



Listas con Arreglos

Mauricio Avilés

Contenido

- Concepto de Lista
- Características
- Clase abstracta List
- Operaciones
- Implementación de ArrayList
- Ejemplo de utilización
- Ejercicios con ArrayList
- Lista ordenada
- Implementación de SortedArrayList
- Ejemplo de utilización
- Ejercicios con SortedArrayList

Lecturas

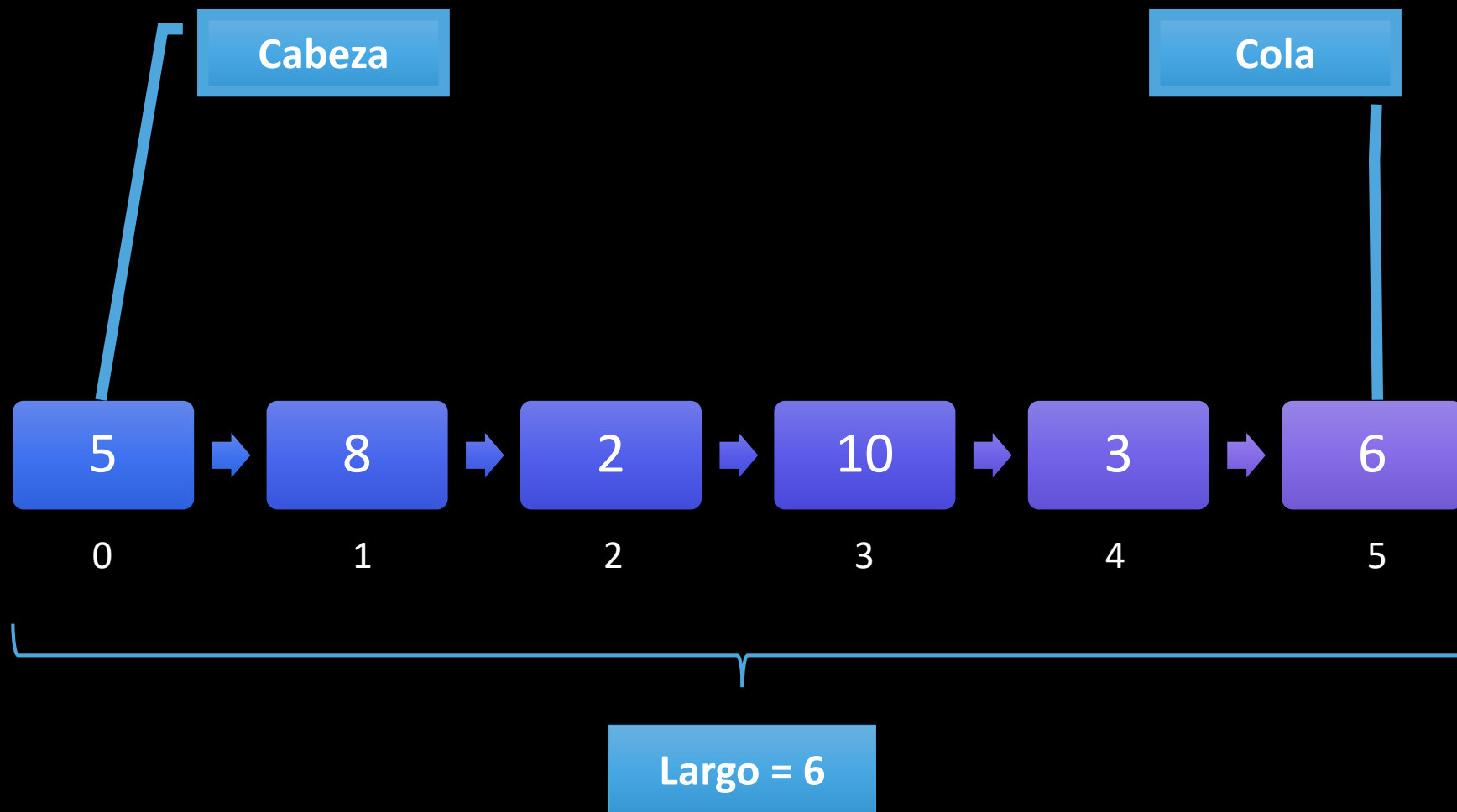
- Sección 4.1
 - Shaffer, C. A. "Data Structures & Algorithm Analysis in C++" (3rd ed., Dover). Mineola, NY. 2011.
- Sección 3.1
 - Goodrich, M. T., Tamassia, R., & Mount, D. M. "Data structures and algorithms in C++" (2nd ed., Wiley). Hoboken, NJ: Wiley. 2011.

