Examen Final del Taller - Libres

Algoritmos y Estructuras de Datos II

Tarea

El alumno deberá implementar el tipo abstracto de dato *ListCalc* en el lenguaje de programación C, utilizando la técnica de ocultamiento de información visto en el taller de la materia.

Es requisito mínimo para aprobar lo siguiente:

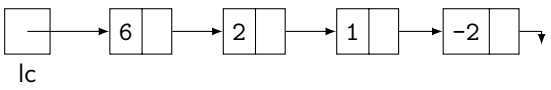
- Implementar en C el TAD *ListCalc* (utilizando punteros a estructuras y manejo dinámico de memoria).
- Implementar en C una función `main`, detallada más abajo.
- El programa resultado no debe tener *memory leaks* ni accesos inválidos a memoria, se chequeará tal condición usando `valgrind`.
- Las funciones deben ser **NO** recursivas.
- **NO** se pueden usar variables globales.
- **NO** se puede hacer una implementación alternativa a la que exige el enunciado.

El TAD ListCalc

El tipo *ListCalc* representa una "lista de cálculo", cuya definición formal se dio en el exámen teórico (ver apéndice al final). La implementación a utilizar será la siguiente:

```
type list_calc = pointer to node
type node = tuple
              value: int
              next: pointer to node
            end
```

Notar que usaremos una **lista enlazada** de números enteros. Por ejemplo, la siguiente es la representación gráfica de una lista de cálculo `lc` de 4 enteros:

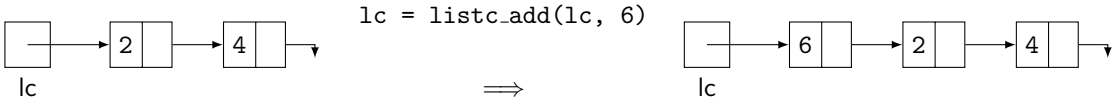


y la lista vacía se representa con un puntero nulo:

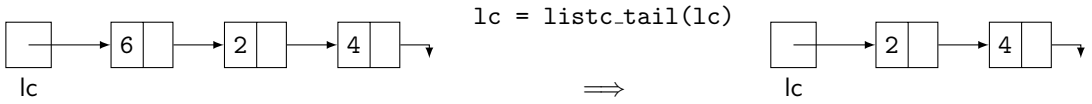
Funciones del TAD

La siguiente es la lista de funciones que debe proveer el TAD:

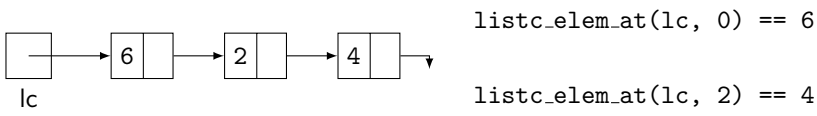
- `listc_t listc_empty(void)`. Devuelve una lista vacía.
- `listc_t listc_add(listc_t lc, int e)`. Agrega un elemento al principio de la lista.



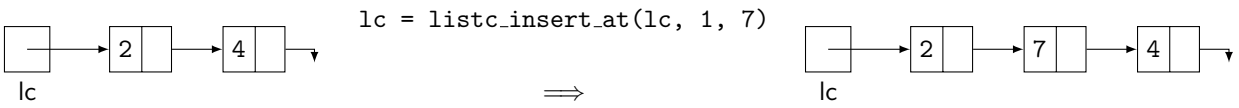
- `bool listc_is_empty(listc_t lc)`. Devuelve `true` si la lista es vacía, y `false` caso contrario.
- `listc_t listc_tail(listc_t lc)`. Elimina el primer elemento y devuelve el resto de la lista.

