

**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.

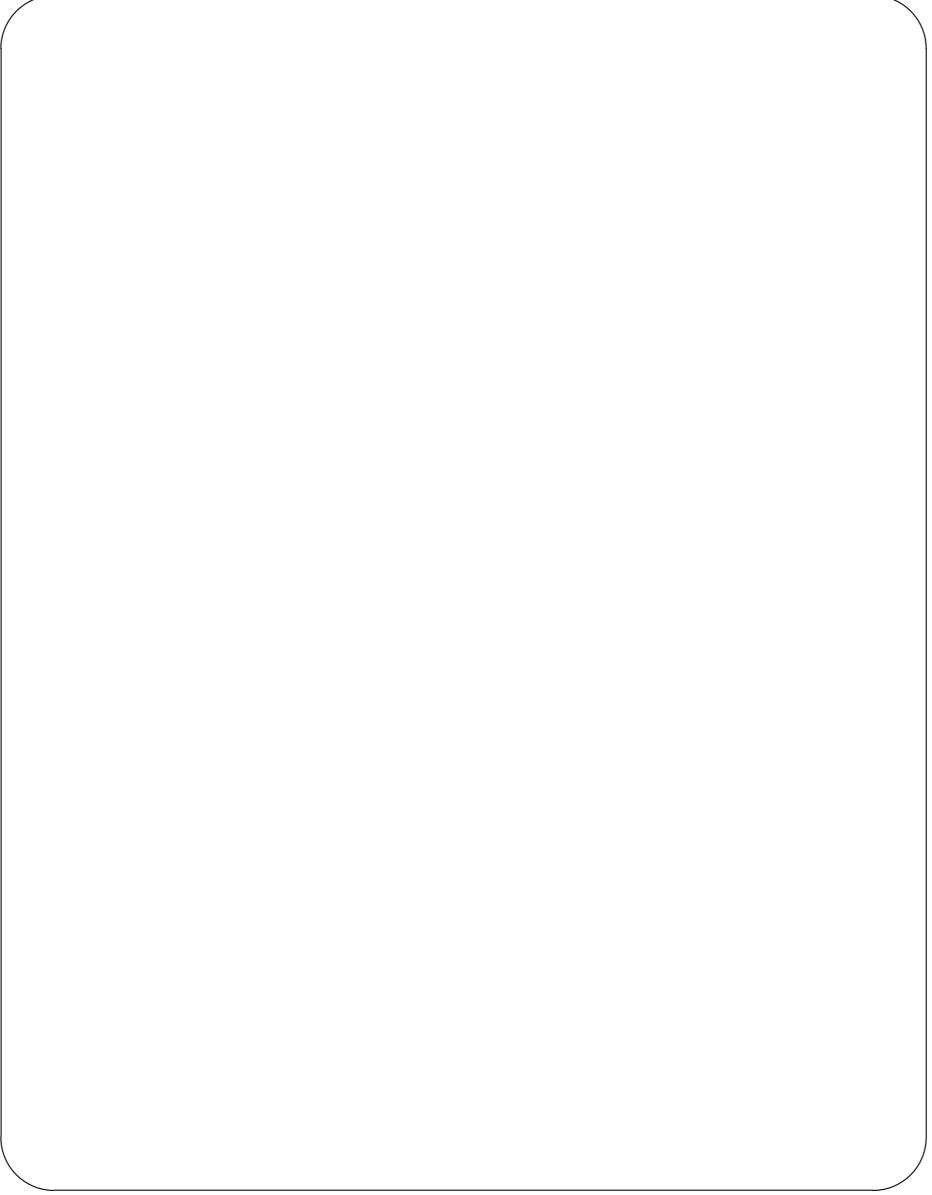
(a) (3 punts) Doneu una implementació d'un nou mètode de la classe `Llista` anomenat `transferir`. Per exemple, si el paràmetre implícit és  $(a, b, c)$  i `dest` és  $(x, y, z)$ , després de la crida el paràmetre implícit ha de ser  $(b, c)$  i `dest` ha de ser  $(x, y, z, a)$ . Cal implementar l'operació 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,  $k \geq 0$ , 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  $v.size() * (\text{longitud de la llista més llarga continguda a } v)$ . 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  $v.size() + (\text{suma de les longituds de les llistes contingudes a } v)$ . 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 dissenyar 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;
}

```

## EXERCICI 1

-----

### APARTAT (a)

```
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 */
{
    node_llista* aux = primer;

    // modificar enllaços del p.i:
    if (act == primer) act = primer->seg;
    primer = primer -> seg;
    if (primer != NULL) primer->ant = NULL;
    else ultim == NULL;

    // modificar enllaços en el node propiament
    aux->ant = dest.ultim;
    aux->seg = NULL;

    // modificar enllaços en dest
    if (dest.primer == NULL) dest.primer = aux;
    else dest.ultim->seg = aux;
    dest->ultim = aux;

    --longitud;
    ++dest.longitud;
}
```

### APARTAT (b)

Caixeta 1:

```
alguna_no_buida = false;
```

Caixeta 2:

```
if (v[i].longitud > 0) {
    v[i].transfer(*this);
    if (v[i].longitud > 0) alguna_no_buida = true;
}
```

Es igualment correcte utilitzar .primer o .ultim != NULL  
en comptes de .longitud > 0

Caixeta 3:

en blanc

### APARTAT (c)

Una manera:

- en comptes del booleà alguna\_no\_buida,  
guardar l'índex de la primera llista no buida que es troba  
per començar la següent iteració des d'allà des d'allà, i  
idem amb la última llista no buida per no seguir fins al final.  
Es fa fàcilment amb un parell de variables int addicionals.

Aquesta solució estalvia força feina si hi ha llistes curtes al principi o al final,  
però en el cas pitjor no s'estalvia gran cosa (per exemple, si la primera i la última  
llista són molt llargues i totes les altres són buides)

La manera bona (i aquest és el nivell d'explicació que s'espera, més o menys).

- guardar una llista de les posicions de v amb llistes no buides  
i recorre només aquesta llista en el for. Si un element de la llista conté el valor i,  
és que v[i] no és buida. Hem de transferir el primer element de v[i] a dest, i  
si després de fer això v[i] queda buida, cal fer l.erase

Cal fer un primer bucle extern posant tothom a dins de la llista o bé que la primera iteració ja transfereixi elements però en comptes de fer erases dels elements que queden buits faci inserts dels elements que no quedin buits. Un booleà primera\_iteracio va bé.

Això dona una solució d'eficiència proporcional a  $V.size() + (\text{suma de longituds de les llistes que hi ha a } V)$

## EXERCICI 2

-----

Solució a:

```
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 */
{
    if (primer_node == NULL) return 0;
    else return rec_iguals(primer_node);
}

int rec_iguals(node_arbreNari* m)
/* Pre: m != NULL */
/* Post: el resultat es el nombre de nodes de la jerarquia de nodes
que comença en el node apuntat per m que tenen el mateix
valor que el seu pare */
{
    int r = 0;
    int s = m->seg.size();
    for (int i = 0; i < s; ++i) {
        if (m->seg[i] != NULL) {
            r += rec_iguals(m->seg[i]);
            if (m->info == m->seg[i]->info) ++r;
        }
    }
    return r;
}
```

Solució b), pitjor (més paràmetres, Post de la funció recursiva difícil d'expressar)

```
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 */
{
    if (primer_node == NULL) return 0;
    else return rec_iguals(primer_node->info, primer_node->seg);
}

static int rec_iguals (const T& w, const vector<node_arbreNari*>& v)
/* Pre: cert */
/* Post: el resultat es la suma del nombre de nodes de les jerarquies de nodes
que comença en els nodes apuntats des de v que tenen el mateix
valor que el seu pare, mes els nodes apuntats des de v que tenen el valor w */
{
    int r = 0;
    int s = v.size();
    for (int i = 0; i < s; ++i) {
        if (v[i] != NULL) {
            r += rec_iguals(v[i]->info, v[i]->seg);
            if (w == v[i]->info) ++r;
        }
    }
    return r;
}
```

Solució c) Com la b), però passant només un punter cada vegada. Té els mateixos inconvenients i requereix codi addicional, per exemple un bucle a l'operació pública.



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.

## Problema 1 (5 punts)

Considerem la representació habitual amb nodes de la classe *Pila* per manegar piles genèriques de tipus T:

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

### Apartat 1.1 (2.5 punts)

Volem afegir una operació sobre piles d'enters denominada *subintervals* amb la següent especificació pre/post:

```
void subintervals (int dif);  
/* Pre: El p.i. conté una pila d'enters ordenada creixentment, el paràmetre  
dif és un enter positiu.  
Post: El p.i. és una pila on tota subseqüència maximal de nombres on la  
diferència entre valors consecutius és menor o igual que dif ha estat  
substituïda pel primer i l'últim de la subseqüència  
*/
```

Per exemple, amb la crida *subintervals(1)* i donada la pila :

1 2 3 5 6 9 10 11 16 17 18 20 21 27 30 34 37

on l'element 1 és el top de la pila, obtindríem:

1 3 5 6 9 11 16 18 20 21 27 30 34 37

i amb la crida *subintervals(4)* obtindríem:

1 11 16 21 27 37

Ens donen la següent implementació incompleta en la qual heu d'acabar d'omplir els forats:

```
void subintervals (int dif) {
    node_pila * api;
    node_pila * apf;
    if (primer != NULL) {
        
        while (apf != NULL) {
            if (api != apf and api->seg != apf) {
                node_pila * aptmp;

                aptmp = 

                --longitud;
            }
            if (apf->seg == NULL or
                
            ) {
                
            }
            apf = apf->seg;
        }
    }
}
```

Cognoms

Nom

DNI

--	--	--

### Apartat 1.2 (2.5 punts)

Ens donen una implementació d'una nova operació pública de la classe Pila anomenada `primer_de_cada_n`, amb la següent especificació:

