Trabajo Práctico

Estructuras de Datos, Universidad Nacional de Quilmes

Primer semestre 2018

Fecha de la defensa: sábado 7 de julio de 2018.

Índice

1. Introducción

Papá Noel ha decidido diversificar sus servicios, ofreciendo entregas de regalos durante todos los días del año. Cualquier persona, niña o niño, puede enviarle una carta a Papá Noel solicitando un regalo para una fecha específica. A lo largo de cada día, los elfos monitorean el comportamiento de todas las personas que pidieron un regalo para ese día. Si los elfos registran que una persona tuvo buen comportamiento, se le hace entrega del regalo pedido. Si registran que tuvo mal comportamiento, no se le entrega el regalo. Supondremos que la fecha de inicio de actividades tiene lugar en el día 0. El objetivo de este TP es implementar un programa en C++ para administrar el emprendimiento de Papá Noel y los elfos.

2. Descripción del sistema

2.1. Interfaz del tipo abstracto `PapaNoel`

Utilizaremos los siguientes renombres de tipos. El tipo `Fecha` es un renombre de `int` que representa la cantidad de días transcurridos desde el inicio de actividades. El tipo `Id` es un renombre de `int` que sirve para identificar personas de manera única.

```
typedef int Fecha;
typedef int Id;
```

También contaremos con un tipo `Estado` que representa el estado de una persona en una fecha. Puede tomar cuatro posibles valores:

```
typedef int Estado;

const Estado Inexistente       = 0;
const Estado Pendiente         = 1;
const Estado Entregado         = 2;
const Estado MalComportamiento = 3;
```

El sistema contará con la siguiente interfaz:

1. `PapaNoel iniciarPN()`
   Propósito: Devuelve un sistema nuevo.

2. `Fecha fechaActualPN(PapaNoel p)`
   Propósito: Devuelve la fecha actual (la "fecha de hoy") del sistema.

3. `void registrarPedidoPN(PapaNoel p, Fecha f, Id x)`
   Propósito: Registra la solicitud de entregarle un regalo a la persona `x` cuando llegue la fecha `f` indicada.
   Precondición: La fecha debe ser posterior a la fecha de hoy, es decir, `f > fechaActualPN(p)`. Además, la persona `x` no puede pedir más de un regalo para la misma fecha.

4. `Estado estadoPedidoPN(PapaNoel p, Id x)`
   Propósito: Devuelve el estado del pedido de la persona `x` en la fecha de hoy. El resultado debe ser uno de los siguientes cuatro valores:

   - `Inexistente` — si la persona no pidió un regalo para la fecha de hoy.
   - `Pendiente` — si la persona pidió un regalo para la fecha de hoy pero su comportamiento todavía no fue monitoreado.
   - `Entregado` — si la persona pidió un regalo para la fecha de hoy y el regalo se entregó, con motivo de su buen comportamiento.
   - `MalComportamiento` — si la persona pidió un regalo para la fecha de hoy y el regalo no fue entregado, con motivo de su mal comportamiento.

5. `void entregarPedidoPN(PapaNoel p, Id x)`
   Propósito: Entrega un regalo a la persona `x`, con motivo de su buen comportamiento, cambiando su estado a `Entregado`.
   Precondición: La persona tiene que tener un regalo pendiente para la fecha de hoy, es decir se debe cumplir que `estadoPedidoPN(p, x) == Pendiente`.

6. `void registrarMalComportamientoPN(PapaNoel p, Id x)`
   Propósito: Registra el mal comportamiento de la persona `x`, cambiando su estado a `MalComportamiento`.
   Precondición: La persona tiene que tener un regalo pendiente para la fecha de hoy, es decir se debe cumplir que `estadoPedidoPN(p, x) == Pendiente`.

7. `void avanzarDiaPN(PapaNoel p)`
   Propósito: Finaliza el día actual, incrementando la fecha de hoy en 1.
   Precondición: No debe haber ninguna persona en estado `Pendiente`.

8. `void finalizarPN(PapaNoel p)`
   Propósito: Libera toda la memoria reservada para el sistema.

2.2. Restricciones de complejidad

Las operaciones deben tener las siguientes complejidades temporales en peor caso:

1. `PapaNoel iniciarPN()` — $O(1)$

2. `Fecha fechaActualPN(PapaNoel p)` — $O(1)$

3. `void registrarPedidoPN(PapaNoel p, Fecha f, Id x)` — $O(\log P)$
   Donde P es la cantidad total de pedidos cargados en el sistema.