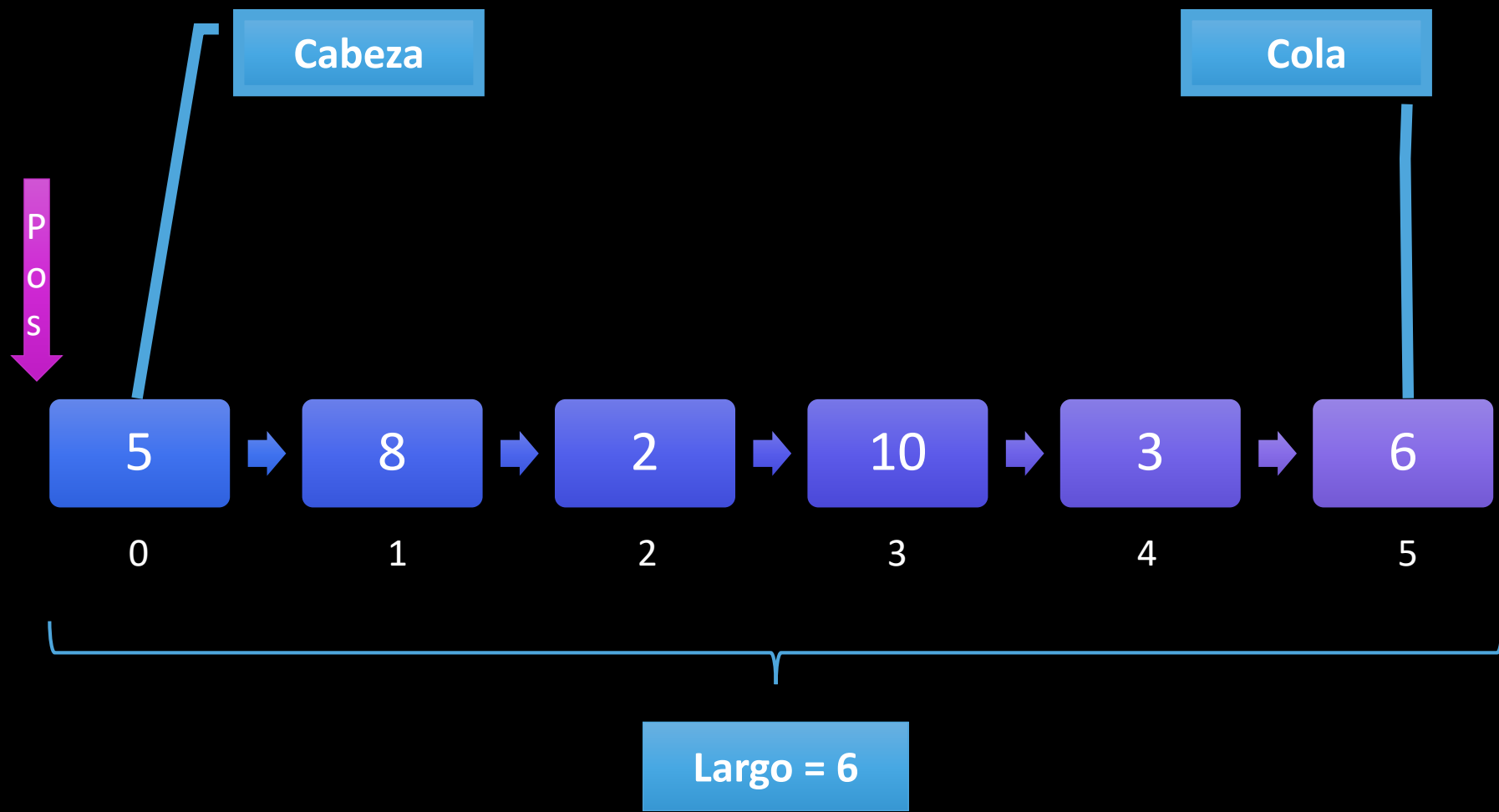
Listas

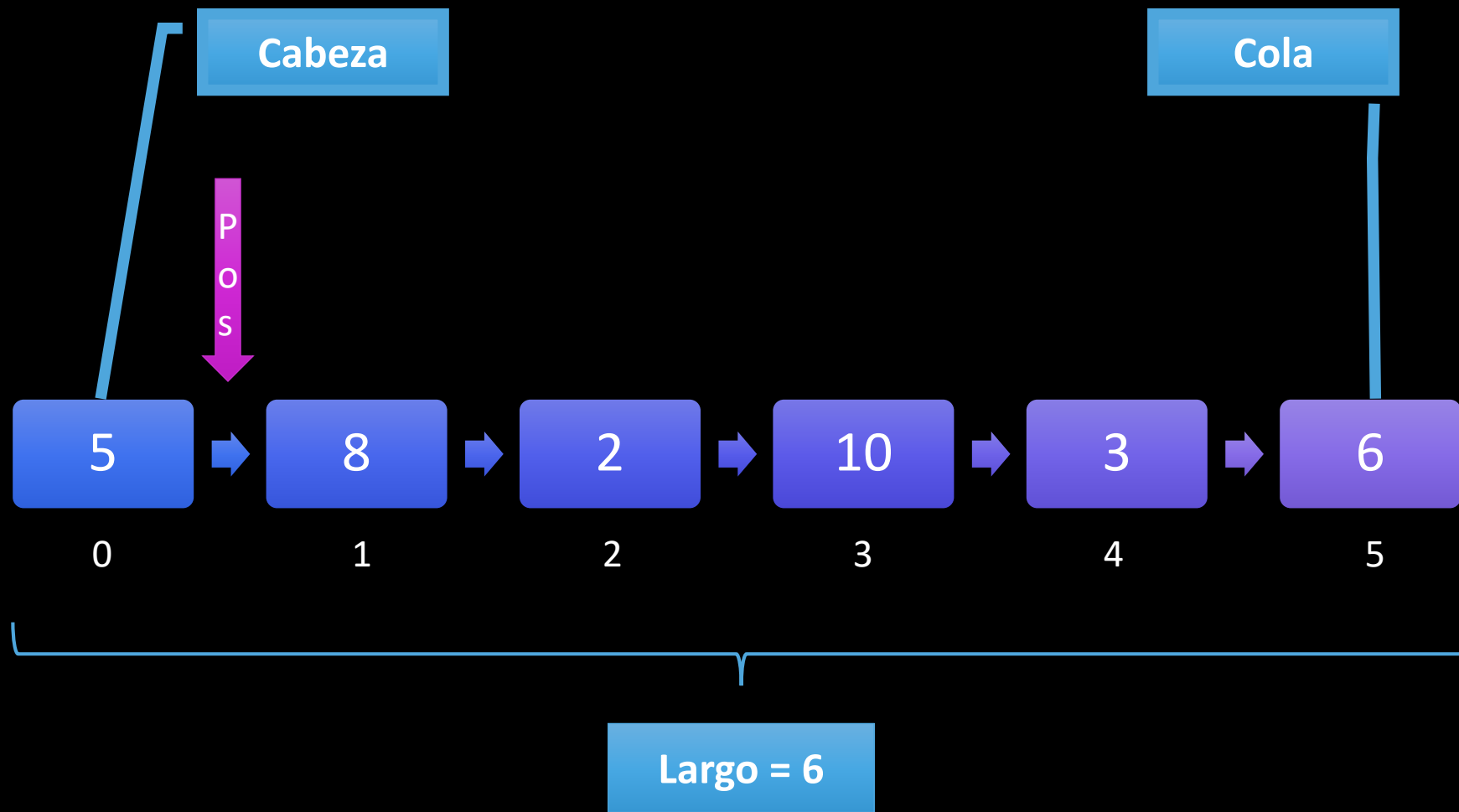
- **Estructura** para almacenar una **secuencia** finita de **valores**
- Los valores son conocidos como **elementos**
- Cada elemento en la lista tiene una **posición**
- Flexible
 - Crecer y hacerse más corta
 - Inserciones y borrados en cualquier posición
- Son **útiles** cuando se desea mantener registro de los elementos y poder agregar o eliminar elementos según sea necesario

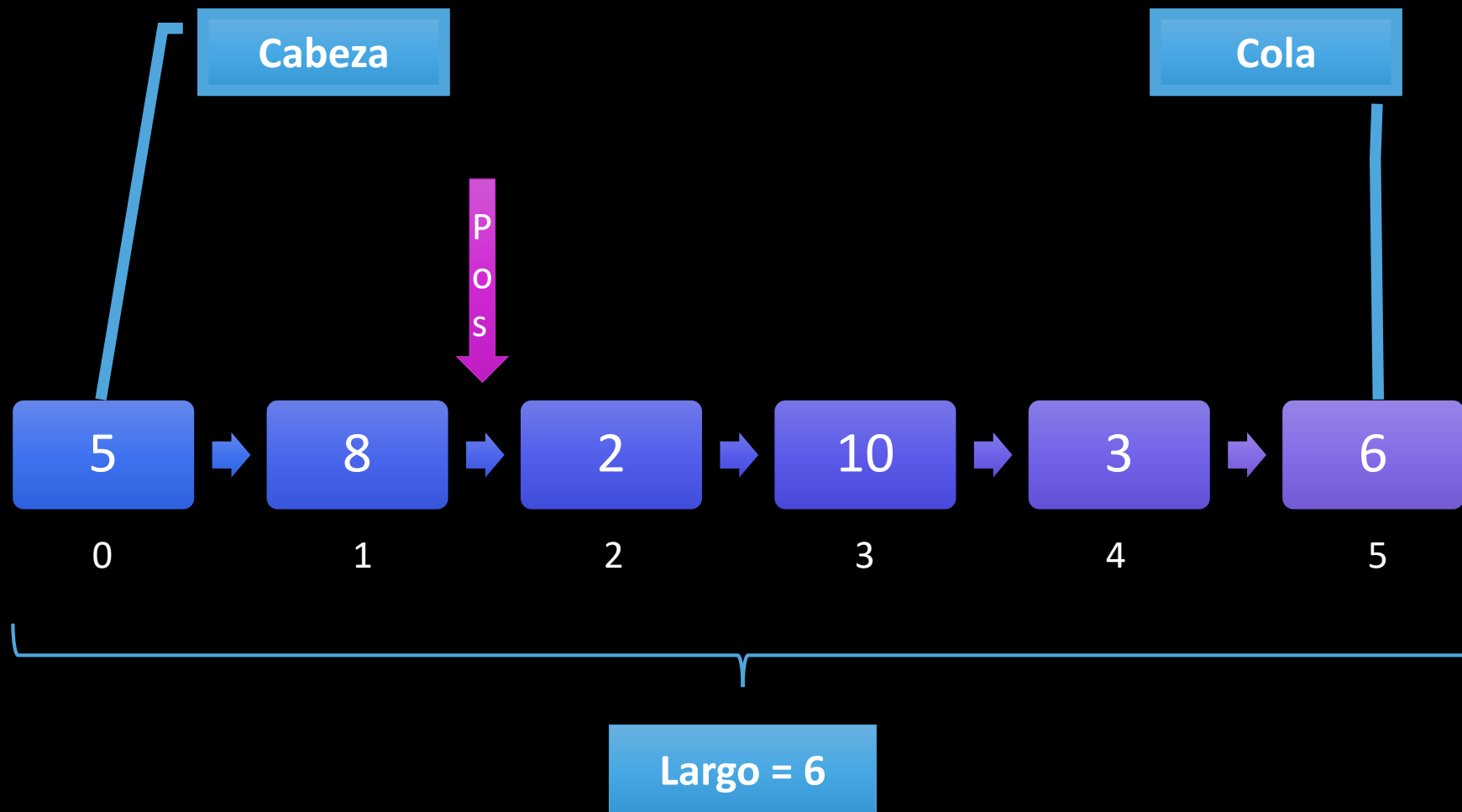
- No existe una **relación** entre la posición de un elemento y su valor
 - Puede estar **ordenada** o **desordenada**
- La lista puede tener **cero elementos**
- El tipo datos puede ser igual o diferente para cada elemento
 - Caso más simple → **mismo tipo**
- Se utilizará un indicador de **posición actual**

