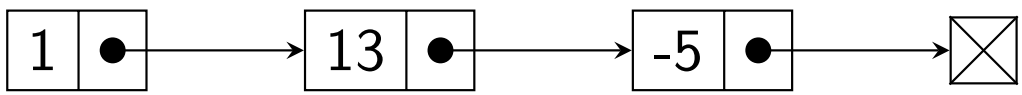
Listas enlazadas

Algoritmos y Estructuras de Datos II

Listas simplemente enlazadas

Una *lista simplemente enlazada* es una estructura que sirve para representar una secuencia de elementos.

Gráficamente



Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.
- Son eficientes para reacomodar elementos (útil para ordenar).

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.
- Son eficientes para reacomodar elementos (útil para ordenar).

¿Cuál es su desventaja?

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.
- Son eficientes para reacomodar elementos (útil para ordenar).

¿Cuál es su desventaja?
Perdemos el *acceso aleatorio* a los elementos.

Lista de Enteros

Implementemos la clase `ListaDeEnt`, sobre una lista simplemente enlazada, con los siguientes métodos:

```
class ListaDeEnt {
    public:
        ListaDeEnt();
        ~ListaDeEnt();
        void agregarAtras(int x);
        int longitud() const;
        int iesimo(int i) const;
}
```

Constructores por copia y memoria dinámica

Un **constructor por copia** es un constructor que tiene como parámetro una referencia a otra instancia de la misma clase.

- La nueva instancia se inicializa como una copia de aquella recibida por parámetro.

```
int main() {
    ListaDeEnt l1;
    l1.agregarAtras(1);
    ListaDeEnt l2(l1);
    l2.agregarAtras(2);
    l1.longitud(); // ??
}
```

Constructores por copia y memoria dinámica

Un **constructor por copia** es un constructor que tiene como parámetro una referencia a otra instancia de la misma clase.

- La nueva instancia se inicializa como una copia de aquella recibida por parámetro.

```
int main() {
    ListaDeEnt l1;
    l1.agregarAtras(1);
    ListaDeEnt l2(l1);
    l2.agregarAtras(2);
    l1.longitud(); // ??
}
```

¿Qué pasa si *no* implementamos nuestro propio constructor por copia?

Constructores por copia y memoria dinámica

El compilador de C++ provee un constructor por copia por defecto, el cual realiza una copia **únicamente** de los campos de la clase (*shallow copy*).

Por lo tanto, al usar memoria dinámica, muy posiblemente tengamos *aliasing* entre instancias (i.e., dos variables distintas apuntando a la misma instancia).

- Corremos peligro de romper otras instancias y de perder memoria (!)

Constructores por copia y memoria dinámica

El compilador de C++ provee un constructor por copia por defecto, el cual realiza una copia **únicamente** de los campos de la clase (*shallow copy*).

Por lo tanto, al usar memoria dinámica, muy posiblemente tengamos *aliasing* entre instancias (i.e., dos variables distintas apuntando a la misma instancia).

- Corremos peligro de romper otras instancias y de perder memoria (!)

Agreguemos entonces:

```
ListaDeEnt(const ListaDeEnt& o);
```

Operador de asignación y memoria dinámica

¿Y cuando realizamos una asignación?

```
int main() {
    ListaDeEnt l1;
    l1.agregarAtras(1);
    ListaDeEnt l2;
    l2.agregarAtras(2);
    l2 = l1;
    l2.agregarAtras(3);
    l1.longitud(); // ??
}
```

Operador de asignación y memoria dinámica

¿Y cuando realizamos una asignación?

```
int main() {
    ListaDeEnt l1;
    l1.agregarAtras(1);
    ListaDeEnt l2;
    l2.agregarAtras(2);
    l2 = l1;
    l2.agregarAtras(3);
    l1.longitud(); // ??
}
```

El compilador de C++ también provee una asignación por defecto que copia los campos de la clase.

