Denilson F. de León A. **IPC2 Sección P 2024 Segundo Semestre**

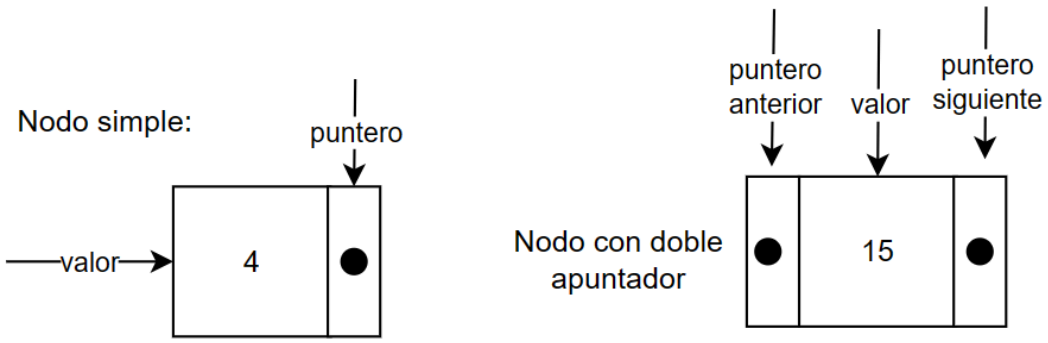
Nodo:

Un nodo es un componente fundamental en la estructura de las listas enlazadas. En el contexto de las listas enlazadas, un nodo es una estructura de datos que contiene dos partes principales: un campo de datos y un enlace o puntero que apunta al siguiente nodo en la secuencia.

Se compone de:

- **Campo de Datos**: En este campo es donde se almacenan los datos o la información que queremos incluir en la lista enlazada. Puede ser un único elemento de datos o una estructura más compleja, dependiendo de la situación.
- **Puntero**: Este componente es crucial para la conexión y enlace de elementos de la lista. Es un puntero que apunta al siguiente nodo en la secuencia. En una lista enlazada simple, cada nodo tiene un enlace que apunta al siguiente nodo. En el último nodo de la lista, el enlace generalmente apunta a un valor especial (como nulo) para indicar el final de la lista.

De forma gráfica se vería como lo siguiente:



En esta representación, se muestra un nodo con un puntero al siguiente nodo y otro nodo con dos punteros, uno al elemento anterior y otro al siguiente. La cantidad de punteros que posee un nodo depende de la estructura que se desea crear.

Utilizando el lenguaje de programación "C", un nodo con un puntero se define como:

```c
struct Nodo {
    int Datos; // Campo de datos donde se asigna un valor
    struct Nodo* PunteroSiguiente; // Puntero al siguiente Nodo
};
```

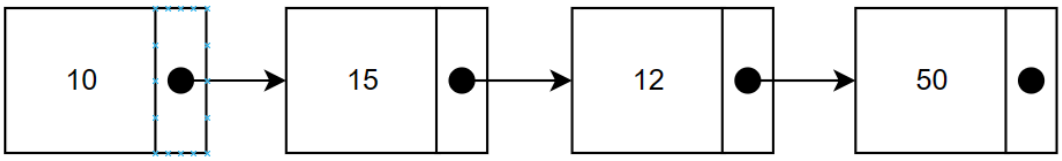
Mientras que uno con doble puntero sería:

```
struct Nodo {
    int Datos; // Campo de datos donde se asigna un valor
    struct Nodo* PunteroAnterior; // Puntero al anterior Nodo
    struct Nodo* PunteroSiguiente; // Puntero al siguiente Nodo
};
```

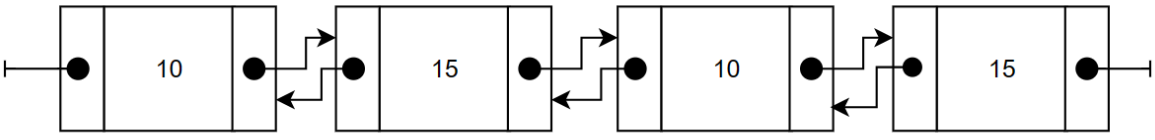
El empleo de este concepto es útil, dado que a diferencia de las estructuras de datos estáticas como los arrays, donde la cantidad de elementos se fija previamente, el uso de nodos permite construir listas enlazadas que pueden crecer o decrecer según las necesidades del programa. Esto resulta en una estructura de datos dinámica y flexible, proporcionando acceso directo a otro nodo, ya sea el siguiente, el anterior, o incluso arriba y abajo en el caso de matrices dispersas.

