TEMA 1.3
Estructura de datos Lineales

Objetivo:

El alumno aplicará las formas de representar y operar en la computadora las principales listas lineales

Generalidades

- Las estructuras de datos lineales están compuestas por una secuencia de 0 o más elementos de algún tipo determinado en ocasiones ordenados de alguna forma.

- Puede crecer o disminuir en el número de elementos y podrán insertarse o eliminarse elementos en determinadas posiciones sin alterar su estructura lógica.

El concepto de Lista

- Una lista es una colección de elementos del mismo tipo
- Para el acceso y manejo de esos elementos se utilizan índices o posiciones
- Una lista tiene operaciones de inserción, eliminación y búsqueda
- Una lista es una estructura de datos flexible en las operaciones que se realizan

1.3.1 Pila

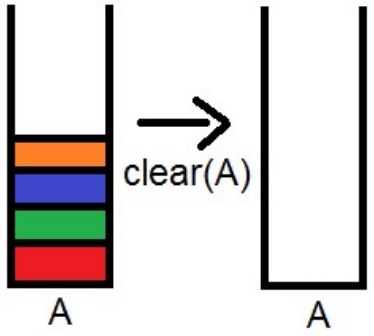
- Una pila (stack) es un tipo abstracto de datos tipo LIFO (*last in, first out*).
- Es un tipo de dato que soporta operaciones de inserción y eliminación de elementos de un conjunto.
- Normalmente se usa en aplicaciones en las que se necesita recuperar información en orden inverso a como ha entrado.

1.3.1 Pila

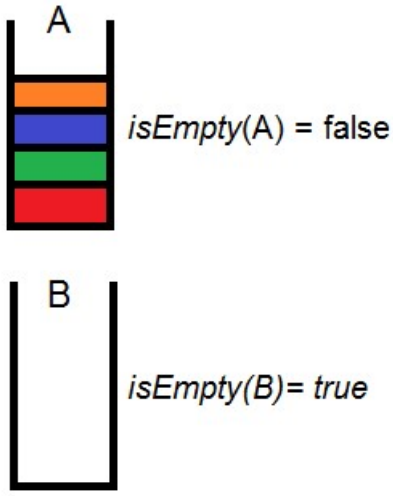
- Esta estructura funciona de forma similar a una torre de elementos del mismo tipo (documentos pendientes, platos, libros, etc).
- Cada vez que llega un elemento, se coloca arriba de los que ya están acumulados, y cuando se van atendiendo se toma el que se encuentra en la parte superior.

1.3.1.1 Operaciones

- Las operaciones que se realizan sobre una pila son:
  - *clear()* Borra todos los elementos de una pila

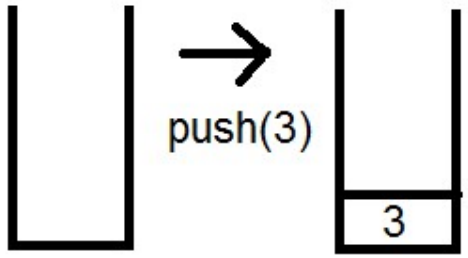


  - *isEmpty()* Verifica si una pila se encuentra vacía

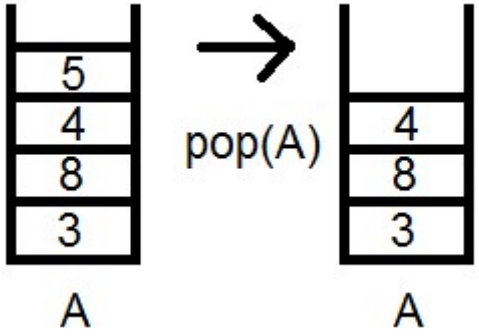


1.3.1.1 Operaciones

- *push()* Mete un elemento *en* el tope de la pila

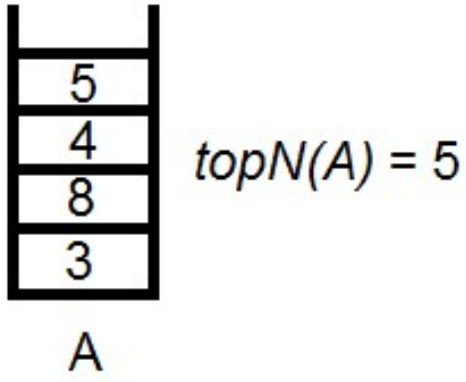


- *pop()*



1.3.1.1 Operaciones

- *top()* Regresa el elemento que se encuentra en la parte superior de la pila sin removerlo.



1.3.1.1 Operaciones

- La complejidad temporal asintótica de las operaciones es:

Operación	Orden
Create	O(1)
clear()	O(n)
isEmpty()	O(1)
Push()	O(1)
Pop()	O(1)
top	O(1)

1.3.1.3 Implementación

```
Pila:
    int tope;
    Lista elementos;

Pila crearPila()
    Pila p;
    Lista lista1;
    p.tope = -1;
    p.elementos = lista1;
    return p;
```

1.3.1.3 Implementación

```
bool esVacia(Pila p)
    if(p.tope==-1)
        return true;
    return false;

void meter(Pila p,int x)   //void push(..)
    p.tope = p.tope+1
    p.elementos[p.tope] = x
```

1.3.1.3 Implementación

```
int sacar(Pila p) // int pop(..)
    if(esVacia(p))
        print(“Error”)
        return -1
    else
        int aux= p.elementos[p.tope]
        p.elementos[p.tope]=NULL;
        p.tope=p.tope-1
        return aux
```

1.3.1.3 Implementación

```
int top(Pila p)
    if(esVacia(p))
        print(“Error”);
        return -1;
    else
        return p.elementos[p.tope]
```

1.3.1.2 Ejemplos de uso

- Evaluación de expresiones aritméticas (1)

Shunting-yard Algorithm

Propuesto por Edsger W. Dijkstra en la década de los 60’s. Utiliza dos pilas, una para operandos y una para operadores.

Para este tipo de expresiones se requieren 2 stacks, una para los operando y otra para los operadores

1.3.1.2 Ejemplos de uso

Dada una expresión aritmética (leyendo de izquierda a derecha)

- Ignorar los paréntesis de apertura
- Operando: **Push(opn)** en stack de operandos
- Operador: **Push(opr)** operadores en stack de operadores
- Al encontrar un paréntesis de cierre
  - **Pop** operando
  - **Pop** operador
  - **Pop** operando
  - Realizar operación
  - **Push** resultado en stack de operandos
- Al finalizar la expresión
  - Pop resultado final.

