Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Cognoms, Nom | D.N.I.

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

Curs: Q1 2018–2019 (1r Parcial)
Data: 15 de Novembre de 2018

1. **(5 punts)** Estem fent l'anàlisi d'un text i desem les paraules que van apareixent junt amb les seves freqüències en una llista `lfrec`. La llista `lfrec` estarà sempre en ordre lexicogràfic ascendent. Suposem que volem actualitzar la informació d'una nova frase, que està emmagatzemada en una llista de strings auxiliar i en ordre lexicogràfic no descendent. Aquesta nova frase pot contenir paraules repetides. Durant l'actualització, si una paraula de la llista auxiliar no pertany a la llista `lfrec`, l'afegirem amb la freqüència que correspongui, i si hi pertany, llavors augmentarem la freqüència.

   L'operació tindrà la següent especificació:

```
/* Pre: lfrec = LFREC i aux estan ordenades lexicogràficament;
        lfrec no conté strings repetits */

void actualitzar_frequencies(list< pair<string,int> >& lfrec,
                             const list<string>& aux);
/* Post: 1) si x és un string d'aux no present a LFREC llavors
         el parell ⟨x,k⟩ estarà a lfrec, on k és el
         nombre de vegades que x apareix a aux
      2) si x és un string d'aux present a LFREC amb freqüència
         f llavors el parell ⟨x,f+k⟩ estarà a lfrec, on k
         és el nombre de vegades que x apareix a aux
      3) si el parell ⟨x,f⟩ està present a LFREC però
         x no pertany a aux, llavors el parell ⟨x,f⟩
         està a lfrec
      4) lfrec està en ordre lexicogràfic ascendent */
```

   Per exemple, si

$$\texttt{lfrec} = [\langle \texttt{a},2\rangle,\langle \texttt{amb},1\rangle,\langle \texttt{de},3\rangle,\langle \texttt{en},2\rangle,\ \langle \texttt{la},3\rangle]$$
$$\texttt{aux} = [\texttt{amic},\texttt{casa},\texttt{de},\texttt{de},\texttt{des},\texttt{es},\texttt{la},\texttt{la},\texttt{la},\texttt{meva}]$$

   llavors després de la crida `actualitzar_frecuencias(lfrec, aux)` tindrem

$$\texttt{lfrec} = [\langle \texttt{a},2\rangle,\langle \texttt{amb},1\rangle,\langle \texttt{amic},1\rangle,\langle \texttt{casa},1\rangle,\backslash$$
$$\langle \texttt{de},5\rangle,\langle \texttt{des},1\rangle,\langle \texttt{en},2\rangle,\langle \texttt{es},1\rangle,\langle \texttt{la},6\rangle,\langle \texttt{meva},1\rangle]$$

   Es demana:

(a) (1.5 punts) Implementa una solució iterativa, usant només un bucle, per a la següent operació:

```
/* Pre: aux és una llista no buida, ordenada lexicogràficament i
que pot contenir strings repetits. it = IT apunta a una posició
d'aux vàlida (it ≠ aux.end()). */

int comptar_repetits (const list<string>& aux,
                      list<string>::const_iterator& it);
/* Post: retorna el nombre de vegades que l'string apuntat per
l'iterador IT apareix a la subllista dels elements d'aux entre
IT i el final d'aux; it apunta al primer element
d'aux estrictament més gran que *IT en ordre lexicogràfic,
o it = aux.end() si no existeix un element així */
```

(b) (1 punt) Escriu l'invariant del bucle de la funció `comptar_repetits`.

(c) (0.5 punts) Escriu la funció de fita del bucle de la funció `comptar_repetits`.

(d) (2 punts) Implementa l'operació `actualitzar_frequencies`, de manera que sigui el més eficient possible. Es recomana utilitzar la funció `comptar_repetits`.

SOLUCIÓ:

(a)

```
int comptar_repetits (const list<string>& aux,
                      list<string>::const_iterator& it) {
    string s = *it;
    int k = 0;
    while (it != aux.end() and *it == s) {
      ++it;
      ++k;
    };
    return k;
}
```

(b) Invariant: $s = *IT \wedge$ tots els strings entre IT i l'anterior al apuntat per `it` són iguals a $s \wedge k$ és el nombre d'elements entre IT i l'anterior a `it` $\wedge\ s \leq$ `*it`.

