Estructuras de Datos: Listas Enlazadas

Cosijopii García

Universidad del Istmo

25 de noviembre de 2024

Contenido

Representación de una Lista Enlazada I

- Una **lista enlazada** es una colección de nodos donde cada nodo almacena un valor y un enlace al siguiente nodo en la lista.
- La lista comienza con un puntero llamado `head` que apunta al primer nodo.
- Si la lista está vacía, `head` es `NULL`.
- Ejemplo básico en C:

```c
struct Nodo {
    int dato;
    struct Nodo *siguiente;
};
struct Nodo* head = NULL;
```

- Cada nodo apunta al siguiente elemento, lo que permite el recorrido secuencial.

Representación de una Lista Enlazada II

- En comparación con arreglos, las listas enlazadas permiten un tamaño dinámico y operaciones de inserción/eliminación eficientes.

Tipos de Listas Enlazadas I

- Existen tres tipos principales de listas enlazadas:
  - **Lista Simplemente Enlazada**: Cada nodo tiene un puntero que apunta solo al siguiente nodo.
  - **Lista Doblemente Enlazada**: Cada nodo tiene un puntero al siguiente y al nodo anterior.
  - **Lista Circular**: El último nodo apunta al primer nodo, formando un ciclo.

Lista Simplemente Enlazada I

- En una **lista simplemente enlazada**, cada nodo tiene un enlace al siguiente nodo.
- Es eficiente en términos de memoria y permite agregar elementos al final o al principio.
- Ejemplo en C:

```c
void insertarInicio(int valor) {
    struct Nodo* nuevoNodo =
    malloc(sizeof(struct Nodo));
    nuevoNodo->dato = valor;
    nuevoNodo->siguiente = head;
    head = nuevoNodo;
}
```

- En el ejemplo, `insertarInicio` agrega un nodo al inicio de la lista.

Lista Doblemente Enlazada I

- En una **lista doblemente enlazada**, cada nodo apunta tanto al nodo siguiente como al nodo anterior.
- Permite recorridos en ambas direcciones, facilitando la eliminación de elementos de manera eficiente.
- Ejemplo en C:

```c
struct NodoDoble {
    int dato;
    struct NodoDoble *anterior;
    struct NodoDoble *siguiente;
};
```

- Se puede recorrer en ambas direcciones mediante los enlaces `anterior` y `siguiente`.
- Se usa comúnmente en aplicaciones como listas de reproducción y navegadores.

Lista Circular I

- En una **lista circular**, el último nodo apunta al primer nodo, formando un bucle continuo.
- Puede ser simplemente o doblemente enlazada.
- Ejemplo en C para una lista circular simplemente enlazada:

```
void insertarFinal(int valor) {
    struct Nodo* nuevoNodo = malloc(sizeof(struct N
    nuevoNodo->dato = valor;
    nuevoNodo->siguiente = head;
    if (head == NULL) {
        head = nuevoNodo;
        nuevoNodo->siguiente = head;
    } else {
        struct Nodo* temp = head;
        while (temp->siguiente != head) {
            temp = temp->siguiente;
```

Lista Circular II

```
            }
            temp->siguiente = nuevoNodo;
        }
    }
```

- Las listas circulares son útiles para aplicaciones que necesitan un bucle continuo, como la rotación de turnos.

Operaciones en una Lista Enlazada I

- Las operaciones principales en una lista enlazada incluyen:
  - **Inserción**: Agregar un nodo en el inicio, final o en una posición específica.
  - **Eliminación**: Remover un nodo del inicio, final o una posición específica.
  - **Búsqueda**: Encontrar un elemento específico en la lista.
  - **Actualización**: Modificar el valor de un nodo existente.
- Ejemplo de búsqueda en una lista simplemente enlazada:

```
int buscar(int valor) {
    struct Nodo* temp = head;
    while (temp != NULL) {
        if (temp->dato == valor) return 1;
        temp = temp->siguiente;
    }
    return 0;
}
```

Aplicaciones de las Listas Enlazadas I

- Las listas enlazadas tienen diversas aplicaciones en informática y programación:
  - **Implementación de otras estructuras de datos**: Como pilas y colas.
  - **Sistemas de gestión de memoria**: Como en los recolectores de basura.
  - **Listas de tareas y listas de reproducción**: Utilizadas para almacenar elementos de manera dinámica.
  - **Gestión de secuencias de datos**: Usadas en juegos, simulaciones y navegadores.
- Su flexibilidad y capacidad para crecer dinámicamente las hace adecuadas para muchas aplicaciones de datos complejos.

