Programación genérica
Plantillas

Estructuras de Datos

Andrea Rueda

Pontificia Universidad Javeriana
Departamento de Ingeniería de Sistemas

Estructuras lineales

¿Cómo implementarlas?

- Necesidad:

  - estructura totalmente dinámica, que ajuste su tamaño en tiempo de ejecución

  - estructura flexible, que acepte diferentes tipos de datos (no mezclados)

Programación genérica
Plantillas

Programación genérica

```
a = b + c;
```

En C++, ¿qué problemas tiene la línea anterior?

Programación genérica

```
int b = 5;
int c = 6;
int a = b + c;
```

```
int suma(    int a,    int b )
{ return( a + b ); }
```

Programación genérica

```
float b = 5.67;
float c = 6.923;
float a = b + c;
```

```
float suma( float a, float b )
{ return( a + b ); }
```

Programación genérica

- Uso de plantillas:
  Generalización → adaptabilidad, flexibilidad.

```
template< class identificador >
declaracion_funcion;

template< typename identificador >
declaracion_funcion;
```

Programación genérica

- Plantilla con un tipo de dato:

```
template< class T >
T suma( T a, T b )
{ return( a + b ); }

int a = suma<int>(5, 7);
double b = suma<double>(6.4, 1.7);
```

Programación genérica

- Plantilla con dos tipos diferentes de dato

```
template< class T, class U >
T suma( T a, U b )
{ return( a + b ); }

int i, j = 25;
long l = 4567;
i = suma<int,long> (j,l);
```

Programación genérica

- Plantilla para clases

```
template< class T >
class vec_par {
  T valores [2];
  public:
    vec_par (T uno, T dos)
    { valores[0] = uno;
      valores[1] = dos; }
    T minimo ();
};
```

Programación genérica

- Plantilla para clases

```
template< class T >
T vec_par<T>::minimo () {
  T resultado;
  if (valores[0]<valores[1])
    resultado = valores[0];
  else
    resultado = valores[1];
  return resultado;
}
```

Programación genérica

- Plantilla para clases

```
vec_par<int> obj_i (115,36);
int res;
res = obj_i.minimo();

vec_par<float> obj_f (32.56,76.98);
float res2;
res2 = obj_f.minimo();
```

Programación genérica

- Ejercicio: implementar el siguiente TAD

  TAD Operación Binaria

  Conjunto mínimo de datos:
  - Operando1, entero, real o carácter; primer elemento de la operación
  - Operando2, entero, real o carácter; segundo elemento de la operación
  - Operación, caracter, operación a aplicar, una entre: '+', '-', '*', '/'

  Comportamiento (operaciones):
  - EvaluarOperación (OperaciónBinaria): retorna el resultado de aplicar la operación sobre los operandos.

Programación genérica

- Ejercicio:

```
template <class N>
struct OpBinaria {
  N op1;
  N op2;
  char operacion;
  N EvaluarOperacion();
};
```

https://repl.it/@Andrea_del_Pila/PC-plantillas-ejercicio

Programación genérica

- Ejercicio:

```
template <class N>
N OpBinaria<N>::EvaluarOperacion() {
  N resul;
  switch( operacion ) {
    case '+': resul = op1 + op2; break;
    case '-': resul = op1 – op2; break;
    case '*': resul = op1 * op2; break;
    case '/': resul = op1 / op2; break;
  }
  return resul;
}
```

https://repl.it/@Andrea_del_Pila/PC-plantillas-ejercicio

Programación genérica

Organización de las librerías con plantillas:

- Encabezado (.h)
- Implementación (.hxx, ¿por qué no .cxx?)
- ESTOS DOS ARCHIVOS ***NO SE COMPILAN***.
  - Se usan en un archivo compilable (.cxx, .cpp) donde se ***INSTANCIAN*** los elementos genéricos.

Programación genérica

Organización de las librerías con plantillas:

- Encabezado o cabecera (.h):
  - Incluye al final el archivo de implementación (.hxx).
- Archivo de código (implementación) (.hxx):
  - Incluye al principio la cabecera (.h).

