Programación Avanzada y Estructuras de Datos

1. Introducción a C++

Víctor M. Sánchez Cartagena

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

25 de septiembre de 2024

Índice

Características del lenguaje C++ (1)

- Lenguaje compilado (Python es interpretado)
- Lenguaje de tipo imperativo y orientado a objetos (igual que Python)
  - Lenguaje imperativo: los programas son conjuntos de instrucciones que indican al computador cómo realizar una tarea
  - Lenguaje declarativo: los programas indican cuál es el resultado deseado de la tarea
- Lenguaje de **tipado fuerte** (Python es un lenguaje no tipado)

Características del lenguaje C++ (y 2)

- Desarrollado en 1980 por Bjarne Stroustroup en los laboratorios At&T
- Extensión orientada a objetos del lenguaje C: creado por Dennis Ritchie entre los años 1970-73
- Código fuente escrito en C puede compilarse como C++ (normalmente): se mantienen algunos inconvenientes del lenguaje C
- Usaremos principalmente C++11

Estructura básica de un programa en C++

```
#include <ficheros de cabecera estándar>
...
#include "ficheros de cabecera propios"
...
using namespace std; // Permite usar cout, string...
...
const ... // Constantes
...
// Variables globales: ¡¡No la usaremos en PAED!!
...
funciones ... // Declaración de funciones
...
int main() { // Función principal
...
}
```

Mi primer programa

```cpp
#include <iostream>

using namespace std;

int main() { // Función principal
    cout << "¡Hola, mundo!" << endl;
    return 0;
}
```

Compilación

- El compilador permite convertir un código fuente en un código objeto
- Usaremos el compilador GNU C++ para transformar el código fuente en C++ en un programa ejecutable
- El compilador de GNU se invoca con el programa `g++` y admite numerosos argumentos
  - `-Wall`: muestra todas as advertencias o *warnings*
  - `-g`: añade información para el depurador
  - `-o`: para indicar el nombre del ejecutable
  - `-std=c++11`: para usar el estándar C++11
  - `--version`: muestra la versión instalada

Ejemplo de uso (terminal)

```
g++ -std=c++11 -Wall -g prog.cc -o prog
```

Índice

Elementos del lenguaje

- Los identificadores son nombres de variables, constantes y funciones
- Han de comenzar por letra minúscula, mayúscula o guión bajo
- C++ distingue entre letras mayúsculas y minúsculas
- En C++ hay palabras reservadas que no se pueden utilizar como nombres definidos por el usuario

```
if while for do int friend long auto public union
...
```

  - Si las usamos como identificadores se producirá un error de compilación difícil de interpretar

Variables: definición y tipos

- Las variables permiten almacenar diferentes tipos de datos
- **Se deben declarar explícitamente**, indicando el **tipo**
- Tipos básicos o primitivos:

Tipo	Tamaño en bits CPU 32 bits	Uso
`int`	32	Número entero: `-273`, `1066`
`char`	8	Carácter: `'a'`, `'z'`, `'9'`
`float`	32	Número en coma flotante: `273.15`, `3.14`
`double`	64	Ídem de doble precisión
`bool`	8	Booleano: `true` o `false`
`void`	-	Funciones que no devuelven nada

- Se puede usar `unsigned` con int para tener solo números

Ejemplos

```
a=2
b=3
c=a+b
print(c)
```

```
int a,b,c;
a=2; b=3;
c=a+b;
cout << c << endl;
```

- ¿Qué pasa si empleamos una variable declarada sin haberle asignado valor?

```
int c;
cout << c << endl;
```

Inicialización de variables

- Las variables no inicializadas pueden tener *cualquier* valor, según del compilador, la máquina, y el estado de la memoria
- Errores en tiempo de ejecución y suspensos
- Es recomendable inicializar las variables a la vez que se declaran:

```
int c=0;
cout << c << endl;
```

- `auto`: el tipo se determina automáticamente a partir del valor (como en Python)

```
auto c=0;
cout << c << endl;
```

Ámbito

- Ámbito de una variable: parte del programa donde se puede acceder a esa variable
- Siempre dentro del bloque entre llaves donde se declaró:

```
int i=0;
int numCajas=0;
if(i<10){
    // numCajas se puede usar aquí
    int numCajas=100; // Mismo nombre pero otro ámbito
    cout << numCajas << endl; // Imprime 100
}
cout << numCajas << endl; // ¿Qué imprime? ¿0 o 100?
```

Variables locales y globales

- Variable local a una función: se declara dentro de la función y no es accesible desde fuera

```
void imprimir(){
  int i=3,j=5; // Al principio de la función
  cout << i << j << endl;
  ...
  int k=7; // En un punto intermedio
  cout << k << endl;
}
```

- Variable global: se declara fuera de cualquier función y es accesible desde cualquier parte del programa
  - Son muy peligrosas y dan lugar a programas difíciles de depurar
  - Rompen el encapsulamiento y dificultan la reutilización del código
  - No las usaremos en Programación Avanzada y Estructuras de Datos
  - Se puede conseguir una funcionalidad parecida con atributos estáticos o de clase en programación orientada a objetos

Constantes

- Tienen un valor fijo (no puede ser cambiado) durante toda la ejecución del programa
- Se declaran anteponiendo const al tipo de dato
- Son útiles para definir valores que se usen en múltiples puntos de un programa y que no cambien de valor

```
const int MAXALUMNOS=600;
const double PI=3.141592;
```

Conversión de tipos

- Implícita: la hace el compilador de manera automática, normalmente cuando no hay pérdida de información al convertir

Tipos	Ejemplo
`char→int`	`int a='A'+2; // a vale 67`
`int→float`	`float pi=1+2.141592;`
`float→double`	`double piMedios=pi/2.0;`
`bool→int`	`int b=true; // b vale 1`
`int→bool`	`bool c=77212; // c vale true`

- Explícita o *cast*: se pone el tipo deseado entre paréntesis

```
char laC=(char)('A'+2); // laC vale 'C'
int pEnteraPi=(int)pi; // pEnteraPi vale 3
```

Índice

Expresiones aritméticas

- Las expresiones aritméticas están formadas por operandos (`int`, `float` y `double`) y operadores aritméticos (`+ - * /`):

```
float i=4*5.7+3; // i vale 25.8
```

- Si aparece un operando de tipo `char` o `bool` se convierte a entero implícitamente:

```
int i=2+'a'; // i vale 99
```

- Si dividimos dos enteros el resultado es un entero:

```
cout << 7/2; // La salida es 3
```

- Si queremos que el resultado de la división entera sea un valor real hay que hacer un cast a `float` o `double`:

```
cout << (float)7/2; // La salida es 3.5
cout << (float)(7/2); // La salida es ...
```

Expresiones aritméticas

- El operador % devuelve el resto de la división entera:

```
cout << 30%7; // La salida es 2
```

- Precedencia de operadores:
  1. ++ (incremento) −− (decremento) ! (negación) − (menos unario)
  2. ⋆ (multiplicación) / (división) % (módulo)
  3. + (suma) − (resta)
- En caso de duda usar paréntesis:

```
cout << 2+3*4; // La salida es 14
// * tiene más precedencia que +
cout << 2+(3*4); // La salida es 14
cout << (2+3)*4; // La salida es ..
cout << 2+3*4%2-1/2; // La salida es ..
```

Operadores de incremento y decremento

- Los operadores ++ y −− se usan para incrementar o decrementar el valor de una variable entera en una unidad
- Preincremento/predecremento: se incrementa/decrementa antes de tomar su valor

```
int i=3,j=3;
int k=++i; // k vale 4, i vale 4
int l=--j; // l vale 2, j vale 2
```

- Postincremento/postdecremento: se incrementa/decrementa después de tomar su valor

```
int i=3,j=3;
int k=i++; // k vale 3, i vale 4
int l=j--; // l vale 3, j vale 2
```

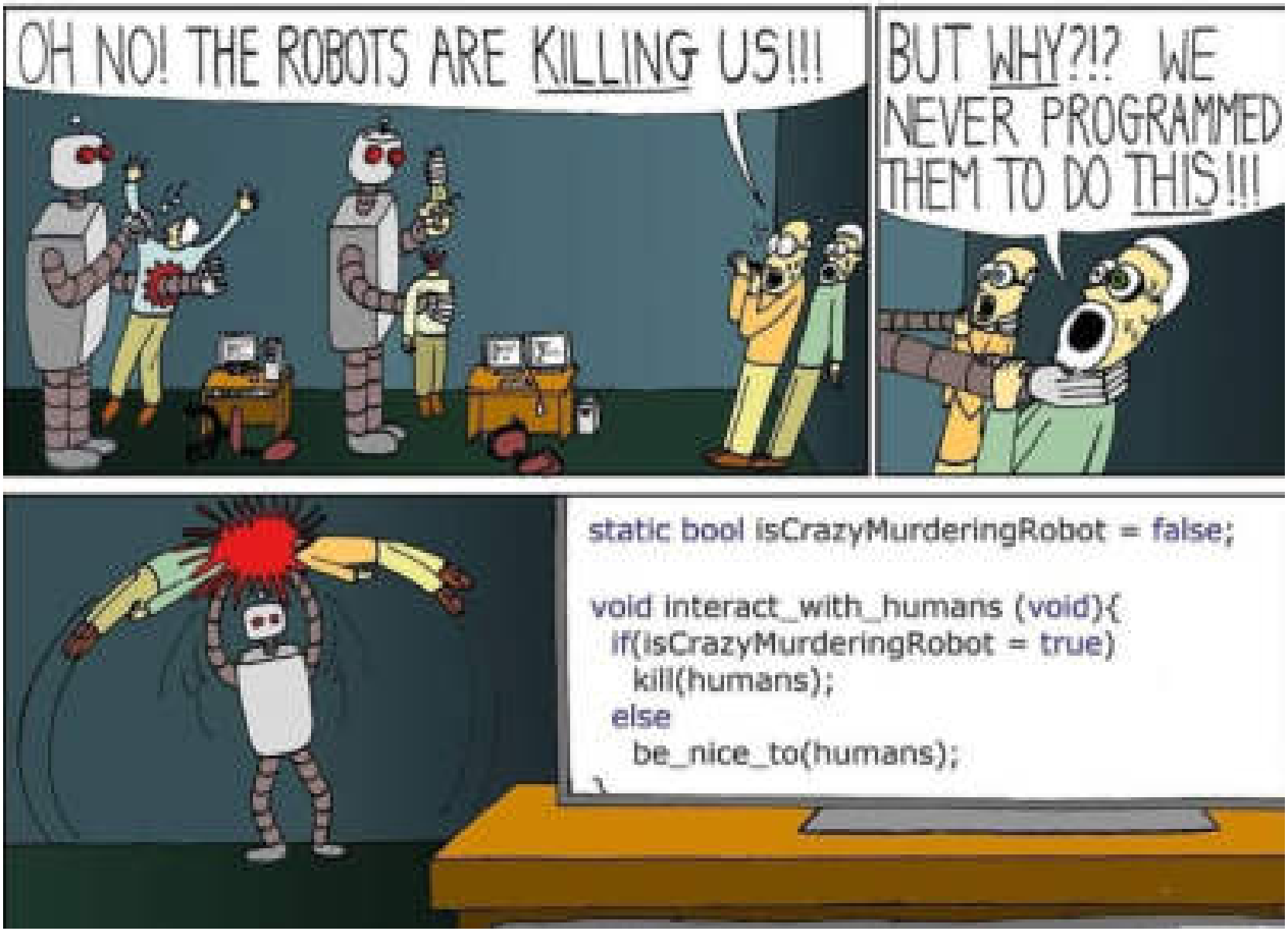
Expresiones relacionales

- Las expresiones relacionales permiten realizar comparaciones entre valores
- Operadores: == (igual), != (distinto), >= (mayor o igual), > (mayor estricto), <= (menor o igual) y < (menor estricto)
- Si los tipos de los operandos no son iguales se convierten (implícitamente) al tipo más general:

```
if(2<3.4){...} // Se transforma en: if(2.0<3.4)
```

- El resultado es 0 si la comparación es falsa y distinto de 0 si es cierta
- **Las expresiones relacionales tienen menor precedencia que las aritméticas**

Expresiones relacionales



Expresiones lógicas

- Las expresiones lógicas permiten relacionar valores booleanos y obtener un nuevo valor booleano
- Operadores: `!` (negación), `&&` (y lógico) y `||` (o lógico)
- Precedencia: `!` > aritméticas > relacionales > `&&` > `||`

```
if(a || b && c){...} // Equivale a: if(a || (b && c))
```

- Evaluación en cortocircuito:
  - Si el operando izquierdo de `&&` es falso, el operando derecho no se evalúa (`false && loquesea` es siempre `false`)
  - Si el operando izquierdo de `||` es cierto, el operando derecho no se evalúa (`true || loquesea` es siempre `true`)

Entrada y salida

- Salida por pantalla con `cout`:

```
int i=7;
cout << i << endl; // Muestra 7 y salto de línea (endl)
```

- Salida de error (por pantalla) con `cerr`:

```
int i=7;
cerr << i << endl; // Muestra 7 y salto de línea (endl)
```

- Entrada por teclado con `cin`:

```
int i;
cin >> i; // Guarda en i un número escrito por teclado
```

Índice

Condicional: `if`

- `if` evalúa una condición y toma un camino u otro
- A diferencia de Python, cada camino es un bloque entre llaves ({ y }), independientemente de cómo esté indentado el texto

```
int num=0;
cin >> num; // Leemos un número por teclado
if(num<5){ // Si num es menor que 5 ejecuta esta parte
  cout << "Camino que se toma si se cumple la condición";
  cout << "El número es menor que cinco";
}
else{ // Si no, ejecuta esta otra
  cout << "Camino que se toma si no se cumple la condición";
  cout << "El número es mayor o igual que cinco";
}
```

Condicional: `if`

- Si no hay llaves,
  sólo se considera que la primera instrucción está dentro del bloque

```
int num=0;
cin >> num; // Leemos un número por teclado
if(num<5) // Si num es menor que 5 sólo ejecuta la primera instrucció
  cout << "Camino que se toma si se cumple la condición";
  cout << "Esta línea se ejecuta siempre";
```

Iteraciones: `while` y `do-while`

- `while` ejecuta instrucciones mientras se cumpla la condición:

```
int i=10;
while(i>=0){
  cout << i << endl; // Hará una cuenta atrás del 10 a 0
  i--; // Si no decrementamos entra en un bucle infinito
}
```

- `do-while` ejecuta el cuerpo del bloque al menos una vez

```
int i=0;
do{ // Muestra el valor de i al menos una vez
  cout << "i vale: " << i << endl;
  i++;
}while(i<10);
```

Iteraciones: `for`

- `for` equivale a un `while`:

```
for(inicialización;condición;finalización){
// Instrucciones
}
```

```
inicialización;
while(condición){
  // Instrucciones
  finalización;
}
```

Iteraciones: `for`

- Tiene una sintaxis más elegante y compacta que while para iterar sobre valores numéricos
- Es una versión de bajo nivel de la estructura `for i in range(...)` de Python

```
for(int i=10;i>=0;i--){
cout << i << endl; // Hará una cuenta atrás del 10 al 0
}
```

Ejercicio

Implementa con un bucle `for`:

- Una cuenta adelante del 0 al 10
- Una cuenta adelante del 0 al 30, incrementando de 2 en 2
- Una cuenta atrás del 30 al 0, decrementando de 2 en 2

Control de flujo: `switch`

- `switch` permite ejecutar un fragmento de código u otro según el valor de una variable
- Es una especie de *`if` múltiple*
- Cuidado: cada bloque no va delimitado entre llaves, sino que acaba con la palabra clave `break`

```
char opcion;
cin >> opcion; // Leemos un carácter de teclado
switch(opcion){
  case 'a': cout << "Opción A" << endl;
  break; // Sale del switch
  case 'b': cout << "Opción B" << endl;
  break;
  case 'c': cout << "Opción C" << endl;
  break;
  default: cout << "Otra opción" << endl;
}
```

Control de flujo: `break`

- La instrucción `break` también se puede utilizar para salir de cualquier bucle

```
int vec[]={1,2,5,7,6,12,3,4,9};
int i=0;
// Código equivalente sin usar break
bool encontrado=false;
while(i<9 && !encontrado){
  if(vec[i]==6)
    encontrado=true;
  else
    i++;
}
```

Control de flujo: `break`

- La instrucción `break` también se puede utilizar para salir de cualquier bucle

```
int vec[]={1,2,5,7,6,12,3,4,9};
int i=0;
// Código equivalente sin usar break
bool encontrado=false;
while(i<9 && !encontrado){
  if(vec[i]==6)
    encontrado=true;
  else
    i++;
}
```

```
int vec[]={1,2,5,7,6,12,3,4,9};
int i=0;
// Salimos del bucle al encontrar el 6 en vec
while(i<9){
  if(vec[i]==6)
    break;
  else
    i++;
}
```

Ejercicios (I)

- Escribe un programa que pida el valor de los dos lados de un rectángulo y muestre el valor de su perímetro y de su área
- Escribe un programa que pida al usuario tres números enteros e imprima el valor del máximo
- Escribe un programa que pida al usuario cinco números enteros e imprima el valor del máximo
- Escribe un programa que pida al usuario un número y compruebe si es primo
- Escribe un programa que imprima todos los números primos en un rango determinado. El primer y último elemento del rango serán pedidos por teclado al usuario

Ejercicios (II)

- Escribe un programa que pida números por teclado hasta que se introduzca un 0, y al final imprima la media de los números introducidos
- Escribe un programa que imprima la siguiente estructura. El número de líneas se leerá de entrada estándar (en el ejemplo el valor es 4):

```
*
**
***
****
```

- Ahora la estructura debe ser esta (en el ejemplo el valor es 4):

```
   *
  **
 ***
****
```

Ejercicios (III)

- Escribe un programa que imprima la siguiente estructura. El número de líneas se leerá de entrada estándar (en el ejemplo el valor es 4):

```
****
 ***
  **
   *
```

- Realiza un programa que imprima el desglose en billetes y monedas de una cantidad entera de euros, utilizando el menor número posible de monedas y billetes. Existen billetes de 500, 200, 100, 50, 20, 10, y 5 euros, y monedas de 2 y 1 euro. El programa pedirá primero al usuario una cantidad, y después imprimirá su desglose, como muestra el siguiente ejemplo:

```
Introduce la cantidad: 434
2 de 200; 1 de 20; 1 de 10; 2 de 2;
```

Índice

Arrays

- Los *arrays* almacenan múltiples valores en una única variable en posiciones de memoria contiguas
- Estos valores pueden ser de cualquier tipo que deseemos, incluso tipos de datos propios (lo veremos más adelante)
- Al declarar un array hay que especificar su tamaño (cuántos elementos almacena) mediante constantes o variables

```
// Tamaño definido mediante constantes
const int MAXALUMNOS=100;
int alumnos[MAXALUMNOS]; // Puede almacenar 100 enteros
bool gruposLlenos[5]; // Puede almacenar 5 booleanos

// Tamaño definido mediante variables
int numElementos;
cin >> numElementos; // No sabemos qué número introducirá
float listaNotas[numElementos];
```

Arrays

- Cuando se inicializa un vector al declararlo no hace falta indicar su tamaño:

```
int numbers[]={1,3,5,2,5,6,1,2};
```

- Si un vector tiene tamaño `TAM`, el primer elemento se halla en la posición `0` y el último en la posición `TAM-1`
- Asignación y acceso a valores mediante el operador `[]`:

```
const int TAM=10;
int vec[TAM];
vec[0]=7;
vec[TAM-1]=vec[TAM-2]+1; // vec[9]=vec[8]+1;
```

- Podemos (o no) tener un fallo en tiempo de ejecución si intentamos leer o escribir en un elemento fuera del vector:

```
int vec[5];
vec[5]=7; // Puede haber fallo en tiempo de ejecución
// El último elemento válido está en vec[4]
```

Matrices

- Una matriz es un vector cuyas posiciones son, cada una de ellas, otro vector
- Hay que dar tamaño a sus dos dimensiones (filas y columnas):

```
const int TAM=10;
char tablero[TAM][TAM]; // Matriz de 10 x 10 elementos
int tabla[5][8]; // Matriz de 5 x 8 elementos
```

- Como los vectores, comienzan en `0` y acaban en `TAM-1`
- Asignación y acceso a valores mediante el operador []:

```
int matriz[8][10];
matriz[2][3]=7; // Hay que indicar fila y columna
```

Ejercicios

- Realiza un programa que imprima el desglose en billetes y monedas de una cantidad entera de euros, utilizando el menor número posible de billetes y monedas. Los valores de billetes y monedas disponibles son: 500, 200, 100, 50, 20, 10, 5, 2, 1. Intente que el código sea lo más compacto posible.
- Escribe un programa que nos diga si los elementos de un array están ordenados. Asume que el array ya tiene datos y su tamaño está almacenado en la constante `TAM`
- Escribe un programa que determine si una matriz cuadrada es triangular superior, es decir, todos los elementos por debajo de la diagonal principal son 0 (como la de abajo). Asume que la matriz ya tiene datos y su tamaño está almacenado en la constante `TAM`

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 12 & 6 \\ 0 & 0 & -3 \end{pmatrix}$$

Índice

Funciones

- Una función es un bloque de código que realizan una tarea
- Permite agrupar operaciones comunes en un bloque reutilizable
- Puede opcionalmente tener parámetros de entrada y devolver un valor como salida

```
tipoRetorno nombreFuncion(parametro1,parametro2,...){
  tipoRetorno ret;
  instruccion1;
  instruccion2;
  ...
  return ret;
}
```

Funciones

- Se puede utilizar más de un `return` en el cuerpo de una función si eso simplifica el código

```
bool buscar(int vec[],int n){ // Dos parámetros
  bool encontrado=false;
  for(int i=0;i<TAM && !encontrado;i++){
    if(vec[i]==n)
       encontrado=true;
  }
  return encontrado; // Un único return
}
```

```
bool buscar(int vec[], int n){
  for(int i=0;i<TAM;i++){
    if(vec[i]==n)
      return true; // Primer return
  }
  return false; // Segundo return
}
```

Paso de parámetros

- Se permite paso de parámetros por valor o por referencia (con &)
  - Por valor: se hace una copia de la variable → las modificaciones en el interior de la función no son visibles fuera
  - Por referencia: no se hace copia → las modificaciones en el interior de la función SÍ son visibles fuera

```
// a y b se pasan por valor, c por referencia
void funcion(int a,int b,bool &c){
  c=a<b; // c mantiene este valor al acabar la función
}
```

- Es recomendable pasar por referencia variables muy grandes cuya copia puede ralentizar el programa
  - Con el modificador `const` indicamos que la variable no se va a modificar dentro de la función (para evitar errores inesperados)

```
void funcion(const string &s){
  // El compilador no hace copia de s, pero si
  // intentamos modificarlo nos da un error
}
```

Pasando arrays y matrices

- Los arrays y matrices sólo se pueden pasar por referencia
- Hay que indicarlo con `[]` en la declaración de la función, omitiendo el tamaño en la primera dimensión

```
void sumar(int v[],int m[][TAM]){
  // En m no se pone el tamaño de la primera dimensión
  ...
}
...
// No se ponen corchetes en la llamada a la función
sumar(v,m);
```

Prototipos

- A veces es necesario utilizar una función antes de que aparezca su código
- En esos casos se debe poner el prototipo de la función

```
void miFuncion(bool,char,double[]); // Prototipo

char otraFuncion(){
  double vr[20];
  // Todavía no se ha declarado miFuncion
  // pero podemos usarla gracias al prototipo
  miFuncion(true,'a',vr);
}

// Declaración de la función
void miFuncion(bool exist,char opt,double vec[]){
  ...
}
```

Ejercicios

- Escribe un programa que imprima todos los números primos en un rango determinado. El primer y último elemento del rango serán pedidos por teclado al usuario. Emplea funciones para simplificar el código del programa.

Índice

Arrays de caracteres

- Para representar cadenas de caracteres, podemos emplear arrays de tipo `char`
- Desventajas:
  - No podemos cambiar el tamaño de la cadena
  - No podemos saber fácilmente su longitud
  - Cualquier operación implica iterar elemento a elemento

```
const int TAM=4;
char cadena[TAM];
cadena[0]='h';cadena[1]='o';cadena[2]='l';cadena[3]='a';

for(int i=0; i< TAM; i++)
   cout<<cadena[i];
cout << endl;
```

Cadenas de caracteres de C

- Último carácter de cada cadena sea `'\0'`
- Fácil manipulación con funciones de `#include<string.h>`

```
#include<string.h>
char cadena[]="hola";
//Las líneas de abajo son equivalentes a la anterior
char cadena[5];
cadena[0]='h';cadena[1]='o';
cadena[2]='l';cadena[3]='a';cadena[4]='\0';

cout<<cadena<<endl;
//Imprime: La longitud es 4
cout << "La longitud es" << strlen(cadena) << endl;
```

- Pero todavía tenemos el problema del tamaño fijo
- No las usaremos en la asignatura

La clase `string`

- Representa una cadena de tamaño variable
- Se usa igual que un tipo básico
- No es necesario preocuparse del `'\0'`

```
string vacia; //Cadena vacía por defecto
string s="hola"; // Almacena 4 caracteres
s="hola a todo el mundo"; // Almacena 20 caracteres*
s="ok"; // Almacena 2 caracteres
```

- Salida por pantalla con cout y cerr:

```
string s="Nota";
int num=10;
cout << s << " -> " << num; // Muestra "Nota -> 10"
```

La clase `string`

- Lectura de entrada estándar con operador >>: lee hasta el primer espacio en blanco

```
string s;
cin >> s;
// El usuario escribe "hola"
// La variable s almacena "hola"
...
// El usuario escribe "buenas tardes"
// La variable s almacena "buenas"
```

- Con `getline`: lee hasta el primer salto de línea

```
string s;
getline(cin,s);
// Si el usuario introduce "buenas tardes"
// en s se almacena "buenas tardes"
```

Métodos de `string`

- `length` devuelve el número de caracteres de la cadena:

```
// unsigned int length()
string s="hola, mundo";
cout << s.length(); // Imprime 11
```

- `find` devuelve la posición en la aparece una subcadena dentro de una cadena

```
// size_t find(const string &s,unsigned int pos=0)
cout << s.find("mundo"); // Imprime 6
// Si no encuentra la subcadena devuelve string::npos
```

Métodos de `string`

- `replace` sustituye una cadena (o parte de ella) por otra

```
// string& replace(unsigned int pos,unsigned int tam,
//const string &s)
string s="hola mundo";
s.replace(0,4,"hello"); // s vale "hello mundo"
```

- `erase` permite eliminar parte de una cadena

```
// string& erase(unsigned int pos=0,unsigned int tam=
//string::npos);
string cad="hola mundo";
cad.erase(4,3); // cad vale "holando"
```

- `substr` devuelve una subcadena de la cadena original

```
// string substr(unsigned int pos=0,unsigned int tam=
//string::npos) const;
string cad="hola mundo";
string subcad=cad.substr(2,5); // subcad vale "la mu"
```

Operadores de `string`

- Comparaciones: == (igual), != (distinto), > (mayor estricto), >= (mayor o igual), < (menor estricto) y <= (menor o igual)

```
string s1,s2;
cin >> s1; cin >> s2;
if(s1==s2) // La comparación es en orden lexicográfico
cout << "Son iguales" << endl;
```

- Asignación

```
string s1="hola";
string s2;
s2=s1;
```

- Concatenación

```
string s1="hola";
string s2="mundo";
string s3=s1+","+s2; // s3 vale "hola, mundo"
```

Operadores de `string`

- Acceso a componentes como si fuera un array de caracteres

```
string s="hola";
char c=s[3]; // s[3] vale 'a'
s[0] = 'H';
cout << s << ":" << c << endl ; // Imprime "Hola:a"
```

- Ejemplo de recorrido de un string carácter a carácter

```
string s="hola, mundo";
for(unsigned int i=0;i<s.length();i++)
    s[i]='f'; // Sustituye cada carácter por 'f'
```

Conversiones de/a `string`

- Convertir un número entero o real a `string`:

```
#include <string> // ¡Ojo! No es lo mismo que <string.h>
...
int num=100;
string s=to_string(num);
```

- Convertir un string a número entero:

```
string s="100";
int num=stoi(s);
```

- Convertir un string a número real:

```
string s="10.5";
float num=stof(s);
```

Ejercicios

- Escribe un programa que indique si la letra de un número de DNI introducido por teclado es correcta. La letra se obtiene dividiendo el número entre 23 y quedándonos con su resto. Esta tabla muestra la letra correspondiente según el resto:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

- Escribe un programa que nos devuelva el número de dígitos que contiene una cadena introducida por teclado
- Escribe un programa que lea cadenas por teclado, una por línea. El programa acabará cuando se introduzca la cadena vacía. Por cada cadena introducida, el programa deberá imprimir cuántos dígitos tiene.

Ejercicios

- Escribe un programa que pida al usuario dos cadenas e imprima el prefijo común más largo. Por ejemplo, para `polinomio` y `polinización` debe imprimir `polin`.
- Escribe una función que determine si una cadena es un palíndromo: se lee igual de izquierda a derecha que de derecha a izquierda.

Índice

Ficheros

- Ficheros (o archivos): forma en la que C++ permite acceder a la información almacenada en disco
- Su tamaño puede variar durante la ejecución del programa según los datos que almacena
- Dos maneras de guardar la información: **ficheros de texto** y ficheros binarios
- Ficheros de texto:
  - Secuencias de caracteres, tal como se mostrarían por pantalla: el valor entero 19 se guardará en fichero como los caracteres 1 y 9
  - Modo de lectura/escritura secuencial

Declaración

- Un tipo de dato más en C++
- Biblioteca `fstream`

```
#include <fstream>
```

- Tres tipos de datos básicos para trabajar con ficheros:

```
ifstream ficheroLec; // Leer de fichero
ofstream ficheroEsc; // Escribir en fichero
fstream ficheroLecEsc; // Leer y escribir en fichero [NO LO USAREMOS]
```

Apertura y cierre

- `open` asocia variable a fichero físico

```
ifstream fichero; // Vamos a leer del fichero
fichero.open("miFichero.txt");
// Ahora ya podemos leer de "miFichero.txt"

//también podemos utilizar strings
string nombreFichero="mifichero.txt"
fichero.open(nombreFichero);
```

Apertura y cierre

- `open` tiene un segundo parámetro que determina el modo de apertura
  - Lectura: `ios::in`
  - Escritura: `ios::out`
  - Añadir al final: `ios::out | ios::app`

```
ifstream ficheroLec;
ofstream ficheroEsc;
// Abrimos solo para leer
ficheroLec.open("miFichero.txt",ios::in);
// Abrimos para añadir información al final
ficheroEsc.open("miFichero.txt",ios::out|ios::app);
```

- Consideraciones:
  - Por defecto, el tipo `ifstream` se abre para lectura y el `ofstream` para escritura
  - Si abrimos un fichero que ya existe para escritura (`ios::out`), se borra todo su contenido
  - Si abrimos con `ios:app`, se añade la nueva información al final
  - Si el fichero no existe, se creará uno nuevo con tamaño inicial 0

Apertura y cierre

- Antes de leer/escribir, se debe comprobar con `is_open` si el fichero se ha abierto correctamente (true) o no (false)
- Al terminar de usar el fichero, se debe liberar con close

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
// Ya podemos trabajar con el fichero
...
fl.close(); // Cerramos el fichero
}
else // Mostrar error de apertura
```

Lectura carácter a carácter

- Función `get`: lee un carácer (no descarta blancos)
- Devuelve `false` cuando se ha llegado al final del fichero: debemos ignorar lo que ha leído

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
  char c;
  while(fl.get(c)){
    cout << c;
  }
  fl.close();
}
else{
  cout << "Error al abrir el fichero" << endl;
}
```

Lectura línea a línea

- Función `getline`: lee una línea completa (no descarta blancos)
- Devuelve `false` cuando se ha llegado al final del fichero: debemos ignorar lo que ha leído

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
  string s;
  while(getline(fi,s)){
    cout << s << endl;
  }
  fl.close();
}
else{
  cout << "Error al abrir el fichero" << endl;
}
```

Lectura con el operador >>

- Con >> se pueden leer datos de cualquier tipo igual que con `cin`
- `eof()` nos indica si se ha alcanzado el final de fichero
  - Al leer datos que estarían fuera del fichero, devuelve `true`
  - Pero cuando se leen los últimos datos válidos, devuelve `false`
- Por ejemplo, si tenemos un fichero que contiene en cada línea una cadena y dos enteros (ej. `Hola 1032 124`)

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
  string s;
  int num1,num2;
  fl >> s;
  while(!fl.eof()){ // Lee el string
    fl >> num1; // Lee el primer número
    fl >> num2; // Lee el segundo número
    cout << s << "," << num1 << "," << num2 << endl;
    fl >> s;
  }
fl.close();
}
```

Ejercicios

- Implementa un programa que lea un fichero `fichero.txt` e imprima por pantalla las líneas del fichero que contienen la cadena `Hola`.
- Diseña una función `inicioFichero` que reciba dos parámetros: el primero debe ser un número entero positivo `n` y el segundo el nombre de un fichero de texto. La función debe mostrar por pantalla las `n` primeras líneas del fichero. El programa también debe funcionar con ficheros menores de *n* líneas.
- Diseña una función `finFichero` que reciba dos parámetros: el primero debe ser un número entero positivo `n` y el segundo el nombre de un fichero de texto. La función debe mostrar por pantalla las `n` últimas líneas del fichero. El programa también debe funcionar con ficheros menores de *n* líneas.

Escritura

- Utilizamos el operador << igual que con `cout`

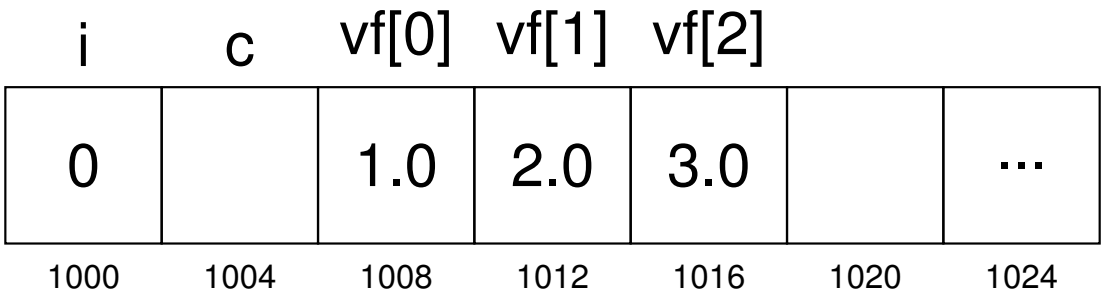
```
ofstream fe("miFichero.txt");
if(fe.is_open()){
  int num=10;
  string s="Hola, mundo";
  fe << "Un numero entero: " << num << endl;
  fe << "Un string: " << s << endl;
  fe.close();
}
else{
  cout << "Error al abrir el fichero" << endl;
}
```

Índice

Organización de la memoria

- Los datos que guardamos en variables y constantes se almacenan en la memoria del ordenador
- Cada dato se guarda una dirección o posición de memoria numerada
- Variables estáticas (las que hemos usado hasta ahora)
  - Siempre asociadas a la misma posición de memoria
  - La posición de memoria queda libre cuando salen de ámbito

```
int i=0;
char c;
float vf[3]={1.0,2.0,3.0};
```



Organización de la memoria

```
int mi_funcion(){
    int a,b;
    a=1;
    b=2;
    int c = a+b;
    return c;
}

void main(){
    int a=0;
    a=mi_funcion();
    cout << a;
}
```

Ejercicio

Haz una traza del contenido de la memoria en este fragmento de código

Punteros

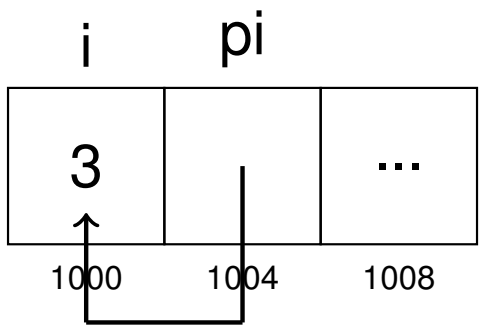
- Un puntero almacena la dirección de memoria donde se encuentra otro dato
- Se dice que el puntero “apunta” a ese dato
- Los punteros se declaran usando el carácter *
- El dato al que apunta el puntero será de un tipo concreto que deberá indicarse al declarar el puntero:

```
int *punteroEntero; // Puntero a entero
char *punteroChar; // Puntero a carácter
int *vecPunterosEntero[20]; // Array de punteros a entero
double **doblePunteroReal; // Puntero a puntero a real
```

Punteros

- El operador * permite acceder al contenido de la variable a la que apunta el puntero
- El operador & permite obtener la dirección de memoria en la que está almacenada una variable:

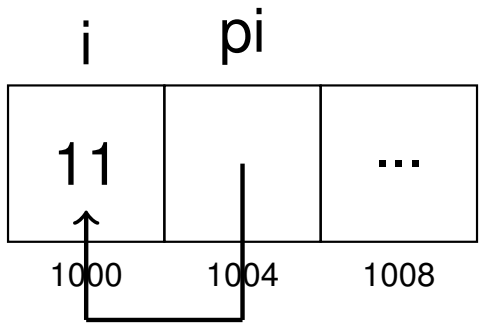
```
int i=3;
int *pi;
pi=&i; // pi contiene la dirección de memoria de i
*pi=11; // Contenido de pi es "11". Por lo tanto i = 11
```



Punteros

- Los punteros y las direcciones de memoria se pueden imprimir, aunque no suele ser necesario

```
int i=11;
int *pi;
pi=&i;
cout << pi << endl; // Muestra "1000"
cout << *pi << endl; // Muestra "11"
cout << &pi << endl; // Muestra "1004"
```



Punteros

- Para indicar que un puntero no apunta todavía a ninguna dirección de memoria, le asignamos el valor `nullptr`
- Es muy recomendable inicializar siempre los punteros con `nullptr`:

```
int i=11;
int *pi=nullptr;

if(...){ // Si se cumple alguna condición
    pi=&i;
}

if(pi != nullptr)
   cout << *pi << endl; // Muestra "11"
```

Punteros

```
int *p;
int mi_funcion(){
  int a,b;
  a=1; b=2;
  p =&a;*p = 3;
  p= &b;*p=4;
  int c = a+b;
  return c;
}

void main(){
    int a=0;
    a=mi_funcion();
    cout << a;
    cout << *p;
}
```

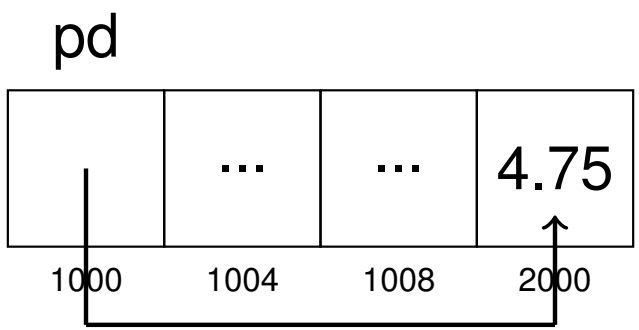
Ejercicio

Haz una traza del contenido de la memoria en este fragmento de código. ¿Hay algún error?

Reserva y liberación de memoria

- El operador `new` permite reservar memoria de manera dinámica
- Devuelve la dirección del bloque de memoria reservado, que se debe almacenar en un puntero
- Si no hay suficiente memoria para la reserva, devuelve `nullptr`

```
double *pd = new double;
if(pd!=nullptr){ // Comprueba que se ha podido reservar
  *pd=4.75;
  cout << *pd << endl; // Muestra "4.75"
}
```



Reserva y liberación de memoria

- El operador `delete` permite liberar memoria reservada con `new`
- **Siempre que se reserva con `new` hay que liberar con `delete`**

```
double *pd;
pd=new double; // Reserva memoria
...
delete pd; // Libera la memoria apuntada por pd
pd=nullptr; // Conveniente si vamos a seguir usando pd
```

- Un puntero se puede reutilizar tras liberar su contenido y reservar memoria otra vez:

```
double *pd;
pd=new double; // Reserva memoria
...
delete pd; // Libera la memoria apuntada por pd
pd=new double; // Reservamos de nuevo memoria
...
```

Reserva y liberación de memoria

```
int *p=NULL;
int mi_funcion(){
    int a,b;
    a=1;
    b=2;
    p =new int;
    *p=3;
    int c = a+b + *p;
    return c;
}

void main(){
    int a=0;
    a=mi_funcion();
    cout << *p;
}
```

Ejercicio

Haz una traza del contenido de la memoria en este fragmento de código. ¿Echas algo de menos?

Punteros y arrays

- Una variable de tipo array es en realidad un puntero al primer elemento del array
- Siempre apunta al primer elemento del array y no se puede modificar

```
int vec[5]={4,5,2,8,12};
cout << vec << endl; // Muestra la dirección de memoria
// del primer elemento del array
cout << *vec << endl; // Muestra "4"
```

- También se pueden obtener punteros a otros elementos del array

```
int vec[20];
int *pVec=vec; // Ambos son punteros a entero
*pVec=58; // Equivalente a vec[0]=58;
pVec=&(vec[7]);
*pVec=117; // Equivalente a vec[7]=117;
```

Punteros y arrays

- Los punteros pueden usarse para crear arrays dinámicos
- Se reservan añadiendo corchetes a `new`
- Se liberan añadiendo corchetes a `delete`

```
int *pv;
pv=new int[10]; // Reserva memoria para 10 enteros
pv[0]=585; // Accedemos como en un array estático
...
delete [] pv; // Liberamos toda la memoria reservada
```

- **Nos permiten cambiar el tamaño del array sin tener que declarar otra variable**

Ejercicio

Completa el siguiente código para que se puedan asignar al array tantos números en cualquier posición

```
void imprimir(int* array, int tamano){
    for(int i=0; i < ?????; i++)
        cout << ????? << " ";
}
void asignar(int* array, int &tamano, int posicion, int numero){
    ??????
}
//Main:
int tam=10;
int* arr = new int[tam];
asignar(arr,tam, 0, 1);
asignar(arr,tam, 1 , 2);
...
asignar(arr,tam, 19,15);
asignar(arr,tam, 20,42);
imprimir(arr,tam);
```

Programación Avanzada y Estructuras de Datos

2. Tipos abstractos de datos en C++

Víctor M. Sánchez Cartagena

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

25 de septiembre de 2024

Índice

Características

- Programación imperativa:
  - Elemento principal: datos (variables)
  - Funciones facilitan operar con los datos
  - Datos y funciones están separados
- Programación orientada a objetos (POO):
  - Elemento principal: objetos
  - Objeto $=$ datos (*atributos*) $+$ funciones (*métodos*) que operan sobre ellos
  - Datos y funciones están unidos

El concepto de objeto

Un objeto es una entidad con un estado (atributos o variables de instancia) y unas funciones (métodos) que pueden acceder y modificar este estado

- Para evaluar las funciones hay que enviar un mensaje al objeto
- Sólo es posible consultar el estado de un objeto mediante alguno de sus métodos
- Barrera de abstracción
- Ocultación de información

Modelado de dominio

La POO permite modelar un dominio (problema a programar, un simulador de deportes, por ejemplo) de una forma muy cercana a la realidad

- Los objetos del programa simulan los objetos (sustantivos) del dominio (por ejemplo, bicicleta, marchas, carretera, etc.)
- Los métodos de los objetos permiten modelar perfectamente las acciones (verbos, por ejemplo cambiar de marcha, pedalear, etc.) que pueden realizar

Características de los lenguajes orientados a objetos

- Objetos (dinámicos) y clases (estáticas).
- Ocultación de información: objetos agrupan datos y funciones
- Los métodos se invocan mediante mensajes
- Herencia: clases se pueden definir usando otras clases como base
- Enlace dinámico: los objetos determinan qué código ejecutar

Índice

Primera clase en C++

Coordenada

- Datos: `x`, `y`, `z`
- Métodos:
  - `obtenerX`
  - `obtenerY`
  - `obtenerZ`
  - `ponerX`
  - `ponerY`
  - `ponerZ`

Primera clase en C++

- Instanciación de la clase `Coordenada`:

```
//Declaración de un objeto de la clase
Coordenada c;

//Declaración de un array estático
Coordenada c[10];

//Declaración de un puntero
Coordenada *ptC = &c;
```

Primera clase en C++

- Declaración de una clase:

```
class NombreClase {

//Atributos y métodos

};
```

- Clase `Coordenada`:

```
class Coordenada {
   int x,y,x;
};
```

Primera clase en C++

- Poniéndolo todo junto:

```
class Coordenada {
    int x,y,z;
};

int main(){
   int i;
   Coordenada c;
   return 0;
}
```

Compilación

```
g++ -std=c++11 -Wall -g prog.cc -o prog
```

Acceso a los miembros de una clase

- Mediante . para variables estáticas

```
Coordenada c;
cout << "Componente x"  << c.x << endl;
cout << "Componente y"  << c.y << endl;
cout << "Componente z"  << c.z << endl;
```

- Mediante -> para punteros

```
Coordenada c;
Coordenada *ptC = &c;
cout << "Componente x"  << ptC->x << endl;
cout << "Componente y"  << ptC->y << endl;
cout << "Componente z"  << ptC->z << endl;
```

Acceso a los miembros de una clase

- Pero el siguiente código da error de compilación: `error: ‘int Coordenada::x’ is private within this context`

```
class Coordenada {
    int x,y,z;
};

int main(){
   Coordenada c;
   c.x=1;
   c.y=2;
   c.z=1;
   return 0;
}
```

Control de acceso

- `public`: Accesible tanto desde la propia clase como desde otros métodos y funciones
- `private`: Accesible exclusivamente desde métodos de la propia clase y funciones y clases amigas
- `protected`: Los accesos permitidos con `private`, más acceso desde clases derivadas (herencia)

Control de acceso

- Implementación:

```
class MiClase{
public:
    //Parte pública
    //Normalmente, sólo métodos y no atributos

protected:
    //Parte protegida

private:
    //Parte privada
    //Normalmente, atributos y métodos auxiliares (no visibles desde e

};
```

Control de acceso

- Versión *arreglada* del código anterior:

```
#include<iostream>
using namespace std;
class Coordenada {
    public:
    int x,y,z;
};
int main(){
   Coordenada c;
   c.x=1;
   c.y=2;
   c.z=1;
   cout << "(" << c.x << ", " << c.y << ", " << c.z << ")" << endl;
   return 0;
}
```

- Atención: rompe ocultación de información

Separación de interfaz e implementación

- La declaración de clase se incluye un fichero `NombreClase.h`
- La implementación de los métodos en un fichero `NombreClase.cc`

Ventajas:

- Permite distribuir fichero `.h` y fichero `.cc` compilado $\rightarrow$ `.o`
- Se puede cambiar la implementación de los métodos sin afectar al programa que utiliza la clase

Desventajas:

- Cambios en atributos privados obligan a cambiar fichero `.h`

Separación de interfaz e implementación

- `Coordenada.h`:

```
  class Coordenada {
    private:
    int x,y,z;
    public:
    void setX(int);
    void setY(int);
    void setZ(int);
    int getX();
    int getY();
    int getZ();
};
```

Separación de interfaz e implementación

- Coordenada.cc:

```
#include "Coordenada.h"
void Coordenada::setX( int ix ){
    x=ix;
}
void Coordenada::setY( int iy ){
    y=iy;
}
void Coordenada::setZ( int iz ){
    z=iz;
}
int Coordenada::getX(){
    return x;
}
int Coordenada::getY(){
    return y;
}
int Coordenada::getZ(){
    return z;
}
```

Separación de interfaz e implementación

- Programa principal. `main.cc`:

```
#include<iostream>
#include "Coordenada.h"
using namespace std;
int main(){
    Coordenada c;
    c.setX(1);
    c.setY(2);
    c.setZ(1);
    cout<< "("<< c.getX()<< ", "<< c.getY()<< ", "<< c.getZ()<< ")"<<
    return 0;
}
```

Compilación

- Compilación de la clase `Coordenada` junto con el programa principal:

```
g++ -std=c++11 -Wall -c Coordenada.cc
g++ -std=c++11 -Wall -c main.cc
g++ -std=c++11 -Wall -o main Coordenada.o main.o
```

Compilación

Fases de la compilacón con `gcc`:

1. Preprocesado: se procesan las directivas que empiezan por `#`: `#include`, `#define`, etc.
   - opción `-E`
2. Compilación y ensamblado: se genera código máquina
   - opción `-S`: compila y no ensambla $\rightarrow$ se puede ver el código ensamblador
   - opción `-c`: compila y ensambla pero no enlaza $\rightarrow$ ficheros con código máquina (`.o`) diferentes para cada clase y programa principal
3. Enlazado: une el código máquina de las diferentes partes (incluidas bibliotecas del sistema, p.ej. `iostream`)

Diseño orientado a objetos

- ¿Se puede hacer este programa más *orientado a objetos*?

```
#include<iostream>
#include "Coordenada.h"
using namespace std;
int main(){
    Coordenada c;
    c.setX(1);
    c.setY(2);
    c.setZ(1);
    cout<< "("<< c.getX()<< ", "<< c.getY()<< ", "<< c.getZ()<< ")"<<
    return 0;
}
```

Diseño orientado a objetos

- ¿Se puede hacer este programa más *orientado a objetos*? **Sí**.

`main.cc`:

```
#include<iostream>
#include "Coordenada.h"
using namespace std;
int main(){
    Coordenada c;
    c.setX(1);
    c.setY(2);
    c.setZ(1);
    c.imprimir();
    return 0;
}
```

Diseño orientado a objetos

- ¿Se puede hacer este programa más *orientado a objetos*? **Sí**.
`coordenada.cc`:

```
#include<iostream>
#include "Coordenada.h"
....
void Coordenada::imprimir(){
    cout<< "("<< c.getX()<< ", "<< c.getY();
    cout<< ", "<< c.getZ()<< ")"<< endl;
}
....
```

Compilación

- ¿Qué pasa si tengo 100 clases?:

```
g++ -std=c++11 -Wall -c Coordenada.cc
g++ -std=c++11 -Wall -c OtraClase.cc
g++ -std=c++11 -Wall -c OtraClaseMas.cc
...
g++ -std=c++11 -Wall -c main.cc
g++ -std=c++11 -Wall -o main Coordenada.o ...
```

La herramienta `make`

- `make` es una herramienta que permite automatizar la construcción de unos ficheros a partir de otros (en la mayoría de los casos, para compilar, pero no exclusivamente)
- Las reglas para construir los ficheros se escriben en un fichero llamado `makefile`
- Sólo hay que teclear la orden `make` para que el proyecto se compile

La herramienta `make`

- Hay que definir el fichero que se quiere generar, qué ficheros se necesitan y la orden a ejecutar:

```
Objetivo: Dependencia1 Dependencia2 ...
        orden a ejecutar
```

- Si una dependencia no existe, se busca una regla para generarla, y así se pueden ejecutar cientos de órdenes

```
Dependecia1: Dependencia11 Dependencia12
        orden a ejecutar
Dependecia2: Dependencia21 Dependencia22
        orden a ejecutar
Dependecia21: Dependencia3 Dependencia4
        orden a ejecutar
```

La herramienta `make`

Ejercicio

Escribe un fichero `makefile` para compilar el programa principal que emplea la clase `Coordenada`

La herramienta `make`

Ejercicio

Escribe un fichero `makefile` para compilar el programa principal que emplea la clase `Coordenada`

```
main: main.o Coordenada.o
        g++ -o main Coordenada.o main.o
main.o: main.cc Coordenada.h
        g++ -std=c++11 -Wall -c main.cc
Coordenada.o: Coordenada.cc Coordenada.h
        g++ -std=c++11 -Wall -c Coordenada.cc
```

La herramienta `make`

Ejercicio

Escribe un fichero `makefile` para compilar el programa principal que emplea la clase `Coordenada`

```
main: main.o Coordenada.o
        g++ -o main Coordenada.o main.o
main.o: main.cc Coordenada.h
        g++ -std=c++11 -Wall -c main.cc
Coordenada.o: Coordenada.cc Coordenada.h
        g++ -std=c++11 -Wall -c Coordenada.cc
```

- Problema: demasiada repetición de opciones

La herramienta `make`

- Se pueden emplear variables para agrupar opciones que se van a repetir mucho

```
COMP=g++
OPC=-std=c++11 -Wall
OBJS=main.o Coordenada.o

main: $(OBJS)
        $(COMP) -o main $(OBJS)
main.o: main.cc Coordenada.h
        $(COMP) $(OPC) -c main.cc
Coordenada.o: Coordenada.cc Coordenada.h
        $(COMP) $(OPC) -c Coordenada.cc
```

Sobrecarga de funciones

- Se pueden definir funciones con el mismo nombre, siempre que tengan distinto número, orden y/o tipo de parámetros

```
#include<iostream>
using namespace std;
int minimo(int a, int b){
    if(a < b)
        return a;
    else
        return b;
}
int minimo(int a, int b, int c){
    return min(a, min(b,c));
}
int main(){
    cout << minimo(1,2,3) << endl;
}
```

Sobrecarga de funciones

- Si la única diferencia es el tipo de salida, se producirá un error:

```
#include<iostream>
using namespace std;
int minimo(int a, int b){
    if(a < b)
        return a;
    else
        return b;
}
int minimo(int a, int b, int c){
    return min(a, min(b,c));
}
float minimo(int a, int b, int c){
    return min(a, min(b,c));
}
int main(){
    cout << minimo(1,2,3) << endl;
}
```

- `error: ambiguating new declaration of 'float minimo(int, int, int)`

Forma canónica de una clase

Métodos que debemos implementar obligatoriamente para asegurarnos de que los objetos de la clase se comportan como variables "normales":

- Constructor
- Destructor
- Constructor de copia
- Operador de asignación

Constructor

- Se invoca automáticamente cada vez que se crea un objeto de la clase
- Debe iniciar todo lo necesario para operar con el objeto $\rightarrow$ recordad cuál es el valor de las variables sin inicializar
  - Inicializar los atributos a valores razonables
  - Reservar memoria, si es necesario
- Declaración:
  - Mismo nombre que la clase
  - No devuelve nada (ni `void`)
  - Normalmente en la parte pública
  - Se puede sobrecargar

Constructor

- Constructor por defecto para `Coordenada`:

```
//En Coordenada.h
public:
Coordenada();
....

//En Coordenada.cc
Coordenada::Coordenada(){
  x=0; y=0; z=0;
}
```

Constructor

- Constructor sobrecargado para `Coordenada`:

```
//En Coordenada.h
public:
Coordenada();
Coordenada(int, int, int);
....

//En Coordenada.cc
Coordenada::Coordenada(int px, int py, int pz){
  x=px; y=py; z=pz;
}
```

- Uso del nuevo constructor en `main.cc`:

```
Coordenada c(2,4,6);
c.imprimir();
```

Constructor de copia

- Se invoca automáticamente cada vez que se copia un objeto de la clase:
  - Cuando declaramos un objeto a partir de otro
  - Al pasar por valor un objeto de la clase a una función
  - Cuando una función devuelve (por valor) un objeto de la clase
- Debe copiar los atributos de un objeto a otro, o reservar memoria si es necesario
- Si no se declara, el compilador crea uno que simplemente copia los atributos
- Declaración:
  - Debe recibir un objeto de la clase por *referencia constante*

Constructor de copia

- Constructor de copia para `Coordenada`:

```
//En Coordenada.h
public:
Coordenada();
Coordenada(int, int, int);
Coordenada(const Coordenada&);
....

//En Coordenada.cc
Coordenada::Coordenada(const Coordenada& c){
  x=c.x; y=c.y; z=c.z;
}
```

- Uso del nuevo constructor en `main.cc`:

```
Coordenada c(2,4,6);
Coordenada c2(c);
c.imprimir();
```

Destructor

- Se invoca automáticamente cada vez que se elimina un objeto de la clase:
  - Cuando sale de ámbito
  - Cuando se invoca al operador `delete` sobre un puntero a un objeto de la clase
- Debe asegurarse de liberar todos los recursos asociados al objeto: `delete` si hay memoria dinámica asociada al objeto
- Si no se declara, el compilador crea uno que no hace nada
- Si hay atributos estáticos, no es necesario hacer nada con ellos, se liberarán automáticamente
- Declaración:
  - $\sim$ seguido del nombre de la clase: `~Coordenada()`
  - No recibe ni devuelve nada

Destructor

- Destructor para `Coordenada`:

```
//En Coordenada.h
public:
Coordenada();
Coordenada(int, int, int);
Coordenada(const Coordenada&);
~Coordenada();
....

//En Coordenada.cc
Coordenada::~Coordenada(){
  x=0; y=0; z=0;
}
```

Constructores, destructores y memoria dinámica

Ejercicio

Imagina que imprimimos un mensaje en el constructor y destructor de `Coordenada`. ¿Qué se imprimiría al ejecutar este programa?

```
#include<iostream>
#include "Coordenada.h"
using namespace std;
int main(){
    Coordenada array[3];
    Coordenada* ptC=nullptr;

    cout << "Reserva memoria" << endl;
    ptC= new Coordenada[3];
    if(ptC == nullptr)
        return 1;

    cout << "Fin del programa" << endl;
    return 0;
}
```

El puntero `this`

- Sólo se puede usar dentro de métodos de una clase, no dentro de funciones
- Contiene la dirección de memoria del objeto que ha invocado al método:

```
Coordenada::Coordenada(int x, int y, int z){
    this->x=x;
    this->y=y;
    this->z=z;
}
```

El puntero `this`

- También se emplea en operadores, para permitir aplicarlos en cascada: `a=b=c`
- Y para evitar que un objeto se asigne a sí mismo
- Es de sólo lectura
  - El siguiente código produciría un error de compilación:

```
Coordenada::Coordenada(const Coordenada &c){
     this=&c;
}
```

El modificador `const`

- Para declarar constantes:

```
const int TAMANO=100;
```

- Para indicar que un parámetro pasado por referencia no va a cambiar:

```
Coordenada::Coordenada(const Coordenada &c){
     x=c.getX();
     ....
```

- Para indicar que un método de una clase no modifica el objeto:

```
class Coordenada{
...
public:
int getX() const;
int getY() const;
int getZ() const;
...
```

El modificador `const`

- Un objeto constante solo puede invocar a métodos marcados con `const`
- Un objeto no constante puede invocar a cualquier método
- El siguiente código fallaría si `getX` no se marca como método constante:

```
Coordenada::Coordenada(const Coordenada &c){
     x=c.getX();
     ....
```

El modificador `const`

- Hay que marcar el método como `const` tanto en el fichero `.h` como en el `.cpp`:

```
//Coordenada.h
class Coordenada{
...
public:
int getX() const;
...

//Coordenada.cc
int Coordenada::getX() const{
   return x;
```

- Un método marcado como `const` no puede modificar el objeto. Error de compilación:

```
//Coordenada.cc
int Coordenada::getX() const{
   x--;
   return x;
}
```

Sobrecarga de operadores

- Sería interesante poder sumar coordenadas empleando el mismo símbolo + que para tipos básicos:

```
Coordenada a(1,2,3);
Coordenada b(4,5,6);
Coordenada c;
c=a+b;
```

Sobrecarga de operadores

- Sería interesante poder sumar coordenadas empleando el mismo símbolo + que para tipos básicos:

```
Coordenada a(1,2,3);
Coordenada b(4,5,6);
Coordenada c;
c=a+b;
```

- Se puede conseguir implementando métodos `operator<símbolo>`:

```
// Equivale a c=a+b;
c.operator=( a.operator+(b) );
```

Operador de asignación

- Es recomendable sobrecargarlo siempre → forma canónica
- Si no se define, se produce una copia atributo a atributo que puede dar problemas con memoria dinámica
- Devuelve un objeto de la clase *por referencia*: siempre `*this`
  - Para poder hacer: `a=b=c` → `a.operator=(b.operator=(c))`
- Recibe un objeto de la clase por referencia constante

Operador de asignación

- `Coordenada.h`:

```
public:
Coordenada& operator=(const Coordenada&);
```

- `Coordenada.cc`:

```
Coordenada& Coordenada::operator=(const Coordenada& c){
    (*this).~Coordenada();
    x=c.x;
    y=c.y;
    z=c.z;
    return (*this);
}
```

- Atención: error si intentamos hacer `a=a;`

Operador de asignación

- Solución: comprobar la dirección de memoria para evitar auto-asignación

```
Coordenada& Coordenada::operator=(const Coordenada& c){
    if(this != &c){
        (*this).~Coordenada();
        x=c.x;
        y=c.y;
        z=c.z;
    }
    return (*this);
}
```

Operadores aritméticos

- No modifican a los operandos y devuelven un nuevo objeto
- Devuelven el tipo del objeto por valor
- Es recomendable que el método sea marcado como constante (operando izquierdo) y el argumento sea una referencia constante (operando derecho)
- Normalmente implican crear un objeto temporal

Operadores aritméticos

- Coordenada.h:

```
public:
Coordenada operator+(const Coordenada&) const;
```

- Coordenada.cc:

```
Coordenada Coordenada:::operator+(const Coordenada& c) const{
   Coordenada cn;
   cn.setX(x+c.x)
   cn.setY(y+c.y)
   cn.setZ(z+c.z)
   return cn;
}
```

Operadores de comparación

- Devuelven un dato de tipo `bool`
- No modifican los operandos
- `Coordenada.h`:

```
public:
bool operator==(const Coordenada&) const;
```

- `Coordenada.cc`:

```
bool Coordenada::operator==(const Coordenada& c) const{
   return (x==c.x && y==c.y && z=c.z)
}
```

Índice

Relaciones entre clases en C++

- Asociación
- Todo-parte
  - Agregación
  - Composición
- Herencia

Asociación

- Relación (unidireccional o bidireccional) entre los objetos
- Los objetos existen de forma independiente
- Si uno de los dos objetos se destruye, el otro se mantiene
- Ejemplo: relación entre una casilla y una pieza en un juego de ajedrez
- Se implementa en C++ como un puntero:

```
class Pieza{
private:
Casilla* casilla;
public:
....
};
```

Agregación

- Todo-parte: un objeto está formado por varios objetos de otro tipo
- Los objetos “parte” pueden pertenecer a varios objetos “todo”
- Si el objeto “todo” se destruye, los objetos “parte” siguen existiendo
- Ejemplo: relación entre un equipo y sus miembros
- Se implementa en C++ como una colección de punteros:

```
class Equipo{
private:
Persona* miembros[MAX_MIEMBROS];
public:
....
};
```

Composición

- Todo-parte: un objeto está formado por varios objetos de otro tipo
- Los objetos “parte” no pueden pertenecer a varios objetos “todo“ ni existir al margen del mismo
- Si el objeto “todo” se destruye, los objetos “parte” también deben destruirse
- Ejemplo: relación entre un libro y sus capítulos
- Se implementa en C++ mediante layering:

```
class Libro{
private:
Capitulo introduccion;
Capitulo cap1;
...
public:
....
};
```

Herencia

- Es-un: una clase es un subtipo o especialización de otra clase
- La clase derivada `hereda` los atributos y métodos de la clase base y añade los suyos propios
- Ejemplo: en un sistema de gestión de datos médicos, un paciente (clase derivada) es una persona (clase base)
- Mecanismo muy potente en C++ que no vamos a estudiar en la asignatura

Layering

- Mecanismo que permite incluir objetos de una clase dentro de otro objeto
- Ejemplo: `Linea.h`:

```
class Linea{
private:
Coordenada c1, c2;
public:
Linea();
Linea(const Coordenada&, const Coordenada&);
Linea(const Linea&)
....
};
```

Layering

- Constructores: se ejecuta automáticamente el código del constructor por defecto de `Coordenada` antes de cualquier constructor de `Linea`
  - Excepto si se usa un *inicializador* en el fichero `.cc`

```
Linea::Linea(const Coordenada& a,const Coordenada& b ):c1(a),c2(b
....
Linea::Linea(const Linea& l ):c1(l.c1),c2(l.c2){
....
```

- Destructor: se ejecuta automáticamente el código del destructor `Coordenada` después de destructor de `Linea`

Guardas de inclusión

- `Linea.h`:

```
#include "Coordenada.h"
```

- `main.cc`:

```
#include "Linea.h"
#include "Coordenada.h"
```

- Resultado: error de compilación porque `Coordenada` se ha definido dos veces

Guardas de inclusión

- En cada fichero `.h`:

```
#ifndef __NOMBRE_CLASE__
#define __NOMBRE_CLASE__
...
#endif
```

- `Coordenada.h`:

```
#ifndef __COORDENADA__
#define __COORDENADA__

class Coordenada{
...

};

#endif
```

Índice

Tipos abstractos de datos

- Concepto precursor de la programación orientada a objetos
- **Tipo de datos**: clasifica los objetos de los programas (variables, parámetros, constantes) y determina
  - Los valores que pueden tomar
  - Las operaciones que se pueden realizar
- **Abstracto**: los usamos en nuestros programas según su **especificación** (qué hace) e independientemente de su **implementación**
- Una especificación $\rightarrow$ múltiples implementaciones

Tipos abstractos de datos (TADs)

- Especificación: defínción de las propiedades y operaciones
  - Especificaciones algebraicas formales (en desuso)
  - Especificaciones informales en lenguaje natural: operaciones (métodos de la clase) + explicación de lo que hacen
- Implementación:
  - Determinar la representación
  - Codificar las operaciones en base a esa representación

Ejemplo

Especificación del TAD `Hora`

Definición: una instancia del TAD `Hora` almacena una hora del día. La hora puede corresponder a cualquier fecha, y no nos interesa almacenar ni representar la fecha. Emplearemos una precisión de segundos, por que nos interesará saber la hora en términos de horas, minutos y segundos, por ejemplo, `13:24:06`. La hora por defecto será `00:00:00`.

