Tipos lineales

- Permiten almacenar relaciones de orden o secuencia entre los elementos:

  Secuencias de elementos de un cierto tipo dispuestos en una dimensión

  e1 e2 e3 e4 e5 e6 e7 … eN

  A C X B A D P … E

Toda secuencia refleja un orden en sus elementos, resultado de colocarlos en una dimensión

  - Mucha variedad de posibles TADs:
    - Con o sin elementos repetidos
    - Criterios de orden en la secuencia:
      - respecto a los propios datos en los elementos (Ej.: por orden alfabético),
      - respecto a criterios externos a los datos (Ej.: por orden de llegada o inserción)
    - Operaciones típicas:
      - *crear, añadir, esVacía?, quitar, pertenece, tamaño, …*
    - Operaciones de acceso y recorrido (iteradores) → según el orden en la secuencia
      - primero, siguiente, esÚltimo?, último, esPrimero?, anterior, …
    - Pueden ser operaciones:
      - limitadas a determinados extremos de la secuencia
      - relativas al último elemento accedido o insertado
      - relativas a la posición en la secuencia

El TAD diccionario. Especificación. Implementaciones lineales

Lección 10

Contenedores (lección 5)

- **Diccionario (o mapa):**
  - Conjunto de 0 ó más elementos formados como pares (clave, valor)
    - DEFINICIÓN: función de elementos de tipo clave en elementos de tipo valor
    - cada clave tiene asociado un valor
  - No se permiten claves repetidas pero varias claves pueden corresponderse con el mismo valor
  - Las claves no pueden cambiar, los valores si
  - Operaciones de acceso y manipulación *por clave*
  - Operaciones típicas:
    - **crear**: crea un diccionario sin elementos
    - **añadir**: dada una clave y un valor, asigna el valor a la clave en el diccionario
    - **pertenece**: dada una clave devuelve un booleano indicando si se encuentra en el diccionario
    - **obtenerValor**: dada una clave devuelve el valor asociado a ella
    - **quitar**: dada una clave la borra del diccionario junto con su valor
    - **cardinal**: devuelve el número de elementos en el diccionario
    - **esVacío?**: comprueba si el diccionario esta vacío (no contiene ningún elemento)
    - …

    Situaciones de error en general:
    - Si se utilizan claves o valores de tipo distinto al de los del diccionario
    - Si se intentan obtener o eliminar para claves no existentes en el diccionario

Especificación Diccionarios

- **Es imprescindible que las claves se puedan comparar por igualdad**
- **No es imprescindible, pero si es habitual, que las claves se puedan comparar y ordenar**

  →Si se pueden ordenar las claves, el diccionario se puede organizar como una secuencia de elementos (clave, valor) ordenada por clave (secuencia ordenada sin repetidos) → *lineal*

espec diccionarios
  usa booleanos, naturales
  parámetros formales
    géneros clave, valor
    operación *{suponemos que en el género de las claves hay definida una función de comparación "<" y otra de igualdad "=" }*
      _=_: clave c1 , clave c2 -> booleano *{verdad si y solo si c1 igual que c2}*
      _<_: clave c1 , clave c2 -> booleano *{verdad si y solo si c1 menor que c2}*

  fpf
  género diccionario *{Los valores del TAD representan conjuntos de pares (clave,valor) en los que no se permiten claves repetidas}*
operaciones
    crear: → diccionario
    {Devuelve un diccionario vacío, sin elementos (pares)}

    añadir: diccionario d , clave c , valor v → diccionario
    {Si en d no hay ningún par con clave c, devuelve un diccionario igual al resultante de añadir el par (c,v) a d; si en d hay un par (c,v'), entonces devuelve un diccionario igual al resultante de sustituir (c,v') por el par (c,v) en d}
  …

Especificación Diccionarios

...

pertenece?: clave c , diccionario d → booleano
{Devuelve verdad si y sólo si en d hay algún par (c,v)}

parcial obtenerValor: clave c , diccionario d → valor
{Devuelve el valor asociado a la clave c en d.
Parcial: la operación no está definida si c no está en d (not pertenece?(c,d)) }

quitar: clave c , diccionario d → diccionario
{Si c está en d, devuelve un diccionario igual al resultante de borrar en d el par con clave c; si c no está en d, devuelve un diccionario igual a d}

cardinal: diccionario d → natural
{Devuelve el nº de elementos (de pares) en el diccionario d}
esVacío?: diccionario d → booleano
{Devuelve verdad si y sólo si d no tiene elementos}

. . . ***{...operaciones de iterador,... }***

fespec

Especificación Diccionarios

...

{operaciones de iterador interno, que permitirá visitar los elementos del diccionario siguiendo el orden por clave (en este caso, de menor a mayor): }

iniciarIterador: diccionario d → diccionario
{ Prepara el iterador y su cursor para que el siguiente elemento (par) a visitar sea el primero del diccionario d (situación de no haber visitado ningún elemento)}

existeSiguiente?: diccionario d → booleano
{ Devuelve falso si ya se ha visitado el último elemento, devuelve verdad en caso contrario}

parcial siguienteClave: diccionario d→ clave
{ Devuelve la clave del siguiente elemento (par) de d.
Parcial: la operación no está definida si no existeSiguiente?(d) }
parcial siguienteValor: diccionario d→ valor
{ Devuelve el valor del siguiente elemento (par) de d.
Parcial: la operación no está definida si no existeSiguiente?(d) }

parcial avanza: diccionario d → diccionario
{ Devuelve el diccionario resultante de avanzar el cursor en d.
Parcial: la operación no está definida si no existeSiguiente?(d) }

fespec

Implementación. Interfaz.

```
módulo genérico diccionarios
parámetros
  tipos clave,valor
  con
    función "<"(c1,c2:clave) devuelve booleano{verdad si c1<c2…}
    función "="(c1,c2:clave) devuelve booleano{verdad si c1=c2…}

exporta
  tipo diccionario
  procedimiento crear(sal d:diccionario)
  procedimiento añadir(e/s d:diccionario;
                       ent c:clave; ent v:valor)

  procedimiento buscar(ent d:diccionario; ent c:clave;
                       sal éxito:booleano; sal v:valor)
  {pertenece? y obtenerValor se implementan en una única operación}

  procedimiento quitar(ent c:clave; e/s d:diccionario)
  función cardinal(d:diccionario) devuelve natural
  función esVacío?(d:diccionario) devuelve booleano
  ...
```

Implementación. Interfaz.

```
{Operaciones para duplicar, comparar y liberar}
procedimiento duplicar(sal dSal:diccionario;
                       ent dEnt:diccionario)
función iguales?(d1,d2:diccionario) devuelve booleano

{Si hacemos una implementación en memoria dinámica,
 tendremos:}
procedimiento liberar(e/s d:diccionario)

{Operaciones de Iterador interno:}
procedimiento iniciarIterador(e/s d:diccionario)
función existeSiguiente?(d:diccionario) devuelve booleano
procedimiento siguienteYAvanza(e/s d:diccionario;
                               sal c:clave; sal v:valor;
                               sal error:booleano)
implementación
  ... {¿REPRESENTACION INTERNA?}
```

