



Listas Enlazadas

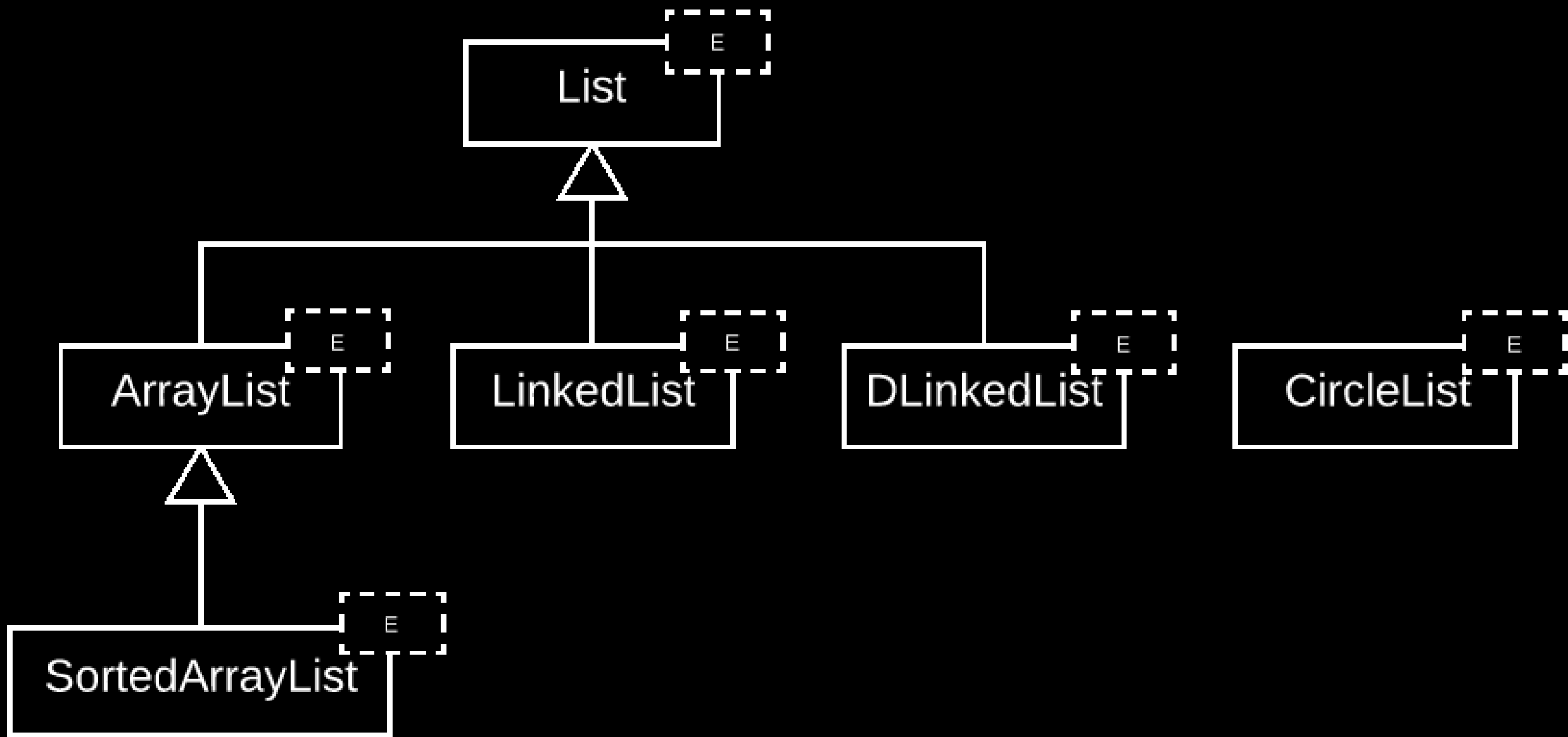
Mauricio Avilés

Contenido

- Listas Enlazadas
 - Clase Node
 - Consideraciones de Diseño
 - Implementación de LinkedList
 - Ejemplo de utilización
 - Ejercicios con LinkedList
- Listas Doblemente Enlazadas
 - Clase DNode
 - Consideraciones de Diseño
 - Implementación de DLinkedList
 - Ejercicios con DLinkedList
- Listas Circulares
 - Operaciones
 - Implementación de CircleList
 - Ejercicios con CircleList

Lecturas

- Sección 4.1
 - Shaffer, C. A. "Data Structures & Algorithm Analysis in C++" (3rd ed., Dover). Mineola, NY. 2011.
- Secciones 3.2, 3.3, 3.4
 - Goodrich, M. T., Tamassia, R., & Mount, D. M. "Data structures and algorithms in C++" (2nd ed., Wiley). Hoboken, NJ: Wiley. 2011.
- Capítulo 10
 - Joyanes, Aguilar, & Martínez. Estructura de datos en C ++. Madrid: McGraw-Hill Interamericana. 2007.



Listas enlazadas

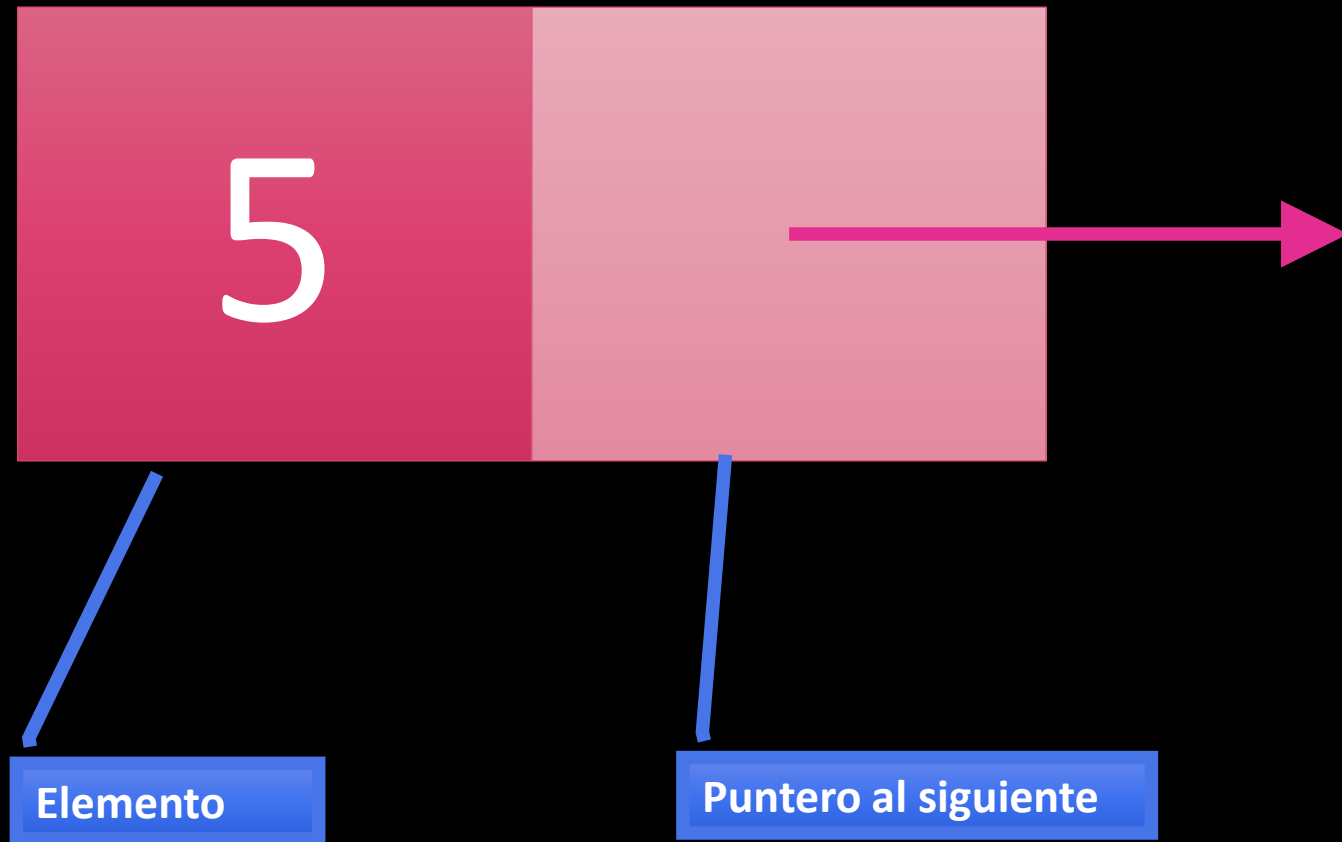
- Implementación con arreglos → **desventajas**
 - **Tamaño** del arreglo no es variable, hay un máximo
 - **Mover elementos** al hacer inserciones
 - Con muchos elementos se genera un **overhead** considerable
- Las listas enlazadas resuelven estos problemas
 - Cada elemento es **independiente** en memoria
 - Cada elemento dice dónde se encuentra el **siguiente** elemento
 - **No** se usan índices

Listas enlazadas

- Hace uso de punteros y memoria dinámica
- Cada elemento se almacena en un nodo
- Se encadenan los nodos
- Un nodo es un objeto diferente, requiere su propia clase



Nodo



```
template <typename E>
class Node {
public:
    E element;
    Node<E>* next;
    Node(E element, Node<E>* next = NULL) {
        this->element = element;
        this->next = next;
    }
    Node(Node<E>* next = NULL) {
        this->next = next;
    }
};
```

Para simplificar el código y dado que la clase Node sólo es utilizada por la lista, todos los miembros se declaran como públicos.


```
template <typename E>
class Node {
public:
    E element;
    Node<E>* next;
    Node(E element, Node<E>* next = NULL) {
        this->element = element;
        this->next = next;
    }
    Node(Node<E>* next = NULL) {
        this->next = next;
    }
};
```

Puntero al siguiente nodo de la lista.

```
template <typename E>
class Node {
public:
    E element;
    Node<E>* next;
    Node(E element, Node<E>* next = NULL) {
        this->element = element;
        this->next = next;
    }
    Node(Node<E>* next = NULL) {
        this->next = next;
    }
};
```