1.3.1.2 Ejemplos de uso

- Evaluación de expresiones aritméticas (2).

Reverse Polish Notation

La ventaja de esta forma de escribir expresiones aritméticas, es que no es necesario utilizar paréntesis ni reglas de precedencia.

En este algoritmo se utiliza una sola pila (stack).

1.3.1.2 Ejemplos de uso

Sea una expresión aritmética (leyendo de izquierda a derecha)

- Si se encuentra un operando:
  - **Push** operando.
- Si se encuentra un operador:
  - **Pop** operando.
  - **Pop** operando.
  - **Push** resultado.

1.3.2 Cola

- Una cola (queue) es un tipo abstracto de datos de tipo FIFO (first in first out).
- Es un tipo de dato que soporta operaciones de inserción y eliminación de elementos de un conjunto de datos

1.3.2 Cola

- Modela situaciones en las que una tarea se realiza en el mismo orden en el que se reciben.
- Ejemplos de ello son tareas que esperan ser atendidas por alguna aplicación en la computadora o en un servidor.
- Los cambios que se realizan, se deben monitorear en ambos extremos de la cola.

1.3.2.1 Operaciones

- Las operaciones que se realizan sobre una cola son:
  - *create()* Crea una cola vacía.
  - *isEmpty(queue C)* Devuelve verdadero si una cola se encuentra vacía.
  - *enqueue(n)* Ingresa un elemento a la cola

1.3.2.1 Operaciones

- *dequeue(q)* Extraer el elemento al inicio de la cola.
- *clear(q)* Borra todos los elementos de la cola.
- *first(n)* Regresa el primer elemento de la cola sin eliminarlo.

1.3.2.1 Operaciones

- La complejidad temporal asintótica de las operaciones es:

Operación	Orden
Create	O(1)
clear()	O(n)
isEmpty()	O(1)
enqueue()	O(1)
dequeue()	O(1)
first(n)	O(1)

1.3.2.2 Implementación

```
Cola:
    int primero;
    int ultimo;
    Lista elementos;

Queue CrearCola()
    Queue c;
    Lista miLista;
    c.primero = 0
    c.ultimo = -1
    c.elementos = miLista;
    return c;
```

1.3.2.2 Implementación

```
bool esVacia(Queue c)
     if(c.primero==c.ultimo+1)
          return true;
     return false;

void encolar(Queue c,int x)
     c.ultimo = c.ultimo+1
     c.elementos[c.ultimo] = x
```

1.3.2.2 Implementación

```
int desencolar(c)     //(dequeue)
    if(esVacia(c))
        "error"
        return -1
    else
        int aux = c.elementos[c.primero]
        c.elementos[c.primero]=null;
        c.primero = c.primero + 1
        if(primero==ultimo+1)
            c=crearCola()
        return aux
```

1.3.2.3 Aplicaciones

- Algunos ejemplos de uso de “queues”:
  - Procesos atendidos por un procesador
  - Documentos pendientes de imprimir
  - En redes de computadoras, los paquetes suelen transportarse a través de colas.

1.3.3 Cola Circular

- Los elementos están de forma “circular” y cada elemento tiene un sucesor y un predecesor.
- El sucesor del último elemento se trata del primer elemento mientras que el antecesor del primero es el último elemento.
- En una cola circular es indispensable definir desde el inicio el tamaño máximo de la cola
- La ventaja de una Cola circular es que el manejo de memoria es más eficiente

1.3.3 Cola Circular (Operaciones)

- Para las operaciones con la cola circular, es posible trabajar con las mismas que una cola básica, sin embargo, **en la implementación** se requiere añadir elementos a las funciones encolar y desencolar, para el correcto manejo de índices (si se implementa con arreglos) o para el correcto manejo de memoria (si se implementa con listas ligadas)

1.3.3 Cola doble

- Una cola doble (dqueue) es un tipo abstracto de datos en donde cada elemento puede ser ingresado y recuperado por ambos lados de la estructura.
- Con esto se da una mayor flexibilidad al tipo de dato ya que el elemento que ingresa puede ser el primero o el último en salir.

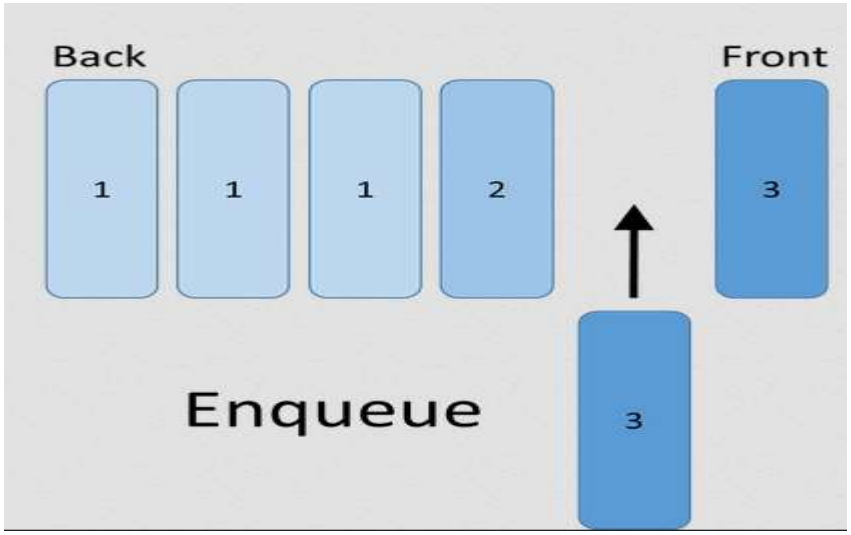
1.3.3.1 Operaciones

- *create()* Crea una cola vacía.
- *isEmpty()* Devuelve verdadero si una cola se encuentra vacía.
- *enqueueEnd():* Ingresa un elemento al final de la cola.
- *dequeueBegin()*: Elimina y devuelve el elemento que se encuentra al principio de la cola.
- ***enqueueBegin():*** Ingresa un elemento al principio de la cola.
- ***dequeueEnd()*:** Elimina y devuelve el elemento que este a final de la cola.

