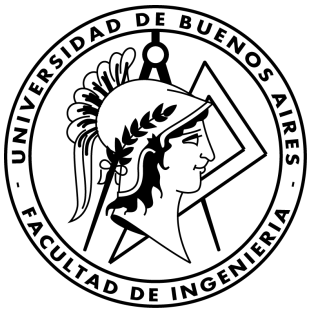

Trabajo Práctico 02
Sala de escape Pokemon

[7541/9515] Algoritmos y Programación II

Primer cuatrimestre de 2022

Autor:	**Email**:	**Padrón**:	**Fecha**:
Luca Lazcano	llazcano@fi.uba.ar	107044	31/04/2022

Índice

1. Introducción

En las siguientes páginas se ensayará una explicación de los TDA (tipo de dato abstracto) estudiados en la cátedra; y también una explicación de su implementación por parte del alumno. Los TDA a implementar fueron lista, pila y cola, con nodos simplemente enlazados. Además, se expondrán con mayor profundidad cómo se implementarían el TDA pila y lista con nodos simplemente enlazados, y el TDA cola circular con vectores estáticos. Todos estos conceptos serán debidamente abordados como introducción teórica a la implementación específica del alumno.

En este trabajo práctico, se pone al alumno como personaje de un juego. Se encuentra en una sala de la cual busca escapar y donde se encuentran ciertos objetos. Se les pueden realizar ciertas acciones (como leer una nota o abrir una pokebola), y pueden interactuar entre ellos de ciertas formas definidas (como usar una llave para abrir un cajón). Los objetos de la sala y las interacciones posibles se proveen en dos archivos de texto.

Se proveen en la carpeta `./src` archivos `.h` con las firmas y descripciones de varias funciones correspondientes a cada TDA. Estas son, además de las operaciones típicas que definen a una lista, pila y cola, otras funciones accesorias. Están definidos también algunos TDA útiles para la implementación del trabajo, como TDA nodo o TDA iterador.

Se provee también un archivo `pruebas.c` donde el alumno debe implementar pruebas para verificar el correcto funcionamiento de su implementación. Finalmente se provee un archivo `ejemplo.c` cuyo propósito es ayudar al alumno a entender mejor el funcionamiento esperado de los TDA.

2. Algo de teoría previa

2.1. Tipos de dato abstracto

Un tipo de dato abstracto, o TDA, es un modelo para definir tipos de datos. Se define mediante un conjunto de datos posibles, más un conjunto de operaciones permitidas con dichos datos.

Definir un TDA es útil para abstraer la implementación específica de un problema y proveerle a un usuario un conjunto mínimo de operaciones (funciones) a utilizar.

2.2. TDA Pila

Una pila es una colección de elementos ordenados, en la que pueden insertarse y sacarse elementos por un extremo, denomido tope. Es del tipo llamado LIFO, last in first out; es decir, el último elemento insertado es el primer elemento en ser extraído. Como una torre de panqueques. Las operaciones mínimas que debe tener una pila son: crear la pila, poner y sacar un elemento, verificar el tope, verificar que esté vacía y por último, destruir la pila.

2.3. TDA Cola

Una cola también es una colección de elementos, pero tiene dos extremos. Por el frente se agregan elementos, por el final se quitan. Es del tipo llamado FIFO, first in first out; es decir, el primer elemento insertado es el primer elemento extraído. Como una fila de supermercado. Las operaciones mínimas que debe tener una cola son: crear la cola, encolar y desencolar un elemento, ver el primer elemento, verificar que esté vacía y por último, destruir la cola.

2.4. TDA Lista

Una lista es también una colección ordenada de elementos, más flexible y versátil que pila o cola. En una lista es posible agregar y quitar elementos de cualquier posición. Las operaciones básicas incluyen: crearla y destruirla, insertar, quitar y ver un elemento en cualquier posición, y verificar que esté vacía. Se puede pensar a los TDA cola y pila como listas a las cuales se les restringen las operaciones posibles. Esta característica será útil para la implementación.

