Laboratorio 1 de Programación 3

Leandro Rabindranath León

- **Este laboratorio es evaluado y la calificación que obtengas cuenta para tu calificación definitiva.**
- Los ejercicios de esta práctica requieren la instalación de la biblioteca $\aleph_\omega$ (Aleph-ω) y TCLAP.
- **Fecha de entrega: lunes 23 de marzo 2015 hasta las 6 pm**
- **Tu entrega consiste de un archivo llamado `test.H`, Por favor, al inicio de este archivo, en comentarios, pon los apellidos, nombres y números de cédula de los miembros de la pareja que está sometiendo el archivo.**

1. Introducción

El fin de este laboratorio es que practiques con la manipulación de secuencias sobre contenedores.

Básicamente, un contenedor es un conjunto de elementos con las operaciones de inserción, búsqueda. eliminación y otras más según el ámbito práctico. Dicho de otro modo: un contenedor es una clase que resuelve el problema fundamental de estructuras de datos. Hay muchas maneras de implementar un contenedor de elementos: arreglos, listas, árboles, etc y a ello se dedicará la mayor parte del curso. Pero en lo que a esta práctica refiere, lo que nos concierne es interpretar el contenedor como una secuencia de elementos y en función de ello resolver algunos ejercicios.

En este laboratorio usaremos listas enlazadas simples como estructura de dato para representar un contenedor. En $\aleph_\omega$ (Aleph-ω) el tipo de dato que representa una lista simple es `DynList<T>`, donde `T` es el tipo de elementos que guarda la secuencia.

No es esencial que conozcas cómo se implementa una lista enlazada simple. Lo que sí es esencial es que conozcas cómo interactuar con ella.

1.1. Iteradores

Un patrón de uso muy útil para la manipulación de un contenedor es llamado "iterador". Este patrón permite recorrer uno a uno los elementos del contenedor. En $\aleph_\omega$ (Aleph-ω) todo contenedor contiene una subclase (es decir, una clase dentro de otra clase) que representa el iterador. El siguiente segmento de código imprime todos los elementos de una lista `l` de números enteros:

```
for (DynList<long>::Iterator it(l); it.has_curr(); it.next())
  cout << l.get_curr();
```

A efectos de ayudarte a comprender este código, desglosémoslo en un `while`:

```
DynList<long>::Iterator it(l);   // (1)
while (it.has_curr())            // (2)
  {
    cout << l.get_curr() << " "; // (3)
    it.next();                   // (4)
  }
```

La línea (1) declara un iterador sobre una lista de enteros llamada `l`. Inicialmente, el iterador está posicionado sobre el primer elemento de la secuencia.

El "elemento actual" sobre el cual está posicionado el iterador se accede mediante el método `get_curr()` (instrucción (2)).

Para avanzar el iterador en una posición se emplea el método `next()` (instrucción (4)). Un `next()` sobre el último elemento de la secuencia deja al iterador fuera de la secuencia. Si se ejecuta `next()` o `get_curr()` sobre un iterador fuera de secuencia, entonces se dispara la excepción `overflow_error`. Es pues muy importante que esto no ocurra.

Para saber si un iterador está posicionado sobre un elemento o si éste está fuera de la secuencia se invoca al método `has_curr()` (instrucción (2)), el cual retorna `true` si el iterador está posicionado sobre un elemento de la secuencia, o `false` si está fuera de la secuencia.

Una manera de aprehender el poder del patrón iterador es compararlo con un `for` que itera sobre los elementos de un arreglo.

Una explicación más detallada sobre este patrón de uso puede encontrarse en la sección 2.3, páginas 72-74 del libro.

Nota que el patrón iterador fuerza el tratamiento del conjunto como una secuencia, independientemente de la implementación. Esto por supuesto no siempre es deseable, pues según la índole de la estructura de dato pueden hacerse procesamientos más eficientes. Sin embargo, para los efectos de está práctica y la las listas enlazadas simples, eso no tiene gran importancia.

1.2. Programación funcional sobre secuencias

Una manera alterna, aunque a menudo tan eficiente como un iterador y mucho más concisa, la constituye el uso de métodos funcionales. Todos los contenedores de $\aleph_\omega$ (Aleph-ω) comparten estos métodos.

El mismo procesamiento de la sección anterior puede expresarse como sigue:

```
l.for_each([] (long i)
  {
    cout << i << " ";
  }); // <- Nota que aquí se cierra paréntesis del método for_each()
```

El método `for_each()` recorre cada elemento `i` de la secuencia `l` e invoca sobre `i` a una "función de primer orden" la cual es recibida como parámetro. Si deseas entender mejor este concepto entonces te recomendamos que tomes un curso de lenguajes de programación o de programación funcional. Por lo pronto, basta con saber que una función de primer orden es una función anónima (no tiene nombre) que es parámetro de alguna función `x` y que será llamada por `x`. En nuestro ejemplo la función de primer orden es:

```
[] (long i)
  {
    cout << i << " ";
  }
```

La cual imprime el valor de `i` y, si te fijas bien, es el parámetro de `for_each()` (nota el paréntesis de cierre de la función).

Este estilo de programación es un tanto distinto al uso de iteradores, pero tiende a ser más conciso, y por tanto más fácil, pero sobretodo, es más expresivo y seguro en el sentido de que hay menos posibilidad de equívocos. La crítica es que a veces el código resultante puede ser más difícil de entender.

Este estilo de programación es la base de la programación concurrente y con mucha probabilidad devendrá más mucho popular en el futuro conforme más se popularicen las plataformas y sistemas paralelos y distribuidos.

En la práctica (la del preparador) se te dará una lista y explicación de la mayoría de los métodos funcionales.

Todos los problemas planteados en esta práctica son resolubles con métodos funcionales en a lo sumo dos líneas de código; la inmensa mayoría en sólo una. Sin embargo, si eres nuevo con esta técnica, entonces te recomendamos que resuelvas los problemas con iteradores.

2. Laboratorio

Las funciones que debes implementar se encuentran en el archivo `test.H`. El nivel requerido es el de un curso inicial de programación.

Puedes hacerla mediante iteradores o mediante métodos funcionales, estos últimos permiten resolver cada problema en una o dos líneas de código. En todo caso, selecciona el estilo que mejor comprendas.

Junto con este enunciado se distribuyen los siguientes archivos:

- `test.C`: contiene una breve prueba de las funciones que debes implementar. Compílalo y ejecuta:

  ```
  ./test --help
  ```

  Para que leas una breve descripción de su uso.

  Se te recomienda que estudies la función `build_list()` y los operadores $<<$ definidos. Ellos te darán una idea del uso de la listas y del método `for_each()`.

  Para que puedas compilar este programa debes tener instalada `TCLAP`.

- `Makefile`: simple makefile para compilar `test`. Editalo y pon los valores de las variables `ALEPH` y `CXX` a los requeridos según tu instalación, y ejecuta

  ```
  make test
  ```

  para compilar el test.

- `test.H`: plantilla dónde escribirás tu código.
- `lab-01.pdf`: este archivo que estás leyendo.

2.1. Búsqueda de un elemento módulo x

Implementa la siguiente función:

```
tuple<long, bool> find_mod_x(const DynList<long> & l, long x)
```

La cual retorna una dupla cuyo primer elemento es el primer elemento de `l` que sea exactamente divisible entre `x`. Si existe tal elemento, entonces el segundo elemento de la dupla resultante es `true`. De lo contrario el primer valor de la dupla es indeterminado y el segundo es `false`.

2.2. Lista de elementos divididos entre x

Instrumenta la siguiente función:

```
DynList<long> divide_by_x(const DynList<long> & l, long x)
```

La cual retorna una nueva lista consistente de cada elemento de `l` dividido entre `x`. El orden resultante debe corresponderse exactamente con el orden de `l`.

2.3. División de los elementos de una lista entre x

Instrumenta:

```
DynList<long> & transform_divided_by_x(DynList<long> & l, long x)
```

La cual divide todos los elementos de `l` entre `x`. El valor de retorno es la misma lista `l`.

Nota que esta función es distinta de la anterior pues modifica toda la lista `l` (la anterior retorna una copia).

2.4. Lista de elementos divisibles entre x

Programa:

```
DynList<long> divisible_by_x(const DynList<long> & l, long x)
```

La cual retorna una lista de los elementos de `l` que son divisibles entre `x`. El orden de aparición en el resultado debe corresponderse con el mismo orden de `l`.

2.5. ¿Son todos divisibles entre x?

Programa:

```
bool are_all_divisible_by_x(const DynList<long> & l, long x)
```

La cual retorna `true` si todos los elementos de `l` son divisibles entre `x`. Si algún elemento no es divisible o la lista está vacía entonces el resultado es `false`.

2.6. ¿Hay un elemento divisible entre x?

Programa:

```
bool exist_divisible_by_x(const DynList<long> & l, long x)
```

La cual retorna `true` si al menos un elemento de `l` es divisible entre `x`. De lo contrario -ninguno es divisible entre `x`- retorna `false`.

2.7. Suma de todos los elementos

Programa:

```
long long sum(const DynList<long> & l)
```

La cual retorna la suma de todos los elementos de `l`.

2.8. Pares de elementos cuya suma es x

Programa:

```
DynList<tuple<long, long>> pairs_whose_sum_is_x(const DynList<long> & l1,
                                                const DynList<long> & l2,
                                                const long x)
```

La cual retorna una lista de duplas de elementos pertenecientes a `l1` y `l2` cuya suma es igual a `x`.

Por ejemplo, si $l_1 = [2, 1, -4, 10]$ y $l_2 = [11, 2, -6, 3]$, entonces el resultado de `pairs_whose_sum_is_x(` $l_1, l_2, 13$ `)` es $[(2, 11), (10, 3)]$.

Precaución y consejo: el evaluador tiene un máximo tiempo de ejecución. Por tanto, programa esta rutina iterando simultáneamente sobre las dos listas. Puedes asumir que las listas tienen el mismo tamaño, aunque esto es indiferente a la lógica del problema.