(c) Funció de fita: el nombre d'elements entre `it` i `aux.end()`.

(d)

```
void actualitzar_frequencies(list< pair<string,int> >& lfrec,
                             const list<string>& aux){
  pair<string,int> p;
  list< pair<string,int> >::iterator it = lfrec.begin();
  list<string>::const_iterator it_aux = aux.begin();
  while(it_aux != aux.end()) {
```

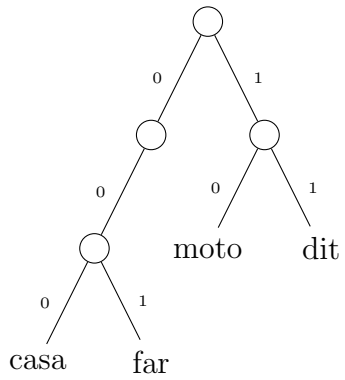
```
    if (it == l.end() or it -> first > *it_aux){
      p.first = *it_aux;
      p.second = comptar_repetits(aux,it_aux);
      l.insert(it,p);
    } else if (it -> first == *it_aux){
      ++(it -> second);
      ++it_aux;
    } else // it -> first < *it_aux
      ++it;
  }
}
```

2. **(5 punts)** Donat un arbre binari T, es diu que és un arbre de codificació d'un conjunt X d'$n \geq 0$ strings si i només si es compleixen les següents condicions:

   (a) L'arbre T té exactament n fulles (és a dir, nodes els subarbres dels quals són tots dos buits). Cadascuna de les n fulles conté un dels strings del conjunt X i tot element d'X està en alguna fulla.

   (b) Tots els nodes amb al menys un subarbre no buit contenen l'string buit.

   Per exemple, si $X = \{\texttt{casa}, \texttt{dit}, \texttt{far}, \texttt{moto}\}$ el següent arbre binari és un arbre de codificació per a X (hi ha d'altres arbres de codificació d'X possibles):



   Donat un arbre de codificació per al conjunt X podem assignar un *codi* a cadascun dels strings; per a un string $x \in X$ el seu codi consisteix en un string binari (amb 0s i 1s) que representa el camí (únic) entre l'arrel de T i la fulla que conté a X (0 = esquerra, 1 = dreta). A l'exemple anterior, el codi de `casa` és `000` i el codi de `moto` és `10`. Observeu que cap codi serà prefixe propi de cap altre codi, doncs els strings codificats sempre estan a les fulles. Observeu també que si un arbre de codificació només consta d'una fulla (una arrel amb dos subarbres buits) llavors el codi de l'string emmagatzemat a la fulla serà el codi buit (l'string de longitud 0).

   L'objetiu d'aquest exercici és dissenyar un procediment `obtenir_codis` que donat un arbre ens torni la llista de parells ⟨string, codi⟩ amb tots els strings i els seus corresponents codis, **en ordre lexicogràfic ascendent de codis**. La solució proposada ha de ser eficient: cap node de l'arbre de codificació hauria de ser examinat/visitat més d'un cop. A la vostra solució, utilitzeu les següents definicions de C++:

```
struct codificacio {
   string s;
   string codi; // el codi binari de l'string s
};
```

```
/* Pre: T és un arbre de codificació, C és una llista buida */

void obtenir_codis(const BinTree<string>& T, list<codificacio>& C);

/* Post: C conté la llista de codificacions dels strings
a l'arbre T, en ordre lexicogràfic creixent de codis */
```

Per exemple, amb l'arbre `T` de la figura i una llista `C` inicialment buida, després de la crida `obtenir_codis(T,C)` tindrem:

$$\texttt{C} = [\langle \texttt{casa,000} \rangle, \langle \texttt{far,001} \rangle, \langle \texttt{moto,10} \rangle, \langle \texttt{dit,11} \rangle].$$

Es demana:

(a) (3 punts) Especifica i implementa un nou procediment `i_obtenir_codis` amb una immersió de paràmetres que ens permiti resoldre de manera eficient l'obtenció dels codis d'un arbre.

(b) (1 punt) Argumenta la correcció del procediment `i_obtenir_codis` de l'apartat anterior.