2.5. Posibles implementaciones

Existen varias formas de implementar los TDA anteriores, e incluso pueden modificarse para generar variantes. Generalmente hablando, se pueden implementar con vectores estáticos o dinámicos, o con nodos enlazados. Cada forma tiene sus ventajas y desventajas, pero la implementación con nodos enlazados es la más versátil, poderosa y más amigable con el manejo de la memoria, ya que no requiere de un conjunto contiguo de memoria.

2.6. Nodos enlazados

Un nodo simple incluye la información a almacenar, más un puntero al siguiente nodo, y un nodo doble tiene además un puntero al nodo anterior. Con los primeros podemos crear un lista de nodos simplemente enlazados, que se recorre únicamente desde el principio hasta el final. Con los segundos podemos crear una lista de nodos doblemente enlazados que permite recorrerla en ambas direcciones. Por último, una variante de las listas consiste en enlazar el último nodo con el primero (en una o ambas direcciones, dependiendo del tipo de nodos usados), y crear una lista circular.

3. Algunas implementaciones

Detallamos a continuación cómo se implementan los siguientes TDA:

- TDA pila utilizando nodos simplemente enlazados.
- TDA cola utilizando vectores estáticos (cola circular).
- TDA lista utilizando nodos simplemente enlazados.

3.1. Pila con nodos simplemente enlazados

Para implementar una pila con nodos simplemente enlazados, necesitamos en primer lugar definir el TDA nodo. A continuación podemos pensar la implementación de

las distintas operaciones.

3.1.1. Crear

Se crea un TDA `pila` asignándole memoria. Cuenta con un puntero al tope de la pila que se apunta a NULL, y eventualmente un contador para registrar la cantidad de elementos de la pila, que se inicia en 0.

Complejidad: O(1), ya que únicamente se trata de crear los elementos.

3.1.2. Poner (push)

Poner un elemento agrega un nodo al principio de la pila y actualiza el `tope` para apuntar al nodo nuevo. Para esto apuntamos el `siguiente` del nuevo nodo al nodo inicial o `tope`, luego actualizamos el `tope` para que apunte al nuevo nodo.

En caso de que la pila esté vacía al poner un elemento, el `siguiente` de nuevo nodo apuntará a NULL, y el `tope` será actualizado de NULL al nuevo nodo.

Complejidad: O(1), ya que es independiente del largo de la pila.

3.1.3. Sacar (pop)

Sacar un elemento saca el nodo en el principio de la lista y actualiza el `tope` para apuntar al siguiente. Para esto utilizamos un puntero auxiliar al `tope`, luego reasignamos el `tope` al `siguiente` de `tope`. Con el puntero auxiliar apuntando al nodo quitado, podemos eliminarlo o devolver el elemento.

Si la lista tenía un solo elemento, `tope` quedará NULL.

Complejidad: O(1), ya que es independiente del largo de la pila.

3.1.4. Tope

Para ver el elemento en el tope copiamos a un puntero auxiliar el puntero `tope`, para ser utilizado.

Complejidad: O(1), ya que se trata simplemente de seguir un puntero.

3.1.5. Está vacía

Para verificar si la pila está vacía podemos verificar que `tope` apunte a NULL, o el contado de la lista es 0.

Complejidad: O(1), ya que se trata de una única verificación.

3.1.6. Iterar

Un iterador, que puede definirse como un TDA que incluya la pila sobre la cual iterar y el elemento al cual está apuntando, permite acceder a cada nodo o elemento de la pila desde el `tope` hasta el final.

Para ello inicializamos un puntero al `tope` y luego para cada nodo reasignamos el elemento actual del iterador al `siguiente` del actual, hasta llegar al final de la pila o el nodo u elemento buscado.

Complejidad: O(n), ya que se realizarán n operaciones para seguir cada elemento de la pila, con n el largo de la pila.

