UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN
INSTITUTO DE INVESTIGACIONES EN MATEMÁTICAS APLICADAS
Y SISTEMAS

ALGORITMOS CONCURRENTES POR CONJUNTOS DE COLAS Y PILAS CON MULTIPLICIDAD

TESIS
QUE PARA OPTAR POR EL GRADO DE
MAESTRO EN EN CIENCIAS E INGENIERÍA DE LA COMPUTACIÓN

PRESENTA:
JOSÉ DAMIÁN LÓPEZ DÍAZ

TUTOR:
DR. ARMANDO CASTAÑEDA ROJANO
INSTITUTO DE MATEMÁTICAS, UNAM

CIUDAD UNIVERSITARIA, CD. MX, OCTUBRE, 2023

Índice general

Capítulo 1

Introducción

Dentro del área computacional se ha comenzado a experimentar un cambio en la forma de fabricar chips, renunciando a aumentar la velocidad con la que funcionan los procesadores, esto debido al sobrecalentamiento que implica el aumento de velocidad. En cambio, las arquitecturas multinúcleo han tenido un gran auge, debido a su diseño es posible evitar el sobrecalentamiento que genera un solo núcleo que trabaja a altas velocidades. Las arquitecturas multinúcleo se pueden analizar con un modelo de cómputo concurrente [14]. Además, dentro de este modelo, se pueden analizar problemas como el desarrollo de arquitecturas de microservicios [16]. Esto implica que, para aumentar la eficiencia, el objetivo cambie a explotar el paralelismo en lugar de aumentar la velocidad del reloj, lo cual es uno de los desafíos más destacados en cómputo moderno.

Los algoritmos en estos sistemas, *algoritmos concurrentes*, son un desafío pues, en los sistemas computacionales, los procesos de los algoritmos concurrentes se comunican a través de una memoria compartida, con el objetivo de solucionar colectivamente un problema. En estos sistemas la comunicación es asíncrona, lo cual significa que la comunicación entre procesos puede verse interrumpida o retrasarse de forma aleatoria debido a interrupciones u otros eventos [23, 20, 25], aún más, el comportamiento asíncrono provoca un número exponencial de posibles ejecuciones.

Los retrasos en los sistemas asíncronos son inherentemente impredecibles y varían ampliamente, provocando que los algoritmos sean propensos a fallas. Esto también implica que las implementaciones correctas de contadores, colas, pilas, grupos y otras estructuras de datos concurrentes requieran una gran cantidad de sincronización entre los procesos [15, 10]. Sin embargo, esto disminuye el rendimiento y la escalabilidad, siendo en algunos casos inevitables este costo [11, 4, 3].

1.1. Contexto

Un aspecto importante en la teoría de la computación es la especificación y verificación de los programas, descrito por medio de la *corrección* del programa. A diferencia de un programa secuencial, la corrección de los programas concurrentes es más compleja debido a su

naturaleza asíncrona. La noción estándar de corrección para los programas concurrentes está dada por la condición de corrección de linealizabilidad, la cual consiste en analizar el orden parcial de las operaciones y extenderlo a un orden total [18, 15]; sin embargo, no es única, existen otras nociones de corrección como la serialización o consistencia secuencial [2, 19]. Muchas veces no son necesarias todas las garantías que ofrece una especificación secuencial linealizable [23].

Recientemente, se ha buscado mejorar el rendimiento de objetos concurrente relajando su semántica. En particular, varios estudios se han centrado en la relajación de colas y pilas, logrando mejoras significativas en el rendimiento [12, 13, 17]. Una relajación de estas estructuras es la multiplicidad [8, 7, 9]. Intuitivamente, en una estructura con multiplicidad, ciertos conjuntos de operaciones simultáneas actúan de modo que el *estado* del objeto cambie como si solo una sola operación hubiera sido ejecutada. Dicho de otro modo, múltiples operaciones actúan como una misma [21].

Las estructuras con esta semántica relajada (multiplicidad) no son objetos secuenciales, pues no puede ser definido por un autómata secuencial, en cambio, una estructura con multiplicidad se especifica formalmente mediante un autómata secuencial por conjuntos. En un autómata secuencial, las transiciones de estado están dadas por la ejecución de alguna operación, la cual cambia el estado de la estructura a un nuevo estado. Mientras que en un autómata secuencial por conjuntos, cada transición está dada por la ejecución de un conjunto de operaciones concurrente, que simultáneamente provocan la transición del estado a uno nuevo. Por otro lado, mientras que la linealizabilidad se utiliza para verificar la corrección de la implementación de una especificación secuencial, para verificar la corrección de una implementación de una especificación secuencial por conjuntos se utiliza la condición de corrección de linealización de conjunto [6, 22].

1.2. Motivación

Dado que todo algoritmo linealizable tiene limitaciones inherentes para algunas estructuras de datos concurrentes [15, 10], lo cual se denomina costo inherente de la linealizabilidad, se ha optado por implementar estructuras de datos con semántica relajada. De modo que los algoritmos, en lugar de demostrar un comportamiento estricto, se permite que proporcionen resultados que no son precisamente los que daría un algoritmo secuencial. Diversos estudios han encontrado que siguiendo este enfoque es posible diseñar e implementar algoritmos concurrentes muy eficientes [8].

Uno de los problemas más complejos que surgen, con los algoritmos concurrentes, es el de coordinar las operaciones de modo que la memoria compartida se encuentre en un estado consistente con la especificación. Por ejemplo, si se busca implementar una cola concurrente, el algoritmo debe ser capaz de mantener en todo momento una representación consistente de una cola en la memoria compartida, a pesar de las diferentes modificaciones concurrentes por parte de los procesos que puedan suceder.

En una cola con multiplicidad, en pocas palabras, se permite que dos o más operaciones tipo *Dequeue* retornen el mismo valor que fue puesto anteriormente en la cola, pero solo si las operaciones ocurrieron concurrentemente. La noción de multiplicidad se propuso en [9, 8], donde se estudiaron sus propiedades más básicas, y después en [7] se mostró que efectivamente es útil para obtener algoritmos eficientes para versiones restringidas de colas y otras estructuras de datos conocidas como work-stealing.

1.3. Objetivos

El presente trabajo consiste en continuar la línea de investigación iniciada en [9, 7], es decir, entender si la noción de multiplicidad ayuda a desarrollar algoritmos más eficientes de pilas y colas. De forma concreta, el objetivo es tomar dos o tres algoritmos conocidos de pilas y colas (sin multiplicidad), y modificarlos para que implementen pilas y colas con multiplicidad, y probar experimentalmente si hay alguna ganancia en cuanto al desempeño. Los principales objetivos son:

1. Estudiar la teoría de computación concurrente necesaria para desarrollar el proyecto. Basándonos en [14], se estudiarán y seleccionarán un par de algoritmos concurrentes para implementar la noción de multiplicidad en estos.

2. Utilizando la noción de semántica relajada de multiplicidad de [9], se modificarán algunos algoritmos concurrentes de pilas y colas; no se exploraron otras estructuras relajadas como aquellas que usan linealizabilidad por intervalos.

3. Se demostrará formalmente que las implementaciones de algoritmos concurrentes por conjuntos de pilas y colas son implementaciones de pilas y colas con multiplicidad; es decir, que satisfacen la definición formal de autómata secuencial por conjuntos.

4. En relación con el punto anterior, se demostrará otras propiedades sobre los algoritmos, como condiciones de progreso y que son implementaciones linealizables por conjuntos [8].

5. Se codificarán todos los algoritmos en Java, junto con las versiones linealizables, y se ejecutarán para medir y comparar el desempeño de los mismos.

1.4. Contribución

Como primera contribución se presenta el algoritmo Set-Queue, el cual es una implementación linealizable por conjuntos de una cola con multiplicidad. Este algoritmo está inspirado en el algoritmo *LockFreeQueue* de [14], siendo su versión con semántica relajada.

Como segunda y tercera contribución se presentan los algoritmos Set-Stack-Logic y Set-Stack-Physic, los cuales son implementaciones linealizables por conjuntos de una pila con multiplicidad. Ambos algoritmos están inspirados en el algoritmo *LockFreeStack* de [14], siendo estas modificaciones no triviales, con semántica relajada, a la versión linealizable. La diferencia entre el algoritmo Set-Stack-Physic y Set-Stack-Logic radica en la representación del estado de

la pila, de manera que la representación del estado para Set-Stack-Logic la hemos denominado *estado lógico* y para Set-Stack-Physic el estado de la pila está representado por el *estado Físico*.

Como contribución general se presenta un análisis detallado de todos los algoritmos anteriores. Lo cual se resume en las demostraciones formales de que son algoritmos linelizables por conjuntos, implementaciones de las colas o pilas con multiplicidad y son *non-blocking*, propiedades que se definen formalmente más adelante.

1.5. Organización de la Tesis

En el capítulo 2 se presenta el modelo de computación concurrente que fue empleado en el presente trabajo. Dentro de la sección 2.1 se definen los conceptos base relacionados con la concurrencia, como hilos, memoria compartida y sección crítica. En la sección 2.2 se definen formalmente las condiciones de progreso y algunos conceptos clave. El modelo de cómputo concurrente se presenta formalmente en 2.3 y las condiciones de correctitud: *Linealizabilidad* y *Linealizabilidad por conjuntos*, se presentan en 2.4.

En el capítulo 3 se presenta la primera contribución: el algoritmo 4: Set-Queue. En la sección 3.1 se define formalmente una pila con multiplicidad; el cual es un autómata secuencial por conjuntos. Se presenta y discute el algoritmo *LockFreeQueue*, véase [14], en la sección 3.2 y el algoritmo Set-Queue en la sección 3.4.

En el capítulo 4 se presentan la segunda y tercera contribución, el algoritmo 7: Set-Stack-Logic y el algoritmo 9: Set-Stack-Physic. En la sección 4.1 se define formalmente una cola con multiplicidad; el cual es un autómata secuencial por conjuntos. Se presenta y discute el algoritmo *LockFreeStack*, véase [14], en la sección 4.2, el algoritmo Set-Stack-Logic en la sección 4.3 y el algoritmo Set-Stack-Physic en la sección 4.4. Finalmente, en el capítulo 6 se presentan las conclusiones.

Capítulo 2

Modelo de Cómputo Concurrente

Los sistemas multiprocesadores que se comunican a través de memoria compartida (también llamados recientemente multinúcleos) presentan fuertes desafíos en su programación, y esto es incluso independientemente de su tamaño [14]. Podemos tomar como ejemplo, a muy pequeña escala, los procesadores dentro de un solo chip que necesita coordinar el acceso a ubicaciones de memoria compartida, y a gran escala, se puede mencionar a los procesadores en una supercomputadora, los cuales necesitan coordinar el envío de datos.

La programación en sistemas multiprocesadores es un desafío, pues los sistemas modernos son de naturaleza asincrónica, es decir, que la actividad puede detenerse o retrasarse sin previo aviso debido a interrupciones, errores de caché, fallas y otros eventos, este tipo de comportamiento debe de tomarse de cuenta en los modelos de computación, pues la interrupción de un proceso podría conllevar graves fallas en un programa. Estas fallas son inherentemente impredecibles y pueden variar ampliamente en severidad, de modo que podrían llegar a retrasar desde unas cuantas instrucciones hasta cientos de millones, o lo que es peor, podrían producir un bloqueo del sistema o programa. Esto nos lleva a la necesidad de especificar un modelo de computación el cual aproveche la capacidad de multiprocesamiento.

2.1. Programación concurrente

La concurrencia, en pocas palabras, se refiere a la ejecución de varios procesos al mismo tiempo en un mismo programa; más adelante se detallan estos conceptos. Dentro del ámbito de la programación, la concurrencia permite potenciar la capacidad de cómputo mediante procesos concurrentes. Esto es importante, ya que las velocidades de reloj de los procesadores ya no aumentan. En cambio, en sistemas multiprocesadores, se obtienen más núcleos con cada nueva generación de chips, por lo que el cómputo concurrente se vuelve indispensable para explotar toda la capacidad de cómputo de estos sistemas.

2.1.1. Procesos e hilos

Los módulos concurrentes en sí vienen en dos tipos diferentes: procesos e hilos. Un proceso es la ejecución de un programa, es decir, los datos e instrucciones están cargados en la memoria principal, ejecutándose o esperando a hacerlo. Un proceso puede crear otros procesos que se conocen como procesos secundarios. El proceso tarda más en terminar y está aislado, lo que significa que no comparte la memoria con ningún otro proceso. Además, los procesos presentan diferentes estados, no tiene por qué estar siempre en ejecución, algunos estados posibles para un proceso son: creado, en ejecución, espera o finalizado.

En cambio, un hilo es un flujo único de ejecución dentro de un proceso, lo cual significa que un proceso puede contener múltiples hilos. También se suele referir a ellos como Threads. En particular, un hilo es una ejecución secuencial de código dentro del contexto de un proceso, esto debido a que los hilos no pueden ejecutarse solos, requieren la supervisión de un proceso. Los hilos al estar contenidos dentro de los procesos siempre tiene un ciclo de vida menor en comparación con el proceso, sin embargo, a diferencia del proceso, los hilos no se aíslan, pueden compartir memoria u objetos.

2.1.2. Memoria y objetos compartidos

Los hilos permiten la ejecución concurrente de varias secuencias de instrucciones asociadas a diferentes funciones dentro de un mismo proceso, compartiendo un mismo espacio de memoria y las mismas estructuras de datos del núcleo. En este contexto, un proceso tradicional es equivalente a un proceso con un único hilo.

La memoria compartida en los algoritmos concurrentes hace referencia al espacio en memoria, el cual es usado por los procesos concurrentes, donde interactúan leyendo y escribiendo en los objetos de la memoria. Dichos objetos se denominan objetos compartidos, y pueden ser manipulados por uno o varios módulos concurrentes. La figura 2.1 representa dos objetos, en donde el Objeto 2 es un objeto compartido por ambos hilos, es decir, ambos hilos pueden acceder a los datos (leer o modificar) del objeto 2 a través de sus métodos.

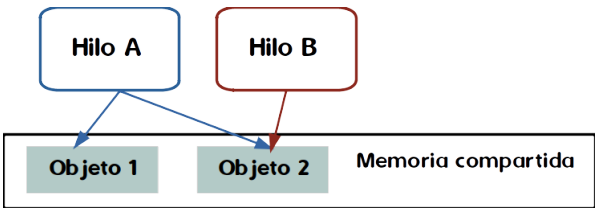


Figura 2.1: Memoria compartida: A y B son dos hilos en el mismo programa, compartiendo los mismos objetos (sin pérdida de generalidad podemos pensar que son objetos en Java).

Tengamos en cuenta que el concepto de memoria compartida es más general que el de objeto compartido, veamos un ejemplo de los problemas que se llegan a presentar cuando dos hilos comparten un objeto.

2.1.3. Memoria y objetos compartidos: Ejemplo

Veamos un ejemplo común de un sistema de memoria compartida. El punto de este ejemplo es mostrar que la programación concurrente es difícil, porque puede tener errores sutiles.

Consideremos un banco que tiene cajeros automáticos, que utilizan una memoria compartida, por lo que todos los cajeros automáticos pueden leer y escribir en los mismos objetos de la cuenta en la memoria. Para no complicar la discusión e ilustrar lo que puede salir mal, tomemos como objeto compartido la variable *saldo*, inicializada en cero; esta variable representa una única cuenta en el banco. Las operaciones que el algoritmo *cajero* posee son *depósito* y *retiro*, los cuales agregan o retiran una unidad en el saldo.

Algorithm 1 cajero

```
   Shared Variables : saldo = 0
 1: procedure DEPOSITAR( )
 2:     temp = saldo
 3:     temp = temp + 1
 4:     saldo = temp
 5: end procedure

 6: procedure RETIRAR( )
 7:     temp = saldo
 8:     temp = temp - 1
 9:     saldo = temp
10: end procedure
```

En este ejemplo simple vamos a considerar transacciones que consistan de un depósito inmediatamente seguido de un retiro. De este modo, cada transacción debería dejar el saldo de la cuenta sin cambios, pues la unidad depositada se retira; sin embargo, al considerar procesos concurrentes y memoria compartida surgen graves problemas incluso con este sencillo ejemplo.

Pensemos que a lo largo de un día un número finito de cajeros automático son prendidos (podemos pensar esto como ejecutar el algoritmo concurrente con número finito de cajeros) y que se ejecutan una transacciones de depósito/retiro en cada cajero. Al final del día, independientemente de cuántos cajeros automáticos estén funcionando o cuántas transacciones procesemos, debemos esperar que el saldo de la cuenta siga siendo 0. Desafortunadamente, veremos que esto no ocurre, para mostrarlo consideremos dos hilos que ejecutan las transacciones de un cajero automático cada uno, además estos hilos se ejecutan concurrentemente. Descubriremos que existe la posibilidad de que el saldo al final del día no sea 0. Si más de una llamada a los métodos de *cajero*() se ejecuta al mismo tiempo, entonces el saldo puede no ser cero al final del día. Veamos que podría estar pasando en estos casos.

Esto se debe a que el objeto compartido sufre modificaciones simultáneamente por ambos hilos. Cuando A y B se ejecutan en paralelo, estas operaciones de bajo nivel se superponen

entre sí, es decir, que son simultáneas o más formalmente *concurrentes*.

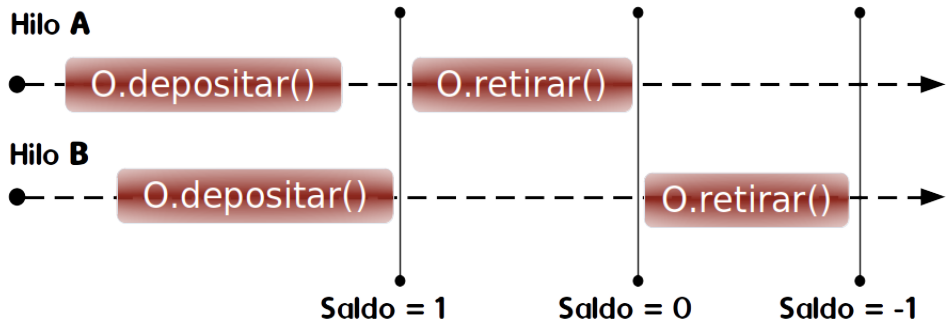


Figura 2.2: Ejemplo de secuencia de operaciones al ejecutar de forma concurrente dos hilos A, B que tienen como objeto compartido el objeto cajero.

En la figura 2.2 el saldo inicial es 0 y las operaciones se ejecutan de la siguiente forma:

- Los hilos A y B leen el saldo inicial al mismo tiempo, ambos leen $saldo = 0$.
- Ambos hilos ejecutan concurrentemente la operación *depositar*, de modo que $saldo == 0$. Sin embargo, al ejecutarse concurrentemente se tiene que $temp_A = 0$ y $temp_B = 0$ (línea 2). En la línea 3 se suma una unidad a la variable *temp*, es decir $temp_A = 1$ y $temp_B = 1$
- Sin pérdida de generalidad supongamos que el hilo A ejecuta la línea 4, y después el hilo B. En cualquier caso, se tendrá que $saldo = 1$. En este punto podemos ver que $saldo = 1$, tal como se marca en la primera división de la figura 2.2.
- Supongamos que en cada hilo se ejecuta una operación *retirar*( ), de forma secuencial, tal como se muestra en la figura 2.2. El saldo final es -1.

Claramente, una unidad se perdió, A y B leen el saldo inicial al mismo tiempo, depositan una unidad, y calculan saldos finales separados, pero al almacenar el nuevo saldo en ambos casos es uno, por lo que el saldo después de los depósitos fue 1. Esto provoca que no se tenga en cuenta el depósito de un hilo, por lo que al retirar se obtiene un saldo incoherente.

2.1.4. Sección crítica

Cuando dos hilos o más leen una línea de código o un proceso, el cual ve modificado por la acción de algún hilo, entonces se entra en una zona de peligro, ya que ambos pueden actualizar el campo al mismo tiempo. Esta sección del código se denomina: *Sección crítica* o *condición de carrera*.

Podemos evitar problemas como el mostrado en el ejemplo anterior si transformamos este proceso en una sección crítica: *Un bloque de código que solo puede ser ejecutado por un hilo a la vez*. Llamamos esta propiedad de los algoritmos *exclusión mutua*. Claramente, al ejecutar una sección crítica no deberíamos tener una superposición de instrucciones, con el fin de

que no se presenten errores como el del ejemplo del cajero. Más adelante podremos dar una definición más precisa de las secciones críticas en términos formales.

2.2. Condiciones de progreso

En un algoritmo concurrente, varios procesos (también llamados hilos) se coordinan al modificar de manera concurrente los datos que se encuentran almacenados en una memoria compartida. Uno de los retos más difíciles de solucionar en este tipo de algoritmos es el de coordinar las modificaciones concurrentes de forma tal que los datos en la memoria compartida siempre estén en un estado consistente, de forma que no ocurran fallas como en el ejemplo de la figura 2.2.

Con el fin de formalizar estas representaciones y la concurrencia veamos las siguientes definiciones, mismas que más adelante serán de gran ayuda.

2.2.1. Eventos e intervalos

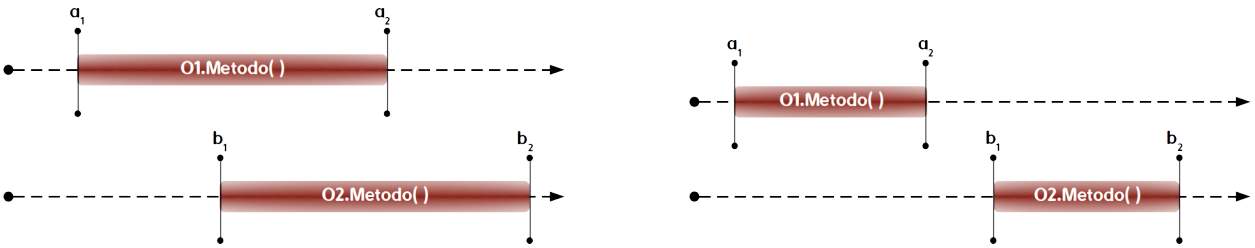
Podemos definir esto más formalmente. Definamos formalmente a un proceso como una máquina de estado y sus transiciones de estado, las cuales denominamos eventos. Regularmente en este trabajo a los eventos se les suele denotar por $\boldsymbol{e}$, en especial en los capítulos 3 y 4, aún más siempre podemos ver las transiciones de estado como

$$\delta(\hat{\boldsymbol{q}}, \boldsymbol{e}) = \hat{\boldsymbol{p}},$$

donde $\hat{\boldsymbol{q}}$ es el estado del proceso *antes* del evento $\boldsymbol{e}$, siendo este un evento cualesquiera y $\hat{\boldsymbol{p}}$ el estado del proceso *después* del evento $\boldsymbol{e}$. Esto tiene una interpretación clara, un proceso concurrente es una máquina de estado y los eventos producen o disparan las transiciones de estado (ejecuciones de instrucciones, las cuales podrían dejar invariante el estado de la máquina).

Los eventos son instantáneos: ocurren en un solo instante de tiempo. Es conveniente exigir que los eventos nunca sean simultáneos, es decir, eventos distintos ocurren en momentos distintos, esto nos permitirá inducir un orden total en el conjunto de eventos de la ejecución de los algoritmos concurrentes.

Un hilo A, produce una secuencia de eventos $a_0, a_1 ... a_n$. Vamos a denotar la $j-$ésima ocurrencia de un evento a_i como a_i^j. Se dice que un evento a precede otro evento b, cuando, a ocurre antes que b, lo denotamos como $a \to b$. La relación de precedencia $\to$ es un orden total de eventos. También podemos dar una noción del tiempo que ocurre entre dos eventos. Sean a_0, a_1 eventos tal que $a_0 \to a_1$, se define el intervalo $I(a_0, a_1)$, como la duración entre a_0, a_1. Se dice que un intervalo $I_A(a_0, a_1)$ precede a otro intervalo $I_B(b_0, b_1)$, cuando $a_1 \to b_0$, lo denotamos como $I_A \to I_B$. Esta relación es un orden parcial en los intervalos. Se definen como intervalos concurrentes a aquellos que no están relacionados, es decir, si $\neg(I_A \to I_B \wedge I_B \to I_A)$.



(a) Intervalos concurrentes (b) Intervalos no concurrentes

Figura 2.3: Representación de intervalos concurrentes. Podemos pensar al intervalo de tiempo entre que se realiza una llamada a un método (evento a_1), y su respuesta (evento a_2) como intervalos. Los métodos tienen efecto en algún instante del intervalo.

Estas definiciones nos permiten describir de manera clara cuando dos operaciones son concurrentes. Sin ningún problema podemos decir que una operación, como la llamada a un método, es un intervalo, donde la invocación al método es un evento, y posteriormente la finalización de dicho método es el evento que marca el fin de la operación.

2.2.2. Condiciones de progreso bloqueantes

La memoria compartida de los procesos concurrentes provoca comportamientos inesperados, como lo visto en el ejemplo del cajero automático. Este comportamiento contrasta con los procesos secuenciales (procesos de un solo hilo) de los algoritmos secuenciales, en donde no hay necesidad de tener objetos compartidos ni secciones críticas. En cambio, los algoritmos concurrentes presentan dos aspectos fundamentes de la concurrencia:

- Condición de progreso (*safety*): La cual garantiza que eventualmente algo bueno sucederá, también llamada la seguridad de los algoritmos.
- Correctitud (*liveness)*: La cual garantiza que nada malo sucederá, también llamada viveza de los algoritmos

La correctitud la detallaremos más adelante. Por otro lado, existen diferentes especificaciones o condiciones de progreso, las cuales pueden ser bloqueantes o no bloqueantes. En las condiciones de progreso bloqueantes se puede llegar a producir un retraso inesperado en cualquier proceso, esto puede retrasar la ejecución de otras instrucciones o el completo impedimento de que los demás procesos continúen.

Los métodos que tienen una condición de progreso bloqueante suelen estar basados en candados (en lenguaje Java son los *Lock*), cuyo funcionamiento se basa en hacer cumplir la exclusión mutua; más precisamente, los candados se usan para indicar que un hilo está ejecutando la sección crítica y estos *bloquean* la entrada a los demás hilos, de modo que se satisface la exclusión mutua. De tal forma que un candado es usado para aislar una sección crítica, de modo que cualquier hilo puede adquirir y liberar la sección crítica siempre y cuando otro hilo no se encuentre en ejecución. En lenguaje Java, cuando dos hilos ejecutan una sección crítica, uno la adquiere (y la bloquea) mientras que el otro hilo espera la respuesta de la solicitud para adquirir acceso a la sección crítica. Sin embargo, cabe mencionar que

ante el fallo o suspensión del primer hilo, se producirá un bloqueo, ya que el segundo hilo podría nunca recibir respuesta del candado (Java posee infraestructura para capturar estas excepciones), pero cabe tenerlo en cuenta. Formalicemos un poco la exclusión mutua en los algoritmos.

Sea $SC_{\boldsymbol{A}}$ el intervalo durante el cual un hilo $\boldsymbol{A}$ ejecuta una sección crítica SC. Formalmente, para que un algoritmo satisfaga la exclusión mutua debe cumplir que las secciones críticas de diferentes hilos no se superponen, es decir, para los hilos $\boldsymbol{A}$ y $\boldsymbol{B}$, consideremos los intervalos $SC_{\boldsymbol{A}}$ y $SC_{\boldsymbol{B}}$ en los cuales se ejecutan las secciones críticas, entonces sucede $SC_{\boldsymbol{A}} \to SC_{\boldsymbol{B}}$ o $SC_{\boldsymbol{B}} \to SC_{\boldsymbol{A}}$.

Se puede mencionar dos condiciones de progreso bloqueantes, las cuales recurren al uso de candados: Deadlock-Free: Si algún hilo $\boldsymbol{A}$ intenta adquirir el candado, entonces existe algún hilo $\boldsymbol{B}$, puede ser el mismo hilo $\boldsymbol{A}$, que tendrá éxito en adquirir el candado (entrar a la sección crítica). *Starvation-free*: Todo hilo que intenta adquirir el candado tiene éxito en algún momento. También se puede decir que toda adquisición de candado eventualmente lo libera.

Nótese que la condición *Starvation-free* implica la condición Deadlock-Free. Además, la condición Deadlock-Free es importante, pues nos asegura que el sistema nunca se va a *bloquear*, ya que aunque los hilos individuales pueden atascarse, esperando la adquisición de un candado, para siempre (lo que se denomina inanición o *starvation*), pero entonces deben existir algunos hilos que sigan ejecutando la sección crítica.

2.2.3. Condiciones de progreso no-bloqueantes

Como hemos visto, las condiciones de progreso bloqueantes recurren al uso de candados para satisfacer la exclusión mutua, sin embargo, es posible definir otro tipo de condiciones de progreso, sin la necesidad de recurrir a los candados, estas son conocidas como condiciones de progreso no-bloqueantes, en las cuales el retraso inesperado de un proceso no retrasa a los demás procesos.

Dos condiciones no-bloqueantes muy importantes son las siguientes. La condición *wait-free*: Un método u operación termina de ejecutarse en número finito de pasos, se dice que un algoritmo es wait-free si todos los métodos u operaciones que lo componen lo son. La condición *non-blocking* o *lock-free*: Un método u operación es non-blocking si se garantiza que en una ejecución infinita de esta, existen infinitas operaciones que se completan en un número finito de pasos.

En otras palabras, la condición *wait-free* garantiza que todo proceso, siempre que no haya fallado súbitamente, eventualmente progresa. Cualquier método *wait-free* es también *non-blocking* pero no al revés. Los métodos *wait-free* pueden llegar a ser ineficientes, es por ello que una propiedad menos restrictiva como *non-blocking* es muy útil. Además, se podría decir que estas condiciones de progreso son la versión de las condiciones *Deadlock-Free* y *Starvation-free* que no recurren a la definición de candados. Esto es sumamente útil, pues

en el siguiente modelo de computación concurrente que se presentará se consideran posibles fallas en los procesos. Aún más, los algoritmos que mostramos como principal contribución no recurren en sí al uso de candados; sin embargo, más adelante se discute esto.

2.3. Modelo de computación

Vamos a considerar el modelo estándar de sistemas concurrentes, el cual ha sido usado en [9, 7], con n procesos asíncronos, $p_1, ..., p_n$ los cuales, pueden llegar a fallar durante una ejecución; formalmente un proceso falla cuando deja de dar pasos. El *índice* del proceso p_i es i. Los procesos se comunican entre sí invocando operaciones atómicas en objetos base compartidos. Un *objeto base* puede proporcionar operaciones atómicas de lectura/escritura, a partir de ahora, dicho objeto se denomina *registro*, u operaciones atómicas más potentes de lectura, modificación y escritura.

Un *objeto concurrente*, T, se define como una máquina de estado que consta de: un conjunto de estados, un conjunto finito de operaciones y un conjunto de transiciones entre estados. Esta especificación no necesariamente tiene que ser secuencial, es decir:

- Un estado puede tener operaciones pendientes
- Las transiciones de estado pueden involucrar varias invocaciones

La noción de objeto concurrente se formaliza en las siguientes subsecciones.

Una implementación de un objeto concurrente T es un algoritmo distribuido A que consta de máquinas de estados locales $A_1, ..., A_n$. Cada máquina local A_i especifica qué operaciones en los objetos base, p_i ejecuta para devolver una respuesta cuando invoca una operación de alto nivel de T. Cada una de estas invocaciones de operación en los objetos base es un paso.

Una *ejecución* de A es una secuencia (posiblemente infinita) de pasos, es decir, ejecuciones de operaciones de objetos base, más invocaciones y respuestas a operaciones del objeto concurrente T, con las siguientes propiedades:

1. Cada proceso es secuencial. Primero invoca una operación, y solo cuando tiene su respuesta correspondiente, puede invocar otra operación, es decir, las ejecuciones están bien formadas.
2. Para cualquier invocación a una operación op, denotada $inv(op)$, para un proceso p_i, los pasos de p_i entre esa invocación y su respuesta correspondiente (si la hay), denotada $res(op)$, son pasos especificados por A cuando p_i invoca op.

Se dice que una operación op es completa si su invocación y su respuesta aparecen en la ejecución. Una operación está pendiente si en la ejecución solo aparece su invocación. Un proceso es correcto en una ejecución, si toma infinitos pasos. Por simplicidad, y sin pérdida de generalidad, identificamos la invocación de una operación con su primer paso y su respuesta con su último paso.

2.4. Linealizabilidad

El comportamiento de los objetos concurrentes se describe por medio de propiedades de seguridad y viveza, a menudo denominadas condiciones de progreso [14]. La verificación de la linealizabilidad de algoritmos concurrentes se puede analizar en un modelo de cómputo concurrente. En este modelo se pueden diseñar algoritmos concurrentes y analizar su corrección a partir de las condiciones de progreso y de corrección.

Diversas nociones de corrección para objetos concurrentes han sido propuestas; sin embargo, la mayoría se basan en alguna noción de equivalencia con el comportamiento secuencial. En [14] se presentan varias condiciones de corrección como la consistencia inactiva (Quiescent consistency), la consistencia secuencial (sequential consistency) o la linealizabilidad. Una propiedad importante de la linealizabilidad es la de ser modular (también llamada local) [15]; esta misma propiedad hace de la linealizabilidad una condición de corrección más fuerte que las primeras dos.

2.4.1. Condición de correctitud: Linealizabilidad

La linealizabilidad es la noción estándar utilizada para definir una implementación concurrente correcta de un objeto definido por una especificación secuencial. Intuitivamente, una ejecución es linealizable si las operaciones pueden ordenarse secuencialmente, sin reordenar operaciones que no se superpongan, de modo que sus respuestas satisfagan la especificación del objeto implementado.

Se define una especificación secuencial de un objeto concurrente T mediante una máquina de estados especificada a través de una función de transición δ. Dado un estado q y una invocación $inv(op)$, $\delta(q, inv(op))$ devuelve la tupla $(q', res(op))$ (o un conjunto de tuplas si la máquina no es determinista) indicando que la máquina pasa al estado q' y la respuesta a op es $res(op)$. En nuestras especificaciones, $res(op)$ se escribe como un salto de tupla $\langle op : r \rangle$, donde r es el valor de salida de la operación. Las secuencias de tuplas invocación-respuesta, $\langle inv(op) : res(op) \rangle$, producidas por la máquina de estado, se denominan ejecuciones secuenciales.

Para formalizar la linealizabilidad debemos definir un orden parcial $<_\alpha$ en las operaciones completadas de una ejecución α, por lo que denotamos $op <_\alpha op'$ si y solo si $res(op) \rightarrow inv(op')$ en α. Decimos que dos operaciones son concurrentes si son incomparables por $<_\alpha$, usamos $op||_\alpha op'$ para denotar que dos operaciones son concurrentes.

Definición 2.4.1 *Linealizabilidad: Sea A una implementación de un objeto concurrente T. Una ejecución α de A es linealizable si existe una ejecución secuencial S de T tal que*

- *S contiene todas las operaciones completadas de α y puede contener algunas operaciones pendientes. Las entradas y salidas de invocaciones y respuestas en S concuerdan con las entradas y salidas en α.*

- *Para cualesquiera operaciones completadas, op y op′ en α, si sucede que op $<_\alpha$ op′, entonces op aparece antes que op′ en S.*

Decimos que A es linealizable si cada una de sus ejecuciones es linealizable.

Decir que un objeto concurrente es correcto, lleva a tratar de encontrar una forma de extender el orden parcial a un orden total. La condición de corrección de linealizabilidad busca justamente esto. Como se ha mencionado anteriormente, dada una ejecución α, la relación $<_\alpha$ es un orden parcial, cuando la ejecución es linealizable la relación $<_\alpha$ se convierte en un orden total.

Podemos explicar el concepto de linealizabilidad por medio de un ejemplo. Si consideramos, dos hilos A y B, cada vez que se ejecute el programa obtendremos una secuencia de llamadas y respuestas de operaciones. Denotemos por simplicidad la i-ésima llamadas en el hilo A (o B) como $IA - i$ (la cual hemos llamado anteriormente $inv(op)$), y la respuesta como $RA - i$ ($res(op)$). Claramente, se pueden dar superposiciones entre las llamadas a métodos, llamadas hechas por el hilo A o B. Pero cada evento (es decir, invocaciones y respuestas) tiene un orden en tiempo real. Entonces, las invocaciones y respuestas de todos los métodos llamados por A y B se pueden asignar a un orden secuencial, el orden puede como se muestra a continuación.