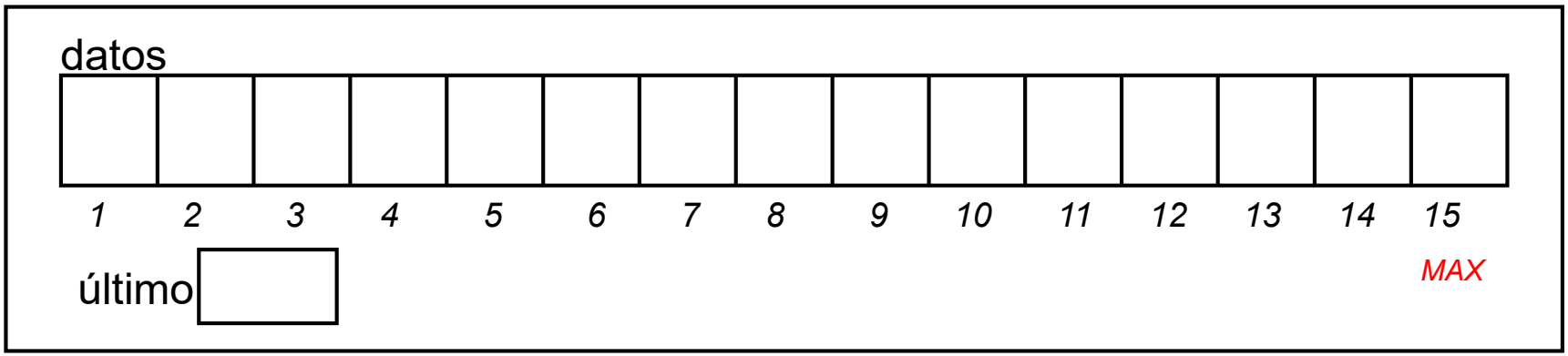
Implementación **estática** del TAD diccionario con listas ordenadas

Implementación estática ordenada

- Representación interna en pseudocódigo:

```
constante max= …;
Tipo
    diccionario = registro
        datos: vector [1..max] de elemento;
        último: 0..max;
    freg
```

```
elemento = registro
        laClave: clave
        elValor: valor
    freg;
```



Θ(Max) en memoria

Implementación estática ordenada

- Implementación similar a las listas vistas en la *lección 6*, pero **ordenadas y sin repetidos** (ya conocidas de primero):
  - Crear, cardinal (longitud), esVacío? :
    - Idénticas a las vistas anteriormente *(Θ(1) en tiempo)*
  - Añadir→ inserción ordenada en un vector (ej: orden de menor a mayor)
    - **Búsqueda o recorrido** desde el primero hasta encontrar elemento. Si se encuentra, actualizar su valor. Si no se encuentra, encontraremos un elemento mayor que el buscado, en cuyo caso ese será el punto en el que hay que hacerle hueco desplazando todos los que le siguen una posición hacia la derecha, o comprobaremos todos los elementos de la lista *(Θ(N) en tiempo)*

Impl. Parcial por el caso de vector lleno

  - Buscar (Pertenece + obtenerValor)→ búsqueda eficiente en un vector ordenado
    - Búsqueda dicotómica *(Θ(log N) en tiempo)*
  - Quitar → borrado en un vector ordenado (ej: orden de menor a mayor)
    - **Búsqueda o recorrido** desde el primero hasta encontrar elemento o, en el caso de que no esté encontrar un elemento mayor que el buscado o comprobar todos los elementos de la lista. Si se encuentra el elemento, desplazar todos los que le siguen una posición hacia la izquierda *(Θ(N) en tiempo)*
  - Operaciones de copia y comparación por igualdad → idénticas a las vistas en la lección 6

Implementación estática con iterador interno

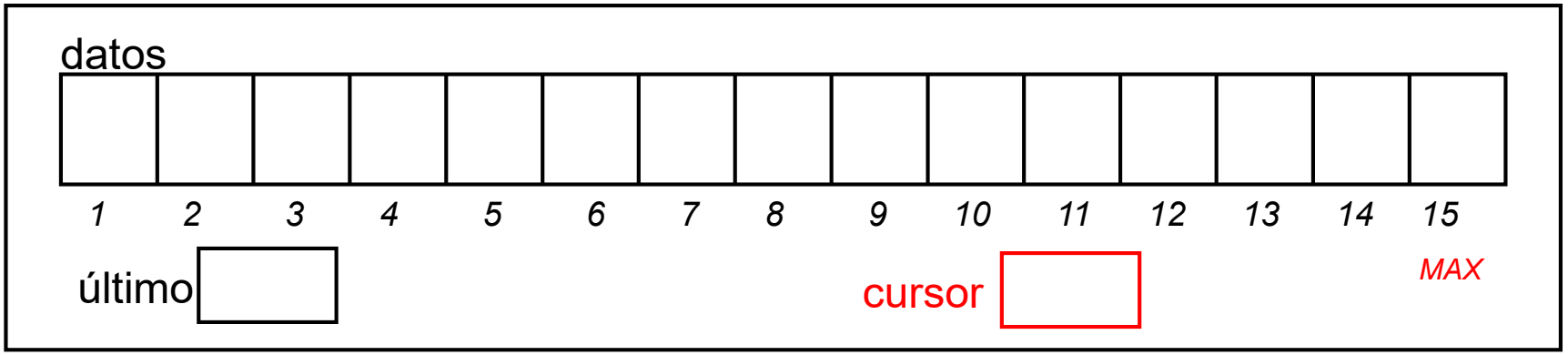
- Representación interna para tener el iterador:

```
constante max= …;
Tipo
    diccionario = registro
        datos: vector [1..max] de elemento;
        último: 0..max;
        cursor:1..(max+1)
    freg
```

```
elemento = registro
        laClave: clave
        elValor: valor
    freg;
```

Θ(Max) en memoria

diccionario



Implementación estática con iterador

Iterador interno:

requiere modificar la representación del TAD

Las demás operaciones del TAD no utilizarán el campo cursor

```
procedimiento iniciarIterador(e/s d:diccionario)
principio
   d.cursor:=1
fin
```

$\Theta(1)$ en tiempo

```
función existeSiguiente?(d:diccionario) devuelve booleano
principio
   devuelve d.último>d.cursor
fin
```

$\Theta(1)$ en tiempo

Implementación estática con iterador

```
procedimiento siguienteYAvanza(e/s d:diccionario;
                               sal c:clave; sal v:valor;
                               sal error:booleano)
principio
  si existeSiguiente?(d) entonces
    error:=falso;
    c:=d.datos[cursor].laClave;
    v:=d.datos[cursor].elValor;
    d.cursor:=d.cursor+1
  sino
    error:=verdad
  fsi
fin
```

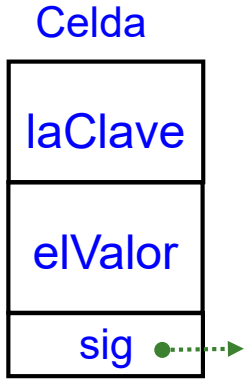
$\Theta(1)$ en tiempo

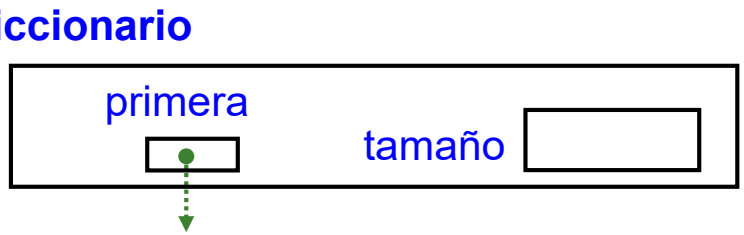
Implementación **dinámica** del TAD diccionario **con** ***lista enlazada ordenada***

Implem. dinámica con listas ordenadas

{El diccionario se representa en memoria dinámica con una lista enlazada tal que las claves se mantienen ordenadas ("<")}

```
tipos punteroCelda = ↑Celda;
      Celda = registro
                  laClave:clave;
                  elValor:valor;
                  sig:punteroCelda
                freg
      diccionario = registro
                      primera:punteroCelda;
                      tamaño:natural
                    freg
...
```





