ESTRUCTURAS DE DATOS Y ALGORITMOS

CURSO 2009 – PRÁCTICO 4

SOLUCIÓN

Ejercicio 2

```
a) ap1=new apunta;
Incorrecto.
Correcto: ap1 = new int;

b) delete* ap1;
Incorrecto.
Correcto: delete ap1; //solamente si antes se hizo new de ap1

c) ap1=ap2;
Correcto: se copia la direccion que contiene ap2 a ap1

d) ap2=*ap2 + *ap1;
Incorrecto. Se esta asignando un valor a una direccion (puntero)
Correcto: *ap2=*ap2 + *ap1; // asignará la direccion contenida en ap2
// el valor de la suma de los valores contenidos en las direcciones
// donde apuntan ap1 y ap2

e) ap1=NULL;
Correcto: no apunta a nada el puntero ap1.

f) ap4=NULL;
Correcto: idem anterior.

g) delete apuntb;
Incorrecto: no se puede hacer delete de un tipo.

h) ap3=*ap4 * *ap1;
Incorrecto: idem d)

i) ap2=new(ap1);
Incorrecto.
Correcto: ap2 = new int (*ap1);

j) ap2=new int[7];
Correcto: se crea un arreglo de enteros.

k) ap2=new int;
Correcto.

l) ap2=new int(5);
Correcto: inicializa con el valor 5.

m) delete ap3;
Correcto (si antes se hizo new de ap3)
```

```
n) free (ap2);
Correcto (si antes se hizo un malloc de ap2 -no un new)

o) ap1=(apunta)malloc(sizeof(int));
Correcto.

p) ap2=(apunta)malloc(sizeof(apunta));
Incorrecto: compila pero reserva menos memoria de la necesaria para
guardar un int.
```

Ejercicio 3

a) De una representación para la lista simple de naturales (LNat).

```
struct LNat {
            int elem;
            LNat* sig;
};
```

c) Implementar las siguientes operaciones procedurales:

```
void Null(LNat *&l) {
      l=NULL;
}

void Cons(int x, LNat *&l) {
      LNat *aux = new LNat;

      aux->elem=x;
      aux->sig=l;
      l=aux;
}

void IsEmpty(LNat *l, bool &res) {
      if (l==NULL)
            res=true;
      else
            res=false;
}

void Head(LNat *l, int &h) {
      h=l->elem;
}

void Tail(LNat *&l) {
      LNat *aux;
      aux=l;
      l=l->sig;
      aux->sig=NULL;
      delete aux;
}
```

Ejercicio 4

Implementar las siguientes operaciones accediendo directamente a la representación, iterativamente y sin usar procedimientos auxiliares:

```
// Verifica si un natural dado pertenece a la lista
bool IsElement(int x, LNat *l) {
      bool encontre = false;

      while (!encontre && l != NULL) {
            if (l->elem == x)
                  encontre = true;
            l = l->sig;
      }

      return encontre;
}

// Retorna la cantidad de elementos de la lista
int Length(LNat *l) {
      int cant=0;
      LNat *tmp = l;

      while (tmp != NULL) {
            cant++;
            tmp=tmp->sig;
      }
      return cant;
}

// Retorna, si la lista l es no vacía, el último elemento
int Last(LNat *l) {
      int ultimo=-1;
      LNat *tmp=l;
      while (tmp!=NULL) {
            ultimo=tmp->elem;
            tmp=tmp->sig;
      }
      return ultimo;
}

// Retorna, si la lista l es no vacía, su máximo elemento
int Max(LNat *l) {
      int max=-1;
      while (l!=NULL) {
            if (max > l->elem)
                  max=l->elem;
            l=l->sig;
      }
      return max;
}
```

```
// Retorna, si la lista l es no vacía, el promedio de sus elementos
float Average(LNat *l) {
	float prom=0.0;
	int cantelem;


	// sumo y cuento los elementos
	while (l!=NULL) {
		prom=prom + l->elem;
		l=l->sig;
		cantelem++;
	}


	// calculo el promedio
	if (cantelem==0)
		return 0.0;
	else
		return prom/cantelem;
}

// Dada la lista l ordenada, inserta a x en l ordenadamente
LNat* Insert(int x, LNat *l) {

	LNat *nueva, *tmp, *aux;
	bool cambia=false;

	if (l==NULL) {
		nueva=new LNat;
		nueva->elem=x;
		nueva->sig=NULL;
	} else {
		nueva=new LNat;
		aux=nueva;
		while (l!=NULL) {
			// 'cambia' la uso para que cambie solo una vez
			if ((x <= l->elem) && (!cambia)) {
				aux->elem=x;
				cambia=true;
			} else {
				aux->elem=l->elem;
			}

			aux->sig=NULL;
			l=l->sig;

			if (l!=NULL) {
				tmp=new LNat;
				aux->sig=tmp;
				aux=tmp;
			} else {
				aux->sig=NULL;
			}
		}
	}
	return nueva;
}
```

```
// Verifica si la lista p está incluida en la lista l
/* El problema consiste en encontrar una subsecuencia (p) dentro de una
 * secuencia (l). Para realizar esto podemos utilizar un índice para
 * recorrer la secuencia y otro para recorrer la subsecuencia.
 * Cuando se encuentre el primer elemento de la subsecuencia dentro de
 * la secuencia, se debe realizar una búsqueda completa de la
 * subsecuencia, ya que es posible que se encuentre a partir de ese
 * elemento. Para esto se comparará el resto de los elementos de la
 * subsecuencia, con los siguientes elementos de la secuencia.
 * Si efectivamente es encontrada se debe retornar el valor
 * correspondiente. En caso contrario, se detectó una falsa alarma y es
 * necesario continuar la búsqueda del primer elemento de la subsecuencia
 * a partir de la siguiente posición de la secuencia.
 * Este proceso se repite hasta encontrar la subsecuencia o agotar la
 * secuencia.
 */
bool IsIncluded(LNat *l, LNat *p) {
      LNat *auxp, *auxl;
      bool included;

      if (p == NULL) {
            return true;
      } else if (l == NULL) {
            return false;
      } else {
            included = false;
            while ((l != NULL) && (!included)) {
                  auxp = p;
                  auxl = l;
                  while (auxl != NULL && auxp != NULL &&
                         auxl->elem != auxp->elem) {
                        auxl = auxl->sig;
                        auxp = auxp->sig;
                  }
                  /* Si se consume auxp en forma completa es porque la
                   * subsecuencia esta presente.
                   * En caso contrario, se debe continuar desde el
                   * elemento siguiente a la falsa alarma [donde se
                   * encontró el primer elemento de la subsecuencia
                   * en la secuencia].
                   */

                  if (auxp == NULL)
                        included = true;
                  else
                        l = l->sig;
            }

            return included;
      }
}
```

Ejercicio 5

Implementar las siguientes operaciones accediendo directamente a la representación, iterativamente, sin usar procedimientos auxiliares y sin que las soluciones retornadas compartan memoria con los parámetros:

```
// Retorna la lista resultado de tomar los primeros i elementos
LNat *Take(int i, LNat *l) {
	LNat *nueva, *tmp;

	if (l!=NULL) {
		tmp=new LNat;
		nueva=tmp;
	} else
		nueva=NULL;

	int j=0;
	while (j<i && l!=NULL) {
		tmp->elem=l->elem;
		j++;
		if (j<i && l->sig!=NULL) {
			tmp->sig=new LNat;
			tmp=tmp->sig;
			l=l->sig;
		} else {
			tmp->sig=NULL;
		}
	}
	return nueva;
}

// Retorna la lista resultado de no tomar los primeros i-1 elementos
LNat *Drop(int i, LNat *l) {
	LNat *nueva, *tmp;
	int j=0;

	while (j<i && l!=NULL) {
		j++;
		l=l->sig;
	}

	j=0;
	if (l!=NULL) {
		tmp=new LNat;
		nueva=tmp;
	} else
		nueva=NULL;

	while (j<i && l!=NULL) {
		tmp->elem=l->elem;
		j++;
		if (j<i && l->sig!=NULL) {
			tmp->sig=new LNat;
			tmp=tmp->sig;
			l=l->sig;
		} else {
			tmp->sig=NULL;
```

```
            }
        }
        return nueva;
}

// Genera una lista fruto de intercalar ordenadamente las listas l y p,
// que vienen ordenadas

/* La función Merge, dadas dos listas ordenadas l y p, devuelve
 * una tercera lista result que es el resultado de intercalar los
 * elementos de las listas l y p de forma que la lista result también
 * esté ordenada. Este problema, tal como lo señala la letra del
 * ejercicio, debe ser resuelto en forma iterativa.
 *
 * Para construir la lista result es preciso decidir como se intercalan
 * los elementos. Para esto utilizo dos variables que me permiten
 * indicar la próxima posición a considerar en cada una de las listas,
 * las cuales al comienzo indican el principio de cada lista.
 *
 * En cada paso:
 * 1. se comparan los elementos ubicados en las posiciones
 *    indicadas por dichas variables,
 * 2. se toma el menor de los elementos y se lo inserta en la lista
 *    result,
 * 3. se avanza el indicador de posición correspondiente a la lista de
 *    la cual tome el elemento en el paso 2.
 *
 * Si se llega al final de alguna de las dos listas y la otra sigue
 * teniendo elementos es necesario agregar todos esos elementos a la
 * lista result.
 */

LNat *Merge(LNat *l, LNat *p) {
        LNat *result, *iter;


        /* La nueva lista se genera en result, iter es el iterador
         * que recorre esa lista. Como las variables l y p son
         * pasadas por copia las utilizo para recorrer las listas
         * pasadas por parámetro.
         */
        result = NULL;
        while (l != NULL && p != NULL) {
                if (result == NULL) {
                        result = new LNat;
                        iter = result;
                } else {
                        iter->sig = new LNat;
                        iter = iter->sig;
                }

                if (l->elem < p->elem) {
                        iter->elem = l->elem;
                        l = l->sig;
                } else {
                        iter->elem = p->elem;
                        p = p->sig;
```

```
            }
        }

        // copio los elementos restantes de l
        while (l != NULL) {
            if (result == NULL) {
                result = new LNat;
                iter = result;
            } else {
                iter->sig = new LNat;
                iter = iter->sig;
            }

            iter->elem = l->elem;
            l = l->sig;
        }


        // copio los elementos restantes de p
        while (p != NULL) {
            if (result == NULL) {
                result = new LNat;
                iter = result;
            } else {
                iter->sig = new LNat;
                iter = iter->sig;
            }

            iter->elem = p->elem;
            p = p->sig;
        }


        /* si l y p no eran vacias se generó una lista,
         * pero falta indicar el final de la misma,
         * el sig del ultimo es NULL
         */

        if (result != NULL) {
            iter->sig = NULL;
        }

        return result;
}
```

En el código se observa la necesidad el caso de creación del primer nodo de la lista resultado.

```
            if (result == NULL) {
                // creo el primer nodo de result
                result = new LNat;
                iter = result;
            } else {
                // agrego un nuevo nodo a result
                iter->sig = new LNat;
                iter = iter->sig;
            }
```