3.1.7. Destruir

Para destruir una pila es necesario liberar los recursos utilizados por cada nodo, y luego por la pila. Para ello iteramos sobre cada nodo, liberando los recursos utilizados, y cuando la pila se encuentre vacía, liberar los recursos utilizados por la pila.

Complejidad: O(n), ya que se debe destruir el elemento contenido en cada nodo.

3.2. Cola con vectores estáticos (cola circular)

Una cola circular es similar a una cola, en el sentido de que tiene un `inicio` y un `fin`, y las operaciones que se pueden realizar son similares. La particularidad de la cola circular reside en el hecho de que el principio y el final del vector se ̈juntan ̈para dar lugar a un vector continuo. Para lograr esto se definen cada posición o índice como:

$$(\texttt{posicion} + 1)\,\%\texttt{largo} = \texttt{indice} \tag{1}$$

Con esto logramos que al llegar a `posicion` + 1 igual al largo del vector, el índice reinicie en cero.

Al ser implementada con vectores estáticos el largo de la cola será predeterminado. Las ventajas de la cola circular respecto a la cola son

A continuación detallamos la implementación de las distintas operaciones.

3.2.1. Crear

Para crear una cola circular definimos un vector del largo deseado, e iniciamos las variables `inicio` y `fin` en la posición 0 del vector.

Complejidad: O(1), ya que se trata únicamente de inicializar la cola.

3.2.2. Encolar (queue)

Para encolar un elemento en la lista circular se agrega el elemento en la posición `fin` + 1, y se actualiza `fin` a `fin` +1.

Complejidad: O(1), ya que se trata de agregar un elemento en una posición a la cual se tiene acceso.

3.2.3. Desencolar (enqueu)

Se quita el elemento en la posición `inicio` y se actualiza `inicio` a `inicio` +1.

Complejidad: O(1), ya que se trata de quitar un elemento en una posición a la cual se tiene acceso.

3.2.4. Ver primero

Para ver el primer elemento se accede a la posición `inicio` del vector cola.

Complejidad: O(1), ya que se trata de ver el elemento en una posición a la cual se tiene acceso.

3.2.5. Iterar

Para iterar en un vector actualizamos el índice desde la posición `inicio` hasta `fin` o hasta llegar al elemento buscado.

Complejidad: O(n), ya que se realizarán n operaciones para seguir cada elemento de la cola, con n el largo de la cola.

3.2.6. Destruir

En el caso de los vectores son automáticos destruidos al salir de la función donde tengan validez, por esto, además, su complejidad es O(1).

3.3. Lista con nodos simplemente enlazados

Esta implementación de listas es muy versátil y poderosa, pero requiere de más cuidado por parte del programador ya que es más sensible el manejo de punteros y memoria. Al basarse en nodos enlazados, no es posible acceder a los elementos sino a través de los punteros de cada nodo, por lo que es imperativo no perder jamás referencia de los elementos siguientes.

A continuación se detallan las implementaciones de las funciones de listas enlazadas.

3.3.1. Crear

Para crear una lista utilizamos un TDA que incluye el puntero al principio de la lista (`head`) y al final (texttttail), y opcionalmente un contador con la cantidad de elementos. Se inicializan los punteros en NULL y el contador en 0.

Complejidad: O(1), ya que únicamente se trata de crear el TDA.

3.3.2. Insertar

Esta función permite insertar un elemento al final de la lista. Para ello creamos un nuevo nodo, cuyo `siguiente` apunte a NULL. Si la lista está vacía, apuntamos tanto el `head` como el `tail` al nuevo nodo. Si la lista no está vacía, actualizamos el siguiente del `tail` de NULL al nuevo nodo.

Complejidad: O(1), ya que es independiente del largo de la lista.

3.3.3. Insertar en posición

Para insertar en una posición arbitraria de la lista, primero se verifica si la lista lista está vacía o la posición a insertar es el final. En este caso, se procede igual que en insertar.