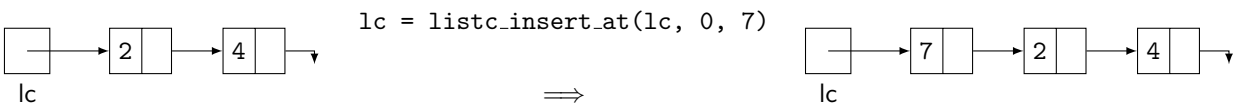


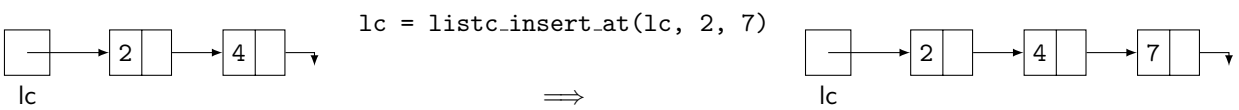
- `unsigned int listc_length(listc_t lc)`. Devuelve la longitud de la lista. Esta función debe ser lineal en la cantidad de nodos.
- `int listc_elem_at(listc_t lc, unsigned int pos)`. Devuelve el elemento que se encuentra en la posición dada, contando a partir de 0 y de izquierda a derecha. Asume que la posición es válida.



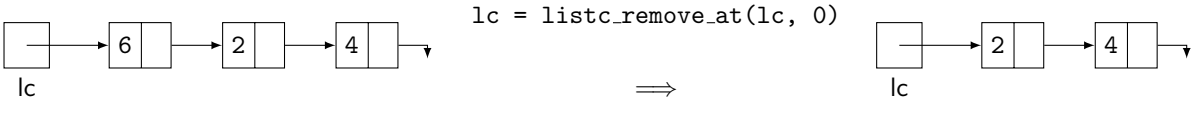
- `listc_t listc_insert_at(listc_t lc, unsigned int pos, int e)`. Inserta un elemento en la posición dada. Asume que `pos <= listc_length(lc)`. Por ejemplo:

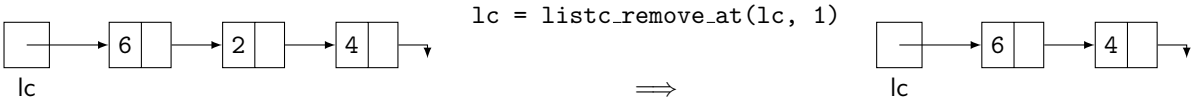




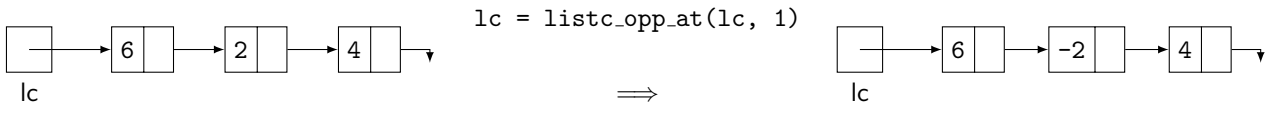


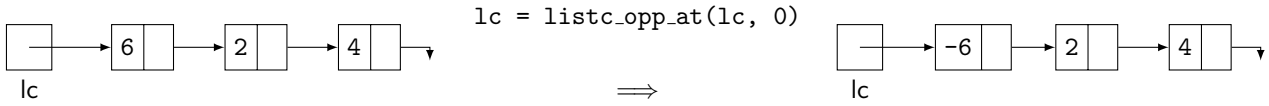
- `listc_t listc_remove_at(listc_t lc, unsigned int pos)`. Elimina el elemento que se encuentra en la posición dada. Asume que la posición es válida.

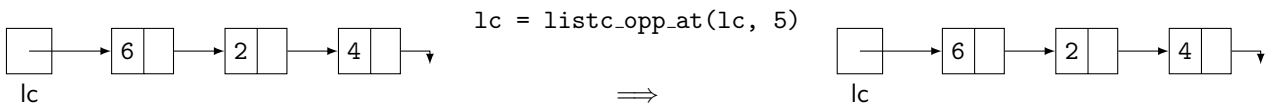




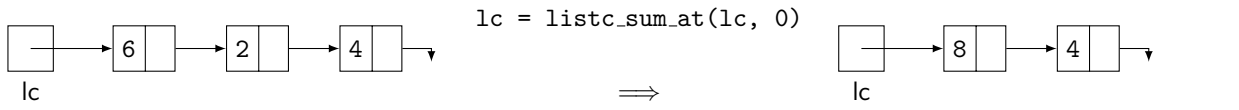
- `listc_t listc_opp_at(listc_t lc, unsigned int pos)`. Reemplaza un elemento en la posición dada por su opuesto. Si la posición es inválida, no hace nada.

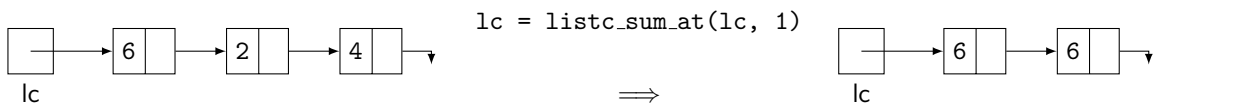


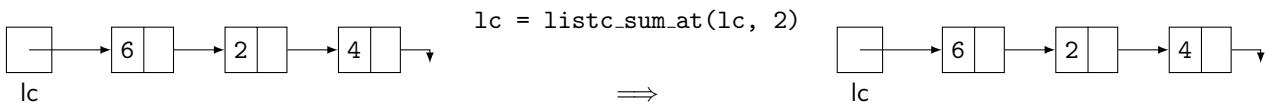




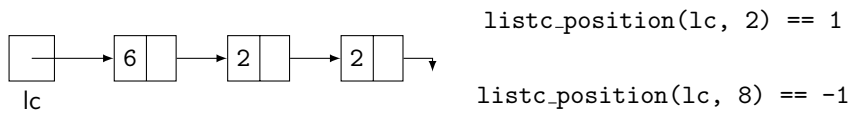
- `listc_t listc_sum_at(listc_t lc, unsigned int pos)`. Elimina los elementos de las posiciones `pos` y `pos+1`, y los reemplaza por su suma. Si `pos` o `pos+1` es una posición inválida, no hace nada.



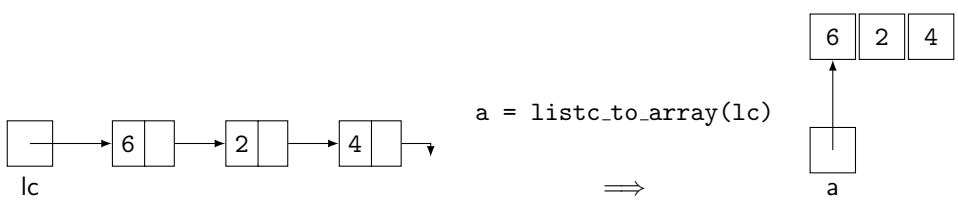




- `void listc_dump(listc_t lc, FILE *fd)`. Imprime los elementos de la lista, separados por espacio, y de izquierda a derecha.
- `listc_t listc_destroy(listc_t list)`. Libera los recursos de memoria utilizados por la lista.
- `int listc_position(listc_t list, int e)`. Devuelve la posición de la primera ocurrencia en la lista del elemento dado, si esa ocurrencia no existe, devuelve `-1`.



- `int *list_to_array(listc_t list)`. Devuelve un arreglo con todos los elementos de la lista.



Función Main

Escribir una función `main` que realice en orden los siguientes pasos, y *luego de cada paso imprimir en pantalla el resultado*. Para imprimr una lista usar la función `show_list` que se muestra más abajo.

- Crear una lista vacía: `lc = list_empty()`.

  `=> [ ] {0}`

- Usar `lc = list_add(lc, x)` para agregar los elementos del `x=0` hasta `x=7`.

  `=> [ 7 6 5 4 3 2 1 0 ] {8}`

- `lc = listc_tail(lc)`.

  `=> [ 6 5 4 3 2 1 0 ] {7}`

- `lc = listc_insert_at(lc, 0, -1)`.

  `=> [ -1 6 5 4 3 2 1 0 ] {8}`

- `lc = listc_insert_at(lc, 2, 8)`.

  `=> [ -1 6 8 5 4 3 2 1 0 ] {9}`

- `lc = listc_insert_at(lc, list_length(lc), 10)`.

  `=> [ -1 6 8 5 4 3 2 1 0 10 ] {10}`

- `lc = listc_remove_at(lc, 0)`.

  `=> [ 6 8 5 4 3 2 1 0 10 ] {9}`

- `lc = listc_remove_at(lc, 4)`.

  `=> [ 6 8 5 4 2 1 0 10 ] {8}`

- `lc = listc_remove_at(lc, list_length(lc) - 1)`.

  `=> [ 6 8 5 4 2 1 0 ] {7}`

- `lc = listc_opp_at(lc, 3)`.