(c) (1 punts) Implementa `obtenir_codis` mitjançant el procediment `i_obtenir_codis`.

N.B. Recordeu que si `s` i `t` són strings llavors `s + t` és l'string que s'obté concatenant `s` i a continuació `t`; de manera anàloga, si `c` és un caràcter, `s + c` és l'string que s'obté en afegir `c` al final de l'string `s`; tanmateix, l'"expressió" `c + s` no és vàlida en C++.

SOLUCIÓ:

(a) El procediment `i_obtenir_codis` afegirà a `C` els strings i els seus corresponents codis d'un cert subarbre t de l'arbre "original" T, precedint tots els codis amb el prefix `pref` que representa al camí des de l'arrel de T a l'arrel del subarbre t. Un recorregut preordre ens permetrà obtenir els string per ordre creixent de codi; els casos de base són que l'arbre sigui buit o que l'arbre consisteixi en una única fulla, llavors la codificació és trivial. Formalment:

```
/* Pre: t és un arbre de codificació,
        C està en ordre ascendent de codis,
        c < pref per a tot codi c de la llista C */

void i_obtenir_codis(BinTree<string>& t,
                     const string& pref,
                     list<codificacio>& C) {
  if (t.empty()) return;  // no fer res
```

```
  if (t.left().empty() and t.right().empty()) {
    // t té només una fulla, afegir el seu string amb codi = pref
    codificacio mc;
    mc.s = t.value();
    mc.codi = pref;
    C.insert(C.end(), mc);
  } else {
    i_obtenir_codis(t.left(), pref + '0', C);
    i_obtenir_codis(t.right(), pref + '1', C);
  }
}
/* Post: C conté el seu valor original i a continuació
         les codificacions dels strings de l'arbre t,
         però amb tots els corresponents codis precedits per
         pref; la llista C està en ordre ascendent
         de codis */
```

(b) Es demostra per inducció. Per la part de la base d'inducció tenim dos casos, que t sigui buit o que t només continguin una fulla. Si l'arbre t és buit no hi ha cap string (ni codi) que s'hagi d'afegir a C i, efectivament, el procediment no fa res. Si l'arbre consisteix en una (única) fulla, llavors l'arrel conté un string $s = t$.`value()` i el seu codi (dintre de l'arbre t) és l'string buit, llavors s'hauria d'afegir a C el parell $\langle s, \texttt{pref}\rangle$ i això és el que fa el procediment.

En el cas recursiu general, l'arrel no conté cap string a codificar. Tots els strings del subarbre esquerre tenen com a codi el que els hi pertoca a t.`left()` precedit d'un '0' i, anàlogament, els del subarbre dret tenen com a codi el que els hi pertoca però precedit d'un '1'. Assumint la correcció de les dues crides (ja que reben subarbres de mida inferior a la mida de t) s'hauran afegit a `C` tots els strings de t amb els seus codis precedits per `pref` i la llista `C` estarà en ordre ascendent, doncs primer afegim els codis de t que comencen amb `0` dins de t, i a continuació tots els codis que comencen amb `1` dins de t, que són tots ells més grans en ordre lexicogràfic. Fixeu-vos tambe que la precondició del procediment implica que es compleix la precondició amb la primera crida recursiva, ja que `pref` $<$ `pref` $+'0'$. Quan acaba aquesta crida s'hauran afegit en ordre ascendent els codis dels strings que hi ha al subarbre t.`left()`, tots precedits per `pref` $+'0'$, i la llista `C` està en ordre ascedent de codis. Donat que qualsevol codi que comença `pref` $+'0'$ és menor en ordre lexicogràfic que l'string `pref` $+'1'$ podem concloure que per a la segona crida recursiva

```
...
i_obtenir_codis(t.right(), pref + '1', C);
...
```

la precondició de `i_obtenir_codis(...)` també es compleix.

La recursió acaba quan la mida n de l'arbre és $n \leq 1$ (0 o 1 fulla). En el cas recursiu les crides es fan sobre subarbres de mida $< n$. Per tant el procediment acaba necessàriament, doncs a cada crida recursiva tenim progrés cap a un cas

de base.

(c)

```
void obtenir_codis(const BinTree<string>& T,
                   list<codificacion>& C) {
  i_obtenir_codis(T, "", C);
}
```