Si la posición a insertar es la inicial, se apunta el `siguiente` del nuevo nodo al `head`, luego se actualiza el `head` para que apunte al nuevo nodo.

Si la posición a insertar es al medio, se usa un puntero auxiliar para apuntar al nodo anterior a aquel cuya posición se busca reemplazar. Se apunta el `siguiente` del nuevo nodo al `siguiente` del nodo anterior. Luego apuntamos el `siguiente` del nodo anterior al nuevo nodo.

Complejidad: O(n), ya que se debe iterar para encontrar la posición.

3.3.4. Quitar

Quitar un elemento saca el último elemento, apuntado por `tail`. Para ello se usa un puntero auxiliar que apuntamos a `tail`.

Luego, si el tamaño de la lista es 1, `head` y `tail` se apuntan a NULL.

Sino, se utiliza un puntero al penúltimo nodo. Luego se apunta el `tail` a dicho nodo, y su `siguiente` a NULL.

Complejidad: O(1), ya que es independiente del largo de la pila.

3.3.5. Quitar de posición

Para quitar un elemento en una posición arbitraria de la lista, primero se verifica que la posición de donde quitar no sea la final. Si es así, se procede igual que en quitar.

Sino, se utiliza un puntero auxiliar. Si la posición de donde se quiere sacar es la inicial, apuntamos este auxiliar al `head`. Luego reasignamos el `head` a su siguiente. Si, además, el tamaño de la lista es 1, apuntamos el `tail` a NULL. En este caso `head` también apuntará a NULL.

Si se quiere sacar un elemento "del medio", apuntamos el auxiliar al `siguiente` del nodo anterior, y luego actualizamos este último a `siguiente` del auxiliar.

Complejidad: O(n), ya que se debe iterar para encontrar la posición.

3.3.6. Está vacía

Se verifica que `head` y `tail` apunten a NULL, o que la cantidad de elementos sea 0.

Complejidad: O(1), ya que se trata de una única verificación.

3.3.7. Elemento en posición

Para obtener el elemento en una posición arbitraria, se itera hasta el nodo en la posición buscada y se devuelve el elemento que contiene.

Complejidad: O(n), ya que se debe iterar para encontrar la posición.

3.3.8. Destruir

Para destruir una pila es necesario liberar los recursos utilizados por cada nodo, y luego por la pila. Para ello iteramos sobre cada nodo, liberando los recursos utilizados, y cuando la pila se encuentre vacía, liberar los recursos utilizados por la pila.

Complejidad: O(n), ya que se debe destruir el elemento contenido en cada nodo.

4. Implementación y decisiones de diseño

En esta sección se explayará sobre algunos puntos específicos a la implementación de quien suscribe, dividida por cada TDA desarrollado (pila, cola y lista).

En primer lugar, considerando que pila y cola se pueden considerar un subconjunto restringido del TDA lista, se comenzó por desarrollar el TDA lista.

4.1. TDA lista

Guiándose por las descripciones de las funciones provistas por la cátedra y luego de una exhaustivo estudio de las distintas operaciones, así como sus posibles casos borde, en base a dibujos con papel y lápiz, se comenzaron a desarrollar las funciones consideradas más "básicas". El criterio para determinar esto fue principalmente que requieren únicamente de una simple verificación o no requieran otras funciones accesorias para implementarse. Estas funciones fueron: `lista_crear`, `lista_primero`, `lista_ultimo`, `lista_vacia` y `lista_tamanio`, que consisten básicamente en devolver el elemento en `head`, `tail` o la cantidad de elementos de la lista.

Para crear la lista se asigna memoria con tamaño de `struct lista`, y se inicializan los punteros a NULL y el contador en 0.

En el caso de `lista_quitar` y `lista_insertar` se siguieron los algoritmos vistos anteriormente, verificando adecuadamente que los punteros provistos y los nodos creados sean válidos; y actualizando el tamaño de la lista como corresponda.