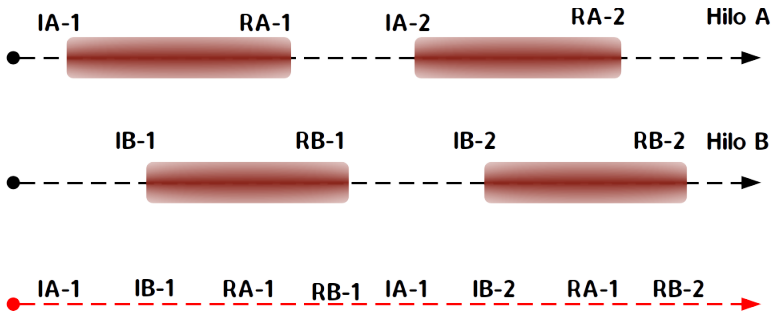


Figura 2.4: Secuencia de eventos en una ejecución en dos hilos A, B

Una ordenación en tiempo real de los eventos se muestra en la flecha punteada roja. El orden de los eventos mostrado es:

$$\text{IA} - 1,\ \text{IB} - 1,\ \text{RA} - 1,\ \text{RB} - 1,\ \text{IA} - 2,\ \text{IB} - 2,\ \text{RA} - 2,\ \text{RB} - 2,$$

Esta secuencia de eventos representa una ejecución α, en [14] se define de forma distinta y se le llama historial; sin embargo, son equivalentes. Los dos puntos de la definición de linealizabilidad informalmente significan que podemos dar una reordenación de los eventos siempre y cuando se respete el orden para las operaciones que cumplen $op <_\alpha op'$. Esto significa que, si el evento respuesta de una operación ocurrió antes que el evento de llamada de otra operación, entonces en la reordenación se debe preservar este orden.

Por ejemplo, las reordenaciones válidas (las cuales formalmente son las ejecuciones secuenciales) para la ejecución mostrada en la figura 4, son:

1. $\text{IA} - 1,\ \text{RA} - 1,\ \text{IB} - 1,\ \text{RB} - 1,\ \text{IB} - 2,\ \text{RB} - 2,\ \text{IA} - 2,\ \text{RA} - 2$

2. $\text{IB}-1,\ \text{RB}-1,\ \text{IA}-1,\ \text{RA}-1,\ \text{IB}-2,\ \text{RB}-2,\ \text{IA}-2,\ \text{RA}-2$

3. $\text{IB}-1,\ \text{RB}-1,\ \text{IA}-1,\ \text{RA}-1,\ \text{IA}-2,\ \text{RA}-2,\ \text{IB}-2,\ \text{RB}-2$

4. $\text{IA}-1,\ \text{RA}-1,\ \text{IB}-1,\ \text{RB}-1,\ \text{IA}-2,\ \text{RA}-2,\ \text{IB}-2,\ \text{RB}-2$

Observemos que solo estamos permutando el orden de los eventos de llamada y respuesta de los métodos que son concurrentes. Debemos remarcar que esto es natural, ya que en métodos concurrentes no sabemos qué método se ejecuta primero, pues la duración de su ejecución es paralela.

Sin embargo, de estas ejecuciones secuenciales, ¿cómo podemos confirmar si nuestra ejecución es correcta? Aquí nos referimos a la ejecución α. La definición formal nos dice que, si al menos una ejecución secuencial es correcta, entonces α es linealizable. Podemos ver esto con un ejemplo sencillo. Considerando el ejemplo anterior, pensemos que las operaciones se refieren a invocaciones, un método que primero es escritura (denotada como evento $Writte : W$) y finaliza con lectura (evento $Read : R$), como que muestra en la figura 5, a continuación.

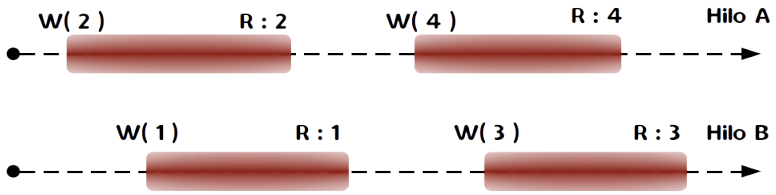


Figura 2.5: Ejecución de un método de lectura (W), escritura (R)

De esta ejecución basta con tomar la ejecución secuencial 2), donde la secuencia es:

$$Writte(1),\ Read(1),\ Writte(2),\ Read(2),\ Writte(3),\ Read(3),\ Writte(4),\ Read(4).$$

La secuencia es válida, ya que sigue la especificación secuencial de un registro, es decir, los *read* obtienen el valor del *write* más reciente.

2.4.2. Condición de correctitud: Linealizabilidad por conjuntos

Hemos visto que la condición de correctitud de linealizabilidad se basa en poder reordenar de manera que la ejecución simule ser una ejecución secuencial, es decir, imitando lo que haría un programa secuencial o de un proceso. Sin embargo, se han propuesto condiciones de correctitud más relajadas. Entre ellas, la linealizabilidad por conjuntos o por intervalos.

La linealización de conjuntos nos permite linealizar varias operaciones en el mismo punto, es decir, todas estas operaciones se ejecutan simultáneamente, mientras que la linealización de intervalos permite linealizar operaciones simultáneamente con varias operaciones no simultáneas.

Se sabe que la linealización de conjuntos tiene estrictamente más poder de expresividad que la linealización, y la linealización de intervalos es estrictamente más poderosa que la

linealización de conjuntos [9]. Además, la linealizabilidad, y tanto la linealización de conjuntos como la linealización de intervalos son propiedades composicionales [6].

Una especificación conjunto-concurrente de un objeto concurrente difiere de una ejecución secuencial en que δ recibe como entrada el estado actual q de la máquina y un conjunto $Inv = \{inv(op_1), ..., inv(op_t)\}$ de invocaciones de operaciones, y $\delta(q, Inv)$ retorna $\delta(q', Res)$, donde q' es el siguiente estado y $Res = \{res(op_1), ..., res(op_t)\}$ son las respuestas de las invocaciones en Inv. Los conjuntos Inv y Res son llamados clases de concurrencia. Debemos mencionar que una especificación de conjunto-concurrente donde las clases de concurrencia tenga un único elemento corresponde a una especificación secuencial.

Definición 2.4.2 *Linealizabilidad por conjuntos: Sea A una implementación de un objeto concurrente T. Una ejecución α de A es linealizable por conjuntos si existe una ejecución de conjunto-concurrente S de T tal que*

- *S contiene todas las operaciones completadas de α y puede contener algunas operaciones pendientes. Las entradas y salidas de invocaciones y respuestas en S concuerdan con las entradas y salidas en α.*
- *Para cualesquiera operaciones completadas, op y op' en α, si sucede que $op <_\alpha op'$, entonces op aparece antes que op' en S.*

Decimos que A es linealizable por conjuntos si cada una de sus ejecuciones es linealizable por conjuntos.

Debemos resaltar que estos conjuntos son de hecho las clases de equivalencia de la relación $||_\alpha$, es decir, son las clases de equivalencia de las operaciones concurrentes. La interpretación es la misma que en la linealizabilidad, pero en este caso estamos considerando un conjunto de llamadas y un conjunto de respuestas.

Capítulo 3

Colas con semánticas relajadas: Multiplicidad

Dado que para algunas estructuras de datos concurrentes todo algoritmo linealizable tiene limitaciones inherentes, se han buscado algoritmos que implementen versiones relajadas de esas estructuras de datos. Es decir, en lugar de que el algoritmo siempre muestre un comportamiento lineal (que sea linealizable), se permite que de vez en cuando proporcione resultados que no son exactamente los que daría un algoritmo linealizable. Dicho de otra forma, estos algoritmos proporcionan soluciones "aproximadas".

3.1. Colas concurrentes por conjuntos con multiplicidad

Se ha propuesto versiones "relajadas" de pilas y colas, las cuales cuentas con una propiedad conocida como multiplicidad. De manera intuitiva, una cola con multiplicidad permite que un conjunto de operaciones $\boldsymbol{Dequeue_i}$, ya sean dos o más, devuelvan un mismo valor que fue encolado anteriormente en la estructura, pero solo si las operaciones $\boldsymbol{Dequeue_i}$ ocurrieron concurrentemente.

La estructura de datos, llamada cola relajada o cola con multiplicidad, está definida con base en esta idea. Los conjuntos de operaciones concurrentes $\boldsymbol{Dequeue_i}$ actúan sobre la estructura provocando una transición de estados correspondiente a la transición que se espera de una cola, es decir, se remueve el primer elemento. Mientras que las operaciones ***Enqueue*** insertan un elemento dado del mismo como que lo haría una cola. En otras palabras, los elementos se devuelven en orden FIFO y no se pierde ningún elemento en cola. Esto se expresa en la siguiente definición.

Definición 3.1.1 *El universo de elementos que se pueden encolar (**Enqueue**) es $N = \{1, 2, ...\}$, y el conjunto de estados Q es el conjunto infinito de cadenas N^*. El estado inicial es la cadena vacía, indicada como $\boldsymbol{\epsilon}$. En el estado $\hat{\boldsymbol{q}}$, el primer elemento en $\hat{\boldsymbol{q}}$ representa la cabeza de la cola, que podría estar vacía si $\hat{\boldsymbol{q}}$ es la cadena vacía. Las transiciones son las siguientes:*

- *Para $\hat{\boldsymbol{q}} \in Q$:*

$$\delta(\hat{\boldsymbol{q}}, \boldsymbol{Enqueue}(\boldsymbol{a})) = (\hat{\boldsymbol{q}} * \boldsymbol{a}, \langle \boldsymbol{Enqueue}(\boldsymbol{a}) : \boldsymbol{True} \rangle) \tag{3.1.0.1}$$

- *Para $\boldsymbol{a} * \hat{\boldsymbol{q}} \in Q$, donde $\boldsymbol{a} \in \boldsymbol{N}$ y t procesos:*

$$\begin{aligned} &\delta(\boldsymbol{a} * \hat{\boldsymbol{q}}, \{\boldsymbol{Dequeue_1}(), ..., \boldsymbol{Dequeue_t}()\}) \\ &\qquad = (\hat{\boldsymbol{q}}, \{\langle \boldsymbol{Dequeue_1}() : \boldsymbol{a} \rangle, ..., \langle \boldsymbol{Dequeue_t}() : \boldsymbol{a} \rangle\}) \end{aligned} \tag{3.1.0.2}$$

- *Para una cola vacía $\boldsymbol{\epsilon} \in Q$:*

$$\delta(\boldsymbol{\epsilon}, \boldsymbol{Dequeue}(\,)) = (\boldsymbol{\epsilon}, \langle \boldsymbol{Dequeue}(\,) : \boldsymbol{\epsilon} \rangle) \tag{3.1.0.3}$$

El siguiente lema, tomado de [9], muestra que cualquier algoritmo que implemente la cola concurrente por conjuntos mantiene el comportamiento de una cola secuencial en ciertos casos. De hecho, la única razón por la que la implementación no proporciona capacidad de linealización se debe únicamente a las operaciones *Dequeue* que son concurrentes.

Lema 3.1.1 *Sea A cualquier implementación linealizable por conjuntos de una cola concurrente por conjuntos con multiplicidad. Entonces se cumple:*

- *Todas las ejecuciones secuenciales de A son ejecuciones de la cola secuencial*
- *Todas las ejecuciones sin operaciones Dequeue concurrentes son linealizables con respecto a la cola secuencial*
- *Todas las ejecuciones con operaciones Dequeue que devuelven valores distintos son linealizables con respecto a la cola secuencial.*
- *Si las operaciones Dequeue devuelven el mismo valor en una ejecución, entonces son concurrentes*

Existen más versiones de pilas y colas con semánticas relajadas. Sin embargo, para este proyecto tiene como objetivo desarrollar algoritmos que implementen pilas y colas para usar las versiones relajadas (con multiplicidad) y estudiar su eficiencia. Debemos remarcar que la propiedad de multiplicidad es un efecto de la condición de corrección de linealizabilidad por conjuntos, ya que al permitir que ocurran operaciones concurrentes (*Pop*, en el caso de pilas, o *Dequeue* en el caso de colas), se obtienen respuestas repetidas.

3.2. Cola concurrente: versión linealizable

El algoritmo para una cola concurrente, la cual es linealizable, se muestra en el pseudocódigo 2. Nótese que no posee multiplicidad, y, por lo tanto, las transiciones de cola son las usuales. El algoritmo está inspirado en el código *LockFreeQueue* de [14], el cual es un algoritmo linealizable de colas *Lock Free*.

Algorithm 2 Lineal-Queue

```
   Shared Variables : Head, Tail
 1: procedure ENQUEUE(x)
 2:     while True do
 3:         t = Tail.get()
 4:         n = t.next.get()
 5:         if n == ϵ then
 6:             if t.next.CompareAndSet(n, x) then
 7:                 Tail.CompareAndSet(t, n)
 8:                 return
 9:             end if
10:         else
11:             Tail.CompareAndSet(t, n)
12:         end if
13:     end while
14: end procedure

15: procedure DEQUEUE
16:     while True do
17:         h = Head.get()
18:         t = Tail.get()
19:         n = h.next.get()
20:         if t == h then
21:             if n == ϵ then
22:                 return ϵ
23:             else
24:                 Tail.CompareAndSet(t, t.next.get())
25:             end if
26:         else
27:             if Head.CompareAndSet(h, n) then
28:                 return n.value
29:             end if
30:         end if
31:     end while
32: end procedure
```

A partir del algoritmo 2 de la cola concurrente, se puede conseguir, bajo una ligera modificación, un algoritmo concurrente por conjuntos de la cola con multiplicidad. Para

comprender el algoritmo concurrente por conjuntos de la pila con multiplicidad, primero veamos la estructura de los nodos (se usan los mismos nodos en el caso linealizable y linealizable por conjuntos).

3.3. Estructura de los nodos

Con el fin de proponer un algoritmo que sea una implementación linealizable por conjuntos de la cola con multiplicidad, primero se describe la estructura de los nodos que tendrá la cola. Es usual que las estructuras de datos, tipo cola o pila, estén formadas por nodos, los cuales contienen elementos de tipos primitivos. Sin embargo, debido tanto a la concurrencia como a la multiplicidad, es necesario definir una estructura más compleja en los nodos.

Para definir cierto tipo de operaciones es útil recurrir a los objetos de tipo ***AtomicReference***, los cuales se proveen en Java. Las variables de tipo ***AtomicReference*** proporciona una referencia (a algún objeto) que se puede leer y escribir atómicamente. Es decir, que múltiples hilos que intentan modificar la misma referencia almacenada en la variable de tipo ***AtomicReference*** lo hacen de manera atómica, de modo que el estado de la referencia será consistente a las modificaciones. Supongamos por el momento que las referencias de una instancia dada de ***AtomicReference*** son objetos $\boldsymbol{nodo_i}$ (más adelante se definirá la estructura de los nodos), los métodos principales de cualquier instancia son:

- ***get*( )**: Este método permite que la referencia almacenada en la variable de tipo ***AtomicReference*** se pueda leer atómicamente.
- ***CompareAndSet*($\boldsymbol{nodo_1}$, $\boldsymbol{nodo_2}$)**: Este método compara la referencia almacenada en la instancia de ***AtomicReference***, llamémosle $\boldsymbol{nodo_0}$, contra una referencia esperada $\boldsymbol{nodo_1}$, y si las dos referencias son iguales (es decir $\boldsymbol{nodo_0} == \boldsymbol{nodo_1}$) entonces el método inserta la nueva referencia $\boldsymbol{nodo_2}$ en la instancia de ***AtomicReference***. Cuando el método ***CompareAndSet*** cambia la referencia en la instancia de ***AtomicReference*** este retorna ***True*** (regularmente nos referimos a este caso diciendo que el método ***CompareAndSet*** fue exitoso). En caso contrario, la referencia no sufre de ningún cambio y el método retorna ***False*** (nos referiremos a este caso diciendo que el método fue fallido o no exitoso).

Los nodos están definidos como una estructura por sí misma, es decir, un objeto. Cada nodo contiene dos atributos, uno llamado ***value***, es cual es el valor de tipo primitivo almacenado en el nodo. Supondremos que estos son enteros por simplicidad y concordancia con la definición 3.1.1. Por otro lado, el atributo ***next*** es un objeto de tipo ***AtomicReference*** $< \boldsymbol{Nodo} >$. Este último atributo funciona como una *referencia* al nodo siguiente, sin embargo, cabe resaltar que esta referencia estará almacenada es un objeto por sí mismo, y para acceder a la referencia, para leerla o modificarla atómicamente, es necesario usar los métodos ***get*** o ***CompareAndSet*** de ***AtomicReference***.

Algorithm 3 Nodo(x)

Nodo
1: ***value*** : Entero de valor ***x***
2: ***next*** : ***AtomicReference*** $<$ ***Nodo*** $>$

La estructura de cola se representará como una lista de nodos; véase figura 3.1. En estas listas cada nodo posee una referencia, ***next.get( )***, la cual hace referencia al siguiente nodo de la lista, mientras que el valor de cada nodo es ***value***. La referencia del último nodo siempre apunta a un ***null***. Sin embargo, debido a la concurrencia, las estructuras de datos deben de ser definidas de una forma distinta en cada caso, en la siguiente sección se detallará el algoritmo y la estructura de la cola con la que se trabajará. En esta sección hemos detallado la estructura de los nodos y una breve descripción de las colas, esta descripción es necesaria, pues comúnmente los nodos son representados como elementos primitivos y las colas como listas.

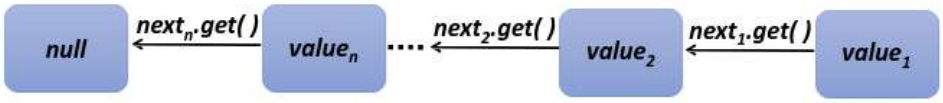


Figura 3.1: Representación de una lista de nodos que es implementada para las colas.

3.4. Algoritmo: Set-Queue

En esta sección se presenta el algoritmo 4: *Set-Queue*, el cual es una implementación de la cola con multiplicidad; más adelante se discute esto formalmente. El algoritmo está inspirado en el algoritmo *LockFreeQueue* de [14], el cual es un algoritmo linealizable de colas *Lock-Free*. Como se discutió anteriormente, las colas se pueden representar como una lista de nodos conectados por las referencias, debido a la naturaleza de la concurrencia es necesario el uso de un nodo que no participa formalmente en el estado de la cola, es decir, el nodo no pertenece a la cola. Definimos a este nodo como, *nodo centinela*, el cual será siempre el primer nodo en la lista de nodos. El algoritmo en sí consta de dos objetos de tipo ***AtomicReference*** $<$ ***Nodo*** $>$:

- ***Head*** : Contiene la referencia al primer nodo de la lista (***Head.get( )*** es la referencia que apunta al primer nodo), es decir, el nodo centinela. Dado que el nodo centinela no forma parte de la cola, su valor no tiene relevancia.
- ***Tail*** : Contiene la referencia al último nodo de la lista, es decir, ***Tail.get( )*** es la referencia que apunta al último nodo.

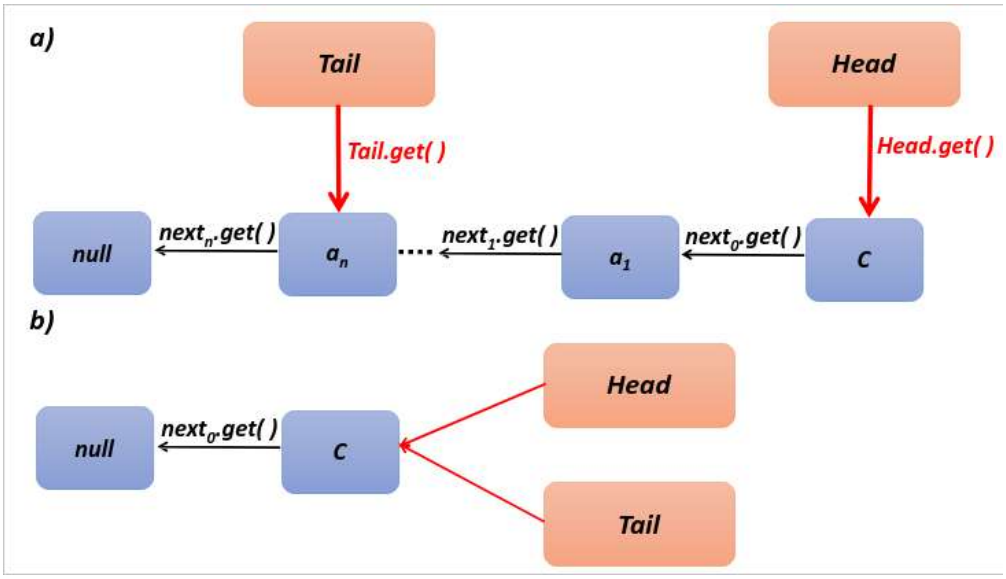


Figura 3.2: *Panel a)* Representación de una estructura de colas, junto con los campos $\boldsymbol{Head}$, $\boldsymbol{Tail}$ y el nodo centinela. *Panel b)* Representación de una estructura de colas sin elementos en la cola, es decir, la cola vacía $\boldsymbol{\epsilon}$.

En la figura 3.2 *panel a)* se representa una cola del algoritmo Set-Queue 4, en donde el campo $\boldsymbol{Head}$ apunta al nodo centinela de valor $\boldsymbol{C}$, sin embargo, el valor de este nodo es irrelevante para la cola. La cola se conforma por los $\boldsymbol{n}$ nodos que siguen, de modo que el primer nodo de la cola es aquel al cual el nodo centinela apunta por medio de su referencia ($\boldsymbol{next_0.get(\,)}$ es la regencia al primer nodo de la cola). Se puede decir que la cola es $\boldsymbol{a_n a_{n-1} ... a_2 a_1}$. En cambio, el campo $\boldsymbol{Tail}$ apunta al último nodo de valor $\boldsymbol{a_n}$, esta referencia atómica funciona como auxiliar para encolar nodos. Cabe mencionar que durante la ejecución de las operaciones y debido a la concurrencia, el campo $\boldsymbol{Tail}$ no siempre apunta al último nodo; sin embargo, el algoritmo está construido para que esto no sea un problema.

Otro aspecto importante es cómo se comporta la cola vacía. Si bien una cola vacía puede ser representada por la ausencia de nodos, de modo que $\boldsymbol{Head}$ y $\boldsymbol{Tail}$ apunten a $\boldsymbol{null}$, la concurrencia hace necesaria otra representación, ya que en caso de que $\boldsymbol{Head}$ y $\boldsymbol{Tail}$ puedan apuntar a $\boldsymbol{null}$, pueden producirse transiciones de estados incongruentes con las transiciones de cola, como por ejemplo estados donde $\boldsymbol{Head}$ apunta a una lista de nodos y $\boldsymbol{Tail}$ apunte a $\boldsymbol{null}$. Una solución es que en el estado de cola vacía ambos campo $\boldsymbol{Head}$ y $\boldsymbol{Tail}$ apunten al nodo centinela, dado que este nodo no es tomado en cuenta en la cola, esto es equivalente a que no haya ningún nodo. En figura 3.2 *panel b)* se representa una cola vacía $\boldsymbol{\epsilon}$ con $\boldsymbol{Head}$ y $\boldsymbol{Tail}$ apuntando al nodo centinela.

Algorithm 4 Set-Queue

```
    Shared Variables : Head, Tail
 1: procedure ENQUEUE(x)
 2:     while True do
 3:         t = Tail.get()
 4:         n = t.next.get()
 5:         if n == ϵ then
 6:             if t.next.CompareAndSet(n, x) then
 7:                 Tail.CompareAndSet(t, n)
 8:                 return
 9:             end if
10:         else
11:             Tail.CompareAndSet(t, n)
12:         end if
13:     end while
14: end procedure

15: procedure DEQUEUE
16:     while True do
17:         h = Head.get()
18:         t = Tail.get()
19:         n = h.next.get()
20:         if t == h then
21:             if n == ϵ then
22:                 return ϵ
23:             else
24:                 Tail.CompareAndSet(t, t.next.get())
25:             end if
26:         else
27:             Head.CompareAndSet(h, n)
28:             return n.value
29:         end if
30:     end while
31: end procedure
```

Se ha discutido la estructura de la cola que implementa el algoritmo Set-Queue, las operaciones o métodos con lo que el algoritmo consta son dos: ***Enqueue*** y ***Dequeue***, los cuales tienen el siguiente funcionamiento:

- ***Enqueue(x)***: Añade un nodo ***x*** en orden FIFO. La operación primero inicia un ciclo ***While*** (sin condiciones de paro). En cada iteración primero se extrae el nodo ***t*** en la referencia ***Tail*** junto con el nodo siguiente. Observemos que, como la referencia atómica es ***t.next***, es necesario usar el método ***get( )*** para obtener el nodo). Después evalúa en la línea 5 si el nodo siguiente es vacío $\boldsymbol{\epsilon}$, si no lo es significa que la referencia ***Tail*** debe avanzar hacia adelante, con un ***CompareAndSet***, para apuntar al

último nodo (pasando a la línea 11). En caso de que $\boldsymbol{n} == \boldsymbol{\epsilon}$ la operación ejecuta un $\boldsymbol{CompareAndSet(n, x)}$ en la línea 6, si es exitoso entonces la referencia cambio de $\boldsymbol{\epsilon}$ a $\boldsymbol{x}$, de modo que el nodo $\boldsymbol{x}$ se introdujo al final, en caso contrario vuelve a iterar para otro intento. Por último, si $\boldsymbol{CompareAndSet(n, x)}$ en la línea 6 fue exitoso, trata de avanzar $\boldsymbol{Tail}$ al nuevo nodo final (puede ser o no exitoso) y finalmente retorna $\boldsymbol{True}$ en la línea 8.

- $\boldsymbol{Dequeue}$: Remueve el primer nodo de la cola, en orden FIFO, y retorna su valor correspondiente. La operación primero inicia un ciclo $\boldsymbol{While}$ (sin condiciones de paro). En cada iteración primero se extrae el nodo $\boldsymbol{h}$ en la referencia $\boldsymbol{Head}$ (observemos que es el nodo centinela), el nodo $\boldsymbol{t}$ en la referencia $\boldsymbol{Tail}$ y el nodo siguiente $\boldsymbol{n}$ (observemos que es en realidad el primer nodo). Cuando los nodos $\boldsymbol{t}$ y $\boldsymbol{h}$ son iguales (línea 20) se evalúa si $\boldsymbol{n}$ es $\boldsymbol{\epsilon}$ (línea 21). En caso afirmativo significa que la cola está vacía, pues el único nodo en la estructura es el nodo centinela, por lo que retorna $\boldsymbol{\epsilon}$. En caso contrario, la cola no está vacía, sino que la referencia $\boldsymbol{Tail}$ está desactualizada y no apunta al último nodo, por lo que pasa a la línea 24 para que $\boldsymbol{Tail}$ avance al siguiente nodo y después vuelve a intentar otra iteración. Cuando $\boldsymbol{t}$ y $\boldsymbol{h}$ no son iguales (línea 20), se ejecuta $\boldsymbol{Head.CompareAndSet(h, n)}$ en la línea 27, independientemente del resultado se retorna el valor del primero nodo de la cola (esto permite la multiplicidad).

Si bien la descripción del algoritmo podría dar una noción del correcto comportamiento del algoritmo, es necesario enunciar y demostrar formalmente las propiedades del mismo. El siguiente teorema describe por completo al algoritmo 4; teoremas de este estilo se han sido probados para algunos algoritmos concurrentes [8, 7, 9, 1].

Teorema 3.4.1 *El algoritmo Set-Queue es una implementación linealizable por conjuntos de la cola con multiplicidad y es non-blocking.*

En la sección 3.4.5 se muestra a detalle la dinámica del algoritmo Set-Queue en algunos escenarios particulares, mientras que las secciones restantes demuestran el teorema 3.4.1. La demostración esta divida de la siguiente forma, en la sección 3.4.1 se demuestra que el algoritmo 4 satisface *non-blocking*, en la sección 3.4.2 se describe la forma en la que se construye la linealización por conjuntos (también en esta sección se dan los *puntos de linealización*), en la sección 3.4.3 se muestra que usando los puntos de linealización de la sección 3.4.2 se obtiene efectivamente una linealización por conjuntos y finalmente en la sección 3.4.4 se demuestra que la linealización por conjuntos, dada por la sección 3.4.3, es una implementación de la cola con multiplicidad, o más formalmente, se cumplen las transiciones de estado que corresponden a la cola, dada por la definición 3.1.1.

Se denota una operación $\boldsymbol{Dequeue}$ que devuelve el valor $\boldsymbol{x}$ por $\langle \boldsymbol{Dequeue}(\,) : \boldsymbol{x} \rangle$, mientras que una operación $\boldsymbol{Enqueue}$ que inserta un valor $\boldsymbol{x}$ la denotamos por $\langle \boldsymbol{Enqueue}(\boldsymbol{x}) : True \rangle$.

3.4.1. Prueba de *non-blocking*: Set-Queue

Veamos que el algoritmo 4 es *non-blocking*. Supongamos una ejecución infinita del algoritmo Set-Queue, sea $\boldsymbol{op}$ una operación cualquiera de un proceso correcto (únicamente puede ser

un ***Dequeue*** o ***Enqueue***), veamos que es *non-blocking*.

- Supongamos que ***op*** es una operación ***Enqueue*** que no se completa (no retorna respuesta). Notemos primero que, para que la operación no sea completada, se deben realizar infinitas iteraciones sobre el ciclo ***while***. Sin embargo, para que la operación itere de forma infinita, basta con que el método ***CompareAndSet*** en la línea 6 sea ***False*** para cada iteración (nunca se alcanza la instrucción ***return***), veamos a detalle que implica esta condición.

  Para el algoritmo 4 estamos suponiendo que cada valor en la cola es una variable con la operación atómica ***CompareAndSet***, al igual que las variables compartidas ***Top*** y ***Tail***, es importante mencionar que es atómica, pues esto implica que efectivamente podemos tomar su ejecución como un evento. Ahora veamos lo siguiente, en el algoritmo 4 hay varias instrucciones que ejecutan un método ***CompareAndSet***, pero la única que altera el valor interno en los nodos es la línea 6 (cuando se ejecuta con éxito, retornando ***True***). Regresando al caso en donde la operación ***op*** es una operación ***Enqueue*** que no se completa, tenemos que en cada iteración la instrucción $\boldsymbol{t.next.CompareAndSet(n, x)}$. falla, lo cual significa que el valor del nodo ***t.next*** fue alterado, y según lo discutido la única instrucción que lo pudo haber alterado fue la ejecución exitosa (***CompareAndSet*** que retorna ***True***) de la línea 6 para alguna otra operación ***Enqueue′***. Sin embargo, al cumplirse esto para ***Enqueue′*** se alcanza la instrucción ***return*** en un número finito de pasos, e incluso, podríamos tomar esta ejecución como el punto donde la operación tiene efecto y tomarla como la respuesta de la operación (más adelante detallaremos esto en los puntos de linealización).

- Supongamos que ***op*** es una operación ***Dequeue*** que no se completa. Al igual que la operación ***Enqueue***, para que la operación ***Dequeue*** no se complete se deben realizar infinitas iteraciones sobre el ciclo ***while***, para esto deben cumplirse un par de condiciones. Primero, debe cumplirse que el nodo en el cabezal de la cola sea el mismo que el nodo final (cola de un único nodo), es decir, $t = h$, pues en caso contrario $t \neq h$ se ejecutan las líneas 27 y 28, en las cuales el nodo cabezal es reemplazado por el nodo siguiente h.next por medio de la operación atómica $CompareAndSet$, independientemente del resultado la operación retornaría al ejecutar la línea 28. Por lo tanto, $t = h$ para cada iteración en el ciclo, aún más, si resulta que no hay nodos, es decir $h.\text{next} = \epsilon$ la operación finaliza retornando $\boldsymbol{\epsilon}$ en la línea 22, en consecuencia, en cada iteración $\boldsymbol{h.\text{next} \neq \epsilon}$.

  Teniendo en cuenta que el estado inicial de la cola es ϵ, es decir que está vacía, es imposible que en cada iteración del ciclo *while* de la operación ***Dequeue*** encuentre un nodo infinitamente, puesto que el estado de la cola está constituido por un número finito de nodos. Por lo tanto, debe existir una operación ***Enqueue*** que ejecute satisfactoriamente $\boldsymbol{t.\text{next}.CompareAndSet(t, n)}$ (que retorne ***True***, lo cual implica que se añade un nodo a la cola), por lo visto en el punto anterior esto implica que la operación ***Dequeue*** se complete. En conclusión, si la operación ***Dequeue*** no se completa, existe una operación ***Enqueue*** que se completa.

En cualquiera de las dos operaciones del algoritmo 4 se tiene que, para que la operación no sea completada, se deben realizar infinitas iteraciones sobre un ciclo, y por cada iteración se completa una operación. Por lo tanto, se completan infinitas operaciones y se concluye que el algoritmo 4 es *non-blocking*.

3.4.2. Procedimiento de linealización por Conjuntos: Set-Queue

Dado que el algoritmo es *non-blocking* cualquier ejecución finita E que tenga operaciones pendientes, se puede dar una ejecución E' que extienda a E, en donde no se invoquen nuevas operaciones. Entonces, eventualmente en E' todas las operaciones pendientes de E se completarán. Por lo que se puede suponer, sin pérdida de generalidad, que cualquier ejecución finita E no tiene operaciones pendientes.

Sea E una ejecución finita sin operaciones pendientes del Algoritmo 4: Set-Queue. En la linealización por conjuntos, el estado del objeto está codificado en los nodos almacenados por la variable ***Head*** (dejando de lado el nodo centinela). Es fácil dar una representación matemática del *estado* de la cola; más adelante se detallará esto.

Para linealizar por conjuntos la ejecución E, procederemos a construir una linealización S de E asignando un punto de linealización a cada operación ***op*** de E. El punto de linealización de ***op***, denotado $LinPt(\boldsymbol{op})$, es un evento primitivo e entre la invocación $inv(\boldsymbol{op})$ y la respuesta $res(\boldsymbol{op})$. Tengamos en cuenta que varias operaciones pueden tener el mismo punto de linealización. Por lo tanto, la linealización S también se proporciona explícitamente. Finalmente, veremos que esta linealización S induce una relación de orden total $<_S$, y que S cumple con la especificación de cola (3.1.1).

Sea ***op*** en E, daremos las instrucciones de linealización definiendo primero el punto de linealización $LinPt(\boldsymbol{op})$, en el cual ***op*** pareciera tener efecto. Recordemos que $LinPt(\boldsymbol{op})$ es un evento primitivo, lo cual significa que es la ejecución de alguna línea del algoritmo 4, no una operación.

1. Si ***op*** es una operación ***Enqueue*** en E: Consideremos la última iteración de su ciclo ***while***. Dentro de esta última iteración, llamémosle $\boldsymbol{e_{CAS}}$ al paso que ejecuta la instrucción ***CompareAndSet*** de la línea 6. Observemos que, dado que suponemos que la operación está completada y es la última iteración, el ***CompareAndSet*** de $\boldsymbol{e_{CAS}}$ debió de ser exitoso; esta observación es primordial, pues es donde el método tiene efecto.

   Linealizamos por conjuntos la operación ***Enqueue*** en el punto $\boldsymbol{e_{CAS}}$, de modo que es una clase de concurrencia por sí misma.

2. Si ***op*** es una operación ***Dequeue*** en E, tenemos dos casos.

   a) Retorna la cadena vacía $\langle \boldsymbol{Dequeue}(\,) : \boldsymbol{\epsilon} \rangle$: Entonces la operación ***Dequeue*** en E retorna en la línea 22. Consideremos la última iteración del ciclo ***while*** de la operación ***Dequeue***, y llamémosle $\boldsymbol{e_{get}}$ al paso correspondiente a la línea 17 de

esta iteración; este paso es donde se extrae el nodo $\boldsymbol{h}$ almacenado de la variable $\boldsymbol{Head}$ por medio del método $\boldsymbol{get}()$. Linealizamos por conjuntos las operaciones $\boldsymbol{Dequeue}$ que retornan $\boldsymbol{\epsilon}$, en el punto $\boldsymbol{e_{get}}$, de modo es una clase de concurrencia por sí misma.

b) Retorna un elemento no nulo $\langle \boldsymbol{Dequeue}(\,) : \boldsymbol{x} \rangle$: Entonces la operación $\boldsymbol{Dequeue}$ retorna en la línea 28.