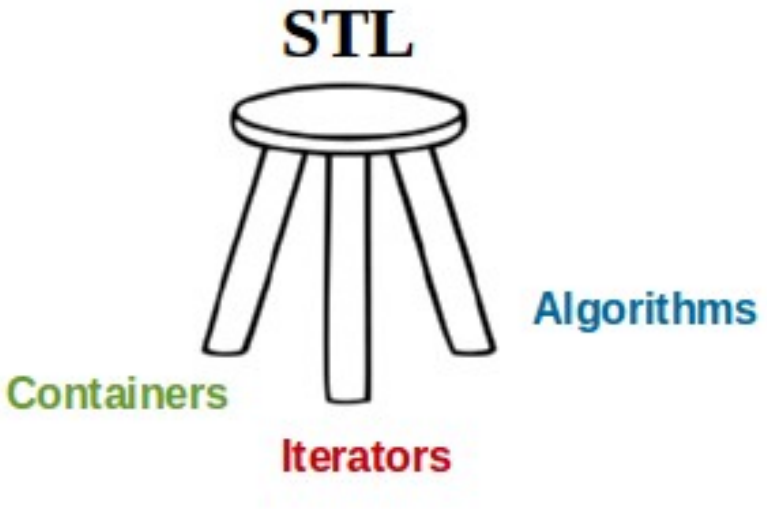
Para usar la librería con plantilla:

- Incluir la cabecera (.h) en un archivo con procedimiento principal (.cpp, .cxx).

STL
Standard Template Library

STL (Standard Template Library)

- ¡Librería con “muchas cosas” genéricas!
- Provee un conjunto de clases comunes, usables con cualquier tipo de dato y con operaciones elementales.
- Tres componentes:
  - Contenedores (*containers*).
  - Algoritmos (*algorithms*).
  - Iteradores (*iterators*).



www.bogotobogo.com/
cplusplus/stl_vector_list.php

http://www.sgi.com/tech/stl

STL (Standard Template Library)

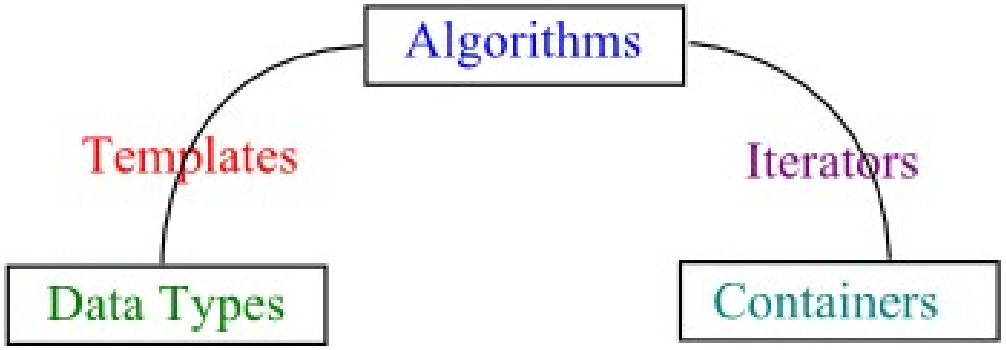
Componentes:

- Contenedores: clases predefinidas para almacenamiento de datos.
- Algoritmos: operaciones básicas como búsqueda y ordenamiento.
- Iteradores: permiten recorrer los datos en los contenedores (similar a apuntadores).

http://www.sgi.com/tech/stl

STL (Standard Template Library)

¿Cómo se conectan estos conceptos?