4. `Estado estadoPedidoPN(PapaNoel p, Id x)` — $O(\log H)$
   Donde H es la cantidad total de pedidos para la fecha de hoy.

5. `void entregarPedidoPN(PapaNoel p, Id x)` — $O(\log H)$

6. `void registrarMalComportamientoPN(PapaNoel p, Id x)` — $O(\log H)$

7. `void avanzarDiaPN(PapaNoel p)` — $O(H + M \cdot \log P)$
   Donde P es la cantidad total de pedidos cargados en el sistema, H es la cantidad de pedidos para la fecha de hoy (antes de avanzar el día) y M es la cantidad de pedidos registrados para la fecha de mañana.

8. `void finalizarPN(PapaNoel p)` — (Sin restricciones de complejidad).

2.3. Detalles de implementación

2.3.1. Estructura de representación

El sistema debe estar implementado de acuerdo con la representación que se indica a continuación. En primer lugar, un `Pedido` es un registro que cuenta con el identificador de una persona, una fecha en la que debe ser entregado, y un indicador de estado:

```
struct Pedido {
    Id persona;
    Fecha fechaEntrega;
    Estado estado;
};
```

Contaremos también con un tipo `ColaPedidos` que representa una cola de prioridad cuyos elementos son pedidos. Si tenemos dos pedidos `p1` y `p2` consideramos que `p1` tiene más prioridad que `p2` si la fecha de entrega de `p1` es anterior a la fecha de entrega de `p2`. En caso de tener la misma fecha de entrega, como criterio de "desempate" se utiliza el identificador de la persona. Es decir:

```
// Devuelve true si p1 tiene mayor prioridad que p2.
bool masPrioritario(Pedido p1, Pedido p2) {
  return p1.fechaEntrega < p2.fechaEntrega ||
         (p1.fechaEntrega == p2.fechaEntrega && p1.persona < p2.persona);
}
```

Los detalles del tipo abstracto `ColaPedidos` se describen en la próxima sección.

El tipo `PapaNoel` se representará internamente como un puntero a una estructura `PapaNoelRepr`.

```
typedef PapaNoelRepr* PapaNoel;

struct PapaNoelRepr {
    int fechaDeHoy;
    ColaPedidos pedidosFuturos;
    int cantPedidosDeHoy;
    Pedido* pedidosDeHoy;
};
```

La estructura `PapaNoelRepr` cumple con el siguiente **invariante de representación**:

1. La fecha de hoy es un entero no negativo (`fechaDeHoy >= 0`).
2. Todos los pedidos en la cola `pedidosFuturos` tienen fecha de entrega posterior a `fechaDeHoy`.
3. Todos los pedidos en la cola `pedidosFuturos` tienen estado `Pendiente`.
4. En la cola `pedidosFuturos` no puede haber dos pedidos para la misma persona con la misma fecha de entrega.
5. El puntero `pedidosDeHoy` apunta a un arreglo con capacidad para almacenar al menos `cantPedidosDeHoy` elementos.
6. Todos los pedidos del arreglo `pedidosDeHoy` tienen fecha de entrega igual a `fechaDeHoy`.
7. Todos los pedidos del arreglo `pedidosDeHoy` tienen estado `Pendiente`, `Entregado` o `MalComportamiento` (pero no pueden tener estado `Inexistente`).
8. Los pedidos del arreglo `pedidosDeHoy` se encuentran ordenados de menor a mayor por el identificador de la persona correspondiente.

2.3.2. Algoritmos

Describimos brevemente cómo realizar cada una de las funciones pedidas utilizando la estructura sugerida:

1. `PapaNoel iniciarPN()` — Reservar memoria para una nueva estructura con `fechaDeHoy = 0`, con una cola vacía y un arreglo vacío.
2. `Fecha fechaActualPN(PapaNoel p)` — Devolver `fechaDeHoy`.
3. `void registrarPedidoPN(PapaNoel p, Fecha f, Id x)` — Registrar el pedido en la cola de prioridad.
4. `Estado estadoPedidoPN(PapaNoel p, Id x)` — Buscar un pedido asociado a la persona en el arreglo haciendo **búsqueda binaria**. Si no tiene un pedido asociado, retornar `Inexistente`. Si tiene un pedido asociado, retornar su estado.
5. `void entregarPedidoPN(PapaNoel p, Id x)` — Buscar el pedido asociado a la persona en el arreglo haciendo búsqueda binaria y cambiar su estado a `Entregado`.
6. `void registrarMalComportamientoPN(PapaNoel p, Id x)` — Buscar el pedido asociado a la persona en el arreglo haciendo búsqueda binaria y cambiar su estado a `MalComportamiento`.
7. `void avanzarDiaPN(PapaNoel p)` — Borrar el arreglo de pedidos de la fecha de hoy. Incrementar el valor de `fechaDeHoy`. Desencolar todos pedidos de la cola, mientras correspondan a la nueva fecha. Crear un nuevo arreglo con dichos pedidos.

   Nota: no se conoce la cantidad de pedidos de antemano. Al sacarlos de la cola de prioridad, se sugiere guardarlos en un arreglo temporal, duplicando el tamaño de dicho arreglo cuando sea necesario.
8. `void finalizarPN(PapaNoel p)` — Liberar toda la memoria reservada.

3. Cola de pedidos

La cola de pedidos es un tipo abstracto de datos que permite encolar pedidos y desencolar el pedido de mayor prioridad. La prioridad está dada de acuerdo con la función `masPrioritario` ya descripta anteriormente.

3.1. Interfaz del tipo abstracto `ColaPedidos`

1. `ColaPedidos nuevaCP()`
   Propósito: Crea una nueva cola.
   Eficiencia: $O(1)$.
2. `int tamCP(ColaPedidos cp)`
   Propósito: Devuelve la cantidad de pedidos en la cola.
   Eficiencia: $O(1)$.
3. `void encolarCP(ColaPedidos cp, Pedido p)`
   Propósito: Encola un pedido.
   Eficiencia: $O(\log n)$, donde n es la cantidad de pedidos en la cola.
4. `Pedido proximoCP(ColaPedidos cp)`
   Propósito: Devuelve el pedido de la cola que tiene mayor prioridad. *Nota:* No modifica la cola de prioridad.
   Precondición: La cola no puede estar vacía (es decir, `tamCP(cp) > 0`).
   Eficiencia: $O(1)$.
5. `void desencolarCP(ColaPedidos cp)`
   Propósito: Desencola el pedido de mayor prioridad.
   Precondición: La cola no puede estar vacía (es decir, `tamCP(cp) > 0`).
   Eficiencia: $O(\log n)$, donde n es la cantidad de pedidos en la cola.
6. `void destruirCP(ColaPedidos cp)`
   Propósito: Libera toda la memoria reservada para la cola de prioridad.
   Eficiencia: (Sin restricciones de eficiencia).

3.2. Detalles de implementación

3.2.1. Estructura de representación

El tipo `ColaPedidos` se representará internamente como un puntero a una estructura `ColaPedidosRepr`. Dicha estructura tiene un único puntero `raiz` a un `Nodo`. Los nodos representan un árbol binario.

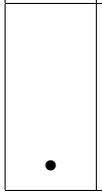
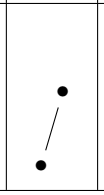
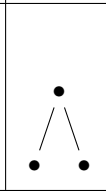
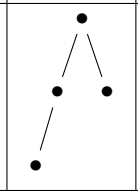
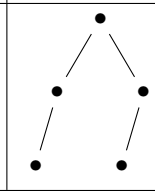
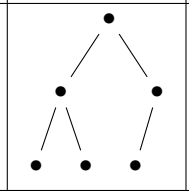
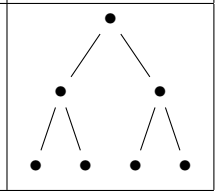
```
typedef ColaPedidosRepr* ColaPedidos;

struct Nodo {
    int tam;
    Pedido pedido;
    Nodo* izq;
    Nodo* der;
};

struct ColaPedidosRepr {
    Nodo* raiz;
};
```

La estructura `ColaPedidosRepr` se representa con un **Braun heap**. Más precisamente, cumple con el siguiente **invariante de representación**:

1. Los nodos con punteros forman un árbol binario.
2. El campo `tam` de cada nodo representa el tamaño de dicho subárbol (es decir, la cantidad de nodos).
3. El árbol tiene forma de **árbol de Braun**. Un árbol vacío es un árbol de Braun. Un árbol no vacío es un árbol de Braun si tiene las siguientes características:
   - El hijo izquierdo es del mismo tamaño o a lo sumo un nodo más grande que su hijo derecho. Más precisamente, si llamamos k_1 a la cantidad de nodos del hijo izquierdo y k_2 a la cantidad de nodos del hijo derecho, entonces se cumple que $k_1 = k_2$ o que $k_1 = k_2 + 1$.
   - Recursivamente, el hijo izquierdo y el hijo derecho son árboles de Braun.
4. Los nodos del árbol están ordenados con el invariante de **heap**. Un árbol vacío es un heap. Un árbol no vacío es un heap si tiene las siguientes características:
   - La raíz tiene mayor prioridad que el resto de los nodos.
   - Recursivamente, el hijo izquierdo y el hijo derecho son heaps.

Ejemplo: árboles de Braun.

Los siguientes son árboles de Braun de tamaño n:

$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$
						

Observar que si n es impar, entonces n se puede escribir como $n = 1 + k + k$, es decir, el hijo izquierdo y el derecho tienen ambos tamaño k. Si n es par, entonces $n = 1 + k + (k - 1)$, es decir, el hijo izquierdo tiene tamaño k y el hijo derecho tiene tamaño $k - 1$.

Se puede probar que un árbol de Braun con n nodos tiene altura $O(\log n)$.

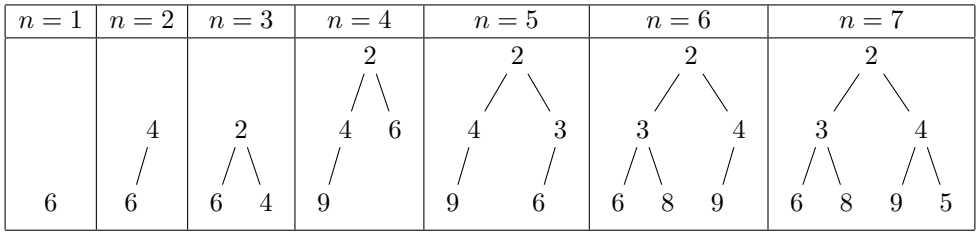
3.2.2. Algoritmos

Describimos únicamente los dos algoritmos interesantes que son el de inserción (`encolarCP`) y borrado (`desencolarCP`).

Inserción en un Braun heap. Queremos insertar un elemento x en un árbol A respetando el invariante de Braun heap.

- Si el árbol A está vacío, el árbol pasa a tener un único nodo con el elemento x en la raíz.
- Si el árbol A no está vacío, tiene tamaño n, una raíz y, un hijo izquierdo I y un hijo derecho D.
  1. Si el elemento que se desea insertar (x) tiene más prioridad que la raíz del árbol (y), intercambiar los valores de x e y.
  2. Insertar recursivamente x en el árbol derecho D.
  3. Si el tamaño original del árbol (n) era impar, intercambiar el hijo izquierdo I con el hijo derecho D.
  4. Incrementar el campo `tam`.

Por ejemplo, si insertamos sucesivamente los números $6, 4, 2, 9, 3, 8, 5$, suponiendo que un número más chico tiene mayor prioridad, obtenemos:



Sugerencia: definir una función auxiliar recursiva:

```
Nodo* insertarEnBraunHeap(Nodo* n, Pedido p)
```

que inserta el pedido `p` en un Braun heap `n` y devuelve el Braun heap resultante.

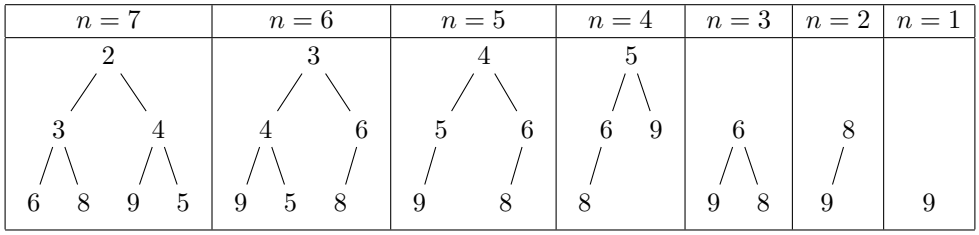
Borrado de un Braun heap. Queremos eliminar el elemento de mayor prioridad (la raíz), respetando el invariante de Braun heap. Si el árbol original tiene n nodos, el primer paso es eliminar algún nodo para obtener un árbol de $n-1$ nodos que tenga la forma correcta, es decir, que siga siendo un árbol de Braun. Para ello:

1. Si el árbol tiene exactamente un nodo, eliminar ese nodo. El árbol queda vacío.
2. Si el árbol tiene al menos dos nodos, eliminar recursivamente un nodo del hijo izquierdo I.
3. Si el tamaño original del árbol (n) era impar, intercambiar el hijo izquierdo con el derecho.
4. Decrementar el campo `tam`.

