Tarea 3
TADs Pila, Cola y Conjunto
Curso 2023

Esta tarea tiene como principales objetivos:

- Continuar trabajando sobre el manejo dinámico de memoria.
- Trabajar con el concepto de Tipo Abstracto de Datos (TAD).
- Trabajar en la implementación de TADs a partir de su especificación y utilizando estructuras vistas con anterioridad.
- Trabajar en el uso de TADs como auxiliares para la resolución de problemas.

La fecha límite de entrega es el **jueves 25 de mayo a las 16:00 horas**. El mecanismo específico de entrega se explica en la Sección 7. Por otro lado, para plantear **dudas específicas de cada paso** de la tarea, se deja un link a un **foro de dudas** al final de cada parte.
A continuación se presenta una **guía** que deberá **seguir paso a paso** para resolver la tarea. Tenga en cuenta que la especificación de cada función se encuentra en el **.h** respectivo, y para cada función se especifica cuál debe ser el orden del tiempo de ejecución en el **peor caso**.

1. Partiendo de la Tarea 2

1. Para comenzar con la Tarea 3 se deben **descargar** los materiales de la sección correspondiente del EVA. Observe que en los archivos *fecha.cpp*, *evento.cpp*, *agendaLS.cpp*, *persona.cpp*, *personasLDE.cpp* y *personasABB.cpp* hay una sección demarcada con el título **PEGAR CÓDIGO TAREA 2**. **Reemplace** estas funciones con el código de la Tarea 2, **sin pisar** el espacio para las nuevas funciones. **Compile** el código de la tarea ejecutando *make*, y verifique que los módulos previos funcionan correctamente **ejecutando** el test tarea2-combinado. Foro de dudas.

2. **Implemente** las funciones insertarInicioDeTPersonasLDE, insertarFinalDeTPersonasLDE, obtenerInicioDeTPersonasLDE y obtenerFinalDeTPersonasLDE dentro del módulo *personasLDE.cpp*. Verifique el funcionamiento de estas funciones **ejecutando** el test personasLDE-insertar-obtener. Foro de dudas.

2. TAD PilaPersona

En esta sección se implementará en *pilaPersona.cpp* el TAD PilaPersona a partir de la especificación dada en *pilaPersona.h*. Para esta implementación se utilizará la lista doblemente encadenada TPersonasLDE ya implementada. Es decir, la estructura de tipo TPilaPersona consistirá de una lista de tipo TPersonasLDE.

1. **Implemente** la estructura rep_pilaPersona que permita almacenar una lista de tipo TPersonasLDE. Foro de dudas.

2. **Implemente** las funciones crearTPilaPersona, liberarTPilaPersona, cantidadEnTPilaPersona y apilarEnTPilaPersona. Recuerde que no debe acceder a la estructura de la lista, solo puede utilizar las funciones del módulo TPersonasLDE. Verifique el funcionamiento de las funciones ejecutando el test pilaPersona1-crear-apilar-cantidad-liberar. Foro de dudas.

3. **Implemente** las funciones cimaDeTPilaPersona y desapilarDeTPilaPersona. **Ejecute** el test pilaPersona2-combinado para verificar el funcionamiento de las funciones. Foro de dudas.

3. TAD ColaPersonasABB

En esta sección se implementará en *colaPersonasABB.cpp* el TAD ColaPersonasABB a partir de la especificación dada en *colaPersonasABB.h*. Este TAD consiste en una cola cuyos elementos son de tipo TPersonasABB, es decir, es una cola de árboles. Para esta implementación se recomienda utilizar una lista simplemente enlazada y cuyos nodos contengan elementos de tipo TPersonasABB. En las operaciones destructoras de este TAD, **los elementos de tipo TPersonasABB no deben liberarse**.

1. **Implemente** la estructura rep_colaPersonasABB que permita almacenar una lista simplemente enlazada. Para poder cumplir con los órdenes de tiempo de ejecución de las operaciones recomendamos que la representación incluya un **cabezal** con un puntero al nodo *inicial* y otro al nodo *final* (además, es posible agregar lo que se considere necesario para cumplir con las restricciones). En este sentido, se debe definir además una **representación auxiliar** para los nodos de la lista, que tengan un elemento *TPersonasABB* y un puntero a un nodo *siguiente*. Foro de dudas.