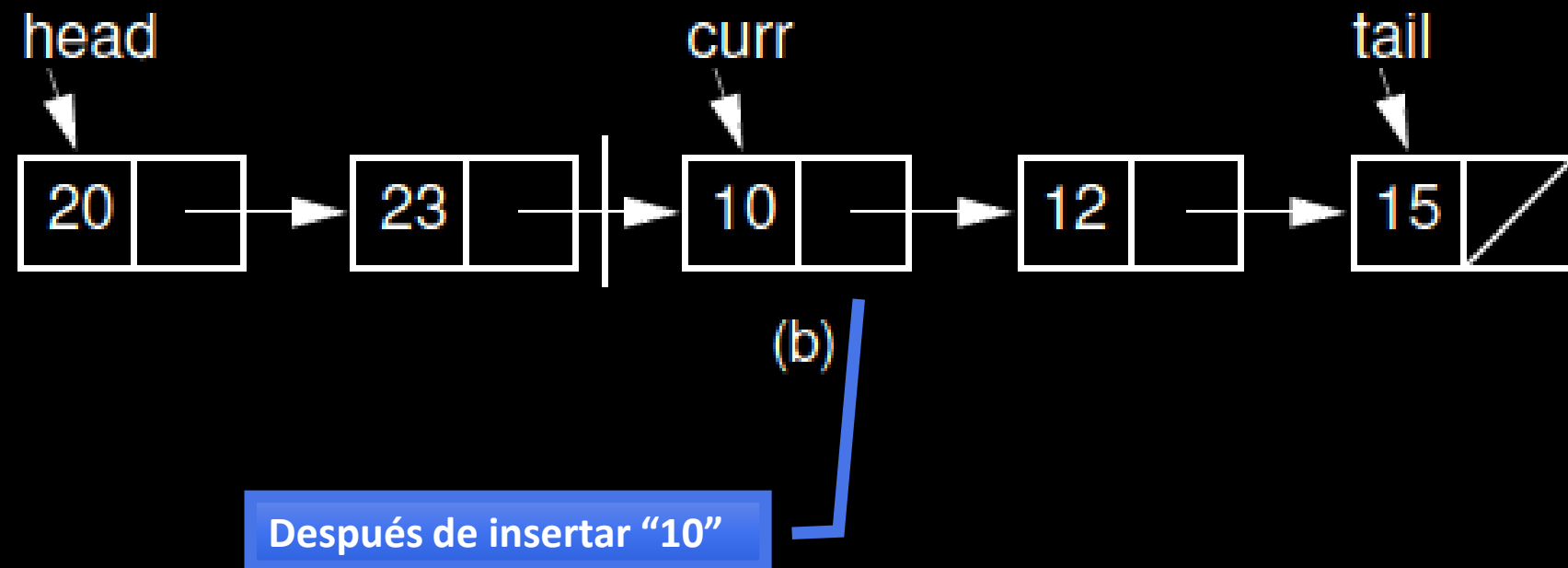
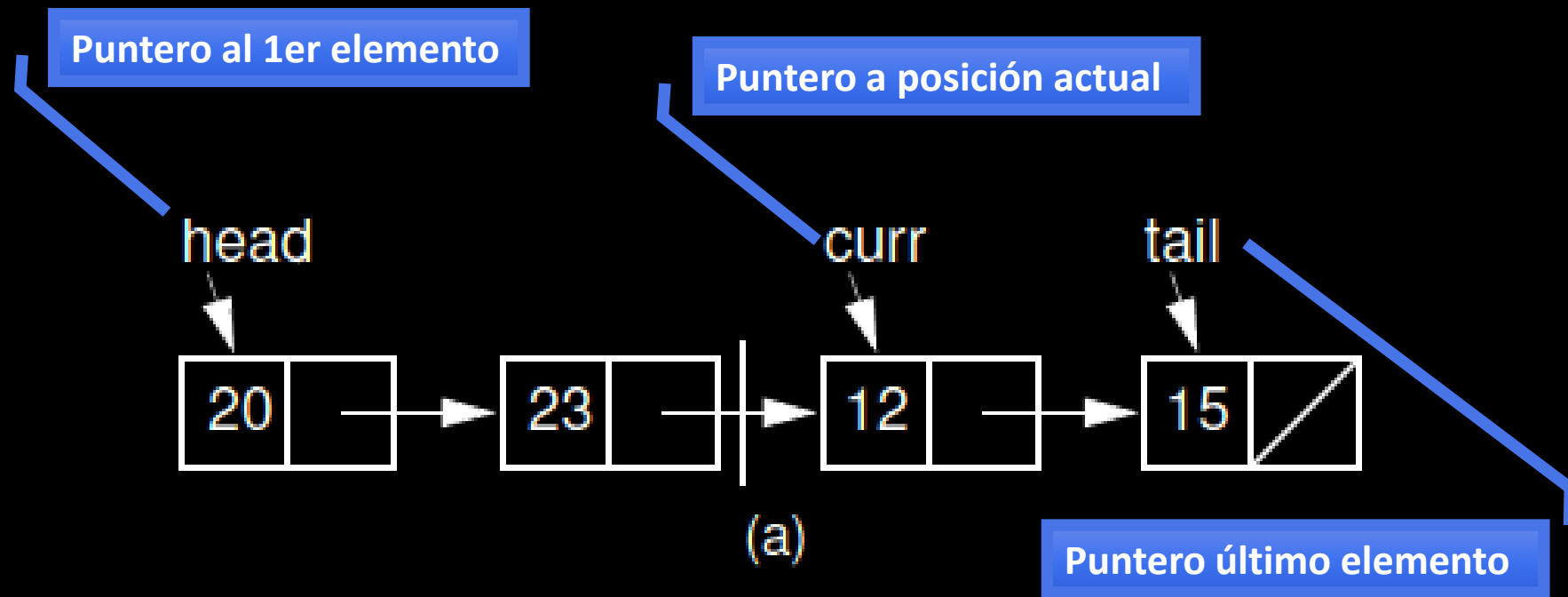
Constructor de la clase. Recibe el elemento a almacenar en el nodo y la dirección donde apuntará next.
Si no se especifica el segundo parámetro, se usará como valor por defecto NULL.

```
template <typename E>
class Node {
public:
    E element;
    Node<E>* next;
    Node(E element, Node<E>* next = NULL) {
        this->element = element;
        this->next = next;
    }
    Node(Node<E>* next = NULL) {
        this->next = next;
    }
};
```

Cuando un parámetro y un atributo de la clase tienen el mismo nombre, es necesario diferenciarlos. Para esto se usa `this`, que es una palabra reservada que retorna un puntero al mismo objeto.

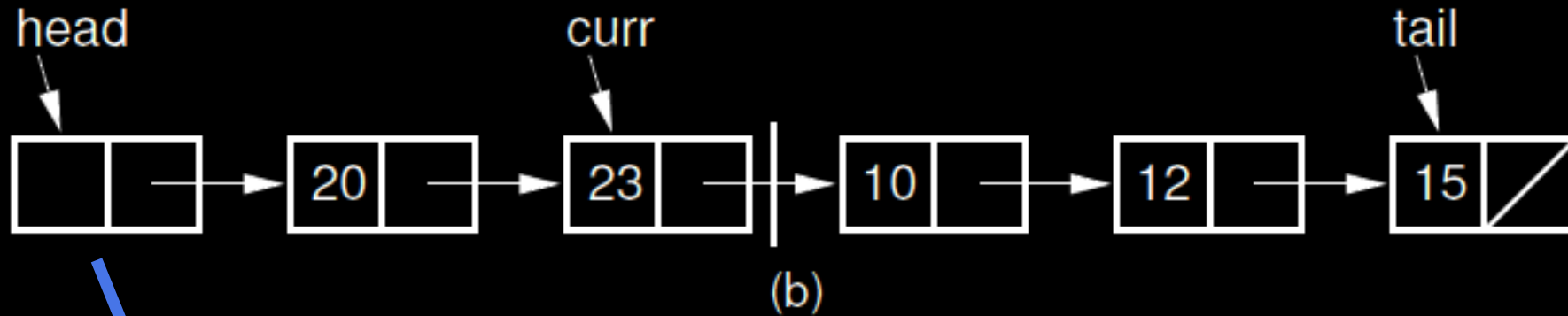
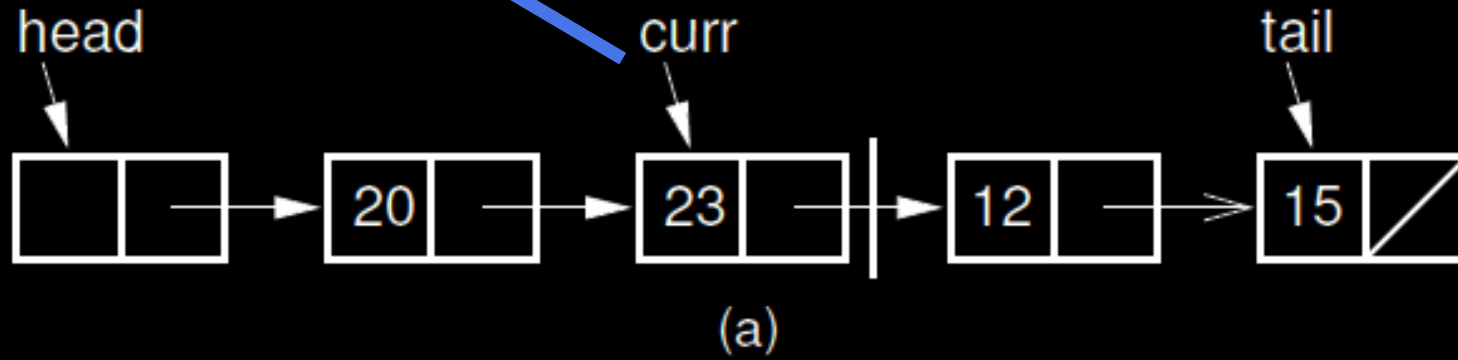
```
template <typename E>
class Node {
public:
    E element;
    Node<E>* next;
    Node(E element, Node<E>* next = NULL) {
        this->element = element;
        this->next = next;
    }
    Node(Node<E>* next = NULL) {
        this->next = next;
    }
};
```

Constructor que inicializa solamente el puntero next, sin inicializar el elemento. Este constructor se utilizará para crear nodos especiales que no guardan ningún elemento.



