TEMA 3: TIPO DE DATOS ABSTRACTO *LISTA ENLAZADA*

ALGORITMOS Y ESTRUCTURAS DE DATOS

M. Colebrook Santamaría
J. Riera Ledesma

Objetivos

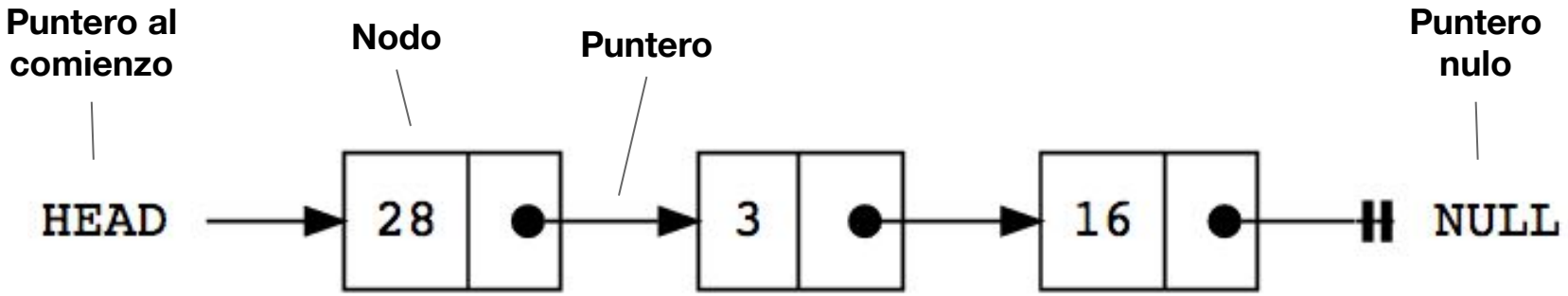
- Especificación formal del TDA lista (simplemente) enlazada.
- Implementación del TDA lista enlazada mediante objetos dinámicos.
- Operaciones sobre listas (simplemente) enlazadas:
  - Inserción
  - Extracción
  - Recorrido
  - Búsqueda
- Implementación de una lista ordenada.

Especificación formal del TDA lista (simplemente) enlazada (1)

- Un **Tipo de Datos Abstracto** ó **TDA** (del inglés ADT, *Abstract Data Type*) está caracterizado por:
  - Un conjunto de **operaciones** (métodos): representa el comportamiento del TDA y se suele denominar **interfaz pública**.
  - La **implementación**: la parte **privada** del TDA está oculta al programa cliente que lo usa.
- Tanto la **implementación de las operaciones** como los **elementos internos** del TDA serán privados al acceso externo y ocultos a cualquier otro nivel.
- Un TDA representa una **abstracción**:
  - Destaca la **especificación** (el qué).
  - Oculta los detalles de la **implementación** (el cómo).

Especificación formal del TDA lista (simplemente) enlazada (2)

- En este contexto, podemos definir una **lista (simplemente) enlazada** (*linked list*) como una **secuencia de nodos**, y en cada nodo se guardan **datos** y un solo **puntero** al nodo **posterior** (por eso es **simple**).



Especificación formal del TDA lista (simplemente) enlazada (3)

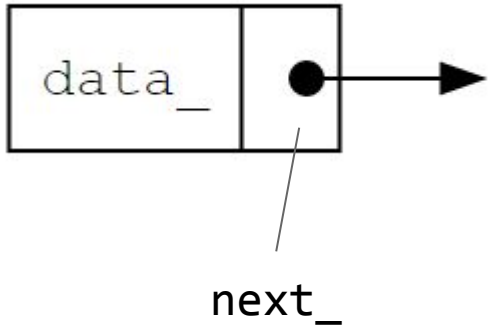
- **Ventajas** (sobre los arrays/vectores):
  - **Estructura dinámica**: permite el redimensionamiento (aumento o reducción) del tamaño de forma muy sencilla.
  - **Inserciones y eliminaciones** en cualquier punto (previamente conocido) en **tiempo constante**.
  - **Orden de los elementos**: puede ser diferente al orden de almacenamiento en la memoria.
- **Desventajas**:
  - **No permite el acceso aleatorio** (o indexado) en tiempo constante: para situarnos en un nodo concreto, tenemos que recorrer los anteriores.
  - **Gasto de memoria** adicional (*overhead*): debido a los punteros.

