Tipus Recursius de Dades IV
Programació 2
Facultat d'Informàtica d'Informàtica, UPC

Conrado Martínez

Primavera 2019

- Apunts basats en els d'en Ricard Gavaldà
- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Part I

Implementacions amb accés a la representació

- Avantatge: Eficiència. Assignació d'apuntadors vs. còpia d'estructures
- Inconvenient: Lligades a una representació. No modulars
- Exemple: `sort` com a mètode de la classe `list` a STL

Cerca d'un element en una pila

```
class Pila {
  ...
/* Pre: cert */
/* Post: retorna cert ssi x apareix a la pila implícita */
bool cerca(const T &x) const;
  ...
};
```

Cerca en una pila: versió iterativa

```
/* Pre: cert */
/* Post: retorna cert ssi x apareix a la pila implícita */
bool cerca(const T &x) const {
    node_pila* act = cim;
    /* Inv: cap node entre [cim, act) té info = x */
    while (act != nullptr) {
        if (act -> info == x) return true;
        act = act -> seguent;
    }
    return false;
}
```

Cerca en una pila: versió recursiva I

```
class Pila {
  ...
/* Pre: cert */
/* Post: retorna cert ssi x apareix a la pila implícita */
bool cerca(const T &x) const;
  ...
};
```

Problema: La recursió és (node $\rightarrow$ node), no (pila $\rightarrow$ pila)!

Cerca en una pila: versió recursiva I

```
class Pila {
  ...
/* Pre: cert */
/* Post: retorna cert ssi x apareix a la pila implícita */
bool cerca(const T &x) const;
  ...
};
```

Problema: La recursió és (node $\rightarrow$ node), no (pila $\rightarrow$ pila)!

Immersió: $\rightarrow$ operació auxiliar, recursiva, amb paràmetre `node_pila*`

la crida inicial fa el pas (pila $\rightarrow$ `node_pila*`)

Cerca en una pila: versió recursiva II

```
  /* Pre: cert */
  /* Post: retorna cert ssi x apareix a la pila implícita */
bool cerca(const T &x) const {
    return cerca_pila_node(cim, x);
}


/* Pre: cert */
/* Post: retorna cert ssi x apareix a la llista
         de nodes que comença a n */
static bool cerca_pila_node(node_pila* n, const T &x);
```

Atenció a l'`static`!

Cerca en una pila: versió recursiva III

```
/* Pre: cert */
/* Post: retorna cert ssi x apareix a la llista
         de nodes que comença a n */
static bool cerca_pila_node(node_pila* n, const T &x) {
    if (n == nullptr) return false;
    else if (n -> info == x) return true;
    else return cerca_pila_node(n -> seg, x);
}
```

Compte: precondició de l'operador $->$

Sumar un valor a tots els elements d'un arbre binari

El plantegem com a nou mètode de la classe arbre binari

```
    ...
/* Pre: A és el valor inicial del 'arbre implícit */
/* Post: l'arbre implícit és l'arbre A però havent sumat k
         a tots els seus elements */
void inc_arbre(const T& k);
    ...
```

Sumar un valor a tots els elements d'un arbre binari

```
/* Pre: A és el valor inicial del arbre implícit */
/* Post: l'arbre implícit és l'arbre A però havent sumat k
         a tots els seus elements */
void inc_arbre(const T& k) {
    inc_node(a.arrel, k);
}


/* Pre: cert */
/* Post: el node apuntat per n i tots els seus descendents tenen
         al camp info la suma de k i el seu valor original */
static void inc_node(node_arbre* n, int k) {
    if (n != nullptr) {
        n -> info += k;
        inc_node(n -> esq, k);
        inc_node(n -> dre, k);
    }
}
```

Substitució de fulles per un arbre I

Substituir totes les fulles de l'arbre implícit que continguin el valor `x` per un altre arbre donat `as`

```
/* Pre: A es el valor inicial del p.i. */
/* Post: l'arbre és com A però havent substituït
       les fulles que contenien x per l'arbre as */
void subst(int x, const ArbreBin<T>& as);
```

Substitució de fulles per un arbre II

```
void subst(int x, const ArbreBin<T>& as) {
    arrel = subst_node(arrel, x, as);
}


/* Pre: n apunta a l'arrel d'un (sub)arbre A */
/* Post: el(l') (sub)arbre binari l'arrel del qual apunta n és el
resultat de substituir cada fulla d'A que contingui el valor x
per una còpia de l'arbre as */
static node_arbre* subst_node(node_arbre* n, int x,
                              const ArbreBin<T>& as);
```

