Práctica 7: Listas

Estructuras de Datos y Algoritmos I

Autor: José Mauricio Matamoros de Maria y Campos

1. Objetivo

El alumno revisará las definiciones, características, procedimientos y ejemplos de las estructuras lineales *lista ligada* y *lista basada en arreglos* tanto en sus versiones de extremos abiertos como circulares para poder comprenderlas a fin de ser capaz de implementarlas.

2. Material

Se asume que el alumno cuenta con un una computadora con arquitectura Intel x86 o compatible, sistema operativo Linux/Unix, y compilador `gcc` instalado.

3. Instrucciones

1. Lea detenidamente los antecedentes teóricos sobre la construcción de colas circulares y colas dobles Sección 4.
2. Realice cada una de las actividades detalladas en la Sección 5 apoyándose en el código de ejemplo del Apéndice A y responda las preguntas de las mismas.

4. Antecedentes

Una lista es un tipo de datos abstracto que contiene una sucesión finita de elementos ordenados y repetibles. Por ejemplo:

$$L : \{1, 1, 2, 3, 5, 8, 13\}$$

Por definición, una estructura de datos tipo lista L soporta las siguientes operaciones:

- $L.count()$ devuelve el número de elementos en la lista L.
- $L.empty()$ verdadero si la lista L está vacía, falso en otro caso.
- $L.insert(i, x)$ inserta al elemento x en la i-ésima posición en la lista L.
- $L.delete(i)$ elimina al elemento x de la i-ésima posición en la lista L.
- $L.search(x)$ busca la primer ocurrencia del x en la lista L y devuelve su índice, o -1 si $x \notin L$.
- $L[i]$ accede al elemento x en la i-ésima posición en la lista L.

Derivado de $L.insert(i, x)$ se pueden abstraer dos casos generales aplicables a cualquier lista: *append* cuando $i = count$ y *prepend* cuando $i = 0$. Así:

- $L.prepend(x)$ inserta al elemento x al principio de la lista L.
- $L.append(x)$ inserta al elemento x al final de la lista L.

Una lista puede tener una de varias formas. Puede ser ligada sencilla o doblemente ligada, puede ser ordenada o no, y puede ser circular o no. Si una lista es sencillamente ligada se omite el apuntador prev en cada elemento. Si la lista es ordenada, el orden lineal de la lista corresponde al orden lineal de las llaves-valor almacenadas en los elementos de la lista. En este caso, el elemento mínimo es entonces el encabezado de la lista y el elemento máximo

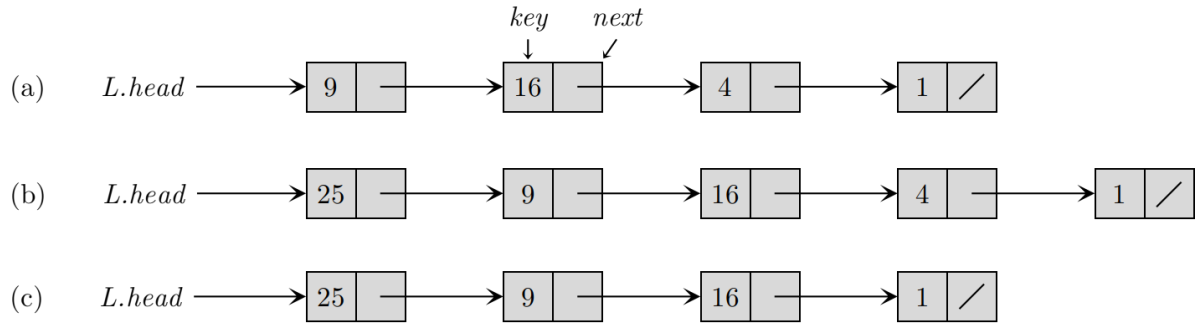

Figura 1: **(a)** Una lista L ligada que representa el conjunto dinámico $L = \{1, 4, 9, 16\}$.Cada elemento de la lista es un objeto con atributos para la llave y un apuntador (mostrado con una flecha) al objeto siguiente o NIL, indicado por una barra diagonal. El atributo $L.head$ apunta a la cabeza. **(b)** Tras la ejecución de LIST-PREPEND(L, x), donde $x.key = 25$, la lista ligada tiene un nuevo objeto con llave-valor 25 como nueva cabeza. Este nuevo objeto apunta a la cabeza anterior con llave-valor 9. **(c)** El resultado de la siguiente llamada LIST-DELETE(L, x), donde x apunta al objeto con la llave-valor 4.

es la cola. Si la lista no es ordenada, los elementos pueden aparecer en cualquier orden. En una lista circular, el apuntador *prev* de la cabeza de la lista apunta a la cola y el apuntador *next* de la cola de la lista apunta a la cabeza. Se puede pensar en una lista circular como un anillo de elementos.

