Solución Práctico Listas

- ***Ejercicio 1***

Dada la siguiente definición de Lista:

```
typedef struct nodo_lista * lista;

struct nodo_lista{
    int dato;
    lista sig;
}
```

Implemente ***iterativamente*** las siguientes operaciones accediendo directamente a la representación y sin usar procedimientos auxiliares *(Si considera necesario, puede pasar la lista por referencia agregando &).*

```
bool IsElement(int x, lista l){
// Retorna true si x pertenece a l, false en caso contrario.
    while ((l != NULL) && (l->dato != n))
          l = l->sig;
    if (l == NULL)
          return false;
    else
          return true;
}

int Length(lista l){
// Retorna la cantidad de elementos de la lista.
    int cant = 0;
    while (l != NULL){
          cant++;
          l = l->sig;
    }
    return cant;
}

int Last(lista l){
// Retorna el último elemento de l.
// Pre: l no es vacía.
    while(l->sig != NULL)
          l = l->sig;
    return l->dato;
}

int Max(lista l);
// Retorna el máximo elemento de l.
// Pre: l no es vacía.
    int max = l->dato;
    while (l->sig != NULL){
          if (max < l->dato)
                max = l->dato:
          l = l->sig;
    }
    return max;
}

float Average(lista l){
// Retorna si la lista no es vacía el promedio de sus elementos.
// Pre: l no es vacía.
    int cant = 0;
    int total = 0;
    while (l != NULL){
          cant++;
          total += l->dato;
          l = l->sig;
    }
    return (float)total/cant;
}
```

```
lista Insert(int x, lista l){
// Inserta ordenadamente el elemento x en la lista ordenada l.
   lista actual, ant;
   actual = l;
   ant = NULL;
   while ((actual != NULL) && (actual->dato > n)){
          ant = actual;
          actual = actual->sig;
   }
   lista aux = new(nodo_lista);
   aux->dato = x;
   aux->sig = actual;
   if (ant != NULL)
          ant->sig = aux;
   else
          l = aux;
   return l;
}

lista Snoc(int x, lista l){
// Inserta el elemento x al final de la lista l.
   lista aux = new(nodo_lista);
   aux->sig = NULL;
   aux->dato = x;
   if (l == NULL)
          return aux;
   else{
          lista inicio = l;
          while (l->sig != NULL)
                 l = l->sig;
          l->sig = aux;
          return inicio;
   }
}

lista Remove(int x, lista l){
// Elimina todas las ocurrencias de x en la lista l.
   lista actual = l;
   lista ant = NULL;
   while (actual != NULL){
          if (actual->dato == x){   // Tengo que eliminar
                 if (ant == NULL){       // Soy el primero
                        l = l->sig;
                        delete actual;
                        actual = l;
                 }else{                  // no soy el primero
                        ant->sig = actual->sig;
                        delete actual;
                        actual = ant->sig;
                 }
          }else{
                 ant = actual;
                 actual = actual->sig;
          }
   }
   return l;
}

bool Equals(lista l, lista p){
// Verifica si las listas l y p son iguales (mismos elementos en el mismo orden).
   while ((l != NULL) && (p != NULL) && (l->dato == p->dato)){
          l = l->sig;
          p = p->sig;
   }
   if ((l == NULL) && (p == NULL))
          return true;
   else
          return false;
}
```