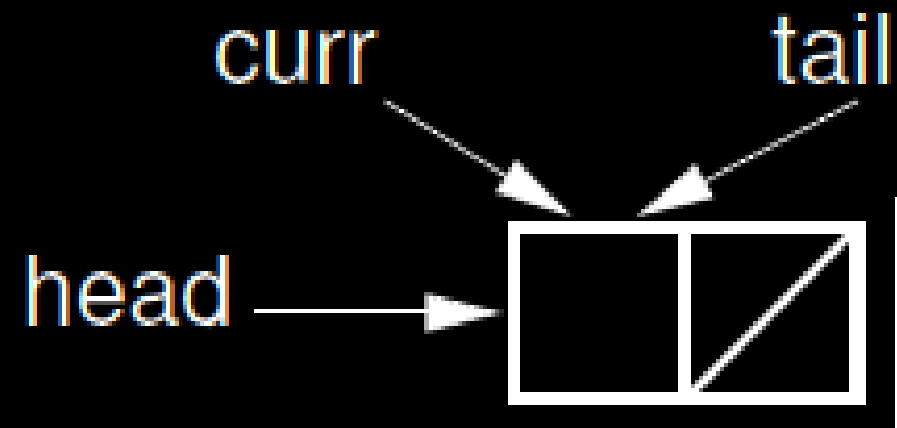
- Para no estar calculando el **largo de la lista** es necesario guardar el valor (size)
- El puntero a la posición actual (size) apunta al elemento actual, pero hay ventajas de apuntar al elemento **anterior** al actual
- Para insertar en una posición siempre es necesario modificar el elemento **anterior** al actual

Si el puntero apunta al elemento anterior al actual, insertar es fácil



Se agrega un nodo especial para evitar casos especiales a la hora de insertar o eliminar. Cuando la lista está vacía, head, curr y tail apuntan a ese nodo especial.

Lista vacía




```
#include "Node.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class LinkedList : public List<E> {
private:
    Node<E>* head;
    Node<E>* tail;
    Node<E>* current;
    int size;
```

También hereda de List, por lo que debe implementar sus métodos abstractos.

```
#include "Node.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class LinkedList : public List<E> {
private:
    Node<E>* head;
    Node<E>* tail;
    Node<E>* current;
    int size;
```

Punteros que señalan al inicio, final y posición actual. El tipo E del nodo coincide con el tipo genérico E de la clase. Esto hace que si se declara la lista con determinado tipo, sus nodos almacenarán elementos del mismo tipo.

```
#include "Node.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class LinkedList : public List<E> {
private:
    Node<E>* head;
    Node<E>* tail;
    Node<E>* current;
    int size;
```

Cantidad de elementos en la lista.

El constructor de la clase no requiere ninguna entrada para inicializar la lista.

```
public:
    LinkedList() {
        head = tail = current = new Node<E>();
        size = 0;
    }
    ~LinkedList() {
        clear();
        delete head;
    }
```

Las asignaciones encadenadas se evalúan de derecha a izquierda.
Se crea el nodo especial en memoria dinámica y los tres punteros head, tail y current apuntan hacia él.

```
public:
    LinkedList() {
        head = tail = current = new Node<E>();
        size = 0;
    }
    ~LinkedList() {
        clear();
        delete head;
    }
```

```
public:
    LinkedList() {
        head = tail = current = new Node<E>();
        size = 0;
    }
    ~LinkedList() {
        clear();
        delete head;
    }
```

El destructor debe eliminar todos los nodos de memoria dinámica. El método clear se encarga de borrar todos los elementos de la lista, será implementado más adelante.
Es necesario eliminar el nodo especial que queda cuando la lista está vacía. Esto se hace por medio del puntero head.

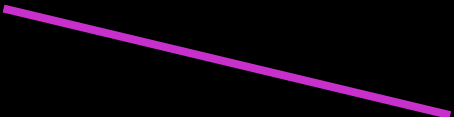
```
void insert(E element) {  
    current->next = new Node<E>(element, current->next);  
    if (current == tail) {  
        tail = tail->next;  
    }  
    size++;  
}  
void append(E element) {  
    tail->next = new Node<E>(element);  
    tail = tail->next;  
    size++;  
}
```

El nuevo nodo se crea con el elemento enviado y su puntero next apuntará al nodo que se encontraba después de current.

El current->next se actualiza para que apunte al nuevo nodo.

```
void insert(E element) {
    current->next = new Node<E>(element, current->next);
    if (current == tail) {
        tail = tail->next;
    }
    size++;
}

void append(E element) {
    tail->next = new Node<E>(element);
    tail = tail->next;
    size++;
}
```



Caso especial, si el nodo nuevo se inserta al final de la lista, es necesario actualizar el puntero tail. Se actualiza el tamaño de la lista.


```
void insert(E element) {
    current->next = new Node<E>(element, current->next);
    if (current == tail) {
        tail = tail->next;
    }
    size++;
}
void append(E element) {
    tail->next = new Node<E>(element);
    tail = tail->next;
    size++;
}
```

Se crea el nuevo nodo con el elemento enviado. Se omite el valor de su puntero next, por lo que queda inicializado en NULL ya que será el nuevo último de la lista.
El puntero next del último se actualiza para apunte al nuevo nodo.

```
void insert(E element) {
    current->next = new Node<E>(element, current->next);
    if (current == tail) {
        tail = tail->next;
    }
    size++;
}
void append(E element) {
    tail->next = new Node<E>(element);
    tail = tail->next;
    size++;
}
```

Se actualiza el puntero tail para que apunte al nuevo nodo y se actualiza el tamaño.

El método falla si no hay un elemento en la posición actual. Como current apunta al nodo anterior al actual, si current->next es NULL, entonces no hay nodo en esa posición y se lanza una excepción.

```
E remove() throw(runtime_error) {  
    if (current->next == NULL) {  
        throw runtime_error("No element to remove.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    current->next = current->next->next;  
    if (current->next == NULL) {  
        tail = current;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

```
E remove() throw(runtime_error) {  
    if (current->next == NULL) {  
        throw runtime_error("No element to remove.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    current->next = current->next->next;  
    if (current->next == NULL) {  
        tail = current;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

Se almacena en una variable el valor del elemento que va a ser eliminado para retornarlo al final.

```
E remove() throw(runtime_error) {  
    if (current->next == NULL) {  
        throw runtime_error("No element to remove.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    current->next = current->next->next;  
    if (current->next == NULL) {  
        tail = current;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

Se utiliza un puntero temporal para no perder la dirección del nodo que se va a eliminar.

```
E remove() throw(runtime_error) {  
    if (current->next == NULL) {  
        throw runtime_error("No element to remove.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    current->next = current->next->next;  
    if (current->next == NULL) {  
        tail = current;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

Se actualiza el puntero `current->next` para que apunte al elemento que se encuentra después del actual. Si el que se está eliminando era el último, este valor es `NULL`, por lo que también funciona la asignación.

```
E remove() throw(runtime_error) {  
    if (current->next == NULL) {  
        throw runtime_error("No element to remove.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    current->next = current->next->next;  
    if (current->next == NULL) {  
        tail = current;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

Caso especial, si el nodo que se elimina es el último, debe actualizarse el valor del puntero tail.

```
E remove() throw(runtime_error) {  
    if (current->next == NULL) {  
        throw runtime_error("No element to remove.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    current->next = current->next->next;  
    if (current->next == NULL) {  
        tail = current;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

Una vez enlazados los punteros de la lista, se elimina el nodo de memoria dinámica.


```
E remove() throw(runtime_error) {  
    if (current->next == NULL) {  
        throw runtime_error("No element to remove.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    current->next = current->next->next;  
    if (current->next == NULL) {  
        tail = current;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

Se actualiza el tamaño y se retorna el valor del elemento eliminado.

```
void clear() {
    current = head;
    while (head != NULL) {
        head = head->next;
        delete current;
        current = head;
    }
    head = tail = current = new Node<E>();
    size = 0;
}

E getElement() throw(runtime_error) {
    if (current->next == NULL) {
        throw runtime_error("No element to get.");
    }
    return current->next->element;
}
```

Se utiliza current para ir recorriendo los nodos y eliminarlos.

```
void clear() {
    current = head;
    while (head != NULL) {
        head = head->next;
        delete current;
        current = head;
    }
    head = tail = current = new Node<E>();
    size = 0;
}

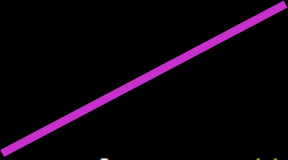
E getElement() throw(runtime_error) {
    if (current->next == NULL) {
        throw runtime_error("No element to get.");
    }
    return current->next->element;
}
```

Se va eliminando el primer elemento de la lista. Al inicio del ciclo current siempre apunta al mismo nodo que head.

Mientras head apunte a un nodo, se le asigna la dirección del siguiente, se elimina al que apunta current y se actualiza current para que apunte nuevamente al nodo que apunta head.

```
void clear() {
    current = head;
    while (head != NULL) {
        head = head->next;
        delete current;
        current = head;
    }
    head = tail = current = new Node<E>();
    size = 0;
}

E getElement() throw(runtime_error) {
    if (current->next == NULL) {
        throw runtime_error("No element to get.");
    }
    return current->next->element;
}
```



Al finalizar se crea nuevamente el nodo auxiliar. Los punteros head, tail y current apuntan a ese nodo. Se actualiza el tamaño de la lista.


```
void clear() {
    current = head;
    while (head != NULL) {
        head = head->next;
        delete current;
        current = head;
    }
    head = tail = current = new Node<E>();
    size = 0;
}

E getElement() throw(runtime_error) {
    if (current->next == NULL) {
        throw runtime_error("No element to get.");
    }
    return current->next->element;
}
```

Obtener el elemento actual de la lista puede fallar si la posición actual es el final de la lista o si la lista está vacía.

```
void clear() {
    current = head;
    while (head != NULL) {
        head = head->next;
        delete current;
        current = head;
    }
    head = tail = current = new Node<E>();
    size = 0;
}

E getElement() throw(runtime_error) {
    if (current->next == NULL) {
        throw runtime_error("No element to get.");
    }
    return current->next->element;
}
```



Se retorna el elemento en el nodo actual.

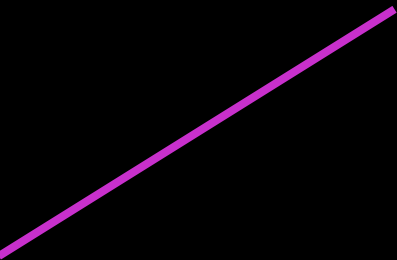
```
void goToStart() {
    current = head;
}
void goToEnd() {
    current = tail;
}
void goToPos(int newPos) throw(runtime_error) {
    if ((newPos < 0) || (newPos > size)) {
        throw runtime_error("Index out of bounds");
    }
    current = head;
    for (int i = 0; i < newPos; i++) {
        current = current->next;
    }
}
```

Actualizar current para que apunte al nodo especial al inicio de la lista.

