Segon examen parcial PRO2 Durada: 2h 30m 9/1/2017

Problema 1 (5 punts)

Donada una cua de parells d'enters on el primer element de cada parell és l'identificador d'un usuari i el segon element és el temps estimat de la gestió que vol realitzar, heu d'implementar una operació que reparteixi de manera justa els usuaris de la cua original en dues cues: l'original i una de nova. Les dues cues que en resultin han de satisfer les propietats següents:

1. Cap usuari ha d'esperar més en la cua a la qual ha estat assignat que un altre usuari que estigués situat darrere seu en la cua original.
2. Tots els usuaris han d'esperar el mínim temps possible.
3. Si un usuari pot estar assignat a ambdues cues satisfent les propietats anteriors, serà assignat a la cua original.

A continuació donem la implementació del tipus de dades `Cua` amb una petita modificació que permet simplificar la implementació de l'operació de distribució de cues de parells d'enters, és a dir, de *(usuari, temps)*.

```
class Cua {
  private:
    struct node {
        int usuari;
        int temps;
        node* seguent;
    };
    int longitud;
    node* primer;
    node* ultim;
};
```

Concretament, heu d'implementar l'acció `distribucio` especificada a continuació. No es poden utilitzar les operacions públiques de la classe `Cua`.

```
void distribucio (Cua& c);
// Pre: c = C i la cua paràmetre implícit són buides.
// Post: Els elements de C estan distribuïts de manera justa entre
// les cues c i paràmetre implícit.
```

Donem tot seguit un exemple de distribució justa de cues. Representem els elements de la cua com a parells d'enters, on el primer element del parell és l'identificador d'un usuari i el segon és el temps estimat de la gestió que vol realitzar. La cua `q1` abans de realitzar la crida a l'operació `distribucio` és:

$$(3,2) \leftarrow (6,3) \leftarrow (2,5) \leftarrow (11,1) \leftarrow (8,4) \leftarrow (5,3) \leftarrow (9,2) \leftarrow (1,3) \leftarrow (7,4) \leftarrow (15,2) \leftarrow (4,3)$$

Si `q2` és una variable de tipus `Cua` que conté una cua buida del tipus descrit anteriorment i fem la crida `q2.distribucio(q1)`, el valor de `q1` després de la crida serà

$$(3,2) \leftarrow (2,5) \leftarrow (5,3) \leftarrow (1,3) \leftarrow (15,2)$$

mentre que el valor de `q2` després de la crida serà

$$(6,3) \leftarrow (11,1) \leftarrow (8,4) \leftarrow (9,2) \leftarrow (7,4) \leftarrow (4,3)$$

Problema 2 (5 punts)

Volem utilitzar *arbres generals* de `string` per representar expressions aritmètiques del llenguatge de programació Lisp. Les expressions que considerem només poden contenir signes corresponents a nombres enters, als operadors binaris $+, -, *, /, <, ==$, a l'operador unari `abs` (que retorna el valor absolut d'un enter) i a la instrucció de control `if`. L'operador `/` representa la divisió entera. Recordeu que n/d està definit per a qualsevol parell d'enters n, d tals que $d \neq 0$. La instrucció de control `if` té tres arguments; la seva sintaxi és `(if exp_bool inst_1 inst_2)` i s'interpreta de la manera següent: si el resultat d'avaluar `exp_bool` és cert, s'avalua `inst_1` i se'n retorna el resultat; altrament, s'avalua `inst_2` i se'n retorna el resultat. Si el resultat d'avaluar `exp_bool` no és un valor booleà (`1` o `0`), es considera indefinit.

Un *arbre d'expressió* és un tipus particular d'*arbre general* instanciat per a valors de tipus `string` que permet representar expressions aritmètiques. Si volem definir una variable `a` que emmagatzemi un *arbre d'expressió*, l'hem de declarar mitjançant la instrucció `ArbreGen<string> a;`. Concretament, un *arbre d'expressió* és un *arbre general* instanciat per a valors de tipus `string` que satisfà la definició següent:

1. L'arbre d'expressió associat a l'expressió buida és l'arbre buit.
2. L'arbre d'expressió associat a l'expressió formada per un `string` e que representa un nombre enter és un arbre amb arrel e i sense fills.
3. L'arbre d'expressió associat a una expressió de la forma `(op_n arg_1 ...arg_n)` és un arbre amb arrel `op_n` i amb `n` fills, on el fill i-èssim és l'arbre d'expressió associat a l'argument i-èssim `arg_i`.

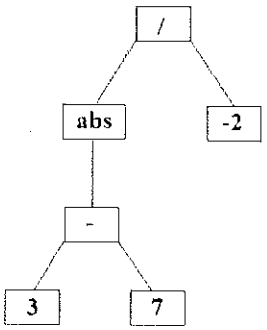
Donem tot seguit la implementació de la classe `ArbreGen`.

```
template <class T> class ArbreGen {

private:

  struct node {
    T info;
    vector<node*> seg;
  };

  node* primer_node;

    ... // especificació i implementació d'operacions privades

public:
    ... // especificació i implementació d'operacions públiques
};
```

Una expressió aritmètica representada mitjançant un arbre d'expressió pot quedar indefinida perquè:

- conté operadors o instruccions no contemplats en la definició del primer paràgraf de l'enunciat d'aquest problema,
- un operador o instrucció que requereix n arguments s'aplica a un nombre d'arguments diferent de n, o bé
- un operador o instrucció se aplica a arguments per als quals no està definit.

Per exemple, l'expressió `(/ (abs (- 3 7)) -2)` es pot representar fent servir l'arbre d'expressió següent:



Feu un disseny recursiu de l'acció `avalua` que provi d'avaluar l'expressió aritmètica representada pel paràmetre implícit (p.i.) i, en cas que aquesta expressió aritmètica estigui definida, retorni el seu resultat a través del paràmetre de sortida `result`.

void *avalua*(**bool**& *def*, **int**& *result*) **const**;
// Pre: El p.i. és un arbre d'expressió que representa una expressió aritmètica
// ben parentitzada i no buida.
// Post: Si l'expressió aritmètica associada al p.i. està definida, assigna el valor
// `true` al paràmetre `def` i el resultat d'avaluar el p.i. al paràmetre `result`;
// altrament, assigna el valor `false` al paràmetre `def`.

En la implementació d'`avalua` podeu fer servir (sense haver d'incloure la seva definició) l'acció següent de conversió de `string` a `int` (nombre enter), que està definida en C++ amb un altre nom.

void *string_a_enter* (**const** *string*& *s*, **bool**& *es_enter*, **int**& *result*);
// Pre: cert
// Post: Si `s` representa un nombre enter, s'assigna `true` al paràmetre de sortida
// `es_enter` i el nombre enter que representa `s` al paràmetre de sortida `result`;
// altrament, s'assigna `false` al paràmetre de sortida `es_enter`.

Escriviu la capçalera, precondició, postcondició i implementació de l'acció d'immersió. Implementeu l'acció `avalua` utilitzant l'acció d'immersió. No es poden fer servir operacions públiques de la classe `ArbreGen`. Es valorarà l'eficiència de la solució.

Per exemple, si *a* és una variable de tipus `ArbreGen` instanciat en `string` que representa l'expressió aritmètica `(/ (abs (- 3 7)) -2)` mitjançant un arbre d'expressió, la crida

a.avalua(*def*, *res*);

assignará a la variable booleana `def` el valor `true` i a la variable entera `res` el valor -2.

De la mateixa manera, si *b* és una variable de tipus `ArbreGen` instanciat en `string` que representa l'expressió aritmètica `(+ (/ 2 (* 0 1)) 7)` mitjançant un arbre d'expressió, la crida

b.avalua(*def*, *res*);

assignarà a la variable `def` el valor `false` i no assignarà cap valor a la variable entera `res`.

Segon examen parcial PRO2 Duració: 2h30m 11/1/2016

Cognoms Nom DNI

OBSERVACIÓ: Cal fer servir els espais indicats per entrar la resposta. Penseu bé la vostra solució abans de començar a escriure-hi. Podeu ser penalitzats fins a 1 punt si heu de demanar un nou full d'examen perquè us heu equivocat, o si la solució és bruta o si ocupa espai important fora de les caixetes.

Noteu que algunes de les caixetes per a codi poden deixar-se en blanc si creieu que no cal cap instrucció en aquell punt.

Exercici 1 - Concatenació per nivells de llistes (6 punts)

Tenim un vector `v` de `Llista<T>`, on `T` és un tipus qualsevol. Volem construir una mètode de la classe que construeixi una sola llista que contingui primer tots els primers elements de les llistes, després tots els segons elements de les llistes, després tots els tercers elements de les llistes, etc.

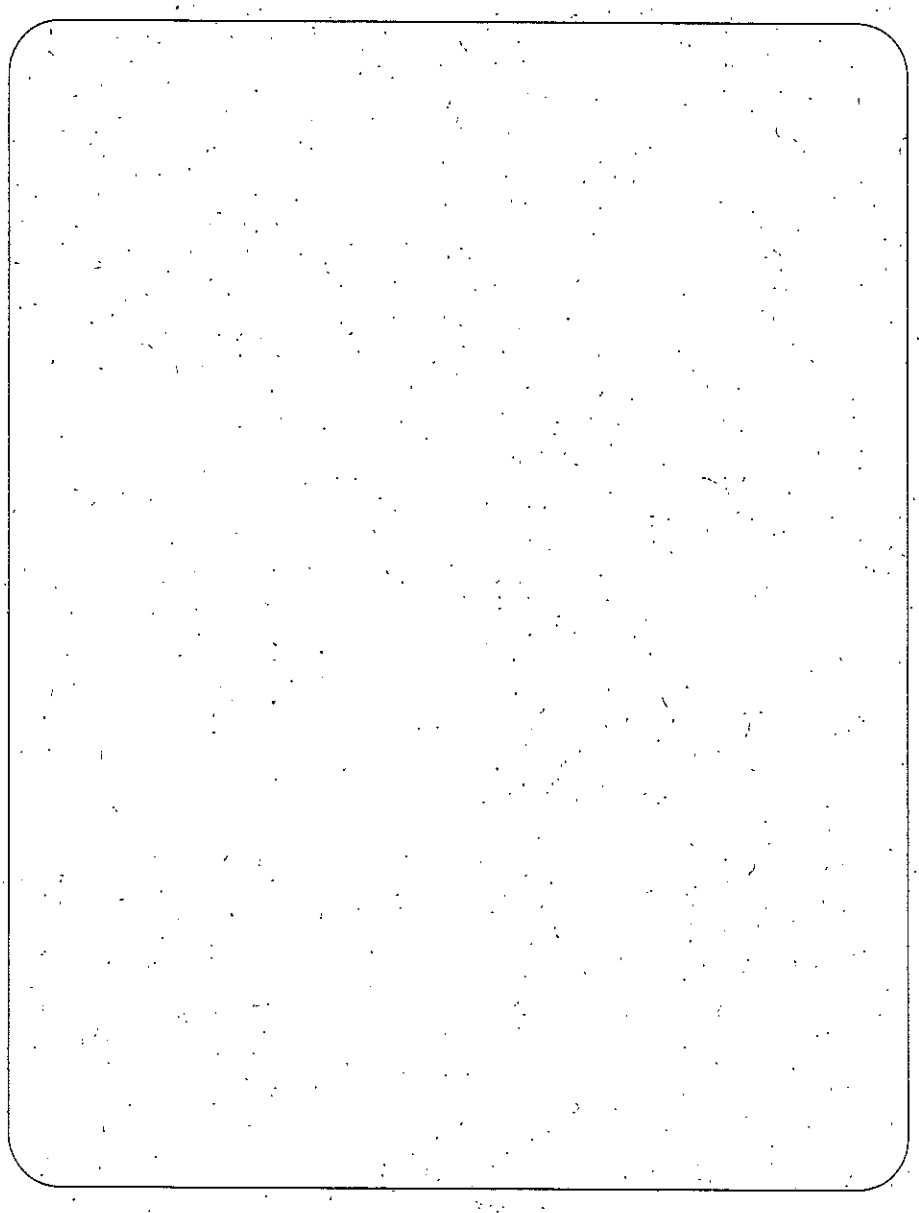
P.ex. si `v` té mida 5 i `v[0]=(a,b,c)`, `v[1]=(d,e,f,g)`, `v[2]=(h)`, `v[3]` és buida i `v[4] = (x,y,z)`, al final la llista destí ha de contenir `(a,d,h,x,b,e,y,c,f,z,g)` i `v` ha de contenir només llistes buides.

(a) (3 punts) Doneu una implementació d'un nou mètode de la classe `Llista` anomenat `transferir`. Per exemple, si el paràmetre implícit és `(a,b,c)` i `dest` és `(x,y,z)`, després de la crida el paràmetre implícit ha de ser `(b,c)` i `dest` ha de ser `(x,y,z,a)`. Cal implementar l'operació accedint directament a la representació privada de la classe i no es pot usar cap operació pública de la classe.

Recordem que la representació privada del tipus `Llista<T>` és

```
struct node_llista {
    T info;
    node_llista * seg;
    node_llista * ant;
};

int longitud;
node_llista * primer_node;
node_llista * ultim_node;
node_llista * act;
```

```
void Llista<T>:: transferir ( Llista<T>& dest)
/* Pre: el p.i. té un primer element x seguit d'una llista L; dest = D;
   el p.i. i dest són objectes diferents */
/* Post: el p.i. conté L; dest conté D seguida de x;
   el punt d'interès del p.i. no canvia si era damunt de L
   i si era damunt de x ara és damunt el primer element de L
   (que pot ser l'element fictici );
   el punt d'interès de dest no es modifica */
{
```



```
}
```

Cognoms Nom DNI

(b) (1 punt) Completeu el codi de la funció `concat_per_nivells` perquè satisfaci la seva especificació i respecti l'invariant donat. L'operació 1) és pròpia de la classe `Llista<T>`, 2) s'ha de programar accedint a la seva representació, 3) s'ha de basar en el mètode `transferir` de l'apartat (a) i 4) no pot fer servir cap altre mètode públic o privat de la classe.

```
void Llista<T>::concat_per_nivells (vector<Llista<T> >& v)
/* Pre: v = V, alguna llista de V no és buida, el p.i. és buit i és
        un objecte diferent de totes les llistes contingudes a v */
/* Post: v conté llistes buides i el p.i. conté la concatenació per nivells de
   les llistes de V */
{
    bool alguna_no_buida = true;
    while (alguna_no_buida) {
        /* Invariant: el p.i. conté, per a alguna k, k≥0, la concatenació
           dels k primers nivells de les llistes contingudes a V;
           aquests k primers nivells han estat esborrats de les llistes de v;
           alguna_no_buida és cert si i només si en aquest moment
           hi ha alguna llista no buida en v */
        ____________________________________
        for (int i = 0; i < v.size(); ++i) {
            ____________________________________
            ____________________________________
            ____________________________________
            ____________________________________
            ____________________________________
            ____________________________________
            ____________________________________
        }
        ____________________________________
    }
}
```

(c) (2 punts) La solució donada té l'inconvenient que farà un nombre d'operacions proporcional a `v.size()` $*$ (longitud de la llista més llarga continguda a `v`). Això és innecessàriament ineficient si, per exemple, `v` conté una sola llista molt llarga i moltes de curtes. Descriviu alguna solució més eficient que la proposada, *sense donar codi* però de forma prou precisa perquè quedi clara la implementació. Fem notar que n'hi ha una que té cost proporcional a `v.size()` + (suma de les longituds de les llistes contingudes à `v`). És millor que doneu alguna millora, encara que no sigui aquesta solució tan òptima, que no pas que deixeu l'apartat en blanc.

Podeu fer servir estructures auxiliars com ara piles, cues o llistes d'int's, però **no** estructures addicionals que continguin elements de tipus `T`, ni en general crear, copiar, o modificar elements de tipus `T`.

Cognoms Nom DNI

Exercici 2 - Comptar nodes iguals al pare en arbre n-ari (4 punts)

Llegiu-vos l'exercici complet abans de començar a resoldre'l.

Recordem primer que la representació privada del tipus `ArbreNari<T>` és

```
struct node_arbreNari {
    T info;
    vector<node_arbreNari*> seg;
};

int N;
node_arbreNari* primer_node;
```

Sigui `T` un tipus amb una operació d'igualtat == i considereu el problema següent: Donat un arbre n-ari, es vol saber quants nodes de l'arbre tenen un valor igual que el del seu pare. Per resoldre'l, volem implementar la funció següent dins de la classe `ArbreNari<T>`, accedint directament a la representació privada de la classe i sense usar cap operació pública de la classe.

```
int iguals () const
 /* Pre: cert */
 /* Post: el resultat és el nombre de nodes de l'arbre n-ari del paràmetre
 implícit que tenen el mateix valor que el seu pare */
{
}
```

Per tal que produeixi el resultat desitjat, cal disenyar i usar una funció auxiliar recursiva que heu de completar també (a la pàgina següent). Ompliu amb codi els forats deixats en blanc en les dues funcions, així com els destinats a la Pre i la Post de la funció auxiliar.

```
static int rec_iguals (                                    )
/* Pre:                                                    */
/* Post:
                                                           */
{
    int r = 0;
    int s =
    for (int i = 0; i < s; ++i) {
    }
    return r;
}
```

Final Programació 2

Enginyeria Informàtica
Juny 2015
Temps estimat: 2h 45min

- Feu bona lletra.
- Lliureu els dos problemes (en full separats, encara que siguin en blanc).
- En tots els problemes està prohibit usar mètodes públics de la classe; s'han de resoldre accedint directament a la implementació.
- Els codis no s'han de justificar, però poden incloure comentaris aclaridors si ho creieu convenient. Si feu servir operacions auxiliars, és necessari que les especifiqueu.
- Es valorarà l'eficiència de les solucions.

Problema 1 (4 punts)

Considerem la representació habitual amb nodes de la classe `Cua` per manegar cues genèriques d'elements de tipus `T`

```
template <class T> class Cua {
  private:
   struct node_cua {
     T info;
     node_cua* seg;
   };
   int longitud;
   node_cua* prim;
   node_cua* ult;
   ... // especificació i implementació d'operacions privades
  public:
   ... // especificació i implementació d'operacions públiques
};
```

Els nodes són simplement encadenats amb punters al següent (`seg`). Una cua té tres atributs: la longitud i dos punters a nodes, un al primer element (`prim`, el més antic) i un per l'últim (`ult`, el més recent).

Volem implementar dins de la classe `Cua` una operació que extregui la subcua més llarga entre dues aparicions d'un element donat (sense cap altra aparició entre elles, i en cas d'empat, la més recent) amb la següent especificació:

```
void subcua_mes_llarga(const T& elem, Cua& sub)
/* Pre: el p.i. conté una cua C, sub és buida */
/* Post: sub és una cua amb els elements de la subcua més llarga entre dues
         aparicions d'elem a C; el p.i. és com C però sense aquesta subcua .
*/
```

Exemples:

1. Si el p.i. és originalment:

   `2 1 2 3 4 1 2 3 4 5 1 2 1 2 3 4`

   i elem és `1`, el p.i. quedaria:

`2 1 2 3 4 1 1 2 1 2 3 4`

i sub contindria la cua:

`2 3 4 5`

2. Si el p.i. és originalment:

   `2 3 4 1 5 6`

   i elem és 1, el p.i. quedaria igual i sub seria la cua buida.