Especificación formal del TDA lista (simplemente) enlazada (4)

- Al ser un TDA, la lista enlazada tiene las siguientes operaciones (**interfaz pública**):
  - Insertar un nuevo nodo al comienzo de la lista.
  - Extraer un nodo del comienzo de la lista.
  - Insertar un nuevo nodo después de otro de la lista.
  - Extraer un nodo posterior a otro.
  - Comprobar si la lista está vacía.
- La **implementación** de estas operaciones se desarrollará usando dos clases:
  - Clase nodo.
  - Clase lista enlazada.

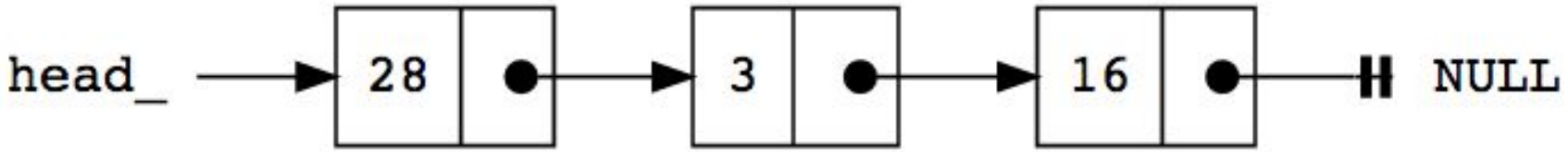
Implementación del TDA lista mediante objetos dinámicos (1)

```
template <class T>
class sll_node_t
{
private:
  T                  data_;
  sll_node_t<T>* next_;

public:
  ...
};
```

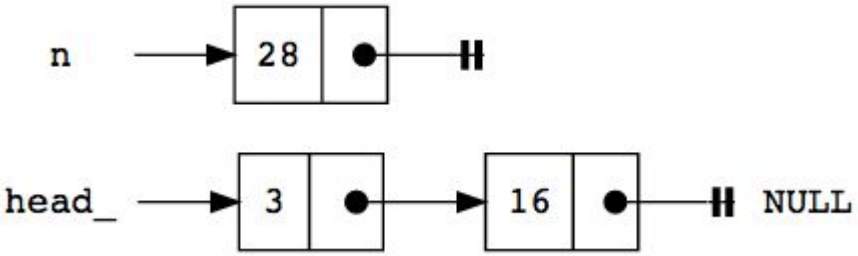


Implementación del TDA lista mediante objetos dinámicos (2)

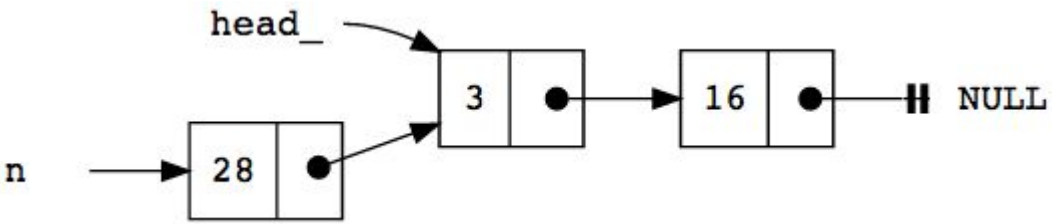
```
template <class T>
class sll_t
{
private:
  sll_node_t<T>* head_;

public:
  ...
};
```



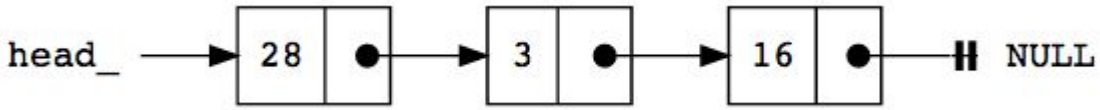
Operaciones: Inserción por delante (1)



(1)



