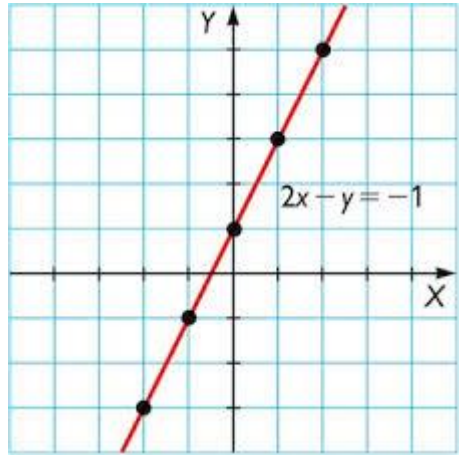
1.7 Estructuras de datos Lineales

1.7.1 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.



1.7.1 Generalidades

- Se puede aumentar o disminuir la cantidad de datos y podrán insertarse o eliminarse elementos en determinadas posiciones sin alterar su estructura lógica.

1.7.1.1 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

1.7.1.1 El concepto de Lista

- Una lista tiene operaciones de inserción, eliminación y búsqueda
- Una lista es una estructura de datos flexible en las operaciones que se realizan

1.7.2 Pila (Stack)

1.7.2 Pila

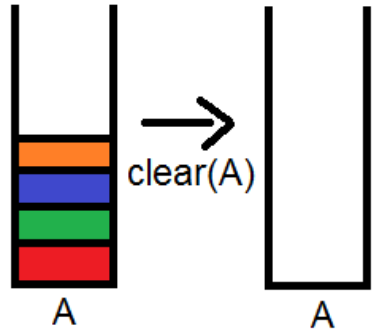
- Una pila (stack) es un tipo abstracto de datos tipo LIFO (*last in, first out*).
- Es un tipo de dato que soporta operaciones de inserción y eliminación de elementos de un conjunto.
- Normalmente se usa en aplicaciones en las que se necesita recuperar información en orden inverso a como ha entrado.

1.7.2 Pila

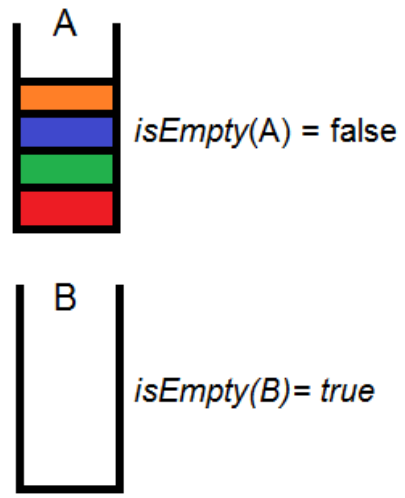
- Esta estructura funciona de forma similar a una torre de elementos del mismo tipo (documentos pendientes, platos, libros, etc).
- Cada vez que llega un elemento, se coloca arriba de los que ya están acumulados, y cuando se van atendiendo se toma el que se encuentra en la parte superior.

Operaciones (Pila)

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

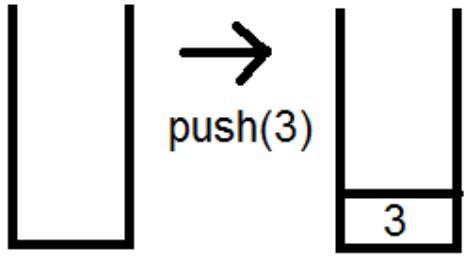


  - *isEmpty()* Verifica si una pila se encuentra vacía

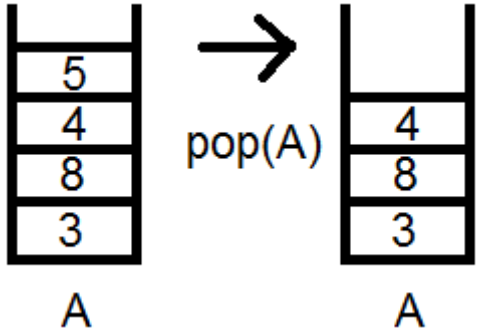


Operaciones (Pila)

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

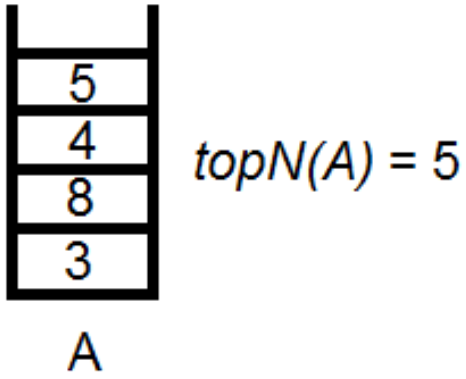


- *pop() Retira el elemento superior*



Operaciones (Pila)

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



Operaciones (Pila)

- 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)

Implementación (Pila)

```
Pila:
     int tope;
     Lista elementos;

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

Implementación (Pila)

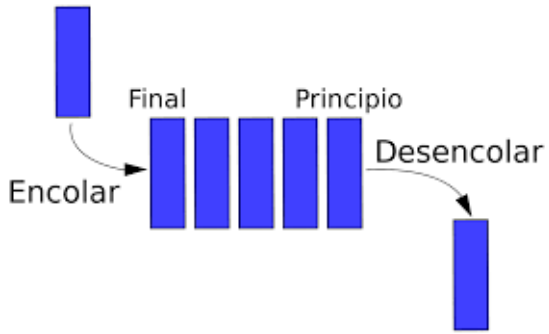
```
bool esVacia(Pila p)          //isEmpty(..)
     if(p.tope==-1)
          return true;
     return false;

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

Implementación (Pila)

```
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
```

Implementación (Pila)

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



1.7.3 Cola (Queue)

1.7.3 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

1.7.3 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 estructura

Operaciones (Queue)

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

Operaciones (Queue)

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

Operaciones (Queue)

- 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)

Implementación (Queue)

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

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

Implementación (Queue)

```
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
```

Implementación (Queue)

```
int desencolar(Queue 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
```

1.7.3.4 Cola Circular

- Los elementos están de forma “circular”, lo que implica que todos los elementos tienen sucesor y 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 el tamaño de la estructura
- La ventaja de una Cola circular es que el manejo de memoria es más eficiente

1.7.3.4 Cola Circular (operaciones)

- Las operaciones con la cola circular, son las mismas que en una “queue” simple (*enqueue, dequeue, first, etc*).
- El usuario no ve diferencia cuando utiliza una u otra
- **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)

1.7.3.5 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.

Cola Doble (dequeue)

Operaciones (Cola doble)

- *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.

Cola de Prioridad

Se caracteriza por admitir inserciones, consulta y eliminación de elementos con valores de prioridad ponderados.