Substitució de fulles per un arbre III

```
static node_arbre* subst_node(node_arbre* n, int x,
                              const ArbreBin<T>& as) {
    if (n == nullptr) return nullptr;

    // n != nullptr
    if (n -> info == x and
        n -> esq == nullptr and n -> dre == nullptr) {
      // n apunta a una fulla que conté el valor x
      delete n; // no cal fer esborra_node_arbre(n);
      n = copia_node_arbre(as.arrel);
    } else {
      n -> esq = subst_node(n -> esq, x, as);
      n -> dre = subst_node(n -> dre, x, as);
    }
    return n;
}
```

Atenció al retorn de l'apuntador a l'arrel de l'arbre resultant.
L'alternativa és passar `n` per referència

Revessar una llista

```
/* Pre: cert */
/* Post:la llista conté els mateixos elements que a l'inici però
        amb l'ordre invertit; el seu punt d'interés apunta
        al mateix element que a l'inici */
void revessar();
```

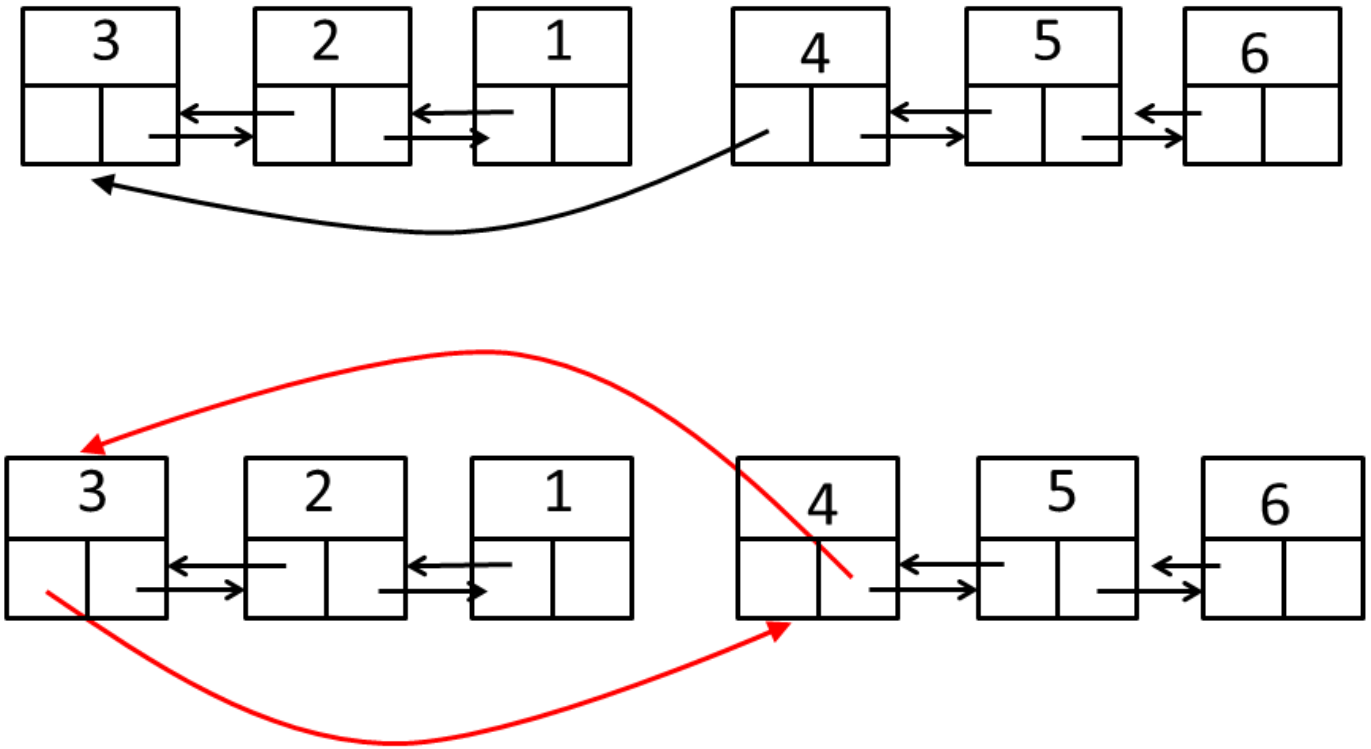
Revessar una llista

```
/* Pre: cert */
/* Post:la llista conté els mateixos elements que a l'inici però
        amb l'ordre invertit; el seu punt d'interés apunta
        al mateix element que a l'inici */
void revessar();
```