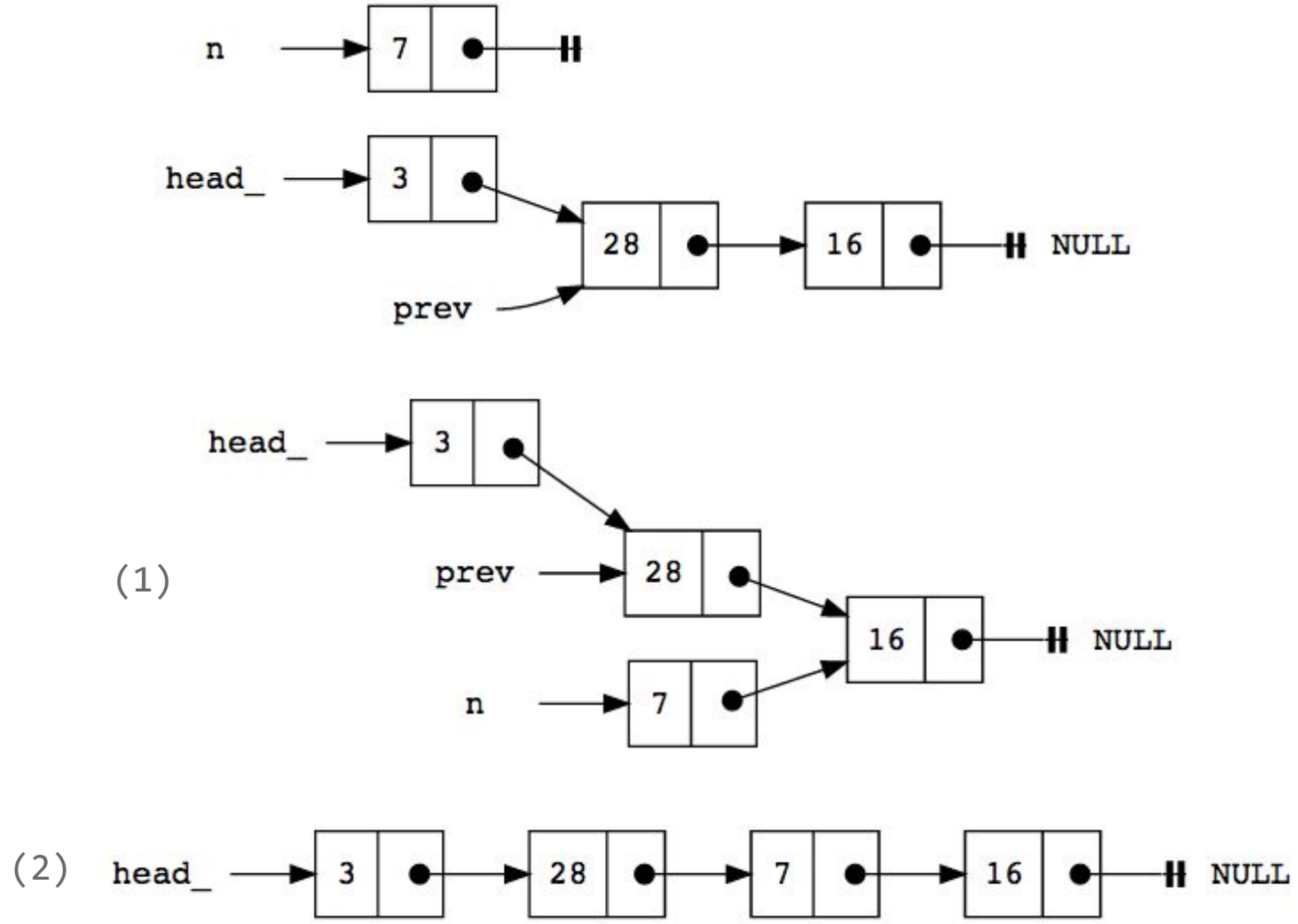
(2)



Operaciones: Inserción por delante (2)

```
template <class T>
void sll_t<T>::insert_head(sll_node_t<T>* n)
{
  n->set_next(head_); // (1)

  head_ = n;          // (2)
}
```