Cuando ingresa un elemento se coloca en la última posición entre aquellos con su misma prioridad

Aplicaciones (Pila)

- 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 requiere hacer una lectura de la expresión revisando cada uno de los caracteres que la conforman

Aplicaciones (Pila)

Dada una expresión aritmética (leyendo de izquierda a derecha)

- Ignorar los paréntesis de apertura
- Operando: **Push(opn)** en stack de operandos
- Operador: **Push(opr)** 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.

Aplicaciones (Pila)

- 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).

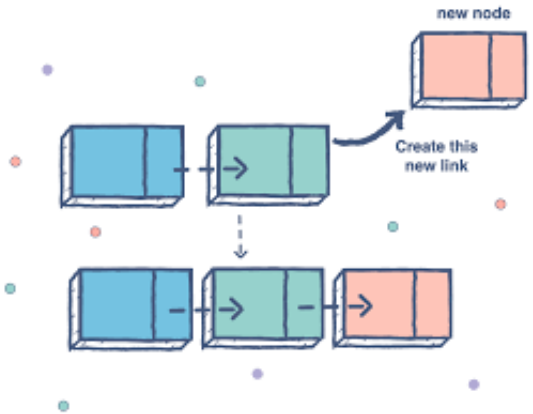
Aplicaciones (Pila)

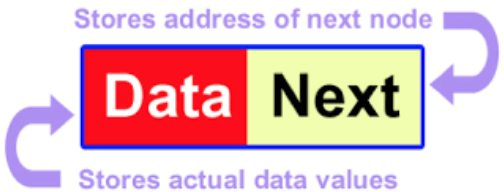
Sea una expresión aritmética (leyendo de izquierda a derecha)

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

Aplicaciones (Queue)

- 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.





1.7.4 Listas Ligadas

1.7.4 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.

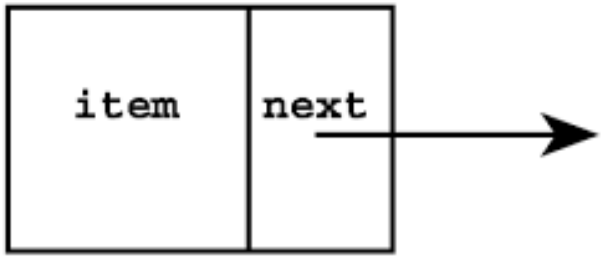
1.7.4 Listas Ligadas

Clasificación

- Listas ligadas simples
- Listas circulares (simples)
- Listas ligadas dobles
- Listas circulares dobles

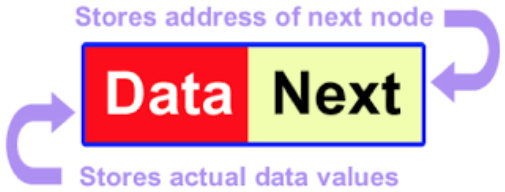
1.7.4.1 El concepto de nodo

- Es el elemento esencial para conformar estructuras de almacenamiento no secuencial
- 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 en su definición se contienen a sí mismos (a través de una referencia).



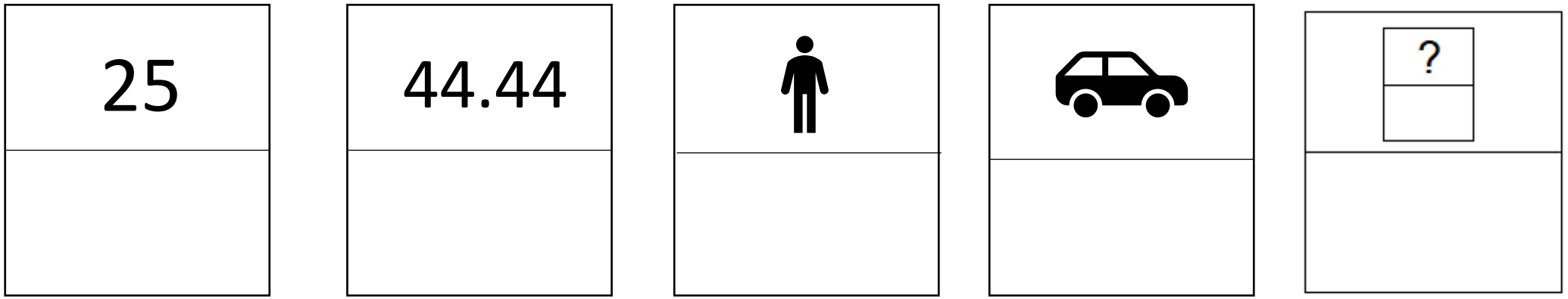
1.7.4.1 El concepto de nodo

- 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



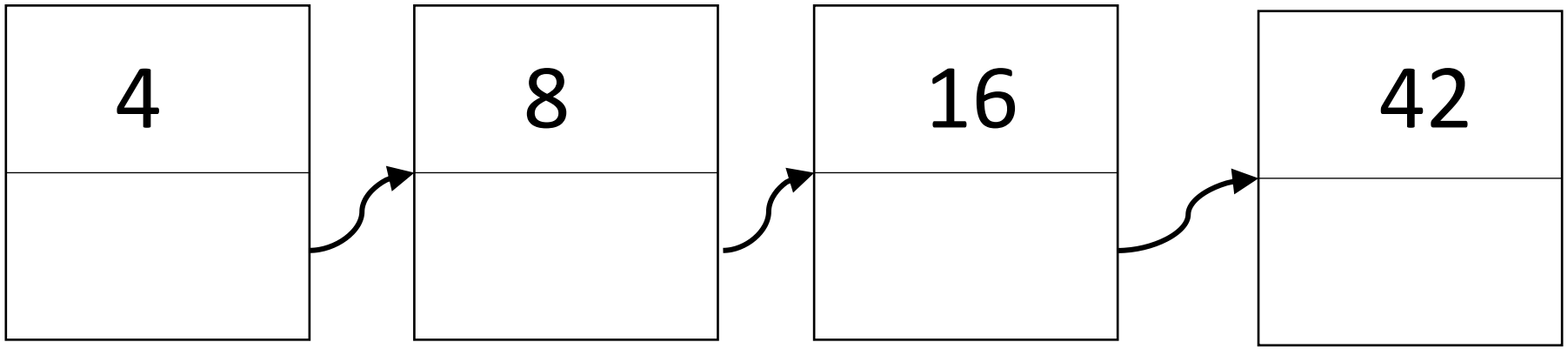
1.7.4.1 El Nodo como tipo de dato abstracto

```
struct Nodo{
    int info;
    struct Nodo next*;
}
```