  `=> [ 6 8 5 -4 2 1 0 ] {7}`

- `lc = listc_sum_at(lc, 0)`.

  `=> [ 14 5 -4 2 1 0 ] {6}`

- `lc = listc_sum_at(lc, 1)`.

  `=> [ 14 1 2 1 0 ] {5}`

- `lc = listc_sum_at(lc, 3)`.

  `=> [ 14 1 2 1 ] {4}`

- `lc = listc_sum_at(lc, 4)`.

  `=> [ 14 1 2 1 ] {4}`

- `pos = listc_position(lc, 15).`

  `=> -1`

- `pos = listc_position(lc, 14).`

  `=> 0`

- `pos = listc_position(lc, 1).`

  `=> 1`

- `a = listc_to_array(lc).`

  `=> 14 1 2 1`

Usar la siguiente función para imprimir listas:

```
void show_list(listc_t list) {
    printf("[ ");
    list_dump(list, stdout);
    printf("] {%d}\n", list_length(list));
}
```

(pegar el código arriba del `main`). Liberar lo que corresponda.

Archivos a entregar

En resumen, se deben entregar los siguientes archivos:

- `listc.h`, el archivo de cabeceras.
- `listc.c`, con la implementación de las funciones.
- `main.c`, con la función main pedida.

Recordar

- Se debe resolver el ejercicio en cuatro horas (o menos).
- Se debe compilar pasando todos los flags usados en los proyectos.
- Comentar e indentar el código apropiadamente, siguiendo el estilo de código ya indicado por la cátedra (indentar con 4 espacios, no pasarse de las 80 columnas, inicializar todas las variables, etc).
- Todo el código tiene que usar la librería estándar de C, y no se puede usar extensiones GNU de la misma.
- El programa resultante **no** debe tener *memory leaks* **ni** accesos (read o write) inválidos a la memoria.
- Las funciones deben ser **NO** recursivas.
- **NO** se pueden usar variables globales.
- **NO** se puede hacer una implementación alternativa a la que exige el enunciado.

Apéndice: Definición formal del TAD

TAD calcu_lista
constructores
vacía : calcu_lista
agregar : entero $\times$ calcu_lista $\rightarrow$ calcu_lista
operaciones
es_vacía : calcu_lista $\rightarrow$ booleano
cabeza : calcu_lista $\rightarrow$ entero {sólo se aplica a listas no vacías}
cola : calcu_lista $\rightarrow$ calcu_lista {sólo se aplica a listas no vacías}
largo : calcu_lista $\rightarrow$ natural
elemento : calcu_lista $\times$ natural $\rightarrow$ entero {sólo se aplica a pares (l, n) con $n <$ largo(l)}
insertar : calcu_lista $\times$ natural $\times$ entero $\rightarrow$ calcu_lista {sólo se aplica a ternas (l, n, e) con $n \leq$ largo(l)}
borrar : calcu_lista $\times$ natural $\rightarrow$ calcu_lista {sólo se aplica a pares (l, n) con $n <$ largo(l)}
opuesto : calcu_lista $\times$ natural $\rightarrow$ calcu_lista
mas : calcu_lista $\times$ natural $\rightarrow$ calcu_lista
ecuaciones
es_vacía(vacía) = verdadero
es_vacía(agregar(e,l)) = falso
cabeza(agregar(e,l)) = e
cola(agregar(e,l)) = l
largo(vacía) = 0
largo(agregar(e,l)) = 1 + largo(l)
elemento(agregar(e,l),0) = e
elemento(agregar(e,l),n+1) = elemento(l,n)
insertar(l,0,e) = agregar(e,l)
insertar(agregar(e',l),n+1,e) = agregar(e',insertar(l,n,e))
borrar(agregar(e,l),0) = l
borrar(agregar(e,l),n+1) = agregar(e,borrar(l,n))
opuesto(vacía,n) = vacía
opuesto(agregar(e,l),0) = agregar(-e,l)
opuesto(agregar(e,l),n+1) = agregar(e,opuesto(l,n))
mas(vacía,n) = vacía
mas(agregar(e,vacía),n) = agregar(e,vacía)
mas(agregar(e,agregar(e',l)),0) = agregar(e+e',l)
mas(agregar(e,agregar(e',l)),n+1) = agregar(e,mas(agregar(e',l),n))