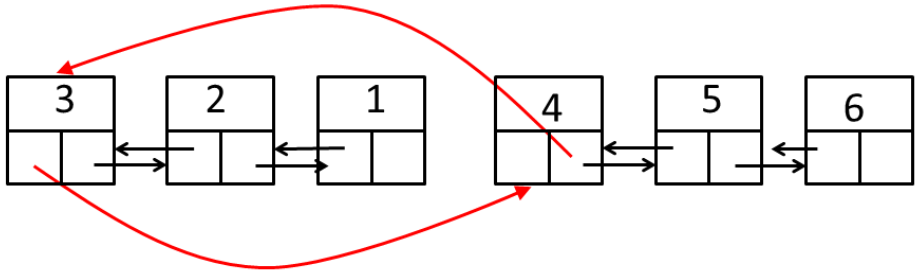
1. Solució amb ops. de la classe: `insert`, còpies de node ...
2. Solució tocant representació: assignacions d'apuntadors

Revessar una llista, v1

Simular “esborrar el primer de l1, afegir-lo primer a l2”

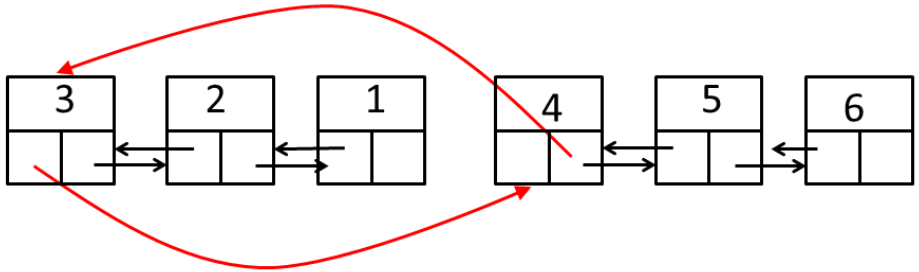


Revessar una llista, v1



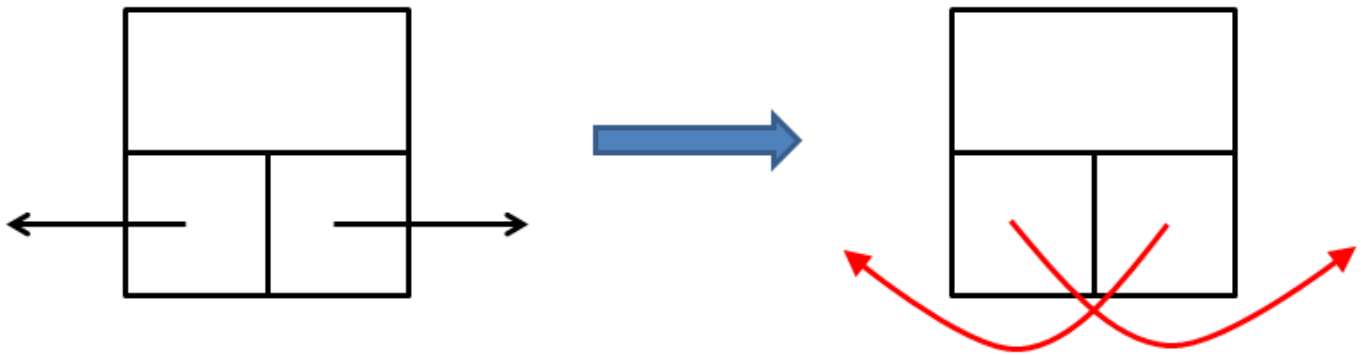

```
void revessar() {
    node_llista* n = primer;
    while (n != ultim) {
      /* Inv: per tots els nodes anteriors al que apunta n,
         els apuntadors a anterior i següent han estat
         intercanviats respecte a l'original */
         node_llista* suc = n -> seg;
         n -> seg = n -> ant;
         n -> ant -> ant = n;
         n = suc;
    }
    ...  // continua
}
```