Otros tipos de “queues”

- **De prioridad**

Se caracteriza por admitir inserciones, consulta y eliminación de elementos con valores de prioridad ponderados, permitiendo que no necesariamente el elemento que se encuentre en la parte frontal sea el primero en ser atendido.



Listas Ligadas

- Un arreglo es una estructura de datos sumamente útil, sin embargo tiene algunas limitaciones
  - Cambiar el tamaño del arreglo.
  - En la memoria, los datos se almacenan de forma secuencial.

- Tales limitaciones pueden superarse con el uso de listas ligadas.

Listas Ligadas

CLASIFICACIÓN

- LISTAS LIGADAS SIMPLES
- LISTAS CIRCULARES (SIMPLES)
- LISTAS LIGADAS DOBLES
- LISTAS CIRCULARES DOBLES

Listas Ligadas

- Son listas que se componen de **nodos.**
- El nodo, en su forma más simple contiene un dato (información) y una referencia a otro nodo.
- Los nodos son objetos de tipo autorreferenciales, ya que el campo de la referencia apunta hacia un objeto del mismo tipo.

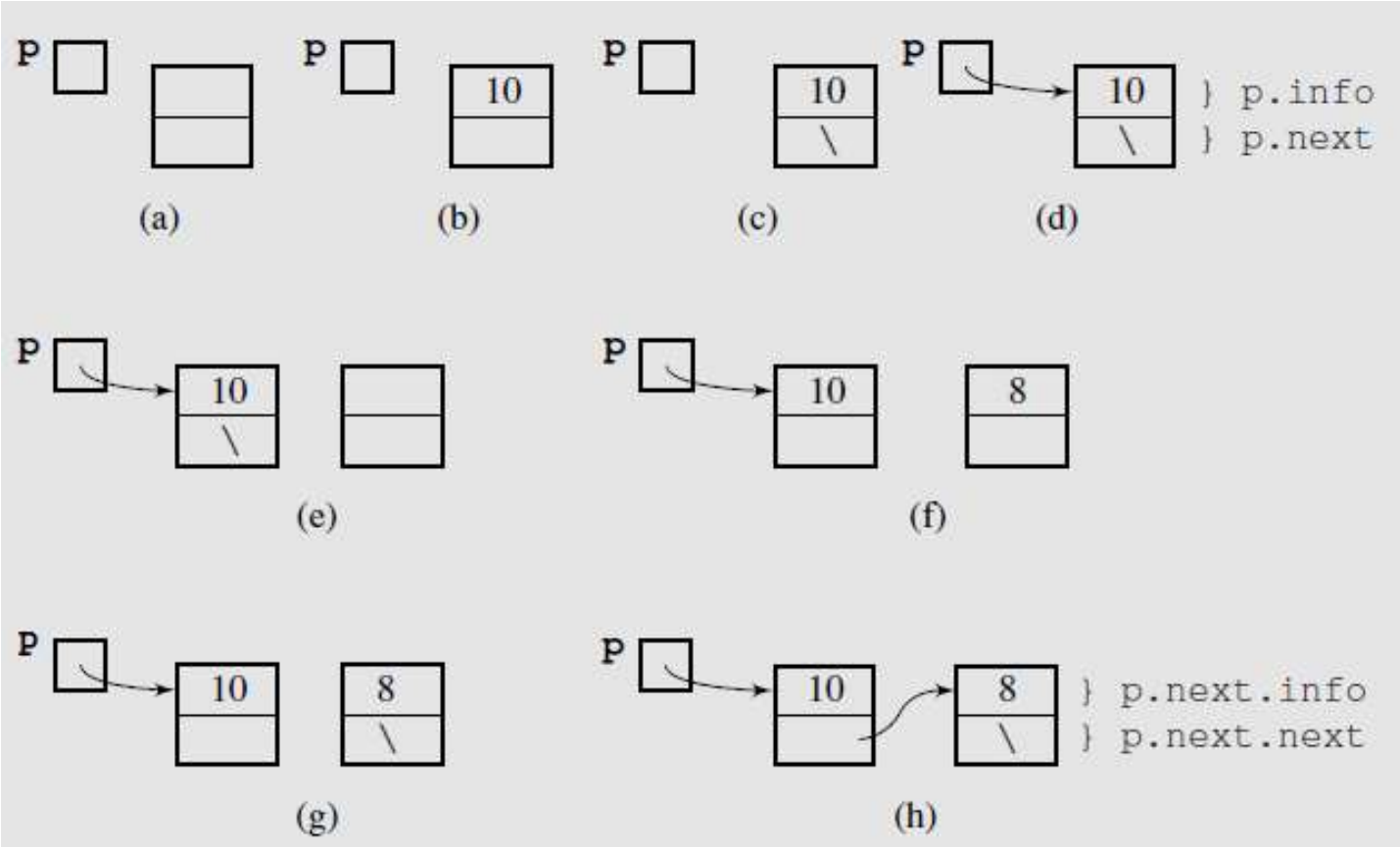
Listas Ligadas Simples

- Cada nodo tiene dos elementos, uno de ellos es la información del nodo y el otro es una referencia a otro nodo.
- Cada nodo tiene una localización en memoria aleatoria.
- Una lista ligada es la unión de varios nodos a través de sus referencias
- Con una sola referencia se puede acceder a cualquier elemento de la lista

