

Titulació: Grau en Enginyeria Informàtica
Assignatura: Programació 2 (PRO2)
Duració: 2h 30m

Curs: Q1 2020–2021 (2n Parcial)
Data: 14 de gener de 2021

1. **(5 punts)** Considereu la següent representació amb apuntadors i memòria dinàmica per a una classe llista d'enters amb punt d'interès (*act*), encadenament doble i sense sentinella.

```
class LlistaInt {
private:
    struct node_llista {
        int info;
        node_llista* seg;
        node_llista* ant;
    };
    int longitud;
    node_llista* primer_node;
    node_llista* ultim_node;
    node_llista* act;
    ... // especificacio i implementacio d'operacions privades

public:
    ... // especificacio i implementacio d'operacions publiques
};
```

Volem incloure dins d'aquesta classe dues noves operacions públiques modificadores, *elim_repes_cons()* i *marca_canvis_signe()*, amb la següent especificació:

```
// Pre: la llista paràmetre implícit = L i no és buida
void elim_repes_cons();
// Post: la llista paràmetre implícit és com L, però havent eliminat els
// elements repetits consecutius de L

// Pre: la llista paràmetre implícit = L i no és buida i no conté zeros
void marca_canvis_signe();
// Post: la llista paràmetre implícit és com L, però per a cada canvi de signe entre
// dos elements consecutius de L, s'ha introduït un element zero al mig dels dos elements
```

Dins del codi d'aquestes dues operacions no s'ha de cridar a cap operació ni pública ni privada de la classe *LlistaInt*, ni de cap altra classe, només consultar i modificar els atributs i els nodes del paràmetre implícit.

Us proporcionem una plantilla del codi i un invariant **obligatori** per a cada operació. També us donem uns pocs exemples de resultats d'aplicació de les dues operacions.

Operació elim_repes_cons()

```
l = [21 -4 3 3 3 8 8 -12 3 3 3 3 -5 15]
l.elim_repes_cons() = [21 -4 3 8 -12 3 -5 15]
```

```
l = [21 -4 3 8 -12 3 -5 15]
l.elim_repes_cons() = [21 -4 3 8 -12 3 -5 15]
```

```
l = [1 1 1 1 1 1 1 1]
l.elim_repes_cons() = [1]
```

```
void elim_repes_cons() {
    node_llista* antact = ;
    act = ;
    while (  ) {
        // Inv: la subllista del parametre implicit fins a l'element apuntat
        // per antact inclos es com la subllista de L fins a l'element
        // anterior al punt d'interes marcat per act, pero havent
        // eliminat d'aquesta els elements repetits consecutius,
        // act = antact->seg,
        // act != nullptr => antact = act->ant

        node_llista* segact = act->seg;
        if (  ) {
            
        }
        else {
            
        }
    }
    ultim_node = ;
}
```

Noteu que `segact` no surt a l'invariant perquè és una variable local del cos del bucle.

Operació marca_canvis_signe()

```
l = [21 -4 3 3 3 8 8 -12 -5 15]
l.marca_canvis_signe() = [21 0 -4 0 3 3 3 8 8 0 -12 -5 0 15]
```

```
l = [1 2 3 4 3 2 1 -1 -2 -1]
l.marca_canvis_signe() = [1 2 3 4 3 2 1 0 -1 -2 -1]
```

```

l = [-1 -1 -1 -1]
l.marca_canvis_signe() = [-1 -1 -1 -1]

void marca_canvis_signe() {
    node_llista* antact = ;
    act = ;
    while (  ) {
        // Inv: la subllista del parametre implicit fins a l'element apuntat
        // per antact inclos es com la subllista de L fins a l'element
        // anterior al punt d'interes marcat per act, pero per a cada
        // canvi de signe entre dos elements consecutius d'aquesta,
        // hem introduit un element zero al mig dels dos elements;
        // act = antact->seg,
        // act != nullptr => antact = act->ant

        if (  ) {
            
        }

        antact = ;
        act = ;
    }
}

```

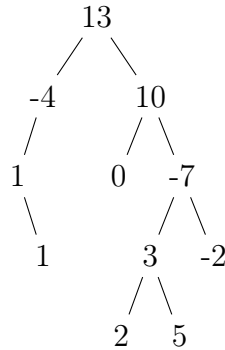
- a) (2,5 punts) Ompliu el codi faltant (només els llocs indicats per les capses) per a implementar eficientment l'operació `elim_repes_cons()`. Cada capsa s'ha d'omplir amb una expressió o amb una o més instruccions.
- b) (2,5 punts) Ompliu el codi faltant (només els llocs indicats per les capses) per a implementar eficientment l'operació `marca_canvis_signe()`. Cada capsa s'ha d'omplir amb una expressió o amb una o més instruccions.

2. (5 punts) Considerem la següent definició d'una classe `ArbreBinari` en C++

```
template <class T>
class ArbreBinari{
private:
    struct node {
        T info;
        node* esq;
        node* dre;
    };
    node* arrel; // apuntador a l'arrel de l'arbre
    ...
public:
    ...
    int frontisses() const;
    ...
};
```

Sigui n un element d'un arbre binari a i suposem que n està situat al node arrel d'un subarbre t de l'arbre a . Direm que n és un element *frontissa* de l'arbre a si es compleix que el valor de n és igual a la diferència entre la suma dels elements del fill dret de t i la suma dels elements del fill esquerre de t . Recordeu que la suma d'un conjunt buit o d'un arbre buit és zero.

Per exemple, en l'arbre binari següent



només existeixen 4 elements que són frontissa: l'arrel 13 (perquè $13 = 11 - (-2)$), l'element 1 situat més a prop de l'arrel ($1 = 1 - 0$), la fulla amb valor 0 ($0 = 0 - 0$) i l'element amb valor 3 ($3 = 5 - 2$).

Es demana implementar la funció pública `frontisses` amb l'especificació següent

```
// Pre: el tipus T dels elements de l'arbre paràmetre implícit té operador + de suma
```

```
int frontisses() const;
```

```
// Post: el resultat es el nombre d'elements frontissa de l'arbre paràmetre implícit
```

incloent l'especificació i implementació de qualsevol operació auxiliar privada que es cridi des d'ella.

Ni en el codi de **frontisses** ni en el de la o les operacions auxiliars d'aquesta es pot cridar cap operació pública de la classe **ArbreBinari** (per això no hi són a l'enunciat), només es pot accedir a la representació de la classe i cridar operacions privades noves.

- a)* (4 punts) Especifica i implementa en C++ un o més mètodes **static** privats de la classe **ArbreBinari** que ens permetin calcular eficientment el nombre d'elements frontissa d'un arbre binari.
- b)* (1 punt) Implementa el mètode públic **frontisses**, usant el o els mètodes privats definits a l'apartat anterior.