4.1. Listas ligadas

Una lista enlazada o ligada es una estructura de datos en la que los objetos se organizan en orden lineal. Sin embargo, a diferencia de un arrglo en el que el orden está determinado por los índices, el orden en una lista ligada lo determina un apuntador en cada eslabón u objeto. Las listas ligadas ofrecen una representación simple y flexible para conjuntos dinámicos que admiten (aunque no necesariamente de manera eficiente) todas las operaciones simples (búsqueda, inserción, eliminación, mínimo, máximo, sucesor y predcesor).

Una lista enlazada simple (o lista ligada) requiere sólo de un elemento: un apuntador al primer elemento de la lista o nodo, y cada nodo n estará formado por una tupla *llave-apuntador* donde $n.key$ es la llave o valor almacenado y $n.next$ es un apuntador al siguiente elemento en la secuencia.

Como se muestra en la Figura 1, cada elemento de una lista L doblemente ligada es un objeto con una llave-valor y un apuntador $x.next$ o elemento siguiente. El objeto también puede contener adicionales o bien tener como llave-valor un apuntador a una estructura más compleja u objeto. Dado un elemento x en la lista, $x.next$ apunta a su sucesor en la lista vinculada. Si $x.next =$ NIL, el elemento x no tiene sucesor y, por lo tanto, es el último elemento (o cola) de la lista. Un atributo $L.head$ apunta al primer elemento de la lista. Si $L : head =$ NIL, la lista está vacía.

4.2. Búsqueda en una lista ligada

El procedimiento LIST-SEARCH(L, k) encuentra el primer elemento con la llave k en la lista L mediante una búsqueda lineal simple, devolviendo un apuntador a este elemento. Si no encuentra ningún objeto con la llave k en la lista, el procedimiento devuelve NIL. Para la lista ligada en Figura 1(a), la llamada a LIST-SEARCH$(L, 4)$ devuelve un apuntador al tercer elemento, y la llamada a LIST-SEARCH$(L, 7)$ devuelve NIL.

Algoritmo 1 Búsqueda de un nodo con llave k en la lista L

```
procedure LIST-SEARCH(L, k)
    x = L.head
    while x ≠ NIL and x.key ≠ k do
        x = x.next
    end while
    return x
end procedure
```

Buscar en una lista de n objetos, el procedimiento LIST-SEARCH toma tiempo $\Theta(n)$ en el peor caso ya que podría tener que buscar en toda la lista.

4.3. Inserción en una lista ligada

Dado un elemento x cuya llave ya ha sido establecida, el procedimiento LIST-PREPEND «empalma» x al inicio de la lista ligada como se muestra en la Figura 1(b).

Algoritmo 2 Inserción de un nodo x al principio de la lista L

```
procedure LIST-PREPEND(L, x)
    x.next = L.head
    L.head = x
end procedure
```

Si, por otro lado, se desea insertar al nodo x al final de la lista L, habrá que desplazarse hasta el último elemento de la lista para realizar la inserción.

Algoritmo 3 Inserción de un nodo x al final de la lista L

```
procedure LIST-APPEND(L, x)
    if L.head == NIL then
        LIST-PREPEND(L, x)
    else
        n = L.head
        while n.next ≠ NIL do
            n = n.next
        end while
        n.next = x
    end if
end procedure
```

El proceso para la inserción generalizada es muy similar, sólo que en esta ocasión habrá que recorrer la lista hasta la posición i-ésima donde se desea insertar el nodo.

Algoritmo 4 Inserción de un nodo x en la i-ésima posición de la lista L

```
procedure LIST-INSERT(L, i, x)
    if L.head == NIL or i == 0 then
        LIST-PREPEND(L, x)
    else
        j = 0
        n = L.head
        while n.next ≠ NIL and j < i do
            n = n.next
            j = j + 1
        end while
        x.next = n.next
        n.next = x
    end if
end procedure
```

4.4. Eliminación de una lista ligada

El procedimiento LIST-DELETE elimina un elemento x de una lista ligada L. Se le debe asignar un apuntador a x, y luego «desacoplar» x fuera de la lista actualizando los apuntadores. Si se desea eliminar a un elemento con con base en su llave, primero se debe llamar a LIST-SEARCH para obtener un apuntador al elemento.