DEFINICION TAD's

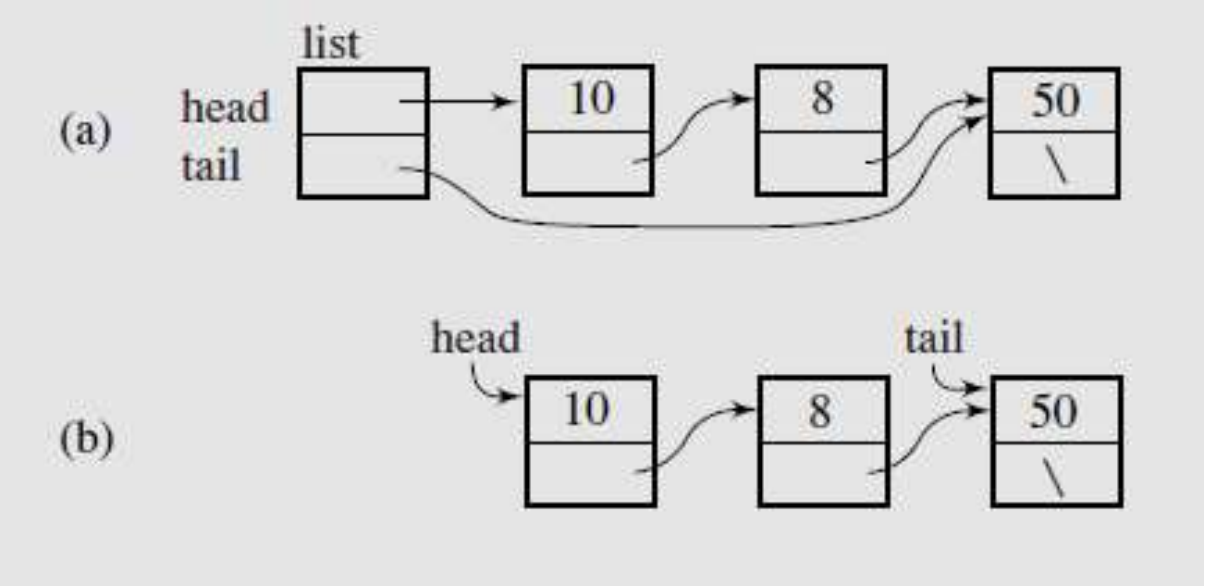
```
Nodo{
    int info;
    Nodo next;
}
ListaL{
    Nodo head;
    Nodo tail
}
```

Listas ligadas simples



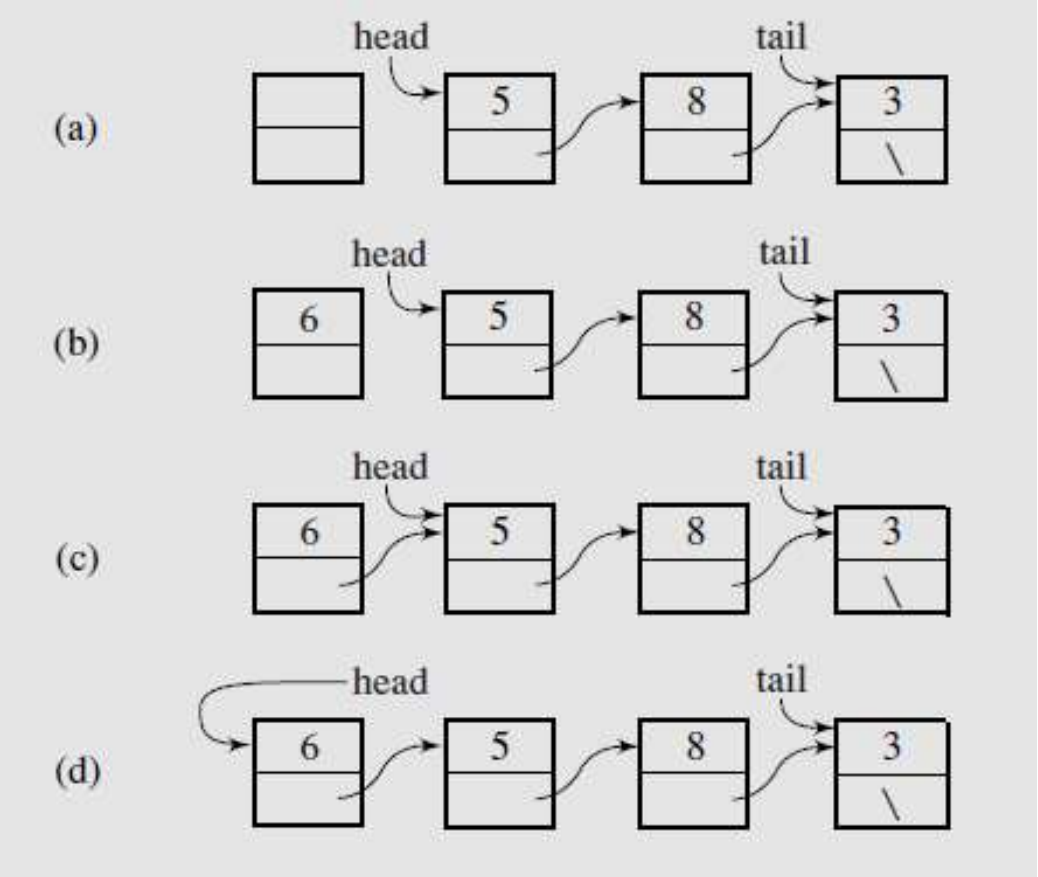
Listas ligadas simples

- El inicio y fin de una lista se especifican por los identificadores *head* y *tail*