Ejemplo

Especificación del TAD `Hora`

Operaciones:

- Modifica la hora siempre que: el parámetro `h`, que representa las horas, esté entre 0 y 23; el parámetro `m`, que representa los minutos, esté entre 0 y 59; y el parámetro `s`, que representa los segundos, esté entre 0 y 59. Si algún parámetro no es correcto, no modifica la hora y devuelve `false`.

```
bool modificaHora(int h, int m, int s);
```

- Devuelve la hora completa en formato `hh:mm:ss`.

```
string getHoraCompleta();
```

Ejemplo

Especificación del TAD `Hora`

Operaciones:

- Devuelve la hora del día (entre 0 y 23)

```
int getHoras();
```

- Devuelve los minutos trasncurridos desde el comienzo de la última hora. Por ejemplo, `24` para las `13:24:06`

```
int getMinutos();
```

- Devuelve los segundos trasncurridos desde el comienzo del último minuto. Por ejemplo, `6` para las `13:24:06`

```
int getSegundos();
```

Ejemplo

Especificación del TAD `Hora`

Operaciones:

- Incrementa la hora en `s` segundos. Si la hora sobrepasa el final del día, vuelve a comenzar tantas veces como sea necesario. Si el parámetro `s` es negativo, la hora no se modifica y se devuelve `false`. En caso contrario, se devuelve `true`

```
bool incrementa(int s);
```

Ejemplo

Implementación del TAD `Hora`

- `Hora.h`:

```
class Hora{
private:
?????
public:
Hora();
Hora(const Hora&);
~Hora();
Hora& operator=(const Hora&)
bool modificaHora(int h, int m, int s);
string getHoraCompleta() const;
int getHoras() const;
int getMinutos() const;
int getSegundos() const;
bool incrementa(int);
};
```

Ejercicio

Implementa el TAD `Hora` eligiendo la mejor **representación**

Programación Avanzada y Estructuras de Datos

3: La eficiencia de los algoritmos

Víctor M. Sánchez Cartagena

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

2 de octubre de 2024

Índice

Índice

Problema introductorio #1

- ¿Cuál de estas dos funciones para buscar un elemento en un vector es más rápida? Asume que el vector tiene cientos de elementos. Sólo una opción es verdadera
  a. `buscar1` siempre es más rápida
  b. `buscar2` siempre es más rápida
  c. Una u otra, depende del contenido del vector
  d. `buscar2` nunca es más rápida que `buscar1`
  e. `buscar1` nunca es más rápida que `buscar2`

```
int buscar1( int[] vec, int n,
     int z ){
  for( int i = 0; i < n i++ )
    if( vec[i] == z )
      return i;
  return -1;
}
```

```
int buscar2( int[] vec, int n,
     int z ){
  int posicion=-1;
  for( int i = 0; i < n; i++ )
    if( vec[i] == z )
      posicion=i;
  return posicion;
}
```

Problema introductorio #2

- ¿Cuál de estas dos funciones es más rápida para vectores muy grandes? Sólo una opción es verdadera
  a. `maximo1` es más rápida
  b. `maximo2` es más rápida
  c. Una u otra, depende del contenido del vector
  d. `maximo2` nunca es más rápida que `maximo1`
  e. `maximo1` nunca es más rápida que `maximo2`

```
int maximo1 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<500000; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

```
int maximo2 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<n; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

Índice

Noción de complejidad

Definición: complejidad de un algoritmo

Es una medida de los recursos que necesita un algoritmo para resolver un problema

Los recursos más usuales son:

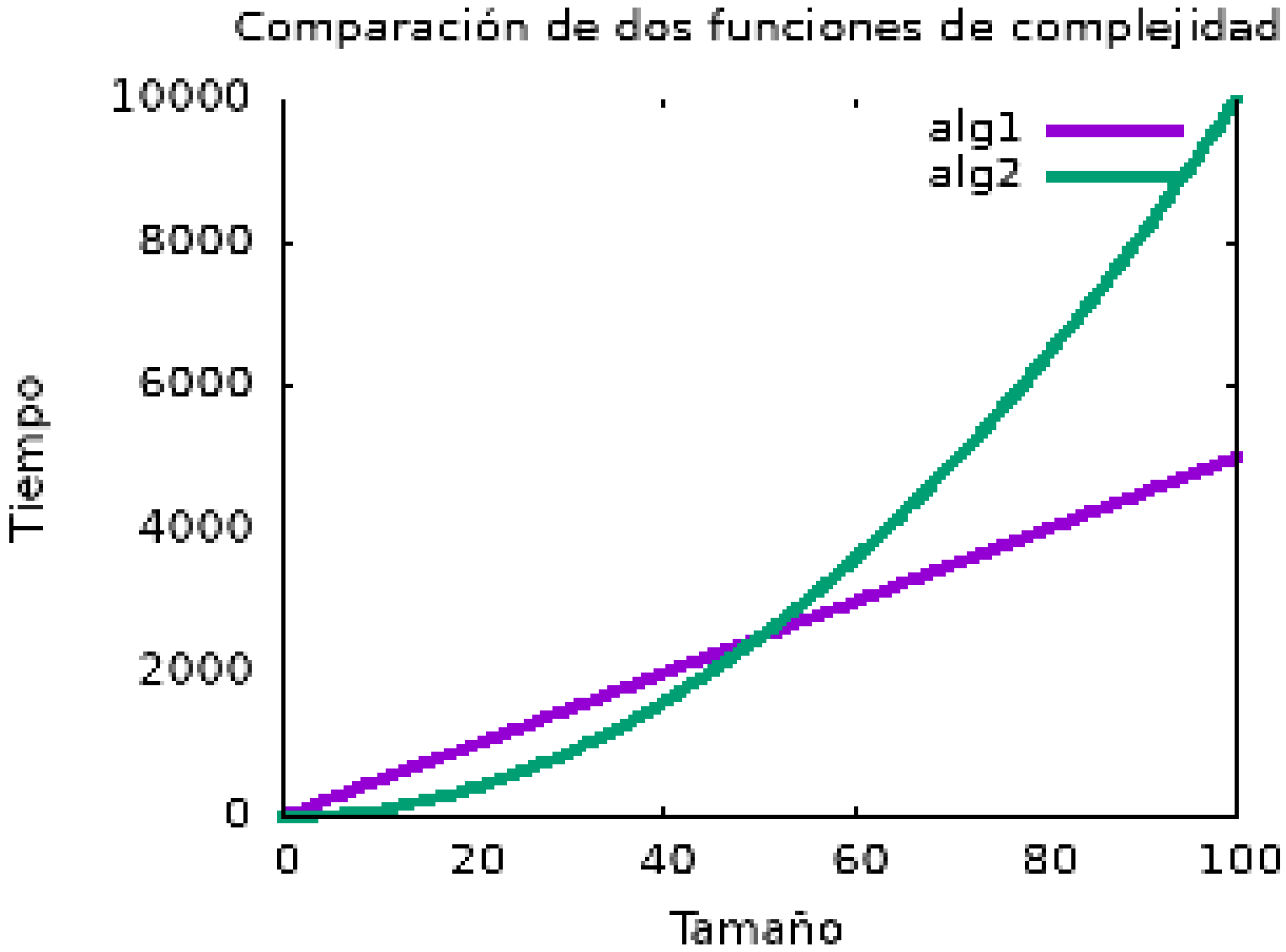
- Tiempo: Complejidad temporal
- Memoria: Complejidad espacial

Se suele expresar en función de la dificultad *a priori* del problema:

- Tamaño del problema: lo que ocupa su representación
- Parámetro representativo: el valor de n si queremos calcular $n!$

- La complejidad nos permite comparar y valorar algoritmos
- Nos centraremos, sobre todo, en la *complejidad temporal*

Noción de complejidad



Tiempo de ejecución de un algoritmo

El tiempo de ejecución de un algoritmo depende de:

Factores externos

- La máquina en la que se va a ejecutar
- El compilador
- La experiencia del programador

Factores internos

- El número de instrucciones que ejecuta el algoritmo y su duración

Principio de invarianza

Principio de invarianza

El tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa

Tiempo de ejecución: $c \cdot T(n)$

- c: constante multiplicativa que depende de factores externos
- $T(n)$: contribución de factores internos

Tiempo de ejecución de un algoritmo

Análisis empírico o *a posteriori*

Ejecutar el algoritmo para distintos valores de entrada y cronometrar el tiempo de ejecución

- ▲ Es una medida del comportamiento del algoritmo en su entorno
- ▼ El resultado depende de los factores externos e internos

Análisis teórico o *a priori*

Obtener una función que represente el tiempo de ejecución (en operaciones elementales) del algoritmo para cualquier valor de entrada

- ▲ El resultado depende sólo de los factores internos
- ▲ No es necesario implementar y ejecutar los algoritmos
- ▼ No obtiene una medida real del comportamiento del algoritmo en el entorno de aplicación

Tiempo de ejecución de un algoritmo

Operaciones elementales

Son aquellas operaciones que realiza el ordenador en un tiempo acotado por una constante

Operaciones elementales

- Operaciones aritméticas básicas
- Asignaciones a variables de tipo predefinido por el compilador
- Los saltos (llamadas a funciones, retorno desde ellos . . .)
- Comparaciones lógicas
- Acceso a estructuras indexadas básicas (vectores o matrices)
- **Cualquier combinación de las anteriores acotada por una constante**

Tiempo de ejecución de un algoritmo

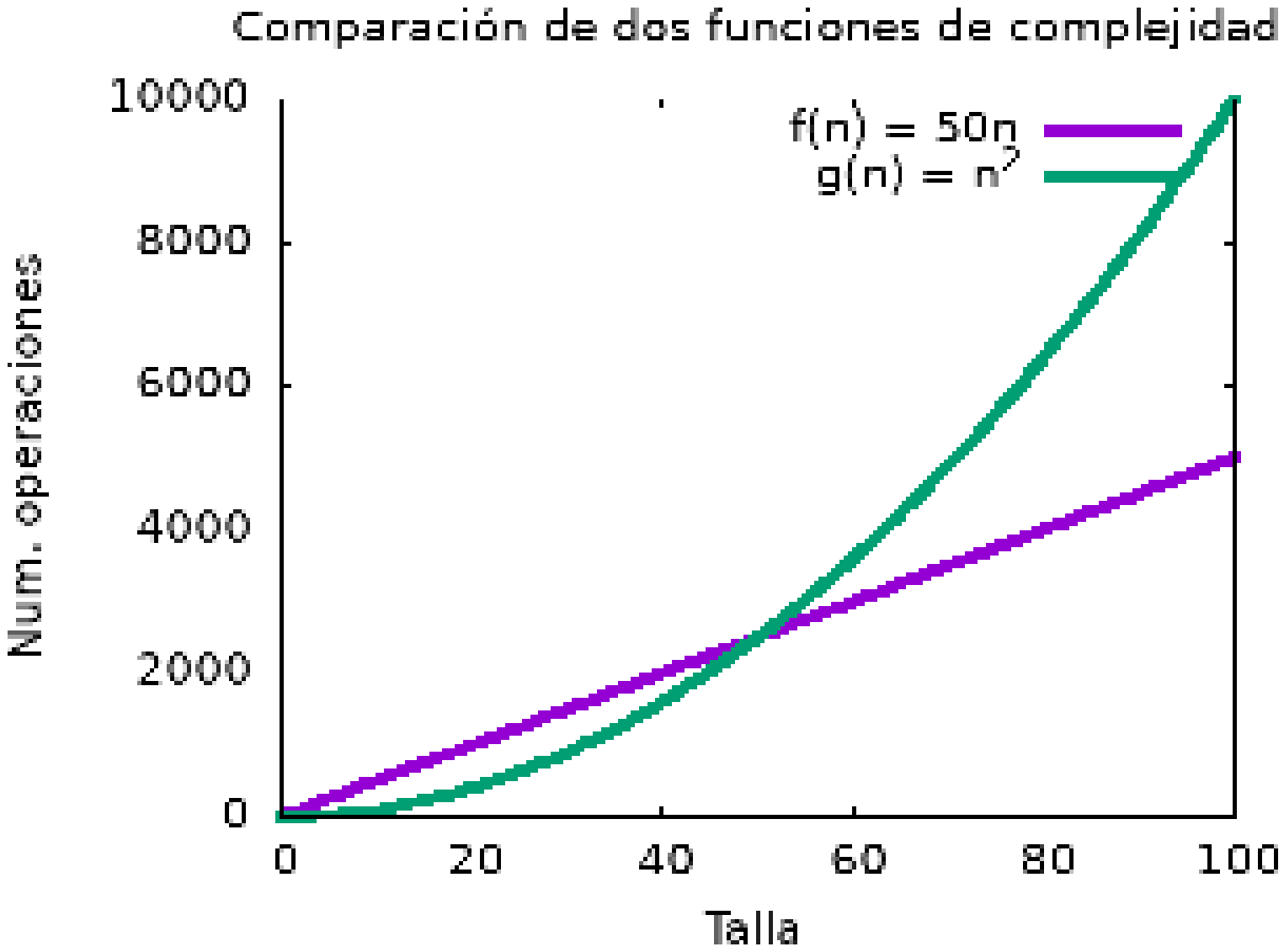
- Se suele considerar que el coste temporal de las operaciones elementales es unitario
- En el contexto de Programación Avanzada y Estructuras de Datos:

Tiempo de ejecución de un algoritmo

Función ($T(n)$) que mide el número de operaciones elementales que realiza el algoritmo cualquier talla de problema n

- *Talla*: tamaño del problema o parámetro representativo, según convenga

Tiempo de ejecución de un algoritmo



Índice

Ejemplos

- Talla: valor de n

Programa 1:

```
int ejemplo1 (int n){
  n+=n;
  return n;
}
```

Programa 2:

```
int ejemplo2 (int n){
  for(int i=0; i<=2000; i++)
    n+=n;
  return n;
}
```

Ejemplos

- Talla: valor de n

Programa 1:

```
int ejemplo1 (int n){
  n+=n;
  return n;
}
```

Programa 2:

```
int ejemplo2 (int n){
  for(int i=0; i<=2000; i++)
    n+=n;
  return n;
}
```

- El programa 1 calcula el resultado en 1 operación: $T(n) = 1$

Ejemplos

- Talla: valor de n

Programa 1:

```
int ejemplo1 (int n){
  n+=n;
  return n;
}
```

Programa 2:

```
int ejemplo2 (int n){
  for(int i=0; i<=2000; i++)
    n+=n;
  return n;
}
```

- El programa 1 calcula el resultado en 1 operación: $T(n) = 1$
- El programa 2 calcula el resultado en 1 operación: $T(n) = 1$

Ejemplos

- Talla: valor de n

Programa 3:

```
int ejemplo3 (int n){
  int i, j;
  j = 2;

  for (i=0; i<=2000; i++)
    j=j*j;
  for (i=0; i<= n; i++){
    j = j + j;
    j = j - 2;
  }
  return j;
}
```

Ejemplos

- Talla: valor de n

Programa 3:

```
int ejemplo3 (int n){
  int i, j;
  j = 2;

  for (i=0; i<=2000; i++)
    j=j*j;
  for (i=0; i<= n; i++){
    j = j + j;
    j = j - 2;
  }
  return j;
}
```

$$T(n) = 1 + (n+1) \cdot 1 = n + 2$$

Ejemplos

- Talla: valor de n

Programa 4:

```
int ejemplo4 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=1; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

Ejemplos

- Talla: valor de n

Programa 4:

```
int ejemplo4 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=1; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

$$T(n) = 1+(n+1)\cdot n\cdot 1 = n^2+n+1$$

Ejemplos

- Talla: valor de n

Programa 5:

```
int ejemplo5 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=i; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

¿ $T(n) = 1 + (n+1) \cdot n \cdot 1$?

Ejemplos

- Talla: valor de n

Programa 5:

```
int ejemplo5 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=i; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

¿ $T(n) = 1 + (n+1) \cdot n \cdot 1$?

ERROR

Resolución de sumatorios

- Sumatorio: expresión matemática para “escribir” una suma de manera compacta

$$\sum_{i=1}^{n}(\text{exp})$$

- ... equivalente a $\text{exp} + \text{exp} + \text{exp} + \ldots$ n veces (desde que i vale 1 hasta que vale n)

Ejemplos:

- $\sum_{i=1}^{4}(2) = 2 + 2 + 2 + 2 = 8$
- $\sum_{i=1}^{4}(2i) = 2 + 4 + 6 + 8 = 20 \rightarrow$ progresión aritmética
- $\sum_{i=1}^{4}(2^i) = 2 + 4 + 8 + 16 = 30 \rightarrow$ progresión geométrica

Resolución de sumatorios

- Sumatorio de una constante:

$$\sum_{i=a}^{b}(C) = (b - a + 1) \cdot C$$

- Suma de una progresión aritmética

$$\sum_{i=a}^{b}(p_i) = \frac{(p_a + p_b)}{2} \cdot (b - a + 1)$$

- No trataremos con progresiones geométricas

Ejemplos

- Talla: valor de n

Programa 5:

```
int ejemplo5 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=i; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

Ejemplo

Programa 5:

```
int ejemplo5 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=i; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

$$T(n) = 1 + \sum_{i=0}^{n}\sum_{j=i}^{n} 1 =$$

$$1 + \sum_{i=0}^{n}(n-i+1)\cdot 1 =$$

$$1 + \frac{n+1+1}{2}\cdot(n+1) =$$

$$1 + (\frac{n}{2}+1)\cdot(n+1) =$$

$$1 + \frac{n^2}{2} + \frac{n}{2} + n + 1 =$$

$$\frac{n^2}{2} + \frac{3n}{2} + 2$$

Ejercicios

Calcula la función tiempo de ejecución, para los siguientes fragmentos de código (talla $=$ valor de `n`):

Ejercicio 1:

```
for(i = sum = 0; i < n; i++) sum += a[i];
```

Ejercicio 2:

```
for(i = 0; i < n; i++) {
   for(j = 1, sum = a[0]; j <= i; j++) sum += a[j];
   cout << "La suma del subarray " << i << " es " << sum << endl; }
```

Ejercicio 3:

```
for(i = 4; i < n; i++) {
   for(j = i-3, sum = a[i-4]; j <= i; j++) sum += a[j];
   cout << "La suma del subarray " << i-4 << " es " << sum << endl; }
```

Índice

¿Por qué necesitamos cotas?

- Dado un vector de enteros `v` y el entero `z`
  - Devuelve el primer índice `i` tal que `v[i] == z`
  - Devuelve `-1` en caso de no encontrarlo

Búsqueda de un elemento

```
int buscar( int[] v, int n, int z ){
  for( int i = 0; i < n; i++ )
    if( v[i] == z )
      return i;
  return -1;
}
```

v	z	Operaciones
$(1,0,2,4)$	1	1
$(1,0,2,4)$	0	2
$(1,0,2,4)$	2	3
$(1,0,2,4)$	4	4
$(1,0,2,4)$	5	5

¿Por qué necesitamos cotas?

- No podemos contar el número de operaciones porque para diferentes entradas de un mismo tamaño de problema se obtienen diferentes complejidades
- ¿Qué podemos hacer?

¿Por qué necesitamos cotas?

- No podemos contar el número de operaciones porque para diferentes entradas de un mismo tamaño de problema se obtienen diferentes complejidades
- ¿Qué podemos hacer?

Solución

Acotar el coste mediante dos funciones que expresen respectivamente, el coste máximo y el coste mínimo del algoritmo (cotas de complejidad)

Cotas de complejidad

- Cuando aparecen diferentes casos para una misma talla n, se introducen las siguientes medidas de la complejidad
  - Caso peor: cota superior del algoritmo $\rightarrow C_s(n)$
  - Caso mejor: cota inferior del algoritmo $\rightarrow C_i(n)$
  - Caso promedio: coste promedio $\rightarrow C_m(n)$
- Todas son funciones de la talla del problema
- El coste promedio es difícil de evaluar a priori
  - Es necesario conocer la distribución de probabilidad de la entrada
  - ¡No es la media de la cota inferior y de la cota superior!

Cotas superior e inferior

Buscar elemento

```
int buscar( int[] v, int n, int z ){
  for( int i = 0; i < n; i++ )
    if( v[i] == z )
      return i;
  return -1;
}
```

- La talla es `n`

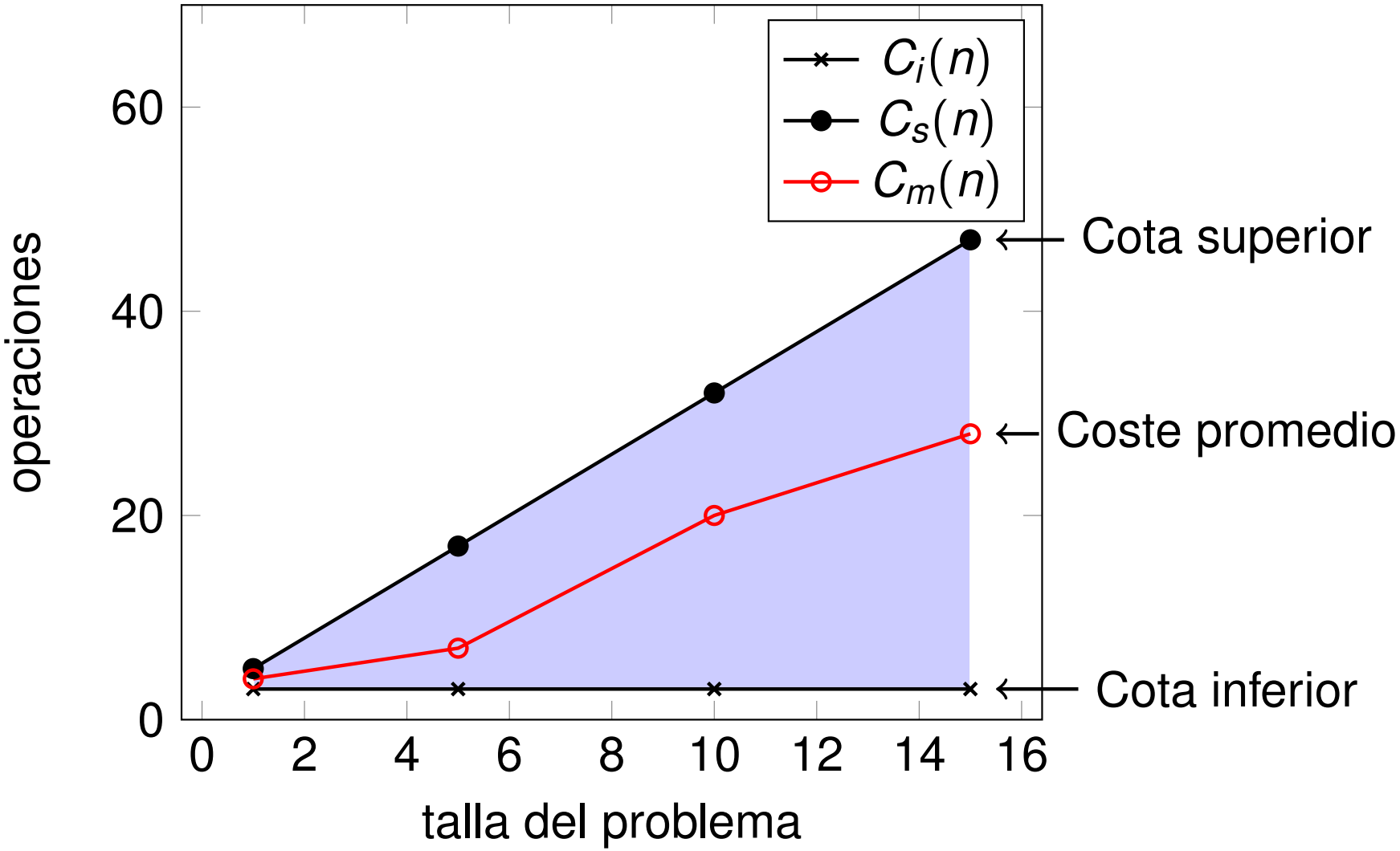
- **Mejor caso**: el elemento buscado está el primero
- **Peor caso**: el elemento buscado no está

Cotas:

$$C_s(n) = 1 + n$$
$$C_i(n) = 1$$

Cotas superior e inferior

- Coste de la función `buscar`



Problema introductorio #1

- ¿Cuál de estas dos funciones para buscar un elemento en un vector es más rápida? Asume que el vector tiene cientos de elementos. Sólo una opción es verdadera
  a. `buscar1` siempre es más rápida
  b. `buscar2` siempre es más rápida
  c. Una u otra, depende del contenido del vector
  d. `buscar2` nunca es más rápida que `buscar1`
  e. `buscar1` nunca es más rápida que `buscar2`

```
int buscar1( int[] vec, int n,
     int z ){
  for( int i = 0; i < n i++ )
    if( vec[i] == z )
      return i;
  return -1;
}
```

```
int buscar2( int[] vec, int n,
     int z ){
  int posicion=-1;
  for( int i = 0; i < n; i++ )
    if( vec[i] == z )
      posicion=i;
  return posicion;
}
```

Problema introductorio #1

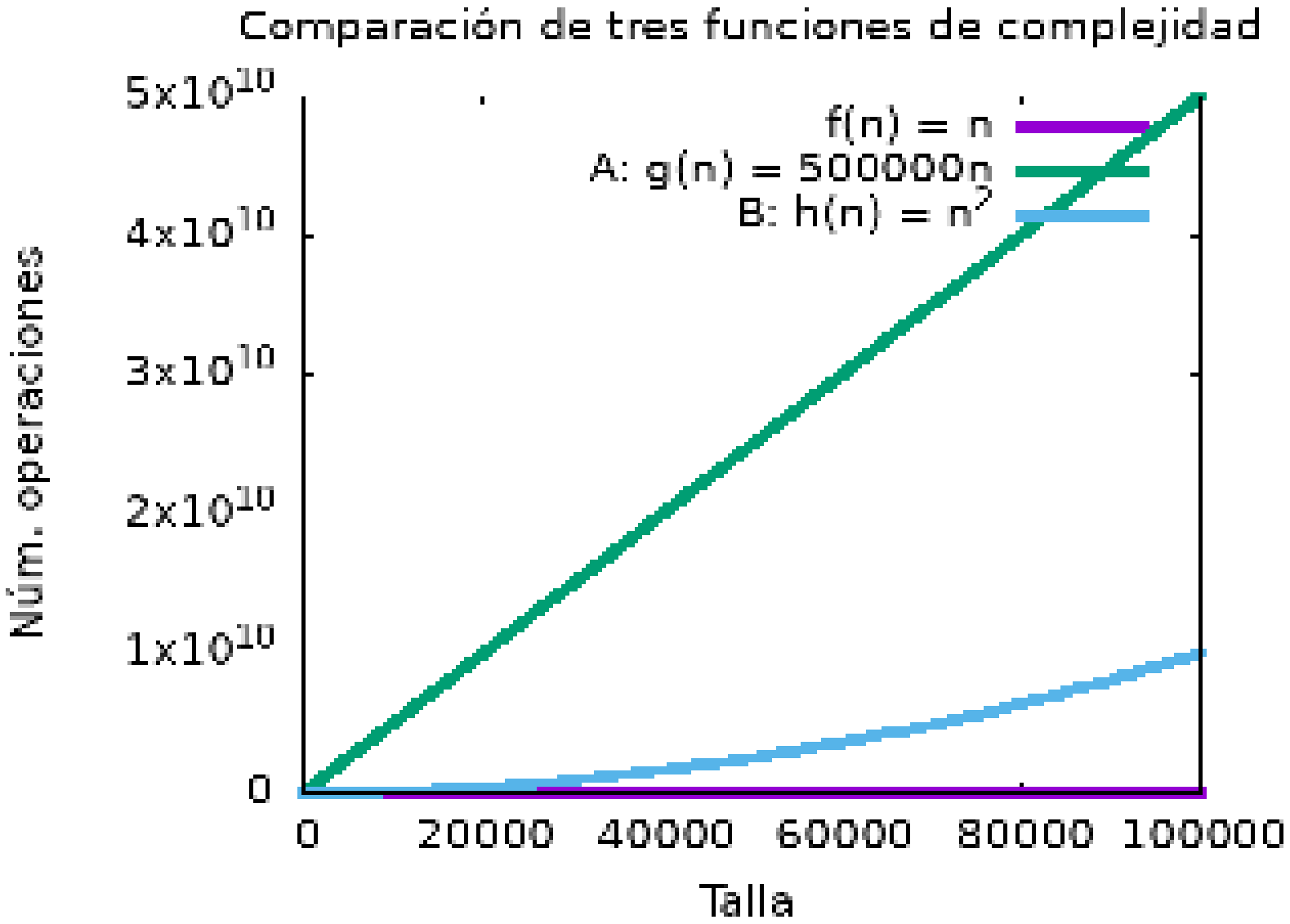
- ¿Cuál de estas dos funciones para buscar un elemento en un vector es más rápida? Asume que el vector tiene cientos de elementos. Sólo una opción es verdadera
  a. `buscar1` siempre es más rápida
  b. `buscar2` siempre es más rápida
  c. Una u otra, depende del contenido del vector
  d. **`buscar2` nunca es más rápida que `buscar1`**
  e. `buscar1` nunca es más rápida que `buscar2`

```
int buscar1( int[] vec, int n,
     int z ){
  for( int i = 0; i < n i++ )
    if( vec[i] == z )
      return i;
  return -1;
}
```

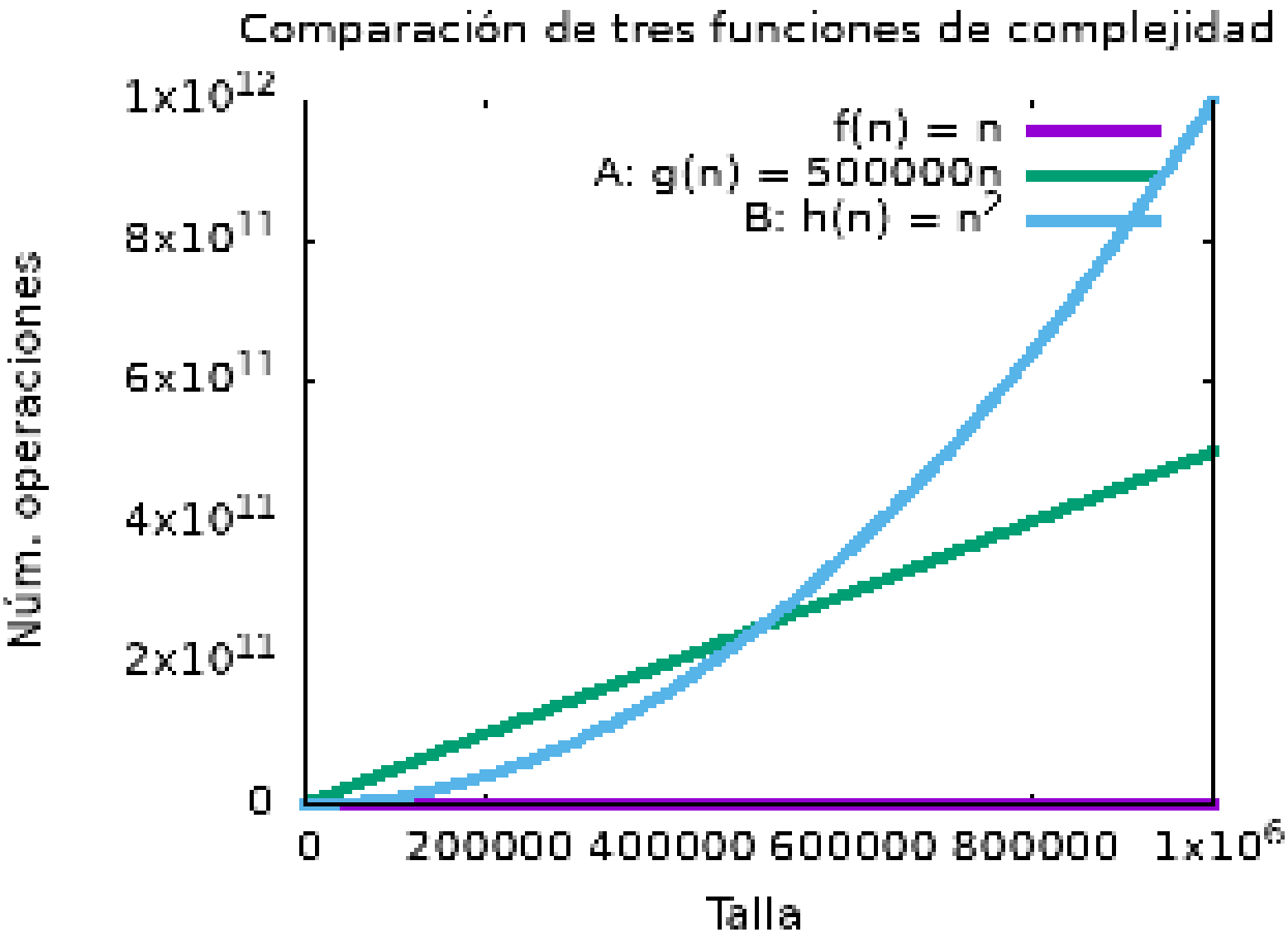
```
int buscar2( int[] vec, int n,
     int z ){
  int posicion=-1;
  for( int i = 0; i < n; i++ )
    if( vec[i] == z )
      posicion=i;
  return posicion;
}
```

Índice

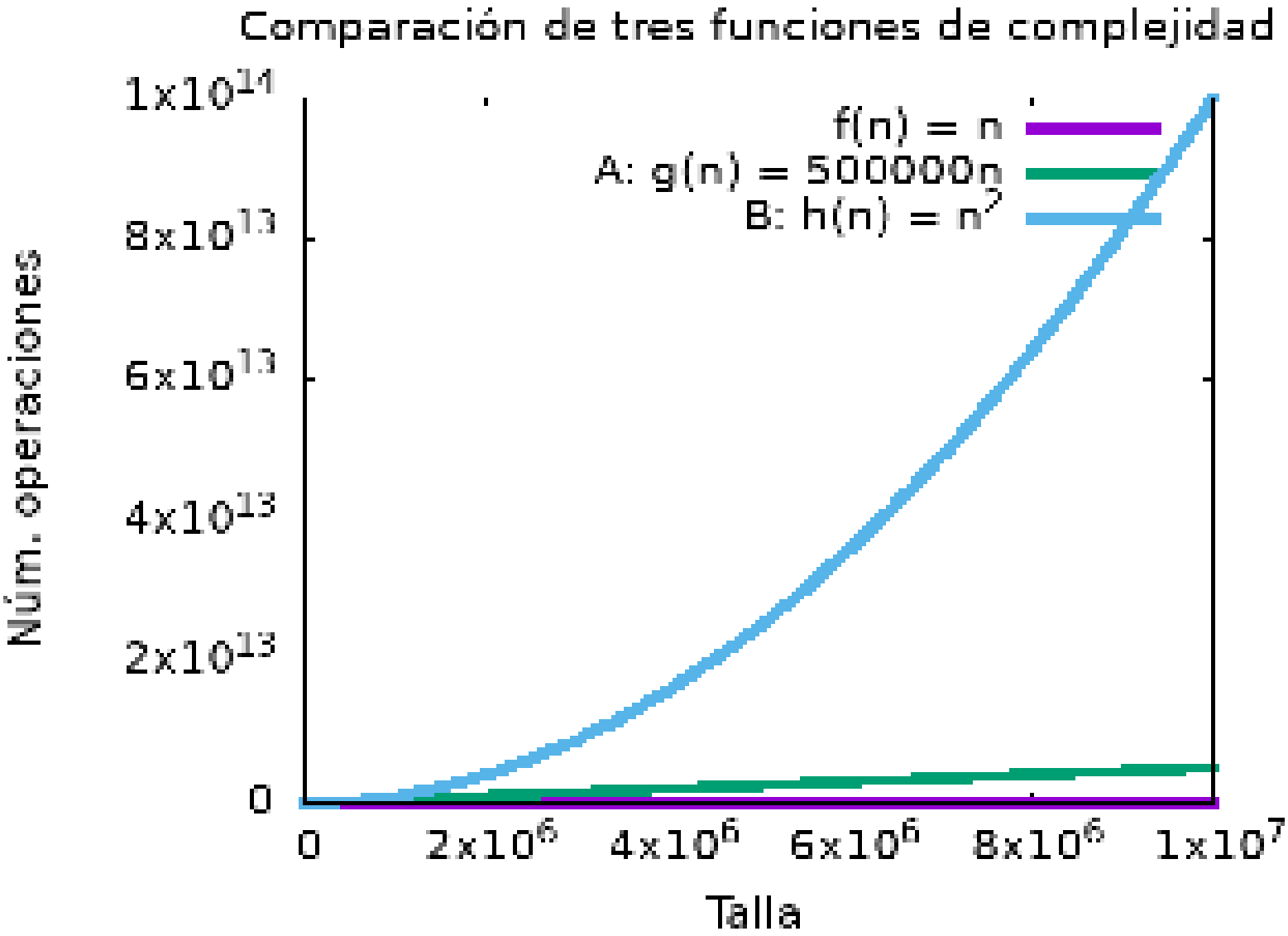
Análisis asintótico



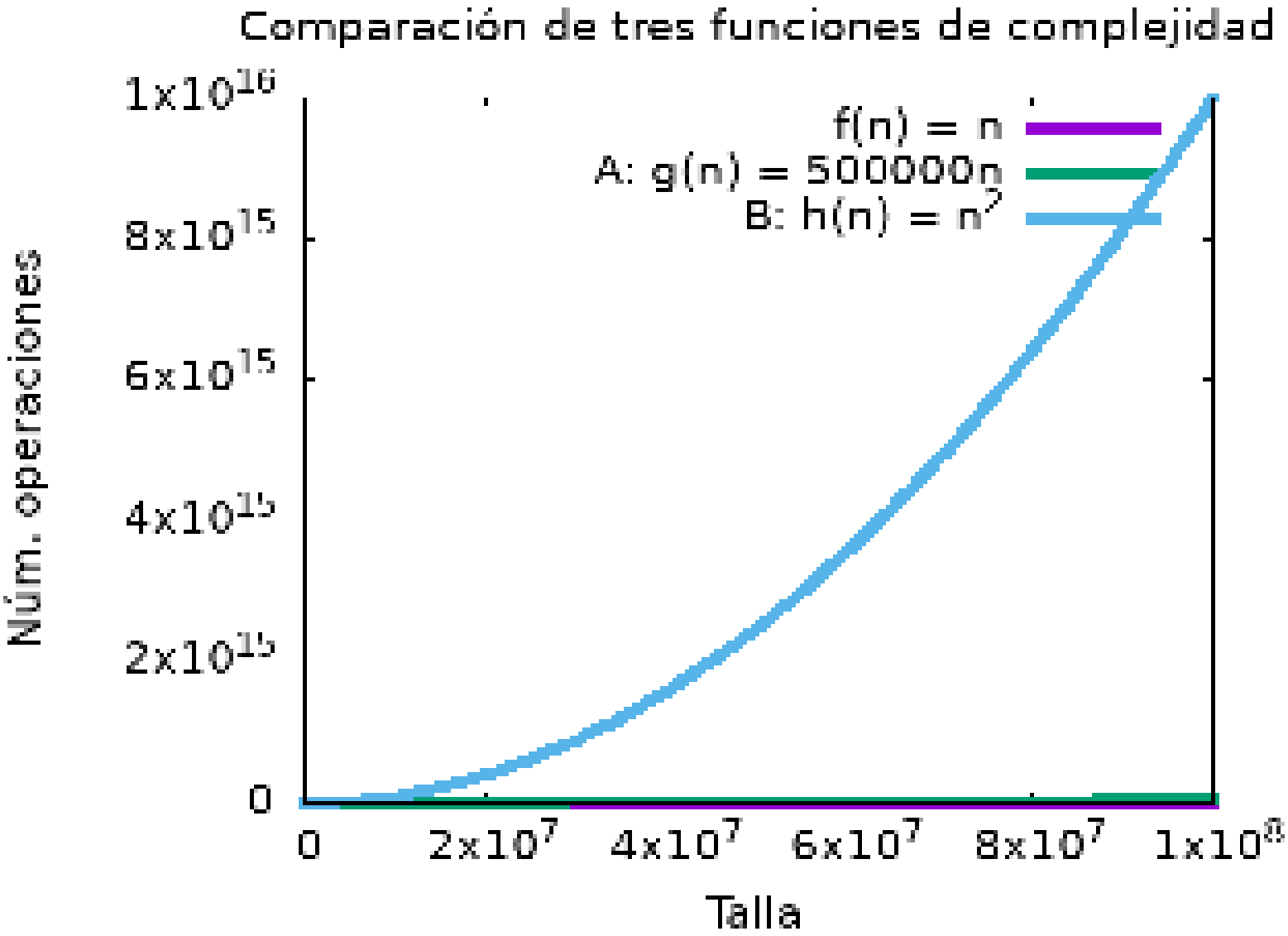
Análisis asintótico



Análisis asintótico



Análisis asintótico



Análisis asintótico

- El estudio de la complejidad resulta interesante para tamaños grandes de problema:
  - Diferencias en tiempo de ejecución de algoritmos con diferente coste para tamaños pequeños del problema son muy pequeñas
  - Es lógico invertir tiempo en el desarrollo de un buen algoritmo sólo si se prevé que éste realizará un gran volumen de operaciones
- Análisis asintótico: estudio de la complejidad para tamaños grandes de problema
  - Permite clasificar las funciones de complejidad de forma que podamos compararlas entre si fácilmente
  - Se definen clases de equivalencia que engloban a las funciones que “crecen de la misma forma”.
- Se emplea la notación asintótica

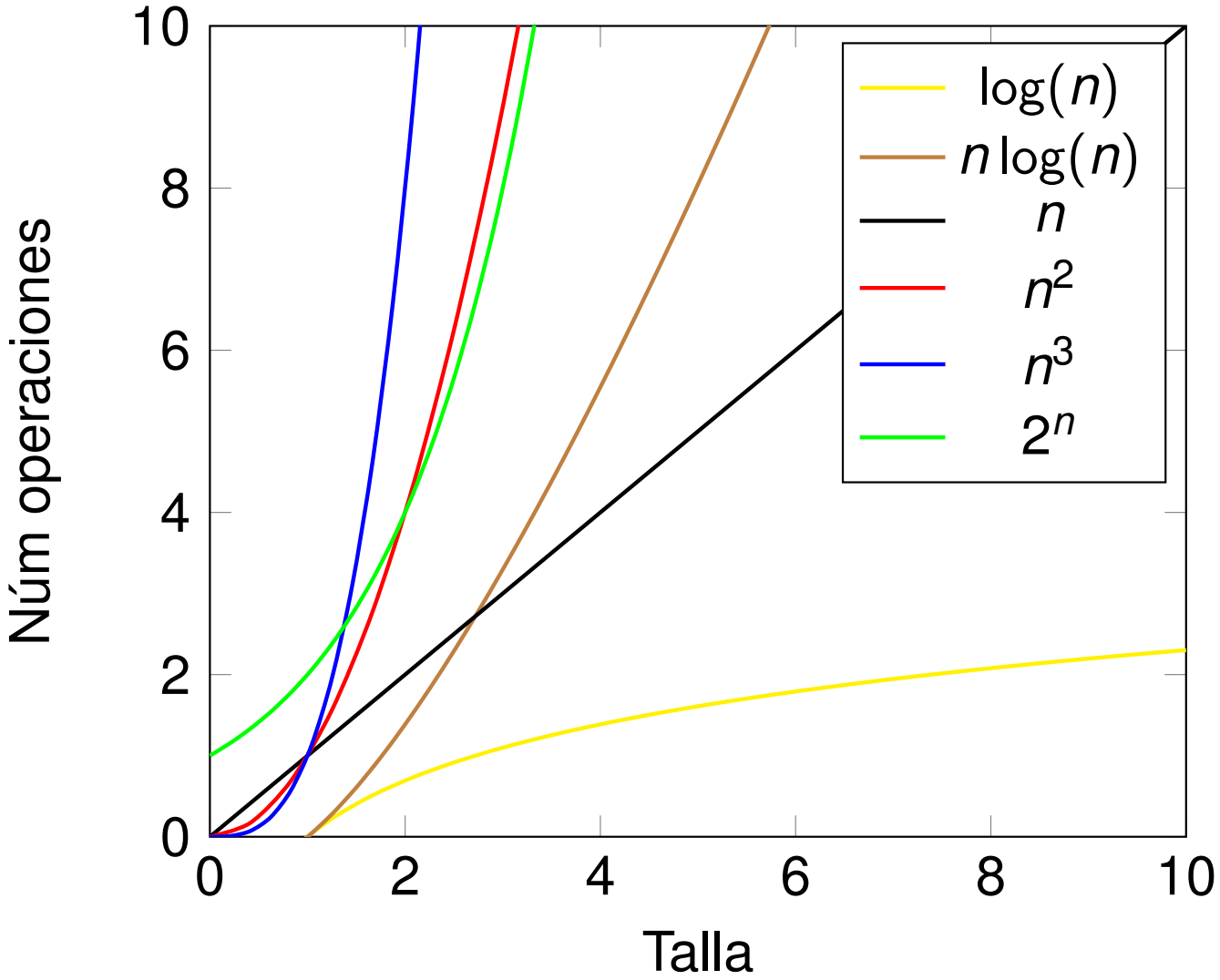
Análisis asintótico

Notación asintótica:

- Notación matemática utilizada para representar la complejidad cuando el tamaño de problema (n) crece ($n \to \infty$)
- Se definen tres tipos de notación:
  - Notación O (ómicron mayúscula o *big omicron*) $\Rightarrow$ cota superior
  - Notación Ω (omega mayúscula o *big omega*) $\Rightarrow$ cota inferior
  - Notación Θ (zeta mayúscula o *big theta*)$\Rightarrow$ coste exacto

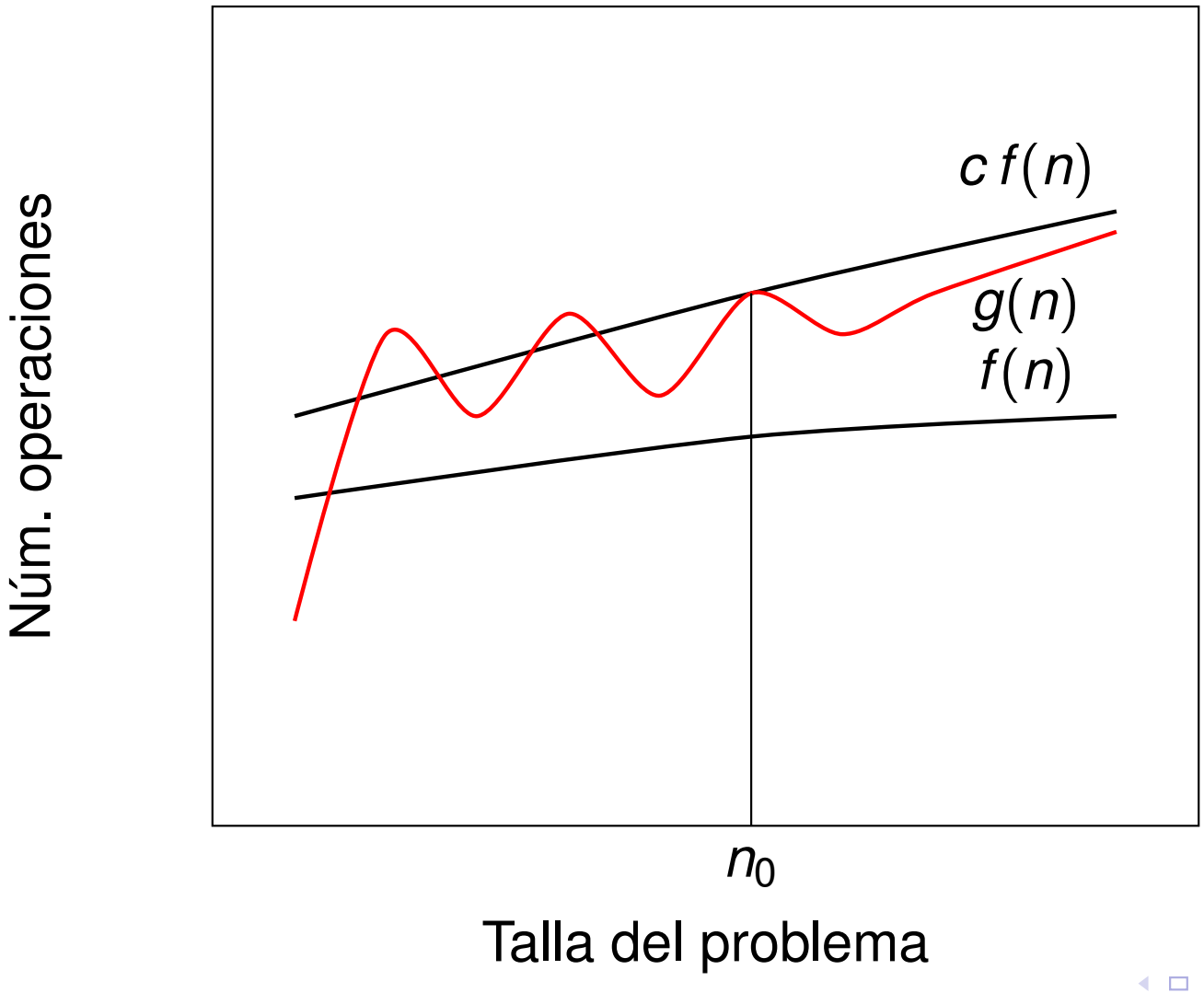
Análisis asintótico

- Permite agrupar en clases funciones con el mismo crecimiento



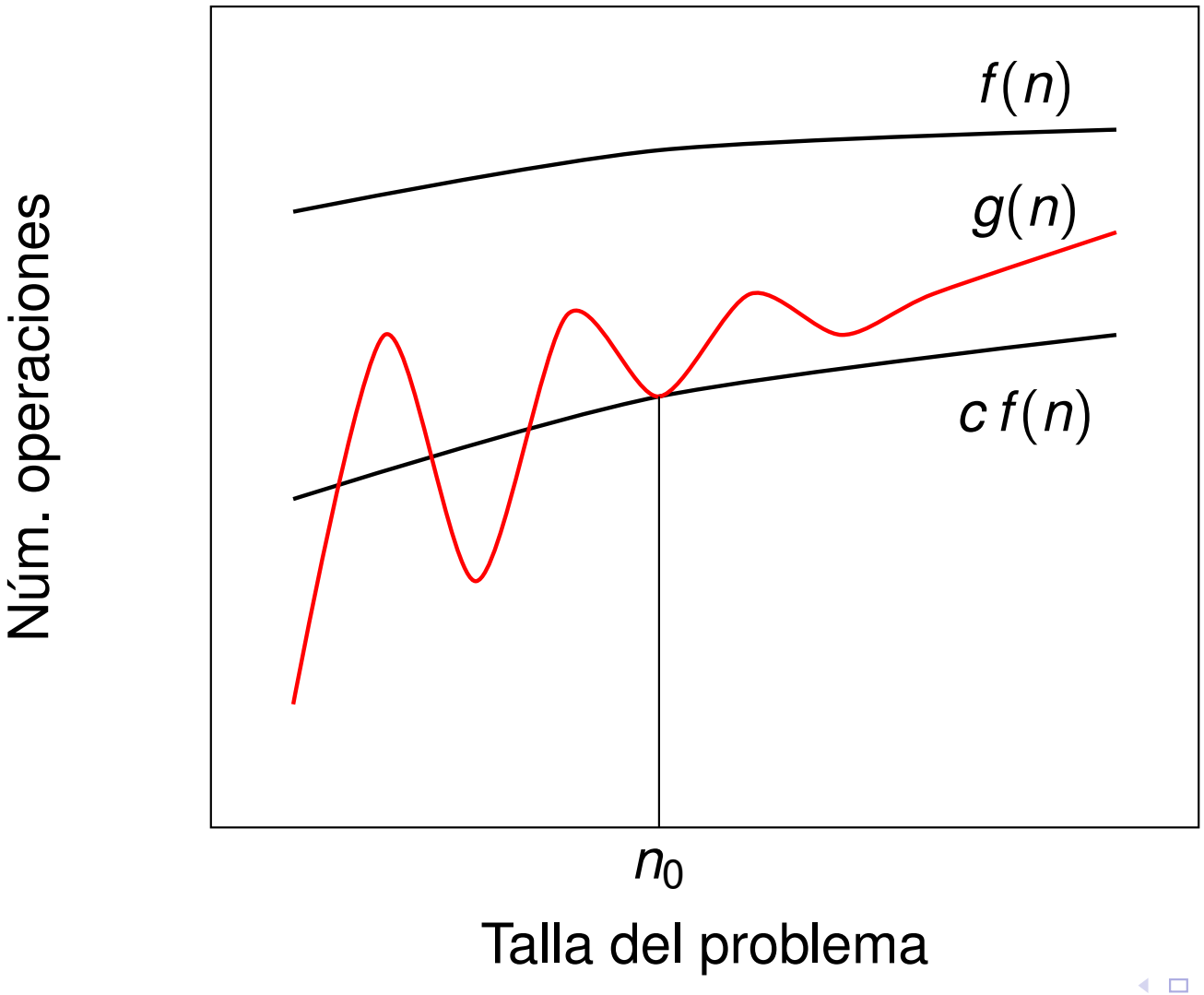
Cota superior. Notación O

- Sea $f : \mathbb{N} \to \mathbb{R}^+$; se define el conjunto $O(f)$ como el conjunto de funciones acotadas superiormente por un múltiplo de f