```
void primer_de_cada_n(Pila &p, int n);  
/* Pre: el p.i. conté una pila, el paràmetre n és un enter  $\geq 2$  i p es la pila buida.  
   Post: el p.i. és una pila on els elements que són el primer de cada n han  
   estat eliminats i la pila p conté el primer element de cada n del p.i.  
*/
```

La implementació que ens donen és la següent:

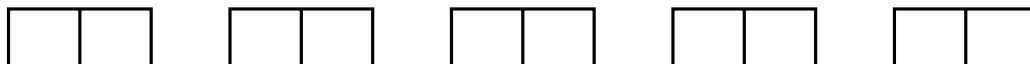
```
void primer_de_cada_n(Pila& p, int n) {  
    if (primer  $\neq$  NULL) {  
        node_pila* ap1;  
        node_pila* ap2;  
  
        p.primer = primer;  
        ap2 = p.primer;  
        primer = primer  $\rightarrow$  seg;  
        ap1 = primer;  
        int cmpt = 1;  
  
        while (ap1  $\neq$  NULL) {  
            while ((cmpt < (n - 1))) {  
                ap1 = ap1  $\rightarrow$  seg;  
                ++cmpt;  
            }  
            if (ap1  $\rightarrow$  seg == NULL) {  
                ap2  $\rightarrow$  seg = NULL;  
                ap1 = ap1  $\rightarrow$  seg;  
            } else {  
                ap2  $\rightarrow$  seg = ap1  $\rightarrow$  seg;  
                ap2 = ap2  $\rightarrow$  seg;  
                ap1  $\rightarrow$  seg = ap2  $\rightarrow$  seg;  
                ap1 = ap1  $\rightarrow$  seg;  
            }  
            cmpt = 1;  
        }  
    }  
}
```

Emplena el gràfic següent indicant els valors i encadenaments dels nodes i els atributs primer i longitud del paràmetre implícit i la pila p (pels encadenaments i primer dibuixeu la fletxa de l'encadenament o poseu NULL segons calgui) que donaria aquesta implementació amb paràmetre  $n = 2$  i la pila d'enters:

1 2 3 4 5

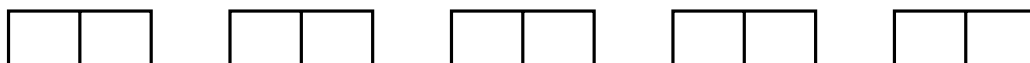
Paràmetre Implícit

Primer:	Longitud:
---------	-----------



Paràmetre p

Primer:	Longitud:
---------	-----------



Si no dóna cap resultat o produeix error d'execució, explica on i per què:

Emplena ara el gràfic següent suposant que tenim la mateixa pila però  $n = 3$ :

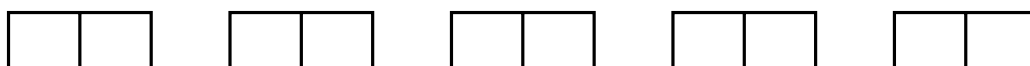
Paràmetre Implícit

Primer:	Longitud:
---------	-----------



Paràmetre p

Primer:	Longitud:
---------	-----------



Si no dóna cap resultat o produeix error d'execució, explica on i per què:

Cognoms

Nom

DNI

## Problema 2 (5 punts)

Sigui  $T$  un tipus en què hi ha definit un ordre amb les operacions de comparació habituals ( $==$ ,  $<$ ,  $<=$ , etc.). L'arbre de cerca associat a un conjunt de  $T$  es defineix així:

- L'arbre associat a un conjunt buit és l'arbre buit.
- L'arbre associat a un conjunt  $C$  no buit té a l'arrel un element  $x$  del conjunt tal que la meitat dels elements de  $C - \{x\}$  són més petits que  $x$ , i l'altra meitat són més grans que  $x$ ; si  $C$  té cardinalitat parella, això no és exactament possible, i llavors el segon conjunt té exactament un element més que el primer. El fill esquerre de l'arbre és l'arbre de cerca dels elements de  $C$  més petits que  $x$ , i el fill dret és l'arbre de cerca dels elements de  $C$  més grans que  $x$ .

Recordem la implementació dels arbres binaris:

```
template <class T> class Arbre {
private:
    struct node {
        T info;
        node* segE; // fill esquerre
        node* segD; // fill dret
    };
    node* primer;
    ... // especificació i implementació d'operacions privades
public:
    ... // especificació i implementació d'operacions públiques
};
```

Per als apartats següents tingueu en compte que:

- Cal resoldre'ls accedint a la implementació; no es poden usar operacions públiques de la classe.
- Podeu fer servir el procediment `sort()` de vectors.
- Es permet fer servir més d'un `return` en una funció si el codi queda clar.
- Es valorarà molt l'eficiència.

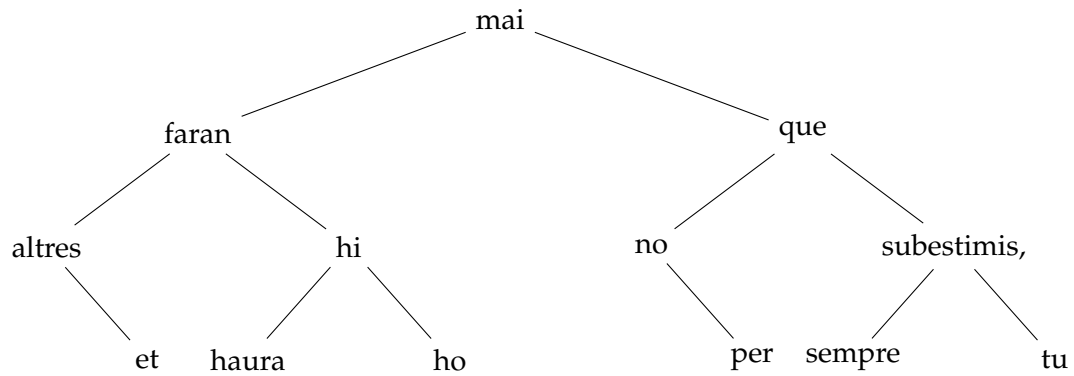
### Apartat 2.1 (2 punts)

Volem fer un mètode de la classe `Arbre<T>` que retorni l'arbre de cerca associat a un conjunt de  $T$  donat com a vector. Del vector es pot suposar únicament que no té elements repetits.

Per exemple, si  $T$  és `string` i el vector donat és

```
[ "mai" "no" "et" "subestimis," "sempre" "hi" "haura" "altres" "que" "ho" "faran" "per" "tu" ]
```

el mètode hauria de deixar en el paràmetre implícit l'arbre



Resoleu el problema omplint les capses donades de la funció `cons_arbre_cerca` i la seva funció d'immersió `i_cons_arbre_cerca`:

*/\* Pre: El paràmetre implícit és l'arbre buit; v = V i no te elements repetits \*/*  
*/\* Post: El paràmetre implícit conté l'arbre de cerca de V \*/*

**void** `cons_arbre_cerca` (*vector*<T>& v) {

}

*/\* Pre:* *\*/*

*/\* Post:* *\*/*

`i_cons_arbre_cerca`(**const** *vector*<T>& v, **int** i, **int** j) {

**if** (  ) {  
     **return** *NULL*;  
**}** **else** {

    }  
 }

**Cognoms**

**Nom**

**DNI**

### Apartat 2.2 (2 punts)

Volem fer un mètode de la classe `Arbre<T>` que digui si un element donat  $x$  és o no és a un conjunt que ens donen com a arbre de cerca. Cal aprofitar fortament que l'arbre té l'estructura descrita abans per assegurar l'eficiència.

Resoleu el problema omplint les capses donades de la funció `cerca` i la seva funció d'immersió `i_cerca`:

`/* Pre: El paràmetre implícit és un arbre de cerca d'algun conjunt C */`

`/* Post: El resultat diu si x pertany a C */`

`bool cerca(const T& x) {`

`}`

`/* Pre:`  `*/`

`/* Post:`  `*/`

`static bool i_cerca(` `, const T& x) {`

`if (`  `)`

`return` `;`

`else if (`  `)`

`return` `;`

`else if (`  `)`

`return` `;`

`else`

`return` `;`

`}`

### Apartat 2.3 (1 punt)

Expliqueu per què el nombre de comparacions entre elements de tipus T que farà cerca amb un arbre de mida  $N$  serà aproximadament  $2 \log_2 N$ , si l'element buscat no és a l'arbre.

(Nota: això és en la solució que s'espera. Si no és el cas per a la vostra solució, digueu quantes en faria i per què).



## Solució del problema 1

~~~~~

### 1.1

L'algorisme complet seria:

```
void subintervals_alt(int dif) {
    node_pila* api;
    node_pila* apf;
    if (primer != NULL) {
        api = primer;
        apf = primer; (o també apf = primer -> seg)
        while (apf != NULL) {
            if (api != apf and api->seg != apf) {
                node_pila* aptmp;
                aptmp = api->seg;
                api->seg = aptmp->seg; (o també api->seg = apf)
                delete aptmp;
                --longitud;
            }
            if (apf -> seg == NULL or
                ((apf -> seg -> info - apf->info) > dif)) {
                api = apf -> seg;
            }
            apf = apf -> seg;
        }
    }
}
```

### 1.2

L'algorisme no funciona correctament en el cas que la darrera seqüència de n elements tingui més d'un element, pero no els n. Si hi ha entre 2 i n-2 elements, fallarà en consultar el següent d'un apuntador que és NULL dins del bucle interior en acabar la seqüència de nodes abans d'arribar a n-1. Si hi ha exactament n-1 elements, fallarà en el següent if. L'algorisme tampoc actualitza la longitud de les llistes.