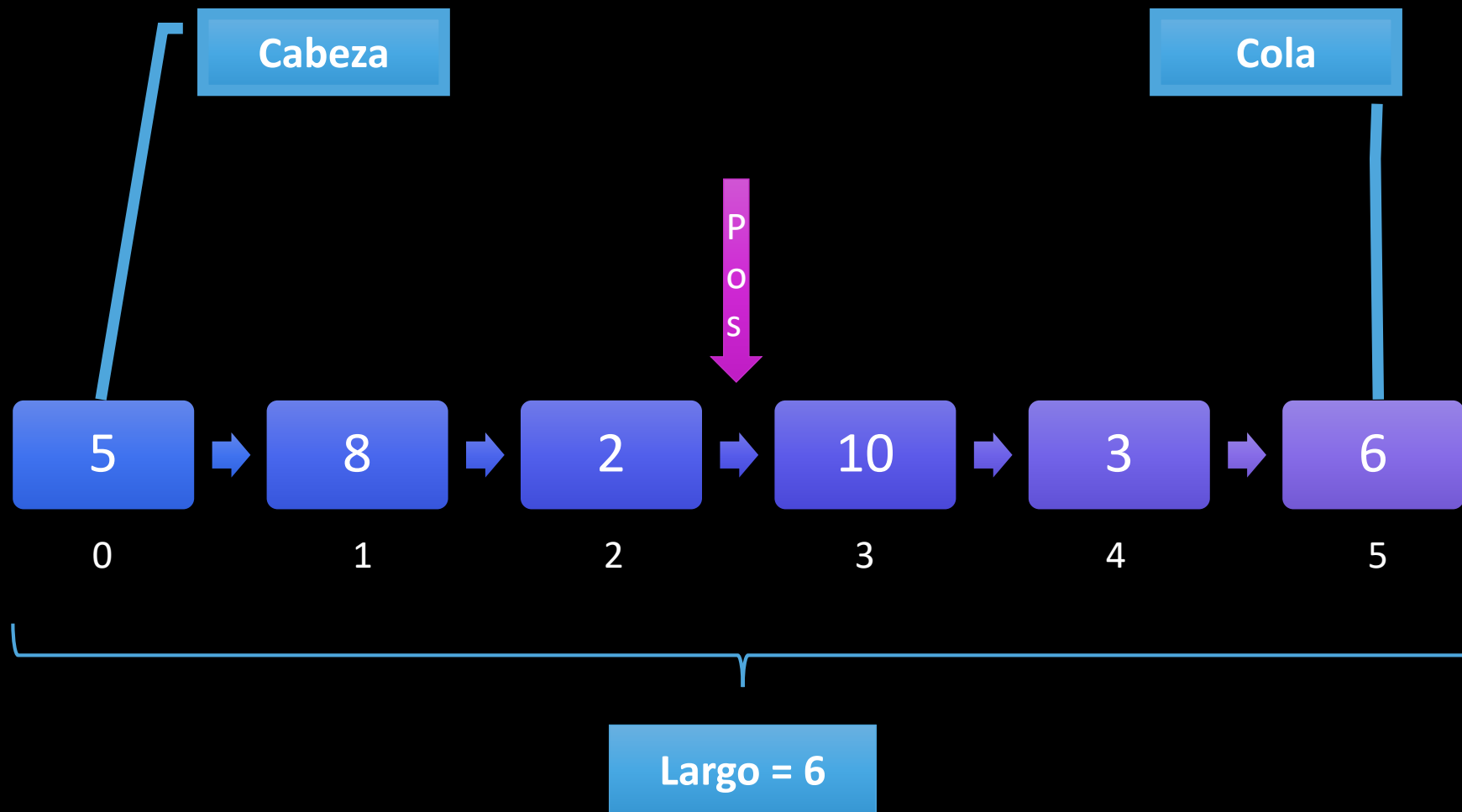


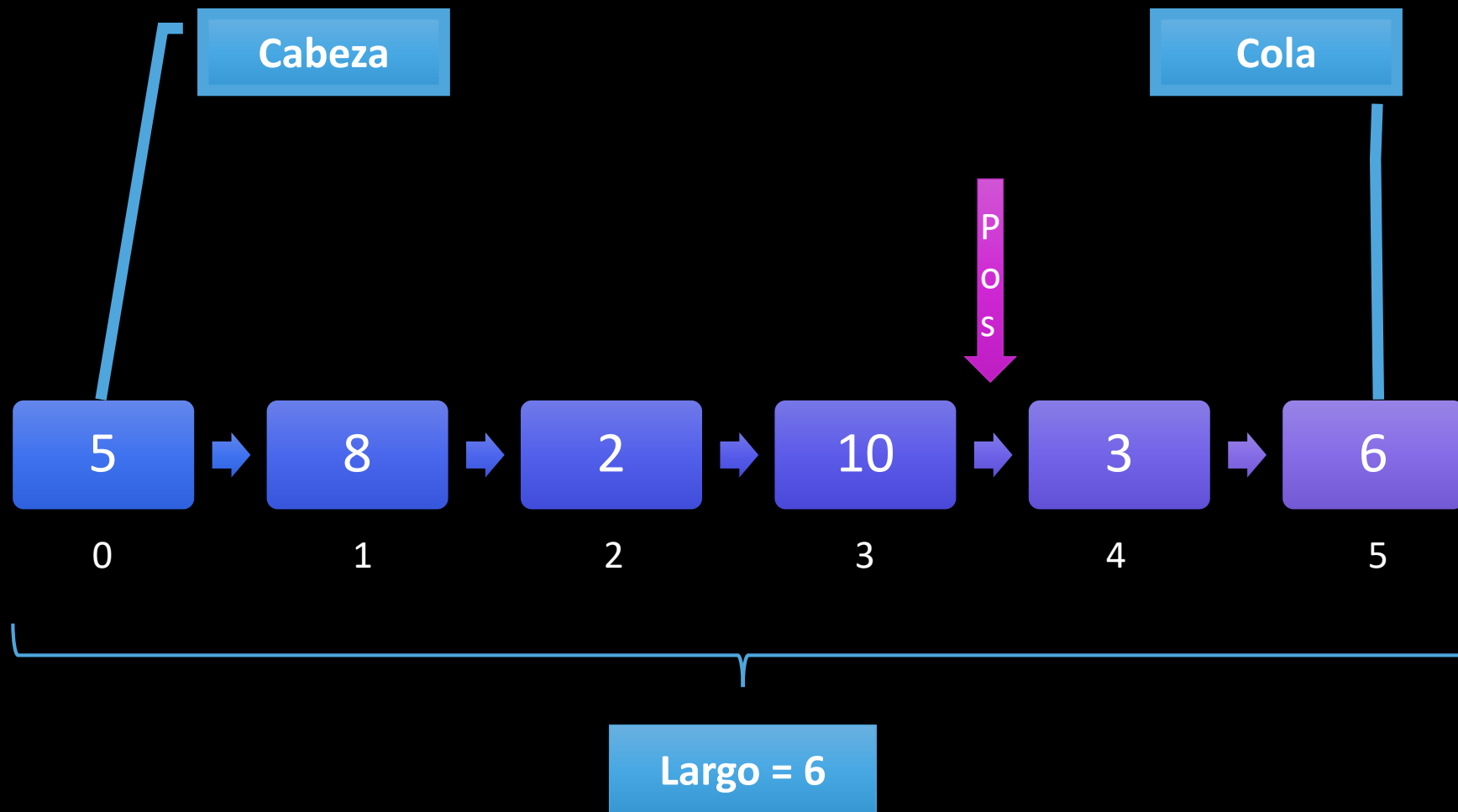


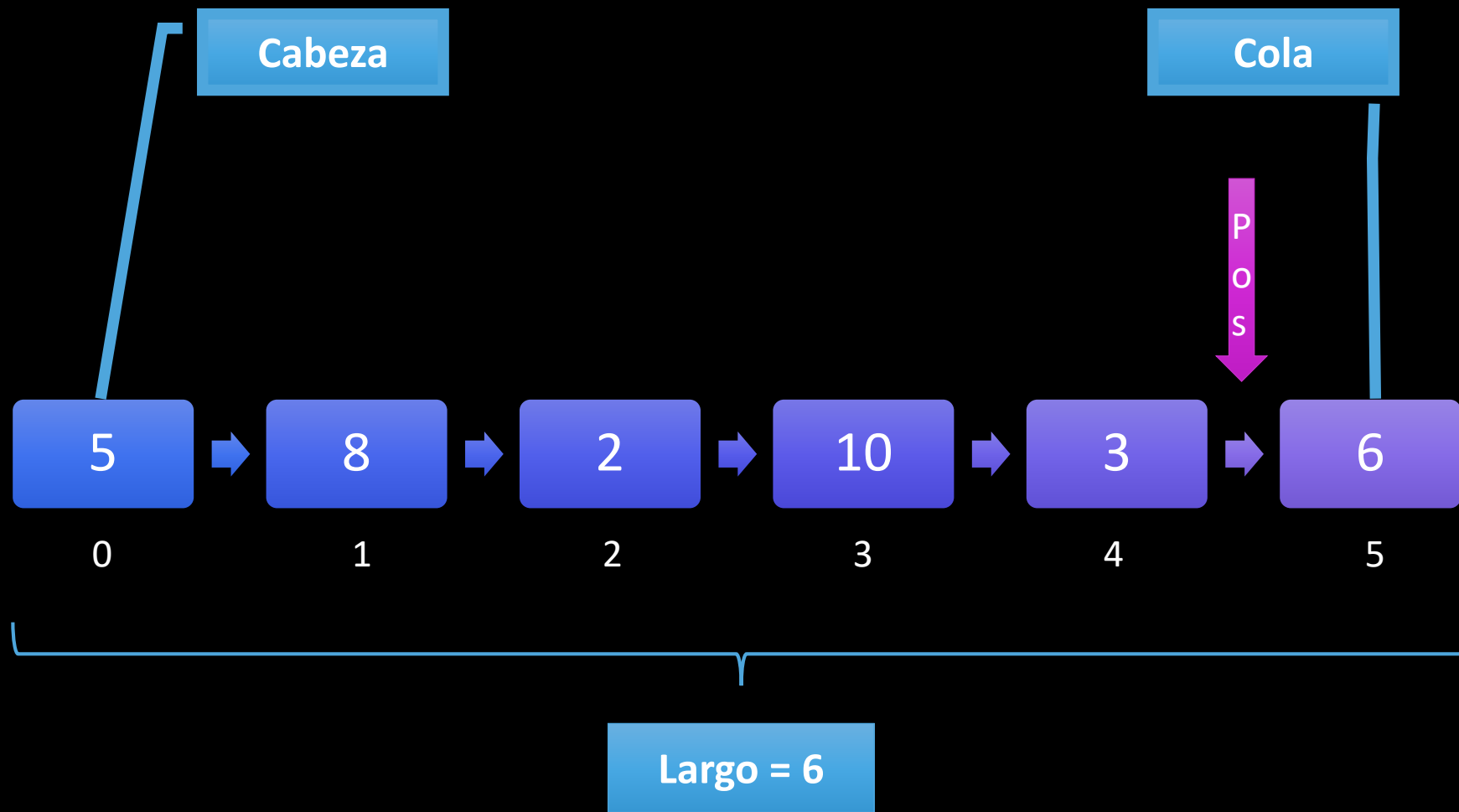


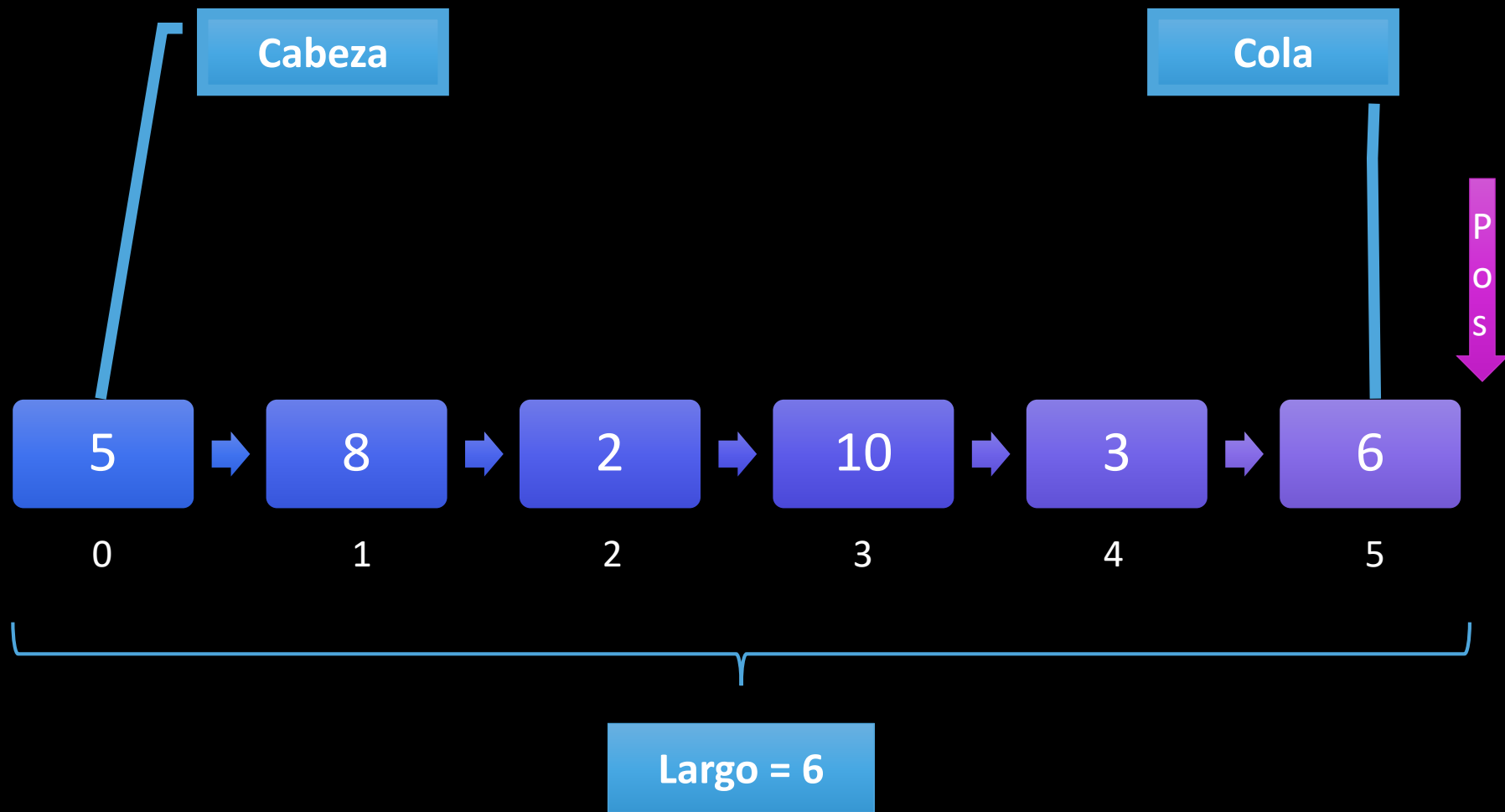












Algunas características

- La lista debe **cambiar su tamaño** según su cantidad de elementos
- El tipo de los elementos debe ser **genérico**
- La lista debe manejar situaciones anómalas por medio de **excepciones**
- Se utilizará una **clase abstracta base** con la definición de las operaciones que debe realizar una lista
- Se harán **diferentes implementaciones** concretas de la lista según la forma en que se manejen internamente los elementos

