

## ICI 313 Estructura de Datos

### Tarea 1

Nombre: Katerina Peñaloza Caballería  
Rut: 20.360.261-8  
Correo: [katerina.penaloza@alumnos.uv.cl](mailto:katerina.penaloza@alumnos.uv.cl)  
Fecha: 15-06-2022

Objetivo: Implementar de manera exitosa en lenguaje C, los tipos de datos abstractos (TDA) relacionados con colecciones de datos.

### Introducción

En el presente informe se desarrollarán los siguientes tipos de datos abstractos (TDA) incluyendo una breve descripción, una tabla con las operaciones a implementar en el código, el orden de complejidad de cada operación, un pequeño diagrama demostrativo de la operación y los axiomas correspondientes de cada uno de los TDA:

1. Lista Secuencial
2. Pila Secuencial
3. Cola Secuencial
4. Lista simplemente enlazada
5. Cola con prioridad

Cada TDA se implementará utilizando el lenguaje de programación C, para compilar el código fuente se usará el gcc de Linux, además se añadirá un archivo README.txt para mayor comprensión de las instrucciones a seguir en la compilación. En caso de algún problema con los archivos, también se adjuntará un enlace al final de este documento con el proyecto en GitHub.

Como propuesta de solución, planteo el siguiente formato:

1. Definición e implementación de las funciones con las operaciones del TDA.
2. Un menú en donde el usuario podrá elegir que operaciones desea utilizar, esto para hacer más amigable y didáctico el uso de las operaciones y el entendimiento de la estructura de datos.

Finalmente se presentará una conclusión del trabajo realizado, verificando si se pudo lograr el objetivo de esta tarea y la solución planteada, junto con una presentación de las dificultades durante la tarea, limitaciones y proyecciones del trabajo

## Lista Secuencial

### Descripción

Una lista se define como un tipo de dato abstracto en el cual almacena elementos (pueden ser de distinto tipo y repetidos) de manera secuencial y ordenada en espacios de memoria consecutivos. Es de tamaño variable, es dinámica ya que su tamaño aumenta a medida que se agregan elementos en ella. Para acceder a sus elementos se usa un índice.

En este caso se implementará la lista usando un arreglo unidimensional.



Figura 1

### Operaciones

Sea L una lista, e un elemento, p una posición en la lista:

Nombre	Descripción	Complejidad	Diagrama
<b>size()</b>	Devuelve el tamaño de la Lista	$O(n)$	
<b>anular()</b>	Vaciar la Lista	$O(n)$	
<b>vacía()</b>	¿La lista está vacía?, Devuelve verdadero si la Lista L está vacía	$O(1)$	
<b>fin()</b>	Devuelve el elemento al final de la lista	$O(1)$	
<b>siguiente(p)</b>	Devuelve el elemento siguiente al elemento de la posición indicada	$O(1)$	
<b>previo(p)</b>	Devuelve el elemento anterior al elemento de la posición indicada	$O(1)$	
<b>imprimir()</b>	Mostrar la Lista	$O(n)$	
<b>insertar(p,e)</b>	Insertar un elemento en la posición indicada de la lista	$O(1)$ + tiempo de búsqueda	
<b>localizar(e)</b>	Encontrar la posición de un elemento	$O(1)$	
<b>recuperar(p)</b>	Mostrar el elemento de la posición dada	$O(1)$	
<b>suprimir(p)</b>	Eliminar un elemento dado una posición	$O(1)$ + tiempo de búsqueda	

### Axiomas

Para una lista (L), elemento (e), posición (p):