Per tant, la crida amb paràmetre n=2 donarà una pila 2 -> 4 -> NULL amb longitud 5 i una pila 1 -> 3 -> 5 -> NULL amb longitud 0.

La crida amb paràmetre n=3 en aquest cas donarà un error d'execució en arribar al final de la pila durant l'execució del while intern i passar a executar l'if que hi ha després d'aquest. La condició d'aquest implica consultar ap1->seg essent ap1 un apuntador NULL.

L'algorisme correcte seria:

/\* Pre: el p.i. conté una pila, el paràmetre n és un enter >= 2 i p es la pila buida.

Post: el p.i. és una pila on els elements que són el primer de cada n han estat eliminats i la pila p conté el primer element de cada n del p.i.

\*/

```
void un_de_cada_n(Pila &p, int n) {
    if (primer != NULL) {
        node_pila* ap1;
        node_pila* ap2;

        p.primer = primer;
        ap2 = p.primer;

        primer = primer -> seg;
        ap1 = primer;

        int cmpt = 1;
        --longitud;
```

```

        ++p.longitud;
    while (ap1 != NULL) {
        while ((cmpt < (n - 1)) and (ap1 != NULL)) {
            ap1 = ap1->seg;
            ++cmpt;
        }
        if (ap1 == NULL) {
            ap2 -> seg = NULL;
        } else if (ap1 -> seg == NULL) {
            ap2 -> seg = NULL;
            ap1 = ap1 -> seg;
        } else {
            ap2 ->seg = ap1 -> seg;
            ap2 = ap2 -> seg;
            ap1 -> seg = ap2 ->seg;
            ap1 = ap1 -> seg;
            --longitud;
            ++p.longitud;
        }
        cmpt = 1;
    }
}
}
}

```

## Solució del problema 2

~~~~~

### 2.1

```

// Pre: el paràmetre implícit és buit, v = V i no té elements repetits
// Post: el paràmetre implícit conté l'arbre de cerca de V
void cons_arbre_cerca(vector<T>& v) {
    v.sort();
    primer = i_cons_arbre_cerca(v,0,v.size()-1);
}

```

```

// Pre: (0 <= i) and (j < v.size())
// Post: el valor retornat apunta a una jerarquia de nodes
// que representa l'arbre de cerca corresponent a v[i..j]
static node* i_cons_arbre_cerca(const vector<T>& v, int i, int j) {
    if (i > j) return NULL;
    else {
        int k = (i+j)/2;
        node* n = new node;
        n->info = v[k];
        n->segE = i_cons_arbre_cerca(v,i,k-1);
        n->segD = i_cons_arbre_cerca(v,k+1,j);
        return n;
    }
}

```

### 2.2

```

/* Pre: El paràmetre implícit és un arbre de cerca d'algun conjunt C */
/* Post: El resultat diu si x pertany a C */
bool cerca(const T& x) {
    return i_cerca(primer,x);
}

```

```

// Pre: m apunta a l'arrel d'una jerarquia de nodes que és l'arbre de cerca d'algun
// conjunt C
// Post: el resultat diu si x pertany a C
static bool i_cerca(node* m, const T& x) {
    if (m == NULL) return false;
    else if (m->info == x) return true;
}

```

```

    else if (m->info > x) return i_cerca(m->segE,s);
    else return i_cerca(m->segD,s);
}

```

### 2.3

Si la jerarquia de nodes correspon a un arbre de cerca i té mida  $N$ , per construcció els dos fills de la jerarquia tenen mida  $\leq N/2$ , i així per cada node de la jerarquia. Això significa que l'altura de l'arbre és  $\log_2 N$ .

A cada crida, l'algorisme fa exactament una crida recursiva amb un dels fills, i si l'element no és a l'arbre s'arribarà a un dels fills buits d'una fulla. Per tant, el nombre de crides recursives és com a molt l'altura de l'arbre, que és  $\log_2 N$ .

A cada crida recursiva amb un arbre no nul es faran dues comparacions de tipus  $T$  ( $=x$  i  $<x$ ), ja que  $x$  no és a l'arbre.

Per tant el nombre de comparacions és  $2\log_2 N$ .

Errors comuns:

- Dir que l'algorisme recorre tot l'arbre. No ho fa.
- Dir o suposar que per ser arbre binari l'altura és  $\log_2 N$ .  
no, això passa amb arbres que són de cerca, però no necessàriament amb arbres binaris.
- Dir que com que l'algorisme fa "una mena de cerca dicotòmica", es fan  $\log_2 N$  comparacions, o bé que a cada comparació es descarten la meitat dels elements. De nou, l'analogia amb la cerca dicotòmica només és certa per la suposició que l'arbre és de cerca, i calia explicar-la com a dalt (les paraules "cerca dicotòmica" no fan màgia).

## 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 és buida.  
// 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 (el primer element de la cua és el de l'esquerra):

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

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

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

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

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



## 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 strings 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`.

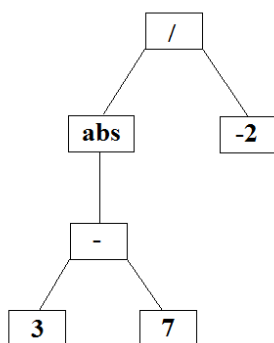
Donem tot seguit la implementació de la classe `ArbreGen`.

```
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  $(/ (\text{abs } (- 3 7)) -2)$  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, precondition, postcondition 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  $(/ (\text{abs } (- 3 7)) -2)$  mitjançant un arbre d'expressió, la crida

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

assignarà 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`.

Cognoms

Nom

DNI

**Problema 1 (6 punts)**

En aquest problema heu implementar alguns mètodes públics de la classe *Llista*, la implementació de la qual heu vist a classe de teoria. Mostrem a continuació la representació del tipus *Llista*, que utilitza nodes *doblement encadenats* amb punters a l'element següent (*seg*) i l'element anterior (*ant*). Aquesta implementació de la classe *Llista* conté els atributs següents: (1) *longitud*, de tipus enter; (2) *primer\_node*, un punter a *Node* que apunta al node que representa el primer element de la llista; (3) *ultim\_node*, un punter a *Node* que apunta al node que representa l'últim element de la llista; i (4) *act*, un punter a *Node* que apunta al node que representa l'element actual de la llista, anomenat *el punt d'interès* de la llista.

```
template <class T> class Llista {
private:
    struct Node {
        T info;
        Node* seg;
        Node* ant;
    };
    int longitud;
    Node* primer_node;
    Node* ultim_node;
    Node* act;      ... // especificació i implementació d'operacions privades
public:            ... // especificació i implementació d'operacions públiques
};
```

En les vostres respostes a aquest problema no podeu utilitzar cap mètode privat o públic de la classe *Llista* que heu vist a classe de teoria (per exemple, *copia\_node\_llista*, *esborra\_node\_llista*, *afegir*, *eliminar*, *concat*, *inici*, *fi*, *avanca*, *retrocedeix*, *actual* o *modifica\_actual*). Si utilitzeu algun mètode privat o públic auxiliar en la vostra resposta a algun apartat, heu d'especificar-lo –escrivint-ne clarament la capçalera, la precondition i la postcondition– i implementar-lo en aquesta resposta.

**1.1** Definiu el mètode públic *pop.back()*, que elimina l'últim element de la llista paràmetre implícit. Tingueu en compte que la llista paràmetre implícit inicialment no és buida, però pot quedar buida després d'aplicar aquesta operació. Per exemple, si *a* és la llista {1,5,3,4,2} i el punt d'interès apuntava a 1, després de la crida *a.pop.back()*, *a* ha de ser la llista {1,5,3,4} i el punt d'interès ha d'apuntar a 1. [2 punts]

```
void pop_back();
```

```
/* Pre: El paràmetre implícit és igual a la llista {e1, ..., en} amb n ≥ 1.*/
```

```
/* Post: El paràmetre implícit és igual a la llista {e1, ..., en-1}. Si el punt d'interès apuntava a en, després d'aplicar aquesta operació el punt d'interès apuntarà al final del parametre implícit.
```

```
Si el punt d'interès no apuntava a en abans d'aplicar aquesta operació, seguirà apuntant al mateix element al qual apuntava abans d'aplicar aquesta operació. */
```



**1.2** Definiu el mètode públic `interseccio_ordenada`, que modifica el paràmetre implícit de manera que contingui la intersecció de les llistes `c1` i `c2`. Per exemple, si `c` és una llista buida, `a` és la llista  $\{1, 3, 5, 7, 8, 9, 10\}$  i `b` és la llista  $\{-1, 1, 2, 3, 7, 8, 9\}$ , després de la crida `c.interseccio_ordenada(a,b)`, `c` ha de ser la llista  $\{1, 3, 7, 8, 9\}$  i el seu punt d'interès ha d'apuntar a 1. [4 punts]

```
void interseccio_ordenada (const Llista & c1, const Llista & c2) {  
    /* Pre: El paràmetre implícit és buit. c1 i c2 estan ordenades en ordre creixent  
    i no contenen elements repetits. */  
    /* Post: El paràmetre implícit conté els elements que pertanyen a la intersecció de  
    c1 i c2 en el mateix ordre en què estan en c1 i c2. El punt d'interès del paràmetre  
    implícit apunta al seu inici. c1 i c2 no canvien.*/
```

