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

Professorat de PRO2

Tardor 2022

- Col·laboracions (en ordre alfabètic): Juan Luis Esteban, Ricard Gavaldà, Conrado Martínez, Fernando Orejas
- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Part I

Ocupació dels nivells d'un arbre N-ari

Donat un arbre N-ari a volem un nou mètode que determini quants nodes té l'arbre implícit a cada un dels seus nivells: concretament voldrem que retorni un vector v de talla $h =$ alçària de l'arbre ímplicit i tal que $v[i] =$ nombre de nodes de l'arbre implícit en el nivell $i, 0 \leq i < h$.

```
template <class T>
class ArbreNari {
  ...
// Pre: cert
// Post: v.size() = alçària de l'arbre implícit i
// v[i] = nombre de nodes en el nivell i de l'arbre implícit,
// 0 ≤ i < v.size()
void ocupacio_nivells(vector<int>& v) const;
 ...
}
```

Ocupació dels nivells d'un arbre N-ari

Per a dissenyar una implementació eficient del mètode `ocupacio_nivells` introduirem un mètode privat de classe (`static`) amb una immersió d'especificació—un punter explícit p a l'arrel del subarbre considerat—i d'eficiència:

1. el vector v per tornar el resultat final es passa com a paràmetre per referència
2. un enter i que representa la profunditat del node apuntat per p

Per alçària(p) ens referirem a l'alçària del subarbre arrelat al node apuntat per p; $N_k(p)$ denotarà el nombre de nodes de nivell $k \geq 0$ en el subarbre arrelat a p

Ocupació dels nivells d'un arbre N-ari

```
class ArbreNari {
    ...
private:
// Pre: v = V, 0 ≤ i ≤ v.size()
// Post: per a tota k, 0 ≤ k < alçària(p),
//       v[i + k] = V[i + k] + N_k(p) si i + k < V.size(),
//       i v[i + k] = N_k(p) si i + k ≥ V.size()
//       v.size() = max{V.size(), i + alçària(p)}
static void i_ocupacio_nivells(node_arbreNari* p, int i,
       vector<int>& v);
       ...
};

template <class T>
void ArbreNari<T>::ocupacio_nivells(vector<int>& v) const {
  i_ocupacio_nivells(root, 0, v);
}
```

Ocupació dels nivells d'un arbre N-ari

```
static void i_ocupacio_nivells(node_arbreNari* p, int i,
       vector<int>& v) {
  if (p != nullptr) {
    if (i == v.size()) v.push_back(1);
    else ++v[i];
    for (int d = 0; d < child.size(); ++d)
       i_ocupacio_nivells(p -> child[d], i+1, v);
  }
}
```

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:

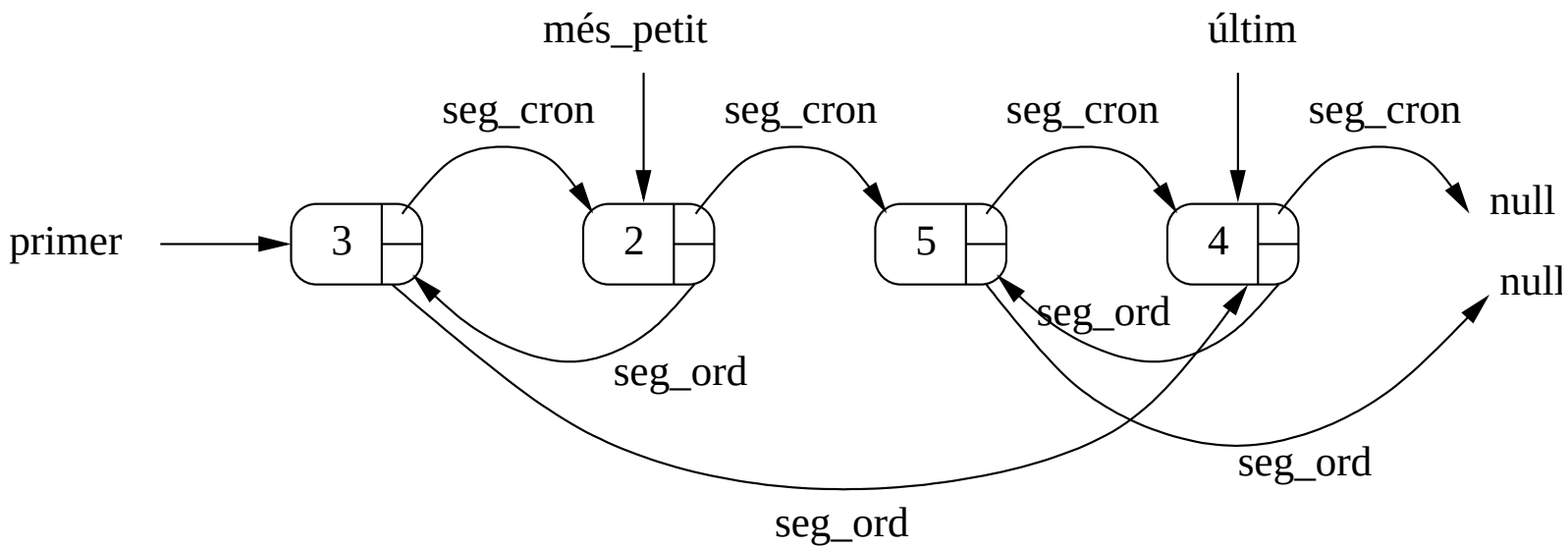
- `primer`, `ultim` i `seg` per gestionar l'ordre d'arribada a la cua (**ordre estàndar de cua, cronològic**).
- `mes_petit` i `seg_ord` per gestionar l'ordre creixent segons el valor dels elements.

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`
- En els apunts estan implementades iterativament

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

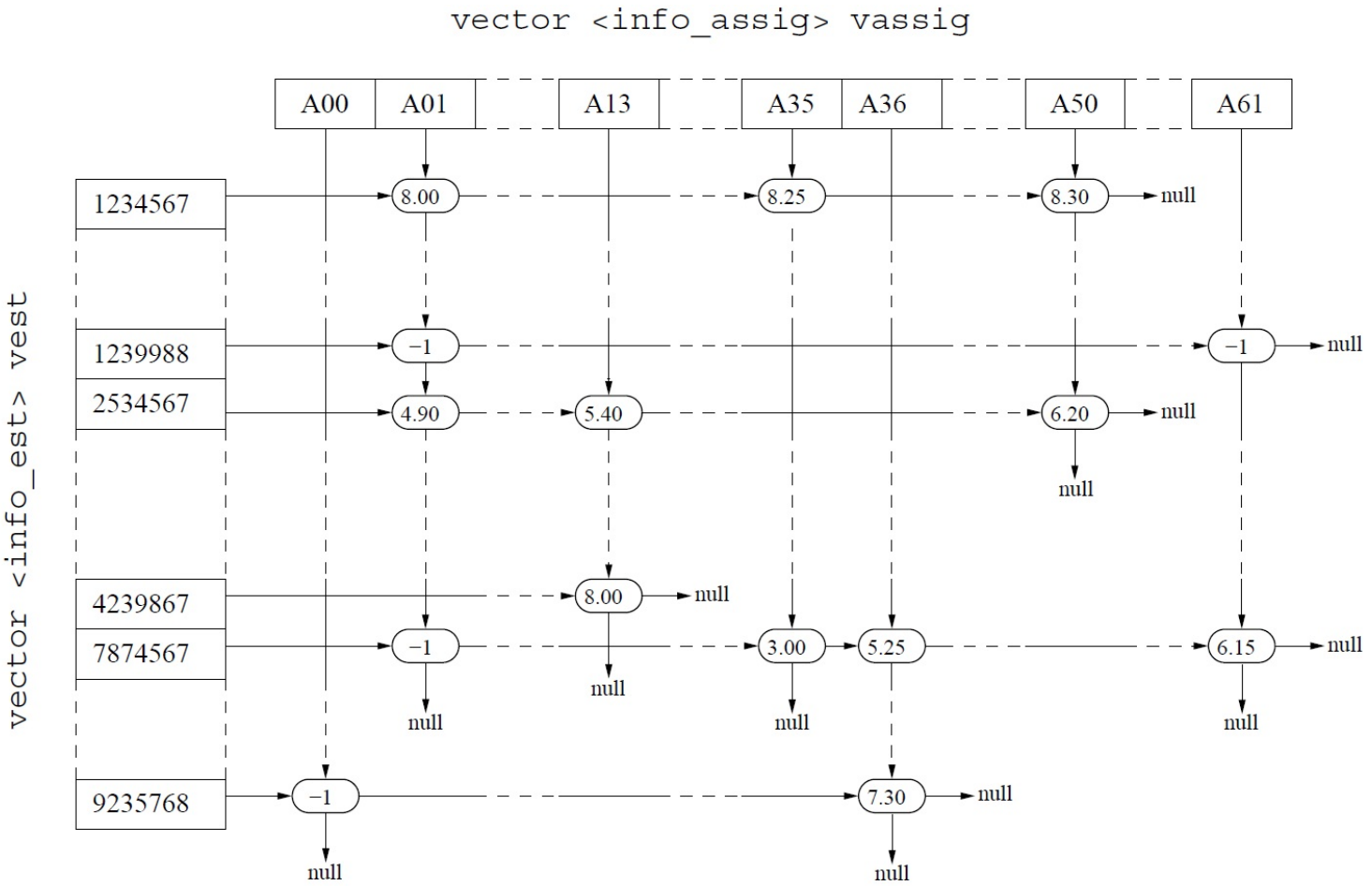
Volem guardar una taula molt gran però molt *esparsa*: molts elements nuls
Necessitem:

- 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

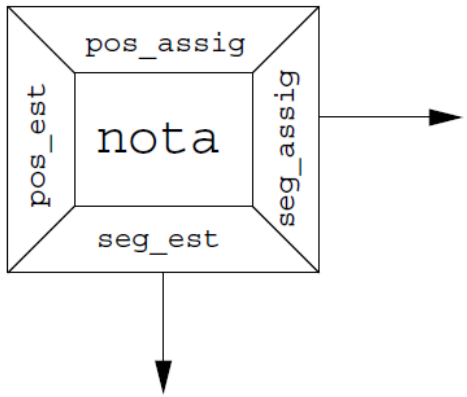
Exemple: Taula per guardar *cursos de la FIB*:

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

Multillistes: Esquema



Multillistes: Node



Implementació i detalls: $\rightarrow$ apunts