Estructures de dades
Curs 2020-21
Práctica 2 - Códigos de Huffman - Sprint 2

17 d'abril de 2021

Durante el sprint anterior construimos la **tabla de frecuencias** a partir del texto contenido en el archivo *entrada.txt*.

Por lo tanto, esta semana nos encontramos en disposición de, por cada uno de los caracteres que aparecen en el texto, crear un **árbol binario** de un solo nodo donde su clave será el carácter y su valor será la frecuencia de aparición de este carácter en el texto. Todos estos árboles binarios deben insertarse en un **heap** (o **cola de prioridad**) donde el criterio de prioridad será la frecuencia de aparición del carácter (a menor frecuencia, más prioridad). Finalmente, utilizando el heap, mostraremos por consola todos los caracteres que aparecen en el texto ordenados por su frecuencia de aparición (de menor a mayor).

Por ejemplo, si el contenido de la tabla de frecuencias es el siguiente:

```
 :  6
a:  3
b:  1
d:  2
e:  6
f:  2
h:  1
i:  1
l:  1
m:  2
n:  2
o:  1
p:  1
r:  2
s:  1
u:  2
x:  2
```

Una vez finalizada la ejecución, la consola debería mostrar:

```
b:  1
i:  1
l:  1
h:  1
o:  1
p:  1
s:  1
n:  2
x:  2
u:  2
d:  2
f:  2
m:  2
r:  2
a:  3
 :  6
e:  6
```

1 Tareas a realizar

Las tareas a realizar durante esta semana son:

1. Implementar el **TAD árbol binario**.
2. Implementar el **TAD cola de prioridad**.
3. **Recorrer** el conjunto que contiene la tabla de frecuencias mediante un **iterador** y, por cada pareja clave-valor:
   (a) Crear un árbol binario de un solo nodo que contenga la pareja clave-valor.
   (b) Insertar el árbol binario de un solo nodo en la cola de prioridad.
4. Una vez todos los árboles se encuentren en la cola de prioridad, extraer todos los árboles de la cola y mostrar el contenido de su raíz por consola.

2 ¿Cómo creamos un árbol binario que contenga la pareja clave-valor?

El TAD árbol binario sólo puede contener un tipo *elem* en su raíz, por lo tanto, si queremos que nuestro árbol binario contenga una pareja clave-valor,

será necesario crear un tipo formado por dos elementos: una clave y un valor. Por ejemplo:

```
type node is record
  caracter: Character;
  frequencia: integer;
end record;
```

Y, en consecuencia, nuestra instancia genérica del TAD árbol binario debería concretar el elemento genérico *elem* con el tipo *nodo*:

```
package darbre is new darbolbinario(elem => node);
use darbre;
```

3 ¿Cómo creamos un árbol binario de un solo nodo?

Con la especificación del TAD árbol binario vista en teoría:

```
generic
  type elem is private;
package darbolbinario is
  type arbol is limited private;

  mal_uso: exception;
  espacio_desbordado: exception;

  procedure avacio(t: out arbol);
  function esta_vacio(t: in arbol) return boolean;
  procedure graft(t: out arbol; lt,rt: in arbol; x: in elem);
  procedure raiz(t: in arbol; x: out elem);
  procedure izq(t: in arbol; lt: out arbol);
  procedure der(t: in arbol; rt: out arbol);
private
  --depende implementacion
end darbolbinario;
```

La única forma de especificar el elemento que debe contener la raíz del árbol es mediante el procedimiento *graft()*, al que le tendremos que pasar por parámetro: el árbol vacío al que queremos asignarle el elemento como raíz y dos árboles vacíos más correspondientes a su hijo izquierdo y derecho, respectivamente.

4 ¿Cómo realizamos la instancia genérica del TAD heap?

El TAD heap o cola de prioridad tiene 4 elementos genéricos:

```
generic
  -- tamano maximo de la cola de prioridad
  size : positive;
  -- tipo de elementos que contendra la cola de prioridad
  type item is private;
  -- funcion que devuelve si el item x1 es menor que el item
   x2
  with function "<" (x1,x2 : in item) return boolean;
  -- funcion que devuelve si el item x1 es mayor que el item
   x2
  with function ">" (x1,x2 : in item) return boolean;
```

Recordad que para poder utilizar el TAD cola de prioridad, debe realizarse la instancia genérica del paquete *d_priority_queue* concretando cada uno de sus elementos genéricos (size, item, "<","[>]").

Los elementos genéricos "< ","[>]"son funciones que reciben dos parámetros de tipo **item** en modo **in** y devuelven un booleano. Para concretar estos elementos genéricos es importante que respetemos el número de parámetros, el tipo de los parámetros, el modo de los parámetros y el tipo de retorno; por lo tanto, podemos concretarlos con cualquier función que tenga dos parámetros en modo **in** con el **mismo tipo con el que concretemos el elemento genérico *item*** y que devuelva un booleano.

5 ¿Cómo creamos un heap de árboles binarios?

Para crear un heap de árboles binarios, deberíamos realizar la instancia genérica del TAD heap indicando que el tipo *item* se concreta con el tipo *arbol*:

```
package d_priority_queue_arbol is new d_priority_queue (
  size => 20,
  item => arbol,
  "<" => menor,
  ">" => major );
```

Pero, si lo hacemos así, obtendremos el error:

```
actual for non-limited "item" cannot be a limited type
```

Recuerde que nuestros TAD están definidos como ***limited*** *private*, y el tipo *arbol* es un **TAD**.

Por lo tanto, la solución consiste en crear un **tipo puntero al tipo árbol** y realizar el **heap de punteros de árboles binarios** (en lugar de un heap de árboles binarios):

```
-- Tipo puntero al tipo arbol
type parbol is access arbol;

package d_priority_queue_arbol is new d_priority_queue (
  size => 20,
  item => parbol,
  "<" => menor,
  ">" => major );
```

Debido a esto, nuestro programa principal no debe trabajar con árboles sino con punteros a árboles; es decir, no debe tener declarada ninguna variable de tipo *arbol* sino que debe declarar variables de tipo *parbol*.

Esto significa que estaremos trabajando todo el tiempo con punteros (direcciones de memoria), por lo tanto, recordad que cada vez que queramos crear un árbol deberemos reservar un bloque de memoria para él:

```
-- Variable de tipo puntero a 'arbol'
a: parbol;
...
-- Reserva un bloque de memoria
-- e identifica la direccion
a := new arbol;
```

Recordad también que para acceder o modificar el contenido de una variable referenciada por un puntero, se utiliza el calificador **all**:

```
-- El procedimiento avacio() espera un parametro
-- de tipo 'arbol' y no de tipo puntero a 'arbol':
-- Para acceder al arbol referenciado por el puntero 'a'
-- utilizamos el calificador 'all': a.all
avacio(a.all);
```