Trabajo Práctico
Estructuras de Datos

Universidad Nacional de Quilmes

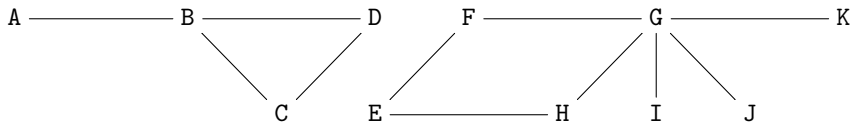
Primer semestre 2019

Fecha de la defensa: sábado 13 de julio de 2018.

Índice

1. Introducción

Este trabajo consiste en implementar en C++ un tipo abstracto para describir y operar con *redes de elementos*. Una red de elementos es una estructura en la que los elementos pueden estar conectados entre sí. Por ejemplo, una red ferroviaria consta de estaciones conectadas por vías, mientras que una red de computadoras consta de servidores conectados por cables. Por ejemplo, una red puede tener el siguiente aspecto:



En C++ una red se representará con un valor del tipo abstracto `Network`. Para poder implementar el tipo `Network` nos concentraremos primero en la implementación de dos tipos auxiliares: el tipo `Stack`, que sirve para manipular pilas de elementos, y el tipo `Map`, que sirve para manipular diccionarios, es decir, asociaciones entre claves y valores.

En las secciones siguientes describimos la interfaz de los tipos `Stack`, `Map` y `Network`, y detallamos cuál es la representación que deben usar para implementarlos. Notar que en este TP está **prohibido** modificar las interfaces de los tipos abstractos, es decir, únicamente se deben modificar los archivos con extensión `.cpp`, mientras que los archivos con extensión `.h` no se deben modificar por ningún motivo.

2. El tipo `Stack`

El tipo `Stack` sirve para representar pilas de números enteros.

2.1. Interfaz de `Stack`

1. `Stack emptyS()` — Crea una pila vacía. *Eficiencia:* debe ser $O(1)$.
2. `void pushS(Stack s, int x)` — Apila `x` en la pila. *Eficiencia:* si la pila se encuentra vacía, el costo de insertar n elementos debe ser $O(n)$ en peor caso[1].
3. `int sizeS(Stack s)` — Devuelve el tamaño de la pila. *Eficiencia:* debe ser $O(1)$.
4. `int topS(Stack s)` — Devuelve el tope de la pila. *Precondición:* la pila no está vacía. *Eficiencia:* debe ser $O(1)$.
5. `void popS(Stack s)` — Desapila el tope de la pila. *Precondición:* la pila no está vacía. *Eficiencia:* debe ser $O(1)$.
6. `void destroyS(Stack s)` — Libera toda la memoria usada por la pila.

2.2. Implementación de `Stack`

La pila se puede implementar de la manera que prefieran, siempre que se respeten las condiciones de eficiencia. Sugerimos alguna de las dos opciones siguientes: (1) lista simplemente enlazada, ó (2) lista sobre arreglo (*array list*).

2.2.1. Opción 1: lista enlazada

Recordar que una lista enlazada se representa con un entero que indica su tamaño, junto con un puntero al nodo que está asociado al tope de la pila:

```
struct Nodo {
    int valor;
    Nodo* siguiente;
};

struct StackRepr {
    int tam;
    Nodo* tope;
};
```

2.2.2. Opción 2: lista sobre arreglo

Recordar que un *array list* se representa con un arreglo, acompañado de una **capacidad** y un **tamaño**:

```
struct StackRepr {
    int* valores;
    int capacidad;
    int tam;
};
```

1. El puntero **valores** es un arreglo de elementos. Los primeros elementos representan el contenido de la pila, de tal modo que `elementos[tamaño - 1]` es el tope de la pila y `elementos[0]` es el fondo de la pila.
2. La **capacidad** es un número positivo que representa el tamaño total del arreglo en el que se almacenan los elementos de la pila, y donde puede haber espacio disponible para almacenar más elementos. La capacidad es siempre una potencia de 2.
3. El **tamaño** es un número entre 0 y `capacidad` que representa el tamaño real de la pila, es decir la cantidad de elementos que contiene.