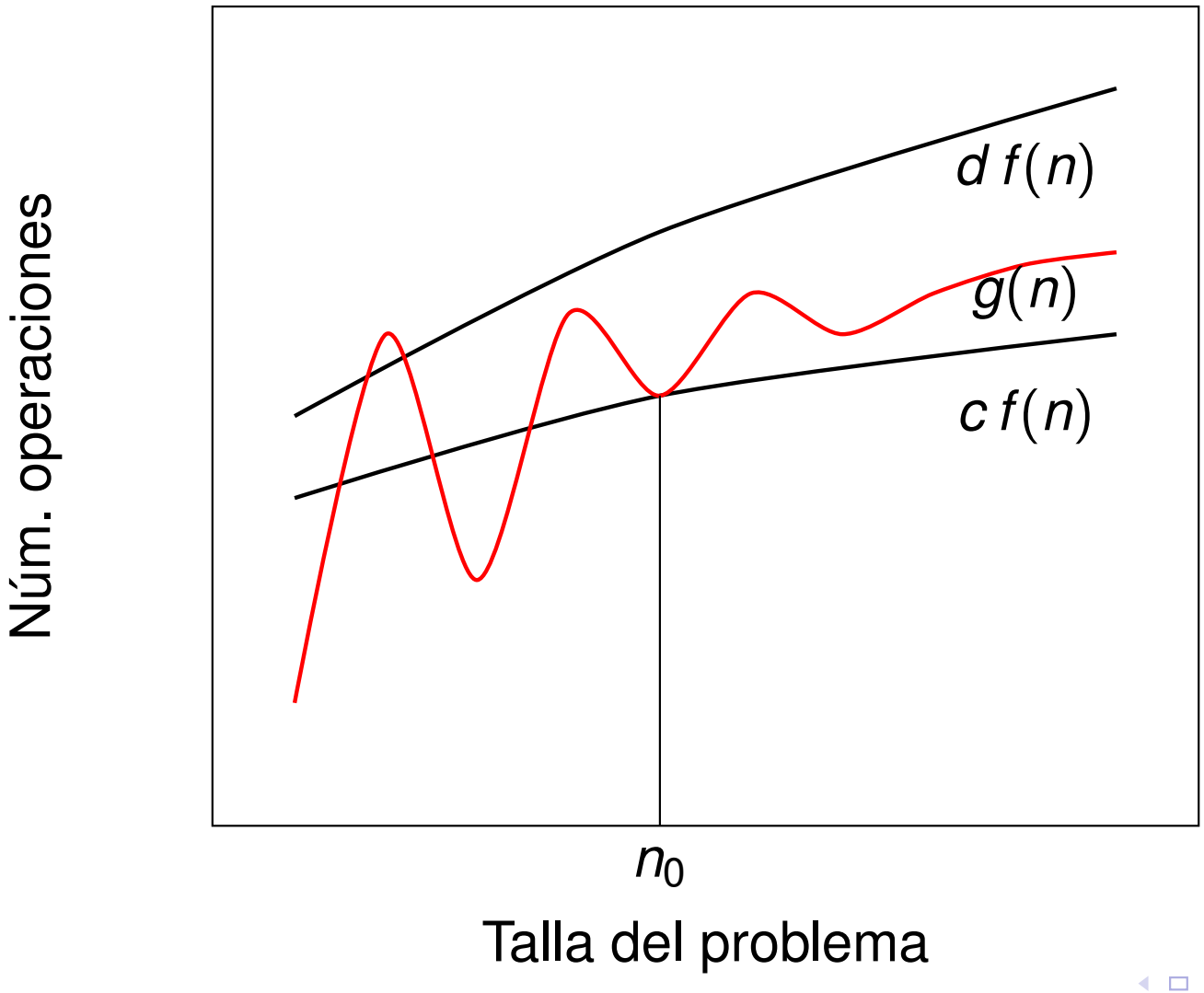
Cota inferior. Notación Ω

- Sea $f : \mathbb{N} \to \mathbb{R}^+$; se define el conjunto $\Omega(f)$ como el conjunto de funciones acotadas inferiormente por un múltiplo de f



Coste exacto. Notación Θ

- Sea $f : \mathbb{N} \to \mathbb{R}^+$; se define el conjunto $\Theta(f)$ como el conjunto de funciones acotadas superior e inferiormente por un múltiplo de f



Propiedades

Propiedades (también válidas para Ω y Θ):

- $f_1 \in O(g_1) \Rightarrow k \cdot f_1 \in O(g_1)$
- $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(\text{máx}\{g_1, g_2\})$

Ejemplos

- $1000n + 1 \in O(n)$
- $n^2 + \log n \in O(n^2)$
- $n^3 + 2^n + n \log n \in O(2^n)$

Jerarquías

- Las clases más utilizadas son:

$$\underbrace{O(1)}_{\text{constantes}} \subset \underbrace{O(\log\log n)}_{\text{sublogarítmicas}} \subset \underbrace{O(\log n) \subset O(log^{a(>1)} n)}_{\text{logarítmicas}}$$

$$\subset \underbrace{O(\sqrt{n})}_{\text{sublineales}} \subset \underbrace{O(n)}_{\text{lineales}} \subset \underbrace{O(n \log n)}_{\text{lineal-logarítmicas}}$$

$$\subset \underbrace{O(n^2) \subset O(n^{a(>2)})}_{\text{polinómicas}} \subset \underbrace{O(2^n)}_{\text{exponenciales}} \subset \underbrace{O(n!) \subset O(n^n)}_{\text{superexponenciales}}$$

Índice

Cálculo de complejidades

1. Determinar la talla: variable de la función de complejidad que se pretende encontrar. Puede ser:
   - Tamaño del problema
   - Parámetro representativo
2. Determinar los casos mejor y peor (instancias para las que el algoritmo tarda más o menos)
   - Para algunos algoritmos, el caso mejor y el caso peor son el mismo ya que se comportan igualmente para cualquier instancia del mismo tamaño
3. Obtención de las cotas para cada caso, expresadas como clases de notación asintótica

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de `s = s.length()`

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de `s` = `s.length()`
2. Casos mejor y peor:

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de `s = s.length()`
2. Casos mejor y peor:
   - Mejor caso: primer y último carácter son distintos

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de `s = s.length()`
2. Casos mejor y peor:
   - Mejor caso: primer y último carácter son distintos
   - Peor caso: es palíndromo
3. Cotas:
   - Mejor caso:

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de `s` = `s.length()`
2. Casos mejor y peor:
   - Mejor caso: primer y último carácter son distintos
   - Peor caso: es palíndromo
3. Cotas:
   - Mejor caso: $c_i(n) = 1 \rightarrow c_i(n) \in \Omega(1)$
   - Peor caso:

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de `s = s.length()`
2. Casos mejor y peor:
   - Mejor caso: primer y último carácter son distintos
   - Peor caso: es palíndromo
3. Cotas:
   - Mejor caso: $c_i(n) = 1 \rightarrow c_i(n) \in \Omega(1)$
   - Peor caso: $c_s(n) = n/2 \rightarrow c_s(n) \in O(n)$

Problema introductorio #2

- ¿Cuál de estas dos funciones es más rápida para vectores muy grandes? Sólo una opción es verdadera
  a. `maximo1` es más rápida
  b. `maximo2` es más rápida
  c. Una u otra, depende del contenido del vector
  d. `maximo2` nunca es más rápida que `maximo1`
  e. `maximo1` nunca es más rápida que `maximo2`

```
int maximo1 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<500000; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

```
int maximo2 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<n; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

Problema introductorio #2

- ¿Cuál de estas dos funciones es más rápida para vectores muy grandes? Sólo una opción es verdadera
  a. `maximo1` es más rápida
  b. **`maximo2` es más rápida**
  c. Una u otra, depende del contenido del vector
  d. `maximo2` nunca es más rápida que `maximo1`
  e. `maximo1` nunca es más rápida que `maximo2`

```
int maximo1 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<500000; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

```
int maximo2 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<n; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

Ejemplos

- La búsqueda binaria permite buscar rápidamente un elemento en un array si sabemos que está ordenado

Búsqueda binaria

```
int busca(int [] v, int n, int z){
    int m;
    int pri=0; int ult=n-1;
    do{
        m=(pri+ult)/2;
        if(v[m] > z)
            ult=m-1;
        else
            pri=m+1;
    } while(pri <= ult && v[m] != z )
    if( v[m] == z)
        return m;
    else
        return -1;
}
```

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $= \mathtt{n}$
2. Casos mejor y peor:

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`
   - Peor caso:

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`
   - Peor caso: `z` no está
3. Cotas:
   - Mejor caso:

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`
   - Peor caso: `z` no está
3. Cotas:
   - Mejor caso: $c_i(n) = 1 \rightarrow c_i(n) \in \Omega(1)$
   - Peor caso:

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`
   - Peor caso: `z` no está
3. Cotas:
   - Mejor caso: $c_i(n) = 1 \rightarrow c_i(n) \in \Omega(1)$
   - Peor caso: $c_s(n) = 1 + m \cdot 1$; $m =$ número iteraciones del bucle

Ejemplos

Cálculo de complejidad para búsqueda binaria

iteración	tam. array (`ult-pri+1`)
1	n
2	$\frac{n}{2}$
3	$\frac{n}{4}$
...	..
i	$\frac{n}{2^{i-1}}$
...	...
m	1

En la última iteración, el tamaño del array es 1, luego

$$\frac{n}{2^{m-1}} = 1$$

$$n = 2^{m-1}$$

$$\log_2 n = m - 1$$

$$m = log_2 n + 1$$

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`
   - Peor caso: `z` no está
3. Cotas:
   - Mejor caso: $c_i(n) = 1 \rightarrow c_i(n) \in \Omega(1)$
   - Peor caso: $c_s(n) = 1 + m \cdot 1 = 1 + log_2 n + 1 \rightarrow c_s(n) \in O(\log n)$

Ejercicios

Calcula la complejidad del siguiente fragmento de código (talla $=$ valor de `n`):

Ejercicio 4:

```
for(i = 0, length = 1; i < n-1; i++) {
   for(i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++, i2++);
   if(length < i2 - i1 + 1) length = i2 - i1 + 1; }
```

Ejercicios

Calcula la complejidad del algoritmo de ordenación por inserción directa: *Insertion sort* en `https://visualgo.net/en/sorting`

Ejercicio 5:

```
int ordenacion_insercion(int [] v, int n){
    int x,j;
    for(int i =1; i < n; i++){
        x=v[i]; j=i-1;
        while(j >= 0 && v[j] > x){
            v[j+1]=v[j];
            j-=1
        }
        v[j+1]=x
    }
}
```

Ejercicios

Cálculo de complejidad para inserción directa

1. Talla: tamaño del array $= \mathtt{n}$
2. Casos mejor y peor:
   - Mejor caso: array ordenado ascendentemente
   - Peor caso: array ordenado descendentemente
3. Cotas:
   - Mejor caso: $c_i(n) = n - 1 \rightarrow c_i(n) \in \Omega(n)$
   - Peor caso: $c_s(n) = \sum_{i=1}^{n} \sum_{j=0}^{i-1} 1 \rightarrow c_s(n) \in O(n^2)$

Ejercicios

Calcula la complejidad del algoritmo de ordenación de la burbuja: *Bubble sort* en `https://visualgo.net/en/sorting`

Ejercicio 6:

```
int ordenacion_burbuja(int [] v, int n){
    for(int fin=n; fin >=2 ; fin--)
        for(int i=1; i < fin ; i++)
            if(v[i] < v[i-1]){
                swap(v[i],v[i-1]);
            }
    }
}
```

Ejercicios

Cálculo de complejidad para el algoritmo de la burbuja

1. Talla: tamaño del array $= \mathtt{n}$
2. Casos mejor y peor: no hay
3. Complejidad: $T(n) = \sum_{f=2}^{n} \sum_{i=1}^{f-1} 1 \rightarrow T(n) \in \Theta(n^2)$

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;
```

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;
```

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();
```

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()`		

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()`		
`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()`		

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)`		
`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()`		

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:

TAD cola

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

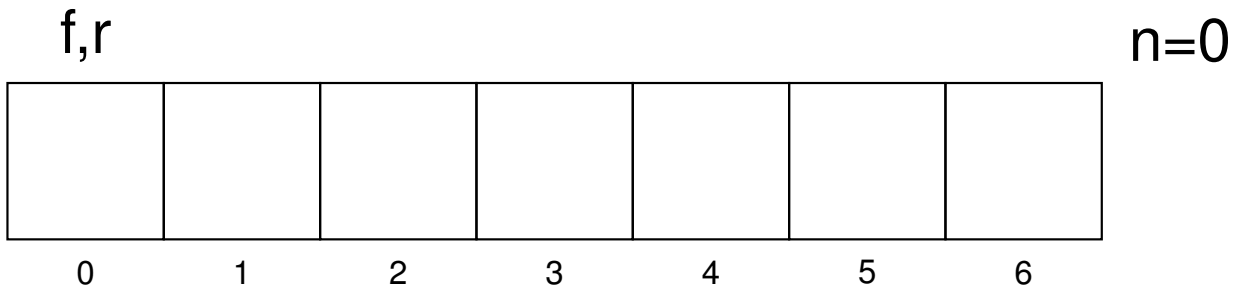
Pregunta

¿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?

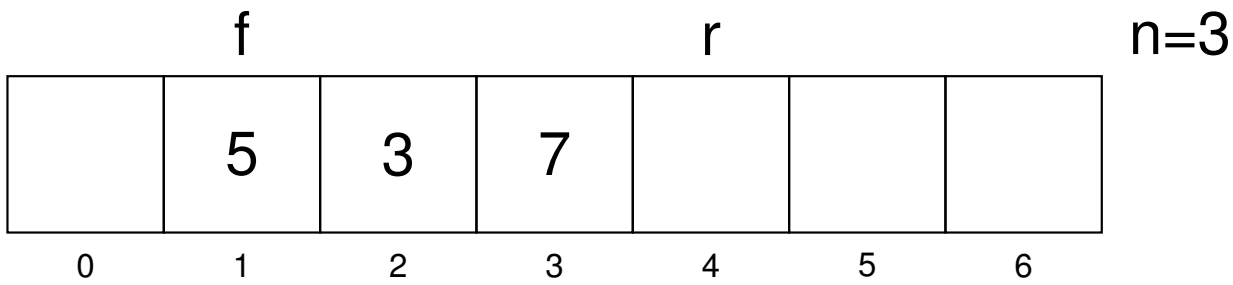
- `front`: $\Theta(1)$
- `enqueue`: $\Omega(1)$ (no hay redimensionamiento); $O(n)$ (hay redimensionamiento). El coste amortizado es $O(1)$
- `dequeue`: $\Theta(n)$

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

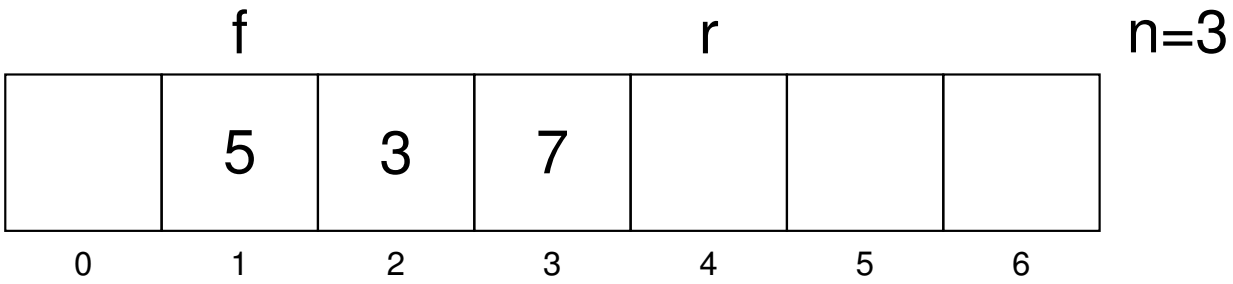
- `enqueue(e):array[r]=e; r++; n++`
- `dequeue(): f++; n--`
- `front(): array[f];`
- Complejidad:

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

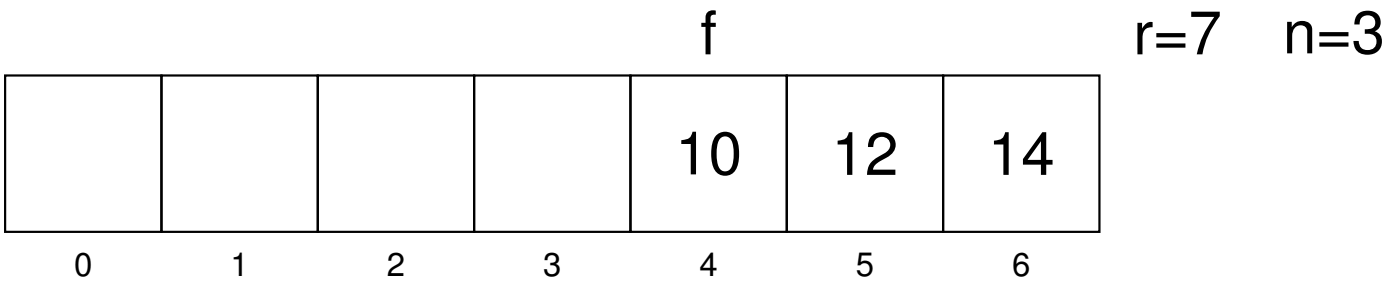
Ejercicio

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



Implementación eficiente de una cola

Solución:

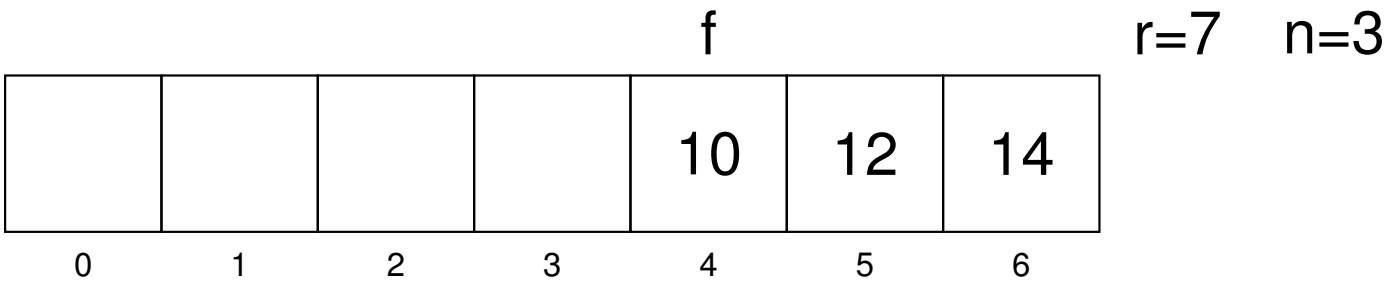


Pregunta

- ¿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?

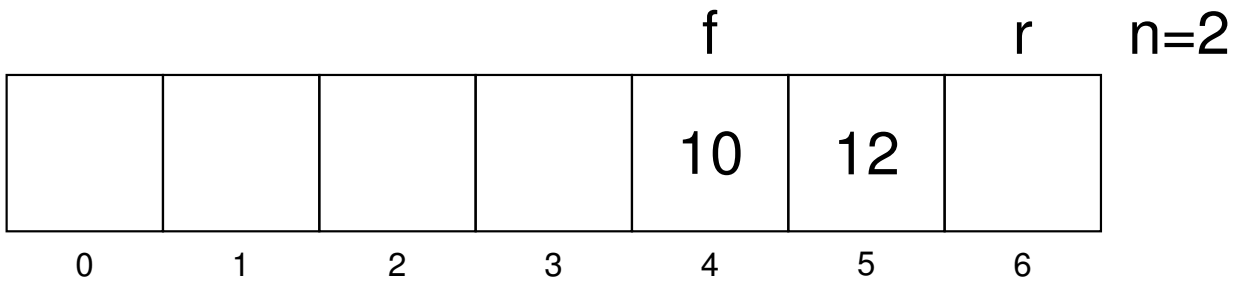
- n es menor que el tamaño del array (N), pero la condición de cola llena es $r == N$. No se puede insertar
- Solución: **colas circulares**

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

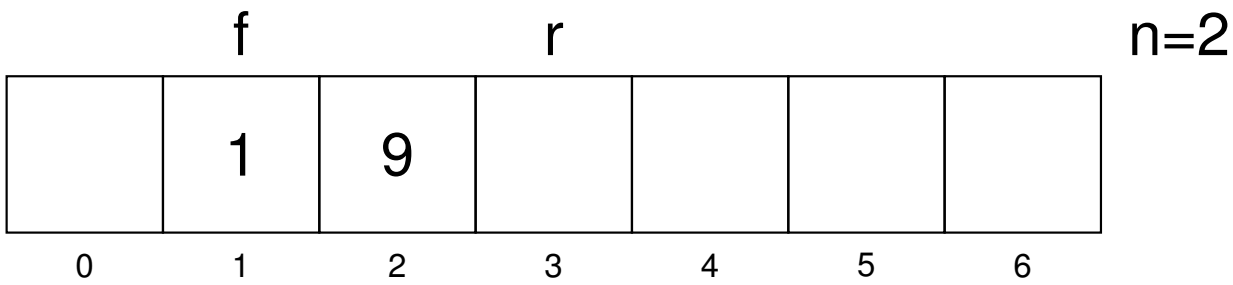
Ejercicio

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

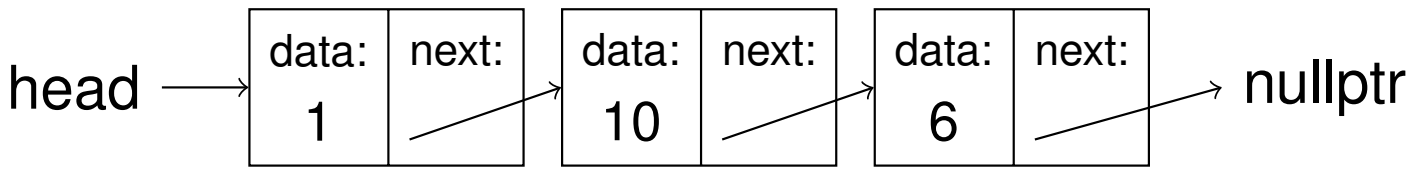
- Se declaran con `queue<TipoDeDato>`:

```
#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

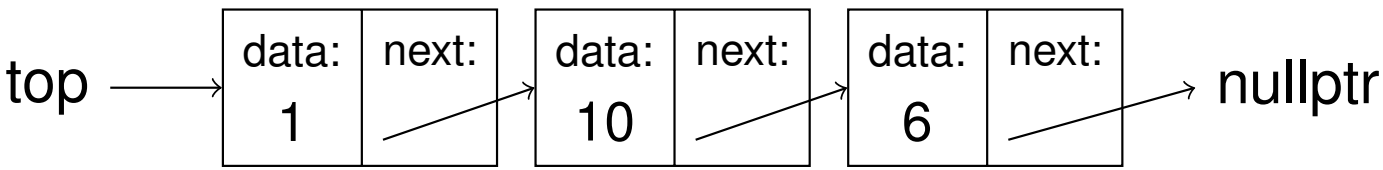
Implementación del TAD pila

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 de una pila con listas enlazadas

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

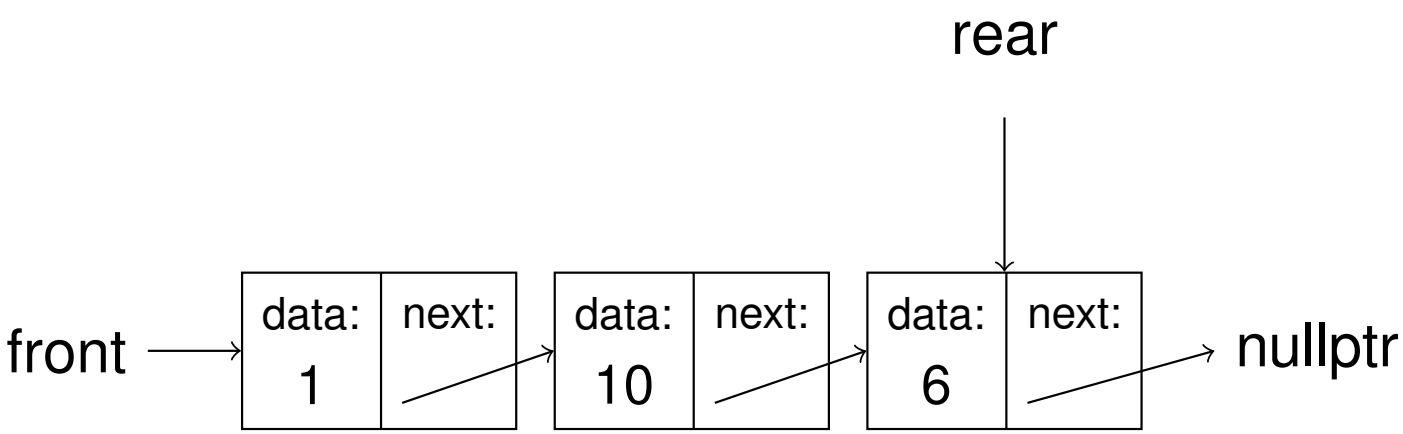
Implementación del TAD cola

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`: $\Theta(1)$

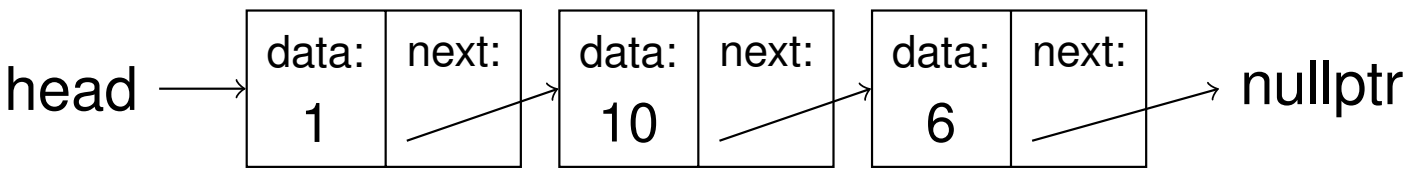
Pregunta

¿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

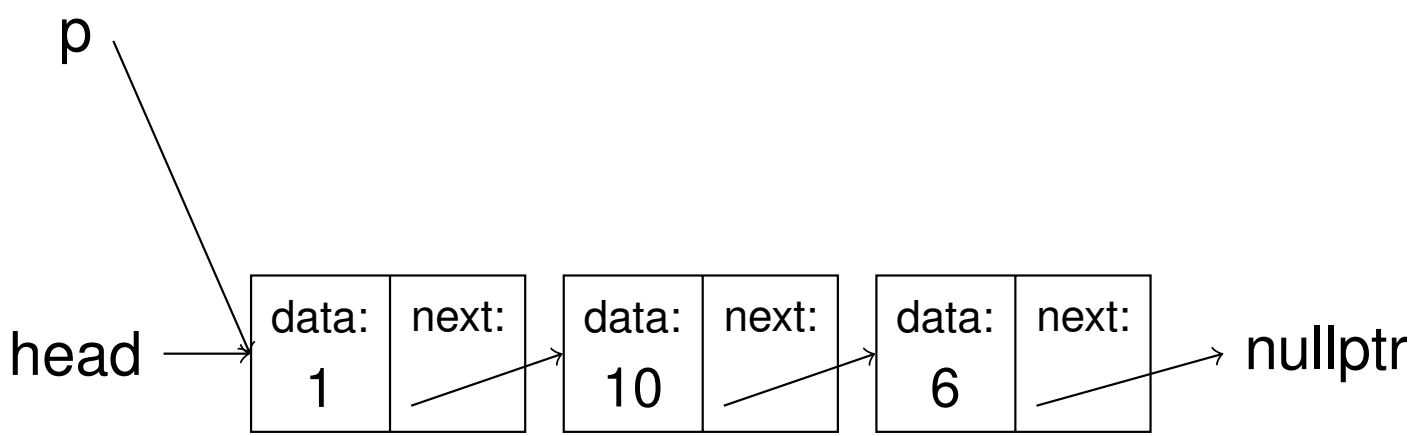
Iterando sobre los elementos de una lista enlazada

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



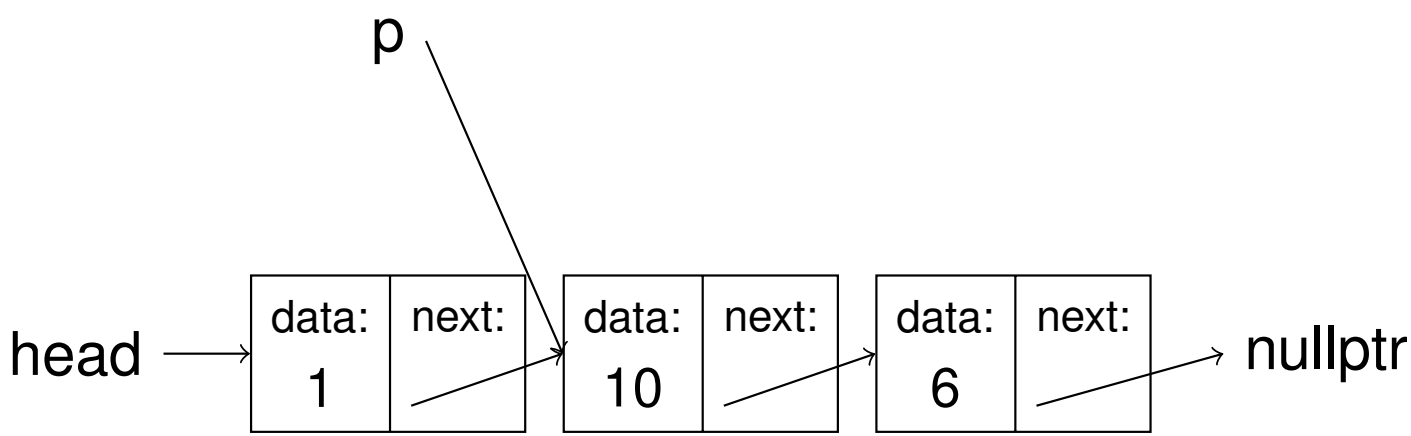
Iterando sobre los elementos de una lista enlazada

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



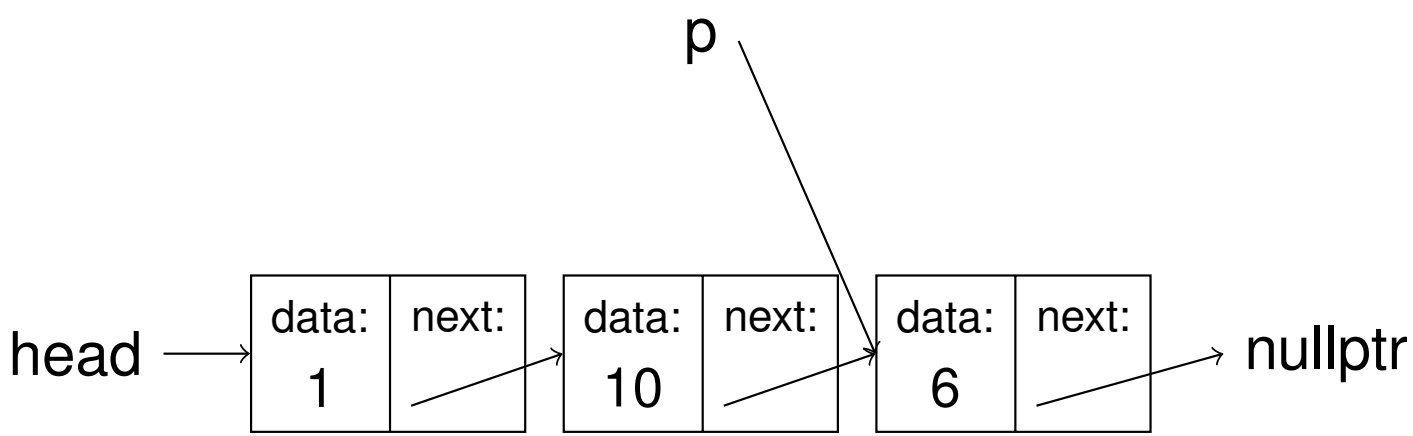
Iterando sobre los elementos de una lista enlazada

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



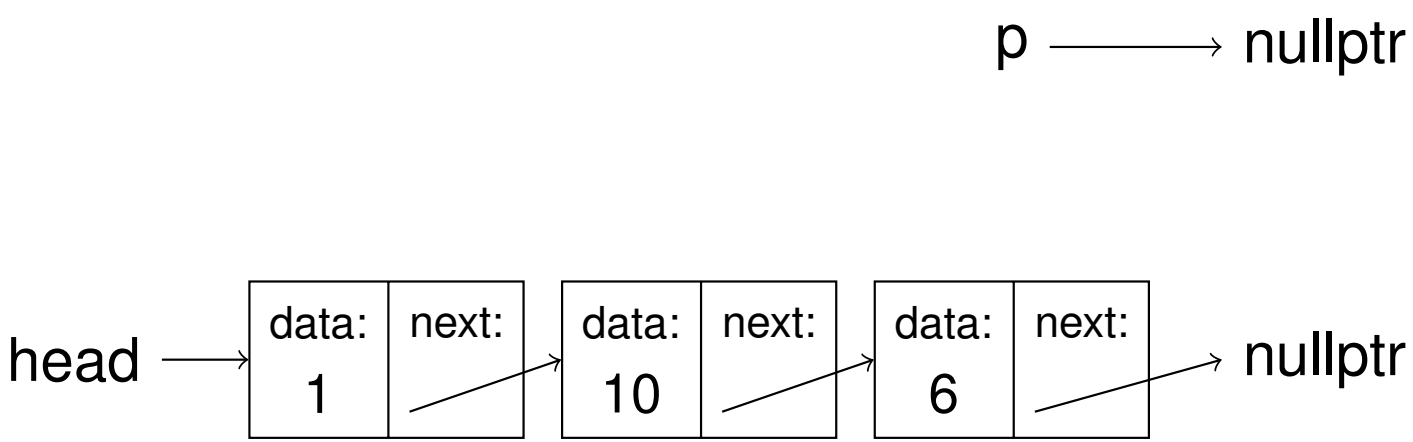
Iterando sobre los elementos de una lista enlazada

```
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

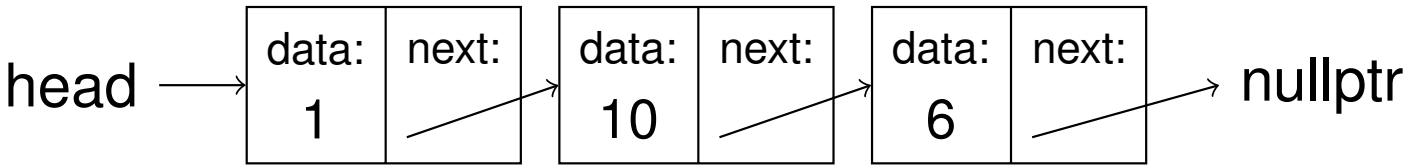
Implementación del TAD lista

- 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

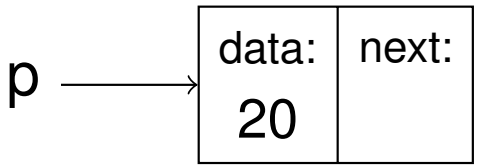
- Inserción al comienzo de la lista

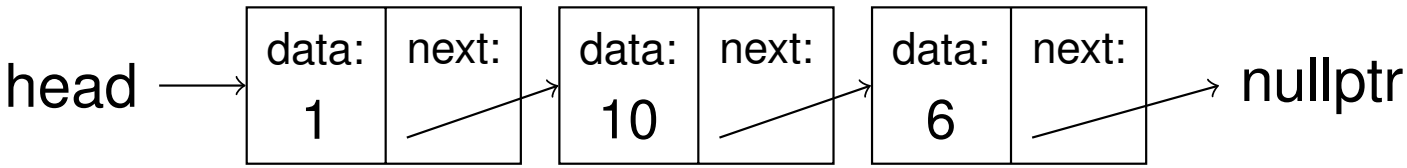


Implementación del TAD lista

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

- Inserción al comienzo de la lista

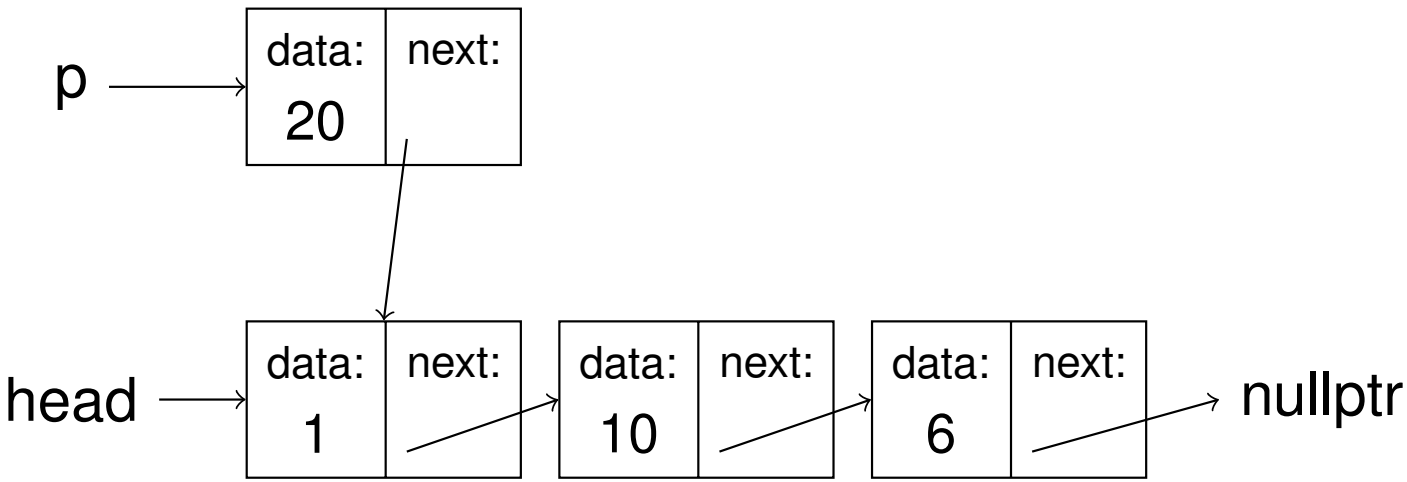




Implementación del TAD lista

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

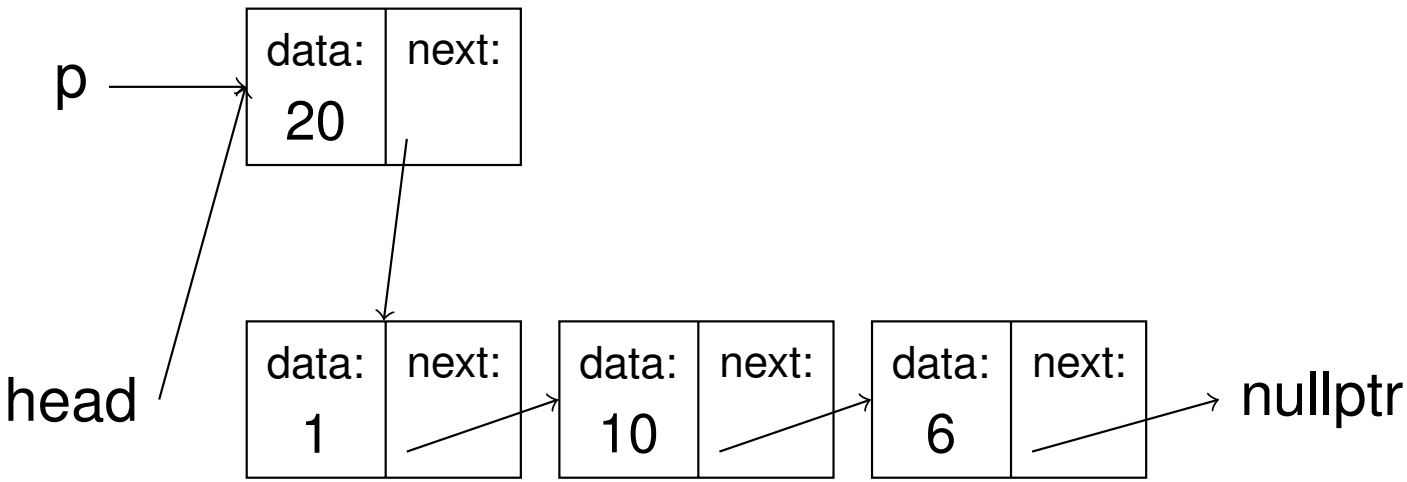
- Inserción al comienzo de la lista



Implementación del TAD lista

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

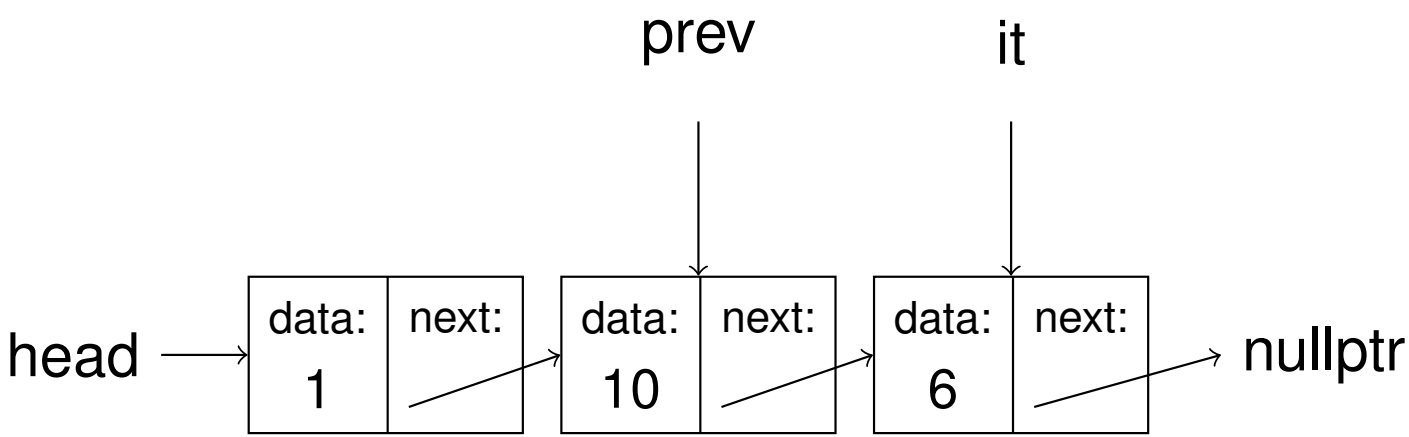
- Inserción al comienzo de la lista



Implementación del TAD lista

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

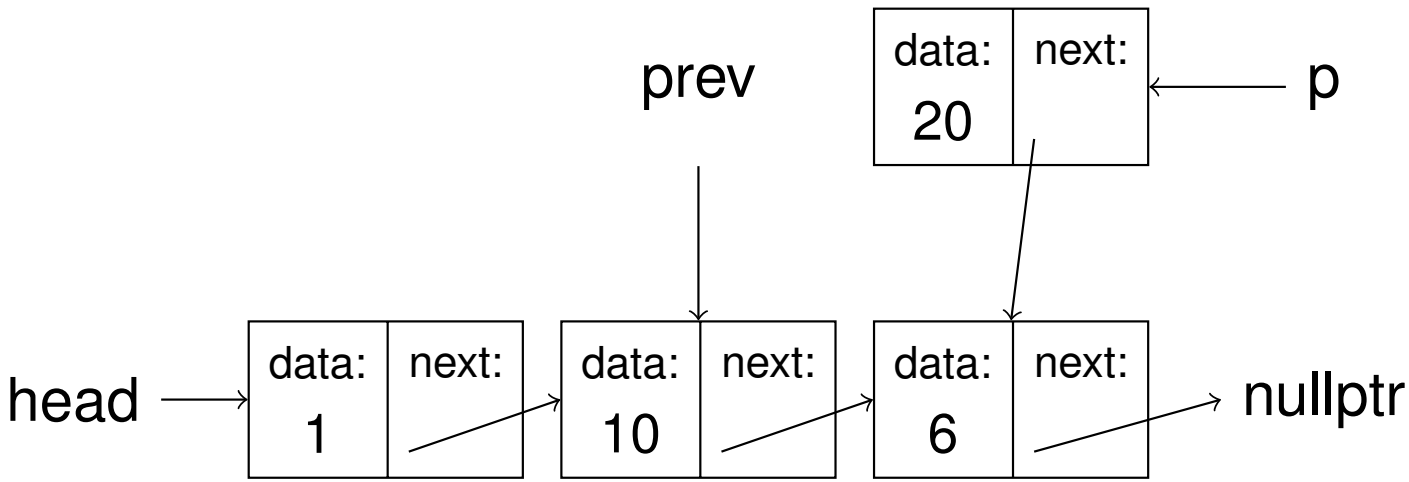
- Inserción en otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a donde vamos a insertar



Implementación del TAD lista

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

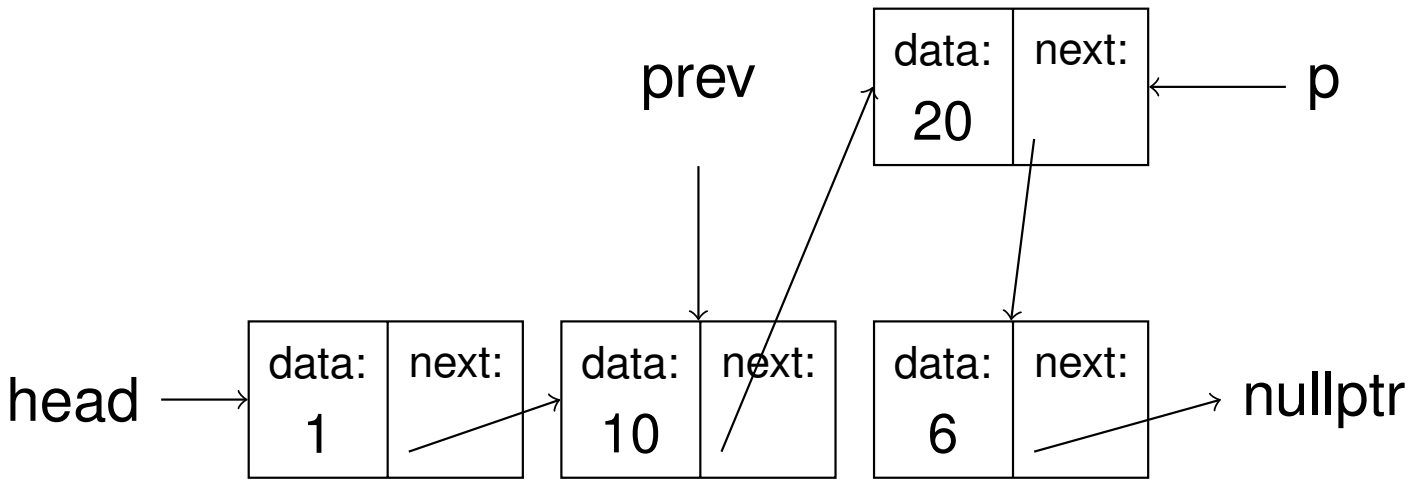
- Inserción en otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a donde vamos a insertar



Implementación del TAD lista

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

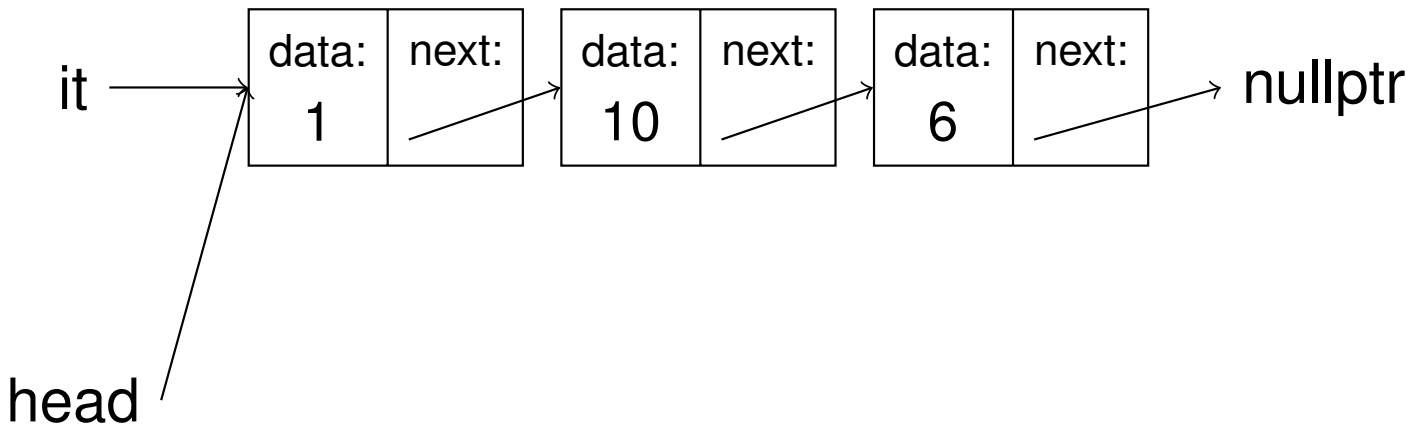
- Inserción en otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a donde vamos a insertar



Implementación del TAD lista

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

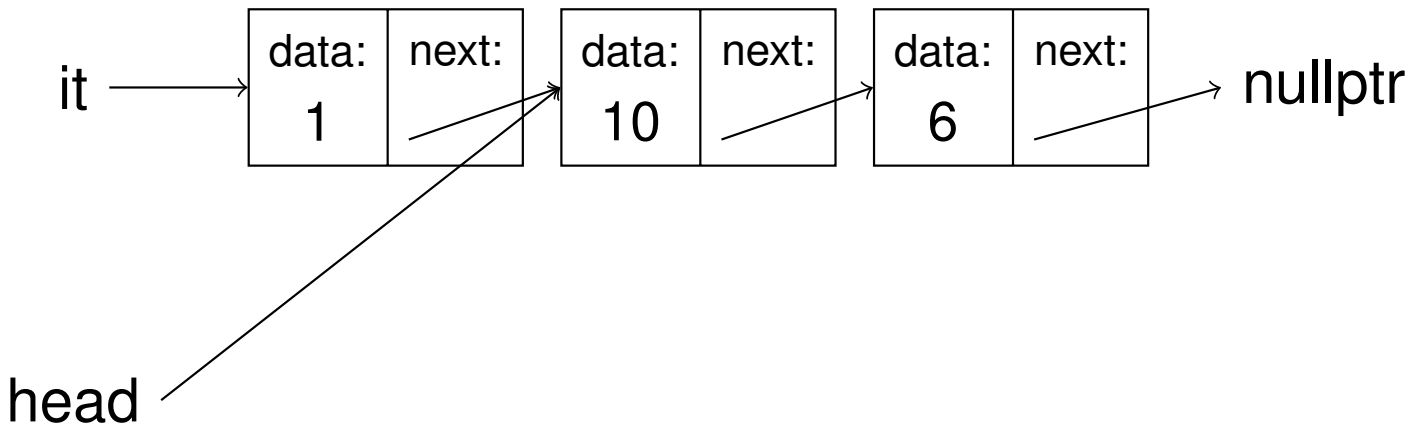
- Borrado de la primera posición



Implementación del TAD lista

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

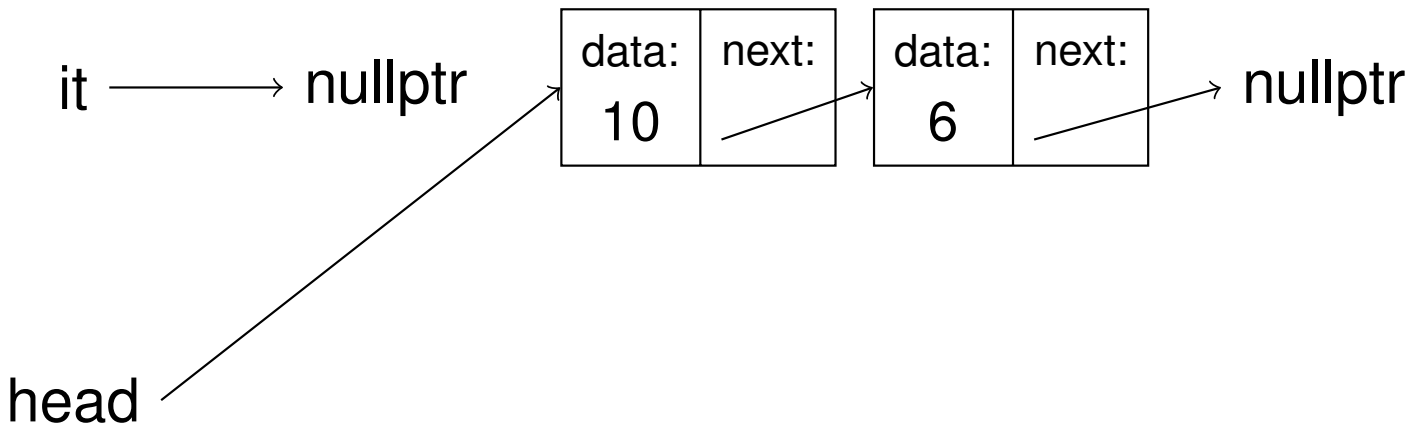
- Borrado de la primera posición



Implementación del TAD lista

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

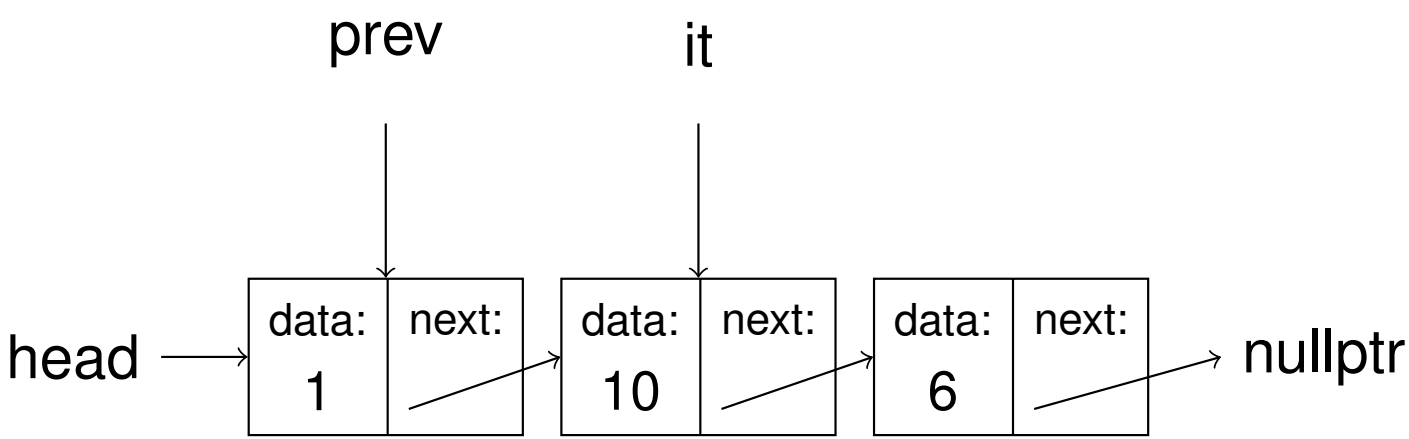
- Borrado de la primera posición



Implementación del TAD lista

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

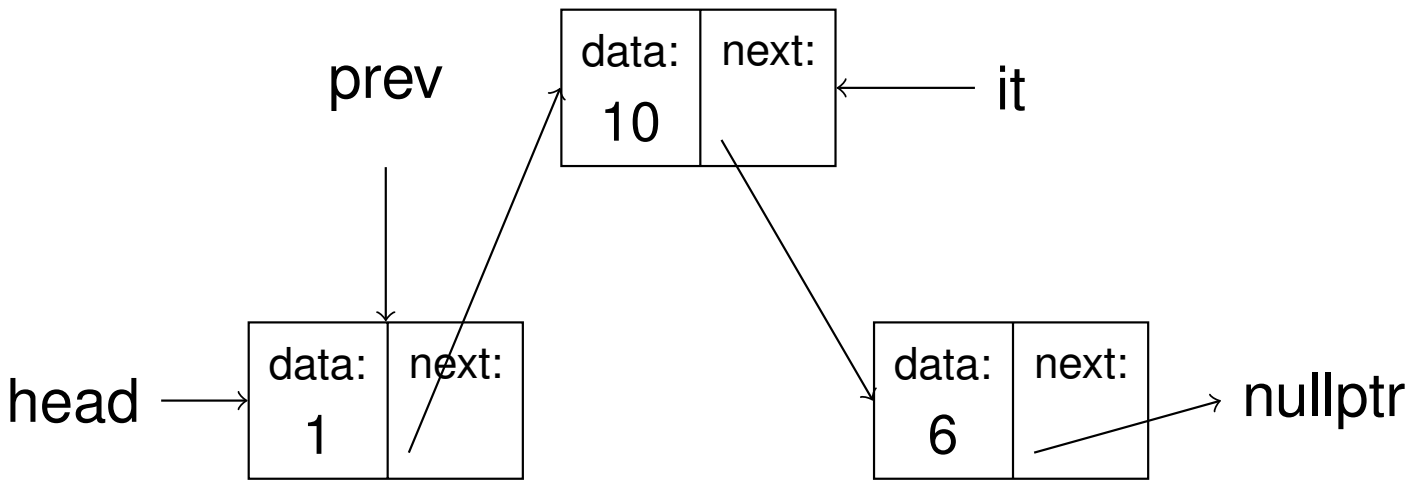
- Borrado de otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a la que vamos a borrar



Implementación del TAD lista

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

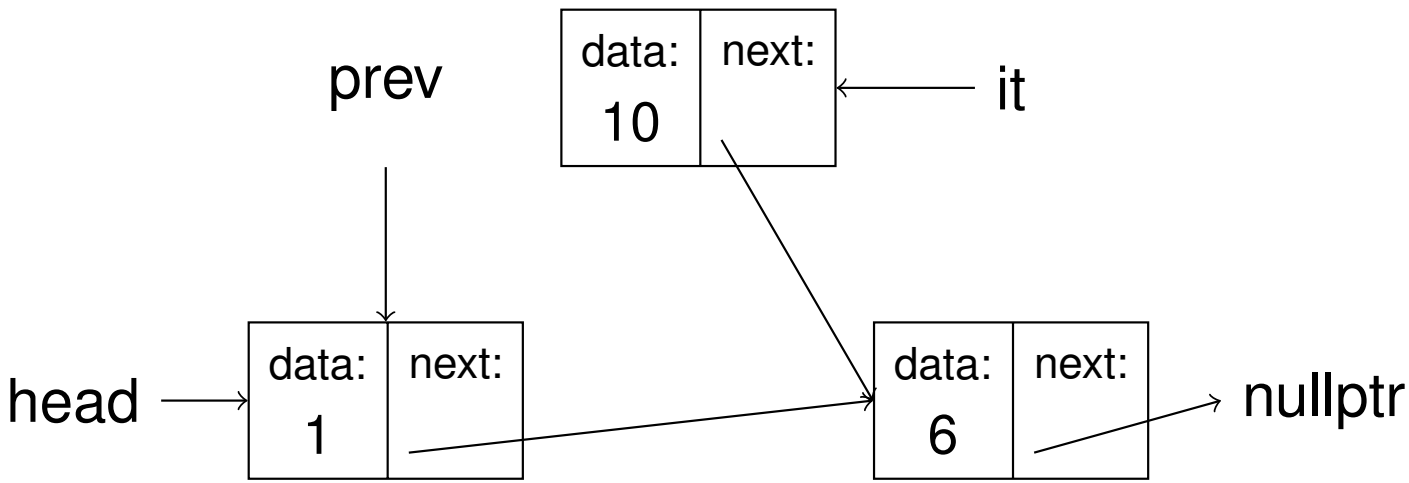
- Borrado de otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a la que vamos a borrar



Implementación del TAD lista

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

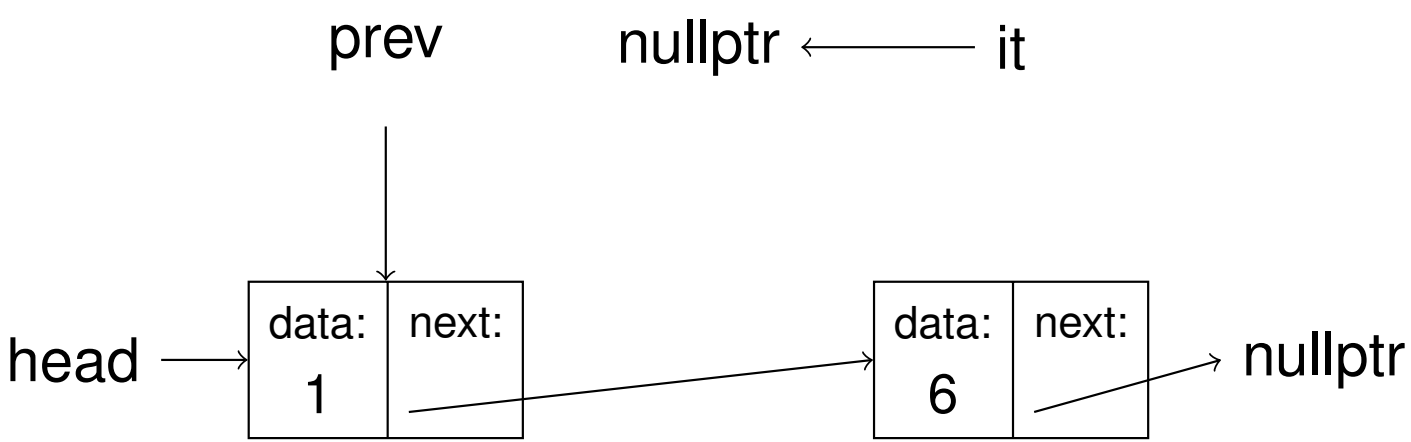
- Borrado de otra posición: disponemos de un puntero `prev` que apunta la posición **anterior** a la que vamos a borrar



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

Programación Avanzada y Estructuras de Datos

5. Árboles

Víctor M. Sánchez Cartagena

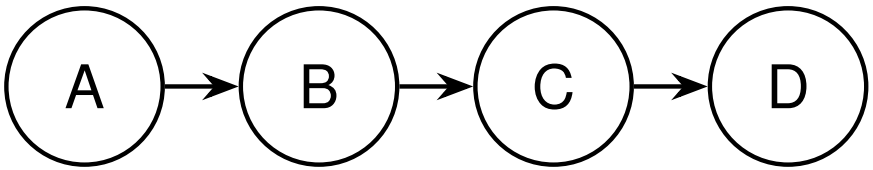
Grado en Ingeniería en Inteligencia Artificial
Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

12 de noviembre de 2024

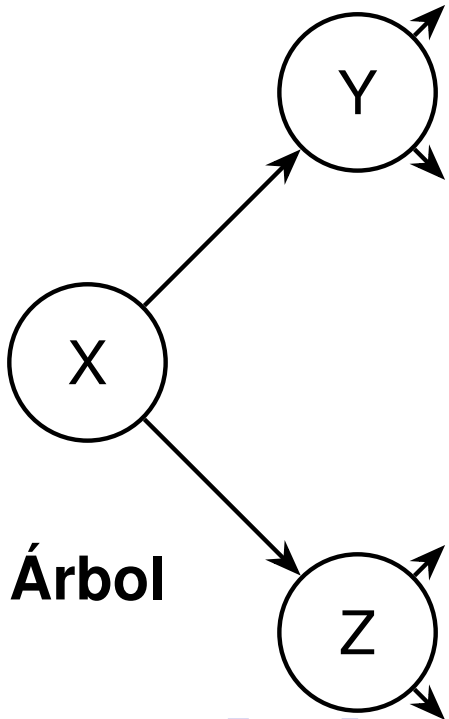
Índice

Idea intuitiva de árbol

- Para solucionar determinados problemas, es necesario organizar los datos de manera **no lineal**
- Tipos lineales (vectores, listas, pilas, colas): cada elemento tiene un sucesor y un predecesor
- Árboles: cada elemento tiene un predecesor, pero puede tener más de un sucesor

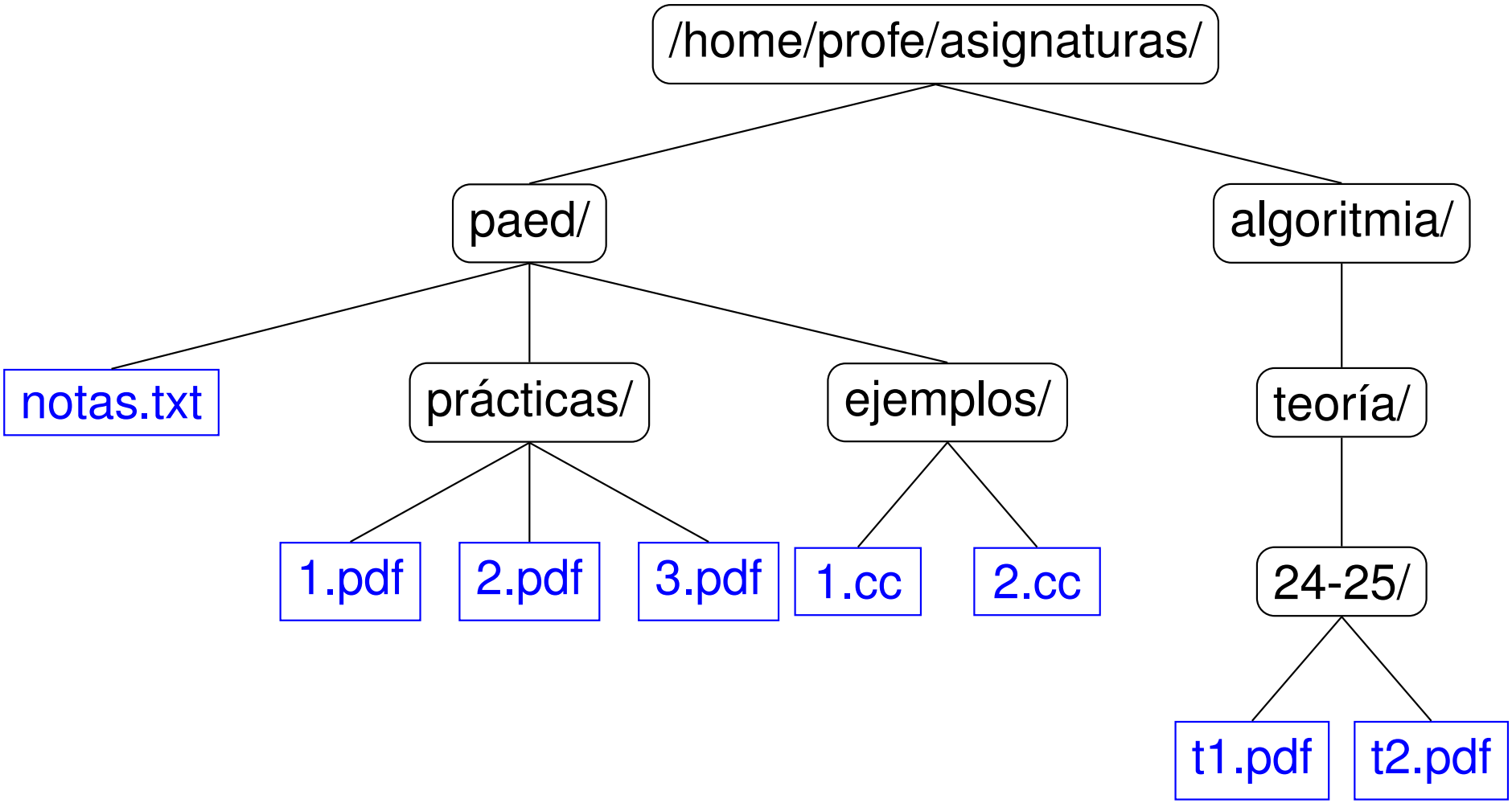


Tipo lineal



Árbol

Idea intuitiva de árbol



Definiciones

- El nodo `paed/` es el **padre** del nodo `notas.txt`
- Los **hijos** del nodo `paed/` son `notas.txt`, `prácticas/` y `ejemplos/`.
- El nodo `/home/profe/asignaturas/` es la **raíz** del árbol

Definición

Un árbol A es un conjunto de nodos que almacenan elementos y mantienen relaciones padre-hijo cumpliendo las siguientes propiedades:

- Si A no es vacío, contiene un único nodo especial, denominado **raíz**, que no tiene padre
- Cada nodo v de A que no es la raíz tiene un único **nodo padre** w; cada nodo cuyo padre es w es un **nodo hijo** de w

Definiciones

- Dos nodos con el mismo padre son **hermanos**
- Un nodo es **hoja** o **externo** si no tiene hijos
- Un nodo es **interno** si tiene uno o más hijos
- El **grado** de un nodo es el número de hijos que tiene
- El **grado** de un árbol es el máximo grado permitido para sus nodos

Definiciones

- Dos nodos con el mismo padre son **hermanos**
- Un nodo es **hoja** o **externo** si no tiene hijos
- Un nodo es **interno** si tiene uno o más hijos
- El **grado** de un nodo es el número de hijos que tiene
- El **grado** de un árbol es el máximo grado permitido para sus nodos

Preguntas

¿Cuántos nodos hoja tiene el árbol anterior?¿Cuántos nodos internos?¿Cuál es el grado del nodo `paed/`?¿Cuál es el grado del árbol?

Definiciones

- Un **camino** es una secuencia de uno o más nodos $n_1, n_2, ..., n_s$ tal que $\forall i \in 1..s-1$, n_i es padre de n_{i+1}
- La **longitud** de un camino es su número de nodos menos uno: existe un camino de longitud 0 de cada nodo hasta sí mismo

Definiciones

- Un **camino** es una secuencia de uno o más nodos $n_1, n_2, ..., n_s$ tal que $\forall i \in 1..s-1$, n_i es padre de n_{i+1}
- La **longitud** de un camino es su número de nodos menos uno: existe un camino de longitud 0 de cada nodo hasta sí mismo

Preguntas

- ¿Existe un camino de `paed/` a `24-25`? En caso afirmativo, enumera sus nodos e indica su longitud
- ¿Existe un camino de `algoritmia/` a `t1.pdf`? En caso afirmativo, enumera sus nodos e indica su longitud

Definiciones

- Un nodo w es **descendiente** de otro nodo v (y v es **ascendente** de w) si existe un camino $v, ..., w$
- Los descendientes de un nodo v, excluyendo al mismo v, se denominan **descendientes propios**
- La misma definición se aplica para los **ascendentes propios**

Definiciones

- Un nodo w es **descendiente** de otro nodo v (y v es **ascendente** de w) si existe un camino $v, ..., w$
- Los descendientes de un nodo v, excluyendo al mismo v, se denominan **descendientes propios**
- La misma definición se aplica para los **ascendentes propios**

Preguntas

- ¿Cuáles son los descendientes del nodo `teoría/`? ¿Cuáles de ellos son propios?
- ¿Cuáles son los ascendentes del nodo `teoría/`? ¿Cuáles de ellos son propios?

Definiciones

- La raíz de un árbol está en el nivel 1
- Los hijos de un nodo que está en el nivel i están en el nivel $i + 1$
- Dicho de otro modo: el nivel de un nodo es el resultado de sumarle 1 a la longitud del camino desde la raíz hasta ese nodo
- La **altura** de un árbol es el máximo nivel de sus nodos

Definiciones

- La raíz de un árbol está en el nivel 1
- Los hijos de un nodo que está en el nivel i están en el nivel $i + 1$
- Dicho de otro modo: el nivel de un nodo es el resultado de sumarle 1 a la longitud del camino desde la raíz hasta ese nodo
- La **altura** de un árbol es el máximo nivel de sus nodos

Preguntas

- ¿Cuál es el nivel de cada uno de estos nodos?: `/home/profe/asignaturas/`, `notas.txt`, `ejemplos/`, `24-25/`, `t1.pdf`
- ¿Cuál es la altura del árbol de ejemplo?

TAD árbol

Especificación del TAD árbol

Operaciones:

- Devuelve el número de nodos en el árbol

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

- Comprueba si el árbol está vacío

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

- Devuelve un iterador que apunta a la raíz del árbol