```
void goToStart() {
    current = head;
}
void goToEnd() {
    current = tail;
}
void goToPos(int newPos) throw(runtime_error) {
    if ((newPos < 0) || (newPos > size)) {
        throw runtime_error("Index out of bounds");
    }
    current = head;
    for (int i = 0; i < newPos; i++) {
        current = current->next;
    }
}
```

Actualizar current para que apunte al último nodo de la lista.


```
void goToStart() {
    current = head;
}
void goToEnd() {
    current = tail;
}
void goToPos(int newPos) throw(runtime_error) {
    if ((newPos < 0) || (newPos > size)) {
        throw runtime_error("Index out of bounds");
    }
    current = head;
    for (int i = 0; i < newPos; i++) {
        current = current->next;
    }
}
```



El método goToPos puede fallar si la posición indicada no es válida dentro de la lista. Debe ser un valor desde 0 y hasta el tamaño de la lista.

```
void goToStart() {
    current = head;
}
void goToEnd() {
    current = tail;
}
void goToPos(int newPos) throw(runtime_error) {
    if ((newPos < 0) || (newPos > size)) {
        throw runtime_error("Index out of bounds");
    }
    current = head;
    for (int i = 0; i < newPos; i++) {
        current = current->next;
    }
}
```

Se ubica la posición actual al inicio de la lista y se avanza hasta llegar a la posición indicada.

```
void previous() {
    if (current != head) {
        Node<E> *temp = head;
        while (temp->next != current) {
            temp = temp->next;
        }
        current = temp;
    }
}

bool atStart() {
    return current == head;
}

void next() {
    if (current != tail) {
        current = current->next;
    }
}

bool atEnd() {
    return current == tail;
}
```

Para mover current a la posición anterior es necesario ir al inicio de la lista y recorrerla hasta encontrar el nodo anterior.

```
void previous() {  
    if (current != head) {  
        Node<E> *temp = head;  
        while (temp->next != current) {  
            temp = temp->next;  
        }  
        current = temp;  
    }  
}  
  
bool atStart() {  
    return current == head;  
}  
  
void next() {  
    if (current != tail) {  
        current = current->next;  
    }  
}  
  
bool atEnd() {  
    return current == tail;  
}
```

Se utiliza un puntero temporal para recorrer la lista. Mientras el siguiente sea diferente a current se avanza al siguiente nodo. Al salir del ciclo se actualiza current.

```
void previous() {  
    if (current != head) {  
        Node<E> *temp = head;  
        while (temp->next != current) {  
            temp = temp->next;  
        }  
        current = temp;  
    }  
}  
  
bool atStart() {  
    return current == head;  
}  
  
void next() {  
    if (current != tail) {  
        current = current->next;  
    }  
}  
  
bool atEnd() {  
    return current == tail;  
}
```

Para saber si la posición actual es al inicio de la fila, basta con evaluar si current es igual a head.

```
void previous() {
    if (current != head) {
        Node<E> *temp = head;
        while (temp->next != current) {
            temp = temp->next;
        }
        current = temp;
    }
}

bool atStart() {
    return current == head;
}

void next() {
    if (current != tail) {
        current = current->next;
    }
}

bool atEnd() {
    return current == tail;
}
```

El método next avanza current al siguiente elemento, esto sólo si no se encuentra al final de la lista.

```
void previous() {
    if (current != head) {
        Node<E> *temp = head;
        while (temp->next != current) {
            temp = temp->next;
        }
        current = temp;
    }
}

bool atStart() {
    return current == head;
}

void next() {
    if (current != tail) {
        current = current->next;
    }
}

bool atEnd() {
    return current == tail;
}
```

Similar al método atStart, sólo es necesario evaluar si current apunta al mismo nodo que tail.

```
int getPos() {  
    int pos = 0;  
    Node<E>* temp = head;  
    while (temp != current) {  
        pos++;  
        temp = temp->next;  
    }  
    return pos;  
}  
int getSize() {  
    return size;  
}  
};
```

Obtener la posición actual no es tan simple como en la implementación con arreglos, ya que no se cuenta con una variable que indique esto. Por esto es necesario usar un puntero temporal que apunte al primer elemento y avanzar por los nodos mientras se incrementa un contador, hasta llegar al nodo donde apunta current.


```
int getPos() {  
    int pos = 0;  
    Node<E>* temp = head;  
    while (temp != current) {  
        pos++;  
        temp = temp->next;  
    }  
    return pos;  
}  
int getSize() {  
    return size;  
}  
};
```

Se retorna el valor del atributo size.

Ejemplos de utilización

- Dado que la clase LinkedList implementa los mismos comportamientos que la clase ArrayList, los mismos ejemplos deben dar exactamente los mismos resultados.
- Sólo debe sustituirse el uso de ArrayList por LinkedList.

Ejercicios con la clase LinkedList

- Agregar el método **contains**.
 - Recibe como parámetro un elemento.
 - Retorna un valor booleano indicando si la lista contiene dicho elemento.
 - No tiene ninguna restricción.
- Agregar el método **indexOf**.
 - Recibe como parámetro un elemento.
 - Retorna un número entero con la posición de dicho elemento dentro de la lista.
 - Si el elemento no se encuentra dentro de la lista, debe retornar -1.
 - No tiene ninguna restricción.
- Agregar el método **extend**.
 - Recibe como parámetro un objeto de tipo List.
 - Agrega al final de la lista actual todos los elementos de la lista que recibe por parámetro, en el mismo orden.
 - La lista enviada por parámetro no debe ser modificada.
- No tiene ninguna restricción.
- Agregar el método **reverse**.
 - No recibe ningún parámetro.
 - Invierte el orden de los elementos de la lista.
 - No tiene ninguna restricción.
- Agregar método **equals**.
 - Recibe como parámetro otro objeto de tipo List.
 - Retorna verdadero si los elementos en la lista actual son los mismos y están en el mismo orden que los elementos de la list enviada por parámetro. De otra forma retorna falso.
- **Elevar los métodos** implementados a la clase abstracta List.
 - Ya que estos métodos también fueron implementados en la clase ArrayList, ubicar su declaración abstracta en la clase List de la que ambas son subclases.

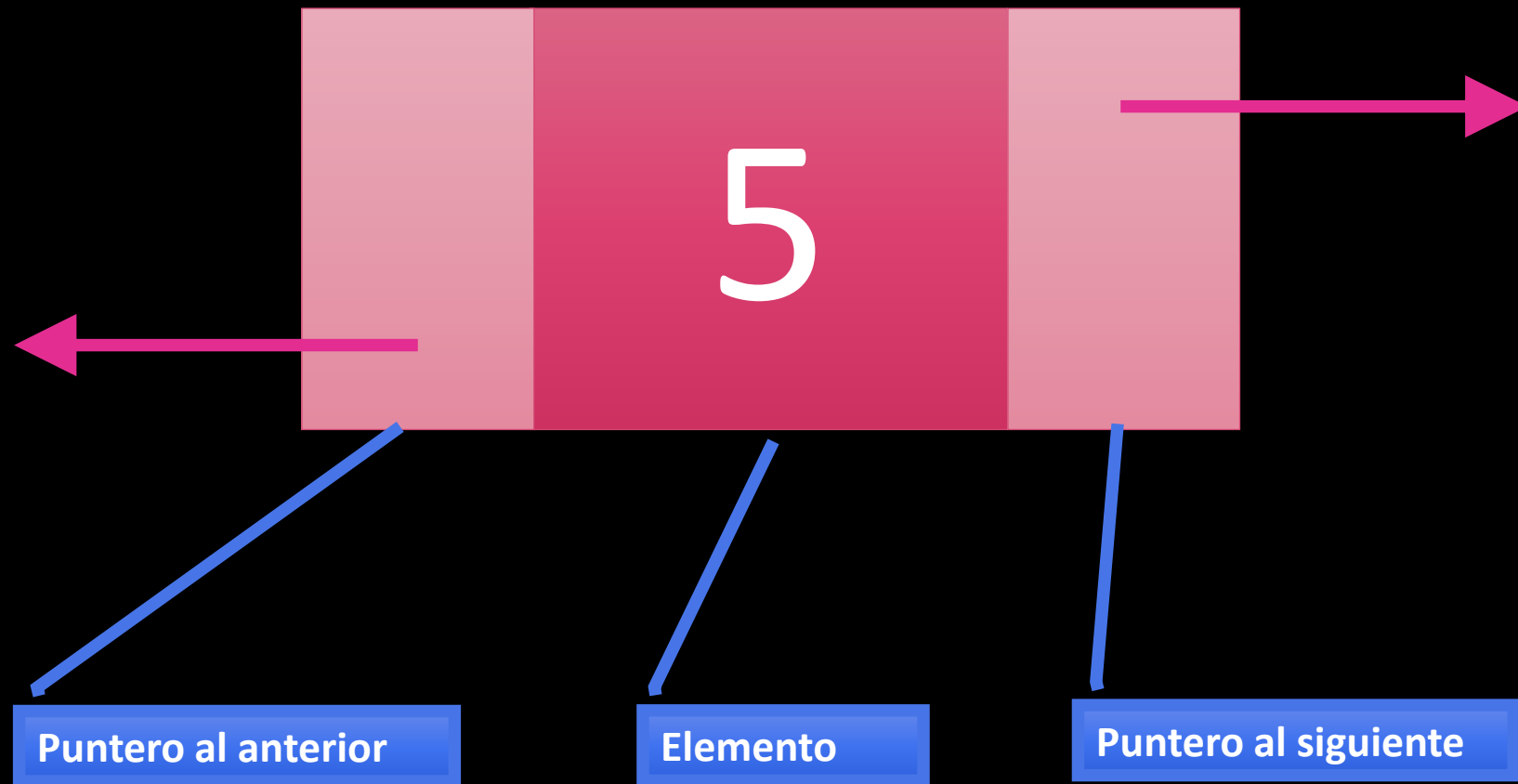
Ejercicios con la clase LinkedList

- Lista de **nodos libres**
 - Modificar la lista enlazada para que maneje una lista de nodos libres.
 - En lugar de borrar los nodos cada vez que se elimina un elemento, agregarlos a una lista de nodos libres.
 - Cuando se crea un nuevo elemento, se puede reutilizar alguno de los nodos en la lista de nodos libres en lugar de solicitar nueva memoria dinámica.
 - Cree métodos privados que ayuden a lograr este comportamiento.
- Crear clase **OrderedLinkedList**
 - Lista que mantiene los elementos ordenados de menor a mayor (si son comparables).
 - Determinar qué métodos de la clase LinkedList son relevantes y cuáles no.
 - Determinar si se requieren métodos nuevos.
 - Utilizar ArrayList como atributo de la clase OrderedLinkedList.
 - Los métodos de esta nueva clase se implementan mediante llamadas a los métodos de LinkedList.
 - Hacer su propia versión de indexOf con búsqueda binaria.

Listas doblemente enlazadas

- Muy **similar** a la lista enlazada pero utiliza un **nodo ligeramente diferente**
- El nodo doble permite el acceso al nodo **anterior** y al **siguiente**
- Cada nodo utiliza **dos punteros**
- Su implementación es más obvia
- Se utilizarán **dos nodos especiales**, para la cabeza y cola de la lista
 - **Misma estrategia** que con las listas enlazadas
 - **Simplifica** la cantidad de casos especiales

Nodo