Θ(N) en memoria

Implem. dinámica con listas ordenadas

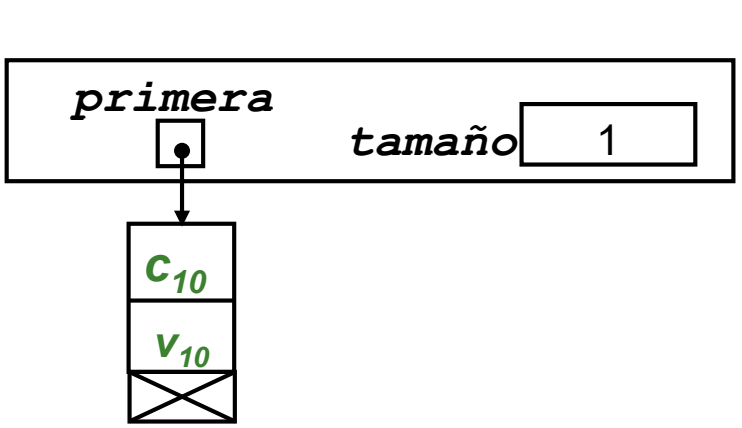
Casos a distinguir:

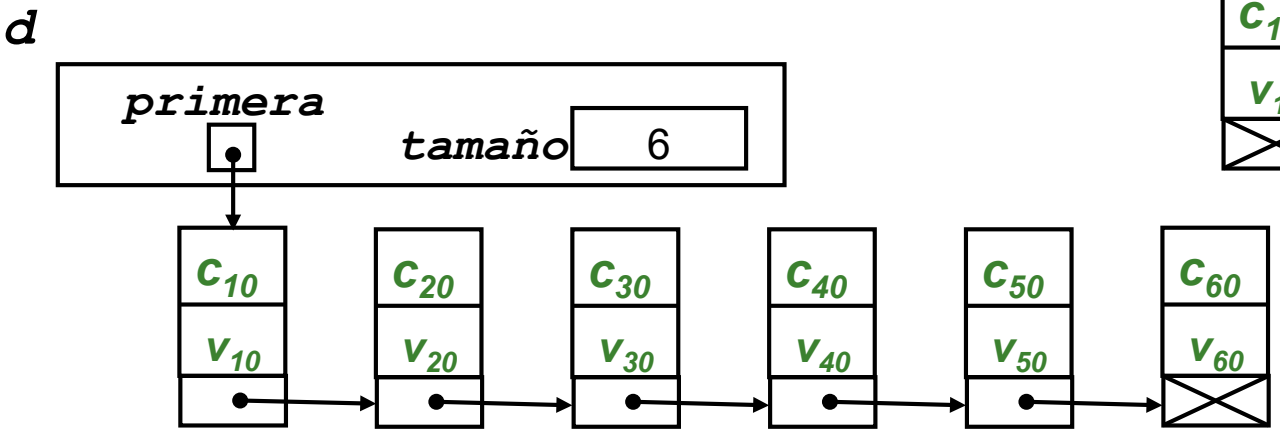
- Diccionario vacío:

$\Theta(N)$ en memoria

- Diccionario NO vacío:

{en su versión más sencilla} →





Implem. dinámica con listas ordenadas

```
procedimiento crear(sal d:diccionario)
principio
   d.primera:=nil; d.tamaño:=0
fin
```

$\Theta(1)$ en tiempo

```
función cardinal(d:diccionario) devuelve natural
principio
   devuelve d.tamaño
fin
```

$\Theta(1)$ en tiempo

```
función esVacío?(d:diccionario) devuelve booleano
principio
   devuelve d.primera=nil
fin
```

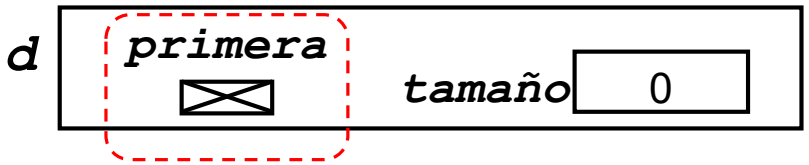
$\Theta(1)$ en tiempo

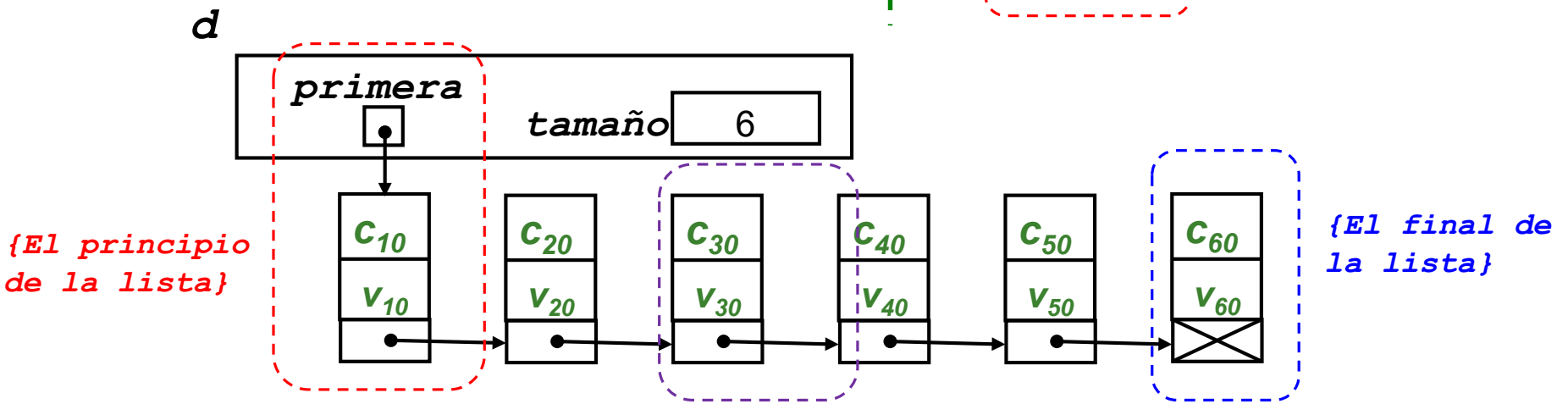
➢ ***Diccionario vacío:***

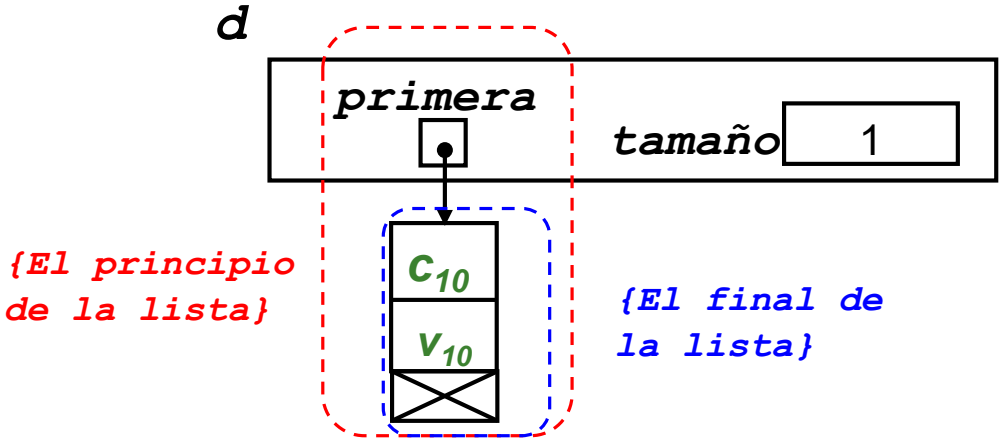
Implem. dinámica con listas ordenadas

- **Casos generales a considerar al tratar con una lista enlazada ordenada (no vacía):**