Listado de Tipos de Listas:

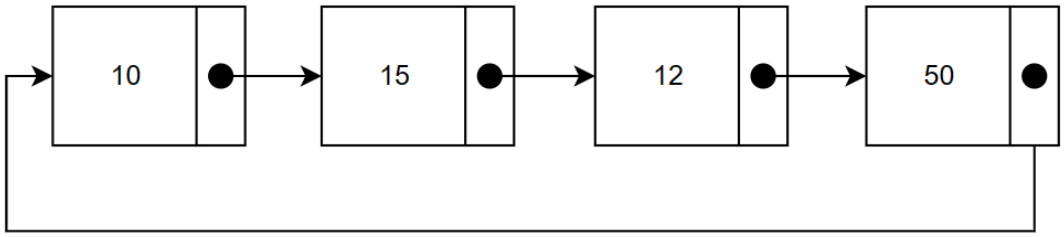
- **Listas Simplemente Enlazadas o Singly Linked Lists:**
  - **Descripción**: En una lista enlazada, cada elemento (nodo) contiene un valor y una referencia al siguiente nodo. Esto permite la creación de estructuras dinámicas donde los elementos pueden insertarse o eliminarse fácilmente.
  - **Ejemplo**:
    En este ejemplo se ilustran 4 nodos en este ejemplo, donde cada nodo posee un apuntador que enlaza al siguiente nodo, creando así una lista simplemente enlazada.



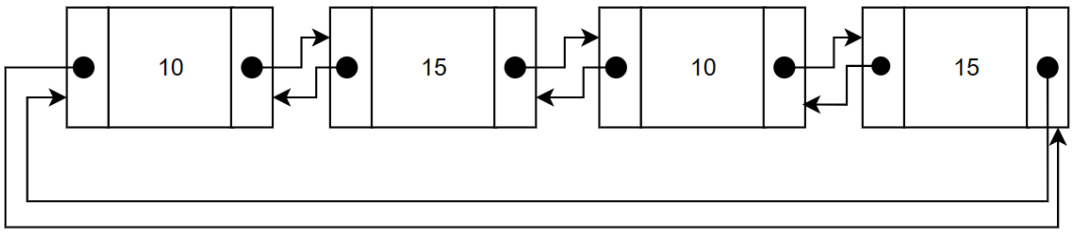
- **Listas Doblemente Enlazadas o Doubly Linked Lists:**
  - **Descripción**: Similar a las listas enlazadas, cada nodo en una lista doblemente enlazada tiene una referencia tanto al nodo siguiente como al anterior. Esto facilita la navegación en ambas direcciones, pero conlleva un mayor uso de memoria.
  - **Ejemplo**:
    En este caso, se muestran 4 nodos, donde cada nodo posee dos apuntadores, uno al anterior y otro al siguiente, creando así una lista doblemente enlazada.



- **Listas Simples Circulares:**
  - **Descripción**: En una lista circular, el último nodo apunta al primer nodo, creando un bucle. Esto significa que no hay un final definido y la lista se puede recorrer infinitamente. Es útil en situaciones donde la secuencia es cíclica.
  - **Ejemplo**:
    Se presentan 4 nodos, similares a las listas simplemente enlazadas, pero el último elemento apunta al primer elemento de la lista como siguiente.



- **Listas Circulares Doblemente Enlazadas:**
  - Descripción: Combina las características de listas doblemente enlazadas y listas circulares. Cada nodo tiene referencias al nodo siguiente y al anterior, y el último nodo apunta al primero, formando un bucle bidireccional.
  - **Ejemplo**:
    En este ejemplo se ilustran 4 nodos, similares a las listas simples circulares. Sin embargo, dado que cada elemento posee 2 apuntadores, el último elemento apunta al primer elemento como siguiente, y a su vez, el primer elemento apunta al último elemento como su anterior.



- **Listas de Acceso Aleatorio:**
  - Descripción: A diferencia de las listas enlazadas, las listas de acceso aleatorio permiten un acceso directo a cualquier elemento mediante índices. Esto facilita la búsqueda y manipulación de elementos específicos.