Revessar una llista, v1

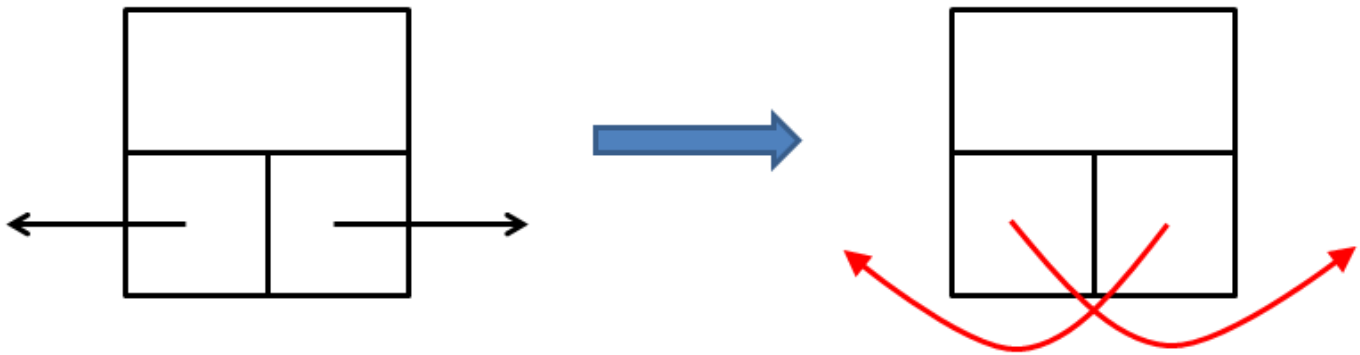



```
   ...
if (n != nullptr and n != primer) {
  // n == ultim != primer (i la llista
  // no és buida!)
      n -> seg = n->ant;
      n -> ant = nullptr;
      ultim = primer;
      primer = n;
  }
```

Revessar una llista

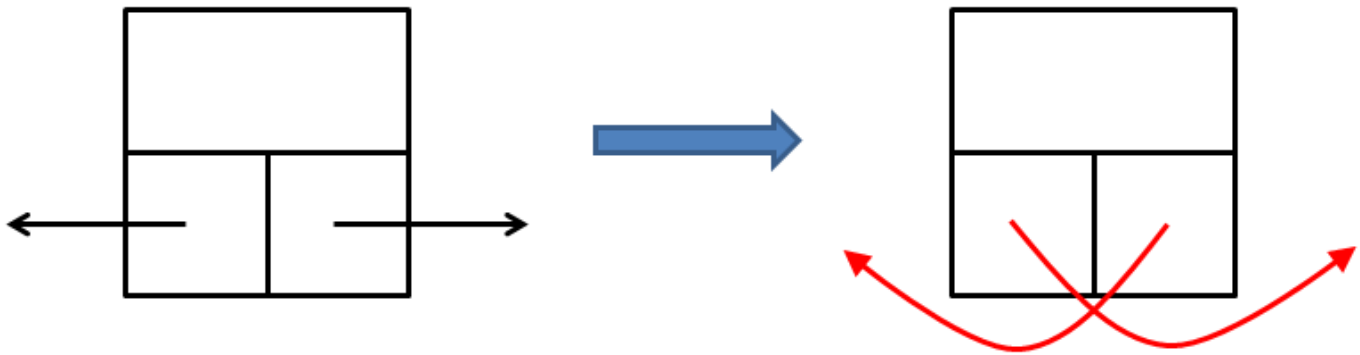


Revessar una llista



```
void revessar() {
    node_llista* n = primer;
    while (n != nullptr) {
      /* Inv: per als nodes anteriors al que apunta n,
         els apuntadors a anterior i seguent han estat
         intercanviats respecte a l'original */
        swap(n -> seg, n -> ant);
        n = n -> ant;
    }
    swap(primer, ultim);
}
```

Revessar una llista



```
void revessar() {
    node_llista* n = primer;
    while (n != nullptr) {
      /* Inv: per als nodes anteriors al que apunta n,
         els apuntadors a anterior i seguent han estat
         intercanviats respecte a l'original */
        swap(n -> seg, n -> ant);
        n = n -> ant;
    }
    swap(primer, ultim);
}
```