Cuando se apila un elemento en la pila, se debe verificar si la pila se encuentra llena (es decir `tam == capacidad`), en cuyo caso se debe reservar espacio para un arreglo del doble de la capacidad actual, y copiar los elementos actualmente contenidos en la pila al nuevo arreglo.

[1] Esto es lo que se conoce como costo **amortizado** $O(1)$.

3. El tipo `Map`

El tipo `Map` sirve para representar diccionarios, es decir asociaciones de claves a valores. En este TP, tanto las claves como los valores son números enteros. Por claridad definimos en el archivo `Map.h` los siguientes renombres de tipo, `Key` para el tipo de las claves y `Value` para el tipo de los valores:

```
typedef int Key;
typedef int Value;
```

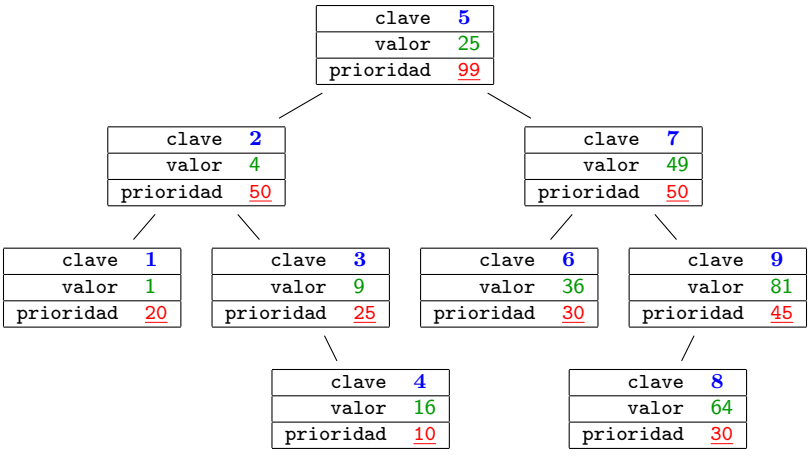
3.1. Interfaz de `Map`

La interfaz del diccionario se detalla a continuación. Al momento de expresar la complejidad de las operaciones, n representa la cantidad de elementos del diccionario. Las complejidades de inserción, búsqueda y borrado corresponden al **caso promedio**, de acuerdo con lo que se detalla en la implementación de `Map`:

1. `Map emptyM()` — Crea un diccionario vacío. *Eficiencia:* debe ser $O(1)$.
2. `int sizeM(Map m)` — Devuelve el tamaño del diccionario, es decir, el número de claves distintas. *Eficiencia:* debe ser $O(1)$.
3. `void insertM(Map m, Key k, Value v)` — Inserta una clave `k` en el diccionario, y la asocia al valor `v`. Si la clave ya se encuentra definida, sobreescribe su valor. *Eficiencia:* debe ser $O(\log n)$ en caso promedio.
4. `bool lookupM(Map m, Key k, Value& v)` — Si la clave `k` está definida en el diccionario, devuelve `true` y modifica el valor del parámetro `v` para que sea el valor asociado a `k`. Si la clave no se encuentra definida en el diccionario, devuelve `false`. *Eficiencia:* debe ser $O(\log n)$ en caso promedio.
5. `void removeM(Map m, Key k)` — Elimina la clave `k` del diccionario. Si la clave no se encuentra definida en el diccionario, esta operación no tiene ningún efecto. *Eficiencia:* debe ser $O(\log n)$ en caso promedio.
6. `void destroyM(Map m)` — Libera toda la memoria usada por el diccionario.

