Tema: Listas Simples

Apellido y Nombre: ..... Fecha:...../...../.....

EJEMPLOS

Ejemplo 1 – Variante de implementación: Modifique la implementación básica de listas simples de enteros (con un único puntero al inicio) de modo que se incluya un elemento para registrar la cantidad de nodos almacenados. Además desarrolle las operaciones *iniciar_lista*, *agregar_inicio*, *quitar_final* y *mostrar_lista* adaptadas a la implementación propuesta. Tenga en cuenta que la operación *agregar_inicio* sólo añadirá un nuevo nodo si éste contiene un valor menor a los ya almacenados; en tanto que la operación *mostrar_lista* debe implementarse mediante un algoritmo *recursivo*.

Implementación modificada

Para realizar la implementación solicitada no es necesario modificar la definición del nodo de la lista simple sino añadir un registro que contenga el puntero al inicio de la lista y el contador de nodos. Un error común es incluir el contador en el nodo, lo que implicaría actualizar el contador en todos los nodos cada vez que se agregue o quite un elemento.

La definición de lista simple correspondiente a esta implementación es la siguiente:

```
TIPOS
pnodo=puntero a tnodo
tnodo=REGISTRO
        dato:ENTERO
        sig:pnodo
      FIN_REGISTRO
tlista=REGISTRO
        inicio:pnodo
        contador:ENTERO
      FIN_REGISTRO
```

```
typedef struct tnodo *pnodo;
typedef struct tnodo {
                       int dato;
                       pnodo sig;
                      };
typedef struct tlista {
                       pnodo inicio;
                       int contador;
                      };
```

La operación *iniciar_lista* se realiza mediante un procedimiento que inicializa la lista. La inicialización crea una lista vacía asignando a *inicio* y *contador* los valores adecuados. En este caso, al campo *inicio* se le asigna NULO (NULL en C/C++), mientras que al campo *contador* se le asigna cero.

```
PROCEDIMIMENTO iniciar_lista(E/S num:tlista)
INICIO
   num.inicio←NULO
   num.contador←0
FIN
```

```
void iniciar_lista(tlista &num)
{
  num.inicio=NULL;
  num.contador=0;
}
```

La operación *agregar_inicio* se realiza mediante un procedimiento que añade un nuevo nodo al inicio de la lista siempre que éste contenga un valor menor a los ya almacenados. Para ello se consideran 2 casos: una lista vacía y una lista con elementos. Al agregar un nodo debe actualizarse el contador, mientras que los nodos no agregados deben ser liberados.

```
PROCEDIMIMENTO agregar_inicio(E/S num:tlista,
                              E nuevo:pnodo)
INICIO
   SI num.contador=0 ENTONCES
      num.inicio←nuevo
      num.contador←num.contador+1
   SINO
     SI num.inicio->dato > nuevo->dato ENTONCES
        nuevo->sig←num.inicio
        num.inicio←nuevo
        num.contador←num.contador+1
     SINO
        liberar(nuevo)
     FIN_SI
   FIN_SI
FIN
```

```
void agregar_inicio(tlista &num, pnodo nuevo)
{ if (num.contador==0)
   { num.inicio=nuevo;
     num.contador++; }
  else
   { if (num.inicio->dato > nuevo->dato)
      { nuevo->sig=num.inicio;
        num.inicio=nuevo;
        num.contador++;
      }
     else
       delete(nuevo);
   }
}
```