Consideremos el conjunto U_x, tal que contiene todas las operaciones $\boldsymbol{Dequeue}$ en E que devuelven el mismo elemento $\boldsymbol{x}$. Esto corresponde a las operaciones $\boldsymbol{Dequeue}$ que retornan $\boldsymbol{n.value}$ tal que el valor de esta cadena es $\boldsymbol{x}$.

Por suposición, cada elemento se coloca en la cola como máximo una única vez, por lo tanto, cada operación $\boldsymbol{Dequeue}$ de U_x extrae al mismo nodo en su última ejecución de la línea 19, denotemos por $\boldsymbol{n_x}$ a este nodo, cuyo valor es $\boldsymbol{x}$.

Consideremos la última iteración de cada operación $\boldsymbol{Dequeue}$ de U_x, cada una de estas ejecutan la línea 27, es decir $\boldsymbol{Head.CompareAndSet(h_x, n_x)}$. Recordemos que este método es una operación atómica, por lo tanto, podemos considerar estas ejecuciones como un conjunto de eventos, dado que los eventos están totalmente ordenados, podemos tomar el primero de todos estos, llamémosle $\boldsymbol{e^x_{CAS}}$, la cual es la primera ejecución de la línea 27 de las operaciones $\boldsymbol{Dequeue}$ de U_x (de nuevo, únicamente estamos pensando en la última iteración de cada operación). Linealizamos todas las operaciones en U_x en el paso $\boldsymbol{e^x_{CAS}}$, es decir, las operaciones en U_x forman una clase de concurrencia ubicada en $\boldsymbol{e^x_{CAS}}$.

Llamémosle $\boldsymbol{op}$ a una clase de concurrencia de S (abusando de la notación). Observemos que podemos clasificar las clases de concurrencia de la siguiente manera:

1. Se tiene que $\boldsymbol{op}$ es una clase de concurrencia correspondiente a una operación $\boldsymbol{Enqueue(a)}$ en E.

2. Se tiene que $\boldsymbol{op}$ es una clase de concurrencia correspondiente a un conjunto operaciones $\boldsymbol{Dequeue(\,)}$ en E, tenemos dos subcasos:

   a) La clase de concurrencia $\boldsymbol{op}$ corresponde a una sola operación $\boldsymbol{Dequeue(\,)}$ tal que $\langle \boldsymbol{Dequeue}(\,) : \boldsymbol{\epsilon} \rangle$.

   b) La clase de concurrencia $\boldsymbol{Dequeue}$ corresponde a un conjunto de operaciones $U_x = \{\boldsymbol{Dequeue_1}(\,), ..., \boldsymbol{Dequeue_t}(\,)\}$ tal que para toda operación $\boldsymbol{Dequeue_i(\,)}$ devuelve: $\langle \boldsymbol{Dequeue_i}(\,) : \boldsymbol{x} \rangle$ con $\boldsymbol{x}$ un elemento no nulo.

Estas clases de concurrencia que hemos definido dan la ejecución por conjuntos secuencial S. Observe que cada clase de concurrencia $\boldsymbol{op}$ de S tiene un punto de linealización por

construcción $LinPt(\boldsymbol{op})$, el cual es a su vez un punto de linealización para una o varias operaciones de E.

Ahora definamos $<_S$ para las clases de concurrencia, dadas dos clases de concurrencia $\boldsymbol{op}, \boldsymbol{op'}$ en S, se define $\boldsymbol{op} <_S \boldsymbol{op'}$ si y solo si $LinPt(\boldsymbol{op}) \rightarrow LinPt(\boldsymbol{op'})$. Esta relación es de orden total sobre las clases de concurrencia de S.

3.4.3. Linealización de la ejecución E

Las clases de concurrencia que hemos definido dan la ejecución secuencial S. Para mostrarlo primero veamos que dada una clase de concurrencia $\boldsymbol{op}$ de S, su punto de linealización $LinPt(\boldsymbol{op})$ siempre está entre los eventos de invocación y de respuesta de su conjunto de operaciones $\boldsymbol{op_E}$, es decir que cumple

$$inv(\boldsymbol{op_E}) \rightarrow LinPt(\boldsymbol{op}) \rightarrow res(\boldsymbol{op_E}). \tag{3.4.3.1}$$

Sea $\boldsymbol{op}$ en S, y $LinPt(\boldsymbol{op})$ su punto de linealización dado por las instrucciones en la sección 3.4.2, probemos que se cumple (3.4.3.1).

1. Si $\boldsymbol{op}$ es una clase de concurrencia correspondiente a una operación $\boldsymbol{Enqueue(a)}$ de E.

   Dado que $LinPt(\boldsymbol{op}) = \boldsymbol{e_{CAS}}$, donde $\boldsymbol{e_{CAS}}$ es la ejecución de la instrucción $\boldsymbol{CompareAndSet}$ de la línea 6 de $\boldsymbol{Enqueue(a)}$, de la última iteración de su ciclo $\boldsymbol{while}$. Claramente $inv(\boldsymbol{Enqueue(a)}) \rightarrow \boldsymbol{e_{CAS}}$, pues la invocación precede a cualquier evento en las iteraciones del ciclo $\boldsymbol{while}$, y $\boldsymbol{e_{CAS}} \rightarrow res(\boldsymbol{Enqueue(a)})$, pues la respuesta de la operación $\boldsymbol{Enqueue(a)}$ corresponde al evento asociado a la ejecución de la línea 8.

2. Si $\boldsymbol{op}$ es una clase de concurrencia correspondiente a un conjunto operaciones $\boldsymbol{Dequeue()}$ en E, tenemos dos subcasos:

   a) La clase de concurrencia $\boldsymbol{op}$ corresponde a una sola operación $\boldsymbol{Dequeue(\,)}$ tal que $\langle \boldsymbol{Dequeue}(\,) : \epsilon \rangle$.

      El punto de linealización es $LinPt(\boldsymbol{op}) = \boldsymbol{e_{get}}$, los argumentos son análogos al punto anterior, pues el punto de linealización es un evento entre la invocación y la respuesta de la operación.

   b) La clase de concurrencia $\boldsymbol{op}$ corresponde a un conjunto de operaciones $U_x = \{\boldsymbol{Dequeue_1}(\,), ..., \boldsymbol{Dequeue_t}(\,)\}$ tal que para toda operación $\boldsymbol{Dequeue_i(\,)}$ devuelve: $\langle \boldsymbol{Dequeue_i}(\,) : \boldsymbol{x} \rangle$ con $\boldsymbol{x}$ un elemento no nulo.

      En este caso, dado que estamos linealizando un conjunto de operaciones, debemos probar que, para toda operación $\boldsymbol{Dequeue_i} \in U_x$ se tiene

      $$inv(\boldsymbol{Dequeue_i}) \rightarrow LinPt(\boldsymbol{op}) \rightarrow res(\boldsymbol{Dequeue_i}),$$

donde U_x es el conjunto de todas las operaciones ***Dequeue*** en E que devuelven el mismo elemento $\boldsymbol{x}$.

Recordemos que $LinPt(\boldsymbol{op}) = \boldsymbol{e^x_{CAS}}$, con $\boldsymbol{e^x_{CAS}}$ el evento descrito en la instrucción 2.b) del procedimiento 3.4.2. Observemos primero lo siguiente: $\boldsymbol{e^x_{CAS}}$ es la ejecución de la instrucción $\boldsymbol{Head.CompareAndSet(h_x, n_x)}$ y esta debe ser exitosa.

Para mostrarlo supongamos lo contrario, entonces el nodo $\boldsymbol{h_x}$ de la variable ***Head*** debió de haber sido reemplazado por otra operación $\boldsymbol{op'}$, y la única instrucción del algoritmo 4 que cambia el valor de ***Head*** en la línea 27. Aún más, dado que el nodo $\boldsymbol{h_x}$ cambió, la operación $\boldsymbol{op'}$ debió de haber ejecutado exitosamente $\boldsymbol{Head.CompareAndSet(h_x, n_x)}$ (con el mismo nodo $\boldsymbol{n_x}$ de $\boldsymbol{op}$, pues es el mismo cabezal $\boldsymbol{h_x}$). Por lo que esta operación $\boldsymbol{op'}$ es un ***Dequeue*** que retorna $\boldsymbol{n_x.value}$, pero este valor es justamente $\boldsymbol{x}$, entonces $\boldsymbol{op'}$ está en U_x y tendríamos que la ejecución de la línea 27 de $\boldsymbol{op'}$ antecede a $\boldsymbol{e^x_{CAS}}$ lo cual contradice su definición. La contradicción viene de suponer que en $\boldsymbol{e^x_{CAS}}$ falla $\boldsymbol{Head.CompareAndSet(h_x, n_x)}$, por lo tanto, se cumple la observación.

Para mostrar primero que $inv(\boldsymbol{Dequeue_i}) \rightarrow LinPt(\boldsymbol{op})$ para toda operación de U_x, supongamos que no se cumple, que existe una operación en U_x tal que $LinPt(\boldsymbol{op}) \rightarrow inv(\boldsymbol{Dequeue'})$. Sabemos por la observación anterior que en $\boldsymbol{e^x_{CAS}}$ la ejecución de $\boldsymbol{Head.CompareAndSet(h_x, n_x)}$ es exitosa, además estamos suponiendo que cada nodo se coloca en la cola como máximo una vez, por lo que cualquier ejecución de $\boldsymbol{Head.get()}$ después de $\boldsymbol{e^x_{CAS}}$ extrae un nodo $\boldsymbol{h}$ distinto de $\boldsymbol{h_x}$.

Dado que $\boldsymbol{e^x_{CAS}} \rightarrow inv(\boldsymbol{Dequeue'})$, para cualquier ejecución de la línea 17 de $\boldsymbol{Dequeue'}$, llamémosle $\boldsymbol{e'}$, se cumple que $inv(\boldsymbol{Dequeue'}) \rightarrow \boldsymbol{e'}$ y por transitividad $\boldsymbol{e^x_{CAS}} \rightarrow \boldsymbol{e'}$. Por lo tanto, todo nodo $\boldsymbol{h'}$ extraído de la línea 17 de $\boldsymbol{Dequeue'}$, es distinto de $\boldsymbol{h_x}$ y $\boldsymbol{n' = h.next.get()}$ también es diferente de $\boldsymbol{n_x}$. Los argumentos son los mismos que $\boldsymbol{h_x}$, lo cual implica que el retorno $\boldsymbol{n'.value}$ de $\boldsymbol{Dequeue'}$ sea diferente de $\boldsymbol{n_x.value}$ contradiciendo que $\boldsymbol{Dequeue'}$ este en U_x. Por lo tanto, debe pasar que $inv(\boldsymbol{Dequeue_i}) \rightarrow LinPt(\boldsymbol{op})$ para toda operación de U_x.

Para finalizar debemos mostrar que $LinPt(\boldsymbol{op}) \rightarrow res(\boldsymbol{Dequeue_i})$ para toda operación de U_x. Este caso es mucho más fácil que el anterior. Basta con observar que $\boldsymbol{e^x_{CAS}}$ es el punto de linealización y por definición es la primera ejecución de la instrucción $\boldsymbol{Head_i.CompareAndSet(h_x, n_x)}$ para cada operación $\boldsymbol{Dequeue_i}$ de U_x, en otras palabras
$\boldsymbol{e^x_{CAS}} \rightarrow \boldsymbol{Head_i.CompareAndSet(h_x, n_x)}$. Además, para cada operación $\boldsymbol{Dequeue_i}$ de U_x su respuesta precede esta instrucción, es decir
$\boldsymbol{Head_i.CompareAndSet(h_x, n_x)} \rightarrow res(\boldsymbol{Dequeue_i})$ y por transitividad se cumple $LinPt(\boldsymbol{op}) \rightarrow res(\boldsymbol{Dequeue_i})$.

Se concluye que: Para toda operación $\boldsymbol{Dequeue_i}$ en U_x se tiene

$$inv(\boldsymbol{Dequeue_i}) \rightarrow LinPt(\boldsymbol{op}) \rightarrow res(\boldsymbol{Dequeue_i}),$$

donde U_x es el conjunto de todas las operaciones $\boldsymbol{Dequeue}$ en E que devuelven el mismo elemento $\boldsymbol{x}$.

Para concluir que S es una linealización por conjuntos de E veamos que, por construcción, E y S tienen las mismas operaciones con las mismas respuestas. Por otro lado, si consideramos las operaciones $\boldsymbol{op_1}$ y $\boldsymbol{op_2}$ de E tal que $\boldsymbol{op_1} <_E \boldsymbol{op_2}$, por lo que $res(\boldsymbol{op_1}) \rightarrow inv(\boldsymbol{op_2})$. En S, cada operación se linealiza en un paso que se encuentra entre la invocación y la respuesta de la operación. Por lo tanto, la clase de concurrencia de $\boldsymbol{op_1}$ aparece antes que la clase de concurrencia de $\boldsymbol{op_2}$ en S, pues se tiene

$$LinPt(\boldsymbol{op_1}) \rightarrow res(\boldsymbol{op_1}) \rightarrow inv(\boldsymbol{op_2}) \rightarrow LinPt(\boldsymbol{op_2})$$

por transitividad $LinPt(\boldsymbol{op_1}) \rightarrow LinPt(\boldsymbol{op_2})$, y por definición $\boldsymbol{o\tilde{p}_1} <_S \boldsymbol{o\tilde{p}_2}$, donde $\boldsymbol{o\tilde{p}_1}$, $\boldsymbol{o\tilde{p}_1}$ son las clases de concurrencia de $\boldsymbol{op_1}$ y $\boldsymbol{op_2}$ respectivamente. Concluimos que S es una linealización por conjunto de E. Ahora falta ver que S cumple con la especificación de cola.

3.4.4. Especificación de cola con multiplicidad: Algoritmo Set-Queue

En esta sección demostramos que el algoritmo Set-Queue satisface la especificación de cola con multiplicidad 3.1.1. Con el fin de simplificar la demostración definiremos primero el *estado* de la cola. El *estado* de la cola tiene la función de representar matemáticamente los nodos almacenados en memoria que representan a la estructura, en este caso a la cola. La definición es la siguiente:

Definición 3.4.1 *Sea* $\hat{\boldsymbol{q}} \in \mathbb{N}^*$*, el cual es de la forma*

$$\hat{\boldsymbol{q}} = \boldsymbol{a_1 a_2 ... a_n}$$

con $\boldsymbol{a_i} \in \mathbb{N}$ *para* $i \in \{1, ..., k\}$*. Decimos que* $\hat{\boldsymbol{q}}$ *representa el estado de la cola si el nodo* $\boldsymbol{t} = \boldsymbol{Head.get()}$ *cumple:*

- *El nodo dado por* $\boldsymbol{t_1} = \boldsymbol{t.next}$ *es de tipo* $\boldsymbol{AtomicReference}$ *y su valor numérico* $\boldsymbol{t_1.value}$ *es igual a* $\boldsymbol{a_1}$*.*
- *Todos para cualquier nodo* $\boldsymbol{t_i} = \boldsymbol{t_{i-1}.next}$ *con* $i = 2, ..., n$ *de tipo* $\boldsymbol{AtomicReference}$ *y su valor numérico* $\boldsymbol{t_i.value}$ *es igual a* $\boldsymbol{a_i}$*.*
- *El último nodo es la cadena nula, es decir* $\boldsymbol{\epsilon} = \boldsymbol{t_n.next}$*.*

Observemos que el estado de la cola está perfectamente representado por la variable $\boldsymbol{Head}$, pues contiene los nodos $\boldsymbol{\epsilon}, \boldsymbol{t_1}, ..., \boldsymbol{t_n}$, tales que conforman a la cola. Además, en memoria, la variable $\boldsymbol{Head}$ contiene el primer nodo $\boldsymbol{h_x} = \boldsymbol{Head.get()}$, mismo que no se toma en cuenta para representar el estado de la cola.

Al primer nodo del cabezal $\boldsymbol{h_x}$ = ***Head.get( )*** se le denomina *nodo centinela*, este nodo, si bien no participa en la representación de la cola, cumple un rol fundamental, pues sin el nodo centinela los apuntadores ***Head*** y ***Tail*** podrían llegar a representar estructuras distintas, es decir, debido a la concurrencia se podrían producir intervalos de tiempo donde la variable ***Tail*** contiene un nodo mientras que ***Head*** contiene un nodo vació $\boldsymbol{\epsilon}$, o viceversa, esto provocaría transiciones de estado que no corresponden a una cola. Más adelante veremos que es posible representar el *estado* de una estructura por medio de restricciones, en este caso la restricción es implícita, pues siempre se ignora el *primer* nodo en el cabezal.

Set-Queue satisface la especificación de cola con multiplicidad.
Demostración:

Sea E una ejecución finita sin operaciones pendientes del algoritmo 4: Set-Queue. Demostremos que cumple con la especificación de cola 3.1.1. Donde el estado de la cola está dado por el estado lógico 3.4.1.

Sea S la linealización por conjuntos de E obtenida por el procedimiento de linealización 3.4.2. Tenemos que las clases de concurrencia están ordenadas como

$$\boldsymbol{op_1} <_S \boldsymbol{op_2} <_S ... <_S \boldsymbol{op_n}$$

Donde op_i es la clase de concurrencia de alguna operación de E y n es el número de clases de concurrencia definidas por $<_S$. Sea E_m el prefijo de E con las operaciones $\{\boldsymbol{op_1}...\boldsymbol{op_m}\}$ y S_m su correspondiente linealización por conjuntos, la cual de hecho induce el orden $\boldsymbol{op_1} <_S ... <_S \boldsymbol{op_m}$.

Demostremos que S_m satisface la especificación de cola 3.1.1 para toda $m \leq n$, cuyo estado está dado por S_m, por inducción sobre m.

El caso base es trivial, pues $E = \emptyset$, por lo que $S = \emptyset$, y el estado de la cola es el mismo después de ejecutar S o E. Como hipótesis de inducción supongamos que la ejecución secuencial por conjuntos S_m de E_m cumple con la especificación de cola, cuyo estado está dado por el estado lógico S_m.

Para el paso inductivo demostremos que se cumple para E_{m+1} cuya linealización por conjuntos es S_{m+1} e induce un orden total en las clases de equivalencia y podemos ordenarlas como:

$$\boldsymbol{op_0} <_S \boldsymbol{op_1} <_S ... <_S \boldsymbol{op_{m+1}}.$$

Por hipótesis de inducción el prefijo E_m cumple con lo requerido, basta mostrar que al ejecutar la operación $\boldsymbol{op_{m+1}}$ la transición de estados corresponde a la especificación de cola 3.1.1.

Sea ***Head*** la variable compartida antes de ejecutar $\boldsymbol{op_{m+1}}$, observemos que por hipótesis de inducción el estado de ***Head*** es el mismo que se obtiene de ejecutar S_m, por lo que se

ejecutan las clases de concurrencia en orden: $\boldsymbol{op_0} <_S \boldsymbol{op_1} <_S ... <_S \boldsymbol{op_m}$.

Sea $\hat{\boldsymbol{q}} \in \mathbb{N}^*$ el estado lógico de la cola después de ejecutar la clase de concurrencia $\boldsymbol{op_m}$.

1. Si $\boldsymbol{op_{m+1}}$ es una clase de concurrencia correspondiente a una operación $\boldsymbol{Enqueue(a)}$ en E_m: El punto de linealización de $\boldsymbol{op}$ es $LinPt(\boldsymbol{Enqueue(a)}) = \boldsymbol{e_{CAS}}$, el cual corresponde a la ejecución de la línea 6 con un $\boldsymbol{CompareAndSet}$ exitoso en su última iteración.

   Basta ver que si el estado en $\boldsymbol{op_m}$ es $\hat{\boldsymbol{q}}$, el estado después de ejecutar $\boldsymbol{Enqueue(a)}$ es $\hat{\boldsymbol{q}} * \boldsymbol{a}$. De este modo se cumpliría formalmente la transición:

   $$\delta(\hat{\boldsymbol{q}}, \boldsymbol{Enqueue(a)}) = (\hat{\boldsymbol{q}} * \boldsymbol{a}, \langle \boldsymbol{Enqueue(a)} : \boldsymbol{True} \rangle)$$

   Observemos que el estado de la cola debe de ser $\hat{\boldsymbol{q}}$ antes del evento $\boldsymbol{e_{CAS}}$, pues en caso contrario algún evento lo modificó, y este evento debe de estar entre $\boldsymbol{op_m}$ y $\boldsymbol{op_{m+1}}$ (podemos asegurar esto debido al orden total entre eventos). Para probar la afirmación anterior supongamos lo contrario, que algún evento $\boldsymbol{e}$ alteró el estado de la cola y que este evento está entre $\boldsymbol{op_m}$ y $\boldsymbol{op_{m+1}}$.

   Dada la definición del estado de la cola, se puede asegurar que las únicas instrucciones que alteran el estado de la cola son la ejecución de la línea 6 o de la línea 27 (con un $\boldsymbol{CompareAndSet}$ exitoso). Esto debido a que las instrucciones de las líneas 7, 11 y 24 alteran a la variable $\boldsymbol{Tail}$ la cual funciona para apuntar al final de la pila, pero en realidad no alteran el estado de la cola dado por la definición 3.1.1. Por otro lado, las instrucciones restantes únicamente son extracciones de una variable por métodos $\boldsymbol{get}$ o evaluaciones.

   Sabemos que el evento $\boldsymbol{e}$ que alteró el estado de la cola forzosamente fue la ejecución de la línea 6 o 27. Sin embargo, en el caso que $\boldsymbol{e}$ es la ejecución de la línea 6, con un $\boldsymbol{CompareAndSet}$ exitoso, corresponde justamente al punto de linealización de alguna operación $\boldsymbol{Enqueue(b)}$ en E (para algún $\boldsymbol{b}$); esto es claro, pues al ser exitoso se ejecutan las líneas 7 y 8 correspondiendo a la última iteración de $\boldsymbol{Enqueue(b)}$. Entonces, como este evento $\boldsymbol{e}$ cumple $\boldsymbol{op_m} \to \boldsymbol{e} \to \boldsymbol{op_{m+1}}$ y es un punto de linealización, tendríamos que existir una operación $\boldsymbol{op_k}$ tal que

   $$\boldsymbol{op_m} \to \boldsymbol{op_k} \to \boldsymbol{op_{m+1}},$$

   contradiciendo que el orden que induce la linealización por conjuntos S_{m+1}.

   Por otro lado, si el evento $\boldsymbol{e}$ que alteró el estado de la cola fue la ejecución de la línea 27, con un $\boldsymbol{CompareAndSet}$ exitoso, entonces fue ejecutado por alguna operación $\boldsymbol{Dequeue'()}$. Veremos que ocurre lo mismo que el caso anterior pero con el punto de linealización de un conjunto de operaciones $\boldsymbol{Dequeue()}$. Para mostrar esto simplemente notemos lo siguiente, si $\boldsymbol{b}$ es el valor retornado por $\boldsymbol{Dequeue'()}$ y U_b el conjunto

de operaciones $\boldsymbol{Dequeue()}$ tal que devolvieron $\boldsymbol{b}$, entonces el punto de linealización es un $\boldsymbol{Head.CompareAndSet(h_n, n_b)}$ exitoso (con $\boldsymbol{b}$ el valor del nodo $\boldsymbol{n_b}$) no puede haber otro $\boldsymbol{CompareAndSet}$ exitoso con los mismos nodos, pues estamos suponiendo que cada nodo solo se encola una única vez (cada nodo es único) entonces debe corresponder al evento $\boldsymbol{e}$. Por lo tanto, el evento $\boldsymbol{e}$, que alteró el estado de la cola, corresponde al punto de linealización de un conjunto de operaciones $\boldsymbol{Dequeue()}$. Usando los mismos argumentos que el caso anterior, se contradice el orden que induce la linealización por conjuntos S_{m+1}.

Por lo tanto, el estado de la cola entre del evento $LinPt(\boldsymbol{op_m})$ y antes del evento $\boldsymbol{e_{CAS}}$ es $\boldsymbol{\hat{q}}$. Para finalizar mostremos en $\boldsymbol{e_{CAS}}$ el estado cambia según la transición 3.1.1 y que sigue cumpliendo con la definición de estado de cola.

Supongamos que el estado es de la forma $\boldsymbol{\hat{q} = a_1...a_k}$ y que $\boldsymbol{t_j}$ es el nodo con valor $\boldsymbol{a_j}$, entonces debemos primero mostrar que $\boldsymbol{t_k}$ es $\boldsymbol{t}$. Supongamos que $\boldsymbol{t}$ es distinto de todos los nodos $\boldsymbol{t_j}$. Como $\boldsymbol{t = Tail.get()}$ podemos decir que $\boldsymbol{Tail}$ no contiene nodos de la cola, empero, estamos suponiendo que el estado inicial de la cosa es tal que $\boldsymbol{i = Head.get() = Tail.get()}$, lo cual corresponde a una cola vacía con $\boldsymbol{Head}$ y $\boldsymbol{Tail}$ apuntando al nodo centinela $\boldsymbol{i}$.

Podemos decir que en algún evento $\boldsymbol{e}$ se ejecutó la instrucción
$\boldsymbol{Tail.CompareAndSet(t, x)}$ exitosamente, donde $\boldsymbol{t}$ si estaba en la cola, pero $\boldsymbol{x}$ no, es decir, que si $\boldsymbol{h = Head.get()}$ con $\boldsymbol{t_1 = h.next}$ y $\boldsymbol{t_i = t_{i-1}.next}$ para $i = 2...k$, el nuevo nodo es distinto para todos los nodos $\boldsymbol{x \neq t_i}$ para toda $i = 1..., k$. Sin embargo, los únicos eventos que cambian $\boldsymbol{Tail}$ son las líneas 7, 11 y 24.

Observemos que en las líneas 11 y 24 es imposible que esto ocurra, pues $\boldsymbol{x = t.next}$, y estas líneas solo se ejecutan cuando $\boldsymbol{x \neq \epsilon}$, por lo que cambian el valor del nodo en $\boldsymbol{Tail}$ por otro que sigue siendo algún $\boldsymbol{t_i}$, pues $\boldsymbol{t}$ si estaba en la cola. Podemos pensar esto como una prueba por inducción.

En el caso de la línea 7, se efectúa solo cuando la instrucción
$\boldsymbol{t.next.CompareAndSet(n, x)}$ es exitosa, por lo tanto, $\boldsymbol{t.next = x}$, de modo que para que $\boldsymbol{x}$ no este en la cola, algún otro evento debió de haber efectuado otro $\boldsymbol{t.next.CompareAndSet(n, y)}$ exitoso antes, pero esto tampoco es posible, pues $\boldsymbol{CompareAndSet}$ es atómico y cualquier otro método de esa forma después de que $\boldsymbol{t.next.CompareAndSet(n, x)}$ sea exitoso falla, por definición del método. Por lo tanto, $\boldsymbol{x}$ sigue estando en la cola. Entonces, si el estado es de la forma $\boldsymbol{\hat{q} = a_1...a_k}$ y que $\boldsymbol{t_j}$ es el nodo con valor $\boldsymbol{a_j}$, entonces algún $\boldsymbol{t_j}$ es $\boldsymbol{t}$. Es fácil ver que debe ser $\boldsymbol{t_k}$, pues si fuera $\boldsymbol{t = t_j}$ con $j < k$ entonces $\boldsymbol{t.next \neq \epsilon}$, y no se habría ejecutado con éxito la línea 5. Por lo que $\boldsymbol{t = t_k}$.

Como el estado antes de $\boldsymbol{e_{CAS}}$ es $\boldsymbol{\hat{q} = a_1...a_k}$ donde $\boldsymbol{t_j}$ es el nodo con valor $\boldsymbol{a_j}$, y después de $\boldsymbol{e_{CAS}}$ el nodo $\boldsymbol{t_k.next = a}$, entonces el estado ahora es $\boldsymbol{\hat{q} * a}$. En conclusión,

$\boldsymbol{e_{CAS}}$ cambia el estado $\hat{\boldsymbol{q}}$ al estado $\hat{\boldsymbol{q}} * \boldsymbol{a}$, ya que inserta el nodo $\boldsymbol{t}$ con valor $\boldsymbol{a}$ tal que $\boldsymbol{t_k.next} = \boldsymbol{t}$. Dicho en otras palabras, para la operación $\boldsymbol{op_{m+1}}$ se cumple la transición:

$$\delta(\hat{\boldsymbol{q}}, \boldsymbol{Enqueue(a)}) = (\hat{\boldsymbol{q}} * \boldsymbol{a}, \langle \boldsymbol{Enqueue(a)} : \boldsymbol{True} \rangle)$$

Por lo que concluimos que S_{m+1} produce una ejecución secuencial de cola válida que es consistente con (3.1.1), donde estado de la cola está representado por el estado codificado en $\hat{\boldsymbol{q}} * \boldsymbol{a}$, almacenado o representado en la variable $\boldsymbol{Head}$.

2. Si $\boldsymbol{op_{m+1}}$ es una clase de concurrencia correspondiente a un conjunto operaciones $\boldsymbol{Dequeue(\,)}$ en E_m, tenemos dos casos:

   a) La clase de concurrencia $\boldsymbol{op_{m+1}}$ corresponde a una sola operación $\boldsymbol{Dequeue(\,)}$ tal que $\langle \boldsymbol{Dequeue}(\,) : \epsilon \rangle$.

   En este caso el punto de linealización es $\boldsymbol{e_{get}}$, correspondiente a la línea 17 de su última iteración, y la operación $\boldsymbol{Dequeue(\,)}$ retorna en la línea 22. Por suposición

   $$LinPt(\boldsymbol{op_m}) \rightarrow \boldsymbol{e_{get}}.$$

   Supongamos que el estado de la cola después de ejecutar $LinPt(\boldsymbol{op_m})$ es $\hat{\boldsymbol{q}}$. Podemos asegurar que entre $LinPt(\boldsymbol{op_m})$ y $LinPt(\boldsymbol{op_m}) = \boldsymbol{e_{get}}$ el estado no cambia, pues si se viera alterado entonces debería de existir algún evento $\boldsymbol{e}$ que lo cambio, esto claramente es lo mismo que el caso anterior, de hecho podemos afirmar de una forma más fuerte que: Entre cualesquiera dos operaciones $\boldsymbol{op_k}$ y $\boldsymbol{op_{k+1}}$ el estado de la cola nunca cambia, pues en caso contrario se contradice el orden impuesto por la linealización por conjuntos. Los argumentos son los mismos que usamos en el punto anterior y esto nos permite acortar la demostración.

   Ahora mostremos que el estado $\hat{\boldsymbol{q}}$ debe de ser $\boldsymbol{\epsilon}$, pues en caso contrario en el evento $\boldsymbol{e_{get}}$ se obtendría un nodo tal que $\boldsymbol{t.next} \neq \boldsymbol{\epsilon}$ (recuérdese la definición de estado de la cola), y esto provocaría que no se ejecutaría la línea 21 satisfactoriamente, por ende tampoco se ejecuta 22 contradiciendo $\langle \boldsymbol{Pop}(\,) : \epsilon \rangle$. Por lo tanto, $\hat{\boldsymbol{q}} = \boldsymbol{\epsilon}$ y en el punto de linealización $\boldsymbol{e_{get}}$ el estado no cambia, pues es únicamente la ejecución de un $\boldsymbol{get}$, de modo que el estado $\boldsymbol{\epsilon}$ de la cola permanece invariante. Dicho en otras palabras, para la operación $\boldsymbol{op_{m+1}}$ se cumple la transición:

   $$\delta(\boldsymbol{\epsilon}, \boldsymbol{Dequeue}(\,)) = (\boldsymbol{\epsilon}, \langle \boldsymbol{Dequeue}(\,) : \boldsymbol{\epsilon} \rangle).$$

   Por lo que concluimos que S_{m+1} produce una ejecución secuencial de cola válida que es consistente con (3.1.1), donde el estado de la cola está representado por el estado codificado en $\boldsymbol{\epsilon}$, almacenado en la variable $\boldsymbol{Head}$.

   b) La clase de concurrencia $\boldsymbol{op_{m+1}}$ corresponde a un conjunto de operaciones $U_x = \{\boldsymbol{Dequeue_1}(\,), ..., \boldsymbol{Dequeue_t}(\,)\}$ tal que para toda operación $\boldsymbol{Dequeue_i(\,)}$ devuelve: $\langle \boldsymbol{Dequeue_i}(\,) : \boldsymbol{x} \rangle$ con $\boldsymbol{x}$ un elemento no nulo. Recordemos que el conjunto U_x es tal que contiene todas las operaciones $\boldsymbol{Dequeue}$ en

E_{m+1} que devuelven el mismo elemento $\boldsymbol{x}$.

Ya sabemos que entre cualesquiera dos operaciones $\boldsymbol{op_k}$ y $\boldsymbol{op_{k+1}}$ el estado de la cola nunca cambia. Por lo tanto, basta con que mostremos que en el punto de linealización $LinPt(\boldsymbol{op})$ el estado de la cola cumple con la correspondiente transición en 3.1.1. Sea $\boldsymbol{a} * \hat{\boldsymbol{q}}$ el estado de la cola después de $\boldsymbol{op_m}$, podemos asegurar que el estado contiene al menos un elemento, pues en caso contrario la ejecución de la línea 21 habría retornado un valor nulo. El punto de linealización en este caso es $LinPt(\boldsymbol{op}) = \boldsymbol{e^x_{CAS}}$, el cual corresponde a la primera ejecución de la línea 27 de todas las operaciones $\boldsymbol{Dequeue_i}$ en U_x (pensando únicamente en la última iteración de cada operación en U_x).

En $\boldsymbol{e^x_{CAS}}$ se ejecuta la instrucción $\boldsymbol{Tail.CompareAndSet(h_x, n_x)}$, donde $\boldsymbol{n_x}$ es el nodo con valor $\boldsymbol{x}$. Como el estado de la cola no cambia entre $\boldsymbol{op_m}$ y $\boldsymbol{op_{m+1}}$, en el punto de linealización el estado debe ser $\boldsymbol{a} * \hat{\boldsymbol{q}}$, y como $\boldsymbol{h_x} = \boldsymbol{Head.get(\,)}$ y $\boldsymbol{h.next.get(\,)}$, entonces el valor de $\boldsymbol{n_x}$ por definición es $\boldsymbol{a}$, en otras palabras $\boldsymbol{x} = \boldsymbol{a}$. Esto también nos permite escribir el estado de la cola como $\boldsymbol{x} * \hat{\boldsymbol{q}}$.

Por otro lado, también sabemos que en $\boldsymbol{e^x_{CAS}}$ la ejecución de la instrucción $\boldsymbol{Tail.CompareAndSet(h_x, n_x)}$ es exitosa (de hecho es la única del conjunto U_x). Después de $\boldsymbol{e^x_{CAS}}$ la variable $\boldsymbol{Head}$ contiene el siguiente nodo, es decir $\boldsymbol{n_x} = \boldsymbol{Head.get(\,)}$, y sucede lo siguiente:

- Si $\hat{\boldsymbol{q}} = \boldsymbol{a_2 ... a_k}$, entonces, como $\boldsymbol{x} * \hat{\boldsymbol{q}}$ es el estado de una cola definición $\boldsymbol{t_2} = \boldsymbol{n_x.next}$ tiene valor $\boldsymbol{a_2}$ y $\boldsymbol{t_i} = \boldsymbol{t_{i-1}.next}$ tiene valor $\boldsymbol{a_i}$ para $i = 3, ...k$. Después de $\boldsymbol{e^x_{CAS}}$, el estado cambia a $\hat{\boldsymbol{q}} = \boldsymbol{a_2 ... a_k}$, basta ver que efectivamente cumple con la definición formal de la cola. Para esto simplemente notemos que después de $\boldsymbol{e^x_{CAS}}$, la variable $\boldsymbol{Head}$ cumple $\boldsymbol{n_x} = \boldsymbol{Head.get(\,)}$ (ahora es el nodo centinela) y por lo anterior se cumple la definición de estado de la cola. Por lo que podemos concluir que $\boldsymbol{e^x_{CAS}}$ cambia el estado de la cola $\boldsymbol{x} * \hat{\boldsymbol{q}}$ al estado $\hat{\boldsymbol{q}}$.
- Si $\hat{\boldsymbol{q}} = \boldsymbol{\epsilon}$, entonces forzosamente $\boldsymbol{n_x.next} = \boldsymbol{\epsilon}$, pues $\boldsymbol{x} * \hat{\boldsymbol{q}}$ es el estado de una cola y por definición ocuparía el papel del primer y último nodo. Después de $\boldsymbol{e^x_{CAS}}$, el nuevo nodo centinela es $\boldsymbol{n_x} = \boldsymbol{Head.get(\,)}$ y cumple $\boldsymbol{n_x.next} = \boldsymbol{\epsilon}$, y además $\hat{\boldsymbol{q}} = \boldsymbol{\epsilon}$, por definición $\hat{\boldsymbol{q}}$ representa el estado de la cola vacía.