- **Diccionario vacío:**







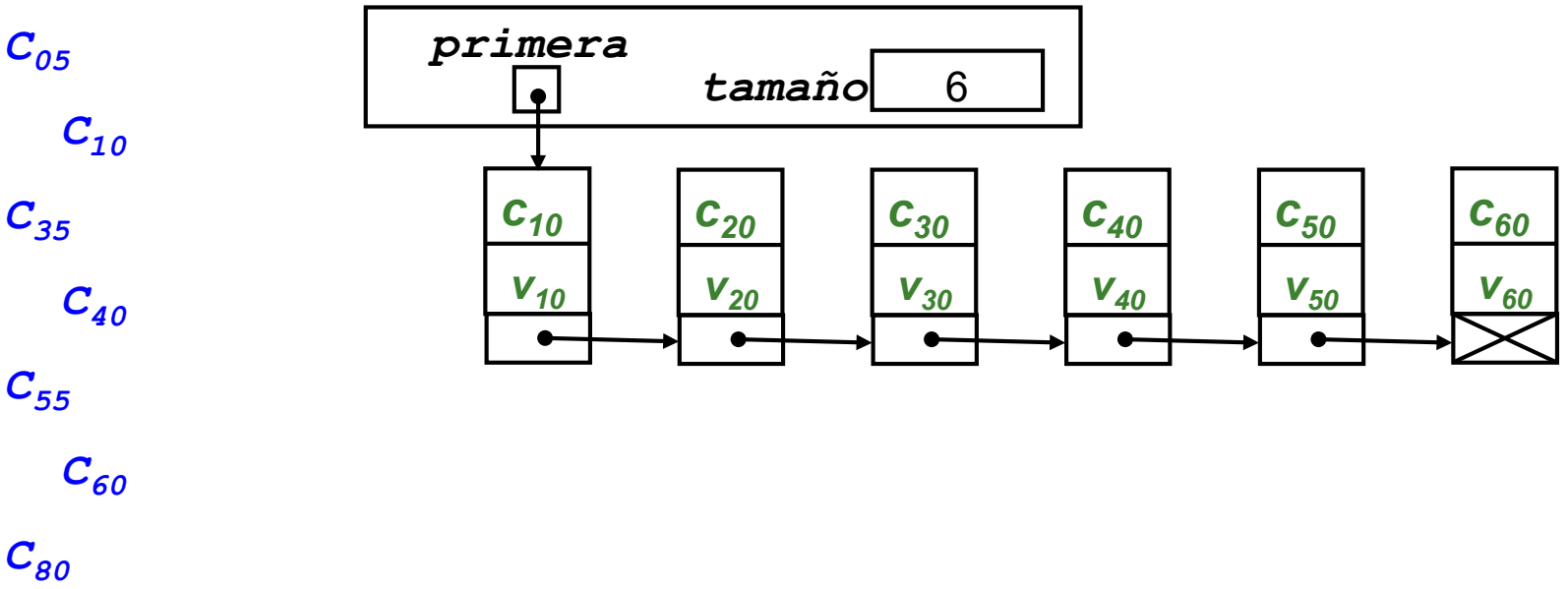
```
procedimiento buscar(ent d:diccionario; ent c:clave;
                     sal éxito:booleano; sal v:valor)
```

➢ *Diccionario vacío:*

{seguro que c no está en d}

➢ *Diccionario NO vacío:*

{buscar:…}



```
procedimiento buscar(ent d:diccionario; ent c:clave;
                     sal éxito:booleano; sal v:valor)
variable pAux:punteroCelda
principio
  pAux:=d.primera;
  mientrasQue pAux≠nil andthen pAux↑.laClave<c hacer
    pAux:=pAux↑.sig
  fmq;
  si pAux=nil entonces
    éxito:=falso
  sino
    si pAux↑.laClave=c entonces
      v:=pAux↑.elValor;
      éxito:=verdad
    sino
      éxito:=falso
    fsi
  fsi
fin
```

evaluación perezosa (o cortocircuitada), en C++: &&

$\Theta(N)$ en tiempo

Evaluación perezosa (o cortocircuitada):

Para evaluar:

(expresiónA AND expresiónB)

1º) se evalúa **expresiónA** y se obtiene su resultado booleano (llamémosle **A**)
2º) se evalúa **expresiónB** y se obtiene su resultado booleano (llamémosle **B**)
3º) por último, se evalúa **(A AND B)** obteniendo el resultado final

(expresiónA ANDTHEN expresiónB)

1º) se evalúa **expresiónA** y se obtiene su resultado booleano (llamémosle **A**)
2º) **Si A** es verdad, **entonces**
se evalúa **expresiónB** y se obtiene su resultado booleano (llamémosle **B**),
que es además el resultado final *{ verdad AND B= B}*
sino *{ A es falso}*
NO se evalúa **expresiónB** porque el resultado final es *falso* *(es decir, A) {falso AND B= falso}*

- *ANDTHEN* obtiene el resultado final de forma más eficiente que *AND*
- En `(pAux≠nil andthen pAux↑.laClave<c)` el ANDTHEN evita que se intente usar pAux con valor NIL para acceder al dato apuntado (pAux↑)

- De forma equivalente, existen el OR y el ORELSE (que evalúa su segundo operando solo si es necesario para decidir el resultado final)

procedimiento añadir(**e/s** d:diccionario; **ent** c:clave; **ent** v:valor)

{Si en d no hay ningún par con clave c, devuelve d con el par (c,v) añadido; si en d había un par (c,v'), entonces devuelve d actualizado con v como valor de c en vez de v'}