La operación *quitar_final* se realiza mediante una función que retorna la dirección de un nodo extraído del final de la lista. Para ello se consideran 3 casos: una lista vacía, una lista con un único elemento y una lista con 2 o más elementos. Si la lista está vacía la función retorna el valor NULO (NULL en C/C++) mientras que si ésta contiene elementos entonces se retorna la dirección del nodo extraído. En particular, si la lista contiene sólo un elemento la operación de extracción generará una lista vacía.

```
FUNCIÓN quitar_final(E/S num:tlista):pnodo
VARIABLES
   i,aux:pnodo
INICIO
   SI num.contador=0 ENTONCES
      aux←NULO
   SINO
     SI num.contador=1 ENTONCES
        aux←num.inicio
        num.inicio←NULO
        num.contador←0
     SINO
        i←num.inicio
        MIENTRAS (i->sig)->sig <> NULO HACER
          i ← i->sig
        FIN_MIENTRAS
        aux←i->sig
        i->sig←NULO
        num.contador←num.contador-1
     FIN_SI
   FIN_SI
   quitar_final←aux
FIN
```

```
pnodo quitar_final(tlista &num)
{ pnodo i,aux;
  if (num.contador==0)
   aux=NULL;
  else
   { if (num.contador==1)
     { aux=num.inicio;
       num.inicio=NULL;
       num.contador=0;
     }
     else

     {for(i=num.inicio;(i->sig)->sig!=NULL;i=i->sig);
       aux=i->sig;
       i->sig=NULL;
       num.contador--;
     }
   }
   return aux;
}
```

Finalmente, la operación *mostrar_lista* se realiza mediante un procedimiento recursivo que muestra el contenido de la lista. Para ello se define el caso base y la regla de descomposición del problema. Por un lado, el caso base contempla 2 posibles situaciones: una lista vacía y una lista con un elemento. Por otro lado, la regla recursiva permite desplazarse sobre la lista reduciendo, en cada llamado, la cantidad de nodos considerados. Obsérvese que el algoritmo utiliza como parámetro un dato de tipo *pnodo* y no *tlista*, esto permite que en cada llamado se trabaje con el mismo tipo de problema (un puntero pnodo). Así, el llamador original al procedimiento podría ser *mostrar_lista(**lista.inicio**)*.

```
PROCEDIMIENTO mostrar_lista(E n:pnodo)
INICIO
   SI n=NULO ENTONCES
     ESCRIBIR "Lista Vacía"
   SINO
     SI n->sig=NULO ENTONCES
        ESCRIBIR n->dato
     SINO
        ESCRIBIR n->dato
        mostrar_lista(n->sig)
     FIN_SI
   FIN_SI
FIN
```

```
void mostrar_lista(pnodo n)
{
  if (n==NULL)
    cout << "Lista Vacia" << endl;
  else
    if (n->sig==NULL)
      cout << n->dato << endl;
    else
     { cout << n->dato << endl;
       mostrar_lista(n->sig);
     }
}
```

EJERCICIOS

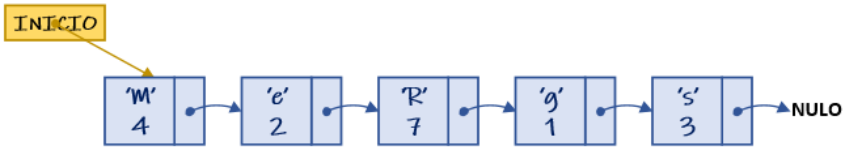
1) En base a la definición de *Lista Simple* (de caracteres), y considerando un único puntero al *inicio* de la lista, implemente sus operaciones fundamentales teniendo en cuenta lo siguiente:
   - Una lista requiere de elementos, llamados nodos, que almacenen datos y que poseen un indicador del próximo elemento de la lista.
   - Una operación de inicialización que permita crear (inicializar) una lista vacía.
   - Una operación que permita crear nodos.
   - Una operación de inserción que permita agregar un nuevo nodo al inicio de la lista.

- Una operación de inserción que permita agregar un nuevo nodo al final de la lista.
- Una operación de inserción que permita agregar, en orden, un nuevo nodo a la lista.
- Una operación que extraiga un nodo del inicio de la lista.
- Una operación que extraiga un nodo del final de la lista.
- Una operación que extraiga un nodo específico (según un valor ingresado por el usuario) de la lista.
- Una operación que permita buscar un nodo (valor) en la lista.
- Una operación que permita mostrar el contenido de la lista.

2) Considerando una lista de enteros, con un único puntero de inicio, modifique la definición y operaciones básicas de listas de modo que se registre la cantidad de valores primos, cantidad de valores Fibonacci y cantidad total de valores guardados. Desarrolle las operaciones *iniciar_lista*, *crear_nodo*, *agregar_inicio*, *quitar_final* y *mostrar_lista*.