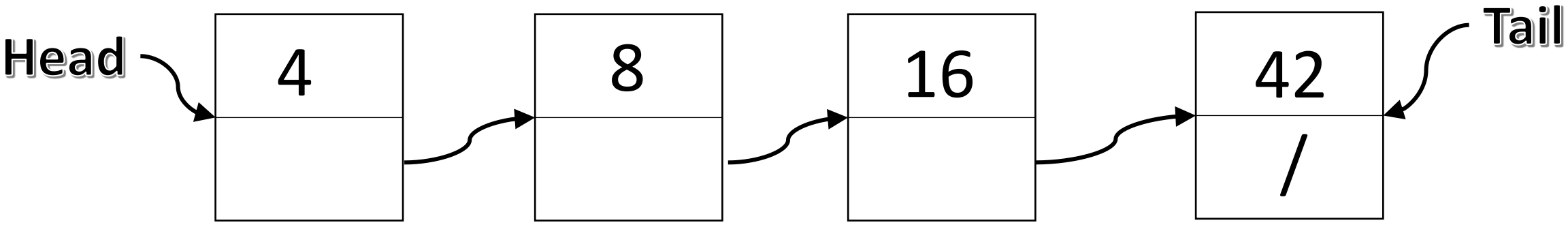
1.7.4.2 Lista simple y operaciones

- Las listas ligadas simples conforman la versión mas “sencilla” de almacenamiento ligado
- Se conforman de nodos que solamente cuentan con una referencia hacia el nodo siguiente

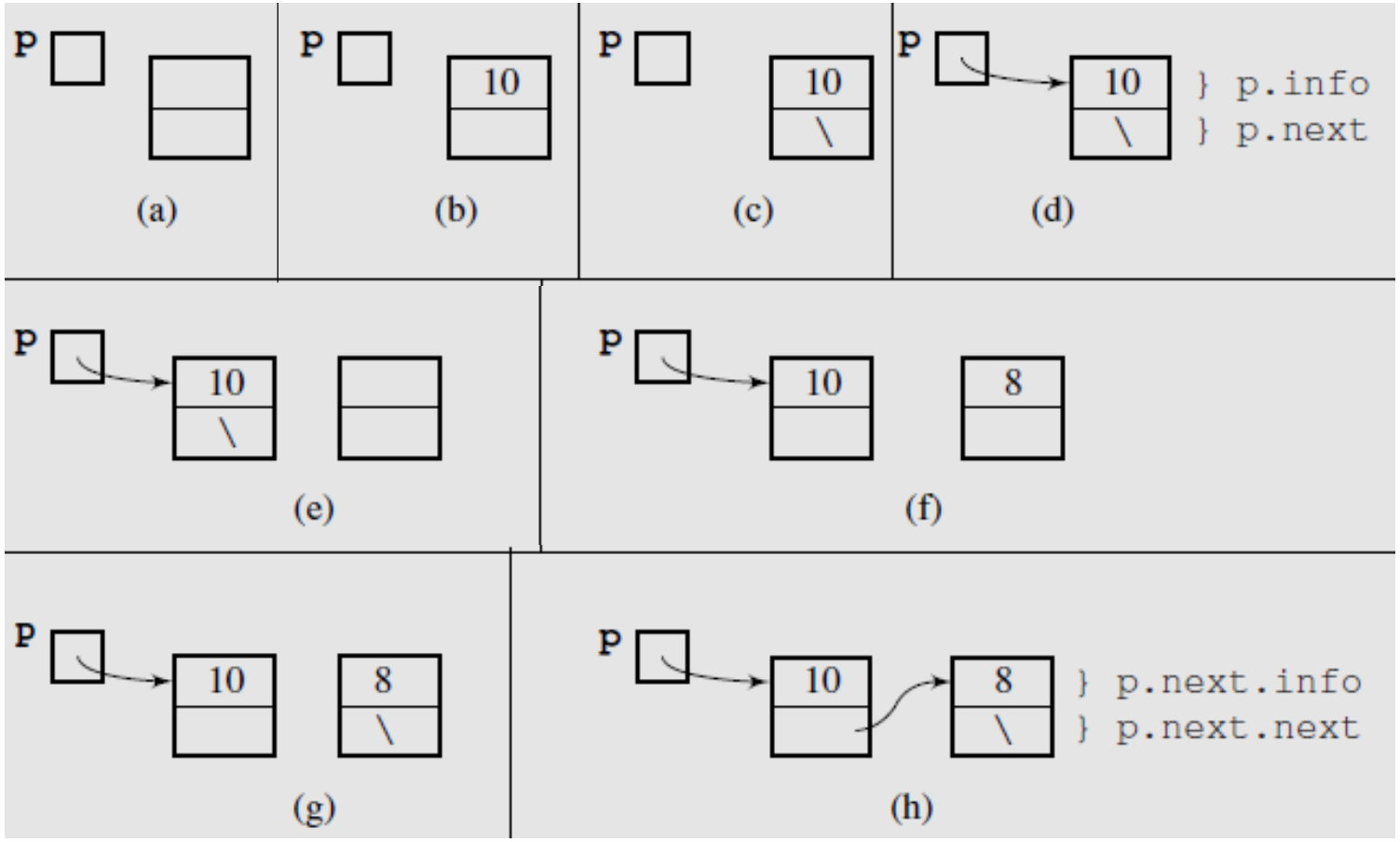


1.7.4.2 Lista simple y operaciones

- La lista se puede trabajar con una sola referencia (Head) o con dos referencias de control (Head y Tail)
- La lista termina cuando la referencia hacia el nodo siguiente tiene un valor nulo.

