Práctico 3

Objetivos

- Trabajar sobre estructuras lineales en memoria dinámica, accediendo directamente o usando operaciones.
- Poner el foco en la diferencia entre el pasaje de parámetros por copia y el pasaje de parámetros por referencia, teniendo en cuenta los efectos secundarios de cada uno.
- Trabajar con funciones y procedimientos tanto totales, como parciales (que contemplan precondiciones).

Ejercicio 1 Lista simple

Considere la representación para una Lista Encadenada de Enteros que se presenta en la Figura 1.

```
struct nodo {
    int elem;
    nodo *sig;
};
typedef nodo *lista;
```

Figura 1: Lista Encadenada de enteros.

(a) Utilizando la representación de la Figura 1, implemente las siguientes **funciones** o **procedimientos** en forma **iterativa** y **sin usar procedimientos auxiliares**. En el caso de inserciones o eliminaciones se debe modificar los parámetros de entrada y compartir memoria con los mismos.

I. `last`: dada una lista no vacía, retorna su último elemento.

II. `average`: dada una lista no vacía, retorna el promedio de sus elementos.

III. `insOrd`: dados un entero *x* y una lista *l* **ordenada**: inserta *x* en *l* ordenadamente. Comparar con la solución recursiva.

IV. `snoc`: dados un entero *x* y una lista *l*, inserta *x* al final de *l*.

V. `removeAll`: dados un entero *x* y una lista *l*, elimina a *x* de *l*.

VI. `isIncluded`: dadas dos listas *l* y *p*, verifica si *l* está incluida en *p*. Una lista l_1 está incluida en l_2 si y sólo sí existen dos listas, l_3 y l_4, tal que cumplen que $l_2 = l_3 l_1 l_4$. Tenga en cuenta que las listas l_i, con $i \in \{1, ..., 4\}$ pueden ser vacías.

(b) Analice las operaciones anteriores indicando cuáles son totales y cuáles parciales. ¿Es posible en cada caso pensar en ambas variantes (total y parcial)?.

Ejercicio 2

Utilizando la representación para Lista Encadenada de Enteros de la Figura 1, implemente las siguientes **funciones** en forma **iterativa** y **sin usar procedimientos auxiliares**. Las soluciones retornadas **no deben compartir memoria** con los parámetros.

(a) `take`: dado un natural i y una lista *l*, retorna la lista resultado de tomar los primeros i elementos de *l*.

(b) `drop`: dado un natural i y una lista *l*, retorna la lista resultado de eliminar los primeros i elementos de *l*.

(c) `mergeSort`: dadas dos listas ordenadas *l* y *p*, genera una lista intercalando ordenadamente ambas listas.

(d) `concat`: dadas dos listas *l* y *p*, retorna una lista que contiene a los elementos de *l* y luego a los elementos de *p*, en el mismo orden. Comparar con la solución recursiva.

Ejercicio 3

(a) Utilizando la representación para Lista Encadenada de Enteros de la Figura 1, implemente los siguientes **procedimientos** de forma **recursiva**.

I. `insOrd`: dados un entero *x* y una lista *l* ordenada, inserta a *x* en *l* ordenadamente.

II. `snoc`: dados un entero *x* y una lista *l*, inserta a *x* al final de *l*.

III. `removeAll`: dados un entero *x* y una lista *l*, elimina a *x* de *l*.

(b) ¿Qué diferencias encuentra entre las soluciones de este ejercicio y las que realizó para el ejercicio 1?

Ejercicio 4 Lista doblemente encadenada

Una variante de listas encadenadas es la llamada **Lista Doblemente Encadenada**. En dicha implementación cada elemento de la lista referencia al siguiente elemento y al elemento anterior. La Figura 2 presenta una representación para esta variante.

```
struct nodo_doble {
  int elem;
  nodo_doble *sig;
  nodo_doble *ant;
};
typedef nodo_doble *lista;
```

Figura 2: Lista Doblemente encadenada de enteros.

(a) ¿Qué ventajas y desventajas presenta esta representación respecto a la presentada en la Figura 1?

(b) Implementar las siguientes operaciones utilizando la representación presentada en la Figura 2:

I. `null`: retorna una lista vacía.

II. `cons`: dados un entero *x* y una lista *l*, retorna el resultado de insertar *x* al principio de *l*.

III. `isEmpty`: dada una lista *l*, verifica si *l* está vacía.

IV. `isElement`: dados un entero *x* y una lista *l*, verifica si *x* pertenece a *l*.

V. `removeAll`: dados un entero *x* y una lista *l*, retorna el resultado de eliminar todas las ocurrencias de *x* de *l*.