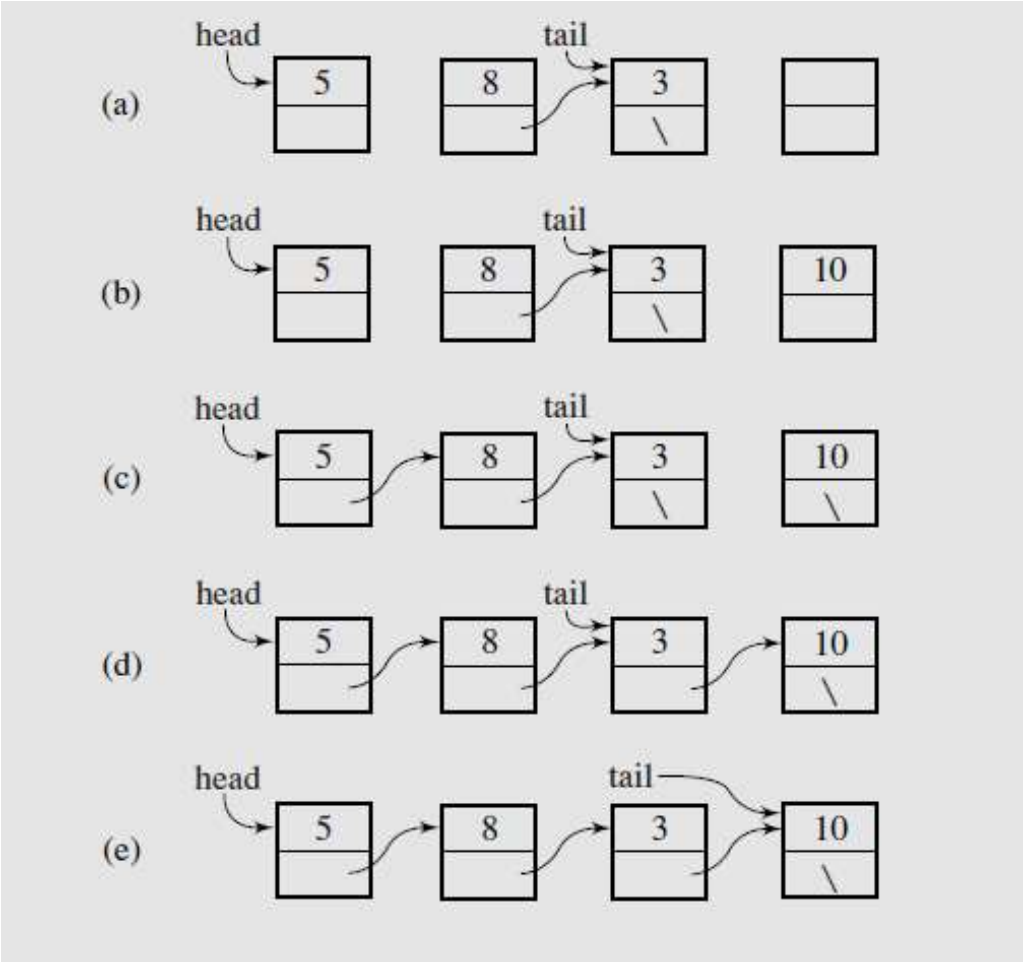
Listas ligadas simples - operaciones

- **Inserción**



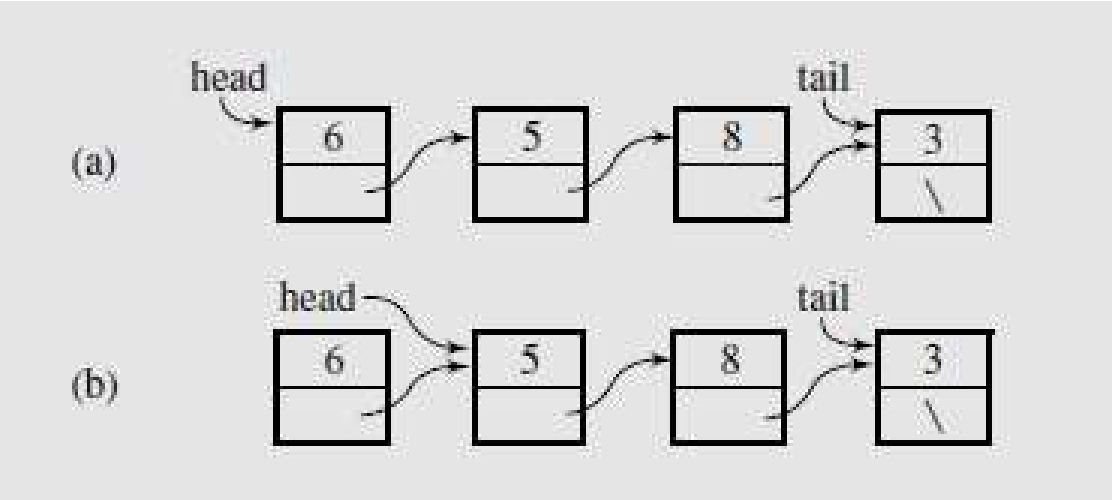
Listas ligadas simples - operaciones

- **Inserción**



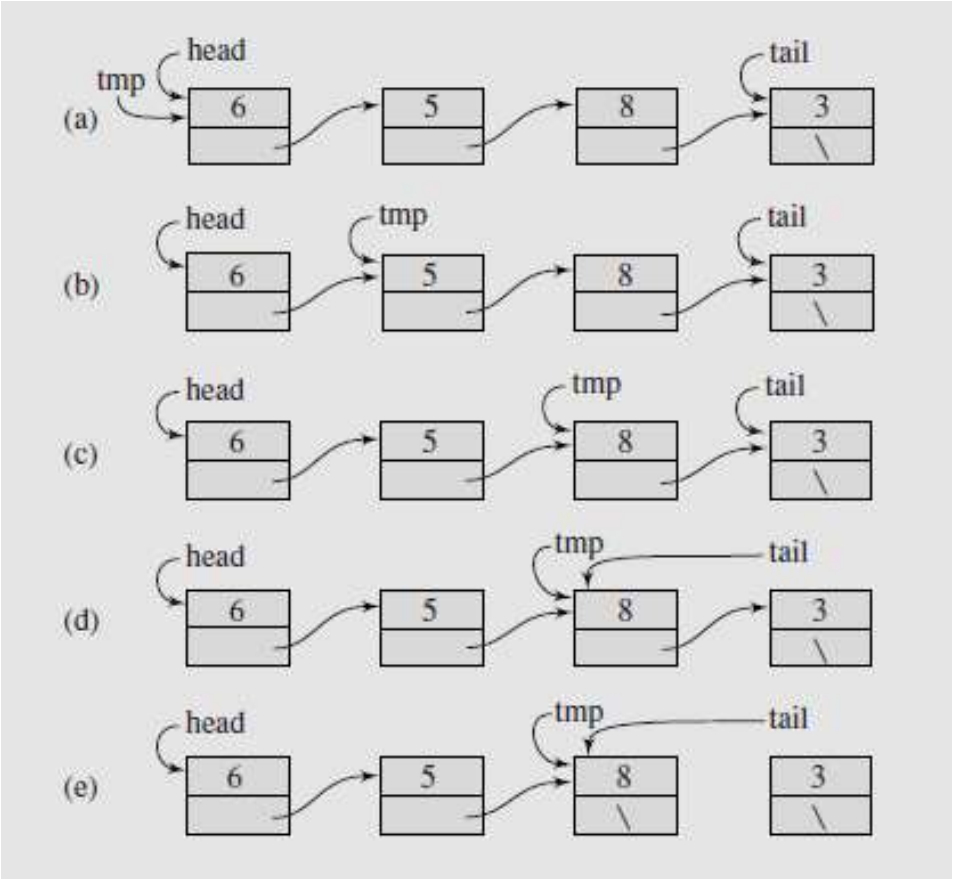
Listas ligadas simples - operaciones

- **Eliminación**



Listas ligadas simples - operaciones

- **Eliminación**



Listas ligadas simples - operaciones

- **Búsqueda**

✓Las operaciones de inserción y eliminación modifican la estructura de la lista.

