Escuela de Minas
Dr. Horacio Carrillo

Estructuras de datos

Unidad 5: TDA Listas dobles

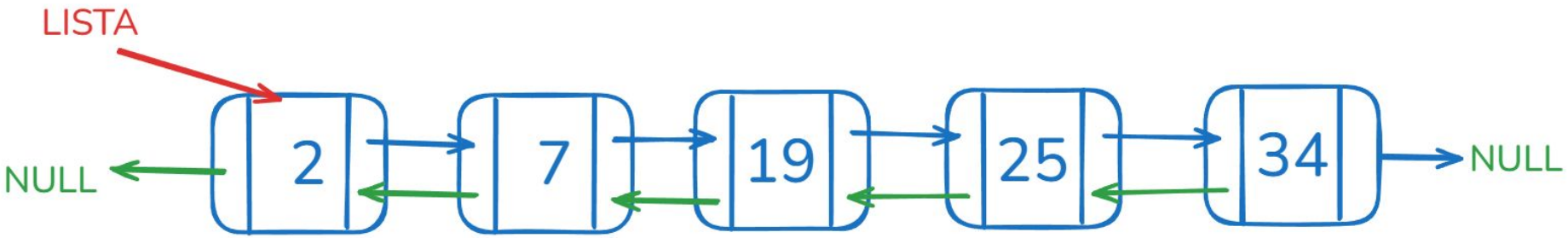
A.P.U. Gustavo Alberto Ordoñez

Concepto de lista doblemente enlazada

Una **lista doblemente enlazada** es una colección de nodos, que a diferencia de una lista simple, los nodos cuentan con tres propiedades o campos: **Puntero siguiente, Puntero anterior y dato.**


```
typedef struct tnodo *pnodo;
typedef struct tnodo{
        int dato;
        pnodo sig;
        pnodo ant;
        };
```

Representación de una lista doblemenete enlazada



Operaciones de TDA Lista Doble

Las operaciones fundamentales de las listas dobles son:

- IniciarLista
- AgregarInicio
- AgregarFinal
- AgregarOrden
- QuitarInicio
- QuitarFinal
- QuitarNodo
- BuscarNodo

Especificación: iniciar_lista

Objetivo: inicializar la lista, generando una lista vacía.

Entrada: una lista (puntero de inicio de la lista).

Salida: devuelve una lista vacía (puntero de inicio de la lista en valor nulo).

Restricciones: ninguna

Especificación: agregar_inicio

Objetivo: agregar un nuevo nodo en el primer lugar de la lista.

Entrada: una lista y un nuevo dato.

Salida: devuelve una lista con un nuevo nodo al principio.

Restricciones: una lista inicializada y espacio en memoria para la creación del nuevo nodo.

Especificación: agregar_final

Objetivo: agregar un nuevo nodo al final de la lista.

Entrada: una lista y un nuevo dato.

Salida: devuelve una lista con un nuevo nodo al final.

Restricciones: una lista inicializada y espacio en memoria para la creación del nuevo nodo.

Especificación: agregar_orden

Objetivo: agregar un nuevo nodo respetando el orden de la lista.

Entrada: una lista y un nuevo dato.

Salida: devuelve una lista ordenada con un nuevo nodo.

Restricciones: una lista inicializada y espacio en memoria para la creación del nuevo nodo.

Especificación: quitar_inicio

Objetivo: eliminar el primer nodo de la lista.

Entrada: una lista.

Salida: devuelve la lista con un nodo menos.

Restricciones: una lista inicializada, la lista no debe estar vacía.

Especificación: quitar_final

Objetivo: eliminar el último nodo de la lista.

Entrada: una lista.

Salida: devuelve la lista con un nodo menos.

Restricciones: una lista inicializada, la lista no debe estar vacía.

Especificación: quitar_nodo

Objetivo: eliminar un nodo específico de la lista.

Entrada: una lista y el valor a extraer.

Salida: devuelve la lista con un nodo menos o un mensaje de error en caso de no encontrar el elemento.

Restricciones: una lista inicializada, la lista no debe estar vacía.

Especificación: mostrar_lista()

Objetivo: visualizar el contenido de los nodos de la lista.