Cognoms

Nom

DNI

**Problema 2 (4 punts)**

Implementeu eficientment el mètode públic `treu_subarbres`, especificat a continuació.

```
void treu_subarbres (const string& x);
/* Pre: El paràmetre implícit és una arbre binari de string A. */
/* Post: Si x és el valor d'algun node de A, el paràmetre implícit és el resultat
d'eliminar de A tots els nodes amb valor x i tots els seus descendents; altrament,
el paràmetre implícit no varia (és a dir, és A). */
```

Per exemple, si  $t$  és igual a l'arbre  $a$  de la figura i  $x$  és "el", després de la crida `t.treu_subarbres(x)`,  $t$  ha de ser l'arbre  $b$  de la figura. De la mateixa manera, si  $s$  és l'arbre  $c$  de la figura i  $z$  és "lo", després de la crida `s.treu_subarbres(z)`,  $s$  no varia, és a dir,  $s$  ha de ser l'arbre  $c$  de la figura.

```
a =   juguen
      /  \
     i    contents
    /  \
   el  el
  /    \
 nen   gos
```

```
b =   juguen
      /  \
     i    contents
```

```
c =   anirem
      /  \
    aquest a
     /    \
    cap    la platja
   /  \
 de setmana
```

Donem a continuació la definició del tipus `Arbre`, que heu d'utilitzar per resoldre aquest problema.

```
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
};
```

Si utilitzeu algun mètode privat o públic de la classe `Arbre` que heu vist a classe de teoria (per exemple, `copia_node_arbre`, `esborra_node_arbre`, `a_buit`, `es_buit`, `arrel`, `plantar` o `fills`) en la vostra resposta, heu d'especificar-lo –escrivint-ne clarament la capçalera, la precondició i la postcondició– i implementar-lo en aquesta resposta.

Concretament, es demana implementar eficientment el mètode públic `treu_subarbres` fent servir diversos mètodes privats auxiliars que treballin directament amb dades de tipus `Node_arbre` i de tipus punter a `Node_arbre`. Heu de

- Escriure la capçalera, la precondició i la postcondició dels mètodes auxiliars.
- Implementar els mètodes auxiliars.
- Implementar el mètode públic `treu_subarbres` utilitzant un dels mètodes auxiliars.

Observeu que el mètode `treu_subarbres` ha d'alliberar la memòria de tots els nodes que s'eliminin del paràmetre implícit.

```
void treu_subarbres (const string& x) {  
    /* Pre: El paràmetre implícit és una arbre binari de string A. */  
    /* Post: Si x és el valor d'algun node de A, el paràmetre implícit és el resultat  
        d'eliminar de A tots els nodes amb valor x i tots els seus descendents; altrament,  
        el paràmetre implícit no varia (és a dir, és A). */
```

**Mètodes auxiliars:** especificació i implementació.

## Problema 1

**public:**

```
void pop_back() {  
    /* Pre: El paràmetre implícit és igual a la llista  $\{e_1, \dots, e_n\}$  amb  $n \geq 1$ .*/  
    /* Post: El paràmetre implícit és igual a la llista  $\{e_1, \dots, e_{n-1}\}$ . Si el punt d'interès apuntava a  $e_n$ ,  
    després d'aplicar aquesta operació el punt d'interès apuntarà al final del paràmetre implícit (i.e. end()).  
    Si el punt d'interès no apuntava a  $e_n$  abans d'aplicar aquesta operació, seguirà apuntant al mateix  
    element al qual apuntava abans d'aplicar aquesta operació. */  
    if (act == ultim_node) act = nullptr;  
    Node* aux = ultim_node;  
    ultim_node = ultim_node->ant;  
    if (ultim_node == nullptr) primer_node = nullptr;  
    else ultim_node->seg = nullptr;  
    delete aux;  
    longitud--;  
}
```

```
void interseccio_ordenada (const Llista & c1, const Llista & c2) {  
    /* Pre: El paràmetre implícit és buit. c1 i c2 estan ordenades en ordre creixent  
    i no contenen elements repetits. */  
    /* Post: El paràmetre implícit conté els elements que pertanyen a la intersecció de  
    c1 i c2 en el mateix ordre en què estan en c1 i c2. El punt d'interès del paràmetre  
    implícit apunta al seu inici. c1 i c2 no canvien.*/  
    Node* i1 = c1.primer_node;  
    Node* i2 = c2.primer_node;  
    while (i1 != nullptr and i2 != nullptr) {  
        if (i1->info == i2->info) {  
            Node* aux = new Node;  
            aux->info = i1->info;  
            if (primer_node == nullptr) {  
                primer_node = aux;  
                ultim_node = aux;  
                act = aux;  
            }  
            else {  
                ultim_node->seg = aux;  
                aux->ant = ultim_node;  
                ultim_node = aux;  
            }  
            longitud++;  
            i1 = i1->seg;  
            i2 = i2->seg;  
        }  
        else if (i1->info < i2->info) i1 = i1->seg;  
        else i2 = i2->seg;  
    }  
    if (primer_node != nullptr) {  
        primer_node->ant = nullptr;  
        ultim_node->seg = nullptr;  
    }  
}
```

## Problema 2

```
#include <string>
```

```
public:
```

```
void treu_subarbres (const string& x) {  
    /* Pre: El parámetro implícito es un árbol binario de string A. */  
    /* Post: Si x es el valor de algún nodo de A, el parámetro implícito es el  
    resultado de eliminar de A los nodos cuyo valor es x y todos sus descendientes;  
    en otro caso, el parámetro implícito no varía (es decir, es A). */  
    if (primer_node != nullptr) treu_subjerarquias (primer_node, x);  
}
```

```
private:
```

```
static void treu_subjerarquias (Node_arbre*& m, const string& x) {  
    /* Pre: m = M. M apunta a una jerarquía de nodos A no vacía. */  
    /* Post: Si x es el valor de algún nodo de A, m apunta a una jerarquía de  
    nodos que es el resultado de 'eliminar' de A los nodos cuyo valor es x y  
    todos sus descendientes; en otro caso, la jerarquía de nodos a la que apunta  
    m no varía (es decir, es A). */  
    if (m->info == x) {  
        esborra_node_arbre (m);  
        m = nullptr;  
    }  
    else {  
        if (m->segE != nullptr) treu_subjerarquias (m->segE, x);  
        if (m->segD != nullptr) treu_subjerarquias (m->segD, x);  
    }  
}
```

```
// Implementar esborra_node_arbre o una operación con otro nombre que haga  
// lo mismo que esborra_node_arbre.
```

Cognoms

Nom

DNI

**Problema 1 (6 punts)**

En aquest problema heu implementar alguns mètodes públics de la classe *Llista*, la implementació de la qual heu vist a classe de teoria. Mostrem a continuació la representació del tipus *Llista*, que utilitza nodes *doblement encadenats* amb punters a l'element següent (*seg*) i l'element anterior (*ant*). Aquesta implementació de la classe *Llista* conté els atributs següents: (1) *longitud*, de tipus enter; (2) *primer\_node*, un punter a *Node* que apunta al node que representa el primer element de la llista; (3) *ultim\_node*, un punter a *Node* que apunta al node que representa l'últim element de la llista; i (4) *act*, un punter a *Node* que apunta al node que representa l'element actual de la llista, anomenat *el punt d'interès* de la llista.

```
template <class T> class Llista {
private:
    struct Node {
        T info;
        Node* seg;
        Node* ant;
    };
    int longitud;
    Node* primer_node;
    Node* ultim_node;
    Node* act;      ...    // especificació i implementació d'operacions privades
public:
    ...                // especificació i implementació d'operacions públiques
};
```

En les vostres respostes a aquest problema no podeu utilitzar cap mètode privat o públic de la classe *Llista* que heu vist a classe de teoria (per exemple, *copia\_node\_llista*, *esborra\_node\_llista*, *afegir*, *eliminar*, *concat*, *inici*, *fi*, *avanca*, *retrocedeix*, *actual* o *modifica\_actual*). Si utilitzeu algun mètode privat o públic auxiliar en la vostra resposta a algun apartat, heu d'especificar-lo –escrivint-ne clarament la capçalera, la precondició i la postcondició– i implementar-lo en aquesta resposta.

**1.1** Definiu el mètode públic *push\_back(x)*, que afegeix l'element *x* al final de la llista paràmetre implícit. Tingueu en compte que el paràmetre implícit pot ser una llista buida o no buida. Per exemple, si *x* és 6 i *a* és la llista {1, 5, 3, 4, 2}, després de la crida *a.push\_back(x)*, *a* ha de ser la llista {1, 5, 3, 4, 2, 6}. [2 punts]

```
void push_back(const T& x) {
    /* Pre: El paràmetre implícit és igual a la llista {e1, ..., en}. */
    /* Post: El paràmetre implícit és igual a la llista {e1, ..., en, x}. */
```

**1.2** Definiu el mètode públic `interseccio_ordenada`, que modifica el paràmetre implícit de manera que en contingui la intersecció amb la llista `c2`. Observeu que aquesta operació ha d'alliberar la memòria de tots els nodes que es descartin del paràmetre implícit en calcular-ne la intersecció amb `c2`. Per exemple, si *a* és la llista  $\{1, 3, 5, 7, 8, 9, 10\}$  i *b* és la llista  $\{-1, 1, 2, 3, 7, 8, 9\}$ , després de la crida `a.interseccio_ordenada(b)`, *a* ha de ser la llista  $\{1, 3, 7, 8, 9\}$  i el seu punt d'interès ha d'apuntar a 1. [4 punts]

```
void interseccio_ordenada (const Llista & c2) {  
    /* Pre: El paràmetre implícit és igual a C1. C1 i c2 estan ordenades en ordre creixent  
    i no contenen elements repetits. */  
    /* Post: El paràmetre implícit conté els elements de C1 que pertanyen a la intersecció  
    de C1 i c2 en el mateix ordre en què estaven en C1. El punt d'interès del paràmetre  
    implícit apunta al seu inici. */
```

