Carnet _____ **Nombres** _____

CI2126 Computación II, Dic 2014/Mar 2015

Examen II (30%)

1.- (11 puntos, 0.75 de la (a) a la (l), y 2 pto la (m))
Indique con **V**(erdadero) o **F**(also) a las expresiones siguientes:

a.- "`typedef union S {int i; S* next} S;`" declara la variable "**S**" _**F**_

b.- Los apuntadores ocupan espacio en memoria proporcional al espacio referenciado _**F**_

c.- La inserción de valores en árboles binarios tiene un peor caso de orden $o(n^2)$ _**F**_

d.- *Selección* es de orden $o(n \log_2 n)$ en tiempo, y su peor caso también _**F**_

e.- *Quicksort* en promedio es $o(n \log_2 n)$ en tiempo, y su peor caso $o(n^2)$ _**V**_

f.- *Mergesort* requiere de espacio adicional $o(n \log_2 n)$ para resultados intermedios _**V**_

g.- La inserción remoción de valores ordenados en árboles es de orden $o(\log_2 n)$ _**V**_

h.- Una Cola puede ser usada como base para un TDA Lista _**V**_

i.- Un archivo binario de registros se puede leer completo de una vez _**V**_

j.- No se puede escribir información adicional a un archivo binario _**F**_

k.- *map*, *filter* y *reduce* se implementan igual en Listas y Arboles _**F**_

l.- De un árbol ordenado, se produce una lista ordenada si se recorre *preorden* _**F**_

m.- Si se elimina la *raíz* de un árbol ordenado, ¿el recorrido *posorden* del árbol resultante es el mismo tanto si se elimina el valor mayor por el hijo izquierdo como si se elimina el menor valor por el hijo derecho? ***(2pto, justifique)***

NO, porque la nueva raíz del árbol cambia, y por lo tanto ya no será igual, para ningún recorrido

2.- (6 puntos)

El algoritmo de Búsqueda Binaria en arreglos entre los límites **[**Li...Ls**)**, y un punto medio $Lm=\frac{1}{2}(Li+Ls)$, de tal manera que se está comparando el ***valor*** a buscar contra **`A[Lm]`** en cada recursión/iteración. Dependiendo si ***valor*** < ***A[Lm]*** hace recursión en **[**Li...Lm-1**)**, si **valor** > ***A[Lm]*** hace recursión en **[**Lm+1...Ls**),** si es igual a ***A[Lm]*** devuelve la posición (Lm). Si NO lo encuentra, o si *Li*>=*Ls* devuelve **-1**.

```
int BusqBinaria(int valor, int A[], int Li, int Ls);
```

En el algoritmo de Búsqueda Ternaria en arreglos visto en el laboratorio, se compara contra DOS valores intermedios, $Lm0=\frac{1}{3}(Li+Ls)$ y $Lm1=\frac{2}{3}(Li+Ls)$, por lo que hay dos valores con los que hacer comparación, [Li .. Lm0-1) ∪ [Lm0+1.. Lm1-1) ∪ [Lm1+1.. Ls) y 3 intervalos donde hacer recursión.

```
int BusqTernaria(int valor, int A[], int Li, int Ls);
```

Generalice y programe una función en C que haga *Búsqueda N-Aria*, es decir, calcula un arreglo de posiciones intermedias *Lm*, con cada $Lm[k]=\frac{k+1}{N}(Li+Ls),\ k=0..N-2$ y hace comparaciones y recursión de **N** intervalos. Si **N**=2 sería equivalente a Búsqueda Binaria, si **N**=3, sería Búsqueda Ternaria, etc.

```
int BusqNAria(int valor, int A[], int N, int Li, int Ld)
{
    int k;
    int Lm[N+1];

    if (Li > Ld) return -1;

    Lm[0] = Li;
    Lm[N] = Ld;

    if (valor == A[Li]) return Li;

    for (k=1; k<N; k++)    // Se crean los límites de intervalos
        Lm[k] = Li + k*(Ld-Li)/N;

    for (k=1; k<=N; k++) { // Se cicla por los intervalos
        if ( valor < A[Lm[k]] )
            return BusqNAria(valor, A, N, Lm[k-1]+1, Lm[k] - 1 );
        else if ( valor == A[Lm[k] ] )
            return Lm[k];
    }
    return -1;  // No se encontró
}

int A = { 1, 3, 4, 5, 7, 8, 12, 20, 25, 29, 34, 35, 42, 53, 67, 72};
   /* 16 elementos */

int N   = 4;  /* cantidad de intervalos */
int pos = BusqNAria( 5, A, N, 0, 15);  /* pos == 3 */
N       = 2; /* Búsqueda Binaria */
pos     = BusqNAria( 5, A, N, 0, 34);  /* pos == 10 */
N       = 3; /* Búsqueda Ternaria */
pos     = BusqNAria( 5, A, N, 0, 72);  /* pos == 15 */
```

3.- (7 puntos)

Para el tipo de datos abstracto **Lista** especificado abajo, implemente el algoritmo *Reordenar* para listas, que recibe una lista desordenada y devuelve una lista ordenada. Cada **ListaCDT_S** tiene tres (3) apuntadores: *primer*, *medio*, y *último*. El atributo "*medio*" es un apuntador que apunta al elemento que está a la mitad menos uno (ver gráfico), es decir, que si se pica la lista en dos, medio apuntaría al último de la primera mitad, y su siguiente sería el primero de la segunda lista. Al picar en mitades o entrelazar, hay que ajustar adecuadamente los tres apuntadores y el tamaño de la lista. Si la lista es de longitud impar, la lista posterior tendrá el nodo extra.

```
typedef int Elem;

typedef struct NodoCDT_S* Nodo;

typedef struct NodoCDT_S {
    Elem info;
    Nodo siguiente;
} NodoCDT_S;

Nodo consNodo(Elem i) {
  Nodo N = calloc(1, sizeof(NodoCDT_S));
    /* Todos los atributos inicializados
       a 0 (cero) o a NULL  */
    N->info = i;
    return N;
}
```

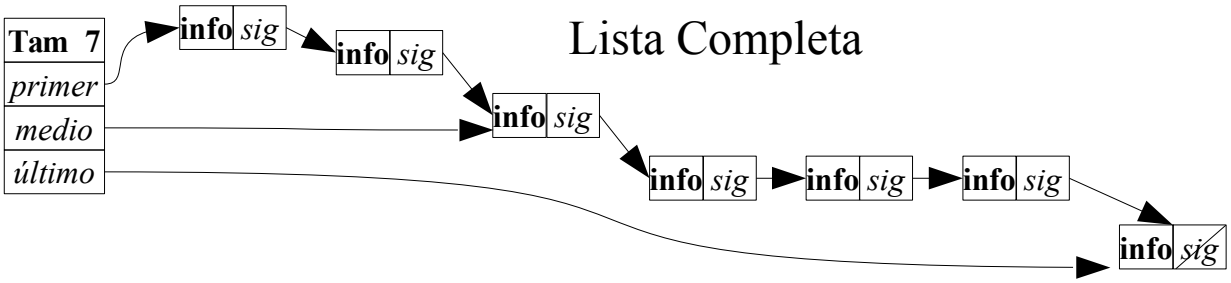
```
typedef struct ListaCDT_S* Lista;

typedef struct ListaCDT_S {
    int  tam;
    Nodo primer; // primer nodo
    Nodo medio;  // nodo "mitad"
    Nodo ultimo; // último nodo
} ListaCDT_S;

     /* Crea una Lista vacía */
Lista consLista () {
    Lista L = calloc(1, sizeof(ListaCDT_S));
    /* Todos los atributos inicializados
       a 0 (cero) o a NULL  */
    return L;
}
```

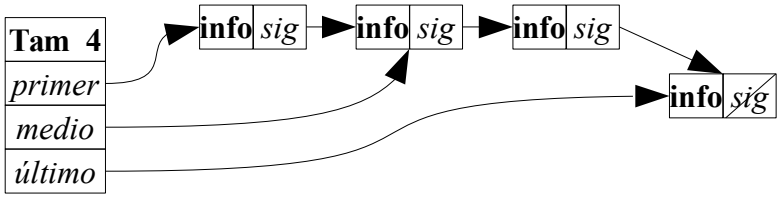
La función "**Lista** *Reordenar*(**Lista** r)" reordena **r** y la devuelve como sigue:

Si **r** es vacía, **r** ya está ordenada
Si **r** no es vacía
    Picar **r** por la mitad en dos listas;
    Reordenar cada mitad de lista por separado;
    Entrelazar las dos mitades de lista en una sola, insertando los nodos por orden ascendente de *info;*

AYUDA:

Picar es un procedimiento ***void***. *Fusionar* es una función que devuelve una lista.
Reordenar para listas **NO** crea ni destruye ***nodos***, sólo los mueve de lista.





```
void sacarPrimer(Lista l, Nodo* r)
{
    /* Suponemos que existe */
}

void ponerUltimo(Lista l, Nodo r)
{
    /* Suponemos que existe */
}

void recalcularMedio(Lista l) {
    int k = 0;
    _pre(l);

    l->medio = l->primer;
    while ((l->medio != NULL) and (k < l->tam/2 - 1)) {
        l->medio = l->medio->siguiente;
        k++;
    }
}

void Picar(Lista l, Lista a, Lista b) {
    _pre(l!=NULL and a!= NULL and b!=NULL);

    a->tam = l->tam/2;
    b->tam = l->tam - a->tam;

    b->primer = l->medio->siguiente;
    b->ultimo = l->ultimo;
    recalcularMedio(b);

    a->primer = l->primer;
    a->ultimo = l->medio;
    a->ultimo->siguiente = NULL;
    recalcularMedio(a);
}

void Entrelazar(Lista a, Lista b, Lista s)
{
    Nodo n;
    _pre(s!=NULL and a!= NULL and b!=NULL);

    while ((a->primer != NULL) and (b->primer != NULL)) {
        if (a->primer->info <= b->primer->info) {
            sacarPrimer(a, &n);
            ponerUltimo(s,  n);
        } else {
            sacarPrimer(b, &n);
            ponerUltimo(s,  n);
        }
    }

    while (a->primer != NULL) {
            sacarPrimer(a, &n);
            ponerUltimo(s,  n);
        }

    while (b->primer != NULL)  {
            sacarPrimer(b, &n);
            ponerUltimo(s,  n);
        }

    recalcularMedio(s);
}
```

```
Lista Reordenar(Lista r) {
    Lista ri, rd, l;

    if (r == NULL)
        return r;

    l  = consLista();
    ri = consLista();
    rd = consLista();

    Picar(r, ri, rd);
    ri = Reordenar(ri);
    rd = Reordenar(rd);
    Entrelazar(ri, rd, l);

    return l;
}

int main ()
{
    Lista L = consLista();
    Lista R = consLista();
    ponerUltimo(L,  consNodo(7));
    ponerUltimo(L,  consNodo(4));
    ponerUltimo(L,  consNodo(9));
    ponerUltimo(L,  consNodo(6));
    ponerUltimo(L,  consNodo(8));
    ponerUltimo(L,  consNodo(1));
    ponerUltimo(L,  consNodo(2));
    ponerUltimo(L,  consNodo(3));

    R = Reordenar(L);
    /* R = 1 --> 2 --> 3 --> 4 --> 6 --> 7 --> 8 --> 9 */
    return 0;
}
```

4.- (4 puntos)

Suponga un arbol con un número variable de hijos. Programe un algoritmo recursivo que imprima los valores *info* de cada nodo del árbol en orden "piso por piso". Ver el ejemplo.

```
typedef int Elem;

typedef struct ArbolCDT_S* Arbol;

typedef struct ArbolCDT_S
{
    Elem info;
    int numHijos;
    Arbol* hijo;
} ArbolCDT_S;
```

```
Arbol consArbol(Elem i, int N, Arbol rama [])
{
    Arbol A = malloc(sizeof(ArbolCDT_S));
    A->info = i;
    A->hijo = malloc(sizeof(N*Arbol));
    int k;
    for (k=0; k<N; k++) {
        A->hijo[k] = rama[k];
    }
    return A;
}
```

AYUDA: Hace falta usar una Cola de apuntadores a estructuras de Arbol como estructura de almacenamiento temporal, que en cada iteración encole los hijos, imprima la información del nodo y siga procesando dicha Cola sacando los primeros elementos hasta que dicha cola quede vacía.

Solución iterativa:

```
void recorrerPorPisos(Cola C)
{
   while (!esVacia(C))
   {
        Arbol a;
        int k;
        desencolar(C, &a);
        printf("%i, ", a->info);
        for (k=0; k<a->numHijos; k++)
            if (a->hijo[k] != NULL)
                encolar(C, a->hijo[k]);
   }
}
```

Solución recursiva:

```
void recorrerPorPisos(Cola C)
{
   if (!esVacia(C))
   {
        Arbol a;
        int k;
        desencolar(C, &a);
        printf("%i, ", a->info);
        for (k=0; k<a->numHijos; k++)
            if (a->hijo[k] != NULL)
                encolar(C, a->hijo[k]);
        recorrerPorPisos( C );
   }
}
. . .

int main() {
    Arbol A = consArbol(1, consArbol(2, NULL, consHoja(4)),
                           consArbol(3, NULL, consArbol(5, NULL,
                                                        consHoja(6))));
    Cola Q = encolar(Q, A);
    recorrerPorPisos(Q);

    /*  Salida   1 2 3 4 5 6  */
```

5.- (3 puntos, solo la parte 5(a))

La función *mapArbol* recibe un árbol binario ***a***, y una función de transformación (*mapa*), y devuelve un árbol binario nuevo al cual se le aplicado la función de transformación a todos los nodos. La funcion *reduceArbol* recibe un árbol binario ***a***, una operación *oper* y un valor inicial *e*, y devuelve el resultado de la operación *oper* acumulada sobre todos los nodos.

```
typedef int Elem;
typedef int Result;

typedef struct ArbolBin_S* Arbol;

typedef struct ArbolBin_S {
    Elem info;
    Arbol hijoIzq;
    Arbol hijoDer;
} ArbolBin_S;
```

```
Arbol consArbolBin(Elem i, Arbol ramaIzq, Arbol ramaDer {
    Arbol a = malloc(sizeof(ArbolBin_S));
    a->info = i;
    a->hijoIzq = ramaIzq;
    a->hijoDer = ramaDer;
    return a;
}

Arbol consHoja(Elem i) {
    return consArbolBin(i, NULL, NULL);
}
```

```
Arbol mapArbol( Arbol a,  Elem (*mapa) (Elem) ) {
    if (a != NULL)
        return consArbolBinario( mapa( a->info ),
                                 mapArbol( a->hijoIzq, mapa ),
                                 mapArbol( a->hijoDer, mapa ) );
    return NULL;
}

Result reduceArbol( Arbol a,  Result (*oper) (Result, Elem),  Result value ) {
    if (a != NULL)
        return oper( reduceArbol( a->hijoIzq,
                                  oper,
                                  reduceArbol( a->hijoDer, oper, value) ),
                     a->info );
    else
        return value;
}
```

```
Elem identidad(Elem n) { return n; }
Elem doble(Elem n)     { return 2*n; }
Elem cuadrado(Elem n)  { return n*n; }

Elem uno(Elem n)       { return 1; }
Elem cero(Elem n)      { return 0; }
```

```
Result suma(Result acum, Elem e) {
    return acum + e;
}

Result producto(Result acum, Elem e) {
    return acum * e;
}
```

De esta manera, para un árbol cualquiera:

`Arbol B = mapArbol(A, cero):`	Crearía un árbol similar a A, con *ceros* en cada nodo
`Arbol C = mapArbol(A, uno):`	Crearía un árbol similar a A, con *unos* en cada nodo
`Arbol D = mapArbol(A, identidad):`	Crearía una copia exacta (clon) del árbol A,
`Arbol E = mapArbol(A, cuadrado):`	Crearía un árbol con la misma forma de A, y en cada nodo la *info* estaría elevada al cuadrado.

`Result sumas = reduceArbol(mapArbol(A, cuadrado), suma, 0):`
Devuelve la *sumatoria* de los *cuadrados* de los valores de los nodos.

`Result prods = reduceArbol(mapArbol(A, doble), producto, 1):`
Devuelve la *productoria* de los dobles de los *valores* de los nodos.

a) Usando solamente las funciones suministradas, como contaría cuantos nodos hay en un árbol.

```
/* Pregunta 5(a) */
Result conteo = reduceArbol( mapArbol( A, uno ), suma, 0 );
printf("Número de nodos: %i\n", conteo);

/* Arbol A:  ( 1 ( 2 (·) 4 ) ( 3 (·) ( 5 8 7 ) ) )
   Número de Nodos: 7                                  */
```

b) y c) Eliminadas

```
/* Pregunta 5(b) */
Arbol mapNodoArbol( Arbol a,  Arbol (*mapa) (Arbol) ) {
    if (a != NULL)
        return mapa( consArbolBinario( a->info ,
                                       mapNodoArbol( a->hijoIzq, mapa ),
                                       mapNodoArbol( a->hijoDer, mapa ) ) );
    return NULL;
}

Result mayor(Result acum, Elem e) {
    return MAX(acum, e);
}

Arbol espejar (Arbol a)
{
    Arbol temp = a->hijoIzq;
    a->hijoIzq = a->hijoDer;
    a->hijoDer = temp;

    return a;
}

Arbol calcAltura(Arbol a)
{
    if (a != NULL) {
        if ( esHoja(a) )
        a->info = 1;
        else if (a->hijoIzq == NULL)
            a->info = 1 + a->hijoDer->info;
        else if (a->hijoDer == NULL)
            a->info = 1 + a->hijoIzq->info;
        else
            a->info = 1 + MAX(a->hijoIzq->info, a->hijoDer->info);
    }
    return a;
}

printf("  Arbol espejo:\n");
Arbol W = mapNodoArbol( A, espejar );

/* Arbol A:  ( 1 ( 2 (·) 4 ) ( 3 (·) ( 5 8 7 ) ) )
   Arbol W:  ( 1 ( 3 ( 5 7 8 ) (·) ) ( 2 4 (·) ) )       */


/* Pregunta 5(c) */
altura = reduceArbol( mapNodoArbol( mapArbol( A, uno), calcAltura ),
                      mayor,
                      0 );
printf("La altura es %i\n", altura);

/* Arbol A:  ( 1 ( 2 (·) 4 ) ( 3 (·) ( 5 8 7 ) ) )
   La altura es 4                                    */
```