- Pero en este caso es aún peor, ya que, si teníamos algún valor, lo acabamos de perder en el éter!

Operador de asignación y memoria dinámica

¿Y cuando realizamos una asignación?

```
int main() {
    ListaDeEnt l1;
    l1.agregarAtras(1);
    ListaDeEnt l2;
    l2.agregarAtras(2);
    l2 = l1;
    l2.agregarAtras(3);
    l1.longitud(); // ??
}
```

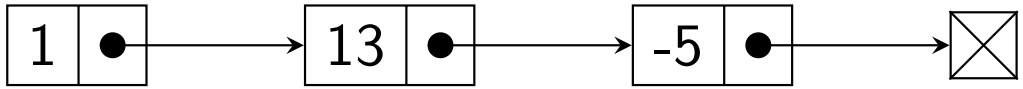
El compilador de C++ también provee una asignación por defecto que copia los campos de la clase.

- Pero en este caso es aún peor, ya que, si teníamos algún valor, lo acabamos de perder en el éter!

Agreguemos:

```
ListaDeEnt& operator= (const ListaDeEnt& o);
```

Lista de Enteros




```
class ListaDeEnt {
    public:
        ListaDeEnt();
        ListaDeEnt(const ListaDeEnt& o);
        ~ListaDeEnt();

        void agregarAtras(int x);
        int longitud() const;
        int iesimo(int i) const;

        ListaDeEnt& operator= (const ListaDeEnt& o);
    private:
        ...
}
```

Lista de Enteros: Una posible solución

```
class ListaDeEnt {
public:
    ListaDeEnt();
    ListaDeEnt(const ListaDeEnt& o);
    ~ListaDeEnt();

    void agregarAtras(int x);
    int longitud() const;
    int iesimo(int i) const;

    ListaDeEnt& operator=(const ListaDeEnt& o);
private:
    struct Nodo {
        int valor;
        Nodo* sig;
        Nodo(int v, Nodo* s) : valor(v), sig(s) {}
    };
    Nodo* prim;

    void copiarNodos(const ListaDeEnt &o);
    void destruirNodos();
};
```

Lista de Enteros: Una posible solución (2)

```
ListaDeEnt::ListaDeEnt() : prim(NULL) { }

ListaDeEnt::ListaDeEnt(const ListaDeEnt& o) : prim(NULL) {
    copiarNodos(o); }

ListaDeEnt::~ListaDeEnt() {
    destruirNodos(); }

void ListaDeEnt::agregarAtras(int x) {
    Nodo* nuevo = new Nodo(x, NULL);
    if (prim == NULL) {
        prim = nuevo;
        return;
    }

    Nodo* actual = prim;
    while(actual->sig != NULL) {
        actual = actual -> sig;
    }
    actual->sig = nuevo;
}
```

Lista de Enteros: Una posible solución (3)

```
int ListaDeEnt::longitud() const {
    Nodo* actual = prim;
    int contador = 0;
    while (actual != NULL) {
        contador++;
        actual = actual->sig;
    }
    return contador;
}

int ListaDeEnt::iesimo(int i) const {
    Nodo* actual = prim;
    for (int j = 0; j < i; ++j) {
        actual = actual->sig;
    }
    return actual->valor;
}
```

Lista de Enteros: Una posible solución (4)

```
ListaDeEnt& ListaDeEnt::operator=(const ListaDeEnt& o) {
    destruirNodos();
    copiarNodos(o);
    return *this;
}

void ListaDeEnt::copiarNodos(const ListaDeEnt &o) {
    Nodo* actual = o.prim;
    while (actual != NULL) {
        agregarAtras(actual->valor);
        actual = actual->sig;
    }
}
void ListaDeEnt::destruirNodos() {
    Nodo* actual = prim;
    while (actual != NULL) {
        Nodo* siguiente = actual->sig;
        delete actual;
        actual = siguiente;
    }
    prim = NULL;
}
```