2. **Implemente** las funciones crearTColaPersonasABB, liberarTColaPersonasABB, cantidadEnTColaPersonasABB y encolarEnTColaPersonasABB. Recuerde que la función liberarTColaPersonasABB **debe eliminar la memoria asociada a la cola, pero no debe liberar la memoria de los elementos de tipo TPersonasABB referenciados**. Verifique el funcionamiento de las funciones ejecutando el test colaPersonasABB1-crear-encolar-cantidad-liberar. Foro de dudas.

3. **Implemente** las funciones frenteDeTColaPersonasABB y desencolarDeTColaPersonasABB. Recuerde que la función desencolarDeTColaPersonasABB **debe eliminar la memoria asociada a la cola, pero no debe liberar la memoria de los elementos de tipo TPersonasABB referenciados**. **Ejecute** el test colaPersonasABB2-combinado para verificar el funcionamiento de las funciones. Foro de dudas.

4. Nuevas funciones en módulo personasABB

En esta sección se implementarán 3 nuevas funciones en el módulo *personasABB.cpp*. Recordar que la estructura de tipo TPersonasABB almacena elementos del tipo TPersona y está implementada como un **árbol binario de búsqueda (ABB)**, **ordenado por el identificador de la persona**.

1. **Implemente** la función amplitudTPersonasABB, que retorna la amplitud del árbol binario. La amplitud se define como la cantidad de nodos en el nivel con más nodos del árbol. La función es $\Theta(n)$ peor caso, siendo n la cantidad de personas en el árbol binario. Recomendamos que la función se implemente realizando una recorrida por niveles utilizando el TAD *colaPersonasABB* como apoyo. **Ejecute** el test personasABB1-amplitud para verificar el funcionamiento de la función. Foro de dudas.

2. **Implemente** la función serializarTPersonasABB. Esta función transforma el árbol en una pila de personas, donde las personas están ordenadas como en una recorrida por niveles del árbol. Es decir, en la cima de la pila debe estar la persona que corresponde al nodo del nivel 1 (la raíz), luego los del nivel 2, etc. El árbol no se debe modificar y los elementos de la pila no comparten memoria con el árbol original. La función es $O(n * m)$ peor caso, siendo n la cantidad de personas en el árbol binario. Recomendamos que la función se implemente realizando una recorrida por niveles utilizando el TAD *colaPersonasABB* como apoyo. **Ejecute** el test personasABB2-serializar para verificar el funcionamiento de la función. Foro de dudas.

3. **Implemente** la función deserializarTPersonasABB. Esta función transforma una pila de personas de tipo TPersona en un árbol binario de tipo TPersonasABB. Las personas en la pila están ordenadas como en una recorrida por niveles del árbol. Es decir, en la cima de la pila está la persona que corresponde al nodo del nivel 1, luego los del nivel 2, etc. **Se asume que el árbol que generó la pila es completo**. Las personas en el árbol resultado deben estar ordenadas de tal forma que si se serializa se obtiene la misma pila. Los elementos del árbol no comparten memoria con la pila original y al final de la función la pila queda vacía y se libera. La función es $O(n * m)$ peor caso, siendo n la cantidad de personas en el árbol binario. Recomendamos el uso del TAD *colaPersonasABB* como apoyo. **Ejecute** el test personasABB3-deserializar para verificar el funcionamiento de la función. Foro de dudas.

4. **Ejecute** los tests personasABB4-combinado y personasABB5-amplitudTiempo. Foro de dudas.

5. TAD ConjuntoIds

En esta sección se implementará en *conjuntoIds.cpp* el TAD ConjuntoIds a partir de la especificación dada en *conjuntoIds.h*. Este TAD consiste en un conjunto **acotado** de identificadores de personas que cumplen $0 < id <= maxCant$, donde maxCant es la cantidad máxima de elementos que puede tener el conjunto. Estos identificadores son de tipo **nat** (tipo natural definido en *utils.h*).