3.2. Implementación de `Map`

En este TP pedimos que el `Map` se represente sobre una estructura de datos conocida como *treap*. Un treap es un árbol binario, en el que cada nodo tiene una **clave**, un valor, una prioridad, y punteros a sus dos hijos. Por ejemplo, el siguiente es un treap:



El treap de arriba representa el diccionario que asocia las siguientes **claves** a los siguientes valores:

$$\mathbf{1} \mapsto 1 \quad \mathbf{2} \mapsto 4 \quad \mathbf{3} \mapsto 9 \quad \mathbf{4} \mapsto 16 \quad \mathbf{5} \mapsto 25 \quad \mathbf{6} \mapsto 36 \quad \mathbf{7} \mapsto 49 \quad \mathbf{8} \mapsto 64 \quad \mathbf{9} \mapsto 81$$

El invariante de un treap combina el invariante de orden en un árbol binario de búsqueda (BST) para las **claves** junto con el invariante de orden en un heap para las prioridades. Observar en el ejemplo de arriba que todas las claves que están a la izquierda de **5** son más chicas (**1**, **2**, **3**, **4**) y todas las claves que están a la derecha son más grandes (**6**, **7**, **8**, **9**), tal como en un árbol binario de búsqueda. Por otro lado, las prioridades que están abajo de 99 son todas más chicas que 99 (10, 20, 25, 30, 45, 50), tal como en un heap.

3.2.1. ¿De dónde salen las prioridades y para qué sirven?

La prioridad de un nodo es un número que se elige "al azar" cada vez que se inserta un elemento en el treap. El objetivo de incorporar prioridades elegidas al azar es conseguir que el árbol quede balanceado. En la materia ya estudiamos el caso de los árboles AVL, que garantizan que el árbol queda **siempre** balanceado, pero con la desventaja de que los algoritmos de inserción y borrado son bastante complejos.

En el caso de los treaps, los algoritmos de inserción y borrado son bastante más sencillos que en el caso de los AVLs. La desventaja de los treaps es que el balanceo puede depender de la **suerte** que uno haya tenido al elegir aleatoriamente las prioridades. Si uno tiene mala suerte, el árbol puede quedar muy desbalanceado. Se puede justificar, usando argumentos probabilísticos, que en promedio un treap de n nodos tendrá altura $O(\log n)$. En general uno debería tener **muy** mala suerte para que la altura treap sea peor que logarítmica.

En este trabajo nos limitaremos a implementar un map sobre treap, sin estudiar con mayor profundidad su eficiencia.

3.2.2. Estructura e invariante de representación de un treap en C++

Para representar las prioridades en C++ usaremos el tipo `Priority`:

```
typedef int Priority;
```

En los archivos que proveemos como esqueleto para implementar el TP se encuentra ya definida una función `Priority randomPriority()` que devuelve una prioridad "al azar". Más precisamente, cada vez que se invoca a esta función, devuelve un número *pseudoaleatorio* (elegido de acuerdo con algún procedimiento del que no nos ocuparemos).

La estructura para representar el `Map` será la siguiente:

```
struct Node {
  Key clave;
  Value valor;
  Priority prioridad;
  Node* hijoIzq;
  Node* hijoDer;
};

struct MapRepr {
    Node* raiz;
    int tam;
};
```

Los punteros `raiz`, `hijoIzq` e `hijoDer` pueden ser `NULL` en caso de que los correspondientes árboles sean vacíos. El invariante es el siguiente:

1. **Tamaño.** El número `tam` coincide con la cantidad de nodos del árbol.
2. **Invariante de BST.** La **clave** de un nodo siempre es estrictamente mayor que la de su hijo izquierdo y estrictamente menor que la de su hijo derecho.
3. **Invariante de heap.** La prioridad de un nodo siempre es estrictamente mayor que la de cualquiera de sus hijos.

3.2.3. Algoritmo de inserción en un treap

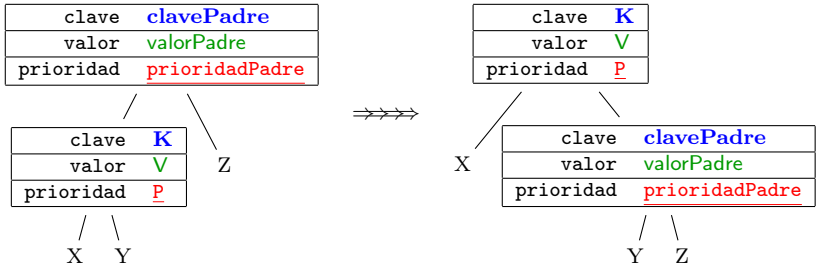
Supongamos que tenemos un treap y queremos insertar una clave **K** con valor V en el treap. Para ello seguimos los siguientes pasos:

1. Nos posicionamos sobre la raíz del árbol y bajamos hacia el lugar en el que tocaría estar ubicada a la clave **K**, de acuerdo con el invariante de BST. Más precisamente, si estamos posicionados sobre un nodo cuya clave es mayor que **K**, debemos bajar hacia la izquierda, y si la clave es menor que **K**, debemos bajar hacia la derecha. Este proceso se repite hasta que llegamos a un nodo que tiene clave **K** o hasta que llegamos a una hoja.
2. Si encontramos un nodo que tiene clave **K**, pisamos su valor con V y damos por finalizada la inserción.

3. En el caso contrario, si llegamos a una hoja, insertamos en esa posición un nodo nuevo con clave **K**, valor V y con una prioridad P elegida al azar, usando la función `randomPriority()`. **Notar:** *En este punto del algoritmo insertamos un nodo respetando el invariante de BST, pero podríamos estar violando el invariante de heap, porque el nodo que acabamos de insertar podría tener mayor* ***prioridad*** *que su padre.*

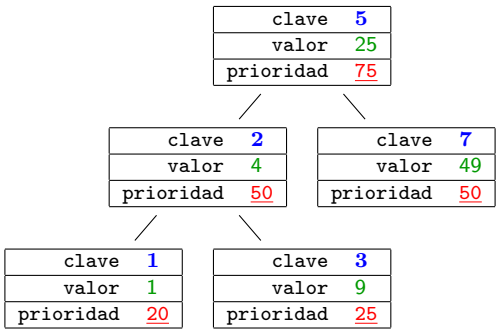
4. Para asegurarnos de cumplir con el invariante de heap, debemos ahora "subir" el nodo para asegurarnos de que no tenga mayor prioridad que su padre.

   Supongamos que el nodo en cuestión es el hijo izquierdo de su padre, y supongamos que tiene mayor prioridad que su padre. En tal caso efectuamos una **rotación**:

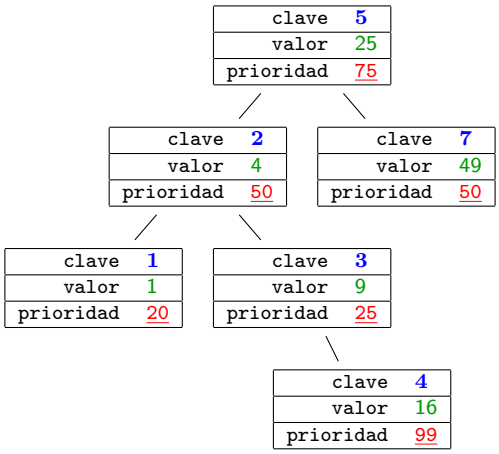


   Si el nodo en cuestión es el hijo derecho de su padre y tiene mayor prioridad que su padre, efectuamos también una rotación (hacia el otro lado). Repetimos este proceso, subiendo el nodo mientras tenga mayor prioridad que su padre, o hasta que lleguemos a la raíz.

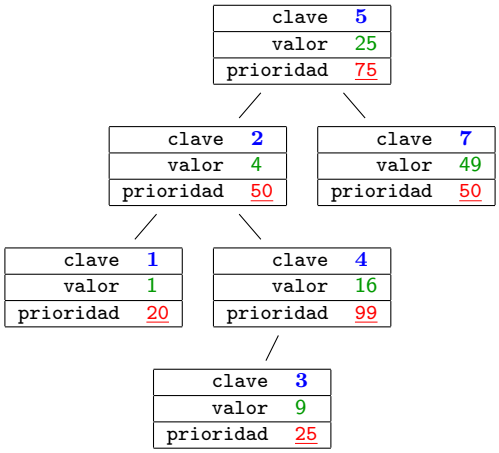
Ejemplo de inserción. Supongamos que queremos insertar la clave **4** con valor 16 en el siguiente treap:



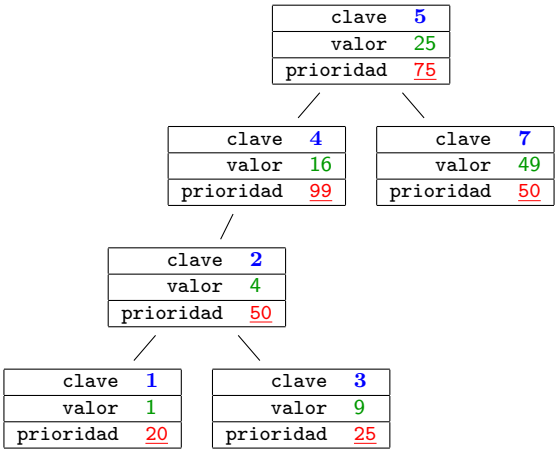
Supongamos que la función `randomPriority()` elige como prioridad 99. El primer paso es insertar un nuevo nodo en el treap con el algoritmo de inserción en un BST:



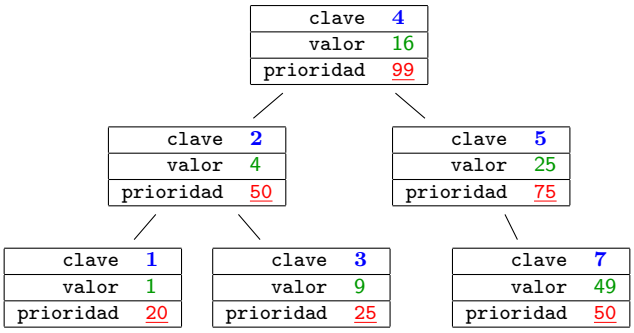
Subimos el nodo por primera vez, porque tiene mayor prioridad que su padre, haciendo una rotación:



Subimos el nodo por segunda vez, porque tiene mayor prioridad que su padre, haciendo una rotación:



Subimos el nodo por tercera (y última) vez, porque tiene mayor prioridad que su padre, haciendo una rotación:



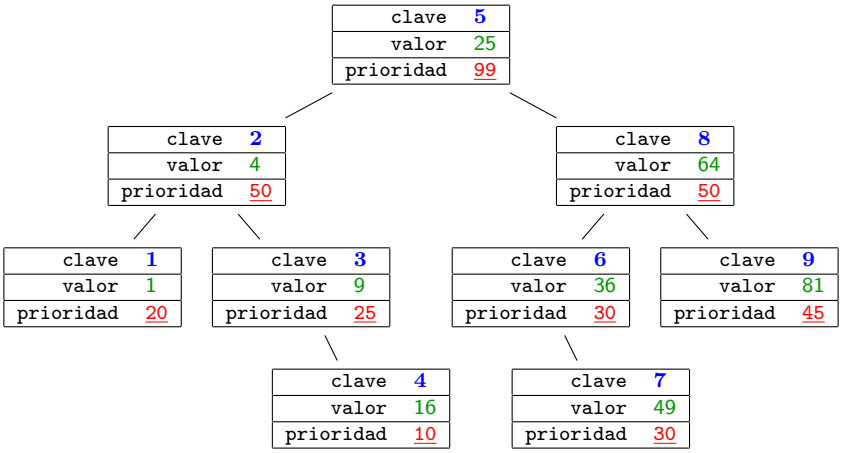
3.2.4. Algoritmo de búsqueda en un treap

El algoritmo de búsqueda sigue la misma lógica que en el caso de los BSTs. Para buscar una clave, se busca el nodo que le corresponda, bajando por el árbol de acuerdo con el invariante de BST.