✓La operación de búsqueda explora una lista existente para saber si un valor está en ella.

✓Nuevamente se utiliza una variable tmp para desplazarse por la lista.

✓El valor almacenado en cada nodo se compara con el número buscado.

✓Si los números son iguales, se sale del ciclo.

✓De lo contrario tmp se actualiza a tmp.next para investigar el siguiente nodo

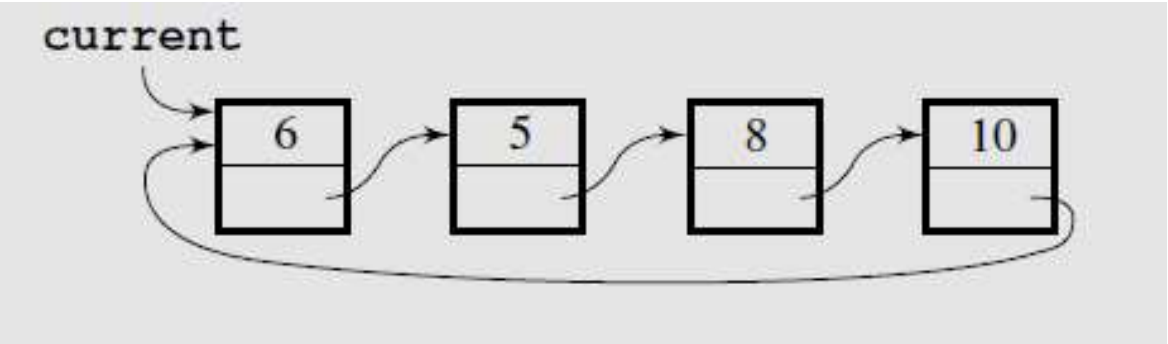
Listas Ligadas simples

- Complejidad Asintótica

Operación	Orden
Insertar	O(n)
Eliminar(n)	O(n)
Buscar(n)	O(n)

1.3.4 Listas ligadas circulares

- Es una lista ligada en la cual el sucesor del último elemento es el primero
- Todos los nodos de esta lista tienen un sucesor, el sucesor del último nodo es el primer nodo de la lista.
- Un ejemplo de esta situación es cuando varios procesos están utilizando el mismo recurso durante la misma cantidad de tiempo y se requiere asegurar que cada proceso tenga una parte justa del recurso

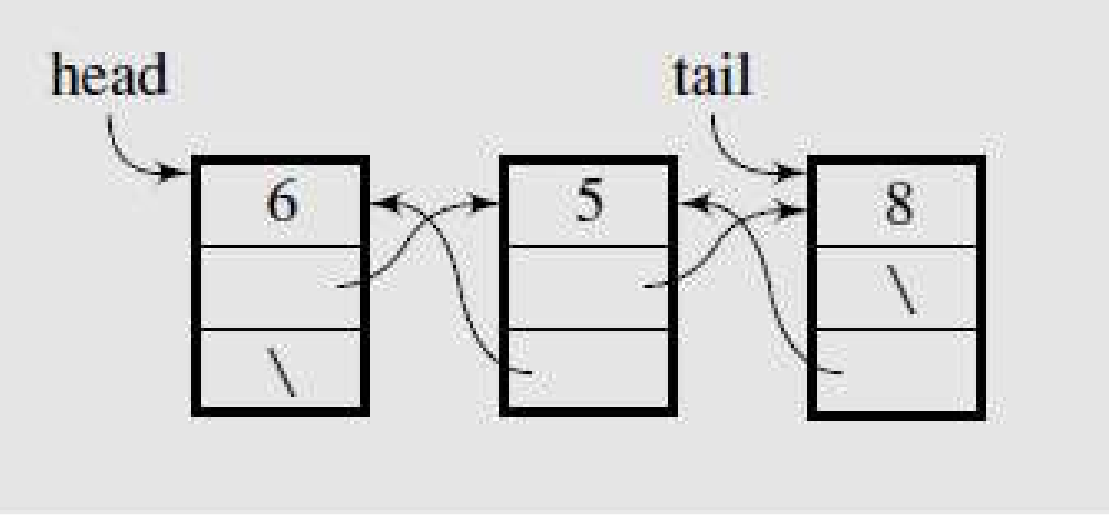


DEFINICION TAD's

```
ListaLCircular{
    int size;
    Nodo head;
}
```

1.3.5 Listas ligadas dobles

- Es una redefinición de la estructura nodo de modo que ahora tiene dos campos de referencia uno para el sucesor y uno para el predecesor

