Estructuras de Datos: Listas Enlazadas

Cosijopii García

Universidad del Istmo

21 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

21 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

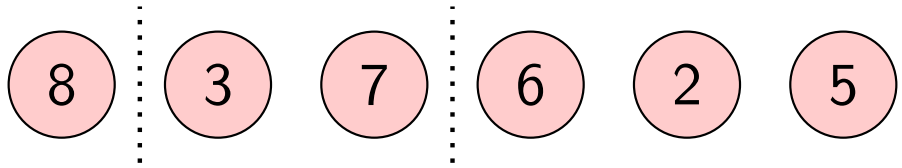
```
Para cada elemento i en la lista
    Para cada elemento j en la lista hasta n-i-1
        Si lista[j] > lista[j+1]
            Intercambiar lista[j] y lista[j+1]
    Fin para
Fin para
```

- 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.