Cognoms

Nom

DNI

**Problema 2 (4 punts)**

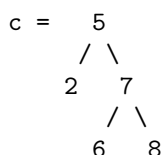
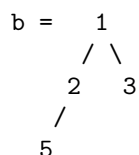
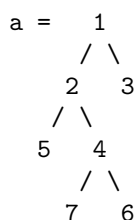
Implementeu eficientment el mètode públic `poda_subarbre`, especificat a continuació.

```
bool poda_subarbre(int x);
```

```
/* Pre: El paràmetre implícit és un arbre binari A de enters. Els valors dels nodes de A
són tots diferents. */
```

```
/* Post: Si x és el valor d'algun node de A, el resultat és cert i el paràmetre implícit és
el resultat d'eliminar de A el node amb valor x i tots els seus descendents; altrament,
el resultat és fals i el paràmetre implícit no varia (és a dir, és A). */
```

Per exemple, si  $t$  és igual a l'arbre  $a$  de la figura i  $x$  és 4, després de la crida `t.poda_subarbre(x)`,  $t$  ha de ser l'arbre  $b$  de la figura i el resultat cert. De la mateixa manera, si  $s$  és l'arbre  $c$  de la figura i  $z$  és 3, després de la crida `s.poda_subarbre(z)`,  $s$  no varia, és a dir,  $s$  ha de ser l'arbre  $c$  de la figura i el resultat fals.



Donem a continuació la definició del tipus `Arbre`, que heu d'utilitzar per resoldre aquest problema.

```
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
};
```

Si utilitzeu algun mètode privat o públic de la classe `Arbre` que heu vist a classe de teoria (per exemple, `copia_node_arbre`, `esborra_node_arbre`, `a_buit`, `es_buit`, `arrel`, `plantar` o `fills`) en la vostra resposta, heu d'especificar-lo –escrivint-ne clarament la capçalera, la precondition i la postcondició– i implementar-lo en aquesta resposta.

Concretament, es demana implementar eficientment el mètode públic `poda_subarbre` fent servir diversos mètodes privats auxiliars que treballin directament amb dades de tipus `Node_arbre` i de tipus punter a `Node_arbre`. Heu de

- Escriure la capçalera, la precondition i la postcondició dels mètodes auxiliars.
- Implementar els mètodes auxiliars.
- Implementar el mètode públic `poda_subarbre` utilitzant un dels mètodes auxiliars.

Observeu que el mètode `poda_subarbre` ha d'alliberar la memòria de tots els nodes que s'eliminin del paràmetre implícit.



```
bool poda_subarbre(int x) {
```

```
/* Pre: El paràmetre implícit és un arbre binari A de enters. Els valors dels nodes de A són  
tots diferents. */
```

```
/* Post: Si x és el valor d'algun node de A, el resultat és cert i el paràmetre implícit és  
el resultat d'eliminar de A el node amb valor x i tots els seus descendents; altrament,  
el resultat és fals i el paràmetre implícit no varia (és a dir, és A). */
```

**Mètodes auxiliars:** especificació i implementació.

## Problema 1

**public:**

```
void push_back(const T& x) {  
/* Pre: El paràmetre implícit és igual a la llista  $\{e_1, \dots, e_n\}$ . */  
/* Post: El paràmetre implícit és igual a la llista  $\{e_1, \dots, e_n, x\}$ . */  
    Node* aux = new Node;  
    aux→info = x;  
    aux→seg = nullptr;  
    aux→ant = ultim_node;  
    if (primer_node == nullptr) primer_node = aux;  
    else ultim_node→seg = aux;  
    ultim_node = aux;  
    longitud++;  
}
```

```
void interseccio_ordenada (const Llista & c2) {  
/* Pre: El paràmetre implícit és igual a C1. C1 i c2 estan ordenades en ordre creixent  
i no contenen elements repetits. */  
/* Post: El paràmetre implícit conté els elements de C1 que pertanyen a la intersecció  
de C1 i c2 en el mateix ordre en què estaven en C1. El punt d'interès del paràmetre  
implícit apunta al seu inici . */  
    Node* i1 = primer_node;  
    Node* i2 = c2.primer_node;  
    primer_node = nullptr;  
    ultim_node = nullptr;  
    act = nullptr;  
    longitud = 0;  
    while (i1 ≠ nullptr and i2 ≠ nullptr) {  
        if (i1→info == i2→info) {  
            if (primer_node == nullptr) {  
                primer_node = i1;  
                ultim_node = i1;  
                act = i1;  
            }  
            else {  
                ultim_node→seg = i1;  
                i1→ant = ultim_node;  
                ultim_node = i1;  
            }  
            i1 = i1→seg;  
            i2 = i2→seg;  
            longitud++;  
        }  
        else if (i1→info < i2→info) {  
            Node* aux = i1;  
            i1 = i1→seg;  
            // Alliberem la memòria dels nodes de C1 que no pertanyen a la intersecció de C1 i c2.  
            delete aux;  
        }  
        else i2 = i2→seg;  
    }  
}
```

```

if (primer_node  $\neq$  nullptr) {
    primer_node→ant = nullptr;
    ultim_node→seg = nullptr;
}
// Alliberem la memòria dels nodes de C1 que no pertanyen a la intersecció de C1 i c2.
while (i1  $\neq$  nullptr) {
    Node* aux = i1;
    i1 = i1→seg;
    delete aux;
}
}

```

## Problema 2

**public:**

```
bool poda_subarbre(int x) {  
    /* Pre: El parámetro implícito es un árbol binario de enteros A. Los valores de  
    los nodos de A son todos distintos. */  
    /* Post: Si x es el valor de algún nodo de A, el resultado es cierto y el parámetro  
    implícito es el resultado de eliminar de A el nodo cuyo valor es x y todos sus  
    descendientes; en otro caso, el resultado es falso y el parámetro implícito no varía  
    (es decir, es A). */  
    return poda_subjerarquia (primer_node, x);  
}
```

**private:**

```
static bool poda_subjerarquia (Node_arbre*& m, int x) {  
    /* Pre: m = M. M apunta a NULL o a una jerarquía de nodos A. Los valores  
    de los nodos de A son todos distintos. */  
    /* Post: Si x es el valor de algún nodo de A, el resultado es cierto, y m  
    apunta a una jerarquía de nodos que es el resultado de 'eliminar' de A  
    el nodo cuyo valor es x y todos sus descendientes; en otro caso, el  
    resultado es falso y la jerarquía de nodos a la que apunta m no varía  
    (es decir, es A). */  
    bool encontrado;  
    if (m ≠ nullptr) {  
        if (m→info == x) {  
            encontrado = true;  
            esborra_node_arbre (m);  
            m = nullptr;  
        }  
        else {  
            encontrado = poda_subjerarquia (m→segE, x);  
            if (not encontrado) encontrado = poda_subjerarquia (m→segD, x);  
        }  
    }  
    else encontrado = false;  
    return encontrado;  
}  
  
// Implementar esborra_node_arbre o una operación con otro nombre que haga  
// lo mismo que esborra_node_arbre.
```

Cognoms

Nom

DNI

**Problema 1 (5 punts)**

Volem construir una estructura de dades lineal i simplement encadenada, anomenada *Memoria*, que doni suport a un sistema operatiu en la gestió de la memòria dinàmica.

En el nostre sistema, els programes fan peticions de memòria al sistema amb un paràmetre  $b$  que és la quantitat de bytes que necessiten. El sistema busca una adreça de memòria a partir de la qual hi hagi  $b$  bytes consecutius disponibles, els reserva i retorna al programa demandant un identificador per al bloc de memòria reservat. Els identificadors es donen de forma successiva, 0, 1, 2, ..., a mesura que hi ha peticions.

La informació es guarda en una seqüència de nodes cadascun dels quals conté la informació d'un bloc de memòria concedit. Un node conté l'identificador que se li ha donat al bloc de memòria que representa aquest node, l'adreça de memòria on comença el bloc, el nombre de bytes que el formen i un apuntador al següent node. Important és que **en la nostra implementació els nodes es mantenen ordenats de manera creixent per adreça de memòria** (no per identificador).

Un objecte de la classe *Memoria* conté: 1) un atribut  $N$  que és la mida de la memòria que gestiona i que s'assigna en crear l'objecte; 2) el darrer identificador assignat fins ara, i 3) un apuntador al primer bloc. Les adreces de memòria van des de 0 a  $N - 1$ . La representació del tipus *Memoria* en C++ és aquesta:

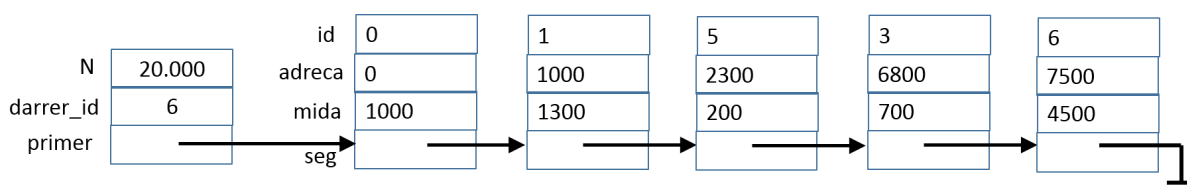
```
class Memoria {
private:
    struct Node {
        int id;
        int adreca;
        int mida;
        Node* seg;
    };

    Node* primer; // apuntador a primer node
    int N;         // la màxima adreça de memòria possible es N-1
    int darrer_id; // darrer identificador lliurat
    ... mètodes privats ...

public:
    ... mètodes públics ...
};
```

Hi ha un bloc de memòria especial, que és el bloc que el sistema operatiu es guarda per a ell mateix al principi de l'execució. Comença a l'adreça 0 (i, per tant, sempre està representat pel primer node), té identificador 0 i no s'esborra mai.