<u>Operación</u>	<u>Descripción</u>
insert	Inserta un nuevo elemento en la posición actual de la lista.
append	Agrega un nuevo elemento en la última posición de la lista, es decir, en la cola.
remove	Elimina el elemento que se encuentra en la posición actual de la lista. Retorna como resultado el valor del elemento que se eliminó.
clear	Elimina los contenidos de la lista y la deja vacía.
getElement	Retorna el elemento ubicado en la posición actual de la lista.
goToStart	Mueve la posición actual al inicio de la lista.
goToEnd	Mueve la posición actual al final de la lista.
goToPos	Mueve la posición actual a la posición indicada por parámetro.
previous	Mueve la posición actual a la posición anterior.
next	Mueve la posición actual a la posición siguiente.
atStart	Dice si la posición actual está al inicio de la lista.
atEnd	Dice si la posición actual está al final de la lista.
getPos	Retorna un número entero con la posición actual.
getSize	Retorna un número entero con el tamaño actual de la lista.


```
template <typename E>
class List {
private:
    void operator =(const List&) {}
    List(const List& obj) {}

public:
    List() {}
    virtual ~List() {}
    virtual void insert(E element) = 0;
    virtual void append(E element) = 0;
    virtual E remove() = 0;
    virtual void clear() = 0;
    virtual E getElement() = 0;
    virtual void goToStart() = 0;
    virtual void goToEnd() = 0;
    virtual void goToPos(int pos) = 0;
    virtual void next() = 0;
    virtual bool atEnd() = 0;
    virtual void previous() = 0;
    virtual bool atStart() = 0;
    virtual int getPos() = 0;
    virtual int getSize() = 0;
};
```

```
template <typename E>
class List {
private:
    void operator =(const List&) {}
    List(const List& obj) {}

public:
    List() {}
    virtual ~List() {}
    virtual void insert(E element) = 0;
    virtual void append(E element) = 0;
    virtual E remove() = 0;
    virtual void clear() = 0;
    virtual E getElement() = 0;
    virtual void goToStart() = 0;
    virtual void goToEnd() = 0;
    virtual void goToPos(int pos) = 0;
    virtual void next() = 0;
    virtual bool atEnd() = 0;
    virtual void previous() = 0;
    virtual bool atStart() = 0;
    virtual int getPos() = 0;
    virtual int getSize() = 0;
};
```

Tipo genérico de los elementos a guardar en la lista. E es el nombre del tipo que se utilizará en los elementos de la clase. Este nombre es personalizable.

Todos los métodos que realizan trabajo con los elementos de la lista deben utilizar este tipo.

El tipo que se utilizará en lugar de E se conocerá en tiempo de ejecución.

```
template <typename E>
class List {
private:
    void operator =(const List&) {}
    List(const List& obj) {}

public:
    List() {}
    virtual ~List() {}
    virtual void insert(E element) = 0;
    virtual void append(E element) = 0;
    virtual E remove() = 0;
    virtual void clear() = 0;
    virtual E getElement() = 0;
    virtual void goToStart() = 0;
    virtual void goToEnd() = 0;
    virtual void goToPos(int pos) = 0;
    virtual void next() = 0;
    virtual bool atEnd() = 0;
    virtual void previous() = 0;
    virtual bool atStart() = 0;
    virtual int getPos() = 0;
    virtual int getSize() = 0;
};
```

Protección del operador de asignación. Esto evita que se puedan asignar variables de este tipo.

```
template <typename E>
class List {
private:
    void operator =(const List&) {}
    List(const List& obj) {}

public:
    List() {}
    virtual ~List() {}
    virtual void insert(E element) = 0;
    virtual void append(E element) = 0;
    virtual E remove() = 0;
    virtual void clear() = 0;
    virtual E getElement() = 0;
    virtual void goToStart() = 0;
    virtual void goToEnd() = 0;
    virtual void goToPos(int pos) = 0;
    virtual void next() = 0;
    virtual bool atEnd() = 0;
    virtual void previous() = 0;
    virtual bool atStart() = 0;
    virtual int getPos() = 0;
    virtual int getSize() = 0;
};
```

Protección del constructor de copia. Esto evita que se pueda utilizar dicho constructor para copiar un objeto de este tipo.

Este método recibe como parámetro un objeto del mismo tipo de la clase. El const indica que el objeto no se puede modificar dentro del método y el & después del tipo indica que se recibe una referencia al objeto.

```
template <typename E>
class List {
private:
    void operator =(const List&) {}
    List(const List& obj) {}

public:
    List() {}
    virtual ~List() {}
    virtual void insert(E element) = 0;
    virtual void append(E element) = 0;
    virtual E remove() = 0;
    virtual void clear() = 0;
    virtual E getElement() = 0;
    virtual void goToStart() = 0;
    virtual void goToEnd() = 0;
    virtual void goToPos(int pos) = 0;
    virtual void next() = 0;
    virtual bool atEnd() = 0;
    virtual void previous() = 0;
    virtual bool atStart() = 0;
    virtual int getPos() = 0;
    virtual int getSize() = 0;
};
```

Constructor y destructor de la clase. El destructor es virtual porque las subclases deben implementarlo a su manera.

```
template <typename E>
class List {
private:
    void operator =(const List&) {}
    List(const List& obj) {}

public:
    List() {}
    virtual ~List() {}
    virtual void insert(E element) = 0;
    virtual void append(E element) = 0;
    virtual E remove() = 0;
    virtual void clear() = 0;
    virtual E getElement() = 0;
    virtual void goToStart() = 0;
    virtual void goToEnd() = 0;
    virtual void goToPos(int pos) = 0;
    virtual void next() = 0;
    virtual bool atEnd() = 0;
    virtual void previous() = 0;
    virtual bool atStart() = 0;
    virtual int getPos() = 0;
    virtual int getSize() = 0;
};
```

Métodos abstractos de la clase. Son virtual porque pueden sobrescribirse en las subclases. El =0 indica que no tiene implementación.