3. Si el p.i. és originalment:

   `1 1 1`

   i elem és 1, el p.i. quedaria igual i sub seria la cua buida.

Es demana una implementació d'aquesta operació que accedeixi directament als atributs de la classe `Cua`.

Problema 2 (7 punts)

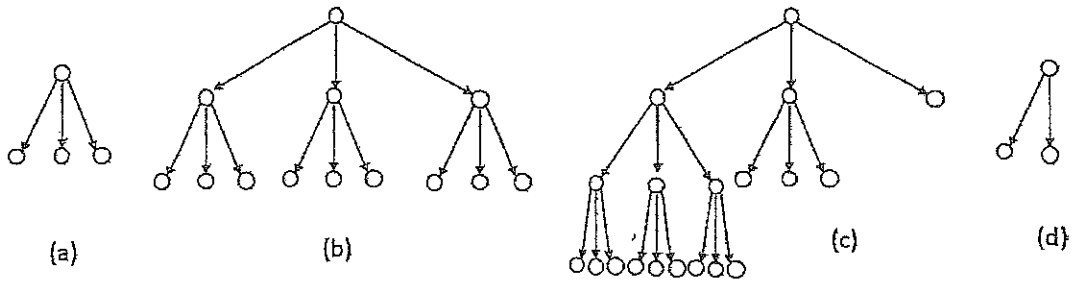
Considerem la classe que representa arbres N-aris, implementada com s'ha explicat a classe:

```
template <class T> class ArbreNari {
  private:
   struct node {
     T info;
     vector<node*> seg;
   };
   int N;   // nombre de fills de cada subarbre
   node* primer;
   ... // especificació i implementació d'operacions privades
  public:
   ... // especificació i implementació d'operacions públiques
};
```

Recordeu que en un arbre N-ari tots els nodes tenen exactament N fills, que poden ser buits, cosa que es tradueix en la representació en què tots els vectors seg tenen com a mida l'atribut N de l'arbre i que algunes de les seves components poden ser `NULL`.

Diem que un arbre N-ari d'altura h és *complet* si conté tots els nodes que pot contenir un arbre N-ari d'altura h. Fixeu-vos que, recursivament, tots els fills d'un arbre complet d'altura h són arbres complets d'altura $h-1$. L'arbre buit també és complet.

Així, dels quatre arbres 3-aris següents, els arbres (a) i (b) són complets; (c) no és complet, malgrat que els seus tres fills ho són, i (d) tampoc no és complet perquè el seu tercer fill és buit. (En aquest dibuix s'ometen els arbres buits).



Atenció: els dos apartats 2.1 i 2.2 són independents. No suggerim usar la funció feta a 2.1 per solucionar 2.2, i no es pot usar a 2.1 la funció `altura` que s'esmenta a 2.2.

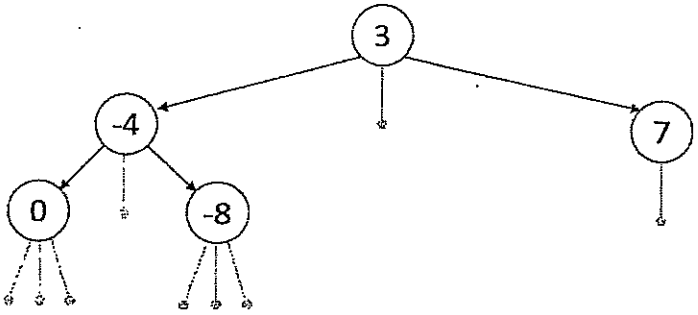
Apartat 2.1) [4 punts] Feu una funció pròpia de la classe `ArbreNari<T>`

```
bool es_complet() const;
```

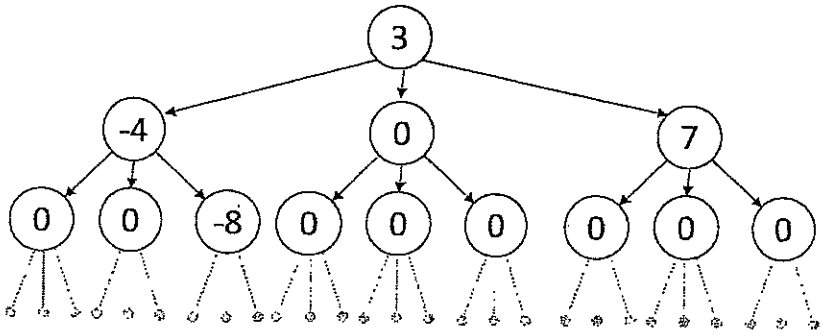
que digui si el seu paràmetre implícit és complet o no.

Apartat 2.2) [3 punts] Volem fer un mètode `completa(const T& v)` de la classe `ArbreNari<T>` que afegeixi el mínim de nodes al seu paràmetre implícit per fer-lo complet. En particular, si ja és complet el deixa intacte. Altrament, tots els nodes que calgui afegir han de contenir el valor `v`.

Per exemple, si `T` és `int`, la crida `completa(0)` aplicada a l'arbre



hauria de transformar-lo en l'arbre



S'ha decidit implementar aquesta funció així:

```
// Pre: el p.i. és un arbre N-ari A
// Post: el p.i. és l'arbre complet que completa A amb el mínim nombre de nodes;
//       els nodes que són al p.i. però no eren a A contenen el valor v
void completa(const T& v) {
    int h = altura(primer);
    i_completa(primer,N,h,v);
}
```

on `static int altura(node* q)` és una funció privada que retorna l'altura de la jerarquia de nodes que penja de q i que se suposa ja implementada. `i_completa` té l'especificació següent:

```
// Pre: h >= altura(p), n es l'aritat de l'ArbreNari<T> al que pertany
// la jerarquia apuntada per p
// Post: la jerarquia apuntada per p ha estat completada a un arbre
// complet d'altura h, afegint els nodes necessaris amb v al camp info
static void i_completa(node*& p, int n, int h, const T& v);
```

Doneu una implementació eficient d'`i_completa`.

Fixeu-vos en el & del paràmetre p, i en què p bé podria ser `NULL` inicialment.

Examen final, Programació 2

Enginyeria Informàtica
Gener 2015
Temps estimat: 2h 30m

- Feu bona lletra.
- Lliureu els dos problemes (en full separats, encara que siguin en blanc).

Problema 1 (5 punts)

Considerem la implementació amb nodes encadenats del tipus `stack<T>`, amb un apuntador al node cim. La recordem a continuació:

```
template <class T> class stack {
   private:
       struct node {
         T info;
         node* seg;
       };

       int altura;
       node* primer;

       ... // especificació d'operacions privades

   public:
       ... // especificació d'operacions públiques
};
```

Suposem que `T` disposa d'una operació d'ordre $<$ (i la seva variant $\leq$). Es demana implementar una operació pròpia de la classe que ordeni el seu paràmetre implícit, de manera que l'element més petit quedi al cim, i tot altre element sigui $\geq$ que el que té damunt.

La implementació no pot usar cap operació pública de la classe, ni pot crear noves piles, ni altres estructures de dades (llistes, cues, vectors, ...).

Us proposem fer servir una variant de l'algorisme de selecció: Repetidament, triar el node amb l'element més *gran* de la pila i traslladar-lo a la primera posició d'una nova cadena de nodes, que al final conté doncs els elements ordenats de petit a gran.

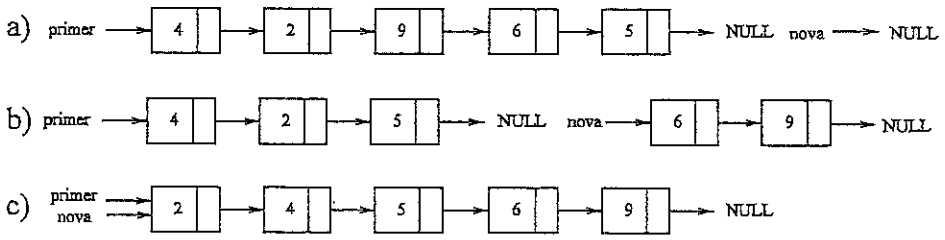
Digueu el codi que cal posar a CONDICIO_1, CONDICIO_2, INSTRUCCIONS_1 i INSTRUCCIONS_2 per implementar aquesta idea en aquest codi:

```
// Pre: el p.i. és una pila P
// Post: el p.i. conté els mateixos elements de P, però ordenats
// del més petit (al cim) al més gran (al fons)
void ordena() {
    node* nova = NULL;
    // a)
    while (CONDICIO_1) {
        node* p = primer;
        node* max = primer;
        node* ant_max = NULL;
        while (CONDICIO_2) {
            INSTRUCCIONS_1
        }
        INSTRUCCIONS_2
        // b)
    }
    primer = nova;
    // c)
}
```

sabent que es tracta de mantenir aquests dos invariants:

- *Invariant del bucle extern:* nova apunta a una cadena de nodes que conté els elements més grans de P, ordenats de més petit a més gran; primer apunta a la cadena de nodes que resulta d'eliminar en P els elements que són a la cadena apuntada per nova.
- *Invariant del bucle intern:* primer no és NULL, p apunta a un node de la cadena que comença a primer, max apunta a un node que té el valor màxim dels que són entre l'apuntat per primer i l'apuntat per p, ambdós inclosos, i ant_max apunta al node anterior a l'apuntat per max, excepte si aquest no té anterior, cas en què ant_max és NULL.

En l'esquema següent es mostra l'estat dels apuntadors primer i nova en els punts a) i c) de la plantilla de codi, i en el punt b) després d'executar dues iteracions del bucle extern.



No cal justificar el codi. Els invariants que us donem són per guiar la vostra implementació. Es valorarà molt l'eficiència de la solució proposada.

Problema 2 (5 punts)

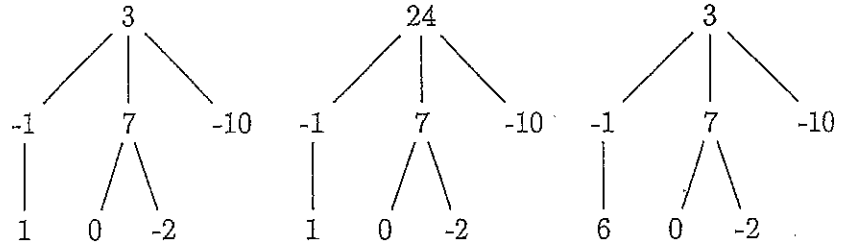
Considerem la representació habitual amb nodes de la classe *ArbreGen* per manegar arbres generals genèrics d'elements de tipus T:

```
template <class T> class ArbreGen {
  private:
    struct node {
      T info;
      vector<node*> seg;
    };
    node* primer_node;
    ... // especificació i implementació d'operacions privades
  public:
    ... // especificació i implementació d'operacions públiques
};
```

Suposem arbres generals on els elements són nombres enters. Volem afegir-hi una nova operació pública amb la següent especificació

```
int suma_max_subarbre() const;
/* Pre: el p.i. no és buit */
/* Post: el resultat és la suma del subarbre del p.i. amb suma màxima */
```

Els tres arbres generals següents



són pràcticament idèntics. La crida a `suma_max_subarbre` amb l'arbre de l'esquerra com a p.i. ha d'obtenir 5 (el subarbre de suma màxima és el que té arrel 7). Amb l'arbre del centre ha d'obtenir 19 (el subarbre de suma màxima és el que té arrel 24). Amb l'arbre de la dreta ha d'obtenir 6 (el subarbre de suma màxima és el que té arrel 6). Fixeu-vos que el p.i. ha de quedar intacte després d'executar aquesta operació.

Heu de dissenyar una implementació d'aquesta operació que **no usi cap operació pública dels arbres generals** (per això no les recordem en aquest enunciat) sinó que accedeixi directament als atributs de la classe *ArbreGen*.

Es valoraran negativament solucions que realitzen visites, còpies o esborraments de nodes innecessàriament.

Rumieu els Problemes 2.1 i 2.2 conjuntament.

Problema 2.1 (1 punt)

Doneu l'especificació (capçalera, pre i post) d'una acció o funció d'immersió privada de nom `i_suma_max_subarbre` que permeti solucionar el problema. Recomanem que aquesta operació no tingui paràmetres que siguin arbres generals.

Problema 2.2 (1 punt)

Doneu una implementació eficient de l'operació `suma_max_subarbre`, fent ús de `i_suma_max_subarbre`.

Problema 2.3 (3 punts)

Doneu una implementació eficient de l'operació `i_suma_max_subarbre`.

Examen de Programació 2

Grau en Informàtica, FIB, UPC
Juny de 2014
Temps estimat: 2h 30m

Notes per als dos exercicis: No cal que justifiqueu la vostra solució, però podeu incloure comentaris aclaridors si ho creieu convenient. Si feu servir operacions auxiliars, és necessari que les especifiqueu i implementeu. En tots dos problemes, es valorarà *molt* l'eficiència de la solució proposada.

Problema 1: *Trams creixents d'una llista* (50%)
Considerem un tipus `T` amb una relació d'ordre, per tant amb operacions `==`, `<`, `>`, `<=`, `>=`.

Un *tram* d'una llista de `T` és una subseqüència màxima tal que cada element és `>=` que l'anterior. Feu un mètode propi de la classe `Llista<T>`:

```
void trams(vector<Llista<T> >& v);
/* Pre: v.size() == 0, el p.i. és una llista L */
/* Post: el p.i. és buit, la mida de v és el nombre de trams de L,
         cada posició v[i] conté una llista amb l'i-èssim tram de L
         i el punt d'interès sobre el seu primer element */
```

A efectes d'aquest exercici, diem que el primer tram d'una llista no buida és el tram 0. Per exemple, aplicat a la llista d'enters (3,7,10,10,4,6,6,12,6,-2,5), el mètode ha de deixar en `v` un vector de mida 4 que contingui les següents 4 llistes:

$$v[0] = (3,7,10,10),\ v[1] = (4,6,6,12),\ v[2] = (6),\ v[3] = (-2,5)$$

Fixeu-vos que el vector retornat no contindrà mai cap llista buida. i que tindrà mida 0 si i només si el paràmetre implícit és inicialment la llista buida.

Algunes condicions per a la resolució del problema:

- Noteu que no es coneix inicialment el nombre de trams de la llista. Hi ha solucions bones que el calculen amb un primer recorregut; també pot fer-se servir la instrucció `v.push_back(Llista())`, que afegeix al final d'un vector `v` una nova component amb la llista buida i incrementa la mida de `v` en una unitat.
- No podeu fer servir cap operació pública de llistes més que la creadora `Llista()`, ni tant sols l'assignació. Cal resoldre el problema accedint a la representació amb nodes doblement encadenats, que us recordem més avall.
- Es penalitzaran fortament les còpies innecessàries d'estructures i d'elements de tipus `T`, que podrien ser molt voluminosos. Per la mateixa raó, es valorarà fer un mínim de comparacions entre elements de tipus `T`.

Representació de la classe Llista<T>:

```
template <class T> class Llista {
  private:
    struct node {
      T info;
      node* seg;
      node* ant;
    };
    int longitud;  // nombre d'elements a la llista
    node* primer;  // apuntador al primer node; NULL en la llista buida
    node* ultim;   // apuntador al darrer node; NULL en la llista buida
    node* act;     // apuntador al punt d'interès;
                   // NULL quan aquest és l'element fictici del final
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
};
```

Problema 2: *Arbres de màxims i mínims d'un arbre general* (50%)
Considerem la representació habitual amb nodes de la classe *ArbreGen* per manegar arbres generals genèrics d'elements de tipus T:

```
template <class T> class ArbreGen {
  private:
    struct node {
      T info;
      vector<node*> seg;
    };
    node* primer_node;
    ...  // especificació i implementació d'operacions privades

  public:
    ...  // especificació i implementació d'operacions públiques
};
```

Un arbre només té un atribut: un apuntador al primer node. Cada node conté la seva info i un vector d'apuntadors, que representa els seus successors. Per a tot node intern, el vector no és buit; per a tota fulla, el vector és buit.

Es vol afegir una operació a la classe ArbreGen, que donat un arbre general ens construeixi dos arbres generals amb la mateixa estructura (forma). A cada node del primer arbre resultat hi ha el màxim valor dels elements de l'arbre original l'arrel del qual és el node que està a la mateixa posició que el node de l'arbre resultat. Al segon arbre resultat, cada node conté el mínim corresponent.

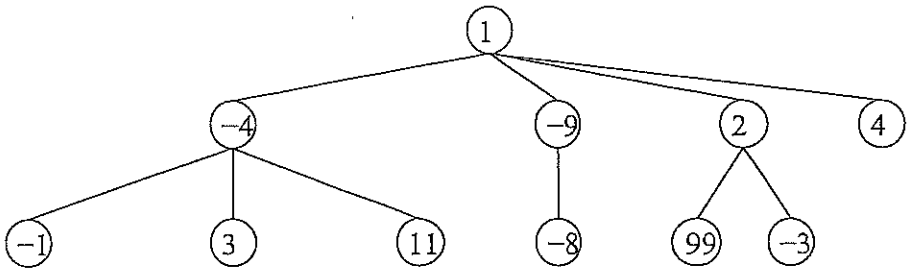


Figura 1: Exemple d'arbre general a

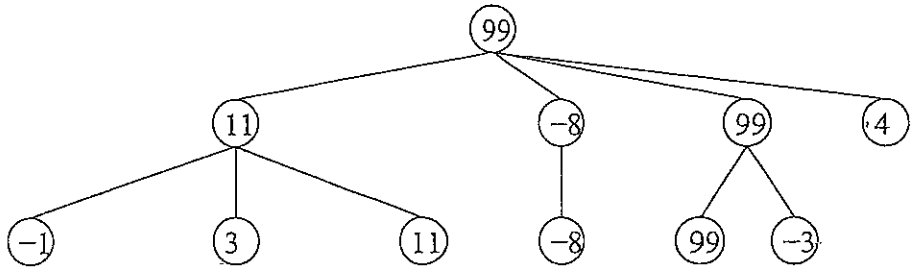


Figura 2: Arbre de màxims d'a

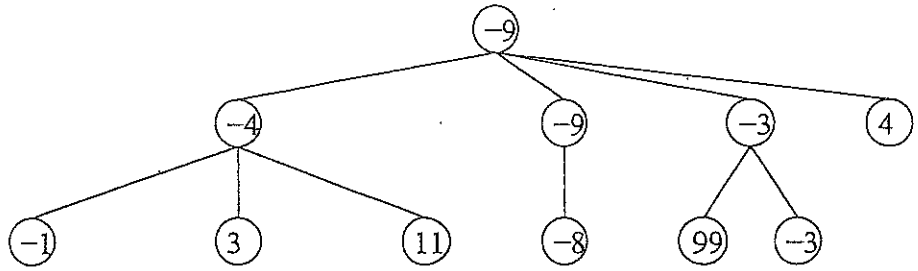


Figura 3: Arbre de mínims d'a

Aquests arbres només s'instanciaran amb tipus que tinguin definida una relació d'ordre, per tant amb les operacions ==, <=, >=, < i >.