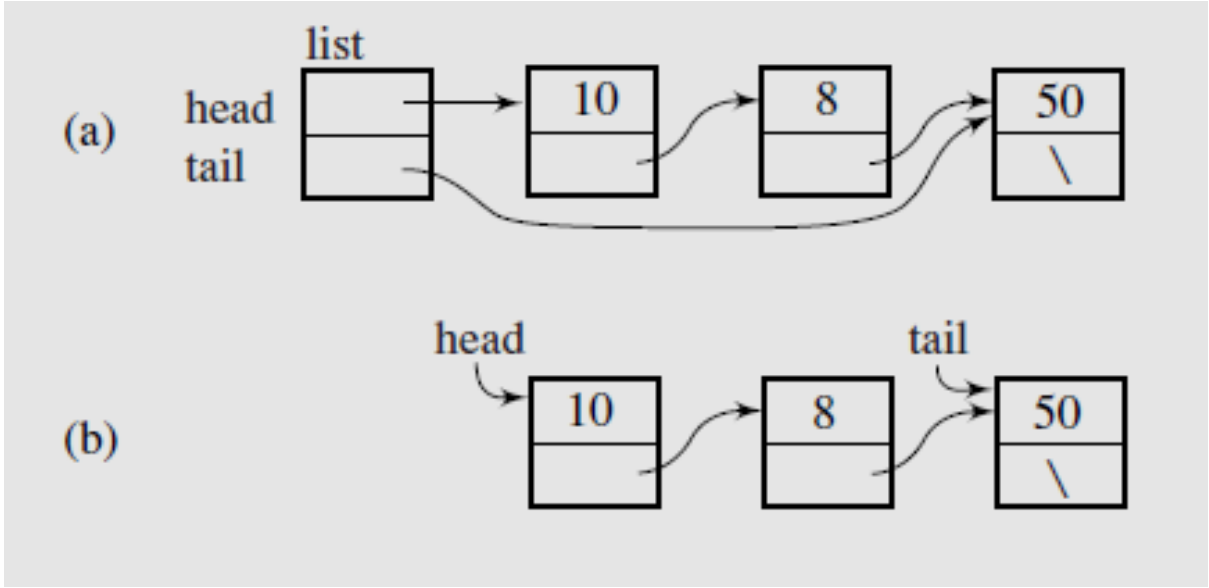


1.7.4.2 Lista simple y operaciones



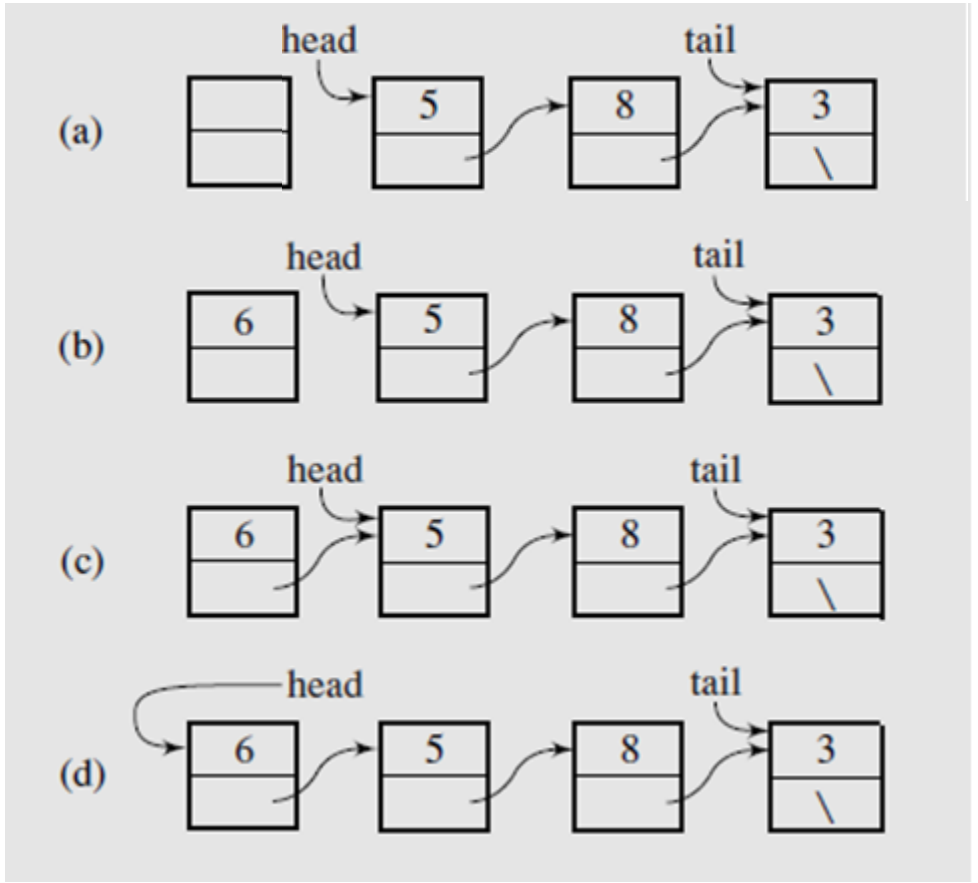
1.7.4.2 Lista simple y operaciones

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



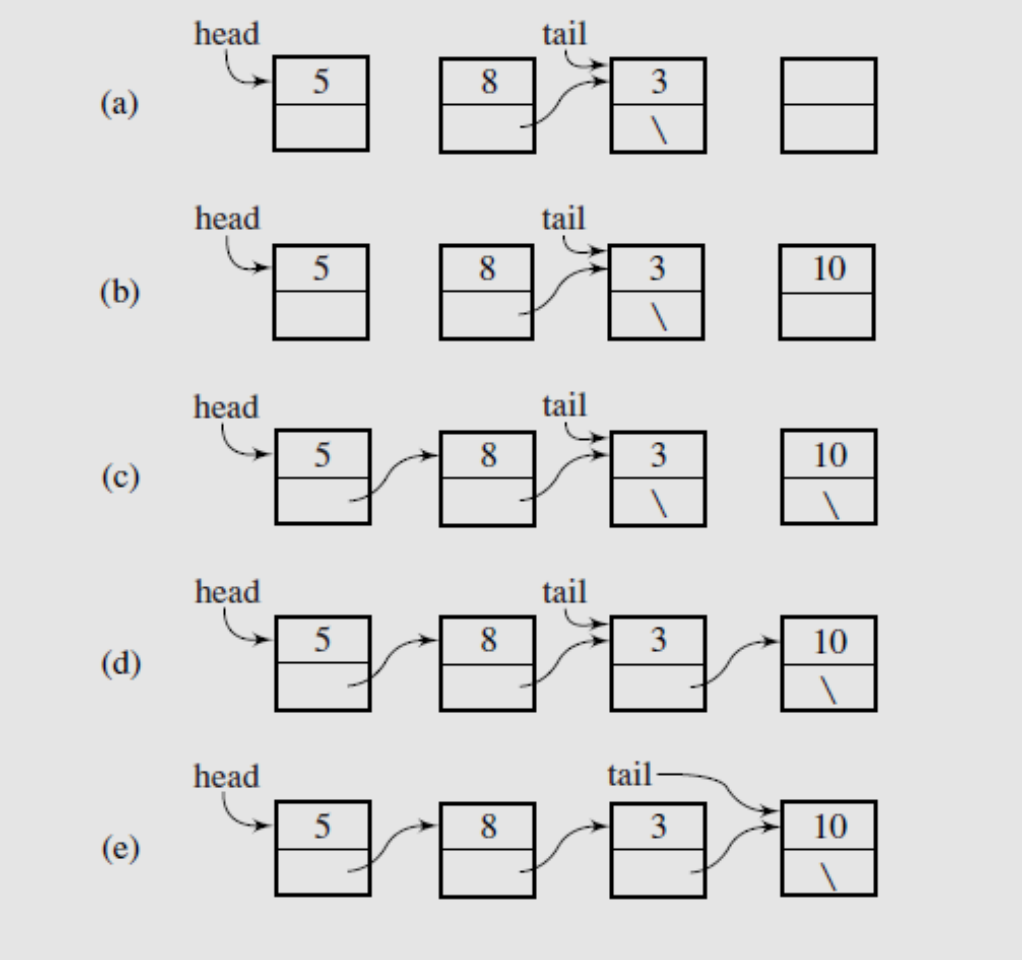
1.7.4.2 Lista simple y operaciones

- **Inserción al inicio**



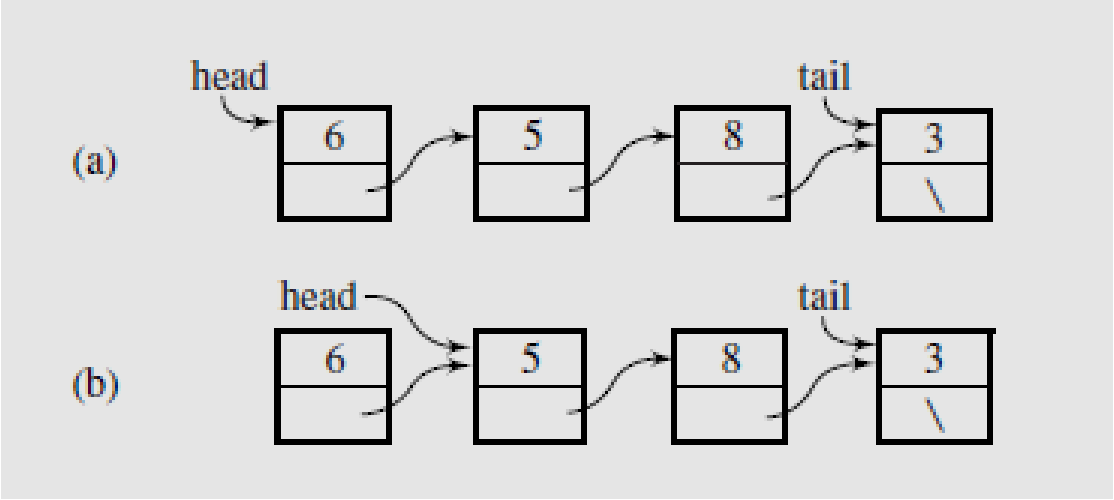
1.7.4.2 Lista simple y operaciones

- **Inserción al final**



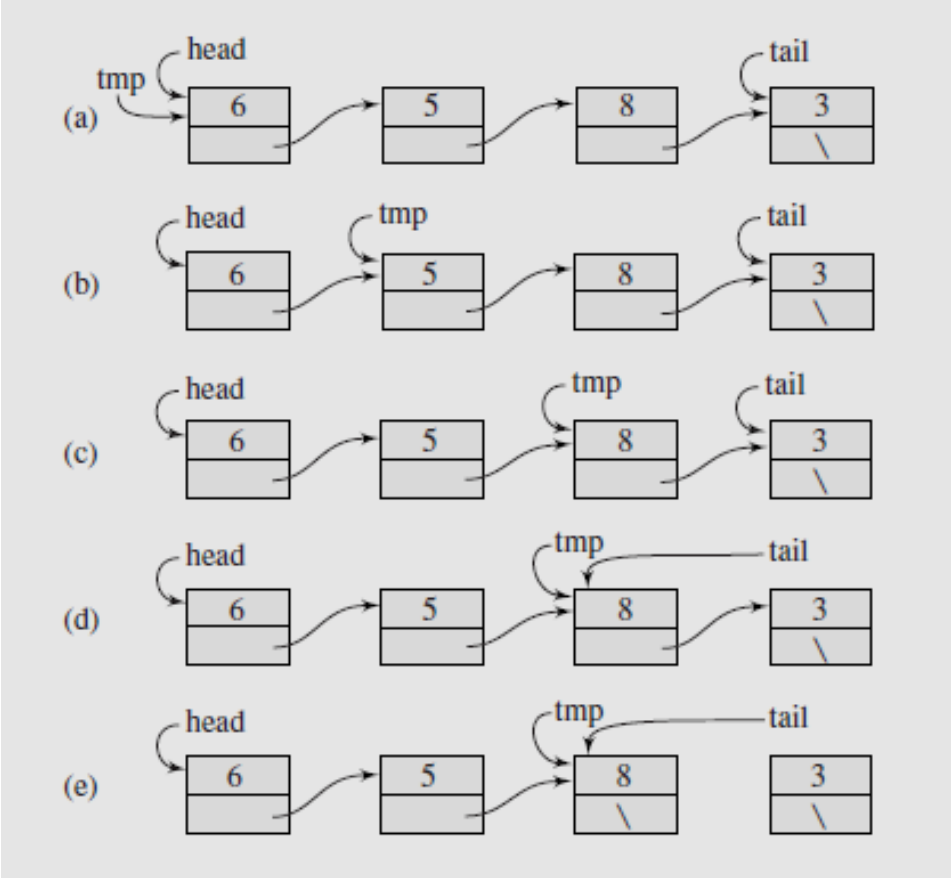
1.7.4.2 Lista simple y operaciones

- **Eliminación al inicio**



1.7.4.2 Lista simple y operaciones

- **Eliminación al final**



1.7.4.2 Lista simple y 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.