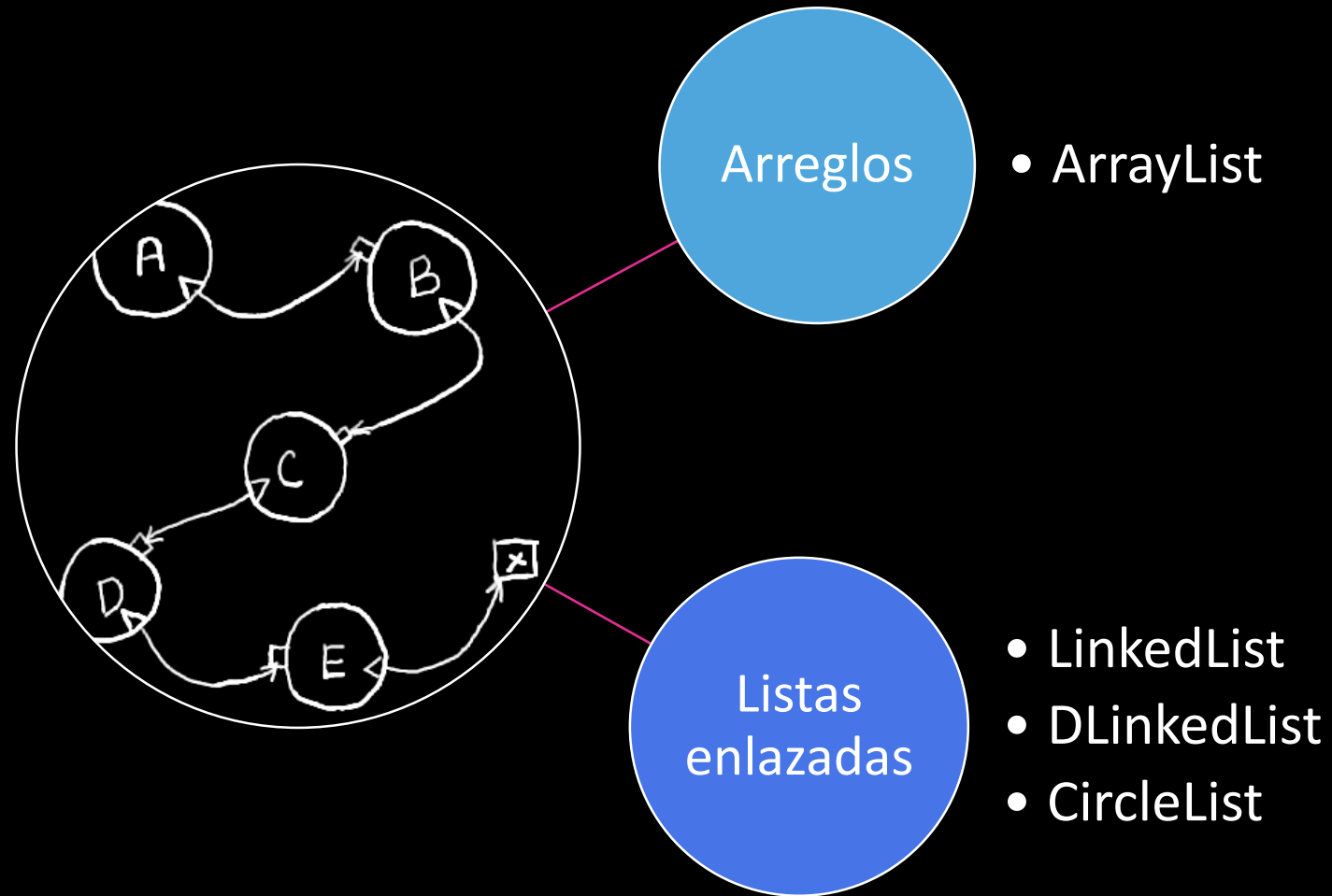
Recorriendo la lista

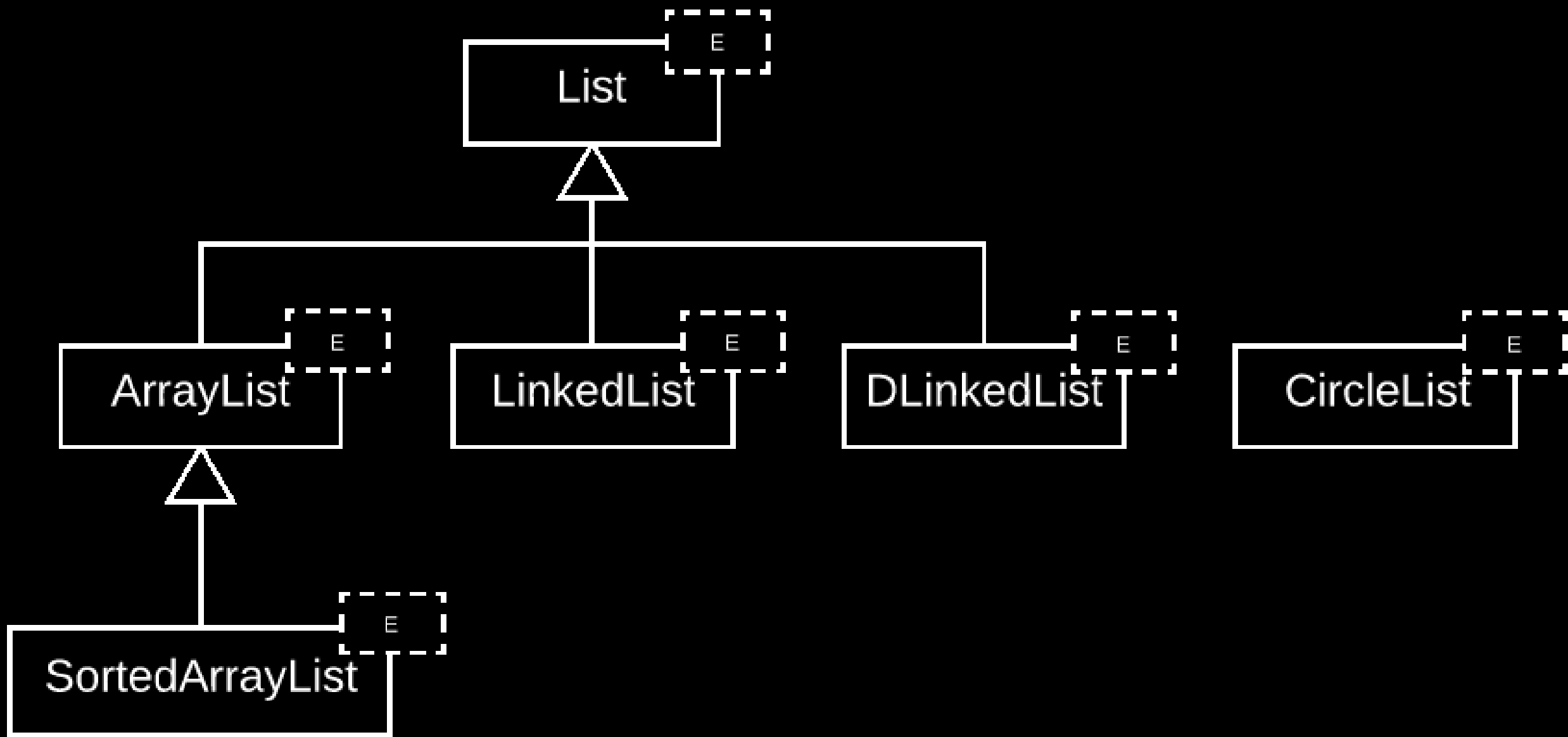
```
for (L->goToStart(); !L->atEnd(); L->next()) {  
    cout << L->getElement() << endl;  
}
```


Encontrar un elemento

```
bool find(List<int> *L, int element) {  
    for (L->goToStart(); !L->atEnd(); L->next()) {  
        if (element == L->getElement()) {  
            return true;  
        }  
    }  
    return false;  
}
```

Implementaciones





Listas implementadas con arreglos

- Implementación basada en arreglos
- Utilizar arreglo en memoria dinámica
- Los arreglos en C++ no pueden cambiar de tamaño una vez creados
- Se puede especificar su tamaño al crearlos

```
#define DEFAULT_MAX_SIZE 1024
#include "List.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayList : public List<E> {
private:
    E* elements;
    int max;
    int size;
    int pos;
```

Constante de compilador. Todas las ocurrencias del identificador DEFAULT_MAX_SIZE serán sustituidas por el valor 1024 a la hora de compilar. Esto se utilizará para indicar el tamaño máximo por defecto.

```
#define DEFAULT_MAX_SIZE 1024
#include "List.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayList : public List<E> {
private:
    E* elements;
    int max;
    int size;
    int pos;
```

Se importa la clase base List que es la que define cuáles métodos deben implementarse.

```
#define DEFAULT_MAX_SIZE 1024
#include "List.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayList : public List<E> {
private:
    E* elements;
    int max;
    int size;
    int pos;
```

Esta biblioteca contiene la definición de diferentes tipos de excepciones.


```
#define DEFAULT_MAX_SIZE 1024
#include "List.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayList : public List<E> {
private:
    E* elements;
    int max;
    int size;
    int pos;
```

Se indica que se utilizará el tipo `runtime_error` para lanzar errores dentro de la clase. Esta clase pertenece al espacio de nombres `std`.

```
#define DEFAULT_MAX_SIZE 1024
#include "List.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayList : public List<E> {
private:
    E* elements;
    int max;
    int size;
    int pos;
```

Tipo genérico de los elementos a guardar en la lista.

```
#define DEFAULT_MAX_SIZE 1024
#include "List.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayList : public List<E> {
private:
    E* elements;
    int max;
    int size;
    int pos;
```

Se indica que la clase ArrayList hereda de la clase List. El modificador public indica que esta herencia es pública, por lo que es posible utilizar referencias de tipo List para apuntar a objetos de sus subclasses.

```
#define DEFAULT_MAX_SIZE 1024
#include "List.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayList : public List<E> {
private:
    E* elements;
    int max;
    int size;
    int pos;
```

Los miembros declarados bajo la etiqueta private no podrán ser accedidos a la hora de usar la lista. Esto se utiliza para garantizar el encapsulamiento.

```
#define DEFAULT_MAX_SIZE 1024
#include "List.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayList : public List<E> {
private:
    E* elements;
    int max;
    int size;
    int pos;
```

Este puntero al tipo E se utilizará para apuntar al arreglo en memoria dinámica donde se almacenan los elementos de la lista. Esto es posible debido a la equivalencia entre punteros y nombres de arreglo.

```
#define DEFAULT_MAX_SIZE 1024
#include "List.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayList : public List<E> {
private:
    E* elements;
    int max;
    int size;
    int pos;
```

Cantidad máxima de elementos que pueden guardarse en la lista. Esto es debido a que el arreglo no puede cambiar de tamaño una vez que ha sido creado, por lo que se especifica un límite.

```
#define DEFAULT_MAX_SIZE 1024
#include "List.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayList : public List<E> {
private:
    E* elements;
    int max;
    int size;
    int pos;
```

Cantidad actual de elementos en la lista.


```
#define DEFAULT_MAX_SIZE 1024
#include "List.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayList : public List<E> {
private:
    E* elements;
    int max;
    int size;
    int pos;
```

Posición actual dentro de la lista. Este atributo puede tener un valor desde 0 hasta la cantidad de elementos en la lista.

public:

```
ArrayList(int maxSize = DEFAULT_MAX_SIZE) {  
    elements = new E[maxSize];  
    max = maxSize;  
    size = 0;  
    pos = 0;  
}  
~ArrayList() {  
    delete [] elements;  
}
```

Constructor de la clase. Se encarga de inicializar los atributos y de solicitar los recursos necesarios para el objeto.

```
public:
    ArrayList(int maxSize = DEFAULT_MAX_SIZE) {
        elements = new E[maxSize];
        max = maxSize;
        size = 0;
        pos = 0;
    }
    ~ArrayList() {
        delete [] elements;
    }
}
```

Esto indica el valor por defecto del parámetro maxSize en caso de que no se especifique ninguno. Se utiliza la constante mencionada anteriormente.

```
public:
    ArrayList(int maxSize = DEFAULT_MAX_SIZE) {
        elements = new E[maxSize];
        max = maxSize;
        size = 0;
        pos = 0;
    }
    ~ArrayList() {
        delete [] elements;
    }
}
```

Creación del arreglo de elementos del tamaño indicado. También se inicializan los valores de los atributos. La lista inicia sin elementos y en la posición actual 0.

```
public:
    ArrayList(int maxSize = DEFAULT_MAX_SIZE) {
        elements = new E[maxSize];
        max = maxSize;
        size = 0;
        pos = 0;
    }
    ~ArrayList() {
        delete [] elements;
    }
```

Destructor de la clase. Se invoca cuando se libera la memoria donde fue creado el objeto. Si el objeto fue creado en memoria estática, se ejecuta cuando termina el alcance de la variable dentro del código. Si fue creado en memoria dinámica se ejecuta cuando se ejecuta la instrucción delete con el puntero respectivo.