1. Templates
   make algorithms independent of the data types
2. Iterators
   make algorithms independent of the containters

www.bogotobogo.com/cplusplus/stl3_iterators.php

http://www.sgi.com/tech/stl

Contenedores STL

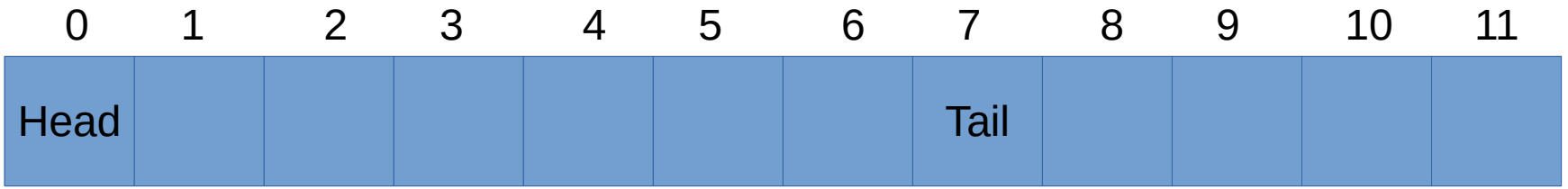
- Contenedores secuenciales estándar:
  - De acceso aleatorio:

    - **`vector`**: Arreglos dinámicos.
    (`std::vector` `#include <vector>`)

    - **`deque`**: Cola de doble cabeza.
    (`std::deque` `#include <deque>`)
  - De acceso iterativo:

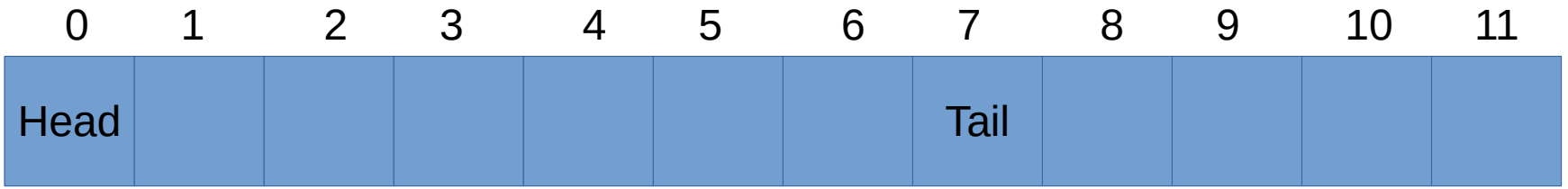
    - **`list`**: Doblemente encadenada.
    (`std::list` `#include <list>`)

vector<T>



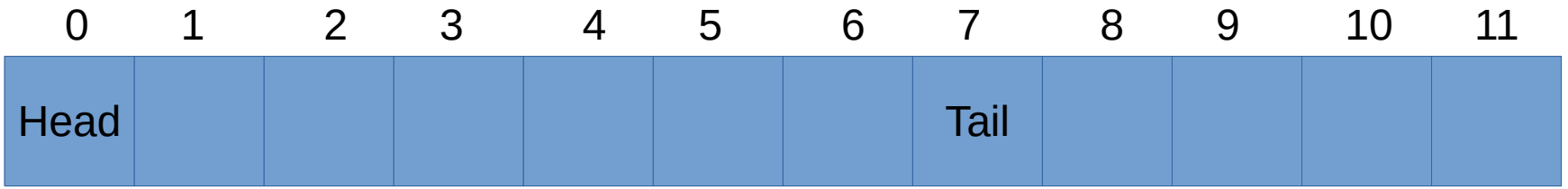
- Representa arreglos que cambian de tamaño (dinámicos)
- Posiciones contiguas de memoria
- Elementos de un mismo tipo de dato (plantilla)
- Crece en memoria por el final (cola, extremo derecho)