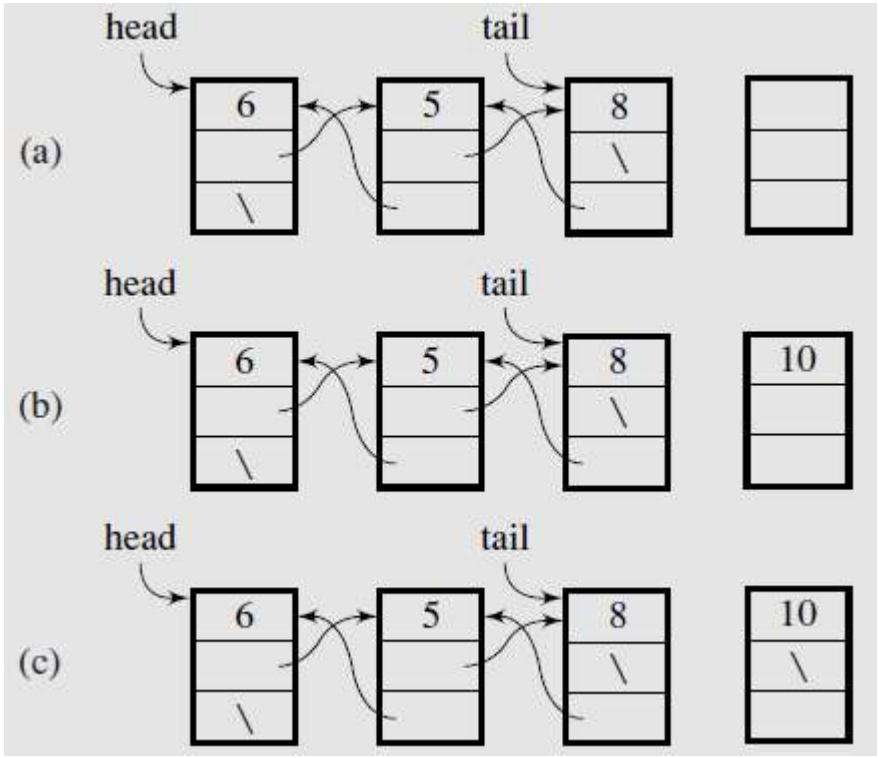


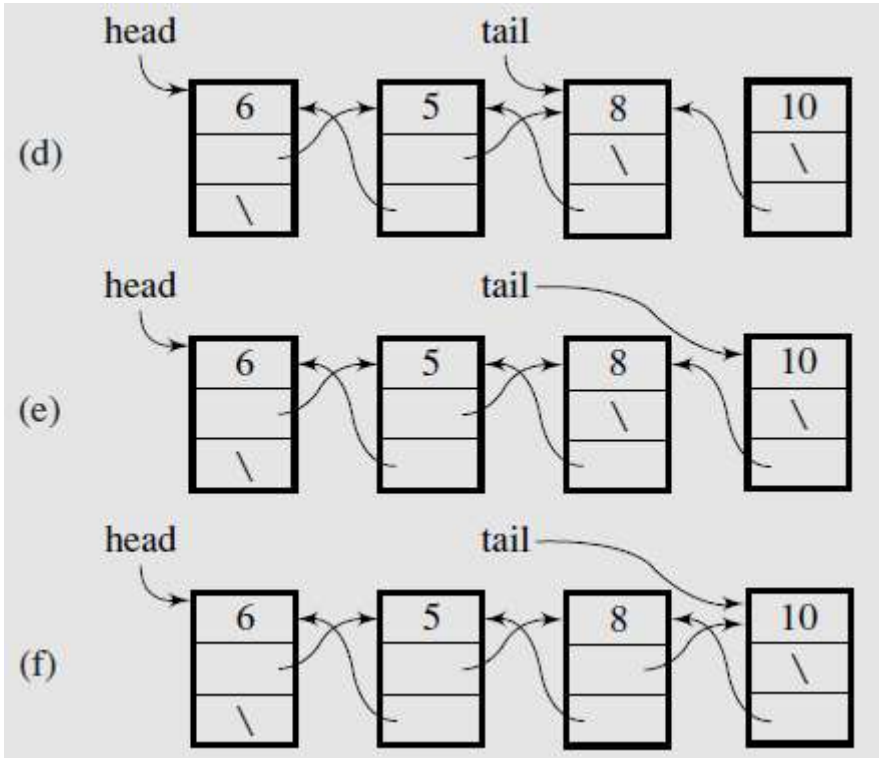
DEFINICION TAD's

```
ListaLDoble{
    int size;
    NodoD head;
    NodoD tail
}
```