Con la variedad de tipos de listas enlazadas disponibles, se aconseja realizar un análisis exhaustivo de las necesidades y requisitos específicos del programa o problema en cuestión. No hay una opción única que sea la mejor para todas las situaciones; la elección depende de varios factores. La versatilidad de las listas enlazadas permite seleccionar entre listas simplemente enlazadas, doblemente enlazadas, circulares, o sus combinaciones, según la estructura de datos más eficiente para la tarea en mano. Además, no hay que olvidar que la elección del tipo de lista enlazada adecuada contribuirá significativamente a la eficacia y eficiencia del programa.

Ejemplo de uso de las listas enlazadas:

```
struct Nodo {
    int Datos; // Campo de datos donde se asigna un valor
    struct Nodo* PunteroAnterior; // Puntero al anterior Nodo
    struct Nodo* PunteroSiguiente; // Puntero al siguiente Nodo
};
int main() {
    // Creación de Nodos
    struct Nodo* nodo1 = (struct Nodo*)malloc(sizeof(struct Nodo));
    nodo1->Datos = 10;
    struct Nodo* nodo2 = (struct Nodo*)malloc(sizeof(struct Nodo));
    nodo2->Datos = 20;
    struct Nodo* nodo3 = (struct Nodo*)malloc(sizeof(struct Nodo));
    nodo2->Datos = 30;
   // Enlazando Nodos, dado que se agregan el siguiente de cada nodo y el
del último apunta al primero, resulta en una lista circular simplemente
enlazada
    nodo1->PunteroSiguiente = nodo2;
    nodo2->PunteroSiguiente = nodo3;
  nodo3->PunteroSiguiente = nodo1;

// Si agregamos los enlaces anteriores, resulta en una lista circular
doblemente enlazada
    nodo1->PunteroAnterior = nodo3;
    nodo2->PunteroAnterior = nodo1;
  nodo3->PunteroAnterior = nodo2;

    // Ejemplo de Acceso a Datos y Navegación en la Lista utilizando
siguientes
    printf("Dato del nodo1: %d\n", nodo1->Datos);
    printf("Dato del nodo2: %d\n", nodo1->PunteroSiguiente->Datos);

   // Ejemplo de Acceso a Datos y Navegación en la Lista utilizando
```

```
anteriores
    printf("Dato del nodo1: %d\n", nodo2->PunteroAnterior->Datos);

// También es posible retroceder desde nodo 1 hacia nodo 2 utilizando los
anteriores y pasando por nodo 3 dado que es una lista circular
    printf("Dato del nodo2: %d\n",
nodo1->PunteroAnterior->PunteroAnterior->Datos);

    // Liberar memoria al finalizar
    free(nodo1);
    free(nodo2);
    free(nodo3);
    return 0;
}
```

Ordenamientos:

Dado que las listas enlazadas contienen valores, es posible aplicar estrategias de ordenamiento para organizar sus elementos. A continuación, se presentan algunos algoritmos comunes y una breve explicación de su funcionamiento:

- **Ordenamiento Burbuja (Bubble Sort):** Este algoritmo compara repetidamente pares de nodos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que la lista esté ordenada.
- **Ordenamiento por Inserción (Insertion Sort):** En este algoritmo, se construye una secuencia ordenada de elementos uno a uno. En cada iteración, se toma un elemento de la lista no ordenada y se inserta en su posición correcta en la lista ordenada.
- **Ordenamiento por Mezcla (Merge Sort):** Este algoritmo utiliza el enfoque de divide y conquista para ordenar la lista. Divide la lista en dos mitades, ordena cada mitad de forma recursiva y luego combina las dos mitades ordenadas.

A continuación, se presenta un ejemplo de implementación del algoritmo de ordenamiento por inserción en el lenguaje C. Este algoritmo trabaja mediante la construcción de una secuencia ordenada, tomando cada elemento de la lista no ordenada y ubicándolo en su posición correcta en la lista ordenada.

```
void insertionSort(struct Nodo* inicio) {
    struct Nodo* actual = inicio->PunteroSiguiente;
    while (actual != NULL) {
        struct Nodo* temp = actual;
        while (temp->PunteroAnterior != NULL && temp->Datos <
temp->PunteroAnterior->Datos) {
            intercambiar(temp, temp->PunteroAnterior);
            temp = temp->PunteroAnterior;
        }
```

```
        actual = actual->PunteroSiguiente;
    }
}
```