Este método libera la memoria dinámica solicitada para el arreglo de elementos. Debe utilizarse [] para indicar que se está borrando un arreglo, si no, liberaría sólo la memoria correspondiente al primer elemento.

```
void insert(E element) throw(runtime_error) {
    if (size == max) {
        throw runtime_error("List full");
    }
    for (int i = size; i > pos; i--) {
        elements[i] = elements[i - 1];
    }
    elements[pos] = element;
    size++;
}

void append(E element) throw(runtime_error) {
    if (size == max) {
        throw runtime_error("List full");
    }
    elements[size] = element;
    size++;
}
```

Método para la inserción de un elemento. El tipo del elemento corresponde con el template al inicio de la clase.

```
void insert(E element) throw(runtime_error) {  
    if (size == max) {  
        throw runtime_error("List full");  
    }  
    for (int i = size; i > pos; i--) {  
        elements[i] = elements[i - 1];  
    }  
    elements[pos] = element;  
    size++;  
}  
void append(E element) throw(runtime_error) {  
    if (size == max) {  
        throw runtime_error("List full");  
    }  
    elements[size] = element;  
    size++;  
}
```

Indica que el método puede lanzar errores de tipo runtime_error.

```
void insert(E element) throw(runtime_error) {  
    if (size == max) {  
        throw runtime_error("List full");  
    }  
    for (int i = size; i > pos; i--) {  
        elements[i] = elements[i - 1];  
    }  
    elements[pos] = element;  
    size++;  
}  
void append(E element) throw(runtime_error) {  
    if (size == max) {  
        throw runtime_error("List full");  
    }  
    elements[size] = element;  
    size++;  
}
```

Chequeo de restricciones. El método puede fallar si ya la lista tiene la cantidad máxima de elementos.


```
void insert(E element) throw(runtime_error) {  
    if (size == max) {  
        throw runtime_error("List full");  
    }  
    for (int i = size; i > pos; i--) {  
        elements[i] = elements[i - 1];  
    }  
    elements[pos] = element;  
    size++;  
}  
void append(E element) throw(runtime_error) {  
    if (size == max) {  
        throw runtime_error("List full");  
    }  
    elements[size] = element;  
    size++;  
}
```

Lanzamiento de excepción. Se especifica un mensaje que explica porqué se lanza el error.

```
void insert(E element) throw(runtime_error) {
    if (size == max) {
        throw runtime_error("List full");
    }
    for (int i = size; i > pos; i--) {
        elements[i] = elements[i - 1];
    }
    elements[pos] = element;
    size++;
}

void append(E element) throw(runtime_error) {
    if (size == max) {
        throw runtime_error("List full");
    }
    elements[size] = element;
    size++;
}
```

Ciclo que mueve los elementos para crear un espacio donde se asignará el nuevo elemento.

```
void insert(E element) throw(runtime_error) {
    if (size == max) {
        throw runtime_error("List full");
    }
    for (int i = size; i > pos; i--) {
        elements[i] = elements[i - 1];
    }
    elements[pos] = element;
    size++;
}

void append(E element) throw(runtime_error) {
    if (size == max) {
        throw runtime_error("List full");
    }
    elements[size] = element;
    size++;
}
```

Se asigna el elemento y aumenta el tamaño de la lista.

```
void insert(E element) throw(runtime_error) {
    if (size == max) {
        throw runtime_error("List full");
    }
    for (int i = size; i > pos; i--) {
        elements[i] = elements[i - 1];
    }
    elements[pos] = element;
    size++;
}

void append(E element) throw(runtime_error) {
    if (size == max) {
        throw runtime_error("List full");
    }
    elements[size] = element;
    size++;
}
```

Método que agrega un elemento nuevo al final de la lista.

```
void insert(E element) throw(runtime_error) {
    if (size == max) {
        throw runtime_error("List full");
    }
    for (int i = size; i > pos; i--) {
        elements[i] = elements[i - 1];
    }
    elements[pos] = element;
    size++;
}

void append(E element) throw(runtime_error) {
    if (size == max) {
        throw runtime_error("List full");
    }
    elements[size] = element;
    size++;
}
```

El atributo size indica la posición del primer espacio libre en el arreglo.


```
E remove() throw(runtime_error) {  
    if (size == 0) {  
        throw runtime_error("Empty list");  
    }  
    if ((pos < 0) || (pos >= size)) {  
        throw runtime_error("Index out of bounds");  
    }  
    E res = elements[pos];  
    for (int i = pos; i < size - 1; i++) {  
        elements[i] = elements[i + 1];  
    }  
    size--;  
    return res;  
}
```

El método remove puede fallar si la lista se encuentra vacía o si la posición actual no es válida, como por ejemplo estar al final de la lista. El chequeo de si la posición es mayor que 0 puede resultar redundante, pero en todo caso es una condición que debiera darse nunca.

```
E remove() throw(runtime_error) {  
    if (size == 0) {  
        throw runtime_error("Empty list");  
    }  
    if ((pos < 0) || (pos >= size)) {  
        throw runtime_error("Index out of bounds");  
    }  
    E res = elements[pos];  
    for (int i = pos; i < size - 1; i++) {  
        elements[i] = elements[i + 1];  
    }  
    size--;  
    return res;  
}
```

Se almacena en una variable temporal el elemento que se desea eliminar.

```
E remove() throw(runtime_error) {  
    if (size == 0) {  
        throw runtime_error("Empty list");  
    }  
    if ((pos < 0) || (pos >= size)) {  
        throw runtime_error("Index out of bounds");  
    }  
    E res = elements[pos];  
    for (int i = pos; i < size - 1; i++) {  
        elements[i] = elements[i + 1];  
    }  
    size--;  
    return res;  
}
```



Este ciclo hace el trabajo inverso que se hace en el insert, es necesario mover todos los elementos desde la posición actual hasta el final un espacio para cerrar el espacio dejado por el elemento eliminado.


```
E remove() throw(runtime_error) {  
    if (size == 0) {  
        throw runtime_error("Empty list");  
    }  
    if ((pos < 0) || (pos >= size)) {  
        throw runtime_error("Index out of bounds");  
    }  
    E res = elements[pos];  
    for (int i = pos; i < size - 1; i++) {  
        elements[i] = elements[i + 1];  
    }  
    size--;  
    return res;  
}
```

Actualizar el tamaño de la lista y retornar el elemento eliminado.

```
void clear() {  
    size = 0;  
    pos = 0;  
    delete [] elements;  
    elements = new E[max];  
}  
E getElement() throw(runtime_error) {  
    if ((pos < 0) || (pos >= size)) {  
        throw runtime_error("Index out of bounds");  
    }  
    return elements[pos];  
}
```

Este método podría implementarse con borrado lógico, es decir simplemente poner size y pos en 0. Sin embargo, si los elementos del arreglo son una clase que necesite liberar recursos y tenga definido un destructor, el acto de borrar el arreglo de elementos implica invocar ese destructor.

```
void clear() {  
    size = 0;  
    pos = 0;  
    delete [] elements;  
    elements = new E[max];  
}  
E getElement() throw(runtime_error) {  
    if ((pos < 0) || (pos >= size)) {  
        throw runtime_error("Index out of bounds");  
    }  
    return elements[pos];  
}
```

El método puede fallar si la posición actual no contiene un elemento, es decir, si es igual a size. Se hace un chequeo de la posición similar al del método getElement.


```
void clear() {  
    size = 0;  
    pos = 0;  
    delete [] elements;  
    elements = new E[max];  
}  
E getElement() throw(runtime_error) {  
    if ((pos < 0) || (pos >= size)) {  
        throw runtime_error("Index out of bounds");  
    }  
    return elements[pos];  
}
```

Se retorna el elemento en la posición actual.

```
void goToStart() {  
    pos = 0;  
}  
void goToEnd() {  
    pos = size;  
}  
void goToPos(int newPos) throw(runtime_error) {  
    if ((newPos < 0) || (newPos > size)) {  
        throw runtime_error("Index out of bounds");  
    }  
    pos = newPos;  
}
```

Los métodos goToStart y goToEnd consisten en asignar a la posición actual cero y el tamaño de la lista, respectivamente.

```
void goToStart() {  
    pos = 0;  
}  
void goToEnd() {  
    pos = size;  
}  
void goToPos(int newPos) throw(runtime_error) {  
    if ((newPos < 0) || (newPos > size)) {  
        throw runtime_error("Index out of bounds");  
    }  
    pos = newPos;  
}
```



El método puede fallar si la posición actual es negativa o es mayor que el tamaño de la lista. Se actualiza el valor de la posición actual.

```
void next() {  
    if (pos < size) {  
        pos++;  
    }  
}  
bool atEnd() {  
    return pos == size;  
}  
void previous() {  
    if (pos > 0) {  
        pos--;  
    }  
}  
bool atStart() {  
    return pos == 0;  
}
```

Aumenta la posición en 1 si la posición es menor que el tamaño de la lista.

```
void next() {  
    if (pos < size) {  
        pos++;  
    }  
}  
bool atEnd() {  
    return pos == size;  
}  
void previous() {  
    if (pos > 0) {  
        pos--;  
    }  
}  
bool atStart() {  
    return pos == 0;  
}
```

Los métodos atStart y atEnd simplemente evalúan si la posición está en 0 o en el tamaño de la lista.


```
void next() {  
    if (pos < size) {  
        pos++;  
    }  
}  
bool atEnd() {  
    return pos == size;  
}  
void previous() {  
    if (pos > 0) {  
        pos--;  
    }  
}  
bool atStart() {  
    return pos == 0;  
}
```

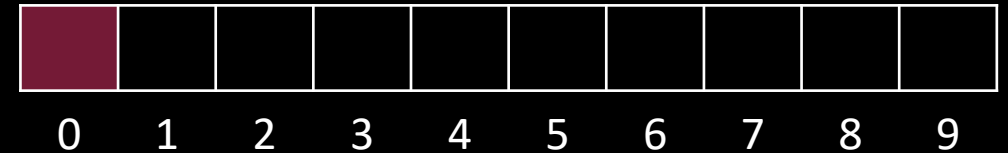
Restar 1 a la posición actual si es mayor que cero.

```
int getPos() {  
    return pos;  
}  
int getSize() {  
    return size;  
}  
};
```

Los métodos getPos y getSize simplemente retornan el valor de los atributos de posición y tamaño actual.