vector<T>



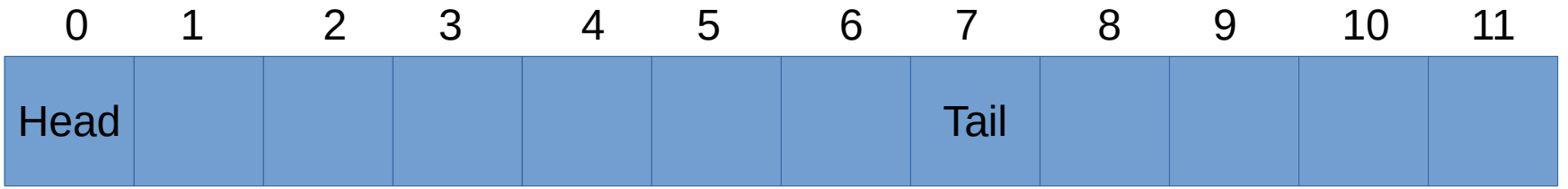
- Métodos soportados:
  - `size`: tamaño
  - `front`: elemento al frente
  - `clear`: vaciar vector
  - `pop_back`: eliminar en cola
  - `pop_front`: eliminar al frente
  - `insert`: insertar en posición
  - `empty`: vector está vacío?
  - `back`: elemento al final
  - `push_back`: insertar en cola
  - `push_front`: insertar al frente
  - `erase`: eliminar en posición

vector<T>



- ¿Orden de complejidad de los métodos?
  - `size, empty`
  - `front, back`
  - `clear`
  - `push_back, pop_back`
  - `push_front, pop_front`
  - `insert, erase`

vector<T>



- ¿Orden de complejidad de los métodos?
  - `size, empty` → **O(1)**
  - `front, back` → **O(1)**
  - `clear` → **O(1)**
  - `push_back, pop_back` → **O(1)**
  - `push_front, pop_front` → **O(n)**
  - `insert, erase` → **O(n)**

vector<T>

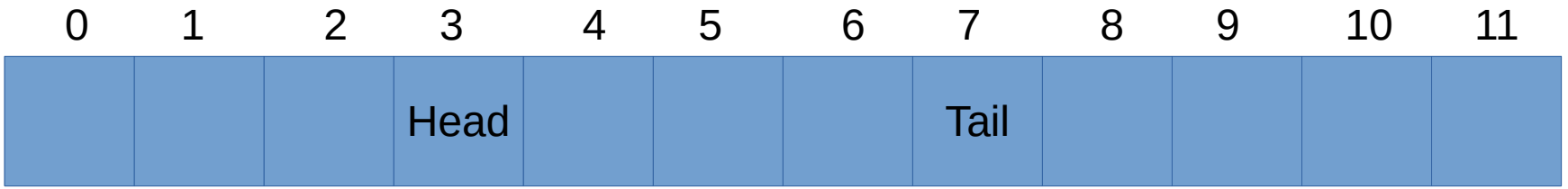
- Declaración
```
std::vector<int> miVec;
```

- Inserción de datos
```
miVec.push_back(1);
```

- Acceso a datos
```
for (int i = 0; i < miVec.size(); i++)
    std::cout << miVec[i] << std::endl;
```

deque<T>

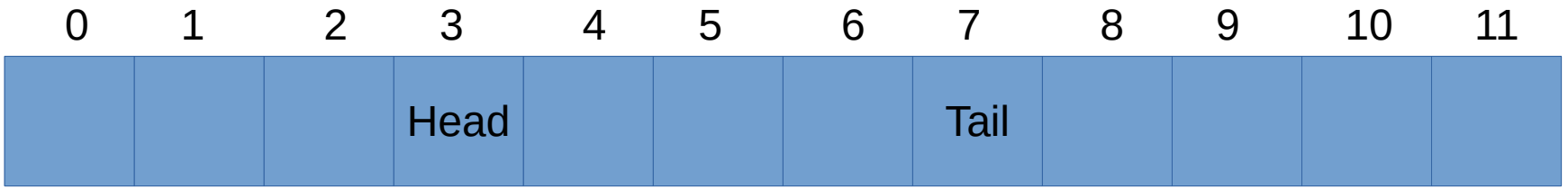
double-ended queue



- Representa arreglos que cambian de tamaño (dinámicos) en ambos extremos
- Posiciones contiguas de memoria (a trozos)
- Elementos de un mismo tipo de dato (plantilla)
- Crece en memoria por el principio y por el final (cabeza y cola, ambos extremos)