Part I

Part I

Cues ordenades

- Modificació de la classe `Cua`: propietat addicional de poder ser recorregudes en ordre creixent respecte al valor dels seus elements
- Dos tipus d'ordre: cronològic (com fins ara) + per valor (nou)
- Cal que hi hagi un operador $<$ definit en el tipus o classe dels elements
- Cal redefinir la implementació amb més apuntadors

Apuntadors:

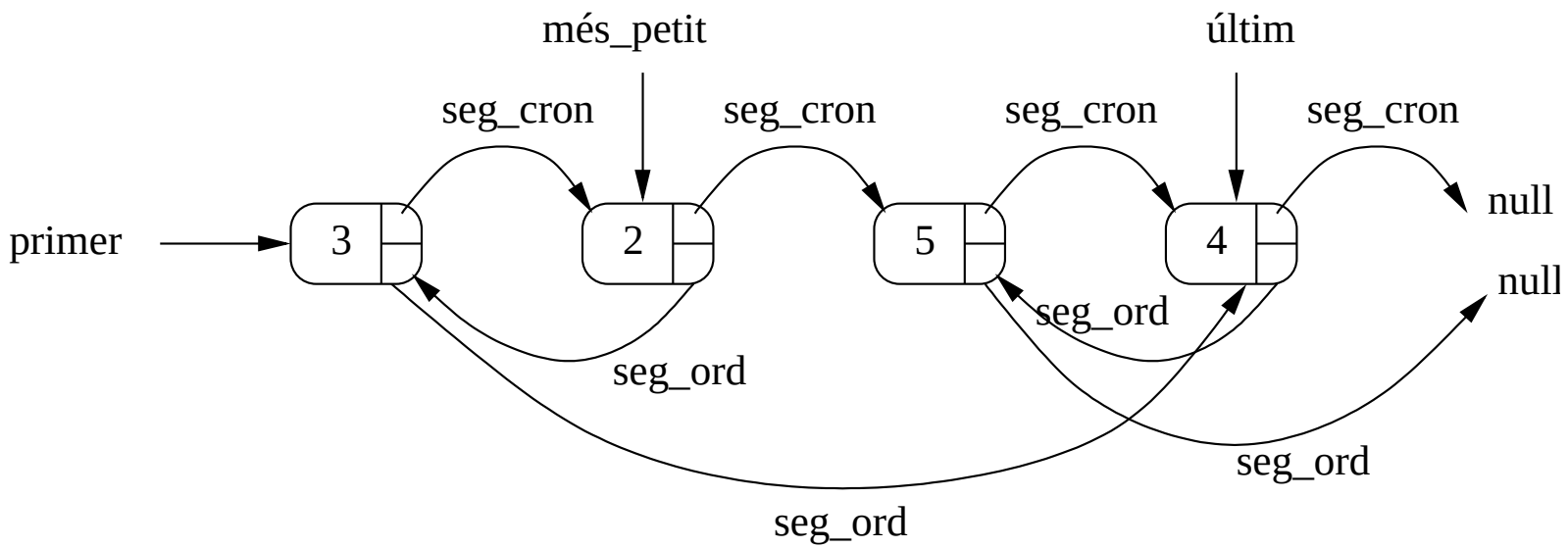
- `primer`, `ultim` i `seg` per gestionar l'ordre d'arribada a la cua (**ordre estàndar de cua, cronològic**).
- `mes_petit` i `seg_ord` per gestionar l'ordre creixent segons el valor dels elements.

Nova definició de la classe

```
template <class T> class CuaOrd {
  private:
    struct node_cuaOrd {
        T info;
        node_cuaOrd* seg_ord;
        node_cuaOrd* seg;
    };
    int longitud;
    node_cuaOrd* primer;
    node_cuaOrd* ultim;
    node_cuaOrd* mes_petit;
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
};
```

Esquema de la implementació

Exemple:



Implementació cues ordenades

- Veurem només dues operacions públiques: `demanar_torn` (`push`) i `concatenar`
- Exercici: especificació i implementació d'altres operacions que caldria incloure

Demanar torn (`push`) I

```
void demanar_torn(const T& x) {
/* Pre: cert */
/* Post: la CuaOrd implícita conté x com a darrer
         element per ordre cronològic i on li pertoca
         en ordre creixent */
...
}
```