- ***Diccionario vacío:***

{seguro que c no está en d}

{añadir:…} C_{10}, V_{10}

- ***Diccionario NO vacío:***

{añadir:…}

C_{10}, Z_{10}

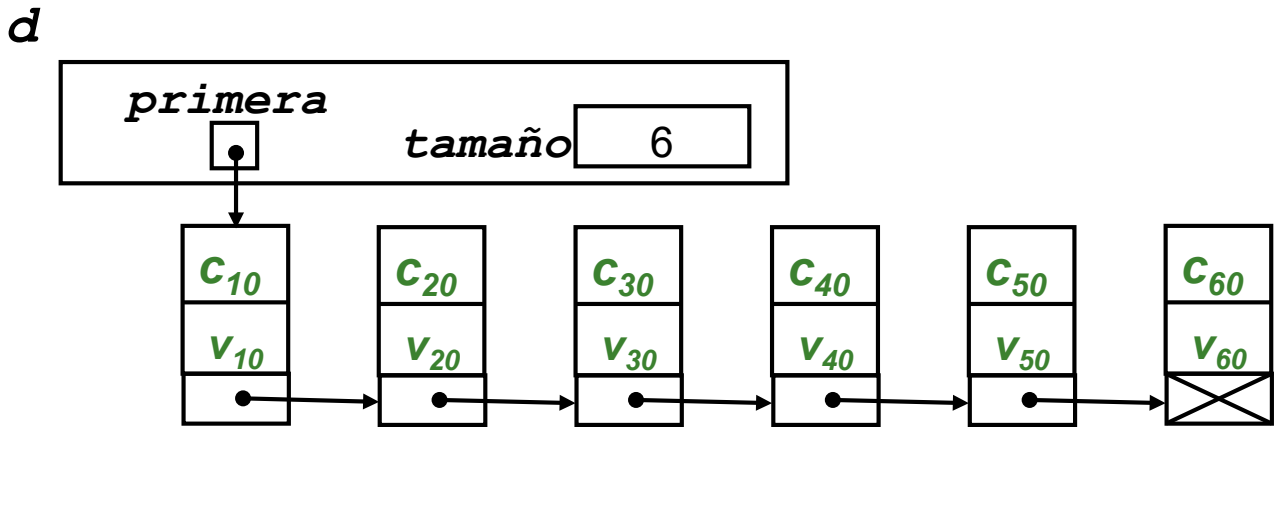
C_{40}, Z_{40}

C_{60}, Z_{60}

C_{05}, V_{05}

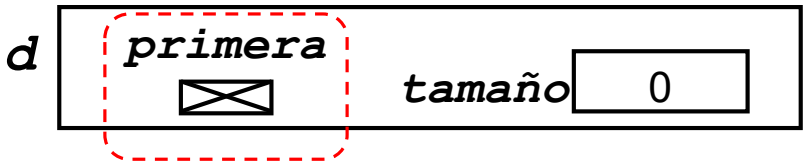
C_{35}, V_{35}

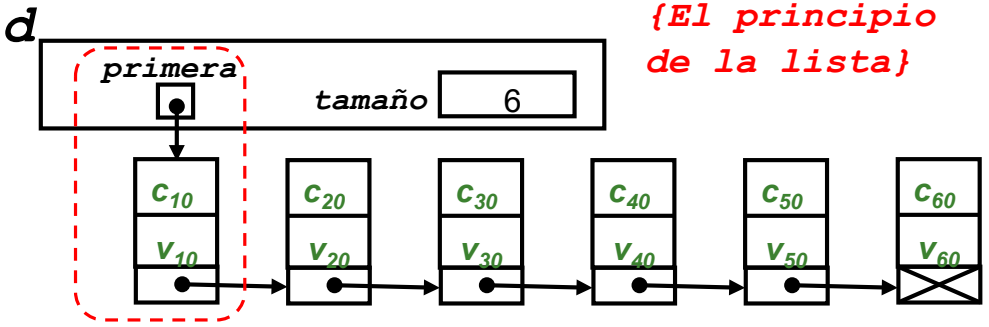
C_{70}, V_{70}



Implem. dinámica con listas ordenadas

```
procedimiento añadir(e/s d:diccionario; ent c:clave; ent v:valor)
variables pAux,nuevo:punteroCelda
principio
  si d.primera=nil entonces {caso: lista vacía}
    nuevoDato(d.primera);
    d.primera↑.laClave:=c;
    d.primera↑.elValor:=v;
    d.primera↑.sig:=nil;
    d.tamaño:=1
  sino {caso: lista NO vacía}
    si c<d.primera↑.laClave entonces {caso:inserción al principio}
      pAux:=d.primera;
      nuevoDato(d.primera);
      d.primera↑.laClave:=c;
      d.primera↑.elValor:=v;
      d.primera↑.sig:=pAux;
      d.tamaño:=d.tamaño+1
    sino
      si c=d.primera↑.laClave entonces{caso:ya existe, cambiar valor}
        d.primera↑.elValor:=v
      {sino: buscar punto de inserción…}
      ...
```

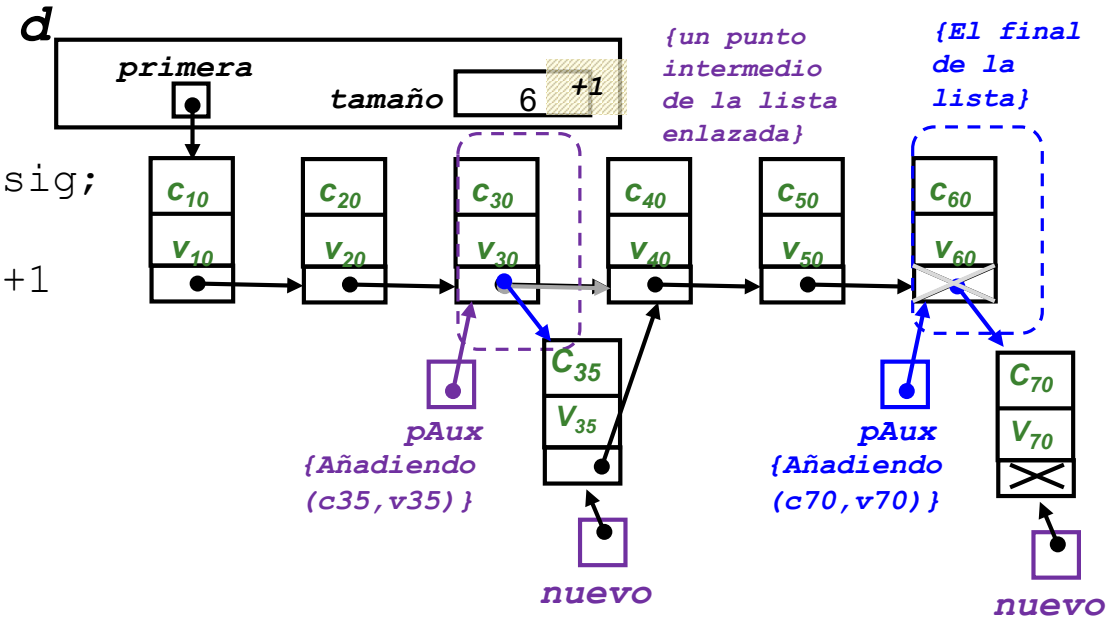




Implem. dinámica con listas ordenadas

```
    ...
    sino {d.primera↑.laClave < c → buscar punto de inserción}
      pAux:= d.primera;
      mq pAux↑.sig≠nil andthen (pAux↑.sig↑.laClave<c) hacer
        pAux:=pAux↑.sig
      fmq;
      si pAux↑.sig≠nil andthen c=pAux↑.sig↑.laClave entonces
        {clave ya existe, cambiar valor}
        pAux↑.sig↑.elValor:=v
      sino {casos: inserción entre dos registros o al final}
        nuevoDato(nuevo);
        nuevo↑.laClave:=c;
        nuevo↑.elValor:=v;
        nuevo↑.sig:=pAux↑.sig;
        pAux↑.sig:=nuevo;
        d.tamaño:=d.tamaño+1
      fsi
    fsi
  fsi
fsi
fin
```

evaluación perezosa (o cortocircuitada), en C++: &&



Θ(N) en tiempo

Implem. dinámica con listas ordenadas

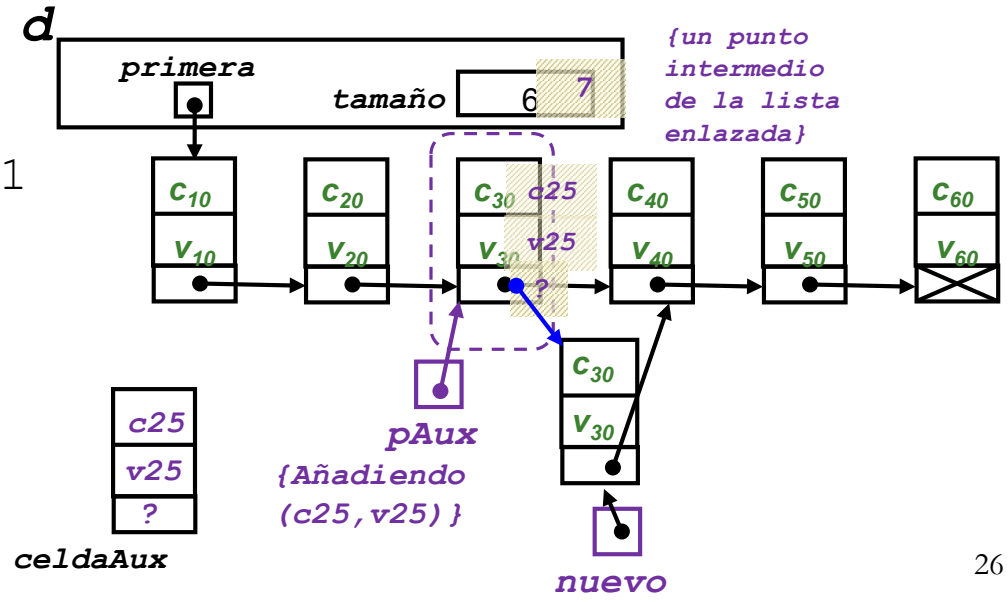
Otra solución diferente:

```
procedimiento añadir(e/s d:diccionario;ent c:clave;ent v:valor)
variables pAux,nuevo:punteroCelda; celdaAux:Celda;
principio
  si d.primera=nil entonces {caso: lista vacía}
    nuevoDato(d.primera);                          {igual que en la solución anterior}
    d.primera↑.laClave:=c;
    d.primera↑.elValor:=v;
    d.primera↑.sig:=nil;
    d.tamaño:=1;
  sino{caso: lista NO vacía}
    si c<d.primera↑.laClave entonces{caso: inserción al principio}
      pAux:= d.primera;
      nuevoDato(d.primera);                        {igual que en la solución anterior}
      d.primera↑.laClave:=c;
      d.primera↑.elValor:=v;
      d.primera↑.sig:=pAux;
      d.tamaño:=d.tamaño+1
   {sino: buscar punto de inserción}
   ...
```