```
Iterador root() const;
```

Especificación del TAD árbol

Operaciones:

- Asigna el valor `e` al nodo del árbol 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) const;
```

Especificación del TAD Iterador

Operaciones:

- Devuelve un iterador al nodo padre del nodo actual

```
Iterador parent() const;
```

- Devuelve un vector de iteradores a los nodos hijos del nodo actual

```
vector<Iterador> children() const;
```

- Comprueba si el nodo actual es la raíz

```
bool isRoot() const;
```

Pregunta

¿Qué imprime el siguiente fragmento de código? Asume que la variable `a` es de tipo `Arbol` y representa el árbol de ejemplo empleado anteriormente

```
Arbol a;
//....
cout << a.size() << endl;
Iterador it=a.root().children()[0].children()[2].children()[1];
cout << a.get(it) << endl;
```

TAD árbol

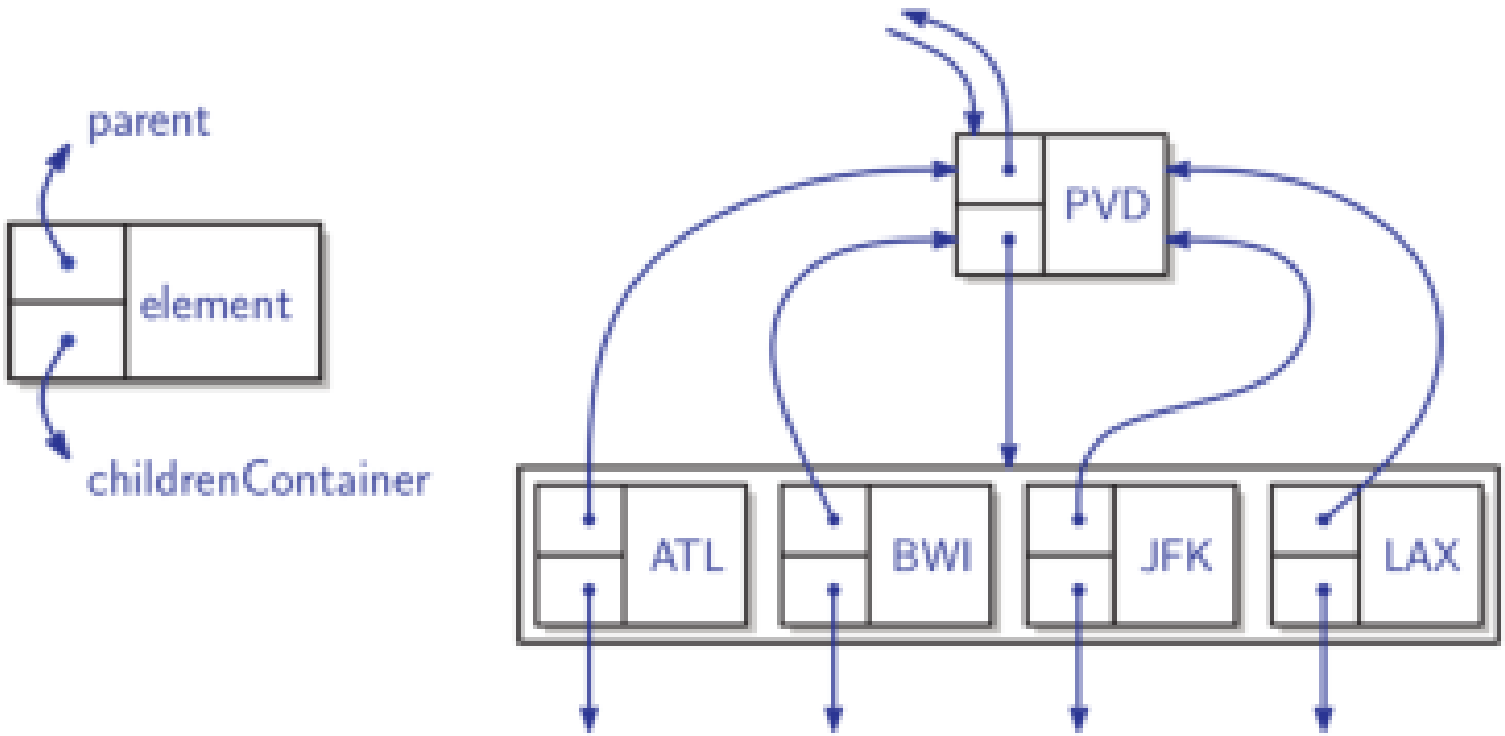
Pregunta

¿Qué imprime el siguiente fragmento de código? Asume que la variable `a` es de tipo `Arbol` y representa el árbol de ejemplo empleado anteriormente

```
Arbol a;
//....
cout << a.size() << endl;
Iterador it=a.root().children()[0].children()[2].children()[1];
cout << a.get(it) << endl;
```

```
15
2.cc
```

Implementación enlazada para árboles generales



fuente: Goodrich, M. T., Tamassia, R., & Mount, D. M. (2011). Data structures and algorithms in C++. John Wiley & Sons.

Implementación enlazada para árboles generales

```
class Arbol{
private:
Nodo* root;
....
};

class Nodo{
private:
   Elem data;
   Nodo* parent;
   vector<Nodo>* childrenContainer;
...
};
```

Implementación enlazada para árboles generales

- Complejidad asintótica respecto al número de nodos en el árbol (`size()`)

Operación	Coste
`root()`	
`empty()`	
`set()`	
`get()`	
`parent()`	
`isRoot()`	
`children()`	

$g =$ grado del árbol

Implementación enlazada para árboles generales

- Complejidad asintótica respecto al número de nodos en el árbol (`size()`)

Operación	Coste
`root()`	$\Theta(1)$
`empty()`	$\Theta(1)$
`set()`	$\Theta(1)$
`get()`	$\Theta(1)$
`parent()`	$\Theta(1)$
`isRoot()`	$\Theta(1)$
`children()`	$\Omega(1)$, $O(g)$

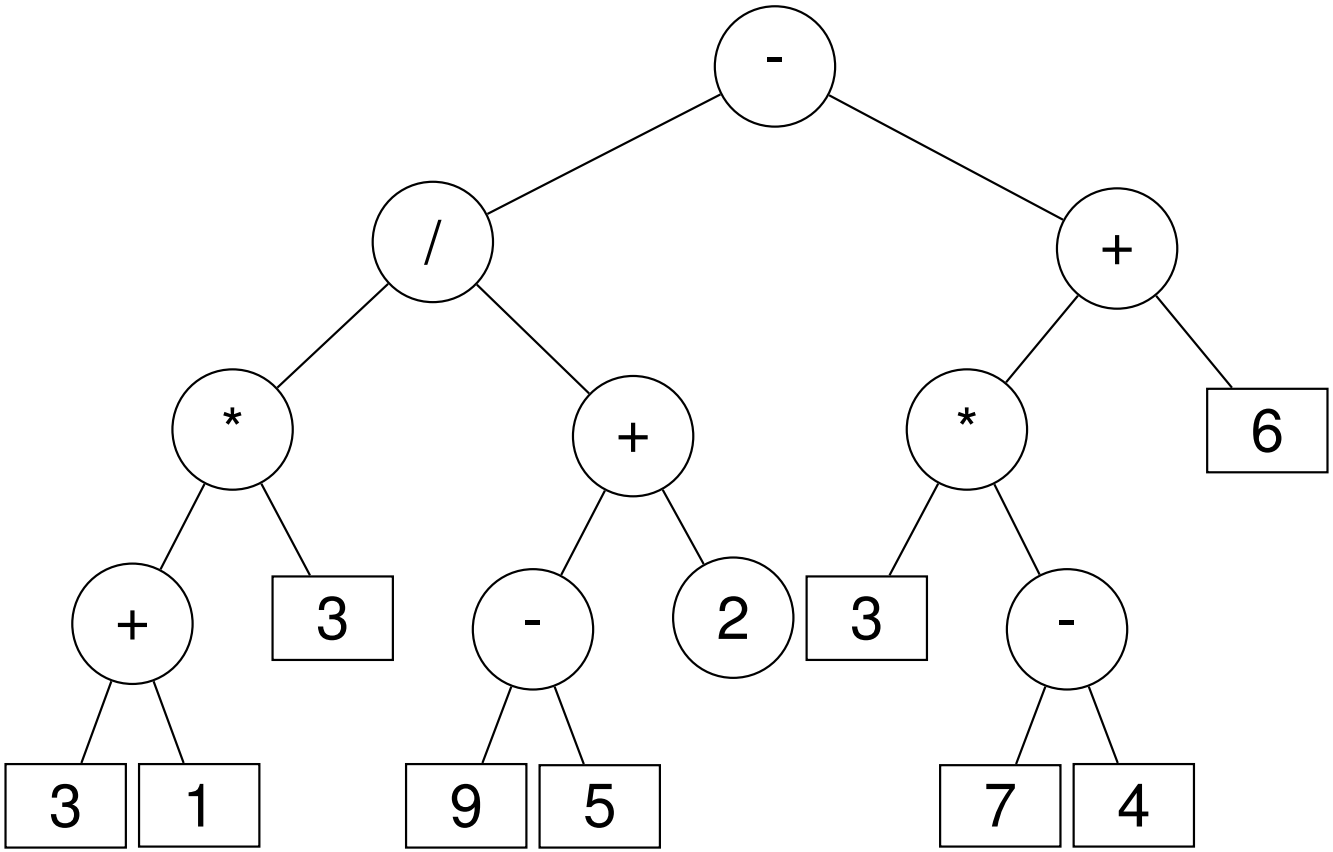
$g =$ grado del árbol

Índice

Idea intuitiva de árbol binario

- Ejemplo: compilación de expresión aritmética

```
(((((3 + 1) * 3)/((9 - 5) + 2)) - ((3 * (7 - 4)) + 6))
```

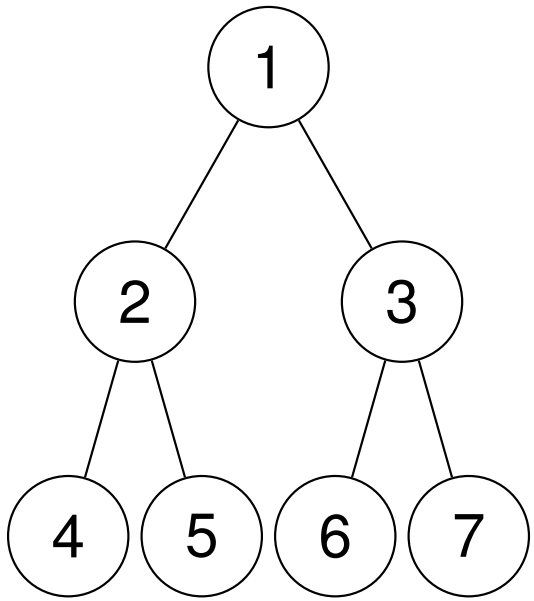


Definiciones

- Un **árbol binario** es un árbol de grado 2: cada nodo puede tener como máximo 2 hijos
- Consideraremos que cada nodo siempre tiene un hijo izquierdo y un hijo derecho
- Uno, o ambos hijos pueden estar vacíos
- Un nodo con los dos hijos vacíos es un **nodo hoja**

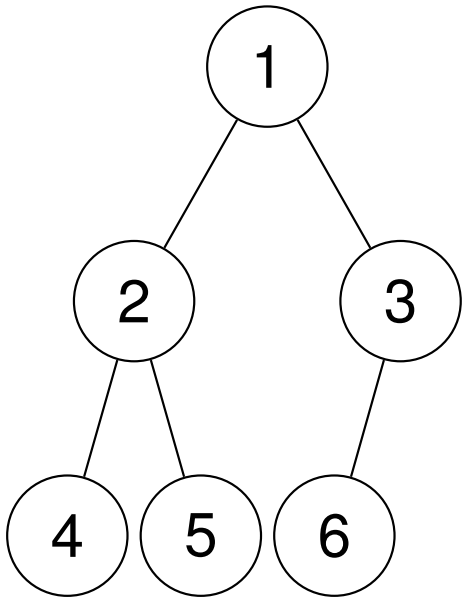
Definiciones

- Un árbol binario **perfecto** es aquel en el que todos los nodos internos tienen dos hijos y todas las hojas están en el mismo nivel



Definiciones

- Un árbol binario **completo** de altura h es aquel en el que los niveles del 1 al $h-1$ tienen el número máximo de nodos (igual que un árbol perfecto), y en el nivel h, los nodos siempre ocupan de manera continua posiciones desde la izquierda



Propiedades

- El máximo número de nodos en el nivel *i* de un árbol binario es 2^{i-1}
- Demostración informal:
  - Nivel 1 (raíz): $2^{1-1} = 2^0 = 1$
  - Nivel 2: $2^{2-1} = 2^1 = 2$
  - Nivel 3: $2^{3-1} = 2^2 = 4$
  - Y así sucesivamente, el número máximo se va duplicando en cada nivel

Propiedades

- El máximo número de nodos en un árbol binario de altura h es $2^h - 1$
- Demostración $\rightarrow$ suma del número de nodos de cada nivel i

$$\sum_{i=1}^{h} 2^{i-1} = 1 \cdot \frac{2^h - 1}{2 - 1} = 2^h - 1$$

- Suma de una progresión geométrica de razón r:

$$S = a_1 \cdot \frac{r^n - 1}{r - 1}$$

TAD árbol binario

Especificación del TAD Iterador

Operaciones:

- Devuelve un iterador al nodo padre del nodo actual

```
Iterador parent() const;
```

- Devuelve un iterador al hijo izquierdo

```
Iterador left() const;
```

- Devuelve un iterador al hijo derecho

```
Iterador right() const;
```

Especificación del TAD Iterador

Operaciones:

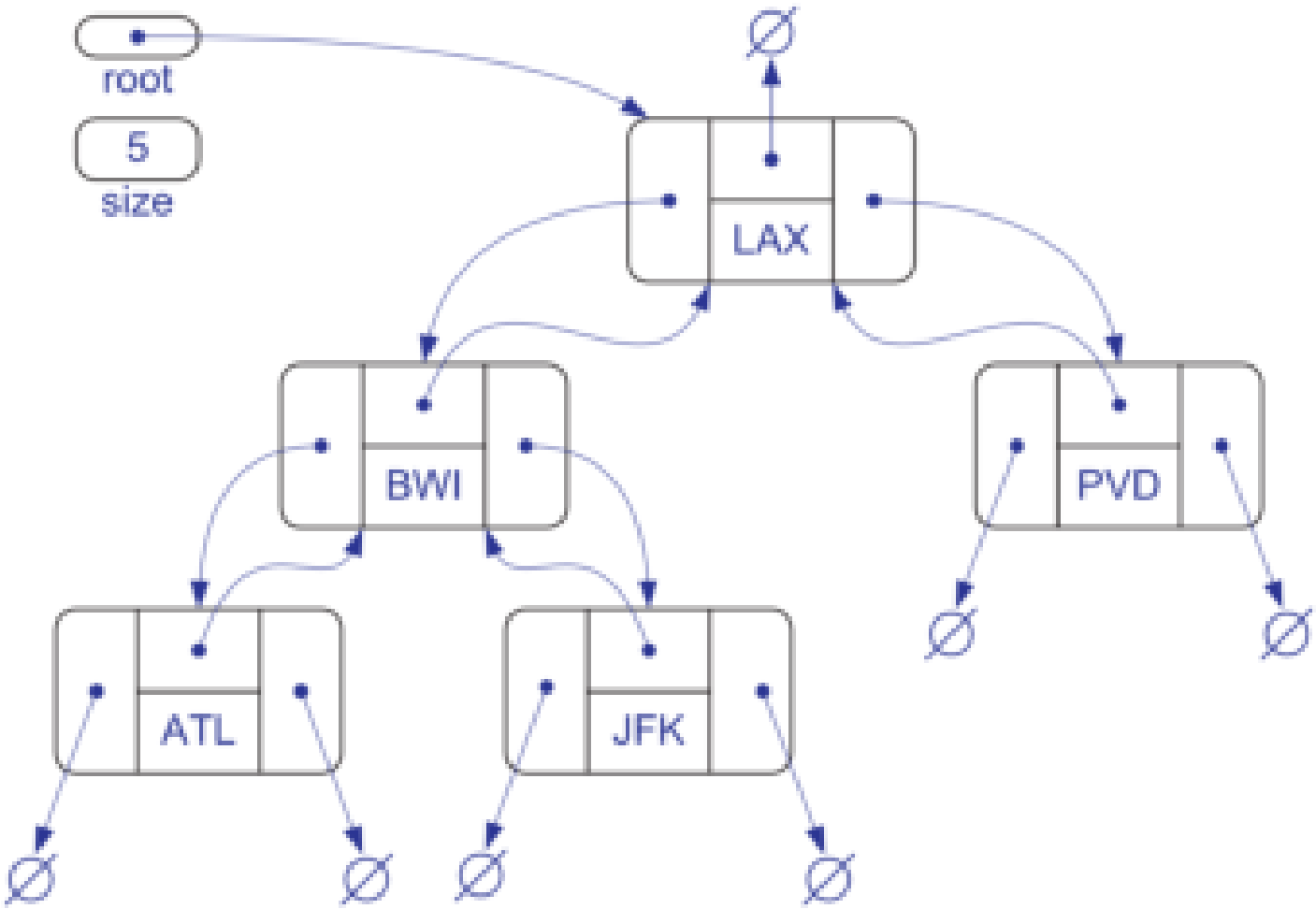
- Comprueba si el iterador apunta al nodo raíz

```
bool isRoot() const;
```

- Comprueba si el iterador apunta a un árbol vacío

```
bool isEmpty() const;
```

Implementación enlazada para árboles binarios



fuente: Goodrich, M. T., Tamassia, R., & Mount, D. M. (2011). Data structures and algorithms in C++. John Wiley & Sons.

Implementación enlazada para árboles binarios

```
class Arbol{
private:
Nodo* root;
....
};

class Nodo{
private:
    Elem data;
    Nodo* parent;
    Nodo* left;
    Nodo* right;
...
};
```

Implementación enlazada para árboles binarios

- Complejidad asintótica respecto al número de nodos en el árbol (`size()`)

Operación	Coste
`root()`	
`empty()`	
`set()`	
`get()`	
`parent()`	
`isRoot()`	
`left()`	
`right()`	

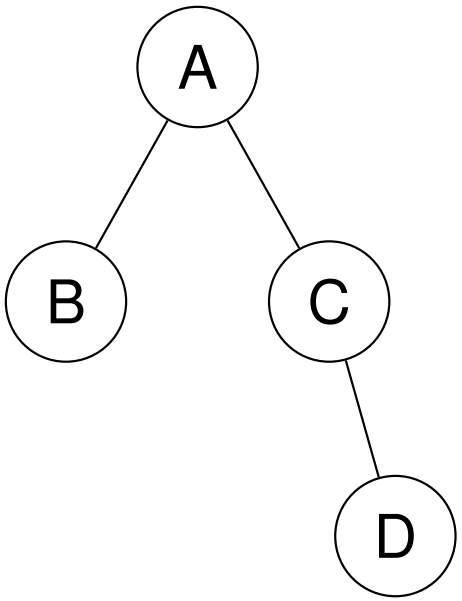
Implementación enlazada para árboles binarios

- Complejidad asintótica respecto al número de nodos en el árbol (`size()`)

Operación	Coste
`root()`	$\Theta(1)$
`empty()`	$\Theta(1)$
`set()`	$\Theta(1)$
`get()`	$\Theta(1)$
`parent()`	$\Theta(1)$
`isRoot()`	$\Theta(1)$
`left()`	$\Theta(1)$
`right()`	$\Theta(1)$

Implementación vector para árboles binarios

- Las posiciones del vector pasan a numerarse comenzando por 1
- El contenido del nodo raíz se almacena en la posición 1
- Si un nodo se almacena en la posición i, su hijo izquierdo se almacena en la posición $2i$, y su hijo derecho en la posicion $2i + 1$
- Si el árbol no es *perfecto*, puede haber posiciones "vacías", que se marcan con un valor especial



A	B	C	#	#	#	D
1	2	3	4	5	6	7

Implementación vector para árboles binarios

- La clase `Iterador` ahora contiene un índice en lugar de un puntero

Pregunta

¿Cómo implementarías el método `isRoot()` de la clase `Iterador`? ¿Y el método `parent()`? Asume que la clase `Iterador` tiene un atributo llamado `index` con el índice del vector al que apunta el iterador

Implementación vector para árboles binarios

- Las complejidades temporales son constantes al igual que en la implementación enlazada
- Si hubiésemos definido operaciones de inserción, éstas no tendrían coste constante en el peor caso (por los redimensionamientos)

Pregunta

¿Cuál es la complejidad **espacial** de la representación vectoral del árbol binario en el peor caso? (Pista: depende de la altura del árbol h y no del número de elementos que almacena)

Implementación vector para árboles binarios

- Las complejidades temporales son constantes al igual que en la implementación enlazada
- Si hubiésemos definido operaciones de inserción, éstas no tendrían coste constante en el peor caso (por los redimensionamientos)

Pregunta

¿Cuál es la complejidad **espacial** de la representación vectoral del árbol binario en el peor caso? (Pista: depende de la altura del árbol h y no del número de elementos que almacena)

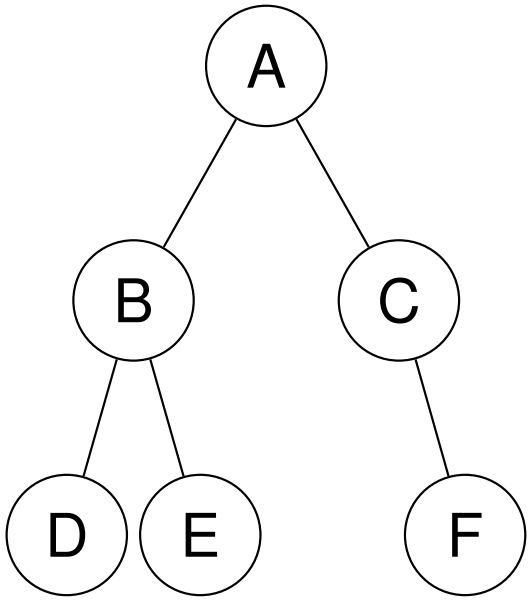
Respuesta: $O(2^h)$ $\leftarrow$ el número de elementos de un árbol perfecto es $2^h - 1$

Índice

Recorridos en árboles binarios

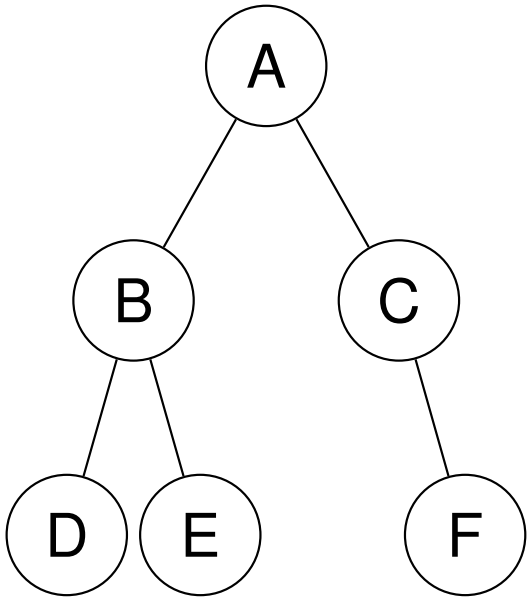
- **Recorrido**: manera sistemática de listar o recorrer todos los nodos de un árbol, sin repetir ninguno
- Dependiendo del tipo de los datos guardados en el árbol y lo que se desee hacer con ellos, elegiremos uno u otro
- Recorridos en profundidad:
  - Preorden
  - Postorden
  - Inorden
- Recorridos en anchura:
  - Niveles

Recorrido preorden




```
function PREORDEN(T,p)
    if !p.isEmpty() then
        PRINT(p)
        PREORDEN(p.left())
        PREORDEN(p.right())
```

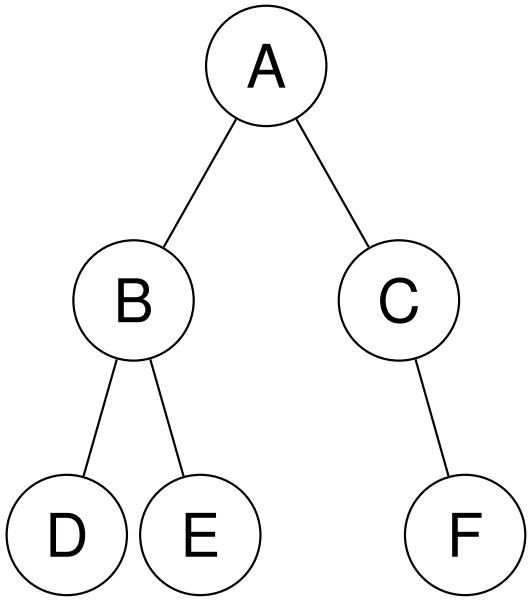
Recorrido preorden




```
function PREORDEN(T,p)
    if !p.isEmpty() then
        PRINT(p)
        PREORDEN(p.left())
        PREORDEN(p.right())
```

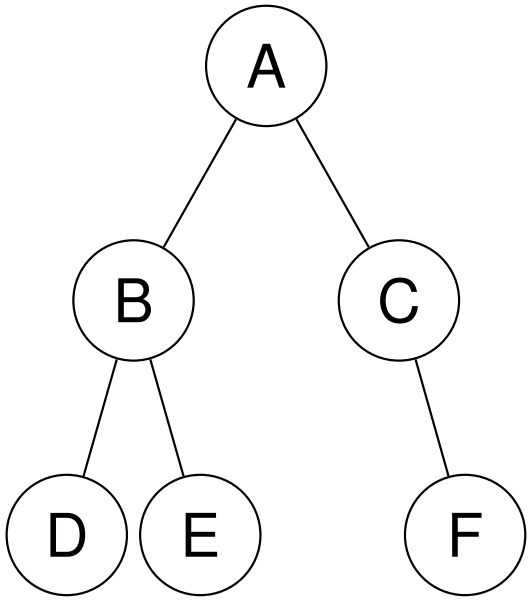
Resultado: A B D E C F

Recorrido postorden




```
function POSTORDEN(T,p)
    if !p.isEmpty() then
        POSTORDEN(p.left())
        POSTORDEN(p.right())
        PRINT(p)
```

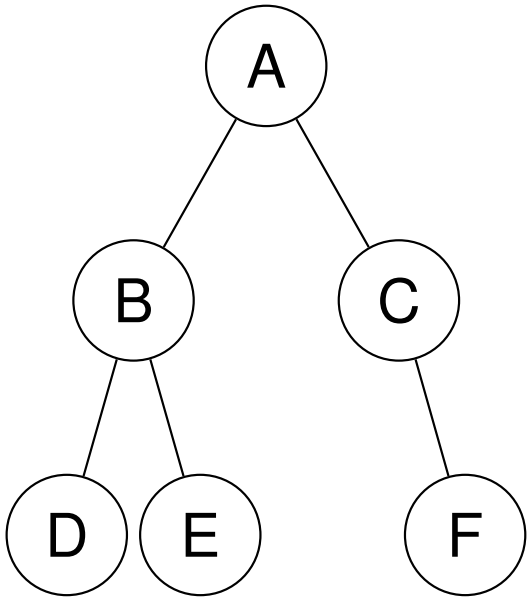
Recorrido postorden




```
function POSTORDEN(T,p)
    if !p.isEmpty() then
        POSTORDEN(p.left())
        POSTORDEN(p.right())
        PRINT(p)
```

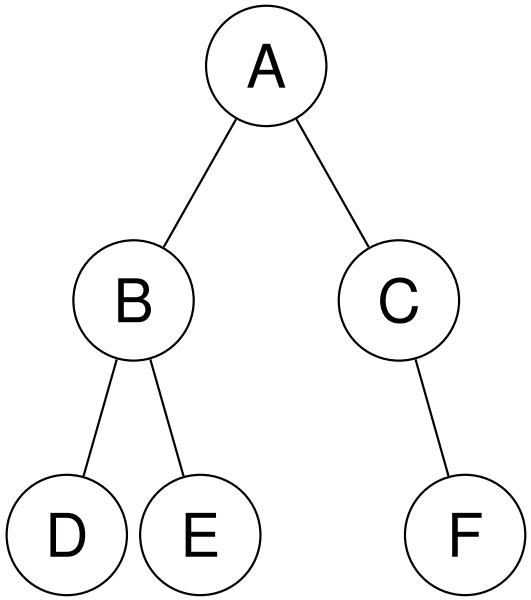
Resultado: D E B F C A

Recorrido inorden




```
function INORDEN(T,p)
    if !p.isEmpty() then
        INORDEN(p.left())
        PRINT(p)
        INORDEN(p.right())
```

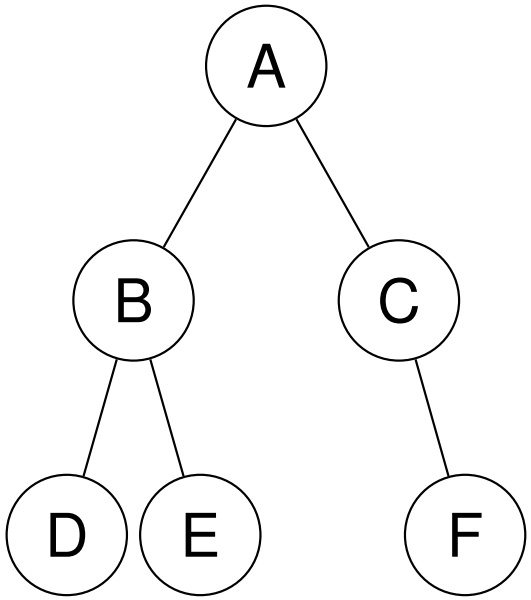
Recorrido inorden




```
function INORDEN(T,p)
    if !p.isEmpty() then
        INORDEN(p.left())
        PRINT(p)
        INORDEN(p.right())
```

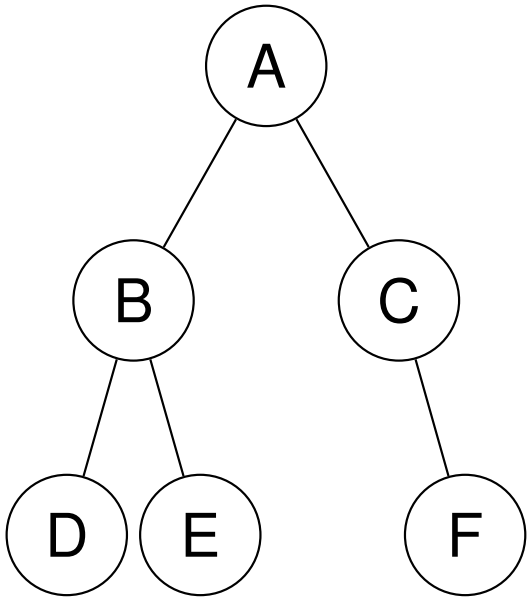
Resultado: D B E A C F

Recorrido niveles




```
function NIVELES(T,p)
    Cola c
    c.enqueue(p)
    while !c.isEmpty() do
        n=c.front()
        c.dequeue()
        PRINT(n)
        if !n.left().isEmpty()
        then
            c.enqueue(n.left())
        if !n.right().isEmpty()
        then
            c.enqueue(n.right())
```

Recorrido niveles



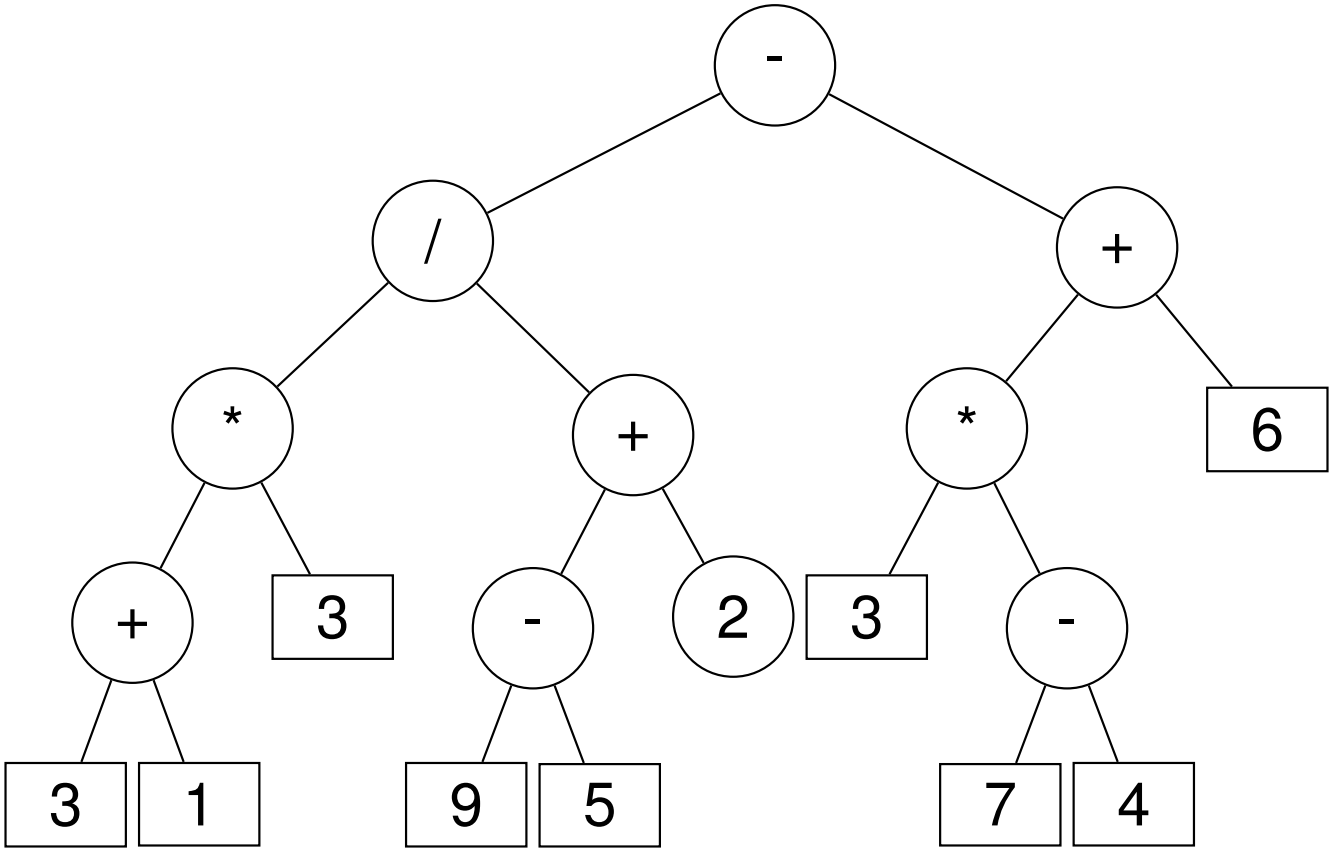

```
function NIVELES(T,p)
    Cola c
    c.enqueue(p)
    while !c.isEmpty() do
        n=c.front()
        c.dequeue()
        PRINT(n)
        if !n.left().isEmpty()
        then
            c.enqueue(n.left())
        if !n.right().isEmpty()
        then
            c.enqueue(n.right())
```

Resultado: A B C D E F

Recorridos en árboles binarios

Pregunta

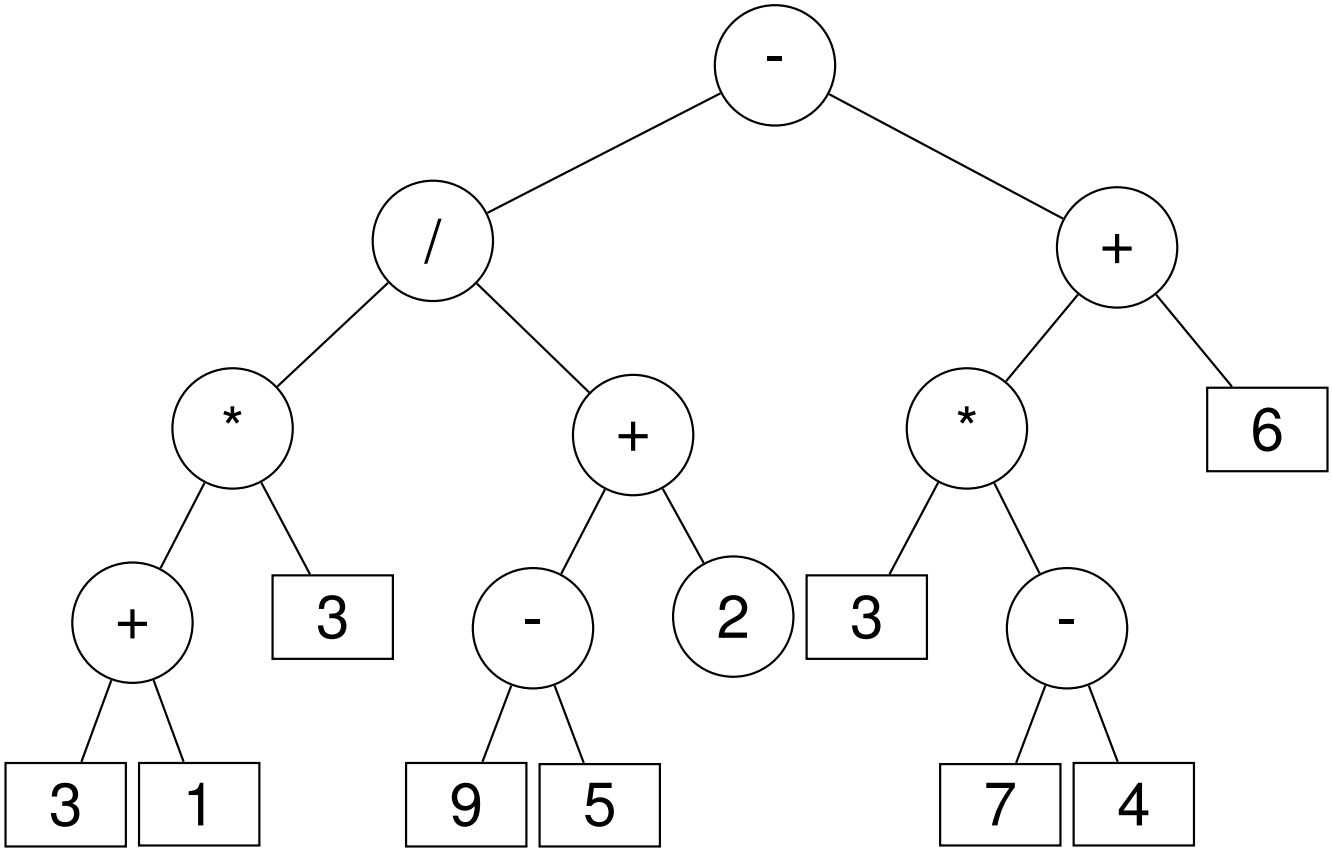
¿Qué recorrido emplearías para calcular el valor de la expresión aritmética a partir de su árbol?



Recorridos en árboles binarios

Pregunta

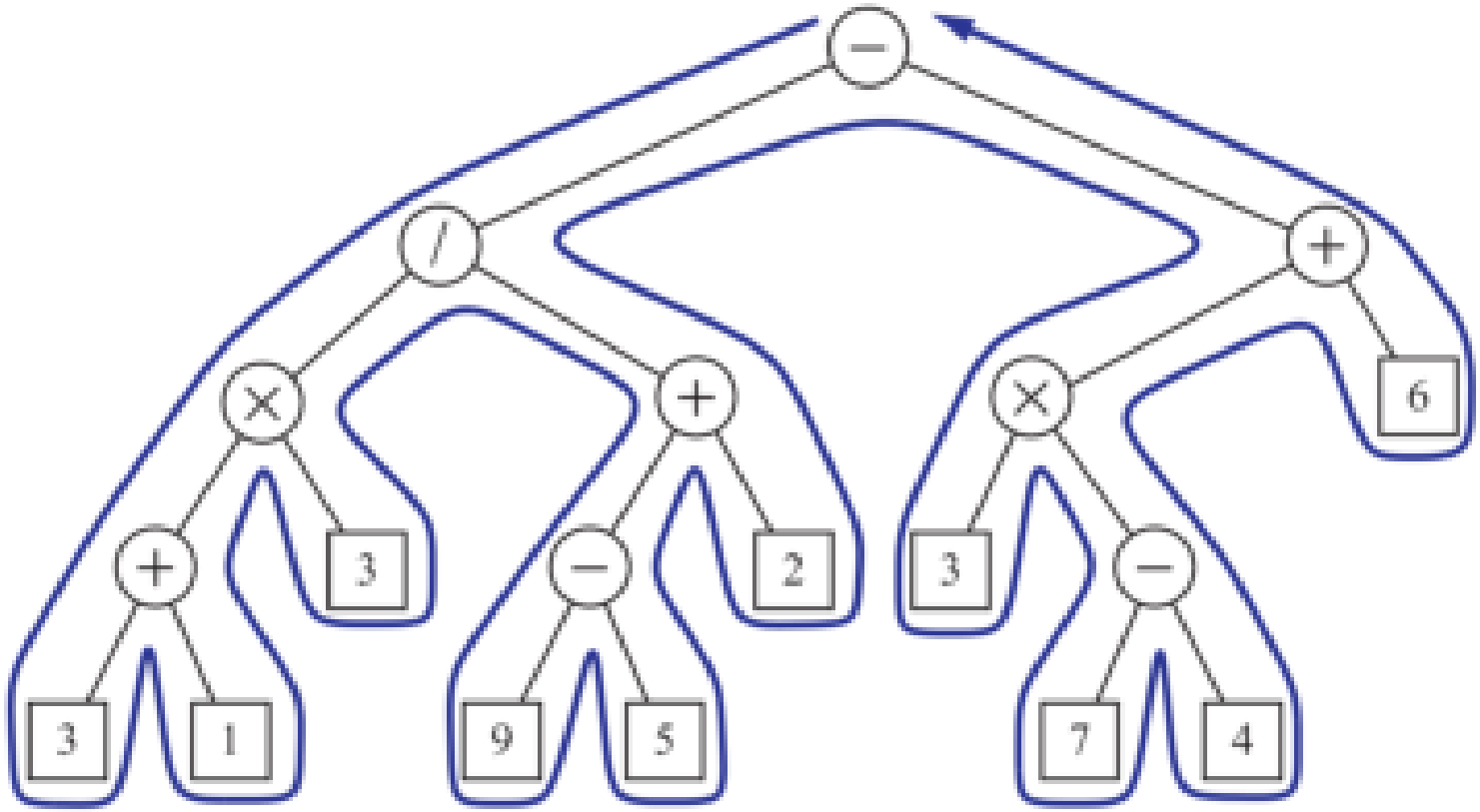
¿Qué recorrido emplearías para calcular el valor de la expresión aritmética a partir de su árbol?



