Programación Avanzada y Estructuras de Datos

4. Tipos lineales

Víctor M. Sánchez Cartagena

Grado en Ingeniería en Inteligencia Artificial
Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

23 de octubre de 2024

Índice

TAD vector

Especificación del TAD vector

Definición: Colección de n elementos almacenados en cierto orden, tal que podemos identificar al primer elemento, segundo, tercero, etc. Podemos acceder a cada elemento usando como índice un entero en el rango $[0, n-1]$. El índice de un elemento e del vector es el número de elementos anteriores a e en el vector: el índice el primer elemento es 0, el del segundo es 1, ..., el del último es $n-1$.

TAD vector

Especificación del TAD vector

Operaciones:

- Obtiene el número de elementos almacenados en el vector

```
int size() const;
```

- Comprueba si el vector está vacío

```
bool empty() const;
```

- Devuelve el elemento en el índice *i*

```
Elem at(int i) const;
```

TAD vector

Especificación del TAD vector

Operaciones:

- Asigna el elemento al índice *i*. Devuelve `false` si *i* no está en el rango $[0, n-1]$ y `true` en caso contrario

```
bool set(int i, const Elem& e);
```

- Inserta el elemento *e* en el índice *i* y desplaza una posición hacia adelante todos los elementos que estaban las posiciones de la *i* en adelante. Devuelve `false` si el índice *i* no está en el rango $[0, n]$ y `true` en caso contrario

```
bool insert(int i, const Elem& e);
```

Especificación del TAD vector

Operaciones:

- Elimina el elemento en el índice *i* y desplaza una posición hacia atrás todos los elementos que estaban las posiciones de la *i* en adelante. Devuelve `false` si *i* no está en el rango $[0, n-1]$ y `true` en caso contrario

```
bool erase(int i);
```

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`		
`insert(0,4)`		
`at(1)`		
`insert(2,2)`		
`at(3)`		
`erase(1)`		
`insert(1,5)`		
`insert(1,3)`		
`insert(4,9)`		
`at(2)`		
`set(3,8)`		

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`	true	(7)
`insert(0,4)`		
`at(1)`		
`insert(2,2)`		
`at(3)`		
`erase(1)`		
`insert(1,5)`		
`insert(1,3)`		
`insert(4,9)`		
`at(2)`		
`set(3,8)`		

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`	true	(7)
`insert(0,4)`	true	(4,7)
`at(1)`		
`insert(2,2)`		
`at(3)`		
`erase(1)`		
`insert(1,5)`		
`insert(1,3)`		
`insert(4,9)`		
`at(2)`		
`set(3,8)`		

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`	true	(7)
`insert(0,4)`	true	(4,7)
`at(1)`	7	(4,7)
`insert(2,2)`		
`at(3)`		
`erase(1)`		
`insert(1,5)`		
`insert(1,3)`		
`insert(4,9)`		
`at(2)`		
`set(3,8)`		

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`	true	(7)
`insert(0,4)`	true	(4,7)
`at(1)`	7	(4,7)
`insert(2,2)`	true	(4,7,2)
`at(3)`		
`erase(1)`		
`insert(1,5)`		
`insert(1,3)`		
`insert(4,9)`		
`at(2)`		
`set(3,8)`		

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`	true	(7)
`insert(0,4)`	true	(4,7)
`at(1)`	7	(4,7)
`insert(2,2)`	true	(4,7,2)
`at(3)`	"error"	(4,7,2)
`erase(1)`		
`insert(1,5)`		
`insert(1,3)`		
`insert(4,9)`		
`at(2)`		
`set(3,8)`		

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`	true	(7)
`insert(0,4)`	true	(4,7)
`at(1)`	7	(4,7)
`insert(2,2)`	true	(4,7,2)
`at(3)`	"error"	(4,7,2)
`erase(1)`	true	(4,2)
`insert(1,5)`		
`insert(1,3)`		
`insert(4,9)`		
`at(2)`		
`set(3,8)`		

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`	true	(7)
`insert(0,4)`	true	(4,7)
`at(1)`	7	(4,7)
`insert(2,2)`	true	(4,7,2)
`at(3)`	"error"	(4,7,2)
`erase(1)`	true	(4,2)
`insert(1,5)`	true	(4,5,2)
`insert(1,3)`		
`insert(4,9)`		
`at(2)`		
`set(3,8)`		

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`	true	(7)
`insert(0,4)`	true	(4,7)
`at(1)`	7	(4,7)
`insert(2,2)`	true	(4,7,2)
`at(3)`	"error"	(4,7,2)
`erase(1)`	true	(4,2)
`insert(1,5)`	true	(4,5,2)
`insert(1,3)`	true	(4,3,5,2)
`insert(4,9)`		
`at(2)`		
`set(3,8)`		

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`	true	(7)
`insert(0,4)`	true	(4,7)
`at(1)`	7	(4,7)
`insert(2,2)`	true	(4,7,2)
`at(3)`	"error"	(4,7,2)
`erase(1)`	true	(4,2)
`insert(1,5)`	true	(4,5,2)
`insert(1,3)`	true	(4,3,5,2)
`insert(4,9)`	true	(4,3,5,2,9)
`at(2)`		
`set(3,8)`		

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`	true	(7)
`insert(0,4)`	true	(4,7)
`at(1)`	7	(4,7)
`insert(2,2)`	true	(4,7,2)
`at(3)`	"error"	(4,7,2)
`erase(1)`	true	(4,2)
`insert(1,5)`	true	(4,5,2)
`insert(1,3)`	true	(4,3,5,2)
`insert(4,9)`	true	(4,3,5,2,9)
`at(2)`	5	(4,3,5,2,9)
`set(3,8)`		

Especificación del TAD vector

Ejemplos (vector inicialmente vacío):

Operación	Salida	Contenido del vector
`insert(0,7)`	true	(7)
`insert(0,4)`	true	(4,7)
`at(1)`	7	(4,7)
`insert(2,2)`	true	(4,7,2)
`at(3)`	"error"	(4,7,2)
`erase(1)`	true	(4,2)
`insert(1,5)`	true	(4,5,2)
`insert(1,3)`	true	(4,3,5,2)
`insert(4,9)`	true	(4,3,5,2,9)
`at(2)`	5	(4,3,5,2,9)
`set(3,8)`	true	(4,3,5,8,9)

TAD vector

Pregunta

¿A qué tipo de datos en Python se asemeja el TAD vector? ¿Podrías establecer a qué operación de ese tipo en Python corresponde cada operación del TAD vector?

TAD vector

Pregunta

¿A qué tipo de datos en Python se asemeja el TAD vector? ¿Podrías establecer a qué operación de ese tipo en Python corresponde cada operación del TAD vector?

El tipo `list`

TAD vector

Pregunta

¿A qué tipo de datos en Python se asemeja el TAD vector? ¿Podrías establecer a qué operación de ese tipo en Python corresponde cada operación del TAD vector?

El tipo `list`

- `size()` →
- `empty()` →
- `at(int i)` →
- `set(int i, const Elem &)` →
- `insert(int i, const Elem &)` →
- `erase(int i)` →
- ????????? →

TAD vector

Pregunta

¿A qué tipo de datos en Python se asemeja el TAD vector? ¿Podrías establecer a qué operación de ese tipo en Python corresponde cada operación del TAD vector?

El tipo `list`

- `size()` → `len()`
- `empty()` → `len(lista) == 0`
- `at(int i)` → `lista[i]`
- `set(int i, const Elem &)` → `lista[i]=`
- `insert(int i, const Elem &)` → `lista.insert(...)`
- `erase(int i)` → `lista.pop( ...)`
- ????????? → `lista.append( ...)`

Implementación del TAD vector

Determinación de la representación:

Implementación del TAD vector

Determinación de la representación: array de C++

Implementación del TAD vector

Determinación de la representación: array de C++

`Vector.h`:

```
typedef int Elem;
const int MAX_ELEMS=100;

class Vector {
private:
Elem data[MAX_ELEMS]; //Contenido del array: siempre del mismo tamaño
int n; //Número de elementos
public:
Vector();
Vector(const Vector &);
~Vector();
Vector& operator=(const Vector &);
int size() const;
bool empty() const;
Elem at(int i) const;
bool set(int i, const Elem& e);
bool insert(int i, const Elem& e);
bool erase(int i);
};
```

Implementación del TAD vector

Implementación de las operaciones: forma canónica

Implementación del TAD vector

Implementación de las operaciones: forma canónica

`Vector.cc`:

```
Vector::Vector(){
  n=0; }

Vector::Vector(const Vector& v){
   n=v.n;
   for(int i=0; i < n; i++)
       data[i]=v.data[i]; }

Vector& Vector::operator=(const Vector &v){
    if(this != &v){
        n=v.n;
        for(int i=0; i < n; i++)
           data[i]=v.data[i];
    }
    return *this; }

Vector::~Vector(){}
```

Implementación del TAD vector

Implementación de las operaciones: `empty`, `size`, `set`

Implementación del TAD vector

Implementación de las operaciones: `empty`, `size`, `set`

`Vector.cc`:

```
int Vector::size() const{
    return n;
}


bool Vector::empty() const{
   return n==0;
}

bool Vector::set(int i, const Elem& e){
   if( i>=0 && i < n){
       data[i]=e;
       return true;
   }
   return false;
}
```

Implementación del TAD vector

Implementación de las operaciones: `at`

- Hay que gestionar el acceso fuera del vector
- Opción 1: lanzar una excepción
- Opción 2: devolver un valor por defecto y lanzar un mensaje de advertencia por la salida de errores

Implementación del TAD vector

Implementación de las operaciones: `at`

- Hay que gestionar el acceso fuera del vector
- Opción 1: lanzar una excepción
- Opción 2: devolver un valor por defecto y lanzar un mensaje de advertencia por la salida de errores

`Vector.cc`:

```
//Opción 1
Elem Vector::at(int i) const{
    if( i >=0 && i < n ){
       return data[i];
    }else{
       throw out_of_range("Position "+to_string(i)+"does not exist");
    }
}
```

Implementación del TAD vector

Implementación de las operaciones: `at`

- Hay que gestionar el acceso fuera del vector
- Opción 1: lanzar una excepción
- Opción 2: devolver un valor por defecto y lanzar un mensaje de advertencia por la salida de errores

`Vector.cc`:

```
//Opción 2
Elem Vector::at(int i) const{
    if( i >=0 && i < n ){
        return data[i];
    }else{
        Elem e;
        cerr << "Position "+to_string(i)+"does not exist" << endl;
        return e;
    }
}
```

Implementación del TAD vector

Implementación de las operaciones: `insert` e `erase`

`Vector.cc`:

```
bool Vector::insert(int i, const Elem& e){
// Completa el código
}

bool Vector::erase(int i){
//Completa el código
}
```

Implementación del TAD vector

Implementación de las operaciones: `insert` e `erase`

`Vector.cc`:

```
bool Vector::insert(int i, const Elem& e){
    if(i <0 || i > n){
          return false;
    }

    for(int j=n-1; j>=i; j--)
        data[j+1]=data[j];

    data[i]=e;
    n++;
    return true;
}
```

Implementación del TAD vector

Implementación de las operaciones: `insert` e `erase`

`Vector.cc`:

```
bool Vector::erase(int i){
    if(i <0 || i >= n){
         return false;
    }

    for(int j=i+1; j<n; j++)
        data[j-1]=data[j];

    n--;
    return true;
}
```

Coste temporal de las operaciones

- Complejidad asintótica respecto al número de elementos en el vector (`size()`)

Operación	Caso mejor	Caso peor
`size`		
`empty`		
`at`		
`set`		
`insert`		
`erase`		

Coste temporal de las operaciones

- Complejidad asintótica respecto al número de elementos en el vector (`size()`)

Operación	Caso mejor	Caso peor
`size`	$\Omega(1)$	$O(1)$
`empty`	$\Omega(1)$	$O(1)$
`at`	$\Omega(1)$	$O(1)$
`set`	$\Omega(1)$	$O(1)$
`insert`	$\Omega(1)$	$O(n)$
`erase`	$\Omega(1)$	$O(n)$

Implementación avanzada del TAD vector

Determinación de la representación:

Implementación avanzada del TAD vector

Determinación de la representación: array dinámico de C++

Implementación avanzada del TAD vector

Determinación de la representación: array dinámico de C++

`Vector.h`:

```
typedef int Elem;
class Vector {
private:
Elem* data; //Contenido del array: dinámico, tamaño puede cambiar
int n; //Número de elementos
int capacity; //Número máximo de elementos (memoria reservada)
public:
Vector();
Vector(const Vector &);
~Vector();
Vector& operator=(const Vector &);
int size() const;
bool empty() const;
Elem at(int i) const;
bool set(int i, const Elem& e);
bool insert(int i, const Elem& e);
bool erase(int i);
};
```

Implementación avanzada del TAD vector

Implementación de las operaciones: forma canónica

Implementación avanzada del TAD vector

Implementación de las operaciones: forma canónica

`Vector.cc`:

```
Vector::Vector(){
  n=0;
  capacity=0;
  data=nullptr;
}

Vector::Vector(const Vector& v){
   n=v.n;
   capacity=v.capacity;
   if(capacity > 0){
      data=new Elem[capacity];
   }else{
      data=nullptr;
   }
   for(int i=0; i < n; i++)
       data[i]=v.data[i];
}
```