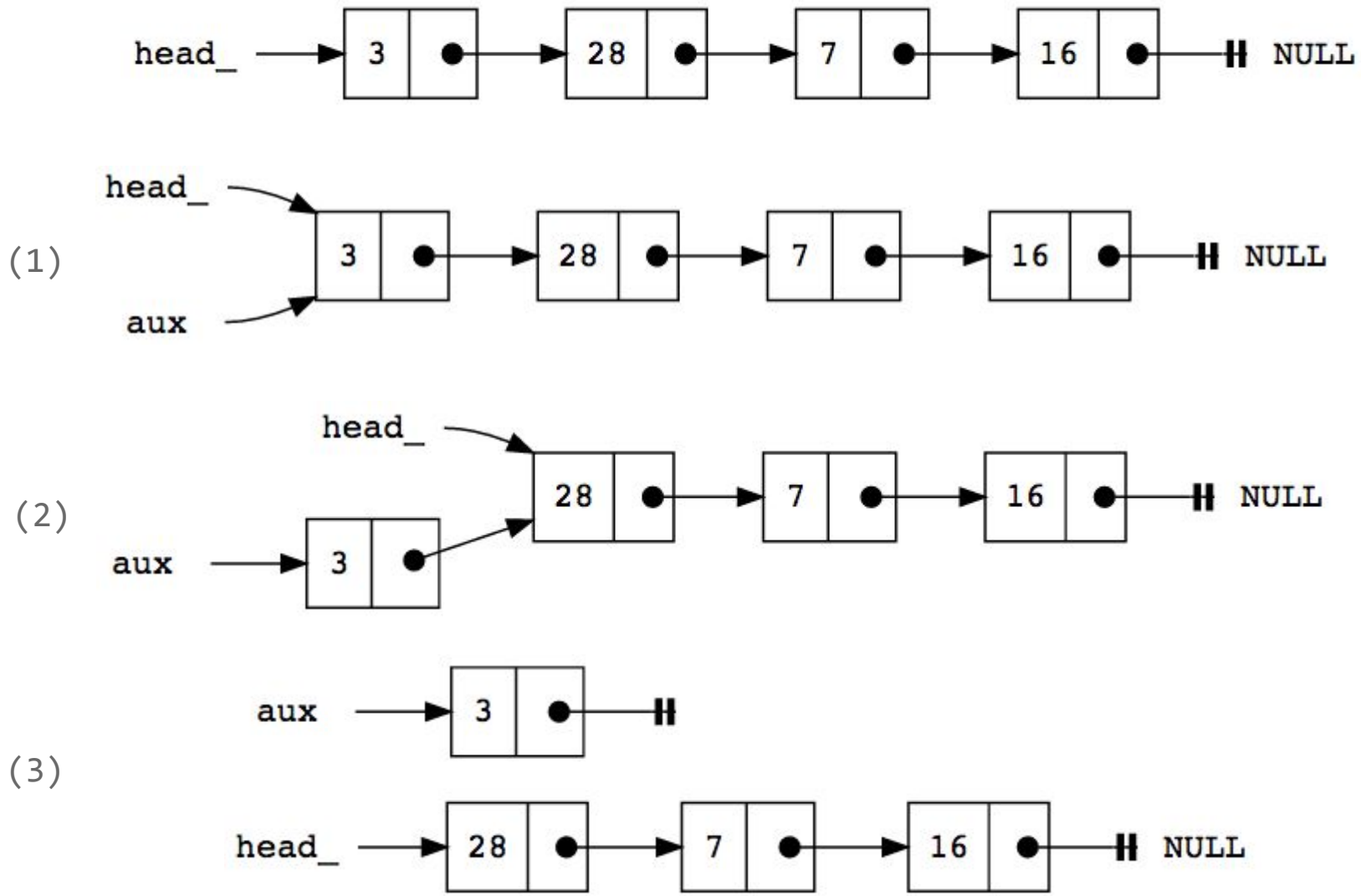
Operaciones: Inserción después de un nodo específico (1)



Operaciones: Inserción después de un nodo específico (2)

```
template <class T>
void sll_t<T>::insert_after(sll_node_t<T>* prev,
                            sll_node_t<T>* n)
{
  n->set_next(prev->get_next()); // (1)

  prev->set_next(n);             // (2)
}
```

