Segundo examen integrador - Programación II
Lunes Noche - 1C 2025

Profesor Monzón, Nicolás Alberto

3 de junio de 2025

Requisitos para aprobar el integrador

A. Por actos de deshonestidad académica será sancionado.

B. Para poder aprobar este examen Ud. deberá estar administrativamente en condiciones de poder rendir el examen. En caso de no estarlo y rendir el examen, este no va a ser corregido y quedará anulado.

C. Deberá comprimir la carpeta `src` de su proyecto junto a los `txt` utilizados (también se admiten `pdf` y `MD`). El archivo comprimido deberá tener como nombre el número de grupo. Deberá enviarlo por mail a `nimonzon@uade.edu.ar`.

D. Deberá desarrollar un set de prueba para demostrar que funciona su código (no se piden test unitarios, pero si alguna prueba en el `main` del proyecto).

E. Solo podrá utilizar técnicas vistas en este curso. Está prohibido el uso de librerías, estructuras de datos abstractas que vienen por defecto en la JRE y el uso de genéricos. No respetar este punto es suficiente para desaprobar el integrador.

F. Deberá ser entregado dentro de las 3 horas y 30 minutos. a partir del horario de inicio.

G. CONDICIONES PARA APROBAR: el examen se mide con una escala logarítmica donde obtener al menos 60 puntos corresponderá a un 4 y se deberá cumplir con todos los puntos anteriores. En caso de no cumplir con alguno de los puntos anteriores, el examen queda desaprobado sin excepción.

Marco teórico

Existe un isomorfimo entre la lógica clásica y la teoría de conjuntos de manera sintáctica. Esto significa que podemos convertir $p \vee q$ a $A \cup B$, $p \wedge q$ a $A \cap B$ y $\neg p$ a A^c. Pero dada una de estas expresiones, por ejemplo $p \vee q$ y $A \cup B$, notamos que para $p \vee q$ tenemos 4 posibilidades, pero para $A \cup B$ infinitas. Una manera de resolverlo es plantear que el equivalente a `false` es el conjunto vacío y el equivalente a `true` el conjunto universal. Esto provoca una pérdida de información ya que los existen más posibilidades si utilizamos conjuntos.

Existe una forma de representar unívocamente un conjunto mediante la lógica clásica. Podemos pensar que si `x` es un número entero posible en Java, $\mathcal{P}(\texttt{x}) = \texttt{x} == \texttt{1}$, si lo evalúo en $\texttt{x} = \texttt{1}$ obtenemos `true` y si no, `false`.

De esta manera, podemos construir recursivamente predicados de la forma

$$\mathcal{P}(\texttt{x}) \ = \mathcal{P}(\texttt{a}) \wedge \mathcal{P}(\texttt{b}) \mid \mathcal{P}(\texttt{a}) \vee \mathcal{P}(\texttt{b}) \mid \neg\mathcal{P}(\texttt{a}) \mid \texttt{x} == \texttt{n}$$

Caso a caso tenemos

- $\mathcal{P}(a) \wedge \mathcal{P}(b)$ representa la intersección de dos conjuntos.
- $\mathcal{P}(a) \vee \mathcal{P}(b)$ representa la unión de dos conjuntos.
- $\neg\mathcal{P}(a)$ representa el complemento del conjunto que representa $\mathcal{P}(a)$.

Notar que contamos con que $\neg(\texttt{x} == \texttt{a}) \Leftrightarrow \texttt{x} \mathrel{!}= \texttt{a}$, y las leyes de Morgan.

El conjunto vacío se puede representar como $\neg\mathcal{P}(\texttt{x}) \wedge \mathcal{P}(\texttt{x})$ y el conjunto universal como $\neg\mathcal{P}(\texttt{x}) \vee \mathcal{P}(\texttt{x})$.

Agregar un elemento a a un conjunto A es equivalente a la unión del conjunto singleton $\{1\}$ y A. Remover un elemento b del conjunto A es equivalente a la diferencia de conjuntos $A - \{b\}$, y recordemos que $A - B = (A \cap B)^C$.

De ahora en adelante, el conjunto universal será denotado como $\mathcal{U}$ y representará cualquier valor de tipo `int` en Java.

Ejercicio 1 (10 %)

Crear los siguientes TDAs:

- `UnitaryPredicate` que deberá tener dos atributos: un enumerado `UnitaryPredicateEnum` y un valor.
- `BiPredicate` que deberá tener tres atributos: un enumerado `BiPredicateEnum` y dos valores.
- `AtomicPredicate` que almacenará un número entero.

