# Tarea 2
# Manejo dinámico de estructuras lineales
## Curso 2020

## Índice

# 1. Introducción y objetivos

En la presente tarea se trabajará sobre el manejo dinámico de memoria con estructuras lineales, en particular con nodos doblemente enlazados.
A partir de esta tarea **se utilizará el analizador de uso de memoria** `valgrind`. Esta herramienta debe estar instalada en las máquinas en las que se prueban los programas. Está disponible en las máqinas `Linux` de la facultad. **El correcto uso de la memoria será parte de la evaluación**.
En esta tarea se puede trabajar únicamente en forma individual
El resto del presente documento se organiza de la siguiente forma. En la Sección 2 se presenta una descripción de los materiales disponibles para realizar la presente tarea, y en la Sección 3 se detalla el trabajo a realizar. Luego, en la Sección 4 se explica mediante ejemplos el comportamiento esperado de algunas de las funciones a implementar. La Sección 5 describe el formato y mecanismo de entrega, así como los plazos para realizar la misma. En la sección 6 se muestra un ejemplo gráfico de una cadena. En la sección 7 se presenta un breve instructivo acerca de `Valgrind`.

# 2. Materiales

Los materiales para realizar esta tarea se extraen de *MaterialesTarea2.tar.gz*. Ver el procedimiento en la Sección **Materiales** de Funcionamiento y Reglamento del Laboratorio.

En esta tarea los archivos en el directorio `include` son **utils.h**, **info.h**, **cadena.h** y **usoTads.h**.

En el directorio `src` se incluyen ya implementados **utils.cpp** e **info.cpp**. Además se incluye el archivo **ejemploCadena.png** que contiene la imagen de una posible implementación de los tipos y algunas operaciones que se piden del módulo *cadena*. El contenido de este archivo **puede** copiarse en el archivo **cadena.cpp**.

# 3. ¿Qué se pide?

Ver la Sección **Desarrollo** de Funcionamiento y Reglamento del Laboratorio.

En esta tarea solo se deben implementar los archivos **cadena.cpp** y **usoTads.cpp**.

Para implementar `cadena.cpp` se puede usar el código que está en `ejemploCadena.png`.

Se sugiere implementar y probar los tipos y las funciones siguiendo el orden de los ejemplos que se encuentran en el directorio **test**. En la primera línea de cada caso se indica que entidades se deben haber implementado. Después de implementar se compila y se prueba el caso.

```
$ make
$ ./principal < test/N.in > test/N.sal
$ diff test/N.out test/N.sal
```

Si los archivos son iguales no se imprime nada.
Al terminar, para confirmar, se compila y prueban todos los casos:

```
$ make testing
```

Los casos se prueban usando `valgrind` y la utilidad `timeout` mediante la cual se corta la ejecución de un proceso si excede el tiempo que se le asigna.

(Ver la Sección *Makefile* de Material Complementario)
Ver también la Sección *Método de trabajo sugerido* de Material Complementario.

# 4. Descripción de los módulos y algunas funciones

En esta sección se describen los módulos que componen la tarea. Además, de los módulos *cadena* y *usoTads* se describen algunas de las funciones que se deben implementar. Tanto de estas funciones como de las otras que también hay que implementar se debe estudiar la especificación en los archivos `.h` correspondientes.
Se recuerda que además del módulo *info* se entrega una implementación completa. Y del módulo *cadena* ejemplos de alguna de las funciones y de la representación de los tipos.

## 4.1. Módulo *utils*

En este módulo se declara el tipo de los enteros no negativos, arreglos de tipos simples, y operaciones de lectura desde la entrada estándar.

## 4.2. Módulo *info*

En este módulo se declara el tipo `TInfo`. Este tipo representa información compuesta por un natural (tipo `nat`) y un punto flotante (tipo `double`).
Se debe notar que en esta tarea para acceder a los componentes de un elemento de tipo `TInfo` se deben usar las operaciones declaradas en `info.h`.