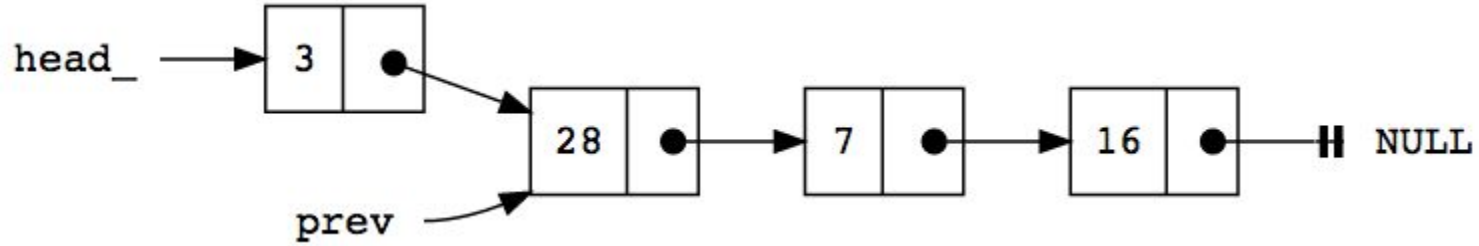
Operaciones: Extracción por delante (1)



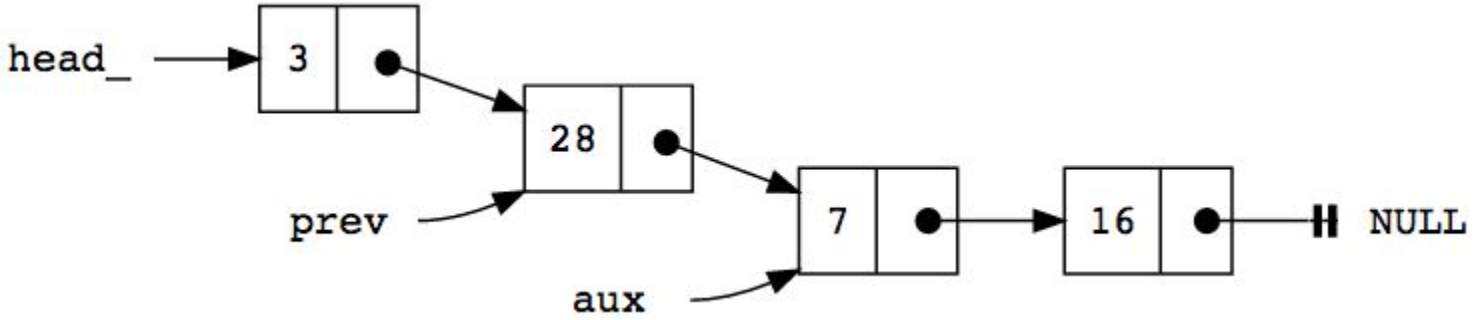
Operaciones: Extracción por delante (2)

```
template <class T>
sll_node_t<T>* sll_t<T>::extract_head(void)
{ assert(head_ != NULL); // !empty()

  sll_node_t<T>* aux = head_;  // (1)

  head_ = head_->get_next();  // (2)

  aux->set_next(NULL);

  return aux;  // (3)
}
```

Operaciones: Extracción de un nodo posterior a otro (1)