Us demanem implementar una operació pública d'aquesta classe, amb la següent especificació pre/post:

```
void arbre_max_min(ArbreGen<T> & amax, ArbreGen<T> & amin) const;
/* Pre: el p.i. no és buit, amax i amin són buits */
/* Post: amax és l'arbre de màxims del p.i, amin és l'arbre de mínims del p.i. */
```

Es demana dissenyar una implementació d'aquesta operació que **no usi cap operació pública dels arbres generals** (per això no les recordem en aquest enunciat) sinó que accedeixi directament als atributs de la classe *ArbreGen*.

Es penalitzarà fer més d'un recorregut del paràmetre implícit.

Examen Programació 2

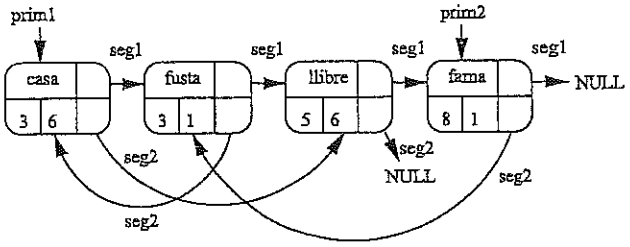
Grau en Informàtica
Gener 2014
Temps estimat: 2h 30m

Problema 1: *Llista amb dos ordres* (50%)
Considerem una representació no genèrica d'una *Llista* que conté elements amb tres atributs: nom, valor1, valor2. Volem que els elements de la llista estiguin ordenats de dues maneres diferents: una en ordre creixent segons el valor1 i l'altra en ordre creixent segons el valor2. En cada ordenació, els elements amb el mateix valor han d'estar ordenats en ordre alfabètic segons el nom.

```
class Llista_doble_ordre {
  private:
    struct node_llista {
        string nom;
        int val1;
        int val2;
        node_llista* seg1;
        node_llista* seg2;
    };

    node_llista* prim1;
    node_llista* prim2;
    ... // especificació i implementació d'operacions privades

  public:
    ... // especificació i implementació d'operacions públiques
};
```

Els nodes són simplement encadenats amb punters al següent element segons el valor1 (`seg1`) i al següent element segons el valor2 (`seg2`). Una llista té dos atributs: dos punters a nodes, un pel primer element segons la primera ordenació (`prim1`) i un altre pel primer element segons la segona ordenació (`prim2`). Noteu que la llista no té element actual ni punter a l'últim i els nodes no tenen punter a l'anterior.

Gràficament, la implementació de l'estructura és



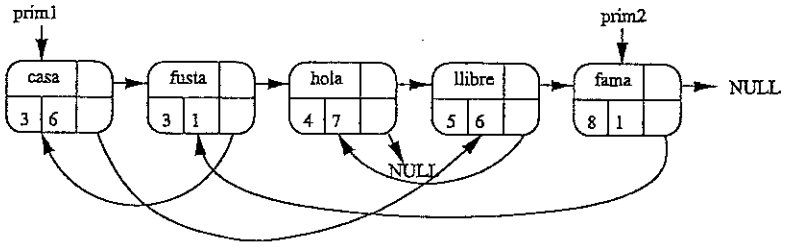
Volem implementar dins d'aquesta classe una operació pública amb la següent especificació pre/post:

```
void afegir(const string& s, int v1, int v2, bool& b)
/* Pre: cert */
/* Post: b indica si existeix un element al paràmetre implícit amb nom igual a s;
   si b és cert, el p. i. no canvia, si b és fals, s'ha afegit al p. i.
   un nou element amb nom igual a s, valor1 igual a v1 i valor2 igual a v2 */
```

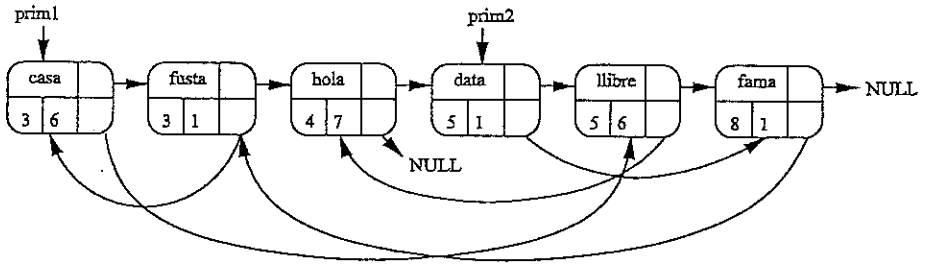
Noteu que quan s'afegeix un nou element a la llista, s'ha de preservar l'ordenació descrita al principi.

Exemples:

- si amb la llista del gràfic cridem `afegir("hola", 4, 7, b)` obtenim `b=false` i



- si amb la llista resultant cridem `afegir("data", 5, 1, b)` obtenim `b=false` i



- si amb la llista resultant cridem `afegir("llibre", 3, 14, b)` obtenim `b=true` i la llista no canvia.

Es demana dissenyar una implementació d'aquesta operació que **no usi cap operació primitiva de les llistes** (per això no les recordem en aquest enunciat) **ni iteradors**, sinó que accedeixin directament als atributs de la classe *Llista_doble_ordre*.

No s'han de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient. Si feu servir operacions auxiliars, és necessari que les especifiqueu, a més d'implementar-les.

Es valorarà l'eficiència de la solució proposada. Aquesta ha de ser lineal respecte a la mida de la llista, ha vistar el mínim nombre possible de nodes i ha de fer el mínim nombre possible de comparacions de strings.

Problema 2: *Antecessor comú més pròxim en un arbre Nari* (50%)
Considerem la representació habitual amb nodes de la classe *ArbreNari* per manegar arbres N-aris genèrics d'elements de tipus T:

```
template <class T> class ArbreNari {
  private:
    struct node_arbreNari {
      T info;
      vector<node_arbreNari*> seg;
    };
    int N;
    node_arbreNari* primer_node;

    ... // especificació i implementació d'operacions privades

  public:
    ... // especificació i implementació d'operacions públiques
};
```

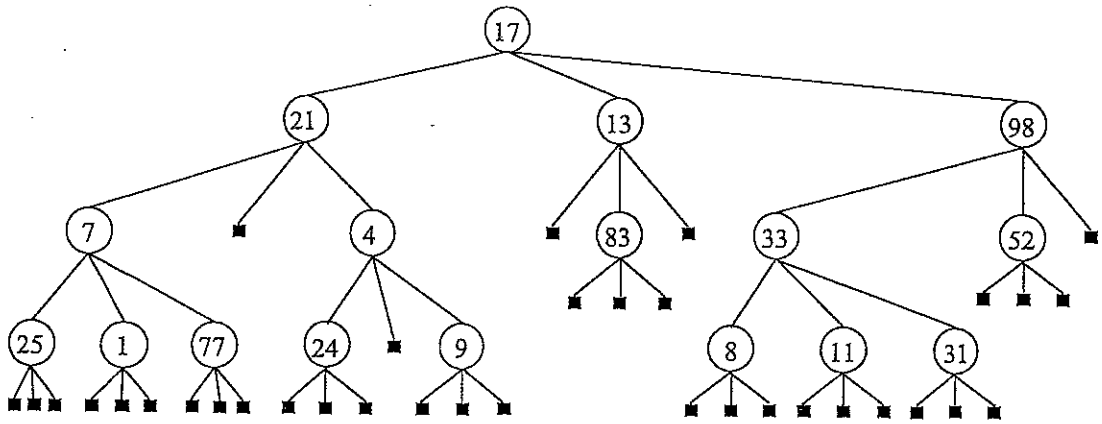
Un arbre té dos atributs: l'aritat (N) i un punter al primer node. Cada node conté la seva info i un vector de N punters a nodes.

Donats dos elements x, y d'un arbre, diem que x és antecessor d'y si y apareix al subarbre que té x com a arrel; en particular, un element és antecessor de sí mateix. Donats tres elements x, y, z d'un arbre, diem que x és l'antecessor comú més proper d'y i z si és antecessor dels dos i no hi ha cap altre antecessor comú d'y i z al subarbre que té x com a arrel.

Us demanem implementar una operació pública d'aquesta classe, amb la següent especificació pre/post

```
void antecessor_comu_mes_proper(const T& y, const T& z, T& x, bool& b) const
/* Pre: el paràmetre implícit té tots els elements diferents, y != z */
/* Post: b indica si y i z hi són al paràmetre implícit; si b, llavors x és
l'antecessor comú més proper d'y i z  */
```

Exemple: si a és el següent arbre 3-ari d'enters



llavors les següents crides produeixen els següents resultats:

- `a.antecessor_comu_mes_proper(25, 77, x, b)` ha de produir `b=true`, `x=7`
- `a.antecessor_comu_mes_proper(4, 11, x, b)` ha de produir `b=true`, `x=17`
- `a.antecessor_comu_mes_proper(98, 52, x, b)` ha de produir `b=true`, `x=98`
- `a.antecessor_comu_mes_proper(13, 3, x, b)` ha de produir `b=false` i `x` pot valer qualsevol cosa

Es demana dissenyar una implementació d'aquesta operació que **no usi cap operació primitiva dels arbres** (per això no les recordem en aquest enunciat) sinó que accedeixi directament als atributs de la classe *ArbreNari*.

No s'ha de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient. Si feu servir operacions auxiliars, és necessari que les especifiqueu, a més d'implementar-les.

Es valorarà l'eficiència de la solució proposada.

Examen Programació 2

Grau en Informàtica
Juny 2013
Temps estimat: 2h 30m

Problema 1: *Trenat de cues* (50%)
Considerem la representació habitual amb nodes de la classe *Cua* per manegar cues genèriques d'elements de tipus T:

```
template <class T> class Cua {
  private:
    struct node_cua {
      T info;
      node_cua* seg;
    };
    int longitud;
    node_cua* prim;
    node_cua* ult;

    ... // especificació i implementació d'operacions privades

  public:
    ... // especificació i implementació d'operacions públiques
};
```

Els nodes són simplement encadenats amb punters al següent (`seg`). Una cua té tres atributs: la `longitud` i dos punters a nodes, un pel primer element (`prim`) i un per l'últim (`ult`).

Diem que el resultat de trenar (castellà: "trenzar") dues cues q_1 i q_2 és una cua q_3 on els elements de q_1 apareixen a les posicions senars (primera, tercera, cinquena, etc) i els elements de q_2 apareixen a les posicions parelles (segona, quarta, sisena, etc). Després de l'últim element de la més curta apareixen la resta dels elements de la més llarga.

Volem implementar dins de la classe `Cua` una operació nova amb la següent especificació pre/post:

```
void trenat(Cua &c)
/* Pre: p.i. = C1, c = C2 */
/* Post: el p.i. passa a ser el resultat de trenar C1 i C2; c passa a ser buida */
```

Exemples:

1. Si el p.i. és originalment

   1 5 14 25 8

i c és

99 3

llavors el p.i. ha de quedar

1 99 5 3 14 25 8

2. Si el p.i. és originalment

67 89

i c és

19 24 5 17 6

llavors el p.i. ha de quedar

67 19 89 24 5 17 6

Es demana dissenyar una implementació d'aquesta operació que no usi cap operació primitiva de les cues (per això no les recordem en aquest enunciat), sinó que accedeixi directament als atributs de la classe *Cua*.

Els codis no s'han de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient. Si feu servir operacions auxiliars, és necessari que les especifiqueu.

Es valorarà l'eficiència de la solució proposada.

Problema 2: *Cerca de subarbres binaris* (50%)
Considerem la representació habitual amb nodes de la classe *Arbre* per manegar arbres binaris genèrics d'elements de tipus T:

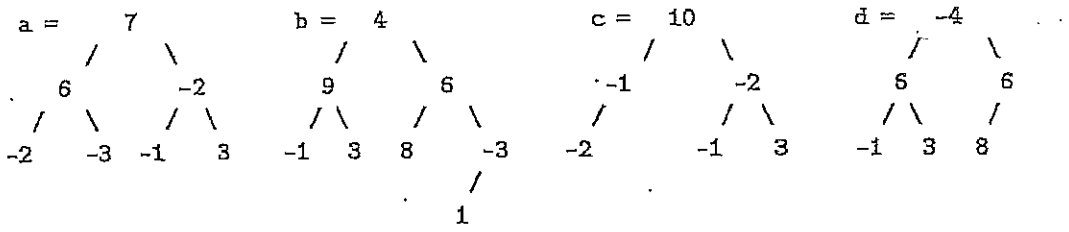
```
template <class T> class Arbre {
  private:
    struct node_arbre {
      T info;
      node_arbre* segE;
      node_arbre* segD;
    };

    node_arbre* primer_node;

    ... // especificació i implementació d'operacions privades

  public:
    ... // especificació i implementació d'operacions públiques
};
```

Un arbre només té un atribut: un punter al primer node. Cada node conté la seva info i dos punters que representen els seus successors.

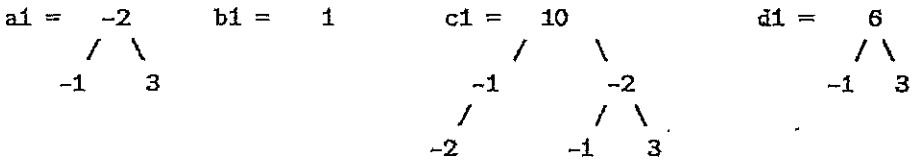
Volem una nova operació de la classe que donat un arbre a i un valor x de tipus T comprovi si x apareix en a; en cas que la cerca sigui exitosa, ha d'obtenir el subarbre que tingui com a arrel l'aparició d'x més propera a l'arrel d'a. En cas d'haver-hi diferents aparicions d'x a la distància mínima, ha d'obtenir el subarbre de més a l'esquerra. Feu servir la següent especificació:

```
void sub_arrel(Arbre& asub, const T& x) const
  /* Pre: p.i. = A, asub és buit */
  /* Post: si A conté x, asub és el subarbre d'A resultat de la cerca;
     si A no conté x, asub és buit */
```

Exemples: siguin els següents arbres d'enters



llavors les instruccions `a.sub_arrel(a1, -2)`, `b.sub_arrel(b1, 1)`, `c.sub_arrel(c1, 10)`, `d.sub_arrel(d1, 6)` han de produir els següents arbres



Es demana dissenyar una implementació d'aquesta operació **que no usi** cap operació pública dels arbres binaris (per això no les recordem en aquest enunciat) sinó que accedeixi directament als atributs de la classe *Arbre*. Si que podeu fer servir l'operació privada

```
static node_arbre* copia_node_arbre(node_arbre* m)
  /* Pre: cert */
  /* Post: el resultat és NULL si m és NULL; en cas contrari, el resultat
     apunta al node arrel d'una jerarquia de nodes que és una còpia de la
     jerarquia de nodes que té el node apuntat per m com a arrel */
```

No s'ha de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient. Si feu servir operacions auxiliars, és necessari que les especifiqueu.

Es valorarà l'eficiència de la solució proposada.

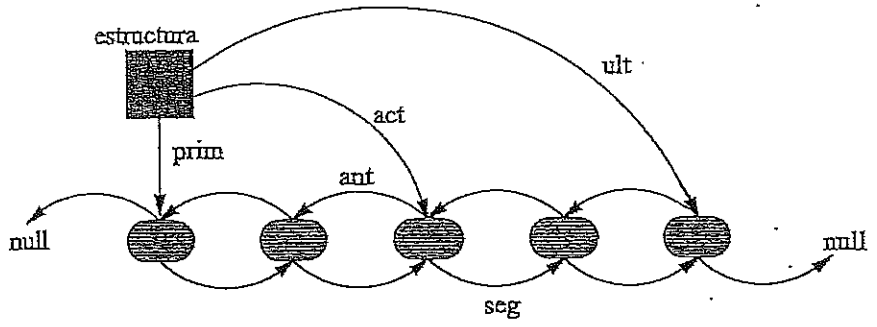
Examen Programació 2

Grau en Informàtica
Gener 2013
Temps estimat: 2h 30m

Problema 1: *Esborrat de totes les aparicions d'un element a una llista* (50%)
Considerem la representació habitual amb nodes de la classe *Llista* per manegar llistes genèriques d'elements de tipus T:

```
template <class T> class Llista {
  private:
    struct node_llista {
        T info;
        node_llista* seg;
        node_llista* ant;
    };

    int longitud;
    node_llista* prim;
    node_llista* ult;
    node_llista* act;
    ...  // especificació i implementació d'operacions privades

  public:
    ...  // especificació i implementació d'operacions públiques
};
```

Els nodes són doblement encadenats amb punters al següent (seg) i a l'anterior (ant). Una llista té quatre atributs: la `longitud` i tres punters a nodes, un pel primer element (`prim`), un per l'últim (`ult`) i un altre per l'element actual (`act`), on tenim situat el punt d'interès de la llista. Gràficament, la implementació de l'estructura és



Recordeu que si l'`act` d'una llista no buida té valor `null` significa que el punt d'interès està situat a la dreta de tot, a sobre d'un element fictici posterior a l'últim element real.

Volem implementar dins d'aquesta classe una operació nova amb la següent especificació pre/post:

```
void esborrar_tots(const T& x)
/* Pre: paràmetre implícit = P */
/* Post: s'han eliminat del paràmetre implícit totes les aparicions d'x (la
resta d'elements queda en el mateix ordre que a P); si el punt d'interès de P
referenciava a una aparició d'x, passa a referenciar al primer element
diferent d'x posterior a aquesta (si no hi ha cap element diferent d'x, passa
a la dreta del tot); en cas contrari, el punt d'interès no canvia */
```

Exemples:

1. si x = 5 i el paràmetre implícit és

   1 3 5 7 12 5 5 5 25

   amb el punt d'interès sobre el 7, llavors el nou paràmetre implícit ha de quedar

   1 3 7 12 25

   amb el punt d'interès sobre el 7;

2. si x = 5 i el paràmetre implícit és

   1 3 5 7 12 5 5 5 5

   amb el punt d'interès sobre el penúltim 5, llavors el nou paràmetre implícit ha de quedar

   1 3 7 12

   amb el punt d'interès a la dreta del tot

Es demana dissenyar una implementació d'aquesta operació que **no usi cap operació primitiva de les llistes** (per això no les recordem en aquest enunciat) **ni iteradors**, sinó que accedeixin directament als atributs de la classe *Llista*.

No s'han de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient. Si feu servir operacions auxiliars, és necessari que les especifiqueu, a més d'implementar-les.