Demanar torn (`push`) II

```
void demanar_torn(const T& x) {
  node_cuaOrd*  n = new node_cuaOrd;
  n -> info = x;
  n -> seg = nullptr;
  if (primer == nullptr) {
    primer = ultim = n;
    mes_petit = n;
    n -> seg_ord = nullptr;
  } else {
    ...
  }
  ++longitud;
}
```

Demanar torn (`push`) III

```
} else {
 // la cua conté altres elements
 // (primer != nullptr => mes_petit != nullptr)
 // 1. el nou node és l'últim en ordre cronològic
 ultim -> seg = n;
 ultim = n;
 // 2. ara inserim el nou node ón pertoca en
 // ordre creixent
 mes_petit = inserta_ord(mes_petit, n);
}
```

Demanar torn (`push`) III

```
// Pre: la cadena que comença a p seguint els apuntadors seg\_ord
// està en ordre creixent de valor, n != nullptr
// Post: retorna un apuntador al primer de la cadena resultant
// d'inserir el node apuntat per n en ordre creixent a la cadena
// que comença a p
static node_cuaOrd* inserta_ord(node_cuaOrd* p, node_cuaOrd* n) {
  if (p == nullptr) return n;
  if (n -> info < p -> info) {
     n -> seg_ord = p;
     return n;
  } else {
     p -> seg_ord = inserta_ord(p -> seg_ord, n)
     return p;
  }
}
```

Concatenar I

```
void concatenar(CuaOrd& c2) {
/* Pre: la cuaOrd implícita és C1, c2 = C2 */
/* Post: la cuaOrd implícita representa la concatenació de C1
i C2 en el ordre cronològic (és a dir, tot element de C2 vé
després de qualsevol element de C1 en ordre cronològic); la cuaOrd
implícita també representa la fusió de C1 i C2 en el ordre
creixent; finalment c2 queda buida */
```

Concatenar II

```
void concatenar(CuaOrd &c2) {
  if (c2.primer == nullptr) return;
  // només caldrà fer alguna cosa si c2 no és buida
  if (primer == nullptr) {
    // si el la cuaOrd implícita és buida, llavors
    // li transferim els continguts de c2
      primer = c2.primer;
      ultim = c2.ultim;
      mes_petit = c2.mes_petit;
  } else {  ...  }
  // la cuaOrd implícita augmenta la seva
  // longitud en tants elements com tenia c2
  longitud += c2.longitud;
  // i buidem la cuaOrd c2
  c2.primer = c2.ultim = c2.mes_petit = nullptr;
  c2.longitud = 0;
}
```

Concatenar III

```
{ // ni la cuaOrd ni c2 són buides
  // connectem la cuaOrd i c2
  // pero orde cronològic
  ultim -> seg = c2.primer; // amb el primer de c2
  ultim = c2.ultim_node;    // i actualitzem l'últim

  // ara fem la fusió dels nodes de les dues cues segon
  // l'ordre creixent;
  mes_petit = fusiona(mes_petit, c2.mes_petit);
}
```

Concatenar IV

```
static node_cuaOrd* fusiona(node_cuaOrd* n1, node_cuaOrd* n2) {
  if (n1 == nullptr) return n2;
  if (n2 == nullptr) return n1;
  // n1 != nullptr and n2 != nullptr
  if (n1 -> info <= n2 -> info) {
    n1 -> seg_ord = fusiona(n1 -> seg_ord, n2);
    return n1;
  } else {
    n2 -> seg_ord = fusiona(n1, n2 -> seg_ord);
    return n2;
  }
}
```

