Segon examen parcial de PRO2 Durada: 2h 50m 12/6/2017

Cognoms Nom DNI

Observació: Heu de fer servir els espais indicats per entrar la resposta d'alguns apartats.

Problema 1 (5 punts)

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.

Donada una llista de enters l no buida, diem que una *escala* de l és una subllista e no buida on tot element de e, tret del darrer, és menor o igual que seu següent. El darrer element de e és, o bé més gran que el seu següent, o bé és el darrer element d'l.

Per exemple, si l és [-5, -6, -2, 2, 0, -1, 4, 2, 2, 6, 4, 8, 9, 7, 10, 6], les seves escales són [-5], [-6, -2, 2], [0], [-1, 4], [2, 2, 6], [4, 8, 9], [7, 10], [6].

Donada una llista l_1 no buida, es vol fer-ne una partició de forma que les seves escales parells (és a dir la segona, la quarta, etc) es moguin a una segona llista l_2. Tant els elements que es moguin com els que no, han de respectar el seu ordre original.

Exemple: Si abans de la partició l_1 és [1, 2, 1, 2, 3, 1, 2, 3, 4] i l_2 és buida, després de la partició l_1 hauria de ser [1, 2, 1, 2, 3, 4] i l_2 hauria de ser [1, 2, 3].

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

```
void split (Llista<int>& l)
/* Pre:  p.i. = P i no és una llista buida, l és buida */
/* Post: p.i. està formada per les escales senars de P i l està formada per
les escales parells de P, respectant l'ordre original dels elements;
el punt d'interès de cada llista queda al seu respectiu principi */
```

Ho volem fer amb un algorisme iteratiu, amb exactament dos bucles (no aniuats), completant l'esquema que veureu a continuació. La solució ha de ser eficient en temps i espai. **No es poden declarar més variables**. A la segona part només es poden escriure assignacions (i les dues condicions dels ifs). No es poden fer servir altres operacions de la classe.

Primera part:

```
node_llista *ant;
bool creixent = true;
int cont = 0;
[ ]
while (act≠NULL and creixent) {
    [ ]
}
```

En aquest punt, *act* ha d'apuntar al primer element de la segona escala de P (o és NULL, si P només té una escala); *ant* ha d'apuntar a l'element de P anterior a l'apuntat per *act* (o a l'ultim element de P, si *act* és NULL); per tant *ant* $\neq$ NULL i apunta a l'ultim element de la primera escala de P; *cont* és la mida de la primera escala de P.

A l'exemple, *ant* apuntaria al primer 2 i *act* al segon 1; *cont* seria igual a 2.

Segona part:

```
if (act ≠ NULL) {
    [ ]
    longitud = cont;
    l.longitud = 1;
    bool senar = false;
```

INVARIANT DEL SEGON BUCLE:

1) *act* apunta a un element del p.i. o és NULL; si *act* no és NULL *ant* apunta a l'element del p.i. o de *l* que era l'anterior d'*act* a P; (*)

2) entre *prim* i *ult* hi són totes les escales senars que apareixien a P abans que *act*, i el tros fins a *ant* (inclòs) de la que contenia *ant*, si aquesta és senar, totes en el seu ordre original;

3) entre *l.prim* i *l.ult* hi són totes les escales parells que apareixien a P abans que *act*, i el tros fins a *ant* (inclòs) de la que contenia *ant*, si aquesta és parell, totes en el seu ordre original;

4) *senar* indica si *ant* apareixia a una escala senar de P;

5) *longitud* és el nombre d'elements de P anteriors a *act* que pertanyien a escales sernars; *l.longitud* és el nombre d'elements de P anteriors a *act* que pertanyien a escales parells.

(*) Als punts 2-5 de l'invariant, on apareix un punter volem referir-nos a *l'element apuntat per* aquest punter.