Implem. dinámica con listas ordenadas

```
sino {buscar punto de inserción}
   pAux:= d.primera;
   mientrasQue pAux↑.laClave<c and pAux↑.sig≠nil hacer
      pAux:=pAux↑.sig;
   fmq;
   si c<pAux↑.laClave entonces{caso: inserción entre dos registros}
      celdaAux.laClave:=c;
      celdaAux.elValor:=v;
      nuevoDato(nuevo);
      nuevo↑:= pAux↑;
      pAux↑:= celdaAux;
      pAux↑.sig:=nuevo;
      d.tamaño:=d.tamaño+1
...
```

{Asignación de registros: copia campo a campo}

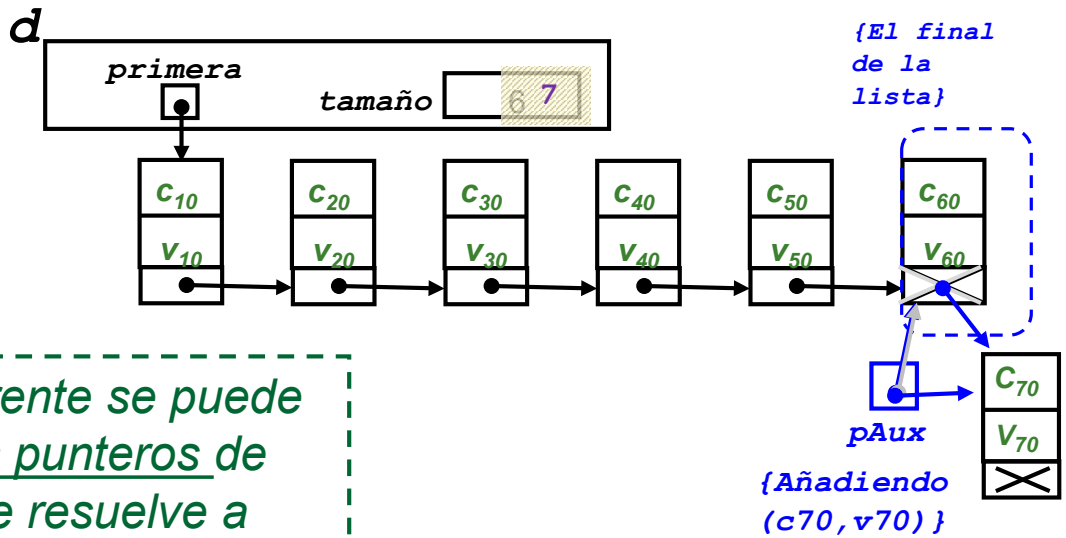


Esta solución no es habitual. Si es copia profunda es muy costosa

Implem. dinámica con listas ordenadas

```
      sino {No es el caso de inserción entre dos registros}
        si c=pAux↑.laClave entonces
          {caso: clave ya existe, cambiar valor}
          pAux↑.elValor:=v
        sino {caso: inserción al final(por ser pAux↑.sig=nil) }
          nuevoDato(pAux↑.sig);
          pAux:=pAux↑.sig;
          pAux↑.laClave:=c;
          pAux↑.elValor:=v;
          pAux↑.sig:=nil;
          d.tamaño:=d.tamaño+1
        fsi
      fsi
    fsi
  fsi
fin
```

$\Theta(N)$ en tiempo



Otra implementación diferente se puede conseguir utilizando dos punteros de forma similar a como se resuelve a continuación la operación quitar….

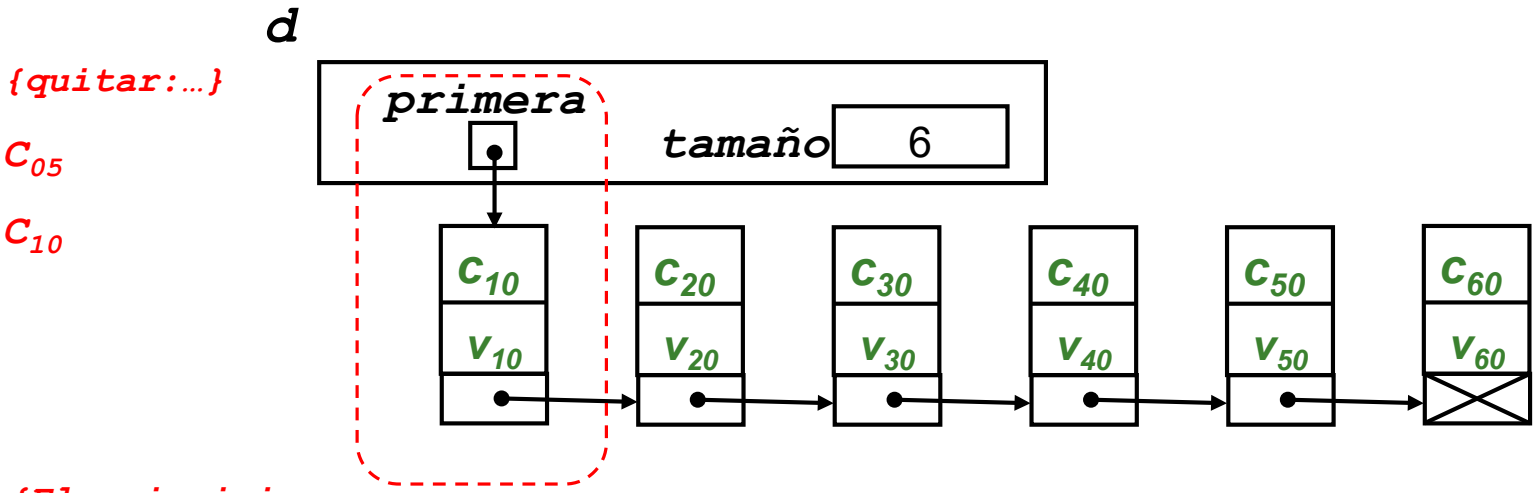
procedimiento quitar(**ent** c:clave; **e/s** d:diccionario)

{Si en d no hay ningún par con clave c, devuelve d sin modificar; si en d había un par (c,v) devuelve d actualizado, sin el par que tenía clave c}

➢ ***Diccionario vacío:***

{seguro que c no está en d}

➢ ***Diccionario NO vacío:***



{quitar:…}

C_{05}

C_{10}

{El principio de la lista}

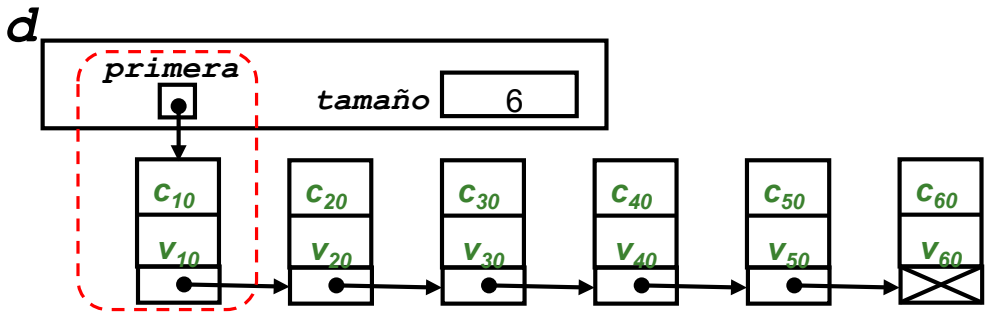
Implem. dinámica con listas ordenadas

Solución **eficiente**: tiene en cuenta el orden

```
procedimiento quitar(ent c:clave; e/s d:diccionario)
{Si en d no hay ningún par con clave c, devuelve d sin modificar; si en d había un par (c,v)
    devuelve d actualizado, sin el par que tenía clave c}
variables pAux1,pAux2:punteroCelda; parar:booleano
principio
  si d.primera≠nil entonces {caso contrario es vacío, y no hacer nada}
    si not(c<d.primera↑.laClave) entonces {caso contrario no está, y no hacer nada}
      si d.primera↑.laClave=c entonces {borrar el primer elemento}
        pAux1:=d.primera;
        d.primera:=d.primera↑.sig;
        disponer(pAux1)
        d.tamaño:=d.tamaño-1
      {sino d.primera↑.laClave<c
            => buscar la clave c a partir del 2° elemento}
      ...
{quitar:…}
```