Per exemple, la figura següent mostra el contingut (o estat) d'un objecte  $t$  de tipus *Memoria* en un moment determinat. En aquest moment hi ha 5 blocs reservats.



El primer bloc comença a l'adreça 0, ocupa 1000 bytes i té identificador 0. El segon bloc comença a l'adreça 1000, ocupa 1300 bytes i té identificador 1. El tercer bloc comença a l'adreça 2300, ocupa 200 bytes i té identificador 5. Hi ha un forat de 4300 bytes lliures entre aquest bloc i el següent, que té identificador 3, comença a l'adreça 6800 i ocupa 700 bytes. A continuació existeix un darrer bloc, que té identificador 6, comença a l'adreça 7500 i ocupa 4500 bytes. Després hi ha un forat amb memòria lliure entre la posició 12000 i la darrera posició, que té l'adreça 19999.

L'identificador del darrer bloc concedit és el 6. Això vol dir que hi ha dos identificadors, el 2 i el 4, que s'han d'haver esborrat de *t* (és a dir, alliberats) abans del moment que mostra la figura.

Concretament, en aquest exercici, es demana implementar eficientment els mètodes públics **demanar** i **alliberar** de la classe *Memoria*, especificats a continuació:

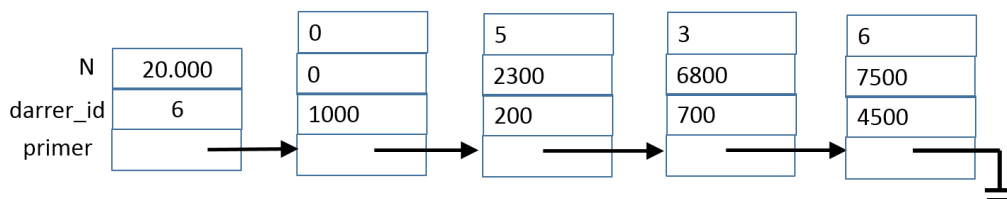
```
void alliberar (int x);
```

```
/* Pre:  $x > 0$ . */
```

```
/* Post: Si el paràmetre implícit (p.i.) contenia un bloc de memòria amb identificador  $x$ , la memòria que ocupava el bloc amb identificador  $x$  ha estat alliberada y l'identificador  $x$ 
```

```
ha deixat de ser vàlid, perquè el p.i. ja no conté un bloc de memòria amb identificador  $x$ . */
```

Per exemple, si *t* és un objecte de tipus *Memoria* amb el contingut mostrat en la figura anterior, després de la crida *t.alliberar*(1) el contingut de *t* hauria de ser el següent:



```
int demanar(int b);
```

```
/* Pre:  $b > 0$ 
```

```
/* Post: Si no hi ha  $b$  bytes consecutius lliures al p.i., el resultat és  $-1$ ; en un altre cas, es reserva un bloc format pels primers  $b$  bytes consecutius lliures (és a dir, no reservats) del p.i., se li assigna com a identificador el enter següent a l'últim identificador assignat i es retorna aquest identificador com a resultat. */
```

Per exemple, si *t* és un objecte de tipus *Memoria* amb el contingut mostrat en la figura anterior, després de la seqüència de crides següent

```
t.demanar(1400);
```

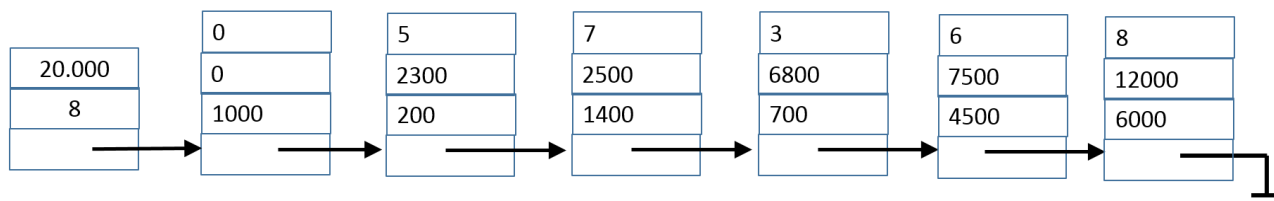
```
t.demanar(6000);
```

el resultat hauria de ser

```
7
```

```
8
```

i el contingut del objecte de tipus *Memoria* *t* el següent:



### Restriccions del problema:

- Cal respondre dins de l'espai donat, respectant l'estructura que es proposa, **però no és imprescindible omplir totes les caixes.**
- No es poden fer servir estructures de dades auxiliars (com ara piles, cues o llistes). Heu de treballar directament amb dades de tipus *Memoria*, *Node* i punter a *Node*.
- Els mètodes públics **demanar** i **alliberar** haurien d'examinar una vegada com a màxim cada node del paràmetre implícit.

**Cognoms**

**Nom**

**DNI**

**1.1 Implementació de alliberar. [2 punts]**

```
void alliberar (int x) {  
    Node* p = primer;
```

```
    while (  ) p = p→seg;
```

```
}
```

**1.2 Implementació de demanar. [3 punts]**

```
int demanar(int b) {  
    Node* p = primer;
```

```
    while (  ) {
```

```
        p = pseg;
```

```
        pseg = pseg→seg;
```

```
    }
```

```
    if (  ) return -1;
```

```
    Node* aux = new node;
```

```
    return darrer_id ; }
```

### Problema 2 (5 punts)

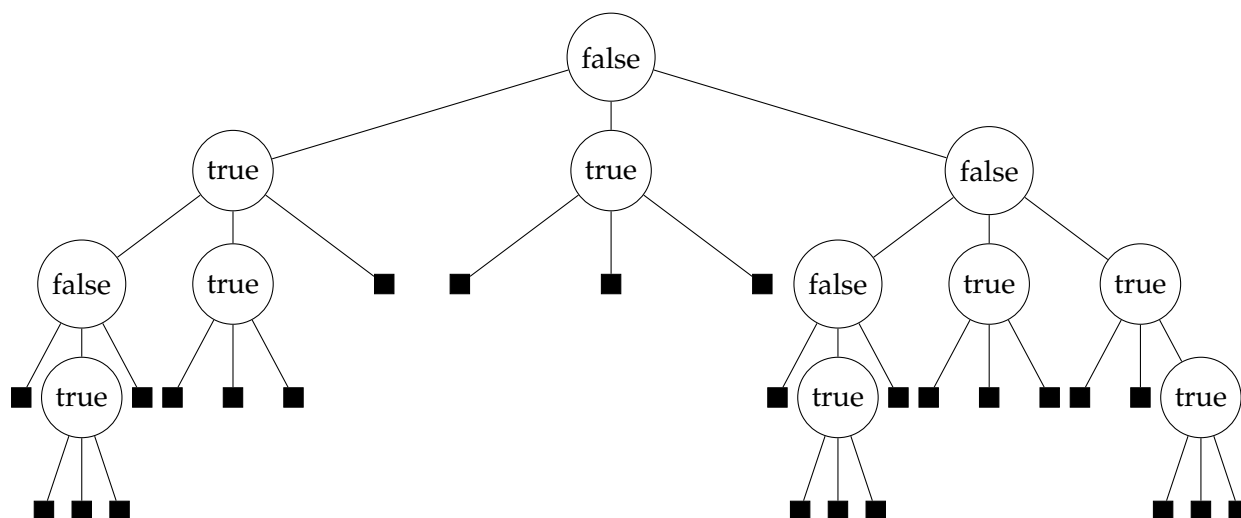
En aquest exercici utilitzarem la classe *ArbreNari*, vista a la classe de teoria, per representar conjunts de paraules formades únicament per lletres minúscules de l'alfabet anglès. Concretament, per representar un conjunt de paraules  $\mathbf{s} = \{p_1, \dots, p_n\}$  utilitzarem un *arbre N-ari de booleans*  $\mathbf{t}$  amb aritat  $N = 26$ . Aquest arbre 26-ari de booleans es defineix de la manera següent:

1. El conjunt de paraules buit (és a dir,  $\{\}$ ) es representa mitjançant l'arbre buit.
2. Si  $\mathbf{s}$  és un conjunt de paraules no buit i  $\mathbf{s}$  conté la paraula buida (és a dir, si hi ha un  $k \in \{1, \dots, n\}$  tal que  $p_k = ""$ ), el camp *info* del node arrel de  $\mathbf{t}$  és igual a *true*.
3. Si  $\mathbf{s}$  és un conjunt de paraules no buit i  $\mathbf{s}$  no conté la paraula buida (és a dir, si no hi ha un  $k \in \{1, \dots, n\}$  tal que  $p_k = ""$ ), el camp *info* del node arrel de  $\mathbf{t}$  és igual a *false*.
4. Cada node de l'arbre  $\mathbf{t}$  amb **info** igual a **true** representa una paraula del conjunt  $\mathbf{s}$ . En particular, el nombre d'elements del conjunt  $\mathbf{s}$  és igual al nombre de nodes de l'arbre  $\mathbf{t}$  amb **info** igual a **true**.
5. Per a tot  $i = 0, \dots, 25$ , el fill  $i$ -èsim de  $\mathbf{t}$  conté el subconjunt de paraules de  $\mathbf{s}$  que comencen per la lletra  $i$ -èsima de l'alfabet anglès (és a dir, pel caràcter resultant d'avaluar l'expressió `char('a' + i)`).

