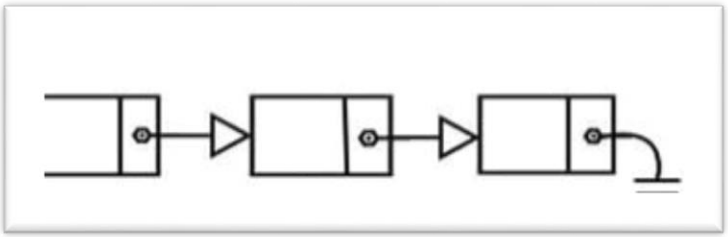
UNIVERSIDAD TECNOLÓGICA NACIONAL

INSTITUTO NACIONAL SUPERIOR
DEL PROFESORADO TÉCNICO



LISTAS ENLAZADAS

[Subtítulo del documento]

DESCRIPCIÓN BREVE

Estructura de Datos

Coordinacion

Programación en C

Tabla de contenido

Estructuras de Datos Auto referenciadas

Listas Simples

Introducción:

Cuando definimos en un programa una variable estática estamos fijado previamente cual va a ser su espacio en memoria y cuáles van a ser los posibles valores que puede tomar a lo largo de la ejecución del programa. Existen problemas complejos que se resuelven más eficazmente mediante la utilización de variables que cambien dinámicamente la cantidad o el tipo de datos que pueden contener. Este tipo de estructuras de datos se denomina estructuras dinámicas.

Las listas enlazadas son un recurso dinámico muy potente que nos permite realizar tareas más eficientemente que con los arrays.

Características de las listas enlazadas:

- Una lista enlazada es una colección lineal de estructuras autoreferenciadas llamadas nodos, conectadas por enlaces de apuntador.
- El acceso a una lista enlazada por medio de un apuntador al primer nodo de la lista y a los nodos subsecuentes por medio del apuntador de enlace almacenado en cada nodo.
- **Fin de lista:** Para marcar el fin de lista, el apuntador de enlace en el último nodo de la lista, se define a NULL.
- Pueden variar el tamaño (Cantidad de datos) a lo largo de la ejecución el programa
- Se adaptan mejor a la naturaleza del problema, aprovechando más eficientemente los recursos de memoria.
- Son más complejas de manejar pues debemos controlar el crecimiento o reducción del espacio de memoria que ocupan.
- Se almacenan en memoria principal
- Una vez definidas se utilizan como cualquier variable estática.
- **Almacenamiento dinámico:** En una lista enlazada lista datos se almacenan dinámicamente, es decir, cada nodo se crea conforme sea necesario.
- Cada nodo puede contener datos de cualquier tipo, incluyendo otras struct. Las pilas y las colas también son estructuras de datos lineales, y como veremos son versiones restringidas de las listas enlazadas.
- Las funciones principales de las listas enlazadas son creación, inserción (como 1º elemento, último elemento, en orden), búsqueda, impresión y eliminar un dato o la lista completa.

Ventajas de la utilización de listas sobre los arrays:

- Una lista enlazada es apropiada cuando no es predecible de inmediato el número de elementos de datos a representarse en la estructura. En el caso de declararse un array sobredimensionado que contenga más espacios de memoria que la cantidad de datos

a ingresar, se estaría desperdiciando memoria. En estas situaciones las listas enlazadas pueden obtener un mejor aprovechamiento de la memoria.

- Puede resultar bastante complicado insertar o eliminar en un dato en un array ya ordenado.
- Tratándose de estructuras de datos que crecen o se reducen en tiempo de ejecución, es posible ahorrar memoria utilizando asignación dinámica de memoria (en vez de los arreglos). Recuerde, sin embargo, que los punteros ocupan espacio, y que la asignación dinámica de memoria incurre en sobrecarga por las llamadas de función, más en el caso de la utilización de técnicas recursivas.
- Son dinámicas, por lo que conforme sea necesario, la longitud de la lista puede aumentar o disminuir, mientras que los arreglos que almacenan las listas de datos no son dinámicos, la memoria del arreglo es asignada en tiempo de compilación.
- Las listas enlazadas sólo se llenan cuando el sistema no tiene suficiente memoria para satisfacer las solicitudes de asignación dinámica de almacenamiento

Desventajas de la utilización de listas sobre los arrays:

- Los elementos de un arreglo se almacenan de forma contigua en memoria. Esto permite acceso inmediato a cualquier elemento del arreglo, porque la dirección de cualquier elemento puede ser calculada directamente, basada en su posición en relación con el principio del arreglo. Las listas enlazadas no proporcionan un acceso inmediato como éste a sus elementos.

Implementación:

Es una estructura autoreferenciada que contiene un miembro de apuntador que apunta a una estructura del mismo tipo de estructura. Por ejemplo, la definición:

Distintos tipos de Definición de Estructuras		
typedef struct elemento { int dato; struct elemento*sig; } nodo;	struct lista { int dato; struct lista*sig; } ; typedef struct lista nodo;	typedef struct nodo { int dato; struct nodo*sig; } nodo;

Se define un tipo de dato, struct lista. El miembro sig apunta a una estructura de tipo struct lista, una estructura del mismo tipo que la que se está declarando aquí, de ahí el término de "estructura autoreferenciada". El miembro sig se conoce como un enlace o vínculo, ya que sirve para vincular una estructura de tipo struct lista con otra estructura del mismo tipo.