Supongamos que el nodo que se eliminó tiene valor x y que la raíz del árbol tiene valor y. El algoritmo procede del siguiente modo:

5. Sobreescribir el valor y de la raíz con el valor x que tenía el nodo que se acaba de eliminar.
6. Aplicar el procedimiento de "bajar la raíz" como es usual en un heap, es decir: mientras la raíz tenga menor prioridad que alguno de sus hijos:
   - Intercambiar la raíz con su hijo más prioritario.
   - Continuar aplicando el mismo procedimiento en dicho hijo.

Por ejemplo, si eliminamos sucesivamente los elementos del heap anteriormente obtenido, nos queda:



Sugerencia: definir una función auxiliar recursiva para implementar los pasos 1..4 de arriba.

```
Nodo* eliminarDeBraunHeap(Nodo* n, Pedido &p)
```

Esta función **no** lee el valor del parámetro `p`. El parámetro `p` se utiliza únicamente como parámetro de **salida**, para guardar el valor que tenía el nodo que se eliminó.

4. Pautas de entrega

La entrega consistirá en enviar la misma carpeta disponible en el repositorio de la materia, pero con los archivos `PapaNoel.cpp` y `ColaPedidos.cpp` completos de acuerdo con lo pedido con el enunciado. La carpeta se debe comprimir en formato `zip`. Antes de comprimir la carpeta, eliminar todos los archivos binarios (ejecutables `.exe` y objetos `.o`), dejando únicamente los archivos fuente (`.h` y `.cpp`). El envío se realiza por mail a la lista de docentes de la materia[1]. Tanto el nombre del zip como el asunto del mail deberán ser: **ED2018s1-TPFinal-COMISION-APELLIDO-NOMBRE**. Ejemplo: "ED2018s1-TPFinal-C1-Ramirez-Soledad".

4.1. Forma de evaluación

Evaluaremos fundamentalmente los siguientes aspectos:

1. Defensa del trabajo. Tendrás que ser capaz de demostrar que entendés tu trabajo. Está bien buscar ideas en internet o consultar con tus compañeros, pero tenés que *demostrar* que entendés lo que estás entregando. Luego de la entrega se tomará un ejercicio adicional, trivial, para demostrar que entendiste de qué trata tu código. Este ejercicio adicional es tan importante como el TP. No es posible aprobar el TP sin realizar correctamente dicho ejercicio. Entonces, procurá entender tu código, ya sea completamente de tu autoría o hecho con ayuda de terceros.

2. El proyecto debe compilar correctamente. Si esto no sucede, el trabajo se considera desaprobado, sin posibilidades de ser defendido. Consejo: **compilá todo el tiempo tu código**. No avances con otras cuestiones hasta que no hayas logrado esto, y pedí ayuda en lo que respecta al lenguaje y las herramientas que usamos.

3. La implementación correcta de la funcionalidad pedida. Recordá que podés pedir ayuda tanto como sea necesario. Pero dicho esto, la implementación debe **ejecutar correctamente** y debe pasar todos los tests provistos en el archivo `Test.cpp`. De no hacerlo, te damos el tiempo de la defensa del TP para que puedas corregirlo. Si no llegás a completar en el lapso estipulado una implementación correcta, el trabajo se considera desaprobado.

4. El estilo de la materia. Que tu implementación compile y funcione correctamente no es garantía de que hayas utilizado los conceptos vistos en la materia. Es muy importante que, por ejemplo, respetes barreras de abstracción, indiques los invariantes de representación, precondiciones, etc. Un trabajo con graves faltas en este sentido no estará aprobado con buena nota, e incluso puede llegar a estar desaprobado, si es que no cumple con los lineamientos de la materia.

[1] `tpi-doc-ed@listas.unq.edu.ar`