1.  $L.size()$ ,  $L.vacia()$ ,  $L.anular()$ ,  $L.insertar(e)$ , siempre están definidas.
2.  $(L.insertar(p,e)).vacia() = false$
3.  $(L.insertar(p,e)).size() = L.size() + 1$
4.  $(L.suprimir(L.size()-1)).size() = L.size() - 1$
5.  $(L.suprimir(e)).vacia() \leq L.size()$
6. Si  $(L.vacia() == true)$ , entonces  $(L.recuperar(L.size() - 1) = error)$
7. Si  $(L.vacia() == true)$ , entonces  $(L.suprimir(L.size() - 1) = error)$
8. Si  $(L.vacia() == true)$ , entonces  $(L.insertar(e)).recuperar(L.size() - 1) = e$
9. Si  $(L.vacia() == true)$ , entonces  $(L.insertar(e)).suprimir(L.size() - 1) = L$
10.  $(L.insertar(p,e)).suprimir(p) = L$
11.  $L.recuperar(p) = (L.insertar(p,e)).recuperar(p+1)$
12.  $L.recuperar(p+1) = (L.suprimir(p)).recuperar(p)$

## Pila Secuencial

### Descripción

Una pila o stack, es una estructura de datos lineal que representa a una pila de objetos secuencial y de tamaño variable, en la cual solo se pueden agregar o retirar objetos desde uno de sus extremos (el tope). El tope es el último elemento agregado y el único elemento visible.

Los elementos se eliminan en el orden inverso a que se insertaron, es decir, sigue el orden **LIFO (last-in, first-out)**, el último en ser agregado será el primero en salir.

En este caso se implementará la pila usando un arreglo unidimensional, por lo que necesita un límite de elementos.

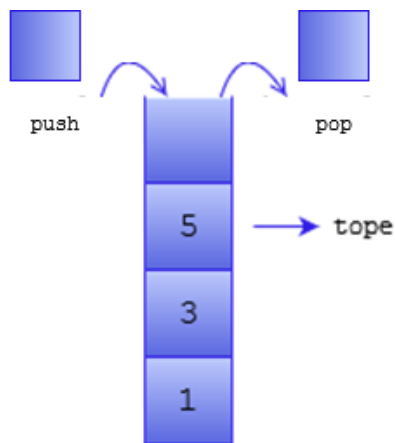
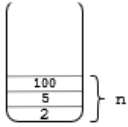

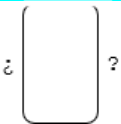
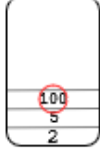
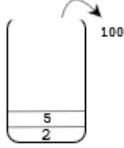
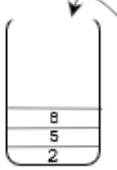


Figura 2

## Operaciones

Sea P una pila, e un elemento:

Nombre	Descripción	Complejidad	Diagrama
<b>size()</b>	Devuelve la cantidad de elementos que contiene la Pila P.	$O(1)$	
<b>anular()</b>	Vaciar la Pila	$O(n)$	
<b>vacía()</b>	¿La Pila está vacía?, Devuelve verdadero si la Pila P está vacía	$O(1)$	
<b>tope()</b>	Devuelve elemento ubicado en la parte superior de la Pila P	$O(1)$	
<b>pop()</b>	Elimina el elemento ubicado en la parte superior de la Pila P	$O(1)$	
<b>push(e)</b>	Inserta un elemento en la parte superior de la Pila P	$O(1)$	

### Axiomas

Para una pila P, un elemento e

1.  $P.size()$ ,  $P.vacia()$ ,  $P.push(e)$  siempre están definidos
2.  $P.pop()$  y  $P.tope()$  estarán definidas si y solo si  $P.vacia() = false$
3.  $P.vacia()$ ,  $P.size()$ ,  $P.tope()$  no cambian la pila
4.  $P.vacia() = true$  si y solo si  $P.size() = 0$
5.  $P.push(e)$  seguido de  $P.pop() = P$
6.  $P.push(e)$  seguido de  $P.tope() = e$
7.  $P.push(e) = P.size + 1$
8.  $P.pop() = P.size - 1$
9. Si  $(P.vacia() == true)$  entonces  $(P.tope() = error)$
10. Si  $(P.vacia() == true)$  entonces  $(P.pop() = error)$
11.  $P.push(e).vacia() = false$
12.  $P.push(e).tope() = e$
13.  $(P.push(e)).pop() = P$

## Cola Secuencial

### Descripción

Una cola es una estructura de datos secuencial de tamaño variable, en donde solo se pueden agregar elementos por un extremo (back), y quitar por el otro extremo (front). Esta estructura sigue el orden **FIFO (first-in, first-out)**, es decir, el primero en entrar es el primero en salir.