Concluimos que, para la operación $\boldsymbol{op_{m+1}}$ se cumple la transición:

$$\delta(\hat{\boldsymbol{q}} * \boldsymbol{a}, \{\boldsymbol{Dequeue_1}(), ..., \boldsymbol{Dequeue_t}()\}) = (\hat{\boldsymbol{q}}, \{\langle \boldsymbol{Dequeue_1}(\,) : \boldsymbol{a}\rangle, ..., \langle \boldsymbol{Dequeue_t}(\,) : \boldsymbol{a}\rangle\})$$

Por lo que concluimos que S_{m+1} produce una ejecución secuencial de cola válida que es consistente con (3.1.1), donde estado de la cola está representado por el estado codificado en $\hat{\boldsymbol{q}}$, representado en la variable $\boldsymbol{Head}$

Por lo que concluimos que S_m produce una ejecución secuencial de cola válida que es consistente con (3.1.1) para toda $m \leq n$. Por consiguiente es válido para E y S.

3.4.5. Dinámica del algoritmo Set-Queue

Esta sección está destinada para mostrar el funcionamiento del algoritmo Set-Queue bajo algunos escenarios particulares. Cada hilo se representará por medio de una línea negra punteada, los eventos se representan como $\boldsymbol{e}$ más algunos índices.

Nótese que el diagrama podría ser analizado por medio de un reloj lógico de Lamport o vectorial [18, 24], los cuales permiten establecer un orden parcial a los eventos que ocurren en los procesos de un sistema distribuido. En este algoritmo, y los siguientes que se presentaran, la estructura misma del algoritmo determina que orden de eventos podría suceder durante la ejecución del algoritmo, y cuáles órdenes no son posibles, pues provocarían una contradicción con la misma estructura del algoritmo; lo cual puede ser catalogado como causalidad. Durante la demostración se pueden observar las contradicciones que surgen de suponer que algunas operaciones sean concurrente, por lo que los eventos de invocación/respuesta deben respetar la causalidad que impone la estructura del algoritmo. Por lo tanto, el algoritmo mismo induce de forma implícita un ordenamiento en ciertos eventos y esto es equivalente a los relojes de Lamport.

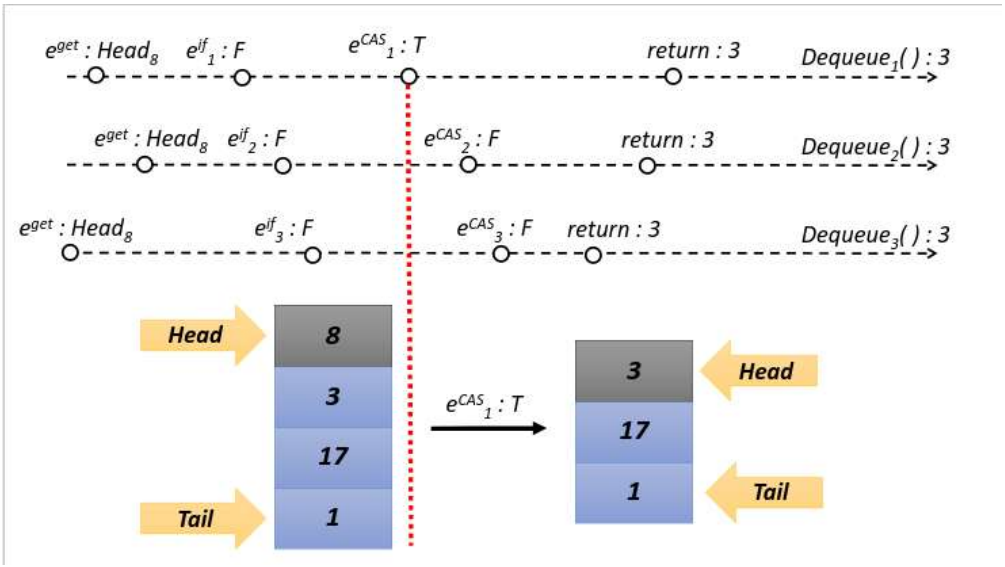


Figura 3.3: Representación de un conjunto dado de operaciones $\boldsymbol{Dequeue_i}$ que retornan todas el mismo valor. En este caso las tres operaciones se comportan como una misma, pues su clase de concurrencia es el conjunto de operaciones $\boldsymbol{Dequeue_i}$.

En la figura 3.3 se representan tres hilos que ejecutan concurrentemente una operación $\boldsymbol{Dequeue_i}$ cada uno. Consideraremos un caso donde todas las operaciones retornan el mismo elemento, en este caso se retorna el elemento 3. Como se puede apreciar durante las demostraciones, la iteración más relevante en las operaciones es siempre la última, en este ejemplo consideremos que estamos en la última iteración de cada $\boldsymbol{Dequeue_i}$. Consideremos que las operaciones ocurren de la siguiente forma:

- La cola está constituida por la lista de nodos $\boldsymbol{(8, 3, 17, 1)}$. El estado de la cola es $\boldsymbol{(3, 17, 1)}$, siendo el nodo con valor 8 el nodo centinela de la cola (recuadro representado

en color gris). Recordemos que el valor del nodo centinela no importa, en estos ejemplos se muestra su valor con el fin de explicar el mecanismo lo más explícitamente.

- La instrucción de la línea 17 se representa por medio de $\boldsymbol{e^{get}}$, donde cada operación extrae al nodo centinela de la referencia por medio de $\boldsymbol{Head.get(\,)}$. Del mismo modo, en la línea 18 cada operación extrae el nodo en la referencia de la cola por medio de $\boldsymbol{Tail.get(\,)}$. En todos los casos el nodo $\boldsymbol{h}$ tiene valor 8 (centinela) y el nodo $\boldsymbol{t}$ tiene valor 1.

- Las operaciones ejecutan la línea 20 (evento $\boldsymbol{e_i^{if}}$), dado que $\boldsymbol{h}$ es distinto de $\boldsymbol{t}$ en las tres operaciones, todas las evaluaciones dan $\boldsymbol{False}$. Por lo que pasan a ejecutar las líneas 27 y 28.

- Representamos la ejecución de la línea 27 como $\boldsymbol{e_i^{CAS}}$ (vamos a abreviar $\boldsymbol{CAS}$ para referirnos a $\boldsymbol{CompareAndSet}$), en donde se ejecuta la instrucción

$$\boldsymbol{Head.CAS(h, n)},$$

  donde $\boldsymbol{n = h.next.get.(\,)}$. Nótese que es el primer nodo de la cola y tiene valor 3. Supongamos que el hilo 1 es el primero de los tres en donde este evento ocurre. Dado que la referencia no ha cambiado, el $\boldsymbol{CAS}$ es exitoso. Por lo que la cola sufre de un cambio de estado, ahora la referencia atómica $\boldsymbol{Head}$ apunta al nodo de valor 3, esto se representa por la línea punteada roja. Este punto donde el estadio de la cola cambia es precisamente el punto de linealización del conjunto de operaciones $\boldsymbol{Dequeue_i}$.

- Los eventos $\boldsymbol{e_i^{CAS}}$ restantes son fallidos, pues la referencia ya cambio. Independientemente de si fallaron o no, todas las operaciones $\boldsymbol{Dequeue_i}$ retornan en la línea 28, en este caso todas retornan el valor 3 del nodo $\boldsymbol{n}$.

Observemos que durante el proceso la referencia $\boldsymbol{Tail}$ no fue alterada, esto es principalmente porque si $\boldsymbol{t \neq h}$ nunca se ejecuta la línea 24. Sin embargo, observemos cómo se comporta el algoritmo cuando $\boldsymbol{t == h}$.

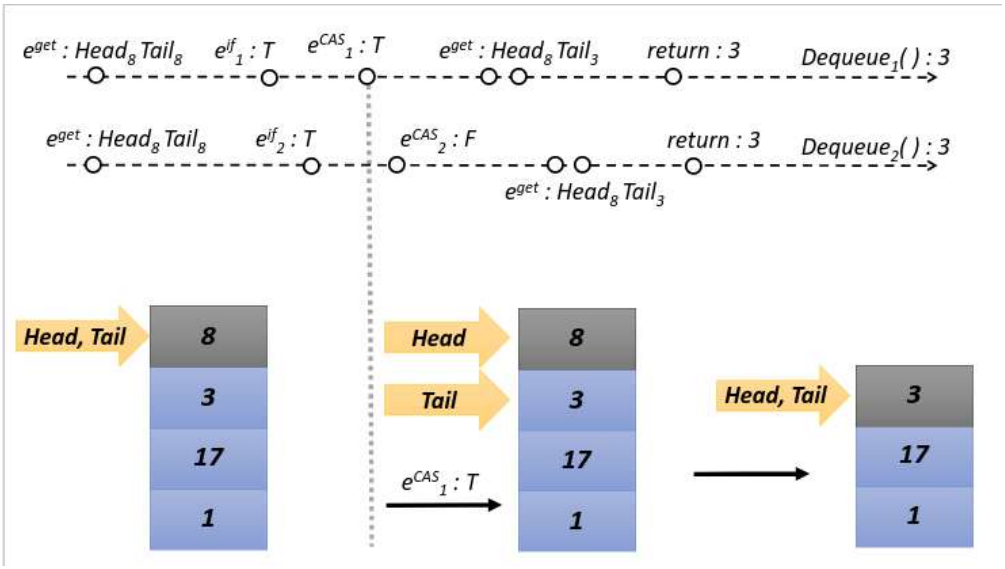


Figura 3.4: Representación de dos operaciones $\boldsymbol{Dequeue_i}$ las cuales retornan el mismo valor. En este caso mostramos una configuración en específico de la cola, en donde la referencia $\boldsymbol{Tail}$ no apunta al último nodo de la cola.

En la figura 3.4 se representan dos hilos que ejecutan concurrentemente una operación $\boldsymbol{Dequeue_i}$ cada uno. Consideraremos un caso donde ambas operaciones retornan el mismo elemento, en este caso 3. Consideremos que las operaciones ocurren de la siguiente forma:

- La cola está constituida por la lista de nodos $\mathbf{(8, 3, 17, 1)}$. El estado de la cola es $\mathbf{(3, 17, 1)}$, siendo el nodo con valor 8 el nodo centinela de la cola (recuadro representado en color gris). En este caso vamos a considerar que la referencia $\boldsymbol{Tail}$ apunta al nodo (8), es decir, al nodo centinela, este efecto podría ocurrir en situaciones donde se realizó un encolamiento, pero las operaciones $\boldsymbol{Enqueue}$ no actualizaron la referencia $\boldsymbol{Tail}$, es decir que se detuvo o continúa ejecutándose.

- Las instrucciones de las líneas 17 y 18 se representa por medio de $\boldsymbol{e^{get}}$, donde cada operación extrae al nodo $\boldsymbol{h}$ de la referencia por medio de $\boldsymbol{Head.get(\,)}$. Del mismo modo, en la línea 18 cada operación extrae el nodo en la referencia $\boldsymbol{t}$ por medio de $\boldsymbol{Tail.get(\,)}$. En ambas operaciones los nodos son iguales $\boldsymbol{h == t}$.

- Las operaciones ejecutan la línea 20 (evento $\boldsymbol{e_i^{if}}$), dado que $\boldsymbol{h}$ es igual a $\boldsymbol{t}$ en las dos operaciones, la evaluación dan $\boldsymbol{True}$. Por lo que pasan a ejecutar las líneas 21 a 25

- Observemos que la cola no está vacía, y que el nodo $\boldsymbol{n = h.next.get(\,)}$ es el nodo de valor 3. Por lo que en la línea 21 se salta a la línea 24. En caso de que $\boldsymbol{n == \epsilon}$ se retorna un elemento vacío, pues la cola estaría vacía.

- Representamos la ejecución de la línea 24 como $\boldsymbol{e_i^{CAS}}$, en donde se ejecuta la instrucción

$$\boldsymbol{Tail.CAS(t, t.next.get(\,))},$$

recordemos que $\boldsymbol{t}$ es el nodo con valor 8. Supongamos que el hilo 1 es el primero de los dos en donde este evento ocurre. Dado que la referencia no ha cambiado, el $\boldsymbol{CAS}$ es exitoso. Sin embargo, la cola no sufre de ningún cambio en su estado, sino que solo cambia la referencia $\boldsymbol{Tail}$, ahora apunta al nodo de valor 3, esto se representa por la línea punteada gris.

- El evento $\boldsymbol{e_2^{CAS}}$ es fallido, pues la referencia ya cambio. Después ambas operaciones vuelven a comenzar otra iteración en el ciclo $\boldsymbol{While}$.
- En las nuevas iteraciones las instrucciones de las líneas 17 y 18 se representa por medio de $\boldsymbol{e^{get}}$ (de nuevo), donde cada operación extrae al nodo $\boldsymbol{h}$ de la referencia por medio de $\boldsymbol{Head.get()}$. Del mismo modo, en la línea 18 cada operación extrae el nodo en la referencia $\boldsymbol{t}$ por medio de $\boldsymbol{Tail.get()}$. Pero ahora $\boldsymbol{h}$ tiene valor 8 y $\boldsymbol{t}$ tiene valor 3, son distintos.
- Ahora el proceso es análogo al caso anterior de la figura 3.3, esto debido a que $\boldsymbol{h \neq t}$. Además de que según las observaciones, cuando los nodos $\boldsymbol{h}$ y $\boldsymbol{t}$ son distintos, la referencia $\boldsymbol{Tail}$ realmente no tiene participación. Entonces ambas operaciones se linealizan en el mismo punto y el estado final de la cola es $\boldsymbol{(17, 1)}$, tal como se muestra en la figura 3.4.

Nótese que el final de las operaciones, la referencia $\boldsymbol{Tail}$ no necesariamente está al final de la cola, al menos cuando las operaciones son $\boldsymbol{Dequeue}$, pues lo que buscan con la línea 20 es observar si realmente la cola está vacía, o la referencia $\boldsymbol{Tail}$ está desactualizada.

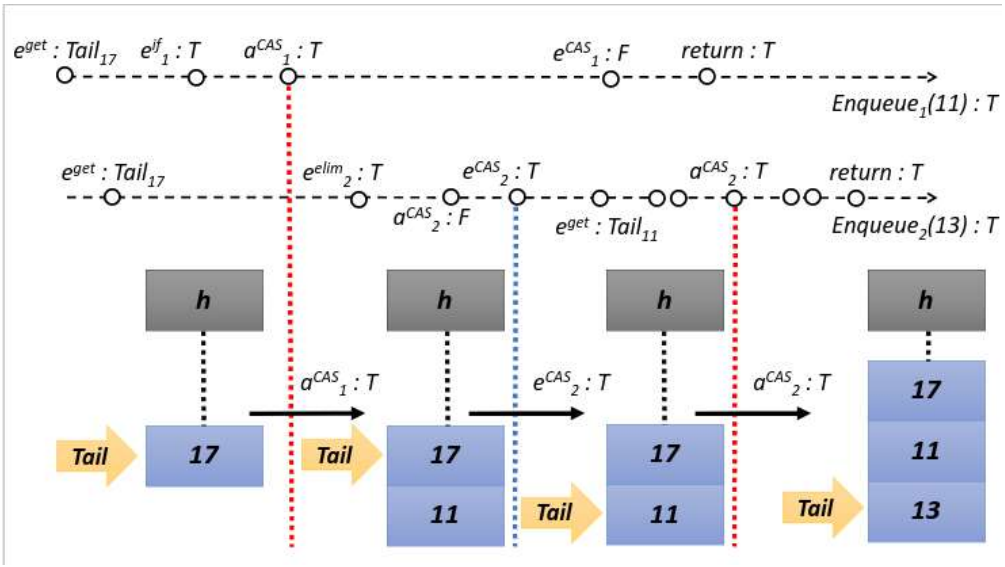


Figura 3.5: Representación de dos operaciones: $\boldsymbol{Enqueue_1(11)}$, $\boldsymbol{Enqueue_2(13)}$. Este ejemplo muestra la interacción entre ambas operaciones y del funcionamiento de la referencia $\boldsymbol{Tail}$.

En la figura 3.5 se representan dos hilos que ejecutan concurrentemente las operaciones $\boldsymbol{Enqueue_1(11)}$ y $\boldsymbol{Enqueue_2(13)}$ (estamos expresando $\boldsymbol{x}$ como en nodo de valor x). Consideremos que las operaciones ocurren de la siguiente forma:

- La cola está constituida por la lista de nodos $\boldsymbol{(h, ..., 17)}$, el único nodo con el que trabajaremos será el de valor 17. El estado de la cola es $\boldsymbol{(..., 17)}$. En este caso vamos a considerar que la referencia $\boldsymbol{Tail}$ apunta al nodo $\mathbf{17}$.

- La instrucción de la línea 3 se representa por medio de $\boldsymbol{e^{get}}$, donde cada operación extrae al nodo $\boldsymbol{t}$ de la referencia por medio de $\boldsymbol{Tail.get(\,)}$. En la línea 4 cada operación extrae el nodo siguiente de $\boldsymbol{t}$ por medio de $\boldsymbol{n = t.next.get(\,)}$. En este caso en ambas operaciones $\boldsymbol{n = \epsilon}$, pues no hay nodos después de $\mathbf{17}$.

- En la línea 5, en ambas operaciones la evaluación es $\boldsymbol{True}$ (representamos este evento por $\boldsymbol{e_i^{if}}$). Después, ambas operaciones ejecutan la línea 6.

- En la línea 6, se ejecuta:

$$\boldsymbol{t.next.CAS(n, x)},$$

  donde $\boldsymbol{x}$ es el nodo a encolar de $\boldsymbol{Enqueue(x)}$. Llamémosle a este evento $\boldsymbol{a_i^{CAS}}$, llamamos diferente a este evento para diferenciarlo del $\boldsymbol{CAS}$ de la línea 11. Consideremos que sucede primero $\boldsymbol{a_1^{CAS}}$ en la operación $\boldsymbol{Enqueue_1(11)}$. Dado que la referencia en $\boldsymbol{t.next}$ no ha cambiado (en nula), el $\boldsymbol{CAS}$ es exitoso y esto cambia el estado de la cola (representado por la línea punteada roja). Pensemos que el hilo 1 me mantiene esperando un tiempo, prácticamente la operación $\boldsymbol{Enqueue_1}$ ya término después del punto de linealización que corresponde a $\boldsymbol{a_1^{CAS}}$.

- Después sucede $\boldsymbol{a_2^{CAS}}$, el cual falla, pues la referencia $\boldsymbol{t.next}$ ya ha cambiado, por lo que esta iteración finaliza. Nótese que si la referencia $\boldsymbol{t.next}$, no fuera un objeto del tipo $\boldsymbol{AtomicReference}$, la referencia podría cambiar sin problemas, alterando el estado de la cola y perdiendo la información de la operación $\boldsymbol{Enqueue_1}$.

- Pensemos que la operación $\boldsymbol{Enqueue_2}$ inicia otra iteración. Dado que la referencia $\boldsymbol{Tail}$ no ha cambiado en esta iteración $\boldsymbol{t == 17}$; sin embargo, ahora $\boldsymbol{n == 11}$. Por lo tanto, en la línea 5 la evaluación de la operación $\boldsymbol{Enqueue_2}$ falla y es redirigida a la línea 11, para ejecutar:

$$\boldsymbol{Tail.CAS(t, n)},$$

  la cual es exitosa. Este último evento lo denotamos $\boldsymbol{e_2^{CAS}}$, aquí debemos resaltar que es un evento completamente diferente a $\boldsymbol{a_i^{CAS}}$. En el evento $\boldsymbol{e_2^{CAS}}$ la referencia $\boldsymbol{Tail}$ se altera y pasa a apuntar al nodo $\mathbf{11}$ (línea punteada azul). Termina esta iteración en el método $\boldsymbol{Enqueue_2}$.

- La operación $\boldsymbol{Enqueue_1}$ continua su ejecución donde había quedado pendiente, es decir, en la línea 7 (pues en el evento $\boldsymbol{a_1^{CAS}}$ en $\boldsymbol{CAS}$ fue exitoso), en donde se ejecuta $\boldsymbol{Tail.CAS(t, n)}$, la cual falla, empero, no importa, pues prosigue a la línea 8 para retornar un valor $\boldsymbol{True}$, indicando que operó exitosamente.

- La operación $\boldsymbol{Enqueue_2}$ inicia una nueva iteración, en donde los nodos $\boldsymbol{t}$, $\boldsymbol{n}$ toman ahora los valores $\mathbf{11}$, $\boldsymbol{\epsilon}$. Entonces la línea 5 la evaluación de correcta y la operación continua hasta la línea 6 en el evento $\boldsymbol{a_2^{CAS}}$, ahora ejecutado con éxito el método $\boldsymbol{CAS}$, por lo que se produce otro cambio de estado representado por la línea punteada roja.

Luego se ejecuta la línea 7 con éxito y retorna en la línea 8. De este modo $\boldsymbol{Enqueue_2}$ finaliza retornando un $\boldsymbol{True}$ indicando que opero con éxito. Nótese que el cambio de estado que produce $\boldsymbol{Enqueue_2}$ se encuentra en la última iteración, en el evento $\boldsymbol{a_2^{CAS}}$.

Con este último ejemplo cubrimos los escenarios más relevantes. Además, podemos resaltar que la operación $\boldsymbol{Enqueue}$ es la operación dedicada a actualizar la referencia $\boldsymbol{Tail}$ al nodo correcto, pues solo se puede encolar un nodo cuando esta referencia se encuentra en el último nodo.

Por otro lado, también podemos resaltar que las operaciones $\boldsymbol{Enqueue}$ y $\boldsymbol{Dequeue}$ solo interaccionarían cuando $\boldsymbol{h == t}$, es decir, cuando las referencias $\boldsymbol{Head}$ y $\boldsymbol{Tail}$ apuntan al mismo nodo. Aunque podría ser un ejemplo interesante, basta con saber que $\boldsymbol{Enqueue}$ actualizaría la referencia $\boldsymbol{Tail}$ hasta encontrar el último nodo, de modo que ambas operaciones presentarían un comportamiento análogo a los ejemplos anteriores, o que $\boldsymbol{Dequeue}$ retornaría un valor vacío en caso de que ambas referencias $\boldsymbol{Head}$ y $\boldsymbol{Tail}$ apuntan al centinela, mientras que $\boldsymbol{Enqueue}$ simplemente encolaría el nodo.

Capítulo 4

Pilas con semánticas relajadas: Multiplicidad

En este capítulo se presentan dos contribuciones de pilas con multiplicidad. Cada una de estas contribuciones posee una representación de los *estados* distinta, lo cual permite una dinámica distinta. A lo largo del capítulo se demuestra que ambos algoritmos son linealizables por conjuntos, que son *non-blocking* y que son, en efecto, una implementación de la pila con multiplicidad. En las secciones 4.3.5 y 4.4.5 se discuten la dinámica y las ventajas que cada uno presentan

4.1. Pilas concurrentes por conjuntos con multiplicidad

En términos generales, una pila relajada permite a las operaciones simultáneas *Pop* obtener el mismo elemento, pero todos los elementos se devuelven en orden LIFO y no se pierde ningún elemento. Formalmente, nuestra pila de conjuntos concurrentes se especifica de la siguiente manera:

Definición 4.1.1 *El conjunto de elementos que se pueden poner (Push) es* $\boldsymbol{N} = \{1, 2, ...\}$*, y el conjunto de estados Q es el conjunto infinito de cadenas* N^**. El estado inicial es la cadena vacía, indicada como* $\boldsymbol{\epsilon}$*. En el estado* $\boldsymbol{q}$*, el primer elemento en* $\boldsymbol{q}$ *representa la parte superior de la pila, que podría estar vacía si* $\boldsymbol{q}$ *es la cadena vacía. Las transiciones son las siguientes:*

- *Para* $\hat{\boldsymbol{q}} \in Q$:

$$\delta(\hat{\boldsymbol{q}}, \boldsymbol{Push(a)}) = (\hat{\boldsymbol{q}} * \boldsymbol{a}, \langle \boldsymbol{Push(a)} : \boldsymbol{True} \rangle) \tag{4.1.0.1}$$

- *Para* $\hat{\boldsymbol{q}} * \boldsymbol{a} \in Q$*, donde* $\boldsymbol{a} \in \boldsymbol{N}$ *y t procesos:*

$$\delta(\hat{\boldsymbol{q}} * \boldsymbol{a}, \{\boldsymbol{Pop_1}(\,), ..., \boldsymbol{Pop_t}(\,)\}) = (\hat{\boldsymbol{q}}, \{\langle \boldsymbol{Pop_1}(\,) : \boldsymbol{a} \rangle, ..., \langle \boldsymbol{Pop_t}(\,) : \boldsymbol{a} \rangle\}) \tag{4.1.0.2}$$

- *Para una pila vacía* $\boldsymbol{\epsilon} \in Q$*:*

$$\delta(\boldsymbol{\epsilon}, \boldsymbol{Pop}(\,)) = (\boldsymbol{\epsilon}, \langle \boldsymbol{Pop}(\,) : \epsilon \rangle) \tag{4.1.0.3}$$

Al igual que el caso de las colas concurrentes por conjuntos, cualquier algoritmo que implemente una pila concurrente por conjuntos se llega a comportar como una pila secuencial en algunos casos. Por lo que tenemos el siguiente lema [9].

Lema 4.1.1 *Sea A cualquier implementación linealizable por conjuntos de una pila concurrente por conjuntos con multiplicidad. Entonces se cumple:*

- *Todas las ejecuciones secuenciales de A son ejecuciones de la pila secuencial*
- *Todas las ejecuciones sin operaciones Pop concurrentes son linealizables con respecto a la pila secuencial*
- *Todas las ejecuciones con operaciones Pop que devuelven valores distintos son linealizables con respecto a la pila secuencial.*
- *Si las operaciones Pop devuelven el mismo valor en una ejecución, entonces son concurrentes*

4.2. Pila concurrente: versión linealizable e *ingenua*

El algoritmo para una pila concurrente, la cual es linealizable, se muestra en el pseudocódigo 5. Nótese que no posee multiplicidad, y, por lo tanto, las transiciones de pila son las usuales. El algoritmo está inspirado en el código *LockFreeStack* de [14], el cual es un algoritmo linealizable de pilas *Lock Free*.

Algorithm 5 Lineal-Stack

```
   Shared Variables : Top
 1: procedure PUSH(x)
 2:     while True do
 3:         t :=Top.get()
 4:         x.next := t
 5:         if Top.CompareAndSet(t, x) then
 6:             return True
 7:         end if
 8:     end while
 9: end procedure

10: procedure POP
11:     while True do
12:         t := Top.get()
13:         if t = ϵ then
14:             return ϵ
15:         end if
16:         if Top.CompareAndSet(t, t.next) then
17:             return t.value
18:         end if
19:     end while
20: end procedure
```

Del mismo modo que en el caso de la cola concurrente, a partir del algoritmo 5 deberíamos obtener una versión linealizable por conjuntos y cuya especificación sea la de una pila con multiplicidad, véase siguiente sección. Sin embargo, el cambio realización en la versión de cola concurrente fue mínimo, y esto permitió obtener un algoritmo de colas con multiplicidad, en el caso de la pila no es tan sencillo.

Veamos que si se intenta realizar una modificación similar a las colas se obtiene un comportamiento incorrecto cuando hay operaciones ***Pop*** y ***Push*** concurrentes. Consideremos una modificación tal que cambie la operación ***Pop*** por:

```
 1: procedure POP
 2:     while True do
 3:         t := Top.get()
 4:         if t = ϵ then
 5:             return ϵ
 6:         end if
 7:         Top.CompareAndSet(t, t.next)
 8:         return t.value
 9:     end while
10: end procedure
```

Llamemos a esta modificación, versión *ingenua* de la pila con multiplicidad. La versión *ingenua* es la misma que se hizo en la versión de colas con multiplicidad. Sin embargo, veamos el siguiente escenario:

- Una operación $\boldsymbol{Pop()}$ y una operación $\boldsymbol{Push(x)}$ que son concurrentes. Ambas obtienen la referencia al nodo $\boldsymbol{t}$ al mismo tiempo. Esto se representa en el *panel a)* de la figura 4.1.
- La operación $\boldsymbol{Push(x)}$ conecta el nodo $\boldsymbol{x}$ a la pila, mediante la línea 5. Esto se representa en el *panel b)* de la figura 4.1. Sin embargo, la pila sigue siendo la original, pues para que cambie se debe actualizar la referencia $\boldsymbol{Top}$
- La operación $\boldsymbol{Push(x)}$ actualiza la referencia $\boldsymbol{Top}$, de forma que ahora apunta hacia el nodo $\boldsymbol{x}$ (modificando la pila). Esto se representa en el *panel c)* de la figura 4.1, donde la operación $\boldsymbol{Push(x)}$ finalizo:

  $$\langle \boldsymbol{Push(x)} \ : \ \boldsymbol{True} \rangle$$

- La operación $\boldsymbol{Pop(\,)}$ intenta actualiza la referencia $\boldsymbol{Top}$; sin embargo, falla, pues la referencia ya fue actualizada en el punto anterior. Independientemente de esto se retorna el valor del nodo $\boldsymbol{t}$. Esto se representa en el *panel d)* de la figura 4.1, donde la operación $\boldsymbol{Pop(\,)}$ finalizo:

  $$\langle \boldsymbol{Pop(\,)} \ : \ \boldsymbol{t.value} \rangle$$

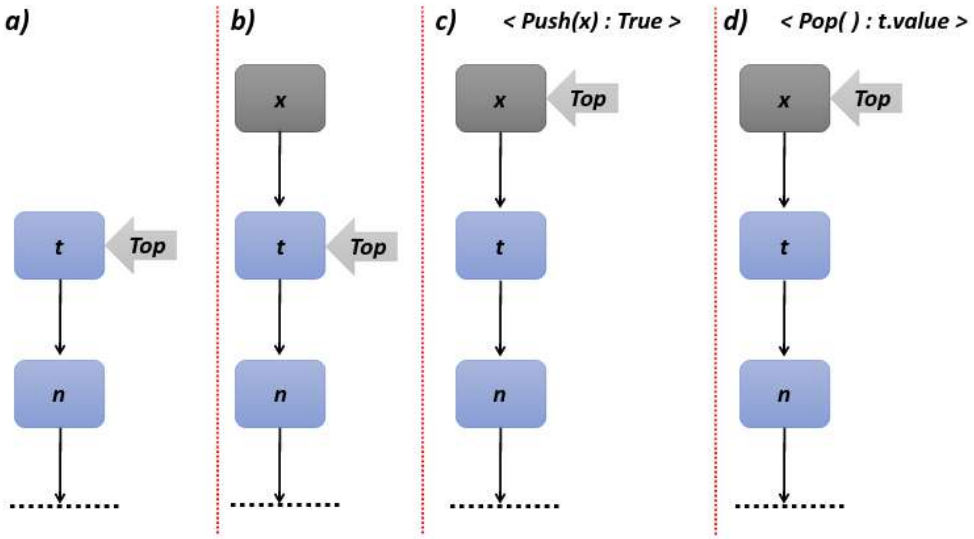


Figura 4.1: Mecanismo de falla en la versión *ingenua* de la pila con multiplicidad

Este escenario nos muestra una falla que puede parecer sutil en la versión *ingenua* de la pila con multiplicidad. Sin embargo, claramente el nodo $\boldsymbol{t}$ no fue eliminado por la operación $\boldsymbol{Pop}$ y peor aún, en este escenario en particular no se satisface la transición de estados de la

pila con multiplicidad, véase 4.1.1. Puesto que al finalizar la operación ***Pop***.

La falla en la versión *ingenua*, se debe a que la operación ***Pop*** no posee la información suficiente para saber si el cambio en ***Top*** fue hecho por una operación ***Pop*** o una operación ***Push***. En cambio, en la cola con multiplicidad, se sabe que la única operación que actúa sobre la referencia ***Head*** es ***Dequeue***, lo cual facilita proveer de multiplicidad al algoritmo. Para solucionar esta falla es necesario introducir algún mecanismo de exclusión mutua o que permita distinguir las acciones que se realizan sobre los nodos, de modo que sea posible saber si el cambio fue hecho por una operación ***Pop*** o ***Push***. Como solución a este problema se llegó a dos versiones distintas de la pila con multiplicidad, las cuales se presentan en las siguientes secciones.

4.3. Algoritmo: Set-Stack-Logic

Con el fin de describir un algoritmo linealizable por conjuntos que sea una implementación de la pila con multiplicidad, primero se describe la estructura de los nodos que forman la pila. Es usual que las estructuras de datos, tipo cola o pila, estén formadas por listas de nodos, los cuales contienen elementos de tipos primitivos. Sin embargo, debido a la concurrencia y a la multiplicidad, es necesario definir una estructura más compleja en los nodos.

Algorithm 6 Nodo(x)

Nodo
1: ***value*** : Entero de valor ***x***
2: ***next*** : ***Nodo***
3: ***elim*** : Variable booleana

Los nodos están definidos como la estructura que se muestra en el algoritmo 6. Cada nodo contiene tres atributos, uno llamado ***value***, es cual es el valor de tipo primitivo almacenado en el nodo, supondremos que estos son enteros por simplicidad y concordancia con la definición 4.1.1, el atributo ***next*** es un objeto de tipo ***Nodo*** cuya función es la de una referencia, y el atributo ***elim*** el cual es una variable booleana. El atributo ***next*** es una referencia al nodo siguiente de la lista de nodos. Por otro lado, cada nodo posee un atributo booleano ***elim*** cuya función es indicar si el nodo se encuentra o no en la pila; más adelante se formalizará esta noción, la cual se ha denominado *estado lógico*.

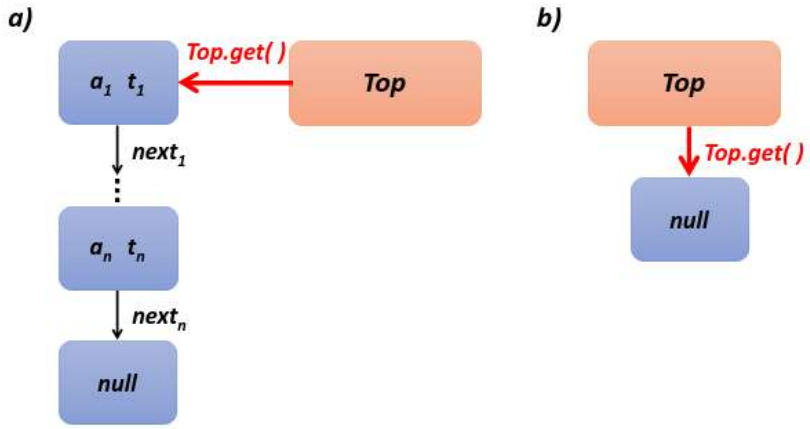


Figura 4.2: *Panel a)* Representación de una estructura de pilas, junto con la referencia atómica $\boldsymbol{Top}$. *Panel b)* Representación de una estructura de pilas sin elementos en la pila, es decir, la pila vacía $\boldsymbol{\epsilon}$.

La estructura de pila se representa como una lista de nodos, véase figura 4.2. En estas listas, cada nodo posee una referencia $\boldsymbol{next}$, la cual hace referencia al siguiente nodo de la lista, mientras que el valor de cada nodo es $\boldsymbol{value}$. La referencia del último nodo siempre apunta a un $\boldsymbol{null}$. Tengamos en cuenta el problema ABA al usar $\boldsymbol{CompareAnSet}$, para nuestro caso se puede evitar el problema ABA (y por claridad) suponiendo que cada nodo nuevo tiene una referencia distinta. Los nodos cuya variable booleana $\boldsymbol{elim}$ es $\boldsymbol{True}$ no son tomados en cuenta como parte de la pila, de modo que esta variable nos indica si el nodo forma parte de la pila. Además, el algoritmo consta de una variable de tipo $\boldsymbol{AtomicReference < Nodo >}$:

- $\boldsymbol{Top}$: Contiene la referencia al primer nodo de la lista; $\boldsymbol{Top.get( )}$ es la referencia que apunta al primer nodo. Regularmente, llamaremos cabezal a la referencia atómica $\boldsymbol{Top}$, la cual se puede decir que encapsula a toda la pila.

En la figura 4.2 *panel a)* se representa una pila del algoritmo Set-Stack-Logic 7, en donde la referencia atómica $\boldsymbol{Top}$ apunta al primer nodo de la lista. Podemos decir que la lista de nodos está conformada por los pares de valores enteros y booleanos: $[(\boldsymbol{a_1, t_1}), (\boldsymbol{a_2, t_2}), ..., (\boldsymbol{a_n, t_n})]$, y los nodos que pertenecen a la pila son aquellos cuya variable booleana es $\boldsymbol{t_i = False}$. En el *panel b)* se representa una pila vacía donde no hay ningún nodo y $\boldsymbol{Top}$ apunta a una referencia nula. Nótese que se puede representar perfectamente una pila vacía con una lista de nodos, como en el *panel a)*, en donde todos los nodos posean $\boldsymbol{t_i = True}$; el estado de la memoria de las pilas vacías no es único.

El algoritmo 7: *Set-Stack-Logic*, que se presenta a continuación, está inspirado en el algoritmo *LockFreeStack* de [14], el cual es un algoritmo linealizable de pilas *Lock Free*.

Algorithm 7 Set-Stack-Logic

```
    Shared Variables : Top
 1: procedure Push(x)
 2:     while True do
 3:         t = Top.get()
 4:         if t.elim == False then
 5:             x.next = t
 6:             if Top.CompareAndSet(t, x) then
 7:                 return True
 8:             end if
 9:         else
10:             Top.CompareAndSet(t, t.next)
11:         end if
12:     end while
13: end procedure

14: procedure Pop
15:     while True do
16:         t = Top.get()
17:         if t = ϵ then
18:             return ϵ
19:         end if
20:         if t.elim == false then
21:             t.elim = True
22:             Top.CompareAndSet(t, t.next)
23:             return t.value
24:         else
25:             Top.CompareAndSet(t, t.next)
26:         end if
27:     end while
28: end procedure
```

Después de discutir la estructura y representación de las pilas, podemos pasar a describir las operaciones del algoritmo Set-Stack-Logic. Las operaciones son las usuales (claro que ahora cumplirán multiplicidad): ***Push*** y ***Pop***.

- ***Push(x)***: Añade un nodo ***x*** en orden LIFO. La operación inicia un ciclo ***While***, en donde cada iteración es un intento para insertar el nodo ***x*** en la pila. En cada iteración se extrae el nodo ***t*** en la referencia del cabezal ***Top***. Si el nodo ***t*** ya fue lógicamente eliminado (línea 4, cuando ***t.elim*** == ***True***), entonces avanza el cabezal ***Top*** al siguiente nodo por medio de un ***CompareAndSet***, en caso de que el nodo al que apunta el cabezal no haya sido lógicamente eliminado ***t.elim*** == ***False***, conecta por medio de su referencia el nodo ***t*** al nodo ***x***. Después hace un cambio en el cabezal ***Top*** por medio del ***CompareAndSet(t, x)***, en caso de un exitoso el nodo ***x*** fue insertado correctamente y retorna en la línea 7, en caso contrario vuelve a iterar para otro intento.

- ***Pop***: Remueve un nodo en orden LIFO de la pila. La operación inicia un ciclo ***While***, en donde cada iteración es un intento para remover el primer nodo en la pila. En cada iteración se extrae el nodo t en la referencia del cabezal ***Top***. Si el nodo t es ϵ la operación retorna un valor nulo, líneas 17 y 18, indicando que la pila está vacía. En caso contrario, cuando el nodo no es nulo, evalúa la línea 20, si $elim \neq False$, el nodo está lógicamente eliminado y pasa el cabezal al siguiente nodo en la línea 25 (nótese que si la pila consta solamente de nodos eliminados lógicamente el cabezal avanza hasta llegar a la referencia nula). Cuando el nodo si está en la pila lógicamente $elim == False$ para a las líneas 21 a 23, primero elimina lógicamente el nodo en la línea 21 y después la referencia ***Top*** avanza al siguiente nodo en 22, finalmente retorna el valor del nodo eliminado $t.value$ en la línea 23.

El siguiente teorema enuncia las propiedades que el algoritmo Set-Stack-Logic cumple. Con esto se podrá apreciar y mostrar formalmente la noción del estado lógico de la pila y la correctitud del algoritmo.

Teorema 4.3.1 *El algoritmo Set-Stack-Logic es una implementación linealizable por conjuntos de la pila con multiplicidad y es non-blocking.*

En la sección 4.3.5 se muestra a detalle la dinámica del algoritmo Set-Stack-Logic en algunos escenarios particulares, mientras que las siguientes secciones demuestran el teorema 4.3.1. La demostración está divida de la siguiente forma, en la subsección 4.3.1 se demuestra que el algoritmo 7 satisface la propiedad *non-blocking*, en la subsección 4.3.2 se describe la forma en la que se construye la linealización por conjuntos, también en esta subsección se definen los *puntos de linealización*. En la subsección 4.3.3 se muestra que usando los puntos de linealización de la subsección 4.3.2 se obtiene efectivamente una linealización por conjuntos y en la subsección 4.3.4 se demuestra que la linealización por conjuntos, dada por la subsección 4.3.3, es una implementación de la pila con multiplicidad, o más formalmente, se cumplen las transiciones de estado que corresponden a la pila, dada por la definición 4.1.1.

Denotamos una operación ***Pop*** que devuelve el valor x por $\langle Pop(\,) : x\rangle$, mientras que una operación ***Push*** que inserta un valor x la denotamos por $\langle Push(x) : True\rangle$.

4.3.1. Prueba de *non-blocking*: Set-Stack-Logic

Veamos que el algoritmo es *non-blocking*. Supongamos una ejecución infinita del algoritmo Set-Stack-Logic, sea ***op*** una operación cualquiera.

- Supongamos que ***op*** es una operación ***Push***, veamos que es *non-blocking*. Para que la operación no sea completada, las evaluaciones de las líneas 4 y 6 deben fallar una cantidad infinita de veces. Si la línea 4 falla, entonces debe existir una operación ***Pop*** la cual modifico el valor de la variable ***elim*** del nodo t, ejecutando la línea 21 (ya que por defecto todos los nodos poseen $t.elim = False$). Notemos que cualquier ***Pop*** que ejecuta la línea 21 retorna en dos instrucciones, y, por lo tanto, se completa. En resumen: Si la línea 4 falla, entonces debe existir una operación ***Pop*** completada.

Por otro lado, si la línea 6 falla siempre, significa que el método ***compareAndSet*** en la línea 6 falla en toda iteración, por lo que existe una operación ***op′***, ya sea ***Pop*** o ***Push*** la cual ejecute con éxito una operación ***compareAndSet*** para cada iteración. Veamos que debe existir una operación completada, si ***op′*** se completa es trivial. Supongamos que ***op′*** no está completada para toda iteración, entonces ***compareAndSet*** debió de ejecutarse en la línea 10, en caso de ser un ***Push*** o en la línea 25, en caso de ser un ***Pop***, sin embargo, dado que el estado inicial de la pila es ϵ no pueden existir una cantidad infinita de nodos, alguna operación ***Push*** debe poner un nodo (lo cual implica que se completó) en algún momento para que la pila no se vacíe, y, por lo tanto, existe una operación completada siempre.

- Para las operaciones ***pop*** es análogo, ya que si una operación ***Pop*** no se completa, entonces la línea 17 o 20 fallan infinitas veces. Si la línea 17 falla infinitas veces, el argumento es similar a lo visto, debe existir una operación $Push$ que ponga nodos. Para la línea 20, el argumento es el mismo que en el caso de la línea 4: Si la línea 20 falla, entonces debe existir una operación ***Pop*** completada.

4.3.2. Procedimiento de linealización por Conjuntos: Set-Stack-Logic

Dado que el algoritmo es *non-blocking*, cualquier operación pendiente no bloquea al algoritmo, es decir, el algoritmo en sí mismo se mantiene completando una infinidad de operaciones por cada operación que no se completa. Por lo que se puede suponer, sin pérdida de generalidad, que cualquier ejecución finita E no tiene operaciones pendientes, pues si las tuviera estas no bloquean a las demás. Sea E una ejecución finita sin operaciones pendientes del algoritmo 7: Set-Stack-Logic.

En la linealización por conjuntos, el estado del objeto está codificado en la variable ***Top*** y en los valores de la etiqueta booleana ***elim*** de los nodos de la pila. Es decir, los elementos de la pila son los nodos cuyo valor booleano ***elim*** es ***False***, llamaremos a esto el estado lógico de la pila (más adelante se define formalmente).

Para linealizar la ejecución E, procederemos a construir una linealización S de E asignando un punto de linealización a cada operación ***op*** de E. El punto de linealización de ***op***, denotado $LinPt(\textbf{\textit{op}})$, es un evento primitivo e entre la invocación $inv(\textbf{\textit{op}})$ y la respuesta $res(\textbf{\textit{op}})$. Tengamos en cuenta que varias operaciones pueden tener el mismo punto de linealización. Por lo tanto, la linealización S también se proporciona explícitamente. Finalmente, veremos que esta linealización S induce una relación de orden total $<_S$ en las clases de concurrencia y que S cumple con la especificación de pila (4.1.1).

Sea ***op*** en E, daremos las instrucciones de linealización definiendo primero el punto de linealización $LinPt(\textbf{\textit{op}})$, en el cual ***op*** pareciera tener efecto. Recordemos que $LinPt(\textbf{\textit{op}})$ es un evento primitivo, lo cual significa que es la ejecución de alguna línea del algoritmo 7, no una operación.

1. Si $\boldsymbol{op}$ es una operación $\boldsymbol{Push}$ en E: Consideremos la última iteración de su ciclo $\boldsymbol{While}$, llamémosle $\boldsymbol{e_{CAS}}$ al paso que ejecuta la instrucción $\boldsymbol{CompareAndSet}$ de la línea 6 (este sería el único $\boldsymbol{CompareAndSet}$ exitoso). Linealizamos por conjuntos la operación $\boldsymbol{Push}$ en el punto $\boldsymbol{e_{CAS}}$, de modo que es una clase de concurrencia por sí misma.

2. Si $\boldsymbol{op}$ es una operación $\boldsymbol{Pop}$ en E, tenemos dos casos.

   a) Retorna la cadena vacía $\langle \boldsymbol{Pop}(\,) : \boldsymbol{\epsilon} \rangle$: Sea $\boldsymbol{op}$ cualquier operación $\boldsymbol{Pop}$ en E tal que retorne en la línea 18 (Esto corresponde a las operaciones $\boldsymbol{Pop}$ que retornen la cadena vacía). Consideremos la última iteración en el ciclo $\boldsymbol{while}$, llamémosle $\boldsymbol{e_{get}}$ al paso correspondiente a la línea 16 de esta iteración, este paso es donde se extrae el nodo $\boldsymbol{t}$ almacenado de la variable $\boldsymbol{Top}$ por medio del método $\boldsymbol{get}()$. La condición en la línea 17 es verdadera, pues estamos en la última iteración y la operación retorna en la línea 18, es decir, el nodo $\boldsymbol{t}$ tiene valor $\boldsymbol{\epsilon}$ en el paso $\boldsymbol{e_{get}}$. Linealizamos por conjuntos las operaciones $\boldsymbol{Pop}$ que retornan $\boldsymbol{\epsilon}$, en el punto $\boldsymbol{e_{get}}$, de modo que es una clase de concurrencia por sí misma.

   b) Retorna un elemento no nulo $\langle \boldsymbol{Pop}(\,) : \boldsymbol{x} \rangle$: Sea $\boldsymbol{op}$ cualquier operación $\boldsymbol{Pop}$ en E tal que retorne en la línea 23 (Esto corresponde a las operaciones $\boldsymbol{Pop}$ que retornen $\boldsymbol{t.value}$ tal que el valor de esta cadena es $\boldsymbol{x}$).

      Consideremos el conjunto U_x, tal que contiene todas las operaciones $\boldsymbol{Pop}$ en E que devuelven el mismo elemento $\boldsymbol{x}$. Por suposición, cada elemento se coloca en la pila como máximo una sola vez, por lo tanto, cada operación $\boldsymbol{Pop} \in U_x$ extrae al mismo nodo en su última ejecución de la línea 16. Denotemos por $\boldsymbol{t_x}$ a este nodo, cuyo valor es $\boldsymbol{x}$.

      Observemos que todas las operaciones de U_x extraen el nodo $\boldsymbol{t_x}$, lo cual corresponde a la ejecución de la línea 16. Consideremos el primer evento (de todas las operaciones $\boldsymbol{Pop}$ en U_x) que escribe en la variable booleana $\boldsymbol{elim}$ de $\boldsymbol{t}$ correspondiente a la línea 21, llamemos $\boldsymbol{e^x_{elim}}$ a este paso. Notemos que $\boldsymbol{e^x_{elim}}$ es el único paso en U_x que realmente cambia el estado de la variable $\boldsymbol{elim}$, de modo que se elimina lógicamente el nodo $\boldsymbol{t_x}$ en el paso $\boldsymbol{e^x_{elim}}$. Linealizamos todas las operaciones en U_x en el paso $\boldsymbol{e^x_{elim}}$, es decir, las operaciones en U_x forman una clase de concurrencia ubicada en $\boldsymbol{e^x_{elim}}$.

Por simplicidad, llamémosle $\boldsymbol{op}$ a una clase de concurrencia de S. Observemos que podemos clasificar las clases de concurrencia de la siguiente manera:

1. Se tiene $\boldsymbol{op}$ es una clase de concurrencia correspondiente a una operación $\boldsymbol{Push(a)}$ en E.

2. Se tiene $\boldsymbol{op}$ es una clase de concurrencia correspondiente a un conjunto operaciones $\boldsymbol{Pop(\,)}$ en E, tenemos dos subcasos:

a) La clase de concurrencia $\boldsymbol{op}$ corresponde a una sola operación $\boldsymbol{Pop(\,)}$ tal que $\langle \boldsymbol{Pop}(\,) : \boldsymbol{\epsilon} \rangle$.

b) La clase de concurrencia $\boldsymbol{op}$ corresponde a un conjunto de operaciones $U_x = \{\boldsymbol{Pop_1}(), ..., \boldsymbol{Pop_t}()\}$ tal que para toda operación $\boldsymbol{Pop_i(\,)}$ devuelve: $\langle \boldsymbol{Pop_i}(\,) : \boldsymbol{x} \rangle$ con $\boldsymbol{x}$ un elemento no nulo.

Estas clases de concurrencia que hemos definido dan la ejecución por conjuntos secuencial S. Observe que cada clase de concurrencia $\boldsymbol{op}$ de S tiene un punto de linealización por construcción $LinPt(\boldsymbol{op})$, el cual es a su vez un punto de linealización para una o varias operaciones de E.

Ahora definamos $<_S$ para las clases de concurrencia, dadas dos clases de concurrencia $\boldsymbol{op}, \boldsymbol{op'}$ en S, se define $\boldsymbol{op} <_S \boldsymbol{op'}$ si y solo si $LinPt(\boldsymbol{op}) \rightarrow LinPt(\boldsymbol{op'})$. Esta relación es de orden total sobre las clases de concurrencia de S.

4.3.3. linealización de la ejecución E

Estas clases de concurrencia que hemos definido dan la ejecución secuencial por conjuntos S. Para mostrarlo primero veamos que dada una clase de concurrencia $\boldsymbol{op}$ de S, su punto de linealización $LinPt(\boldsymbol{op})$ siempre está entre los eventos de invocación y de respuesta de su conjunto de operaciones $\boldsymbol{op_E}$, es decir que cumple

$$inv(\boldsymbol{op_E}) \rightarrow LinPt(\boldsymbol{op}) \rightarrow res(\boldsymbol{op_E}). \tag{4.3.3.1}$$

Sea $\boldsymbol{op}$ en S, y $LinPt(\boldsymbol{op})$ su punto de linealización dado por las instrucciones en la subsección 3.2.1, probemos que se cumple (4.3.3.1).

1. Si $\boldsymbol{op}$ es una clase de concurrencia correspondiente a una operación $\boldsymbol{Push(a)}$ de E.

   Dado que $LinPt(\boldsymbol{op}) = \boldsymbol{e_{CAS}}$, donde $\boldsymbol{e_{CAS}}$ es la ejecución de la instrucción $\boldsymbol{CompareAndSet}$ de la línea 6 de $\boldsymbol{Push(a)}$, de la última iteración de su ciclo $\boldsymbol{While}$. Claramente $inv(\boldsymbol{Push(a)}) \rightarrow \boldsymbol{e_{CAS}}$, pues la invocación precede a cualquier evento en las iteraciones del ciclo $\boldsymbol{While}$, y $\boldsymbol{e_{CAS}} \rightarrow res(\boldsymbol{Push(a)})$, pues la respuesta de la operación $\boldsymbol{Push(a)}$ corresponde al evento asociado a la ejecución de la línea 7.

2. Si $\boldsymbol{op}$ es una clase de concurrencia correspondiente a un conjunto operaciones $\boldsymbol{Pop(\,)}$ en E, tenemos dos subcasos:

   a) La clase de concurrencia $\boldsymbol{op}$ corresponde a una sola operación $\boldsymbol{Pop(\,)}$ tal que $\langle \boldsymbol{Pop}(\,) : \epsilon \rangle$.

      El punto de linealización es $LinPt(\boldsymbol{op}) = \boldsymbol{e_{get}}$, al igual que antes $inv(\boldsymbol{Pop(\,)}) \rightarrow \boldsymbol{e_{get}}$, pues la invocación precede a cualquier evento en las iteraciones del ciclo $\boldsymbol{While}$ y $\boldsymbol{e_{get}} \rightarrow res(\boldsymbol{Pop(\,)})$, pues la respuesta de la operación $\boldsymbol{Pop(\,)}$ corresponde al evento asociado a la ejecución de la línea 18.

b) La clase de concurrencia $\boldsymbol{op}$ corresponde a un conjunto de operaciones $U_x = \{\boldsymbol{Pop_1}(), ..., \boldsymbol{Pop_t}()\}$ tal que para toda operación $\boldsymbol{Pop_i()}$ devuelve: $\langle \boldsymbol{Pop_i}( ) : \boldsymbol{x} \rangle$ con $\boldsymbol{x}$ un elemento no nulo.

En este caso, dado que estamos linealizando un conjunto de operaciones, debemos probar que, para toda operación $\boldsymbol{Pop_i} \in U_x$ se tiene

$$inv(\boldsymbol{Pop_i}) \rightarrow LinPt(\boldsymbol{op}) \rightarrow res(\boldsymbol{Pop_i}),$$

donde U_x es el conjunto de todas las operaciones $\boldsymbol{Pop}$ en E que devuelven el mismo elemento $\boldsymbol{x}$.
Recordemos que $LinPt(\boldsymbol{op}) = \boldsymbol{e^x_{elim}}$, con $\boldsymbol{e^x_{elim}}$ el evento descrito en la instrucción 2.b) del Procedimiento 4.3.2. Dado que $\boldsymbol{e^x_{elim}}$ es el primer evento que cambia el estado de $\boldsymbol{t.elim}$ de $\boldsymbol{False}$ a $\boldsymbol{True}$, debe pasar que $inv(\boldsymbol{Pop_i}) \rightarrow \boldsymbol{e^x_{elim}}$, ya que en caso contrario $\boldsymbol{e^x_{elim}} \rightarrow inv(\boldsymbol{Pop_i})$, pero entonces la ejecución de la $\boldsymbol{t_x == False}$ resultaría fallida lo que haría que pasara a la línea 25 alterando el valor de $\boldsymbol{Top}$ y en la siguiente iteración el nodo $\boldsymbol{t}$ tendría un valor distinto $\boldsymbol{y \neq x}$, ya que el elemento $\boldsymbol{x}$ se inserta una única vez, contradiciendo el hecho de que $\boldsymbol{Pop_i}$ retorna $\boldsymbol{t_x}$.

Por lo tanto, debe pasar que $inv(\boldsymbol{Pop_i}) \rightarrow \boldsymbol{e^x_{elim}}$ para toda $\boldsymbol{Pop_i}$ en U_x.

Para ver que $\boldsymbol{e^x_{elim}} \rightarrow res(\boldsymbol{Pop_i})$ es más fácil, ya que $\boldsymbol{e^x_{elim}}$ es el primer evento que cambia el estado de $\boldsymbol{t.elim}$, por lo que para cualquier $\boldsymbol{Pop_i} \in U_x$ se tiene que $\boldsymbol{e^x_{elim}} \rightarrow \boldsymbol{e_{elim}}$ donde $\boldsymbol{e_{elim}}$ es la ejecución de la línea 21 de $\boldsymbol{Pop_i}$ en la última iteración. Pero $\boldsymbol{e_{elim}} \rightarrow res(\boldsymbol{Pop_i})$ y por transitividad de la precedencia $\boldsymbol{e^x_{elim}} \rightarrow res(\boldsymbol{Pop_i})$.
Se concluye que: Para toda operación $\boldsymbol{Pop_i}$ en U_x se tiene

$$inv(\boldsymbol{Pop_i}) \rightarrow LinPt(\boldsymbol{op}) \rightarrow res(\boldsymbol{Pop_i}),$$

donde U_x es el conjunto de todas las operaciones $\boldsymbol{Pop}$ en E que devuelven el mismo elemento $\boldsymbol{x}$.

Para concluir que S una linealización por conjuntos de E veamos lo siguiente. Por construcción, E y S tienen las mismas operaciones con las mismas respuestas. Considere las operaciones $\boldsymbol{op_1}$ y $\boldsymbol{op_2}$ de E tal que $\boldsymbol{op_1} <_E \boldsymbol{op_2}$, por lo que $res(\boldsymbol{op_1}) \rightarrow inv(\boldsymbol{op_2})$. En S, cada operación se linealiza en un paso que se encuentra entre la invocación y la respuesta de la operación. Por lo tanto, la clase de concurrencia de $\boldsymbol{op_1}$ aparece antes que la clase de concurrencia de $\boldsymbol{op_2}$ en S, pues se tiene

$$LinPt(\boldsymbol{op_1}) \rightarrow res(\boldsymbol{op_1}) \rightarrow inv(\boldsymbol{op_2}) \rightarrow LinPt(\boldsymbol{op_2})$$

por transitividad $LinPt(\boldsymbol{op_1}) \rightarrow LinPt(\boldsymbol{op_2})$, y por definición $\boldsymbol{o\tilde{p}_1} <_S \boldsymbol{o\tilde{p}_2}$, donde $\boldsymbol{o\tilde{p}_1}$, $\boldsymbol{o\tilde{p}_1}$ son las clases de concurrencia de $\boldsymbol{op_1}$ y $\boldsymbol{op_2}$ respectivamente. Concluimos que S es una linealización por conjunto de E. Ahora falta ver que S cumple con la especificación de pila.

4.3.4. Especificación de pila con multiplicidad: Algoritmo Set-Stack-Logic

En esta sección demostramos que el algoritmo Set-Stack-Logic satisface la especificación de pila con multiplicidad. Para la demostración primero definiremos el estado lógico de la pila formalmente, lo cual nos facilitara la demostración:

Definición 4.3.1 *Sea* $\hat{\boldsymbol{q}} \in \mathbb{N}^*$*, el cual es de la forma*

$$\hat{\boldsymbol{q}} = \boldsymbol{a_1 a_2 ... a_k}$$

con $\boldsymbol{a_i} \in \mathbb{N}$ *para* $i \in \{1, ..., k\}$*. Decimos que* $\hat{\boldsymbol{q}}$ *es el estado lógico que representa a la pila* $\boldsymbol{t_1}, ..., \boldsymbol{t_m}$*, si para cada elemento* $\boldsymbol{a_i}$ *existe un nodo* $\boldsymbol{t_j}$ *del algoritmo 7 tal que* $\boldsymbol{t_j.elim} = \boldsymbol{False}$ *y su valor es* $\boldsymbol{a_i}$ *con* $\boldsymbol{t_1.next} = \boldsymbol{\epsilon}$ *y* $\boldsymbol{t_1} = \boldsymbol{Top.get()}$

Observemos que el estado de la memoria está representado por los nodos $\boldsymbol{\epsilon}, \boldsymbol{t_1}, ..., \boldsymbol{t_m}$, mientras que el *estado lógico* está representado por los nodos que están lógicamente en la pila; es decir, aquellos que cumplen $\boldsymbol{t_i.elim} = \boldsymbol{False}$, los nodos que no cumplen esta condición no participan en la pila de modo lógico.

Set-Stack-Logic satisface la especificación de pila con multiplicidad.
Demostración:

Sea E una ejecución finita sin operaciones pendientes del Algoritmo 1: Set-Stack-Logic. Demostremos que cumple con la especificación de pila 4.1.1. Donde el estado de la pila está dado por el estado lógico 4.3.1.

Sea S la linealización por conjuntos de E obtenida por el procedimiento de linealización 4.3.2. Tenemos que las clases de concurrencia están ordenadas como

$$\boldsymbol{op_1} <_S \boldsymbol{op_2} <_S ... <_S \boldsymbol{op_M}$$

Donde op_i es la clase de concurrencia de alguna operación de E y M es el número de clases de concurrencia definidas por $<_S$. Sea E_m el prefijo de E con las operaciones $\{\boldsymbol{op_1}...\boldsymbol{op_m}\}$ y S_m su correspondiente linealización por conjuntos, la cual de hecho induce el orden $\boldsymbol{op_1} <_S ... <_S \boldsymbol{op_m}$.

Demostremos que S_m satisface la especificación de pila 4.1.1 para toda $m \leq M$, cuyo estado está dado por T_m, por inducción sobre m.

El caso base es trivial, pues $E = \emptyset$, por lo que $S = \emptyset$, y el estado de la pila es el mismo después de ejecutar S o E.

Como hipótesis de inducción supongamos que la ejecución secuencial por conjuntos S_m de E_m cumple con la especificación de pila, cuyo estado está dado por el estado lógico T_m.

Para el paso inductivo demostremos que se cumple para E_{m+1} cuya linealización por conjuntos es S_{m+1}, dada por el procedimiento 4.3.2. Induce un orden total en las clases de equivalencia y podemos ordenarlas como:

$$\boldsymbol{op_0} <_S \boldsymbol{op_1} <_S ... <_S \boldsymbol{op_{m+1}}$$

Por hipótesis de inducción el prefijo E_m cumple con lo requerido, basta mostrar que al ejecutar la operación $\boldsymbol{op_{m+1}}$ la transición de estados corresponde a la especificación de pila 4.1.1.

Sea $\boldsymbol{Top}$ la variable compartida antes de ejecutar $\boldsymbol{op_{m+1}}$, observemos que por hipótesis de inducción el estado de $\boldsymbol{Top}$ es el mismo que se obtiene de ejecutar S_m, por lo que se ejecutan las clases de concurrencia en orden: $\boldsymbol{op_0} <_S \boldsymbol{op_1} <_S ... <_S \boldsymbol{op_m}$.

Sea $\hat{\boldsymbol{q}} \in \mathbb{N}^*$ el estado lógico de la pila después de ejecutar la clase de concurrencia $\boldsymbol{op_m}$, el cual representa el estado lógico de la pila $\boldsymbol{t_1}, ..., \boldsymbol{t_m}$.

1. Si $\boldsymbol{op_{m+1}}$ es una clase de concurrencia correspondiente a una operación $\boldsymbol{Push(a)}$ en E_m: El punto de linealización de $\boldsymbol{op}$ es $LinPt(\boldsymbol{Push(a)}) = \boldsymbol{e_{CAS}}$, el cual corresponde a la ejecución de la línea 6 con un $\boldsymbol{CompareAndSet}$ exitoso en la última iteración. Este punto también es el único evento en $\boldsymbol{Push}$ en el cual el estado lógico de la pila cambia, de $\hat{\boldsymbol{q}}$ a $\hat{\boldsymbol{q}} * \boldsymbol{a}$.

   Basta ver que si el estado en $\boldsymbol{op_m}$ es $\hat{\boldsymbol{q}}$, con $\hat{\boldsymbol{q}} \in \mathbb{N}^*$, el estado después de ejecutar $\boldsymbol{Push(a)}$ es $\hat{\boldsymbol{q}} * \boldsymbol{a}$. De este modo se cumpliría formalmente la transición:

   $$\delta(\hat{\boldsymbol{q}}, \boldsymbol{Push(a)}) = (\hat{\boldsymbol{q}} * \boldsymbol{a}, \langle \boldsymbol{Push(a)} : \boldsymbol{True} \rangle).$$

   Dado que $\boldsymbol{op_m} <_S \boldsymbol{op_{m+1}}$, podemos asegurar que el estado de la pila en el evento $\boldsymbol{e_{CAS}}$ cambian de $\hat{\boldsymbol{q}}$ a $\hat{\boldsymbol{q}} * \boldsymbol{a}$, pues si el estado de la pila es diferente de $\hat{\boldsymbol{q}}$, entonces algún evento lo modifico entre $\boldsymbol{op_m}$ y $\boldsymbol{op_{m+1}}$.

   Podemos asegurar que no existe ningún evento que modifique el estado lógico de la pila $\hat{\boldsymbol{q}}$, pues los únicos eventos que cambian el estado lógico son las ejecuciones de la línea 6 o 21 (en caso de un $\boldsymbol{CompareAndSet}$ exitoso, el cual añada un nodo, para la línea 6) pero justo estos eventos corresponderían a puntos de linealización de alguna operación $\boldsymbol{op'}$, de modo que $\boldsymbol{op_m} <_S \boldsymbol{op'} <_S \boldsymbol{op_{m+1}}$, contradiciendo que el orden que induce la linealización por conjuntos S_{m+1}.

   Por lo tanto, el estado lógico después del evento $LinPt(\boldsymbol{op_m})$ y antes del evento $\boldsymbol{e_{CAS}}$ es $\hat{\boldsymbol{q}}$, pero en $\boldsymbol{e_{CAS}}$ el método $\boldsymbol{CompareAndSet}$ es exitoso, cambiando el estado lógico de la pila a $\hat{\boldsymbol{q}} * \boldsymbol{a}$, ya que inserta el nodo $\boldsymbol{t}$ con valor $\boldsymbol{a}$ tal que $\boldsymbol{t.elim} = \boldsymbol{False}$. Ahora el estado en memoria es $\boldsymbol{t_1}, ..., \boldsymbol{t_m}, \boldsymbol{t}$ cuyo estado lógico es $\hat{\boldsymbol{q}} * \boldsymbol{a}$, pues el orden lo fija la lista $\boldsymbol{t_1}, ..., \boldsymbol{t_m}, \boldsymbol{t}$. Entonces, para la operación $\boldsymbol{op_{m+1}}$ se cumple la transición:

   $$\delta(\hat{\boldsymbol{q}}, \boldsymbol{Push(a)}) = (\hat{\boldsymbol{q}} * \boldsymbol{a}, \langle \boldsymbol{Push(a)} : \boldsymbol{True} \rangle)$$

   Por lo que concluimos que S_{m+1} produce una ejecución secuencial por conjuntos de pila válida que es consistente con (4.1.1), donde estado de la pila está representado por el estado lógico codificado en $\hat{\boldsymbol{q}} * \boldsymbol{a}$, representado en la variable $\boldsymbol{Top}$

2. Si $\boldsymbol{op_{m+1}}$ es una clase de concurrencia correspondiente a un conjunto operaciones $\boldsymbol{Pop(\,)}$ en E_m, tenemos dos casos:

   a) La clase de concurrencia $\boldsymbol{op_{m+1}}$ corresponde a una sola operación $\boldsymbol{Pop(\,)}$ tal que $\langle \boldsymbol{Pop}(\,) : \epsilon \rangle$.

   El punto de linealización es $\boldsymbol{e_{get}}$, correspondiente a la línea 16 de la última iteración. Además, la operación $\boldsymbol{Pop(\,)}$ retorna en la línea 18. Por suposición

   $$LinPt(\boldsymbol{op_m}) \rightarrow \boldsymbol{e_{get}}.$$

   Por lo que el estado lógico de la pila después de ejecutar $LinPt(\boldsymbol{op_m})$ debe de ser $\boldsymbol{\epsilon}$, pues en caso contrario en el evento $\boldsymbol{e_{get}}$ se obtendría un nodo $\boldsymbol{t} \neq \boldsymbol{\epsilon}$, y no se ejecutaría la línea 4 satisfactoriamente, contradiciendo $\langle \boldsymbol{Pop}(\,) : \epsilon \rangle$.
   Además, observemos que las operaciones $\boldsymbol{Pop}$ que devuelven una cadena nula: $\langle \boldsymbol{Pop}(\,) : \boldsymbol{\epsilon} \rangle$, en realidad no alteran el estado lógico de la pila, pues nunca ejecutan la línea 21. Por lo que se ejecute o no $\boldsymbol{op}$ el estado lógico permanece invariante, y es $\boldsymbol{\epsilon}$. Dicho en otras palabras, para la operación $\boldsymbol{op_{m+1}}$ se cumple la transición:

   $$\delta(\boldsymbol{\epsilon}, \boldsymbol{Pop}(\,)) = (\boldsymbol{\epsilon}, \langle \boldsymbol{Pop}(\,) : \boldsymbol{\epsilon} \rangle).$$

   Concluimos que S_{m+1} produce una ejecución secuencial por conjuntos de pila válida que es consistente con (4.1.1), donde estado de la pila está representado por el estado lógico codificado en $\boldsymbol{\epsilon}$, representado en la variable $\boldsymbol{Top}$

   b) La clase de concurrencia $\boldsymbol{op_{m+1}}$ corresponde a un conjunto de operaciones $U_x = \{\boldsymbol{Pop_1}(\,), ..., \boldsymbol{Pop_t}(\,)\}$ tal que para toda operación $\boldsymbol{Pop_i(\,)}$ devuelve: $\langle \boldsymbol{Pop_i}(\,) : \boldsymbol{x} \rangle$ con $\boldsymbol{x}$ un elemento no nulo. Recordemos que el conjunto U_x es tal que contiene todas las operaciones $\boldsymbol{Pop}$ en E_{m+1} que devuelven el mismo elemento $\boldsymbol{x}$.

   El punto de linealización es $LinPt(\boldsymbol{op}) = \boldsymbol{e^x_{elim}}$, el cual corresponde a la primera ejecución de la línea 21 de todas las operaciones $\boldsymbol{Pop_x}$ en U_x, este punto también es el único evento en el cual el estado lógico de la pila cambia (cambiando $\boldsymbol{t_x.elim = False}$ a $\boldsymbol{t_x.elim = True}$ en el evento $\boldsymbol{e^x_{elim}}$ para $\boldsymbol{t_x}$ el nodo que contiene el valor de $\boldsymbol{x}$).

   Basta ver que si el estado en $\boldsymbol{op_m}$ es $\boldsymbol{\hat{q} = \hat{p} * a}$, con $\boldsymbol{\hat{p}} \in \mathbb{N}^*$ y $\boldsymbol{a} \in \mathbb{N}$, el estado después de ejecutar $\boldsymbol{op_{m+1}}$ es $\boldsymbol{\hat{p}}$. Dado que $\boldsymbol{op_m} <_S \boldsymbol{op_{m+1}}$, podemos asegurar que el estado de la pila en el evento $\boldsymbol{e^x_{elim}}$ cambian de $\boldsymbol{\hat{p} * a}$ a $\boldsymbol{\hat{p}}$, pues si el estado la pila cambia entre $\boldsymbol{op_m}$ y $\boldsymbol{op_{m+1}}$, entonces algún evento lo cambio. Podemos asegurar que no existe ningún evento que cambie el estado lógico de la pila $\boldsymbol{\hat{p} * a}$, pues los únicos eventos que cambian el estado lógico son las ejecuciones de la línea 6 o 21 (en caso de un $\boldsymbol{CompareAndSet}$ exitoso, el cual añada un nodo, para la línea 6) pero justo estos eventos corresponderían a puntos de linealización de alguna operación cuya, con clase de concurrencia $\boldsymbol{op'}$, de modo que $\boldsymbol{op_m} <_S \boldsymbol{op'} <_S \boldsymbol{op_{m+1}}$,

contradiciendo el orden dado por S_{m+1}.

Por lo tanto, antes del evento $\boldsymbol{e^{x}_{elim}}$ el estado lógico es $\hat{\boldsymbol{p}} * \boldsymbol{a}$. Además, podemos asegurar que para el nodo $\boldsymbol{t_m} = \boldsymbol{Top.get()}$ se tiene que $\boldsymbol{t_m.elim} == \boldsymbol{False}$, por la línea 20. Entonces, dado que es el último nodo, por definición, su valor es $\boldsymbol{a}$. Entonces, $\boldsymbol{e^{x}_{elim}}$ cambia $\boldsymbol{t_m.elim} = \boldsymbol{True}$ con $\boldsymbol{t_m}$, cambiando el estado lógico de la pila a $\hat{\boldsymbol{p}}$, pues ahora no existe nodo alguno cuyo valor sea $\boldsymbol{a}$ y se encuentre lógicamente en la pila. También se tiene que $\boldsymbol{x} = \boldsymbol{a}$, pues la línea 23 devuelve $\boldsymbol{t_a.value}$ el cual es $\boldsymbol{a}$. Dicho en otras palabras, para la operación $\boldsymbol{op_{m+1}}$ se cumple la transición:

$$\delta(\hat{\boldsymbol{p}} * \boldsymbol{a}, \{\boldsymbol{Pop_1}(\,), ..., \boldsymbol{Pop_t}(\,)\}) = (\hat{\boldsymbol{p}}, \{\langle \boldsymbol{Pop_1}(\,) : \boldsymbol{a}\rangle, ..., \langle \boldsymbol{Pop_t}(\,) : \boldsymbol{a}\rangle\}).$$

Por lo que concluimos que S_{m+1} produce una ejecución secuencial de pila válida que es consistente con (4.1.1), donde estado de la pila está representado por el estado lógico codificado en $\hat{\boldsymbol{q}}$, representado en la variable $\boldsymbol{Top}$.

Por lo que concluimos que S_m produce una ejecución secuencial de pila válida que es consistente con (4.1.1) para toda $m \leq n$. Por consiguiente es válido para E y S.

4.3.5. Dinámica del algoritmo Set-Stack-Logic

Esta sección está destinada para explicar el funcionamiento del algoritmo Set-Stack-Logic bajo algunos posibles escenarios, así como mostrar la transición de estados de forma explícita. Principalmente, para dar ejemplos de como se puede llegar a comportar la memoria del sistema durante una ejecución concurrente. Cada hilo se representará por medio de una línea negra punteada, los eventos se representan como $\boldsymbol{e}$ más algunos índices.

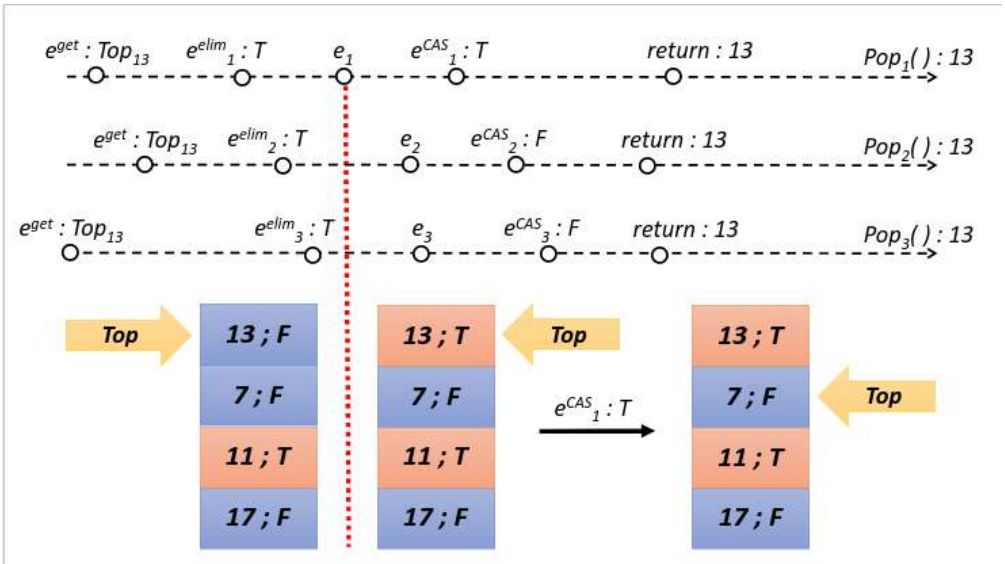


Figura 4.3: Representación de un conjunto dado de operaciones $\boldsymbol{Pop_i}$ que retornan todas el mismo valor. En este caso, el conjunto es la clase de concurrencia de las operaciones.

Para los siguientes dos ejemplos consideremos tres hilos que se ejecutan concurrentemente. El primer ejemplo está representado en la figura 4.3, donde cada nodo ejecuta una operación $\boldsymbol{Pop_i}$, además consideraremos el caso donde todas las operaciones retornan el mismo elemento. Como se puede apreciar durante las demostraciones, la iteración más relevante en las operaciones es siempre la última, en este ejemplo consideremos que estamos en la última iteración de cada $\boldsymbol{Pop_i}$. Consideremos que las operaciones ocurren de la siguiente forma:

- La pila está constituida por la siguiente lista de nodos:

$$[(\mathbf{17}, \boldsymbol{False}), (\mathbf{11}, \boldsymbol{True}), (\mathbf{7}, \boldsymbol{False}), (\mathbf{13}, \boldsymbol{False})]\,.$$

  El estado lógico de la pila es $\mathbf{(17, 7, 13)}$, siendo el nodo con valor 13 el primero en la pila.

- La instrucción de la línea 3 se representa por medio de $\boldsymbol{e^{get}}$, donde cada operación extra al nodo de la referencia por medio de $\boldsymbol{Top.get(\,)}$. Dado que en esta iteración todas las operaciones $\boldsymbol{Pop_i}$ finalizaron en valor de dicho nodo es 13, y su valor booleano $\boldsymbol{False}$.

- Todas las operaciones encuentran un nodo distinto de $\boldsymbol{\epsilon}$, línea 17 a 19.

- Cada operación ejecuta la línea 20 (evento representado por $\boldsymbol{e_i^{elim}}$) y dado que el primer nodo es $\mathbf{(13, }\boldsymbol{False}\mathbf{)}$ todas las operaciones pasan a la línea 21.

- Llamémosle $\boldsymbol{e_i}$ a la ejecución de la línea 21. Consideremos, sin pérdida de generalidad, que la operación $\boldsymbol{Pop_1}$ es la primera en ejecutar dicha línea. Por lo tanto, $\boldsymbol{e_1}$ es el punto de linealización de este conjunto de operaciones y en este evento es donde se produce un cambio en el estado lógico de la pila, esto se representa por la línea punteada roja.

- Después del evento $\boldsymbol{e_1}$ el estado lógico de la pila cambio a $\mathbf{(17, 7)}$, pero veamos que la referencia $\boldsymbol{Top}$ aún apunta al nodo $\mathbf{(13, }\boldsymbol{True}\mathbf{)}$. Todas las operaciones ejecutan ahora la línea 22 (evento $\boldsymbol{e_i^{CAS}}$), el primer $\boldsymbol{CompareAndSet}$ es el que cambia la referencia de $\boldsymbol{Top}$ (no tiene mucha relevancia cuál es el primero). Finalmente, todas las operaciones retornan 13.

- En este último evento $\boldsymbol{e_i^{CAS}}$ la referencia atómica $\boldsymbol{Top}$ cambia, de modo que la memoria real de la pila cambia tal como se representa por medio de la flecha negra en la figura 4.3; sin embargo, el verdadero cambio del estado lógico se encuentra en la línea roja punteada, el cual es también el punto de linealización por conjuntos.

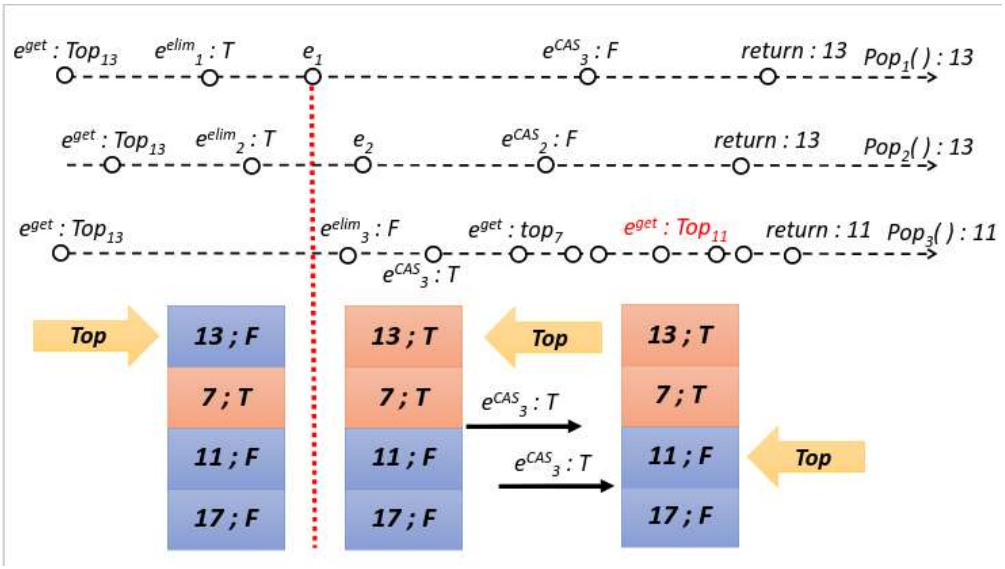


Figura 4.4: Representación de un conjunto dado de operaciones $\boldsymbol{Pop_i}$ que retornan diferentes valores. En este caso se tendrían dos clases de concurrencia.

Como siguiente ejemplo consideremos un conjunto de operaciones $\boldsymbol{Pop}_i$ que ocurren de la siguiente manera:

- La pila está constituida por la lista de nodos

$$[(\mathbf{17}, \boldsymbol{False}), (\mathbf{11}, \boldsymbol{False}), (\mathbf{7}, \boldsymbol{True}), (\mathbf{13}, \boldsymbol{False})]\,.$$

  El estado lógico de la pila es $(\mathbf{17}, \mathbf{11}, \mathbf{13})$.

- La instrucción de la línea 3 se representa por medio de $\boldsymbol{e^{get}}$. Todas las operaciones extraen la misma referencia al nodo $(\mathbf{13}, \boldsymbol{False})$.

- Usando la misma notación donde, la línea 20 es el evento representado por $\boldsymbol{e_i^{elim}}$ y $\boldsymbol{e_i}$ a la ejecución de la línea 21. En este caso los eventos $\boldsymbol{e_i^{elim}}$ las operaciones $\boldsymbol{Pop_1}$ y $\boldsymbol{Pop_2}$ ocurren antes del evento $\boldsymbol{e_1}$, en donde el estado lógico de la pila cambia. Pero el evento $\boldsymbol{e_3^{elim}}$ ocurre después, de modo que el $\boldsymbol{if}$ no permite ejecutar la línea 21 para $\boldsymbol{Pop_3}$.

- Las operaciones $\boldsymbol{Pop_1}$ y $\boldsymbol{Pop_2}$ se completan de una forma similar al caso anterior. En el evento $\boldsymbol{e_3^{CAS}}$ la referencia $\boldsymbol{Top}$ cambia de apuntar al nodo $(\mathbf{13}, \boldsymbol{True})$ al nodo $(\mathbf{7}, \boldsymbol{True})$ (instrucción de la línea 25). Observemos que el nodo $(\mathbf{7}, \boldsymbol{True})$ no está lógicamente en la pila.

- La operación $\boldsymbol{Pop_3}$ vuelve a iterar para otro intento de remover el nodo al inicio de la pila. Dado que el nodo $(\mathbf{7}, \boldsymbol{True})$ no está lógicamente en la instrucción 20, falla y pasa otra vez a la línea 25, donde se realiza otro $\boldsymbol{e_3^{CAS}}$ para cambiar la referencia de $\boldsymbol{Top}$ del nodo $(\mathbf{7}, \boldsymbol{True})$ al nodo $(\mathbf{11}, \boldsymbol{False})$.

- La operación $\boldsymbol{Pop_3}$ vuelve a iterar, extrayendo el nodo $\boldsymbol{e^{get}}$ del cabezal y después se ejecutan las líneas 20 a 23 retornando el valor 11.

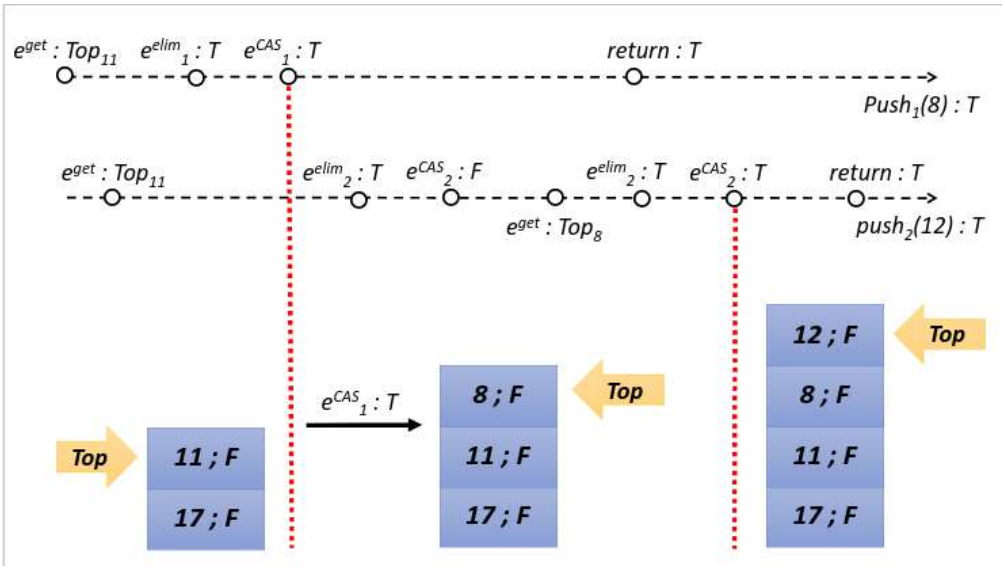


Figura 4.5: Representación de la interacción entre dos operaciones concurrentes: $\boldsymbol{Push_1((8, False)}$ y $\boldsymbol{Push_2((12, False))}$.

Consideremos ahora un conjunto de operaciones $\boldsymbol{Push_1((8, False)}$ y $\boldsymbol{Push_2((12, False))}$, por simplicidad ejemplificaremos solo un par de operaciones, las cuales ocurren de la siguiente manera:

- La pila está constituida por la siguiente lista de nodos:

  $$[\boldsymbol{(17, False)}, \boldsymbol{(11, False)}] .$$

  El estado lógico de la pila es $\boldsymbol{(17, 11)}$.

- La instrucción de la línea 3 se representa por medio de $\boldsymbol{e^{get}}$. Ambas operaciones extraen la misma referencia al nodo $\boldsymbol{(11, False)}$.

- En la línea 5 la operación $\boldsymbol{Push_1}$ conecta el nuevo nodo $\boldsymbol{(8, False)}$ al primero nodo, en el evento y después en la línea 6 se realiza un $\boldsymbol{CompareAndSet}$ el cual cambia la referencia hacia el nuevo nodo, esto en el evento $\boldsymbol{e_1^{CAS}}$. En este último evento el estado lógico y la memoria real cambian, tan como se muestra en la figura. Después de este cambio de estado lógico, la operación $\boldsymbol{Push_2}$ ejecuta un $\boldsymbol{CompareAndSet}$ en el evento $\boldsymbol{e_2^{CAS}}$; sin embargo, falla, pues la referencia ya cambio.

- La operación $\boldsymbol{Push_2}$ vuelve a iterar siguiendo un comportamiento similar al de la operación $\boldsymbol{Push_1}$, insertando el nuevo nodo $\boldsymbol{(12, False)}$ en el evento $\boldsymbol{e_2^{CAS}}$.

- El estado final de la pila después de la ejecución de las operaciones $\boldsymbol{Push_1}$ y $\boldsymbol{Push_2}$ es

  $$[\boldsymbol{(17, False)}, \boldsymbol{(11, False)}, \boldsymbol{(8, False)}, \boldsymbol{(12, False)}] .$$

  Observemos que en este caso existen dos puntos de linealización distintos, cada uno en la última iteración de su respectiva operación $\boldsymbol{Push_i}$, la cual es una clase de concurrencia por sí misma.

Observemos que una vez que ocurre el punto de linealización de la primera operación, la segunda operación inevitablemente fallará en el ***CompareAndSet***; es decir, en el evento $\boldsymbol{e_2^{CAS}}$ se retorna ***False***. Esto es debido a que cambia el estado lógico de la pila en el punto de linealizacion. Esta propiedad nos permite linealizar por conjunto las operaciones ***Push*** en clases de concurrencia de un único elemento. Mientras que las operaciones ***Pop*** poseen un punto de linealización que permite que un conjunto dado de operaciones puedan ser linealizadas en un mismo punto de linealización.

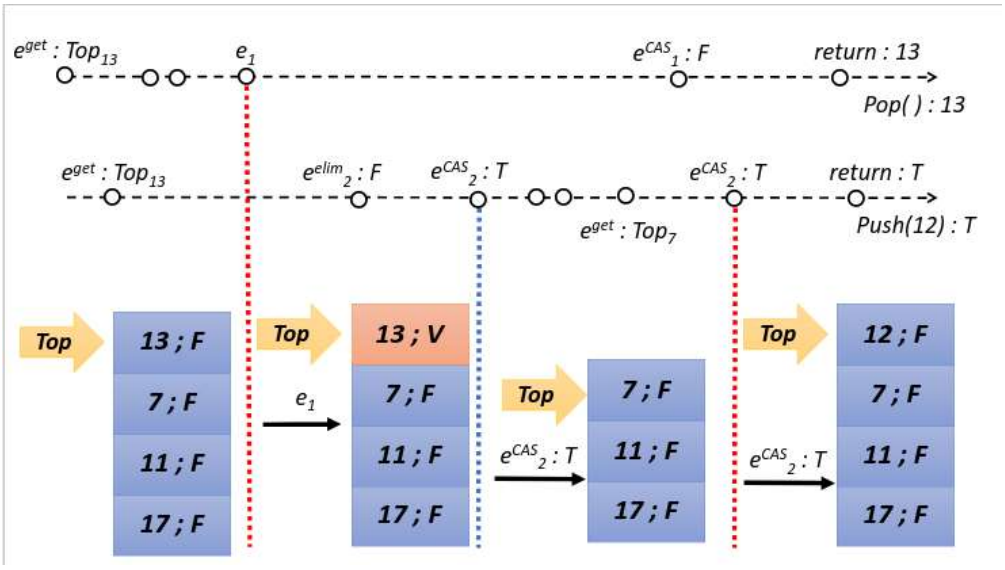


Figura 4.6: Representación de la interacción entre dos operaciones concurrentes: $\boldsymbol{Push((12, False))}$ y $\boldsymbol{Pop(\,)}$.

Como último ejemplo consideremos una operación $\boldsymbol{Push((12, False))}$ y otra ***Pop***, con el siguiente comportamiento:

- La pila está constituida por la lista de nodos

$$[\boldsymbol{(17, False)}, \boldsymbol{(11, False)}, \boldsymbol{(7, False)}, \boldsymbol{(13, False)}]\,.$$

  El estado lógico de la pila es $\boldsymbol{(17, 11, 7, 13)}$.

- La instrucción de la línea 3 y 16 se representa por medio de $\boldsymbol{e^{get}}$. Ambas operaciones extraen la misma referencia al nodo $\boldsymbol{(13, False)}$.

- La operación ***Pop*** ejecuta primero la instrucción de la línea 21, en donde ocurre el punto de linealización $\boldsymbol{e_1}$. La pila cambia su estado lógico como se muestra en la primera transición, pero la referencia continúa apuntando al nodo $\boldsymbol{(13, True)}$.

- La operación $\boldsymbol{Push((12, False))}$ ejecuta la línea 4 (fallando, pues $\boldsymbol{(13, True).elim == True}$), y después la línea 10, donde hace un cambio de referencia de ***Top*** al nodo $\boldsymbol{(7, False)}$.

- La ejecución del ***CompareAndSet*** en la línea 22 falla, independientemente de esto la operación se completa retornando el valor del nodo en la línea 23.

- En la operación ***Push*** inicia otra iteración y obtiene de ***Top*** la referencia al nodo $\mathbf{(7, \mathit{False})}$. Después de ejecutar los debidos eventos llega a la línea 6 y en el evento $\boldsymbol{e_2^{CAS}}$ ejecuta un ***CompareAndSet*** exitoso, por lo que cambia la referencia al nuevo nodo añadido a la pila, el estado de la pila cambia en $\boldsymbol{e_2^{CAS}}$ y termina en $\mathbf{(17, 11, 7, 12)}$.

Observemos que no hay problema si en lugar de una operación ***Pop*** consideramos un conjunto de operaciones: ya sea de 3 operaciones, como en el ejemplo de la figura 4.3, o un conjunto arbitrario de operaciones pero todas pertenecientes a la misma clase de concurrencia. Esto es gracias a que la única operación que cambia el estado lógico en la pila es aquella que contiene el punto de linealización del conjunto de operaciones $\boldsymbol{Pop_i}$ (siempre y cuando retornen el mismo elemento).

4.4. Algoritmo: Set-Stack-Physic

El segundo algoritmo que surge como una solución a la linealización por conjuntos de la pila con multiplicidad requiere el uso de una estructura más compleja para los nodos. Este algoritmo, denominado Set-Stack-Physic, recurre al uso de listas doblemente ligadas, por lo que cada nodo posee dos referencias, una la cual apunta al nodo *inferior* y otra que apunta al nodo *superior*.

Algorithm 8 Nodo-Set-Stack-Physic

Nodo
1: ***value*** : Entero de valor $\boldsymbol{x}$
2: ***upper*** : $\boldsymbol{AtomicMarkableReference < Nodo >}$
3: ***lower*** : ***Nodo***

La estructura de los nodos se compone de tres atributos, uno llamado ***value***, el cual es el valor de tipo primitivo almacenado en el nodo, al igual que antes es un entero. El atributo ***lower*** el cual es un objeto de tipo ***Nodo*** que funciona como una referencia al nodo siguiente o nodo inferior (está de hecho es lo mismo que la referencia ***next*** del algoritmo Set-Stack-Logic). Por último, un objeto de tipo $\boldsymbol{AtomicMarkableReference < Nodo >}$ el cual es una referencia atómica con *marca*, esta referencia sirve para apuntar al nodo que se encuentra *arriba*, la cual es una propiedad que carece el algoritmo Set-Stack-Logic, donde los nodos solamente pueden apuntar a la referencia inferior.

Antes de continuar cabe mencionar el funcionamiento de $\boldsymbol{AtomicMarkableReference}$, en especial el concepto de *marca*. Las variables de tipo $\boldsymbol{AtomicMarkableReference}$ proporcionan una referencia a un nodo que se puede leer y escribir atómicamente. Los principales métodos de cualquier objeto de este tipo son:

- ***get***( ): Retorna el valor o nodo actual de la referencia.

- $\boldsymbol{CompareAndSet(nodo_e, nodo_f, marca_e, marca_f)}$: Actualiza atómicamente el nodo de la referencia y la marca, usando los valores dados $\boldsymbol{nodo_f}$ y $\boldsymbol{marca_f}$, si la referencia actual es igual a la referencia esperada $\boldsymbol{nodo_e}$ y la marca actual es igual a la marca esperada $\boldsymbol{marca_e}$.
- $\boldsymbol{attemptMark(nodo_e, marca_f)}$: Actualiza atómicamente el valor de la marca, usando el valor dado $\boldsymbol{marca_f}$, si la referencia actual es igual a la referencia esperada $\boldsymbol{nodo_e}$.

Algorithm 9 Set-Stack-Physic

Shared Variables : Top

```
1: procedure PUSH(x)
2:     while True do
3:         t := Top.get( )
4:         u := t.upper.get( )
5:         l := t.lower
6:         x.lower = t
7:         if u == ϵ then
8:             if t.upper.isMarked( ) then
9:                 if t.upper.CompareAndSet(u, x, True, True) then
10:                    Top.CompareAndSet(t, t.upper)
11:                    return True
12:                end if
13:            else
14:                l.upper.CompareAndSet(t, ϵ, True, True)
15:                Top.CompareAndSet(t, l)
16:            end if
17:        else
18:            Top.CompareAndSet(t, u)
19:        end if
20:    end while
21: end procedure

22: procedure POP
23:    while True do
24:        t := Top.get( )
25:        u := t.upper.get( )
26:        l := t.lower.get( )
27:        if u == ϵ then
28:            if l == ϵ then
29:                return ϵ
30:            end if
31:            if t.upper.isMarked( ) then
32:                if t.upper.attemptMark(ϵ,False) then
33:                    l.upper.CompareAndSet(t, ϵ, True, True)
34:                    Top.CompareAndSet(t, l)
35:                    return t.value
36:                end if
37:            else
38:                l.upper.CompareAndSet(t, ϵ, True, True)
39:                Top.CompareAndSet(t, l)
40:            end if
41:        else
42:            Top.CompareAndSet(t, u)
43:        end if
44:    end while
45: end procedure
```

El algoritmo Set-Stack-Physic se muestra en 9, este hace uso de los nodos 8 así como de los métodos de ***AtomicMarkableReference***. Las operaciones son:

- ***Push*(*x*)**: Añade un nodo ***x*** en orden LIFO. La operación inicia un ciclo ***While*** en donde cada iteración es un intento para insertar el nodo ***x*** en la pila. En cada iteración extrae el nodo ***t*** de la referencia ***Top*** y a los nodos ***u***, ***l***, en las respectivas referencias ***upper*** y ***lower***, del nodo ***t*** (líneas 3 a 5). Conecta la referencia ***x.lower*** al nodo ***t***, línea 6. Si existe un nodo ***u*** *arriba* del nodo ***t***, línea 7, entonces actualiza la referencia ***Top*** una posición superior, línea 18. En caso contrario, revisa si la referencia ***t.upper*** tiene marca ***True***, si no la tiene procede a *eliminar* el nodo marcado con ***False*** (líneas 14 y 15). Si tiene marca ***True*** intenta un ***CompareAndSet*** para actualizar la referencia ***upper***, línea 9. En caso de ejecutar con éxito ***CompareAndSet*** se añade el nodo ***x*** a la pila, actualiza la referencia ***Top*** y retornar, líneas 10 y 11. En caso contrario algo debió haber cambiado y vuelve a intentar en otra iteración.

- ***Pop*( )**: Remueve un nodo en orden LIFO de la pila. La operación inicia un ciclo ***While***, en donde cada iteración es un intento para remover el primer nodo en la pila. En cada iteración extrae el nodo ***t*** de la referencia ***Top*** y a los nodos ***u***, ***l*** en las respectivas referencias ***upper***, ***lower*** del nodo ***t***. Si existe un nodo ***u*** *arriba* del nodo ***t***, línea 27, entonces actualiza la referencia ***Top*** una posición superior, línea 42. En caso contrario, revisa si la pila está vacía, es decir, $\boldsymbol{l} == \boldsymbol{\epsilon}$, si esta vacía retorna $\boldsymbol{\epsilon}$, líneas 28 y 29. En caso contrario, continúa y revisa si la referencia ***t.upper*** tiene marca ***True***, si no la tiene procede a *eliminar* el nodo marcado con ***False***, línea 38 y 39. Si tiene marca ***True*** procede a intentar marcar la referencia. En la línea 32 ejecuta ***t.upper.attemptMark***, en caso de éxito, lo cual equivale a eliminar el nodo, se intenta eliminar el nodo marcado y retorna su valor, línea 33 a 35.

El algoritmo Set-Stack-Physic 9 satisface las mismas propiedades que el algoritmo Set-Stack-Logic 4, pero en este caso la especificación de pila implementa una definición distinta del estado de la pila. Más formalmente, lo enunciamos en el siguiente teorema:

Teorema 4.4.1 *El algoritmo Set-Stack-Physic es una implementación linealizable por conjuntos de la pila con multiplicidad y es non-blocking.*

En la sección 4.4.5 se muestra a detalle la dinámica del algoritmo Set-Stack-Physic en algunos escenarios particulares, mientras que las siguientes secciones demuestran el teorema 4.4.1. La demostración esta divida de la siguiente forma, en la sección 4.4.1 se demuestra que el algoritmo 9 satisface *non-blocking*, en la sección 4.4.2 se describe la forma en la que se construye la linealización por conjuntos, también en esta sección se dan los *puntos de linealización*, en la sección 4.4.3 se muestra que usando los puntos de linealización de la sección 4.4.2 se obtiene efectivamente una linealización por conjuntos y finalmente en la sección 4.4.4 se demuestra que la linealización por conjuntos, dada por la sección 4.4.3, es una implementación de la pila con multiplicidad, o más formalmente, se cumplen las transiciones de estado que corresponden a la pila, dada por la definición 4.1.1.

4.4.1. Prueba de *non-blocking*: Set-Stack-Physic

Veamos que el algoritmo 9 es *non-blocking*. Supongamos una ejecución infinita del algoritmo Set-Stack-Physic, sea ***op*** una operación cualquiera (únicamente puede ser un ***Push*** o ***Pop***), veamos que es *non-blocking*.

- Supongamos que ***op*** es una operación ***Push*** que no se completa (no retorna respuesta). Notemos primero que, para que la operación no sea completada, se deben realizar infinitas iteraciones sobre el ciclo ***while***. En esta operación hay una serie de condiciones que provocan que el ciclo itere sin detenerse, veamos cada una a detalle:

  1. La condición en la línea 7 falla en cada iteración. Para que esto suceda en cada iteración debe de cumplirse que $\boldsymbol{u} \neq \boldsymbol{\epsilon}$ en cada iteración. Lo cual significa que la pila nunca acaba, pues no hay nodo cuya referencia ***upper***, la cual hace referencia al nodo superior, sea $\boldsymbol{\epsilon}$. Esto solo es posible si es estado inicial de la pila es una pila con infinitos nodos o en cada iteración existe una operación ***Push*** efectiva la cual logra colocar un nodo con un ***CAS*** exitoso en la línea 9, lo cual lleva a su finalización. Es decir, en cada iteración una operación ***Push*** se completa.
  2. Cuando $\boldsymbol{u} == \boldsymbol{\epsilon}$ en la línea 7 se tienen dos posibilidades: ***t.upper*** está marcada o no. En caso de que no esté marcada ***t.upper.isMarked***( ) retorna ***False***, pero todo nodo ***n*** posee por default un valor ***True*** para la marca de ***n.upper*** y el único evento que altera el valor de la marca a ***False*** es un ***CAS*** exitoso en la línea 32 para alguna operación ***Pop***, lo cual lleva a su finalización. Por lo que en este caso siempre existe una operación ***Pop*** que se completa.

     En otro caso, ***t.upper.isMarked***( ) retorna ***True***, lo cual lleva a la línea 9. En caso de que el método ***CAS*** sea exitoso en la línea 9 lleva a la finalización de la operación, por lo tanto, debe fallar. Sin embargo, y como se ha mostrado en los algoritmos pasados, el método ***CAS*** solo falla cuando la referencia o la marca son diferentes a las esperadas. Ya sabemos que la marca cambia de ***True*** a ***False*** por una operación ***Pop*** que se completa. Supongamos que lo que cambio fue la referencia, es decir ***u*** cambio por la acción de alguna otra operación y las únicas instrucciones que cambian la referencia ***upper*** de un nodo son las líneas 9, 14, 33 y 38. En el caso de las líneas 14, 33 y 38 es inmediato ver que corresponden a la finalización de alguna operación ***Pop***, pues ocurren después de marcar o encontrar un nodo marcado. El caso de la línea 9 es correspondiente a la finalización de una operación ***Push***. Por siempre existe una operación ***Pop*** que se completa.

- Supongamos que ***op*** es una operación ***Pop*** que no se completa. La deducción es ahora más directa, pues menos condiciones para que la operación itere sin detenerse. La primera es cuando la condición en la línea 27 falla en cada iteración, lo cual es análogo a la falla en la línea 7, por lo tanto, existe una operación ***Push*** que se completa.

  La segunda condición es cuando encuentra ***t.upper*** marcada y el método ***attemptMark*** falla, pues si no falla lleva a la finalización de la operación. Sabemos que cuando método ***attemptMark*** falla es debido a que la referencia esperada cambio, por lo que ocurrió algún evento que altero la referencia ***upper*** del nodo y por lo discutido en el punto

anterior sabemos que es debido a la finalización de alguna operación ***Push***. Por último, se tiene el caso en donde se encuentra ***t.upper*** con valor ***False*** en la línea 31, es análogo al segundo punto del análisis para la operación ***Push***, de modo que se concluye que existe una operación que se completa.

Por lo tanto, se completan infinitas operaciones y se concluye que el algoritmo 9 es *non-blocking*.

4.4.2. Procedimiento de linealización por Conjuntos: Set-Stack-Physic

Sea ***op*** en E, daremos la instrucción de linealización definiendo primero el punto de linealización $LinPt(\boldsymbol{op})$, en el cual un evento en el cual ***op*** tiene efecto. Al igual que en los pasados algoritmos, siempre consideraremos los eventos de la última iteración del ciclo ***while***, por lo que omitiremos mencionarlo.

1. Si ***op*** es una operación ***Push*** en E: Observemos que si una operación ***Push*** retorna en la línea 11, en la línea 9 el método ***CompareAndSet*** se ejecutó con éxito. Sea $\boldsymbol{e_{CAS}}$ el evento donde la instrucción ***CompareAndSet***, de la línea 9, es ejecutada con éxito. Linealizamos por conjuntos la operación ***Push*** en el evento $\boldsymbol{e_{CAS}}$, de modo es una clase de concurrencia por sí misma.

2. Si ***op*** es una operación ***Pop*** en E, tenemos dos casos.

   a) Retorna la cadena vacía $\langle \boldsymbol{Pop}(\,) : \boldsymbol{\epsilon} \rangle$: Sea ***op*** cualquier operación ***Pop*** en E tal que retorne $\boldsymbol{\epsilon}$ en la línea 29.

   Llamémosle $\boldsymbol{e_{get}}$ al paso correspondiente a la línea 24 de esta iteración, este paso es donde se extrae el nodo $\boldsymbol{t}$ almacenado de la variable ***Top*** por medio del método ***get***(). Linealizamos por conjuntos las operaciones ***Pop*** que retornan $\boldsymbol{\epsilon}$, en el punto $\boldsymbol{e_{get}}$, de modo es una clase de concurrencia por sí misma.

   b) Retorna un elemento no nulo $\langle \boldsymbol{Pop}(\,) : \boldsymbol{x} \rangle$: Sea ***op*** cualquier operación ***Pop*** en E tal que retorne un valor no nulo $\boldsymbol{x}$.

   Sea U_x el conjunto que contiene todas las operaciones ***Pop*** en E que devuelven el mismo elemento $\boldsymbol{x}$. Por suposición, cada elemento se coloca en la pila como máximo una única vez, por lo tanto, cada operación $\boldsymbol{Pop} \in U_x$ extrae al mismo nodo en la ejecución de la línea 24, denotemos por $\boldsymbol{t_x}$ a este nodo, cuyo valor es $\boldsymbol{t_x.value} = \boldsymbol{x}$.

   Dado que cada operación ***Pop*** en U_x retorna de la línea 35, entonces el evento donde se ejecuta el método ***attemptMark*** de la línea 32 es exitoso. Llamémosle $\boldsymbol{e^x_{MARK}}$ al primero de estos eventos en U_x, donde el método ***attemptMark*** de la línea 32 es exitoso. Linealizamos todas las operaciones en U_x en el paso $\boldsymbol{e^x_{MARK}}$, es decir, las operaciones forman una clase de concurrencia ubicada en $\boldsymbol{e_{MARK}}$.

4.4.3. Linealización de la ejecución E

Estas clases de concurrencia que hemos definido dan la ejecución secuencial S. Análogo al algoritmo Set-Stack-Logical, basta ver que dada una clase de concurrencia $\boldsymbol{op}$ de S, su punto de linealización $LinPt(\boldsymbol{op})$ siempre está entre los eventos de invocación y de respuesta de su conjunto de operaciones $\boldsymbol{op_E}$, es decir que cumple

$$inv(\boldsymbol{op_E}) \rightarrow LinPt(\boldsymbol{op}) \rightarrow res(\boldsymbol{op_E}). \tag{4.4.3.1}$$

Sea $\boldsymbol{op}$ en S, y $LinPt(\boldsymbol{op})$ su punto de linealización dado por las instrucciones en la sección 4.4.2, probemos que se cumple (4.4.3.1), la deducción es análoga a la del algoritmo Set-Stack-Logical.

1. Si $\boldsymbol{op}$ es una clase de concurrencia correspondiente a una operación $\boldsymbol{Push(a)}$ de E, tenemos que su punto de linealización es $LinPt(\boldsymbol{op}) = \boldsymbol{e_{CAS}}$, donde $\boldsymbol{e_{CAS}}$ es la ejecución exitosa de la instrucción $\boldsymbol{CompareAndSet}$ de la línea 9. De loa algoritmos anteriores es fácil ver que se cumple (4.4.3.1).