Para las funciones `lista_quitar_de_posicion` y `lista_insertar_en_posicion` se tuvo que desarrollar alguna función que permita llegar a los nodos auxiliares necesarios (nodo anterior por ejemplo). En primera instancia se utilizó un ciclo `while` para llegar al nodo en cuestión, pero para la implementación final se desarrollaron las funciones de iteración, y finalmente se modularizó a una función que devuelva el nodo en cierta posición.

Las funciones de iteración no requieren mayor análisis ya que consisten en simples verificaciones o actualizaciones de punteros.

4.1.1. `nodo_en_posicion`

Esta función recibe como parámetros un puntero a la lista, y un `size_t` con la posición buscada. Devuelve un puntero al nodo en la posición buscada o NULL en caso de error.

Luego de verificar que la lista sea válida y no esté vacía, se crea un puntero `lista_iterador_t` y un puntero `nodo_t`. Si la posición buscada es 0, nodo será el primer elemento del iterador. Luego se destruye el iterador y se devuelve el puntero al nodo buscado.

Si la posición buscada es otra, se itera en un ciclo `for` hasta la posicion buscada, avanzando el iterador en cada iteración. Luego se asigna a nodo el nodo apuntado por el iterador, se destruye el iterador y se devuelve el nodo.

Esta función permite abstraer del código cualquier parte que requiera avanzar hasta un nodo, generalmente útil para nodos auxiliares.

4.1.2. `lista_quitar_de_posicion` y `lista_insertar_en_posicion`

Luego de verificar la validez de los punteros provistos, se verifica que la posición a quitar o insertar no sea la última. Si es así, se invoca y devuelve directamente `lista_quitar` `lista_insertar`. Sino se procede con el algoritmo ya visto, utilizando `nodo_en_posicion` para los auxiliares.

4.1.3. `lista_con_cada_elemento` o iterador interno

Este iterador permite iterar sobre cada elemente contenido en los nodos. Recibe como parámetros punteros a la lista, a una función que se habrá de aplicar sobre cada elemento, y un contexto que se usará como parámetro de la función provista como parámetro. Devuelve un `size_t` que puede ser 0 en caso de error o lista vacía, o la cantidad de elementos iteradoa. En esta función, luego de verificar que la función y la lista sean punteros válidos, y la lista no sea vacía, se crea un iterador. En un ciclo `while`, mientras el elemento actual del iterador sea distinto de NULL (o sea que llegamos al final de la lista) y la función provista por parámetro aplicada al elemento actual no sea `false`, se continúa avanzando el iterador y sumando al contador de elementos iterados (habiendo sido inicializado en 1). Si se llegó al final de la lista se devuelve el contador -1, sino, el contador.

4.2. TDA cola

Se utilizaron funciones de lista casteadas al tipo `cola_t`, así, `cola_crear` devuelve `(cola_t *)lista_crear();`, encolar inserta en posición 0 de la lista, desencolar utiliza quitar de lista.

4.3. TDA pila

Al igual que en cola, se implementó la pila utilizando las funciones de lista y casteando. Así, `pila_crear` devuelve `(pila_t *)lista_crear();`, apilar inserta en posición 0 de la lista, desapilar utiliza quitar de posición de la lista.

5. Compilación y ejecución

En esta ocasión ha sido provisto un `makefile` que compila el programa con `gcc` y varias `flags`. Se pueden ejecutar el archivo `ejemplo.c` y `pruebas.c` con o sin `Valgrind`. Para ejecutar el programa de pruebas con Valgrind:

```
make valgrind-pruebas
```

6. Pruebas

Como parte del trabajo fue necesario desarrollar una serie de pruebas para probar el funcionamiento deseado del programa y diferentes casos borde.

Las funciones de pruebas desarrolladas fueron:

- `creo_una_lista_y_devuelve_una_lista_vacia()`: