Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Cognoms, Nom | D.N.I.

Titulació: Grau en Enginyeria Informàtica
Assignatura: Programació 2 (PRO2)
Duració: 1h 30m

Curs: Q1 2021–2022 (1r Parcial)
Data: 10 de novembre de 2021

1. **Transformación de vector a BinTree** [5 puntos]

   Se dice que un árbol binario a es *completo* si todos sus niveles, excepto el más profundo, están llenos de nodos y el último nivel contiene el resto de sus nodos, sin ningún hueco, empezando por su extremo izquierdo. Una representación conveniente para un árbol binario completo es un vector v de $n+1$ posiciones, siendo n el número de nodos del árbol, tal que la posición 0 no se usa y los nodos que resultan del recorrido por niveles del árbol aparecen consecutivamente entre las posiciones 1 y n. Esta representación tiene las siguientes propiedades interesantes:

   - Para todo i entre 1 y n, si $2*i \leq n$ entonces $v[2*i]$ contiene el nodo raíz del hijo izquierdo no vacío del subárbol cuyo nodo raíz está en $v[i]$; pero si $2*i > n$ entonces el hijo izquierdo es vacío.
   - Para todo i entre 1 y n, si $2*i+1 \leq n$ entonces $v[2*i+1]$ contiene el nodo raíz del hijo derecho no vacío del subárbol cuyo nodo raíz está en $v[i]$; pero si $2*i+1 > n$ entonces el hijo derecho es vacío.
   - $2 \leq i \leq n$ implica que $v[i/2]$ es el nodo padre del nodo $v[i]$.

   También se puede decir que todo subárbol no vacío b de a es el árbol completo incluido en v a partir de la posición $i \geq 1$ tal que $v[i]$ es la raíz del subárbol b.

   Por ejemplo, si a es el siguiente árbol binario completo de enteros

```
          4
        /   \
       /     \
      3       9
     / \     / \
    1   5   4   2
   / \
  3   6
```

entonces $n = 9$ y el vector v es de tamaño $n + 1 = 10$ y su contenido es [* 4 3 9 1 5 4 2 3 6], donde v[0]=* significa cualquier valor entero (irrelevante).

En ocasiones, nos puede interesar, dado el vector v, construir una representación explícita de la estructura del árbol a, en nuestro caso mediante la clase genérica `BinTree`. Considerad la especificación de la siguiente operación:

// Pre: $v.size() \geq 2$
// Post: el resultado es el árbol completo incluido en v cuya raíz es $v[1]$

```
BinTree<int> vector_to_BinTree(const vector<int>& v);
```

Se pide:

(a) (2.5 puntos) Especifica (completando la cabecera y la Pre/Post) e implementa una función de inmersión recursiva

// Pre: $v.size() \geq 2$, ...
// Post: el resultado es ... si ..., y el resultado es ... si ...

```
BinTree<int> i_vector_to_BinTree(const vector<int>& v, ...);
```

que permita una solución directa de `vector_to_BinTree` mediante una única llamada a `i_vector_to_BinTree`.

(b) (0.5 puntos) Escribe el código de la operación original (no recursiva) `vector_to_BinTree`.

(c) (2 puntos) Justifica la corrección de la función de inmersión `i_vector_to_BinTree`, incluyendo la demostración de que termina siempre.

SOLUCIÓ:

(a) (2.5 puntos) Especifica (completando la cabecera y la Pre/Post) e implementa una función de inmersión recursiva `i_vector_to_BinTree` que permita una solución directa de `vector_to_BinTree` mediante una única llamada a `i_vector_to_BinTree`.

// Pre: $v.size() \geq 2$, $i \geq 1$
// Post: el resultado es un árbol vacío si $i \geq v.size()$, y es el árbol incluido en $v[i..v.size() - 1]$ cuya raíz es $v[i]$, si $i < v.size()$

```
BinTree<int> i_vector_to_BinTree(const vector<int>& v, int i) {
    if (i >= v.size())
        return BinTree<int>();
    else {
        BinTree<int> l,r;
        l = i_vector_to_BinTree(v,2*i);
        r = i_vector_to_BinTree(v,2*i+1);
        return BinTree<int>(v[i],l,r);
    }
}
```

(b) (0.5 puntos) Escribe el código de la operación original (no recursiva) `vector_to_BinTree`.

// Pre: $v.size() \geq 2$
// Post: el resultado es el árbol completo incluido en v cuya raíz es $v[1]$

```
BinTree<int> vector_to_BinTree(const vector<int>& v) {
    return i_vector_to_BinTree(v,1);
}
```

(c) (2 puntos) Justifica la corrección de la función de inmersión `i_vector_to_BinTree`, incluyendo la demostración de que termina siempre.

- *Terminación:* Tomamos como función de tamaño $f = v.size() - i$. Claramente, si $f \leq 0$ estamos en el caso base ($i >= v.size()$). Por otro lado, en cada llamada recursiva, la función f decrece porque, al ser $i \geq 1$ por la Pre, tanto $2 * i$ como $2 * i + 1$ son mayores siempre que i.
- *Caso base:* Si $i \geq v.size()$, el resultado ha de ser un árbol vacío, porque así lo especifica la Post.
- *Los argumentos en las llamadas recursivas cumplen la Pre:* Si $1 \leq i < v.size()$ entonces $2 * i \geq 1$ y $2 * i + 1 \geq 1$, mientras que el vector v no se modifica.
- *Caso recursivo:* Si $i < v.size()$, la Post dice que el resultado ha de ser el árbol incluido en $v[i..v.size() - 1]$ cuya raíz es $v[i]$. Por tanto, se ha de construir y retornar un árbol no vacío con $v[i]$ como raíz, un subárbol izquierdo que o bien no es vacío con raíz en $v[2 * i]$ o es vacío si $2 * i >= v.size()$, y un subárbol derecho que o bien no es vacío con raíz en $v[2*i+1]$ o es vacío si $2*i+1 >= v.size()$. Por hipótesis de inducción, las dos llamadas recursivas devuelven respectivamente un árbol izquierdo y un árbol derecho que cumplen dichas condiciones.

2. **Convolución** [5 puntos]

Dado un vector de enteros `x` que representa una señal discreta 1D de tamaño n y un vector de enteros `z` que representa un filtro 1D de tamaño m impar, tales que $n \geq m \geq 1$ y existe una $k \geq 0 : m = 2k + 1$, la *convolución* de `x` y `z` es una operación matemática que genera una señal `y` de salida resultado de procesar la señal de entrada `x` con el filtro `z`. Si se desea que el vector `y` tenga el mismo tamaño n que `x`, es necesario rellenar arbitrariamente algunos valores en los dos extremos de `y`, por ejemplo con ceros (*zero padding*). Concretamente, definimos la siguiente función abstracta CONV (que no es usable como parte de ningún código) de la siguiente manera:

$$
\begin{array}{lll}
\mathtt{y} = \mathrm{CONV}(\mathtt{x}, \mathtt{z}) \quad \text{si i sólo si} & & \\
\quad \forall i : 0 \leq i < k : & y[i] = 0 \\
\quad \forall i : k \leq i < n - k : & y[i] = \sum_{j=0}^{2k} z[j] * x[i - k + j] \\
\quad \forall i : n - k \leq i < n : & y[i] = 0
\end{array}
$$

Nótese que, excepto en los extremos, es necesario visitar la ventana de los m valores $\mathtt{x}[i - k..i + k]$ de la señal de entrada para calcular $\mathtt{y}[i]$, la salida en la posición i. Por ejemplo, si $\mathtt{x} = [3, 4, 2, 2, 2, 1, 0, 1, 3, 2, 1, 2]$ y $\mathtt{z} = [-1, 4, -2]$ entonces $\mathtt{y} = \mathrm{CONV}(\mathtt{x}, \mathtt{z}) = [0, 9, 0, 2, 4, 2, -3, -2, 7, 3, -2, 0]$; donde $n = 12$, $m = 3$, $k = 1$ y el primer elemento calculado es $y[1] = -1 * 3 + 4 * 4 - 2 * 2 = 9$.

Se desea diseñar un procedimiento `conv_modif` que recibe dos vectores de enteros `x` y `z` y calcula su convolución, pero guardando el resultado en el mismo vector `x`. Entonces nuestro procedimiento `conv_modif` puede especificarse asi:

// Pre: `x` = `X`, `x`.size() $\geq$ `z`.size() $\geq$ 1, `z`.size() es impar
// Post: `x` = CONV(`X`,`z`)

```
void conv_modif(vector<int>& x, const vector<int>& z);
```

Existen implementaciones correctas de `conv_modif` que o bien usan y copian un vector auxiliar de n elementos para el resultado o bien usan y desplazan un vector auxiliar de m elementos para guardar la ventana de `X` a visitar. Sin embargo, una solución algo más eficiente usa una lista auxiliar `w` de m elementos para guardar y actualizar dicha ventana. En ese caso, podremos usar la siguiente función auxiliar, `prod_escalar`, que os damos especificada e implementada:

// Pre: m = w.size() = z.size() ≥ 1, w $= [w_1, \ldots, w_m]$
// Post: el resultado es $\sum_{j=0}^{m-1} z[j] * w_{j+1}$

```
int prod_escalar(const list<int>& w, const vector<int>& z, int m) {
    int prod = 0, i = 0;
    list<int>::const_iterator it = w.begin();
    // Inv: ...
    // Cota = ...
    while (i < m) {
        prod += z[i] * (*it);
        ++it; ++i;
    }
    return prod;
}
```

Se pide:

(a) (2 puntos) Completa la implementación del procedimiento `conv_modif`, utilizando la plantilla que se proporciona a continuación y llamando cuando sea necesario a la operación auxiliar `prod_escalar`.