1. **Implemente** la estructura rep_conjuntoIds, que almacena un conjunto acotado de naturales y que permita satisfacer los órdenes de tiempo de ejecución solicitados en *conjuntoIds.h*. Foro de dudas.
2. **Implemente** las funciones crearTConjuntoIds, insertarTConjuntoIds, imprimirTConjuntoIds y liberarTConjuntoIds. Verifique el funcionamiento de las funciones ejecutando el test conjuntoIds1-crear-insertar-imprimir-liberar. Foro de dudas.
3. **Implemente** las funciones esVacioTConjuntoIds y borrarTConjuntoIds. **Ejecute** el test conjuntoIds2-esVacio-borrar para verificar el funcionamiento de la función. Foro de dudas.
4. **Implemente** las funciones perteneceTConjuntoIds, cardinalTConjuntoIds y cantMaxTConjuntoIds. Verifique el funcionamiento de las funciones ejecutando el test conjuntoIds3-pertenece-cardinal-cantMax. Foro de dudas.
5. **Implemente** la función unionTConjuntoIds. **Ejecute** el test conjuntoIds4-union para verificar el funcionamiento de la función. Foro de dudas.
6. **Implemente** la función interseccionTConjuntoIds. **Ejecute** el test conjuntoIds5-interseccion para verificar el funcionamiento de la función. Foro de dudas.
7. **Implemente** la función diferenciaTConjuntoIds. **Ejecute** el test conjuntoIds6-diferencia para verificar el funcionamiento de la función. Foro de dudas.

6. Módulo de aplicaciones

En esta sección se implementarán las funciones del módulo *aplicaciones.cpp*. Estas funciones hacen uso de los TADs implementados anteriormente como parte de su implementación. Recordar que no se debe acceder a la representación de los TADs, solamente se deben usar las funciones definidas de cada TAD.

1. **Implemente** la función menoresQueElResto que recibe una lista de tipo TPersonasLDE y devuelve una pila de tipo TPilaPersona con las personas de la lista cuyas edades son menores estrictas que las de cada una de las siguientes. Es decir, en el resto de la lista no hay ninguna persona cuya edad es menor o igual. En la TPilaPersona resultado los elementos deben estar en orden reverso al que estaban en la lista. Que la TPilaPersona resultado esté en orden reverso implica que la edad de la persona de la cima es mayor que todas las demás.

   Por ejemplo:

   - Si se recibe una lista de personas cuyas edades son: 1, 4, 8, 2, 6, 7, 14, 8, (la izquierda es el inicio de la lista), las edades menores que el resto son: 1, 2, 6, 7 y 8 (tener en cuenta que el último elemento de la lista siempre cumple la condición de menor que el resto). Por lo tanto, la pila resultado es: 8, 7, 6, 2, 1 (donde 8 está en la cima).
   - Si se recibe una lista de personas cuyas edades son: 10, 8, 4, 3, 1, la pila resultado es: 1.
   - Si se recibe una lista de personas cuyas edades son: 1, 3, 8, 10, 20, la pila resultado es: 20, 10, 8, 3, 1.

   La TPilaPersona resultado no comparte memoria con los elementos de la lista y al final de la ejecución de la función la lista debe quedar vacía. No se deben usar estructuras auxiliares ni definir funciones auxiliares. El tiempo de ejecución debe ser $O(n * m)$ peor caso, siendo n la cantidad de elementos de 'lista' y 'm' la cantidad de eventos de la agenda con mas eventos entre todas las personas de 'lista'. **Verifique** el funcionamiento de las funciones **ejecutando** los tests aplicaciones1-menoresQueElResto y aplicaciones2-menoresQueElRestoTiempo. Foro de dudas.

2. **Implemente** la función sumaPares que recibe un natural 'k' y un TConjuntoIds 'c'. Esta función determina si hay un par de ids pertenecientes a 'c' tales que su suma es igual a 'k'. La función es $\Theta(n)$ peor caso, siendo n la cantidad máxima de elementos de 'c'. **Ejecute** el test aplicaciones3-sumaPares para verificar el funcionamiento de las funciones. Foro de dudas.

7. Test final y entrega de la Tarea

Para finalizar con la prueba del programa utilice la regla *testing* del Makefile y verifique que no hay errores en los tests públicos. Esta regla se debe utilizar **únicamente luego de realizados todos los pasos anteriores (instructivo especial para PCUNIX en paso 3)**.