- Para esta operación se utiliza una referencia **“tmp”** para desplazarse por la lista.

1.7.4.2 Lista simple y operaciones

Búsqueda

- El valor almacenado en cada nodo (info) se compara con el dato buscado.
- Si los números son iguales, se dice ha encontrado el valor y se puede terminar el procedimiento
- De lo contrario tmp se actualiza a tmp.next para investigar el siguiente nodo

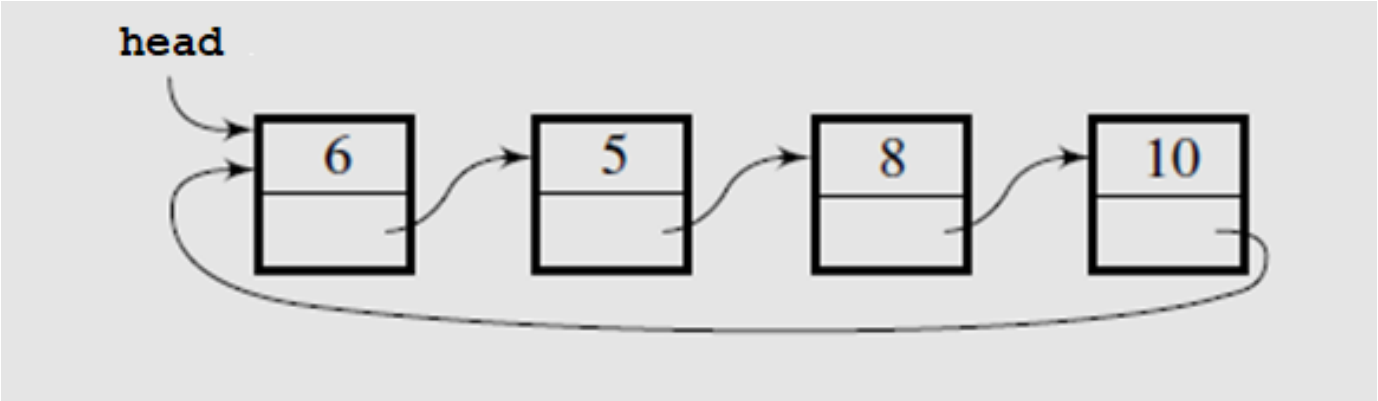
1.7.4.2 Lista simple y operaciones

- Complejidad Asintótica

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

1.7.4.3 Lista ligada circular

- 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.