Es valorarà l'eficiència de la solució proposada.

Problema 2: *Suma màxima dels camins d'un arbre general* (50%)
Considerem la representació habitual amb nodes de la classe *ArbreGen* per manegar arbres generals genèrics d'elements de tipus T:

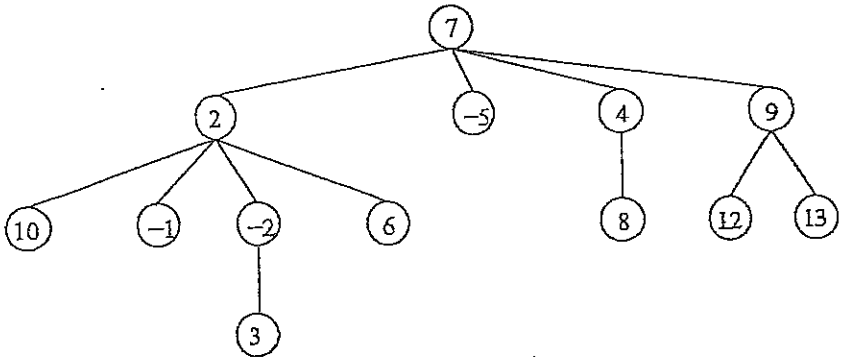
```
template <class T> class ArbreGen {
  private:
    struct node_arbreGen {
      T info;
      vector<node_arbreGen*> seg;
    };
    node_arbreGen* primer_node;
    ... // especificació i implementació d'operacions privades

  public:
    ... // especificació i implementació d'operacions públiques
};
```

Un arbre només té un atribut: un punter al primer node. Cada node conté la seva info i un vector de punters, que representa els seus successors. Per a tot node intern, el vector no és buit; per a tota fulla, el vector és buit.

Recordeu que hem definit un *camí* a un arbre com una successió de nodes que van de l'arrel a una fulla. Us demanem implementar una operació nova d'aquesta classe, amb la següent especificació pre/post, on el tipus T ha de tenir definides les operacions + (suma) i > (més gran que):

```
T max_suma_cami() const
/* Pre: el paràmetre implícit no és buit */
/* Post: el resultat és la suma del camí de suma màxima del paràmetre implícit */
```

Exemple: si a és el següent arbre general d'enters



llavors la crida a.max_suma_cami() ha de retornar 29 (7+9+13). Si en comptes dels valors 12 i 13 tinguéssim 1 i -2, el resultat hauria de ser 19 (7+2+10 o també 7+4+8).

Es demana dissenyar una implementació d'aquesta operació que no usi cap operació primitiva dels arbres (per això no les recordem en aquest enunciat) sinó que accedeixi directament als atributs de la classe *ArbreGen*.

No s'ha de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient. Si fen servir operacions auxiliars, és necessari que les especifiqueu, a més d'implementar-les.

Es valorarà l'eficiència de la solució proposada.

Examen Programació 2

Grau en Informàtica
Juny 2012
Temps estimat: 2h 30m

Problema 1: *Transvasament d'elements d'una cua* (50%)
Considerem la representació habitual amb nodes de la classe *Cua* per manegar cues genèriques d'elements de tipus T:

```
template <class T> class Cua {
  private:
    struct node_cua {
      T info;
      node_cua* seg;
    };
    int longitud;
    node_cua* prim;
    node_cua* ult;

    ... // especificació i implementació d'operacions privades

  public:
    ... // especificació i implementació d'operacions públiques
};
```

Els nodes són simplement encadenats amb punters al següent (seg). Una cua té tres atributs: la longitud i dos punters a nodes, un pel primer element (prim) i un per l'últim (ult).

Volem implementar dins d'aquesta classe una operació nova amb la següent especificació pre/post:

```
void mou_grans_cua(Cua &c, const T &k)
/* Pre: p.i.cua buida, c=C */
/* Post: s'han eliminat de c i afegit al paràmetre implicit tots els elements
   de C més grans que k, en el mateix ordre relatiu en que apareixien a C;
   c conté la resta d'elements de C en l'ordre relatiu original */
```

Exemples:

1. si k=10 i C és

   1 5 14 25

   llavors el paràmetre implícit ha de quedar

14 25

i c ha de quedar

1 5

2. si k=13 i C és

19 24 5 17 6 9 13

llavors el paràmetre implícit ha de quedar

19 24 17

i c ha de quedar

5 6 9 13

3. si k=2 i C és

7 4 25 11

llavors el paràmetre implícit ha de quedar

7 4 25 11

i c ha de quedar buida

Es demana dissenyar una implementació d'aquesta operació que **no usi cap operació primitiva de les cues** (per això no les recordem en aquest enunciat), sinó que accedeixi directament als atributs de la classe *Cua*.

Els codis no s'han de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient. Si feu servir operacions auxiliars, és necessari que les especifiqueu.

Es valorarà l'eficiència de la solució proposada.

Problema 2: *Aproximació d'arbres generals mitjançant arbres N-aris* (50%)
Considerem la representació habitual amb nodes de la classe *ArbreNari* per manegar arbres N-aris genèrics d'elements de tipus T:

```
template <class T> class ArbreNari {
  private:
    struct node_arbreNari {
      T info;
      vector<node_arbreNari*> seg;
    };
    int N;
    node_arbreNari* primer_node;

    ... // especificació i implementació d'operacions privades

  public:
    ... // especificació i implementació d'operacions públiques
};
```

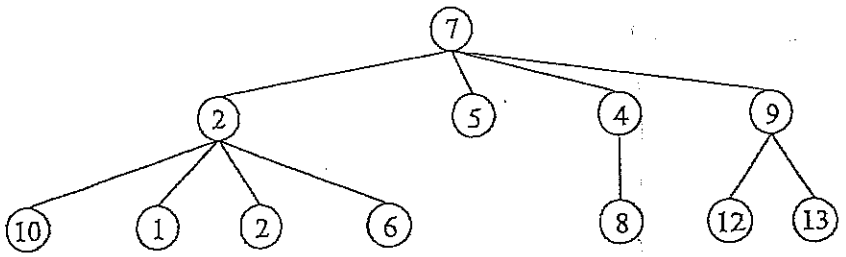
Un arbre té dos atributs: l'aritat (N) i un punter al primer node. Cada node conté la seva info i un vector de N punters a nodes.

Diem que un arbre N-ari a *aproxima* un arbre general g si a conté els mateixos elements i en la mateixa posició (profunditat i número de fills) que g, excepte aquells que pertanyen a qualsevol subarbre de g que sigui fill i-èssim amb un i més gran que N; la resta de subarbres que completen l'aritat N en cada subarbre no buit d'a són arbres buits.

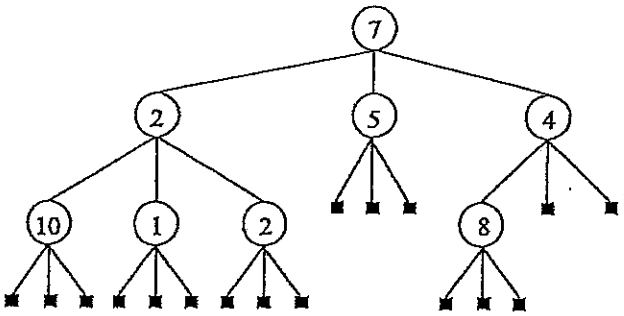
Demanem implementar dins la classe *ArbreNari* una nova constructora amb la següent especificació:

```
ArbreNari (int n, ArbreGen<T> &g)
/* Pre: n>=2, g=G */
/* Post: el resultat és un arbre n-ari que aproxima l'arbre general G */
```

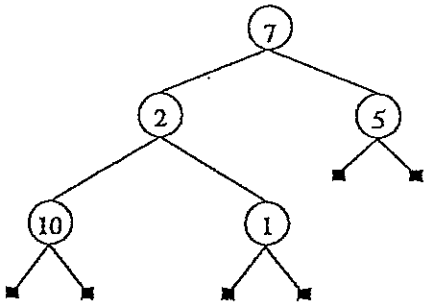
Exemples: sigui a el següent arbre general d'enters



llavors la instrucció `ArbreNari b(3,a)` ha de produir el següent arbre 3-ari b



si en comptes d'això fem `ArbreNari b(2,a)`, l'arbre `b` (en aquest cas 2-ari) ha de ser



Es demana dissenyar una implementació d'aquesta operació que no usi cap operació primitiva dels arbres N-aris (per això no les recordem en aquest enunciat) sinó que accedeixi directament als atributs de la classe *ArbreNari*.

No s'ha de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient. Si feu servir operacions auxiliars, és necessari que les especifiqueu.

Es valorarà l'eficiència de la solució proposada.

Us recordem l'especificació de les operacions que necessiteu de la clase *ArbreGen*:

```
bool es_buit() const
/* Pre: cert */
/* Post: el resultat indica si el p.i. és un arbre buit */

int nombre_fills() const
/* Pre: el p.i. no és buit */
/* Post: el resultat és el nombre de fills del p.i. */

T arrel() const
/* Pre: el p.i. no és buit */
/* Post: el resultat és l'arrel del p.i. */

void fills(vector<ArbreGen> &v)
/* Pre: el p.i. no és buit i li diem A, v és un vector buit */
/* Post: el p.i. és buit, v passa a contenir els fills de l'arbre A */
```

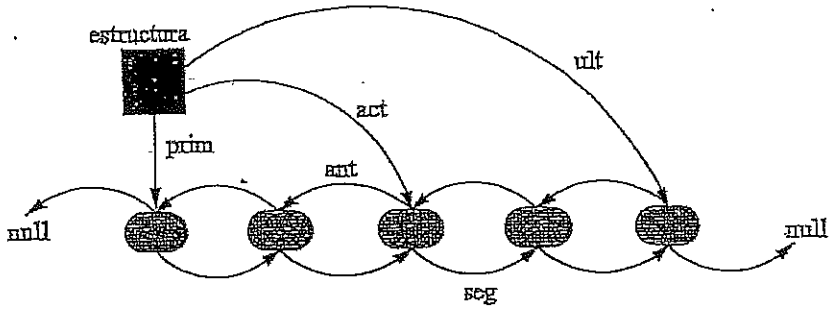
Examen Programació 2

Grau en Informàtica
Gener 2012
Temps estimat: 2h 50m

Problema 1: *"Merge" de llistes* (50%)
Considerem la representació habitual amb nodes de la classe *Llista* per manegar llistes genèriques d'elements de tipus T:

```
template <class T> class Llista {
  private:
    struct node_llista {
        T info;
        node_llista* seg;
        node_llista* ant;
    };

    int longitud;
    node_llista* prim;
    node_llista* ult;
    node_llista* act;
    ... // especificació i implementació d'operacions privades

  public:
    ... // especificació i implementació d'operacions públiques
};
```

Els nodes són doblement encadenats amb punters al següent (`seg`) i a l'anterior (`ant`). Una llista té quatre atributs: la `longitud` i tres punters a nodes, un pel primer element (`prim`), un per l'últim (`ult`) i un altre per l'element actual (`act`), on tenim situat el punt d'interès de la llista. Gràficament, la implementació de l'estructura és



Volem implementar dins d'aquesta classe una operació nova amb la següent especificació pre/post:

```
void merge(Llista &l)
/* Pre: el paràmetre implícit = P, l = L, L i P estan ordenades creixentment */
/* Post: el paràmetre implícit està format pels elements de L i P i està ordenat
creixentment; el punt d'interès queda al principi de tot; l és una llista buida */
```

Exemples:

1. si el paràmetre implícit és

   1 5 14 25

   i l és

   3 7 12 18 20

   llavors el nou paràmetre implícit ha de ser

   1 3 5 7 12 14 18 20 25

   amb el punt d'interès sobre el 1 i l ha de quedar buida.

2. si el paràmetre implícit és

   1 5 6 9 13

   i l és

   9 13 15 20 20

   llavors el nou paràmetre implícit ha de ser

   1 5 6 9 9 13 13 15 20 20

   amb el punt d'interès sobre el 1 i l ha de quedar buida.

Es demana dissenyar una implementació d'aquesta operació que **no usi cap operació primitiva de les llistes** (per això no les recordem en aquest enunciat) **ni iteradors**, sinó que accedeixin directament als atributs de la classe *Llista*.

No s'han de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient. Si feu servir operacions auxiliars, és necessari que les especifiqueu.

Es valorarà l'eficiència de la solució proposada.

Problema 2: *Cerca amb profunditat a arbres N-aris* (50%)
Considerem la representació habitual amb nodes de la classe *ArbreNari* per manegar arbres N-aris genèrics d'elements de tipus T:

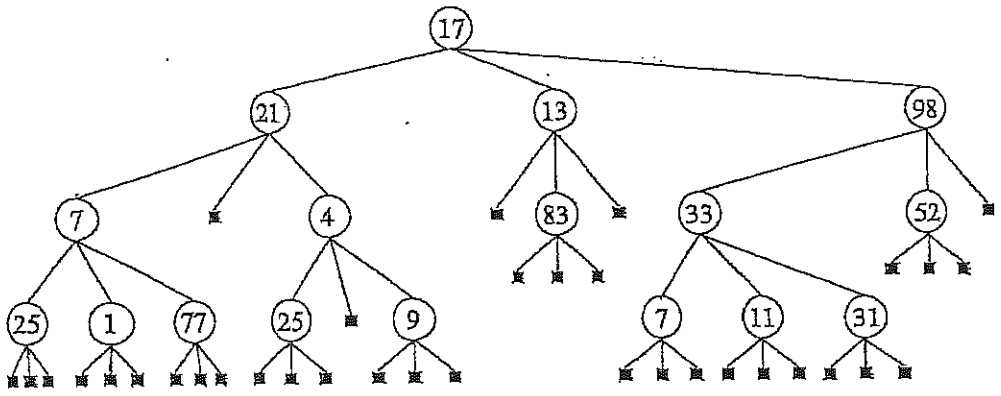
```
template <class T> class ArbreNari {
  private:
    struct node_arbreNari {
      T info;
      vector<node_arbreNari*> seg;
    };
    int N;
    node_arbreNari* primer_node;

    ... // especificació i implementació d'operacions privades

  public:
    ... // especificació i implementació d'operacions públiques
};
```

Un arbre té dos atributs: l'aritat (N) i un punter al primer node. Cada node conté la seva info i un vector de N punters a nodes.

Diem que la *profunditat* d'un element d'un arbre és la distància des de l'arrel de l'arbre a l'element. Noteu que si un element està repetit a un arbre, llavors pot tenir diferents profunditats. Us demanem implementar una operació nova d'aquesta classe, amb la següent especificació pre/post:

```
int cerca_prof_min(const T &x) const
/* Pre: cert */
/* Post: el resultat indica si x hi és al paràmetre implícit: -1 si no hi és;
   0 si hi és a l'arrel; la profunditat mínima a la resta de casos */
```

Exemple: sigui a el següent arbre 3-ari



llavors les següents crides han de produir els següents resultats:

1. `a.cerca_prof_min(5)` ha de retornar -1
2. `a.cerca_prof_min(17)` ha de retornar 0
3. `a.cerca_prof_min(13)` ha de retornar 1
4. `a.cerca_prof_min(11)` ha de retornar 3
5. `a.cerca_prof_min(7)` ha de retornar 2 (n'hi ha dues aparicions, a profunditats 2 i 3, però volem la profunditat mínima)

Es demana dissenyar una implementació d'aquesta operació que no usi cap operació primitiva dels arbres (per això no les recordem en aquest enunciat) sinó que accedeixi directament als atributs de la classe *ArbreNari*.

No s'ha de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient. Si feu servir operacions auxiliars, és necessari que les especifiqueu.

Es valorarà l'eficiència de la solució proposada.

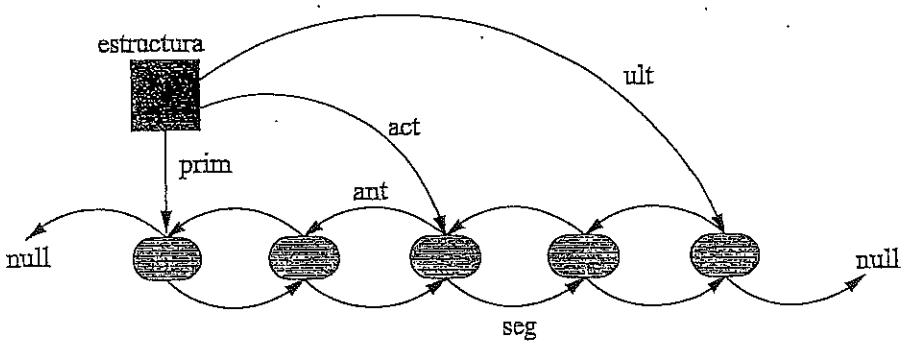
Examen Programació 2

Grau en Informàtica
Juny 2011
Temps estimat: 2h 50m

Problema 1: *Preu just i "splice" de llistes* (50%)
Suposem la següent representació de tipus basada en nodes d'una classe *Llista* per manegar llistes genèriques d'elements de tipus T:

```
template <class T> class Llista {
  private:
    struct node_llista {
        T info;
        node_llista* seg;
        node_llista* ant;
    };

    int longitud;
    node_llista* prim;
    node_llista* ult;
    node_llista* act;
    ... // especificació i implementació d'operacions privades

  public:
    ... // especificació i implementació d'operacions públiques
};
```

Els nodes són doblement encadenats amb punters al següent (`seg`) i a l'anterior (`ant`), i la llista té quatre atributs: la `longitud` i tres punters a nodes, un pel primer element (`prim`), un per l'últim (`ult`) i un altre per l'element actual (`act`), on tenim situat el punt d'interès de la llista. Gràficament, la implementació de l'estructura és



Noteu, per tant, que es tracta d'una implementació sense sentinella. Recordeu que l'element act d'una llista sense sentinella pot ser null fins i tot si la llista no és buida. És en aquest cas quan diem que el punt d'interès hi és a la dreta del tot.

Volem implementar dins d'aquesta classe dues operacions noves amb les següents especificacions pre/post:

```
void moure_punt_preu_just(const T &x)
/* Pre: el paràmetre implícit no té elements repetits */
/* Post: el punt d'interès del paràmetre implícit passa a estar sobre
   l'element més gran del paràmetre implícit que sigui estrictament menor
   que x, si n'hi ha; en cas contrari passa a estar a la dreta del tot */

void splice(Llista &l)
/* Pre: el paràmetre implícit = P, l = L */
/* Post: el paràmetre implícit conté el resultat d'inserir la llista L
   en la llista P a l'esquerra del punt d'interès de P; el punt d'interès
   del paràmetre implícit no canvia; l és una llista buida */
```

Exemples:

1. moure_punt_preu_just: si $x = 5$ i el paràmetre implícit és

   -1 3 -7 14 -2 5 -6 9 1 4 2

   el nou punt d'interès ha de quedar sobre el 4.