C_{05}

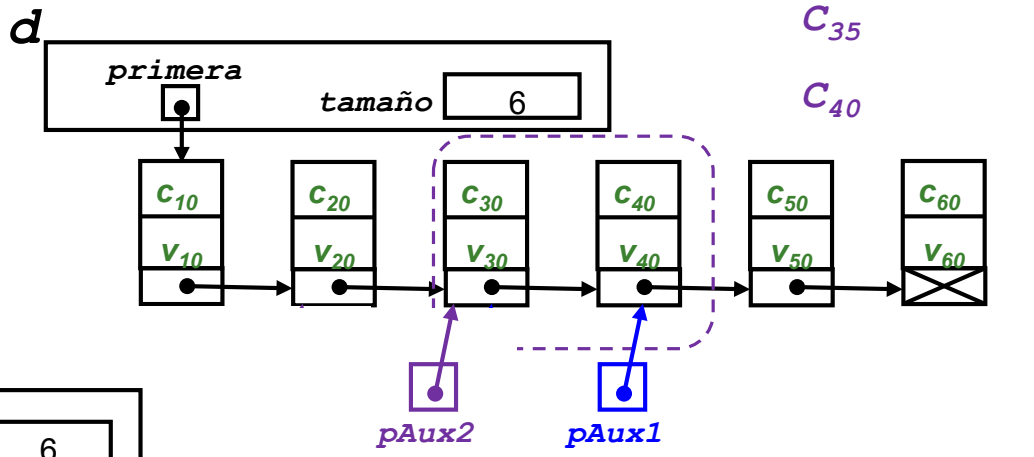
C_{10}

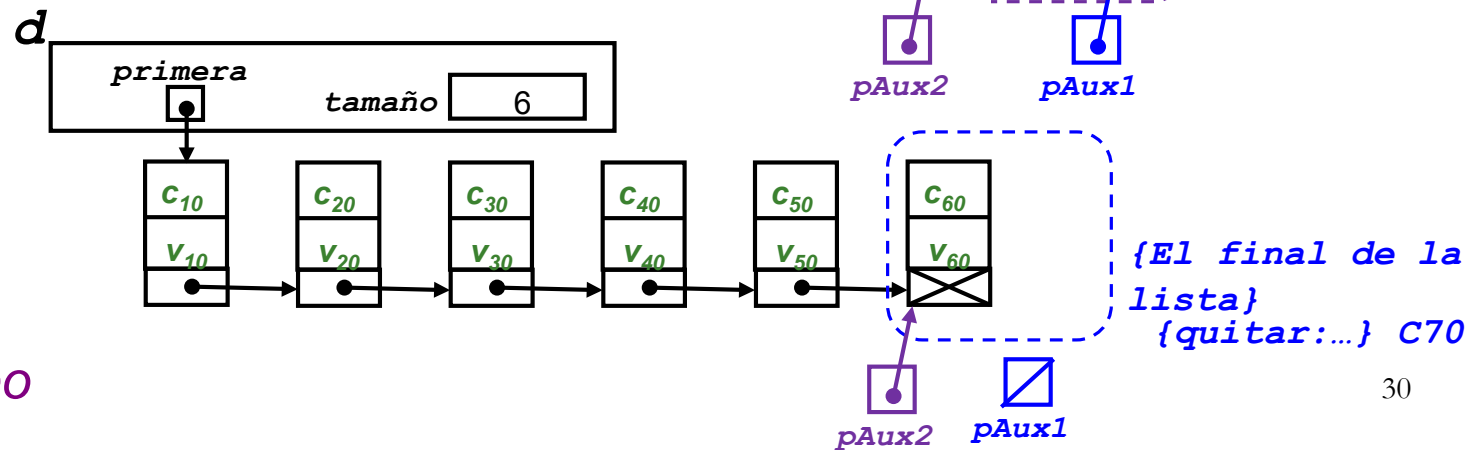


{El principio de la lista}

Implem. dinámica con listas ordenadas

```
    sino {buscar la clave c a partir del 2º elemento}
      parar:=falso;
      pAux1:=d.primera↑.sig; pAux2:=d.primera;
    - mientrasQue pAux1≠nil and not parar hacer
        si c<pAux1↑.laClave entonces {clave c no está, parar ya}
          parar:=verdad
        sino_si c=pAux1↑.laClave entonces {borrar el registro}
          pAux2↑.sig:=pAux1↑.sig;
          disponer(pAux1);
          parar:=verdad;
          d.tamaño:=d.tamaño-1
        sino
       {pAux1↑.laClave<c => avanzar}
          pAux2:=pAux1;
          pAux1:=pAux1↑.sig
        fsi
    - fmq
      fsi
    fsi
  fsi
fin
```





Θ(N) en tiempo

Implem. dinámica con listas ordenadas

```
función iguales?(d1,d2:diccionario) devuelve booleano
variables pAux1,pAux2:punteroCelda; igual:booleano
principio
  si esVacío?(d1) and esVacío?(d2) entonces
     devuelve verdad
  sino_si cardinal(d1)≠cardinal(d2) entonces
     devuelve falso
  sino {ambos tienen el mismo número (no nulo) de claves}
    igual:= verdad;
    pAux1:=d1.primera; pAux2:=d2.primera;
    mientrasQue igual and pAux1≠nil hacer
      igual:=(pAux1↑.laClave=pAux2↑.laClave) and
                  (pAux1↑.elValor=pAux2↑.elValor);
      pAux1:=pAux1↑.sig; pAux2:=pAux2↑.sig
    fmq;
    devuelve igual
  fsi
fin
```

Θ(N) en tiempo

Implem. dinámica con listas ordenadas

```
procedimiento duplicar(sal dSal:diccionario;
                       ent dEnt:diccionario)
variables pAuxEnt,pAuxSal:punteroCelda
principio
  si esVacío?(dEnt) entonces crear(dSal)
  sino {dEnt no vacío => tiene una primera celda}
    nuevoDato(dSal.primera); {copiar el primer par (clave, valor)}
    dSal.primera↑.laClave:=dEnt.primera↑.laClave;
    dSal.primera↑.elValor:=dEnt.primera↑.elValor;
    pAuxEnt:=dEnt.primera↑.sig; pAuxSal:=dSal.primera;
    mientrasQue pAuxEnt≠nil hacer {copiar el resto…}
      nuevoDato(pAuxSal↑.sig);
      pAuxSal:=pAuxSal↑.sig;
      pAuxSal↑.laClave:=pAuxEnt↑.laClave;
      pAuxSal↑.elValor:=pAuxEnt↑.elValor;
      pAuxEnt:=pAuxEnt↑.sig
    fmq;
    pAuxSal↑.sig:=nil;
    dSal.tamaño:=dEnt.tamaño
  fsi
fin
```