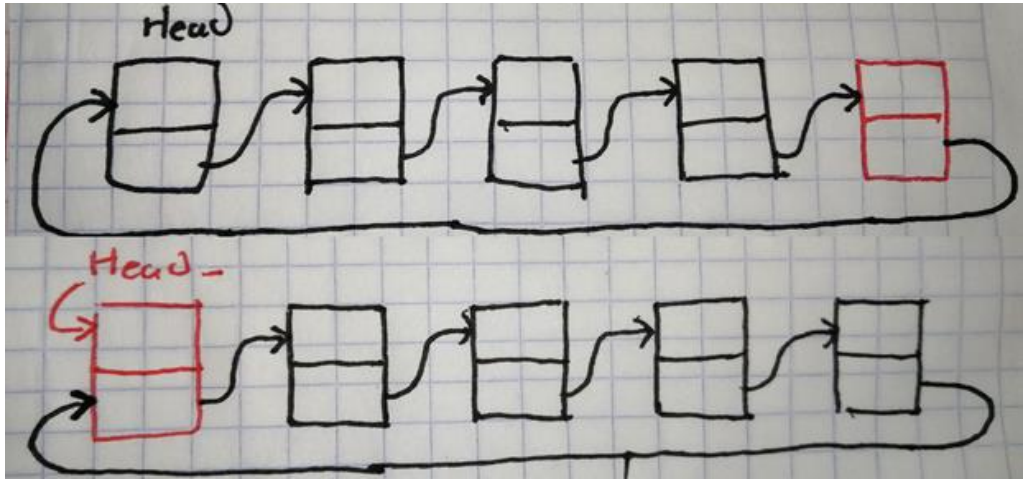


1.7.4.3 Lista ligada circular

- 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
- Las listas ligadas circulares se utilizan para implementar colas circulares (simples y dobles)

1.7.4.3 Lista ligada circular - operaciones

- Añadir principio / final




```
nodo tmp
tmp=head
while (tmp.next!=head){
    tmp = tmp.next
}
nodo nuevo
nuevo.info=20
nuevo.next=tmp.next
tmp.next=nuevo
head=tmp.next   //add inicio
```

1.7.4.4 Lista doble y operaciones

Es una lista ligada en la que cada nodo tiene dos referencias:

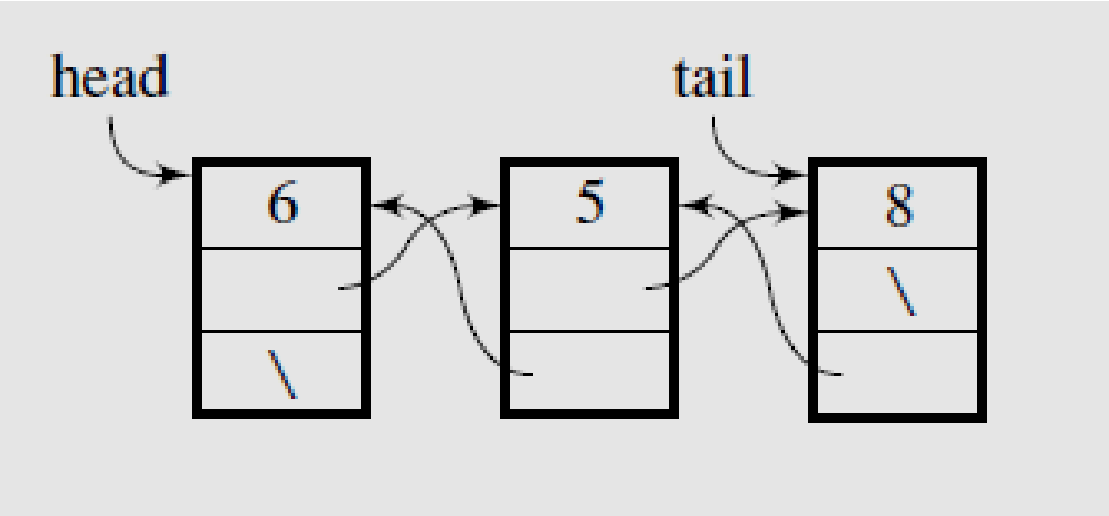
- Hacia el nodo siguiente
- Hacia el nodo anterior

En este tipo de lista sí es posible desplazarse hacia atrás

```
struct Nodo{
        int info;
        struct Nodo next*;
        struct Nodo prev*;
}
```