- ***Ejercicio 2***

Implemente ***iterativamente*** las siguientes operaciones accediendo directamente a la representación y sin usar procedimientos auxiliares y sin que las soluciones retornadas compartan memoria con los parámetros.

```
lista Take(int i, lista l){
// Retorna la lista resultado de tomar los primeros i elementos.
// l no comparte memoria con la lista resultado.
   lista ret = NULL;
   lista fin = NULL;
   while ((i> 0) && l != NULL){
          if (fin == NULL){ // Soy el primero
                ret = new(nodo_lista);
                ret->dato = l->dato;
                fin = ret;
          }else{
                fin->sig = new(nodo_lista);
                fin = fin->sig;
                fin->dato = l->dato;
          }
          i--;
          l = l->sig;
   }
   if (fin != NULL)
          fin->sig = NULL;
   return ret;
}

lista Drop(int u, lista l){
// Retorna la lista resultado de no tomar los primeros u elementos.
// l no comparte memoria con la lista resultado.
   lista ret = NULL;
   lista fin = NULL;
   while (l != NULL){
          if (u <= 0){
                if (fin == NULL){ // Soy lo primero que copio
                      ret = new(nodo_lista);
                      ret->dato = l->dato;
                      ret->sig = NULL;
                      fin = ret;
                }else{
                      fin->sig = new(nodo_lista);
                      fin = fin->sig;
                      fin->sig = NULL;
                      fin->dato = l->dato;
                }
          }
          l = l->sig;
          u--;
   }
   return ret;
}

lista Merge(lista l, lista p){
// Genera una lista fruto de intercalar ordenadamente las listas.
// l y p que vienen ordenadas.
// l y p no comparten memoria con la lista resultado.
   lista m = NULL;
   lista u = NULL;  // ultimo
   while ((l != NULL) || (p != NULL)){
          if (m == NULL){
                m = new(nodo_lista);
                m->sig = NULL;
                u = m;
          }else{
                u->sig = new(nodo_lista);
                u = u->sig;
                u->sig = NULL;
          }
          if ((l == NULL) || ((p != NULL) && (l->dato > p->dato))){
                u->dato = p->dato;
                p = p->sig;
```

```
            }else{
                  u->dato = l->dato;
                  l = l->sig;
            }
      }
      return m;
}

lista Append(lista l, lista p){
// Agrega la lista p al final de la lista l.
// l y p no comparten memoria con la lista resultado.
      lista a = NULL;
      lista fin = NULL;
      while ((l != NULL) || (p != NULL)){
            if (a == NULL){
                  a = new(nodo_lista);
                  a->sig = NULL;
                  fin = a;
            }else{
                  f->sig = new(nodo_lista);
                  f = f->sig;
                  f->sig = NULL;
            }
            if (l != NULL){
                  f->dato = l->dato;
                  l = l->sig;
            }else{
                  f->dato = p->dato;
                  p = p->sig;
            }
      }
      return a;
}
```

- ***Ejercicio 3***

Implemente ***recursivamente*** las siguientes operaciones sin que las soluciones retornadas compartan memoria con los parámetros.

```
lista Take(int i, lista l){
// Retorna la lista resultado de tomar los primeros i elementos.
// l no comparte memoria con la lista resultado.
      if (i == 0)
            return NULL;
      else if (l == NULL)
            return NULL;
      else{
            lista aux = new(nodo_lista);
            aux->sig = Take(i-1, Tail(l));
            aux->dato = l->dato;
            return aux;
      }
}

lista Drop(int u, lista l){
// Retorna la lista resultado de no tomar los primeros u elementos.
// l no comparte memoria con la lista resultado.
      if (l == NULL)
            return NULL;
      else if(u == 0){
            lista aux = new(nodo_lista);
            aux->dato = l->dato;
            aux->sig = Drop(0, l->sig);
            return aux;
      }else
            return Drop(u-1, l->sig);
}
```

```
lista Merge(lista l, lista p);
// Genera una lista fruto de intercalar ordenadamente las listas
// l y p que vienen ordenadas.
// l y p no comparten memoria con la lista resultado.
   if ((l == NULL) && (p == NULL)){
          return NULL;
   else if (l == NULL){
          lista aux = new(nodo_lista);
          aux->sig = Merge(l, p->sig);
          aux->dato = p->dato;
          return aux;
   }elseif (p == NULL){
          lista aux = new(nodo_lista);
          aux->sig = Merge(l->sig, p);
          aux->dato = l->dato;
          return aux;
   }else{ // l != NULL && p != NULL
          lista aux = new(nodo_lista);
          if (l->dato < p->dato){
                 aux->sig = Merge(l->sig, p);
                 aux->dato = l->dato;
          }else{
                 aux->sig = Merge(l, p->sig);
                 aux->dato = p->dato;
          }
          return aux;
   }
}

lista Append(lista l, lista p){
// Agrega la lista p al final de la lista l.
// l y p no comparten memoria con la lista resultado.
   if ((l == NULL) && (p == NULL)){
          return NULL;
   }else{
          lista aux = new (nodo_lista);
          if (l != NULL){
                 aux->sig = Append (l->sig, p);
                 aux->dato = l->dato;
                 return aux;
          }else{
                 aux->sig = Append (NULL, p->sig);
                 aux->dato = p->dato;
                 return aux;
          }
   }
}
```