Respuesta: postorden

Recorridos en árboles binarios

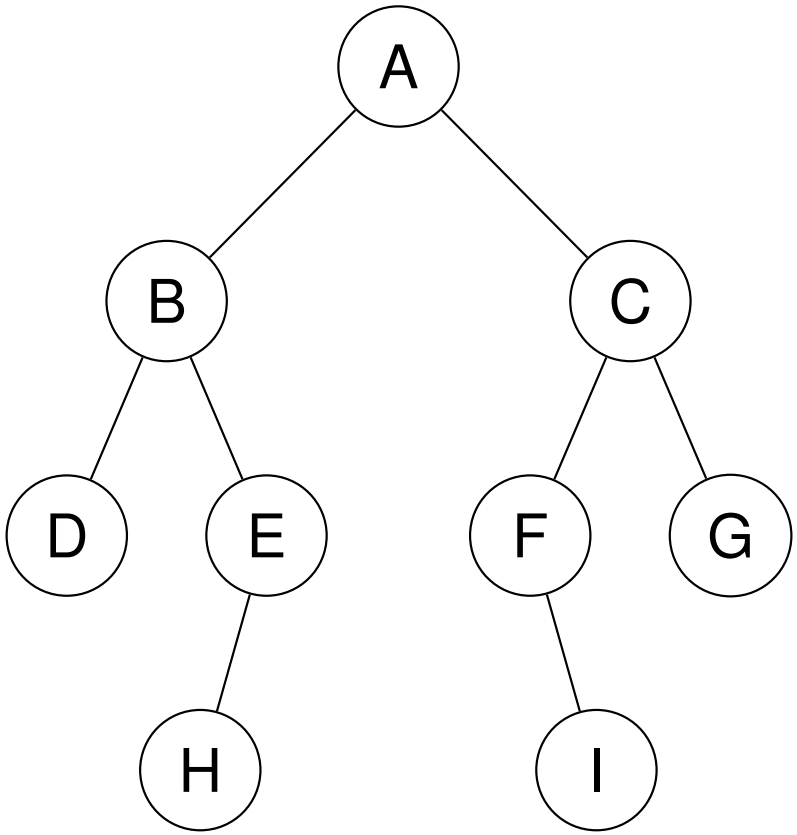
- El método del *recorrido de Euler* permite calcular los recorridos de manera gráfica



Recorridos en árboles binarios

Pregunta

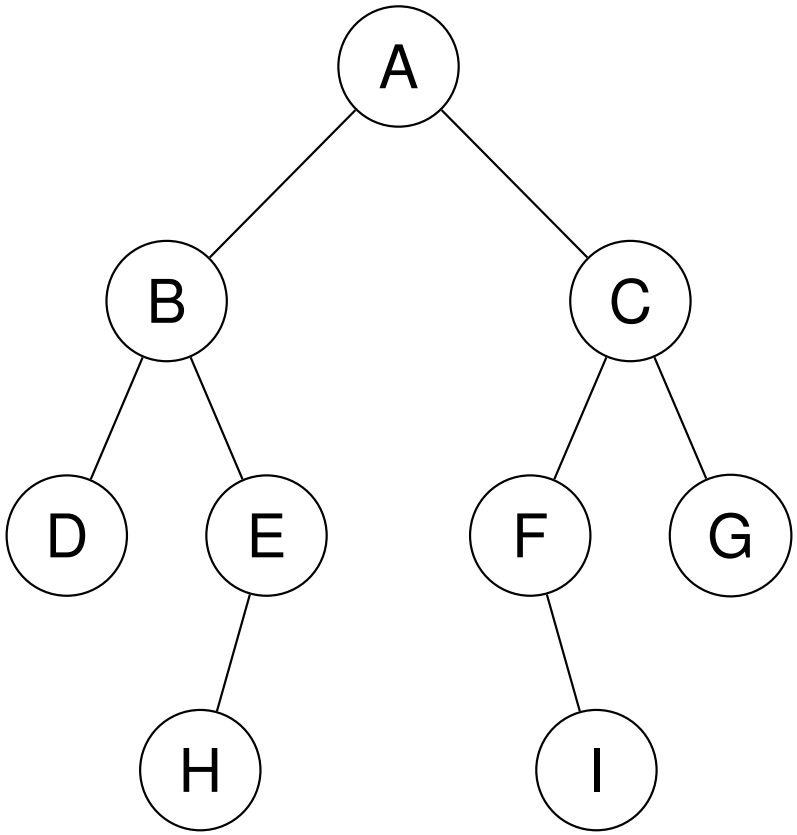
Calcula los recorridos preorden, postorden, inorden y niveles del siguiente árbol



Recorridos en árboles binarios

Pregunta

Calcula los recorridos preorden, postorden, inorden y niveles del siguiente árbol

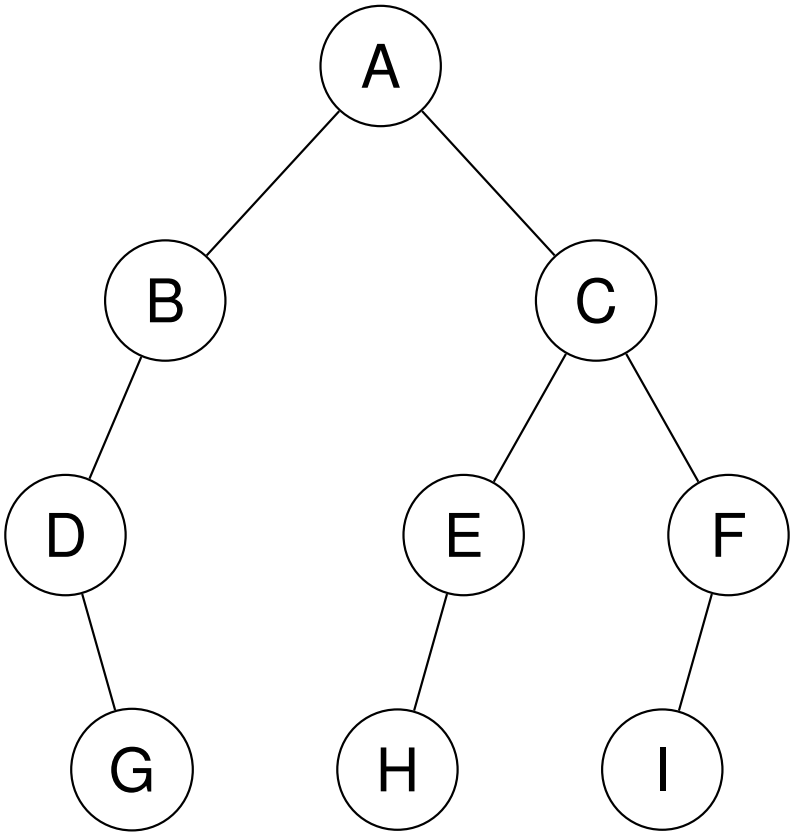


- Preorden: ABDEHCFIG
- Postorden: DHEBIFGCA
- Inorden: DBHEAFICG
- Niveles: ABCDEFGHI

Recorridos en árboles binarios

Ejercicio

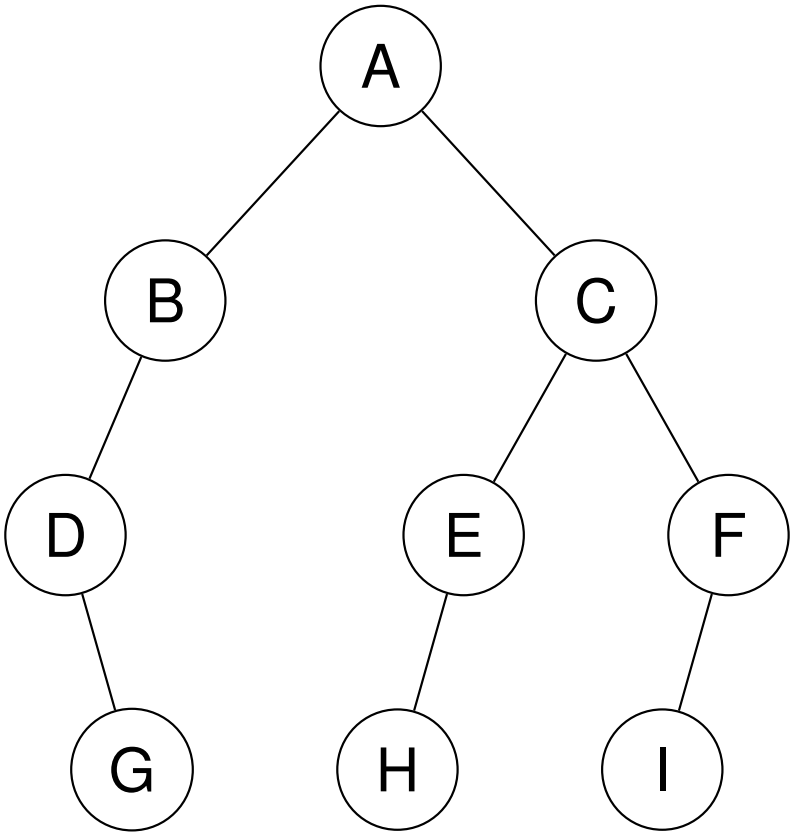
Calcula los recorridos preorden, postorden, inorden y niveles del siguiente árbol



Recorridos en árboles binarios

Ejercicio

Calcula los recorridos preorden, postorden, inorden y niveles del siguiente árbol



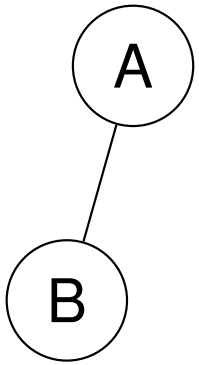
- Preorden: ABDGCEHFI
- Postorden: GDBHEIFCA
- Inorden: DGBAHECIF
- Niveles: ABCDEFGHI

Reconstrucción de árboles binarios

Pregunta

Dado un recorrido (por ejemplo, preorden), ¿crees que es posible reconstruir un único árbol a partir del mismo?

Para contestar a la pregunta, te puede ser útil calcular el recorrido preorden del siguiente árbol y luego intentar buscar otro árbol con el mismo recorrido:

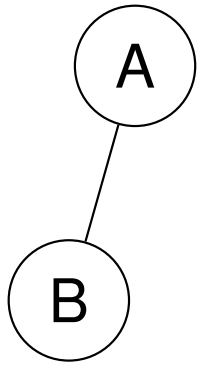


Reconstrucción de árboles binarios

Pregunta

Dado un recorrido (por ejemplo, preorden), ¿crees que es posible reconstruir un único árbol a partir del mismo?

Para contestar a la pregunta, te puede ser útil calcular el recorrido preorden del siguiente árbol y luego intentar buscar otro árbol con el mismo recorrido:



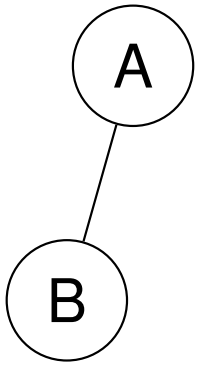
Respuesta: No es posible reconstruir un único árbol a partir de un recorrido

Reconstrucción de árboles binarios

Pregunta

¿Cuántos recorridos es necesario conocer para poder reconstruir un único árbol? ¿Cuáles?

Para contestar a la pregunta, te puede ser útil calcular los cuatro recorridos del siguiente árbol, y luego buscar pareja de recorridos casa con un único árbol

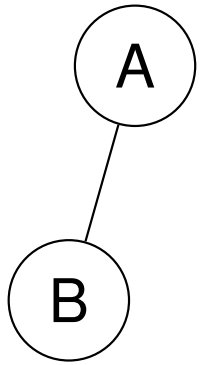


Reconstrucción de árboles binarios

Pregunta

¿Cuántos recorridos es necesario conocer para poder reconstruir un único árbol? ¿Cuáles?

Para contestar a la pregunta, te puede ser útil calcular los cuatro recorridos del siguiente árbol, y luego buscar pareja de recorridos casa con un único árbol



Respuesta: Dos. Inorden y (preorden o postorden o niveles)

Reconstrucción de árboles binarios

Algoritmo para la reconstrucción de un árbol a partir de sus recorridos preorden e inorden

1. Identifica el primer elemento del recorrido preorden: *r* es la raíz
2. Todos los elementos a la izquierda de *r* en el recorrido inorden forman parte del subárbol hijo izquierdo de *r*
3. Todos los elementos a la derecha de *r* en el recorrido inorden forman parte del subárbol hijo derecho de *r*
4. Vuelve al primer paso para cada subárbol

Reconstrucción de árboles binarios

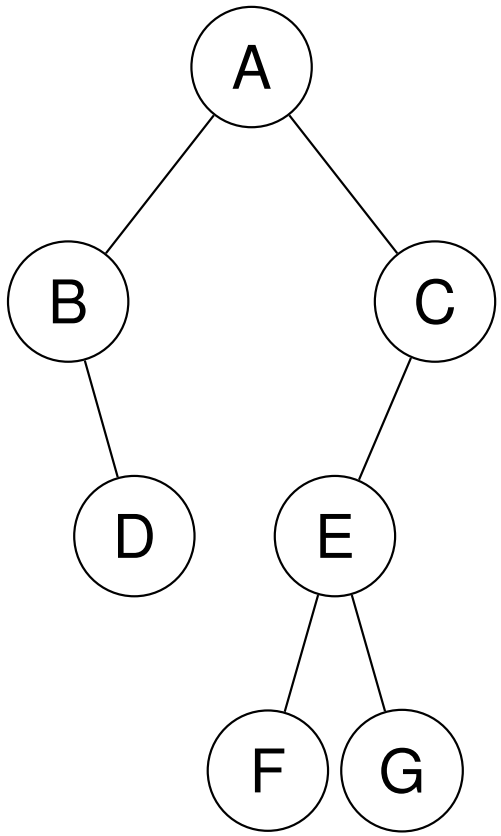
Reconstrucción de un árbol

- Preorden: ABDCEFG
- Inorden: BDAFEGC

Reconstrucción de árboles binarios

Reconstrucción de un árbol

- Preorden: ABDCEFG
- Inorden: BDAFEGC



Reconstrucción de árboles binarios

Ejercicio

Reconstruye un árbol binario a partir de los siguientes recorridos

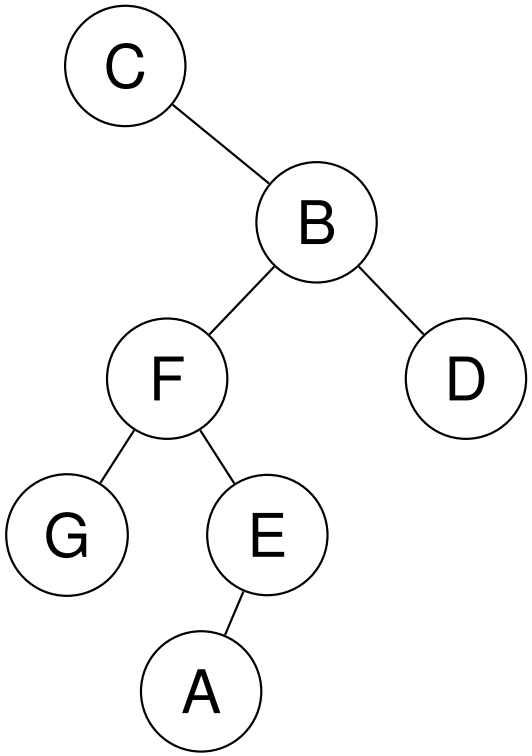
- Preorden: CBFGEAD
- Inorden: CGFAEBD

Reconstrucción de árboles binarios

Ejercicio

Reconstruye un árbol binario a partir de los siguientes recorridos

- Preorden: CBFGEAD
- Inorden: CGFAEBD



Reconstrucción de árboles binarios

Pregunta

¿Se te ocurre alguna situación bajo la que podría reconstruirse un árbol a partir de un único recorrido?

Reconstrucción de árboles binarios

Pregunta

¿Se te ocurre alguna situación bajo la que podría reconstruirse un árbol a partir de un único recorrido?

Respuesta: Si conocemos alguna otra propiedad del árbol, como su **estructura**. Se puede reconstruir un árbol conociendo su recorrido por niveles si sabemos que es perfecto o completo

Repaso

Pregunta

Contesta verdadero o falso a cada una de estas afirmaciones

- El nivel de un nodo coincide con la longitud del camino desde la raíz a ese nodo
- El grado de un árbol es la altura maxima que puede tener
- Un árbol siempre tiene más nodos internos que nodos hoja
- Existe un camino entre cualquier par de nodos de un árbol
- Un nodo hoja no tiene descendientes
- Un árbol binario con un único nodo es un árbol completo
- Todo árbol binario con dos nodos es completo
- El número máximo de nodos que se puede encontrar en el nivel 1 de un árbol binario es 2
- Un árbol binario de altura 3 obligatoriamente debe tener más de 3 nodos

Índice

Definición

Definición

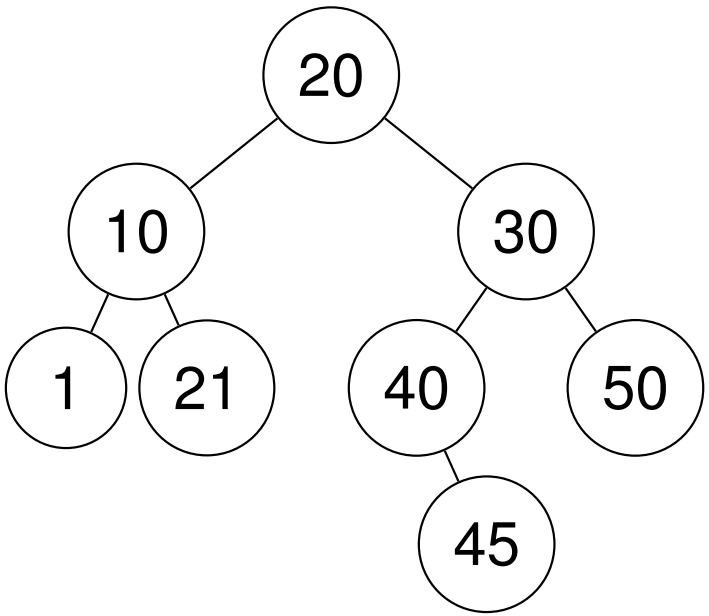
Un árbol binario de búsqueda (ABB) es un árbol binario que almacena un elemento en cada nodo. Los elementos deben ser de un tipo sobre el que esté definido una relación de orden total ($\leq$). Dado un nodo que almacena un elemento e, todos los elementos almacenados en su subárbol izquierdo deben ser $\leq e$. Todos los elementos almacenados en su subárbol derecho deben ser $> e$.

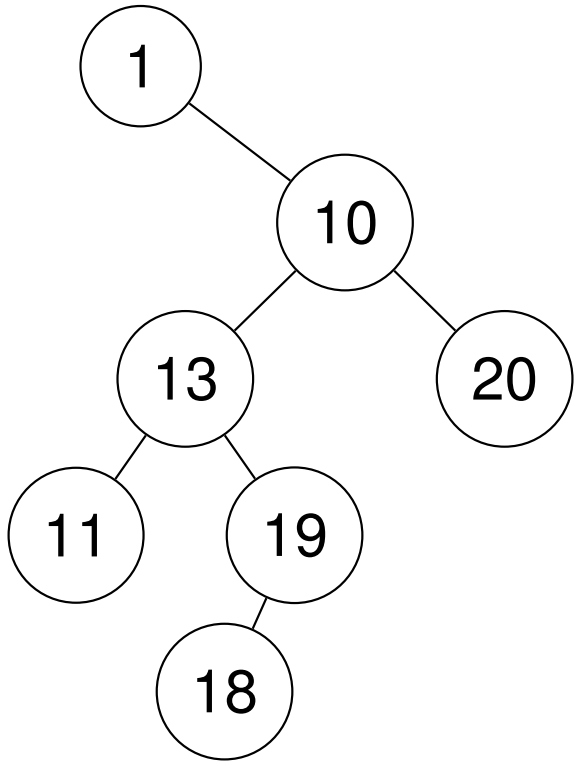
- Aunque su definición formal lo permite, asumiremos que no puede haber elementos repetidos en un árbol binario de búsqueda
- Un ABB almacena, por tanto, un **conjunto** de elementos

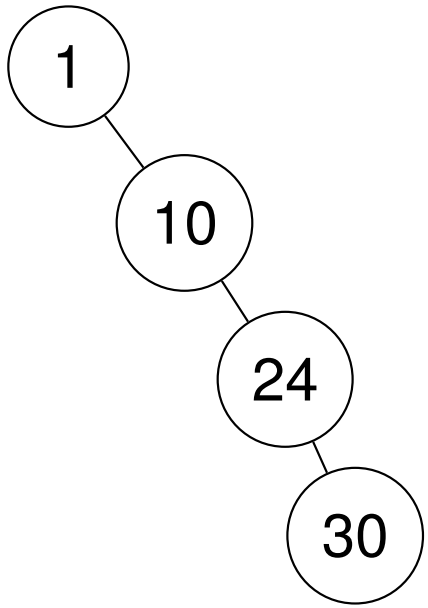
Definición

Pregunta

Indica cuáles de los siguientes árboles son árboles binarios de búsqueda







Recorridos sobre árboles binarios de búsqueda

Pregunta

¿Hay algún recorrido que nos devuelva los datos ordenados según la relación de orden parcial definida sobre los elementos almacenados?

Recorridos sobre árboles binarios de búsqueda

Pregunta

¿Hay algún recorrido que nos devuelva los datos ordenados según la relación de orden parcial definida sobre los elementos almacenados?

Respuesta: inorden

Reconstrucción de árboles binarios de búsqueda

Pregunta

¿Podemos reconstruir un árbol binario de búsqueda a partir de un único recorrido?

Reconstrucción de árboles binarios de búsqueda

Pregunta

¿Podemos reconstruir un árbol binario de búsqueda a partir de un único recorrido?

Respuesta: Sí. Preorden, postorden o niveles

Reconstrucción de árboles binarios de búsqueda

Algoritmo para la reconstrucción de un árbol binario de búsqueda a partir de sus recorrido preorden

1. Identifica el primer elemento del recorrido preorden: r es la raíz
2. Todos los elementos menores que r en el recorrido preorden forman parte del subárbol hijo izquierdo de r
3. Todos los elementos mayores que r en el recorrido preorden forman parte del subárbol hijo derecho de r
4. Vuelve al primer paso para cada subárbol

Reconstrucción de árboles binarios de búsqueda

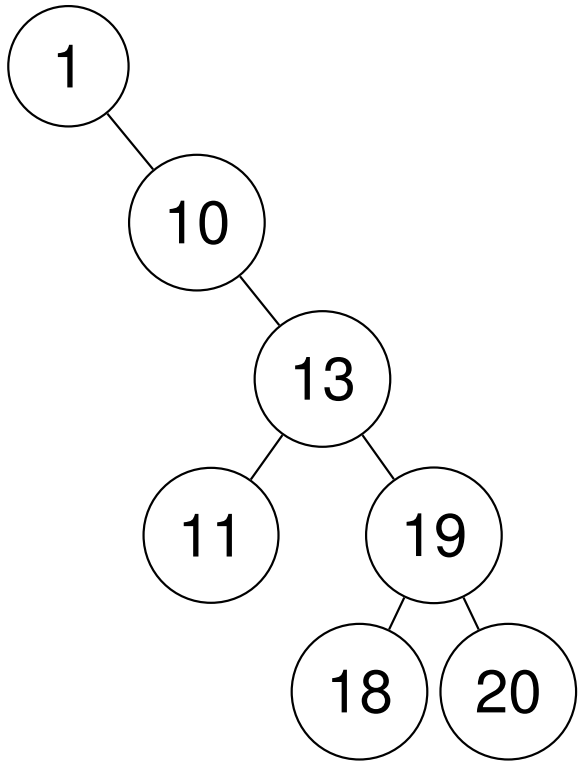
Reconstrucción de un árbol binario de búsqueda

- Preorden: 1,10,13,11,19,18,20

Reconstrucción de árboles binarios de búsqueda

Reconstrucción de un árbol binario de búsqueda

- Preorden: 1,10,13,11,19,18,20



Reconstrucción de árboles binarios de búsqueda

Ejercicio

Reconstruye un árbol binario de búsqueda a partir del siguiente recorrido

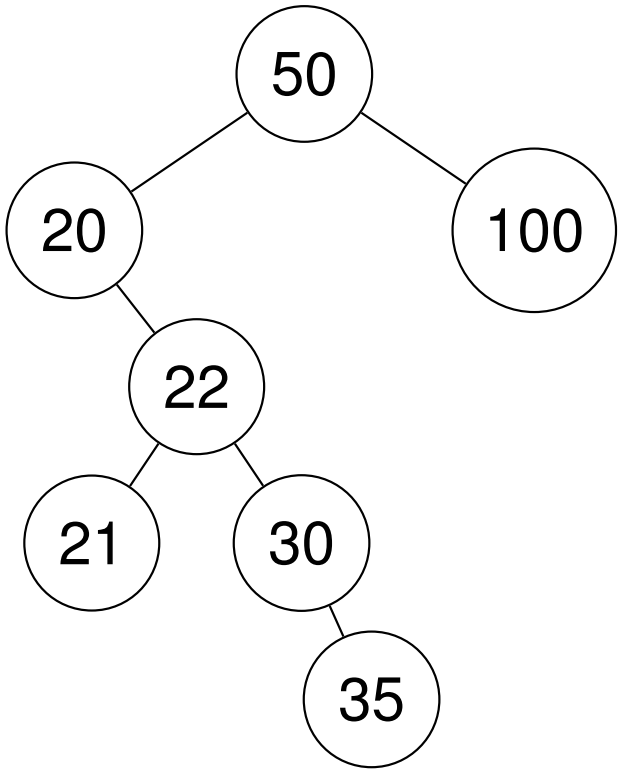
- Preorden: 50,20,22,21,30,35,100

Reconstrucción de árboles binarios de búsqueda

Ejercicio

Reconstruye un árbol binario de búsqueda a partir del siguiente recorrido

- Preorden: 50,20,22,21,30,35,100



Operaciones sobre árboles binarios de búsqueda

- Abordaremos las operaciones de búsqueda, inserción y borrado desde el punto de vista de las estructuras de datos
- Más adelante, definiremos tipos abstractos de datos (conjuntos) que pueden ser implementados mediante árboles binarios de búsqueda

Búsqueda

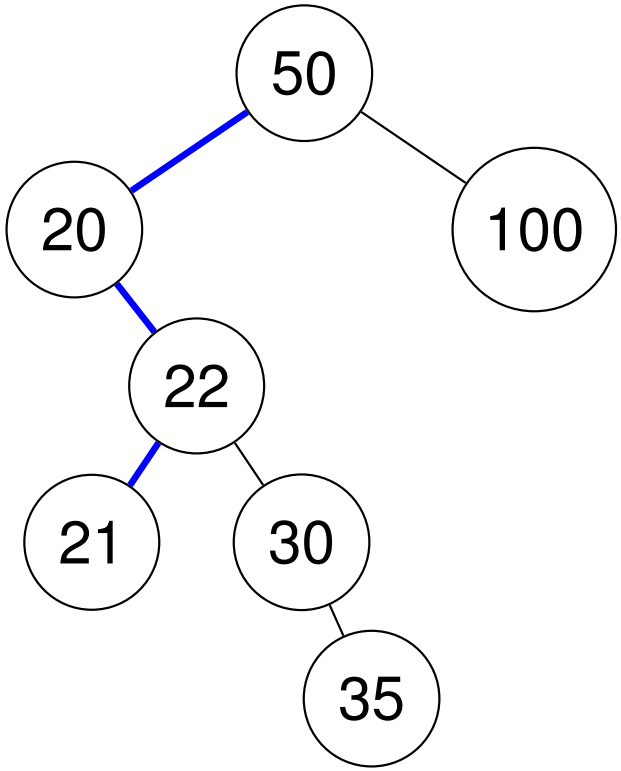
Algoritmo de búsqueda de un elemento k en un árbol binario de búsqueda:

- Si el árbol está vacío, k no está en el árbol
- En caso contrario, comparar k con el nodo raíz del árbol, al que llamaremos r:
  - Si $k = r$, el elemento está en el árbol
  - Si $k < r$, buscar k en el subárbol izquierdo
  - Si $k > r$, buscar k en el subárbol derecho

Búsqueda

Ejemplo

Búsqueda del elemento 21



Búsqueda

Pregunta

¿Cuál es la complejidad temporal de buscar en un árbol binario de búsqueda con n nodos?

Búsqueda

Pregunta

¿Cuál es la complejidad temporal de buscar en un árbol binario de búsqueda con n nodos?

- Mejor caso: el elemento es la raíz: $\Omega(1)$
- Peor caso: el elemento no está $\rightarrow$ coste proporcional a la altura $\rightarrow$

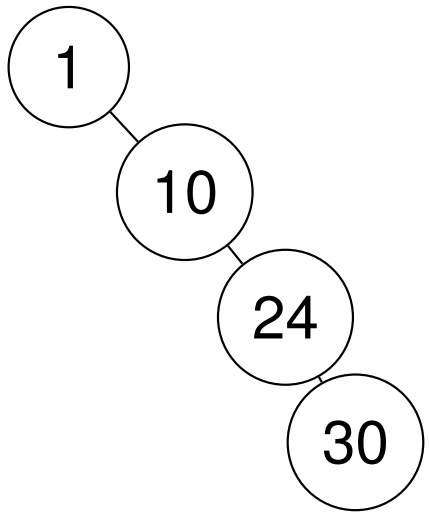
Búsqueda

Pregunta

¿Cuál es la complejidad temporal de buscar en un árbol binario de búsqueda con n nodos?

- Mejor caso: el elemento es la raíz: $\Omega(1)$
- Peor caso: el elemento no está $\rightarrow$ coste proporcional a la altura $\rightarrow O(n)$

Árbol degenerado:



Búsqueda

Pregunta

¿Cuál es la complejidad temporal de buscar en un árbol binario de búsqueda perfecto con n nodos?

Búsqueda

Pregunta

¿Cuál es la complejidad temporal de buscar en un árbol binario de búsqueda perfecto con n nodos?

- Mejor caso: el elemento es la raíz: $\Omega(1)$
- Peor caso: el elemento no está $\rightarrow$ coste proporcional a la altura $\rightarrow$

Búsqueda

Pregunta

¿Cuál es la complejidad temporal de buscar en un árbol binario de búsqueda perfecto con n nodos?

- Mejor caso: el elemento es la raíz: $\Omega(1)$
- Peor caso: el elemento no está $\rightarrow$ coste proporcional a la altura $\rightarrow O(\log n)$

Demostración:

- Un árbol perfecto de altura h tiene $2^h - 1$ nodos
- Un árbol perfecto con n nodos tiene la siguiente altura:

$$n = 2^h - 1$$

$$n + 1 = 2^h$$

$$\log_2(n + 1) = h$$

Inserción

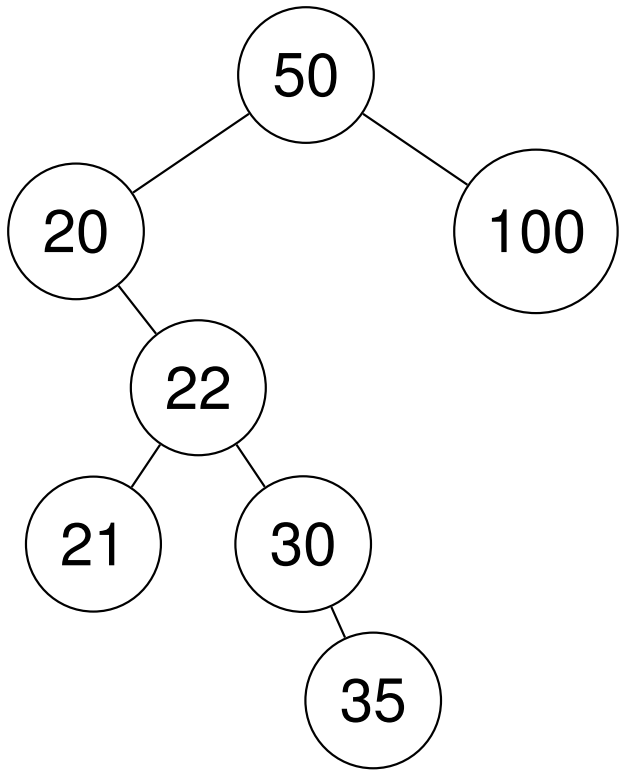
Algoritmo de inserción de un elemento k en un árbol binario de búsqueda:

- Si el árbol está vacío, k pasa a ser la raíz del árbol
- En caso contrario, comparar k con el nodo raíz del árbol, al que llamaremos r:
  - Si $k = r$, no se puede insertar
  - Si $k < r$, insertar k en el subárbol izquierdo
  - Si $k > r$, insertar k en el subárbol derecho

Inserción

Ejemplo

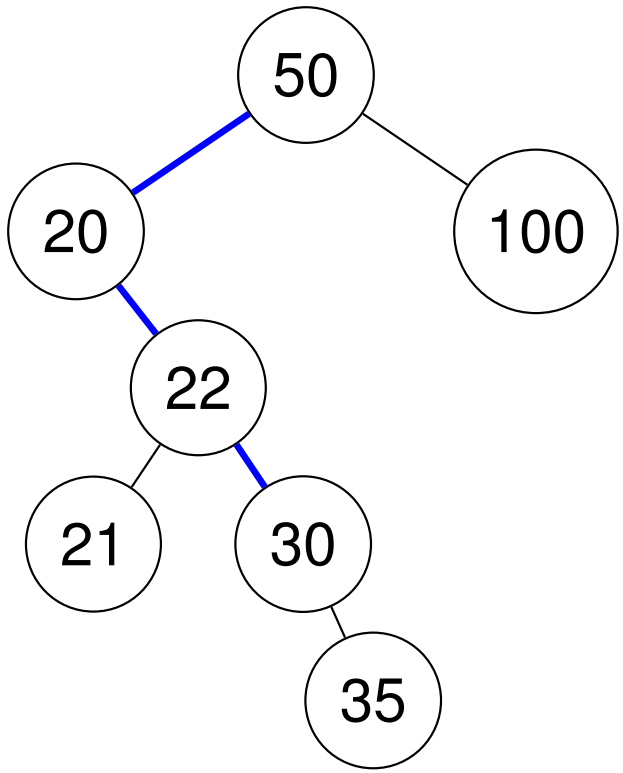
Inserción del elemento 28



Inserción

Ejemplo

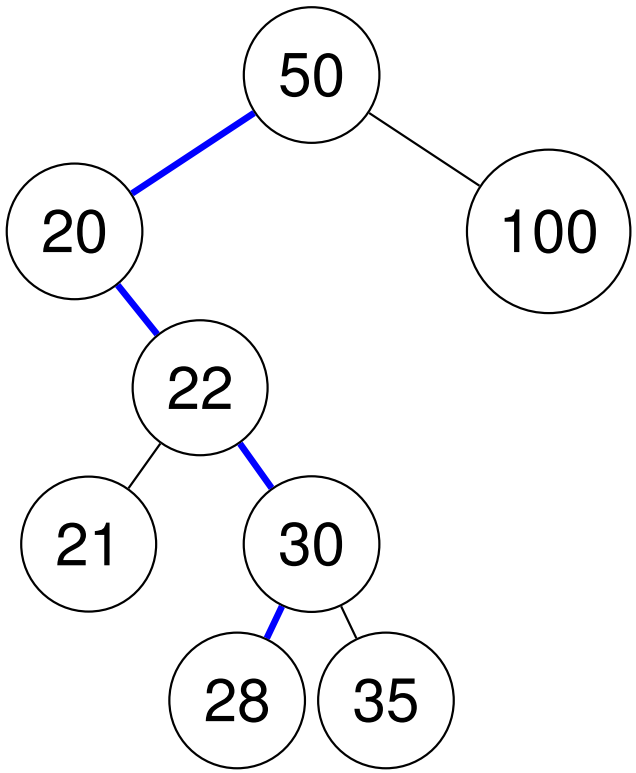
Inserción del elemento 28



Inserción

Ejemplo

Inserción del elemento 28



Inserción

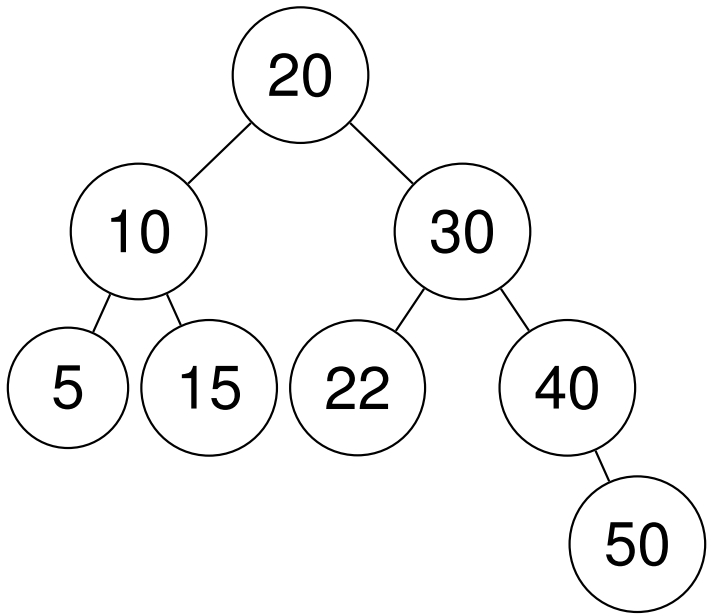
Ejercicio

Inserta los siguientes elementos en un árbol binario de búsqueda inicialmente vacío: 20, 10, 30, 40, 5, 15, 50, 22,

Inserción

Ejercicio

Inserta los siguientes elementos en un árbol binario de búsqueda inicialmente vacío: 20, 10, 30, 40, 5, 15, 50, 22,



Inserción

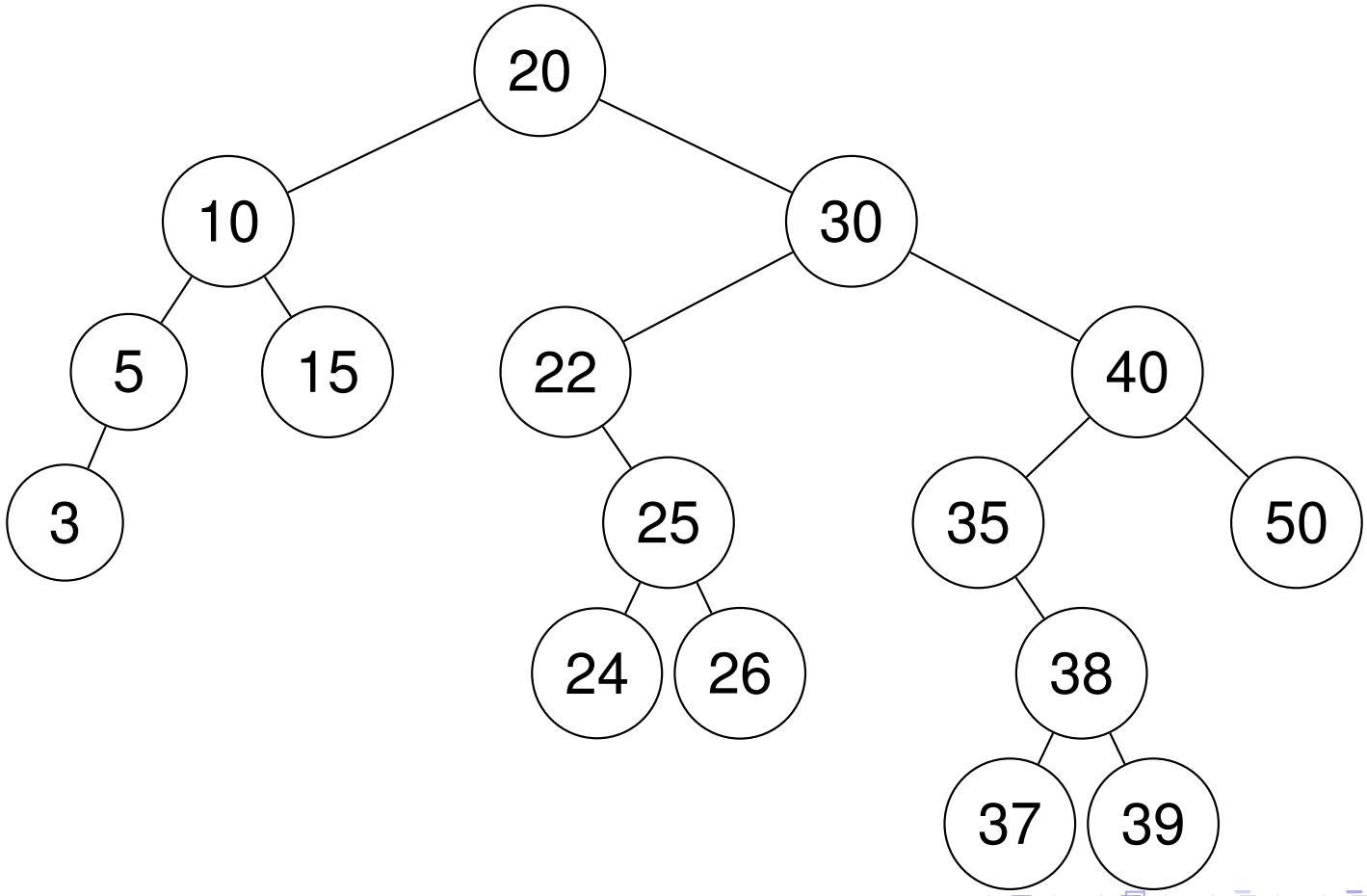
Ejercicio

Inserta los siguientes elementos en el árbol obtenido en el ejercicio anterior: 25, 24, 26, 3, 35, 38, 39, 37

Inserción

Ejercicio

Inserta los siguientes elementos en el árbol obtenido en el ejercicio anterior: 25, 24, 26, 3, 35, 38, 39, 37



Borrado

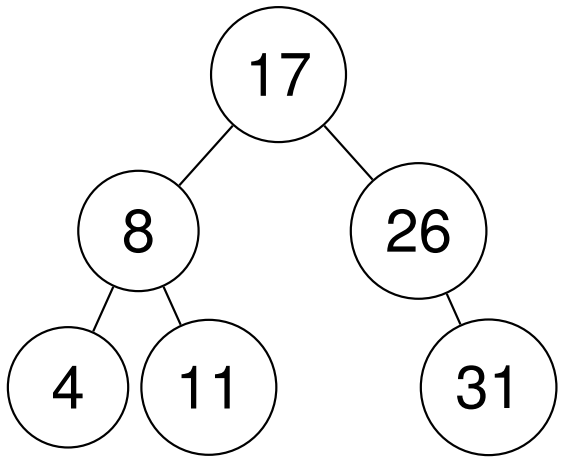
Algoritmo de borrado de un elemento k en un árbol binario de búsqueda:

Se aplica el algoritmo de búsqueda hasta encontrar k. Se pueden dar tres situaciones:

- k está en un nodo hoja $\rightarrow$ se elimina el nodo directamente

Ejemplo

Borrado del 11



Borrado

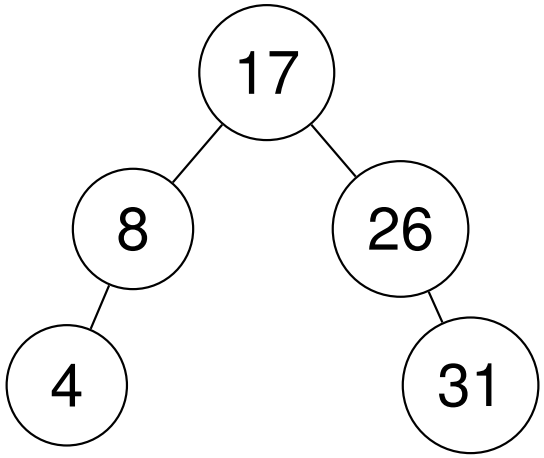
Algoritmo de borrado de un elemento k en un árbol binario de búsqueda:

Se aplica el algoritmo de búsqueda hasta encontrar k. Se pueden dar tres situaciones:

- k está en un nodo hoja $\rightarrow$ se elimina el nodo directamente

Ejemplo

Borrado del 11



Borrado

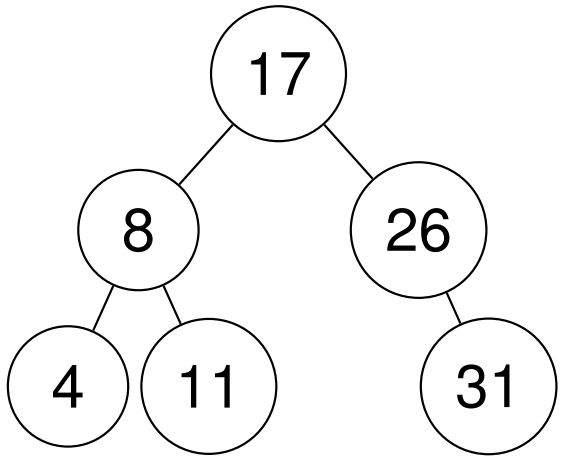
Algoritmo de borrado de un elemento k en un árbol binario de búsqueda:

Se aplica el algoritmo de búsqueda hasta encontrar k. Se pueden dar tres situaciones:

- k está en un nodo que tiene un solo hijo $\rightarrow$ el nodo se sustituye por su único hijo

Ejemplo

Borrado del 26



Borrado

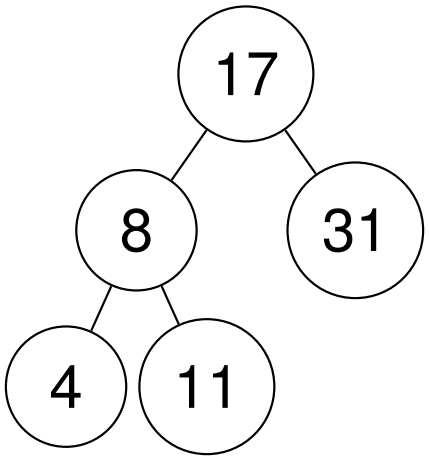
Algoritmo de borrado de un elemento k en un árbol binario de búsqueda:

Se aplica el algoritmo de búsqueda hasta encontrar k. Se pueden dar tres situaciones:

- k está en un nodo que tiene un solo hijo $\rightarrow$ el nodo se sustituye por su único hijo

Ejemplo

Borrado del 26



Borrado

Algoritmo de borrado de un elemento k en un árbol binario de búsqueda:

Se aplica el algoritmo de búsqueda hasta encontrar k. Se pueden dar tres situaciones:

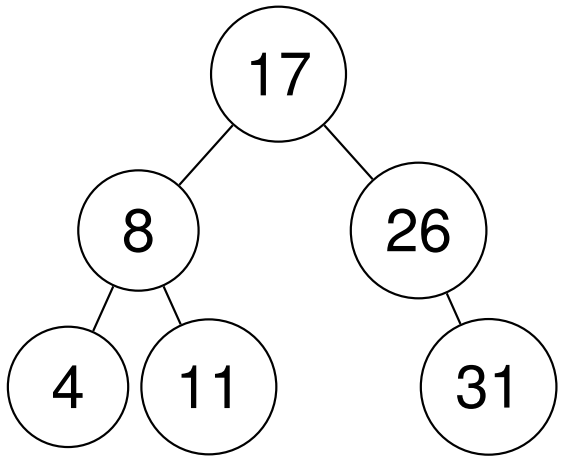
- k está en un nodo con dos hijos $\rightarrow$ k se sustiye por el mayor elemento de su subárbol izquierdo (M_i) o por el menor elemento de su subárbol derecho (m_d); a continuación, se vuelve a aplicar el algoritmo para borrar M_i o m_d

Borrado

Algoritmo de borrado de un elemento k en un árbol binario de búsqueda:

Ejemplo

Borrado del 17. Estrategia: sustitución por M_i

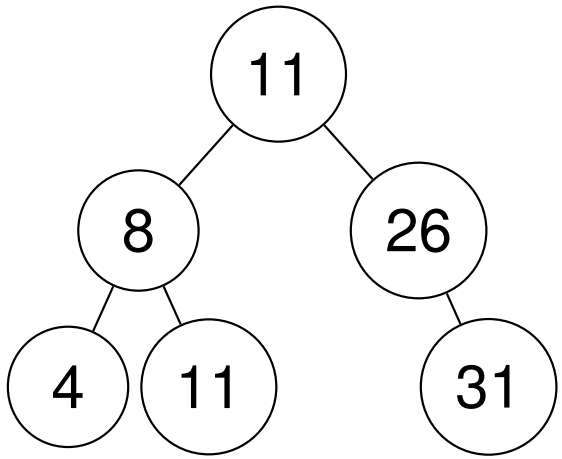


Borrado

Algoritmo de borrado de un elemento k en un árbol binario de búsqueda:

Ejemplo

Borrado del 17. Estrategia: sustitución por M_i

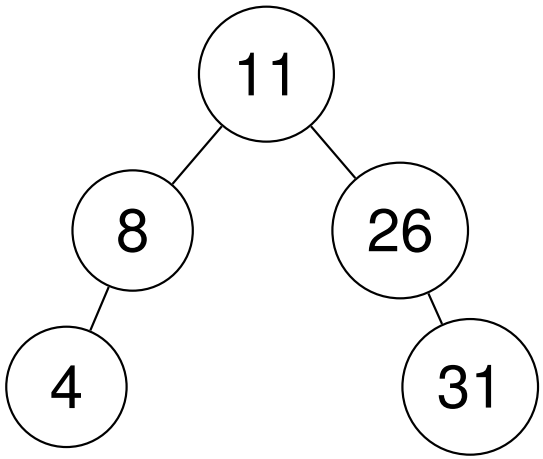


Borrado

Algoritmo de borrado de un elemento k en un árbol binario de búsqueda:

Ejemplo

Borrado del 17. Estrategia: sustitución por M_i

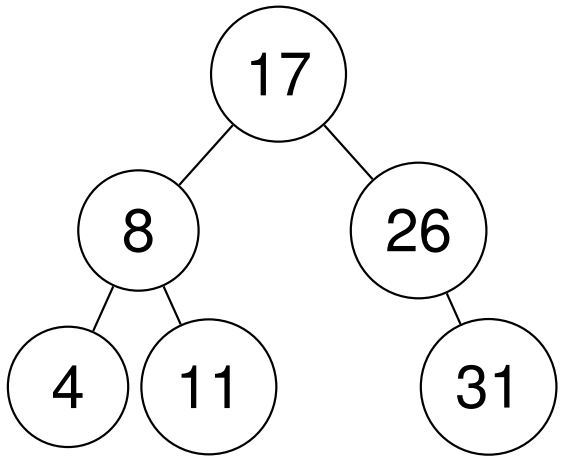


Borrado

Algoritmo de borrado de un elemento k en un árbol binario de búsqueda:

Ejemplo

Borrado del 17. Estrategia: sustitución por m_d

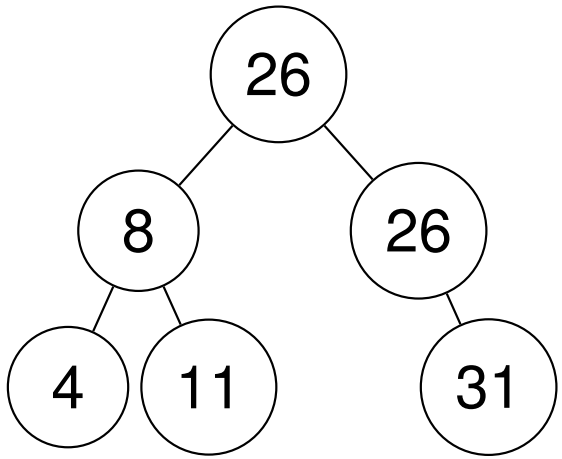


Borrado

Algoritmo de borrado de un elemento k en un árbol binario de búsqueda:

Ejemplo

Borrado del 17. Estrategia: sustitución por m_d

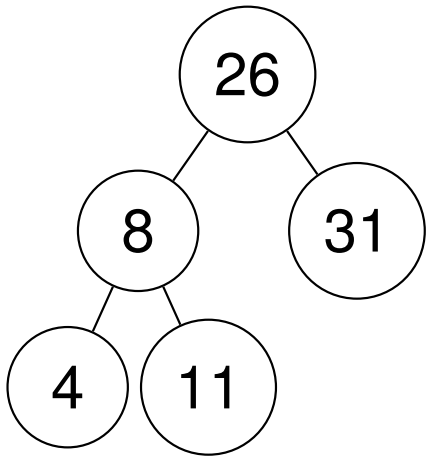


Borrado

Algoritmo de borrado de un elemento k en un árbol binario de búsqueda:

Ejemplo

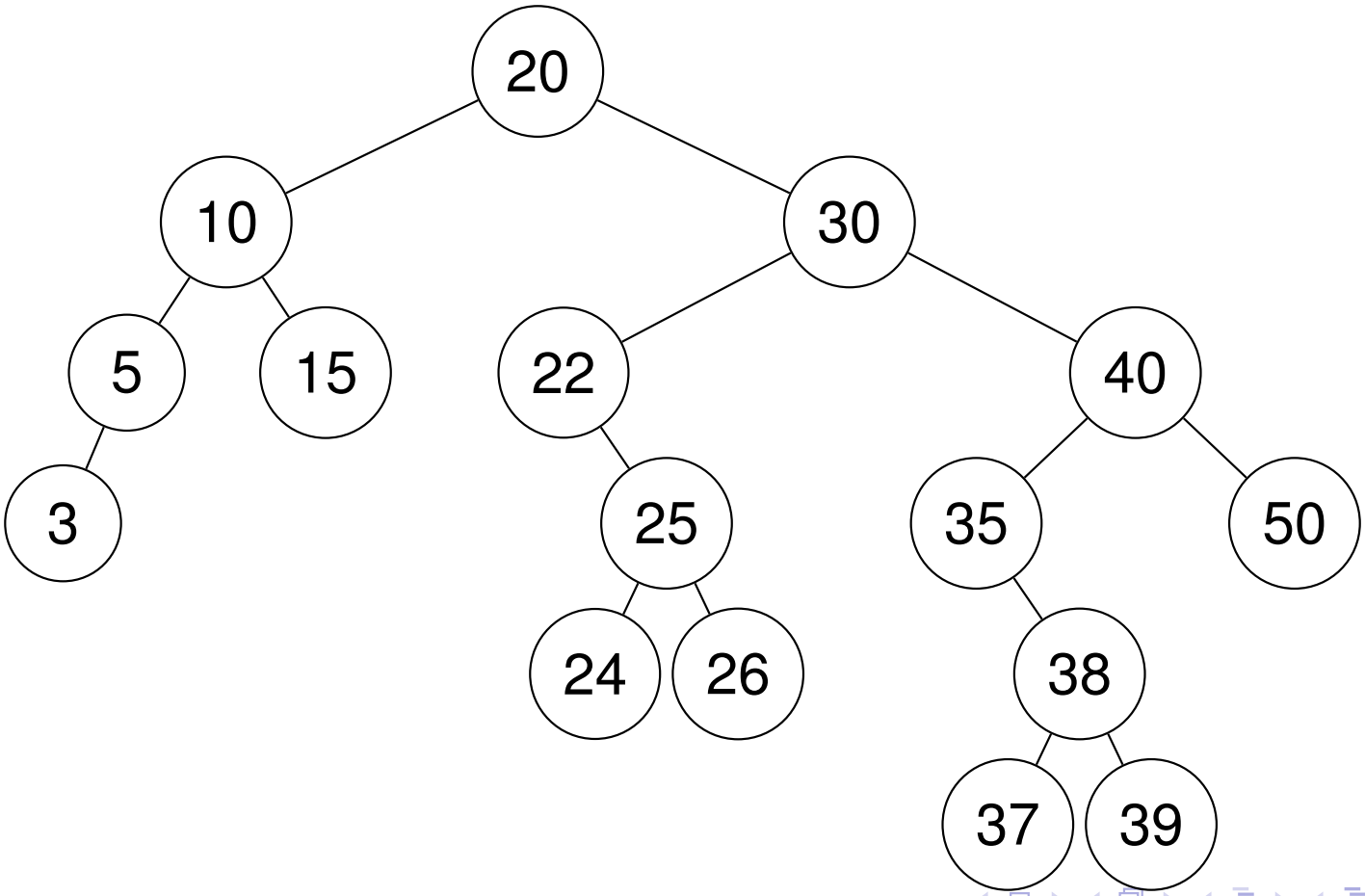
Borrado del 17. Estrategia: sustitución por m_d



Borrado

Ejercicio

Borra los siguientes elementos del árbol que se muestra a continuación: 5, 3, 30, 22, 39. Criterio: sustitución por m_d

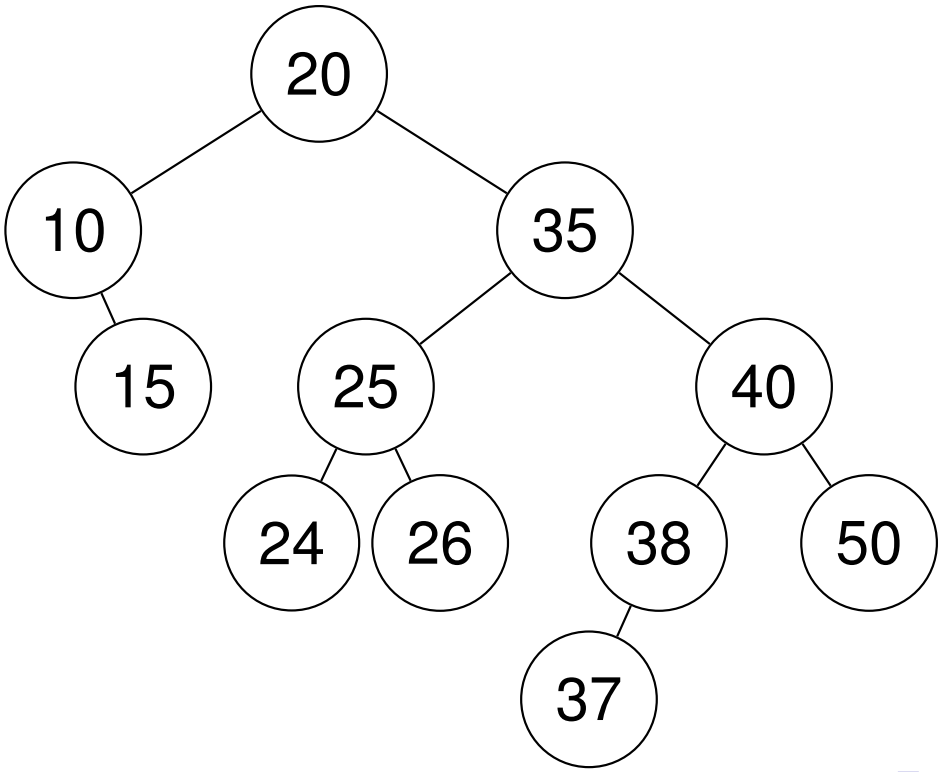


Borrado

Ejercicio

Borra los siguientes elementos del árbol que se muestra a continuación: 5, 3, 30, 22, 39. Criterio: sustitución por m_d

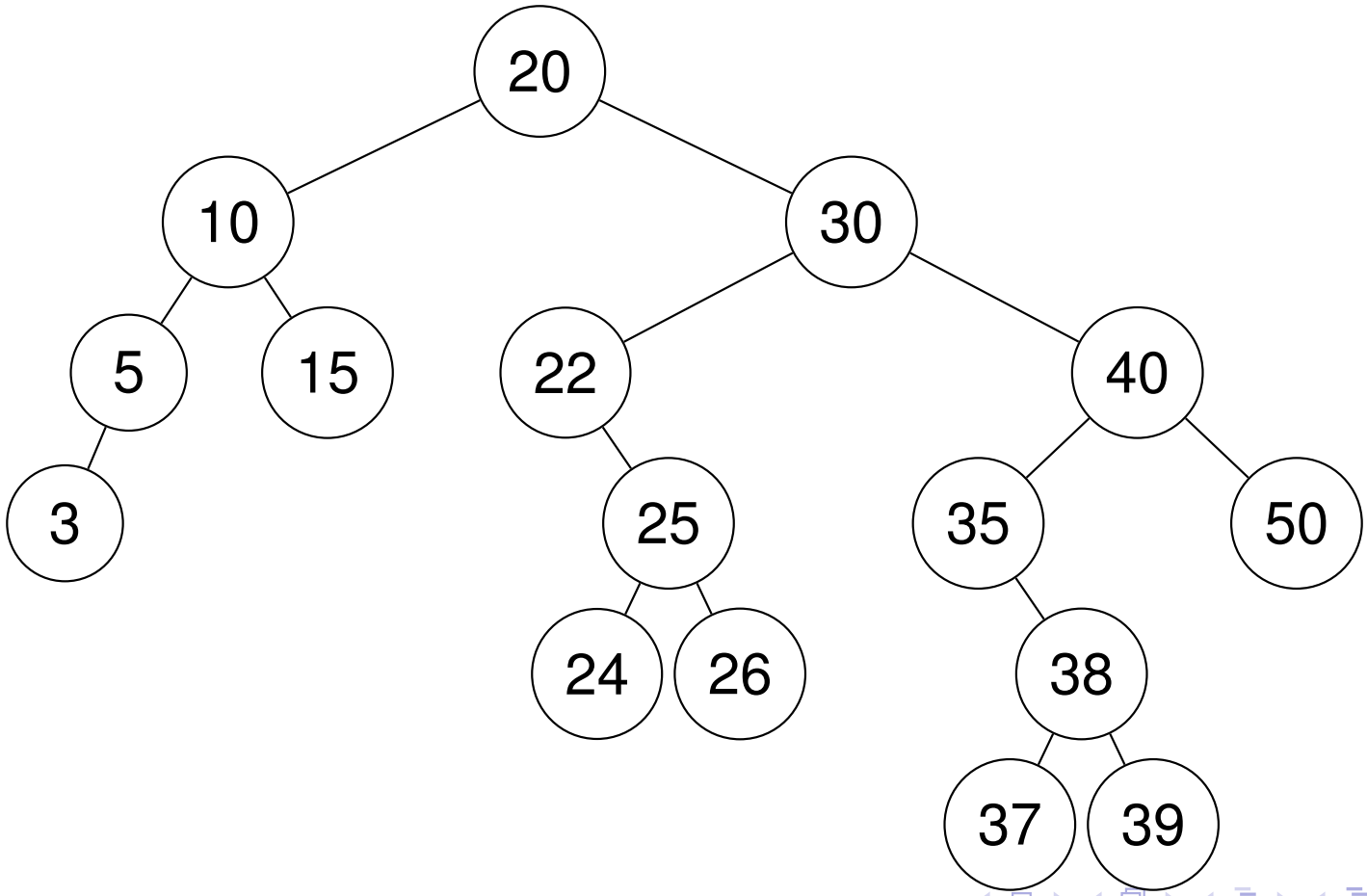
Solución:



Borrado

Ejercicio

Borra los siguientes elementos del árbol que se muestra a continuación: 5, 3, 30, 22, 39. Criterio: sustitución por M_i

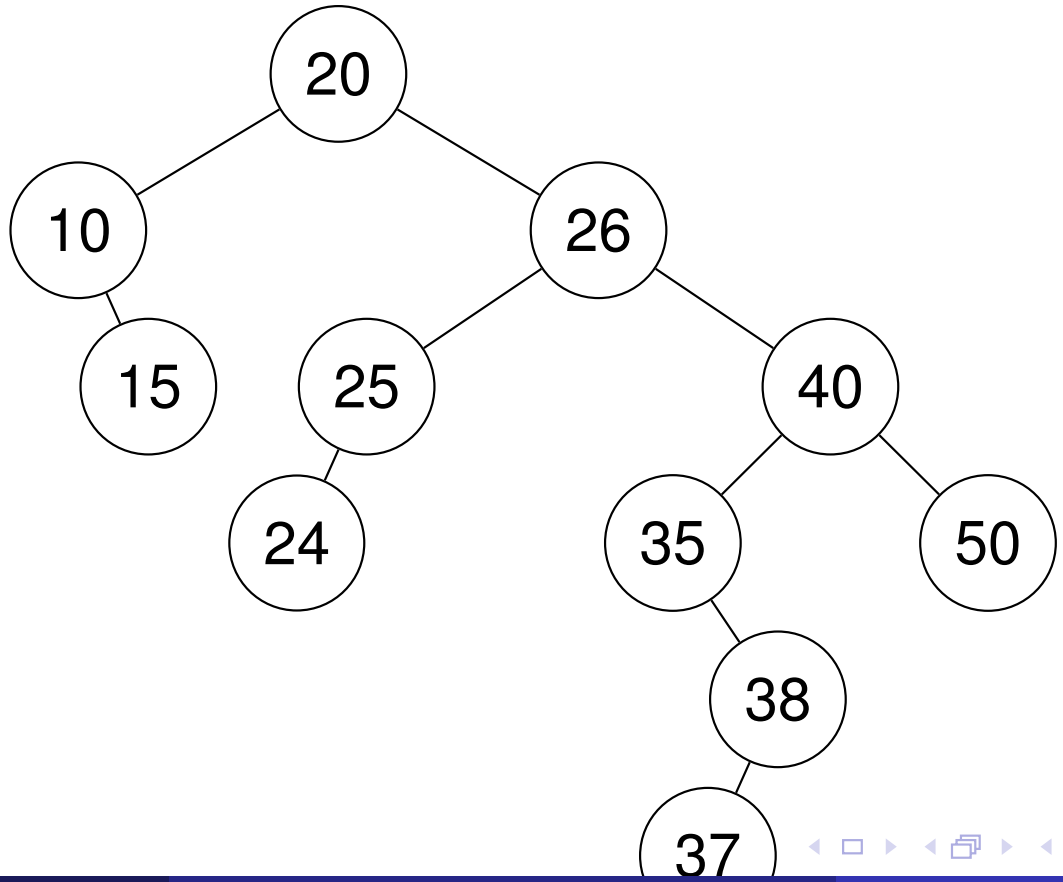


Borrado

Ejercicio

Borra los siguientes elementos del árbol que se muestra a continuación: 5, 3, 30, 22, 39. Criterio: sustitución por M_i

Solución:



Búsqueda

Pregunta

¿Cuál es la complejidad temporal de insertar un elemento en un árbol binario de búsqueda con n nodos?¿Y de borrar un elemento? (con implementación enlazada)

Búsqueda

Pregunta

¿Cuál es la complejidad temporal de insertar un elemento en un árbol binario de búsqueda con n nodos?¿Y de borrar un elemento? (con implementación enlazada)

- Inserción: $\Omega(1)$; $O(n)$
- Borrado: $\Omega(1)$; $O(n)$

Pregunta

¿Podrías dibujar un ejemplo de árbol para cada caso?