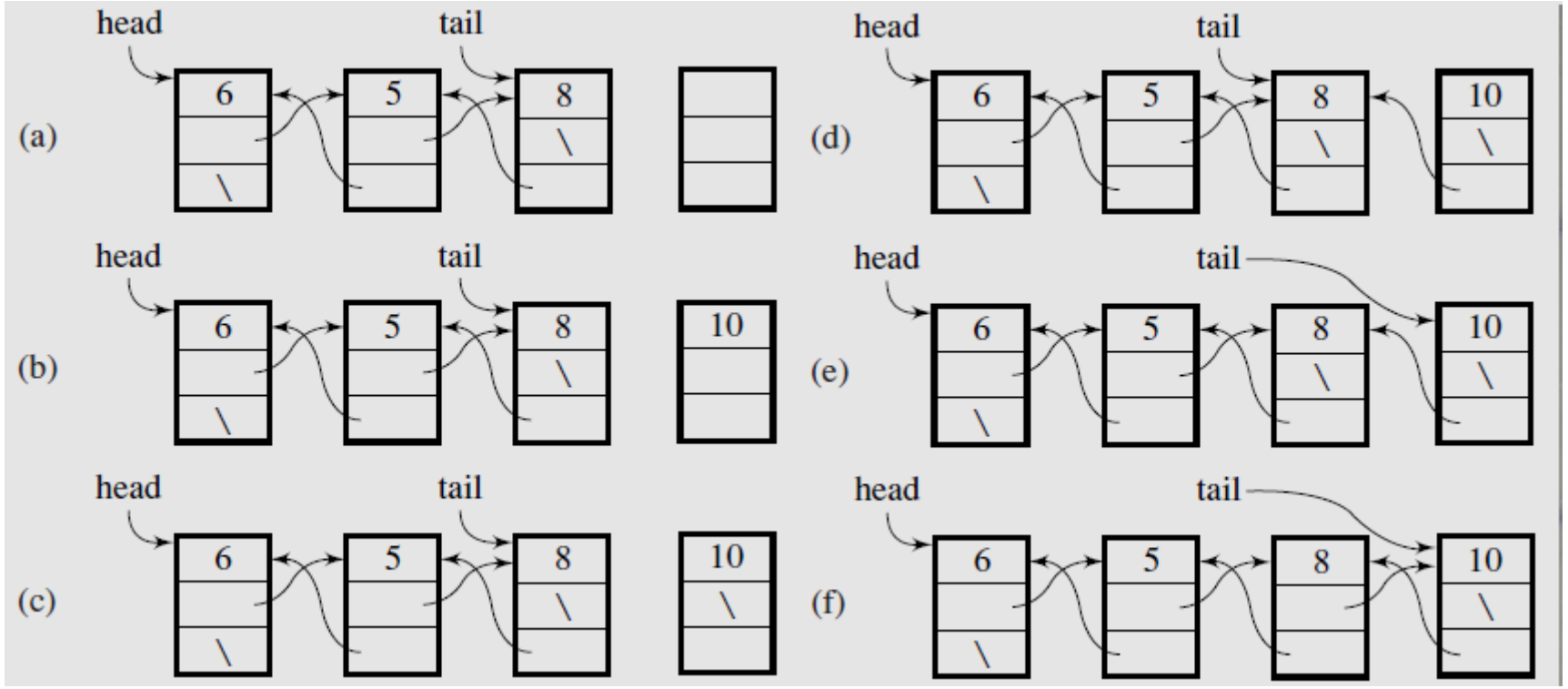
1.7.4.4 Lista doble y operaciones

Lista Ligada doble



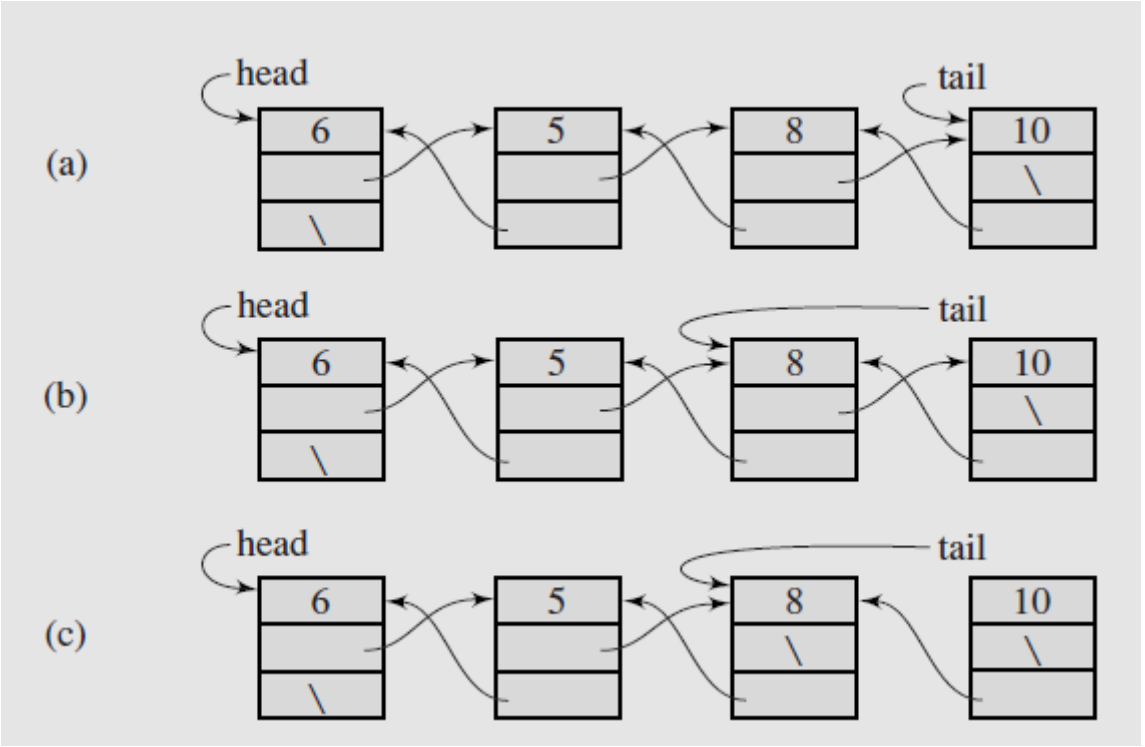
1.7.4.4 Lista doble y operaciones

- Agregar nodo al final



1.7.4.4 Lista doble y operaciones

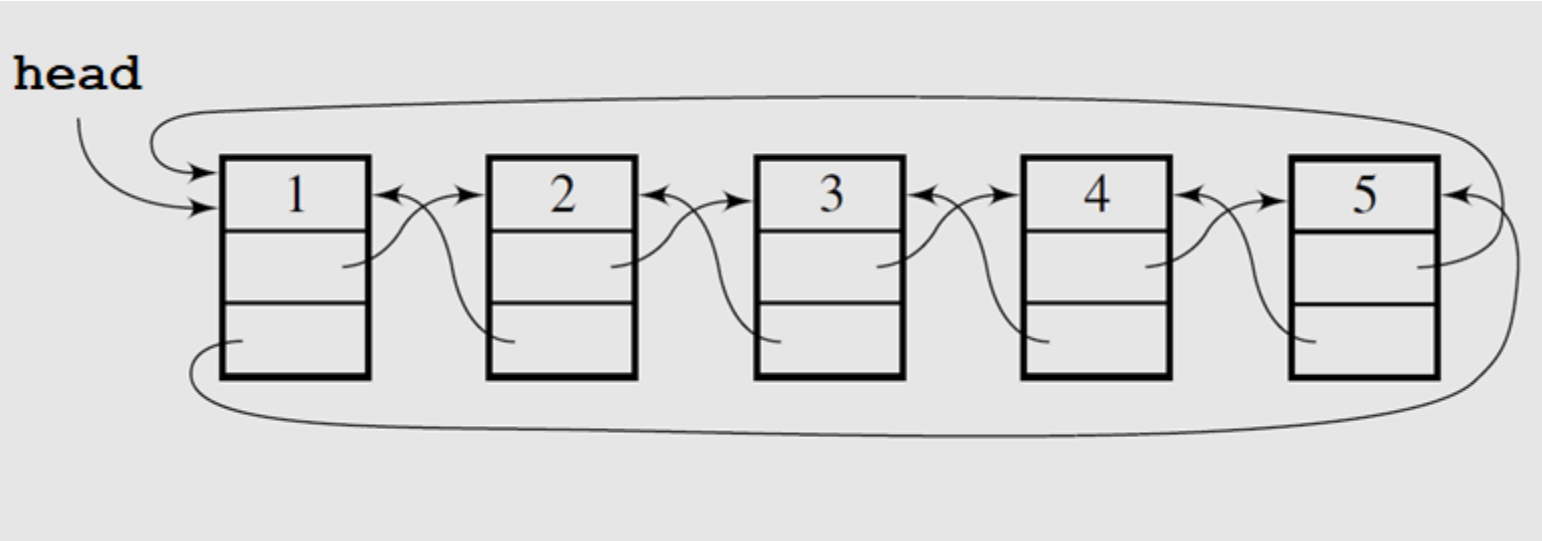
- Eliminar nodo al final



1.7.4.5 Lista ligada circular doble

Es una lista ligada en la que cada nodo tiene dos referencias

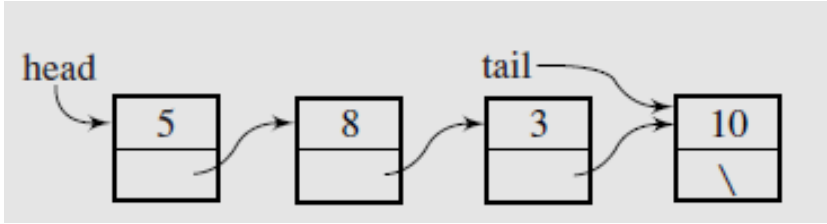
No existe ninguna referencia nula, el siguiente del último es el primero y el anterior del primero es el último

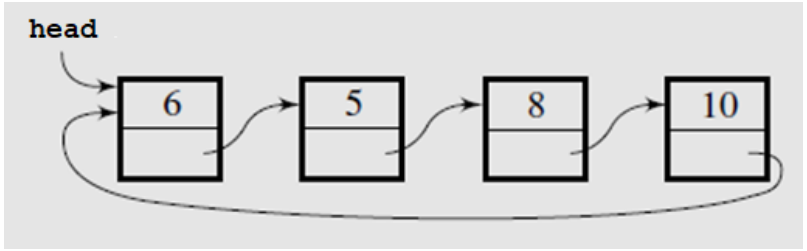


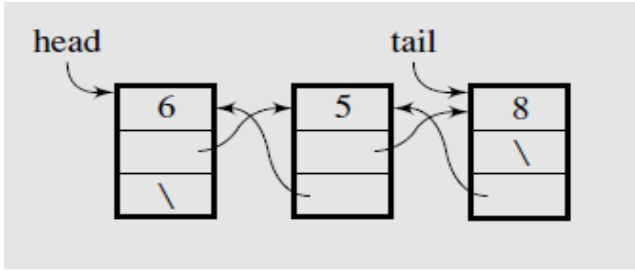
1.7.4 Listas Ligadas

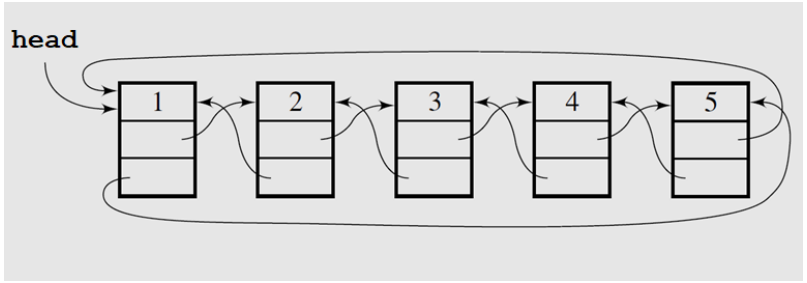
Clasificación

- **Listas ligadas simples**
- **Listas circulares (simples)**
- **Listas ligadas dobles**
- **Listas circulares dobles**









1.7.4 Listas Ligadas

Operaciones en las listas

- **Inserción**
- **Eliminación**
- **Búsqueda**

Se pueden realizar en cualquier tipo de lista ligada.

✓ **Recorridos (Es lo que cambia en las diferentes versiones de las listas ligadas)**

Definición de Listas como TDA

```