deque<T>

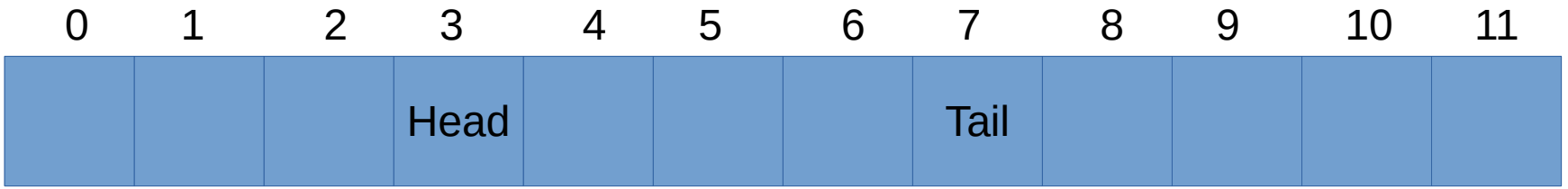
double-ended queue



- ¿Orden de complejidad de los métodos?
  - `size, empty`
  - `front, back`
  - `clear`
  - `push_back, pop_back`
  - `push_front, pop_front`
  - `insert, erase`

deque<T>

double-ended queue



- ¿Orden de complejidad de los métodos?
  - `size, empty` → **O(1)**
  - `front, back` → **O(1)**
  - `clear` → **O(1)**
  - `push_back, pop_back` → **O(1)**
  - `push_front, pop_front` → **O(1)**
  - `insert, erase` → **O(n)**

deque<T>

- Declaración

```
std::deque<int> miDeq;
```

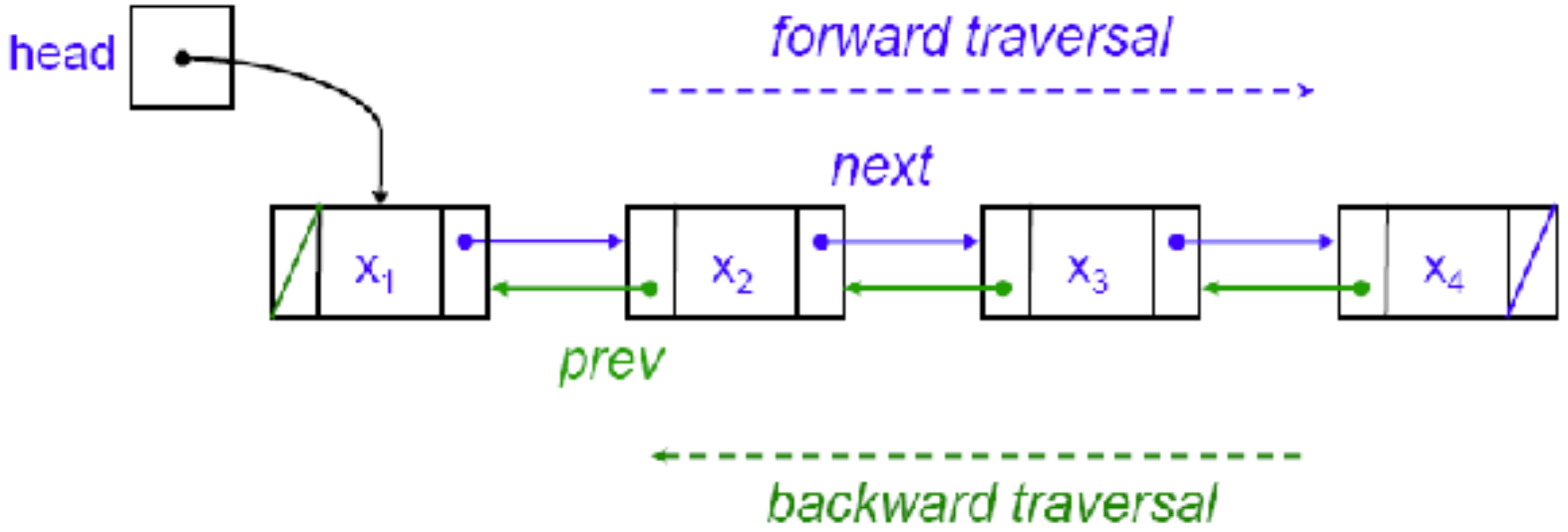
- Inserción de datos

```
miDeq.push_back(1);
miDeq.push_front(2);
```