2. splice: si el paràmetre implícit és

   1 2 3 4 5

   amb el punt d'interès sobre el 3 i la llista l és

   6 7 8 9

   llavors el nou paràmetre implícit ha de ser

   1 2 6 7 8 9 3 4 5

   amb el punt d'interès sobre el 3 i l ha de quedar buida.

Es demana dissenyar implementacions d'aquestes operacions que **no usin cap operació primitiva de les llistes** (per això no les recordem en aquest enunciat) **ni iteradors**, sinó que accedeixin directament als atributs de la classe *Llista*.

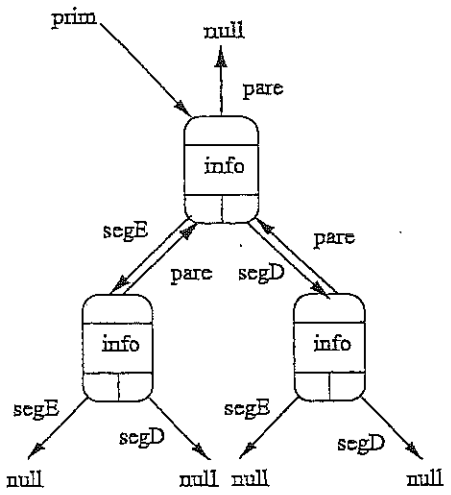
No s'han de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient.

Es valorarà l'eficiència de les solucions proposades.

Problema 2: *Arbres binaris amb punter al pare* (50%)
Una possible extensió de la classe *Arbre* que hem vist pels arbres binaris consisteix en afegir operacions per saber si un arbre té pare (és a dir, si és fill esquerre o dret d'un altre arbre) i, cas que en tingui, obtenir el seu arbre pare. Anomenarem aquesta nova classe *ArbrePlus* i proposem per ella la següent representació basada en nodes:

```
template <class T> class ArbrePlus {
  private:
    struct node_arbrePlus {
        T info;
        node_arbrePlus* segE;
        node_arbrePlus* segD;
        node_arbrePlus* pare;
    };

    node_arbrePlus* prim;
    ... // especificació i implementació d'operacions privades

  public:
    ... // especificació i implementació d'operacions públiques
};
```

Els nodes tenen punters al primer node del fill esquerre (segE), al primer node del fill dret (segD) i a l'arrel del pare (pare), si en tenen; si no, el seu valor és `null` a qualsevol dels tres casos. L'arbre només té un atribut: un punter al primer node (`prim`). Gràficament, la implementació de l'estructura és



Us demanem implementar dues operacions d'aquesta classe, la constructora de còpia i la de plantar un nou arbre, amb les següents especificacions pre/post:

```
ArbrePlus(const ArbrePlus &orig)
/* Pre: cert */
/* Post: el resultat és una còpia de l'arbre orig */

void plantar(const T &x, ArbrePlus &a1, ArbrePlus &a2)
/* Pre: el paràmetre implícit és buit, a1=A1, a2=A2 */
/* Post: el paràmetre implícit té x com a arrel, A1 com a fill
         esquerre i A2 com a fill dret i, per tant, passa a ser
         el pare dels arbres A1 i A2; a1 i a2 són buits */
```

La constructora ArbrePlus només haurà de cridar una operació privada recursiva, que també heu de dissenyar a partir de la següent especificació:

```
node_arbrePlus* còpia_node_arbrePlus(node_arbrePlus* m)
/* Pre: cert */
/* Post: el resultat és NULL si m és NULL; en cas contrari, el
         resultat apunta al node arrel d'una jeraquia de nodes
         que és una còpia de la jerarquia de nodes que té el node
         apuntat per m com a arrel */
```

Es demana dissenyar implementacions d'aquestes operacions que **no usin cap operació primitiva dels arbres** (per això no les recordem en aquest enunciat) sinó que accedeixin directament als atributs de la classe *ArbrePlus*.

No s'han de justificar, però podeu incloure comentaris aclaridors si ho creieu convenient.

Es valorarà l'eficiència de les solucions proposades.

Examen Programació 2

Grau en Informàtica
Gener 2011
Temps estimat: 2h 30m

Problema 1: *Producte de finestres d'un vector* (50%)
Donat un vector d'enters `v` de mida `N>1` i un enter `k` tal que `1<k` i `k<=N`, volem obtenir un nou vector d'enters `w` de mida `N-k+1` on `w[i]` contingui para cada `i` en `[0..N-k]` el *producte* dels elements del subvector `v[i..i+k-1]`, és a dir, d'una finestra de `k` elements de `v` que comença amb `v[i]`.

Es demana dissenyar i justificar una implementació eficient d'aquesta operació, fent servir només iteracions `while`. Tingueu en compte que el vector `v` pot contenir un nombre qualsevol de zeros, amb l'impacte que això pot tenir de cara al càlcul dels productes.

Per exemple, si tenim el vector `v`

-1	4	-7	0	-2	5	-6	0	0	1	3	2

i l'enter `k=3`, el resultat ha de ser el vector `w`

28	0	0	0	60	0	0	0	0	6

L'especificació de l'operació és la següent:

```
vector<int> prod_finestres(const vector<int> &v, int k, int N);
/* Pre:  N=v.size()>1, 1<k<=N */
/* Post: el resultat és un vector de mida N-k+1, que conté a cada posició i
   en [0..N-k] el producte dels elements del subvector v[i..i+k-1] */
```

Al disseny heu d'incloure la **justificació** de tots els elements: invariant, inicialitzacions, condició de sortida, cos del bucle i acabament.

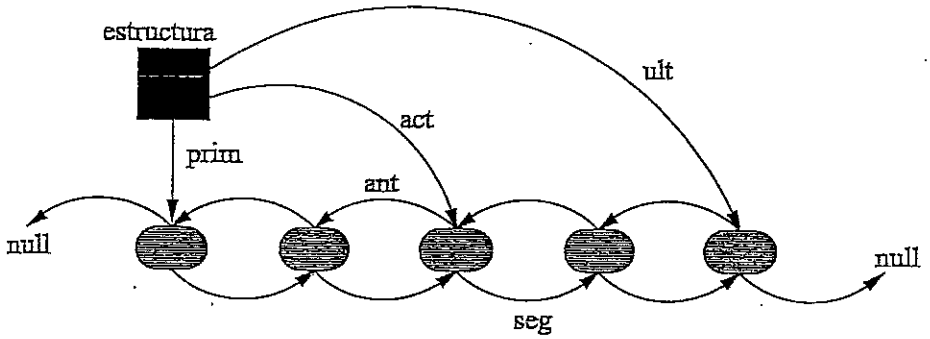
Es valorarà **negativament** fer recorreguts innecessaris dels vectors. Noteu, però, que sempre és millor una solució ineficient correcta, que un intent incorrecte de solució eficient.

Problema 2: *Operacions amb tipus recursius* (50%)
Suposem la següent representació de tipus basada en nodes d'una classe *LlistaInt* per manegar llistes d'enters:

```
class LlistaInt {
  private:
    struct node_llista {
        int info;
        node_llista* seg;
        node_llista* ant;
    };
```

```
    int longitud;
    node_llista* prim;
    node_llista* ult;
    node_llista* act;
    ...  // especificació operacions privades

  public:
    ...  // especificació operacions públiques
};
```

Els nodes són doblement encadenats amb punters al següent (seg) i a l'anterior (ant), i la llista té quatre atributs: la longitud i tres punters a nodes, un pel primer element (prim), un per l'últim (ult) i un altre per l'element actual (act), on tenim situat el punt d'interès de la llista. Gràficament, la implementació de l'estructura és



Volem implementar dins d'aquesta classe una operació nova amb la següent especificació pre/post:

```
LlistaInt reagrupament(int k) const;
/* Pre: cert */
/* Post: el paràmetre implícit no canvia (incloent el valor d'act); el resultat
   conté els mateixos elements que al p.i. però reagrupats de manera que primer
   van tots els que són <= k, en el mateix ordre que al p.i., seguits de tots els
   que són > k, també en el mateix ordre que al p.i.; el punter act del resultat
   apunta al primer node amb valor > k (o a NULL si no existeix cap d'aquests) */
```

Es demana dissenyar una implementació d'aquesta operació que **no usi cap operació primitiva de les llistes** (per això no les recordem en aquest enunciat) **ni iteradors**, sinó que accedeixi directament als atributs de *LlistaInt*.

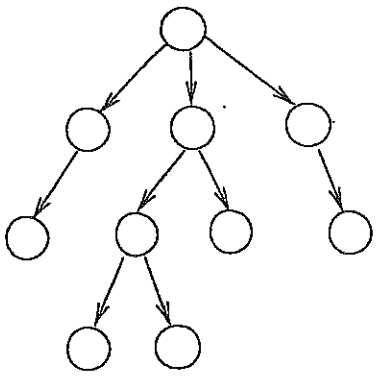
No s'ha de justificar, però es poden incloure comentaris aclaridors si ho creieu convenient.

Examen Pràctiques de Programació

Enginyeria Informàtica
Juny 2010
Temps estimat: 2h 30m

Problema 1: *Arbres maxN-aris: ús* (50%, no hi ha nota mínima)
Un arbre maxN-ari és una estructura arborescent que pot ser buida o bé tenir una arrel i un nombre de fills entre 0 i N, que són també arbres maxN-aris. Si un arbre maxN-ari té m fills, els seus fills estaran numerats entre 1 i m segons l'ordre en que van ser afegits.

Exemple d'arbre maxN-ari (amb $N = 3$):



Com veieu, l'arbre té 10 elements. Alguns elements tenen tres fills, uns altres elements en tenen 2, uns altres 1 i uns altres cap.

Podem especificar el mòdul Arbre maxN-ari de la següent manera:

```
Mòdul Arbre maxN-ari;
Especificació

{Descripcio: permet estructurar la informació de forma arborescent amb màxim N fills.
N és una constant que es pot fer servir al codi}

Tipus Arb_maxnari

func arb_buit() ret a: Arb_maxnari;
{Pre: cert}
{Post: a és un arbre buit}

funcio arb_zero_fills(x: elem) ret a: Arb_maxnari;
{Pre: cert}
{Post: a és un arbre amb arrel x i sense fills }

accio afegir_fill(e/s a: Arb_maxnari; ent f: Arb_maxnari);
{Pre: a no és buit i "nombre de fills d'a" < N}
{Post: a ha estat modificat, afegint-li f com a últim fill}

accio eliminar_ifill(e/s a: Arb_maxnari; ent i: nat);
{Pre: a no és buit i 1<= i <= "nombre de fills d'a"}
{Post: a ha estat modificat, traient-li l'i-èsim fill}
```

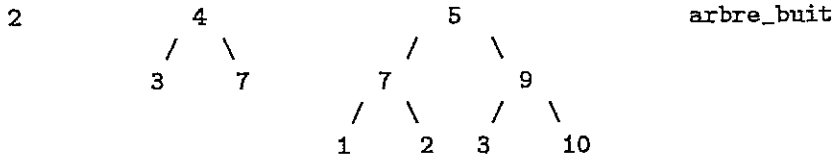
```
funcio ifill (a: Arb_maxnari; i: nat) ret f: Arb_maxnari;
{Pre: a no és buit i 1<= i <= "nombre de fills d'a"}
{Post: f és el fill i-èsim d'a}

funcio arrel (a: Arb_maxnari) ret x: elem;
{Pre: a no és buit}
{Post: x és el valor de l'arrel d'a}

funcio nombre_fills (a: Arb_maxnari) ret n: nat;
{Pre: a no és buit}
{Post: n = nombre de fills d'a}

funcio és_buit (a: Arb_maxnari) ret b: bool;
{Pre: cert}
{Post: b = a és buit}
```

Un arbre maxN-ari es diu **complet** si té tots els nodes possibles a tots els nivells on té nodes o, equivalentment, si cada nivell té 0 o N subarbres del mateix tamany. Usant les operacions de l'especificació d'arbres maxN-aris (de fet, només calen les consultores i no pas totes), implementeu una funció que donat un arbre maxN-ari de naturals ens digui si és complet. Feu servir la següent especificació.

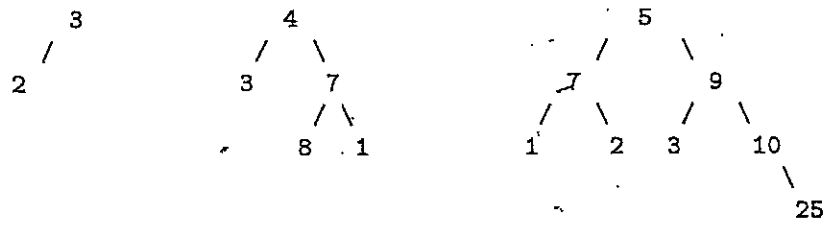
```
funcio  complet (a : Arb_maxnari de nat) ret b : bool
{ Pre: cert}

{ Post: b = a-és complet }
```

Exemples (amb N=2):

Complets (obviament, el segon i el tercer no ho serien si N>2):



No complets:



Noteu que al primer arbre li falta (o sobra) un node per ser complet, al segon li en falten (o sobren) dos i al tercer li en falten set (o li en sobra un).

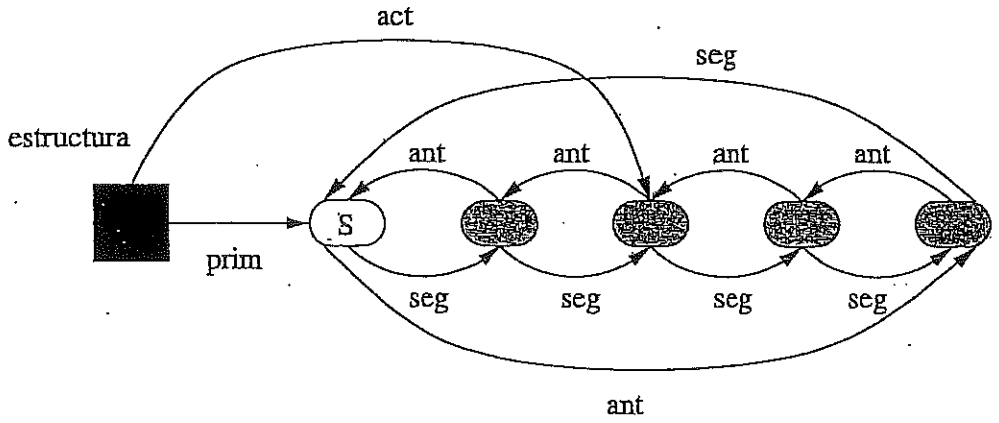
Al disseny heu d'incloure la **justificació** de tots els elements: invariant, inicialitzacions, condició de sortida, cos del bucle i acabament, per les iteracions, i casos base i recursius, validesa de les crides i decreixement, per la recursivitat.

Es valorarà molt l'eficiència de la solució proposada. Per tant, heu d'evitar la repetició de càlculs amb immersions d'eficiència.

Problema 2: *Implementació amb tipus recursius* (50%, no hi ha nota mínima)
Volem implementar una estructura lineal doblement enllaçada amb un element destacat, anomenat *actual*, que és consultable i que es pot moure endavant i enrera seguint els enllaços. A més, volem operacions per afegir i treure elements, a part d'altres manipulacions.

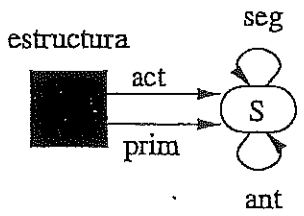
Per simplificar el codi de les operacions, imposem que l'estructura mai tingui punters amb valor null. En tot moment contindrà un node *sentinella*, que no comptarà com a membre de l'estructura i no emmagatzemarà cap info, però servirà perquè tots els nodes estiguin enllaçats.

Gràficament, una possible implementació de l'estructura seria



L'element sentinella (marcat amb S) és sempre el primer de l'estructura. No està prohibit que l'element actual sigui el sentinella però, si l'és, no es pot esborrar ni consultar. Diem que el darrer element de l'estructura és l'anterior del sentinella. A més, definim el propi sentinella com al següent del darrer.

Si l'estructura és buida, el sentinella s'apuntarà a si mateix. Tindrà l'aspecte



Completeu la següent implementació. Notareu que algunes operacions ja es donen fetes com a mostra, per tant només queden per fer les que tenen un ??? al codi. Potser caldria introduir-ne alguna més perquè l'estructura sigui completament operativa, però això se surt de l'abast d'aquest examen.

```
Mòdul str;
Implementació

Tipus str=  tupla                Tipus node= tupla
             prim: node*;                     info: elem;
             act:  node*;                     ant:  node*;
            ftupla;                           seg:  node*;
                                             ftupla;
```

```
func str_buida () ret s:str;
{Pre: cert} var aux: node*;
  nou(aux);  // s'ha de posar el sentinella, pero no cal que tingui info
  *aux.sig := aux;
  *aux.ant := aux;
  s.prim := aux;
  s.act := aux;
{Post: s és una estructura buida}

accio afegir (e/s s:str; ent x:elem);
{Pre: cert}
  ???  // no cal tractar de forma diferent el cas que s sigui buida
{Post: s te un element més per la inserció d'x com a l'anterior de l'element actual}

accio afegir_final (e/s s:str; ent x:elem);
{Pre: cert}
  ???  // no cal tractar de forma diferent el cas que s sigui buida
{Post: s te un element més per la inserció d'x com al següent de l'últim element}

accio eliminar (e/s s:str);
{Pre: s.act != s.prim }
  ???  // no cal tractar de forma diferent el cas que s només tingui un element
{Post: s s'ha modificat eliminant-ne l'element actual;
el nou actual passa a ser l'anterior de l'antic actual }

accio invertir (e/s s:str);
{Pre: cert}
  ???  // no cal tractar de forma diferent el cas que s sigui buida
{Post: s'ha intercanviat la relació d'anterior i següent de tots els elements d's}

accio avançar (e/s s:str);
{Pre: cert}
  s.act := *(s.act).sig;
{Post: el nou actual d's és el següent de l'antic}

accio retrocedir (e/s s:str);
{Pre: cert}
  s.act := *(s.act).ant;
{Post: el nou actual d's és l'anterior de l'antic}

funcio actual (s:str) ret x:elem;
{Pre: s.act != s.prim}
  x := *(s.act).info
{Post: x és el valor de l'element actual d's}

funcio és_buida (s:str) ret b:bool;
{Pre: cert}
  b := s.prim = *(s.prim).sig;
{Post: b = s és buida}
```

Examen Pràctiques de Programació

Enginyeria Informàtica
Gener 2010
Temps estimat: 2h 30m

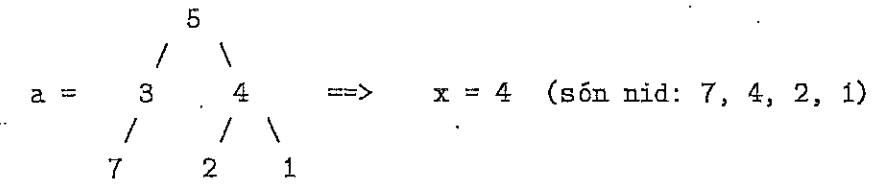
Problema 1: *Nodes individualment dominadors* (50%)

Diem que un node d'un arbre binari de naturals a és **individualment dominador** si cap dels nodes situats a sota seu és estrictament més gran que ell.