1.3.5 Listas ligadas dobles

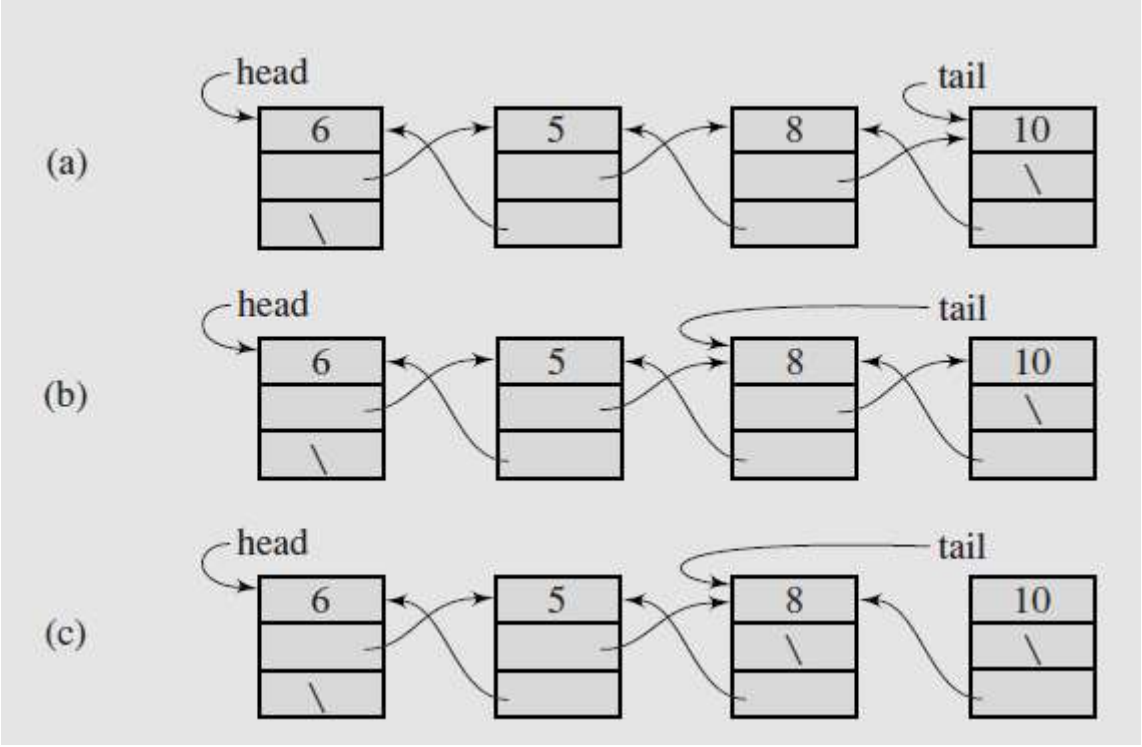
- Agregar nodo



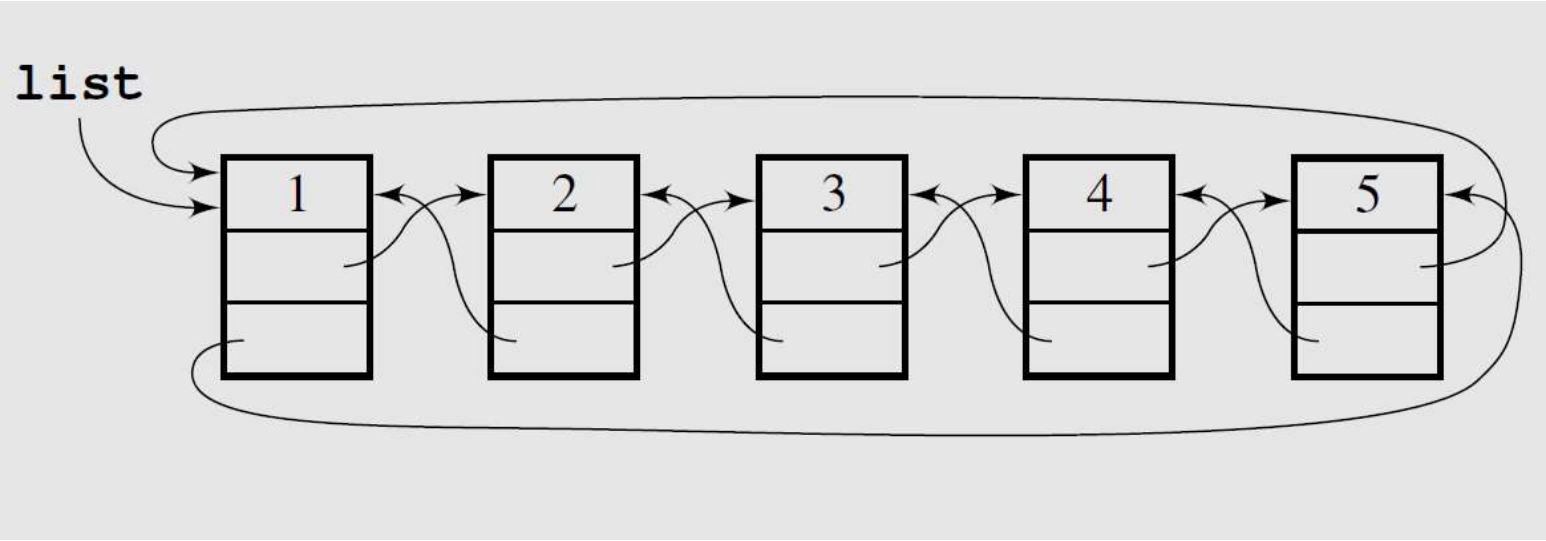


1.3.5 Listas ligadas dobles

- Eliminar nodo al final



1.3.5 Lista ligada circular doble



Encapsulando

```
void agregarInicio(Lista *lista,int x){
     Nodo nuevo
     nuevo.info=x
     nuevo.next=lista.head;
     lista.head=nuevo
}
void agregarFinal(Lista *lista, int x){
     Nodo tmp=lista.head;
     while(tmp.next.next!=Null)
          tmp=tmp.next;
     Nodo nuevo;
     nuevo.info=x
     nuevo.next=NULL;
     tmp.next=nuevo
}
```

Encapsulando

```
int Busqueda(int x, Lista lista){
    int encontrado=0;
    Nodo tmp=lista.head
    while(tmp!=null){
        if(tmp.info==x){
            encontrado=1;
            break;
        }
        else
            tmp=tmp.next;
    }
    return encotrado;
}
```

Encapsulando

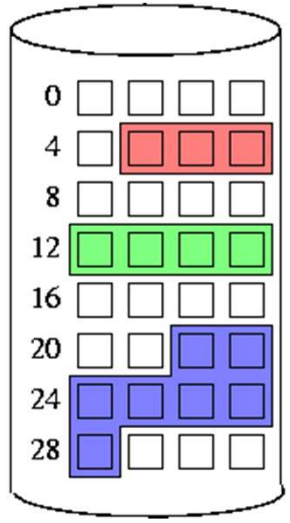
- void addNesimo(Lista lista, int posicion, int x)
- void addMedio(Lista *lista, int x)
- int BuscaPosicion(Lista lista, int x)
- int BuscaApariciones(Lista lista, int x)
- void eliminarInicio(Lista *lista)
- void eliminarFinal(Lista *lista)
- void eliminarNesimo(Lista *lista, int posicion)
- void invertirLista(Lista *lista)

Consideraciones sobre almacenamiento contiguo y ligado

- Cuando se maneja el *almacenamiento contiguo*, se requiere que cada elemento almacenado ocupe un conjunto de direcciones contiguas en el disco, su asignación es definida por la dirección del primer bloque de memoria y la longitud del elemento.
- Este tipo de almacenamiento resulta limitado en el sentido de que una vez definido no se puede incrementar o disminuir.

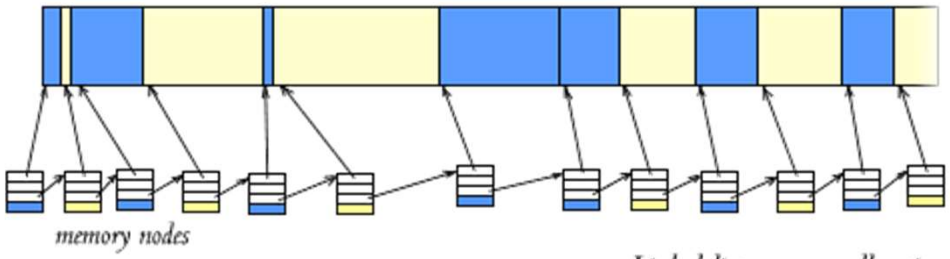
Consideraciones sobre almacenamiento contiguo y ligado

- La asignación contigua presenta algunos problemas, como la fragmentación externa. (la variabilidad de los espacios de memoria disponibles).
- Lo cual, hace difícil encontrar bloques contiguos de espacio de tamaño suficiente.



Consideraciones sobre almacenamiento contiguo y ligado

- En el caso del almacenamiento ligado o encadenado. Cada elemento se maneja enlazando “bloques de memoria” y se cuenta con apuntadores al primer y al último elemento.
- La asignación se hace con bloques individuales, cada bloque contendrá un puntero al siguiente bloque de la cadena.



Linked list memory allocation

Consideraciones sobre almacenamiento contiguo y ligado

- El registro de asignación de memoria requiere una sola entrada por cada elemento almacenado que muestre el bloque de comienzo y la longitud del mismo, cualquier bloque puede añadirse a la cadena.
- No es necesario preocuparse por la fragmentación externa porque solo se necesita un bloque cada vez. Esto hace el almacenamiento más flexible y con un mejor aprovechamiento de la memoria