Algoritmos de Ordenación y Búsqueda

Cosijopii García

Universidad del Istmo

25 de noviembre de 2024

Introducción: Algoritmos de Ordenación y Búsqueda

- Los algoritmos de ordenación y búsqueda son fundamentales en computación.
- Permiten organizar y localizar datos de manera eficiente.
- Aplicaciones:
  - Bases de datos.
  - Procesamiento de datos en tiempo real.
  - Sistemas de recuperación de información.

Métodos de Ordenación

- La ordenación organiza elementos en un orden específico, generalmente ascendente o descendente.
- Dos enfoques comunes:
  - **Ordenación interna:** Los datos se ordenan en memoria principal.
  - **Ordenación externa:** Los datos se ordenan en almacenamiento externo.
- Evaluamos:
  - Eficiencia temporal (complejidad).
  - Eficiencia espacial.

Método de Ordenación: Burbuja

- Es un método sencillo basado en comparar y cambiar pares de elementos adyacentes.
- Repite el proceso hasta que no haya más intercambios.
- Tiempo de ejecución:
  - Mejor caso: $O(n)$ (lista ordenada).
  - Peor caso: $O(n^2)$ (lista inversamente ordenada).
- Desventaja: No es eficiente para listas grandes.

Pseudocódigo: Ordenación Burbuja

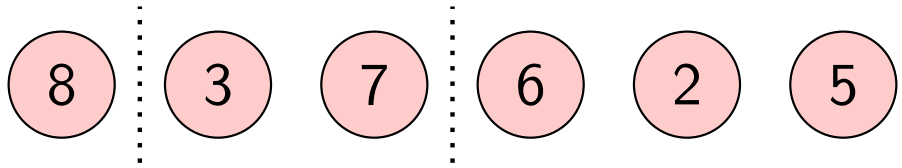
```
Para cada elemento i en la lista
    Para cada elemento j en la lista hasta n-i-1
        Si lista[j] > lista[j+1]
            Intercambiar lista[j] y lista[j+1]
    Fin para
Fin para
```

- El algoritmo es intuitivo pero no es eficiente para grandes conjuntos de datos.

Método de Ordenación: Shell

- El método Shell es una mejora del método de inserción.
- Ordena elementos que están a cierta distancia (gap) entre sí.
- Reduce gradualmente el gap hasta que sea 1.
- Tiempo de ejecución: $O(n^{3/2})$ en promedio.
- Ventaja: Es significativamente más rápido que Burbuja para listas grandes.

Ejemplo Visual: Ordenación Shell



Gap inicial

- Primero ordena por el gap inicial (por ejemplo, 3).
- Reduce el gap gradualmente y aplica inserción en sublistas.

Pseudocódigo: Ordenación Shell I

```
1. gap ← tamaño_lista / 2
2. Mientras gap > 0 hacer:
   3. Para cada elemento desde índice 'gap' hasta
   tamaño_lista - 1 hacer:
       4. elemento_actual ← lista[indice]
       5. j ← indice
       6. Mientras j >= gap y lista[j - gap] >
       elemento_actual hacer:
          7. lista[j] ← lista[j - gap]
          8. j ← j - gap
       Fin Mientras
       9. lista[j] ← elemento_actual
   Fin Para
   10. gap ← gap / 2
Fin Mientras
```

Pseudocódigo: Ordenación Shell II

- Este enfoque mejora el rendimiento para conjuntos de datos grandes.

Algoritmo Quicksort

- **Quicksort** es un algoritmo de ordenamiento eficiente que utiliza el enfoque **divide y vencerás**.
- Divide el arreglo en dos subarreglos basándose en un **pivote**, de modo que:
  - Los elementos menores al pivote quedan en un subarreglo.
  - Los elementos mayores al pivote quedan en otro subarreglo.
- Este proceso se repite recursivamente hasta que el arreglo esté ordenado.