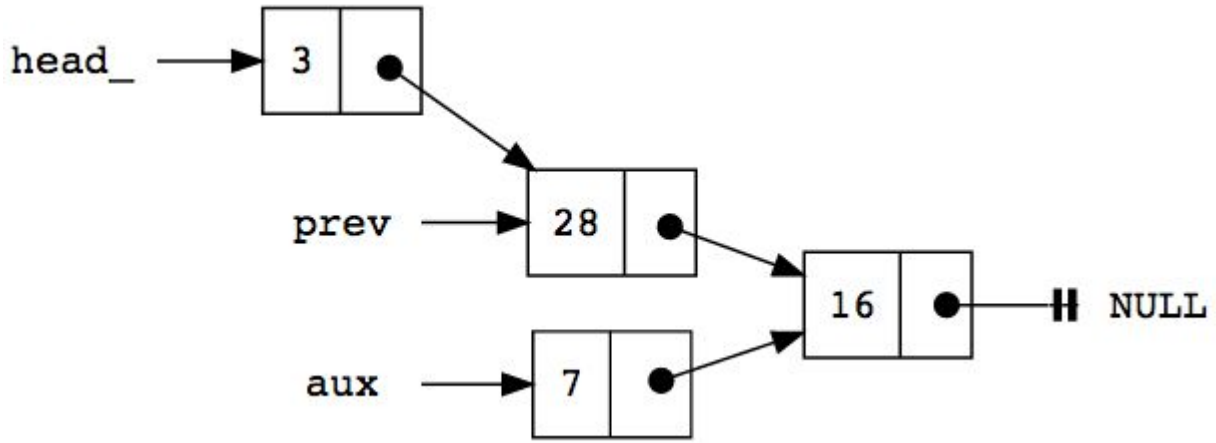


(1)

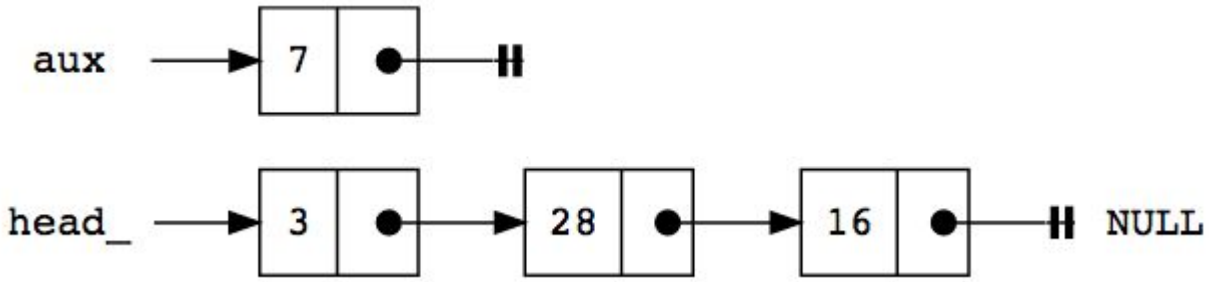


Operaciones: Extracción de un nodo posterior a otro (2)

(2)