Las estructuras autoreferenciadas pueden ser enlazadas juntas para formar útiles estructuras de datos como son las listas, las colas de espera, las pilas y los árboles. En la figura se ilustran dos estructuras autoreferenciadas, enlazadas juntas para formar una lista. La diagonal de la figura representa un apuntador NULL, el fin de la estructura de datos.

Creación de una Lista Enlazada:

Habiendo definido la Estructura de la Lista, la primera operación de creación es asignar la lista a NULL:

```
nodo*lista=NULL;
```

Asignación Dinámica De Memoria

La creación y el mantenimiento de estructuras dinámicas de datos, requiere de la asignación dinámica de memoria, es decir, la capacidad por parte de un programa de obtener, en tiempo de ejecución más espacio de memoria para contener nuevos nodos, y poder liberar espacio ya no requerido.

Las funciones **malloc** y **free** y el operador **sizeof**, son esenciales en la asignación dinámica de memoria. La función malloc toma como argumento el número de bytes a asignarse, y regresa un apuntador de tipo void* (apuntador a void) a la memoria asignada.

A continuación se deberá crear una función cuyo tipo de valor de retorno será el mismo del prototipo. La función creará una estructura dinámica con el tamaño del prototipo, en Lenguaje C disponemos de la función malloc para la asignación dinámica de memoria y nos devuelva la dirección donde podremos almacenar los datos, un puntero a una nueva zona de memoria o NULL en caso de que no exista memoria disponible, estableciéndose un valor inicial para los elementos de la estructura.
Ejemplo de la función que recibe como parámetro un dato de tipo entero, asigna memoria dinámica y devuelve un puntero a nodo con el dato y el enlace a NULL.

Ejemplo de la función que recibe como parámetro un dato de tipo entero, asigna memoria dinámica y devuelve un puntero a nodo con el dato y el enlace a NULL;

```
nodo * nuevonodo(int x)
{
 nodo *a;
 a=(nodo*)malloc(sizeof(nodo));
 a->dato=x;
 a->sig=NULL;
 return a;
 }
```

Este enunciado evalúa sizeof (nodo) para determinar el tamaño en bytes de una estructura de tipo nodo, asigna en memoria una nueva área de tamaño sizeof (nodo) bytes, y almacena en la variables nuevo nodo un apuntador a la memoria asignada. Si no existe memoria disponible, malloc regresa NULL sino la función retornará la dirección de la lista creada (valor retornado por malloc). Para liberar la memoria asignada dinámicamente mediante una llamada malloc previa, utilice un enunciado:

```
free (lista);
```

Inserción de Datos:

Luego de haber creado la lista, hay diferentes tipos de inserción, además se deberán hacer comparaciones con fines de prevenir errores, como cuando la lista está vacía, o el lugar donde se va a ingresar el nuevo elemento, etc.

Nota: Para no utilizar una variable auxiliar, el método de trabajo es utilizar pasajes por Dirección de Dobles Punteros (Por referencia), técnica que nos permite conservar el puntero a la lista original.

Forma Iterativa de Inserción en una Lista

```
void insertar(nodo** x, int y)
{
 nodo *nuevo,*anterior,*p;
 nuevo=nuevonodo(y);
 if(*x==NULL) *x=nuevo;
  else if(y<(*x)->dato)
       {
            nuevo->sig=*x;
            *x=nuevo;
     } else
            {
             anterior=p=*x;
             while((p->sig!=NULL)&&(y>p->dato))
              {
               anterior=p;
               p=p->sig;
              }
             if(y>p->dato) // se inserta último
                anterior=p;
             nuevo->sig=anterior->sig;
            anterior->sig=nuevo;
            }

 }
```

Forma Recursiva de Inserción en una Lista		
Insertar al Final (La más sencilla)	Insertar al Principio	Insertar en Orden

void insertar(nodo** x, int y)
{
nodo *nuevo;
nuevo=nuevonodo(y);
if(*x==NULL) *x=nuevo;
else insertar(&(*x)->sig,y);
}
	void insertar(nodo** x, int y)
	{
	nodo *nuevo;
	nuevo=nuevonodo(y);
	if(*x==NULL) *x=nuevo;
	else
	{
	nuevo->sig=*x;
	*x=nuevo;
	}
	}
		void insertar(nodo** x, int y)
		{
		nodo *nuevo;
		nuevo=nuevonodo(y);
		if(*x==NULL) *x=nuevo;
		else if((*x)->dato>y)
		{
		nuevo->sig=*x;
		*x=nuevo;
		} else insertar(&(*x)->sig,y);
		}

Inserción de Datos sin Repeticiones:

Para insertar un elemento se evalúa si existe en la lista, en caso de existir incrementa el contador de eventos.