En este caso se implementará la cola usando un arreglo unidimensional.

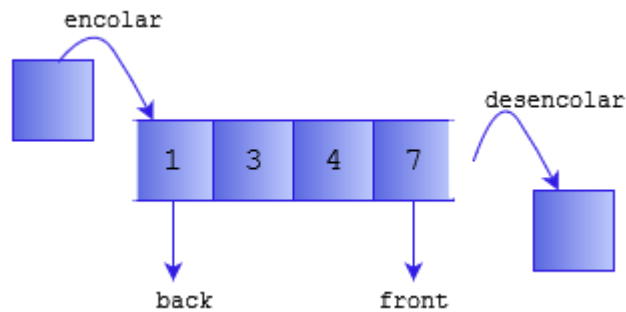
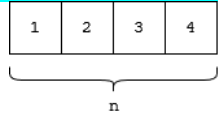


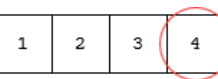
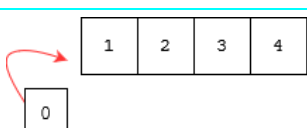
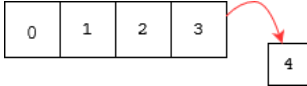


Figura 3

### Operaciones

Sea C una cola, e un elemento:

Nombre	Descripción	Complejidad	Diagrama
<b>size()</b>	Devuelve la cantidad de elementos que contiene la Cola C.	$O(1)$	
<b>anular()</b>	Vaciar la Cola	$O(n)$	
<b>vacía()</b>	¿La Cola está vacía?, Devuelve verdadero si la Cola C está vacía	$O(1)$	
<b>frente()</b>	Devuelve el elemento ubicado al principio (frente, primer lugar) de la Cola	$O(1)$	
<b>encolar(e)</b>	Inserta un elemento al final de la Cola	$O(1)$	
<b>desencolar(e)</b>	Elimina el elemento que esta al principio (frente, primer lugar) de la cola	$O(1)$	

### Axiomas

1.  $C.size()$ ,  $C.vacia()$ ,  $C.encolar(e)$  siempre están definidos
2.  $C.desencolar()$  y  $C.frente()$  están definidos si y solo si  $C.vacia() = false$
3.  $C.vacia()$ ,  $C.size()$ ,  $C.frente()$  no cambian  $C$
4.  $C.vacia() = true$  si y solo si  $C.size() = 0$
5.  $C.encolar(e) = C.size() + 1$
6.  $C.desencolar() = C.size() - 1$
7.  $(C.encolar(e)).vacia() = false$
8. Si  $(C.vacia() == true)$  entonces  $(C.frente() = error)$
9. Si  $(C.vacia() == true)$  entonces  $(C.desencolar() = error)$
10. Si  $(C.vacia() == true)$  entonces  $(C.encolar(e)).frente() = e$
11. Si  $(C.vacia() == true)$  entonces  $(C.encolar(e)).desencolar() = C$



## Lista Simplemente Enlazada

### Descripción

Una lista simplemente enlazada es una estructura dinámica de datos que contiene una colección de datos del mismo tipo de manera ordenada, es decir, cada elemento, menos el primero, tiene un predecesor, y cada elemento, menos el último, tiene un sucesor. Los nodos deben leerse de manera secuencial y la inserción y eliminación de un elemento se puede hacer desde cualquier parte de la lista.