Pel que fa al punt 5 de la definició anterior, cal aclarir que:

- En aquest exercici enumerem els fills d'un arbre 26-ari de booleans  $\mathbf{t}$  de 0 a 25, en lloc d'1 a 26, per similitud amb la notació de vectors que s'utilitza per representar els punters que apunten a aquests fills en la representació del tipus *Node* de la classe *ArbreNari*. De la mateixa manera, enumerem les lletres de l'alfabet anglès de 0 a 25, de manera que la lletra 0-èsima correspon al caràcter 'a', la lletra 1-èsima correspon al caràcter 'b', ... i la lletra 25-èsima al caràcter 'z'.
- Si el conjunt  $\mathbf{s}$  no conté cap paraula que comenci per la lletra  $i$ -èsima de l'alfabet anglès, el fill  $i$ -èsim de  $\mathbf{t}$  és buit.
- Si denotem amb  $\mathbf{s}_i$  el subconjunt de paraules de  $\mathbf{s}$  que comencen per la lletra  $i$ -èsima de l'alfabet anglès i  $\mathbf{s}_i$  és no buit, llavors el fill  $i$ -èsim de  $\mathbf{t}$  conté el conjunt de paraules resultant d'eliminar el primer caràcter de totes les paraules de  $\mathbf{s}_i$ .

En l'exemple següent, considerem únicament conjunts de paraules formades per les tres primeres lletres de l'alfabet (és a dir,  $\{a, b, c\}$ ) i fem servir, per tant, un arbre ternari de booleans per representar aquests conjunts en lloc d'un arbre 26-ari de booleans. L'arbre ternari de booleans següent representa el conjunt de paraules  $x = \{a, aab, ab, b, cab, cb, cc, ccc\}$ .

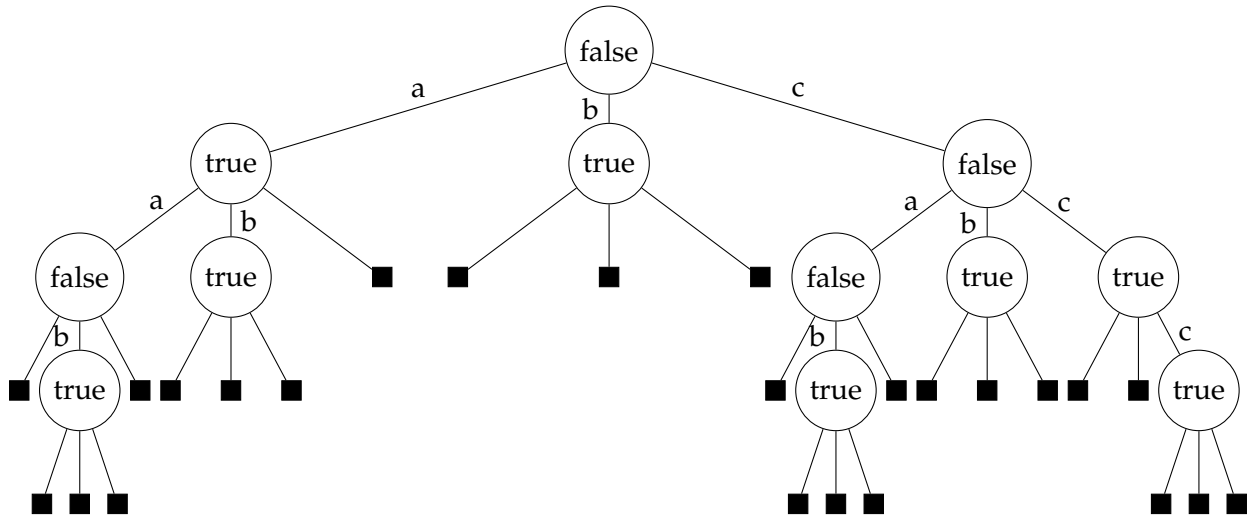


Observeu que dibuixem els nodes de l'arbre mitjançant cercles que contenen el valor del seu camp **info**. Els enllaços de cada node es representen mitjançant eixos que apunten a altres nodes o al valor **NULL** (concretament els enllaços **seg[0]**, **seg[1]**, **seg[2]** es representen d'esquerra a dreta, respectivament). El



valor **NULL** es representa mitjançant un quadrat negre. En el context d'aquest enunciat, la constant **NULL** equival a la constant **nullptr**.

**Indicació:** Concretament, si a l'arbre anterior etiquetem els enllaços **seg[0]**, **seg[1]**, **seg[2]** amb les lletres de l'alfabet associades a cadascun (és a dir, amb **a**, **b**, **c**, respectivament), les paraules del conjunt representat per aquest arbre  $x = \{a, aab, ab, b, cab, cb, cc, ccc\}$  s'obtenen concatenant els caràcters dels enllaços que condueixen des de l'arrel fins a cadascun dels nodes amb info igual a **true**. Observeu que només etiquetem els enllaços que condueixen a nodes, no els que apunten a **NULL**.



Concretament, es demana implementar els mètodes públics **pertany** i **totes** i el mètode privat **comencen\_prefix**.

```
bool pertany(const string& p) const;
```

```
/* Pre: p és una cadena de caràcters formada per lletres minúscules de l'alfabet anglès. */
```

```
/* Post: El resultat indica si p pertany al conjunt de paraules representat pel p.i. */
```

```
void totes ( list <string>& v) const;
```

```
/* Pre: v = V. V és una llista buida. */
```

```
/* Post: v conté les paraules del conjunt representat pel p.i. ordenades en ordre alfabètic. */
```

En la implementació del mètode **totes** heu d'utilitzar el mètode privat **comencen\_prefix** especificat a continuació.

```
static void comencen_prefix(Node* m, const string& prefix, list <string>& v);
```

```
/* Pre: V = (p1, ..., pk). V està ordenada en ordre alfabètic. Les paraules de V precedeixen a la paraula 'prefix' en l'ordre lexicogràfic (és a dir, alfabètic). */
```

```
/* Post: v = (p1, ..., pk, q1, ..., qh), on (q1, ..., qh) és la llista de paraules de la forma (prefix + e1, ..., prefix + eh) i (e1, ..., eh) és la llista de paraules del conjunt que representa la jerarquia de nodes apuntada per m ordenades en ordre alfabètic. */
```

**Observació:** Recordeu que l'operador **+** permet concatenar dues strings, o un string i un caràcter. Per exemple, el resultat d'avaluar l'expressió **"abc" + "wyz"** és **"abcwyz"**. De la mateixa manera, el resultat d'avaluar l'expressió **"awyz" + 'z'** és **"awyz"**. En la vostra solució podeu utilitzar també els mètodes públics de les classes **string** i **list** de la STL de C++.

Per exemple, si inicialment la variable **d** és igual a l'string **c**, la variable **w** és igual a la llista de paraules (**a**, **aab**, **ab**, **b**) i la variable **p** apunta al node arrel del fill 2-èsim (és a dir, el tercer fill) de l'arbre de la figura, després de la crida **comencen\_prefix(p, d, w)** la variable **w** ha de ser igual a la llista de paraules (**a**, **aab**, **ab**, **b**, **cab**, **cb**, **cc**, **ccc**).

També podeu utilitzar altres mètodes privats auxiliars que treballen directament amb dades de tipus **Node** i de tipus punter a **Node**. En aquest cas, heu de: 1) escriure la capçalera, la precondition i la postcondition de cada mètode auxiliar; 2) implementar-lo, i 3) implementar els mètodes **pertany**, **totes** i **comencen\_prefix** fent servir aquests mètodes auxiliars. Donem a continuació la definició del tipus **ArbreNari**, que heu de fer servir per resoldre aquest problema.

**Cognoms**

**Nom**

**DNI**

```
template <class T> class ArbreNari {
```

```
private:
```

```
    struct Node {
```

```
        T info;
```

```
        vector<Node*> seg;
```

```
    };
```

```
    int N;
```

```
    Node* primer;
```

```
    ...
```

```
        // especificació i implementació de mètodes privats
```

```
public:
```

```
    ...
```

```
        // especificació i implementació de mètodes públics ;
```

**2.1 Implementació de **pertany**.** Donada una paraula  $p$  de longitud  $m$ , el mètode **pertany** hauria d'examinar com a màxim  $m$  nodes del p.i (l'arbre que representa el conjunto de paraules). [2.5 punts].

```
bool pertany(const string &p) const;
```

```
/* Pre: p és una cadena de caràcters formada per lletres minúscules de l'alfabet anglès. */
```

```
/* Post: El resultat indica si p pertany al conjunt de paraules representat pel p.i. */
```

**Cognoms**

**Nom**

**DNI**

**2.2 Implementació de `totes` i de `comencen_prefix`.** El mètode **`comencen_prefix`** hauria d'examinar una i solament una vegada cada node del p.i. (l'arbre que representa el conjunt de paraules). **[2.5 punts]**.

```
void totes ( list <string>& v) const;
```

```
/* Pre: v = V. V és una llista buida. */
```

```
/* Post: v conté les paraules del conjunt de paraules representat pel p. i. ordenades en ordre alfabètic. */
```

[illegible]

--	--	--	--	--	--	--	--

**Titulació:** Grau en Enginyeria Informàtica

**Assignatura:** Programació 2 (PRO2)

**Duració:**

**Curs: Q1 2018–2019 (2n Parcial)**

**Data:** 14 de Gener de 2019

1. (50%) Considerem la següent classe `Llista`:

```
template <class T>
class Llista {
public:
    ...
    void splice(const T& x, Llista& l2);
    ...
private:
    struct node_llista {
        T info;
        node_llista* ant;
        node_llista* seg;
    };
    int longitud;
    node_llista* primer_node;
    node_llista* ultim_node;
    node_llista* act;

    void nullify();
    static node_llista* search(node_llista* p, const T& x);
};
```

que representa una llista d'Ts mitjançant una cadena doblement enllaçada de nodes. Els apuntadors `primer_node` i `ultim_node` apunten al primer i últim nodes de la llista, respectivament. Si la llista és buida llavors `primer_node == ultim_node == act == nullptr`. Per a una llista no buida, el predecessor del primer element és nul (`primer_node -> ant == nullptr`) i el successor de l'últim element és nul (`ultim_node -> seg == nullptr`). L'apuntador `act` apunta al punt d'interès. L'atribut `longitud` és el nombre d'elements de la llista.