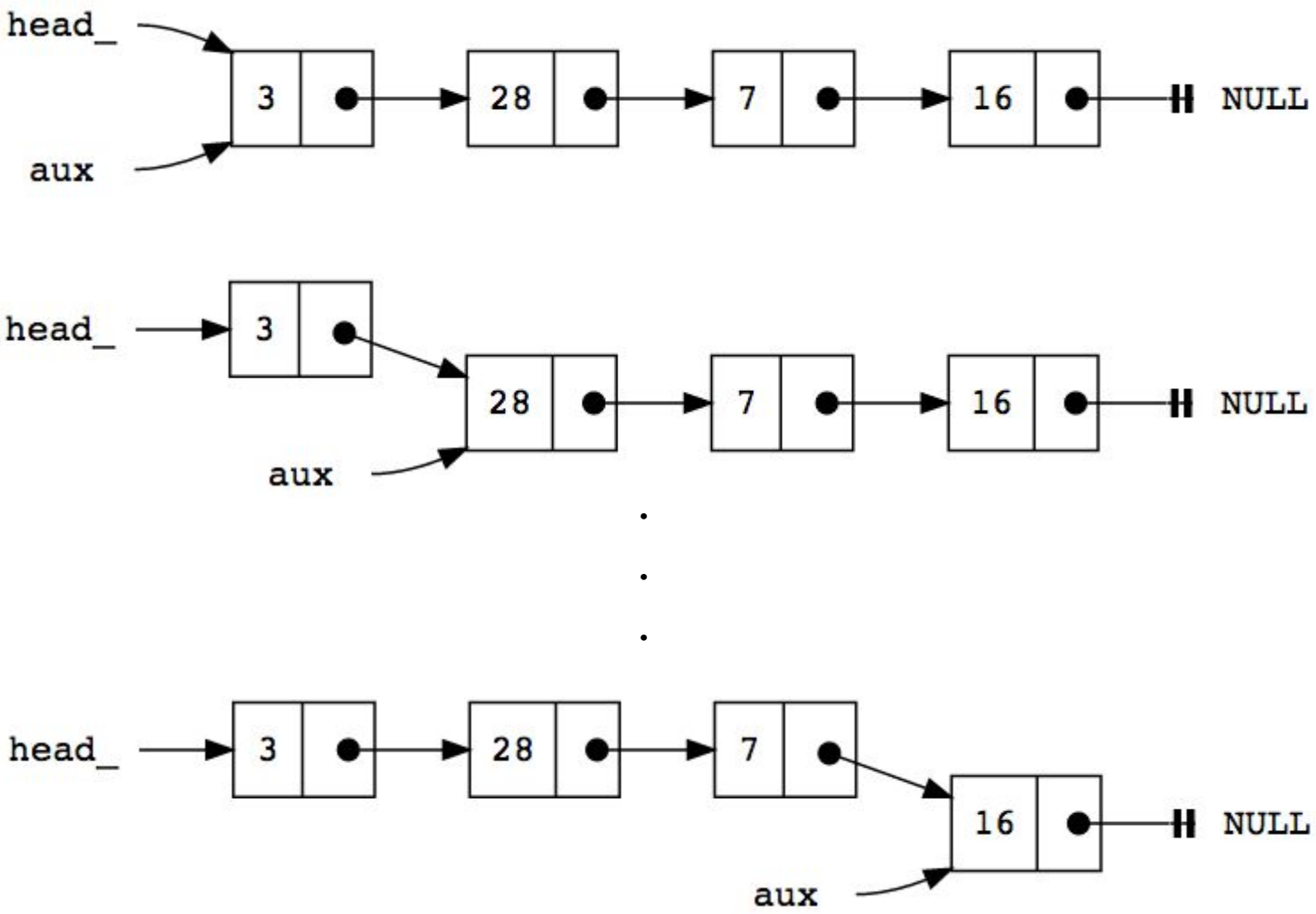
(3)



Operaciones: Extracción de un nodo posterior a otro (3)

```
template <class T>
sll_node_t<T>* sll_t<T>::extract_after(sll_node_t<T>* prev)
{ assert(prev != NULL);

  sll_node_t<T>* aux = prev->get_next();  // (1)

  assert(aux != NULL);

  prev->set_next(aux->get_next());  // (2)

  aux->set_next(NULL);

  return aux;  // (3)
}
```

Operaciones: Recorrido (1)

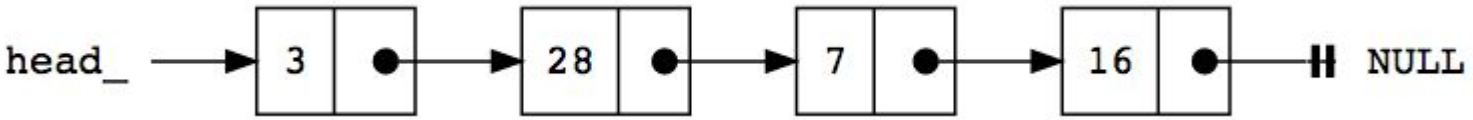


Operaciones: Recorrido (2)

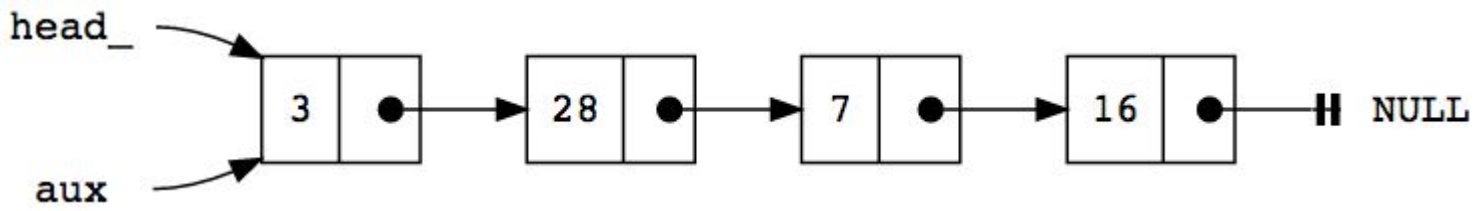
```
sll_node_t<T>* aux = head_;

while (aux != NULL)
{
  // procesar nodo aux: mostrar en pantalla,
  // eliminar, escribir en fichero, etc.

  aux = aux->get_next();
}
```