{copia profunda o deep copy}

Θ(N) en tiempo

Implem. dinámica con listas ordenadas

```
procedimiento liberar(e/s d:diccionario)
variable pAux:punteroCelda
principio
  pAux:=d.primera;
  mientrasQue pAux≠nil hacer
    d.primera:=d.primera↑.sig;
    disponer(pAux);
    pAux:=d.primera
  fmq;
  d.tamaño:=0 {y d.primera tiene valor nil en diccionario vacío…
               o utilizar: crear(d)}
fin
```

$\Theta(N)$ en tiempo

Implem. dinámica con listas ordenadas

Iterador interno:

requiere modificar la representación del TAD

```
diccionario = registro
                  primera,actual:punteroCelda;
                  tamaño:natural
              freg
```

diccionario

*Las demás operaciones del TAD **no** utilizarán el puntero actual (salvo para inicializarlo en crear)*

```
procedimiento iniciarIterador(e/s d:diccionario)
principio
  d.actual:=d.primera
fin
```

$\Theta(1)$ en tiempo

```
función existeSiguiente?(d:diccionario) devuelve booleano
principio
  devuelve d.actual≠nil
fin
```

$\Theta(1)$ en tiempo

Implem. dinámica con listas ordenadas

```
procedimiento siguienteYAvanza(e/s d:diccionario;
                               sal c:clave; sal v:valor;
                               sal error:booleano)
principio
  si existeSiguiente?(d) entonces
    error:=falso;
    c:=d.actual↑.laClave; v:=d.actual↑.elValor;
    d.actual:=d.actual↑.sig
  sino
    error:=verdad
  fsi
fin
```

$\Theta(1)$ en tiempo

En C++ (en un fichero *diccionario.hpp*)

En la parte pública NO DEBEN aparecer detalles de implementación: No aparecen nodos, ni celdas, ni punteros,….

```
// Interfaz del TAD. Pre-declaraciones:
template<typename K, typename V> struct Diccionario;
   //Los tipos que se vayan a usar en sustitución de K y V
   // tendrán que tener implementadas las operaciones:
   // … operator==… K … K   // o la operación que corresponda a la usada para comparar K's por igualdad
   // … operator< …….K…K   // o la operación que corresponda a la usada para ordenar K's
   // … operator== …V …V   // o la operación que corresponda a la usada para comparar V's por igualdad
template<typename K, typename V> void crear(Diccionario<K,V>& d);
template<typename K, typename V> void anyadir(Diccionario<K,V>& d, const K& k, const V& v);
template<typename K, typename V> void quitar(Diccionario<K,V>& d, const K& k);
template<typename K, typename V> bool buscar(const Diccionario<K,V>& d, const K& k, V& v);
template<typename K, typename V> int cardinal(const Diccionario<K,V>& d);
template<typename K, typename V> bool esVacio(const Diccionario<K,V>& d);
template<typename K, typename V>
        void duplicar(const Diccionario<K,V>& dEnt, Diccionario<K,V>& dSal);
template<typename K, typename V>
         bool operator==(const Diccionario<K,V>& d1, const Diccionario<K,V>& d2);
template<typename K, typename V> void liberar(Diccionario<K,V>& d);
template<typename K, typename V> void iniciarIterador(Diccionario<K,V>& d);
template<typename K, typename V> bool existeSiguiente(const Diccionario<K,V>& d);
template<typename K, typename V> bool siguienteYAvanza(Diccionario<K,V>& d, K& k, V& v);
 // sigue . . .
```

En C++ (en un fichero *diccionario.hpp*)

```
// Parte privada: Declaración de la representación interna
template<typename K, typename V>
struct Diccionario {
        friend void crear<K,V>(Diccionario<K,V>& d);
        friend void anyadir<K,V>(Diccionario<K,V>& d, const K& k, const V& v);
        friend void quitar<K,V>(Diccionario<K,V>& d, const K& k);
        friend bool buscar<K,V>(const Diccionario<K,V>& d, const K& k, V& v);
        friend int cardinal<K,V>(const Diccionario<K,V>& d);
        friend bool esVacio<K,V>(const Diccionario<K,V>& d);
        friend void duplicar<K,V>(const Diccionario<K,V>& dEnt, Diccionario<K,V>& dSal);
        friend bool operator==<K,V>(const Diccionario<K,V>& d1, const Diccionario<K,V>& d2);
        friend void liberar<K,V>(Diccionario<K,V>& d);
        friend void iniciarIterador<K,V>(Diccionario<K,V>& d);
        friend bool existeSiguiente<K,V>(const Diccionario<K,V>& d);
        friend bool siguienteYAvanza<K,V>(Diccionario<K,V>& d, K& k, V& v);
// Representación interna de los valores del TAD:
  private:
        ... // definición…

}; //fin definición del struct Diccionario<K,V>
```


// sigue . . .

En C++ (en un fichero *diccionario.hpp*)

```
// Parte privada: Implementación de las operaciones:

template<typename K, typename V>
void crear(Diccionario<K,V>& d) {
        ... // implementación …
}

... // etc etc implementación de las demás operaciones
```

Ejercicio:

- Intenta responder a las siguientes preguntas… →

¿Qué hace este código?

```
procedimiento candidato1 (sal dSal:diccionario; ent dEnt:diccionario)
variables pAux1,pAux2:punteroCelda
principio
  si esVacío?(dEnt) entonces crear(dSal)
  sino
     nuevoDato(dSal.primera);
     dSal.primera↑.laClave:=dEnt.primera↑.laClave;
     dSal.primera↑.elValor:=dEnt.primera↑.elValor;
     pAux1:=dEnt.primera↑.sig; pAux2:=dSal.primera;
     mientrasQue pAux1≠nil hacer
      pAux2:=pAux2↑.sig;
      nuevoDato(pAux2);
      pAux2↑.laClave:=pAux1↑.laClave;
      pAux2↑.elValor:=pAux1↑.elValor;
      pAux1:=pAux1↑.sig;
     fmq;
     pAux2↑.sig:=nil; dSal.tamaño:=dEnt.tamaño
  fsi
fin
```

¿Qué hace este código?

```
procedimiento candidato2 (sal dSal:diccionario; ent dEnt:diccionario)
variables pAux1,pAux2:punteroCelda
principio
  si esVacío?(dEnt) entonces crear(dSal)
  sino
     nuevoDato(dSal.primera);
     dSal.primera↑.laClave:=dEnt.primera↑.laClave;
     dSal.primera↑.elValor:=dEnt.primera↑.elValor;
     pAux1:=dEnt.primera↑.sig; pAux2:=dSal.primera↑.sig;
     mientrasQue pAux1≠nil hacer
      nuevoDato(pAux2);
      pAux2↑.laClave:=pAux1↑.laClave;
      pAux2↑.elValor:=pAux1↑.elValor;
      pAux1:=pAux1↑.sig; pAux2:=pAux2↑.sig;
     fmq;
     pAux2↑.sig:=nil; dSal.tamaño:=dEnt.tamaño
  fsi
fin
```

¿Qué hace este código?

```
procedimiento candidato3 (sal dSal:diccionario; ent dEnt:diccionario)
variables pAux1,pAux2:punteroCelda
principio
  si esVacío?(dEnt) entonces crear(dSal)
  sino
     pAux2:=dSal.primera;
     nuevoDato(pAux2);
     pAux2↑.laClave:=dEnt.primera↑.laClave;
     pAux2↑.elValor:=dEnt.primera↑.elValor;
     pAux1:=dEnt.primera↑.sig; pAux2:=pAux2↑.sig;
     mientrasQue pAux1≠nil hacer
      nuevoDato(pAux2);
      pAux2↑.laClave:=pAux1↑.laClave;
      pAux2↑.elValor:=pAux1↑.elValor;
      pAux1:=pAux1↑.sig; pAux2:=pAux2↑.sig;
     fmq;
     pAux2↑.sig:=nil; dSal.tamaño:=dEnt.tamaño
  fsi
fin
```

¿Cuál de los *candidatos* te parece mejor?