Para practicar

`https://www.cs.usfca.edu/~galles/visualization/BST.html`

Criterio: M_i

Para practicar

- Inserta los elementos 1,2,3,4,5,6,7,8,9 en un árbol binario de búsqueda inicialmente vacío. ¿Cuál es la complejidad de insertar n elementos ordenados ascendentemente?
- Borra los elementos en el árbol anterior en el mismo orden. ¿Cuál es la complejidad de cada borrado? (con implementación enlazada)

Para practicar

- Inserta los elementos 16, 6, 27, 3, 8, 23, 15, 25, 5, 19, 12, 7, 17, 24 (generados aleatoriamente) en un árbol binario de búsqueda inicialmente vacío. ¿Cuál piensas que es la altura promedio de un árbol binario de búsqueda tras n inserciones?
- Borra (M_i) elementos del árbol obtenido en el apartado anterior, en el siguiente orden: 25, 17, 23, 24, 6, 3, 16

Repaso

Pregunta

Contesta verdadero o falso a cada una de estas afirmaciones

- El menor elemento en un árbol binario de búsqueda siempre se encuentra en un nodo hoja
- Al borrar un elemento que se encuentra en un nodo con dos hijos, sustituir el valor a borrar por el menor elemento del hijo izquierdo es una estrategia válida
- El coste temporal, en el peor caso, de insertar un elemento en un árbol binario de búsqueda perfecto (con implementación enlazada) es lineal

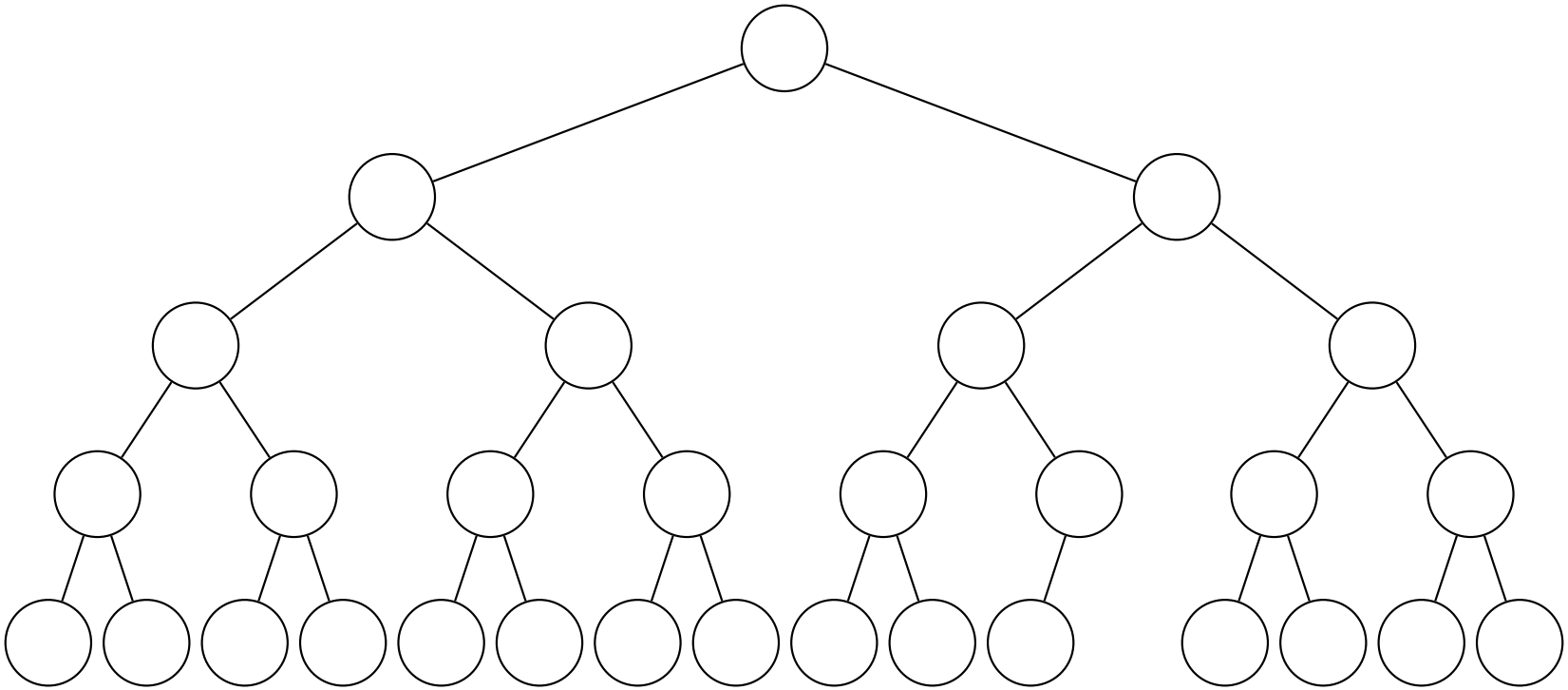
Índice

Árboles AVL: motivación

- El coste de buscar un elemento en un árbol binario de búsqueda en el peor caso es $O(n)$ (árbol degenerado)
- En un árbol de búsqueda perfecto, dicho coste es $O(\log(n))$
- **Idea**: ¿Podemos modificar un ABB tras cada inserción o borrado para que siempre sea lo más parecido posible a un árbol perfecto?
- **Requisito:** Realizar esas modificaciones en tiempo constante respecto al número de elementos almacenados en el árbol
- **Resultado:** Inserciones, búsquedas y borrados en $O(\log(n))$

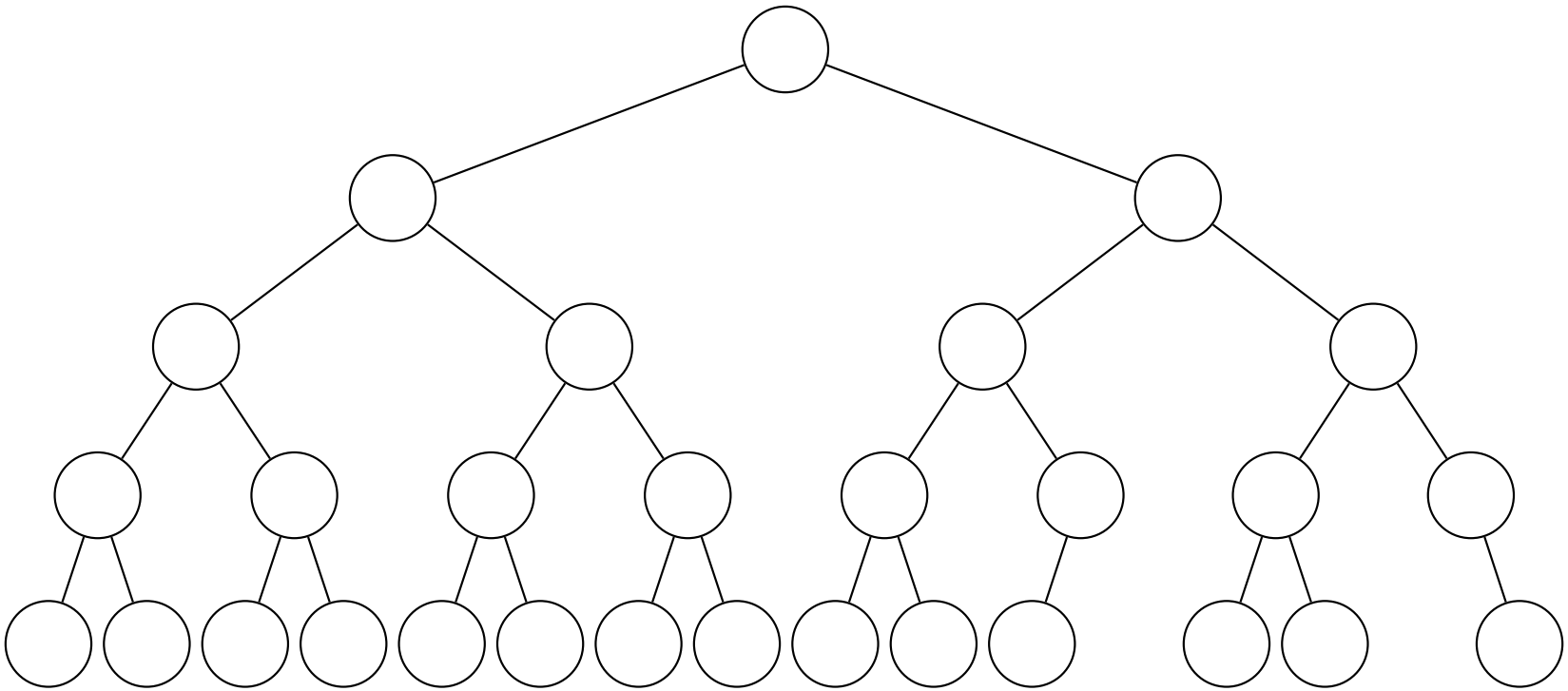
Árboles completamente equilibrados

- Un árbol perfecto, por definición, tiene un número impar de elementos
- Un árbol **completamente equilibrado** es aquel en el que, para todo nodo, el número de nodos de su subárbol izquierdo y de su subárbol derecho difiere como mucho en uno:



Árboles completamente equilibrados

- El siguiente árbol no es completamente equilibrado:



Árboles AVL

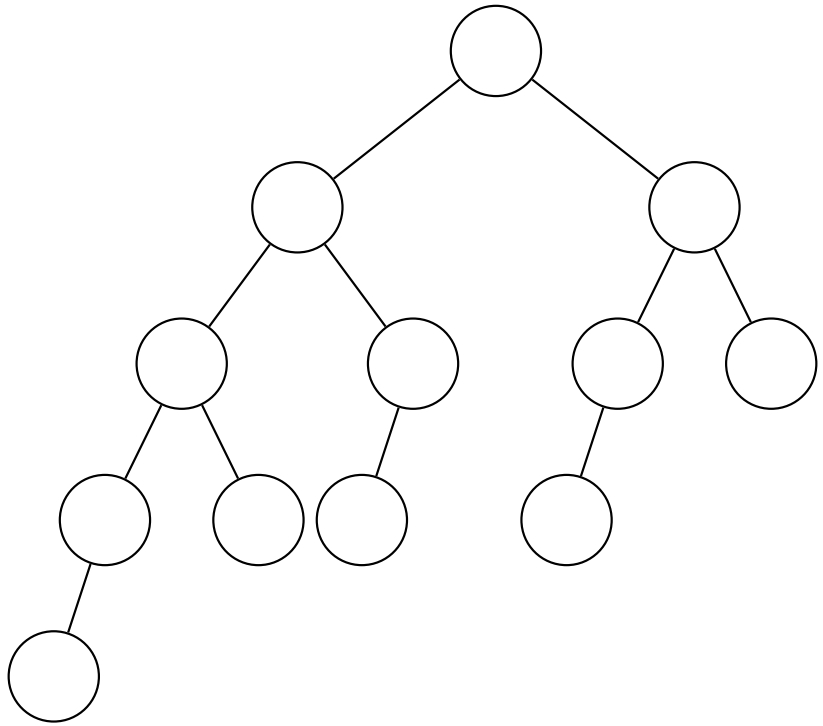
- Los árboles completamente equilibrados requerirían la modificación del árbol tras prácticamente cada inserción o borrado
- **Árboles AVL**: árboles equilibrados respecto a la altura, desarrollados por Adelson-Velskii y Landis en 1962

Definición

Un árbol está equilibrado respecto a la altura si y sólo si, para cada uno de sus nodos, las **alturas** de sus subárboles izquierdo y derecho difieren como máximo en 1

Pregunta

¿El siguiente árbol es completamente equilibrado? ¿Y equilibrado respecto a la altura?



Árboles AVL

- Un árbol equilibrado respecto a la altura de altura h puede tener menos de $2^h - 1$ nodos
- No obstante, se ha demostrado que un árbol equilibrado respecto a la altura con n nodos tiene como máximo una altura de $2\log(n) + 2$
- La complejidad de la búsqueda en el peor caso es $O(\log(n))$

Factor de equilibrio

Definición

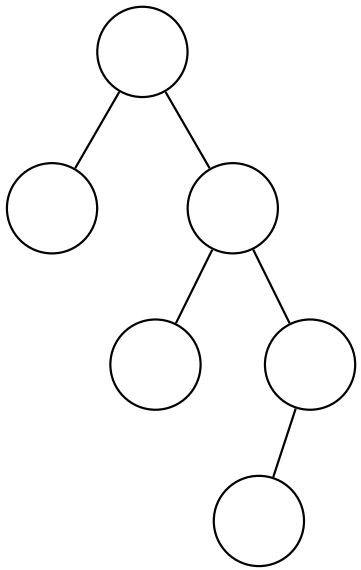
El factor de equilibrio (f_e) de un nodo se define como $h_r - h_l$, siendo h_r la altura de su subárbol derecho y h_l la altura de su subárbol izquierdo

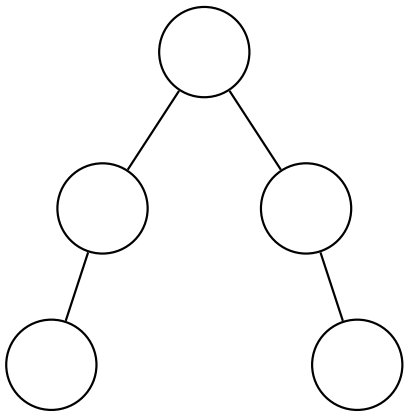
- Un árbol es equilibrado respecto a la altura si y sólo si todos sus nodos se cumple que $f_e \in \{-1, 0, 1\}$

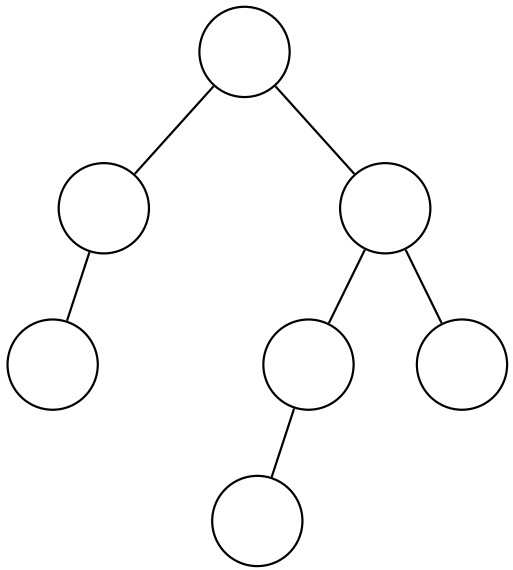
Factor de equilibrio

Pregunta

Calcula el factor de equilibrio de cada nodo para cada uno de estos árboles e indica si son equilibrados respecto a la altura







Implementación enlazada para árboles AVL

```
class Arbol{
private:
Nodo* root;
....
};

class Nodo{
private:
    Elem data;
    int balanceFactor; //Factor de equilibrio
    Nodo* parent;
    Nodo* left;
    Nodo* right;
...
};
```

Pregunta

¿Piensas que la implementación mediante vector es adecuada para un árbol binario de búsqueda o AVL?

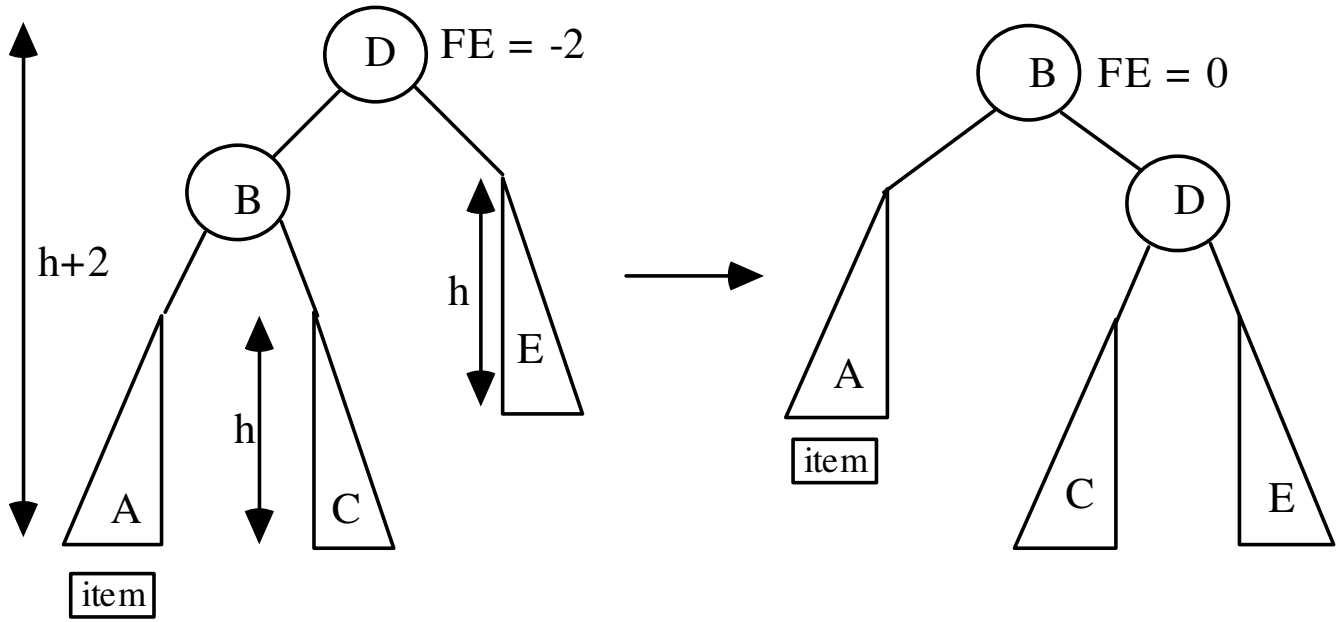
Inserción en árboles AVL

Algoritmo de inserción en árboles AVL:

1. Insertar el elemento siguiendo el algoritmo general de inserción en árboles binarios de búsqueda
2. Actualizar el factor de equilibrio de cada nodo en el camino del nodo insertado a la raíz
3. Tan pronto aparezca un factor de equilibrio 2 o -2, realizar una **rotación**

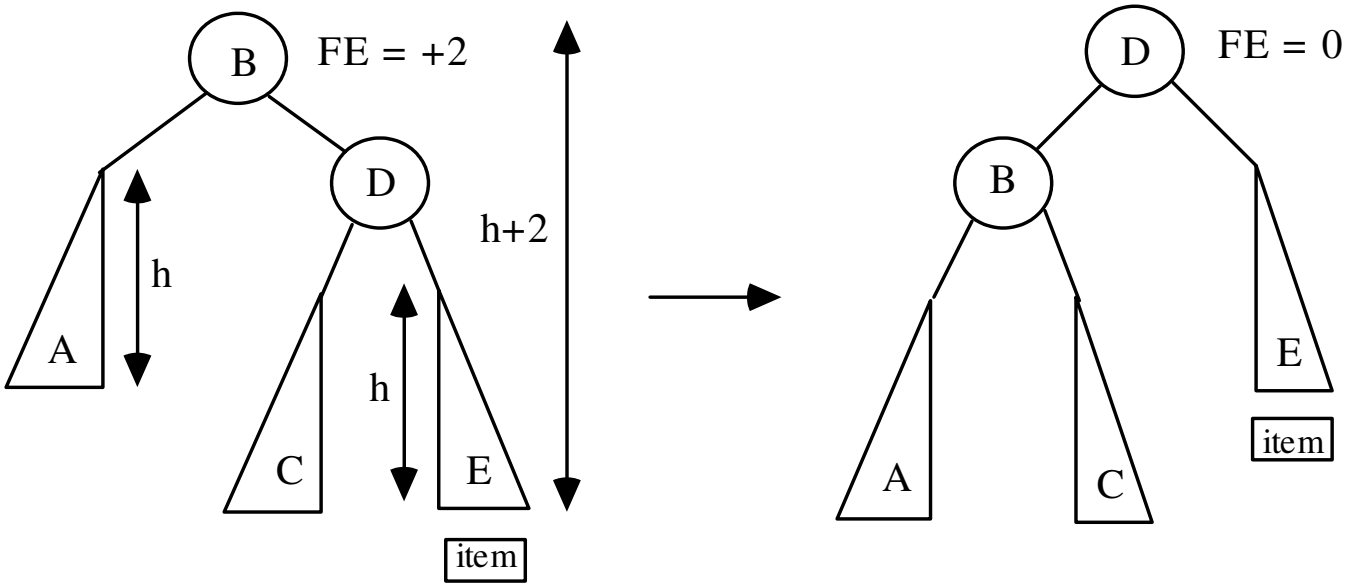
Inserción en árboles AVL

Rotación II: $f_e = -2$; hijo izquierdo con $f_e = -1$



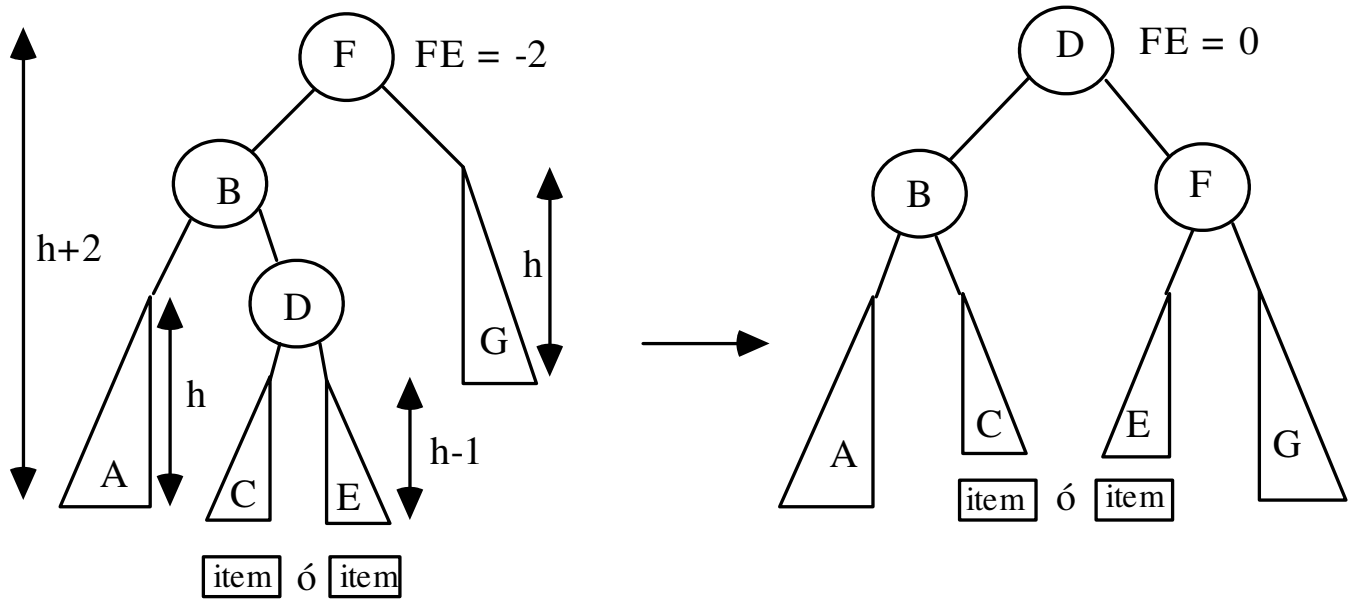
Inserción en árboles AVL

Rotación DD: $f_e = 2$; hijo derecho con $f_e = 1$



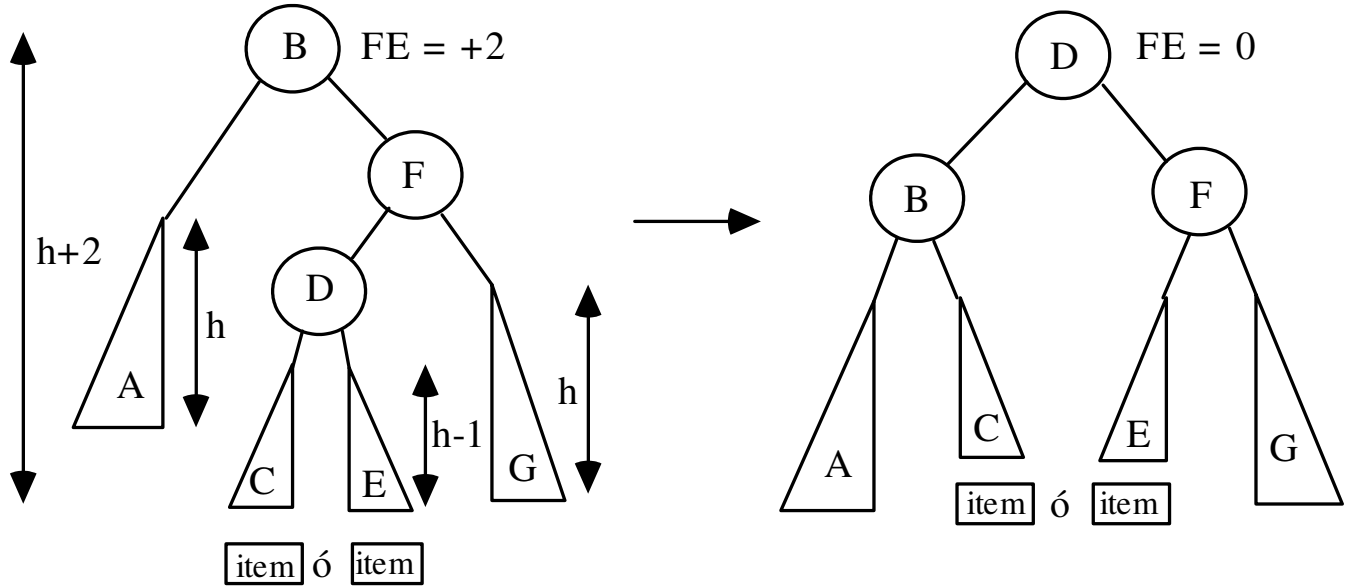
Inserción en árboles AVL

Rotación ID: $f_e = -2$; hijo izquierdo con $f_e = 1$



Inserción en árboles AVL

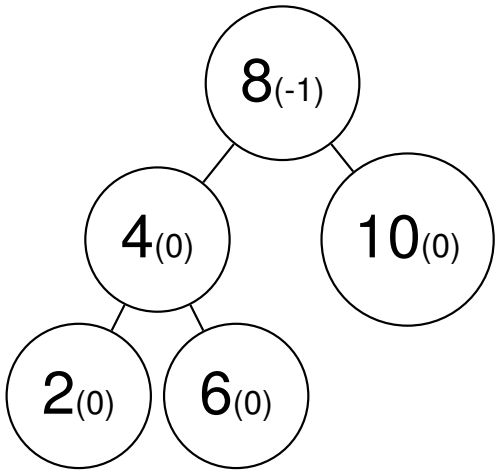
Rotación DI: $f_e = 2$; hijo derecho con $f_e = -1$



Inserción en árboles AVL

Ejemplo

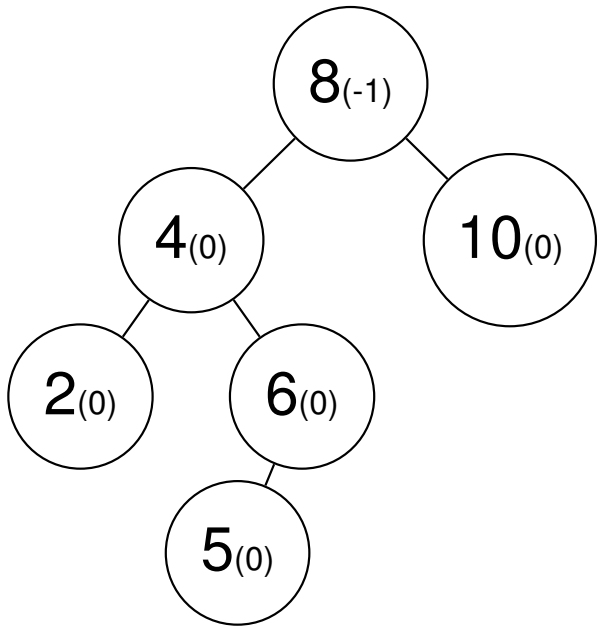
Inserta en el siguiente árbol AVL los elementos 5 y 12



Inserción en árboles AVL

Ejemplo

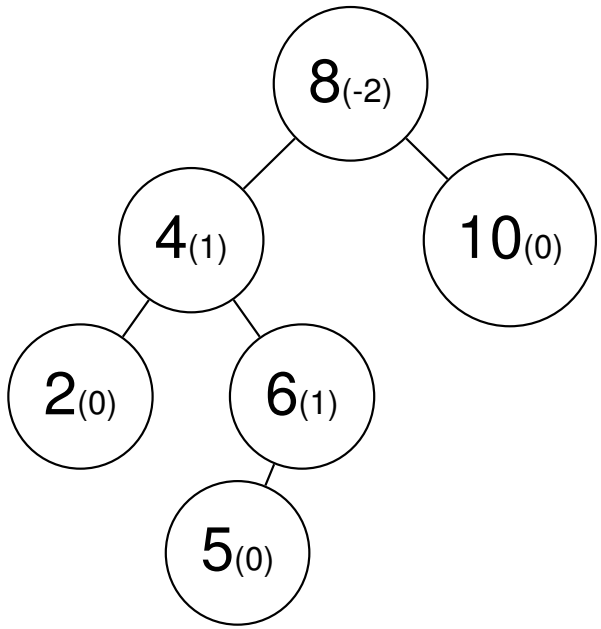
Inserta en el siguiente árbol AVL los elementos 5 y 12



Inserción en árboles AVL

Ejemplo

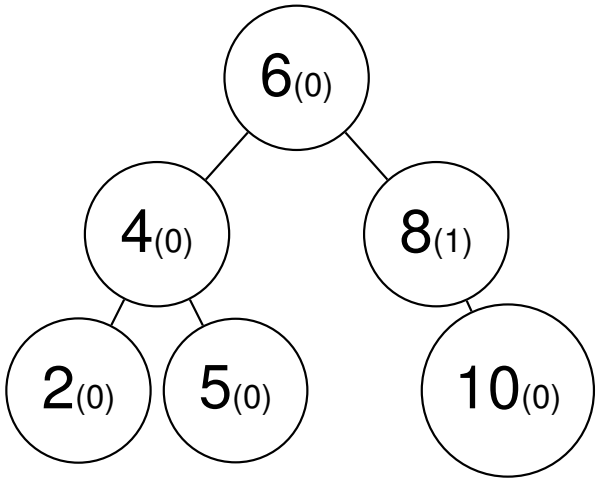
Inserta en el siguiente árbol AVL los elementos 5 y 12



Inserción en árboles AVL

Ejemplo

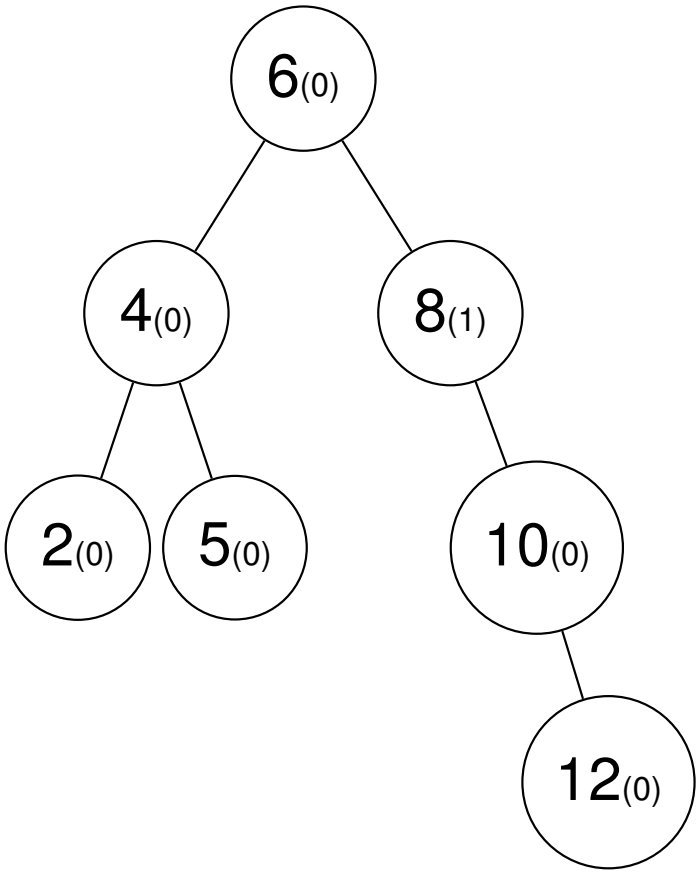
Inserta en el siguiente árbol AVL los elementos 5 y 12



Inserción en árboles AVL

Ejemplo

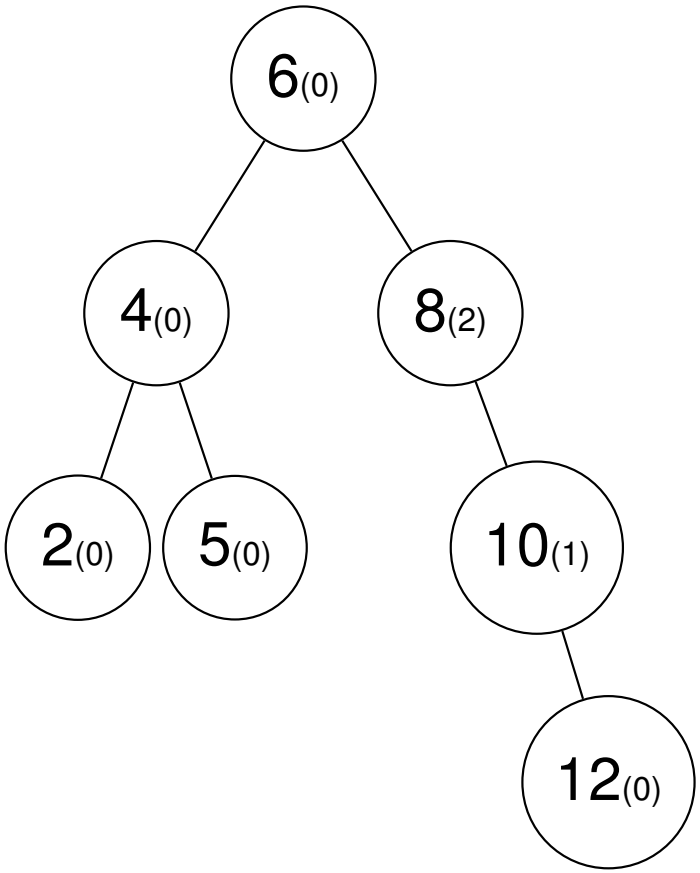
Inserta en el siguiente árbol AVL los elementos 5 y 12



Inserción en árboles AVL

Ejemplo

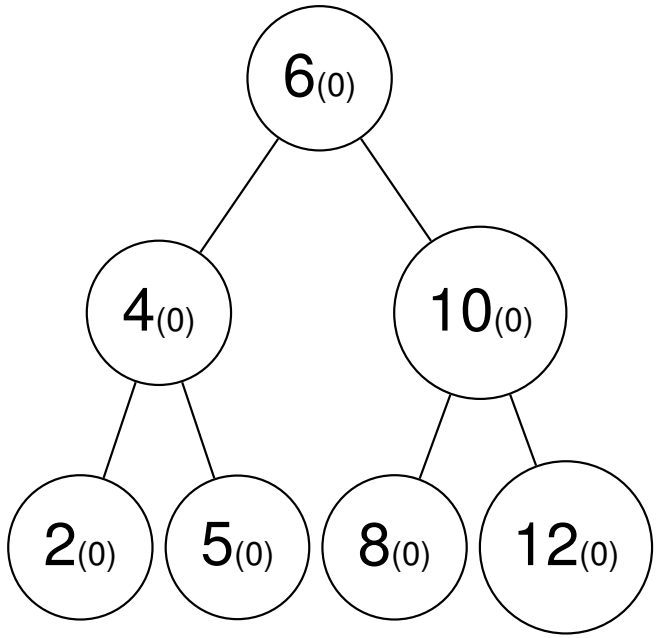
Inserta en el siguiente árbol AVL los elementos 5 y 12



Inserción en árboles AVL

Ejemplo

Inserta en el siguiente árbol AVL los elementos 5 y 12



Inserción en árboles AVL

Ejercicio

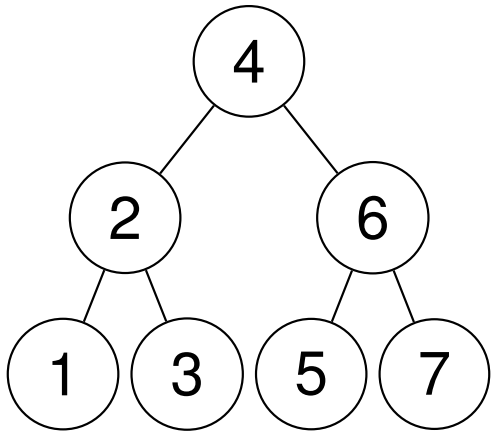
Inserta los siguientes elementos en un árbol AVL inicialmente vacío: 4, 5, 7, 2, 1, 3, 6. Indica los tipos de rotaciones realizadas.

Inserción en árboles AVL

Ejercicio

Inserta los siguientes elementos en un árbol AVL inicialmente vacío: 4, 5, 7, 2, 1, 3, 6. Indica los tipos de rotaciones realizadas.

Solución: DD, II, ID, DI



Inserción en árboles AVL

Ejercicio

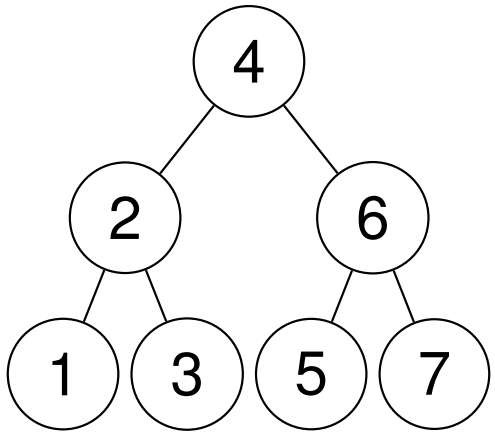
Inserta los siguientes elementos en un árbol AVL inicialmente vacío: 1, 2, 3, 4, 5, 6, 7. Indica los tipos de rotaciones realizadas.

Inserción en árboles AVL

Ejercicio

Inserta los siguientes elementos en un árbol AVL inicialmente vacío: 1, 2, 3, 4, 5, 6, 7. Indica los tipos de rotaciones realizadas.

Solución: DD, DD, DD, DD



Inserción en árboles AVL

Pregunta

¿Cuántas rotaciones como máximo se producen tras una inserción?¿Cuál es la complejidad temporal asintótica de una inserción en un árbol AVL respecto al número de nodos n del árbol?

Inserción en árboles AVL

Pregunta

¿Cuántas rotaciones como máximo se producen tras una inserción?¿Cuál es la complejidad temporal asintótica de una inserción en un árbol AVL respecto al número de nodos n del árbol?

- Una rotación como máximo
- $\Omega(1)$; $O(\log(n))$

Borrado en árboles AVL

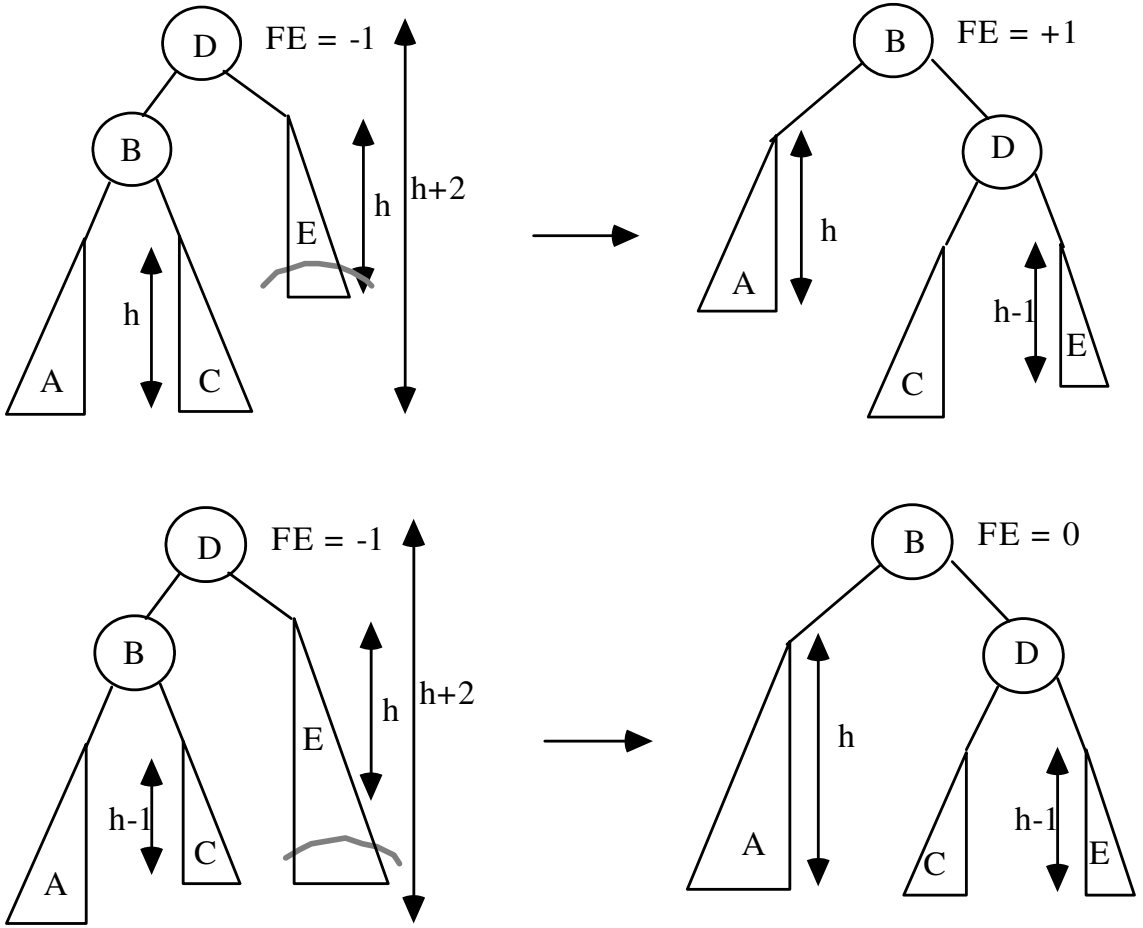
Algoritmo de borrado en árboles AVL:

1. Borrar el elemento siguiendo el algoritmo general de borrado en árboles binarios de búsqueda
2. Actualizar el factor de equilibrio de cada nodo en el camino del nodo borrado[1] a la raíz
3. Tan pronto aparezca un factor de equilibrio 2 o -2, realizar una **rotación**
4. Tras la rotación, continuar actualizando el factor de equilibria hasta la raíz, y repetir el paso 3 tantas veces como sea necesario

[1] Si se ha borrado un elemento en un nodo con dos hijos, hay que empezar a actualizar los factores de equilibrio desde el lugar donde estaban M_i o m_d

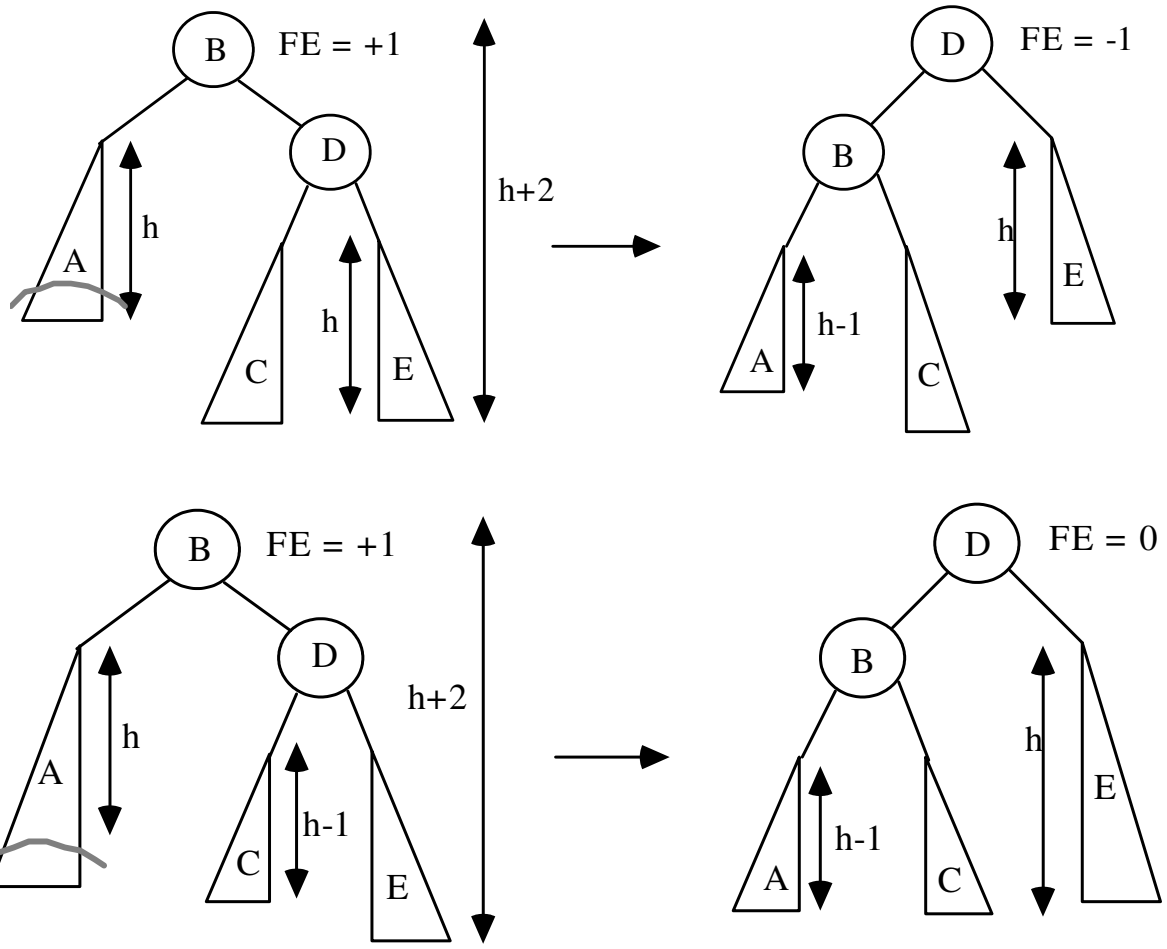
Borrado en árboles AVL

Rotación II: $f_e = -2$; hijo izquierdo con $f_e = 0$ o $f_e = -1$



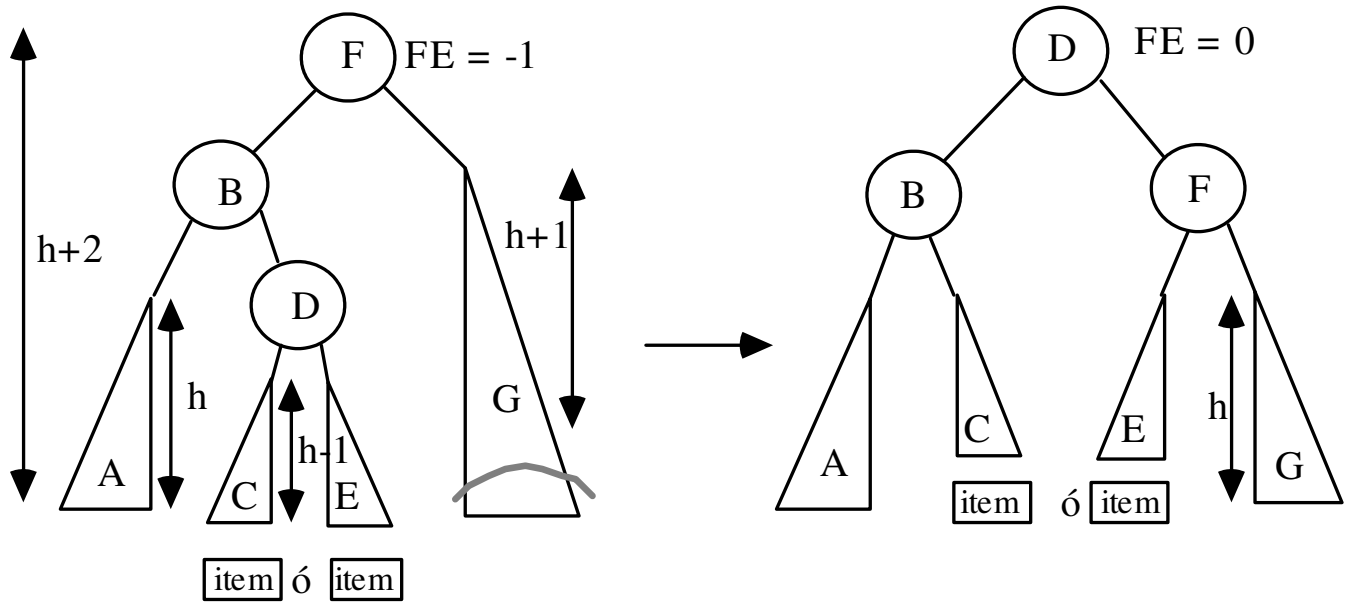
Borrado en árboles AVL

Rotación DD: $f_e = 2$; hijo izquierdo con $f_e = 0$ o $f_e = 1$



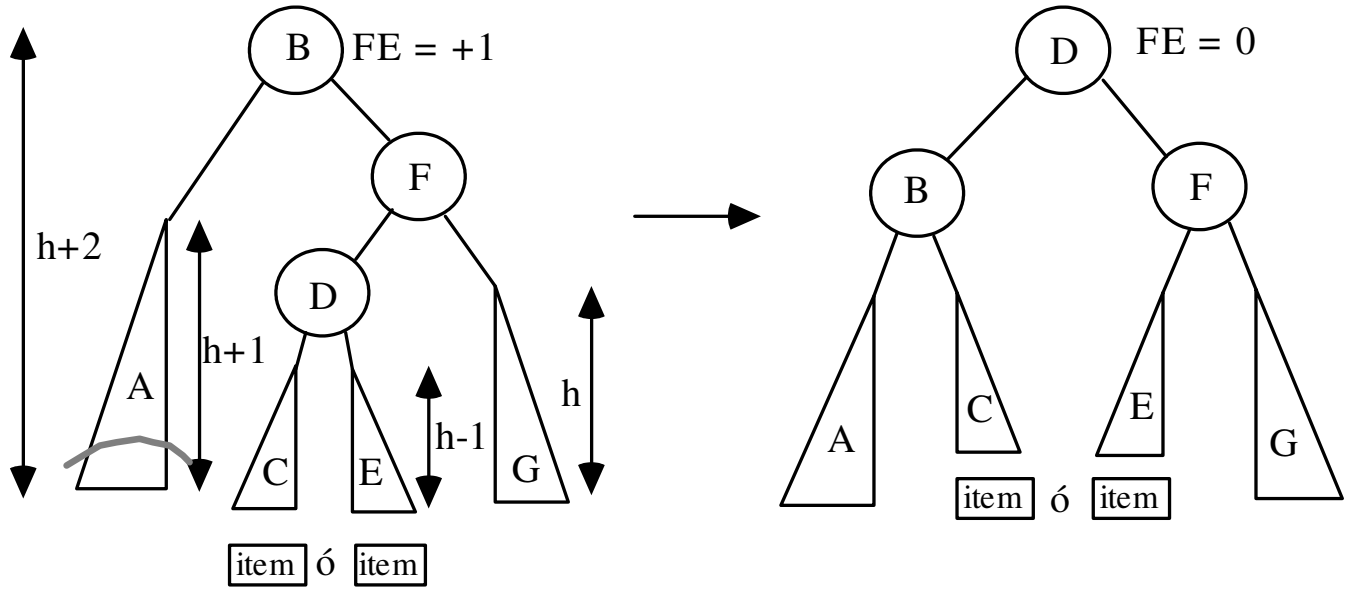
Borrado en árboles AVL

Rotación ID: $f_e = -2$; hijo izquierdo con $f_e = 1$



Borrado en árboles AVL

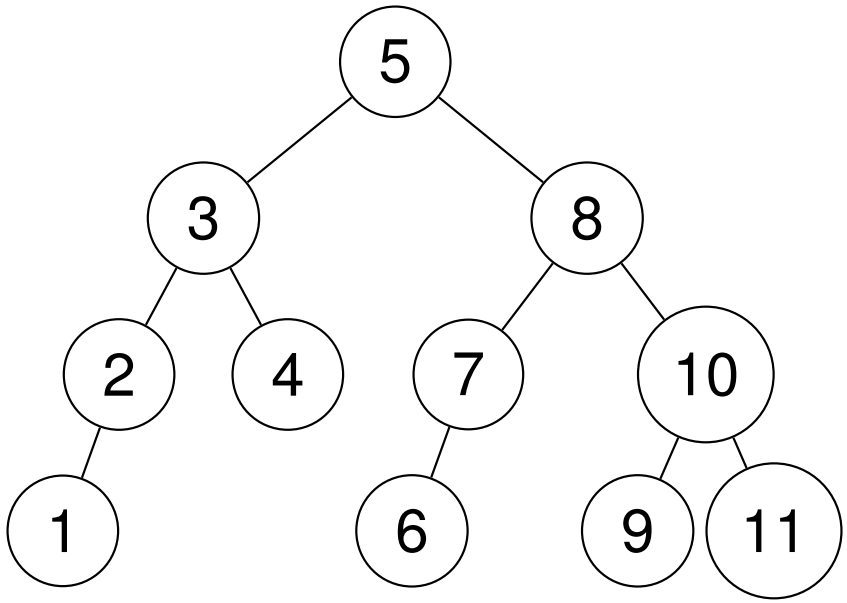
Rotación DI: $f_e = 2$; hijo derecho con $f_e = -1$



Borrado en árboles AVL

Ejercicio

Dado el siguiente árbol AVL, borra los siguientes elementos: 4, 8, 6, 5, 2, 1, 7. Criterio: M_i. Indica los tipos de rotaciones realizadas.

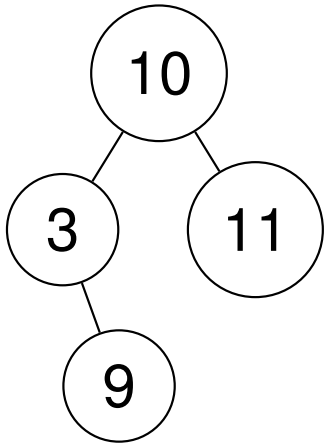


Borrado en árboles AVL

Ejercicio

Dado el siguiente árbol AVL, borra los siguientes elementos: 4, 8, 6, 5, 2, 1, 7. Criterio: M_i. Indica los tipos de rotaciones realizadas.

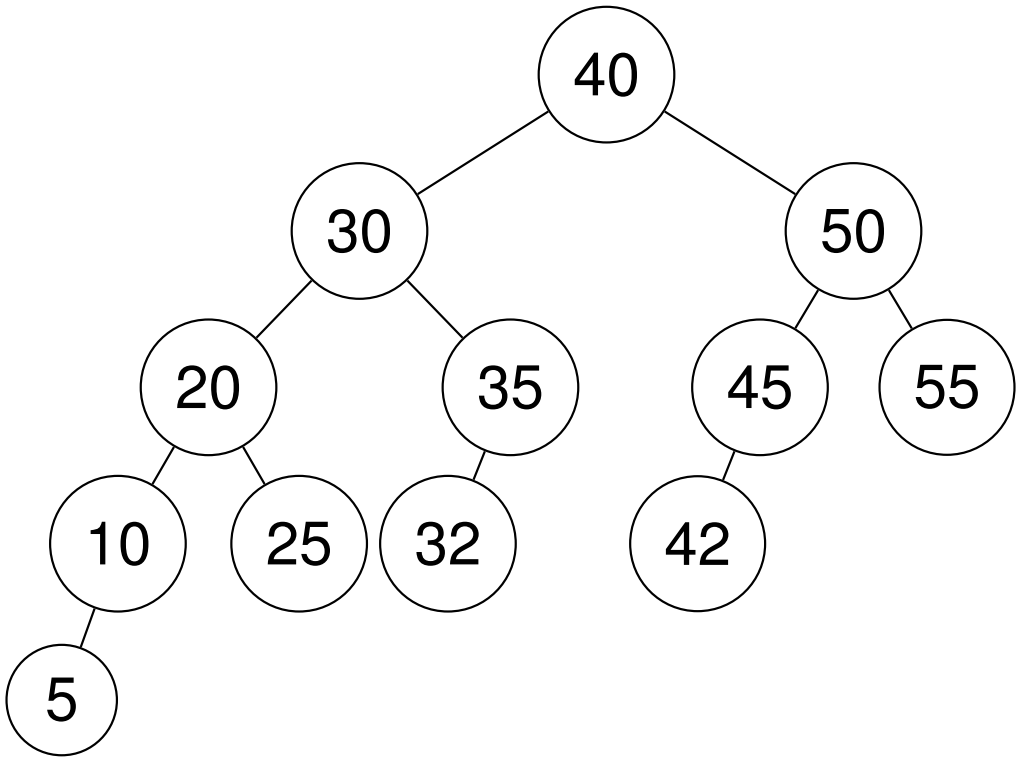
Solución: II, DD, DI, DD



Borrado en árboles AVL

Ejercicio

Dado el siguiente árbol AVL, borra los siguientes elementos: 55, 32, 40, 30. Criterio: M_i. Indica los tipos de rotaciones realizadas.