Implementación avanzada del TAD vector

Implementación de las operaciones: forma canónica

`Vector.cc`:

```
Vector& Vector::operator=(const Vector &v){
    if(this != &v){
        this->~Vector();
        n=v.n;  capacity=v.capacity;
        if(capacity > 0)
          data=new Elem[capacity];
        else
          data=nullptr;
        for(int i=0; i < n; i++)
           data[i]=v.data[i];
    }
    return *this;
}
Vector::~Vector(){
   if(capacity > 0){
       delete[] data;
       data=nullptr;
       capacity=0; }
   n=0;
}
```

Implementación avanzada del TAD vector

Implementación de las operaciones: `empty`, `size`, `set`, `erase`

Implementación avanzada del TAD vector

Implementación de las operaciones: `empty`, `size`, `set`, `erase`

`Vector.cc`:

```
// Igual que en el TAD Vector básico
```

Implementación avanzada del TAD vector

Implementación de las operaciones: `insert`

`Vector.h`:

```
typedef int Elem;
class Vector {
private:
int* data; //Contenido del array: dinámico, tamaño puede cambiar
int n; //Número de elementos
int capacity; //Número máximo de elementos (memoria reservada)
void expand();
public:
Vector();
Vector(const Vector &);
~Vector();
Vector& operator=(const Vector &);
int size() const;
bool empty() const;
Elem at(int i) const;
bool set(int i, const Elem& e);
bool insert(int i, const Elem& e);
bool erase(int i);
};
```

Implementación avanzada del TAD vector

Implementación de las operaciones: `insert`

`Vector.cc`:

```
bool Vector::insert(int i, const Elem& e){
    if(i <0 || i > n){
         return false;
    }

    if(n==capacity)
        expand();

    for(int j=n-1; j>=i; j--)
        data[j+1]=data[j];

    data[i]=e;
    n++;
    return true;
}
```

Implementación avanzada del TAD vector

Implementación de las operaciones: `insert`

`Vector.cc`:

```
void Vector::expand(){
    int newCapacity;
    if(capacity == 0)
        newCapacity=10;
    else
        newCapacity=capacity*2;

    int* newData= new Elem[newCapacity];

    // Ejercicio: continúa el código
}
```

Implementación avanzada del TAD vector

Implementación de las operaciones: `insert`

`Vector.cc`:

```
void Vector::expand(){
    int newCapacity;
    if(capacity == 0)
        newCapacity=10;
    else
        newCapacity=capacity*2;

    int* newData= new Elem[newCapacity];

    // Ejercicio: continúa el código
    for(int i=0; i < n ; i++)
        newData[i]=data[i];

    if(data != nullptr)
        delete[] data;

    capacity=newCapacity;
    data=newData;

}
```

Coste temporal de las operaciones

- Complejidad asintótica de la operación `insert` respecto al número de elementos en el vector (`size()`):

Operación	Caso mejor	Caso peor
`insert`		

Coste temporal de las operaciones

- Complejidad asintótica de la operación `insert` respecto al número de elementos en el vector (`size()`):

Operación	Caso mejor	Caso peor
`insert`	$\Omega(1)$	$O(n)$

- Ahora el caso peor se puede producir también al insertar un elemento al final
- El análisis amortizado nos dice que el coste de insertar n elementos al final del vector está en $\Theta(n)$
- El coste amortizado de insertar un elemento está, por tanto, en $O(1)$
- En términos de coste asintótico, nuestro vector avanzado no es peor que el vector básico sin redimensionamiento

Vectores en C++ STL

- No es necesario implementar nuestras propias estructuras de datos para trabajar con ellos de manera eficiente en C++
- Tampoco es necesario reservar ni liberar memoria manualmente
- Standard Template Library (STL) de C++ contiene estructuras de datos (casi) equivalentes a listas, conjuntos y diccionarios de Python
- **Template**: mecanismo que permite emplear la misma clase contenedora (por ejemplo, `Vector`), para albergar datos de tipos diferentes

Vectores en C++ STL

- Se declaran con `vector<TipoDeDato>`:

```
#include <vector>
#include <iostream>
using namespace std;
...
//Constructor por defecto: vector vacío
vector<int> v1;
cout << v1.size() << endl; //Imprime 0

//Constructor sobrecargado: número de elementos y valor
vector<int> v2(10, 0); //10 elementos con valor 0
cout << v2.size() << endl; //Imprime 10

//Constructor a partir de inicializador
vector<int> v3({8, 4, 6});
cout << v3.size() << endl; //Imprime 3
```

Vectores en C++ STL

- Acceso con `at`: excepción si nos salimos del rango

```
for(int i=0; i<v2.size(); i++ )
    cout << v2.at(i) << " ";     //Imprime: 0 0 0 0 0 0 0 0 0 0
for(int i=0; i<v3.size(); i++ )
    cout << v3.at(i) << " ";  //Imprime: 8 4 6
cout << v3.at(8); //Se produce una excepción y el programa acaba
```

- Acceso con `[]`: no se comprueba el rango (más rápido)

```
for(int i=0; i<v2.size(); i++ )
    cout << v2[i] << " ";     //Imprime: 0 0 0 0 0 0 0 0 0 0
for(int i=0; i<v3.size(); i++ )
    cout << v3[i] << " ";  //Imprime: 8 4 6
cout << v3[8]; //Imprime:
```

Vectores en C++ STL

- El método `at`
  y el operador `[]` también se pueden utilizar para modificar el vector

```
v3[0]=10;
v3.at(1)=20;

for(int i=0; i<v3.size(); i++ )
    cout << v3[i] << " ";  //Imprime: 10 20 6

v3[10]=80; //No se produce error
v3.at(10)=80; //Se produce una excepción y el programa acaba
```

Vectores en C++ STL

- Inserción al final con `push_back`

```
v3.push_back(40);

for(int i=0; i<v3.size(); i++ )
    cout << v3[i] << " ";  //Imprime: 10 20 6 40
```

- Inserción en cualquier posición con `insert(v.begin()+posicion,valor)`

```
v3.insert(v3.begin(),44);
for(int i=0; i<v3.size(); i++ )
    cout << v3[i] << " ";  //Imprime: 44 10 20 6 40

v3.insert(v3.begin()+2,33);
for(int i=0; i<v3.size(); i++ )
    cout << v3[i] << " ";  //Imprime: 44 10 33 20 6 40
```

Vectores en C++ STL

- Borrado del final con `pop_back`

```
v3.pop_back();

for(int i=0; i<v3.size(); i++ )
    cout << v3[i] << " ";  //Imprime: 44 10 33 20 6
```

- Borrado en cualquier posición con `erase(v.begin()+posicion)`

```
v3.erase(v3.begin());
for(int i=0; i<v3.size(); i++ )
    cout << v3[i] << " ";  //Imprime: 10 33 20 6

v3.erase(v3.begin()+2);
for(int i=0; i<v3.size(); i++ )
    cout << v3[i] << " ";  //Imprime: 10 33 6
```

Vectores en C++ STL

Ejercicio

Escribe un programa que lea números enteros de entrada estándar y los vaya insertando en un vector, de manera que éste siempre está ordenado. El programa acabará cuando se lea el número `-1`

```
#include<iostream>
#include<vector>
using namespace std;

int main(){
  vector<int> v;
  int num;
  cin >> num;
  while(num != -1){
     //Completa el código aquí

     cin >> num;
  }
  return 0;
}
```

Ejercicio

Escribe un programa que lea números enteros de entrada estándar y los vaya insertando en un vector, de manera que éste siempre está ordenado ascendentemente. El programa acabará cuando se lea el número -1

```
//Completa el código aquí
int i=0;
while(i<v.size()){
  if(num<v[i])
     break;
  i++;
}
v.insert(v.begin()+i,num);
```

Pregunta

¿Cuál es la complejidad asintótica de insertar un elemento en un vector ordenado de tamaño n? ¿Cuál es la complejidad asintótica de buscar en dicho vector?

Vectores en C++ STL

Pregunta

¿Cuál es la complejidad asintótica de insertar un elemento en un vector ordenado de tamaño n? ¿Cuál es la complejidad asintótica de buscar en dicho vector?

- Inserción: $\Theta(n)$
- Búsqueda: $\Omega(1)$ (elemento está en el centro); $O(\log n)$ (elemento no está)

TAD vector: resumen

- Colección de elementos almacenados en cierto orden
- Equivalente al tipo `list` de Python
- Insertar “en medio” del vector es ineficiente: $O(n)$
- Ampliación de memoria automática con coste $O(n)$
- La ampliación es infrecuente pero puede ser problemática si queremos todas las inserciones sean rápidas
- Permite búsqueda binaria en $O(\log n)$

Índice

Ejemplo introductorio: historial de URLs

Imagina que necesitamos una clase para gestionar el historial de URLs de un navegador web. El navegador invocará al método `click` cada vez que el usuario haga click en un enlace. El navegador invocará al método `back` cuando el usuario haga click en el botón de "atrás" y debe devolver la URL a la que debe redirigir el navegador.

Ejemplo introductorio: historial de URLs

Especificación del TAD Historial

Operaciones:

- Almacena la URL a la que se acaba de acceder

```
void click(const string &);
```

- Descarta la URL actual y devuelve la anterior

```
string back();
```

Ejemplo introductorio: historial de URLs

Especificación del TAD Historial

Ejemplos (historial inicialmente vacío):

Operación	**Salida**	**Contenido**
click(“eps.ua.es”)		“eps.ua.es”
click(“ua.es”)		“eps.ua.es”,“ua.es”
click(“cloud.ua.es”)		“eps.ua.es”,“ua.es”,“cloud.ua.es”
back()	“ua.es”	“eps.ua.es”,“ua.es”
back()	“eps.ua.es”	“eps.ua.es”
click(“cplusplus.com”)		“eps.ua.es”,“cplusplus.com”
back()	“eps.ua.es”	“eps.ua.es”

Ejemplo introductorio: historial de URLs

Impementación del TAD Historial

Determinación de la representación:

Ejemplo introductorio: historial de URLs

Impementación del TAD Historial

Determinación de la representación: vector STL

Ejemplo introductorio: historial de URLs

Impementación del TAD Historial

Determinación de la representación: vector STL

`Historial.h`:

```
class Historial{
private:
  vector<string> data;
public:
  Historial();
  Historial(const Historial&);
  ~Historial();
  Historial& operator=(const Historial&);
  void click(const string &);
  string back();
};
```

Ejemplo introductorio: historial de URLs

Impementación del TAD Historial

Implementación de las operaciones: forma canónica

Ejemplo introductorio: historial de URLs

Impementación del TAD Historial

Implementación de las operaciones: forma canónica

`Historial.cc`:

```
Historial::Historial(){}

Historial::Historial(const Historial& h):data(h.data){}

Historial::~Historial(){}

Historial& Historial::operator=(const Historial& h){
    if(&h != this){
        (*this).~Historial();
        data=h.data;
    }
    return (*this);
}
```

Ejemplo introductorio: historial de URLs

Impementación del TAD Historial

Implementación de las operaciones: `click` y `back`

Ejemplo introductorio: historial de URLs

Impementación del TAD Historial

Implementación de las operaciones: `click` y `back`

`Historial.cc`:

```
void Historial::click(const string &s){
    data.push_back(s);
}

string Historial::back(){
    if(data.size() > 1){
        data.pop_back();
        return data[data.size()-1];
    }
    return "";
}
```

Ejemplo introductorio: historial de URLs

Pregunta

¿Cuál es la complejidad asintótica de la operaciones `click` y `back` para un historial con n URLs? ¿Cambiaría la complejidad si insertáramos las URLs al principio del vector?

Ejemplo introductorio: historial de URLs

Pregunta

¿Cuál es la complejidad asintótica de la operaciones `click` y `back` para un historial con n URLs? ¿Cambiaría la complejidad si insertáramos las URLs al principio del vector?

- `click`: $\Omega(1)$ (no hay redimensionamiento); $O(n)$ (hay redimensionamiento). El coste amortizado es $O(1)$
- `back`: $\Theta(1)$

Si insertamos al principio del vector:

- `click`: $\Theta(n)$
- `back`: $\Theta(n)$

TAD pila

- Podemos ver la lista de URLs como una “pila” de platos sucios
- Sólo podemos acceder al plato de más arriba, que es último que hemos “apilado”
- Se trata de una estructura de datos *LIFO: Last In, First Out*
- El uso de este tipo de estructuras es muy común en informática:
  - Pilas de llamadas a funciones en compiladores
  - Resaltado de parejas de paréntesis en editores de texto
  - Procesamiento de HTML en navegadores web

TAD pila

Especificación del TAD pila

Definición: Colección de *n* elementos almacenados en cierto orden, tal que podemos identificar al primer elemento, segundo, tercero, etc. Sólo podemos conocer el valor del primer elemento (cima), y las inserciones y borrados sólo pueden realizarse al principio de la colección.