Operaciones: Búsqueda (1)

- Vamos a buscar el elemento con el dato **7** en la siguiente lista:

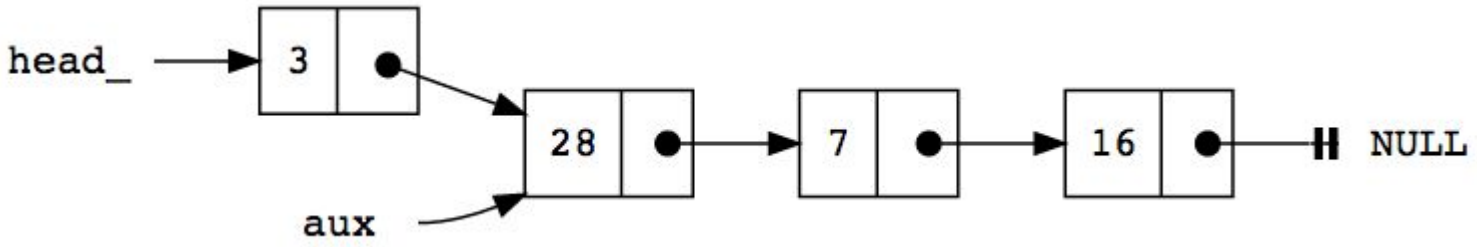


- Para ello, necesitamos un puntero auxiliar aux y una variable `encontrado = false`, que indica si el `data_` actual es igual al valor buscado.

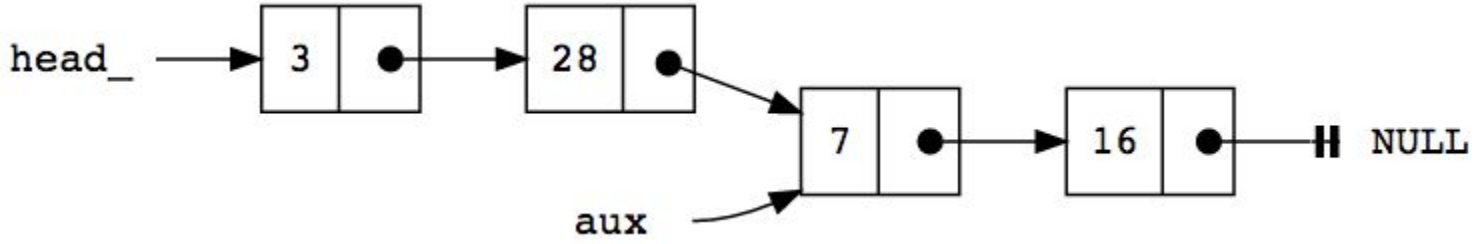


Operaciones: Búsqueda (2)

- Adelantamos el puntero aux y comprobamos si el `data_` es igual a **7**. Como no lo es, `encontrado` sigue siendo **`false`**.



- Seguimos con el siguiente nodo comprobando el campo `data_`. Como es igual, `encontrado` toma el valor **`true`**.



Operaciones: Búsqueda (3)

```
template <class T>
sll_node_t<T>* sll_t<T>::search(const T& d)
{ bool encontrado = false;
  sll_node_t<T>* aux = head_;

  while (aux != NULL && !encontrado) {
    if (aux->get_data() == d) encontrado = true;
    else                      aux = aux->get_next();
  }

  return (encontrado ? aux : NULL);
}
```

Referencias

- ★ Olsson, M. (2018), “C++ 17 Quick Syntax Reference”, Apress. Disponible en PDF en la BBTK-ULL: https://doi.org/10.1007/978-1-4842-3600-0
- ★ Stroustrup, B. (2002), “El Lenguaje de Programación C++”, Addison Wesley.
- ★ C++ Syntax Highlighting (código en colores): tohtml.com/cpp
- ★ Gráficos de las listas: www.webgraphviz.com