1. **Ejecute**:

```
$ make testing
```

Si la salida no tiene errores, al final se imprime lo siguiente:

```
-- RESULTADO DE CADA CASO --
1111111111111111111
```

Donde un 1 simboliza que no hay error y un 0 simboliza un error en un caso de prueba, en este orden:

```
tarea2-combinado
personasLDE-insertar-obtener
pilaPersona1-crear-apilar-cantidad-liberar
pilaPersona2-combinado
colaPersonasABB1-crear-encolar-cantidad-liberar
colaPersonasABB2-combinado
personasABB1-amplitud
personasABB2-serializar
personasABB3-deserializar
personasABB4-combinado
personasABB5-amplitud-tiempo
conjuntoIds1-crear-insertar-imprimir-liberar
conjuntoIds2-esVacio-borrar
conjuntoIds3-pertenece-cardinal-cantMax
conjuntoIds4-union
conjuntoIds5-interseccion
conjuntoIds6-diferencia
aplicaciones1-menoresQueElResto
aplicaciones2-menoresQueElResto-tiempo
aplicaciones3-sumaPares
```

Foro de dudas.

2. **Prueba de nuevos tests**. Si se siguieron todos los pasos anteriores el programa creado debería ser capaz de ejecutar todos los casos de uso presentados en los tests públicos. Para asegurar que el programa es capaz de ejecutar correctamente ante nuevos casos de uso es importante realizar tests propios, además de los públicos. Para esto **cree un nuevo archivo en la carpeta test**, con el nombre *test_propio.in*, y **escriba una serie de comandos** que permitan probar casos de uso que no fueron contemplados en los casos públicos. **Ejecute el test** mediante el comando:

```
$ ./principal < test/test_propio.in
```

y verifique que la salida en la terminal es consistente con los comandos ingresados. La creación y utilización de casos de prueba propios, es una forma de robustecer el programa para la prueba de los casos de test privados. Foro de dudas.

3. **Prueba en pcunix**. Es importante probar su resolución de la tarea con los materiales más recientes y en una pcunix, que es el ambiente en el que se realizarán las correcciones. Para esto siga el procedimiento explicado en Sugerencias al entregar.

   IMPORTANTE: Debido a un problema en los *pcunix*, al correrlo en esas máquinas se debe iniciar valgrind **ANTES** de correr *make testing* como se indica a continuación:

   Ejecutar los comandos:

   ```
   $ make
   $ valgrind ./principal
   ```

   Aquí se debe **ESPERAR** hasta que aparezca:

   ```
   $ valgrind ./principal
   ==102508== Memcheck, a memory error detector
   ==102508== Copyright (C) 2002-2022, and GNU GPL'd, by Julian $ Seward et al.
   ==102508== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
   ==102508== Command: ./principal
   ==102508==
   $ 1>
   ```

   Luego se debe ingresar el comando **Fin** y recién luego ejecutar:

   ```
   $ make testing
   ```

   Foro de dudas.

4. **Armado del entregable**. El archivo entregable final debe generarse mediante el comando:

   ```
   $ make entrega
   ```

   Con esto se empaquetan los módulos implementados y se los comprime generando el archivo `EntregaTarea3.tar.gz`.

   El archivo a entregar **DEBE** ser generado mediante este procedimiento. Si se lo genera mediante alguna otra herramienta (por ejemplo, usando un entorno gráfico) **la tarea no será corregida**, independientemente de la calidad del contenido. Tampoco será corregida si el nombre del archivo se modifica en el proceso de entrega. Foro de dudas.

5. **Subir la entrega al receptor**. Se debe entregar el archivo **EntregaTarea3.tar.gz**, que contiene los módulos a implementar **fecha.cpp**, **evento.cpp**,**agendaLS.cpp**,**personasLDE.cpp** y **personasABB.cpp**. Una vez generado el entregable según el paso anterior, es necesario subirlo al receptor ubicado en la sección Laboratorio del EVA del curso. **Recordar que no se debe modificar el nombre del archivo generado mediante** `make entrega`. Para verificar que el archivo entregado es el correcto se debe acceder al receptor de entregas y hacer click sobre lo que se entregó para que automáticamente se descargue la entrega.
   IMPORTANTE: Se puede entregar **todas las veces que quieran** hasta la fecha final de entrega. La última entrega **reemplaza a la anterior** y es la que será tenida en cuenta. Foro de dudas.