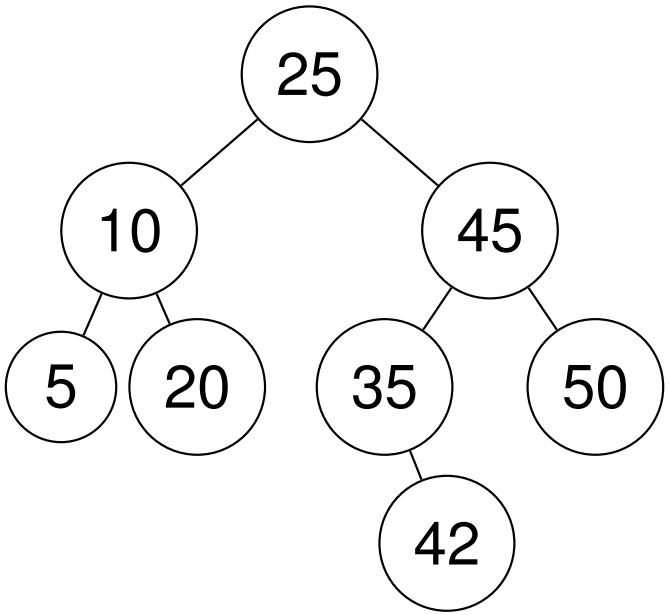


Borrado en árboles AVL

Ejercicio

Dado el siguiente árbol AVL, borra los siguientes elementos: 55, 32, 40, 30. Criterio: M_i. Indica los tipos de rotaciones realizadas.

Solución: II, II, DD, II



Borrado en árboles AVL

Pregunta

¿Cuántas rotaciones como máximo se producen tras un borrado?¿Cuál es la complejidad temporal asintótica de una borrado en un árbol AVL respecto al número de nodos n del árbol?

Borrado en árboles AVL

Pregunta

¿Cuántas rotaciones como máximo se producen tras un borrado?¿Cuál es la complejidad temporal asintótica de una borrado en un árbol AVL respecto al número de nodos n del árbol?

- Como máximo tantas rotaciones como nodos en el camino de búsqueda hasta el elemento a ser borrado
- $\Omega(1)$; $O(\log(n))$

Programación Avanzada y Estructuras de Datos

6. Conjuntos y mapas

Víctor M. Sánchez Cartagena

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

27 de noviembre de 2024

Índice

Ejemplo introductorio

Queremos contar el número de palabras *diferentes* que hay en un texto. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

```
ifstream f;
    f.open("texto.txt");
    if(f.is_open()){
            string s;
            f >> s;
            while(!f.eof()){
                cout << "Palabra: " << s << endl;
                //Añadir palabra a tipo de datos
                f>>s;
            }
            f.close();
            //Imprimir número de palabras diferentes
    }
```

Ejemplo introductorio

Queremos contar el número de palabras *diferentes* que hay en un texto. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

Ejemplo introductorio

Queremos contar el número de palabras *diferentes* que hay en un texto. Se puede iterar fácilmente sobre las palabras de un texto con el operador `>>`. ¿Qué tipos de datos podríamos utilizar?

- Vector (desordenado), agregando cada palabra al final, comprobando antes que no estaba
- Vector (ordenado), agregando cada palabra de manera que queden ordenadas, comprobando antes que no estaba
- Árbol AVL

Ejemplo introductorio

Solución con vector desordenado:

```
ifstream f;
f.open("texto.txt");
if(f.is_open()){
        vector<string> v;
        string s;
        f >> s;
        while(!f.eof()){
            cout << "Palabra: " << s << endl;
            //Añadir palabra a tipo de datos
            int found=false;
            for(int i=0; i<v.size();i++)
                if(v[i]==s){
                    found=true;
                    break;}
            if (! found)
                v.push_back(s);
            f>>s;
        }
        f.close();
        //Imprimir número de palabras diferentes
        cout << "Hay " << v.size() << " palabras diferentes" << endl;
}
```

Ejemplo introductorio

Pregunta

¿Cuál es la complejidad temporal de comprobar si un elemento no está e insertarlo si es necesario? ¿y de obtener el número de elementos final?

Operación	Caso mejor	Caso peor
vector desordenado: insertar		
vector desordenado: contar		
vector ordenado: insertar		
vector ordenado: contar		
árbol AVL: insertar		
árbol AVL: contar		

*Si se modifica ligeramente para almacenar en un atributo entero el número de elementos

Ejemplo introductorio

Pregunta

¿Cuál es la complejidad temporal de comprobar si un elemento no está e insertarlo si es necesario? ¿y de obtener el número de elementos final?

Operación	Caso mejor	Caso peor
vector desordenado: insertar	$\Omega(1)$	$O(n)$
vector desordenado: contar	$\Omega(1)$	$O(1)$
vector ordenado: insertar	$\Omega(1)$	$O(n)$
vector ordenado: contar	$\Omega(1)$	$O(1)$
árbol AVL: insertar	$\Omega(1)$	$O(\log(n))$
árbol AVL: contar	$\Omega(1)$	$O(1)$*

*Si se modifica ligeramente para almacenar en un atributo entero el número de elementos

Ejemplo introductorio

¿Por qué no hacemos que el tipo de datos (clase) contenga una operación para insertar un elemento únicamente si no está?

```
ifstream f;
f.open("texto.txt");
if(f.is_open()){
        set<string> st;
        string s;
        f >> s;
        while(!f.eof()){
            cout << "Palabra: " << s << endl;
            st.insert(s);
            f>>s;
        }
        f.close();
        cout << "Hay " << st.size() << " palabras distintas" << endl;
}
```

TAD conjunto

Especificación del TAD conjunto

Definición: Colección de n elementos almacenados sin un orden definido, tal que todos los elementos son distintos

TAD conjunto

Especificación del TAD conjunto

Operaciones:

- Obtiene el número de elementos almacenados en el conjunto

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

- Añade el elemento `e` al conjunto.

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

- Devuelve `true` si el elemento `e` se encuentra en el conjunto y `false` en caso contrario

```
bool find(const Elem &e) const;
```

- Elimina el elemento `e`. Devuelve `true` si el elemento estaba en el conjunto y `false` en caso contrario

```
bool erase(const Elem& e);
```

TAD conjunto

Especificación del TAD conjunto

Operaciones:

- Devuelve un iterador que apunta al primer elemento. Devuelve el mismo valor que `end()` si el conjunto está vacío

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

- Devuelve un iterador que apunta al elemento imaginario que se encuentra tras el último elemento del conjunto

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

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

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

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	Salida	Contenido del conjunto
`insert(7)`		
`insert(4)`		
`size()`		
`insert(4)`		
`size()`		
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	$\{7\}$
`insert(4)`		
`size()`		
`insert(4)`		
`size()`		
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`		
`insert(4)`		
`size()`		
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`		
`size()`		
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`		
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	Salida	Contenido del conjunto
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	Salida	Contenido del conjunto
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`	true	{7,4}
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`	true	{7,4}
`find(8)`	false	{7,4}
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`	true	{7,4}
`find(8)`	false	{7,4}
`erase(8)`	false	{7,4}
`erase(7)`		
`size()`		

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	Salida	Contenido del conjunto
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`	true	{7,4}
`find(8)`	false	{7,4}
`erase(8)`	false	{7,4}
`erase(7)`	true	{4}
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`	true	{7,4}
`find(8)`	false	{7,4}
`erase(8)`	false	{7,4}
`erase(7)`	true	{4}
`size()`	1	{4}

Especificación del TAD conjunto

Operaciones (conjunto *fusionable*):

- Reemplaza al conjunto actual (A) con la unión de A y B: $A \leftarrow A \cup B$

```
void union(const Conjunto& B);
```

- Reemplaza al conjunto actual (A) con la intersección de A y B: $A \leftarrow A \cap B$

```
void interseccion(const Conjunto& B);
```

Implementación del TAD conjunto

Determinación de la representación:

- Vector desordenado
- Vector ordenado
- Árbol AVL
- Vector de bits

Implementación del TAD conjunto

Determinación de la representación:

- Complejidad espacial (n: número de elementos almacenados en el conjunto; u: tamaño conjunto universal):
  - Vector desordenado:
  - Vector ordenado:
  - Árbol AVL:
  - Vector de bits:

Implementación del TAD conjunto

Determinación de la representación:

- Complejidad espacial (n: número de elementos almacenados en el conjunto; u: tamaño conjunto universal):
  - Vector desordenado: $O(n)$
  - Vector ordenado: $O(n)$
  - Árbol AVL: $O(n)$
  - Vector de bits: $O(u)$

Implementación del TAD conjunto

Determinación de la representación:

- Complejidad temporal (n: número de elementos almacenados en el conjunto; u: tamaño conjunto universal):

Operación	v. desordenado	v. ordenado	AVL	v. bits
`find`				
`insert`				
`union`				

Implementación del TAD conjunto

Determinación de la representación:

- Complejidad temporal (n: número de elementos almacenados en el conjunto; u: tamaño conjunto universal):

Operación	v. desordenado	v. ordenado	AVL	v. bits
`find`	$O(n)$	$O(\log(n))$	$O(\log(n))$	$O(1)$
`insert`	$O(n)$	$O(n)$	$O(\log(n))$	$O(1)$
`union`	$O(n^2)$	$O(n)$	$O(n \cdot \log(n))$	$O(u)$

Índice

Tablas hash

Pregunta

¿Podemos reducir la complejidad espacial del vector de bits sin renunciar a la complejidad temporal constante para búsqueda e inserción?

Tablas hash

Pregunta

¿Podemos reducir la complejidad espacial del vector de bits sin renunciar a la complejidad temporal constante para búsqueda e inserción?

Sí, empleando una **tabla hash**

Tablas hash

- Se divide el conjunto universal en N subconjuntos diferentes.
- Tabla hash: **vector** en el que a cada posición le corresponde un subconjunto del conjunto universal

Para poder mantener tiempos constantes de búsqueda e inserción necesitamos:

- Decidir qué subconjunto del conjunto universal corresponde a cada posición. **Función de hash**: objeto a almacenar $\rightarrow$ posición que le corresponde en la tabla hash
- Un "plan B" para cuando la posición que corresponde a un objeto está ocupada: **estrategia de redispersión**

Función de hash

- Si los elementos a almacenar en la tabla hash son enteros, simplemente se emplea la siguiente **función de compresión**, que garantiza un valor en $[0, N-1]$

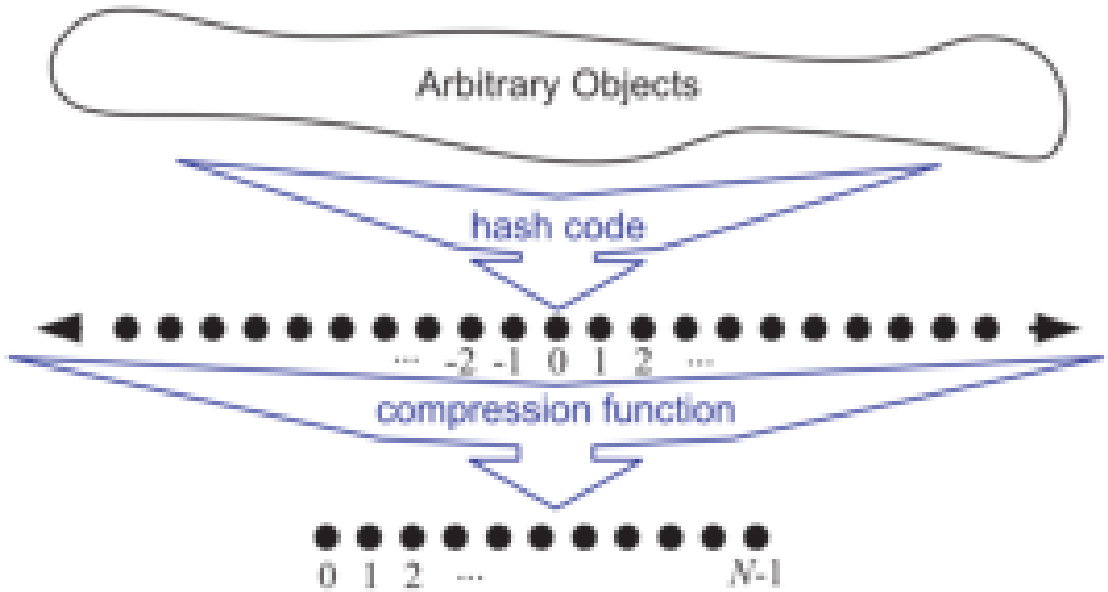
$$h(x) = |x| \text{ mód } N$$

  donde:
  - x: elemento a almacenar en la tabla
  - $h(x)$: posición que le corresponde
  - N: tamaño del vector (tabla hash)
- Ejemplos ($N = 11$):
  - $h(1) = 1 \text{ mód } 11 = 1$
  - $h(200) = 200 \text{ mód } 11 = 2$
  - $h(-30) = 30 \text{ mód } 11 = 8$
- Existen otras funciones de compresión más sofisticadas, como MAD (*multiply add and divide*)

Función de hash

- Si los elementos a almacenar no son enteros, es necesario convertirlos a enteros antes de aplicar la función de compresión
- El valor entero asociado a un objeto abitrario se denomina **código hash**

Función hash = código hash + función de compresión



fuente: Goodrich, M. T., Tamassia, R., & Mount, D. M. (2011). Data structures and algorithms in C++. John Wiley & Sons.

Función de hash

Características de una buena función de hash:

- Se puede calcular rápidamente
- Distribuye los elementos del conjunto universal de manera uniforme en las posiciones de la tabla hash

Pregunta

Queremos guardar cadenas de caracteres (`string`) en una tablas hash. Sabemos que nuestras cadenas contendrán palabras en español. Utilizamos el código ASCII del primer caracter de la cadena como código hash. ¿Es una buena función de hash?¿Se puede mejorar?

Función de hash

Características de una buena función de hash:

- Se puede calcular rápidamente
- Distribuye los elementos del conjunto universal de manera uniforme en las posiciones de la tabla hash

Pregunta

Queremos guardar cadenas de caracteres (`string`) en una tablas hash. Sabemos que nuestras cadenas contendrán palabras en español. Utilizamos el código ASCII del primer caracter de la cadena como código hash. ¿Es una buena función de hash?¿Se puede mejorar?

No. La frecuencia de la primera letra de las palabras no es uniforme: pocas palabras empiezan por x o y, por ejemplo. Habría que usar información sobre todos los caracteres de la cadena

Estrategia de redispersión

- **Colisión**: al insertar x, la casilla $h(x)$ está ocupada
- **Estrategia de redispersión**: ¿qué hago cuando se produce una colisión?

Dos familias de estrategias:

- Hash cerrado (*Open Addressing*)
- Hash abierto (*Separate Chaining*)

Estrategia de redispersión: hash cerrado

- Cuando la posición $h(x)$ está ocupada (colisión), se reintenta en las posiciones $h_1(x)$, $h_2(x)$, ..., $h_{N-1}(X)$
- $h_i(x) \rightarrow$ función de redispersión para el reintento número i
- Hay $N - 1$ funciones de redispersión diferentes
- Si tras probar las $N - 1$ funciones de redispersión, no es posible insertar en ninguna, la tabla está llena
- Una buena función de redispersión es imprescincible para obtener una complejidad promedio $\Theta(1)$ en búsquedas

Estrategia de redispersión: hash cerrado

Estrategia de redispersión lineal

- La estrategia más simple
- Se intenta insertar en la siguiente posición de la tabla
- Al llegar al final se vuelve al principio
- $h_i(x) = h_{i-1}(x) + 1 \mod N$ $(h_0(x) = h(x))$
- $h_i(x) = h(x) + i \mod N$

Estrategia de redispersión: hash cerrado

Ejemplo

Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión lineal (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejemplo

Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión lineal (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

14	18	23	9	30	12	6
0	1	2	3	4	5	6

14 intentos en total

Estrategia de redispersión: hash cerrado

Pregunta

¿Cuańtos intentos son necesarios para insertar el valor 25 en la tabla anterior?¿Cuál es la complejidad de la inserción en el peor caso?

14	18	23	9	30	12	6
0	1	2	3	4	5	6

Estrategia de redispersión: hash cerrado

Pregunta

¿Cuańtos intentos son necesarios para insertar el valor 25 en la tabla anterior?¿Cuál es la complejidad de la inserción en el peor caso?

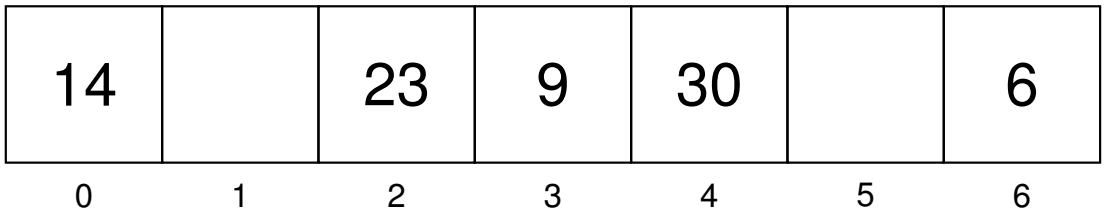
14	18	23	9	30	12	6
0	1	2	3	4	5	6

7 intentos $\rightarrow O(n)$ ($n =$ número de elementos del conjunto)

Estrategia de redispersión: hash cerrado

Pregunta

¿Qué pasa si intentamos buscar el 30 en esta tabla? ¿Cómo sería el algoritmo de búsqueda en una tabla hash con dispersión cerrada?¿Y su complejidad en el peor caso?



Estrategia de redispersión: hash cerrado

Pregunta

¿Qué pasa si intentamos buscar el 30 en esta tabla? ¿Cómo sería el algoritmo de búsqueda en una tabla hash con dispersión cerrada?¿Y su complejidad en el peor caso?

14		23	9	30		6
0	1	2	3	4	5	6

- $h(30) = 2$; en la casilla 2 está el 23; pero el 30 está sí en la tabla

Estrategia de redispersión: hash cerrado

Pregunta

¿Qué pasa si intentamos buscar el 30 en esta tabla? ¿Cómo sería el algoritmo de búsqueda en una tabla hash con dispersión cerrada?¿Y su complejidad en el peor caso?

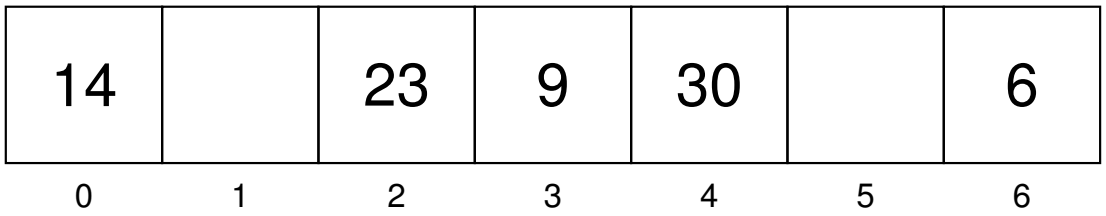
14		23	9	30		6
0	1	2	3	4	5	6

- $h(30) = 2$; en la casilla 2 está el 23; pero el 30 está sí en la tabla
- Hay que buscar en las casillas $h_i(x)$ hasta encontrar el elemento o llegar a una casilla vacía $\rightarrow O(n)$

Estrategia de redispersión: hash cerrado

Pregunta

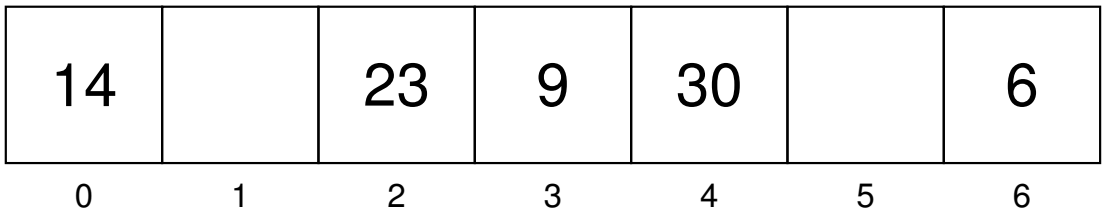
¿Qué pasa si borramos el 9 y luego intentamos buscar el 30? ¿Cómo solucionarías el problema?



Estrategia de redispersión: hash cerrado

Pregunta

¿Qué pasa si borramos el 9 y luego intentamos buscar el 30? ¿Cómo solucionarías el problema?



- La búsqueda del 30 se detendría en la casilla 3

Estrategia de redispersión: hash cerrado

Pregunta

¿Qué pasa si borramos el 9 y luego intentamos buscar el 30? ¿Cómo solucionarías el problema?

14		23	9	30		6
0	1	2	3	4	5	6

- La búsqueda del 30 se detendría en la casilla 3
- Los borrados se marcan con un valor especial: la casilla cuenta como vacía para la inserción pero como llena para la búsqueda:

14		23	X	30		6
0	1	2	3	4	5	6

Estrategia de redispersión: hash cerrado

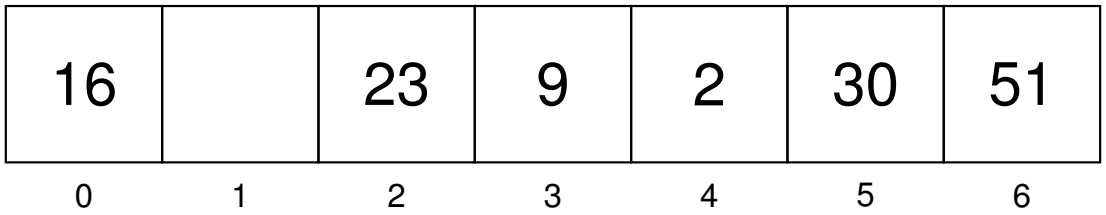
Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión lineal (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión lineal (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción



1, 2, 3, 4, 5, y 6 intentos respectivamente. Total: 21

Estrategia de redispersión: hash cerrado

- x e y son elementos **sinónimos** si y sólo si $h(x) = h(y)$
- La estrategia de redispersión lineal hace que el número de intentos se incremente drásticamente cuando hay muchos elementos sinónimos
- Este fenómeno se llama **amontonamiento**

Estrategia de redispersión: hash cerrado

Estrategia de redispersión aleatoria

- El siguiente intento se realiza c casillas más adelante (en lugar de una)
- c y N no deben tener factores primos comunes mayores que 1
- $h_i(x) = h_{i-1}(x) + c \mod N \ (h_0(x) = h(x))$
- $h_i(x) = h(x) + c \cdot i \mod N$
- Continúa habiendo amontonamiento

Estrategia de redispersión: hash cerrado

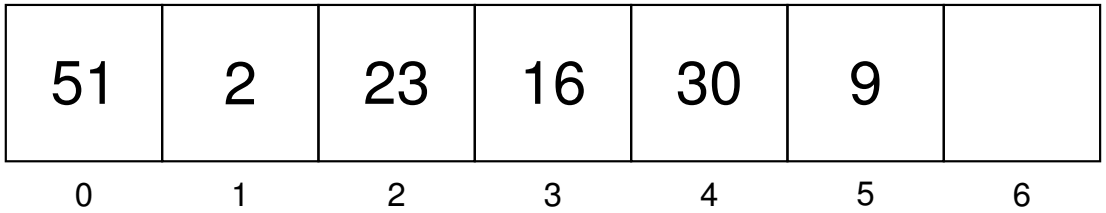
Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión aleatoria con $c = 3$ (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión aleatoria con $c = 3$ (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción



1, 2, 3, 4, 5, y 6 intentos respectivamente. Total: 21

Estrategia de redispersión: hash cerrado

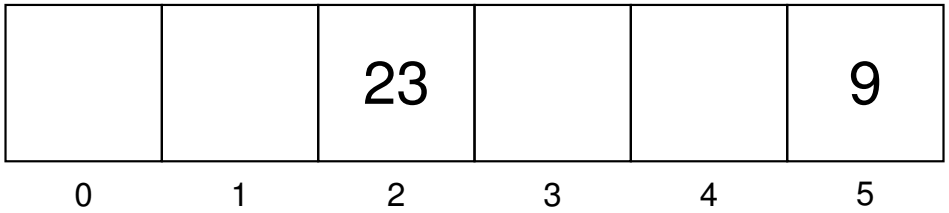
Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 6$ y estrategia de redispersión aleatoria con $c = 3$ (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 6$ y estrategia de redispersión aleatoria con $c = 3$ (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción



Sólo se pueden insertar 23 y 9. 1 y 2 intentos respectivamente

Estrategia de redispersión: hash cerrado

Estrategia de redispersión con segunda función hash

- El siguiente intento se realiza un número variable $k(x)$ de casillas más adelante
- $k(x)$ depende del elemento a ser insertado
- $k(x) = (x \mod N - 1) + 1$
- N debe ser primo para evitar que $k(x)$ tenga factores primos en común con N
- $h_i(x) = h_{i-1}(x) + k(x) \mod N$ $(h_0(x) = h(x))$
- $h_i(x) = h(x) + k(x) \cdot i \mod N$
- Evita el amontonamiento

Estrategia de redispersión: hash cerrado

Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión con segunda función hash (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión con segunda función hash (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

51	16	23	30		2	9
0	1	2	3	4	5	6

1, 2, 2, 2, 4, y 5 intentos respectivamente. Total: 16

Estrategia de redispersión: hash cerrado

Ejercicio

Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión segunda función hash (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejercicio

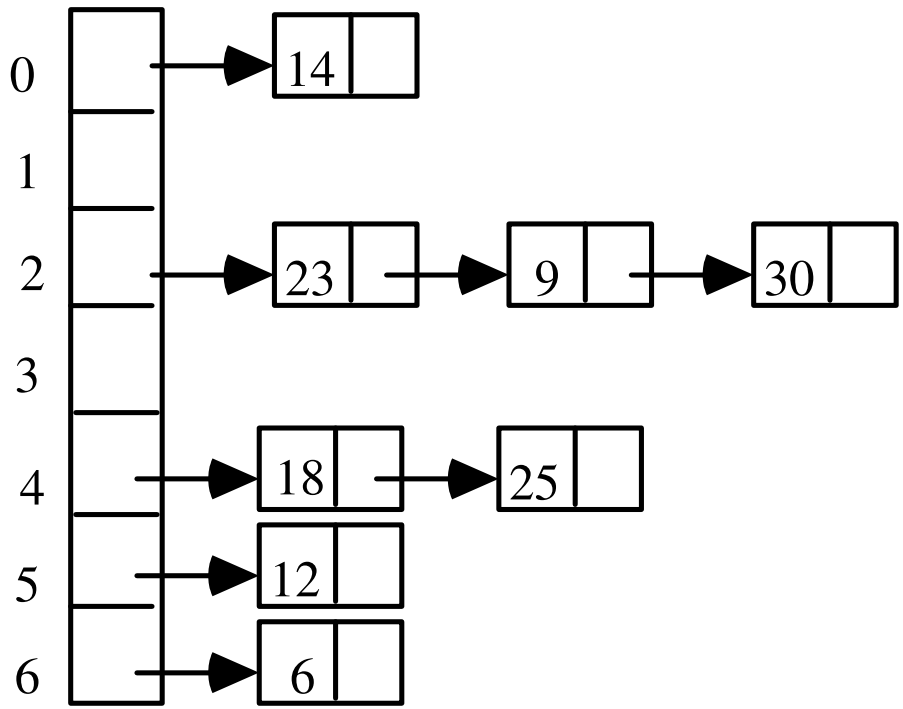
Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión segunda función hash (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

14	6	23	30	18	12	9
0	1	2	3	4	5	6

11 intentos en total

Estrategia de redispersión: hash abierto

- Las colisiones se resuelven mediante una lista enlazada (se inserta al final)
- Elimina las colisiones entre elementos no sinónimos
- El número de intentos se calcula como la longitud de la lista enlazada + 1



Estrategia de redispersión: hash abierto

Ejercicio

Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión hash abierto. Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash abierto

Ejercicio

Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión hash abierto. Cuenta los intentos (número de accesos a la tabla) para cada inserción

(el resultado es la tabla de la diapositiva anterior, sin el elemento 25).
Total: 10 intentos

Factor de carga

Sea n el número de elementos almacenados en la tabla hash, y N el tamaño de la tabla:

Factor de carga: $\lambda = \frac{n}{N}$

- Hash cerrado: $\lambda \in [0, 1]$
- Hash abierto: $\lambda \in [0, \infty)$

Complejidades promedio

- Las complejidades en el peor caso para insertar y buscar en una tabla hash están en $O(n)$
- Para factores de carga bajos y buenas funciones de hash, el coste promedio de ambas operaciones está en $\Theta(1)$
  - Hash cerrado: $\lambda < 0.5$
  - Hash abierto: $\lambda < 0.9$
- Cuando el factor de carga sobrepasa el límite, hay que crear una nueva tabla más grande (normalmente el doble): *rehashing*
- El coste amortizado promedio de insertar n elementos (incluyendo la creación de tantas nuevas tablas como sea necesario) está en $\Theta(n)$
- Las tablas hash suelen ser la implementación de referencia para conjuntos, salvo que se desee listar sus elementos en orden

Ejercicio adicional

Ejercicio

Inserta en una tabla hash ($N = 11$) los siguientes elementos: 23, 14, 10, 15, 3, 5, 7, 8, 36, 47, 4. Cuenta los intentos totales. Emplea las siguientes estrategies de redispersión:

- Hash cerrado, redispersión lineal (37 intentos)
- Hash cerrado, redispersión aleatoria con $c = 4$ (34 intentos)
- Hash cerrado, segunda función de hash (22 intentos)
- Hash abierto (18 intentos)

Índice

Conjuntos en C++ STL

Hay disponibles dos implementaciones del TAD conjunto en la biblioteca C++ STL:

- `set`: implementado como un árbol rojo-negro (variante del AVL). Al listar sus elementos, éstos aparecen en orden según su `operator<`
- `unordered_set`: implementado como una tabla hash con redispersión **abierta** (**hay que definir funcion hash para tipos de datos propios**) Al listar sus elementos, éstos no aparecen en ningún orden particular.

Características:

- Ambas soportan las mismas operaciones principales
- Son implementaciones de conjunto *no fusionables*
- Hay unas pocas operaciones dependientes de la implementación: por ejemplo, gestionar factor de carga y *rehashing*
- Más información: `https://cplusplus.com/reference/stl/`

Conjuntos en C++ STL: `set`

```cpp
#include<set>
using namespace std;

//...

//Es necesario que el tipo tenga un operator<
set<int> conjunto;

//conjunto vacío: 0 elementos
cout << conjunto.size() << endl;

//Inserción: reequilibrado automático
conjunto.insert(10);
conjunto.insert(20);
conjunto.insert(30);
conjunto.insert(40);
conjunto.insert(50);
conjunto.insert(20);

//5 elementos
cout << conjunto.size() << endl;
```

Conjuntos en C++ STL: `set`

- Búsqueda: `find` devuelve un iterador

```
if(conjunto.find(40)!=conjunto.end())
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

- Búsqueda: `count` devuelve 1 o 0

```
if(conjunto.count(40)==1)
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

- Borrado: `erase` devuelve el número de elementos borrados

```
cout <<conjunto.erase(40) << endl; //1
cout <<conjunto.erase(35) << endl; //0
```

Conjuntos en C++ STL: `set`

- Iteración en orden ascendente

```
//Imprime: 10 20 30 50
for (set<int>::iterator it=conjunto.begin();
                                    it!=conjunto.end(); ++it)
     cout << ' ' << *it;
cout << endl;
```

- Iteración en orden descendente

```
//Imprime: 50 30 20 10
for (set<int>::reverse_iterator it=conjunto.rbegin();
                                    it!=conjunto.rend(); ++it)
     cout << ' ' << *it;
cout << endl;
```

Conjuntos en C++ STL: `set`

- El tipo de datos `auto` permite simplificar mucho el código:

```
//Imprime: 10 20 30 50
for (auto it=conjunto.begin();
                                  it!=conjunto.end(); ++it)
     cout << ' ' << *it;
cout << endl;

//Imprime: 50 30 20 10
for (auto it=conjunto.rbegin();
                                  it!=conjunto.rend(); ++it)
     cout << ' ' << *it;
cout << endl;
```

Conjuntos en C++ STL: `set`

Pregunta

¿Cómo implementarías un método que devuelve el mayor elemento almacenado en un conjunto? ¿Y el menor?

Conjuntos en C++ STL: `unordered_set`

```cpp
#include<unordered_set>
using namespace std;

//...

//Es necesario que C++ sepa calcular la función hash del tipo
unordered_set<int> conjunto;

//conjunto vacío: 0 elementos
cout << conjunto.size() << endl;

//Inserción
conjunto.insert(10);
conjunto.insert(20);
conjunto.insert(30);
conjunto.insert(40);
conjunto.insert(50);
conjunto.insert(20);

//5 elementos
cout << conjunto.size() << endl;
```

Conjuntos en C++ STL: `unordered_set`

- Búsqueda: `find` devuelve un iterador

```
if(conjunto.find(40)!=conjunto.end())
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

- Búsqueda: `count` devuelve 1 o 0

```
if(conjunto.count(40)==1)
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

- Borrado: `erase` devuelve el número de elementos borrados

```
cout <<conjunto.erase(40) << endl; //1
cout <<conjunto.erase(35) << endl; //0
```

Conjuntos en C++ STL: `unordered_set`

- Iteración en orden arbitrario (depende de hash)

```
//Imprime: 50 30 20 10
for (unordered_set<int>::iterator it=conjunto.begin();
                        it!=conjunto.end(); ++it)
     cout << ' ' << *it;
cout << endl;
```

- No hay iteración en orden ascendente o descendente
- Se puede simplificar el código con `auto`

```
//Imprime: 50 30 20 10
for (auto it=conjunto.begin();
                        it!=conjunto.end(); ++it)
     cout << ' ' << *it;
cout << endl;
```

Conjuntos en C++ STL: `unordered_set`

- Tenemos acceso a las características de la tabla hash
  - Tamaño de la tabla: `bucket_count()`
  - Factor de carga: `load_factor()`
  - Posición donde está almacenado un elemento: `bucket(valor)`

```
cout << "Tamaño:" << conjunto.bucket_count() << endl;
cout << "Factor de carga:" << conjunto.load_factor() << endl;

//Imprime: 50(11) 30(4) 20(7) 10(10)
for (unordered_set<int>::iterator it=conjunto.begin();
                        it!=conjunto.end(); ++it)
     cout << ' ' << *it << '(' <<  conjunto.bucket(*it)  << ')';
cout << endl;
```

Conjuntos en C++ STL

Pregunta

Vuelve a implementar el programa que calcula el número de palabras distintas en un texto empleando conjuntos de la biblioteca STL de C++. ¿Qué sería más eficiente: `set` o `unordered_set`?

Índice

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

```
ifstream f;
    f.open("texto.txt");
    if(f.is_open()){
            string s;
            f >> s;
            while(!f.eof()){
                cout << "Palabra: " << s << endl;
                //Añadir palabra a tipo de datos e incrementar frecuencia
                f>>s;
            }
            f.close();
            //Imprimir número de palabras diferentes con su frecuencia
    }
```

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

Solución ineficiente: vector de palabras y vector de frecuencias

Ejemplo introductorio

Solución con dos vectores:

```
ifstream f;
f.open("texto.txt");
if(f.is_open()){
        vector<string> palabras;
        vector<int> frecuencias;
        string s;
        f >> s;
        while(!f.eof()){
            //Añadir palabra a tipo de datos e incrementar frecuencia
            int posicion=-1;
            for(int i=0; i<palabras.size();i++)
                if(palabras[i]==s){
                    posicion=i;
                    break;}
            if (posicion==-1){
                palabras.push_back(s); frecuencias.push_back(0);
                posicion=frecuencias.size()-1; }
            frecuencias[posicion]+=1;
            f>>s;
        }
        f.close();
}
```

Ejemplo introductorio

Solución con dos vectores:

```
 //.....
 f.close();

//Imprimir palabras diferentes:
cout << "Hay " << palabras.size() << " diferentes" << endl;

//Imprimir la frecuencia de cada una:
for(int i=0; i<palabras.size(); i++)
     cout << palabras[i] << ": " << frecuencias[i] << endl;
```

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

- Solución ineficiente: vector de palabras y vector de frecuencias $\rightarrow O(n)$ por cada inserción, siendo n el número de palabras diferentes

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

- Solución ineficiente: vector de palabras y vector de frecuencias $\rightarrow O(n)$ por cada inserción, siendo n el número de palabras diferentes
- Solución eficiente: ¿podemos asociar un valor (la frecuencia) a cada elemento de un conjunto? El tipo abstracto de datos resultante se llama **mapa**

TAD mapa

Especificación del TAD mapa

Definición: Colección de n pares clave-valor, de manera que cada uno de esos pares tiene una clave diferente. No hay un orden particular entre los pares.

Especificación del TAD mapa

Operaciones:

- Obtiene el número de pares clave-valor en el mapa

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

- Añade el par *c*-*v* si el mapa no tenía ningún par con clave *c*. En caso contrario, sustituye el valor del par existente por *v*

```
void put(const Clave &c, const Valor &v);
```

- Devuelve un iterador que apunta al par clave-valor con clave *c*. Si la clave no existe, devuelve un iterador que apunta a `end()`

```
IteradorMapa find(const Clave &c) const;
```

TAD mapa

Especificación del TAD mapa

- Elimina el par con clave `c`. Devuelve `true` si un par con clave `c` existía en el mapa y `false` en caso contrario

```
bool erase(const Clave &c);
```

- Devuelve un iterador que apunta al primer par. Devuelve el mismo valor que `end()` si el conjunto está vacío

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

- Devuelve un iterador que apunta al par imaginario que se encuentra tras el último par del mapa

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

Especificación del TAD mapa

- Devuelve la clave del par aputado por el iterador `it`

```
Clave get(IteradorMapa it) const;
```

- Devuelve el valor del par aputado por el iterador `it`

```
Valor get(IteradorMapa it) const;
```

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`		
`put(7,B)`		
`put(2,C)`		
`put(2,E)`		
`size()`		
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`		
`put(2,C)`		
`put(2,E)`		
`size()`		
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`		
`put(2,E)`		
`size()`		
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`		
`size()`		
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`		
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`	end	{(5,A), (7,B), (2,E)}
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`	end	{(5,A), (7,B), (2,E)}
`find(2)`	p:(2,E)	{(5,A), (7,B), (2,E)}
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`	end	{(5,A), (7,B), (2,E)}
`find(2)`	p:(2,E)	{(5,A), (7,B), (2,E)}
`erase(5)`	true	{(7,B), (2,E)}
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`	end	{(5,A), (7,B), (2,E)}
`find(2)`	p:(2,E)	{(5,A), (7,B), (2,E)}
`erase(5)`	true	{(7,B), (2,E)}
`erase(6)`	false	{(7,B), (2,E)}
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`	end	{(5,A), (7,B), (2,E)}
`find(2)`	p:(2,E)	{(5,A), (7,B), (2,E)}
`erase(5)`	true	{(7,B), (2,E)}
`erase(6)`	false	{(7,B), (2,E)}
`find(5)`	end	{(7,B), (2,E)}

Implementación del TAD mapa

Determinación de la representación:

- Árbol AVL (o rojo-negro): cada nodo contiene la clave y puntero al valor. Los nodos se comparan por clave
- Tablas hash: cada posición contiene la clave y puntero al valor. La función hash se calcula sobre la clave

Índice

Mapas en C++ STL

Hay disponibles dos implementaciones del TAD mapa en la biblioteca C++ STL:

- `map`: implementado como un árbol rojo-negro (variante del AVL). Al listar sus pares, éstos aparecen en orden según el `operator<` de su clave
- `unordered_map`: implementado como una tabla hash con redispersión **abierta** (**hay que definir funcion hash para claves de tipos de datos propios**). Al listar sus pares, éstos no aparecen en ningún orden particular

Características:

- Ambas soportan las mismas operaciones principales
- Hay unas pocas operacions dependientes de la implementación: por ejemplo, gestionar factor de carga y *rehashing*
- Más información: `https://cplusplus.com/reference/stl/`

Mapas en C++ STL: `map`

```cpp
#include<map>
using namespace std;

//...

//Es necesario que el tipo de la clave tenga un operator<
map<int,string> mapa;

//mapa vacío: 0 elementos
cout << mapa.size() << endl;

//Inserción: reequilibrado automático
mapa.insert(pair<int,string>(10,"diez"));
mapa.insert(pair<int,string>(20,"viente"));
mapa.insert(pair<int,string>(30,"treinta"));
mapa.insert(pair<int,string>(40,"cuarenta"));
mapa.insert(pair<int,string>(50,"cincuenta"));
//No sobreescribe
mapa.insert(pair<int,string>(20,"veinte"));

//5 elementos
cout << mapa.size() << endl;
```

Mapas en C++ STL: `map`

- Búsqueda: `find` devuelve un iterador

```
if(mapa.find(40)!=mapa.end())
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

- Se puede emplear el iterador para acceder al valor o modificarlo

```
auto it=mapa.find(40);
if(it!=mapa.end()){
      cout << "encontrado: "<< it->second << endl;
      it->second="Cuarenta";}
else
      cout << "no encontrado" << endl;
```

- Búsqueda: `count` devuelve 1 o 0

```
if(mapa.count(40)==1)
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

Mapas en C++ STL: `map`

- El operador `[]` permite accesos para lectura y escritura. Si la clave no existe, crea una con un valor obtenido mediante el constructor por defecto del tipo valor

```
cout <<mapa[40] << endl; //Cuarenta
mapa[40]="cuarenta";
mapa[20]="veinte";
cout <<mapa[15] << endl; //Cadena vacía
```

- Borrado: `erase` devuelve el número de elementos borrados

```
cout <<mapa.erase(40) << endl; //1
cout <<mapa.erase(35) << endl; //0
```

Mapas en C++ STL: `map`

- Iteración en orden ascendente (permite modificaciones de valores)

```
//Imprime: 10->diez 15-> 20->veinte 30->treinta 50->cincuenta
for (auto it=mapa.begin();
                                    it!=mapa.end(); ++it){
     cout << ' ' << it->first << "->" << it->second;
     it->second=it->second+";";
}
cout << endl;
```

- Iteración en orden descendente (permite modificaciones de valores)

```
//Imprime: 50->cincuenta; 30->treinta; 20->veinte; 15->; 10->diez;
for (auto it=mapa.rbegin();
                                    it!=mapa.rend(); ++it){
     cout << ' ' << it->first << "->" << it->second;
}
cout << endl;
```

Mapas en C++ STL: `map`

- Como los iteradores permiten modificar el mapa, cuando el mapa sobre el que queremos iterar no se puede modificar (es constante), debemos utilizar **iteradores constantes**:

```
//Imprime: 10->diez; 15->; 20->veinte; 30->treinta; 50->cincuenta;
for (auto it=mapa.cbegin();
                                    it!=mapa.cend(); ++it)
     cout << ' ' << it->first << "->" << it->second;
cout << endl;

//Imprime: 50->cincuenta; 30->treinta; 20->veinte; 15->; 10->diez;
for (auto it=mapa.crbegin();
                                    it!=mapa.crend(); ++it){
     cout << ' ' << it->first << "->" << it->second;
}
cout << endl;
```

Mapas en C++ STL: unordered_map

```cpp
#include<unordered_map>
using namespace std;

//...

//Es necesario que C++ sepa calcular la función hash del tipo de la clave
unordered_map<int,string> mapa;

//mapa vacío: 0 elementos
cout << mapa.size() << endl;

//Inserción
mapa.insert(pair<int,string>(10,"diez"));
mapa.insert(pair<int,string>(20,"viente"));
mapa.insert(pair<int,string>(30,"treinta"));
mapa.insert(pair<int,string>(40,"cuarenta"));
mapa.insert(pair<int,string>(50,"cincuenta"));
//No sobreescribe
mapa.insert(pair<int,string>(20,"veinte"));

//5 elementos
cout << mapa.size() << endl;
```

Mapas en C++ STL: `unordered_map`

- Búsqueda: `find` devuelve un iterador

```
if(mapa.find(40)!=mapa.end())
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

- Se puede emplear el iterador para acceder al valor o modificarlo

```
auto it=mapa.find(40);
if(it!=mapa.end()){
      cout << "encontrado: "<< it->second << endl;
      it->second="Cuarenta";}
else
      cout << "no encontrado" << endl;
```

- Búsqueda: `count` devuelve 1 o 0

```
if(mapa.count(40)==1)
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

Mapas en C++ STL: `unordered_map`

- El operador `[]` permite accesos para lectura y escritura. Si la clave no existe, crea una con un valor obtenido mediante el constructor por defecto del tipo valor

```
cout <<mapa[40] << endl; //Cuarenta
mapa[40]="cuarenta";
mapa[20]="veinte";
cout <<mapa[15] << endl; //Cadena vacía
```

- Borrado: `erase` devuelve el número de elementos borrados

```
cout <<mapa.erase(40) << endl; //1
cout <<mapa.erase(35) << endl; //0
```

Mapas en C++ STL: unordered_map

- Iteración en orden arbitrario (permite modificaciones de valores)

```
//Imprime: 15-> 50->cincuenta 30->treinta 20->veinte 10->diez
for (auto it=mapa.begin();
                                    it!=mapa.end(); ++it){
     cout << ' ' << it->first << "->" << it->second;
     it->second=it->second+";";
}
cout << endl;
```

- Como los iteradores permiten modificar el mapa, cuando el mapa sobre el que queremos iterar no se puede modificar (es constante), debemos utilizar **iteradores constantes**:

```
//Imprime: 15->; 50->cincuenta; 30->treinta; 20->veinte; 10->diez;
for (auto it=mapa.cbegin();
                                   it!=mapa.cend(); ++it)
     cout << ' ' << it->first << "->" << it->second;
cout << endl;
```

Mapas en C++ STL: `unordered_map`

- Tenemos acceso a las características de la tabla hash
  - Tamaño de la tabla: `bucket_count()`
  - Factor de carga: `load_factor()`
  - Posición donde está almacenado un elemento: `bucket(clave)`

```
cout << "Tamaño:" << mapa.bucket_count() << endl;
cout << "Factor de carga:" << mapa.load_factor() << endl;

//Imprime:
//15->;(2) 50->cincuenta;(11) 30->treinta;(4)
// 20->veinte;(7) 10->diez;(10)
for (auto it=mapa.begin();
                        it!=mapa.end(); ++it)
     cout << ' ' << it->first << "->" << it->second << '(' <<  mapa.b
cout << endl;
```

Mapas en C++ STL

Ejercicio

Vuelve a implementar el programa que calcula el número de palabras distintas en un texto y sus frecuencias empleando mapas de la biblioteca STL de C++. ¿Qué sería más eficiente: `map` o `unordered_map`?

Ejercicio

Imagina que tenemos una serie de documentos de texto y queremos implementar un buscador que, dada una palabra, nos diga rápidamente en qué documento(s) se encuentra. ¿Cómo lo implementarías?

Programación Avanzada y Estructuras de Datos

7. Cola de prioridad

Víctor M. Sánchez Cartagena

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

4 de diciembre de 2024

Índice

Ejemplo introductorio

Imagina que queremos gestionar la cola virtual de un sitio web de venta de entradas. Cada petición se codifica en un objeto `Peticion` que guarda el nombre de usuario, la dirección IP, el tipo de usuario (no registrado, registrado, premium, etc.), el gasto total que ha efectuado en el sitio web de venta de entradas, y la hora a la que ha accedido al sitio web. Todos estos criterios deben tenerse en cuenta para ordenar las peticiones en la cola virtual. La clase `Peticion` dispone de un `operator<` que nos indica si la primera petición tiene menos prioridad que la segunda.

Ejemplo introductorio

- Cada vez que un nuevo usuario accede la web de venta de entradas, debe agregarse su petición a la cola virtual
- Cada vez que el servidor cuenta con capacidad de cómputo suficiente, debe sacar de la cola virtual la petición con más prioridad y darle acceso al servicio de venta

Ejemplo introductorio

```
class Peticion{
private:
string nombre_usuario;
int IP;
/*Tipo usuario: 0: sin registrar; 1: registrado; 2: premium*/
int tipo_usuario;
float gasto_total;
time_t hora_acceso;
public:
Peticion();
Peticion(const string &, int, int, float);
Peticion(const Peticion &);
Peticion& operator=(const Peticion&);
~Peticion();
bool operator<(const Peticion& ) const;
//Getters y setters
string getNombreUsuario() const;
//....
};
```

Ejemplo introductorio

Pregunta

¿En qué estructura de datos almacenarías las peticiones para minimizar el coste temporal de: insertar elementos; saber cuál es el elemento con máxima prioridad; sacar de la cola el elemento con máxima prioridad?

Ejemplo introductorio

Pregunta

¿En qué estructura de datos almacenarías las peticiones para minimizar el coste temporal de: insertar elementos; saber cuál es el elemento con máxima prioridad; sacar de la cola el elemento con máxima prioridad?

- Lista enlazada ordenada descendentemente según `operator<`
  - Insertar: $\Omega(1)$; $O(n)$
  - Obtener elemento con máxima prioridad: $O(1)$
  - Borrar elemento con máxima prioridad: $O(1)$
- Árbol AVL ordenado según `operator<`
  - Insertar: $\Theta(\log(n))$
  - Obtener elemento con máxima prioridad: $\Theta(\log(n))$
  - Borrar elemento con máxima prioridad: $\Theta(\log(n))$

Ejemplo introductorio

```
//Añadir una petición a la cola
void add_to_queue(list<Peticion> & q, const Peticion & p){
    bool inserted=false;
    for(auto it= q.begin(); it != q.end() ;++it){
        if(*it < p ){
            //Insert before it
            q.insert(it,p);
            inserted=true;
            break;
        }
    }
    if(! inserted){
        q.push_back(p);
    }
}

list<Peticion> priority_queue;
//Obtener petición con máxima prioridad
cout << priority_queue.front().getNombreUsuario() << endl;
//Sacarla de la cola
priority_queue.pop_front();
```

TAD cola de prioridad

Especificación del TAD cola de prioridad

Definición: Colección de n elementos almacenados sin un orden definido. Cada elemento tiene asociada una prioridad. Sólo se puede elimininar y acceder al elemento con máxima prioridad

TAD cola de prioridad

Especificación del TAD cola de prioridad

Operaciones:

- Obtiene el número de elementos almacenados en la cola

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

- Añade el elemento `e` a la cola

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

- Devuelve el elemento con máxima prioridad

```
Elem max() const;
```

- Elimina el elemento con máxima prioridad. Devuelve `false` si la cola estaba vacía y `true` en caso contrario

```
bool removeMax();
```

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`		
`insert(9)`		
`insert(2)`		
`max()`		
`removeMax()`		
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	Salida	Contenido de la cola
`insert(5)`	-	{5}
`insert(9)`		
`insert(2)`		
`max()`		
`removeMax()`		
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`		
`max()`		
`removeMax()`		
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`		
`removeMax()`		
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`		
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	Salida	Contenido de la cola
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`	5	{5,2}
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`	5	{5,2}
`removeMax()`	true	{2}
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`	5	{5,2}
`removeMax()`	true	{2}
`removeMax()`	true	{}
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`	5	{5,2}
`removeMax()`	true	{2}
`removeMax()`	true	{}
`size()`	0	{}
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`	5	{5,2}
`removeMax()`	true	{2}
`removeMax()`	true	{}
`size()`	0	{}
`removeMax()`	false	{}

Implementación del TAD cola de prioridad

- Lista ordenada: `insert`: $O(n)$, `max`: $O(1)$, `removeMax`: $O(1)$
- Árbol AVL: `insert`: $O(\log n)$, `max`: $O(\log n)$, `removeMax`: $O(\log n)$

Implementación del TAD cola de prioridad

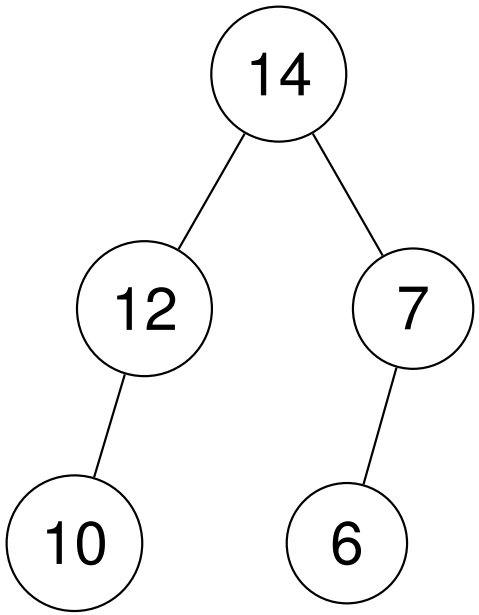
- Lista ordenada: `insert`: $O(n)$, `max`: $O(1)$, `removeMax`: $O(1)$
- Árbol AVL: `insert`: $O(\log n)$, `max`: $O(\log n)$, `removeMax`: $O(\log n)$
- Montículo o *heap*: `insert`: $O(\log n)$, `max`: $O(1)$, `removeMax`: $O(\log n)$

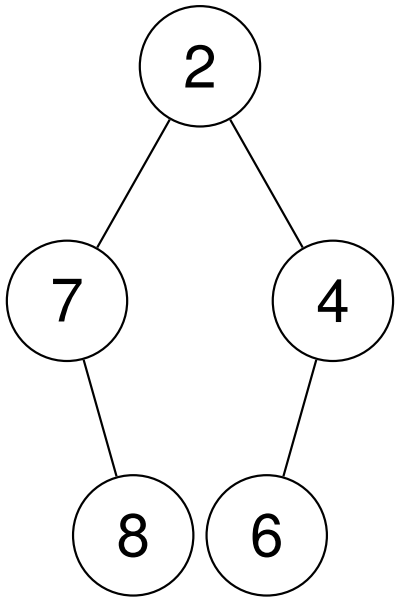
Índice

Heap

Árbol mínimo (o máximo)

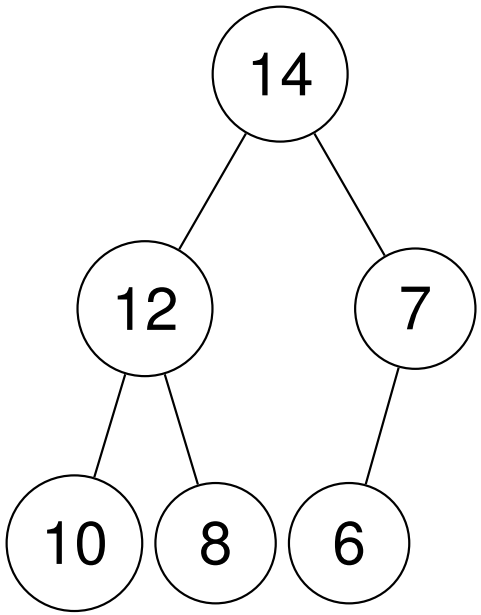
Árbol en el que la etiqueta de cada nodo es menor (o mayor) que la de sus hijos

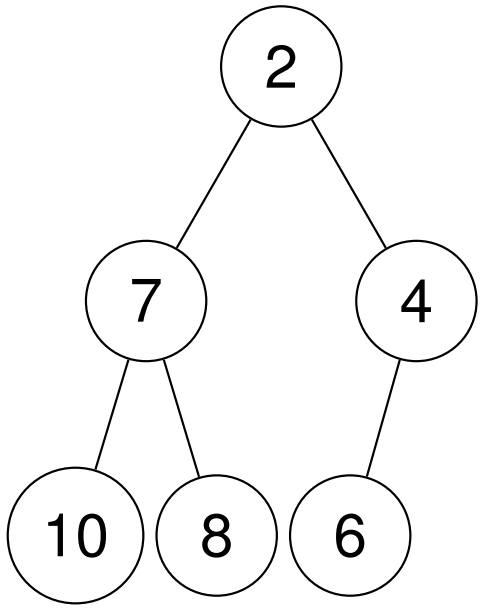




Heap mínimo (o máximo)

Árbol mínimo (o máximo) que además es completo





Heap

Pregunta

¿Dónde está el elemento máximo de un heap máximo?

Heap

Pregunta

¿Dónde está el elemento máximo de un heap máximo?