Entrada: una lista (puntero de inicio de la lista).

Salida: se muestra por pantalla los datos almacenados en los nodos.

Restricciones: una lista inicializada, la lista no debe estar vacía.

Especificación: buscar

Objetivo: buscar un valor específico en la lista.

Entrada: una lista, el valor a buscar.

Salida: devuelve *true* si el dato está en la lista y *false* en caso contrario.

Restricciones: una lista inicializada, la lista no debe estar vacía.

Implementación: TDA Nodo

La creación de una lista enlazada requiere de una variable para almacenar la información y puntero de acceso a la lista, que apunta al primer nodo de la lista. Opcionalmente se puede declarar un puntero al nodo final de la lista.

```
typedef struct tnodo *pnodo;
typedef struct tnodo{
        int dato;
        pnodo sig;
        pnodo ant;
        };
```

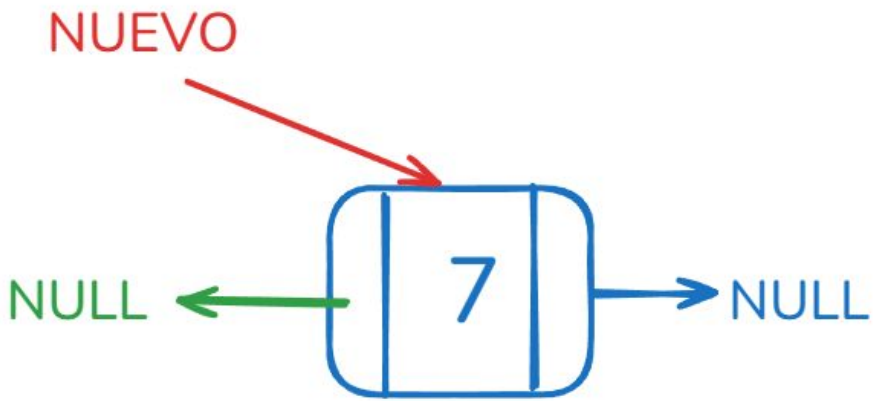
- Donde *pnodo es un puntero al registro pnodo.
- int **dato** se utiliza para guardar datos de tipo entero.
- pnodo **siguiente** es el enlace al siguiente nodo.
- pnodo **anterior** es el enlace al anterior nodo.

Implementación: iniciar_lista

```
void IniciarLista(pnodo &inicio)
{
    inicio = NULL;
}
```

Implementación: crear_nodo

```
void CrearNodo(pnodo &nuevo, int valor)
{
    nuevo = new tnodo;
    if(nuevo != NULL)
    {
        nuevo -> dato = valor;
        nuevo -> sig = NULL;
        nuevo -> ant = NULL;
    }
}
```



Implementación: agregar_inicio

Se contemplan dos situaciones:

1. Si la lista está vacía.
2. Si la lista NO está vacía.

```
void agregar_inicio(pnodo &inicio, pnodo nuevo)
{
    if(inicio == NULL){
        inicio = nuevo;
    }
    else{
        nuevo -> sig = inicio;
        inicio -> ant = nuevo;
        inicio = nuevo;
    }
}
```

Implementación: agregar_final

Se contemplan dos situaciones:

1. Si la lista está vacía.
2. Si la lista NO está vacía.

```
void AgregarFinal(pnodo &inicio, pnodo nuevo)
{
    pnodo i;
    if(inicio == NULL){
        inicio = nuevo;
    }
    else{
        for(i = inicio; i -> sig != NULL; i = i -> sig);
        i -> sig = nuevo;
        nuevo -> ant = i;
    }
}
```

Implementación: agregar_orden

Se contemplan 3 situaciones:

1. Si la lista está vacía.
2. Agregar al inicio.
3. Buscar su posición.

```
void AgregarOrden(pnodo &inicio, pnodo nuevo)
{
   pnodo i;
   if(inicio == NULL){
      inicio = nuevo;
   }
   else{
      if(nuevo -> dato < inicio -> dato)
      {
         nuevo -> sig = inicio;
         inicio -> ant = nuevo;
         inicio = nuevo;
      }
      else{
         for(i = inicio; i -> sig != NULL && (i -> sig) -> dato < nuevo -> dato; i = i -> sig);
         if(i -> sig != NULL){
            nuevo -> sig = i -> sig;
            nuevo -> ant = i;
            (i -> sig) -> ant = nuevo;
            i -> sig = nuevo;
         }
         else{
            i -> sig = nuevo;
            nuevo -> ant = i;
         }
      }
   }
}
```