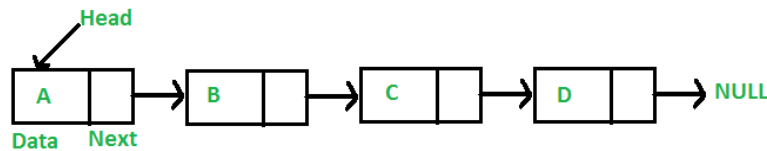


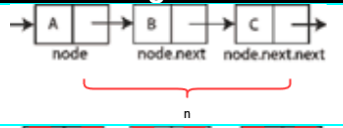

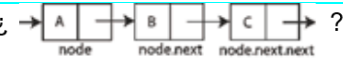
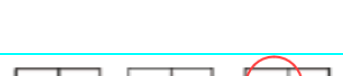
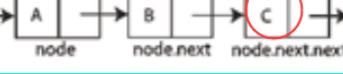
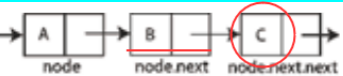
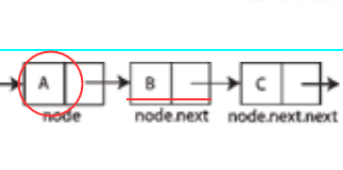
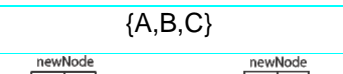

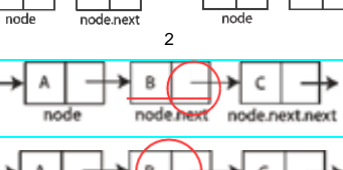
Figura 4<sup>1</sup>

---

<sup>1</sup> Imagen rescatada de <https://www.geeksforgeeks.org/data-structures/linked-list/>

## Operaciones

Sea L una lista s.e (simplemente enlazada), e un elemento, p una posición en la lista s.e, head el cabezal de la lista:

Nombre	Descripción	Complejidad	Diagrama
size(head)	Devuelve la cantidad de elementos de la Lista s.e	$O(n)$	
anular(&head)	Vaciar la Lista s.e	$O(n)$	
vacía()	¿La lista está vacía?, Devuelve verdadero si la Lista s.e L está vacía	$O(1)$	
fin()	Devuelve el elemento al final de la lista s.e	$O(n)$	
siguiente(p)	Devuelve el elemento siguiente al elemento de la posición indicada)	$O(1)$	
previo(p)	Devuelve el elemento anterior al elemento de la posición indicada)	$O(1)$	
imprimir(head)	Mostrar la Lista s.e	$O(n)$	{A,B,C}
insertar(head,p,e)	Insertar un elemento en la posición indicada de la lista s.e	$O(1)$	
localizar(e)	Encontrar la posición de un elemento	$O(n)$ peor caso	
recuperar(p)	Mostrar el elemento de la posición dada	$O(n)$ peor caso	
suprimir(&head,p)	Eliminar un elemento dado una posición	$O(1)$	

<sup>2</sup> De Derrick Coetzee - en:Image:Singly linked list insert after.png, Dominio público, <https://commons.wikimedia.org/w/index.php?curid=2754486>

<sup>3</sup> De Derrick Coetzee - en:Image:Singly linked list delete after.png, Dominio público, <https://commons.wikimedia.org/w/index.php?curid=2754528>

### Axiomas

Para una lista simplemente enlazada (L), elemento (e), posición (p):

1.  $L.size()$ ,  $L.vacia()$ ,  $L.anular()$ ,  $L.insertar(e)$  siempre están definidas.
2.  $(L.insertar(p,e)).vacia = false$
3.  $(L.insertar(p,e)).size() = L.size() + 1$
4.  $(L.suprimir(L.size()-1)).size() = L.size() - 1$
5. Si  $(L.vacia() == true)$ , entonces  $(L.recuperar(L.size() - 1) = error)$
6. Si  $(L.vacia() == true)$ , entonces  $(L.suprimir(L.size() - 1) = error)$
7. Si  $(L.vacia() == true)$ , entonces  $(L.insertar(e)).recuperar(L.size() - 1) = e$
8. Si  $(L.vacia() == true)$ , entonces  $(L.insertar(e)).suprimir(L.size() - 1) = L$
9.  $(L.insertar(p,e)).suprimir(p) = L$
10.  $L.recuperar(p) = (L.insertar(p,e)).recuperar(p+1)$
11.  $L.recuperar(p+1) = (L.suprimir(p)).recuperar(p)$