Dissenyeu una funció que compti quants nodes individualment dominadors conté un arbre binari de naturals.

func comptar_nid (a : arbre de naturals) **dev** x : enter
{Pre : *cert*}
{Post : x = nombre de nodes individualment dominadors d'a}

Exemple:



Al disseny heu d'incloure la **justificació** de tots els elements: casos base i recursius, validesa de les crides i acabament o decreixement.

Es valorarà molt l'eficiència de la solució proposada. Per tant, heu d'evitar la repetició de càlculs amb immersions d'eficiència.

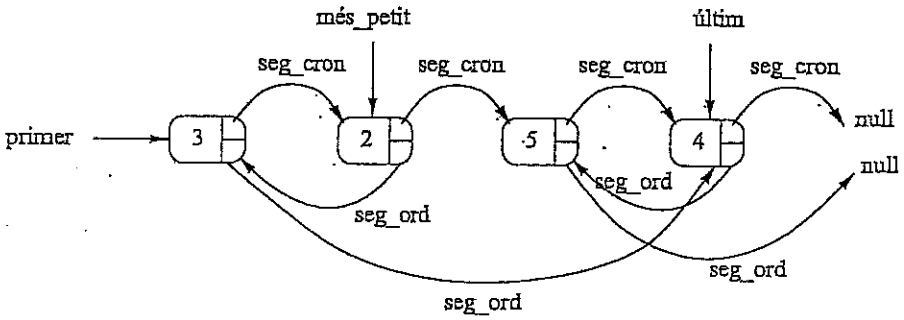
Problema 2: *Cues ordenades* (50%)

Volem modificar el mòdul Cua, per produir cues de nombres enters que tinguin la propietat addicional de poder ser recorregudes en ordre creixent respecte al valor dels seus elements. Per a això, redefinim la implementació amb més punters.

```
Mòdul CuaOrd { Tipus de mòdul: dades }

Implementació

Tipus CuaOrd = tupla
                 long: enter
                 prim: *node_cua
                 ult: *node_cua
                 més_petit: *node_cua // apunta a l'element de la cua
                                      // amb valor més petit, si n'hi ha
               ftupla
```

```
Tipus node_cua = tupla
                   info: enter
                   seg_cron: *node_cua
                   seg_ord: *node_cua
                 ftupla
```

Els punters *primer*, *últim* i *seg_cron* es fan servir per gestionar l'ordre d'arribada a la cua, que anomenem ordre cronològic, mentre que els punters *més_petit* i *seg_ord* es fan servir per gestionar l'ordre creixent segons el valor dels elements. Un exemple gràfic d'una cua amb aquesta implementació seria



Pel que fa a les operacions, només volem implementar-ne dues, concretament l'operació *demanar_torn* i l'operació *concatenar*, especificades de la següent manera.

```
accio demanar_torn (ènt x:enter; e/s c:CuaOrd)
{Pre: cert }
{Post: s'ha afegit x a c com a darrer element a l'ordre cronològic
i on li toqui a l'ordre creixent}
```

```
acció concatenar (e/s c1: CuaOrd, e/s c2: CuaOrd);
{Pre: cert }
{Post: modifica c1 posant després del seu últim element (cronològic)
tots els elements de c2 en el mateix ordre cronològic en el qual hi
eren a c2, i amb tots els elements reorganitzats per satisfer l'ordre
creixent; modifica c2 deixant-la buida}
```

Es valorarà l'eficiència de les solucions proposades. No cal justificar la correcció de les operacions.

Examen Pràctiques de Programació

Enginyeria Informàtica
Juny 2009
Temps estimat: 2h 30m

Problema 1: *Substitucions* (50%)
Sigui N un natural més gran que 0. Considerem dos vectors d'enters v_1 i v_2 amb posicions $[1..N]$, on es compleix que $v_2[1] = 0$. Especifiqueu i dissenyeu una funció que calculi el número k més petit tal que després de substituir per k tots els zeros de v_2, la suma de cada prefix no buit de v_2 sigui més gran o igual que la suma del corresponent prefix de v_1.
Teniu en compte que els vectors $[1..N]$ tenen N prefixos no buits: $[1..1]$, $[1..2]$, ..., $[1..N]$.
Exemple: amb els vectors $v_1, v_2[1..6]$

posició:	1	2	3	4	5	6
v1:	1	-4	3	12	-6	-9
v2:	0	3	0	-2	3	0

el valor k ha de ser 6 ja que, en aquest cas, totes les sumes dels prefixos de v_2 superen a les seves corresponents de v_1

posició:	1	2	3	4	5	6
sumes prefixos v1:	1	-3	0	12	6	-3
sumes prefixos v2 amb k=6:	6	9	15	13	16	22

i noteu que amb valors més petits de k falla sempre la suma del prefix $[1..4]$.

Al disseny heu d'incloure la justificació de tots els elements: casos base i recursius, validesa de les crides i acabament o decreixement, si les funcions són recursives, i invariant, inicialitzacions, condició de sortida, cos del bucle i acabament si són iteratives.
Es valorarà molt negativament fer recorreguts innecessaris dels vectors.

Problema 2: *Sumes selectives en una matriu* (50%)
Sigui m una matriu d'enters amb posicions $[1..N, 1..M]$. Volem calcular la suma màxima que es pot obtenir amb elements de m amb les següents condicions:

1. de cada fila, es pot sumar un element com a màxim
2. de la primera fila, es pot sumar qualsevol element (o cap)
3. per a cada element sumat $m[i,j]$, els elements sumats de les files més grans que i han de pertànyer a columnes més grans que j.

Exemple: amb la matriu $m[1..3, 1..6]$

```
fila 1:   5    -7    4   -1    3    6

fila 2:   1    -4    3   12   -6   -9

fila 3:  10     3    8   -2    3    2
```

la suma màxima sota les condicions demanades és 20, és a dir, $5 + 12 + 3$. Noteu, però, que si el 8 de la tercera fila fos un 18, llavors la suma màxima seria 23, és a dir, $5 + 18$.

Us proporcionem l'especificació pre/post de l'operació que heu d'implementar.

```
funcio suma_max_sel (m: matriu [1..N, 1..M] d'enters, i,j: enter) ret max: enter
{Pre: 1<=i<=N+1, 1<=j<=M+1 }

{Post: max = suma màxima que es pot obtenir amb elements de m[i..N, j..M] amb les
            condicions de l'enunciat}
```

Aquesta funció calcula el valor buscat amb la crida `suma_max_sel(m,1,1)`.

Al disseny heu d'incloure la justificació de tots els elements: casos base i recursius, validesa de les crides i acabament o decreixement, de totes les funcions recursives, i invariant, inicialitzacions, condició de sortida, cos del bucle i acabament de totes les iteracions.

Es valorarà l'eficiència de la vostra funció. En particular, s'ha d'evitar fer sumes i comparacions innecesàries.

Examen Pràctiques de Programació

Enginyeria Informàtica
Gener 2009
Temps estimat: 2h 30m

Problema 1: *Canvis de signe* (50%)
Considerem un vector $v[1..N]$ d'enters. Anomenem *bagatge* d'un element $v[i]$ a la suma dels elements anteriors amb ell mateix inclòs, és a dir, la suma del subvector $v[1..i]$. Volem una funció que calculi quants elements del vector v tenen un bagatge de signe diferent al del bagatge del seu element anterior.

Teniu en compte que:

1. L'element $v[1]$ no té element anterior.
2. Per simplificar, suposem que el 0 té signe positiu.

Exemple: al vector $v[1..6]$

```
posició: 1   2   3   4   5   6

valor:   1  -4   3  12  -6  -9
```

hi ha exactament 3 elements amb canvi de signe respecte al bagatge:

```
v[2]: bagatge seu = -3 ; bagatge anterior = 1
v[3]: bagatge seu = 0; bagatge anterior  = -3
v[6]: bagatge seu = -3; bagatge anterior = 6
```

Podeu fer servir la següent especificació:

func canvis_de_signe (v : vector $[1..N]$ d'enter) dev csb : natural
{Pre : $N > 0$}

{Post : csb = nombre de canvis de signe del bagatge dels elements de v}

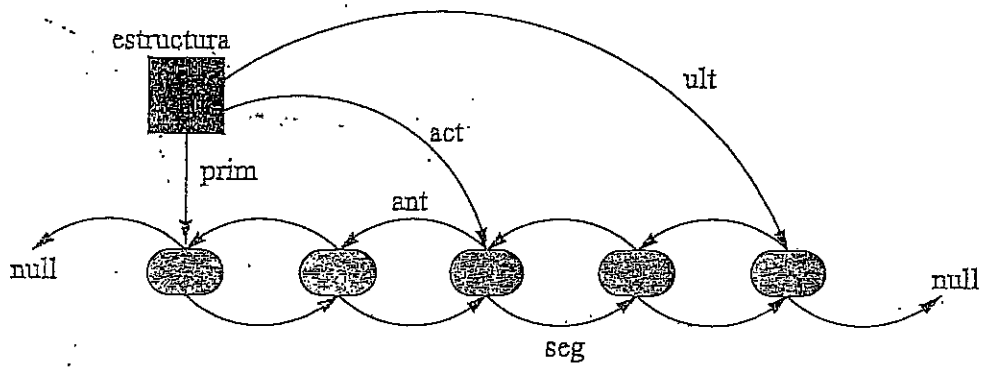
Al disseny heu d'incloure la justificació de tots els elements: casos base i recursius, validesa de les crides i acabament o decreixement, si les funcions són recursives, i invariant, inicialitzacions, condició de sortida, cos del bucle i acabament si són iteratives.

Es valorarà molt negativament fer recorreguts innecessaris dels vectors.

Problema 2: *Operacions amb tipus recursius* (50%)

Tenim implementada una estructura lineal que permet *afegir* i *eliminar* nombres naturals pels seus extrems. L'*últim* i el *primer* elements són respectivament l'últim i el primer que s'han afegit. A més, si l'estructura no és buida, manté sempre un element, anomenat *actual*, que és l'únic consultable i que es pot moure endavant i enrera seguint l'ordre seqüencial en el qual els elements van ser afegits a l'estructura.

Gràficament, la implementació de l'estructura és



El codi per implementar els tipus és

```
Mòdul str;
Implementació

Tipus str= tupla              Tipus node= tupla
            act: *node;                   info: *nat;
            prim: *node;                  ant: *node;
            ult: *node;                   seg: *node;
           ftupla;                      ftupla;
```

Volem afegir-hi tres operacions. Una que concateni dues estructures, una altra que inverteixi una estructura i una tercera que donada una estructura n'elimini tots els elements amb info igual a un valor donat x.

Us proporcionem les especificacions pre/post de les operacions que heu d'implementar.

```
acció concatenar (e/s s1: str, e/s s2: str)
{Pre: cert }

{Post: afegeix els elements de s2 al final de s1, amb l'ordre original;
l'element actual passa a ser el primer de l'estructura resultant,
si n'hi ha algun; s2 es converteix en buida}

acció invertir (e/s s: str);
{Pre: cert }

{Post: modifica s convertint l'últim element en el primer (i viceversa) segons
l'ordre seqüencial en el qual els elements van ser afegits a l'estructura i
alterant conseqüentment la relació d'anterior i següent de tots els elements;
l'element actual no varia}

acció eliminar (e/s s: str; ent x: nat);
{Pre: cert }

{Post: modifica s traient-ne tots els elements amb info igual a x; l'element
actual passa a ser el primer de l'estructura resultant, si n'hi ha algun}
```

Examen Pràctiques de Programació

Enginyeria Informàtica
Juny 2008
Temps estimat: 2h 30m

Problema 1: *Classificar els elements d'un vector* (50%)
Considerem les constants naturals M i N. Sigui v un vector $[1..M]$ de naturals i d un natural. Hem de classificar els valors que apareixen a v en $N+1$ categories. Les N primeres són de tamany d i la $N+1$ conté els valors que no entren en cap de les altres categories. La primera categoria conté els valors entre 0 i $d-1$. A partir d'aquí, cada categoria, excepte l'última, conté els d següents números respecte a l'anterior.

Per exemple, si $N=6$ i $d=5$, les categories són: la primera de 0 a 4, la segona de 5 a 9, la tercera de 10 a 14, la quarta de 15 a 19, la cinquena de 20 a 24, la sisena de 25 a 29, i la setena els més grans o igual a 30.

Especifiqueu i dissenyeu una funció que donat un vector v d'aquest tipus i un valor d, retorni un vector de naturals w, indexat amb $[1..N+1]$, on a cada posició k aparegui el nombre d'elements que hi ha de la k-éssima categoria.

Per exemple, amb $M=12$, $N=6$ i $d=5$, si tenim al vector v els valors:

posició	1	2	3	4	5	6	7	8	9	10	11	12
valor	2	14	7	25	12	10	27	10	53	4	16	18

el vector w ha de tenir $w[1]=2$, $w[2]=1$, $w[3]=4$, $w[4]=2$, $w[5]=0$, $w[6]=2$ i $w[7]=1$.

Al disseny heu d'incloure l'**especificació** i la **justificació** de tots els elements: casos base i recursius, validesa de les crides i acabament o decreixement, si les funcions són recursives, i invariant, inicialitzacions, condició de sortida, cos del bucle i acabament si són iteratives.

Es valorarà molt negativament fer recorreguts innecessaris dels vectors i, en general, tota repetició innecessària de càlculs.

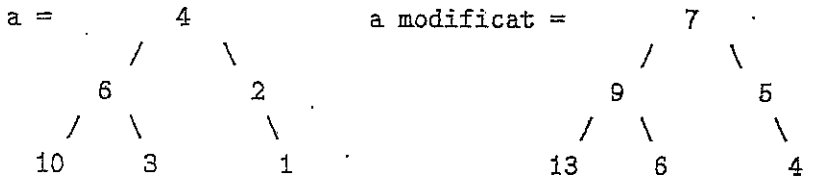
Problema 2: *Implementacions directes d'operacions d'arbres* (50%)
Considereu la implementació d'arbres binaris amb tipus recursius vista durant el curs.

```
Tipus arbre = tupla
                primer_node : *node_arbre
              ftupla

Tipus node_arbre = tupla
                     info: ELEM
                     segI: *node_arbre
                     segD: *node_arbre
                   ftupla
```

Suposem que el tipus arbnat és el resultat de substituir el tipus ELEM pel tipus nat (números naturals) en aquesta implementació. Volem dues noves operacions bàsiques, que han de ser implementades directament sobre l'estructura, sense usar les operacions del mòdul arbre. La primera operació és:

```
accio inc (e/s a: arbnat, entrada k: nat);
{Pre: --- }

{Post: a ha estat modificat incrementant en k tots els seus valors }
```

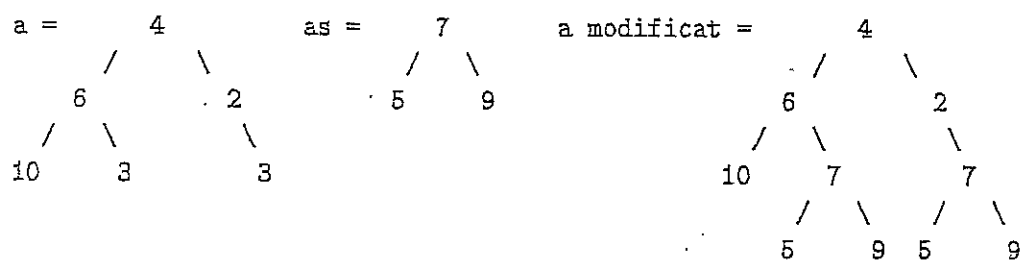
Exemple: si k=3,



La segona operació és:

```
accio subst (e/s a: arbnat, entrada x:nat, entrada as:arbnat);
{Pre: --- }

{Post: a ha estat modificat substituint les fulles que contenen x per
       l'arbre as }
```

Exemple: si x=3,



Examen Pràctiques de Programació

Enginyeria Informàtica
Gener 2008
Temps estimat: 2h 20m

Problema 1: *Comptatge de dominadors* (50%)
Considerem les constants naturals M i N. Sigui v un vector $[1..M]$ de parells de naturals, on cada parell conté un identificador i un pes. Els identificadors tenen valors entre 1 i N i els pesos tenen valors qualssevol. Els identificadors poden estar repetits qualsevol nombre de vegades.

Diem que un element $v[i]$ és un *dominador* si el seu pes és mes gran o igual que la suma dels pesos dels elements anteriors (és a dir, els de $v[1..i-1]$) que tenen el mateix identificador que ell.

Dissenyeu una funció que donat un vector v d'aquest tipus, retorni un vector de naturals w, indexat amb $[1..N]$, on a cada posició k aparegui el nombre d'elements dominadors d'v amb identificador k.

Per exemple, amb $M = 5$ i $N = 3$, si tenim al vector v els parells:

```
  1     2     3     4     5
(2,4) (3,4) (2,7) (1,5) (3,1)
```

el vector w ha de tenir $w[1] = 1$, $w[2] = 2$ i $w[3] = 1$.