2.9. Lista de múltiplos

Programa:

```
 DynList<long>
multiples_of(const DynList<long> & l, const DynList<long> & mults)
```

La cual retorna una lista de elementos pertenecientes a `l` que son múltiplos de algún elemento de la lista `mults`.

2.10. Valores contenidos en un rango

Programa:

```
 DynList<tuple<long, size_t>>
range_pos(const DynList<long> & ll, long l, long r)
```

La cual retorna un lista de duplas de elementos de `l` contenidos en el rango $[l, r]$. El primer elemento de una dupla es el valor de un elemento de `l` que está dentro del rango. El segundo elemento de la dupla es la posición ordinal dentro de la lista. El orden de aparición de los pares debe corresponderse con el orden de aparición en `l`.

Por ejemplo, si $l = [3, 5, 7, 1, 13, 15, 20, 6]$ y el intervalo es $[5, 15]$ entonces el resultado es $[(5, 1), (7, 2), (13, 3), (15, 4), (11, 6)]$.

3. Evaluación

La fecha de entrega de este laboratorio es el lunes 23 de marzo de 2015 hasta las 6 pm. Puedes entregar antes, recibir corrección e intentar cuantas veces prefieras (limitado por supuesto a nuestra capacidad computacional).

Para evaluarte debes enviar el archivo `test.H` a la dirección:

`leandro.r.leon@gmail.com`

Antes de la fecha y hora de expiración. El "subject" debe **obligatoria y exclusivamente** contener el texto **PR3-LAB-01**. Si fallas con el subject entonces probablemente tu laboratorio no será evaluado, pues el mecanismo automatizado de filtrado no podrá detectar tu trabajo. **No comprimas `test.H` y sólo envía ese archivo**. La nota que se te adjudicará será la de tu último intento.

En el cuerpo del mensaje pon los nombres, apellidos y cédula de la pareja que somete.

Por favor, cuando sometas varias veces, no hagas reply de un correo previo, pues el sistema lo asume como un hilo.

En el mensaje y en el inicio del fuente del programa debes colocar los nombres, apellidos y números de cédula de la pareja que somete el laboratorio. **Por favor, no uses otros medios para enviar la solución ni escribas otros comentarios adicionales en el email**.

La evaluación esta dividida en dos partes. La primera verifica la correctitud de tus rutinas. La segunda es el estilo y elegancia de código.

Atención: si tu programa no compila, entonces el evaluador no compila. Si una de tus rutinas se cae, entonces el evaluador se cae. Si una de tus rutinas cae en un lazo infinito o demora demasiado, entonces el evaluador cae en un lazo infinito o se demora demasiado. Exactamente sucedería lo mismo si tu programa fuese parte de un sistema de aviónica: ¡se estrella el avión!. Por esa razón, en todos estos casos será imposible darte una nota, lo que simplemente se traduce en que tienes cero. Si una de tus rutinas dispara una excepción inesperada, el evaluador no la capturará y tu programa abortará enteramente.

No envíes programas que no compilan o se caen. Hacen perder los tiempos de red, de cpu, el tuyo y el mío. Si estás al tanto de que una rutina se te cae y no la puedes

corregir, entonces trata de aislar la falla y disparar una excepción cuando ésta ocurra. Si no logras implementar una rutina, entonces haz que dé un valor de retorno el cual, aunque estará incorrecto y no se te evaluará, le permitirá al evaluador proseguir con otras rutinas. De este modo no tumbarás al evaluador y eventualmente éste podría darte nota para algunos casos (o todos si tienes suerte). Si no logras aislar una falla, entonces deja la rutina vacía (tal como te fue dada), pero asegúrate de que dé un valor de retorno. De este modo, otras rutinas podrán ser evaluadas.

4. Recomendaciones

1. *"¡El tiempo no espera a nadie!"*[1]. Así que comienza este laboratorio lo más pronto posible. Aprovecha la evaluación automática y los múltiples intentos y somete tu solución cuando menos 48 horas antes del plazo de entrega. De este modo, tendrás tiempo para corregir y no meramente ser evaluado. Recuerda que una evaluación no tiene mucho sentido de aprendizaje si no hay corrección.

2. Usa el foro para plantear tus dudas de comprensión. Pero de ninguna manera compartas código, pues es considerado **plagio**.

   Por favor, **no uses el correo electrónico ni el correo del facebook para plantear tus dudas. ¡Usa el foro!**

3. Para todos los problemas, **¡todos!**, uno o más procedimientos de solución son explicables en lenguaje coloquial. Es decir, dada la entrada del problema, es fácilmente explicable cómo resolverlo. Así que te recomendamos fehacientemente, especialmente si estás en el curso de PR3, que te asegures de poder explicar correcta y completamente la solución y probarla para varios casos de prueba manuales. Una vez hecho esto, puedes comenzar a codificar el algoritmo.

4. Si no tienes gran experiencia con la programación funcional, entonces plantea tu solución, al menos la inicial, con iteradores.

[1] Maquiavelo.