## Cola con Prioridad

### Descripción

Una cola con prioridad es una estructura de datos similar a una cola, pero con una prioridad asignada a cada elemento. En una cola con prioridad un elemento con mayor prioridad será descolado antes que un elemento de menor prioridad, si dos elementos tienen la misma prioridad, se descolarán siguiendo el orden de cola.

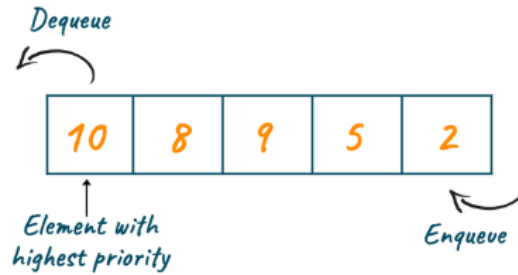


Figura 5<sup>4</sup>

En este caso se implementará la cola con prioridad usando una lista enlazada.

La representación de la cola con prioridad implementadas con listas enlazadas:

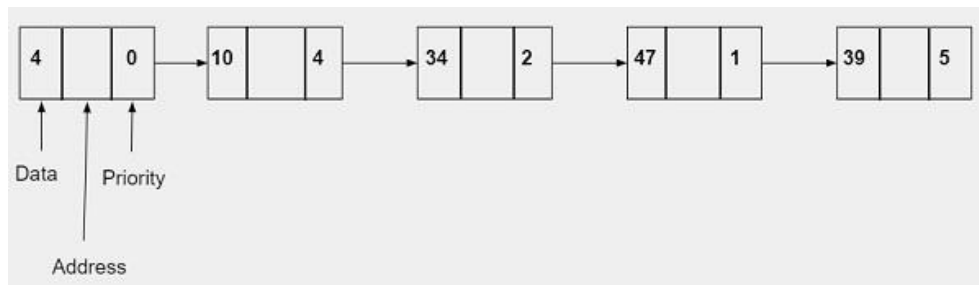


Figura 6<sup>5</sup>

<sup>4</sup> <https://favtutor.com/blogs/priority-queue-cpp>

<sup>5</sup> <https://www.tutorialspoint.com/priority-queue-using-linked-list-in-c>

## Operaciones

Sea CP una cola con prioridad, e un elemento, p prioridad, head un cabezal:

<b>Nombre</b>	<b>Descripción</b>	<b>Complejidad</b>
<b>crear(e, p)</b>	Crea una cola con prioridad	$O(1)$
<b>longitud(head)</b>	Devuelve la cantidad de elementos de la Cola con prioridad	$O(1)$
<b>anular(&amp;head)</b>	Vaciar la Cola con prioridad	$O(n)$
<b>vacía(&amp;head)</b>	¿La cola con prioridad está vacía?, Devuelve verdadero si la Cola con prioridad CP está vacía	$O(1)$
<b>elem_min(head)</b>	Devuelve el elemento que tiene la mínima prioridad de la cola con prioridad	$O(n)$
<b>elem_max(&amp;head)</b>	Devuelve el elemento que tiene la máxima prioridad de la cola con prioridad	$O(1)$
<b>insertar(&amp;head, e, p)</b>	Insertar un elemento con prioridad p en la cola con prioridad	$O(n)$
<b>prioridad_min(head)</b>	Devuelve la mínima prioridad presente en la cola con prioridad	$O(n)$
<b>prioridad_max(&amp;head)</b>	Devuelve la máxima prioridad presente en la cola con prioridad	$O(1)$
<b>eliminar_minimo()</b>	Elimina el elemento con la mínima prioridad	$O(1)$

En este caso la complejidad de las operaciones relacionadas a un mínimo son de complejidad n en peor caso ya que recorren toda la lista en su búsqueda, mientras que las relacionadas a una máxima prioridad son constantes.