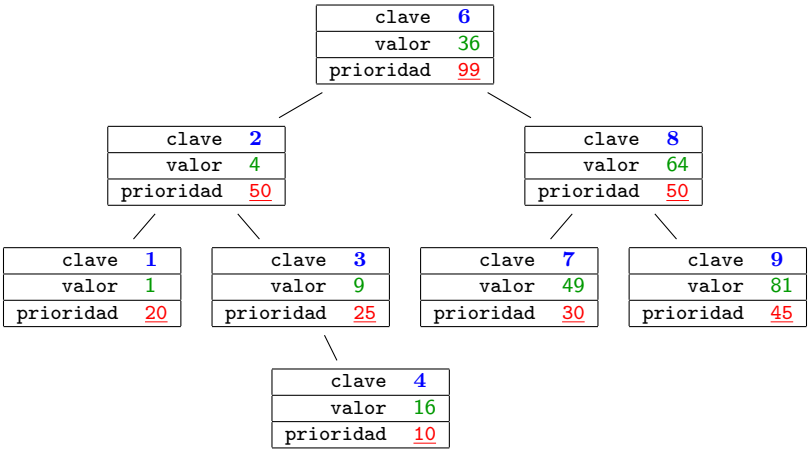
3.2.5. Algoritmo de borrado de un treap

Utilizaremos una variante simplificada del algoritmo de borrado, que sigue la misma lógica que en el caso de los BSTs. Para borrar una clave, se ubica primero el nodo correspondiente, y se ubica a su nodo **sucesor**. El **sucesor** de un nodo X es el nodo que aparece en el hijo derecho de X y que tiene la clave más chica. Una vez ubicado el sucesor, se reemplaza la clave de X por la clave de su sucesor, se reemplaza el valor de X por el valor de su sucesor, y se elimina al nodo sucesor. Observar que **no** se reemplaza la `prioridad` de X.

Ejemplo de borrado. Si queremos eliminar la clave **5** del siguiente treap, identificamos a su nodo sucesor, que es el nodo con clave **6**.



Reemplazamos la **clave** y el valor del nodo original con la clave y el valor de su sucesor, pero **sin** alterar la prioridad, y eliminamos al sucesor:



4. El tipo `Network`

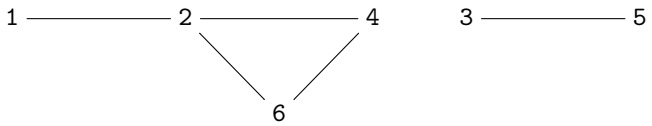
El tipo `Network` sirve para representar redes de elementos que pueden estar conectados. Los elementos de la red se identifican con un número que por claridad representamos con el siguiente tipo, ya definido en el archivo `Network.h`:

```
typedef int Id;
```

4.1. Interfaz de `Network`

La interfaz de la red de elementos se detalla a continuación. Una red de elementos permite hacer las siguientes operaciones: agregar un elemento, conectar **directamente** dos elementos, preguntar si dos elementos están **conectados** y deshacer una acción.

Dos elementos están **conectados** si se puede llegar del uno al otro pasando por conexiones directas. Por ejemplo, en la siguiente red hay seis elementos numerados de 1 a 6 con las conexiones que se muestran en el gráfico. Los elementos 1 y 2 están **directamente conectados**, mientras que 1 y 6 están **conectados**, pero la conexión no es directa. Por otro lado, los elementos 1 y 3 no están conectados.



Una característica importante de la red es que permite aplicar una acción `undoN` para deshacer la última acción.

1. `Network emptyN()` — Crea una red vacía.
2. `void addN(Network net, Id id)` — Agrega un elemento a la red, con el identificador `id`. *Precondición:* el elemento no está en la red aún.
3. `bool elemN(Network net, Id id)` — Devuelve verdadero si el elemento está en la red.
4. `void connectN(Network net, Id id1, Id id2)` — Establece una conexión **directa** entre dos elementos de la red. Si los elementos ya están conectados (ya sea directa o indirectamente), esta acción no tiene efecto. *Precondición:* `id1` e `id2` identifican a dos elementos de la red.
5. `bool connectedN(Network net, Id id1, Id id2)` — Devuelve verdadero si `id1` e `id2` identifican a dos elementos de la red que están conectados, ya sea directa o indirectamente. *Precondición:* `id1` e `id2` identifican a dos elementos de la red.
6. `void undoN(Network net)` — Deshace la última acción, que puede consistir en agregar un elemento o en conectar dos elementos. *Precondición:* debe haber habido una última acción.
7. `void destroyN(Network net)` — Libera toda la memoria usada por la red de elementos.