TAD pila

Especificación del TAD pila

Operaciones:

- Obtiene el número de elementos almacenados en la pila

```
int size() const;
```

- Comprueba si la pila está vacía

```
bool empty() const;
```

- Devuelve el elemento en la cima de la pila

```
Elem top() const;
```

TAD pila

Especificación del TAD pila

Operaciones:

- Apila un elemento

```
void push(const Elem& e);
```

- Desapila un elemento. Devuelve `false` si la pila estaba vacía, y `true` si se ha podido desapilar el elemento

```
bool pop();
```

Implementación del TAD pila

Determinación de la representación: vector STL

`Pila.h`:

```
typedef int Elem;
class Pila{
private:
  vector<Elem> data;
public:
  Pila();
  Pila(const Pila&);
  ~Pila();
  Pila& operator=(const Pila&);
  int size() const;
  bool empty() const;
  Elem top() const;
  void push(const Elem& e);
  bool pop();
};
```

Implementación del TAD pila

Implementación de las operaciones:

`Pila.cc`:

```
Elem Pila::top(){
   //Ejercicio: implementa este método
}

void Pila::push(const Elem& e){
   //Ejercicio: implementa este método
}

bool Pila::pop(){
  //Ejercicio: implementa este método
}
```

Implementación del TAD pila

Implementación de las operaciones:

`Pila.cc`:

```
Elem Pila::top(){
    Elem e;
    if(empty())
        return e;
    else
        return data[data.size()-1];
}
void Pila::push(const Elem& e){
    data.push_back(e);
}
bool Pila::pop(){
  if(empty())
        return false;
  data.pop_back();
  return true;
}
```

TAD pila

Pregunta

¿Cuál es la complejidad asintótica de la operaciones `top`, `push` y `pop` para una pila con n elementos?

TAD pila

Pregunta

¿Cuál es la complejidad asintótica de la operaciones `top`, `push` y `pop` para una pila con n elementos?

- `top`: $\Theta(1)$
- `push`: $\Omega(1)$ (no hay redimensionamiento); $O(n)$ (hay redimensionamiento). El coste amortizado es $O(1)$
- `pop`: $\Theta(1)$

Pilas en C++ STL

- Se declaran con `stack<TipoDeDato>`:

```
#include <stack>
#include <iostream>
using namespace std;
...
//Constructor por defecto: pila vacía
stack<int> s;

//Posible "segmentation fault"
cout << s.top()<< endl; //Devuelve un valor indeterminado

//Posible "segmentation fault"
s.pop(); //Devuelve void, no hace nada

s.push(4);
s.push(5);
s.push(6);
cout << s.top()<< endl; //Imprime: 6
s.pop();
cout << s.top()<< endl; //Imprime: 5
```

Pilas en C++ STL

Ejercicio

Implementa las operaciones `next` y `back` del TAD Historial asumiendo que la representación es ahora de tipo `stack`

Pilas en C++ STL

Ejercicio

Implementa las operaciones `next` y `back` del TAD Historial asumiendo que la representación es ahora de tipo `stack`

`Historial.cc`:

```
void Historial::click(const string &s){
   data.push(s);
}
string Historial::back(){
    if(data.size() > 1){
        data.pop();
        return data.top(=;
    }
    return "";
}
```

Índice

TAD cola

- Para algunas aplicaciones, se hace necesario que el elemento a extraer de la pila no sea el último que se ha insertado, sino el primero
- Tendríamos, por tanto, una “cola” de datos (similar a una cola de personas), en la cual los elementos se insertan al “fondo” y se extraen del “frente”
- Estructura *FIFO: First In, First Out*
- Su uso también es muy común:
  - Colas de impresión
  - Colas para el procesamiento de peticiones en servicios web
  - Recorridos de árboles (lo veremos más adelante en la asignatura)

TAD cola

Especificación del TAD cola

Definición: Colección de n elementos almacenados en cierto orden, tal que sólo se puede acceder y eliminar el primer elemento (frente), y sólo se pueden insertar elementos al final (fondo).

TAD cola

Especificación del TAD cola

Operaciones:

- Obtiene el número de elementos almacenados en la cola

```
int size() const;
```

- Comprueba si la cola está vacía

```
bool empty() const;
```

- Devuelve el elemento en el frente de la cola

```
Elem front() const;
```

TAD cola

Especificación del TAD cola

Operaciones:

- Añade un elemento al fondo de la cola

```
void enqueue(const Elem& );
```

- Elimina el elemento al frente de la cola. Si la cola está vacía y no hay elemento en el frente de la cola, devuelve `false`. Devuelve `true` en caso contrario.

```
bool dequeue();
```

TAD cola

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`		
`enqueue(3)`		
`front()`		
`size()`		
`dequeue()`		
`enqueue(7)`		
`dequeue()`		
`front()`		
`dequeue()`		
`dequeue()`		
`empty()`		

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`	-	(5)
`enqueue(3)`		
`front()`		
`size()`		
`dequeue()`		
`enqueue(7)`		
`dequeue()`		
`front()`		
`dequeue()`		
`dequeue()`		
`empty()`		

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(5,3)
`front()`		
`size()`		
`dequeue()`		
`enqueue(7)`		
`dequeue()`		
`front()`		
`dequeue()`		
`dequeue()`		
`empty()`		

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(5,3)
`front()`	5	(5,3)
`size()`		
`dequeue()`		
`enqueue(7)`		
`dequeue()`		
`front()`		
`dequeue()`		
`dequeue()`		
`empty()`		

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(5,3)
`front()`	5	(5,3)
`size()`	2	(5,3)
`dequeue()`		
`enqueue(7)`		
`dequeue()`		
`front()`		
`dequeue()`		
`dequeue()`		
`empty()`		

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(5,3)
`front()`	5	(5,3)
`size()`	2	(5,3)
`dequeue()`	true	(3)
`enqueue(7)`		
`dequeue()`		
`front()`		
`dequeue()`		
`dequeue()`		
`empty()`		

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(5,3)
`front()`	5	(5,3)
`size()`	2	(5,3)
`dequeue()`	true	(3)
`enqueue(7)`	-	(3,7)
`dequeue()`		
`front()`		
`dequeue()`		
`dequeue()`		
`empty()`		

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(5,3)
`front()`	5	(5,3)
`size()`	2	(5,3)
`dequeue()`	true	(3)
`enqueue(7)`	-	(3,7)
`dequeue()`	-	(7)
`front()`		
`dequeue()`		
`dequeue()`		
`empty()`		

TAD cola

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(5,3)
`front()`	5	(5,3)
`size()`	2	(5,3)
`dequeue()`	true	(3)
`enqueue(7)`	-	(3,7)
`dequeue()`	-	(7)
`front()`	7	(7)
`dequeue()`		
`dequeue()`		
`empty()`		

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(5,3)
`front()`	5	(5,3)
`size()`	2	(5,3)
`dequeue()`	true	(3)
`enqueue(7)`	-	(3,7)
`dequeue()`	-	(7)
`front()`	7	(7)
`dequeue()`	true	()
`dequeue()`		
`empty()`		

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(5,3)
`front()`	5	(5,3)
`size()`	2	(5,3)
`dequeue()`	true	(3)
`enqueue(7)`	-	(3,7)
`dequeue()`	-	(7)
`front()`	7	(7)
`dequeue()`	true	()
`dequeue()`	false	()
`empty()`		

Especificación del TAD cola

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**frente** $\leftarrow$ **cola** $\leftarrow$ **fondo**
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(5,3)
`front()`	5	(5,3)
`size()`	2	(5,3)
`dequeue()`	true	(3)
`enqueue(7)`	-	(3,7)
`dequeue()`	-	(7)
`front()`	7	(7)
`dequeue()`	true	()
`dequeue()`	false	()
`empty()`	true	()

Implementación del TAD cola

Determinación de la representación:

Implementación del TAD cola

Determinación de la representación: vector STL

Implementación del TAD cola

Determinación de la representación: vector STL

`Cola.h`:

```
typedef int Elem;
class Cola{
private:
  vector<Elem> data;
public:
  Cola();
  Cola(const Cola&);
  ~Cola();
  Cola& operator=(const Cola&);
  int size() const;
  bool empty() const;
  Elem front() const;
  void enqueue(const Elem& e);
  bool dequeue();
};
```

Implementación del TAD cola

Implementación de las operaciones:

`Cola.cc`:

```
Elem Cola::front(){
    Elem e;
    if(empty())
        return e;
    else
        return data[0];
}
void Cola::enqueue(const Elem& e){
    data.push_back(e);
}
bool Cola::dequeue(){
  if(empty())
        return false;
  else
        data.erase(data.begin());
  return true;
}
```

TAD cola

Pregunta

¿Cuál es la complejidad asintótica de la operaciones `front`, `enqueue` y `dequeue` para una cola con n elementos en la implementación que acabamos de presentar?

TAD cola

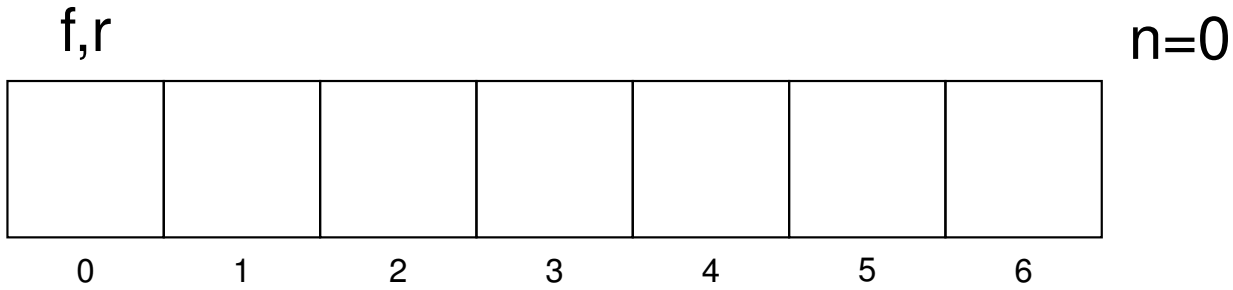
Pregunta

¿Cuál es la complejidad asintótica de la operaciones `front`, `enqueue` y `dequeue` para una cola con n elementos en la implementación que acabamos de presentar?

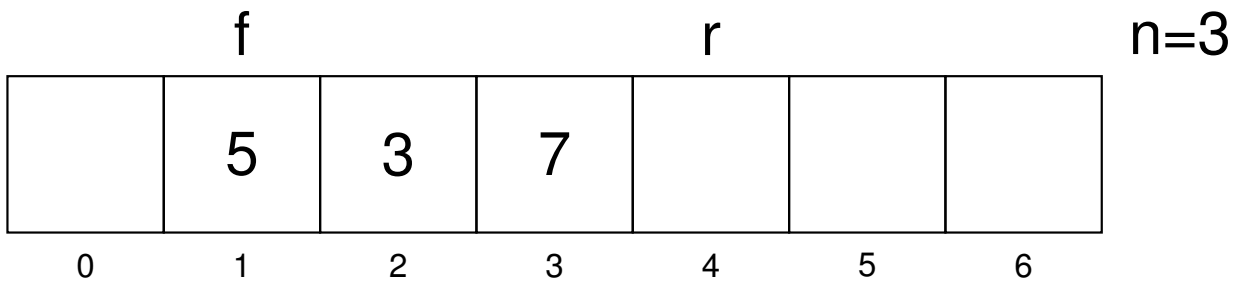
- `front`: $\Theta(1)$
- `enqueue`: $\Omega(1)$ (no hay redimensionamiento); $O(n)$ (hay redimensionamiento). El coste amortizado es $O(1)$
- `dequeue`: $\Theta(n)$

Implementación eficiente de una cola

- Empleo de un array de C++ (estático)
- 3 variables controlan el acceso:
  - n: número de elementos
  - f: posición del frente de la cola
  - r: posición **siguiente** al fondo de la cola
- Inicialmente, $n = f = r = 0$



- Tras una serie de operaciones:



Implementación eficiente de una cola

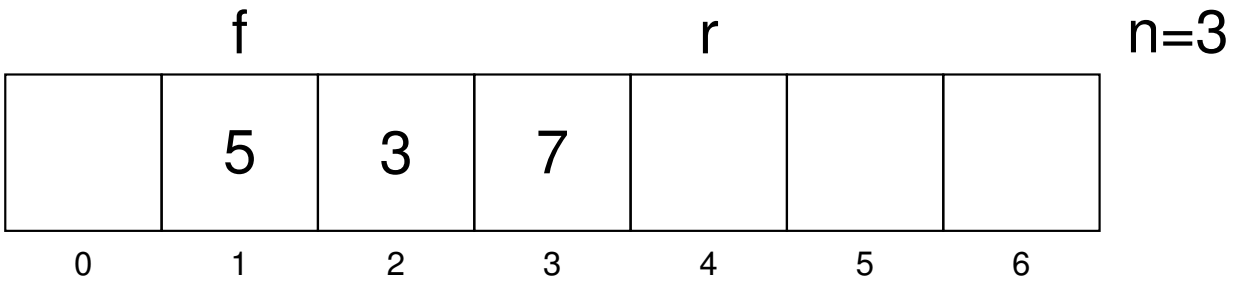
- `enqueue(e):array[r]=e; r++; n++`
- `dequeue(): f++; n--`
- `front(): array[f];`
- Complejidad:

Implementación eficiente de una cola

- `enqueue(e):array[r]=e; r++; n++`
- `dequeue(): f++; n--`
- `front(): array[f];`
- Complejidad: $\Theta(1)$ para las tres operaciones

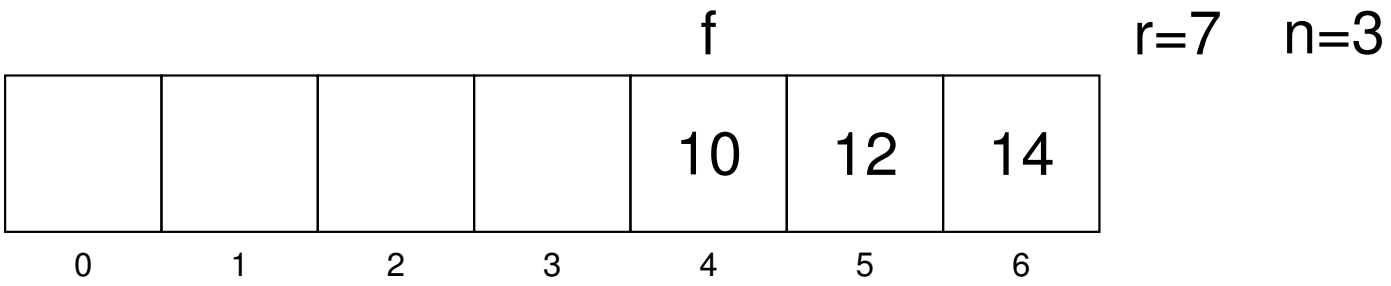
Ejercicio

Aplica estas operaciones sobre la siguiente cola: `dequeue();` `dequeue(); enqueue(10); enqueue(12);dequeue();` `enqueue(14)`



Implementación eficiente de una cola

Solución:

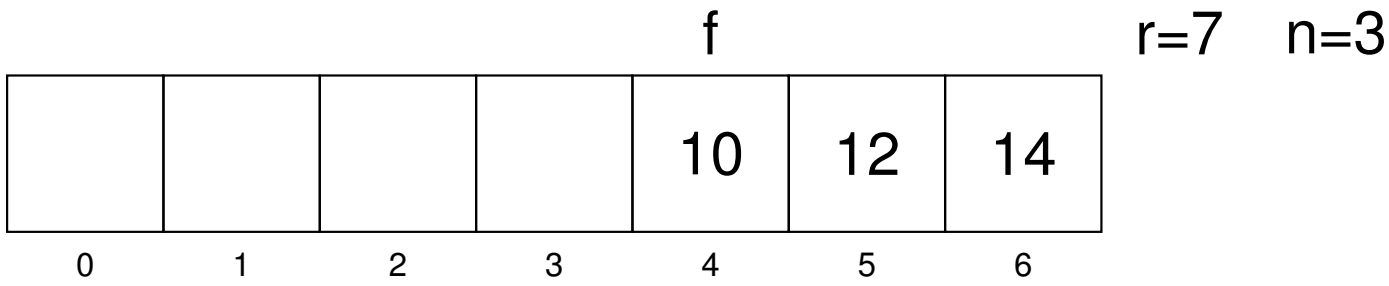


Pregunta

- ¿La cola está llena? ¿Puedo insertar?
- ¿Hay alguna manera de aprovechar mejor la memoria?

Implementación eficiente de una cola

Solución:



Pregunta

- ¿La cola está llena? ¿Puedo insertar?
- ¿Hay alguna manera de aprovechar mejor la memoria?

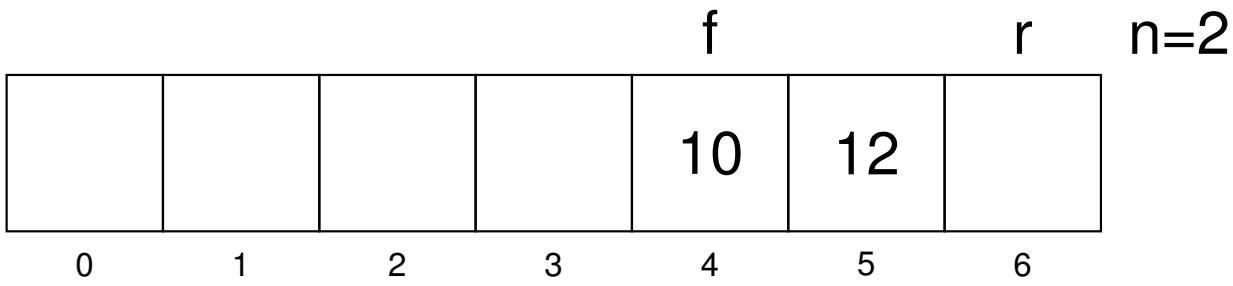
- n es menor que el tamaño del array (N), pero la condición de cola llena es $r == N$. No se puede insertar
- Solución: **colas circulares**

Colas circulares

- La cola continúa desde el final al principio
- Operaciones:
  - `enqueue(e):array[r]=e; r=(r+1) % N; n++`
  - `dequeue(): f=(f+1) % N; n--`
  - `front(): array[f];`
- La condición de cola llena pasa a ser `n == N`
- Complejidad: $\Theta(1)$ para las tres operaciones

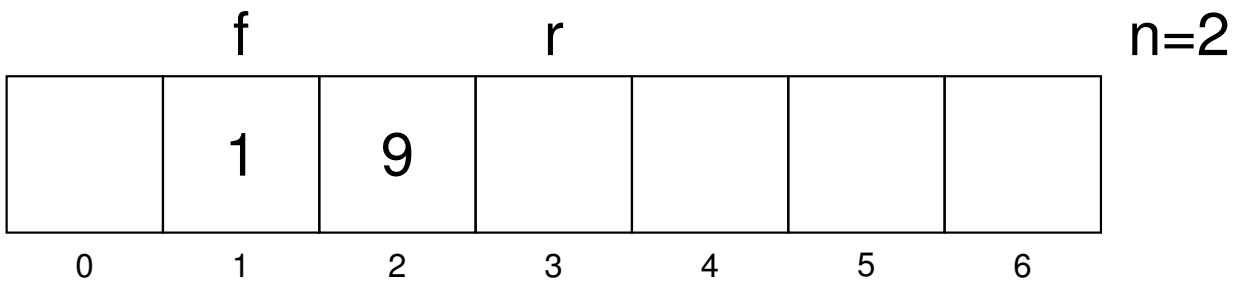
Ejercicio

Aplica estas operaciones sobre la siguiente cola: `enqueue(3); enqueue(7);enqueue(1);enqueue(9); dequeue() x 4`



Colas circulares

Solución:



Colas circulares

- La cola circular también se puede redimensionar cuando se llena
- El coste de redimensionar una cola con n elementos es $\Theta(n)$
- El algoritmo es algo más complicado que para vectores: se deja como ejercicio su implementación
- El coste de la operación `enqueue` pasa a ser $O(n)$, pero de nuevo el coste amortizado de "encolar" n elementos es $O(n)$ y el de "encolar" uno, $O(1)$

Colas en C++ STL

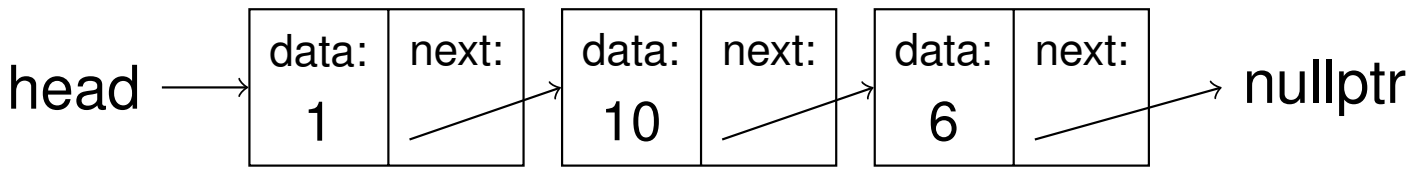
- Se declaran con `queue<TipoDeDato>`:

```
#include <queue>
#include <iostream>
using namespace std;
...
//Constructor por defecto: cola vacía
queue<int> q;

//Posible "segmentation fault"
//cout << q.front()<< endl; //Devuelve un valor indeterminado

//Posible "segmentation fault"
//q.pop(); //"dequeue". Devuelve void

q.push(4); // "enqueue"
q.push(5);
q.push(6);
cout << q.front()<< endl; //Imprime: 4
q.pop();
cout << q.front()<< endl; //Imprime: 5
```

Índice

Listas enlazadas

- Existe una alternativa al empleo de arrays dinámicos para el almacenamiento de un número variable de datos
- Conjunto de objetos instancia de una clase `Nodo`
- Cada nodo contiene un dato y un puntero al siguiente nodo
- El puntero del último nodo tiene valor `nullptr`



Listas enlazadas

```
typedef int Elem;
class Node{
private:
Node* next;
Elem data;
public:
//Constructor sobrecargado
Node(const Elem&, Node*);
//Getters and setters
//etc.
};
class List{
private:
Node* head;

public:
//...

};
```

Implementación de una pila con listas enlazadas

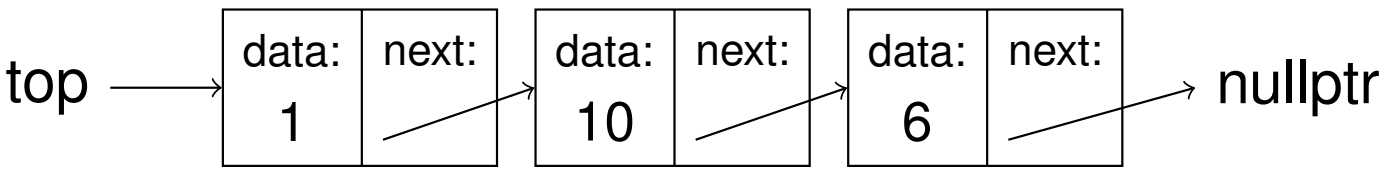
Implementación del TAD pila

Determinación de la representación: lista enlazada

`Pila.h`:

```
typedef int Elem;
class Pila{
private:
  Node* top;
public:
  Pila();
  Pila(const Pila&);
  ~Pila();
  Pila& operator=(const Pila&);
  int size() const;
  bool empty() const;
  Elem top() const;
  void push(const Elem& e);
  bool pop();
};
```

Implementación de una pila con listas enlazadas



Implementación de una pila con listas enlazadas

Implementación del TAD pila

Implementación de las operaciones: `push` y `pop`

`Pila.cc`:

```
void Pila::push(const Elem& e){
    Node* node=new Node(e,top);
    top=node;
}
bool Pila::pop(){
    if(empty())
        return false;
    Node* aux=top;
    top=top->getNext();
    delete aux;
    return true;
}
```

Implementación de una pila con listas enlazadas

Implementación del TAD pila

Implementación de las operaciones: `top` y `empty`

`Pila.cc`:

```
Elem Pila::top(){
    //Ejercicio: implementa este método
}
bool Pila::empty(){
    //Ejercicio: implementa este método
}
```

Implementación del TAD pila

Implementación de las operaciones: `top` y `empty`

`Pila.cc`:

```
Elem Pila::top(){
    Elem e;
    if(empty())
        return e;
    return top->getElem();
}
bool Pila::empty(){
    //Ejercicio: implementa este método
    return top==nullptr;
}
```

Implementación de una pila con listas enlazadas

Pregunta

¿Cuál es la complejidad asintótica de la operaciones `push` y `pop` para una pila con n elementos en la implementación que acabamos de presentar?

Implementación de una pila con listas enlazadas

Pregunta

¿Cuál es la complejidad asintótica de la operaciones `push` y `pop` para una pila con n elementos en la implementación que acabamos de presentar?