- Acceso a datos

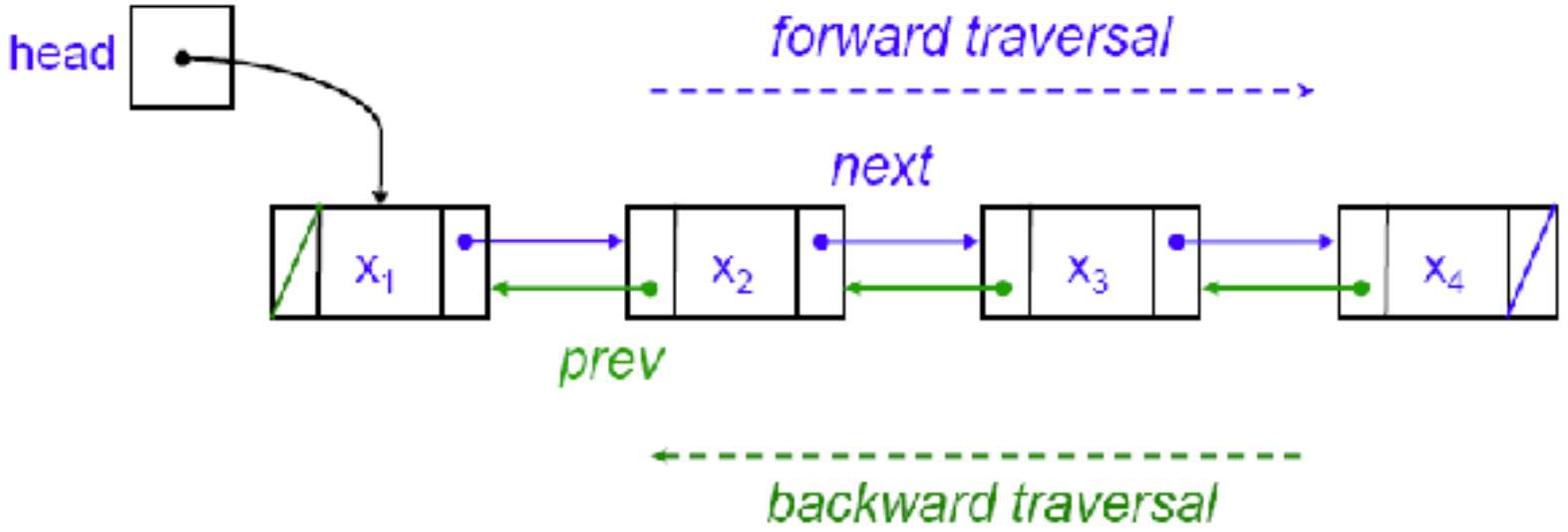
```
for (int i = 0; i < miDeq.size(); i++)
    std::cout << miDeq[i] << std::endl;
```