### Axiomas

1. `elem_min` devuelve la mínima prioridad presente en la cola con prioridad, si la cola está vacía lanza error
2. El tamaño de una cola nueva es 0
3.  $\text{size}(\text{insertar}(e, p)) = \text{size}() + 1$
4.  $\text{vacía}(\text{cola nueva}) = \text{true}$
5.  $\text{vacía}(\text{insertar}(e)) = \text{false}$
6.  $\text{elem\_min}(\text{cola nueva}) = \text{error}$
7.  $\text{eliminar\_minimo}() = \text{size}() - 1$

## Conclusión

En conclusión, en esta tarea describimos los diferentes tipos de datos abstractos que nos pedían desarrollar, describimos las operaciones con un dibujo incluido y se hizo un análisis de complejidad a cada una de las operaciones, para un mejor entendimiento de las estructuras al momento de implementarlas en código.

El objetivo de esta tarea se cumplió ampliamente, ya que se debía implementar exitosamente las estructuras de datos pedidas, lo cual se cumplió en su mayoría. Sin embargo hubo algunas dificultades a la hora de implementar el código, por ejemplo que los TDA implementadas con arreglos debían tener un tamaño máximo a pesar de que son estructuras de tamaño dinámico, otro “problema” es que para una mayor abstracción y simples del problema a resolver, se decidió que los elementos insertados en el TDA fueran de tipo entero. El problema más grande fue que las listas enlazadas no pude crear una función que insertara en la posición deseada, solo pude hacer que se insertaran al final de la lista.

Las limitaciones en esta tarea como se mencionó antes fue el hecho de implementar algunos TDA con arreglos, ya que esto supuso un trabajo extra en cuanto a la lógica del TDA, la otra limitación fue el pobre conocimiento sobre listas enlazadas, arreglos, punteros, etc. en general un conocimiento bajo sobre el lenguaje de programación C, esto llevó a ocupar un mayor tiempo implementando el código al requerir tiempo para repasar estos contenidos.

En un futuro se espera tener conocimientos más sólidos en programación y estructuras de datos para poder implementar nuevas operaciones o implementar los TDA con otros tipos de estructuras de datos más avanzadas, de manera que sean aún más eficientes.

## Bibliografía

Link a GitHub → <https://github.com/KaterinaPenaloza/Tarea01-ED.git>

PPTS vistos en clases

Mayoría de Diagramas y figuras de creación propia (los que no, tienen insertada una nota al pie de página con la fuente)

<https://docs.oracle.com/javase/7/docs/api/java/util/List.html>

<https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html>

[https://www.cs.fsu.edu/~lacher/lectures/Output/adts/index.html?\\$\\$\\$slide01.html\\$\\$\\$](https://www.cs.fsu.edu/~lacher/lectures/Output/adts/index.html?$$$slide01.html$$$)

[http://agrega.juntadeandalucia.es/repositorio/02122016/a5/es-an\\_2016120212\\_9131705/33\\_listas.html](http://agrega.juntadeandalucia.es/repositorio/02122016/a5/es-an_2016120212_9131705/33_listas.html)

<https://www.geeksforgeeks.org/data-structures/linked-list/>

<https://www.geeksforgeeks.org/priority-queue-using-linked-list/>

[https://www.tutorialspoint.com/data\\_structures\\_algorithms/dsa\\_queue.htm](https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm)

[https://es.wikipedia.org/wiki/Lista\\_enlazada#Implementaci%C3%B3n\\_de\\_una\\_lista\\_enlazada\\_en\\_C](https://es.wikipedia.org/wiki/Lista_enlazada#Implementaci%C3%B3n_de_una_lista_enlazada_en_C)

[https://en.wikipedia.org/wiki/Priority\\_queue](https://en.wikipedia.org/wiki/Priority_queue)

<https://prepinsta.com/c-program/implementation-of-priority-queue-using-linked-list/>