Existe una estrategia de programación que permite pasar por alto este caso especial. Consiste en considerar que al principio de la lista resultado existe una celda especial (llamada celda "dummy") la cual no almacena ningún elemento. La representación de la lista vacia para una lista con celda "dummy" consiste en una celda con el puntero al siguiente elemento en NULL. Teniendo en cuenta lo antes mencionado pasamos a la implementación en C/C++ obteniendo una primer solución:

```
LNat* Merge(LNat *l, LNat* p) {
      LNat *result, *iter;


      /* La nueva lista se genera en result, iter es el iterador
       * que recorre esa lista. Se genera un nodo dummy en result
       * para facilitar la insercion de elementos con el iterador.
       */

      result = new LNat;
      iter = result;

      while (l != NULL && p != NULL) {
            iter->sig = new LNat;
            iter = iter->sig;

            if (l->elem < p->elem) {
                  iter->elem = l->elem;
                  l = l->sig;
            } else {
                  iter->elem = p->elem;
                  p = p->sig;
            }
      }

      while (l != NULL) {
            iter->sig = new LNat;
            iter = iter->sig;
            iter->elem = l->elem;
            l = l->sig;
      }

      while (p != NULL) {
            iter->sig = new LNat;
            iter = iter->sig;
            iter->elem = p->elem;
            p = p->sig;
      }


      /* hay que hacer que el siguiente del último
       * sea NULL y hay que eliminar la celda dummy
       */

      iter->sig = NULL;
      iter = result;
      result = result->sig;
      delete iter;
      return result;
}
```

Ejercicio 6

Implementar las siguientes operaciones recursivamente sin que las soluciones retornadas compartan memoria con los parámetros:

```
// Genera una lista fruto de intercalar ordenadamente las listas l y p,
// que vienen ordenadas
LNat* Merge(LNat *l, LNat *p) {
	LNat *nueva;

	if (l==NULL && p==NULL)
		return NULL;
	else {
		nueva=new LNat;
		if (l==NULL) {
			nueva->elem=p->elem;
			nueva->sig=Merge(l, p->sig);
		} else if (p==NULL) {
			nueva->elem=l->elem;
			nueva->sig=Merge(l->sig, p);
		} else {
			if (l->elem>=p->elem) {
				nueva->elem=p->elem;
				nueva->sig=Merge(l, p->sig);
			} else {
				nueva->elem=l->elem;
				nueva->sig=Merge(l->sig, p);
			}
		}
		return nueva;
	}
}

//Agrega la lista p al final de la lista l
LNat *Append(LNat *l, LNat *p) {

	LNat *nueva;

	if (l==NULL && p==NULL)
		return NULL;
	else {
		nueva=new LNat;

		if (l==NULL) {//Importante preguntar primero por l
			nueva->elem=p->elem;
			nueva->sig=Append(l, p->sig);
		} else if (p==NULL) {
			nueva->elem=l->elem;
			nueva->sig=Append(l->sig, p);
		} else {
			nueva->elem=l->elem;
			nueva->sig=Append(l->sig, p);
		}
		return nueva;
	}
}
```

Ejercicio 7

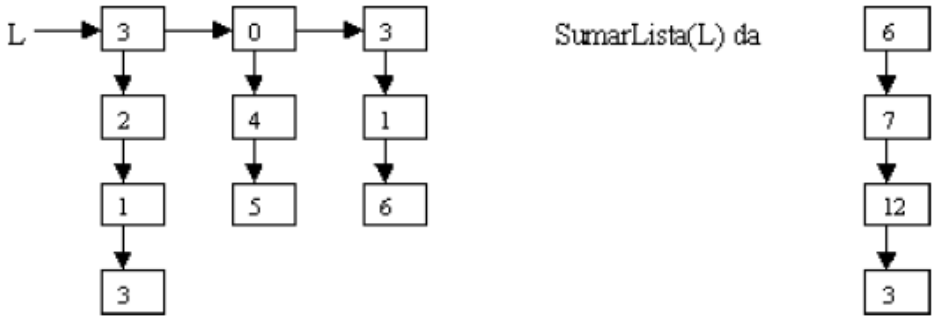
Una variante a la implementación clásica de listas encadenadas es la llamada Lista Doblemente Encadenada. En esa implementación cada elemento de la lista referencia no sólo a su siguiente elemento sino también al anterior. Dar una representación e implementar las siguientes operaciones para Lista Doblemente Encadenada de Naturales:

```
struct LNat {
            int elem;
            LNat* sig;
            LNat* ant;
};

// Crea la lista vacia
//Se podría implementar con celda DUMMY
LNat *Null() {
      return NULL;
}

// Inserta un elemento al principio de la lista
LNat* Cons(int x, LNat *l) {

      LNat *aux = new LNat;

      aux->elem = x;

      if (l==NULL) {
            aux->sig = aux->ant = NULL;
      } else {
            aux->sig = l;
            aux->ant = NULL;
            l->ant = aux;
            l = aux;
      }

      return l;
}

// Verifica si la lista está vacía.
bool IsEmpty(LNat* l) {
      return (l == NULL);
}

// Verifica si un natural pertenece a la lista
bool IsElement(int x, LNat *l) {
      bool encontre = false;

      while (!encontre && l != NULL) {
            if (l->elem == x)
                  encontre = true;
            l = l->sig;
      }
      return encontre;
}
```

Ejercicio 11

1. Introducción

Las llamadas Listas Generales de Naturales son Listas Encadenadas donde cada elemento de la lista es una Lista Encadenada de Naturales. Este ejercicio consiste en implementar una función "**SumarLista**" que dada una Lista General de Naturales devuelva la suma de todos los números de la lista del siguiente modo:



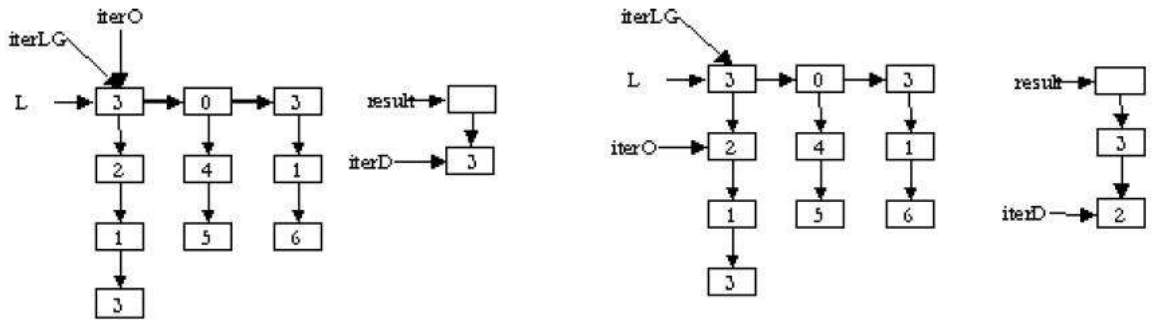
Se sugiere considerar que la Lista General está compuesta por lo menos por una sublista y que no se almacenan sublistas vacías.

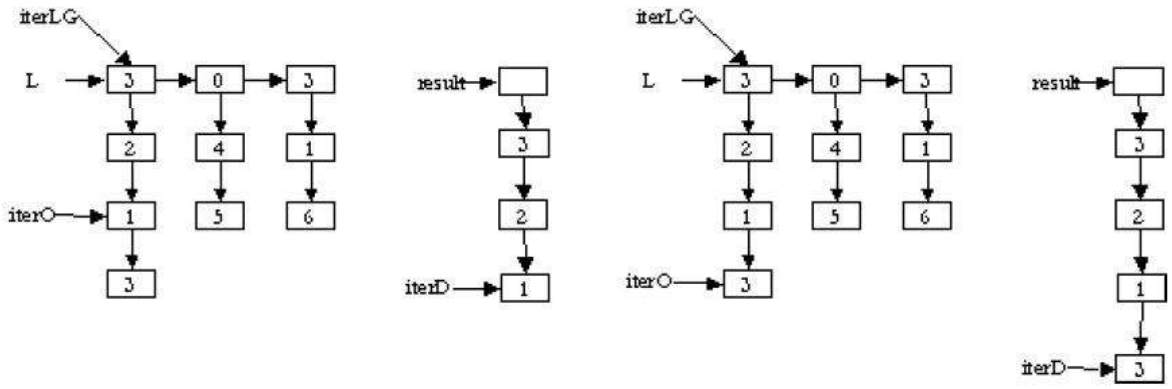
2. Un enfoque iterativo

La solución consiste en recorrer cada una de las listas contenidas en la Lista General. Para cada elemento de cada una de las listas es necesario ver si existe un nodo en la posición correspondiente dentro de la lista resultado que se está generando. Si el nodo no existe, se crea un nodo nuevo, si existe se actualiza el valor sumándole el valor del elemento actual de la lista. Adicionalmente, se utilizará una celda dummy para simplicar las inserciones en la lista resultado.

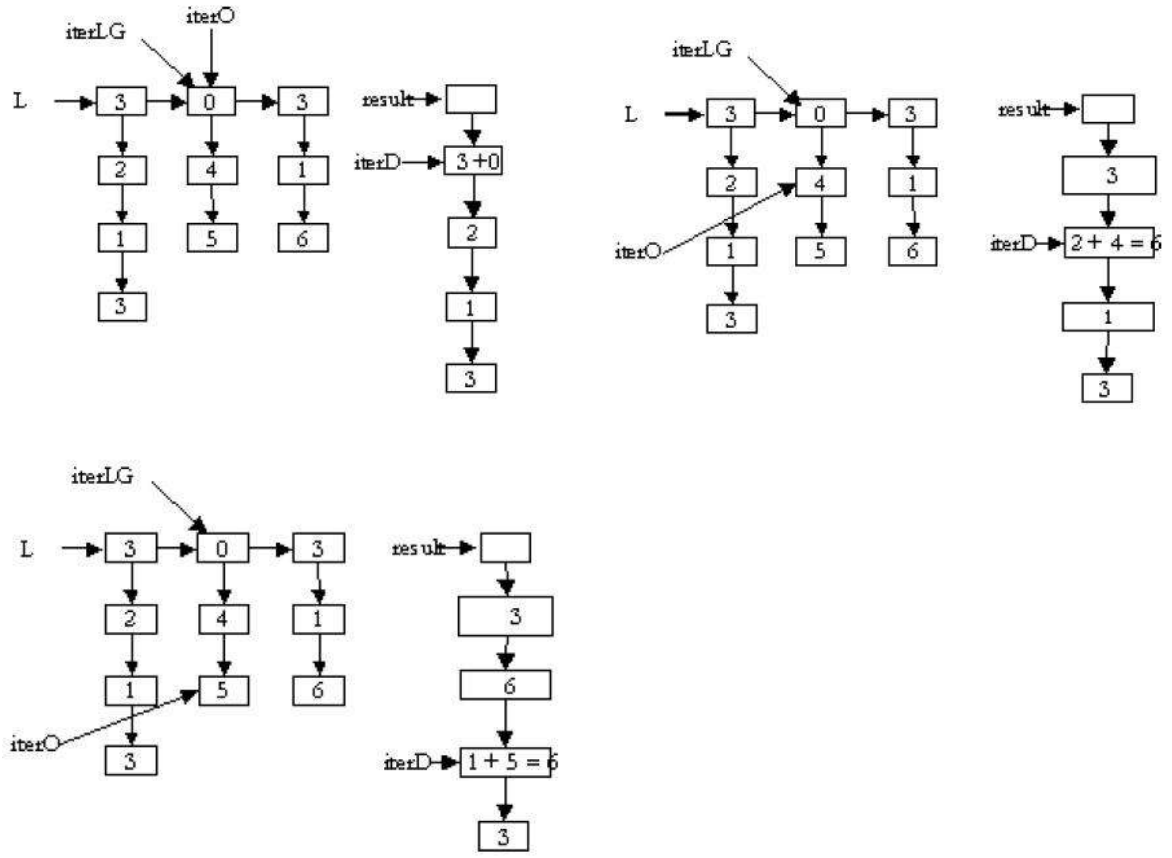
A continuación se presenta un ejemplo al cual nos referiremos para fijar ideas. La idea es recorrer la lista general L con un puntero auxiliar (iterLG) y cada una de las listas de la lista general mediante el puntero iterO (origen). A medida que se itera sobre la lista origen se controla que en la lista destino (iterD) exista un elemento correspondiente. Si efectivamente existe, se actualiza su valor sumándole el valor del nodo actual de la lista origen, si no existe se crea un nuevo nodo que contiene el valor de la lista origen.

Primera iteración:





Segunda iteración:



3. Implementación del enfoque iterativo

A continuación presentamos la implementación iterativa de la solución propuesta accediendo directamente a la representación tanto de la Lista de Naturales como de la Lista General de Naturales.

```
typedef struct LNat {
      int valor;
      LNat *sig;
};
```

```
typedef struct LGNat {
      LNat* elem;
      LGNat* sig;
};
```

```
// Devuelve la suma de los elementos de una lista general
// de naturales NO vacía
LNat* SumaLista(LGNat* l) {
      LGNat *iterLG;
      LNat *result, *iterD, *iterO;

      // result apunta al comienzo de la lista a devolver
      result = new LNat;
      result->sig = NULL;
      iterD = result;
      iterLG = l;

      // iterLG itera sobre la lista general
      while (iterLG != NULL) {
            // iterO recorre cada una de las listas de la lista general
            iterO = iterLG->elem;

            while (iterO != NULL) {
                  if (iterD->sig == NULL) {
                        // si no existe el nodo se crea y el sig es NULL

                        iterD->sig = new LNat;
                        iterD = iterD->sig;
                        iterD->valor = iterO->valor;
                        iterD->sig = NULL;
                  } else {
                        // si ya existe el nodo se actualiza el valor
                        // como el valor que posee mas el elemento de la
                        // lista origen

                        iterD->sig->valor = iterD->sig->valor +
                                            iterO->valor;
                        iterD = iterD->sig;
                  }

                  // se avanza el puntero sobre la lista origen
                  iterO = iterO->sig;
            } // end while de iteracion sobre cada lista

            iterLG = iterLG->sig;
            iterD = result;
      } // end while de iteracion sobre la lista general

      iterD = result;
      result = result->sig;
      delete iterD;
      return result;
}
```

4. Un enfoque recursivo

A continuación presentamos una solución recursiva para este problema.

```
SumarLista: LGNat -> LNat

SumarLista(x.[]) = x
SumarLista(x.Xs) = b
                   siendo b = Suma (x , SumarLista(Xs))


Suma: LNat * LNat -> LNat

Suma([],[]) = []
Suma(Xs,[]) = Xs
Suma([],Ys) = Ys
Suma(x.Xs, y.Ys) = (x+y).Suma(Xs,Ys)
```

Esta solución se basa en el uso de la función auxiliar Suma, que devuelve una Lista de Naturales donde cada nodo de la lista es la suma de los nodos de igual nivel. La lista resultado tiene igual largo que la lista más larga pasada como argumento a la función.

5. Implementación del enfoque recursivo

A continuación presentamos la implementación recursiva de la solución propuesta accediendo directamente a la representación tanto de la Lista de Naturales como de la Lista General de Naturales.

```
// Modifica la lista result, sumando los elementos de la
// lista l, donde cada elemento de la lista resultado es
// la suma de los correspondientes.
void Sumar(LNat* l, LNat *&result) {
      if (l != NULL) {
            if (result != NULL) {
                  result->valor = result->valor + l->valor;
                  Sumar(l->sig, result->sig);
            } else {
                  result = new LNat;
                  result->valor = l->valor;
                  result->sig = NULL;
                  Sumar(l->sig, result->sig);
            }
      }
}

// Devuelve la suma de los elementos de una lista general
// de naturales NO vacía
void SumaLista(LGNat* lg, LNat *&result) {
      if (lg == NULL) {
            result = NULL;
      } else {
            SumaLista(lg->sig, result);
            Sumar(lg->elem, result);
      }
}
```