```
template <typename E>
```

```
class DNode {
```

```
public:
```

```
    E element;
```

```
    DNode<E>* next;
```

```
    DNode<E>* previous;
```

```
    DNode(E element, DNode<E>* next = NULL, DNode<E>* previous = NULL) {
```

```
        this->element = element;
```

```
        this->next = next;
```

```
        this->previous = previous;
```

```
    }
```

```
    DNode(DNode<E>* next = NULL, DNode<E>* previous = NULL) {
```

```
        this->next = next;
```

```
        this->previous = previous;
```

```
    }
```

```
};
```

Clase muy similar al nodo simple de las listas enlazadas.

```
template <typename E>
```

```
class DNode {
```

```
public:
```

```
    E element;
```

```
    DNode<E>* next;
```

```
    DNode<E>* previous;
```

Atributo que apunta al nodo en la posición anterior de la lista.

```
    DNode(E element, DNode<E>* next = NULL, DNode<E>* previous = NULL) {
```

```
        this->element = element;
```

```
        this->next = next;
```

```
        this->previous = previous;
```

```
    }
```

```
    DNode(DNode<E>* next = NULL, DNode<E>* previous = NULL) {
```

```
        this->next = next;
```

```
        this->previous = previous;
```

```
    }
```

```
};
```



```
template <typename E>
class DNode {
public:
```

```
    E element;
    DNode<E>* next;
    DNode<E>* previous;
```

El constructor recibe el valor del elemento, el puntero siguiente y el puntero anterior. Estos dos tienen NULL como valor por defecto.

```
    DNode(E element, DNode<E>* next = NULL, DNode<E>* previous = NULL) {
        this->element = element;
        this->next = next;
        this->previous = previous;
    }
```

```
    DNode(DNode<E>* next = NULL, DNode<E>* previous = NULL) {
        this->next = next;
        this->previous = previous;
    }
```

```
};
```

```
template <typename E>
class DNode {
public:
    E element;
    DNode<E>* next;
    DNode<E>* previous;
```

```
    DNode(E element, DNode<E>* next = NULL, DNode<E>* previous = NULL) {
        this->element = element;
        this->next = next;
        this->previous = previous;
    }
```

Nuevamente se utiliza this para diferenciar los parámetros de los atributos de la clase.

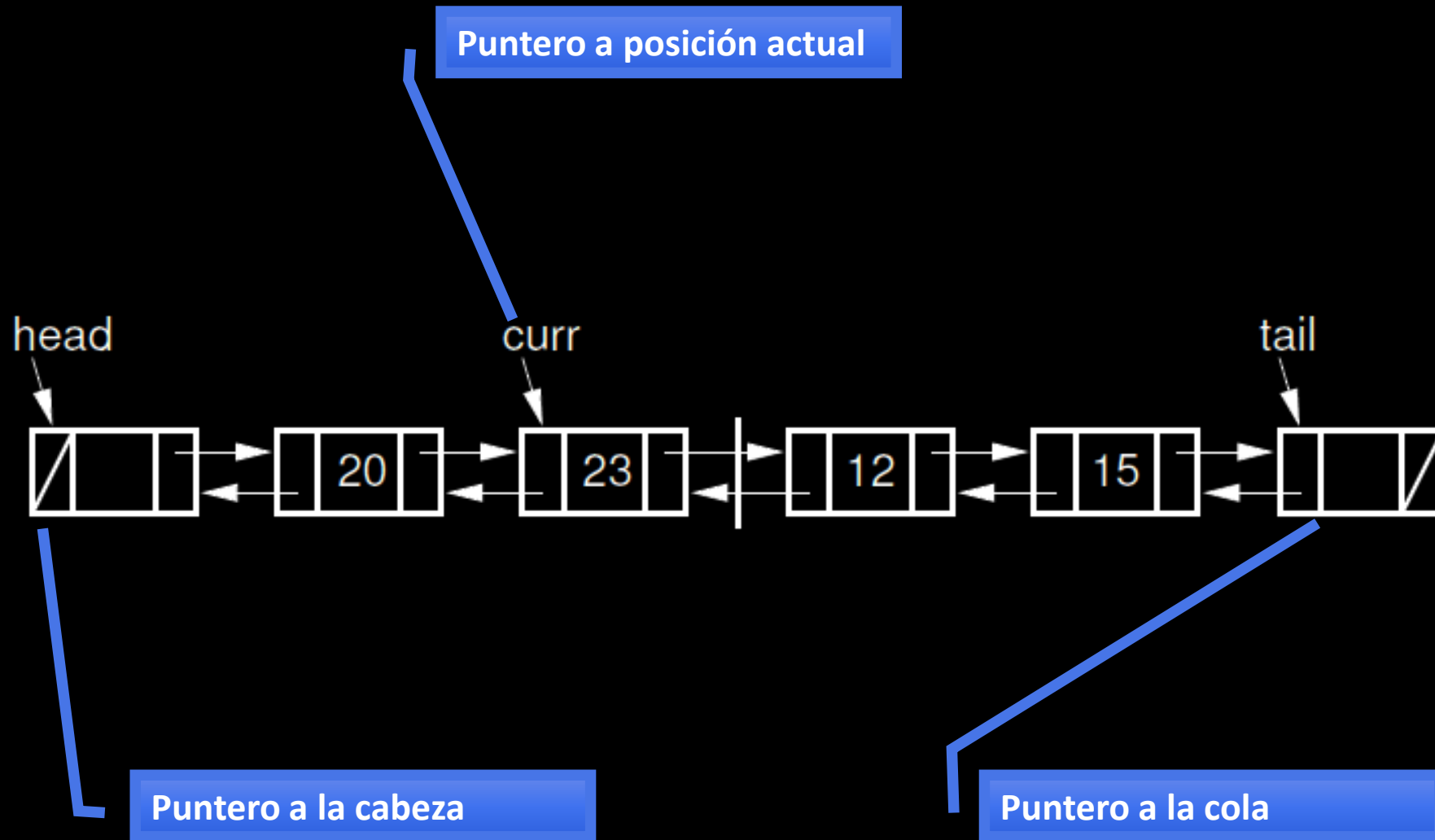
```
    DNode(DNode<E>* next = NULL, DNode<E>* previous = NULL) {
        this->next = next;
        this->previous = previous;
    }
};
```

```
template <typename E>
class DNode {
public:
    E element;
    DNode<E>* next;
    DNode<E>* previous;

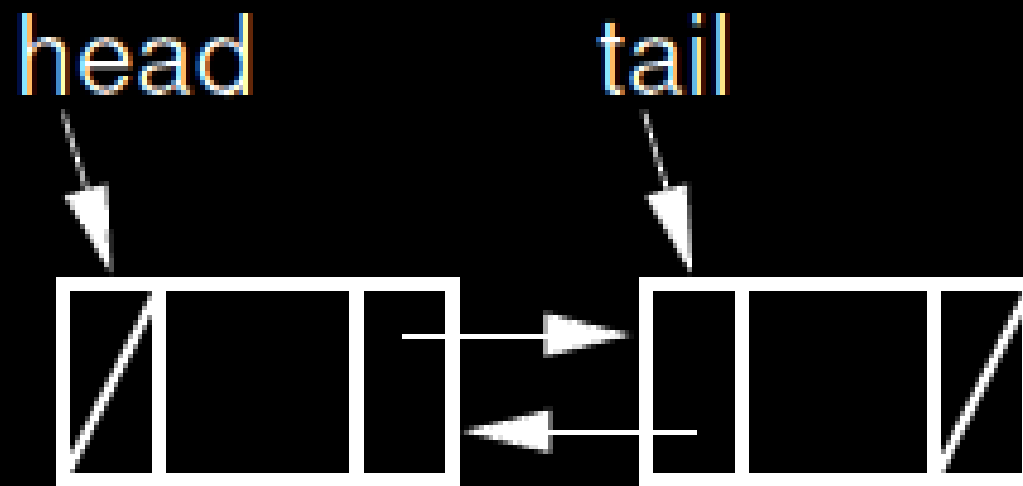
    DNode(E element, DNode<E>* next = NULL, DNode<E>* previous = NULL) {
        this->element = element;
        this->next = next;
        this->previous = previous;
    }

    DNode(DNode<E>* next = NULL, DNode<E>* previous = NULL) {
        this->next = next;
        this->previous = previous;
    }
};
```

Constructor que no inicializa el valor del elemento.
Los punteros tienen NULL como valor por defecto.



Lista vacía



```
#include <stdexcept>
#include "DNode.h"

using std::runtime_error;

template <typename E>
class DLinkedList : public List<E> {
private:
    DNode<E>* head;
    DNode<E>* tail;
    DNode<E>* current;
    int size;

    void init() {
        head = new DNode<E>();
        tail = new DNode<E>();
        head->next = tail;
        tail->previous = head;
        current = head;
        size = 0;
    }
}
```

Los atributos son iguales a los de LinkedList.
Puntero para la cabeza, cola y actual de la lista.
Atributo para conocer la cantidad de elementos.