Part I

Multillistes: Motivació

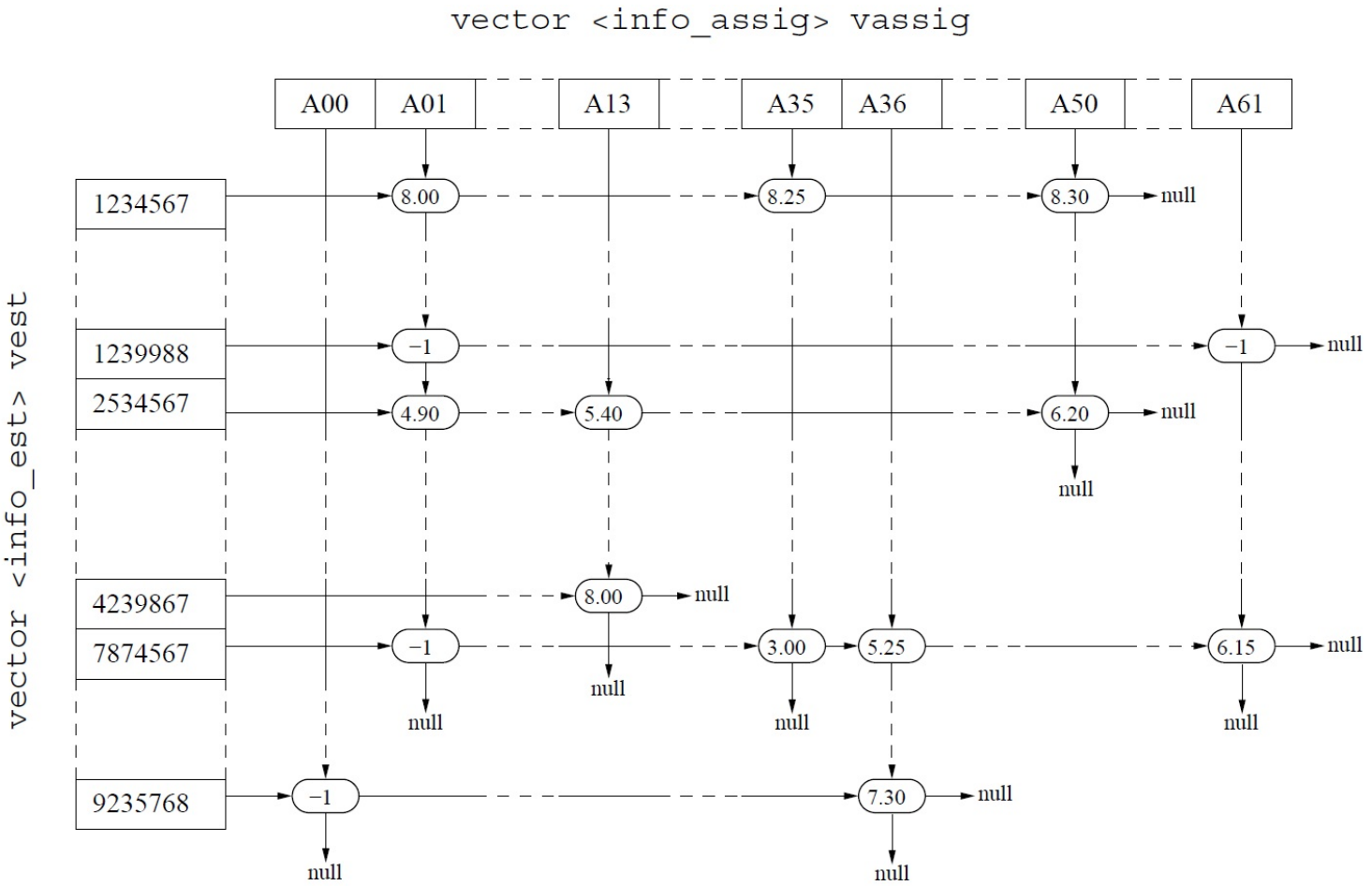
Volem guardar una taula molt gran però molt *esparsa*: molts elements nuls
Necessitem:

- Donat un índex de fila, recuperar tots els elements no nuls de la fila
- Donat un índex de columna, recuperar tots els elements no nuls de la columna

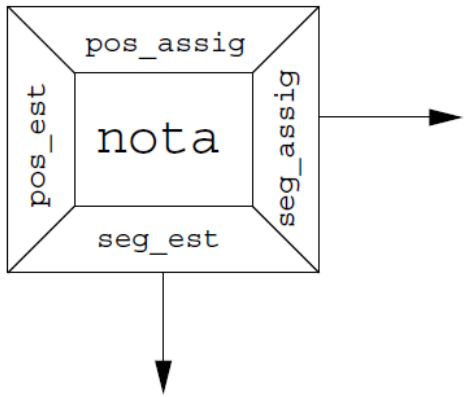
Exemple: Taula per guardar *cursos de la FIB*:

> "l'estudiant X estava matriculat a Y i ha tret nota Z"

Multillistes: Esquema



Multillistes: Node



Implementació i detalls: $\rightarrow$ apunts