2. Si $\boldsymbol{op}$ es una clase de concurrencia correspondiente a un conjunto operaciones $\boldsymbol{Pop(\,)}$ en E, tenemos dos subcasos:

   a) La clase de concurrencia $\boldsymbol{op}$ corresponde a una sola operación $\boldsymbol{Pop(\,)}$ tal que $\langle \boldsymbol{Pop}(\,) : \boldsymbol{\epsilon} \rangle$. El punto de linealización es $LinPt(\boldsymbol{op}) = \boldsymbol{e_{get}}$, al igual que antes es inmediato que se cumple (4.4.3.1).

   b) La clase de concurrencia $\boldsymbol{op}$ corresponde a un conjunto de operaciones $U_x = \{\boldsymbol{Pop_1}(), ..., \boldsymbol{Pop_t}()\}$ tal que para toda operación $\boldsymbol{Pop_i()}$ devuelve: $\langle \boldsymbol{Pop_i}(\,) : \boldsymbol{x} \rangle$ con $\boldsymbol{x}$ un elemento no nulo.

   En este caso, dado que estamos linealizando un conjunto de operaciones, debemos probar que, para toda operación $\boldsymbol{Pop_i} \in U_x$ se tiene

   $$inv(\boldsymbol{Pop_i}) \rightarrow LinPt(\boldsymbol{op}) \rightarrow res(\boldsymbol{Pop_i}),$$

   donde U_x es el conjunto de todas las operaciones $\boldsymbol{Pop}$ en E que devuelven el mismo elemento $\boldsymbol{x}$. Recordemos que $LinPt(\boldsymbol{op}) = \boldsymbol{e^x_{MARK}}$.

   Veamos que $inv(\boldsymbol{Pop_i}) \rightarrow \boldsymbol{e^x_{MARK}}$, ya que en caso contrario se da una contradicción. Supongamos que $\boldsymbol{e^x_{MARK}} \rightarrow inv(\boldsymbol{Pop_0})$ para alguna operación en U_x. Recordemos que $\boldsymbol{e^x_{MARK}}$ es el primer evento que ejecuta la línea 31 con éxito, es decir, es el primer evento en cambiar la marca de $\boldsymbol{t_x.upper}$ a $\boldsymbol{False}$.
   Como el elemento $\boldsymbol{x}$ se inserta una única vez y $\boldsymbol{e^x_{MARK}} \rightarrow inv(\boldsymbol{Pop_0})$, en la iteración de $\boldsymbol{Pop_0}$ donde la línea 24 extrae al nodo $\boldsymbol{t_x}$ este posee una marca en la referencia $\boldsymbol{upper}$ la cual es $\boldsymbol{False}$, por lo que falla en la evaluación de la línea 31 y no es posible que retorne $\boldsymbol{t_x.value}$. Lo cual contradice que $\boldsymbol{Pop_0}$ este en U_x. Por lo tanto, debe pasar que $inv(\boldsymbol{Pop_i}) \rightarrow \boldsymbol{e^x_{MARK}}$ para toda $\boldsymbol{Pop_i}$ en U_x.
   Para ver que $\boldsymbol{e^x_{MARK}} \rightarrow res(\boldsymbol{Pop_i})$ es más directo. Dado que $\boldsymbol{e^x_{MARK}}$ es el primero de todos los eventos de U_x donde el método $\boldsymbol{attemptMark}$ es exitoso, uno por

cada operación $\boldsymbol{Pop_i}$, y cada uno de estos eventos antecede a $res(\boldsymbol{Pop_i})$. Por la propiedad de transitividad es inmediato que $\boldsymbol{e^x_{MARK}} \rightarrow res(\boldsymbol{Pop_i})$.
Se concluye que: Para toda operación $\boldsymbol{Pop_i}$ en U_x se tiene

$$inv(\boldsymbol{Pop_i}) \rightarrow LinPt(\boldsymbol{op}) \rightarrow res(\boldsymbol{Pop_i}),$$

donde U_x es el conjunto de todas las operaciones $\boldsymbol{Pop}$ en E que devuelven el mismo elemento $\boldsymbol{x}$.

Usando los mismos argumentos que se dan al final de la subsección 4.3.3, podemos concluir que S una linealización por conjuntos de E. Basta probar que S cumple con la especificación de pila.

4.4.4. Implementación de pila con multiplicidad: Algoritmo Set-Stack-Physic

En esta sección demostramos que el algoritmo 9 satisface la especificación de pila con multiplicidad. Como se comentó anteriormente, a diferencia del algoritmo Set-Stack-Logic el estado de la pila, en ese caso, no está dado por el estado lógico, sino por todos los nodos *almacenados* en la memoria. Para simplificar la discusión podemos definir el *estado físico* de la pila en el algoritmo Set-Stack-Physic como sigue:

Definición 4.4.1 *Sea $\hat{\boldsymbol{q}} \in \mathbb{N}^*$, el cual es de la forma*

$$\hat{\boldsymbol{q}} = \boldsymbol{a_1 a_2 ... a_k}$$

con $\boldsymbol{a_i} \in \mathbb{N}$ para $i \in \{1, ..., k\}$. Decimos que $\hat{\boldsymbol{q}}$ representa el estado físico de la pila si se cumple:

- *Para cada elemento $\boldsymbol{a_i}$ existe un nodo $\boldsymbol{t_i}$ del algoritmo 8, tal que $\boldsymbol{t_i.value} = \boldsymbol{a_i}$ y la marca de $\boldsymbol{t_i.upper}$ es $\boldsymbol{True}$.*
- *Los nodos forman una lista doblemente enlazada, es decir, para cada nodo $\boldsymbol{t_i}$ se cumple que $\boldsymbol{t_i.upper.get()} = \boldsymbol{t_{i+1}}$ y $\boldsymbol{t_i.lower} = \boldsymbol{t_{i-1}}$. En particular, para los nodos extremos: $\boldsymbol{t_k.upper.get()} = \boldsymbol{\epsilon}$ y $\boldsymbol{t_1.lower} = \boldsymbol{c}$, con $\boldsymbol{c}$ el nodo centinela.*
- *La referencia $\boldsymbol{Top}$ apunta a algún nodo de $\{\boldsymbol{c}, \boldsymbol{t_1}, ..., \boldsymbol{t_k}\}$ o, a un nodo $\boldsymbol{t}$ tal que $\boldsymbol{t.lower} = \boldsymbol{t_k}$, $\boldsymbol{t.upper.get()} = \boldsymbol{\epsilon}$ y la marca de $\boldsymbol{t.upper}$ es $\boldsymbol{False}$.*

Observemos que el estado de la pila está representado por los nodos en memoria, más dos posibles nodos, por un lado, el nodo centinela está siempre presente, mientras que el otro representa un estado donde ha sido ejecutado una operación $\boldsymbol{Pop}$ y el nodo del cabezal ha sido eliminado. Esto es muy similar al estado lógico, pero en este algoritmo este nodo eliminado, cuya marca de $\boldsymbol{t.upper}$ es $\boldsymbol{False}$, es eliminado antes de cualquier operación tenga efecto, a diferencia del algoritmo Set-Stack-Logic.

Primero veamos lo siguiente, lo cual funcionara como el principal argumento de la demostración, y resultara más claro si se enuncia como un lema:

Lema 4.4.2 *Dada la definición del estado de la pila 4.4.1, las únicas instrucciones que cambian el estado de la pila son:*

- *El evento de la línea 9 donde*

$$\boldsymbol{t.upper.CompareAndSet(u, x, True, True)}$$

  es exitoso. Cambia el estado de $\hat{\boldsymbol{p}} = \boldsymbol{b_1}...\boldsymbol{b_k}$ *al estado* $\hat{\boldsymbol{p}} * \boldsymbol{x}$.

- *El evento de la línea 32 donde*

$$\boldsymbol{t.upper.attemptMark(u, False)}$$

  es exitoso. Cambia el estado de $\hat{\boldsymbol{p}} = \boldsymbol{b_1}...\boldsymbol{b_k}$ *al estado* $\boldsymbol{b_1}...\boldsymbol{b_{k-1}}$

Demostración:

Veamos cada una a detalle, sea $\hat{\boldsymbol{p}} = \boldsymbol{b_1}...\boldsymbol{b_k}$ un estado cualesquiera:

- Antes de probar que las líneas 9 y 32 cambian el estado físico de la pila, veamos que, en caso de ejecutar los métodos con éxito, estos siempre actúan sobre el nodo $\boldsymbol{t_k}$ asociado a $\boldsymbol{b_k}$, es decir, en el nodo cabezal. Basta con ver que, para que se ejecuten la línea 9 la línea 32 se debió de haber cumplido que $\boldsymbol{u} == \boldsymbol{\epsilon}$, y en ambos métodos ***CompareAndSet*** y ***attemptMark*** se espera que la referencia $\boldsymbol{u}$ no haya cambiado. En el escenario donde los métodos son exitosos es debido a que la referencia no fue modificada. Por lo tanto, si algunos de estos dos métodos actuara exitosamente sobre un nodo $\boldsymbol{t_i}$ asociado a $\boldsymbol{b_i}$, tal que $i \neq k$, se tendría que $\boldsymbol{t_i.upper}$ es la referencia al nodo $\boldsymbol{t_{i+1}} \neq \boldsymbol{\epsilon}$, lo cual entra en contradicción con el hecho de que el método haya sido exitoso, pues la referencia esperada es $\boldsymbol{u} == \boldsymbol{\epsilon}$. Por lo que ***CompareAndSet*** y ***attemptMark*** solo pueden ser exitosos al actuar sobre $\boldsymbol{t_k}$. Esto es en efecto esencial para que las transiciones en los estados físicos de la pila correspondan a los de una pila con multiplicidad.

- La línea 9 ejecuta un

$$\boldsymbol{t_k.upper.CompareAndSet(u, x, True, True)},$$

  exitoso, supongamos que el estado antes de ejecutar la línea 9 es $\hat{\boldsymbol{p}}$. El nodo $\boldsymbol{t_k}$ se ve modificado tal que ahora $\boldsymbol{t_k.upper.get(\,) = x}$ y debido a que se ejecutó la línea 6 se cumple $\boldsymbol{x.lower} = \boldsymbol{t_k}$. Por lo tanto, el evento donde se ejecuta la línea 9 con un ***CompareAndSet*** exitoso, el estado de la pila cambia de $\hat{\boldsymbol{p}}$ al estado $\hat{\boldsymbol{p}} * \boldsymbol{x}$, además el evento es atómico. Obsérvese que la marca de $\boldsymbol{t_k.upper}$ no cambia en la línea 9 y que en caso de que ***CompareAndSet*** no sea exitoso el estado de la pila no cambia.

- La línea 32 ejecuta el método

$$\boldsymbol{t_k.upper.attemptMark(\epsilon, True)}$$

  exitoso, supongamos que el estado antes de ejecutar la línea 32 es $\hat{\boldsymbol{p}}$. Si el método ***attemptMark*** es exitoso significa que la marca de $\boldsymbol{t_k.upper}$ es ***False*** y que

$\boldsymbol{t_k.upper, get(\,) = \epsilon}$ (pues es la referencia esperada). Por lo tanto, el evento donde se ejecuta la línea 32 con un $\boldsymbol{attemptMark}$ exitoso, el estado de la pila cambia de $\boldsymbol{b_1, ..., b_k}$ a $\boldsymbol{b_1, ..., b_{k-1}}$, pues la marca de $\boldsymbol{t_k.upper}$ es $\boldsymbol{False}$ (esto lo elimina de la pila). Claramente en caso de fallas no modifica el estado físico de la pila.

Un hecho muy importante sobre la ejecución de un $\boldsymbol{attemptMark}$ exitoso es que puede haber un conjunto de operaciones $\boldsymbol{Pop}$ que ejecuten

$$\boldsymbol{t_k.upper.attemptMark(\epsilon, True)}$$

de forma exitosa, pero solo el primero produce el cambio de estado de la pila y este es el evento asociado a los puntos de linealización.

Con este análisis podemos decir que, en efecto, los eventos de la línea 9 con un $\boldsymbol{CompareAndSet}$ exitoso y la línea 32 con un $\boldsymbol{attemptMark}$ exitoso, cambian el estado de la pila. Ahora veamos que son las *únicas* instrucciones del algoritmo que cambian el estado físico:

- La línea 10 cambiar la referencia de la variable $\boldsymbol{Top}$, empero, la línea 10 se ejecuta solo si el $\boldsymbol{CompareAndSet}$ de la línea 9 es exitoso, lo cual implica que el estado al ejecutar la línea 10 es, por ejemplo, $\boldsymbol{\hat{p} * x}$ y la variable $\boldsymbol{Top}$ apunta a $\boldsymbol{t_k}$, por lo que la línea 10 únicamente cambia la referencia de $\boldsymbol{Top}$ sin alterar realmente el estado físico de la pila.

- Supongamos que el estado está representado por la lista de nodos $\boldsymbol{t_1, ..., t_k}$. Las líneas 14 y 15 se ejecutan cuando la marca de $\boldsymbol{t.upper}$ es $\boldsymbol{False}$, por lo que ese nodo no se encuentra en la pila, es decir, $\boldsymbol{t \neq t_i}$ para $i \in \{1, ..., k\}$. La línea 14 cambia la referencia $\boldsymbol{upper}$ de $\boldsymbol{l}$ y la línea 15 cambia la referencia de $\boldsymbol{Top}$, sin embargo, el estado en ambas ejecuciones es $\boldsymbol{t_1, ..., t_k}$. El mismo argumento es válido para los pares de instrucciones 33, 34 y 38, 39.

- La línea 18, se ejecuta únicamente si $\boldsymbol{u \neq \epsilon}$, es decir que $\boldsymbol{t.upper}$ apunta a un nodo no nulo.

  Observemos lo siguiente, la única instrucción que cambia la referencia $\boldsymbol{t.upper}$ a algún nodo $\boldsymbol{x \neq \epsilon}$ en la línea 9, todas las demás líneas que ejecutan $\boldsymbol{t.upper.CompareAndSet}$ cambian la referencia a $\boldsymbol{\epsilon}$, aún más, ya sabemos que estas líneas no cambian el estado físico de la pila. Entonces, $\boldsymbol{u}$ debe pertenecer a la pila, pues fue insertado por medio de algún evento que ejecuto la línea 9 exitosamente. Por lo que el estado físico de la pila al ejecutar la línea 18 debe estar representado por una lista de nodos: $\boldsymbol{t_0 ... t, u, ..., t_k}$. Con lo que se concluye que la línea 18 cambia la referencia $\boldsymbol{t}$ de $\boldsymbol{Top}$ a $\boldsymbol{u}$, pero sin alterar el estado físico de la pila. Lo mismo ocurre en la línea 42.

Por otro lado, las instrucciones restantes únicamente son extracciones de una variable por métodos $\boldsymbol{get}$ o evaluaciones. Por lo que queda demostrado el lema 4.4.2.

Por último, demostremos que Set-Stack-Physic satisface la especificación de pila con multiplicidad, con lo cual quedara demostrado el Teorema 4.4.1.

Set-Stack-Physic satisface la especificación de pila con multiplicidad. Demostración:

Sea E una ejecución finita sin operaciones pendientes del Algoritmo 9: Set-Stack-Physic. Demostremos que cumple con la especificación de pila 4.1.1. Donde el estado de la pila está dado por el estado físico 4.4.1.

Sea S la linealización por conjuntos de E obtenida por el procedimiento de linealización 4.4.2. La demostración es similar al caso del algoritmo Set-Stack-Physic. Tenemos que las clases de concurrencia están ordenadas como

$$\boldsymbol{op_1} <_S \boldsymbol{op_2} <_S ... <_S \boldsymbol{op_m}$$

Donde $\boldsymbol{op_i}$ es la clase de concurrencia de alguna operación de E y n es el número de clases de concurrencia definidas por $<_S$. Sea E_k el prefijo de E con las operaciones $\{\boldsymbol{op_1}...\boldsymbol{op_k}\}$ y S_k su correspondiente linealización por conjuntos, la cual de hecho induce el orden $\boldsymbol{op_1} <_S ... <_S \boldsymbol{op_k}$.

Demostremos, por inducción sobre m, que S_k satisface la especificación de pila 4.1.1 para toda $k \leq m$. De los algoritmos anteriores sabemos que basta mostrar que al ejecutar la operación $\boldsymbol{op_{m+1}}$ la transición de estados corresponde a la especificación de pila 4.1.1.

Además, por el 4.4.2 se puede asegurar que los únicos eventos que cambian el estado físico de la pila son forzosamente la ejecución exitosa de la línea 9 o 32. Sea $\hat{\boldsymbol{q}} \in \mathbb{N}^*$ el estado lógico de la pila después de ejecutar la clase de concurrencia $\boldsymbol{op_m}$.

Observemos que el estado de la pila debe de ser $\hat{\boldsymbol{q}}$ antes del evento $LinPt(\boldsymbol{op_{m+1}})$, pues en caso contrario algún evento lo modifico, y este evento debe de estar entre $\boldsymbol{op_m}$ y $\boldsymbol{op_{m+1}}$, lo cual podemos asegurar debido al orden total entre eventos. Para probar la afirmación anterior supongamos lo contrario, que algún evento $\boldsymbol{e}$ altero el estado de la pila y que este evento está entre $\boldsymbol{op_m}$ y $\boldsymbol{op_{m+1}}$.

Sabemos que el evento $\boldsymbol{e}$ que altero el estado de la pila forzosamente fue la ejecución exitosa de la línea 9 o 32. Sin embargo, en el caso que $\boldsymbol{e}$ es la ejecución de la línea 9, con un $\boldsymbol{CompareAndSet}$ exitoso, corresponde justamente al punto de linealización de alguna operación $\boldsymbol{Push(x)}$ en E (para algún $\boldsymbol{x}$), esto es por definición del punto de linealización.

Entonces, como este evento $\boldsymbol{e}$ cumple $\boldsymbol{op_m} \to \boldsymbol{e} \to \boldsymbol{op_{m+1}}$ y es un punto de linealización, tendría que existir una operación $\boldsymbol{op_k}$ tal que

$$\boldsymbol{op_m} \to \boldsymbol{op_k} \to \boldsymbol{op_{m+1}},$$

contradiciendo que el orden que induce la linealización por conjuntos S_{m+1}.

Por otro lado, si el evento $\boldsymbol{e}$ que altero el estado de la pila fue la ejecución de la línea 32, con un $\boldsymbol{CompareAndSet}$ exitoso, entonces fue ejecutado por una alguna operación $\boldsymbol{Pop'}()$. Veremos que ocurre lo mismo que el caso anterior pero con el punto de linealización de un conjunto de operaciones $\boldsymbol{Pop}()$.

Para mostrar esto simplemente notemos lo siguiente, si $\boldsymbol{a}$ es el valor retornado por $\boldsymbol{Pop'}()$ y U_a el conjunto de operaciones $\boldsymbol{Pop}()$ tal que devolvieron $\boldsymbol{a}$, entonces el punto de linealización de U_a es una ejecución exitosa de $\boldsymbol{t_k.upper.attemptMark(u, False)}$, con $\boldsymbol{a}$ el

valor del nodo $\boldsymbol{t_k}$. Por definición fue el primero, de modo que en el punto de linealización la marca de $\boldsymbol{t_k.upper}$ pasa de $\boldsymbol{True}$ a $\boldsymbol{True}$ (cambiando el estado físico de la pila), sin embargo, cualquier ejecución de $\boldsymbol{t_k.upper.attemptMark(u, False)}$, que sea efectuada después, no cambia la marca realmente, pues la marca es $\boldsymbol{False}$. Por lo tanto, únicamente el punto de linealización cambia el estado físico de la pila, con lo que se concluye que el evento $\boldsymbol{e}$, que altero el estado de la pila, corresponde al punto de linealización de un conjunto de operaciones $\boldsymbol{Pop(\,)}$. Usando los mismos argumentos que el caso anterior, se contradice el orden que induce la linealización por conjuntos S_{m+1}.

Por lo tanto, el estado de la pila entre del evento $LinPt(\boldsymbol{op_m})$ y antes del evento $LinPt(\boldsymbol{op_{m+1}})$ es $\hat{\boldsymbol{q}}$. Para finalizar la demostración mostremos en $LinPt(\boldsymbol{op_{m+1}})$ el estado físico cambia según la transición 4.1.1 y que sigue cumpliendo con la definición de estado de pila.

1. Si $\boldsymbol{op_{m+1}}$ es una clase de concurrencia correspondiente a una operación $\boldsymbol{Push(x)}$ en E_m: El punto de linealización de $\boldsymbol{op}$ es $LinPt(\boldsymbol{Push(x)}) = \boldsymbol{e_{CAS}}$, el cual corresponde a la ejecución de la línea 9 con un $\boldsymbol{CompareAndSet}$ exitoso en la última iteración. Este punto también es el único evento en $\boldsymbol{Push}$ en el cual el *estado físico* de la pila cambia, de $\hat{\boldsymbol{q}}$ a $\hat{\boldsymbol{q}} * \boldsymbol{x}$. Esto lo podemos afirmar gracias al lema 4.4.2

   Por lo que se cumple formalmente la transición:

   $$\delta(\hat{\boldsymbol{q}}, \boldsymbol{Push(x)}) = (\hat{\boldsymbol{q}} * \boldsymbol{x}, \langle \boldsymbol{Push(x)} : \boldsymbol{True} \rangle)$$

   Se concluye que S_{m+1} produce una ejecución secuencial de pila válida que es consistente con (4.1.1), donde estado de la pila está representado por el *estado físico* codificado en $\hat{\boldsymbol{q}} * \boldsymbol{x}$.

2. Si $\boldsymbol{op_{m+1}}$ es una clase de concurrencia correspondiente a un conjunto operaciones $\boldsymbol{Pop(\,)}$ en E_m, tenemos dos casos:

   a) La clase de concurrencia $\boldsymbol{op_{m+1}}$ corresponde a una sola operación $\boldsymbol{Pop(\,)}$ tal que $\langle \boldsymbol{Pop}(\,) : \epsilon \rangle$.

      El punto de linealización es $\boldsymbol{e_{get}}$, correspondiente a la línea 24 de la última iteración. Por suposición

      $$LinPt(\boldsymbol{op_m}) \rightarrow \boldsymbol{e_{get}}$$

      Por lo que el *estado físico* de la pila después de ejecutar $LinPt(\boldsymbol{op_m})$ debe de ser $\boldsymbol{\epsilon}$, pues en caso contrario en el evento $\boldsymbol{e_{get}}$ se obtendría un nodo $\boldsymbol{t}$ tal que $\boldsymbol{l} \neq \boldsymbol{\epsilon}$, y no se ejecutaría la línea 27 satisfactoriamente, contradiciendo $\langle \boldsymbol{Pop}(\,) : \epsilon \rangle$.
      Concluimos que para la operación $\boldsymbol{op_{m+1}}$ se cumple la transición:

      $$\delta(\boldsymbol{\epsilon}, \boldsymbol{Pop}(\,)) = (\boldsymbol{\epsilon}, \langle \boldsymbol{Pop}(\,) : \boldsymbol{\epsilon} \rangle)$$

      Por lo que concluimos que S_{m+1} produce una ejecución secuencial de pila válida que es consistente con (4.1.1), donde estado de la pila está representado por el estado físico codificado en $\boldsymbol{\epsilon}$.

b) La clase de concurrencia $\boldsymbol{op_{m+1}}$ corresponde a un conjunto de operaciones $U_x = \{\boldsymbol{Pop_1}(\,), ..., \boldsymbol{Pop_t}(\,)\}$ tal que para toda operación $\boldsymbol{Pop_i}(\,)$ devuelve: $\langle \boldsymbol{Pop_i}(\,) : \boldsymbol{x}\rangle$ con $\boldsymbol{x}$ un elemento no nulo.

Recordemos que el conjunto U_x es tal que contiene todas las operaciones $\boldsymbol{Pop}$ en E_{m+1} que devuelven el mismo elemento $\boldsymbol{x}$.

El punto de linealización es $LinPt(\boldsymbol{op}) = \boldsymbol{e^x_{MARK}}$, el cual corresponde a la primera ejecución de la línea 32 de todas las operaciones $\boldsymbol{Pop_i}$ en U_x (pensando en la última iteración siempre). Gracias al lema 4.4.2 y lo discutido anteriormente sabemos que este evento es el único donde el *estado físico* de la pila cambia de modo que se cumple la transición:

$$\delta(\hat{\boldsymbol{q}} * \boldsymbol{x}, \{\boldsymbol{Pop_1}(\,), ..., \boldsymbol{Pop_t}(\,)\}) = (\hat{\boldsymbol{q}}, \{\langle \boldsymbol{Pop_1}(\,) : \boldsymbol{x}\rangle, ..., \langle \boldsymbol{Pop_t}(\,) : \boldsymbol{x}\rangle\})$$

Por lo que concluimos que S_{m+1} produce una ejecución secuencial de pila válida que es consistente con (4.1.1), donde estado de la pila está representado por el estado físico codificado en $\hat{\boldsymbol{q}}$.

Por lo que concluimos que S_m produce una ejecución secuencial de pila válida que es consistente con (4.1.1) para toda $m \leq n$. Por consiguiente es válido para E y S.

4.4.5. Dinámica del algoritmo Set-Stack-Physic

Esta sección está destinada a mostrar más a detalle la dinámica de las operaciones ***Push*** y ***Pop***. Primero veamos brevemente la estructura de la pila, la cual, en este caso, está representada mediante una lista doblemente ligada, de forma que cada nodo posee dos referencias, una denominada ***upper*** la cual *apunta* al nodo superior y otra referencia llamada ***lower*** la cual apunta al nodo inferior. En la figura 4.7 *panel a)* la referencia ***lower*** se representa por una flecha apuntando hacia arriba con una casilla misma que representa la *marca* de la referencia, mientras que la referencia ***upper*** se representa por una flecha que apunta hacia abajo.

En la figura 4.7 *panel b)* se muestra un único nodo y sus referencias. Debido a que el algoritmo hace uso de los objetos a los cuales apuntan las referencias, es decir, ***c.upper*** y ***c.lower*** conviene fijar el estado inicial de la pila en donde la referencia ***Top*** apunte a un inicio nodo centinela cuyo funcionamiento es impedir que la referencia ***Top*** apunte a $\boldsymbol{\epsilon}$, lo cual provocaría una falla en implementación y algunas partes del algoritmo perderían sentido.

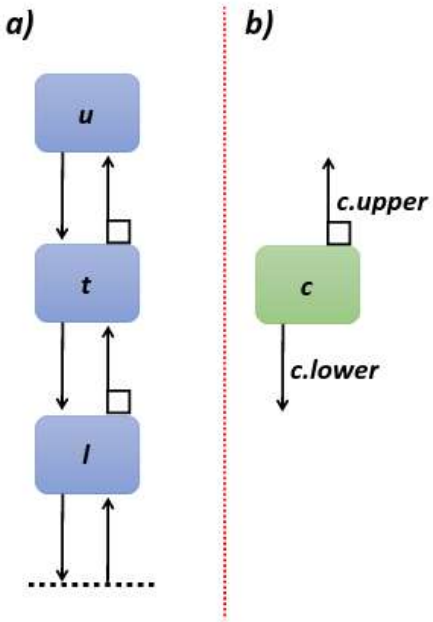


Figura 4.7: *Panel a)*: Representación gráfica de una pila que implementa una lista doblemente ligada. *Panel b)*: Representación de un único nodo o nodo centinela.

Como es de esperar, al implementar una lista doblemente ligada, las operaciones requieren de mecanismos más complejos para operar correctamente. Sin embargo, siguen guardando similitudes con las versiones anteriores, para esto se representa gráficamente el mecanismo de inserción y eliminación de cada nodo.

En la figura 4.8 se representa el mecanismo para que una operación ***Push*** tenga éxito. Para que una operación ***Push*** tenga éxito claramente se necesita que $\boldsymbol{u} == \boldsymbol{\epsilon}$, línea 7, por lo que supondremos que ***upper*** apunta a $\boldsymbol{\epsilon}$ (en la figura ***null***). Además de esto, en la línea 8 es necesario que la marca de $\boldsymbol{t.upper}$ sea ***True*** (casilla blanca), esto se representa en el *panel a)* junto con el nodo $\boldsymbol{x}$ que se pondrá en la pila.

La acción de la línea 6 se representa por el *panel b)* de la figura 4.8, donde la referencia $\boldsymbol{x.lower}$ se actualiza para apuntar al nodo cabecilla. Nótese que las condiciones $\boldsymbol{u} == \boldsymbol{\epsilon}$ y que la marca de $\boldsymbol{t.upper}$ sea ***True*** deben permanecer inalteradas.

La acción de la línea 9 se representa en el *panel c)* de la figura 4.8, donde el método ***CompareAndSet*** tiene éxito y la referencia $\boldsymbol{t.upper}$ ahora apunta a $\boldsymbol{x}$. Nótese que las condiciones en $\boldsymbol{u}$ y la marca de $\boldsymbol{t.upper}$ deben ser las mismas que al inicio y además no altera la marca. Por la acción de la línea 6 se puede estar seguro de que al finalizar la ejecución exitosa de la línea 9 esta es una lista doblemente ligada. Finalmente, se actualiza la referencia ***Top*** en la línea 10, esto se representa por la flecha punteada roja en el *panel c)* de la figura 4.8 y la operación $\boldsymbol{Push(x)}$ retorna ***True***.

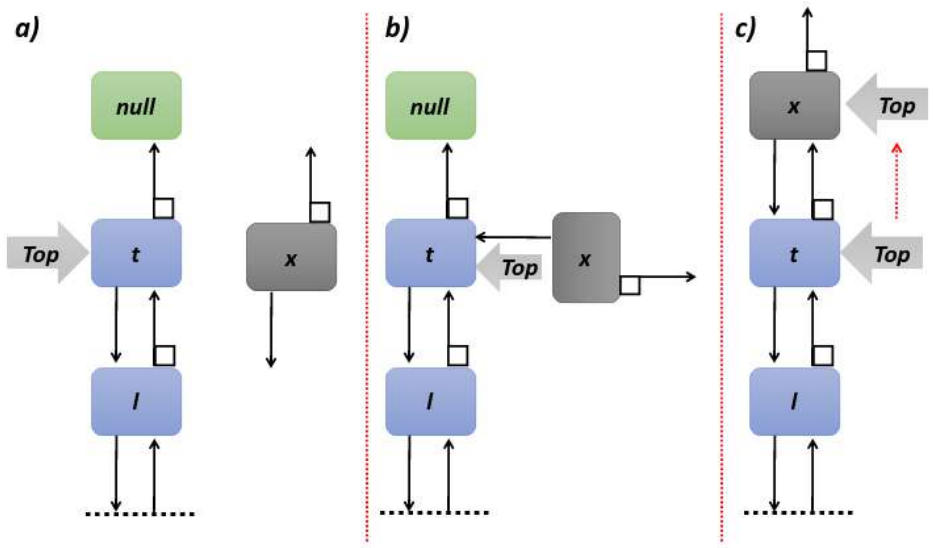


Figura 4.8: Mecanismo de ejecución de la operación $\boldsymbol{Push(x)}$.

En la figura 4.8 se puede ver lo siguiente: Para que el mecanismo tenga éxito se deben conservar las condiciones de $\boldsymbol{u}$ y la marca de $\boldsymbol{t.upper}$ del inicio. El otro punto es que, ninguna operación $\boldsymbol{Push(x)}$ altera la marca de los nodos, más bien, evalúan si su marca $\boldsymbol{upper}$ es $\boldsymbol{True}$. Esto permite crear un mecanismo de exclusión mutua, pues la marca indica si la operación $\boldsymbol{Push(x)}$ se le permite añadir el nodo *arriba* de algún nodo o lo que es lo mismo, actualizar la referencia $\boldsymbol{upper}$ de algún nodo. En el algoritmo 9 podemos ver que la única operación que cambia la marca $\boldsymbol{upper}$ es $\boldsymbol{Pop}$, esto se demuestra formalmente más adelante.

La figura 4.8 muestra gráficamente el funcionamiento de una operación $\boldsymbol{Pop(\,)}$. Al igual que en la operación $\boldsymbol{Push}$ es necesario que se cumplan las condiciones en $\boldsymbol{u}$ y en la marca de $\boldsymbol{t.upper}$, estas condiciones están representadas en el *panel a)* de la figura 4.8. Consideremos también que la pila no está vacía, es decir, hay al menos un nodo arriba del centinela. Luego, en el *panel b)* de la figura 4.8, se muestra el paso donde la línea 32 es ejecutada con éxito, y la marca de $\boldsymbol{t.upper}$ es marcada como $\boldsymbol{False}$ (representado por una casilla roja). Aquí es importante mencionar que $\boldsymbol{t.upper.attemptMark}$ tendrá éxito solo si la referencia $\boldsymbol{upper}$ sigue apuntando a $\boldsymbol{\epsilon}$, por lo tanto, si alguna operación $\boldsymbol{Push}$ añade un nodo antes, el método $\boldsymbol{attemptMark}$ fallará. En evento donde la marca de $\boldsymbol{t.upper}$ cambiar, es donde el nodo $\boldsymbol{t}$ ha sido *eliminado* de la pila, más adelante se formalizara esto, por el momento se le puede considerar como una eliminación lógica. Los siguientes dos pasos, la ejecución de la línea 33, representado en el *panel c)* y la línea 34 en el *panel d)* de la figura 4.8.

Podemos resaltar que una vez que la marca de $\boldsymbol{t.upper}$ se marca ninguna operación $\boldsymbol{Push}$ tendrá efecto. Esto es de suma importancia, pues esto provoca una exclusión mutua entre las operaciones $\boldsymbol{Pop}$ y $\boldsymbol{Push}$.

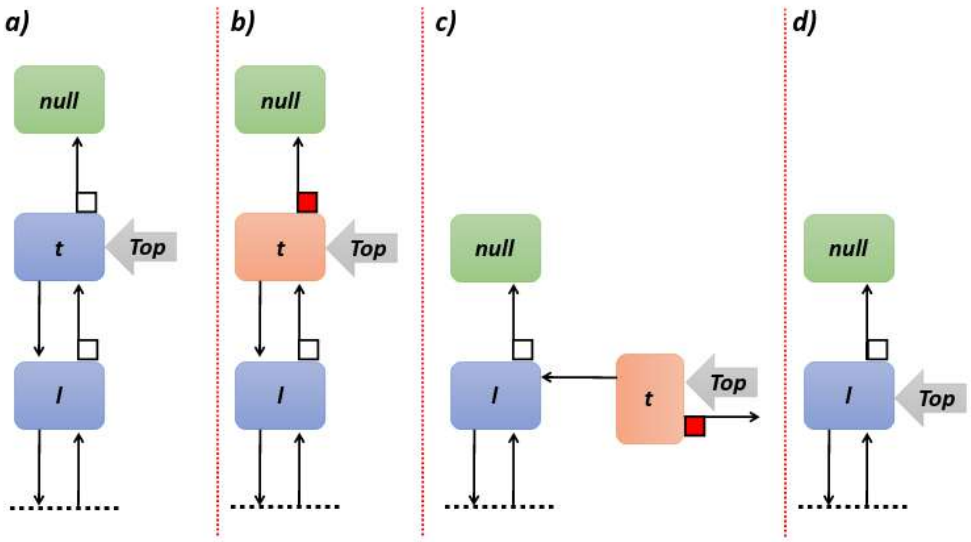


Figura 4.9: Mecanismo de ejecución de la operación ***Pop*( )**.

Si bien el funcionamiento individual de cada operación puede ser correcto, para que el algoritmo sea correcto este debe ser linealizable por conjuntos, y esto implica que el funcionamiento de las operaciones concurrentes sea correcto. Debido a esto es importante mencionar la dinámica en una interacción de dos operaciones ***Pop*** y ***Push***.