Ejemplo. Por ejemplo, la siguiente secuencia de comandos tiene el siguiente efecto:

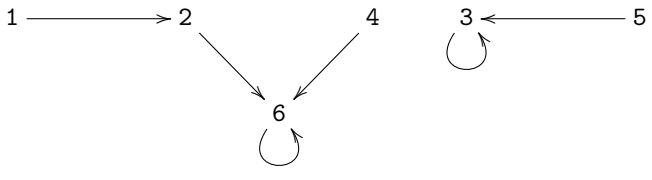
`Network n = emptyN();`	
`addN(n, 10);`	10
`addN(n, 20);`	10 20
`connectN(n, 10, 20);`	10 —— 20
`addN(n, 30);`	10 —— 20 30
`connectN(n, 20, 30);`	10 —— 20 —— 30
`undoN(n);`	10 —— 20 30
`undoN(n);`	10 —— 20
`undoN(n);`	10 20

4.2. Implementación de `Network`

Usaremos una variante simplificada de la estructura conocida como `union-find`. La estructura cuenta con dos campos: por un lado un map que representa la estructura de la red, y por otro lado una pila de números que guarda información para deshacer las acciones. En C++:

```
struct NetworkRepr {
    Map flechas;
    Stack acciones;
};
```

Los elementos de la red están dados por las claves del diccionario `flechas`. El diccionario `flechas` representa "flechas" entre elementos. De cada elemento sale exactamente una flecha, pero a un elemento pueden llegar ninguna, una, o más flechas. Por ejemplo, el diccionario $\{(1 \mapsto 2), (2 \mapsto 6), (3 \mapsto 3), (4 \mapsto 6), (5 \mapsto 3), (6 \mapsto 6)\}$ se puede visualizar así:



Se describe brevemente el mecanismo para implementar las operaciones de la interfaz:

- `addN` — Para agregar un elemento x a la red, agregar en el diccionario un clave x con una flecha $x \to x$ en la que x apunta a sí mismo. Meter x en la pila para poder deshacer esta acción.
- `connectN` — Para conectar dos elementos x, y:
  1. Seguir la flecha que sale de x, repetidamente $x \to x_1 \to x_2 \to \ldots \to x_n \to x'$ hasta llegar a un elemento x' que tiene una flecha hacia sí mismo $x' \to x'$.

2. Análogamente, seguir la flecha que sale de y, repetidamente $y \to y_1 \to y_2 \to \ldots \to y_m \to y'$ hasta llegar a un elemento y' que tiene una flecha hacia sí mismo $y' \to y'$.

Si $x' \neq y'$, reemplazar en el diccionario la flecha $x' \to x'$ por una flecha $x' \to y'$. Si $x' = y'$, no hay que hacer ninguna modificación en el diccionario. En cualquier caso, se debe meter x' en la pila para poder deshacer esta acción.

- `connectedN` — Para determinar si dos elementos están conectados, igual que en el caso anterior seguir la flecha que sale de x hasta llegar a un elemento x' que apunta a sí mismo. Análogamente, seguir la flecha que sale de y hasta llegar a un elemento y' que apunta a sí mismo. Los elementos x e y están conectados si y solamente si $x' = y'$.

- `undoN` — Para deshacer la última acción, mirar el elemento x que está en el tope de la pila de acciones y sacarlo de la pila. Si el elemento apunta a sí mismo, es decir $x \to x$ en el diccionario de flechas, la última acción consistió en agregar el elemento x, de modo que eliminamos x del diccionario de flechas. Si en cambio tenemos que $x \to y$ con $x \neq y$, la última acción consistió en establecer una conexión desde x hacia y, de modo que reemplazamos la flecha $x \to y$ por una flecha $x \to x$ en la que x apunta a sí mismo.

5. Pautas de entrega

La entrega consistirá en enviar la misma carpeta disponible en el repositorio de la materia, pero con los archivos `Stack.cpp`, `Map.cpp` y `Network.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[2]. Tanto el nombre del zip como el asunto del mail deberán ser: **ED2019s1-TPFinal-COMISION-APELLIDO-NOMBRE**. Ejemplo: "ED2018s1-TPFinal-C1-Ramirez-Soledad".

5.1. 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.

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