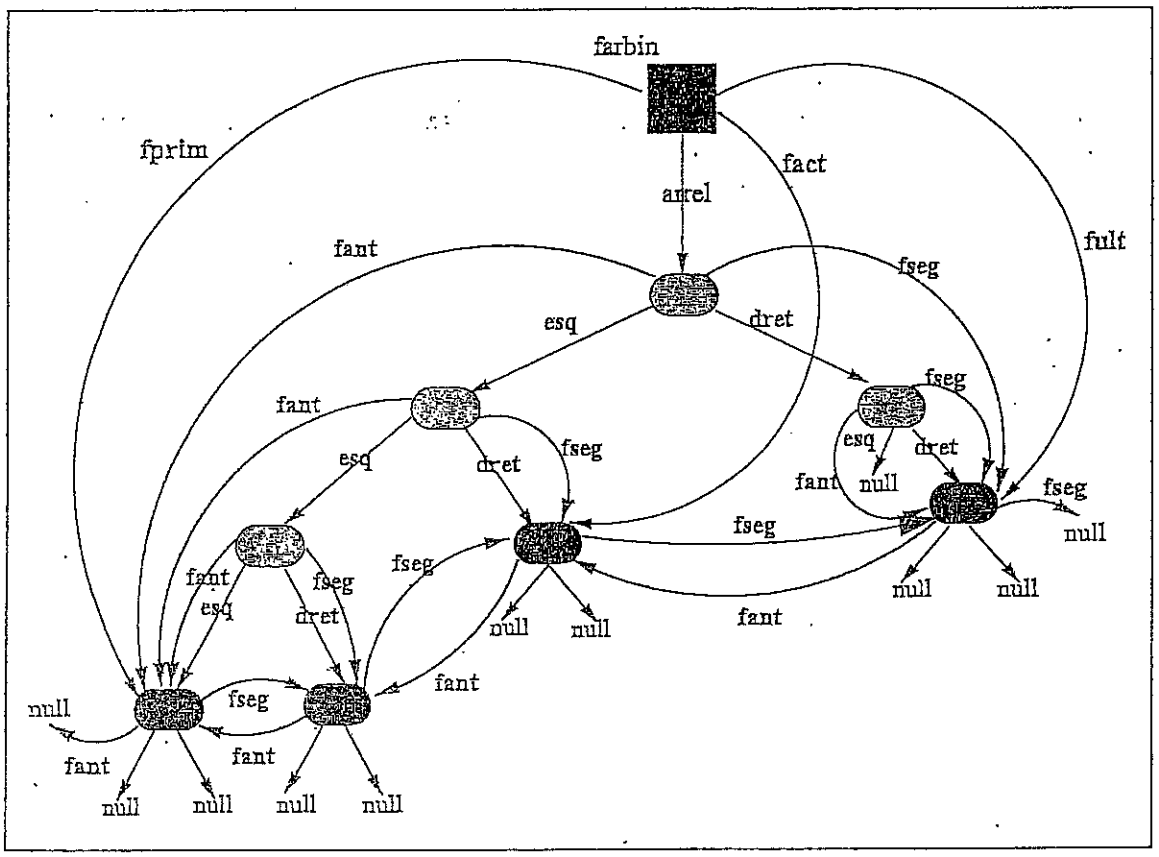
Podeu fer servir la següent especificació:

```
tipus parell = tupla
                 id  : nat;
                 pes : nat;
               ftupla;

func comptatge_dominadors (v : vector [1..M] de parell) dev w : vector [1..N] de natural
{Pre : — — —}

{Post : a cada posició d'w es troba el nombre d'elements dominadors d'v
amb identificador igual a tal posició }
```

Al disseny heu d'incloure la **justificació** de tots els elements: casos base i recursius, validesa de les crides i acabament o decreixement, si les funcions són recursives, i invariant, inicialitzacions, condició de sortida, cos del bucle i acabament si són iteratives.

Es valorarà molt negativament fer recorreguts innecessaris dels vectors.

Problema 2: *Arbres i fulles* (50%)
Volem implementar una estructura d'arbre binari amb les operacions usuals però que, a més, permet accedir i realitzar operacions directament a les fulles. Per això, si l'estructura no és buida, manté sempre un element, anomenat *fact*, que és l'única fulla consultable i que es pot situar a la primer fulla i moure's d'esquerra a dreta seguint l'ordre seqüencial de les fulles. Gràficament, una possible implementació de l'estructura seria



Noteu que els atributs *fant* i *fseg*, representen coses diferents si estem en una fulla o no. En cas de ser una fulla apunten respectivament a la fulla anterior i a la fulla següent d'esquerra a dreta. En cas de ser un node intern *fant* apunta a la primera fulla del subarbre que penja d'ell i *fseg* apunta a l'última fulla del subarbre que penja d'ell, sempre d'esquerra a dreta.

Direm que la fulla *següent* de l'actual és la primera fulla que ens trobem si recorrem l'arbre d'esquerra a dreta. Així, *avançar* la fulla actual és canviar-la per la seva fulla següent. Noteu que aquesta operació no és aplicable si l'arbre és buit o l'actual no té següent. Les operacions que es demana que implementeu són (algunes operacions típiques no es demanen per ser molt similars a d'altres):

crear un *arbre buit*	*plantar* un element sobre dos arbres
obtenir *fill esquerre*	demanar si *és buit.*
situar l'actual en la *primera fulla*	*avançar* la fulla actual
consultar si l'actual *té següent*	*consultar l'actual*

Cognoms i Nom: DNI:

Ompliu els espais en blanc amb les instruccions necessàries per implementar les operacions de l'estructura definida anteriorment que apareixen en aquest full (per les dues cares) i el següent. No podeu usar més espai que el que hi ha disponible en cada operació.

```
Mòdul farbin;
Implementació

Tipus farbin= tupla              Tipus node= tupla
        arrel: *node;                    inf: elem;
        fprim: *node;                    esq: *node;
         fact: *node;                   dret: *node;
         fult: *node;                   fseg: *node;
       ftupla;                          fant: *node;
                                       ftupla;
```

1) `func arbre_buit() ret a: farbin;`
   `{Pre: --- }`

   `{Post: a es un arbre sense elements (buit)}`

2) `accio fill_esq (e/s a: farbin);`
   `{Pre: a no es buit }`

   `{Post: a es converteix en el seu fill esquerre`
   `l'actual passa a ser la primera fulla de l'a resultant }`

3) func es_buit (a: farbin) ret b: bool;
{Pre: --- }

{Post: b ens diu si a es buit o no }

4) accio plantar(ent x: elem; e/s aI: farbin; e/s aD: farbin);
{Pre: cert}

{Post: el nou aI té x com a arrel, l'aI original com a fill esquerre i aD com a fill dret, aD es destrueix i l'actual passa a ser la primera fulla de l'aI resultant }

Cognoms i Nom: DNI:

5) accio primera_fulla(e/s a: farbin);
{Pre: a no es buit }

{l'actual passa a ser la primera fulla d'a }

6) accio avançar(e/s a: farbin);
{Pre: a no es buit i actual d'a té següent}

{l'actual passa a ser la fulla següent d'a }

7) func té_següent (a: farbin) ret b: bool;
{Pre: a no es buit }

{Post: b ens diu si l'actual d'a té següent o no }

8) func actual (a: farbin) ret x : elem;
{Pre: a no es buit }

{Post: x es l'element actual d'a }

Examen Pràctiques de Programació

Enginyeria Informàtica
Juny 2007
Temps estimat: 2h 30m

Problema 1: *Distribució justa de cues* (50%) Considerem una cua de parells de naturals, on cada parell conté un identificador i un pes. La cua representa persones (*identificadors*) que esperen per fer un tràmit burocràtic que té un cost en temps (*pes*). En un determinat moment s'obre una nova finestreta d'atenció al públic i les persones de la cua original s'han de distribuir de forma justa en dues noves cues tenint en compte el cost en temps associat als tràmits.

Una distribució es justa si cap persona ha d'esperar més que una altra que tenia darrera en la cua original i tothom espera el mínim temps possible.

Exemple:

```
          cua original       cua nova 1       cua nova 2
primer --->( 3,2)              ( 3,2)           ( 6,3)
             ( 6,3)            ( 2,5)           (11,1)
             ( 2,5)            ( 5,3)           ( 8,4)
             (11,1)            ( 1,3)           ( 9,2)
             ( 8,4)            (15,2)           ( 7,4)
             ( 5,3)                             ( 4,3)
             ( 9,2)
             ( 1,3)
             ( 7,4)
             (15,2)
últim  --->( 4,3)
```

Dissenyeu una funció que retorni una distribució justa d'una cua de parells, amb la següent especificació:

tipus parell = tupla
id : nat;
pes : nat;
ftupla;

func distribució (*co* : cua de parell) dev *c1*, *c2* : cua de parell
{Pre : — — —}

{Post : *c1* i *c2* són una distribució justa de *co*}

Al disseny heu d'incloure la justificació de tots els elements: casos base i recursius, validesa de les crides i acabament o decreixement, si les funcions són recursives, i invariant, inicialitzacions, condició de sortida, cos del bucle i acabament si són iteratives.

Es valorarà molt l'eficiència de la solució proposada. Per tant, heu d'evitar la repetició de càlculs.

Problema 2: *Mínim comú ancestre* (50%)
Donat un arbre a d'enters, diem que z es un *ancestre* d'x dins d'a si x apareix en el subarbre de a que té z com a arrel. Noteu que si x apareix dins d'a llavors x és un ancestre d'x dins d'a. Diem que z és un ancestre comú d'x i d'y dins d'a si z és ancestre d'x i d'y dins d'a. Finalment, diem que z és el mínim comú ancestre (*mca*) d'x i d'y dins d'a si és un ancestre comú dels dos i no hi ha cap altre ancestre comú d'x i d'y en els fills del subarbre d'a que té z com a arrel.

Dissenyeu una funció que retorni el mínim ancestre comú de dos enters en un arbre, amb la següent especificació:

func mca (a : arbre d'enters, x, y : enter) dev z : enter
{Pre : x i y apareixen només un cop dins d'a}
{Post : z és el mínim comú ancestre d'x i d'y dins d'a}

Exemples:

```
            5
          /   \
a =     2     -4
       / \    / \
     -3   9  1   7
     / \
     6 4
```

```
mca(a,-3,-4) =  5
mca(a, 9, 6) =  2
mca(a, 1, 7) = -4
mca(a, 6,-3) = -3
mca(a, 2, 4) =  2
mca(a,-3,-3) = -3
```

Al disseny heu d'incloure la justificació de tots els elements: casos base i recursius, validesa de les crides i acabament o decreixement, si les funcions són recursives, i invariant, inicialitzacions, condició de sortida, cos del bucle i acabament si són iteratives.

Es valorarà molt l'eficiència de la solució proposada. Per tant, heu d'evitar la repetició de càlculs.

Examen Pràctiques de Programació

Enginyeria Informàtica
Gener 2007. Temps estimat: 2h 30m

Problema 1: *Comparació d'arbres binaris* (50 %)

Diem que un arbre binari de naturals a és igual que un altre arbre binari de naturals b si tenen la mateixa forma (és a dir, exactament les mateixes posicions amb valors) i a cada posició tenen el mateix valor.

Per definir que un arbre binari de naturals a és menor que un altre arbre binari de naturals b, farem servir la suma dels seus elements. Si la suma dels elements d'a es menor que la suma dels elements de b, llavors diem que a és menor que b. En cas que les sumes siguin iguals, i cap dels dos arbres no siguin nuls, diem que a és menor que b si el fill esquerre d'a és menor que el fill esquerre d'b, amb aquesta mateixa definició. Si el fill esquerre d'a és igual que el fill esquerre d'b, diem que a és menor que b si el fill dret d'a és menor que el fill dret d'b, amb aquesta mateixa definició.

Noteu que tot parell d'arbres a i b que tenen la mateixa forma són comparables, és a dir, a és menor que b, b és menor que a, o són iguals. Fixeu-vos també que si la suma de dos arbres és diferent, llavors la comparació ja queda determinada, però si és igual, llavors qualsevol resultat és encara possible.

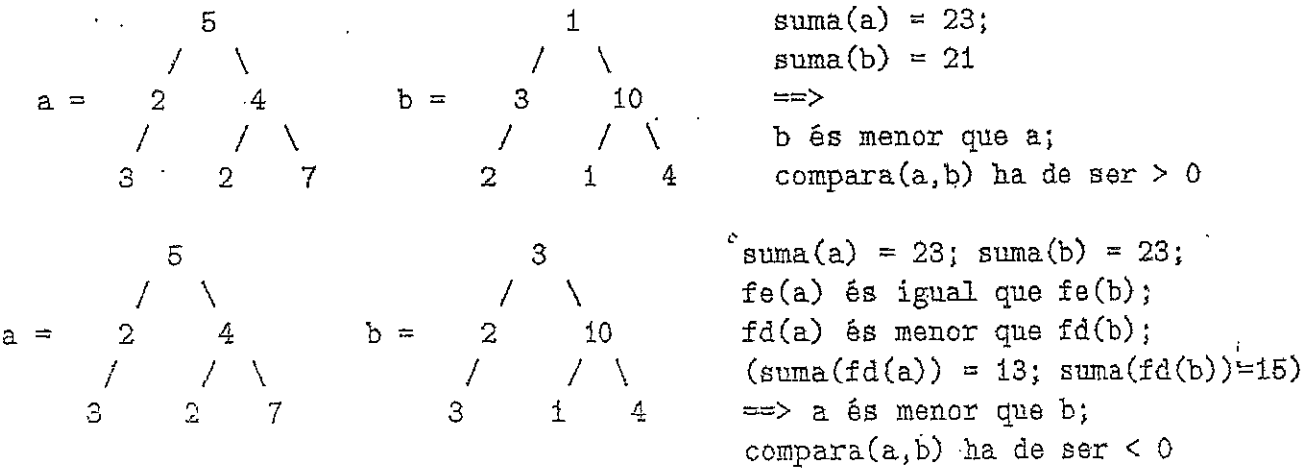
Dissenyeu una funció que compari dos arbres binaris de igual forma, amb la següent especificació:

func compara_arbres (a, b : arbre de naturals) dev x : enter
{Pre : a i b tenen la mateixa forma}
{Post : $x < 0$ si a és menor que b; $x = 0$ si a i b són iguals; $x > 0$ si b és menor que a}

Al disseny heu d'incloure la justificació de tots els elements: casos base i recursius, validesa de les crides i acabament o decreixement.

Es valorarà molt l'eficiència de la solució proposada. Per tant, heu d'evitar la repetició de càlculs amb immersions d'eficiència.

Exemples:



Problema 2: *Camins entre dues posicions d'un vector de cues* (50 %).
Sigui v un vector $[1..N]$ de cues, amb la propietat que a cada posició i del vector hi ha una cua que només pot contenir valors a l'interval $[i+1..N]$, ordenats de forma estrictament creixent.

Donats dos valors i, j de l'interval $[1..N]$, tals que $i \leq j$, diem que un subconjunt ordenat de valors de l'interval $[1..N]$, per exemple $x_1, x_2, \cdots, x_n$, és un camí entre les posicions i i j de v si

- $x_1 = i$ i $x_n = j$ (el primer és i i el darrer és j)
- per a tot k, amb $1 \leq k < n$, el valor x_{k+1} apareix a la cua de la posició $v[x_k]$.

Dissenyeu una funció que donat un vector de cues v i dos valors i, j en les anteriors condicions, ens digui si existeix un camí des de la posició i fins a la posició j de v.

Feu servir la següent especificació

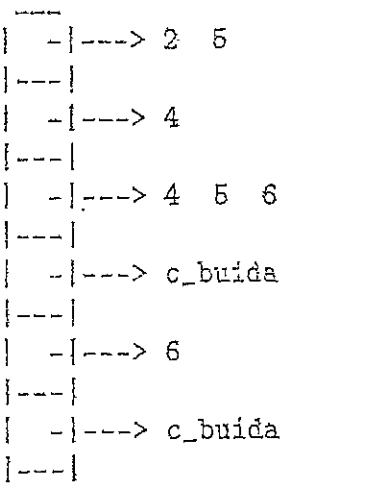
func camí (v : vector $[1..N]$ de cues, i, j : nat) dev b : bool
{Pre : $1 \leq i \leq j \leq N \wedge$ per a tot $\alpha \in [1..N]$ la cua de la posició $v[\alpha]$ només conté elements més grans que α, tots diferents i ordenats creixentment}

{Post : $b =$ existeix un camí entre la posició i i la posició j de v}

Al disseny heu d'incloure la justificació de tots els elements: casos base i recursius, validesa de les crides i acabament o decreixement, de totes les funcions recursives, i invariant, inicialitzacions, condició de sortida, cos del bucle i acabament de totes les iteracions.

No cal que optimitzeu al màxim l'eficiència de la vostra funció, però no s'admet la solució *força bruta*, és a dir, generar una per una totes les combinacions de valors i provar si alguna és un camí vàlid mitjançant una funció auxiliar.

Exemple:




```
v és un vector [1..6] de cues

Hi ha camí a v entre 1 i 6: 1, 5, 6

No hi ha camí a v entre 1 i 3
```

Examen Pràctiques de Programació

Enginyèria Informàtica
Juny 2006
Temps estimat: 2h 20m

Problema 1: *Vector equilibrat* (50%)
Sigui v un vector $[1..N]$ de naturals. Donat un valor i entre 0 i N, definim el *prefix* de v de mida i com el subvector $v[1..i]$. Anàlogament, definim el *sufix* de v de mida i com el subvector $v[N-i+1..N]$. Noteu que els casos extrems són quan $i = 0$, on resulten subvectors buits, i quan $i = N$, on resulten subvectors idèntics al vector original.

Diem que v és un *vector equilibrat* si, per a tot i entre 0 i N, la diferència entre el nombre de valors del prefix de mida i de v que no valen zero i el nombre de valors del sufix de mida i de v que no valen zero es menor o igual que un.

Dissenyeu una funció que calculi si v és un vector equilibrat. Al disseny heu d'incloure la justificació de tots els elements: casos trivials i recursius i acabament o decreixement, si ho feu recursivament, o invariant, inicialitzacions, condició de sortida, cos del bucle i acabament si ho feu iterativament.

Es valorarà molt negativament fer recorreguts innecessaris del vector.

Problema 2: *Implementació de noves operacions de piles* (50%)
Volem afegir al mòdul Pila amb altura un parell d'operacions més, concretament l'operació *concatenar* i l'operació *freqüència*. Per a la operació *freqüència* suposarem que el tipus elem té una operació d'igualtat, que denotarem com es habitual amb el símbol =. L'especificació quedaria ampliada de la següent manera

```
Mòdul Pila { Tipus de mòdul: dades }
Especificació

Paràmetre formal elem;

Tipus pila;

{ Descripcio general: És una estructura que conté elements de tipus
    elem i que permet consultar i eliminar només l'últim element afegit.
    El tipus elem té una operació d'igualtat '=' }


Operacions
```

```
acció concatenar (e/s p1, e/s p2: pila);
{Pre: cert }
{Post: modifica p1 posant-li tots els elements de p2 a sota en el
mateix ordre i modifica p2 deixant-la buida}


funció freqüència (x:elem, p: pila) ret n: nat;
{Pre: cert }
{Post: n = 'nombre d'aparicions d'x a p'}

FEspecificacio
```

Recordem la implementació del tipus Pila que ja coneixem i que heu de fer servir como base per a aquest exercici.

```
Mòdul Pila { Tipus de mòdul: dades }

Implementació

Paràmetre formal elem;


Tipus pila = tupla
               alt: ent
               prim: *node_pila
             ftupla

Tipus node_pila = tupla
                    info: ELEM
                    seg: *node_pila
                  ftupla
```

Examen Pràctiques de Programació. Juny 2006. Problema 2

Cognoms i Nom: DNI:

Apartat 2.1 Usant aquesta implementació del tipus pila amb altura, implementeu les dues noves operacions, concatenar i freqüència. Poseu instruccions on hi ha fletxes (->) i poseu els paràmetres que faltin a les crides. Tingueu en compte que, en principi, existeix una solució on només cal posar una instrucció (una assignació o un Si amb una sola assignació i sense Sino) per a cada fletxa que trobeu. Si us cal, podeu posar més instruccions que fletxes, però no supereu l'espai total que hi ha disponible en cada operació. No feu servir les operacions de pila (empilar, desempilar, etc.) en la implementació. No cal justificar cap de les operacions que heu d'implementar.

```
acció concatenar (e/s p1, e/s p2: pila);
{Pre: cert } var aux: *node_pila fvar;

    ->
  sino
    aux:=p1.prim;
    Mentre *aux.seg != nul fer
     ->

    fMentre

    ->
  fSi

  ->
  ->
  ->

{Post: modifica p1 posant-li tots els elements de p2 a sota en el
mateix ordre i modifica p2 deixant-la buida}

funció freqüència (x:elem, p: pila) ret n:nat;
{Pre: cert }
  n:=freq_node(                    );
{Post: n = 'nombre d'aparicions d'x a p'}

funció freq_node (x:elem, np: *node_pila) ret n:nat;
{Pre: --- }
  Si np = nul llavors

    ->
  sino

    ->
    ->
  fSi
{Post: n = 'nombre d'aparicions d'x als nodes connectats des d'np'}
```

Apartat 2.2 Si heu implementat correctament l'operació `concatenar`, us n'haureu adonat de que es podria fer de forma molt eficient si a la representació d'una pila es disposés de una mica més d'informació. Modifiqueu lleugerament la implementació del tipus `Pila` i feu una nova implementació de l'operació `concatenar` que sigui molt més eficient que l'anterior. Feu-ho en l'espai que teniu en aquest full.