Pseudocódigo de Quicksort (Explícito)

```
Algoritmo Quicksort(arreglo, inicio, fin):
    Si inicio < fin:
        pivote = particion(arreglo, inicio, fin)
        Quicksort(arreglo, inicio, pivote - 1)
        Quicksort(arreglo, pivote + 1, fin)

Algoritmo particion(arreglo, inicio, fin):
    pivote = arreglo[fin]
    indice = inicio - 1
    Para j desde inicio hasta fin - 1:
        Si arreglo[j] < pivote:
         indice = indice + 1
         Swap arreglo[indice] con arreglo[j]
    Swap arreglo[indice + 1] con arreglo[fin]
    Devolver indice + 1
```

Ejemplo de Quicksort

Arreglo inicial:

$$8, 4, 2, 7, 1, 3, 6, 5$$

1. El pivote inicial es el último elemento: **5**.
2. Se divide el arreglo en dos subarreglos:
   - Elementos menores a 5: $\{\mathbf{4, 2, 1, 3}\}$.
   - Elementos mayores o iguales a 5: $\{\mathbf{8, 7, 6}\}$.
3. Aplicamos recursividad en ambos subarreglos.

Primera partición I

Arreglo:

$$8, 4, 2, 7, 1, 3, 6, 5$$

1. Comparar cada elemento con el pivote (5):
   - $8 > 5$: No intercambiamos.
   - $4 < 5$: Intercambiamos (posición 0):

     $$4, 8, 2, 7, 1, 3, 6, 5$$

     .
   - $2 < 5$: Intercambiamos (posición 1):

     $$4, 2, 8, 7, 1, 3, 6, 5$$

     .
   - $1 < 5$: Intercambiamos (posición 2):

     $$4, 2, 1, 7, 8, 3, 6, 5$$

     .

Primera partición II

- $3 < 5$: Intercambiamos (posición 3):

$$4, 2, 1, 3, 8, 7, 6, 5$$

.

2. Finalmente, intercambiamos el pivote (posición 4):

$$4, 2, 1, 3, 5, 7, 6, 8$$

.

División en subarreglos I

Después de la primera partición:

- Subarreglo izquierdo:

$$4, 2, 1, 3$$

.

- Subarreglo derecho:

$$7, 6, 8$$

.

Llamadas recursivas:

- Ordenar

$$4, 2, 1, 3$$

.

- Ordenar

$$7, 6, 8$$

.

Resolviendo el subarreglo izquierdo:

$$4, 2, 1, 3$$

I

Pivote: 3

1. Comparaciones:
   - $4 > 3$: No intercambiamos.
   - $2 < 3$: Intercambiamos:

     $$2, 4, 1, 3$$

     .
   - $1 < 3$: Intercambiamos:

     $$2, 1, 4, 3$$

     .

Resolviendo el subarreglo izquierdo:

$$4, 2, 1, 3$$

II

2. Finalmente, intercambiamos el pivote:

$$2, 1, 3, 4$$

.

Subarreglos:

- Izquierdo:

$$2, 1$$

.

- Derecho:

$$4$$

.

Ordenamiento final

1. Ordenar

$$2, 1$$

: El pivote es 1.

   - Resultado:

$$1, 2$$

.

2. Ordenar

$$7, 6, 8$$

: El pivote es 8.

   - Resultado:

$$6, 7, 8$$

.

Arreglo ordenado final:

$$1, 2, 3, 4, 5, 6, 7, 8$$

.

Algoritmo MergeSort - Introducción

El algoritmo **MergeSort** es un algoritmo de ordenación basado en la técnica de **divide y vencerás**. Su funcionamiento se puede dividir en dos fases principales:

1. **División**: El arreglo se divide en dos mitades hasta que cada subarreglo tenga un solo elemento.
2. **Fusión (Merge)**: Los subarreglos de un solo elemento se fusionan de forma ordenada hasta obtener el arreglo completo y ordenado.

Fase de División

El algoritmo comienza dividiendo el arreglo en dos mitades. Cada mitad se divide recursivamente hasta que todos los subarreglos tengan un solo elemento.