VI. `insOrd`: dados un entero *x* y una lista *l* ordenada, retorna el resultado de insertar *x* en *l* ordenadamente.

Ejercicio 5 Lista indizada (Manejo explícito de posiciones)

Una variante importante de listas son las Listas con manejo explícito de posiciones (a veces llamadas Listas Indizadas). Las operaciones más conocidas de estas listas ya fueron vistas anteriormente en el práctico 1 y en este práctico se quiere implementarlas mediante estructuras pedidas en forma dinámica.

En estas listas, se manipulan los elementos mediante posiciones, generalizando la idea de los arreglos para estructuras no acotadas. La posición del primer elemento es la posición 0. Se dice que la posición p está definida si está entre 0 y m -1, siendo m la longitud de la lista. Utilizando la representación de lista doblemente encadenada, implemente las siguientes operaciones:

(a) `isDefined`: dados un natural p y una lista *l*, retorna verdadero si, y solamente si, existe un elemento en esa posición.

(b) `insert`: dados un entero *x*, un natural p y una lista *l* de longitud m, inserta a *x* en la lista. Si p no esta definida, inserta a *x* en la posición m. En otro caso, inserta a *x* en la posición p y desplaza en una posición los elementos que estuvieran en las posiciones siguientes.

(c) `element`: dados un natural p y una lista *l*, retorna el elemento en la posición p. Tiene como precondición que la posición p esté definida.

(d) `remove`: dados un natural p y una lista *l*, elimina de la lista el elemento que se encuentra en la posición p. Si la posición no está definida, la operación no tiene efecto. Si la posición está definida, elimina el elemento en dicha posición y desplaza en una posición los elementos que estuvieran en las posiciones posteriores a p (contrae la lista).

```
struct nodo{
    int elem;
    nodo *sig;
};
struct cabezal{
    nodo  *primero;
    nodo  *actual;
};
typedef cabezal *lista;
```

Figura 3: Lista indizada.

Ejercicio 6 Lista indizada (Manejo implícito de posiciones)

En las Listas con manipulación de elementos por posiciones, éstas se pueden manejar de forma explícita o implícita. En el ejercicio anterior el manejo es explícito. Veremos en este ejercicio el manejo implícito. Una posible implementación para estas listas consiste en, además de la referencia al *inicio* de la estructura, mantener otra posición (llamémosle posición *actual*). El tipo `lista` que se presenta de la Figura 3 implementa esta variante. Usando esta representación implemente las siguientes operaciones:

(a) `null`: crea una lista vacía.

(b) `start`: dada una lista *l* no vacía, coloca la posición actual al inicio de *l*.

(c) `next`: dada una lista *l* no vacía, mueve la posición actual al siguiente nodo (elemento). En caso de que la posición actual sea el final de la lista, coloca la posición actual al inicio de la lista (tiene un comportamiento circular).

(d) `insert`: dados un entero *x* y una lista *l*, inserta el elemento *x* luego de la posición actual en la lista. La posición actual pasa a ser el elemento (nodo) recién insertado. Si la lista *l* está vacía, el resultado es la lista unitaria que contiene a *x*, siendo este elemento la posición actual en la lista resultado.

(e) `element`: dada una lista *l* no vacía, retorna el elemento en la posición actual de *l*.

Ejercicios complementarios

Ejercicio 7 Lista circular

Otra variante para listas encadenadas es la llamada **Lista Encadenada Circular**. En esta implementación el último elemento de la lista referencia al primero.

(a) Proponga una representación diferente a la de las Figuras 1 y 2 para representar esto.

(b) ¿Qué ventajas y desventajas presenta esta representación respecto a las presentadas en las Figuras 1 y 2?

(c) Implementar las siguientes operaciones:

I. `null`: retorna una lista vacía.

II. `isEmpty`: dada una lista *l*, verifica si *l* está vacía.

III. `tail`: dada una lista *l* no vacía, retorna la lista sin su primer elemento.

IV. `last`: dada una lista *l* no vacía, retorna su último elemento.

V. `insOrd`: dados un entero *x* y una lista *l* ordenada, retorna el resultado de insertar *x* en *l* ordenadamente.