list<T>



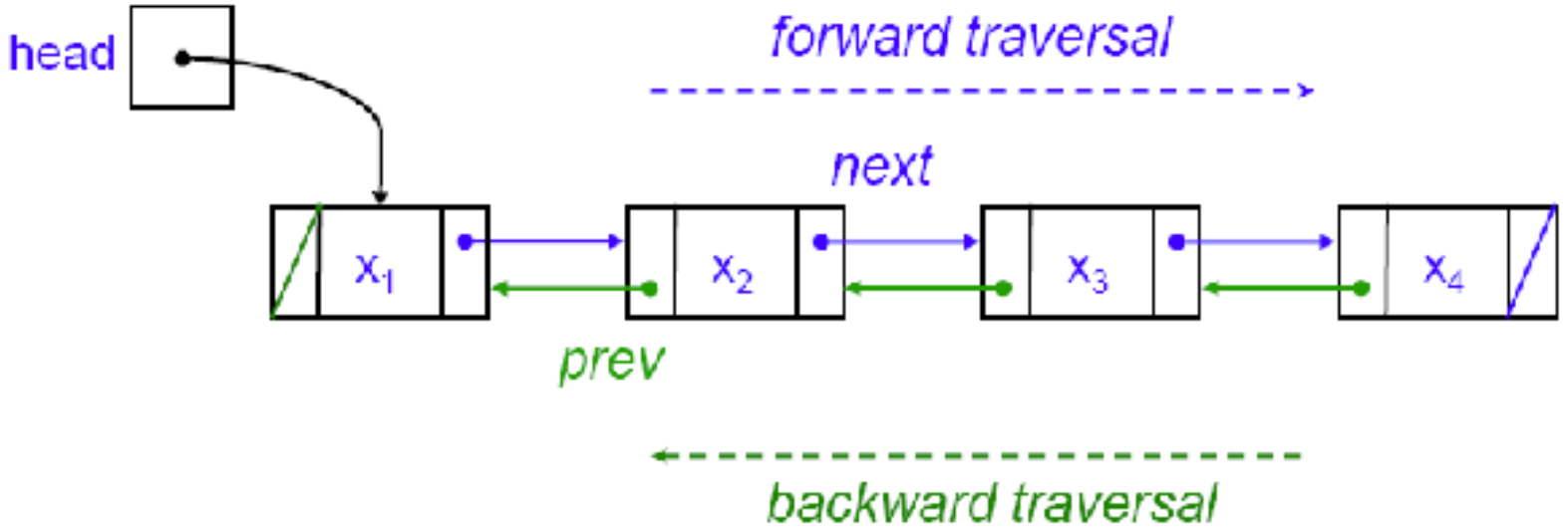
- Representa listas doblemente encadenadas (recorrido hacia adelante y hacia atrás)
- Posiciones separadas en memoria (cualquier parte)
- Elementos de un mismo tipo de dato (plantilla)
- Crece en memoria en cualquier punto de la secuencia

list<T>



- ¿Orden de complejidad de los métodos?
  - `size, empty`
  - `front, back`
  - `clear`
  - `push_back, pop_back`
  - `push_front, pop_front`
  - `insert, erase`

list<T>



- ¿Orden de complejidad de los métodos?
  - `size, empty` → **O(1)**
  - `front, back` → **O(1)**
  - `clear` → **O(n)**
  - `push_back, pop_back` → **O(1)**
  - `push_front, pop_front` → **O(1)**
  - `insert, erase` → **O(1)***

list<T>

- Declaración

```
std::list<int> miList;
```

- Inserción de datos

```
miList.push_back(1);
miList.push_front(2);
```

- Acceso a datos

```
std::list<int>::iterator miIt;
for (miIt = miList.begin();
     miIt != miList.end(); miIt++)
   std::cout << *miIt << std::endl;
```

Órdenes de complejidad

	vector	deque	list
size	O(1)	O(1)	O(1)
clear	O(1)	O(1)	O(n)
empty	O(1)	O(1)	O(1)
push_front	O(n)	O(1)	O(1)
pop_front	O(n)	O(1)	O(1)
push_back	O(1)	O(1)	O(1)
pop_back	O(1)	O(1)	O(1)
insert	O(n)	O(n)	O(1)
erase	O(n)	O(n)	O(1)
¿Acceso aleatorio?	si	si	no

Criterios para escoger

- Evolución de la secuencia:
  - Estática:
    - ¿Inserción solo por final? ¿O por final y cabeza?
  - Dinámica:
    - ¿Inserción solo por final? ¿O por final y cabeza? ¿O en cualquier punto?
- ¿Memoria limitada?
- Tipo de acceso:
  - ¿Necesita acceso aleatorio?
  - ¿Predominan las iteraciones sobre toda la secuencia?

Referencias

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms, 3rd edition. MIT Press, 2009.
- L. Joyanes Aguilar, I. Zahonero. Algoritmos y estructuras de datos: una perspectiva en C. McGraw-Hill, 2004.