Examen Pràctiques de Programació

Enginyeria Informàtica
Gener 2006
Temps estimat: 2h 20m

Problema 1: *Pila continguda* (50%, no hi ha nota mínima)
Siguin p i q dues piles de naturals sense elements repetits. Dissenyeu una funció recursiva que calculi si p està continguda en q. Considerem que p està continguda en q si cada element de p és a q, i en el mateix ordre relatiu.
Exemple: Considereu les piles p1, p2, p3 i q:

p1	p2	p3	q
5	5	5	6
1	1	1	5
7	7	8	8
	4	7	1
			7
			0

Es donen les següents situacions:

- p1 està continguda en q.
- p2 no està continguda en q (falta el 4).
- p3 no està continguda en q (els elements no estan en el mateix ordre).

Feu servir l'especificació

func incl (p, q : *pila*) dev b : *boolea*
{Pre : ni p ni q no tenen elements repetits }
{Post : b = tots els elements de p estan inclosos a q en el mateix ordre relatiu}

Al disseny heu d'incloure la justificació de tots els elements: casos trivials i recursius i acabament o decreixement.

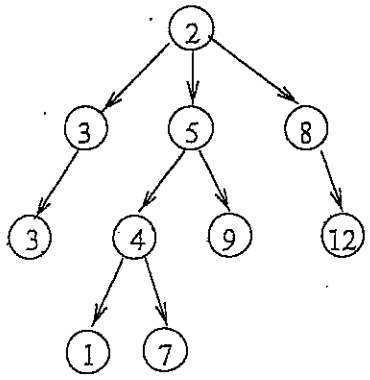
Es valorarà molt negativament fer recorreguts innecessaris de les piles.

Podeu fer servir les següents operacions de piles. Tota altra operació l'heu d'especificar, dissenyar i justificar.

```
funcio pila_buida() ret p: pila
funcio empilar(x: elem; p: pila) ret q: pila
funcio desempilar(p: pila) ret q: pila
funcio cim (p: pila) ret x : elem
funcio es_buida (p: pila) ret b: bool
```

Problema 2: *Arbres maxN-aris: ús i implementació* (50%, no hi ha nota mínima)
Un arbre maxN-ari és una estructura arborescent que pot ser buit o bé tenir una arrel i fins a un màxim de N fills (això inclou el cas de no tenir fills), que són també arbres maxN-aris. Si un arbre maxN-ari té m fills, els seus fills estaran numerats entre 1 i m segons l'ordre en que van ser afegits.

Exemple d'arbre maxN-ari (amb $N = 3$):



Com veieu, l'arbre té 10 elements. L'arrel és 2. Alguns elements tenen tres fills, uns altres elements en tenen 2, uns altres 1 i uns altres cap.

Podem especificar el mòdul Arbre maxN-ari (amb $N = 100$) de la següent manera:

```
Mòdul Arbre maxN-ari;
Especificació

{ Descripcio: permet estructurar la informacio de forma
              arborescent amb màxim N fills, amb N=100}
Const N=100;

Tipus Arb_maxnari

func arb_buit() ret a: Arb_maxnari;
{Pre: cert}
{Post: "a" és un arbre buit}

funcio arb_zero_fills(x: elem) ret a: Arb_maxnari;
{Pre: cert}
{Post: "a" és un arbre amb arrel "x" i sense fills }

accio afegir_fill(e/s a: Arb_maxnari; ent f: Arb_maxnari);
{Pre: "a" no és buit i nombre de fills d'"a"< N}
{Post: "a" ha estat modificat, afegint-li "f" com a últim fill}
```

```
accio eliminar_ifill(e/s a: Arb_maxnari; ent i: nat);
{Pre: "a" no és buit i i<= "i" <= nombre de fills d'"a"}
{Post: "a" ha estat modificat, traient-li l'iessim fill}

funcio ifill (a: Arb_maxnari; i: nat) ret f: Arb_maxnari;
{Pre: "a" no és buit i i<= "i" <= nombre de fills d'"a"}
{Post: "f" és el fill iessim d'"a"}

funcio arrel (a: Arb_maxnari) ret x: elem;
{Pre: "a" no és buit}
{Post: "x" és el valor de l'arrel d'"a"}

funcio nombre_fills (a: Arb_maxnari) ret n: nat;
{Pre: "a" no és buit}
{Post: n = (nombre de fills d'"a")}

funcio és_buit (a: Arb_maxnari) ret b: bool;
{Pre: cert}
{Post: b = ("a" és buit)}
```

Apartat 2.1 Usant les operacions de l'especificació d'arbres maxN-aris, implementeu una funció que donat un arbre maxN-ari de naturals retorni l'element de valor màxim que aparegui a l'arbre.

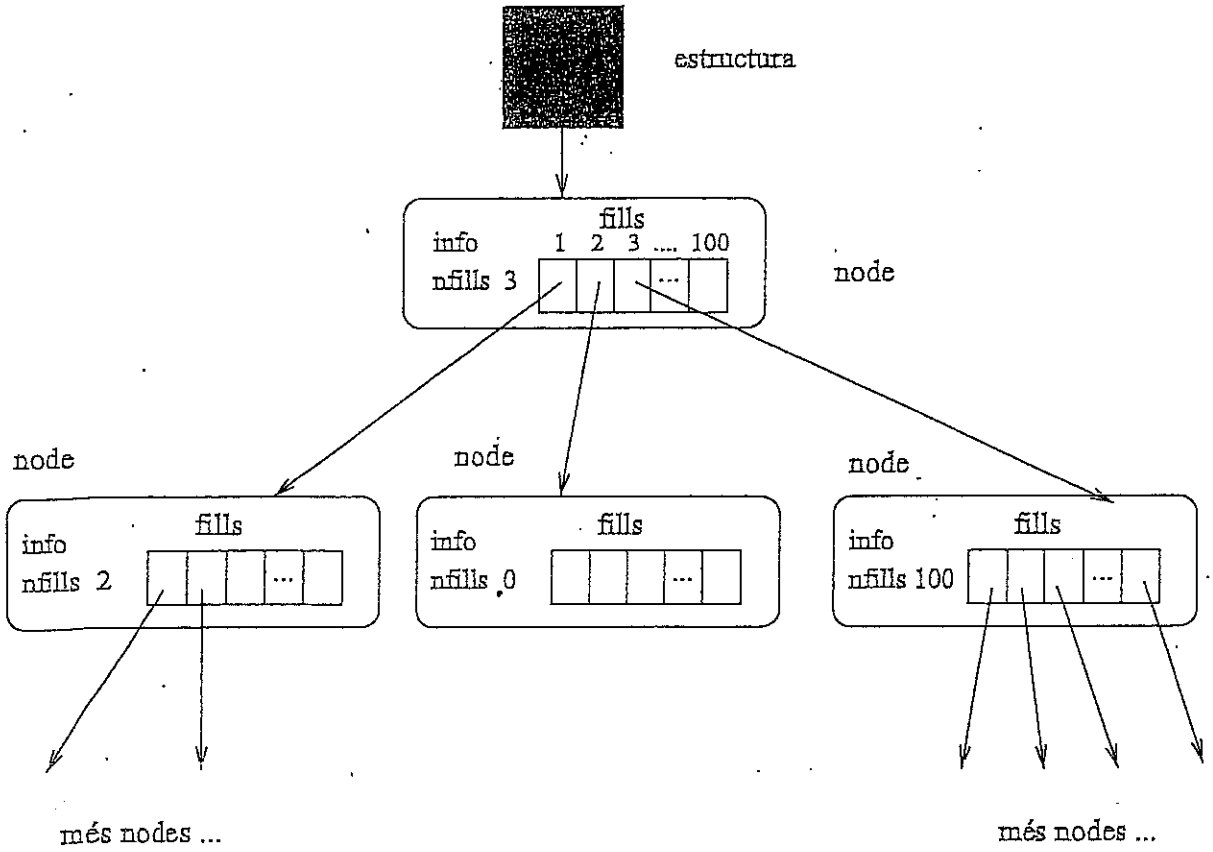
Feu servir la següent especificació i acabeu d'omplir la implementació (resoleu aquest problema en un full a part). Noteu que on posem ???? s'han de posar expressions, crides a funció o una o més instruccions.

```
funcio  maxNarb (a : Arb_maxnari de nat) ret m : nat
{ Pre: "a" no és buit }
    var i,k:nat fvar;
    m:= ????;
    i:= 1;
    {Inv: m = (valor màxim entre l'arrel i els fills[1.. i-1] d'"a")
              1 <= i <= nombre_fills(a)+1 }
    Mentre i<=nombre_fills(a) fer
      Si ???? llavors
        k:= ????;
        {HI: k= ???? ; Dec.: tamany d'"a"}
        ????
      fSi;
      i:=i+1;
    fMentre
{ Post: m= valor màxim d'"a" }
```

Al disseny heu d'incloure la justificació de tots els elements de la iteració: inicialitzacions, condició de sortida, cos del bucle (segons l'invariant i l'especificació indicada) i acabament o decreixement de la iteració.

Apartat 2.2 Implementació de l'estructura.

Considerem la següent representació gràfica per a la implementació de l'estructura descrita al principi del Problema 2 (és important que torneu a mirar aquesta descripció):



Al següent full trobareu l'esquema per resoldre l'apartat 2.2. Aquest full l'haureu de lliurar amb la vostra solució.

Cognoms i Nom: DNI:

Ompliu els espais amb fletxes (->) amb les instruccions necessàries per implementar l'estructura definida anteriorment. Tingueu en compte que, en principi, només cal posar una instrucció per cada fletxa que trobeu. Si us cal, podeu posar més instruccions, però no supereu l'espai total que hi ha disponible en cada operació.

```
Mòdul Arb_maxnari;
Implementació


Tipus Arb_maxnari=  tupla
                     ini: *node;
                    ftupla;

Tipus node= tupla
             info: elem;
             fills:  taula[1..N] de *node;
             nfills: nat;
            ftupla;


func arb_buit() ret a: Arb_maxnari;
{Pre: cert}
  a.ini := null;
{Post: "a" és un arbre buit}


funcio arb_zero_fills(x: elem) ret a: Arb_maxnari;
{Pre: cert} var n: node fvar;

  *n.info := x;
  ->
  ->

{Post: "a" és un arbre amb arrel "x" i sense fills }


accio afegir_fill(e/s a: Arb_maxnari; ent f: Arb_maxnari);
{Pre: "a" no és buit i nombre de fills d'"a"< N}

  ->
  ->

{Post: "a" ha estat modificat, afegint-li "f" com a últim fill}
```

```
accio eliminar_ifill(e/s a: Arb_maxnari; ent i: nat);
{Pre: "a" no és buit i 1<= "i" <= nombre de fills d'"a"}
  Mentre i < *(a.ini).nfills fer

    ->
    ->

  fMentre

  ->

{Post: "a" ha estat modificat, traient-li l'iessim fill}


funcio ifill (a: Arb_maxnari; i: nat) ret f: Arb_maxnari;
{Pre: "a" no és buit i 1<= "i" <= nombre de fills d'"a"}
  ->

{Post: "f" és el fill iessim d'"a"}


funcio arrel (a: Arb_maxnari) ret x: elem;
{Pre: "a" no és buit}
  ->

{Post: "x" és el valor de l'arrel d'"a"}


funcio nombre_fills (a: Arb_maxnari) ret n: nat;
{Pre: "a" no és buit}
  ->

{Post: n = (nombre de fills d'"a")}


funcio és_buit (a: Arb_maxnari) ret b: bool;
{Pre: cert}
  ->

{Post: b = ("a" és buit)}
```

Examen Pràctiques de Programació

Enginyeria Informàtica
Juny 2005
Temps estimat: 2h

Problema 1: *Minimitzar piles* (65%, no hi ha nota mínima)
Sigui p una pila de naturals. Dissenyeu una funció recursiva que retorni la pila q resultant de substituir cada element x de la pila p per la diferència entre x i el valor mínim de la subpila de p que té x com a cim (és a dir, el mínim d'x i tots els que estan per sota seu, si n'hi ha). Noteu que el mínim d'una pila buida de naturals no està definit.

Exemple:

p	q
5	4
1	0
7	4
3	0
5	1
4	0

Feu servir la següent especificació

func minimitzar (p : *pila*) dev q : *pila*
{Pre : cert}
{Post : q = pila resultant de substituir cada element de p per la diferència entre el seu valor i el valor mínim de la subpila de p que el té com a cim}

Al disseny heu d'incloure la justificació de tots els elements: condicionals ben construïts, casos trivials i recursius i acabament o decreixement.

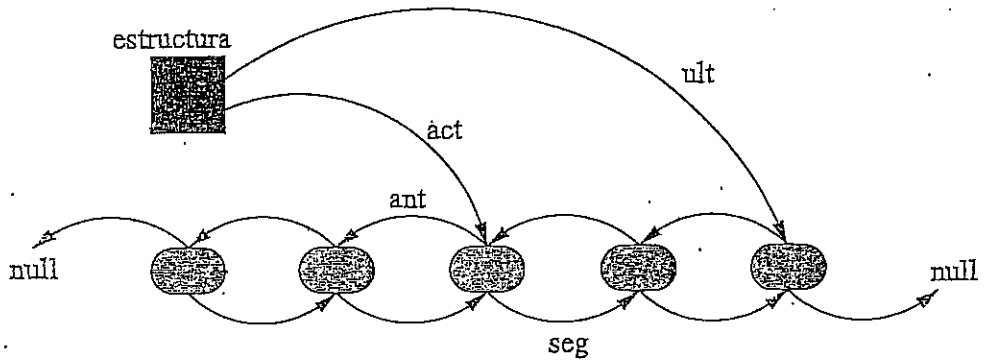
Es valorarà molt l'eficiència de la solució proposada. Per tant, heu d'evitar la repetició de càlculs amb immersions d'eficiència.

Podeu fer servir les següents operacions de piles. Tota altra operació l'heu d'especificar, dissenyar i justificar.

```
funcio pila_buida() ret p: pila
funcio empilar(x: elem; p: pila) ret q: pila
funcio desempilar(p: pila) ret q: pila
funcio cim (p: pila) ret x : elem
funcio es_buida (p: pila) ret b: bool
```

Problema 2: *Implementació amb tipus recursius* (35%, no hi ha nota mínima)
Volem implementar una estructura lineal que permeti *afegir* i *eliminar* elements per un extrem. L'*últim* element és l'últim que s'ha afegit. A més, si l'estructura no és buida, manté sempre un element, anomenat *actual*, que és l'únic consultable i que es pot moure endavant i enrera seguint l'ordre seqüencial en que els elements van ser afegits a l'estructura.

Gràficament, una possible implementació de l'estructura seria



Direm que el *següent* de l'element actual és l'element que va ser afegit just després de l'actual (sense tenir en compte els elements eliminats). Així, *avançar* l'actual és canviar-lo pel seu següent en l'estructura. Noteu que aquesta operació no és aplicable si l'estructura es buida o l'actual no té següent.

Igualment, direm que l'*anterior* de l'element actual és l'element que va ser afegit just abans de l'actual (sense tenir en compte els elements eliminats). Finalment, *retrocedir* l'actual és canviar-lo pel seu anterior en l'estructura. Noteu que aquesta operació no és aplicable si l'estructura es buida o l'actual no té anterior.

Les operacions de que disposarem seran

- crear una estructura buida
- afegir un element a l'estructura
- esborrar l'últim element afegit
- consultar l'element actual
- avançar l'element actual
- retrocedir l'element actual
- consultar si l'estructura és buida
- consultar si l'actual té un element següent
- consultar si l'actual té un element anterior

Cognoms i Nom: DNI:

Ompliu els espais amb fletxes (->) amb les instruccions necessàries per implementar l'estructura definida anteriorment i que anomenarem *str*. Tingueu en compte que, en principi, existeix una solució on només cal posar una instrucció (una assignació o un Si amb una sola assignació i sense Sino) per a cada fletxa que trobeu. Si us cal, podeu posar més instruccions que fletxes, però no supereu l'espai total que hi ha disponible en cada operació.

```
Mòdul str;
Implementació

Tipus str=  tupla                Tipus node= tupla
             act: *node;                       inf: elem;
             ult: *node;                       ant: *node;
            ftupla;                            seg: *node;
                                              ftupla;


func str_buida () ret s:str;
{Pre: cert}
 s.act := null;
 s.ult := null
{Post: s és una estructura buida}


accio afegir (e/s s:str; ent x:elem);
{Pre: cert} var n: *node fvar;
 *n.inf := x;
 *n.seg := null;
 Si (s.ult = null) llavors

 ->
 ->

 Sino

 ->
 ->

 fSi;
 s.ult := n;
{Post: s s'ha modificat posant x com a últim element;
 si l's inicial era buida llavors x és també l'element actual}
```

```
{Pre: s no és buida}


 ->
 ->
 ->


{Post: s s'ha modificat eliminant el seu últim element;
 si l'últim i l'actual de l's inicial coincidien, llavors
 el nou actual passa a ser el nou últim, altrament l'actual no canvia}

accio avançar (e/s s:str);
{Pre: s no és buida i l'actual té següent}
 ->

{Post: el nou actual d's és el següent de l'antic}

accio retrocedir (e/s s:str);
{Pre: s no és buida i l'actual té anterior}
 ->

{Post: el nou actual d's és l'anterior de l'antic}

funcio actual (s:str) ret x:elem;
{Pre: s no és buida}
 ->

{Post: x és el valor de l'element actual d's}

funcio és_buida (s:str) ret b:bool;
{Pre: cert}
 ->

{Post: b = "s és buida"}

funcio té_següent (s:str) ret b:bool;
{Pre: s no és buida}
 ->

{Post: b = "l'actual d's té un element següent"}

funcio té_anterior (s:str) ret b:bool;
{Pre: s no és buida}
 ->

{Post: b = "l'actual d's té un element anterior"}
```