```
#include <stdexcept>
#include "DNode.h"

using std::runtime_error;

template <typename E>
class DLinkedList : public List<E> {
private:
    DNode<E>* head;
    DNode<E>* tail;
    DNode<E>* current;
    int size;

    void init() {
        head = new DNode<E>();
        tail = new DNode<E>();
        head->next = tail;
        tail->previous = head;
        current = head;
        size = 0;
    }
}
```


Método que se encarga de inicializar la lista con los nodos especiales. Será invocado desde el constructor y desde el método clear. Se separa de esta forma para simplificar el método clear.

```
#include <stdexcept>
#include "DNode.h"

using std::runtime_error;

template <typename E>
class DLinkedList : public List<E> {
private:
    DNode<E>* head;
    DNode<E>* tail;
    DNode<E>* current;
    int size;

    void init() {
        head = new DNode<E>();
        tail = new DNode<E>();
        head->next = tail;
        tail->previous = head;
        current = head;
        size = 0;
    }
}
```



Se asume que los punteros no están apuntando a ningún nodo actualmente.
Se crean los dos nodos especiales y se enlazan los punteros para crear la lista vacía. Se inicializa el tamaño de la lista.

public:

```
DLinkedList() {  
    init();  
}  
~DLinkedList() {  
    clear();  
    delete head;  
    delete tail;  
}
```

El constructor se encarga de inicializar la lista invocando al método init.

```
public:
    DLinkedList() {
        init();
    }
    ~DLinkedList() {
        clear();
        delete head;
        delete tail;
    }
```

El destructor libera la memoria de todos los nodos que contienen elementos invocando a clear. Luego libera la memoria de los nodos especiales.

Se crea el nodo a insertar con el valor del elemento. El puntero next del nuevo nodo apunta al nodo en este momento es el actual. El puntero previous apunta al anterior al actual, es decir, a current.

```
void insert(E element) {
    current->next->previous = new DNode<E>(element, current->next, current);
    current->next = current->next->previous;
    size++;
}

void append(E element) {
    tail->previous->next = new DNode<E>(element, tail, tail->previous);
    tail->previous = tail->previous->next;
    size++;
}
```

El previous del nodo que era el actual, ahora apunta al nuevo nodo, que se convierte en el actual.

```
void insert(E element) {  
    current->next->previous = new DNode<E>(element, current->next, current);  
    current->next = current->next->previous;  
    size++;  
}  
void append(E element) {  
    tail->previous->next = new DNode<E>(element, tail, tail->previous);  
    tail->previous = tail->previous->next;  
    size++;  
}
```

El next del nodo anterior al actual ahora apunta también al nuevo nodo. Se aumenta el tamaño.

```
void insert(E element) {
    current->next->previous = new DNode<E>(element, current->next, current);
    current->next = current->next->previous;
    size++;
}

void append(E element) {
    tail->previous->next = new DNode<E>(element, tail, tail->previous);
    tail->previous = tail->previous->next;
    size++;
}
```

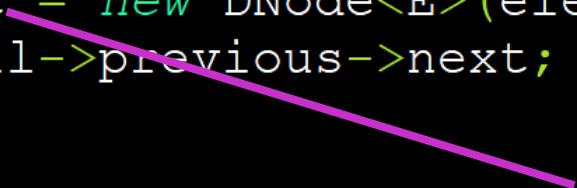
```
void insert(E element) {
    current->next->previous = new DNode<E>(element, current->next, current);
    current->next = current->next->previous;
    size++;
}

void append(E element) {
    tail->previous->next = new DNode<E>(element, tail, tail->previous);
    tail->previous = tail->previous->next;
    size++;
}
```

Similar al método insert pero el trabajo se realiza al final de la lista, con ayuda del nodo especial. El puntero next del nuevo nodo debe apuntar al nodo especial y el previous debe apuntar al antiguo último nodo, apuntado por el previous del nodo especial.

```
void insert(E element) {
    current->next->previous = new DNode<E>(element, current->next, current);
    current->next = current->next->previous;
    size++;
}

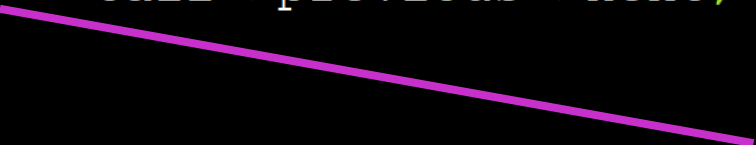
void append(E element) {
    tail->previous->next = new DNode<E>(element, tail, tail->previous);
    tail->previous = tail->previous->next;
    size++;
}
```



El next del antiguo último nodo ahora debe apuntar al nodo nuevo.

```
void insert(E element) {
    current->next->previous = new DNode<E>(element, current->next, current);
    current->next = current->next->previous;
    size++;
}

void append(E element) {
    tail->previous->next = new DNode<E>(element, tail, tail->previous);
    tail->previous = tail->previous->next;
    size++;
}
```



Actualizar el previous del nodo especial para que apunte al nuevo nodo. Aumentar tamaño.


```
E remove() throw(runtime_error) {  
    if (current->next == tail) {  
        throw runtime_error("No element to remove.");  
    }  
    E res = current->next->element;  
    DNode<E>* temp = current->next;  
    current->next = current->next->next;  
    current->next->previous = current;  
    delete temp;  
    size--;  
    return res;  
}  
void clear() {  
    while (head != NULL) {  
        current = head;  
        head = head->next;  
        delete current;  
    }  
    init();  
}
```

El método puede fallar si la posición actual se encuentra al final de la lista y no hay elemento actual.

```
E remove() throw(runtime_error) {
    if (current->next == tail) {
        throw runtime_error("No element to remove.");
    }
    E res = current->next->element;
    DNode<E>* temp = current->next;
    current->next = current->next->next;
    current->next->previous = current;
    delete temp;
    size--;
    return res;
}

void clear() {
    while (head != NULL) {
        current = head;
        head = head->next;
        delete current;
    }
    init();
}
```

Se almacena en una variable temporal el valor del elemento que se elimina.

```
E remove() throw(runtime_error) {
    if (current->next == tail) {
        throw runtime_error("No element to remove.");
    }
    E res = current->next->element;
    DNode<E>* temp = current->next;
    current->next = current->next->next;
    current->next->previous = current;
    delete temp;
    size--;
    return res;
}

void clear() {
    while (head != NULL) {
        current = head;
        head = head->next;
        delete current;
    }
    init();
}
```

Puntero para mantener la dirección del nodo que se elimina.

```
E remove() throw(runtime_error) {
    if (current->next == tail) {
        throw runtime_error("No element to remove.");
    }
    E res = current->next->element;
    DNode<E>* temp = current->next;
    current->next = current->next->next;
    current->next->previous = current;
    delete temp;
    size--;
    return res;
}

void clear() {
    while (head != NULL) {
        current = head;
        head = head->next;
        delete current;
    }
    init();
}
```

El siguiente del nodo anterior al actual apunta al elemento que se encuentra después del actual.

```
E remove() throw(runtime_error) {
    if (current->next == tail) {
        throw runtime_error("No element to remove.");
    }
    E res = current->next->element;
    DNode<E>* temp = current->next;
    current->next = current->next->next;
    current->next->previous = current;
    delete temp;
    size--;
    return res;
}

void clear() {
    while (head != NULL) {
        current = head;
        head = head->next;
        delete current;
    }
    init();
}
```

Se libera la memoria del nodo en cuestión. Se actualiza el tamaño. Se retorna el valor del elemento que fue eliminado.

```
E remove() throw(runtime_error) {
    if (current->next == tail) {
        throw runtime_error("No element to remove.");
    }
    E res = current->next->element;
    DNode<E>* temp = current->next;
    current->next = current->next->next;
    current->next->previous = current;
    delete temp;
    size--;
    return res;
}

void clear() {
    while (head != NULL) {
        current = head;
        head = head->next;
        delete current;
    }
    init();
}
```

El ciclo se repite hasta que la lista no tenga nodos. Se utiliza current para apuntar al nodo que se elimina, mientras head avanza al siguiente nodo de la lista. Se elimina el nodo por medio de current.

```
E remove() throw(runtime_error) {
    if (current->next == tail) {
        throw runtime_error("No element to remove.");
    }
    E res = current->next->element;
    DNode<E>* temp = current->next;
    current->next = current->next->next;
    current->next->previous = current;
    delete temp;
    size--;
    return res;
}

void clear() {
    while (head != NULL) {
        current = head;
        head = head->next;
        delete current;
    }
    init();
}
```

El método init se encarga de crear los nodos especiales e inicializar la lista vacía. Esto se hace por conveniencia ya que no es necesario tomar en cuenta los nodos especiales para borrar la lista.

```
E getElement() throw(runtime_error) {
    if (current->next == tail) {
        throw runtime_error("No element to get.");
    }
    return current->next->element;
}

void goToStart() {
    current = head;
}

void goToEnd() {
    current = tail->previous;
}

void goToPos(int newPos) throw(runtime_error) {
    if (newPos < 0 || newPos >= size) {
        throw runtime_error("Index out of bounds.");
    }
    current = head;
    for (int i = 0; i < newPos; i++) {
        current = current->next;
    }
}
```

El método puede fallar si no hay un elemento actual.


```
E getElement() throw(runtime_error) {
    if (current->next == tail) {
        throw runtime_error("No element to get.");
    }
    return current->next->element;
}

void goToStart() {
    current = head;
}

void goToEnd() {
    current = tail->previous;
}

void goToPos(int newPos) throw(runtime_error) {
    if (newPos < 0 || newPos >= size) {
        throw runtime_error("Index out of bounds.");
    }
    current = head;
    for (int i = 0; i < newPos; i++) {
        current = current->next;
    }
}
```

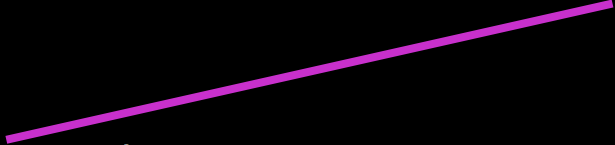
El puntero current apunta al nodo especial, que es el inicio de la lista.