- ***Ejercicio 4***

Dada la siguiente definición del TAD Lista:

```
lista Null();
// Crea la lista vacía.

lista Cons(int x, lista l);
// Inserta el elemento x al principio de la lista l.

bool IsEmpty(lista l);
// Retorna true si l es vacía, false en caso contrario.

int Head(lista l);
// Retorna el primer elemento de la lista.
// Pre: l no vacía.

lista Tail(lista l);
// Retorna la lista sin su primer elemento.
// Pre: l no vacía.
```

Implemente las siguientes operaciones ***recursivamente*** utilizando exclusivamente las operaciones anteriores *(sin acceder a la representación interna)*:

```
bool IsElement(int x, lista l){
// Retorna true si x pertenece a l, false en caso contrario.
   if (IsEmpty(l))
         return false;
   else if (Head(l) == x)
         return true;
   else
         return IsElement(x, Tail(l));
}

lista Remove(int x, lista l){
// Retorna la lista fruto de eliminar x en l.
// l no comparte memoria con la lista resultado.
   if (IsEmpty(l))
         return NULL;
   else if (Head(l) == x)
         return Remove(x, Tail(l));
   else
         return Cons(Head(l), Remove(x, Tail(l)));
}

int Length(lista l){
// Retorna la cantidad de elementos de la lista.
   if (IsEmpty(l))
         return 0;
   else
         return 1 + Length(Tail(l));
}

lista Snoc(int x, lista l){
// Retorna la lista fruto de insertar el elemento x al final de la lista l.
// l no comparte memoria con la lista resultado.
   if (IsEmpty(l))
         return Cons(x, Null());
   else
         return Cons(Head(l), Snoc(x, Tail(l)));
}

lista Append(lista l, lista p){
// Retorna la lista fruto de agregar la lista p al final de la lista l.
// l no comparte memoria con la lista resultado.
   if (!IsEmpty(l))
         return Cons(Head(l), Append(Tail(l), p));
   else if (!IsEmpty(p))
         return Cons(Head(p), Append(Null(), Tail(p)));
   else
         return Null();
}

lista Insert(int x, lista l){
// Retorna la lista fruto de insertar ordenadamente el elemento x en la lista ordenada l.
// l no comparte memoria con la lista resultado.
   if (IsEmpty(l))
         return Cons(x, Null());
   else if (Head(l) < x)
         return Cons(Head(l), Insert(x, Tail(l)));
   else if (IsEmpty(Tail(l)))
         return Cons(x, Cons(Head(l), Null()));
   else
         return Cons(x, Cons(Head(l), Insert(Head(Tail(l)), Tail(l))));
                                        // Para que siga copiando en "if (Head(l) < x)"
}

int Last(lista l){
// Retorna el último elemento.
// Pre: l no vacía.
   if (IsEmpty(Tail(l)))
         return Head(l);
   else
         return Last(Tail(l));
}
```

```
int HowMany(int x, lista l){
// Cuenta las ocurrencias del natural x en la lista l
   if (IsEmpty(l))
         return 0;
   else if (Head(l) == x)
         return 1 + HowMany(x, Tail(l));
   else
         return 0 + HowMany(x, Tail(l));
}

int Max(lista l){
// Retorna el máximo elemento de l.
// Pre: l no vacía.
   if (IsEmpty(Tail(l)))
         return Head(l);
   else{
         int max_tail = Max(Tail(l));
         if (Head(l) > max_tail)
               return Head(l);
         else
               return max_tail;
   }
}

bool IsSorted(lista l){
// Retorna true si l está ordenada de menor a mayor, false en caso contrario.
   if (IsEmpty(l))
         return true;
   else if IsEmpty(Tail(l))
         return true;
   else if (Head(l) < Head(Tail(l)))
         return IsSorted(Tail(l));
   else
         return false;
}

lista Change(int x, int y, lista l){
// Retorna una nueva lista fruto de cambiar x por y en l.
// l no comparte memoria con la lista resultado.
   if (IsEmpty(l))
         return Null();
   else if (Head(l) == x)
         return Cons(y, Change(x, y, Tail(l)));
   else
         return Cons(Head(l), Change(x, y, Tail(l)));
}

lista InsBefore(int x, int y, lista l){
// Retorna una nueva lista fruto de insertar x antes de y en l.
// l no comparte memoria con la lista resultado
   if (IsEmpty(l))
         return Null();
   else if (y == Head(l))
         return Cons(x, Cons(y, InsBefore(x, y, Tail(l))));
   else
         return Cons(Head(l), InsBefore(x, y, Tail(l)));
}

lista InsAround(int x, int y, lista l){
// Retorna una nueva lista fruto de insertar x antes y después de y en l.
// l no comparte memoria con la lista resultado.
   if (IsEmpty(l))
         return Null();
   else if (y == Head(l))
         return Cons(x, Cons(y, Cons(x, InsBefore(x, y, Tail(l)))));
   else
         return Cons(Head(l), InsBefore(x, y, Tail(l)));
}
bool Equals(lista l, lista p){
```

```
   // Retorna true si las listas l y p son iguales (mismos elementos en el mismo orden)
   // false en caso contrario.
      if (IsEmpty(l) && IsEmpty(p))
            return true;
      else if (IsEmpty(l))
            return false;
      else if (IsEmpty(p)
            return false;
      else if (Head(l) == Head(p))
``          return Equals(Tail(l), Tail(p));
      else
            return false;
   }

   void Show(lista l){
   // Muestra los elementos de la lista l separados por comas.
      if(!IsEmpty(l)){
            cout << Head(l);
            if (!IsEmpty (Tail (l))){
                  cout << ", ";
                  Show(Tail(l));
            }
      }
   }
```

- ***Ejercicio 5***

Las llamadas listas generales de naturales son listas encadenadas donde cada elemento de la lista es una lista encadenada de naturales.

```
typedef nodo_listagral * listagral;

struct nodo_listagral{
   lista dato;
   listagral sig;
}
```

```
typedef struct nodo_lista * lista;

struct nodo_lista{
      int dato;
      lista sig;
}
```

Implemente las siguientes operaciones manipulando la representación interna:

```
   listagral NullLG(){
   // Crea la lista general vacía.
      return NULL;
   }

   listagral ConsLG(listagral lg, lista l){
   // Inserta la lista elemento l al principio de la lista general lg.
      listagral aux = new(nodo_listagral);
      aux->sig = lg;
      aux->dato = l;
      return aux;
   }

   bool IsEmptyLG(listagral lg){
   // Verifica si la lista general está vacía.
      return (lg == NULL);
   }

   lista HeadLG(listagral lg){
   // Retorna la primer lista elemento.
   // Pre: lg no vacía.
      return lg->dato;
   }

   listagral TailLG(listagral lg){
   // Retorna lg sin su primer elemento.
   // Pre: lg no vacía.
      return lg->sig;
   }
```

Implementar ***recursivamente*** las siguientes operaciones utilizando las operaciones anteriores y las del Ejercicio 4:

```
int LengthLG(listagral lg){
// Retorna la cantidad de naturales de la lista general lg.
   if (IsEmptyLG(lg))
          return 0;
   else
          return Length(HeadLG(lg)) + LengthLG(TailLG(lg));
}

void ShowLG(listagral lg){
// Muestra la lista general separando los naturales y listas de naturales por comas y
// encerrando cada lista de naturales entre paréntesis.
   if(!IsEmptyLG (lg)){
          cout << "(";
          Show(HeadLG(lg));
          cout << ")";
          if (!IsEmptyLG (TailLG (lg))){
                 cout << ", ";
                 ShowLG(TailLG(lg));
          }else
                 cout << endl;
   }
}
```