En la raíz, por eso la complejidad de la operación `max` es $O(1)$

Inserción en un heap

Algoritmo de inserción en un heap:

1. Insertar el elemento en la posición libre de más a la izquierda del último nivel, para que el árbol continúe siendo completo
2. Repetir la siguiente operación mientras que el árbol no cumpla las propiedades de heap:
   - Árbol mínimo: intercambiar el elemento insertado con su padre si el elemento es menor que su padre
   - Árbol máximo: intercambiar el elemento insertado con su padre si el elemento es mayor que su padre

Inserción en un heap

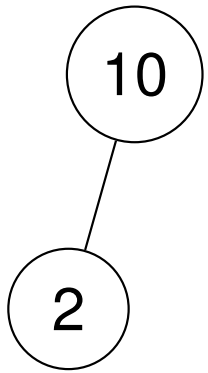
Ejemplo

Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21

Inserción en un heap

Ejemplo

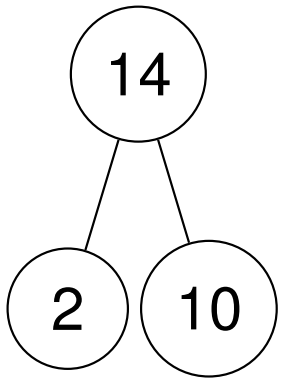
Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21



Inserción en un heap

Ejemplo

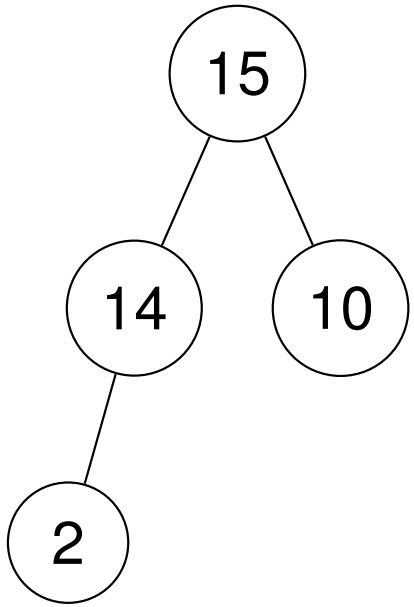
Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21



Inserción en un heap

Ejemplo

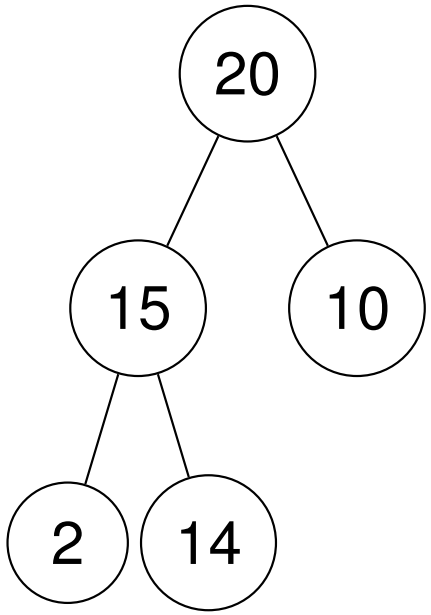
Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21



Inserción en un heap

Ejemplo

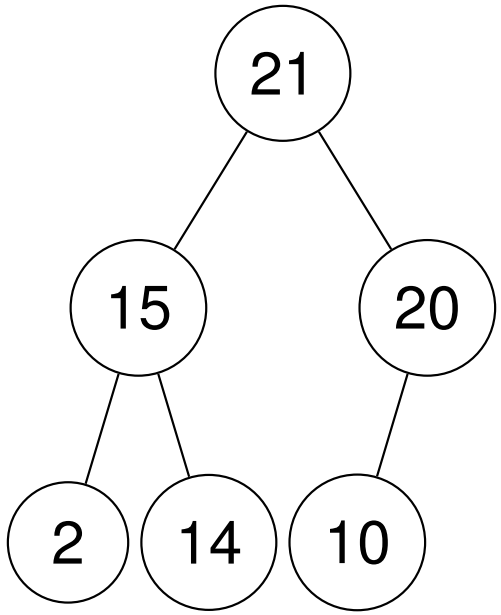
Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21



Inserción en un heap

Ejemplo

Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21



Inserción en un heap

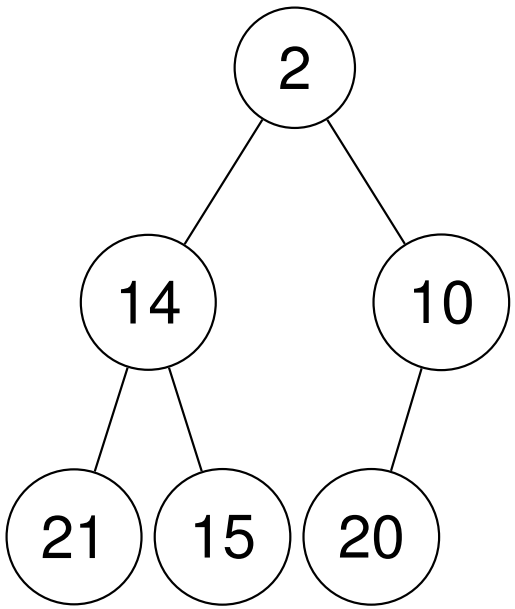
Ejercicio

Inserta en un heap mínimo inicialmente vacío los siguientes valores: 21, 20, 15, 2, 14, 10

Inserción en un heap

Ejercicio

Inserta en un heap mínimo inicialmente vacío los siguientes valores: 21, 20, 15, 2, 14, 10



Inserción en un heap

Pregunta

¿Cuándo se producen el mejor y peor caso en la inserción en un heap máximo? ¿Cuál es la complejidad asintótica respecto al número de elementos almacenados?

Inserción en un heap

Pregunta

¿Cuándo se producen el mejor y peor caso en la inserción en un heap máximo? ¿Cuál es la complejidad asintótica respecto al número de elementos almacenados?

- Mejor caso: el elemento a insertar es menor que su padre. $\Omega(1)$ (con implementación como vector)
- Peor caso: el elemento a insertar es mayor que sus ascendentes (y que el resto de elementos). $O(\log n)$

Borrado en un heap

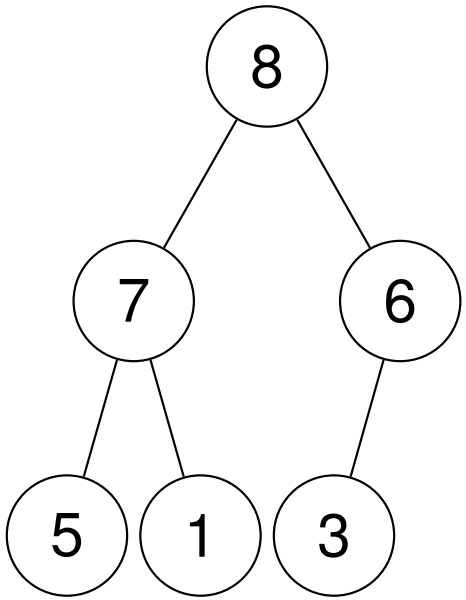
Algoritmo de borrado en un heap:

1. Mover el elemento más a la derecha del último nivel a la raíz
2. Repetir la siguiente operación mientras que el árbol no cumpla las propiedades de heap:
   - Árbol mínimo: intercambiar el elemento movido con el menor de sus hijos
   - Árbol máximo: intercambiar el elemento movido con el mayor de sus hijos

Borrado en un heap

Ejemplo

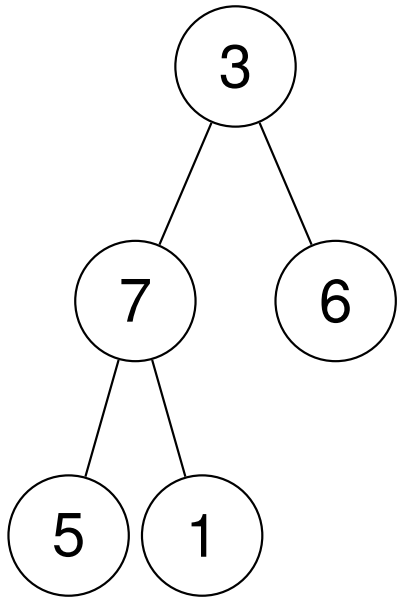
Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Ejemplo

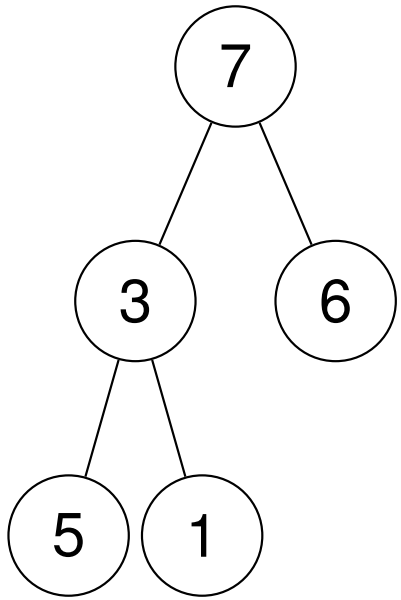
Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Ejemplo

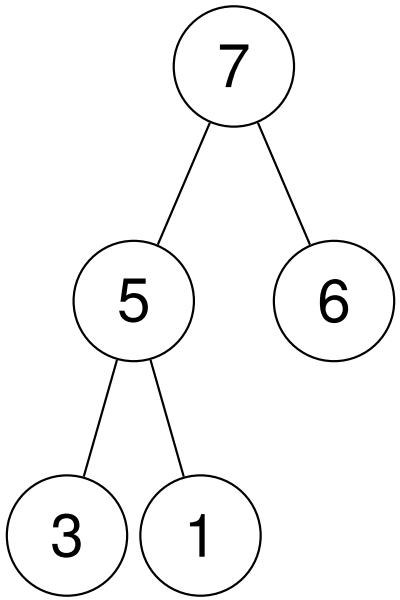
Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Ejemplo

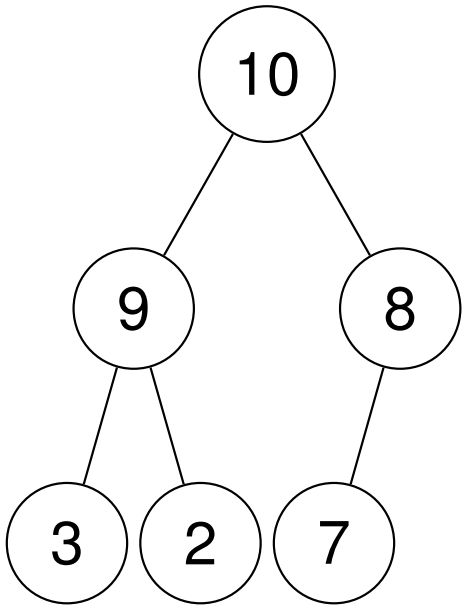
Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Ejercicio

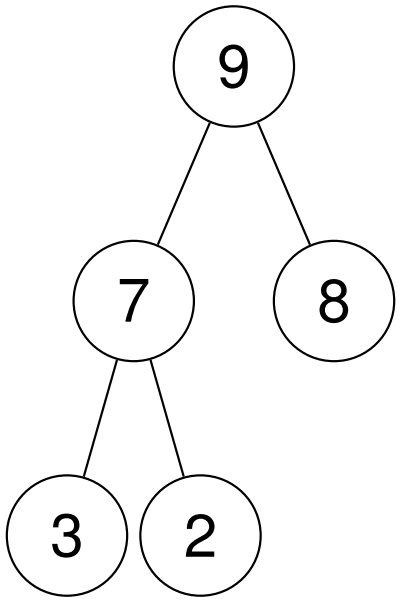
Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Ejercicio

Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Pregunta

¿Cuándo se producen el mejor y peor caso en el borrado en un heap máximo? ¿Cuál es la complejidad asintótica respecto al número de elementos almacenados?

Borrado en un heap

Pregunta

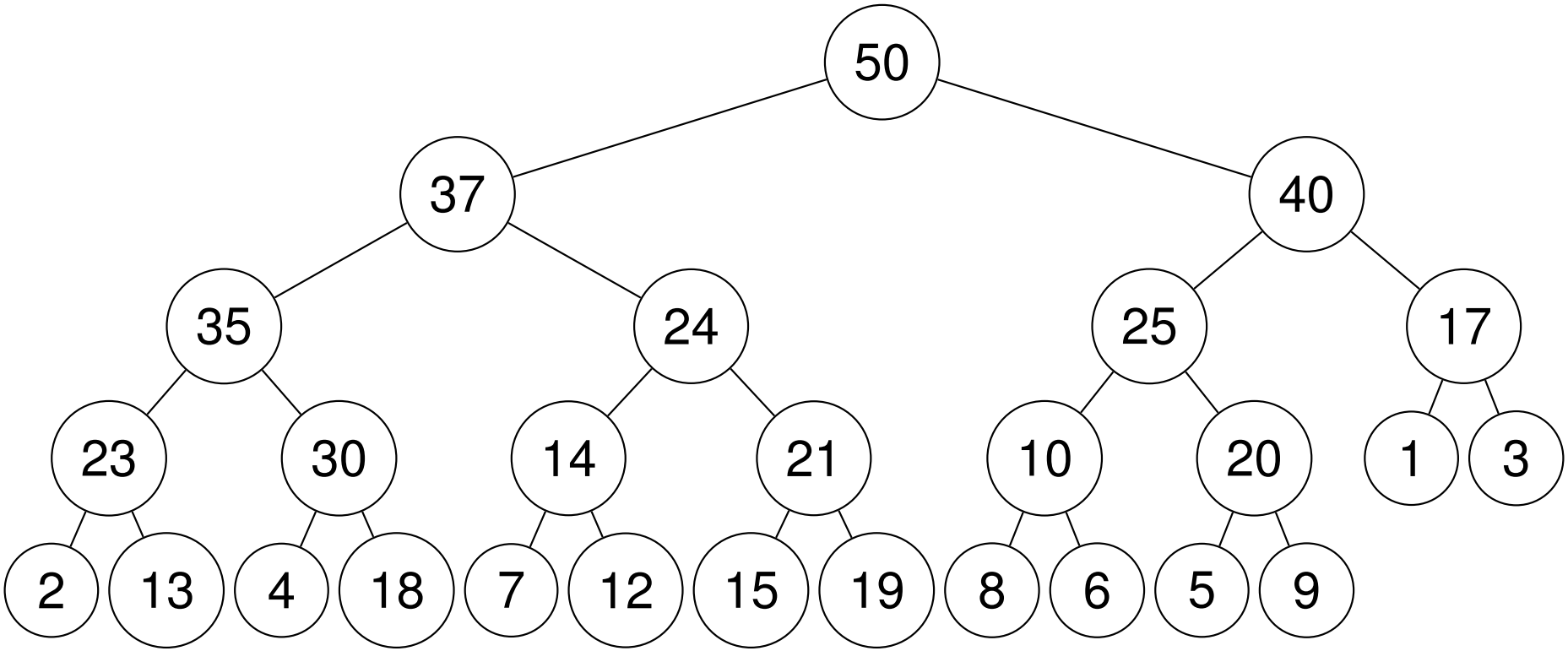
¿Cuándo se producen el mejor y peor caso en el borrado en un heap máximo? ¿Cuál es la complejidad asintótica respecto al número de elementos almacenados?

- Mejor caso: sólo se hace un intercambio de la raíz. $\Omega(1)$ (con implementación como vector)
- Peor caso: el elemento que se mueve a la reíz es menor que el resto de elementos del heap$\rightarrow$la raíz se intercambia hasta llegar a las hojas. $O(\log n)$

Ejercicios

Ejercicio

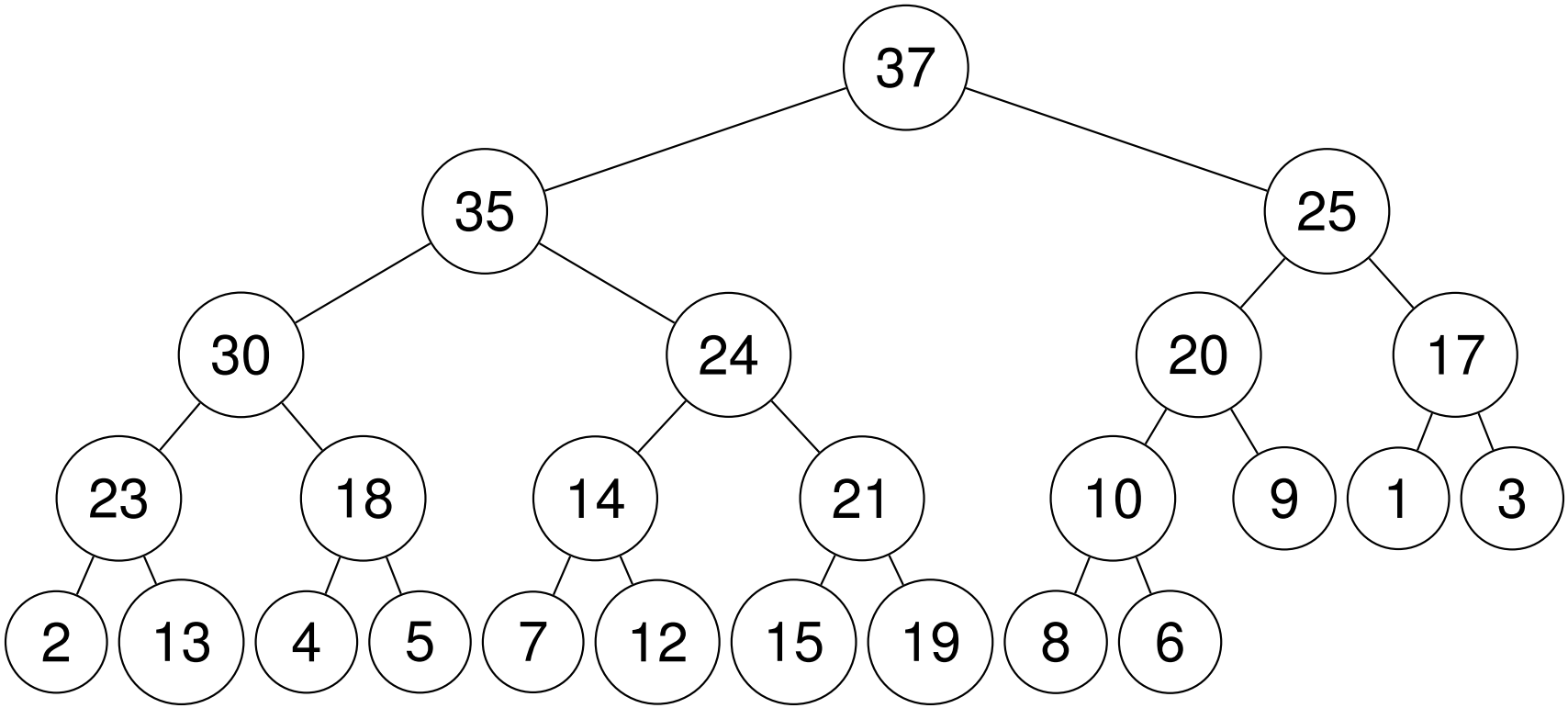
Realiza dos borrados sobre el siguiente montículo máximo:



Ejercicios

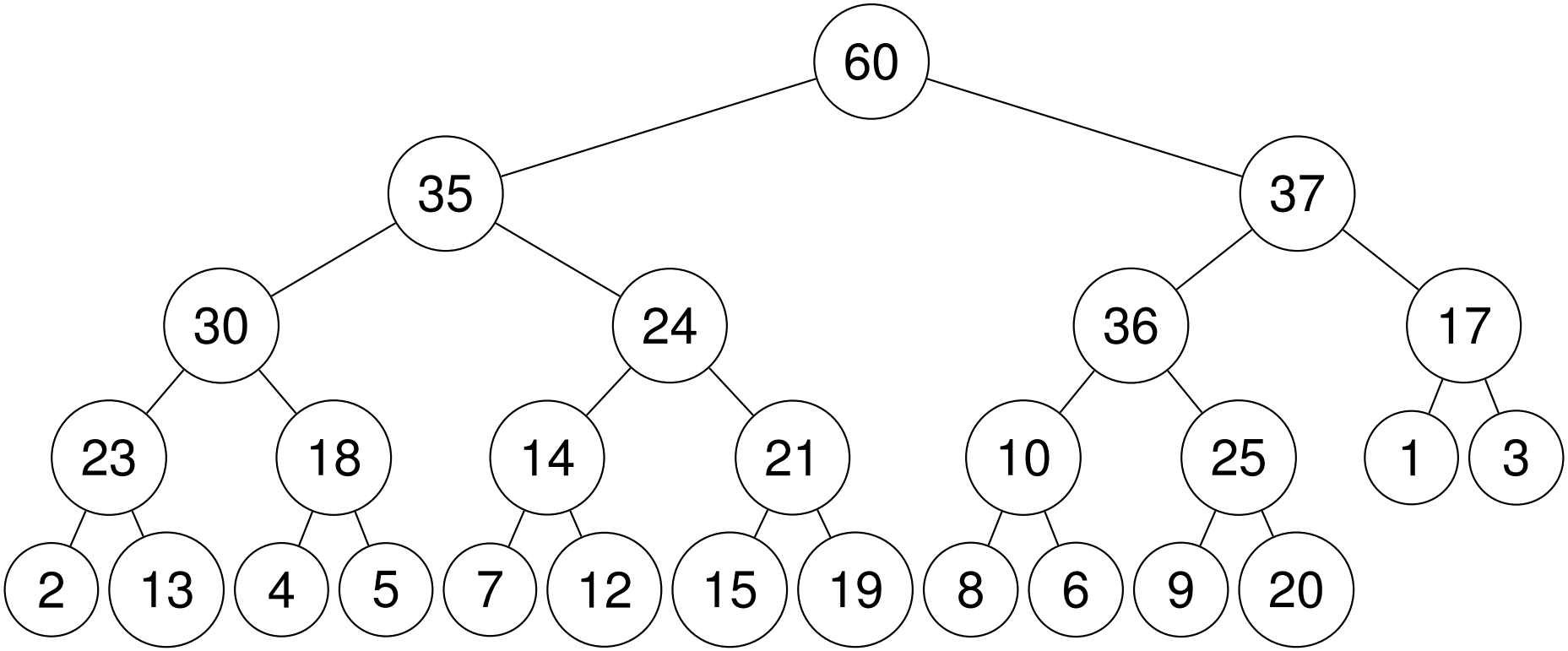
Ejercicio

Inserta 60, 36 en el siguiente montículo máximo:



Ejercicios

Resultado:



Representación

Pregunta

Teniendo en cuenta que un heap siempre es un árbol completo y el tipo de intercambios que se hacen, ¿qué representación es más adecuada: vector o enlazada?

Representación

Pregunta

Teniendo en cuenta que un heap siempre es un árbol completo y el tipo de intercambios que se hacen, ¿qué representación es más adecuada: vector o enlazada?

Las características del heap hacen que las grandes desventajas de la implementación vector desaparezcan: no hay desperdicio de memoria al ser un árbol completo, y los intercambios no implican desplazamientos de elementos en el vector. Además, insertar un elemento al final del último nivel tiene coste constante.

Índice

Algoritmo heapsort

- Puede emplearse la eficiencia de los heaps para ordenar un vector rápidamente
- Los algoritmos más simples para ordenación de vectores tienen complejidad $O(n^2)$
- *Heapsort*, el algoritmo de ordenación basado en heaps tiene una complejidad de $\Theta(n \cdot \log(n))$, la menor complejidad conocida para ordenación de vectores

Algoritmo heapsort

Primera fase:

- Parte izquierda del vector: heap
- Parte derecha del vector: elementos sin ordenar
- Se insertan los elementos de la parte desordenada de uno en uno

Segunda fase:

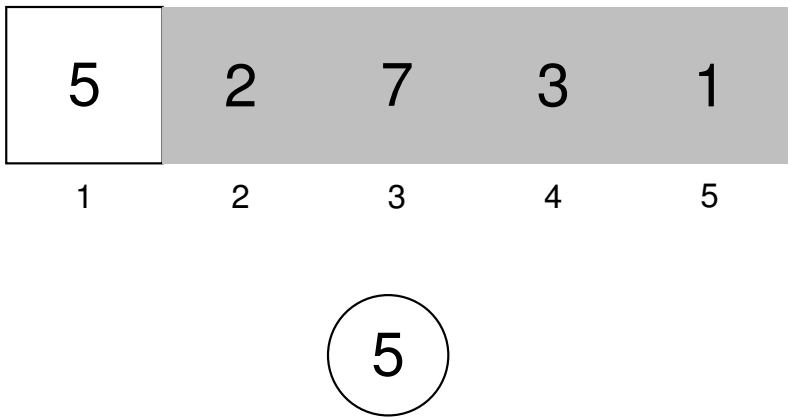
- Parte izquierda del vector: heap
- Parte derecha del vector: elementos ordenados
- Se borra iterativamente la raíz del heap y se pone en la parte derecha

Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 1:

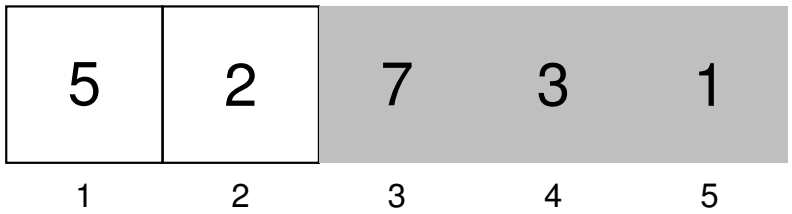


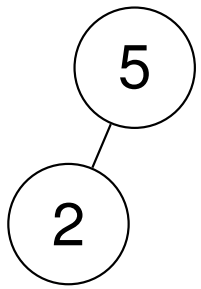
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 1:



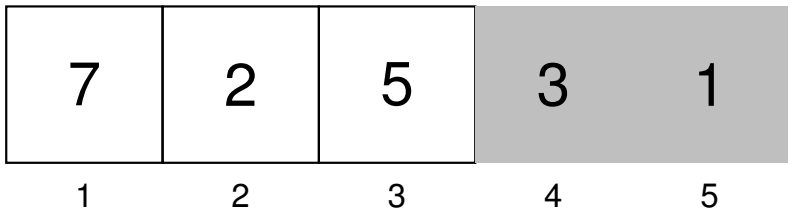


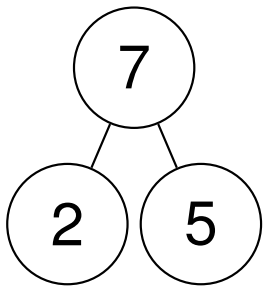
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 1:



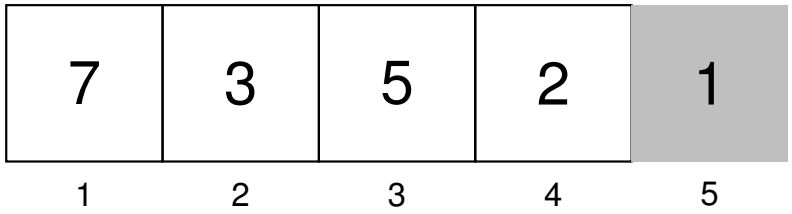


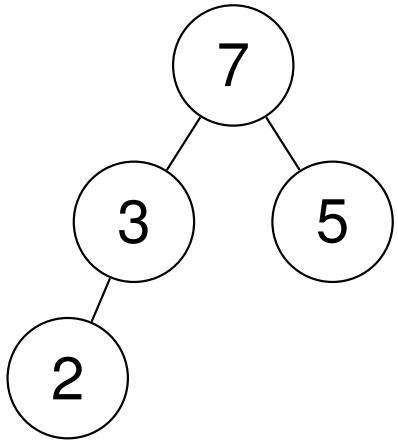
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 1:



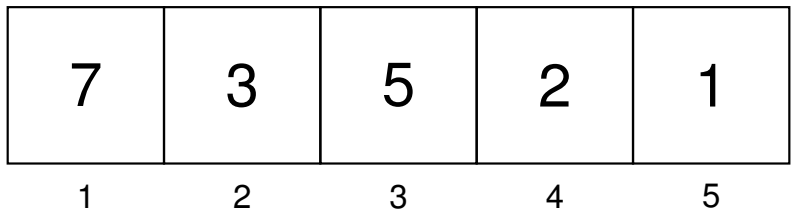


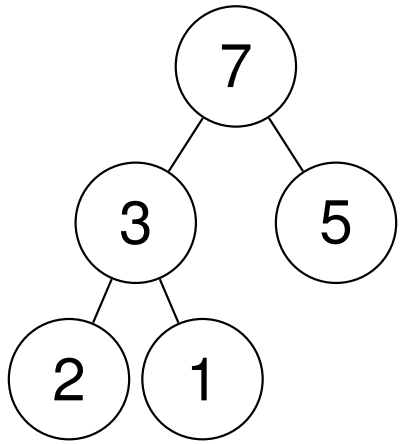
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 1:



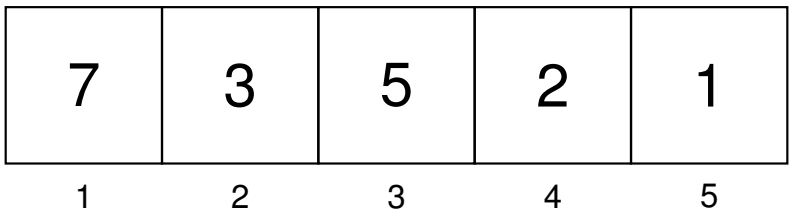


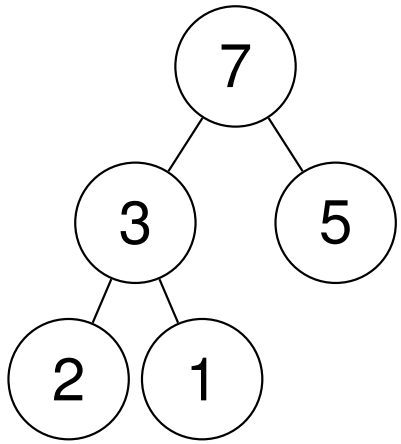
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 2:



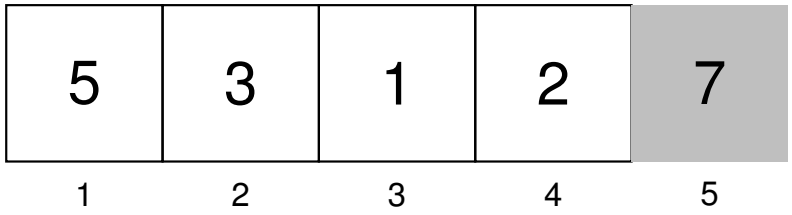


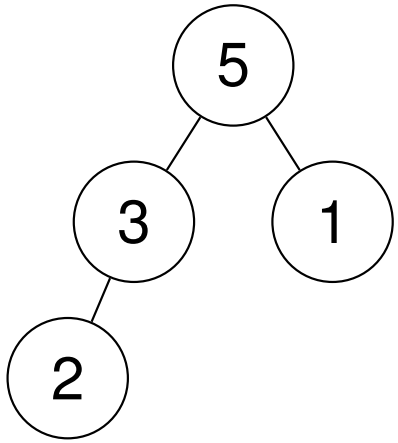
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 2:



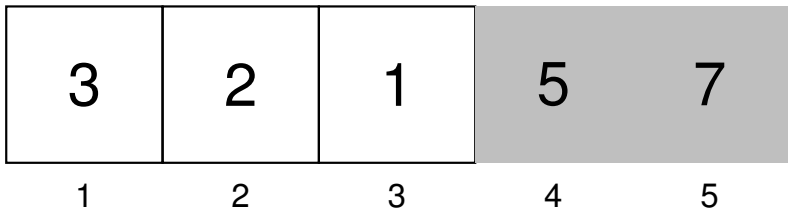


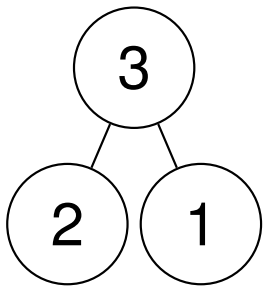
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 2:



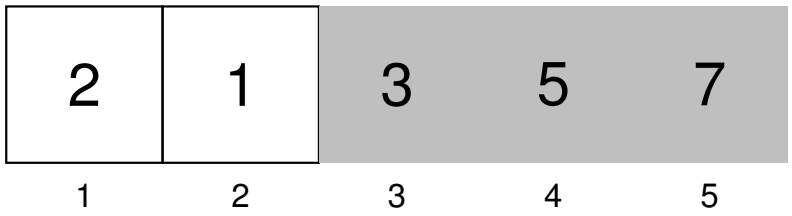


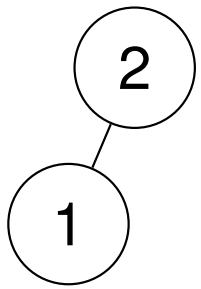
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 2:



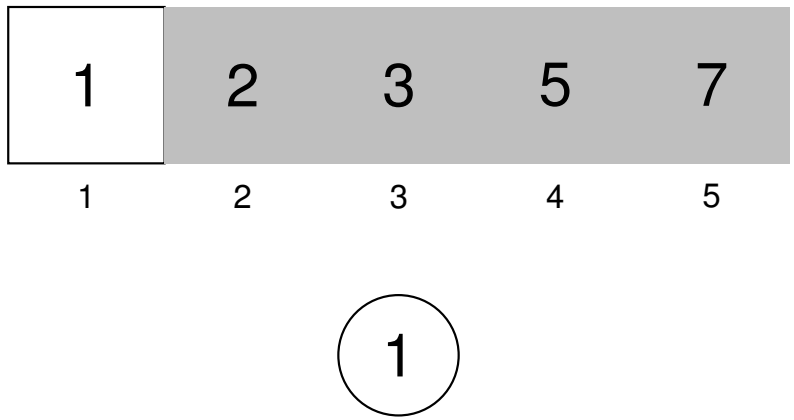


Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 2:



Algoritmo heapsort

Ejercicio

Ordena el siguiente vector empleando un heap mínimo (de mayor a menor): 9 5 7 4 8 6 2 1

Índice

- Implementada como un heap almacenado en vector

```
#include<queue>
using namespace std;
//...
priority_queue<int> q;

//Inserción en la cola con "push"
q.push(20);
q.push(100);
q.push(30);

//Obtención del elemento con máxima prioridad
cout << q.top() << endl; //100

//Eliminación del elemento con máxima prioridad
q.pop();

//Número de elementos en la colsa
cout << q.size() << endl;
```

Almacenando objetos

- Cualquier objeto que implemente un `operator<` puede almacenarse en la cola
- Se ordenarán de mayor a menor prioridad

```
#include<queue>
using namespace std;
//...
priority_queue<Peticion> q;


//Inserción en la cola con "push"
q.push(...);

//Obtención del elemento con máxima prioridad
cout << q.top().getNombreUsuario(); << endl;

//Eliminación del elemento con máxima prioridad
q.pop();
```

Programación Avanzada y Estructuras de Datos

8. Grafos

Víctor M. Sánchez Cartagena

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

11 de diciembre de 2024

Índice

Ejemplo introductorio

¿Qué estructura de datos emplearías para representar los siguientes problemas?

- Obtener la combinación de vuelos más barata para ir de un aeropuerto a otro
- Obtener la ruta por carretera más corta entre dos ciudades
- Crear redes de comunicaciones con mínimo coste para que todas las ciudades deseadas estén conectadas

Ejemplo introductorio

¿Qué estructura de datos emplearías para representar los siguientes problemas?

- Obtener la combinación de vuelos más barata para ir de un aeropuerto a otro
- Obtener la ruta por carretera más corta entre dos ciudades
- Crear redes de comunicaciones con mínimo coste para que todas las ciudades deseadas estén conectadas

Un **grafo**

Definición

Definición de grafo

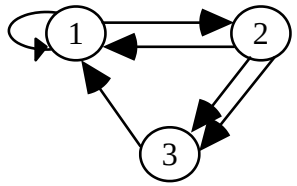
Un grafo G está formado por dos conjuntos V y A: $G = (V, A)$

- V es un conjunto finito no vacío de **vértices**
- A es un conjunto de **aristas** o **arcos**, tal que cada arista a_i es un par de vértices $a_i = (v_j, v_k)$

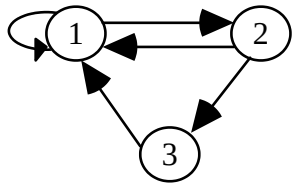
Tipos de grafo

Dependiendo de las restricciones sobre $a_i = (v_j, v_k)$:

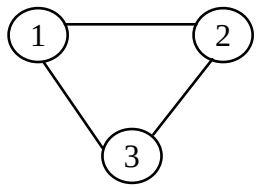
- **Multigrafo**: sin restricciones, existen arcos reflexivos y múltiples ocurrencias del mismo arco



- **Digrafo**: no hay múltiples ocurrencias del mismo arco



- **Grafo no dirigido**: las **aristas** indican que los vértices están conectados en ambos sentidos. No hay aristas reflexivas



Tipos de grafos

Pregunta

¿Cuál es el máximo número de aristas en un grafo no dirigido? ¿Cuál es el máximo número de arcos en un digrafo con n vértices?

Tipos de grafos

Pregunta

¿Cuál es el máximo número de aristas en un grafo no dirigido? ¿Cuál es el máximo número de arcos en un digrafo con n vértices?

No dirigido: $\frac{n \cdot (n-1)}{2}$ aristas

Tipos de grafos

Pregunta

¿Cuál es el máximo número de aristas en un grafo no dirigido? ¿Cuál es el máximo número de arcos en un digrafo con n vértices?

No dirigido: $\frac{n \cdot (n-1)}{2}$ aristas

Dirigido: n^2 arcos

Conceptos

- (Grafos no dirigidos) Los vértices v_1 y v_2 son adyacentes si $(v_1, v_2) \in A \vee (v_2, v_1) \in A$. La arista (v_1, v_2) (o (v_2, v_1)) es incidente a ambos vértices
- (Grafos dirigidos) El vértice v_1 es adyacente hacia v_2 y el vértice v_2 es adyacente desde v_1 si $(v_1, v_2) \in A$. El arco (v_1, v_2) es incidente a v_1 y v_2
- (Grafos no dirigidos) Adyacencia de un vértice: conjunto de vértices tal que hay una arista que los relaciona:

$$\mathrm{Ay}(x) = \{v_i : v_i \in V \wedge (x, v_i) \in A\}$$

Conceptos

- (Grafos dirigidos) Adyacencia de entrada:

$$\mathrm{Ay_E}(x) = \{v_i : v_i \in V \wedge (v_i, x) \in A\}$$

- (Grafos dirigidos) Adyacencia de salida:

$$\mathrm{Ay_S}(x) = \{v_i : v_i \in V \wedge (x, v_i) \in A\}$$

- (Grafos dirigidos) Grado de entrada: $|\mathrm{Ay_E}(x)|$
- (Grafos dirigidos) Grado de salida: $|\mathrm{Ay_S}(x)|$
- (Grafos dirigidos) Grado: $|\mathrm{Ay_E}(x)| + |\mathrm{Ay_S}(x)|$
- (Grafos no dirigidos) Grado: $|\mathrm{Ay}(x)|$

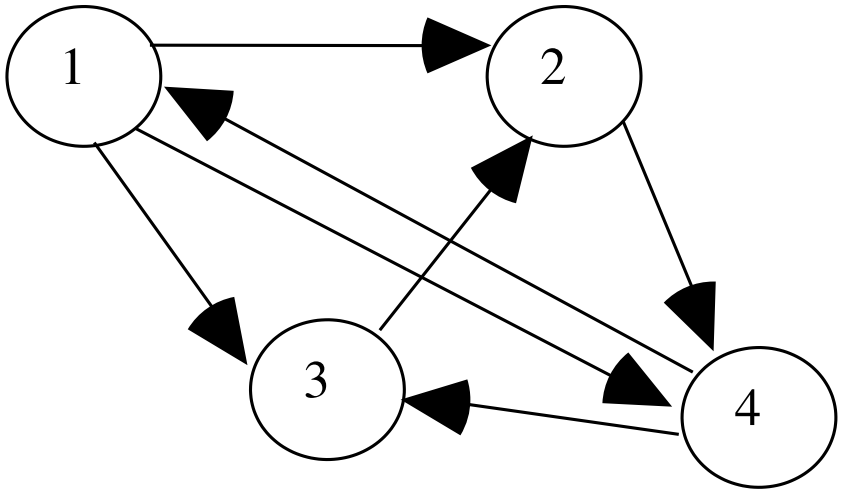
Conceptos

- Un camino desde v_p hasta v_q es una secuencia de vértices $(v_p, v_1, v_2, ..., v_n, v_q)$, tal que (v_p, v_1), (v_1, v_2), ..., (v_n, v_q) son arcos/aristas en A
- La longitud de un camino es el número de arcos/aristas que contiene
- Un camino simple no tiene vértices repetidos (excepto el primer y último)
- Un ciclo es un camino simple en el que el primer y último vértice coinciden

Conceptos

Pregunta

- ¿$(3, 2, 4, 3)$ es un ciclo? ¿De qué longitud?
- ¿$(3, 2, 4, 1, 4, 3)$ es un ciclo? ¿De qué longitud?



Conceptos

- Un subgrafo de un grafo $G = (V, A)$ es un grafo $G' = (V', A')$ tal que $V' \subseteq V$ y $A' \subseteq A$
- Un árbol extendido de un grafo $G = (V, A)$ es un subgrafo $T = (V', A')$ de G tal que T es un árbol y $V' = V$

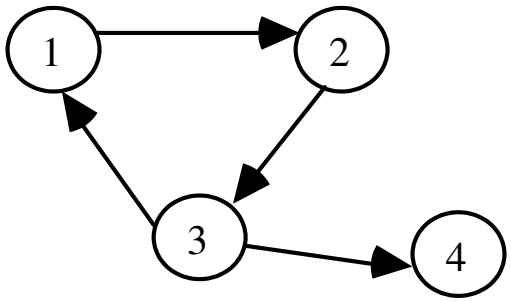
Conceptos

- Un grafo es conexo si $\forall v_i, v_j \in V$, existe un camino de v_i a v_j
- Las componentes fuertemente conexas de un grafo son el conjunto maximal de subgrafos conexos
  - Un grafo conexo tiene una única componente fuertemente conexa
- Un grafo acíclico es aquel que no tiene ciclos

Índice

Representación

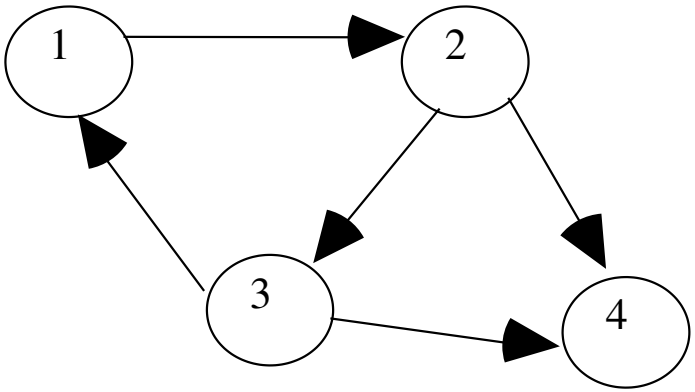
- Dado un grafo $G = (V, A)$ con $n = |V|$ (n vértices), la **matriz de adyacencia** es una matriz M cuadrada n x n tal que:
  - $M[i][j] = 1$ si $(v_i, v_j) \in A$
  - $M[i][j] = 0$ si $(v_i, v_j) \notin A$
- Para grafos no dirigidos, la matriz es simétrica respecto a la diagonal principal, que tiene todos los valores a 0.

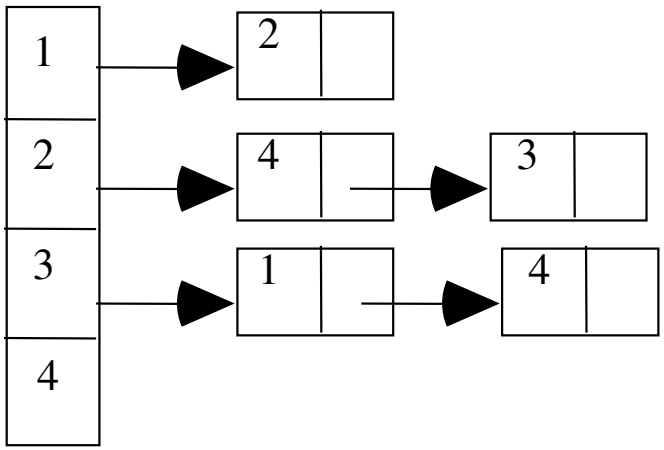


$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Representación

- **Lista de adyacencia**: Lista enlazada con la adyacencia de salida para cada vértice





Representación

Pregunta

¿Cuál es la complejidad temporal en el peor caso, para un digrafo con n vértices, de las siguientes operaciones en cada representación?

- Búsqueda (comprobar si hay un arco de v_1 a v_2)
- Inserción de un arco
- Cálculo de la adyacencia de salida de un vértice
- Cálculo de la adyacencia de entrada de un vértice

Operación	Matriz	Lista
Búsqueda		
Inserción		
$\mathrm{Ay_E}(x)$		
$\mathrm{Ay_S}(x)$		

Representación

Pregunta

¿Cuál es la complejidad temporal en el peor caso, para un digrafo con n vértices, de las siguientes operaciones en cada representación?

- Búsqueda (comprobar si hay un arco de v_1 a v_2)
- Inserción de un arco
- Cálculo de la adyacencia de salida de un vértice
- Cálculo de la adyacencia de entrada de un vértice

Operación	Matriz	Lista
Búsqueda	$O(1)$	$O(n)$
Inserción	$O(1)$	$O(n)$
$\mathrm{Ay_E}(x)$	$O(n)$	$O(n^2)$
$\mathrm{Ay_S}(x)$	$O(n)$	$O(n)$

Representación

Pregunta

¿Hay algún motivo para emplear lista de adyacencia en vez de matriz de adyacencia?

Representación

Pregunta

¿Hay algún motivo para emplear lista de adyacencia en vez de matriz de adyacencia?

Sí, la complejidad espacial. En la matriz siempre es $\Theta(n^2)$, y en la lista es proporcional al número de arcos. Preferiremos la representación en lista para grafos dispersos.

Índice

Recorrido en profundidad

- **Recorrer** un grafo consiste en listar todos sus vértices, una vez cada uno
- El recorrido en profundidad es una generalización del preorden aplicado a grafos

```
visitados ← {}
function DFS(v,G)
    visitados ← visitados ∪ {v}
    visitar(v)
    for w ∈ Ay_S(v) do
        if w ∉ visitados then
            DFS(w, G)
```

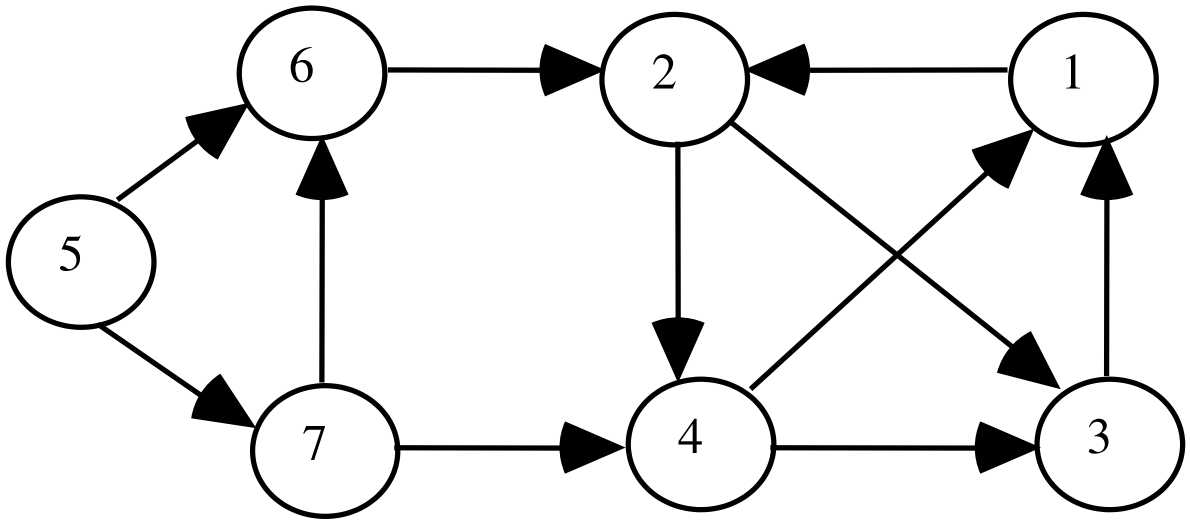
Recorrido en profundidad

- Los arcos (v, w) permiten construir un **árbol extendido en profundidad**
- Una vez construido el árbol, se pueden clasificar los arcos del grafo:
  - De árbol: los que forman parte del árbol extendido en profundidad
  - De retroceso: van de un vértice a un ascendente en el árbol
  - De avance: van de un vértice a un descendiente en el árbol
  - De cruce: ninguno de los anteriores
- **Bosque extendido en profundidad**: al terminar $DFS(v)$, se vuelve a iniciar el algoritmo si todavía no se han visitado todos los vértices
- El grafo es acíclico $\Leftrightarrow$ el bosque extendido en profundidad no tiene arcos de retroceso
- El árbol extendido en profundidad no contiene todos los vértices $\rightarrow$ el grafo no es conexo

Recorrido en profundidad

Ejercicio

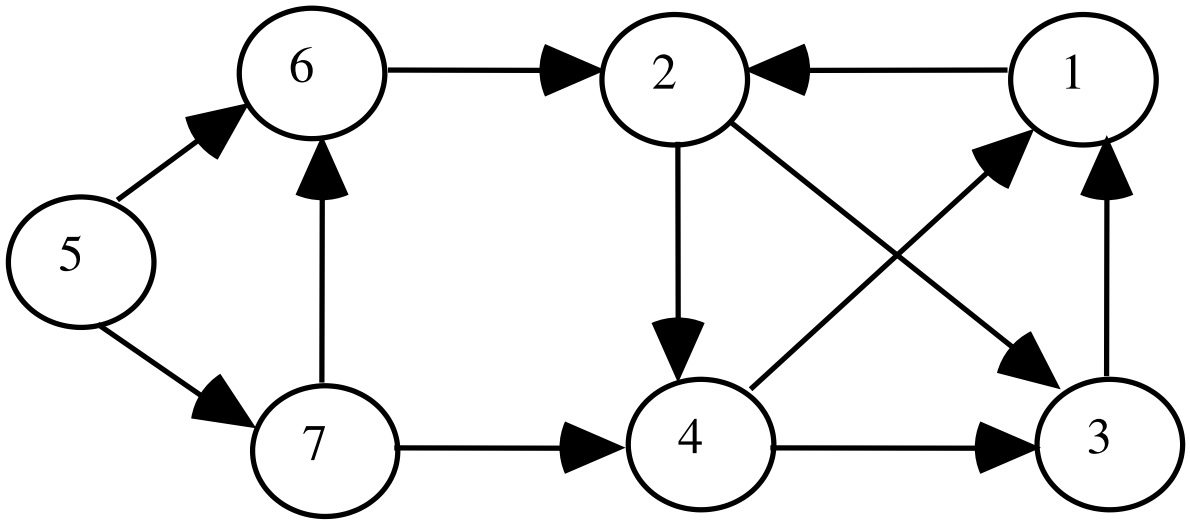
Calcula el recorrido en profundidad (DFS) del siguiente grafo, partiendo de los vértices 1,2,3,4 y 5. Recorre el conjunto Ay_S en orden de menor a mayor.



Recorrido en profundidad

Ejercicio

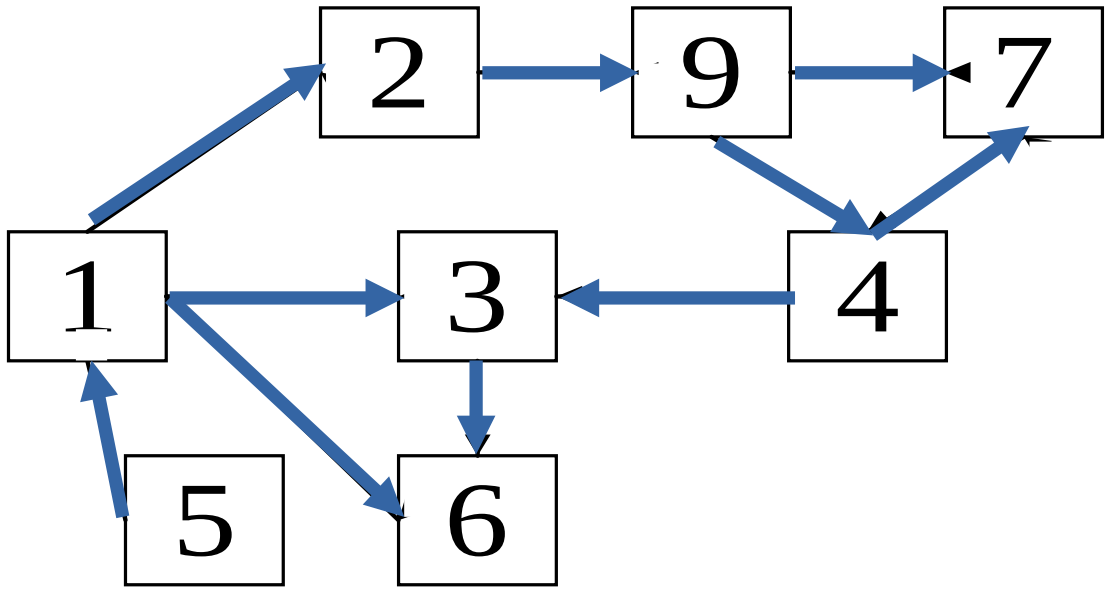
Calcula el bosque extendido en profundidad del siguiente grafo partiendo del vértice 5. Recorre el conjunto Ay_S en orden de mayor a menor. Clasifica los arcos.



Recorrido en profundidad

Ejercicio

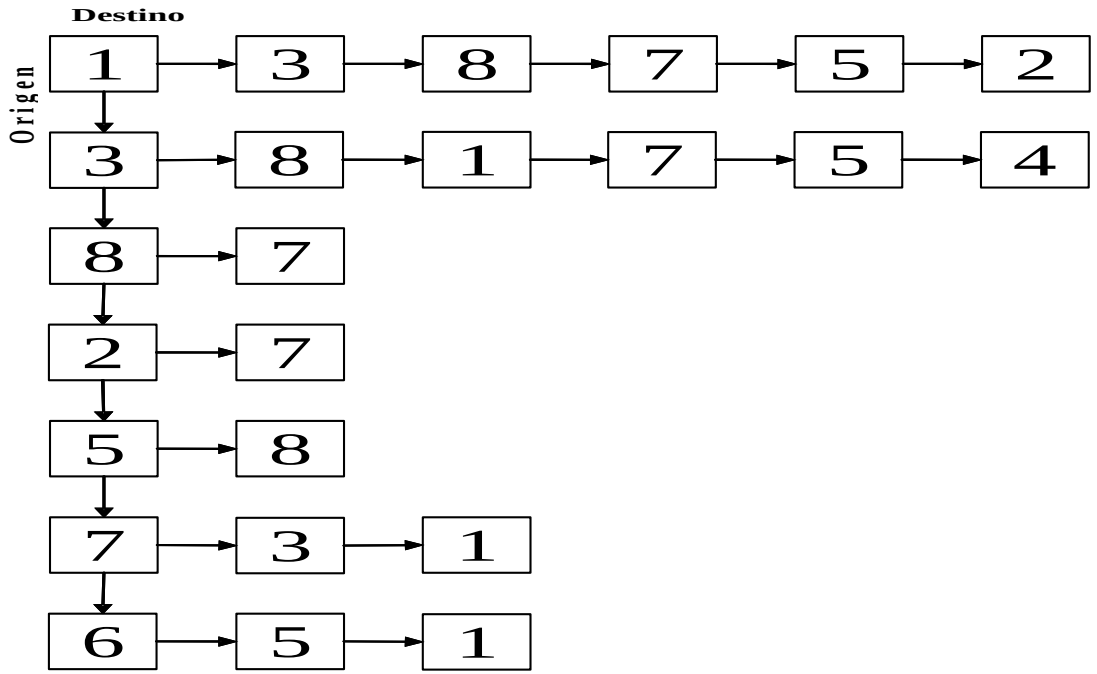
Calcula el bosque extendido en profundidad del siguiente grafo partiendo del vértice 1. Recorre el conjunto Ay_S en orden de menor a mayor. Clasifica los arcos. ¿Este grafo tiene ciclos? ¿Es conexo?



Recorrido en profundidad

Ejercicio

Calcula el bosque extendido en profundidad del siguiente grafo partiendo del vértice 1. Recorre el conjunto Ay_S en el mismo orden que aparece en las listas de adyacencia. Clasifica los arcos. ¿Este grafo tiene ciclos? ¿Es conexo?



Índice

Recorrido en anchura

function BFS(v,G)
    visitados $\leftarrow \{\}$
    cola $\leftarrow ()$
    visitados $\leftarrow$ visitados $\cup \{v\}$
    enqueue(*cola*, *v*)
    visitar(v)
    while cola $\neq ()$ **do**
        $w_1 \leftarrow$ front(*cola*)
        dequeue(*cola*)
        for $w_2 \in \mathrm{Ay}_S(w_1)$ **do**
            if $w_2 \notin$ visitados **then**
                visitar(w_2)
                enqueue(*cola*, w_2)
                visitados $\leftarrow$ visitados $\cup \{w_2\}$

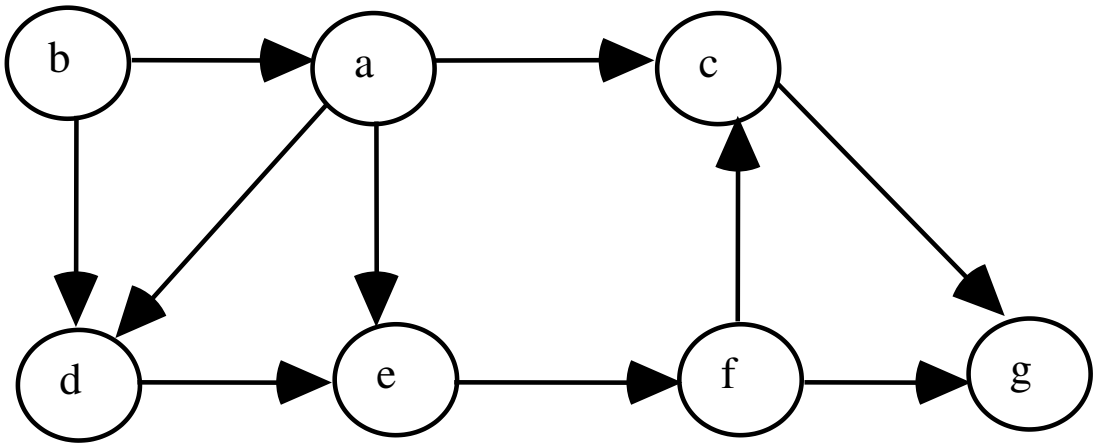
Recorrido en anchura

- Los arcos (w_1, w_2) permiten construir un **árbol extendido en anchura**
- Los arcos se pueden clasificar del mismo modo que en el recorrido en profundidad
- Las propiedades sobre ciclos y grafos conexos también se mantienen
- El bosque extendido en anchura contiene los **caminos más cortos** desde el vértice inicial a cada vértice

Recorrido en anchura

Ejercicio

Calcula el recorrido en anchura (BFS) del siguiente grafo, partiendo de los vértices a, b y d. Recorre el conjunto Ay_S en orden alfabético. Dibuja los correspondientes árboles extendidos en anchura.



Recorridos en grafos no dirigidos

- Se aplican los mismos algoritmos DFS y BFS presentados anteriormente
- Sólo hay dos tipos de aristas:
  - De árbol
  - De retroceso: forman ciclos
- El grafo es conexo $\Leftrightarrow$ el árbol extendido en profundidad contiene todos los vértices

Recorridos en grafos no dirigidos

Ejercicio

Calcula el bosque extendido en profundidad y el bosque extendido en anchura a partir del vértice 1. Clasifica los arcos. Recorre el conjunto Ay de menor a mayor. El grafo es **no dirigido** y la lista de adyacencia sólo contiene las aristas en uno de los dos sentidos.

$1 \to 3 \to 4 \to 6 \to 8 \to 9$
$2 \to 4 \to 5 \to 7 \to 10$
$3 \to 5 \to 6$
$4 \to 6 \to 8$
$5 \to 7 \to 8 \to 10$
$6 \to 7 \to 8 \to 9$
$7 \to 8 \to 9$
$8 \to 9 \to 10$
$9 \to 10$
10

Ejercicio adicional

Ejercicio

Calcula el bosque extendido en profundidad a partir del vértice 1 del siguiente grafo **dirigido**. Clasifica los arcos. Recorre el conjunto Ay_S de menor a mayor. Identifica al menos 4 ciclos en el grafo.

$1 \to 3 \to 8$
$2 \to 4 \to 5 \to 7 \to 9 \to 10$
$3 \to 5 \to 6 \to 10$
$4 \to 2 \to 6 \to 8$
$5 \to 7$
$6 \to 7 \to 8 \to 10$
7
$8 \to 1 \to 10$
$9 \to 4$
$10 \to 3 \to 5$
$11 \to 10 \to 12 \to 13$
$12 \to 2 \to 3 \to 11$

Índice

Grafos acíclicos dirigidos

- La detección de ciclos (arcos de retroceso) permite comprobar si un grafo es acíclico
- Los grafos acíclicos tienen múltiples aplicaciones:
  - Planificación de tareas: los vértices representan tareas y los arcos, requisitos: es necesario completar la tarea A antes de comenzar la tarea B
  - Dependencias entre asignaturas en un plan de estudios
  - Herencia entre clases de un programa en C++
  - etc.
- **Ordenamiento topológico**: ordenar los vértices de tal manera que para todo arco (v_i, v_j), v_i aparece antes que v_j en el ordenamiento
  - Para la planificación de tareas, implica ordenar las tareas de manera que todos los requisitos se cumplen