(b) (1 punto) Completa el invariante y la función de cota del segundo bucle `while`, el "principal", de `conv_modif`.

(c) (2 puntos) Escribe el invariante y la función de cota del bucle de la operación `prod_escalar` y justifica que `prod_escalar` siempre termina y es correcta.

SOLUCIÓ:

(a) (2 puntos) Completa la implementación del procedimiento `conv_modif`, utilizando la plantilla que se proporciona a continuación y llamando cuando sea necesario a la operación auxiliar `prod_escalar`.

```
void conv_modif(vector<int>& x, const vector<int>& z) {
   int n = x.size();
   int m = z.size();
   int k = m/2;
   list<int> w;
   // Inicializar la lista con el bucle for siguiente
   for (int j = 0;  j < m  ; ++j)
       w.insert(w.end(),x[j]);     // o w.push_back(x[j]);
   int i = 0;
   while ( i < k ) {  // zero padding entre i=0 y k-1
       x[i] = 0;
       ++i;
   }
   // A continuación el bucle "principal":
   while ( i < n-k-1 ) {           // para i entre k y n-k-2
       x[i] = prod_escalar(w,z,m);
       w.erase(w.begin());         // o w.pop_front();
       w.insert(w.end(),x[i+k+1]); // o w.push_back(x[i+k+1]);
       ++i;
   }
   x[i] = prod_escalar(w,z,m);     // para i=n-k-1
   ++i;
   while ( i < n ) {  // zero padding entre i=n-k y n-1
       x[i] = 0;
       ++i;
   }
}
```

(b) (1 punto) Completa el invariante y la función de cota del segundo bucle `while`, el "principal", de `conv_modif`.

```
// Inv: k <= i <= n-k-1,
//      w contiene los elementos de X[i-k..i+k] en el mismo orden,
//      para todo p tal que k <= p < i, se cumple que
//        x[p] = suma entre j=0 i j=2k de z[j]*X[p-k+j]
// Cota = n-k-1-i  // también sirven n-k-i o n-i,
//                    aunque son menos precisas
```

(c) (2 puntos) Escribe el invariante y la función de cota del bucle de la operación `prod_escalar` y justifica que esta operación siempre termina y es correcta.

- *Invariante*: $0 \leq i \leq m$, it referencia a w_{i+1} si $i < m$ o val $w.end()$ si $i = m$, $prod = \sum_{j=0}^{i-1} z[j] * w_{j+1}$
- *Función de cota*: $f = m - i$, también sirven $f = |w[it, w.end())|$ o $f = |w[it,:)|$
- *Terminación*: La condición $0 \leq i \leq m$ del invariante implica que $f = m - i >= 0$ siempre y $f > 0$ si se entra en el bucle (porque $i < m$). La función f decrece en 1 a cada iteración del bucle al incrementar la i.
- *Inicializaciones*: Asignando $i = 0$ se cumple $0 \leq i \leq m$ ya que $m \geq 1$ por la Pre. Asignando $it = w.begin()$ hacemos que it referencie a w_1, el primer elemento de w. Y como el sumatorio entre $j = 0$ y -1 es vacío, asignando $prod = 0$ ya se cumple todo el invariante.
- *Condición del bucle*: La negación de la condición del bucle es $i \geq m$. Esta condición conjuntamente con $i \leq m$ del invariante implica que, al salir del bucle, tendremos que $i = m$. Por tanto, por la última condición del invariante, $prod = \sum_{j=0}^{m-1} z[j] * w_{j+1}$, que es lo que requiere la Post para el resultado que se retorna ($prod$).
- *Cuerpo del bucle*: Para progresar hemos de sumar a $prod$ el producto $z[i] * w_{i+1}$, que es el siguiente término del sumatorio, y lo conseguimos haciendo $prod \mathrel{+}= z[i] * (*it)$, ya que $(*it) = w_{i+1}$ por el invariante ($it = w.end()$ solamente al salir del bucle, cuando $i = m$). Una vez añadido ese término, avanzando el iterador it e incrementando i se vuelve a cumplir el invariante.