Ejemplo de utilización

```
ArrayList<int> L1(10);  
L1.append(5);  
L1.insert(7);  
L1.goToEnd();  
L1.insert(8);  
L1.previous();  
int a = L1.remove();  
L1.insert(10);
```



Este objeto se crea en memoria estática. Se invoca al constructor con el valor 10 como parámetro.

```
ArrayList<int> L1(10);
```

```
L1.append(5);
```

```
L1.insert(7);
```

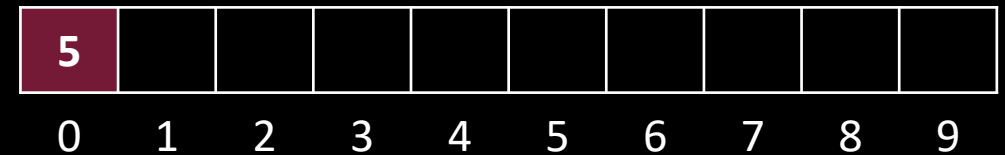
```
L1.goToEnd();
```

```
L1.insert(8);
```

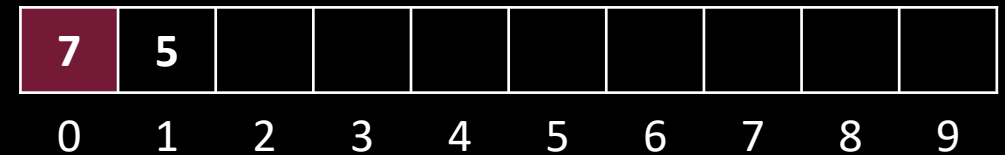
```
L1.previous();
```

```
int a = L1.remove();
```

```
L1.insert(10);
```

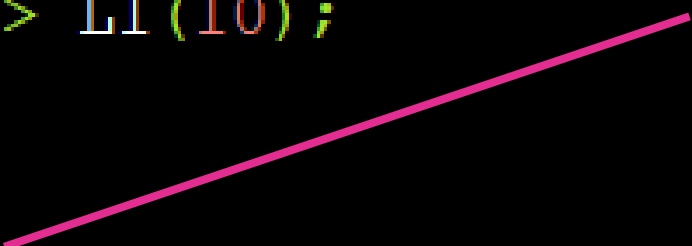


```
ArrayList<int> L1(10);  
L1.append(5);  
L1.insert(7);  
L1.goToEnd();  
L1.insert(8);  
L1.previous();  
int a = L1.remove();  
L1.insert(10);
```



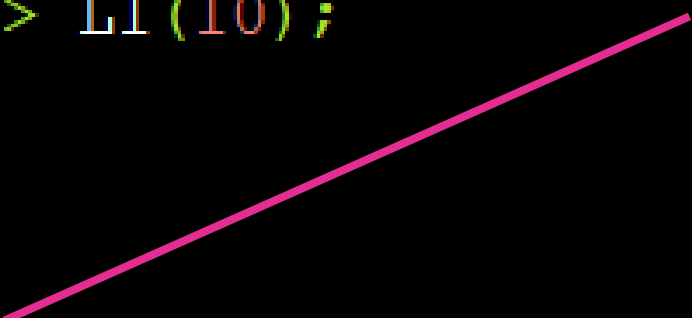
7	5								
0	1	2	3	4	5	6	7	8	9

```
ArrayList<int> L1(10);  
L1.append(5);  
L1.insert(7);  
L1.goToEnd();  
L1.insert(8);  
L1.previous();  
int a = L1.remove();  
L1.insert(10);
```



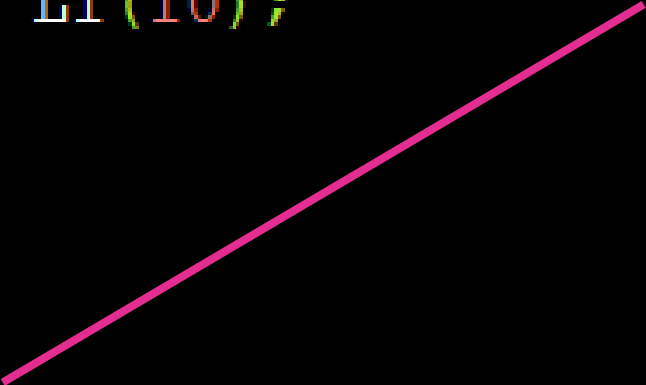
7	5								
0	1	2	3	4	5	6	7	8	9

```
ArrayList<int> L1(10);  
L1.append(5);  
L1.insert(7);  
L1.goToEnd();  
L1.insert(8);  
L1.previous();  
int a = L1.remove();  
L1.insert(10);
```



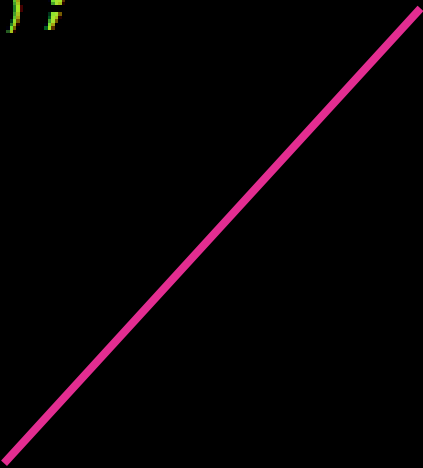
7	5	8							
0	1	2	3	4	5	6	7	8	9

```
ArrayList<int> L1(10);  
L1.append(5);  
L1.insert(7);  
L1.goToEnd();  
L1.insert(8);  
L1.previous();  
int a = L1.remove();  
L1.insert(10);
```



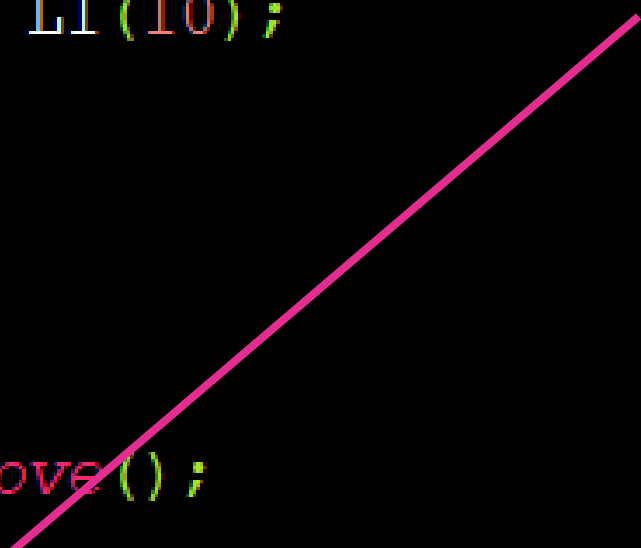
7	5	8							
0	1	2	3	4	5	6	7	8	9


```
ArrayList<int> L1(10);  
L1.append(5);  
L1.insert(7);  
L1.goToEnd();  
L1.insert(8);  
L1.previous();  
int a = L1.remove();  
L1.insert(10);
```



7	8								
0	1	2	3	4	5	6	7	8	9

```
ArrayList<int> L1(10);  
L1.append(5);  
L1.insert(7);  
L1.goToEnd();  
L1.insert(8);  
L1.previous();  
int a = L1.remove();  
L1.insert(10);
```



7	10	8							
0	1	2	3	4	5	6	7	8	9

Ejemplo de utilización

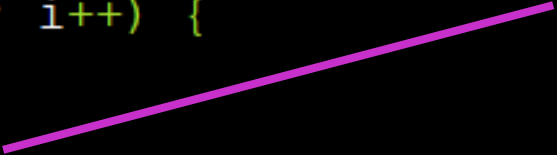
```
List<int> *L = new ArrayList<int>();
for (int i = 0; i < 100; i++) {
    L->append(i);
}
for (L->goToStart(); !L->atEnd(); L->next()) {
    cout << L->getElement() << endl;
}
cout << (find(L, 40)? "Encontrado." : "No encontrado.") << endl;
delete L;
```

L es un puntero a un objeto tipo List. Se le asigna la creación de un objeto tipo ArrayList. La asignación es permitida porque ArrayList es una subclase de List.

```
List<int> *L = new ArrayList<int>();  
for (int i = 0; i < 100; i++) {  
    L->append(i);  
}  
for (L->goToStart(); !L->atEnd(); L->next()) {  
    cout << L->getElement() << endl;  
}  
cout << (find(L, 40)? "Encontrado." : "No encontrado.") << endl;  
delete L;
```

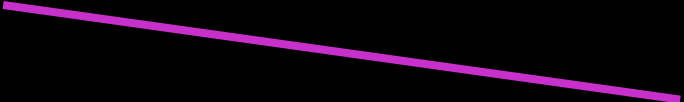
Ciclo que agrega los números desde 0 hasta 99 en la lista.

```
List<int> *L = new ArrayList<int>();  
for (int i = 0; i < 100; i++) {  
    L->append(i);  
}  
for (L->goToStart(); !L->atEnd(); L->next()) {  
    cout << L->getElement() << endl;  
}  
cout << (find(L, 40)? "Encontrado." : "No encontrado.") << endl;  
delete L;
```



Ciclo que recorre todos los elementos de la lista y los imprime en pantalla.

```
List<int> *L = new ArrayList<int>();  
for (int i = 0; i < 100; i++) {  
    L->append(i);  
}  
for (L->goToStart(); !L->atEnd(); L->next()) {  
    cout << L->getElement() << endl;  
}  
cout << (find(L, 40)? "Encontrado." : "No encontrado.") << endl;  
delete L;
```



Utilización de la función find mostrada anteriormente.

```
List<int> *L = new ArrayList<int>();  
for (int i = 0; i < 100; i++) {  
    L->append(i);  
}  
for (L->goToStart(); !L->atEnd(); L->next()) {  
    cout << L->getElement() << endl;  
}  
cout << (find(L, 40)? "Encontrado." : "No encontrado.") << endl;  
delete L;
```

El objeto lista fue creado en memoria dinámica, por lo tanto debe liberarse la memoria reservada cuando el objeto ya no va a ser utilizado.

Ejercicios con la clase ArrayList

- 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.
 - Puede fallar si la cantidad de elementos total es mayor que `maxSize`.
- Agregar el método **reverse**.
 - No recibe ningún parámetro.
 - Invierte el orden de los elementos de la lista.
 - No tiene ninguna restricción.
- Modificar el método **insert** y **append** para que sea **dinámico**.
 - Si la lista llega al tamaño máximo, sustituir el arreglo por uno más grande (puede ser el doble).
 - Traspasar los elementos al nuevo arreglo y liberar el viejo.
- 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.