Esta operación sería trivial para una lista doblemente ligada, pues cada nodo conoce a su predecesor. En el caso de las listas ligadas simples la cosa se complica un poco pues hay que localizar a dicho predecesor.

Algoritmo 5 Inserción de un nodo x de la lista L

```
procedure LIST-DELETE(L, x)
    if L.head == x then
        L.head = x.next
    else
        n = L.head
        while n ≠ NIL and n.next ≠ x do
            n = n.next
        end while
        if n ≠ NIL then
            n.next = x.next
        else
            Error!
        end if
    end if
end procedure
```

La Figura 1(c) muestra cómo se elimina un elemento de una lista ligada. LIST-DELETE se ejecuta en tiempo $O(n)$, debido a que no se conoce al elemento predecesor, por lo que en el peor caso habrá que recorrer toda la lista para encontrarlo.

4.5. Listas ligadas circulares

Una lista ligada circular es aquella en el que el último elemento apunta al primer elemento de la lista; es decir $x_n.next = L.head$.

Los algoritmos para operar sobre listas circulares son exactamente los mismos, con la salvedad de que la condición $x.next == \text{NIL}$ nunca será verdadera pues la lista forma un anillo. Para romper esta condición de circularidad al buscar un elemento (LIST-SEARCH y LIST-APPEND), los bucles prueban si $x.next == L.head$ y detienen la búsqueda cuando esta condición se cumple.

4.6. Listas basadas en arreglos

Una lista basada en arreglos utiliza un arreglo como estructura base para implementar la lista. En una lista basada en arreglos, cada nodo conoce siempre a su predecesor y a su sucesor, por lo que se asemejan más a las listas doblemente ligadas que a las listas ligadas simples.

Si bien este tipo de implementación permite un acceso indizado más rápido a sus elementos (del orden de $O(1)$ por la cualidad intrínseca del arreglo de ser una estructura basada en índices), el modificar la lista se vuelve mucho más tardado pues todos los elementos del arreglo tendrán que reacomodarse cada vez que se elimina o inserta un nuevo elemento en la lista; especialmente cuando las modificaciones se hacen con base en un caché de nodos y no con base en búsquedas por llaves.

La implementación de este tipo de estructuras se deja a responsabilidad del alumno.

5. Desarrollo de la práctica

Lea cuidadosamente los problemas descritos a continuación e implemente la estructura de datos necesaria para hacer que los programas propuesto funcionen.

5.1. Actividad 1

El programa de la Apéndice A.2 requiere de una estructura de datos de tipo lista para operar, pero esta no está implementada.

Complete el programa de la Apéndice A.2 implementando la estructura de datos correspondiente, ejecute el programa y responda las preguntas que se presentan a continuación.

¿Qué estructura de datos requiere el programa? Explique [1 punto]: _____

¿Qué salida produce el programa? [1 punto]: _____

¿Cómo se implementó la función `insert`? Anote y explique su código a continuación [2 puntos]:

¿Cómo se implementó la función `delete`? Anote y explique su código a continuación [2 puntos]:

5.2. Actividad 2

El programa de la Apéndice A.2 requiere de una estructura de datos de tipo lista para operar, pero esta no está implementada.

Modifique la implementación del programa de la **Actividad 1** de tal forma que éste permita recorrer la lista en un bucle infinito y utilícela para que el programa de la Apéndice A.3 pueda funcionar correctamente.

¿Qué estructura de datos requiere el programa? Explique [1 punto]: _____

_____

_____

¿Qué salida produce el programa? [1 punto]: _____

_____

_____

¿Qué modificaciones realizó a la estructura de datos empleada? Anote y explique su código a continuación [2 puntos]:

A. Código de ejemplo

A.1. Archivo `src/Makefile`

src/Makefile

```
CC       = gcc
CFLAGS   = -Wall -O0
SILENT   = @
PROGRAMS = prog1 prog2 prog3
.PHONY: all clean $(PROGRAMS)

all: $(PROGRAMS)

$(PROGRAMS): %: %.c
  $(SILENT) $(CC) $(CFLAGS) -o $@ $<

clean:
  rm -f *.o
```

A.2. Archivo `src/prog1.c`

src/prog1.c

```c
#include <stdio.h>
#include <stdlib.h>

// Estructuras de datos
struct tnode{
  int value;            //key
  struct tnode* next;
} typedef node;

struct{
  node* first;
  int count;
} typedef list;

// Prototipos
void append(list* l, int x);
void insert(list* l, int ix, int x);
void delete(list* l, int ix);
int list_get(list* l, int ix);
int list_size(list* l);
void list_print(list* l);
list* list_init();
node* node_init(int value);


int main(void){
  int x;
  srabd(69);
  // 1. Inicializar estructura de datos
  list *l = list_init();

  // 2. Llenar la lista
  int n = 1 + rand() % 100;
  for(size_t i = 0; i < n; ++i){
    append(l, 1 + rand() % 1000);
  }
  printf("List size: %d\n", l->count);
  delete(l, rand() % list_size(l));
  delete(l, rand() % list_size(l));
  delete(l, rand() % list_size(l));
  printf("List size: %d\n", l->count);
  x = 1 + rand() % 100;
  insert(l, rand() % list_size(l), x);
  x = 1 + rand() % 100;
  insert(l, rand() % list_size(l), x);
  printf("List size: %d\n", l->count);

  // 3. Imprimir la lista
  list_print(l);
  return 0;
}

void list_print(list* l){
  printf("L: ");
  for(int ix = 0; ix < list_size(l); ++ix)
    printf("%d -> ", list_get(l, ix));
  printf("NULL\n");
}
```

A.3. Archivo `src/prog2.c`

src/prog2.c

```c
#include <stdio.h>
#include <stdlib.h>
void append(list* l, int x);
void insert(list* l, int ix, int x);
void delete(list* l, int ix);
node* list_first(list* l);
int value(node* node);
node* next(node* node);
int list_get(list* l, int ix);
int list_size(list* l);
void list_print_segment(list* l, int offset, int count);
list* list_init();
node* node_init(int value);


int main(void){
  int x;
  // 1. Inicializar estructura de datos
  list *l = list_init();
  // 2. Llenar la lista
  int n = 1 + rand() % 100;
  for(size_t i = 0; i < n; ++i){
    append(l, 1 + rand() % 1000);
  }

  printf("List size: %d\n", l->count);
  delete(l, rand() % list_size(l));
  delete(l, rand() % list_size(l));
  delete(l, rand() % list_size(l));
  printf("List size: %d\n", l->count);
  x = 1 + rand() % 100;
  insert(l, rand() % list_size(l), x);
  x = 1 + rand() % 100;
  insert(l, rand() % list_size(l), x);
  printf("List size: %d\n", l->count);

  // 3. Imprimir la lista
  list_print_segment(l, 25, 5);
  return 0;
}

void list_print_segment(list* l, int offset, int count){
  printf("L: ");
  node* n = list_first(l);
  int ix = 0;
  while((n != NULL) && (ix < offset)){
    ++ix;
    n = next(n);
  }
  while((n != NULL) && (count > 0)){
    printf("%d -> ", value(n));
    n = next(n);
    --count;
  }
  printf("NULL\n");
}
```

B. Reporte Escrito

El reporte de la práctica deberá ser entregada en un archivo en formato `PDF` siguiendo las siguientes especificaciones:

- La primera página del documento deberá ser la carátula oficial para prácticas de laboratorio disponible en `lcp02.fi-b.unam.mx/`
- El nombre del documento `PDF` deberá ser `nn-XXXX-L07.pdf`, donde:
  - `nn` es el número de lista del alumno a dos dígitos forzosos (ej. 01, 02, etc.).
  - `XXXX` corresponden a las dos primeras letras del apellido paterno seguidas de la primera letra del apellido materno y la primera letra del nombre, en mayúsculas y evitando cacofonías; es decir, los cuatro primeros caracteres de su RFC o CURP.
- El reporte consiste en un documento de redacción propia donde el alumno explica de forma concisa y a detalle las actividades realizadas en la práctica, y reportando los resultados obtenidos.
- La longitud del reporte no deberá exceder las 3 páginas, sin contar la carátula.
- El reporte deberá seguir todos los lineamientos para documentos escritos establecidos al inicio del curso.
- Todas las referencias deberán estar debidamente citadas.

IMPORTANTE: No se aceptan archivos en otros formatos ni con nombres distintos a los especificados.