3) Dada una lista de caracteres, con un único puntero de inicio, realice lo siguiente:
   a) Consigne la declaración de tipos y variables de la estructura, considerando que se limita la capacidad de la lista a 15 elementos.
   b) Diseñe un procedimiento/función que permita *agregar* valores al final de la lista. Tenga en cuenta que los nodos no agregados deben ser liberados.
   c) Diseñe un procedimiento/función que determine si la lista contiene únicamente minúsculas.
   d) Diseñe un procedimiento/función **recursivo** que muestre el contenido de la lista.

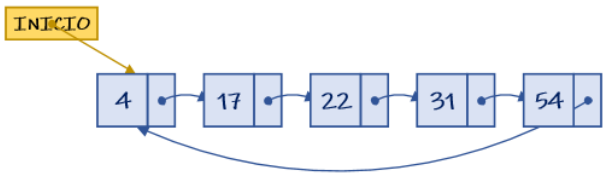
4) Dada la siguiente lista de caracteres, con un único puntero de inicio, realice lo siguiente:



   a) Consigne la declaración de tipos y variables de la estructura. Considere que, por cada carácter, se crea un único nodo que registra las veces que intentó almacenarse un dato determinado.
   b) Diseñe un procedimiento/función que permita *agregar* un nuevo elemento *al inicio* de la lista o actualizar la cantidad de apariciones si el dato ya fue almacenado. Los nodos no agregados deben ser liberados.
   c) Diseñe un procedimiento/función que permita *quitar* un valor específico de la lista. Téngase en cuenta que si el carácter se agregó 2 o más veces sólo debe actualizarse la cantidad de apariciones.
   d) Diseñe un procedimiento/función **recursivo** que muestre el contenido de la lista de *inicio* a *final* o viceversa según un parámetro de opción.

5) Dada una lista de valores reales, con un único puntero de inicio, realice lo siguiente:
   a) Consigne la declaración de tipos y variables de la estructura considerando que debe registrarse la cantidad de nodos almacenados.
   b) Diseñe un procedimiento/función que permita agregar valores a la lista por el *inicio* o *final* según un parámetro de opción.
   c) Diseñe un procedimiento/función que ordene, de forma creciente, el contenido de la lista. Considere que sólo se mueven (copian) los datos, sin modificar la posición de los nodos. Aplique el algoritmo de ordenación *Burbuja*.
   d) Diseñe un procedimiento/función que libere todos los nodos de la lista.

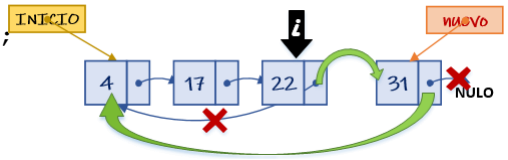
6) Dada la siguiente lista **CIRCULAR**:



a) defina las estructuras de datos que permitan representarla.

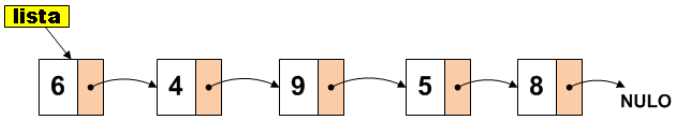
b) Considerando que en las listas circulares el último elemento se conecta al primero, ¿cómo se modifican las operaciones básicas *iniciar_lista*, *agregar_inicio*, *quitar_final* y *mostrar_lista*? Tome como referencia el siguiente ejemplo.

```
void agregar_final(pnodo &lista, pnodo nuevo)
{ pnodo i;
  if (lista==NULL)
   { lista=nuevo;
     nuevo->sig=lista;
   }
   else
   { for(i=lista;i->sig!=lista;i=i->sig);
     i->sig=nuevo;
     nuevo->sig=lista;
   }
}
```



7) Dada una lista CIRCULAR de caracteres, y suponiendo que sólo podrá almacenar hasta 10 elementos, modifique la definición básica de listas e implemente las operaciones *iniciar_lista*, *agregar_orden* (decreciente), *quitar_nodo* y *mostrar_lista* adaptadas a la implementación circular. Considere que los nodos no agregados deben ser liberados.