struct ListaLigada{
        int size;
        struct Nodo head;
        struct Nodo tail;
}
```

- void addPrincipio(ListaLigada *miLista, int x)
- void addFinal(ListaLigda *miLista, int x)
- int BusquedaBasica(ListaLigada miLista, int x)

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!=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 encontrado;
}
```

Encapsulando

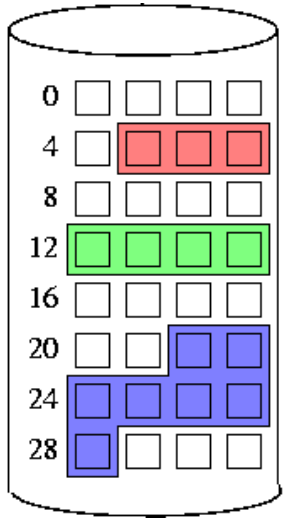
- void addNesimo(Lista lista, int posicion, int x)
- void addMedio(Lista *lista, int x)
- int BuscaPosicion(Lista lista, int x)
- int BuscaApariciones(Lista lista, int x)
- void eliminarInicio(Lista *lista)
- void eliminarFinal(Lista *lista)
- void eliminarNesimo(Lista *lista, int posicion)
- void invertirLista(Lista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

1.7.9 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.

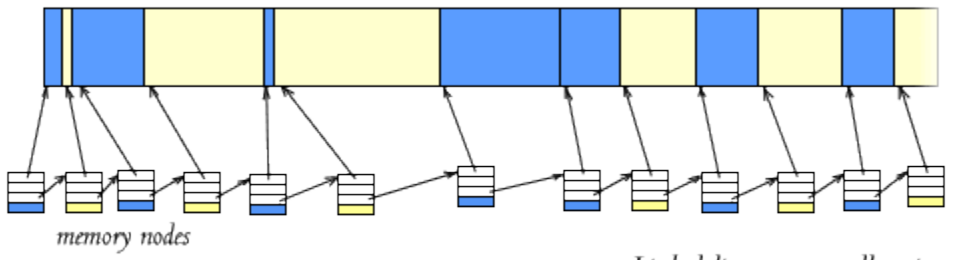
1.7.9 Consideraciones sobre almacenamiento contiguo y ligado

- 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.



1.7.9 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

1.7.9 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