Además

- `UnitaryPredicateEnum` tendrá 2 valores: `IDENTITY`, `NOT`.
- `BiPredicateEnum` tendrá 2 valores: `AND`, `OR`.

`AtomicPredicate` para un número entero `a` representa el predicado

$$\mathcal{P}(\texttt{x}) = \texttt{x} == \texttt{a}$$

Los tres TDAs deberán implementar una interfaz `Predicate`, y sus valores tendrán como tipo también `Predicate`. Esta interfaz será

```
1 public interface Predicate {
2
3     boolean eval(int a);
4
5 }
```

Notar que si el predicado representa un conjunto A, `eval` para un valor a representa la evaluación $a \in A$.

Ejercicio 2 (20 %)

Utilizar `Predicate` para crear una implementación de `Set` y justificar si es estática o dinámica, destructiva o no. No se permite utilizar otra estructura adicional como un arreglo, nodo, lista, pila, etc.

En el caso del método `choose`, no hará falta devolver un número aleatorio pero deberá diseñar una estrategia para devolver un número en el caso de que se pueda. En el caso de que no se les ocurra una estrategia, pueden usar alguna del siguiente listado:

- Hacer recursión estructural. Simple, pero no lo vimos en la materia, tienen que investigar.
- Agregar un booleano como marca de agua para detectar al conjunto universal.
- Trabajar con números enteros del rango $[0, 1000)$, y así poder iterar todos los valores. Esto mismo serviría para detectar el conjunto universal

Esta misma estrategia deberá servir para el método `isEmpty`.
Algunos ejemplos del uso de `Predicate`:

```
6  Predicate p = new AtomicPredicate(3); // {3}
7  Predicate p2 = new AtomicPredicate(5); // {5}
8  Predicate p3 = new BiPredicate(BiPredicateEnum.AND, p, p2);
       // {}
9  Predicate p4 = new BiPredicate(BiPredicateEnum.OR, p, p2);
       // {3, 5}
10 Predicate p5 = new BiPredicate(BiPredicateEnum.AND, p4, p);
       // {3}
11 Predicate p6 = new UnitaryPredicate(UnitaryPredicateEnum.NOT
       , p); // U - {3}
```

Ejercicio 3 (20 %)

Escribir una función que optimice un predicado, utilizando 3 de las siguientes propiedades. Indicar claramente cuáles se usaron.

$$\left(A^C\right)^C = A$$
$$A \cup A = A$$
$$A \cup \mathcal{U} = \mathcal{U}$$
$$A \cup \emptyset = A$$
$$A \cap A = A$$
$$A \cap \mathcal{U} = A$$
$$A \cap \emptyset = \emptyset$$
$$A - A = \emptyset$$
$$A - \mathcal{U} = \emptyset$$
$$A - \emptyset = A$$

Tip: Usando un getter sobre un predicado contenido en otro predicado, si creamos un conjunto a partir de ese predicado (quizás con un constructor privado que reciba como parámetro el predicado) podemos aprovechar su método `isEmpty` *para saber si estamos frente a un conjunto vacío. Para el conjunto universal, recurrir a la definición que dimos para este.*

Ejercicio 4 (20 %)

Sean los pares (p, q) de una cola con prioridad, suponer que representa una función que cumple $f(p) = q$ con las restricciones que esto representa. Utilizar como base la cola con prioridad vista en clase para crear `SpecialPriorityQueue` que debe contar con un método que indique si existe un *pico*. Supongamos una secuencia como $[1, 2, 3, 4, 3, 2, 4, 6, 9, 8]$, tenemos a 4 y a 9 como picos, ya que respecto a su elemento anterior cada uno crece, mientras que respecto al posterior decrece.

Calcular la complejidad computacional de este método.

Tip: Imitar búsqueda binaria.

Ejercicio 5 (10 %)

Escribir un método `intersection` que calcule los elmentos comunes entre

1. Una pila y una cola.
2. Las prioridades y valores de una cola con prioridad.
3. Una pila de duplas y una cola con prioridad.

El ítem a realizar será el resultado de sumar los legajos, calcular el resto de dividir por 3 y sumar 1 al resultado.

Ejercicio 6 (10 %)

Modificar cola con prioridad para que no admita prioridades repetidas. Además

1. Usar búsqueda binaria en el método `add`.
2. Usar búsqueda binaria en el método `remove`.
3. Calcular la complejidad computacional.

El ítem a realizar será el resultado de sumar los legajos, calcular el resto de dividir por 3 y sumar 1 al resultado.

Ejercicio 7 (10 %)

Crear un método que reciba una pila y un elemento, y agregue el elemento en el fondo si no existe en la pila. El único tipo de ciclo permitido es la recursión.