Implementación: quitar_inicio

Se contemplan 3 situaciones:

1. Si la lista está vacía.
2. Solo hay un elemento en la lista.
3. Hay más de un elemento en la lista.

```
pnodo QuitarInicio(pnodo &inicio)
{
    pnodo extraido;
    if(inicio == NULL)
    {
        extraido = NULL;
    }
    else{
        if(inicio -> sig == NULL)
        {
            extraido = inicio;
            inicio = NULL;
        }
        else{
            extraido = inicio;
            inicio = inicio -> sig;
            inicio -> ant = NULL;
            extraido -> sig = NULL;
        }
    }
    return extraido;
}
```

Implementación: quitar_final

Se contemplan 2 situaciones:

1. Si la lista está vacía.
2. La lista no está vacía.

```
pnodo QuitarFinal(pnodo &inicio)
{
    pnodo extraido, i;
    if(inicio == NULL)
    {
        extraido = NULL;
    }
    else{
        for(i = inicio; i -> sig != NULL; i = i -> sig);
        extraido = i;
        (i -> ant) -> sig = NULL;
        i -> ant = NULL;
    }
    return extraido;
}
```

Implementación: quitar_nodo

Se contemplan 3 situaciones:

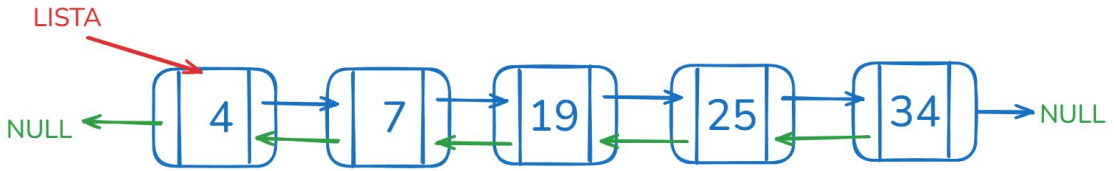
1. Si la lista está vacía.
2. El nodo a quitar es el primero.
3. El nodo a quitar NO es el primero y se requiere buscarlo.

```
pnodo QuitarNodo(pnodo &inicio, int valor)
{
   pnodo extraido, i;
   if(inicio == NULL)
   {
      extraido = NULL;
   }
   else{
      if(inicio -> dato == valor)
      {
         extraido = inicio;
         inicio = inicio->sig;
         inicio->ant = NULL;
         extraido->sig = NULL;
      }
      else{
         for(i = inicio; i != NULL && i -> dato != valor; i = i -> sig);
            if(i != NULL){
               extraido = i;
               (i -> ant) -> sig = i -> sig;
               i -> sig = i -> ant;
               extraido -> sig = NULL;
               extraido -> ant = NULL;
            }
      }
   }
   return extraido;
}
```

Implementación: mostrar_lista

La lista debe contener elementos.

```
void MostrarLista(pnodo inicio)
{
    pnodo i;
    if(inicio != NULL)
    {
        for(i = inicio; i != NULL; i = i -> sig){
            cout << i -> dato << endl;
        }
    }
}
```



En resumen

- Una lista doblemente enlazada es aquélla en la que cada nodo tiene una referencia a su sucesor y otra a su predecesor.
- Se pueden recorrer en ambos sentidos.
- Las operaciones básicas son: agregar, quitar, recorrer lista.
- El recorrido de una lista enlazada significa pasar por cada nodo y procesarlo.
- El proceso de cada nodo puede consistir en escribir su contenido, modificar el campo dato, etc.

Bibliografía

Joyanes Aguilar. Estructuras de datos en C++. McGraw Hill. 2007.

Hernandez Roberto. Estructuras de datos y algoritmos. Prentice Hall. 2001.

Joyanes Aguilar, Luis. Fundamentos de Programación. Mc Graw Hill. 1996.