8) Analice los siguientes fragmentos de código, describa las acciones que realizan y determine sus propósitos. Utilice la lista que se muestra a continuación para comprobar los algoritmos, no olvide los casos esenciales:




```
int enigma (pnodo lista)
{
  if (lista==NULL)
   return 0;
  else
   return enigma(lista->sig)+lista->dato;
}
```

```
int incognita (pnodo lista)
{
  if (lista==NULL)
    return 0;
  else
    incognita(lista->sig)+1;
}
```

```
pnodo misterio (pnodo lista, int m)
{
   if (lista==NULL)
     return NULL;
   else
    { if (lista->dato==m)
        return lista;
      else
       return misterio(lista->sig,m);
    }
}
```

```
bool secreto (pnodo lista)
{
  if (lista==NULL)
    return false;
  else
  { if (lista->dato%2==0)
    {  if (lista->sig==NULL)
         return true;
       else
         return secreto(lista->sig);
    }
    else
      return false;
  }
}
```

```
bool oculto (pnodo lista1, pnodo lista2)
{ if (lista1==NULL)
  { if (lista1==lista2)
       return true;
    else
       return false;
  }
  else
  { if (lista1->dato==lista2->dato)
      oculto(lista1->sig,lista2->sig);
    else
      return false;
  }
}
```

```
pnodo misterio (pnodo &lista)
{ pnodo p;
  if (lista==NULL)
      p=NULL;
  else
    { if (lista->sig==NULL)
       { p=lista;
         lista=NULL;
       }
      else
        p=misterio(lista->sig);
   }
   return p;
}
```

```
int enigma (pnodo lista)
{ int n;
  if (lista==NULL)
    return -1;
  else
   if (lista->sig==NULL)
          return lista->dato;
   else
    { n=enigma(lista->sig);
      if (n>lista->dato)
        n=lista->dato;
      return n;
    }
}
```

```
pnodo oculto (pnodo &lista, int n)
{ pnodo p=NULL;
  if (lista!=NULL)
  { if (lista->dato!=n)
      p=oculto(lista->sig,n);
    else
     { p=lista;
       lista=lista->sig;
       p->sig=NULL;
     }
  }
   return p;
}
```

9) El bibliotecario de la FI desea registrar información acerca de los libros disponibles para préstamos. Considerando esto, por cada libro debe guardarse: código de libro, título, autor, isbn, info editorial (cantidad de páginas, año de publicación, editorial), total de ejemplares y ejemplares disponibles para préstamo. Para ello, se solicita:
   a) Consigne la declaración de tipos y variables de la estructura que represente la situación planteada.
   b) Diseñe los procedimientos/funciones que listen los libros (título, autor y año de publicación) que no tengan ejemplares disponibles. Indique cuántos libros en esta situación se encontraron.
   c) Diseñe los procedimientos/funciones que muestre el libro (título, isbn y editorial) con la mayor cantidad de ejemplares.

10) El secretario académico de la FI desea registrar información acerca de las carreras que se dictan en la institución y de los alumnos que cursan sus estudios en ésta. Para ello, solicita la implementación de un sistema informático que almacene la siguiente información:
   - Carreras: código de carrera, nombre de la carrera, duración, total de materias y título emitido.
   - Alumnos: libreta universitaria, apellido, nombre, dni, fecha de nacimiento (día, mes, año), fecha de ingreso (día, mes, año), domicilio, cantidad de materias aprobadas y código de carrera en la que está inscripto.

   Considerando esto se solicita:
   a) Definir las estructura de datos que permitan representar las carreras y alumnos de la facultad.
   b) Diseñar los procedimientos/función que permitan listar, por carrera, los alumnos (apellido, nombre, año de ingreso) inscriptos, indicando cuántos cursan cada una de ellas.
   c) Diseñar los procedimientos/función que muestre los alumnos (libreta, apellido, nombre y título al que aspira el estudiante) a los que les falten 10 o menos materias por aprobar.