- `push`: $\Theta(1)$
- `pop`: $\Theta(1)$

Implementación de una cola con listas enlazadas

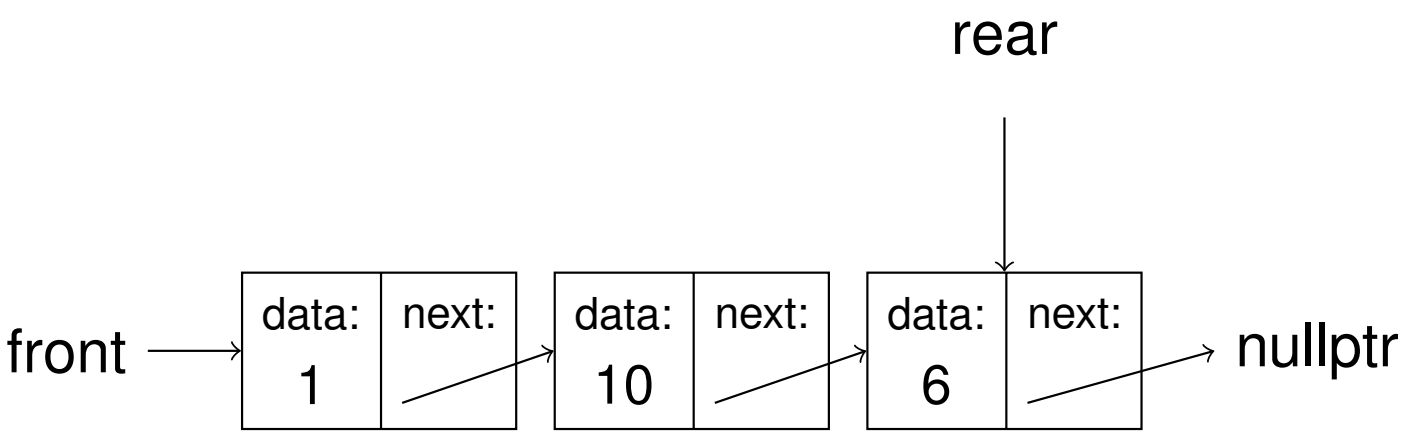
Implementación del TAD cola

Determinación de la representación: lista enlazada

`Cola.h`:

```
typedef int Elem;
class Cola{
private:
  Node* front;
  Node* rear;
public:
  Cola();
  Cola(const Cola&);
  ~Cola();
  Cola& operator=(const Cola&);
  int size() const;
  bool empty() const;
  Elem front() const;
  void enqueue(const Elem& e);
  bool dequeue();
};
```

Implementación de una cola con listas enlazadas



Implementación de una cola con listas enlazadas

Implementación del TAD cola

Implementación de las operaciones: `enqueue` y `dequeue`

`Cola.cc`:

```
void Cola::enqueue(const Elem& e){
    Node* node = new Node(e,nullptr);
    if (!empty()){
         rear->setNext(node);
         rear=node;
    }else{
        front=rear=node;
    }}
bool Cola::dequeue(){
    if(empty())
        return false;
    Node* aux=front;
    front=front->getNext();
    delete aux;
    if(front==nullptr)
         rear=nullptr;
    return true;   }
```

Implementación de una cola con listas enlazadas

Pregunta

¿Cuál es la complejidad asintótica de la operaciones `enqueue` y `dequeue` para una cola con n elementos en la implementación que acabamos de presentar?

Implementación de una cola con listas enlazadas

Pregunta

¿Cuál es la complejidad asintótica de la operaciones `enqueue` y `dequeue` para una cola con n elementos en la implementación que acabamos de presentar?

- `enqueue`: $\Theta(1)$
- `dequeue`: $\Theta(1)$

Implementación de una cola con listas enlazadas

Pregunta

¿Cuál es la complejidad asintótica de la operaciones `enqueue` y `dequeue` para una cola con n elementos en la implementación que acabamos de presentar?

- `enqueue`: $\Theta(1)$
- `dequeue`: $\Theta(1)$

Pregunta

¿C++ STL utiliza arrays en vez de listas enlazadas para implementar `stack` y `queue`. ¿Por qué?

Implementación de una cola con listas enlazadas

Pregunta

¿Cuál es la complejidad asintótica de la operaciones `enqueue` y `dequeue` para una cola con n elementos en la implementación que acabamos de presentar?

- `enqueue`: $\Theta(1)$
- `dequeue`: $\Theta(1)$

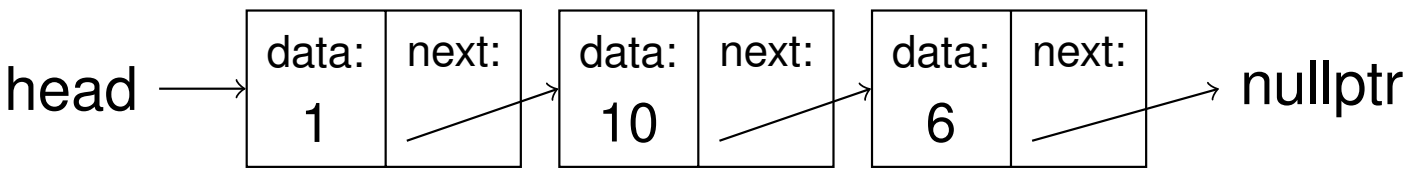
Pregunta

¿C++ STL utiliza arrays en vez de listas enlazadas para implementar `stack` y `queue`. ¿Por qué?

- Complejidad espacial: aunque ambas implementaciones tiene complejidad espacial $\Theta(n)$, la implementación con listas enlazadas requiere un puntero por cada dato almacenado

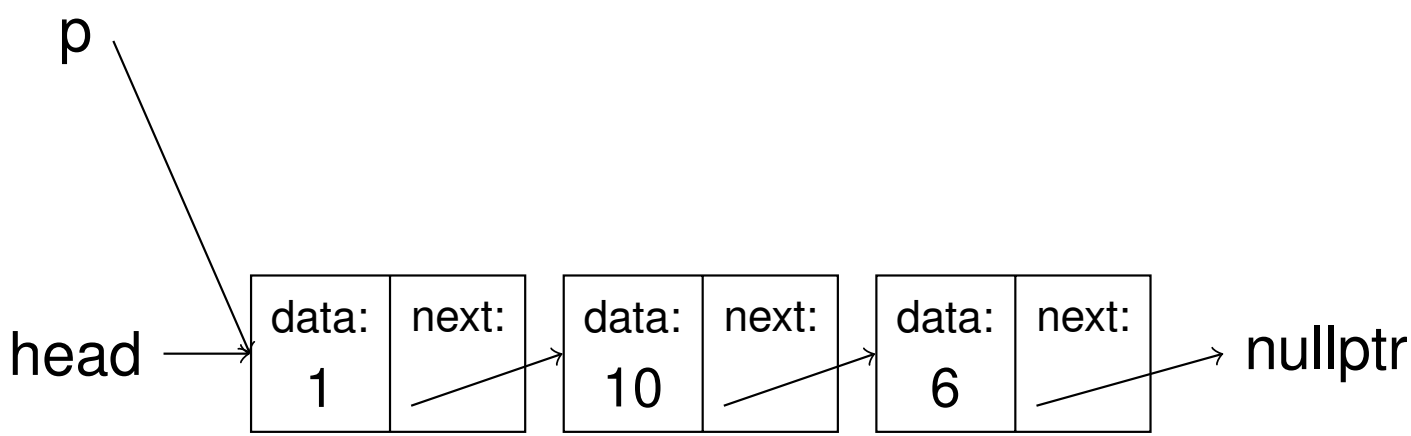
Iterando sobre los elementos de una lista enlazada

```
Node* p=head;
while(p != nullptr){
    cout << p->getData() << endl;
    p=p->getNext();
}
```



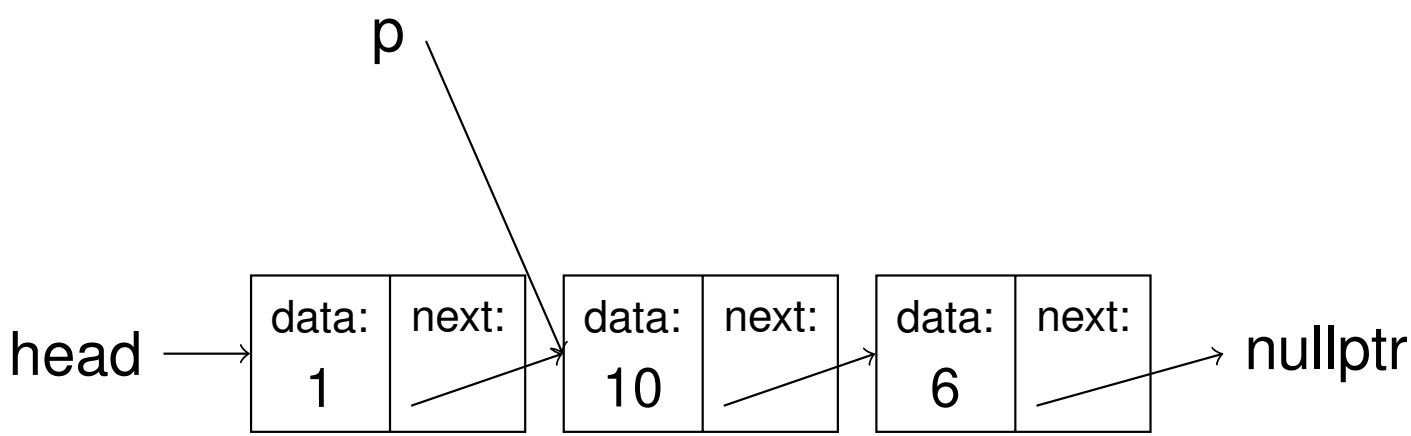
Iterando sobre los elementos de una lista enlazada

```
Node* p=head;
while(p != nullptr){
    cout << p->getData() << endl;
    p=p->getNext();
}
```



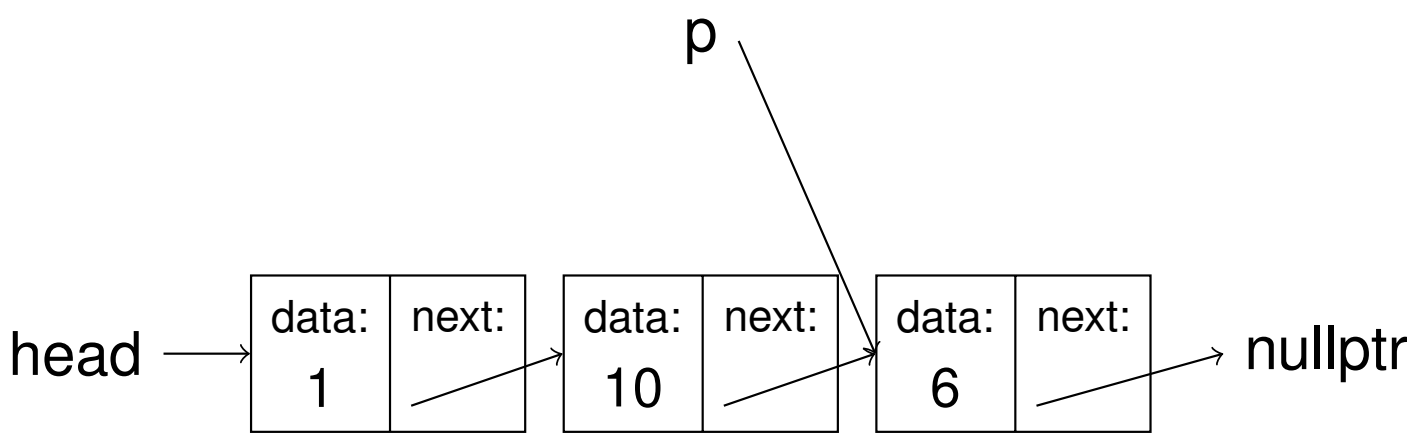
Iterando sobre los elementos de una lista enlazada

```
Node* p=head;
while(p != nullptr){
    cout << p->getData() << endl;
    p=p->getNext();
}
```



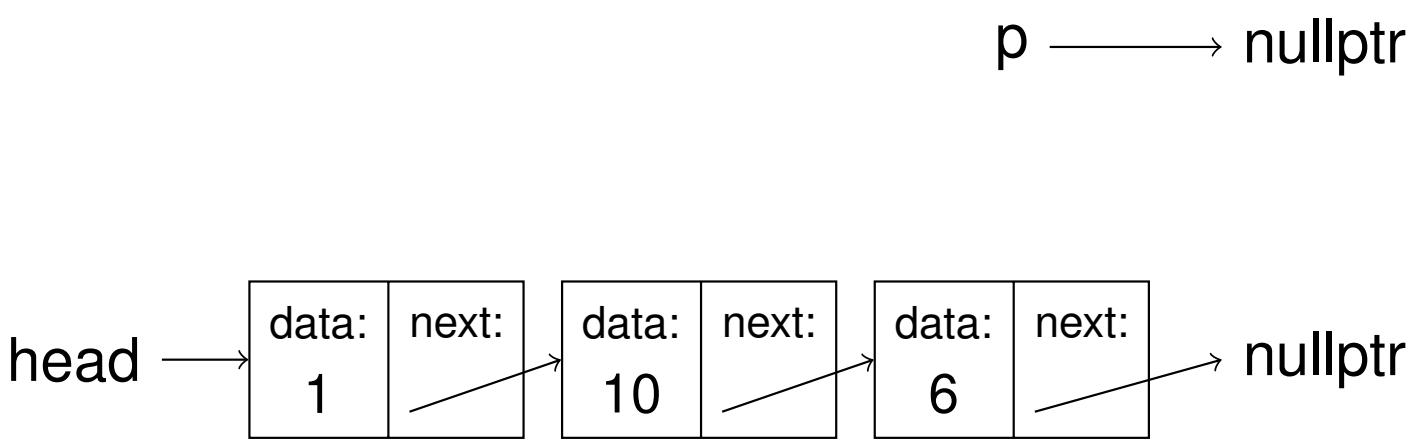
Iterando sobre los elementos de una lista enlazada

```
Node* p=head;
while(p != nullptr){
    cout << p->getData() << endl;
    p=p->getNext();
}
```



Iterando sobre los elementos de una lista enlazada

```
Node* p=head;
while(p != nullptr){
    cout << p->getData() << endl;
    p=p->getNext();
}
```



Listas enlazadas

Preguntas

- ¿Se puede emplear una lista enlazada como representación del TAD vector?
- En caso afirmativo, ¿cuál sería la complejidad de las operaciones `at`, `set`, `insert` y `erase`?

