Laboratorio 6 de Programación 3

Leandro Rabindranath León

- **Este laboratorio es evaluado y la calificación que obtengas cuenta para tu calificación definitiva.**
- **Fecha de entrega: lunes 6 de julio 2015 hasta las 6 pm**
- **Tu entrega consiste de un archivo llamado `stats.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.**
- En este laboratorio el desempeño en velocidad aportará aproximadamente el 40 % de la calificación.
- En este laboratorio eres enteramente libre de diseñar tus estructuras de datos.

1. Introducción

El fin de este laboratorio es que aprehendas la importancia y uso de estructuras de datos en las cuales la posición en el orden de claves y su acceso rápido es esencial para el desempeño. Para tales efectos codificarás un simple recolector de muestras estadísticas el cual permitiría calcular estadísticas básicas rápidamente, aún para conjuntos muy grandes.

Para la realización de este laboratorio requieres:

1. La suite clang
2. La biblioteca $\aleph_\omega$ (Aleph-ω).

1.1. Estadísticas básicas

Por simplicidad en este trabajo sólo usaremos números enteros.
La estadísticas a calcular son:

Mínimo: El menor elemento de la muestra.

Primer cuartil; El elemento situado en la posición ordenada $n/4$, dónde n es el tamaño de la muestra[1].

[1]Por simplicidad este término y el 3er cuartil se llevan a un valor discreto.

Media: la media aritmética o promedio de los datos de la muestra

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Desviación estándar: la raíz cuadrada de la varianza, dónde

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 = \left(\frac{1}{n}\sum_{i=1}^{n} x_i^2\right) - \bar{x}^2 \tag{1}$$

Mediana: El elemento situado en la posición $\lfloor n/2 \rfloor$, si el tamaño de la muestra es impar, o $\frac{x_{n/2}+x_{n/2+1}}{2}$, si es par.

Tercer quartil: El elemento situado en la posición $\frac{3}{4}n$

Máximo: El mayor elemento de la muestra.

Tamaño: El número de elementos de la muestra.

1.2. Otras operaciones

Aparte de la construcción y copia de una muestra, caracterizada por el tipo `Sampler` , la mayoría de las operaciones consisten en extracción mutable o no de submuestras ordenadas; es decir, de una muestra subconjunto de otra muestra.

Por ejemplo, el método:

```
DynList<long> extract_position(size_t l, size_t r) const
```

Retorna una lista ordenadas de las muestras entre la posición `l` y `r`. Por ejemplo, si la muestra es $\{1, 3, 4, 7, 9, 11, 14, 19\}$, entonces `extract_pos(2,5)` retorna $\{4, 7, 9\}$.

1.3. Esquema simple de solución

Un esquema de solución bastante simple, que debes seriamente considerar para tu implementación, consiste en almacenar las muestras desordenadas en un arreglo. De este modo, la inserción de una muestra es sumamente veloz, $\mathcal{O}(1)$ y con bajo coste constante.

En este caso, en la mayoría de los cómputos se ordenar el arreglo de muestras y, en función de los índices en el arreglo, se efectúan los cómputos necesarios.Por ejemplo, el primer cuartil, la mediana y el tercer cuartil se encontrarían en las posiciones del arreglo $n/4$, $n/2$ y $3n/4$ respectivamente.

El problema de desempeño con este enfoque, pero no de simplicidad de instrumentación, está dado por la necesidad de repetir el ordenamiento si la muestra se modifica: se añaden o eliminan elementos. En este caso, el coste por ordenar puede ser bastante severo, especialmente si la muestra se modifica muy a menudo.

1.4. Esquema de solución de alto desempeño

El esquema de alto desempeño y escala subyace en el empleo de árboles binarios de búsqueda equilibrados con rangos para almacenar los elementos de la muestra. En este caso, la inserción tiende a $\mathcal{O}(\lg n)$, pero no se requiere ordenar el arreglo. El acceso por posición de ordenamiento proporciona un coste $\mathcal{O}(\lg n)$, que es el coste que se pagaría por obtener el primer cuartil, la mediana y el tercer cuartil respectivamente. Además, la extracción de un subconjunto de tamaño m podría hacerse con complejidad $\mathcal{O}(\lg n)$, en lugar de $\mathcal{O}(n \lg n) + \mathcal{O}(m)$ que se requeriría si la muestra se almacena en un arreglo.

El problema con este enfoque es doble. En primer lugar, el uso de árboles sólo se compensa con escalas de muestras muy grandes, del orden de los millones de elementos; con escalas menores el arreglo es tan o más rápido. En segundo lugar, la instrumentación con árboles es más compleja, especialmente para la extracción de subconjuntos.

Por supuesto, un enfoque intermedio es perfectamente permitido. Por ejemplo, podrías usar árboles binarios de búsqueda más rápidos, pero ejecutar las operaciones de extracción por su recorrido infijo.

1.5. Dilema instrumental

Uno de los fines de que en este laboratorio el desempeño tenga una ponderación importante es presentarte y forzarte a confrontar con un dilema ingenieril que es más frecuente de lo que se podría presumir: ¿debo hacer una implementación simple y probablemente más segura, o una de alto desempeño y escala, pero más propensa a errores y más difícil de depurar?

$\aleph_\omega$ (Aleph-ω) soporta dos tipos de árboles con rango:

1. `Rand_Tree`: árbol aleatorizado.

2. `Treap_Rk`: un treap con rangos.

   Este tipo de árbol no ha sido estudiado en el curso, pero es matemáticamente equivalente a un árbol aleatorizado y su coste constante es una tres veces menor.

Nuestra implementación de referencia, con la que serás evaluado, se basa en árboles aleatorizados.

La manera más propicia y simple de usar estos árboles es a través de la clase `DynSetTree`. Por ejemplo, la declaración

```
DynSetTree<long, Rand_Tree> tree;
```

Declara un conjunto de elementos de tipo `long` representado con árboles aleatorizados.

Para manejar árboles con claves repetidas usa los métodos cuyo sufijo sea `dup`.

Pondera en tu dilema que si hay muchas repeticiones entonces el desempeño de la inserción en un árbol binario de búsqueda puede degradarse.

2. Laboratorio

Tu trabajo consistirá en implementar la clase `Sampler` , ubicada en el archivo `stats.H`.

Algunas de los métodos que implementarás no serán directamente evaluados, pero si serán usados por el evaluador. Por consiguiente, es esencial que estén correctos, pues de lo contrario te arriesgas a que no pases algunos casos de prueba.

2.1. Constructores

Implementa el constructor por omisión y el siguiente de copia:

```
Sampler(const Sampler & s)
```

Implementa también:

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

El cual recibe una lista de elementos a poner en la muestra. Aunque este constructor no es evaluado, sí es usado por algunas pruebas del evaluador.

2.2. `swap()`

En caso de que uses árboles con rango y optes por una implementación de alto desempeño, entonces puede serte muy útil implementar:

```
void swap(Sampler & s)
```

El cual intercambia en $\mathcal{O}(1)$ el contenido de `this` con `s`.

2.3. Comparación

Implementa:

```
bool operator == (const Sampler & s) const
```

El cual compara dos muestras, `this` y `s` y retorna `true` si ambas muestras contienen exactamente la misma cantidad de elementos.

2.4. Tamaño de la muestra

Programa:

```
size_t size() const
```

El cual retorna el tamaño de la muestra.

2.5. Media

Programa

```
long double mean() const
```

El cual retorna la media aritmética de la muestra.

2.6. Varianza

Programa

```
long double var() const
```

El cual retorna la varianza de la muestra. Usa la fórmula (1).

2.7. Obtención de elemento

Programa

```
long get_sample(size_t i)
```

El cual retorna el i-ésimo menor elemento de la muestra.

2.8. Añadidura de elemento

Programa

```
void add_sample(long sample)
```

El cual añade el valor de `sample` a la muestra.

2.9. Lista de muestras

Programa

```
DynList<long> list() const
```

El cual retorna una lista ordenada de todos los elementos de la muestra.

2.10. Estadísticas básicas

Programa

```
BasicStats stats() const
```

El cual retorna un objeto de tipo `BasicStats`. Este objeto está completamente definido en el archivo `common.H`.

2.11. Lista por posición

Instrumenta

```
DynList<long> get_by_position(size_t l, size_t r) const
```

El cual retorna una lista de los elementos comprendidos en el rango $[l, r)$. Atención, cualquier rango $[i, i)$ es válido y consiste del elemento en la posición i.

Si `l` es mayor que `r` entonces se debe generar la excepción `domain_error(‘‘crossed indexes’’)`.

Si el tamaño de la muestra es cero entonces se debe generar la excepción `underflow_error(‘‘empty sample’’)`.

Si alguno de los índices es mayor o igual que el tamaño de la muestra entonces se debe generar la excepción `out_of_range(‘‘index out of range’’)`.

Es propicio el momento para introducirte una ligera dificultad, la cual sólo se te presentará si optas por una implementación de alto desempeño basada en árboles con rangos. En este caso, podrías extraer el rango deseado mediante el método `split_pos()`. No obstante, y he aquí la dificultad, el método está declarado como inmutable (el cualificador `const`). Para poder extraer el rango mediante `split_pos()` es necesario modificar temporalmente el objeto y después restaurarlo a su estado original. En C++, y es probable que sea uno de los pocos lenguajes dónde esto sea permitido, puedes, a tu cuenta, riesgo y responsabilidad, cambiar temporalmente el carácter constante de un objeto mediante un cast y modificar el objeto. El compromiso es que restaures el objeto al estado original antes de la modificación, pues si no, las premisas de generación de código objeto serán comprometidas.

Aparte de informar al compilador, la rutina se declara `const` porque no modifica al objeto. La lista resultante es una copia.

Una alternativa es que tú elimines el cualificador `const` de la declaración. Este enfoque, aunque más sincero, plantea dos problemas potenciales. El primero es que al asumir el objeto es modificable el compilador podría dejar de efectuar algunas optimizaciones. El segundo es que, aunque el evaluador no verifica explícitamente por el cualificador, éste podría no compilar. Es a tu riesgo si optas por esta alternativa.

En general, salvo por la herencia de clases y eventualmente situaciones como esta presente, el casting es indeseable y debe usarse con máxima prudencia.

2.12. Muestra por posición

Programa

```
Sampler sample_by_position(size_t l, size_t r) const
```

El cual retorna una muestra con los elementos de `this` comprendidos en el rango $[l, r)$.

Si `l` es mayor que `r` entonces se debe generar la excepción `domain_error(‘‘crossed indexes’’)`.

Si el tamaño de la muestra es cero entonces se debe generar la excepción `underflow_error(‘‘empty sample’’)`.

Si alguno de los índices es mayor o igual que el tamaño de la muestra entonces se debe generar la excepción `out_of_range(‘‘index out of range’’)`.

2.13. Obtención de elementos por lista

Programa

```
DynList<Spair> get_by_key(const DynList<long> & l) const
```

El tipo `Spair` se define como:

```
using Spair = tuple<long, long>;
```

En este caso `get_by_key()` recibe como entrada una lista de claves a buscar dentro de la muestra. El método retorna una lista de pares donde el primer elemento es la clave y el segundo es la posición de la clave dentro de la muestra. Si el elemento no se encuentra en la muestra, entonces la posición debe ser `-1`.

La lista resultado debe estar ordenada por la clave, esté o no dentro de la muestra.

2.14. Obtención de elementos por clave

Programa

```
DynList<long> get_by_key(long sl, long sr) const
```

El cual retorna una lista ordenada de las claves que son iguales o mayores que `sl` y menores o iguales que `sr`.

Nota, y ten mucho cuidado con una eventual confusión, que los valores `sl` y `sr` **no son posiciones respeto a la secuencia ordenada**, tal como en el método anterior.

Nota que un camino solución de esta rutina consiste en usar la anterior y luego construir la muestra a través del constructor que recibe la lista. Es una manera muy simple de resolver el problema, una sola línea, pero acarrea un coste por la copia de elementos (2 copias). Este dilema también se te presentará en otras rutinas. Es a ti la decisión, ¡pondérala!.

Si `sl >sr` entonces se debe generar la excepción `domain_error(‘‘crossed keys’’)`.

2.15. Muestra por clave

Instrumenta

```
Sampler sample_by_key(long sl, long sr) const
```

El cual retorna una nueva muestra consistente de los elementos de `this` que son iguales o mayores que `sl` y menores o iguales que `sr`.

Si `sl >sr` entonces se debe generar la excepción `domain_error(‘‘crossed keys’’)`.

2.16. Corte por clave

Instrumenta

```
Sampler & cut_key(long sl, long sr)
```

La cual elimina de `this` todos los elementos que son menores que`sl` y mayores que `sr`.

Si `sl >sr` entonces se debe generar la excepción `domain_error(‘‘crossed keys’’)`.

Nota que en este y en el siguiente método el objeto es mutable, es decir, será modificado.

2.17. Corte por posición

Programa

```
Sampler & cut_position(size_t l, size_t r)
```

El cual elimina de `this` todos los elementos fuera del intervalo $[l, r)$]

Si `l` es mayor que `r` entonces se debe generar la excepción `domain_error(‘‘crossed indexes’’)`.

Si el tamaño de la muestra es cero entonces se debe generar la excepción `underflow_error(‘‘empty sample’’)`.

Si alguno de los índices es mayor o igual que el tamaño de la muestra entonces se debe generar la excepción `out_of_range(‘‘index out of range’’)`.

3. Evaluación

La fecha de entrega de este laboratorio es el lunes 6 de julio de 2015 hasta las 6 pm. Puedes entregar antes, recibir corrección hasta un máximo de 4 intentos.

Para evaluarte debes enviar el archivo `stats.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-06**. 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 `stats.H` y sólo envía ese archivo**.

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 tres partes. La primera y segundas son automatizadas. La primera parte verifica la correctitud de tus rutinas. La segunda consiste en un test global de desempeño. En ambas parte el programa evaluador ejecuta casos de prueba que se cotejan con implementaciones de referencia. Para cada una de tus rutinas se ejecuta una serie de pruebas (aproximadamente 10) y se comparan con las pruebas y salidas de las implementaciones de referencia. La más mínima desavenencia acarrea penalidades. Por cada test que falle, se te reportará la entrada que causó la falla.

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 hará lo posible por capturarla y reportártela; sin embargo, esto no es un compromiso; podría estrellarse el avión. En todo caso, si el evaluador no logra capturar la excepción, no se te dará nota pero se te reportará la excepción.

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.

La tercera parte es subjetiva y manual. Consiste en revisar tu fuente con la intención de evaluar tu estilo de codificación. Es recomendable que sigas las reglas de estilo comentadas en clase, pues éstas armonizan con la persona que subjetivamente te evaluará. Esta parte aporta aproximadamente el 10 % de la nota.

4. Recomendaciones

1. Lee enteramente este enunciado antes de proceder al diseño e implementación. Asegúrate de comprender bien tu diseño.

   Haz un boceto de tu estructura de dato y cómo esperas utilizarla. Por cada rutina, plantea qué es lo que vas hacer, cómo vas a resolver el problema. Este es una situación que amerita una estrategia de diseño y desarrollo.

2. La distribución del laboratorio contiene un pequeño test. No asuma que tu implementación es correcta por el hecho de pasar el test. Construye tus casos de prueba, verifica condiciones frontera y manejo de alta escala.

3. Este es un problema en el cual los refinamientos sucesivos son aconsejables. La recomendación general es que obtengas una correcta versión operativa lo más simplemente posible. Luego, si lo prefieres, puedes optar por mejorar el rendimiento.

4. Si usas árboles con rango y estás dispuesto a sacarles el máximo provecho, entonces, en las operaciones inmutables saca una copia de `this` antes de proceder a modificar temporalmente el objeto. Luego, cuando lo restaures, coteja el estado restaurado con la copia. Esta técnica te puede ayudar a detectar bugs. No olvides eliminar la copia en la versión que someterás.

5. Si decides apostar por los arreglos, ten muchísimo cuidado con caer en una complejidad cuadrática, pues con certitud harás que expiren los máximos tiempos

de ejecución del evaluador y éste abortará sin poder evaluar tu laboratorio. Especialmente, medita concienzudamente sobre el método de ordenamiento que vayas a utilizar.

6. Ten cuidado con el manejo de la memoria. Especialmente si usas árboles con rangos y `split()`. El evaluador manejará muestras de millones de elementos. Así que si dejas huecos de memoria (memory leaks), entonces te arriesgas a que el evaluador caiga en una situación de "trashing", lo que de seguro impactará sobre el tiempo de ejecución.

   valgrind es un buen amigo.

7. 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 Piazza!**

8. No incluyas headers en tu archivo `stats.H` (algo como `# include ...`), pues puedes hacer fallar la compilación. Si requieres un header especial, el cual piensas no estaría dentro del evaluador, exprésalo en el foro para así incluirlo.