- Ejemplo inicial: $[38, 27, 43, 3, 9, 82]$
- Se divide en dos mitades: $[38, 27, 43]$ y $[3, 9, 82]$
- La división continúa recursivamente:

$$[38, 27] \quad \text{y} \quad [43] \quad \text{y} \quad [3, 9] \quad \text{y} \quad [82]$$

- Finalmente, los subarreglos resultantes son:
  $[38], [27], [43], [3], [9], [82]$

Fase de Fusión (Merge)

Cuando los subarreglos tienen solo un elemento, se comienza la fase de fusión. Los subarreglos se combinan de manera ordenada.

- Primero se fusionan los subarreglos de un solo elemento: [38] y [27]:

$$\text{Fusionar } [38] \text{ y } [27] \quad \rightarrow \quad [27, 38]$$

- Después, se fusiona $[27, 38]$ con $[43]$:

$$\text{Fusionar } [27, 38] \text{ y } [43] \quad \rightarrow \quad [27, 38, 43]$$

- Similarmente, se fusionan los subarreglos $[3, 9]$ y $[82]$:

$$\text{Fusionar } [3, 9] \text{ y } [82] \quad \rightarrow \quad [3, 9, 82]$$

Fusionando las Mitades Ordenadas

Una vez que cada mitad está ordenada, las dos mitades se fusionan en un solo arreglo ordenado.

- Se fusionan las dos mitades ordenadas: $[27, 38, 43]$ y $[3, 9, 82]$
- El resultado final es:

$$[3, 9, 27, 38, 43, 82]$$

- Así, el algoritmo completa la ordenación del arreglo original.

Caso Base - Subarreglos de un Solo Elemento

Cuando un subarreglo tiene solo un elemento, se considera **ordenado** por definición, ya que un solo elemento no necesita ser comparado ni reordenado.

- Ejemplo: Si tenemos [38] y [27], no se puede dividir más.
- El algoritmo no realiza ninguna operación de comparación en este caso, simplemente retorna el subarreglo.
- Esto permite que la recursión termine, y el algoritmo empiece a fusionar los subarreglos.

Resumen del Proceso

El algoritmo **MergeSort** realiza los siguientes pasos:

1. Divide el arreglo en subarreglos más pequeños hasta que cada uno tenga solo un elemento.
2. Funde los subarreglos ordenados de vuelta en un arreglo ordenado.

El proceso de división recursiva asegura que el arreglo esté ordenado de forma eficiente.

MergeSort: Definición de Funciones y Fusión I

```c
#include <stdio.h>

// Funci n para fusionar dos subarreglos
void merge(int arr[], int left, int mid, int right)
    int n1 = mid - left + 1;  // Tama o del subarr
    int n2 = right - mid;     // Tama o del subarr

    int leftArr[n1], rightArr[n2];  // Arreglos tem

    // Copiar los datos a los arreglos temporales
    for (int i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }
    for (int i = 0; i < n2; i++) {
        rightArr[i] = arr[mid + 1 + i];
    }
```

MergeSort: Definición de Funciones y Fusión II

```
// Fusionar los arreglos temporales
int i = 0, j = 0, k = left;
while (i < n1 && j < n2) {
    if (leftArr[i] <= rightArr[j]) {
        arr[k] = leftArr[i];
        i++;
    } else {
        arr[k] = rightArr[j];
        j++;
    }
    k++;
}

// Copiar los elementos restantes de leftArr[]
while (i < n1) {
    arr[k] = leftArr[i];
```

MergeSort: Definición de Funciones y Fusión III

```
            i++;
            k++;
        }

        // Copiar los elementos restantes de rightArr[]
        while (j < n2) {
            arr[k] = rightArr[j];
            j++;
            k++;
        }
    }
```

MergeSort: Llamada Recursiva y Ordenamiento I

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {  // Condici n para continu
        int mid = left + (right - left) / 2;  // En

        // Llamada recursiva a la primera mitad
        mergeSort(arr, left, mid);

        // Llamada recursiva a la segunda mitad
        mergeSort(arr, mid + 1, right);

        // Fusionar las dos mitades
        merge(arr, left, mid, right);
    }
}

int main() {
```

MergeSort: Llamada Recursiva y Ordenamiento II

```
    int arr[] = {38, 27, 43, 3, 9, 82};
    int size = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, size - 1);

    printf("Arreglo ordenado: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```