Listas enlazadas

Preguntas

- ¿Se puede emplear una lista enlazada como representación del TAD vector?
- En caso afirmativo, ¿cuál sería la complejidad de las operaciones `at`, `set`, `insert` y `erase`?

- Sí, aunque no es lo más recomendable:
  - `at`: $\Omega(1)$, $O(n)$
  - `set`: $\Omega(1)$, $O(n)$
  - `insert`: $\Omega(1)$, $O(n)$
  - `erase`: $\Omega(1)$, $O(n)$
- Dado un índice numérico, el acceso al nodo correspondiente cuesta $O(n)$, y eso ralentiza todas las operaciones
- Recorrer todos los elementos del vector con un bucle `for` tiene un coste de $O(n^2)$

Índice

TAD lista

- El acceso mediante índice del TAD vector hace que emplear listas enlazadas para implementarlo sea muy ineficiente
- El TAD lista representa una secuencia de elementos (al igual que el TAD vector), pero el acceso a los mismos se realiza mediante un *iterador* en lugar de un índice
- Un iterador es una abstracción del concepto de posición en la lista
- El iterador normalmente comienza representando la primera posición de la lista, y se va incrementando para avanzar en la misma
- El empleo de iteradores permite ocultar los punteros a nodo y encapsular correctamente la información

TAD lista

Especificación del TAD lista

Definición: Colección de *n* elementos almacenados en cierto orden, tal que podemos identificar al primer elemento, segundo, tercero, etc. Podemos acceder a cada elemento usando un iterador. Un iterador es otro TAD que representa la posición de un elemento en una lista. Una lista permite obtener un iterador que apunta a su primer elemento, y el acceso al resto de elementos se realiza mediante incrementos sucesivos del iterador

Especificación del TAD lista

Operaciones:

- Devuelve un iterador que apunta al primer elemento de la lista. Devuelve el mismo valor que `end()` si la lista está vacía

```
Iterador begin() const;
```

- Devuelve un iterador que apunta al elemento imaginario que se encuentra tras el último elemento de la lista

```
Iterador end() const;
```

- Devuelve el elemento en la posición aputada por el iterador `it`

```
Elem get(Iterator it) const;
```

Especificación del TAD lista

Operaciones:

- Comprueba si la lista está vacía

```
bool empty() const;
```

- Inserta un elemento al principio de la lista

```
void insertFront(const Elem& e);
```

- Inserta un elemento exactamente antes de la posición apuntada por el iterador `it`

```
void insert(Iterador it,const Elem& e);
```

Especificación del TAD lista

Operaciones:

- Asigna el valor `e` al elemento de la lista que ocupa la posición a la que apunta el iterador `it`.Devuelve `false` si el iterador no apuntaba a ninguna posición ocupada por un elemento y `true` en caso contrario

```
bool set(Iterador it, const Elem & e);
```

- Obtiene el elemento que ocupa la posición de la lista a la que apunta el iterador `it`

```
Elem get(Iterador it);
```

Especificación del TAD lista

Operaciones:

- Elimina el primer elemento de la lista. Devuelve `false` si la lista está vacía y no se puede borrar el primer elemento y `true` en caso contrario

```
bool eraseFront();
```

- Elimina el elemento que se ocupa la posición apuntada por el iterador `it`. Invalida el iterador. Devuelve `false` si el iterador no apuntaba a ninguna posición ocupada por un elemento y `true` en caso contrario

```
bool erase(Iterador &it);
```

Especificación del TAD iterador

Operaciones:

- Avanza el iterador a la siguiente posición de la lista. Si la posición actual es la última, el iterador pasará a apuntar al elemento imaginario que hay tras el último elemento de la lista. Si el iterador ya apunta a este elemento imaginario, la operación `step()` no hace nada

```
void step();
```

Especificación del TAD lista

Ejemplos (lista inicialmente vacía):

Operación	Salida	Lista
`insertFront(8)`		
`p=begin()`		
`p.step()`		
`insert(p,5)`		
`q=begin()`		
`set(q,1)`		
`eraseFront()`		
`set(q,1)`		

Especificación del TAD lista

Ejemplos (lista inicialmente vacía):

Operación	Salida	Lista
`insertFront(8)`	-	(8)
`p=begin()`		
`p.step()`		
`insert(p,5)`		
`q=begin()`		
`set(q,1)`		
`eraseFront()`		
`set(q,1)`		

Especificación del TAD lista

Ejemplos (lista inicialmente vacía):

Operación	Salida	Lista
`insertFront(8)`	-	(8)
`p=begin()`	p: (8)	(8)
`p.step()`		
`insert(p,5)`		
`q=begin()`		
`set(q,1)`		
`eraseFront()`		
`set(q,1)`		

Especificación del TAD lista

Ejemplos (lista inicialmente vacía):

Operación	**Salida**	**Lista**
`insertFront(8)`	-	(8)
`p=begin()`	p: (8)	(8)
`p.step()`	p: (-)	(8)
`insert(p,5)`		
`q=begin()`		
`set(q,1)`		
`eraseFront()`		
`set(q,1)`		

Especificación del TAD lista

Ejemplos (lista inicialmente vacía):

Operación	**Salida**	**Lista**
`insertFront(8)`	-	(8)
`p=begin()`	p: (8)	(8)
`p.step()`	p: (-)	(8)
`insert(p,5)`	-	(8,5)
`q=begin()`		
`set(q,1)`		
`eraseFront()`		
`set(q,1)`		

Especificación del TAD lista

Ejemplos (lista inicialmente vacía):

Operación	**Salida**	**Lista**
`insertFront(8)`	-	(8)
`p=begin()`	p: (8)	(8)
`p.step()`	p: (-)	(8)
`insert(p,5)`	-	(8,5)
`q=begin()`	q: (8)	(8,5)
`set(q,1)`		
`eraseFront()`		
`set(q,1)`		

Especificación del TAD lista

Ejemplos (lista inicialmente vacía):

Operación	**Salida**	**Lista**
`insertFront(8)`	-	(8)
`p=begin()`	p: (8)	(8)
`p.step()`	p: (-)	(8)
`insert(p,5)`	-	(8,5)
`q=begin()`	q: (8)	(8,5)
`set(q,1)`	true	(1,5)
`eraseFront()`		
`set(q,1)`		

Especificación del TAD lista

Ejemplos (lista inicialmente vacía):

Operación	Salida	Lista
`insertFront(8)`	-	(8)
`p=begin()`	p: (8)	(8)
`p.step()`	p: (-)	(8)
`insert(p,5)`	-	(8,5)
`q=begin()`	q: (8)	(8,5)
`set(q,1)`	true	(1,5)
`eraseFront()`	true	(5)
`set(q,1)`		

Especificación del TAD lista

Ejemplos (lista inicialmente vacía):

Operación	**Salida**	**Lista**
`insertFront(8)`	-	(8)
`p=begin()`	p: (8)	(8)
`p.step()`	p: (-)	(8)
`insert(p,5)`	-	(8,5)
`q=begin()`	q: (8)	(8,5)
`set(q,1)`	true	(1,5)
`eraseFront()`	true	(5)
`set(q,1)`	“error”	(5)

Especificación del TAD lista

Ejemplos

- Acceso a todos los elementos de la lista empleando un iterador

```
Lista l;
//Inserción de elementos en la lista
//..
for(Iterador it=l.begin(); it != l.end(); it.step())
    cout << l.get(it) << endl;
```

Implementación del TAD lista

- La implementación más natural para el TAD lista es una lista enlazada
- La complejidad de todas las operaciones es

TAD lista

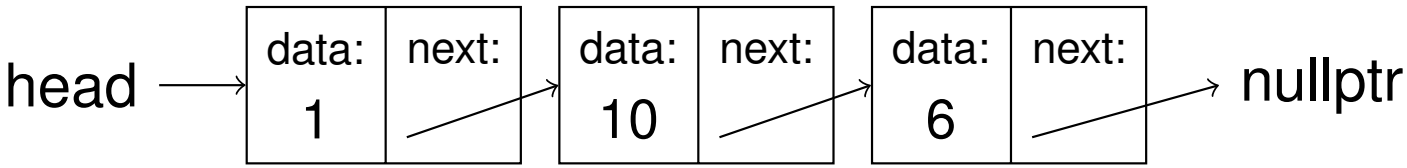
Implementación del TAD lista

- La implementación más natural para el TAD lista es una lista enlazada
- La complejidad de todas las operaciones es $\Theta(1)$
- El iterador se implementa como un puntero a nodo
- Pero la necesidad de incrementar el iterador hace que muchas operaciones tengan coste lineal en la práctica
- El TAD lista también puede implementarse mediante un array

Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

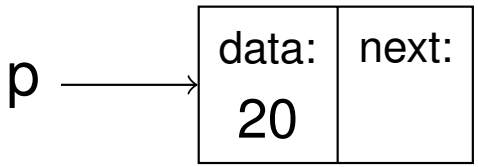
- Inserción al comienzo de la lista

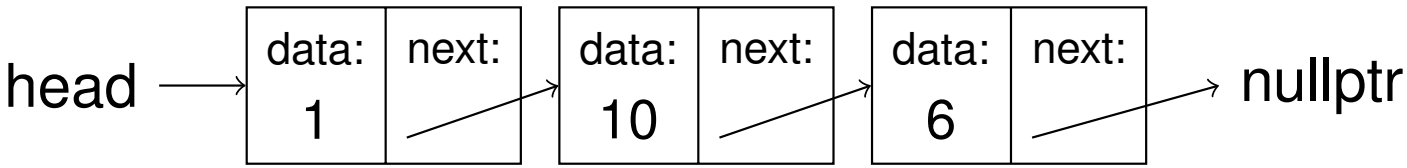


Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

- Inserción al comienzo de la lista

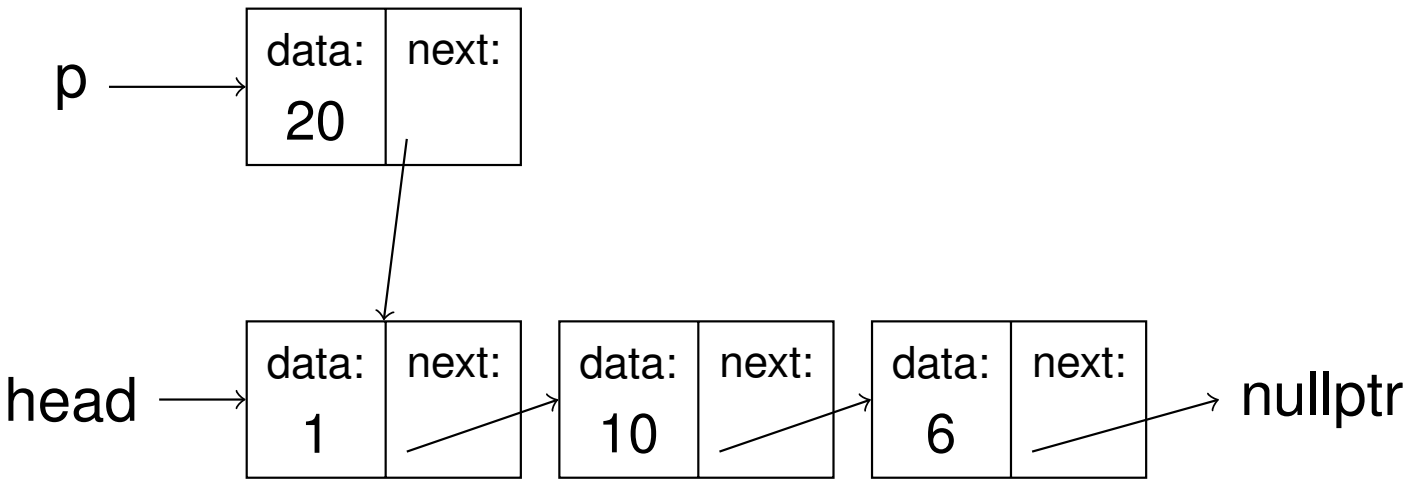




Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

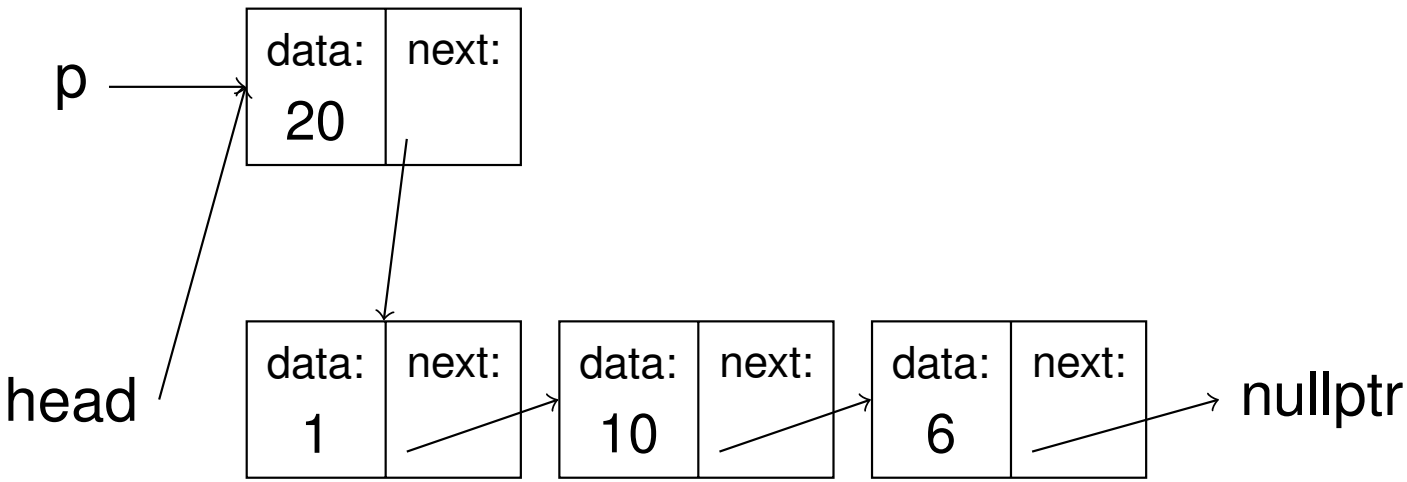
- Inserción al comienzo de la lista



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

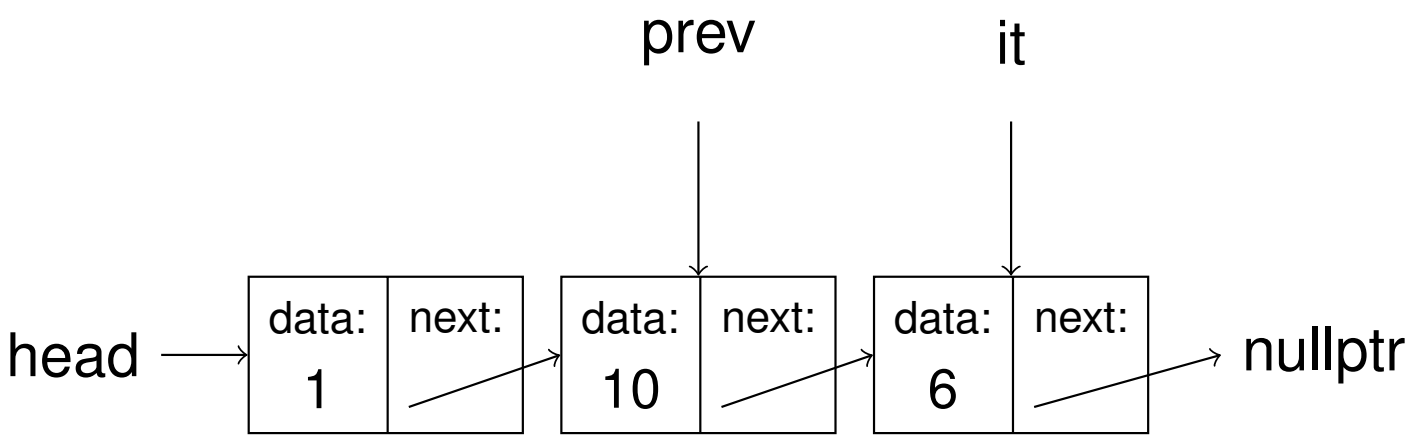
- Inserción al comienzo de la lista



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

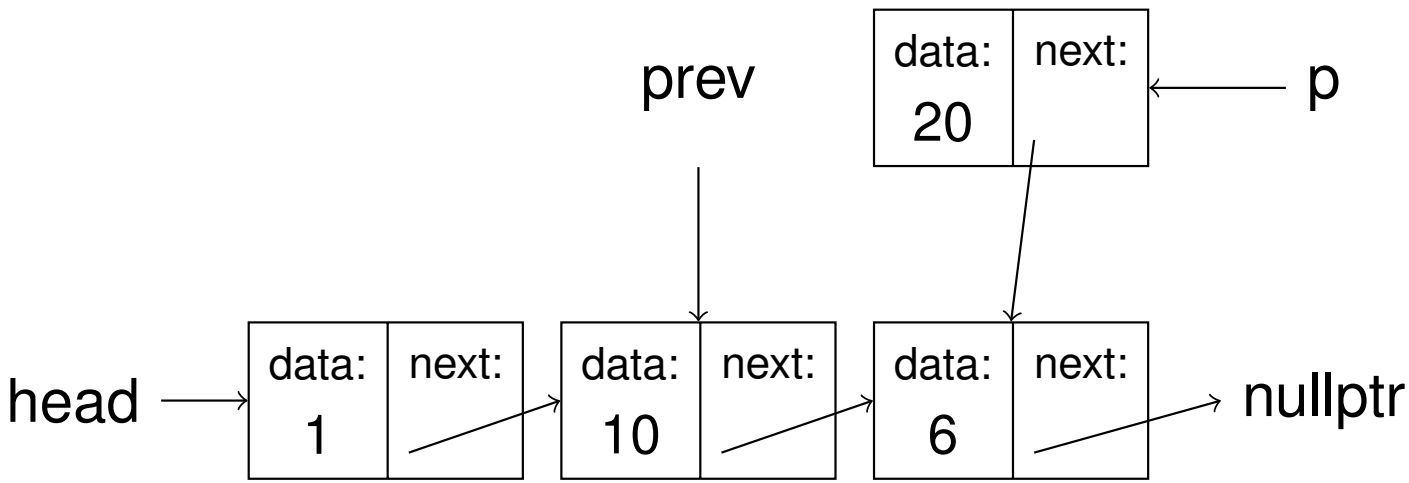
- Inserción en otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a donde vamos a insertar



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

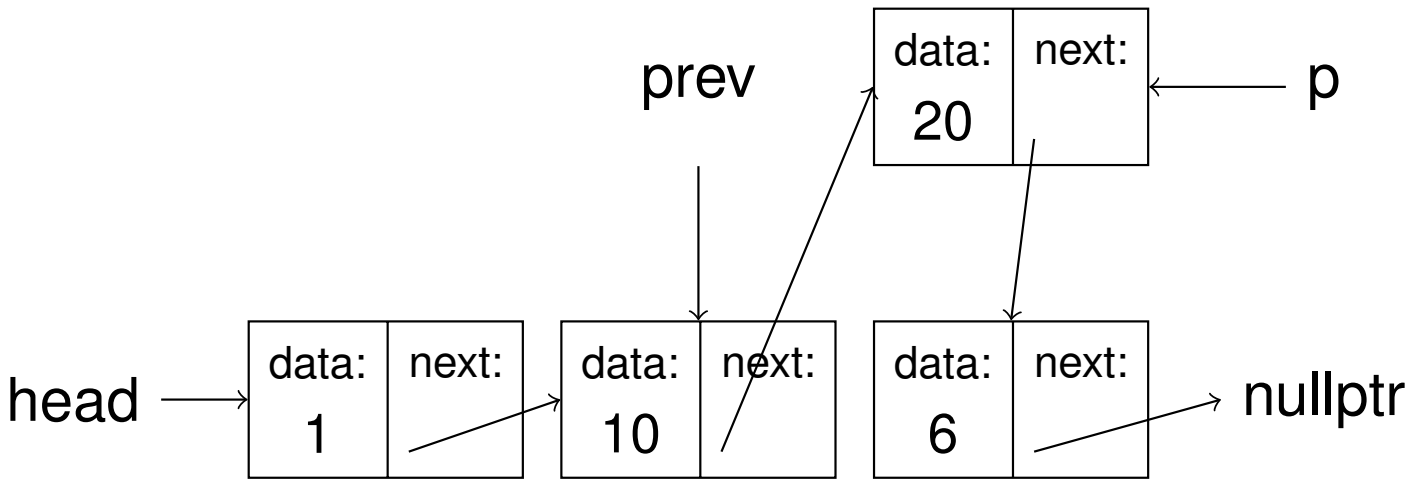
- Inserción en otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a donde vamos a insertar



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

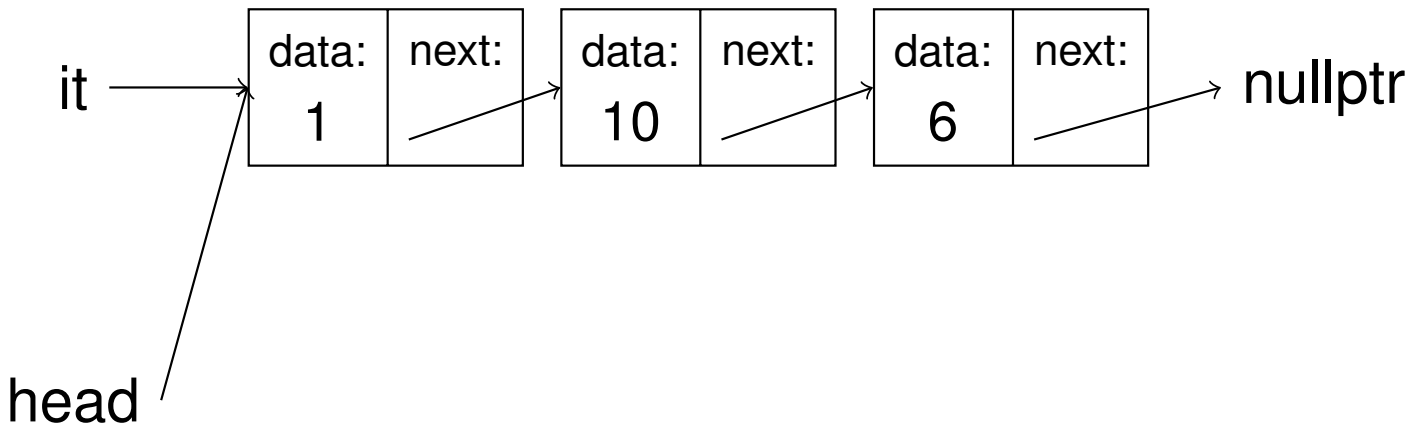
- Inserción en otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a donde vamos a insertar