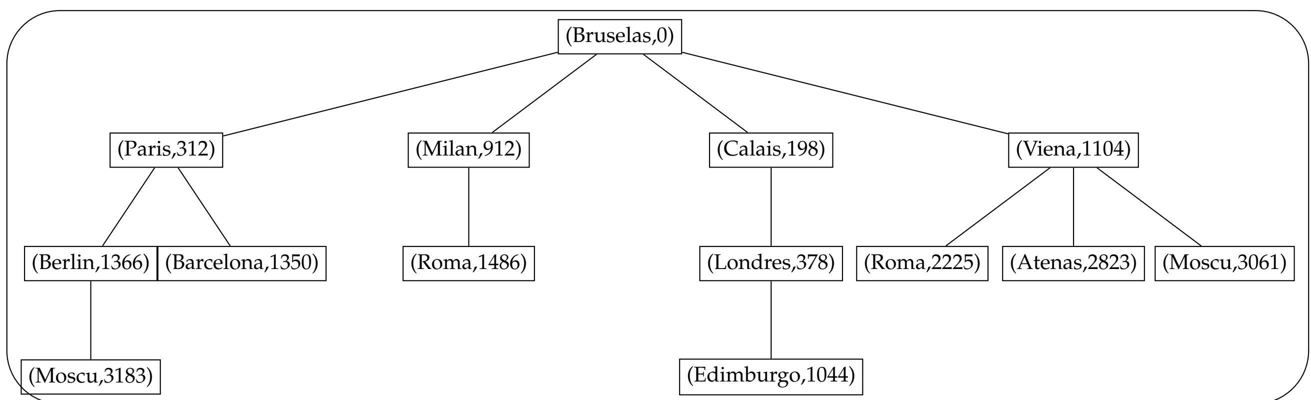
```
    while (act ≠ NULL) {
      if (senar) { // ant pertany a una escala senar
        if (                    ) { // act i ant pertanyen a la mateixa escala



        }
        else { // act i ant NO pertanyen a la mateixa escala



        }
      }
      else { // ant pertany a una escala parell
        if (                    ) { // act i ant pertanyen a la mateixa escala



        }
        else { // act i ant NO pertanyen a la mateixa escala



        }
      } // avancem el bucle


    } // instruccions posteriors al bucle
    ult→seg = l.ult→seg = NULL;
    l.act = l.prim;
  }
  act = prim;
}
```

Cognoms	**Nom**	**DNI**

Problema 2 (5 punts)

En aquest problema emprem un *arbre general* per representar possibles itineraris que pot fer un estudiant per la xarxa de ferrocarrils Europea, suposant que comença el viatge en una ciutat `alfa`. Cada node de l'arbre, juntament amb el camí que el connecta amb l'arrel de l'arbre, representa un itinerari en tren des de la ciutat origen `alfa` fins la ciutat a què fa referència el node. Concretament, cada node conté en el camp `info` un struct de tipus `Etapa` amb dos camps: 1) `ciudad`, que correspon al nom d'una ciutat connectada amb tren amb la ciutat origen `alfa`; i 2) `kms_recorridos`, que especifica la distancia en kilòmetres que hi ha entre la ciutat `alfa` i la que representa el node seguint l'itinerari associat al node.

Per exemple, l'arbre de la figura representa varis itineraris que l'estudiant podria fer sortint de Brussel·les. Observeu que pot arribar a una ciutat (e.g. Roma o Moscú) seguint diferents itineraris.



Un arbre d'itineraris és, per tant, un arbre general de dades tipus `Etapa` (un `ArbreGen<Etapa>`) en el què *el camí des del node arrel fins cada node representa un itinerari* que permet anar des de la ciutat associada al node arrel `alfa` a la ciutat que representa el node `omega`. Les diferents etapes d'un itinerari concret són les ciutats per las que es passa per anar des d'`alfa` a `omega` seguint l'itinerari, amb la particularitat de que en cada etapa s'anoten els kilòmetres recorreguts des de l'inici del viatge.

Problema 2.1 (3.5 punts): A partir d'un arbre d'itineraris amb arrel `(alfa,0)`, l'estudiant desitja trobar un itinerari que el permeti arribar des de la ciutat `alfa` a una ciutat donada `omega`. En primer lloc, es planteja construir un mètode que determini si existeix un itinerari que li permeti arribar a una ciutat donada `omega` i, si existeix, que també calculi la distància mínima que caldrà recórrer per arribar a `omega` utilitzant un dels itineraris de l'arbre. Concretament, es demana implementar eficientment el mètode **distancia**, especificat a continuació. No es poden utilitzar les operacions públiques de la classe `ArbreGen`.

```
pair<bool,int> distancia(string omega) const;
// Pre: Cert
// Post: La primera component del resultat té valor cert si existeix un node en el p.i.
// que representa una Etapa amb 'ciudad' igual a 'omega'; altrament, la primera component
// del resultat té valor fals. Si la primera component del resultat té valor cert, la
// segona component del resultat conté el mínim valor del camp 'kms_recorridos' dels
// nodes del p.i. que representen una Etapa amb 'ciudad' igual a 'omega'.
```