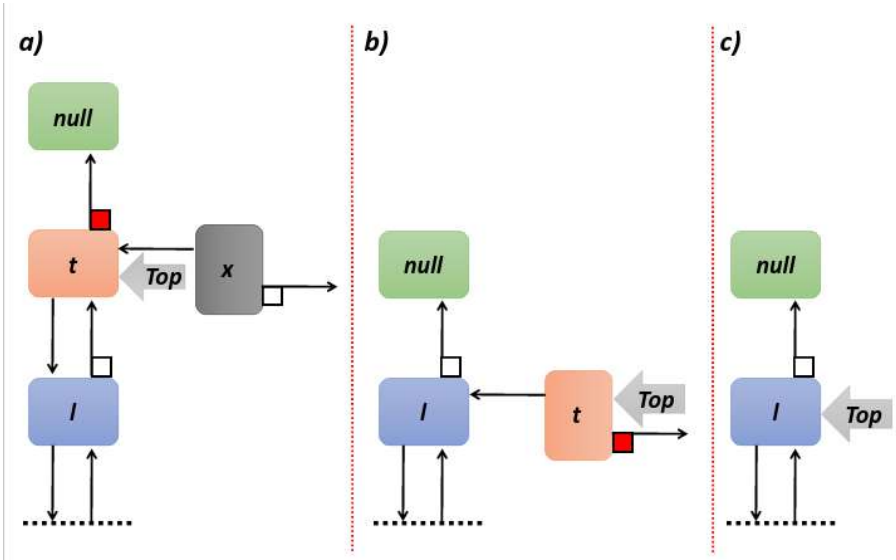


Figura 4.10: Mecanismo de exclusión mutua en un caso particular, donde la operación ***Push*(*x*)** falla.

En el *panel a)* de la figura 4.10 se muestra un escenario posible, donde una operación ***Push*** encuentra en la línea 8, que la referencia ***t.upper*** se encuentra marcada. Podemos pensar que esto es debido a que alguna operación ***Pop*** le *ganó* al evento donde se ejecuta la línea 9. Por lo tanto, la operación ***Push*** pasa a ejecutar las líneas 14 y 15, representadas en

los paneles b), c) respectivamente. Remarquemos dos puntos: Primero, las líneas 14, 15 son equivalentes a las líneas 33, 34 y a las líneas 38, 39. Segundo, si alguna operación se ejecutara entre la línea 14 y 15, encontraría un modo marcado, por lo que procedería a ejecutar algún par de líneas equivalentes.

Capítulo 5

Rendimiento experimental

Se evaluaron las tres implementaciones linealizables por conjuntos que se propusieron: *Set-Queue*, *Set-Stack-Logic* y *Set-Stack-Physic*, y las dos implementaciones linealizables *LockFreeQueue* y *LockFreeStack*. Por un lado, se está comparando el rendimiento experimental de los algoritmos de colas, es decir, comparar el rendimiento de *Set-Queue* y *LockFreeQueue*. Por otro lado, estamos comparando el rendimiento experimental de los algoritmos de pilas, es decir, de los algoritmos *LockFreeStack*, *Set-Stack-Logic* y *Set-Stack-Physic*. Esto con el fin de comparar estructurar de datos equivalentes pero con una condición de corrección o de linealizabilidad diferente.

5.1. Plataforma y entorno experimental

Todos los resultados que se muestran en este capítulo fueron obtenidos de experimentos realizados en una máquina AMD Threadripper 3970X con 32 núcleos, cada uno multiplexando 2 hilos de hardware, permitiendo 64 hilos en total; cada núcleo tiene cachés L1 y L2 privados y comparte un caché L3. Esta máquina ha sido utilizada en [5]. Los algoritmos se implementaron en la plataforma Java; OpenJDK 17.0.7. Esto con el fin de probarlos y que pudieran ser reproducidos en cualquier entorno informático multiplataforma.

5.2. Metodología

Con el fin de analizar el rendimiento de los algoritmos se ha utilizado el *Experimento* representado en el pseudocódigo 10, cuyo esquema general es el siguiente:

- Se elige y se fija un algoritmo concurrente; llamémosle $\boldsymbol{\mathcal{C}}$. Este algoritmo puede ser, ya sea alguno de los algoritmos linealizable por conjuntos, que se han presentado en tesis como contribuciones, o a las versiones linealizables en las que están inspiradas las contribuciones. Dicho de otra forma, el algoritmo $\boldsymbol{\mathcal{C}}$ pertenece al siguiente conjunto de algoritmos:

$$\{LockFreeQueue, LockFreeStack, SetQueue, SetStackLogic, SetStackPhysic\},$$

Algorithm 10 Experimento

$\boldsymbol{\mathcal{C}} \in \{LockFreeQueue, LockFreeStack, SetQueue, SetStackLogic, SetStackPhysic\}$

1: **for** $h \in \{1, 2, ..., 64\}$ **do**
2: `Se crean` h `hilos`
3: `Para cada hilo se ejecuta el siguiente proceso:`
4: **procedure** TAREA RUNNABLE($\boldsymbol{\mathcal{C}}$)
5: `Al iniciar simultaneamente los` h `hilos se registra el` *Tiempo inicial*
6: **for** $e \in \{1, 2, ..., L/h\}$ **do**
7: $\boldsymbol{\mathcal{C}.op_{in}}(e)$
8: $\boldsymbol{\mathcal{C}.op_{out}}(\,)$
9: **end for**
10: **end procedure**
11: `Al finalizar los` h `hilos se registra el` *Tiempo final*
12: $T :=$ `Tiempo final - Tiempo inicial`
13: **end for**

- Dado un algoritmo $\boldsymbol{\mathcal{C}}$ fijo, se inicia un ciclo ***for***, con el cual h representa el número de hilos. El fin de cada iteración es evaluar el desempeño del algoritmo $\boldsymbol{\mathcal{C}}$ bajo la ejecución de h procesos que se ejecutan simultáneamente.

- Para cada iteración se crean h hilos; línea 2. La cantidad de hilos va desde 1 hasta 64, de esta forma se busca evaluar el desempeño del algoritmo $\boldsymbol{\mathcal{C}}$ en un entorno secuencial con $h = 1$, hasta un entorno concurrente donde se ocupan todos los procesadores de la máquina con $h = 64$.

- Los h hilos ejecutan concurrentemente el proceso TAREA RUNNABLE; línea 3. En la implementación se realiza esto mediante un *Thread Pool*, donde cada hilo realiza ejecuta el método *Runnable* representado por el proceso TAREA RUNNABLE.

- Se registra el *Tiempo inicial*, donde los h hilos inician el proceso al mismo tiempo; línea 5. Este punto es crucial, puesto que los h hilos deben iniciar simultáneamente la ejecución del proceso TAREA RUNNABLE. Para esto se implementó un objeto *CyclicBarrier* y su método *await*, con el cual se duermen los hilos hasta que los h hilos lleguen al mismo punto. Esto establece la misma condición inicial para los h hilos.

- De forma abstracta usaremos la siguiente nomenclatura: La operación $\boldsymbol{op_{in}}(e)$ representa una operación $\boldsymbol{Enqueue}(e)$ o $\boldsymbol{Push}(e)$, dependiendo de si el algoritmo es de una cola o pila. La operación $\boldsymbol{op_{out}}()$ representa una operación $\boldsymbol{Dequeue}(\,)$ o $\boldsymbol{Pop}(\,)$, dependiendo de si el algoritmo es de una cola o pila.

- Cada hilo ejecuta un ciclo ***for*** en donde en cada iteración se ejecutan las operaciones $\boldsymbol{op_{in}}(e)$ y $\boldsymbol{op_{out}}(\,)$ del algoritmo $\mathcal{C}$ con e de 1 hasta L/h; línea 6 a 8. El entero L se supone fijo, y representa el número de operaciones totales que se realizaran para los h hilos. Dicho de otra forma, estamos distribuyendo L operaciones en h hilos, para que se operen el mismo número de operaciones independientemente de h. Por ejemplo, para $h = 1$, se realizan el mismo número de operaciones que en $h = 64$.

- Al finalizar los h hilos se registra el *Tiempo final* y se mide el tiempo que tardo el proceso; línea 12 y 13.

El rendimiento experimental se representa por el tiempo T que tardaron los h hilos en completar las L operaciones y es dependiente, únicamente, del número de hilos; pues se suponen fijos el número de operaciones L y el algoritmo $\boldsymbol{\mathcal{C}}$.

Antes de presentar los resultados tengamos en cuenta lo siguiente. Cuando varios subprocesos o hilos intentan adquirir o acceder al mismo recurso compartido, ocurre un fenómeno conocido como: *contención*; la cual provoca que los hilos contendientes se ejecutan más lentamente de lo que lo haría si los otros hilos no se estuvieran ejecutando. Una contención alta significa que hay muchos hilos de este tipo, y contención baja significa lo contrario.

Esto motiva a probar el conjunto de algoritmos en ambos escenarios. Con el fin de evaluar y reportar el rendimiento de los algoritmos y de sus contra-partes linealizables se ha dividido el análisis dos esquemas: *Experimento de alta contención* y *experimento de baja contención*; estos se especifican en las secciones 5.3 y 5.4, respectivamente.

En las siguientes secciones se reportan el promedio en el rendimiento $T(h) := \sum_{i=1}^{100} T_i(h)$, donde $T_i(h)$ es el rendimiento experimental para la i-ésima muestra con h hilos. Para ambos esquemas se tomó una muestra de $N = 100$ ejecuciones del *Experimento* con $L = 49,008,960$ operaciones.

5.3. Resultados: *Experimento de baja contención*

Esta sección se centra en un escenario de baja contención, para el cual se utiliza el esquema general ilustrado en el pseudo-algoritmo 10 con una ligera modificación; nos referiremos a este esquema como *experimento de baja contención*.

Tengamos en cuenta que todas las operaciones ***enqueue*** y ***dequeue***, para las colas, y ***push*** y ***pop***, para las pilas, iteran sobre un ciclo ***while***. Aún más, recordemos que en este ciclo cada iteración es un intento por efectuar la operación correspondiente. Esto, junto con el hecho que todos los algoritmos son *non-blocking* nos indican que cada en iteración donde una operación no ha finalizado implica que otra si finalizo; la idea es similar a la deducción que se usó para demostrar *non-blocking*. Podemos entender este proceso como una lectura a un recurso compartido que se encuentra ocupado y que adquiere acceso hasta que se desocupa; muy similar al mecanismo de candados con la gran ventaja de ser *non-blocking*. De aquí se tiene una observación importante; una gran cantidad de operaciones intentando adquirir acceso a un mismo recurso compartido implica una gran contención, esto a pesar de que las operaciones son *non-blocking*.

En esta tesis se utilizó un enfoque simple para disminuir la contención. Se ha visto que la contención disminuye cuando las operaciones *esperan* por un tiempo, dando a los hilos de la competencia la oportunidad de finalizar. En los algoritmos de colas y pilas concurrentes

y concurrentes por conjuntos se utiliza un *delay* antes de finalizar cada iteración en los ciclos ***while*** de las operaciones; la `Operacion` representa ya sea un ***pop***, ***push***, ***dequeue*** o ***enqueue***, y el delay se emplea de la siguiente forma:

Algorithm 11 Modificación para experimento de baja contención

Variables Compartidad
1: **procedure** OPERACIÓN(***Entrada***)
2: `Instrucción 1`
3: ...
4: `Instrucción i`
5: ...
6: `Instrucción m`
7: ***Delay***(lim)
8: **end procedure**

Después de que las m instrucciones de la `Operación` se han ejecutado, se emplea una operación ***Delay***, la cual *duerme* al hilo antes de volver a intentar otra iteración. Para asegurarse de que los hilos en conflicto simultáneos se distribuyan y no vuelvan a tratar de adquirir acceso al mismo tiempo, el hilo duerme por una duración aleatoria, la cual se encuentra entre cero y el tiempo lim; es necesario un límite para evitar que los hilos desafortunados duerman demasiado tiempo.

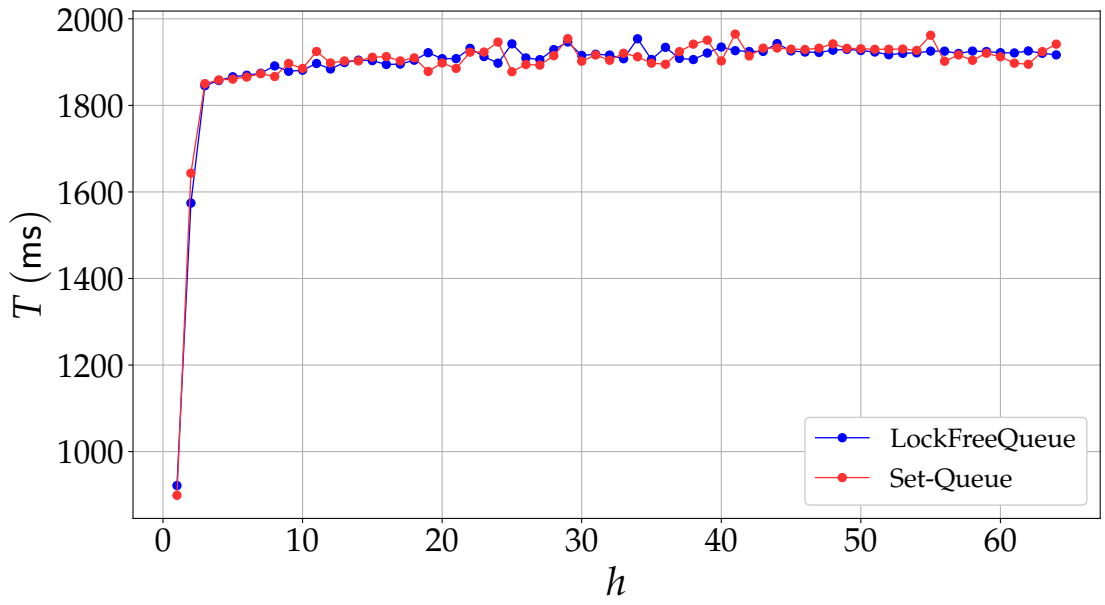


Figura 5.1: Tiempo en milisegundos (ms) para realizar $L = 49,008,960$ pares de operaciones ***enqueue*** y ***dequeue*** contra el número hilos h.

En la figura 5.1 se muestra el rendimiento experimental promedio $T(h)$ para los algoritmos de colas. El rendimiento experimental fue obtenido implementando el esquema general para

el experimento, representado en el pseudo-algoritmo 10 con los algoritmos de colas *LockFree-Queue* y *Set-Queue*, empleando la modificación para el *experimento de baja contención* 11 en las operaciones ***enqueue*** y ***dequeue***.

Se muestra que el rendimiento experimental promedio $T(h)$ de ambos algoritmos *Lock-FreeQueue* y *Set-Queue* es similar en el *experimento de baja contención*. Vemos que el mejor rendimiento corresponde a un solo hilo, con $h = 1$, siendo $T(h = 1) \approx 921.59$ ms el valor del mejor rendimiento promedio para *LockFreeQueue* y $T(h = 1) \approx 898.97$ ms para *Set-Queue*. Por otro lado, al aumentar en el número de hilos aumenta el tiempo de ejecución para ambos algoritmos. Para $3 \leq h$ el rendimiento promedio $T(h)$ se encuentra acotado entre 1845 ms y 1964 ms, manteniéndose casi constante, independientemente del número de hilos. Esto nos indica que el mejor rendimiento, para ambos algoritmos de colas, corresponde al caso secuencial, donde un solo hilo realiza todas las operaciones. Aún más, ambos algoritmos tienen un comportamiento similar, lo cual tiene es de esperar al ser *Set-Queue* una modificación mínima de *LockFreeQueue*.

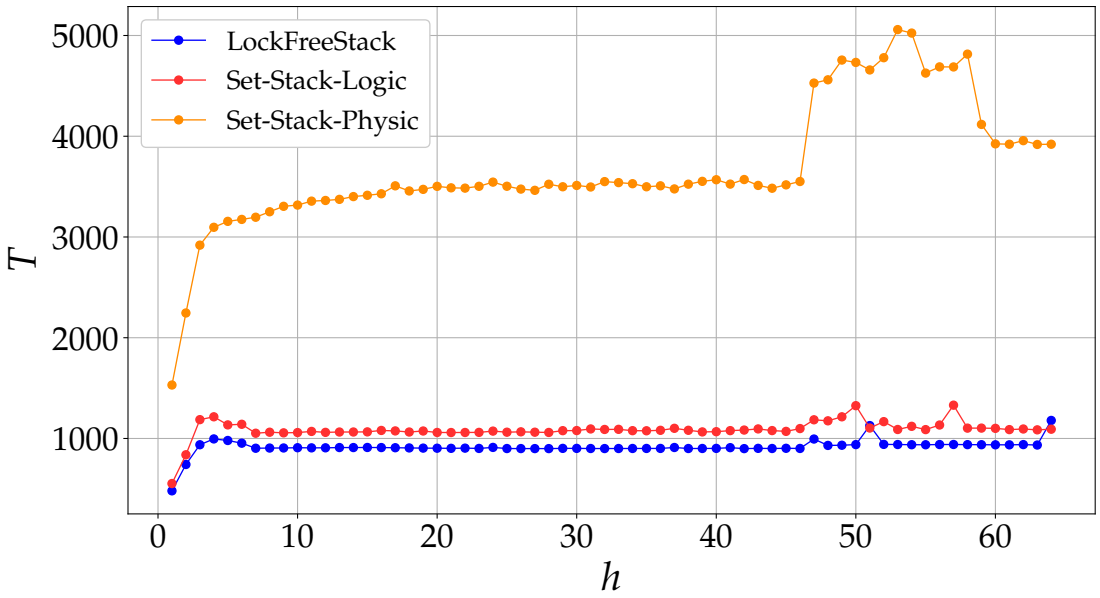


Figura 5.2: Tiempo en milisegundos para realizar $L = 49,008,960$ pares de operaciones ***push*** y ***pop*** contra el número hilos h.

En la figura 5.2 se muestra el rendimiento experimental promedio $T(h)$ para los algoritmos de pilas. El rendimiento experimental fue obtenido implementando el esquema general para el experimento, representado en el pseudo-algoritmo 10 con los algoritmos de pilas *LockFreeStack*, *Set-Stack-Logic* y *Set-Stack-Physic*, empleando la modificación para el *experimento de baja contención* 11 en las operaciones ***pop*** y ***push***.

Podemos notar que el comportamiento es bastante parecido para los algoritmos de pilas, manteniendo un rendimiento $T(h)$ constante para $3h$; con una clara excepción para el algoritmo

Set-Stack-Physic. El algoritmo *Set-Stack-Physic* muestra un peor rendimiento en comparación con los algoritmos *LockFreeStack* y *Set-Stack-Logic*. Posiblemente, esto sea debido a la compleja estructura de *Set-Stack-Physic* así como al uso de ***AtomicMarkableReference*** en los nodos. Aún más, el algoritmo *Set-Stack-Physic* no muestra un comportamiento estable en el rendimiento $T(h)$ al aumentar el número de hilos h, presentando una gran variación a partir de $45 \leq h$. Por otro lado, los algoritmos *LockFreeStack* y *Set-Stack-Logic* presentan un comportamiento y rendimiento similar, siendo *LockFreeStack* el algoritmo con mejor rendimiento. En todos los algoritmos se aprecia que el rendimiento $T(h)$ se mantiene casi constante para $3h$. Al igual que los algoritmos de colas, el mejor rendimiento se presenta para el caso secuencial $h = 1$, esto para todos los algoritmos de pilas.

5.4. Resultados: *Experimento de alta contención*

Esta sección se centra en un escenario de alta contención, para el cual se utiliza el esquema general ilustrado en el pseudo-algoritmo 10 sin ningún tipo de modificación; nos referiremos a este esquema como *experimento de alta contención*.

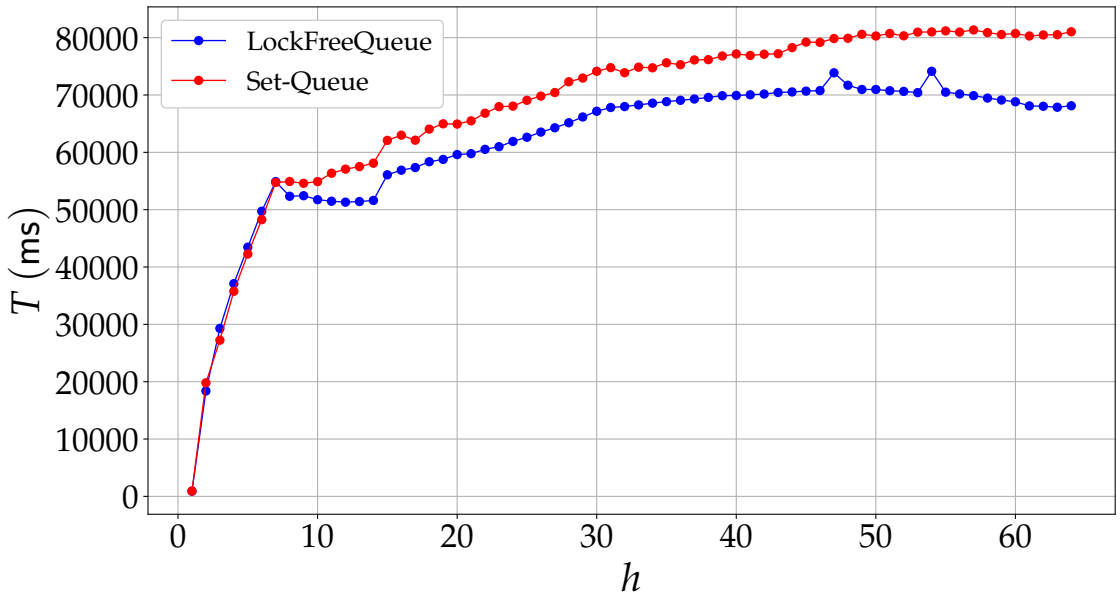


Figura 5.3: Tiempo en milisegundos (ms) para realizar $L = 49,008,960$ pares de operaciones ***enqueue*** y ***dequeue*** contra el número hilos h.

En la figura 5.3 se muestra el rendimiento experimental promedio $T(h)$ para los algoritmos de colas. El rendimiento experimental fue obtenido implementando el esquema general para el experimento, representado en el pseudo-algoritmo 10 con los algoritmos de colas *LockFree-Queue* y *Set-Queue*, sin ningún tipo de modificación, para el *experimento de alta contención*.

En el *experimento de alta contención* el rendimiento experimental de ambos algoritmos es similar. Sin embargo, a diferencia del rendimiento experimental en el *experimento de baja contención*, se muestra una diferencia a partir de $h = 8$ hilos, siendo el algoritmo *LockFree-Queue* el cual posee un mejor rendimiento. Este comportamiento es de hecho inesperado, puesto que el algoritmo *Set-Queue* es una versión ligeramente más simple que el algoritmo *LockFreeQueue*.

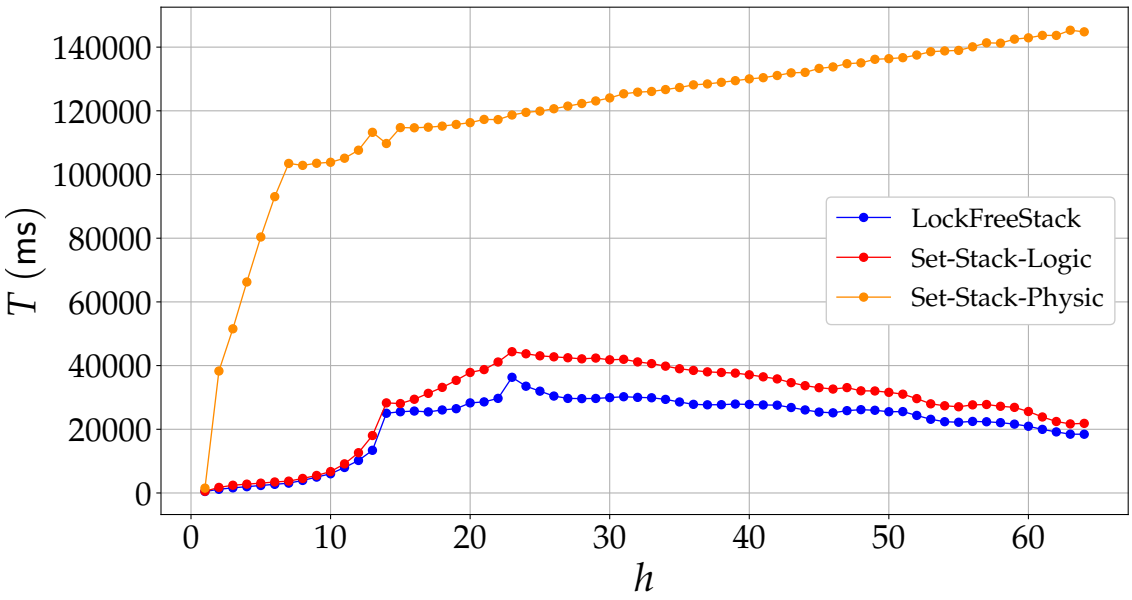


Figura 5.4: Tiempo en milisegundos para realizar $L = 49,008,960$ pares de operaciones ***push*** y ***pop*** contra el número hilos h.

En la figura 5.4 se muestra el rendimiento experimental promedio $T(h)$ para los algoritmos de pilas. El rendimiento experimental fue obtenido implementando el esquema general para el experimento, representado en el pseudo-algoritmo 10 con los algoritmos de pilas *LockFreeStack*, *Set-Stack-Logic* y *Set-Stack-Physic*, sin emplear modificación alguna para el *experimento de alta contención*.

Se observa que, a diferencia de los resultados obtenidos en el *experimento de baja contención*, el comportamiento de los algoritmos al aumentar el número de hilos no es constante. En el caso del algoritmo *Set-Stack-Physic* se observa un comportamiento creciente y lineal, del rendimiento experimental promedio $T(h)$, con respecto al número de hilos h; esto a partir de cierto número de hilos. Por otro lado, el rendimiento $T(h)$ de los algoritmos *LockFreeStack* y *Set-Stack-Logic* es creciente en $1 < h \leq 14$ y después de cierto número de hilos este se vuelve decreciente, predominando *LockFreeStack* con el mejor rendimiento.

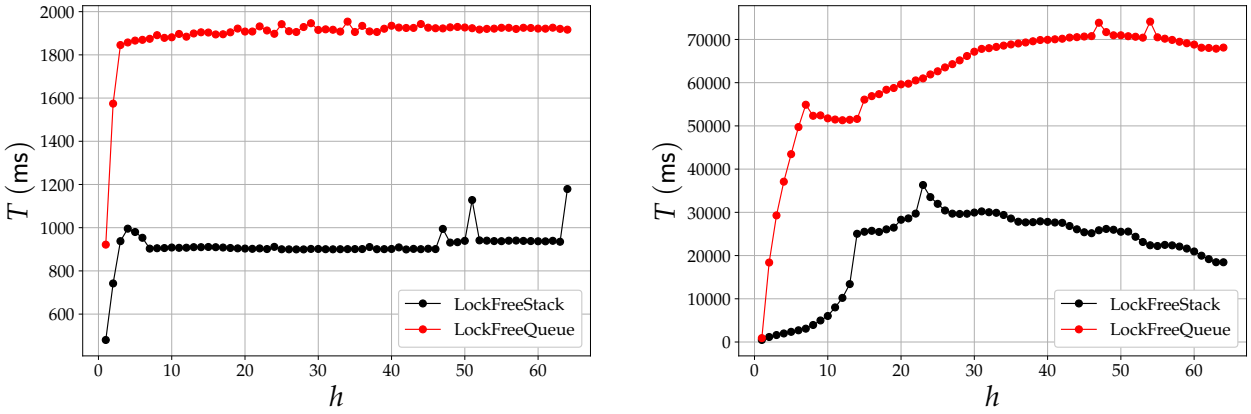


Figura 5.5: Tiempo en milisegundos para realizar $L = 49,008,960$ pares de operaciones ***push*** y ***pop*** contra el número hilos h.

Con base en los resultados del *experimento de baja contención* y del *experimento de alta contención*, se ha observado que los algoritmos *LockFreeQueue* y *LockFreeStack* muestran un rendimiento igual o mejor al de sus respectivas versiones linealizables por conjuntos. Resaltando el rendimiento de *Set-Stack-Physic* como el que posee el rendimiento más deficiente. Esto podría deberse, en parte, al uso de objetos como ***AtomicMarkableReference***, indicando que el uso de referencias como estas implica un alto costo computacional. Esta idea se ve reforzada al comparar los recudimentos de *LockFreeQueue* y *LockFreeStack*.

En la figura 5.5 se muestran los rendimientos $T(h)$ de *LockFreeQueue* y *LockFreeStack*, en el *panel izquierdo* se comparan los resultados para el *experimento de baja contención* y en el *panel derecho* se comparan los resultados para el *experimento de alta contención*. Si bien se trata de estructuras de datos distintas, es claro que el rendimiento de *LockFreeStack* es mejor que el de *LockFreeQueue*, esto independientemente del esquema y del número de hilos. Dado que el algoritmo *LockFreeQueue* hace uso de referencias del tipo ***AtomicReference*** en los nodos, esto podría ser un indicativo de que las referencias que consisten en objetos son mucho más costosas.

Capítulo 6

Conclusiones y discusión

En este trabajo, se han presentado nuevas contribuciones de algoritmos para colas y pilas concurrentes por conjuntos (con semántica relajada de multiplicidad): *Set-Queue*, *Set-Stack-Logic* y *Set-Stack-Physic*. Para cada uno de los algoritmos anteriores se desarrolló un análisis detallado en su comportamiento concurrente y se demostró formalmente que satisfacen la condición de correctitud, cumpliendo ser linealizables por conjuntos. Además, se demostró que estas contribuciones son algoritmos *non-blocking* y que efectivamente son implementaciones de sus respectivas estructuras, es decir, pilas o colas con multiplicidad. Estas contribuciones introducen nuevos mecanismos de sincronización para los algoritmos concurrentes por conjuntos. En trabajos como [8, 7, 22] se han utilizado mecanismos de sincronización basados en operaciones *Write/Read*, mientras que en este trabajo los mecanismos de sincronizaciones están basados en operaciones ***CompareAndeSet***.

Se mostró y discutió las diferentes modificaciones que permitieron a los algoritmos *LockFreeQueue* y *LockFreeStack* poseer multiplicidad, es decir, relajar su semántica. Mostrando que, en el caso del algoritmo concurrente de colas, la modificación para obtener una relajación en su semántica fue mínima, lo cual es gracias a la estructura misma de la cola. Esto podría ser un indicador de que la estructuras de colas poseen una concurrencia por conjuntos inherentes. Sin embargo, las modificaciones en la pila concurrente fue no trivial, pues condujo a dos algoritmos distintos (cada uno con su propia representación de estados). Además, estos algoritmos son más sofisticados y complejos que la versión original. Esto muestra que las implementaciones de colas y pilas con multiplicidad poseen una clara representación por medio de autómatas secuenciales por conjuntos; aún más, la representación de estados permitió demostrar formalmente que se satisfacen las transiciones de estado. Este tipo de representaciones podría ser extendido para formalizar la definición, por medio de máquinas de estados, de otras estructuras de datos.

Durante el presente trabajo se definió formalmente el *estado* de la respectiva estructura de datos concurrente. Se mostró que mediante la definición formal de estados, de colas o pilas, se puede apreciar mejor las *transición de estados* mostrando formalmente la implementación de colas o pilas concurrentes por conjuntos. Aún más, se ha mostrado en este trabajo que diferentes definiciones de estados, como los mostrados *estado lógico* y *estado físico*, implican un algoritmo distinto, teniendo cada uno sus propias características. Hemos visto que el *estado lógico* permite un algoritmo más claro y directo, el *estado físico* permite un mayor

control en la memoria, más explícitamente, el *estado lógico* permite que exista información no relevante, siendo esta los nodos que no están lógicamente en la pila, pero si se encuentran en la memoria compartida, mientras que el *estado físico* está representado por completo, salvo un par de nodos (lo cual es un número casi constante, acotado por dos nodos), por la memoria compartida (la memoria física).

A partir de los resultados experimentales se observa que los algoritmos concurrentes *LockFreeQueue* y *LockFreeStack* poseen un mejor rendimiento a comparación de las versiones linealizables por conjuntos. Esto se podría atribuir a que los algoritmos de pilas son modificaciones no triviales cuya estructura vuelve al algoritmo más complejo y costoso. Sin embargo, los resultados experimentales en los algoritmos de colas indican que la concurrencia por conjunto podría ser más costosa experimentalmente por sí misma. Pues, a partir del *experimento de baja contención*, se sabe que los rendimientos de *LockFreeQueue* y *Set-Queue* son aproximadamente iguales. Además, *Set-Queue* posee prácticamente la misma estructura que *LockFreeQueue* con una leve modificación que no aumenta el número de instrucciones, de hecho las disminuye. Esto contrasta con los resultados experimentales del *experimento de alta contención*, en donde si existe una diferencia en los rendimientos, siendo el algoritmo *LockFreeQueue* el cual predomina con mejores resultados al aumentar el número de hilos.

Bibliografía

[1] Afek, Y., Gafni, E., and Morrison, A. Common2 extended to stacks and unbounded concurrency. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2006), PODC '06, Association for Computing Machinery, p. 218–227.

[2] Alur, R., McMillan, K., and Peled, D. Model-checking of correctness conditions for concurrent objects. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science* (1996), pp. 219–228.

[3] Attiya, H., Guerraoui, R., Hendler, D., and Kuznetsov, P. The complexity of obstruction-free implementations. vol. 56, Association for Computing Machinery.

[4] Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M. M., and Vechev, M. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, Association for Computing Machinery, p. 487–498.

[5] Castañeda, A., and Piña, M. Modular Baskets Queue. *arXiv e-prints* (May 2022), arXiv:2205.06323.

[6] Castañeda, A., Rajsbaum, S., and Raynal, M. Unifying concurrent objects and distributed tasks: Interval-linearizability. vol. 65, Association for Computing Machinery.

[7] Castañeda, A., and Pina, M. Fully read/write fence-free work-stealing with multiplicity. In *International Symposium on Distributed Computing* (2021), pp. 16:1–16:20.

[8] Castañeda, A., Rajsbaum, S., and Raynal, M. Set-linearizable implementations from read/write operations: Sets, fetch &increment, stacks and queues with multiplicity. *Distributed Computing 36*, 2 (Jun 2023), 89–106.

[9] Castañeda, A., Rajsbaum, S., and Raynal, M. Relaxed queues and stacks from read/write operations. *OPODIS* (2020), 13:1–13:19.

[10] Ellen, F., and Brown, T. Concurrent data structures. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2016), PODC '16, Association for Computing Machinery, p. 151–153.

[11] Ellen, F., Hendler, D., and Shavit, N. On the inherent sequentiality of concurrent objects. *SIAM Journal on Computing 41*, 3 (2012), 519–536.

[12] Haas, A., Lippautz, M., Henzinger, T. A., Payer, H., Sokolova, A., Kirsch, C. M., and Sezgin, A. Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation. CF '13, Association for Computing Machinery.

[13] Henzinger, T. A., Kirsch, C. M., Payer, H., Sezgin, A., and Sokolova, A. Quantitative relaxation of concurrent data structures. vol. 48, Association for Computing Machinery, p. 317–328.

[14] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming, Revised Reprint*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[15] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. vol. 12, Association for Computing Machinery, p. 463–492.

[16] Höglund, J. An analysis of a distributed tracing systems effect on performance jaeger and opentracing api. `https://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-175889`, 2020.

[17] Kirsch, C. M., Payer, H., Röck, H., and Sokolova, A. Performance, scalability, and semantics of concurrent fifo queues. In *Algorithms and Architectures for Parallel Processing* (Berlin, Heidelberg, 2012), Y. Xiang, I. Stojmenovic, B. O. Apduhan, G. Wang, K. Nakano, and A. Zomaya, Eds., Springer Berlin Heidelberg, pp. 273–287.

[18] Lamport, L. Time, clocks, and the ordering of events in a distributed system. vol. 21, Association for Computing Machinery, p. 558–565.

[19] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. vol. 28, IEEE Computer Society, p. 690–691.

[20] McCaffrey, C. The verification of a distributed system. vol. 59, Association for Computing Machinery, p. 52–55.

[21] Michael, M. M., Vechev, M. T., and Saraswat, V. A. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2009), PPoPP '09, Association for Computing Machinery, p. 45–54.

[22] Neiger, G. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1994), PODC '94, Association for Computing Machinery, p. 396.

[23] Shavit, N. Data structures in the multicore age. vol. 54, Association for Computing Machinery, p. 76–84.

[24] Singhal, M., and Kshemkalyani, A. An efficient implementation of vector clocks. *Information Processing Letters 43*, 1 (1992), 47–52.

[25] Sukhija, N., Bautista, E., James, O., Gens, D., Deng, S., Lam, Y., Quan, T., and Lalli, B. Event management and monitoring framework for hpc environments using servicenow and prometheus. In *Proceedings of the 12th International Conference on Management of Digital EcoSystems* (New York, NY, USA, 2020), MEDES '20, Association for Computing Machinery, p. 149–156.