Agregar a la Estructura Original el Campo cont	Insertar en Orden sin Repeticiones

typedef struct nodo
{
int dato;
int cont;
struct nodo*sig;
} nodo;
	void insertar(nodo** x, int y)
	{
	nodo *nuevo;
	nuevo=nuevonodo(y);
	if(*x==NULL) *x=nuevo;
	else if((*x)->dato>y)
	{
	nuevo->sig=*x;
	*x=nuevo;
	} else if (y==(*x)->dato)(*x)->cont++;
	else insertar(&(*x)->sig,y);
	}

Búsqueda de Elementos de la Lista:

Búsqueda Iterativa	Búsqueda Recursiva

nodo* buscar(nodo*x, int y)
{ nodo*aux;
aux=x;
while(aux) {
if (aux->dato!= y) aux = aux->sig;
else break;
}
return aux;} /* Retornamos la dirección del elemento encontrado */
	nodo*buscar(nodo*x, int y)
	{
	if(!x) return (NULL);
	else if(y==x->dato) return(x);
	else return(buscar(x->sig,y));
	}

Imprimir Datos de la Lista

Forma Iterativa	Forma Recursiva

void listar(nodo*x)
{
  while(x!=NULL)
  {
  p("%4d",x->dato);
  x=x->sig;
  }
}
	void listar(nodo*x)
	{
	  if(x)
	  {
	  p("%4d",x->dato);
	  listar(x->sig);
	  }
	}

Eliminar un Nodo de la Lista

Secuencia de Pasos para Eliminar un Elemento por Dirección

- Necesitaremos inicio de lista y dirección del elemento a eliminarla.

En la Forma iterativa:

- Utilizamos una variable auxiliar para buscar el elemento anterior al elemento a eliminar recorriendo toda la lista.
- En caso de no encontrar el elemento comparando su dirección retornamos 0 (FALSE).
- Comprobamos si el elemento a eliminar es el último, en tal caso el elemento anterior apuntará a NULL, de lo contrario el elemento anterior al que eliminamos apuntará al elemento que apuntaba el elemento que eliminamos.

En ambas formas liberamos la memoria del elemento a eliminar.

Eliminar un Elemento de la Lista	
Forma Iterativa	Forma Recursiva

```
int borranodo(nodo*x, nodo*inf)
{
  nodo*aux;
  aux = x;
  while (aux)
   {
    if (aux->sig!= inf)  aux = aux->sig;
        else    break;
   }
 if (aux == NULL) return 0;
      else if (inf->sig == NULL)   aux->sig = NULL;
           else aux->sig = inf->sig;
 free(inf);
 return 1;
}
```

```
void borranodo(nodo**x, int y)
{
 nodo*aux;
 if(*x)
 {
  aux=*x;
  if((*x)->dato==y)
   { p("\nNodo a eliminar:%3d",(*x)->dato);
    getch();
   *x=aux->sig;
    free(aux);
   } else  borranodo(&(*x)->sig,y);
 }
}
```

Eliminar la Lista

Forma Iterativa	Forma Recursiva

```
void eliminar(nodo**x)
 {
  nodo*aux;
  while (*x != NULL )
  {
   aux = *x;
   p("\nNodo a eliminar:%3d",(*x)->dato);
   getch();
   * x = (*x)->sig;
   free(aux);
 }
}
```

```
void eliminar(nodo**x)
 {
 nodo*aux;
 if(*x)
 {
  aux=*x;
  p("\nNodo a eliminar:%3d",(*x)->dato);
   getch();
   *x=aux->sig;
    free(aux);
   eliminar(x);
 }
}
```

Ejercitación:

1. Ingresar datos en una lista en orden sin repeticiones mostrarla y extraer de la lista original los nodos con datos pares y almacenarlos en la lista par. Contar la cantidad de nodos.(ParesBis.c)
2. Cargar los datos de una lista en un archivo binario. (ListArch.c).

3. Cargar una lista original sin repeticiones y una lista secundaria de las mismas características. Se pide extraer el nodo completo de cada lista y generar una tercera lista con la inserción en orden de ambas listas. (Gusti19.c).

4. Dadas las siguientes estructuras:

```
typedef struct
{
 int cod_cli;
 char nom_cli[30];
 int cuenta;
 } registro;
```

```
typedef struct lista
{
registro datos;
struct lista*sig;
} nodo;
```

Cargar el archivo de texto separado por comas con registros repetidos en una lista ordenada por la primera palabra de cada renglón, posincrementando el campo de cuenta. Mostrar la lista y posteriormente cargar la lista en un archivo binario.(Listabin.c).

5. Generar un programa que permita aplicar el siguiente menú con listas simples.

```
MENU DE OPCIONES (LISTAS)

- Ingresar Datos al Principio de la Lista <1>
- Ingresar Datos al Final de la Lista     <2>
- Listar Datos                            <3>
- Borrar Nodo                             <4>
- Borrar Lista                            <5>
- Salir                                   <6>

Ingrese Opcion:
```