Lista ordenada

- Lista que mantiene los elementos **ordenados** de menor a mayor
- Esto se puede lograr si los elementos son comparables y permiten la utilización de **operadores** `==`, `!=`, `<`, `<=`, `>`, `>=`
- La **posición actual** no importa cuando se utiliza `insert` o `append`
- El método `indexOf` puede implementarse de forma eficiente con **búsqueda binaria**
- Si se crea una clase que herede de `ArrayList`, se pueden **rescribir** los métodos que son diferentes
- Es necesario que los miembros privados de `ArrayList` sean declarados como **protected** para que sean accesibles

```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```

La clase hereda de ArrayList, de esta forma va a poseer los mismos miembros e implementación que tiene la clase base.

```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```

La clase ArrayList fue alterada para que los miembros marcados como private, sean protected. Esto permite que las subclases tengan acceso a estos miembros.

```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```

La clase no tiene ningún atributo propio, solamente los heredados de ArrayList.

```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```

```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```

Indica que debe ejecutarse el constructor de la clase base, donde se inicializan los atributos de ArrayList. Aparte de eso el constructor no inicializa nada más porque no tiene nuevos atributos.

```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```

La clase no tiene destructor propio porque no tiene ningún recurso que retornar. No es necesario invocar el destructor de la clase base, eso se efectúa automáticamente.

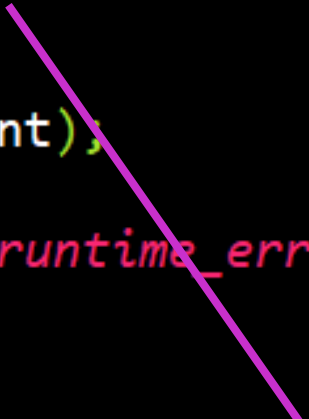

```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```

El método insert puede fallar si falla el método de la clase base.


```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```

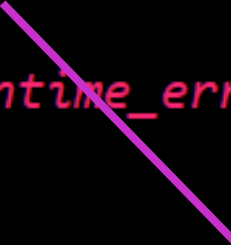
De esta forma se invoca al método goToStart
declarado en la clase base.

```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```



Ciclo que busca de forma secuencial la posición donde debe insertarse el elemento. Esto podría mejorarse utilizando búsqueda binaria

```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```



Se inserta el elemento en la posición encontrada.

```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```

Dado que la lista es ordenada, el método append debe hacer lo mismo que hace el insert, por lo que se invoca.

```
template <typename E>
class SortedArrayList : public ArrayList<E> {
public:
    SortedArrayList(int maxSize = DEFAULT_MAX_SIZE) : ArrayList<E>(maxSize) {}
    void insert(E element) throw (runtime_error) {
        ArrayList<E>::goToStart();
        while (!ArrayList<E>::atEnd() && element >= ArrayList<E>::getElement()) {
            ArrayList<E>::next();
        }
        ArrayList<E>::insert(element);
    }
    void append(E element) throw (runtime_error) {
        insert(element);
    }
    bool contains(E element) {
        return indexOf(element) != -1;
    }
}
```

El método contains se implementa invocando al método indexOf, si su resultado es -1, entonces el elemento no se encuentra en la lista, de otro modo, sí se encuentra en la lista.

```
int indexOf(E element) {
    int min = 0;
    int max = ArrayList<E>::size - 1;
    while (min <= max) {
        int mid = (min + max) / 2;
        if (element == ArrayList<E>::elements[mid]) {
            while (ArrayList<E>::elements[mid - 1] == ArrayList<E>::elements[mid]) {
                mid--;
            }
            return mid;
        } else if (element < ArrayList<E>::elements[mid]) {
            max = mid - 1;
        } else {
            min = mid + 1;
        }
    }
    return -1;
}

};
```

```
int indexOf(E element) {  
    int min = 0;  
    int max = ArrayList<E>::size - 1;  
    while (min <= max) {  
        int mid = (min + max) / 2;  
        if (element == ArrayList<E>::elements[mid]) {  
            while (ArrayList<E>::elements[mid - 1] == ArrayList<E>::elements[mid]) {  
                mid--;  
            }  
            return mid;  
        } else if (element < ArrayList<E>::elements[mid]) {  
            max = mid - 1;  
        } else {  
            min = mid + 1;  
        }  
    }  
    return -1;  
}  
};
```

Este método implementa la búsqueda binaria para determinar la posición de un elemento dentro de la lista.

```
int indexOf(E element) {  
    int min = 0;  
    int max = ArrayList<E>::size - 1;  
    while (min <= max) {  
        int mid = (min + max) / 2;  
        if (element == ArrayList<E>::elements[mid]) {  
            while (ArrayList<E>::elements[mid - 1] == ArrayList<E>::elements[mid]) {  
                mid--;  
            }  
            return mid;  
        } else if (element < ArrayList<E>::elements[mid]) {  
            max = mid - 1;  
        } else {  
            min = mid + 1;  
        }  
    }  
    return -1;  
}  
};
```

Variables que marcan las posiciones del menor y mayor elemento del rango donde se realiza la búsqueda.


```
int indexOf(E element) {  
    int min = 0;  
    int max = ArrayList<E>::size - 1;  
    while (min <= max) {  
        int mid = (min + max) / 2;  
        if (element == ArrayList<E>::elements[mid]) {  
            while (ArrayList<E>::elements[mid - 1] == ArrayList<E>::elements[mid]) {  
                mid--;  
            }  
            return mid;  
        } else if (element < ArrayList<E>::elements[mid]) {  
            max = mid - 1;  
        } else {  
            min = mid + 1;  
        }  
    }  
    return -1;  
}  
};
```

El ciclo se repite mientras los índices que marcan el menor y mayor del rango de búsqueda no se entrecrucen.

```
int indexOf(E element) {
    int min = 0;
    int max = ArrayList<E>::size - 1;
    while (min <= max) {
        int mid = (min + max) / 2;
        if (element == ArrayList<E>::elements[mid]) {
            while (ArrayList<E>::elements[mid - 1] == ArrayList<E>::elements[mid]) {
                mid--;
            }
            return mid;
        } else if (element < ArrayList<E>::elements[mid]) {
            max = mid - 1;
        } else {
            min = mid + 1;
        }
    }
    return -1;
}

};
```

Se calcula la posición del elemento en el medio y se compara con el buscado.

```
int indexOf(E element) {
    int min = 0;
    int max = ArrayList<E>::size - 1;
    while (min <= max) {
        int mid = (min + max) / 2;
        if (element == ArrayList<E>::elements[mid]) {
            while (ArrayList<E>::elements[mid - 1] == ArrayList<E>::elements[mid]) {
                mid--;
            }
            return mid;
        } else if (element < ArrayList<E>::elements[mid]) {
            max = mid - 1;
        } else {
            min = mid + 1;
        }
    }
    return -1;
}

};
```

Si el elemento en la posición media es igual al buscado, se entra en un ciclo que busca hacia el lado de los menores la primera ocurrencia del elemento. Esto se hace por si hay elementos repetidos y se desea retornar la primera ocurrencia del elemento buscado en la lista.

```
int indexOf(E element) {  
    int min = 0;  
    int max = ArrayList<E>::size - 1;  
    while (min <= max) {  
        int mid = (min + max) / 2;  
        if (element == ArrayList<E>::elements[mid]) {  
            while (ArrayList<E>::elements[mid - 1] == ArrayList<E>::elements[mid]) {  
                mid--;  
            }  
            return mid;  
        } else if (element < ArrayList<E>::elements[mid]) {  
            max = mid - 1;  
        } else {  
            min = mid + 1;  
        }  
    }  
    return -1;  
}  
};
```

Si el elemento buscado no se encuentra en la posición del medio, se determina en qué mitad se encuentra y se actualizan los índices del rango de búsqueda para repetir el ciclo.

```
int indexOf(E element) {  
    int min = 0;  
    int max = ArrayList<E>::size - 1;  
    while (min <= max) {  
        int mid = (min + max) / 2;  
        if (element == ArrayList<E>::elements[mid]) {  
            while (ArrayList<E>::elements[mid - 1] == ArrayList<E>::elements[mid]) {  
                mid--;  
            }  
            return mid;  
        } else if (element < ArrayList<E>::elements[mid]) {  
            max = mid - 1;  
        } else {  
            min = mid + 1;  
        }  
    }  
    return -1;  
};
```

Si se llega a esta línea, es porque los índices se entrecruzaron y esto significa que el elemento no se encuentra dentro del arreglo, por lo que se retorna -1.

Ejemplo de utilización

```
SortedArrayList<int> L;
srand(time(0));
cout << "SortedArrayList: Insertando números aleatorios." << endl;
for (int i = 0; i < 20; i++) {
    L.insert(rand());
}
L.insert(9999);
L.insert(9999);
L.goToStart();
cout << "SortedArrayList: Imprimiendo valores." << endl;
while (!L.atEnd()) {
    cout << L.getElement() << endl;
    L.next();
}
cout << "SortedArrayList::indexOf(9999): " << L.indexOf(9999) << endl;
cout << "SortedArrayList::contains(9999): " << L.contains(9999) << endl;
cout << "SortedArrayList::contains(9998): " << L.contains(9998) << endl;
```

SortedArrayList: Insertando números aleatorios.

SortedArrayList: Imprimiendo valores.

1045

4673

5661

7430

7486

7528

9353

9999

9999

10601

11050

13325

13774

13883

16154

20801

29016

29055

29472

29565

29645

30439

SortedArrayList::indexOf(9999): 7

SortedArrayList::contains(9999): 1

SortedArrayList::contains(9998): 0

Ejemplo

```
SortedArrayList L;  
srand(time(0));  
cout << "Sorted ArrayList:  
for (int i = 0; i < 20; i++)  
    L.insert(rand() % 30000);  
}  
L.insert(9999);  
L.insert(9999);  
L.goToStart();  
cout << "Sorted ArrayList:  
while (!L.atEnd())  
    cout << L.get() << " ";  
L.next();  
}
```

Ejercicios con SortedArrayList

- Hacer que el método de inserción utilice búsqueda binaria en lugar de secuencial para determinar el lugar donde debe insertarse el elemento.
- Implemente la clase SortedArrayList sin utilizar herencia, usando un ArrayList como atributo de la clase. Compare y contraste ambas implementaciones. Establezca ventajas y desventajas de cada una.



Listas con Arreglos

Mauricio Avilés