```
E getElement() throw(runtime_error) {
    if (current->next == tail) {
        throw runtime_error("No element to get.");
    }
    return current->next->element;
}

void goToStart() {
    current = head;
}

void goToEnd() {
    current = tail->previous;
}

void goToPos(int newPos) throw(runtime_error) {
    if (newPos < 0 || newPos >= size) {
        throw runtime_error("Index out of bounds.");
    }
    current = head;
    for (int i = 0; i < newPos; i++) {
        current = current->next;
    }
}
```



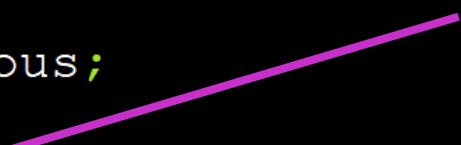
El puntero current apunta al nodo antes del nodo especial del final de la lista.

```
E getElement() throw(runtime_error) {
    if (current->next == tail) {
        throw runtime_error("No element to get.");
    }
    return current->next->element;
}

void goToStart() {
    current = head;
}

void goToEnd() {
    current = tail->previous;
}

void goToPos(int newPos) throw(runtime_error) {
    if (newPos < 0 || newPos >= size) {
        throw runtime_error("Index out of bounds.");
    }
    current = head;
    for (int i = 0; i < newPos; i++) {
        current = current->next;
    }
}
```



El método puede fallar si la posición indicada no es una posición válida.

```

E getElement() throw(runtime_error) {
    if (current->next == tail) {
        throw runtime_error("No element to get.");
    }
    return current->next->element;
}

void goToStart() {
    current = head;
}

void goToEnd() {
    current = tail->previous;
}

void goToPos(int newPos) throw(runtime_error) {
    if (newPos < 0 || newPos >= size) {
        throw runtime_error("Index out of bounds.");
    }
    current = head;
    for (int i = 0; i < newPos; i++) {
        current = current->next;
    }
}

```

Se utiliza la misma estrategia que en LinkedList, se ubica la posición actual al inicio de la lista y se utiliza un contador para avanzar hasta la posición indicada.

```
void next() {  
    if (current != tail->previous) {  
        current = current->next;  
    }  
}  
bool atEnd() {  
    return current->next == tail;  
}  
void previous() {  
    if (current != head) {  
        current = current->previous;  
    }  
}  
bool atStart() {  
    return current == head;  
}
```

Los métodos next, previous, atEnd y atStart son muy similares a los de LinkedList. El método next es diferente porque debe utilizar el anterior al nodo especial de tail.

```
int getPos() {  
    int pos = 0;  
    DNode<E>* temp = head;  
    while (temp != current) {  
        temp = temp->next;  
        pos++;  
    }  
    return pos;  
}  
int getSize() {  
    return size;  
}  
};
```

Implementación similar a la de LinkedList, se utiliza un contador mientras se avanza en la lista hasta encontrar la posición actual.

```
int getPos() {  
    int pos = 0;  
    DNode<E>* temp = head;  
    while (temp != current) {  
        temp = temp->next;  
        pos++;  
    }  
    return pos;  
}  
int getSize() {  
    return size;  
}  
};
```

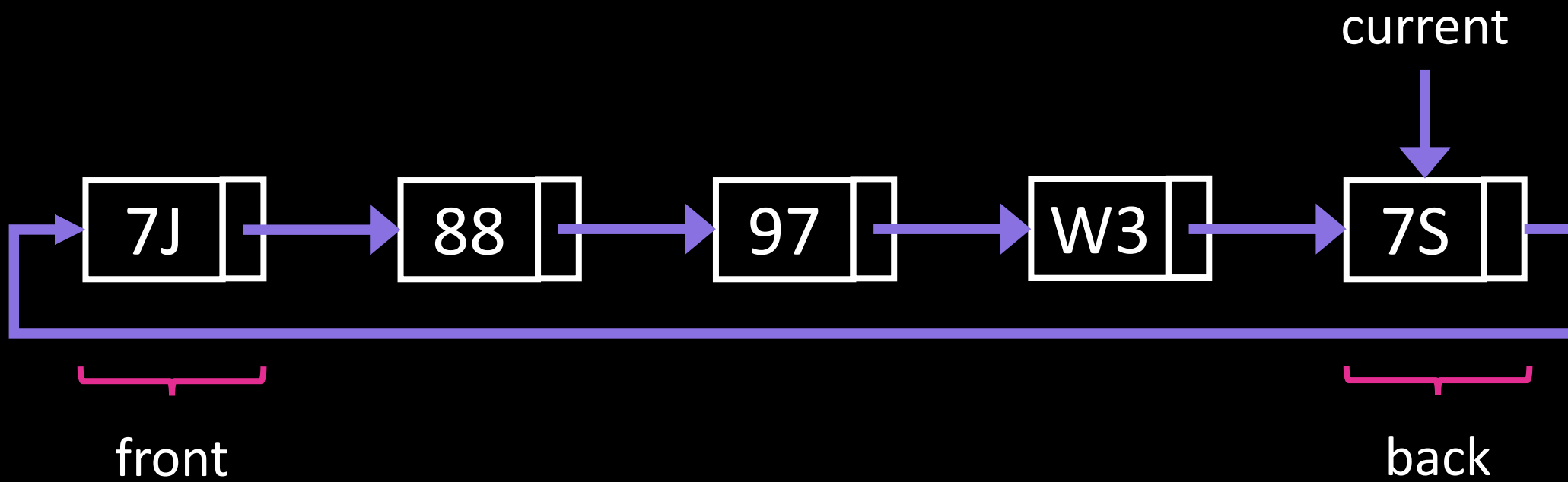
Se retorna el valor del atributo size.

Ejercicios con la clase DLinkedList

- Implementar los métodos que se agregaron a la clase abstracta List.
 - `contains`.
 - `indexOf`.
 - `extend`.
 - `reverse`.
 - `equals`.
- Crear clase `OrderedDLinkedList`, similar a la que se hizo con `LinkedList`.

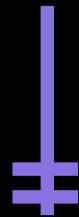
Listas circulares

- Son listas enlazadas donde el último elemento **apunta al primer elemento**
- Si es doblemente enlazada, el puntero al elemento anterior del **primer elemento apunta al último**
- Presenta algunos retos de programación, pero el código resulta más simple porque **no hay punteros nulos**

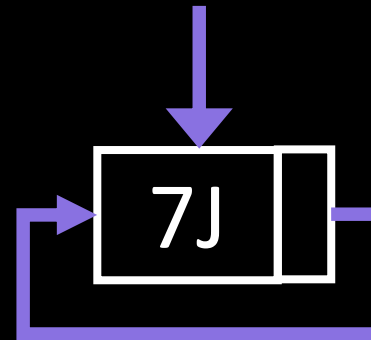


- Definición de nodos es **igual** a la lista simple o doblemente enlazada
- Desaparece la necesidad de dos punteros para inicio y final de la lista
- La lista **vacía** corresponde a un puntero **nulo**
- Surge un caso especial cuando sólo hay **un elemento**, debe apuntar a sí mismo

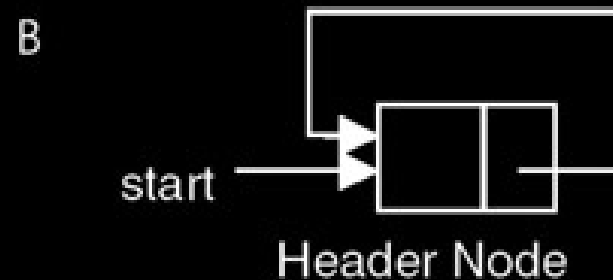
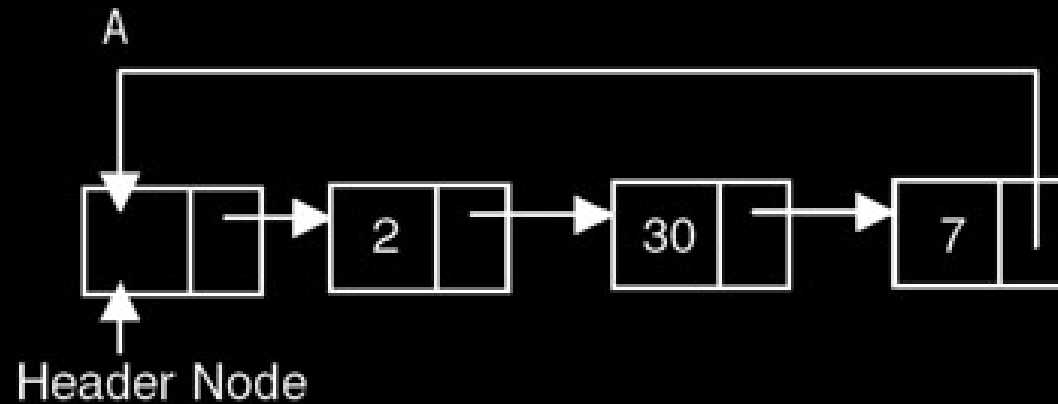
current

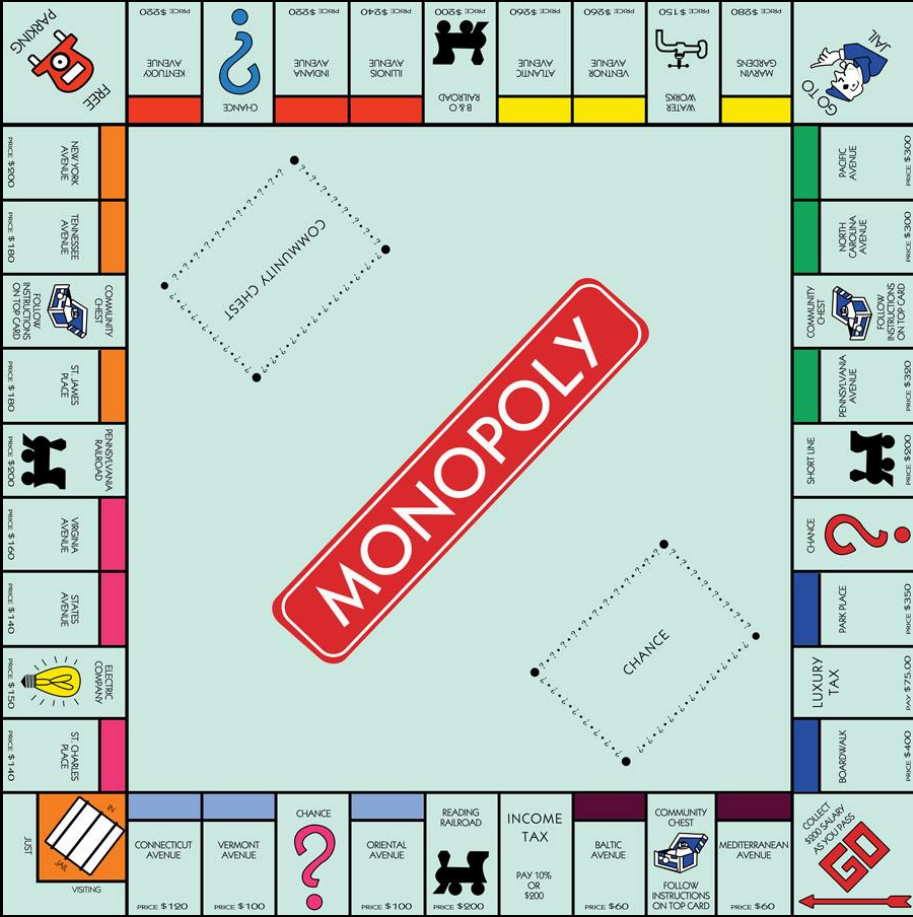


current



- También puede implementarse con un **nodo especial**
- El nodo ayuda a identificar cuál debe ser considerado el **primer** nodo de la lista
- Debe **diferenciarse** del resto con algún valor especial





<u>Operación</u>	<u>Descripción</u>
Insert	Inserta un nuevo elemento en el frente de la lista.
remove	Elimina el elemento que se encuentra en el frente de la lista. Retorna como resultado el valor del elemento que se eliminó.
clear	Elimina los contenidos de la lista y la deja vacía.
getFront	Retorna el elemento ubicado en el frente de la lista.
getBack	Retorna el elemento ubicado en el final de la lista.
previous	Mueve la posición actual a la posición anterior.
next	Mueve la posición actual a la posición siguiente.
getSize	Retorna un número entero con el tamaño actual de la lista.