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

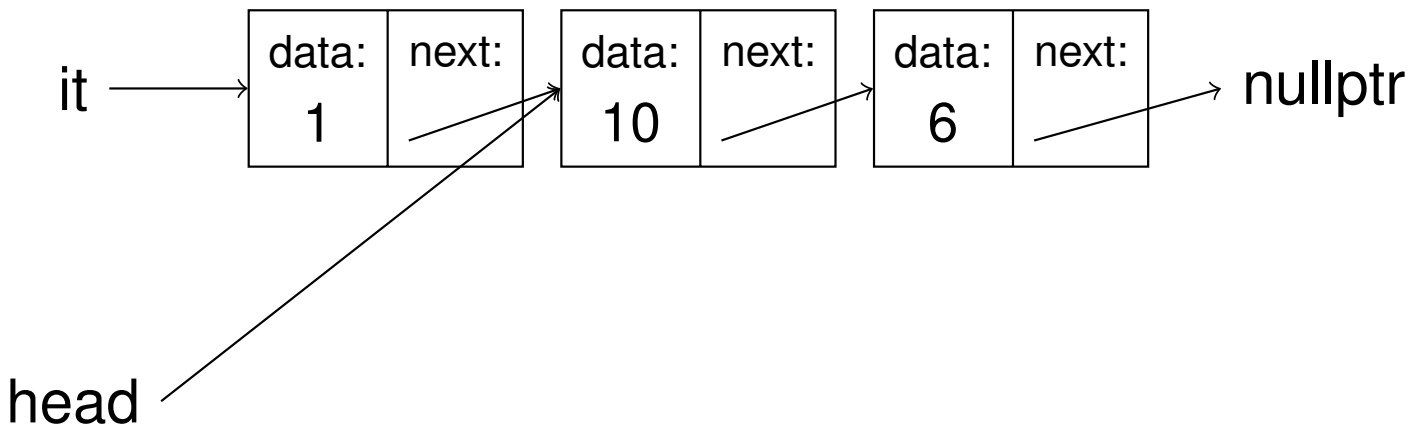
- Borrado de la primera posición



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

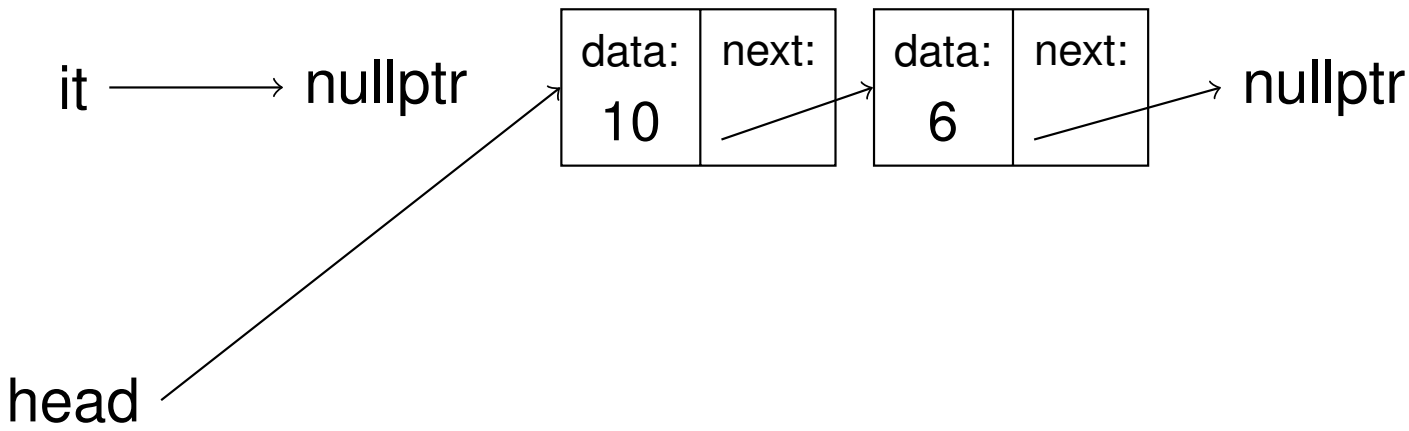
- Borrado de la primera posición



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

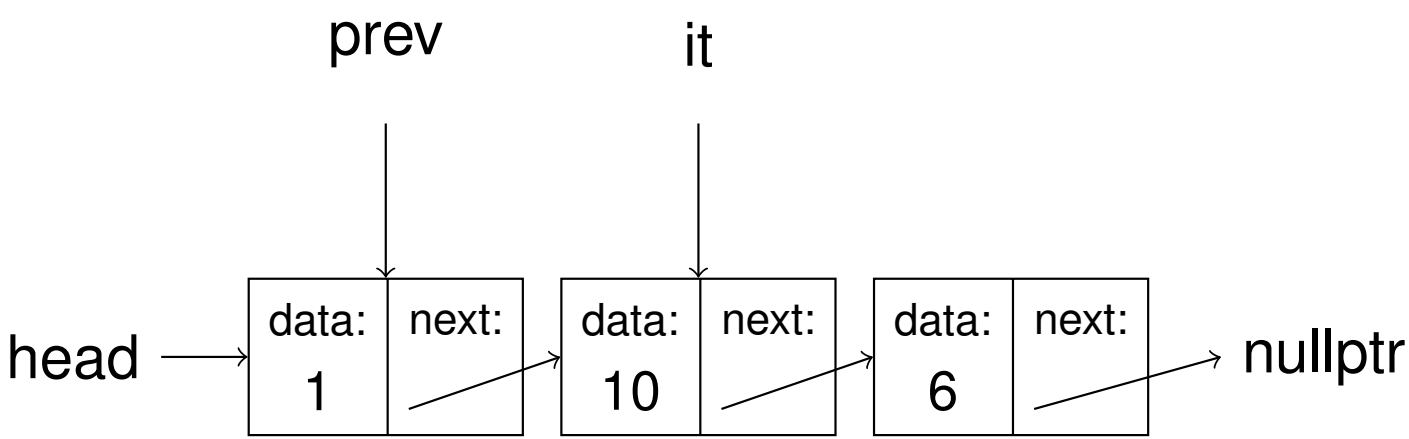
- Borrado de la primera posición



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

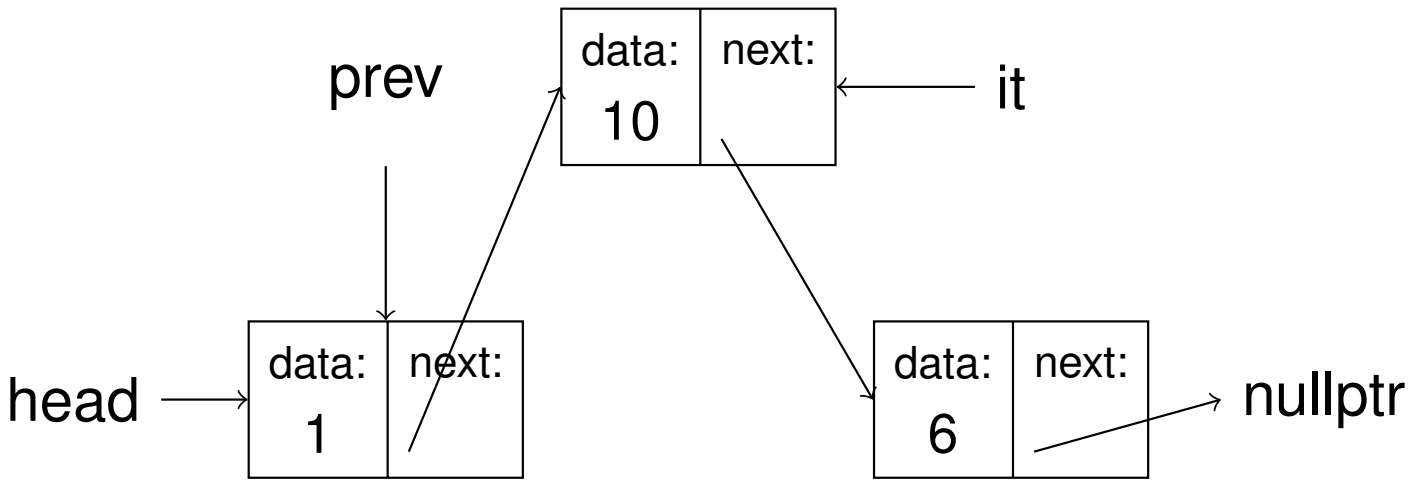
- Borrado de otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a la que vamos a borrar



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

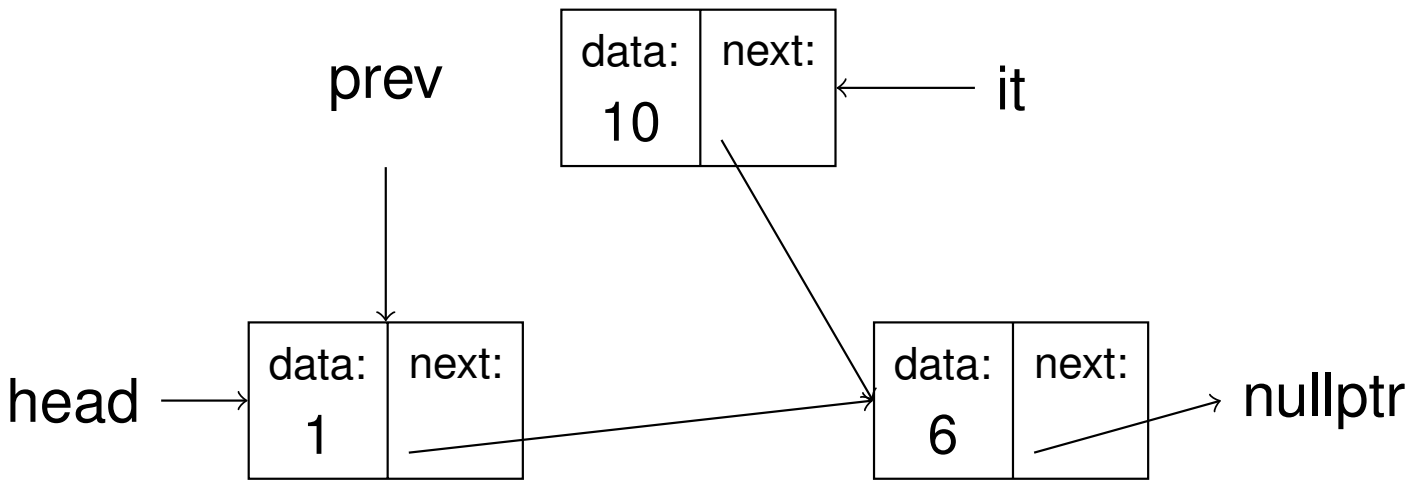
- Borrado de otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a la que vamos a borrar



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

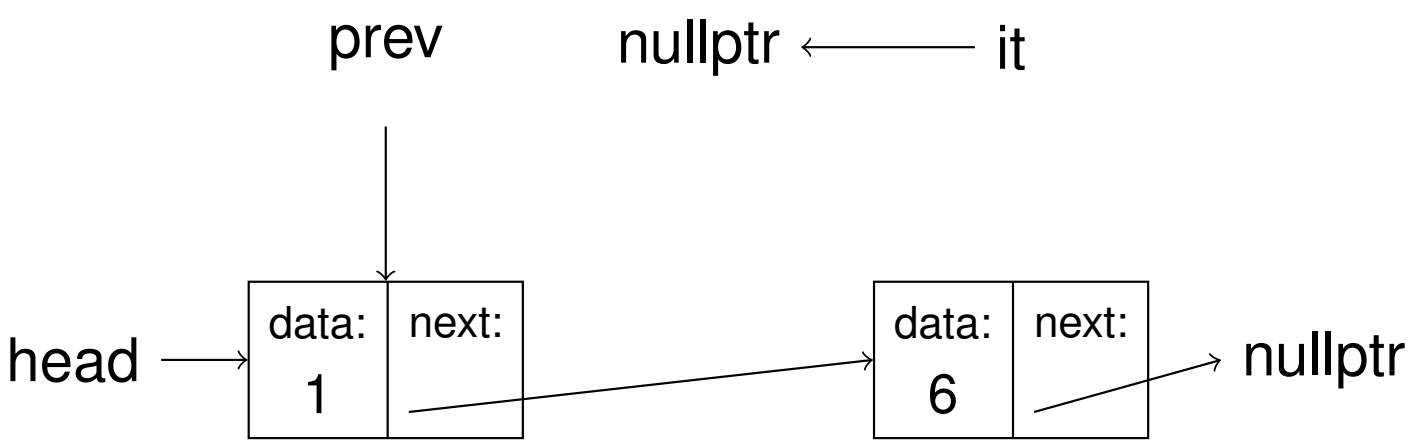
- Borrado de otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a la que vamos a borrar



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

- Borrado de otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a la que vamos a borrar



Implementación del TAD lista

Implementación enlazada: ejemplos de gestión de punteros en inserciones y borrados

- Para que las operaciones se puedan realizar en $\Theta(1)$ es necesario obtener de manera rápida el puntero al nodo anterior
- Iterar desde el principio de la lista no es una opción
- **Solución** $\rightarrow$ listas **doblemente enlazadas**: cada nodo tiene un puntero al nodo anterior y al nodo siguiente
- Las listas doblemente enlazadas también permiten iterar desde la última posición a la primera

Listas en C++ STL

- Están implementadas como listas de nodos doblemente enlazadas
- Se declaran con `list<TipoDeDato>`:

```
#include <list>
#include <iostream>
using namespace std;
...
//Constructor por defecto: lista vacía
list<int> l;

l.push_front(10); //Inserta al comienzo de la lista
l.push_front(8);
l.push_back(6); // Inserta al final de la lista

cout << l.front(); // Imprime 8
cout << l.back(); // Imprime 6

//Imprime 8 10 6
for(list<int>::const_iterator it=l.begin(); it != l.end(); ++it)
    cout << *it << " ";
```

Listas en C++ STL

- Están implementadas como listas de nodos doblemente enlazadas
- Se declaran con `list<TipoDeDato>`:

```
//La lista contiene 8 10 6
l.pop_front(); //Elimina el primer elemento: 10 6
l.pop_back(); //Elimina el último elemento: 10
cout << l.front(); // Imprime 10
cout << l.size(); // Imprime 1

l.push_back(30);
l.push_front(20); //20 10 30

//Suma 2 a todos los elementos
for(list<int>::iterator it=l.begin(); it != l.end(); ++it)
    *it=*it + 2;

//Imprime 22 12 32
for(list<int>::const_iterator it=l.begin(); it != l.end(); ++it)
    cout << *it << " ";
```

Listas en C++ STL

- Están implementadas como listas de nodos doblemente enlazadas
- Se declaran con `list<TipoDeDato>`:

```
//La lista contiene 22 12 32

//Inserta un elemento en la segunda posición
list<int>::iterator it=l.begin();
it++;
l.insert(it,15);

//Imprime 22 15 12 32
for(list<int>::const_iterator it=l.begin(); it != l.end(); ++it)
    cout << *it << " ";

//Elimina el segundo elemento
it=l.begin(); it++:
l.erase(it);

//Imprime 22 12 32
for(list<int>::const_iterator it=l.begin(); it != l.end(); ++it)
    cout << *it << " ";
```

El TAD lista

- Las listas enlazadas son útiles cuando:
  - No se necesita el acceso a cualquier elemento en tiempo constante: no se va a emplear búsqueda binaria
  - No conocemos de antemano el número de elementos a almacenar
  - No podemos permitirnos el coste $O(n)$ de los redimensionamientos del vector
  - Vamos a hacer muchas inserciones y borrados en posiciones diferentes de la primera y la última