de dispersión son superiores a la lista enlazada simple.

Referencias

[1] A. Alfred V, H. John E, and U. Jeffrey D, *Data Structures and Algorithms*, 1st ed. Micheal A, Harrison, 1983.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. The MIT Press, 2009.

APÉNDICE A
CÓDIGO DE LOS ALGORITMOS

El código se muestra en los algoritmos 1, 2, 3 y 4.

Algoritmo 1 El algoritmo LIST-SEARCH busca de manera iterativa una clave k en una lista enlazada simple L. El algoritmo LIST-INSERT inserta un nuevo nodo x después de un nodo y en una lista enlazada simple. Actualiza los punteros del nodo y para enlazar correctamente el nuevo nodo x en la lista.

```
LIST-SEARCH (L, k)
    x = L.head
    while x != NIL and x.key != k
    x = x.next
    return x
LIST-INSERT (x, y)
    x.next = y.next
    if y.next != NIL
        y.next = x
    y.next = x
```

Algoritmo 2 El algoritmo ITERATIVE-TREE-SEARCH busca de manera iterativa una clave k en un árbol binario de búsqueda (BST). El algoritmo TREE-INSERT inserta de manera iterativa un nuevo nodo z en un árbol binario de búsqueda T, manteniendo la propiedad de orden del árbol. :

```
ITERATIVE-TREE-SEARCH (x, k)
    while x != NIL and k != x.key
        if k < x.key
            x = x.left
        else x = x.right
    return x
TREE-INSERT (T, z)
    x = T.root
    y = NIL
    while x != NIL
        y = x
        if z.key < x.key
            x = x.left
        else x = x.right
    z.p = y
    if y == NIL
    T.root = z
    elseif z.key < y.key
        y.left = z
    else y.right = z
```

Algoritmo 3 RB-INSERT-FIXUP ajusta los colores y realiza rotaciones para mantener las propiedades del árbol rojinegro después de la inserción de un nodo. RB-INSERT inserta un nodo manteniendo el orden del árbol y llama a RB-INSERT-FIXUP para el ajuste necesario. ITERATIVE-TREE-SEARCH realiza una búsqueda eficiente navegando desde la raíz hasta encontrar la clave buscada en un árbol binario de búsqueda. :

```
RB-INSERT-FIXUP (T,z)
while z.p.color == RED
    if z.p == zp.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                LEFT-ROTATE(T,z)
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)
    else
        // same as lines 3-15,
        // but with "right" and "left" exchanged
T.root .color = BLACK
RB-INSERT (T,z)
x = T.root
y = T.nil
while x != T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y = = T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP (T, z)
ITERATIVE-TREE-SEARCH (x, k)
    while x != nil and k != x.key
        if k < x.key
            x = x.left
        else x = x.right
    return x
```

Algoritmo 4 El método Insert inserta un nuevo valor k en una tabla de dispersión utilizando una función hash para determinar la posición. Search busca un valor k en la misma tabla de dispersión, devolviendo un puntero a la clave encontrada o nullptr si no está presente. La función HashFunction calcula el índice utilizando el operador módulo :

```
void Insert(const T& k) {
    int index = HashFunction(k);
    llnode<T>* newNode = new llnode<T>(k);
    table[index].Insert(newNode);
}
T* Search(const T& k) {
    int index = HashFunction(k);
    llnode<T>* foundNode = table[index].Search(k);
    if (foundNode != nullptr) {
        return &foundNode->getKey();
    } else {
        return nullptr;
    }
}
int HashFunction(const T& k) const {
    return k % size;
}
```