- (a) (10%) Implementa el mètode privat

```
// Pre: si no està buida, la cadena de nodes del
// p.i. està compartida amb una altra llista
// Post: el p.i. representa a una llista buida
void nullify();
```

Com que la cadena de nodes de la llista implícita està compartida (si no és buida) i no volem efectes secundaris sobre altres llistes, aquesta operació **no** ha de destruir la cadena de nodes.

(b) (15%) Implementa el mètode privat

```
// Pre: cert
// Post: retorna un apuntador al primer node que conté x
// en la cadena de nodes que comença a p, o nullptr
// si no hi ha cap node a partir de p que contingui x
static node_llista* search(node_llista* p, const T& x);
```

(c) (25%) Implementa el mètode públic

```
// Pre: cert
// Post: insereix la llista l2 just davant de la primera
// aparició de l'element x en el p.i. si x està present,
// o al final del p.i. si x no està present; en qualsevol cas
// la llista l2 queda buida, i el punt d'interès del p.i.
// queda inalterat
void splice(const T& x, Llista& l2);
```

Per exemple, si tenim les llistes  $l1 = [3, 7, -2, 5, 0, 7]$  i  $l2 = [3, 4, 3, 4]$  llavors després de la crida `l1.splice(7, l2)` tindrem  $l1 = [3, 3, 4, 3, 4, 7, -2, 5, 0, 7]$  i  $l2 = []$ . Si la crida hagués estat `l1.splice(8, l2)` llavors, un cop acabada la crida, tindríem  $l1 = [3, 7, -2, 5, 0, 7, 3, 4, 3, 4]$  i  $l2 = []$ . Si  $l1 = []$  i  $l2 = L2$  aleshores després de la crida `l1.splice(x, l2)` tindrem  $l1 = L2$  i  $l2 = []$ , siguin quins siguin el valor de  $x$  i els continguts de  $l2$ .

Es valorarà l'eficiència de la solució proposada. La teva solució no pot utilitzar memòria extra (a part de variables auxiliars senzilles com ara booleans o apuntadors) i no ha de crear ni destruir nodes. Pots suposar que els operadors Booleans `==` i `!=` entre objectes de la classe `T` estan definits.

Vigila que la teva solució tracti adequadament, a més dels que ja apareixen en els exemples, la resta dels casos extrems, com ara que la llista  $l2$  sigui una llista buida, o que la llista  $l2$  s'hagi de transferir just davant del primer element de la llista implícita.

---

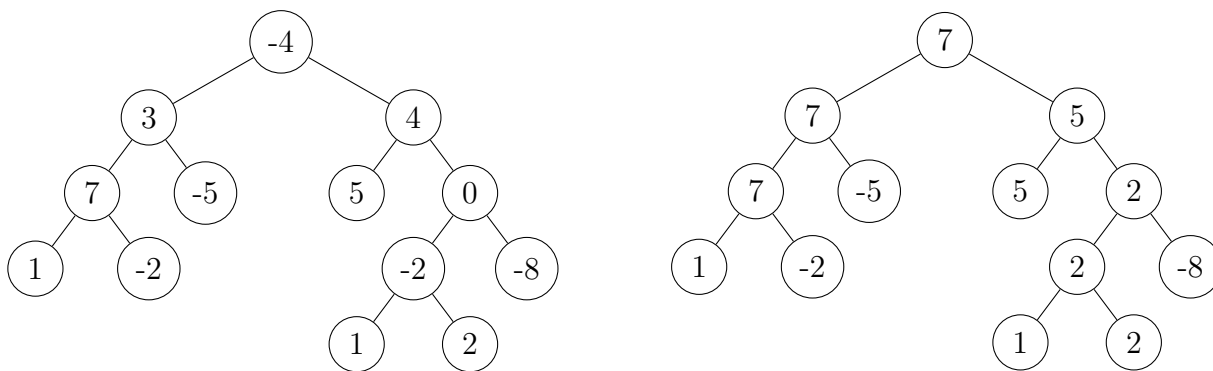
**SOLUCIÓ:**

2. (50%) Considerem la següent classe **Arbre**:

```
template <class T>
class Arbre {
public:
    ...
    // Pre: el p.i. no és buit i tot node té exactament zero
    //       o 2 fills no buits.
    // Post: retorna l'arbre de màxims corresponent al p.i.
    //       (s'explica a continuació)
    Arbre arbre_maxims();
    ...
private:
    struct node_arbre {
        T info;
        node_arbre* segE;
        node_arbre* segD;
    };
    node_arbre* primer_node;
    // altres operacions que pots afegir com a auxiliars
    // d'arbre_maxims
};
```

que representa a un arbre binari d'Ts, mitjançant un únic apuntador (`primer_node`) a l'arrel de l'arbre.

Implementa el mètode `arbre_maxims` que torna un `Arbre` amb idèntica estructura que el paràmetre implícit i ón cada node conté el màxim dels nodes del subarbre corresponent en l'arbre original. Per exemple si `a` és l'arbre a la part esquerra de la figura, aleshores `a.arbre_maxims()` retorna l'arbre de la part dreta.



Es valorarà l'eficiència de la teva solució. El mètode `arbre_maxims` ha de tenir cost lineal respecte al nombre de nodes de l'arbre implícit. Pots assumir que tots els operadors Booleans de comparació (`<`, `<=`, `>`, `>=`, ...) entre objectes de la classe `T` estan definits.

---

---

**SOLUCIÓ:**

## Problema 1.

```
// Apartat 1
void nullify() {
    primer_node = ultim_node = act = nullptr;
    longitud = 0;
}

// Apartat 2
static node_llista* search(node_llista* p, const T& x) {
    while (p != nullptr and p->info != x)
        p = p->seg;
    return p;
}

// L'apartat 3 hi és a la pàgina següent.
```

```

// Apartat 3
void splice(const T& x, Llista& l2) {
    if (l2.primer_node != nullptr) { // si l2 es buida no cal fer res

        if (primer_node == nullptr) { // la llista implícita es buida
            primer_node = l2.primer_node;
            ultim_node = l2.ultim_node;
            longitud = l2.longitud;
            l2.nullify();
        }
        else { // la llista implícita no es buida i l2 tampoc

            node_llista* p = search(primer_node, x);
            // p == nullptr i x no està a la llista o
            // p apunta a la primera aparició d'x en la llista implícita

            if (p == nullptr) {
                // l2 s'ha de transferir al final de la llista implicita
                // i per tant canvia l'ultim
                ultim_node->seg = l2.primer_node;
                l2.primer_node->ant = ultim_node;
                ultim_node = l2.ultim_node;
            }
            else if (p != primer_node) {
                // p != nullptr i p != primer_node
                // l2 s'ha de transferir a un punt intermig de la llista implicita
                // no canvia ni el primer ni l'últim
                node_llista* q = p->ant;
                q->seg = l2.primer_node;
                l2.primer_node->ant = q;
                p->ant = l2.ultim_node;
                l2.ultim_node->seg = p;
            }
            else {
                // p == primer_node != nullptr
                // l2 s'ha de transferir al principi de la llista implicita
                // i per tant canvia el primer
                primer_node->ant = l2.ultim_node;
                l2.ultim_node->seg = primer_node;
                primer_node = l2.primer_node;
            }
            longitud += l2.longitud;
            l2.nullify();
        }
    }
}

```



## Problema 2.

Definim `node_arb_max`, mètode static privat de la classe `Arbre`, com a immersió d'`arbre_maxims`. Adaptant bé l'especificació d'`arbre_maxims` podrem simplificar molt l'especificació (i el codi) de `node_arb_max`.

Donada una jerarquia de nodes apuntada per un punter a `node_arbre`, definim la seva “jerarquia de màxims” de manera equivalent a l'arbre de màxims.

També farem servir la funció `max3(x,y,z)`, que retorna el màxim de 3 elements de tipus `T`.

```
// Pre: p != nullptr, i tot node de la jerarquia apuntada
//      per p té 0 o dos següents diferents de nullptr
// Post: el resultat apunta a la jerarquia de maxims de la jerarquia
//      apuntada per p
static node_arbre* node_arb_max(node_arbre* p) {
    // per la Pre, com que p != nullptr, el resultat apuntarà com a mínim
    // a un node
    node_arbre* n = new node_arbre;
    if (p->segE == nullptr) {
        // per la pre, p->segD també es igual a nullptr
        n->segE = n->segD = nullptr;
        n->info = p->info;
    } else {
        // p->segE != nullptr, p->segD != nullptr, podem fer crides recursives
        n->segE = node_arb_max(p->segE);
        n->segD = node_arb_max(p->segD);
        // HI: n->segE apunta a la jerarquia de maxims de la jerarquia
        //      apuntada per p->segE; n->segD apunta a la jerarquia de maxims
        //      de la jerarquia apuntada per p->segD;
        n->info = max3(n->segE->info, n->segD->info, p->info);
    }
    return n;
}
```

Ara programem la funció original fent servir la immersió. Noteu que la precondició d'`arbre_maxims` implica la de `node_arb_max`.

```
Arbre arbre_maxims(){
    Arbre a;
    a.primer_node = node_arb_max(primer_node);
    return a;
}
```