```
#include <stdexcept>
#include "Node.h"

using std::runtime_error;

template <typename E>
class CircleList {
private:
    Node<E> *current;
    int size;
```

Sólo se necesita un puntero a la lista y el tamaño.

public:

```
CircleList() {  
    current = NULL;  
    size = 0;  
}  
~CircleList() {  
    clear();  
}
```

La inicialización es sencilla, simplemente se asigna nulo al puntero y se asigna 0 al tamaño.


```
public:
    CircleList() {
        current = NULL;
        size = 0;
    }
    ~CircleList() {
        clear();
    }
```

El método clear se encarga de borrar todos los elementos de la lista. No es necesario eliminar nada más porque no se cuenta con un nodo especial.

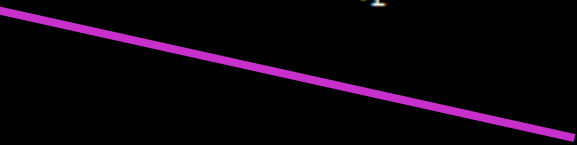
```
void insert(E pElement) {  
    if (current == NULL) {  
        current = new Node<E>(pElement);  
        current->next = current;  
    } else {  
        current->next = new Node<E>(pElement, current->next);  
    }  
    size++;  
}
```

Una de las consecuencias de no utilizar un nodo especial en esta lista es que hay que tener casos especiales para cuando la lista está vacía.

```
void insert(E pElement) {  
    if (current == NULL) {  
        current = new Node<E>(pElement);  
        current->next = current;  
    } else {  
        current->next = new Node<E>(pElement, current->next);  
    }  
    size++;  
}
```

Se crea el nodo con el elemento nuevo y se asigna la dirección al puntero current. Luego se asigna la dirección del nodo como siguiente del mismo.

```
void insert(E pElement) {  
    if (current == NULL) {  
        current = new Node<E>(pElement);  
        current->next = current;  
    } else {  
        current->next = new Node<E>(pElement, current->next);  
    }  
    size++;  
}
```



En caso de que la lista no esté vacía, se crea el nodo con el elemento enviado y se le asigna como siguiente la dirección del nodo que se encuentra de primero en la lista.
La dirección del nodo nuevo se asigna como siguiente del último nodo en la lista.
El nodo queda insertado como primero de la lista.

```
E remove() throw(runtime_error) {  
    if (current == NULL) {  
        throw runtime_error("Can't remove from an empty list.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    if (current == temp) {  
        current = NULL;  
    } else {  
        current->next = current->next->next;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

El método puede fallar si la lista no tiene ningún elemento.

```
E remove() throw(runtime_error) {  
    if (current == NULL) {  
        throw runtime_error("Can't remove from an empty list.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    if (current == temp) {  
        current = NULL;  
    } else {  
        current->next = current->next->next;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

Se guarda el valor del elemento a eliminar para retornarlo. También el puntero al nodo que se va a eliminar para no perder su dirección.
Se elimina el primero de la lista.

```
E remove() throw(runtime_error) {  
    if (current == NULL) {  
        throw runtime_error("Can't remove from an empty list.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    if (current == temp) {  
        current = NULL;  
    } else {  
        current->next = current->next->next;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

Si el puntero temporal es igual a current significa que sólo había un nodo en la lista. Podría evaluarse también mediante el atributo size.
Si esto se cumple, current debe ser nulo.

```
E remove() throw(runtime_error) {  
    if (current == NULL) {  
        throw runtime_error("Can't remove from an empty list.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    if (current == temp) {  
        current = NULL;  
    } else {  
        current->next = current->next->next;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

Si la lista tiene más de un elemento, el último nodo debe apuntar al nodo siguiente del que se elimina.


```
E remove() throw(runtime_error) {  
    if (current == NULL) {  
        throw runtime_error("Can't remove from an empty list.");  
    }  
    E result = current->next->element;  
    Node<E> *temp = current->next;  
    if (current == temp) {  
        current = NULL;  
    } else {  
        current->next = current->next->next;  
    }  
    delete temp;  
    size--;  
    return result;  
}
```

Se libera la memoria del nodo, se actualiza el tamaño de la lista y se retorna el valor del elemento eliminado.

```
void clear() {
    while (current != NULL) {
        remove();
    }
}

E getFront() throw(runtime_error) {
    if (current == NULL) {
        throw runtime_error("There's no element. Empty list.");
    }
    return current->next->element;
}

E getBack() throw(runtime_error) {
    if (current == NULL) {
        throw runtime_error("There's no element. Empty list.");
    }
    return current->element;
}
```

Mientras hayan elementos en la lista, se elimina el primero. También podría evaluarse el atributo de tamaño.

```
void clear() {  
    while (current != NULL) {  
        remove();  
    }  
}  
E getFront() throw(runtime_error) {  
    if (current == NULL) {  
        throw runtime_error("There's no element. Empty list.");  
    }  
    return current->next->element;  
}  
E getBack() throw(runtime_error) {  
    if (current == NULL) {  
        throw runtime_error("There's no element. Empty list.");  
    }  
    return current->element;  
}
```

El método puede fallar si la lista no tiene ningún elemento.

```
void clear() {
    while (current != NULL) {
        remove();
    }
}

E getFront() throw(runtime_error) {
    if (current == NULL) {
        throw runtime_error("There's no element. Empty list.");
    }
    return current->next->element;
}

E getBack() throw(runtime_error) {
    if (current == NULL) {
        throw runtime_error("There's no element. Empty list.");
    }
    return current->element;
}
```

Se retorna el elemento del nodo que se encuentra al inicio de la lista.

```
void clear() {  
    while (current != NULL) {  
        remove();  
    }  
}  
E getFront() throw(runtime_error) {  
    if (current == NULL) {  
        throw runtime_error("There's no element. Empty list.");  
    }  
    return current->next->element;  
}  
E getBack() throw(runtime_error) {  
    if (current == NULL) {  
        throw runtime_error("There's no element. Empty list.");  
    }  
    return current->element;  
}
```

Falla en las mismas condiciones que getFront.

```
void clear() {  
    while (current != NULL) {  
        remove();  
    }  
}  
E getFront() throw(runtime_error) {  
    if (current == NULL) {  
        throw runtime_error("There's no element. Empty list.");  
    }  
    return current->next->element;  
}  
E getBack() throw(runtime_error) {  
    if (current == NULL) {  
        throw runtime_error("There's no element. Empty list.");  
    }  
    return current->element;
```

Se retorna el elemento del último nodo de la lista.

```
void next() {  
    if (current != NULL && size > 1) {  
        current = current->next;  
    }  
}  
void previous() {  
    if (current != NULL && size > 1) {  
        Node<E>* temp = current;  
        while (current->next != temp) {  
            current = current->next;  
        }  
    }  
}  
int getSize() {  
    return size;  
}  
};
```

Si la lista tiene más de un elemento, se avanza el puntero current al siguiente nodo.

```
void next() {
    if (current != NULL && size > 1) {
        current = current->next;
    }
}

void previous() {
    if (current != NULL && size > 1) {
        Node<E>* temp = current;
        while (current->next != temp) {
            current = current->next;
        }
    }
}

int getSize() {
    return size;
}

};
```

Se busca el anterior sólo si la lista tiene más de un elemento. Se utiliza un puntero temporal que inicia en current y avanza hasta que el puntero siguiente es igual a current.


```
void next() {  
    if (current != NULL && size > 1) {  
        current = current->next;  
    }  
}  
void previous() {  
    if (current != NULL && size > 1) {  
        Node<E>* temp = current;  
        while (current->next != temp) {  
            current = current->next;  
        }  
    }  
}  
int getSize() {  
    return size;  
}  
};
```

Retornar el tamaño de la lista.

Ejercicios con la clase CircleList

- Implementar los métodos que se agregaron a las clases anteriores.
 - `contains`.
 - `extend`.
 - `reverse`.
 - `equals`.
- Implementar una clase llamada `DCircleList`
 - Similar a la clase `CircleList`, pero con nodos doblemente enlazados



Listas Enlazadas

Mauricio Avilés