## 4.3. Módulo *cadena*

El módulo declara dos conceptos, denominados `TCadena` y `TLocalizador`. Con `TCadena` se implementan cadenas doblemente enlazadas de elementos de `TInfo` con una cabecera con punteros al inicio y al final. El tipo `TLocalizador` es un puntero a un nodo de una cadena, o no es válido, en cuyo caso su valor es `NULL`. Con el uso de los localizadores se pueden realizar algunas operaciones de manera eficiente (sin tener que hacer un recorrido de la cadena).
A continuación se muestran gráficamente ejemplos de ejecución de algunas de las operaciones que se solicitan implementar. Cuando la ubicación de los localizadores es relevante se indica mediante colores.

#### 4.3.1. *insertarAntes(i, loc, cad)*

| i | cad (loc) | cad luego de la ejecución |
|---|---|---|
| (3,2.5) | [(1,1.5),(5,3.2),(2,56.1),(8,13.9)] | [(3,2.5),(1,1.5),(5,3.2),(2,56.1),(8,13.9)] |
| (3,2.5) | [(1,1.5),(5,3.2),(2,56.1),(8,13.9)] | [(1,1.5),(5,3.2),(3,2.5),(2,56.1),(8,13.9)] |

#### 4.3.2. *removerDeCadena(loc, cad)*

| cad (loc) | cad luego de la ejecución |
|---|---|
| [(1,1.5),(5,3.2),(2,56.1),(8,13.9)] | [(5,3.2),(2,56.1),(8,13.9)] |
| [(1,1.5),(5,3.2),(2,56.1),(8,13.9)] | [(1,1.5),(5,3.2),(2,56.1)] |
| [(1,1.5),(5,3.2),(2,56.1),(8,13.9)] | [(1,1.5),(5,3.2),(8,13.9)] |

#### 4.3.3. *intercambiar(loc1, loc2, cad)*

| cad (loc1 y loc2) | cad luego de la ejecución |
|---|---|
| [(1,1.5),(3,8.1),(5,3.2),(9,7.4),(6,-4.7)] | [(1,1.5),(9,7.4),(5,3.2),(3,8.1),(6,-4.7)] |

#### 4.3.4. *siguienteClave(clave, loc, cad)*

| clave | cad (loc) | resultado (representado en cad) |
|---|---|---|
| 8 | [] (loc es ignorado) | localizador no válido |
| 3 | [(1,1.5),(4,0.9),(2,56.1)] | localizador no válido |
| 9 | [(1,1.5),(4,0.9),(2,56.1),(9,7.4)] | [(1,1.5),(4,0.9),(2,56.1),(9,7.4)] |
| 4 | [(1,1.5),(4,0.9),(2,56.1),(9,7.4)] | [(1,1.5),(4,0.9),(2,56.1),(9,7.4)] |

#### 4.3.5. *borrarSegmento(desde, hasta, cad)*

| cad (desde y hasta) | cad luego de la ejecución |
|---|---|
| [(1,1.5),(3,8.1),(5,3.2),(9,7.4),(6,-4.7)] | [(6,-4.7)] |
| [(1,1.5),(3,8.1),(5,3.2),(9,7.4),(6-4.7)] | [(1,1.5),(3,8.1)] |
| [(1,1.5),(3,8.1),(5,3.2),(9,7.4),(6,-4.7)] | [(1,1.5),(6,-4.7)] |

### 4.4. Módulo *usoTads*

Este módulo corresponde a funciones que usan las operaciones declaradas en los módulos *info* y *cadena*.

## 5. Entrega

Se mantienen las consideraciones reglamentarias y de procedimiento de la tarea anterior.
En particular, la tarea otorga dos puntos a quienes la aprueben en la primera instancia de evaluación y uno o cero a los que aprueben en la segunda instancia. La adjudicación de los puntos de las tareas de laboratorio queda condicionada a la respuesta correcta de una pregunta sobre el laboratorio en el segundo parcial.

Se debe entregar el siguiente archivo, que contiene los módulos implementados `cadena.cpp` y `usoTads.cpp`:

- **Entrega2.tar.gz**

Este archivo se obtiene al ejecutar la regla `entrega` del archivo *Makefile*:

```
$ make entrega
tar zcvf Entrega2.tar.gz -C src cadena.cpp usoTads.cpp
cadena.cpp
usoCadena.cpp
```

Con esto se empaquetan los módulos implementados y se los comprime.
**Nota:** En la estructura del archivo de entrega los módulos implementados deben quedar en la raíz, NO en el directorio `src` (y por ese motivo se usa la opción `-C src` en el comando `tar`).