Si utilitzeu una operació auxiliar per implementar el mètode consultor `distancia`, haureu

- d'escriure la capçalera, precondició i postcondició de l'operació auxiliar (**0.75 punts**),
- implementar l'operació auxiliar (**2.25 punts**), i
- implementar el mètode consultor `distancia` utilitzant l'operació auxiliar (**0.5 punts**).

Per exemple, si `a` és una variable de tipus `ArbreGen<Etapa>` que representa l'arbre de la figura, el resultat de la crida `a.distancia("Moscu")` haurà de ser el parell `(true,3061)`, i el resultat de la crida `a.distancia("San_Petersburgo")` haurà de ser un parell amb la primera component `false`.

Donem a continuació les definicions del tipus de dades `Etapa` i de la classe `ArbreGen`.

```
struct Etapa {
  string ciudad;
  int kms_recorridos;
};

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
};
```

Problema 2.2 (5 punts o 1.5 punts): En segon lloc, l'estudiant es planteja estendre el mètode implementat a l'apartat anterior calculant també l'itinerari que minimitza la distancia a recórrer per arribar a una ciutat donada `omega`, cas que existeixi un itinerari en el paràmetre implícit que permeti anar en tren des de la ciutat origen a `omega`. Representem un itinerari amb una pila de dades tipus `Etapa` (és a dir, un `stack<Etapa>`) tal que els seus elements són les diferents etapes de l'itinerari de forma que l'ordre d'extracció dels elements de la pila coincideixi amb l'ordre en el que es recorren les etapes de l'itinerari que et porta des de la ciutat origen fins `omega`. Concretament, es demana implementar eficientment el mètode **itinerario**. No es poden utilitzar les operacions públiques de la classe `ArbreGen`.

IMPORTANT: Podeu lliurar tan sols la solució a l'apartat 2.2 si esteu segurs de que és la correcta.

```
pair<bool,int> itinerario(string omega, stack<Etapa>& ruta) const;
// Pre: ruta = RUTA és una pila buida.
// Post: La primera component del resultat té valor cert si existeix un node en el p.i.
// que representa una Etapa amb 'ciudad' igual a 'omega'; altrament, la primera component
// del resultat té valor fals. Si la primera component del resultat té valor cert, la
// segona component del resultat conté el valor del camp 'kms_recorridos' del node 'n_min'
// del p.i. que representa una Etapa amb 'ciudad' igual a 'omega' i el seu valor del camp
// 'kms_recorridos' és mínim. En cas d'empat, 'n_min' és el primer node del p.i. en pre-ordre
// amb aquestes característiques. A més, si la primera component del resultat té valor cert,
// el paràmetre 'ruta' conté la informació dels nodes que formen el camí que connecta el node
// arrel del p.i. amb el node 'n_min', és a dir, una de las rutes més curtes des de la ciutat
// representada pel node arrel del p.i. fins la ciutat 'omega'.
```

Per exemple, si `a` és una variable de tipus `ArbreGen<Etapa>` que representa l'arbre de la figura i `r` és una variable de tipus `stack<Etapa>` que representa una pila buida, el resultat de la crida `a.itinerario("Roma",r)` haurà de ser el parell `(true,1486)`, i després de la crida la variable `r` haurà de contenir la pila

(*Bruselas*, 0)
(*Milan*, 912)
(*Roma*, 1486)

De la mateixa forma el resultat de la crida `a.itinerario("Venecia",r)` haurà de ser un parell amb la primera component `false`.

Observació: En aquest apartat es permet l'ús de les operacions primitives `push`, `pop`, `top` i `empty` de la classe `stack` de la STL. També es pot utilitzar l'operació `swap` de la classe `stack`, que intercanvia els continguts del paràmetre implícit (p.i.) i del seu argument en temps constant. Per exemple, si `p1` i `p2` són variables de tipus `stack<Etapa>`, la instrucció `p1.swap(p2)` intercanvia els seus continguts.