### 5.1. Plazos de entrega

El plazo para la entrega es el **miércoles 22 de abril a las 14 horas**.

### 5.2. Identificación de los archivos de las entregas

Cada uno de los archivos a entregar debe contener, en la primera línea del archivo, un comentario con el número de cédula del estudiante, **sin el guión y sin dígito de verificación**. Ejemplo:

```
/* 1234567  */
```

### 5.3. Individualidad

Ver la Sección **Individualidad** de Funcionamiento y Reglamento del Laboratorio.

## 6. Ejemplo de cadena

En la Figura 1 se muesta un ejemplo de `TCadena`.

La variable *cad* de tipo `TCadena` es un puntero a una estructura de dos miembros, *inicio* y *final* que son punteros a `nodo`. Un elemento de tipo `nodo` es una estructura con tres miembros: *anterior* y *siguiente* de tipo puntero a `nodo`, y *dato* de tipo `TInfo`. El tipo `TInfo` es un puntero a una estructura de dos miembros: *n*, de tipo `nat` (con valor 13 en el ejemplo) y *r*, de tipo `double` (con valor 6.5 en el ejemplo).

Las variables *loc1*, *loc2* y *loc3* son de tipo `TLocalizador`, que es un puntero a `nodo`.

En este ejemplo, accediendo a la representación de `cadena`, se puede afirmar que se cumplen los predicados

- `loc1 == cad->inicio->siguiente`,
- `loc2 == cad->final->siguiente`,
- `loc2 == NULL`.

Usando las operaciones de `cadena` se cumplen, entre otros:

- `! esVaciaCadena(cad)`,
- `loc1 == siguiente(inicioCadena(cad), cad)`,
- `loc2 == siguiente(finalCadena(cad), cad)`,
- `! esLocalizador(loc2)`,
- `localizadorEnCadena(loc1, cad)`,

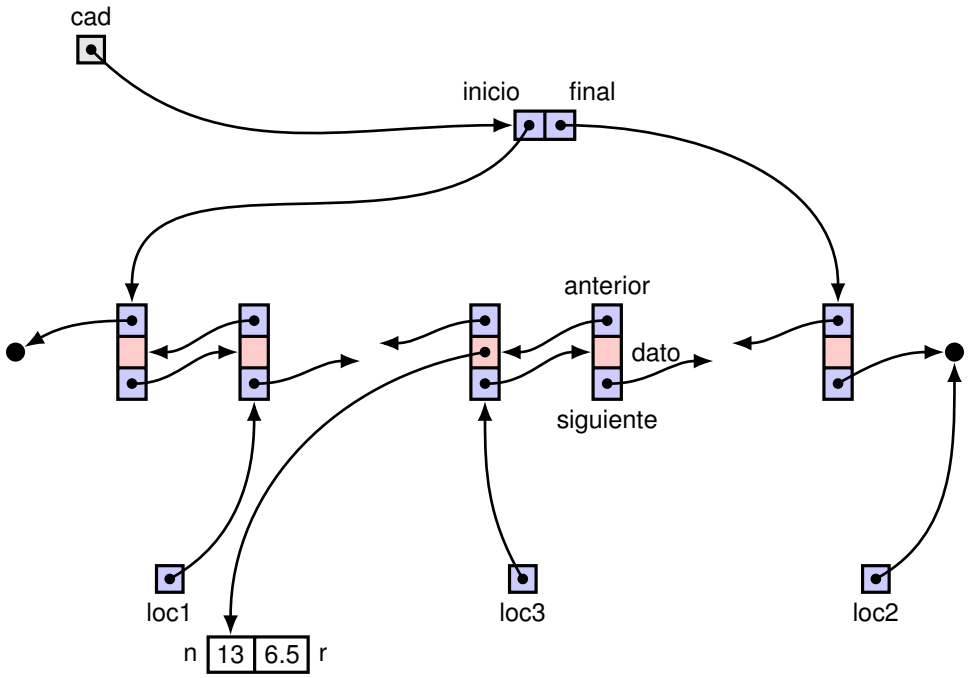


Figura 1: Ejemplo de cadena

- `! localizadorEnCadena(loc2, cad)`,
- `precedeEnCadena(loc1, loc3, cad)`.

Nótese que no se puede expresar `loc3->dato->n == 13` porque desde `cadena` no se puede acceder a la representación de `TIinfo`. En cambio es válido `numeroInfo(loc3->dato) == 13` o, usando las operaciones de `cadena`,
`numeroInfo(infoCadena(loc3, cad)) == 13`.

## 7. Control de manejo de memoria

La memoria que se obtiene dinámicamente (mediante `new`) debe ser liberada (con `delete`). Esto debe hacerse de manera sistemática: por cada instrucción que solicita memoria debe haber alguna o algunas correspondientes que la liberen. No obstante, además de esta buena práctica de programación, se debe controlar que el programa hace un correcto manejo de le memoria. Una vez que se sabe que el programa cumple la especificación funcional se debe correr alguna herramienta de análisis. Un ejemplo de ello es `valgrind`.

El siguiente es el resultado de la ejecución de un programa que deja sin liberar 1 bloque de 16 bytes:

```
$ valgrind ./principal < test/01.in
==26226== Memcheck, a memory error detector
==26226== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==26226== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==26226== Command: ./principal
==26226==
1>#  prueba crear_info y liberar_info.
2>Fin.
==26226==
==26226== HEAP SUMMARY:
==26226==     in use at exit: 16 bytes in 1 blocks
==26226==   total heap usage: 5 allocs, 4 frees, 1,122,321 bytes allocated
==26226==
```

```
==26226== LEAK SUMMARY:
==26226==    definitely lost: 16 bytes in 1 blocks
==26226==    indirectly lost: 0 bytes in 0 blocks
==26226==      possibly lost: 0 bytes in 0 blocks
==26226==    still reachable: 0 bytes in 0 blocks
==26226==         suppressed: 0 bytes in 0 blocks
==26226== Rerun with --leak-check=full to see details of leaked memory
==26226==
==26226== For counts of detected and suppressed errors, rerun with: -v
==26226== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

---

Al comando de ejecución se le puede agregar la opción `--leak-check=full` para obtener más informacion:

---

```
$ valgrind --leak-check=full ./principal < test/01.in
==26400== Memcheck, a memory error detector
==26400== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==26400== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==26400== Command: ./principal
==26400==
1>#  prueba crear_info y liberar_info.
2>Fin.
==26400==
==26400== HEAP SUMMARY:
==26400==     in use at exit: 16 bytes in 1 blocks
==26400==   total heap usage: 5 allocs, 4 frees, 1,122,321 bytes allocated
==26400==
==26400== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==26400==    at 0x4C2E1FC: operator new(unsigned long) (vg_replace_malloc.c:334)
==26400==    by 0x400F97: crear_info(int, char*) (info.cpp:23)
==26400==    by 0x400A06: main (principal.cpp:47)
==26400==
==26400== LEAK SUMMARY:
==26400==    definitely lost: 16 bytes in 1 blocks
==26400==    indirectly lost: 0 bytes in 0 blocks
==26400==      possibly lost: 0 bytes in 0 blocks
==26400==    still reachable: 0 bytes in 0 blocks
==26400==         suppressed: 0 bytes in 0 blocks
==26400==
==26400== For counts of detected and suppressed errors, rerun with: -v
==26400== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

---

Se muestra que en la línea 47 de `principal.cpp` se llama a `crear_info`, que en la línea 23 de `info.cpp` usa el operador `new` para obtener memoria. Ese segmento de memoria obtenido es el que no fue liberado. Luego de agregar el `delete` en la función que debe devolver ese bloque se puede ver que no hay errores ni memoria perdida.

---

```
$ valgrind ./principal < test/01.in
==26905== Memcheck, a memory error detector
==26905== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==26905== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==26905== Command: ./principal
==26905==
1>#  prueba crear_info y liberar_info.
```

---

```
2>Fin.
==26905==
==26905== HEAP SUMMARY:
==26905==     in use at exit: 0 bytes in 0 blocks
==26905==   total heap usage: 5 allocs, 5 frees, 1,122,321 bytes allocated
==26905==
==26905== All heap blocks were freed -- no leaks are possible
==26905==
==26905== For counts of detected and suppressed errors, rerun with: -v
==26905== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```